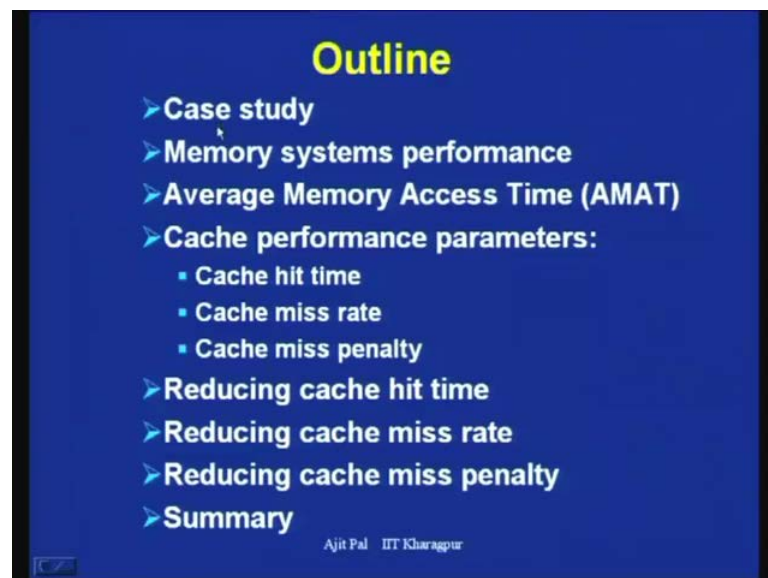


High Performance Computer Architecture
Prof. Ajit Pal
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 24
Hierarchical Memory Organization (Contd.)

Hello and welcome to today's lecture on Hierarchical Memory Organization. In the last couple of lectures, the important issues of hierarchical memory organization has been discussed, particularly we have focused on cache memory. Today, we shall start our discussion on how the performance of the cache memory organization be improved.

(Refer Slide Time: 01:25)

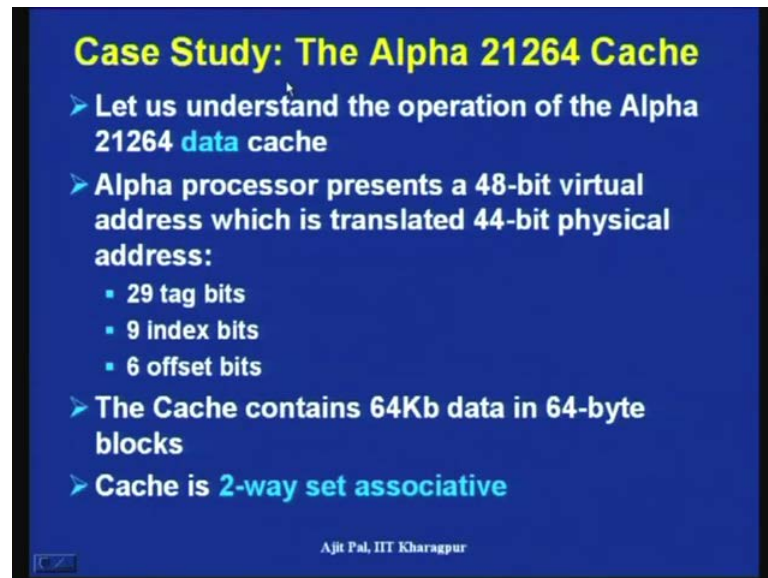


These are the, this is the outline of today's lecture, in fact these topics will be covered in a couple of lectures. First I shall considered a case study, which will histrolized the idea of hierarchical memory organization, and it will also highlight the important steps through which the memory access will take place. And then I shall discuss about memory system performance, that is the main topic of today's lecture. And we shall discuss introduced a concept called average memory access time, by which the performance of memory system is specified.

Then we shall see that, the cache performance is related to three important parameters, number one is cache hit time, second is cache miss rate and third is cache miss penalty. And obviously, to improve the performance of a memory system, hierarchical memory

system it will be necessary to improve one or more of these parameters, what you have to do, you have to reduce cache hit time, you have to reduce cache miss rate or and also you have to reduce cache miss penalty. So, we shall discuss various approaches that can be used for reducing these parameter values and we shall obviously, this cannot be covered in a single lecture, in few lecture we shall consider that.

(Refer Slide Time: 03:02)



Case Study: The Alpha 21264 Cache

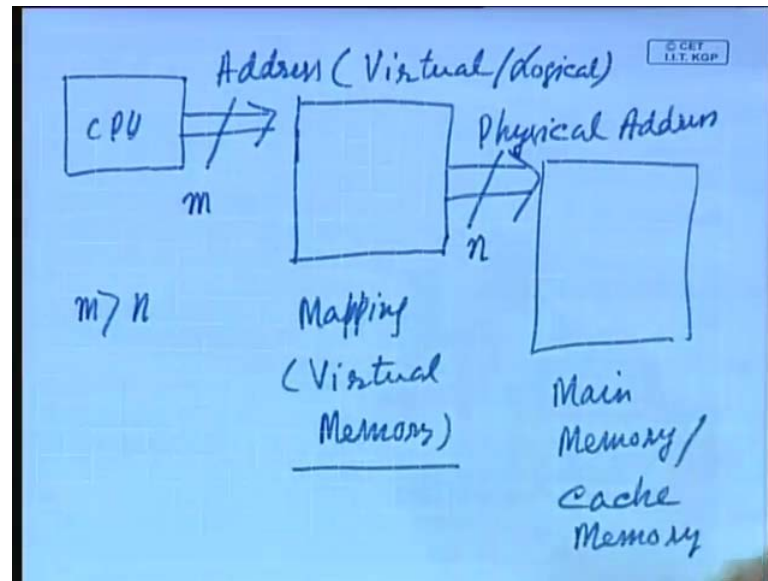
- Let us understand the operation of the Alpha 21264 **data** cache
- Alpha processor presents a 48-bit virtual address which is translated 44-bit physical address:
 - 29 tag bits
 - 9 index bits
 - 6 offset bits
- The Cache contains 64Kb data in 64-byte blocks
- Cache is **2-way set associative**

Ajit Pal, IIT Kharagpur

First let's consider a case study with the help of the Alpha 21264 processor's cache memory, particularly the data cache; as I have already mentioned in all present processors you have got two separate memories, particularly the L1 cache, there is separate data cache and instruction cache. So, we are focusing on data cache, the operation is somewhat similar in case of instruction cache, so it is not necessary to discuss both. And the Alpha processor presents a 48-bit virtual address, which is translated to 44-bit physical address.

So, this particular concept will be introducing detail, when we shall be considering virtual memory, but for today's discussion I mean, so that you can consider it properly. Let me briefly introduce what you really mean by virtual address and physical address. In fact, there are several levels of memory hierarchy, we have introduced only the cache memory hierarchy. And similarly main memory may also be considered another level of memory hierarchy.

(Refer Slide Time: 04:27)



So, in that situation what will be done this CPU, CPU generates an address, which we usually call virtual or logical address. So, we address that is generated by the CPU, is directly does not go to the physical memory, and that is the reason why virtual and logical address is introduced. Then this address is translated, there is some translation mechanism or mapping, that is done in the virtual memory system, this is translated to physical memory, physical address.

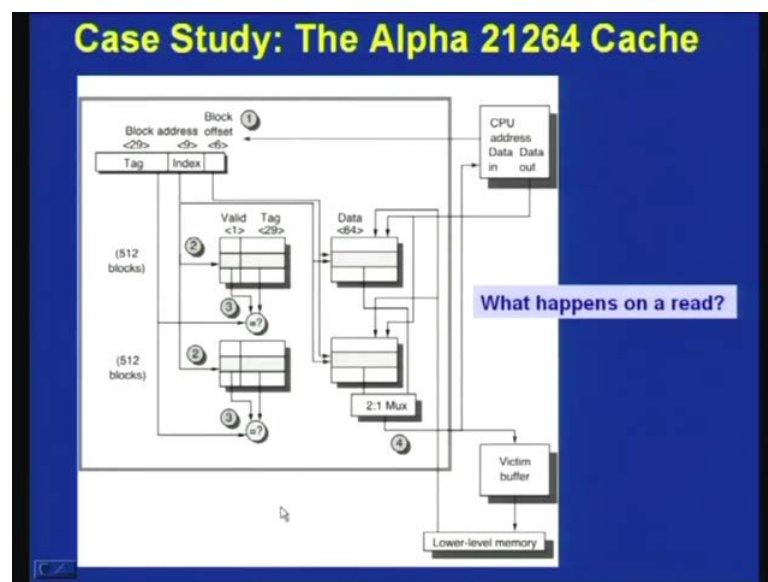
That means, that virtual memory system translates the virtual or logical address to physical address, and that address goes to your main memory. And since, nowadays we are using cache memory, the same address also goes to the cache memory. So, we find that there is a translation involved from virtual address to physical address and that physical address is used, so far we are focusing on cache memory. So, these aspects we have not discussed, this translation from virtual to physical address, which I shall discuss in detail later on, when we shall consider virtual memory concept.

So, in case of Dake alpha processor 48-bit virtual address is generated, then this is translated to 44- bit physical address, you seen the number of address lines that is present here that is present here may not the same. So, it may generate m-bit and it may generate n-bit, usually this m is greater than n, the reason for that is the size of the virtual memory is much more than the size of the physical memory. So, that is the reason why m is larger

than n , and the translation is done with the help of memory mapping, which I discussed later.

So, in this particular case, the translation is done, we are assuming that translation is done and we are getting 48, 44-bit of physical address comprising (Refer Time: 07:11) 29 tag bits, 9 index bits, 6 offset bits. So, the cache contains 64 Kilo byte of data, in 64-byte blocks. So, in case of Dake alpha processor, each cache the instruction cache and data cache, comprises 64 Kilo byte and the size of the block is 64-byte. And cache is 2-way set associative, so it is does not use direct mapping, it uses 2-way set associative mapping.

(Refer Slide Time: 07:48)



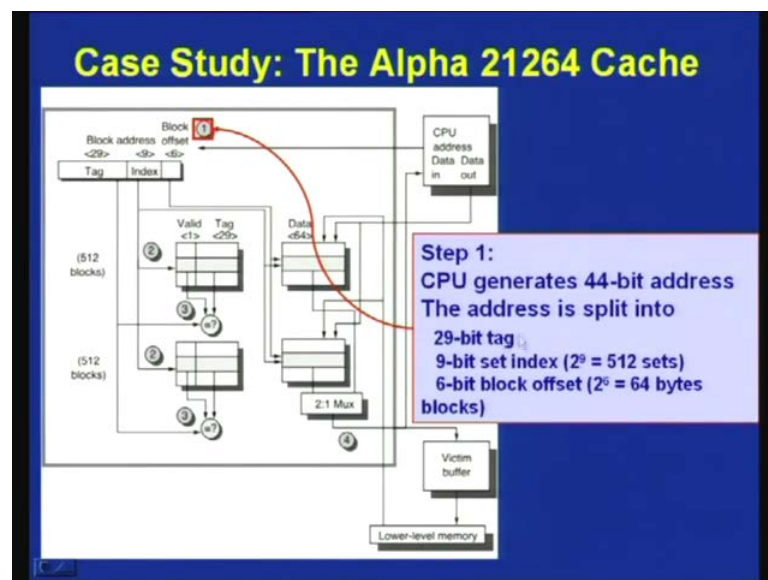
And with this background, let us now see the cache memory that is present particularly the data cache, that is present in your Dake alpha 21264, which is one of the very, one important processor where many innovative novel concepts where used, and which was one of the fastest processor when it was introduced, commercially made available. So, that is why he will find this particular process is referred in many case studies, because the new features, new techniques, innovative techniques where introduced in Dake alpha many such things.

Now, here as I have mentioned, this is the physical address 29 bit tag field, 9 bit index field and 6 bit offset field. And this since it is 2-way set associative, as you can see there are two blocks in a single set which are shown here, and since 64-bytes are present in

data that is also shown here. And you can see the tag and valid bit part and data part has been separately shown, particularly data exchange take place with the CPU and the memory, and the main memory.

On the other hand, these valid bit, tag field these are used essentially for the purpose of finding out, whether there is cache miss or cache hit for that purpose. And so the operation of read, read operation can be considered as taking place in four steps. So, I shall discuss about the four steps, by which the read operation is performed, so let us see how it occurs.

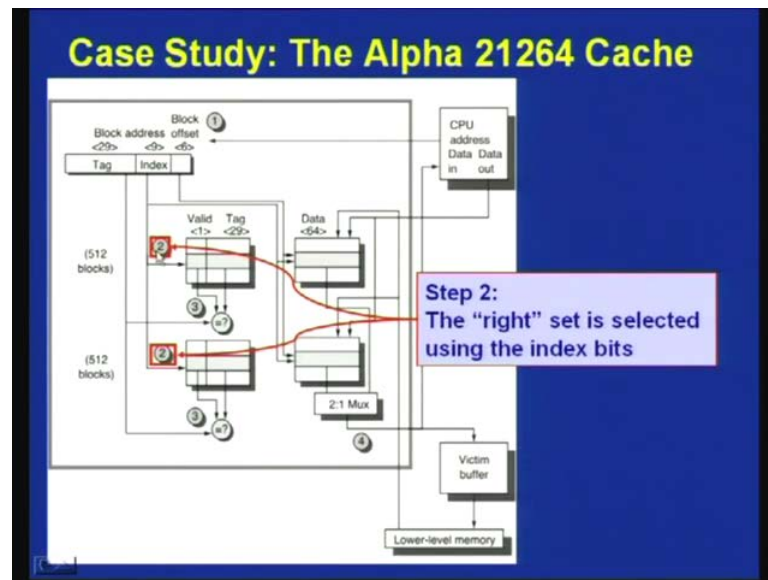
(Refer Slide Time: 09:46)



So, in the step 1, the CPU generates a 44-bit address, the address is split into 29 bit tag, 9 bit set index and 6 bit block offset. So, since you have got 2 to the power 6, 64 bytes for block, so you can get I mean 2 to the power, see you will get this figure 2 to the power 9 from the, if you divide 64 Kilo byte by this 2 to the power of 6 64 bytes, you will get this 2 to the power of 9 and also you have got two sets. I mean in a single set you have got 2 block that will provide you this 2 to the power of 9.

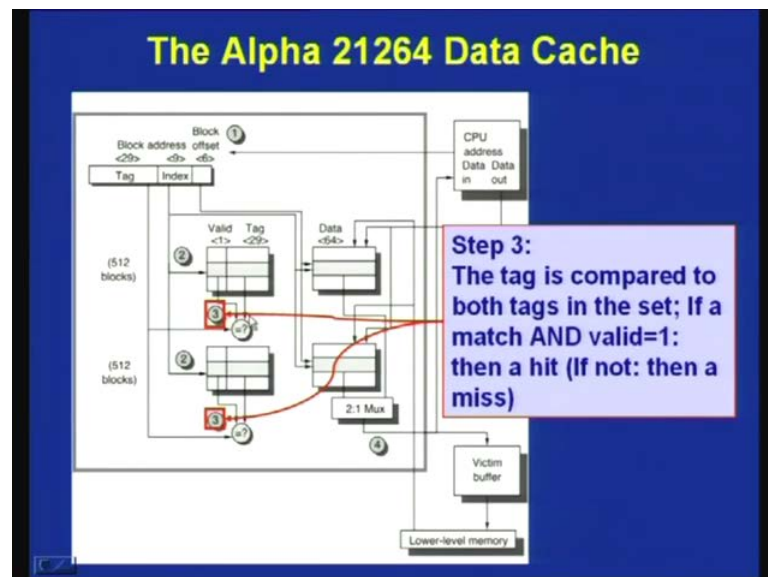
So, by this we are getting in the next field of 9 bits, so this block this address, this physical address is applied to the memory. So, it goes to the cache memory and that is performed step 1.

(Refer Slide Time: 10:50)



Now, as you go to step 2, as we know that index field identifies the right set by index field is applied here, and it identifies which particular set is related to this particular address, so this is done in step 2. And you can see this is pointing to the set, which has been identified by this index field, so the right set is selected using the index bits.

(Refer Slide Time: 11:23)

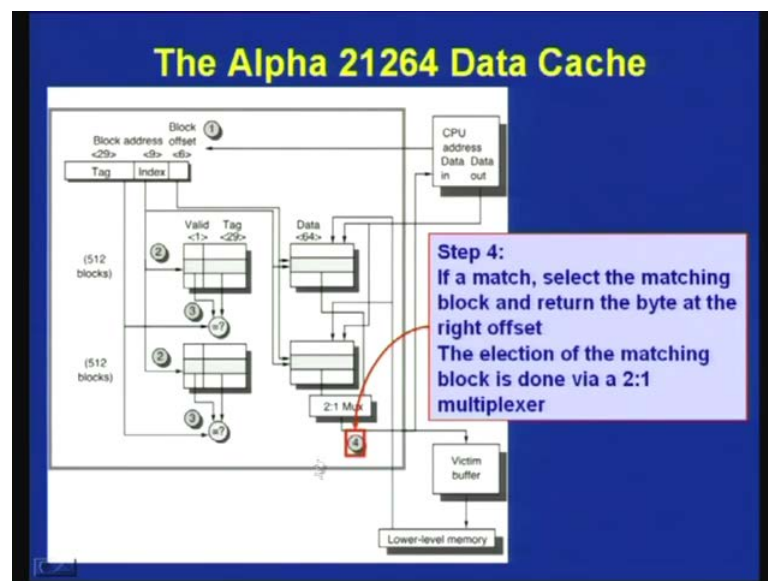


And then in the step 3, what you do the tag is compared to both tags in the set, as we know in case of 2-way set associative memory, we will have 2 tag fields, and you will required two comparators for the purpose of comparison. And these two comparisons

will take place in parallel that is what is happening here, this is one block and this is another block, and you can see the you have got two comparators where the comparison is taking place parallelly. And the tag is compared with both tags field, if a match then AND value bit is 1.

Obviously, your value bit has to be considered, because in the beginning the value bits are 0 and only as a consequence after the power is turned on value bit will be 0. And subsequently the data is transferred from the main memory to the cache memory, only then the value bit is 1. So, it is necessary that valid bit also should be considered for the purpose of identifying, whether it is a miss or hit that means, the valid bit is 1 and if by comparing the tag fields there is a match, then it is a hit otherwise, it is a miss as we know.

(Refer Slide Time: 12:41)

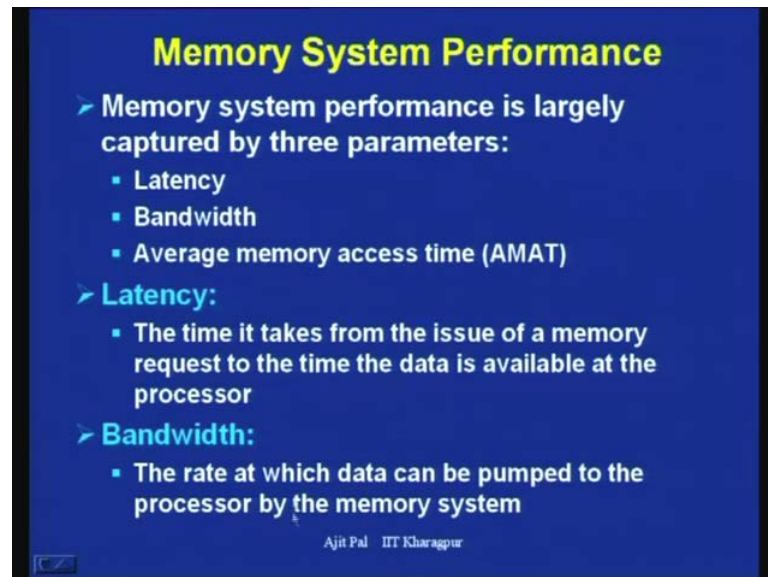


Then we go to the step 4, if there is a match, select the matching block and returned the byte at the right offset. So, you can see obviously, there cannot be matched with both, there may not be any match or if there is a match, then match will take place with one of these blocks. In a single set you have got 2 blocks, with 1 of the blocks there will be a match and using a multiplexer, you have to select the right block and that will be provide to the CPU, so here this CPU goes to the data in.

So, this is the case for read, for write it will be little different, because you have to write the data into the proper block, so that is little different from read. So, since we are

considering read that is quite simple, from here it directly goes to the CPU, so data is address in case of a hit. Of course, in case of miss, then you have to read it from the lower level of a memory and in such a case, it involve additional steps, let us keep those steps for the time being.

(Refer Slide Time: 13:57)



Memory System Performance

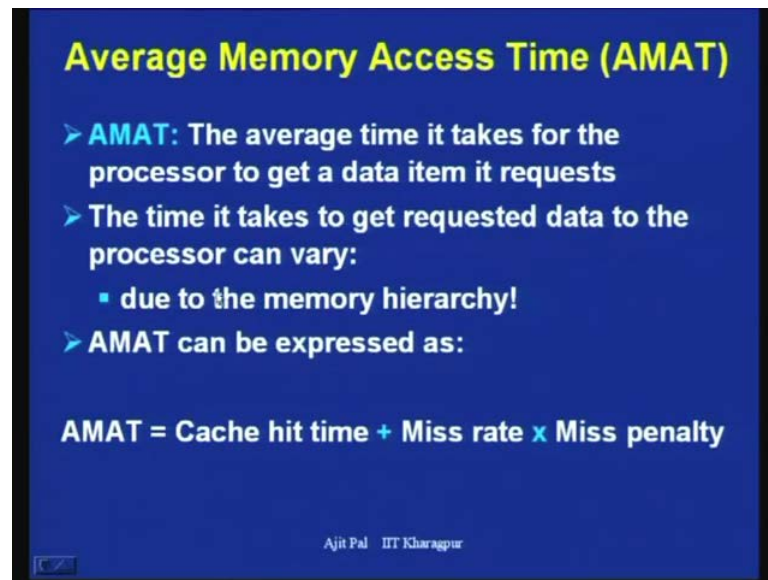
- Memory system performance is largely captured by three parameters:
 - Latency
 - Bandwidth
 - Average memory access time (AMAT)
- **Latency:**
 - The time it takes from the issue of a memory request to the time the data is available at the processor
- **Bandwidth:**
 - The rate at which data can be pumped to the processor by the memory system

Ajit Pal IIT Kharagpur

Now, let us focus on the memory system performance, based on the various discussion that we have done and the case study that I have considered in this lecturer. You see there are three important parameters, which captures the performance of a memory system. Number 1 is latency, number 2 is bandwidth, number 3 is average memory access time, what you really mean by these three parameters. First one is latency, the time it takes from the issue of a memory requests, to the time data is available at the processor. So, this is the latency, latency is dependent on various factors, like the technology that is being used, size of the memory and so on.

So, various whether it is on chip or off chip, all these factors will decide what is the latency, how much time it takes from the issue of a memory request to the time data is available on the processor. On the other hand, the bandwidth the rate at which data can be pumped to the processor by the memory system, is the bandwidth the rate at which the memory system can provide data to the processor, that again depends on the organization of the memory. Whether it is direct mapping, whether it is a set associative or 2-way set associative and other parameters that will decide the bandwidth.

(Refer Slide Time: 15:33)



Average Memory Access Time (AMAT)

- **AMAT:** The average time it takes for the processor to get a data item it requests
- The time it takes to get requested data to the processor can vary:
 - due to the memory hierarchy!
- AMAT can be expressed as:

$$\text{AMAT} = \text{Cache hit time} + \text{Miss rate} \times \text{Miss penalty}$$

Ajit Pal IIT Kharagpur

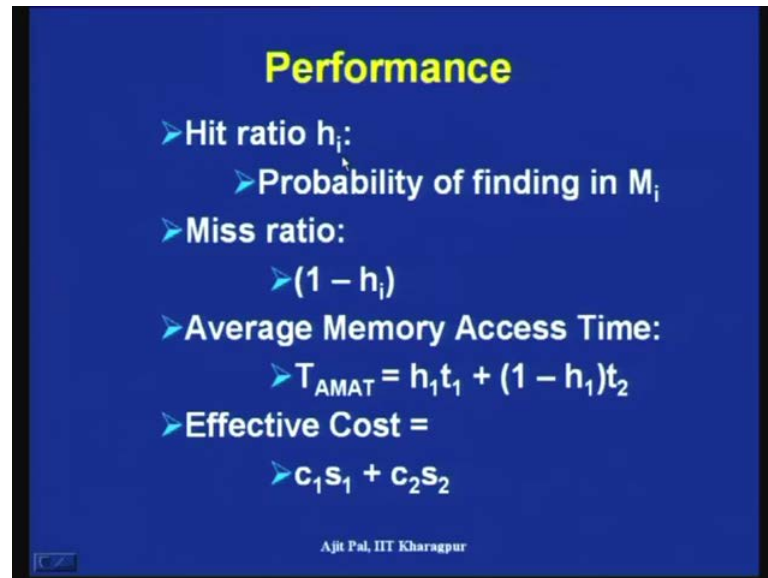
Then the third parameter that is your average memory access time, that is the average time it takes for the processor to get a data item it requests. So, your fetching instructions of data continuously, particularly instruction fetching takes place continuously one after the other, that is what the processor does after it is stand on, and the time that is required to get data or instruction from memory is not uniform. That will be dependent, whether it was hit, whether it was miss, how much time it is required for hit, how much time it is required for decide, whether it is an miss and what is the penalty, from where it will be read and so on.

And particularly this time it takes to get requested data to the processor, is a variable parameter. So, it is variable due to the memory hierarchy that is being used in the system, and the average memory access time can be expressed as I mean AMAT, Average Memory Access Time is equal to cache hit time plus miss rate into miss penalty. That means, whenever there is a hit, then the others two factors are not involved, it is decided by the cache hit time. So, cache hit time will be, whenever there is a hit he will get the data from the cache and cache hit time will be taken into consideration in case of hits.

But, whenever there is a miss than that miss will be decided by the rate, miss rate occur since we are considering the average, we have to consider the average number when there is hit and when a miss occurs that also you have to take into consideration, miss rate has to be taken into consideration. And also the miss penalty, whenever a miss

occurs, what is the penalty when a miss occurs, how many clock cycle is necessary to get the data, whenever miss occurs that is actually the miss penalty. So, these three factors parameters together, will give you information about the average memory access time.

(Refer Slide Time: 17:56)



Performance

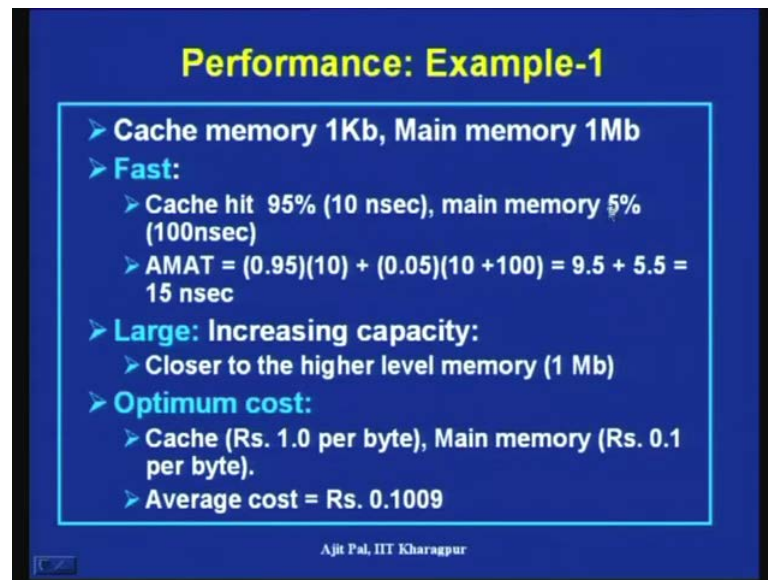
- Hit ratio h_i :
 - Probability of finding in M_i
- Miss ratio:
 - $(1 - h_i)$
- Average Memory Access Time:
 - $T_{AMAT} = h_1 t_1 + (1 - h_1) t_2$
- Effective Cost =
 - $C_1 S_1 + C_2 S_2$

Ajit Pal, IIT Kharagpur

And let us see with the help of an example, I mean let us see how it is been done in terms of probability; so let us assume hit ratio h_i that is the probability of finding in a hierarchical memory system in the level i . So, miss ratio will be equal to 1 minus h_i and average memory access time considering we have got only one level of memory hierarchy, that is equal to h_1 into t_1 . That means, h_1 is the hit ratio for the level L_1 , and access time for the level L_1 and this is the miss ratio and obviously, the access time for the second level, so that you have to multiply and that will give you the average memory access time.

So, we are also interested in the effective cost, as I mentioned in the beginning our objective is to have a very large memory, as large as the largest capacity memory, it should be fast, as fast as the fastest memory at the same time cost should be optimum. So, that is why the effective cost you have to take into consideration, that is C_1 into S_1 plus C_2 into S_2 , C_1 is cost per byte and S_1 is the size of memory in the level L_1 ; C_2 is cost per byte in the second level and S_2 is the size of the memory in the second level.

(Refer Slide Time: 19:36)



Performance: Example-1

- Cache memory 1Kb, Main memory 1Mb
- **Fast:**
 - Cache hit 95% (10 nsec), main memory 5% (100nsec)
 - $AMAT = (0.95)(10) + (0.05)(10 + 100) = 9.5 + 5.5 = 15 \text{ nsec}$
- **Large: Increasing capacity:**
 - Closer to the higher level memory (1 Mb)
- **Optimum cost:**
 - Cache (Rs. 1.0 per byte), Main memory (Rs. 0.1 per byte).
 - Average cost = Rs. 0.1009

Ajit Pat, IIT Kharagpur

So, let me consider an example, let us assume cache memory is 1 Kilo byte, main memory is 1 Mega byte, so you can see three orders of magnitude, difference is there between the cache memory and main memory and in fact, it may be more. And how fast it is, let us see, so assuming that cache hit is 95 percent and 10 nanosecond is the access time of the cache memory. And obviously, the main memory the hit will be, I mean that would be whatever is the miss that is 5 percent 1 minus hit ratio that is your 0.95, 95 percent.

So, 5 percent is the main memory hit and that you have to multiply with the excess time of the main memory that is 100 nanoseconds, it has been assumed that cache memory access time is 10 nanoseconds, and main memory access time is 100 nanoseconds. So, the average memory access time is equal to 0.95 into 10 plus 0.05 into 10 plus 100, so as you can see whenever there is a miss you have to take the two times that means, the first of all you have to access the cache memory, then you have to access the main memory, so it will involve accessing both.

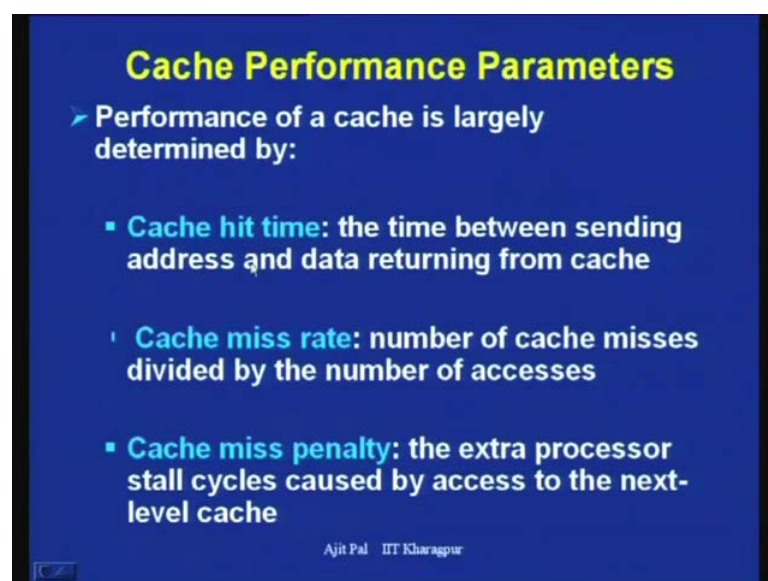
So, taking into consideration together you get average access time is 15 nanoseconds, so we find that this 15 nanoseconds is very close to the access time of the cache memory. So, it is closer to 10 nanoseconds than 100 nanoseconds that means, we are getting the average memory access time, which is close to the fastest memory in the system, that is one of our objectives. So, far as the size is concerned as I mentioned to the user, the size of the memory is equal to the size of the main memory.

So, he does not know whether cache memory is present or not, because he can write a program and which can be stored in the main memory, I mean as long as it can be accommodated in the main memory, it is good enough. That means, to the user the size of the main memory is important, and that is the capacity of the main memory is effectively the size the programmers and users visualize. So, that is closer to the higher memory hierarchy, so let us now focus on the cost.

So, assuming that the cache memory cost is rupees 1 per byte and main memory cost is 0.01 per byte, this is a typical example nothing to do with practical things. Because, as I mentioned with time the cost is decreasing, in the beginning I have shown your figure where shows that with time cost of main memory, cost of the cache memory all are decreasing. So, this is just a representative thing and so we should not take absolute values.

But, if you consider the average cost that is $C_1 \times S_1 + C_2 \times S_2$, you get average cost of 0.1009, so that is the average cost byte and you find it, that it is very close to the cost of the main memory, not the cost of the cache memory. So, we can see that our objective of fast, as fast as the highest the fastest memory, as large as the largest memory. And cost closer to the cost of the slower memory that is being achieved, and that is being achieved and that is clear with this example.

(Refer Slide Time: 23:24)



Cache Performance Parameters

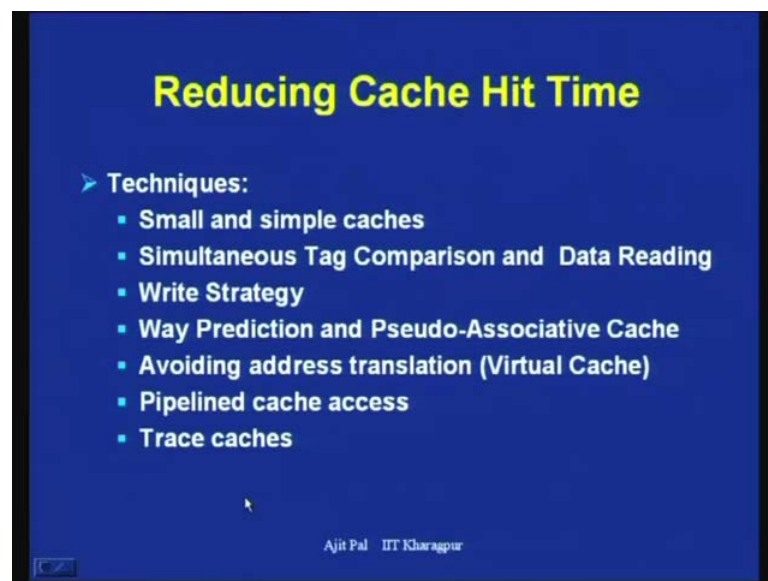
- Performance of a cache is largely determined by:
 - **Cache hit time:** the time between sending address and data returning from cache
 - **Cache miss rate:** number of cache misses divided by the number of accesses
 - **Cache miss penalty:** the extra processor stall cycles caused by access to the next-level cache

Ajit Pal IIT Kharagpur

Now, let us focus on the performance and as I have already mentioned performance of cache is largely determined by cache hit time, the time between sending of address and data returning from cache, cache miss rate, the number of cache misses divided by the number of accesses and cache miss penalty. The extra processor stall cycles caused by access to the next level cache, so we have to reduce these three parameters to improve the performance.

So, we shall focus on the cache hit time fast, how we can reduce the cache hit time, so we shall consider each of them one after the other. But, first let us consider the cache hit time, the reason for considering cache hit time, first is you see most as we know the probability of hit is much more 95 percent or it may be 99 percent depending on the size and other parameters. So, let us consider the common case first, common case is hit and whenever a hit occurs, what is the cache hit time and how can you reduce the cache hit time.

(Refer Slide Time: 24:45)

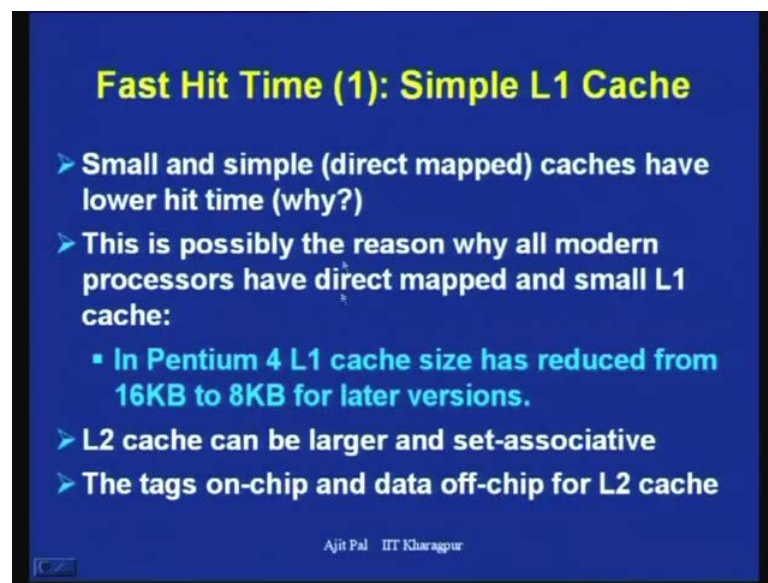


So, reduce the cache hit time, there are several techniques, many techniques have involved and even today research is going on, those who working in the area of computer architecture, and they are working on how the cache hit time can be reduce. So, I shall considered some of the popular techniques, which have been implemented in some of the real life processors. So, first technique is small and simple caches, second is

simultaneous tag comparison and data reading, third is write use of appropriate, write strategy.

Fourth is way prediction and pseudo associative cache, fifth is avoiding address translation which is known as virtual cache, then the last, but one is pipelined cache access and finally, trace caches. So, let us discuss these techniques one after the other, first one is small and simple cache.

(Refer Slide Time: 25:53)



Fast Hit Time (1): Simple L1 Cache

- Small and simple (direct mapped) caches have lower hit time (why?)
- This is possibly the reason why all modern processors have direct mapped and small L1 cache:
 - In Pentium 4 L1 cache size has reduced from 16KB to 8KB for later versions.
- L2 cache can be larger and set-associative
- The tags on-chip and data off-chip for L2 cache

Ajit Pal IIT Kharagpur

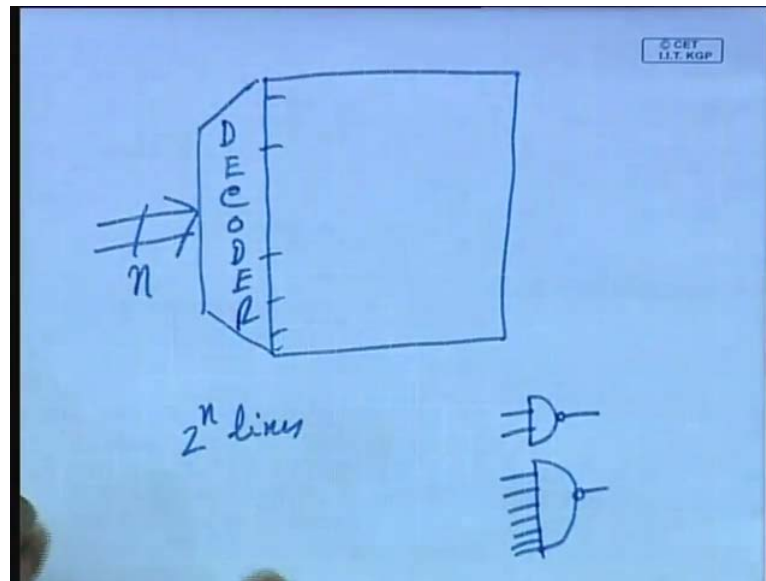
So, what you really mean by small and simple, by simple we mean direct mapping, as we have already seen that the cache memory is simple whenever we use direct mapping, the reason is for that is, in case of direct mapping you can use standard of the self memory. There is no need for modification of the memory, whenever direct mapping is used, it will receive address, then based on the access time, after that access time it will produce the data. Whether you access I mean store instruction of data it does not matter, it will produce instruction, if it is used for instruction cache data will be produced, if it is used for data cache.

So, you can use standard of the self memory and they are simple in organization, so the time required for the access that is access time will be smaller compared to other types of memory. Say if you use full associativity, you have to use content addressable memory which is quite complex and that content addressable memory, will involve comparison with the content of each and every memory location. And obviously, the parallel

comparison has to be done, and it will require a lot of hardware and whenever you put a lot of hardware it will involve more delay that means, access time will be longer.

And that is the reason why small and simple, simple is direct map then question of small, why do you say small, why the access time will be different for small and large memory.

(Refer Slide Time: 27:47)



The reason for that is, we will see that whenever we use a memory, the first step or first component that is in memory is the decoder, you are applying the address, so n bit address being applied. And this decoder will generate 2^n lines and the complexity of this decoder is dependent on n that means, if the size is large then this decoder will be quite complex, because it has to produce many lines and the hard work will be more and more complex. That means, the access time will be heavily dependent on the complexity of the decoder, and those who have attended a course VLSI circuit, they know that.

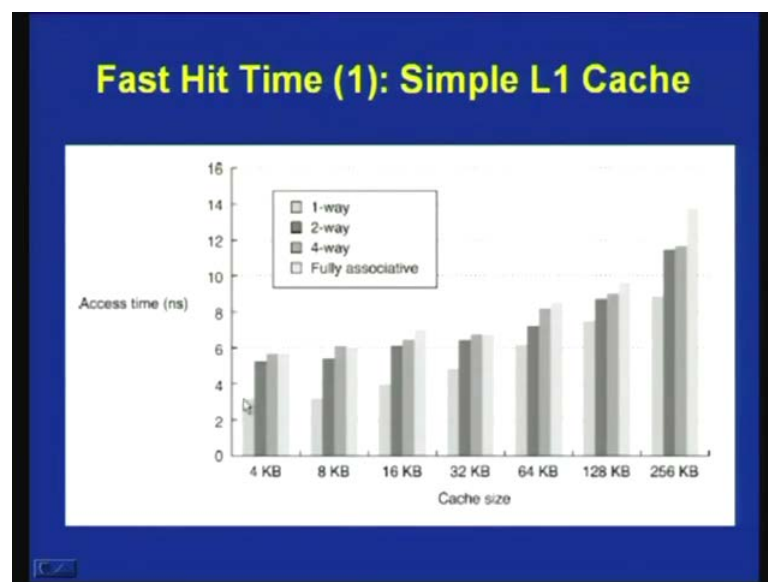
For example, the delay of a 2 input NAND gate is much slower than the delay of a say three input or say 8 inputs NAND gate, the reason I am not going to into the details of that. So, as the number of lines increases, the decoder will require gates with larger number of inputs NAND in and NAND out, you have to use multi level circuit to realize the circuit. In other words, the decoder will be taking longer time, so that is the reason why small and simple cache is advised, whenever we go for to reduce the hit time. So,

this is possibly the reason why all modern processors have direct mapped and small L 1 cache.

So, I have already given you some example, there you have seen that L 1 cache is always direct mapped and usually the size is small for example, in Pentium 4 L 1 cache size has been reduced from 16 Kilo bytes to 8 Kilo bytes for later versions. So, when it was introduced, they introduced with a larger size, but subsequently the size was reduced to reduce the hit time. However, there are L 2 cache can be larger and you can use set associative, 2-way set associative or 4-way set associative, and the tags on another technique that can be used the tags can be on chip, and data can be off chip for L 2 cache.

So, there are some relations in which to improve the performance, the tags are kept on chip, so that the comparison can be done quickly, since the tags are of the L 2 on chip on the other hand, the data cache is off chip, because you want larger size. So, that is the reason why, in the earlier diagram I have shown the tag size, tag part and data part separately, because the tag part may remain on the chip, on the data can be off the chip.

(Refer Slide Time: 30:58)

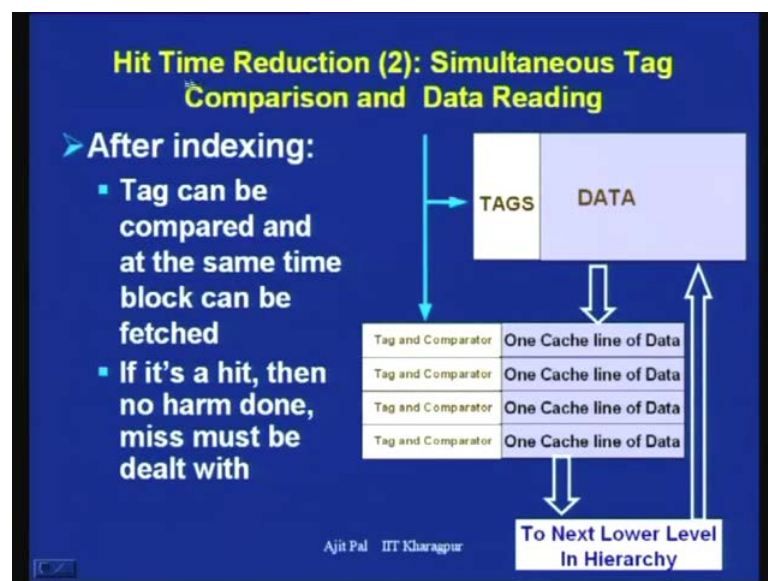


So, this diagram shown how the access time changes with the size as well as with the associative, so you can see this is the case for 1-way that is direct mapping. So, as you are increasing size from 4 Kilo bytes to 256 Kilo bytes, the access time is increasing, increasing from may be from 3 nanosecond to this one is roughly 9 nanoseconds. So, it is

becoming three times, as you were increasing the cache size from 4 Kilo byte to 256 Kilo bytes. And similarly whenever you are increasing the associativity, you are making a complex you can see, compared to direct map, the 2-way data associatively, this is 4-way set associativity, this is fully associativity.

You can see, as you are increasing the complexity, as you are going to for I mean for the same size higher associatively, the access time is increasing. So, that is true irrespective of size, so for all sizes for example, if you are having 256 Kilo byte of cache memory, this is the access time for direct mapping that is a little more than 8 nanoseconds. It becomes more than 11 nanoseconds for 2-way and 4-way data associative, it becomes goes to 14 nanoseconds for fully associative. So, you can see as the size increases, as the complexity increases, the access time increases that is the reason why smaller reason and the simple cache is advised in the L 1 cache reduce the hit time.

(Refer Slide Time: 32:39)



Now, another way of reducing the hit time is to performs simultaneous tag comparison with the data reading, we know that we have to first compare the tag to check, whether there, it is a hit or not miss. If it is a hit then we read the data from the cache, if it is a miss then data has to be valid from the main memory. So, that means, we do it sequentially, first tag comparison and if there is a match or hit, then we read from the cache memory otherwise, we read from the cache memory, so this is done sequentially.

On the other hand, to improve the hit time what you can do, to reduce the hit time what you can do, tag can be compared and at the same time block can be fetched. So, this is your cache memory, so you can do tag comparison and the data from the cache, memory can be read simultaneously. So, you can start initiate both of them together and if it is a hit there is no problem, so no harm done, however if it is a miss, then you have to undo this reading, and you have to read it from the main memory.

So, in any case you have to that will involve more time whenever you read it from the main memory, then from the main memory it will come, you have to load it in the cache memory and also you have to transfer it to the processor. So, you can see this is how the time reduction can be done by simultaneously that comparison and data reading.

(Refer Slide Time: 34:27)

**Reducing Hit Time (3):
Write Strategy**

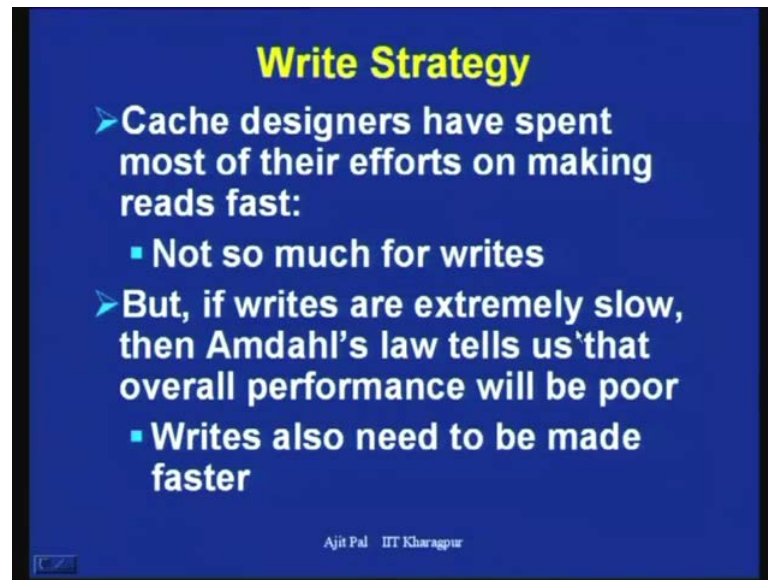
- There are many more reads than writes
 - All instructions must be read!
 - Consider: 37% load instructions, 10% store instructions
 - The fraction of memory accesses that are writes is: $.10 / (1.0 + .37 + .10) \sim 7\%$
 - The fraction of data memory accesses that are writes is: $.10 / (.37 + .10) \sim 21\%$
- Remember the fundamental principle: make the common cases fast

Ajit Pal IIT Kharagpur

Now, reducing hit time by using appropriate right strategy, so there are actually it has been observed that, there are many more reads than writes. So, for example, all instructions must be read, so and 37 percent are load instructions that is essentially read and only 10 percent are store instructions. So, you see that read is much more than the write, and the fraction of memory accesses the writes are only very small, so 7 percent. So, this is the 10 percent 10 by 1 plus 0.37 into 0.1 that gives you 7 percent that the fraction of data memory access, that are writes are also, this is the fraction of memory access for instruction is 7 percent, reflection of data memory access for write series is 21 percent.

So, you find that, for write the number of access is small, now that is the result why in the beginning more attention were focused on read, how to reduce the hit time which I have already discussed. So, for writing lesser attention was given however, you have to remember that fundamental principle, I mean make the common case fast that was done.

(Refer Slide Time: 36:03)



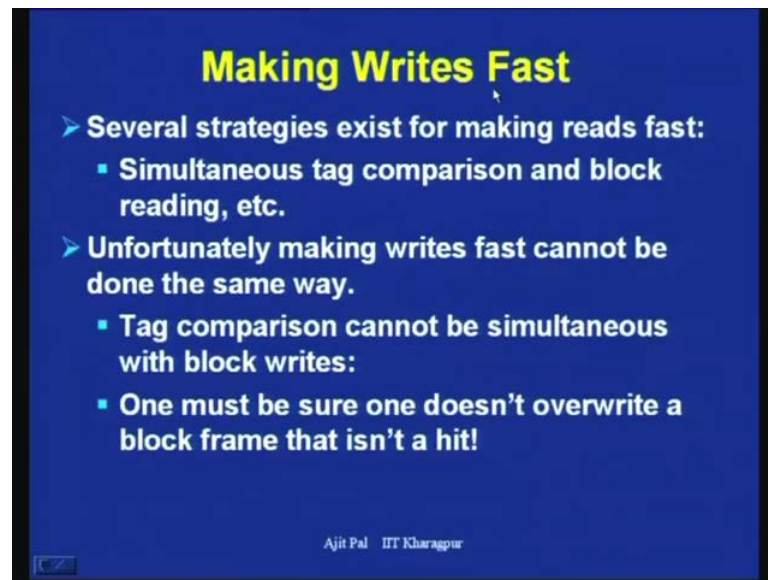
Write Strategy

- Cache designers have spent most of their efforts on making reads fast:
 - Not so much for writes
- But, if writes are extremely slow, then Amdahl's law tells us that overall performance will be poor
 - Writes also need to be made faster

Ajit Pal IIT Kharagpur

But, we have to focus on write, because if the writes are extremely slow, then Amdahl's law tells us that overall performance will be poor, so writes also need to be made faster. So, what I am trying to tell here instead of considering only the common cases fast, which is essential we have to focus on write although the number of writes is small.

(Refer Slide Time: 36:31)



Making Writes Fast

- Several strategies exist for making reads fast:
 - Simultaneous tag comparison and block reading, etc.
- Unfortunately making writes fast cannot be done the same way.
 - Tag comparison cannot be simultaneous with block writes:
 - One must be sure one doesn't overwrite a block frame that isn't a hit!

Ajit Pal IIT Kharagpur

So, let us see what are the, how you can really make that writes fast, so there are several strategies exists for making reads fast, which I have already discussed simultaneous tag comparison and block reading. Unfortunately these cannot be done for write, you see for read you can simultaneously read, I mean even not making any modification in the cache. So, if you read from the cache and do not use it anywhere, because if it is a miss not use it, you will ultimately write from the main memory. But, in case of write that cannot be done, reason for that is if you make any change in the cache and if it is a miss, then you have meant something which cannot be undone.

So, that is the reason why you cannot use this approach for write, so unfortunately making writes fast cannot be done in the same way. So, tag comparison cannot be simultaneous with block writes, so for block read it can be done, but it cannot be done for block writes. So, one must be sure one does not overwrite a block frame that is not hit, so as I have already explained, if it is a hit there is no problem, but in case of miss it will lead to problem.

(Refer Slide Time: 37:49)

**Write Policy 1:
Write-Through Vs Write-Back**

- **Write-through:** all writes update cache and underlying memory/cache
 - Can always discard cached data - most up-to-date data is in memory
 - Cache control bit: only a **valid** bit
- **Write-back:** all writes only update cache
 - Memory write during block replacement
 - Cache control bits: both **valid** and **update** bits

Ajit Pal IIT Kharagpur

So, let us see we shall consider two write policies, which are used one is write through another is write back, I have already introduced these two concepts earlier, in case of write through as you know all writes update cache as well as underlying memory and cache. So, this can always discard cached data, and most up dated data is in memory, so whenever you are using write through, you are making change in the cache memory as well as in the main memory.

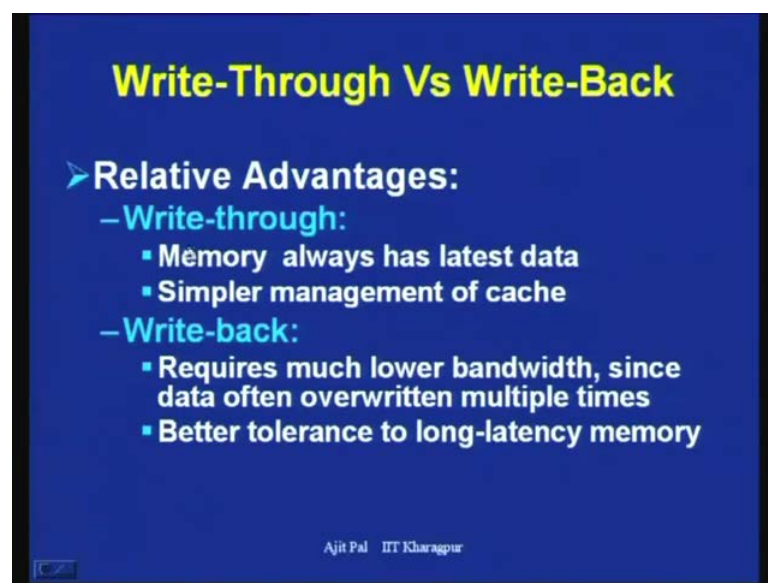
So, in such a case, there is no problem whenever you are doing write through, however the cache control bit in this case also you will require only a valid bit. But, whenever you going to write back all writes up date cache that means, in case of write back scheme as

I have already mentioned in such a case you are updating, writing only in the cache that means, you are having only cache memory and main memory. You are not modifying the main memory, you are modifying only the cache memory whenever you are performing write.

So, in such a case how do you keep track of this, because at the time of replacement you have to you have to write the modify the data from the cache into the main memory, you have to transfer cache memory to the main memory at the time of replacement. So, main memory write during the block replacement has to be performed, and you will require one additional bit, update bit for housekeeping purpose that means, along with valid bit as I explained in my earlier lecturer you will require one update bit.

So, if the update bit is 0, then whenever replacement is taking place, it is not necessary to write the cache copy, the cache into the memory, you can avoid that. On the other hand, update flag is set that means, you have modified the data I mean the cache content, so it is necessary to copy the updated data in the main memory. So, the update field has to be checked and you will require this housekeeping, I mean this additional bit for this house keeping purpose. And whenever you use this write back scheme, so this will definitely make the cache hit time faster; so write back scheme will make cache time faster, but with additional complexity.

(Refer Slide Time: 40:39)



Write-Through Vs Write-Back

➤ **Relative Advantages:**

- **Write-through:**
 - Memory always has latest data
 - Simpler management of cache
- **Write-back:**
 - Requires much lower bandwidth, since data often overwritten multiple times
 - Better tolerance to long-latency memory

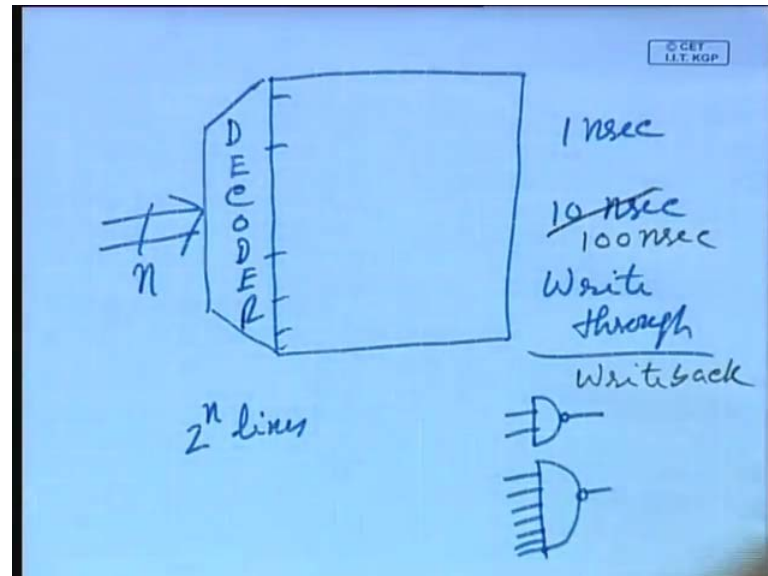
Ajit Pal IIT Kharagpur

So, relative advantage is that, write-through memory always has the latest data simple management of cache. On the other hand, write-back policy requires much lower bandwidth. So, it is faster since data often overwritten multiple times, and this is best on the observation that, once you transfer for a block of data from the main memory to the cache memory many times write will take place. Five each times write it back into the main memory, only when it is replaced then write back, that is what is being stated by this particular sentence.

And it gives you better tolerance to long latency memory, so whenever the memory has longer latency, that means main memory is very slow than each time you perform write, your time required for hit will be longer. On other hand, if you write back, then that will be smaller, because you are dealing with only cache memory, so it has got write back has

better tolerance to long latency memory. In fact, there is a kind of trade off, between the difference in access time of the two memory hierarchies, will it the difference is small.

(Refer Slide Time: 42:12)



Suppose, cache memory access time is only 1 nanosecond and main memory access time is 10 nanoseconds, not much difference, not 100 nanoseconds or 1000 nanoseconds, in such a case write through approach may be used on. On other hand, instead of 100 nanosecond, if it is a I mean 10 nanosecond, if it say 100 nanosecond or longer, then it is better to use write back.

(Refer Slide Time: 42:49)

Write Policy 2: Write Allocate Vs Non-Allocate

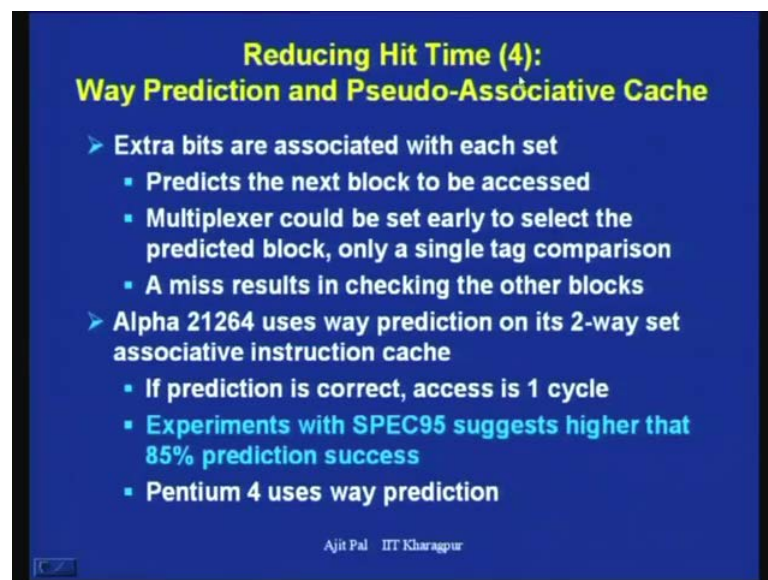
- **Write allocate:** allocate new block on a miss:
 - Usually means that you have to do a "read miss" to fill in rest of the cache-line!
- **Write non-allocate (or "write-around"):**
 - Simply send write data through to underlying memory/cache - don't allocate new cache line!

Ajit Pal IIT Kharagpur

Another write policy is write allocate versus non allocate, so what is the basic idea of write allocate, allocate a new block on a miss, we have seen that whenever there is a miss you have to transfer from the main memory to the cache memory. So, what you do, you use a separate block, new block, so each time whenever new block, whenever a write miss occur, you allocate a new block. And as you do that, this will imply that you have to do read miss to fill the rest of the cache lines, that means whenever you are using write allocate a block will have a number of words.

So, you have to read more number of words and whenever to fill the cache lines that will lead to read miss, so this is one approach. Another approach is write non allocate or write around, so this simply send write data through to underlying memory or cache is does not allocate new cache line. So, in such a case there is no problem, it uses that compensational approach write non allocate, so these two approaches are available just like your write through and write back. And depending on the application, depending on the requirement you can use one of them obviously, each of them has its own advantages and disadvantages.

(Refer Slide Time: 44:25)



Reducing Hit Time (4):
Way Prediction and Pseudo-Associative Cache

- Extra bits are associated with each set
 - Predicts the next block to be accessed
 - Multiplexer could be set early to select the predicted block, only a single tag comparison
 - A miss results in checking the other blocks
- Alpha 21264 uses way prediction on its 2-way set associative instruction cache
 - If prediction is correct, access is 1 cycle
 - Experiments with SPEC95 suggests higher than 85% prediction success
 - Pentium 4 uses way prediction

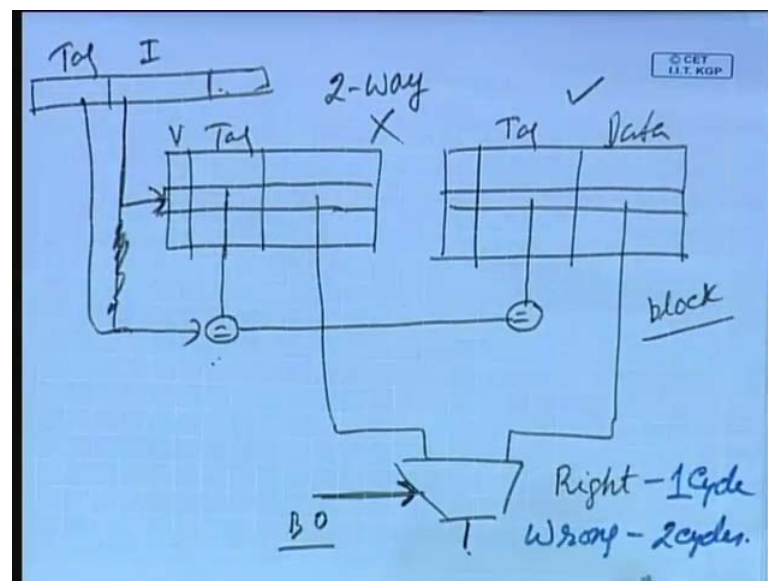
Ajit Pal IIT Kharagpur

Now, let us consider the fourth technique for reducing hit time that is by using way prediction and pseudo associative cache. You may recall that, we discussed about branch prediction and whenever we do branch prediction, what is being done we predict that a branch will be taken or untaken. And accordingly we proceed to fetch instructions in that

direction, in a similar way what is being done it predicts the next block to be accessed, so extra bits are associated with each set and it predicts the next block to be accessed.

So, in this way the prediction is done and as you do the prediction, then the multiplexer could be set early to select the prediction block, only and a single comparison is required, so this is in the context of 2-way or I mean set associative.

(Refer Slide Time: 45:34)



As we know you have got a single set will have more than one block, so you have got valid bit, you have got tag bit, you have got data. So, normally as you know, you will require comparison here, you will require comparison here, with the tag bit that is available from the, I mean tag field of the address that is coming from the processor. So, this will be compared and this will also be compared, this is the index field. So, index field is different, this is index, this is tag, so tag field has to be counted, now we find that we have to perform two type of comparison, if it is 2-way set associative.

And also you will require a multiplexer. this two data will go to a multiplexer and you have to select one of them. and you have to apply from this block up set. you have to apply the value, to select data to the appropriate data to this. So, this is how it will proceed however, if you do the prediction then you can apply this branch of set early, there is no need to wait for this company. Because, and also there is no need to perform two tag comparisons, if this is the predicted a way not this one, then you will only compare this one and you will only select this data.

That means, whenever you are assuming that this particular, I mean if it is a 2-way set this particular block will be used that is been predicted. So, you will fetch the data by predicting by selected this multiplexer and also only this comparator will be checked, not the other one. And if you have got say 4-way, then you can afford 4-way comparisons hence, if it is a 8-way you can comparisons, only one comparisons is required and you can apply the appropriate value here, you select quickly the data.

(Refer Slide Time: 48:13)

**Reducing Hit Time (4):
Way Prediction and Pseudo-Associative Cache**

- Extra bits are associated with each set
 - Predicts the next block to be accessed
 - Multiplexer could be set early to select the predicted block, only a single tag comparison
 - A miss results in checking the other blocks
- Alpha 21264 uses way prediction on its 2-way set associative instruction cache
 - If prediction is correct, access is 1 cycle
 - Experiments with SPEC95 suggests higher than 85% prediction success
 - Pentium 4 uses way prediction

Ajit Pal IIT Kharagpur

That means, in such a case multiplexer would set early to select the predicted block, and only a single tag comparison will be involved, as I explained a miss result in checking the other blocks. However, you cannot always expect that your prediction will be correct as you have seen in our branch prediction techniques. So, here we have the prediction may turn out to be wrong, and whenever it is wrong, then a miss can occur and that will involve checking of other blocks that is present in a particular set.

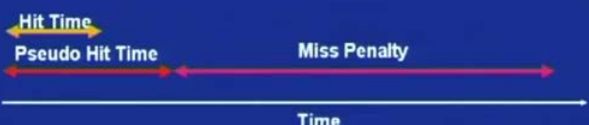
So, it has been used in L I processors 21264 uses way prediction on it is 2-way set associative instruction cache, if prediction is correct access time is 1 cycle. So, what is the capability let us have been look, that means if this prediction is right, then he will require access time of 1 cycle. And if it is wrong, then it will involve 2 cycle, because you have to consider only this block, other block also has to be compared now, whenever the prediction turns out to be wrong. So, that will involve 2 cycles, because that Dake alpha has got 2-way data associative.

So, experiments with SPEC95 suggest that higher than 85 percent prediction can success that means, for 85 percent of the cases you will require 1 cycle, and only for 15 percent of the cases you will require 2 cycles for accessing the cache. So, this is a essential gain and that is the reason why this way prediction technique is used, not only in Dake alpha 21264, but also in Pentium 4 processor.

(Refer Slide Time: 50:13)

Pseudo-Associativity

- How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?
- Divide cache: on a miss, check other half of cache to see if there, if so have a **pseudo-hit** (slow hit)



The diagram illustrates the timing of different cache access scenarios on a horizontal 'Time' axis. A yellow arrow labeled 'Hit Time' represents the shortest duration. A red arrow labeled 'Pseudo Hit Time' is longer than the hit time. A long red arrow labeled 'Miss Penalty' starts after the pseudo hit time and extends significantly further, representing the time taken to access main memory after a full miss.

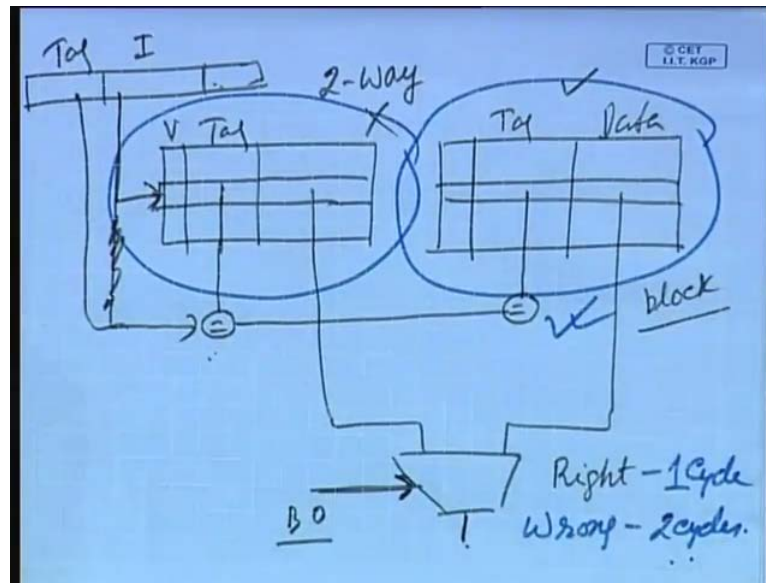
- Drawback: CPU pipeline would have to use slower cycle time if hit takes 1 or 2 cycles
 - Suitable for caches not tied directly to processor (L2)
 - Used in MIPS R1000 L2 cache, similar in UltraSPARC

Ajit Pal IIT Kharagpur

Now, let us consider another approach that is known as pseudo associativity, in case of what do you really mean by pseudo associativity, we have seen that, the direct mapping gives you fast hit time. On the other hand, 2-way set associative gives you slower access time, but it give you lower conflict misses. That means, it gives you higher performance heat will be more, in case of 2-way set associative, how can we combine the advantages of both that is been done in case of your 2-way set associative approach.

So, what is been done it divides the cache into two parts, so on a miss check the other half, of the cache to see if it is there, and if so you have got pseudo hit that means, if it is a hit you have 2 blocks.

(Refer Slide Time: 51:16)

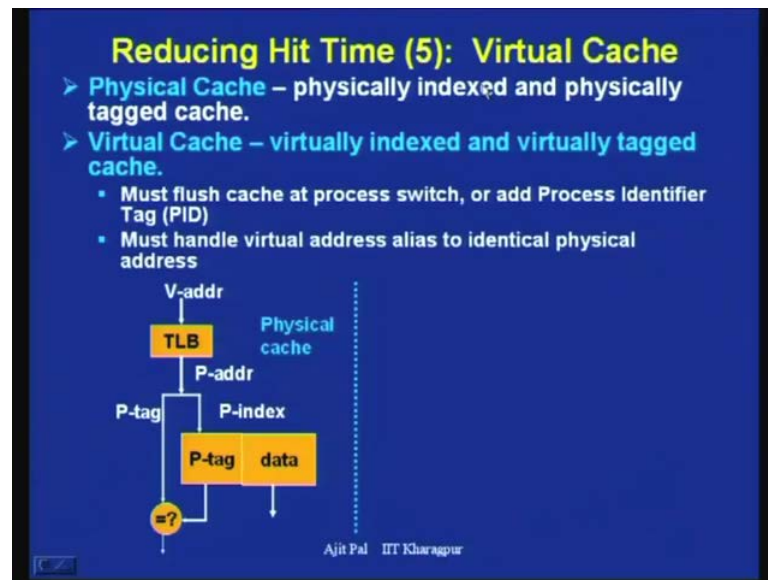


Let us consider instead of 2-way associativity, we have divided into two parts, this is one half and this is another half, you check this half first. And if it is hit, then your hit time will be smaller, as you can see hit time will be represented by this, this is smaller and if it is hit, then there is no need to check the other half for the cache. But, if it is miss then you have to check the other half of the cache and that will lead to additional half known as a pseudo hit time and of course, there is a possibility that it may not be hit is given in the other half that means, the data may not be present in the cache.

In such a case it will involve miss and that will lead to miss penalty, so drawback is that CPU pipeline would have to use slower cycle time, if it takes 1 or 2 cycles, as we know the clock frequency, the clock rate of the processor is decided by the speed of the memory. And since it can be either 1 clock cycle or 2 clock cycle that means, the processor accordingly has to be organized and the CPU pipeline cycle time has to be adjusted. So, this is suitable for caches not tied directly to the processor.

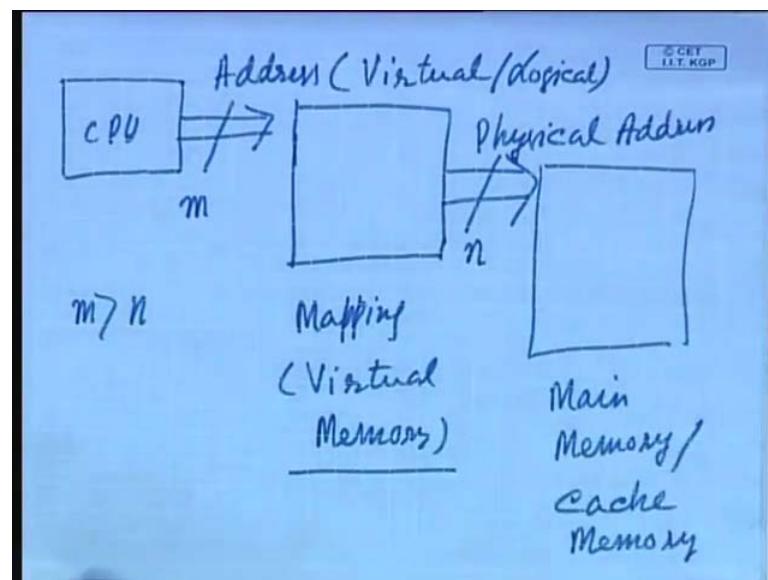
That means, this is not suitable for the on chip cache, it is suitable for off chip cache and that is the reason why it is used in MIPS R 10000 in L2 cache and also it is also used in ultra spark processor.

(Refer Slide Time: 53:04)



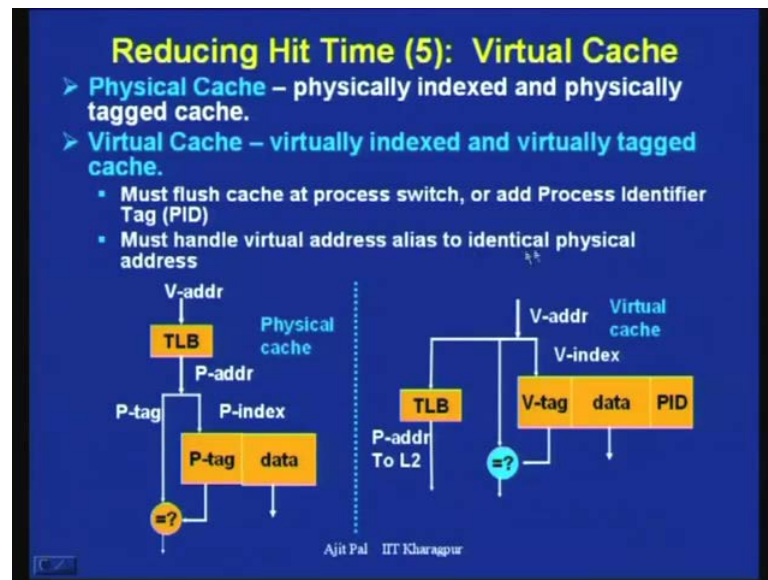
So, you can see how you can use reduce hit time by using a concept of virtual cache, as I have already mentioned that, we can use we have this virtual memory, and virtual address and physical address.

(Refer Slide Time: 53:19)



Now, if we assume that the same address will be used for the physical address, then it is called virtual cache.

(Refer Slide Time: 53:31)



So, in case of physical cache, physical indexed and physically tagged tax, we can see this is the physical cache. So, you have got virtual address and a translation is done from to generate the physical address from the virtual address, will be help of a hardware, known as translation look aside, which I shall introduce later on. So, this translation look aside buffer does translation, we get the physical address and physical address is used for the purpose of tag comparison and also for the purpose of gaining the address. So, that physical index address part is used and tag field is also used.

So, this is the physical cache, now in case of virtual cache what is being done, you can see we are using the virtual address for the purpose of indexing, and also for the purpose of comparison of the tag field. So, virtual index and virtually tagged cache, in such a case we must flush cache at process switches that means, whenever there is a context switching, process switching takes place. In such a case you have to flush the cache memory and that is taken care of a with the help of a additional bits, known as process identifier tag PID.

You have to add PID field as part of the cache memory, so this also must handle virtual address or aliasing to identical physical address, this aliasing approach arises, because sometimes the operating system for operating system an user you have got different virtual address, but the same physical address.

(Refer Slide Time: 55:27)

Virtual Cache

- Cache both indexed and tag checked using virtual address:
 - Virtual to Physical translation not necessary for cache hit
- Issues:
 - How to get page protection information?
 - Page-level protection information is checked during virtual to physical address translation
 - How can process context switch (different physical address for the same virtual address)?
 - How can synonyms (OS and user programs may use two different virtual address for the same physical address) be handled?
 - Solution: antialiasing by hardware, or by software

Ajit Pal IIT Kharagpur

So, in such a case problem arises, and that problem has to be overcome, so cache indexed and tag checked using virtual address. So, this is known as virtual cache, virtual to physical translation is not necessary in case cache hit, and there are several issues involved unique. First one is how to get page protection information, whenever you use virtual memory, you will see the page protection bits are available, read only, write only, access only, so for different users it can be different.

That type of production has to be provided page level protection information is checked during the virtual to physical address translation, this has to be done. And how to process context switch as I was scanning that you can have same virtual address, but they can be mapped to different physical memory at different instances of execution as, it happens in case of multi tasking in computer systems.

So, in such a case this has to be tackled, so how to process context switch that has to be address, different physical address for the same virtual address. Third is how can synonyms, properly as I told you that operating system and user program may have two different virtual address for the same physical address, and this has to be handled. And solution for this can be provided by hardware, and the approach of that hardware based approach is known as antialiasing and there is also software based approach, we are not going into details of that.

(Refer Slide Time: 57:09)

Virtually Addressed Cache

- Cache indexed using virtual address:
 - Tag compared using physical address.
 - Indexing is carried out while virtual-to physical translation is occurring
- Issues:
 - No PID needs to be associated with cache blocks
 - No protection information needed for cache blocks
 - Synonym handling not a problem

Ajit Pal IIT Kharagpur

And you can see for virtual address cache, cache indexed using virtual address, tag compared using physical address, index is carried out while virtual to physical translation is occurring. So, the various issues that I have told no PID needs to be associated with cache blocks, no protection needed for cache blocks and synonym handling is not a problem.

(Refer Slide Time: 57:32)

Virtually Indexed, Physically Tagged Cache

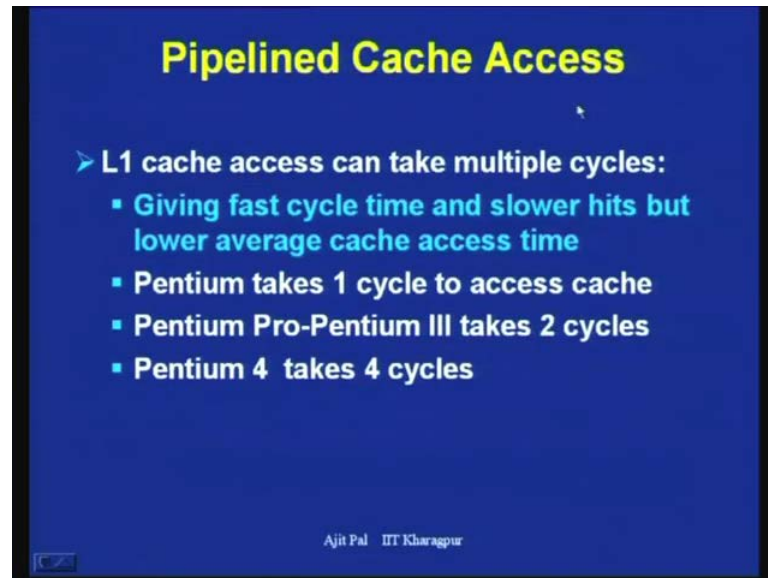
- Motivation:
 - Fast cache hit by parallel TLB access
- Issue:
 - Avoids process ID to be associated with cache entries

Ajit Pal IIT Kharagpur

So, this is the situation for virtually indexed, physically tagged cache, here the motivation is, so you can see virtual address, but physically addressed tag. So, fast cache

by parallel TLB access, so you are being parallel TLB access with the physical tag for reading the data. So, this avoids process ID to be associated with the cache identity, so you do not require the PID field here as you can see, so these are the techniques.

(Refer Slide Time: 58:07)



Pipelined Cache Access

- L1 cache access can take multiple cycles:
 - Giving fast cycle time and slower hits but lower average cache access time
 - Pentium takes 1 cycle to access cache
 - Pentium Pro-Pentium III takes 2 cycles
 - Pentium 4 takes 4 cycles

Ajit Pal IIT Kharagpur

And finally, another approach that can be used that is known as pipelined cache access. So, L1 cache access can take multiple cycles, so you can pipeline, in the cache lining you can pipeline. So, whenever you do that, then it gives you fast cycle time and slower hit, but lower average cache access time. So, Pentium takes 1 cycle to access cache by using this pipelining, and Pentium 4 through Pentium 3 takes 2 cycles and Pentium for takes 4 cycles, for this pipe line approach cache memory access.

So, with this we have come to the end of today's lecture, where we have discussed various approaches to reduce hit time, cache time. And in my subsequent lecture, we shall discuss about other two, I mean techniques for reducing other two parameters.

Thank you.