

As these examples illustrate, relative cache penalties increase as a processor becomes faster. Furthermore, if a processor improves both clock rate and CPI, it suffers a double hit:

1. The lower the CPI, the more pronounced the impact of stall cycles.
2. The main memory system is unlikely to improve as fast as processor cycle time, primarily because the performance of the underlying DRAM is not getting much faster. When calculating CPI, the cache miss penalty is measured in processor clock cycles needed for a miss. Therefore, if the main memories of two processors have the same absolute access times, a higher processor clock rate leads to a larger miss penalty.

Thus, the importance of cache performance for processors with low CPI and high clock rates is greater, and consequently the danger of neglecting cache behavior in assessing the performance of such processors is greater. As we will see in Section 7.6, the use of fast, pipelined processors in desktop PCs and workstations has led to the use of sophisticated cache systems even in computers selling for less than a \$1000.

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. With pipelines deeper than five stages, 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.

The next subsection discusses alternative cache organizations that decrease miss rate but may sometimes increase hit time; additional examples appear in Fallacies and Pitfalls (Section 7.7).

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. There is actually a whole range of schemes for placing blocks. At one extreme is direct mapped, where a block can be placed in exactly one location.

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 (at least two) 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.

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

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

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. For example, Figure 7.13 shows where block 12 may be placed in a cache with eight blocks total, according to the block placement policy for direct-mapped, two-way set-associative, and fully associative caches.

We can think of every block placement strategy as a variation on set associativity. Figure 7.14 shows the possible associativity structures for an eight-block cache. A direct-mapped cache is simply a one-way set-associative cache: each

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.

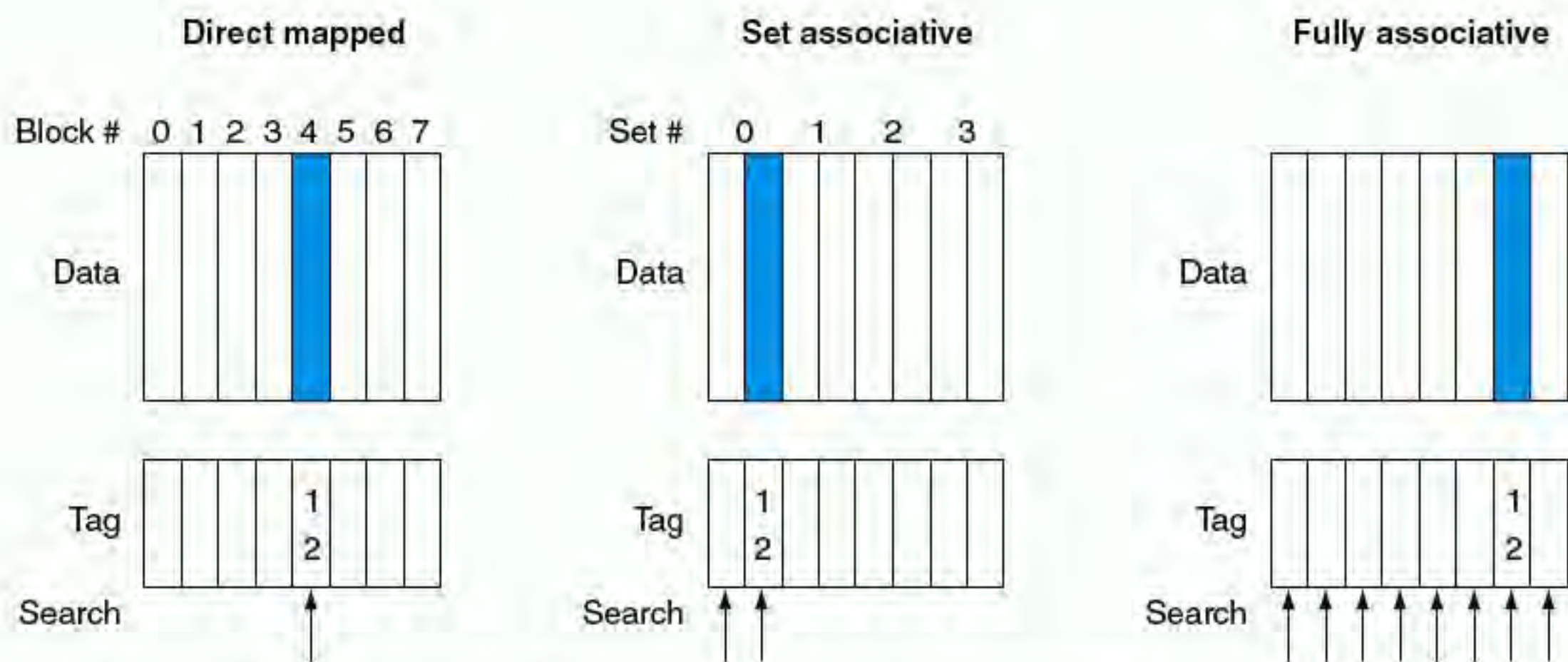


FIGURE 7.13 The location of a memory block whose address is 12 in a cache with 8 blocks varies for direct-mapped, set-associative, and fully associative placement. In direct-mapped placement, there is only one cache block where memory block 12 can be found, and that block is given by $(12 \bmod 8) = 4$. 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.

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 an increase in the hit time.

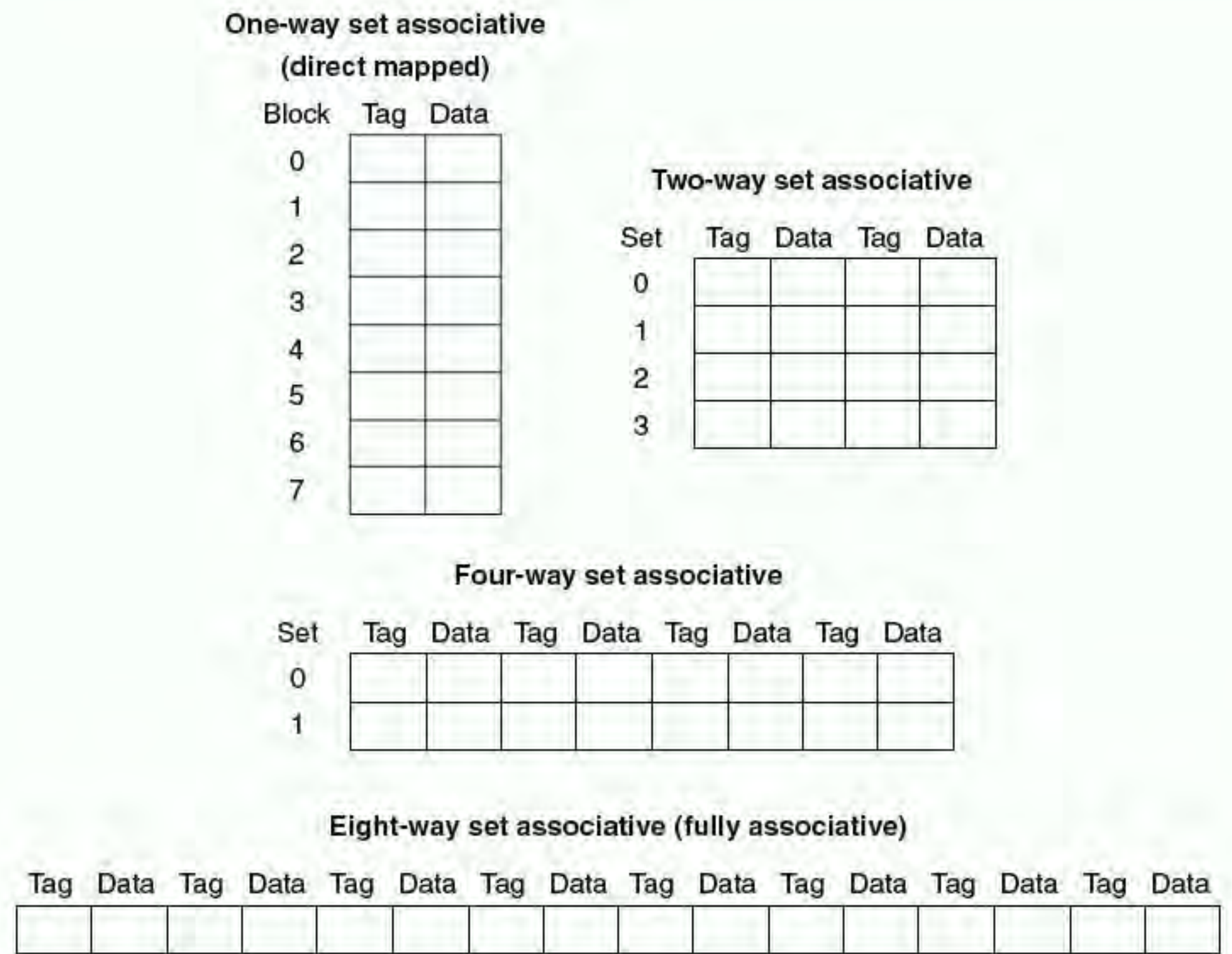


FIGURE 7.14 An eight-block cache configured as direct mapped, two-way set associative, four-way set associative, and fully associative. The total size of the cache in blocks is equal to the number of sets times the associativity. Thus, for a fixed cache size, increasing the associativity decreases the 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, 8.

EXAMPLE

ANSWER

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

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

Now we can fill in the cache contents after each reference, using a blank entry to mean that the block is invalid, colored text to show a new entry added to the cache for the associate reference, and a 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 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 three caches would have the same number of misses. This change in miss rate shows us 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 7.15 shows the improvement for the SPEC2000 benchmarks for a 64 KB 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 7.15 The data cache miss rates for an organization like the Intrinsity FastMATH processor for SPEC2000 benchmarks with associativity varying from one-way to eight-way. These results for 10 SPEC2000 programs are from Hennessy and Patterson [2003].

Locating a Block in the Cache

Now, let’s consider the task of finding a block in a cache that is set associative. Just as in a direct-mapped cache, each block in a set-associative cache includes an address tag that gives the block address. The tag of every cache block within the appropriate set is checked to see if it matches the block address from the processor. Figure 7.16 shows how the address is decomposed. 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 two in associativity doubles the number of blocks per set and halves the number of sets. Accordingly, each factor-of-two 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, such as in Figure 7.7 on page 478, only a single comparator is needed, because the entry can be in only one block, and we access the cache simply by indexing. Figure 7.17 shows that in a four-way set-associative cache, four comparators are needed, together with a 4-to-1 multiplexor to choose

Tag	Index	Block Offset
-----	-------	--------------

FIGURE 7.16 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.

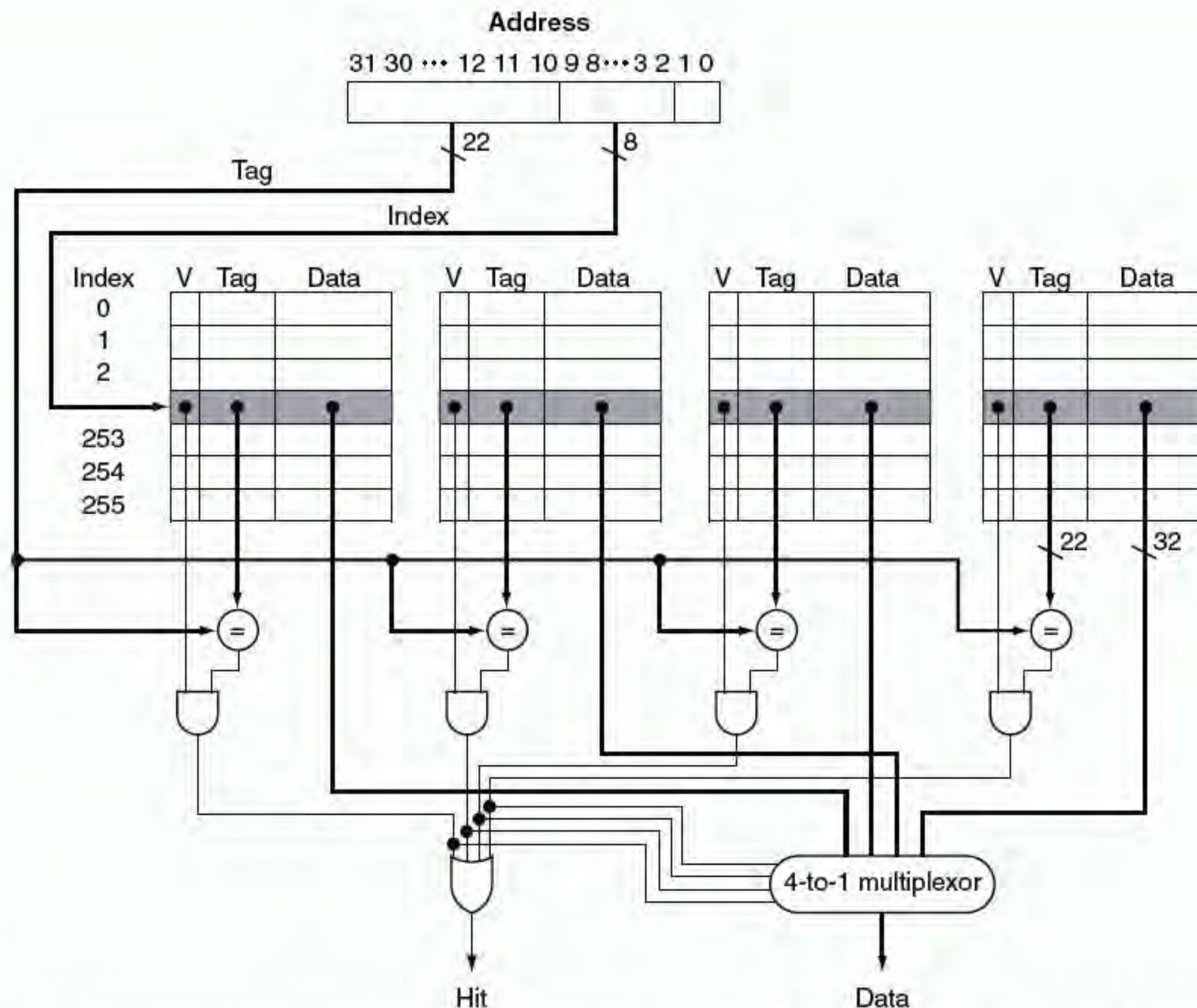


FIGURE 7.17 The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor.

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.

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.

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.

EXAMPLE**ANSWER****Size of Tags versus Set Associativity**

Increasing associativity requires more comparators, and more tag bits per cache block. Assuming a cache of 4K blocks, a four-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.

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 $\log_2(4K) = 12$; hence, the total number of tag bits is $(28 - 12) \times 4K = 16 \times 4K = 64$ Kbits.

Each degree of associativity decreases the number of sets by a factor of two and thus decreases the number of bits used to index the cache by one and increases the number of bits in the tag by one. Thus, for a two-way set-associative cache, there are 2K sets, and the total number of tag bits is $(28 - 11) \times 2 \times 2K = 34 \times 2K = 68$ Kbits. For a four-way set-associative cache, the total number of sets is 1K, and the total number of tag bits is $(28 - 10) \times 4 \times 1K = 72 \times 1K = 72$ Kbits.

For a fully associative cache, there is only one set with 4K blocks, and the tag is 28 bits, leading to a total of $28 \times 4K \times 1 = 112K$ tag bits.

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. 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 7.5, 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.

Reducing the Miss Penalty Using Multilevel Caches

All modern computers make use of caches. In most cases, these caches are implemented on the same die as the microprocessor that forms the processor. To further close the gap between the fast clock rates of modern processors and the relatively long time required to access DRAMs, many microprocessors support an additional level of caching. This second-level cache, which can be on the same chip or off-chip in a separate set of SRAMs, 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 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 secondary cache contains the data, a main memory access is required, and a larger miss penalty is incurred.

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

Performance of Multilevel Caches

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 5 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

$$\frac{100 \text{ ns}}{0.2 \frac{\text{ns}}{\text{clock cycle}}} = 500 \text{ clock cycles}$$

EXAMPLE

ANSWER

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,

$$\text{Total CPI} = 1.0 + \text{Memory-stall cycles per instruction} = 1.0 + 2\% \times 500 = 11.0$$

With two levels of cache, 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

$$\frac{5 \text{ ns}}{0.2 \frac{\text{ns}}{\text{clock cycle}}} = 25 \text{ clock cycles}$$

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:

$$\begin{aligned} \text{Total CPI} &= 1 + \text{Primary stalls per instruction} \\ &\quad + \text{Secondary stalls per instruction} \\ &= 1 + 2\% \times 25 + 0.5\% \times 500 = 1 + 0.5 + 2.5 = 4.0 \end{aligned}$$

Thus, the processor with the secondary cache is faster by

$$\frac{11.0}{4.0} = 2.8$$

Alternatively, we could have computed the stall cycles by summing the stall cycles of those references that hit in the secondary cache $((2\% - 0.5\%) \times 25 = 0.4)$ and 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 $(0.5\% \times (25 + 500) = 2.6)$. The sum, $1.0 + 0.4 + 2.6$, is again 4.0.

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, while allowing the secondary cache to focus on miss rate to reduce the penalty of long memory access times.

The interaction of the two caches permits such a focus. The miss penalty of the primary cache is significantly reduced by the presence of the secondary cache, allowing the primary to be smaller and have a higher miss rate. For the secondary cache, access time becomes less important with the presence of the primary cache, since the access time of the secondary cache affects the miss penalty of the primary cache, rather than directly affecting the primary cache hit time or the processor cycle time.

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 often uses a smaller block size, to go with the smaller cache size and reduced miss penalty. In comparison, the secondary cache will often be 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 often will use a larger block size than appropriate with a single-level cache

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

In Chapter 2, we saw that Quicksort had an algorithmic advantage over Bubble Sort that could not be overcome by language or compiler optimization. Figure 7.18(a) shows instructions executed by item searched for Radix Sort versus Quicksort. Indeed, for large arrays, Radix Sort has an algorithmic advantage over Quicksort in terms of number of operations. Figure 7.18(b) shows time per key instead of instructions executed. We see that the lines start on the same trajectory as Figure 7.18(a), but then the Radix Sort line diverges as the data to sort increases. What is going on? Figure 7.18(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 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.

Understanding Program Performance

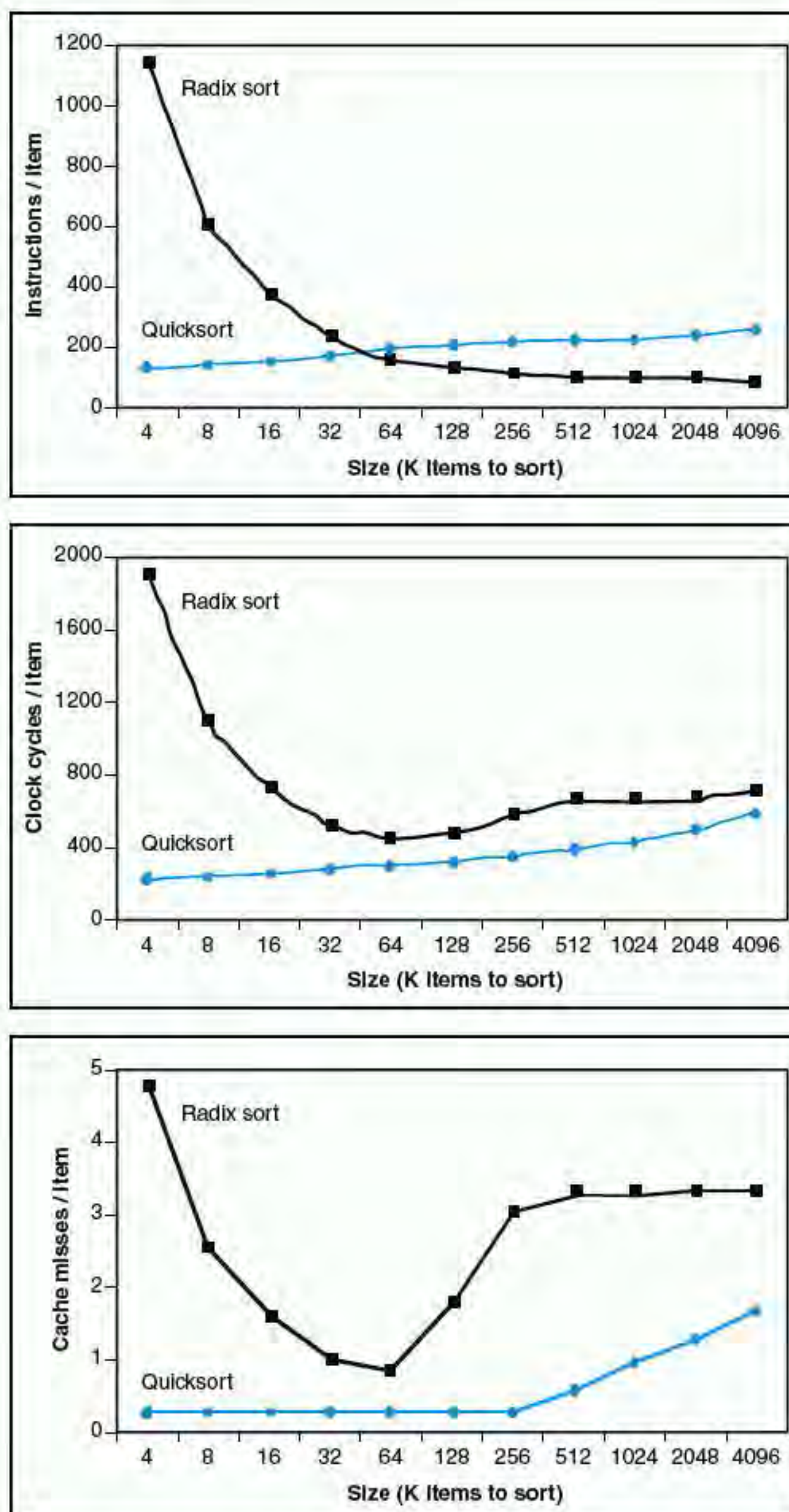


FIGURE 7.18 Comparing Quicksort and Radix Sort by (a) instructions executed per item sorted, (b) time per item sorted, and (c) cache misses per item sorted. This data is from a paper by LaMarca and Ladner [1996]. Although the numbers would change for newer computers, the idea still holds. Due to such results, new versions of Radix Sort have been invented that take memory hierarchy into account, to regain its algorithmic advantages (see Section 7.7). The basic idea of cache optimizations is to use all the data in a block repeatedly before it is replaced on a miss.

Elaboration: Multilevel caches create several complications. First, there are now several different types of misses and corresponding miss rates. In the example on page 499, 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. 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 page 499, 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.

Additional complications arise because the caches may have different block sizes to match the larger or smaller total size. Likewise, the associativity of the cache may change. On-chip caches are often built with associativity of four or higher, while off-chip caches rarely have associativity of greater than two. On chip L1 caches tend to have lower associativity than one chip L2 caches since fast hit time is more important for L1 caches. These changes in block size and associativity introduce complications in the modeling of the caches, which typically mean that all levels need to be simulated together to understand the behavior.

Elaboration: With out-of-order processors, performance is more complex, since they execute instructions during the miss penalty. Instead of instruction miss rate 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 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.

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.

Check Yourself

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

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

Summary

In this section, we focused on three topics: cache performance, using associativity to reduce miss rates, and the use of multilevel cache hierarchies to reduce miss penalties.

Since the total number of cycles spent on a program is the sum of the processor cycles and the memory-stall cycles, the memory system can have a significant effect on program execution time. In fact, as processors get faster (by lowering CPI or by increasing the clock rate or both), the relative effect of the memory-stall cycles increases, making good memory systems critical to achieving high performance. The number of memory-stall cycles depends on both the miss rate and the miss penalty. The challenge, as we will see in Section 7.5, 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. This search is usually implemented by having a comparator per cache block and searching the tags in parallel. The cost of the comparators makes 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.

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

secondary cache and its access time depend on a number of aspects of the implementation.

7.4 Virtual Memory

In the previous section, we saw how caches provided fast access to recently used portions of a program's code and data. Similarly, the main memory can act as a "cache" for the secondary storage, 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, and to remove the programming burdens of a small, limited amount of main memory. Four decades after its invention, it's the former reason that reigns today.

Consider a collection of programs running at once on a computer. The total memory required by all the programs may be much larger than the amount of main memory available on the computer, but only a fraction of this memory is actively being used at any point in time. Main memory need contain only the active portions of the many programs, 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. Of course, to allow multiple programs to share the same memory, we must be able to protect the programs from each other, ensuring that a program can only read and write the portions of main memory that have been assigned to it.

We cannot know which programs will share the memory with other programs when we compile them. In fact, the programs sharing the memory change dynamically while the programs are running. Because of this dynamic interaction, we would like to compile each program into its own *address space*—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 programs.

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

... 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.

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

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 7.19 shows the virtually addressed memory with pages mapped to main memory. This process is called *address mapping* or **address translation**. Today, the two memory hierarchy levels controlled by virtual memory are DRAMs

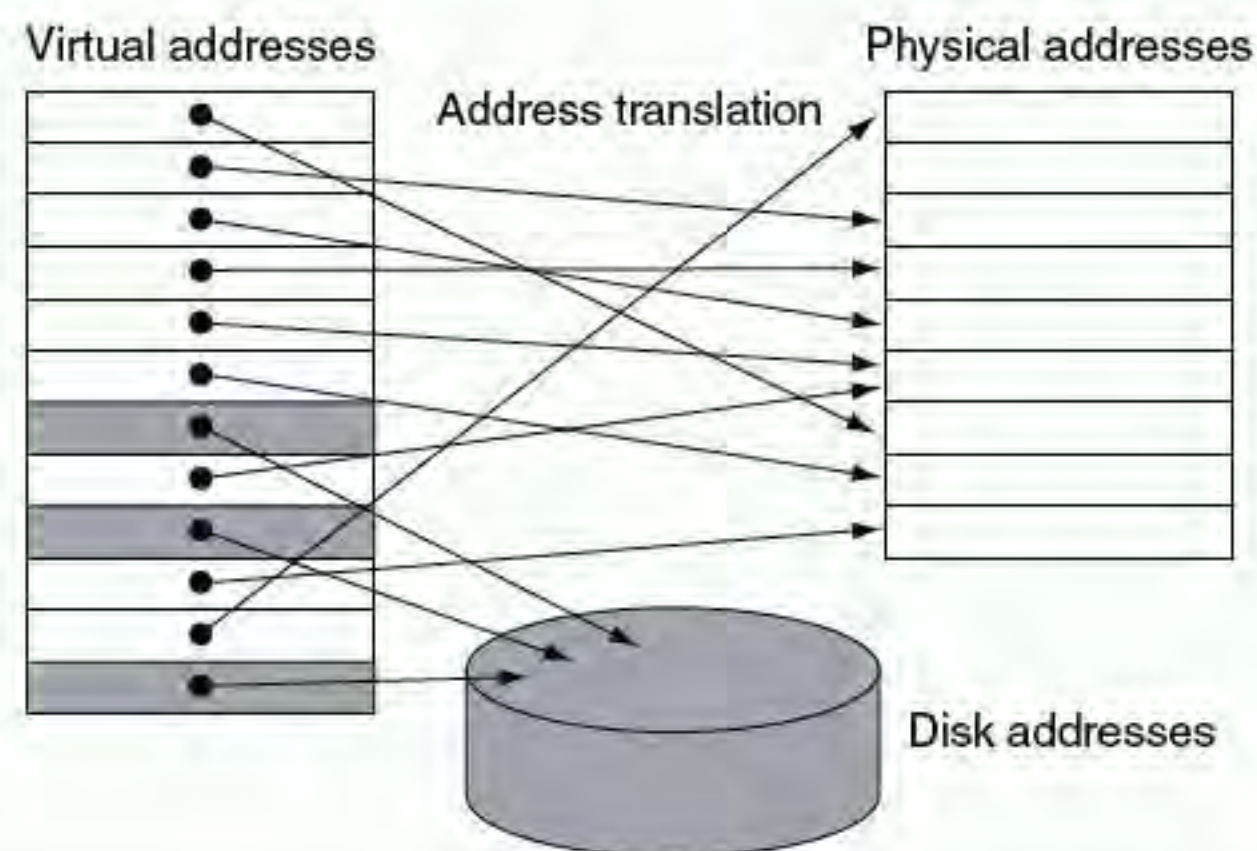


FIGURE 7.19 In virtual memory, blocks of memory (called *pages*) are mapped from one set of addresses (called *virtual addresses*) to another set (called *physical addresses*). The processor generates virtual addresses while the memory is accessed using physical addresses. Both the virtual memory and the physical memory are broken into pages, so that a virtual page is really 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, residing instead 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.

and magnetic disks (see Chapter 1, pages 5, 13 and 23). 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. Formerly, relocation problems required special hardware and special support in the operating system; today, virtual memory also provides this function.

In virtual memory, the address is broken into a *virtual page number* and a *page offset*. Figure 7.20 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

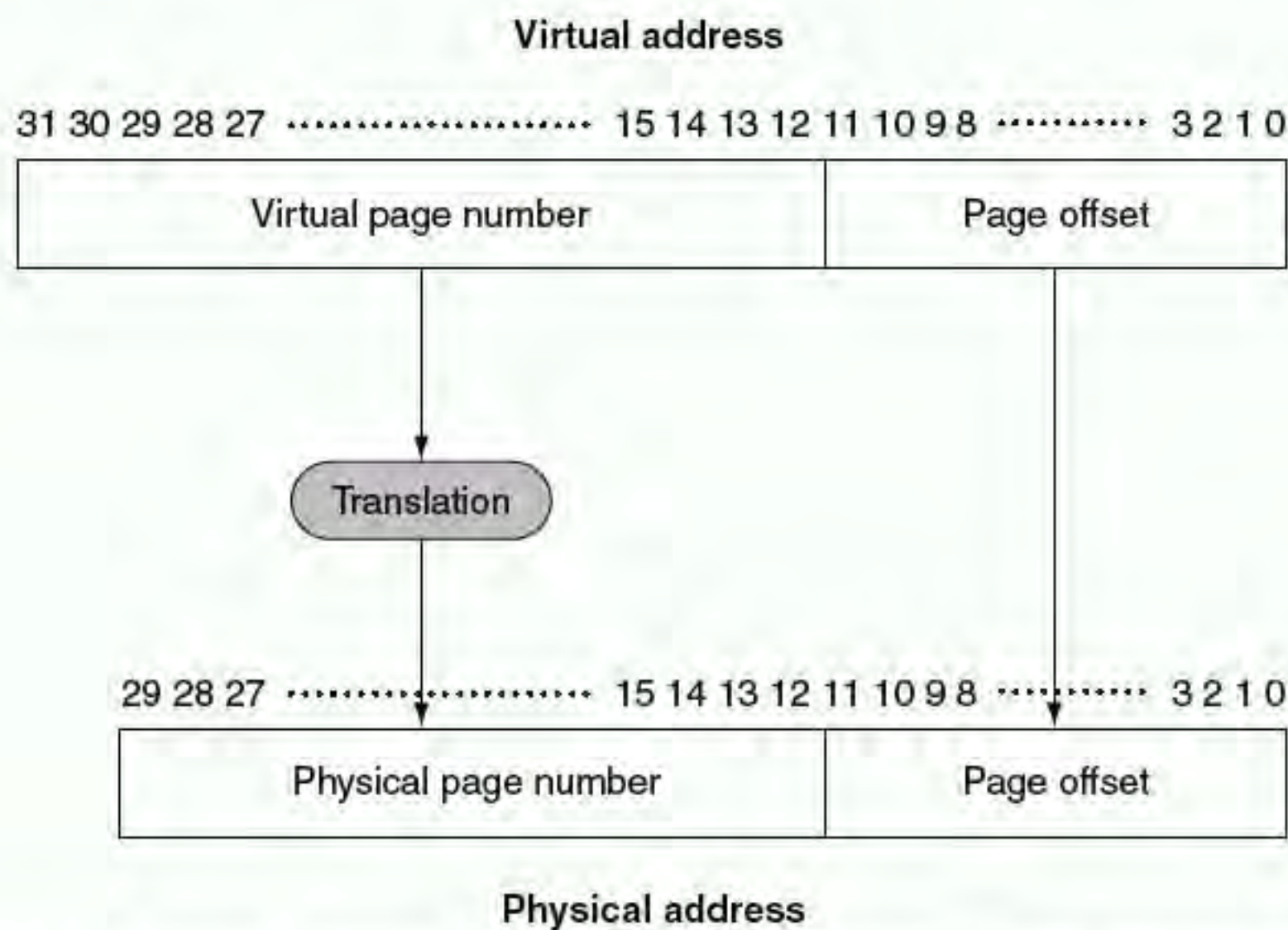


FIGURE 7.20 Mapping from a virtual to a physical address. The page size is $2^{12} = 4$ KB. The number of physical pages allowed in memory is 2^{18} , since the physical page number has 18 bits in it. Thus, main memory can have at most 1 GB, while the virtual address space is 4 GB.

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.

Many design choices in virtual memory systems are motivated by the high cost of a miss, which in virtual memory is traditionally called a *page fault*. A page fault will take millions of clock cycles to process. (The table on page 469 shows that main memory is about 100,000 times faster 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 KB to 16 KB are typical today. New desktop and server systems are being developed to support 32 KB and 64 KB pages, but new embedded systems are going in the other direction, to 1 KB 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: 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 can benefit, but a collection of programs running at the same time can benefit from not having to be swapped to memory or by running on parallel processors. Given that Moore's law applies to DRAM, 32-bit processors are already problematic for servers and soon for desktops.

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.

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 register 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.

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, which 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 7.3, 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 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.

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

Hardware Software Interface

The page table, together with the program counter and the registers, specifies the *state* of a program. If we want to allow another program to use the processor, we must save this state. Later, after restoring this state, the program 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 7.21 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.

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 discuss later in this section. Once the operating system gets control, it must find the page in the next level of the hierarchy (usually 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.

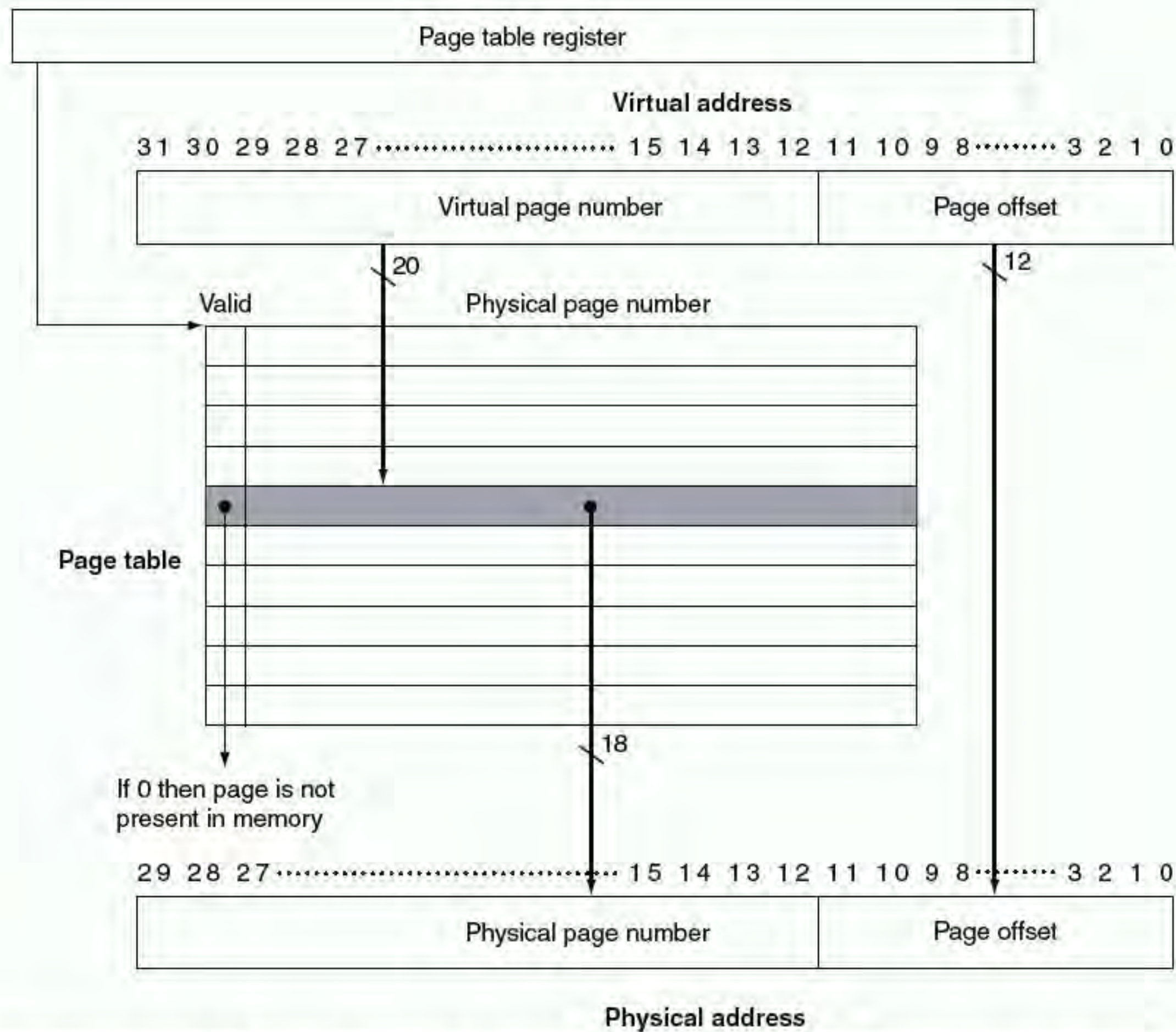


FIGURE 7.21 The page table is indexed with the virtual page number to obtain the corresponding portion of the physical address. The starting address of the page table is given by the page table pointer. In this figure, the page size is 2^{12} bytes, or 4 KB. The virtual address space is 2^{32} bytes, or 4 GB, and the physical address space is 2^{30} bytes, which allows main memory of up to 1 GB. 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.

Because we do not know ahead of time when a page in memory will be chosen to be replaced, the operating system usually creates the space on disk for all the pages of a process when it creates the process. This disk 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 7.22

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

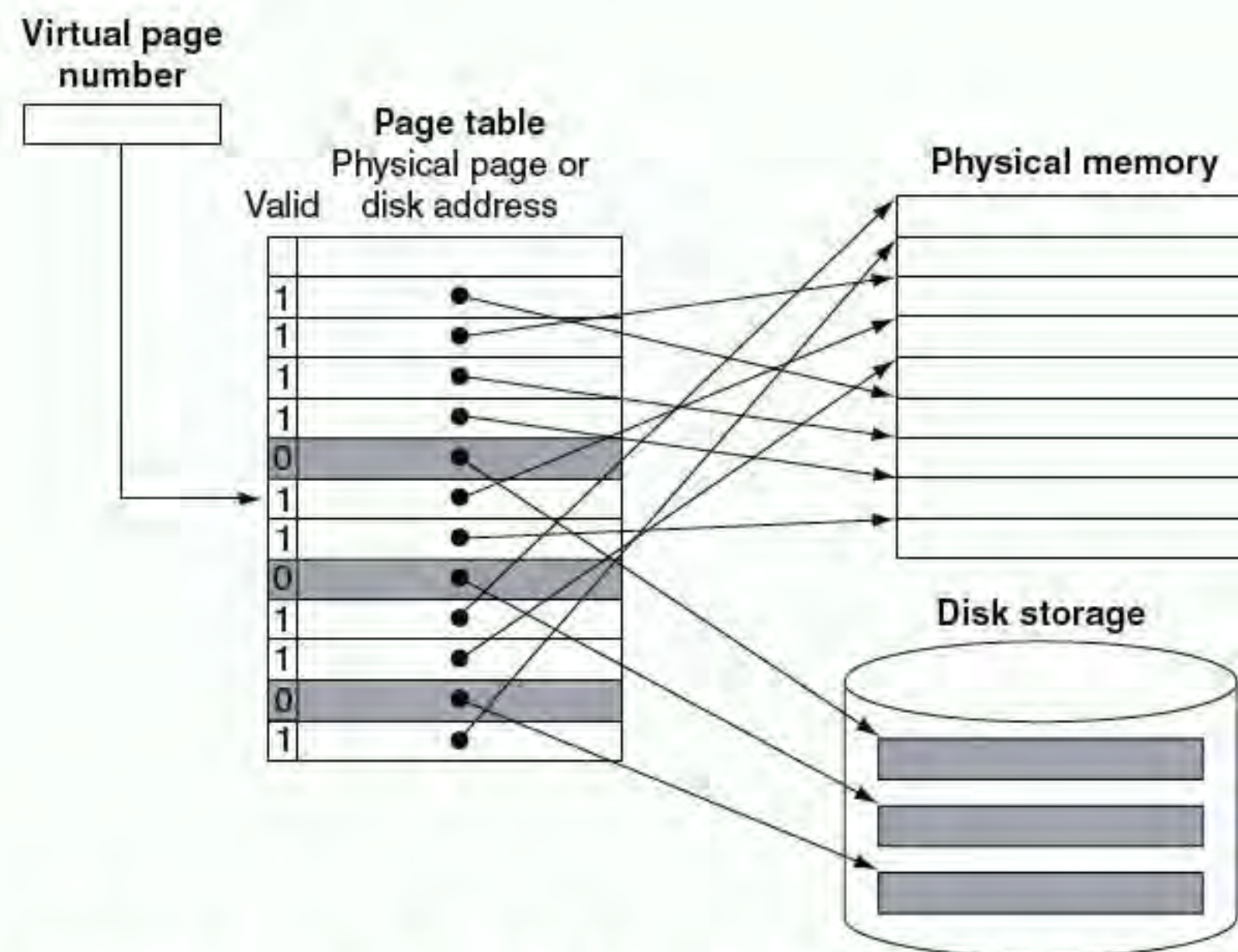


FIGURE 7.22 The page table maps each page in virtual memory to either a page in main memory or a page stored on disk, which is the next level in the hierarchy. The virtual page number is used to index the page table. If the valid bit is on, the page table supplies the physical page number (i.e., the 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 identical in size.

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 7.3. The operating system searches for the least recently used page, making the assumption 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.

For example, suppose the page references (in order) were 10, 12, 9, 7, 11, 10, and then we referenced page 8, which was not present in memory. The LRU page is 12; in LRU replacement, we would replace page 12 in main memory with page 8. If the next reference also generated a page fault, we would replace page 9, since it would then be the LRU among the pages present in memory.

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 **use bit** or **reference 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 KB pages, and 4 bytes per page table entry, we can compute the total page table size:

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

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

That is, we would need to use 4 MB of memory for each program in execution at any time. On a computer with tens to hundreds of active programs and a fixed-size page table, most or all of the memory would be tied up in page tables!

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 segment is specified by the high-order address bit, 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 Elaboration on page 514, 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 data structure 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 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 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) 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. This copying back to the lower level in the hierarchy is the source of the other name for this technique of handling writes, namely, *copy back*.

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.

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.

translation-lookaside buffer (TLB) A cache that keeps track of recently used address mappings to avoid an access to the page table.

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 7.23 shows that each tag entry in the TLB holds a portion of the virtual page number, and each data entry of the TLB holds a physical page number. Because we will no longer access the page table on every reference, instead accessing the TLB, the TLB will need to include other bits, such as the dirty and the reference bit.

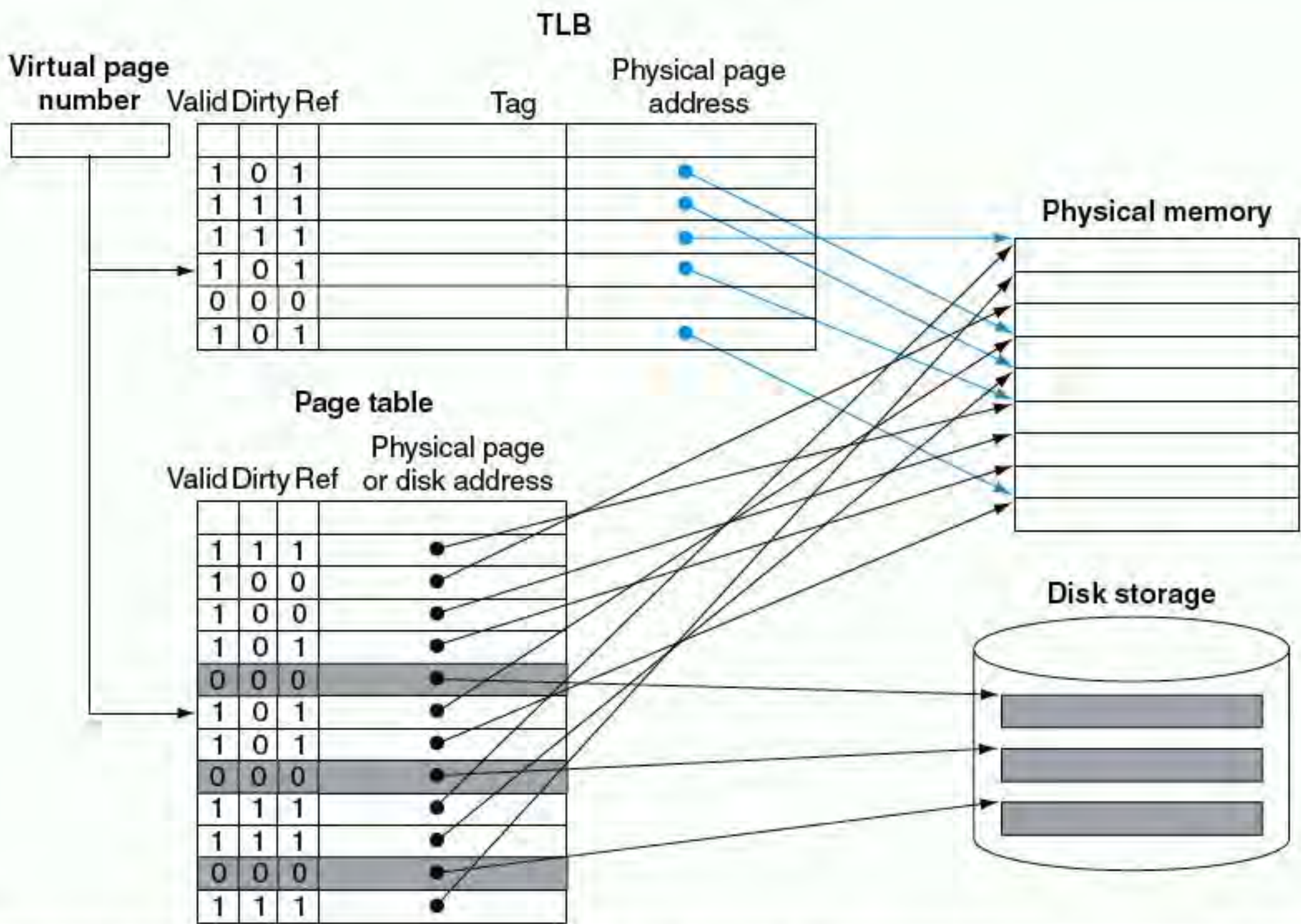


FIGURE 7.23 The TLB acts as a cache on the page table for the entries that map to physical pages only. The TLB contains a subset of the virtual-to-physical page mappings that are in the page table. The TLB mappings are shown in color. Because the TLB is a cache, it must have a tag field. If there is 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, it is *not* a cache,

On every reference, we look up the virtual page number in the TLB. If we get a hit, the physical page number is used to form the address, and the corresponding reference bit is turned on. If the processor is performing a write, the dirty bit is also turned on. If a miss in the TLB occurs, we must determine whether it is a page fault or merely a TLB miss. If the page exists in memory, then the TLB miss indicates only that the translation is missing. In such cases, the processor can handle the TLB miss by loading the translation from the 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 7.5.

The Intrinsity FastMATH TLB

To see these ideas in a real processor, let's take a closer look at the TLB of the Intrinsity FastMATH. The memory system uses 4 KB pages and a 32-bit address space; thus, the virtual page number is 20 bits long, as in the top of Figure 7.24. The physical address is the same size as the virtual address. The TLB contains 16 entries, is fully associative, and 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.

Figure 7.24 shows the TLB and one of the caches, while Figure 7.25 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 534) 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 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 plays an important role in maintaining 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 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

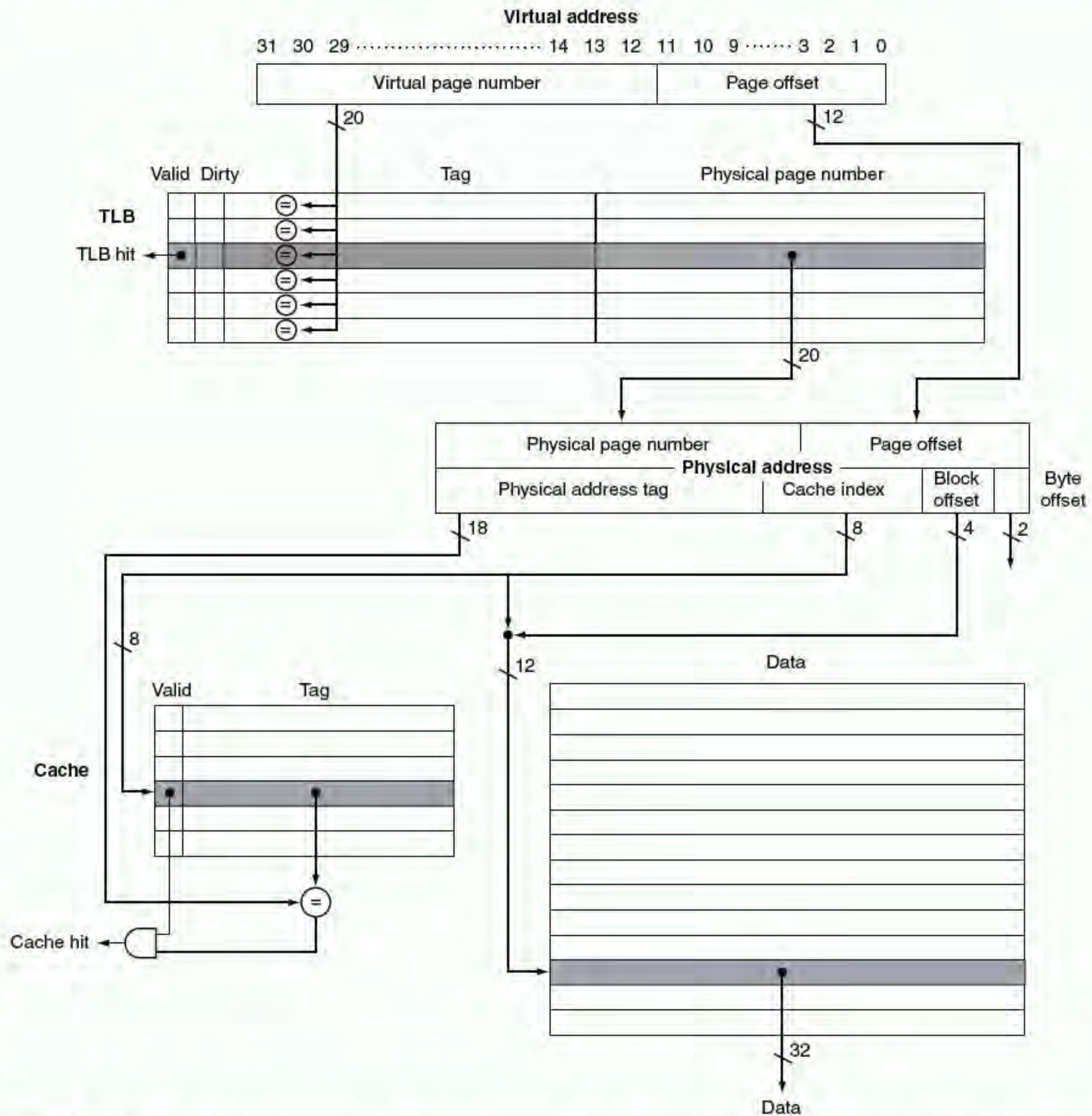


FIGURE 7.24 The TLB and cache implement the process of going from a virtual address to a data item in the Intrinsity Fast-MATH. This figure shows the organization of the TLB and the data cache assuming a 4 KB page size. This diagram focuses on a read; Figure 7.25 describes how to handle writes. Note that unlike Figure 7.9 on page 486, 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. 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. (The Intrinsity actually has a 16 KB page size; the Elaboration on page 528 explains how it works.)

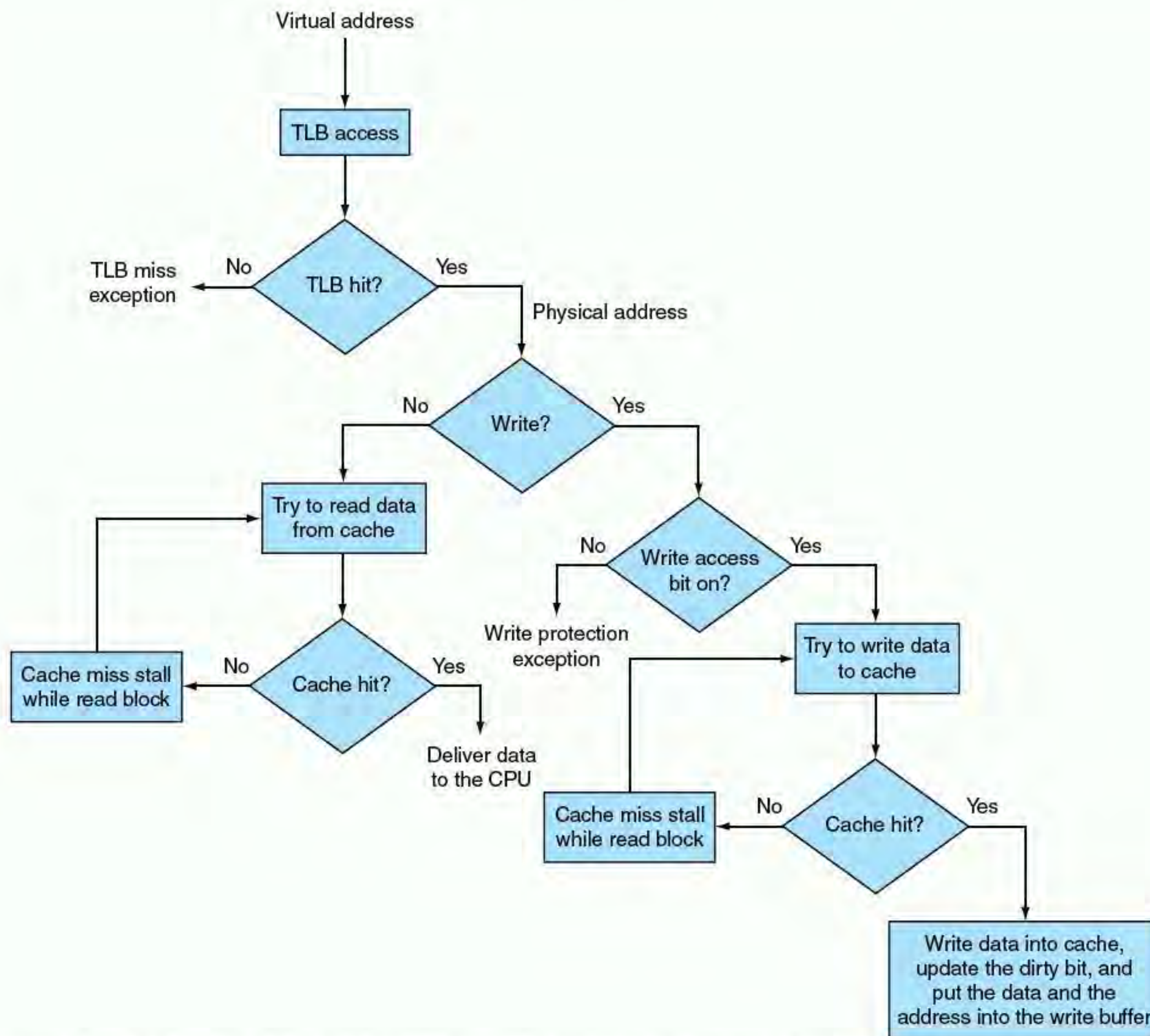


FIGURE 7.25 Processing a read or a write through in the Intrinsic FastMATH TLB and cache. If the TLB generates a hit, the cache can be accessed with the resulting physical address. For a read, the cache generates a hit or miss and supplies the data or causes a stall while the data is brought from memory. 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.

the memory hierarchy: the TLB, the page table, and the cache. The following example illustrates these interactions in more detail..

Overall Operation of a Memory Hierarchy

In a memory hierarchy like that of Figure 7.24 that 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.

Figure 7.26 shows the possible circumstances and whether they can arise in practice or not.

EXAMPLE

ANSWER

Elaboration: Figure 7.26 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

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 7.26 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 the same object is accessed by two addresses; 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.

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.

When the cache is accessed with a virtual address and pages are shared between programs (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.

Figure 7.24 assumed a 4 KB page size, but it's really 16 KB. The Intrinsic FastMATH uses such a memory system organization. The cache and TLB are still accessed in parallel, so the upper 2 bits of the cache index must be virtual. Hence, up to four cache entries could be aliased to the same physical memory address. As the L2 cache on the chip includes all entries in the L1 caches, on a L1 miss it checks the other three possible cache locations in the L2 cache for aliases. If it finds one, it flushes it from the caches to prevent aliases from occurring.

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 untranslated), 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. The L1 data cache of the Pentium 4 is an example as would the Intrinsic if the page size was 4 KB. To pull off this trick, there must be careful coordination between the minimum page size, the cache size, and associativity.

Elaboration: The FastMATH TLB is a bit more complicated than in Figure 7.24. MIPS includes two physical page mappings per virtual page number, thereby mapping an even-odd pair of virtual page numbers into two physical page numbers. Hence, the tag is 1 bit narrower since each entry corresponds to two pages. The least significant bit of the virtual page number selects between the two physical pages. There are separate book-keeping bits for each physical page. This optimization doubles the amount of memory mapped per TLB entry. As the Elaboration on page 530 explains, the tag field actually includes an 8-bit address space ID field to reduce the cost of context switches. To support the variable page sizes mentioned on page 537, there is also a 32-bit mask field that determines the dividing line between the virtual page address and the page offset.

Implementing Protection with Virtual Memory

One of the most important functions for virtual memory 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. For example, if the program that maintains student grades were running on a computer at the same time as the programs of the students in the first programming course, we wouldn't want the errant program of a beginner to write over someone's grades. 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.

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.

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.

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!

Hardware Software Interface

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

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

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. Yet, 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, handling TLB misses and page faults are more complex. A TLB miss occurs when no entry in the TLB matches a virtual address. 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.

How do we know which of these two circumstances has occurred? When we process the TLB miss, we will look for a page table entry to bring into the TLB. If the matching page table entry has a valid bit that is turned off, then the corresponding page is not in memory and we have a page fault, rather than just a TLB miss. If the valid bit is on, we can simply retrieve the desired entry.

A TLB miss can be handled in software or hardware because it will require only a short sequence of operations to copy a valid page table entry from memory into the TLB. MIPS traditionally handles a TLB miss in software. It brings in the page table entry from memory and then reexecutes the instruction that caused the TLB miss. Upon reexecuting 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 Chapters 5 and 6, 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.

Register	CPO 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 7.27 MIPS control registers. These are considered to be in coprocessor 0, and hence are read using `mfc0` and written using `mtc0`.

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.

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 and Cause register. EPC and Cause are two of the special control registers that help with exceptions, TLB misses, and page faults; Figure 7.27 shows the rest. The operating system can then reenale 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.

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.

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 reexecutes the instruction that faulted, accesses the requested page successfully, and continues execution.

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

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

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

For processors with much more complex instructions that may 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, some 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.

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 0000hex`, the location of the TLB miss **handler**. To find the physical address for the missing page, the TLB miss routine indexes

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

handler Name of a software routine invoked to "handle" an exception or interrupt.

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 0180hex`, a different address than 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.

Figure 7.28 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.

Save state				
Save GPR	addi	\$k1,\$sp, -XCPSIZE	# save space on stack for state	
	sw	\$sp, XCT_SP(\$k1)	# save \$sp on stack	
	sw	\$v0, XCT_V0(\$k1)	# save \$v0 on stack	
	...		# save \$v1, \$a1, \$s1, \$t1, ... on stack	
	sw	\$ra, XCT_RA(\$k1)	# save \$ra on stack	
Save Hi, Lo	mfhi	\$v0	# copy Hi	
	mflo	\$v1	# copy Lo	
	sw	\$v0, XCT_HI(\$k1)	# save Hi value on stack	
	sw	\$v1, XCT_LO(\$k1)	# save Lo value on stack	
Save Exception Registers	mfc0	\$a0, \$cr	# copy cause register	
	sw	\$a0, XCT_CR(\$k1)	# save \$cr value on stack	
	...		# save \$v1, ...	
	mfc0	\$a3, \$sr	# copy Status Register	
	sw	\$a3, XCT_SR(\$k1)	# save \$sr on stack	
Set sp	move	\$sp, \$k1	# sp = sp - XCPSIZE	
Enable nested exceptions				
	andi	\$v0, \$a3, MASK1	# \$v0 = \$sr & MASK1, enable exceptions	
	mtc0	\$v0, \$sr	# \$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	# arg1 = pointer to exception stack	
	jal	xcpt_deliver	# call C code to handle exception	
Restoring state				
Restore most GPR, Hi, Lo	move	\$at, \$sp	# temporary value of \$sp	
	lw	\$ra, XCT_RA(\$at)	# restore \$ra from stack	
	...		# restore \$t0, ..., \$a1	
	lw	\$a0, XCT_A0(\$k1)	# restore \$a0 from stack	
Restore Status Register	lw	\$v0, XCT_SR(\$at)	# load old \$sr from stack	
	li	\$v1, MASK2	# mask to disable exceptions	
	and	\$v0, \$v0, \$v1	# \$v0 = \$sr & MASK2, disable exceptions	
	mtc0	\$v0, \$sr	# set Status Register	
Exception return				
Restore \$sp and rest of GPR used as temporary registers	lw	\$sp, XCT_SP(\$at)	# restore \$sp from stack	
	lw	\$v0, XCT_V0(\$at)	# restore \$v0 from stack	
	lw	\$v1, XCT_V1(\$at)	# restore \$v1 from stack	
	lw	\$k1, XCT_EPC(\$at)	# copy old \$epc from stack	
	lw	\$at, XCT_AT(\$at)	# restore \$at from stack	
Restore ERC and return	mtc0	\$k1, \$epc	# restore \$epc	
	eret	\$ra	# return to interrupted instruction	

FIGURE 7.28 MIPS code to save and restore state on an 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 exception entry point code and the exception stack in unmapped memory. MIPS hardware translates virtual addresses $8000\ 0000_{\text{hex}}$ to $\text{BFFF FFFF}_{\text{hex}}$ 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 7.28 shows the MIPS-32 exception return sequence. MIPS-I uses `rfe` and `jr` instead of `eret`.

Summary

Virtual memory is the name for the level of memory hierarchy that manages caching between the main memory and disk. Virtual memory allows a single program to expand its address space beyond the limits of main memory. More importantly, in recent computer systems virtual memory supports sharing of the main memory among multiple, simultaneously active processes, which together require far more total physical main memory than exists. To support sharing, virtual memory also provides mechanisms for memory protection.

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. Blocks, called 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 disk 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 back to disk.

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. To ensure 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 have too much overhead and 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 disk and 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 reexamine 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 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 megabyte directly: $64 \times 4 \text{ KB} = 0.25 \text{ MB}$. 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 KB page, MIPS hardware supports 16 KB, 64 KB, 256 KB, 1 MB, 4 MB, 16 MB, 64 MB, and 256 MB pages. Hence, if a program uses large page sizes, it can access more memory directly without TLB misses.

The practical challenge is getting the operating system to allow programs to select these larger page sizes. Once again, the more complex solution to reducing TLB misses is to reexamine the algorithm and data structures to reduce the working set of pages.

Understanding Program Performance

Check Yourself

Match the memory hierarchy element on the left with the closest phrase on the right:

1. L1 cache

2. L2 cache

3. Main memory

4. TLB
- a. A cache for a cache

b. A cache for disks

c. A cache for a main memory

d. A cache for page table entries

7.5

A Common Framework for Memory Hierarchies

By now, you’ve recognized that the different types of memory hierarchies share 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 7.29 shows how some of the quantitative characteristics of memory hierarchies can differ. In the rest of this section, we will discuss the common operational aspects of 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.

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 cache Associativity	Associativity (typically 2–16)
Fully associative	1	Number of blocks in the cache

The advantage of increasing the degree of associativity is that it usually decreases the miss rate. The improvement in miss rate comes from reducing misses that compete for the same location. We will examine these in more detail shortly. First, let’s look at how much improvement is gained. Figure 7.30 shows the data for a workload consisting of the SPEC2000 benchmarks with caches of 4

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	4000–250,000	16,000–250,000	16–512
Total size in kilobytes	16–64	500–8000	250,000–1,000,000,000	0.25–16
Block size in bytes	32–64	32–128	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 7.29 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 2004. 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.

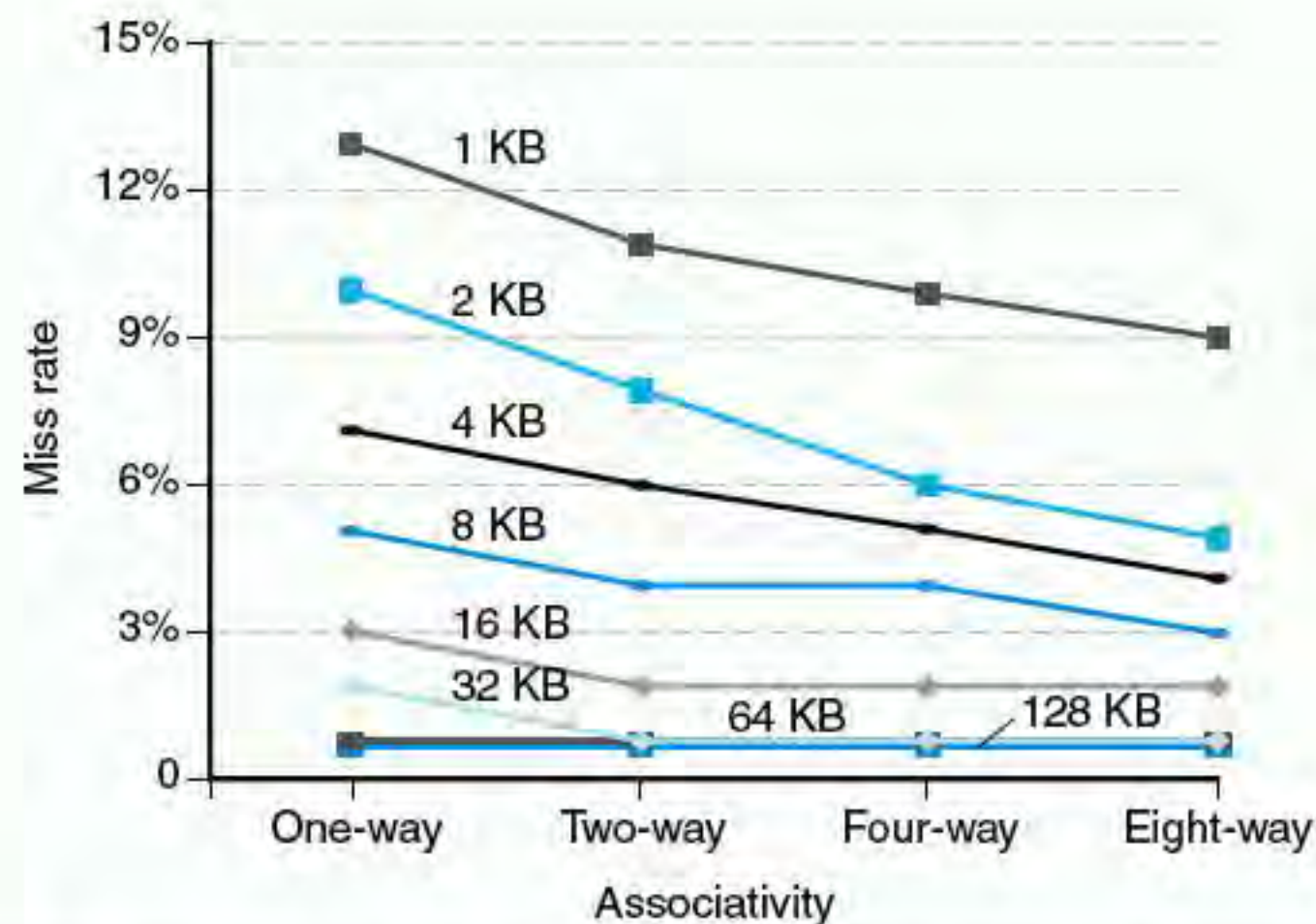


FIGURE 7.30 The data cache miss rates for each of eight cache sizes improve as the associativity increases. While the benefit of going from one-way (direct-mapped) to two-way set associative is significant, the benefits of further associativity are smaller (e.g., 1%–10% going from two-way to four-way 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 7.15 explains how this data was collected.

KB to 512 KB, varying from direct mapped to eight-way set associative. The largest gains are obtained in going from direct mapped to two-way set associative, which yields between a 20% and 30% reduction in the miss rate. As cache sizes grow, the relative improvement from associativity increases only slightly; since the overall miss rate of a larger cache is lower, the opportunity for improving the miss rate decreases and the absolute improvement in the miss rate from associativity

shrinks significantly. The potential disadvantages of associativity, as we mentioned earlier, are increased cost and slower access time.

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 four 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.
4. The large page size means the page table size overhead is relatively small. (The use of a separate lookup table, like a page table for virtual memory, is not practical for a cache because the table would be much larger than a page table and could not be accessed quickly.)

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.

We have already mentioned 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 of a 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 is 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 to the block in the lower level of the memory hierarchy (main memory for a cache). The caches in Section 7.2 used this scheme.
- *Write-back* (also called *copy-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 7.4.

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 in a high-speed system, a write-through cache will need to use a write buffer.

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 (disk). As processors continue to increase in performance at a faster rate than DRAM-based main memory, the rate at which writes are generated by a processor will exceed the rate at which the memory system can process them, even allowing for physically and logically wider memories. Consequently, more and more caches are using a write-back strategy.

While caches, TLBs, and virtual memory may initially look very different, they rely on the same two principles of locality and can be understood by looking at how they deal with 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.

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

In this section, 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**.

The BIG Picture

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 block 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.

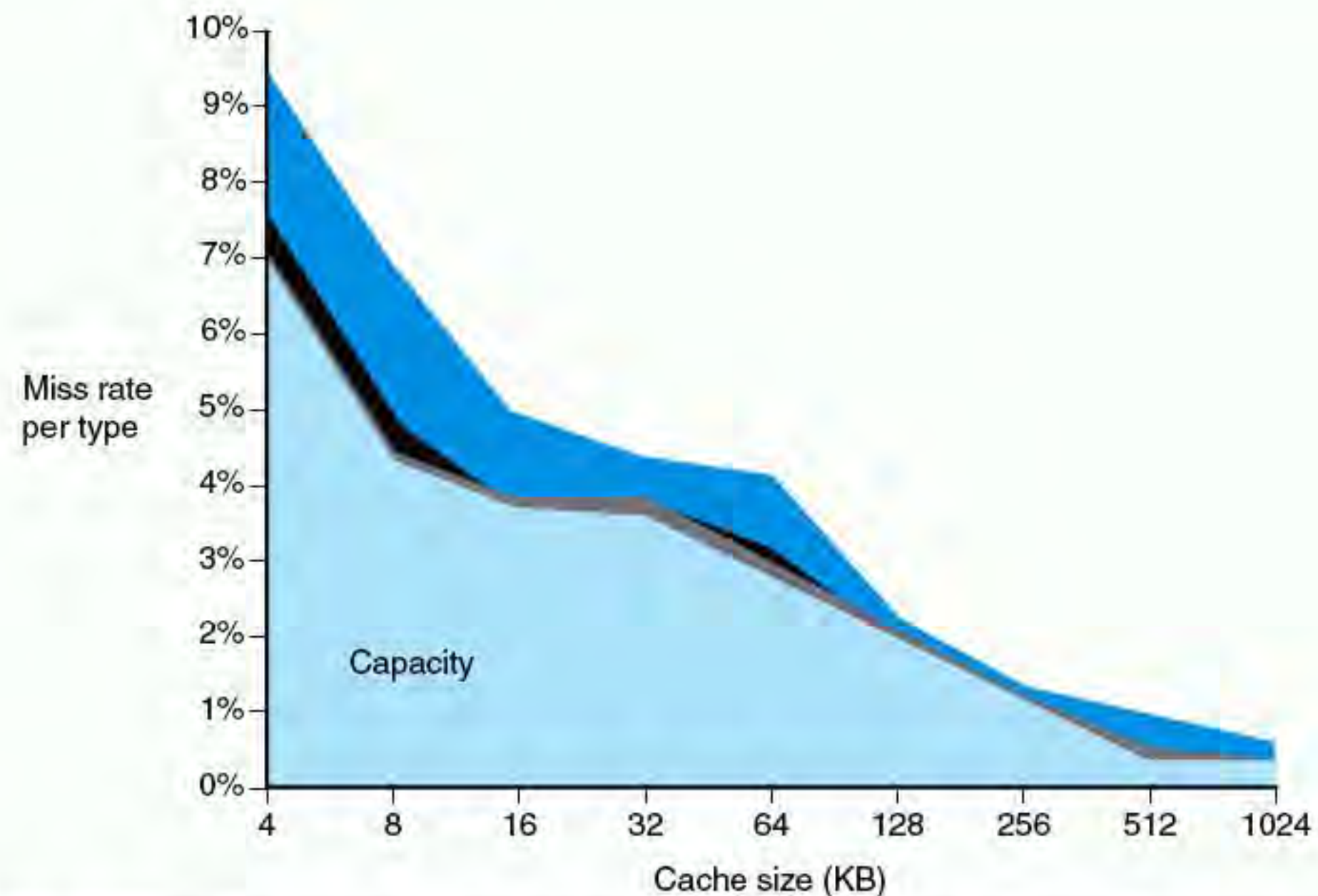


FIGURE 7.31 The miss rate can be broken into three sources of misses. This graph shows the total miss rate and its components for a range of cache sizes. This data is for the SPEC2000 integer and floating-point benchmarks and is from the same source as the data in Figure 7.30. The compulsory miss component 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 *eight-way*, *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.

Figure 7.31 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.

Design change	Effect on miss rate	Possible negative performance effect
Increase cache size	decreases capacity misses	may increase access time
Increase associativity	decreases miss rate due to conflict misses	may increase access time
Increase 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 7.32 Memory hierarchy design challenges.

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

The challenge in designing memory hierarchies is that every change that potentially improves the miss rate can also negatively affect overall performance, as Figure 7.32 summarizes. This combination of positive and negative effects is what makes the design of a memory hierarchy interesting.

The BIG Picture

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

Check Yourself

7.6

Real Stuff: The Pentium P4 and the AMD Opteron Memory Hierarchies

In this section, we will look at the memory hierarchy in two modern microprocessors: the Intel Pentium P4 and the AMD Opteron processor. In 2004, the P4 is used in a variety of PC desktops and small servers. The AMD Opteron processor is finding its way into higher-end servers and clusters.

Figure 7.33 shows the Opteron die photo, and Figure 1.9 on page 21 in Chapter 1 shows the P4 die photo. Both have secondary caches on the main processor die. Such integration reduces access time to the secondary cache and also reduces the number of pins on the chip, since there is no need for a bus to an external secondary cache.

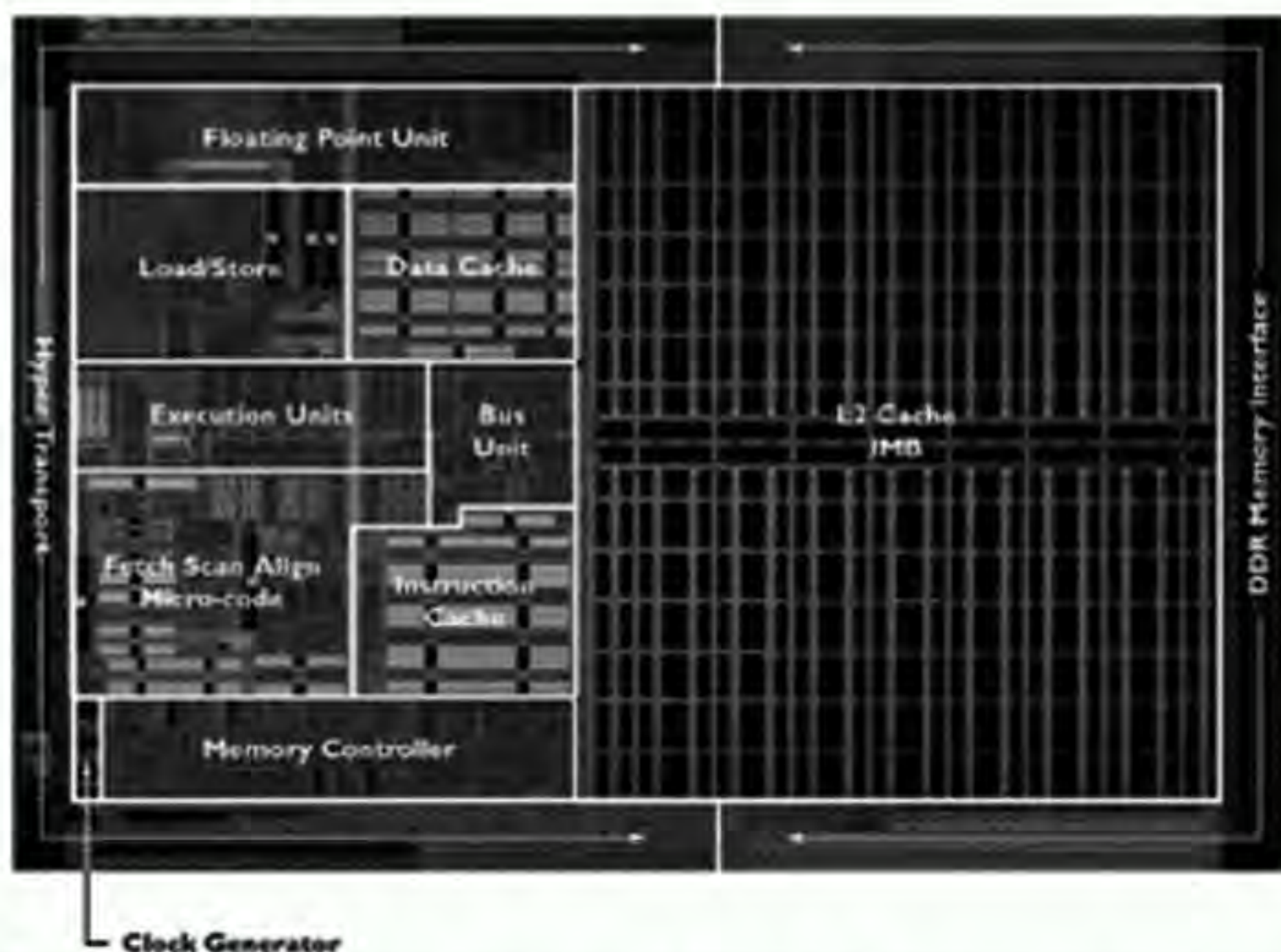


FIGURE 7.33 An AMD Opteron die processor photo with the components labeled. The L2 cache occupies 42% of the die. The remaining components in order of size are HyperTransport™: 13%, DDR memory: 10%, Fetch/Scan/Align/Microcode: 6%, Memory controller: 4%, FPU: 4%, Instruction cache: 4%, Data cache: 4%, Execution units: 3%, Bus unit: 2%, and clock generator: 0.2%. In a 0.13 technology, this die is 193 mm².

The Memory Hierarchies of the P4 and Opteron

Figure 7.34 summarizes the address sizes and TLBs of the two processors.

Note that the AMD Opteron has four TLBs while the P4 has two and that the virtual and physical addresses do not have to match the word size. AMD implements only 48 of the potential 64 bits of its virtual space and 40 of the potential 64 bits of its physical address space. Intel increases the physical address space to 36-bits, although no single program can address more than 32 bits.

Figure 7.35 shows their caches. Note that both the L1 data cache and the L2 caches are larger in the Opteron and that P4 uses a larger block size for its L2 cache than its L1 data cache.

Although the Opteron runs the same IA-32 programs as the Pentium P4, its biggest difference is that it has added a 64-bit addressing mode. Just as the 80386 added a flat 32-bit address space and 32-bit registers to the prior 16-bit 80286 architecture, Opteron adds a new mode with flat 64-bit address space and 64-bit registers to the IA-32 architecture, called *AMD64*. It increases the program counter to 64 bits, extends eight 32-bit registers to 64 bits, adds eight new 64-bit registers, and doubles the number of SSE2 registers. In 2004 Intel announced that future IA-32 processors will include their 64-bit address extension.

Techniques to Reduce Miss Penalties

Both the Pentium 4 and the AMD Opteron have additional optimizations that allow them to reduce the miss penalty. The first of these is the return of the

Characteristic	Intel Pentium P4	AMD Opteron
Virtual address	32 bits	48 bits
Physical address	36 bits	40 bits
Page size	4 KB, 2/4 MB	4 KB, 2/4 MB
TLB organization	1 TLB for instructions and 1 TLB for data Both are four-way set associative Both use pseudo-LRU replacement Both have 128 entries TLB misses handled in hardware	2 TLBs for instructions and 2 TLBs for data Both L1 TLBs fully associative, LRU replacement Both L2 TLBs are four-way set associativity, round-robin LRU Both L1 TLBs have 40 entries Both L2 TLBs have 512 entries TLB misses handled in hardware

FIGURE 7.34 Address translation and TLB hardware for the Intel Pentium P4 and AMD Opteron. The word size sets the maximum size of the virtual address, but a processor need not use all bits. The physical address size is independent of word size. The P4 has one TLB for instructions and a separate identical TLB for data, while the Opteron has both an L1 TLB and an L2 TLB for instructions and identical L1 and L2 TLBs for data. 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	Intel Pentium P4	AMD Opteron
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	8 KB for data, 96 KB trace cache for RISC instructions (12K RISC operations)	64 KB each for instructions/data
L1 cache associativity	4-way set associative	2-way set associative
L1 replacement	Approximated LRU replacement	LRU replacement
L1 block size	64 bytes	64 bytes
L1 write policy	Write-through	Write-back
L2 cache organization	Unified (instruction and data)	Unified (instruction and data)
L2 cache size	512 KB	1024 KB (1 MB)
L2 cache associativity	8-way set associative	16-way set associative
L2 replacement	Approximated LRU replacement	Approximated LRU replacement
L2 block size	128 bytes	64 bytes
L2 write policy	Write-back	Write-back

FIGURE 7.35 First-level and second-level caches in the Intel Pentium P4 and AMD Opteron. The primary caches in the P4 are physically indexed and tagged; for a discussion of the alternatives, see the Elaboration on page 527.

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

requested word first on a miss, as described in the Elaboration on page 490. Both allow the processor to continue to execute instructions that access the data cache during a cache miss. This technique, called a **nonblocking cache**, is commonly used as designers attempt to hide the cache miss latency by using 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 desktop systems, 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.

Both microprocessors prefetch instructions and have a built-in hardware prefetch mechanism for data accesses. They look 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.

A significant challenge facing cache designers is to support processors like the P4 and Opteron that can execute more than one memory instruction per clock cycle. Multiple requests can be supported in the first-level cache by two different techniques. The cache can be multiported, allowing more than one simultaneous access to the same cache block. Multiported caches, however, are often too expensive, since

the RAM cells in a multiported memory must be much larger than single-ported cells. The alternative scheme is to break the cache into banks and allow multiple, independent accesses, provided the accesses are to different banks. The technique is similar to interleaved main memory (see Figure 7.11 on page 489).

To reduce the memory traffic in a multiprocessor configuration, Intel has other versions of the P4 with much larger on-chip caches in 2004. For example, the Intel Pentium P4 Xeon comes with *third-level cache on chip* of 1 MB and is intended for dual-processor servers. A more radical example is the Intel Pentium P4 Extreme Edition, which comes with 2 MB of L3 cache but no support for multiprocessing. These two chips are much larger and more expensive. For example, in 2004 a Precision Workstation 360 with a 3.2 GHz P4 costs about \$1900. Upgrading to the Extreme Edition processor adds \$500 to the price. The Dell Precision Workstation 450, which allows dual processors, costs about \$2000 for a 3.2 GHz Xeon with 1 MB of L3 cache. Adding a second processor like that one adds \$1500 to the price.

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. Future advances in processor pipeline designs, together with the increased use of multiprocessing that presents its own problems in memory hierarchies, provide many new challenges for designers.

Elaboration: Perhaps the largest difference between the AMD and Intel chips is the use of a trace cache for the P4 instruction cache, while the AMD Opteron uses a more traditional instruction cache.

Instead of organizing the instructions in a cache block sequentially to promote spatial locality, a *trace cache* finds a dynamic sequence of instructions *including taken branches* to load into a cache block. Thus, the cache blocks contain dynamic *traces* of the executed instructions as determined by the CPU rather than static sequences of instructions as determined by memory layout. It folds branch prediction (Chapter 6) into the cache, so the branches must be validated along with the addresses in order to have a valid fetch. In addition, the P4 caches the micro-operations (see Chapter 5) rather than the IA-32 instructions as in the Opteron.

Clearly, trace caches have much more complicated address mapping mechanisms, since the addresses are no longer aligned to power-of-two multiples of the word size.

Trace caches can improve utilization of cache blocks, however. For example, very long blocks in conventional caches may be entered from a taken branch, and hence the first portion of the block occupies space in the cache that might not be fetched. Similarly, such blocks may be exited by taken branches, so the last portion of the block might be wasted. Given that taken branches or jumps occur every 5–10 instructions, effective block utilization is a real problem for processors like the Opteron, whose 64-byte block would likely include 16–24 80x86 instructions. Trace caches store instructions only from the branch entry point to the exit of the trace, thereby avoiding such

Byte address 300 is block address

$$\left\lfloor \frac{300}{32} \right\rfloor = 9$$

The number of blocks in the cache is

$$\left\lfloor \frac{256}{32} \right\rfloor = 8$$

Block number 9 falls into cache block number $(9 \bmod 8) = 1$.

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

This could easily be written as a fallacy: “Programmers can ignore memory hierarchies in writing code.” We illustrate with an example using matrix multiply, to complement the sort comparison in Figure 7.18 on page 508.

Here is the inner loop of the version of matrix multiply from Chapter 3:

```
for (i=0; i!=500; i=i+1)
  for (j=0; j!=500; j=j+1)
    for (k=0; k!=500; k=k+1)
      x[i][j] = x[i][j] + y[i][k] * z[k][j];
```

When run with inputs that are 500×500 double precision matrices, the CPU runtime of the above loop on a MIPS CPU with a 1 MB secondary cache was about half the speed compared to when the loop order is changed to k, j, i (so i is innermost)! The only difference is how the program accesses memory and the ensuing effect on the memory hierarchy. Further compiler optimizations using a technique called *blocking* can result in a runtime that is another four times faster for this code!

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.

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. Given the size of DRAMs and Moore's law, many of today's 32-bit systems are facing similar problems.

7.8

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. Figure 7.36 compares the memory hierarchy of microprocessors aimed at desktop, server, and embedded applications. The L1 caches are similar across applications, with the primary differences being L2 cache size, die size, processor clock rate, and instructions issued per clock.

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.

Because processor speeds continue to improve faster than either DRAM access times or disk access times, memory will increasingly be the factor that limits performance. Processors increase in performance at a high rate, and DRAMs are now doubling their density about every two years. The *access time* of DRAMs, however, is improving at a much slower rate—less than 10% per year. Figure 7.37 plots processor performance against a 7% annual performance improvement in DRAM latency. While latency improves slowly, recent enhancements in DRAM technology (double data rate DRAMs and related techniques) have led to greater

MPU	AMD Opteron	Intrinsity FastMATH	Intel Pentium 4	Intel PXA250	Sun UltraSPARC IV
Instruction set architecture	IA-32, AMD64	MIPS32	IA-32	ARM	SPARC v9
Intended application	server	embedded	desktop	low-power embedded	server
Die size (mm ²) (2004)	193	122	217		356
Instructions issued/clock	3	2	3 RISC ops	1	4 × 2
Clock rate (2004)	2.0 GHz	2.0 GHz	3.2 GHz	0.4 GHz	1.2 GHz
Instruction cache	64 KB, 2-way set associative	16 KB, direct mapped	12000 RISC op trace cache (~96 KB)	32 KB, 32-way set associative	32 KB, 4-way set associative
Latency (clocks)	3?	4	4	1	2
Data cache	64 KB, 2-way set associative	16 KB, 1-way set associative	8 KB, 4-way set associative	32 KB, 32-way set associative	64 KB, 4-way set associative
Latency (clocks)	3	3	2	1	2
TLB entries (I/D/L2 TLB)	40/40/512/ 512	16	128/128	32/32	128/512
Minimum page size	4 KB	4 KB	4 KB	1 KB	8 KB
On-chip L2 cache	1024 KB, 16-way set associative	1024 KB, 4-way set associative	512 KB, 8-way set associative	—	—
Off-chip L2 cache	—	—	—	—	16 MB, 2-way set associative
Block size (L1/L2, bytes)	64	64	64/128	32	32

FIGURE 7.36 Desktop, embedded, and server microprocessors in 2004. From a memory hierarchy perspective, the primary differences between categories is the L2 cache. There is no L2 cache for the low-power embedded, a large on-chip L2 for the embedded and desktop, and 16 MB off chip for the server. The processor clock rates also vary: 0.4 GHz for low-power embedded, 1 GHz or higher for the rest. Note that UltraSPARC IV has two processors on the chip.

increases in memory bandwidth. This potentially higher memory bandwidth has enabled designers to increase cache block sizes with smaller increases in the miss penalty.

Recent Trends

The challenge in designing memory hierarchies to close this growing gap, as we noted in the Big Picture on page 545, is that all the hardware design choices for memory hierarchies have both a positive and negative effect on performance. This means that for each level of the hierarchy there is an optimal performance point per program, which must include some misses. If this is the case, how can we overcome the growing gap between processor speeds and lower levels of the hierarchy? This question is currently the topic of much research.

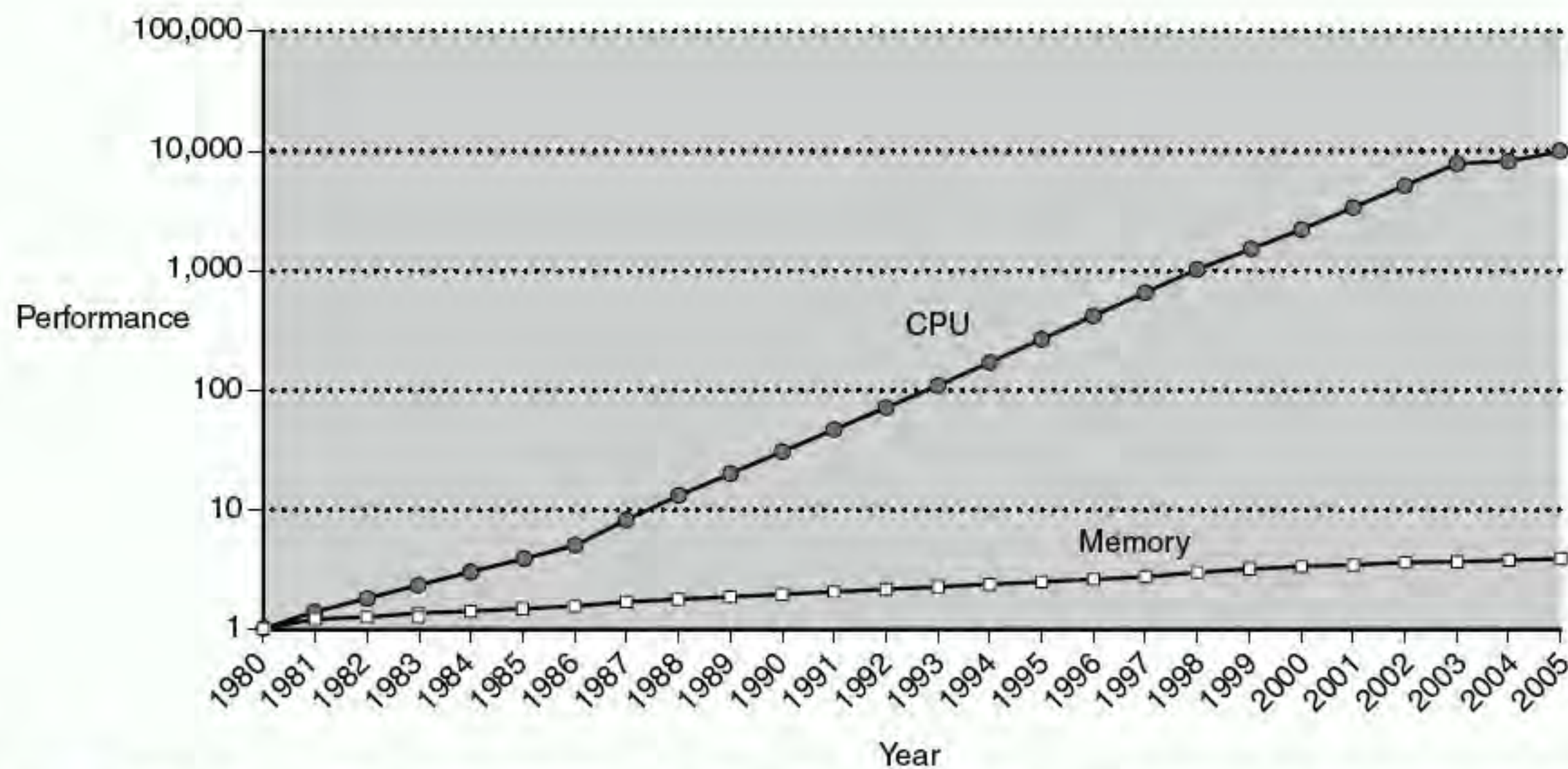


FIGURE 7.37 Using their 1980 performance as a baseline, the access time of DRAMs versus the performance of processors is plotted over time Note that the vertical axis must be on a logarithmic scale to record the size of the processor-DRAM performance gap. The memory baseline is 64 KB DRAM in 1980, with three years to the next generation until 1996 and two years thereafter, with a 7% per year performance improvement in latency. The processor line assumes a 35% improvement per year until 1986, and a 55% improvement until 2003. It slows thereafter.

On-chip first-level caches initially helped close the gap that was growing between processor clock cycle time and off-chip SRAM cycle time. To narrow the gap between the small on-chip caches and DRAM, second-level caches became widespread. Today, all desktop computers use second-level caches on chip, and third-level caches are becoming popular in some segments. Multilevel caches also make it possible to use other optimizations more easily for two reasons. First, the design parameters of a second- or third-level cache are different from a first-level cache. For example, because a second- or third-level cache will be much larger, it is possible to use larger block sizes. Second, a second- or third-level cache is not constantly being used by the processor, as a first-level cache is. This allows us to consider having the second- or third-level cache do something when it is idle that may be useful in preventing future misses.

Another possible direction 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. By

restructuring the loops that access the arrays, substantially improved locality—and, therefore, cache performance—can be obtained. The example on page 551 showed how effective even a simple change of loop structure could be.

Another direction is to try to use compiler-directed **prefetching**. In prefetching, a block of data is brought into the cache before it is actually referenced. The compiler tries to identify data blocks needed in the future and, using special instructions, tells the memory hierarchy to move the blocks into the cache. When the block is actually referenced, it is found in the cache, rather than causing a cache miss. The use of secondary caches has made prefetching even more attractive, since the secondary cache can be involved in a prefetch, while the primary cache continues to service processor requests.

As we will see in  **Chapter 9**, memory systems are also a central design issue for parallel processors. The growing importance of the memory hierarchy in determining system performance in both uniprocessor and multiprocessor systems means that this important area will continue to be a focus of both designers and researchers for some years to come.

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.



Historical Perspective and Further Reading

This history section  gives an overview of memory technologies, from mercury delay lines to DRAM, the invention of the memory hierarchy and protection mechanisms, and concludes with a brief history of operating systems, including CTSS, MULTICS, UNIX, BSD UNIX, MS-DOS, Windows, and Linux.

7.10

Exercises

7.1 [5] <§7.1> SRAM is commonly used to implement small, fast, on-chip caches while DRAM is used for larger, slower main memory. In the past, a common design for supercomputers was to build machines with no caches and main memories made entirely out of SRAM (the Cray C90, for example, a very fast computer in its day). If cost were no object, would you still want to design a system this way?

7.2 [10] <§7.2> Describe the general characteristics of a program that would exhibit very little temporal and spatial locality with regard to data accesses. Provide an example program (pseudocode is fine).

7.3 [10] <§7.2> Describe the general characteristics of a program that would exhibit very high amounts of temporal locality but very little spatial locality with regard to data accesses. Provide an example program (pseudocode is fine).

7.4 [10] <§7.2> Describe the general characteristics of a program that would exhibit very little temporal locality but very high amounts of spatial locality with regard to data accesses. Provide an example program (pseudocode is fine).

7.5 [3/3] <§7.2> A new processor can use either a write-through or write-back cache selectable through software.

- a. Assume the processor will run data intensive applications with a large number of load and store operations. Explain which cache write policy should be used.
- b. Consider the same question but this time for a safety critical system in which data integrity is more important than memory performance.

7.6 [10] <§7.2>  **For More Practice:** Locality.

7.7 [10] <§7.2>  **For More Practice:** Locality.

7.8 [10] <§7.2>  **For More Practice:** Locality.

7.9 [10] <§7.2> Here is a series of address references given as word addresses: 2, 3, 11, 16, 21, 13, 64, 48, 19, 11, 3, 22, 4, 27, 6, and 11. Assuming a direct-mapped cache with 16 one-word blocks that is initially empty, label each reference in the list as a hit or a miss and show the final contents of the cache.

7.10 [10] <§7.2> Using the series of references given in Exercise 7.9, show the hits and misses and final cache contents for a direct-mapped cache with four-word blocks and a *total size* of 16 words.

7.11 [15] <§7.2> Given the following pseudocode:

```
int array[10000,100000];
for each element array[i][j] {
    array[i][j] = array[i][j]*2;
}
```

write two C programs that implement this algorithm: one should access the elements of the array in row-major order, and the other should access them in column-major order. Compare the execution times of the two programs. What does this tell you about the effects of memory layout on cache performance?

7.12 [10] <§7.2> Compute the total number of bits required to implement the cache in Figure 7.9 on page 486. This number is different from the size of the

cache, which usually refers to the number of bytes of data stored in the cache. The number of bits needed to implement the cache represents the total amount of memory needed for storing all the data, tags, and valid bits.

7.13 [10] <§7.2> Find a method to eliminate the AND gate on the valid bit in Figure 7.7 on page 478. (Hint: You need to change the comparison.)

7.14 [10] <§7.2> Consider a memory hierarchy using one of the three organizations for main memory shown in Figure 7.11 on page 489. Assume that the cache block size is 16 words, that the width of organization (b) of the figure is four words, and that the number of banks in organization (c) is four. If the main memory latency for a new access is 10 memory bus clock cycles and the transfer time is 1 memory bus clock cycle, what are the miss penalties for each of these organizations?

7.15 [10] <§7.2>  **For More Practice:** Cache Performance.

7.16 [15] <§7.2> Cache C1 is direct-mapped with 16 one-word blocks. Cache C2 is direct-mapped with 4 four-word blocks. Assume that the miss penalty for C1 is 8 memory bus clock cycles and the miss penalty for C2 is 11 memory bus clock cycles. Assuming that the caches are initially empty, find a reference string for which C2 has a lower miss rate but spends more memory bus clock cycles on cache misses than C1. Use word addresses.

7.17 [5] <§7.2>  **In More Depth:** Average Memory Access Time

7.18 [5] <§7.2>  **In More Depth:** Average Memory Access Time

7.19 [10] <§7.2>  **In More Depth:** Average Memory Access Time

7.20 [10] <§7.2> Assume a memory system that supports interleaving either four reads or four writes. Given the following memory addresses in order as they appear on the memory bus: 3, 9, 17, 2, 51, 37, 13, 4, 8, 41, 67, 10, which ones will result in a bank conflict?

7.21 [3 hours] <§7.3> Use a cache simulator to simulate several different cache organizations for the first 1 million references in a trace of gcc. Both dinero (a cache simulator) and the gcc traces are available—see the Preface of this book for information on how to obtain them. Assume an instruction cache of 32 KB and a data cache of 32 KB using the same organization. You should choose at least two kinds of associativity and two block sizes. Draw a diagram like that in Figure 7.17 on page 503 that shows the data cache organization with the best hit rate.

7.22 [1 day] <§7.3> You are commissioned to design a cache for a new system. It has a 32-bit physical byte address and requires separate instruction and data caches. The SRAMs have an access time of 1.5 ns and a size of $32K \times 8$ bits, and you have a total of 16 SRAMs to use. The miss penalty for the memory system is $8 + 2 \times \text{Block}$

size in words. Using set associativity adds 0.2 ns to the cache access time. Using the first 1 million references of gcc, find the best I and D cache organizations, given the available SRAMs.

7.23 [10] <§7.2, B.5>  [For More Practice](#): Cache Configurations

7.24 [10] <§7.2, B.5>  [For More Practice](#): Cache Configurations

7.25 [10] <§7.3>  [For More Practice](#): Cache Operation

7.26 [10] <§7.3>  [For More Practice](#): Cache Operation

7.27 [10] <§7.3>  [For More Practice](#): Cache Operation

7.28 [5] <§7.3> Associativity usually improves the miss ratio, but not always. Give a short series of address references for which a two-way set-associative cache with LRU replacement would experience more misses than a direct-mapped cache of the same size.

7.29 [15] <§7.3> Suppose a computer's address size is k bits (using byte addressing), the cache size is S bytes, the block size is B bytes, and the cache is A -way set-associative. Assume that B is a power of two, so $B = 2^b$. Figure out what the following quantities are in terms of S , B , A , b , and k : the number of sets in the cache, the number of index bits in the address, and the number of bits needed to implement the cache (see Exercise 7.12).

7.30 [10] <§7.3>  [For More Practice](#): Cache Configurations.

7.31 [10] <§7.3>  [For More Practice](#): Cache Configurations.

7.32 [20] <§7.3> Consider three processors with different cache configurations:

- *Cache 1*: Direct-mapped with one-word blocks
- *Cache 2*: Direct-mapped with four-word blocks
- *Cache 3*: Two-way set associative with four-word blocks

The following miss rate measurements have been made:

- *Cache 1*: Instruction miss rate is 4%; data miss rate is 6%.
- *Cache 2*: Instruction miss rate is 2%; data miss rate is 4%.
- *Cache 3*: Instruction miss rate is 2%; data miss rate is 3%.

For these processors, one-half of the instructions contain a data reference. Assume that the cache miss penalty is $6 + \text{Block size in words}$. The CPI for this workload was measured on a processor with cache 1 and was found to be 2.0. Determine which processor spends the most cycles on cache misses.

7.33 [5] <§7.3> The cycle times for the processors in Exercise 7.32 are 420 ps for the first and second processors and 310 ps for the third processor. Determine which processor is the fastest and which is the slowest.

7.34 [15] <§7.3> Assume that the cache for the system described in Exercise 7.32 is two-way set associative and has eight-word blocks and a total size of 16 KB. Show the cache organization and access using the same format as Figure 7.17 on page 503.

7.35 [10] <§§7.2, 7.4> The following C program is run (with no optimizations) on a processor with a cache that has eight-word (32-byte) blocks and holds 256 bytes of data:

```
int i,j,c,stride,array[512];
...
for (i=0; i<10000; i++)
    for (j=0; j<512; j=j+stride)
        c = array[j]+17;
```

If we consider only the cache activity generated by references to the array and we assume that integers are words, what is the expected miss rate when the cache is direct mapped and stride = 256? How about if stride = 255? Would either of these change if the cache were two-way set associative?

7.36 [10] <§§7.3, B.5>  [For More Practice](#): Cache Configurations

7.37 [5] <§§7.2–7.4>  [For More Practice](#): Memory Hierarchy Interactions

7.38 [4 hours] <§§7.2–7.4> We want to use a cache simulator to simulate several different TLB and virtual memory organizations. Use the first 1 million references of gcc for this evaluation. We want to know the TLB miss rate for each of the following TLBs and page sizes:

1. 64-entry TLB with full associativity and 4 KB pages
2. 32-entry TLB with full associativity and 8 KB pages
3. 64-entry TLB with eight-way associativity and 4 KB pages
4. 128-entry TLB with four-way associativity and 4 KB pages

7.39 [15] <§7.4> Consider a virtual memory system with the following properties:

- 40-bit virtual byte address
- 16 KB pages
- 36-bit physical byte address

What is the total size of the page table for each process on this processor, assuming that the valid, protection, dirty, and use bits take a total of 4 bits and that all the virtual pages are in use? (Assume that disk addresses are not stored in the page table.)

7.40 [15] <§7.4> Assume that the virtual memory system of Exercise 7.39 is implemented with a two-way set-associative TLB with a total of 256 TLB entries. Show the virtual-to-physical mapping with a figure like Figure 7.24 on page 525. Make sure to label the width of all fields and signals.

7.41 [10] <§7.4> A processor has a 16-entry TLB and uses 4 KB pages. What are the performance consequences of this memory system if a program accesses at least 2 MB of memory at a time? Can anything be done to improve performance?

7.42 [10] <§7.4> Buffer overflows are a common exploit used to gain control of a system. If a buffer is allocated on the stack, a hacker could overflow the buffer and insert a sequence of malicious instructions compromising the system. Can you think of a hardware mechanism that could be used to prevent this?

7.43 [15] <§7.4>  [For More Practice](#): Hierarchical Page Tables

7.44 [15] <§7.4>  [For More Practice](#): Hierarchical Page Tables

7.45 [5] <§7.5> If all misses are classified into one of three categories—compulsory, capacity, or conflict (as discussed on page 543)—which misses are likely to be reduced when a program is rewritten so as to require less memory? How about if the clock rate of the processor that the program is running on is increased? How about if the associativity of the existing cache is increased?

7.46 [5] <§7.5> The following C program could be used to help construct a cache simulator. Many of the data types have not been defined, but the code accurately describes the actions that take place during a read access to a direct-mapped cache.

```
word ReadDirectMappedCache(address a)
    static Entry cache[CACHE_SIZE_IN_WORDS];
    Entry e = cache[a.index];
    if (e.valid == FALSE || e.tag != a.tag) {
        e.valid = true;
        e.tag = a.tag;
        e.data = load_from_memory(a);
    }
    return e.data;
```

Your task is to modify this code to produce an accurate description of the actions that take place during a read access to a direct-mapped cache with multiple-word blocks.

7.47 [8] <§7.5> This exercise is similar to Exercise 7.46, except this time write the code for read accesses to an n -way set-associative cache with one-word blocks. Note that your code will likely suggest that the comparisons are sequential in nature when in fact they would be performed in parallel by actual hardware.

7.48 [8] <§7.5> Extend your solution to Exercise 7.46 by including the specification of a new procedure for handling write accesses, assuming a write-through policy. Be sure to consider whether or not your solution for handling read accesses needs to be modified.

7.49 [8] <§7.5> Extend your solution to Exercise 7.46 by including the specification of a new procedure for handling write accesses, assuming a write-back policy. Be sure to consider whether or not your solution for handling read accesses needs to be modified.

7.50 [8] <§7.5> This exercise is similar to Exercise 7.48, but this time extend your solution to Exercise 7.47. Assume that the cache uses random replacement.

7.51 [8] <§7.5> This exercise is similar to Exercise 7.49, but this time extend your solution to Exercise 7.47. Assume that the cache uses random replacement.

7.52 [5] <§§7.7–7.8> Why might a compiler perform the following optimization?

```
/* Before */  
  
for (j = 0; j < 20; j++)  
    for (i = 0; i < 200; i++)  
        x[i][j] = x[i][j] + 1;  
  
/* After */  
  
for (i = 0; i < 200; i++)  
    for (j = 0; j < 20; j++)  
        x[i][j] = x[i][j] + 1;
```

§7.1, page 472: 1.

§7.2, page 491: 1 and 4: A lower miss penalty can lead to smaller blocks, yet higher memory bandwidth usually leads to larger blocks, since the miss penalty is only slightly larger.

§7.3, page 510: 1.

§7.4, page 538: 1-a, 2-c, 3-c, 4-d.

§7.5, page 545: 2.

**Answers To
Check Yourself**

Computers in the Real World

Saving the World's Art Treasures

Problem: Find a way to help conserve artwork threatened by environmental factors and aging or damaged by earlier attempts at restoration without causing further harm to irreplaceable artworks.

Solution: Use computers and scientific instrumentation to analyze the artwork and its setting, enabling art conservators and restorers to undertake a more informed and successful preservation of an artwork.

Art conservation and restoration have developed into high-technology fields that make extensive use of computing and scientific instrumentation. For example, one of the most challenging forms of art to restore and maintain are frescoes, which are painted in the wet plaster of a wall or ceiling. Moisture and heat change the surface and cause deterioration; similarly air pollution, smoke from candles, and other contaminants directly attack the paint as well as add dirt and grime that cover the original artwork.

During the restoration of Michelangelo's frescoes in the Sistine Chapel, computers were used to survey the ceiling, finding cracks and precisely mapping the surface and the frescos.

Since the width of the ceiling and walls varies from about three feet to almost six feet, there are significant differences in thermal behavior, which in turn affects the surface painting. Computers were used to model the entire structure, including the high humidity generated when a thousand people stand inside the chapel on a warm day! This led to a computer-controlled climate system that uses sensors placed in strategic locations. The goal is to keep the visitors cool while preserving Michelangelo's masterpiece for generations to come.



A laser scan of Michelangelo's statue of David

Perhaps the area of art conservation that has been most affected by the availability of low-cost, high-performance computation has been painting restoration. Three techniques—*infrared reflectography*, *ultraviolet imaging*, and *X-radiography*—have found the heaviest use. Because of the need for highly precise, high resolution imaging, computer controlled cameras or X-ray scanners are used in all these techniques. This results in a patchwork of images, which are then stitched together by a computer. The combination of computer controlled motion of a camera or X-ray scanner and subsequent computer composition of tens to thousands of images permits that scanning of large surfaces at very high resolution.

Infrared reflectography uses light in the near-infrared spectrum and a digital camera to detect the intensity of reflection of the light from the surface of a painting, mural, or fresco. This technique is useful for finding the underdrawing that most artists use to initially sketch out the forms in a painting. The underdrawing, typically done in black, often using charcoal, absorbs the infrared light. In the fig-

ure below, are two images of a painting: one shown in normal light (on the left) and one using the *infrared reflectography* technique (on the right).

Restorers use *ultraviolet imaging* to look at the original colors of a painting that has been retouched. *X-radiography* provides similar information, since white and yellow pigments that were covered or painted over appear darker due to their lead content.

Scanning technologies have also been applied to three-dimensional art objects, such as sculpture. Michelangelo's *David* was scanned using a laser range finder by a group led by Professor Marc Levoy at Stanford. The resulting database for a scan with 0.29 mm resolution consists of over 2 billion polygons and 32 gigabytes of data. The Digital Michelangelo project has created a detailed model of the famous sculpture useful both for conservation as well as an educational tool for students around the world. Two of the many images that can be derived from the three-dimensional scan as shown opposite.

To learn more see these references on the  library

Conserving paintings, a site dedicated to Harvard University's digital imaging lab.

Sistine Chapel, a short background on the Sistine Chapel.

The Digital Michelangelo project



An image from the Sistine Chapel in normal light (left) and in infrared (right).

8

Storage, Networks, and Other Peripherals

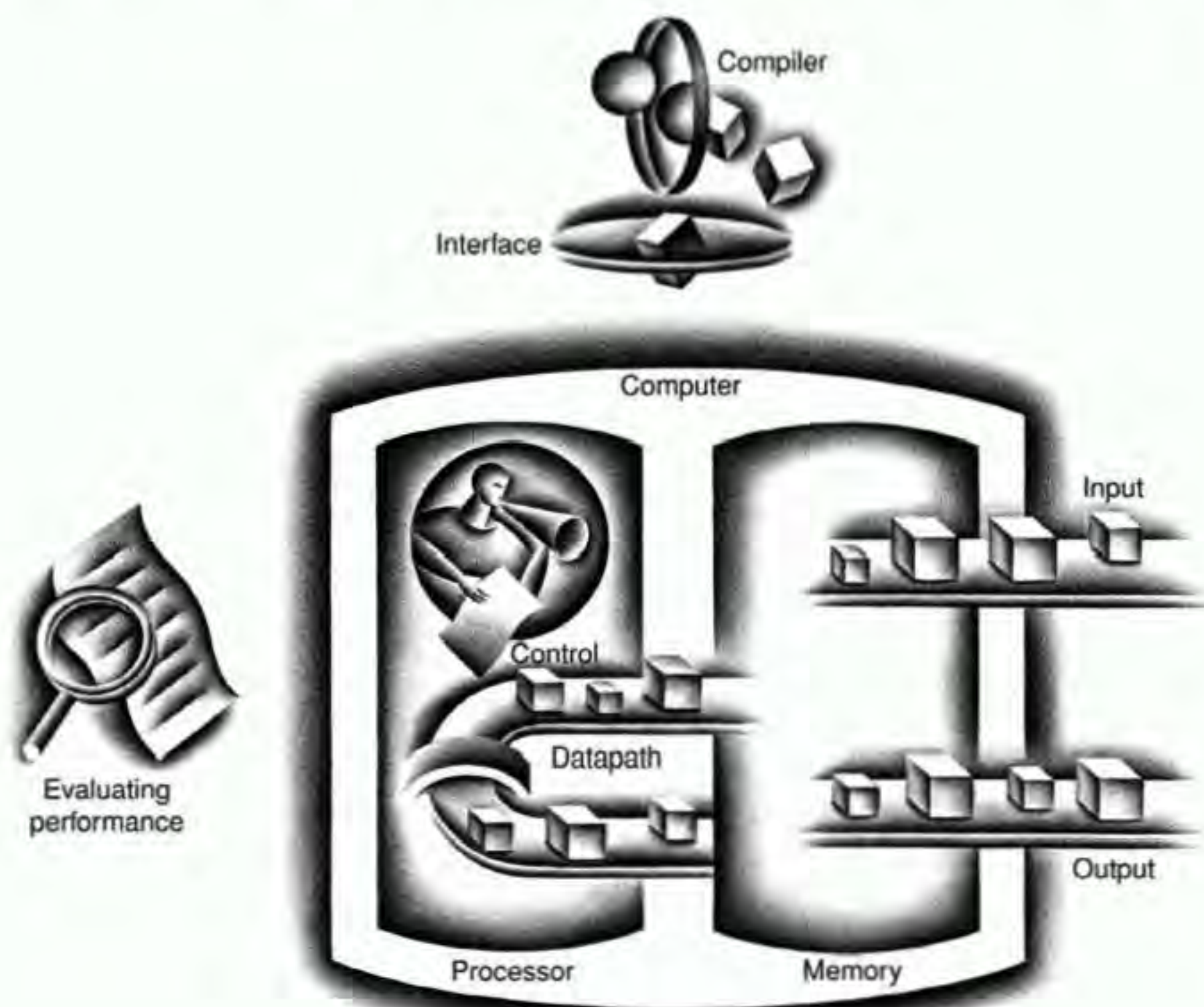
*Combining bandwidth and storage . . .
enables swift and reliable access to
the ever-expanding troves of content
on the proliferating disks and . . .
repositories of the Internet.*

George Gilder

The End Is Drawing Nigh, 2000

8.1	Introduction	566
8.2	Disk Storage and Dependability	569
 8.3	Networks	580
8.4	Buses and Other Connections between Processors, Memory, and I/O Devices	581
8.5	Interfacing I/O Devices to the Processor, Memory, and Operating System	588
8.6	I/O Performance Measures: Examples from Disk and File Systems	597
8.7	Designing an I/O System	600
8.8	Real Stuff: A Digital Camera	603
8.9	Fallacies and Pitfalls	606
8.10	Concluding Remarks	609
 8.11	Historical Perspective and Further Reading	611
8.12	Exercises	611

The Five Classic Components of a Computer



8.1 Introduction

Although users can get frustrated if their computer hangs and must be rebooted, they become apoplectic if their storage system crashes and they lose information. Thus, the bar for dependability is much higher for storage than for computation. Networks also plan for failures in communication, including several mechanisms to detect and recover from such failures. Hence, I/O systems generally place much greater emphasis on dependability and cost, while processors and memory focus on performance and cost.

I/O systems must also plan for expandability and for diversity of devices, which is not a concern for processors. Expandability is related to storage capacity, which is another design parameter for I/O systems; systems may need a lower bound of storage capacity to fulfill their role.

Although performance plays a smaller role for I/O, it is more complex. For example, with some devices we may care primarily about access latency, while with others throughput is crucial. Furthermore, performance depends on many aspects of the system: the device characteristics, the connection between the device and the rest of the system, the memory hierarchy, and the operating system. Figure 8.1 shows the structure of a simple system with its I/O. All of the components, from the individual I/O devices to the processor to the system software, will affect the dependability, expandability, and performance of tasks that include I/O.

I/O devices are incredibly diverse. Three characteristics are useful in organizing this wide variety:

- *Behavior:* Input (read once), output (write only, cannot be read), or storage (can be reread and usually rewritten).
- *Partner:* Either a human or a machine is at the other end of the I/O device, either feeding data on input or reading data on output.
- *Data rate:* The peak rate at which data can be transferred between the I/O device and the main memory or processor. It is useful to know what maximum demand the device may generate.

For example, a keyboard is an *input* device used by a *human* with a *peak data rate* of about 10 bytes per second. Figure 8.2 shows some of the I/O devices connected to computers.

In Chapter 1, we briefly discussed four important and characteristic I/O devices: mice, graphics displays, disks, and networks. In this chapter we go into much more depth on disk storage and networks.

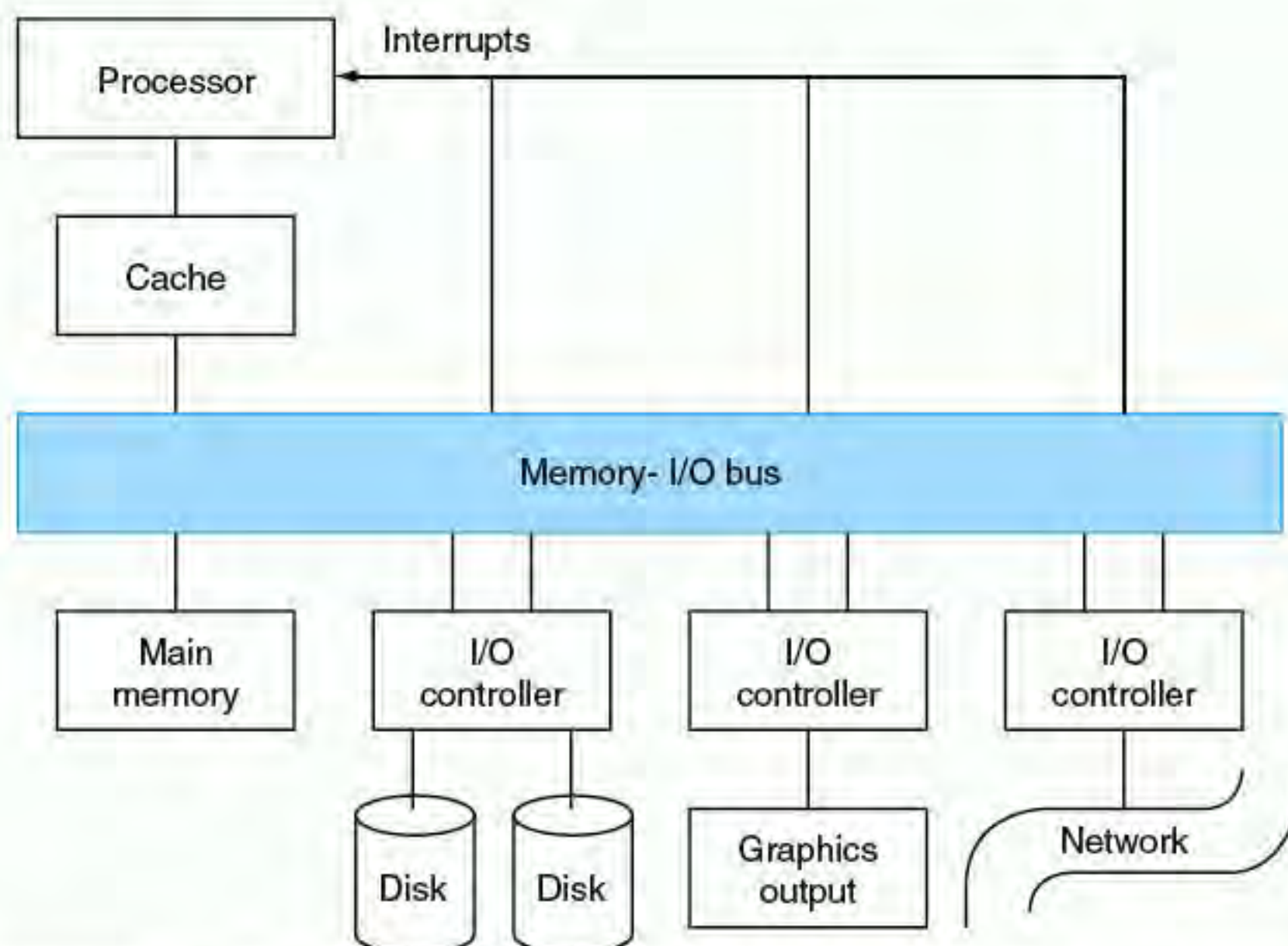


FIGURE 8.1 A typical collection of I/O devices. The connections between the I/O devices, processor, and memory are usually called *buses*. Communication among the devices and the processor use both interrupts and protocols on the bus, as we will see in this chapter. Figure 8.11 on page 585 shows the organization for a desktop PC.

How we should assess I/O performance often depends on the application. In some environments, we may care primarily about system throughput. In these cases, I/O bandwidth will be most important. Even I/O bandwidth can be measured in two different ways:

1. How much data can we move through the system in a certain time?
2. How many I/O operations can we do per unit of time?

Which performance measurement is best may depend on the environment. For example, in many multimedia applications, most I/O requests are for long streams of data, and transfer bandwidth is the important characteristic. In another environment, we may wish to process a large number of small, unrelated accesses to an I/O device. An example of such an environment might be a tax-processing office of the National Income Tax Service (NITS). NITS mostly cares about processing a large number of forms in a given time; each tax form is stored separately and is fairly small. A system oriented toward large file transfer may be satisfactory, but an I/O system that can support the simultaneous transfer of many small files may be cheaper and faster for processing millions of tax forms.

Device	Behavior	Partner	Data rate (Mbit/sec)
Keyboard	input	human	0.0001
Mouse	input	human	0.0038
Voice input	input	human	0.2640
Sound input	input	machine	3.0000
Scanner	input	human	3.2000
Voice output	output	human	0.2640
Sound output	output	human	8.0000
Laser printer	output	human	3.2000
Graphics display	output	human	800.0000–8000.0000
Modem	input or output	machine	0.0160–0.0640
Network/LAN	input or output	machine	100.0000–1000.0000
Network/wireless LAN	input or output	machine	11.0000–54.0000
Optical disk	storage	machine	80.0000
Magnetic tape	storage	machine	32.0000
Magnetic disk	storage	machine	240.0000–2560.0000

FIGURE 8.2 The diversity of I/O devices. I/O devices can be distinguished by whether they serve as input, output, or storage devices; their communication partner (people or other computers); and their peak communication rates. The data rates span eight orders of magnitude. Note that a network can be an input or an output device, but cannot be used for storage. Transfer rates for devices are always quoted in base 10, so that 10 Mbit/sec = 10,000,000 bits/sec.

I/O requests Reads or writes to I/O devices.

In other applications, we care primarily about response time, which you will recall is the total elapsed time to accomplish a particular task. If the **I/O requests** are extremely large, response time will depend heavily on bandwidth, but in many environments most accesses will be small, and the I/O system with the lowest latency per access will deliver the best response time. On single-user machines such as desktop computers and laptops, response time is the key performance characteristic.

A large number of applications, especially in the vast commercial market for computing, require both high throughput and short response times. Examples include automatic teller machines (ATMs), order entry and inventory tracking systems, file servers, and Web servers. In such environments, we care about both how long each task takes *and* how many tasks we can process in a second. The number of ATM requests you can process per hour doesn't matter if each one takes 15 minutes—you won't have any customers left! Similarly, if you can process each ATM request quickly but can only handle a small number of requests at once, you won't be able to support many ATMs, or the cost of the computer per ATM will be very high.

In summary, the three classes of desktop, server, and embedded computers are sensitive to I/O dependability and cost. Desktop and embedded systems are more

focused on response time and diversity of I/O devices, while server systems are more focused on throughput and expandability of I/O devices.

8.2

Disk Storage and Dependability

As mentioned in Chapter 1, magnetic disks rely on a rotating platter coated with a magnetic surface and use a moveable read/write head to access the disk. Disk storage is **nonvolatile**—the data remains even when power is removed. A magnetic disk consists of a collection of platters (1–4), each of which has two recordable disk surfaces. The stack of platters is rotated at 5400 to 15,000 RPM and has a diameter from an inch to just over 3.5 inches. Each disk surface is divided into concentric circles, called **tracks**. There are typically 10,000 to 50,000 tracks per surface. Each track is in turn divided into **sectors** that contain the information; each track may have 100 to 500 sectors. Sectors are typically 512 bytes in size, although there is an initiative to increase the sector size to 4096 bytes. The sequence recorded on the magnetic media is a sector number, a gap, the information for that sector including error correction code (see  **Appendix B**, page B-64), a gap, the sector number of the next sector, and so on. Originally, all tracks had the same number of sectors and hence the same number of bits, but with the introduction of zone bit recording (ZBR) in the early 1990s, disk drives changed to a varying number of sectors (and hence bits) per track, instead keeping the spacing between bits constant. ZBR increases the number of bits on the outer tracks and thus increases the drive capacity.

As we saw in Chapter 1, to read and write information the read/write heads must be moved so that they are over the correct location. The disk heads for each surface are connected together and move in conjunction, so that every head is over the same track of every surface. The term *cylinder* is used to refer to all the tracks under the heads at a given point on all surfaces.

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

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 has decided to calculate 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 14 ms, but, depending on the application and scheduling of disk requests, the actual average seek time may be only 25% to 33% of the

nonvolatile Storage device where data retains its value even when power is removed.

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.

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

rotation latency Also called 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.

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. Because the disks rotate at 5400 RPM to 15,000 RPM, the average rotational latency is between

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

and

$$\begin{aligned}\text{Average rotational latency} &= \frac{0.5 \text{ rotation}}{15,000 \text{ RPM}} = \frac{0.5 \text{ rotation}}{15,000 \text{ RPM} / \left(60 \frac{\text{seconds}}{\text{minute}}\right)} \\ &= 0.0020 \text{ seconds} = 2.0 \text{ ms}\end{aligned}$$

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 2004 are between 30 and 80 MB/sec. The 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 may be up to 320 MB/sec in 2004. Today, most disk transfers are multiple sectors in length.

A *disk controller* usually handles the detailed control of the disk and the transfer between the disk and the memory. The controller adds the final component of disk access time, *controller time*, which is the overhead the controller imposes in performing an I/O access. The average time to perform an I/O operation will consist of these four times plus any wait time incurred because other processes are using the disk.

EXAMPLE

Disk Read Time

What is the average time to read or write a 512-byte sector for a typical disk rotating at 10,000 RPM? The advertised average seek time is 6 ms, the transfer rate is 50 MB/sec, and the controller overhead is 0.2 ms. Assume that the disk is idle so that there is no waiting time.

ANSWER

Average disk access time is equal to Average seek time + Average rotational delay + Transfer time + Controller overhead. Using the advertised average seek time, the answer is

$$6.0 \text{ ms} + \frac{0.5 \text{ rotation}}{10,000 \text{ RPM}} + \frac{0.5 \text{ KB}}{50 \text{ MB/sec}} + 0.2 \text{ ms} = 6.0 + 3.0 + 0.01 + 0.2 = 9.2 \text{ ms}$$

If the measured average seek time is 25% of the advertised average time, the answer is

$$1.5 \text{ ms} + 3.0 \text{ ms} + 0.01 \text{ ms} + 0.2 \text{ ms} = 4.7 \text{ ms}$$

Notice that when we consider measured average seek time, as opposed to advertised average seek time, the rotational latency can be the largest component of the access time.

Disk densities have continued to increase for more than 50 years. The impact of this compounded improvement in density and the reduction in physical size of a disk drive has been amazing, as Figure 8.3 shows. The aims of different disk designers have led to a wide variety of drives being available at any particular time. Figure 8.4 shows the characteristics of three magnetic disks. In 2004, these disks from a single manufacturer cost between \$0.50 and \$5 per gigabyte, depending on size, interface, and performance. The smaller drive has advantages in power and volume per byte.

Elaboration: Most disk controllers include caches. Such caches allow for fast access to data that was recently read between transfers requested by the CPU. They use write through and do not update on a write miss. They often also include prefetch algorithms to try to anticipate demand. Of course, such capabilities complicate the measurement of disk performance and increase the importance of workload choice.

Dependability, Reliability, and Availability

Users crave dependable storage, but how do you define it? In the computer industry, it is harder than looking it up in the dictionary. After considerable debate, the following is considered the standard definition (Laprie 1985):

Computer system dependability is the quality of delivered service such that reliance can justifiably be placed on this service. The service delivered by a system is its observed actual behavior as perceived by other system(s) interacting with this system's users. Each module also has an ideal specified behavior, where a service specification is an agreed description of the expected behavior. A system failure occurs when the actual behavior deviates from the specified behavior.

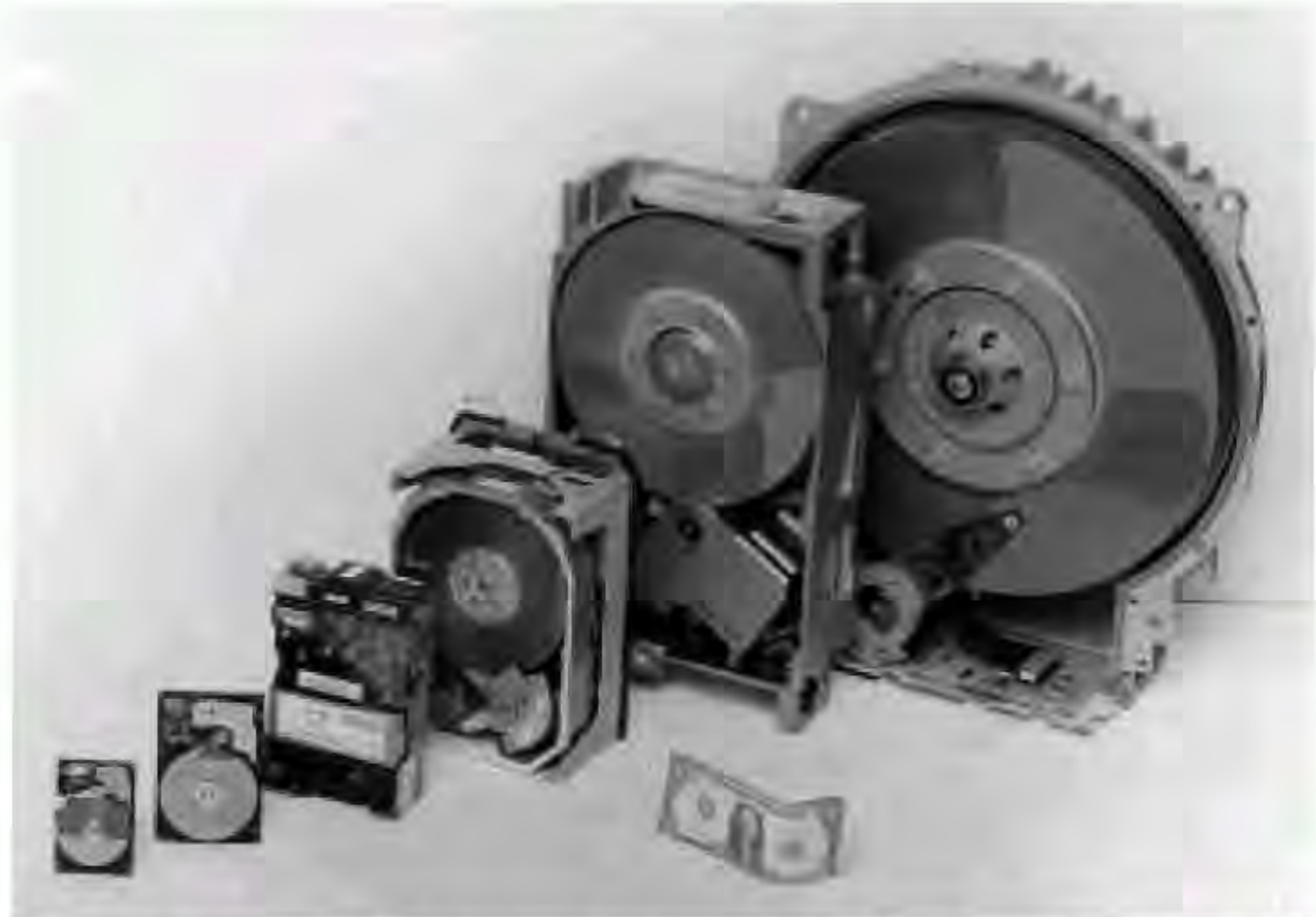


FIGURE 8.3 Six magnetic disks, varying in diameter from 14 inches down to 1.8 inches. The IBM microdrive, not shown, has a 1-inch diameter. The pictured disks were introduced over more than 15 years ago and hence are not intended to be representative of the best capacity of modern disks of these diameters. This photograph does, however, accurately portray their relative physical sizes. The widest disk is the DEC R81, containing four 14-inch diameter platters and storing 456 MB. It was manufactured in 1985. The 8-inch diameter disk comes from Fujitsu, and this 1984 disk stores 130 MB on six platters. The Micropolis RD53 has five 5.25-inch platters and stores 85 MB. The IBM 0361 also has five platters, but these are just 3.5 inches in diameter. This 1988 disk holds 320 MB. In 2004, the most dense 3.5-inch disk had 2 platters and holds 200 GB in the same space, yielding an increase in density of about 600 times! The Conner CP 2045 has two 2.5-inch platters containing 40 MB and was made in 1990. The smallest disk in this photograph is the Integral 1820. This single 1.8-inch platter contains 20 MB and was made in 1992. Figure 8.11 on page 585 shows a 10-inch drive that holds 340 MB.

Thus, you need a reference specification of expected behavior to be able to determine dependability. 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 to diagnose when a system oscillates between the two states; permanent failures are much easier to diagnose. This definition leads to two related terms: reliability and availability.

Characteristics	Seagate ST373453	Seagate ST3200822	Seagate ST94811A
Disk diameter (inches)	3.50	3.50	2.50
Formatted data capacity (GB)	73.4	200.0	40.0
Number of disk surfaces (heads)	8	4	2
Rotation speed (RPM)	15,000	7200	5400
Internal disk cache size (MB)	8	8	8
External interface, bandwidth (MB/sec)	Ultra320 SCSI, 320	Serial ATA, 150	ATA, 100
Sustained transfer rate (MB/sec)	57–86	32–58	34
Minimum seek (read/write) (ms)	0.2/0.4	1.0/1.2	1.5/2.0
Average seek read/write (ms)	3.6/3.9	8.5/9.5	12.0/14.0
Mean time to failure (MTTF) (hours)	1,200,000 @ 25°C	600,000 @ 25°C	330,000 @ 25°C
Warranty (years)	5	3	—
Nonrecoverable read errors per bits read	<1 per 10 ¹⁵	< 1 per 10 ¹⁴	< 1 per 10 ¹⁴
Temperature, vibration limits (operating)	5°–55°C, 400 Hz @ 0.5 G	0°–60°C, 350 Hz @ 0.5 G	5°–55°C, 400 Hz @ 1 G
Size: dimensions (in.), weight (pounds)	1.0" × 4.0" × 5.8", 1.9 lbs	1.0" × 4.0" × 5.8", 1.4 lbs	0.4" × 2.7" × 3.9", 0.2 lbs
Power: operating/idle/standby (watts)	20?/12/—	12/8/1	2.4/1.0/0.4
GB/cu. in., GB/watt	3 GB/cu.in., 4 GB/W	9 GB/cu.in., 16 GB/W	10 GB/cu.in., 17 GB/W
Price in 2004, \$/GB	≈ \$400, ≈ \$5/GB	≈ \$100, ≈ \$0.5/GB	≈ \$100, ≈ \$2.50/GB

FIGURE 8.4 Characteristics of three magnetic disks by a single manufacturer in 2004. The disks shown here either interface to SCSI, a standard I/O bus for many systems, or ATA, a standard I/O bus for PCs. The first disk is intended for file servers, the second for desktop PCs, and the last for laptop computers. Each disk has an 8 MB cache. The transfer rate from the cache is 3–6 times faster than the transfer rate from the disk surface. The much lower cost of the ATA 3.5-inch drive is primarily due to the hypercompetitive PC market, although there are differences in performance and reliability between it and the SCSI drive. The service life for these disks is 5 years, although Seagate offers a 5-year guarantee only on the SCSI drive, with a 1-year guarantee on the other two. Note that the quoted MTTF assumes nominal power and temperature. Disk lifetimes can be much shorter if temperature and vibration are not controlled. See the link to Seagate at www.seagate.com for more information on these drives.

Reliability is a measure of the continuous service accomplishment—or, equivalently, of the time to failure—from a reference point. Hence, the mean time to failure (MTTF) of disks in Figure 8.4 is a reliability measure. 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 a measure of the 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 quantifiable measures, rather than just synonyms for dependability.

What is the cause of failures? Figure 8.5 summarizes many papers that have collected data on reasons for computer systems and telecommunications systems to fail. Clearly, human operators are a significant source of failures.

small computer systems interface (SCSI) A bus used as a standard for I/O devices.

Operator	Software	Hardware	System	Year data collected
42%	25%	18%	Data center (Tandem)	1985
15%	55%	14%	Data center (Tandem)	1989
18%	44%	39%	Data center (DEC VAX)	1985
50%	20%	30%	Data center (DEC VAX)	1993
50%	14%	19%	U.S. public telephone network	1996
54%	7%	30%	U.S. public telephone network	2000
60%	25%	15%	Internet services	2002

FIGURE 8.5 Summary of studies of reasons for failures. Although it is difficult to collect data to determine if operators are the cause of errors, since operators often record the reasons for failures, these studies did capture that data. There were often other categories, such as environmental reasons for outages, but they were generally small. The top two rows come from a classic paper by Jim Gray [1990], which is still widely quoted almost 20 years after the data was collected. The next two rows are from a paper by Murphy and Gent who studied causes of outages in VAX systems over time (“Measuring system and software reliability using an automated data collection process,” *Quality and Reliability Engineering International* 11:5, September–October 1995, 341–53). The fifth and sixth rows are studies of FCC failure data about the U.S. public switched telephone network by Kuhn (“Sources of failure in the public switched telephone network,” *IEEE Computer* 30:4, April 1997, 31–36) and by Patty Enriquez. The most recent study of three Internet services is from Oppenheimer, Ganapath, and Patterson [2003].

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. A failure in 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, which applies primarily to hardware faults
3. *Fault forecasting:* predicting the presence and creation of faults, which applies to hardware and software faults

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 by people, software, and hardware.

redundant arrays of inexpensive disks (RAID) An organization of disks that uses an array of small and inexpensive disks so as to increase both performance and reliability.

RAID

Leveraging redundancy to improve the availability of disk storage is captured in the phrase **Redundant Arrays of Inexpensive Disks**, abbreviated **RAID**. At the time the term was coined, the alternative was large, expensive disks, such as the larger ones in Figure 8.3. The argument was that by replacing a few large disks

with many small disks, performance would improve because there would be more read heads, and there would be advantages in cost, power, and floor space since smaller disks are much more efficient per gigabyte than larger disks. Redundancy was needed because the many more smaller disks had lower reliability than a few large disks.

By having many small disks, the cost of extra redundancy to improve dependability is small relative to the large disks. Thus, dependability was more affordable if you constructed a redundant array of inexpensive disks. In retrospect, this was the key advantage.

How much redundancy do you need? Do you need extra information to find the faults? Does it matter how you organize the data and the extra check information on these disks? The paper that coined the term gave an evolutionary answer to these questions, starting with the simplest but most expensive solution. Figure 8.6 shows the evolution and example cost in number of extra check disks. To keep track of the evolution, the authors numbered the stages of RAID, and they are still used today.

No Redundancy (RAID 0)

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

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

Mirroring (RAID 1)

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

Error Detecting and Correcting Code (RAID 2)

RAID 2 borrows an error detection and correction scheme most often used for memories (see  [Appendix B](#)). Since RAID 2 has fallen into disuse, we’ll not describe it here.

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

mirroring Writing the identical data to multiple disks to increase data availability.

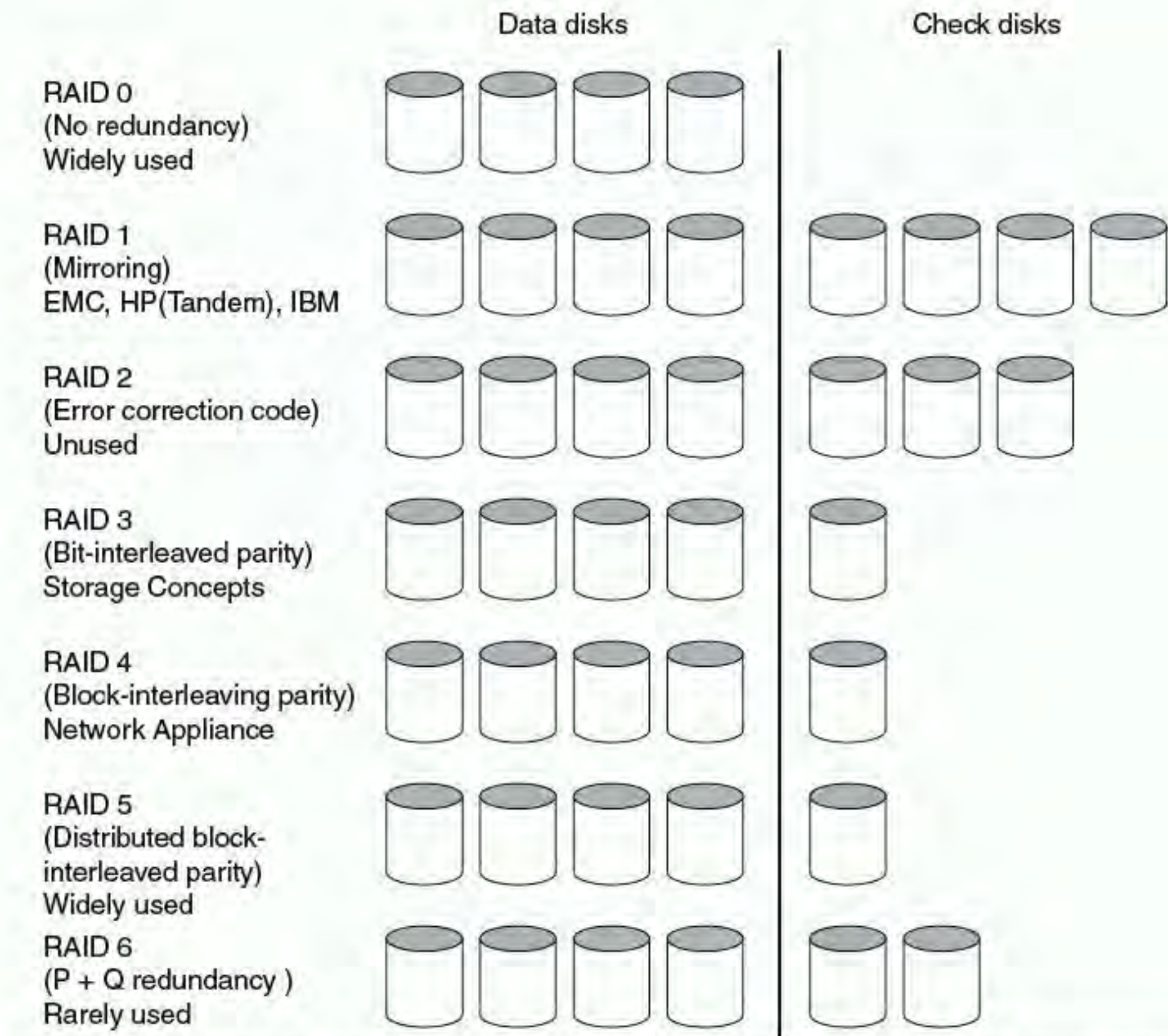


FIGURE 8.6 RAID for an example of four data disks showing extra check disks per RAID level and companies that use each level. Figures 8.7 and 8.8 explain the difference between RAID 3, RAID 4, and RAID 5.

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

Bit-Interleaved Parity (RAID 3)

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

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

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

Block-Interleaved Parity (RAID 4)

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

In RAID 3, every access went to all disks. However, some applications prefer smaller accesses, allowing independent accesses to occur in parallel. That is the purpose of the RAID levels 4 to 6. Since error detection information in each sector is checked on reads to see if data are correct, such “small reads” to each disk can occur independently as long as the minimum access is one sector. In the RAID context, a small access goes to just one disk in a protection group while a large access goes to all the disks in a protection group.

Writes are another matter. It would seem that each small write would demand that all other disks be accessed to read the rest of the information needed to recalculate the new parity, as in Figure 8.7. A “small write” would require reading the old data and old parity, adding the new information, and then writing the new parity to the parity disk and the new data to the data disk.

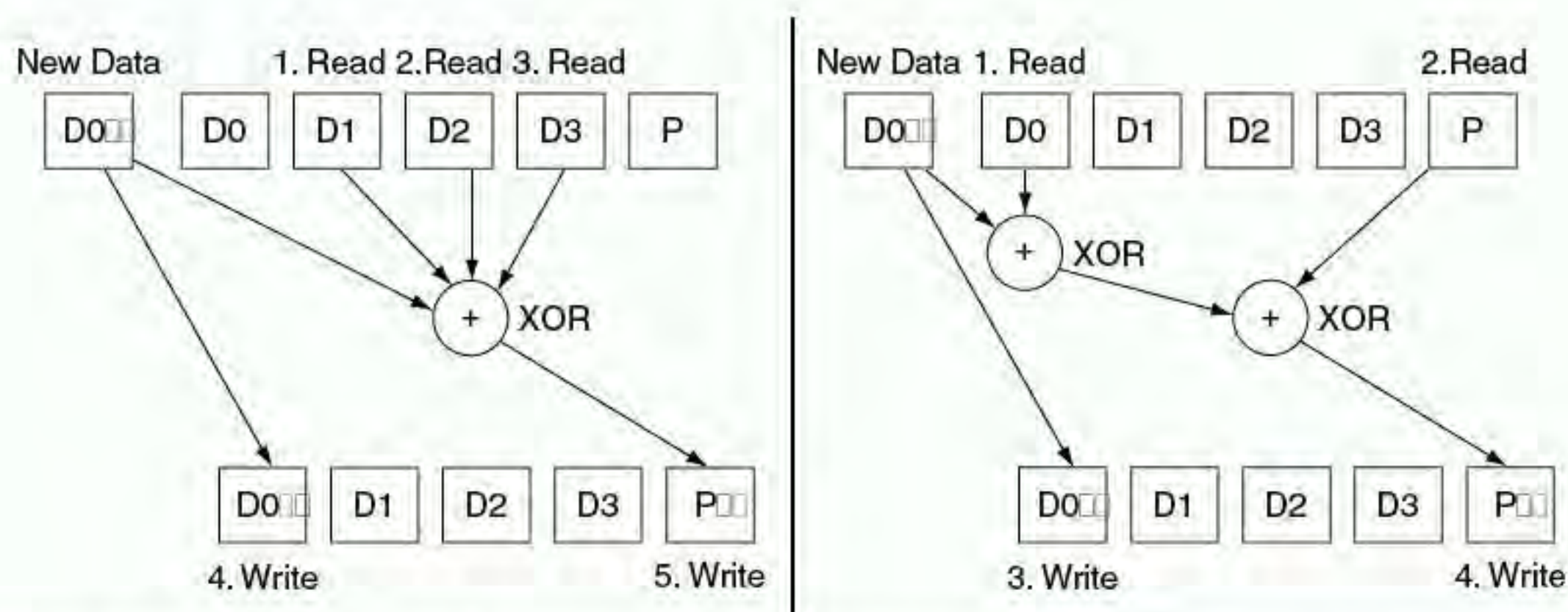


FIGURE 8.7 Small write update on RAID 3 versus RAID 4. This optimization for small writes reduces the number of disk accesses as well as the number of disks occupied. This figure assumes we have four blocks of data and one block of parity. The straightforward RAID 3 parity calculation in the left of the figure reads blocks D1, D2, and D3 before adding block D0' to calculate the new parity P'. (In case you were wondering, the new data D0' comes directly from the CPU, so disks are not involved in reading it.) The RAID 4 shortcut on the right reads the old value D0 and compares it to the new value D0' to see which bits will change. You then read the old parity P and then change the corresponding bits to form P'. The logical function exclusive OR does exactly what we want. This example replaces three disk reads (D1, D2, D3) and two disk writes (D0', P') involving all the disks for two disk reads (D0, P) and two disk writes (D0', P'), which involve just two disks. Increasing the size of the parity group increases the savings of the shortcut. RAID 5 uses the same shortcut.

The key insight to reduce this overhead is that parity is simply a sum of information; by watching which bits change when we write the new information, we need only change the corresponding bits on the parity disk. Figure 8.7 shows the shortcut. We must read the old data from the disk being written, compare old data to the new data to see which bits change, read the old parity, change the corresponding bits, then write the new data and new parity. Thus, the small write involves four disk accesses to two disks instead of accessing all disks. This organization is RAID 4.

Distributed Block-Interleaved Parity (RAID 5)

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

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

Figure 8.8 shows how data are distributed in RAID 4 versus RAID 5. As the organization on the right shows, in RAID 5 the parity associated with each row of data blocks is no longer restricted to a single disk. This organization allows multiple writes to occur simultaneously as long as the parity blocks are not located to the same disk. For example, a write to block 8 on the right must also access its parity block P2, thereby occupying the first and third disks. A second write to block 5 on the right, implying an update to its parity block P1, accesses the second and fourth disks and thus could occur concurrently with the write to block 8. Those same writes to the organization on the left result in changes to blocks P1 and P2, both on the fifth disk, which is a bottleneck.

P + Q Redundancy (RAID 6)

Parity-based schemes protect against a single self-identifying failure. When a single failure correction is not sufficient, parity can be generalized to have a second calculation over the data and another check disk of information. This second check block allows recovery from a second failure. Thus, the storage overhead is twice that of RAID 5. The small write shortcut of Figure 8.7 works as well, except now there are six disk accesses instead of four to update both P and Q information.

RAID Summary

RAID 1 and RAID 5 are widely used in servers; one estimate is 80% of disks in servers are found in some RAID system.

One weakness of the RAID systems is repair. First, to avoid making the data unavailable during repair, the array must be designed to allow the failed disks to

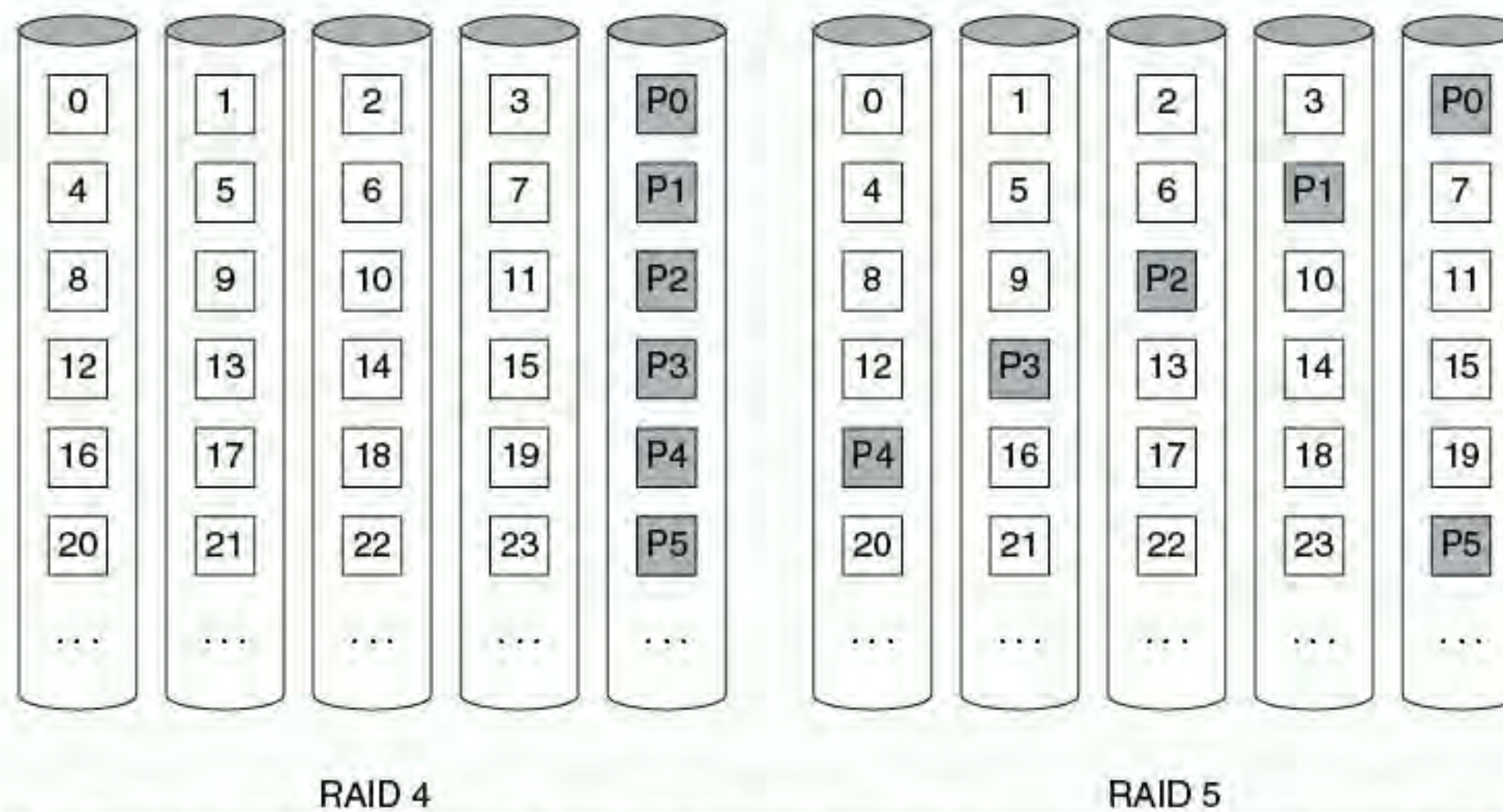


FIGURE 8.8 Block-interleaved parity (RAID 4) versus distributed block-interleaved parity (RAID 5). By distributing parity blocks to all disks, some small writes can be performed in parallel.

be replaced without having to turn off the system. RAIDs have enough redundancy to allow continuous operation, but **hot swapping** disks places demands on the physical and electrical design of the array and the disk interfaces. Second, another failure could occur during repair, so the repair time affects the chances of losing data: the longer the repair time, the greater the chances of another failure that will lose data. Rather than having to wait for the operator to bring in a good disk, some systems include **standby spares** so that the data can be reconstructed immediately upon discovery of the failure. The operator can then replace the failed disks in a more leisurely fashion. Third, although disk manufacturers quote very high MTTF for their products, those numbers are under nominal conditions. If a particular disk array has been subject to temperature cycles due to, say, the failure of the air conditioning system, or to shaking due to a poor rack design, construction, or installation, the failure rates will be much higher. The calculation of RAID reliability assumes independence between disk failures, but disk failures could be correlated because such damage due to the environment would likely happen to all the disks in the array. Finally, a human operator ultimately determines which disks to remove. As Figure 8.5 shows, operators are only human, so they occasionally remove the good disk instead of the broken disk, leading to an unrecoverable disk failure.

Although RAID 6 is rarely used today, a cautious operator might want its extra redundancy to protect against expected hardware failures plus a safety margin to protect against human error and correlated failures due to problems with the environment.

hot swapping Replacing a hardware component while the system is running.

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

Check Yourself

Which of the following are true about dependability?

1. If a system is up, then all its components are accomplishing their expected service.
2. Availability is a quantitative measure of the percentage of time a system is accomplishing its expected service.
3. Reliability is a quantitative measure of continuous service accomplishment by a system.
4. The major source of outages today is software.

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

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

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



Networks

Networks are growing in popularity over time, and unlike other I/O devices, there are many books and courses on them. For readers who have not taken courses or read books on networking, Section 8.3 on the  CD gives a quick overview of the topics and terminology, including internetworking, the OSI model, protocol families such as TCP/IP, long-haul networks such as ATM, local area networks such as Ethernet, and wireless networks such as IEEE 802.11.

8.4**Buses and Other Connections between Processors, Memory, and I/O Devices**

In a computer system, the various subsystems must have interfaces to one another. For example, the memory and processor need to communicate, as do the processor and the I/O devices. For many years, this has been done with a *bus*. A bus is a shared communication link, which uses one set of wires to connect multiple subsystems. The two major advantages of the bus organization are versatility and low cost. By defining a single connection scheme, new devices can easily be added, and peripherals can even be moved between computer systems that use the same kind of bus. Furthermore, buses are cost-effective because a single set of wires is shared in multiple ways.

The major disadvantage of a bus is that it creates a communication bottleneck, possibly limiting the maximum I/O throughput. When I/O must pass through a single bus, the bandwidth of that bus limits the maximum I/O throughput. Designing a bus system capable of meeting the demands of the processor as well as connecting large numbers of I/O devices to the machine presents a major challenge.

One reason bus design is so difficult is that the maximum bus speed is largely limited by physical factors: the length of the bus and the number of devices. These physical limits prevent us from running the bus arbitrarily fast. In addition, the need to support a range of devices with widely varying latencies and data transfer rates also makes bus design challenging.

As it becomes difficult to run many parallel wires at high speed due to clock skew and reflection, the industry is in transition from parallel shared buses to high-speed serial point-to-point interconnections with switches. Thus, such networks are gradually replacing buses in our systems.

As a result of this transition, this section has been revised in this edition to emphasize the general problem of connecting I/O devices, processors, and memory rather than focus exclusively on buses.

Bus Basics

Classically, a bus generally contains a set of control lines and a set of data lines. The control lines are used to signal requests and acknowledgments, and to indicate what type of information is on the data lines. The data lines of the bus carry information between the source and the destination. This information may consist of data, complex commands, or addresses. For example, if a disk wants to write some data into memory from a disk sector, the data lines will be used to indicate the address in memory in which to place the data as well as to carry the

bus transaction A sequence of bus operations that includes a request and may include a response, either of which may carry data. A transaction is initiated by a single request and may take many individual bus operations.

processor-memory bus A bus that connects processor and memory and that is short, generally high speed, and matched to the memory system so as to maximize memory-processor bandwidth.

backplane bus A bus that is designed to allow processors, memory, and I/O devices to coexist on a single bus.

synchronous bus A bus that includes a clock in the control lines and a fixed protocol for communicating that is relative to the clock.

asynchronous bus A bus that uses a handshaking protocol for coordinating usage rather than a clock; can accommodate a wide variety of devices of differing speeds.

actual data from the disk. The control lines will be used to indicate what type of information is contained on the data lines of the bus at each point in the transfer. Some buses have two sets of signal lines to separately communicate both data and address in a single bus transmission. In either case, the control lines are used to indicate what the bus contains and to implement the bus protocol. And because the bus is shared, we also need a protocol to decide who uses it next; we will discuss this problem shortly.

Let's consider a typical **bus transaction**. A bus transaction includes two parts: sending the address and receiving or sending the data. Bus transactions are typically defined by what they do to memory. A *read* transaction transfers data *from* memory (to either the processor or an I/O device), and a *write* transaction writes data *to* the memory. Clearly, this terminology is confusing. To avoid this, we'll try to use the terms *input* and *output*, which are always defined from the perspective of the processor: an input operation is inputting data from the device to memory, where the processor can read it, and an output operation is outputting data to a device from memory where the processor wrote it.

Buses are traditionally classified as **processor-memory buses** or *I/O buses*. Processor-memory buses are short, generally high speed, and matched to the memory system so as to maximize memory-processor bandwidth. I/O buses, by contrast, can be lengthy, can have many types of devices connected to them, and often have a wide range in the data bandwidth of the devices connected to them. I/O buses do not typically interface directly to the memory but use either a processor-memory or a **backplane bus** to connect to memory. Other buses with different characteristics have emerged for special functions, such as graphics buses.

The I/O bus serves as a way of expanding the machine and connecting new peripherals. To make this easier, the computer industry has developed several standards. The standards serve as a specification for the computer manufacturer and for the peripheral manufacturer. A standard ensures the computer designer that peripherals will be available for a new machine, and it ensures the peripheral builder that users will be able to hook up their new equipment. Figure 8.9 summarizes the key characteristics of the two dominant I/O bus standards: Firewire and USB. They connect a variety of devices to the desktop computer, from keyboards to cameras to disks.

The two basic schemes for communication on the bus are **synchronous** and **asynchronous**. If a bus is synchronous, it includes a clock in the control lines and a fixed protocol for communicating that is relative to the clock. For example, for a processor-memory bus performing a read from memory, we might have a protocol that transmits the address and read command on the first clock cycle, using the control lines to indicate the type of request. The memory might then be required to respond with the data word on the fifth clock. This type of protocol can be implemented easily in a small finite state machine. Because the protocol is predetermined and involves little logic, the bus can run very fast and the interface logic will be small.

Characteristic	Firewire (1394)	USB 2.0
Bus type	I/O	I/O
Basic data bus width (signals)	4	2
Clocking	asynchronous	asynchronous
Theoretical peak bandwidth	50 MB/sec (Firewire 400) or 100 MB/sec (Firewire 800)	0.2 MB/sec (low speed), 1.5 MB/sec (full speed), or 60 MB/sec (high speed)
Hot pluggable	yes	yes
Maximum number of devices	63	127
Maximum bus length (copper wire)	4.5 meters	5 meters
Standard name	IEEE 1394, 1394b	USE Implementors Forum

FIGURE 8.9 Key characteristics of two dominant I/O bus standards.

Synchronous buses have two major disadvantages, however. First, every device on the bus must run at the same clock rate. Second, because of clock skew problems, synchronous buses cannot be long if they are fast (see [Appendix B](#) for a discussion of clock skew). Processor-memory buses are often synchronous because the devices communicating are close, small in number, and prepared to operate at high clock rates.

An asynchronous bus is not clocked. Because it is not clocked, an asynchronous bus can accommodate a wide variety of devices, and the bus can be lengthened without worrying about clock skew or synchronization problems. Both Firewire and USB 2.0 are asynchronous. To coordinate the transmission of data between sender and receiver, an asynchronous bus uses a **handshaking protocol**. A handshaking protocol consists of a series of steps in which the sender and receiver proceed to the next step only when both parties agree. The protocol is implemented with an additional set of control lines.

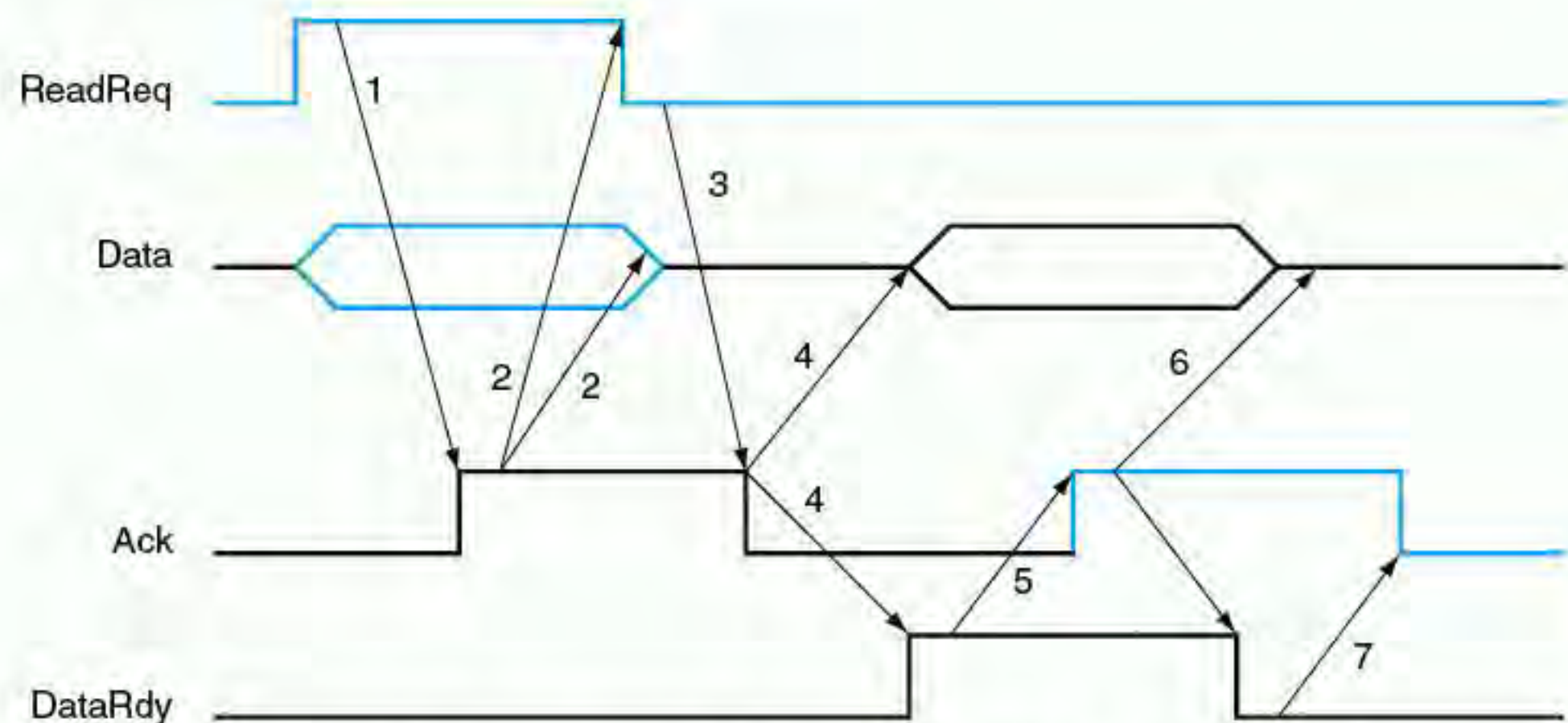
A simple example will illustrate how asynchronous buses work. Let's consider a device requesting a word of data from the memory system. Assume that there are three control lines:

1. *ReadReq*: Used to indicate a read request for memory. The address is put on the data lines at the same time.
2. *DataRdy*: Used to indicate that the data word is now ready on the data lines. In an output transaction, the memory will assert this signal since it is providing the data. In an input transaction, an I/O device would assert this signal, since it would provide data. In either case, the data is placed on the data lines at the same time.
3. *Ack*: Used to acknowledge the *ReadReq* or the *DataRdy* signal of the other party.

handshaking protocol A series of steps used to coordinate asynchronous bus transfers in which the sender and receiver proceed to the next step only when both parties agree that the current step has been completed.

In an asynchronous protocol, the control signals ReadReq and DataRdy are asserted until the other party (the memory or the device) indicates that the control lines have been seen and the data lines have been read; this indication is made by asserting the Ack line. This complete process is called *handshaking*. Figure 8.10 shows how such a protocol operates by depicting the steps in the communication.

Although much of the bandwidth of a bus is decided by the choice of a synchronous or asynchronous protocol and the timing characteristics of the bus, several other factors affect the bandwidth that can be attained by a single transfer. The most important of these are the data bus width, and whether it supports block transfers or it transfers a word at a time.



The steps in the protocol begin immediately after the device signals a request by raising ReadReq and putting the address on the Data lines:

1. When memory sees the ReadReq line, it reads the address from the data bus and raises Ack to indicate it has been seen.
2. I/O device sees the Ack line high and releases the ReadReq and data lines.
3. Memory sees that ReadReq is low and drops the Ack line to acknowledge the ReadReq signal.
4. This step starts when the memory has the data ready. It places the data from the read request on the data lines and raises DataRdy.
5. The I/O device sees DataRdy, reads the data from the bus, and signals that it has the data by raising Ack.
6. The memory sees the Ack signal, drops DataRdy, and releases the data lines.
7. Finally, the I/O device, seeing DataRdy go low, drops the Ack line, which indicates that the transmission is completed.

A new bus transaction can now begin.

FIGURE 8.10 The asynchronous handshaking protocol consists of seven steps to read a word from memory and receive it in an I/O device. The signals in color are those asserted by the I/O device, while the memory asserts the signals shown in black. The arrows label the seven steps and the event that triggers each step. The symbol showing two lines (high and low) at the same time on the data lines indicates that the data lines have valid data at this point. (The symbol indicates that the data is valid, but the value is not known.)

Elaboration: Another method for increasing the effective bus bandwidth is to release the bus when it is not being used for transmitting information. This type of protocol is called a **split transaction protocol**. The advantage of such a protocol is that, by freeing the bus during the time data is not being transmitted, the protocol allows another requestor to use the bus. This can improve the effective bus bandwidth for the entire system if the memory is sophisticated enough to handle multiple overlapping transactions. Multiprocessors sharing a memory bus may use split transaction protocols.

split transaction protocol A protocol in which the bus is released during a bus transaction while the requester is waiting for the data to be transmitted, which frees the bus for access by another requester.

The Buses and Networks of the Pentium 4

Figure 8.11 shows the I/O system of a PC based on the Pentium 4. The processor connects to peripherals via two main chips. The chip next to the processor is the memory controller hub, commonly called the *north bridge*, and the one connected to it is the I/O controller hub, called the *south bridge*.

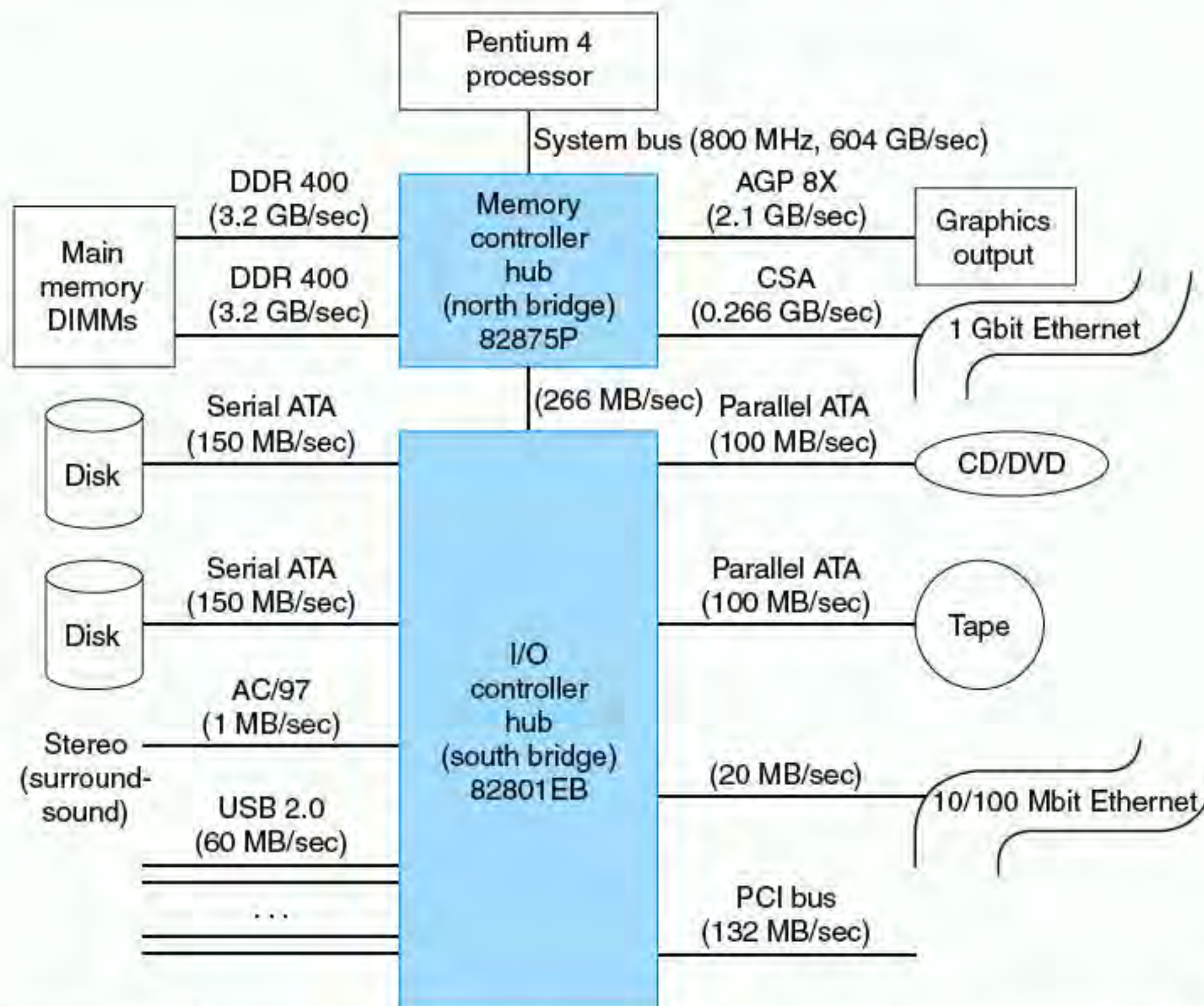


FIGURE 8.11 Organization of the I/O system on a Pentium 4 PC using the Intel 875 chip set. Note that the maximum transfer rate between the north bridge (memory hub) and south bridge (I/O hub) is 266 MB/sec, which is why Intel put the AGP bus and Gigabit Ethernet on the north bridge.

The north bridge is basically a DMA controller, connecting the processor to memory, the AGP graphic bus, and the south bridge chip. The south bridge connects the north bridge to a cornucopia of I/O buses. Intel and others offer a wide variety of these chip sets to connect the Pentium 4 to the outside world. To give a flavor of the options, Figure 8.12 shows two of the chip sets.

As Moore's law continues, an increasing number of I/O controllers that were formerly available as optional cards that connected to I/O buses have been co-opted into these chip sets. For example, the south bridge chip of the Intel 875 includes a striping RAID controller, and the north bridge chip of the Intel 845GL includes a graphics controller.

	875P chip set	845GL chip set
Target segment	Performance PC	Value PC
System bus (64 bit)	800/533 MHz	400 MHz
Memory controller hub ("north bridge")		
Package size, pins	42.5 × 42.5 mm, 1005	37.5 × 37.5 mm, 760
Memory speed	DDR 400/333/266 SDRAM	DDR 266/200, PC133 SDRAM
Memory buses, widths	2 × 72	1 × 64
Number of DIMMs, DRAM Mbit support	4, 128/256/512 Mbits	2, 128/256/512 Mbits
Maximum memory capacity	4 GB	2 GB
Memory error correction available?	yes	no
AGP graphics bus, speed	yes, 8X or 4X	no
Graphics controller	external	Internal (Extreme Graphics)
CSA Gigabit Ethernet interface	yes	no
South bridge interface speed (8 bit)	266 MHz	266 MHz
I/O controller hub ("south bridge")		
Package size, pins	31 × 31 mm, 460	31 × 31 mm, 421
PCI bus: width, speed, masters	32-bit, 33 MHz, 6 masters	32-bit, 33 MHz, 6 masters
Ethernet MAC controller, interface	100/10 Mbit	100/10 Mbit
USB 2.0 ports, controllers	8, 4	6, 3
ATA 100 ports	2	2
Serial ATA 150 controller, ports	yes, 2	no
RAID 0 controller	yes	no
AC-97 audio controller, interface	yes	yes
I/O management	SMbus 2.0, GPIO	SMbus 2.0, GPIO

FIGURE 8.12 Two Pentium 4 I/O chip sets from Intel. The 845GL north bridge uses many fewer pins than the 875 by having just one memory bus and by omitting the AGP bus and the Gigabit Ethernet interface. Note that the serial nature of USB and Serial ATA means that two more USB ports and two more Serial ATA ports need just 39 more pins in the south bridge of the 875 versus the 845GL chip sets.

These two chips demonstrate the gradual evolution from parallel shared buses to high-speed serial point-to-point interconnections with switches via the past and future versions of ATA and PCI.

Serial ATA is a serial successor to the parallel ATA bus used by magnetic and optical disks in PCs. The first generation transfers at 150 MB/sec compared to the 100 MB/sec of parallel ATA-100 bus. Its distance is 1 meter, twice the maximum length of ATA-100. It uses just 7 wires, with one 2-wire data channel in each direction, compared to 80 for ATA-100.

The south bridge in Figure 8.11 demonstrates the transitory period between parallel buses and serial networks by providing both parallel and serial ATA buses.

PCI Express is a serial successor to the popular PCI bus. Rather than 32–64 shared wires operating at 33 MHz–133 MHz with a peak bandwidth of 132–1064 MB/sec, PCI Express uses just 4 wires in each direction operating at 625 MHz to offer 300 MB/sec per direction. The bandwidth per pin of PCI Express is 5–10 times its predecessors. A computer can then afford to have several PCI Express interfaces to get even higher bandwidth.

Although the chips in Figure 8.11 only show the parallel PCI bus, Intel plans to replace the AGP graphics bus and the bus between the north bridge and the south bridge with PCI Express in the next generation of these chips.

Buses and networks provide electrical interconnection among I/O devices, processors, and memory, and also define the lowest-level protocol for communication. Above this basic level, we must define hardware and software protocols for controlling data transfers between I/O devices and memory, and for the processor to specify commands to the I/O devices. These topics are covered in the next section.

Both networks and buses connect components together. Which of the following are true about them?

1. Networks and I/O buses are almost always standardized.
2. Shared media networks and multimaster buses need an arbitration scheme.
3. Local area networks and processor-memory buses are almost always synchronous.
4. High-performance networks and buses use similar techniques compared to their lower-performance alternatives: they are wider, send many words per transaction, and have separate address and data lines.

**Check
Yourself**

8.5**Interfacing I/O Devices to the Processor, Memory, and Operating System**

A bus or network protocol defines how a word or block of data should be communicated on a set of wires. This still leaves several other tasks that must be performed to actually cause data to be transferred from a device and into the memory address space of some user program. This section focuses on these tasks and will answer such questions as the following:

- How is a user I/O request transformed into a device command and communicated to the device?
- How is data actually transferred to or from a memory location?
- What is the role of the operating system?

As we will see in answering these questions, the operating system plays a major role in handling I/O, acting as the interface between the hardware and the program that requests I/O.

The responsibilities of the operating system arise from three characteristics of I/O systems:

1. Multiple programs using the processor share the I/O system.
2. I/O systems often use interrupts (externally generated exceptions) to communicate information about I/O operations. Because interrupts cause a transfer to kernel or supervisor mode, they must be handled by the operating system (OS).
3. The low-level control of an I/O device is complex because it requires managing a set of concurrent events and because the requirements for correct device control are often very detailed.

**Hardware
Software
Interface**

The three characteristics of I/O systems above lead to several different functions the OS must provide:

- The OS guarantees that a user's program accesses only the portions of an I/O device to which the user has rights. For example, the OS must not allow a program to read or write a file on disk if the owner of the file has not granted access to this program. In a system with shared I/O devices, protection could not be provided if user programs could perform I/O directly.
- The OS provides abstractions for accessing devices by supplying routines that handle low-level device operations.

- The OS handles the interrupts generated by I/O devices, just as it handles the exceptions generated by a program.
- The OS tries to provide equitable access to the shared I/O resources, as well as schedule accesses in order to enhance system throughput.

To perform these functions on behalf of user programs, the operating system must be able to communicate with the I/O devices and to prevent the user program from communicating with the I/O devices directly. Three types of communication are required:

1. The OS must be able to give commands to the I/O devices. These commands include not only operations like read and write, but also other operations to be done on the device, such as a disk seek.
2. The device must be able to notify the OS when the I/O device has completed an operation or has encountered an error. For example, when a disk completes a seek, it will notify the OS.
3. Data must be transferred between memory and an I/O device. For example, the block being read on a disk read must be moved from disk to memory.

In the next few sections, we will see how these communications are performed.

Giving Commands to I/O Devices

To give a command to an I/O device, the processor must be able to address the device and to supply one or more command words. Two methods are used to address the device: memory-mapped I/O and special I/O instructions. In **memory-mapped I/O**, portions of the address space are assigned to I/O devices. Reads and writes to those addresses are interpreted as commands to the I/O device.

For example, a write operation can be used to send data to an I/O device where the data will be interpreted as a command. When the processor places the address and data on the memory bus, the memory system ignores the operation because the address indicates a portion of the memory space used for I/O. The device controller, however, sees the operation, records the data, and transmits it to the device as a command. User programs are prevented from issuing I/O operations directly because the OS does not provide access to the address space assigned to the I/O devices and thus the addresses are protected by the address translation. Memory-mapped I/O can also be used to transmit data by writing or reading to select addresses. The device uses the address to determine the type of command, and the data may be provided by a write or obtained by a read. In any event, the address encodes both the device identity and the type of transmission between processor and device.

memory-mapped I/O An I/O scheme in which portions of address space are assigned to I/O devices and reads and writes to those addresses are interpreted as commands to the I/O device.

Actually performing a read or write of data to fulfill a program request usually requires several separate I/O operations. Furthermore, the processor may have to interrogate the status of the device between individual commands to determine whether the command completed successfully. For example, a simple printer has two I/O device registers—one for status information and one for data to be printed. The Status register contains a *done bit*, set by the printer when it has printed a character, and an *error bit*, indicating that the printer is jammed or out of paper. Each byte of data to be printed is put into the Data register. The processor must then wait until the printer sets the done bit before it can place another character in the buffer. The processor must also check the error bit to determine if a problem has occurred. Each of these operations requires a separate I/O device access.

I/O instructions A dedicated instruction that is used to give a command to an I/O device and that specifies both the device number and the command word (or the location of the command word in memory).

Elaboration: The alternative to memory-mapped I/O is to use dedicated **I/O instructions** in the processor. These I/O instructions can specify both the device number and the command word (or the location of the command word in memory). The processor communicates the device address via a set of wires normally included as part of the I/O bus. The actual command can be transmitted over the data lines in the bus. Examples of computers with I/O instructions are the Intel IA-32 and the IBM 370 computers. By making the I/O instructions illegal to execute when not in kernel or supervisor mode, user programs can be prevented from accessing the devices directly.

polling The process of periodically checking the status of an I/O device to determine the need to service the device.

Communicating with the Processor

The process of periodically checking status bits to see if it is time for the next I/O operation, as in the previous example, is called **polling**. Polling is the simplest way for an I/O device to communicate with the processor. The I/O device simply puts the information in a Status register, and the processor must come and get the information. The processor is totally in control and does all the work.

Polling can be used in several different ways. Real-time embedded applications poll the I/O devices since the I/O rates are predetermined and it makes I/O overhead more predictable, which is helpful for real time. As we will see, this allows polling to be used even when the I/O rate is somewhat higher.

The disadvantage of polling is that it can waste a lot of processor time because processors are so much faster than I/O devices. The processor may read the Status register many times, only to find that the device has not yet completed a comparatively slow I/O operation, or that the mouse has not budged since the last time it was polled. When the device completes an operation, we must still read the status to determine whether it was successful.

The overhead in a polling interface was recognized long ago, leading to the invention of interrupts to notify the processor when an I/O device requires attention from the processor. **Interrupt-driven I/O**, which is used by almost all systems

interrupt-driven I/O An I/O scheme that employs interrupts to indicate to the processor that an I/O device needs attention.

for at least some devices, employs I/O interrupts to indicate to the processor that an I/O device needs attention. When a device wants to notify the processor that it has completed some operation or needs attention, it causes the processor to be interrupted.

An I/O interrupt is just like the exceptions we saw in Chapters 5, 6, and 7, with two important exceptions:

1. An I/O interrupt is asynchronous with respect to the instruction execution. That is, the interrupt is not associated with any instruction and does not prevent the instruction completion. This is very different from either page fault exceptions or exceptions such as arithmetic overflow. Our control unit need only check for a pending I/O interrupt at the time it starts a new instruction.
2. In addition to the fact that an I/O interrupt has occurred, we would like to convey further information such as the identity of the device generating the interrupt. Furthermore, the interrupts represent devices that may have different priorities and whose interrupt requests have different urgencies associated with them.

To communicate information to the processor, such as the identity of the device raising the interrupt, a system can use either vectored interrupts or an exception Cause register. When the processor recognizes the interrupt, the device can send either the vector address or a status field to place in the Cause register. As a result, when the OS gets control, it knows the identity of the device that caused the interrupt and can immediately interrogate the device. An interrupt mechanism eliminates the need for the processor to poll the device and instead allows the processor to focus on executing programs.

Interrupt Priority Levels

To deal with the different priorities of the I/O devices, most interrupt mechanisms have several levels of priority: UNIX operating systems use four to six levels. These priorities indicate the order in which the processor should process interrupts. Both internally generated exceptions and external I/O interrupts have priorities; typically, I/O interrupts have lower priority than internal exceptions. There may be multiple I/O interrupt priorities, with high-speed devices associated with the higher priorities.

To support priority levels for interrupts, MIPS provides the primitives that let the operating system implement the policy, similar to how MIPS handles TLB misses. Figure 8.13 shows the key registers, and Section A.7 in  [Appendix A](#) gives more details.

The Status register determines who can interrupt the computer. If the interrupt enable bit is 0, then none can interrupt. A more refined blocking of interrupts is available in the interrupt mask field. There is a bit in the mask corresponding to

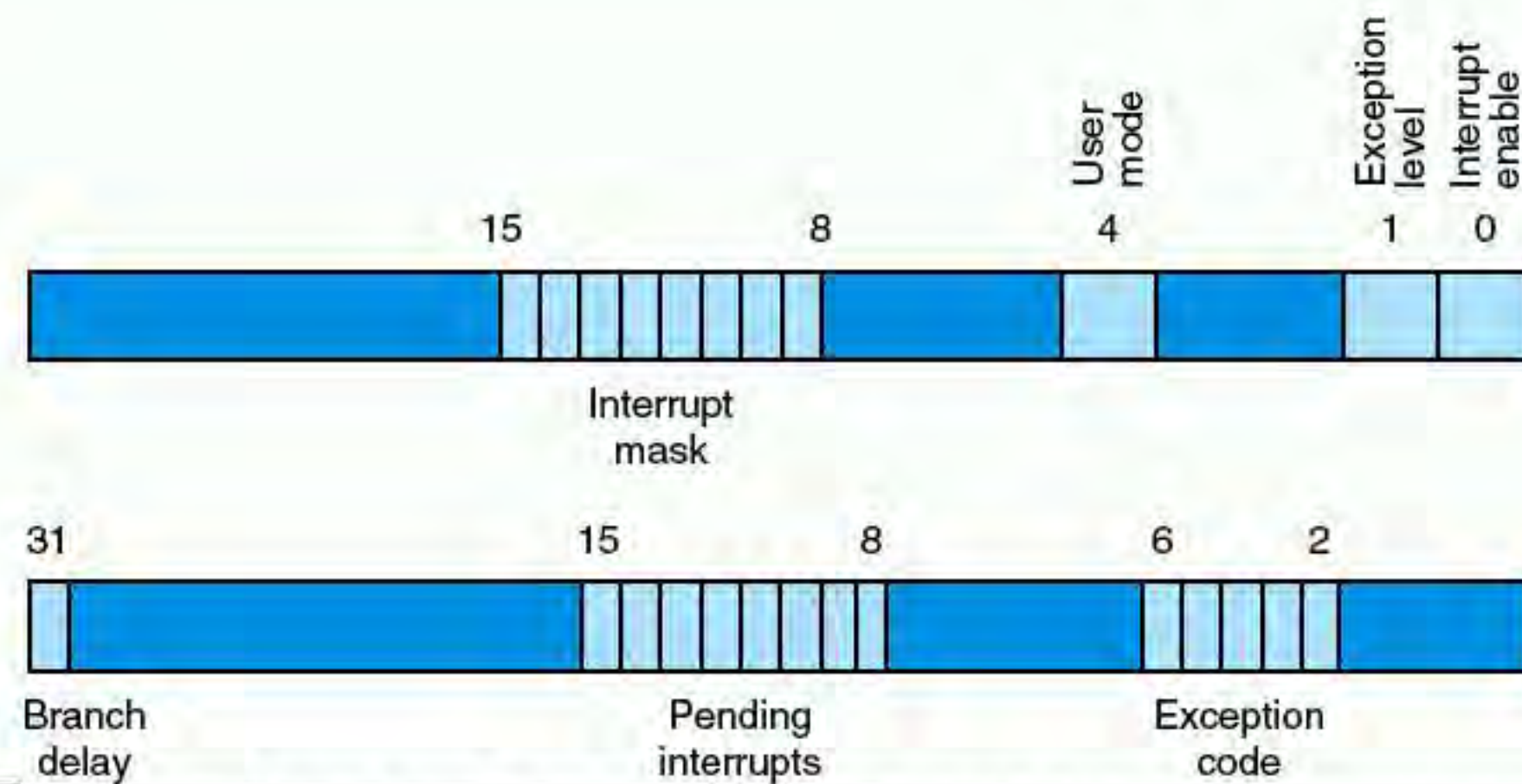


FIGURE 8.13 The Cause and Status registers. This version of the Cause register corresponds to the MIPS-32 architecture. The earlier MIPS I architecture had three nested sets of kernel/user and interrupt enable bits to support nested interrupts. Section A.7 in [Appendix A](#) has more details about these registers.

each bit in the pending interrupt field of the Cause register. To enable the corresponding interrupt, there must be a 1 in the mask field at that bit position. Once an interrupt occurs, the operating system can find the reason in the exception code field of the Status register: 0 means an interrupt occurred, with other values for the exceptions mentioned in Chapter 7.

Here are the steps that must occur in handling an interrupt:

1. Logically AND the pending interrupt field and the interrupt mask field to see which enabled interrupts could be the culprit. Copies are made of these two registers using the `mfc0` instruction.
2. Select the higher priority of these interrupts. The software convention is that the leftmost is the highest priority.
3. Save the interrupt mask field of the Status register.
4. Change the interrupt mask field to disable all interrupts of equal or lower priority.
5. Save the processor state needed to handle the interrupt.
6. To allow higher-priority interrupts, set the interrupt enable bit of the Cause register to 1.
7. Call the appropriate interrupt routine.
8. Before restoring state, set the interrupt enable bit of the Cause register to 0. This allows you to restore the interrupt mask field.

🔗 **Appendix A** shows an exception handler for a simple I/O task on pages A-36 to A-37.

How do the *interrupt priority levels* (IPL) correspond to these mechanisms? The IPL is an operating system invention. It is stored in the memory of the process, and every process is given an IPL. At the lowest IPL, all interrupts are permitted. Conversely, at the highest IPL, all interrupts are blocked. Raising and lowering the IPL involves changes to the interrupt mask field of the Status register.

Elaboration: The two least significant bits of the pending interrupt and interrupt mask fields are for software interrupts, which are lower priority. These are typically used by higher-priority interrupts to leave work for lower-priority interrupts to do once the immediate reason for the interrupt is handled. Once the higher-priority interrupt is finished, the lower-priority tasks will be noticed and handled.

Transferring the Data between a Device and Memory

We have seen two different methods that enable a device to communicate with the processor. These two techniques—polling and I/O interrupts—form the basis for two methods of implementing the transfer of data between the I/O device and memory. Both these techniques work best with lower-bandwidth devices, where we are more interested in reducing the cost of the device controller and interface than in providing a high-bandwidth transfer. Both polling and interrupt-driven transfers put the burden of moving data and managing the transfer on the processor. After looking at these two schemes, we will examine a scheme more suitable for higher-performance devices or collections of devices.

We can use the processor to transfer data between a device and memory based on polling. In real-time applications, the processor loads data from I/O device registers and stores them into memory.

An alternative mechanism is to make the transfer of data interrupt driven. In this case, the OS would still transfer data in small numbers of bytes from or to the device. But because the I/O operation is interrupt driven, the OS simply works on other tasks while data is being read from or written to the device. When the OS recognizes an interrupt from the device, it reads the status to check for errors. If there are none, the OS can supply the next piece of data, for example, by a sequence of memory-mapped writes. When the last byte of an I/O request has been transmitted and the I/O operation is completed, the OS can inform the program. The processor and OS do all the work in this process, accessing the device and memory for each data item transferred.

Interrupt-driven I/O relieves the processor from having to wait for every I/O event, although if we used this method for transferring data from or to a hard disk, the overhead could still be intolerable, since it could consume a large fraction of the processor when the disk was transferring. For high-bandwidth devices like hard disks, the transfers consist primarily of relatively large blocks of data

direct memory access (DMA)

A mechanism that provides a device controller the ability to transfer data directly to or from the memory without involving the processor.

bus master A unit on the bus that can initiate bus requests.

(hundreds to thousands of bytes). Thus, computer designers invented a mechanism for offloading the processor and having the device controller transfer data directly to or from the memory without involving the processor. This mechanism is called **direct memory access** (DMA). The interrupt mechanism is still used by the device to communicate with the processor, but only on completion of the I/O transfer or when an error occurs.

DMA is implemented with a specialized controller that transfers data between an I/O device and memory independent of the processor. The DMA controller becomes the **bus master** and directs the reads or writes between itself and memory. There are three steps in a DMA transfer:

1. The processor sets up the DMA by supplying the identity of the device, the operation to perform on the device, the memory address that is the source or destination of the data to be transferred, and the number of bytes to transfer.
2. The DMA starts the operation on the device and arbitrates for the bus. When the data is available (from the device or memory), it transfers the data. The DMA device supplies the memory address for the read or the write. If the request requires more than one transfer on the bus, the DMA unit generates the next memory address and initiates the next transfer. Using this mechanism, a DMA unit can complete an entire transfer, which may be thousands of bytes in length, without bothering the processor. Many DMA controllers contain some memory to allow them to deal flexibly with delays either in transfer or those incurred while waiting to become bus master.
3. Once the DMA transfer is complete, the controller interrupts the processor, which can then determine by interrogating the DMA device or examining memory whether the entire operation completed successfully.

There may be multiple DMA devices in a computer system. For example, in a system with a single processor-memory bus and multiple I/O buses, each I/O bus controller will often contain a DMA processor that handles any transfers between a device on the I/O bus and the memory.

Unlike either polling or interrupt-driven I/O, DMA can be used to interface a hard disk without consuming all the processor cycles for a single I/O. Of course, if the processor is also contending for memory, it will be delayed when the memory is busy doing a DMA transfer. By using caches, the processor can avoid having to access memory most of the time, thereby leaving most of the memory bandwidth free for use by I/O devices.

Elaboration: To further reduce the need to interrupt the processor and occupy it in handling an I/O request that may involve doing several actual operations, the I/O controller can be made more intelligent. Intelligent controllers are often called *I/O proces-*

sors (as well as *I/O controllers* or *channel controllers*). These specialized processors basically execute a series of I/O operations, called an *I/O program*. The program may be stored in the I/O processor, or it may be stored in memory and fetched by the I/O processor. When using an I/O processor, the operating system typically sets up an I/O program that indicates the I/O operations to be done as well as the size and transfer address for any reads or writes. The I/O processor then takes the operations from the I/O program and interrupts the processor only when the entire program is completed. DMA processors are essentially special-purpose processors (usually single-chip and nonprogrammable), while I/O processors are often implemented with general-purpose microprocessors, which run a specialized I/O program.

Direct Memory Access and the Memory System

When DMA is incorporated into an I/O system, the relationship between the memory system and processor changes. Without DMA, all accesses to the memory system come from the processor and thus proceed through address translation and cache access as if the processor generated the references. With DMA, there is another path to the memory system—one that does not go through the address translation mechanism or the cache hierarchy. This difference generates some problems in both virtual memory systems and systems with caches. These problems are usually solved with a combination of hardware techniques and software support.

The difficulties in having DMA in a virtual memory system arise because pages have both a physical and a virtual address. DMA also creates problems for systems with caches because there can be two copies of a data item: one in the cache and one in memory. Because the DMA processor issues memory requests directly to the memory rather than through the processor cache, the value of a memory location seen by the DMA unit and the processor may differ. Consider a read from disk that the DMA unit places directly into memory. If some of the locations into which the DMA writes are in the cache, the processor will receive the old value when it does a read. Similarly, if the cache is write-back, the DMA may read a value directly from memory when a newer value is in the cache, and the value has not been written back. This is called the *stale data problem* or *coherence problem*.

In a system with virtual memory, should DMA work with virtual addresses or physical addresses? The obvious problem with virtual addresses is that the DMA unit will need to translate the virtual addresses to physical addresses. The major problem with the use of a physical address in a DMA transfer is that the transfer cannot easily cross a page boundary. If an I/O request crossed a page boundary, then the memory locations to which it was being transferred would not necessarily be contiguous in the virtual memory. Consequently, if we use physical addresses, we must constrain all DMA transfers to stay within one page.

Hardware Software Interface

One method to allow the system to initiate DMA transfers that cross page boundaries is to make the DMA work on virtual addresses. In such a system, the DMA unit has a small number of map entries that provide virtual-to-physical mapping for a transfer. The operating system provides the mapping when the I/O is initiated. By using this mapping, the DMA unit need not worry about the location of the virtual pages involved in the transfer.

Another technique is for the operating system to break the DMA transfer into a series of transfers, each confined within a single physical page. The transfers are then *chained* together and handed to an I/O processor or intelligent DMA unit that executes the entire sequence of transfers; alternatively, the operating system can individually request the transfers.

Whichever method is used, the operating system must still cooperate by not remapping pages while a DMA transfer involving that page is in progress.

We have looked at three different methods for transferring data between an I/O device and memory. In moving from polling to an interrupt-driven to a DMA interface, we shift the burden for managing an I/O operation from the processor to a progressively more intelligent I/O controller. These methods have the advantage of freeing up processor cycles. Their disadvantage is that they increase the cost of the I/O system. Because of this, a given computer system can choose which point along this spectrum is appropriate for the I/O devices connected to it.

Before discussing the design of I/O systems, let's look briefly at performance measures of them.

Hardware Software Interface

The coherency problem for I/O data is avoided by using one of three major techniques. One approach is to route the I/O activity through the cache. This ensures that reads see the latest value while writes update any data in the cache. Routing all I/O through the cache is expensive and potentially has a large negative performance impact on the processor, since the I/O data is rarely used immediately and may displace useful data that a running program needs. A second choice is to have the OS selectively invalidate the cache for an I/O read or force write-backs to occur for an I/O write (often called *cache flushing*). This approach requires some small amount of hardware support and is probably more efficient if the software can perform the function easily and efficiently. Because this flushing of large parts of the cache need only happen on DMA block accesses, it will be relatively infrequent. The third approach is to provide a hardware mechanism for selectively flushing (or invalidating) cache entries. Hardware invalidation to ensure cache coherence is typical in multiprocessor systems, and the same technique can be used for I/O; we discuss this topic in detail in Chapter 9.

In ranking of the three ways of doing I/O, which statements are true?

1. If we want the lowest latency for an I/O operation to a single I/O device, the order is polling, DMA, and interrupt driven.
2. In terms of lowest impact on processor utilization from a single I/O device, the order is DMA, interrupt driven, and polling

**Check
Yourself**

8.6

I/O Performance Measures: Examples from Disk and File Systems

How should we compare I/O systems? This is a complex question because I/O performance depends on many aspects of the system and different applications stress different aspects of the I/O system. Furthermore, a design can make complex trade-offs between response time and throughput, making it impossible to measure just one aspect in isolation. For example, handling a request as early as possible generally minimizes response time, although greater throughput can be achieved if we try to handle related requests together. Accordingly, we may increase throughput on a disk by grouping requests that access locations that are close together. Such a policy will increase the response time for some requests, probably leading to a larger variation in response time. Although throughput will be higher, some benchmarks constrain the maximum response time to any request, making such optimizations potentially problematic.

In this section, we give some examples of measurements proposed for determining the performance of disk systems. These benchmarks are affected by a variety of system features, including the disk technology, how disks are connected, the memory system, the processor, and the file system provided by the operating system.

Before we discuss these benchmarks, we need to address a confusing point about terminology and units. The performance of I/O systems depends on the rate at which the system transfers data. The transfer rate depends on the clock rate, which is typically given in $\text{GHz} = 10^9$ cycles per second. The transfer rate is usually quoted in GB/sec. In I/O systems, GBs are measured using base 10 (i.e., $1 \text{ GB} = 10^9 = 1,000,000,000$ bytes), unlike main memory where base 2 is used (i.e., $1 \text{ GB} = 2^{30} = 1,073,741,824$). In addition to adding confusion, this difference introduces the need to convert between base 10 ($1\text{K} = 1000$) and base 2 ($1\text{K} = 1024$) because many I/O accesses are for data blocks that have a size that is a power of two. Rather than complicate all our examples by accurately converting one of the two measurements, we make note here of this distinction and the fact that treating the two measures as if the units were identical introduces a small error. We illustrate this error in Section 8.9.

transaction processing A type of application that involves handling small short operations (called transactions) that typically require both I/O and computation. Transaction processing applications typically have both response time requirements and a performance measurement based on the throughput of transactions.

I/O rate Performance measure of I/Os per unit time, such as reads per second.

data rate Performance measure of bytes per unit time, such as GB/second.

Transaction Processing I/O Benchmarks

Transaction processing (TP) applications involve both a response time requirement and a performance measurement based on throughput. Furthermore, most of the I/O accesses are small. Because of this, TP applications are chiefly concerned with **I/O rate**, measured as the number of disk accesses per second, as opposed to **data rate**, measured as bytes of data per second. TP applications generally involve changes to a large database, with the system meeting some response time requirements as well as gracefully handling certain types of failures. These applications are extremely critical and cost-sensitive. For example, banks normally use TP systems because they are concerned about a range of characteristics. These include making sure transactions aren't lost, handling transactions quickly, and minimizing the cost of processing each transaction. Although dependability in the face of failure is an absolute requirement in such systems, both response time and throughput are critical to building cost-effective systems.

A number of transaction processing benchmarks have been developed. The best-known set of benchmarks is a series developed by the Transaction Processing Council (TPC).

TPC-C, initially created in 1992, simulates a complex query environment. TPC-H models ad hoc decision support—the queries are unrelated and knowledge of past queries cannot be used to optimize future queries; the result is that query execution times can be very long. TPC-R simulates a business decision support system where users run a standard set of queries. In TPC-R, preknowledge of the queries is taken for granted, and the DBMS can be optimized to run these queries. TPC-W is a Web-based transaction benchmark that simulates the activities of a business-oriented transactional Web server. It exercises the database system as well as the underlying Web server software. The TPC benchmarks are described at www.tpc.org.

All the TPC benchmarks measure performance in transactions per second. In addition, they include a response time requirement, so that throughput performance is measured only when the response time limit is met. To model real-world systems, higher transaction rates are also associated with larger systems, both in terms of users and the size of the database that the transactions are applied to. Finally, the system cost for a benchmark system must also be included, allowing accurate comparisons of cost-performance.

File System and Web I/O Benchmarks

File systems, which are stored on disks, have a different access pattern. For example, measurements of UNIX file systems in an engineering environment have found that 80% of accesses are to files of less than 10 KB and that 90% of all file accesses are to data with sequential addresses on the disk. Furthermore, 67% of the accesses were reads, 27% were writes, and 6% were read-modify-write accesses, which read data, modify it, and then rewrite the same location. Such

measurements have led to the creation of synthetic file system benchmarks. One of the most popular of such benchmarks has five phases, using 70 files:

- *MakeDir*: Constructs a directory subtree that is identical in structure to the given directory subtree
- *Copy*: Copies every file from the source subtree to the target subtree
- *ScanDir*: Recursively traverses a directory subtree and examines the status of every file in it
- *ReadAll*: Scans every byte of every file in a subtree once
- *Make*: Compiles and links all the files in a subtree

As we will see in Section 8.7, the design of an I/O system involves knowing what the workload is.

In addition to processor benchmarks, SPEC offers both a file server benchmark (SPECFS) and a Web server benchmark (SPECWeb). SPECFS is a benchmark for measuring NFS (Network File System) performance using a script of file server requests; it tests the performance of the I/O system, including both disk and network I/O, as well as the processor. SPECFS is a throughput-oriented benchmark but with important response time requirements. SPECWeb is a Web server benchmark that simulates multiple clients requesting both static and dynamic pages from a server, as well as clients posting data to the server.

I/O Performance versus Processor Performance

Amdahl's law in Chapter 2 reminds us that neglecting I/O is dangerous. A simple example demonstrates this.

Impact of I/O on System Performance

Suppose we have a benchmark that executes in 100 seconds of elapsed time, where 90 seconds is CPU time and the rest is I/O time. If CPU time improves by 50% per year for the next five years but I/O time doesn't improve, how much faster will our program run at the end of five years?

We know that

$$\text{Elapsed time} = \text{CPU time} + \text{I/O time}$$

$$100 = 90 + \text{I/O time}$$

$$\text{I/O time} = 10 \text{ seconds}$$

EXAMPLE

ANSWER

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

After n years	CPU time	I/O time	Elapsed time	% I/O time
0	90 seconds	10 seconds	100 seconds	10%
1	$\frac{90}{1.5} = 60$ seconds	10 seconds	70 seconds	14%
2	$\frac{60}{1.5} = 40$ seconds	10 seconds	50 seconds	20%
3	$\frac{40}{1.5} = 27$ seconds	10 seconds	37 seconds	27%
4	$\frac{27}{1.5} = 18$ seconds	10 seconds	28 seconds	36%
5	$\frac{18}{1.5} = 12$ seconds	10 seconds	22 seconds	45%

The improvement in CPU performance over five years is

$$\frac{90}{12} = 7.5$$

However, the improvement in elapsed time is only

$$\frac{100}{22} = 4.5$$

and the I/O time has increased from 10% to 45% of the elapsed time.

Check Yourself

Are the following true or false? Unlike processor benchmarks, I/O benchmarks

1. concentrate on throughput rather than latency
2. can require that the data set scale in size or number of users to achieve performance milestones
3. come from organizations rather than from individuals

8.7

Designing an I/O System

There are two primary types of specifications that designers encounter in I/O systems: latency constraints and bandwidth constraints. In both cases, knowledge of the traffic pattern affects the design and analysis.

Latency constraints involve ensuring that the latency to complete an I/O operation is bounded by a certain amount. In the simple case, the system may be

unloaded, and the designer must ensure that some latency bound is met either because it is critical to the application or because the device must receive certain guaranteed service to prevent errors. Examples of the latter are similar to the analysis we looked at in the previous section. Likewise, determining the latency on an unloaded system is relatively easy, since it involves tracing the path of the I/O operation and summing the individual latencies.

Finding the average latency (or distribution of latency) under a load is a much more complex problem. Such problems are tackled either by queuing theory (when the behavior of the workload requests and I/O service times can be approximated by simple distributions) or by simulation (when the behavior of I/O events is complex). Both topics are beyond the limits of this text.

Designing an I/O system to meet a set of bandwidth constraints given a workload is the other typical problem designers face. Alternatively, the designer may be given a partially configured I/O system and be asked to balance the system to maintain the maximum bandwidth achievable as dictated by the preconfigured portion of the system. This latter design problem is a simplified version of the first.

The general approach to designing such a system is as follows:

1. Find the weakest link in the I/O system, which is the component in the I/O path that will constrain the design. Depending on the workload, this component can be anywhere, including the CPU, the memory system, the backplane bus, the I/O controllers, or the devices. Both the workload and configuration limits may dictate where the weakest link is located.
2. Configure this component to sustain the required bandwidth.
3. Determine the requirements for the rest of the system and configure them to support this bandwidth.

The easiest way to understand this methodology is with an example.

I/O System Design

Consider the following computer system:

- A CPU that sustains 3 billion instructions per second and averages 100,000 instructions in the operating system per I/O operation
- A memory backplane bus capable of sustaining a transfer rate of 1000 MB/sec
- SCSI Ultra320 controllers with a transfer rate of 320 MB/sec and accommodating up to 7 disks
- Disk drives with a read/write bandwidth of 75 MB/sec and an average seek plus rotational latency of 6 ms

EXAMPLE

ANSWER

If the workload consists of 64 KB reads (where the block is sequential on a track) and the user program needs 200,000 instructions per I/O operation, find the maximum sustainable I/O rate and the number of disks and SCSI controllers required. Assume that the reads can always be done on an idle disk if one exists (i.e., ignore disk conflicts).

The two fixed components of the system are the memory bus and the CPU. Let's first find the I/O rate that these two components can sustain and determine which of these is the bottleneck. Each I/O takes 200,000 user instructions and 100,000 OS instructions, so

Maximum I/O rate of CPU =

$$\frac{\text{Instruction execution rate}}{\text{Instructions per I/O}} = \frac{3 \times 10^9}{(200 + 100) \times 10^3} = 10,000 \frac{\text{I/Os}}{\text{second}}$$

Each I/O transfers 64 KB, so

$$\text{Maximum I/O rate of bus} = \frac{\text{Bus bandwidth}}{\text{Bytes per I/O}} = \frac{1000 \times 10^6}{64 \times 10^3} = 15,625 \frac{\text{I/Os}}{\text{second}}$$

The CPU is the bottleneck, so we can now configure the rest of the system to perform at the level dictated by the CPU, 10,000 I/Os per second.

Let's determine how many disks we need to be able to accommodate 10,000 I/Os per second. To find the number of disks, we first find the time per I/O operation at the disk:

$$\begin{aligned} \text{Time per I/O at disk} &= \text{Seek} + \text{rotational time} + \text{Transfer time} \\ &= 6 \text{ ms} + \frac{64 \text{ KB}}{75 \text{ MB/sec}} = 6.9 \text{ ms} \end{aligned}$$

Thus, each disk can complete 1000 ms/6.9 ms or 146 I/Os per second. To saturate the CPU requires 10,000 I/Os per second, or $10,000/146 \approx 69$ disks.

To compute the number of SCSI buses, we need to check the average transfer rate per disk to see if we can saturate the bus, which is given by

$$\text{Transfer rate} = \frac{\text{Transfer size}}{\text{Transfer time}} = \frac{64 \text{ KB}}{6.9 \text{ ms}} \approx 9.56 \text{ MB/sec}$$

The maximum number of disks per SCSI bus is 7, which won't saturate this bus. This means we will need $69/7$, or 10 SCSI buses and controllers.

Notice the significant number of simplifying assumptions that are needed to do this example. In practice, many of these simplifications might not hold for critical I/O-intensive applications (such as databases). For this reason, simulation is often the only realistic way to predict the I/O performance of a realistic workload.

8.8**Real Stuff: A Digital Camera**

Digital cameras are basically embedded computers with removable, writable, non-volatile, storage, and interesting I/O devices. Figure 8.14 shows our example.



FIGURE 8.14 The Sanyo VPC-SX500 with Flash memory card and IBM Microdrive. Although newer cameras offer more pixels per picture, the principles are the same. This 1360×1024 pixel digital camera stores pictures either using CompactFlash memory or using a IBM Microdrive. This photo was taken using a 340 MB microdrive and a 8 MB CompactFlash memory. As Figure 8.15 shows, in 2004 the capacities are as large as 1 GB to 4 GB. It is 4.3 inches wide $\times$ 2.5 inches high $\times$ 1.6 inches deep, and it weighs 7.4 ounces. In addition to taking a still picture and converting it to JPEG format every 0.9 seconds, it can record a Quick Time video clip at VGA size (640×480). One technological advantage is the use of a custom system on a chip to reduce size and power, so the camera only needs two AA batteries to operate versus four in other digital cameras.

When powered on, the microprocessor first runs diagnostics on all components and writes any error messages to the liquid crystal display (LCD) on the back of the camera. This camera uses a 1.8-inch low-temperature polysilicon TFT color LCD. When photographers take pictures, they first hold the shutter halfway so that the microprocessor can take a light reading. The microprocessor then keeps the shutter open to get the necessary light, which is captured by a charged-couple device (CCD) as red, green, and blue pixels.

For the camera in Figure 8.14, the CCD is a 1/2-inch, 1360×1024 pixel, progressive-scan chip. The pixels are scanned out row by row and then passed through routines for white balance, color, and aliasing correction, and then stored in a 4 MB frame buffer. The next step is to compress the image into a standard format, such as JPEG, and store it in the removable Flash memory. The photographer picks the compression, in this camera called either fine or normal, with a compression ratio of 10 to 20 times. A fine-quality compressed image takes less than 0.5 MB, and a normal-quality compressed image takes about 0.25 MB. The microprocessor then updates the LCD display to show that there is room for one less picture.

Although the previous paragraph covers the basics of a digital camera, there are many more features that are included: showing the recorded images on the color LCD display; sleep mode to save battery life; monitoring battery energy; buffering to allow recording a rapid sequence of uncompressed images; and, in this camera, video recording using MPEG format and audio recording using WAV format.

This camera allows the photographer to use a Microdrive disk instead of CompactFlash memory. Figure 8.15 compares CompactFlash and the IBM Microdrive.

Characteristics	Sandisk Type I CompactFlash SDCFB-128-768	Sandisk Type II CompactFlash SDCFB-1000-768	Hitachi 4 GB Microdrive DSCM-10340
Formatted data capacity (MB)	128	1000	4000
Bytes per sector	512	512	512
Data transfer rate (MB/sec)	4 (burst)	4 (burst)	4–7
Link speed to buffer (MB/sec)	6	6	33
Power standby/operating (W)	0.15/0.66	0.15/0.66	0.07/0.83
Size: height $\times$ width $\times$ depth (inches)	1.43 $\times$ 1.68 $\times$ 0.13	1.43 $\times$ 1.68 $\times$ 0.13	1.43 $\times$ 1.68 $\times$ 0.16
Weight in grams (454 grams/pound)	11.4	13.5	16
Write cycles before sector wear-out	300,000	300,000	not applicable
Mean time between failures (hours)	> 1,000,000	> 1,000,000	(see caption)
Best price (2004)	\$40	\$200	\$480

FIGURE 8.15 Characteristics of three storage alternatives for digital cameras. Hitachi matches the Type II form factor in the Microdrive, while the CompactFlash card uses that space to include many more Flash chips. Hitachi does not quote MTTF for the 1.0-inch drives, but the service life is five years or 8800 powered-on hours, whichever is first. They rotate at 3600 RPM and have 12 ms seek times.

The CompactFlash standard package was proposed by Sandisk Corporation in 1994 for the PCMCIA-ATA cards of portable PCs. Because it follows the ATA interface, it simulates a disk interface including seek commands, logical tracks, and so on. It includes a built-in controller to support many types of Flash memory and to help with chip yield for Flash memories by mapping out bad blocks.

The electronic brain of this camera is an embedded computer with several special functions embedded on the chip. Figure 8.16 shows the block diagram of a chip similar to the one in the camera. Such chips have been called *systems on a chip* (SOC) because they essentially integrate into a single chip all the parts that were found on a small printed circuit board of the past. SOC generally reduces size and lowers power compared to less integrated solutions. The manufacturer claims the SOC enables the camera to operate on half the number of batteries and to offer a smaller form factor than competitors' cameras.

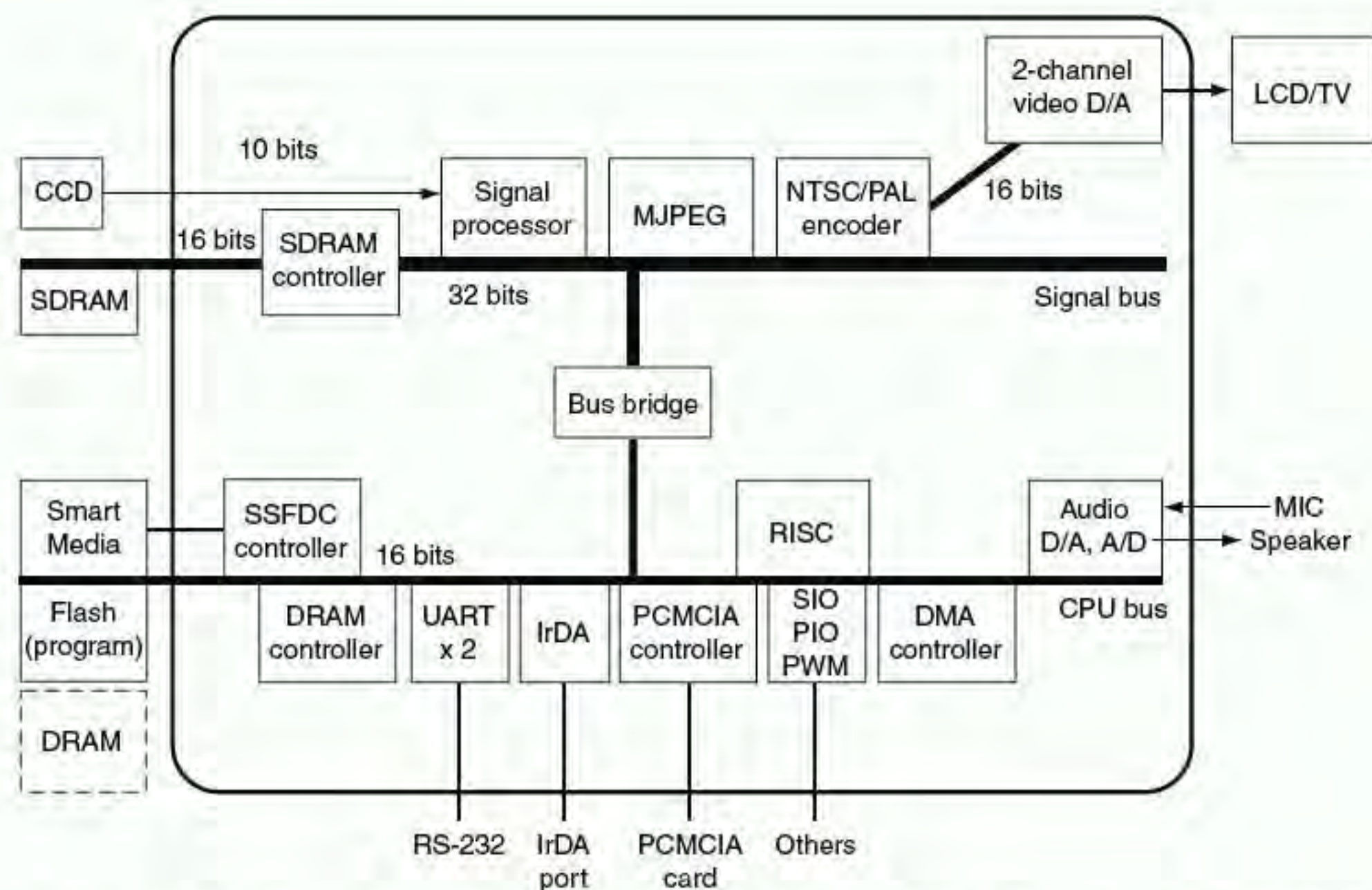


FIGURE 8.16 The system on a chip (SOC) found in Sanyo digital cameras. This block diagram is for the predecessor of the SOC in the camera in Figure 8.14. The successor SOC, called Super Advanced IC, uses three buses instead of two, operates at 60 MHz, consumes 800 mW, and fits 3.1M transistors in a 10.2×10.2 mm die using a 0.35-micron process. Note that this embedded system has twice as many transistors as the state-of-the-art, high-performance microprocessor in 1990! The SOC in the figure is limited to processing 1024×768 pixels, but its successor supports 1360×1024 pixels. (See Okada, Matsuda, Yamada, and Kobayashi [1999]).

For higher performance, it has two buses. The 16-bit bus is for the many slower I/O devices: Smart Media interface, program and data memory, and DMA. The 32-bit bus is for the SDRAM, the signal processor (which is connected to the CCD), the Motion JPEG encoder, and the NTSC/PAL encoder (which is connected to the LCD). Unlike desktop microprocessors, note the large variety of I/O buses that this chip must integrate. The 32-bit RISC MPU is a proprietary design and runs at 28.8 MHz, the same clock rate as the buses. This 700 mW chip contains 1.8M transistors in a 10.5×10.5 mm die implemented using a 0.35-micron process.

8.9

Fallacies and Pitfalls

Fallacy: The rated mean time to failure of disks is 1,200,000 hours or almost 140 years, so disks practically never fail.

The current marketing practices of disk manufacturers can mislead users. How is such an MTTF calculated? Early in the process manufacturers will put thousands of disks in a room, run them for a few months, and count the number that fail. They compute MTTF as the total number of hours that the disks were cumulatively up divided by the number that failed.

One problem is that this number far exceeds the lifetime of a disk, which is commonly assumed to be five years or 43,800 hours. For this large MTTF to make some sense, disk manufacturers argue that the calculation corresponds to a user who buys a disk, and then keeps replacing the disk every five years—the planned lifetime of the disk. The claim is that if many customers (and their great-grandchildren) did this for the next century, on average they would replace a disk 27 times before a failure, or about 140 years.

A more useful measure would be percentage of disks that fail. Assume 1000 disks with a 1,200,000-hour MTTF and that the disks are used 24 hours a day. If you replaced failed disks with a new one having the same reliability characteristics, the number that would fail over five years (43,800 hours) is

$$\text{Failed disks} = \frac{1000 \text{ drives} \times 43,800 \text{ hours/drive}}{1,200,000 \text{ hours/failure}} = 36$$

Stated alternatively, 3.6% would fail over the 5-year period.

Pitfall: Using the peak transfer rate of a portion of the I/O system to make performance projections or performance comparisons.

Many of the components of an I/O system, from the devices to the controllers to the buses, are specified using their peak bandwidths. In practice, these peak bandwidth measurements are often based on unrealistic assumptions about the system

or are unattainable because of other system limitations. For example, in quoting bus performance, the peak transfer rate is sometimes specified using a memory system that is impossible to build. For networked systems, the software overhead of initiating communication is ignored.

The 32-bit, 33 MHz PCI bus has a peak bandwidth of about 133 MB/sec. In practice, even for long transfers, it is difficult to sustain more than about 80 MB/sec for realistic memory systems. As mentioned above, users of wireless networks typically achieve only about a third of the peak bandwidth.

Amdahl's law also reminds us that the throughput of an I/O system will be limited by the lowest-performance component in the I/O path.

Fallacy: Magnetic disk storage is on its last legs and will be replaced shortly.

This is both a fallacy and a pitfall. Such claims have been made constantly for the past 20 years, though the string of failed alternatives in recent years seems to have reduced the level of claims for the death of magnetic storage. Among the unsuccessful contenders are magnetic bubble memories, optical storage, and holographic storage. None of these systems has matched the combination of characteristics that favor magnetic disks: high reliability, nonvolatility, low cost, reasonable access time, and rapid improvement. Magnetic storage technology continues to improve at the same—or faster—pace that it has sustained over the past 25 years.

Pitfall: Using magnetic tapes to back up disks.

Once again, this is both a fallacy and a pitfall.

Magnetic tapes have been part of computer systems as long as disks because they use similar technology as disks, and hence historically have followed the same density improvements. The historic cost-performance difference between disks and tapes is based on a sealed, rotating disk having lower access time than sequential tape access but removable spools of magnetic tape mean many tapes can be used per reader and they can be very long and so have high capacity. Hence, in the past a single magnetic tape could hold the contents of many disks, and since it was 10 to 100 times cheaper per gigabyte than disks, it was a useful backup medium.

The claim was that magnetic tapes must track disks since innovations in disks must help tapes. This claim was important because tapes were a small market and could not afford a separate large research and development effort. One reason the market is small is that desktop owners generally do not back up disks onto tape, and so while desktops are by far the largest market for disks, desktops are a small market for tapes.

Alas, the larger market has led disks to improve much more quickly than tapes. Starting in 2000 to 2002, the largest popular disk was larger than the largest popular tape. In that same time frame, the price per gigabyte of ATA disks dropped below that of tapes. Tape apologists now claim that tapes have compatibility requirements that are not imposed on disks; tape readers must read or write the current and previous generation of tapes, and must read the last four generations

of tapes. As disks are closed systems, disk heads need only read the platters enclosed with them, and this advantage explains why disks are improving much more rapidly.

Today, some organizations have dropped tapes altogether, using networks and remote disks to replicate the data geographically. The sites are picked so that disasters would not take out both sites, enabling instantaneous recovery time. (Long recovery time is another serious drawback to the serial nature of magnetic tapes.) Such a solution depends on advances in disk capacity and network bandwidth to make economic sense, but these two are getting much greater investment and hence have better recent records of accomplishment than tape.

Fallacy: A 100 MB/sec bus can transfer 100 MB of data in 1 second.

First, you generally cannot use 100% of any computer resource. For a bus, you would be fortunate to get 70% to 80% of the peak bandwidth. Time to send the address, time to acknowledge the signals, and stalls while waiting to use a busy bus are among the reasons you cannot use 100% of a bus.

Second, the definition of a megabyte of storage and a megabyte per second of bandwidth do not agree. As we discussed on page 597, I/O bandwidth measures are usually quoted in base 10 (i.e., 1 MB/sec = 10^6 bytes/sec), while 1 MB of data is typically a base 2 measure (i.e., 1 MB = 2^{20} bytes). How significant is this distinction? If we could use 100% of the bus for data transfer, the time to transfer 100 MB of data on a 100-MB/sec bus is actually

$$\frac{100 \times 2^{20}}{100 \times 10^6} = \frac{1,048,576}{1,000,000} = 1.048576 \approx 1.05 \text{ second}$$

A similar but larger error is introduced when we treat a gigabyte of data transferred or stored as equivalent, meaning 10^9 versus 2^{30} bytes.

Pitfall: Trying to provide features only within the network versus end to end.

The concern is providing at a lower level features that can only be accomplished at the highest level, thus only partially satisfying the communication demand. Saltzer, Reed, and Clark [1984] give the *end-to-end argument* as

The function in question can completely and correctly be specified only with the knowledge and help of the application standing at the endpoints of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible.

Their example of the pitfall was a network at MIT that used several gateways, each of which added a checksum from one gateway to the next. The programmers of the application assumed the checksum guaranteed accuracy, incorrectly believing that the message was protected while stored in the memory of each gateway. One gateway developed a transient failure that swapped one pair of bytes per million

bytes transferred. Over time the source code of one operating system was repeatedly passed through the gateway, thereby corrupting the code. The only solution was to correct the infected source files by comparing to paper listings and repairing the code by hand! Had the checksums been calculated and checked by the application running on the end systems, safety would have been assured.

There is a useful role for intermediate checks, however, provided that end-to-end checking is available. End-to-end checking may show that *something* is broken between two nodes, but it doesn't point to where the problem is. Intermediate checks can discover *what* is broken. You need both for repair.

Pitfall: Moving functions from the CPU to the I/O processor, expecting to improve performance without a careful analysis.

There are many examples of this pitfall trapping people, although I/O processors, when properly used, can certainly enhance performance. A frequent instance of this fallacy is the use of intelligent I/O interfaces, which, because of the higher overhead to set up an I/O request, can turn out to have worse latency than a processor-directed I/O activity (although if the processor is freed up sufficiently, system throughput may still increase). Frequently, performance falls when the I/O processor has much lower performance than the main processor. Consequently, a small amount of main processor time is replaced with a larger amount of I/O processor time. Workstation designers have seen both these phenomena repeatedly.

Myer and Sutherland [1968] wrote a classic paper on the trade-off of complexity and performance in I/O controllers. Borrowing the religious concept of the "wheel of reincarnation," they eventually noticed they were caught in a loop of continuously increasing the power of an I/O processor until it needed its own simpler coprocessor:

We approached the task by starting with a simple scheme and then adding commands and features that we felt would enhance the power of the machine.

Gradually the [display] processor became more complex . . . Finally the display processor came to resemble a full-fledged computer with some special graphics features. And then a strange thing happened. We felt compelled to add to the processor a second, subsidiary processor, which, itself, began to grow in complexity. It was then that we discovered the disturbing truth. Designing a display processor can become a never-ending cyclical process. In fact, we found the process so frustrating that we have come to call it the "wheel of reincarnation."

8.10

Concluding Remarks

I/O systems are evaluated on several different characteristics: dependability; the variety of I/O devices supported; the maximum number of I/O devices; cost; and

performance, measured both in latency and in throughput. These goals lead to widely varying schemes for interfacing I/O devices. In the low-end and midrange systems, buffered DMA is likely to be the dominant transfer mechanism. In the high-end systems, latency and bandwidth may both be important, and cost may be secondary. Multiple paths to I/O devices with limited buffering often characterize high-end I/O systems. Typically, being able to access the data on an I/O device at any time (high availability) becomes more important as systems grow. As a result, redundancy and error correction mechanisms become more and more prevalent as we enlarge the system.

Storage and networking demands are growing at unprecedented rates, in part because of increasing demands for all information to be at your fingertips. One estimate is that the amount of information created in 2002 was 5 exabytes—equivalent to 500,000 copies of the text in the U.S. Library of Congress—and that the total amount of information in the world doubled in the last three years [Lyman and Varian 2003].

Future directions of I/O include expanding the reach of wired and wireless networks, with nearly every device potentially having an IP address, and the continuing transformation from parallel buses to serial networks and switches. However, consolidation in the disk industry may lead to a slowdown in improvement in disk capacity to earlier rates, which have doubled every year between 2000 and 2004.

Understanding Program Performance

The performance of an I/O system, whether measured by bandwidth or latency, depends on all the elements in the path between the device and memory, including the operating system that generates the I/O commands. The bandwidth of the buses, the memory, and the device determine the maximum transfer rate from or to the device. Similarly, the latency depends on the device latency, together with any latency imposed by the memory system or buses. The effective bandwidth and response latency also depend on other I/O requests that may cause contention for some resource in the path. Finally, the operating system is a bottleneck. In some cases, the OS takes a long time to deliver an I/O request from a user program to an I/O device, leading to high latency. In other cases, the operating system effectively limits the I/O bandwidth because of limitations in the number of concurrent I/O operations it can support.

Keep in mind that while performance can help sell an I/O system, users overwhelmingly demand dependability and capacity from their I/O systems.



Historical Perspective and Further Reading

The history of I/O systems is a fascinating one. This  [Section 8.11](#) gives a brief history of magnetic disks, RAID, databases, the Internet, the World Wide Web, and how Ethernet continues to triumph over its challengers.



Exercises

8.1 [10] <§§8.1–8.2> Here are two different I/O systems intended for use in transaction processing:

- System A can support 1500 I/O operations per second.
- System B can support 1000 I/O operations per second.

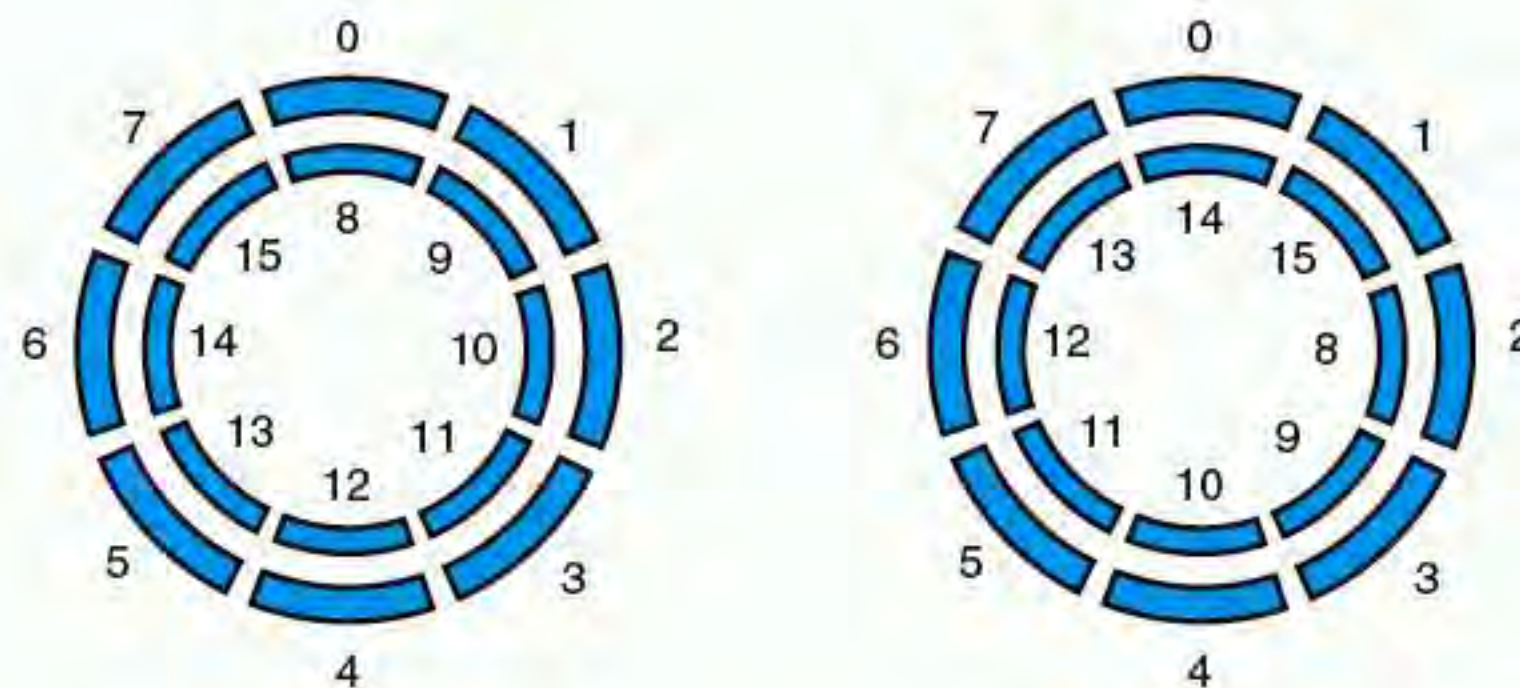
The systems use the same processor that executes 500 million instructions per second. Assume that each transaction requires 5 I/O operations and that each I/O operation requires 10,000 instructions. Ignoring response time and assuming that transactions may be arbitrarily overlapped, what is the maximum transaction-per-second rate that each machine can sustain?

8.2 [15] <§§8.1–8.2> The latency of an I/O operation for the two systems in Exercise 8.1 differs. The latency for an I/O on system A is equal to 20 ms, while for system B the latency is 18 ms for the first 500 I/Os per second and 25 ms per I/O for each I/O between 500 and 1000 I/Os per second. In the workload, every 10th transaction depends on the immediately preceding transaction and must wait for its completion. What is the maximum transaction rate that still allows every transaction to complete in 1 second and that does not exceed the I/O bandwidth of the machine? (For simplicity, assume that all transaction requests arrive at the beginning of a 1-second interval.)

8.3 [5] <§§8.1–8.2> Suppose we want to use a laptop to send 100 files of approximately 40 MB each to another computer over a 5 Mbit/sec wireless connection. The laptop battery currently holds 100,000 joules of energy. The wireless networking card alone consumes 5 watts while transmitting, while the rest of the laptop always consumes 35 watts. Before each file transfer we need 10 seconds to choose which file to send. How many complete files can we transfer before the laptop's battery runs down to zero?

8.4 [10] <§§8.1–8.2> Consider the laptop's hard disk power consumption in Exercise 8.3. Assume that it is no longer constant, but varies between 6 watts when it is spinning and 1 watt when it is not spinning. The power consumed by the laptop apart from the hard disk and wireless card is a constant 32 watts. Suppose that the hard disk's transfer rate is 50 MB/sec, its delay before it can begin transfer is 20 ms, and at all other times it does not spin. How many complete files can we transfer before the laptop's battery runs down to zero? How much energy would we need to send all 100 files? (Consider that the wireless card cannot send data until it is in memory.)

8.5 [5] <§8.3> The following simplified diagram shows two potential ways of numbering the sectors of data on a disk (only two tracks are shown and each track has eight sectors). Assuming that typical reads are contiguous (e.g., all 16 sectors are read in order), which way of numbering the sectors will be likely to result in higher performance? Why?



8.6 [20] <§8.3> In this exercise, we will run a program to evaluate the behavior of a disk drive. Disk sectors are addressed sequentially within a track, tracks sequentially within cylinders, and cylinders sequentially within the disk. Determining head switch time and cylinder switch time is difficult because of rotational effects. Even determining platter count, sectors/track, and rotational delay is difficult based on observation of typical disk workloads.

The key is to factor out disk rotational effects by making consecutive seeks to individual sectors with addresses that differ by a linearly increasing amount starting with 0, 1, 2, and so forth. The Skippy algorithm, from work by Nisha Talagala and colleagues of U.C. Berkeley [2000], is

```
fd = open("raw disk device");
for (i = 0; i < measurements; i++) {
    //time the following sequence, and output <i, time>
    lseek(fd, i * SINGLE_SECTOR, SEEK_CUR);
    write(fd, buffer, SINGLE_SECTOR);
}
close(fd);
```

The basic algorithm skips through the disk, increasing the distance of the seek by one sector before every write, and outputs the distance and time for each write. The raw device interface is used to avoid file system optimizations. `SINGLE_SECTOR` is the size of a single sector in bytes. The `SEEK_CUR` argument to `lseek` moves the file pointer an amount relative to the current pointer. A technical report describing Skippy and two other disk drive benchmarks (run in seconds or minutes rather than hours or days) is at <http://sunsite.berkeley.edu/Dienst/UI/2.0/Describe/ncstrl.ucb/CSD-99-1063>.

Run the Skippy algorithm on a disk drive of your choosing.

- a. What is the number of heads?
- b. The number of platters?
- c. What is the rotational latency?
- d. What is the head switch time (the time to switch the head that is reading from one disk surface to another without moving the arm; that is, in the same cylinder)?
- e. What is the cylinder switch time? (It is the time to move the arm to the next sequential cylinder.)

8.7 [20] <§8.3> Figure 8.17 shows the output from running the benchmark Skippy on a disk.

- a. What is the number of heads?
- b. The number of platters?
- c. What is the rotational latency?
- d. What is the head switch time (the time to switch the head that is reading from one disk surface to another without moving the arm; that is, in the same cylinder)?
- e. What is the cylinder switch time (the time to move the arm to the next sequential cylinder)?

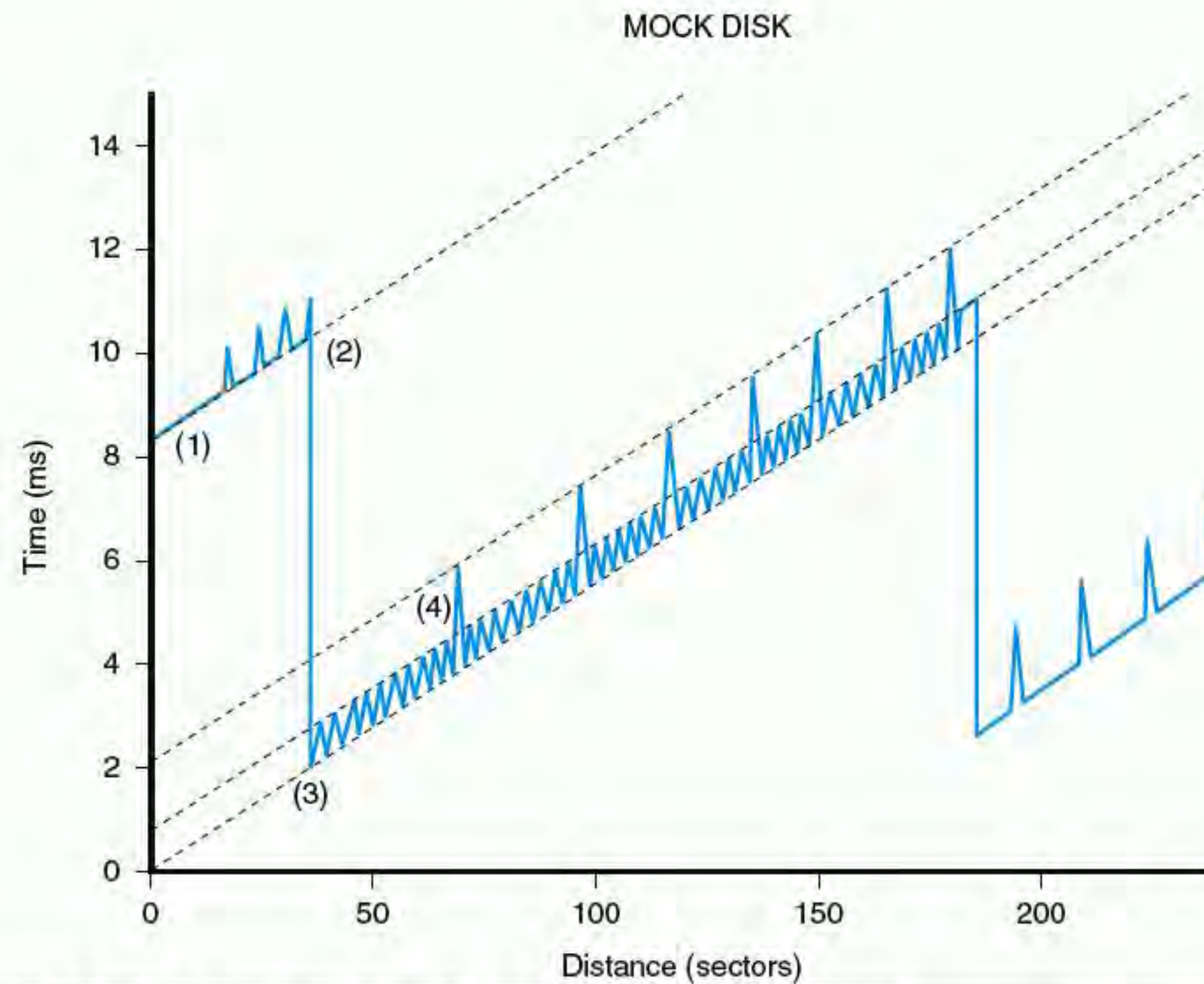


FIGURE 8.17 Example output of Skippy for a hypothetical disk.

8.8 [10] <§8.3> Consider two RAID disk systems that are meant to store 10 terabytes of data (not counting any redundancy). System A uses RAID 1 technology, and System B uses RAID 5 technology with four disks in a “protection group.”

- How many more terabytes of storage are needed in System A than in System B?
- Suppose an application writes one block of data to the disk. If reading or writing a block takes 30 ms, how much time will the write take on System A in the worst case? How about on System B in the worst case?
- Is System A more reliable than System B? Why or why not?

8.9 [15] <§8.3> What can happen to a RAID 5 system if the power fails between the write update to the data block and the write update to the check block so that only one of the two is successfully written? What could be done to prevent this from happening?

8.10 [5] <§8.3> The speed of light is approximately 3×10^8 meters per second, and electrical signals travel at about 50% of this speed in a conductor. When the

term *high speed* is applied to a network, it is the bandwidth that is higher, not necessarily the velocity of the electrical signals. How much of a factor is the actual “flight time” for the electrical signals? Consider two computers that are 20 meters apart and two computers that are 2000 kilometers apart. Compare your results to the latencies reported in the example on page 8.3-7 in  [Section 8.3](#).

8.11 [5] <§8.3> The number of bytes in transit on a network is defined as the flight time (described in Exercise 8.10) multiplied by the delivered bandwidth. Calculate the number of bytes in transit for the two networks described in Exercise 8.10, assuming a delivered bandwidth of 6 MB/sec.

8.12 [5] <§8.3> A secret agency simultaneously monitors 100 cellular phone conversations and multiplexes the data onto a network with a bandwidth of 5 MB/sec and an overhead latency of 150 μ s per 1 KB message. Calculate the transmission time per message and determine whether there is sufficient bandwidth to support this application. Assume that the phone conversation data consists of 2 bytes sampled at a rate of 4 KHz.

8.13 [5] <§8.3> Wireless networking has a much higher bit error rate (BER) than wired networking. One way to cope with a higher BER is to use an error correcting code (ECC) on the transmitted data. A very simple ECC is to triplicate each bit, encoding each zero as 000 and each one as 111. When an encoded 3-bit pattern is received, the system chooses the most likely original bit.

- If the system received 001, what is the most likely value of the original bit?
- If 000 was sent but a double-bit error causes it to be received as 110, what will the receiver believe was the original bit's value?
- How many bit errors can this simple ECC correct?
- How many bit errors can this ECC detect?
- If 1 out of every 100 bits sent over the network is incorrect, what percentage of bit errors would a receiver using this ECC not detect?

8.14 [5] <§8.3> There are two types of parity: even and odd. A binary word with even parity and no errors will have an even number of 1s in it, while a word with odd parity and no errors will have an odd number of 1's in it. Compute the parity bit for each of the following 8-bit words if even parity is used:

- 01100111
- 01010101

8.15 [10] <§8.3>

- If a system uses even parity, and the word 0111 is read from the disk, can we tell if there is a single-bit error?

- b. If a system uses odd parity, and the word 0101 appears on the processor-memory bus, we suspect that a single-bit error has occurred. Can we tell which bit the error occurs in? Why or why not?
- c. If a system uses even parity and the word 0101 appears on the processor-memory bus, can we tell if there is a double-bit error?

8.16 [10] <§8.3> A program repeatedly performs a three-step process: It reads in a 4 KB block of data from disk, does some processing on that data, and then writes out the result as another 4 KB block elsewhere on the disk. Each block is contiguous and randomly located on a single track on the disk. The disk drive rotates at 10,000 RPM, has an average seek time of 8 ms, and has a transfer rate of 50 MB/sec. The controller overhead is 2 ms. No other program is using the disk or processor, and there is no overlapping of disk operation with processing. The processing step takes 20 million clock cycles, and the clock rate is 5 GHz. What is the overall speed of the system in blocks processed per second?

8.17 [5] <§8.4> The OSI network protocol is a hierarchy of layers of abstraction, creating an interface between network applications and the physical wires. This is similar to the levels of abstraction used in the ISA interface between software and hardware. Name three advantages to using abstraction in network protocol design.

8.18 [5] <§§8.3, 8.5> Suppose we have a system with the following characteristics:

1. A memory and bus system supporting block access of 4 to 16 32-bit words.
2. A 64-bit synchronous bus clocked at 200 MHz, with each 64-bit transfer taking 1 clock cycle, and 1 clock cycle required to send an address to memory.
3. Two clock cycles needed between each bus operation. (Assume the bus is idle before an access.)
4. A memory access time for the first four words of 200 ns; each additional set of four words can be read in 20 ns.

Assume that the bus and memory systems described above are used to handle disk accesses from disks like the one described in the example on page 570. If the I/O is allowed to consume 100% of the bus and memory bandwidth, what is the maximum number of simultaneous disk transfers that can be sustained for the two block sizes?

8.19 [5] <§8.5> In the system described in Exercise 8.18, the memory system took 200 ns to read the first four words, and each additional four words required 20 ns. Assuming that the memory system takes 150 ns to read the first four words and 30 ns to read each additional four words, find the sustained bandwidth and the latency for a read of 256 words for transfers that use 4-word blocks and for trans-

fers that use 16-word blocks. Also compute the effective number of bus transactions per second for each case.

8.20 [5] <§8.5> Exercise 8.19 demonstrates that using larger block sizes results in an increase in the maximum sustained bandwidth that can be achieved. Under what conditions might a designer tend to favor smaller block sizes? Specifically, why would a designer choose a block size of 4 instead of 16 (assuming all of the characteristics are as identified in Exercise 8.19)?

8.21 [15] <§8.5> This question examines in more detail how increasing the block size for bus transactions decreases the total latency required and increases the maximum sustainable bandwidth. In Exercise 8.19, two different block sizes are considered (4 words and 16 words). Compute the total latency and the maximum bandwidth for all of the possible block sizes (between 4 and 16) and plot your results. Summarize what you learn by looking at your graph.

8.22 [15] <§8.5> This exercise is similar to Exercise 8.21. This time fix the block size at 4 and 16 (as in Exercise 8.19), but compute latencies and bandwidths for reads of different sizes. Specifically, consider reads of from 4 to 256 words, and use as many data points as you need to construct a meaningful graph. Use your graph to help determine at what point block sizes of 16 result in a reduced latency when compared with block sizes of 4.

8.23 [10] <§8.5> This exercise examines a design alternative to the system described in Exercise 8.18 that may improve the performance of writes. For writes, assume all of the characteristics reported in Exercise 8.18 as well as the following:

The first 4 words are written 200 ns after the address is available, and each new write takes 20 ns. Assume a bus transfer of the most recent data to write, and a write of the previous 4 words can be overlapped.

The performance analysis reported in the example would thus remain unchanged for writes (in actuality, some minor changes might exist due to the need to compute error correction codes, etc., but we'll ignore this). An alternative bus scheme relies on separate 32-bit address and data lines. This will permit an address and data to be transmitted in the same cycle. For this bus alternative, what will the latency of the entire 256-word transfer be? What is the sustained bandwidth? Consider block sizes of 4 and 8 words. When do you think the alternative scheme would be heavily favored?

8.24 <20> <§8.5> Consider an asynchronous bus used to interface an I/O device to the memory system described in Exercise 8.18. Each I/O request asks for 16 words of data from the memory, which, along with the I/O device, has a 4-word bus. Assume the same type of handshaking protocol as appears in Figure 8.10 on page 584 except that it is extended so that the memory can continue the

transaction by sending additional blocks of data until the transaction is complete. Modify Figure 8.10 (both the steps and diagram) to indicate how such a transfer might take place. Assuming that each handshaking step takes 20 ns and memory access takes 60 ns, how long does it take to complete a transfer? What is the maximum sustained bandwidth for this asynchronous bus, and how does it compare to the synchronous bus in the example?

8.25 [1 day–1 week] <§§8.2–8.5>  [For More Practice](#): Writing Code to Benchmark I/O Performance

8.26 [3 days–1 week] <§§8.3–8.5>  [In More Depth](#): Ethernet Simulation

8.27 [15] <§8.5> We want to compare the maximum bandwidth for a synchronous and an asynchronous bus. The synchronous bus has a clock cycle time of 50 ns, and each bus transmission takes 1 clock cycle. The asynchronous bus requires 40 ns per handshake. The data portion of both buses is 32 bits wide. Find the bandwidth for each bus when performing one-word reads from a 200-ns memory.

8.28 [20] <§8.5> Suppose we have a system with the following characteristics:

1. A memory and bus system supporting block access of 4 to 16 32-bit words.
2. A 64-bit synchronous bus clocked at 200 MHz, with each 64-bit transfer taking 1 clock cycle, and 1 clock cycle required to send an address to memory.
3. Two clock cycles needed between each bus operation. (Assume the bus is idle before an access.)
4. A memory access time for the first four words of 200 ns; each additional set of four words can be read in 20 ns. Assume that a bus transfer of the most recently read data and a read of the next four words can be overlapped.

Find the sustained bandwidth and the latency for a read of 256 words for transfers that use 4-word blocks and for transfers that use 16-word blocks. Also compute the effective number of bus transactions per second for each case. Recall that a single bus transaction consists of an address transmission followed by data.

8.29 [10] <§8.5> Let's determine the impact of polling overhead for three different devices. Assume that the number of clock cycles for a polling operation—including transferring to the polling routine, accessing the device, and restarting the user program—is 400 and that the processor executes with a 500-MHz clock.

Determine the fraction of CPU time consumed for the following three cases, assuming that you poll often enough so that no data is ever lost and assuming that the devices are potentially always busy:

1. The mouse must be polled 30 times per second to ensure that we do not miss any movement made by the user.

2. The floppy disk transfers data to the processor in 16-bit units and has a data rate of 50 KB/sec. No data transfer can be missed.
3. The hard disk transfers data in four-word chunks and can transfer at 4 MB/sec. Again, no transfer can be missed.

8.30 [15] <§§8.3–8.6> For the I/O system described in Exercise 8.45, find the maximum instantaneous bandwidth at which data can be transferred from disk to memory using as many disks as needed. How many disks and I/O buses (the minimum of each) do you need to achieve the bandwidth? Since you need only achieve this bandwidth for an instant, latencies need not be considered.

8.31 [20] <§§8.3–8.6>  [In More Depth](#): Disk Arrays versus Single Disk

8.32 [10] <§§8.3–8.6>  [In More Depth](#): Disk Arrays Bandwidth

8.33 [5] <§8.6> Suppose you are designing a microprocessor that uses special instructions to access I/O devices (instead of mapping the devices to memory addresses). What special instructions would you need to include? What additional bus lines would you need this microprocessor to support in order to address I/O devices?

8.34 <§8.6> An important advantage of interrupts over polling is the ability of the processor to perform other tasks while waiting for communication from an I/O device. Suppose that a 1 GHz processor needs to read 1000 bytes of data from a particular I/O device. The I/O device supplies 1 byte of data every 0.02 ms. The code to process the data and store it in a buffer takes 1000 cycles.

- a. If the processor detects that a byte of data is ready through polling, and a polling iteration takes 60 cycles, how many cycles does the entire operation take?
- b. If instead, the processor is interrupted when a byte is ready, and the processor spends the time between interrupts on another task, how many cycles of this other task can the processor complete while the I/O communication is taking place? The overhead for handling an interrupt is 200 cycles.

8.35 [20] <§§8.3–8.6>  [For More Practice](#): Finding I/O Bandwidth Bottlenecks

8.36 [15] <§§8.3–8.6>  [For More Practice](#): Finding I/O Bandwidth Bottlenecks

8.37 [15] <§§7.3, 7.5, 8.5, 8.6>  [For More Practice](#): I/O System Operation

8.38 [10] <§8.6> Write a paragraph identifying some of the simplifying assumptions made in the analysis below:

Suppose we have a processor that executes with a 500-MHz clock and the number of clock cycles for a polling operation—including transferring to the polling routine, accessing the device, and restarting the user program—is 400. The hard disk

transfers data in four-word chunks and can transfer at 4 MB/sec. Assume that you poll often enough that no data is ever lost and assume that the hard disk is potentially always busy. The initial setup of a DMA transfer takes 1000 clock cycles for the processor, and the handling of the interrupt at DMA completion requires 500 clock cycles for the processor. The hard disk has a transfer rate of 4 MB/sec and uses DMA. Ignore any impact from bus contention between the processor and the DMA controller. Therefore, if the average transfer from the disk is 8 KB, the fraction of the 500-MHz processor consumed if the disk is actively transferring 100% of the time is 0.2%.

8.39 [8] <§8.6> Suppose we have the same hard disk and processor we used in Exercise 8.18, but we use interrupt-driven I/O. The overhead for each transfer, including the interrupt, is 500 clock cycles. Find the fraction of the processor consumed if the hard disk is only transferring data 5% of the time.

8.40 [8] <§8.6> Suppose we have the same processor and hard disk as in Exercise 8.18. Assume that the initial setup of a DMA transfer takes 1000 clock cycles for the processor, and assume the handling of the interrupt at DMA completion requires 500 clock cycles for the processor. The hard disk has a transfer rate of 4 MB/sec and uses DMA. If the average transfer from the disk is 8 KB, what fraction of the 500-MHz processor is consumed if the disk is actively transferring 100% of the time? Ignore any impact from bus contention between the processor and DMA controller.

8.41 [2 days–1 week] <§8.6, Appendix A>  [For More Practice](#): Using SPIM to Explore I/O

8.42 [3 days–1 week] <§8.6, Appendix A>  [For More Practice](#): Writing Code to Perform I/O

8.43 [3 days–1 week] <§8.6, Appendix A>  [For More Practice](#): Writing Code to Perform I/O

8.44 [15] <§§8.3–8.7> Redo the example on page 601, but instead assume that the reads are random 8-KB reads. You can assume that the reads are always to an idle disk, if one is available.

8.45 [20] <§§8.3–8.7> Here are a variety of building blocks used in an I/O system that has a synchronous processor-memory bus running at 800 MHz and one or more I/O adapters that interface I/O buses to the processor-memory bus.

- *Memory system*: The memory system has a 32-bit interface and handles four-word transfers. The memory system has separate address and data lines and, for writes to memory, accepts a word every clock cycle for 4 clock cycles and then takes an additional 4 clock cycles before the words have been stored and it can accept another transaction.

- *DMA interfaces:* The I/O adapters use DMA to transfer the data between the I/O buses and the processor-memory bus. The DMA unit arbitrates for the processor-memory bus and sends/receives four-word blocks from/to the memory system. The DMA controller can accommodate up to eight disks. Initiating a new I/O operation (including the seek and access) takes 0.1 ms, during which another I/O cannot be initiated by this controller (but outstanding operations can be handled).
- *I/O bus:* The I/O bus is a synchronous bus with a sustainable bandwidth of 100 MB/sec; each transfer is one word long.
- *Disks:* The disks have a measured average seek plus rotational latency of 8 ms. The disks have a read/write bandwidth of 40 MB/sec, when they are transferring.

Find the time required to read a 16 KB sector from a disk to memory, assuming that this is the only activity on the bus.

8.46 [5] <§8.7> In order to perform a disk or network access, it is typically necessary for the user to have the operating system communicate with the disk or network controllers. Suppose that in a particular 5 GHz computer, it takes 10,000 cycles to trap to the OS, 20 ms for the OS to perform a disk access, and 25 μ s for the OS to perform a network access. In a disk access, what percentage of the delay time is spent in trapping to the OS? How about in a network access?

8.47 [5] <§8.7> Suppose that in the computer in Exercise 8.46 we can somehow reduce the time for the OS to communicate with the disk controller by 60%, and we can reduce the time for the OS to communicate with the network by 40%. By what percentage can we reduce the total time for a network access? By what percentage can we reduce the total time for a disk access? Is it worthwhile for us to spend a lot of effort improving the OS trap latency in a computer that performs many disk accesses? How about in a computer that performs many network accesses?

§8.2, Page 580: Dependability: 2 and 3. RAID: All are true.

§8.3, Page 8.3-10: 1.

§8.4, Page 587: 1 and 2.

§8.5, Page 597: 1 and 2.

§8.6, Page 600: 1 and 2.

**Answers To
Check
Yourself**

Computers in the Real World

Saving Lives through Better Diagnosis

Problem: Find a way to examine internal organs to diagnose psychological problems without the use of invasive surgery or harmful radiation.

Solution: The development of magnetic resonance imaging (MRI), a three-dimensional scanning technology, has been one of the most important breakthroughs in modern medical technology. MRI uses a combination of radio-frequency pulses and magnetic fields to scan tissue. The organ to be imaged is scanned in a series of two-dimensional slices, which are then composed to create a three-dimensional image.

In addition to this computationally intensive task of composing the slices to create a volumetric image, extensive computation is used to extract the initial two-dimensional images, since the signal-to-noise ratio is often

low. The development of MRI has allowed the scanning of soft tissues, such as the brain, for which X-rays are not as effective and exploratory surgery is dangerous. Without a cost-effective computing capability, MRI would remain slow and expensive.

The two illustrations shows a series of MRI images of the human brain; the images below represent two-dimensional slices, while those on the facing page show a three-dimensional reconstruction. Once an image is in digital form, a physician can manipulate the image, removing outer layers, examining the image from different viewpoints, or looking at the three-dimensional structure to help in diagnosis.

The major benefits of MRI are twofold:

- It can reduce the need for unnecessary exploratory surgery. A physician may be able to determine whether a patient ex-



MRI images of a human brain, in two-dimensional view

perienicing headaches has a brain tumor, which requires surgery, or simply needs medication for a headache.

- By providing a surgeon with an accurate three-dimensional image, MRI can improve the surgical planning process and hence the outcome. For example, in operating on the brain to remove a tumor without accurate images of the tumor, the surgeon likely would have to enter the brain and then create a plan on the fly depending on the size and exact placement of the tumor. Furthermore, minimally invasive techniques (e.g. endoscopic surgery), which have become quite effective, would be impossible without accurate images.

There are many new interesting uses of MRI technology, which rely on faster and more cost effective computing. Some of the most promising are

- real-time imaging of the heart and blood vessels to enhance diagnosis of cardiac and cardiovascular disease;
- Combining real-time images and MRI images during surgery to help surgeons

accurately perform surgery, particularly when using minimally invasive techniques.

- Functional MRI (fMRI): a new type of application that uses MRI to examine brain function, primarily by analyzing blood flow in various portions of the brain. fMRI is being used for a number of applications, including exploring the physiological bases for cognitive problems such as dyslexia, pain management, planning for neurosurgery, and understanding neurological disorders.

To learn more see these references on the  library

MRI scans from the National Institutes of Health's Visible Human project

Principles of MRI and its application to medical imaging (long and reasonably detailed, but only a little mathematics)

Using MRI to do real-time cardiac imaging and angiography (imaging of blood vessels)

Functional MRI, www.fmri.org/fmri.htm

Visualization and imaging (including MRI and CT images): high-performance computing for complex images



MRI images of a human brain in three dimensions

Glossary

absolute address A variable's or routine's actual address in memory.

abstraction A model that renders lower-level details of computer systems temporarily invisible in order to facilitate design of sophisticated systems.

acronym A word constructed by taking the initial letters of string of words. For example: RAM is an acronym for Random Access Memory, and CPU is an acronym for Central Processing Unit.

active matrix display A liquid crystal display using a transistor to control the transmission of light at each individual pixel.

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

address A value used to delineate the location of a specific data element within a memory array.

addressing mode One of several addressing regimes delimited by their varied use of operands and/or addresses.

advanced load In IA-64, a speculative load instruction with support to check for aliases that could invalidate the load.

aliasing A situation in which the same object is accessed by two addresses; can occur in virtual memory when there are two virtual addresses for the same physical page.

alignment restriction A requirement that data be aligned in memory on natural boundaries

Amdahl's law A rule stating that the performance enhancement possible with a given improvement is limited by the amount that the improved feature is used.

antidependence Also called **name dependence**. An ordering forced by the reuse of a name, typically a register, rather than by a true dependence that carries a value between two instructions.

antifuse A structure in an integrated circuit that when programmed makes a permanent connection between two wires.

application binary interface (ABI) The user portion of the instruction set plus the operating system interfaces used by application programmers. Defines a standard for binary portability across computers.

architectural registers The instruction set visible registers of a processor; for example, in MIPS, these are the 32 integer and 16 floating-point registers.

arithmetic mean The average of the execution times that is directly proportional to total execution time.

assembler directive An operation that tells the assembler how to translate a program but does not produce machine instructions; always begins with a period.

assembler A program that translates a symbolic version of instructions into the binary version.

assembly language A symbolic language that can be translated into binary.

asserted signal A signal that is (logically) true, or 1.

asynchronous bus A bus that uses a handshaking protocol for coordinating usage rather than a clock; can accommodate a wide variety of devices of differing speeds.

atomic swap operation An operation in which the processor can both read a location and write it in the same bus operation, preventing any other processor or I/O device from reading or writing memory until it completes.

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.

backplane bus A bus that is designed to allow processors, memory, and I/O devices to coexist on a single bus.

barrier synchronization A synchronization scheme in which processors wait at the barrier and do not proceed until every processor has reached it.

basic block A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning).

behavioral specification Describes how a digital system operates functionally.

biased notation A notation that represents the most negative value by $00 \dots 000_{\text{two}}$ and the most positive value by $11 \dots 11_{\text{two}}$, with 0 typically having the value $10 \dots 00_{\text{two}}$, thereby biasing the number such that the number plus the bias has a nonnegative representation.

binary digit Also called a bit. One of the two numbers in base 2 (0 or 1) that are the components of information.

bit error rate The fraction in bits of a message or collection of messages that is incorrect.

block The minimum unit of information that can be either present or not present in the two-level hierarchy.

blocking assignment In Verilog, an assignment that completes before the execution of the next statement.

branch delay slot The slot directly after a delayed branch instruction, which in the MIPS architecture is filled by an instruction that does not affect the branch.

branch not taken A branch where the branch condition is false and the program counter (PC) becomes the address of the instruction that sequentially follows the branch.

branch prediction A method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

branch prediction buffer Also called **branch history table**. A small memory that is indexed by the lower portion of the address of the branch instruction and that contains one or more bits indicating whether the branch was recently taken or not.

branch taken A branch where the branch condition is satisfied and the program counter (PC) becomes the branch target. All unconditional branches are taken branches.

branch target address The address specified in a branch, which becomes the new program counter (PC) if the branch is taken. In the MIPS architecture the branch target is given by the sum of the offset field of the instruction and the address of the instruction following the branch.

branch target buffer A structure that caches the destination PC or destination instruction for a branch. It is usually organized as a

cache with tags, making it more costly than a simple prediction buffer.

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.

bus master A unit on the bus that can initiate bus requests.

bus transaction A sequence of bus operations that includes a request and may include a response, either of which may carry data. A transaction is initiated by a single request and may take many individual bus operations.

cache coherency Consistency in the value of data between the versions in the caches of several processors.

cache coherent NUMACC-NUMA A non-uniform memory access multiprocessor that maintains coherence for all caches.

cache memory A small, fast memory that acts as a buffer for a slower, larger memory.

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

callee A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

callee-saved register A register saved by the routine making a procedure call.

caller The program that instigates a procedure and provides the necessary parameter values.

caller-saved register A register saved by the routine being called.

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

carrier signal A continuous signal of a single frequency capable of being modulated by a second data-carrying signal.

cathode ray tube (CRT) display A display, such as a television set, that displays an image using an electron beam scanned across a screen.

central processor unit (CPU) Also called processor. The active part of the computer, which contains the datapath and control and which adds numbers, tests numbers, signals I/O devices to activate, and so on.

clock cycle Also called tick, clock tick, clock period, clock, cycle. The time for one clock period, usually of the processor clock, which runs at a constant rate.

clock cycles per instruction (CPI) Average number of clock cycles per instruction for a program or program fragment.

clock period The length of each clock cycle.

clock skew The difference in absolute time between the times when two state elements see a clock edge.

clocking methodology The approach used to determine when data is valid and stable relative to the clock.

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

combinational logic A logic system whose blocks do not contain memory and hence compute the same output given the same input.

commit unit The unit in a dynamic or out-of-order execution pipeline that decides when it is safe to release the result of an operation to programmer-visible registers and memory.

compiler A program that translates high-level language statements into assembly language statements.

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.

conditional branch An instruction that requires the comparison of two values and

that allows for a subsequent transfer of control to a new address in the program based on the outcome of the comparison.

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.

constellation A cluster that uses an SMP as the building block.

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.

control The component of the processor that commands the datapath, memory, and I/O devices according to the instructions of the program.

control hazard Also called **branch hazard**. An occurrence in which the proper instruction cannot execute in the proper clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

control signal A signal used for multiplexor selection or for directing the operation of a functional unit; contrasts with a data signal, which contains information that is operated on by a functional unit.

correlating predictor A branch predictor that combines local behavior of a particular branch and global information about the behavior of some recent number of executed branches.

CPU execution time Also called **CPU time**. The actual time the CPU spends computing for a specific task.

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

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.

data hazard Also called pipeline data hazard. An occurrence in which a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.

data parallelism Parallelism achieved by having massive data.

data rate Performance measure of bytes per unit time, such as GB/second.

data segment The segment of a UNIX object or executable file that contains a binary representation of the initialized data used by the program.

data transfer instruction A command that moves data between memory and registers.

datapath The component of the processor that performs arithmetic operations.

datapath element A functional unit used to operate on or hold data within a processor. In the MIPS implementation the datapath elements include the instruction and data memories, the register file, the arithmetic logic unit (ALU), and adders.

deasserted signal A signal that is (logically) false, or 0.

decoder A logic block that has an n -bit input and 2^n outputs where only one output is asserted for each input combination.

defect A microscopic flaw in a wafer or in patterning steps that can result in the failure of the die containing that defect.

delayed branch A type of branch where the instruction immediately following the branch is always executed, independent of whether the branch condition is true or false.

desktop computer A computer designed for use by an individual, usually incorporating a graphics display, keyboard, and mouse.

die The individual rectangular sections that are cut from a wafer, more informally known as chips.

DIMM (dual inline memory module) A small board that contains DRAM chips on both sides. SIMMs have DRAMs on only one side. Both DIMMs and SIMMs are meant to be plugged into memory slots, usually on a motherboard.

direct memory access (DMA) A mechanism that provides a device controller the ability to transfer data directly to or from the memory without involving the processor.

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

directory A repository for information on the state of every block in main memory, including which caches have copies of the block, whether it is dirty, and so on. Used for cache coherence.

dispatch An operation in a microprogrammed control unit in which the next microinstruction is selected on the basis of one or more fields of a macroinstruction, usually by creating a table containing the addresses of the target microinstructions and indexing the table using a field of the macroinstruction. The dispatch tables are typically implemented in ROM or programmable logic array (PLA). The term *dispatch* is also used in dynamically scheduled processors to refer to the process of sending an instruction to a queue.

distributed memory Physical memory that is divided into modules, with some placed near each processor in a multiprocessor.

distributed shared memory (DSM) A memory scheme that uses addresses to access remote data when demanded rather than retrieving the data in case it might be used.

dividend A number being divided.

divisor A number that the dividend is divided by.

don't-care term An element of a logical function in which the output does not depend on the values of all the inputs. Don't-care terms may be specified in different ways.

double precision A floating-point value represented in two 32-bit words.

dynamic branch prediction Prediction of branches at runtime using runtime information.

dynamic multiple issue An approach to implementing a multiple-issue processor where many decisions are made during execution by the processor.

dynamic pipeline scheduling Hardware support for reordering the order of instruction execution so as to avoid stalls.

dynamic random access memory (DRAM) Memory built as an integrated circuit, it provides random access to any location.

edge-triggered clocking A clocking scheme in which all state changes occur on a clock edge.

embedded computer A computer inside another device used for running one predetermined application or collection of software.

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

Ethernet A computer network whose length is limited to about a kilometer. Originally capable of transferring up to 10 million bits per second, newer versions can run up to 100 million bits per second and even 1000 million bits per second. It treats the wire like a bus with multiple masters and uses collision detection and a back-off scheme for handling simultaneous accesses.

exception Also called interrupt. An unscheduled event that disrupts program execution; used to detect overflow.

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.

executable file A functional program in the format of an object file that contains no unresolved references, relocation information, symbol table, or debugging information.

exponent In the numerical representation system of floating-point arithmetic, the value that is placed in the exponent field.

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.

false sharing A sharing situation in which 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.

field programmable devices (FPD) An integrated circuit containing combinational logic, and possibly memory devices, that is configurable by the end user.

field programmable gate array A configurable integrated circuit containing both combinational logic blocks and flip-flops.

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.

firmware Microcode implemented in a memory structure, typically ROM or RAM.

flat panel display, liquid crystal display A display technology using a thin layer of liquid polymers that can be used to transmit or block light according to whether a charge is applied.

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.

floating point Computer arithmetic that represents numbers in which the binary point is not fixed.

floppy disk A portable form of secondary memory composed of a rotating mylar platter coated with a magnetic recording material.

flush (instructions) To discard instructions in a pipeline, usually due to an unexpected event.

formal parameter A variable that is the argument to a procedure or macro; replaced by that argument once the macro is expanded.

forward reference A label that is used before it is defined.

forwarding Also called **bypassing**. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible registers or memory.

fraction The value, generally between 0 and 1, placed in the fraction field.

frame pointer A value denoting the location of the saved registers and local variables for a given procedure.

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

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

gate A device that implements basic logic functions, such as AND or OR.

general-purpose register (GPR) A register that can be used for addresses or for data with virtually any instruction.

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

global pointer The register that is reserved to point to static data.

guard The first of two extra bits kept on the right during intermediate calculations of floating-point numbers; used to improve rounding accuracy.

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

handshaking protocol A series of steps used to coordinate asynchronous bus transfers in which the sender and receiver proceed to the next step only when both parties agree that the current step has been completed.

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.

hardware synthesis tools Computer-aided design software that can generate a gate-level design based on behavioral descriptions of a digital system.

hardwired control An implementation of finite state machine control typically using programmable logic arrays (PLAs) or collections of PLAs and random logic.

hexadecimal Numbers in base 16.

high-level programming language A portable language such as C, Fortran, or Java composed of words and algebraic notation that can be translated by a compiler into assembly language.

hit rate The fraction of memory accesses found in a cache.

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.

hold time The minimum time during which the input must be valid after the clock edge.

hot swapping Replacing a hardware component while the system is running.

I/O instructions A dedicated instruction that is used to give a command to an I/O device and that specifies both the device number and the command word (or the location of the command word in memory).

I/O rate Performance measure of I/Os per unit time, such as reads per second.

I/O requests Reads or writes to I/O devices.

implementation Hardware that obeys the architecture abstraction.

imprecise interrupt Also called imprecise exception. Interrupts or exceptions in pipelined computers that are not associated with the exact instruction that was the cause of the interrupt or exception.

in-order commit A commit in which the results of pipelined execution are written to the programmer-visible state in the same order that instructions are fetched.

input device A mechanism through which the computer is fed information, such as the keyboard or mouse.

instruction format A form of representation of an instruction composed of fields of binary numbers.

instruction group In IA-64, a sequence of consecutive instructions with no register data dependences among them.

instruction latency The inherent execution time for an instruction.

instruction mix A measure of the dynamic frequency of instructions across one or many programs.

instruction set architecture Also called architecture. An abstract interface between the hardware and the lowest level software of a machine that encompasses all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory access, I/O, and so on.

instruction set The vocabulary of commands understood by a given architecture.

instruction-level parallelism The parallelism among instructions.

integrated circuit Also called **chip**. A device combining dozens to millions of transistors.

interrupt An exception that comes from outside of the processor. (Some architectures use the term *interrupt* for all exceptions.)

interrupt-driven I/O An I/O scheme that employs interrupts to indicate to the processor that an I/O device needs attention.

interrupt handler A piece of code that is run as a result of an exception or an interrupt.

issue packet The set of instructions that issues together in 1 clock cycle; the packet may be determined statically by the compiler or dynamically by the processor.

issue slots The positions from which instructions could issue in a given clock cycle; by analogy these correspond to positions at the starting blocks for a sprint.

Java bytecode Instruction from an instruction set designed to interpret Java programs.

Java Virtual Machine (JVM) The program that interprets Java bytecodes.

jump address table Also called **jump table**. A table of addresses of alternative instruction sequences.

jump-and-link instruction An instruction that jumps to an address and simultaneously saves the address of the following instruction in a register (\$ra in MIPS).

Just In Time Compiler (JIT) The name commonly given to a compiler that operates at runtime, translating the interpreted code segments into the native code of the computer.

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

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.

latency (pipeline) The number of stages in a pipeline or the number of stages between two instructions during execution.

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

least significant bit The rightmost bit in a MIPS word.

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.

linker Also called **link editor**. A systems program that combines independently assembled machine language programs and resolves all undefined labels into an executable file.

loader A systems program that places an object program in main memory so that it is ready to execute.

load-store machine Also called **register-register machine**. An instruction set architecture in which all operations are between registers and data memory may only be accessed via loads or stores.

load-use data hazard A specific form of data hazard in which the data requested by a load instruction has not yet become available when it is requested.

local area network (LAN) A network designed to carry data within a geographically confined area, typically within a single building.

local label A label referring to an object that can be used only within the file in which it is defined.

local miss rate The fraction of references to one level of a cache that miss; used in multi-level hierarchies.

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

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.

loop unrolling A technique to get more performance from loops that access arrays, in which multiple copies of the loop body are made and instructions from different iterations are scheduled together.

machine language Binary representation used for communication within a computer system.

macro A pattern-matching and replacement facility that provides a simple mechanism to name a frequently used sequence of instructions.

magnetic disk (also called hard disk) A form of nonvolatile secondary memory composed of rotating platters coated with a magnetic recording material.

megabyte Traditionally 1,048,576 (2^{20}) bytes, although some communications and secondary storage systems have redefined it to mean 1,000,000 (10^6) bytes.

memory The storage area in which programs are kept when they are running and that contains the data needed by the running programs.

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

memory-mapped I/O An I/O scheme in which portions of address space are assigned to I/O devices and reads and writes to those addresses are interpreted as commands to the I/O device.

MESI cache coherency protocol A write-invalidate protocol whose name is an acronym for the four states of the protocol: Modified, Exclusive, Shared, Invalid.

message passing Communicating between multiple processors by explicitly sending and receiving information.

metastability A situation that occurs if a signal is sampled when it is not stable for the required set-up and hold times, possibly causing the sampled value to fall in the indeterminate region between a high and low value.

microarchitecture The organization of the processor, including the major functional units, their interconnection, and control.

microcode The set of microinstructions that control a processor.

microinstruction A representation of control using low-level instructions, each of which asserts a set of control signals that are active on a given clock cycle as well as specifies what microinstruction to execute next.

micro-operations The RISC-like instructions directly executed by the hardware in recent Pentium implementations.

microprogram A symbolic representation of control in the form of instructions, called microinstructions, that are executed on a simple micromachine.

microprogrammed control A method of specifying control that uses microcode rather than a finite state representation.

million instructions per second (MIPS) A measurement of program execution speed based on the number of millions of instructions. MIPS is computed as the instruction count divided by the product of the execution time and 10^6 .

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).

mirroring Writing the identical data to multiple disks to increase data availability.

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, and insert it in the level that experienced the miss.

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

most significant bit The leftmost bit in a MIPS word.

motherboard A plastic board containing packages of integrated circuits or chips, including processor, cache, memory, and connectors for I/O devices such as networks and disks.

multicomputer Parallel processors with multiple private addresses.

multicycle implementation Also called multiple clock cycle implementation. An implementation in which an instruction is executed in multiple clock cycles.

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

multiple issue A scheme whereby multiple instructions are launched in 1 clock cycle.

multiprocessor Parallel processors with a single shared address.

multistage network A network that supplies a small switch at each node.

NAND gate An inverted AND gate.

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.

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

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.

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

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

nonvolatile memory A form of memory that retains data even in the absence of a power source and that is used to store programs between runs. Magnetic disk is nonvolatile and DRAM is not.

nonvolatile Storage device where data retains its value even when power is removed.

nop An instruction that does no operation to change state.

NOR A logical bit-by-bit operation with two operands that calculates the NOT of the OR of the two operands.

NOR gate An inverted OR gate.

normalized A number in floating-point notation that has no leading 0s.

NOT A logical bit-by-bit operation with one operand that inverts the bits; that is, it replaces every 1 with a 0, and every 0 with a 1.

object-oriented language A programming language that is oriented around objects rather than actions, or data versus logic.

opcode The field that denotes the operation and format of an instruction.

operating system Supervising program that manages the resources of a computer for the benefit of the programs that run on that machine.

out-of-order execution A situation in pipelined execution when an instruction blocked from executing does not cause the following instructions to wait.

output device A mechanism that conveys the result of a computation to a user or another computer.

overflow (floating-point) A situation in which a positive exponent becomes too large to fit in the exponent field.

package Basically a directory that contains a group of related classes.

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

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

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

PC-relative addressing An addressing regime in which the address is the sum of the program counter (PC) and a constant in the instruction.

physical address An address in main memory.

physically addressed cache A cache that is addressed by a physical address.

pipeline stall Also called bubble. A stall initiated in order to resolve a hazard.

pipelining An implementation technique in which multiple instructions are overlapped in execution, much like to an assembly line.

pixel The smallest individual picture element. Screen are composed of hundreds of thousands to millions of pixels, organized in a matrix.

poison A result generated when a speculative load yields an exception, or an instruction uses a poisoned operand.

polling The process of periodically checking the status of an I/O device to determine the need to service the device.

precise interrupt Also called **precise exception**. An interrupt or exception that is

always associated with the correct instruction in pipelined computers.

predication A technique to make instructions dependent on predicates rather than on branches.

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.

primary memory Also called main memory. Volatile memory used to hold programs while they are running; typically consists of DRAM in today's computers.

procedure A stored subroutine that performs a specific task based on the parameters with which it is provided.

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.

procedure frame Also called **activation record**. The segment of the stack containing a procedure's saved registers and local variables.

processor-memory bus A bus that connects processor and memory and that is short, generally high speed, and matched to the memory system so as to maximize memory-processor bandwidth.

program counter (PC) The register containing the address of the instruction in the program being executed

programmable array logic (PAL) Contains a programmable and-plane followed by a fixed or-plane.

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 generating product terms of the inputs and input complements and the second generating sum

terms of the product terms. Hence, PLAs implement logic functions as a sum of products.

programmable logic device (PLD) An integrated circuit containing combinational logic whose function is configured by the end user.

programmable ROM (PROM) A form of read-only memory that can be programmed when a designer knows its contents.

propagation time The time required for an input to a flip-flop to propagate to the outputs of the flip-flop.

protected A Java keyword that restricts invocation of a method to other methods in that package.

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.

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

pseudoinstruction A common variation of assembly language instructions often treated as if it were an instruction in its own right.

public A Java keyword that allows a method to be invoked by any other method.

quotient The primary result of a division; a number that when multiplied by the divisor and added to the remainder produces the dividend.

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.

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

recursive procedures Procedures that call themselves either directly or indirectly through a chain of calls.

redundant arrays of inexpensive disks (RAID) An organization of disks that uses an array of small and inexpensive disks so as to increase both performance and reliability.

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.

reg In Verilog, a register.

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.

register renaming The renaming of registers, by the compiler or hardware, to remove antidependences.

register-use convention Also called **procedure call convention**. A software protocol governing the use of registers by procedures.

relocation information The segment of a UNIX object file that identifies instructions and data words that depend on absolute addresses.

remainder The secondary result of a division; a number that when added to the product of the quotient and the divisor produces the dividend.

reorder buffer The buffer that holds results in a dynamically scheduled processor until it is safe to store the results to memory or a register.

reservation station A buffer within a functional unit that holds the operands and the operation.

response time Also called **execution time**. The total time required for the computer to complete a task, including disk accesses,

memory accesses, I/O activities, operating system overhead, CPU execution time, and so on.

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

return address A link to the calling site that allows a procedure to return to the proper address; in MIPS it is stored in register `$ra`.

rotation latency Also called 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.

round Method to make the intermediate floating-point result fit the floating-point format; the goal is typically to find the nearest number that can be represented in the format.

scientific notation A notation that renders numbers with a single digit to the left of the decimal point.

secondary memory Nonvolatile memory used to store programs and data between runs; typically consists of magnetic disks in today's computers.

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.

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

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.

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.

semiconductor A substance that does not conduct electricity well.

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

sensitivity list The list of signals that specifies when an always block should be reevaluated.

separate compilation Splitting a program across many files, each of which can be compiled without knowledge of what is in the other files.

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.

Server A computer used for running larger programs for multiple users often simultaneously and typically accessed only via a network.

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

set-up time The minimum time that the input to a memory device must be valid before the clock edge.

shared memory A memory for a parallel processor with a single address space, implying implicit communication with loads and stores.

sign-extend To increase the size of a data item by replicating the high-order sign bit of the original data item in the high-order bits of the larger, destination data item.

silicon A natural element which is a semiconductor.

silicon crystal ingot A rod composed of a silicon crystal that is between 6 and 12 inches in diameter and about 12 to 24 inches long.

simple programmable logic device (SPLD) Programmable logic device usually containing either a single PAL or PLA.

single precision A floating-point value represented in a single 32-bit word.

single-cycle implementation Also called single clock cycle implementation. An implementation in which an instruction is executed in one clock cycle.

small computer systems interface (SCSI) A bus used as a standard for I/O devices.

snooping cache coherency A method for maintaining cache coherency in which all cache controllers monitor or snoop on the bus to determine whether or not they have a copy of the desired block.

source language The high-level language in which a program is originally written.

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

speculation An approach whereby the compiler or processor guesses the outcome of an instruction to remove it as a dependence in executing other instructions.

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.

split transaction protocol A protocol in which the bus is released during a bus transaction while the requester is waiting for the data to be transmitted, which frees the bus for access by another requester.

stack pointer A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found.

stack segment The portion of memory used by a program to hold procedure call frames.

stack A data structure for spilling registers organized as a last-in-first-out queue.

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

state element A memory element.

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.

static method A method that applies to the whole class rather than to an individual object. It is unrelated to static in C.

static multiple issue An approach to implementing a multiple-issue processor where many decisions are made by the compiler before execution.

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.

sticky bit A bit used in rounding in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit.

stop In IA-64, an *explicit indicator of a break between independent and dependent instructions*.

stored-program concept The idea that instructions and data of many types can be stored in memory as numbers, leading to the stored program computer.

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

structural hazard An occurrence in which a planned instruction cannot execute in the proper clock cycle because the hardware cannot support the combination of instructions that are set to execute in the given clock cycle.

structural specification Describes how a digital system is organized in terms of a hierarchical connection of elements.

sum of products A form of logical representation that employs a logical sum (OR) of products (terms joined using the AND operator).

supercomputer A class of computers with the highest performance and cost; they are configured as servers and typically cost millions of dollars.

superscalar An advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle.

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

switched network A network of dedicated point-to-point links that are connected to each other with a switch.

symbol table A table that matches names of labels to the addresses of the memory words that instructions occupy.

symmetric multiprocessor (SMP) or uniform memory access (UMA) A multiprocessor in which accesses to main memory take the same amount of time no matter which processor requests the access and no matter which word is asked.

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

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.

synchronous bus A bus that includes a clock in the control lines and a fixed protocol for communicating that is relative to the clock.

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.

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.

system CPU time The CPU time spent in the operating system performing tasks on behalf of the program.

system performance evaluation cooperative (SPEC) benchmark A set of standard CPU-intensive, integer and floating point benchmarks based on real programs.

systems software Software that provides services that are commonly useful, including operating systems, compilers, and assemblers.

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.

temporal locality The principle stating that if a data location is referenced then it will tend to be referenced again soon.

terabyte Originally 1,099,511,627,776 (2^{40}) bytes, although some communications and secondary storage systems have redefined it to mean 1,000,000,000,000 (10^{12}) bytes.

text segment The segment of a UNIX object file that contains the machine language code for routines in the source file.

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.

tournament branch predictor A branch predictor with multiple predictions for each branch and a selection mechanism that chooses which predictor to enable for a given branch.

trace cache An instruction cache that holds a sequence of instructions with a given starting address; in recent Pentium implementations the trace cache holds microoperations rather than IA-32 instructions.

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

transaction processing A type of application that involves handling small short operations (called transactions) that typically require both I/O and computation. Trans-

action processing applications typically have both response time requirements and a performance measurement based on the throughput of transactions.

transistor An on/off switch controlled by an electric signal.

translation-lookaside buffer (TLB) A cache that keeps track of recently used address mappings to avoid an access to the page table.

underflow (floating-point) A situation in which a negative exponent becomes too large to fit in the exponent field.

units in the last place (ulp) The number of bits in error in the least significant bits of the significand between the actual number and the number that can be prepresented.

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

unresolved reference A reference that requires more information from an outside source in order to be complete.

untaken branch One that falls through to the successive instruction. A taken branch is one that causes transfer to the branch target.

user CPU time The CPU time spent in a program itself.

vacuum tube An electronic component, predecessor of the transistor, that consists of a hollow glass tube about 5 to 10 cm long from which as much air has been removed as possible and which uses an electron beam to transfer data.

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

vector processor An architecture and compiler model that was popularized by supercomputers in which high-level operations work on linear arrays of numbers.

vectored interrupt An interrupt for which the address to which control is transferred is

determined by the cause of the exception.

verilog One of the two most common hardware description languages.

very large scale integrated (VLSI) circuit A device containing hundreds of thousands to millions of transistors.

VHDL One of the two most common hardware description languages.

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.

virtual machine A virtual computer that appears to have nondelayed branches and loads and a richer instruction set than the actual hardware.

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

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

volatile memory Storage, such as DRAM, that only retains data only if it is receiving power.

wafer A slice from a silicon ingot no more than 0.1 inch thick, used to create chips.

weighted arithmetic mean An average of the execution time of a workload with weighting factors designed to reflect the presence of the programs in a workload; computed as the sum of the products of weighting factors and execution times.

wide area network A network extended over hundreds of kilometers which can span a continent.

wire In Verilog, specifies a combinational signal.

word The natural unit of access in a computer, usually a group of 32 bits; corresponds to the size of a register in the MIPS architecture.

workload A set of programs run on a computer that is either the actual collection of applications run by a user or is constructed from real programs to approximate such a mix. A typical workload specifies both the programs as well as the relative frequencies.

write buffer A queue that holds data while the data are 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.

write-invalidate A type of snooping protocol in which the writing processor causes all copies in other caches to be invalidated before changing its local copy, which allows it to update the local data until another processor asks for it.

write-through A scheme in which writes always update both the cache and the memory, ensuring that data is always consistent between the two.

yield The percentage of good dies from the total number of dies on the wafer.

Index

CD information is listed by chapter and section number followed by page ranges (CD9.1:1-2). In More Depth references are listed by chapter number followed by page ranges (IMD4:5-6). Page references preceded by a single letter refer to appendices.

A

Absolute addresses, A13

Abstractions, 21–22, 24

Accumulator architectures, CD2.19:1–2

Accumulator instructions, IMD2:7

Acronyms, 9–10

ACS, CD6.13:4

Activation record, 86

Active matrix display, 18

Ada, 173

add, 49–51, 301

Adder, 292

add immediate, 58

add immediate unsigned, 172

Addition, 170–176

 carry lookahead, B38–47

 floating point, 197–201

Address (addressing)

 absolute, A13

 base, 55

 calculation, 385, 390, 392, 402

 exception, 342–343

 in large-scale parallel processors,

 CD9.4:23–25

 memory, 54

 PC-relative, 98

 physical, 511, 512, 513–514

 translation, 512, 521–524

 virtual, 512

Addressing, MIPS

 branches and jumps, 97–99, 294–295

 decoding machine language, 100–104

 mode summary, 100

 32-bit immediate operands, 95–96

Addressing modes

 IA-32, 138

 MIPS, 100

 RISC, D5–9

add unsigned, 172

Advanced Research Project Agency
(ARPA), CD7.9:9, CD8.3:5,

 CD8.11:7

Advance load, 442

Agarwala, Tilak, CD6.13:4

Aho, Al, CD2.19:8

Aiken, Howard, CD1.7:3

Air bags, 281

Algol, CD2.19:6–7

Aliasing, 528

Alignment restriction, 56

Allan, Fran, CD2.19:8

Allocate-on-miss, 484

Alpha architecture, CD5.12:3, D27–28

Alto, 16, CD1.7:7–8, CD7.9:10, CD8.11:7

ALU. *See* Arithmetic logic unit

ALUOp, 301–305

ALUOut, 319, 320, 327

AMD, 136

Amdahl, Gene, CD5.12:1

Amdahl's law, 179, 267, 494, CD9.2:9,
 CD9.9:40, IMD4:5–6

AMD Opteron, memory hierarchy,
 546–550

and (AND), 70, 301, 321, B6

AND gate, CD3.10:5

and immediate, 71

Andreessen, Marc, CD8.11:7

Antidependence, 439

Antifuse, B77

Antilock brakes, 281

Apple II, CD1.7:5

Application binary interface (ABI), 22

Applications software, 11

Archeological sites, 236–237

Architectural registers, 448

Architecture. *See* Instruction set
 architecture

Arithmetic

 addition, 170–176

 division, 183–189

 fallacies and pitfalls, 220–224

 floating point, 189, 191–220

 mean, 257–258

 multiplication, 176–182

 signed and unsigned numbers,
 160–170

 subtraction, 170–176

Arithmetic-logical instructions,
 292–293, 298

 multiple-cycle implementation,
 327, 329

 single-cycle implementation, 300–318

Arithmetic logic unit (ALU), 177, 179,
 184, 187, 201

 adders and, 292, 294

 ALUOp, 301–305

 ALUOut, 319, 320, 327

 constructing, B26–38

 control, 301–303, C4–8

datapaths and, 286, 292, 294, 296
 MIPS, B32–38
 multicycle implementation, 318–340
 1-bit, B26–29
 single-cycle implementation, 300–318
 32-bit, B29–36
 ARM, D36–38
 ARPANET, CD8.3:5, CD8.11:7
 Arrays
 of logic elements, B18–19
 versus pointers, 130–134
 Art, restoration of, 562–563
 ASCII (American Standard Code for Information Interchange), 90–91
 versus binary numbers, 162
 Assembler directives, A5
 Assemblers, 13, 107–108, A4, 10–17
 Assembly language, 13, 107, A3–10
 See also MIPS assembly language
 disadvantages of, A9–10
 when to use, A7–9
 Asserted signal, 290, B4
 Assert signal, 290
 Associativity, in caches, 499–502
 Asynchronous bus, 582–583
 Asynchronous inputs, B75–77
 Atanasoff, John, CD1.7:3
 AT&T Bell Labs, CD7.9:8–9
 Atomic swap operation, CD9.3:18
 Automatic storage class, 85
 Availability, 573
 Average Memory Access Time (AMAT),
 IMD7:1

B

Bachman, Charles, CD8.11:4, 5
 Backpatching, A13
 Backplane, 582
 Backus, John, CD2.19:6, 7
 Barrier synchronization, CD9.3:15
 Base address, 55, 100
 Base register, 55
 Base stations, CD8.3:9
 Base 2 to represent numbers, 160–161
 Basic block, 75
 Basket, Forrest, CD7.9:9
 Behavioral specification, B21
 Bell Labs, CD7.9:8–9
 Benchmarks, 254–255
 EEMBC, 255, IMD4:17–18
 kernel, CD4.7:2, IMD4:7–8
 SPEC CPU, 254–255, 259–266,
 CD4.7:2–3, IMD4:7–8
 SPECweb99, 262–266
 synthetic, CD4.7:1–2, IMD4:11–12
 Berkeley Computer Corp. (BCC),
 CD7.9:8, 9
 Berkeley Software Distribution (BSD),
 CD7.9:9
 Berners-Lee, Tim, CD8.11:7
 Biased notation, 170, 194
 Bigelow, Julian, CD1.7:3
 Big Endian, 56, A43
 Big-interleaved parity (RAID 3),
 576–577
 BINAC, CD1.7:4
 Binary digits (numbers), 12, 60
 adding and subtracting, 170–176
 ASCII versus, 162
 converting to decimal floating
 point, 196
 converting to decimals, 164
 hexadecimal-binary conversion
 table, 62
 scientific notation, 191
 use of, 160–161
 Binary point, 191
 Bit(s), 12, 60
 in a cache, 479
 dirty, 521
 fields, IMD2:13–14
 least significant, 161
 map, 18
 most significant, 161
 reference/use, 519
 sign, 163
 sticky, 215
 Bit error rate (BER), CD8.3:9
 Blaauw, Gerrit, CD6.13:2
 Block, Barbara, 156–157
 Blocking assignment, B24
 Block-interleaved parity (RAID 4),
 577–578
 Blocks
 defined, 470
 finding, 540–541
 locating in caches, 502–504
 placement of, 538–540
 reducing cache misses with, 496–502
 replacing, 504, 541–542
 Bonding, 30
 Boolean algebra, B6
 Booth's algorithm, IMD3:5–9
 Bounds check shortcut, 168
 Branch (es)
 addressing in, 97–99, 294–295
 delayed, 297, 382, 418–419, A41
 delay slot, 423
 history table, 421
 loop, 421–422
 multiple-cycle implementation,
 327–328, 336
 not taken, 295, 418
 prediction, 382, 421–423
 prediction buffer, 421
 taken, 295
 target address, 294–296
 target buffer, 423
 Branch equal (beq), 294, 297, 300–318
 Branch/control hazards, 379–382,
 416–424
 delayed, 297, 382, 418–419
 dynamic branch prediction, 421–423
 not taken, 295, 418
 untaken, 381
 Verilog and, CD6.7:8–9
 Brooks, Fred, Jr., CD6.13:2
 Bubble Sort, 129
 Burks, Arthur W., 48, CD1.7:3, CD3.10:1
 Buses, 291–292
 advantages/disadvantages of, 581
 asynchronous, 582–583
 backplane, 582
 basics of, 581–585
 defined, 581, B18–19
 master, 594
 Pentium 4, 585–587

- processor-memory or I/O, 582
- shared, 322–324
- synchronous, 582–583
- transaction, 582
- Bypassing, 376–377
- Byte addressing, 56
- Byte order, A43
- C**
- C
 - bit fields, IMD2.13–14
 - converting floating points to MIPS
 - assembly code, 209–213
 - development of, CD2.19:7
 - logical operations, 68–71
 - overflows, 172
 - procedures, 81–88
 - sort example, 121–129
 - strings, 92–93
 - translating hierarchy, 106–111
 - while loop in, 74–75
- Cache coherency
 - multiprocessor, CD9.3:12–20
 - protocols, CD9.3:13, 16–18
 - snooping, CD9.3:13
 - synchronization using, CD9.3:18–20
- Cache-coherent nonuniform memory
 - access (CC-NUMA), CD9.4:22
- Caches
 - accessing, 476–482
 - associativity, 499–502
 - basics of, 473–491
 - bits in, 479
 - blocks, locating in, 502–504
 - blocks used to reduce misses, 496–502
 - defined, 473
 - direct-mapped, 474–475, 497
 - example of simple, 474–476
 - fully associative, 497
 - Intrinsity FastMATH processor
 - example, 485–487
 - mapping address to multiword
 - block, 480
 - memory, 20
 - memory system design to support, 487–491
 - misses, handling, 482–483, 496–502
 - multilevel, 492, 505–510
 - nonblocking, 445, 548
 - performance, measuring and improving, 492–511
 - performance with increased clock rate, 495–496
 - reducing miss penalty using multi-level, 505–509
 - set associative, 497, 504
 - split, 487
 - tags, 475, 504
 - three Cs model, 543–545
 - valid bit, 476
 - writes, handling, 483–485
- Callee, 80, A23
- Caller, 80, A23
- Cal TSS, CD7.9:8
- Capacity misses, 543
- Carnegie Mellon University, CD6.13:5
- Carrier signal, CD8.3:8
- Carry lookahead, B38–47
- Carry save adders, 181, IMD3:17–18
- Case statement, 76
- Cathode ray tubes (CRTs), 18
- Cause register, 342
- CauseWrite, 342
- Central processor unit (CPU), 20
 - execution time, 244–245
 - performance, 245, 246–253
 - time, 244–245
- Cerf, Vint, CD8.11:7
- Chamberlin, Donald, CD8.11:5
- Characters, Java, 93–95
- Chavín de Huántar, 236–237
- Chips, 20, 30
- Clearing words in memory
 - arrays and, 130–132
 - comparing both methods, 133–134
 - pointers and, 132–133
- Clock cycles, 245, B47
 - finite state machines, 332
 - multicycle implementation, 318–340
 - single-cycle implementation, 300–318
- Clock cycles, breaking execution into
 - arithmetic-logical instruction, 327, 329
 - branches, 327–328
 - decode instruction and register fetch, 326–327
 - fetch instruction, 325–326
 - jump, 328
 - memory read, 329
 - memory reference, 327, 328
- Clock cycles per instruction (CPI), 248–251
 - in multicycle CPU, 330–331
- Clocking methodology, 290–292, B47
 - edge-triggered, 290–291, B47
 - level-sensitive, B74–75
 - timing methodologies, B72–77
- Clock period, 245, B47
- Clock rate, 245
- Clocks, B47–49
- Clock skew, B73–74
- CLU, CD2.19:7
- Clusters, CD9.1:4, CD9.5:25–26
- CMOS (Complementary Metal Oxide Semiconductor), 31, 264
- Coarse-grained multithreading, CD9.7:31–33
- Cobol, CD2.19:6, CD8.11:4
- Cocke, John, CD2.19:8, CD6.13:2, 4
- Codd, Ted, CD8.11:4, 5
- Code generation, CD2.12:9
- Code motion, 119
- Code size, fallacy of using, IMD4:18–19
- Coherence problem, 595
- Cold-start misses, 543
- Collision misses, 543
- Color, 292
- Colossus, CD1.7:3
- Combinational control units, C4–8
- Combinational elements, 289
- Combinational logic, B5, 8–20, 23–25
- Compact disks (CDs), 25

- Compaq Computers, CD8.11:6
 - Comments, 50
 - Commit unit, 443
 - Common subexpression elimination, 117
 - Compilers
 - C, 107
 - functions of, 11–12
 - historical development of, CD2.19:7–8
 - how they work, CD2.12:1–9
 - Java, 114–115
 - optimization, 116–121
 - structure of, 116
 - translating high-level language into instructions that hardware can execute, 12–15, A5–6
 - Compulsory misses, 543
 - Computers
 - applications, 5–7
 - components of, 15–16
 - historical development of, CD1.7:1–10
 - organization of, 16
 - what it looks like inside, 18–22
 - Computer technology, advances in, 4
 - Conditional branches, 72–73
 - Condition codes, 140
 - Conflict misses, 543
 - Constant folding, 118
 - Constant propagation, 118
 - Constants, 57, 58
 - loading 32-bit, 96
 - Constellations, CD9.5:26
 - Context switch, 530
 - Control, 20
 - hardwired, 348, CD5.12:2
 - pipelined, 399–402
 - Control Data Corp. (CDC), CD1.7:5, CD6.13:2
 - Control hazards. *See* Branch hazards
 - Controller time, 570
 - Control signals
 - list of, 306, 324
 - write, 290, 294
 - Control unit
 - adding, 299
 - combinational, C4–8
 - designing main, 303–312
 - exceptions, 340–346
 - fallacies and pitfalls, 350–352
 - finite state machines, 330, 331–340, C8–20
 - interrupts, 340–341
 - jumps, 313–314
 - microprogramming, 330, CD5.7:4–10
 - multicycle implementation, 318–340
 - single-cycle implementation, 300–318
 - Coonen, Jerome T., CD3.10:7
 - Copy back, 521
 - Copy propagation, 118
 - Corbato, John, CD7.9:7, 11
 - Correlating predictors, 423
 - Cosmic Cube, CD9.11:52
 - C++, CD2.19:7
 - CPU. *See* Central processor unit
 - Cray, Seymour, CD1.7:5, CD3.10:4, CD6.13:2
 - Cray Research, Inc., CD1.7:5, CD3.10:4–5, CD6.13:5
 - Critical word first, 482
 - Crossbar network, CD9.6:30
 - CTSS (Compatible Time-Sharing System), CD7.9:7–11
 - Culler, David, 157
 - Culler Scientific, CD6.13:4
 - Cutler, David, CD7.9:9
 - Cydrome Co., CD6.13:4, 5
 - Cylinder, use of term, 569
- D**
- Dahl, Ole-Johan, CD2.19:7
 - Databases, history of, CD8.11:4–5
 - Data General, CD8.11:6
 - Data hazards
 - defined, 376–379
 - forwarding, 402–412
 - load-use, 377
 - stalls, 413–416
 - Data parallelism, CD9.11:48
 - Datapath, 20
 - building a, 292–300
 - elements, 292
 - fallacies and pitfalls, 350–352
 - jumps, 313–314
 - logic design conventions, 289–292
 - multicycle implementation, 318–340
 - operation of, 306–312
 - pipelined, 384–399
 - single-cycle implementation, 300–318
 - Data rate, 598
 - Data segment, A13, 20
 - Data selector, 286
 - Data transfer instructions, 54–55
 - Data types, Verilog, B21–22
 - Dawson, Todd, 157
 - Dead code elimination, 118
 - Dead store elimination, 118
 - Deasserted signal, 290, B4
 - Deassert signal, 290
 - Debugging information, A13
 - DEC (Digital Equipment Corp.), CD1.7:5, CD4.7:2, CD7.9:9, CD8.11:11
 - Decimal numbers, 60, 161
 - converting binary numbers to, 164
 - converting binary to decimal floating point, 196
 - dividing, 183
 - multiplying, 176–177
 - scientific notation, 189
 - Decision-making instructions, 72–74
 - Decoders, B8–9
 - Decoding, 333
 - Dedicated register, CD2.19:2
 - Defects, 30
 - Delayed branch, 297, 382, 418–419, A41
 - Dell Computer Corp.
 - SPEC CPU benchmarks, 254–255, 259–266
 - SPECweb99 benchmark, 262–266

- DeMorgan's laws, B6
 - DeMorgan's theorems, B11
 - Dependability, disk, 569–580
 - Dependence detection, 406
 - Dependent instructions, 403
 - Desktop computers, 5
 - performance benchmarks, 255
 - Destination register, 64
 - Deutsch, Peter, CD7.9:8
 - D flip-flop, B51–53
 - Dhrystone synthetic benchmark,
 - CD4.7:2, IMD4:11–12
 - Dies, 30
 - Digital cameras, 236–237, 603–606
 - Digital signal-processing extensions, D19
 - DIMMs. *See* Dual inline memory modules
 - Directives, data layout, A14–15
 - Direct-mapped cache, 474–475, 497
 - Direct memory access (DMA), 594–596
 - Directories, CD9.4:24
 - Dirty bit, 521
 - Disabled people, technology for, 366–367
 - Disk(s)
 - arrays, IMD8:2
 - controller, 570, 571
 - drives, 19, 20
 - fallacies and pitfalls, 606–609
 - read time, 570–571
 - storage and dependability, 569–580,
 - CD8.11:1–4
 - Dispatch, 350
 - Displacement addressing, 100
 - Distributed block-interleaved parity
 - (RAID 5), 578
 - Distributed memory, CD9.4:22
 - Distributed shared memory (DSM),
 - CD9.4:24
 - divide, 188–189
 - Dividend, 183
 - divide unsigned, 188–189
 - Division, 183–189
 - Divisor, 183
 - Don't-care terms, 303, B16–18
 - Double, 192
 - Double extended precision, 218
 - Double precision, 192
 - Double data rate synchronous DRAMs
 - (DDR SDRAMs), 490–491
 - DRAM. *See* Dynamic random access memory
 - Dual inline memory modules
 - (DIMMs), 22
 - DVD drive, 19, 20
 - DVDs (digital video disks), 25
 - Dynamically linked libraries (DLLs),
 - 112–114
 - Dynamic branch prediction, 382,
 - 421–423
 - Dynamic data, A22
 - Dynamic multiple issue, 433, 442–445
 - Dynamic pipeline scheduling, 443–445
 - Dynamic random access memory
 - (DRAM), 20, 469, 487–488,
 - 490–491, 513, B60, 63–65
 - historical development of, CD7.9:3–4
- E**
- Early restart, 481–482
 - Eckert, J. Presper, CD1.7:1, 2, 4, CD7.9:1
 - Eckert-Mauchly Computer Corp., CD1.7:4
 - Edge-triggered clocking methodology,
 - 290–291, B47
 - EDSAC (Electronic Delay Storage Automatic Calculator), CD1.7:2,
 - CD5.12:1
 - EDVAC (Electronic Discrete Variable Automatic Computer), CD1.7:1–2
 - EEMBC benchmarks, 255, IMD4:17–18
 - 802.11 standard, CD8.3:9–10
 - Eispack, CD3.10:3
 - Elaborations, 8
 - Elapsed time, 244
 - Ellison, Larry, CD8.11:5
 - Embedded computers, 6–8, CD1.7:8–9, A7
 - performance benchmarks, 255,
 - IMD4:17–18
 - EMC, CD8.11:6
 - Emulation, CD5.12:1–2
 - Encoder, B9
 - Energy efficiency problems, 263–265
 - Engelbart, Doug, 16
 - ENIAC (Electronic Numerical Integrator and Calculator), CD1.7:1–2, 3,
 - CD7.9:1
 - Environmental problems, technology
 - and, 156–157
 - EPCWrite, 342
 - Error-correcting codes, B65
 - Error-detecting codes, B65–67
 - Ethernet, 26, CD8.3:5, CD8.11:7–8,
 - IMD8:1–2
 - Evolution versus revolution,
 - CD9.10:45–47
 - Exception enable, 532
 - Exception program counter (EPC), 173,
 - 341–342, 429–431
 - Exceptions, 173, A33–38
 - address, 342–343
 - control checking of, 343–346
 - defined, 340–341
 - handling of, 341–343, A35, 36–38
 - imprecise, 432
 - pipeline, 427–432
 - Executable file, 109
 - Execution time, 242, 244–245
 - use of total, 257–259
 - Executive process, 529
 - Exponent, 191
 - Extended accumulator, CD2.19:2
 - External labels, A11
- F**
- Failures
 - mean time between failures
 - (MTBF), 573
 - mean time to failure (MTTF), 573, 574
 - mean time to repair (MTTR), 573, 574
 - reasons for, 574
 - synchronizer, B76
 - Fallacies, 33
 - False sharing, CD9.3:14
 - Fanout, 32
 - Fast carry, B38–47

Fetch-on-miss/write, 484
 Field programmable devices (FPDs),
 B77–78
 Field programmable gate arrays
 (FPGAs), B77
 Fields
 defined, 61
 MIPS, 63–64
 File system benchmarks, 598–599
 Fine-grained multithreading,
 CD9.7:31–33
 Finite state machines, 330, 331–340,
 B47–72, C8–20
 Firewire, 582, 583
 Firmware, CD5.12:2
 Fisher, Josh, CD6.13:4
 Fishman, Harvey, 366–367
 Flags, 140
 FLASH, 23, 25
 Flat-panel display, 18
 Flip-flops, 290, B50–53
 Floating point, 189, 191–220
 addition, 197–201
 converting binary to decimal floating
 point, 196
 defined, 191
 historical development of,
 CD3.10:1–9
 IA-32, 217–220
 MIPS, 206–213
 multiplication, 202–205
 representation, 191–197
 rounding, 214–215
 Floating Point Systems, CD6.13:4, 5
 Floating vectors, CD3.10:2
 Floppy disks, 25, CD1.7:6
 Floppy drives, 25
 Flush instructions, 418
 Flynn, Michael, CD6.13:3
 Formal parameter, A16
 Forrester, J., CD7.9:1
 FORTRAN, CD2.19:6, 7–8
 overflows, 172, 173
 Forwarding, 376–377, 402–412, CD6.7:3
 Forward reference, A11

Fraction, 191, 193
 Frame buffer, 18
 Frame pointer, 86
 Front end of compiles, CD2.12:1–9
 Fully associative cache, 497
 Fully connected network, CD9.6:28
 Function code, 63

G

Gates, B7–8, C4–8
 Gateways, CD8.3:6
 General-purpose register (GPR), 135,
 138, CD2.19:2–3
 Generate, carry lookahead, B39–47
 Geometric mean, IMD4:9–11
 Gibson, Garth, CD8.11:6
 Global common subexpression elimina-
 tion, 118
 Global labels, A11
 Global miss rate, 509
 Global optimization, 117–121,
 CD2.12:4–6
 Global pointer, 85
 Goldstine, Herman H., 48, CD1.7:1–2, 3,
 CD3.10:1
 Google, CD9.8:34–39
 News, 465
 Gosling, James, CD2.19:7
 Graph coloring, CD2.12:7–8
 Graphics display, 18
 Gray, Jim, CD8.11:5
 Gray-scale display, 18
 Guard, 214–215

H

Half words, 94
 Hamming code, B67
 Handler, 533
 Handshaking protocol, 583–584
 Hard disk, magnetic, 23
 Hard drive, 19, 20
 Hardware
 description language, B20–25

 functions of, 15
 performance affected by, 10
 synthesis tools, B21
 Hardwired control, 348, CD5.12:2
 Harvard architecture, CD1.7:3
 Hazards
 See also Pipelining hazards
 detection unit, 413–415
 Heap, allocating space for data on, 87–88
 Heat sink, 22
 Held, Gerald, CD8.11:5
 Hewlett-Packard, CD2.19:5,
 CD3.10:6–7, CD4.7:2
 PA-RISC 2.0, D34–36
 Hexadecimal-binary conversion table, 62
 Hi, 181
 High-level optimization, 116–117
 High-level programming languages
 advantages of, 14–15
 architectures, CD2.19:4
 defined, 13
 translating into instructions that
 hardware can execute, 12–15
 Hit(s)
 Average Memory Access Time
 (AMAT), IMD7:1
 defined, 470
 rate/ratio, 470–471
 time, 471
 Hitachi, SuperH, D39–40
 Hold time, B53
 Hot swapping, 579
 Hubs, CD8.3:7

I

IBM
 disk storage, CD8.11:1–4
 early computers, CD1.7:5
 floating points, CD3.10:2, 3–4
 floppy disks, CD1.7:6, CD8.11:2
 history of programming languages,
 CD2.19:6
 microprogramming, CD5.12:1–2
 multiple issue, CD6.13:4

- PowrPC, D32–33, IMD2:17–20, IMD3:10
- RAID, CD8.11:6
- Stretch computer, CD6.13:1–2
- virtual memory, CD7.9:5–7, 10
- Winchester disks, CD8.11:2, 4
- IEEE 754 floating-point standard, 193–196, CD3.10:7–9
- If-then-else statements, compiling into conditional branches, 72–73
- Immediate addressing, 100
- Implementation, 22, 24
- Imprecise interrupts/exceptions, 432, CD6.13:3
- IMS, CD8.11:4
- Induction variable elimination, 119–120
- Infinity, 193
- Ingres, CD8.11:5
- In-order commit, 445
- In-order completion, 445
- Input devices, 15, 566, A38–40
- Input don't cares, B16
- Input operation, 582
- Inputs, asynchronous, B75–77
- Instruction decode, 385, 390, 392, 402
- Instruction encoding, MIPS floating-point, 208
- Instruction group, 440
- Instruction fetch, 385, 388–389, 392, 400
- Instruction format, 61
- Instruction latency, 452
- Instruction-level parallelism (ILP), CD9.7:33, 433, CD6.13:5
- Instruction mix, 253
- Instruction register (IR), 319, 321
- Instruction sets
 - addressing, 95–105
 - architecture, 22, 24
 - compiler optimization, 116–121
 - decision-making instructions, 72–74
 - defined, 48
 - designing, for pipelining, 374–375
 - historical development of, CD2.19:1–9
 - logical operations, 68–71
 - operands of hardware, 52–60
 - operations of hardware, 49–52
 - to process text, 90–95
 - representing instructions to computer, 60–68
 - styles, IMD2:7–9
 - supporting procedures, 79–90
 - translating and starting a program, 106–115
- Integers, signed versus unsigned, 165
- Integrated circuits (ICs)
 - costs, IMD1:1–2
 - defined, 20, 27–28
 - how they are manufactured, 28–33
- Integrated Data Store (IDS), CD8.11:4
- Intel, CD1.7:5, 6, CD8.11:8
 - See also* Pentium 4
 - 8086, 135, CD2.19:2, 4, 5
 - 8087, 135, CD3.10:7
 - 80286, 135, CD2.19:5
 - 80386, CD2.19:5
 - 80486, 135, CD2.19:5
 - iSC 860 and Paragon, CD9.11:52
 - Pentium and Pentium Pro, 135, CD2.19:5, 448–450
 - SPEC CPU benchmarks, 254–255, 259–266
 - SPECweb99 benchmark, 262–266
- Intel IA-32, 59
 - addressing modes, 138
 - complexity of, 347–348
 - conclusions, 142–143
 - fallacies and pitfalls, 143–144
 - floating point, 217–220
 - historical development of, 134–137, CD2.19:4–5
 - instruction encoding, 140–142
 - integer operations, 138–140
 - registers, 137–138
- Intel IA-64, 435, CD5.12:3
 - architecture, 440–442, CD6.13:4–5
- Intel Streaming SIMD Extensions (SSE), 135–136
- Intel Streaming SIMD Extension 2 (SSE2), 136
- floating points, 220
- Interface message processor (IMP), CD8.3:5
- Interference graph, CD2.12:7
- Interleaving, 489
- Intermediate representation, CD2.12:2–3
- Internet, CD8.11:7
 - news services, 464–465
- Internetworking, CD8.3:1–4
- Interrupt-driven I/O, 590–591
- Interrupts, 173, A33–38
 - handler, A33
 - imprecise, 432, CD6.13:3
 - priority levels, 591–593
 - use of term, 340–341
- Intrinsity FastMATH processor example, 485–487, 524
- Invalid operations, 193
- I/O
 - buses, 582
 - communicating with processor, 590–591
 - designing a system, 600–603
 - devices, 15, 566, A38–40
 - digital camera example, 603–606
 - diversity of, 568
 - fallacies and pitfalls, 606–609
 - giving commands to devices, 589–590
 - historical development of, CD8.11:1–9
 - instructions, 590
 - interfacing devices to processor, memory, and operating system, 588–596
 - interrupt-driven, 590–591
 - interrupt priority levels, 591–593
 - measuring performance, 567
 - memory-mapped, 589–590
 - performance, 597–600
 - rate, 598
 - requests, 568
 - transferring data between devices and memory, 593–595
- Issue packet, 435
- Issue slots, 434

J

Java
 bytecode, 114, CD2.14:1, 2
 characters and strings, 93–95
 compiling, CD2.14:4–6
 development of, CD2.19:7
 interpreting, CD2.14:1–3
 invoking methods, CD2.14:6
 logical operations, 68–71
 translating hierarchy, 114–115
 sort and swap, CD2.14:6–13
 while loop, CD2.14:3–4, 5–6
 Java Virtual Machine (JVM), 115, CD2.14:3
 Jhai Foundation, PC network, 44–45
 Jobs, Steven, CD1.7:5
 Johnson, Reynold B., CD8.11:1
 Joy, Bill, CD7.9:9
 J-type, 97
 jump, 73, 77, 80, 89, 296
 addressing in, 97–99
 datapath and control and, 313–314, 321, 328, 336
 Jump address table, 76, 77, IMD2:15–16
 jump-and-link, 79–80, 89
 jump register, 76
 Just-in-Time (JIT) compiler, 115

K

Kahan, William, CD3.10:5–7, 8, 9
 Kahn, Robert, CD8.11:7
 Karnaugh maps, B18
 Katz, Randy, CD8.11:6
 Kay, Alan, CD2.19:7
 Kernel benchmarks, CD4.7:2, IMD4:7–8
 Kernel process, 529
 Knuth, Donald, CD2.19:8

L

Labels, external/global and local, A11
 Lampson, Butler, CD7.9:8, 11

Laptop computers, performance versus power versus energy efficiency, 263–265
 Latches, B59–53
 Latency
 instruction, 452
 pipeline, 383
 Leaf procedures, 83, 93
 Least recently used (LRU), 504, 518, 519
 Least significant bit, 161
 Level-sensitive clocking, B74–75
 Link editor, 109
 Linkers, 108–111, A4, 18–19
 Linpack, CD3.10:3, CD4.7:2
 Linux, 11, CD7.9:11
 Liquid crystal displays (LCDs), 18
 Lisp, CD2.19:6
 Little Endian, 56, A43
 Live range, CD2.12:7
 Livermore Loops, CD4.7:2
 Lo, 181
 Load, 54, 57
 advanced, 442
 byte, 91, 164
 byte unsigned, 164
 half, 94, 164
 halfword unsigned, 164
 linked, CD9.3:19
 locked, CD9.3:19
 upper immediate, 95
 word, 54, 57, 59, 294, 300–318
 Loader, 112
 Loading, A19–20
 Loading 32-bit constant, 96
 Load-use data hazard, 377
 Local area networks (LANs), 26, CD8.3:5–8, CD8.11:7–8
 Locality, principle of, 468–469
 Local labels, A11
 Local miss rate, 509
 Local optimization, 117–121, CD2.12:3–4
 Lock, CD9.1:5
 Lock variables, CD9.3:18

Logic

arrays of logic elements, B18–19
 combinational, B5, 8–20, 23–25
 equations, B6–7, C12–13
 sequential, B5, 55–57
 two-level, B10–14
 Logical operations, 68–71, B6, IMD2:21–22
 Logic design conventions, 289–292
 Long-haul networks, CD8.3:5
 Long instruction word (LIW), CD6.13:4
 Lookup tables (LUTs), B78
 Loops, 74–75
 branch, 421–422
 unrolling, 117, 438–440
 Lorie, Raymond, CD8.11:5

M

Machine code, 61
 Machine language, 61, A3
 decoding, 100–104
 MIPS floating-point, 207
 object file and, 108
 MacOS, 11
 Macros, A4, 15–17
 Magnetic disks, 23, 569
 differences between main memory and, 24
 memory hierarchies and, 469, 513
 Magnetic resonance imaging (MRI), 622–623
 Magnetic tape, 25
 Main memory, 23
 differences between magnetic disks and, 24
 Make the common case fast, 267, 285
 Mark machines, CD1.7:3
 Mask, 70
 Mauchly, John, CD1.7:1, 2, 4
 McCarthy, John, CD2.19:6, CD7.9:7, 11
 McKeeman, William, CD2.19:8
 Mealy, George, 338
 Mealy machine, 338, 340, B68

- Mean time between failures (MTBF), 573
- Mean time to failure (MTTF), 573, 574, 606
- Mean time to repair (MTTR), 573, 574
- Megabyte, 23
- Memories, 290
- Memory, 8
 - access, 385, 390, 392, 402
 - allocation, 87–88
 - Average Memory Access Time (AMAT), IMD7:1
 - board, 20
 - cache, 20
 - cards, 25
 - consistency model, CD9.3:15
 - defined, 20, 23
 - direct memory access (DMA), 594–596
 - distributed, CD9.4:22, 24
 - dynamic random access (DRAM), 20, 469, 487–488, 490–491, 513, B60, 63–65
 - historical development of, CD7.9:1–12
 - main, 23
 - mapping, 512
 - nonvolatile, 23
 - operands, 54–55
 - primary, 23
 - random access (RAM), 20
 - read only (ROM), B14, 16, C13–19
 - secondary, 23
 - shared, CD9.1:4–5, CD9.4:22, 24
 - static random access (SRAM), 20, 469, B57–60
 - transferring data between devices and, 593–595
 - unit, 292
 - usage, A20–22
 - virtual, 511–538
 - volatile, 23
- Memory data register (MDR), 319, 328
- Memory elements
 - latches, flip-flops, and register files, B49–57
 - SRAMs and DRAMs, B57–67
- Memory hierarchy
 - caches, 473–511
 - defined, 469
 - fallacies and pitfalls, 550–552
 - framework for, 538–545
 - historical development of, CD7.9:5–7
 - levels, 470–471
 - methods for building, 469–470
 - overall operation of, 527–528
 - Pentium P4 and AMD Opteron, 546–550
 - trends for, 553–555
 - virtual, 511–538
- Memory-mapped I/O, 589–590
- Memory-memory instructions, IMD2:8
- Memory reference, 327, 328, 334–335
- MESI cache coherency protocol, CD9.3:16, 18
- Message passing, CD9.1:6, CD9.4:22–23
- Metastability, B75–76
- MFLOPS (million floating-point operations per second), IMD4:15–17
- Microarchitecture, 448
- Microcode, 348
- Microinstructions, 348–349, CD5.7:1
 - fields, CD5.7:3, 5–9
 - format, CD5.7:2–4
- Microoperations, 348
- Microprocessors
 - first, CD1.7:5
 - future of, CD9.10:44–45
- Microprogramming
 - controller, 348, CD5.12:2
 - creating a program, CD5.7:4–10
 - defined, 330, 346
 - fallacies and pitfalls, 350–352
 - historical development of, CD5.12:1–4
 - implementing the program, CD5.7:10–12
 - microinstruction format defined, CD5.7:2–4
 - simplifying design with, CD5.7:1–13
- Microsoft Corp., CD1.7:5, CD7.9:10, CD8.11:5, 6
- Minicomputers, first, CD1.7:5
- Minterms, B12
- MIPS, 49
 - addressing, 95–105
 - allocation of memory, 87
 - arithmetic logic unit (ALU), B32–38
 - compiling statements into, 50–51
 - decision-making instructions, 72–73
 - exception code, 535
 - fields, 63–64
 - floating point, 206–213
 - implementation, 285–289
 - instruction encoding table, 64, 103
 - instruction set, 49
 - logical operations, 68–71
 - machine language, summary of, 67, 78, 90
 - mapping registers into numbers, 60–68
 - operands, summary of, 59, 67, 71, 89, 105, 169
 - registers, 52–53, 79–80, 85, 88, 532
 - RISC core subset, D9–16, 20–24
 - RISC instructions for MIPS16, D41–43
 - RISC instructions for MIPS64, D25–27
 - translating assembly into machine language, 65–66
- MIPS assembly language
 - add, 49–51
 - add immediate, 58, 59
 - add immediate unsigned, 172
 - add unsigned, 172
 - AND, 69, 70
 - and immediate, 71, 89
 - conditional and unconditional branches, 72–73
 - divide, 188–189
 - divide unsigned, 188–189
 - floating point, 207
 - jump, 73, 80
 - jump address table, 76
 - jump-and-link, 79–80
 - load word, 54–59, 294
 - move from hi, 181
 - move from lo, 181
 - multiply, 181

- multiply unsigned, 181
 - nor (NOR), 69, 70
 - or (OR), 69, 70
 - or immediate, 71, 89
 - set on less than, 75
 - set on less than immediate, 77, 165
 - set on less than immediate
 - unsigned, 165
 - set on less than unsigned, 165
 - shifts, 69
 - store word, 54–59, 294
 - subtract, 49–51
 - subtract unsigned, 172
 - summary of, 51, 59, 67, 71, 77, 89, 105, 169, 175, 190, 207, 226–228
 - xor, IMD2:21–22
 - MIPS assembly language, R2000
 - addressing modes, A45–47
 - arithmetic and logical instructions, A51–57
 - assembler syntax, A47–49
 - branch instructions, A59–63
 - comparison instructions, A57–59
 - constant manipulating instructions, A57
 - data movement instructions, A70–73
 - encoding instructions, A49
 - exception and interrupt instructions, A80–81
 - floating-point instructions, A73–80
 - instruction format, A49–51
 - jump instructions, A63–64
 - load instructions, A66–68
 - store instructions, A68–70
 - trap instructions, A64–66
 - MIPS (million instructions per second)
 - equation, 268
 - peak versus relative, IMD4:13–14
 - problem with using as a performance measure, 268–270
 - Mirroring, 575
 - Miss, 470
 - Misses
 - Average Memory Access Time (AMAT), IMD7:1
 - cache, 482–483, 496–502
 - capacity, 543
 - cold-start, 543
 - collision, 543
 - compulsory, 543
 - conflict, 543
 - TBL, 531
 - Miss penalty, 471
 - reducing, using multilevel caches, 505–509
 - Miss rate/ratio, 471
 - global, 509
 - local, 509
 - Mitsubishi, M32R, D40–41
 - Moore, Edward, 338
 - Moore, Gordon, 28
 - Moore machine, 338, B68
 - Moore's law, 28, 181
 - Mosaic, CD8.11:7
 - Most significant bit, 161
 - Motherboard, 19, 20
 - Motorola
 - PowrPC, D32–33, IMD2:17–20, IMD3:10
 - 68881, CD3.10:8
 - Mouse, 16–17
 - move from hi, 181
 - move from lo, 181
 - Move from system control, 173
 - M32R, D40–41
 - Multicomputers, CD9.11:52
 - MULTICS (Multiplexed Information and Computing Service), CD7.9:8
 - Multicycle implementation, 318–340
 - Multiflow Co., CD6.13:4
 - Multilevel caching, 492, 505–510
 - Multimedia extensions of desktop/server RISCs, D16–19
 - Multiple instruction multiple data (MIMD), CD9.11:51–53
 - Multiple instruction single data (MISD), CD9.11:51
 - Multiple issue
 - defined, 433
 - dynamic, 433, 442–445
 - IBM's work on, CD6.13:4
 - static, 433, 435–442
 - Multiplexors, 286, B9–10
 - Multiplicand, 176
 - Multiplication, 176–182
 - floating point, 202–205
 - Multiplier, 176
 - multiply, 181
 - multiply unsigned, 181
 - Multiprocessors
 - connected by a network, CD9.4:20–25
 - connected by a single bus, CD9.3:11–20
 - defined, CD9.1:4, CD9.11:52
 - future of, CD9.10:43–44
 - history development of, CD9.11:47–55
 - inside a chip and multithreading, CD9.7:30–34
 - networks, CD9.4:20–25, CD9.6:27–30
 - programming, CD9.2:8–10
 - types of, CD9.1:4–8
 - Multistage network, CD9.6:29–30
 - Multithreading, CD9.7:30–34
- ## N
- Name dependence, 439
 - NaN (Not a Number), 193
 - NAND gate, B8
 - NCR, CD8.11:6
 - Negation shortcut, 166
 - Nested procedures, 83–85
 - Netscape, CD8.11:7
 - Network bandwidth
 - defined, CD9.6:27
 - fully connected, CD9.6:28
 - total, CD9.6:27–28
 - Networks, 25–27
 - characteristics of, CD8.3:1
 - crossbar, CD9.6:30
 - internetworking, CD8.3:1–4
 - local area, CD8.3:5–8

- long-haul, CD8.3:5
- multiprocessors connected by,
 - CD9.4:20–25, CD9.6:27–30
- multistaged, CD9.6:29–30
- Pentium 4, 585–587
- wireless local area, CD8.3:8–10
- Next-state function, 331, B67, C12–13, 21–27
- No-allocate-on-write, 484
- No-fetch-on-write, 484
- Nonblocking assignment, B24
- Nonblocking caches, 445, 548
- Nonuniform memory access (NUMA)
 - multiprocessors, CD9.1:6, CD9.4:22
- Nonvolatile memory, 23
- Nonvolatile storage device, 569
- Nop, 413–414
- nor (NOR), 70, 301, B8
- Normalized number, 189
- Northrop, CD1.7:4
- NOT, 70, B6
- Numbers
 - ASCII versus binary, 162
 - base to represent, 160–161
 - converting binary to decimal, 164
 - loads, 164
 - negative and positive, 165
 - shortcuts, 166–168
 - sign and magnitude, 162
 - sign bit, 163
 - signed and unsigned, 160–170
 - two's complement representation, 163
- Nygaard, Kristen, CD2.19:7

O

- Oak, CD2.19:7
- Object files
 - defined, 108, A10
 - format, A13–14
 - linking, 109–111
- Object-oriented language
 - defined, 130, CD2.14:1
 - Javas, CD2.14:1–13
- Offset, 55, 56

- Opcode, 63, 303, 305, 306
- Open Source Foundation, CD7.9:9
- Open Systems Interconnect (OSI)
 - CD8.3:2
- Operands
 - for computer hardware, 52–60
 - constant or immediate, 57
 - memory, 54–55
 - MIPS, summary of, 59, 67, 71, 89, 105, 169
 - MIPS floating point, 207
- Operating systems
 - examples of, 11
 - functions of, 11–12, 588–589
 - historical development of,
 - CD7.9:7–11
- Operations, for computer hardware, 49–52
- Operators, Verilog, B21–22
- Optical disks, 25
- Optimizations
 - high-level, 116–117
 - local and global, 117–120,
 - CD2.12:3–6
 - summary of, 120–121
- or (OR), 70, 301, 321, B6
- Oracle, CD8.11:5
- or immediate, 71
- Out-of-order execution, 445
- Output devices, 15, A38–40
- Output don't cares, B16
- Output operation, 582
- Overflow, CD3.10:5
 - adding and subtracting and, 171–174
 - division and, 189
 - exceptions, detection of, 343
 - floating point and, 192
 - multiplying and, 181
- Overlays, 511–512

P

- Packets, CD8.3:5
- Page, 512
 - placing and find, 515–516
- Page faults, 512, 514, 516–521, 531

- Page offset, 513, 514
- Page table, 515–516
- Palmer, John F., CD3.10:7
- Parallel processing program, CD9.1:4,
 - CD9.2:8–10, CD9.4:22–23
 - addressing, CD9.4:23–25
 - fallacies and pitfalls, CD9.9:39–42
- PA-RISC 2.0, D34–36
- Parity
 - big-interleaved (RAID 3), 576–577
 - block-interleaved (RAID 4), 577–578
 - distributed block-interleaved
 - (RAID 5), 578
- Parsing, CD2.12:1
- Pascal, CD2.19:6–7
- Patterson, David, CD8.11:6
- PC-relative addressing, 98, 100
- PCSpim, A42, CDA:1–3
- PCSrc control and signal, 305
- PCWrite, 321
- PCWriteCond, 321
- Peer-to-peer architecture, CD8.3:9–10
- Pentium 4
 - buses and networks of, 585–587
 - implementation of, 348–350
 - manufacturing of, 28–33
 - memory hierarchies, 546–550
 - pipeline, 448–450
- Pentium processors
 - SPEC CPU benchmarks, 254–255, 259–266
 - SPECweb99 benchmark, 262–266
- Performance
 - See also* Pipelining
 - benchmarks, 254–255, IMD4:7–8, 11–18
 - of caches, 253
 - of caches, measuring and improving, 492–511
 - comparing, 252–253, 256–259, 425–426
 - CPU, 245, 246–253
 - defined, 241–244
 - equation, 248–249
 - evaluating, 254–259
 - factors affecting, 251

- fallacies and pitfalls, 266–270
 - historical review, CD4.7:1–4
 - how hardware and software affect, 10
 - I/O, 597–600
 - measuring, 244–246
 - per unit cost of technologies, 27
 - of the pipeline, 253
 - relative, 243–244
 - reports, 255–256
 - single-cycle machines and, 315–318
 - SPEC CPU benchmarks, 254–255, 259–266, CD4.7:2–3, IMD4:7–8
 - SPECweb99 benchmark, 262–266
 - system, 245
 - versus power and energy efficiency, 263–265
 - Personal computers, early, CD1.7:5–8
 - Peterman, Mark, 366–367
 - Physical addresses, 511, 512, 513–514
 - Physically addressed cache, 528
 - Physical page number, 513
 - Pipeline stalls, 377–379, 413–416, CD6.7:5–7
 - Pipelining
 - advanced methods for extracting more performance, 432–445
 - control, 399–402
 - datapath, 384–399
 - defined, 370
 - designing instruction sets for, 374–375
 - exceptions, 427–432
 - fallacies and pitfalls, 451–384
 - forwarding, 376–377, 402–412, CD6.7:3
 - graphic representation, 395–399
 - historical development of, CD6.13:1–13
 - instruction execution sped up by, 372–374
 - latency, 383
 - overview of, 370
 - Pentium 4 example, 448
 - stalls, 377–379, 413–416, CD6.7:5–7
 - Verilog used to describe and model, CD6.7:1–9
 - Pipelining hazards, branch/control, 379–382, 416–424, CD6.7:8–9
 - Pipelining hazards data
 - defined, 376–379
 - forwarding, 402–412
 - load-use, 377
 - stalls, 413–416
 - structural, 375
 - Pitfalls, 33–34
 - Pixels, 18
 - Pointers, arrays versus, 130–134
 - Poison, 442
 - Polling, 590
 - Pop, 80
 - Pop-up satellite archival tags (PSATs), 156–157
 - Positive numbers, multiplying, 176–180
 - Power, 30–32
 - consumption, problems with, 263–265
 - PowerPC
 - addressing, IMD2:17–20
 - instructions, D32–33
 - multiply-add instruction, IMD3:10
 - Prediction, 382, 421–423
 - IA-64, 441
 - Primary memory, 23
 - Procedure call conventions, A22–33
 - Procedure call frame, A23
 - Procedures
 - allocating space for data on heap, 87–88
 - allocating space for data on stack, 86
 - C, 81–88
 - defined, 79
 - frame, 86
 - inlining, 116
 - leaf, 83, 93
 - nested, 83–85
 - preserved versus not preserved, 85
 - recursive, A26, 29
 - steps, 70
 - Processor, 20
 - communicating with, 590–591
 - cores, 6–7
 - memory buses, 582
 - Process switch, 530
 - Product of sums, B10–12
 - Product terms, B12
 - Program counter (PC), 80, 292
 - Programmable logic arrays (PLAs), B12–14, C7, 19–20
 - Programmable logic devices (PLDs), B77
 - Programmable read only memory (PROM), B14, 16
 - Programming languages, history of, CD2.19:6–7
 - Propagate, carry lookahead, B39–47
 - Propagation time, B77
 - Protection group, 576
 - Protocol families/suites, CD8.3:1–2
 - Protocol stack, CD8.3:3
 - Pseudodirect addressing, 100
 - Pseudoinstructions, 107, A17
 - Push, 80
 - Putzolu, Gianfranco, CD8.11:5
- ## Q
- Quicksort, 129, 507–508
 - Quotient, 183
- ## R
- Radio communication, CD8.3:8–9
 - Radix Sort, 507–508
 - RAID (redundant arrays of inexpensive disks)
 - big-interleaved parity (RAID 3), 576–577
 - block-interleaved parity (RAID 4), 577–578
 - distributed block-interleaved parity (RAID 5), 578
 - error detecting and correcting code (RAID 2), 575
 - historical development of, CD8.11:5–6
 - mirroring (RAID 1), 575
 - no redundancy (RAID 0), 575
 - P + Q redundancy (RAID 6), 578
 - summary of, 578–579
 - use of term, 574–574

- Random access memory (RAM), 20
 - Raster cathode ray tubes (CRTs), 18
 - Raster refresh buffer, 18
 - Rau, Bob, CD6.13:4
 - Read only memory (ROM), B14, 16, C13–19
 - Read/write head, 23
 - Reals, 189
 - Receive message routine, CD9.1:6
 - Recursive procedures, A26, 29
 - Reduced instruction set computer (RISC), CD2.19:4
 - addressing modes and instruction formats, D5–9
 - Alpha, D27–28
 - architecture, CD5.12:3
 - ARM, D36–38
 - desktop versus embedded, D3–5
 - digital signal-processing extensions, D19
 - MIPS, D9–16, 20–24
 - MIPS16, D41–43
 - MIPS64, D25–27
 - M32R, D40–41
 - multimedia extensions, D16–19
 - PA-RISC 2.0, D34–36
 - PowrPC, D32–33
 - SPARCv.9, D29–32
 - SuperH, D39–40
 - Thumb, D38–39
 - Redundancy. *See* RAID (redundant arrays of inexpensive disks)
 - Reference bit, 519
 - Refresh rate, 18
 - reg, B21–22
 - Register addressing, 100
 - Register file, 293–294, B49, 53–55
 - read, 385, 390, 392, 402
 - Registers, 52–53, 59, 88, 290, 532
 - allocation, CD2.12:7–9
 - architectural, 448
 - dedicated, CD2.19:2
 - destination, 64
 - general-purpose, 135, 138, CD2.19:2–3
 - global pointer, 85
 - IA-32, 137–138
 - jump, 76, 80
 - mapping into numbers, 60–68
 - number, 294
 - renaming, 439
 - special-purpose, CD2.19:2
 - spilling, 58, 80
 - Register use conventions, A22–33
 - Relational databases, CD8.11:4–5
 - Reliability, 573
 - Relocation, 513
 - Relocation information, A13
 - Remainder, 183
 - Remington-Rand, CD1.7:4
 - Remote access times, CD9.1:7
 - Reorder buffer, 443
 - Reproducibility, 255–256
 - Requested word first, 482
 - Reservation stations, 443
 - Response time, 242, 244
 - Restartable instruction, 533
 - Restorations, 572
 - Return address, 80
 - Rings, CD7.9:7
 - Ring topology, CD9.6:27
 - Ripple carry, B39, 44–45
 - RISC. *See* Reduced instruction set computer
 - Ritchie, Dennis, CD2.19:7, CD7.9:8, 11
 - Rotational delay, 570
 - Rotational latency, 570
 - Rounding, 214–215, CD3.10:2–4
 - Routers, CD8.3:6
 - R-type instructions, 292–293, 298
- S**
- Sandisk Corp., 605
 - Scanning, CD2.12:1
 - Scientific notation, 189, 191
 - Secondary memory, 23
 - Sectors, 569
 - Seek, 569
 - Seek time, 569–570
 - Segmentation, 514–515
 - Selector value, B9
 - Selinger, Patricia, CD8.11:5
 - Semantic analysis, CD2.12:1
 - Semaphores, CD9.3:18
 - Semiconductor, 29
 - Send message routine, CD9.1:6
 - Sensitivity list, B24
 - Separate compilation, A18
 - Sequential elements, 290
 - Sequential logic, B5, 55–57
 - Servers, 5
 - Set associative cache, 497, 504
 - set on less than, 75, 77, 165, 301
 - set on less than immediate, 77, 165
 - set on less than immediate unsigned, 165, 169
 - set on less than unsigned, 165, 169
 - Set-up time, B53
 - Shadowing, 575
 - Shared memory, CD9.4:22, 24
 - Shared-memory processors, CD9.1:4–5
 - Shared virtual memory, CD9.4:24
 - Shift amount, 69
 - Shifts, 69
 - Sign and magnitude, 162, 191
 - Sign bit, 163
 - Signed division, 187–188
 - Signed multiplication, 180
 - Signed numbers, 160–170
 - Sign extension, 164, 167–168, 294, 296
 - Significand, 193
 - Silicon, 29
 - Silicon crystal ingot, 29
 - Silicon Graphics. *See* MIPS
 - Simple programmable logic devices (SPLDs), B77
 - Simplicity, 285
 - Simputer, 45
 - Simula-67, CD2.19:7
 - Simultaneous multithreading (SMT), CD9.7:31–34
 - Single address space multiprocessors, CD9.1:4–6
 - Single bus, multiprocessors connected by a, CD9.3:11–20

- Single-cycle implementation scheme, 300–318
 - pipelined performance versus, 372–374
- Single instruction multiple data (SIMD), CD9.11:47–49, 51
- Single instruction single data (SISD), IMD 2.12, CD9.11:47
- Single precision, 192
- Small computer systems interface (SCSI), 573
- Smalltalk, CD2.19:7
- Smith, Jim, CD6.13:2
- Snooping cache coherency, CD9.3:13
- Software
 - applications, 11–12
 - performance affected by, 10
 - systems, 11
 - third-party of shrink-wrap, 5
- sort
 - body for for loop, 126–127
 - code for the body of, 124–126
 - full procedure, 127–128
 - Java, CD2.14:6–14
 - passing parameters, 127
 - preserving registers, 127
 - register allocation, 123
- Source language, A6
- SPARCV.9, D29–32
- Spatial locality, 468–469
- SPEC (System Performance Evaluation Corp.)
 - CPU benchmarks, 254–255, 259–266, CD4.7:2–3, IMD4:7–8
 - file server benchmarks, 599
 - Web server benchmarks, 599
- SPEC ratio, 259
- Speculation, 434–435
- SPECweb99 benchmark, 262–266
- Speedup, IMD4:5
- Spilling registers, 58, 80
- Spilt caches, 487
- SPIM, A40–45, CDA:1–2
 - command-line options, A42, CDA:1–3
- Spin waiting, CD9.3:19, 20
- Split transaction protocol, 585
- SRAM. *See* Static random access memory
- SRT division, 188
- Stack, 80
 - allocating space for data on, 86
 - instructions, CD2.19:3–4, IMD2:8–9
 - pointer, 80
 - segment, A22
- Stale data problem, 595
- Stallman, Richard, CD2.19:8
- Standby spares, 579
- Stanford DASH multiprocessor, CD9.11:52
- State elements, 289–290, B47–48
- Static data segment, 87, A20–22
- Static multiple issue, 433, 435–442, CD6.13:4
- Static random access memory (SRAM), 20, 469, B57–60
- Static storage class, 85
- Stewart, Robert G., CD3.10:7
- Sticky bit, 215
- Stone, Harold S., CD3.10:7
- Stonebraker, Mike, CD8.11:5
- Stop, 440
- Storage
 - for digital cameras, 603–606
 - disk, 569–580, CD8.11:1–4
- Storage classes, types of, 85
- store, 57
- Store buffer, 445, 485
- store byte, 91
- store conditional, CD9.3:19–20
- Stored-program concept, 49, 215
- store half, 94
- store word, 57–59, 294, 300–318
- Strength reduction, 118
- Stretch computer, CD6.13:1–2
- Strings
 - C, 92–93
 - Java, 93–95
- Striping, 575
- Stroustrup, Bjarne, CD2.19:7
- Structural hazards, 375
- Structural specification, B21
- Structured Query Language (SQL), CD8.11:4–5
- Subroutines, CD5.7:2
- subtract, 49–51, 301
- Subtraction, 170–176
- subtract unsigned, 172
- Sum of products, B10–12
- Sun Microsystems, CD4.7:2, CD7.9:9
 - SPARCV.9, D29–32
- Supercomputers
 - defined, 5
 - first CD1.7:5
- SuperH, D39–40
- Superscalar processors, 348, 442–445, CD6.13:4
- Supervisor process, 529
- swap
 - code for the body of, 122–123
 - full procedure, 123
 - Java, CD2.14:6–14
 - register allocation, 122
 - space, 517
- Switched networks, CD8.3:5
- Switches, CD8.3:7
- Switch statement, 76
- Sybase, CD8.11:5
- Symbol table, 108, A12, 13
- Symmetric multiprocessors (SMPs), CD9.1:6
- Synchronization
 - barrier, CD9.3:15
 - coherency and, CD9.3:18–20
 - defined, CD9.1:5
 - failure, B76
- Synchronizers, B75–77
- Synchronous bus, 582–583
- Synchronous system, B48
- Synthetic benchmarks, CD4.7:1–2, IMD4:11–12
- System call, 529, A43–45
- System CPU time, 245
- System performance, 245
- System R, CD8.11:4, 5
- Systems software, 11

T

Tags, cache, 475, 504
 Tail recursion, IMD2:10–11
 Target language, A6
 Taylor, George S., CD3.10:8–9
 Taylor, Robert, CD7.9:9–10
 TCP/IP, CD8.3:4, CD8.11:7
 Temporal locality, 468
 Terabytes, 5
 Text segment, 87, A13, 20
 Thacker, Chuck, CD7.9:8
 Thinking Machines, CD9.11:52
 Thompson, Ken, CD7.9:8, 11
 Thornton, J. E., CD6.13:2
 Thrashing, 537
 Thread-level parallelism (TLP),
 CD9.7:33
 Three Cs model, 543–545
 Throughput, 242
 Thumb, D38–39
 Time, definitions of, 244
 Time-sharing systems, CD7.9:7–11
 Timing methodologies, B72–77
 Tomasulo, Robert, CD6.13:2, 3
 Tomasulo's algorithm, CD6.13:2
 Torvald, Linus, CD7.9:10
 Tournament branch predictors, 423
 Trace cache, 349
 Tracks, 569
 Traiger, Irving, CD8.11:5
 Trains, computer controlled, 280–281
 Transaction processing (TP), 598
 Transaction Processing Council (TPC), 598
 Transfer time, 570
 Transistors, 27, 29
 Translating microprogram to hardware,
 C27–31
 Translation hierarchy for C, 106
 assembler, 107–108
 compiler, 107
 linker, 108–111
 loader, 112

Translation hierarchy for Java, 114
 compiler, 114–115
 Java Virtual Machine, 115
 Just in Time compiler, 115
 Translation-lookaside buffer (TLB),
 522–534, CD7.9:5
 Transportation, technology and, 280–281
 Truth tables, 302–303, B5, C5, 14, 15, 16
 Tucker, Stewart, CD5.12:1–2
 Turing, Alan, CD1.7:3
 TVM (transmission voie-machine),
 280–281
 Two-level logic, B10–14
 Two's complement representation, 163
 Types
 checking, CD2.12:1
 examples of, 85

U

Ullman, Jeff, CD2.19:8
 Unconditional branches, 73
 Undefined instruction, exception detec-
 tion of, 343
 Underflow, 192, CD3.10:5
 Unicode, 93–94
 Uniform memory access (UMA) multi-
 processors, CD9.1:6, CD9.4:22
 Units in the last place (ulp), 215
 UNIVAC I (Universal Automatic Com-
 puter), CD1.7:4
 UNIX
 development of, CD2.19:7,
 CD7.9:8–11
 loader, 112
 object file for, 108
 Unmapped, 536
 Unresolved references, A4
 Unsigned numbers, 160–170
 Untaken branch hazards, 381
 USB, 582, 583
 Use bit, 519
 User CPU time, 245

V

Valid bit, cache, 476
 VAX, CD5.12:2–3, CD7.9:9
 Vectored interrupts, 342
 Vector processing, CD9.11:49–51
 Verilog, CD5.8:1–7
 combinational logic in, B23–25
 data types and operators, B21–22
 description of, B20–25
 MIPS arithmetic logic unit (ALU),
 B36–38
 program structure, B23
 sequential logic, B55–57
 used to describe and model a pipe-
 line, CD6.7:1–9
 Very large scale integrated (VLSIs) cir-
 cuits, 20, 27–28, 29
 Very long instruction word (VLIW),
 CD6.13:4
 VHDL, B20, 21
 Virtual address, 512
 Virtually addressed cache, 527
 Virtual machine, simulation of, A41–42
 Virtual memory
 address translation, 512, 521–524
 defined, 511
 design, 514–521
 implementing protection with,
 528–530
 overlays, 511–512
 page, 512
 page, placing and find, 515–516
 page faults, 512, 514, 516–521
 page offset, 513, 514
 page table, 515–516
 reasons for, 511–512
 translation-lookaside buffer (TLB),
 522–534
 write-backs, 521
 Virtual page number, 513
 Volatile memory, 23
 von Neumann, John, 48, CD1.7:1–2, 3,

CD3.10:1–2, 3
Vyssotsky, Victor, CD2.19:8

W

Wafers, 29–30
Wall-clock time, 244
WARP, CD6.13:5
Web server benchmarks, 599
Weighted arithmetic mean, 258
Whetstone synthetic benchmarks,
 CD4.7:1–2, IMD4:11–12
while loop, 74–75, 98–99
 in Java, CD2.14:3–4, 5–6
Whirlwind project, CD1.7:4, CD7.9:1
Wide area networks (WANs), 26,
 CD8.11:11
WiFi, 44–45, CD8.3:8
Wilkes, Maurice, CD1.7:2, CD5.12:1,
 CD7.9:6
Wilkinson, James H., CD3.10:2

Windows, 11
wire, B21–22
Wired Equivalent Privacy, CD8.3:10
Wireless local area networks (WLANs),
 CD8.3:8–10
Wireless technology, 27, 156–157
Wirth, Niklaus, CD2.19:6
Wong, Gene, CD8.11:5
Word, in MIPS architecture, 52
Working set, 537
Workload, 254
World Wide Web, CD8.11:7
Wozniak, Stephen, CD1.7:5
Write-around, 484
Write-back, 385, 392, 402, 484–485, 521,
 542
Write buffer, 483–484
Write control signal, 290, 294
Write invalidate, CD9.3:14, 17
Writes
 handling cache, 483–485

 handling virtual memory, 521
Write-through, 483, 542

X

Xerox Palo Alto Research Center
 (PARC), 16, CD1.7:7–8,
 CD7.9:9–10, CD8.11:7, 8
xor, IMD2:21–22
xspim, A42, CDA:1–4

Y

Yield, 30

Z

Zip drive, 19, 20, 25
Zone bit recording (ZBR), 569
Zuse, Konrad, CD1.7:3

Further Reading

Chapter 1

Bell, C. G. [1996]. *Computer Pioneers and Pioneer Computers*, ACM and the Computer Museum, videotapes.

Two videotapes on the history of computing, produced by Gordon and Gwen Bell, including the following machines and their inventors: Harvard Mark-I, ENIAC, EDSAC, IAS machine, and many others.

Burks, A. W., H. H. Goldstine, and J. von Neumann [1946]. "Preliminary discussion of the logical design of an electronic computing instrument," Report to the U.S. Army Ordnance Department, p. 1; also appears in *Papers of John von Neumann*, W. Aspray and A. Burks, eds., MIT Press, Cambridge, MA., and Tomash Publishers, Los Angeles, 1987, 97–146.

A classic paper explaining computer hardware and software before the first stored-program computer was built. We quote extensively from it in Chapter 3. It simultaneously explained computers to the world and was a source of controversy because the first draft did not give credit to Eckert and Mauchly.

Campbell-Kelly, M., and W. Aspray [1996]. *Computer: A History of the Information Machine*, Basic Books, New York.

Two historians chronicle the dramatic story. The New York Times calls it well written and authoritative.

Ceruzzi, P. F. [1998] *A History of Modern Computing*. MIT Press, Cambridge, MA.

Contains a good description of the later history of computing: the integrated circuit and its impact, personal computers, UNIX, and the Internet.

Goldstine, H. H. [1972]. *The Computer: From Pascal to von Neumann*, Princeton University Press, Princeton, NJ.

A personal view of computing by one of the pioneers who worked with von Neumann.

Hennessy, J. L., and D. A. Patterson [2003]. Sections 1.3 and 1.4 of *Computer Architecture: A Quantitative Approach*, third edition, Morgan Kaufmann Publishers, San Francisco.

These sections contain much more detail on the cost of integrated circuits and explain the reasons for the difference between price and cost.

B.W. Lampson. *Personal distributed computing: The Alto and Ethernet software*. In ACM Conference on the History of Personal Workstations, January 1986.

C. R Thacker. *Personal distributed computing: The Alto and Ethernet hardware*. In ACM Conference on the History of Personal Workstations, January 1986.

These two papers describe the software and hardware of the landmark Alto.

Metropolis, N., Howlett, J, and G-C Rota, eds. [1980] *A History of Computing in the Twentieth Century*, Academic Press, New York.

A collection of essays that describe the people, software, computers, and laboratories involved in the first experimental and commercial computers. Most of the authors were personally involved in the projects. An excellent bibliography of early reports concludes this interesting book.

Public Broadcasting System [1992]. *The Machine that Changed the World*, videotapes.

These five one-hour programs include rare footage and interviews with pioneers of the computer industry.

Slater, R. [1987]. *Portraits in Silicon*, MIT Press, Cambridge, MA.

Short biographies of 31 computer pioneers.

Stern, N. [1980]. "Who invented the first electronic digital computer?" *Annals of the History of Computing* 2:4 (October) 375–76.

A historian's perspective on Atanasoff vs. Eckert and Mauchly.

Wilkes, M. V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, MA.

A personal view of computing by one of the pioneers.

Chapter 2

Bayko, J. [1996]. "Great Microprocessors of the Past and Present," available at www.mkp.com/books_catalog/cod/links.htm.

A personal view of the history of representative or unusual microprocessors, from the Intel 4004 to the Patriot Scientific ShBoom!

Kane, G., and J. Heinrich [1992]. *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ.

This book describes the MIPS architecture in greater detail than Appendix A.

Levy, H., and R. Eckhouse [1989]. *Computer Programming and Architecture: The VAX*, Digital Press, Boston.

This book concentrates on the VAX, but also includes descriptions of the Intel 80x86, IBM 360, and CDC 6600.

Morse, S., B. Ravenal, S. Mazor, and W. Pohlman [1980]. "Intel Microprocessors—8080 to 8086," *Computer* 13:10 (October).

The architecture history of the Intel from the 4004 to the 8086, according to the people who participated in the designs.

Wakerly, J. [1989]. *Microcomputer Architecture and Programming*, Wiley, New York.

The Motorola 680x0 is the main focus of the book, but it covers the Intel 8086, Motorola 6809, TI 9900, and Zilog Z8000.

Chapter 3

Burks, A. W., H. H. Goldstine, and J. von Neumann [1946]. “Preliminary discussion of the logical design of an electronic computing instrument,” *Report to the U.S. Army Ordnance Dept.*, p. 1; also in *Papers of John von Neumann*, W. Aspray and A. Burks, eds., MIT Press, Cambridge, MA, and Tomash Publishers, Los Angeles, 97–146, 1987.

This classic paper includes arguments against floating-point hardware.

Goldberg, D. [1991]. “What every computer scientist should know about floating-point arithmetic,” *ACM Computing Surveys* 23(1), 5–48.

Another good introduction to floating-point arithmetic by the same author, this time with emphasis on software.

Goldberg, D. [2002]. “Computer arithmetic,” Appendix A of *Computer Architecture: A Quantitative Approach*, third edition, J. L. Hennessy and D. A. Patterson, Morgan Kaufmann Publishers, San Francisco. (This appendix is online.)

A more advanced introduction to integer and floating-point arithmetic, with emphasis on hardware. It covers Sections 3.4–3.6 of this book in just 10 pages, leaving another 45 pages for advanced topics.

Kahan, W. [1972]. “A survey of error-analysis,” in *Info. Processing 71* (Proc. IFIP Congress 71 in Ljubljana), vol. 2, pp. 1214–39, North-Holland Publishing, Amsterdam.

This survey is a source of stories on the importance of accurate arithmetic.

Kahan, W. [1983]. “Mathematics written in sand,” *Proc. Amer. Stat. Assoc. Joint Summer Meetings of 1983, Statistical Computing Section*, pp. 12–26.

The title refers to silicon and is another source of stories illustrating the importance of accurate arithmetic.

Kahan, W. [1990]. “On the advantage of the 8087’s stack,” unpublished course notes, Computer Science Division, University of California at Berkeley.

What the 8087 floating-point architecture could have been.

Kahan, W. [1997]. Available via a link to Kahan’s homepage at www.mkp.com/books_catalog/cod/links.htm.

A collection of memos related to floating point, including “Beastly numbers” (another less famous Pentium bug), “Notes on the IEEE floating point arithmetic” (including comments on how some features are atrophying), and “The baleful effects of computing benchmarks” (on the unhealthy preoccupation on speed versus correctness, accuracy, ease of use, flexibility, . . .).

Koren, I. [2002]. *Computer Arithmetic Algorithms*, second edition, A. K. Peters, Natick, MA.

A textbook aimed at seniors and first-year graduate students that explains fundamental principles of basic arithmetic, as well as complex operations such as logarithmic and trigonometric functions.

Wilkes, M. V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, MA.

This computer pioneer’s recollections include the derivation of the standard hardware for multiply and divide developed by von Neumann.

Chapter 4

Curnow, H. J., and B. A. Wichmann [1976]. "A synthetic benchmark," *The Computer J.* 19 (1):80.

Describes the first major synthetic benchmark, Whetstone, and how it was created.

Flemming, P. J., and J. J. Wallace [1986]. "How not to lie with statistics: The correct way to summarize benchmark results," *Comm. ACM* 29:3 (March) 218–21.

Describes some of the underlying principles in using different means to summarize performance results.

McMahon, F. M. [1986]. "The Livermore FORTRAN kernels: A computer test of numerical performance range," Tech. Rep. UCRL-55745, Lawrence Livermore National Laboratory, Univ. of California, Livermore (December).

Describes the Livermore Loops—a set of Fortran kernel benchmarks.

Smith, J. E. [1988]. "Characterizing computer performance with a single number," *Comm. ACM* 31:10 (October) 1202–06.

Describes the difficulties of summarizing performance with just one number and argues for total execution time as the only consistent measure.

SPEC [2000]. *SPEC Benchmark Suite Release 1.0*, SPEC, Santa Clara, CA, October 2.

Describes the SPEC benchmark suite. For up-to-date information, see the SPEC Web page via a link at www.mkp.com/books_catalog/cod/links.htm.

Weicker, R. P. [1984]. "Dhrystone: A synthetic systems programming benchmark," *Comm. ACM* 27:10 (October) 1013–30.

Describes the Dhrystone benchmark and its construction.

Chapter 5

A basic Verilog tutorial is included on the CD. There are also many books both on Verilog and on digital design using Verilog.

Kidder, T. [1981]. *Soul of a New Machine*, Little, Brown, and Co., New York.

Describes the design of the Data General Eclipse series that replaced the first DG machines such as the Nova. Kidder records the intimate interactions among architects, hardware designers, microcoders, and project management.

Levy, H. M., and R. H. Eckhouse, Jr. [1989]. *Computer Programming and Architecture: The VAX*, Second ed., Digital Press, Bedford, MA.

Good description of the VAX architecture and several different microprogrammed implementations.

Patterson, D. A. [1983]. "Microprogramming," *Scientific American* 248:3 (March) 36–43.

Overview of microprogramming concepts.

Tucker, S. G. [1967]. "Microprogram control for the System/360," *IBM Systems J.* 6:4, 222–41.

Describes the microprogrammed control for the 360, the first microprogrammed commercial machine.

Wilkes, M. V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, MA.

Intriguing biography with many stories about industry pioneers and the trials and successes in building early machines.

Wilkes, M. V., and J. B. Stringer [1953]. "Microprogramming and the design of the control circuits in an electronic digital computer," *Proc. Cambridge Philosophical Society* 49:230–38. Also reprinted in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 158–63, 1982, and in "The Genesis of Microprogramming," in *Annals of the History of Computing* 8:116.

These two classic papers describe Wilkes's proposal for microcode.

Chapter 6

Bhandarkar, D., and D. W. Clark [1991]. "Performance from architecture: Comparing a RISC and a CISC with similar hardware organizations," *Proc. Fourth Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Palo Alto, CA, 310–19.

A quantitative comparison of RISC and CISC written by scholars who argued for CISCs as well as built them; they conclude that MIPS is between 2 and 4 times faster than a VAX built with similar technology, with a mean of 2.7.

Fisher, J. A., and B. R. Rau [1993]. *Journal of Supercomputing* (January), Kluwer.

This entire issue is devoted to the topic of exploiting ILP. It contains papers on both the architecture and software and is a wonderful source for further references.

Hennessy, J. L., and D. A. Patterson [2001]. *Computer Architecture: A Quantitative Approach*, third ed., San Francisco: Morgan Kaufmann.

Chapters 3 and 4 go into considerably more detail about pipelined processors (over 200 pages), including superscalar processors and VLIW processors.

Jouppi, N. P., and D. W. Wall [1989]. "Available instruction-level parallelism for superscalar and superpipelined processors," *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Boston, 272–82.

A comparison of deeply pipelined (also called superpipelined) and superscalar systems.

Kogge, P. M. [1981]. *The Architecture of Pipelined Computers*, New York: McGraw-Hill.

A formal text on pipelined control, with emphasis on underlying principles.

Russell, R. M. [1978]. "The CRAY-1 computer system," *Comm. of the ACM* 21:1 (January) 63–72.

A short summary of a classic computer, which uses vectors of operations to remove pipeline stalls.

Smith, A., and J. Lee [1984]. "Branch prediction strategies and branch target buffer design," *Computer* 17:1 (January) 6–22.

An early survey on branch prediction.

Smith, J. E., and A. R. Plezkun [1988]. "Implementing precise interrupts in pipelined processors," *IEEE Trans. on Computers* 37:5 (May) 562–73.

Covers the difficulties in interrupting pipelined computers.

Thornton, J. E. [1970]. *Design of a Computer: The Control Data 6600*, Glenview, IL: Scott, Foresman.

A classic book describing a classic computer, considered the first supercomputer.

Chapter 7

Conti, C., D. H. Gibson, and S. H. Pitowsky [1968]. "Structural aspects of the System/360 Model 85, part I: General organization," *IBM Systems J.* 7:1, 2–14.

A classic paper that describes the first commercial computer to use a cache and its resulting performance.

Jason F. Cantin and Mark D. Hill [2001]. "Cache performance for selected SPEC CPU2000 benchmarks," *SIGARCH Computer Architecture News*, 29:4 (September), 13 – 18.

A reference paper of cache miss rates for many cache sizes for the SPEC2000 benchmarks.

Hennessy, J., and D. Patterson [2003]. Chapter 5 in *Computer Architecture: A Quantitative Approach*, Third edition, Morgan Kaufmann Publishers, San Francisco.

For more in-depth coverage of a variety of topics including protection, cache performance of out-of-order processors, virtually addressed caches, multilevel caches, compiler optimizations, additional latency tolerance mechanisms, and cache coherency.

Kilburn, T., D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner [1962]. "One-level storage system," *IRE Transactions on Electronic Computers* EC-11 (April) 223–35. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 135–48, 1982.

This classic paper is the first proposal for virtual memory.

LaMarca, A. and R. E. Ladner [1996]. "The influence of caches on the performance of heaps," *ACM J. of Experimental Algorithmics*, vol.1, www.jea.acm.org/1996/LaMarcaInfluence/.

This paper shows the difference between complexity analysis of an algorithm, instruction count performance, and memory hierarchy for four sorting algorithms.

Przybylski, S. A. [1990]. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*, Morgan Kaufmann Publishers, San Francisco.

A thorough exploration of multilevel memory hierarchies and their performance.

Ritchie, D.M. and K. Thompson. "UNIX Timesharing System: The UNIX Timesharing System." *Bell System Technical Journal*, August 1978, pp. 1991–2019.

A paper describing the most elegant operating system ever invented.

Ritchie, Dennis. "The Evolution of the UNIX Timesharing System." *AT&T Bell Laboratories Technical Journal*, August 1984, pp. 1577–1593.

The history of UNIX from one of its inventors.

Silberschatz, A., P. Galvin, and G. Grange[2003]. *Operating System Concepts*, sixth edition, Addison-Wesley, Reading, MA.

An operating systems textbook with a thorough discussion of virtual memory, processes and process management, and protection issues.

Smith, A. J. [1982]. "Cache memories," *Computing Surveys* 14:3 (September) 473–530.

The classic survey paper on caches. This paper defined the terminology for the field and has served as a reference for many computer designers.

Smith, D.K. and R.C. Alexander. *Fumbling the Future: How Xerox Invented, Then Ignored, the First Personal Computer*. New York: Morrow, 1988.

A popular book that explains the role of Xerox PARC in laying the foundation for today's computing, which Xerox did not substantially benefit from.

Tanenbaum, A. [2001]. *Modern Operating Systems*, second edition, Prentice Hall, Upper Saddle River, NJ.

An operating system textbook with a good discussion of virtual memory.

Wilkes, M. [1965]. "Slave memories and dynamic storage allocation," *IEEE Trans. Electronic Computers* EC-14:2 (April) 270–71.

The first, classic paper on caches.

Chapter 8

Bashe, C. J., L. R. Johnson, J. H. Palmer, and E. W. Pugh [1986]. *IBM's Early Computers*, Cambridge, MA: MIT Press.

Describes the I/O system architecture and devices in IBM's early computers.

Brenner, P. [1997]. *A Technical Tutorial on the IEEE 802.11 Protocol* found on many Web sites.

A widely referenced short tutorial that outlives the startup company for which the author worked.

Chen, P. M., E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson [1994]. "RAID: High-performance, reliable secondary storage," *ACM Computing Surveys* 26:2 (June), 145–88.

A tutorial covering disk arrays and the advantages of such an organization.

Gray, J. [1990]. "A census of Tandem system availability between 1985 and 1990," *IEEE Transactions on Reliability* 39:4 (October), 409–18.

One of the first papers to categorize, quantify, and publish reasons for failures. It is still widely quoted.

Gray, J., and A. Reuter [1993]. *Transaction Processing: Concepts and Techniques*, San Francisco: Morgan Kaufmann.

A description of transaction processing, including discussions of benchmarking and performance evaluation.

Hennessy, J., and D. Patterson [2003]. *Computer Architecture: A Quantitative Approach*, third ed., San Francisco: Morgan Kaufmann Publishers, Chapters 7 and 8.

Chapter 7 focuses on storage, including an extensive discussion of RAID technologies and dependability. Chapter 8 focuses on networks.

Kahn, R. E. [1972]. "Resource-sharing computer communication networks," *Proc. IEEE* 60:11 (November), 1397–1407.

A classic paper that describes the ARPANET.

Laprie, J.-C. [1985]. "Dependable computing and fault tolerance: concepts and terminology," *15th Annual Int'l Symposium on Fault-Tolerant Computing FTCS 15*, Digest of Papers, Ann Arbor, MI (June 19–21), 2–11.

The paper that introduced standard definitions of dependability, reliability, and availability.

Levy, J. V. [1978]. "Buses: The skeleton of computer structures," in *Computer Engineering: A DEC View of Hardware Systems Design*, C. G. Bell, J. C. Mudge, and J. E. McNamara, eds., Bedford, MA: Digital Press.

This is a good overview of key concepts in bus design with some examples from DEC machines.

Lyman, P., and H. R. Varian [2003]. "How much information? 2003," <http://www.sims.berkeley.edu/research/projects/how-much-info-2003/>.

This project estimates the amount of information in the world from all possible sources.

Metcalfe, R. M., and D. R. Boggs [1976]. "Ethernet: Distributed packet switching for local computer networks," *Comm. ACM* 19:7 (July), 395–404.

A classic paper that describes the Ethernet network.

Myer, T. H., and I. E. Sutherland [1968]. "On the design of display processors," *Communications of the ACM* 11:6 (June), 410–14.

Another classic that notes how building powerful coprocessors can be a never-ending cycle.

Okada, S., Y. Matsuda, T. Yamada, and A. Kobayashi [1999]. "System on a chip for digital still camera," *IEEE Trans. on Consumer Electronics* 45:3 (August), 584–90.)

Oppenheimer, D., A. Ganapathi, and D. Patterson [2003]. "Why do Internet services fail, and what can be done about it?," *4th Usenix Symposium on Internet Technologies and Systems*, March 26–28, Seattle, WA.

A recent update on Gray's classic paper, this time focused on Internet sites.

Patterson, D., G. Gibson, and R. Katz [1988]. "A case for redundant arrays of inexpensive disks (RAID)," *SIGMOD Conference*. 109–16.

A classic paper that advocates arrays of smaller disks and introduces RAID levels.

Saltzer, J. H., D. P. Reed, and D. D. Clark [1984]. "End-to-end arguments in system design," *ACM Trans. on Computer Systems* 2:4 (November), 277–88.

A classic paper that defines the end-to-end argument.

Smotherman, M. [1989]. "A sequencing-based taxonomy of I/O systems and review of historical machines," *Computer Architecture News* 17:5 (September), 5–15.

Describes the development of important ideas in I/O.

Talagala, N., R. Arpaci-Dusseau, and D. Patterson [2000]. "Micro-benchmark based extraction of local and global disk characteristics," *U.C. Berkeley Technical Report CSD-99-1063*, June 13.

Describes a simple program to automatically deduce key parameters of disks.

Chapter 9

Almasi, G. S., and A. Gottlieb [1989]. *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, CA.

A textbook covering parallel computers.

Amdahl, G. M. [1967]. "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. AFIPS Spring Joint Computer Conf.*, Atlantic City, NJ, (April) 483–85.

Written in response to the claims of the Illiac IV, this three-page article describes Amdahl's law and gives the classic reply to arguments for abandoning the current form of computing.

Andrews, G. R. [1991]. *Concurrent Programming: Principles and Practice*, Benjamin/Cummings, Redwood City, CA.

A text that gives the principles of parallel programming.

Archibald, J., and J.-L. Baer [1986]. "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Trans. on Computer Systems* 4:4 (November), 273–98.

Classic survey paper of shared-bus cache coherence protocols.

Arpaci-Dusseau, A., R. Arpaci-Dusseau, D. Culler, J. Hellerstein, and D. Patterson [1997]. "High-performance sorting on networks of workstations," *Proc. ACM SIGMOD/PODS Conference on Management of Data*, Tucson, AZ, May 12–15.

How a world record sort was performed on a cluster, including architecture critique of the workstation and network interface. By April 1, 1997, they pushed the record to 8.6 GB in 1 minute and 2.2 seconds to sort 100 MB.

Bell, C. G. [1985]. "Multis: A new class of multiprocessor computers," *Science* 228 (April 26), 462–67.

Distinguishes shared address and nonshared address multiprocessors based on microprocessors.

Culler, D. E., and J. P. Singh, with A. Gupta [1998]. *Parallel Computer Architecture*, Morgan Kaufmann, San Francisco.

A textbook on parallel computers.

Falk, H. [1976]. "Reaching for the Giga-flop," *IEEE Spectrum* 13:10 (October), 65–70.

Chronicles the sad story of the Illiac IV: four times the cost and less than one-tenth the performance of original goals.

Flynn, M. J. [1966]. "Very high-speed computing systems," *Proc. IEEE* 54:12 (December), 1901–09.

Classic article showing SISD/SIMD/MISD/MIMD classifications.

Hennessy, J., and D. Patterson [2003]. Chapters 6 and 8 in *Computer Architecture: A Quantitative Approach*, third edition, Morgan Kaufmann Publishers, San Francisco.

A more in-depth coverage of a variety of multiprocessor and cluster topics, including programs and measurements.

Hord, R. M. [1982]. *The Illiac-IV, the First Supercomputer*, Computer Science Press, Rockville, MD.

A historical accounting of the Illiac IV project.

Hwang, K. [1993]. *Advanced Computer Architecture with Parallel Programming*, McGraw-Hill, New York.
Another textbook covering parallel computers.

Kozyrakis, C., and D. Patterson [2003]. "Scalable vector processors for embedded systems," *IEEE Micro* 23:6 (November–December), 36–45.

Examination of a vector architecture for the MIPS instruction set in media and signal processing.

Menabrea, L. F. [1842]. "Sketch of the analytical engine invented by Charles Babbage," *Bibliothèque Universelle de Genève* (October).

Certainly the earliest reference on multiprocessors, this mathematician made this comment while translating papers on Babbage's mechanical computer.

Pfister, G. F. [1998]. *In Search of Clusters: The Coming Battle in Lowly Parallel Computing*, second edition, Prentice-Hall, Upper Saddle River, NJ.

An entertaining book that advocates clusters and is critical of NUMA multiprocessors.

Seitz, C. [1985]. "The Cosmic Cube," *Comm. ACM* 28:1 (January), 22–31.

A tutorial article on a parallel processor connected via a hypertree. The Cosmic Cube is the ancestor of the Intel supercomputers.

Slotnick, D. L. [1982]. "The conception and development of parallel processors—A personal memoir," *Annals of the History of Computing* 4:1 (January), 20–30.

Recollections of the beginnings of parallel processing by the architect of the Illiac IV.

Appendix A

Sweetman, D. [1999]. *See MIPS Run*, Morgan Kaufmann Publishers, San Francisco, CA.

A complete, detailed, and engaging introduction to the MIPS instruction set and assembly language programming on these machines.

Detailed documentation on the MIPS32 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)
 (<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)
 (<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)
 (<http://mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/ArchitectureProgrammingPublicationsforMIPS32/MD00090-2B-MIPS32PRA-AFP-02.00.pdf/getDownload>)

Aho, A., R. Sethi, and J. Ullman [1985]. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA.

Slightly dated and lacking in coverage of modern architectures, but still the standard reference on compilers.

Appendix B

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*, second edition, Reading, MA: Addison Wesley.

A general text on logic design.

Wakerly, J. F. [2000]. *Digital Design: Principles and Practices*, third ed., Englewood Cliffs, NJ: Prentice-Hall.

A general text on logic design.