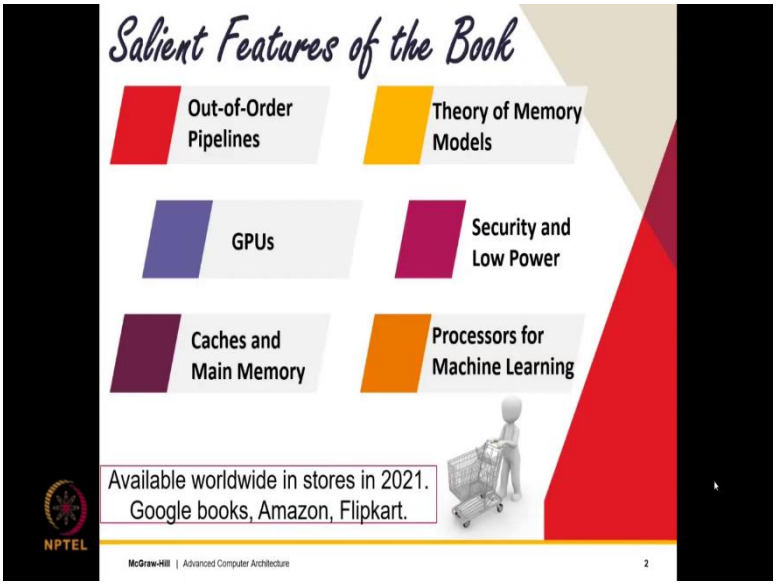
Advanced Computer Architecture

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

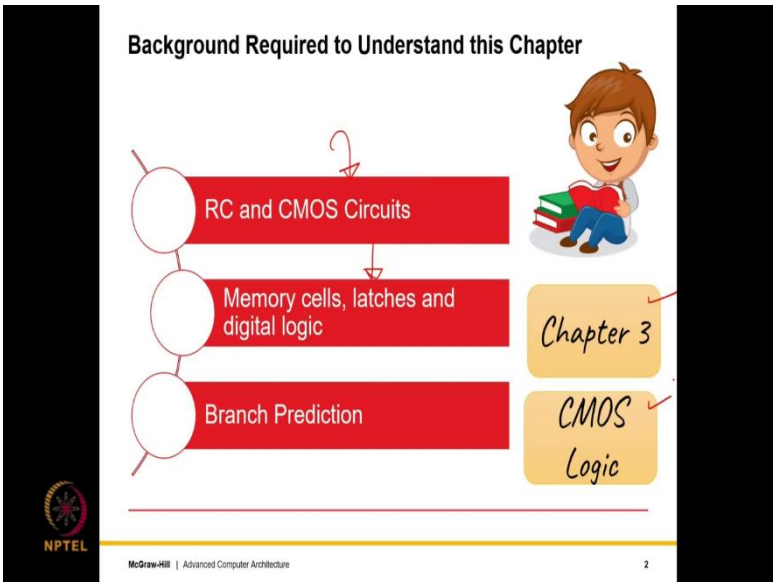
Lecture - 19
Caches Part - I

(Refer Slide Time: 00:23)



Welcome to the chapter on Caches. The design of caches and the memory system is by far one of the most important parts of a modern computer system and hence, we will discuss this fairly long chapter over a series of several lectures.

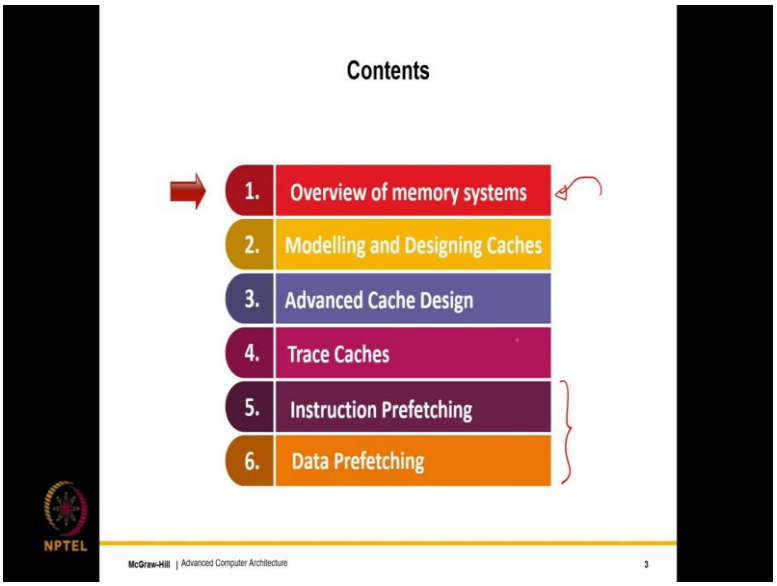
(Refer Slide Time: 00:53)



So, first the necessary background. So, a cache is pretty much made of S-RAM and D-RAM arrays. So, we would need a background of RC circuits and CMOS circuits. So, we do not need a lot of extensive background; but at least a little bit is required such that this part can be understood. So, the notion of a time constant in an RC circuit, somebody should know that and also all about current sources and voltage sources.

Of course, memory cells, latches and digital logic it kind of flows from here and we will develop mechanisms to guess which memory addresses, we will access in the future and pre fetch them which means read them early. So, for this the knowledge of branch prediction which was discussed in chapter 3, this also is required. So, chapter 3 is definitely required and of course, knowledge of CMOS logic is required. Keeping these in mind, let us proceed.

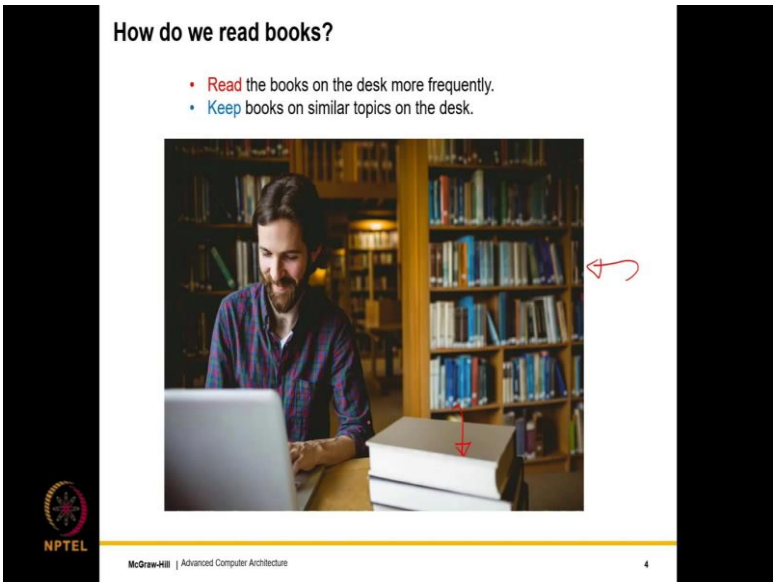
(Refer Slide Time: 01:58)



So, we will first discuss an overview of memory systems. This is essentially an extension of undergraduate knowledge is think of this as a recap and a little bit more. Methods of modelling and designing caches after this, where we will look at how tools such as cacti model and design caches. Then, we will look at advanced aspects of cache design. Intel has produced a very novel design called a Trace Cache.

So, we will discuss the idea of trace caches next. And finally, the last two parts of this chapter will focus on prefetching. So, prefetching's is conceptually similar to branch prediction. Here, we guess the addresses that will be accessed in the future and we try to fetch them from memory.

(Refer Slide Time: 02:58)

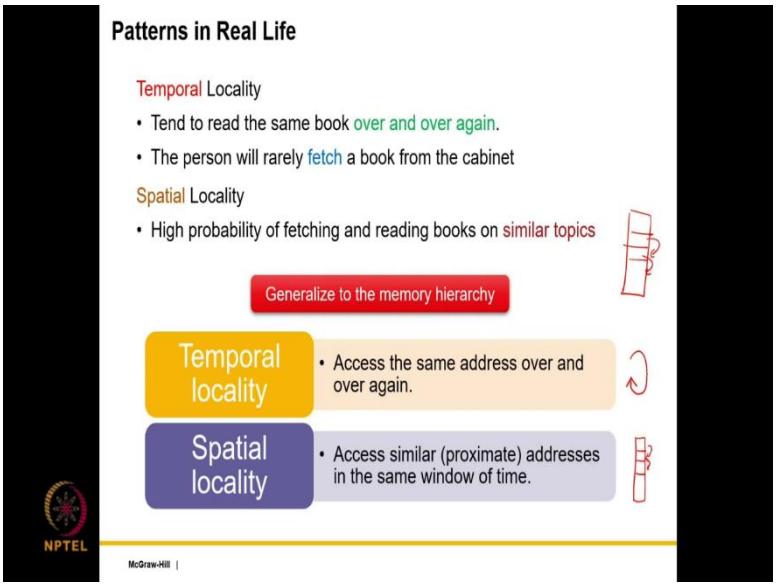


So, without further ado, let us try to introduce the memory system. So, before introducing the memory system, I would request all of you to take a look at the figure here. So, what this person is doing is that this person is accessing his laptop. There are a few books over here and of course, behind him you see a shelf of books.

So, what is his typical behaviour? Well, the typical behaviour is that the most frequent data, the most frequent instructions that he is accessing on his laptop something that is less frequently required is placed slightly further away and these books are on the desk and then, we have a large number of books on the shelves. These books he will rarely access.

But nevertheless, they are required and you can clearly see that the laptop is a small device; the books are bigger and the shelves are even, the shelves are much much bigger than the books. So, the first insight is that whatever you need frequently, you keep it close to you, number 1; number 2, you keep books on similar topics on the desk. So, for example, if he is searching for ideal vacation spots, then he would also have a little bit about adventure sports and so on, he would be reading other similar topics and these items will be on his desk.

(Refer Slide Time: 04:43)



So, this kind of mimics patterns in real life. So, we are talking of two important patterns in real life namely temporal locality and spatial locality. So, temporal locality is that you tend to do the same task over and over again in a short window of time. See in this case, the person is reading the same book over and over again.

So, this is an example of temporal locality and furthermore, what you would see is that the person will rarely fetch a book from the cabinet. So, basically, the desk you can think of as a work area which is the most frequently accessed. Spatial locality is as I discussed a high probability of fetching and reading books on similar topics.

In the sense, that well if he is thinking about his vacation, he might be reading books on adventure sports, he might be reading books on cooking. So, basically he always tend to do in a small window of time similar kind of activity. So, this is a pattern in real life. This is a pattern that is a pattern of human behaviour.

The question is do we have something similar in computers? So, well, we can generalize this behaviour to the memory hierarchy. So, the temporal locality is that the way he translates is that he access the same address or the same set of addresses over and over again in the same time window.

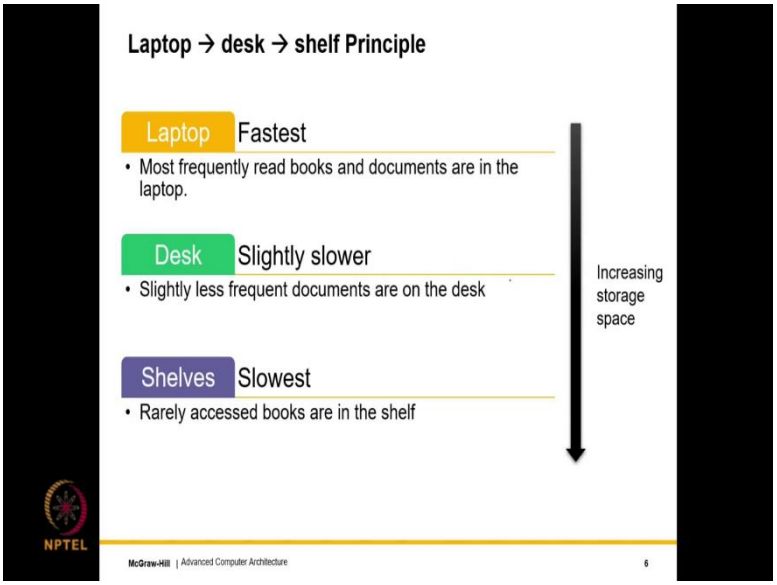
So, in the same window of time, the same set of addresses are accessed over and over againand spatial locality is that he access similar addresses which are close by, which are also called proximate addresses in the same window of time.

So, temporal locality is the same thing over and over again and spatial locality is that in a small window of time, we access similar addresses, nearby addresses. Why is this the case? Well, program spend most of their time in loops. So, given that a set of instructions keep looping. So, at least at the level of instructions, we will access the same instructions over and over again, that is temporal locality.

Second, programs typically work on let us say large arrays or similar data structures in that case, what happens is that you sequentially scan the array levels. So, this is an example of spatial locality because if the first element was accessed, then we will access the second, then the third then the fourth and so on. So, this is an example of spatial locality.

So, because of the way that programs typically are, so we can even think of a regular program as an array also. So, we access one instruction, then the next, then the next and so on. Often sometimes we loop, but at least this is the this is the default behaviour without taken branches. Here, there is spatial locality and because of loops, we have temporal locality.

(Refer Slide Time: 08:07)

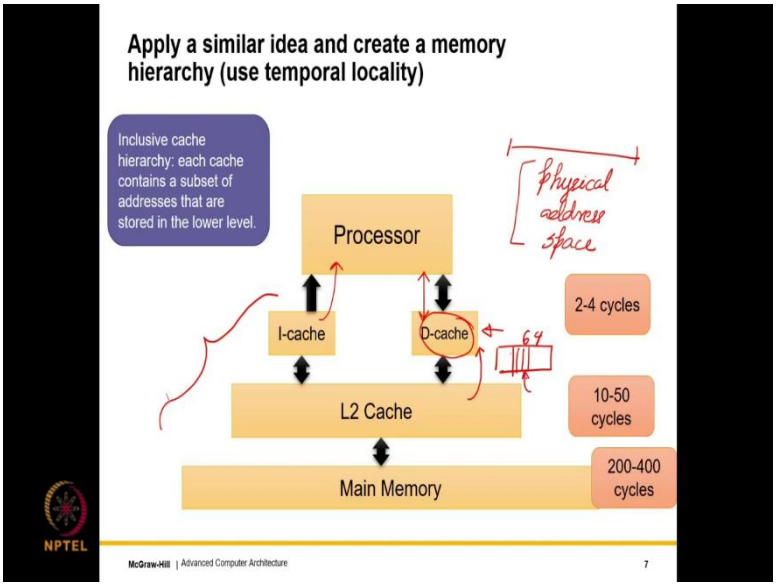


So, the we are in a position to make a laptop, desk, shelf principle. The laptop is the fastest. So, the most frequently read books and documents are in the laptop. The desk is comparatively slightly slower. So, slightly less frequently accessed documents are on the desk; but it clearly has as you can see more storage space.

Well, again that is not clear. This is just an analogy. But you can have let us say 10 more laptops on the desk which will give you more storage space and then, you have shelves. The shelves are clearly the slowest as you can see, that we have extremely rarely access books on the shelf, but they are also the largest.

So, in terms of storage, it is increasing because it is not it is strictly apples to apples comparison in laptop and a desk, but you can assume that the desk has other, it can have other laptops, storage drives, pen drives and so on. Say in principle, it is bigger than the resources of a single laptop.

(Refer Slide Time: 09:18)



So, we can apply a similar principle to the memory system, where we will have the processor and the processor in this case is the person who is reading or writing the data. Typically, we have two small memories that are attached with the processor. So, one is an I-cache, the other is a D-cache, they are like the desk or let us say they are like the laptop in front of the processor and they are small and fast.

So, smaller is the memory, faster it is. The latency of an I-cache or a D-cache is 2 to 4 clock cycles and the I-cache supplies instructions to the processor and the D-cache exchanges data with the processor. So, you can either read or write the data cache.

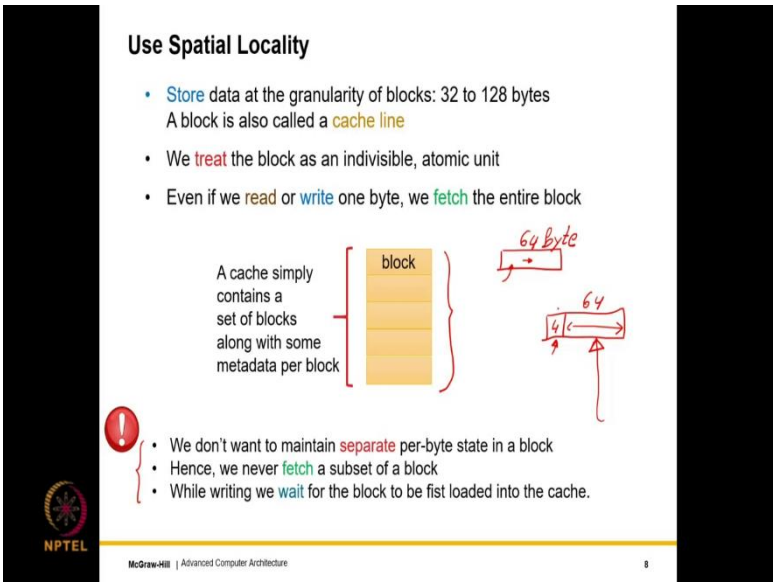
So, these are very fast memories, but they are fast because they are small. So, they store a subset of the address space, the most frequently accessed data. Then, we have the L2 cache. The L2 cache is a level 2 cache, which is comparatively much slower; its latency is between 10 and 50 cycles.

Then we can optionally have an L3 cache or an L4 cache which are bigger and slower. In modern server processors, they often have an L3 cache, even though having an L4 cache is very rare. After the L3 cache, we have the off chip main memory which is not there in the processor chip, but its outside. It is on the motherboard. It has a long access latency something between 300 and 400 cycles; 200 and 400 cycles.

So, the hierarchy is inclusive. What this essentially means is that the I-cache contains a subset of all the data in the L2 cache and the L2 cache contains a subset of all the data in the main memory right. So, the hierarchy in this sense is inclusive. So, you can have an exclusive cache design, but that is very rare. So, that is the reason in the rest of this chapter, we will assume that the entire hierarchy is inclusive.

So, the set of addresses in L2 cache the superset of the set of addresses in the data cache and the same holds true for the addresses in the main memory and the L2 cache. So, all the addresses that a program uses, so they this address range is known as the physical address space. So, we will look at this abstraction a little later; but for the time being; let us assume that the main memory contains data for all the addresses that the program will use.

(Refer Slide Time: 12:11)



So, let us look at spatial locality how do we leverage it. So, for spatial locality, what we can do is that instead of storing data at the byte level or at the level of what is called memory words which are essentially blocks of 4 bytes, we can store data at the granularity of blocks which are 32 to 128 bytes. A block is also called a cache line.

We treat the block as an indivisible atomic unit. In the sense, we fetch an entire block in one go and we never kind of like fetch half a block or transfer half a block. Say block is an atomic unit for us. The advantage is that even if we read or write a single byte within the block because of spatial locality, there is a very high likelihood that we will access the other bytes within the block.

So, because of that, what we do is that we fetch and operate on the entire block in one go. What is a cache? Well, a cache simply contains a set of blocks with some metadata. Metadata is some additional information per block. So, what do we do? Well, we create these blocks.

Let us say we create 64 byte blocks, because of spatial locality, if we access any memory word which is a 4 byte chunk. If we access any memory word within a block is a very high likelihood that we will access other memory words within the same block and a cache basically contains a list of the most frequent blocks, that the program is expected to access.

So, in this case, what will happen is that the processor will access the cache with a very high likelihood. It will find the block within the cache and then, it will read or write the corresponding bytes. So, there are certain important points here to keep in mind. The first is that we do not want to maintain any state, any per byte state in a block.

So, which basically means that we will keep some high level information for a block which is basically its memory address and so on. But at the level of individual bytes, we do not want to spend any space keeping any extra information. So, that is the reason that let us say that we have a 64 byte block and maybe we want to write the first 4 bytes.

Let us assume that this block is not there in the cache. Well, then you can always argue that if I just want to write and the data is going to be overwritten anyway, I can simply create space for a block and write the first 4 bytes. But that is not the way it is going to work because what about the remaining 60?

So, the remaining 60 have to be fetched from the lower level and even if you are writing 4, since a block for us is an indivisible atomic unit, we never want to have a like a partially formed block, where some blocks are written; the rest of the contents of the rest of the bytes are not known. We never want to have a partially formed block with some written bytes and the rest unknown bytes, we do not want to do that.

So, what we do is that if we want to write even 1 byte to a 64 byte block, what we do is that if the block is not there in the cache. For example, if the block is not there in L2 cache, we fetch it from the lower level. Let us say it is found in L2 cache. We fetch the entire 64 bytes. Once the entire 64 bytes are there in the data cache, then we access the corresponding bytes and we write to them. This is important. This is kind of a trick question and many students often miss it.

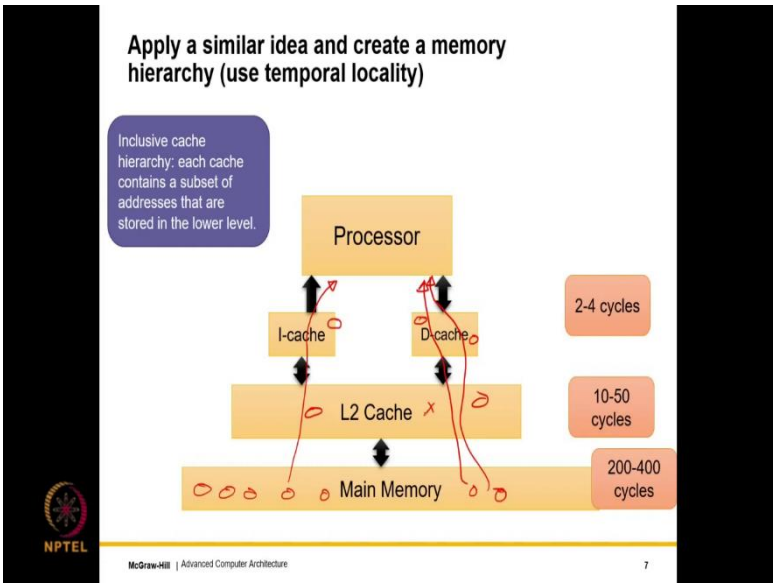
Because they say that look if I need to write to the 8th, 9th, 10th and 11th byte, what is actually the need for fetching the rest of the bytes? Well, the need is that if you do not fetch, you will somehow have to remember the fact that look only the 8th to 11th bytes are new, they have been written and the rest of the bytes pretty much are junk.

Remembering this requires computational effort, it requires storage space.

So, this is nontrivial. It also breaks our simple abstraction that a block is a single atomic unit. Hence, we never operate on subsets of it per say. While reading, it is easy because the entire block will be there. If we want to read the 10th byte, we go ahead and read it.

But while writing, we first ensure that the entire block is there in the cache and only then, we write, not before that. So, what is the basic protocol here that I am talking of? The basic protocol that I am talking of here is like this that when a program starts, all the blocks that it needs are in the main memory.

(Refer Slide Time: 17:37)

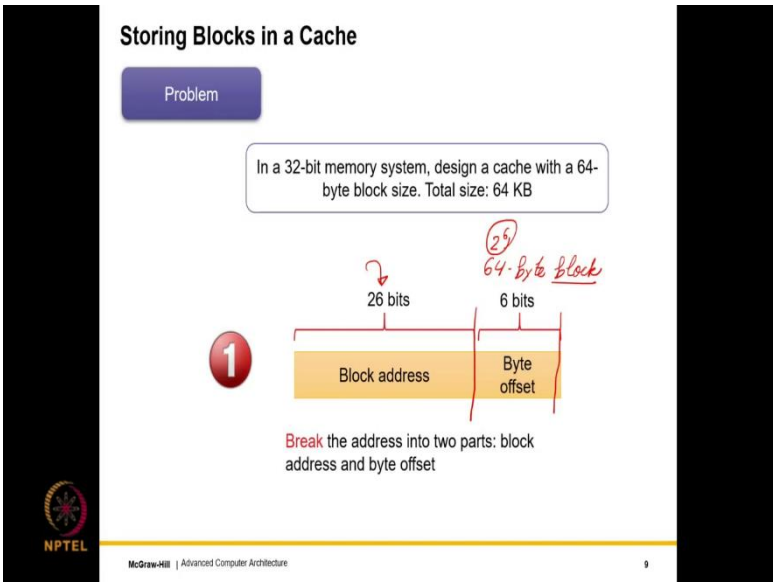


Gradually, what will happen is that the program will start executing. So, let us say it needs a certain block, it is not there. So, then it is kind of fetched from main memory. So, copy of it is maintained in the L2 cache; a copy of it is maintained in the I-cache and the same for data blocks as well; a copy is maintained in the L2 cache and a copy is maintained in the data cache.

So, the processor will always read and write from the immediate caches which are the I-cache and the D-cache. The processor will not directly access the rest of the caches. But the I-cache and D-cache they are like the desk in front of a person or in our example the laptop. So, so they are the immediate places, where the processor needs to read or write and pretty much they are small.

So, even though, they do cache most of the frequent blocks; occasionally, we will not find one. So, we again need to fetch it from a lower level and because our hierarchy is inclusive, it will never be the case that there is a copy of a block in the D-cache, but it is not there in L2 cache. This is never going to happen.

(Refer Slide Time: 19:04)

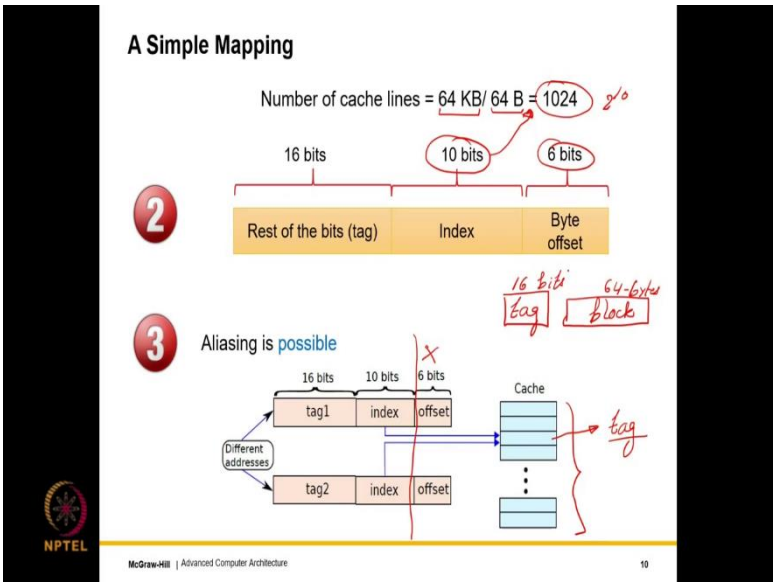


So, how do we store blocks in the cache? that is an important question. So, in a 32 bit memory system, let us design a cache with a 64 byte block size and the total size of the cache will be 64 kilobytes. So, let this be our running example that our memory addresses are 32 bits long, our total cash size is 64 kilobytes and the size of each block is 64 bytes.

So, with this assumption in mind, with this problem in mind, let us try to design a cache for it. So, what we do is we break the address into two parts; we divide one part, we call one part a block address and we call the other part a byte offset.

So, given that we have a 64 byte block, 64 bytes is $(2)^6$ bits. So, we need 6 bits to address a byte within the block. So, let us refer to this as the byte offset. The remaining 26 bits, because the length of the entire memory is because the length of a memory address is 32 bits, the remaining 26 bits are the block address. So, they will be used to uniquely identify a block.

(Refer Slide Time: 20:45)



So, let us proceed; the number of cache lines is 64 kilobytes divided by 64 bytes. So, block is also called a cache line and these terms are typically used interchangeably, even though we will make a slight difference slight distinction. A block will be referred to as the contents the 64 byte contents and the cache line will be referred to the location in a cache that stores a single block.

So, the number of cache lines that we will have is 64 kilobytes divided by 64 bytes which is 1024. This is 2 $(2)^{10}$. So, we will have 6 bits to specify the byte offset. We will have 10 bits to specify, I am talking of a simple mapping here.

So, we can use as I said 6 bits to specify the byte offset within the block and given the fact that a cache can store 1024 blocks, we will use the next 10 bits known as the index to actually uniquely address each block. So, this is shown over here that we take two 32 bit addresses; we remove the bottom 6 bits because they specify the offset of the byte in the block.

So, this part is removed. The next 10 bits are what are known as the index and they tell us that in this cache with 1024 blocks which block does it map to. So, this is a direct mapped addressing scheme. So, this was kind of similar to the way that we were actually mapping saturating counters in branch predictors.

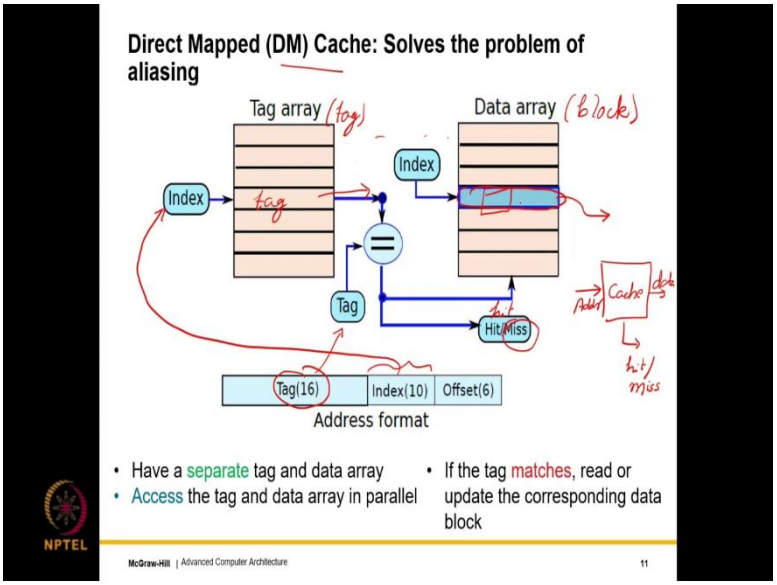
So, here unfortunately, we will have the problem of destructive aliasing as we had in branch predictors. So, there is one important difference here in branch predictors, even if there was destructive interference, there was aliasing, it was not that big of an issue. Mainly because even if there is a mistake in a branch predictor, it is not a correctness issue.

In the worst case, we will have a wrong prediction and that will lead to a loss in performance. But there will be no issue in correctness. But in this case, there will be an issue with correctness because we can clearly see that different addresses are mapping to the same entry.

So, we can use the same trick. We can have a tag. In this case, the tag is the remaining bits, the remaining 16 bits and the tag can also be stored along with the block. So, we, so this is what I was referring to as the metadata. We can store the tag and then, we can store the contents of the block.

So, this is of course, 64 bytes and in this case, the tag is 16 bits. So, if we do that, then along with this mapping which is something similar to what we have done in branch prediction, there will be no aliasing. Because we will of course, we will always verify the tag and only if the tag matches, we will actually access the block.

(Refer Slide Time: 24:08)



So, this has been in a sense been a stand been a standard narrative in this course from the days of branch prediction. So, we can use this simple idea to design a simple cache known as a direct mapped cache which solves the problem of aliasing. So, we have two arrays; we have a tag array and a data array. The tag array stores the tag and the data array stores the block.

How these arrays are created? Well, that is a subject of future study in the next subsection, when we will discuss the S-RAM and CAM arrays. But let us bear in mind that we have two arrays in a cache; one is the tag array, the other is the data array and of course, they are the same number of entries with a one-to-one correspondence.

So, using the index which in this case in our running example is the 10 bits over here, we access the tag array. We read the tag and along with that we also fetch the tag part of the address of the memory address that is being given to the cache. So, think of the cache as kind of a black box and let us see if it is a load, if it is a read, we give it the address and out comes the data, if the data is there; otherwise, we get a hit miss signal.

Along with that we get a hit miss signal. So, the miss tells us that the data is actually not there. So, from the address, we extract its upper 16 bits which is the tag part of the address and we compare the stored tag with the tag part of the address.

If they are equal, well we say it is a hit. If they are not equal, we say it is a miss. If we have a cache hit which means that the tag part of the address matches the tag, then we go to the same index over here and access the data array entry which is this block.

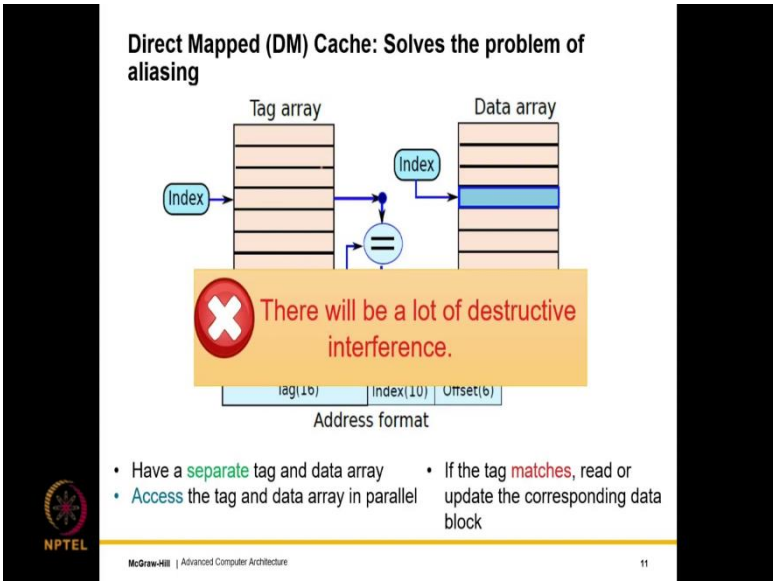
If it is a read, well if you want to read 4 bytes out of this, well we read those 4 and send it to the output. If it is a write, then we will also typically write to 4 or 8 bytes at a time. So, we will of course, select those 4 or 8 bytes and write to them. Often, just to improve performance, we access the tag array and the data array in parallel.

So, basically, we access these arrays in parallel because as we have seen we often do extra work in the interest of performance, even though some of that work might get discarded. Say in this case, what we do is that we access this entry of the tag array and we access this entry of the data array in parallel and we read both of them.

If it is a miss, well nothing we just discard this data. If it is a hit, then well then there is a performance advantage because we have anyway read this entry. So, we can directly send the bytes that the processor wanted to the processor or we can if you have returned to a temporary copy, we can transfer it back.

So, how exactly we do the read and write that as I said is the subject of a later section. But writes are not that important because they are not on the critical path. What are on the critical path is the read and hence, let us focus on reads. So, in the read, what we can do is we can read the tag and the data in parallel. If there is a hit, well it will save us some time. So, we can send the relevant part of the line to the processor. If there is a miss, then we just discard whatever we have read.

(Refer Slide Time: 28:21)

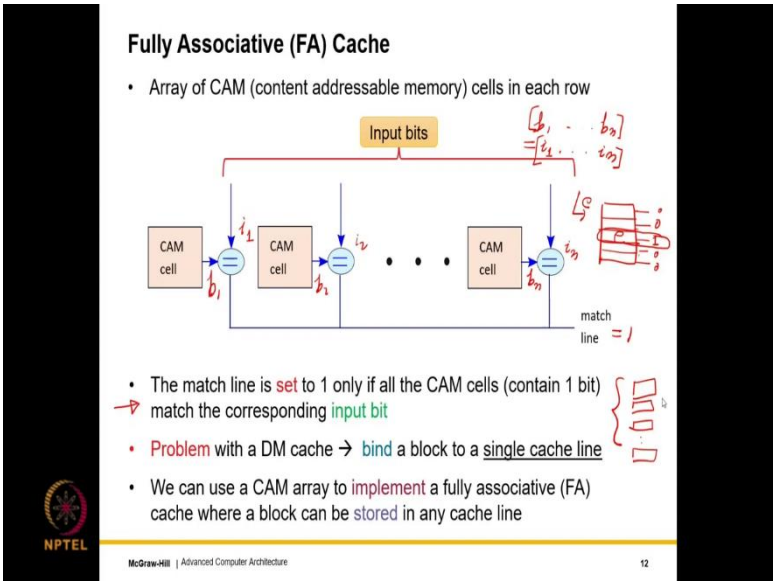


What is the problem? Well, the problem is that you are not able to see because of the ink. So, let me take out the ink. So, now, the problem is clearly visible that there will be a lot of destructive interference. Well, that there will be because often you know you might have some hot spots forming at some cache lines; in a sense, because of destructive interference some cache lines will actually see a lot of parallel accesses from different addresses and so, they will essentially be displacing each other's data leading to a large number of cache misses.

To avoid this, we should design a cache in a different in a different way to kind of reduce the probability of destructive interference aliasing because even though, we

even if we have enough space, it might still be the case that some entries in the cache, there is heavy contention; multiple addresses are mapping to them.

(Refer Slide Time: 29:21)



So, let us define a different kind of cell. So, let us design a different kind of cell. Let us call it a CAM cell or a content addressable memory cell. We have already seen this, if you would recall in the logic for retirement, where we talked about a CAM based RAT design. So, in this case, we have an array of CAM cells.

So, the way that CAM cells function is like this that each CAM cell stores a single bit. We compare this single bit with one of the input bits. So, let us say this is bit b 1 and the input bit is i 1, we compare them. Similarly, we compare b 2 and i 2. If all of them are equal which means if the entire bit string b 1 to b n is equal to, if both of these are equal, then the match line is set to 1; otherwise, the match line is set to 0.

So, what we have in a CAM cell is we have a we have an array of entries right and also we have an input entry. So, the input is the content. So, the entry will typically match with one of them. So, all of these, let me just maybe delete a little bit. So, we provide an entry and the output is a match lines.

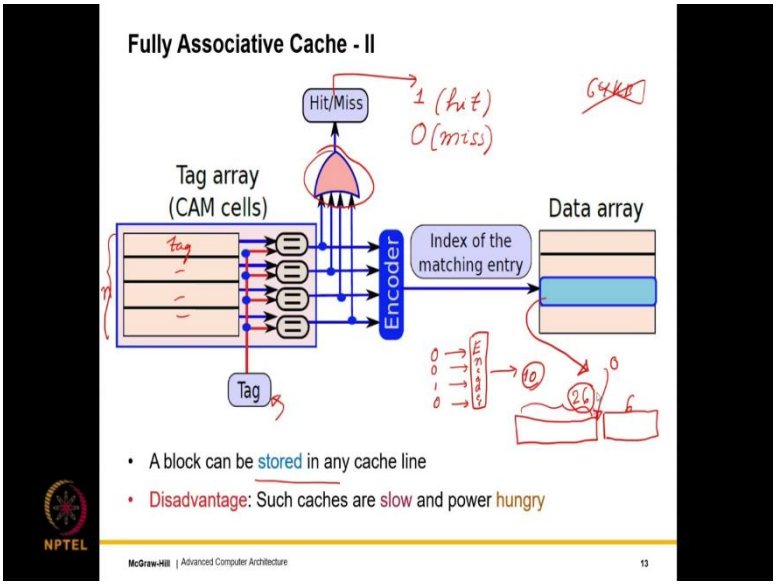
So, only one of the match lines will be set to 1 and the rest will be set to 0, because they will have different entries. So, this will tell us which row has the same has the matching entry has the same contents as e. So, we will see that this information is very

useful and it can be used to design a fully associative cache, where a block can actually reside in any cache line.

So, you have 1024 cache lines, say block can reside in any cache line and the advantage is that this will minimize the probability of aliasing or destructive interference. So, let us go back to the description over here, the match line is set to 1 only if all the CAM cells for each cancel contains 1 bit, all of them match the input.

So, to reduce destructive aliasing, we want to use this CAM mechanism and match lines to ensure that a block can reside anywhere in the cache. So, the cache has 1024 lines, the block can reside anywhere. So, this is what needs to be ensured.

(Refer Slide Time: 32:40)



So, the way that the fully associative cache will work is that again, it will have two arrays; a tag array and a data array. So, tag array is an array of CAM cells. So, we send it the tag part of the address, we will discuss how long it is and then, the we compare the tag part of the address with each of the stored tags, then the results. So, internally, we are essentially comparing.

See if there are n entries over here. So, in this case if n is 1024, we will have 1024 match lines coming out. So, we can or all of them compute a logical or of all of them, if you compute a logical or of all of them if one of them is 1, the final output will be 1 and then that will indicate a hit.

But if all of them are 0 which means that none of the tags actually matched, then the final output will be a miss. And also if you were to find out which cache line actually mapped actually matched, we need to use an encoder circuit. So, recall that an encoder circuit works like this.

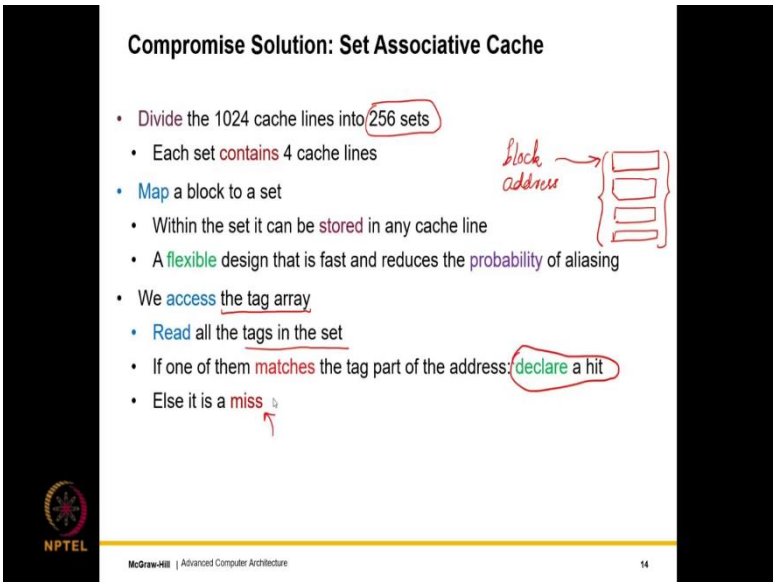
So, the encoder finds the index of that input which is 1. For example, in a 4 bit encoder if the third input is 1, the output; so this is the count starts from 0 0 1 2. So, the output is 1 0. So, in this case the advantage of having a setup like this is that the encoder can identify which cache line, the index of that cache line that has the matching entry.

Since there is a one-to-one correspondence between the entries of the tag array and the entries of the data array, what we can do is that we can fetch the corresponding entry from the data array. And we can provide that as the output.

So, of course, in this case, we will not be accessing the data and tag arrays in parallel, because we will have to then access all the entries of the data array which is not practical. So, what is the key idea a block can be stored in any cache line. The disadvantage is that such caches are slow and power hungry and we would typically not want to use such fully associative caches.

The reason being that in terms of power and performance, the trade offs are quite adverse. However, there are a couple of structures in which they are still preferred. But these structures are small, so they are not as large as 64 kilobytes. So, that no way this is too large. But these are much smaller, so they would have maybe 16 or 32 entries. In that case, a fully associative cache is preferred only if it is small.

(Refer Slide Time: 35:57)



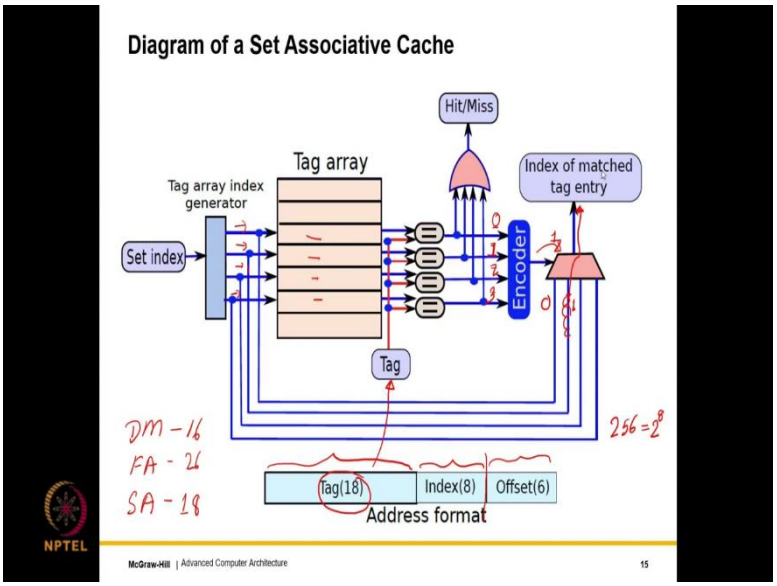
So, what is a compromise solution? The compromise solution is that I can have a set associative cache. Say in this what I can do is I can divide the 1024 cache lines into 256 sets, where each set contains 4 cache lines with contiguous indices.

So, instead of mapping a block to a cache line as was the direct mapped approach, I map a block to a set. So, what I do is that given the block address, I map it to a set and the set can contain let us say in this case, it contains 4 cache lines. So, map it to a set. Within the set similar to the fully associative scheme, it can be stored in any cache line.

So, I have to do a limited search, it is not a full search and also, I need not use the expensive CAM cell because the CAM cell is much more power I mean it demands much more power than the regular S-RAM cell that is used in the normal array. So, this set associative approach, where we have flexibility within a set is kind of a compromise.

It provides a degree of flexibility and it also reduces the probability of aliasing and it is sort of the best of both worlds. So, what we do the broad approach is that we access the tag array. So, we will have four tags in the set, we read all the tags in the set, then we compare. So, in this case, we use traditional comparators. If there is a successful comparison, we declare a hit; else, we declare a miss.

(Refer Slide Time: 38:03)



So, let us look at the diagram of a set associative cache, this will be clear. So, what I can do over here is that I will divide my bits in a different manner. So, the bottom 6 bits will remain the same because they specify the offset of the byte within the block. So, these will remain the remain the same; subsequently, we have 256 sets. So, 256 = $(2)^8$.

So, $(2)^8$ in this case is the index and the index uniquely maps the address to a set and the remaining bits which are unspecified which essentially any two alias addresses might have well any two alias addresses, these bits will be different. And they are the remaining 18 bits.

So, where does 18 come from? (8 + 6) = 14; (32 – 14) = 18. So, in this case, the tag instead of being 16 bits, the tag is 18 bits that is important to keep in mind. So, this is the right point of time to actually go back and answer one unanswered question in the fully associative cache which is how large is the tag in the full associative cache.

So, let me go back, quickly answer that before I proceed because I mean that is a vital part of this discussion as well. So, in the fully associative cache, well the bottom 6 bits will remain the same; but then, we are not specifying anything about which block can be mapped to which index, it is pretty much free for all.

So, if you think about it the tag will actually be the next 26 bits which is the entire block address because we are not using any part of the block address to index the cache line. So, since no part of the block address is being used, the entire 26 bits is the tag. Recall that in a direct mapped cache, the tag was 16 bits mainly because the index was 10.

In the case of a fully associative cache, there is no index, the index is 0 bits. Hence, the tag is the remaining bits which is 26. So, we can kind of think of the fully associative cache as a set associative cache, where of course, there is a single set and it encompasses all the lines. Now, coming back, so what have we seen?

What we have seen is that in a direct map cache, the size of the tag is 16 bits; in a fully associative cache, it is 26 bits. In the case of a set associative cache with a what is called a four way set which means that we have 4 cache lines in the set. So, this is referred to as a four way set. In this case, what we have is that we have an 18 bit tag alright.

So, the method of indexing such a cache is that well we first discard these bits; we take the 8 bit index. Based on this, we generate four addresses which are the addresses of the 4 cache lines within the set, which are these 4 cache lines. Then, we also take the tag part of the address and we compare the tag part of the address with the stored tags.

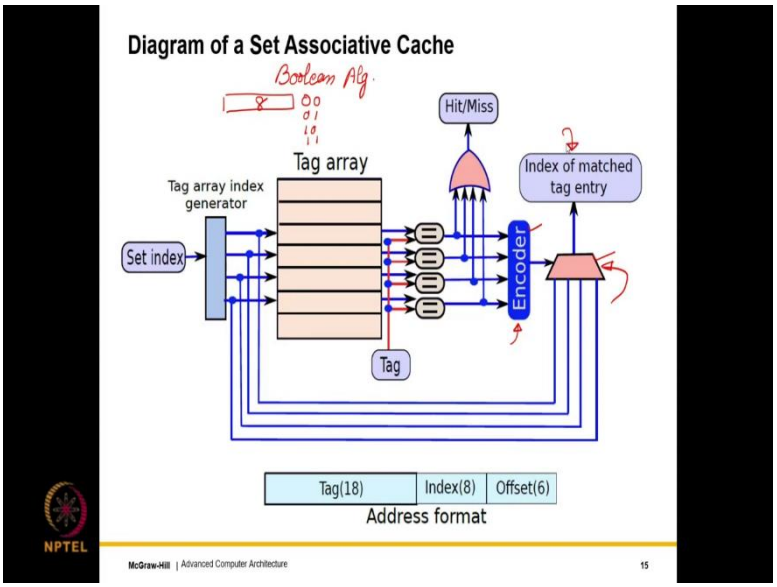
So, since there are 4 of them, we can have a small OR gate. A small OR gate can be used to compute a hit or a miss and if there is a hit, well that is good. So, what we do is then we use an encoder. So, the encoder tells us which one of these lines actually had a hit.

So, we can number them 0, 1, 2 and 3 which one of them had a hit and simultaneously, the four addresses of the lines are being sent to a multiplexer. So, the output of the encoder is going to the multiplexes that will help us choose what is the final index of the matched entry.

For example, if this is the final index of the matched entry, its number is 1 that is what the encoder will output as 1 and then basically, this input over here will be chosen and this address will be the final index of the matched tag entry, if there is a hit.

Now, the important thing is the way that I have described, it should be understandable to all, if it is not, here is what you need to do. What you need to do is you need to understand the following things and that will help you understand this slide better.

(Refer Slide Time: 43:14)



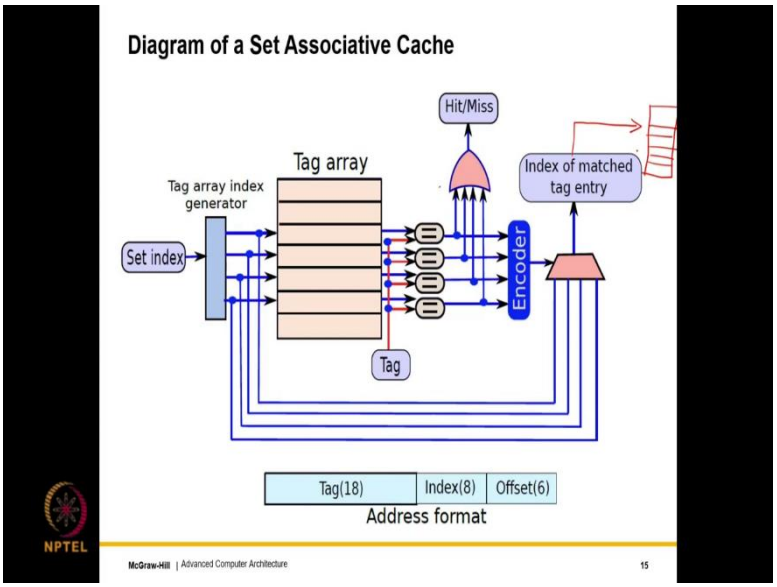
The first is you need to understand how an encoder works. Any book on digital logic will have this. Second, you need to understand that how do you generate four addresses from the set index. Well, that is easy; we take the set index like this. These are the 8 bits.

We can just generate four addresses 0 0 1 1; just concatenate with it 1 0 and 1 1. Now, the question is are you able to understand what I said or not? If you are not able to understand, then you need to understand more of Boolean algebra. So, pick up a book and go over Boolean algebra; the rest should be straightforward.

And are you able to understand the connection between this encoder and this multiplexer. For most students of electronics and electrical engineering, this should not be hard. If you are finding that it to be difficult, then you need to understand how a multiplexer works, then you can clearly appreciate why the final output will be the index of the matched tag entry.

Let me clear off the ink and explain the basic idea at the highest level. So, at the highest level, we have a set. We read four tags in the set and compare it with the tag part of the address. So, if there is a hit, it gives us the index of the matched entry.

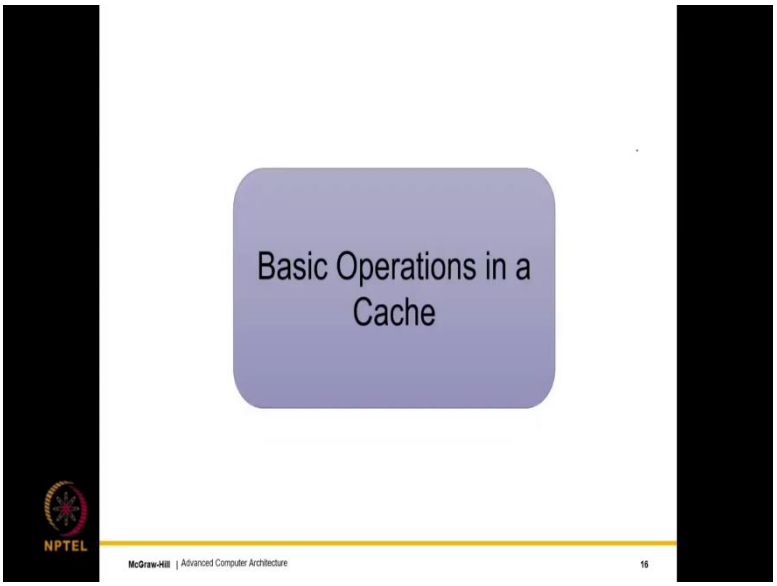
(Refer Slide Time: 44:42)



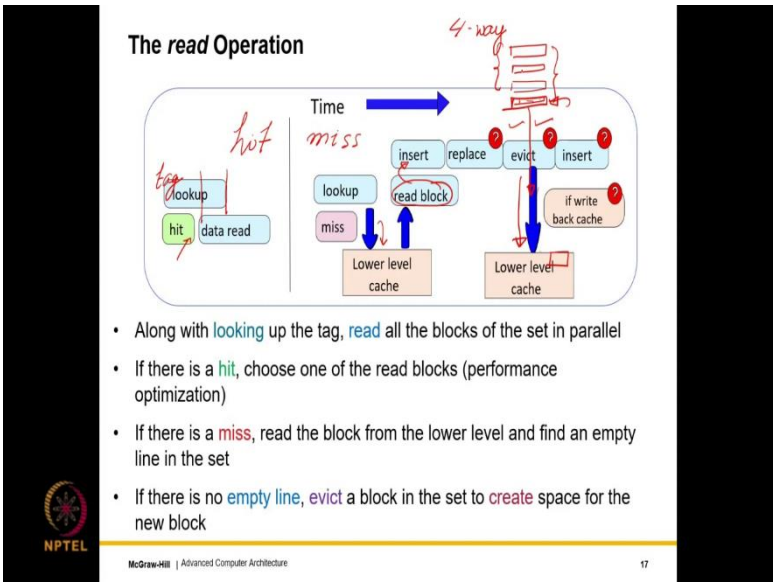
So, what I am not showing in this figure is that we go and access the data array with that index. We can use the same trick over here which is that we can read the four tags in parallel. If we read the four tags in parallel along with reading the data array, along with reading the four corresponding data array entries, so this is a performance enhancing trick.

In the sense that we do not know which entry will match if at all, but at least the performance enhancing trick over here is that the tags are being read, the tag and the data array entries all of them are being read in parallel and this is going to give us an advantage in terms of time. Of course, not in terms of power because we are doing wasted work. But at least for speeding up reads, this is going to be a very useful technique.

(Refer Slide Time: 45:36)



(Refer Slide Time: 45:43)



Now, let us look at basic operations in a cache. So, let us look at two scenarios, when there is a hit and there is a miss. So, what we can do? So, what we have discussed is that for the sake of a data read, we can do the lookup which is the tag lookup basically.

And also, read the data blocks that the blocks in the data array, both of them can be done in parallel. So, there can be an overlap between them. So, this as you can see for a hit, this is a performance enhancing exercise because there is clearly an overlap in terms of time. Of course, if there is a hit, we choose one of the red blocks that is a

performance optimization. If there is a miss, well so this is scenario 1 for a hit; if there is a miss, this is what we do.

So, what we do is that we do a lookup, if there is a miss. We send the request to the lower level. Again, in the lower level, we have the same access protocol. If there is a hit, we supply the data; otherwise we go down. So, finally, when we get the data, we read the block.

And we also try to insert it in the cache. So, how do we insert it? Well, we look at the four entries in a set; if it is a four way associative cache. So, in a set associative cache, if let us say there are four tags per set, it is called a four way associative cache. If there are k tags per set, it is called a k way associative cache. So, if any one of these is empty, then well we can write the block to that entry; otherwise, we need to pretty much throw out one of these blocks which is called eviction.

So, we first find a replacement candidate which as we shall see there are various heuristics. We never want to replace frequently used blocks. So, this is not something that we would like to do. So, we find a candidate for replacement. So, the reason I have a question mark here is basically because there is a possibility that this essentially captures the possibility that one no entry in the set is free no; all the entries are full.

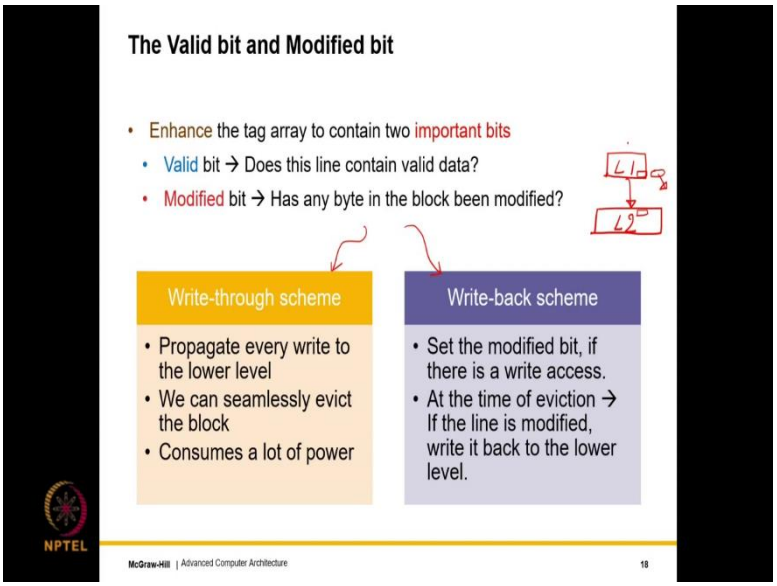
So, if all the entry entries are not empty, sorry then what we need to do is that we need to find a candidate for replacement, evict it which means write it back to the lower level and I will discuss what is the right back cache.

But the quick idea over here is that if it has been written to if it has been modified, then only there is a need to actually write the contents to the lower level. However, if it has not been modified, there is no need to write it because anyway there is a copy at the lower level cache.

So, since there is a copy and the copies are guaranteed to be the same because there has been no write at the upper level, we need not do a write and we will see there is there is a variant of this scheme as well in the next slide. But this is the broad idea and after creating space, so let us say that all the four entries were not empty. So, after we create space in the sense, if we let us say evict this block; then, space is created and the new block can be inserted over here.

So, this process of evicting a block creating space and inserting, this is of course, optional subject to the fact that we do not have an empty entry in the set. So, the flowchart for a miss is clearly more elaborate in the sense and in the sense, you have a replace and evict operation. So, let us take a look at this in some more detail.

(Refer Slide Time: 49:47)



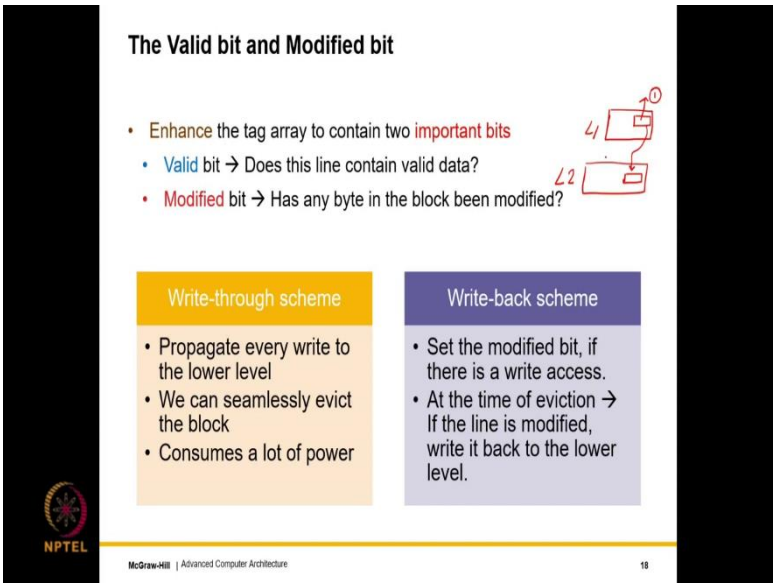
So, along with the tag in the tag array along with the tag, we have two additional bits; they are the valid bit and the modified bit. So, the valid bit indicates if the line contains valid data or not or in a sense is it empty or full, this is the job of the valid bit. The modified bit on the other hand says that after the block has come into the cache has any byte of the block been modified. So, it captures this fact that whether this block any part of it has been written to or not.

So, in the context of the modified bit, in the context of writing to a block, there are two schemes; a write-through scheme and a write-back scheme. So, in the write-through scheme, what we do is that we propagate every write to the lower level. See if you let us say we have the L1 cache and we have the L2 cache, whenever we have a write, every single write is propagated to the lower level. So, this of course, is expensive in terms of messages, this is expensive in terms of power.

But the advantage is that if we want to throw out any block from the cache, we just throw it out there is no reason to write it to the lower level at that point. The reason is because in any case the two replicas of the block at the upper and lower level are

synchronized, they are the same. This is power expensive; expensive in terms of power, but nevertheless, eviction is simple. The more performance optimal scheme is actually the write-back scheme.

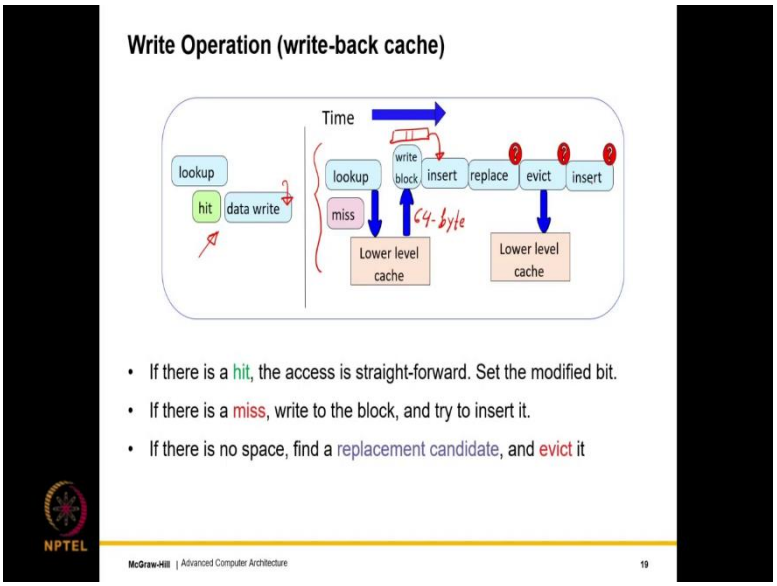
(Refer Slide Time: 51:39)



See here what we do is at the upper level, let us say at the L1 level; when we write to a block, we do not propagate the write; instead we set the modified bit to one. So, once we set the modified bit to one, we remember it, so by default it is 0.

At the time of eviction, if this line is supposed to be thrown out, then what we do is we check if the line is modified; only if it is modified, we write it back to the lower level; otherwise, there is no need to write it back because the replicas are the same. So, the write-back scheme does introduce some amount of additional state. The additional state is limited to 1 bit which is the modified bit, 1 bit per block basically; but it helps us reduce the number of messages and save power.

(Refer Slide Time: 52:34)



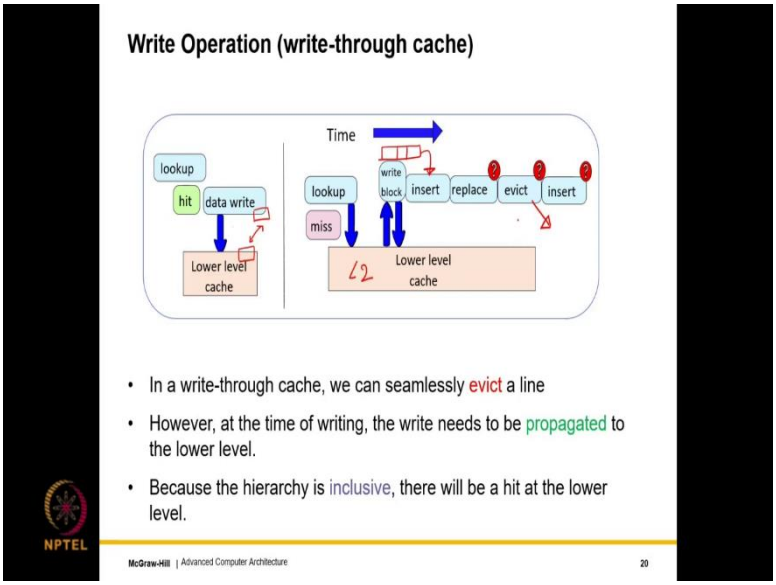
So, now let us look at the write-back cache. Well, the write-back cache for a write operation would look like this that we will do a lookup. So, we cannot have the same optimization that we had for a read.

So, in the sense that even if it is a four way set associative cache, we cannot write to or four all four blocks. So, we need to find which index there is a hit, we access that entry and we do a data write that is the broad idea. And once this is done, the job, the write is done. Here also if there is a miss, there is more work.

So, as we have argued in the past, even if we want to write it to a single byte, we still need to fetch the entire block. So, there is a miss we want to write to let us say four bytes, we still need to fetch the entire 64 byte block and this needs to be inserted in the cache and then of course.

So, we write to the block, we let us say put it in a temporary buffer, we write to those four bytes and the entire block needs to be inserted into the cache. If you have space for it, it is great; otherwise we replace, we evict and we insert.

(Refer Slide Time: 54:01)



And this is the standard way in which we do things. In comparison in a write-through cache, we do things slightly differently. See if we just compare the hit when there is a cache hit, here we do not do anything; but in a write-through cache whenever there is a cache hit, every single write is propagated to the lower level.

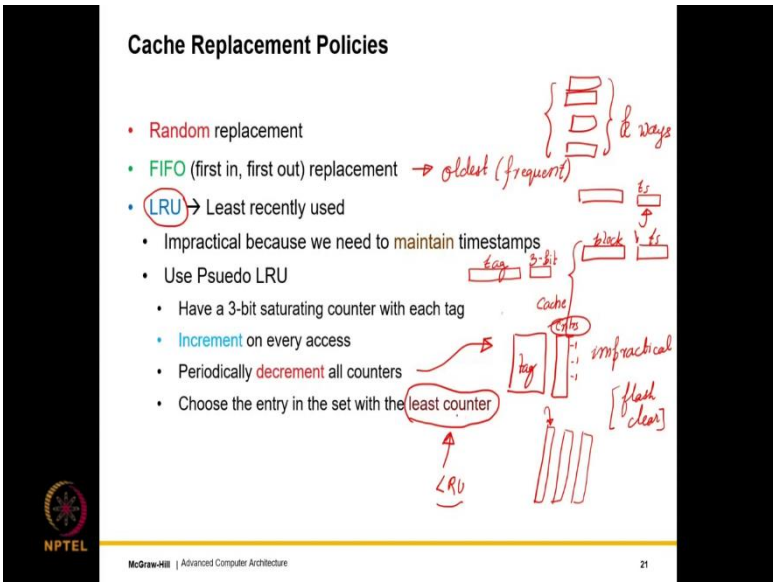
This ensures that the replica of the block which is there at the lower level and the replica which is there at the upper level, they always remain the same and furthermore, for the benefit of this is that when we are actually servicing a miss, we can save some work; let us see how. So, let us assume that there is a mess at the upper level. So, we access the lower level.

So, let us say there is a miss in L1 access L2, then what we do is since it is a write-through cache, we perform a write in the L2 cache as well and we fetch the block, the entire contents of the block and we write to it and then, we try to insert it. If you are not able to find a free line, we find a candidate for replacement, we will see how and then, we just seamlessly evict the line.

So, we evict the block. So, then, so the block the reason, it can be seamlessly evicted is because the copy of the block in the upper level and the lower level is the same and that is why it just can be thrown out; seamlessly evicted. Once space is created, we can insert the block, the new block into the L1 cache.

So, this does take into account the fact that the hierarchy is inclusive. In the sense that if there is any block which is in let us say the L1 cache, there will always be an entry for it in the L2 cache. In the case of a write-through cache L1 and L2 will always have the same copy, not necessarily true in a write-back cache.

(Refer Slide Time: 56:12)



So, what are the replacement policies? Well, the replacement policy would be like this. Let us say it is a k way set associative cache say in every set, there will be these k ways. So, one is random. So, random replacement in practice is not that bad, but it is not optimal.

So, it is not the best clearly; one is first in first out. So, in first in first out, the way that we will implement this is that we need to associate some sort of a timestamp associated with each line. So, the timestamp will record the fact that when the line was brought in. So, this does increase the storage overhead and this does increase this; the bookkeeping space that is required.

But then, what will happen is anytime we need to evict, we need to evict the oldest one. The reason this does not work well in practice is because the oldest line in a set might also be the one that is the most frequently accessed. So, the oldest one might be the most frequent one.

So, we are not capturing this. This is why FIFO is not used that much in practice. What actually works the best and also in many theoretical settings, it can be proven that this is indeed optimal is the LRU, the least recently used which means you define a window of time, within that a given line should be the most reason should be the least recently used.

So, ideally, theoretically, LRU is impractical. Why? Because here, again you need to maintain timestamps and whenever you access a block, you need to update its timestamps; you need to say that look I accessed it now. Doing this in software is easy for other for higher level programs, but doing this at the level of hardware and that too at the level of a cache is impractical.

So, you cannot you know FIFO at least was slightly practical because you just create the timestamp once which is while inserting the block; but in this case every time, you access it, you will have to update the timestamp which is not practical. So, what instead is done is that we use Pseudo LRU.

So, along with each tag, we have a 3-bit saturating counter or a 4-bit I mean a k-bit saturating counter. So, I say it does not matter, I mean 2-bit, 3-bit, 4-bits does not matter. On every axis, we increment the saturating counter. So, this in the sense shows the recency, shows the fact that something that is more recently used will have a higher value of the counter; but then what will happen is that over time, ultimately all the counters will saturate.

This is not what we want right. Since we do not want this, what we do is that we periodically decrement all the counters in the entire cache right. So, well, whether we decrement or we set them to 0, well we will see. But let us assume that from a simplistic point of view, if this is the tag array, see you have all the tags over here, nicely stored and then you have all these counters.

So, periodically, we just decrement them by one. So, decrementing in general is a hard operation, but what we can do is that we can actually store these counters as 3. So, let us assume it is a 3-bit saturating counters. We can store them as 3 arrays and what we can do is periodically maybe we can just set one of these entire arrays to 0; setting a full array to 0 is comparatively a much easier operation. So, you can see in the book, there is a mechanism called flash clear.

So, periodically, what we can do is that you know we can set one of these arrays to 0. So, this will essentially ensure that all of them you know to a certain extent are getting reduced right and this is faster than decrementing. Then, what will happen is that all of them will get reduced and then, gradually they will start increasing.

So, with this method which is called Pseudo LRU, you will see that the entries that are the most recently used, they will have higher values of these counters and the counters are at least recently used will have lower values.

So, we choose the entry in the set with the least value of the counter and so, this is this means that because of temporal locality, you can say with high confidence that most likely in the future also this line will not be accessed. So, you will choose this as a target for replacement and you will evict it. In this place of that the new block can be brought in.