Locality of reference :

Analysis of a large number of typical programs has shown that the references to memory at any given interval of time tend to be confined within a few localized areas in memory. This phenomenon is known as the property of *locality of reference*. The reason for this property may be understood considering that a typical computer program flows in a straight-line fashion with program loops and subroutine calls encountered frequently. When a program loop is executed, the CPU repeatedly refers to the set of instructions in memory that constitute the loop. Every time a given subroutine is called, its set of instructions are fetched from memory. Thus loops and subroutines tend to localize the references to memory for fetching instructions. To a lesser degree, memory references to data also tend to be localized. Table-lookup procedures repeatedly refer to that portion in memory where the table is stored. Iterative procedures refer to common memory locations and array of numbers are confined within a local portion of memory. The result of all these observations is the locality of reference property, which states that over a short interval of time, the addresses generated by a typical program refer to a few localized areas of memory repeatedly, while the remainder of memory is accessed relatively infrequently.

A distinction is made in the literature between spatial locality and temporal locality. **Spatial locality** refers to the tendency of execution to involve a number of memory locations that are clustered. This reflects the tendency of a processor to access instructions sequentially. Spatial location also reflects the tendency of a program to access data locations sequentially, such as when processing a table of data. **Temporal locality** refers to the tendency for a processor to access memory locations that have been used recently. For example, when an iteration loop is executed, the processor executes the same set of instructions repeatedly.

Cache :

If the active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. Such a fast small memory is referred to as a *cache memory*. It is placed between the CPU and main memory as illustrated in Fig. 12-1. The cache memory access time is less than the access time of main memory by a factor of 5 to 10. The cache is the fastest component in the memory hierarchy and approaches the speed of CPU components.

The fundamental idea of cache organization is that by keeping the most frequently accessed instructions and data in the fast cache memory, the aver-

age memory access time will approach the access time of the cache. Although the cache is only a small fraction of the size of main memory, a large fraction of memory requests will be found in the fast cache memory because of the locality of reference property of programs.

The basic operation of the cache is as follows. When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. A block of words containing the one just accessed is then transferred from main memory to cache memory.

The performance of cache memory is frequently measured in terms of a quantity called *hit ratio*. When the CPU refers to memory and finds the word in cache, it is said to produce a *hit*. If the word is not found in cache, it is in main memory and it counts as a *miss*. The ratio of the number of hits divided by the total CPU references to memory (hits plus misses) is the hit ratio. The hit ratio is best measured experimentally by running representative programs in the computer and measuring the number of hits and misses during a given interval of time. Hit ratios of 0.9 and higher have been reported. This high ratio verifies the validity of the locality of reference property.

Operation of Two-Level Memory

The locality property can be exploited in the formation of a two-level memory. The upper-level memory (M1) is smaller, faster, and more expensive (per bit) than the lower-level memory (M2). M1 is used as a temporary store for part of the contents of the larger M2. When a memory reference is made, an attempt is made to access the item in M1. If this succeeds, then a quick access is made. If not, then a block of memory locations is copied from M2 to M1 and the access then takes place via M1. Because of locality, once a block is brought into M1, there should be a number of accesses to locations in that block, resulting in fast overall service.

To express the average time to access an item, we must consider not only the speeds of the two levels of memory, but also the probability that a given reference can be found in M1. We have

$$\begin{aligned} T_s &= H \times T_1 + (1 - H) \times (T_1 + T_2) \\ &= T_1 + (1 - H) \times T_2 \end{aligned} \tag{4.1}$$

where

T_s = average (system) access time
T_1 = access time of M1 (e.g., cache, disk cache)
T_2 = access time of M2 (e.g., main memory, disk)
H = hit ratio (fraction of time reference is found in M1)

Figure 4.2 shows average access time as a function of hit ratio. As can be seen, for a high percentage of hits, the average total access time is much closer to that of M1 than M2.

Performance

Let us look at some of the parameters relevant to an assessment of a two-level memory mechanism. First consider cost. We have

$$C_s = \frac{C_1 S_1 + C_2 S_2}{S_1 + S_2} \tag{4.2}$$

where

C_s = average cost per bit for the combined two-level memory
C_1 = average cost per bit of upper-level memory M1
C_2 = average cost per bit of lower-level memory M2
S_1 = size of M1
S_2 = size of M2

We would like $C_s \approx C_2$. Given that $C_1 \gg C_2$, this requires $S_1 \ll S_2$.

Figure 4.4 depicts the structure of a cache/main-memory system. Main memory consists of up to 2^n addressable words, with each word having a unique

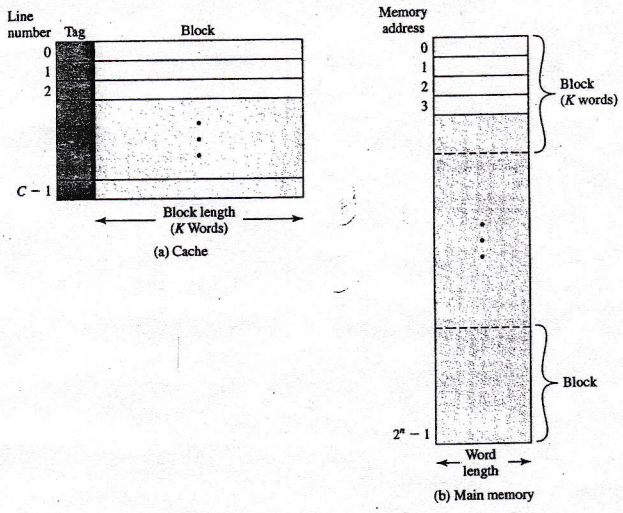


Figure 4.4 Cache/Main Memory Structure

n-bit address. For mapping purposes, this memory is considered to consist of a number of fixed-length blocks of K words each. That is, there are $M = 2^n/K$ blocks. The cache consists of C **lines.** Each line contains K words, plus a tag of a few bits; the number of words in the line is referred to as the **line size.** The number of lines is considerably less than the number of main memory blocks ($C \ll M$). At any time, some subset of the blocks of memory resides in lines in the cache. If a word in a block of memory is read, that block is transferred to one of the lines of the cache. Because there are more blocks than lines, an individual line cannot be uniquely and permanently dedicated to a particular block. Thus, each line includes a **tag** that identifies which particular block is currently being stored. The tag is usually a portion of the main memory address, as described later in this section.

Direct Mapping : The simplest technique, known as direct mapping, maps each block of main memory into only one possible cache line. The jth block of the main memory will reside at ith line of the cache, where

$$i = j \text{ modulo } m$$

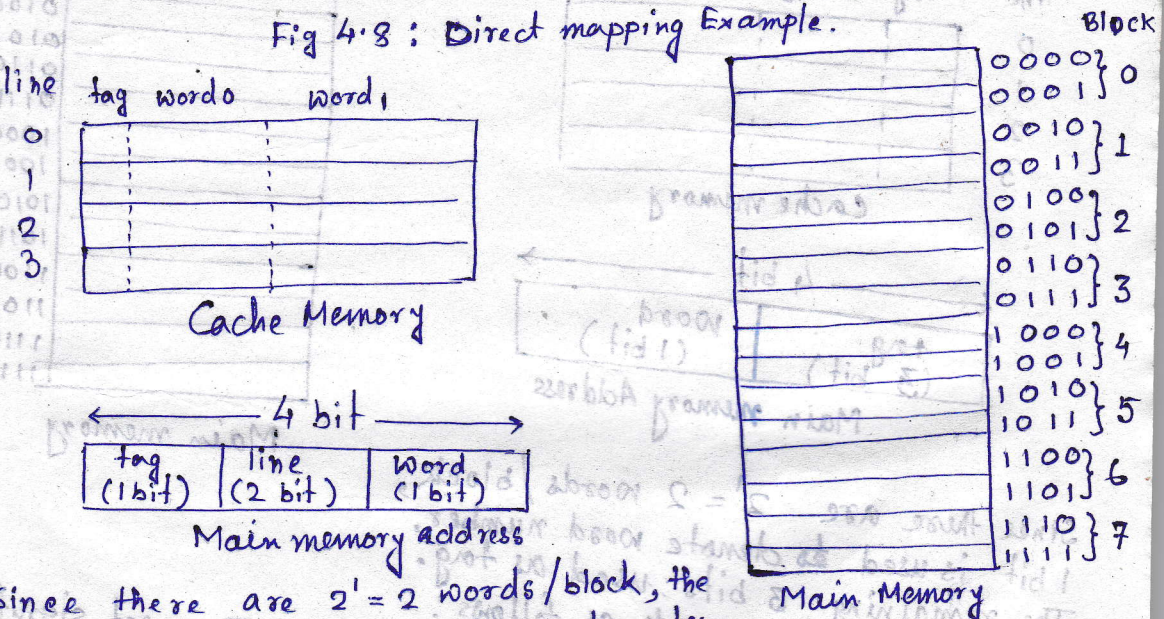
where m is the number of lines in the cache.

For purpose of cache access, each main memory address can be viewed as consisting of three fields. The least significant w bits identify a unique word within a block of main memory. The cache logic interprets the remaining s bits as a tag of s-r bits (most significant portion) and line field of r bits. The later field identifies one of the 2^r lines of the cache.

Figure 4.8 shows our example system using direct mapping. In the example, main memory consists of 16 words, with 2-words per block. Cache consists of 4 lines. The mapping becomes

Cache line	block no.
0	0, 4
1	1, 5
2	2, 6
3	3, 7.

Fig 4.8 : Direct mapping Example.



Since there are $2^1 = 2$ words/block, the least significant 1 bit denotes the word number. The cache consists of $2^2 = 4$ lines. Thus 2 bits is used to identify line number. The remaining 1 bit is used as tag.

A read operation work as follows:

A word is reffered by its 4 bit address. The middle 2-bit line number is used as an index into the cache to access a particular line. If the 1 bit tag number currently stored in that line matches with the tag of reffered address, the one bit word number is used to select one of the two words in that line. otherwise the reffered page brought into cache with

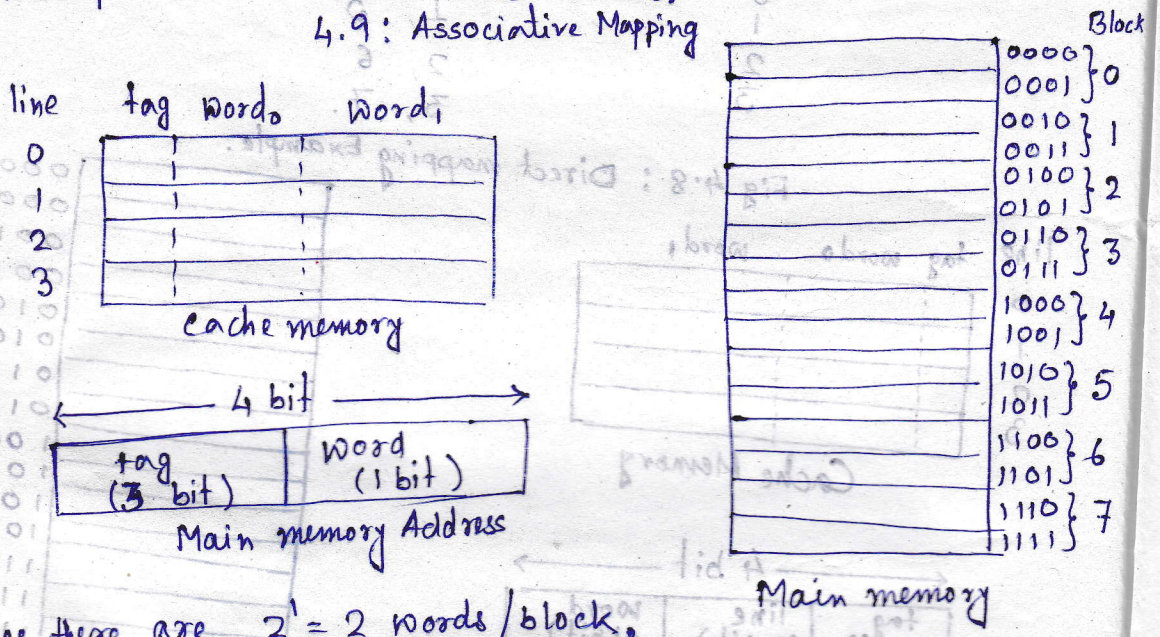
The direct mapping technique is simple and inexpensive to implement. Its main disadvantage is that there is a fixed cache location for any given block. Thus if a program happens to reference word repeatedly from two different block that maps into the same line, then the blocks will be continuously swapped in and out in the cache, and the hit ratio will be low.

Associative Mapping: Associative mapping overcomes the disadvantage of direct mapping by permitting each main memory block to be loaded into any line of the cache. In this case, the main memory address consists of two parts: tag field & word field. The tag field uniquely identify a block of main memory. To determine a block is in the cache, all the cache lines are examined simultaneously.

Each main memory address can be viewed as consisting of two field; tag & word. The least significant w bits identify a unique word within a block of main memory. The remaining s bits is used as tag.

Figure 4.9 shows our example system using associative mapping. In this example main memory consists of 16 words with 2 word per block. Cache consists 4 lines.

4.9: Associative Mapping



Since there are $2^1 = 2$ words/block, 1 bit is used to denote word number. The remaining 3 bits used as tag.

A read operation work as follows:

An word is reffered by its 4-bit address. The most significant 3 bits is searched in all the cache line tag, simultaneously for a match. If a match occurs, the 1 bit word number is used to select one of the two words in that line. Otherwise the reffered page brought into the cache with its tag, in any free line of the cache. If no free line is available, one of the cache line is selected by replacement algorithm and the line is replaced by currently reffered block.

Though Associative mapping solve the disadvantages of direct mapping, the principal disadvantage of associative mapping is the complex circuitry required to examine the tags of all cache lines in parallel.

Set-Associative Mapping : Set-associative mapping is compromise that exhibits the strength of both the direct and associative mapping while reducing their disadvantages. In this case, the cache is divided into v sets, each of which consists of K lines. Thus

$$m = vK$$, where m is the number of lines in the cache

This is referred to as k-way set associative mapping. The Block B_j can be mapped into any line of set i, where

$$i = j \text{ module } v$$

In this case main memory address consists of three fields: tag, set and word. The d set bits specify one of the $v = 2^d$ sets. The w bits identify a unique word within a block. The remain s bits is used as tag.

Writing into Cache

An important aspect of cache organization is concerned with memory write requests. When the CPU finds a word in cache during a read operation, the main memory is not involved in the transfer. However, if the operation is a write, there are two ways that the system can proceed.

Write through: The simplest and most commonly used procedure is to update main memory with every memory write operation, with cache memory being updated in parallel if it contains the word at the specified address. This is called the *write-through* method. This method has the advantage that main memory always contains the same data as the cache. This characteristic is important in systems with direct memory access transfers. It ensures that the data residing in main memory are valid at all times so that an I/O device communicating through DMA would receive the most recent updated data.

Write Back: The second procedure is called the *write-back* method. In this method only the cache location is updated during a write operation. The location is then marked by a flag so that later when the word is removed from the cache it is copied into main memory. The reason for the write-back method is that during the time a word resides in the cache, it may be updated several times; however, as long as the word remains in the cache, it does not matter whether the copy in main memory is out of date, since requests from the word are filled from the cache. It is only when the word is displaced from the cache that an accurate copy need be rewritten into main memory. Analytical results indicate that the number of memory writes in a typical program ranges between 10 and 30 percent of the total references to memory.

Virtual Memory:

In a memory hierarchy system, programs and data are first stored in auxiliary memory. Portions of a program or data are brought into main memory as they are needed by the CPU. *Virtual memory* is a concept used in some large computer systems that permit the user to construct programs as though a large memory space were available, equal to the totality of auxiliary memory. Each address that is referenced by the CPU goes through an address mapping from the so-called virtual address to a physical address in main memory. Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory. A virtual memory system provides a mechanism for translating program-generated addresses into correct main memory locations. This is done dynamically, while programs are being executed in the CPU. The translation or mapping is handled automatically by the hardware by means of a mapping table.