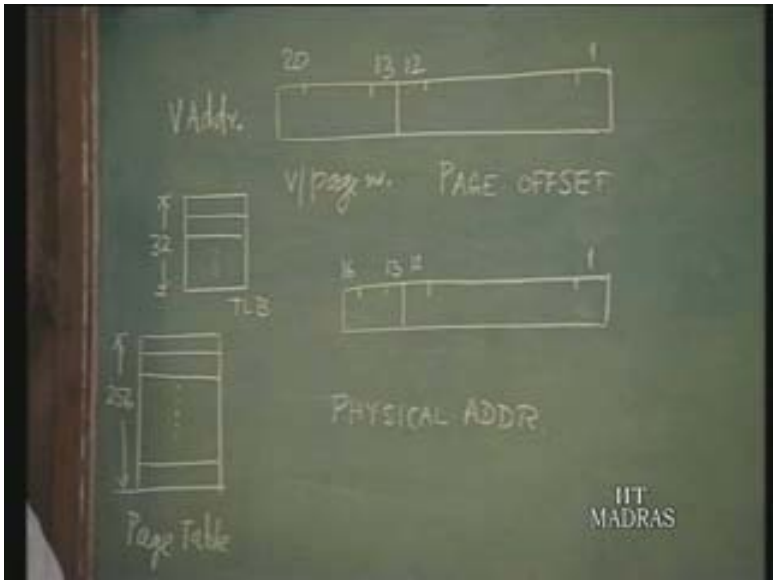
Computer Organization
Part – II
Memory
Prof. S. Raman
Department of Computer Science & Engineering
Indian Institute of Technology, Madras
Lecture – 21
Performance Calculation

In the previous lecture we had seen how the translation from virtual address to physical address was taking place. Essentially, once the page size is in bits, you find that the page offset bits would be the same whether it is a virtual address or the physical address. However the virtual page is going to be indicated by the remaining of the bits. In the example we had chosen, we had a 20-bit virtual address and the 16-bit physical address, so the 8 bits that are there in the virtual address, which actually corresponds to virtual page number, is actually a set of 256 virtual pages. Corresponding to that, in the physical memory you have only 4 bits, which means there are only 16 pages.

So 16 of the virtual pages will be available in the physical memory. Then we had assumed a page size of 4 KB. Now assume that there is a program of size 8 KB; that means half of the program is already available in the physical memory. If that is so, you can see that this virtual address to physical address translation need not always be done because once you know which page it is – because essentially there will be two pages that will be required – then you see that given a particular page, the translation from this 8 bit to 4 bit need be done just once and you can possibly carry on unless you go to another page.

You can carry on with the same particular address, and that is why this virtual address to physical address translation involves referring to that page table. This page table is in the memory, so any access to data consists of the two components as we saw in the previous lecture; that is, first referring to the page, which is in the memory, and then, getting the physical page information and then actually constructing the physical address and then making use of the physical address to arrive at the data. So, to speed up this process, because frequent referring to page table is not necessary, you introduce a cache like structure in-between, which is in fact a buffer, specifically in this case, which is a translation look aside buffer with not as many as 256 entries, that particular buffer with may be 16 or 32. In our example we had assumed a 32-size buffer. Now this buffer is also going to give the same information, which the page table gives. Being a cache, this is not going to contain all the entries, that is, 256 virtual pages that are there in the buffer. You are not going to have all the 256; we are going to have the most recently used. In our case, it is 32 pages because we assumed size of 32.

(Refer Slide Time: 10:26 min)



Where is this 32 coming from really? How do you really do it? There are 8 bits; for 32, you need 5 bits; so 3 bits are remaining. These 3 bits really indicate 8 different blocks of the 32, which constitute the total 256 to understand what is going on. Remember this is something like a cache. Let us take look at the cache arrangement, something which we have seen earlier; take a look at the chart; it is something which you have seen earlier. Now here you see a cache; you can take it as the translation look aside buffer and here you have the main memory. Actually this corresponds to the page table. It is a cache in main memory; just understand that this is TLB, and this is the page table. Now you can see here, in this particular arrangement, any one of these four can be in this particular location. So obviously, there must be an indication of which of these four; given that, we know that it is called a tag bit.

In this particular arrangement you need two tag bits which will indicate which of these four, because it indicates 1 of 2^2. Similarly, in this particular translation we have 3 bits here, which will indicate which of one of those eight groups and then the remaining five correspond to the 32 entries, most recently used entries. Now you might ask after all finally what we want is this particular physical page number, which is already available here and you are going to make it available something like duplicate in duplicate here also that's what it is. What exactly do we gain in this whole process? Remember that this page table is in the memory, whereas TLB is like a cache. Whatever advantage we get out of cache we have to get the same thing here. For instance in TLB the access time maybe 20 nanoseconds whereas the main memory access time may be 100 nanoseconds. Now if we assume a say 4 KB size, that is, 4000 and odd times, instead of accessing this 4000 times, every time taking 100 nanoseconds, if you can access this every time taking only 20 nanoseconds, there is savings.

That is precisely what we gain from this. So that is the purpose played by the buffer in this case? It is something like a cache. We also saw in the previous lecture that the CPU places the address, virtual address, and then from the virtual address the physical address will have to be got. First it will look into the translation look aside buffer for the corresponding page; if the information is there, it will take it – then we call it a TLB hit. If it is TLB miss, then we have to look into the page table to see whether that particular page is available or not. Present page may not be present, we do not know. If it is TLB hit, then page is present.

Now we have to remember this point again and again for read cycle and for write cycle. Whenever something is written in the address location we should see that the data integrity is maintained, that the data is faithfully available to all parts of the computing system. The same data must be available in anything. For instance, we have a copy here. Suppose some change is made here and not there, then this is not truly reflective. So whenever a write is there, we have to see that wherever it is relevant, it must be written. Anyway, we will talk about it later if necessary. So TLB hit means it is available in the page; about TLB miss, we do not know – we have to look into the page and see whether the page is present in the physical memory.

(Refer Slide Time: 14:54 min)

If it is not present, then it is a page fault. If it is present, we proceed. That is, the page number is available, and with that, we construct the physical address and make use of the physical address to look into the next point, cache. Then again, there may be a cache hit or a cache misses. Now in case it is a TLB hit, all that we say is instead of reading from here, the physical page number can be read from here and then concatenated with the page offset; this will be the physical page number, this will be the page offset whether it is virtual or physical address.

It is only that this will change – this 4-bit physical page number and then 8-bit virtual page number is only going to change. Once you have the address, again one has to look in to the cache. So even if in this case of TLB hit all that it says is physical address, it can be immediately computed. Then again it must be looked into the cache. In case there is a page fault, that is, the corresponding page that is required is not available in the physical memory, similar to cache miss we have the fault page fault. Now in the example we have taken, there can be as many as 256 virtual pages, but only 16 pages can be accommodated in the memory.

A page fault occurs when one of the pages, which does not belong to these 16, is the one that is to be accessed because the virtual address indicates a page, which is not present. You remember we have a present bit in the page table, so that if it is 0, that means that particular virtual page is not available in the physical. Now suppose the physical memory has been fully filled in with 16 pages, the problem arises about which one of these 16 pages can be removed and the newly required page brought in. For this particular thing we say which of the page can be swapped out. Which page we can swap out and which page can we swap in? Remember I had used these terms earlier also. So basically by swapping out some page and swapping in another page, we are replacing. So this is also called a page replacement. We are replacing a page, which is possibly not required anymore and then replacing that with a new page, which is swapped in. So there must be some algorithm.

There are essentially two types of algorithms: one is called an LRU algorithm; that is, a page replacement algorithm. Sometimes it is also called a page replacement policy. The page replacement policy is based on some algorithm. What is LRU? LRU is least recently used algorithm; it is not that the algorithm was least recently used; it is just that the page had been least recently used. Why do we have to bring in an algorithm? Why do we have to bother about an algorithm? In the case of cache specifically, let us take a look at the chart. In the case of the cache, of course, we have seen only one specific type, direct mapped cache. What about the cache? Just like page in the other case, in the case of cache we have block.

Every cache arrangement is based on an algorithm. In the direct map of this particular cache, location 1 can have one of these main memory locations: location number 1 or 11 or 21 or 31. Suppose block 1 is available in cache location 1 at a given instant, and then 11 is the next block, there is no other choice here – 1 will have to be replaced by 11. Even though 1 would have been used, just the previous location 1 will have to be moved out and 11 have to come. Now this is fixed whereas in the case of pages, any virtual page can be in any physical page. We will talk about it little more lately again. So since that is not the case with the cache, the direct map arrangement there is fixed, either 1 or 11 or 21 or 31 can only go into cache location 1. That is not the case here. Virtual page number 1 can go into physical page number 1 or 2 or 3 or 4. So that is why you need a page replacement algorithm.

Now we say the specific algorithm LRU means the least, that is, swap out the page which has been least recently used, meaning keep all those most recently used pages intact, whatever had come, and whatever page that had been used least. These are two different things: it is an important word that is used because the nature of an algorithm will tell you why you must pay attention to this particular word used. The other one is called a FIFO algorithm, means it is first in first out. This particular thing does not talk about the usage of the page. This algorithm says whichever page had been brought in or swapped in first, swap it out whenever it is required. It is possible that this particular page would have been very recently used but still this will, because it had been swapped in first, have to be swapped out whereas that is not the case here.

In the other one, suppose we had brought in 16 pages and page 1, let us say, is the oldest page, as per FIFO algorithm, that will be swapped out irrespective of whether page 1 was used or not in the previous occasion, whereas in this particular one, if page 1 had been used just before, even though it may have been swapped in first, would continue to be there, because that is the least recently used. So essentially, you have these two algorithms. Working out some example, we will see how this works and I am not specifically bothered about telling you how to implement this. This is very easy; basically, with each page, we have to associate a number; with each page we have to associate a number and the moment it is swapped in, you keep incrementing the number or resetting to 0 – just work out how it is. So these extra attribute fields associated with the page will give us some idea of keeping track of whether it is the least recently used page or the page, which has been swapped in earliest and so on and so forth.

Just like production bits, some extra bits are required, using which the mechanism of implementing this or this algorithm is possible. Just try to recall why are we doing all these; we are doing all these things so that the user may be given an impression that he can write a very large program, even though physically that resource is not available. So all this translation is going on – the user is not really concerned with it as a page fault occurs not because of the fault of the programmer, it is because we have restricted space; this is the system problem.

If a fault occurs, it means the CPU will have to wait until the required page is swapped in, and there must be extra software, which takes care of the management, which manages all these things. All these things are going to take time. Suppose there is a program, which frequently leads to page fault. Then the pages will be swapped in, swapped out, and the CPU utilization will come down. That will have to be avoided. Anyway, there is a name – when the programs cause frequent page fault, such programs causing frequent page faults are said to be trashing. That is the terminology that is used; there is a term for causing frequent page faults, called trashing.

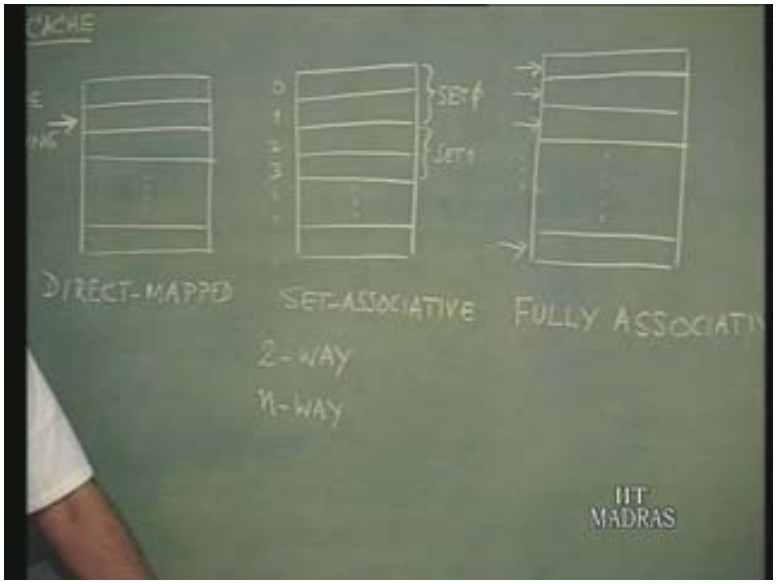
So programs should not be trashing; meaning, once loaded, all the required pages must be available. In other words, misses and faults will add penalty and bring down the processor utilization. I think we have said enough. We had somewhere left our discussion on cache and come down to this because we said that whatever we discuss here is meaningful in the other context also. So that is what we will develop next. We had started

with a cache arrangement; so we will go back to cache and also make use of some of the concepts we have developed here, and then try to study both of them together. The very first cache arrangement we have talked about was a direct mapped, in which we recently saw – you can take a look at the chart again – the cache consists of blocks. Now, given a specific memory address, it maps on to specifically one cache, that is, the direct mapped arrangement – 21 means it will go only into 1; the tag will be 2; 11 means the tag will be 1. It will also map on to cache location 1; 25 means the tag will have 2 and the memory data from 25 will map into cache location 5; 25 means 5. In other words given a specific memory address, it will uniquely map to one specific location, that is, direct mapped arrangement.

Have you seen something else? Not in cache but in virtual memory discussion, we had seen something else; we will come to that. This one we will say is a unique mapping. Given a specific memory location, it gets uniquely mapped to 1. This is at one end we may say; at the other end, we had seen during our discussion on virtual memory, a virtual page was going into any one of the physical pages; meaning, given a specific page address, it will go into this or this or this or any of these. This is the arrangement we had in the mapping between virtual page and physical page. So we are talking about 256 virtual pages and only 16 physical pages indicated by 4 bits here; that is the physical page number and 8 bits for the virtual page number.

Given a virtual page, it will map into any of the 16 physical pages, and that is why we also needed a table, because we need to keep track of which virtual page is exactly in which physical page. We needed that information that is why we needed page table also, whereas we did not need this kind of special arrangement in the cache because it is fixed. So this arrangement, where you have a unique mapping that we said was direct mapped, we can also have a cache in which given an address can go into any block. Given a specific memory address that can be a block address, it will go into any of the cache blocks. Of course, we would need a table like the page table there; anyway this arrangement is the other extreme. This is unique; this one is in fact called a fully associative arrangement. So these are in fact two extreme ends; that is, at one end, we have direct map at the other end, we have fully associative – we can go anywhere. Do we have anything in-between, a via media? Yes we do; because this is fully associative you can say that it is something like a partial association, not called partially associative; I am just saying. That one is called set associative; what is the meaning of that? Here we are talking about blocks number 0, 1, 2 and so on, the number of blocks.

(Refer Slide Time: 28:44 min)



What happens is that some of the blocks will be formed into set; for instance, this one will be called a set 0, say block number 2 and 3 will be formed into set 1 and so on and so forth. That is why it is called a set associative. Now what I have indicated above is only a specific instance or example, because there is no harm if you combine three blocks and form a set. So this is an instant in which I have combined two blocks and then formed a set. This is specifically called a two-way set associative arrangement, cache arrangement. Since two is the specific thing, the more general one will be n.

You can combine any of the n blocks. You can see here that n is 1; it is direct mapped when n is n; when there is this entire thing, then it is fully associative. So n can be 2 or 3 or 4; now we are just taking one particular instant; that is, n is 2. The meaning of this is, given a memory address here, it must be uniquely looked only into that location. Shall we take a look at the chart and then see? Suppose instead of this arrangement, that is, 21, we must map only into 1. Suppose I say 21, maybe in 1 or 0 that is what I mean by set 0 and 1 will form a set. Then 21 would be in either 1 or 0 as far as the cache is concerned. The same thing is true for a 20; 20 also can be in either 0 or 1. So I form a set like this: 0 and 1 as a set, then if 20 comes I will look for either here or there, that is, within the set. Similarly 21 is the memory address; it will be looked for either here or here. Within that particular set the associative set exists.

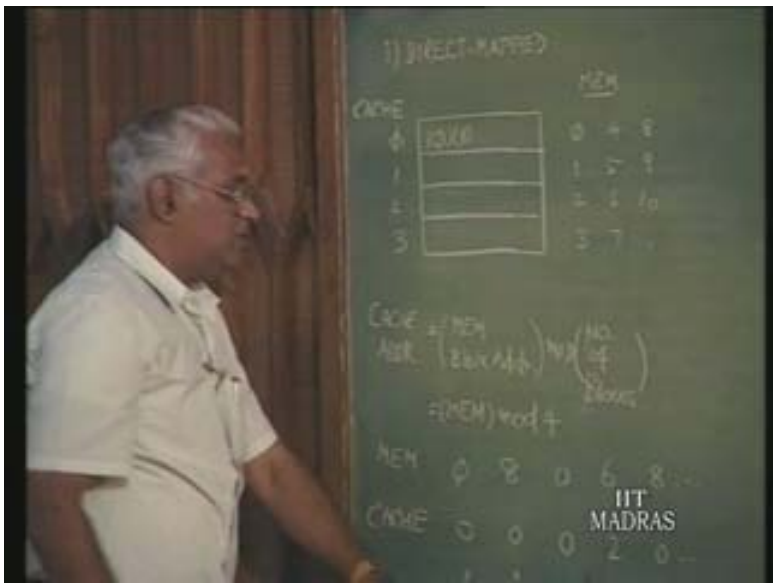
This way the cache content availability can be improved; it all depends upon the sequence in which these things come, the memory addresses come. So let us try to see this through specific example. We will take a specific problem of three caches: one direct mapped arrangement and one specifically let us take a two-way set associative arrangement, and then one with fully associative arrangement. We have to make certain assumptions to make the problem quiet simple. We will make assumption like the block is a 4; here it is very general.

I will just reduce it to 4, and then work out specific problem to see how, depending on the cache size and arrangement; the miss penalty can be reduced. That is our idea; the miss penalty or fault penalty must be reduced. We will just see with the specific example. So for the problem, let us assume a cache of size 4 blocks or four 1-word blocks, let us say. That is, there are just four cache locations; we will assume some specific sequence in which the memory address comes. Now let us recall how the cache address is calculated. You take the memory block address, that is, the memory address that comes and then you do a modulo arithmetic that is mod and then the number will be the number of blocks. How many blocks do we have here? We have four, so the cache address will be calculated as the memory address and then mod, which is four because we have four blocks.

So let us assume a sequence of memory block addresses as generated by the CPU, and then take a look at the cache address. Suppose we have 0, then 8, then 0, then 6, then 8 and so on. This is the sequence in which the addresses are generated by the CPU. Now where will the corresponding cache block address be? Before we compute, we will just take a look at it. In the normal sense, the memory address will be something like this 0; memory address will go here; 1 here; 2 here; and 3 here.

Memory address 4 will map into cache 4; that is a memory block address; 5 will be here; 6 will be here; 7; 8; 9; 10; and so on. We will just see that memory address 0 must be here; memory address 4 here; memory address 8 here; we will compute the same thing. Now memory 0 mod 4; so 0 divided by 4 and remainder is also 0; 8 mod 4 is 8 divided by 4 is 2; the remainder is 0. This is what you find: 8 mapped on to 4. So 0 again is the same; for memory address 6, it is 6 mod 4; 6 divided by 4 is 1 and the remainder 2; check 6 maps on 2, cache address 2.

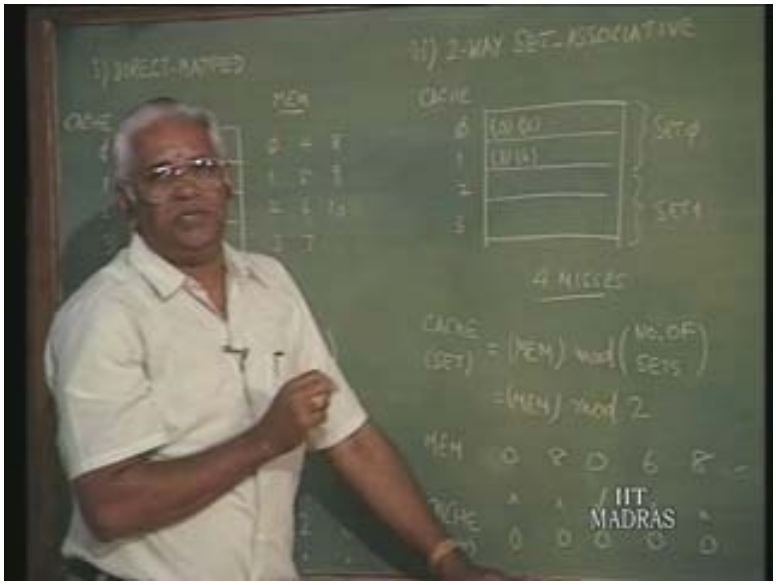
(Refer Slide Time: 37:04 min)



Now see here, this is the sequence in which the addresses are generated. In other words, to start with, the cache will contain all valid bit 0. Valid 1 means it has relevant data. It will all be 0. So the first time when 0 comes, it will look for so this; it will be a case of a miss. Nevertheless, after the first it will be a miss because the cache will not originally contain anything; so it will have to be a miss. Now we are only going to talk about the data part. So the data coming from memory location 0; if you want to put in that way, 0 will go into this cache location. So I will just mark it as from memory address 0, the data part will be in this particular location after the miss. First, of course, there is a miss; so there is a miss here.

Then 8 also have to be mapped here. Now you can see that the valid bit will be 1, but the data content will be data location 0. Again, it is a miss, so the data content from location 8 will be entered into this, into cache location 0. Now the previous data brought from location 0 is gone because it is direct mapped. Again, it is a question of miss. Next, 0 is placed and it is the same. This cache location 0 has contents not from memory location 0, but of 8. So again it is a question of miss. Then, the next address is 6; 6 maps on to cache location 2. It is again a miss because, for the first time, this particular thing is accessed. Afterwards, after the miss, data from location 6 will be entered. Again it is a question of miss. Now the next memory address is 8 – you find 8 maps on to cache location 0, which is here, and it does not contain. So again it is a question of miss and after the miss, contents from 8 will be available. In this way, to the end of this particular sequence, cache location 0 will contain data from memory location 8 and cache location 2 will contain data from memory location 6; that is what it indicates. So it is a question of five misses; this leads to five misses. All of them are misses. Let us just work out the second arrangement, that is, the set associative. We have to be specific – it is a two-way set associative. That is, we are combining two blocks and form a set, two-way set associative arrangement. We will assume again 4 blocks. As far as the cache is concerned, this is block 0; block 1; block 2; block 3 as before; there is no change.

(Refer Slide Time: 45:36 min)



But here these two form set 0 and these two locations form set 1. How do you compute the cache address? More specifically, when we talk about cache address, may be what we mean is the set address. Pay attention to that point – it is a set address. Instead of explaining I will just tell you. In this particular one, this is computed like this: take the memory address and then earlier we were doing mod, the number of blocks, because they were uniquely available numbers of blocks, which were independent. Here we do not have number blocks because we are not going to distinguish between these two. Given a particular thing we say it may be either here or here as long as it is within the set, so we have to put it as MEM mod number of sets – number of sets and what we have here is two. So specifically, it is memory address mod 2 is what we have. For the same memory sequence, let us work out. Not equal to this is the memory sequence: 0 8 0 6 8 and so on.

Now the cache part of it actually means the set. Now apply this memory address mod 2. You can see 0 mod 2 will be 0 8; mod 2 will be 0 because everything is divisible by 2 and the remainder is always 0. So you see that all these are going into cache address, which is, set 0. But set 0 consists of two locations actually. We will just work out how it is. Memory 0, the first thing is, it will be a miss. It will be a case of a miss; I will mark it here. So after the miss, arbitrarily we will say within the set, it takes the first location within the set; it will have to be loaded. There must be some algorithm within the set we have to use. So you just put it there, then we have 8; 8 is also mapped to set address 0, which means here, and now you find this is free. So 8 can go into this. Nevertheless, it is a question of a miss, because, to start with, data from memory location 8 is not available. It is a question of miss anyway, to start with. Next, this is 0; 0 is available in the set 0. So this is a question of a hit. Memory address 6 corresponds to cache address 0. Now an algorithm must come in.

So suppose we have, say, least recently used, then it is 0, which is least recently used. So 6 will come and replace not 0 but 8, because 0 had been used. But if it was first in first out, 6 will be placed; this depends on the algorithm. So we will assume least recently used; since 0 has been used, this will be replaced. Again it is a question of a miss because 6 are not available and it is being replaced. Remember I have assumed the user LRU algorithm in the replacement. So very much like page replacement, we also have the cache replacement and this comes over mainly because of the set problem. Next, 8 are 0. Since this has been replaced, this again is a question of a miss; 8 are not available. Now since this has recently come in, and we have used it, this will be 8 and it will go into this. It will replace this particular one; it is also a question of miss. So now here as per this arrangement, we have four misses.

Now one thing that is to be noticed about set associative arrangement is that there will be a tag for both the locations. So instead of talking about a tag, valid bit, tag bit, and the data field, valid field, tag field and data field for each location, we generally talk about them for the whole set, which means in this particular case it will be something like this: 2 for a set, given a particular set, there will be one tag, one data, and another tag and another data. So that is tag 1 and the data 1; tag 2 and data 2. The moment an address is computed, immediately in that set, both the tags will be checked.

And of course, the valid bits are also there – I am not specifically indicating here. I am saying both mainly because it contains 2 locations; if you have n locations all by n, they can be checked, let us say, in parallel and then choose whichever is free. If there is contention, then we solve it with some algorithm. That will have to be done in the set associative case. Now let us see the third category, namely, the fully associative one, the third category that is fully associative.

(Refer Slide Time: 51:16 min)

In fully associative, there is really no computation of address and so on and so forth; we will see why. In cache, any memory block address can go into any cache block. Now let us just take memory, the same sequence 0 8 0 6 8; we can work out directly. This is a sequence you assumed so far. So let us see about that. In fully associative, anything can go anywhere. First it is a miss, so let us assume 0 goes here. It is a miss. For 8, we will assume you can go anywhere; that is the way we have been following. So 8 goes into the next one; this also a miss because initially it is not there; 0 is a hit. It is available here; 6 is a miss and let us place it here. And 8 is a hit because it is available here. So now you can see that there are three misses.

Depending on the different arrangements we have different number of misses. Also, it is dependent on other things. Suppose we change the block size, say, from 4 to 6. Now let us just see for the same two-way set associative, you change from block 4 to block 6; just see for yourself what happens. There are six of them here. So if you do set associative, then for the same 0 that is memory address, 0 8 0 6 8, we have to have mod 3 because we have three sets here: set 0 1 and 2 3 sets. So the cache address correspondingly will be 0 2 0 0 0. Mod 3 will be 0; mod 3 is 0; mod 3 here will be 2. Now see what is happening. This will go into 0, so it will be here, let us say. Then, 8 goes into 2, set 2. Anyway, it is a question of a miss; to start with it is a miss; 2 again is a miss. As 0 is available, it is a hit, and 6 is a miss but it can be accommodated here.

And then, 8 are already available. So now you have three misses in this because there are two hits. Now compare this with what we had earlier. We had four misses and now we have three misses. So basically the lesson we learnt from this is that this particular performance depends on the cache size and different arrangements. You can work out the same thing; for instance, instead of having this, you go for an 8 block. We have 6 blocks here; you go for cache of 8-block size and work it out. Depending on the cache size, the misses will vary. In other words, the performance will vary. So the performance depends on the cache size also. Another point is, why talk about only one cache? You can have multi-caches, multi-level caches: cache 1; cache 2; cache 3; and so on and so forth. As a similar concept, we introduce the translation look aside buffer also the same way. All these things are in use; anyway the end result is that the miss penalty must be reduced – that is the main thing.