CHAPTER 6

Memory System Design I

In this chapter, we study the computer memory system. It was stated in Chapter 3 that without a memory no information can be stored or retrieved in a computer. It is interesting to observe that as early as 1946 it was recognized by Burks, Goldstine, and Von Neumann that a computer memory has to be organized in a hierarchy. In such a hierarchy, larger and slower memories are used to supplement smaller and faster ones. This observation has since then proven essential in constructing a computer memory. If we put aside the set of CPU registers (as the first level for storing and retrieving information inside the CPU, see Chapter 5), then a typical memory hierarchy starts with a small, expensive, and relatively fast unit, called the *cache*. The cache is followed in the hierarchy by a larger, less expensive, and relatively slow *main memory* unit. Cache and main memory are built using solid-state semiconductor material. They are followed in the hierarchy by far larger, less expensive, and much slower magnetic memories that consist typically of the (hard) disk and the tape. Our deliberation in this chapter starts by discussing the characteristics and factors influencing the success of a memory hierarchy of a computer. We then direct our attention to the design and analysis of cache memory. Discussion on the (main) memory unit is conducted in Chapter 7. Also discussed in Chapter 7 are the issues related to *virtual memory* design. A brief coverage of the different *read-only memory* (ROM) implementations is also provided in Chapter 7.

6.1. BASIC CONCEPTS

In this section, we introduce a number of fundamental concepts that relate to the memory hierarchy of a computer.

6.1.1. Memory Hierarchy

As mentioned above, a typical memory hierarchy starts with a small, expensive, and relatively fast unit, called the *cache*, followed by a larger, less expensive, and relatively slow *main memory* unit. Cache and main memory are built using solid-state

Fundamentals of Computer Organization and Architecture, by M. Abd-El-Barr and H. El-Rewini
ISBN 0-471-46741-3

semiconductor material (typically CMOS transistors). It is customary to call the fast memory level the *primary memory*. The solid-state memory is followed by larger, less expensive, and far slower magnetic memories that consist typically of the (hard) disk and the tape. It is customary to call the disk the *secondary memory*, while the tape is conventionally called the *tertiary memory*. The objective behind designing a memory hierarchy is to have a memory system that performs as if it consists entirely of the fastest unit and whose cost is dominated by the cost of the slowest unit.

The memory hierarchy can be characterized by a number of parameters. Among these parameters are the *access type*, *capacity*, *cycle time*, *latency*, *bandwidth*, and *cost*. The term *access* refers to the action that physically takes place during a *read* or *write* operation. The capacity of a memory level is usually measured in bytes. The cycle time is defined as the time elapsed from the start of a read operation to the start of a subsequent read. The latency is defined as the time interval between the request for information and the access to the first bit of that information. The bandwidth provides a measure of the number of bits per second that can be accessed. The cost of a memory level is usually specified as dollars per megabytes. Figure 6.1 depicts a typical memory hierarchy. Table 6.1 provides typical values of the memory hierarchy parameters.

The term *random access* refers to the fact that any access to any memory location takes the same fixed amount of time regardless of the actual memory location and/or the sequence of accesses that takes place. For example, if a *write* operation to memory location 100 takes 15 ns and if this operation is followed by a *read* operation to memory location 3000, then the latter operation will also take 15 ns. This is to be compared to sequential access in which if access to location 100 takes 500 ns, and if a consecutive access to location 101 takes 505 ns, then it is expected that an access to location 300 may take 1500 ns. This is because the memory has to cycle through locations 100 to 300, with each location requiring 5 ns.

The effectiveness of a memory hierarchy depends on the principle of moving information into the fast memory infrequently and accessing it many times before replacing it with new information. This principle is possible due to a phenomenon called *locality of reference*; that is, within a given period of time, programs tend to reference a relatively confined area of memory repeatedly. There exist two forms of locality: spatial and temporal locality. *Spatial locality* refers to the

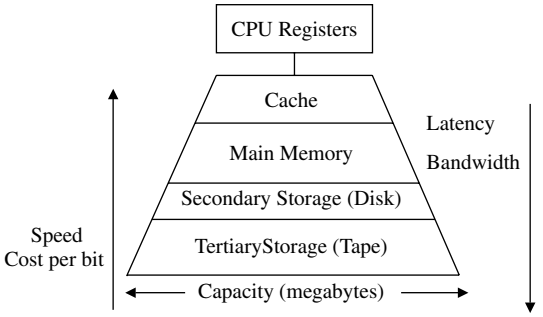

Figure 6.1 Typical memory hierarchy

TABLE 6.1 Memory Hierarchy Parameters

	Access type	Capacity	Latency	Bandwidth	Cost/MB
CPU registers	Random	64–1024 bytes	1–10 ns	System clock rate	High
Cache memory	Random	8–512 KB	15–20 ns	10–20 MB/s	$500
Main memory	Random	16–512 MB	30–50 ns	1–2 MB/s	$20–50
Disk memory	Direct	1–20 GB	10–30 ms	1–2 MB/s	$0.25
Tape memory	Sequential	1–20 TB	30–10,000 ms	1–2 MB/s	$0.025

phenomenon that when a given address has been referenced, it is most likely that addresses near it will be referenced within a short period of time, for example, consecutive instructions in a straightline program. *Temporal locality*, on the other hand, refers to the phenomenon that once a particular memory item has been referenced, it is most likely that it will be referenced next, for example, an instruction in a program loop.

The sequence of events that takes place when the processor makes a request for an item is as follows. First, the item is sought in the first memory level of the memory hierarchy. The probability of finding the requested item in the first level is called the *hit ratio*, h_1. The probability of not finding (missing) the requested item in the first level of the memory hierarchy is called the *miss ratio*, $(1 - h_1)$. When the requested item causes a "*miss*," it is sought in the next subsequent memory level. The probability of finding the requested item in the second memory level, the hit ratio of the second level, is h_2. The miss ratio of the second memory level is $(1 - h_2)$. The process is repeated until the item is found. Upon finding the requested item, it is brought and sent to the processor. In a memory hierarchy that consists of three levels, the average memory access time can be expressed as follows:

$$\begin{aligned} t_{\text{av}} &= h_1 \times t_1 + (1 - h_1)[t_1 + h_2 \times t_2 + (1 - h_2)(t_2 + t_3)] \\ &= t_1 + (1 - h_1)[t_2 + (1 - h_2)t_3] \end{aligned}$$

The average access time of a memory level is defined as the time required to access one word in that level. In this equation, t_1, t_2, t_3 represent, respectively, the access times of the three levels.

6.2. CACHE MEMORY

Cache memory owes its introduction to Wilkes back in 1965. At that time, Wilkes distinguished between two types of main memory: The conventional and the *slave memory*. In Wilkes terminology, a slave memory is a second level of unconventional high-speed memory, which nowadays corresponds to what is called *cache memory* (the term cache means a safe place for hiding or storing things).

The idea behind using a cache as the first level of the memory hierarchy is to keep the information expected to be used more frequently by the CPU in the cache

(a small high-speed memory that is near the CPU). The end result is that at any given time some active portion of the main memory is duplicated in the cache. Therefore, when the processor makes a request for a memory reference, the request is first sought in the cache. If the request corresponds to an element that is currently residing in the cache, we call that a cache hit. On the other hand, if the request corresponds to an element that is not currently in the cache, we call that a cache miss. A *cache hit ratio*, h_c, is defined as the probability of finding the requested element in the cache. A *cache miss ratio* $(1 - h_c)$ is defined as the probability of not finding the requested element in the cache.

In the case that the requested element is not found in the cache, then it has to be brought from a subsequent memory level in the memory hierarchy. Assuming that the element exists in the next memory level, that is, the main memory, then it has to be brought and placed in the cache. In expectation that the next requested element will be residing in the neighboring locality of the current requested element (spatial locality), then upon a cache miss what is actually brought to the main memory is a *block* of elements that contains the requested element. The advantage of transferring a block from the main memory to the cache will be most visible if it could be possible to transfer such a block using one main memory access time. Such a possibility could be achieved by increasing the rate at which information can be transferred between the main memory and the cache. One possible technique that is used to increase the bandwidth is *memory interleaving*. To achieve best results, we can assume that the block brought from the main memory to the cache, upon a cache miss, consists of elements that are stored in different memory modules, that is, whereby consecutive memory addresses are stored in successive memory modules. Figure 6.2 illustrates the simple case of a main memory consisting of eight memory modules. It is assumed in this case that the block consists of 8 bytes.

Having introduced the basic idea leading to the use of a cache memory, we would like to assess the impact of temporal and spatial locality on the performance of the memory hierarchy. In order to make such an assessment, we will limit our

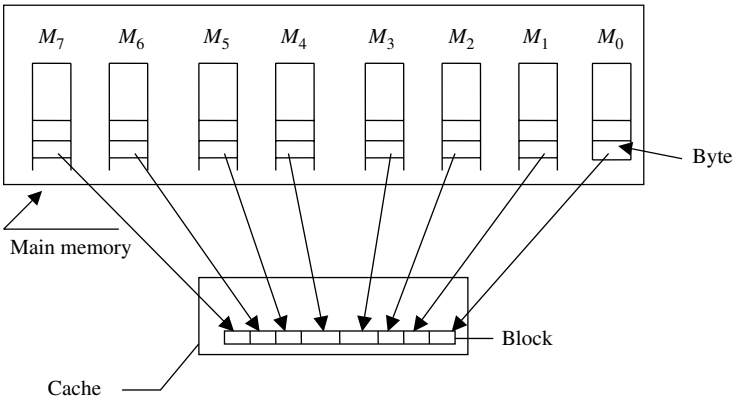

Figure 6.2 Memory interleaving using eight modules

deliberation to the simple case of a hierarchy consisting only of two levels, that is, the cache and the main memory. We assume that the main memory access time is t_m and the cache access time is t_c. We will measure the impact of locality in terms of the average access time, defined as the average time required to access an element (a word) requested by the processor in such a two-level hierarchy.

6.2.1. Impact of Temporal Locality

In this case, we assume that instructions in program loops, which are executed many times, for example, n times, once loaded into the cache, are used more than once before they are replaced by new instructions. The average access time, t_{av}, is given by

$$t_{av} = \frac{nt_c + t_m}{n} = t_c + \frac{t_m}{n}$$

In deriving the above expression, it was assumed that the requested memory element has created a cache miss, thus leading to the transfer of a main memory block in time t_m. Following that, n accesses were made to the same requested element, each taking t_c. The above expression reveals that as the number of repeated accesses, n, increases, the average access time decreases, a desirable feature of the memory hierarchy.

6.2.2. Impact of Spatial Locality

In this case, it is assumed that the size of the block transferred from the main memory to the cache, upon a cache miss, is m elements. We also assume that due to spatial locality, all m elements were requested, one at a time, by the processor. Based on these assumptions, the average access time, t_{av}, is given by

$$t_{av} = \frac{mt_c + t_m}{m} = t_c + \frac{t_m}{m}$$

In deriving the above expression, it was assumed that the requested memory element has created a cache miss, thus leading to the transfer of a main memory block, consisting of m elements, in time t_m. Following that, m accesses, each for one of the elements constituting the block, were made. The above expression reveals that as the number of elements in a block, m, increases, the average access time decreases, a desirable feature of the memory hierarchy.

6.2.3. Impact of Combined Temporal and Spatial Locality

In this case, we assume that the element requested by the processor created a cache miss leading to the transfer of a block, consisting of *m elements*, to the cache (that take t_m). Now, due to spatial locality, all m elements constituting a block were requested, one at a time, by the processor (requiring mt_c). Following that, the originally requested element was accessed $(n - 1)$ times (temporal locality), that is, a

total of n times access to that element. Based on these assumptions, the average access time, t_{av}, is given by

$$t_{av} = \frac{\left(\frac{mt_c + t_m}{m}\right) + (n-1)t_c}{n} = \frac{t_c + \left(\frac{t_m}{m}\right) + (n-1)t_c}{n} = \frac{t_m}{nm} + t_c$$

A further simplifying assumption to the above expression is to assume that $t_m = mt_c$. In this case the above expression will simplify to

$$t_{av} = \frac{mt_c}{nm} + t_c = t_c + \frac{t_c}{n} = \frac{n+1}{n} t_c$$

The above expression reveals that as the number of repeated accesses n increases, the average access time will approach t_c. This is a significant performance improvement.

It should be clear from the above discussion that as more requests for items that do not exist in the cache (cache miss) occur, more blocks would have to be brought to the cache. This should raise two basic questions: Where to place an incoming main memory block in the cache? And in the case where the cache is totally filled, which cache block should the incoming main memory block replace? Placement of incoming blocks and replacement of existing blocks are performed according to specific protocols (algorithms). These protocols are strongly related to the internal organization of the cache. Cache internal organization is discussed in the following subsections. However, before discussing cache organization, we first introduce the cache-mapping function.

6.2.4. Cache-Mapping Function

Without loss of generality, we present cache-mapping function taking into consideration the interface between two successive levels in the memory hierarchy: primary level and secondary level. If the focus is on the interface between the cache and main memory, then the cache represents the primary level, while the main memory represents the secondary level. The same principles apply to the interface between any two memory levels in the hierarchy. In the following discussion, we focus our attention to the interface between the cache and the main memory.

It should be noted that a request for accessing a memory element is made by the processor through issuing the address of the requested element. The address issued by the processor may correspond to that of an element that exists currently in the cache (cache hit); otherwise, it may correspond to an element that is currently residing in the main memory. Therefore, address translation has to be made in order to determine the whereabouts of the requested element. This is one of the functions performed by the memory management unit (MMU). A schematic of the address mapping function is shown in Figure 6.3.

In this figure, the system address represents the address issued by the processor for the requested element. This address is used by an address translation function inside the MMU. If address translation reveals that the issued address corresponds to an element currently residing in the cache, then the element will be made

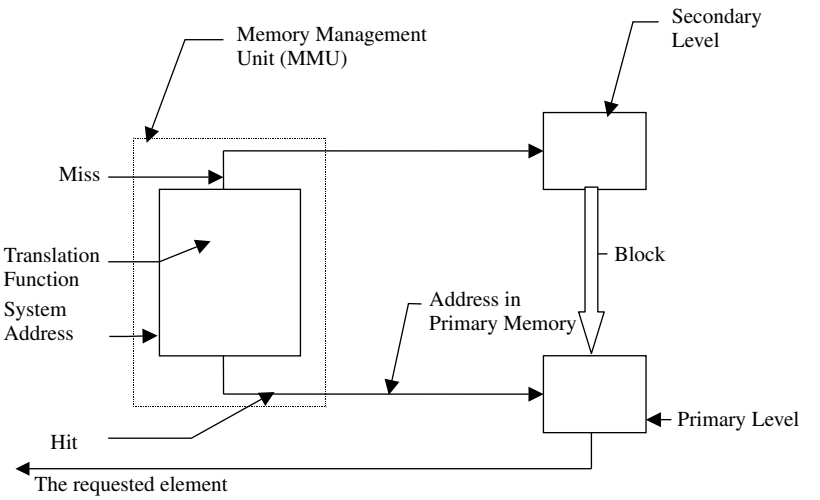

Figure 6.3 Address mapping operation

available to the processor. If, on the other hand, the element is not currently in the cache, then it will be brought (as part of a block) from the main memory and placed in the cache and the element requested is made available to the processor.

6.2.5. Cache Memory Organization

There are three main different organization techniques used for cache memory. The three techniques are discussed below. These techniques differ in two main aspects:

1. The criterion used to place, in the cache, an incoming block from the main memory.
2. The criterion used to replace a cache block by an incoming block (on cache full).

Direct Mapping This is the simplest among the three techniques. Its simplicity stems from the fact that it places an incoming main memory block into a specific fixed cache block location. The placement is done based on a fixed relation between the incoming block number, i, the cache block number, j, and the number of cache blocks, N:

$$j = i \bmod N$$

Example 1 Consider, for example, the case of a main memory consisting of 4K blocks, a cache memory consisting of 128 blocks, and a block size of 16 words. Figure 6.4 shows the division of the main memory and the cache according to the direct-mapped cache technique.

As the figure shows, there are a total of 32 main memory blocks that map to a given cache block. For example, main memory blocks 0, 128, 256, 384, . . . , 3968 map to cache block 0. We therefore call the direct-mapping technique a

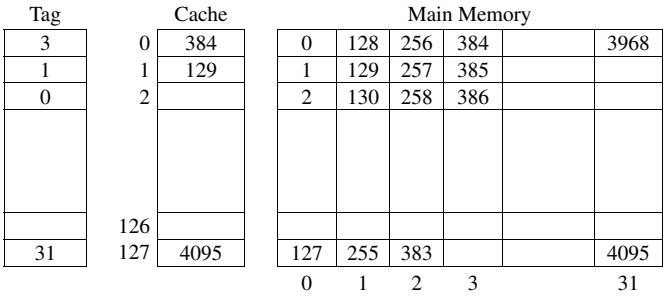

Figure 6.4 Mapping main memory blocks to cache blocks

many-to-one mapping technique. The main advantage of the direct-mapping technique is its simplicity in determining where to place an incoming main memory block in the cache. Its main disadvantage is the inefficient use of the cache. This is because according to this technique, a number of main memory blocks may compete for a given cache block even if there exist other empty cache blocks. This disadvantage should lead to achieving a low cache hit ratio.

According to the direct-mapping technique the MMU interprets the address issued by the processor by dividing the address into three fields as shown in Figure 6.5. The lengths, in bits, of each of the fields in Figure 6.5 are:

1. Word field $= \log_2 B$, where B is the size of the block in words.
2. Block field $= \log_2 N$, where N is the size of the cache in blocks.
3. Tag field $= \log_2 (M/N)$, where M is the size of the main memory in blocks.
4. The number of bits in the main memory address $= \log_2 (B \times M)$

It should be noted that the total number of bits as computed by the first three equations should add up to the length of the main memory address. This can be used as a check for the correctness of your computation.

Example 2 Compute the above four parameters for Example 1.

Word field $= \log_2 B = \log_2 16 = \log_2 2^4 = 4$ bits
Block field $= \log_2 N = \log_2 128 = \log_2 2^7 = 7$ bits
Tag field $= \log_2(M/N) = \log_2(2^2 \times 2^{10}/2^7) = 5$ bits

The number of bits in the main memory address $= \log_2 (B \times M) = \log_2 (2^4 \times 2^{12}) = 16$ bits.

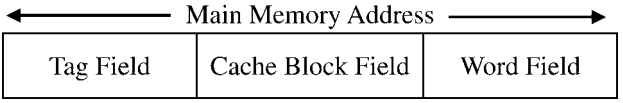

Figure 6.5 Direct-mapped address fields

Having shown the division of the main memory address, we can now proceed to explain the protocol used by the MMU to satisfy a request made by the processor for accessing a given element. We illustrate the protocol using the parameters given in the example presented above (Fig. 6.6).

The steps of the protocol are:

1. Use the *Block field* to determine the cache block that should contain the element requested by the processor. The *Block field* is used directly to determine the cache block sought, hence the name of the technique: direct-mapping.
2. Check the corresponding *Tag memory* to see whether there is a match between its content and that of the *Tag field*. A match between the two indicates that the targeted cache block determined in step 1 is currently holding the main memory element requested by the processor, that is, a *cache hit*.
3. Among the elements contained in the cache block, the targeted element can be selected using the *Word field*.
4. If in step 2, no match is found, then this indicates a *cache miss*. Therefore, the required block has to be brought from the main memory, deposited in the cache, and the targeted element is made available to the processor. The cache *Tag memory* and the cache block memory have to be updated accordingly.

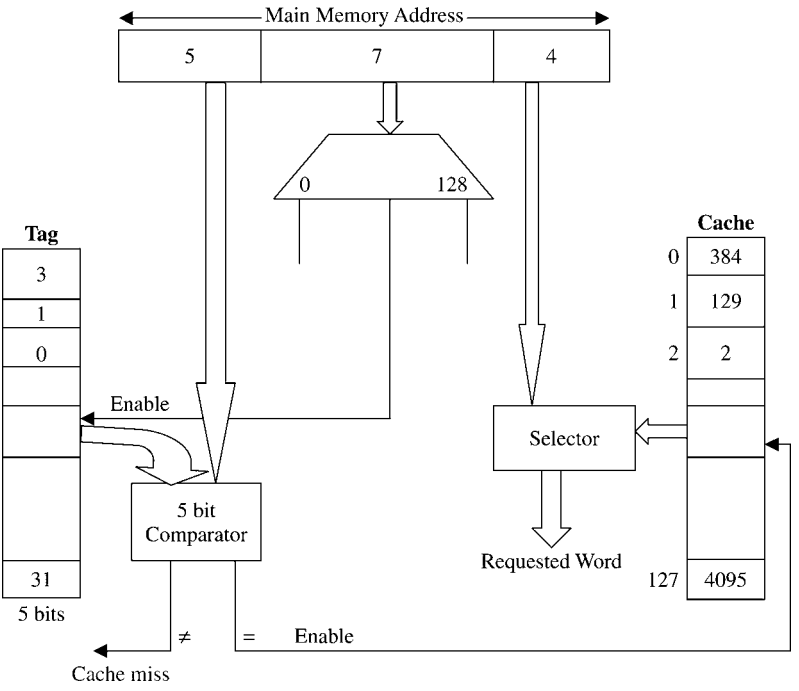

Figure 6.6 Direct-mapped address translation

The direct-mapping technique answers not only the placement of the incoming main memory block in the cache question, but it also answers the replacement question. Upon encountering a totally filled cache while a new main memory block has to be brought, the replacement is trivial and determined by the equation $j = i \bmod N$.

The main advantages of the direct-mapping technique is its simplicity measured in terms of the direct determination of the cache block; no search is needed. It is also simple in terms of the replacement mechanism. The main disadvantage of the technique is its expected poor utilization of the cache memory. This is represented in terms of the possibility of targeting a given cache block, which requires frequent replacement of blocks while the rest of the cache is not used. Consider, for example, the sequence of requests made by the processor for elements held in the main memory blocks 1, 33, 65, 97, 129, and 161. Consider also that the cache size is 32 blocks. It is clear that all the above blocks map to cache block number 1. Therefore, these blocks will compete for that same cache block despite the fact that the remaining 31 cache blocks are not used.

The expected poor utilization of the cache by the direct mapping technique is mainly due to the restriction on the placement of the incoming main memory blocks in the cache (the many-to-one property). If such a restriction is relaxed, that is, if we make it possible for an incoming main memory block to be placed in any empty (available) cache block, then the resulting technique would be so flexible that efficient utilization of the cache would be possible. Such a flexible technique, called the *Associative Mapping* technique, is explained next.

Fully Associative Mapping According to this technique, an incoming main memory block can be placed in any available cache block. Therefore, the address issued by the processor need only have two fields. These are the *Tag* and *Word* fields. The first uniquely identifies the block while residing in the cache. The second field identifies the element within the block that is requested by the processor. The MMU interprets the address issued by the processor by dividing it into two fields as shown in Figure 6.7. The length, in bits, of each of the fields in Figure 6.7 are given by:

1. Word field $= \log_2 B$, where B is the size of the block in words
2. Tag field $= \log_2 M$, where M is the size of the main memory in blocks
3. The number of bits in the main memory address $= \log_2 (B \times M)$

It should be noted that the total number of bits as computed by the first two equations should add up to the length of the main memory address. This can be used as a check for the correctness of your computation.

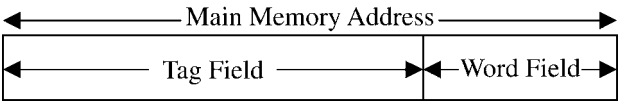

Figure 6.7 Associative-mapped address fields

Example 3 Compute the above three parameters for a memory system having the following specification: size of the main memory is 4K blocks, size of the cache is 128 blocks, and the block size is 16 words. Assume that the system uses associative mapping.

$$\text{Word field} = \log_2 B = \log_2 16 = \log_2 2^4 = 4 \text{ bits}$$
$$\text{Tag field} = \log_2 M = \log_2 2^7 \times 2^{10} = 12 \text{ bits}$$

The number of bits in the main memory address $= \log_2 (B \times M) = \log_2 (2^4 \times 2^{12}) = 16$ bits.

Having shown the division of the main memory address, we can now proceed to explain the protocol used by the MMU to satisfy a request made by the processor for accessing a given element. We illustrate the protocol using the parameters given in the example presented above (see Fig. 6.8). The steps of the protocol are:

1. Use the *Tag field* to search in the *Tag memory* for a match with any of the tags stored.
2. A match in the tag memory indicates that the corresponding targeted cache block determined in step 1 is currently holding the main memory element requested by the processor, that is, a *cache hit*.

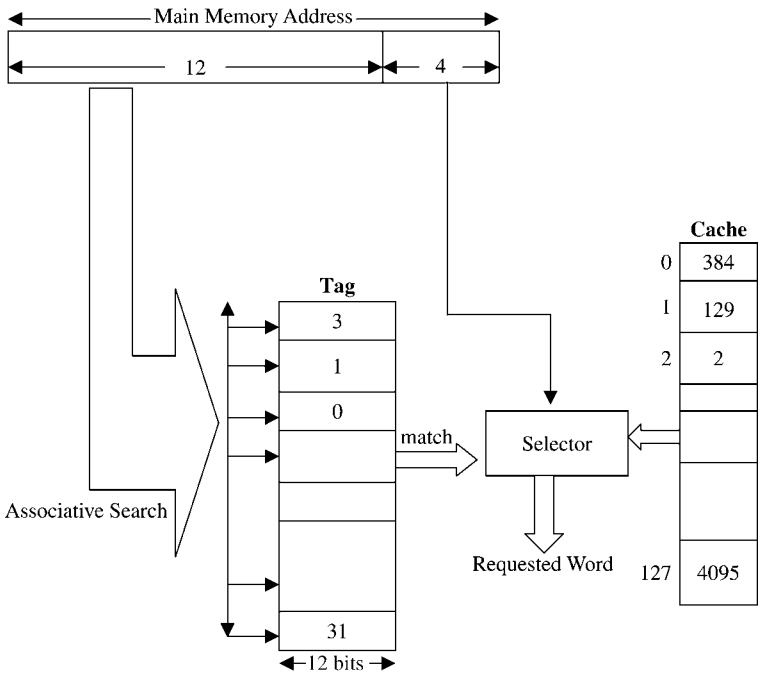

Figure 6.8 Associative-mapped address translation

3. Among the elements contained in the cache block, the targeted element can be selected using the *Word field*.
4. If in step 2, no match is found, then this indicates a *cache miss*. Therefore, the required block has to be brought from the main memory, deposited in the first available cache block, and the targeted element (word) is made available to the processor. The cache *Tag memory* and the cache block memory have to be updated accordingly.

It should be noted that the search made in step 1 above requires matching the *tag field* of the address with each and every entry in the *tag memory*. Such a search, if done sequentially, could lead to a long delay. Therefore, the *tags* are stored in an *associative (content addressable) memory*. This allows the entire contents of the tag memory to be searched in parallel (associatively), hence the name, associative mapping.

It should be noted that, regardless of the cache organization used, a mechanism is needed to ensure that any accessed cache block contains *valid* information. The validity of the information in a cache block can be checked via the use of a single bit for each cache block, called the *valid bit*. The valid bit of a cache block should be updated in such a way that if *valid bit* $= 1$, then the corresponding cache block carries valid information; otherwise, the information in the cache block is invalid. When a computer system is powered up, all valid bits are made equal to 0, indicating that they carry invalid information. As blocks are brought to the cache, their statuses are changed accordingly to indicate the validity of the information contained.

The main advantage of the associative-mapping technique is the efficient use of the cache. This stems from the fact that there exists no restriction on where to place incoming main memory blocks. Any unoccupied cache block can potentially be used to receive those incoming main memory blocks. The main disadvantage of the technique, however, is the hardware overhead required to perform the associative search conducted in order to find a match between the tag field and the tag memory as discussed above.

A compromise between the simple but inefficient direct cache organization and the involved but efficient associative cache organization can be achieved by conducting the search over a limited set of cache blocks while knowing ahead of time where in the cache an incoming main memory block is to be placed. This is the basis for the set-associative mapping technique explained next.

Set-Associative Mapping In the set-associative mapping technique, the cache is divided into a number of sets. Each set consists of a number of blocks. A given main memory block maps to a specific cache set based on the equation $s = i \bmod S$, where S is the number of sets in the cache, i is the main memory block number, and s is the specfic cache set to which block i maps. However, an incoming block maps to any block in the assigned cache set. Therefore, the address issued by the processor is divided into three distinct fields. These are the *Tag*, *Set*, and *Word* fields. The *Set* field is used to uniquely identify the specific cache set

that ideally should hold the targeted block. The *Tag* field uniquely identifies the targeted block within the determined set. The *Word* field identifies the element (word) within the block that is requested by the processor. According to the set-associative mapping technique, the MMU interprets the address issued by the processor by dividing it into three fields as shown in Figure 6.9. The length, in bits, of each of the fields of Figure 6.9 is given by

1. Word field $= \log_2 B$, where B is the size of the block in words
2. Set field $= \log_2 S$, where S is the number of sets in the cache
3. Tag field $= \log_2 (M/S)$, where M is the size of the main memory in blocks. $S = N/B_s$, where N is the number of cache blocks and B_s is the number of blocks per set
4. The number of bits in the main memory address $= \log_2 (B \times M)$

It should be noted that the total number of bits as computed by the first three equations should add up to the length of the main memory address. This can be used as a check for the correctness of your computation.

Example 4 Compute the above three parameters (Word, Set, and Tag) for a memory system having the following specification: size of the main memory is 4K blocks, size of the cache is 128 blocks, and the block size is 16 words. Assume that the system uses set-associative mapping with four blocks per set.

$$S = \frac{128}{4} = 32 \text{ sets.}$$

1. Word field $= \log_2 B = \log_2 16 = \log_2 2^4 = 4$ bits
2. Set field $= \log_2 32 = 5$ bits
3. Tag field $= \log_2 (4 \times 2^{10}/32) = 7$ bits

The number of bits in the main memory address $= \log_2 (B \times M) = \log_2 (2^4 \times 2^{12}) = 16$ bits.

Having shown the division of the main memory address, we can now proceed to explain the protocol used by the MMU to satisfy a request made by the processor for accessing a given element. We illustrate the protocol using the parameters given in the example presented above (see Fig. 6.10). The steps of the protocol are:

1. Use the *Set field* (5 bits) to determine (directly) the specified set (1 of the 32 sets).

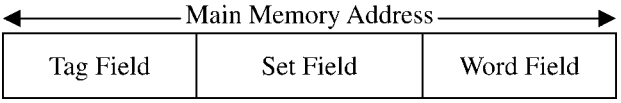

Figure 6.9 Set-associative-mapped address fields

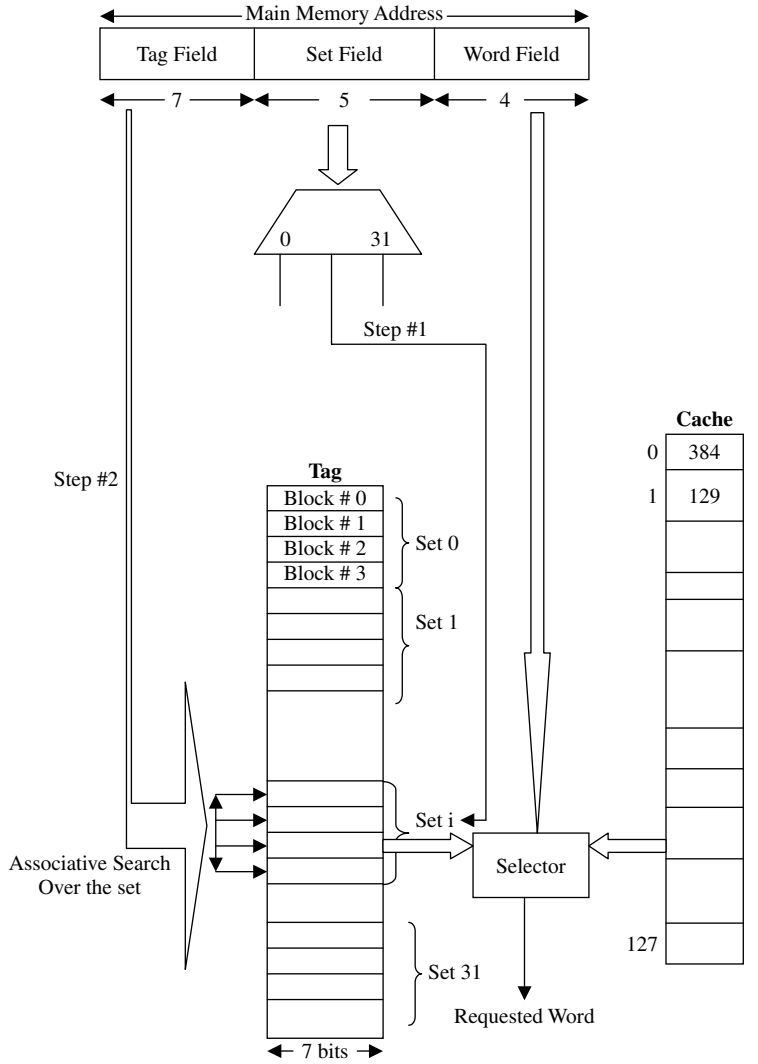

Figure 6.10 Set-associative-mapped address translation

2. Use the *Tag field* to find a match with any of the (four) blocks in the determined set. A match in the tag memory indicates that the specified set determined in step 1 is currently holding the targeted block, that is, a *cache hit*.
3. Among the 16 words (elements) contained in hit cache block, the requested word is selected using a selector with the help of the *Word field*.

4. If in step 2, no match is found, then this indicates a *cache miss*. Therefore, the required block has to be brought from the main memory, deposited in the specified set first, and the targeted element (word) is made available to the processor. The cache *Tag memory* and the cache block memory have to be updated accordingly.

It should be noted that the search made in step 2 above requires matching the *tag field* of the address with each and every entry in the *tag memory* for the specified set. Such a search is performed in parallel (associatively) over the set, hence the name, set-associative mapping. The hardware overhead required to performing the associative search within a set in order to find a match between the tag field and the tag memory is not as complex as that used in the case of the fully associative technique.

The set-associative-mapping technique is expected to produce a moderate cache utilization efficiency, that is, not as efficient as the fully associative technique and not as poor as the direct technique. However, the technique inherits the simplicity of the direct mapping technique in terms of determining the target set.

An overall qualitative comparison among the three mapping techniques is shown in Table 6.2. Owing to its moderate complexity and moderate cache utilization, the set-associative technique is used in the Intel Pentium line of processors.

The discussion above shows how the associative-mapping and the set-associative techniques answer the question about the placement of the incoming main memory block in the cache. The other important question that was posed at the beginning of the discussion on cache memory is that of replacement. Specifically, upon encountering a totally filled cache while a new main memory block has to be brought, which of the cache blocks should be selected for replacement? This is discussed below.

6.2.6. Replacement Techniques

A number of replacement techniques can be used. These include a randomly selected block (*random selection*), the block that has been in the cache the longest (*first-in-first-out, FIFO*), and the block that has been used the least while residing in the cache (*least recently used, LRU*).

Let us assume that when a computer system is powered up, a random number generator starts generating numbers between 0 and $(N - 1)$. As the name indicates,

TABLE 6.2 Qualitative Comparison Among Cache Mapping Techniques

Mapping technique	Simplicity	Associative tag search	Expected cache utilization	Replacement technique
Direct	Yes	None	Low	Not needed
Associative	No	Involved	High	Yes
Set-associative	Moderate	Moderate	Moderate	Yes

random selection of a cache block for replacement is done based on the output of the random number generator at the time of replacement. This technique is simple and does not require much additional overhead. However, its main shortcoming is that it does not take locality into consideration. Random techniques have been found effective enough such that they have been first used by Intel in its *iAPX* microprocessor series.

The FIFO technique takes the time spent by a block in the cache as a measure for replacement. The block that has been in the cache the longest is selected for replacement regardless of the recent pattern of access to the block. This technique requires keeping track of the lifetime of a cache block. Therefore, it is not as simple as the random selection technique. Intuitively, the FIFO technique is reasonable to use for straightline programs where locality of reference is not of concern.

According to the LRU replacement technique, the cache block that has been recently used the least is selected for replacement. Among the three replacement techniques, the LRU technique is the most effective. This is because the history of block usage (as the criterion for replacement) is taken into consideration. The LRU algorithm requires the use of a cache controller circuit that keeps track of references to all blocks while residing in the cache. This can be achieved through a number of possible implementations. Among these implementations is the use of counters. In this case each cache block is assigned a counter. Upon a cache hit, the counter of the corresponding block is set to 0, all other counters having a smaller value than the original value in the counter of the hit block are incremented by 1, and all counters having a larger value are kept unchanged. Upon a cache miss, the block whose counter is showing the maximum value is chosen for replacement, the counter is set to 0, and all other counters are incremented by 1.

Having introduced the above three cache mapping technique, we offer the following example, which illustrates the main observations made about the three techniques.

Example 5 Consider the case of a 4×8 two-dimensional array of numbers, A. Assume that each number in the array occupies one word and that the array elements are stored column-major order in the main memory from location 1000 to location 1031. The cache consists of eight blocks each consisting of just two words. Assume also that whenever needed, *LRU* replacement policy is used. We would like to examine the changes in the cache if each of the above three mapping techniques is used as the following sequence of requests for the array elements are made by the processor:

$a_{0,0}, a_{0,1}, a_{0,2}, a_{0,3}, a_{0,4}, a_{0,5}, a_{0,6}, a_{0,7}$
$a_{1,0}, a_{1,1}, a_{1,2}, a_{1,3}, a_{1,4}, a_{1,5}, a_{1,6}, a_{1,7}$

Solution The distribution of the array elements in the main memory is shown in Figure 6.11. Shown also is the status of the cache before the above requests were made.

Direct Mapping Table 6.3 shows that there were 16 cache misses (not a single cache hit) and that the number of replacements made is 12 (these are shown

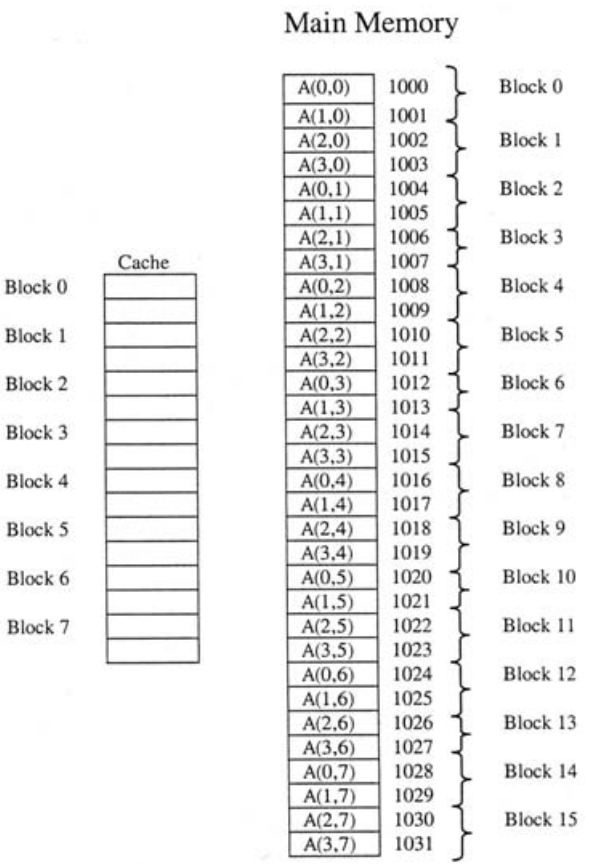

Figure 6.11 Array elements in the main memory

tinted). It also shows that out of the available eight cache blocks, only four (0, 2, 4, and 6) are used, while the remaining four are inactive all the time. This represents a 50% cache utilization.

Fully Associative Mapping Table 6.4 shows that there were eight cache hits (50% of the total number of requests) and that there were no replacements made. It also shows a 100% cache utilization.

Set-Associative Mapping (With Two Blocks per Set) Table 6.5 shows that there were 16 cache misses (not a single cache hit) and that the number of replacements made is 12 (these are shown tinted). It also shows that of the available four cache sets, only two sets are used, while the remaining two are inactive all the time. This represents a 50% cache utilization.

TABLE 6.3 Direct Mapping

Request	Cache hit/miss	MM block number (i)	Cache block number (j)	Cache status							
				BL0	BL1	BL2	BL3	BL4	BL5	BL6	BL7
A(0,0)	Miss	0	0	0 1 0 0							
A(0,1)	Miss	2	2	0 1 0 0		0 1 1 1					
A(0,2)	Miss	4	4	0 1 0 0		0 1 1 1		0 1 2 2			
A(0,3)	Miss	6	6	0 1 0 0		0 1 1 1		0 1 2 2		0 1 3 3	
A(0,4)	Miss	8	0	0 1 4 4		0 1 1 1		0 1 2 2		0 1 3 3	
A(0,5)	Miss	10	2	0 1 4 4		0 1 5 5		0 1 2 2		0 1 3 3	
A(0,6)	Miss	12	4	0 1 4 4		0 1 5 5		0 1 6 6		0 1 3 3	
A(0,7)	Miss	14	6	0 1 4 4		0 1 5 5		0 1 6 6		0 1 7 7	
A(1,0)	Miss	0	0	0 1 0 0		0 1 5 5		0 1 6 6		0 1 7 7	
A(1,1)	Miss	2	2	0 1 0 0		0 1 1 1		0 1 6 6		0 1 6 6	
A(1,2)	Miss	4	4	0 1 0 0		0 1 1 1		0 1 2 2		0 1 6 6	
A(1,3)	Miss	6	6	0 1 0 0		0 1 1 1		0 1 2 2		0 1 3 3	
A(1,4)	Miss	8	0	0 1 4 4		0 1 1 1		0 1 2 2		0 1 3 3	
A(1,5)	Miss	10	2	0 1 4 4		0 1 5 5		0 1 2 2		0 1 3 3	
A(1,6)	Miss	12	4	0 1 4 4		0 1 5 5		0 1 6 6		0 1 3 3	
A(1,7)	Miss	14	6	0 1 4 4		0 1 5 5		0 1 6 6		0 1 7 7	

6.2.7. Cache Write Policies

Having discussed the main issues related to cache mapping techniques and the replacement policies, we would like to address a very important related issue, that is, cache coherence. Coherence between a cache word and its copy in the main memory should be maintained at all times, if at all possible. A number of policies (techniques) are used in performing write operations to the main memory blocks while residing in the cache. These policies determine the degree of coherence that

TABLE 6.4 Fully Associative Mapping

Request	Cache hit/miss	MM block number (i)	Cache block number	Cache status															
				BL0		BL1		BL2		BL3		BL4		BL5		BL6		BL7	
A(0,0)	Miss	0	0	0	1														
				0	0														
A(0,1)	Miss	2	1	0	1	0	1												
				0	0	1	1												
A(0,2)	Miss	4	2	0	1	0	1	0	1										
				0	0	1	1	2	2										
A(0,3)	Miss	6	3	0	1	0	1	0	1	0	1								
				0	0	1	1	2	2	3	3								
A(0,4)	Miss	8	4	0	1	0	1	0	1	0	1	0	1						
				0	0	1	1	2	2	3	3	4	4						
A(0,5)	Miss	10	5	0	1	0	1	0	1	0	1	0	1	0	1				
				0	0	1	1	2	2	3	3	4	4	5	5				
A(0,6)	Miss	12	6	0	1	0	1	0	1	0	1	0	1	0	1	0	1		
				0	0	1	1	2	2	3	3	4	4	5	5	6	6		
A(0,7)	Miss	14	7	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
				0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
A(1,0)	Hit	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
				0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
A(1,1)	Hit	2	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
				0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
A(1,2)	Hit	4	2	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
				0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
A(1,3)	Hit	6	3	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
				0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
A(1,4)	Hit	8	4	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
				0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
A(1,5)	Hit	10	5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
				0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
A(1,6)	Hit	12	6	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
				0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
A(1,7)	Hit	14	7	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
				0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7

can be maintained between cache words and their counterparts in the main memory. In the following paragraphs, we discuss these write policies. In particular, we discuss two main cases: cache write policies upon a cache hit and the cache write policies upon a cache miss. We also discuss the cache read policy upon a cache miss. Cache read upon a cache hit is straightforward.

Cache Write Policies Upon a Cache Hit There are essentially two possible write policies upon a cache hit. These are the *write-through* and the *write-back*.

TABLE 6.5 Set-Associative Mapping

Request	Cache hit/miss	MM block number (i)	Cache block number	Cache status							
				Set # 0		Set # 1		Set # 2		Set # 3	
				BL0	BL1	BL2	BL3	BL4	BL5	BL6	BL7
A(0,0)	Miss	0	0	0	1						
				0	0						
A(0,1)	Miss	2	2	0	1			0	1		
				0	0			1	1		
A(0,2)	Miss	4	0	0	1	0	1	0	1		
				0	0	2	2	1	1		
A(0,3)	Miss	6	2	0	1	0	1	0	1	0	1
				0	0	2	2	1	1	3	3
A(0,4)	Miss	8	0	0	1	0	1	0	1	0	1
				4	4	2	2	1	1	3	3
A(0,5)	Miss	10	2	0	1	0	1	0	1	0	1
				4	4	2	2	5	5	3	3
A(0,6)	Miss	12	0	0	1	0	1	0	1	0	1
				4	4	6	6	5	5	3	3
A(0,7)	Miss	14	2	0	1	0	1	0	1	0	1
				4	4	6	6	5	5	7	7
A(1,0)	Miss	0	0	0	1	0	1	0	1	0	1
				0	0	6	6	5	5	7	7
A(1,1)	Miss	2	2	0	1	0	1	0	1	0	1
				0	0	6	6	1	1	7	7
A(1,2)	Miss	4	0	0	1	0	1	0	1	0	1
				0	0	2	2	1	1	7	7
A(1,3)	Miss	6	2	0	1	0	1	0	1	0	1
				0	0	2	2	1	1	3	3
A(1,4)	Miss	8	0	0	1	0	1	0	1	0	1
				4	4	2	2	1	1	3	3
A(1,5)	Miss	10	2	0	1	0	1	0	1	0	1
				4	4	2	2	5	5	3	3
A(1,6)	Miss	12	0	0	1	0	1	0	1	0	1
				4	4	6	6	5	5	3	3
A(1,7)	Miss	14	2	0	1	0	1	0	1	0	1
				4	4	6	6	4	4	2	2

In the write-through policy, every write operation to the cache is repeated to the main memory at the same time. In the write-back policy, all writes are made to the cache. A write to the main memory is postponed until a replacement is needed. Every cache block is assigned a bit, called the *dirty bit*, to indicate that at least one write operation has been made to the block while residing in the cache. At replacement time, the dirty bit is checked; if it is set, then the block is written back to the main memory, otherwise, it is simply overwritten by the incoming

block. The write-through policy maintains coherence between the cache blocks and their counterparts in the main memory at the expense of the extra time needed to write to the main memory. This leads to an increase in the average access time. On the other hand, the write-back policy eliminates the increase in the average access time. However, coherence is only guaranteed at the time of replacement.

Cache Write Policy Upon a Cache Miss Two main schemes can be used. These are *write-allocate* whereby the main memory block is brought to the cache and then updated. The other scheme is called *write-no-allocate* whereby the missed main memory block is updated while in the main memory and not brought to the cache.

In general, *write-through* caches use *write-no-allocate* policy while *write-back* caches use *write-allocate policy*.

Cache Read Policy Upon a Cache Miss Two possible strategies can be used. In the first, the main memory missed block is brought to the cache while the required word is forwarded immediately to the CPU as soon as it is available. In the second strategy, the missed main memory block is entirely stored in the cache and the required word is then forwarded to the CPU.

Having discussed the issues related to the design and analysis of cache memory, we briefly present formulae for the average access time of a memory hierarchy under different cache write policies.

Case No. 1: Cache Write-Through Policy

Write-allocate In this case, the average access time for a memory system is given by

$$t_a = t_c + (1 - h)t_b + w(t_m - t_c)$$

where t_b is the time required to transfer a block to the cache, $(t_m - t_c)$ is the additional time incurred due to the write operations, w is the fraction of write operations. It should be noted that if the data path and organization allow, then $t_b = t_m$; otherwise, $t_b = Bt_m$, where B is the block size in words.

Write-no-allocate In this case, the average access time can be expressed as

$$t_a = t_c + (1 - w)(1 - h)t_b + w(t_m - t_c)$$

Case No. 2: Cache Write-Back Policy The average access time for a system that uses a write-back policy is given by $t_a = t_c + (1 - h)t_b + w_b(1 - h)t_b$, where w_b is the probability that a block has been altered while being in the cache.

6.2.8. Real-Life Cache Organization Analysis

Intel's Pentium IV Processor Cache Intel's Pentium 4 processor uses a two-level cache organization as shown schematically in Figure 6.12. In this figure, L1 represents an 8 KB data cache. This is a four-way set-associative. The block size is 64 bytes. Consider the following example (tailored after the L1 Pentium cache).

Example 6

Cache organization	Set-associative
Main Memory size	16 MB
Cache L1 size	8 KB
Number of blocks per set	Four
CPU addressing	Byte addressable

The main memory address should be divided into three fields: Word, Set, and Tag (Fig. 6.13). The length of each field is computed as follows.

Number of main memory blocks $M = 2^{24}/2^6 = 2^{18}$ blocks
Number of cache blocks $N = 2^{13}/2^6 = 128$ blocks
$S = 128/4 = 32$ sets
Set field $= \log_2 32 = 5$ bits
Word field $= \log_2 \mathrm{B} = \log_2 64 = \log_2 2^6 = 6$ bits
Tag field $= \log_2 (2^{18}/2^5) = 13$ bits
Main memory address $= \log_2 (B \times M) = \log_2 (2^6 \times 2^{18}) = 24$ bits

The second cache level in Figure 6.11 is L2. This is called the *advanced transfer cache*. It is organized as an eight-way set-associative cache having a 256 KB total size and 128-byte block size. Following a similar set of steps as shown above for the L1 level, we obtain the following:

Number of main memory blocks $M = 2^{24}/2^7 = 2^{17}$ blocks

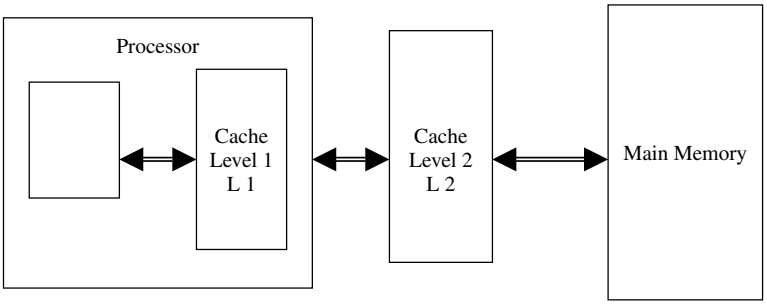

Figure 6.12 Pentium IV two-level cache

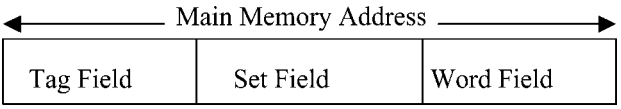

Figure 6.13 Division of main memory address

Number of cache blocks $N = 2^{18}/2^7 = 2^{11}$ blocks
$S = 2^{11}/2^3 = 2^8$ sets
Set field $= \log_2 2^8 = 8$ bits
Word field $= \log_2 B = \log_2 128 = \log_2 2^7 = 7$ bits
Tag field $= \log_2 (2^{17}/2^8) = 9$ bits

The following tables summarize the L1 and L2 Pentium 4 cache performance in terms of the cache hit ratio and cache latency.

CPU	L1 Hit ratio	L2 Hit ratio	L1 Latency	L2 Latency	Average latency
Pentium 4 at 1.5 GHz	90%	99%	1.33 ns	6.0 ns	1.8 ns

PowerPC 604 Processor Cache The PowerPC cache is divided into data and instruction caches, called *Harvard Organization*. Both the instruction and the data caches are organized as 16 KB four-way set-associative. The following table summarizes the PowerPC 604 cache basic characteristics.

Cache organization	Set-associative
Block size	32 bytes
Main memory size	4 GB ($M = 128$ Mega blocks)
Cache size	16 KB ($N = 512$ blocks)
Number of blocks per set	Four
Number of cache sets (S)	128 sets

The main memory address should be divided into three fields: Word, Set, and Tag (Fig. 6.13). The length of each field is computed as follows:

Number of main memory blocks $M = 2^{32}/2^5 = 2^{27}$ blocks
Number of cache blocks $N = 2^{14}/2^5 = 512$ blocks
$S = 512/4 = 128$ sets
Set field $= \log_2 128 = 7$ bits
Word field $= \log_2 B = \log_2 32 = \log_2 2^5 = 5$ bits
Tag field $= \log_2 (2^{27}/2^7) = 20$ bits
Main memory address $= \log_2 (B \times M) = \log_2 (2^5 \times 2^{27}) = 32$ bits

PMC-Sierra RM7000A 64-bit MIPS RISC Processor The RM7000 uses a different cache organization compared to that of the Intel and the PowerPC. In this case, three separate caches are included. These are:

1. Primary instruction cache: A 16 KB, four-way set-associative cache with 32-byte block size (eight instructions)
2. Primary data cache: A 16 KB, four-way set-associative cache with 32 bytes block size (eight words)
3. Secondary cache: A 256 KB, four-way set-associative cache for both instructions and data

In addition to the three on-chip caches, the RM7000 provides a dedicated tertiary cache interface, which supports tertiary cache sizes of 512 KB, 2 MB, and 8 MB. This tertiary cache is only accessed after a secondary cache miss.

The primary caches require one cycle each to access. Each of these caches has 64-bit read data path and 128-bit write data path. Both caches can be accessed simultaneously, giving an aggregate bandwidth of over 4 GB per second. The secondary cache has a 64-bit data path and is accessed only on a primary cache miss. It has a three-cycle miss penalty. Owing to the unusual cache organization of the RM7000, it uses the two cache access schemes described below.

Non-Blocking Caches In this scheme, the caches do not stall on a miss, rather the processor continues to operate out of the primary caches until one of the following events takes place:

1. Two cache misses are outstanding and a third load/store instruction appears on the instruction bus.
2. A subsequent instruction requires data from either of the instructions that caused a cache miss.

The use of nonblocking caches improves the overall performance by allowing the cache to continue operating even though a cache miss has occurred.

Cache Locking In this scheme, critical code or data segments are locked into the primary and secondary caches. The locked contents can be updated on a write hit, but cannot be selected for replacement on a miss. RM7000 allows each of the three caches to be locked separately. However, only two of the available four sets of each cache can be locked. In particular, RM7000 allows a maximum of 128 KB of data or code to be locked in the secondary cache, a maximum of 8 KB of code to be locked in the instruction cache, and a maximum of 8 KB of data to be locked in the data cache.

6.3. SUMMARY

In this chapter, we consider the design and analysis of the first level of a memory hierarchy, that is, the cache memory. In this context, the locality issues were

discussed and their effect on the average access time was explained. Three cache mapping techniques, namely direct, associative, and set-associative mappings were analyzed and their performance measures compared. We have also introduced three replacement techniques: Random, FIFO, and LRU replacement. The impact of the three techniques on the cache hit ratio was analyzed. Cache writing policies were also introduced and analyzed. Our discussion on cache ended with a presentation of the cache memory organization and characteristics of three real-life examples: Pentium IV, PowerPC, and PMC-Sierra RM7000, processors. In Chapter 7, we will discuss the issues related to the design aspects of the internal and external organization of the main memory. We will also discuss the issues related to virtual memory.

EXERCISES

1. Determine the memory interleaving factor required to obtain an average access time of less that 60 ns given that the main memory has an access time of 100 ns and the cache has an access time of 20 ns. What is the average access time of the resulting system?
2. What is the average access time of a system having three levels of memory, a cache memory, a semiconductor main memory, and a magnetic disk secondary memory, if the access times of the memories are 20 ns, 100 ns, and 1 ms, respectively. The cache hit ratio is 90% and the main memory hit ratio is 95%.
3. A computer system has an MM consisting of 16 MB 32-bit words. It also has an 8 KB cache. Assume that the computer uses a byte-addressable mechanism. Determine the number of bits in each field of the address in each of the following organizations:
   (a) Direct mapping with block size of one word
   (b) Direct mapping with a block size of eight words
   (c) Associative mapping with a block size of eight words
   (d) Set-associative mapping with a set size of four block and a block size of one word.
4. Consider the execution of the following program segment on an 8×8 array A.

```
For i: = 0 to 7 do
SUM: = 0
    For j: = 0 to 7 do
        SUM: = SUM + A(i,j)
    End for
AVE(i): = SUM/8
End for
```

   Assume that the main memory is divided into eight interleaved memory blocks and that each cache memory block consists of eight elements. Assume also that the cache memory access time is 10 ns and that the memory access time is ten times the cache memory access time. Compute the average access time per element of the array A.

5. Consider the execution of the following program segment on a 4×10 array A. The two-dimensional array A is stored in the main memory in a column-major order. Assume that there are eight blocks in the cache, each is just one word, and that the LRU is used for replacement. Show a trace of the contents of the cache memory blocks for the different values of indices j and k assuming three different cache memory organizations, that is, *direct*, *associative*, and *set-associative mapping*. Provide your observations on the results obtained.

```
SUM: = 0
For j: = 0 to 9 do
    SUM: = SUM + A(0,j)
End for
AVE: = SUM/10
For k: = 0 to 9 do
    A(0,k): = A(0, k)/AVE
End for
```

6. Consider the case of a 4×8 two-dimensional array of numbers, A. Assume that each number in the array occupies one word and that the array elements are stored row-major in the main memory for location 1000 to location 1031. The cache consists of eight blocks each consisting of four words. Assume also that whenever needed, LRU replacement policy is used. We would like to examine the changes in the cache if each of the above three mapping techniques is used as the following sequence of requests for the array elements is made:

   $a_{0,0}, a_{0,1}, a_{0,2}, a_{0,3}, a_{0,4,}, a_{0,5}, a_{0,6}, a_{0,7}$
   $a_{1,0}, a_{1,1}, a_{1,2}, a_{1,3}, a_{1,4}, a_{1,5}, a_{1,6}, a_{1,7}$
   $a_{2,0}, a_{2,1}, a_{2,2}, a_{2,3}, a_{2,4}, a_{2,5}, a_{2,6}, a_{2,7}$
   $a_{3,0}, a_{3,1}, a_{3,2}, a_{3,3}, a_{3,4}, a_{3,5}, a_{3,6}, a_{3,7}$

   Show the status of the cache before and after the given requests were made, the number of replacements made, and an estimate of the cache utilization.

7. A computer system has an MM consisting of 1 M 16-bit words. It also has a 4 K word cache organized in the block-set-associative manner, with four blocks per set and 64 words per block. Assume that the cache is initially empty. Suppose that the CPU fetches 4352 words from locations 0, 1, 2, . . . , 4351 (in that order). It then repeats this fetch sequence nine more times. If the cache is 10 times faster than the MM, estimate the improvement factor resulting from the use of the cache. Assume that whenever a block is to be brought from the MM and the correspondence set in the cache is full, the new block replaces the least recently used block of this set. Repeat for the case of using the *most recently used* replacement technique; that is, if the cache is full, then the new block will replace the most recently used block in the cache. Note: This example is quoted from Reference #3.

REFERENCES AND FURTHER READING

S. D. Burd, *Systems Architecture*, 3rd ed., Thomson Learning Ltd, Boston, USA, 2001.

H. Cragon, *Memory Systems and Pipelined Processors*, Jones and Bartlett: Sudbury, MA, 1996.

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

J. L. Hennessy, and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, CA, 1996.

V. P. Heuring, and H. F. Jordan, *Computer Systems Design and Architecture*, Addison-Wesley, NJ, USA, 1997.

D. A. Patterson, and J. L. Hennessy, *Computer Organization & Design: The Hardware/Software Interface*, Morgan Kaufmann, San Mateo, CA, 1994.

H. S. Stone, *High-Performance Computer Architecture*, Addison-Wesley, Amsterdam, Netherlands, 1987.

M. Wilkes, Slave memories and dynamic storage allocation, IEEE Trans. Electron. Comput., EC-14(2), 270–271, (1965).

B. Wilkinson, *Computer Architecture: Design and Performance*, Prentice-Hall, Hertfordshire, UK, 1996.

Websites

http://www.sysopt.com

http://www.intel.com

http://www.AcerHardware.com

http://www.pmc-sierra.com/products/details/rm7000a

http://physinfo.ulb.ac.be/divers_html/PowerPC_Programming_Info/into_to_ppc/ppc2_hardware.html