

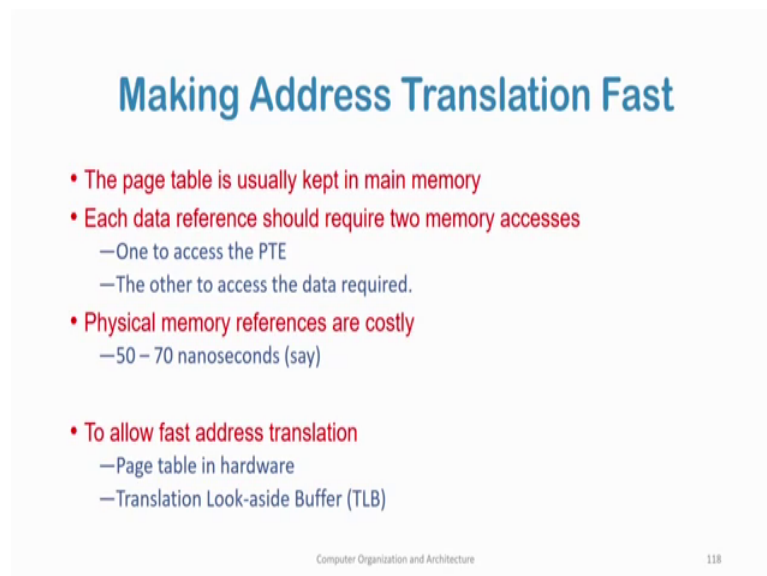
Computer Organization and Architecture: A Pedagogical Aspect
Prof. Jatindra Kr. Deka
Dr. Santosh Biswas
Dr. Arnab Sarkar
Department of Computer Science & Engineering
Indian Institute of Technology, Guwahati

Lecture – 29
TLBs and Page Fault Handling

In the last lecture we saw that in the absence of any measure the sizes of page tables can be huge. Therefore, we studied different techniques by which the sizes of page tables can be controlled.

This lecture we will start with the discussion on how address translation using page tables can be made faster.

(Refer Slide Time: 00:52)



Making Address Translation Fast

- The page table is usually kept in main memory
- Each data reference should require two memory accesses
 - One to access the PTE
 - The other to access the data required.
- Physical memory references are costly
 - 50 – 70 nanoseconds (say)
- To allow fast address translation
 - Page table in hardware
 - Translation Look-aside Buffer (TLB)

Computer Organization and Architecture 118

What is the motivation? As we discussed page tables are usually kept in main memory; therefore, for each data reference we will typically require two memory accesses if we do not take any measure.

One access of which will be to access the page table entry itself and then when we access the page table entry we get the main memory address or we get the main memory page number and we generate the physical address subsequent to that and then the second memory reference will be to access the actual data that we require from main memory.

Now, main memory references are typically very costly with respect to if we had found the data in cache let us say; main memory accesses we said would be of the order of 50 to 70 nanoseconds; as against cache which could be around 5 to 10, 1 to 10 nanoseconds ok, around 5 to 10 nanoseconds.

And therefore, it is necessary that we take action to reduce this page table access in main memory. Now, there are two typical strategies which are employed here. The first one is to implement the page table in hardware and the other one is to use a translation look aside buffer.

(Refer Slide Time: 02:20)

Page Table in Hardware

- The page table may be implemented as a set of dedicated registers
 - Applicable in smaller platforms – embedded systems
 - During a context switch
 - the CPU dispatcher reloads these registers along with other registers and the program counter
 - to restore the saved state of a process it wants to activate.
- Example: DEC PDP 11 architecture
 - 16-bit logical addresses with 8 KB page size
 - Page table consists of 8 entries in fast registers
- Impractical in computers with large logical address spaces
 - A 32-bit computer may require millions of PTEs

Computer Organization and Architecture 119

When we implement the page table in hardware how do we do it? We implement the page table using a dedicated set of registers and obviously, it is applicable for systems where the page table sizes will typically be smaller. For example, in embedded systems. Now, during a context switch when a new process has to be brought into the CPU.

The CPU dispatcher will reload these page table registers along with other registers and the program counter. In order to restore the save state of a process and activated; So, basically during a context switch what happens? The saved state of a process is brought into cache.

And what are what is the typical what do we mean by the saved state of a process? The contents of its page table and the contents of its other registers and also the program counter.

So, therefore, after the context switch when the process is brought into memo memory the because we have the saved program counter known, so the process can start from the from the place where it was previously evicted from CPU.

Now, when the page table is in hardware I have to reload all the registers in page table during a context switch because that is part of the saved state. If the page table is in memory it is sufficient to load the page table base register corresponding to this process. Because the page table is in memory; I only need where in memory the page table starts.

So, therefore the page table base register only needs to be brought in during a context switch. However, when I am implementing this page table in hardware I need to bring in the entire set of registers which is the page table into the hardware after the during the contact switch.

An example of such hardware implemented page tables is the DEC PDP 11 architecture. the DEC PDP 11 architecture is a 16 bit small computer. So, it has 16 bit logical address space with 8 KB page size. So, therefore, if it is a page 8 KB page size, so therefore, the offset part of the virtual address is 2 to the power 13 bits. So, 2 to the power 10 into 38 KB; so, 2 to the power 13 bits is the size of the page offset part in the virtual address.

So, the number of pages is just, you consumes only 3 bits; so, 16 minus 13 3 bits. So, therefore, the page table the system contains only 8 pages and therefore, the page table consists of 8 entries in fast registers. However obviously, such hardware implementations of page tables are impractical for computers with very large address spaces.

For example, if we have a 32 bit computer and let us say in that computer we use 4 KB pages for example, then I will use 12 bits for the page offset part and therefore, I will have 20 bits for the page numbers.

And therefore, we have 2 to the power 10 into 2 to the power 10; 2 to the power 20 entries in the page table which is 1 million; 2 to the power 20; 1 million page table

entries and obviously, such very huge sizes of page tables cannot be implemented through registers in hardware.

(Refer Slide Time: 06:10)

Page table in hardware – Example

- A machine has a CPU with a 32-bit address space and uses 8K pages. The page table is entirely in hardware, with one 32-bit word per entry. When a process starts, the page table is copied to the hardware from memory, at a rate of one word every 100 nanoseconds. If each process runs for 100 milliseconds (including time to load the page table), what fraction of the CPU time is devoted to loading the page tables?

Computer Organization and Architecture 120

Before proceeding we take an example. A machine has a CPU with a 32 bit address space and uses 8K pages. The page table is entirely in hardware with one 32-bit word per entry. When a process starts the page table is copied to the hardware from memory. At a rate of one word every 100 nanoseconds.

So, therefore, the page table is in hardware. So, we are trying to look at what is the problem for big computers in a 32 bit computer if the entire page table is in memory. We will look at one of the computers in addition to space.

So, the machine has 32 bit address space and it uses 8K pages; the page table is entirely in hardware with one 32-bit entry per word 32-bit word per entry. When a process starts the page table is copied to the hardware from memory at a rate of one word every 100 nanoseconds.

So, to copy one word from memory it takes 100 nanoseconds. If each process runs for 100 milliseconds and this includes the time to load the page table what fraction of the CPU time is devoted to loading the page tables.

(Refer Slide Time: 07:34)

Page Table in Hardware – Example

- A machine has a CPU with a 32-bit address space and uses 8K pages. The page table is entirely in hardware, with one 32-bit word per entry. When a process starts, the page table is copied to the hardware from memory, at a rate of one word every 100 nanoseconds. If each process runs for 100 milliseconds (including time to load the page table), what percentage of the CPU time is devoted to loading the page tables?

- Page size = 8K = 2^{13}
- Total number of page table entries = $2^{32} - 2^{13} = 2^{19} = 1024 * 512 = 524288$
- Total time to load page table = $524288 * 100 / 10^6 \approx 52.4$ milliseconds
- Thus, 52.4 % of the CPU time is spent loading page tables.

Computer Organization and Architecture 124

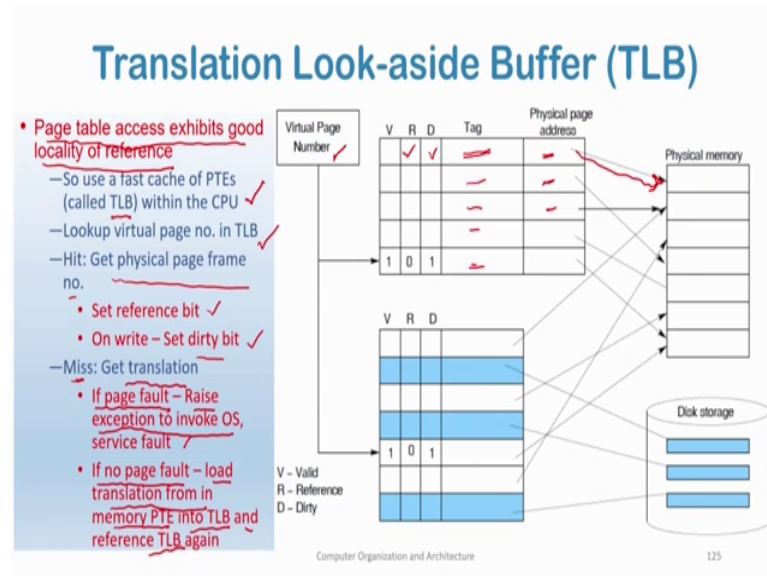
So, the page size is 8K or 2 to the power 13. So, 13 bits we consume for the page tables. So, therefore the total number of page table entries becomes 2 to the power 32 minus 2 to the power 13; which is 2 to the power 19; which is 2 to the power 10 into 2 to the power 9 or 524288. So, these many page table entries are there in the page table. All these have to be copied from memory into hardware during a context switch.

So, the total time to load the page table will be what; 524288 and each of them consumes 100 nanoseconds. If you convert it to milliseconds divided by 10 to the power 6, we get 52.4 milliseconds. So, therefore, 52.4 percent of the total CPU time in this case he spent loading page tables.

So, I have a time slot size in which I will execute a process and I execute a process for 100 milliseconds, for each process I give 100 milliseconds and out of that 52.4 milliseconds is only consumed for loading the page table from memory into hardware.

So, basically I am not being able to do anything any good work other than loading the page table from memory into hardware which is very costly. So, therefore, when the page table is in hardware it is only possible in cases where the address space virtual address space sizes are small and we have a small number of page table entries.

(Refer Slide Time: 09:25)



So, then how to what is the other way around when we have large address spaces? So, how to control the time consumed for accessing page tables and doing address translation for large computers with large virtual address spaces?

For example, says 32 bit computers, how will we do that? We typically do that using in memory page tables. So, we now we do not keep the page tables in hardware. We keep the page tables in memory itself. However; we use the property of the locality of reference of page tables. So, therefore, here is the first point page table access page table access exhibits good locality of reference.

So, therefore, once a page table entries access it is supposed to be accessed soon again. So, the same page because our memory references, our memory accesses tend to be clustered in both time as well as space. So, the data that I am consuming nearby data I can I would consume in the recent future; nearby data I will access in the recent future and the same data I may access again in the recent future.

So, I have both temporal locality as well as spatial locality for page memory accesses. And therefore, the same happens for page table accesses page table entries, the same page table entries is likely to be accessed soon again. So, therefore we can use a fast cache for page table entries and this cache is called the translation look aside buffer within the CPU.

So, we look up the virtual page number in the TLB. So, what do we have? We get the virtual page number is generated by the process that is the CPU and I look up that virtual I divide that virtual page number into the page offset part and the page number part. So, the page number part is now floated to the TLB. So, the TLB, the TLB the stack part of the TLB contains virtual page numbers.

So, we look up the virtual page number in the TLB. If I get a match, if I get a match with the tag I get the corresponding physical page number from the data part of this TLB. So, the tag part of the TLB contains a virtual page number and the data part of this TLB contains the corresponding physical page number. So, when I get the physical page number I can access the physical memory by adding up with the page offset which is directly obtained from the virtual address itself.

Now, during a hit what happens? We get the physical page number and we get the and as I said I get I can get the data from physical memory. I said the physic I said the reference bit.

So, this one R is the reference bit; I said the reference bit to indicate that in this in the in the in the current time epoch I have referenced this page to indicate that I said the reference bit. If this I am writing on to this on to main memory, if I am writing on to main memory I also said the dirty bit on; So, therefore, to indicate that this page has been modified.

However, when I get a miss, if the if my virtual page number does not match with any of the tags present in the in the in the translation look aside buffer, then I have a miss in the page table ok. If I have a miss in the page table I need to get the translation from memory from the page table. Now, in this case there can be two distinct cases that I have a miss in the TLB, but the data is in memory.

Therefore, the data is also in the page table and I just get the translation from the corresponding page table entry in memory and I bring it back into the TLB and then reference the TLB again and get the data from physical memory. Otherwise I can induce a page fault.

If the data that I am looking for the corresponding to the virtual page number there is this physical page number translation this virtual page number to physical page number

translation is not there in the page table in the memory itself; which means that the page is not there in the memory I incur a page fault.

And therefore, then I need to find out a free page frame in the memory, bring the page from the disc into main memory and then populate the page table accordingly and half subsequently I need to bring that page table into the translation look aside buffer and then subsequently when I access I get a hit in the TLB and I get the page number from the TLB itself and you and access the physical memory.

So, if there is a page fault; that means, that the page is not there in the memory. The CPU raises an exception; that means, it traps the OS; it traps to the OS it generates a trap instruction and the OS is invoked I go to the supervisor mode as the OS is invoked to service the fault in the way as we said we will see it in more depth in the in the subsequent slides.

How a if there is no page fault; that means, I load the translation from where; from the in memory page table, from the in memory page table I bring the page table entry from the in memory page table into the TLB and reference the TLB again; I reference the TLB again. Now, I get a hit in the TLB and access physical memory.

(Refer Slide Time: 15:48)

Translation Look-aside Buffer (TLB)

- Typical values for a TLB ✓
 - Size: 16–512 PTEs; Block size: 1–2 PTEs (4–8 bytes)
 - Hit time: 0.5–1 cycles; Miss penalty: 10–100 cycles; 0.01%–0.1% miss rate
- Associativity ✓
 - Fully associative: for small TLBs ✓
 - Smaller associativity for larger TLBs
- Replacement ✓
 - LRU: expensive to implement for large TLBs with high associativity ✓
 - Solution: Random replacement ✓
 - Typically write-back: to copy reference and dirty bits back to PTE on replacement

• Efficient – since TLB miss rates are expected to be small

Computer Organization and Architecture 126

So, a few more details with the translation look aside buffer. Typical values for the TLB are as follows. Typical for a TLB: the size of a TLB are of the order of let us say 16 to

512 page table entry. So, page table entries and each such page table entry could be 4 to 8 bytes. So, therefore, TLBs are typically small in size. The block size for a TLB is 1 to 2 page table entries and if a page table entry is let us say 4 bytes; so, 2 page table entries; therefore, mean just 8 bytes.

The hit time of a TLB is very fast typically of the order of 0.5 to 1 cycle. A miss penalty would result in what? Would result in going to the lower level of memory and therefore, I it will be around 10 to 100 cycles. Why, why this big difference? Firstly, because different architectures can support different and different access times and a miss penalty for corresponding to TLB miss, I may get the data either in the data cache for the corresponding TLB entry or I may not have the data corresponding to the TLB entry in the data cache. In that case the TLB entry must be brought from the memory itself.

So, therefore, there can be two possible cases when I have a miss and I want to get the page table entry into the TLB. And typical miss rates for a TLB are of the order of 0.01 to 0.1 1 percent. So, therefore, the locality of reference for corresponding to TLB entry is typically very high and so, smaller TLBs suffice. So, it is around ninety 99.9 percent to 99.99 percent is the hit rate.

The associativity of a TLB: TLBs are very often implemented in a fully associative fashion; which means that all the entries of the TLB; So, which means that because the TLB is small, I have the option because the TLB is small what happens is that; when it is fully associative the page table entry can be brought in to any entry of the TLB when it is fully associative.

I do not restrict the page table a page table entry to go into certain specific locations in the TLB. The entire page table all entries in the page table can hold any all entries in the TLB can hold any page table entry when it is fully associative.

The drawback is that I when it is fully associative all the entries of the TLB must be searched in parallel for the tag. So, therefore, the search time and also the replacement time we will as we will see later becomes high and therefore, the cost of a fully associative TLB is high.

And this is affordable for small TLBs; however, when the sizes of TLB is grow, we nowadays we have a higher sizes of TLBs; smaller associativitys are preferred as we understand why.

Replacement strategies how do we replace TLB entries? When I do not find when there is a miss in the TLB, I need to replace. So, least recently used TLB entry if I want to replace this is typically expensive to implement for large TLBs with high associativity. Why? Because when the TLB sizes are large finding the least recently used in implementing the least recently used in hardware finding which is the least presently used at any time which entry, keeping that account is expensive.

How do we keep account at any time which entry is least recently used? So, therefore, I need to maintain some kind of a clock or a queue corresponding to each TLB entry and that has to be implemented in hardware. So, that at any point in time I can keep track of the least recently used TLB entry.

And when the associativity becomes higher, this search or keeping this maintaining this becomes higher. Because the number of places within, for example, when it is fully associative any entry in the TLB can be least recently used. So, I may need to search the entire TLB. In a set associative one, I only need to search within the set. So therefore; when the associativity is high and when the associativity is high, it is expensive to implement for large TLBs; LRU is expensive to implement for large TLBs, especially and when the associativity is high.

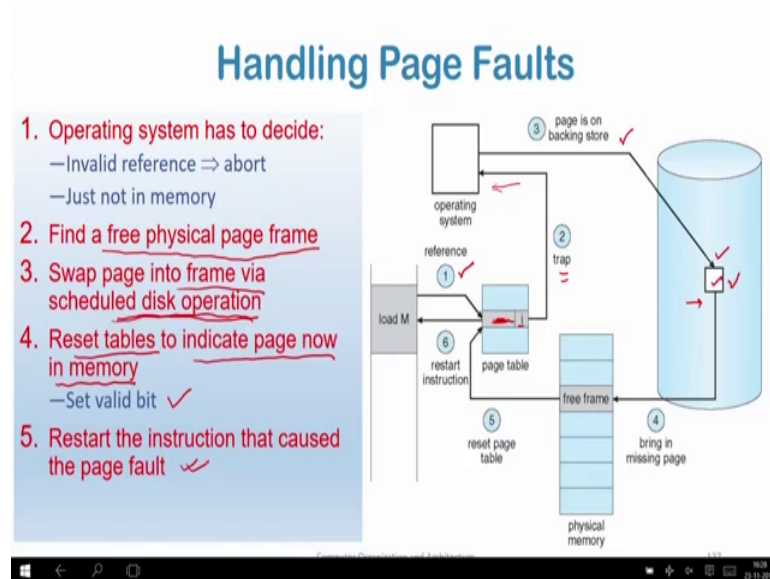
So, what is the solution? It is a bypass. What is the solution? The bypass is that we use random replacement. I just find out a TLB and replace it randomly. I do not go for the least recently used. For a TLB we typically implement write back instead of a right through.

So, why do we write? We copy the reference and dirty bits back into the TLB entry on replacement. We said that when there is a TLB hit, what happens is that I may set the reference bit, I may set the reference bit to indicate that this page table entry was reference. I may also said the dirty bit to indicate that and that this page table this page table entry was written to this page was written to ok.

Now, when I am replacing this TLB entry I must write back the reference and dirty bits the values of the reference and dirty bits into the corresponding page table entry in memory. And I do not do this and I am using a write back means that, I do not do this writing when I am basically changing the reference or dirty bits at a given page table entry access. That becomes very expensive because I have to write it back to the next level to the page table entry in memory if I want to write through and this becomes very costly.

And as we said that the page table miss rates are expected to be small as we saw it is of the order of 0.01 percent to 0.1 percent. So, page table miss rates are small. So, write back is very good in this case and it saves a lot of time and the implementation is much more efficient when we use a write back scheme instead of a write through and that is I write only during replacement and I do not write and I do not write on every reference to the page table entry, every reference to the page table.

(Refer Slide Time: 23:09)



So, now we will take a deeper look into page faults. So, during a page fault when I do not have the required data in memory, I incur a page fault. So, at that time the page table entry corresponding to the page that I want to access shows that it is invalid. That is the valid bit is 0; that means, the corresponding physical page number is not mapped to the page table entry.

In that case what happens is that; firstly, I need to decide whether this translation that I want to do is for an invalid reference or for a valid reference. If it is for a invalid reference, that means, that this particular virtual address is not present in my virtual address space itself.

This can happen for scattered virtual address spaces for of a process. If this virtual address is not part of the is not part of the address space of the process itself, then this is an invalid reference and we immediately abort. Otherwise we see that the valid bit is 0 and therefore, the page is just not in memory. So, therefore, it has to be brought from the disc or the secondary memory.

So, then what happens? In I have to find a physical page frame; I need to find a physical page frame. So, if you see the sequences that are happening here I try to reference in number 1, I try to reference and I see that the corresponding mapping is not there; it is it is not there. Therefore, I will have to trap the OS; I will have to trap the operating system to indicate that this is a page fault.

You know the operating system we will first find out what kind of trap has it received. So, it will then find that this is a page fault type of a trap. And therefore, then what the OS will do? It will find a physical page frame in the physical memory, it will find a physical page frame in the physical memory and then maybe through replacement of an existing page in physical memory which will not bother right now will do will see bit later. Then it will swap page into this frame via schedule disc operation.

So, now, when I have found a free page frame in physical memory I will need to bring the page that I require from the backing store or the secondary memory into the physical memory. And I said that how do we get that? In a in a in a class earlier class we said that in addition to the mapping of a virtual memory corresponding to virtual memory and a physical memory, physical page number the I also the page table additionally logically maybe in a different physical table.

It also contains where in the backing store I have this page I have this page; that means, corresponding to a virtual page number the page table in addition to keeping the physical page number, it also keeps the address of where in the secondary memory this page is resident.

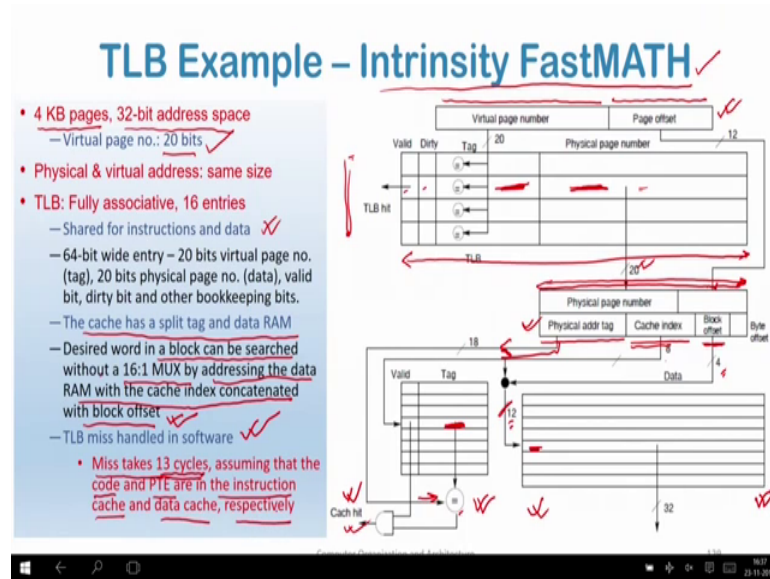
So, when I get a trap the OS finds out firstly, the physical frame where which is free and where I can give get the page into the physical memory where I can bring it and then it finds out where in the in the secondary memory the page is corresponding to this virtual page number.

Then it gets the page from the secondary memory through scheduled disc operation. So, this schedule disc operation means that for the disc I have a queue of a of requests I have a queue of requests who want to access pages from the disc and I have to incur I have to go through this queue and get it from the secondary memory and I have to incur seek time and less latency time of the disc. And then through this disc operation I will swap in this page into the physical memory.

After I have brought in this page I know the page number. So, therefore I have to reset the tables, I have to reset the tables to indicate that the page is now in memory. So, I have to I have to update the page table entry corresponding to this page, corresponding to this virtual page, I will update the page table entry and write where what is the current physical page number corresponding to this virtual page number. I will also set the valid bit to indicate that the that the virtual this virtual page is currently now in physical memory ok, I will said the word to valid bit.

Then I will restart the instruction that caused the page fault. So, I will then restart the instruction. So, in a particular memory access I had a page fault, then I service the page fault, then I will restart the memory and then in the subsequent memory reference I will get the data that I sought for in the physical memory.

(Refer Slide Time: 29:00)



Now, we will take an example of a practical architecture which the TLB of a practical architecture ok. So, we will take a look at the intensity FastMATH architecture and in this architecture we have a 4 KB pages and 32 bit virtual address space.

So, if we see here in this virtual page number we see that it has a 20 bit virtual page number and 12 bit page off offset; which means that see these page size is 4 KB 12 bits. Why? Because right for 2 to the power 2 into 10 KB. So, basically I have to using 12 bits I have I have a page offset of 12 bits and so, the page size is 4 KB.

And because I have a 32 bit address space, the virtual page number is 20 bits. Now, here we see that the physical page number, the physical page number here because this one is also of 20 bits, this one is also of 20 bits; the physical page number and the physical the physical address space is also of the same size as the virtual address space. The physical page physical address is also 32 bits ok.

So, the TLB is fully associative; as we as we told what it means it is fully associative and it has 16 entries in the TLB. So, the TLB is fully associative having 16 entries and this TLB is shared for instructions and data. So, it is a translation look aside buffer for both instructions and data. Each TLB entry is 64 bits wide. So, this one is 64 bits wide and it contains the 20 bit virtual address. So, this part is 20 bits; it contains the sorry 20 bit virtual page number. So, this one is 20 bits; it contains the 20 bit physical page number.

So, this one is again 20 bits; it contains the valid bit, dirty bits and other bookkeeping bits. So, the total address so, the total size of one page table entry is 64 bits.

The we if you note the cache is split; cache has a split tag and data part. So, basically what happens? When there is a TLB hit, when there is a TLB hit I get the corresponding let us say the TLB I the virtual page number matched in this entry and I got the corresponding physical page number; the 20 bit physical page number and therefore, I generated the physical address here. After generating the physical address I divide the physical address into 3 parts, I split it. One is the physical address tag part, the cache index part and the block offset part; So to access what? To access the cache ok.

Now, here now subsequently after dividing this physical address into splitting it into three parts; the tag part, the cache index part and the block offset part I will access the cache. And this cache is implemented again as a split tag and data. So, what happens here? The tag part if you see here, the tag part is the physical tag part is matched with the tag part of the of the cache. The tag part of the cache is here, I match it with the tag part of the physical address, I match it with the tag part of the physical address here, I match it and then if the valid bit is on I get a cache hit ok.

Now, at the same time because I have now split the data part of the cache what do I do? Why do have I done that? I now am by splitting it I now am able to concatenate the cache index part and the block offset part as a 12 bit value and this 12 bit using these 12 bits I directly access the data data data word that I require; I directly access the word. So, if I did not concatenate this, so, I what would I have to do? I would have to access the given block and within that block I have to do because the block offset is contains 4 bytes; which means that I have 16 words within each block.

Now, I have to I had to compare this the 16 words; I had to compare because each I had to compare these 16 words and using a 16 cross 1 multiplexer to find out which data word I actually need. Now, what happens? I have the tag part; I have the tag part and the block offset part.

So, basically I have got the exact data word. I do not have to do this, I do not have to go into the block and then using a 16 cross 1 MUX and the block offset, I do not have using the using the block offset as the select line, I do not have to use the 16 cross 1 MUX with block off offset as the select line to and to know which exact word within this block

should I access. I do not have to do this. I do not have to use a 16 cross 1 MUX with the block offset as the select line to know which word within the block do should, I correctly access.

Instead I have concatenated the cache index part and the block offset part to generate a 12 bit value and I directly access the data I directly access this data RAM part of the cache and get the required word and I know that the required word is correct because I have a cache hit here ok. So, the desired word in a block: the cache has a split tag and a data RAM. The desired word in a block can be searched without a 16 cross 1 MUX by addressing the data RAM with the cache index concatenated with block offset ok.

Now, in this architecture a TLP miss is handled in software. So, how do I handle this? If I have a TLB miss what do I do? I take the virtual page number and I save it you know hardware register. Then I trap the OS and say that I have a TLB miss. So based on this, the OS generates special instructions to go into the to find the page table entry using the page table base register and the virtual page number part; virtual page number part and the page table base register combination. It gets into the page table entry and brings the required page table entry.

Now, the page in a TLB miss requires only 13 cycles in this system when we when we consider when we assume that the code and the page table entry are in the instruction and instruction cache and data cache respectively. So, this TLB miss has an overhead of only 13 cycles when the code and the TLB entry are in the instruction cache and in the data cache.

(Refer Slide Time: 37:19)

Memory Hierarchy Operation - Example

- In a memory hierarchy organized with a physically indexed physically tagged cache and physical memory along with a TLB for fast accesses, a memory reference can encounter three different types of hits/misses: a TLB hit or miss, a page table hit or page fault, a cache hit or miss. Consider all the combinations of these three events (eight possibilities). For example, one combination is that: a memory reference resulted in a TLB miss but page table hit as well as cache hit. For each possibility, state whether this event can actually occur and under what circumstances. ✓

TLB	Page Table	Cache	Possible? If so, under what circumstance?
H	H	H	Possible; but we will never look in the page table
H	H	M	-00-
M	H	H	possible:
M	H	M	possible:
M	M	M	possible:
H	M	M	Impossible:
H	M	H	Impossible:
M	M	H	Impossible:

Now, we will continue our discussion and take an example here to understand to illustrate as to how the memory hierarchy works in unison together. How they cooperate among each other and what happens when they work all together? So, memory hierarchy in operation, the example goes like this.

In a memory hierarchy organized with a physically indexed physically tagged cache and physical memory along with the TLB for fast accesses. So, what I have? I have a physically indexed physically tagged cache; meaning that the cache is addressed by the physical address by the physical address.

It is not using virtual addresses to address the cache. As opposed to other we will see over a technique in the next class, two techniques in the next class; where part of the full or part of the virtual address is used to address the cache. Now, in this hierarchy this does not happen. So, I generate the physical address and then from the physical address I address the cash.

So, it is a physically indexed physically tagged cache. So, I index the cache using the part of the physical address and I tag the cache using part of the physical address. So, in a I have a memory organized with a physically index physically tagged cache and physical memory along with. So, I have a cache and I also have a physical memory along with a TLB for a fast accesses. So, a memory reference can encounter three different, in this system a memory reference can encounter three different types of hits or misses. It

can encounter a TLB hit or miss, it can encounter a page table hit or miss or and a cache hit or miss.

So, consider all the combinations of these three events. So, you have 8 possibilities, you have 8 possibilities. For example, one combination is that a memory reference resulted in a TLB miss, but a page hit and a so, I have a TLB miss, but I have a page hit and a cache hit. For each possibility we need to say whether this event can actually occur? Whether this event can actually occur?

Whether it is possible in practice that these that that are given three even that they get that an event of a combination of hit and a hit or miss will actually occur. And if so, under what circumstances can it occur? For example, can we have a hit in the TLB, a hit in the page table and a hit in the cache? Yes we can.

But obviously, when I have a hit in the TLB what happens? When I have a hit in the TLB, so this is possible, this is possible. But, but we will never look in the page table, page table. Because I have a TLB hit because I have a TLB hit we will never look in the page table ok. So, and then I have when I have got it in the page table, when I have got it in the page table I have a cache hit. Then this is possible. So, the page table hit I got and then I got the physical page number and after I got the physical page number, I got the data subsequently in cache.

Now, I have a TLB hit and a page table hit and a cache miss this is also possible; DO. But what happens? This is possible, but obviously, I will not check the page table because I have already got a hit in the cache. So, I will not go get into memory and look actually, look into the page table because I have a hit in the TLB. Now when I have a miss in the TLB, yes I can have a miss in the TLB a possible, possible. So, I can have a miss in the TLB and then I will I will go into the page table lower level of hierarchy and find out whether the page table entry is there or not.

And if I have a I then basically here it says that I have a page table hit. If I have a page table hit then basically it is in memory and if it is in memory it may or may not reside in cache. In this case I am saying that it is a hit in the cache. So, it is in the hit in the page table. Therefore, the data is in memory and it is a hit in the cache, so the data is also in the cache.

So, this is possible. The next case I have a miss in the TLB, I have a hit in the page table and I have a miss in the cache. Yes, this is also possible as we said I have a miss in the page table. So, I go to get into the next level, get the page table entry. I will find that the page; for the page table entry the data the actual page actually resides in physical memory.

However, when I generate the physical address breaking into a tag, cache index, block offset, etcetera for accessing the cache; I see that the data is not present in cache. So, therefore, I have to incur a memory reference physical memory access and get the data from physical memory, I will not get it from the cache directly. It is also possible that I will have a miss in the TLB, subsequently, I will look into the page table and I see that the data is not in the page table.

So therefore, I will basically the data does not reside in the main memory and I will also have a miss in the cache. So, this is also possible. This is also possible. So, I will obviously, because there is a page table miss I will also have a miss in the cache. It cannot be that my page it is there it is not there in the main memory or data is not there in the main memory, but the data is there in the cache; this cannot happen.

Now, let us see the next case. I have a hit in the page table, but I have a miss in the TLB. Is this possible? This is not possible because if I have a hit in the TLB, TLB acts as a cache for the page table. So, if I have a hit in the TLB, the data must also be there in the page table and the entry must also be there in the page table. So, it is not possible that I have a hit in the TLB, but I have a miss in the page table.

Similarly, this also this is also impossible. It cannot be that I have a hit in the TLB and a miss in the page table. I have a miss in the page table, I have a miss in the TLB, I have a miss in the page table and but I have a hit in the cache. Is this possible? This is also impossible. Why? Because; I have a miss in the page TLB fine; So, therefore, I have a miss in the page table; this part is fine.

However, I have a miss in the page table means that I have a page fault. I have a miss in the page table means I have a page fault. So, the page is not resident in main memory. So, when a particular page is not resident in main memory, the

corresponding block cannot reside in the cache. So, this is impossible. So, we will end this lecture with a small example.

(Refer Slide Time: 45:43)

Page Fault - Example

- If an instruction takes time m if there is no page fault, and time n if there is a page fault, what is the effective instruction time if page faults occur once every j instructions?

$\text{time } m : \text{page hit}$
 $\text{time } n : \text{page fault}$

$$m + \frac{n-m}{j}$$

$$\frac{(j-1)m + n}{j}$$

→ Effective instruction time

Computer Organization and Architecture 131

If an instruction takes time m if there is no page fault and time n if there is a page fault, what is the effective instruction time if page faults occur every j instructions? So, we see that time m when page hit, I have a page hit and time n when I have a page fault ok. So now, I am saying that in a group of j instructions I have one miss and the all other are hits. So, in this system page faults occur once every j instruction.

So, for j minus 1 instructions, I have the time required is m and for 1 instruction the time required is n . So, every j instructions what happens is that for j minus 1 instructions, I take m time and for one other instruction I take n time ok. So, therefore the effective instruction time; So, what is the average instruction time? Then the average instruction time is given by this. So, this is basically m plus m plus what m plus n minus m by j . So, this is the effective instruction time.

With this we come to the end of this lecture.