Computer Organization
Part – II
Memory

Prof. S. Raman
Department of Computer Science & Engineering
Indian Institute of Technology, Madras

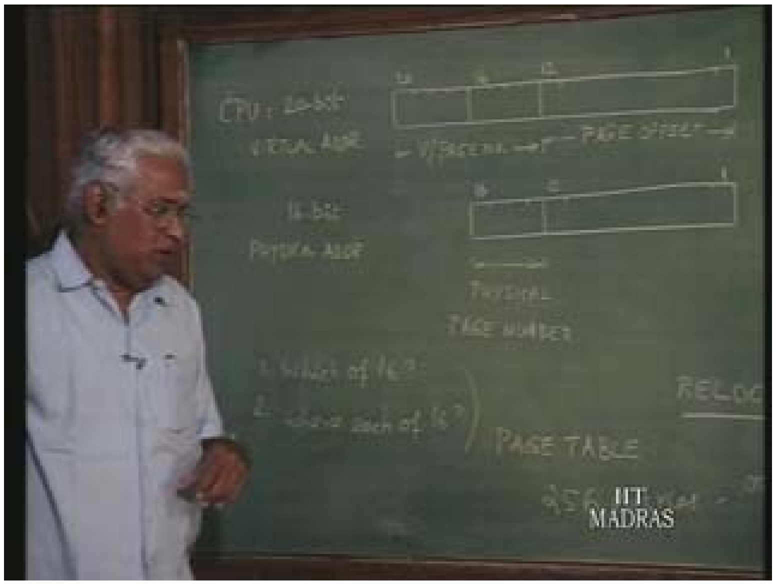
Lecture – 20
Virtual Memory (Contd…)

In the discussion we have considered in the previous lecture, we had assumed that the virtual address is characterized of 20-bit address, which means the CPU is capable of generating a 20-bit address whether there is physically that much memory or not. Now 2^{20} would mean 1 MB; that is, as far as the virtual address space is concerned or virtual storage space, we may even call it a disk address space. Now physically in the memory, we have assumed that 64 KB is available, which means essentially we have to deal with a 16-bit physical address. So there is a translation from 20-bit virtual address to 16-bit physical address – this is one aspect of it. Then we had also assumed a page size of 4 KB. For 4 KB, essentially what you need will be 12 bits – we were talking about virtual pages as well as physical pages. The CPU generates a virtual address because in the CPU the program is loaded and the user has developed that particular program and he has been given the impression that the address, the total address, space available is 2^{20}. He is not really bothered about what physically exists; that is why it is called virtual.

We were calling the unit that passes a block in the cache and here we are calling it a page. In this particular one, the unit processes between the virtual storage in the physical storage and the page, that is, one full page will be getting loaded. So you can see that the in the 20-bit virtual address the least significant 12 bits really will correspond to the page size. In other words, for the page size of 4 KB, essentially you need 12 bits of addressing. You can call it a virtual page or a physical page; it is of the same size and as far as addressing any location is concerned, we must be able to address at the level of byte, whatever may be the page size. So within a page, what we are saying is there are 4 KB. For addressing any location within a page, you need 12 bits and this particular 12 bit will be called a page offset, which would remain the same whether it is virtual or physical; here also it holds good.

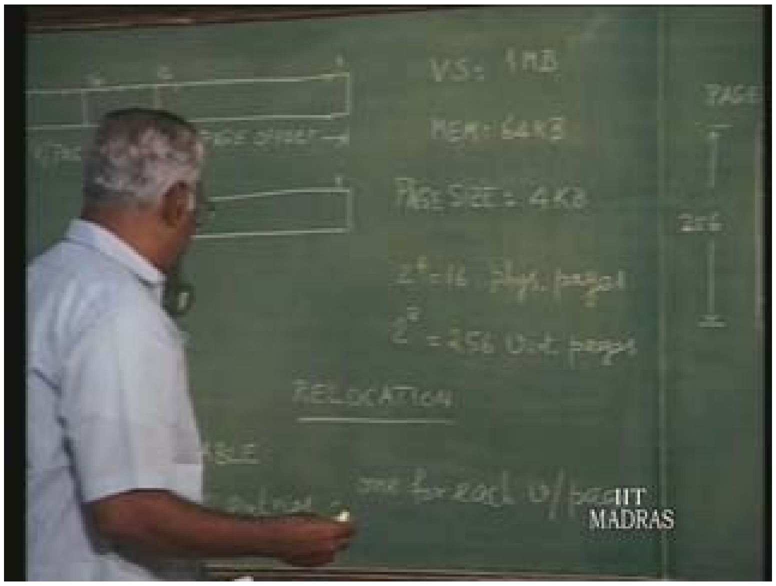
While translating from the 20-bit virtual address to 16-bit physical address in this particular example, you would find that the least (4: 40) significant 12 bit will not change because it is the same page that goes from virtual to physical. These 12 bits essentially tell us which specific byte within the given page is available. That is the page offset even in the physical address. Now what about the remaining 4 bits in the physical address? What about the remaining 4 bits? This particular thing will specifically tell what the physical page number is. We have 4 bits, which means essentially we can have only 16 pages; this is what we have seen earlier. So these particular 4 bits will give you the physical page number, which means essentially what is physically available, is in reality, which the CPU can access. The CPU may generate a different size address but CPU can access only this particular thing because that is what is physically available there. So it is 2^4, which means there are 16 pages. We can call these physical pages because we are going to use the same term page both in physical and virtual; that is about it in the physical page.

(Refer Slide Time: 18:18 min)



Now let us go to the 20-bit virtual address; we have 8 bits there, that is, bit 30 to bit 20 so 8.This particular thing will be actually giving physical page number; this one will be giving the virtual page number – this is the virtual page number. This is what actually we were talking about in the previous class also. That is, we have 2^8 because, in the virtual address, we have 8 bits remaining and 2^8 correspond to 256 virtual pages.

(Refer Slide Time: 18:45 min)



At any given time, only 16 out of these 256 pages will be available; this is what we were saying in the previous class. Now we must have these 2 bits of information, which I was elaborating on in the previous lecture.

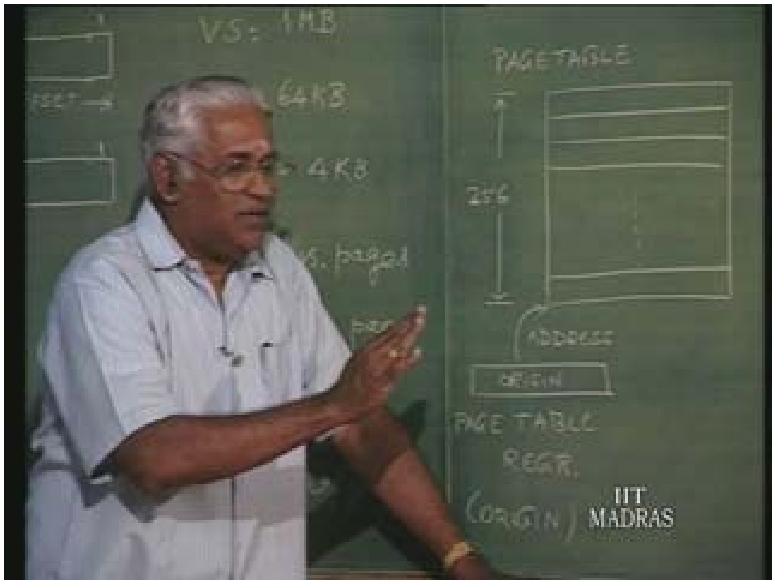
Now which of these 16 out of 256 pages are available and the second thing is that, given a specific virtual page, what is that 16 out of the 256 virtual pages will be in physical page? So given as specific virtual page, where exactly it is? In other words, where each of this 16 is located. The CPU must know this, otherwise how can it proceed? The CPU generates an address and that address will give a virtual page number and physically only 16 of the 256 are available. So the CPU must also know this: which of these 16 and, given a specific virtual page, where exactly it is. Now why do we need this? It is because we said it is meaningful that any virtual page, any of these 256 virtual pages, if you can load into any physical page, it gives some flexibility. Remember in the cache, we are not going for this kind of an arrangement. In the cache, we are talking only about one specific arrangement, which is, direct mapped.

That is, given a cache location into that one of three or four memory locations, physical memory locations, only will be available. This is slightly different; we will take this issue again and then discuss later. At this point I would give a simple example for you to understand. It is like this: supposing you have a book, let us assume I have written a book by title Computer Organization. Now you are going to locate this in the library; you will look for Computer Organization by Raman – this is the title and author – you have this information. This book can be placed anywhere in the library, though it is not strictly anywhere. We may say given specific one or two almirahs, it will be placed anywhere within those two almirahs; so this is precisely what we have here.

The virtual page is something like a book title, and the physical page is thus specific physical location where you can find that specific book. It is meaningful to place a book anywhere but not strictly anywhere. We will come to that because it is not going to be helpful. Of course, those who can locate the book just by the title will be able to tell immediately to go to the particular place and so on. But still, this specific physical location can keep varying from time to time. Now the fact that you can place a virtual page anywhere within the physical page is called the feature of relocation; that is, you are able to locate anywhere that you want.

This gives quite bit of freedom, assuming – I will just give an example – one specific memory chip is down and that may correspond to a set of memory locations, in that case we can always bypass that memory chip, defective chip, and place our program anywhere. That way also it helps; so while locating or relocating a program when it is run, there is no guarantee that the program will always find it physically in some specific portion. You may be able to locate it anywhere; that is essentially called the feature of relocation. Now let us get back. We have this problem: now we can have only 16 pages out of the 256 and we also want this relocation that is, locating a virtual page anywhere. So we also need this particular thing, where each of these 16 is available.

(Refer Slide Time: 20:23 min)



Obviously CPU must have this information. So these bits of information that we can pack into it will be placed in a table called a page table, that is, essentially a table, which will have enough entries – a page table which will have enough entries, which will indicate which virtual page is available. Now where precisely this page table will be if CPU has to access, where else can it be? It cannot be in the storage, and so it has to be only in the physical memory, that is, in the DRAM. Some portion of the main memory will be occupied by this page table and the CPU must always have information on which of the 16 are available.

This means basically that it itself fixes the size of the table; the table must have 256 entries. It must have 256 entries, one for each virtual page. In that particular table, there is one entry for each virtual page. Now by having some entries in that particular table, what are the things we can have? We can say whether that particular virtual page is available in the physical memory or not by having 1 bit and then, if it is available, where exactly it is available. So in a page table in our case with 256 entries, and one entry for each virtual page precisely 16 entries will only be having a particular bit, which indicates that it is those 16 pages that are available in the physical memory, and for those 16 entries, we can also add this information on where exactly it is available. So that is what we have to introduce.

Let us now take a look at translation of 20-bit address, and a physical 16-bit address, so I will just indicate first page table, which will be useful. Let us say that is the page table with, as we said, 256 entries. So you must have 256 entries. Before we proceed further, let us see how exactly the table is organized and what the contents are. Let us see one more thing: I said page table must be accessed by the CPU and hence it will be in the physical memory. Now if it is in the physical memory, where exactly can we find it in the physical memory? Again we are coming back to the fact that there should be recourse specifically to some locations for this page table, in which case it can be fixed. Suppose, instead of 4 KB I go for a different design, let us say 1 KB page size, then when I have 1 KB, I can have many pages.

That is, four times what I have – instead of 256 pages, I will end up with 1 K, which means, four times this or 1024 precisely. Then the page table must have as many as 1024 entries instead of 256, which means there must be a way in which the CPU will know where exactly the page table starts because the page table size can vary depending on our choosing the size of the page.

So there must be a mechanism by which we can tell the CPU. The specific location where the table starts is generally called a table origin register. That is a very general term; essentially it will be called a page table register, which means it tells precisely where exactly the page table starts. So really speaking, it is a table origin register, so it will tell the origin of the table. The contents of that will give some address, which essentially will point the contents of this, which will tell where the table originates. So that will point to the start address; essentially it is an address, start address of the table, so that can be located anywhere; that is the idea. Where will this register be? You take a look at it – generally a register is a CPU. So this must be part of CPU. Now you can just see how exactly certain software considerations like memory hierarchy are influencing the design of the CPU also. The CPU must have this register; this has nothing to do really with the user program execution. This is purely from the point of view of this system, organizing and making use of the different resources like memory, etc. So we can just see that the CPU, which can take care of higher level software features, will have these extra things as part of their architecture. It has really nothing to do with the application program and so, this in fact, is something from the point view of the system program, that is, incidentally. So there will be one register, which is part of CPU, and that will give the start address of the page table. Now you may ask me, how about the size of the page table? There can be one more register, which can indicate the size of the page table also and so on and so forth. The CPU's complexity can be going on increasing.

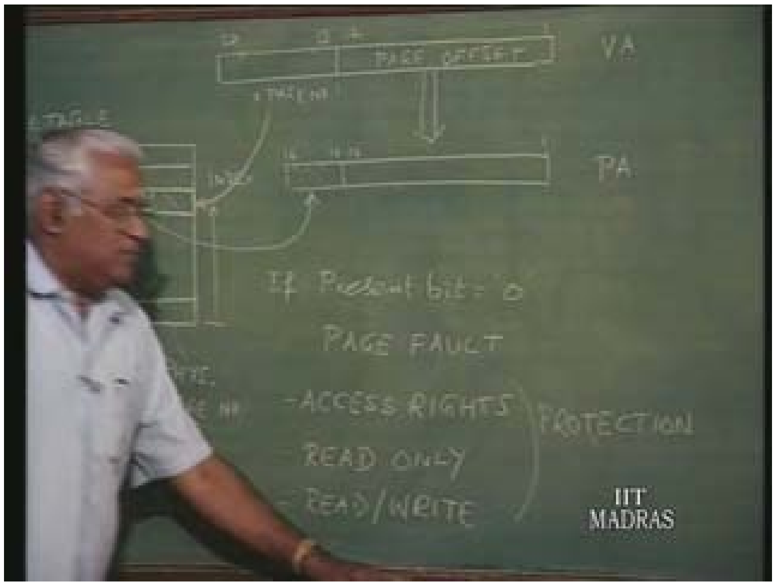
Now let us get to the page table: there must be at least 1 bit in this – there are 256 entries; so at least there is 1 bit in this. This in fact is typical; let us say that is a typical page table entry and we start with page table – this is the point. It has to indicate whether a given virtual page is in physical memory or not, and if so, where exactly it is. One of these bits will be indicating whether this particular bit – I am talking about 1 bit – whether the given virtual page is present in physical memory or not. Hence this is called a present bit; some people may call it valid bit also. Now what about the rest of the bits? That is, let us say, this is the present bit. Now what about the rest of the bits; what should they give us? Essentially it should tell where we have 256 entries, one for each virtual page. So given a particular virtual page, you should tell where exactly that particular virtual page is, where can it be; and how many physical pages are there. There are only 16 physical pages, so it must give essentially an address; that is, in this case, a 4-bit address. That is what we are specifically working out for this example: it must give a 4-bit address, which will identify this physical page number. You have this 8-bit virtual page numbers; so this 8 corresponds to 256, so this 256 is what you have, starting from 1.

So given a specific page number, the contents of that particular one will be actually these 4 bits. In other words, the physical page number will be available. These 4 bits is the physical page number. So this page table itself will be in the memory. Now, given this, let us complete this translation part, that is, essentially the 20-bit virtual address. So we have these 18 bits; this in fact is the page offset, and then we have 16 bits. Here is our virtual address and now this part is the page offset.

Basically this part of the address indicates just within the page, so there will not be any change. Whatever the virtual address, the same 12 bits will be there in the physical page also. This one, which gives the virtual page number, we say indexes specifically into one of these entries in the page table.

This is essentially some index – what index is it? We have 8 bit, 2^8 or 256; in other words, you can see that from this particular address, it gives this offset. So if you go on like this with reference to that, basically it is an offset within so the virtual page number specifically tells you which entry in the page table. There are 256 entries, one for each virtual page. That is, in fact, indexes and this 4-bit information is the physical page number. These will be entered into this; so this is, in fact, the translation procedure. Now when does this hold good? It holds good only if the present bit indicates that this particular virtual page is in physical memory, which, in other words, means this particular bit must be 1. When the present bit is 1 and the virtual page number will specifically index into the table, it indicates which one of the 256 entries is available. So it takes that particular entry and sees whether the present bit is 1 or not. If it is 1, that means it is available in the physical memory. Assuming so, this part of the table entry will give the 4-bit physical page number. Now we say page is available in the memory. If the page is not available, this present bit will be 0. We can note that. If present bit is 0, it means it is not present; in other words, a page fault occurs. If present bit is not 0, then we say there is a page fault; meaning the virtual page is not available in the physical page. Where is it available then? It is available in the disk; it is not in the main memory.

(Refer Slide Time: 32:12 min)



So a page fault occurs. Now compare this: in cache we discussed how the CPU looks for something in the cache. If it is available, then we say it is hit; if it is not available, it is a miss. This is similar to this. Here we call it page fault, but the penalty is different. Do you remember why? Recall the figures: something like 20 nanoseconds and 100 nanoseconds; that is the speed difference between cache and DRAM.

But within DRAM and the other one, if this is 100 nanoseconds the other one is 20 milliseconds, the penalty that we are causing will be much lower than the if it is a fault penalty. Now this present bit is a must as we are seen. So that is 1 bit; what harm is there if I add a few more bits? I can possibly have 1 bit, which indicates that this particular page is only for the system's use or for the user's use. That is, I use one bit for reserving the use of the page. I can have another bit, which says this page can only be read; you cannot write it. So these are called something like access rights, meaning only the system's software has access to a particular page; the user cannot access it.

And then you can restrict the user's access by saying things like this particular page can only be read, it can be used only for reading or you may say that particular page can be read or written, that is, read or write. In this way, you can go on adding an entry in the page table as shown here, where only present bit is included. In addition, we can have other things. By having these things, we incorporate the second feature, which is protection. That is, we are specifying access rights, whether system use only or user use only – you protect the user from accessing the system's address space because multiple users are there. The system software will be only one, which supervises the entire thing. When there are multiple users, we have to see that the users do not access the system's address space indiscriminately and corrupt that. In this way, you can implement the protection – what is it you are seeing here? We are seeing how exactly the address is translated. This is the translation part process, and then, by adding a few more features, we can talk about the protection.

Do you recall we were talking about translation and protection? In other words, what we have seen essentially is the basic principle of memory management. How exactly is the memory managed? Essentially translation and protection are the two aspects of memory management. You would see that in some systems, there may be some special chips for incorporating this memory management because it involves looking into something and reading from it; if not, take some corrective steps in case of fault, miss and so on. You find that in some systems you may be having special chips; they are called memory management units, which essentially is a hardware solution for managing the memory. What is it we have seen so far? We were talking about CPU generating an address, and what is the end purpose of this? The purpose of this is to access some word, may be instruction or may be data. What is it we have ended up with? We have this address and in translation the CPU is looking into another location because its page table is in the memory and from there it proceeds and gets another address and then this address will have to be used. As per this arrangement, the minimum we find is that the CPU, when it generates an address, has to look into the memory at least twice. It has to look into the page table once, this is memory access, and then, after it gets the physical address, again it has to look into the memory.

In these two steps, it actually gets the address; but really it is not just two steps, some more steps are there. But now what happened is this: now we are actually reducing the speed of the CPU memory access. This will have to be improved upon. So there must be some schemes by which it is not just the CPU placing an address and getting the data. This is how we started all our exercises. Now the CPU places an address, gets another address, and then it proceeds. So at a minimum, there are two steps involved; and what happens in case the page is not available? We need to take a closer look at this translation process, which we will do immediately.

So when the CPU accesses the memory, we have seen that there are two steps. The first thing is that from the virtual address, we make use of the virtual page number and index into the page table and then the page table will indicate whether the particular virtual page is available in the physical memory, in which case it will have the physical page number. That will be used, and that in fact is the address. Now assuming the page table, usually what happens is the page table sizes can grow bigger and bigger because it is variable anyway. It all depends on our page size. So now, the other element is the entire translation process. In this translation process, why do we all the time have to keep looking into this page table? Is it necessary? Recall the principle of locality. Normally the page size is 4 KB; that is what we started with and that is what we have been working out here. When the page size is 4 KB, you take one particular instruction. After that instruction is executed, normally the next instruction is executed; where is that next instruction? There is a high probability that it is in the same page. That is the principle of locality.

If that is so, for the next instruction, which is available in the next few bytes, why keep looking into that page table? Instead of this particular indexing into the page directly from which we take the physical page number, why not introduce another buffer arrangement? This is essentially a buffer arrangement; it has a special name and we will come to that. This is some buffer arrangement, which can hold the most recently used few locations, hold the most recently used pages, information about those pages. Suppose a particular page has been accessed and you have all these and that particular one. So what else do we have, we have 256 pages. Arbitrarily we can say most recently used 32 pages or 16 or 64 can be seen. Then what we say is that out of these 256 entries, the most recently used 32 pages can be made available here; and this can be a fast memory, faster than the other one in the RAM itself.

So we can have a buffer, which holds information about the most recently used pages, let us say, 16 or 32, in which case this virtual page number can index into this. That is, basically it is going to indicate this. So from here, we have to see; but now there is a small problem. Remember, the page table has 256 entries, whereas here if it is only for 32 pages, we have only 32 entries. But this may be the faster one; and what this one can have is not like this particular thing. Actually it is coming in-between this and the other one, and also 32 out of the 256 alone are available. This is something like a cache, with which we are familiar. Now how does this help actually? This can help such that the moment the virtual page number comes, that indexes into this particular buffer and then, if this particular virtual page is available in the physical memory, it must be because we have been talking about the recently used ones. But then there is also the possibility that it may not be. So we will look into the organization of these entries a little later, in which case, if it is available, it will generate directly a physical address.

That is, essentially what we will be having here is the required physical address – it will be directly available without any more translation. Otherwise what happens here? It looks for a particular entry in the page table and then it takes this particular physical page number and the other page offset and then generates the address. Instead assuming it is a recently used page and it is still the page which is in use, that information also is available there. If it is so, then this one can give the physical address directly. Some part of it can give the physical address completely. What does this require? This particular thing requires an arrangement so that it takes any entry – there are 32 entries.

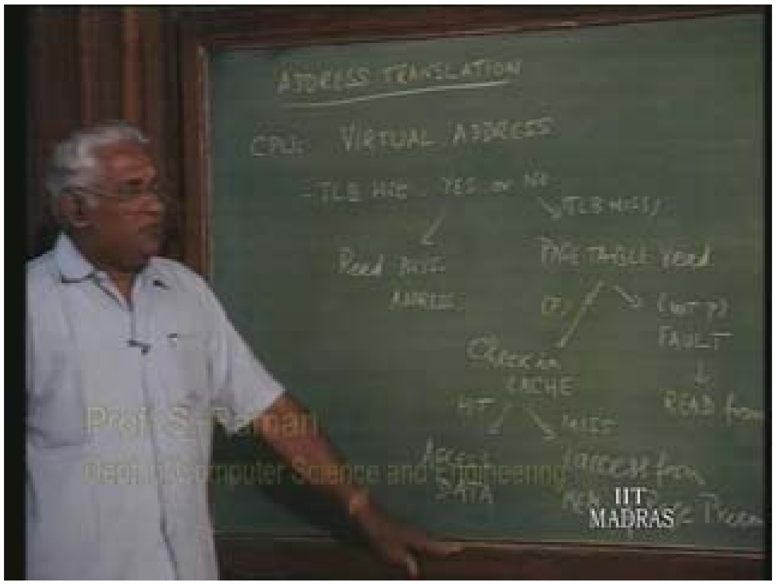
You take any entry; I have already indicated what it must be – it is like a cache. This will have a valid bit similar to what we had in the cache; in addition to this valid bit, it will have to have another field similar to what we had in the cache, a tag field, because this tag is the one, which will tell us about the availability because there are only 32 out of the 256 available here. So the tag is down, which will indicate which one is 256 by 32, which would be 8. So which of these 8 is available in this depends on that specific arrangement; so it is very much like a cache.

As the CPU was making use of a cache, for translation also we can have a cache and I was just calling it a buffer because it is specifically being used for translation. This particular thing is called a translation look aside buffer or TLB for short. TLB actually stands for translation look aside buffer; that precisely is a function of that TLB. So the virtual page number will index into the 32 entries of translation look aside buffer, and the entries here, like in cache, will indicate whether the particular thing is currently valid or not and then it will also indicate which one because it is only a subset, 32 of 256, which of these is currently available here; if there is, there is no more translation. This directly it will give the physical address that is required. This is what we may call an architectural arrangement to speed up the entire process.

Now this way, there is no translation from this; we do it by looking into it and then concatenating forming the physical address and then placing and so on. We have seen enough of the address translation algorithm. Now why don't we take a quick look at what is happening in this address translation starting from the very beginning? What is it that we started with? The CPU generates an address, that is, the virtual address. The CPU generates the virtual address; now let us look at the home scenario. Let us not go in to the details of how this.

We now have to find out from the virtual address, which one gives the virtual page number and the page offset. That is made use of to see whether there is a hit in TLB because the translation look aside buffer is very much like a cache. Here also we can talk about a TLB hit and TLB miss – it is not different from cache. First of all, if it is a TLB hit, that answer will be yes or no. So depending on the answer, if it is a hit then what does it read? We take the physical page address, physical address or the page number, and concatenate and proceed. Let us say it is read physical address; that is straightforward, that is, the address is readily available. That is, it is already available because a hit basically means it is a recently used page and so this translation has already been done; it is readily available. If it is a no, then we talk about a TLB miss.

(Refer Slide Time: 51:52 min)



Then there is no other way; one has to go into the page table and page table must be verified. That is, we have the page table, which must be read because it is not available here. So it must read from the page table; page table reading must go on. There are two things: the page table indicates whether that particular thing is present or not. It depends on whether the page is present or not – let us say the page is present, the other page is not present. Why don't we take the easiest way out? In case the page is not present, it is the same as page fault occurring. The read from storage, what is read from storage is not available in the main memory. So we have to discuss this issue later. Now if the page is present, the next page is present; that is, this is available in the physical thing.

So, well, you know what it is. What is it? Page is present; so where do you go next? You read and check in the cache because we should not forget to check in the cache; that is, the page is present now. Is it available in the cache – that is the next point? That is, we say that the page is present in the physical memory; now you go from physical memory if it is in cache. We talk about cache hit or a cache miss – if it is a cache hit, then we say access the data; the data can now be accessed. If it is a cache miss, then we access from main memory if the page is present. Here again, access from main memory. So we just put it as memory if page is present. If it is not present, we know it is a fault. These may have been indicated earlier also. So these are the various things that are involved from the virtual address, which the CPU places till the time the memory responds with data.