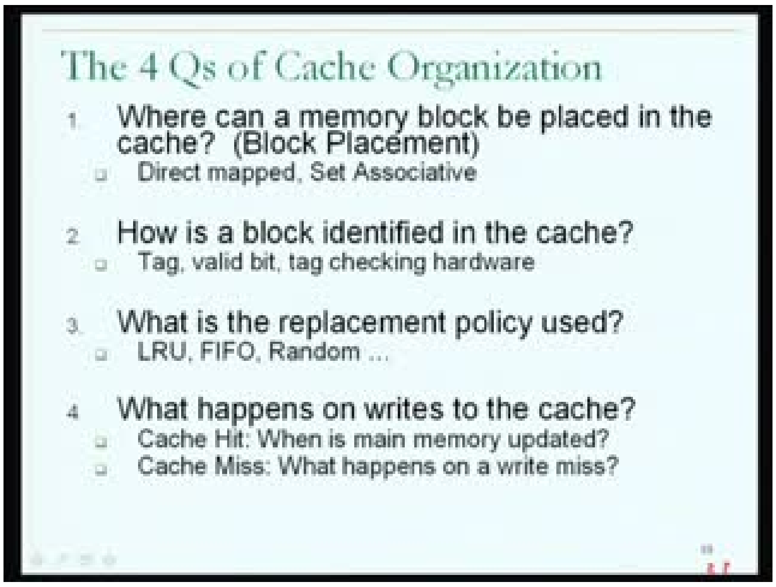
High Performance Computing

Prof. Matthew Jacob
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No. # 06
Lecture No. # 28

Welcome to lecture 28 of the course on high performance computing. We are currently looking at the operation of hardware cache memory which will be important in deciding the performance of our programs. Now, we are using the following frame work in which to understand the organization of cache memories.

(Refer Slide Time: 00:42)



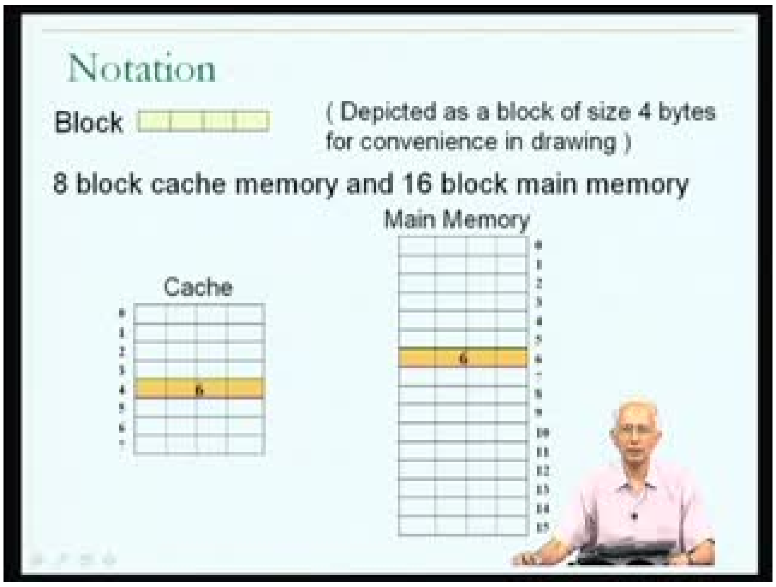
The 4 Qs of Cache Organization

1. Where can a memory block be placed in the cache? (Block Placement)
   - Direct mapped, Set Associative
2. How is a block identified in the cache?
   - Tag, valid bit, tag checking hardware
3. What is the replacement policy used?
   - LRU, FIFO, Random …
4. What happens on writes to the cache?
   - Cache Hit: When is main memory updated?
   - Cache Miss: What happens on a write miss?

We are looking at the 4 primary questions which a hardware designer who is designing a cache memory would have to make decisions about and I describe these as the 4 questions or the 4 Qs of cache organization. Formally, they are referred to as block placement which is the decision regarding where in the cache memory a particular main memory block could be located, if it is at all present inside the cache.

Secondly, there is a problem of replacement which we have seen. A very similar problem in the context of virtual memory and third, there are issues relating to how write modifications will be handled as far as the cache is concerned. Finally, the mechanics of how a block is actually identified when if it is present inside the cache.

(Refer Slide Time: 01:34)



Notation

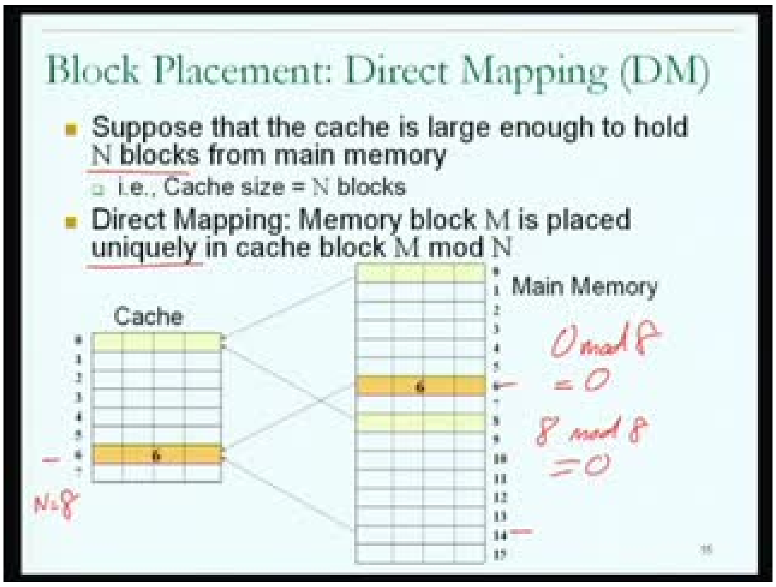
Block (Depicted as a block of size 4 bytes for convenience in drawing)

8 block cache memory and 16 block main memory

So, we are going to look at each of these and understand the various possibilities which could be present in the cache memories that we may encounter when our programs on different kinds of computers. Now, the notation that we are using is that I will represent diagrammatically a block of cache on main memory by a unit comprising in this particular example 4 entities (Refer Slide Time: 02:00).

Remember, that both cache and main memory are organized in terms not of individual bytes but, in terms of blocks from the perspective of the cache. A block being a neighborhood in the address space in which the least significant bits or the only bits in which the individual elements of the block differ in terms of their addresses for the small diagrams that I will be drawing. I will show an 8 block cache memory and a 16 block main memory to illustrate the ideas as shown in this diagram.

(Refer Slide Time: 02:24)



Block Placement: Direct Mapping (DM)

- Suppose that the cache is large enough to hold N blocks from main memory
  - i.e., Cache size = N blocks
- Direct Mapping: Memory block M is placed uniquely in cache block M mod N

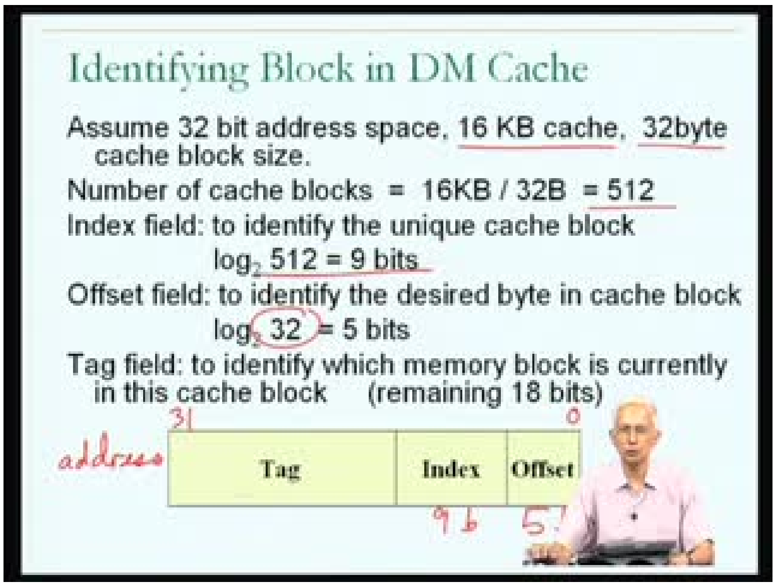
Cache
Main Memory
0 mod 8 = 0
8 mod 8 = 0
N=8
6

Now, we had seen the first example of a block placement policy. In other words, the strategy that could be used by a hardware designer in designing a cache in terms of where a main memory block, could be present inside the cache. The first option, what is known as direct mapping and in direct mapping we assume that if we know the size of the cache. Let us say, the size of the cache is N blocks as for example, in this particular diagram, I show you a block cache of size 8 blocks.

So, the idea in direct mapping is that the main memory block number M, whichever, it may be placed uniquely in cache block number M mod N. So, each main memory block could have associated with it only one location in the cache. One block in the cache where it could be present and the simple example if you consider, for example, main memory block 0 doing the calculation 0 mod 8 you find out that main memory block 0 could be present only in cache block 0. There is a unique relationship between each main block and exactly one of the cache blocks. Similarly, in main memory block 8 could also only be present in cache block 0 because 8 mod 8 is also equal to 0.

So, this is the idea of direct mapping and we could see that the other example was, if you look at main memory block 6 and main memory block 14, both of them could uniquely map to cache block 6; since, 6 mod 8 as well as 14 mod 8 are both equal to 6. Therefore, only of them could be present in the cache at a given point in time.

(Refer Slide Time: 04:10)



Identifying Block in DM Cache

Assume 32 bit address space, 16 KB cache, 32byte cache block size.

Number of cache blocks = 16KB / 32B = 512

Index field: to identify the unique cache block

$\log_2 512 = 9$ bits

Offset field: to identify the desired byte in cache block

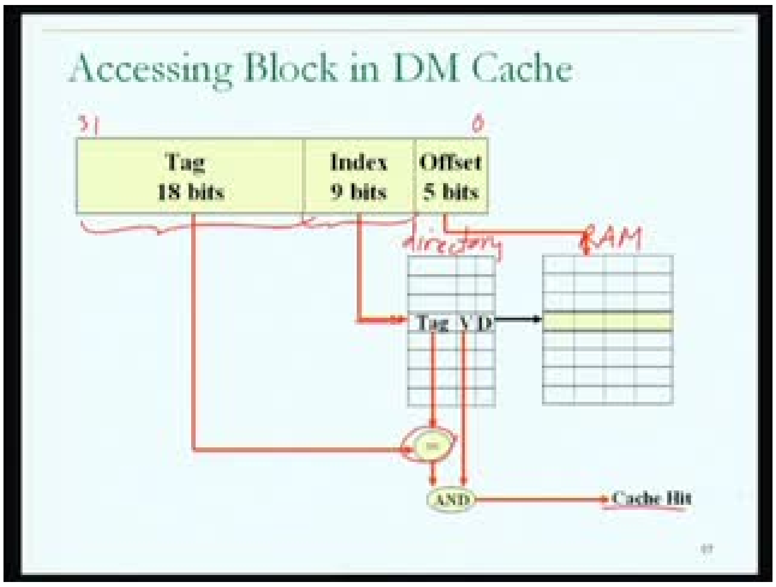
$\log_2 32 = 5$ bits

Tag field: to identify which memory block is currently in this cache block (remaining 18 bits)

Now, the in terms of the direct map cache, we looked at the example of a specific cache, just to make sure we understand the mechanics of what is happening and the example I used was 16 kilobyte cache where each block is of size 32 bytes. I will also tell you the size of the address associated with the accesses into the cache and we actually worked out using this information that the size of the offset field. Do remember that the address is used by the hardware of the cache by breaking it up into 3 fields. So, this is a particular address which may be sent by the processor to the cache and the least significant bits are at this end; the more significant bits are at this end (Refer Slide Time: 04:45).

Since it is a 32 bit address, the bit number 31 is what we call the most significant bit and we are able to calculate that the number of bits in the offset field would be determined by the size of a cache block.

(Refer Slide Time: 05:20)

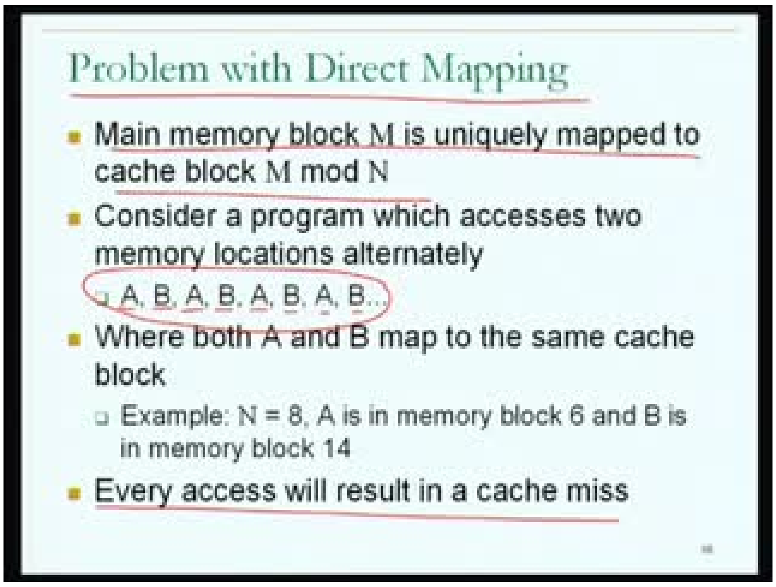


So, that would be in this case 5 bits the number of bits in the index field would be determent by the number of blocks in the cache which in this case would be 9 bits and the remaining bits would serve the purpose of tag to differentiate between the different main memory blocks which could be present in anyone of the cache blocks at a time and we looked at the hardware which might be involved behind the scenes in checking in doing the look up in an determining if the particular main memory block is present.

So, once again if you look at the cache the address again remember, whenever I show you a block like this is an address. So, 32 bits of the address the index bits of the address I used index into the cache directory; remember that the cache is made up of memory which is capable of storing small amount but accessible at very high speed as well as the directory which is a look up table of the concurrent contents of the cache.

The cache hardware uses the index bits to decide which particular entry in the directory to do the look up from there, compares one of the fields within the directory with the tag bits from the actual address which is currently required by the processor. If they are the same and in addition, if that particular cache block is known to be valid because it currently contains a piece of data then, this is what is referred to as a cache hit. The situation where the cache actually does contain the data or instruction required by the processor and then using the offset bits from the address they require byte or half word or word, can be extracted from the cache RAM and provided to the processor.

(Refer Slide Time: 06:40)



Problem with Direct Mapping

- Main memory block M is uniquely mapped to cache block M mod N
- Consider a program which accesses two memory locations alternately
  - A, B, A, B, A, B, A, B…
- Where both A and B map to the same cache block
  - Example: N = 8, A is in memory block 6 and B is in memory block 14
- Every access will result in a cache miss

Now, there is of course a problem with direct mapping. Otherwise, they would not be alternatives at the disposal of the cache designer and the main problem is arises from the mapping function. Basically, if main memory block M is uniquely mapped to cache block M mod N where N is the number of blocks that can be contained in the cache then, we have this problem that there would be many main memory blocks that would actually be contending for each of the cache blocks.

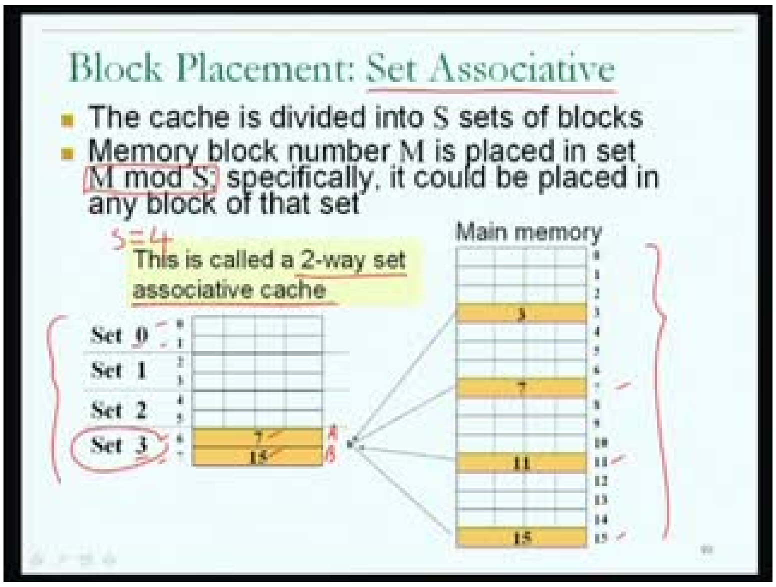
If I consider a program which accesses 2 memory locations alternately, in other words, first accesses memory location A, then it accesses memory location B and so on as its normal behavior. If it happens that both memory block A and memory block B map to the same cache block then, what we would have as a situation where first memory block A would be in the cache but then, when the reference to memory block B comes up, memory block B would be fetched and replace memory block A from the cache and so on.

So, they would consecutively replace each other from the cache and end up giving us a situation where all of the accesses would have to actually take place out of main memory and none of the accesses would be satisfied out of the cache. In effect, every access would result in a cache miss and you will get none of the benefits of having the fast cache, the fast RAM present in the cache.

In fact, every single memory reference would have to take place out of main memory to out of the magnitude slower than the processor. So, this is the serious problem with direct mapping. So, in that they could be programs. You will notice that this kind of behavior which I described over here is not illegal; it is certainly some kind of behavior which a program could show. and there If there could be programs which are not badly behaved that could result in extremely poor performance, then one must have alternatives to the design of the cache.

Now, here the problem is that the problem direct mapping, is that there is a unique mapping between a main memory block and a cache block and therefore, any alternative mapping strategy should provide some flexibility in order to overcome this problem.

(Refer Slide Time: 08:45)


Block Placement: Set Associative

- The cache is divided into S sets of blocks
- Memory block number M is placed in set M mod S; specifically, it could be placed in any block of that set

S=4

This is called a 2-way set associative cache

Now, one of the alternatives to direct mapping is a block placement strategy known as a set associative strategy. The idea in the set associative strategy is that rather than having a unique main memory cache block associated with each main memory block, the cache itself is divided into sets of blocks and subsequently a main memory block would be mapped, not into an individual cache block but, into an individual set of cache blocks specifically. Subsequently, could be placed in any one block of that set. As a consequence, they could be more than 1 cache block into which a main memory block could go.

If you look back at our previous example where I have main memory block A and main memory block B, alternately required by the processor and therefore, having to be inside the cache; if I have a situation where the size of each set of blocks is 2 then, I could actually have both A and B present in the cache. Therefore, I would not suffer a huge number of cache misses. That is the idea of the set associated strategy where, the cache is divided into sets of blocks and any main memory block could be present in any one of the sets uniquely associated with a set based on its M mod S where S is a number of sets in the cache.

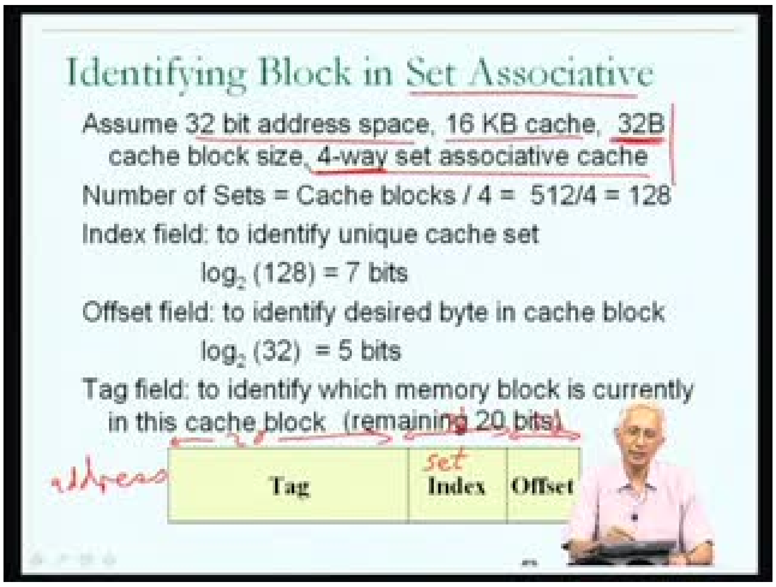
So, if we look at an example where once again we have 16 main memory blocks and we have 8 cache blocks. But, I have chosen to use a set associative mapping and what I have assumed in this diagram is that there are 4 sets of 2 blocks each. In other words, each of the sets which I have numbered 0 through 3, comprises 2 blocks for example set 0 comprises block 0 and block 1

Set 3 comprises block 6 and 7 of the cache. Now, when the time comes to map a main memory block into the cache, we do the calculation of the main memory block number mod S, where in this particular example S would be equal to 4 since, there are 4 sets in the cache. So, for example, if you consider main memory blocks 3 or 7 or 11 or 15 all of them, have the property that M mod 4 is equal to 3. 3 mod 4 is equal to 3 7 mod 4 is equal to 3 11 mod 4 is equal to 3 and 15 mod 4 is equal to 3.

So, all 4 of these blocks would map to set number 3; within set number 3 there are 2 blocks; block 6 and block 7. Therefore, at any given point in time, any two of these blocks could be present inside the cache and this particular diagram (Refer Slide Time: 11:12) I am showing you, situation where currently block 7 and block 15 from among those for are actually both present in cache at the same time. Hence, even if I had the situation where this is my block A from the previous slide and this is my block B, both could be present in cache and therefore, the performance of the program would not suffer as badly as I did in the case of the direct map cache.

So, this particular kind of an organization in general, is called set associative mapping. But, to also describe the fact that each set contains 2 blocks; this kind of mechanism that I have drawn in the diagram is often referred to as a 2-way set associative mapping or the cache might be described as a 2-way set associative cache, giving you a clear indication that there are 2 locations or 2 blocks in the cache where a main memory block could be present.

(Refer Slide Time: 12:21)



Identifying Block in Set Associative

Assume 32 bit address space, 16 KB cache, 32B cache block size, 4-way set associative cache

Number of Sets = Cache blocks / 4 = 512/4 = 128

Index field: to identify unique cache set

$\log_2(128) = 7$ bits

Offset field: to identify desired byte in cache block

$\log_2(32) = 5$ bits

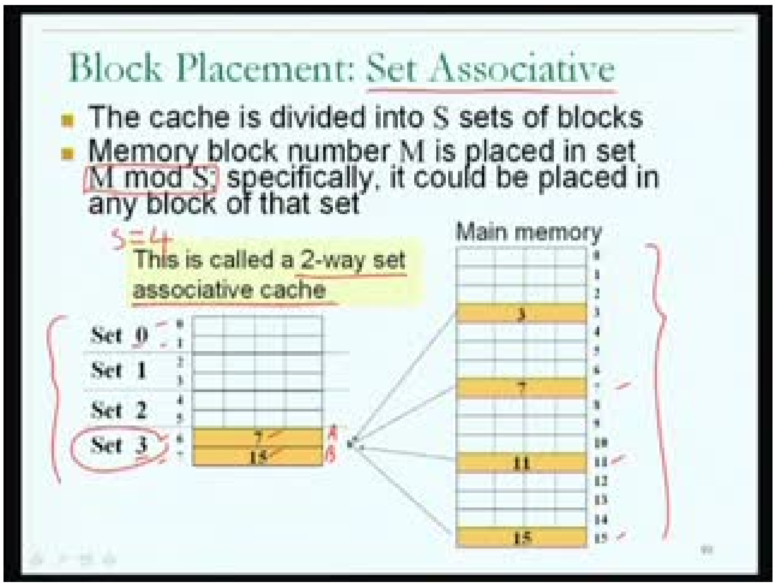
Tag field: to identify which memory block is currently in this cache block (remaining 20 bits)

Now, following the our mechanism that we used in understanding the direct map cache, we will now try to understand what happens to an address as generated by the processor and the various steps that happen prior 2 is actually being the processor getting the data or instruction that it had requested. I will use a slightly different example. Once again, I will assume that we have a 16 kilobyte cache and that the size of each cache block is 32 bytes; 32 bit addresses is what we will assume. I will assume that we have a 4-way set associative cache and by this we must understand that it is a set associative cache. In other words, it is not direct map. In other words, there are alternatives of where in cache a main memory block could be present and in this particular situation, the cache is designed so that there are in fact 4 alternatives any main memory block could be present in a set of the cache and each set comprises 4 blocks.

So, how do we work out? How the cache hardware views and address we have to bear in mind that once again, we are talking about an offset which is the offset which specifies which particular byte or half word or word in the block is being referred to. Then, we have the index field which now is not referring to a specific block but, to a specific set and therefore, to figure out how many bits are required in each of these fields we have to use the data provided in the statement of the organization of the cache. So, once again I can calculate the number of sets in the cache by dividing the number of cache blocks by 4 because I know that there are 4 blocks within each set. Therefore, the number of sets will be given by the number of cache blocks divided by 4 which is going to be 128 from which I can infer the number of bits that are required for the set index that would be given by log base 2 of 128 which is 7 bits.

Similarly, the calculation of how many bits are required for the offset will be exactly the same. As for the direct map cache because here as in the direct map cache, we have a 32 byte block size. Therefore, to identify any particular byte within a block requires log base 2 of 32 which is 5 bits, the remaining bits of the address which in this example would be 20 bits would comprise the tag bits.

(Refer Slide Time: 08:45)



Block Placement: Set Associative

- The cache is divided into S sets of blocks
- Memory block number M is placed in set M mod S; specifically, it could be placed in any block of that set

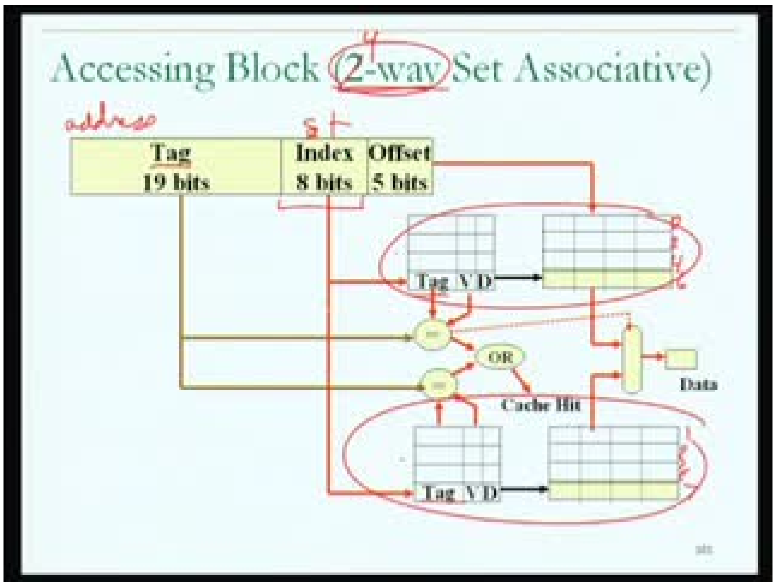
This is called a 2-way set associative cache

So, now if you look at a diagram describing this, I am actually going to do a slightly different example from what I had worked out over here. In this particular example, I had talked about the 4-way set associative cache but you will notice in the diagram I am going to refer to a 2-way set associative cache. This once again, is just for ease of drawing the diagram as for reasons that we will see shortly.

Now, if I talk about a 2-way set associative cache, recall, I am referring to a situation where then each set of cache blocks comprises 2 blocks. So, the way I am going to draw it in this diagram is, I am going to divide the cache directory and the cache RAM into 2.

So, that is another way of viewing the 2-ways. Any particular main memory block could either be present in a block in the upper half or could be present in a block in the lower half. I am just dividing rather than showing you the set comprising consecutive blocks of the cache. I am showing you of slightly different organization, just to speed up the understanding of how the look up might be done.

(Refer Slide Time: 15:48)



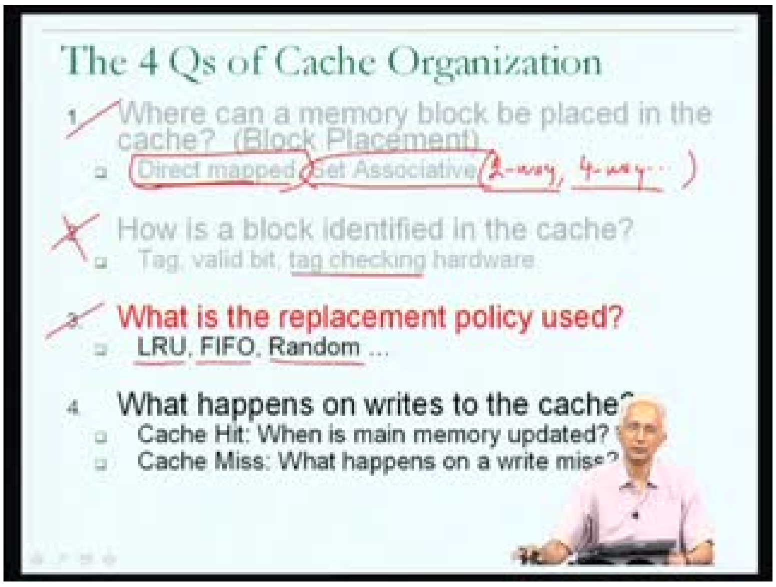
So, from the previous diagram you might say that this might be cache block number 6 and this might be cache block number 7. I am just showing it divided in a slightly different way; so, 0 2 4 1 3 5. The numbers of the cache blocks in order that is important to us at this point. But, we do know that when an address goes to the cache hardware based on the kind of calculations that we did in the previous slide, some number of bits are used to index into the cache directory.

Now, in this particular perspective on the cache directory, we understand that the cache directory has been divided into 2 and that therefore, the look up of the 8 bit set index into the 2 parts to the cache directory could be done at the same time which I am going to show by this diagram. The 8 bits from the address simultaneously or indexed, I use to index into the upper half cache directory and the lower half cache directory.

In both cases, the tag bits from the cache directory will be compared with the tag bits from the address that will require 2 quality pieces of hardware, 2 comparators and in addition, the valid bits in the 2 halves of the cache directory would have to be checked and if there is a match in either the upper or the lower half that would be known as a cache hit. Subsequently, the offset bits could be used to extract the appropriate part of the data from the cache block.

Now, if I had a 4-way set associative cache instead then, the argument would go very much the same way expected; that I would have to show you not 2 sets of blocks but 4 sets of blocks and the diagram would have become a little bit busier and a little harder to understand. But, the functioning of the cache would have been very much along the same lines. In other words, using the index bits all the 4 cache directory would have been looked into the comparison would have been done with all the 4 and if one of them had, did contain happen to contain the information required then, that would be called a cache hit.

(Refer Slide Time: 17:54)



The 4 Qs of Cache Organization

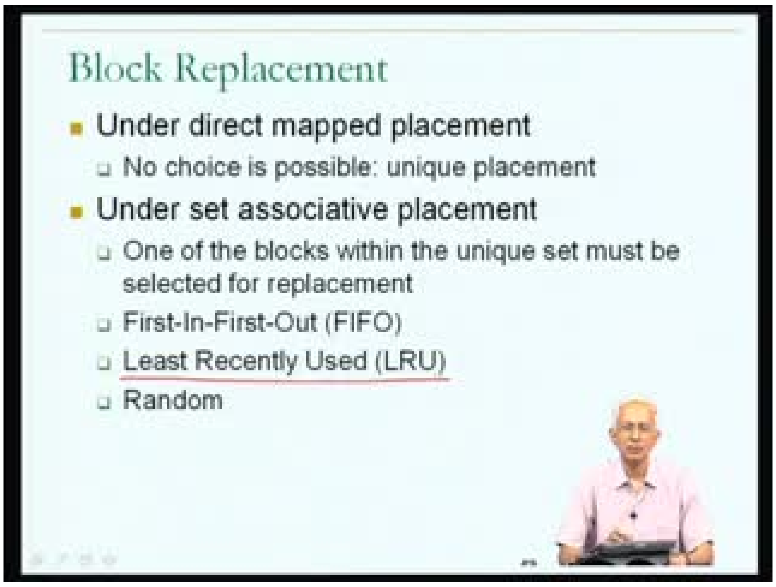
1. Where can a memory block be placed in the cache? (Block Placement)
   - Direct mapped, Set Associative (2-way, 4-way ...)
2. How is a block identified in the cache?
   - Tag, valid bit, tag checking hardware
3. What is the replacement policy used?
   - LRU, FIFO, Random ...
4. What happens on writes to the cache?
   - Cache Hit: When is main memory updated?
   - Cache Miss: What happens on a write miss?

So, with this we have got fairly good handle on the two main alternatives for block placement in terms of the 4 Qs of cache organization. We have seen idea of the direct map cache; we have seen the idea of the set associative cache and we understand that if somebody tells you that there is a certain that you are the hardware you are using include the set associative cache, they may also specify whether it is 2-way or 4-way. In other words, the number of cache blocks present within each set and you can imagine that they could even be larger associative caches; for example, they could be caches which are 8 way set associative and so on.

We have also seen along the way how the cache hardware views the address and then uses the different bit fields within the address to do the look up tag checking. In other words, comparing the tag bits within the directory with the tag bits from the address to determine whether the instruction of the data represented by the address as requested by the processor, is present in the cache or not. With this we can move on to the third Q. The third question of cache organization namely, the question of replacement and we have seen something about replacement before when we talked about virtual memory where we talked about replacement policies such as LRU, first in first out, or random. Now, we should also understand that the same problem that may it cache page replacement necessary, could arise in the case of the cache. That is a situation where the processors send an address to the cache and the cache hardware using the mechanisms. That we saw determines that the particular piece of data is not present in the cache; this is equivalent of the page fault; we call it a cache miss in the context of the hardware cache.

Very clearly, there may be a need to replace a block which is currently present inside the cache, in order to make way for the block it must be fetched from main memory into the cache. So, that is why the whole idea of our replacement policy enters the picture now. We need to think separately about how replacement may have to be done for the different kinds of cache placement that we have seen. We have seen a few different cache placement policies; direct mapped 2-way set associative, 4-way set associative.

(Refer Slide Time: 20:14)



Block Replacement

- Under direct mapped placement
  - No choice is possible: unique placement
- Under set associative placement
  - One of the blocks within the unique set must be selected for replacement
  - First-In-First-Out (FIFO)
  - Least Recently Used (LRU)
  - Random

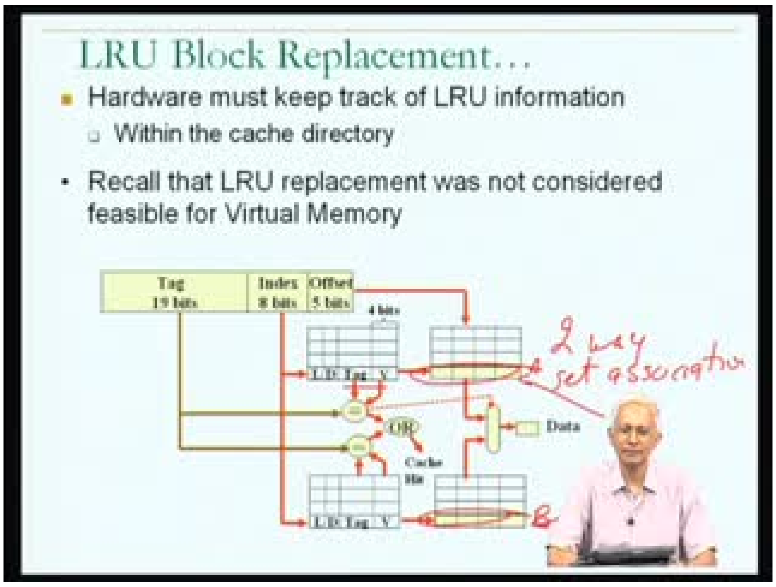
So, we need to figure out how block replacement could be done for each of those. Let us think first about under the direct mapped placement policy. Now, recall that under the direct map placement policy, there is a unique relationship between a main memory block and exactly one of the cache blocks. Therefore, if on reference to a particular main memory block it is found that that main memory block is not present in the cache, then there is really no decision that has to be made about which cache block should be replaced because, there is a unique relationship between a main memory block and the cache block. Therefore, one could clearly see that there is no block replacement policy required, because there is a unique relationship between a main memory block and the cache block. So, no choice is possible since because this unique placement property under direct mapping and therefore, there is no need to have a block replacement policy. The hardware does not have to make a decision; the decision is decided by the placement policy.

Now, this is not the same if one considers set associative scenario. Remember, that under set associative placement, a main memory block could be present anywhere within a set of cache blocks. In other words, if the size of a set is 2 it could be present in either of 2 cache blocks, or if size of a set is 4, it could be present in any one of 4 cache blocks and therefore, if it is found that the main memory block which is required is not present in the cache. Subsequently, it could be present in any one of the elements of that set. One of the blocks which is currently occupying the cache within that set, will have to be selected for replacement.

Therefore, there is a decision to be made by the cache hardware which is why replacement policy would be necessary. One of the blocks within the unique set must be selected for replacement and just as we argued in the case of virtual memory, one could think about alternative block replacement policies. First in first out is a possibility; least recently used is a possibility and random is also a possibility.

Now, when we talked about virtual memory page replacement we had this concept of the principle of locality of reference which we used as a basis on which to model the future behavior of a program. We realize that in making a replacement decision one must try to take into account, how the program is likely to behave in the near future and on the basis of the principle of locality of reference, the least recently used policy was actually viewed as the one which made more sense for the well-being of the program.

(Refer Slide Time: 23:05)



LRU Block Replacement…

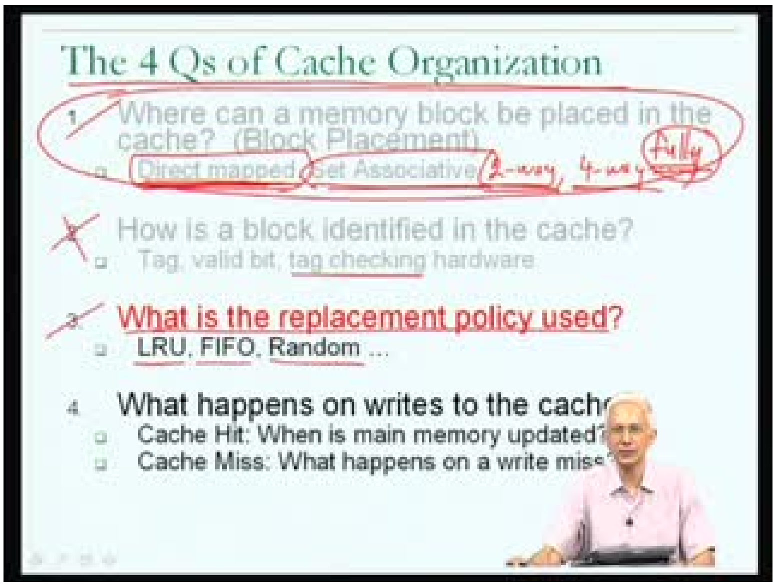
- Hardware must keep track of LRU information
  - Within the cache directory
- Recall that LRU replacement was not considered feasible for Virtual Memory

Unfortunately, we found that it was not a feasible policy in the context of virtual memory because, of the vast amount of information that may have to be maintained to accurately keep track of which page was least recently used. Now, fortunately in the case of a cache hardware, the problem may not be that hard in the sense that, it may be fairly easy to keep track of information about which cache block within a set was least recently used by this maintaining that information within the cache directory.

You may want to just look at this way; if you look at the example of the 2-way set associative cache which is the one we had drawn the diagram for. I had drawn the diagram by showing you the 2 sets or the 2 blocks associative with the set as being one of them in the upper half, one of them in the lower half of this particular break up and when the time comes to make a block replacement decision, we basically have to decide whether to replace the block from the top or the block from the bottom. In order to keep track of which of these had been least recently used, we will notice that we feel the need to keep track of only 1 bit of information. Therefore, the amount of information that has to be maintained and how frequently that bit of information has to be modified, is far smaller than was the case for virtual memory.

So, we had to keep track of, among all the pages, if you are currently in virtual memory which had been least recently used here is just a question of determining between the block A and the block B which had been least recently used. That could be maintained, it 1 bit of information which could be maintained in the cache directory. Therefore, one has to notice that though replacement by exact LRU was deemed by as infeasible in the case of virtual memory. In the case of cache memory, it might in fact not be impossible. It could in fact, may not be very expensive to maintain exact LRU information. In the case of the 2-ways set associative cache, it might just take 1 bit of LRU information to be associated with each set. If I had a 4-way set associative cache, the number of bits of information that would have to be maintained would be little bit more and the amount the frequency with which or the amount of work that had to be done on every access to the cache, would have may have to be a little bit more. But, still it might be viewed as being something that could be done by not too complicated hardware.

(Refer Slide Time: 25:35)



The 4 Qs of Cache Organization

1. Where can a memory block be placed in the cache? (Block Placement)
   - Direct mapped, Set Associative, 2-way, 4-way, Fully
2. How is a block identified in the cache?
   - Tag, valid bit, tag checking hardware
3. What is the replacement policy used?
   - LRU, FIFO, Random ...
4. What happens on writes to the cach
   - Cache Hit: When is main memory updated?
   - Cache Miss: What happens on a write miss

Now, the question which you may be asking is, what was the difference between virtual memory and cache memory? We very quickly dispense with LRU as a feasible policy in the case of virtual memory when the case of cache memory we had we did not and to fully understand what is happening here we will just go back to our 4 Qs and ask a question which I had raised in the previous lecture and that was when we talked about virtual memory. We did not actually raise the question of the 4 Qs of virtual memory organization but, we could try to place our discussion of virtual memory in the context of our discussion of cache memory.

In other words, if I had to describe the virtual memory schemes that we talked about in terms of how they address the first question. In other words, how do they decide where in main memory a particular virtual page could be present, would I describe it as direct mapped or where I describe it as 2-way set associative etcetera.

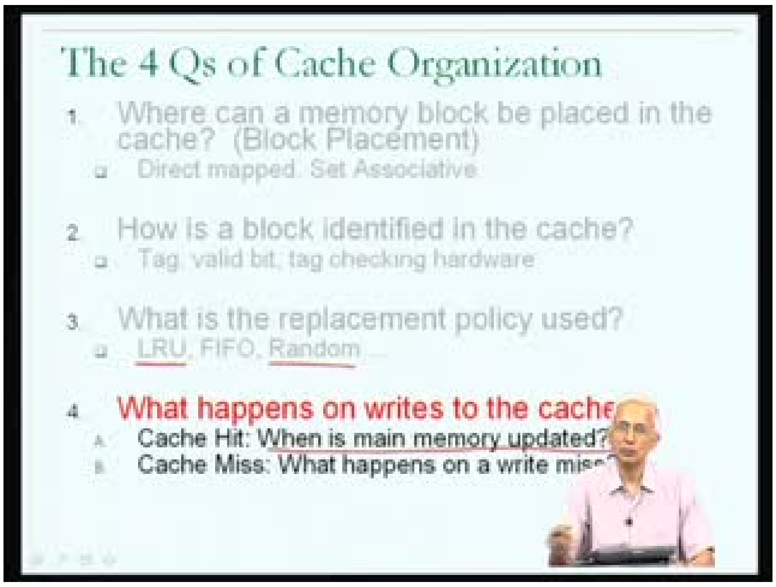
So, which of these placement policies is actually what we were assuming when we talked about virtual memory and you will recall that if you look back or recall what was happening when we talked about virtual memory. We left the option about where a particular virtual page could be present within main memory open in the sense that are particular virtual page could actually be present in any of the main memory pages. There was no restriction on where a virtual page could be present in the physical memory.

That was not direct map remember that in direct mapping there is a unique relationship between any cache block and a particular, a specific, between any main memory block and as particular cache block. Whereas, in the case of virtual memory, we had this open possibility any virtual page could be present in any physical page.

So, that was not direct mapping; it was not 2-way set associative; it was not 4-way set associative; but it seems have been something in the extreme end of set associative caches. Something, which you might call fully associative where there is an open mapping in that. If I use a pralines of caches any main memory block could be present in any cache block, I might refer to that as fully associative. In other words, there is 1 set of cache blocks and any main memory block could be present in any of the cache blocks. So, that seems to have been the placement policy that was being used in the case of virtual memory. In that, any virtual page could be present in any physical page we place; no restrictions of any kind on that mapping and this was the reason that the amount of information that had to be maintained in order to keep track of least recently used pages became so large; because we had to keep track of the least recently used information across the entire collection of pages currently present in physical memory. That problem is eliminated in the case of caches by using these more restrictive mapping strategies partly from the interest of making the look up fast.

Since we want the cache memory to operate fast, we restricted the mappings, the different places in the cache where a main memory block could be present and the consequence is that the word, the replacement policy becomes a little bit easier to make because, there are less alternatives of what could be replaced. As we saw in the case of the 2-way set associative cache, only 1 of 2 cache blocks need be considered for replacement unlike in virtual memory where potentially any physical page would have to be considered for replacement.

(Refer Slide Time: 28:48)



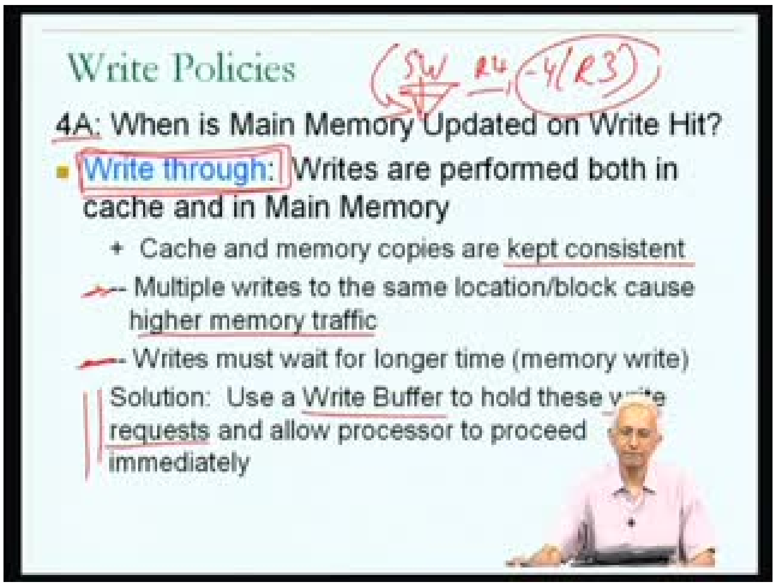
The 4 Qs of Cache Organization

1. Where can a memory block be placed in the cache? (Block Placement)
   - Direct mapped, Set Associative
2. How is a block identified in the cache?
   - Tag, valid bit, tag checking hardware
3. What is the replacement policy used?
   - LRU, FIFO, Random ...
4. What is happens on writes to the cache
   A. Cache Hit: When is main memory updated?
   B. Cache Miss: What happens on a write miss

So, it turns out that LRU replacement is feasible for caches and we might expect to see exact LRU information used and maintained in the cache directories to make a very well informed choice about blocks which are least slightly to be required in the near future based on their LRU property.

With this, we might assume that many caches actually used in LRU, is an exact LRU replacement policy but of course, if the caches are highly associative it is conceivable that the amount of information required to maintain exact LRU may become higher than is deemed feasible for the fast operation of the cache and some alternative may under be in use.

(Refer Slide Time: 30:12)



Write Policies

SW R4, -4(R3)

4A: When is Main Memory Updated on Write Hit?

- Write through: Writes are performed both in cache and in Main Memory
    - + Cache and memory copies are kept consistent
    - - Multiple writes to the same location/block cause higher memory traffic
    - - Writes must wait for longer time (memory write)

    Solution: Use a Write Buffer to hold these write requests and allow processor to proceed immediately

Now, this brings us to the fourth issue in connection with the organization of caches and fourth issue has 2 sub questions in it. The fourth question: what happens on writes to the cache? The first question 4A relates to, what happens on a cache hit? In other words, if there is an access made by the processor and it is a store instruction and it is found that the piece of data is present in the cache then, the question becomes, when is main memory updated? Will main memory be updated right away or main memory be updated at some later point in time? The 2 terms which are given from these two alternatives or to describe them by the 2 terms write through.

So, the first is what we will call write through the idea on the write through policy in connection with question 4A, is that when there is a store instruction which results in a hit in the cache, then the writes are performed not only in the cache but also in main memory write away. Therefore, associated with every write hit in the cache there will be an immediate main memory access and the benefit of doing this is that they will always be one to one correspondence between the current contents of a cache block and the current contents of the main memory block that it corresponds to and is a copy of and therefore, there is some value in having the consistency between the 2 copies.

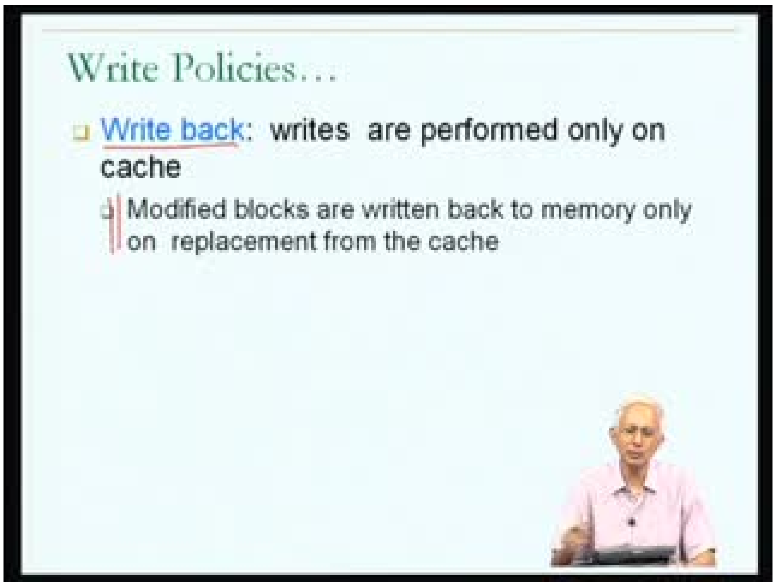
There will never be a situation where the main memory copy of a piece of data is different from the cache copy of the same piece of data, if you use a write through policy for question 4A.

Now, there is of course in negative to using the write through policy. So, if the hardware cache designer decides to implement hits cache so that the answer to question 4A is write through, in other words, main memory is updated along with the cache in the event of a store hit then, there will be more memory traffic because, for every store instruction that is a hit they will not only be update in the cache but they will also be a main memory access which could take to 100 nanoseconds or more. Therefore, this is a negative which is why I show it as negatives in front of here, this; it will increase in their use of memory which was, may potentially slow down the programmer little bit. It also makes it necessary for the store instruction to take a little more time to execute than let us say a load instruction because, there is store instruction; may have to actually take as much time as it takes to update the main memory, which could be 100 nanoseconds.

So, this also has to be viewed as a negative but this second negative can actually be overcome by allowing the main memory to be updated after the instruction has been deemed to be completed execution (Refer Slide Time: 32:45). In other words, if there is a store instruction and this is what the store instruction looks like for example, in the mips instruction set.

So, this is the effective address; some address goes to the cache along with a data. If it is found that it is a hit, the cache memory could be updated immediately and the activity of causing main memory to update to be updated could be initiated. But, the processor could proceed to the next instruction in the program right away and that is the idea of this solution over here the solution uses a piece of hardware called write buffer. Basically, it is a small piece of hardware which remembers the write request and sends it to memory to the main memory but, allows the processor to proceed with the next instruction, once the cache memory has been updated with the data corresponding to the store instruction. Therefore, the second negative that we talked about is not that big a problem. The first negative is still a problem because, whenever there is a store instruction it will cause a little bit more memory traffic which will keep the memory a little bit busier. It need not cause the processor to wait for the store instruction to complete and therefore, need not have an impact on the execution time of a single program; but, could cause the activity in the memory to be a little higher than if I had chosen some other alternative in answering question 4A.

(Refer Slide Time: 33:50)



Write Policies…

- Write back: writes are performed only on cache
  - Modified blocks are written back to memory only on replacement from the cache

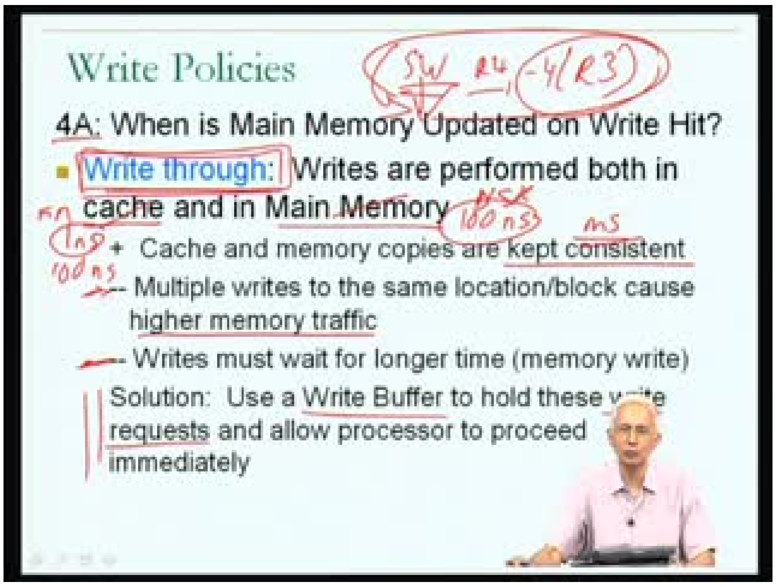
The alternative to write through in question 4A is what is known as write-back and in write-back the idea is that when there is a hit on execution of a store instruction a cache hit on execution of a store instruction then the write is performed only on the cached copy or not immediately on the main memory copy.

The advantage of doing this of course, is that the main memory need not be updated write a way and therefore the access will be fast the cache can be. The execution of the store instruction can be very fast since it will complete as soon as the cache has been updated.

Now, obviously if this is the way that the cache is being updated then very soon the contents of the cache will become different from the contents of the main memory copy of the cache block. Therefore, they will be a need to keep track of which cache blocks have been modified and to ultimately write the contents of a cache block back to main memory, if the cache block is replaced from the cache.

This is a longer line of what we had talked about for virtual memory. This seems to have been the default policy that made sense for virtual memory and hence, this does not come as big surprise to us. This also explains why we are not surprised to see a dirty bit or modified bit associated with every cache block in the cache directory.

(Refer Slide Time: 35:09)



Write Policies

4A: When is Main Memory Updated on Write Hit?

- Write through: Writes are performed both in cache and in Main Memory
  - + Cache and memory copies are kept consistent
  - - Multiple writes to the same location/block cause higher memory traffic
  - - Writes must wait for longer time (memory write)

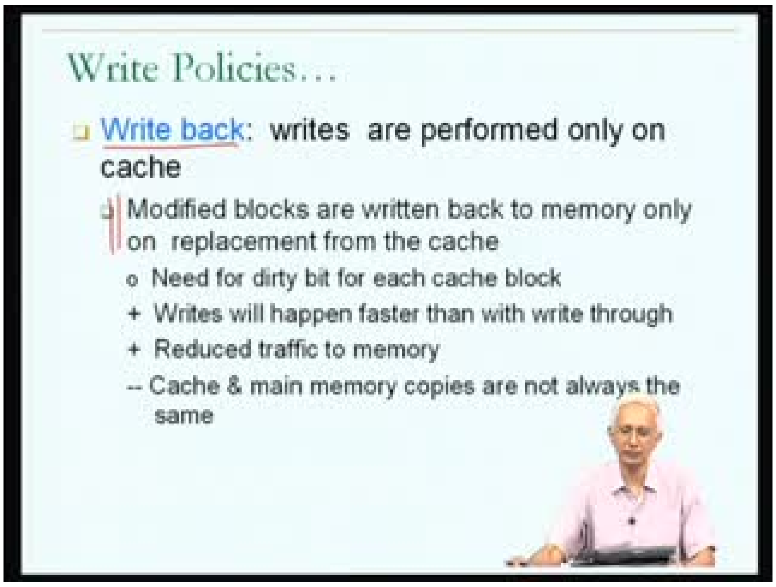
  Solution: Use a Write Buffer to hold these write requests and allow processor to proceed immediately

But, it does raise the question for us to quickly think about when we talked about question 4A in the context of virtual memory. We immediately came up with a solution which was write back and we never considered the write through as a solution. It even entered the realm of possibility, why was that? Now, if you think about the situation as far as virtual memory is concerned, bear in mind that we are talking about a situation where the virtual address spaces are actually maintained on disk. In other words, the actual contents of the virtual pages are available on a disk and at any given point in time some collection of subset of the disk pages, virtual pages will be present in the main memory.

Therefore, if we are concerned about the cache when we talked about updating a copy, we were talking about a copy in hardware cache and a copy in main memory. the hardware cache operates on a timescale of about 1 nanosecond. The main memory operates on timescale of about 100 nanoseconds and that is why we had the possibility of using write through because, there was only a 100 nanosecond delay. If one wrote through to main memory, but if you look back to our discussion of the virtual memory the situation was, we were not talking about cache and main memory, but about main memory which was on a timescale of 100 nanoseconds and disk which was on a timescale of milliseconds 1 or 2 or 3 milliseconds. This is the reason that we did not even consider an alternative to write back.

So, the idea of write through did not make sense in the case of virtual memory because if that was the case, then for store instruction which was not a page fault there would be a consequent disk access, meaning, a few milliseconds of activity as far as the implementation of this alternative to write back is concerned. Which is why, the write through was not even a feasible policy as far as virtual memory was concerned; but in the case of cache memory, it is not be ruled out they can be caches which are designed with a write through policy in mind.

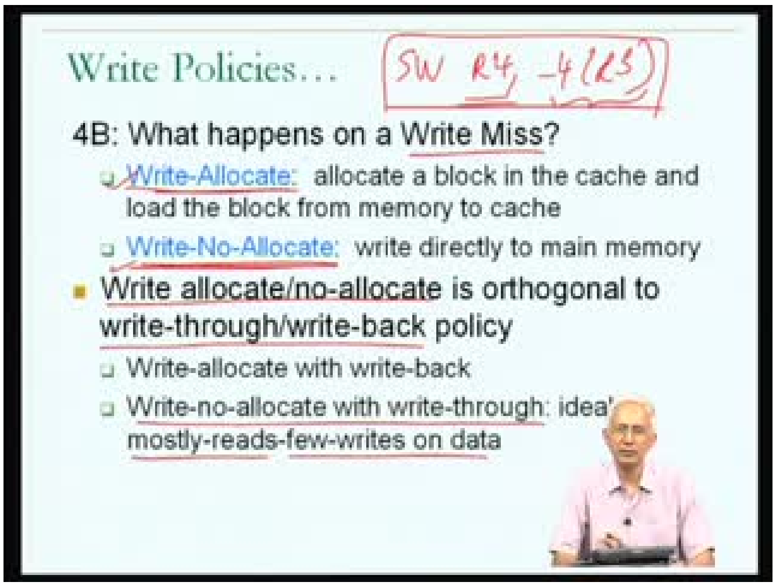
(Refer Slide Time: 37:09)



Write Policies…

- Write back: writes are performed only on cache
  - Modified blocks are written back to memory only on replacement from the cache
    - o Need for dirty bit for each cache block
    - + Writes will happen faster than with write through
    - + Reduced traffic to memory
    - -- Cache & main memory copies are not always the same

(Refer Slide Time: 37:21)



Write Policies…

$\boxed{SW\ R4, -4(R3)}$

4B: What happens on a Write Miss?

- Write-Allocate: allocate a block in the cache and load the block from memory to cache
- Write-No-Allocate: write directly to main memory

- Write allocate/no-allocate is orthogonal to write-through/write-back policy
  - Write-allocate with write-back
  - Write-no-allocate with write-through: ideal mostly-reads-few-writes on data

Basically, in connection with question 4A we see that there are these two strategies; one of which we have in fact seen before. The write-back strategy we have seen before when we talked about virtual memory and we know the positives and negatives of the write-back strategy from that discussion. Now, you recall that in the 4 Qs, we had one more issue regarding write policies and that was question 4B what happens on a write miss and once again, just remember we are talking about a store instruction. When the store instruction is executed, store instruction is requesting a particular memory location to be updated with a particular piece of data and when the store instruction is executed, it is determined that there is a cache miss. In other words, the required piece of data, the require block is not present in the cache.

Now, the question is, how could the cache be designed to handle this situation? The answer is the cache could be designed in a few different ways. One possibility is that the cache could be designed so that, when there is a write miss it allocates a block in the cache and loads the copy of the block from main memory into the cache as you would expect. In other words, proceeds as for a read miss. In the case of a read miss, the replacement happens and the copy of the block is brought from main memory into the cache. So, something similar could be done in the case of the write miss as well.

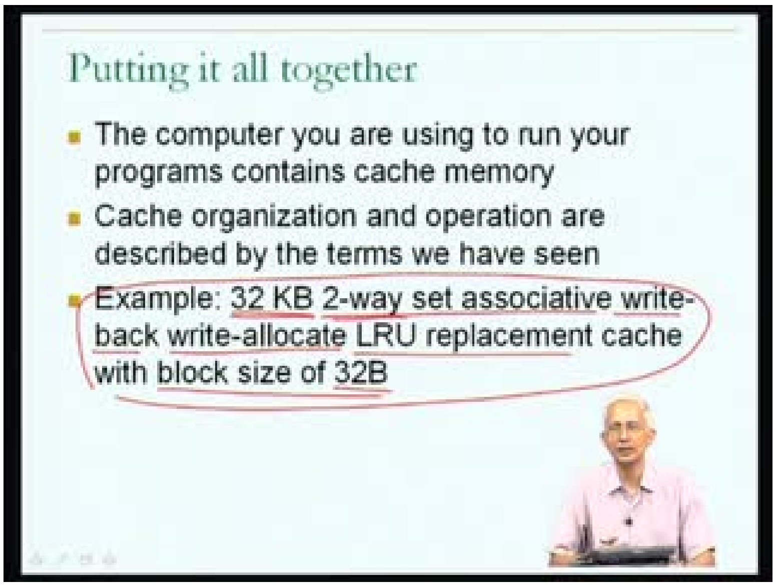
Now, there is an alternative; in the alternative is what is called write no-allocate. The idea of write no-allocate as the name suggests, is that we if there is a cache miss on a store instruction then, rather than going through the work of possibly replacing a cache block and then copying a block from main memory into the cache and then updating executing the store instruction to update the copy of the block inside the cache. One could just immediately update main memory and do not bother to fetch the block from the main memory into the cache at all. In other words, do not even allocate a block for this particular store miss inside the cache. Just update the main memory and you notice that this does make sense from the perspective of the time involved in the execution of the store instruction because, here we are talking about a write miss which means that if I was to use the write allocate strategy, there would be a need to fetch the block from the main memory into the cache which is going to take a 100 nanoseconds or more anyway. Therefore, in the case of the write no allocate strategy, the ideas is to, if you are going to do the 100 nanoseconds memory access anyway, then just update main memory write a way and do not bother to fetch the copy from main memory into the cache at all. This might make sense for certain kinds of programs but, this makes more sense for other programs.

You can clearly see that if, I have a program which frequently modifies variables then, it might make more sense to do write no allocate but, if I have a program which modifies the variable and then reads the same variable then, it might make sense for me to use the right allocate strategy. Therefore, for different kinds of program behaviors, it might turn out that one or the other of these makes more sense and at the time that hardware designers designing a cache, he cannot try to take into account the different kinds of program behavior. Therefore, we have to choose one or the other based on its best understanding of what kinds of programs are likely to run on the hardware that he is designing.

Now, the one important thing to note in connection with the answers to question 4B from our 4 Qs of cache design are that the write-allocate or write no-allocate decision from the perspective of the cache designer, is actually not closely related to the decision from question 4A. In other words, the write through or write-back decision and that in effect, one could have cache is which use write allocate with write-back.

Write no-allocate which with through etcetera we need different possibilities. Therefore, certain kinds of frequently occurring program behavior for example, if there was a situation where data is typically read but in frequently written, then a policy which might make lot of sense is write now allocate with write-through and so on for different kinds of typical kinds of program behavior. One could think of combinations of the answers to question 4A and question 4B which seem to be sensible and this is the kind of reasoning that a hardware designer would take into account.

(Refer Slide Time: 41:31)



Putting it all together

- The computer you are using to run your programs contains cache memory
- Cache organization and operation are described by the terms we have seen
- Example: 32 KB 2-way set associative write-back write-allocate LRU replacement cache with block size of 32B

Now, just to put together everything that we have said about the 4 Qs of cache hardware, let me just describe it briefly this way. Ultimately, when you run a program that you have written on a piece of hardware on a computer, you are aware now that a computer uses cache memory and that therefore frequently, when your program fetches an instruction or accesses piece of data, it will be able to fetch the instruction or access the piece of data using the cache memory rather than having to get the instruction or the piece of data out of main memory. Thereby, the operation may happen in a nanosecond rather than in 100 nanoseconds.

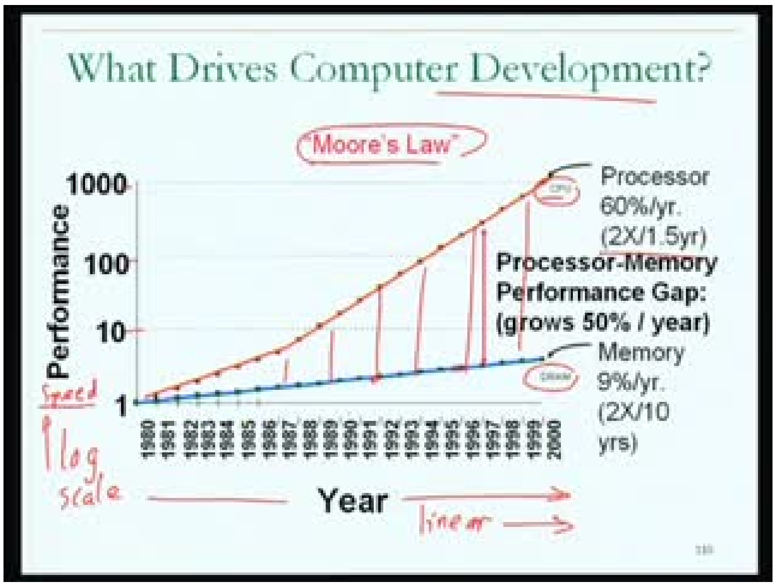
Further, we understand that the kinds of decisions which are made by the hardware within the cache memory are described by the answers that we saw for the 4 Q questions. The cache organization and operation are described by the terms that we have seen and therefore, if you are suddenly told that your program is running on a computer which has let us say, a 32 kilobyte 2-way set associative write-back write-allocate LRU replacement cache with block size of 32 bytes, you realize that you have actually got an answer to all of the questions that we had raised in the 4 Qs in this brief description.

We have here a description of the size of the cache. The cache is of size 32 kilobytes; we know what the size of each block is, we know the placement strategy it is 2-way set associative.

We know exactly how the replacement is done; we know the answer to question 4A; we know the answer to question 4B. In short, we know everything that is to know how about this cache operates and therefore, we should have from this clear understanding of how our program will be affected by this particular cache.

So, the key to understanding, how to make a program better from the perspective of cache behavior is, to start off with clearly understanding, would each of these terms means and we have sort of tackled that problem by understanding this in the light of the frame work of the fork use of cache organization and along the way we have look at a few examples of how these numbers come into play in deciding where in the cache a particular instruction or particular piece of data might be present and we will now leverage this in trying to look at code examples to get a clear understanding of what is happening.

(Refer Slide Time: 43:50)



Now, before proceeding to understand how we can improve the quality of our programs we saw the behavior in connection with cache memory and like to just back up of little bit in give you a sort of (( )) what is happening in the world of computer hardware by trying to address this question question of what drives computer development and this is the question which you may have often thought about in the light of the news papers.

When you look at the ads for computers in newspapers, you notice that there is a constant churning and new ideas which seem to be coming up more and more powerful processes bigger and bigger memories etcetera. New technologies which are being developed and may be it would help us to understand a little bit about what is driving this development. Often when people are asked this question they will give an answer based on something called Moore’s law and I will just give you some rough idea about Moore’s law. I am not suggesting that we need to understand; this to improve the quality of our programs.

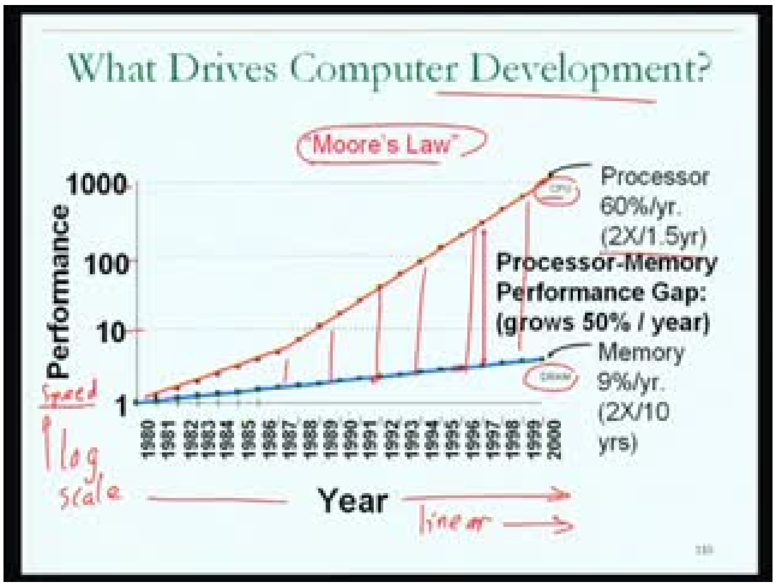
But, just to get a general idea of what has driven computer development in the past. Now, the Moore’s law could be described briefly; in words or could be described using a graphical representation of a certain trends which have been observed in the development of computers typically people would draw this by having a graph in which there is an x axis which is time.

This particular x axis is showing time from the 1980s onwards so 1980 on the left of the x axis and going into the 2000s on the right and on the y axis, what I have shown is an axis label performance - something to do the performance we could also label this as speed.

Since, we know that right now when we talk about performance, we are concerned about how fast things are and the important thing to notice that the x axis is a linear axis in other words in stepping from 1980 to 1981 and for 81 to 82 in other words for every year along the timeline there is an equal spacing along the x axis whereas on the y axis I have what is called the log scale. a logarithmic scale the different between 1 and 10 just the same as the distance between 10 and a 100 or between a 100 and a 1000

So that is what is called a log scale logarithmic scale; so, just note that the y axis is on a log scale and the x axis is on a linear scale. Now, what is being charted on this graph? The 2 lines which have been plotted one is the line which you can see in small is labeled CPU.

(Refer Slide Time: 43:50)



So that has something to do the speed of CPUs and the other is labelled DRAM. We have come across the word RAM before RAM has something to do with memories so we suspect that the second has something to do with the speed of memories and in fact that is roughly what this diagram is meant to illustrate something about the speed with which processes were becoming faster in the time period between 1980 and the year 2000 and the speed with which or the rate at which memories are becoming faster.

So we talk about the speed of a processer possibly by how fast it can execute a program we might talk about the speed of a memory by in terms of how much time it takes to access it therefore the small of the amount of time the faster the memory is.

Now the important thing to know from this is that memories and processors have both been becoming faster and faster. In other words, the circuitry underlying the processors and the memories were both becoming faster and faster. Even though this was a logarithmic scale, the speed was linear which means that in effect, the growth is much faster than linear; it is actually an exponentially fast growth.

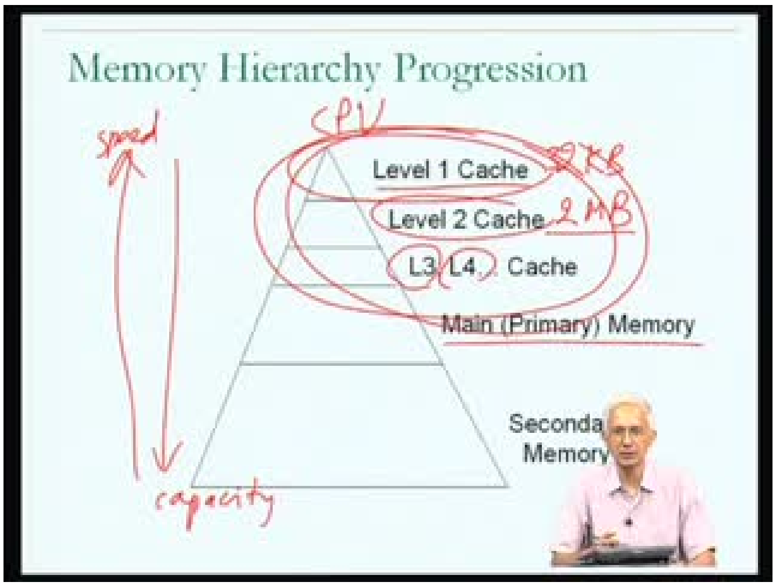
Bear in mind that this is log scale and people (( )) quantify this in those periods by saying something like processors were going faster. They were approximately doubling in speed every 18 months or every year and a half and in that time period you did frequently see that if you looked at advertisement for a computer and then 18 months later you looked at the advertisement of current computers as available. The current computers were typically twice as fast than the computers from 18 months before and that was a trend that had been observed and was for described as Moore’s law.

Now, the important thing from this particular graph is to notice that while memories also were growing from fast at very fast rate, linear on this logs scale, they were not growing fast as fast as processors were going fast. In other words, the speed disparity between processors and memories was in fact growing as time went on.

We talked about a 100 nanoseconds speed disparity between the timescale of the processor and the timescale of memory and that may have been in the year 1991 and a few years later, you may find that the disparity has grown that is what this graph is unfortunately telling us. So, in doing the time frame of this diagram, memories were growing fast. Every 2 years they were growing twice as fast in a 10 year period while processors were growing twice as fast in one and half year period and this was the scary message of Moore’s law.

Now why do I describe this as what drives computer development? Very clearly, both processors and memories were growing faster and faster, thanks to this kind of projection and people anticipated that this trend would continue and therefore, more and more exciting ideas were put into the design of the processors and the memories. But, in addition, it had to be borne in mind that the projection suggested that the speed disparity between the processors and the memories would itself keep on growing.

(Refer Slide Time: 49:37)



So, the processor memory performance gap was growing at a fairly fast clip. What this means in terms of our understanding of the memory hierarchy inside a computer system?. Now, if I told you that one and half years ago processor memory speed disparity was 100 nanosecond and then I told you that 2 years from now, that maybe 1000 nanoseconds then, you rationally have to reassess what a reasonable memory hierarchy might be 2 years from now.

If a memory hierarchy with a cache main memory and secondary memory is adequate today than 2 or 3 years from now, when the speed disparity between processors and main memories is growing more, then there may be need to have more layers in the memory hierarchy; more levels in the memory hierarchy to bridge the speed disparity with growing speed disparity between the processors and the main memories.

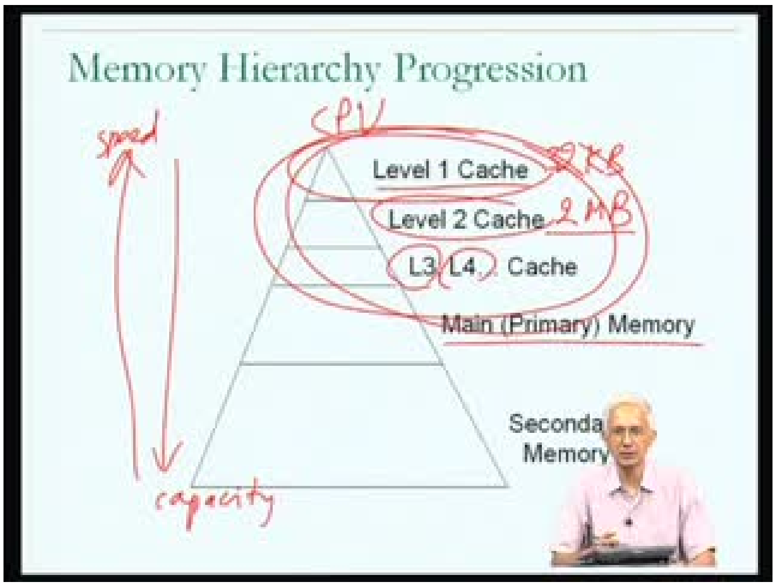
Which is why the main fact has been a situation where it became necessary to have not only a cache of long lines about we had before but, a smaller much faster cache in order to keep track of the speed disparity differences between the processor and the memories and as time went on it might be necessary to have even more levels of caches to bridge that speed disparity.

So, in some sense, one could see that if one has a picture of what is happening in terms of the reason that the rate at which the technology trends or driving, let us say the piece of different kinds of circuits where might be able to anticipate what kinds of changes may happen in a memory hierarchy. Today, this is not that unusual scenario where you have computer systems in which there are level one caches which are almost as fast operating a timescales close to the timescale of the processors might have level 2 caches which are much larger than the level 1 caches.

For example, the level 1 caches might just be about 32 kilobytes in size because, they have to be fast but the level 2 cache does not have to be so fast it could be a mega 1 or 2 megabytes in size. The idea of the level 2 cache is that it will be much faster than main memory has we understand from this a memory hierarchy diagram. The higher up you are the faster you are the lower down you are the more capacity you have the bigger you are.

So the level 2 caches would be bigger; they can contain a lot more blocks but they would be able to provide the data that is required at the level 1 cache in the event of a cache miss, much faster than main memory. Hence, they help to bridge the speed gap between the processor and the main memory and as a speed rap goes it may be necessary. It has level 3 and may be level 4 caches; 2 bits that speed gap in the interest of the performance of the program. The program is operating on a CPU which itself is becoming faster, was becoming faster and faster based on that Moore's law projection.

(Refer Slide Time: 49:37)



Now, with this we have a reasonable idea of what is happening behind the scenes. We will not take into account very complicated memory hierarchies in the discussion that we are going to get into; we will try to deal with somewhat simple scenario where they might just be 1 level of cache in order to understand the behavior of a program.

But, we need to actually start looking at examples of programs and understand how they interact with the cache, in order to understand the behavior of the program which is what we start doing in the next lecture, thank you.