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.

—Bill Gates

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 book. 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 memory 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, 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 (read-only memory) 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 different 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.

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, and secondary memory.

Today's computers each have a small amount of very high-speed memory, called a *cache*, 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, composed of a hard disk and various removable media. 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 further 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, due to 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.

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 at addresses $X + 1$, $X + 2$, . . . , 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*, which often allows the processor to access the data returned for addresses $X + 1$, $X + 2$, and so on. Thus, although there is one miss to, say

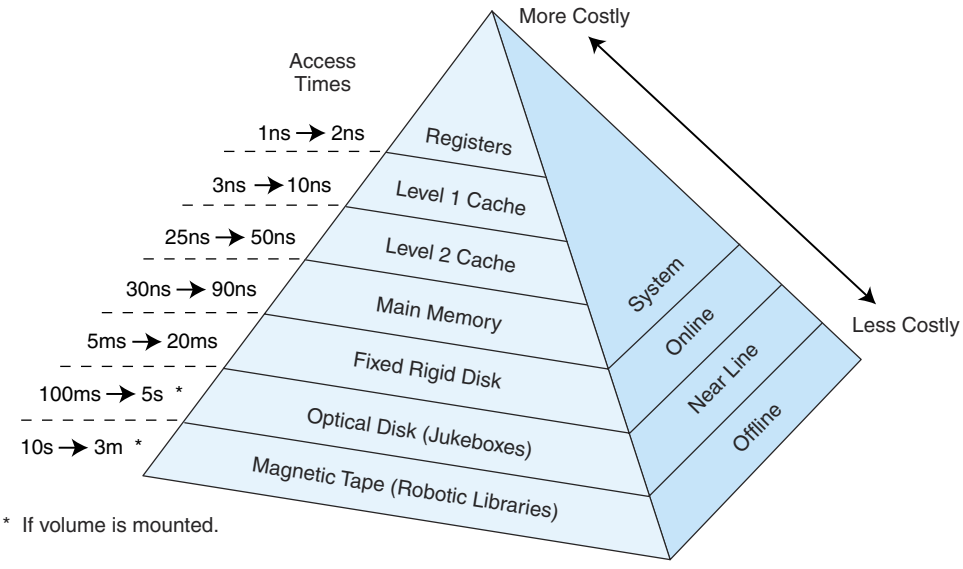

FIGURE 6.1 The Memory Hierarchy

cache, for *X*, there may be several hits in cache on the newly retrieved block afterward, due to locality.

6.3.1 Locality of Reference

In practice, processors tend to access memory in a very 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 further 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 a number up 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 very 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" remain 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 or 16K. L1 cache resides on the processor, whereas L2 cache 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.

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 60ns 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 2E3 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.

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 word exists there. When the requested word is not found in cache, the entire main memory block in which the word resides is loaded into cache. As previously mentioned, this scheme is successful because of the principle of locality—if a word was just referenced, there is a good chance words in the same general vicinity will soon be referenced as well. Therefore, one missed word often results in several found words. 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 a cache word (a tool already in the basement) is faster than accessing a main memory word (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 word in the block; this can be done using a different portion of the main memory address called the *word field*. All cache mapping schemes require a word field; however, the remaining fields are determined by the mapping scheme. We discuss the three main cache mapping schemes on the next page.