RAM /abr./: Rarely Adequate Memory, because the more memory a computer has, the faster it can produce error messages.

—Anonymous

640K [of memory] ought to be enough for anybody.

—Anonymous

CHAPTER 6

Memory

6.1 INTRODUCTION

Most computers are built using the Von Neumann model, which is centered on memory. The programs that perform the processing are stored in memory. We examined a small 4 × 3-bit memory in Chapter 3, and we learned how to address memory in Chapters 4 and 5. We know memory is logically structured as a linear array of locations, with addresses from 0 to the maximum memory size the processor can address. In this chapter, we examine the various types of memory and how each is part of the memory hierarchy system. We then look at cache memory (a special high-speed memory) and a method that utilizes memory to its fullest by means of virtual memory implemented via paging.

6.2 TYPES OF MEMORY

A common question many people ask is, "Why are there so many different types of computer memory?" The answer is that new technologies continue to be introduced in an attempt to match the improvements in CPU design—the speed of memory has to, somewhat, keep pace with the CPU, or the memory becomes a bottleneck. Although we have seen many improvements in CPUs over the past few years, improving main memory to keep pace with the CPU is actually not as critical because of the use of **cache memory**. Cache memory is a small, high-speed (and thus high-cost) type of memory that serves as a buffer for frequently accessed data. The additional expense of using very fast technologies for memory cannot always be justified because slower memories can often be "hidden" by high-performance cache systems. However, before we discuss cache memory, we will explain the various memory technologies.

Even though a large number of memory technologies exist, there are only two basic types of memory: **RAM** (**random access memory**) and **ROM** (**read-only memory**). RAM is somewhat of a misnomer; a more appropriate name is read-write memory. RAM is the memory to which computer specifications refer; if you buy a computer with 128 megabytes of memory, it has 128MB of RAM. RAM is also the "main memory" we have continually referred to throughout this text. Often called primary memory, RAM is used to store programs and data that the computer needs when executing programs, but RAM is volatile and loses this information once the power is turned off. There are two general types of chips used to build the bulk of RAM in today's computers: SRAM and DRAM (static and dynamic random access memory).

Dynamic RAM is constructed of tiny capacitors that leak electricity. DRAM requires a recharge every few milliseconds to maintain its data. **Static RAM** technology, in contrast, holds its contents as long as power is available. SRAM consists of circuits similar to the D flip-flops we studied in Chapter 3. SRAM is faster and much more expensive than DRAM; however, designers use DRAM because it is much denser (can store many bits per chip), uses less power, and generates less heat than SRAM. For these reasons, both technologies are often used in combination: DRAM for main memory and SRAM for cache. The basic operation of all DRAM memories is the same, but there are many flavors, including **Multibank DRAM** (**MDRAM**), **Fast-Page Mode** (**FPM**) **DRAM**, **Extended Data Out** (**EDO**) **DRAM**, **Burst EDO DRAM** (**BEDO DRAM**), **Synchronous Dynamic Random Access Memory** (**SDRAM**), **Synchronous-Link** (**SL**) **DRAM**, **Double Data Rate** (**DDR**) **SDRAM**, **Rambus DRAM** (**RDRAM**), and

Direct Rambus (**DR**) **DRAM**. The different types of SRAM include asynchronous SRAM, synchronous SRAM, and pipeline burst SRAM. For more information about these types of memory, refer to the references listed at the end of the chapter.

In addition to RAM, most computers contain a small amount of ROM that stores critical information necessary to operate the system, such as the program necessary to boot the computer. ROM is not volatile and always retains its data. This type of memory is also used in embedded systems or any systems where the programming does not need to change. Many appliances, toys, and most automobiles use ROM chips to maintain information when the power is shut off. ROMs are also used extensively in calculators and peripheral devices such as laser printers, which store their fonts in ROMs. There are five basic types of ROM: ROM, PROM, EPROM, EEPROM, and flash memory. **PROM** (**programmable read-only memory**) is a variation on ROM. PROMs can be programmed by the user with the appropriate equipment. Whereas ROMs are hardwired, PROMs have fuses that can be blown to program the chip. Once programmed, the data and instructions in PROM cannot be changed. **EPROM** (**erasable PROM**) is programmable with the added advantage of being reprogrammable (erasing an EPROM requires a special tool that emits ultraviolet light). To reprogram an EPROM, the entire chip must first be erased. **EEPROM** (**electrically erasable PROM**) removes many of the disadvantages of EPROM: No special tools are required for erasure (this is performed by applying an electric field) and you can erase only portions of the chip, one byte at a time. **Flash memory** is essentially EEPROM with the added benefit that data can be written or erased in blocks, removing the one-byte-at-a-time limitation. This makes flash memory faster than EEPROM. Flash memory has become a very popular type of storage and is used in many different devices, including cell phones, digital cameras, and music players. It is also being used in solid-state disk drives. (Chapter 7 contains more information on flash memory devices.)

6.3 THE MEMORY HIERARCHY

One of the most important considerations in understanding the performance capabilities of a modern processor is the memory hierarchy. Unfortunately, as we have seen, not all memory is created equal, and some types are far less efficient and thus cheaper than others. To deal with this disparity, today's computer systems use a combination of memory types to provide the best performance at the best cost. This approach is called **hierarchical memory**. As a rule, the faster memory is, the more expensive it is per bit of storage. By using a hierarchy of memories, each with different access speeds and storage capacities, a computer system can exhibit performance above what would be possible without a combination of the various types. The base types that normally constitute the hierarchical memory system include registers, cache, main memory, secondary memory, and off-line bulk memory.

Registers are storage locations available on the processor itself. **Cache memory** is a very-high-speed memory where data from frequently used memory locations may be temporarily stored. This cache is connected to a much larger **main memory**, which is typically a medium-speed memory. This memory is complemented by a very large **secondary memory**, typically composed of hard disk drives containing data not directly accessible by the CPU; instead, secondary memory must have its contents transferred to main memory when the data is needed. Hard drives can either be magnetic or **solid state** (flash-based hard drives that are faster and sturdier than rotating magnetic disks). **Off-line bulk memory (**which includes **tertiary memory** and **off-line storage)** requires either human or robotic intervention before any data can be accessed; the data must be transferred from the storage media to secondary memory. Tertiary memory includes things such as optical jukeboxes and tape libraries, which are typically under robotic control (a robotic arm mounts and dismounts the tapes and disks). It is used for enterprise storage in large systems and networks and is not something an average computer user sees often. These devices typically have nonuniform access times, as the time to retrieve data depends on whether the device is mounted. Off-line storage includes those devices that are connected, loaded with data, and then disconnected from the system, such as floppy disks, flash memory devices, optical disks, and even removable hard drives. By using such a hierarchical scheme, one can improve the effective access speed of the memory, using only a small number of fast (and expensive) chips. This allows designers to create a computer with acceptable performance at a reasonable cost.

We classify memory based on its "distance" from the processor, with distance measured by the number of machine cycles required for access. The closer memory is to the processor, the faster it should be. As memory gets farther from the main processor, we can afford longer access times. Thus, slower technologies are used for these memories, and faster technologies are used for memories closer to the CPU. The better the technology, the faster and more expensive the memory becomes. Thus, faster memories tend to be smaller than slower ones, because of cost.

The following terminology is used when referring to this memory hierarchy:

- **Hit**—The requested data resides in a given level of memory (typically, we are concerned with the hit rate only for upper levels of memory).
- **Miss**—The requested data is not found in the given level of memory.
- **Hit rate**—The percentage of memory accesses found in a given level of memory.
- **Miss rate**—The percentage of memory accesses not found in a given level of memory. Note: Miss Rate = 1 – Hit Rate.
- **Hit time**—The time required to access the requested information in a given level of memory.
- **Miss penalty**—The time required to process a miss, which includes replacing a block in an upper level of memory, plus the additional time to deliver the requested data to the processor. (The time to process a miss is typically significantly larger than the time to process a hit.)

The memory hierarchy is illustrated in Figure 6.1. This is drawn as a pyramid to help indicate the relative sizes of these various memories. Memories closer to the top tend to be smaller in size. However, these smaller memories have better performance and thus a higher cost (per bit) than memories found lower in the pyramid. The numbers given to the left of the pyramid indicate typical access times, which generally increase as we progress from top to bottom. Register accesses typically require one clock cycle.

There is one exception to access times increasing as we move down the hierarchy, in part because of new technologies. The off-line memory devices typically have faster access times than most tertiary devices. Of particular interest are USB flash drives. Both solid-state drives and USB flash drives use the same technology; thus both exhibit similar access times regarding accessing the data on the device. However, USB flash drives have slower access times than solid-state hard drives because of the USB interface. Even so, USB flash drives are faster than the other nonsolid-state types of off-line storage. Removable magnetic hard drives have access times of 12–40ms, and USB flash drives have access times of 0.5–33.8ms. Based only on access times, these latter devices do not belong at the bottom of our pyramid.

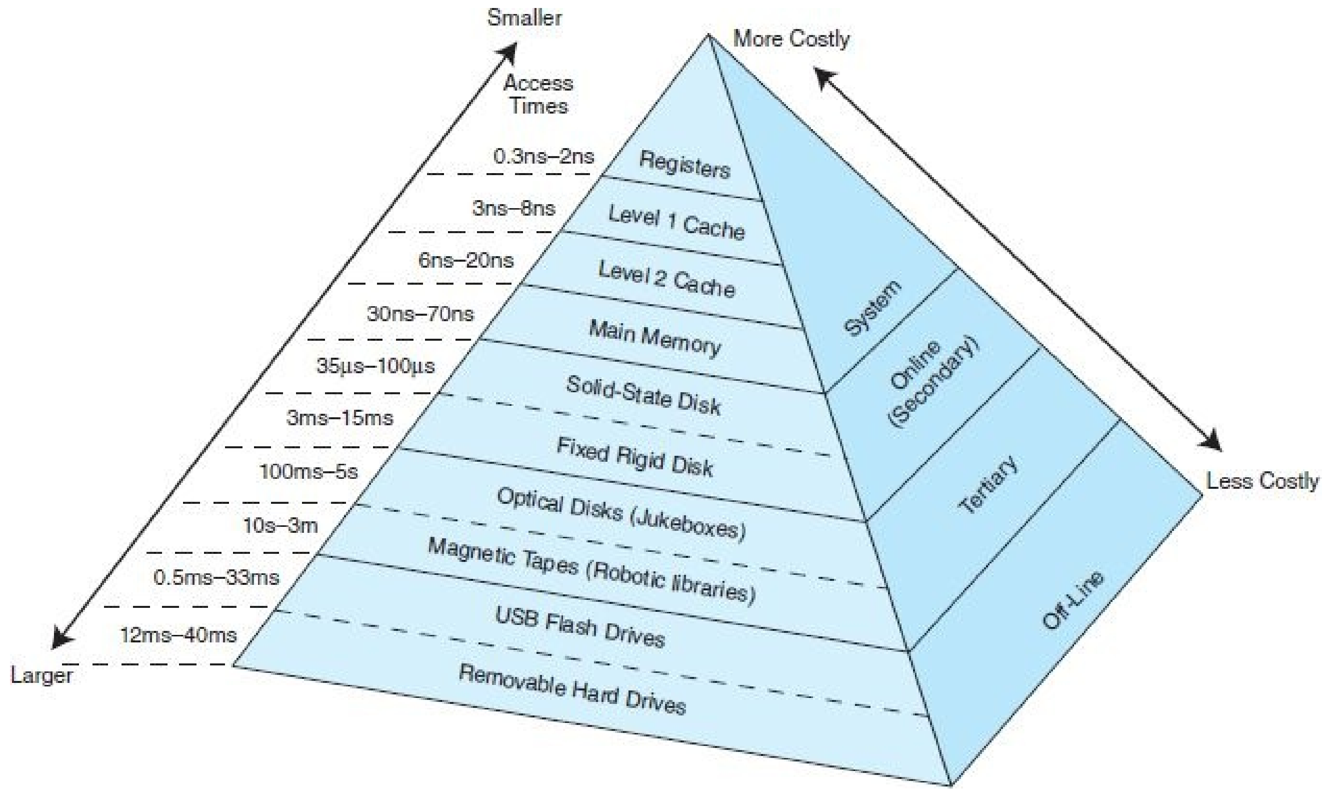


FIGURE 6.1 The Memory Hierarchy

Figure 6.1 represents not only a memory hierarchy, but also a storage hierarchy. We are most interested in the memory hierarchy that involves registers, cache, main memory, and **virtual memory** (nonsystem memory that acts as an extension to main memory—this is discussed in detail later in this chapter). Virtual memory is typically implemented using a hard drive; it gives the impression that a program may have a large, contiguous working main memory, when in fact the program may exist, in fragments, in main memory, and on disk. Virtual memory increases the available memory your computer can use by extending the address space from RAM to the hard drive. Ordinarily, the memory hierarchy would stop at the hard drive level. However, with the advent of solid-state technology, the definition of virtual memory is changing. We previously mentioned that USB flash drives have very fast access times. They are so fast, in fact, that some operating systems (including some versions of Unix and Windows XP and later) allow a user to utilize a USB flash drive as virtual memory. The Windows operating systems that are mentioned come with software called ReadyBoost that allows various types of removable solid-state devices, such as USB flash drives and SD cards, to augment virtual memory by acting as disk cache. These solidstate devices service requests up to 100 times faster than traditional hard drives; although they are not a replacement for hard drives, they are quickly becoming an interesting enhancement.

For any given data, the processor sends its request to the fastest, smallest partition of memory (typically cache, because registers tend to be more special purpose). If the data is found in cache, it can be loaded quickly into the CPU. If it is not resident in cache, the request is forwarded to the next lower level of the hierarchy, and this search process begins again. If the data is found at this level, the whole block in which the data resides is transferred into cache. If the data is not found at this level, the request is forwarded to the next lower level, and so on. The key idea is that when the lower (slower, larger, and cheaper) levels of the hierarchy respond to a request from higher levels for the content of location X, they also send, at the same time, the data located “around” X (…, $X - 2$, $X - 1$, X, $X + 1$, $X + 2$, …), thus returning an entire block of data to the higher-level memory. The hope is that this extra data will be referenced in the near future, which, in most cases, it is. The memory hierarchy is functional because programs tend to exhibit a property known as **locality**, in which the processor tends to access the data returned for addresses $X - 2$, $X - 1$, $X + 1$, $X + 2$, and so on. Thus, although there is one miss to, say, cache for *X*, there may be several hits in cache on the newly retrieved block afterward, because of locality.

6.3.1 Locality of Reference

In practice, processors tend to access memory in a patterned way. For example, in the absence of branches, the PC in MARIE is incremented by one after each instruction fetch. Thus, if memory location *X* is accessed at time *t*, there is a high probability that memory location $X + 1$ will also be accessed in the near future. This clustering of memory references into groups is an example of **locality of reference**. This locality can be exploited by implementing the memory as a hierarchy; when a miss is processed, instead of simply transferring the requested data to a higher level, the entire block containing the data is transferred. Because of locality of reference, it is likely that the additional data in the block will be needed in the near future, and if so, this data can be loaded quickly from the faster memory.

There are three basic forms of locality:

- **Temporal locality**—Recently accessed items tend to be accessed again in the near future.
- **Spatial locality**—Accesses tend to be clustered in the address space (for example, as in arrays or loops).
- **Sequential locality**—Instructions tend to be accessed sequentially.

The locality principle provides the opportunity for a system to use a small amount of very fast memory to effectively accelerate the majority of memory accesses. Typically, only a small amount of the entire memory space is being accessed at any given time, and values in that space are being accessed repeatedly. Therefore, we can copy those values from a slower memory to a smaller but faster memory that resides higher in the hierarchy. This results in a memory system that can store a large amount of information in a large but low-cost memory, yet provide nearly the same access speeds that would result from using very fast but expensive memory.

6.4 CACHE MEMORY

A computer processor is very fast and is constantly reading information from memory, which means it often has to wait for the information to arrive, because the memory access times are slower than the processor speed. A cache memory is a small, temporary, but fast memory that the processor uses for information it is likely to need again in the very near future.

Noncomputer examples of caching are all around us. Keeping them in mind will help you to understand computer memory caching. Think of a homeowner with a very large tool chest in the garage. Suppose you are this homeowner and have a home improvement project to work on in the basement. You know this project will require drills, wrenches, hammers, a tape measure, several types of saws, and many different types and sizes of screwdrivers. The first thing you want to do is measure and then cut some wood. You run out to the garage, grab the tape measure from a huge tool storage chest, run down to the basement, measure the wood, run back out to the garage, leave the tape measure, grab the saw, and then return to the basement with the saw and cut the wood. Now you decide to bolt some pieces of wood together. So you run to the garage, grab the drill set, go back down to the basement, drill the holes to put the bolts through, go back to the garage, leave the drill set, grab one wrench, go back to the basement, find out the wrench is the wrong size, go back to the tool chest in the garage, grab another wrench, run back downstairs … wait! Would you really work this way? No! Being a reasonable person, you think to yourself, “If I need one wrench, I will probably need another one of a different size soon anyway, so why not just grab the whole set of wrenches?” Taking this one step further, you reason, “Once I am done with one certain tool, there is a good chance I will need another soon, so why not just pack up a small toolbox and take it to the basement?” This way, you keep the tools you need close at hand, so access is faster. You have just cached some tools for easy access and quick use! The tools you are less likely to use remain stored in a location that is farther away and requires more time to access. This is all that cache memory does: It stores data that has been accessed and data that might be accessed by the CPU in a faster, closer memory.

Another cache analogy is found in grocery shopping. You seldom, if ever, go to the grocery store to buy one single item. You buy any items you require immediately in addition to items you will most likely use in the future. The grocery store is similar to main memory, and your home is the cache. As another example, consider how many of us carry around an entire phone book. Most of us have a small address book instead. We enter the names and numbers of people we tend to call more frequently; looking up a number in our address book is much quicker than finding a phone book, locating the name, and then getting the number. We tend to have the address book close at hand, whereas the phone book is probably located in our home, hidden in an end table or bookcase somewhere. The phone book is something we do not use frequently, so we can afford to store it in a little more out-of-the-way location. Comparing the size of our address book to the telephone book, we see that the address book “memory” is much smaller than that of a telephone book. But the probability is high that when we make a call, it is to someone in our address book.

Students doing research offer another commonplace cache example. Suppose you are writing a paper on quantum computing. Would you go to the library, check out one book, return home, get the necessary information from that book, go back to the library, check out another book, return home, and so on? No; you would go to the library and check out all the books you might need and bring them all home. The library is analogous to main memory, and your home is, again, similar to cache.

And as a last example, consider how one of your authors uses her office. Any materials she does not need (or has not used for a period of more than six months) get filed away in a large set of filing cabinets. However, frequently used “data” remains piled on her desk, close at hand, and easy (sometimes) to find. If she needs something from a file, she more than likely pulls the entire file, not simply one or two papers from the folder. The entire file is then added to the pile on her desk. The filing cabinets are her “main memory,” and her desk (with its many unorganized-looking piles) is the cache.

Cache memory works on the same basic principles as the preceding examples, by copying frequently used data into the cache rather than requiring an access to main memory to retrieve the data. Cache can be as unorganized as your author’s desk or as organized as your address book. Either way, however, the data must be accessible (locatable). Cache memory in a computer differs from our real-life examples in one important way: The computer really has no way to know, *a priori*, what data is most likely to be accessed, so it uses the locality principle and transfers an entire block from main memory into cache whenever it has to make a main memory access. If the probability of using something else in that block is high, then transferring the entire block saves on access time. The cache location for this new block depends on two things: the cache mapping policy (discussed in the next section) and the cache size (which affects whether there is room for the new block).

The size of cache memory can vary enormously. A typical personal computer’s level 2 (L2) cache is

256K or 512K. Level 1 (L1) cache is smaller, typically 8K to 64K. L1 cache resides on the processor, whereas L2 cache often resides between the CPU and main memory. L1 cache is, therefore, faster than L2 cache. The relationship between L1 and L2 cache can be illustrated using our grocery store example: If the store is main memory, you could consider your refrigerator the L2 cache, and the actual dinner table the L1 cache. We note that with newer processors, instead of incorporating L2 cache into the motherboard, some CPUs actually incorporate L2 cache as well as L1 cache. In addition, many systems employ L3 cache; L3 cache is specialized to work in unison with L1 and L2 caches.

The purpose of cache is to speed up memory accesses by storing recently used data closer to the CPU, instead of storing it in main memory. Although cache is not as large as main memory, it is considerably faster. Whereas main memory is typically composed of DRAM with, say, a 50ns access time, cache is typically composed of SRAM, providing faster access with a much shorter cycle time than DRAM (a typical cache access time is 10ns). Cache does not need to be very large to perform well. A general rule of thumb is to make cache small enough so that the overall average cost per bit is close to that of main memory, but large enough to be beneficial. Because this fast memory is quite expensive, it is not feasible to use the technology found in cache memory to build all of main memory.

What makes cache “special”? Cache is not accessed by address; it is accessed by content. For this reason, cache is sometimes called **content addressable memory** or **CAM**. Under most cache mapping schemes, the cache entries must be checked or searched to see if the value being requested is stored in cache. To simplify this process of locating the desired data, various cache mapping algorithms are used.

6.4.1 Cache Mapping Schemes

For cache to be functional, it must store useful data. However, this data becomes useless if the CPU can’t find it. When accessing data or instructions, the CPU first generates a main memory address. If the data has been copied to cache, the address of the data in cache is not the same as the main memory address. For example, data located at main memory address 0x2E3 could be located in the very first location in cache. How, then, does the CPU locate data when it has been copied into cache? The CPU uses a specific mapping scheme that “converts” the main memory address into a cache location.

This address conversion is done by giving special significance to the bits in the main memory address. We first divide the bits into distinct groups we call **fields**. Depending on the mapping scheme, we may have two or three fields. How we use these fields depends on the particular mapping scheme being used. The mapping scheme determines where the data is placed when it is originally copied into cache and also provides a method for the CPU to find previously copied data when searching cache. The mapping schemes include direct mapping, fully associative mapping, and set associative mapping.

Before we discuss these mapping schemes, it is important to understand how data is copied into cache. Main memory and cache are both divided into the same size blocks (the size of these blocks varies). When a memory address is generated, cache is searched first to see if the required data at that address exists there. When the requested data is not found in cache, the entire main memory block in which the requested memory address resides is loaded into cache. As previously mentioned, this scheme is successful because of the principle of locality—if an address was just referenced, there is a good chance addresses in the same general vicinity will soon be referenced as well. Therefore, one missed address often results in several found addresses. For example, when you are in the basement and you first need tools, you have a “miss” and must go to the garage. If you gather up a set of tools that you might need and return to the basement, you hope that you’ll have several “hits” while working on your home improvement project and don’t have to make many more trips to the garage. Because accessing data in cache (a tool already in the basement) is faster than accessing data in main memory (going to the garage yet again!), cache memory speeds up the overall access time.

So how do we use fields in the main memory address? One field of the main memory address points us to a location in cache in which the data resides if it is resident in cache (this is called a **cache hit**), or where it is to be placed if it is not resident (which is called a **cache miss**). (This is slightly different for associative mapped cache, which we discuss shortly.) The cache block referenced is then checked to see if it is valid. This is done by associating a **valid bit** with each cache block. A valid bit of 0 means the cache block is not valid (we have a cache miss), and we must access main memory. A valid bit of 1 means it is valid (we may have a cache hit, but we need to complete one more step before we know for sure). We then compare the tag in the cache block to the **tag field** of our address. (The tag is a special group of bits derived from the main memory address that is stored with its corresponding block in cache.) If the tags are the same, then we have found the desired cache block (we have a cache hit). At this point, we need to locate the desired data in the block; this can be done using a different portion of the main memory address called the **offset field**. All cache mapping schemes require an offset field; however, the remaining fields are determined by the mapping scheme. We now discuss the three main cache mapping schemes. In these examples, we assume that if the cache block is empty, it is not valid; if it contains data, it is valid. Therefore, we do not include the valid bit in our discussions.

Before we discuss the mapping schemes, we mention one very important point. Some computers are byte addressable; others are word addressable. Key in determining how the mapping schemes work is knowing how many addresses are contained in main memory, in cache, and in each block. If a machine is byte addressable, we are concerned with the number of bytes; if it is word addressable, we are concerned with the number of words, regardless of the size of the word.

Direct Mapped Cache

Direct mapped cache assigns cache mappings using a modular approach. Because there are more main memory blocks than there are cache blocks, it should be clear that main memory blocks compete for cache locations. Direct mapping maps block X of main memory to block Y of cache, mod N, where N is the total number of blocks in cache. For example, if cache contains 4 blocks, then main memory block 0 maps to cache block 0, main memory block 1 maps to cache block 1, main memory block 2 maps to cache block 2, main memory block 3 maps to cache block 3, main memory block 4 maps to cache block 0, and so on. This is illustrated in Figure 6.2. In Figure 6.2a, we see 8 main memory blocks mapping to 4 cache blocks, resulting in 2 memory blocks mapping to each cache block. In Figure 6.2b, there are 16 memory blocks mapping to 4 cache blocks; we have doubled the size of memory and doubled the contention for each cache block. (We will soon see that this contention is one of the major disadvantages of direct mapped cache.)

You may be wondering, if main memory blocks 0 and 4 both map to cache block 0, how does the CPU know which block actually resides in cache block 0 at any given time? The answer is that each block is copied to cache and identified by the tag previously described. This means the tag for a particular block must be stored in cache with that block, as we shall see shortly.

To perform direct mapping, the binary main memory address is partitioned into the fields shown in Figure 6.3.

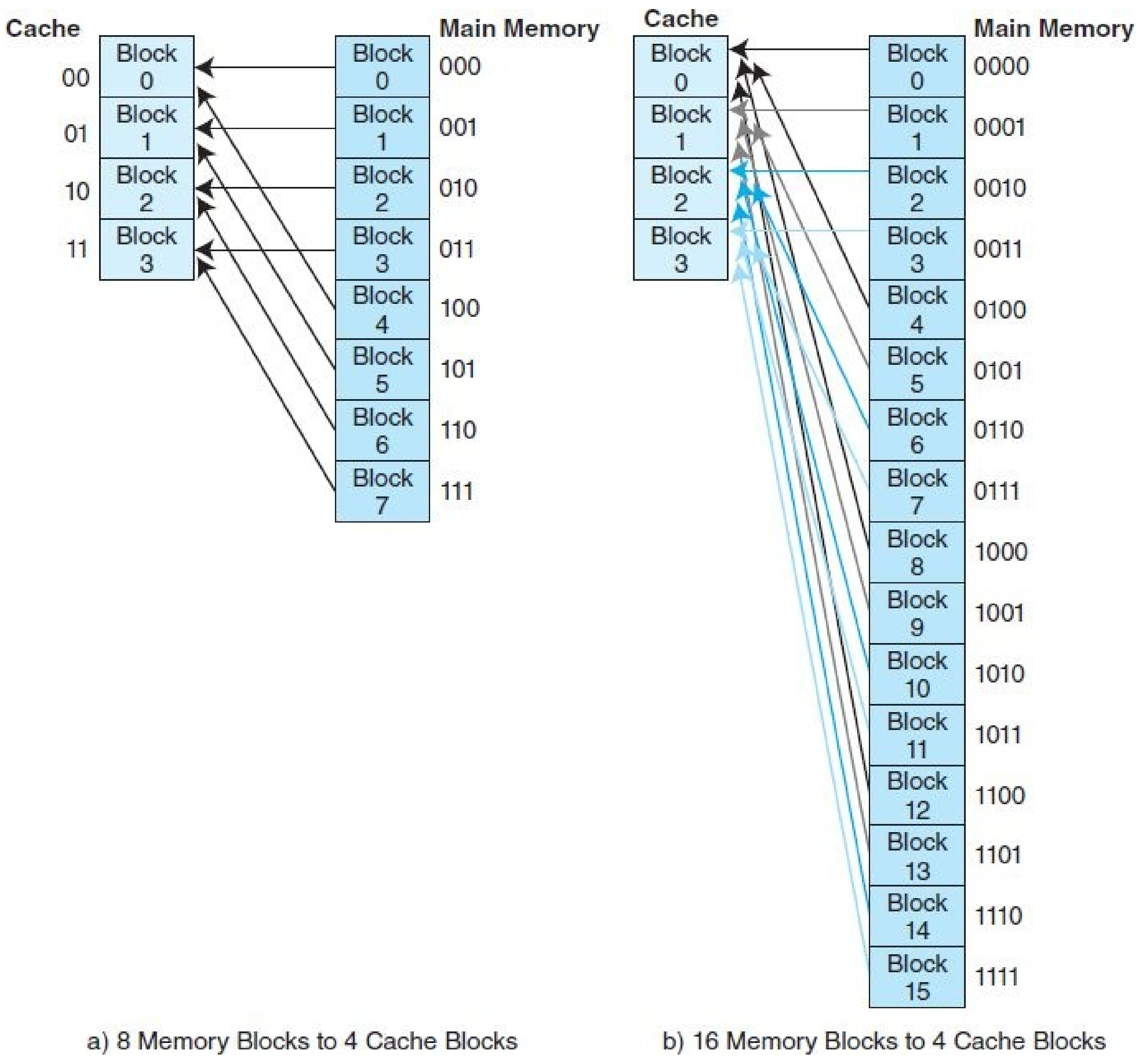


FIGURE 6.2 Direct Mapping of Main Memory Blocks to Cache Blocks

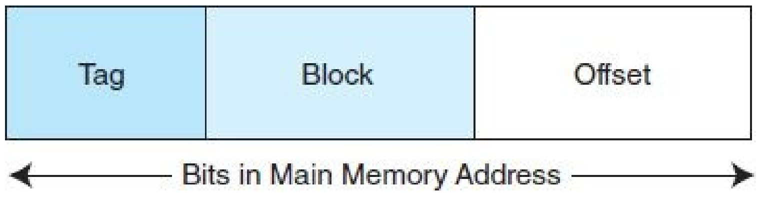


FIGURE 6.3 The Format of a Main Memory Address Using Direct Mapping

The size of each field depends on the physical characteristics of main memory and cache. The **offset** field uniquely identifies an address within a specific block; therefore, it must contain the appropriate number of bits to do this. The number of bytes (if the machine is byte addressable) or words (if the machine is word addressable) in each block dictates the number of bits in the offset field. This is also true of the **block** field—it must select a unique block of cache. (The number of blocks in cache dictates the number of bits in the block field.) The **tag** field is whatever is left over. When a block of main memory is copied to cache, this tag is stored with the block and uniquely identifies this block. The total of all three fields must, of course, add up to the number of bits in a main memory address. Let’s look at some examples.

EXAMPLE 6.1 Consider a byte-addressable main memory consisting of 4 blocks, and a cache with 2 blocks, where each block is 4 bytes. This means Block 0 and 2 of main memory map to Block 0 of cache, and Blocks 1 and 3 of main memory map to Block 1 of cache. (This is easy for us to determine simply by using modular arithmetic, because 0 mod 2 = 0, 1 mod 2 = 1, 2 mod 2 = 0, and 3 mod 2 = 1. However, the computer must use the fields in the memory address itself.) Using the tag, block, and offset fields, we can see how main memory maps to cache, as shown in Figure 6.4 and explained as follows.

First, we need to determine the address format for mapping. Each block is 4 bytes, so the offset field must contain 2 bits; there are 2 blocks in cache, so the block field must contain 1 bit; this leaves 1 bit for the tag (because a main memory address has 4 bits because there are a total of $2^4 = 16$ bytes). The format is shown in Figure 6.4a.

Suppose we need to access main memory address 0x03 (0011 in binary). If we partition 0011 using the address format from Figure 6.4a, we get Figure 6.4b. This tells us that the main memory address 0011 maps to cache block 0 (because the block field evaluates to 0). Figure 6.4c shows this mapping, along with the tag that is also stored with the data.

Suppose we now access main memory address 0x0A = 1010_2. Using the same address format, we see that it maps to cache block 0 (see Figure 6.4d). However, if we compare the tag from address 1010 (the tag is 1) to what is currently stored in cache at block 0 (that tag is 0), they do not match. Therefore, the data currently in cache block 0 is removed, and the data from main memory block 3 replaces it and changes the tag, resulting in Figure 6.4e.

Let's consider the memory format for a larger example.

EXAMPLE 6.2 Assume that a byte-addressable memory consists of 2^{14} bytes, cache has 16 blocks, and each block has 8 bytes. From this, we determine that $\frac{2^{14}}{2^3} = 2^{11}$ memory blocks. We know that each main memory address requires 14 bits. Of this 14-bit address field, the rightmost 3 bits reflect the offset field (we need 3 bits to uniquely identify one of 8 bytes in a block). We need 4 bits to select a specific block in cache, so the block field consists of the middle 4 bits. The remaining 7 bits make up the tag field. The fields with sizes are illustrated in Figure 6.5.

As mentioned previously, the tag for each block is stored with that block in the cache. In this example, because main memory blocks 0 and 16 both map to cache block 0, the tag field would allow the system to differentiate between block 0 and block 16. The binary addresses in block 0 differ from those in block 16 in the upper leftmost 7 bits, so the tags are different and unique.

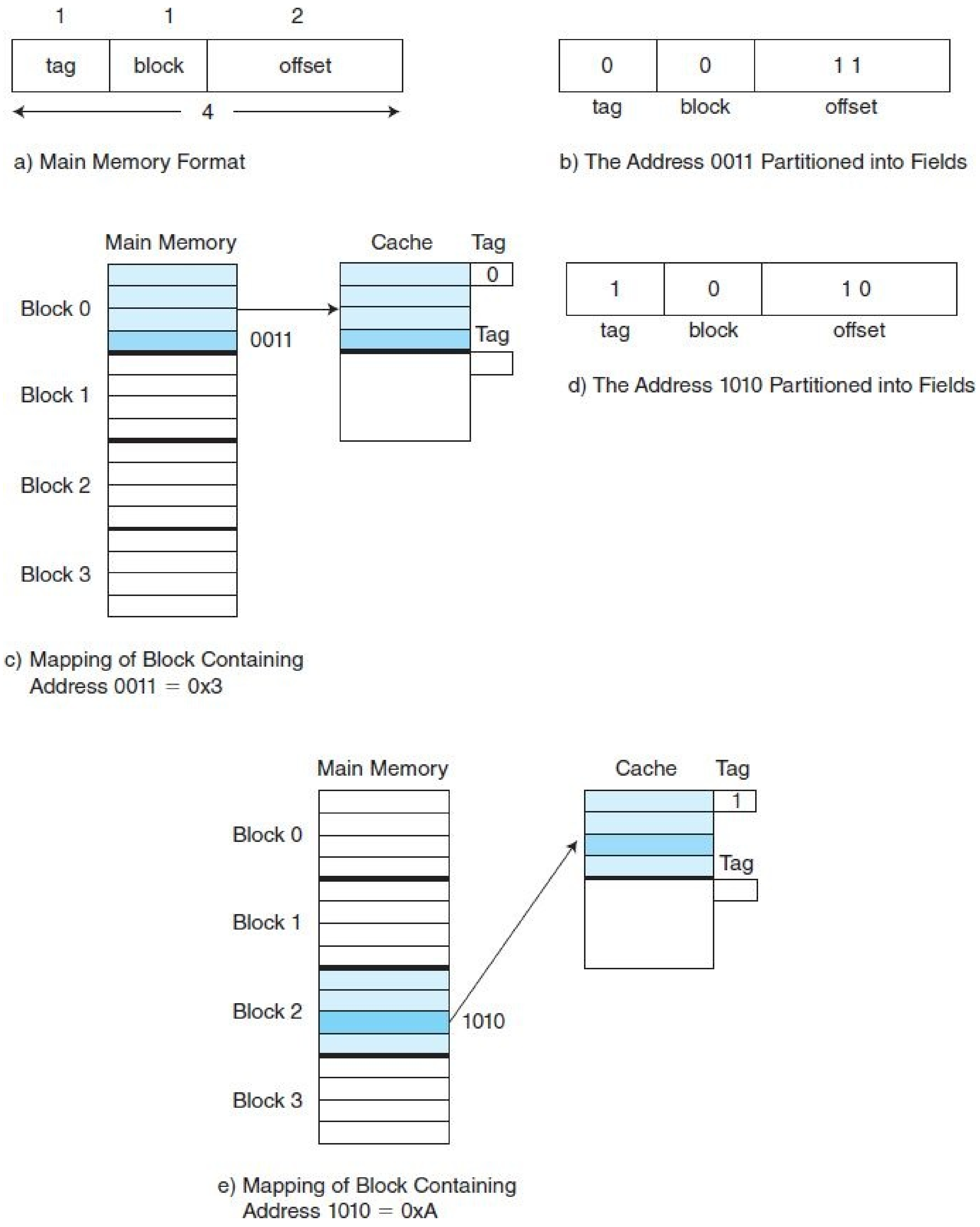


FIGURE 6.4 Diagrams for Example 6.1

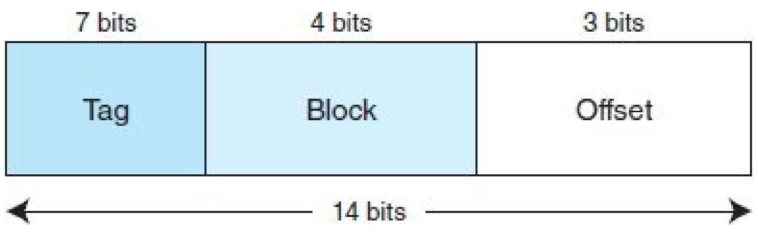


FIGURE 6.5 The Main Memory Address Format for Example 6.2

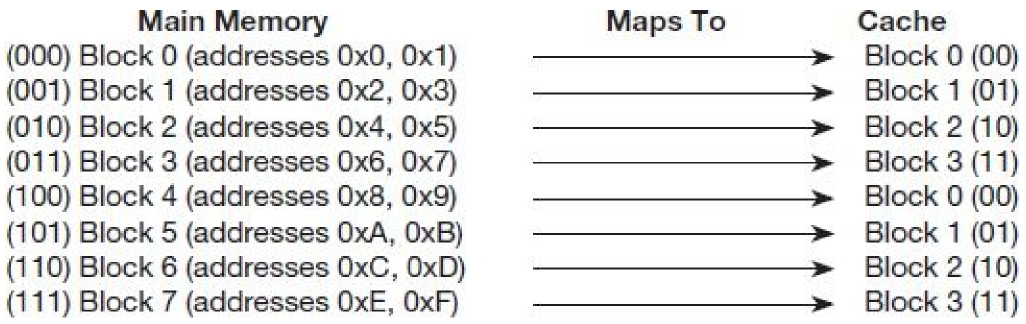
EXAMPLE 6.3 Let's look at a slightly larger example. Suppose we have a byte-addressable system using direct mapping with 16 bytes of main memory divided into 8 blocks (so each block has 2 bytes). Assume the cache is 4 blocks in size (for a total of 8 bytes). Figure 6.6 shows how the main memory blocks map to cache.

We know:

- A main memory address has 4 bits (because there are 2^4 or 16 bytes in main memory).
- This 4-bit main memory address is divided into three fields: The offset field is 1 bit (we need only 1 bit to differentiate between the two words in a block); the block field is 2 bits (we have 4 blocks in cache memory and need 2 bits to uniquely identify each block); and the tag field has 1 bit (this is all that is left over).

The main memory address is divided into the fields shown in Figure 6.7.

Suppose we generate the main memory address 0x9. We can see from the mapping listing in Figure 6.6 that address 0x9 is in main memory block 4 and should map to cache block 0 (which means the contents of main memory block 4 should be copied into cache block 0). The computer, however, uses the actual main memory address to determine the cache mapping block. This address, in binary, is represented in Figure 6.8.



Main Memory	Maps To	Cache
(000) Block 0 (addresses 0x0, 0x1)	⟶	Block 0 (00)
(001) Block 1 (addresses 0x2, 0x3)	⟶	Block 1 (01)
(010) Block 2 (addresses 0x4, 0x5)	⟶	Block 2 (10)
(011) Block 3 (addresses 0x6, 0x7)	⟶	Block 3 (11)
(100) Block 4 (addresses 0x8, 0x9)	⟶	Block 0 (00)
(101) Block 5 (addresses 0xA, 0xB)	⟶	Block 1 (01)
(110) Block 6 (addresses 0xC, 0xD)	⟶	Block 2 (10)
(111) Block 7 (addresses 0xE, 0xF)	⟶	Block 3 (11)

FIGURE 6.6 The Memory from Example 6.3 Mapped to Cache

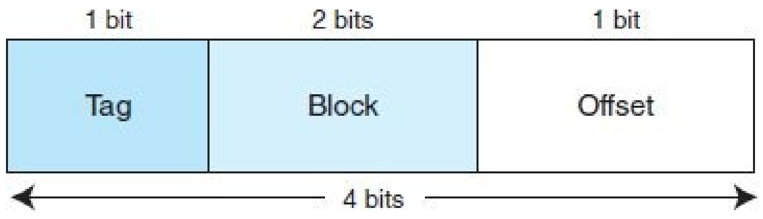


FIGURE 6.7 The Main Memory Address Format for Example 6.3

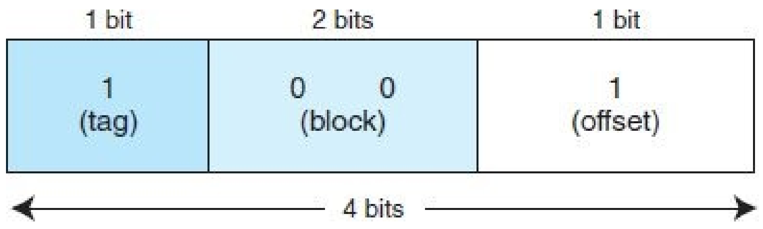


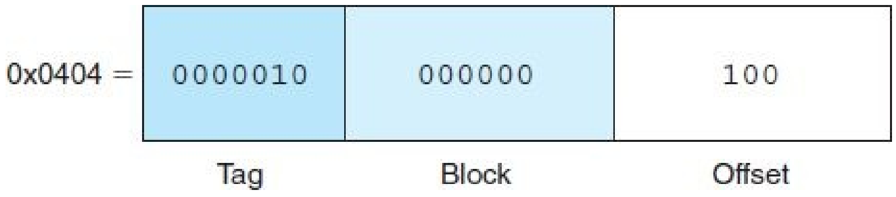
FIGURE 6.8 The Main Memory Address 9 = 1001_2 Split into Fields

When the CPU generates this address, it first takes the block field bits 00 and uses these to direct it to the proper block in cache. 00 indicates that cache block 0 should be checked. If the cache block is valid, it then compares the tag field value of 1 (in the main memory address) to the tag associated with cache block 0. If the cache tag is 1, then block 4 currently resides in cache block 0. If the tag is 0, then block 0 from main memory is located in block 0 of cache. (To see this, compare main memory address 0x9 = 1001_2, which is in block 4, to main memory address 0x1 = 0001_2, which is in block 0. These two addresses differ only in the leftmost bit, which is the bit used as the tag by the cache.) Assuming that the tags match, which means that block 4 from main memory (with addresses 0x8 and 0x9) resides in cache block 0, the offset field value of 1 is used to select one of the two bytes residing in the block. Because the bit is 1, we select the byte with offset 1, which results in retrieving the data copied from main memory address 0x9.

Suppose the CPU now generates address 0x4 = 0100_2. The middle two bits (10) direct the search to cache block 2. If the block is valid, the leftmost tag bit (0) would be compared to the tag bit stored with the cache block. If they match, the first byte in that block (of offset 0) would be returned to the CPU. To make sure you understand this process, perform a similar exercise with the main memory address 0xC = 1100_2.

Let's move on to a larger example.

EXAMPLE 6.4 Suppose we have a byte-addressable system using 16-bit main memory addresses and 64 blocks of cache. If each block contains 8 bytes, we know that the main memory 16-bit address is divided into a 3-bit offset field, a 6-bit block field, and a 7-bit tag field. If the CPU generates the main memory address:



it would look in block 0 of cache, and if it finds a tag of 0000010, the byte at offset 4 in this block would be returned to the CPU.

So, to summarize, direct mapped cache implements a mapping scheme that results in main memory blocks being mapped in a modular fashion to cache blocks. To determine how the mapping works, you must know several things:

- How many bits are in the main memory address (which is determined by how many addresses exist in main memory)
- How many blocks are in cache (which determines the size of the block field)
- How many addresses are in a block (which determines the size of the offset field)

Once you know these values, you can use the direct mapping address format to locate the block in cache to which a main memory block maps. Once you find the cache block, you can determine if the block currently in cache is the one you want by checking the tag. If the tags match (the tag from the memory address field and the tag in cache), use the offset field to find the desired data in the cache block.

Fully Associative Cache

Direct mapped cache is not as expensive as other caches because the mapping scheme does not require any searching. Each main memory block has a specific location to which it maps in cache; when a main memory address is converted to a cache address, the CPU knows exactly where to look in the cache for that memory block by simply examining the bits in the block field. This is similar to your address book: The pages often have an alphabetic index, so if you are searching for "Joe Smith," you would look under the "S" tab.

Instead of specifying a unique location for each main memory block, we can look at the opposite extreme: allowing a main memory block to be placed anywhere in cache. The only way to find a block mapped this way is to search all of cache. (This is similar to your author's desk!) This requires the entire cache to be built from **associative memory** so it can be searched in parallel. That is, a single search must compare the requested tag to *all* tags in cache to determine whether the desired data block is present in cache. Associative memory requires special hardware to allow associative searching and is, thus, quite expensive.

Let's look at associative memory in more detail so we can better understand why it is so expensive. We have already mentioned that fully associative cache allows a memory block to be placed anywhere in cache, which means we now have to search to find it, and to make this searching efficient, we search in parallel. But how do we do this in hardware? First, there must be a comparator circuit for each block in cache; this circuit outputs a 1 if the two input values match.

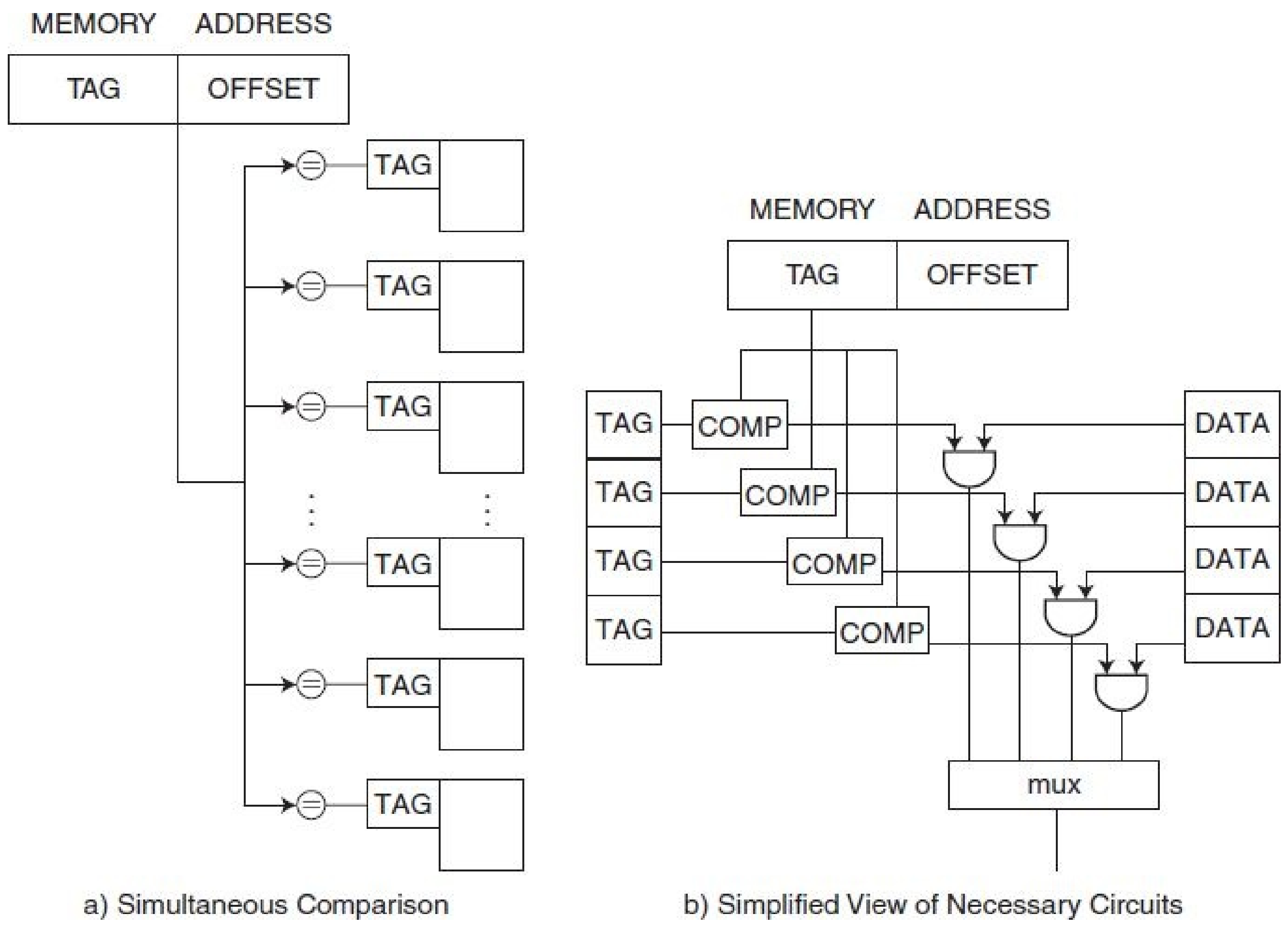


FIGURE 6.9 Associative Cache

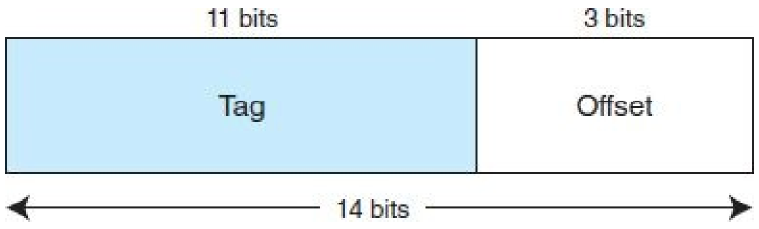


FIGURE 6.10 The Main Memory Address Format for Associative Mapping

The tag field from the main memory address is compared, simultaneously, to each tag from every cache block, as we see in Figure 6.9a. This memory is expensive because it requires not only a set of multiplexers to select the appropriate data once the correct cache block is identified but also the additional circuitry to implement the comparators. Figure 6.9b shows a simplified view of the circuits required for fully associative memory.

Using associative mapping, the main memory address is partitioned into two pieces, the tag and the offset. Recall Example 6.2, which had a memory with 2^{14} bytes, a cache with 16 blocks, and blocks of 8 bytes. Using fully associative mapping instead of direct mapping, we see from Figure 6.10 that the offset field is still 3 bits, but now the tag field is 11 bits. This tag must be stored with each block in cache. When the cache is searched for a specific main memory block, the tag field of the main memory address is compared to all the valid tag fields in cache; if a match is found, the block is found. (Remember, the tag uniquely identifies a main memory block.) If there is no match, we have a cache miss and the block must be transferred from main memory.

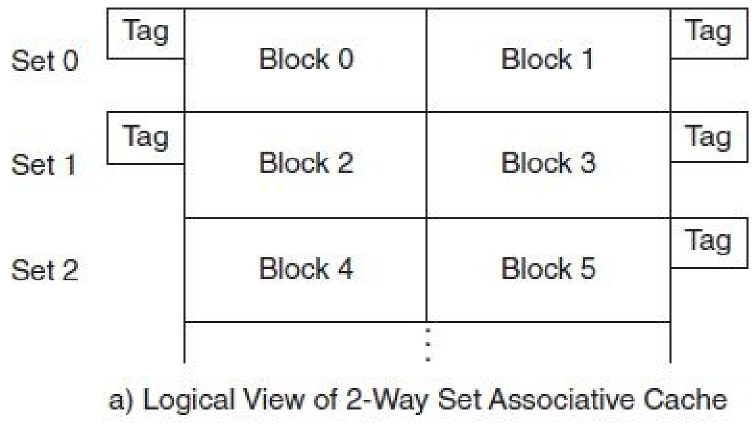
With direct mapping, if a block already occupies the cache location where a new block must be placed, the block currently in cache is removed (it is written back to main memory if it has been modified or simply overwritten if it has not been changed). With fully associative mapping, when cache is full, we need a replacement algorithm to decide which block we wish to throw out of cache (we call this our **victim block**). A simple first-in, first-out algorithm would work, as would a least recently used algorithm. There are many replacement algorithms that can be used; these are discussed in Section 6.4.2.

So, to summarize, fully associative cache allows a main memory block to be mapped to *any* block in cache. Once a main memory block resides in cache, to locate a specific byte, the computer compares the tag field of the main memory address to all tags stored in cache (in one comparison). Once the specific cache block is located, the offset field is used to locate the required data within that block. If the tag of the memory address doesn't match any tags in cache, the memory block containing the desired data must be transferred into cache. This transfer may require picking a victim block to transfer out of cache.

Set Associative Cache

Because of its speed and complexity, associative cache is very expensive. Although direct mapping is inexpensive, it is very restrictive. To see how direct mapping limits cache usage, suppose we are running a program on the architecture described in Example 6.3. Suppose the program is using block 0, then block 4, then 0, then 4, and so on as it executes instructions. Blocks 0 and 4 both map to the same location, which means the program would repeatedly throw out 0 to bring in 4, then throw out 4 to bring in 0, even though there are additional blocks in cache not being used. Fully associative cache remedies this problem by allowing a block from main memory to be placed anywhere. However, it requires a larger tag to be stored with the block (which results in a larger cache) in addition to requiring special hardware for searching of all blocks in cache simultaneously (which implies a more expensive cache). We need a scheme somewhere in the middle.

The third mapping scheme we introduce is ***N*-way set associative cache mapping**, a combination of these two approaches. This scheme is similar to direct mapped cache, in that we use the address to map the block to a certain cache location. The important difference is that instead of mapping to a single cache block, an address maps to a **set** of several cache blocks. All sets in cache must be the same size. This size can vary from cache to cache. For example, in a 2-way set associative cache, there are 2 cache blocks per set, as seen in Figure 6.11. It is easiest to view set associative cache logically as a two-dimensional cache. In Figure 6.11a, we see a 2-way set associative cache, where each set contains 2 blocks, thus giving us rows and columns in cache. However, cache memory is actually linear. In Figure 6.11b, we see how set associative cache is implemented in a linear memory. A 4-way set associative cache has 4 blocks per set; an 8-way has 8 blocks per set, and so on. Once the desired set is located, the cache is treated as associative memory; the tag of the main memory address must be compared to the tags of each block in the set. Instead of requiring a comparator circuit for each block in cache, set associative cache only needs a comparator for each block in a set. For example, if there are 64 total cache blocks, using 4-way set associative mapping, the cache needs only 4 total comparators, not 64. Direct mapped cache is a special case of *N*-way set associative cache mapping where the set size is one. Fully associative cache, where cache has *n* blocks, is a special case of set associative mapping in which there is only one set of size *n*.



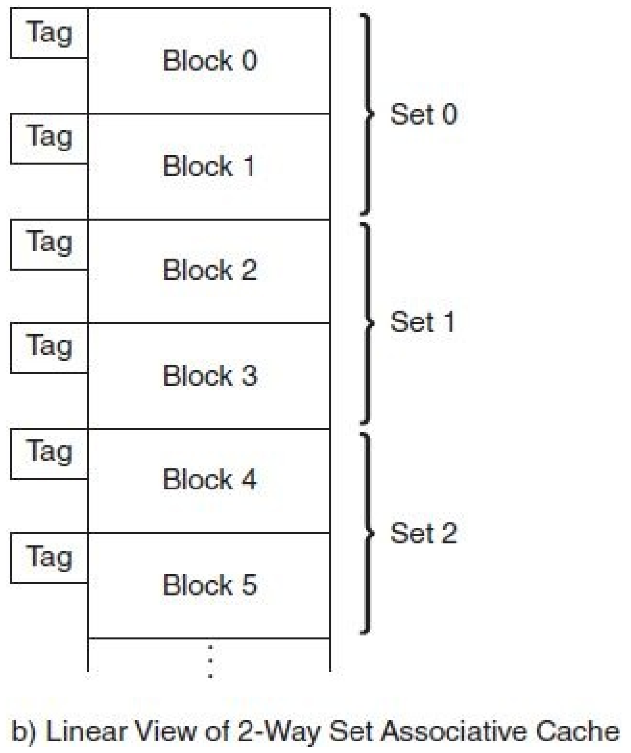


FIGURE 6.11 A 2-Way Set Associative Cache

In set associative cache mapping, the main memory address is partitioned into three pieces: the tag field, the set field, and the offset field. The tag and offset fields assume the same roles as before; the set field indicates into which cache set the main memory block maps, as we see in the following example.

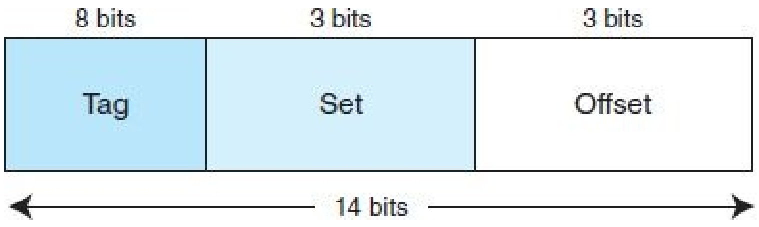


FIGURE 6.12 Format for Set Associative Mapping for Example 6.5

EXAMPLE 6.5 Suppose we are using 2-way set associative mapping with a byte-addressable main memory of 2^{14} bytes and a cache with 16 blocks, where each block contains 8 bytes. If cache consists of a total of 16 blocks, and each set has 2 blocks, then there are 8 sets in cache. Therefore, the set field is 3 bits, the offset field is 3 bits, and the tag field is 8 bits. This is illustrated in Figure 6.12.

The set field of a main memory address specifies a unique set in cache for the given main memory block. All blocks in that cache set are then searched for a matching tag. An associative search must be performed, but the search is restricted to the specified set instead of the entire cache. This significantly reduces the cost of the specialized hardware. For example, in a 2-way set associative cache, the hardware searches only 2 blocks in parallel.

Let’s look at an example to illustrate the differences among the various mapping schemes.

EXAMPLE 6.6 Suppose a byte-addressable memory contains 1MB and cache consists of 32 blocks, where each block contains 16 bytes. Using direct mapping, fully associative mapping, and 4-way set associative mapping, determine where the main memory address 0x326A0 maps to in cache by specifying either the cache block or cache set.

First, we note that a main memory address has 20 bits. The main memory format for direct mapped cache is shown in Figure 6.13:

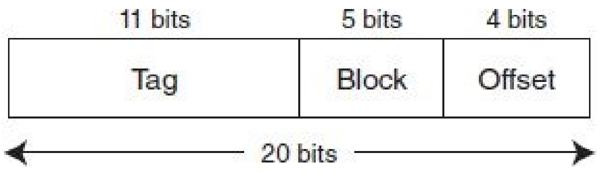


FIGURE 6.13 Direct Mapped Memory Format for Example 6.6

If we represent our main memory address 0x326A0 in binary and place the bits into the format, we get the fields shown in Figure 6.14:

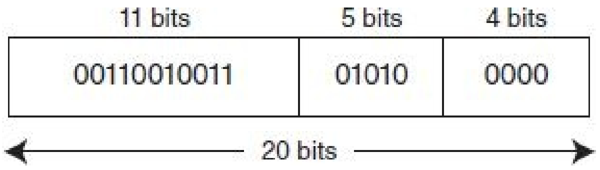


FIGURE 6.14 The Address 0x326A0 from Example 6.6 Divided into Fields for Direct Mapping

This tells us that memory address 0x326A0 maps to cache block 01010 (or block 10).

If we are using fully associative cache, the memory address format is:

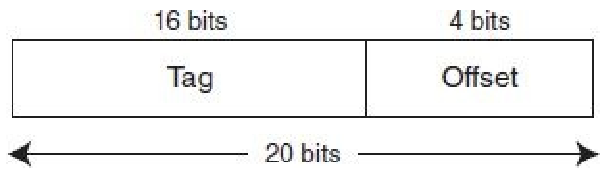


FIGURE 6.15 Fully Associative Memory Format for Example 6.6

However, dividing the main memory address into these fields won't help us determine the mapping location for the main memory block containing this address because with fully associative cache, the block can map anywhere. If we are using 4-way set associative mapping, the memory address format is shown is Figure 6.16:

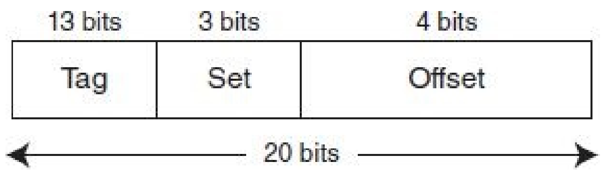


FIGURE 6.16 4-Way Set Associative Mapped Memory Format for Example 6.6

The set field has 3 bits because there are only 8 sets in cache (where each set holds 4 blocks). If we divide our main memory address into these fields, we get:

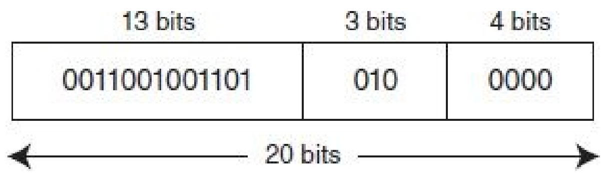


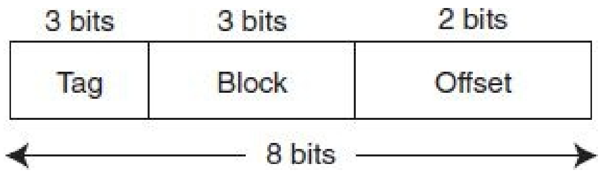
FIGURE 6.17 The Address 0x326A0 from Example 6.6 Divided into Fields for Set Associative Mapping

which tells us that main memory address 0x326A0 maps to cache set $010_2 = 2$. However, the set would still need to be searched (by comparing the tag in the address to all the tags in cache set 2) before the desired data could be found.

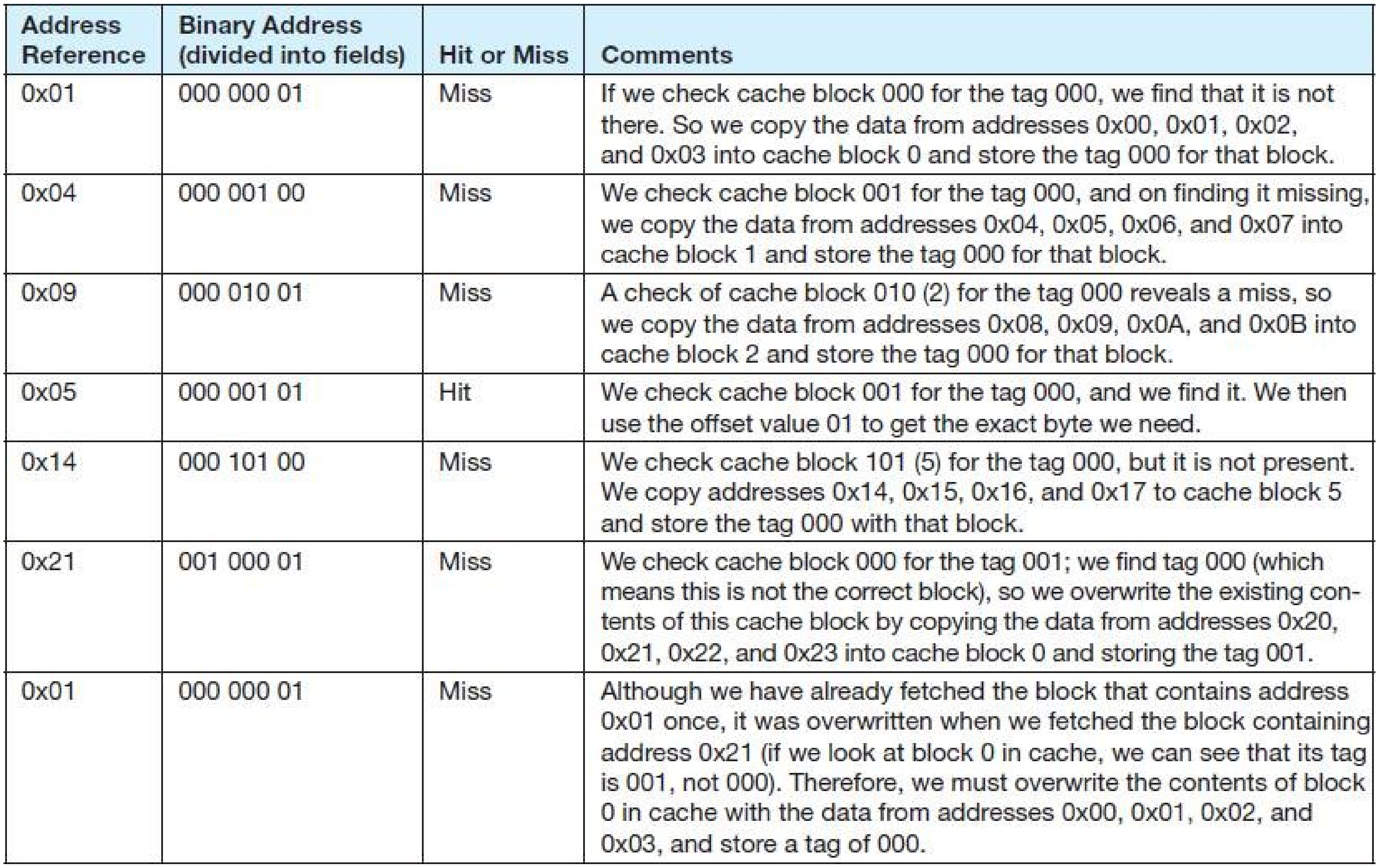
Let's look at one more example to reinforce the concepts of cache mapping.

EXAMPLE 6.7 Suppose we have a byte-addressable computer with a cache that holds 8 blocks of 4 bytes each. Assuming that each memory address has 8 bits and cache is originally empty, for each of the cache mapping techniques, direct mapped, fully associative, and 2-way set associative, trace how cache is used when a program accesses the following series of addresses in order: 0x01, 0x04, 0x09, 0x05, 0x14, 0x21, and 0x01.

We start with direct mapped. First, we must determine the address format. Each block has 4 bytes, which requires 2 bits in the offset field. Cache has a total of 8 blocks, so we need 3 bits in the block field. An address has 8 bits, so that leaves 3 bits for the tag field, resulting in:



Now that we know the address format, we can begin tracing the program.



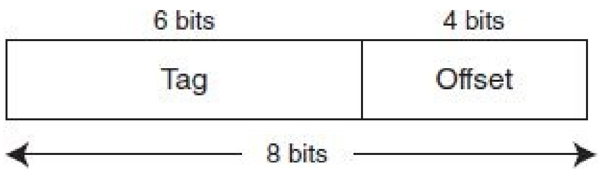
Address Reference	Binary Address (divided into fields)	Hit or Miss	Comments
0x01	000 000 01	Miss	If we check cache block 000 for the tag 000, we find that it is not there. So we copy the data from addresses 0x00, 0x01, 0x02, and 0x03 into cache block 0 and store the tag 000 for that block.
0x04	000 001 00	Miss	We check cache block 001 for the tag 000, and on finding it missing, we copy the data from addresses 0x04, 0x05, 0x06, and 0x07 into cache block 1 and store the tag 000 for that block.
0x09	000 010 01	Miss	A check of cache block 010 (2) for the tag 000 reveals a miss, so we copy the data from addresses 0x08, 0x09, 0x0A, and 0x0B into cache block 2 and store the tag 000 for that block.
0x05	000 001 01	Hit	We check cache block 001 for the tag 000, and we find it. We then use the offset value 01 to get the exact byte we need.
0x14	000 101 00	Miss	We check cache block 101 (5) for the tag 000, but it is not present. We copy addresses 0x14, 0x15, 0x16, and 0x17 to cache block 5 and store the tag 000 with that block.
0x21	001 000 01	Miss	We check cache block 000 for the tag 001; we find tag 000 (which means this is not the correct block), so we overwrite the existing contents of this cache block by copying the data from addresses 0x20, 0x21, 0x22, and 0x23 into cache block 0 and storing the tag 001.
0x01	000 000 01	Miss	Although we have already fetched the block that contains address 0x01 once, it was overwritten when we fetched the block containing address 0x21 (if we look at block 0 in cache, we can see that its tag is 001, not 000). Therefore, we must overwrite the contents of block 0 in cache with the data from addresses 0x00, 0x01, 0x02, and 0x03, and store a tag of 000.

A few things to note: We always copy an entire block from memory to cache any time there is a miss. We don’t take 4 bytes beginning with the address we missed; we must take the 4-byte block that contains the address we missed. For example, when we missed for memory address 0x09, we brought in 4 bytes of data beginning with address 0x08. Because 0x09 is at an offset of 1 in the block, we know the block begins with address 0x08. Also note the contention for block 0 of cache; this is a typical problem for direct mapped cache. The final contents of cache are:

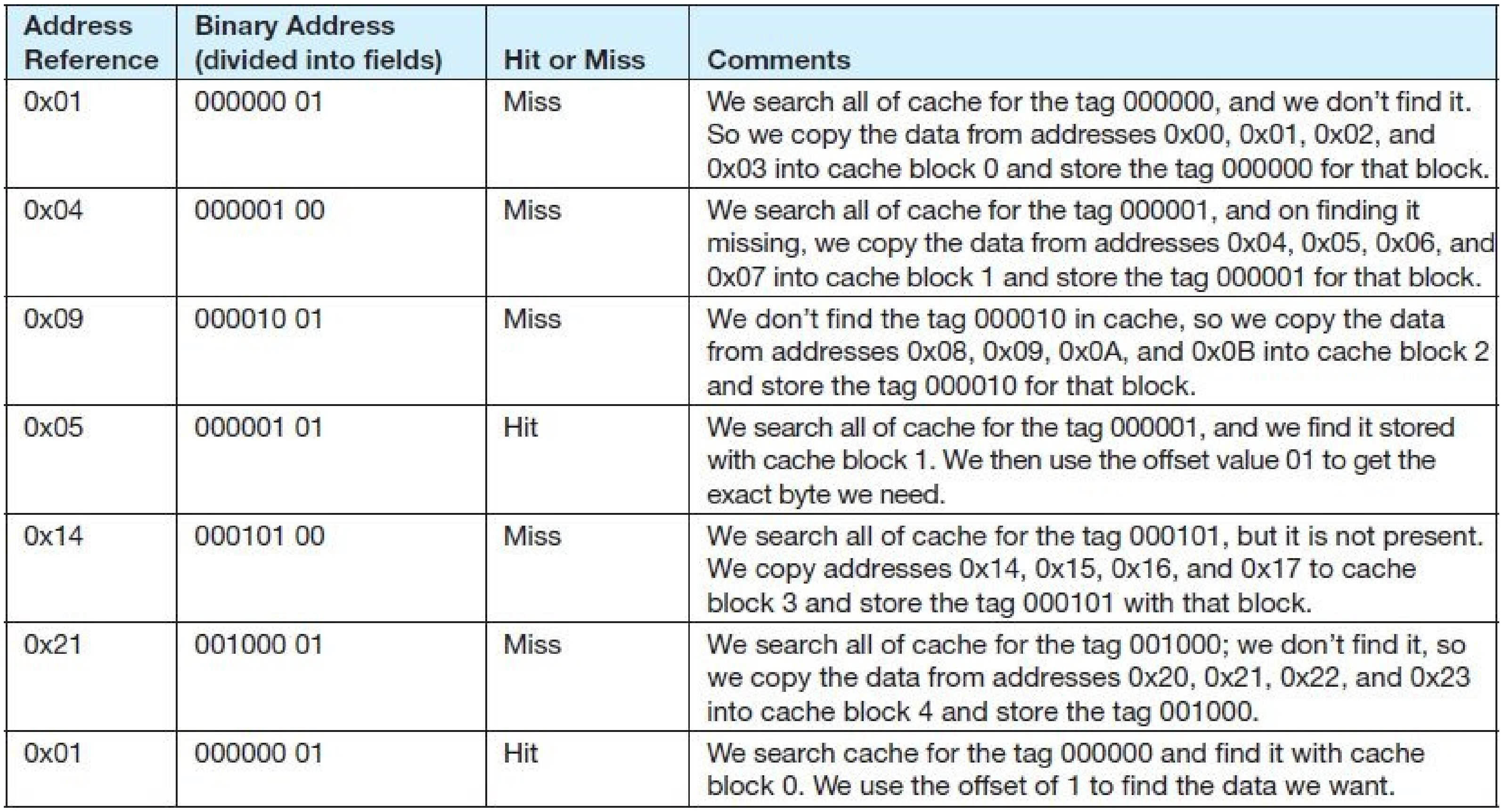
Block	**Cache Contents (represented by address)**	**Tag**
0	0x00, 0x01, 0x02, 0x03	000
1	0x04, 0x05, 0x06, 0x07	000
2	0x08, 0x09, 0x0A, 0x0B	000
3		
4		
5	0x14, 0x15, 0x16, 0x17	000

6		
7		

How do things change if we use fully associative cache? First, the address format becomes:



With fully associative cache, the block from memory can be stored anywhere. Let’s assume we will store it in the first available location; if all locations are full, we will “roll over” to the beginning and start again.

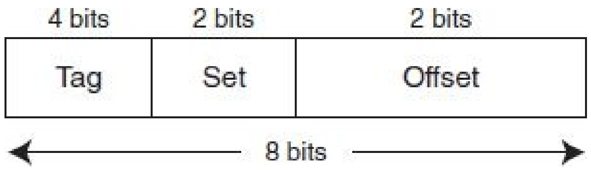


Address Reference	Binary Address (divided into fields)	Hit or Miss	Comments
0x01	000000 01	Miss	We search all of cache for the tag 000000, and we don’t find it. So we copy the data from addresses 0x00, 0x01, 0x02, and 0x03 into cache block 0 and store the tag 000000 for that block.
0x04	000001 00	Miss	We search all of cache for the tag 000001, and on finding it missing, we copy the data from addresses 0x04, 0x05, 0x06, and 0x07 into cache block 1 and store the tag 000001 for that block.
0x09	000010 01	Miss	We don’t find the tag 000010 in cache, so we copy the data from addresses 0x08, 0x09, 0x0A, and 0x0B into cache block 2 and store the tag 000010 for that block.
0x05	000001 01	Hit	We search all of cache for the tag 000001, and we find it stored with cache block 1. We then use the offset value 01 to get the exact byte we need.
0x14	000101 00	Miss	We search all of cache for the tag 000101, but it is not present. We copy addresses 0x14, 0x15, 0x16, and 0x17 to cache block 3 and store the tag 000101 with that block.
0x21	001000 01	Miss	We search all of cache for the tag 001000; we don’t find it, so we copy the data from addresses 0x20, 0x21, 0x22, and 0x23 into cache block 4 and store the tag 001000.
0x01	000000 01	Hit	We search cache for the tag 000000 and find it with cache block 0. We use the offset of 1 to find the data we want.

The final contents of cache are:

Block	**Cache Contents (represented by address)**	**Tag**
0	0x00, 0x01, 0x02, 0x03	000000
1	0x04, 0x05, 0x06, 0x07	000001
2	0x08, 0x09, 0x0A, 0x0B	000010
3	0x14, 0x15, 0x16, 0x17	000101
4	0x20, 0x21, 0x22, 0x23	001000
5		
6		

Now let's take a look at using 2-way set associative cache. Because cache is 8 blocks, and each set contains 2 blocks, cache has 4 sets. The address format for 2-way set associative cache is:



If we trace through the memory references, we get the following:

Address Reference	Binary Address (divided into fields)	Hit or Miss	Comments
0x01	0000 00 01	Miss	We search in set 0 of cache for a block with the tag 0000, and we find it is not there. So we copy the data from addresses 0x00, 0x01, 0x02, and 0x03 into set 0 (so now set 0 has one used block and one free block) and store the tag 0000 for that block. It does not matter which set we use; for consistency, we put the data in the first set.
0x04	0000 01 00	Miss	We search set 1 for a block with the tag 0000, and on finding it missing, we copy the data from addresses 0x04, 0x05, 0x06, and 0x07 into set 1, and store the tag 0000 for that block.
0x09	0000 10 01	Miss	We search set 2 (10) for a block with the tag 0000, but we don't find one, so we copy the data from addresses 0x08, 0x09, 0x0A, and 0x0B into set 2 and store the tag 0000 for that block.
0x05	0000 01 01	Hit	We search set 1 for a block with the tag 0000, and we find it. We then use the offset value 01 within that block to get the exact byte we need.
0x14	0001 01 00	Miss	We search set 1 for a block with the tag 0001, but it is not present. We copy addresses 0x14, 0x15, 0x16, and 0x17 to set 1 and store the tag 0001 with that block. Note that set 1 is now full.
0x21	0010 00 01	Miss	We search cache set 0 for a block with the tag 0010; we don't find it, so we copy the data from addresses 0x20, 0x21, 0x22, and 0x23 into set 0 and store the tag 0010. Note that set 0 is now full.
0x01	0000 00 01	Hit	We search cache set 0 for a block with the tag 0000, and we find it. We use the offset of 1 within that block to find the data we want.

Set	Block	Cache Contents (represented by address)	Tag
0	First	0x00, 0x01, 0x02, 0x03	0000
	Second	0x20, 0x21, 0x22, 0x23	0010
1	First	0x04, 0x05, 0x06, 0x07	0000
	Second	0x14, 0x15, 0x16, 0x17	0001
2	First	0x08, 0x09, 0x0A, 0x0B	0000
	Second		
3	First		
	Second		

Set associative cache mapping is a good compromise between direct mapped and fully associative cache. Studies indicate that it exhibits good performance, and that 2-way up to 16-way caches perform almost as well as fully associative cache at a fraction of the cost. Therefore, most modern computer systems use some form of set associative cache mapping, with 4-way set associative being one of the most common.

6.4.2 Replacement Policies

In a direct-mapped cache, if there is contention for a cache block, there is only one possible action: The existing block is kicked out of cache to make room for the new block. This process is called **replacement**. With direct mapping, there is no need for a replacement policy because the location for each new block is predetermined. However, with fully associative cache and set associative cache, we need a replacement algorithm to determine the "victim" block to be removed from cache. When using fully associative cache, there are *K* possible cache locations (where *K* is the number of blocks in cache) to which a given main memory block may map. With *N*-way set associative mapping, a block can map to any of *N* different blocks within a given set. How do we determine which block in cache should be replaced? The algorithm for determining replacement is called the **replacement policy**.

There are several popular replacement policies. One that is not practical but that can be used as a benchmark by which to measure all others is the **optimal** algorithm. We like to keep values in cache that will be needed again soon, and throw out blocks that won't be needed again, or that won't be needed for some time. An algorithm that could look into the future to determine the precise blocks to keep or eject based on these two criteria would be best. This is what the optimal algorithm does. We want to replace the block that will not be used for the longest period of time in the future. For example, if the choice for the victim block is between block 0 and block 1, and block 0 will be used again in 5 seconds, whereas block 1 will not be used again for 10 seconds, we would throw out block 1. From a practical standpoint, we can't look into the future—but we can run a program and then rerun it, so we effectively *do* know the future. We can then apply the optimal algorithm on the second run. The optimal algorithm guarantees the lowest possible miss rate. Because we cannot see the future on every single program we run, the optimal algorithm is used only as a metric to determine how good or bad another algorithm is. The closer an algorithm performs to the optimal algorithm, the better.

We need algorithms that best approximate the optimal algorithm. We have several options. For example, we might consider temporal locality. We might guess that any value that has not been used recently is unlikely to be needed again soon. We can keep track of the last time each block was accessed (assign a timestamp to the block) and select as the victim block the block that has been used least recently. This is the **least recently used** (**LRU**) algorithm. Unfortunately, LRU requires the system to keep a history of accesses for every cache block, which requires significant space and slows down the operation of the cache. There are ways to approximate LRU, but that is beyond the scope of this text. (Refer to the references at the end of the chapter for more information.)

First-in, first-out (**FIFO**) is another popular approach. With this algorithm, the block that has been in cache the longest (regardless of how recently it has been used) would be selected as the victim to be removed from cache memory.

Another approach is to select a victim at **random**. The problem with LRU and FIFO is that there are degenerate referencing situations in which they can be made to **thrash** (constantly throw out a block, then bring it back, then throw it out, then bring it back, repeatedly). Some people argue that random replacement, although it sometimes throws out data that will be needed soon, never thrashes.

Unfortunately, it is difficult to have truly random replacement, and it can decrease average performance.

The algorithm selected often depends on how the system will be used. No single (practical) algorithm is best for all scenarios. For that reason, designers use algorithms that perform well under a wide variety of circumstances.

6.4.3 Effective Access Time and Hit Ratio

The performance of a hierarchical memory is measured by its **effective access time** (**EAT**), or the average time per access. EAT is a weighted average that uses the hit ratio and the relative access times of the successive levels of the hierarchy. The actual access time for each level depends on the technology and method used for access.

Before we can determine the average access time, we have to know something about how cache and memory work. When data is needed from cache, there are two options for retrieving that data. We could start an access to cache and, at the same time, start an access to main memory (in parallel). If the data is found in cache, the access to main memory is terminated, at no real cost because the accesses were overlapped. If the data is not in cache, the access to main memory is already well on its way. This overlapping helps reduce the cost (in time) for a cache miss. The alternative is to perform everything sequentially. First, cache is checked; if the data is found, we're done. If the data is not found in cache, then an access is started to retrieve the data from main memory. The method used affects the average (effective) access time, as we see below.

For example, suppose the cache access time is 10ns, main memory access time is 200ns, and the cache hit rate is 99%. Using overlapped (parallel) access, the average time for the processor to access an item in this two-level memory would then be:

$$\text{EAT} = \underbrace{0.99(10\text{ns})}_{\text{cache hit}} + \underbrace{0.01(200\text{ns})}_{\text{cache miss}} = 9.9\text{ns} + 2\text{ns} = 11\text{ns}$$

Using nonoverlapped (sequential) access, the average access time would change to:

$$\text{EAT} = 0.99(10\text{ns}) + 0.01(\overbrace{10\text{ns}}^{\text{check cache}} + \overbrace{200\text{ns}}^{\text{go to main memory}}) = 9.9\text{ns} + 2.1\text{ns}$$

What, exactly, does this mean? If we look at the access times over a long period of time, this system performs as if it had a single large memory with an 11ns or 12ns access time. A 99% cache hit rate allows the system to perform very well, even though most of the memory is built using slower technology with an access time of 200ns.

The formula for calculating effective access time for a two-level memory consisting of cache and main memory is given by:

$$\text{EAT} = H \times \text{Access}_C + (1 - H) \times \text{Access}_{MM}$$

where H = cache hit rate, Access_C = cache access time, and Access_{MM} = main memory access time.

This formula can be extended to apply to three- or even four-level memories, as we will see shortly.

6.4.4 When Does Caching Break Down?

When programs exhibit locality, caching works quite well. However, if programs exhibit bad locality, caching breaks down and the performance of the memory hierarchy is poor. In particular, object-oriented programming can cause programs to exhibit less than optimal locality. Another example of bad locality can be seen in two-dimensional array access. Arrays are typically stored in row-major order. Suppose, for purposes of this example, that one row fits exactly in one cache block and cache can hold all but one row of the array. Assume, also, that cache uses a first-in, first-out replacement policy. If a program accesses the array one row at a time, the first row access produces a miss, but once the block is transferred into cache, all subsequent accesses to that row are hits. So a 5 × 4 array would produce 5 misses and 15 hits over 20 accesses (assuming that we are accessing each element of the array). If a program accesses the array in column-major order, the first access to the column, entry (1,1), results in a miss, after which an entire row is transferred in. However, the second access to the column, entry (2,1), results in another miss. The access to entry (3,1) produces a miss, as does the access to entry (4,1). Reading entry (5,1) produces a miss, but bringing in the fifth row requires us to throw out the first row. Because we have removed the first row, our next access to entry (1,2) causes a miss as well. This continues throughout the entire array access; the data being transferred in for each row is not being used because the array is being accessed by column. Because cache is not large enough, this would produce 20 misses on 20 accesses. A third example would be a program that loops through a linear array that does not fit in cache. There would be a significant reduction in the locality when memory is used in this manner.

6.4.5 Cache Write Policies

In addition to determining which victim to select for replacement, designers must also decide what to do with so-called **dirty blocks** of cache, or blocks that have been modified. When the processor writes to main memory, the data may be written to the cache instead under the assumption that the processor will probably read it again soon. If a cache block is modified, the cache **write policy** determines when the actual main memory block is updated to match the cache block. There are two basic write policies:

- **Write-through**—A write-through policy updates both the cache and the main memory simultaneously on every write. This is slower than write-back, but ensures that the cache is consistent with the main system memory. The obvious disadvantage here is that every write now requires a main memory access. Using a write-through policy means every write to the cache necessitates a main memory write, thus slowing the system (if all accesses are write, this essentially slows down the system to main memory speed). However, in real applications, the majority of accesses are reads, so this slow-down is negligible.
- **Write-back**—A write-back policy (also called **copyback**) only updates blocks in main memory when the cache block is selected as a victim and must be removed from cache. This is normally faster than write-through because time is not wasted writing information to memory on each write to cache. Memory traffic is also reduced. The disadvantage is that main memory and cache may not contain the same value at a given instant of time, and if a process terminates (crashes) before the write to main memory is done, the data in cache may be lost.

To improve the performance of cache, one must increase the hit ratio by using a better mapping algorithm (up to roughly a 20% increase), better strategies for write operations (potentially a 15% increase), better replacement algorithms (up to a 10% increase), and better coding practices, as we saw in the earlier example of row versus column-major access (up to a 30% increase in hit ratio). Simply increasing the size of cache may improve the hit ratio by roughly 1–4% but is not guaranteed to do so.

Caching and Hashing and Bits, Oh My!

Suppose you have a large set of data, and you are writing a program to search that data set. You could store the data in an array, at which point you have two options. If the array is unsorted, a linear search could be used to locate a particular piece of information. If the data is sorted, a binary search could be used. However, the running times of both types of searches are dependent on the data set size with which you are working.

Hashing is a process that allows us to store and retrieve data in an average time that does not depend on the size of the data set we are searching. Hashing is simply the process of putting an item into a structure by transforming a key value into an address. Typically, a **hash table** is used to store the hashed values, and a **hashing function** performs the key-to-address transformation. When we want to look up a value, the key for the value is input into the hash function, and the output is the address in the table at which the value is stored. Hash tables were invented by compiler writers in the 1960s to maintain symbol tables (as discussed in Chapter 4).

Suppose you are tired of looking up names in an extremely large phone book. Even though the names are sorted, with so many pages, it takes you a long time to find a particular phone number. You could create a hash function to store those names and numbers in a hash table on your computer. When you want a phone number, all you would need to do is input the name into the hash function and it would point you to the correct location of the corresponding phone number in the hash table. Computerized dictionaries often use hash tables for storing words and definitions because the lookup time is much faster than it would be with binary search.

Hashing works because of good hash functions—good hash functions give a uniform distribution even if the key values are not distributed well. A perfect hash function is a one-to-one mapping of the values into the table. However, perfect hash functions are difficult to construct. Hash functions that map most values to unique locations are acceptable, provided that the number of **collisions** (cases in which two values map to the same location) are minimal. Collisions are handled in several different ways, the easiest of which is chaining. **Chaining** is simply the process of creating a list of items that map to a particular location. When a key value maps to a list, more time is required to find the item because we have to search the entire list.

Hash functions can be simple or complex. Examples of simple hash functions with numeric keys include (1) modulo arithmetic, in which the number of items in the table is estimated, that number is used as a divisor into the key, and the remainder is the hashed value; (2) radix transformation, a process in which the digital key's number base is changed, resulting in a different sequence of digits, and a specific "substring" of those digits is used as the hash value; (3) key transposition, in which the digits in the portion of the key are simply scrambled, and the new scrambled value is used as the hash value; and (4) folding, a method in which the key is portioned into several parts, the parts are added together, and a portion of the result is used as the hash value. For the above phone book example, we could substitute a character's ASCII value for each letter in the name, and use a radix transformation by selecting some base, multiplying each ASCII value by a power of that base, and adding the results to yield the hash value. For example, if the name is "Tim" and the base is 8, the hash value would be $(84 \times 8^2) + (105 \times 8^1) + (109 \times 8^0) = 6325$, meaning we could find Tim in location 6325 of our hash table.

Hashing is used in many computer science applications. For example, web browsers use a hashed cache to store recently viewed web pages. Hashing is used to store and retrieve items in indexed database systems. There are several well-known hash functions used in cryptography, message authentication, and error checking. In large file systems, both caching and hashing (to determine file location) are used to increase performance.

You should notice a similarity between hashing and the cache mapping schemes discussed in this chapter. The “key” used for cache mapping is the middle field of the address, which is the block field for direct mapped cache and the set field for set associative cache. The hash function used by direct mapping and set associative mapping is simple modulo arithmetic (fully associative mapping uses no hashing and requires a full search of the cache to find the desired data element). In direct mapping, if we have a collision, the value currently in cache is replaced by the incoming value. With set associative cache, the set is similar to a list in chaining—if there is a collision, depending on the size of the set, multiple values that hash to the same location can coexist. To retrieve a value in set associative cache, once the middle bits determine the location (or set) in cache, the set must be searched for the desired value. Recall that a tag is stored with each value and is used to identify the correct item.

Different items can be used as keys for hashing. We use the middle bits of the physical address of a desired value as the key in cache mapping. But why use the middle bits? The lower bits (in the rightmost word field) are used to determine the offset within the block. However, the higher bits (in the leftmost tag field) could be used as the key instead of the middle bits (and the middle bits could, correspondingly, be used as the tag). Why did designers choose the middle bits?

There are two options for memory interleaving: high-order interleaving and low-order memory interleaving. High-order interleaving uses the high-order bits to determine the memory module location and places consecutive memory addresses in the same module. This is precisely what would happen if we used the high-order bits of the memory address to determine the cache location—values from consecutive memory addresses would map to the same location in cache. Spatial locality tells us that memory locations close to each other tend to be referenced in clusters. If values from adjacent memory locations are mapped to the same cache block or set, spatial locality implies that there will be collisions (larger sets result in fewer collisions). However, if the middle bits are used as the key instead, adjacent memory locations are mapped to different cache locations, resulting in fewer collisions, and, ultimately, better hit ratios.

6.4.6 Instruction and Data Caches

In our discussion of cache, we have focused mainly on what is called a **unified** or **integrated cache**, that is, cache that holds the more recently used data *and* instructions. However, many modern computers employ a **Harvard cache**, which means they have separate data and instruction caches.

A **data cache** is reserved for storing data only. When a request is made for data from memory, this cache is checked first. If the data is found, it is retrieved from data cache and the execution continues. If not found, the data is retrieved from memory and placed in the data cache. Data that exhibits good locality yields higher hit rates. For example, data accessed from an array exhibits better locality than data accessed from a linked list. Thus, the way in which you program can affect the data cache hit rate.

Just as high data cache hit rates are desirable, high **instruction cache** hit rates are imperative for good performance. Most program instructions are executed sequentially, branching only occasionally for procedure calls, loops, and conditional statements. Therefore, program instructions tend to have high locality. Even when a procedure is called or a loop is executed, these modules are often small enough to fit in one block of an instruction cache (a good motivator for small procedures!), thus increasing performance.

Separating instructions from data in cache allows the accesses to be less random and more clustered. However, some designers prefer an integrated cache; to get the same performance as a system with separate data and instruction caches, the integrated cache must be larger, thus introducing more latency in retrieving values. In addition, a unified cache has only one port for data and instructions, resulting in conflicts between the two.

Some processors also have what is known as a **victim cache**, a small, associative cache used to hold blocks that have been thrown out of the CPU cache due to conflicts. The idea is that if the victim block is used in the near future, it can be retrieved from the victim cache with less penalty than going to memory. This basically gives the data that was evicted from cache a “second chance” to be used before being sent to memory. Another type of specialized cache, **trace cache**, is a variant of an instruction cache. Trace caches are used to store instruction sequences that have already been decoded, so that if the instructions are needed again, no decoding is required. In addition, the processor can deal with blocks of instructions and not worry about branches in the execution flow of the program. This cache is so named because it stores traces of the dynamic instruction stream, making noncontiguous instructions appear to be contiguous. The Intel Pentium 4 uses this type of cache to increase performance.

6.4.7 Levels of Cache

Clearly, caching provides an increase in performance, provided the hit rate is satisfactory. Increasing the size of cache might be your first thought to produce a higher hit rate; however, larger caches are slower (the access time is larger). For years, manufacturers have been trying to balance the higher latency in larger caches with the increase in hit rates by selecting just the right size cache. However, many designers are applying the concepts of levels of memory to cache itself, and they are now using a **multilevel cache hierarchy** in most systems—caches are using caching for increased performance!

As we have seen, Level 1 (L1) cache is the term used for cache that is resident on the chip itself, and it is the fastest, smallest cache. L1 cache, often referred to as “internal cache,” typically ranges in size from 8KB to 64KB. When a memory access is requested, L1 is checked first, with typical access speeds of approximately 4ns. If the data is not found in L1, the Level 2 (L2) cache is checked. The L2 cache is typically located external to the processor, and it has access speeds of 15–20ns. L2 cache can be located on the system motherboard, on a separate chip, or on an expansion board. L2 cache is larger, but slower than L1 cache. Typical sizes for L2 cache range from 64KB to 2MB. If data is missing from L1, but found in L2, it is loaded into L1 cache (in some architectures, the data being replaced in L1 is swapped with the requested data in L2). More manufacturers have begun to include L2 cache in their architectures by building L2 cache onto the same die as the CPU itself so the cache runs at CPU speed (but not on the same circuit with the CPU, such as is the case with L1 cache). This reduces the access speed of L2 cache to approximately 10ns. **L3 cache** is the term now used to refer to the extra cache between the processor and memory (that used to be called L2 cache) on processors that include L2 cache as part of their architecture. L3 caches range in size from 256MB to 8MB.

Inclusive caches are caches at different levels that allow the data to be present at multiple levels concurrently. For example, in the Intel Pentium family, data found in L1 may also exist in L2. A **strictly inclusive cache** guarantees that all data at one level is also found in the next lower level. **Exclusive**

caches guarantee the data to be in, at most, one level of cache.

We discussed separate data and instruction caches in the previous section. Typically, architectures use integrated caches at levels L2 and L3. However, many have separate instruction and data caches at L1. For example, the Intel Celeron uses two separate L1 caches, one for instructions and one for data. In addition to this nice design feature, Intel uses **nonblocking cache** for its L2 caches. Most caches can process only one request at a time (so on a cache miss, the cache has to wait for memory to provide the requested value and load it into cache). Nonblocking caches can process up to four requests concurrently.

6.5 VIRTUAL MEMORY

You now know that caching allows a computer to access frequently used data from a smaller but faster cache memory. Cache is found near the top of our memory hierarchy. Another important concept inherent in the hierarchy is **virtual memory**. The purpose of virtual memory is to use the hard disk as an extension of RAM, thus increasing the available address space a process can use. Most personal computers have a relatively small amount (typically around 4GB) of main memory. This is usually not enough memory to hold multiple applications concurrently, such as a word processing application, an email program, and a graphics program, in addition to the operating system itself. Using virtual memory, your computer addresses more main memory than it actually has, and it uses the hard drive to hold the excess. This area on the hard drive is called a **page file**, because it holds chunks of main memory on the hard drive. The easiest way to think about virtual memory is to conceptualize it as an imaginary memory location in which all addressing issues are handled by the operating system.

The most common way to implement virtual memory is by using **paging**, a method in which main memory is divided into fixed-size blocks and programs are divided into the same size blocks. Typically, chunks of the program are brought into memory as needed. It is not necessary to store contiguous chunks of the program in contiguous chunks of main memory. Because pieces of the program can be stored out of order, program addresses, once generated by the CPU, must be translated to main memory addresses. Remember, in caching, a main memory address had to be transformed into a cache location. The same is true when using virtual memory; every virtual address must be translated into a physical address. How is this done? Before delving further into an explanation of virtual memory, let’s define some frequently used terms for virtual memory implemented through paging:

- **Virtual address**—The logical or program address that the process uses. Whenever the CPU generates an address, it is always in terms of virtual address space.
- **Physical address**—The real address in physical memory.
- **Mapping**—The mechanism by which virtual addresses are translated into physical ones (very similar to cache mapping)
- **Page frames**—The equal-size chunks or blocks into which main memory (physical memory) is divided.
- **Pages**—The chunks or blocks into which virtual memory (the logical address space) is divided, each equal in size to a page frame. Virtual pages are stored on disk until needed.
- **Paging**—The process of copying a virtual page from disk to a page frame in main memory.
- **Fragmentation**—Memory that becomes unusable.
- **Page fault**—An event that occurs when a requested page is not in main memory and must be copied

into memory from disk.

Because main memory and virtual memory are divided into equal-size pages, pieces of the process address space can be moved into main memory but need not be stored contiguously. As previously stated, we need not have all of the process in main memory at once; virtual memory allows a program to run when only specific pieces are present in memory. The parts not currently being used are stored in the page file on disk.

Virtual memory can be implemented with different techniques, including paging, segmentation, or a combination of both, but paging is the most popular. (This topic is covered in great detail in the study of operating systems.) The success of paging, like that of cache, is dependent on the locality principle. When data is needed that does not reside in main memory, the entire block in which it resides is copied from disk to main memory, in hopes that other data on the same page will be useful as the program continues to execute.

6.5.1 Paging

The basic idea behind paging is quite simple: Allocate physical memory to processes in fixed-size chunks (page frames) and keep track of where the various pages of the process reside by recording information in a **page table**. Every process has its own page table that typically resides in main memory, and the page table stores the physical location of each virtual page of the process. The page table has *N* rows, where *N* is the number of virtual pages in the process. If there are pages of the process currently not in main memory, the page table indicates this by setting a **valid bit** to 0; if the page is in main memory, the valid bit is set to 1. Therefore, each entry of the page table has two fields: a valid bit and a frame number.

Additional fields are often added to relay more information. For example, a **dirty bit** (or a **modify bit**) could be added to indicate whether the page has been changed. This makes returning the page to disk more efficient, because if it is not modified, it does not need to be rewritten to disk. Another bit (the **usage bit**) can be added to indicate the page usage. This bit is set to 1 whenever the page is accessed. After a certain time period, the usage bit is set to 0. If the page is referenced again, the usage bit is set to 1. However, if the bit remains 0, this indicates that the page has not been used for a period of time, and the system might benefit by sending this page out to disk. By doing so, the system frees up this page's location for another page that the process eventually needs (we discuss this in more detail when we introduce replacement algorithms).

Virtual memory pages are the same size as physical memory page frames. Process memory is divided into these fixed-size pages, resulting in potential **internal fragmentation** when the last page is copied into memory. The process may not actually need the entire page frame, but no other process may use it. Therefore, the unused memory in this last frame is effectively wasted. It might happen that the process itself requires less than one page in its entirety, but it must occupy an entire page frame when copied to memory. Internal fragmentation is unusable space *within* a given partition (in this case, a page) of memory.

Now that you understand what paging is, we will discuss how it works. When a process generates a virtual address, the operating system must dynamically translate this virtual address into the physical address in memory at which the data actually resides. (For purposes of simplicity, let's assume that we have no cache memory for the moment.) For example, from a program viewpoint, we see the final byte of a 10-byte program as address 0x09, assuming 1-byte instructions and 1-byte addresses, and a starting address of 0x00. However, when actually loaded into memory, the logical address 0x09 (perhaps a reference to the label *X* in an assembly language program) may actually reside in physical memory location 0x39, implying that the program was loaded starting at physical address 0x30. There must be an easy way to convert the logical, or virtual, address 0x09 to the physical address 0x39.

To accomplish this address translation, a virtual address is divided into two fields: a **page field** and an **offset field**, to represent the offset within that page where the requested data is located. This address translation process is similar to the process we used when we divided main memory addresses into fields for the cache mapping algorithms. And similar to cache blocks, page sizes are usually powers of 2; this simplifies the extraction of page numbers and offsets from virtual addresses.

To access data at a given virtual address, the system performs the following steps:

1. Extract the page number from the virtual address.

2. Extract the offset from the virtual address.

3. Translate the page number into a physical page frame number by accessing the page table.

  A. Look up the page number in the page table (using the virtual page number as an index).

  B. Check the valid bit for that page.

    1. If the valid bit = 0, the system generates a page fault and the operating system must intervene to

      a. Locate the desired page on disk.

      b. Find a free page frame (this may necessitate removing a “victim” page from memory and copying it back to disk if memory is full).

      c. Copy the desired page into the free page frame in main memory.

      d. Update the page table. (The virtual page just brought in must have its frame number and valid bit in the page table modified. If there was a “victim” page, its valid bit must be set to zero.)

      e. Resume execution of the process causing the page fault, continuing to Step B2.

    2. If the valid bit = 1, the page is in memory.

      a. Replace the virtual page number with the actual frame number.

      b. Access data at offset in physical page frame by adding the offset to the frame number for the given virtual page.

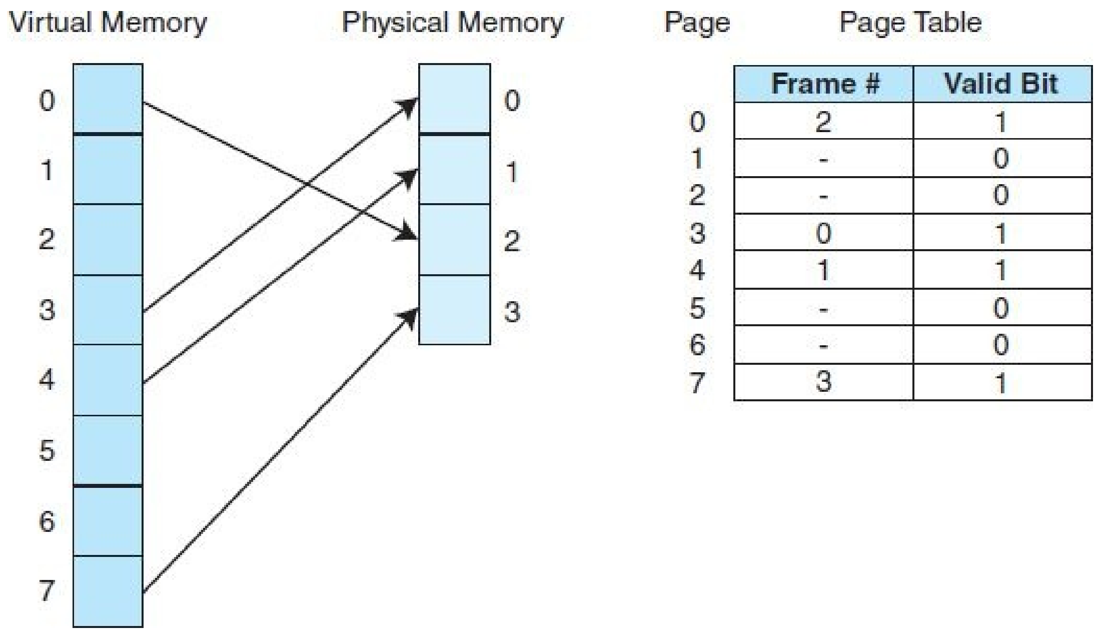
Please note that if a process has free frames in main memory when a page fault occurs, the newly retrieved page can be placed in any of those free frames. However, if the memory allocated to the process is full, a victim page must be selected. The replacement algorithms used to select a victim are quite similar to those used in cache. FIFO, random, and LRU are all potential replacement algorithms for selecting a victim page. (For more information on replacement algorithms, see the references at the end of this chapter.)

Let’s look at an example.

EXAMPLE 6.8 Suppose that we have a virtual address space of 2^8 bytes for a given process (this means the program generates addresses in the range 0x00 to 0xFF, which is 0 to 255 in base 10), and physical memory of 4 page frames (no cache). Assume also that pages are 32 bytes in length. Virtual addresses contain 8 bits, and physical addresses contain 7 bits (4 frames of 32 bytes each is 128 bytes, or 2^7). Suppose, also, that some pages from the process have been brought into main memory. Figure 6.18 illustrates the current state of the system.

Each virtual address has 8 bits and is divided into 2 fields: the page field has 3 bits, indicating that there are 2^3 pages of virtual memory, $\left(\frac{2^8}{2^5}\right) = 2^3$. Each page is $2^5 = 32$ bytes in length, so we need 5 bits for the page offset (assuming a byte-addressable memory). Therefore, an 8-bit virtual address has the format shown in Figure 6.19.

Suppose the system now generates the virtual address 0x0D = 00001101_2. Dividing the binary address into the page and offset fields (see Figure 6.20), we see the page field P = 000_2 and the offset field equals 01101_2. To continue the translation process, we use the 000 value of the page field as an index into the page table. Going to the 0th entry in the page table, we see that virtual page 0 maps to physical page frame 2 = 10_2. Thus the translated physical address becomes page frame 2, offset 13. Note that a physical address has only 7 bits (2 for the frame, because there are 4 frames, and 5 for the offset). Written in binary, using the two fields, this becomes 1001101_2, or address 0x4D, and is shown in Figure 6.21.



Page	Frame #	Valid Bit
0	2	1
1	-	0
2	-	0
3	0	1
4	1	1
5	-	0
6	-	0
7	3	1

FIGURE 6.18 Current State Using Paging and the Associated Page Table

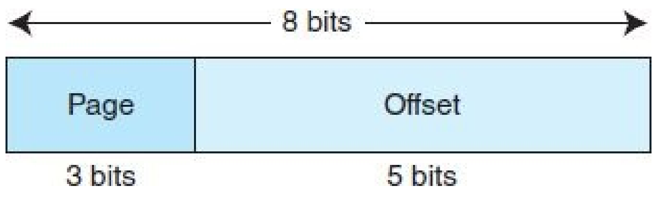


FIGURE 6.19 Format for an 8-Bit Virtual Address with 2^5 = 32-Byte Page Size

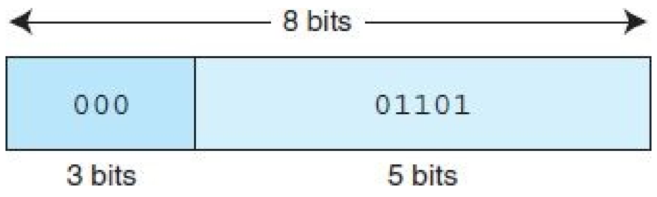


FIGURE 6.20 Format for Virtual Address 00001101_2 = 0x0D

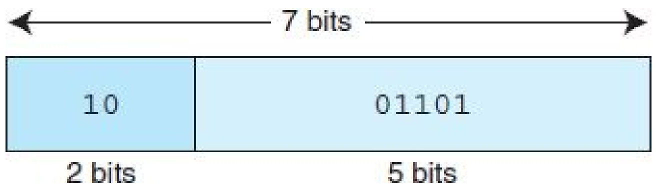


FIGURE 6.21 Format for Physical Address $1001101_2 = 4D_{16}$

Let’s look at a complete example in a real (but small) system (again, with no cache).

EXAMPLE 6.9 Suppose a program is 16 bytes long, has access to an 8-byte memory that uses byte addressing, and a page is 2 bytes in length. As the program executes, it generates the following address reference string: 0x0, 0x1, 0x2, 0x3, 0x6, 0x7, 0xA, 0xB. (This address reference string indicates that address 0x0 is referenced first, then address 0x1, then address 0x2, and so on.) Originally, memory contains no pages for this program. When address 0x0 is needed, both address 0x0 and address 0x1 (in page 0) are copied to page frame 2 in main memory. (It could be that frames 0 and 1 of memory are occupied by another process and thus unavailable.) This is an example of a page fault, because the desired page of the program had to be fetched from disk. When address 0x1 is referenced, the data already exists in memory (so we have a page hit). When address 0x2 is referenced, this causes another page fault, and page 1 of the program is copied to frame 0 in memory. This continues, and after these addresses are referenced and pages are copied from disk to main memory, the state of the system is as shown in Figure 6.22a. We see that address 0x0 of the program, which contains the data value “A,” currently resides in memory address $0x4 = 100_2$. Therefore, the CPU must translate from virtual address 0x0 to physical address 0x4, and uses the translation scheme described previously to do this. Note that main memory addresses contain 3 bits (there are 8 bytes in memory), but virtual addresses (from the program) must have 4 bits (because there are 16 bytes in the virtual address). Therefore, the translation must also convert a 4-bit address into a 3-bit address.

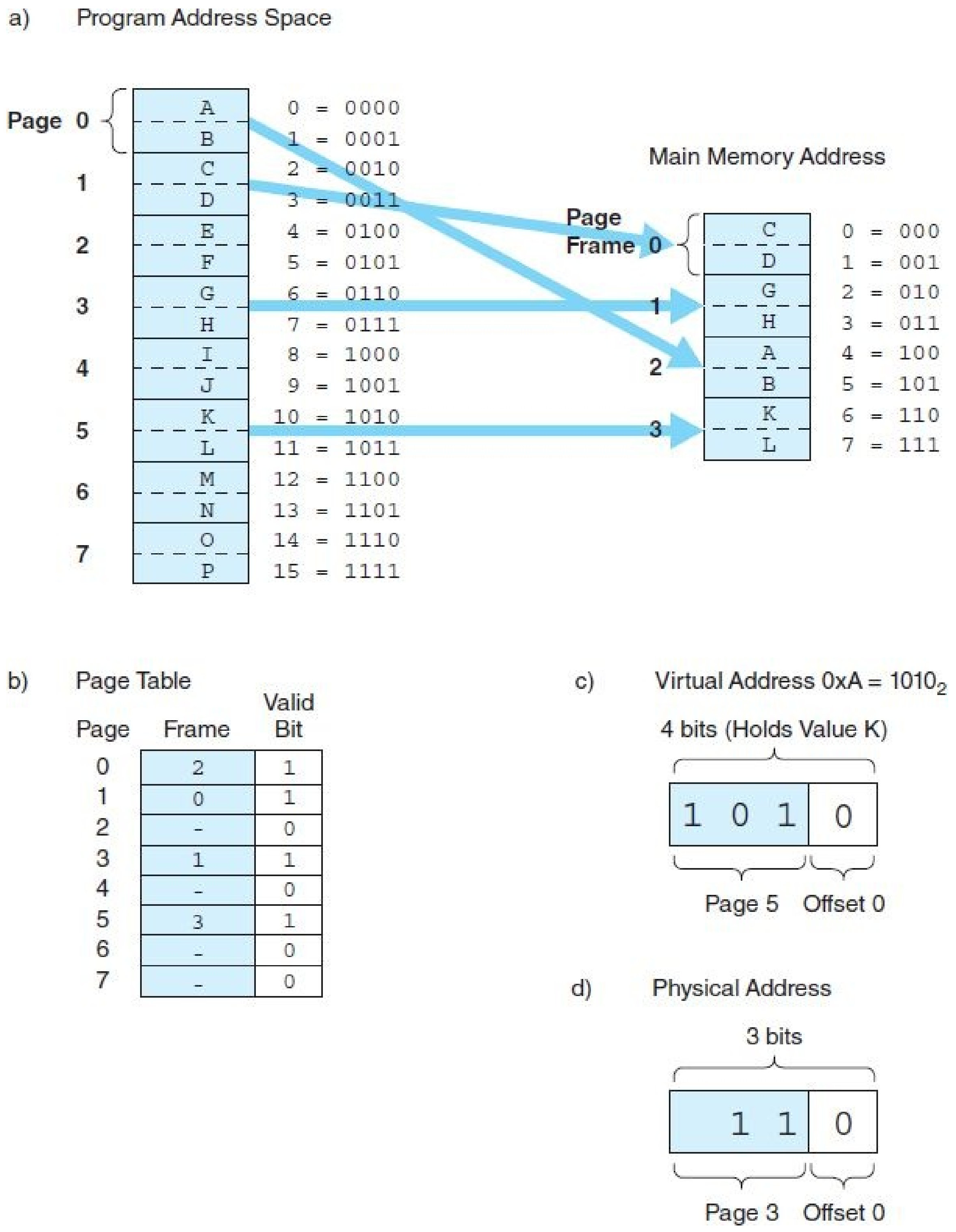


FIGURE 6.22 A Small Memory from Example 6.9 Using Paging

Figure 6.22b depicts the page table for this process after the given pages have been accessed. We can see that pages 0, 1, 3, and 5 of the process are valid, and thus reside in memory. Pages 2, 6, and 7 are not valid and would each cause page faults if referenced.

Let's take a closer look at the translation process. Using the architecture from Example 6.9, suppose the CPU now generates program, or virtual, address 0xA = 1010_2 for a second time. We see in Figure 6.22a that the data at this location, "K," resides in main memory address 0x6 = 0110_2. However, the computer must perform a specific translation process to find the data. To accomplish this, the virtual address, 1010_2, is divided into a page field and an offset field. The page field is 3 bits long because there are 8 pages in the program. This leaves 1 bit for the offset, which is correct because there are only 2 words on each page. This field division is illustrated in Figure 6.22c.

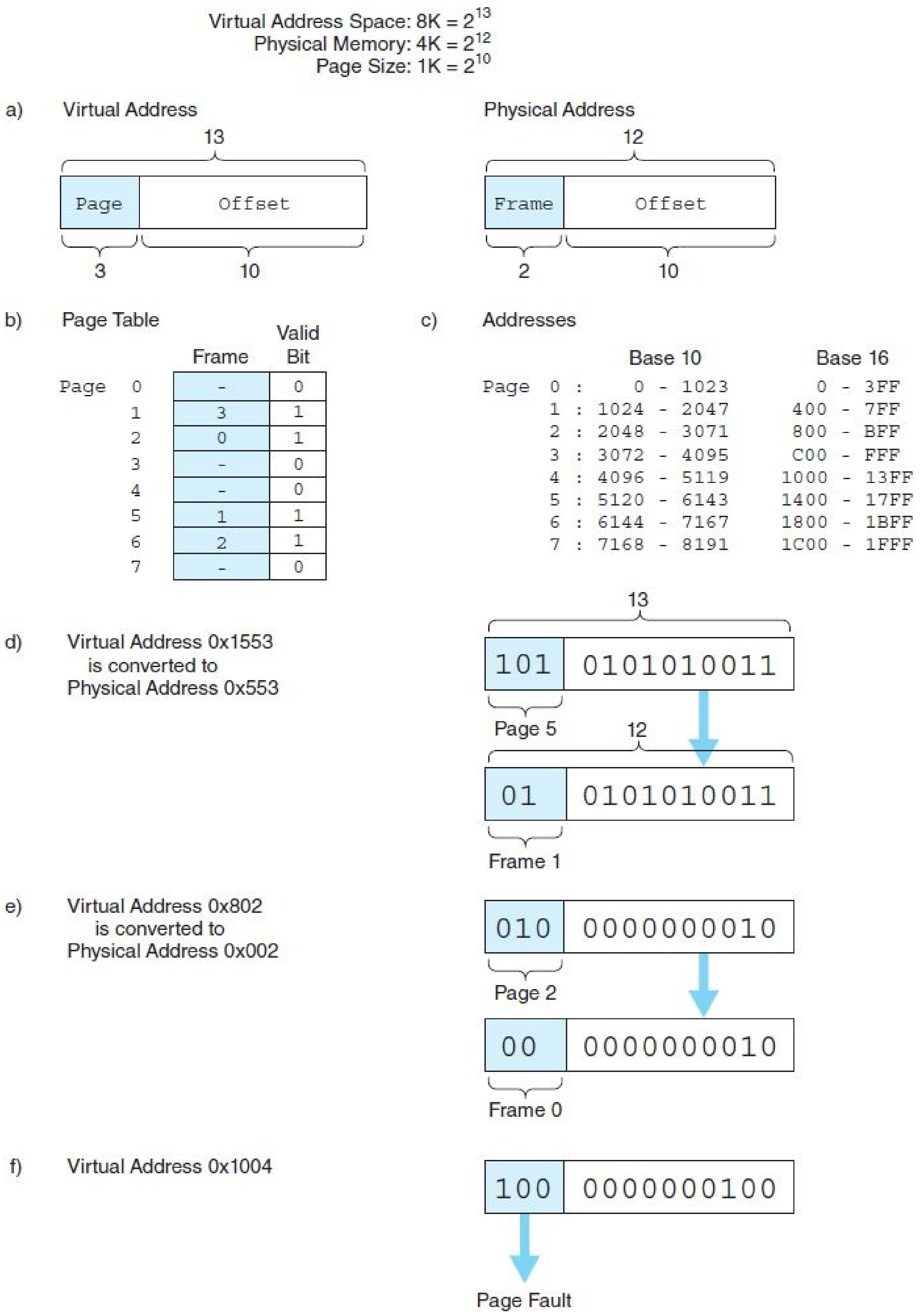
Once the computer sees these fields, it is a simple matter to convert to the physical address. The page field value of 101_2 is used as an index into the page table. Because $101_2 = 5$, we use 5 as the offset into the page table (Figure 6.22b) and see that virtual page 5 maps to physical frame 3. We now replace the $5 = 101_2$ with $3 = 11_2$, but keep the same offset. The new physical address is 110_2, as shown in Figure 6.22d. This process successfully translates from virtual addresses to physical addresses and reduces the number of bits from four to three as required.

Now that we have worked with a small example, we are ready for a larger, more realistic example.

EXAMPLE 6.10 Suppose we have a virtual address space of 8K bytes, a physical memory size of 4K bytes that uses byte addressing, and a page size of 1K bytes. (There is no cache on this system either, but we are getting closer to understanding how memory works and eventually will use both paging and cache in our examples.) A virtual address has a total of 13 bits ($8K = 2^{13}$), with 3 bits used for the page field (there are $\frac{2^{13}}{2^{10}} = 2^3$ virtual pages), and 10 used for the offset (each page has 2^{10} bytes). A physical memory address has only 12 bits ($4K = 2^{12}$), with the first 2 bits as the page field (there are 2^2 page frames in main memory) and the remaining 10 bits as the offset within the page. The formats for the virtual address and physical address are shown in Figure 6.23a.

For purposes of this example, let's assume we have the page table indicated in Figure 6.23b. Figure 6.23c shows a table indicating the various main memory addresses that is useful for illustrating the translation.

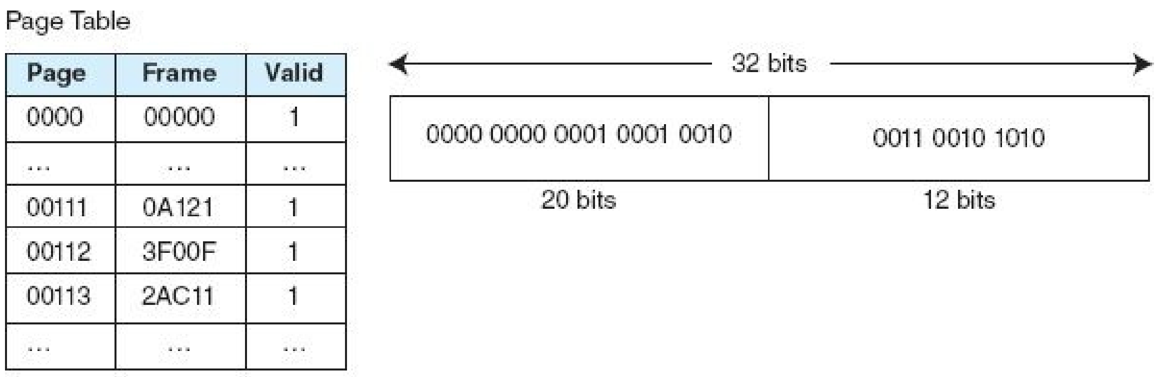
Suppose the CPU now generates virtual address $0x1553 = 1010101010011_2$. Figure 6.23d illustrates how this address is divided into the page and offset fields and how it is converted to the physical address $0x553 = 010101010011_2$. Essentially, the page field 101 of the virtual address is replaced by the frame number 01, because page 5 maps to frame 1 (as indicated in the page table). Figure 6.23e illustrates how virtual address 0x802 is translated to physical address 0x002. Figure 6.23f shows virtual address 0x1004 generating a page fault; page $4 = 100_2$ is not valid in the page table.


Virtual Address Space: 8K = 2^{13}
Physical Memory: 4K = 2^{12}
Page Size: 1K = 2^{10}
a) Virtual Address
13
Page
Offset
3
10
Physical Address
12
Frame
Offset
2
10
b) Page Table
c) Addresses
d) Virtual Address 0x1553 is converted to Physical Address 0x553
13
101
0101010011
Page 5
12
01
0101010011
Frame 1
e) Virtual Address 0x802 is converted to Physical Address 0x002
010
0000000010
Page 2
00
0000000010
Frame 0
f) Virtual Address 0x1004
100
0000000100
Page Fault

Page	Frame	Valid Bit
0	-	0
1	3	1
2	0	1
3	-	0
4	-	0
5	1	1
6	2	1
7	-	0

Page	Base 10	Base 16
0 :	0 - 1023	0 - 3FF
1 :	1024 - 2047	400 - 7FF
2 :	2048 - 3071	800 - BFF
3 :	3072 - 4095	C00 - FFF
4 :	4096 - 5119	1000 - 13FF
5 :	5120 - 6143	1400 - 17FF
6 :	6144 - 7167	1800 - 1BFF
7 :	7168 - 8191	1C00 - 1FFF

FIGURE 6.23 A Larger Memory Example Using Paging


Page Table

Page	Frame	Valid
0000	00000	1
...	...	...
00111	0A121	1
00112	3F00F	1
00113	2AC11	1
...	...	...

FIGURE 6.24 Format for Virtual Address 0x1211232A

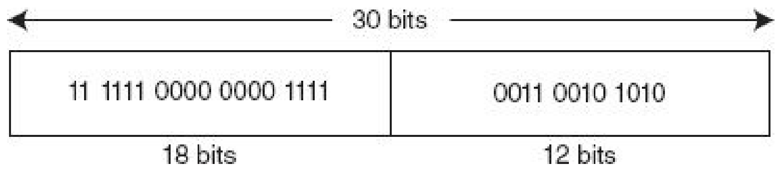


FIGURE 6.25 Format for Physical Address 0x3F00F32A

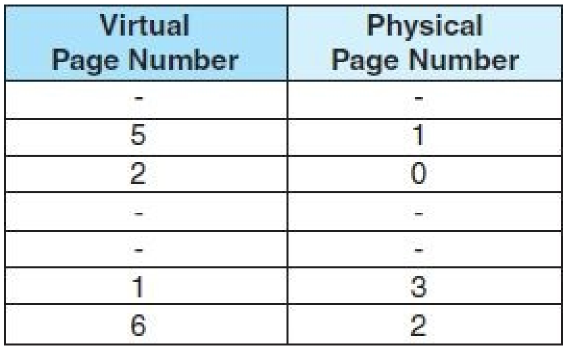
EXAMPLE 6.11 Suppose a computer has 32-bit virtual addresses, pages of size 4K, and 1GB of byte-addressable main memory. Given the partial page table above, convert virtual address 0x0011232A to a physical address. The page number and frame number values for this example are in hexadecimal.

The format for virtual memory address 0x0011232A is shown in Figure 6.24. There are 32 bits in a virtual address, and 12 of those must refer to the offset within a page (because pages are of size 4K = 2^{12}). The remaining 20 bits are used to determine the virtual page number: $0000\ 0000\ 0001\ 0001\ 0010_2$ = 0x00112. The format for a physical address is shown in Figure 6.25. The offset field must still contain 12 bits, but a physical address only has 30 bits (1GB = 2^{30}), so the frame field has only 18 bits. If we check the page table for virtual page 00112, we see that it is valid and mapped to frame number 3F00F. If we substitute 3F00F into the frame field, we end up with the physical address of $11\ 1111\ 0000\ 0000\ 1111_2$ = 0x3F00F32A, as shown in Figure 6.25.

It is worth mentioning that selecting an appropriate page size is very difficult. The larger the page size is, the smaller the page table is, thus saving space in main memory. However, if the page is too large, the internal fragmentation becomes worse. Larger page sizes also mean fewer actual transfers from disk to main memory because the chunks being transferred are larger. However, if they are too large, the principle of locality begins to break down and we are wasting resources by transferring data that may not be necessary.

6.5.2 Effective Access Time Using Paging

When we studied cache, we introduced the notion of effective access time. We also need to address EAT while using virtual memory. There is a time penalty associated with virtual memory: For each memory access that the processor generates, there must now be *two* physical memory accesses—one to reference the page table and one to reference the actual data we wish to access. It is easy to see how this affects the effective access time. Suppose a main memory access requires 200ns and the page fault rate is 1% (that is, 99% of the time we find the page we need in memory). Assume that it costs us 10ms to access a page not in memory. (This time of 10ms includes the time necessary to transfer the page into memory, update the page table, and access the data.) The effective access time for a memory access is now:



Virtual Page Number	Physical Page Number
-	-
5	1
2	0
-	-
-	-
1	3
6	2

FIGURE 6.26 Current State of the TLB for Figure 6.23

$$\text{EAT} = .99(200\text{ns} + 200\text{ns}) + .01(10\text{ms}) = 100.396\text{ns}$$

Even if 100% of the pages were in main memory, the effective access time would be:

$$\text{EAT} = 1.00(200\text{ns} + 200\text{ns}) = 400\text{ns},$$

which is double the access time of memory. Accessing the page table costs us an additional memory access because the page table itself is stored in main memory.

We can speed up the page table lookup by storing the most recent page lookup values in a page table cache called a **translation look-aside buffer** (**TLB**). Each TLB entry consists of a virtual page number and its corresponding frame number. A possible state of the TLB for the page table from Example 6.10 is indicated in Figure 6.26.

Typically, the TLB is implemented as associative cache, and the virtual page/frame pairs can be mapped anywhere. Here are the steps necessary for an address lookup when using a TLB (see Figure 6.27):

1. Extract the page number from the virtual address.
2. Extract the offset from the virtual address.
3. Search for the virtual page number in the TLB using parallel searching.
4. If the (virtual page #, page frame #) pair is found in the TLB, add the offset to the physical frame number and access the memory location.
5. If there is a TLB miss, go to the page table to get the necessary frame number. If the page is in memory, use the corresponding frame number and add the offset to yield the physical address.

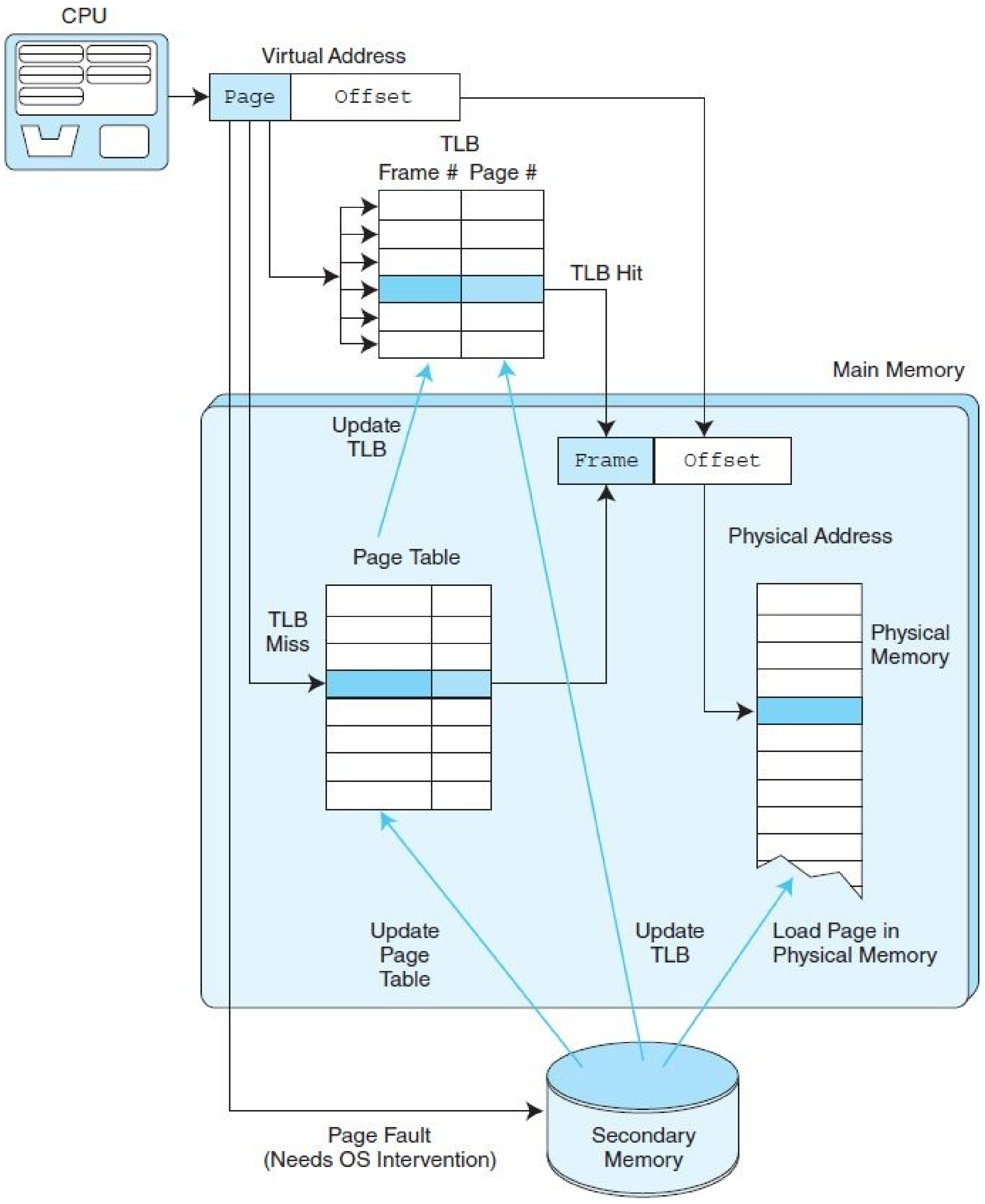


FIGURE 6.27 Using the TLB

6. If the page is not in main memory, generate a page fault and restart the access when the page fault is complete.

In Section 6.4, we presented cache memory. In our discussion of paging, we have introduced yet another type of cache—the TLB, which caches page table entries. TLBs use associative mapping. As is the case with L1 cache, computers often have two separate TLBs—one for instructions and another for data.

6.5.3 Putting It All Together: Using Cache, TLBs, and Paging

Because the TLB is essentially a cache, putting all these concepts together can be confusing. A walkthrough of the entire process will help you to grasp the overall idea. When the CPU generates an address, it is an address relative to the program itself, or a virtual address. This virtual address must be converted into a physical address before the data retrieval can proceed. There are two ways this is accomplished: (1) use the TLB to find the frame by locating a recently cached (page, frame) pair; or (2) in the event of a TLB miss, use the page table to find the corresponding frame in main memory (typically the TLB is updated at this point as well). This frame number is then combined with the offset given in the virtual address to create the physical address.

At this point, the virtual address has been converted into a physical address but the data at that address has not yet been retrieved. There are two possibilities for retrieving the data: (1) search cache to see if the data resides there; or (2) on a cache miss, go to the actual main memory location to retrieve the data (typically cache is updated at this point as well).

Figure 6.28 illustrates the process of using a TLB, paging, and cache memory.

6.5.4 Advantages and Disadvantages of Paging and Virtual Memory

In Section 6.5.2, we discussed how virtual memory implemented through paging adds an extra memory reference when accessing data. This time penalty is partially alleviated by using a TLB to cache page table entries. However, even with a high hit ratio in the TLB, this process still incurs translation overhead. Another disadvantage of virtual memory and paging is the extra resource consumption (the memory overhead for storing page tables). In extreme cases (very large programs), the page tables may take up a significant portion of physical memory. One solution offered for this latter problem is to page the page tables, which can get very confusing indeed! Virtual memory and paging also require special hardware and operating system support.

The benefits of using virtual memory must outweigh these disadvantages to make it useful in computer systems. But what are the advantages of virtual memory and paging? It is quite simple: Programs are no longer restricted by the amount of physical memory that is available. Virtual memory permits us to run individual programs whose virtual address space is larger than physical memory. (In effect, this allows one process to share physical memory with itself.) This makes it much easier to write programs because the programmer no longer has to worry about the physical address space limitations. Because each program requires less physical memory, virtual memory also permits us to run more programs at the same time. This allows us to share the machine among processes whose total address space sizes exceed the physical memory size, resulting in an increase in CPU utilization and system throughput.

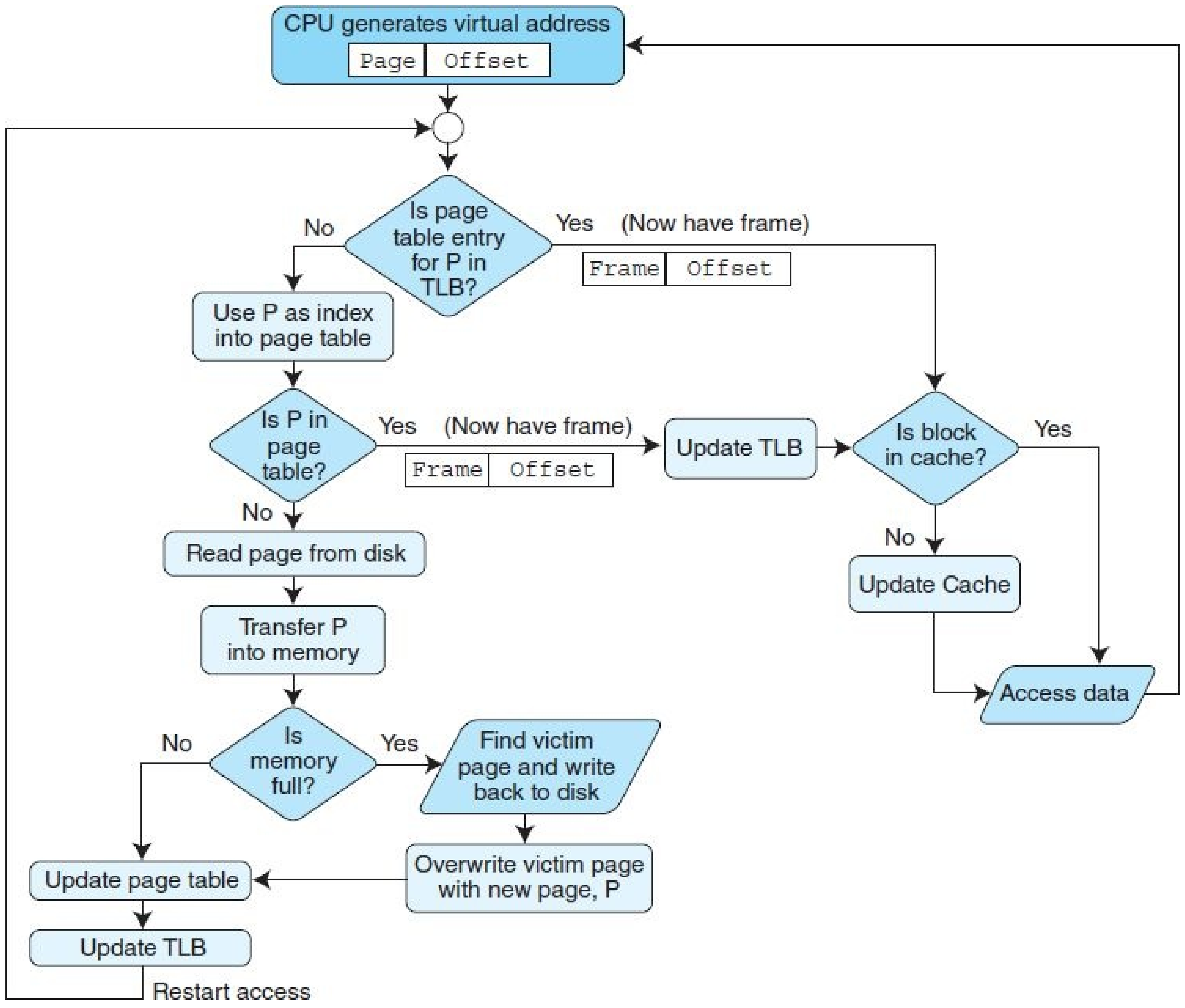


FIGURE 6.28 Putting It All Together: The TLB, Page Table, Cache, and Main Memory

The fixed size of frames and pages simplifies both allocation and placement from the perspective of the operating system. Paging also allows the operating system to specify protection ("this page belongs to User X and you can't access it") and sharing ("this page belongs to User X but you can read it") on a per page basis.

6.5.5 Segmentation

Although it is the most common method, paging is not the only way to implement virtual memory. A second method employed by some systems is **segmentation**. Instead of dividing the virtual address space into equal, fixed-size pages, and the physical address space into equal-size page frames, the virtual address space is divided into logical, variable-length units, or **segments**. Physical memory isn't really divided or partitioned into anything. When a segment needs to be copied into physical memory, the operating system looks for a chunk of free memory large enough to store the entire segment. Each segment has a base address, indicating where it is located in memory, and a bounds limit, indicating its size. Each program, consisting of multiple segments, now has an associated **segment table** instead of a page table. This segment table is simply a collection of the base/bounds pairs for each segment.

Memory accesses are translated by providing a segment number and an offset within the segment. Error checking is performed to make sure the offset is within the allowable bound. If it is, then the base value for that segment (found in the segment table) is added to the offset, yielding the actual physical address. Because paging is based on a fixed-size block and segmentation is based on a logical block, protection and sharing are easier using segmentation. For example, the virtual address space might be divided into a code segment, a data segment, a stack segment, and a symbol table segment, each of a different size. It is much easier to say, “I want to share all my data, so make my data segment accessible to everyone” than it is to say, “OK, in which pages does my data reside, and now that I have found those four pages, let’s make three of the pages accessible, but only half of that fourth page accessible.”

As with paging, segmentation suffers from fragmentation. Paging creates internal fragmentation because a frame can be allocated to a process that doesn’t need the entire frame. Segmentation, on the other hand, suffers from **external fragmentation**. As segments are allocated and deallocated, the free chunks that reside in memory become broken up. Eventually, there are many small chunks, but none large enough to store an entire segment. The difference between external and internal fragmentation is that, with external fragmentation, enough total memory space may exist to allocate to a process, but this space is not contiguous—it exists as a large number of small, unusable holes. With internal fragmentation, the memory simply isn’t available because the system has overallocated memory to a process that doesn’t need it. To combat external fragmentation, systems use some sort of **garbage collection**. This process simply shuffles occupied chunks of memory to coalesce the smaller, fragmented chunks into larger, usable chunks. If you have ever defragmented a disk drive, you have witnessed a similar process, collecting the many small free spaces on the disk and creating fewer, larger ones.

6.5.6 Paging Combined with Segmentation

Paging is not the same as segmentation. Paging is based on a purely physical value: The program and main memory are divided up into chunks of the same physical size. Segmentation, on the other hand, allows for logical portions of the program to be divided into variable-sized partitions. With segmentation, the user is aware of the segment sizes and boundaries; with paging, the user is unaware of the partitioning. Paging is easier to manage: Allocation, freeing, swapping, and relocating are easy when everything’s the same size. However, pages are typically smaller than segments, which means more overhead (in terms of resources to both track and transfer pages). Paging eliminates external fragmentation, whereas segmentation eliminates internal fragmentation. Segmentation has the ability to support sharing and protection, both of which are very difficult to do with paging.

Paging and segmentation both have their advantages; however, a system does not have to use one or the other—these two approaches can be combined, in an effort to get the best of both worlds. In a combined approach, the virtual address space is divided into segments of variable length, and the segments are divided into fixed-size pages. Main memory is divided into the same size frames.

Each segment has a page table, which means every program has multiple page tables. The physical address is divided into three fields. The first field is the segment field, which points the system to the appropriate page table. The second field is the page number, which is used as an offset into this page table. The third field is the offset within the page.

Combined segmentation and paging is advantageous because it allows for segmentation from the user’s point of view and paging from the system’s point of view.

6.6 A REAL-WORLD EXAMPLE OF MEMORY MANAGEMENT

Because the Pentium exhibits fairly characteristic traits of modern memory management, we present a short overview of how this processor deals with memory.

The Pentium architecture allows for 32-bit virtual addresses and 32-bit physical addresses. It uses either 4KB or 4MB page sizes when using paging. Paging and segmentation can be applied in different combinations, including unsegmented, unpaged memory; unsegmented, paged memory; segmented, unpaged memory; and segmented, paged memory.

The Pentium has two caches, L1 and L2, both utilizing a 32-byte block size. L1 is next to the processor, whereas L2 is between the processor and memory. The L1 cache is actually two caches; the Pentium (like many other machines) separates L1 cache into cache used to hold instructions (called the **I-cache**) and cache used to hold data (called the **D-cache**). Both L1 caches utilize an LRU bit for dealing with block replacement. Each L1 cache has a TLB; the D-cache TLB has 64 entries and the I-cache has only 32 entries. Both TLBs are 4-way set associative and use a pseudo-LRU replacement. The L1 D-cache and I-cache both use 2-way set associative mapping. The L2 cache can be from 512KB (for earlier models) up to 1MB (in later models). The L2 cache, like both L1 caches, uses 2-way set associative mapping.

To manage access to memory, the Pentium I-cache and the L2 cache use the MESI cache coherency protocol. Each cache line has two bits that store one of the following MESI states: (1) M: modified (cache is different from main memory); (2) E: exclusive (cache has not been modified and is the same as memory); (3) S: shared (this line/block may be shared with another cache line/block); and (4) I: invalid (the line/block is not in cache). Figure 6.29 presents an overview of the Pentium memory hierarchy.

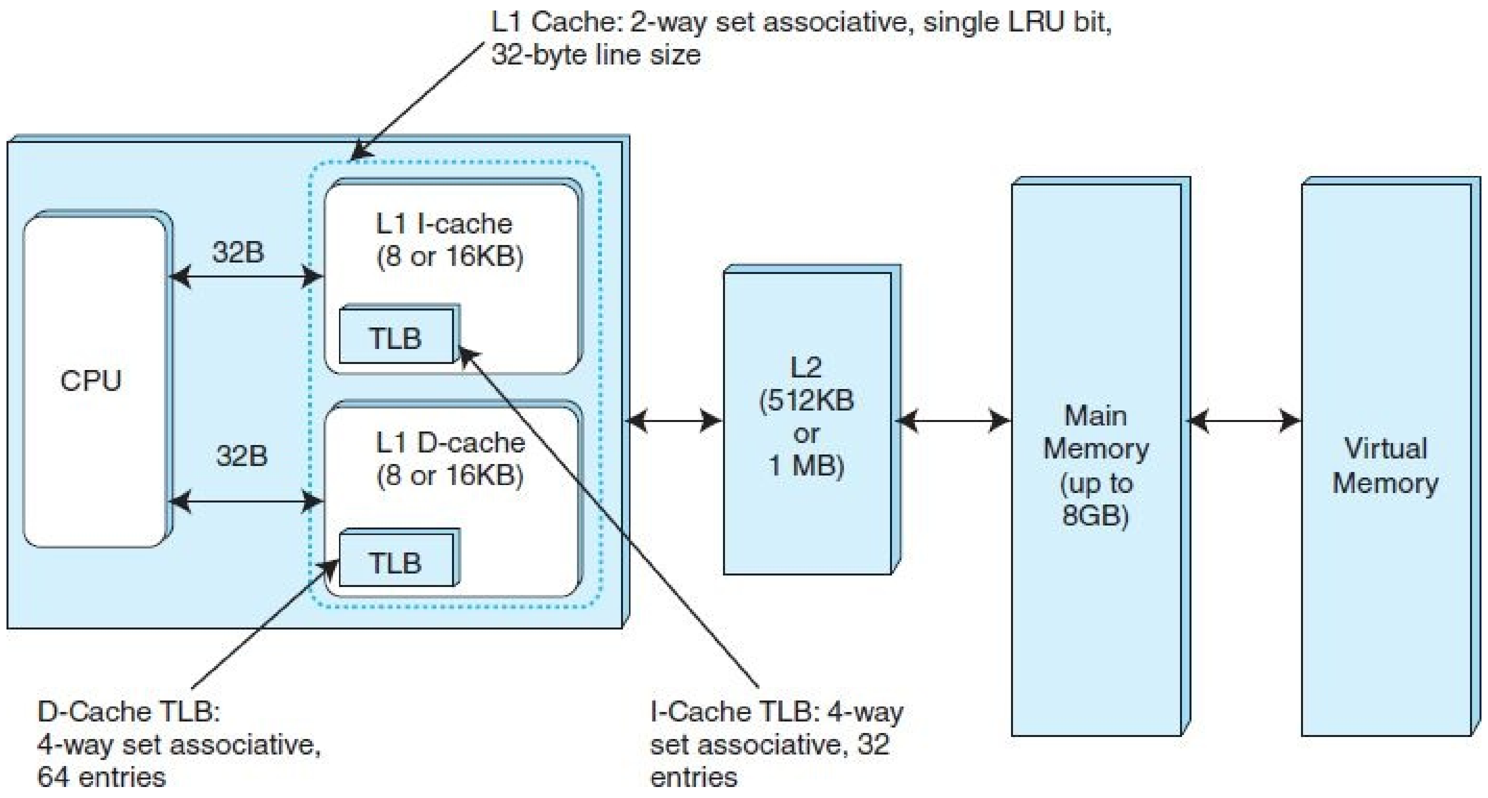

L1 Cache: 2-way set associative, single LRU bit, 32-byte line size
CPU
32B
32B
L1 I-cache (8 or 16KB)
TLB
L1 D-cache (8 or 16KB)
TLB
L2 (512KB or 1 MB)
Main Memory (up to 8GB)
Virtual Memory
D-Cache TLB: 4-way set associative, 64 entries
I-Cache TLB: 4-way set associative, 32 entries

FIGURE 6.29 Pentium Memory Hierarchy

We have given only a brief and basic overview of the Pentium and its approach to memory management. If you are interested in more details, please check the “Further Reading” section.

CHAPTER SUMMARY

Memory is organized as a hierarchy, with larger memories being cheaper but slower, and smaller memories being faster but more expensive. In a typical memory hierarchy, we find a cache, main memory, and secondary memory (usually a disk drive). The principle of locality helps bridge the gap between successive layers of this hierarchy, and the programmer gets the impression of a very fast and very large memory without being concerned about the details of transfers among the various levels of this hierarchy.

Cache acts as a buffer to hold the most frequently used blocks of main memory and is close to the CPU. One goal of the memory hierarchy is for the processor to see an effective access time very close to the access time of the cache. Achieving this goal depends on the behavioral properties of the programs being executed, the size and organization of the cache, and the cache replacement policy. Processor references that are found in cache are called cache hits; if not found, they are cache misses. On a miss, the missing data is fetched from main memory, and the entire block containing the data is loaded into cache. A unified cache holds both data and instructions, whereas a Harvard cache uses a separate cache for data and a separate cache for instructions. A multilevel cache hierarchy is used to increase cache performance.

The organization of cache determines the method the CPU uses to search cache for different memory addresses. Cache can be organized in different ways: direct mapped, fully associative, or set associative. Direct mapped cache needs no replacement algorithm; however, fully associative and set associative must use FIFO, LRU, or some other placement policy to determine the block to remove from cache to make room for a new block, if cache is full. LRU gives very good performance but is difficult to implement.

Another goal of the memory hierarchy is to extend main memory by using the hard disk itself, also called virtual memory. Virtual memory allows us to run programs whose virtual address space is larger than physical memory. It also allows more processes to run concurrently. The disadvantages of virtual memory implemented with paging include extra resource consumption (storing the page table) and extra memory accesses (to access the page table), unless a TLB is used to cache the most recently used virtual/physical address pairs. Virtual memory also incurs a translation penalty to convert the virtual address to a physical one as well as a penalty for processing a page fault should the requested page currently reside on disk instead of main memory. The relationship between virtual memory and main memory is similar to the relationship between main memory and cache. Because of this similarity, the concepts of cache memory and the TLB are often confused. In reality, the TLB *is* a cache. It is important to realize that virtual addresses must be translated to physical ones first, before anything else can be done, and this is what the TLB does. Although cache and paged memory appear to be similar, the objectives are different: Cache improves the effective access time to main memory, whereas paging extends the size of main memory.

FURTHER READING

Mano (2007) has a nice explanation of RAM. Stallings (2013) also gives a very good explanation of RAM. Hamacher, Vranesic, and Zaky (2002) contains an extensive discussion of cache. For good coverage of virtual memory, see Stallings (2012), Tanenbaum (2013), or Tanenbaum and Woodhull (2006). For more information on memory management in general, check out the books by Flynn and McHoes (2010), Stallings (2013), Tanenbaum and Woodhull (2006), or Silberschatz, Galvin, and Gagne (2013). Hennessy and Patterson (2012) discuss issues involved with determining cache performance. For an online tutorial on memory technologies, see www.kingston.com/tools/umg. George Mason University also has a set of workbenches on various computer topics. The workbench for virtual memory is located at cs.gmu.edu/cne/workbenches/vmemory.html.

REFERENCES

Flynn, I. M., & McHoes, A. M. *Understanding Operating Systems*, 6th ed. Boston, MA: Thomson Course Technology, 2010.

Hamacher, V. C., Vranesic, Z. G., & Zaky, S. G. *Computer Organization*, 5th ed. New York: McGraw-Hill, 2002.

Hennessy, J. L., & Patterson, D. A. *Computer Architecture: A Quantitative Approach*, 5th ed. San Francisco: Morgan Kaufmann, 2012.

Mano, M. *Digital Design*, 4th ed. Upper Saddle River, NJ: Prentice Hall, 2007.

Silberschatz, A., Galvin, P., & Gagne, G. *Operating System Concepts*, 9th ed. New York, NY: John Wiley & Sons, 2013.

Stallings, W. *Computer Organization and Architecture*, 9th ed. Upper Saddle River, NJ: Prentice Hall, 2013.

Stallings, W. *Operating Systems*, 7th ed. Upper Saddle River, NJ: Prentice Hall, 2012.

Tanenbaum, A. *Structured Computer Organization*, 6th ed. Englewood Cliffs, NJ: Prentice Hall, 2013.

Tanenbaum, A., & Woodhull, S. *Operating Systems, Design and Implementation*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 2006.

REVIEW OF ESSENTIAL TERMS AND CONCEPTS

1. Which is faster, SRAM or DRAM?

2. What are the advantages of using DRAM for main memory?

3. Name three different applications where ROMs are often used.

4. Explain the concept of a memory hierarchy. Why did your authors choose to represent it as a pyramid?

5. Explain the concept of locality of reference, and state its importance to memory systems.

6. What are the three forms of locality?

7. Give two noncomputer examples of the concept of cache.

8. Which of L1 or L2 cache is faster? Which is smaller? Why is it smaller?

9. Cache is accessed by its ________, whereas main memory is accessed by its _______.

10. What are the three fields in a direct mapped cache address? How are they used to access a word located in cache?

11. How does associative memory differ from regular memory? Which is more expensive and why?

12. Explain how fully associative cache is different from direct mapped cache.

13. Explain how set associative cache combines the ideas of direct and fully associa tive cache.

14. Direct mapped cache is a special case of set associative cache where the set size is 1. So fully associative cache is a special case of set associative cache where the set size is ___.

15. What are the three fields in a set associative cache address, and how are they used to access a location in cache?

16. Explain the four cache replacement policies presented in this chapter.

17. Why is the optimal cache replacement policy important?

18. What is the worst-case cache behavior that can develop using LRU and FIFO cache replacement policies?

19. What, exactly, is effective access time (EAT)?

20. Explain how to derive an effective access time formula.

21. When does caching behave badly?

22. What is a dirty block?

23. Describe the advantages and disadvantages of the two cache write policies.

24. Explain the difference between a unified cache and a Harvard cache.

25. What are the advantages of a Harvard cache?

26. Why would a system contain a victim cache? A trace cache?

27. Explain the differences among L1, L2, and L3 cache.

28. Explain the differences between inclusive and exclusive cache.

29. What is the advantage to a nonblocking cache?

30. What is the difference between a virtual memory address and a physical memory address? Which is larger? Why?

31. What is the objective of paging?

32. Discuss the pros and cons of paging.

33. What is a page fault?

34. What causes internal fragmentation?

35. What are the components (fields) of a virtual address?

36. What is a TLB, and how does it improve EAT?

37. What are the advantages and disadvantages of virtual memory?

38. When would a system ever need to page its page table?

39. What causes external fragmentation, and how can it be fixed?

EXERCISES

◆ **1.** Suppose a computer using direct mapped cache has 2^{20} bytes of byte-addressable main memory and a cache of 32 blocks, where each cache block contains 16 bytes.

a) How many blocks of main memory are there?

b) What is the format of a memory address as seen by the cache; that is, what are the sizes of the tag, block, and offset fields?

c) To which cache block will the memory address 0x0DB63 map?

2. Suppose a computer using direct mapped cache has 2^{32} bytes of byte-addressable main memory and a cache of 1024 blocks, where each cache block contains 32 bytes.

a) How many blocks of main memory are there?

b) What is the format of a memory address as seen by the cache; that is, what are the sizes of the tag, block, and offset fields?

c) To which cache block will the memory address 0x000063FA map?

3. Suppose a computer using direct mapped cache has 2^{32} bytes of byte-addressable main memory and a cache size of 512 bytes, and each cache block contains 64 bytes.

a) How many blocks of main memory are there?

b) What is the format of a memory address as seen by cache; that is, what are the sizes of the tag, block, and offset fields?

c) To which cache block will the memory address 0x13A4498A map?

◆ **4.** Suppose a computer using fully associative cache has 2^{16} bytes of byte-addressable main memory and a cache of 64 blocks, where each cache block contains 32 bytes.

a) How many blocks of main memory are there?

b) What is the format of a memory address as seen by the cache; that is, what are the sizes of the tag and offset fields?

c) To which cache block will the memory address 0xF8C9 map?

5. Suppose a computer using fully associative cache has 2^{24} bytes of byte-addressable main memory and a cache of 128 blocks, where each cache block contains 64 bytes.

a) How many blocks of main memory are there?

b) What is the format of a memory address as seen by the cache; that is, what are the sizes of the tag and offset fields?

c) To which cache block will the memory address 0x01D872 map?

6. Suppose a computer using fully associative cache has 2^{24} bytes of byte-addressable main memory and a cache of 128 blocks, where each block contains 64 bytes.

a) How many blocks of main memory are there?

b) What is the format of a memory address as seen by cache; that is, what are the sizes of the tag and offset fields?

c) To which cache block will the memory address 0x01D872 map?

◆ **7.** Assume that a system's memory has 128M bytes. Blocks are 64 bytes in length, and the cache consists of 32K blocks. Show the format for a main memory address assuming a 2-way set associative cache mapping scheme and byte addressing. Be sure to include the fields as well as their sizes.

8. A 2-way set associative cache consists of 4 sets. Main memory contains 2K blocks of 8 bytes each and byte addressing is used.

a) Show the main memory address format that allows us to map addresses from main memory to cache. Be sure to include the fields as well as their sizes.

b) Compute the hit ratio for a program that loops 3 times from addresses 0x8 to 0x33 in main memory. You may leave the hit ratio in terms of a fraction.

9. Suppose a byte-addressable computer using set associative cache has 2^{16} bytes of main memory and a cache of 32 blocks, and each cache block contains 8 bytes.

a) If this cache is 2-way set associative, what is the format of a memory address as seen by the cache; that is, what are the sizes of the tag, set, and offset fields?

b) If this cache is 4-way set associative, what is the format of a memory address as seen by the cache?

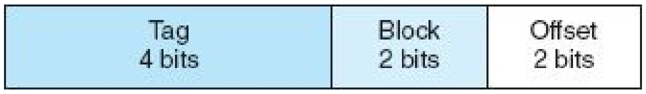
10. Suppose a byte-addressable computer using set associative cache has 2^{21} bytes of main memory and a cache of 64 blocks, where each cache block contains 4 bytes.

a) If this cache is 2-way set associative, what is the format of a memory address as seen by the cache; that is, what are the sizes of the tag, set, and offset fields?

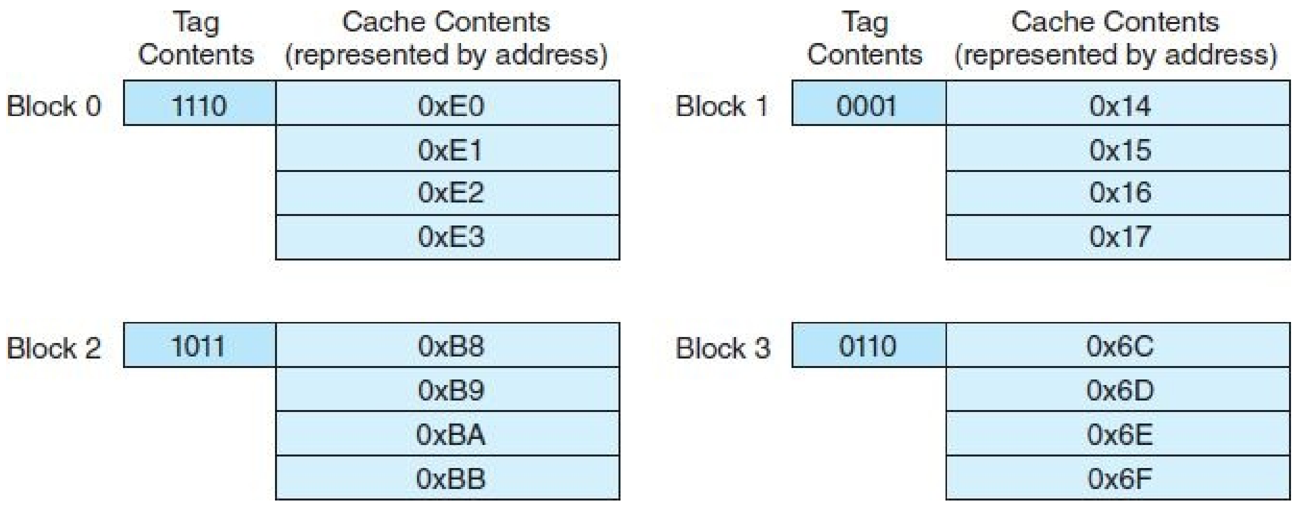
b) If this cache is 4-way set associative, what is the format of a memory address as seen by the cache?

*** 11.** Suppose we have a computer that uses a memory address word size of 8 bits. This computer has a 16-byte cache with 4 bytes per block. The computer accesses a number of memory locations throughout the course of running a program.

Suppose this computer uses direct-mapped cache. The format of a memory address as seen by the cache is shown here:



The system accesses memory addresses in this exact order: 0x6E, 0xB9, 0x17, 0xE0, 0x4E, 0x4F, 0x50, 0x91, 0xA8, 0xA9, 0xAB, 0xAD, 0x93, and 0x94. The memory addresses of the first four accesses have been loaded into the cache blocks as shown below. (The contents of the tag are shown in binary, and the cache "contents" are simply the address stored at that cache location.)



a) What is the hit ratio for the entire memory reference sequence given above, assuming that we count the first four accesses as misses?

b) What memory blocks will be in the cache after the last address has been accessed?

12. Given a byte-addressable memory with 256 bytes, suppose a memory dump yields the results shown below. The address of each memory cell is determined by its row and column. For example, memory address 0x97 is in the 9th row, 7th column, and contains the hexadecimal value 43. Memory location 0xA3 contains the hexadecimal value 58.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	DC	D5	9C	77	C1	99	90	AC	33	D1	37	74	B5	82	38	E0
1	49	E2	23	FD	D0	A6	98	BB	DE	9A	9E	EB	04	AA	86	E5
2	3A	14	F3	59	5C	41	B2	6D	18	3C	9D	1F	2F	78	44	1E
3	4E	B7	29	E7	87	3D	B8	E1	EF	C5	CE	BF	93	CB	39	7F
4	6B	69	02	56	7E	DA	2A	76	89	20	85	88	72	92	E9	5B
5	B9	16	A8	FA	AE	68	21	25	34	24	B6	48	17	83	75	0A
6	40	2B	C4	1D	08	03	0E	0B	B4	C2	53	FB	E3	8C	0C	9B
7	31	AF	30	9F	A4	FE	09	60	4F	D7	D9	97	2E	6C	94	BC
8	CD	80	64	B3	8D	81	A7	DB	F1	BA	66	BE	11	1A	A1	D2
9	61	28	5D	D4	4A	10	A2	43	CC	07	7D	5A	C0	D3	CF	67
A	52	57	A3	58	55	0F	E8	F6	91	F0	C3	19	F9	BD	8B	47
B	26	51	1C	C6	3B	ED	7B	EE	95	12	7C	DF	B1	4D	EC	42
C	22	0D	F5	2C	62	B0	5E	DD	8E	96	A0	C8	27	3E	EA	01
D	50	35	A9	4C	6A	00	8A	D6	5F	7A	FF	71	13	F4	F8	46
E	1B	4B	70	84	6E	F7	63	3F	CA	45	65	73	79	C9	FC	A5
F	AB	E6	2D	54	E4	8F	36	6F	C7	05	D8	F2	AD	15	32	06

The system from which this memory dump was produced contains 4 blocks of cache, where each block consists of 8 bytes. Assume that the following sequence of memory addresses takes place: 0x2C, 0x6D, 0x86, 0x29, 0xA5, 0x82, 0xA7, 0x68, 0x80, and 0x2B.

a) How many blocks of main memory are there?

b) Assuming a direct mapped cache:

i. Show the format for a main memory address assuming that the system uses direct mapped cache. Specify field names and sizes.

ii. What does cache look like after the 10 memory accesses have taken place? Draw the cache and show contents and tags.

iii. What is the hit rate for this cache on the given sequence of memory accesses?

c) Assuming a fully associative cache:

i. Show the format for a main memory address. Specify field names and sizes.

ii. Assuming that all cache blocks are initially empty, blocks are loaded into the first available empty cache location, and cache uses a first-in, first-out replacement policy, what does cache look like after the 10 memory accesses have taken place?

iii. What is the hit rate for this cache on the given sequences of memory accesses?

d) Assuming a 2-way set associative cache:

i. Show the format for a main memory address. Specify field names and sizes.

ii. What does cache look like after the 10 memory accesses have taken place?

iii. What is the hit ratio for this cache on the given sequence of memory accesses? iv. If a cache hit retrieves a value in 5ns, and retrieving a value from main memory requires 25ns, what is the average effective access time for this cache, assuming that all memory accesses exhibit the same hit rate as the sequence of 10 given, and assuming that the system uses a nonoverlapped (sequential) access strategy?

13. A direct mapped cache consists of 8 blocks. Byte-addressable main memory contains 4K blocks of 8 bytes each. Access time for the cache is 22ns, and the time required to fill a cache slot from main memory is 300ns. (This time allows us to determine that the block is missing and bring it into cache.) Assume that a request is always started in parallel to both cache and to main memory (so if it is not found in cache, we do not have to add this cache search time to the memory access). If a block is missing from cache, the entire block is brought into the cache and the access is restarted. Initially, the cache is empty.

a) Show the main memory address format that allows us to map addresses from main memory to cache. Be sure to include the fields as well as their sizes.

b) Compute the hit ratio for a program that loops 4 times from addresses 0x0 to 0x43 in memory.

c) Compute the effective access time for this program.

14. Consider a byte-addressable computer with 24-bit addresses, a cache capable of storing a total of 64KB of data, and blocks of 32 bytes. Show the format of a 24-bit memory address for:

a) direct mapped

b) associative

c) 4-way set associative

*** 15.** Suppose a byte-addressable computer using 4-way set associative cache has 2^{16} words of main memory (where each word is 32 bits) and a cache of 32 blocks, where each block is 4 words. Show the main memory address format for this machine. (Hint: Because this architecture is byte addressable, and the number of addresses is critical in determining the address format, you must convert everything to bytes.)

16. Assume a direct mapped cache that holds 4096 bytes, in which each block is 16 bytes. Assuming that an address is 32 bits and that cache is initially empty, complete the table that follows. (You should use hexadecimal numbers for all answers.) Which, if any, of the addresses will cause a collision (forcing the block that was just brought in to be overwritten) if they are accessed one right after the other?

Address	TAG	Cache Location (block)	Offset within Block
0x0FF0FABA			
0x00000011			
0x0FFFFFFE			
0x23456719			
0xCAFEBABE			

17. Redo exercise 16, assuming now that cache is 16-way set associative.

Address	TAG	Cache Location (set)	Offset within Block
0x0FF0FABA			
0x00000011			
0x0FFFFFFE			
0x23456719			
0xCAFEBABE			

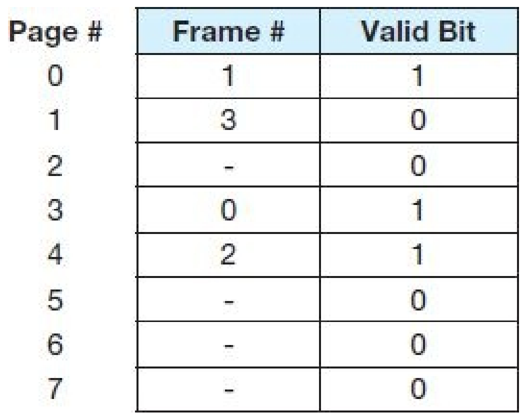
18. Suppose a process page table contains the entries shown below. Using the format shown in Figure 6.17a, indicate where the process pages are located in memory.

Frame	Valid Bit
1	1
-	0
0	1
3	1
-	0
-	0
2	1
-	0

19. Suppose a process page table contains the entries shown below. Using the format shown in Figure 6.22a, indicate where the process pages are located in memory.

Frame	Valid Bit
-	0
3	1
-	0
-	0
2	1
0	1
-	0
1	1

20. Suppose you have a byte-addressable virtual address memory system with eight virtual pages of 64 bytes each, and four page frames. Assuming the following page table, answer the questions below:



Page #	Frame #	Valid Bit
0	1	1
1	3	0
2	-	0
3	0	1
4	2	1
5	-	0
6	-	0
7	-	0

a) How many bits are in a virtual address?

b) How many bits are in a physical address?

c) What physical address corresponds to the following virtual addresses? (If the address causes a page fault, simply indicate this is the case.)

i. 0x0

ii. 0x44

iii. 0xC2

iv. 0x80

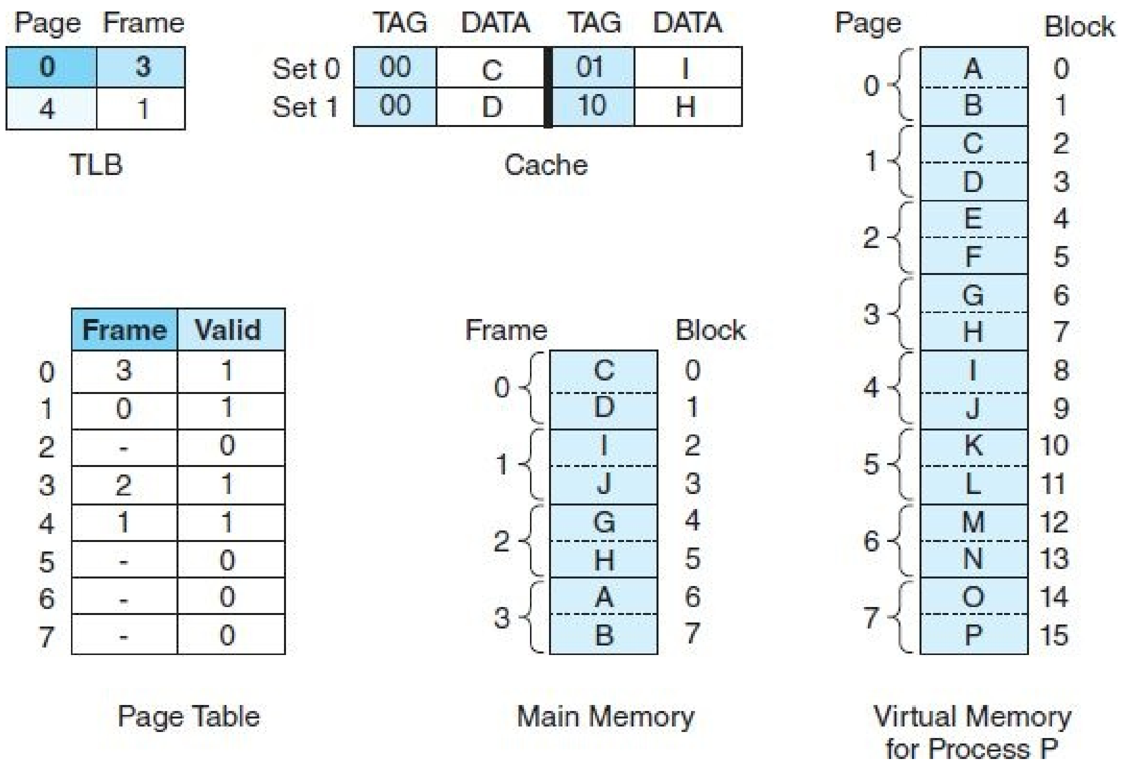
21. Suppose we have 2^{10} bytes of virtual memory and 2^8 bytes of physical main memory. Suppose the page size is 2^4 bytes.

a) How many pages are there in virtual memory?

b) How many page frames are there in main memory?

c) How many entries are in the page table for a process that uses all of virtual memory?

* **22.** You have a byte-addressable virtual memory system with a two-entry TLB, a 2-way set associative cache, and a page table for a process P. Assume cache blocks of 8 bytes and page size of 16 bytes. In the system below, main memory is divided into blocks, where each block is represented by a letter. Two blocks equal one frame.



Page	Frame
0	3
4	1

TLB

	TAG	DATA	TAG	DATA
Set 0	00	C	01	I
Set 1	00	D	10	H

Cache

	Frame	Valid
0	3	1
1	0	1
2	-	0
3	2	1
4	1	1
5	-	0
6	-	0
7	-	0

Page Table

Frame		Block
0	C	0
	D	1
1	I	2
	J	3
2	G	4
	H	5
3	A	6
	B	7

Main Memory

Page		Block
0	A	0
	B	1
1	C	2
	D	3
2	E	4
	F	5
3	G	6
	H	7
4	I	8
	J	9
5	K	10
	L	11
6	M	12
	N	13
7	O	14
	P	15

Virtual Memory for Process P

Given the system state as depicted above, answer the following questions:

a) How many bits are in a virtual address for process P? Explain.

b) How many bits are in a physical address? Explain.

c) Show the address format for virtual address 0x12 (specify field name and size) that would be used by the system to translate to a physical address and then translate this virtual address into the corresponding physical address. (Hint: Convert the address to its binary equivalent and divide it into the appropriate fields.) Explain how these fields are used to translate to the corresponding physical address.

d) Given virtual address 0x06 converts to physical address 0x36. Show the format for a physical address (specify the field names and sizes) that is used to determine the cache location for this address. Explain how to use this format to determine where physical address 0x36 would be located in cache. (Hint: Convert 0x36 to binary and divide it into the appropriate fields.)

e) Given virtual address 0x19 is located on virtual page 1, offset 9. Indicate exactly how this address would be translated to its corresponding physical address and how the data would be accessed. Include in your explanation how the TLB, the page table, cache, and memory are used.

23. Given a virtual memory system with a TLB, a cache, and a page table, assume the following:

- A TLB hit requires 5ns.
- A cache hit requires 12ns.
- A memory reference requires 25ns.
- A disk reference requires 200ms (this includes updating the page table, cache, and TLB).
- The TLB hit ratio is 90%.
- The cache hit rate is 98%.
- The page fault rate is .001%.
- On a TLB or cache miss, the time required for access includes a TLB and/or cache update, but the access is *not* restarted.
- On a page fault, the page is fetched from disk, and all updates are performed, but the access *is* restarted.
- All references are sequential (no overlap, nothing done in parallel).

For each of the following, indicate whether or not it is possible. If it is possible, specify the time required for accessing the requested data.

a) TLB hit, cache hit

b) TLB miss, page table hit, cache hit

c) TLB miss, page table hit, cache miss

d) TLB miss, page table miss, cache hit

e) TLB miss, page table miss

Write down the equation to calculate the effective access time.

24. Does a TLB miss always indicate that a page is missing from memory? Explain.

25. A system implements a paged virtual address space for each process using a one-level page table. The maximum size of virtual address space is 16MB. The page table for the running process includes the following valid entries (the → notation indicates that a virtual page maps to the given page frame; that is, it is located in that frame):

Virtual page 2 → Page frame 4	Virtual page 4 → Page frame 9
Virtual page 1 → Page frame 2	Virtual page 3 → Page frame 16
Virtual page 0 → Page frame 1	

The page size is 1024 bytes and the maximum physical memory size of the machine is 2MB.

a) How many bits are required for each virtual address?

b) How many bits are required for each physical address?

c) What is the maximum number of entries in a page table?

d) To which physical address will the virtual address 0x5F4 translate?

e) Which virtual address will translate to physical address 0x400?

26. **a)** If you are a computer builder trying to make your system as price-competitive as possible, what features and organization would you select for its memory hierarchy?

b) If you are a computer buyer trying to get the best performance from a system, what features would you look for in its memory hierarchy?

*** 27.** Consider a system that has multiple processors where each processor has its own cache, but main memory is shared among all processors.

a) Which cache write policy would you use?

b) **The Cache Coherency Problem**. With regard to the system just described, what problems are caused if a processor has a copy of memory block *A* in its cache and a second processor, also having a copy of *A* in its cache, then updates main memory block *A*? Can you think of a way (perhaps more than one) of preventing this situation, or lessening its effects?

*** 28.** Pick a specific architecture (other than the one covered in this chapter). Do research to find out how your architecture approaches the concepts introduced in this chapter, as was done for Intel's Pentium.

29. Name two ways that, as a programmer, you can improve cache performance.

30. Look up a specific vendor's specifications for memory, and report the memory access time, cache access time, and cache hit rate (and any other data the vendor provides).