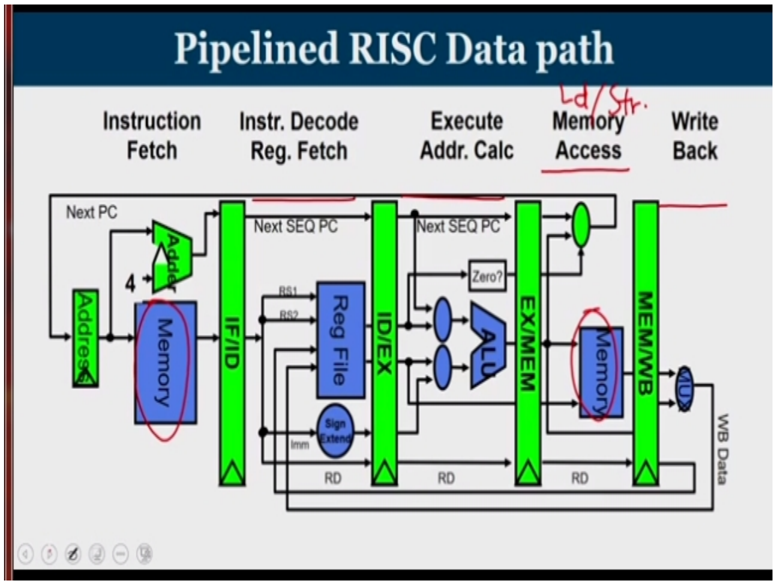
Advanced Computer Architecture
Prof. Dr. John Jose
Assistant Professor
Department of Computer Science & Engineering
Indian Institute of Technology-Guwahati

Lecture-18
Introduction to Cache Memory

Hello welcome everybody, today's our lecture number 13 we are moving onto cache memory concepts. Over the last 12 lectures our exploration was on how processor works especially in the context of instruction pipelining. We have seen a couple of datas level parallelism friendly architectures as well in lecture number 12. Now we move onto yet another aspect of computer architecture that is all about storage.

We start from on-chip storage that is a very first thing moving onto the very next level that is going to be your off-chip storage in terms of main memory and then hard disk. Let us try to understand what is the concept of cache memory and it is fundamental principles in this lecture.

(Refer Slide Time: 01:21)



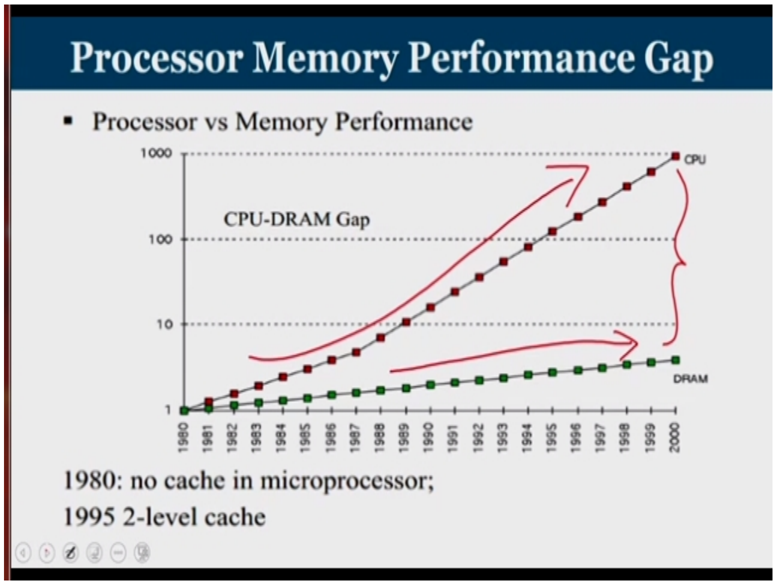
Pipelined RISC Data path

This slide is about pipelined RISC data path, we have started our discussion in this course with a 5 stage instruction pipeline consisting of instruction fetch, instruction decode, execution, memory access and write back stages. Now we have trying to understand what is the role of memory in this context we can see that here we have a memory component and here also we have a memory component.

The first memory component is accessed by every instruction and that is called fetching of an instruction from the memory and we have seen that to take care of structural hazards we are having separate memory for instruction as well as data. So instruction fetch uses memory and memory access stage also when you have load or store instruction also this we are going to access memory and that is called your data memory.

All other 3 stages, stage number 2, stage number 3 and stage number 5 are completely internal to processor and the control logic and the register file together get the things done as for the particular instruction in decode, execute and write back stages.

(Refer Slide Time: 02:36)



Processor Memory Performance Gap

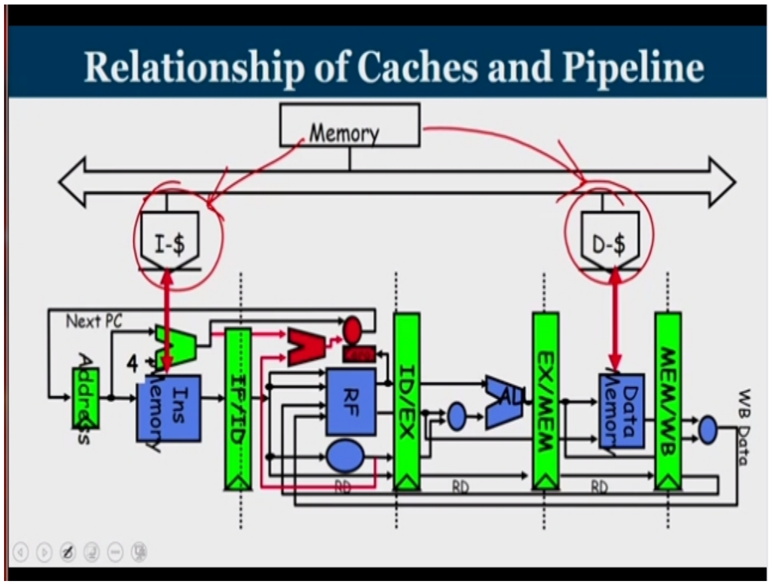
- Processor vs Memory Performance

1980: no cache in microprocessor;

1995 2-level cache

This graph shows over the years what is the trend in performance as for as CPU is concern, so we can see that there is a massive improvement in the performance of CPU. And we could see that there is improvement in terms DRAM memory as well. But if you can see that there exist a huge gap the rate at which the performance growth happens at the CPU side as well as on the DRAM side.

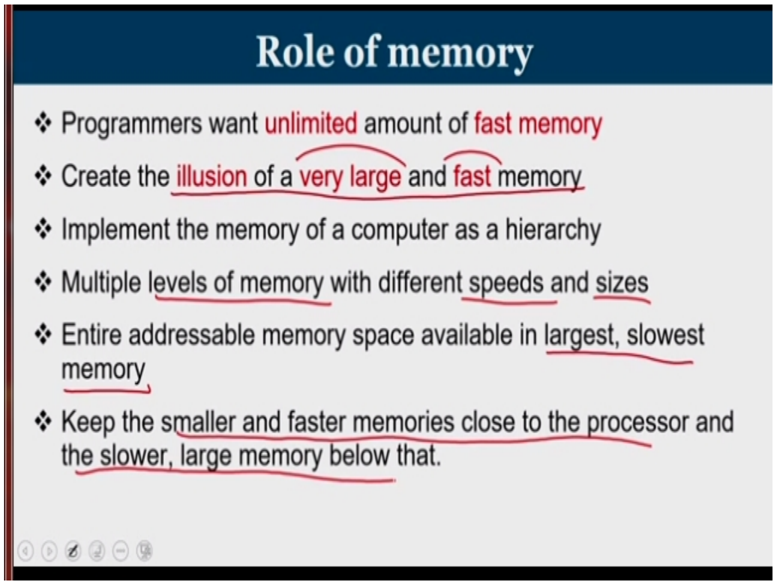
(Refer Slide Time: 03:03)



Relationship of Caches and Pipeline

Now coming to the relationship of cache and pipeline, we have seen that in a 5 stage pipeline the instruction is been accessed by cache and your data portion whenever you wanted to access your load and store, there is also been interacting with the cache and cache is been supplied from the main memory. So the next couple of lectures our focus of attention is earn is memory aspect and the remaining.

(Refer Slide Time: 03:32)



Role of memory

- ❖ Programmers want unlimited amount of fast memory
- ❖ Create the illusion of a very large and fast memory
- ❖ Implement the memory of a computer as a hierarchy
- ❖ Multiple levels of memory with different speeds and sizes
- ❖ Entire addressable memory space available in largest, slowest memory
- ❖ Keep the smaller and faster memories close to the processor and the slower, large memory below that.

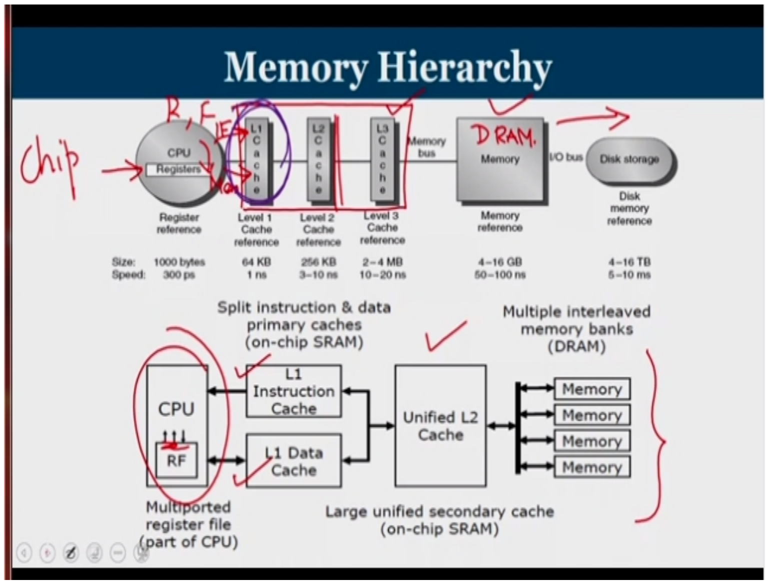
Let us try to understand what is the role of memory in this context. Programmers always want unlimited amount of fast and large memories. So what we can do is we can create an illusion of making your memory very fast as well as your memory very large. It is impossible to make a very large memory at the same time it is fast but this illusion can be created when you organize your memory as a hierarchy, so we how to understand want do you mean by a hierarchy.

So hierarchy means we how different levels of memory, different types of memory each memory has it is own property in terms of speed, in terms of total storage, in terms of the number of reads or writes that you can do. Considering all these aspects memories of different property are stacked one after another and then processor is trying to access these memories based upon this requirement.

So when you create such a hierarchy of memory we can provide a very large memory or we can in turn provide an illusion that your memory is very large as the same time it is going to be relatively fast as well. So that it can mitigate with the speed at which an instruction pipeline works. So like what I have mentioned when you go for multiple levels of memory there can be differences in speed and the size of the memory.

And your entire addressable memory space is typically kept in the largest at the same time slowest memory. Keep the smaller and faster memories closer to the processor, so this is the idea, can you keep your smaller and faster memory closer to the processor and your slower and large memory just after that.

(Refer Slide Time: 05:16)



So this is the typical way how you look at, if you look at the CPU it has internal register that is what have seen, we have integer registers and we have floating point registers which are marked as R for integer register and F for floating point register when we were going through the topics of instruction pipeline. So these registers are very fast, CPU can access them generally wherever in our programs we try to keep our operands in the register possible.

And we can see that is not possible to have huge set of registers, generally the registers are in the order of few bytes and the accessing to registers is in the order of picoseconds. And then we can look into we have multiple levels of cache memories; this is what we are going to learn today. And these cache memories are relatively very fast and the L1 cache which is the closest to the processor will be in the order of kilobytes and accessing them would be at the same order as that of accessing any other stage of an instruction pipeline.

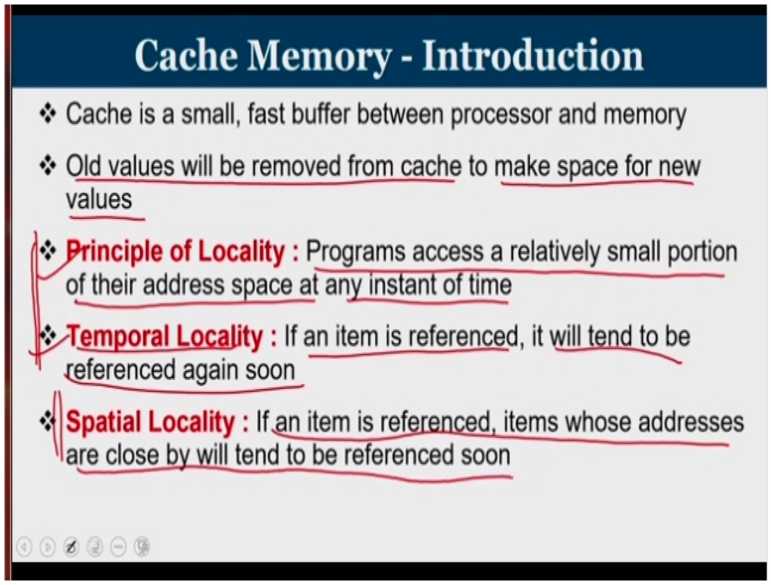
So it is during your instruction fetch stage you are going to take something from L1 cache and your mem stage also you are going to access something from the L1 cache. So your instruction pipeline is interacting with this L1 cache alone this is the cache that we are talking about where instruction cache is directly interacting. And then we have a little bit larger next level cache that is called your L2 which will take close to 10 nanosecond or even slightly more than that.

It will be roughly in the order of 256kb or 512kb and sometimes it can be up to 1mb as well. And some processors of the third level of cache that is the L3 cache, so whatever be the cache we have the terminology which is called LLC- last level cache, last level cache it can be either L2 sometimes if that is a last cache hierarchy that you have on the chip or it is can be sometimes L3 cache as well.

So the whole thing is inside your processor chip, this is your chip, now off chip, the first level of off chip memory is your main memory, it is also known as DRAM now a days. Because the based on the technology that is been used dynamic random access memory and then we have a hard disk that is a permanent storage. We can see there once you come to off chip then your memory capacity is in the order of gigabytes and terabytes and access speed go into the milliseconds range.

Looking at a different perspective of how this whole memory hierarchy looks like, I wanted to draw your attention to this, this is your CPU which has registers through which I can read and write. The registers as read and write ports, then you have separate I and D cache which is your L1 cache and then you have L2 cache it is a unified cache and then we have different chunks of main memory together we call it as the main memory unit.

(Refer Slide Time: 08:10)



Cache Memory - Introduction

- Cache is a small, fast buffer between processor and memory
- Old values will be removed from cache to make space for new values
- **Principle of Locality :** Programs access a relatively small portion of their address space at any instant of time
- **Temporal Locality :** If an item is referenced, it will tend to be referenced again soon
- **Spatial Locality :** If an item is referenced, items whose addresses are close by will tend to be referenced soon

So we have seen what is memory hierarchy all about starting from registers all the way L1, L2, L3 caches and then we go to main memory and then finally the disk. Let us now specifically focus on what is cache memory technique, what are the fundamental principles by which cache memory operates or how can you design a good cache memory. That are going to be the next focus points, a bit of introduction with respect to cache memory.

Cache is a small, fast buffer between processor and the memory and then what we do is old values will be removed from cache to make space for new values that is a whole idea of cache. And it is done with 2 techniques driven by principle of locality of reference, so what do you mean by principle of locality of reference. Programs access relatively small portion of their address space at any instant of time, what about temporal locality.

If an item is referenced it will tend to be a referenced again soon, that is basically the idea of temporal locality. And then we have spatial locality, if an item referenced, the item whose addresses are close by will tend to be referenced soon. Let us try to understand the principle of locality of reference in the context of a storage system. Principle of locality of reference has 2 sub divisions in it, the first one is called temporal locality.

Let us try to understand what is this temporal locality all about, assume that I am going to read 1 paragraph in a textbook. Now if there is a very high probability that I am going to read the same paragraph again in the near future I call it as temporal locality. Something that is access now, it is going to be re-accessed again, that is called temporal locality in terms of program context. Let me define like this.

If I am going to access a location L at time T then the probability of accessing the same location L at a future time T + delta T is very high, meaning. If I access a particular instruction or a data now there is very high possibility that I am going to access it again. Before relating this concept to cache memory, let me draw your attention to the second component which is called spatial locality. If I access a location now, let us say location L at time T then the probability of accessing location L + delta L that is the nearby locations of L, in time T + delta T is very high.

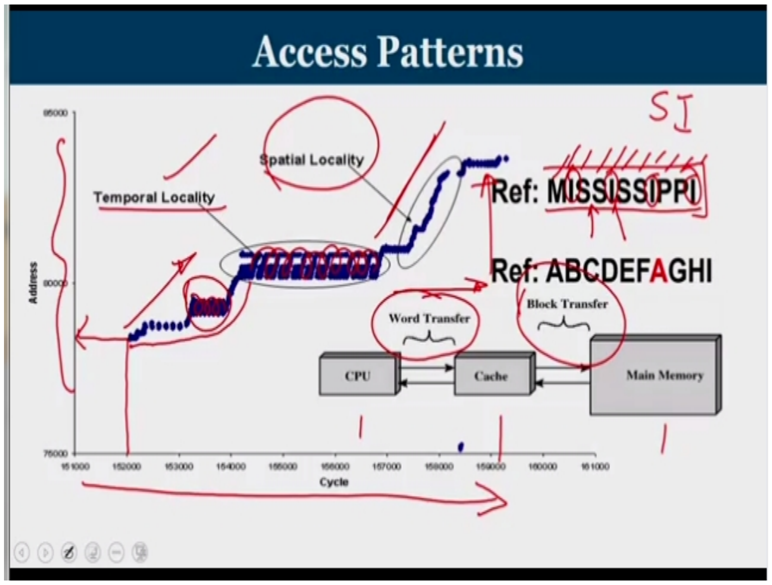
So if I access an instruction or a data then the possibility of the adjacent instructions or adjacent data locations in the very near future, possibility of accessing these locations in the near future is very high. Now let me put together temporal locality and spatial locality when CPU is accessing a memory or CPU is requesting for an address if it is not there let us say in your cache memory you go to the next level of memory bring it why you bring it.

Because there is a possibility that the same address will be requested by the processor in the near future temporal locality. So when there is high possibility of temporal locality even though in my current reference I do not get it in my fast cache memory, I go to next level of memory keep it in cache why I keep it in cache because my future access to the same address it can be supplied from the cache, that is how I make use of temporal locality.

The how I am going to make use of spatial locality, when I miss some location now, I how to understand that as per spatial locality, the nearby addresses will be requested in the future. So rather than bring only the requested word let me bring the adjacent words also such that where I bring something when I bring something like a block or a bulk of data my future request to adjacent locations can be serviced.

So in the concept of cache memory when CPU is requesting for an address I look in my cache if it is there if so I will supply it, if it is not there I go to the next level of memory bring a chunk of data and keep it in cache such that any request to the same address or nearby address in the future can be supplied from cache a faster storage. That is the whole idea of principle of locality of references.

(Refer Slide Time: 13:01)



Look at this graph on the x axis we have the clock cycles and the y axis we have the address locations. So let us say my program is relatively spread across address 75000 to 85000 these are the locations in the main memory where a particular program is currently accessing. Now as time progresses that is called x axis as time progresses, let us say this particular time this is the address somewhere around 78000 that is address where the program is currently working on.

Now as the time progress we can see that the address locations changes, going up means it is going to higher address. And when you can see that in this region the same address is being revisited you come down to the same address. Similarly in this region also we can see that the same address is being repeated in the near future that refers to temporal locality if the address is been re-referenced the near future we call it as temporal locality.

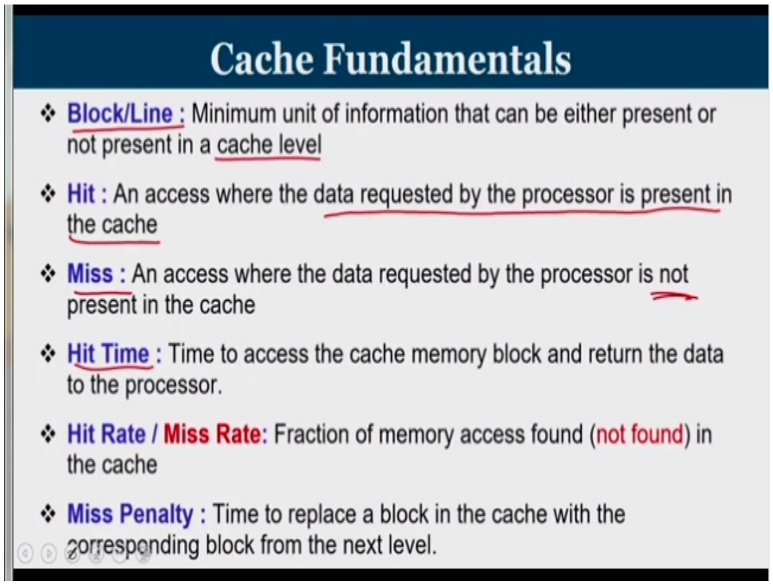
And if you look at this range we can see that as time progresses adjacent addresses are been accessed and that is what is known as spatial locality. Let me consider 2 examples, let us say 2 references we imagine that each of the English alphabet represents an address. So if these are the set of addresses that the processor is demanding MISSISSIPPI we can find that there are lot of S, lot of I that is being repeated, that is essentially what we are discussing right now it is called temporal locality.

The address S and I are frequently re-referenced, whereas if you look at this string you can see that if every address correspond to a location and A and B they maybe nearby locations we can see that I am going to repeat the adjacent addresses, I am going to access the adjacent addresses only, that is called a whole idea of spatial locality. So when I look into your memory concept as we have mentioned as discussed we have processor a main memory which is really big.

And then we have a cache which is lying in between your processor and main memory. As on when processor request for words it contact cache and cache is going to supply them. So the unit of transfer between cache and CPU is words. When cache misses something it goes to main memory and bring a bulk of data not only one word. Because a spatial locality will bring a larger unit of data that is what is known as a block.

So please remember this fact processor, contact cache first if the requested word is there supply the requested words. So the unit of transfer between cache and a processor is always words and a unit of transfer between cache and main memory is always in terms of big block of data.

(Refer Slide Time: 16:15)



Cache Fundamentals

- **Block/Line :** Minimum unit of information that can be either present or not present in a cache level
- **Hit :** An access where the data requested by the processor is present in the cache
- **Miss :** An access where the data requested by the processor is not present in the cache
- **Hit Time :** Time to access the cache memory block and return the data to the processor.
- **Hit Rate / Miss Rate:** Fraction of memory access found (not found) in the cache
- **Miss Penalty :** Time to replace a block in the cache with the corresponding block from the next level.

So move to fundamental concepts that we see in cache memory is block or line both are same, it is a minimum unit of information that can be either present or not present in a cache. So you bring a block of data from main memory into cache or bring a line of data from main memory into cache. So a block is either present or not present in the cache that scenario you cannot have part of a block that is being present in the cache.

Then we will move onto hit, hit is a scenario where in an access to the data requested by the processor is there in the cache. If something is present in the cache we call it as a hit, if whatever is requested by the processor is not present in the cache is not present then it is known as a miss. And the time to access a cache memory block and return the data to the processor if there is a hit it is known as hit time.

So hit time means when the processor is giving the address from that point when will you get the word from the cache. So given address there is a process by which you come to know whether it is present in the cache or not. If it is present locate the corresponding word and then transfer this particular word to the processor and that much time is known as hit time. And then we have hit rate, the same time we can define miss rate as well, it is a fraction access that is found or not found in the cache.

Now let us try to understand what is parameter of hit rate, let us say I am going to access cache 100 times out of it if I am able to get the data if there is a hit for all my accesses. All my access then I call hit rate as 1, if I am getting hit only in 90 out of the 100 time that I am visiting cache then it is called hit rate of 0.9,

$$90 / 100 = 0.9$$

and miss rate is

$$10 / 100 = 0.1$$

So, always

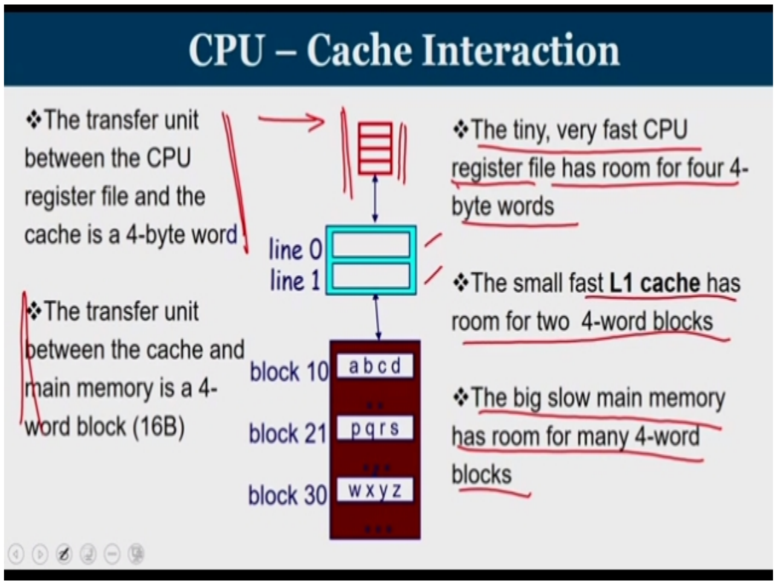
$$\text{hit rate} + \text{miss rate} = 1$$

Ideally we expect caches to how hit rate as close as possible to 1 and hit rate cannot be larger than 1 if every access is present in the cache then it is called a hit rate of 1. Then the last term is called miss penalty it is a time to replace a block in the cache with the corresponding block from the next level. So what is miss penalty, miss penalty means the moments you all your access in the cache may not be a hit at times you are get miss also.

So once it is a miss that time onwards you go to the next level of memory bring a block of data keep it in the appropriate place in the cache and then supply the word. So the moment cache miss occurs from that time onwards until the missed word is ready in the cache that is called miss penalty. The additional cycles required in processing a miss that is called the term miss penalty.

So we have learned about what is a block or a line, hit, miss, hit rate, miss rate, hit time and miss penalty. So these are some of the very common terms that have been use in the context of cache memories.

(Refer Slide Time: 19:31)



CPU – Cache Interaction

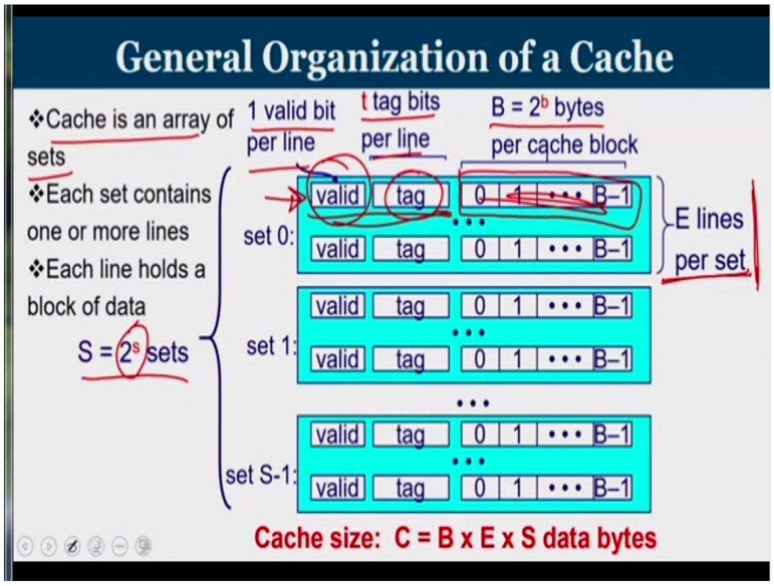
- ❖The transfer unit between the CPU register file and the cache is a 4-byte word
- ❖The transfer unit between the cache and main memory is a 4-word block (16B)
- ❖The tiny, very fast CPU register file has room for four 4-byte words
- ❖The small fast **L1 cache** has room for two 4-word blocks
- ❖The big slow main memory has room for many 4-word blocks

Look at this slide; here we have a different interaction schematic of processor cache and main memory. So what is given here is your processor, so processor is a tiny fast CPU the CPU through registers. So processor the tiny very fast CPU registers has room for only very few words. In this context I am taking four 4-byte words. So each of this registers can accommodate 4 bytes similarly I have 4 such registers.

Then when you come to L1 cache they are relatively small when compare to main memory but bigger than that of processor registers. Assume that you can accommodate 4 blocks, assume that you have 2 blocks each can accommodate 4 words each. So I have 4 registers each can accommodate 1 word I have 2 cache blocks each can accommodate 4 words. And then I am going to main memory, main memory is big but it is slow, it has room for many 4 word blocks like what you have.

And we have seen that the transfer unit between CPU, register file and the cache is 4 byte words whereas the transfer between cache and the main memory is 4 word blocks.

(Refer Slide Time: 20:59)



General Organization of a Cache

- Cache is an array of sets
- Each set contains one or more lines
- Each line holds a block of data

$S = 2^s$ sets

Cache size: C = B x E x S data bytes

Let us now look into the general organization of a cache, a cache memory is an array of sets that is what you can see that there these array of sets is been named as set 0, 1 all the way up to S-1. There you assume you have S sets and this S can always represent as a power of 2 let us say 2 power small s = capital S. Now each of these set has multiple lines in it or multiple blocks in it, so you how E lines per block and E is also typically a power of 2.

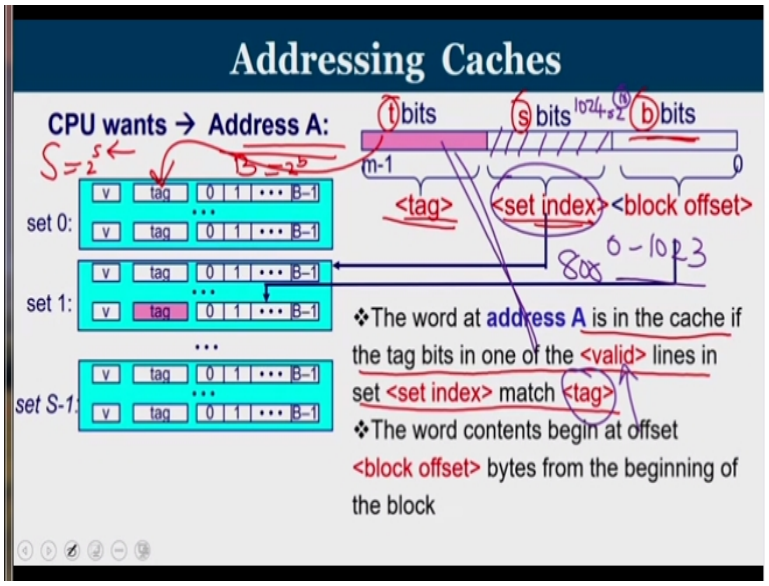
And each of these lines we have this is the data storage portion where I keep data that is been brought from main memory, you have few control section, control information which is called a valid line or valid bit, 1 valid bit is there per line and t tag bits is there for each cache line. So when you have each cache line this is the place where your data is located you have a valid bit and you have a set of tag bits.

Each cache line holds a block of data that is what we have seen and this block is also power of 2. So what is the total capacity of the cache, the capacity of the cache is defined as

$$C = B * E * S$$

where S is the number of sets, E is number of lines per set and B is the number of bytes in each of the line.

(Refer Slide Time: 22:33)



Addressing Caches

❖The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>

❖The word contents begin at offset <block offset> bytes from the beginning of the block

Now consider the context that CPU want an address A, the physical address of A. So once you get the physical address of A that physical address is divided into 3 things tag, set index and block offset. So with this number of bits that is there for block of set, for set index and tag it is easy for us correlate what happened. So S tells a number of set, so if you have capital S which is defined as 2 power small s this will tell you how many bits are there in set index.

And then your block is B bytes, which is

$$2 \wedge b$$

so this will tell you how many bits are there in the block offset region. And last portion is your tag, so whatever is balance that is called tag bits and these t bits are been stored here, so this is your tag area. Now what we do, the word at address A is in the cache if the tag bits in one of the valid lines in the set index matches with the tag.

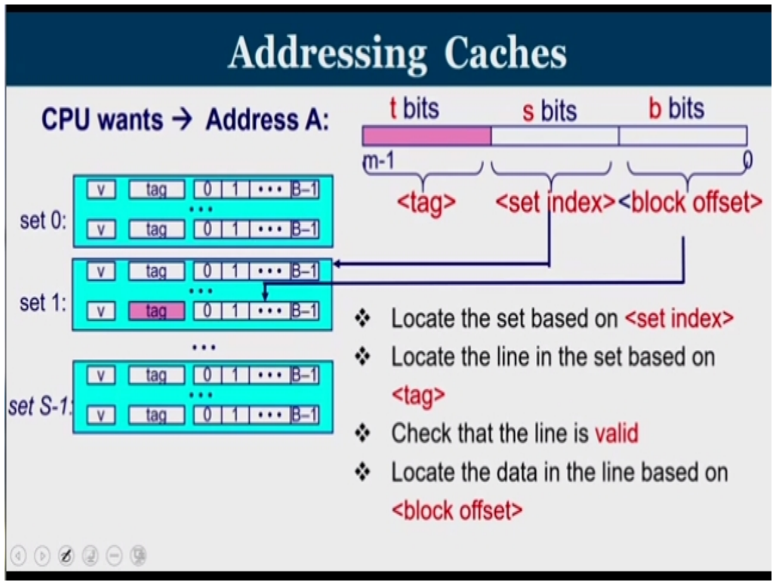
So what generally we do once you get an address we look into these portions of the address what is the index bit and how will you get it, it is equal to the number of bits there is use to find out the total number of sets in the given cache. If the total number of set is say 1024 then

$$1024 = 2 \wedge 10,$$

so 10 bits of the address is used for the set index portion go to that set index it will give a number in this context it will give an number ranging from 0 to 1023.

Let us say the number is 800, go to 800 to set check whether the valid bit is equal to 1, if the valid bit is equal to 1 check whether the tag is matching with this tag whatever is being proposed. If it is perfectly matching then that represents a hit, once you know it is a hit using the block offset portion, the word contents begin at offset bytes from the beginning of the block.

(Refer Slide Time: 24:53)



Addressing Caches

CPU wants → Address A:

t bits | s bits | b bits

m-1 ... 0

<tag> <set index> <block offset>

set 0: v tag 0 1 ••• B–1

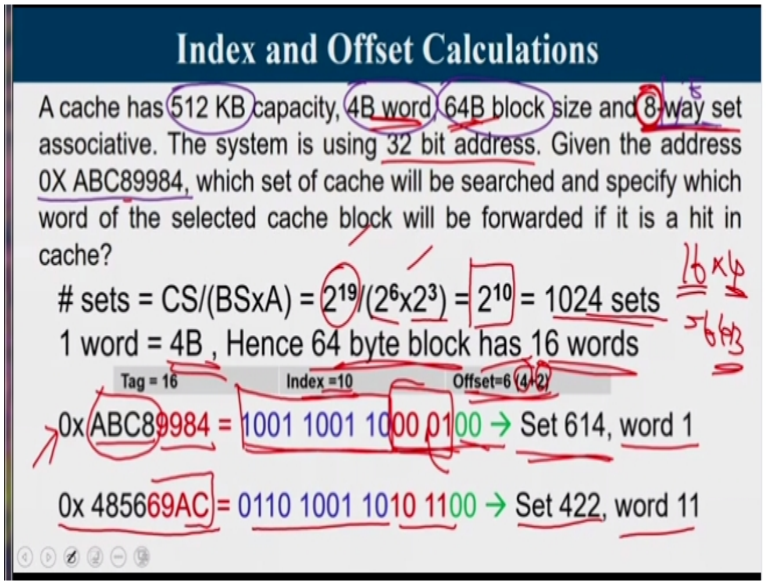
set 1: v tag 0 1 ••• B–1

set S-1: v tag 0 1 ••• B–1

- Locate the set based on <set index>
- Locate the line in the set based on <tag>
- Check that the line is valid
- Locate the data in the line based on <block offset>

So what do you do to address a cache, first you have to locate the set based on the set index portion of the address, locate the line in the set where there is a tag there is perfectly matching wherever there is a tag match check whether the valid bit is equal to 1 or not. If the valid bit is equal to 1 locate the data portion within the line based upon the block offset.

(Refer Slide Time: 25:20)



Index and Offset Calculations

A cache has 512 KB capacity, 4B word, 64B block size and 8-way set associative. The system is using 32 bit address. Given the address 0X ABC89984, which set of cache will be searched and specify which word of the selected cache block will be forwarded if it is a hit in cache?

sets = CS/(BSxA) = $2^{19}/(2^6 \times 2^3)$ = 2^{10} = 1024 sets

1 word = 4B , Hence 64 byte block has 16 words

Tag = 16 Index = 10 Offset=6 (4+2)

0x ABC89984 = 1001 1001 1000 0100 → Set 614, word 1

0x 485669AC = 0110 1001 1010 1100 → Set 422, word 11

16 x 4 = 64B

Let me now draw your attention to a small numerical problem which will help you in understanding about this cache, tag, index and offset concept. Consider a cache that has 512 kilobytes capacity, 4 byte word, 64 byte block size and 8 way set associative. The system is using 32 bit address, let us imagine that an address 0XABC89984 is been given by the processor, now which set of the cache will be searched to find out this address.

And specify which word of the selected cache block will be forwarded if it is a hit in the cache. So first we have to understand how many sets are there in the cache, so I have a cache let us say capacity of 512KB and a word is 4 byte and each of my block in my cache is 64 byte block and it is an 8 way associative cache. 8 way associative cache means your value of E there are 8 blocks in a given set.

The number of sets is divined by cache size divided by block size into associativity here you have a cache of 512 kilobytes that is called 2 power 19, this is called 512 kilobytes divided by block size, block size is 64 bytes that is 2 power 6 into associativity it is 8-way associative cache. So it is 2 power 3 the answer is 2 power 10, you have to understand both denominator and numerator should be in bytes, so one should not be in words.

$$(2 ^ 19) / ((2 ^ 6) * (2 ^ 3))$$

In this context this word property is not used we will use a bit later, so

$$2 ^ 10 = 1024$$

So this particular cache has 1024 sets, the 1 word is 4 bytes, so 64 byte block means I am keeping 16 words there. Because

$$16 * 4,$$

16 word each word is 4 byte together is my 64 byte that I am talking about. So I have 10 bits for the index and I have 6 bit for the offset out of the 6 the first 4 bit will tell you word number and the next 2 bit will tell you what is the byte within that word anyway last 2 bits is not relevant.

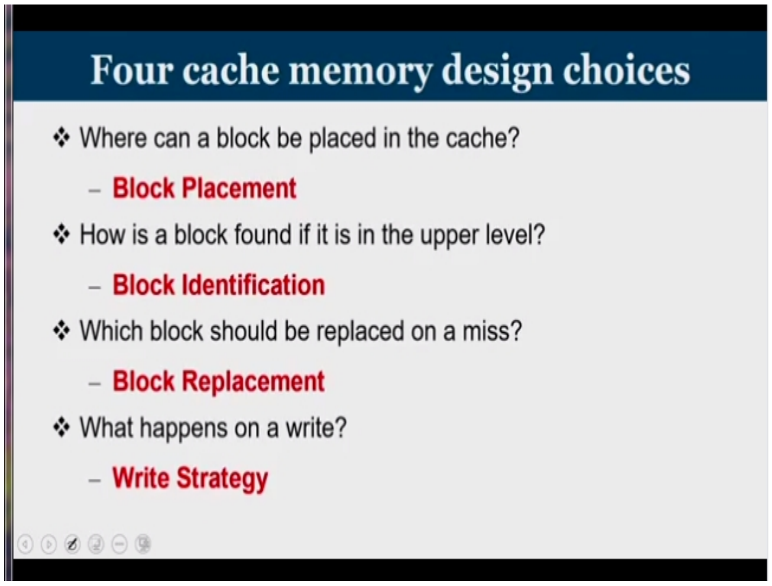
If it is a 32 bit address that is what it is been mentioned out of the 32 bit the middle 10 bits is for index, the last 6 bit is for offset and the most significant 16 bit used for tag. Now what we are asking is in this given address 0XABC89984 that is a address that is been given here, this address I have to split into 16, 10 and 2. So this black portion indicates is it a hexadecimal, so this black portion indicates it is a tag, the remaining 9984 that is a 16 bit value.

Since this is a hexadecimal representation I am going to write that 9 bit value, the first 10 bits shown in blue color indicate the set index. Then the 4 bit indicates the word number that is a red color is word number and the green color is byte within the word. So for this particular address if you take up the blue portion alone then the decimal value is 614, so given this address it is mapped to do set number 614.

Now in 614 there will be 8 lines that is why it is called an 8 way associative cache it is clearly mentioned. There are 8 different blocks we call it as 8 different ways in each of the way you see whether there is a tag match. So this is your tag, if these 16 bit is stored in one of those 8 lines then if you find a tag match it is called a hit. If it is a hit then extract word number 1, how you got word number 1 these 4 bits that is given in the red color that will tell you what is the word number.

There are 16 words possible arranging from 0000 all the way up to 1111, similarly one more example is given let us say if this is the address 0X485669AC is the address, I am writing only the red portion that is set index portion and offset. So the blue 10 bit will tell you what is the set number then the 4 bit after that will tell you what is the word number, so it set 422 and word is 11.

(Refer Slide Time: 30:00)



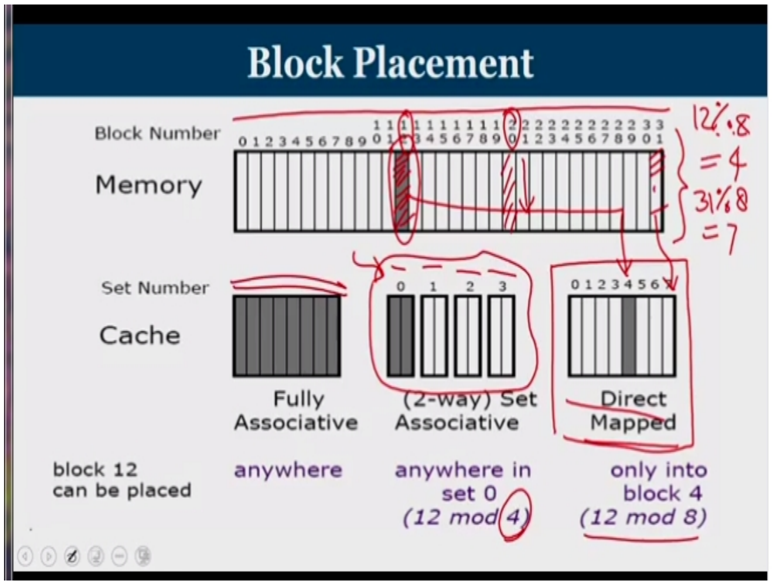
Four cache memory design choices

- ❖ Where can a block be placed in the cache?
  - Block Placement
- ❖ How is a block found if it is in the upper level?
  - Block Identification
- ❖ Which block should be replaced on a miss?
  - Block Replacement
- ❖ What happens on a write?
  - Write Strategy

While designing of cache memory there are basically 4 design choices, the first one where can we place a block in the cache and that is been defined by the block placement technology. And how a block is found if it is there in the cache or not that is called block identification. Third one if you come to know it is a miss then which of the block is to be replace, so that the missed word can be kept that is called block replacement technique.

And then when there is a write miss what will we do that is called write strategy. So designing of cache memory involves 4 different challenges one is placement of blocks, cache placement, block identification, block replacement and write strategy.

(Refer Slide Time: 30:44)



Block Placement

block 12 can be placed: anywhere (Fully Associative); anywhere in set 0 (12 mod 4) ((2-way) Set Associative); only into block 4 (12 mod 8) (Direct Mapped)

Let us now go to block placement scenario consider you have a main memory that has 32 blocks; block is nothing but continuous words. So if we have a main memory of 32 blocks ranging from 0 all the way up to 31. Imagine that you have a cache, we call this cache as direct mapped cache; let us say our discussion is in this particular block, block number 12. Any block in main memory is mapped to that block modular 8 because I am talking about a cache memory which has 8 sets and each set has only 1 line.

So I have 8 sets in my cache and each set has only one block so altogether I have 8 blocks. So this block, block number 12 where it is going into the cache that is been found out by

$$12 \bmod 8 = 4$$

So block 12 get mapped to set number 4 in the cache similarly block number 31 if I do then

$$31 \bmod 8 = 7$$

So this get mapped into 7, the problem here is let us imagine I am going to bring 20 also, so

$$20 \bmod 8 = 4,$$

$$12 \bmod 8 = 4$$

So now we are talking about 2 addresses block number 12 and block number 20 both are mapping into the same location 4. That means contents of block number 12 and contents of block number 20 in the main memory cannot co-exist in the cache. Such kind of caches where they are exist a unique location for every main memory block that is known as direct mapped cache.

So there exist a restriction where can we place a block on the other hand if there is no restriction I can keep 12 anywhere within the cache that is known as fully associative cache. Both of it is own pros and cons. Let me draw your attention into another mechanism by which it is being stored, you assume that the entire 8 blocks is divided into 4 sets and each set has 2 block each. Then by 12 get mapped into

$$12 \bmod 4,$$

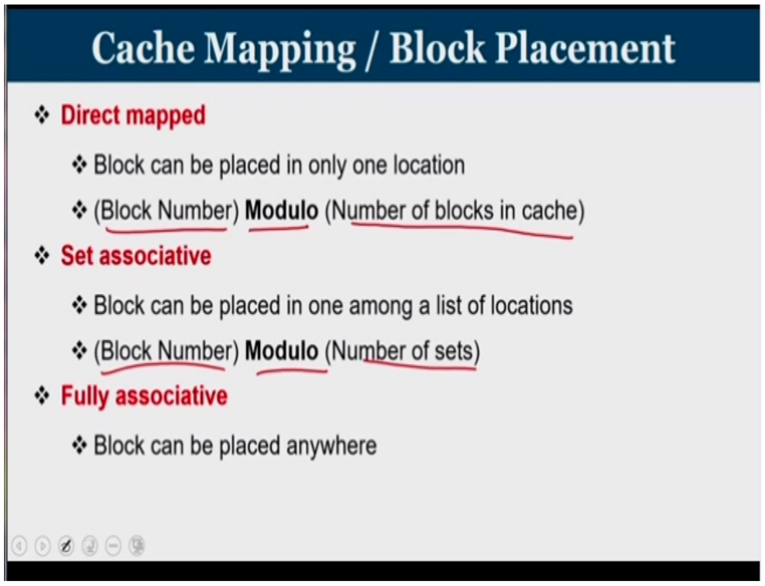
why 4 because I have only 4 sets, 0, 1, 2 and 3, I have 4 sets, so 12 mod 4 is 0.

So the contents of block number 12 will get mapped into set 0, the in set 0 there are 2 blocks either of the block you can decide. So it is a fusion of direct mapped cache as well as fully associative cache. This concept of different mapping technique in cache is very important let me draw an analogy what you generally use in real life. Consider the case that when you join in a college, let us say in the very first year you all have notebooks for each subject there is a notebook.

So whenever that corresponding teacher enters the classroom you are going to take that appropriate notebook let it be physics. Then you have a physics notebook, when the mechanics teacher come you go and take mechanics notebook. So every subject has a unique place that is what is known as a direct mapping concept. Now think of a case in certain times once you move up to the higher semesters the same notebook maybe used for 2 subjects one you start writing from the front.

Let us say data structures you write from the front and other course is databases you write from the other side. So both data structures and database the same notebook that is being used that is called 2 way associative notebook. Let us say when you move to final year you may use a same notebook for 4 subjects, so you define the first 10 pages it is for this subject next in pages like that.
So for all this 4 subject this particular notebook is used, for the next 4 subjects another notebook is used. Then with this 2 notebooks you can effectively handle the whole semester. Whereas if you could have used separate technique then you require 8 separate notebooks, that is called a 4 way associative cache. Similarly I can think of 8 way associative cache where 8 different addresses are mapped onto that or 8 different blocks are coexisting or 8 different subjects are coexisting inside a same book.

And when you do not have any restriction at all, for any subject you can write anywhere that is called a fully associative notebook. So I hope this example will give you little bit of clarity about how this concept of placement, block placement is introduced into the cache memories.
(Refer Slide Time: 35:43)

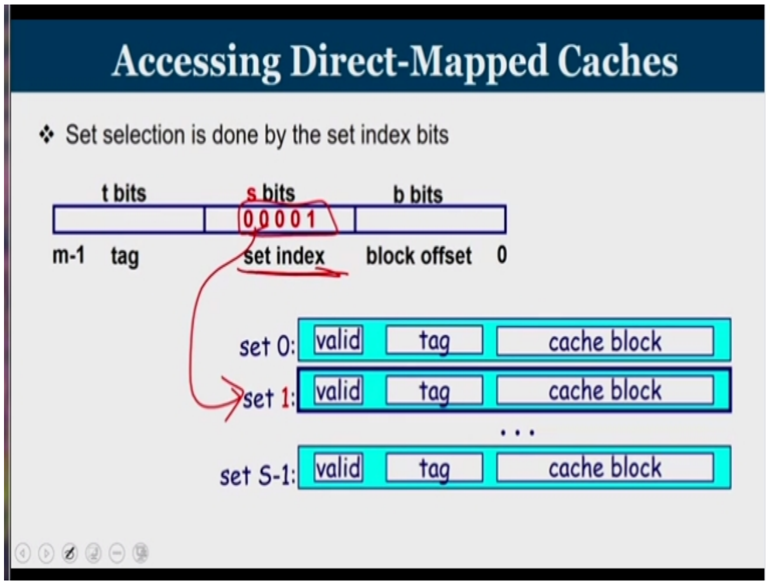


Cache Mapping / Block Placement

- **Direct mapped**
  - Block can be placed in only one location
  - (Block Number) **Modulo** (Number of blocks in cache)
- **Set associative**
  - Block can be placed in one among a list of locations
  - (Block Number) **Modulo** (Number of sets)
- **Fully associative**
  - Block can be placed anywhere

So coming to cache block placement the summary is direct map means a block can be placed in only in one location and that is been find out what is the block number in main memory modulo number of blocks in the cache. The second is called set associative block can be placed in one among a list of locations, so in this case block number modulo number of sets in the cache and the third one is called fully associative block can be placed anywhere.

(Refer Slide Time: 36:12)



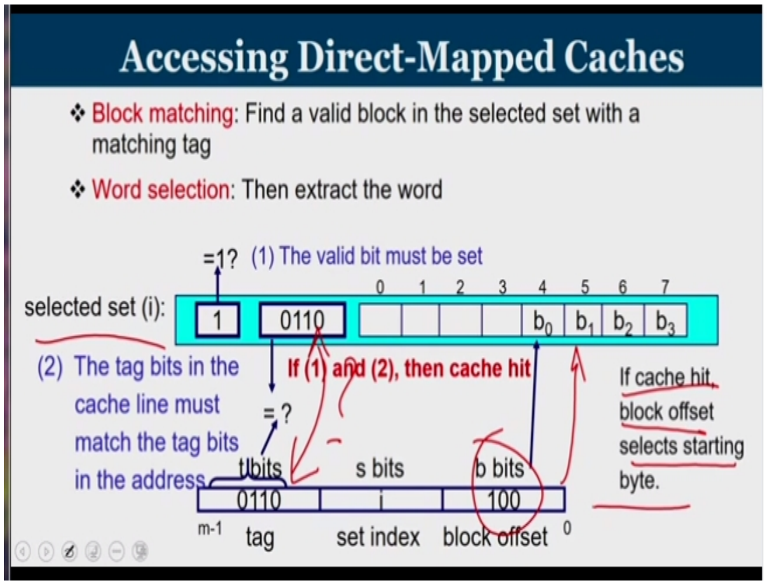
Accessing Direct-Mapped Caches

- Set selection is done by the set index bits

Now little bit of illustration about direct mapped cache and set associative cache direct map is a simplest kind of cache it is very easy to build that is only one tag to compare. So once you get into a set you know that the particular set has only one line, so you go and search whether the valid bit is equal to 1 or not. If the valid bit is 1 check the tag if it is same as the tag that you want if so this is your data using your offset try to extract.

So consider this case let us say in an example I am interested in the set index value let us say the set index value is 00001 it is a 5 set index value means I have 32 sets in my cache. So go to set number 1, that is a meaning of this, in set number 1 what are we going to do.

(Refer Slide Time: 37:01)



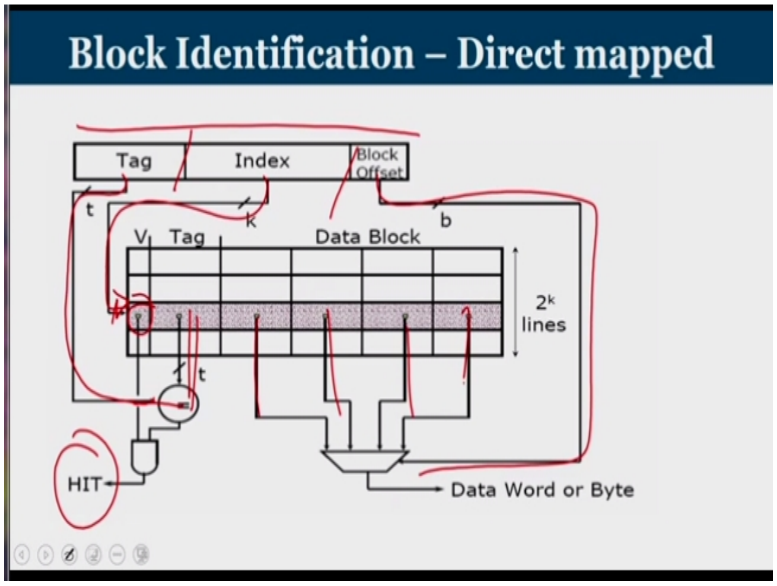
Accessing Direct-Mapped Caches

- Block matching: Find a valid block in the selected set with a matching tag
- Word selection: Then extract the word

=1?
(1) The valid bit must be set
selected set (i):
1
0110
0 1 2 3 4 5 6 7
b_0 b_1 b_2 b_3
(2) The tag bits in the cache line must match the tag bits in the address
=?
If (1) and (2), then cache hit
If cache hit, block offset selects starting byte.
t bits
s bits
b bits
0110
i
100
m-1
tag
set index
block offset
0

So once you get the set index portion once you read that particular set then 1 set the valid bit is equal to 1 or not with the valid bit is equal to 1 then search the tag in the given address is same as the tag that is already stored there in the cache. The tag bits in the cache line must match the tag bits in the address if that is happening then that is known as a hit. And if it is a hit there is if 1 and 2 both access together then it is a cache hit and using your offset bits you are trying to extract what is the word. If it is a cache bit then block offset selects the starting byte of this word.

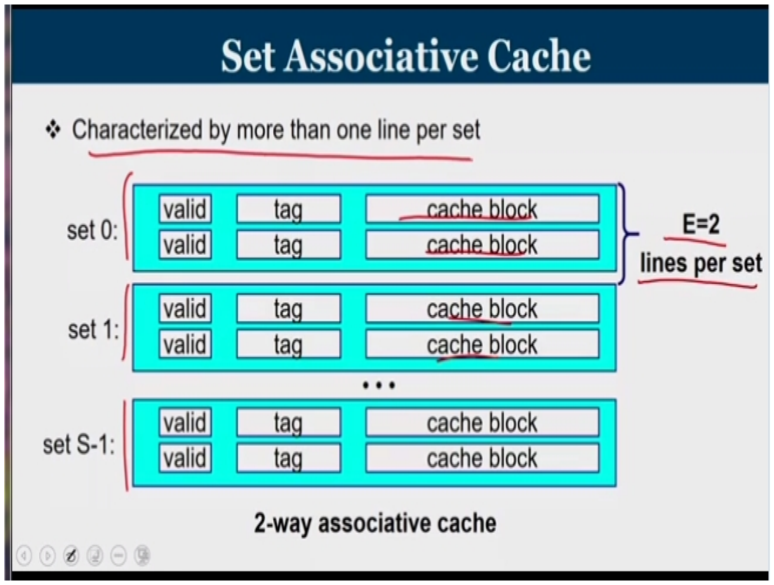
(Refer Slide Time: 37:42)


Block Identification – Direct mapped

This is how a direct mapped cache will look like in organizational level whatever is the address that you get from CPU the physical address you divide into tag index and offset using index, using a decoder will uniquely go in 1 particular set of the cache, extract the tag value. So tag value is extracted and it is compared with incoming tag and if it is matching it is known as a hit, so for matching the valid bit also should be 1.

And once if the match occurs using the block of set you could extract the corresponding word that is all about direct mapped cache.

(Refer Slide Time: 38:16)


Set Associative Cache

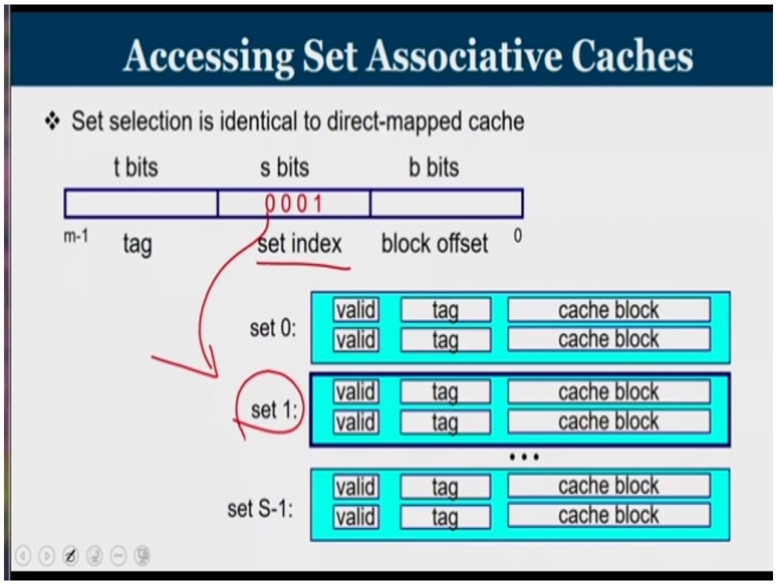
- Characterized by more than one line per set

Now coming to set associative cache you know that this is a 2 way associative cache where

$$E = 2,$$

so every set has 2 cache blocks. If every set has 4 cache block we call it 4 way associative cache, if you have n blocks inside a set then it is called n way associative cache. So a set associative cache is characterized by more than 1 line per set.

(Refer Slide Time: 38:41)

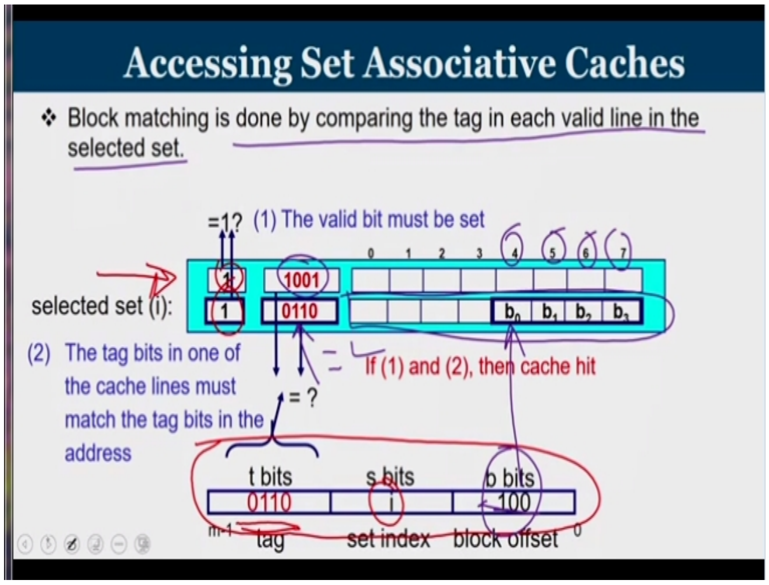


Accessing Set Associative Caches

❖ Set selection is identical to direct-mapped cache

Now in this case consider the case your set index was 0001 and that points to set number 1.

(Refer Slide Time: 38:40)



Accessing Set Associative Caches

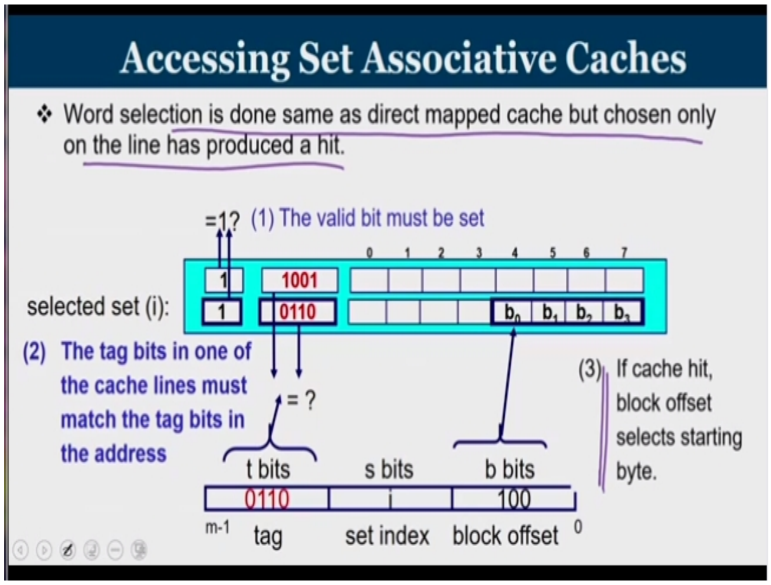
❖ Block matching is done by comparing the tag in each valid line in the selected set.

=1?
(1) The valid bit must be set
selected set (i):
1
1001
0110
0 1 2 3 4 5 6 7
b0 b1 b2 b3
(2) The tag bits in one of the cache lines must match the tag bits in the address
= ?
If (1) and (2), then cache hit
t bits
s bits
b bits
0110
i
100
m-1
tag
set index
block offset
0

Now once you go to this let us say this is the address that is being given and already selected a set. Now in this set I look whether both are valid or not yes both valid bits are 1. Now what I am interested is whether my tag bit 0110, the first one the tag bit is not matching this is a different tag whereas in this case the tag is matching 0110. Once a tag is matching this is the data that we are looking for but I do not know from where I have to start this offset will tell you from where to start.

Here the offset value is 4, so start from byte number 4 onwards 4, 5, 6, 7 to 4 bytes will give you the corresponding word. So block matching is done by comparing tag with each valid line in the selected set. So valid bit should be 1, tag should match if both 1 and 2 then it is a cache hit.

(Refer Slide Time: 39:42)



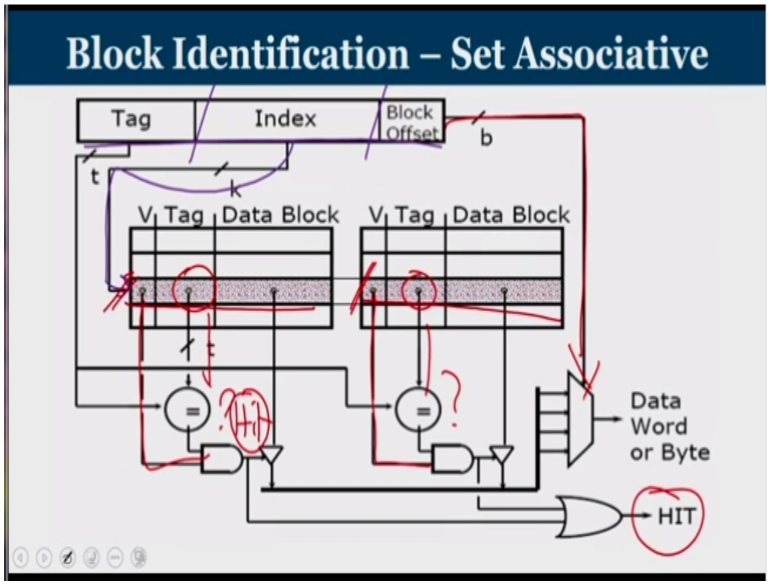
Accessing Set Associative Caches

- Word selection is done same as direct mapped cache but chosen only on the line has produced a hit.

=1?
(1) The valid bit must be set
0 1 2 3 4 5 6 7
1
1001
selected set (i):
1
0110
b_0 b_1 b_2 b_3
(2) The tag bits in one of the cache lines must match the tag bits in the address
= ?
(3) If cache hit, block offset selects starting byte.
t bits
s bits
b bits
0110
i
100
m-1
tag
set index
block offset
0

And once if there is a cache hit then what we do is word selection done same as direct mapped cache by choosing the line that he has produced the result or the hit. If it is a cache hit then the block offset select this starting point.

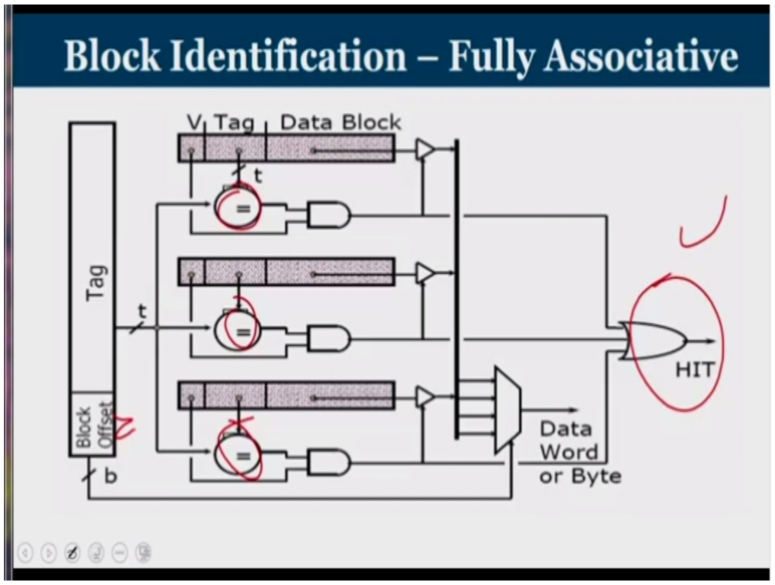
(Refer Slide Time: 39:58)



So this is how a conventional 2 way set associative cache looks like, you know that the address is being splitted up into tag index and offset using index you go into 1 particular set. In that set, this is the difference, in that set you how know 2 lines this is way 0 and this is way 1. Now parallely in both the ways I extract the tag value and the tag value is compared in both these cases and the valid bit also should be 1.

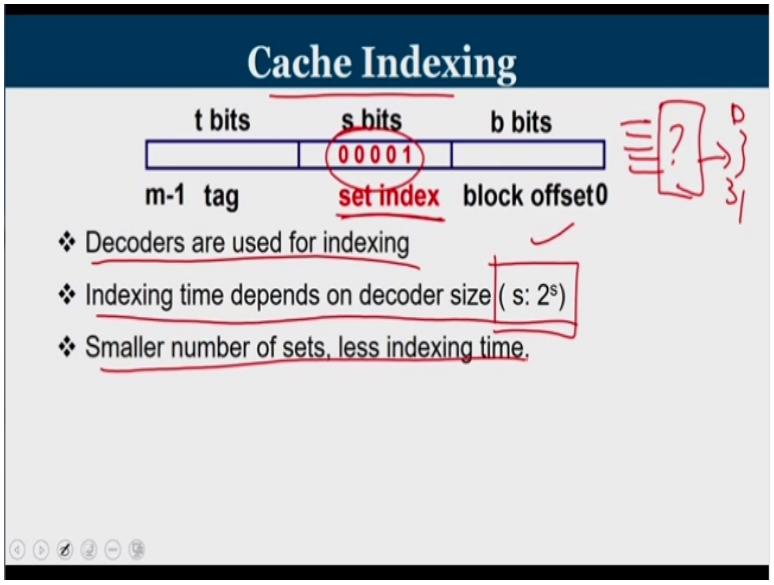
So if the tag is matching and valid bit is 1 and that determines whether you got a hit in this place or in this case effectively it is a hit and using the offset wherever you got a hit the offset will return the corresponding data word or data byte.

(Refer Slide Time: 40:45)



Where it comes to fully associative cache you do not have an index portion, the index portion will tell you where basically in which set you how look into. So such a kind of a localized help is not available you have to search everywhere, so what do you do is go to each of the data block and then you perform the tag comparison everywhere and then that determines the hit. Wherever or whichever place you get a hit then that is it, you got it but then you will take your offset value to extract the corresponding word ok.

(Refer Slide Time: 41:17)



Cache Indexing

- Decoders are used for indexing
- Indexing time depends on decoder size (s: 2^s)
- Smaller number of sets, less indexing time.

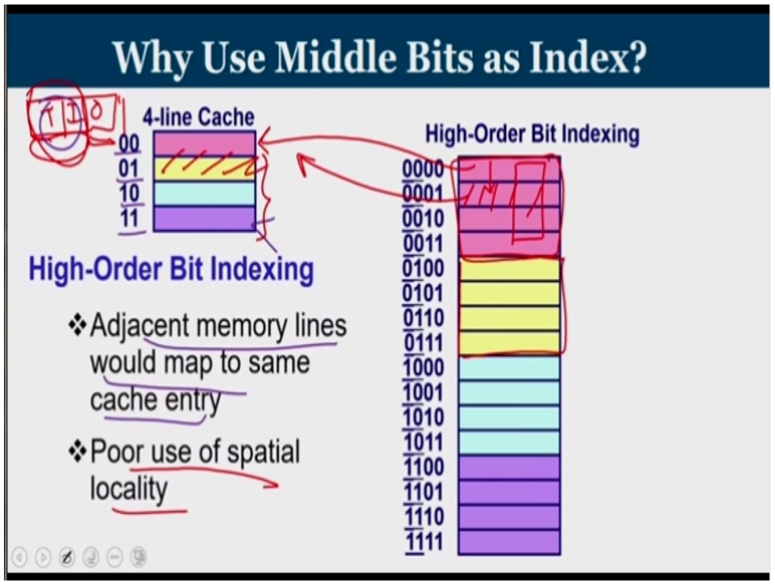
So to summarize that aspect what we have learned now, we have learned about what is direct mapped cache, set associative cache and fully associative cache and it is functioning the concept of division of set index, the set index is use basically to locate or to localize your search inside the cache memory. So in a cache we are not going to search the entire location from the address, a property of address which is nothing but few bits in the address are been extracted out and these bits will tell you where to look into the cache.

And once you go into the location this tag will tell you whether the data that is stored there is same as what you are looking for and once you get the hit then using the offset to try to extract. Let us now try to understand what is a complexity involved in locating one particular index. So this cache indexing if you look it, in the address our first duty is to find out the set index it is a few number of bits and these bits in this case you have 5.

So it is basically 5 bits are been passed onto a unit and they will choose one among the 32 sets, so it is basically a 5:32 decoder. So decoders are used for indexing and indexing time depends on the decoder size, so we have a decoder whose size is S:2 to the power of s. So when you have smaller number of bits in the set index the decoder that you are going to use is of smaller size that means effectively the time to decode is less, you will reach the cache memory faster.

So when the cache memory is big meaning that you have more number of sets it will take more time to index because you are using a big more complex decoder it will take more time to return the word. So smaller number of sets, less is the indexing time, less is the cache access time.

(Refer Slide Time: 43:15)



Why Use Middle Bits as Index?

High-Order Bit Indexing

- Adjacent memory lines would map to same cache entry
- Poor use of spatial locality

Now out of this 3 combination that is tag, index and offset why I am using the middle bits for indexing, why cannot I use the most signifiant bits for indexing? You know that out of the given address the offset portion are the bits which will vary for the bytes within a block, let us now not discuss about it and a more interested in this tag and index portion. So this tag and index portion together define your block number in main memory, this is the offset within a block.

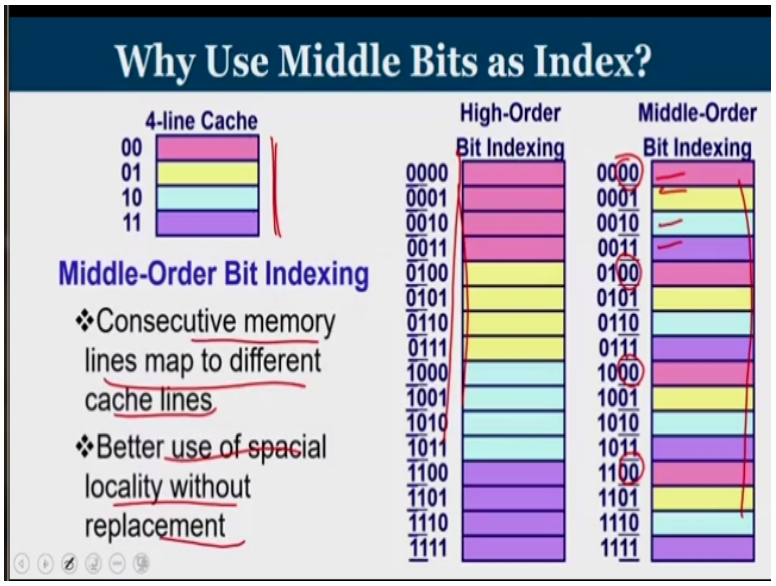
So this portion tag and index together defined what is the block number inside main memory, so consider I am talking about a 4 line cache let us say I am going to give the sets as 00, 01, 10 and 11. Now consider my main memory if I you see higher order for indexing, so consider higher order for indexing let us say I am having a main memory of total 16 lines and let us say these are the addresses given to the lines or line numbers basically.

If we use the most significant bits for cache mapping, all lines whose mass significant bit 00 will get mapped into this set, all lines whose mass significant bit is 01 will get map to the yellow set and 10 will get to set number 10 and 11 will get mapped to set number 11. So it is a higher order bit indexing that is there, so what happens is your adjacent memory lines would map to the same cache memories.

So consider a case you have a music program that is residing inside your main memory, now the first line of the music program to played I how to bring into this block. Once it is over let us say the address goes to the very next location even though by remaining 3 blocks in my cache memory is full it will evict out the previous one. Because the second one is also mapping into the same location, so it will evict out 1 and then only it will bring next.

So this is actually a poor use of spatial locality that means the nearby blocks will evict the just immediate block. So as the entire pink region will occupy one place in the cache one at a time the entire yellow origin will occupy only one place in the cache, this yellow portion.

(Refer Slide Time: 45:56)



Why Use Middle Bits as Index?

Middle-Order Bit Indexing

- ❖Consecutive memory lines map to different cache lines
- ❖Better use of spacial locality without replacement

If we are using lower order bits for indexing that is middle order bits for indexing then it is better usage a spatial locality you can see that. If these 2 bits are been used then these are the blocks that get mapped into the same location they are not adjacent to that. So consecutive memory lines are actually map into different lines, if you look it here they all are mapping to different lines, it is better use of spatial locality without replacement.

In this case we have lot of replacements whereas in this case such a kind of a replacement is not happening as we move to adjacent locations. So with this we come to the end this lecture where we had a quick introduction of the concepts of cache memory over the next lectures we will be moving into slightly advanced concepts of cache. Kindly go through the textbook and the video material multiple times if there is any query please let me know, thank you.