Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
**National Institute of Technology, Meghalaya**

Lecture - 30
Cache Memory (Part I)

Welcome to the next lecture on Cache Memory. I have already discussed and made the ground for cache memory. Now, I will be discussing in detail about the read and write strategies, the block replacement strategies, and of course; the mapping techniques that are used.

We know that cache memory is a fast memory that is in between processor and main memory. Frequently used data or instructions are brought into this memory and they are brought to the processor in turn for execution. For faster execution frequently used instruction and data brought here. So, the next time when the processor needs that data or instruction, it is getting from the cache.

So the benefit we are getting is most of the time we are accessing the data from the cache rather than from main memory, but for the first time there will be only miss and the data must be brought from lower level of the memories that is main memory or L2 cache or L3 cache into your L1 cache memory.

(Refer Slide Time: 01:45)
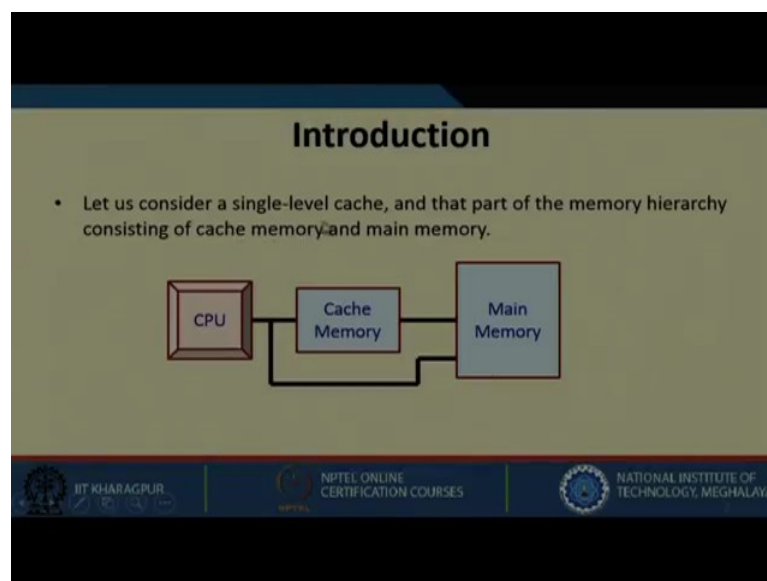
Let us consider a single level cache, that is, the memory hierarchy consists of the cache memory and main memory. So, this is the CPU, this is the main memory and this is the cache memory.

(Refer Slide Time: 01:58)

- Cache memory is logically divided into *blocks* or *lines*, where every block (line) typically contains 8 to 256 bytes.
- When the CPU wants to access a word in memory, a special hardware first checks whether it is present in cache memory.
  - If so (called *cache hit*), the word is directly accessed from the cache memory.
  - If not, the block containing the requested word is brought from main memory to cache.
  - For writes, sometimes the CPU can also directly write to main memory.
- Objective is to keep the commonly used blocks in the cache memory.
  - Will result in significantly improved performance due to the property of locality of reference.
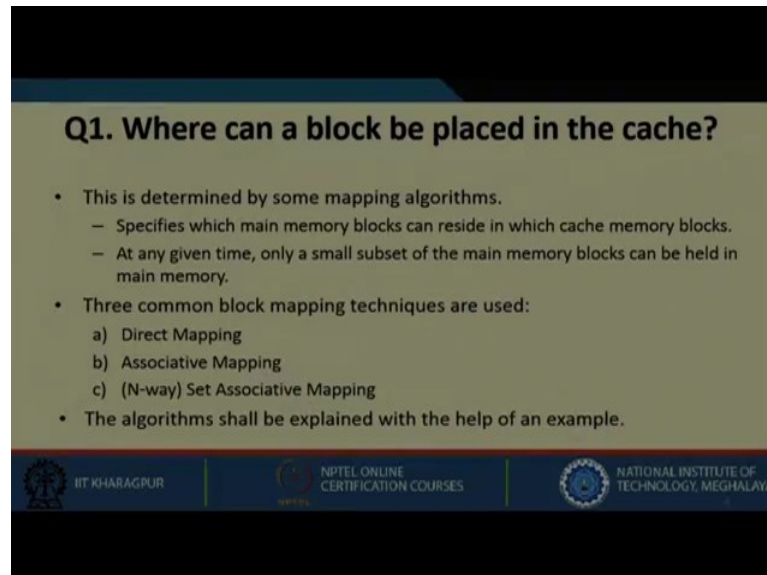
Cache memory is logically divided into blocks or lines, where every block or line typically contains 8 to 256 bytes. When the CPU wants to access a word in memory, a special hardware first checks whether it is present in cache memory. If so we call it a cache hit and the word is directly accessed from the cache memory. If not, the block containing the requested word is brought from main memory to cache memory.

For writes, sometime the CPU can also directly write to the main memory, or it can be written back into the cache, and later it is updated into the main memory. So, cache memory is divided into blocks and whenever the CPU generates some address, we first check whether the requested word is present in the cache or not. If it is present in the cache that particular word is sent to the processor. If that particular word is not present in the cache then we bring that particular word from main memory into cache memory, and then it is transferred to the processor.

So whenever a cache hit occurs; that means, the word is in cache memory, we can directly access the word from there. If it is a miss, then the requested word is brought from main memory to cache memory; similarly for write also. We will see in detail that in write what exactly happens. Objective is to keep the commonly used blocks in the

cache memory. It will result in significantly improved performance due to property of locality of reference.

(Refer Slide Time: 04:43)

You remember we said that we will be answering 4 questions. The first one is, where can a block be placed in the cache? Firstly, where my block will be placed in the cache is determined by mapping algorithms. This specifies which main memory block can reside in which cache memory block. Which block of the main memory will be in which block of the cache must be determined using some mapping technique.
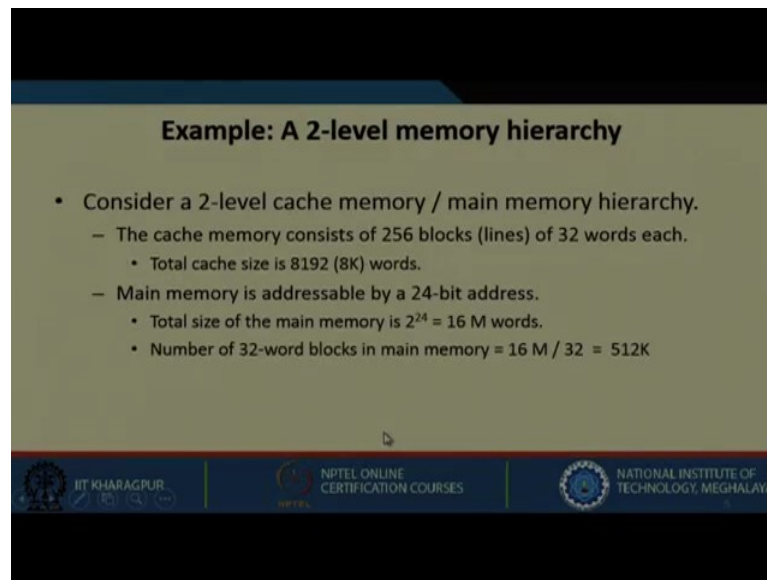
In this context we have 3 mapping techniques: direct mapping, associative mapping, and set associative mapping. One thing is clear, that cache is small so we cannot bring everything into cache. Some blocks of main memory at a time can be brought into cache. So, some blocks that are required can be brought into in the cache, and can be used by the processor; and when some other blocks are required then some of the blocks that are present in the cache must be replaced such that we can bring new blocks from the cache. Again for that we have separate strategies we will look into that late little later.

Now let us see the algorithms one by one. Consider a 2-level memory hierarchy having cache memory and main memory; with this example we will be taking into consideration all the mapping algorithms. The cache memory consists of 256 blocks or lines of 32 words. So, each block is having 32 words. Total cache sizes 8192, that is 8 Kwords. And how is the main memory organized? Main memory is addressable by a 24 bit address.
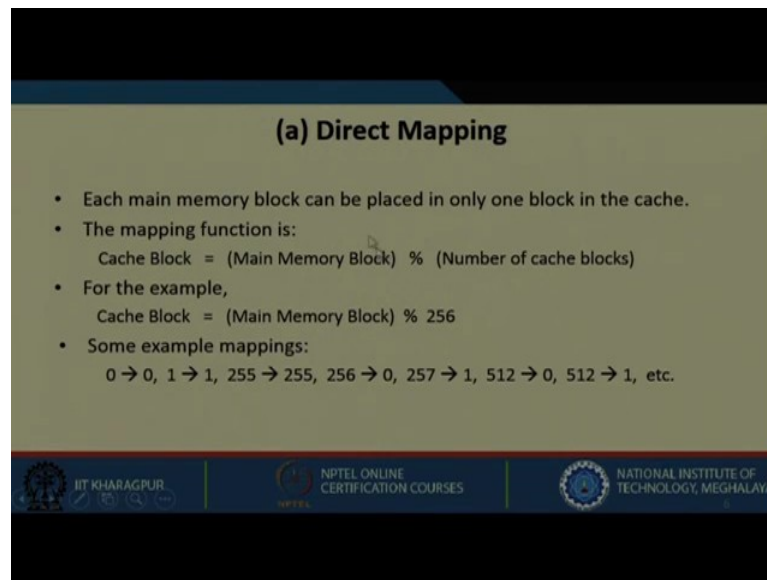
So, the main memory is addressable by 24 bit address; it is having 16 M words. So, 16 M words; that means, the total number of blocks in main memory will be total 16 M divided by 32. So, we have 512 K blocks in main memory. So, what is important here is to know how many blocks in main memory is there and how many blocks in cache memory is there. So, we have a total of 512 K blocks in main memory with this organization, and we have a total of 256 blocks in cache memory.

(Refer Slide Time: 08:20)

Now let us see direct mapping. What happens in direct mapping? The name suggests it is very direct, so let us see how it is. Each main memory block can be placed in only one block in the cache depending on this particular function. What is that function? Which main memory block will be placed in which cache block is determined by this particular formula. We get a main memory block; we make a modulus with number of cache blocks. So, we already know total number of cache blocks is 256, and any main memory block modulo this will give me the particular cache block.

So any main memory block can be placed in some particular cache memory block using this particular formula. What is that formula? Main memory block % total number of cache blocks. So, with the example if we take this particular formula into consideration, 0 will be mapped to 0 block of cache; how? 0 modulo 256 will be 0. So, 1 modulo 256 will be 1. Similarly 255 modulo 256 will be 255.

Similarly 256 modulo 256 is 0; so it will be again placed in block 0, 257 modulo 256 will be 1 so this will be placed in 1. So, the idea is now you see that block 0 of main memory block 256 of main memory both will be placed in block 0 of cache memory. So, there are many blocks of main memory that can be mapped into the same block of cache memory. This is direct mapping, where we have given a formula; through that formula you are mapping any block of the main memory into some block of the cache memory.

So at a time we cannot have block 0 and block 256 at the same time in the cache. This is a problem. If in a program you require both block 0 as well as block 256; then this can be a problem.

(Refer Slide Time: 12:44)
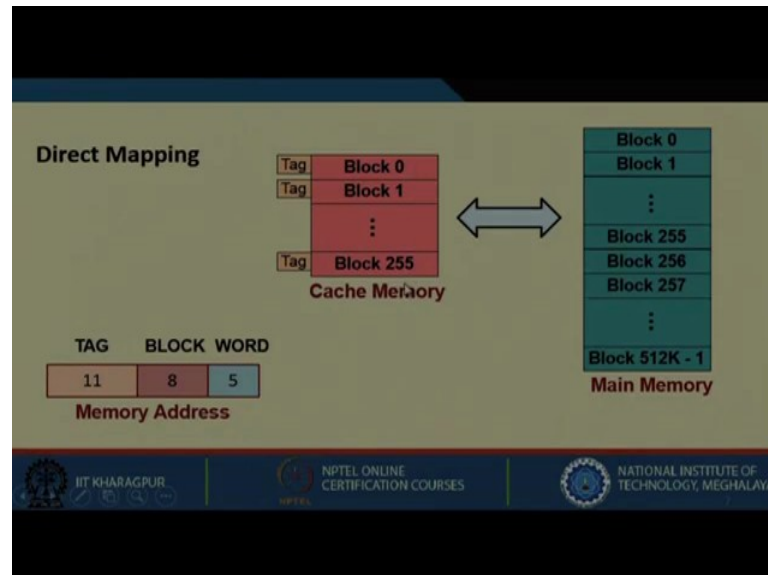
Now in direct mapping the memory address is divided into 3 parts basically we call it TAG, BLOCK and WORD. Now each block is having 32 words. So, how many bits will be required to access a word within that block? We will be requiring 5 bits, because we have 32 words. So, 5 bits will give you any one of the word within a block.

So 5 least significant bits will be required to access a WORD within this block. Now we need to know how many blocks are there. The total number of blocks here is 256. Now if total number of blocks is 256 then we require 8 bits to represent a block. So, this BLOCK will have 8 bits.

(Refer Slide Time: 14:31)
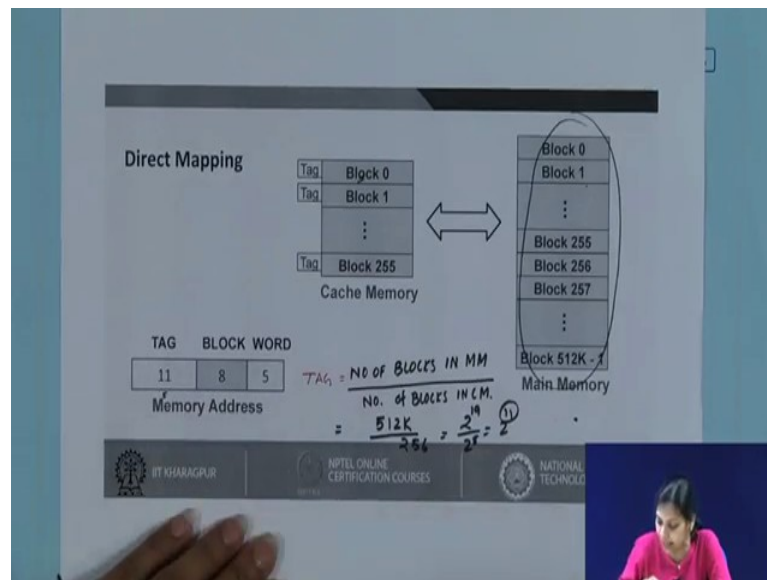
Now, finally, is the TAG, which will tell basically that which block of main memory is mapped to a particular block of cache memory.

So let us see here what happens is that as we can see that there are many blocks in main memory and there are few blocks in cache memory how we can find the number of bits in the TAG.
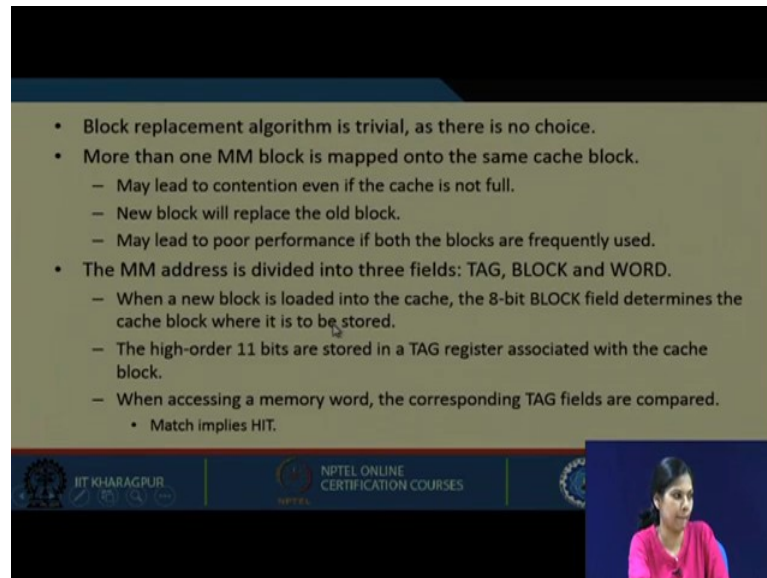
(Refer Slide Time: 15:16)

The TAG field for direct mapping will be number of blocks in main memory divided by number of blocks in cache memory. Here it will require 11 bits.

So, first we match the particular TAG; if that is present then we get that word and then it is transferred to processor. If it is a miss then, first it is brought from main memory to this cache memory, and then it is transferred to the processor.

(Refer Slide Time: 17:10)

So the block replacement algorithm is trivial as there is no choice. But more than one main memory block are mapped onto the same cache block. This may lead to contention even if the cache is not full; that means, even if there is space in the cache, because of that mapping formula restriction we cannot place any block of the main memory anywhere.
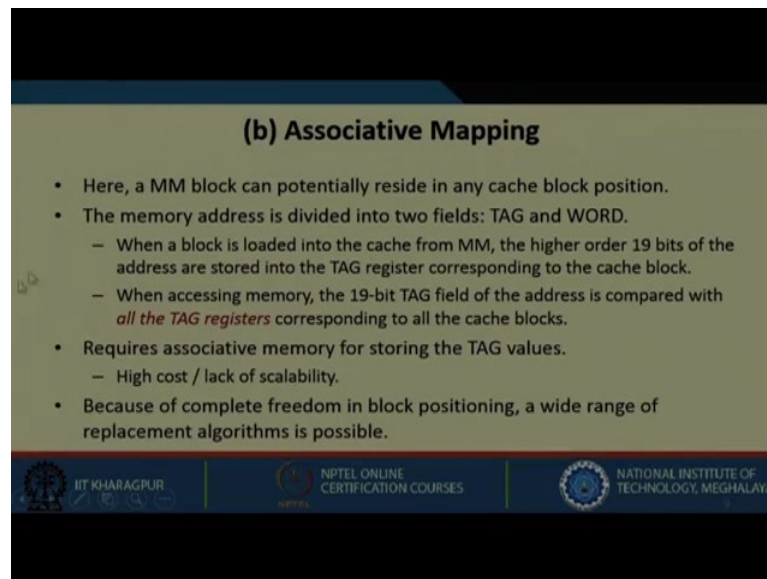
So, we cannot have any other option new block will replace the old block. So, old block has to be replaced when a new block is brought in; may lead to poor performance if both blocks are frequently used. We are trying to say that in a program we require both block 0 and block 256 simultaneously in a loop. Block 0 and block 256 both cannot stay at the same time.. So, when one is staying then the other has to be removed and then again the other has to be brought in the other has to be removed.

So if the blocks both the blocks are required frequently then this kind of mapping will give poor performance. The main memory address is divided into 3 fields as we already have seen, TAG, BLOCK and WORD. So, when a new block is loaded into cache, the 8-bit BLOCK field determines the cache block, where it is to be stored with that formula, the high order 11 bits are stored in the TAG register associated with the cache block. So,

when accessing a memory word, the corresponding TAG fields are compared; match implies a hit.

So basically when we say that whether the word is found in cache or not we actually match that TAG. So, if the TAG matches then we say that that particular word is present.

(Refer Slide Time: 19:38)

Next, let us move on with associative mapping. Here a main memory block can potentially reside in any cache block position. In case of direct mapping, we have seen the problem, even if there is space we cannot keep a block. We cannot bring any block from main memory because of that condition. So, here in this associative mapping this condition is relaxed.

Here any block of main memory can be brought into any block of the cache. This of course, will make the utilization much more. The memory address is divided into 2 fields only, we have TAG and WORD, because there is no concept of block. Any block can be brought in here; when a block is loaded into the cache from main memory the higher order 19 bits of the address are stored into the TAG register corresponding to the cache block.

When accessing memory, the 19 bit TAG field of the address is compared with all the TAG registers corresponding to all the cache blocks. So, this is an disadvantage again.

When the processor checks the TAG field, it has to check all the TAG registers whether that particular tag is present corresponding to any of the blocks.

So there we have 256 blocks. So, TAG associated with 256 blocks must be checked to know whether it is a hit or not. So, when accessing memory the 19 bit TAG field of the address is compared with all the TAG registers corresponding to all the cache blocks. This requires associative memory for storing the TAG values. Associative memory is much more costlier; results in higher cost and lack of scalability. We cannot have very large associative memory in place. Because of complete freedom in block positioning a wide range of replacement algorithm is possible.
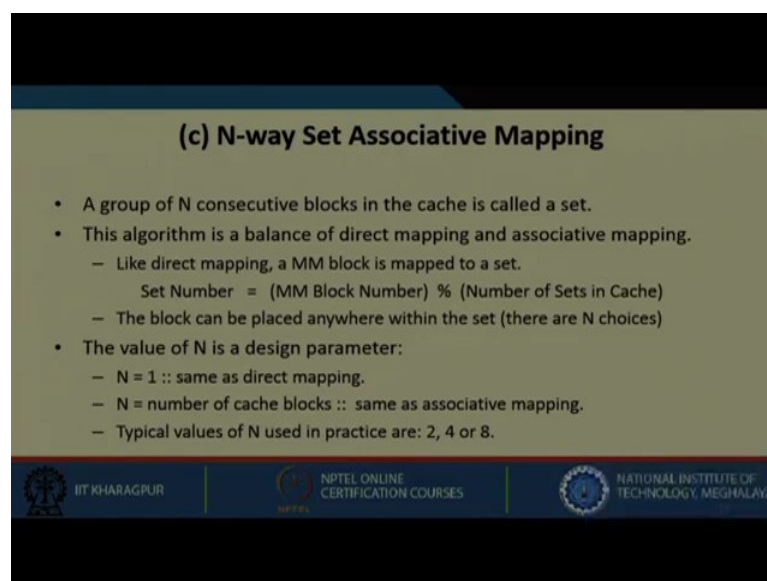
So this particular associative mapping we can clearly see is much more efficient because the space will be utilized to the maximum; any main memory block can be kept in any cache block. So, the entire space of the cache is utilized very nicely; we have to replace a particular block when the cache is full, if the cache is having any empty space then any block can be brought in, but with that what we are adding up is the checking. We need to have an associative memory for storing the TAG values.

(Refer Slide Time: 23:47)
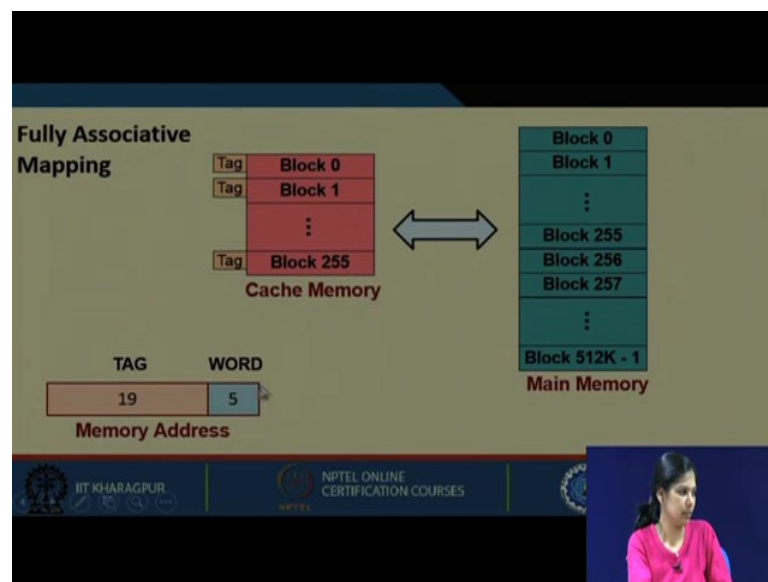
So in fully associative mapping we have total of 256 blocks in cache memory. Any 256 blocks from main memory can be brought in here, and which block of main memory is here is determined by this 19 bit TAG field.

(Refer Slide Time: 24:25)

Let us now come to another mapping algorithm that is called N-way set associative mapping. We have seen that in direct mapping there is some advantage and disadvantage, and in associative mapping of course, we have advantage and as well as some disadvantage. So, what we are trying to do let us take some of the properties of direct mapping and some of the properties of associative mapping. We combine these two and we have set associative mapping. What this algorithm says is a group of N consecutive blocks in cache is called a set.

So words are combined to form a block, and blocks are combined to form a set. This algorithm is a balance of both direct and associative mapping. Like in direct mapping, a main memory block is mapped to a particular set. So, the set number of the cache can be determined by main memory block number modulo number of sets in the cache; that means, in direct mapping we have main memory block number modulo number of blocks in cache, but here we have number of sets in cache. So, the block can be placed anywhere within the set that is there are N choices for it.

The value of N is a design parameter. So, when N=1, it is same as direct mapping, and when N is the total number of cache block then it is same as associative mapping. But the typical value of N that is used in practice can be 2, 4, 8, or 16.

(Refer Slide Time: 27:28)
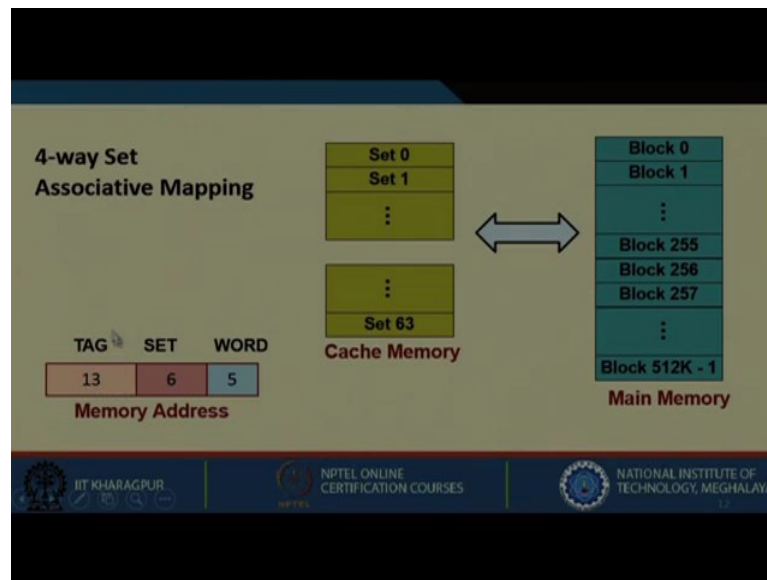
Let us consider a 4 way set associative mapping where we have 64 sets and we have 512K memory blocks.

(Refer Slide Time: 28:40)
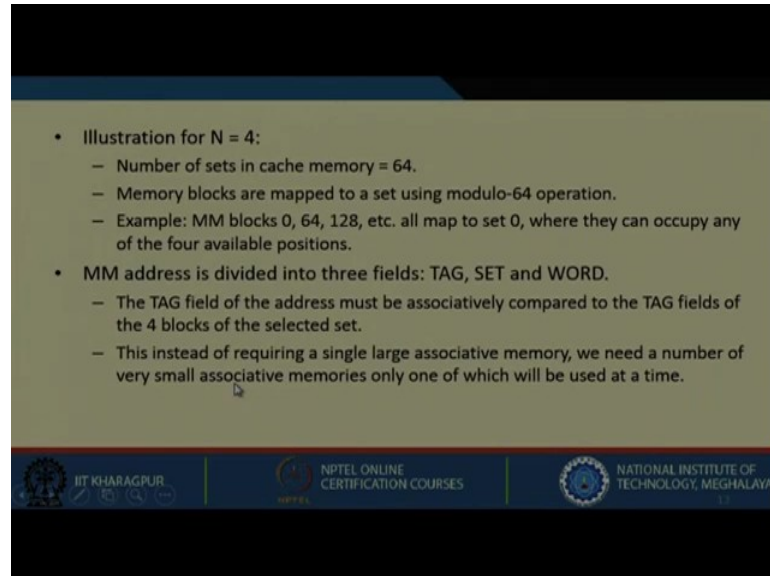
We will determine the TAG here in the same way for direct mapping; instead of total number of blocks here we will have total number of sets in cache memory. So, in the same way like direct mapping here it was having total number of blocks we are having total number of sets here. So, let us say total number of blocks is 512 K and total number

of set is 64. So, it is 2 to the power 19 divided by 2 to the power 6 coming to 2 to the power 13.

(Refer Slide Time: 29:48)
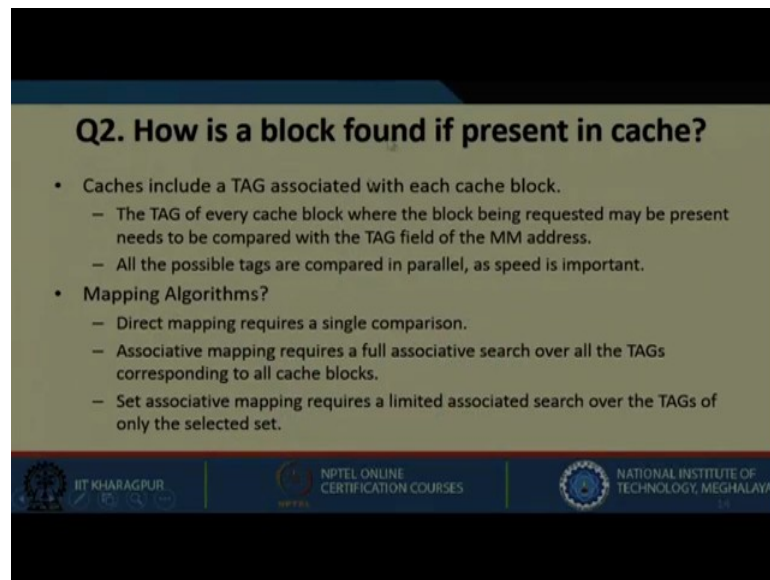
So if you see we have 13 bit for TAG, SET is 6 and WORD is 5. So, let us illustrate for N equals to 4. For 4-way associative set associative mapping, number of sets in cache memory is 64. Memory blocks are mapped to a set using modulo 64 operation. So, main memory blocks 0, 64, 128, etc. all map to set 0 where they can occupy any of the 4 available positions.

This instead of requiring a single large associative memory we need a number of very small associative memories only one of which will be used at a time.

(Refer Slide Time: 32:00)
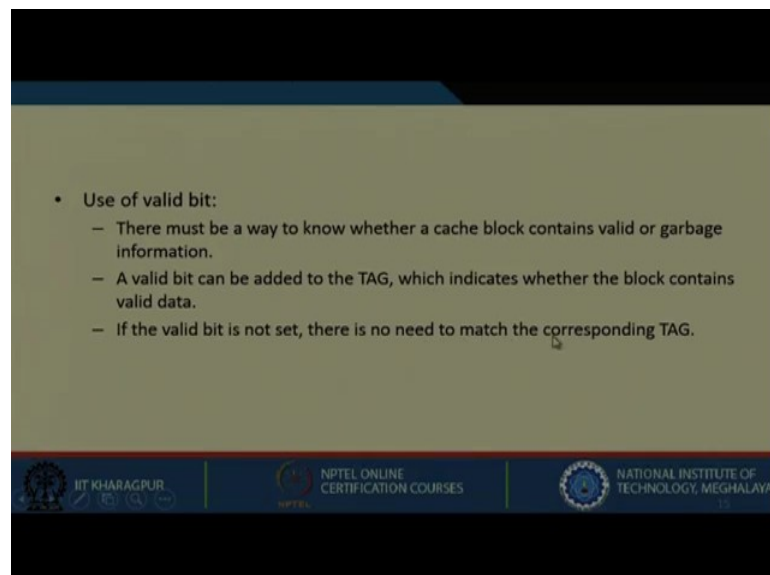
So cache include a TAG associated with each cache block. The TAG of every cache block needs to be compared with the TAG field of the main memory address. So, the TAG field of the main memory address is compared with the TAG of every cache block; all the possible tags are compared in parallel as speed is very important.
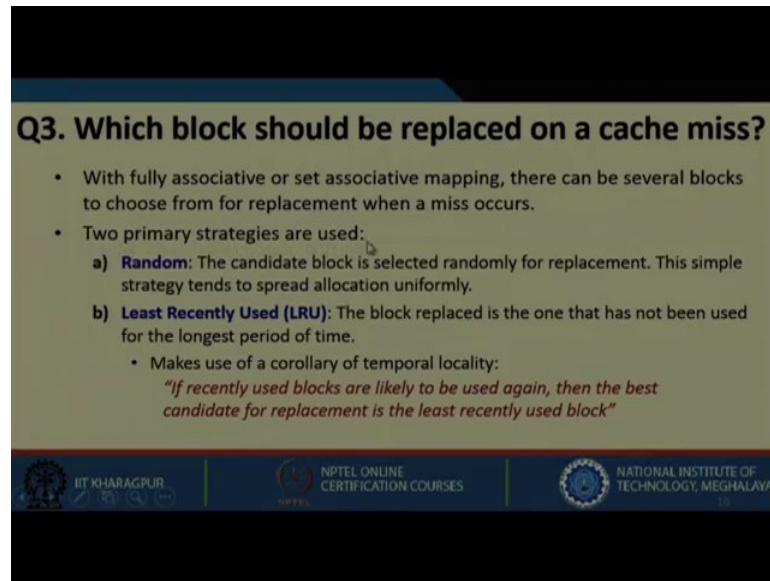
(Refer Slide Time: 33:18)

(Refer Slide Time: 34:41)

The next question arises which block should be replaced on a cache miss. With fully associative or set associative mapping, there can be several blocks to choose for the replacement when a miss occurs; that means, within an associative or if you think of set associative where you have some blocks.

Now for that two primary strategies are used, one is random. The candidate block is selected randomly for replacement. This simple strategy tends to spread allocation uniformly. Another is least recently used. Here what it says that the block replaced is the one that has not been used for longest period of time.

Say we have 4 blocks associated with a particular set, and we want to replace a particular block. This least recently used says that a block which is in the cache for longest period of the time, but it has not been used; we will replace that particular block.
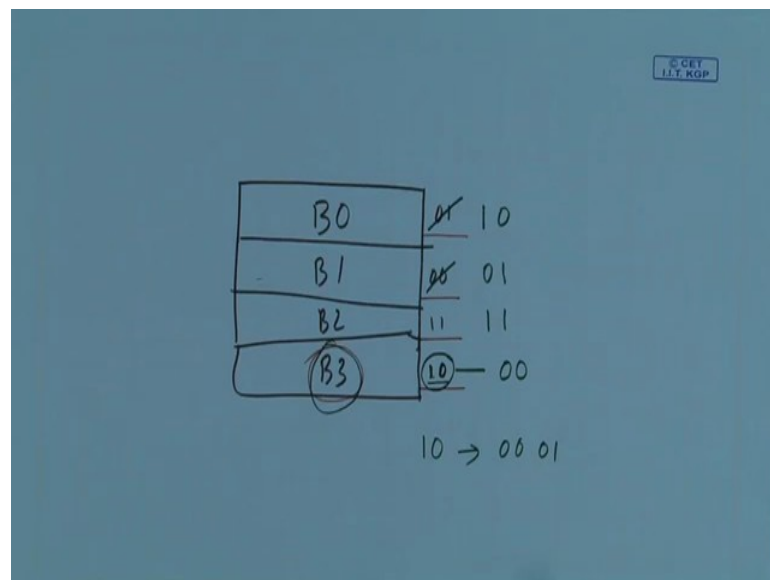
To implement LRU algorithm the cache controller must track the LRU block as the computation proceeds. We need to keep track of the LRU block. A 2-bit counter can be used with every block for this purpose, as explained.

So, let us say we have we have 4 blocks this is block 0, block 1, block 2, and block 3; and the values are like this; this is 0 1, this is 0 0, this is 1 1, and this is 1 0.

What the algorithm say the counter of the reference block is reset to 0. The counters with values originally lower than the referenced one are incremented by 1, and others remain

unchanged. So, this is the current value of the counter. So, the current value of the counter is 0 1, this is 0 0, this is 1 1, and this is 1 0; now let us see I am referencing B3 block again. So, the counter associated with this will become 0 0 and all the counter value less than this will be incremented by 1 and all other counter values will remain unchanged. This process continues.

(Refer Slide Time: 42:04)

(Refer Slide Time: 44:34)

Now, we will move on with an example with 4 blocks. Initially, nothing is there. Now say block 2 is referenced. So, the counter value associated it is a miss. Initially it will be

a miss because this is all empty. Now in the block 2 a value is brought in and the reference block value become 0, next block 0 is brought in the value associated with the new block become 0 and the previous block is incremented by 1; it has become 1, next block 3 this is also miss.

So the counter value associated with this will become 0, and all others will get incremented by 1; similarly block 1 is accessed and these are the values associated with this will become 0 and all others will get incremented by 1. 1 has become 2, 2 has become 3, and this continues.

 (Refer Slide Time: 48:35)
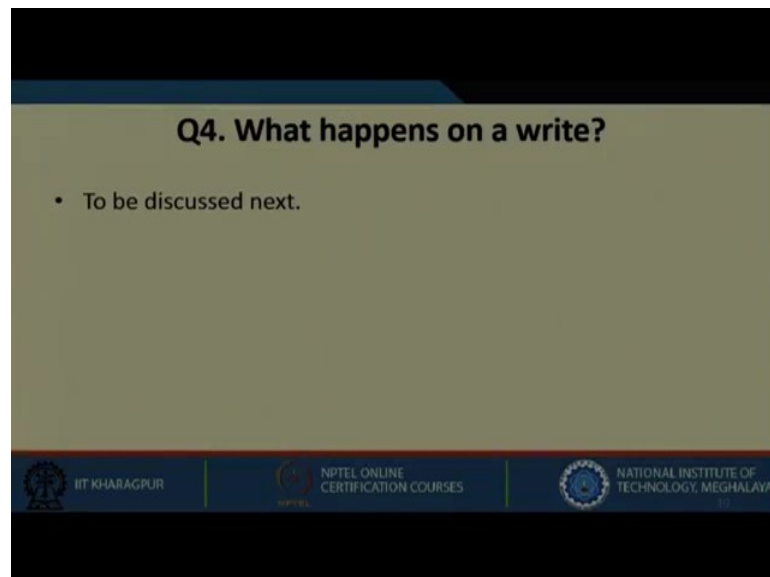
So, the next question is what happens on a right will be discussed next. We have come to the end of lecture 30 where we have discussed about the various mapping techniques and the block replacement strategies.

Thank you.