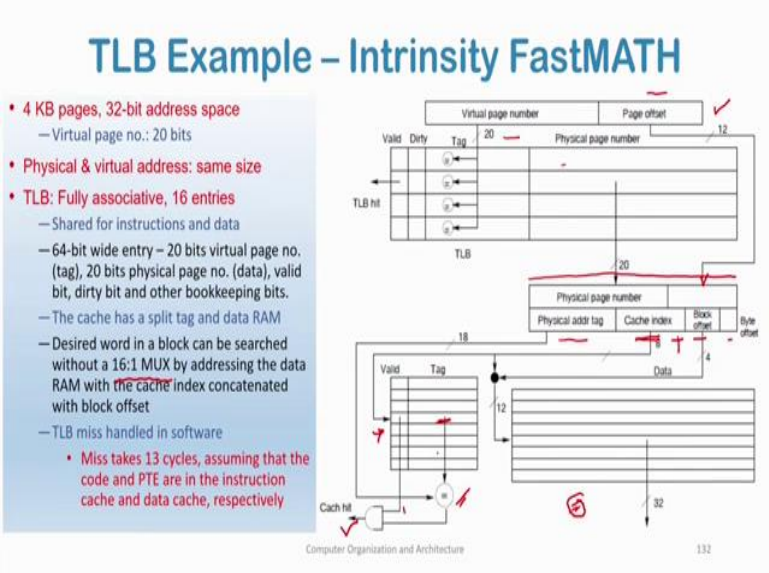
Computer Organization and Architecture: A Pedagogical Aspect

Prof. Jatindra Kr. Deka
Dr. Santosh Biswas
Dr. Arnab Sarkar
Department of Computer Science & Engineering
Indian Institute of Technology, Guwahati

Lecture – 30
Cache Indexing and Tagging Variations, Demand Paging

Welcome. In this lecture, we continue our discussion with Virtual Memories. Towards the end of the last lecture, we took an example of a practical architecture of Intrinsity FastMATH. We will begin today's lecture by briefly recapitulating that example.

(Refer Slide Time: 00:49)



So, we said that the Intrinsity FastMATH architecture consists of a 32 bit address space in which you have a 20 bit virtual page number and 12 bit of page offset. And this 20 bit of virtual page number is goes to the TLB and is matched in parallel to a fully in a fully associative TLB. And if there is a tag match corresponding to the virtual page number, you generate a physical page number; the physical page number is also 20 bits. That means, the virtual the virtual address space and the physical address space has the same size and the page offset goes unchanged into the physical address.

So, we generate the physical page complete physical address here. And, after generating the physical address we go for the cache access and we do so, by dividing the physical page address

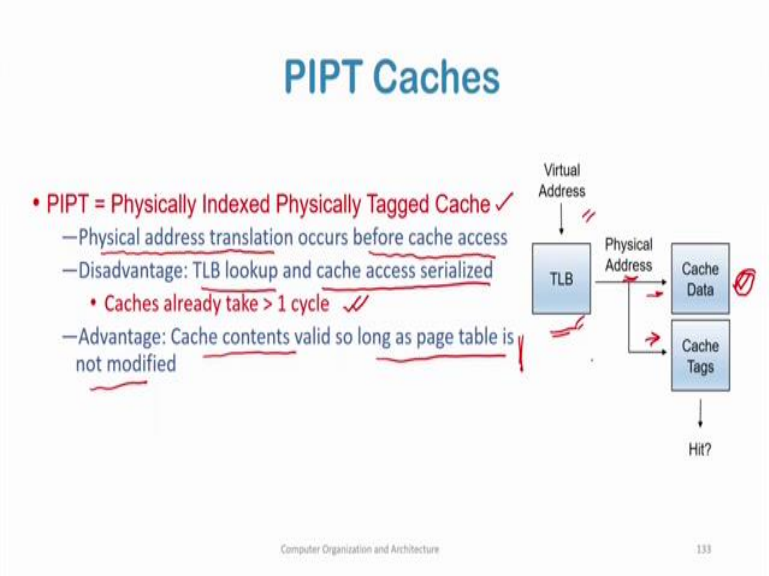
physical address in to different parts one is the physical address tag the cache index the block offset and byte offset. And we said in the last class that we looked at the cache, we looked at a split cache in which the tag part and the data part of the cache was divided physically, logically, it is one cache, but physically the tag part and the data part is divided and we said why we did that.

So, the physical address tag. So, I use the cache index and to go into the tag part and index the tag and when there is a match corresponding to this cache index, I take the tag value, I match the tag value with the physical address tag and then I also match with the valid bit and if the valid bit is on here, I get a cache hit. So, if the valid bit is on and if the tag matches with the physical address tag here and I combine I get a cache hit here.

And I said that I have I had, we have divided this tag part and the data part because we clubbed the index and block offset to generate a 12 bit indexing of this data part of the cache. So, instead of a 8 bit indexing of the cache. So, we have a split tag part and a data part and we said that we had divided the data part to directly access the word within a cache instead of a block.

So, when we address this data part where the cache index is appended with the block offset and we use this 12 bit in indexing of the data part, I directly go into a word otherwise if we did not do so, what we would have what would happen is that we would we would index the data part and we go into we would go into a block and then we would require a 16 × 1 MUX to go into the particular required word within that block. So, by keeping this tag and data split into 2 parts; we can do away with the 16 × 1 MUX.

(Refer Slide Time: 04:21)



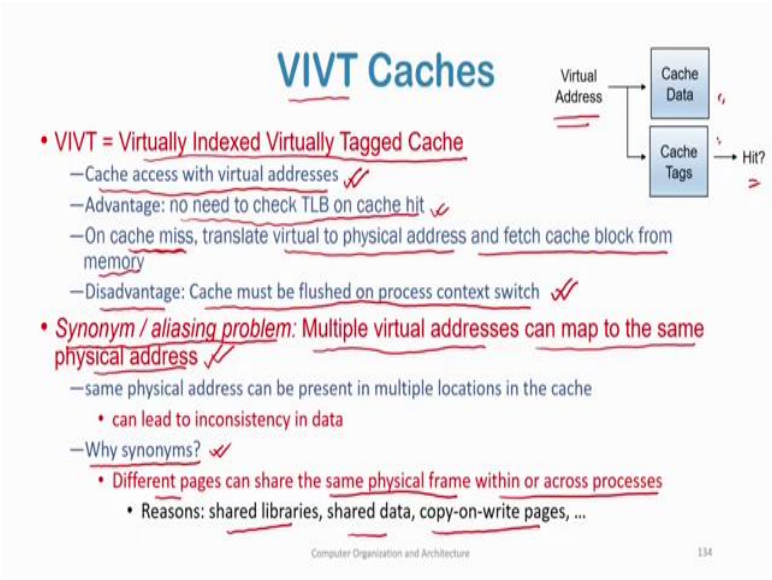
Now, the point for starting with this example again is to reiterate that this was a physically indexed physically tagged cache that we were looking into. So, what is a physically indexed physically tagged cache? So, in a physically, indexed physically tagged cache the physical address translation occurs before cache access. So, first I take the virtual address go into the TLB generate the physical address and based on that physical address, I access the cache this is what happened with the Intrinsity FastMATH architecture that we just looked.

The only problem with this architecture is that the TLB comes into the critical path of data access. So, suppose even if I have the data in cache, I have to go through the TLB and then obtain the physical address and then be able to access the cache, if the page is not present in the TLB, if there is a TLB miss, then we have to go to the main memory to fetch the page table entry required page table entry and get the physical address.

So, even if the data is there in cache we may need to go into memory, because the page table entry corresponding to this corresponding to this data is not present in the TLB. So, there is, so the caches. So, this is the disadvantage of TLB lookup TLB lookup and cache access gets serialized and the cache takes greater than one cycle time and it may take multiple cycles, because if there is a TLB miss I need to go into the main memory assuming that the page table is stored in main memory in this architecture. So, I have to go to the main memory, bring back the page table entry fill the TLB bring it to the TLB and then access again get the physical address and then go into the cache.

So, this happens even if the data is present in the cache; however, the advantage of this scheme is that cache contents remain valid. So, long as the page table is not modified, we will be able to appreciate this advantage a bit later when we go into seeing how this problem of this TLB being within the critical path of data access is solved.

(Refer Slide Time: 06:56)



So, we try to solve; that means, we try to do away with the we try to take the TLB out of critical path by using virtually addressed caches and the first type we will look into is the virtually indexed virtually tagged cache.

So, we look into the virtually indexed virtually tagged cache. So, instead of so what it directly from the name we understand, what happens is that instead of using the using a physical tag address and a physical indexing of the cache, I use the virtual address to both index and tag the cache.

So, therefore, because I directly use virtual addresses, I break the virtual address again into tag part and index part and go into the data and tag part of the cache ok. So, the cache access is done you with the virtual addresses in virtually indexed virtually tagged caches the advantage is that we don't need to check TLB on cache hit because I have a process that process generates virtual addresses directly based on the virtual addresses, I will go that corresponding to these virtual addresses; address do I have the data corresponding to this virtual address.

So, the page the physical address could be anything, but that has been brought into the cache because that has been brought into the cache. Now, it is stored corresponding to the virtual address that data. So, I directly understand that corresponding to this process corresponding to the virtual address generated by this process is does the cache content my required data or not, I don't need to go to the TLB I don't need to I don't need to go to the TLB to address to get the physical address.

However on a cache miss I need to do that, on a cache miss I need to translate the virtual address to physical address by going through the TLB if there is a TLB means going to the memory bringing back the page table entry and then fetching the cache block from memory because on a cache miss, what I have to do there is there is no there is if when there is a cache miss the data I required is not in cache. So, I have to go to physical memory and bring back the data to cache. So, how do I do that for that I need to know the physical address of this data how do I do that? I the only way is to go to the TLB get to the page table entry get the physical address and then go to memory bring back the data and put into the appropriate entry in cache such that the virtual address next time can be able to access the data that I need.

The disadvantage of this scheme there are there are a few disadvantages the first big disadvantage is that the cache must be flushed on process context switch. So, remember that each process has the same virtual address space. So, it is very common that the same set of virtual addresses will be generated for each process and the virtual addresses of different processes may be meaning different things, it is local to the process virtual addresses are local to the process.

So, therefore, when there is a context switch, I cannot keep the cache contents anymore, I have to flush the cache and I have to flush everything that was there in the cache; so right. And therefore, when the new process comes in, again, I will have a set of compulsory misses, I need to I will incur a set of compulsory cache misses always, right. So, this is the first disadvantage that the cache needs to be flushed at every context which when I use virtually indexed virtually tagged caches.

The second problem is that of synonym or aliasing, it is called the synonym problem or the aliasing problem. The problem is that multiple virtual addresses can now map to the same physical address. So, the same physical address can be present in multiple locations in the cache. So, the same physical address because what has happened is that suppose there are 2

virtual addresses of the same process of the same process, I have 2 virtual addresses which point to the same physical memory location.

Why can how can that happen that 2 virtual addresses can point to the 2 virtual addresses can point to the same physical memory location because can share same physical page frame within or across processes, reasons we have we saw we have shared libraries, share data, copy on write pages. So, in case of a for example, let us say I have a shared printf function which is called from different portions of the process.

So, instead of keeping different versions of that same printf function copy the same printf function, whenever I need, I can keep in physical memory one version of the printf function, the code of the printf function and then I can call I can have stubs within the virtual memory which will call the same printf function from physical memory, if it did not have this; what would what would I have to do? I will have to keep several versions of the printf function. So, whenever it is called whenever I when whenever my process prints something, I need to keep the code of the printf.

Now, now it is better to share this code of printf and keep it in one place. So, what does this sharing mean? This sharing means that different virtual addresses will now map to the same printf function which is the same set of physical locations in the physical memory. Therefore, multiple virtual addresses will point to the same physical address in this case and because I have a virtually addressed cache this same data the same. So, therefore, I will what I will have is that the same data or the same code will be there will be accessed from multiple virtual addresses.

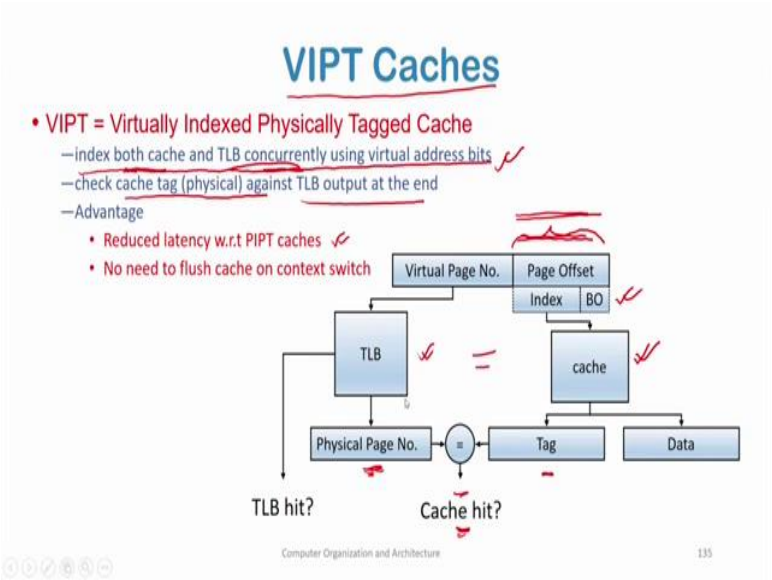
So, therefore, the same physical address can be present in multiple locations in the cache. So, data corresponding to the rather, we can we can say it more precisely that data corresponding to the same physical address can be present in multiple locations in the cache now this may lead to potential inconsistency because it actually means the same physical location in the physical memory suppose one virtual memory writes in writes in to their data and the other one reads it.

Now, I have 2 copies of the data in cache; both meaning the same physical memory location, yes, both meaning the same physical memory location, I have 2 data elements and these 2 data elements are mapped by different virtual addresses. So, therefore, in a virtually addressed cache they will be stored twice in different locations in the cache, and because I have the same

physical data element in two points, in 2 places within the cache, this may lead to potential data inconsistencies. So, this may lead to potential data inconsistency.

So, these are the 2 major problems with virtual indexed virtually tagged caches. So, it tries to solve the problem of bringing or of taking out TLB from the critical path of a data access this is why this was the motivation of bringing in virtually addressed caches, but it introduced 2 new problems. The first one was that of was that the cache needs to be flushed at every context switch. And that may lead to potential latencies due to cache misses of a later process which could possibly have been potentially some of these cache misses can be avoided if I did not have to flush the cache and the second of the second problem was that of synonym or aliasing which meant that the same physical location can the data corresponding to the same physical location can be present in multiple locations in the cache being pointed to by different virtual addresses and this may lead to inconsistency of data.

(Refer Slide Time: 16:01)



Now, to handle these problems; so, to handle these problems while keeping the advantage, people looked into virtually indexed physically tagged caches. So, in this what happens? Both the in the index both cache and TLB concurrently using virtual address bits. So, what happened in previously was that previously what happened was that I used the virtual address and using the virtual I broke the virtual address and then for the tag and data, I used the virtual address for both the tag and the indexing of the cache.

Now, here what happens? I will I will do the indexing of the cache principally using the offset part of this virtual address ok. So, this part of the virtual address will be used for indexing the cache. So, it is virtually indexed because I use the virtual address to index the cache this is virtual, this is virtual indexing.

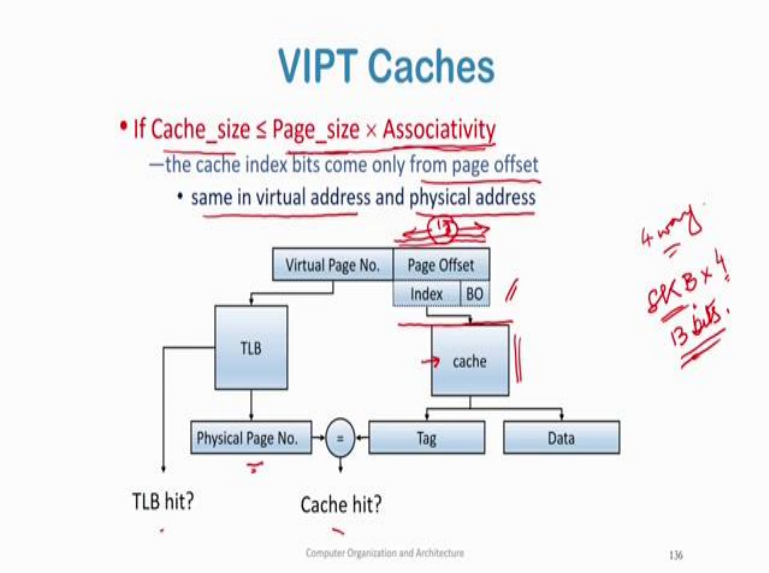
Now, in parallel when I am indexing the cache at the same time I take the virtual page number go into the TLB and get the page number physical page number, if there is a page hit, I go to get the physical page number, otherwise, I need to go to the main memory, but if there is a page hit say, then I get immediately get the page number and therefore, I can match with the tag that is the tag that is available here and understand if there is a cache hit.

So, essentially there is no latency involved I am still. So, I am still having the advantage that I had with virtually indexed and virtually tagged caches why because this TLB access and the cache access this cache indexing and the TLB indexing is happening in parallel concurrently in hardware. And therefore, if there is a TLB miss, I just index the cache get the data and to check whether this cache this was a cache hit subsequent. So, after this indexing is done I just check the page number obtained from TLB does it match with the tag part of the cache if there if it is. So, I get a cache hit.

So, so, both; so, I index both the cache and the TLB concurrently using virtual address bits and then check the tag physical cache tag against TLB output at the end here, the advantage is that I have reduced latency with respect to physically indexed physically tagged caches, because I don't generate the whole page number. And then a subsequent to generating the entire page address, I access the cache, I don't do that.

I am doing this in parallel and we don't need to flush the cache on a context switch why is it. So, I don't need to flush the cache on a context switch because the page offset the page offset corresponding to the page offset corresponding to the virtual address remains unchanged in the physical address and therefore, there the problem of synonym if this is the case here, what the case has the case for the case here; I have no problem of synonym either.

(Refer Slide Time: 19:41)



So, now what has happened the first scenario what we actually looked at in the last slide where I had completely been able to avoid the problem of synonym was this case was this case. Now elaborate this case one this case. So, this case can happen only if the entire cache can be indexed by only using the page offset bits of the virtual address. So, if the cache can be fully indexed by only using the page offset part of the virtual address, I have no problem of synonym as we will discuss in detail.

Now, when does this happen; when the cache size is less than the page size into associativity. So, when the cache size is at most page size into associativity, then I can use only these bits why because page size tells me; how many bits I use for the offset, let us say, I have I have a page size of 8 kb, then I use 13 bits for indexing the page I have 13 bits for this one, this part has 13 bits, 13 bits. And then let us say, it is a 4 way set associative cache, if it is a 4 way set associative cache, then the size of the cache is. So, I have 4 × 8 KB is the size of the cache ok.

So now, what happens these this associate this 4 way set associative means that I will go to the particular I only need to identify the set in the cache, using these 13 bits I will be able to find the set in the cache and all the 4 sets all the 4 blocks within this set will be searched in parallel to find a cache hit. So, therefore, if the cache size is less than page size in through into associativity, I can only use this page offset part of the virtual address to index the cache. And I do not have any problem of synonym in this case, why because this page offset part the cache

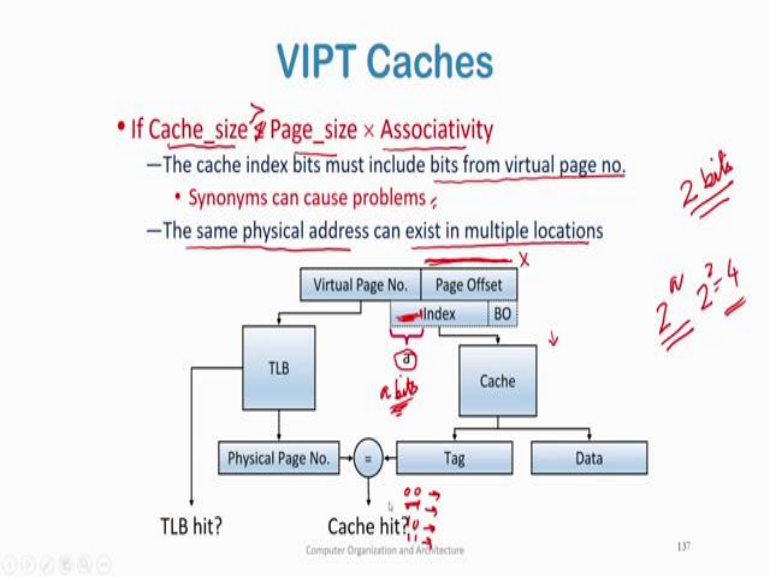
index will come only from the page offset and this page offset is same both in the virtual address and the physical address.

Now, why is this not a problem because now the page offset will tell me which set in the cache I need to go find and then the virtual page number I will go to the TLB, I will get the physical page number and then based on the page offset I have indexed the cache. So, the page offset plus physical page number then tells me my data.

So, it is as if it is very similar to the physically indexed physically tagged cache in its operation because in the I have I cannot have a case in which the same physical memory location can exist in two location multiple locations in the cache why, because this page offset part of the physical memory tells me where in the cache I will have. And from that cache I will just take check for the tag and this check for the tag will be based on page number and I will get a cache hit.

Now, this problem is solved only therefore, for small caches. So, if the I can so, because I only have these 13 bits essentially to address the cache to index the cache in this case the number of page offset bits, I can only use to address the cache either I have to increase page sizes or I have to keep the cache very small now both has limits.

(Refer Slide Time: 23:50)

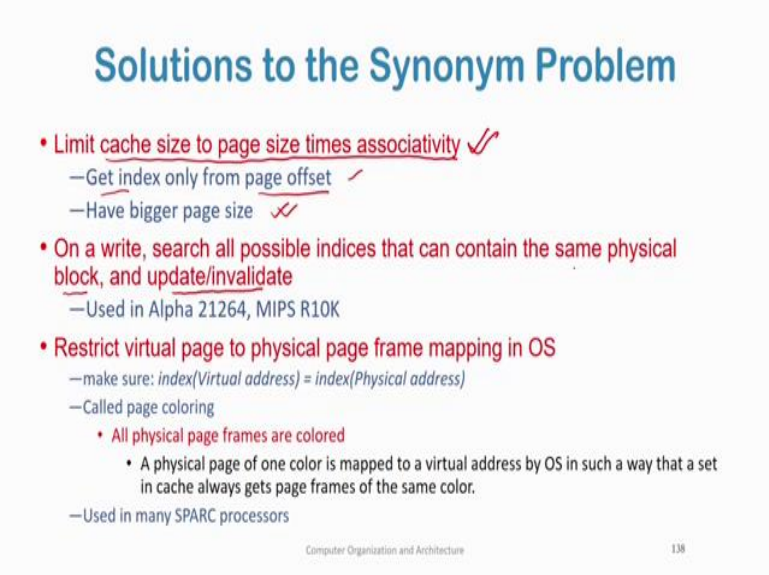


So, therefore, sometimes what happens in the so, in practice, what happens is that I have to have a cache who size whose size is such that it cannot be indexed solely by the bits in the page

offset part, I cannot use only this to access the entire cache. So, I have to borrow a few of the page virtual page number bits as well and here in again comes a bit of a problem.

So, when the cache size is greater than page size into associativity when the cache size is greater than page size into associativity, then only these offset page offset bits cannot is not sufficient to index the cache the cache index bits must include bits from the virtual page number. So, this part, now here I can have, I will again have the problem of synonyms which I had for virtually indexed virtually tagged caches. So, therefore, the same physical address can now exist. Again this problem has come back the same physical address can exist in multiple locations in the cache.

(Refer Slide Time: 25:10)



Now, how do people try to solve this problem of synonyms? Now the first way as we told is that you limit cache size to page size times associativity you do this we already discussed this. So, how do you do that? So, then you get the index only from the page offset part have a bigger page size or small caches, what is the second solution approach on a right search all possible indices that can contain the same physical block and update / invalidate ok. So, what am I doing in this case let us say I have a bits. So, what are the different sets in cache in into which the same physical address can belong the same physical address can belong in at most 2^a different sets ok.

For example, let us say $a = 2$ bits so; that means, so; that means, that to index the cache I have used parts of the page offset for this part for this part the physical address and the virtual address

is same, I have no problem for these 2 bits, this a part is 2 bits, this these 2 bits, I can have 4 different sets in which the same physical address can reside I can have 2^2 equals to 4 different places in which this a particular physical page can then reside.

So, now, for all these 4 locations for all these 4 sets I have to check. So, on a write on a write what happens we have to search all possible indices that can contain the same physical block on I want to write some data into the cache and then what happens during this, I have to search all possible places to retain consistency so that the same physical address is not present in multiple locations in the cache, I have to check in all possible positions in the cache where this data can reside. So, I have to check all the 4 different sets in which this particular in which this particular data can reside and this technique is used in architectures such as alpha and MIPS; this architecture R10 K.

The third strategy is by restricting the virtual page to physical page frame mapping in the operating system ok. So, here I am restricting the placement of physical page frames into the cache, what do I want to ensure that I will derive the same set in the cache by indexing the virtual address as I would do by indexing the physical address this is what I want to ensure and this is done through a mechanism called page coloring.

So, in this scheme, I statically colored the physical I statically color all physical page frames ok into using different colors. So, what will be the number of colors the number of colors will be at least equals to this 2^a so; that means, if I have 2 bits here I will statically color the physical page frames in physical memory with 4 different colors. So, of a set of page frames will be in red, the say another set of page frames will be black, another set of page frames will be blue, another set of page frames will be yellow. So, I will use 4 different types of colors corresponding to each page frame. So, if the page frames in physical memory will be separated into 4 sets and each set will get a different color ok.

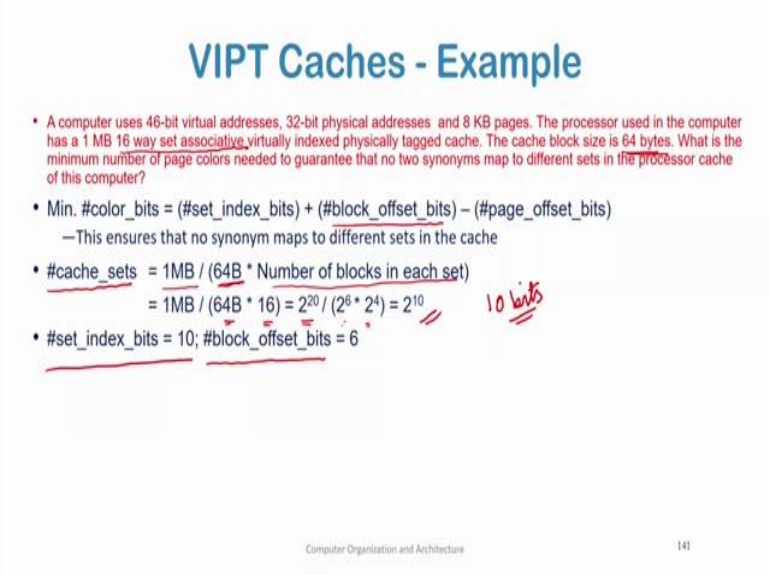
Then a physical page of one color so, after coloring is done a physical page of one color is mapped to a virtual address by the OS in such a way that a set in cache always gets page frames of the same color. So, a physical page of one color so, I have already statically colored; I have already statically colored the page frames ok. Now a physical page of one color is mapped by the virtual address map to a virtual address. So, what will happen? The OS will allocate physical pages to virtual addresses. So, when I need physical addresses.

So, now, it will restrict the OS will restrict which virtual addresses can get which physical pages which virtual page numbers can get which physical page numbers there will be a restriction on that. What will be the restriction? it will map such physical it will map physical page frames to virtual pages in such a way that a set in cache always gets page frames of the same color that a set in cache always gets page frames of the same color.

So, my vert suppose, these 2 bits of these a bits I can have what I these 2 bits can be can be 00, 01, 10 and 11. So, I have 4 colors one color is 00 other is 01, 10 and 11. So, what will happen it will, if this virtual address is 00 I will only I will only allow I will only allow. So, if a virtual address or virtual page number or virtual address has this particular bits 00, I will only allow pages with color 00, let us say 00 is red, I will only allow pages of colors 00 to be mapped to this virtual address, by this I will be able to ensure that corresponding to this virtual address the physical address the physical addresses will always be mapped to the same set ok, this is what is the concept of page coloring.

So, a physical page of one color is mapped to a virtual address the physical page of one color is mapped to a virtual address by the OS in such a way that a set in cache always gets page frames of the same color though. So, a set in cache will always get page frames of the same color. So, this is how I avoid the problem of synonyms in virtually indexed physically tagged caches and it is used in many SPARC processors as well.

(Refer Slide Time: 32:29)



So, before proceeding further we will take an example with virtually indexed physically tagged caches. So, I have a computer which uses 46 bit virtual addresses, it uses 32 bit physical addresses and an 8 KB and 8 KB pages. So, page size is 8 KB, physical address space is 32 bit, virtual address space is 46 bit and the processor used in the computer has a 1 MB 16 way set associative virtually indexed physically tagged cache. So, I have a VIPT cache, 1 MB is the size of the cache and it is 16 way set associative and the cache block size is 64 bytes ok.

So, the question is; what is the minimum number of page colors that will be needed to guarantee that no 2 synonyms map to different sets in the processor cache of this computer. So, how do I determine this? Firstly, what is the minimum number of page colors required the minimum number of page colored bits is given by the (#set index bits) + (#block offset bits) – (#page offset bits). So, set index bits is what I the number of sets in the cache. So, the number of sets the number of bits required to index a set in the cache is will be the number of hash set index bits or the number of set index bits that will be required for a cache is the number of bits that will be required to identify an individual set in the cache.

Block offset bits are what? the number of blocks in each set and then these together when added has to be subtracted from the page offset bits because page offset bits tell me the this part which is not varying the page offset bits is constant. This is same for the virtual address and the physical address. So, that does not vary, the remaining bits vary and those bits must be used to obtain the number of colors.

So, this ensures minimum number of colors these colors ensure that no synonym maps to different sets in cache. So, no synonyms map to different sets in cache ok. So, what was a synonym the synonym the synonym meant that I have 2 virtual addresses pointing to the same physical address and because the virtual addresses are different and this VIPT spurs partially virtually addressed cache. So, therefore, potentially this can happen that these 2 virtual addresses will map to different locations in the cache. And therefore, the same physical page the same physical data same data in physical memory can reside in 2 different locations in the cache, because of this mapping because of these bits because of these bits present in the address.

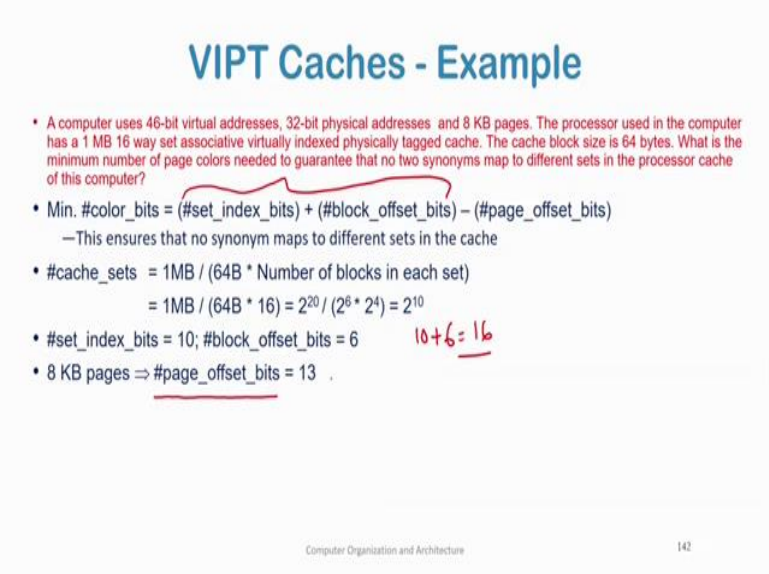
Now, by coloring I ensure that a physical page of one color will only go to a particular set in cache. So, if my page physical page frame is red I know that it has to be if the virtual address will always map it in such a way that it gets to that that set always gets the same page color.

So, then what happens the number of cache sets, how do I determine the number of cache sets? I have the total cache size is 1 MB and I have 64 I have a 64 byte.

So, a cache block is 64 bytes; so, 64 bytes in a block. So, block offset is 64 bytes 64 bytes and the number of blocks in each set the number of blocks in each set is 16, I have a 16 way set associative cache. So, 16 way set associative cache. So, the number of blocks in each set is 16 and therefore, one MB is 2^{20}; 64 is 2^6 and 2^4. This is the number of blocks in each set. And therefore, the number of sets in the cache is 2^{10}.

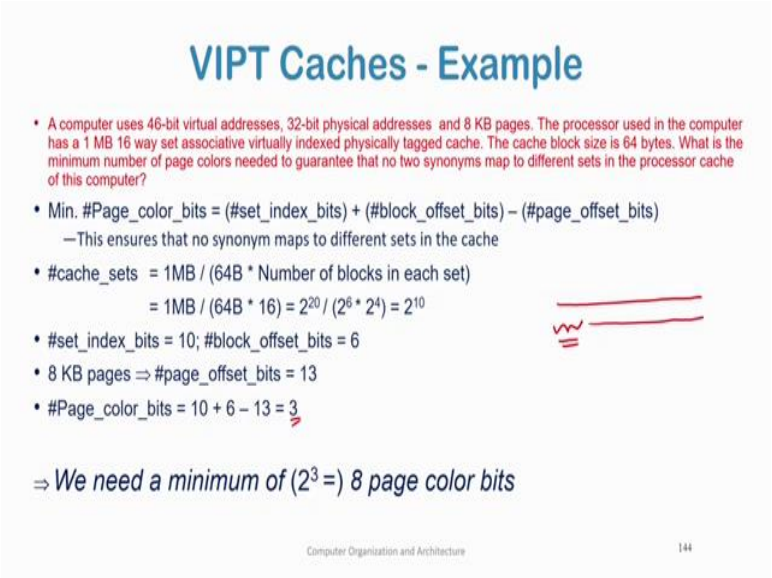
Therefore, I require 10 bits for set index bits. So, set index bits is 10 and block offset bits as we saw here block offset number of blocks in the in a in a set number of number of bytes in a block is the block offset, sorry, sorry, the number of bytes in a block is the block offset. So, the block offset bit is 6.

(Refer Slide Time: 37:54)



So, now then what happens is that um. So, now, so, this plus this plus this is 10 + 6 = 16. Now I have 8 KB pages. So, number of page offset bits is 13.

(Refer Slide Time: 38:12)



And so, number of page colored bits that will be required is 10 + 6 - 13 is 3. Hence, we need a minimum of 2^3 because I have 3 bits. So, the page offset is 13 bits, but I am using 16 bits to access the index the cache. And therefore, these 3 bits bring in the problem of synonyms. And therefore, I colored my page frames of my physical memory into 8 different colors such that when I map a page of the same color, I will map the virtual address to physical address will be mapped in such a way that a set in cache always gets pages of the same color.

(Refer Slide Time: 39:02)