Chapter 2:

Memory System Architecture:

The memory of computer system can be divided into three main groups:

1. Internal processer memory
   This comprises a small set of high-speed registers used as a working memory for temporary storage of instructions and data.
2. Main memory
   This is a relatively large fast memory used for program and data storage during execution.
3. Secondary memory
   This is generally much larger in capacity but also much slower than main memory. It is use for storing system program and large data files. This type of memory has the following groups:
   - Magnetic tape
   - Floppy disk
   - Hard disk
   - CD-ROM
   - Flash

Read Only Memory (ROM)

ROM type:

1. Programmable ROM(PROM)
2. Erasable PROM(EPROM)
3. Electrically EPROM(EEPROM)

Random Access Memory (RAM)

A memory unit, in which any particular word can be accessed independently, is known as a *random-access memory* (RAM). In a random-access memory the time required for accessing a word is the same for all words. CPU can read and write. RAM is a volatile which main that it loses their information content wherever the power is turned off, thus it used as temporary storage.

Semiconductor RAM

There are two types of memories: ***static*** **(SRAM) and** ***dynamic*** **(DRAM)**.

SRAM

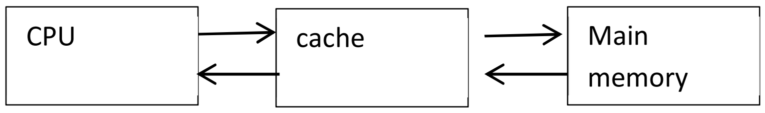
Static memories hold the stored data for long time as the power is on, or until new data are stored in them. This memory used mostly in CPU register and other high speed device, although some computers use them for caches and main memory. It is currently the faster and most expensive of the semiconductor memory.

DRAM

There devices are made with cells that store data as charge of capacitors together with a single transistor, unfortunately the capacitor slowly lose their charges due to leakage so periodic charge, refreshing is necessary to maintain data storage.
The implementation of the refreshing circuit may appear as a disadvantage for DRAM design; but, in general, a cell of a static memory requires more transistors than a cell of a dynamic memory. So the refreshing circuit is considered an acceptable and unavoidable cost of DRAM.

Cache Memory

One problem designers face in constructing a processor is the bottleneck associated with memory speeds. Because fetches from main memory require considerably more time when compared to the overall speeds in the processor, designers spend a lot of time and effort making memory speeds as fast as possible.
One way to make the memory appear faster is to reduce the number of times main memory has to be accessed. If a small amount of fast memory is installed and at any point in time part of a program is loaded in this fast memory, then, due to the property of locality of reference, the number of references to the main memory will be reduced. Such a fast memory unit, used temporarily to store a portion of the data and instructions (from the main memory) for immediate use, is known as *cache memory*.
Cache memory is a small fast memory placed between a processor and main memory.



Because cache memory is expensive, a computer system can have only a limited amount of it installed. Therefore, in a computer system there is a relatively large and

slower main memory coupled together with a smaller, faster cache memory. The cache contains copies of some blocks of the main memory. Therefore, when the CPU requests a word (if the word is in the fast cache), there will be no need to go to the larger, slower main memory.
The performance of a system can be greatly improved if the cache is placed on the same chip as the processor. In this case, the outputs of the cache can be connected to the ALU and registers through short wires, significantly reducing access time.
Cache operation. When the CPU generates an address for memory reference, the generated address is first sent to the cache. Based on the contents of the cache, a *hit* or a *miss* occurs. A hit occurs when the requested word is already present in the cache. In contrast, a miss happens when the requested word is not in the cache.
Two types of operations can be requested by the CPU: a read request and a write request. When the CPU generates a read request for a word in memory, the generated request is first sent to the cache to check if the word currently resides in the cache. If the word is not found in the cache (i.e., a read miss), the requested word is supplied by the main memory. If the cache is full, a predetermined replacement policy is used to swap out a word from the cache in order to accommodate the new word. If the requested word is found in the cache (i.e., a read hit), the word is supplied by the cache. Thus no fetch from main memory is required. This speeds up the system considerably.
When the CPU generates a write request for a word in memory, the generated request is first sent to the cache to check if the word currently resides in the cache. If the word is not found in the cache (i.e., a write miss), a copy of the word is brought from the memory into the cache. Next, a write operation is performed. Also, a write operation is performed when the word is found in the cache (i.e., a write hit).

To perform a write operation, there are two main approaches that the hardware may employ: *write through*, and *write back*. In the write-through method, the word is modified in both the cache and the main memory. The advantage of the write-through method is that the main memory always has consistent data with the cache. However, it has the disadvantage of slowing down the CPU because all write operations require subsequent accesses to the main memory, which are time consuming.
In the write-back method, every word in the cache has a bit associated with it, called a *dirty bit* (also called an *inconsistent bit*), which tells if it has been changed while in the cache. In this case, the word in the cache may be modified during the write operation, and the dirty bit is set. All changes to a word are performed in the cache. When it is time for a word to be swapped out of the cache, it checks to see if the

word’s dirty bit is set: if it is, it is written back to the main memory in its updated form.
The advantage of the write-back method is that as long as a word stays in the cache it may be modified several times and, for the CPU, it does not matter if the word in the main memory has not been updated.
The disadvantage of the write-back method is that, although only one extra bit has to be associated with each word, it makes the design of the system slightly more complex.

Basic cache organization. The basic motivation behind using cache memories in computer systems is their speed. Most of the time, the presence of the cache is not apparent to the user. Since it is desirable that
very little time be wasted when searching for words in a cache, usually the cache is managed through hardware-based algorithms. The translation of the memory address, specified by the CPU, into the possible location of the corresponding word in the cache is referred to as a *mapping* process. Based on the mapping process used, cache organization can be classified into three types:

1. Associative-mapping cache
2. Direct-mapping cache
3. Set-associative mapping cache

The following sections explain these cache organizations. To illustrate these three different cache Organizations, the memory organization shown in Figure 2.39c is used. In this figure, the CPU communicates with the cache as well as the main memory. The main memory stores 64K words (16-bit address) of 16 bits each. The cache is capable of storing 256 of these words at any given time. Also, in the following discussion it is assumed that the CPU generates a read request and not a write request. (The write request would be handled in a similar way.)

Associative Mapping. In an associative-mapping cache (also referred to as fully associative cache), both the address and the contents are stored as one word in the cache. As a result, a memory word is allowed to be stored at any location in the cache, making it the most flexible cache organization. Figure 2.47 shows the organization of an associative-mapping cache for a system with 16-bit addressing and 16-bit data. Note
that the words are stored at arbitrary locations regardless of their absolute addresses in the main memory.

The organization of an associative-mapping cache can be viewed as a combination of an associative memory and a RAM, as shown in Figure 2.47 (all numbers are in hexadecimal). Since each associative memory cell is many times more expensive than a RAM cell, only the addresses of the words are stored in the associative part, while the data can be stored in the RAM part of the cache because only the address is used for associative search. This will not increase the access time of the cache significantly, but will result in a significant drop in cost. In this organization, when the CPU generates an address for memory reference, it is passed into the argument register and is compared, in parallel, with the address fields of all words currently stored in the cache for a matching address. Once the location has been determined, the corresponding data can be accessed from the RAM.

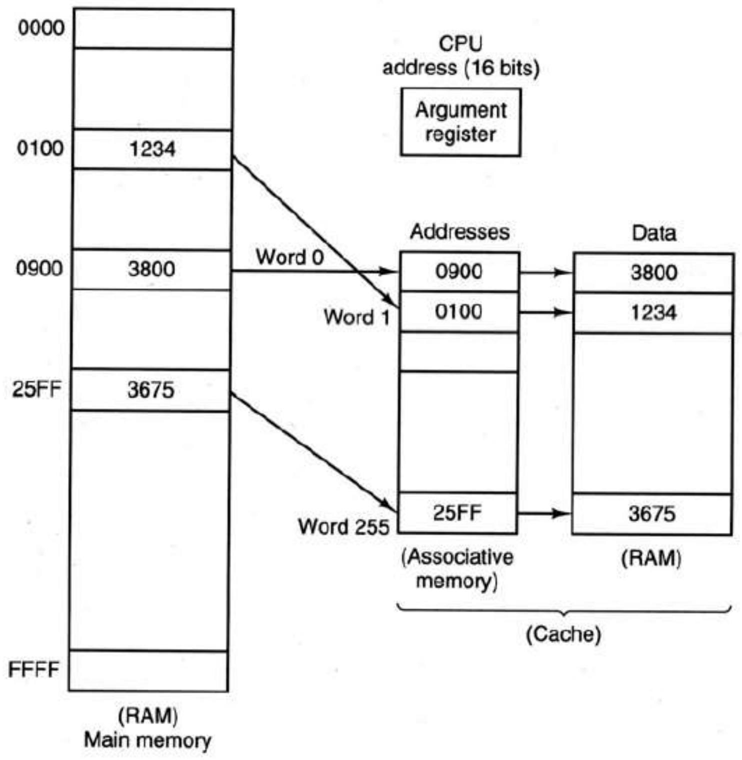


Figure 2.47 Associative-mapping cache (all numbers are in hexadecimal).

The major disadvantage of this method is its need for a large associative memory, which is very expensive and increases the access time of the cache.

Direct Mapping. In a direct-mapping cache, the requested memory address is divided into two parts, an *index* field, which refers to the lower part of the address, and a *tag* field, which refers to the upper part. The index is used as an address to a location in the cache where the data are located. At this index, a tag and a data value are stored in the cache. If the tag of the requested memory address matches the tag of cache, the data value is sent to the CPU. Otherwise, the main memory is accessed, and the corresponding data value is fetched and sent to the CPU. The data value, along with the tag part of its address, also replaces any word currently occupying the corresponding index location in the cache.

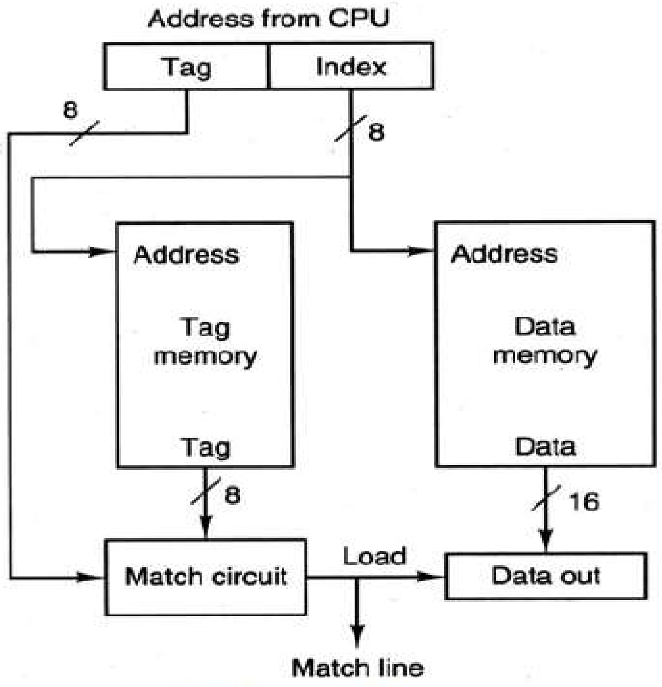


Figure 2.48 Architecture of a direct-mapping cache.

Figure 2.48 represents an architecture for the direct-mapping cache. The design consists of three main components: data memory, tag memory, and match circuit. The data memory holds the cached data. The tag memory holds the tag associated with each cached datum and has an entry for each word of the data memory. The match circuit sets the match line to 1, indicating that the referenced word is in the cache.

An example illustrating the direct-mapping cache operation is shown in Figure 2.49 (all numbers are in hexadecimal). In this example, the memory address consists of 16 bits and the cache has 256 words. The eight least significant bits of the address constitute the index field, and the remaining eight bits constitute the tag field. The 8 index bits determine the address of a word in the tag and data memories. Each word in the tag memory has 8 bits, and each word in the data memory has 16 bits. Initially, the content of address 0900 is stored in the cache. Now, if the CPU wants to read the contents of address 0100, the index (00) matches, but the tag (01) is now different. So the content of main memory is accessed, and the data word 1234 is transferred to the CPU. The tag memory and the data memory words at index address 00 are then replaced with 01 and 1234, respectively.

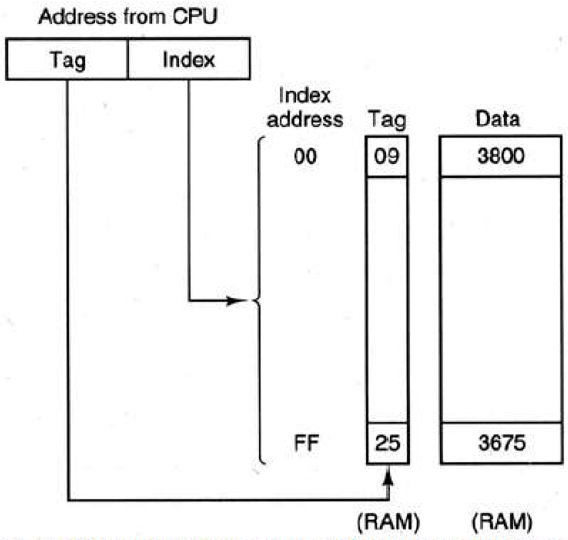


Figure 2.49 Direct-mapping cache (all numbers in hexadecimal).

The advantage of direct mapping over associative mapping is that it requires less overhead in terms of the number of bits per word in the cache. The major disadvantage is that the performance can drop considerably if two or more words having the same index but different tags are accessed frequently. For example, memory addresses 0100 and 0200 both have to be put in the cache at position 00, so a great deal of time is spent swapping them back and forth. This slows the system down, thus defeating the purpose of the cache in the first place. However, considering the property of locality of reference, the probability of having two words with the same index is low. Such words are located 2*k* bits apart in the main memory, where *k* denotes the number of bits in the index field. In our example, such a situation will only occur if the CPU

requests reference to words that are 256 (28) words apart. To further reduce the effects of such situations often an expanded version of the direct-mapping cache, called a *set-associative cache*, is used.
The following section describes the basic structure of a set-associative mapped cache.

Set-Associative Mapping. The set-associative mapping cache organization (also referred to as set associative cache) is an extension of the direct-mapping cache. It solves the problem of direct mapping by providing storage for more than one data value with the same index. For example, a set-associative cache with *m* memory blocks, called *m*-way set associative, can store *m* data values having the same index, along with their tags. Figure 2.50 represents an architecture for a set-associative mapping cache with *m* memory blocks. Each memory block has the same structure as a direct-mapping cache. To determine that a referenced word is in the cache, its tag is compared with the tag of cached data in all memory blocks in parallel. A match in any of the memory blocks will enable (set to 1) the signal match line to indicate that the data are in the cache. If a match occurs, the corresponding data value is passed on to the CPU.
Otherwise, the data value is brought in from the main memory and sent to the CPU. The data value, along with its tag, is then stored in one of the memory blocks.

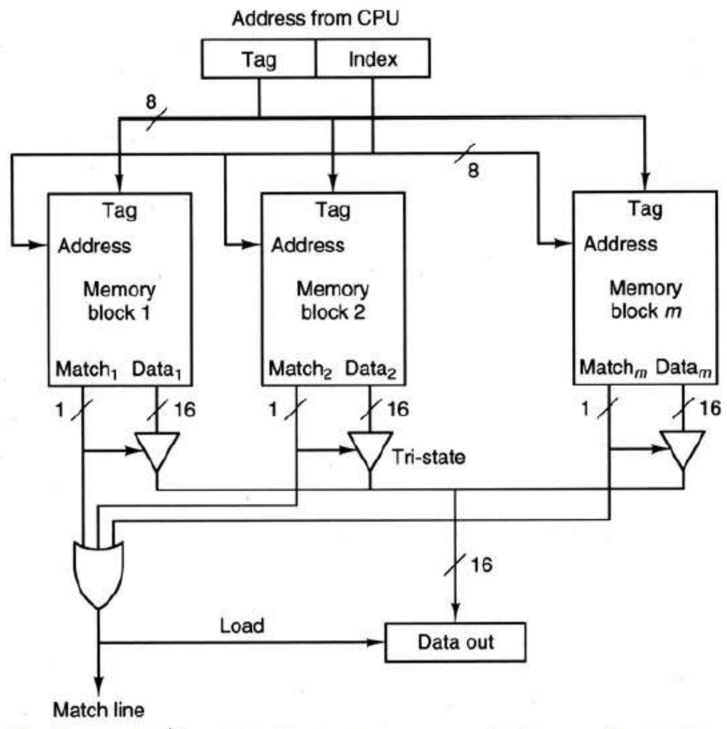


Figure 2.50 Architecture of an *m*-way set-associative mapping cache.

An example illustrating the set-associative mapping cache operation is shown in Figure 2.51 (all numbers are in hexadecimal). This figure represents a two-way set-associative mapping cache. The content of address 0900 is stored in the cache under index 00 and tag 09. If the CPU wants to access address 0100, the index (00) matches, but the tag is now different. Therefore, the content of main memory is accessed, and the data value 1234 is transferred to the CPU. This data with its tag (01) is stored in the second memory block of the cache. When there is no space for a particular index in the cache, one of the two data values stored under that index will be replaced according to some predetermined replacement policy (discussed next).

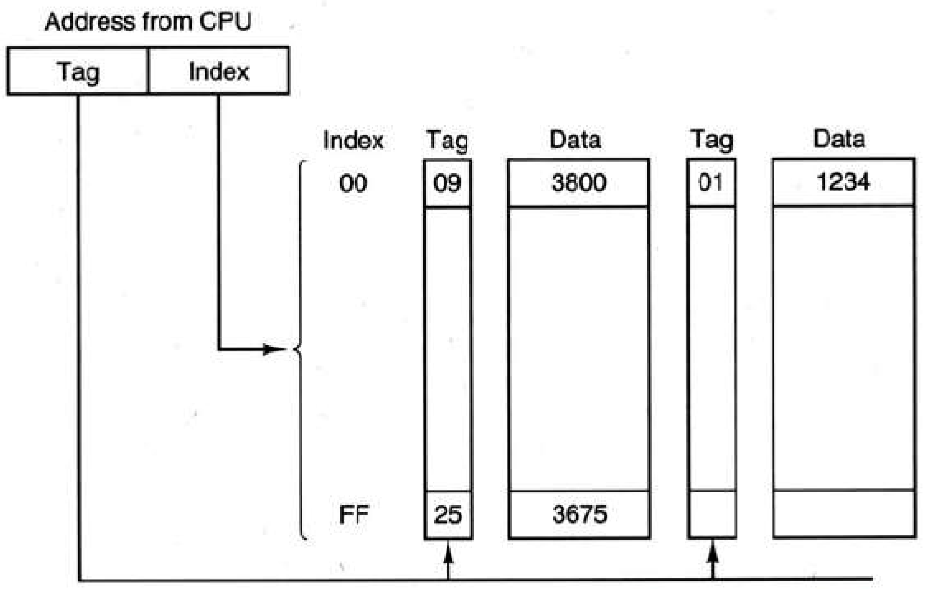


Figure 2.51 Two-way set-associative mapping cache (all numbers are in hexadecimal).