it would look in block 0 of cache, and if it finds a tag of 000010, the word at offset 4 in this block would be returned to the CPU.

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.

Using associative mapping, the main memory address is partitioned into two pieces, the tag and the word. For example, using our previous memory configuration with 2^{14} words, a cache with 16 blocks, and blocks of 8 words, we see from Figure 6.8 that the word 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

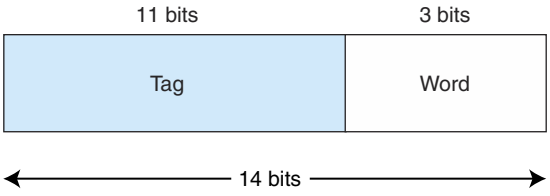

FIGURE 6.8 The Main Memory Address Format for Associative Mapping

would a least-recently used algorithm. There are many replacement algorithms that can be used; these are discussed shortly.

Set Associative Cache

Owing to 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 our previous examples. Suppose the program is using block 0, then block 16, then 0, then 16, and so on as it executes instructions. Blocks 0 and 16 both map to the same location, which means the program would repeatedly throw out 0 to bring in 16, then throw out 16 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 two cache blocks per set, as seen in Figure 6.9. In this figure, we see that set 0 contains two blocks, one that is valid and holds the data A, B, C, . . . , and another that is not valid. The same is true for Set 1. Set 2 and Set 3 can also hold two blocks, but currently, only the second block is valid in each set. In an 8-way set associative cache, there are 8 cache blocks per set. Direct mapped cache is a special case of *N*-way set associative cache mapping where the set size is one.

In set-associative cache mapping, the main memory address is partitioned into three pieces: the tag field, the set field, and the word field. The tag and word fields assume the same roles as before; the set field indicates into which cache set the main memory block maps. Suppose we are using 2-way set associative mapping with a main memory of 2^{14} words, a cache with 16 blocks, where each block contains 8 words. If cache consists of a total of 16 blocks, and each set has 2

Set	Tag	Block 0 of set	Valid	Tag	Block 1 of set	Valid
0	00000000	Words A, B, C, . . .	1	----------		0
1	11110101	Words L, M, N, . . .	1	----------		0
2	----------		0	10111011	P, Q, R, . . .	1
3	----------		0	11111100	T, U, V, . . .	1

FIGURE 6.9 A Two-Way Set Associative Cache

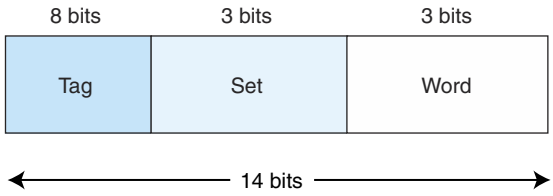

FIGURE 6.10 Format for Set Associative Mapping

blocks, then there are 8 sets in cache. Therefore, the set field is 3 bits, the word field is 3 bits, and the tag field is 8 bits. This is illustrated in Figure 6.10.

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

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

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 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 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. 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 we are accessing each element of the array). If a program accesses the array in column-major order, the first access to the column results in a miss, after which an entire row is transferred in. However, the second access to the column results in another miss. 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 fashion.

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 the system down 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