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.

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 10 blocks, then main memory block 0 maps to cache block 0, main memory block 1 maps to cache block 1, . . . , main memory block 9 maps to cache block 9, and main memory block 10 maps to cache block 0. This is illustrated in Figure 6.2. Thus, main memory blocks 0 and 10 (and 20, 30, and so on) all compete for cache block 0.

You may be wondering, if main memory blocks 0 and 10 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

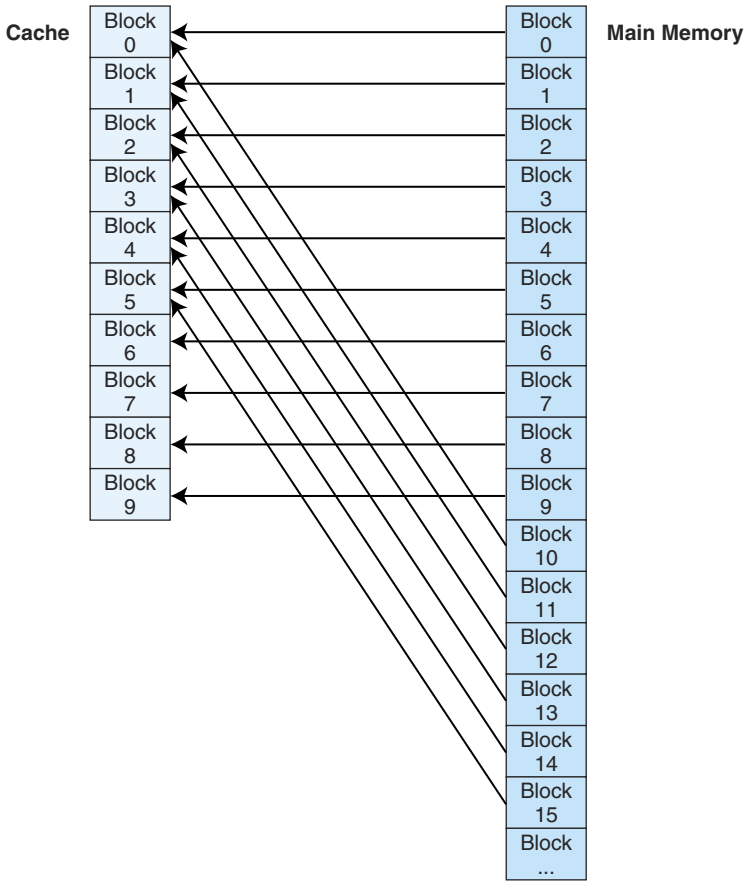

FIGURE 6.2 Direct Mapping of Main Memory Blocks to Cache Blocks

Block	Tag	Data	Valid
0	00000000	words A, B, C,...	1
1	11110101	words L, M, N,...	1
2	----------		0
3	----------		0

FIGURE 6.3 A Closer Look at Cache

by the tag previously described. If we take a closer look at cache, we see that it stores more than just that data copied from main memory, as indicated in Figure 6.3. In this figure, there are two valid cache blocks. Block 0 contains multiple words from main memory, identified using the tag "00000000". Block 1 contains words identified using tag "11110101". The other two cache blocks are not valid.

To perform direct mapping, the binary main memory address is partitioned into the fields shown in Figure 6.4.

The size of each field depends on the physical characteristics of main memory and cache. The *word* field (sometimes called the *offset* field) uniquely identifies a word from a specific block; therefore, it must contain the appropriate number of bits to do this. This is also true of the *block* field—it must select a unique block of cache. 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.

Consider the following example: Assume memory consists of 2^{14} words, cache has 16 blocks, and each block has 8 words. From this we determine that memory has $\frac{2^{14}}{2^3} = 2^{11}$ blocks. We know that each main memory address requires 14 bits. Of this 14-bit address field, the rightmost 3 bits reflect the word field (we need 3 bits to uniquely identify one of 8 words 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

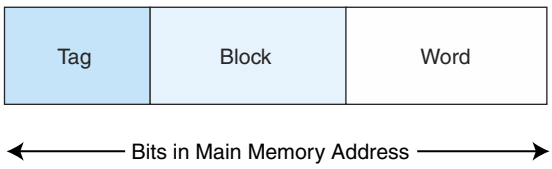

FIGURE 6.4 The Format of a Main Memory Address Using Direct Mapping

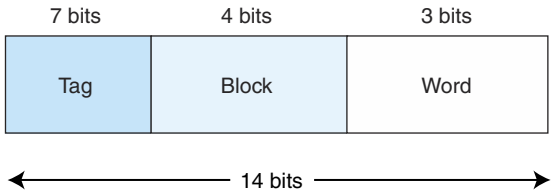

FIGURE 6.5 The Main Memory Address Format for Our Example

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.

To see how these addresses differ, let's look at a smaller, simpler example. Suppose we have a system using direct mapping with 16 words of main memory divided into 8 blocks (so each block has 2 words). Assume the cache is 4 blocks in size (for a total of 8 words). Table 6.1 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 words in main memory).
- This 4-bit main memory address is divided into three fields: The word 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 main 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 6.6.

Main Memory	Maps To	Cache
Block 0 (addresses 0, 1)	→	Block 0
Block 1 (addresses 2, 3)	→	Block 1
Block 2 (addresses 4, 5)	→	Block 2
Block 3 (addresses 6, 7)	→	Block 3
Block 4 (addresses 8, 9)	→	Block 0
Block 5 (addresses 10, 11)	→	Block 1
Block 6 (addresses 12, 13)	→	Block 2
Block 7 (addresses 14, 15)	→	Block 3

TABLE 6.1 An Example of Main Memory Mapped to Cache

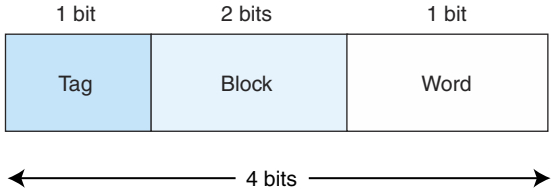

FIGURE 6.6 The Main Memory Address Format for a 16-Word Memory

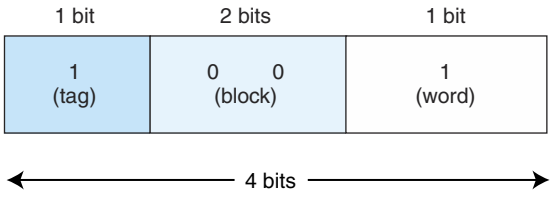

FIGURE 6.7 The Main Memory Address 9 = 1001_2 Split into Fields

Suppose we generate the main memory address 9. We can see from the mapping listing above that address 9 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 6.7.

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 9 = 1001_2, which is in block 4, to main memory address 1 = 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 the tags match, which means that block 4 from main memory (with addresses 8 and 9) resides in cache block 0, the word field value of 1 is used to select one of the two words residing in the block. Because the bit is 1, we select the word with offset 1, which results in retrieving the data copied from main memory address 9.

Let's do one more example in this context. Suppose the CPU now generates address 4 = 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 word 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 12 = 1100_2.

Let's move on to a larger example. Suppose we have a system using 15-bit main memory addresses and 64 blocks of cache. If each block contains 8 words, we know that the main memory 15-bit address is divided into a 3-bit word field, a 6-bit block field, and a 6-bit tag field. If the CPU generates the main memory address: