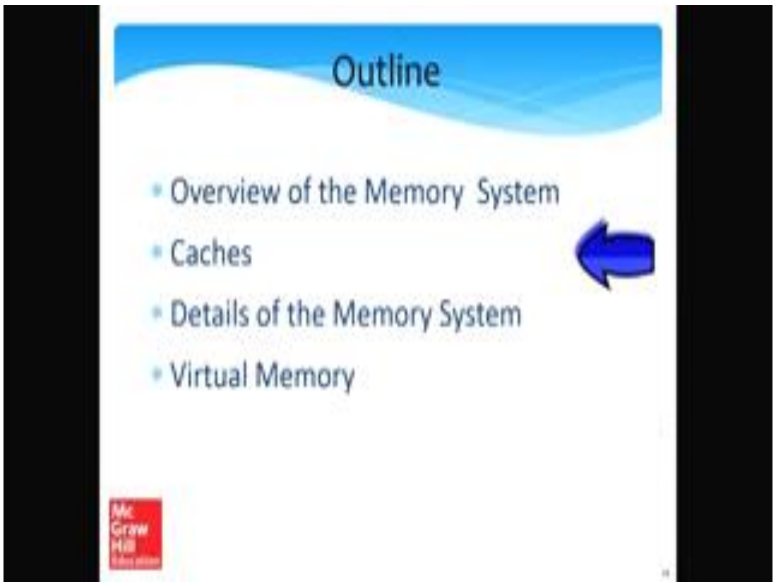
Computer Architecture

Prof. Smruti Ranjan Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 31
The Memory Systems Part-II

(Refer Slide Time: 00:26)

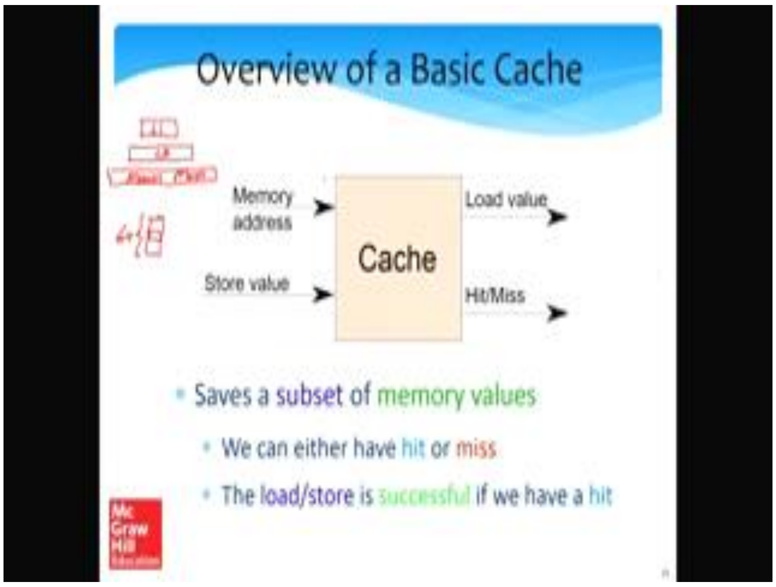


Outline

- Overview of the Memory System
- Caches
- Details of the Memory System
- Virtual Memory

Let us start our discussion of caches. So, caches are very interesting and there is a huge amount of research on how to make caches efficient. So, we will not be able to cover most of it in this lecture, but at least we will be able to cover most of the basic mechanisms that underlie today's modern caches.

(Refer Slide Time: 00:50)



Overview of a Basic Cache

- Saves a subset of memory values
  - We can either have hit or miss
  - The load/store is successful if we have a hit

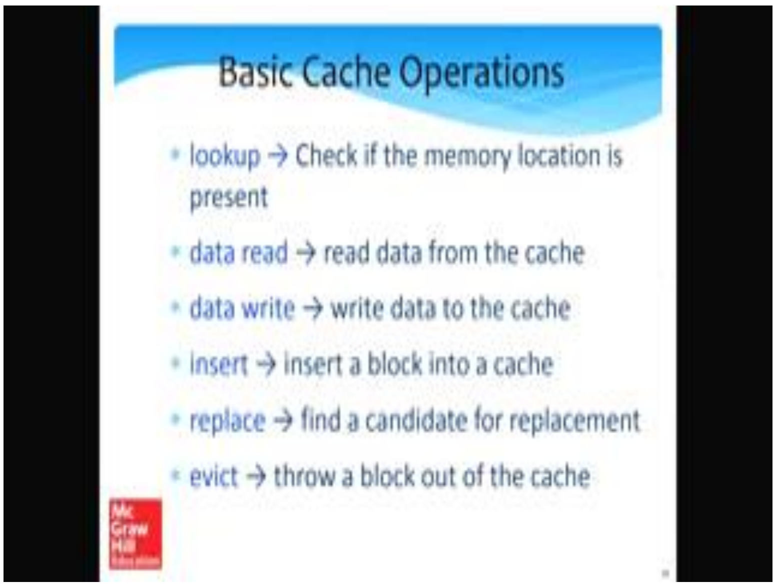
So, the overview of a basic cache is as follows, it is a black box, let us first treat it as a black box as a memory that contains a set of memory addresses and their values arranged into blocks of course. So, for blocks you need to go back to the previous lecture. So, what we had discussed in a previous lecture is that the memory system is organized hierarchically. So, we have an L 1 cache which is small and fasts then we have an L 2 cache which is slightly larger and slower.

And finally, we can have an L 3 or we can have a main memory which is made of dram cells right which is which contains all the addresses and. So, these addresses in L 1 or 6 stripped subset of the addresses in L 2; and the addresses in L 2 are a stripped subset of the addresses in main memory. So, we had discussed that and also take advantage of special locality right for because having cache hierarchy takes advantage of temporary locality. So, to take advantage of special locality, we are discussing grouping memory addresses into blocks.

So, if we group them into blocks of 64 and fetch a block in one go from a lower level, where a block is an atomic unit. So, we never add half a block or third of a block. So, if we fetch a block in one go, so most likely for accessing you know one data item within a block, most likely we will be accessing in the near future, some other data item which is also within the block. Since that has already been fetched, we do not have to fetch it once again, so this will increase our cache hit rate. So, any cache can be looked at like this that we have a memory address that goes in, say if it is a load then a load value will come out, and also hit on miss decision; of course, if it is a miss load value does not make any sense.

So, for a load basically we have an address, and a value for a store. Same way, if the block is not there we will have a miss where if the addresses of the block are there we can perform a store, so the store will only have a memory address and store value. So, for a store a load value makes no sense. So, what are the basic problems we need to make a hit or miss decision first, if there is a cache hit, we need to perform the load or store.

(Refer Slide Time: 03:31)



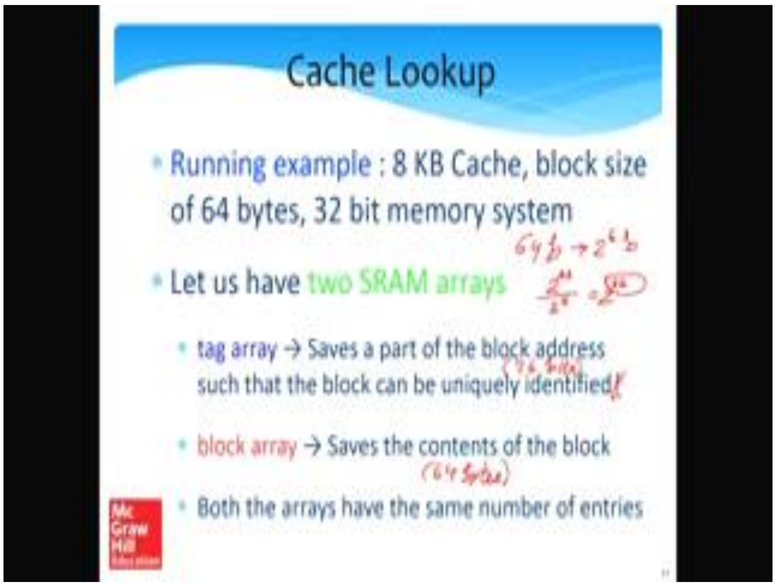
Basic Cache Operations

- lookup → Check if the memory location is present
- data read → read data from the cache
- data write → write data to the cache
- insert → insert a block into a cache
- replace → find a candidate for replacement
- evict → throw a block out of the cache

So, let us look at the basic cache operations and describe the operation of an entire cache on the basis of some of its primitive operations. So, the first operation let me move to a laser pointer. So, the first operation is lookup, if we check if a given memory location a given memory address is present inside the cache that is the first thing that we check. If it is not present inside the cache well we declare a miss and we just move out, but if it is present inside the cache, then we can do several things. If it is a load we can read data from the cache as a data read operation; if it is a write we can do a data write operation, we can write data to the cache it will be done.

Now, assume that there was a cache miss and then the block comes to the lower level then we need to insert the block inside the cache and to insert a block sometimes it might be necessary that we might find that the cache is full. So, then we have to throw out one block. So, we need to first find a candidate for replacement then we need to evict it through a block out of the cache and insert the new block in its place. So, these are six basic operations lookup, data read, data write if there is a miss we need to do you know insert, replace, evict these three operations need to be done. So, we shall look at each of these operations in great detail now.

(Refer Slide Time: 05:15)



Cache Lookup

- Running example : 8 KB Cache, block size of 64 bytes, 32 bit memory system
- Let us have two SRAM arrays
  - tag array → Saves a part of the block address such that the block can be uniquely identified
  - block array → Saves the contents of the block (64 bytes)
  - Both the arrays have the same number of entries

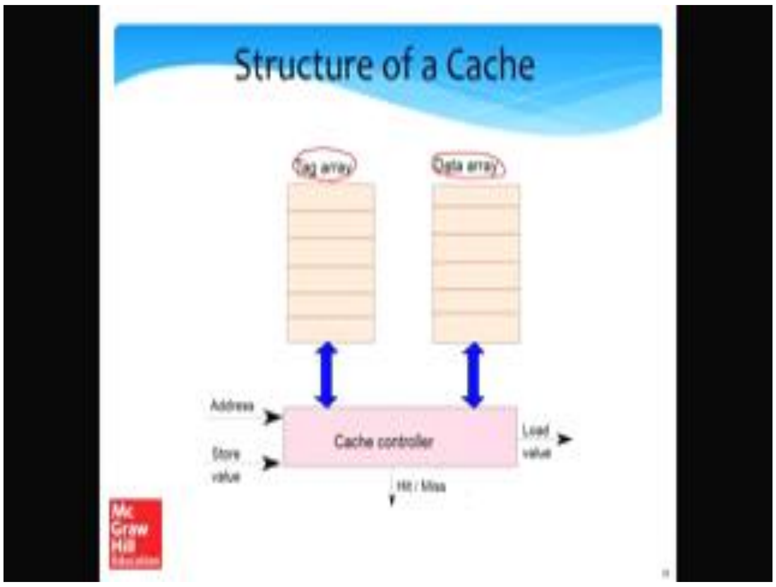
$64b \rightarrow 2^6 b$

So, let us first define a running example. So, we will pretty much devote at least this entire sub section. This entire sub section will be using this running example because having some example sort of streamlines are thinking and instead of using symbol; it is better to use a concrete example. So, let us try to design a 8 kilobyte cache with a block size of 64 bytes, and a 32-bit memory system. So, what again is the configuration it is a 8 kilobyte cache, 8 kb block size of 64 bytes in a 32-bit memory system. Furthermore, inside the cache, let us have two SRAM arrays - two arrays of SRAM cells, let us have a tag array it saves a part of the block address such that we can uniquely identify a block. So, a tag array this contains a part of the address.

So, let us look at let us may be you know discus block address slightly more before moving to the block array. So, block contains 64 bytes. So, 64 bytes is 2 raised to the power of 6 bytes. Since, a block is an atomic unit, so we will never in a partition a block. So, in the 32-bit memory system, how many possible bits blocks can we have will be 2 to the power of 32 that is the total size of the memory space divided by 2 to the power of six which is the block size. So, maximum we can have 2 raise to the power of 26 blocks. So, a block address is 26 bits, a block address because you know 6 bits are taken away by the address of the byte inside of a block. So, the maximum number of blocks, we can have is 2 to the power of 26. So, block address at the most is 26 bits which means that we need 26 bits to uniquely identify a block.

So, let me go to the block address and just add 26 bits here. So, mind you if I change these parameters then these parameters will also change, but let us at the moment discuss everything in the context for running example. So, then we have along with the tag array, we have a block array which saves the contents of the block. So, the contents of the block how large is it, the contents are 64 bytes; here is one block we have defined to be 64 bytes it saves the contents of the block. So, both the arrays the tag array and the block array need to have the same number of entries. In fact, they can have a one-to-one correspondence right. So, they absolutely need to have the same number of entries.

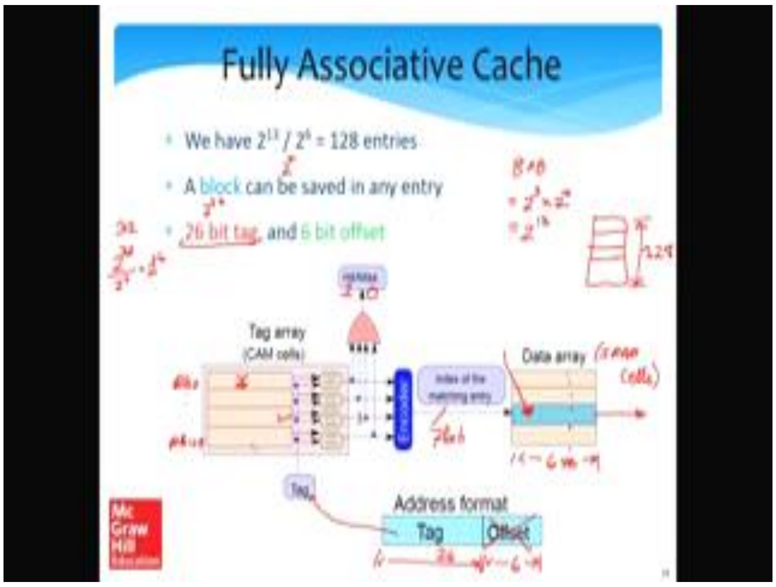
(Refer Slide Time: 08:29)



Structure of a Cache

So, the structure of a cache is now as follows. We have defined two arrays a tag array and block array. Of course, they are not drawn to scale and they have the same size the same number of entries; both of them exchange data with a cache controller which is a piece of logic that controls the cache. So, the cache controller gets the address in the case of a store the stored value, it interacts with both the arrays. It has two outputs one is the hit miss decision and other is a value of a load, if the operation is a load operation.

(Refer Slide Time: 09:06)



Fully Associative Cache

- We have $2^{13} / 2^{6} = 128$ entries
- A block can be saved in any entry
- 26 bit tag, and 6 bit offset

So, let us discuss one of the simplest kinds of caches called a fully associative cache. So, let us do a little bit of math. So, our cache size is 8 kilobytes. So, let us try to find out how much this is in bytes, this is 2 raised to the power 3, which is 8 multiplied with 2 raised to the power 10 which is 1 kilo. So, 1 kilo means 1024 which is 2 raised to the power of 10. So, this 2 to the power of 3 times 2 to the power of 10 is 2 to the power of 13. So, this is the number of bytes that we have in our 8 kilobyte cache 2 to the power of 13. And since one block is 2 to the power of 64 bytes; so what is the maximum number of blocks that we can have in the cache 2 to the power of 13 divided by 2 to the power of 6 which is 2 to the power 7, so 128. So, we can have 128 blocks of the cache fill up 128 entries.

So, in this case in a fully associative version of our cache, we will have 128 entries. So, 128 entries in the block array 128 entries in the tag array with a one-to-one correspondence. And in a fully associative cache, if we have 128 entries given a block, it can be stored in any entry all right it is a given a block, it can be stored anywhere inside the cache in any of the 128 entries. So, recall what we have done in the previous slide we have said that we can have 2 to the power 26 possible blocks, the reason we said that is basically because we have a 32-bit memory system and a block size is 64 bytes. So, 2 to the power 32 divided by 2 to the power of 6 is 2 to the power of 26. So, our block address is 26-bits. And then since a block is 64 bytes which is 2 to the power of 6, this is 6 bit offset of a byte inside of a block.

So, let us do one thing since our block address is 26-bits let us save the entire 26 bits in the tag array. So, we will require it, but let us try to understand this gradually. So, in the tag array let us save 26-bits, which is the block address all right. So, you know we will have different sets of 26-bits for different entries; and in the data array we will have the contents of the block which is essentially 64 bytes, capital B for byte.

So, what we see over here the way that we do the addressing is that given memory address, we first complete the tag part of the memory address which is the upper 26-bits, and then there will be a byte offset part which is the 6 bits. So, where does 6 again come from it comes from the fact that the 2 to the power of 6 is 64, and 64 bytes is that block size. So, these lower 6-bits we will ignore, we will take the upper 26-bits which is this tag over here, and will compare them with each of the entries of the tag array right. So, with every single entry of the tag array, we will compare 26-bits, you know more significant 26 bits of the address to find out if there is a match.

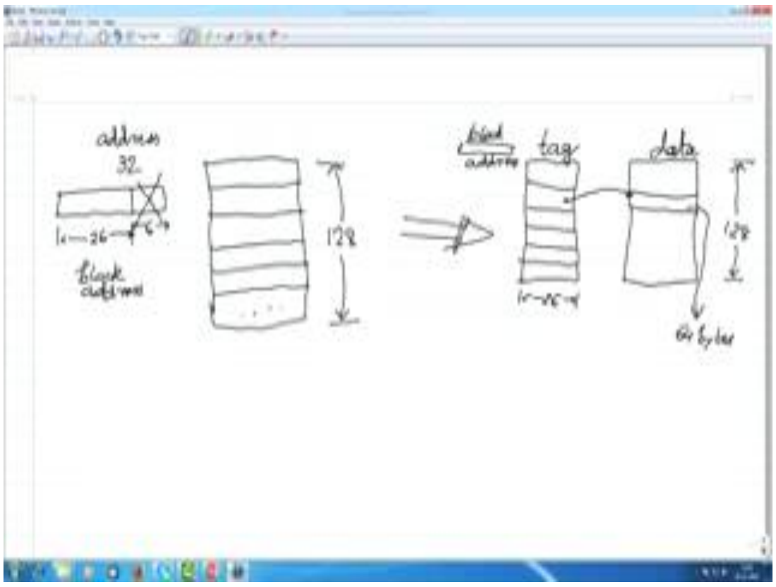
So, because this is the block address inside the address, so we will compare each of these 26-bits, with the 26-bits stored in the tag array. So, mind you what are these 26 bits they are the addresses of different blocks. So, address of block 0, you write the block address of block 127. Now, from we have been given a new block address we need to check with each one of them. So, for this the best structure that we can use is a CAM recall from chapter six that a CAM is a content addressable memory where instead of accessing the memory via its row address, we access it via its contents. And this is an ideal application of a cam where given the block address in the address given the block part of the address this is also called the tag part. We compare the 26-bits with each of the 26-bits stored here and. So, this is done in a typical cam array.

And after each comparison we send them to an OR gate. So, if let us say, so since there is no duplication inside a cache, if there is a hit, it will hit in only one entry. So, let us say it hits in this entry. So, this will be 1. So, the output of the OR gate is one. So, if this output of this OR gate is 1 it means that there has been a hit, otherwise if the block address does not match any of these entries any of these tag entries then it means that there is no hit and there is a miss. So, basically this is a hit the OR gate will give a one which means there has been some match here, otherwise the OR gate will yield a 0 which means there has been no match after that you know if there is a hit only one of the lines will be one to get the address of that line. We will use an encoder which will tell us this case what is the address of that line and it will be a 7 bit address because there are 128 bits here. So, this will be a 7 bit address right for which line actually hit.

And the data array will be a regular array of SRAM cells. You know a simple regular array of SRAM cells nothing very sophisticated about it. So, let us say there is a hit in. So, this is 00 01 and 1 0. So, basically it is a second entry in which we have had a hit. So, we will go to the second entry of the data array, read the data and that will be the output. So, if it is a load the output is the 64 bytes, or any subset of this, if it is stored then we will essentially write to this entry. So, in any case we are doing lookup for aim of the lookup stage is to indicate in which entry of the data array does the memory locations value exist?

So, let me just quickly summarize what is a fully associative cache. A fully associative cache the block can reside within any entry of the cache. So, we did some amount of math and from that we figured out that we will have 128 entries in our cache. So, in this case the block can reside in any one of the entries. Now the block is residing in any one of the entries how do we know which one? So, our look for a lookup operation; so let me I think let me cleanup small part of it and such that I actually ended up erasing all of the text on the slide, so that is ok. Let us just abstract the problem. So, let me may be move out of power point.

(Refer Slide Time: 17:15)



So, the problem that we have at hand is that we have a cache with one 128 entries. So, a block comes and address comes. An address is a 32-bit quantity and we need to map I can may be change the fine with the color. So, what we need to do is that we need to find out this address where exactly this address is contained within the cache. So, what we do is that we consider the 32-bits, and we try to break the address into two parts; one part is the offset of the byte within the block and the other is a block address. So, since a block is 64 bytes and 64 is 2 raised to the power 6 this means for the first six bits represent the address of the byte within the block. So, this is something that we can ignore or discard the remaining part which is the next 26-bits is what constitutes the block address. So, this part is the block addresses the 26-bits.

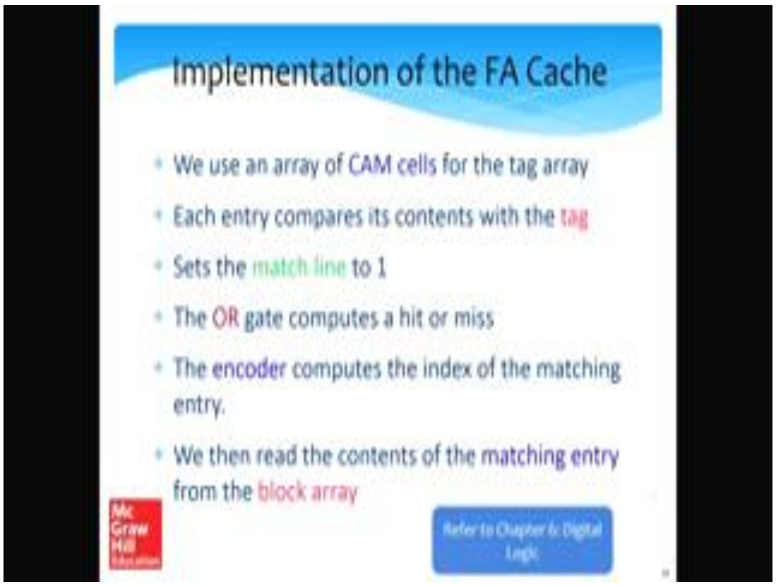
So, what we need to do is that we need to find out which entry out of these 128 corresponds to the block address which is represented by these 26-bits. So, this cache can further be divided into two parts one is I think it is fine, so sorry medium, so one is the tag array and the other is the data array. So, basically the tag array contains a portion of the address. So, I said we can uniquely map the block. So, in this case, since the block can be present in any entry the tag need to contain the block address for the entry that it corresponds to. So, recall that both the tag array as well as the data array, we will have 128 entries right both of them we will have 128 entries. So, and there is a one to one correspondence between a tag entry and a data entry.

So, what we need to do is that we need to compare each tag with the block address which is 26-bits. So, the tag also needs to be 26-bits, because it also contents need to content the address of the block whose contents are in the data array. So, we compare the block address that we get the block address that we get with each and every entry of the tag array. If there is a match we figure out for which entry there is a match assume that there is a match for this entry then we access the corresponding entry in the data array. And we return if it is a load we return the contents of the data array which is 64 bytes out of which we can choose a subset or if it is a store then we write to the relevant potion of the block inside the data array.

So, the basic problem here is essentially search problem that given 128 entries in the tag array we need to find if any of the entries in the tag array matches the value of the block address; if it does we are done. The way we do that is basically via the CAM array over here. So, recall that a cam is content addressable memory. So, the block address which is essentially the tag part of the address this thing is compared with each and every entry stored in a tag array. So, we either if we do not find an entry it is a miss, if we find an entry. So, we will never have duplicates then we access that entry of the data array. So, this is called a fully associative cache.

Well where does a name come from the name comes from the fact that a given block can be associated with any entry in the data or tag arrays which means it is completely your fully associative, so that is why the name comes from right can be associated. So, associative and in particular is also called a 128 way associative in this case because there are 12 entries in any entry we can map. So, we will any way discuss about ways later, but the basic idea always in the look up stage of any cache is the search. Once we do the search we can progress to the next stages.

(Refer Slide Time: 23:17)



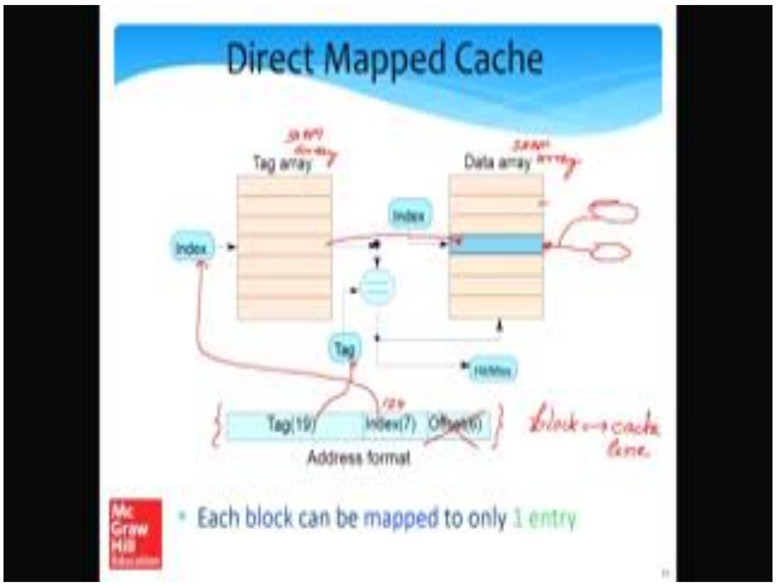
Implementation of the FA Cache

- We use an array of CAM cells for the tag array
- Each entry compares its contents with the tag
- Sets the match line to 1
- The OR gate computes a hit or miss
- The encoder computes the index of the matching entry.
- We then read the contents of the matching entry from the block array

Refer to Chapter 6: Digital Logic

So, basically this slide essentially discusses much of what we have talked in the previous slide that in the can it summaries what the CAM array the tag array does. And the for any doubts on how a content addressable memory works, readers can go back and refer to chapter 6 - the chapter on digital logic right where we discuss registers memories in logic, so that chapters.

(Refer Slide Time: 23:46)



Direct Mapped Cache

- Each block can be mapped to only 1 entry

Now, let us discuss a very different kind of cache. So, what was the problem with cam arrays that we talked in chapter 6? So, one problem was that cam arrays instead of a 6 transited cell they have a 10 transited cell, so they are larger as a result they are slower. And also we need they are not fare efficient at all because we need to compare the block address with every single entry of the tag array which is not fare efficient by any means. So, let us look at a different part of the spectrum. So, let us have a cache again with a tag array and the data array, but let us divide the address in a different manner. So, if you consider the address the lower six bits will always remain the offset within the blocks this is something that we can happily ignored, but for the upper 26, let us divide them into two feats a 19 bit tag and a 7 bit index.

So, 7-bit means it can specify it can uniquely address 128 entries. So, let us use this index to map to only one element of the tag array. So, direct mapped cache means that given block can reside an only one entry of the direct map cache unlike a fully associative cache where it could reside anywhere. This can reside in only one entry. And how is that entry decided we basically remove the 6, you know remove the block part, the block address part which is 6 bits in this case because it is a 64 bite block. We take the remaining 7 entries why seven, because our tag array has 128 entries, and 128 is 2 to the power 7.

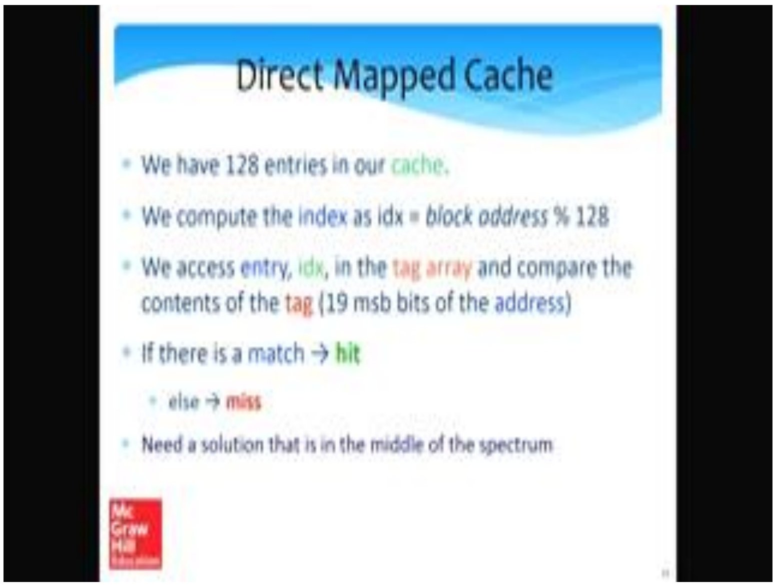
So, we take the remaining seven entries we compute an index with that index we access the tag array. And we read the tag that is stored and we compare that with the tag part of the address. If the comparison results in equality, well it is a hit; otherwise it is a miss, if it is a hit then with the same value of the index. There is a one to one mapping here again this we have located the block inside the data array that we need to either read or we need to write.

So, what is the advantage of a direct mapped cache? The advantage of a direct mapped cache as follows that is very simple unlike a fully associative cache, it is very simple. We do not need to use CAMS at all. So, the tag array can be an SRAM array. So, both the tag array and the data arrays can be an SRAM array right array of SRAM cells and both of them. Also in this case, we will have 128 8 entries is, but it is just that a given block cannot reside any where it has a fixed place. And the place is decided by the lower 7- bits least significant seven bits of the 26 bits that remain after the 26 bits of the block address (Refer Time: 27:12) block address.

So, this is a nice simple circuits SRAM are much faster than CAMS. So, it is a very, very simple circuit that can be used, but is just that it is also faster. So, a direct mapped cache is also faster than a fully associative cache. The only problem is that it will have lower hit rate and the reason being that assume that there are two lines say block is also called a cache line, say block is also refer to as a cache line. So, block well we will use both the terms interchangeably. So, assume that you know two cache lines mapped to the same entry. So, if they mapped to the same entry then what will happen they will just be displacing each other and that will lead to a lower hit rate or a higher miss rate. But at this in a fully associative cache, they would have found space within the cache and one could have may be resided here and another could have resided over here.

But in this case since multiple conflicting lines or cache blocks can reside in the same entry, we will have a more displacement from the cache as a result a lower hit rate.

(Refer Slide Time: 28:40)

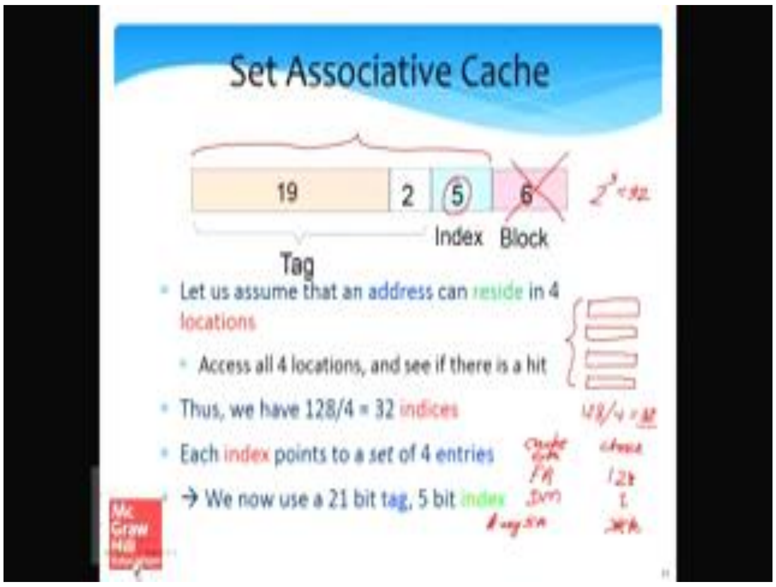


Direct Mapped Cache

- We have 128 entries in our cache.
- We compute the index as idx = *block address* % 128
- We access entry, idx, in the tag array and compare the contents of the tag (19 msb bits of the address)
- If there is a match → hit
  - else → miss
- Need a solution that is in the middle of the spectrum

So, let us now a look at something, which is in the middle. So, this slide I am not going through because we have just discussed this operation of direct mapped cache in a previous slide. So, without any further discussion, I will just move to the next slide.

(Refer Slide Time: 29:00)


Set Associative Cache

- Let us assume that an address can reside in 4 locations
  - Access all 4 locations, and see if there is a hit
- Thus, we have 128/4 = 32 indices
- Each index points to a set of 4 entries
- → We now use a 21 bit tag, 5 bit index

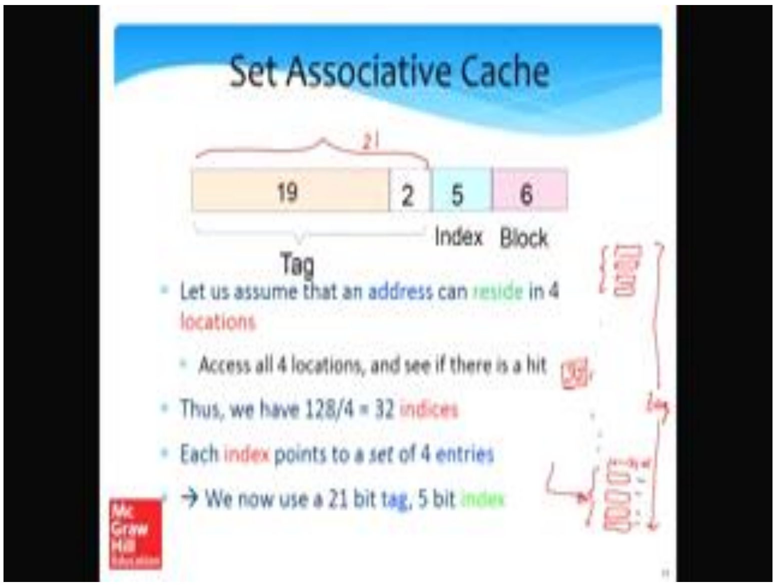
Which talks about an intermediate solution called a set associative cache? So, what was our assumptions fully associative cache? That a line can reside in any of the 128 entries for our running example of course; for a direct map cache, we said at line can only in reside in our particular specific entry well. So, in a set associative cache, what we will do is that we will make sets of let say 4 entries. So, let consider 4 entries. So, we will call them a sets right a set of 4 entries is the set. So, what we will do is that if we create such four entries sets how many sets will we have, we will have 128 divided by 4 - 32 sets. So, the logic is as follows.

So, that we first we will consider the block address part, the part that determines the addressing of the block at the lower six bits of course, we will discard. Then since we have 32 sets we will first take the lower five bits of what is remaining and 2 to the power 5 is 32. So, just so that we have a discussion in the right context we will use a lower 5 bits.

So, first index and access the right set. Once we access the right set, we will find four entries over there; inside the four entries we will compare the block address with each of these entries. And wherever there is a match we will if there is a match just among these four entries, then we will declare a hit, otherwise we will declare a miss. So, what is the basic idea, the basic idea is that this is solution which is in the middle of the spectrum between a fully associative cache and a direct map cache. So, in a fully associative cache or choice was ok let us talk about the cache type and the choice right.

So, in a fully associated cache where a choice of 128 entries a block could be there anywhere; in a direct map cache let us choice only one entry; in a set associative cache our choice in this case is 4, which means that the line can be anywhere in the set of 4 entries. So, we in general we can define what is called a k way set associative cache where k is a number of entries in each set. So, instead 4, it will become k. So, this particular example is a 4 ways set associative cache. So, if you want me to explain at in a different way I can do that as well. So, let me just clean up the ink on the slide. So, what we can do is we can take all the 128 entries.

(Refer Slide Time: 32:22)



Set Associative Cache

21
19
2
5
6
Index
Block
Tag

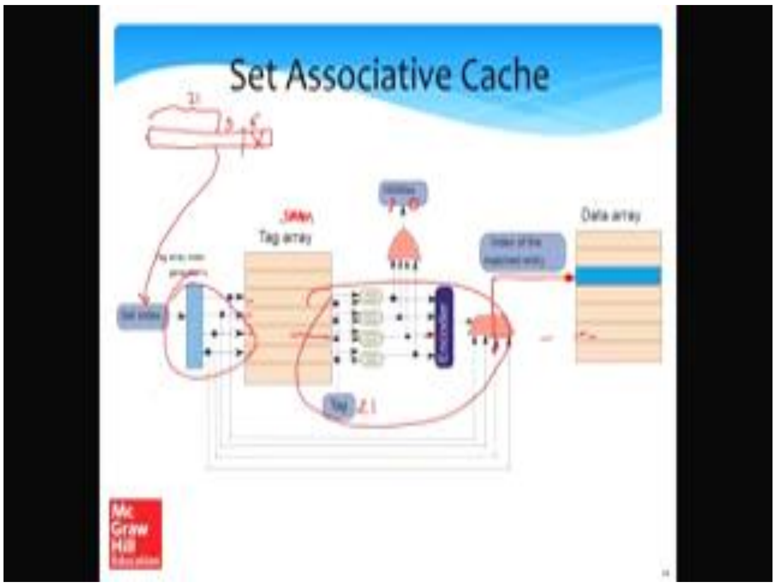
- Let us assume that an address can reside in 4 locations
  - Access all 4 locations, and see if there is a hit
- Thus, we have 128/4 = 32 indices
- Each index points to a set of 4 entries
- → We now use a 21 bit tag, 5 bit index

And we just have to group them into entries of 4 each right create such kind of right. So, if this is a tag array, so the data array remains exactly the same, there is no different data array remains the same in all the caches more or less in both direct map set associative, fully associative. The data array remains the same is only the tag array which we change. So, we will first create 32 such sets where each set contains four entries. Now, let say we take a block and we read its five index blocks. So, index is a set index. So, we read it five bits after discarding the lower six. So, let us say it points to this sets; after this we compare the remaining part of the address because these five are fixed right on the basis of these five, we have come to this set.

So, we need not considered it anymore then we consider the remaining part which is 21-bits. So, block address I really was 26, but out of that 26, we took 5 out and we use those five to access a set. So, all the entries in the set will have those 5 bits in common. So, we did not store it we take the remaining 28 bits and. So, here we have a 21-bit tag and we extract the 21-bit tag from the address as well and we just compare with each and every entry in this set. What do we compare, we compare what is stored in this in these entries and the tag part of the address which is 21 bits if there is a match we declare a hit otherwise we declare a miss.

And furthermore, if there is a hit then we take where there is hit and read the corresponding entry from the data array.

(Refer Slide Time: 34:27)



So, there is a nice diagram over here. So, what the diagram says is that we first consider the address. So, the address will have lower 6 bits will be the block address I am sorry lower six bits will be the address of the bit within the block, so that is something that is discarded then we take 5 more bits, so that is the set index. And using the set index we access four entries of the set. And we take all four entries and compare that with the tag part of the address which is 21-bits.

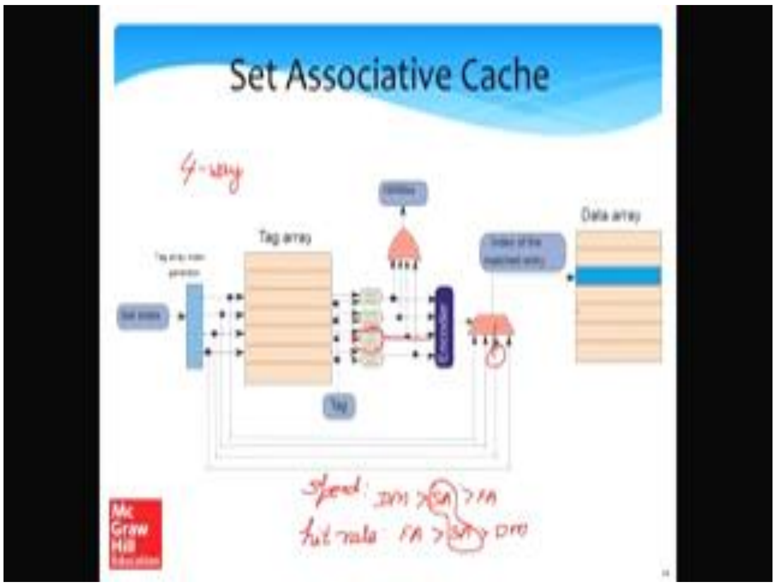
So, we do four comparisons. So, this can be an SRAM array does not have to be a CAM. So, we can still be verily fast. So, we compare the four entries, the four tags that are stored here with the tag part of the address; and after comparison we send the result to an

OR gate. So, any of the comparisons was successful it means to the data is there in the cache. So, we can declare a hit; otherwise we declare a miss.

And further more the encoded tells us that there was a match in which entry right this one this one this one or this one, and then what needs to be done. Well what we need to do is that we need to take the four indices inside the tag array and we want to choose one of them because one of them must have matched. So, let us assume it is this one. So, this one will flow via the multiplex to the entry in the data array and this one actually corresponds to this entry. So, basically corresponds to this entry, and we will go to the corresponding entry in the data array which is this one. If let us say we are a hit in this entry then we would have gone to another one. So, it does not matter.

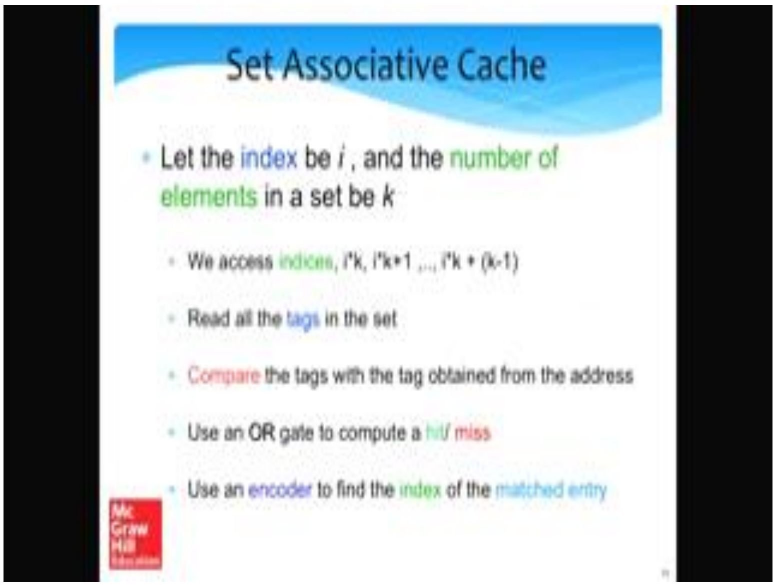
What we need to remember is that each entry in a tag array has a corresponding entry in the data array is a one to one correspondence. So, what is the new addition in these circuits? When the new addition in this circuit is this block and this block; so let me erase the ink once again and explain these blocks only in some amount of greater detail. So, instead of searching for a block inside all the entries we only search for it within a given set of lines that is the reasons called set associative cache this particular say each of this lines is called a way.

(Refer Slide Time: 37:11)


Set Associative Cache

So, this particular set associative cache one second this particular set associative cache is called a four way set associative cache which means that any of the four ways is the any of the four entries can contain the line. So, we take the set index which we again get from the address and we access the set in the part in specific the four entries within the set for each of them we combine. If there is a match when we get a hit and also we find out which one matched say with this line that is fine then we choose the corresponding entry which is this entry and we use it to access the data array. So, here is the mathematical look at it.

(Refer Slide Time: 38:12)

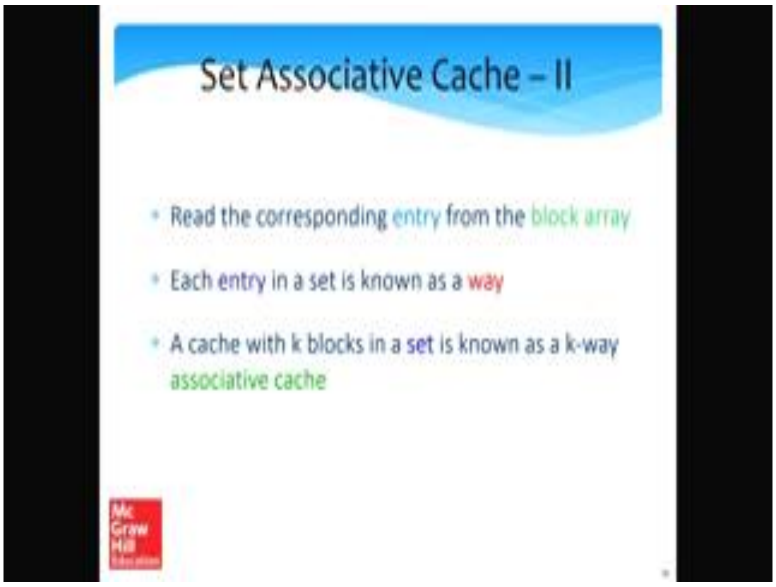


Set Associative Cache

- Let the index be *i* , and the number of elements in a set be *k*
  - We access indices, i*k, i*k+1 ,.., i*k + (k-1)
  - Read all the tags in the set
  - Compare the tags with the tag obtained from the address
  - Use an OR gate to compute a hit/ miss
  - Use an encoder to find the index of the matched entry

So, let the index be i, and the number of elements in a set be k. So, then the inde the indices that we access pretty much i times k plus i times k plus 1 till i times k plus k minus 1. We read all the tags in a set, we compare the tags with the tag obtain from the address within use an OR gate to compute a hit or miss. And finally, we use an encoder to find the index of the matched entry.

(Refer Slide Time: 38:44)

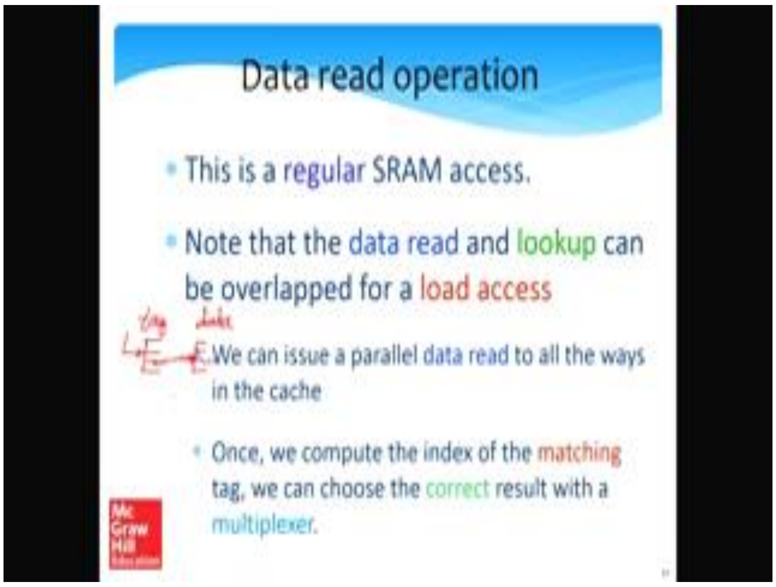


Set Associative Cache – II

- Read the corresponding entry from the block array
- Each entry in a set is known as a way
- A cache with k blocks in a set is known as a k-way associative cache

After that we read the corresponding entry from the block array further more each entry in a set is known as a way and if there are k blocks in a sets its called a k way set associative cache or a k way associative cache.

(Refer Slide Time: 38:59)



Data read operation

- This is a regular SRAM access.
- Note that the data read and lookup can be overlapped for a load access
  - We can issue a parallel data read to all the ways in the cache
  - Once, we compute the index of the matching tag, we can choose the correct result with a multiplexer.

So, before I going to the data read operation, I would like to look at this particular figure once again and the best way to look is to it is what we are did in earlier. So, in this particular cache what you would recall is. So, let us take a look at the tag size tag is what is stored in the tag array in the case of a direct mapped cache the tag was only nineteen bits right. So, in a cache of a direct mapped cache tag was only 19 bits in the cache of a fully associative cache the tag was 26-bits, because we needed a larger tag in the case of a set associative of cache the tag can vary between 26 and 19. And the reason being that we allocate some of the bits for the index for the set index for finding out with set, and the remaining bits become part of the tag because they are require to uniquely identify a block, because the total block address is 26 bits.

But we can use a part of the bits to first find the set; and essentially once that gets fixed, the remaining bits have to be use to compare with the tag array. So, as we can see is that as we increase the number of sets. So, as we increase the number of sets, the tag will actually gets smaller and smaller and smaller till you cannot increase it beyond this point which means at every set at this point contains one entry. Similarly, if we decrease the number of sets then what will happen is there a tag will get larger and larger and the in index part will get smaller and smaller till we reach the fully associative point.

At this point, a block can be there in any of the 128 entries of the cache. So, there is no index the entire thing the entire cache is like one set at this point the tag becomes the largest which is 26 bits. So, the advantage again of this design is like this that it is somewhere in between fully associative cache and a direct map cache. So, if you consider speed, speed of operation then clearly a direct map cache is the fastest followed by a set associative cache followed by a fully associative cache.

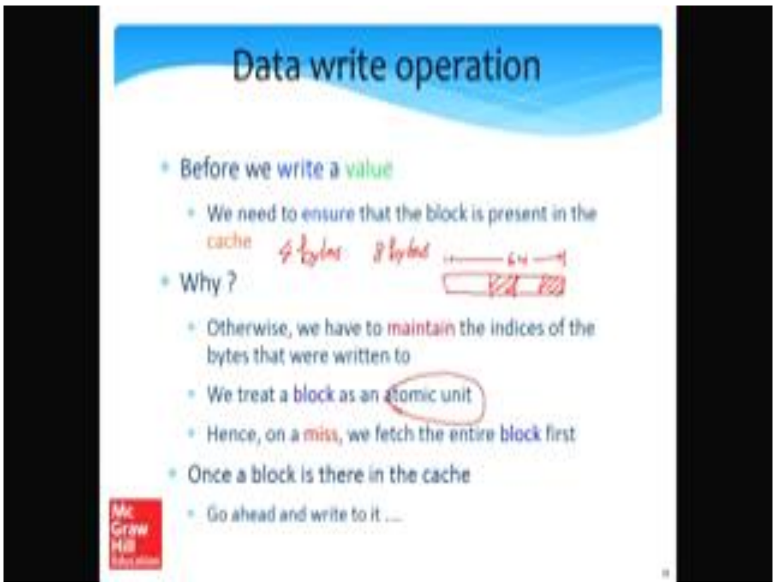
Now, we consider the hit rate we have the maximum amount of flexibility in a fully associative cache. We reduce the flexibility in a set associative cache and further reduce it in a direct map cache. So, as a result you know so there is no hardened fast rule that in a particular situation a particular kind of cache needs to be used, but as a general rule people prefer the middle path which is the set associative cache, it either a two way or four way in some cases 8 way associatively.

So, now let us take a look at the data read operation. So, this is a regular SRAM access into the data array in some cases the read and the lookup can be overlapped for a load access. So, let us see how. See, if we consider a set associative cache, what we can do is that we can. So, we will the first thing that we will do is that we will compute the index of the set; and if the set is 4 ways, we need to compare the tag with each of the with contents of each of the ways in the tag array.

Simultaneously, what we need to what we can do is that we can do a little bit of additional work; we can read the contents of all the four ways from the data array. So, in this sense, we are overlapping the data read with the lookup after that once we compute the index of the matching tag. So, let us assume that the third entry was the matching tag then the third entry of the data array is what can be chosen. So, we can sort of overlap both in the sense we are doing additional work otherwise what would we have done otherwise we would have read all the four tag entries done a comparison, found a match then for that particular match we would have read the entry in the data array.

So, let us say this is the tag array and this is the; but now we will let us do some amount of additional work, and let us read all the four entries are data array and in parallel read and compare all the four entries are the tag array. If we find a match, we can then choose one of them, but at least the time for reading the data we could overlap.

(Refer Slide Time: 44:23)



Data write operation

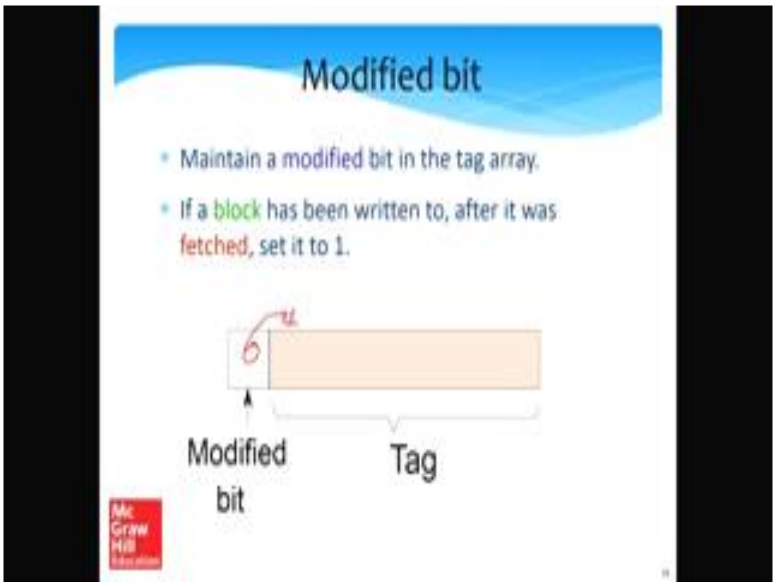
- Before we write a value
  - We need to ensure that the block is present in the cache
- Why ?
  - Otherwise, we have to maintain the indices of the bytes that were written to
  - We treat a block as an atomic unit
  - Hence, on a miss, we fetch the entire block first
- Once a block is there in the cache
  - Go ahead and write to it ....

4 bytes
8 bytes
64

Now, let us take a look at right operation. So, before we write a value we need to ensure that the block is present in the cache. So, most of the time students are not able to understand why is this case, why should this be done well it look if this is not done. So, so there were students you know I am going to write. So, I am going to write new data any way, but there is a problem the problem is that most of the time you are either writing 5 bytes if it is an int, you are writing 8 bytes if it is a long int or a double, but how large is a block. A block can be 32 or 64 bytes. And since a block we are treating as an indivisible unit, if we just let us replace this part of the block with new data, well we will have to remember this somewhere may be after this we can have another write operation which will write to this part.

So, if that is done, we will need to maintain some amount of state somewhere to remember which part of the block was written when. And you know which parts are untouched and which parts are new that is a lot of additional work. Also it breaks our abstraction that a block is an atomic unit. So, we have always had this neat and convenient abstraction, but a block is like one indivisible unit. Hence what we do is that if we have a miss, we first fetch the entire block old contents of the block after that we write the new data.

(Refer Slide Time: 46:04)

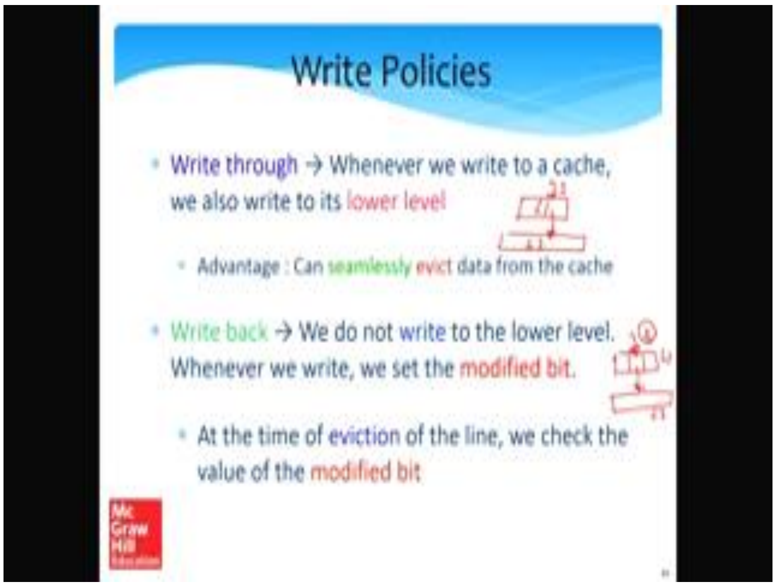


Modified bit

- Maintain a modified bit in the tag array.
- If a block has been written to, after it was fetched, set it to 1.

So, how do we do this? So, so we write the new data we also need to remember if you know new data was written to a block after it came to a cache. So, we maintain a modified bit in the tag array. If a block has been written to, after it was fetched into a cache we set it to 1; otherwise by default the entry remains a 0, but if you write to it then yes 0 becomes a 1.

(Refer Slide Time: 46:37)



Write Policies

- Write through → Whenever we write to a cache, we also write to its lower level
  - Advantage : Can seamlessly evict data from the cache
- Write back → We do not write to the lower level. Whenever we write, we set the modified bit.
  - At the time of eviction of the line, we check the value of the modified bit

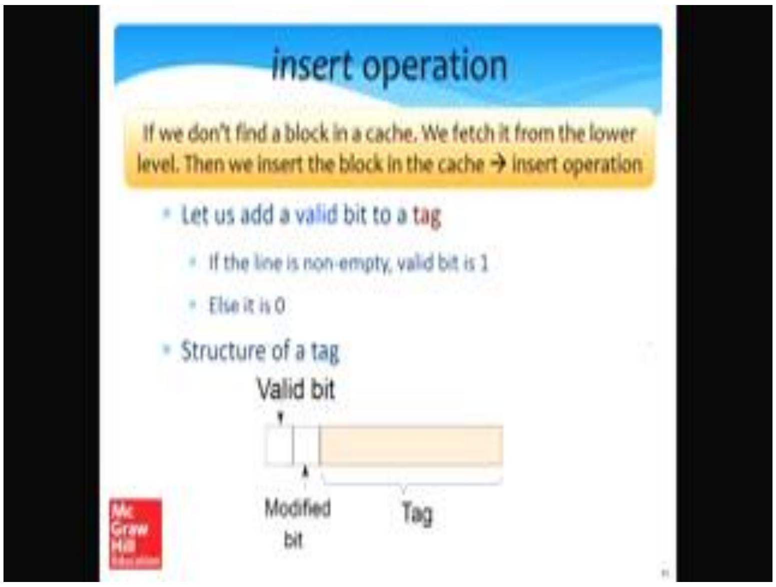
So, there are two policies one is write through and one is write back. So, whenever we write to a cache well. So, because we have inclusive we have an inclusive cache hierarchy any block which is present in the L 1 cache a block will also be present the same block the same address will be present in the L 2 cache; however, the block in L 1 cache can have newer data. So, there are two ways of handling this one is write through and write back. So, write through says that whenever we write to a cache we also write to its lower level and in a lower level you will have the block that is guaranteed by the property of inclusion.

So, this basically means that if I have a system like this with an L 1 and an L 2, I write to any block I do a store to any block in L 1 it gets propagated to the copy of the block in L 2. So, this has some advantages one advantage is that I do not have to maintain a modified bit. Well we needed in some cases, but in general, no. If I want to throw this block out of the cache i can happily throw it. So, we can sort of seamlessly evict data from the cache.

So, in that sense for write through actually you know unless we when we are doing a simple discussion a modified bit is not required, but in the case of a write back the idea is different, we do not write to the lower level and also whenever we write we set the modified bit to 1. So, the advantage of write back over write through is that we are sort of reducing the number of accesses to the lower level.

So, let us assume that there are hundreds or thousands of accesses to the same block in L 1 if it is write though every single access will go to L 2, so that will waste a lot of power and create a lot of traffic to L 2. What the write back cache instead would do is that it will keep on writing to L 1 and I will just set the modified bit to 1, nothing else. So, in write through cache in advantage was I needs to throw something out of L 1; I can just throw it out. In the case of write back, it is slightly different at the time of eviction of the line eviction means when it is been thrown out, we need to check the value of the modified bit. If it is not modified, well we can throw it out; if it is modified, we need to write it to the lower level right such that the changes are at least there.

(Refer Slide Time: 49:27)



insert operation

If we don't find a block in a cache. We fetch it from the lower level. Then we insert the block in the cache → insert operation

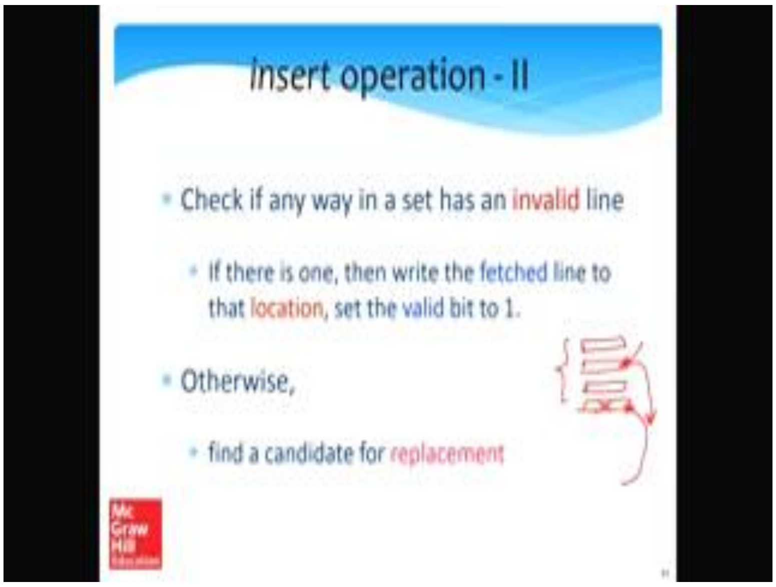
- Let us add a valid bit to a tag
  - If the line is non-empty, valid bit is 1
  - Else it is 0
- Structure of a tag

Valid bit
Modified bit
Tag

So, after write back and write through here again; what is the trade off actually let me just go back one slide. The trade off is that write through is far more traffic at the lower level, but evictions are cheap. Write back the advantage is the traffic at the lower level is low, but evictions are expensive.

Now, let us take look at the insert replace and evict operations. So, if we do not find a block in a cache we fetch it from the lower level when we insert block in the cache within insert operation. So, let us add some new state let us add a valid bit to a tag, if the corresponding line in the data array is nonempty which means there is some valid data, the valid bit is 1; otherwise it is 0. So, what is a structure of tag in a tag array? Now we have the tag part of the address, we have a modified bit and a valid bit.

(Refer Slide Time: 50:30)

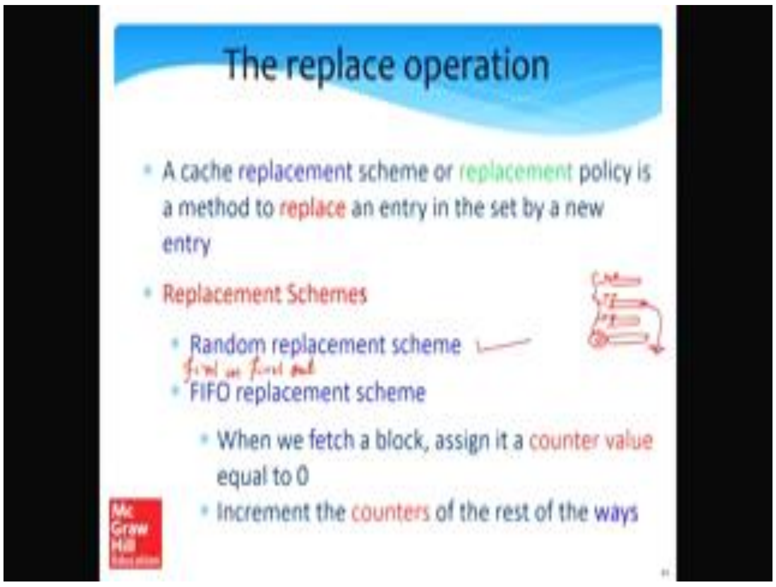


Insert operation - II

- Check if any way in a set has an invalid line
  - If there is one, then write the fetched line to that location, set the valid bit to 1.
- Otherwise,
  - find a candidate for replacement

So, when we are inserting a line, let us say it is coming from a lower level, we check if any way in a set has an invalid line, which means its valid bit is 0. So, line is empty, the entry is empty basically. If there is 1, then we write the fetched line to that location and set the valid bit to 1. So, consider a four way associative cache which has 4 entries for each set after mapping to a set we find that the last entry is empty. So, if any new line is coming that mapped to this set it can be put in the last entry. If that is not the case, we need to find a candidate for replacement throw it out which basically means evict and insert in this new position.

(Refer Slide Time: 51:21)



The replace operation

- A cache replacement scheme or replacement policy is a method to replace an entry in the set by a new entry
- Replacement Schemes
  - Random replacement scheme

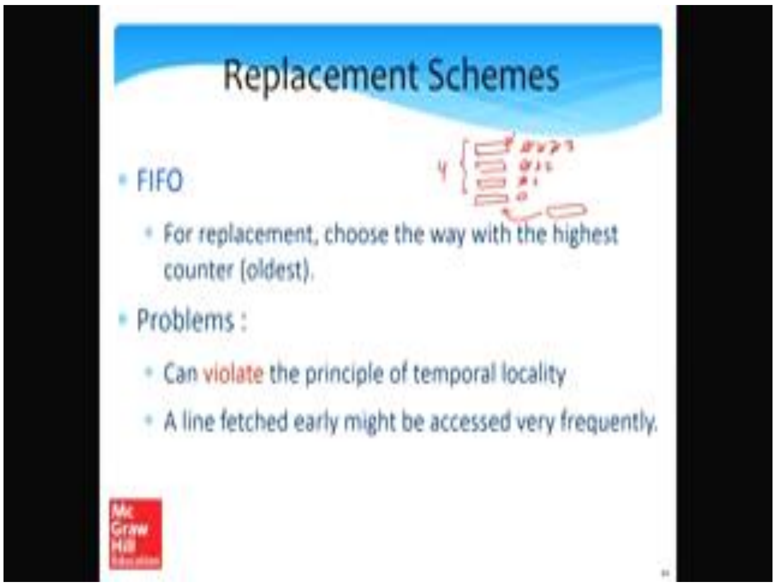
  first in first out
  - FIFO replacement scheme
    - When we fetch a block, assign it a counter value equal to 0
    - Increment the counters of the rest of the ways

So, replace operation means that we are searching for a candidate in a set for replacement. So, a cache replacement scheme or replacement policy is basically to find you know one victim right in the set. So, there can be many schemes they have their pros and cons. So, we can have a random replacement scheme where inside a set we can choose any of the ways at random right if all of them have valid data choose any one. So, recall that a fully associative cache is also a generalization of a set associative cache where the entire cache is one set. So, the same schemes applied.

So, one solution is just at random chooses any line and just throw it out. At this advantage is the temporal locality is being hampered and the reason that is happening is because may be you know one of these lines is very frequently accessed and this gets thrown out we will have more misses we can then have a FIFO replacement schemes FIFO is first in first out. So, in this scheme when we fetch a block we will assign it a counter value equal to zero and for the rest of the ways. So, we will keep a countered with every way in the tag array. So, when we fetch a block for the first time we will keep its counter as 0 and we will increment the counters of the blocks in the rest of the ways in the set. So, this will ensure that how do we know which block came the first by the one that has the highest counter.

So, if we have a three bit counter well the counter can at best reach seven, but that is all right. We will at least find one of the entries that are the oldest we will be able to throw it out.

(Refer Slide Time: 53:26)

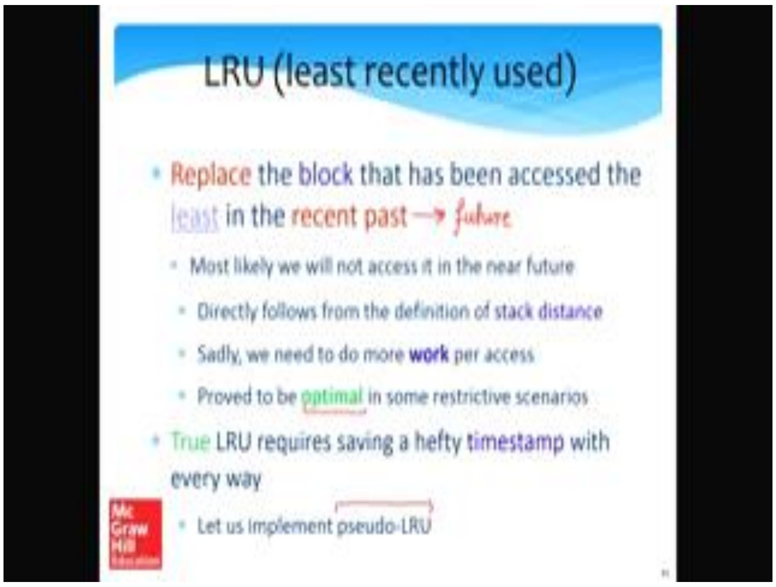

Replacement Schemes

- FIFO
  - For replacement, choose the way with the highest counter (oldest).
- Problems :
  - Can violate the principle of temporal locality
  - A line fetched early might be accessed very frequently.

As mentioned on the previous slide we choose the way with the highest counter for replacement. So, let me give an example. Let us consider a four way associative cache with fours entries. So, assume that initially everything is empty and we bring in the first block. So, it has a counter of 0. After that we bring in one more block into the set. So, that has a counter of 0, and this is incremented to 1, then we bring in one more block. So, we increment the rest of the counters this to 1 and this to 2, then again we bring in one more block. So, we increment this to 0, this is 0 to 1, 1 to 2, and 2 to 3. Then it so happens that this block which you brought in which is essentially the earliest the oldest is accessed extremely frequently right, this is this block is extremely popular. So, it is accessed extremely frequently.

And then we try to bring in one more block into this set the one that will get evicted in this block, but this is may be the one that should not get evicted and may be this block was getting accessed extremely frequently. So, this will violate the principle of temporal locality and. So, a line that is fetched early might be accessed very frequently. So, this might not be the right thing to do. What we want to do instead is evict that line which has the least probability of being accessed in the future. There is no way of knowing what is going to happen in the future, where at least we should look back into the past and try to find a line which most likely, we will not access in the future right in near future.

(Refer Slide Time: 55:23)



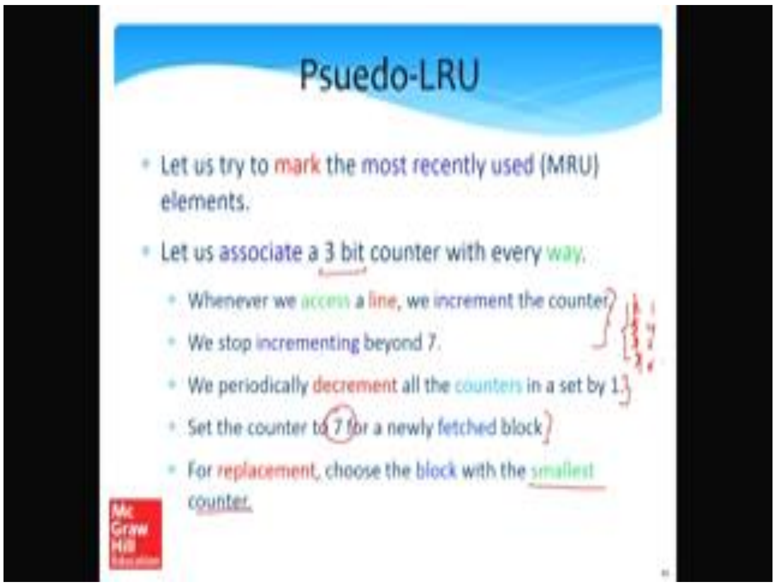
LRU (least recently used)

- Replace the block that has been accessed the least in the recent past $\rightarrow$ future
  - Most likely we will not access it in the near future
  - Directly follows from the definition of stack distance
  - Sadly, we need to do more **work** per access
  - Proved to be optimal in some restrictive scenarios
- True LRU requires saving a hefty timestamp with every way
  - Let us implement pseudo-LRU

So, one such scheme is LRU least recently used. So, the idea here is that we replace the block that has been accessed the least in the recent past. So, basically from the past we are trying to predict the future right from the past we are trying to predict the future. So, what we do now is that we try to find that block which let us say in a in a recent past right which was may be accessed the least in the last hundred cycles or something like that. So, this follows directly from the definition of stack distance.

So, we are essentially choosing a line which is the lowest in the stack, but we cannot really maintain a stack. It will involve a lot of work and it is not possible to that with a limited number of bits in hardware. Even though this approach has been proven to be optimal in some scenarios a true LRU is hard let us instead implement something called a pseudo LRU which in a sense gives us little most of what LRU promises.

(Refer Slide Time: 56:45)



Psuedo-LRU

- Let us try to mark the most recently used (MRU) elements.
- Let us associate a 3 bit counter with every way.
  - Whenever we access a line, we increment the counter.
  - We stop incrementing beyond 7.
  - We periodically decrement all the counters in a set by 1.
  - Set the counter to 7 for a newly fetched block.
  - For replacement, choose the block with the smallest counter.

So, pseudo LRU is like this let us try to mark the most recently used elements something that is not most recently used we can think of it as least recently used. So, let us associate a 3 bit counter with every way. So, we will pretty much argument an entry in the tag array to have additional 3 bits. Whenever we access a line we increment the counter and we will stop incrementing beyond 7. So, we will assume that 7 plus 1 is equal to 7, and then what we do is that. So, so let me may be you know till this point show an example.

So, let us say that we have different line. So, they can have different counts. So, then we access this line. So, from 4 it will become 5. Then we access this line it will become from 2, it will become 3, we can access this, this will become 3, again we access is becomes 4, we access this, this becomes 6 and so on. Then what we do we periodically decrement all the counters in a set by 1. So, periodically every few hundred cycles or thousand block cycles we decrement the counters by 1. So, instead of 3, 6, 4, 3, we will replace it right with instead of 3, 6, 4, 3, what will we have we will have 2, 5, 5 and 2. So, why do we this because if you do not do this ultimately all the counters will read 7 and remain at 7.

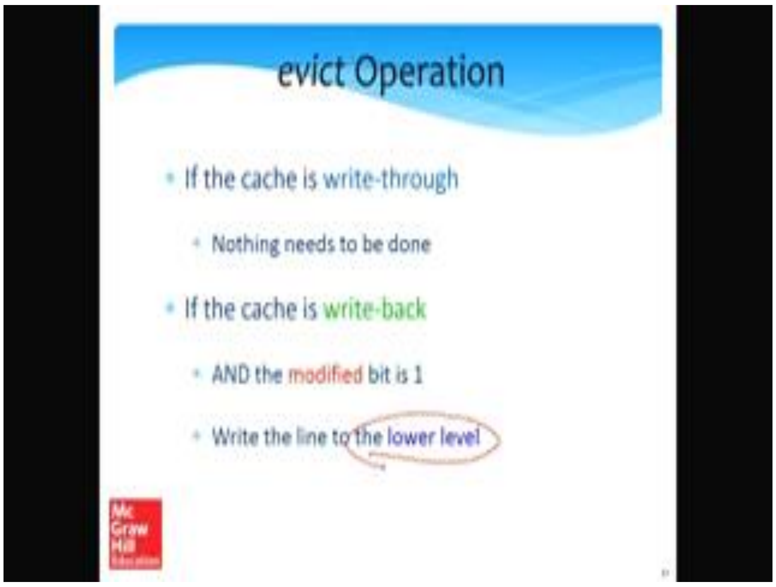
After this, what we do is that whenever there is a new access we will keep on incrementing. Now, let us say a new block comes. So, if a new block comes which one will a evict well the one we should evict the one with the smallest counter this means in the recent past it has been accessed the least. So, let us say there are two with count value of two. So, let us try to evict this one. So, when we throw this out, the question that remains is what should be the count of the new block. Well, the new block is the most latest say it should have the highest priority as per temporal locality. So, we will set its counter to 7.

Let us assume that after some time you know may be in the next thousand cycles none of these are accessed, so then gradually we will decrement that count by one. So, this will become 1, 4, 2 and 6. Then assume that we have some more accesses. So, then the counts will keep increasing. So, the logic here is that we always evict the block with the smallest counter and that is because this means that in a recent past this block has been accessed the least right.

So, if it in the recent past it has been accessed the least this means that even in the near future will most likely not access it, but something that has been accessed quite a bit in the recent past. We will be accessing it also quite a bit in the near future, so that is the reason we do not evict it. And also we set the counter to seven which is the highest we can for three bits for a newly fetched block. And the reason is that if a block is coming just new, we want to keep it for some time right as per the definition of temporal locality.

So, pseudo LRU is very popular it is very commonly used and it has a lot of benefits associated with its. So, it is known to be one of the best replacement schemes and the idea is very simple we have a logic for incrementing whenever we access the increment do not go beyond seven in this case. Periodically we decrement because unless we do that all the counters will remain at 7. Whenever a block comes in, it comes in the highest priority which is 7 and for replacement we choose the block with the lowest counter; and the lowest counter means in the past it has hardly been accessed.

(Refer Slide Time: 61:03)

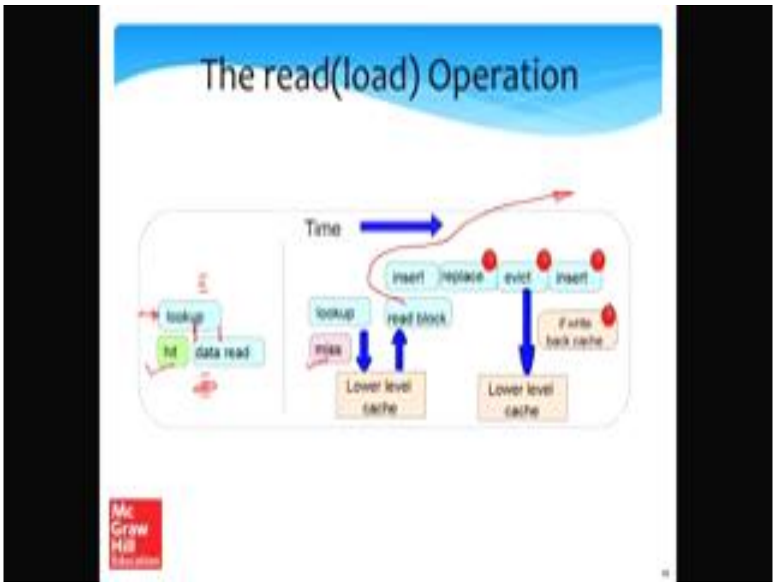


evict Operation

- If the cache is write-through
  - Nothing needs to be done
- If the cache is write-back
  - AND the modified bit is 1
  - Write the line to the lower level

Then we consider the evict operation well this is very very simple, if the cache is write through nothing needs to be done we can throw out the line. If the cache is write back, and the modified bit is 0 nothing needs to be done we can evict it. If the modified bit is 1, it means that new data has been written to this line. So, we need to write the line to the lower level and then evict that is the only difference.

(Refer Slide Time: 61:34)



The read(load) Operation

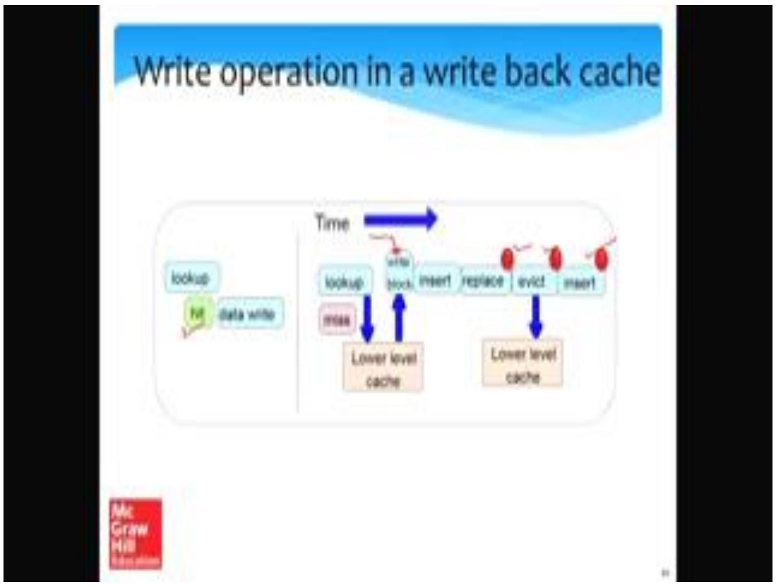
So, let us take a look at all of these operations in a diagrammatic fashion that will hopefully clear up things. So, let us take a look at the load operation right to load something. So, we can do a lookup. So, lookup essentially means that we access the set in the tag array, and we do we compare each entry in the tag array with the tag part of the address. So, a part of this particularly after computing the set index a part of the lookup process can be overlapped to the data read operation. So, this can be the period of overlap. And we have already discussed how we can get an overlap and this mind you happens in the case of a hit and this is what we do in the case of a miss.

So, let us discuss the case of a hit first. So, in this case, once we have started the process of reading all the four tags in the four way associative cache, we can also read they corresponding data values from the data array. So, we can start reading them in parallel even though all four will not be required, but that is ok we are sort of trying to reduce the time.

Once at the end of the lookup stage we know whether it is a hit or miss, so if it is a hit and we know exactly which index has the data value, we can then after the read finished choose that one. So, at least by a little bit of overlap, we save some time. Instead if there is a miss then so basically we do a lookup in the lower level cache, we read the block and then we insert it right, and after inserting the block it is possible. So, these question marks indicate that it will not happen all the time it can happen sometime. We find a candidate for replacement right one of the lines and then we evict it.

So, if it is a write back cache it there is a necessity of there can be a necessity of writing it to the lower level if it is modified and then again we insert the line that has come. After that we again go to this stage where we well in this case we know where exactly we put it. So, we can issue a data read after this. I am sorry we do not have to actually go here, I am sorry about that I just take back what I said in the last fifteen seconds I am sorry. What we need to do is that we do not issue a data read because it already read the data from the lower level. So, we do not need to read it once again from the higher level. So, since we have read the data, the values are already available. So, this can be passed as an output fine.

(Refer Slide Time: 64:43)



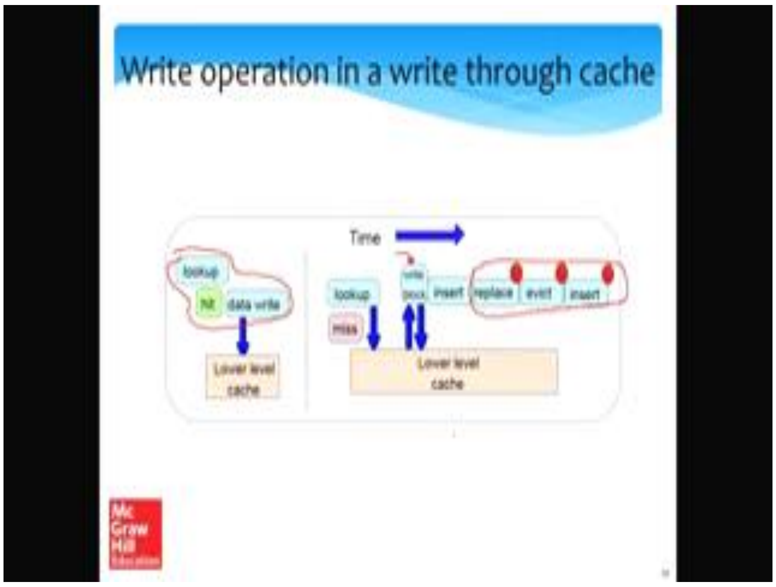
Write operation in a write back cache

So, let us take a look at a write operation in a write back cache. So, in the case of a write operation, if there is a hit, we cannot have an overlap between lookup and data write, so that is not possible because otherwise you know we cannot write to all the four data blocks right we will end up corrupting at least three. So, the right approach is that we do a lookup first if the data is there, we address the right entry in the data array and we do a write to it.

Similarly, however, if there is a miss then we do a lookup in a lower level cache, we get the block and we try to write the block. So, the way that we try to write the block is something like this that we first insert it. So, first write the block which basically means that the new data that we want to write is written to the block that we read from lower level, and then we try to insert the you know block with the new contents into the cache. If you find a place to insert well and good otherwise we need to follow the replace evict insert you know this sequence of three operation.

So, you find a candidate for replacement we evict it. So, there can be a necessity of writing it to a lower level also, because it is a write back cache and then we insert the new line.

(Refer Slide Time: 66:27)



Write operation in a write through cache

If it is a write through cache, something else needs to be done. So, if it is a write through cache will this part remains the same, in the case of hit we do a lookup and a data write, but we also send the right to the lower level. So, this was what was if you look back at the previous slide this was not happening over here, but in the case of a write through cache, whenever we write we need to send it to the lower level.

If there is a miss, what we do is that we do a lookup in a lower level cache, we get the block, we write the new contents to the block write, so whatever integer or float or double whatever you wanted to write. And the new contents of the block are once again written to the lower level cache. And also at the same time we try to insert it in the upper level cache. We find a place well and good otherwise we need to do a replace you know the sequence of replace evict and insert operations.

So, note that in this replace, evict, insert sequence nothing needs to be done, we do not need to access the lower level cache. And the reason is that is a write through cache and we do not have any lines in the upper level which have any modified data. So, the modified bit essentially for all write through caches lines is 0. So, this is essentially because we have written the block at the time of at an earlier point of time which is this point to the lower level.