Memory Systems

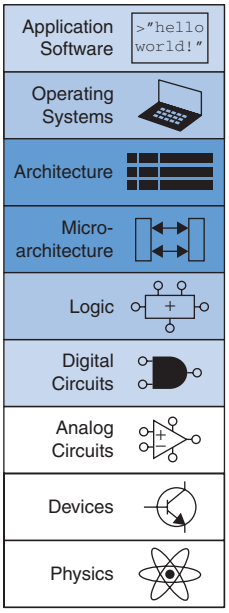
8

8.1 INTRODUCTION

Computer system performance depends on the memory system as well as the processor microarchitecture. Chapter 7 assumed an ideal memory system that could be accessed in a single clock cycle. However, this would be true only for a very small memory—or a very slow processor! Early processors were relatively slow, so memory was able to keep up. But processor speed has increased at a faster rate than memory speeds. DRAM memories are currently 10 to 100 times slower than processors. The increasing gap between processor and DRAM memory speeds demands increasingly ingenious memory systems to try to approximate a memory that is as fast as the processor. The first half of this chapter investigates memory systems and considers trade-offs of speed, capacity, and cost.

The processor communicates with the memory system over a *memory interface*. Figure 8.1 shows the simple memory interface used in our multicycle RISC-V processor. The processor sends an address over the *Address* bus to the memory system. For a read, *MemWrite* is 0 and the memory returns the data on the *ReadData* bus. For a write, *MemWrite* is 1 and the processor sends data to memory on the *WriteData* bus.

The major issues in memory system design can be broadly explained using a metaphor of books in a library. A library contains many books on the shelves. If you were writing a term paper on the meaning of dreams, you might go to the library[1] and pull Freud's *The Interpretation of Dreams* off the shelf and bring it to your cubicle. After skimming it, you might put it back and pull out Jung's *The Psychology of the Unconscious*. You might then go back for another quote from *Interpretation of Dreams*, followed by yet another trip to the stacks for Freud's *The Ego and the Id*. Pretty soon,



[1] We realize that library usage is plummeting among college students because of the Internet. But we also believe that libraries contain vast troves of hard-won human knowledge that are not electronically available. We hope that Web searching does not completely displace the art of library research.

Digital Design and Computer Architecture, RISC-V Edition. DOI: 10.1016/B978-0-12-820064-3.00008-8

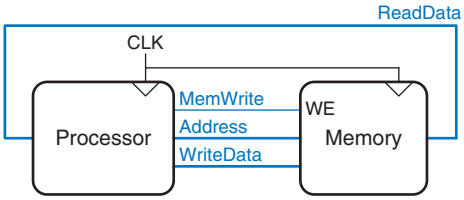


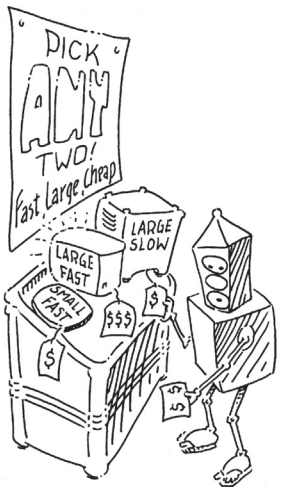
Figure 8.1 The memory interface

you would get tired of walking from your cubicle to the stacks. If you are clever, you would save time by keeping the books in your cubicle rather than schlepping them back and forth. Furthermore, when you pull a book by Freud, you could also pull several of his other books from the same shelf.

This metaphor emphasizes the principle, introduced in Section 6.2.1, of making the common case fast. By keeping books that you have recently used or might likely use in the future at your cubicle, you reduce the number of time-consuming trips to the stacks. In particular, you use the principles of *temporal* and *spatial locality*. Temporal locality means that if you have used a book recently, you are likely to use it again soon. Spatial locality means that when you use one particular book, you are likely to be interested in other books on the same shelf.

The library itself makes the common case fast by using these principles of locality. The library has neither the shelf space nor the budget to accommodate all of the books in the world. Instead, it keeps some of the lesser-used books in deep storage in the basement. Also, it may have an interlibrary loan agreement with nearby libraries so that it can offer more books than it physically carries.

In summary, you obtain the benefits of both a large collection and quick access to the most commonly used books through a hierarchy of storage. The most commonly used books are in your cubicle. A larger collection is on the shelves. And an even larger collection is available, with advanced notice, from the basement and other libraries. Similarly, memory systems use a hierarchy of storage to quickly access the most commonly used data while still having the capacity to store large amounts of data.



Memory subsystems used to build this hierarchy were introduced in Section 5.5. Computer memories are primarily built from dynamic RAM (DRAM) and static RAM (SRAM). Ideally, the computer memory system is fast, large, and cheap. In practice, a single memory has only two of these three attributes; it is either slow, small, or expensive. But computer systems can approximate the ideal by combining a fast, small, cheap memory and a slow, large, cheap memory. The fast memory stores the most commonly used data and instructions; so, on average, the memory system appears fast. The large memory stores the remainder of the data and instructions; so, the overall capacity is large. The combination of two cheap memories is much less expensive than a single large fast memory. These principles extend to using an entire hierarchy of memories of increasing capacity and decreasing speed.

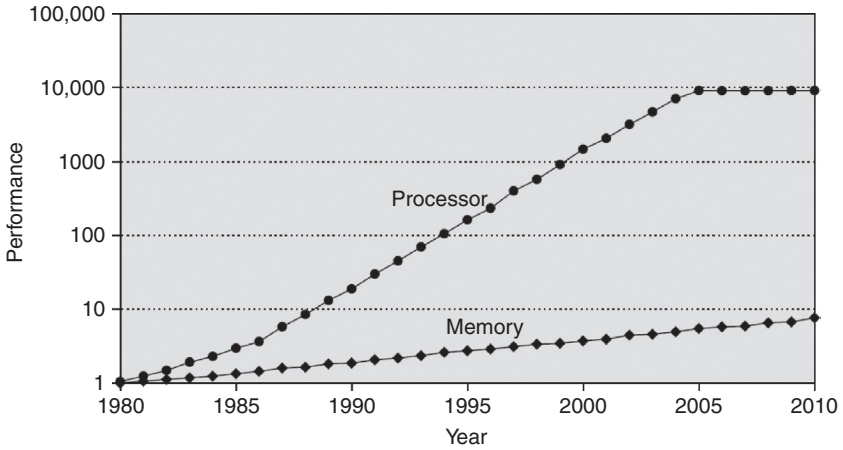


Figure 8.2 Diverging processor and memory performance. Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2011

Remember that speed is characterized by both latency and throughput. Memory latency is the time to access the first byte of information. Throughput is the number of bytes per second that can be delivered. Many memories have good throughput but long latency.

Computer memory is generally built from DRAM chips. In 2021, a typical PC had a *main memory* consisting of 8 to 32 GiB of DRAM, and DRAM cost about $3 per gibibyte (GiB). DRAM prices have declined at 15% to 25% per year for the last three decades and memory capacity has grown at the same rate, so the total cost of the memory in a PC has remained roughly constant. Unfortunately, DRAM speed has improved by only about 7% per year, whereas processor performance has improved at a rate of 25% to 50% per year, as shown in Figure 8.2. The plot shows memory (DRAM) and processor speeds with the 1980 speeds as a baseline. In about 1980, processor and memory speeds were the same. However, performance has diverged since then, with memories badly lagging.[2]

DRAM could keep up with processors in the 1970's and early 1980's, but it is now woefully too slow. The DRAM access time is one to two orders of magnitude longer than the processor cycle time (tens of nanoseconds, compared to less than one nanosecond). DRAM throughput is good, on the order of 30 GB/s.

To counteract this trend, computers store the most commonly used instructions and data in a faster but smaller memory, called a *cache*. The cache is usually built out of SRAM on the same chip as the processor. The cache speed is comparable to the processor speed because SRAM is inherently faster than DRAM and because the on-chip memory eliminates lengthy delays caused by traveling to and from a separate chip.

[2] Although recent single-processor performance has remained approximately constant, as shown in Figure 8.2 for the years 2005 to 2010, the increase in multicore systems (not depicted on the graph) only worsens the gap between processor and memory performance.

In 2021, on-chip SRAM costs were on the order of $100/GiB, but the cache is relatively small (kibibytes to several mebibytes), so the overall cost is low. Caches can store both instructions and data, but we will refer to their contents generically as "data." SRAM latency ranges from a few tenths of a nanosecond for a 16 KiB cache to several nanoseconds for a 4 MiB cache. Throughput can reach hundreds of GB/s.

If the processor requests data that is available in the cache, it is returned quickly. This is called a cache *hit*. Otherwise, the processor retrieves the data from main memory (DRAM). This is called a cache *miss*. If the cache hits most of the time, then the processor seldom has to wait for the slow main memory, and the average access time is low.

The third level in the memory hierarchy is the hard drive. In the same way that a library uses the basement to store books that do not fit in the stacks, computer systems use the hard drive to store data that does not fit in main memory. In 2021, a hard disk drive (HDD), built using magnetic storage, cost less than $0.03/GB and had an access time of about 5 to 10 ms. Throughput is on the order of 100 MB/s for large files down to 1 MB/s for random accesses to small (4 KiB) files. Hard disk costs have decreased at 60% per year but access times scarcely improved. Solid state drives (SSDs), built using flash memory technology, are an increasingly common alternative to HDDs. SSDs have been used by niche markets for over two decades, and they were introduced into the mainstream market in 2007. SSDs overcome some of the mechanical failures of HDDs, but they cost 3 to 4 times as much at $0.10/GB. Since SSDs hit the market, the price difference between them and HDDs has shrunk, and the popularity of SSDs over HDDs has increased accordingly. SSDs have access times of less than 0.1 ms. Throughput can be 500 to 3,000 MB/s for large files down to 50 to 250 MB/s for 4 KiB files.

The hard drive provides an illusion of more capacity than actually exists in the main memory. It is thus called *virtual memory*. Like books in the basement, data in virtual memory takes a long time to access. Main memory, also called *physical memory*, holds a subset of the virtual memory. Hence, the main memory can be viewed as a cache for the most commonly used data from the hard drive.

Figure 8.3 summarizes the memory hierarchy of the computer system discussed in the rest of this chapter. The processor first seeks data in a small but fast cache that is usually located on the same chip. If the

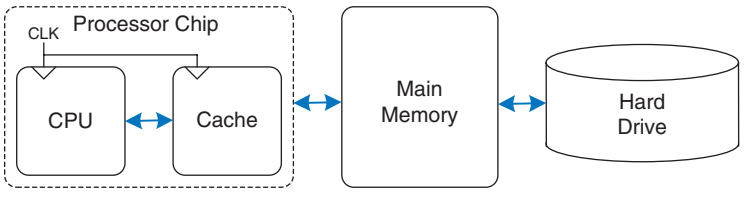
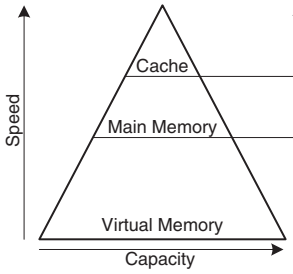


Figure 8.3 A typical memory hierarchy



Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$100	0.2–3	100+
DRAM	$3	10–50	30
SSD	$0.10	20,000	0.05–3
HDD	$0.03	5,000,000	0.001–0.1

Figure 8.4 Memory hierarchy components, with typical characteristics in 2021

data is not available in the cache, the processor then looks in main memory. If the data is not there either, the processor fetches the data from virtual memory on the large but slow hard disk. Figure 8.4 illustrates this capacity and speed trade-off in the memory hierarchy and lists typical costs, access times, and bandwidth in 2021 technology. As access time decreases, speed increases.

Section 8.2 introduces memory system performance analysis. Section 8.3 explores several cache organizations, and Section 8.4 delves into virtual memory systems.

8.2 MEMORY SYSTEM PERFORMANCE ANALYSIS

Designers (and computer buyers) need quantitative ways to measure the performance of memory systems to evaluate the cost-benefit trade-offs of various alternatives. Memory system performance metrics are *miss rate* or *hit rate* and *average memory access time*. Miss and hit rates are calculated as:

$$\begin{aligned} \textit{Miss Rate} &= \frac{\textit{Number of misses}}{\textit{Number of total memory accesses}} = 1 - \textit{Hit Rate} \\ \textit{Hit Rate} &= \frac{\textit{Number of hits}}{\textit{Number of total memory accesses}} = 1 - \textit{Miss Rate} \end{aligned} \tag{8.1}$$

Example 8.1 CALCULATING CACHE PERFORMANCE

Suppose a program has 2,000 data access instructions (loads or stores) and 1,250 of these requested data values are found in the cache. The other 750 data values are supplied to the processor by main memory or disk memory. What are the miss and hit rates for the cache?

Solution The miss rate is 750/2000 = 0.375 = 37.5%. The hit rate is 1250/2000 = 0.625 = 1 – 0.375 = 62.5%.

Average memory access time (*AMAT*) is the average time a processor must wait for memory per load or store instruction. In the typical computer system from Figure 8.3, the processor first looks for the data in the cache. If the cache misses, the processor then looks in main memory. If the main memory misses, the processor accesses virtual memory on the hard disk. Thus, *AMAT* is calculated as:

$$AMAT = t_{\text{cache}} + MR_{\text{cache}}(t_{MM} + MR_{MM}t_{VM}) \quad (8.2)$$

where t_{cache}, t_{MM}, and t_{VM} are the access times of the cache, main memory, and virtual memory, and MR_{cache} and MR_{MM} are the cache and main memory miss rates, respectively.

Example 8.2 CALCULATING AVERAGE MEMORY ACCESS TIME

Suppose a computer system has a memory organization with only two levels of hierarchy, a cache and main memory. What is the average memory access time given the access times and miss rates in Table 8.1?

Solution The average memory access time is 1 + 0.1(100) = 11 cycles.

Table 8.1 Access times and miss rates

Memory Level	Access Time (Cycles)	Miss Rate
Cache	1	10%
Main Memory	100	0%

Gene Amdahl, 1922–2015
Most famous for Amdahl's Law, an observation he made in 1965. While in graduate school, he began designing computers in his free time. This side work earned him his Ph.D. in theoretical physics in 1952. He joined IBM immediately after graduation and later went on to found three companies, including one called Amdahl Corporation in 1970.

Example 8.3 IMPROVING ACCESS TIME

An 11-cycle average memory access time means that the processor spends ten cycles waiting for data for every one cycle actually using that data. What cache miss rate is needed to reduce the average memory access time to 1.5 cycles given the access times in Table 8.1?

Solution If the miss rate is m, the average access time is 1 + 100m. Setting this time to 1.5 and solving for m requires a cache miss rate of 0.5%.

As a word of caution, performance improvements might not always be as good as they sound. For example, making the memory system ten times faster will not necessarily make a computer program run ten times as fast. If 50% of a program's instructions are loads and stores, a tenfold memory system improvement means only a 1.82-fold improvement in program performance. This general principle is called *Amdahl's*

Law, which says that the effort spent on increasing the performance of a subsystem is worthwhile only if the subsystem affects a large percentage of the overall performance.

8.3 CACHES

A cache holds commonly used memory data. The number of data words that it can hold is called the *capacity*, C. Because the capacity of the cache is smaller than that of main memory, the computer system designer must choose what subset of the main memory is kept in the cache.

When the processor attempts to access data, it first checks the cache for the data. If the cache hits, the data is available immediately. If the cache misses, the processor fetches the data from main memory and places it in the cache for future use. To accommodate the new data, the cache must *replace* old data. This section investigates these issues in cache design by answering the following questions: (1) What data is held in the cache? (2) How is data found? and (3) What data is replaced to make room for new data when the cache is full?

Cache: a hiding place especially for concealing and preserving provisions or implements.
—Merriam Webster Online Dictionary. 2021. www.merriam-webster.com

When reading the next sections, keep in mind that the driving force in answering these questions is the inherent spatial and temporal locality of data accesses in most applications. Caches use spatial and temporal locality to predict what data will be needed next. If a program accesses data in a random order, it would not benefit from a cache.

As we explain in the following sections, caches are specified by their capacity (C), number of sets (S), block size (b), number of blocks (B), and degree of associativity (N).

Although we focus on data cache loads, the same principles apply for fetches from an instruction cache. Data cache store operations are similar and are discussed further in Section 8.3.4.

8.3.1 What Data is Held in the Cache?

An ideal cache would anticipate all of the data needed by the processor and fetch it from main memory ahead of time so that the cache has a zero miss rate. Because it is impossible to predict the future with perfect accuracy, the cache must guess what data will be needed based on the past pattern of memory accesses. In particular, the cache exploits temporal and spatial locality to achieve a low miss rate.

Recall that temporal locality means that the processor is likely to access a piece of data again soon if it has accessed that data recently. Therefore, when the processor loads or stores data that is not in the cache, the data is copied from main memory into the cache. Subsequent requests for that data hit in the cache.

Recall that spatial locality means that, when the processor accesses a piece of data, it is also likely to access data in nearby memory locations.

Therefore, when the cache fetches one word from memory, it may also fetch several adjacent words. This group of words is called a *cache block* or *cache line*. The number of words in the cache block, b, is called the *block size*. A cache of capacity C contains $B = C/b$ blocks.

The principles of temporal and spatial locality have been experimentally verified in real programs. If a variable is used in a program, the same variable is likely to be used again, creating temporal locality. If an element in an array is used, other elements in the same array are also likely to be used, creating spatial locality.

8.3.2 How is Data Found?

A cache is organized into S *sets*, each of which holds one or more blocks of data. The relationship between the address of data in main memory and the location of that data in the cache is called the *mapping*. Each memory address maps to exactly one set in the cache. Some of the address bits are used to determine which cache set contains the data. If the set contains more than one block, the data may be kept in any of the blocks in the set.

Caches are categorized based on the number of blocks in a set. In a *direct mapped* cache, each set contains exactly one block, so the cache has $S = B$ sets. Thus, a particular main memory address maps to a unique block in the cache. In an *N-way set associative* cache, each set contains N blocks. The address still maps to a unique set, with $S = B/N$ sets. But the data from that address can go in any of the N blocks in that set. A *fully associative* cache has only $S = 1$ set. Data can go in any of the B blocks in the set. Hence, a fully associative cache is another name for a B-way set associative cache.

To illustrate these cache organizations, we will consider a RISC-V memory system with 32-bit addresses and 32-bit words. The memory is byte-addressable, and each word is four bytes, so the memory consists of 2^{30} words aligned on word boundaries. We analyze caches with an eight-word capacity (C) for the sake of simplicity. We begin with a one-word block size (b), then generalize later to larger blocks.

Direct Mapped Cache

A *direct mapped* cache has one block in each set, so it is organized into $S = B$ sets. To understand the mapping of memory addresses onto cache blocks, imagine main memory as being mapped into b-word blocks, just as the cache is. An address in block 0 of main memory maps to set 0 of the cache. An address in block 1 of main memory maps to set 1 of the cache, and so forth until an address in block $B - 1$ of main memory maps to block $B - 1$ of the cache. There are no more blocks of the cache, so the mapping wraps around, such that block B of main memory maps to block 0 of the cache.

This mapping is illustrated in Figure 8.5 for a direct mapped cache with a capacity of eight words and a block size of one word. The cache

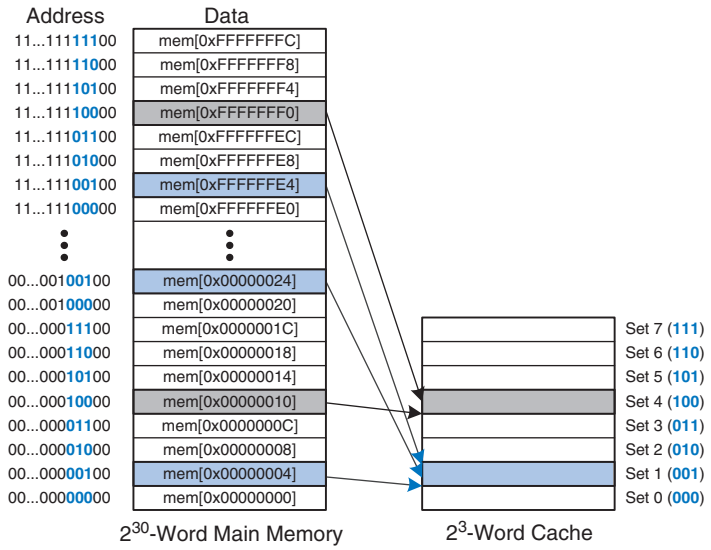


Figure 8.5 Mapping of main memory to a direct mapped cache

has eight sets, each of which contains a one-word block. The bottom two bits of the address are always 00, because they are word aligned. The next $\log_2 8 = 3$ bits indicate the set onto which the memory address maps. Thus, the data at addresses 0x00000004, 0x00000024, …, 0xFFFFFFE4 all map to set 1, as shown in blue. Likewise, data at addresses 0x00000010, …, 0xFFFFFFF0 all map to set 4, and so forth. Each main memory address maps to exactly one set in the cache.

Example 8.4 CACHE FIELDS

To what cache set in Figure 8.5 does the word at address 0x00000014 map? Name another address that maps to the same set.

Solution The two least significant bits of the address are 00, because the address is word aligned. The next three bits are 101, so the word maps to set 5. Words at addresses 0x34, 0x54, 0x74, …, 0xFFFFFFF4 all map to this same set.

Because many addresses map to a single set, the cache must also keep track of the address of the data actually contained in each set. The least significant bits of the address specify which set holds the data. The remaining most significant bits are called the *tag* and indicate which of the many possible addresses is held in that set.

In our previous examples, the two least significant bits of the 32-bit address are called the *byte offset* because they indicate the byte within the word. The next three bits are called the *set bits* because they indicate the set to which the address maps. (In general, the number of set bits is $\log_2 S$.) The remaining 27 tag bits indicate the memory address of the

Figure 8.6 Cache fields for address 0xFFFFFFE4 when mapping to the cache in Figure 8.5

data stored in a given cache set. Figure 8.6 shows the cache fields for address 0xFFFFFFE4. It maps to set 1 and its tag is all 1's.

Example 8.5 CACHE FIELDS

Find the number of set and tag bits for a direct mapped cache with 1024 (2^{10}) sets and a one-word block size. The address size is 32 bits.

Solution A cache with 2^{10} sets requires $\log_2(2^{10}) = 10$ set bits. The two least significant bits of the address are the byte offset, and the remaining $32 - 10 - 2 = 20$ bits form the tag.

Sometimes, such as when the computer first starts up, the cache sets contain no data at all. The cache uses a *valid bit* for each set to indicate whether the set holds meaningful data. If the valid bit is 0, the contents are meaningless.

Figure 8.7 shows the hardware for the direct mapped cache of Figure 8.5. The cache is constructed as an eight-entry SRAM. Each entry, or set, contains one line consisting of 32 bits of data, 27 bits of tag, and 1 valid bit. The cache is accessed using the 32-bit address. The two least significant bits, the byte offset bits, are ignored for word accesses. The next three bits, the set bits, specify the entry or set in the cache. A load instruction reads the specified entry from the cache and checks

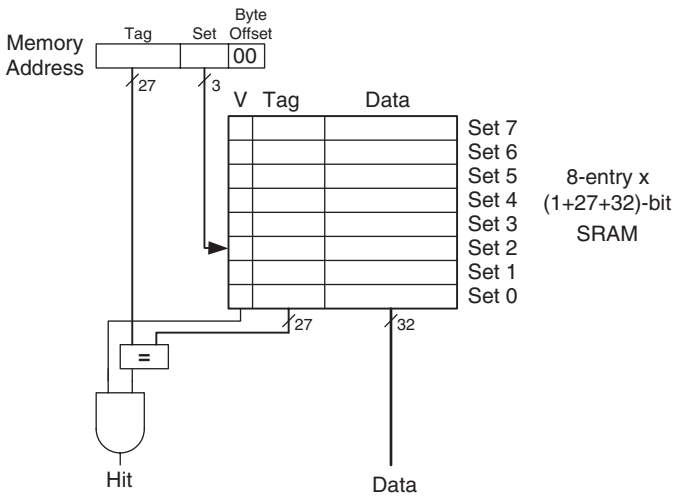


Figure 8.7 Direct mapped cache with 8 sets

the tag and valid bits. If the tag matches the most significant 27 bits of the requested address and the valid bit is 1, the cache hits and the data is returned to the processor. Otherwise, the cache misses and the memory system must fetch the data from main memory.

Example 8.6 TEMPORAL LOCALITY WITH A DIRECT MAPPED CACHE

Loops are a common source of temporal and spatial locality in applications. Using the eight-entry cache of Figure 8.7, show the contents of the cache after executing the following silly loop in RISC-V assembly code. Assume that the cache is initially empty. What is the miss rate?

```
       addi s0, zero, 5
       addi s1, zero, 0
LOOP:  beq  s0, zero, DONE
       lw   s2, 4(s1)
       lw   s3, 12(s1)
       lw   s4, 8(s1)
       addi s0, s0, -1
       j    LOOP
DONE:
```

Solution The program contains a loop that repeats for five iterations. Each iteration involves three memory accesses (loads), resulting in 15 total memory accesses. The first time the loop executes, the cache is empty and the data must be fetched from main memory locations 0x4, 0xC, and 0x8 into cache sets 1, 3, and 2, respectively. However, the next four times the loop executes, the data is found in the cache. Figure 8.8 shows the contents of the cache during the last request to memory address 0x4. The tags are all 0 because the upper 27 bits of the addresses are 0. The miss rate is 3/15 = 20%.

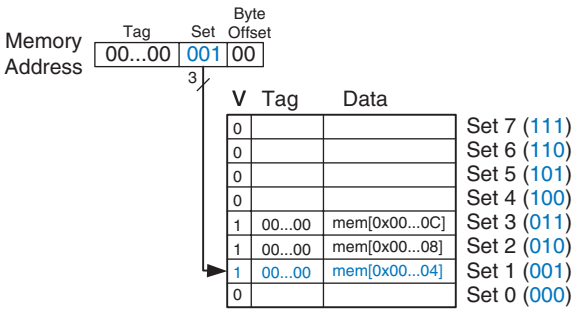


Figure 8.8 Direct mapped cache contents

When two recently accessed addresses map to the same cache block, a *conflict* occurs, and the most recently accessed address *evicts* the previous one from the block. Direct mapped caches have only one block in each

set, so two addresses that map to the same set always cause a conflict. Example 8.7 illustrates conflicts.

Example 8.7 CACHE BLOCK CONFLICT

What is the miss rate when the following loop is executed on the eight-word direct mapped cache from Figure 8.7? Assume that the cache is initially empty.

```
       addi s0, zero, 5
       addi s1, zero, 0
LOOP:  beq  s0, zero, DONE
       lw   s2, 0x4(s1)
       lw   s4, 0x24(s1)
       addi s0, s0, -1
       j    LOOP

DONE:
```

Solution Memory addresses 0x4 and 0x24 both map to set 1. During the initial execution of the loop, data at address 0x4 is loaded into set 1 of the cache. Then, data at address 0x24 is loaded into set 1, evicting the data from address 0x4. Upon the second execution of the loop, the pattern repeats and the cache must refetch data at address 0x4, evicting data from address 0x24. The two addresses conflict and the miss rate is 100%.

Multiway Set Associative Cache

An *N-way set associative* cache reduces conflicts by providing *N* blocks in each set where data mapping to that set might be found. Each memory address still maps to a specific set, but it can map to any one of the *N* blocks

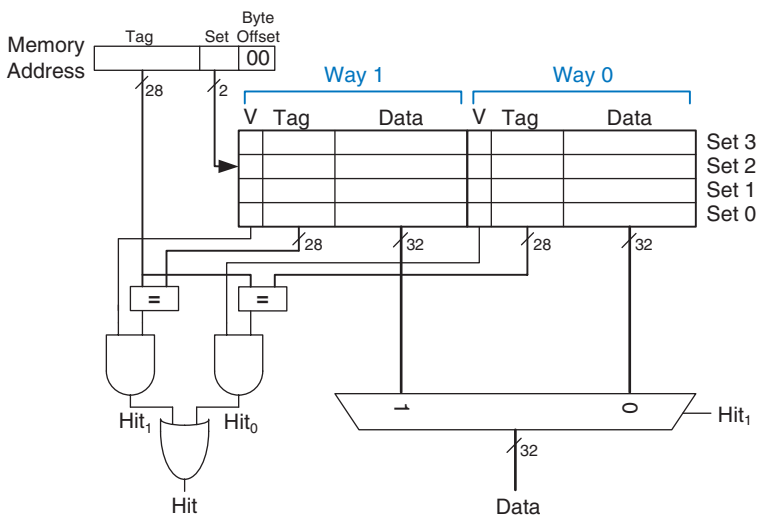


Figure 8.9 Two-way set associative cache

in the set. Hence, a direct mapped cache is another name for a one-way set associative cache. N is also called the *degree of associativity* of the cache.

Figure 8.9 shows the hardware for a $C = 8$-word, $N = 2$-way set associative cache. The cache now has only $S = 4$ sets rather than 8. Thus, only $\log_2 4 = 2$ set bits rather than 3 are used to select the set. The tag increases from 27 to 28 bits. Each set contains two *ways* or degrees of associativity. Each way consists of a data block and the valid and tag bits. The cache reads blocks from both ways in the selected set and checks the tags and valid bits for a hit. If a hit occurs in one of the ways, a multiplexer selects data from that way.

Set associative caches generally have lower miss rates than direct mapped caches of the same capacity because they have fewer conflicts. However, set associative caches are usually slower and somewhat more expensive to build because of the output multiplexer and additional comparators. They also raise the question of which way to replace when both ways are full; this is addressed further in Section 8.3.3. Most commercial systems use set associative caches.

Example 8.8 SET ASSOCIATIVE CACHE MISS RATE

Repeat Example 8.7 using the eight-word two-way set associative cache from Figure 8.9.

Solution Both memory accesses, to addresses 0x4 and 0x24, map to set 1. However, the cache has two ways, so it can accommodate data from both addresses. During the first loop iteration, the empty cache misses both addresses and loads both words of data into the two ways of set 1, as shown in Figure 8.10. On the next four iterations, the cache hits. Hence, the miss rate is 2/10 = 20%. Recall that the direct mapped cache of the same size from Example 8.7 had a miss rate of 100%.

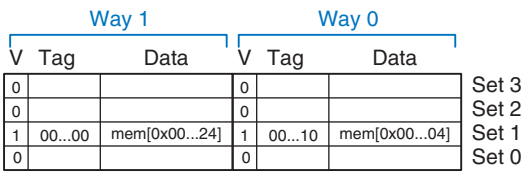


Figure 8.10 Two-way set associative cache contents

Fully Associative Cache

A *fully associative* cache contains a single set with B ways, where B is the number of blocks. A memory address can map to a block in any of these ways. A fully associative cache is another name for a B-way set associative cache with one set.

Figure 8.11 shows the SRAM array of a fully associative cache with eight blocks. Upon a data request, eight tag comparisons (not shown) must be made because the data could be in any block. Similarly, an 8:1

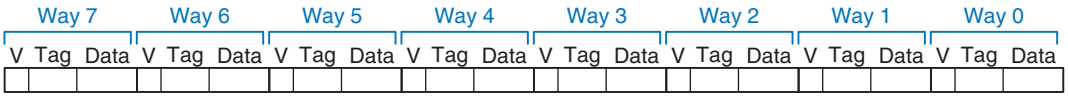


Figure 8.11 Eight-block fully associative cache

multiplexer chooses the proper data if a hit occurs. Fully associative caches tend to have the fewest conflict misses for a given cache capacity, but they require more hardware for additional tag comparisons. They are best suited to relatively small caches because of the large number of comparators.

Block Size

The previous examples were able to take advantage only of temporal locality because the block size was one word. To exploit spatial locality, a cache uses larger blocks to hold several consecutive words.

The advantage of a block size greater than one is that when a miss occurs and the word is fetched into the cache, the adjacent words in the block are also fetched. Therefore, subsequent accesses are more likely to hit because of spatial locality. However, a large block size means that a fixed-size cache will have fewer blocks. This may lead to more conflicts, increasing the miss rate. Moreover, it takes more time to fetch the missing cache block after a miss because more than one data word is fetched from main memory. The time required to load the missing block into the cache is called the *miss penalty*. If the adjacent words in the block are not accessed later, the effort of fetching them is wasted. Nevertheless, most real programs benefit from larger block sizes.

Figure 8.12 shows the hardware for a $C = 8$-word direct mapped cache with a $b = 4$-word block size. The cache now has only $B = C/b = 2$ blocks. A direct mapped cache has one block in each set, so this cache

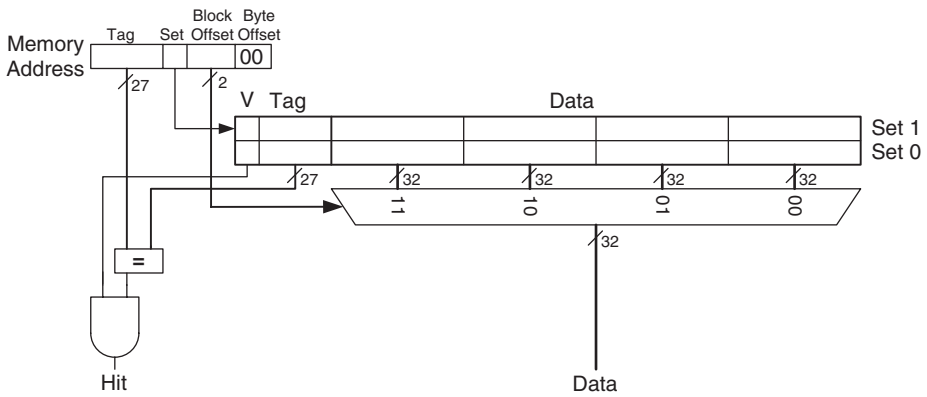


Figure 8.12 Direct mapped cache with two sets and a four-word block size

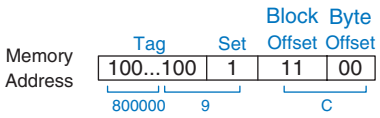


Figure 8.13 Cache fields for address 0x8000009C when mapping to the cache of Fig. 8.12

is organized as two sets. Thus, only $\log_2 2 = 1$ bit is used to select the set. A multiplexer is now needed to select the word within the block. The multiplexer is controlled by the $\log_2 4 = 2$ *block offset bits* of the address. The most significant 27 address bits form the tag. Only one tag is needed for the entire block, because the words in the block are at consecutive addresses.

Figure 8.13 shows the cache fields for address 0x8000009C when it maps to the direct mapped cache of Figure 8.12. The byte offset bits are always 0 for word accesses. The next $\log_2 b = 2$ block offset bits indicate the word within the block and the next bit indicates the set. The remaining 27 bits are the tag. Therefore, word 0x8000009C maps to set 1, word 3 in the cache. The principle of using larger block sizes to exploit spatial locality also applies to associative caches.

Example 8.9 SPATIAL LOCALITY WITH A DIRECT MAPPED CACHE

Repeat Example 8.6 for the eight-word direct mapped cache with a four-word block size.

Solution Figure 8.14 shows the contents of the cache after the first memory access. On the first loop iteration, the cache misses on the access to memory address 0x4. This access loads data at addresses 0x0 through 0xC into the cache block. All subsequent accesses (as shown for address 0xC) hit in the cache. Hence, the miss rate is 1/15 = 6.67%.

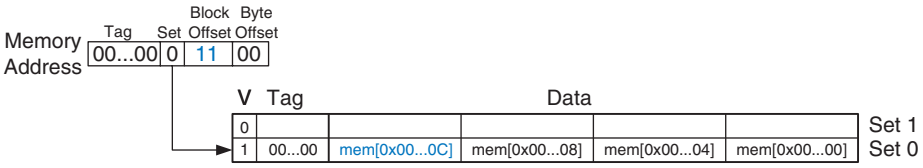


Figure 8.14 Cache contents with a block size *b* of four words

Putting it All Together

Caches are organized as two-dimensional arrays. The rows are called *sets*, and the columns are called *ways*. Each entry in the array

Table 8.2 Cache organizations

Organization	Number of Ways (N)	Number of Sets (S)
Direct Mapped	1	B
Set Associative	$1 < N < B$	B/N
Fully Associative	B	1

consists of a data block and its associated valid and tag bits. Caches are characterized by

- capacity C
- block size b (and number of blocks, $B = C/b$)
- number of blocks in a set (N)

Table 8.2 summarizes the various cache organizations. Each address in memory maps to only one set but can be stored in any of the ways.

Cache capacity, associativity, set size, and block size are typically powers of two. This makes the cache fields (tag, set, and block offset bits) subsets of the address bits.

Increasing the associativity N usually reduces the miss rate caused by conflicts. But higher associativity requires more tag comparators. Increasing the block size b takes advantage of spatial locality to reduce the miss rate. However, it decreases the number of sets in a fixed sized cache and, therefore, could lead to more conflicts. It also increases the miss penalty.

8.3.3 What Data is Replaced?

In a direct mapped cache, each address maps to a unique block and set. If a set is full when new data must be loaded, the block in that set is replaced with the new data. In set associative and fully associative caches, the cache must choose which block to evict when a cache set is full. The principle of temporal locality suggests that the best choice is to evict the least recently used block because it is least likely to be used again soon. Hence, most associative caches have a *least recently used* (*LRU*) replacement policy.

In a two-way set associative cache, a *use bit*, U, indicates which way within a set was least recently used. Each time one of the ways is used, U is adjusted to indicate the other way. For set associative caches with more than two ways, tracking the least recently used way becomes complicated. To simplify the problem, the ways are often divided into two groups and U indicates which *group* of ways was least recently used. Upon replacement, the new block replaces a random

block within the least recently used group. Such a policy is called *pseudo-LRU* and is good enough in practice.

Example 8.10 LRU REPLACEMENT

Show the contents of an eight-word two-way set associative cache after executing the following code. Assume LRU replacement, a block size of one word, and an initially empty cache.

```
addi t0, zero, 0
lw   s1, 0x4(t0)
lw   s2, 0x24(t0)
lw   s3, 0x54(t0)
```

Solution The first two instructions load data from memory addresses 0x4 and 0x24 into set 1 of the cache, shown in Figure 8.15(a). $U = 0$ indicates that data in way 0 was the least recently used. The next memory access, to address 0x54, also maps to set 1 and replaces the least recently used data in way 0, as shown in Figure 8.15(b). The use bit U is set to 1 to indicate that data in way 1 was the least recently used.

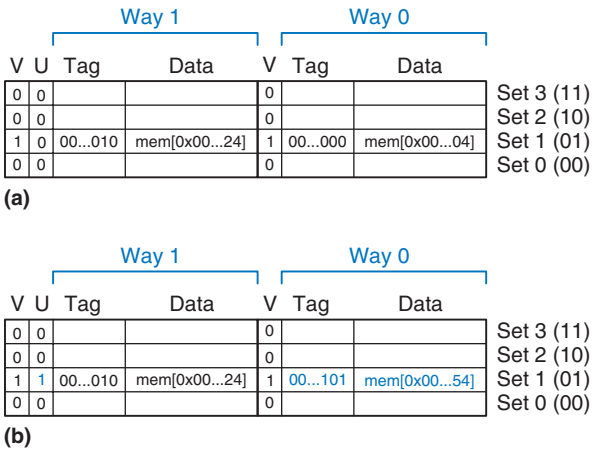


Figure 8.15 Two-way associative cache with LRU replacement

8.3.4 Advanced Cache Design*

Modern systems use multiple levels of caches to decrease memory access time. This section explores the performance of a two-level caching system and examines how block size, associativity, and cache capacity affect miss rate. The section also describes how caches handle stores, or writes, by using a write-through or write-back policy.

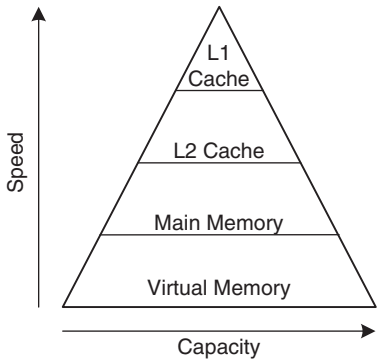


Figure 8.16 Memory hierarchy with two levels of cache

Multiple-Level Caches

Large caches are beneficial because they are more likely to hold data of interest and, therefore, have lower miss rates. However, large caches tend to be slower than small ones. Modern systems often use at least two levels of caches, as shown in Figure 8.16. The first-level (L1) cache is small enough to provide a one- or two-cycle access time. The second-level (L2) cache is also built from SRAM but is larger—and, therefore, slower—than the L1 cache. The processor first looks for the data in the L1 cache. If the L1 cache misses, the processor looks in the L2 cache. If the L2 cache misses, the processor fetches the data from main memory. Many modern systems add even more levels of cache to the memory hierarchy because accessing main memory is so slow.

Example 8.11 SYSTEM WITH AN L2 CACHE

Use the system of Figure 8.16 with access times of 1, 10, and 100 cycles for the L1 cache, L2 cache, and main memory, respectively. Assume that the L1 and L2 caches have miss rates of 5% and 20%, respectively. Specifically, of the 5% of accesses that miss the L1 cache, 20% of those also miss the L2 cache. What is the average memory access time (*AMAT*)?

Solution Each memory access checks the L1 cache. When the L1 cache misses (5% of the time), the processor checks the L2 cache. When the L2 cache misses (20% of the time), the processor fetches the data from main memory. Using Equation 8.2, we calculate the average memory access time as follows: 1 cycle + 0.05[10 cycles + 0.2(100 cycles)] = 2.5 cycles.

The L2 miss rate is high because it receives only the "hard" memory accesses, those that miss in the L1 cache. If all accesses went directly to the L2 cache, the L2 miss rate would be about 1%.

Reducing Miss Rate

Cache misses can be reduced by changing capacity, block size, and/or associativity. The first step to reducing the miss rate is to understand the causes of the misses. The misses can be classified as compulsory, capacity, and conflict. The first request to a cache block is called a *compulsory miss*, because the block must be read from memory regardless of the cache design. *Capacity misses* occur when the cache is too small to hold all concurrently used data. *Conflict misses* are caused when several addresses map to the same set and evict blocks that are still needed.

Changing cache parameters can affect one or more types of cache miss. For example, increasing cache capacity can reduce conflict and capacity misses, but it does not affect compulsory misses. On the other hand, increasing block size could reduce compulsory misses (due to spatial locality) but might actually *increase* conflict misses (because more addresses would map to the same set and could conflict).

Memory systems are complicated enough that the best way to evaluate their performance is by running benchmarks while varying cache parameters. Figure 8.17 plots miss rate versus cache size and degree of associativity for the SPEC2000 benchmark. This benchmark has a small number of compulsory misses, shown by the dark region near the x-axis. As expected, when cache size increases, capacity misses decrease. Increased associativity, especially for small caches, decreases the number of conflict misses shown along the top of the curve. Increasing associativity beyond four or eight ways provides only small decreases in miss rate.

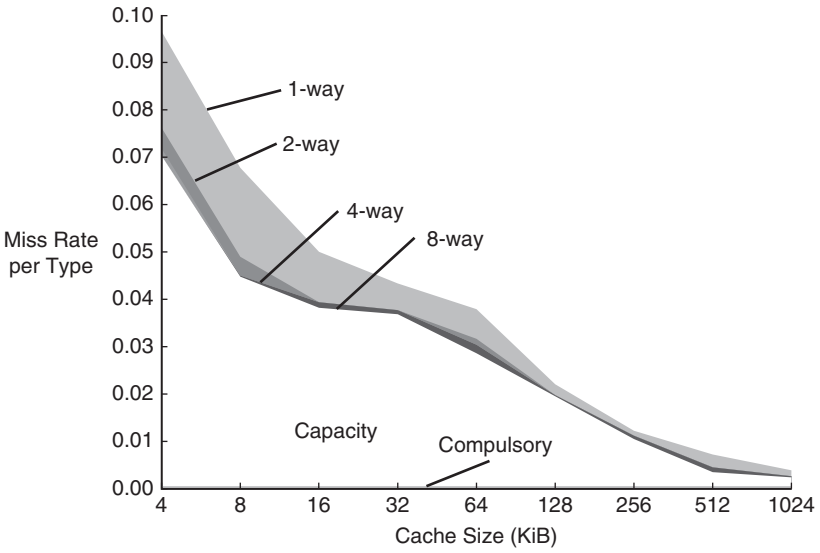


Figure 8.17 Miss rate versus cache size and associativity on SPEC2000 benchmark Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2012

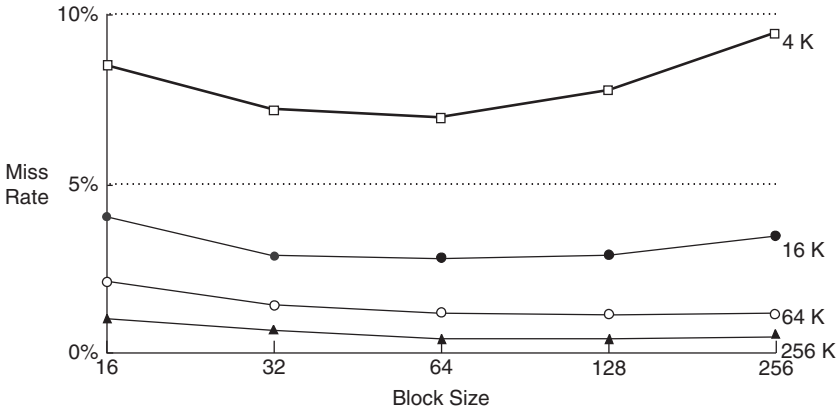


Figure 8.18 Miss rate versus block size and cache size on SPEC92 benchmark
Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2012

As mentioned, miss rate can also be decreased by using larger block sizes that take advantage of spatial locality. But as block size increases, the number of sets in a fixed-size cache decreases, increasing the probability of conflicts. Figure 8.18 plots miss rate versus block size (in number of bytes) for caches of varying capacity. For small caches, such as the 4 KiB cache, increasing the block size beyond 64 bytes *increases* the miss rate because of conflicts. For larger caches, increasing the block size beyond 64 bytes does not change the miss rate. However, large block sizes might still increase execution time because of the larger miss penalty, the time required to fetch the missing cache block from main memory.

Write Policy

The previous sections focused on memory loads. Memory stores, or writes, follow a similar procedure as loads. Upon a memory store, the processor checks the cache. If the cache misses, the cache block is fetched from main memory into the cache, and then the appropriate word in the cache block is written. If the cache hits, the word is simply written to the cache block.

Caches are classified as either write-through or write-back. In a *write-through* cache, the data written to a cache block is simultaneously written to main memory. In a *write-back* cache, a *dirty bit* (D) is associated with each cache block. D is 1 when the cache block has been written and 0 otherwise. Dirty cache blocks are written back to main memory only when they are evicted from the cache. A write-through cache requires no dirty bit but usually requires more main memory writes than a write-back cache. Modern caches are usually write-back because main memory access time is so large.

Example 8.12 WRITE-THROUGH VERSUS WRITE-BACK

Suppose a cache has a block size of four words. How many main memory accesses are required by the following code when using each write policy: write-through or write-back?

```
addi t5, zero, 0
sw   t1, 0(t5)
sw   t2, 12(t5)
sw   t3, 8(t5)
sw   t4, 4(t5)
```

Solution All four store instructions write to the same cache block. With a write-through cache, each store instruction writes a word to main memory, requiring four main memory writes. A write-back policy requires only one main memory access, when the dirty cache block is evicted.

8.4 VIRTUAL MEMORY

Most modern computer systems use a *hard drive* made of magnetic or solid-state storage as the lowest level in the memory hierarchy (see Figure 8.4). Compared with the ideal large, fast, cheap memory, a hard drive is large and cheap but terribly slow. It provides a much larger capacity than is possible with a cost-effective main memory (DRAM). However, if a significant fraction of memory accesses involve the hard drive, performance is dismal. You may have encountered this on a PC when running too many programs at once.

Figure 8.19 shows a hard drive made of magnetic storage, also called a *hard disk*, with the lid of its case removed. As the name implies, the hard disk contains one or more rigid disks or *platters*, each of which has a *read/write head* on the end of a long triangular arm. The head moves to the correct location on the disk and reads or writes data magnetically as the disk rotates beneath it. The head takes several milliseconds to *seek* the correct location on the disk, which is fast from a human perspective but millions of times slower than the processor. Hard disk drives are increasingly being replaced by solid-state drives because reading is orders of magnitude faster (see Figure 8.4) and they are not as susceptible to mechanical failures.

The objective of adding a hard drive to the memory hierarchy is to inexpensively give the illusion of a very large memory while still providing the speed of faster memory for most accesses. A computer with only 16 GiB of DRAM, for example, could effectively provide 128 GiB of memory using the hard drive. This larger 128 GiB memory is called *virtual memory*, and the smaller 16 GiB main memory is called *physical*

Figure 8.19 Hard disk

A computer with 32-bit addresses can access a maximum of 2^{32} bytes = 4 GiB of memory. This is one of the motivations for moving to 64-bit computers, which can access far more memory.

memory. We will use the term *physical memory* to refer to main memory throughout this section.

Programs can access data anywhere in virtual memory, so they must use *virtual addresses* that specify the location in virtual memory. The physical memory holds a subset of most recently accessed virtual memory. In this way, physical memory acts as a cache for virtual memory. Thus, most accesses hit in physical memory at the speed of DRAM, yet the program enjoys the capacity of the larger virtual memory.

Virtual memory systems use different terminologies for the same caching principles discussed in Section 8.3. Table 8.3 summarizes the analogous terms. Virtual memory is divided into *virtual pages*, typically 4 KiB in size. Physical memory is likewise divided into *physical pages* of the same size. A virtual page may be located in physical memory (DRAM) or on the hard drive. For example, Figure 8.20 shows a virtual memory that is larger than physical memory. The rectangles indicate pages. Some virtual pages are present in physical memory, and some are located on the hard drive. The process of determining the physical

Table 8.3 Analogous cache and virtual memory terms

Cache	Virtual Memory
Block	Page
Block size	Page size
Block offset	Page offset
Miss	Page fault
Tag	Virtual page number

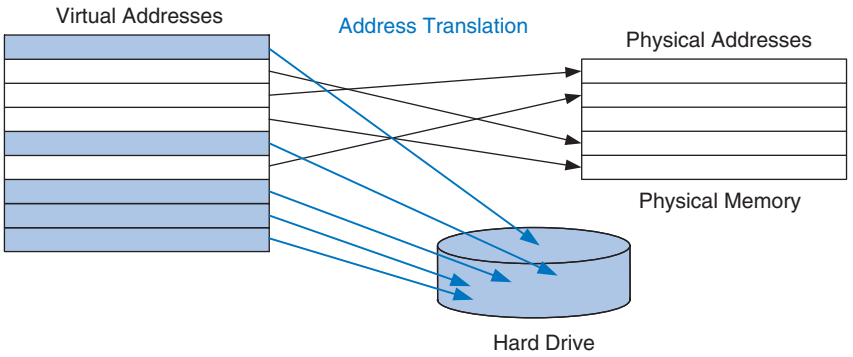


Figure 8.20 Virtual and physical pages

address from the virtual address is called *address translation*. If the processor attempts to access a virtual address that is not in physical memory, a *page fault* occurs and the operating system (OS) loads the page from the hard drive into physical memory.

To avoid page faults caused by conflicts, any virtual page can map to any physical page. In other words, physical memory behaves as a fully associative cache for virtual memory. In a conventional fully associative cache, every cache block has a comparator that checks the most significant address bits against a tag to determine whether the request hits in the block. In an analogous virtual memory system, each physical page would need a comparator to check the most significant virtual address bits against a tag to determine whether the virtual page maps to that physical page.

A realistic virtual memory system has so many physical pages that providing a comparator for each page would be excessively expensive. Instead, the virtual memory system uses a page table to perform address translation. A page table contains an entry for each virtual page, indicating its location in physical memory or that it is on the hard drive. Each load or store instruction requires a page table access followed by a physical memory access. The page table access translates the virtual address

used by the program to a physical address. The physical address is then used to actually read or write the data.

The page table is usually so large that it is located in physical memory. Hence, each load or store involves two physical memory accesses: a page table access and a data access. To speed up address translation, a translation lookaside buffer (TLB) caches the most commonly used page table entries.

The remainder of this section elaborates on address translation, page tables, and TLBs.

8.4.1 Address Translation

In a system with virtual memory, programs use virtual addresses so that they can access a large memory. The computer must translate these virtual addresses to either find the address in physical memory or take a page fault and fetch the data from the hard drive.

Recall that virtual memory and physical memory are divided into pages. The most significant bits of the virtual or physical address specify the virtual or physical *page number*. The least significant bits specify the word within the page and are called the *page offset*.

Figure 8.21 illustrates the page organization of a virtual memory system with 2 GiB of virtual memory and 128 MiB of physical memory divided into 4 KiB pages. MIPS accommodates 32-bit addresses. With a 2 GiB = 2^{31}-byte virtual memory, only the least significant 31 virtual address bits are used; the 32nd bit is always 0. Similarly, with a 128 MiB = 2^{27}-byte physical memory, only the least significant 27 physical address bits are used; the upper 5 bits are always 0.

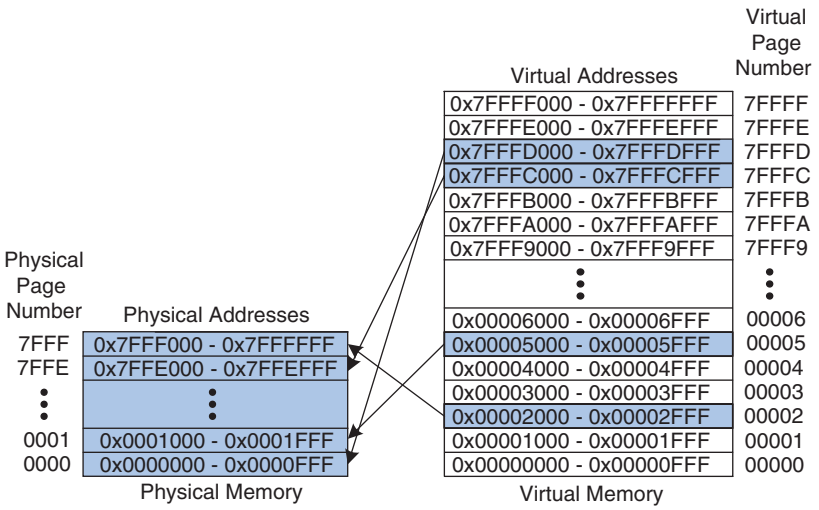


Figure 8.21 Physical and virtual pages

Because the page size is 4 KiB = 2^{12} bytes, there are $2^{31}/2^{12} = 2^{19}$ virtual pages and $2^{27}/2^{12} = 2^{15}$ physical pages. Thus, the virtual and physical page numbers are 19 and 15 bits, respectively. Physical memory can only hold up to 1/16th of the virtual pages at any given time. The rest of the virtual pages are kept on the hard drive.

Figure 8.21 shows virtual page 5 mapping to physical page 1, virtual page 0x7FFFC mapping to physical page 0x7FFE, and so forth. For example, virtual address 0x53F8 (an offset of 0x3F8 within virtual page 5) maps to physical address 0x13F8 (an offset of 0x3F8 within physical page 1). The least significant 12 bits of the virtual and physical addresses are the same (0x3F8) and specify the page offset within the virtual and physical pages. Only the page number needs to be translated to obtain the physical address from the virtual address.

Figure 8.22 illustrates the translation of a virtual address to a physical address. The least significant 12 bits indicate the page offset and require no translation. The upper 19 bits of the virtual address specify the *virtual page number* (*VPN*) and are translated to a 15-bit *physical page number* (*PPN*). The next two sections describe how page tables and TLBs are used to perform this address translation.

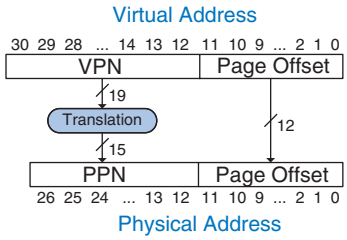


Figure 8.22 Translation from virtual address to physical address

Example 8.13 VIRTUAL ADDRESS TO PHYSICAL ADDRESS TRANSLATION

Find the physical address of virtual address 0x247C using the virtual memory system shown in Figure 8.21.

Solution The 12-bit page offset (0x47C) requires no translation. The remaining 19 bits of the virtual address give the virtual page number, so virtual address 0x247C is found in virtual page 0x2. In Figure 8.21, virtual page 0x2 maps to physical page 0x7FFF. Thus, virtual address 0x247C maps to physical address 0x7FFF47C.

8.4.2 The Page Table

The processor uses a *page table* to translate virtual addresses to physical addresses. The page table contains an entry for each virtual page. This entry contains a physical page number and a valid bit. If the valid bit

is 1, the virtual page maps to the physical page specified in the entry. Otherwise, the virtual page is found on the hard drive.

Because the page table is so large, it is stored in physical memory. Let us assume for now that it is stored as a contiguous array, as shown in Figure 8.23. This page table contains the mapping of the memory system of Figure 8.21. The page table is indexed with the virtual page number (VPN). For example, entry 5 specifies that virtual page 5 maps to physical page 1. Entry 6 is invalid ($V = 0$), so virtual page 6 is located on the hard drive.

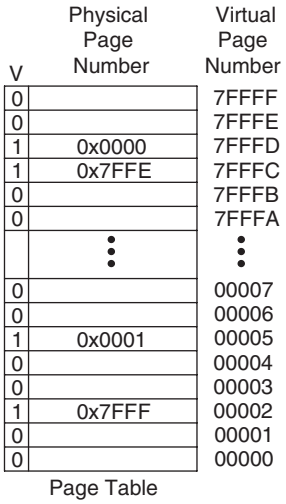


Figure 8.23 The page table for Figure 8.21

Example 8.14 USING THE PAGE TABLE TO PERFORM ADDRESS TRANSLATION

Find the physical address of virtual address 0x247C using the page table shown in Figure 8.23.

Solution Figure 8.24 shows the virtual address to physical address translation for virtual address 0x247C. The 12-bit page offset requires no translation. The remaining 19 bits of the virtual address are the virtual page number, 0x2, and give the index into the page table. The page table maps virtual page 0x2 to physical page 0x7FFF. So, virtual address 0x247C maps to physical address 0x7FFF47C. The least significant 12 bits are the same in both the physical and the virtual address.

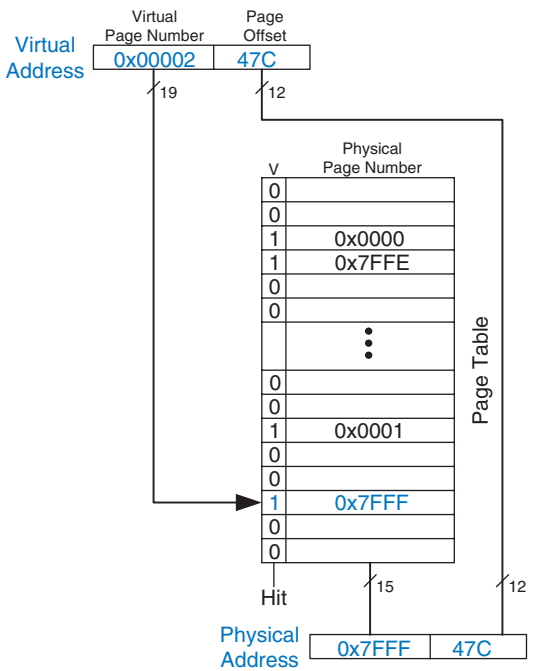


Figure 8.24 Address translation using the page table

The page table can be stored anywhere in physical memory at the discretion of the OS. The processor typically uses a dedicated register, called the *page table register*, to store the base address of the page table in physical memory.

To perform a load or store, the processor must first translate the virtual address to a physical address and then access the data at that physical address. The processor extracts the virtual page number from the virtual address and adds it to the page table register to find the physical address of the page table entry. The processor then reads this page table entry from physical memory to obtain the physical page number. If the entry is valid, it merges this physical page number with the page offset to create the physical address. Finally, it reads or writes data at this physical address. Because the page table is stored in physical memory, each load or store involves two physical memory accesses.

8.4.3 The Translation Lookaside Buffer

Virtual memory would have a severe performance impact if it required a page table read on every load or store, doubling the delay of loads and stores. Fortunately, page table accesses have great temporal locality. The temporal and spatial locality of data accesses and the large page size mean that many consecutive loads or stores are likely to reference the same page. Therefore, if the processor remembers the last page table entry that it read, it can probably reuse this translation without rereading the page table. In general, the processor can keep the last several page table entries in a small cache called a *translation lookaside buffer* (*TLB*). The processor "looks aside" to find the translation in the TLB before having to access the page table in physical memory. In real programs, the vast majority of accesses hit in the TLB, avoiding the time-consuming page table reads from physical memory.

A TLB is organized as a fully associative cache and typically holds 16 to 512 entries. Each TLB entry holds a virtual page number and its corresponding physical page number. The TLB is accessed using the virtual page number. If the TLB hits, it returns the corresponding physical page number. Otherwise, the processor must read the page table in physical memory. The TLB is designed to be small enough that it can be accessed in less than one cycle. Even so, TLBs typically have a hit rate of greater than 99%. The TLB decreases the number of memory accesses required for most load or store instructions from two to one.

Example 8.15 USING THE TLB TO PERFORM ADDRESS TRANSLATION

Consider the virtual memory system of Figure 8.21. Use a two-entry TLB or explain why a page table access is necessary to translate virtual addresses 0x247C and 0x5FB0 to physical addresses. Suppose that the TLB currently holds valid translations of virtual pages 0x2 and 0x7FFFD.

Solution Figure 8.25 shows the two-entry TLB with the request for virtual address 0x247C. The TLB receives the virtual page number of the incoming address, 0x2, and compares it to the virtual page number of each entry. Entry 0 matches and is valid, so the request hits. The translated physical address is the physical page number of the matching entry, 0x7FFF, concatenated with the page offset of the virtual address. As always, the page offset requires no translation.

The request for virtual address 0x5FB0 misses in the TLB. So, the request is forwarded to the page table for translation.

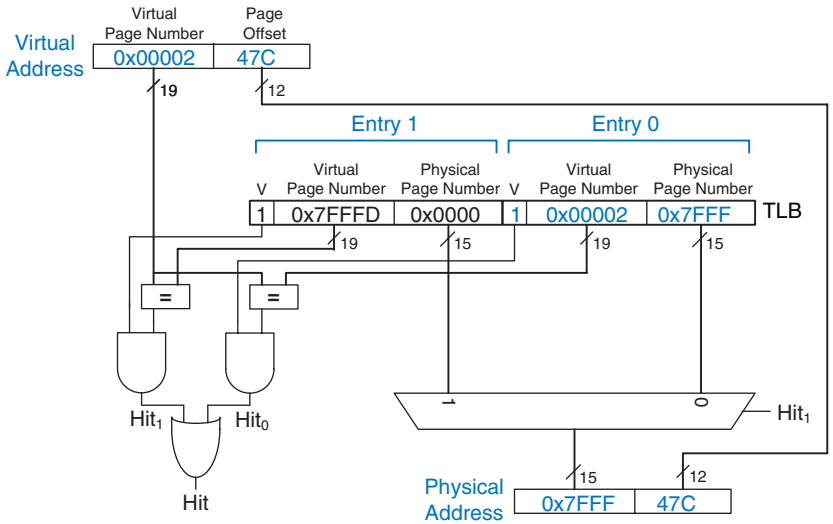


Figure 8.25 Address translation using a two-entry TLB

8.4.4 Memory Protection

So far, this section has focused on using virtual memory to provide a fast, inexpensive, large memory. An equally important reason to use virtual memory is to provide protection between concurrently running programs.

As you probably know, modern computers typically run several programs or *processes* at the same time. All of the programs are simultaneously present in physical memory. In a well-designed computer system, the programs should be protected from each other so that no program can crash or hijack another program. Specifically, no program should be able to access another program's memory without permission. This is called *memory protection*.

Virtual memory systems provide memory protection by giving each program its own *virtual address space*. Each program can use as much memory as it wants in that virtual address space, but only a portion of

the virtual address space is in physical memory at any given time. Each program can use its entire virtual address space without having to worry about where other programs are physically located. However, a program can access only those physical pages that are mapped in its page table. In this way, a program cannot accidentally or maliciously access another program's physical pages because they are not mapped in its page table. In some cases, multiple programs access common instructions or data. The OS adds control bits to each page table entry to determine which programs, if any, can write to the shared physical pages.

8.4.5 Replacement Policies*

Virtual memory systems use write-back and an approximate least recently used (LRU) replacement policy. A write-through policy, where each write to physical memory initiates a write to the hard drive, would be impractical. Store instructions would operate at the speed of the hard drive instead of the speed of the processor (milliseconds instead of nanoseconds). Under the writeback policy, the physical page is written back to the hard drive only when it is evicted from physical memory. Writing the physical page back to the hard drive and reloading it with a different virtual page is called *paging*, and the hard drive in a virtual memory system is sometimes called *swap space*. The processor pages out one of the least recently used physical pages when a page fault occurs, then replaces that page with the missing virtual page. To support these replacement policies, each page table entry contains two additional status bits: a dirty bit D and a use bit U.

The dirty bit is 1 if any store instructions have changed the physical page since it was read from the hard drive. When a physical page is paged out, it needs to be written back to the hard drive only if its dirty bit is 1; otherwise, the hard drive already holds an exact copy of the page.

The use bit is 1 if the physical page has been accessed recently. As in a cache system, exact LRU replacement would be impractically complicated. Instead, the OS approximates LRU replacement by periodically resetting all of the use bits in the page table. When a page is accessed, its use bit is set to 1. Upon a page fault, the OS finds a page with $U = 0$ to page out of physical memory. Thus, it does not necessarily replace the least recently used page, just one of the least recently used pages.

8.4.6 Multilevel Page Tables*

Page tables can occupy a large amount of physical memory. For example, the page table from the previous sections for a 2 GiB virtual memory with 4 KiB pages would need 2^{19} entries. If each entry is 4 bytes, the page table is $2^{19} \times 2^2$ bytes $= 2^{21}$ bytes $= 2$ MiB.

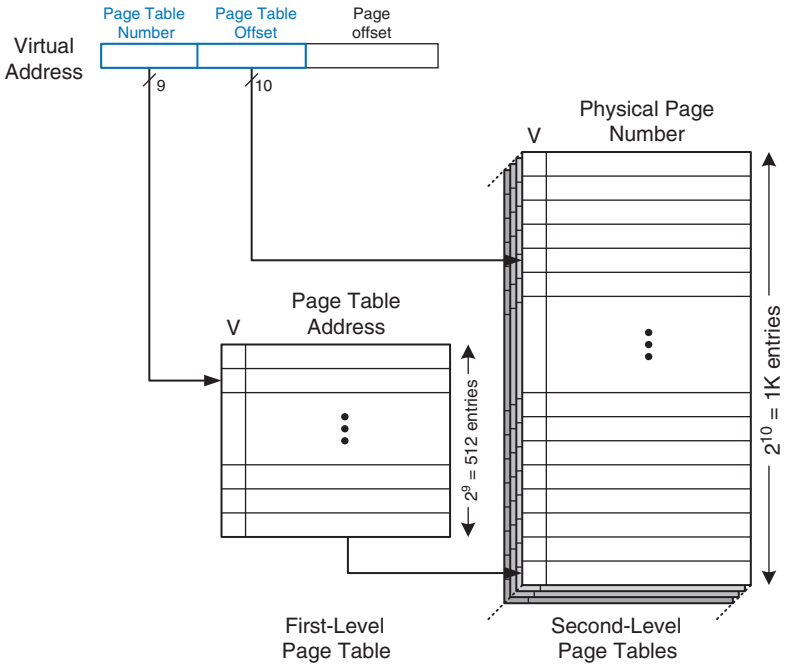


Figure 8.26 **Hierarchical page tables**

To conserve physical memory, page tables can be broken up into multiple (usually two) levels. The first-level page table is always kept in physical memory. It indicates where small second-level page tables are stored in virtual memory. The second-level page tables each contain the actual translations for a range of virtual pages. If a particular range of translations is not actively used, the corresponding second-level page table can be paged out to the hard drive so it does not waste physical memory.

In a two-level page table, the virtual page number is split into two parts: the *page table number* and the *page table offset*, as shown in Figure 8.26. The page table number indexes the first-level page table, which must reside in physical memory. The first-level page table entry gives the base address of the second-level page table or indicates that it must be fetched from the hard drive when V is 0. The page table offset indexes the second-level page table. The remaining 12 bits of the virtual address are the page offset, as before, for a page size of $2^{12} = 4$ KiB.

In Figure 8.26, the 19-bit virtual page number is broken into 9 and 10 bits to indicate the page table number and the page table offset, respectively. Thus, the first-level page table has $2^9 = 512$ entries. Each of these 512 second-level page tables has $2^{10} = 1$ Ki entries. If each of the first- and second-level page table entries is 32 bits (4 bytes) and only two second-level page tables are present in physical memory at once, the

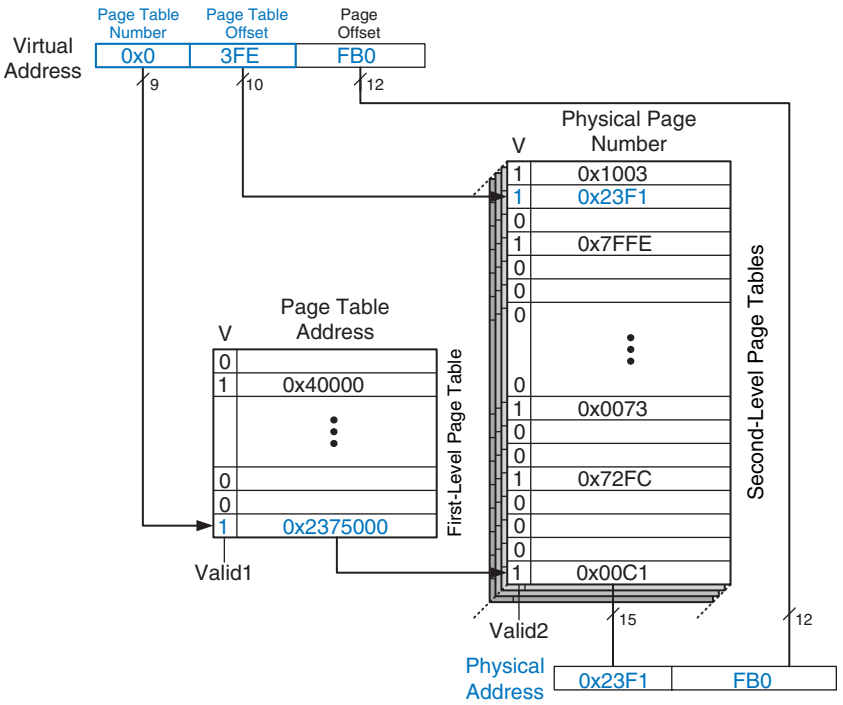


Figure 8.27 Address translation using a two-level page table

hierarchical page table uses only (512 × 4 bytes) + 2 × (1 Ki × 4 bytes) = 10 KiB of physical memory. The two-level page table requires a fraction of the physical memory needed to store the entire page table (2 MiB). The drawback of a two-level page table is that it adds yet another memory access for translation when the TLB misses.

Example 8.16 USING A MULTILEVEL PAGE TABLE FOR ADDRESS TRANSLATION

Figure 8.27 shows the possible contents of the two-level page table from Figure 8.26. The contents of only one second-level page table are shown. Using this two-level page table, describe what happens on an access to virtual address 0x003FEFB0.

Solution As always, only the virtual page number requires translation. The most significant nine bits of the virtual address, 0x0, give the page table number, the index into the first-level page table. The first-level page table at entry 0x0 indicates that the second-level page table is resident in memory ($V = 1$) and its physical address is 0x2375000.

The next ten bits of the virtual address, 0x3FE, are the page table offset, which gives the index into the second-level page table. Entry 0 is at the bottom of the second-level page table, and entry 0x3FF is at the top. Entry 0x3FE in the second-level page table indicates that the virtual page is resident in physical memory ($V = 1$) and that the physical page number is 0x23F1. The physical page number is concatenated with the page offset to form the physical address, 0x23F1FB0.

8.5 SUMMARY

Memory system organization is a major factor in determining computer performance. Different memory technologies—such as DRAM, SRAM, and hard drives—offer trade-offs in capacity, speed, and cost. This chapter introduced cache and virtual memory organizations that use a hierarchy of memories to approximate an ideal large, fast, inexpensive memory. Main memory is typically built from DRAM, which is significantly slower than the processor. A cache reduces access time by keeping commonly used data in fast SRAM. Virtual memory increases the memory capacity by using a hard drive to store data that does not fit in the main memory. Caches and virtual memory add complexity and hardware to a computer system, but the benefits usually outweigh the costs. All modern personal computers use caches and virtual memory. Most processors also use the memory interface to communicate with input/output (I/O) devices. This is called *memory-mapped I/O*. Programs use load and store operations to access I/O devices, which are discussed in Chapter 9, an online supplemental chapter available on this book's companion website (see the Preface).

EPILOGUE

This chapter brings us to the end of our journey together into the realm of digital systems. We hope this book has conveyed the beauty and thrill of the art as well as the engineering knowledge. You have learned to design combinational and sequential logic using schematics and hardware description languages. You are familiar with larger building blocks such as multiplexers, ALUs, and memories. Computers are one of the most fascinating applications of digital systems. You have learned how to program a RISC-V processor in its native assembly language and how to build the processor and memory system using digital building blocks. Throughout, you have seen the application of abstraction, discipline, hierarchy, modularity, and regularity. With these techniques, we have pieced together the puzzle of a microprocessor's inner workings. From cell phones to digital television to Mars rovers to medical imaging systems, our world is an increasingly digital place.

Imagine what Faustian bargain Charles Babbage would have made to take a similar journey a century and a half ago. He merely aspired to calculate mathematical tables with mechanical precision. Today's digital systems are yesterday's science fiction. Might Dick Tracy have listened to iTunes on his cell phone? Would Jules Verne have launched a constellation of global positioning satellites into space? Could Hippocrates have cured illness using high-resolution digital images of the brain? But, at the same time, George Orwell's nightmare of ubiquitous government surveillance becomes closer to reality each day. Hackers and governments wage undeclared cyberwarfare, attacking industrial infrastructure and financial networks. And rogue states develop nuclear weapons using laptop computers more powerful than the room-sized supercomputers that simulated Cold War bombs. The microprocessor revolution continues to accelerate. The changes in the coming decades will surpass those of the past. You now have the tools to design and build these new systems that will shape our future. With your newfound power comes profound responsibility. We hope that you will use it, not just for fun and riches, but also for the benefit of humanity.

Exercises

Exercise 8.1 In less than one page, describe four everyday activities that exhibit temporal or spatial locality. List two activities for each type of locality and be specific.

Exercise 8.2 In one paragraph, describe two short computer applications that exhibit temporal and/or spatial locality. Describe how. Be specific.

Exercise 8.3 Come up with a sequence of addresses for which a direct mapped cache with a size (capacity) of 16 words and block size of 4 words outperforms a fully associative cache with least recently used (LRU) replacement that has the same capacity and block size.

Exercise 8.4 Repeat Exercise 8.3 for the case when the fully associative cache outperforms the direct mapped cache.

Exercise 8.5 Describe the trade-offs of increasing each of the following cache parameters while keeping the others the same:

(a) block size

(b) associativity

(c) cache size

Exercise 8.6 Is the miss rate of a two-way set associative cache always, usually, occasionally, or never better than that of a direct mapped cache of the same capacity and block size? Explain.

Exercise 8.7 Each of the following statements pertains to the miss rate of caches. Mark each statement as true or false. Briefly explain your reasoning; present a counterexample if the statement is false.

(a) A two-way set associative cache always has a lower miss rate than a direct mapped cache with the same block size and total capacity.

(b) A 16 KiB direct mapped cache always has a lower miss rate than an 8 KiB direct mapped cache with the same block size.

(c) An instruction cache with a 32-byte block size usually has a lower miss rate than an instruction cache with an 8-byte block size, given the same degree of associativity and total capacity.

Exercise 8.8 A cache has the following parameters: b, block size given in numbers of words; S, number of sets; N, number of ways; and A, number of address bits.

(a) In terms of the parameters described, what is the cache capacity, C?

(b) In terms of the parameters described, what is the total number of bits required to store the tags?

(c) What are S and N for a fully associative cache of capacity C words with block size b?

(d) What is S for a direct mapped cache of size C words and block size b?

Exercise 8.9 A 16-word cache has the parameters given in Exercise 8.8. Consider the following repeating sequence of `lw` addresses (given in hexadecimal):

40 44 48 4C 70 74 78 7C 80 84 88 8C 90 94 98 9C 0 4 8 C 10 14 18 1C 20

Assuming least recently used (LRU) replacement for associative caches, determine the effective miss rate if the sequence is input to the following caches, ignoring start-up effects (i.e., compulsory misses).

(a) direct mapped cache, $b = 1$ word

(b) fully associative cache, $b = 1$ word

(c) two-way set associative cache, $b = 1$ word

(d) direct mapped cache, $b = 2$ words

Exercise 8.10 Repeat Exercise 8.9 for the following repeating sequence of `lw` addresses (given in hexadecimal) and cache configurations. The cache capacity is still 16 words.

74 A0 78 38C AC 84 88 8C 7C 34 38 13C 388 18C

(a) direct mapped cache, $b = 1$ word

(b) fully associative cache, $b = 2$ words

(c) two-way set associative cache, $b = 2$ words

(d) direct mapped cache, $b = 4$ words

Exercise 8.11 Suppose you are running a program with the following data access pattern (given in hexadecimal). The pattern is executed only once.

0 8 10 18 20 28

(a) If you use a direct mapped cache with a cache size of 1 KiB and a block size of 8 bytes (2 words), how many sets are in the cache?

(b) With the same cache and block size as in part (a), what is the miss rate of the direct mapped cache for the given memory access pattern?

(c) For the given memory access pattern, which of the following would decrease the miss rate the most? (Cache capacity is kept constant.) Circle one.

(i) Increasing the degree of associativity to 2.

(ii) Increasing the block size to 16 bytes.

(iii) Either (i) or (ii).

(iv) Neither (i) nor (ii).

Exercise 8.12 You are building an instruction cache for a RISC-V processor. It has a total capacity of $4C = 2^{c+2}$ bytes. It is $N = 2^n$-way set associative ($N \geq 8$), with a block size of $b = 2^{b'}$ bytes ($b \geq 8$). Give your answers to the following questions in terms of these parameters.

(a) Which bits of the address are used to select a word within a block?

(b) Which bits of the address are used to select the set within the cache?

(c) How many bits are in each tag?

(d) How many tag bits are in the entire cache?

Exercise 8.13 Consider a cache with the following parameters: N (associativity) = 2, b (block size) = 2 words, W (word size) = 32 bits, C (cache size) = 32 Ki words, A (address size) = 32 bits. You need consider only word addresses.

(a) Show the tag, set, block offset, and byte offset bits of the address. State how many bits are needed for each field.

(b) What is the size of *all* the cache tags in bits?

(c) Suppose that each cache block also has a valid bit (V) and a dirty bit (D). What is the size of each cache set, including data, tag, and status bits?

(d) Design the cache using the building blocks in Figure 8.28 and a small number of two-input logic gates. The cache design must include tag storage, data

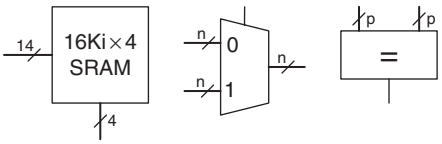


Figure 8.28 Building blocks

storage, address comparison, data output selection, and any other parts you feel are relevant. Note that the multiplexer and comparator blocks may be any size (n or p bits wide, respectively), but the SRAM blocks must be 16Ki × 4 bits. Be sure to include a neatly labeled block diagram. You need only design the cache for reads.

Exercise 8.14 You've joined a hot new Internet start-up to build wristwatches with a built-in pager and Web browser. It uses an embedded processor with a multilevel cache scheme depicted in Figure 8.29. The processor includes a small on-chip cache in addition to a large off-chip second-level cache. (Yes, the watch weighs 3 pounds, but you should see it surf!)

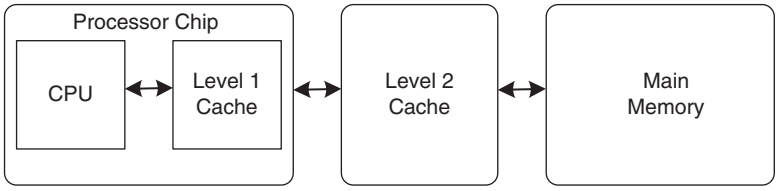


Figure 8.29 Computer system

Assume that the processor uses 32-bit physical addresses but accesses data only on word boundaries. The caches have the characteristics given in Table 8.4. The DRAM has an access time of t_m and a size of 512 MiB.

Table 8.4 Memory characteristics

Characteristic	On-chip Cache	Off-chip Cache
Organization	Four-way set associative	Direct mapped
Hit rate	A	B
Access time	t_a	t_b
Block size	16 bytes	16 bytes
Number of blocks	512	256 Ki

(a) For a given word in memory, what is the total number of locations in which it might be found in the on-chip cache and in the second-level cache?

(b) What is the size, in bits, of each tag for the on-chip cache and the second-level cache?

(c) Give an expression for the average memory read access time. The caches are accessed in sequence.

(d) Measurements show that, for a particular problem of interest, the on-chip cache hit rate is 85% and the second-level cache hit rate is 90%. However, when the on-chip cache is disabled, the second-level cache hit rate shoots up to 98.5%. Give a brief explanation of this behavior.

Exercise 8.15 This chapter described the least recently used (LRU) replacement policy for multiway associative caches. Other less common replacement policies include first-in-first-out (FIFO) and random policies. FIFO replacement evicts the block that has been there the longest, regardless of how recently it was accessed. Random replacement randomly picks a block to evict.

(a) Discuss the advantages and disadvantages of each of these replacement policies.

(b) Describe a data access pattern for which FIFO would perform better than LRU.

Exercise 8.16 You are building a computer with a hierarchical memory system that consists of separate instruction and data caches followed by main memory. You are using the RISC-V multicycle processor from Figure 7.44 running at 1 GHz.

(a) Suppose the instruction cache is perfect (i.e., always hits) but the data cache has a 5% miss rate. On a cache miss, the processor stalls for 60 ns to access main memory, then resumes normal operation. Taking cache misses into account, what is the average memory access time?

(b) How many clock cycles per instruction (CPI) on average are required for load and store word instructions considering the nonideal memory system?

(c) Consider the benchmark application of Example 7.7 that has 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions[3]. Taking the non-ideal memory system into account, what is the average CPI for this benchmark?

(d) Now suppose that the instruction cache is also non-ideal and has a 7% miss rate. What is the average CPI for the benchmark in part (c)? Take into account both instruction and data cache misses.

[3] Data from Patterson and Hennessy, *Computer Organization and Design*, 4th Edition, Morgan Kaufmann, 2011. Used with permission.

Exercise 8.17 Repeat Exercise 8.16 with the following parameters.

(a) The instruction cache is perfect (i.e., always hits) but the data cache has a 15% miss rate. On a cache miss, the processor stalls for 200 ns to access main memory, then resumes normal operation. Taking cache misses into account, what is the average memory access time?

(b) How many clock cycles per instruction (CPI) on average are required for load and store word instructions considering the non-ideal memory system?

(c) Consider the benchmark application of Example 7.7 that has 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions. Taking the nonideal memory system into account, what is the average CPI for this benchmark?

(d) Now, suppose that the instruction cache is also nonideal and has a 10% miss rate. What is the average CPI for the benchmark in part (c)? Take into account both instruction and data cache misses.

Exercise 8.18 If a computer uses 64-bit virtual addresses, how much virtual memory can it access? Note that 2^{40} bytes = 1 *terabyte (tebibyte)*, 2^{50} bytes = 1 *petabyte (pebibyte)*, and 2^{60} bytes = 1 *exabyte (exbibyte)*.

Exercise 8.19 A supercomputer designer chooses to spend $1 million on DRAM and the same amount on hard disks for virtual memory. Using the prices from Figure 8.4, how much physical and virtual memory will the computer have? How many bits of physical and virtual addresses are necessary to access this memory?

Exercise 8.20 Consider a virtual memory system that can address a total of 2^{32} bytes. You have unlimited hard drive space but are limited to only 8 MiB of semiconductor (physical) memory. Assume that virtual and physical pages are each 4 KiB in size.

(a) How many bits is the physical address?

(b) What is the maximum number of virtual pages in the system?

(c) How many physical pages are in the system?

(d) How many bits are the virtual and physical page numbers?

(e) Suppose that you come up with a direct mapped scheme that maps virtual pages to physical pages. The mapping uses the least significant bits of the virtual page number to determine the physical page number. How many virtual pages are mapped to each physical page? Why is this "direct mapping" a bad plan?

(f) Clearly, a more flexible and dynamic scheme for translating virtual addresses into physical addresses is required than the one described in part (e). Suppose that you use a page table to store mappings (translations from virtual page number to physical page number). How many page table entries will the page table contain?

(g) Assume that, in addition to the physical page number, each page table entry also contains some status information in the form of a valid bit (*V*) and a dirty bit (*D*). How many bytes long is each page table entry? (Round up to an integer number of bytes.)

(h) Sketch the layout of the page table. What is the total size of the page table in bytes?

Exercise 8.21 Consider a virtual memory system that can address a total of 2^{50} bytes. You have unlimited hard drive space but are limited to 2 GiB of semiconductor (physical) memory. Assume that virtual and physical pages are each 4 KiB in size.

(a) How many bits is the physical address?

(b) What is the maximum number of virtual pages in the system?

(c) How many physical pages are in the system?

(d) How many bits are the virtual and physical page numbers?

(e) How many page table entries will the page table contain?

(f) Assume that, in addition to the physical page number, each page table entry also contains some status information in the form of a valid bit (*V*) and a dirty bit (*D*). How many bytes long is each page table entry? (Round up to an integer number of bytes.)

(g) Sketch the layout of the page table. What is the total size of the page table in bytes?

Exercise 8.22 You decide to speed up the virtual memory system of Exercise 8.20 by using a translation lookaside buffer (TLB). Suppose that your memory system has the characteristics shown in Table 8.5. The TLB and cache miss rates indicate how

Table 8.5 Memory characteristics

Memory Unit	Access Time (Cycles)	Miss Rate
TLB	1	0.05%
Cache	1	2%
Main memory	100	0.0003%
Hard drive	1,000,000	0%

often the requested entry is not found. The main memory miss rate indicates how often page faults occur.

(a) What is the average memory access time of the virtual memory system before and after adding the TLB? Assume that the page table is always resident in physical memory and is never held in the data cache.

(b) If the TLB has 64 entries, how big (in bits) is the TLB? Give numbers for data (physical page number), tag (virtual page number), and valid bits of each entry. Show your work clearly.

(c) Sketch the TLB. Clearly label all fields and dimensions.

(d) What size SRAM would you need to build the TLB described in part (c)? Give your answer in terms of depth × width.

Exercise 8.23 You decide to speed up the virtual memory system of Exercise 8.21 by using a translation lookaside buffer (TLB) with 128 entries.

(a) How big (in bits) is the TLB? Give numbers for data (physical page number), tag (virtual page number), and valid bits of each entry. Show your work clearly.

(b) Sketch the TLB. Clearly label all fields and dimensions.

(c) What size SRAM would you need to build the TLB described in part (b)? Give your answer in terms of depth × width.

Exercise 8.24 Suppose that the RISC-V multicycle processor described in Section 7.4 uses a virtual memory system.

(a) Sketch the location of the TLB in the multicycle processor schematic.

(b) Describe how adding a TLB affects processor performance.

Exercise 8.25 The virtual memory system you are designing uses a single-level page table built from dedicated hardware (SRAM and associated logic). It supports 25-bit virtual addresses, 22-bit physical addresses, and 2^{16}-byte (64 KiB) pages. Each page table entry contains a physical page number, a valid bit (V), and a dirty bit (D).

(a) What is the total size of the page table, in bits?

(b) The operating system team proposes reducing the page size from 64 to 16 KiB, but the hardware engineers on your team object on the grounds of added hardware cost. Explain their objection.

(c) The page table is to be integrated on the processor chip, along with the on-chip cache. The on-chip cache deals only with physical (not virtual) addresses. Is it possible to access the appropriate set of the on-chip cache concurrently with the page table access for a given memory access? Explain briefly the relationship that is necessary for concurrent access to the cache set and page table entry.

(d) Is it possible to perform the tag comparison in the on-chip cache concurrently with the page table access for a given memory access? Explain briefly.

Exercise 8.26 Describe a scenario in which the virtual memory system might affect how an application is written. Be sure to include a discussion of how the page size and physical memory size affect the performance of the application.

Exercise 8.27 Suppose that you own a personal computer (PC) that uses 32-bit virtual addresses.

(a) What is the maximum amount of virtual memory space each program can use?

(b) How does the size of your PC's hard drive affect performance?

(c) How does the size of your PC's physical memory affect performance?

Interview Questions

The following exercises present questions that have been asked on interviews.

Question 8.1 Explain the difference between direct mapped, set associative, and fully associative caches. For each cache type, describe an application for which that cache type will perform better than the other two.

Question 8.2 Explain how virtual memory systems work.

Question 8.3 Explain the advantages and disadvantages of using a virtual memory system.

Question 8.4 Explain how cache performance might be affected by the virtual page size of a memory system.