Concept of hierarchical memory

A memory is just like a human brain. It is used to store data and instructions. Computer memory is the storage space in the computer, where data is to be processed and instructions required for processing are stored. The memory is divided into large number of small parts called cells. Each location or cell has a unique address, which varies from zero to memory size minus one. For example, if the computer has 64k words, then this memory unit has 64 * 1024 = 65536 memory locations. The address of these locations varies from 0 to 65535.

Memory is primarily of three types −

- Cache Memory
- Primary Memory/Main Memory
- Secondary Memory

Cache Memory

Cache memory is a very high speed semiconductor memory which can speed up the CPU. It acts as a buffer between the CPU and the main memory. It is used to hold those parts of data and program which are most frequently used by the CPU. The parts of data and programs are transferred from the disk to cache memory by the operating system, from where the CPU can access them.

Advantages

The advantages of cache memory are as follows −

- Cache memory is faster than main memory.
- It consumes less access time as compared to main memory.
- It stores the program that can be executed within a short period of time.
- It stores data for temporary use.

Disadvantages

The disadvantages of cache memory are as follows −

- Cache memory has limited capacity.
- It is very expensive.

Primary Memory (Main Memory)

Primary memory holds only those data and instructions on which the computer is currently working. It has a limited capacity and data is lost when power is switched off. It is generally made up of semiconductor device. These memories are not as fast as registers. The data and instruction required to be processed resides in the main memory. It is divided into two subcategories RAM and ROM.

Characteristics of Main Memory

- These are semiconductor memories.
- It is known as the main memory.
- Usually volatile memory.
- Data is lost in case power is switched off.
- It is the working memory of the computer.
- Faster than secondary memories.
- A computer cannot run without the primary memory.

Secondary Memory

This type of memory is also known as external memory or non-volatile. It is slower than the main memory. These are used for storing data/information permanently. CPU directly does not access these memories, instead they are accessed via input-output routines. The contents of secondary memories are first transferred to the main memory, and then the CPU can access it. For example, disk, CD-ROM, DVD, etc.

Characteristics of Secondary Memory

- These are magnetic and optical memories.
- It is known as the backup memory.
- It is a non-volatile memory.
- Data is permanently stored even if power is switched off.
- It is used for storage of data in a computer.
- Computer may run without the secondary memory.
- Slower than primary memories.

RAM (Random Access Memory) is the internal memory of the CPU for storing data, program, and program result. It is a read/write memory which stores data until the machine is working. As soon as the machine is switched off, data is erased.

Access time in RAM is independent of the address, that is, each storage location inside the memory is as easy to reach as other locations and takes the same amount of time. Data in the RAM can be accessed randomly but it is very expensive.

RAM is volatile, i.e. data stored in it is lost when we switch off the computer or if there is a power failure. Hence, a backup Uninterruptible Power System (UPS) is often used with computers. RAM is small, both in terms of its physical size and in the amount of data it can hold.

RAM is of two types −

- Static RAM (SRAM)
- Dynamic RAM (DRAM)

Static RAM (SRAM)

The word **static** indicates that the memory retains its contents as long as power is being supplied. However, data is lost when the power gets down due to volatile nature. SRAM chips use a matrix of 6-transistors and no capacitors. Transistors do not require power to prevent leakage, so SRAM need not be refreshed on a regular basis.

There is extra space in the matrix, hence SRAM uses more chips than DRAM for the same amount of storage space, making the manufacturing costs higher. SRAM is thus used as cache memory and has very fast access.

Characteristic of Static RAM

- Long life
- No need to refresh
- Faster
- Used as cache memory
- Large size
- Expensive
- High power consumption

Dynamic RAM (DRAM)

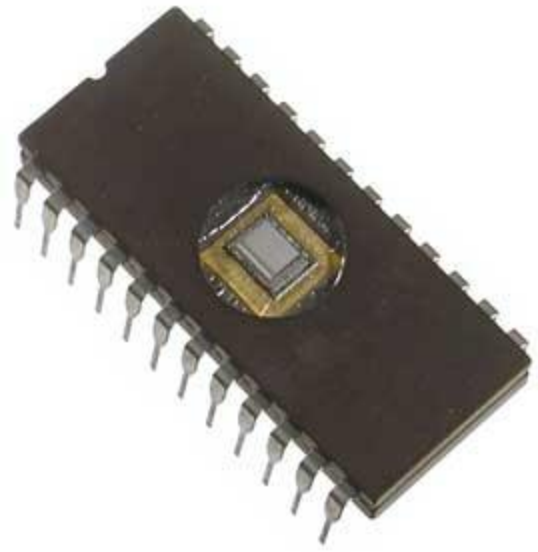
DRAM, unlike SRAM, must be continually **refreshed** in order to maintain the data. This is done by placing the memory on a refresh circuit that rewrites the data several hundred times per second. DRAM is used for most system memory as it is cheap and small. All DRAMs are made up of memory cells, which are composed of one capacitor and one transistor.

Characteristics of Dynamic RAM

- Short data lifetime
- Needs to be refreshed continuously

- Slower as compared to SRAM
- Used as RAM
- Smaller in size
- Less expensive
- Less power consumption

ROM stands for **Read Only Memory**. The memory from which we can only read but cannot write on it. This type of memory is non-volatile. The information is stored permanently in such memories during manufacture. A ROM stores such instructions that are required to start a computer. This operation is referred to as **bootstrap**. ROM chips are not only used in the computer but also in other electronic items like washing machine and microwave oven.



Let us now discuss the various types of ROMs and their characteristics.

MROM (Masked ROM)

The very first ROMs were hard-wired devices that contained a pre-programmed set of data or instructions. These kind of ROMs are known as masked ROMs, which are inexpensive.

PROM (Programmable Read Only Memory)

PROM is read-only memory that can be modified only once by a user. The user buys a blank PROM and enters the desired contents using a PROM program. Inside the PROM chip, there are small fuses which are burnt open during programming. It can be programmed only once and is not erasable.

EPROM (Erasable and Programmable Read Only Memory)

EPROM can be erased by exposing it to ultra-violet light for a duration of up to 40 minutes. Usually, an EPROM eraser achieves this function. During programming, an electrical charge is trapped in an insulated gate region. The charge is retained for more than 10 years because the charge has no leakage path. For erasing this charge, ultra-violet light is passed through a quartz crystal window (lid). This exposure to ultra-violet light dissipates the charge. During normal use, the quartz lid is sealed with a sticker.

EEPROM (Electrically Erasable and Programmable Read Only Memory)

EEPROM is programmed and erased electrically. It can be erased and reprogrammed about ten thousand times. Both erasing and programming take about 4 to 10 ms (millisecond). In EEPROM, any location can be selectively erased and programmed. EEPROMs can be erased one byte at a time, rather than erasing the entire chip. Hence, the process of reprogramming is flexible but slow.

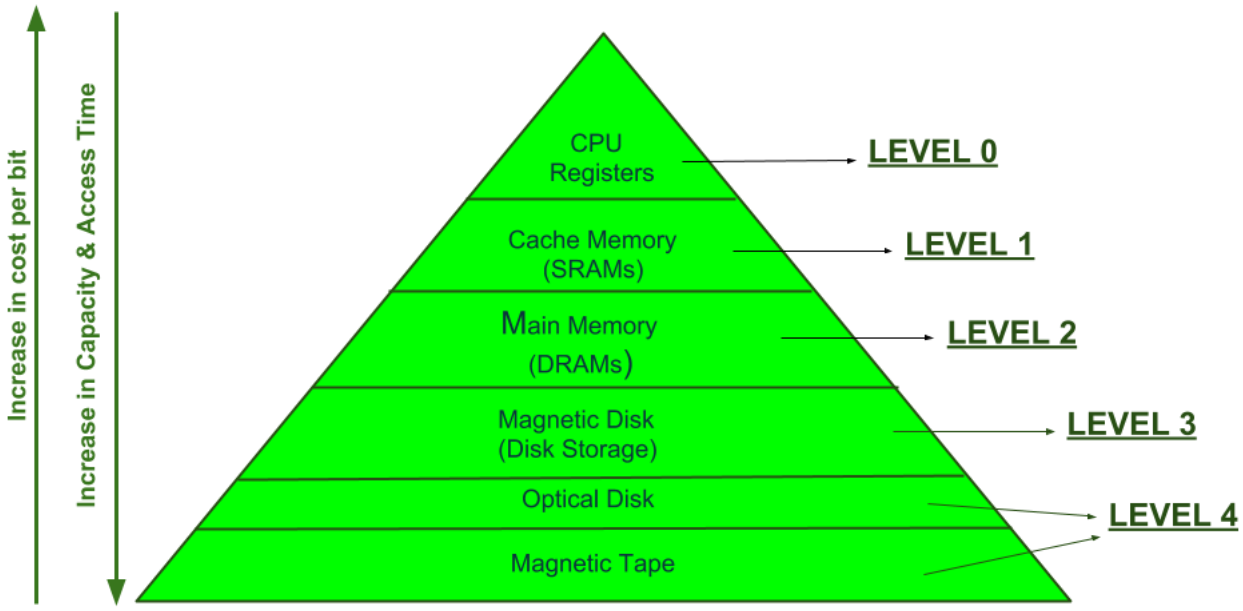
Advantages of ROM

The advantages of ROM are as follows −

- Non-volatile in nature
- Cannot be accidentally changed
- Cheaper than RAMs
- Easy to test
- More reliable than RAMs
- Static and do not require refreshing
- Contents are always known and can be verified

Memory Hierarchy Design and its Characteristics

In the Computer System Design, Memory Hierarchy is an enhancement to organize the memory such that it can minimize the access time. The Memory Hierarchy was developed based on a program behavior known as locality of references.The figure below clearly demonstrates the different levels of memory hierarchy :



MEMORY HIERARCHY DESIGN

This Memory Hierarchy Design is divided into 2 main types:

1. **External Memory or Secondary Memory –**
   Comprising of Magnetic Disk, Optical Disk, Magnetic Tape i.e. peripheral storage devices which are accessible by the processor via I/O Module.
2. **Internal Memory or Primary Memory –**
   Comprising of Main Memory, Cache Memory & CPU registers. This is directly accessible by the processor.

We can infer the following characteristics of Memory Hierarchy Design from above figure:

1. **Capacity:**
   It is the global volume of information the memory can store. As we move from top to bottom in the Hierarchy, the capacity increases.
2. **Access Time:**
   It is the time interval between the read/write request and the availability of the data. As we move from top to bottom in the Hierarchy, the access time increases.
3. **Performance:**
   Earlier when the computer system was designed without Memory Hierarchy design, the speed gap increases between the CPU registers and Main Memory due to large difference in access time. This results in lower performance of the system and thus, enhancement was required. This enhancement was made in the form of Memory Hierarchy Design because of which the performance of the system increases. One of the most significant ways to increase system performance is minimizing how far down the memory hierarchy one has to go to manipulate data.
4. **Cost per bit:**
   As we move from bottom to top in the Hierarchy, the cost per bit increases i.e. Internal Memory is costlier than External Memory.

Byte Addressable :

The smallest unit of information is known as bit (binary digit), and in one memory cell we can store one bit of information. 8 bit together is termed as a byte.

The maximum size of main memory that can be used in any computer is determined by the addressing scheme.
A computer that generates 16-bit address is capable of addressing upto 2^{16} which is equal to 64K memory location.

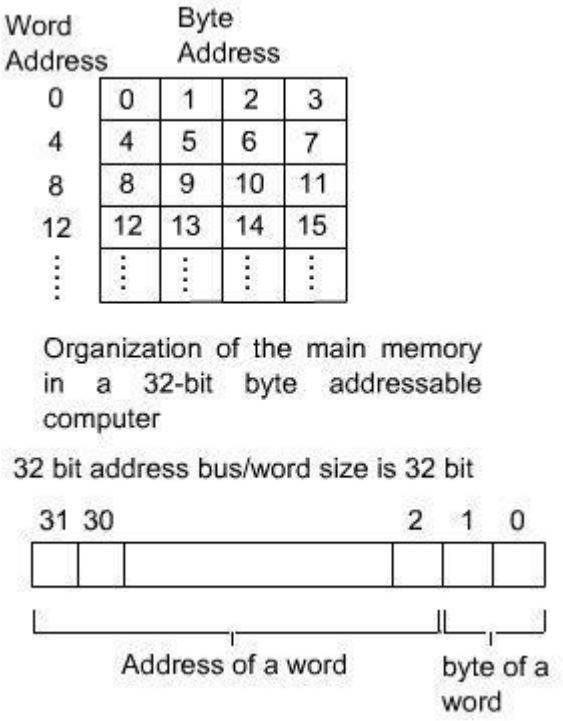
Similarly, for 32 bit addresses, the total capacity will be 2^{32} which is equal to 4G memory location.
In some computer, the smallest addressable unit of information is a memory word and the machine is called word-addressable.

In some computer, individual address is assigned for each byte of information, and it is called **byte-addressable computer**. In this computer, one memory word contains one or more memory bytes which can be addressed individually.

A byte addressable 32-bit computer, each memory word contains 4 bytes. A possible way of address assignment is shown in figure3.1. The address of a word is always integer multiple of 4.

The main memory is usually designed to store and retrieve data in word length quantities. The word length of a computer is generally defined by the number of bits actually stored or retrieved in one main memory access.

Consider a machine with 32 bit address bus. If the word size is 32 bit, then the high order 30 bit will specify the address of a word. If we want to access any byte of the word, then it can be specified by the lower two bit of the address bus.



Organization of the main memory in a 32-bit byte addressable computer

Figure 3.1: Address assignment to a 4-byte word

The data transfer between main memory and the CPU takes place through two CPU registers.

- **MAR :** Memory Address Register
- **MDR :** Memory Data Register.

Function of MDR & MAR

If the MAR is *k*-bit long, then the total addressable memory location will be 2^k.

If the MDR is *n*-bit long, then the *n* bit of data is transferred in one memory cycle.

The transfer of data takes place through memory bus, which consist of address bus and data bus. In the above example, size of data bus is *n*-bit and size of address bus is *k* bit.

It also includes control lines like Read, Write and Memory Function Complete (MFC) for coordinating data transfer. In the case of byte addressable computer, another control line to be added to indicate the byte transfer instead of the whole word.

For memory operation, the CPU initiates a memory operation by loading the appropriate data i.e., address to MAR.

If it is a memory read operation, then it sets the read memory control line to 1. Then the contents of the memory location is brought to MDR and the memory control circuitry indicates this to the CPU by setting MFC to 1.

If the operation is a memory write operation, then the CPU places the data into MDR and sets the write memory control line to 1. Once the contents of MDR are stored in specified memory location, then the memory control circuitry indicates the end of operation by setting MFC to 1.

A useful measure of the speed of memory unit is the time that elapses between the initiation of an operation and the completion of the operation (for example, the time between Read and MFC). This is referred to as **Memory Access Time**. Another measure is memory cycle time. This is the minimum time delay between the initiation two independent memory operations (for example, two successive memory read operation). Memory cycle time is slightly larger than memory access time.

The storage part is modelled here with SR-latch, but in reality it is an electronics circuit made up of transistors. The memory consttucted with the help of transistors is known as semiconductor memory. The semiconductor memories are termed as Random Access Memory(RAM), because it is possible to access any memory location in random.
Depending on the technology used to construct a RAM, there are two types of RAM -

SRAM: StaticRandomAccessMemory.
DRAM: Dynamic Random Access Memory.

Dynamic Ram (DRAM):A DRAM is made with cells that store data as charge on capacitors. The presence or absence of charge in a capacitor is interpreted as binary 1 or 0.
Because capacitors have a natural tendency to discharge due to leakage current, dynamic RAM require periodic charge refreshing to maintain data storage. The term dynamic refers to this tendency of the stored charge to leak away, even with power continuously applied.
Static RAM (SRAM):

In an SRAM, binary values are stored using traditional flip-flop constructed with the help of transistors. A static RAM will hold its data as long as power is supplied to it.
SRAM Versus DRAM :

- Both static and dynamic RAMs are volatile, that is, it will retain the information as long as power supply is applied.

- A dynamic memory cell is simpler and smaller than a static memory cell. Thus a DRAM is more dense, i.e., packing density is high(more cell per unit area). DRAM is less expensive than corresponding SRAM.

- DRAM requires the supporting refresh circuitry. For larger memories, the fixed cost of the refresh circuitry is more than compensated for by the less cost of DRAM cells

- SRAM cells are generally faster than the DRAM cells. Therefore, to construct faster memory modules(like cache memory) SRAM is used.

Internal Organization of Memory Chips

A memory cell is capable of storing 1-bit of information. A number of memory cells are organized in the form of a matrix to form the memory chip. One such organization is shown in the Figure 3.5.

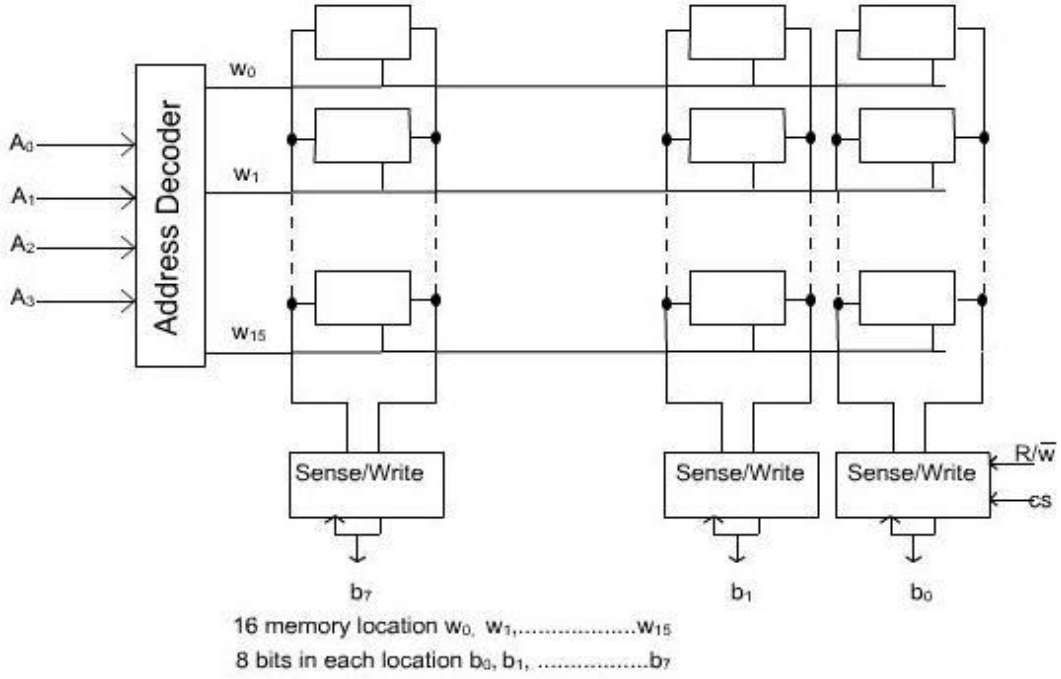


Figure 3.5: 16 X 8 Memory Organization

Each row of cells constitutes a memory word, and all cell of a row are connected to a common line which is referred as word line. An address decoder is used to drive the word line. At a particular instant, one word line is enabled depending on the address present in the address bus. The cells in each column are connected by two lines. These are known as bit lines. These bit lines are connected to data input line and data output line through a Sense/Write circuit. During a Read operation, the Sense/Write circuit sense, or read the information stored in the cells selected by a word line and transmit this information to the output data line. During a write operation, the sense/write circuit receive information and store it in the cells of the selected word.

A memory chip consisting of 16 words of 8 bits each, usually referred to as 16 x 8 organization. The data input and data output line of each Sense/Write circuit are connected to a single bidirectional data line in order to reduce the pin required. For 16 words, we need an address bus of size 4. In addition to address and data lines, two control lines, $R/\overline{W}$ and CS, are provided. The $R/\overline{W}$ line is to used to specify the required operation about read or write. The CS (Chip Select) line is required to select a given chip in a multi chip memory system.

Internal Organization of Memory Chips

A memory cell is capable of storing 1-bit of information. A number of memory cells are organized in the form of a matrix to form the memory chip. One such organization is shown in the Figure.

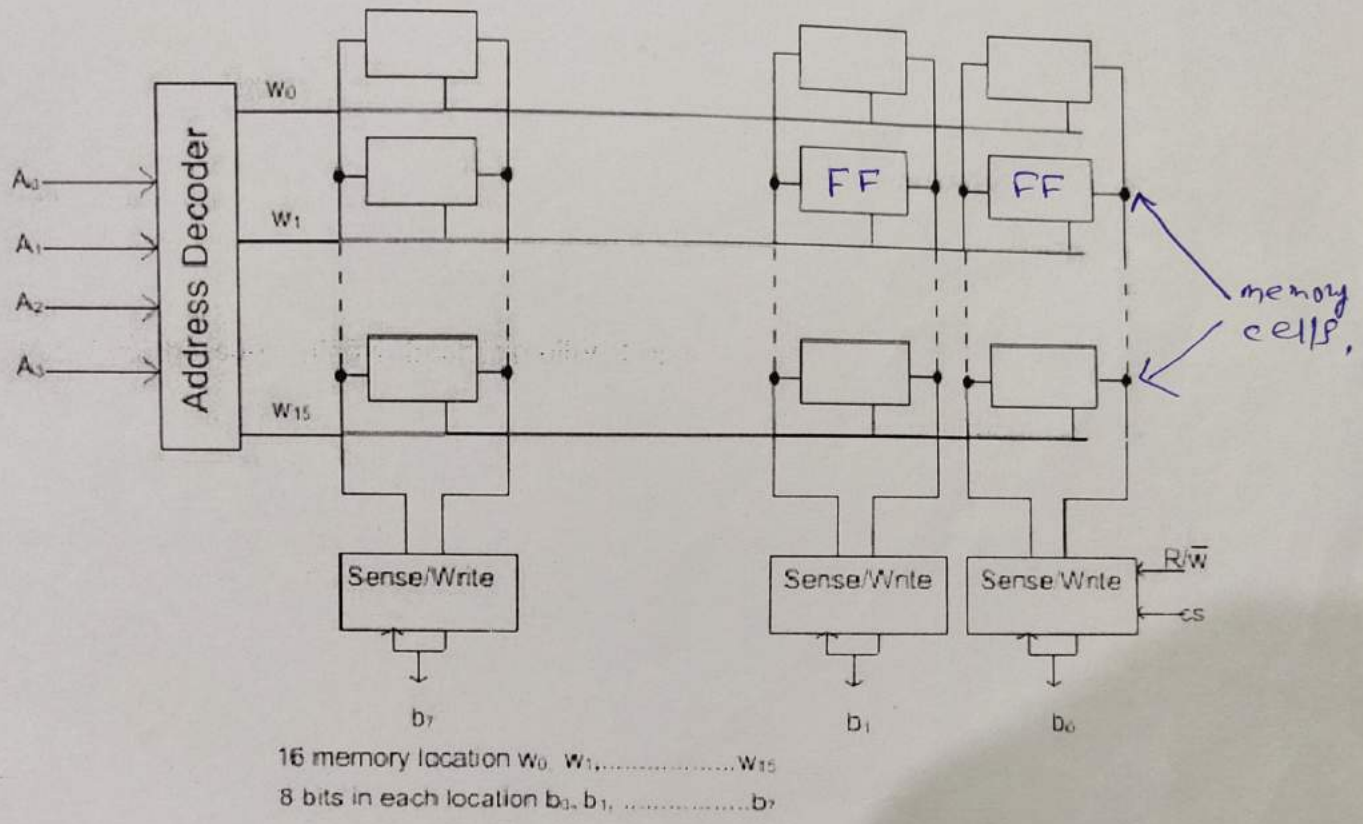


Figure : 16 X 8 Memory Organization

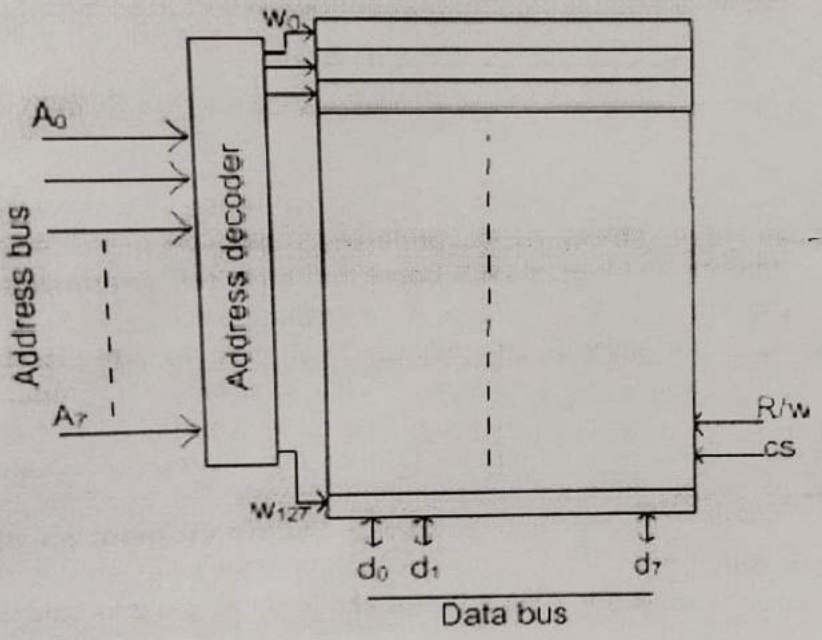
Each row of cells constitutes a memory word, and all cell of a row are connected to a common line which is referred as word line. An address decoder is used to drive the word line. At a particular instant, one word line is enabled depending on the address present in the address bus. The cells in each column are connected by two lines. These are known as bit lines. These bit lines are connected to data input line and data output line through a Sense/Write circuit. During a Read operation, the Sense/Write circuit sense, or read the information stored in the cells selected by a word line and transmit this information to the output data line. During a write

operation, the sense/write circuit receive information and store it in the cells of the selected word.A memory chip consisting of 16 words of 8 bits each, usually referred to as 16 x 8 organization. The data input and data output line of each Sense/Write circuit are connected to a single bidirectional data line in order to reduce the pin required. For 16 words, we need an address bus of size 4. In addition to address and data lines, two control lines, and CS, are provided. The line is to used to specify the required operation about read or write. The CS (Chip Select) line is required to select a given chip in a multi chip memory system.

Consider a slightly larger memory unit that has 1K (1024) memory cells...

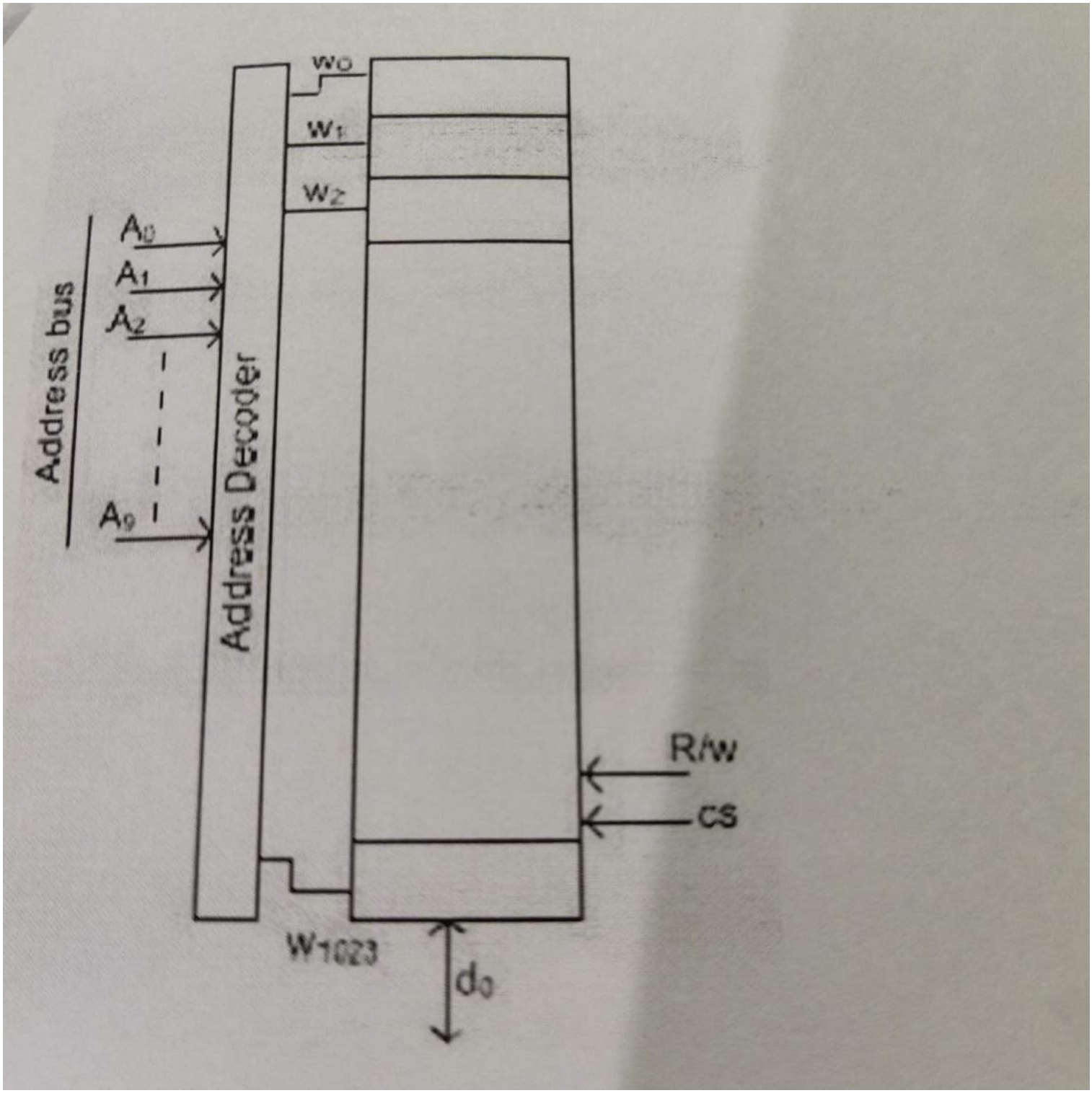
128 x 8 memory chips:

If it is organized as a 128 x 8 memory chips, then it has got 128 memory words of size 8 bits. So the size of data bus is 8 bits and the size of address bus is 7 bits (2^7=128).



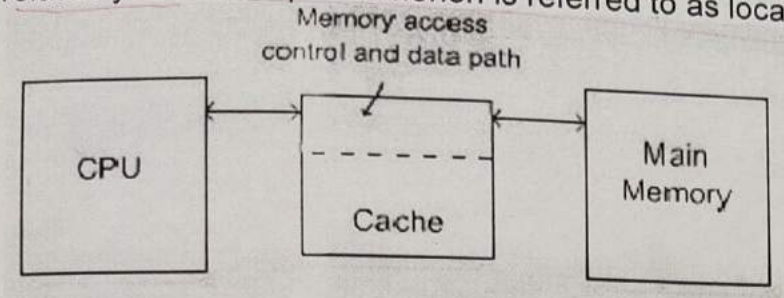
1024 x 1 memory chips:

If it is organized as a 1024 x 1 memory chips, then it has got 1024 memory words of size 1 bit only.Therefore, the size of data bus is 1 bit and the size of address bus is 10 bits (2^{10}=1024).A particular memory location is identified by the contents of memory address bus. A decoder is used to decode the memory address. There are two ways of decoding of a memory address depending upon the organization of the memory module. In one case, each memory word is organized in a row. In this case whole memory address bus is used together to decode the address of the specified location.


W0
W1
W2
A0
A1
A2
A9
Address bus
Address Decoder
R/W
CS
W1023
d0

Cache Memory

Analysis of large number of programs has shown that a number of instructions are executed repeatedly. This may be in the form of a simple loops, nested loops, or a few procedures that repeatedly call each other. It is observed that many instructions in each of a few localized areas of the program are repeatedly executed, while the remainder of the program is accessed relatively less. This phenomenon is referred to as locality of reference.



Cache memory between the CPU and the main memory

Now, if it can be arranged to have the active segments of a program in a fast memory, then the tolal execution time can be significantly reduced. It is the fact that CPU is a faster device and memory is a relatively slower device.Memory access is the main bottleneck for the performance efficiency. If a faster memory device can be inserted between main memory and CPU, the efficiency can be increased. The faster memory that is inserted between CPU and Main Memory is termed as Cache memory. To make this arrangement effective, the cache must be considerably faster than the main memory, and typically it is 5 to 10 time faster than the main memory. This approach is more economical than the use of fast memory device to implement the entire main memory. This is also a feasible due to the locality of reference that is present in most of the program, which reduces the frequent data transfer between main memory and cache memory.

Operation of Cache Memory

The memory control circuitry is designed to take advantage of the property of locality of reference. Some assumptions are made while designing the memory control circuitry:

1. The CPU does not need to know explicitly about the existence of the cache.
2. The CPU simply makes Read and Write request. The nature of these two operations are same whether cache is present or not.
3. The address generated by the CPU always refer to location of main memory.
4. The memory access control circuitry determines whether or not the requested word currently exists in the cache.

When a Read request is received from the CPU, the contents of a block of memory words containing the location specified are transferred into the cache. When any of the locations in this block is referenced by the program, its contents are read directly from the cache.The cache memory can store a number of such blocks at any given time.The correspondence between the Main Memory Blocks and those in the cache is specified by means of a mapping function.

When the cache is full and a memory word is referenced that is not in the cache, a decision must be made as to which block should be removed from the cache to create space to bring the new block to the cache that contains the referenced word. **Replacement algorithms** are used to make the proper selection of block that must be replaced by the new one. When a write request is received from the CPU, there are two ways that the system can proceed. In the first case, the cache location and the main memory location are updated simultaneously. This is called the **store through method** or **write through method**.

The alternative is to update the cache location only. During replacement time, the cache block will be written back to the main memory. If there is no new write operation in the cache block, it

is not required to write back the cache block in the main memory. This information can be kept with the help of an associated bit. This bit it set while there is a write operation in the cache block. During replacement, it checks this bit, if it is set, then write back the cache block in main memory otherwise not. This bit is known as **dirty bit**. If the bit gets dirty (set to one), writting to main memory is required. This write through method is simpler, but it results in unnecessary write operations in the main memory when a given cache word is updated a number of times during its cache residency period. Consider the case where the addressed word is not in the cache and the operation is a read. First the block of the words is brought to the cache and then the requested word is forwarded to the CPU. But it can be forwarded to the CPU as soon as it is available to the cache, instead of whole block to be loaded into the cache. This is called load through, and there is some scope to save time while using load through policy.

During a write operation, if the address word is not in the cache, the information is written directly into the main memory. A write operation normally refers to the location of data areas and the property of locality of reference is not as pronounced in accessing data when write operation is involved. Therefore, it is not advantageous to bring the data block to the cache when there a write operation, and the addressed word is not present in cache.

Mapping Functions

The mapping functions are used to map a particular block of main memory to a particular block of cache. This mapping function is used to transfer the block from main memory to cache memory. Three different mapping functions are available:

Direct mapping:

A particular block of main memory can be brought to a particular block of cache memory. So, it is not flexible.

Associative mapping:

In this mapping function, any block of Main memory can potentially reside in any cache block position.This is much more flexible mapping method.

Block-set-associative mapping:

In this method, blocks of cache are grouped into sets, and the mapping allows a block of main memory to reside in any block of a specific set. From the flexibility point of view, it is in between to the other two methods.

All these three mapping methods are explained with the help of an example.

Consider a cache of 4096 (4K) words with a block size of 32 words. Therefore, the cache is organized as 128 blocks.For 4K words, required address lines are 12 bits. To select one of the block out of 128 blocks, we need 7 bits of address lines and to select one word out of 32 words, we need 5 bits of address lines. So the total 12 bits of address is divided for two groups, lower 5 bits are used to select a word within a block, and higher 7 bits of address are used to select any block of cache memory.

Let us consider a main memory system consisting 64K words. The size of address bus is 16 bits. Since the block size of cache is 32 words, so the main memory is also organized as block size of 32 words. Therefore, the total number of blocks in main memory is 2048 (2K x 32 words = 64K words). To identify any one block of 2K blocks, we need 11 address lines. Out of 16 address lines of main memory, lower 5 bits are used to select a word within a block and higher 11 bits are used to select a block out of 2048 blocks.

Number of blocks in cache memory is 128 and number of blocks in main memory is 2048, so at any instant of time only 128 blocks out of 2048 blocks can reside in cache menory. Therefore, we need mapping function to put a particular block of main memory into appropriate block of cache memory.

Locality of Reference in cache memory

Locality of Reference (Principle of Locality)

Locality of reference is the tendency of a process to access data & instructions from a certain localized areas of memory.

The Locality of reference is also of two types

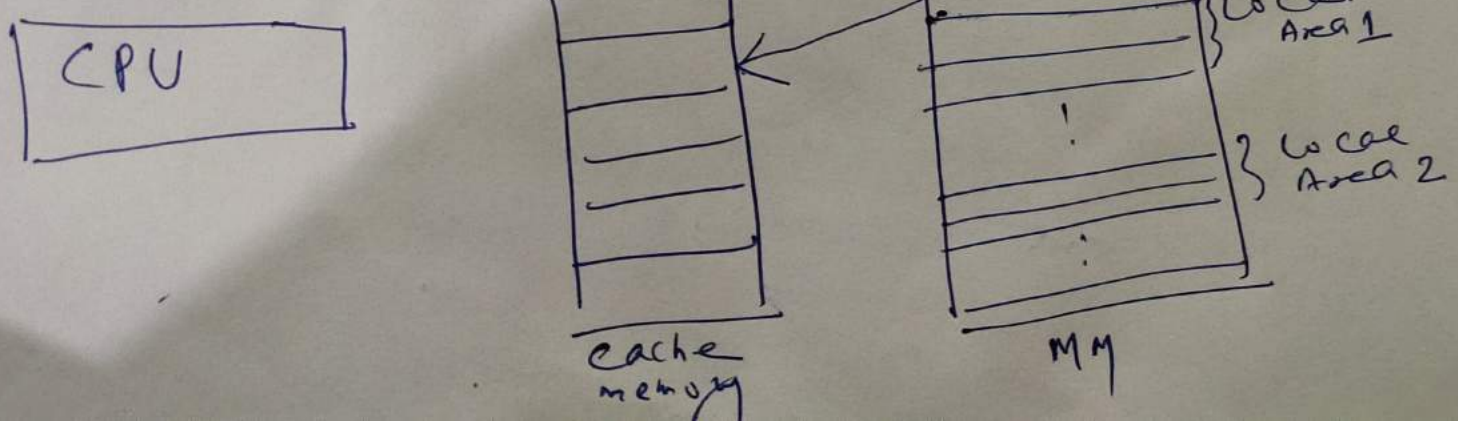
(1) Spatial Locality

If an item is accessed in memory, then items with nearby addresses will tend to be referenced next. (Referencing instruction in sequence)

(2) Temporal Locality:

If an item is accessed in memory, then same item will tend to be referenced again in near future

(Referencing instruction in a loop)



Cache memory mapping:

It means how data is copied (mapped) from main memory to cache memory.

Mapping Technique:

In this mapping, main memory blocks are copied to a fixed block of cache memory, but one at a time.

Let cache memory block no. = c_b

main memory block no. = m_b

No. of block in cache = C

No. of block in main = m

Cache memory block no. = (main memory block no.) mod (No. of cache blocks)

$$c_b = m_b \bmod C$$

Block size = Group of words

$$\text{No. of main memory blocks} = \frac{\text{Total main memory words}}{\text{block size}}$$

$$\text{No. of cache memory block} = \frac{\text{Total cache memory words}}{\text{block size}}$$

$$\text{No. of main memory blocks in one block of cache} = \frac{\text{no. of main memory blocks}}{\text{No. of cache memory blocks}} = \frac{2048}{128} = 16 \text{ blocks}$$

Direct Mapping Technique:

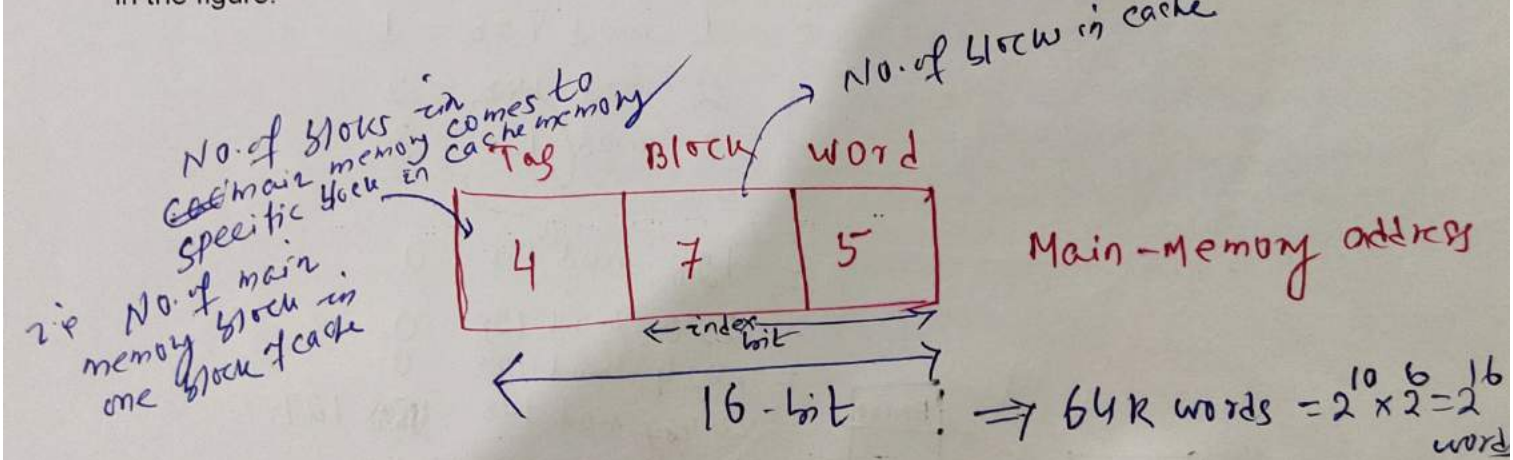
The simplest way of associating main memory blocks with cache block is the direct mapping technique. In this technique, block k of main memory maps into block k modulo m of the cache, where m is the total number of blocks in cache. In this example, the value of m is 128. In direct mapping technique, one particular block of main memory can be transfered to a particular block of cache which is derived by the modulo function.

Since more than one main memory block is mapped onto a given cache block position, contention may arise for that position. This situation may occurs even when the cache is not full. Contention is resolved by allowing the new block to overwrite the currently resident block. So the replacement algorithm is trivial.
The detail operation of direct mapping technique is as follows:

The main memory address is divided into three fields. The field size depends on the memory capacity and the block size of cache. In this example, the lower 5 bits of address is used to identify a word within a block. Next 7 bits are used to select a block out of 128 blocks (which is the capacity of the cache). The remaining 4 bits are used as a TAG to identify the proper block of main memory that is mappéd to cache.

When a new block is first brought into the cache, the high order 4 bits of the main memory address are stored in four TAG bits associated with its location in the cache. When the CPU generates a memory request, the 7-bit blockaddress determines the corresponding cache block. The TAG field of that block is compared to the TAG field of the address. If they match, the desired word specified by the low-order 5 bits of the address is in that block of the cache.

If there is no match, the required word must be accessed from the main memory, that is, the contents of that block of the cache is replaced by the new block that is specified by the new address generated by the CPU and correspondingly the TAG bit will also be changed by the high order 4 bits of the address. The whole arrangement for direct mapping technique is shown in the figure:



Given Block size = 32 words

Total cache memory words = 4096 (4K) words
(Cache size)

Total main memory words = 64K words

No. of main memory blocks = $\frac{\text{Total main memory words}}{\text{block size}}$

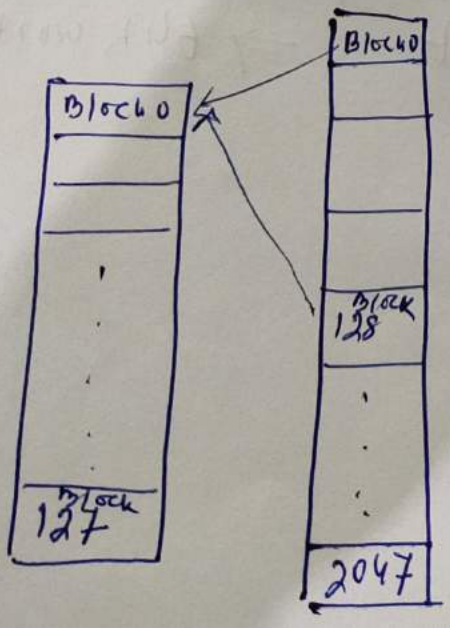
$= \frac{64K}{32} = \frac{2^{16}}{2^5} = 2^{11} = 2048 = 2K$ blocks

No. of cache memory blocks = $\frac{\text{Total cache memory words}}{\text{block size}}$

$= \frac{4K}{2^5} = \frac{2^{12}}{2^5} = 2^7 = 128$ no. of blocks

cache memory block no. = $M_b \bmod C$

$= 0 \bmod 128 = 0$
$= 1 \bmod 128 = 1$
$= 2 \bmod 128 = 2$
$= 3 \bmod 128 = 3$
.
.
$= 128 \bmod 128 = 0$
$= 256 \bmod 128 = 0$
$= 1024 \bmod 128 = 0$
$= 20487 \bmod 128 = 127$



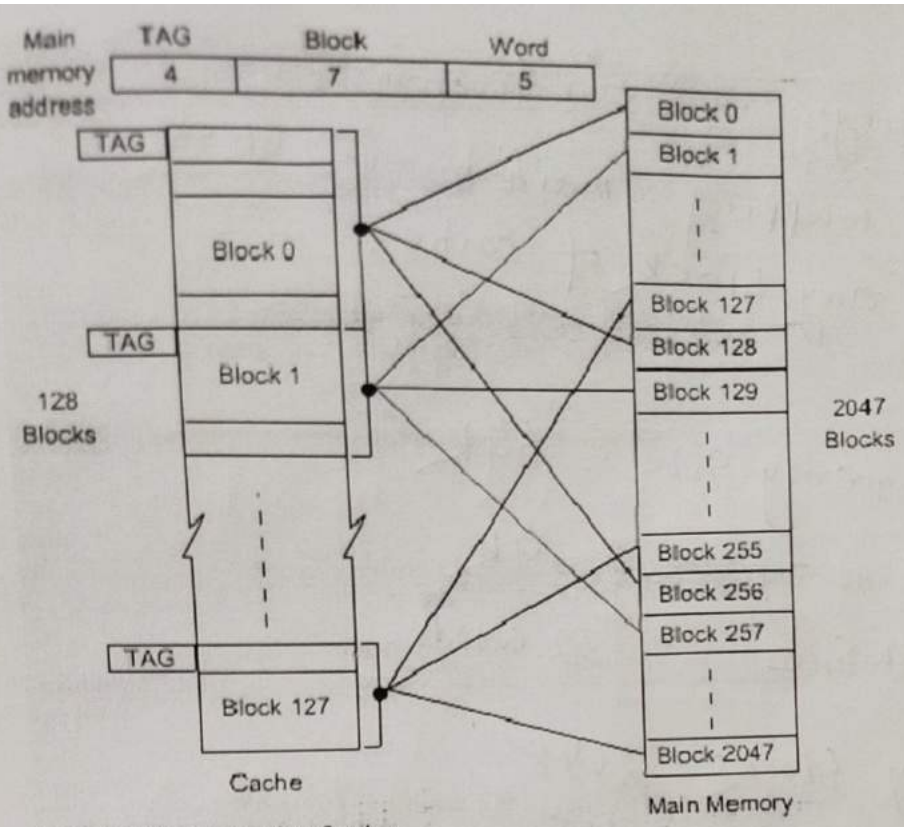
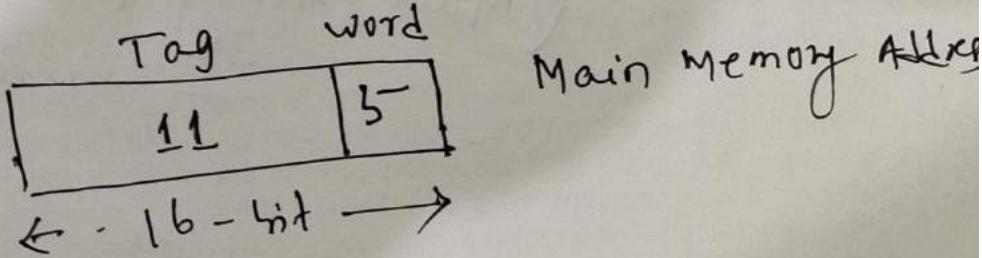


Figure : Direct-mapping Cache

Associated Mapping Technique:

In the associative mapping technique, a main memory block can potentially reside in any cache block position. In this case, the main memory address is divided into two groups, low-order bits identifies the location of a word within a block and high-order bits identifies the block. In the example here, 11 bits are required to identify a main memory block when it is resident in the cache , high-order 11 bits are used as TAG bits and low-order 5 bits are used to identify a word within a block. The TAG bits of an address received from the CPU must be compared to the TAG bits of each block of the cache to see if the desired block is present. In the associative mapping, any block of main memory can go to any block of cache, so it has got the complete flexibility and we have to use proper replacement policy to replace a block from cache if the currently accessed block of main memory is not present in cache. It might not be practical to use this complete flexibility of associative mapping technique due to searching overhead, because the TAG field of main memory address has to be compared with the TAG field of all the cache block. In this example, there are 128 blocks in cache and the size of TAG is 11 bits. The



Associative Mapping: (No Restriction)

In associative mapping main memory blocks are copied into any block of cache.

Given

main memory size = 64K

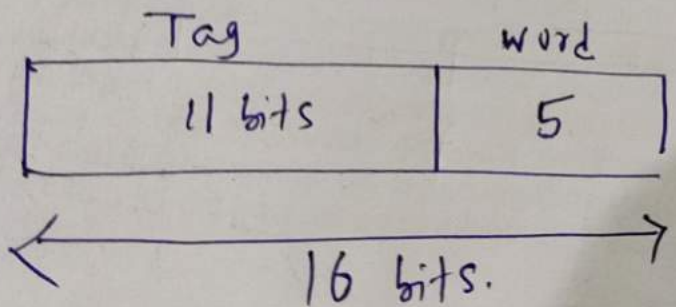
cache memory size = 4K

block size = 32 words.

No. of main memory blocks = 2048

No. of cache memory blocks = 128

Here no. of main memory blocks come into one block of cache = 2048 = 2K = $2 \times 2^{10} = 2^{11}$ (Tag bit = 11 bit)



{Main-memory Address}

Mapping Technique is shown in the figure

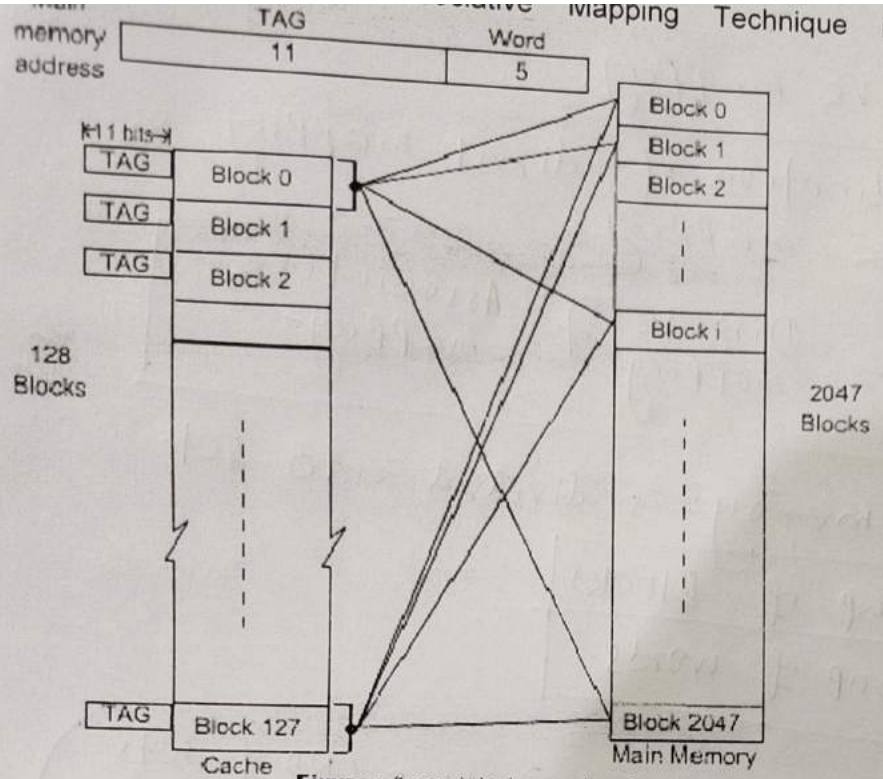


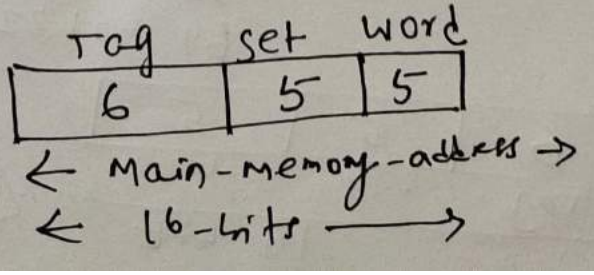
Figure : Associated-mapping Cache

Block-Set-Associative Mapping Technique:

This mapping technique is intermediate to the above two techniques. Blocks of the cache are grouped into sets, and the mapping allows a block of main memory to reside in any block of a specific set. Therefore, the flexibity of associative mapping is reduced from full freedom to a set of specific blocks. This also reduces the searching overhead, because the search is restricted to number of sets, instead of number of blocks. Also the contention problem of the direct mapping is eased by having a few choices for block replacement.

Consider the same cache memory and main memory organization of the previous example. Organize the cache with 4 blocks in each set. The TAG field of associative mapping technique is divided into two groups, one is termed as SET bit and the second one is termed as TAG bit. Since each set contains 4 blocks, total number of set is 32. The main memory address is grouped into three parts: low-order 5 bits are used to identifies a word within a block. Since there are total 32 sets present, next 5 bits are used to identify the set. High-order 6 bits are used as TAG bits.

The 5-bit set field of the address determines which set of the cache might contain the desired block. This is similar to direct mapping technique, in case of direct mapping, it looks for block, but in case of block-set-associative mapping, it looks for set. The TAG field of the address must then be compared with the TAGs of the four blocks of that set. If a match occurs, then the block is present in the cache; otherwise the block containing the addressed word must be brought to the cache. This block will potentially come to the corresponding set only. Since, there are four blocks in the set, we have to choose appropriately which block to be replaced if all the blocks are occupied. Since the search is restricted to four block only, so the searching complexity is reduced. The whole arrangement of block-set-associative mapping technique is shown in the figure.



set - Associative mapping:

It is a combination of direct mapping & ~~set~~ associative mapping

set-Associative mapping = Direct mapping + Associative mapping

In this, Cache memory is divided into set

set = group of blocks
Blocks = group of words

$$\text{Cache memory set-no} = (\text{main memory block no.}) \text{ MOD } (\text{No. of sets in cache memory})$$

$$\text{No. of sets in cache memory} = \frac{\text{No. of cache memory blocks}}{\text{set size}}$$

Already calculated, No. of main memory block = 2048

No. of Cache memory " = 128

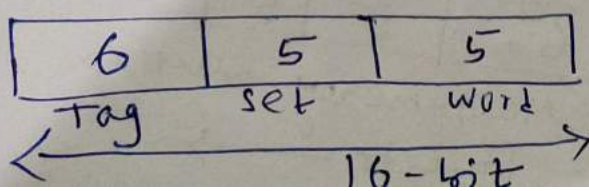
let set size = 4 blocks in cache (4-way set associative)

so, no. of sets in cache $= \frac{128}{4} = 32$ (sets)

Cache memory set-no. $\Rightarrow$

$0 \bmod 32 = 0$
$1 \bmod 32 = 1$
$32 \bmod 32 = 0$
$64 \bmod 32 = 0$

} mapping to cache (go to setno)



(main memory generated)

No. of MM Blocks in one set of cache (4 at a time)
= Main memory blocks / cache memory sets
= 2048/32 = 64 = 2^6 (Tag field)

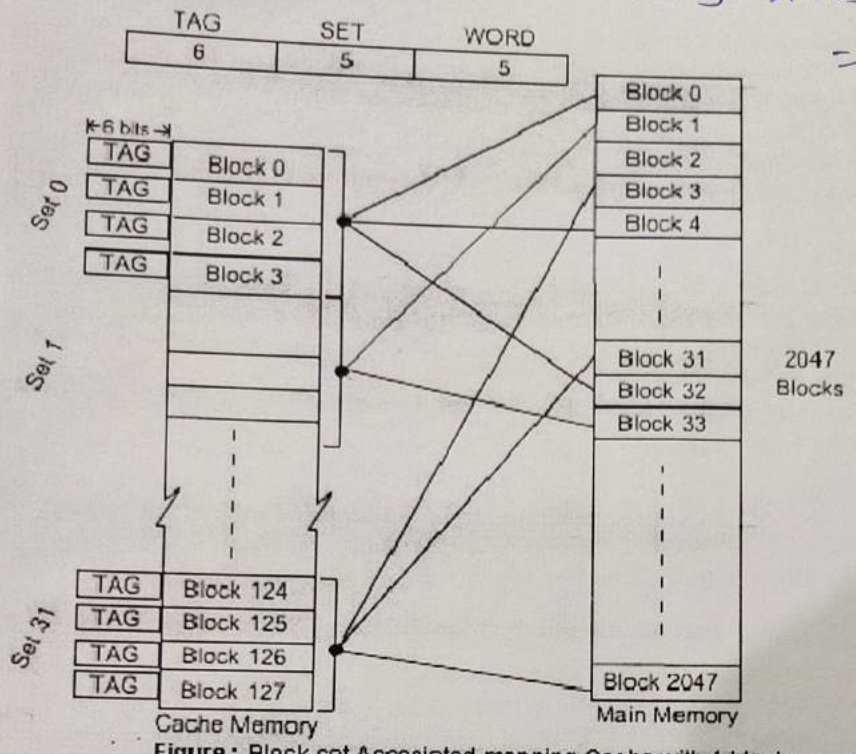


Figure : Block-set-Associated mapping Cache with 4 blocks per set

It is clear that if we increase the number of blocks per set, then the number of bits in SET field is reduced. Due to the increase of blocks per set, complexity of search is also increased. The extreme condition of 128 blocks per set requires no set bits and corrsponds to the fully associative mapping technique with **11 TAG** bits. The other extreme of one block per set is the direct mapping method.

Replacement Algorithms

When a new block must be brought into the cache and all the positions that it may occupy are full, a decision must be made as to which of the old blocks is to be overwritten. In general, a policy is required to keep the block in cache when they are likely to be referenced in near future. However, it is not easy to determine directly which of the block in the cache are about to be referenced. The property of locality of reference gives some clue to design good replacement policy.

Least Recently Used (LRU) Replacement policy:

Since program usually stay in localized areas for reasonable periods of time, it can be assumed that there is a high probability that blocks which have been referenced recently will also be referenced in the near future. Therefore, when a block is to be overwritten, it is a good decision to overwrite the one that has gone for longest time without being referenced. This is defined as the least recently used (LRU) block. Keeping track of LRU block must be done as computation proceeds.

Consider a specific example of a four-block set. It is required to track the LRU block of this four-block set. A 2-bit counter may be used for each block.

When a hit occurs, that is, when a read request is received for a word that is in the cache, the counter of the block that is referenced is set to 0. All counters which values originally lower than the referenced one are incremented by 1 and all other counters remain unchanged.

When a miss occurs, that is, when a read request is received for a word and the word is not present in the cache, we have to bring the block to cache.

There are two possibilities in case of a miss:

If the set is not full, the counter associated with the new block loaded from the main memory is set to 0,and the values of all other counters are incremented by 1.
If the set is full and a miss occurs, the block with the counter value 3 is removed , and the new block is put in its palce. The counter value is set to zero. The other three block counters are incremented by 1.It is easy to verify that the counter values of occupied blocks are always distinct. Also it is trivial that highest counter value indicates least recently used block.

First In First Out (FIFO) replacement policy:
A reasonable rule may be to remove the oldest from a full set when a new block must be brought in. While using this technique, no updation is required when a hit occurs. When a miss occurs and the set is not full, the new block is put into an empty block and the counter values of the occupied block will be increment by one. When a miss occurs and the set is full, the block with highest counter value is replaced by new block and counter is set to 0, counter value of all other blocks of that set is incremented by 1. The overhead of the policy is less, since no updation is required during hit.

Random replacement policy:
The simplest algorithm is to choose the block to be overwritten at random. Interestingly enough, this simple algorithm has been found to be very effective in practice.

Main Memory

The main working principle of digital computer is *Von-Neumann* stored program principle. First of all we have to keep all the information in some storage, mainly known as main memory, and CPU interacts with the main memory only. Therefore, memory management is an important issue while designing a computer system.On the otherhand, everything cannot be implemented in hardware, otherwise the cost of system will be very high. Therefore some of the tasks are performed by software program. Collection of such software programs are basically known as operating systems. So operating system is viewed as extended machine. Many more functions or instructions are implemented through software routine. The operating system is mainly memory resistant, i.e., the operating system is loaded into main memory.Due to that, the main memory of a computer is divided into two parts. One part is reserved for operating system. The other part is for user program. The program currently being executed by the CPU is loaded into the user part of the memory. In a uni-programming system, the program currently being executed is loaded into the user part of the memory. In a multiprogramming system, the user part of memory is subdivided to accommodate multiple process. The task of subdivision is carried out dynamically by opeartingsystem and is known as memory management.

8. The Memory Hierarchy (2) - The Cache

The uppermost level in the memory hierarchy of any modern computer is the *cache.* It first appeared as the memory level between the CPU and the main memory. It is the fastest part of the memory hierarchy, and the smallest in dimensions.

Many modern computers have more than one cache, it is common to find an instruction cache together with a data cache. and in many systems the caches are hierarchy structured by themselves: most microprocessors in the market today have an internal cache, with a size of a few KBytes, and allow an external cache with a much larger capacity, tens to hundreds of KBytes.

Some Values

There is a large variety of caches with different parameters. Below are listed some of the parameters for the external cache of a DEC 7000 system which is built around the 21064 ALPHA chip:

Block size (line size)	64 Bytes
Hit time	5 clock cycles
Miss penalty	340 ns
Access time	280 ns
Transfer time	60 ns
Cache size	4 MBytes
CPU clock rate	182 MHz

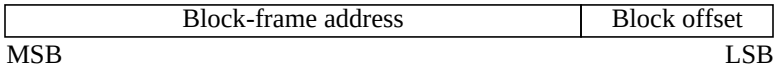
Placing a block in the cache

Freedom of placing a block into the cache ranges from absolute, when the block can be placed anywhere in the cache, to zero, when the block has a strictly predefined position.

- a cache is said to be **directly mapped** if every block has a unique, predefined place in the cache;
- if the block can be placed anywhere in the cache the cache is said to be **fully** associative;
- if the block can be placed in a restricted set of places then the cache is called **set associative.** A set is a group of two or more blocks; a block belongs to some predefined set, but inside the set it can be placed anywhere. If a set contains n blocks then the cache is called **n-way set associative.**

Obviously *direct-mapped* and *fully-associative* are particular names for a 1-way set associative and k-way set associative (for a cache with k blocks) respectively.

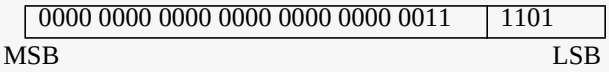
Transfers between the lower level of the memory and the cache occur in blocks: for this reason we can see the memory address as divided in two fields:



Example 8.1 MEMORY ADDRESS:

What is the size of the two fields in an address if the address size is 32 bits and the block is 16 Byte wide?

Answer:
Assuming that the memory is byte addressable there are 4 bits necessary to specify the position of the byte in the block. The other 28 bits in the address identify a block in the lower level of the memory hierarchy.



The address above refers to block number 3 in the lower level; inside that block the byte number 13 will be accessed.

The usual way to map blocks to positions in the cache is:

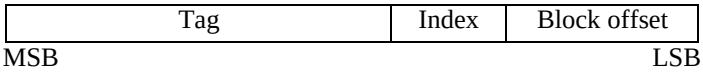
- for a direct mapped cache:
  index = (Block-frame address) modulo (number of blocks in the cache);

- for a set associative cache:
  index = (block-frame address) modulo (number of sets in the cache).

For a cache that has a power of two blocks (suppose 2^m blocks), finding the position is a direct mapped cache is trivial: position (index) is indicated by the last (the least significant) $\log_2 m$ bits of the block-frame address.

For a set associative cache that has a power of two sets (suppose 2^k sets), the set where a given block has to be mapped is indicated by the last (the least significant) $\log_2 k$ bits of the block-frame address.

The address can be viewed as having three fields: the block-frame address is split into two fields, the tag and the index, plus the block offset address:



In the case of a direct mapped cache the index field specifies the position of the block in the cache. For a set associative cache the index fields specifies in which set the block belongs. As for a fully associative cache this field has zero length.

Example 8.2 POSITION OF BLOCKS:

A CPU has a 7 bit address; the cache has 4 blocks 8 bytes each. The CPU addresses the byte at address 107. Suppose this is a miss and show where will be the corresponding block placed.

Answer:
$(107)_{10} = (1101011)_2$

With an 8 bytes block the least significant three bits of the address (011) are used to indicate the position of a byte within a block.
The most significant four bits ($(1101)_2 = 13_{10}$) represent the block-frame address, i.e. the number of the block in the lower level of the memory.
Because it is a direct mapped cache, the position of block number 13 in the cache is given by:

(Block-frame address) modulo (number of blocks in the cache)
$= 13 \bmod 4 = 1$

Hence the block number 13 in the lower level of the memory hierarchy will be placed in position 1 into the cache. This is precisely the same as using the last

$\log_2 4 = 2$

bits (01), the index, of the block-frame address (1101).

Figure 8.2 is a graphical representation for this example. Figures 8.1 and are graphical representations of the same problem we have in example but for fully associative and set associative caches respectively.

Because the cache is smaller than the memory level below it, there are several blocks that will map to the same position in the cache; using the Example 8.2 it is easy to see that blocks number 1, 5, 9, 13 will all map to the same position. The question now is: how can we determine if the block in the memory is the one we are looking for, or not?

Finding a Block in the Cache

Each line in the cache is augmented with a *tag* field that holds the tag field of the address corresponding to that block. When the CPU issues an address, there are, possibly, several blocks in the cache that could contain the desired information. The one will be chosen that has the same tag as that of the address issued by the CPU.

Figure 8.4 presents the same cache we had in figures 8.1 to 8.3, improved with the tag fields. In the case of a fully associative cache all tags in the cache must be checked against the address's tag field; this because in a fully associative cache blocks may be placed anywhere. Because the cache must be very fast, the checking process must be done in parallel, all cache's tags

must be compared at the same time with the address tag fields. For a set associative cache there is less work than in a fully associative cache: there is only one set in which the block can be; therefore only the tags of the blocks in that set have to be compared against the address tag field.

If the cache is direct mapped, the block can have only one position in the cache: only the tag of that block is compared with the address tag field.

There must also be a way to indicate the content of a block must be ignored. When the system starts up for instance, there will be some binary configurations in every tag of the cache; they are meaningless at this moment; however some of them could match the tag of an address issued by the processor thus delivering bad data. The solution is a bit for every cache line, which indicates if that line contains valid data. This bit is called the **valid bit** and is initialized to Non-valid (0) when the system starts up.

Figure 8.5 presents a direct mapped cache schematic; a comparator (COMP) is used to check if the Tag field of the CPU address matches the content of the tag field in the cache at address Index. The valid bit at that address must be Valid (1) to have a hit when the tags are the same. The multiplexor (MUX) at the Data outputs is used to select that part of the block we need.

Figure 8.6 presents the status of a four line, direct mapped cache, similar to the one we had in Example 8.2 after a sequence of misses; suppose that after reset (or power-on), the CPU issues the following sequence of reads at addresses (in decimal notation): 78, 79, 80, 77, 109, 27, 81. Hits don't change the state of the cache when only reads are performed; therefore only the state of the cache after misses is presented in Figure 8.6. Below is the binary representation of addresses involved in the process:

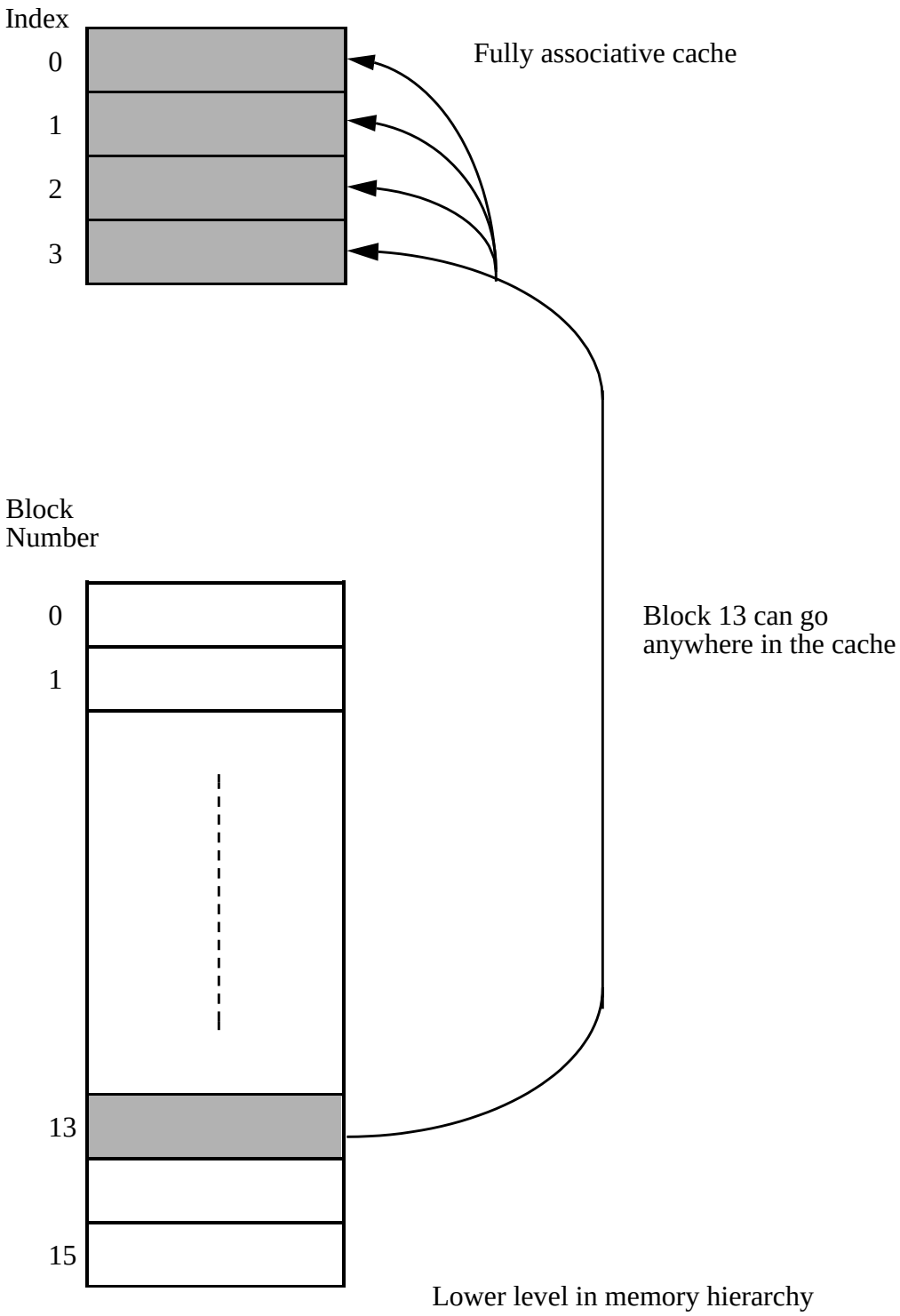


FIGURE 8.1 A fully associative four blocks (lines) cache connected to a 16 blocks.

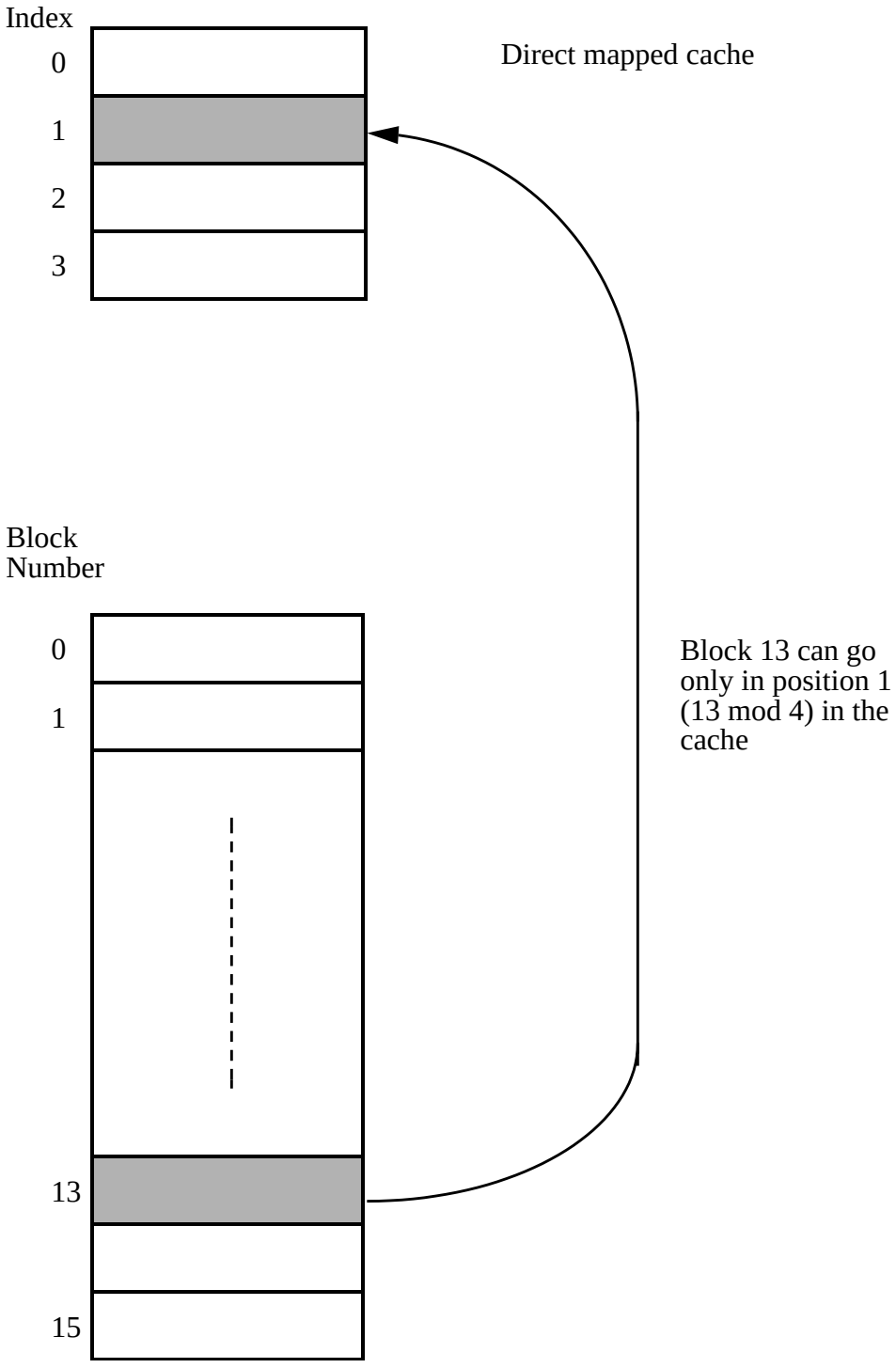


FIGURE 8.2 A Direct mapped, four blocks (lines) cache connected to a 16 blocks memory.

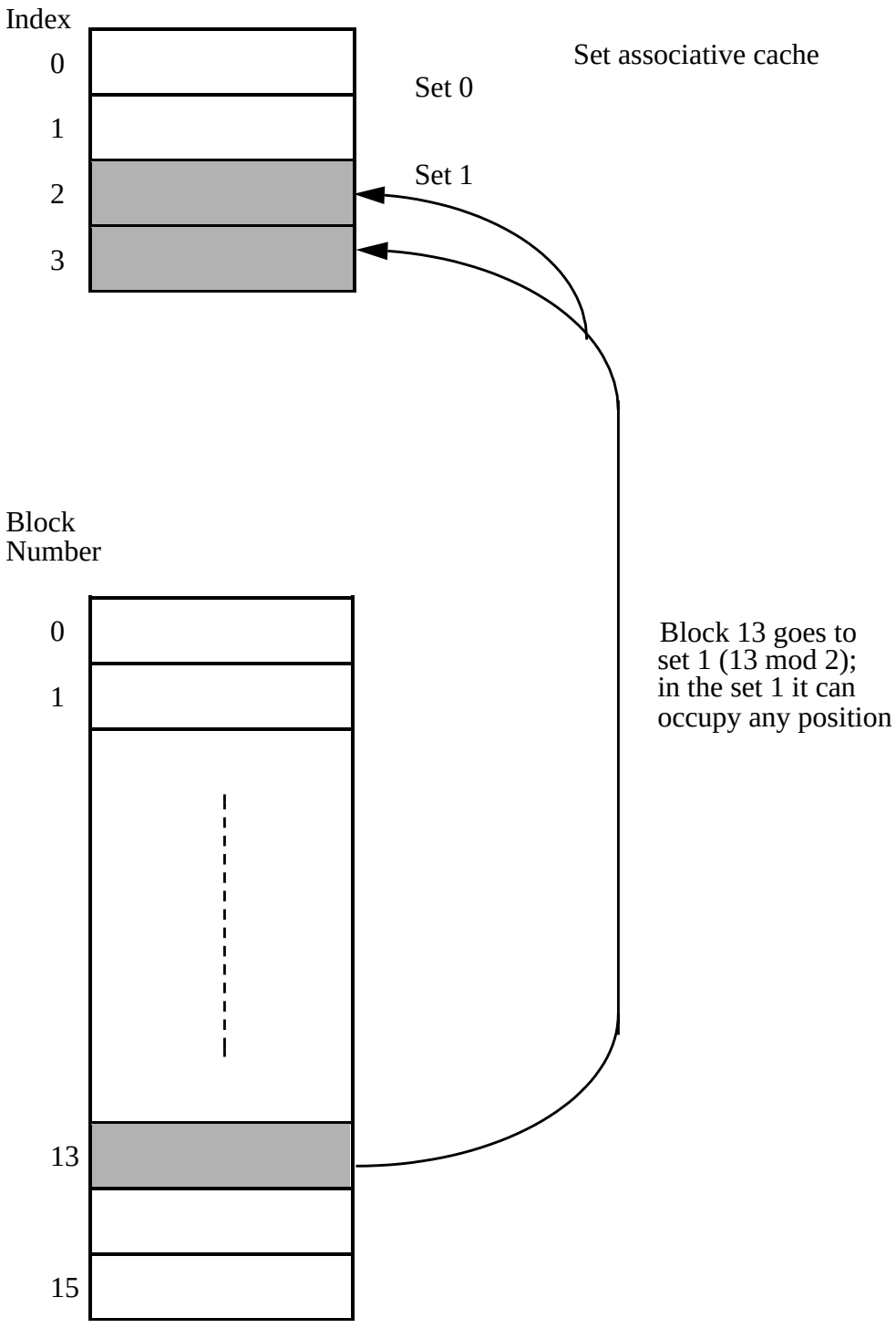


FIGURE 8.3 A 2-way set-associative cache connected to a 16 blocks memory.

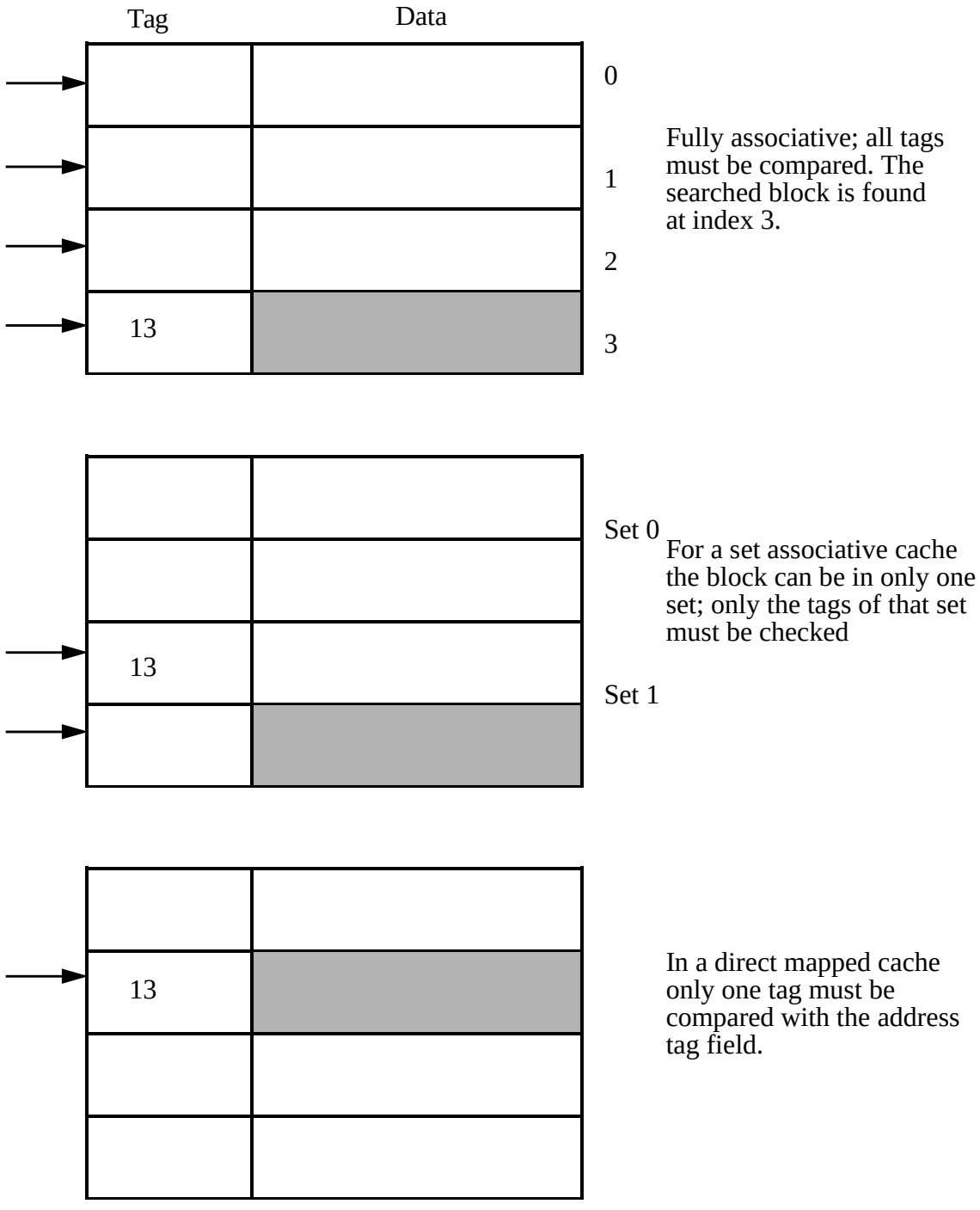


FIGURE 8.4 Finding a block in the cache implies comparing the tag field of the actual address with the content of one or more tags in the cache.

Address	Tag	Index	Block offset
78	10	01	110
79	10	01	111
80	10	10	000
77	10	01	101
109	11	01	101
27	00	11	011
81	10	10	001

- Address 78: miss because the valid bit is 0 (Not Valid); a block is brought and placed into the cache in position Index = 01
- Address 79: hit; as Figure 8.6.b points out the content of this memory address is already in the cache
- Address 80: miss because the valid bit at index 10 in the cache is 0 (Not Valid); a block is brought into the cache and placed at this index.
- Address 77: hit, found at index 01 in the cache.
- Address 109: miss; the block being transferred from the lower level of the hierarchy is placed in the cache at index 01, thus replacing the previous block.
- Address 27: miss; block transferred into the cache at index 11.
- Address 81: hit; the item is found in the cache at index 10.

It is a common mistake to neglect the tag field when computing the amount of memory necessary for a cache.

Example 8.3 COMPUTATION OF MEMORY REQUIRED BY A CACHE:

A 16 KB cache is being designed for a 32 bit system. The block size is 16 bytes, and the cache is direct mapped. Which the total amount of memory needed to implement this cache?

Answer:
The cache will have a number of lines equal with:

$$\frac{\text{cache capacity}}{\text{blocksize}} = \frac{16\text{KB}}{16\text{ B}} = 1\text{ KB} = 2^{10}\text{lines}$$

Hence the number of bits in the index field of an address is 10. The tag field in an address is:

32 - 3 - 10 = 19 bits(3 bits are needed as block offset)

Each line in the cache needs a number of bits equal to:

1 + 19 + 16 * 8 = 148 bits

The total amount of memory for the cache is:

line_size * number_of_lines = 148 * 2^{10} = 151.5 Kbit = 18.9 KB roughly

This figure is by 18% larger than the “useful” size of the cache, and is hardly negligible.

Replacing Policies

Or in other words, answering the question “which block should go out in the case of a cache miss?”. The replacing policy depends upon the type of cache. For a direct mapped cache the decision is very simple: because a block can go in only one place, the block in that position will be replaced. This simplifies the hardware (remember that a cache is hardware managed).

For fully associative and set-associative caches a block may go in several positions (at different indexes), and, as a result, there are different possibilities to choose a block that will be replaced. Note that, due to the high hit rates in the caches (high hit rates are a must for good access times), the decision is painful, with a high probability we will replace blocks that contain useful information.

The most used policies for replacement are:

- **random:** this technique is very simple, one block is selected at random and replaced.
- **LRU** *(Least Recently Used)*: in this approach accesses to the cache are recorded; the block that will be replaced is the one that has been unused (unaccessed) for the longest period of time. This technique is a direct consequence of the temporal locality principle: if blocks tend to be accessed again soon then it seems natural to discard the one that has been of little use in the past.

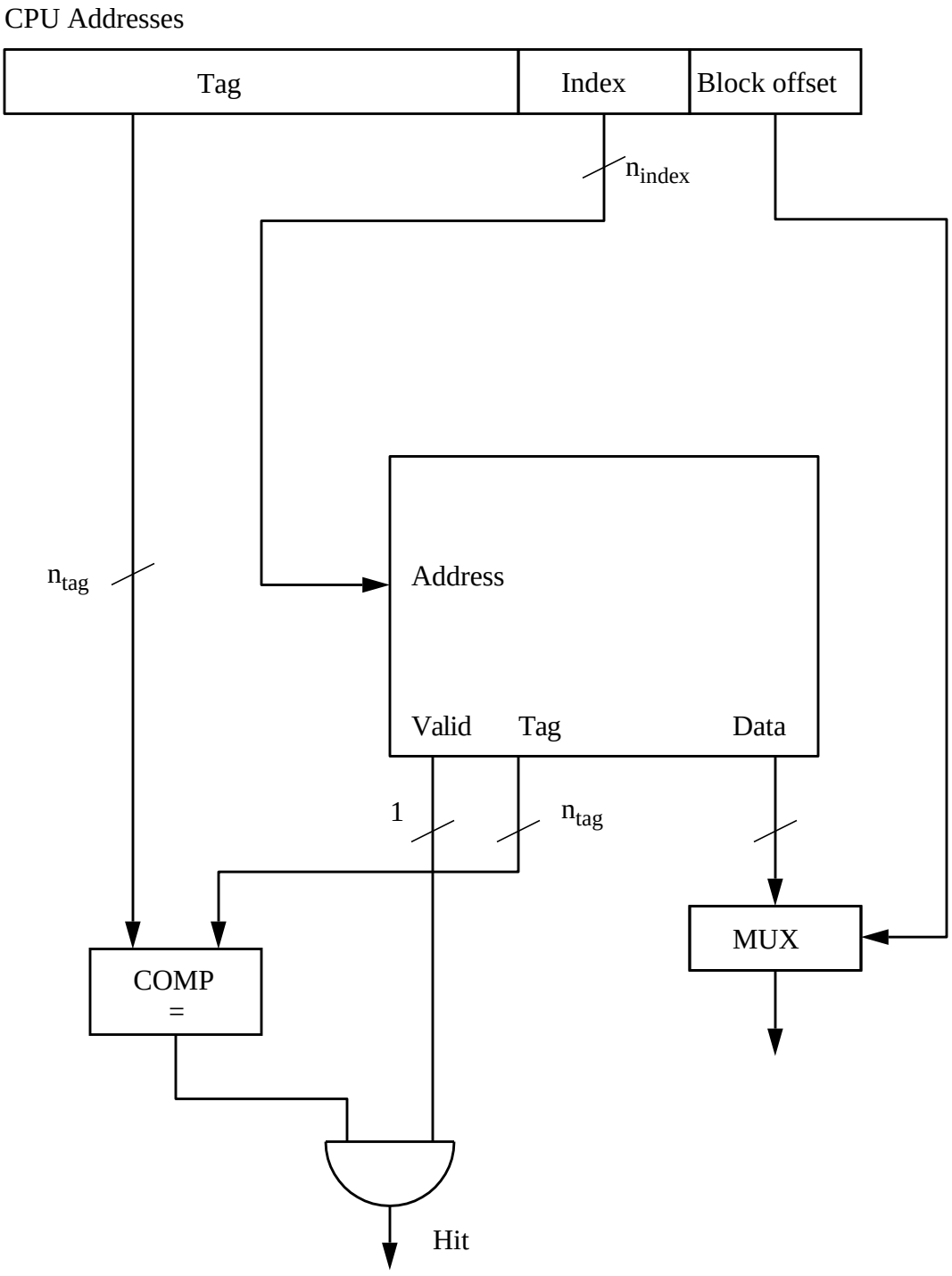


FIGURE 8.5 a direct mapped cache schematic.

Index	V	Tag	Data							
00	0									
01	0									
10	0									
11	0									

a. The initial state of cache after power on. The Tag and Data fields contain some arbitrary binary configurations which are not shown.

Index	V	Tag	Data							
00	0									
01	1	10	M[72]	M[73]	M[74]	M[75]	M[76]	M[77]	M[78]	M[79]
10	0									
11	0									

b. After the miss at address 78.

Index	V	Tag	Data							
00	0									
01	1	10	M[72]	M[73]	M[74]	M[75]	M[76]	M[77]	M[78]	M[79]
10	1	10	M[80]	M[81]	M[82]	M[83]	M[84]	M[85]	M[86]	M[87]
11	0									

c. After the miss at address 80.

Index	V	Tag	Data							
00	0									
01	1	11	M[104]	M[105]	M[106]	M[107]	M[108]	M[109]	M[110]	M[111]
10	1	10	M[80]	M[81]	M[82]	M[83]	M[84]	M[85]	M[86]	M[87]
11	0									

d. After the miss at address 109. The previous block at index 01 has been replaced.

Index	V	Tag	Data							
00	0									
01	1	11	M[104]	M[105]	M[106]	M[107]	M[108]	M[109]	M[110]	M[111]
10	1	10	M[80]	M[81]	M[82]	M[83]	M[84]	M[85]	M[86]	M[87]
11	1	00	M[24]	M[25]	M[26]	M[27]	M[28]	M[29]	M[30]	M[31]

e. After the miss at address 27.

FIGURE 8.6 The cache after handling the sequence of addresses: 78 (miss), 79 (hit), 80 (miss), 77 (hit), 109 (miss), 81 (hit).

- **FIFO** *(First In First Out)*: the oldest block in the cache (or in the set for a set associative cache) is selected for replacement. This policy does not take into account the addressing pattern in the past: it may happen the block has been heavily used in the previous addressing cycles, and yet it is chosen for replacement. The FIFO policy is outperformed by the random policy which has, as a plus, the advantage of being easier to implement.

As a matter of fact, almost all cache implementations use either random or LRU for block replacement decision. The LRU policy delivers slightly better performance than random, but it is more difficult to implement: at every access the least recently used block must be determined and marked somehow. For instance, each block could have associated a hardware counter (a software one would be too slow), called *age counter;* when a block is addressed its counter is set to zero, and all other ones are incremented by one. When a block must be replaced, the decision block must find the block with the highest value in its age counter. Obviously the hardware resources are more expensive than for a random policy, and, what is worse, the algorithm is complicated enough to slow down the cache, as compared with a random decision.

Example 8.4 CONTENTS OF A CACHE:

Consider a fully associative four block cache, and the following stream of block-frame addresses: 2, 3, 4, 2, 5, 2, 3, 1, 4, 5, 2, 2, 2, 3. Show the content of the cache in two cases:
a) using a LRU algorithm for replacing blocks;
b) using a FIFO policy.

Answer:

For the LRU replacement policy:

Address:

2	3	4	2	5	2	3	1	4	5	2	2	2	3
2_1	2_2	2_3	2_1	2_2	2_1	2_2	2_3	2_4	5_1	5_2	5_3	5_4	5_5
	3_1	3_2	3_3	3_4	3_5	3_1	3_2	3_3	3_4	2_1	2_1	2_1	2_2
		4_1	4_2	4_3	4_4	4_5	1_1	1_2	1_3	1_4	1_5	1_6	3_1
				5_1	5_2	5_3	5_4	4_1	4_2	4_3	4_4	4_5	4_6
M	M	M		M			M	M	M	M			M

For the FIFO policy:

Address:

2	3	4	2	5	2	3	1	4	5	2	2	2	3
2_*	2_*	2_*	2_*	2_*	2_*	2_*	1	1	1	1	1	1	1
	3	3	3	3	3	3	3_*	3_*	3_*	2	2	2	2
		4	4	4	4	4	4	4	4	4_*	4_*	4_*	3
				5	5	5	5	5	5	5	5	5	5_*
M	M	M		M			M			M			M

For the LRU policy, the subscripts indicate the age of the blocks in the cache. For the FIFO policy a star is used to indicate which is the next block to be replaced. The Ms under the columns of tables indicate the misses.

For the short sequence of block-frame addresses in this example, the FIFO policy yields a smaller number of misses, 7 as compared with 9 for the LRU. However in most cases the LRU strategy proves to be better than FIFO.

Cache Write Policies

So far we have discussed about how reads are handled in a cache. Writes are more difficult and affect the performance more than reads do. If we take a closer look at the block scheme in Figure 8.5 we realize that, in the case of a read, the two basic operations are performed in parallel: the tag and reading the block are read at the same time. Further, the tags must be compared, and the delay in the comparator (COMP) is slightly higher then the delay through the multiplexor (MUX): if we have a hit then the data is already stable at the cache's outputs; if there a miss there is no harm in reading some improper data from the cache, we simply ignore it.

When we come to writes we realize that the sequence of operations is longer than for a read: the problem is that, for most caches, only a part of the block will be modified; if the block is 16 Bytes wide, and the CPU writes a byte, then only that byte must be changed. This implies a read-modify-write sequence in the cache: read the whole block, modify the needed portion, write the new configuration of the block. Of course the block can not be changed until a hit/miss decision is taken.

There are two options when writing into the cache, depending upon how the information in the lower lever of the hierarchy is updated:

- **write through:** the item is written both into the cache and into the corresponding block in the lower level of the hierarchy; as a

result, the blocks in the lower level of the hierarchy contains at every moment the same information as the blocks in the cache;

- **write back:** writes occur only in the cache; the modified block is written into the lower level of the hierarchy only when it has to be replaced.

With the write-back policy there is useless to write back a block (i.e. to write a block into the lower level of the hierarchy) if the block has not been modified while in the cache. To keep track if a block was modified or not, a bit, called the **dirty bit**, is used for every block in the cache; when the block is brought into the cache this bit is set to Not-dirty (0); the first write in that block sets the bit to Dirty (1). When the replacement decision is taken, the control checks if the block is *dirty* or *clean*. If the block is dirty it has to be to the lower level of the memory; otherwise a new block coming from the lower level of the hierarchy can simply overwrite that block in the cache.

For fully or set associative caches, where several bocks may candidate for replacement, it is common to prefer the one which is clean (if any), thus saving the time necessary to transfer a block from the cache to the lower level of the memory.

The two cache write policies have their advantages and disadvantages:

- write through: this is easy to implement, and has the advantage that the memory has the most recent value of data; this property is especially attractive in multiprocessing and I/O. The drawback is that writes going to the lower level in memory are slower. When the CPU has to wait for a write to complete it is said to **write stall**. A simple way to reduce write stalls is to have a **write buffer**. which allows CPU to continue working while the memory is updated; this works fine as long as the rate at which writes occur is lower than the rate at which transfers from the buffer to the memory can be done.

- write back: is more difficult to implement but has the advantage that writes occur at the cache's speed; moreover writes are local to the cache and don't require access to the system bus, unless a dirty block has to be transferred from the cache to the memory. So this write policy uses less memory bandwidth, which is attractive for multiprocessing where several CPUs share the system's resources. Another disadvantage, besides greater hardware complexity, is that read misses may require writes to the memory, in the case a block has to be transferred into the lower level of the hierarchy.

The Cache Performance

As we discussed very early in this course, the ultimate goal of a designer is to reduce the CPU_{time} for a program. When connected with a memory, we must account both for the execution time of the CPU and for its stalls:

$CPU_{time} = (CPU_{exec} + Memory\_stalls) * T_{ck}$

where both the execution time and stalls are expressed in clock cycles.

Now the natural question we may ask is: do we include the cache access time in the CPU_{exec} or in Memory_stalls? Both ways are possible: it is possible to consider the cache access time in Memory_stalls, simply because the cache is a part of the memory hierarchy. On the other hand, because the cache is supposed to be very fast, we can include the hit time in the CPU execution time as the item sought in the cache will be delivered very quickly, maybe during the same execution cycle. As a matter of fact this is the widely accepted convention.

Memory_stalls will include the stall due to misses, for reads and writes:

Memory_stalls = Mem_accesses_per_program * miss_rate * miss_penalty

We now get for the CPU_{time}:

$CPU_{time} = (CPU_{exec} + Mem\_accesses\_per\_program*miss\_rate*miss\_penalty)*T_{ck}$

which can be further modified by factoring the IC (Instruction Count):

$CPU_{time} = IC*(CPI_{exec} + Mem\_accesses\_per\_instruction*miss\_rate*miss\_penalty)*T_{ck}$

The above formula can be also written using misses per instruction as:

$CPU_{time} = IC*(CPI_{exec} + Misses\_per\_instruction*miss\_penalty)*T_{ck}$

Example 8.5 CPU PERFORMANCE WITH CACHE:

The average execution time for instructions in some CPU is 7 (ignoring stalls); the miss penalty is 10 clock cycles, the miss rate is 5%, and there are, on average, 2.5 memory accesses per instruction. What is the CPU performance if the cache is taken into account?

Answer:

$CPU_{time} = IC*(CPI_{exec} + Mem\_accesses\_per\_instruction*miss\_rate*miss\_penalty)*T_{ck}$

CPU_{time} (with cache) = IC*(7 + 2.5*0.05*10)*T_{ck} = IC*8.25*T_{ck}

The IC and T_{ck} are the same in both cases, with and without cache, so the result of including the cache's behavior is an increase in CPU_{time} by $\frac{8.25}{7} - 1 = 17.8\%$

The following example presents the impact of the cache for a system with a lower CPI (as is the case with pipelined CPUs):

Example 8.6 CPU PERFORMANCE WITH CACHE AND CPI:

The CPI for a CPU is 1.5, there are on the average 1.4 memory accesses per instruction, the miss rate is 5%, and the miss penalty is 10 clock cycles. What is the performance if the cache is considered?

Answer:

CPU_{time} = IC*(CPI_{exec} + Mem_accesses_per_instruction*miss_rate*miss_penalty)*T_{ck}

CPU_{time} (with cache) = IC*(1.5 + 1.4*0.05*10)*T_{ck} = IC*2.2*T_{ck}

This means an increase in CPU_{time} by 46%.

Note that for a machine with lower CPI the impact of the cache is more significant than for a machine with a higher CPI.

The following example shows the impact of the cache on system with different clock rates.

Example 8.7 CPU PERFORMANCE WITH CACHE, CPI AND CLOCK RATES:

The same architecture is implemented using two different technologies, one which allows a clock cycle of 20ns and another one which permits a 10ns clock cycle. Two systems, built around CPUs in the two technologies, use the same type of circuits for their main memories: the miss penalty is, in both cases, 140ns. How does the cache behavior affect the CPU performance? Assume that the ideal CPI is 1.5, the miss rate is 5%, and there are 1.4 memory accesses per instruction on average.

Answer:

CPU_{time} = IC*(CPI_{exec} + Mem_accesses_per_instruction*miss_rate*miss_penalty)*T_{ck}

For the CPU running with a 20ns clock cycle, the miss penalty is 140/20 = 7 clock cycles, and the performance is given by:

CPU_{time1} = IC*(1.5 + 1.4*0.05*7)*T_{ck1} = IC*1.99*T_{ck1}

The effect of the cache, for this machine, is to stretch the execution time by 32%. For the machine running with a 10 ns clock cycle, the miss penalty is 140/10 = 14 clock cycles, and the performance is:

$CPU_{time2} = IC*(1.5 + 1.4*0.05*14)*T_{ck2} = IC*2.48*T_{ck2}$

The cache increases the CPU_{time}, for this machine, by 65%.

Example 8.7 clearly points out that the cache behavior gets more important while CPU are running faster. Neglecting the cache may completely compromise the performance of a CPU. For a given instruction set and a specified program the CPI_{exec} can be measured; the Instruction Count can also be measured, and T_{ck} is known for the given machine. Reducing the CPU_{time} can be achieved by:

- reducing the miss rate: the easy way is to increase the cache size; however there is a serious limitation in doing so for on-chip caches: the space. Most on-chip caches are only a few kilobytes in size.

- reducing the miss penalty: for most cases the access time dominates the miss penalty; while the access time is given by the technology used for memories, and, as a result can not be easily lowered, it is possible to use intermediate levels of cache between the internal cache (on-chip) and main memory.

Here is a short description of internal caches for several popular CPUs:

CPU	Instruction	Data
Intel 80486	8 KB	
Motorola 68040	4 KB	4 KB
Intel PENTIUM	8 KB	8 KB
DEC Alpha	8 KB	8 KB
Sun MicroSPARC	4 KB	2 KB
Sun SuperSPARC	20 KB	16 KB
Hewlett-Packard PA 7100	-	-
MIPS R4000	8 KB	8 KB
MIPS R4400	16 KB	16 KB
PowerPC 601	32 KB	

Sources for Cache Misses

Misses in a cache can have one of the three following sources:

- **compulsory:** when the program starts running the cache is empty (no block for that program yet);
- **capacity:** if the cache does not contain all the blocks needed for the execution of the program, then some blocks will be replaced and then, later, brought back into the cache;
- **conflict:** this happens in direct mapped and set associative caches if too many blocks map to the same position.

There is little to do against compulsory misses: increasing the block size reduces indeed the number of compulsory misses as the cache will be filled faster; the drawback is that bigger blocks may increase the number of conflict misses as there are fewer blocks in the cache.

Conflict misses seem to be easiest to resolve: a fully associative cache has no conflicts. However full associativity is very expensive in terms of hardware: more hardware tends to slow down the clock, yielding an overall poorer performance.

As for capacity misses, the solution is larger caches, both internal and external. If the cache is too small to fit the requirement of some program, then most of the time will be spent in transferring blocks between the cache and the lower level of the hierarchy; this is called **trashing.** A trashing memory hierarchy has a performance that is close to that of the memory in the lower level, or even poorer due to misses overhead.

Unified Caches or Instruction/Data Only?

Initial caches were meant to hold both data and instructions. This caches are called **unified** or mixed. It is possible however to have separate caches for instructions and data, as the CPU knows if it is fetching an instruction or loading/storing data. Having separate caches allows the CPU to perform an instruction fetch at the same time with a data read/write, as it happens in pipelined implementations. As the table in section 8.6 shows, most of the today's architectures have separate caches. Separate caches give the designer the opportunity to separately optimize each cache: they may have different sizes, different organizations, and block sizes. The main observation is that instruction caches have lower miss rates as data caches, for the main reason that instructions expose better spatial locality than data.

Exercises

Draw a fully associative cache schematic. Which are the hardware resources besides the ones required by a direct mapped cache? You must pick some cache capacity and some block size.

Redo the design in problem 8.1 but for a 4-way set associative cache. Compare your design with the fully associative cache and the direct mapped cache.

Design a 16 KB direct mapped cache for a 32 bit address system. The block size is 4 bytes (1 word). Compare the result with the result in Example 8.3.

Design (gate level) a 4 bit comparator. While most MSI circuits provide three outputs indicating the relation between the A and B inputs (A > B, A = B, A < B), your design must have only one output which gets active (1) when the two inputs are equal.

Assume you have two machines with the same CPU and same main memory, but different caches:
cache 1: a 16 set, 2-way associative cache, 16 bytes per block, write through;
cache 2: a 32 lines direct mapped cache, 16 bytes per block, write back.
Also assume that a miss takes 10 longer than a hit, for both machines. A word write takes 5 times longer than a hit, for the write through cache; the transfer of a block from the cache to the memory takes 15 times as much as a hit.
a) write a program that makes machine 1 run faster than machine 2 (by as much as possible);
b) write a program that makes machine 2 run faster than machine 1 (by as much as possible).

Write Through and Write Back in Cache

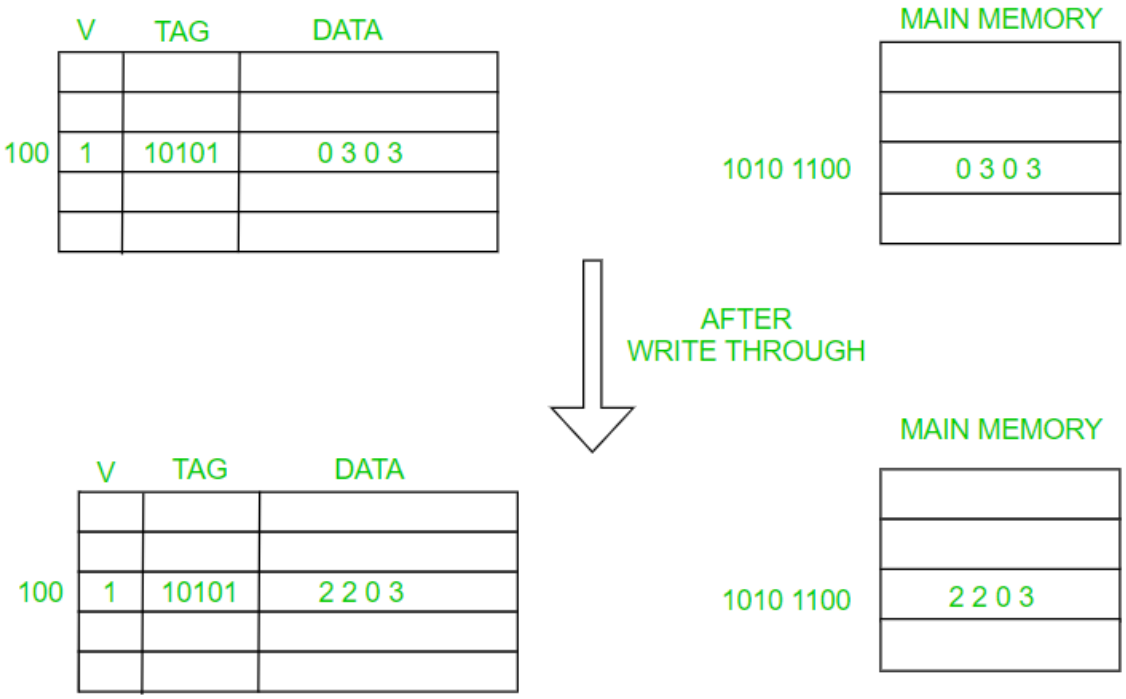
Cache is a technique of storing a copy of data temporarily in rapidly accessible storage memory. Cache stores most recently used words in small memory to increase the speed in which a data is accessed. It acts like a buffer between RAM and CPU and thus increases the speed in which data is available to the processor.

Whenever a Processor wants to **write** a word, it checks to see if the address it wants to write the data to, is present in the cache or not. If address is present in the cache i.e., **Write Hit**.

We can update the value in the cache and avoid a expensive main memory access.But this results in **Inconsistent Data** Problem.As both cache and main memory have different data, it will cause problem in two or more devices sharing the main memory (as in a multiprocessor system).

This is where **Write Through** and **Write Back** comes into picture.

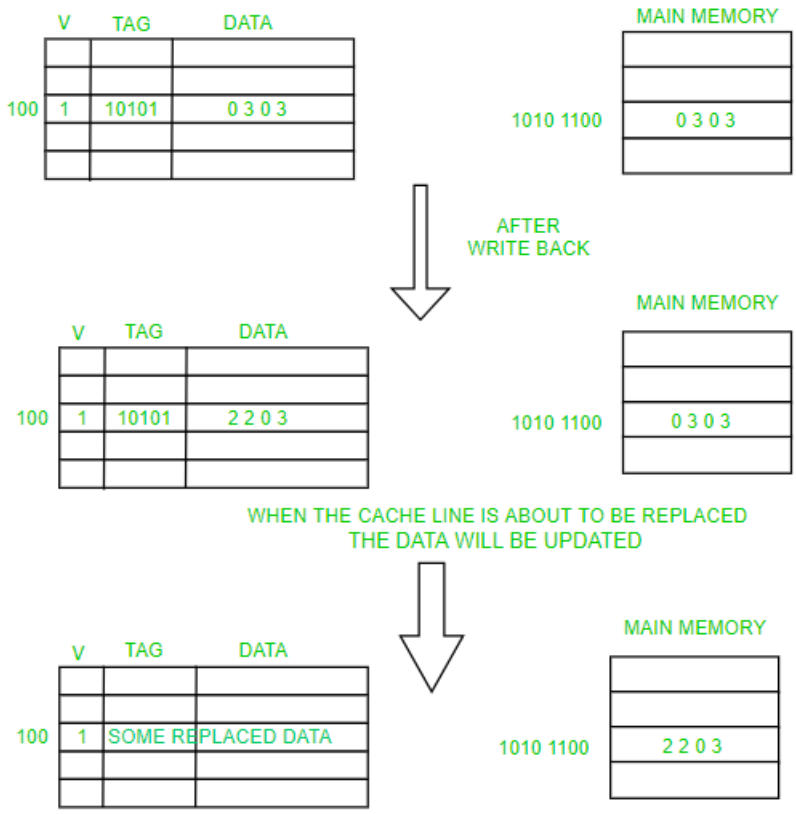
Write Through:



In write through, data is **simultaneously updated to cache and memory**. This process is simpler and more reliable.This is used when there are no frequent writes to the cache(Number of write operation is less).

It helps in data recovery (In case of power outage or system failure). A data write will experience latency (delay) as we have to write to two locations (both Memory and Cache). It Solves the inconsistency problem. But it questions the advantage of having a cache in write operation (As the whole point of using a cache was to avoid multiple accessing to the main memory).

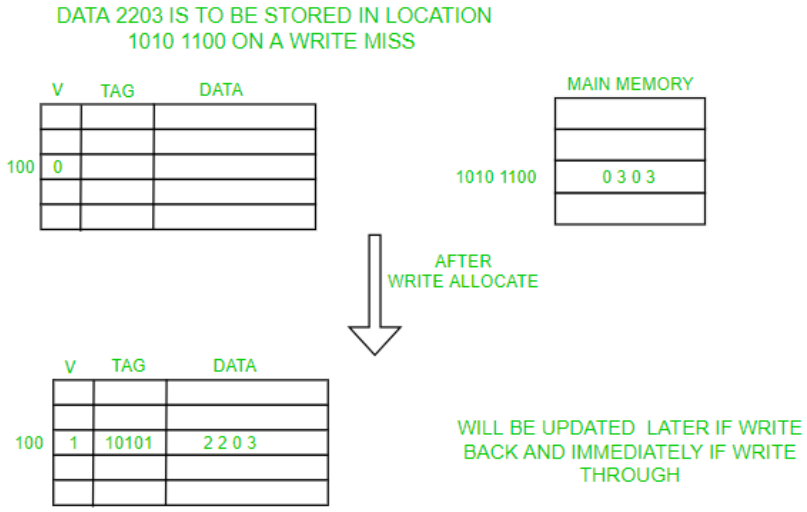
Write Back:



The data is updated only in the cache and updated into the memory in later time. Data is updated in the memory only when the cache line is ready to replaced (cache line replacement is done using Belady's Anomaly, Least Recently Used Algorithm, FIFO, LIFO and others depending on the application). Write Back is also known as Write Deferred.

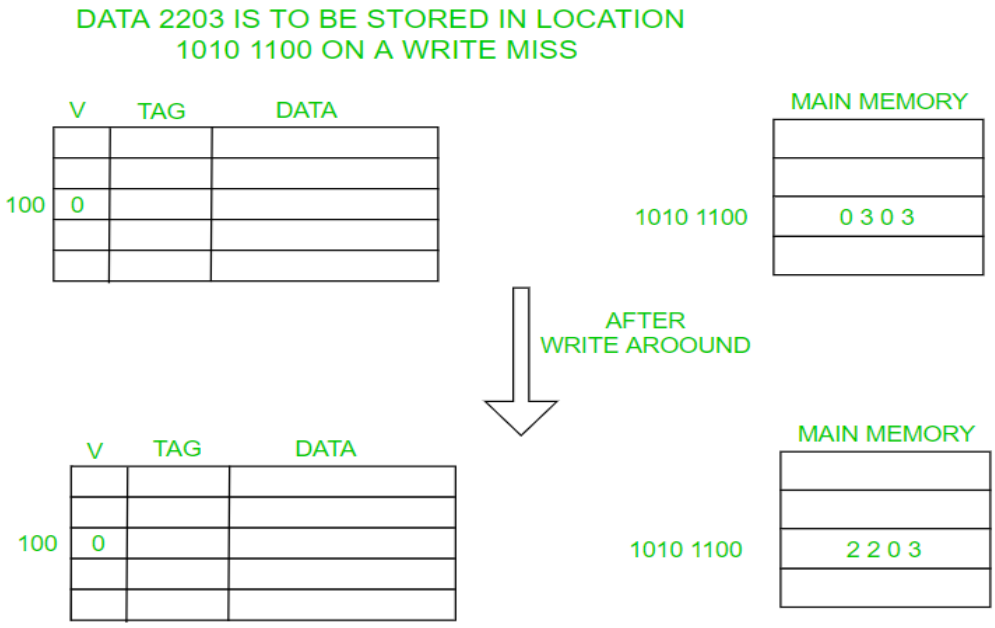
If write occurs to a location that is not present in the Cache(Write Miss), we use two options, **Write Allocation** and **Write Around**.

Write Allocation



In Write Allocation data is loaded from the memory into cache and then updated. Write allocation works with both Write back and Write through.But it is generally used with Write Back because it is unnecessary to bring data from the memory to cache and then updating the data in both cache and main memory. Thus Write Through is often used with No write Allocate.

Write Around:



Here data is Directly written/updated to main memory without disturbing cache.It is better to use this when the data is not immediately used again.