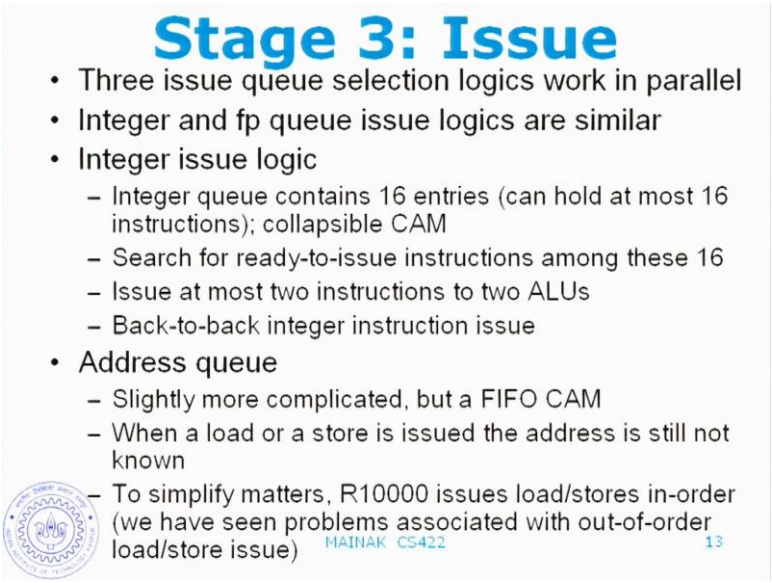
Computer Architecture
Prof. Mainak Chaudhuri
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture - 32
Case Study: MIPS R10000

(Refer Slide Time: 00:13)

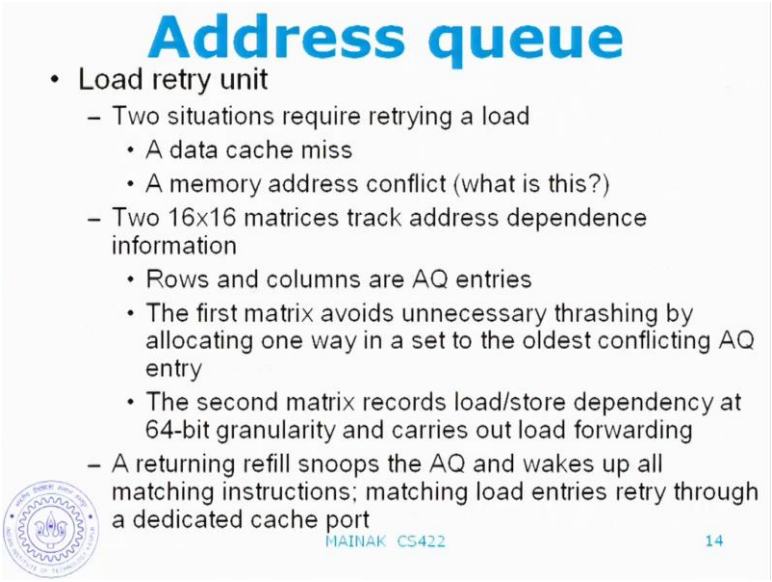


Last time we discussed 2 pipeline stages fetch and decode combined. So; moving on today, we will look at the remaining back stages coming on. So, the third stage is the issue instruction issue stage and there are 3 issue queues that you have last time integer queue, floating point queue and the address queue 3 issues logics worked in parallel because they are independent queues. Integer and floating point queue issue logics are several. So, I will not go with the details of this because we discussed it we discussed they look very similar. So, here is the summary of the integer which should be logic.

The integer queue contains 16 entries which can hold at most 16 instructions and its collapsible for bit data memory. So, these are also we discussed last time which allows you to compare value with all the entries and select certain entries. So, this is needed for comparing the register files to make up instructions; addresses changing in a instruction for ready to issue instructions among these 16 issue, at most 2 instructions in 2 ALUs and it can issue back to back integer instructions. So, that is what we discussed its possible because we can think up the values with the address scheme is slightly more complicated it is not a collapsible cam.

So, this 1 is collapsible because we can issue instructions all of the program orders which mean whichever gets ready the issue that will create all holes in the queue. So, to free up the slots towards the collapsible. So, that the holes can move to 1 end where you can keep allocated; this 1 is a FIFO CAM meaning that instructions cannot issue all the time they will only issue in order and whether load or a store is issued the address is still not. So; we had also discussed this problem earlier that, what kind of complications that can arise because of this. So, to simplify matters what R ten k does is that it issues loads and stores in order. So, which is like this a FIFO CAM it is not a collapsible CAM. So, that is the address queue.

(Refer Slide Time: 02:48)

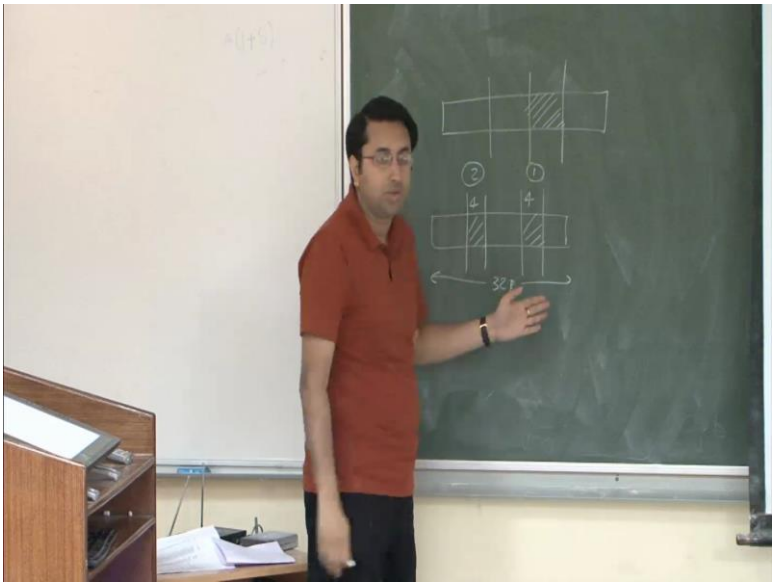


There are, couple of more logics associated with the address queue. So, there is a load retry unit. So, let us try to first understand what we require retrying a load instruction. First of all is the data cache miss that is load instruction that misses the data cache. So, what we essentially have to do is that you launch a miss request. So, that a data comes back from somewhere hierarchy and when the data comes back the load will have to retry it once again. So, reissue the load and hope fully this time we will hit the cache. And there is a memory address at the second situation, where you really want to retry a load is from memory address conflict. So, this 1 we have to discuss by discussing caches also little.

So, what happens is that suppose we have a cache miss. So, what you need to do is you issue the miss sequence to the next level of the memory hierarchy and also you start selecting replacement candidate. So, this where the data come back when the cache.

Now, the now the problem arises if why that, miss is outstanding to get 1 more request to the same caching list which also misses the cache right. So; now, of course, we can say that well I can continue during my replacement running by the replacement policy without the already selected replacement candidate.

(Refer Slide Time: 04:26)



So, essentially if I have a 4 way cache I have 4 candidates to select from the replacement. So, I will from the first cache list already select this 1 from the replacement and before this 1 comes back, there is another request that goes to the same cache limits. So, what are the options do I have; well, I can say that you cannot use this 1 for replacement anymore then this 1 is already kind of ok. You can only select 1 from these three. So, instead of with that does is that these are all the second request will actually retry once this miss which essentially says that, you can have at most 1 cache miss request per cache set outstanding it.

So, that is 1 example of memory address conflict. The second example is easier where, suppose you have a load instruction which missed in the cache and the and the request has gone out to be fetched and within a second load instruction which maps you same cache block that has just missed. If there is a cache block with several bytes, so I can

have 2 load instructions this is my cache block and instruction loading from here, another load instruction there is loading from here.

So; these are may be 4 bytes each whereas, I have a cache blocks say thirty 2 bytes and suppose that this 1 is, this 1 happens first and this 1 goes second alright. So, when this miss happens we have already set a request from this cache block. So, of course, then it does not make any sense when you have this request again. So, in this case what we will do is that; we will reject this load and we will retry to the cache block that is another example of a memory address conflict, it is not exactly a conflict its essentially a request mapping to a cache block which is already outstand. So, for both of these you retry your load later.

So, there are 216 plus 16 matrices that track address dependence information and rows and columns are address queue because there are the first matrix avoids a necessary thrashing by allocating 1 way in a set to the oldest conflicting address to. So, you make sure that, the oldest entry always makes by allocating replacement candidate. The second matrix records those 2 dependency and 64 bit and caries out low forwarding. So, this all we discussed already that, if a load overlaps with an already outstanding store in the load can take the value from the store, if it overlaps completely. So, that is that is what is tagged by the settlement. Returning refill. So, a refill is essentially a message that happens responsible cache miss.

So, e send a cache miss request to eventually the refill comes back in the cache. So, returning refill snoops the address queue and makes up all matching instructions. So, there may be multiple load store instructions making for this particular cache block, when that comes back you can essentially compare the cache block address with all the pending addresses with 16 entries to the queue and anybody that matches will reach through a dedicated cache block. So, the this 4 allows 1 retry per second. So, we retry 1 after another.

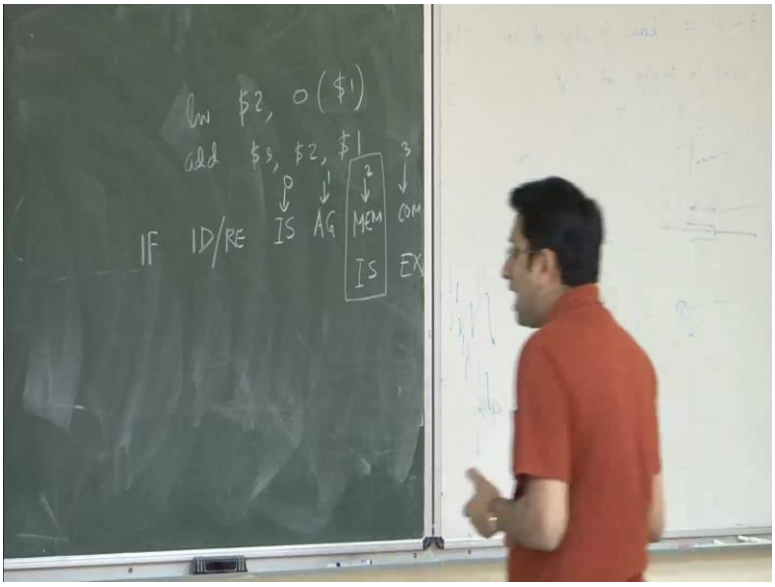
So, that is pretty much the address queue logic. So, as you can see there are 2 simplifying factors that help in the design, 1 is that the address queue is a FIFO queue. So, in order that takes care of your load by passing a store in the same address and getting a wrong value that we will discuss earlier in the class. And the second simplification is this 1 that you can now only at most 1 cache list from cache index.

Yeah.

So, this matrix essentially keeps track of your index conflicts, this 1 alright and if there are 2 cache requests, 2 address queue entries that require the same cache index we can make sure the oldest entry a gets a way it is out. So, that we make progress alright. So, if you read the other way if we allocate the longer entry within this particular way, the older entry may actually replace it before the longer entry coms back. So, they keep on requesting each other, it does not make any sense of it. So, because remember that between a little and a retry, there is a window of time where the refill block may actually get replaced in the refill may generate 1 more caches and you and because this may lead to this may keep on happily forever if there are 2 entries that are each other.

The problem is more severe in because their l 1 caches are 2 ways set associative so; that means, to this problem become more ways then becomes much easier. So, any more questions understood alright. So, this is what point about the dependent instructions that take value from a load this essentially talking about the situation, where you have load instruction, they look like this.

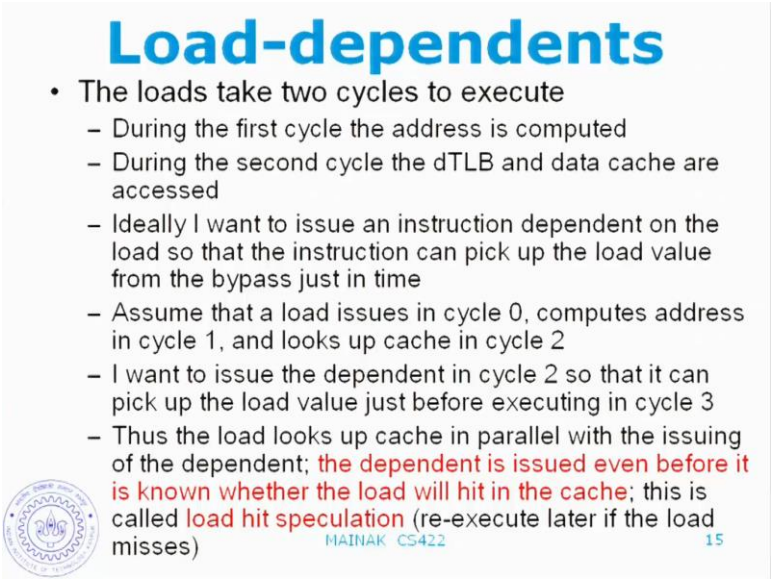
(Refer Slide Time: 10:24)



So, then we have let us say an add instruction that consumes dollar too; we are talking about; so, these are dependent instruction in the load. So, the load state 2 cycle space during the first cycle the address is computed in the second cycle the data key and the data cache are accessed. So, essentially if you look at the pipeline of the load instruction

it will look like this. So; first stage is cache, second stage is decode reading then it issues; then it I will call it address generation and then the memory stage and then, it will commit some time memory. So, this is roughly what; this is roughly what, if your load pipeline will look like.

(Refer Slide Time: 11:29)



So, ideally I want to issue an instruction dependent on the load; so, that the instruction can pick up the load when from the bypass just in time. So, if you look at this instruction add instruction, I will basically want my add instructions execution stage to be positioned here. So, that you can pick up the value from the bypass from the memory ok

So, moving backward what it means is that; I must issue the instruction here because for add instruction, this is essentially the execution stage and then, you know decode reading. So, assume that the load issues in cycle 0. So, this is cycle 0; so, time moves in this direction compute address in address 1, cycle 1. So, this is cycle 1 and looks up cache in cycle 2; so, this is cycle 2. And remember that, this particular stage made itself a cycle depending on whether I hit or miss in the cache; what I am showing here, is the best possible that I hit in the cache. So, I want to issue the dependent in cycle 2 as I have shown here; so, that this can become the in the execution stage. So, that it can pick up the load value just before executing cycle 3; so, cycle 3. So, this commit may not be exactly here in.

So, the load looks up cache in parallel bit issuing of the dependence. So, this is that is happening in the cycle while the load is looking up cache and issuing the dependence. So, I am assuming is that this load is going to hit the cache which is why I will it here otherwise I won not here. So, this is the tedious speculation we talked about this earlier. So, dependent on issue dependent before it is in the cache this is called load hit speculation and essentially what will happen is that, your speculation may go wrong in cases when the load misses the cache.

Now. In fact, I mean what is the big deal anyway we are going to lose a large number of cycles the load has missed the cache. What is important is that, instead of issuing these loads if I knew this load is actually going to miss the cache; instead of issuing this load in the cycle I could have issued that independent instruction, which wouldn't have wasted the cycle essentially I am now, wasting this particular cycle particular issue slot. So, that is where I start losing performance because of this type of this speculation.

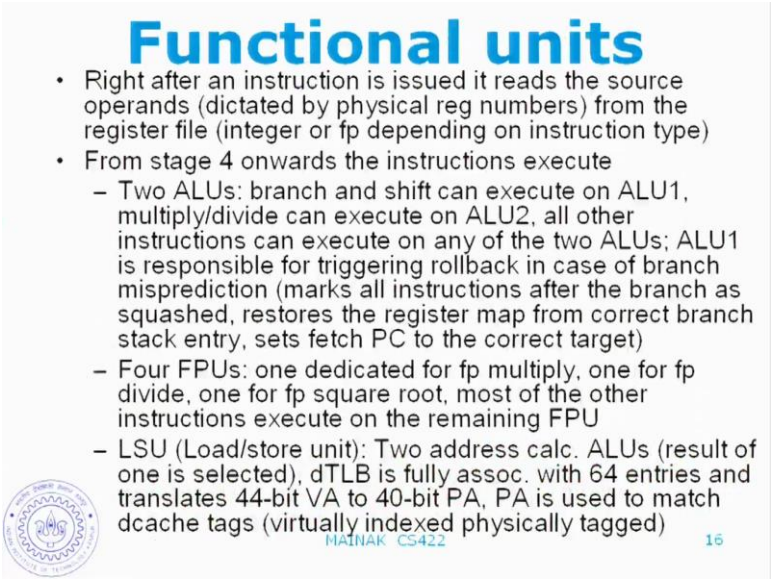
The good news is that this particular speculation is correct most of the time because hit rate is normally, high in caches. So, almost programs you can easily assume that your different would be in access of ninety percent. So, may the likelihood of 0.9 ore than a more than 0.9 would be assured that this will not the cache right. So, alright is it is it clear to everybody this particular speculation

Now, we will we will we will look at pentium 4 very soon probably next week sometime we will find that it has actually multiple issue stages the pentium 4 pipeline is much deeper than this pipeline. So, now, again you can you can imagine what is going to happen here this dependent issues here alright and it will go to several pipe stage before it goes to execution and based on that decision dependent of that particular instructions for example, there could be another instruction here which computes dollar 3 I will assume that this add will actually pick up the value in the right time and assuming that I will issue this instruction also before we can go in anything what is going to happen this this particular instruction. So, here we are talking about wasting just 1 issue slot where you find that this particular this speculation can make you several master cycles you know this alright when you come that point any question on this.

So, does not anything actually to improve this particular activation just always assumes assumes that it would not actually have it actually have a predictor here which would

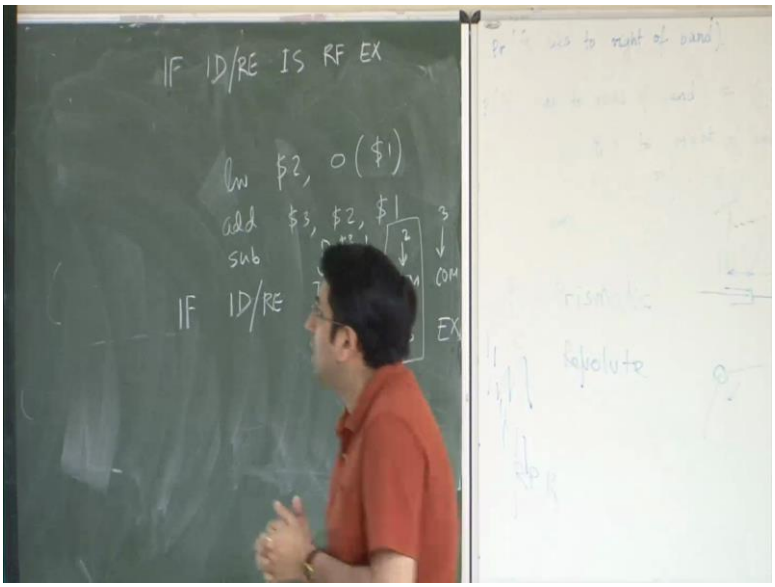
learn the behavior of a load for example, its known at certain instructions miss heavily in the cache they are missing the cache. So, for those dependence you could actually learn to this speculation alright. So, that is all we will look at 1 such predictor we discussed the large processor next alright.

(Refer Slide Time: 16:13)



So, this is the summary of the functional units sorry the the slide is very dense. So, right after instruction is issued it reads the source operands dictated by the physical register number from the register file and from stage 4 onward instruction executes. So, let me see ok. So, yeah that in this example I skipped 1 stage here just for.

(Refer Slide Time: 16:42)

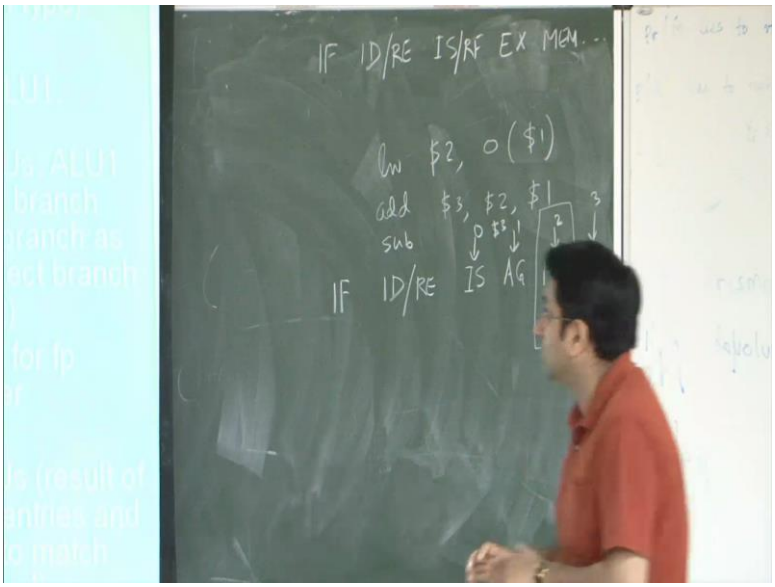


Actually it will have 1 more page in between. So, decode rename the issue with the register fetch there executes. So, this is initially the stage 4 right. So, stage 1 stage 2 oh sorry. So, together. So, this I actually the…

So there 2 a l us branch and shift can execute on l u 1 multiplied why you can execute in a l u 2 1 of the instructions can execute 1 any of the 2 a l us alright. So, a l u 1 is responsible for triggering roll back in case of branch misprediction because because it executes the branch instructions and this prediction recovery here I talked about this marks all instructions after the as squashed restores the register map from correct branch stack entry. So, it is the fetch p c to the correct target.

There are 4 floating point units 1 dedicated for floating point multiplier 1 for fourteen point divide 1 for fourteen point square root most of the other instructions execute and the remaining floating point unit and there is the load store unit has 2 address calculators. So, result of 1 is actually selected. So, I will I will soon explain it data t l bs fully associative with sixty 4 entries translates forty 4 with virtual address to forty bit physical address physical address used to match data cache tags which is virtually indexed physically tagged.

(Refer Slide Time: 18:33)



So, your tag line looks like this issue register fetch then execute or what I call then address generation stage here memory and then they commit sometimes alright. So, any question on this. So, can somebody guess why this is done like this, there are 2 address generation units, but you select only one of them why should I have 2 here.

64 bit processor under what circumstances you it is clear that you are doing something on 2 a l us and the reason why you are doing using both the a l us is because we probably do not know which 1 we produce the correct when we start the computation later point you get to know which 1 should be selected. So, since you are doing some kind of this speculation what could that be… So, this is. So, this a l u these 2 a l us always received load store instructions program counters we had to this particular area for computing well actually the second stage is this say you only computes the comparison problem that is it. computing.

Yeah.

These 2 are used to produce 2 addresses one of which is correct.

Sorry.

No no no no nothing like that first of all do not understand what you mean by that.

I mean load is…

If I have an instruction a load instruction.

Load instruction.

Right.

p c.

No there is no p c here mixing branched over here these are these are loaded actually address you will have register with an offset so

So, I think what he is trying to say is that change your base register.

How can it.

For example, if we have structure.

Right.

That's what yeah

So…

And where is the load instruction.

After this.

After the effects

Effects.

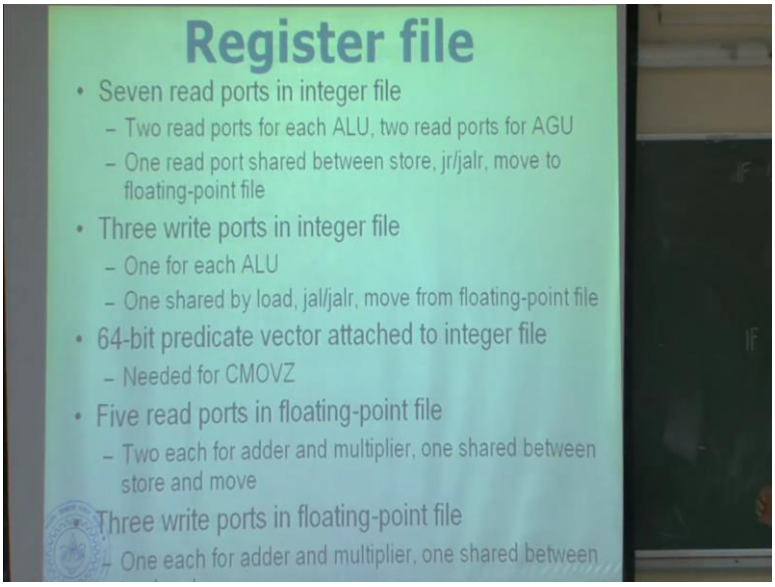
I do that, but that is not what is done here.

So the answer is that miss ardently has 2 different formats of load instructions which have not seen and this format is not decoded until the load reaches this point. So, you sent the the the instruction before they use 1 you will assume 1 format and compute the address other you will assume in other format and compute the address in parallel you decode the path for the load instruction alright and finally, you put a multiplexer which you select 1 of this based on the decoder.

So, let us see how many read ports and write ports are there in register file there are several read ports. So, you got that we have 2 separate register files 1 for integer instruction 1 for integer values 1 for floating point values each of the 64 entries, we discussed that last time

So, what are these seven read ports. So, we got.

(Refer Slide Time: 23:00)



We have 2 integer a l us right we had just discussed in last slide. So, for those you require 2 read ports each because then the 2 operands that makes it for there are 2 read ports for the address generation a l u. So, this one. So, there are 2 a l us 2 address generation a l us each of them may require 1 register within data alright.

Ah. So, that makes six six and 1 read port is shared between store because store will require ah store will require getting 1 register for the value that is store j r and jalr they will they will need to be 1 register to know the branch target because these are integrate branches and move to fourteen point register file.

So, these are these are your m t c instructions move to alright. So, that will require one. So, these 3 actually share a port. So, it leaves that there will be a scheduler will show you hopefully make a fair schedule between this see rights of instructions to give that give the access to that particular.

So, that makes you seven. So, again notice 1 interesting design trade off here he could ask well why did not they have my reports. So, then I should actually get rid of this particular sharing the point is that these 3 types of instructions are very in frequent. So, it does not make any sense to dedicate 1 port for them in a rarely rarely used alright.

So, if I say that well even I have 1 port very rarely I will have a cycle where 2 of these actually are containing from this load most of the time this port will be giving you the instruction 1 instruction containing.

3 write ports 1 for each a l u. So, these 2 a l us are address generation unit does not it does not write to the register file generates an address which is used to look up the t l b at the cache and 1 sheared by load jal and jalr and move from floating point file. So, these instructions would mean to write the written address to the register alright and again the same agreement actually follows that these are more frequent instructions a l u instructions these are not. So, frequent. So, you can share alright.

Now, and again the second point is that rarely you will have a cycles that would have you know 2 of these instructions and they will for the we talked about the predicate predicate bit last time we talked about conditional move instructions right. So, there is a sixty 4 with predicate vector attached to integer file needed for executing conditional move on 0 instructions.

So, this is essentially a bit attach to each integer register. So, of course, I mean the way implement it is there there they separate sixty 4 bit register which is as a vector alright and I need bit corresponds to the ith predicate which actually says that whether register I is 0 or not. So, that is what you have to mention.

So, whenever you produce the value in register file you will also compute this virtual predictor to and say the corresponding there are five read ports in the floating point file. So, let us see what these are 2 each for adder and multiplier. So, required 2 operands to be there and 1 shared between store and move.

So, why do not we need any read port for load instructions floating point loads.

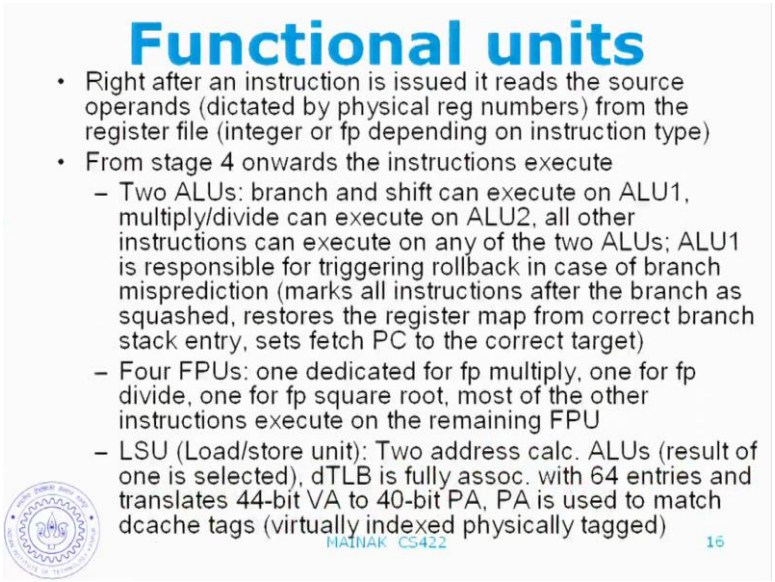
Sorry.

Say again exactly right. So, the only thing that you need for a load instruction is an integer register which is required to compute the address address is an integer value register files, but it will require a right port. So, that is what is mentioned here. So, 3 write ports 1 each for adder and multiplier. So, these 2 and 1 shared between load and move.

So, here this move instructions that we are talking about what are these move instructions move from what particular register that is what we are talking about it oh sorry yeah move from the from the floating point file to the integer file alright.

And these 1 is essentially moved from integer file because floating point the floating point move instructions that that move from 1 floating point register to another floating point register.

(Refer Slide Time: 27:53)



That execute on on on the remaining yeah in the remaining.

So, 1 for multiply 1 for divide 1 for square root and add subtract move and everything else negate etcetera etcetera last. So, floating point stores will require a floating point value

So, there would be.

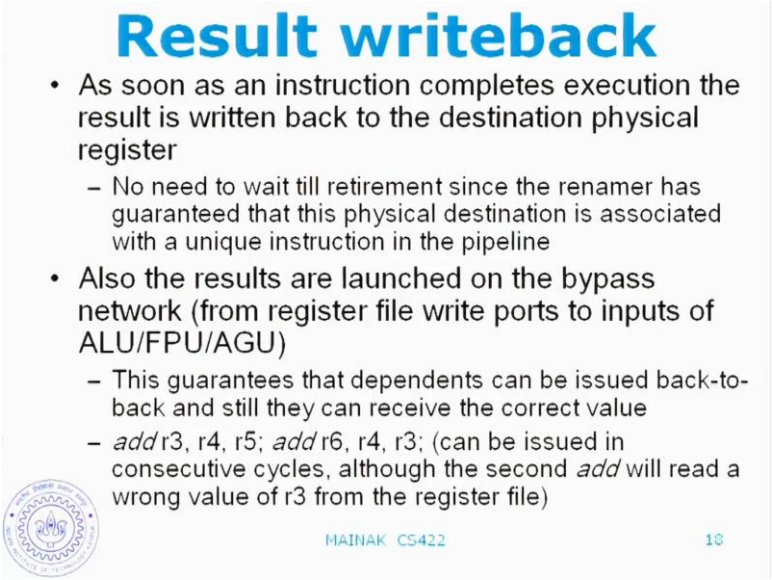
Yes.

.

Yes floating point and integer load would store share the same address.

result right back as soon as the instruction completes execution the result is written back to the [destination] physical register. So, as I just mentioned no need to wait till retirement. So, has guaranteed that this physical description associated with the unique instruction in the pipeline also the results are launched on the bypass network from the register file write ports to this guarantees that the dependents can issued back to back and still they can receive the correct value. So, for example, here right if the load hits this guy will pick it pick up the value from the bypass

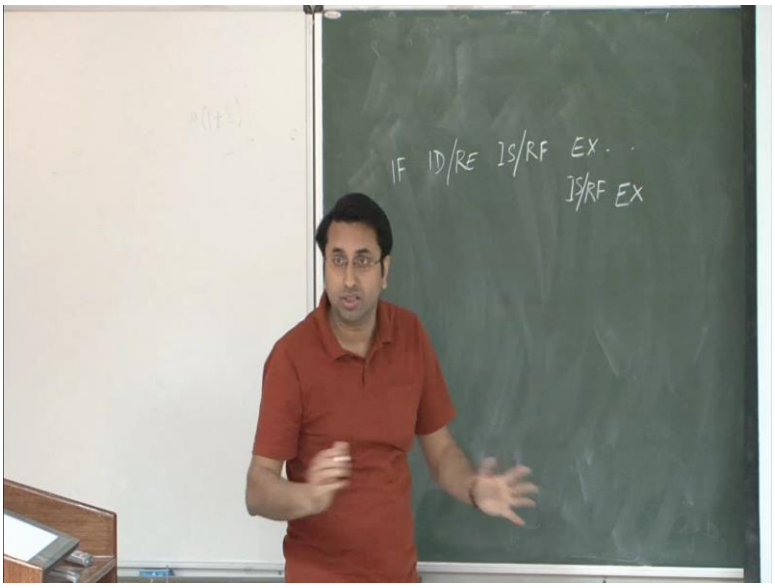
(Refer Slide Time: 29:18)



Only because you will get this. So, as soon as the instruction completes you launch the result in a bypass.

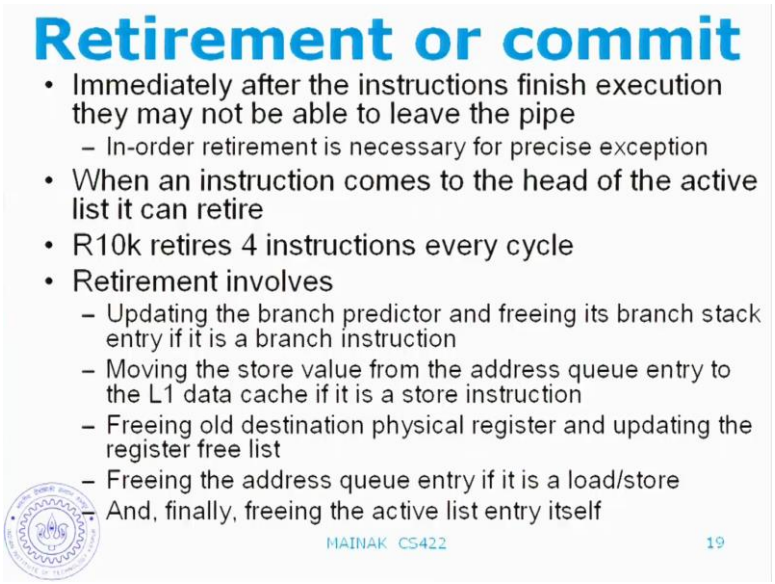
Yeah. So, for example, here alright I can issue these 2 instructions back to back right. So, if you look at the pipeline timing for those instructions.

(Refer Slide Time: 29:43)



So, the first add instruction etcetera and I all do position the second add instructions execute stage here. So, that you can pick it from the bypass. So, I will be issuing it here essentially what it means is that on 2 consecutive cycles I am issuing the 2 value instructions. So, 1 after another without any bubble input right and still the second time we will get the work on time. So, if the second add will actually read read r 3 from the register file here it would be a wrong value which will get a here from the bypass value. So, retirement or commit.

(Refer Slide Time: 30:30)



We this is the last stage of the instruction immediately after the instructions finish finish execution they may not be able to leave the pipe because you have to guarantee not a retirement which is necessary for precise exception when an instruction comes to the head of the active list it can retire because it is a fifo queue. So, when it comes to head you know that retirement.

R ten k retires 4 instructions every cycles. So, we discussed that last time which is why your r o b sorry active list is. So, what does the retirement involve it updates the branch predictor and frees the branch stack entry if it is a branch instruction which moves the store value from the address queue entry to the l 1 data cache if it is a store instruction

So, remember that the store values are not move to the memory for the cache until the instruction retires because you do not really know the instruction retires because there will be a branch before the store instruction which may miss predict alright and the store may not even retire.

So, these are the implication which is that the the in address queue entries must have a filled to both the value of the store alright it frees the old destination physical register and updates the register free list. So, this 1 we discussed last time right frees the address queue entry if it is load store. So, remember that. So, this is very interesting an integral floating point instruction frees the issue queue entry immediately after it issues, but a load store instruction holds the address queue entry until it retires right. So, why is that what is the difference actually.

So, what I am saying is that this add instruction for example, you free the issue frequency right here as soon as it leaves the issue queue, but if it was a load instruction it would not do that it hold the address queue entry until it commits what is the reasons for holding. So, long what.

It has to retry and it has to store the value.

So stores I again understand, but retries.

So, load instructions.

Yeah sure. So, I can say that well those which miss the cache may be entry 1 they complete execution alright.

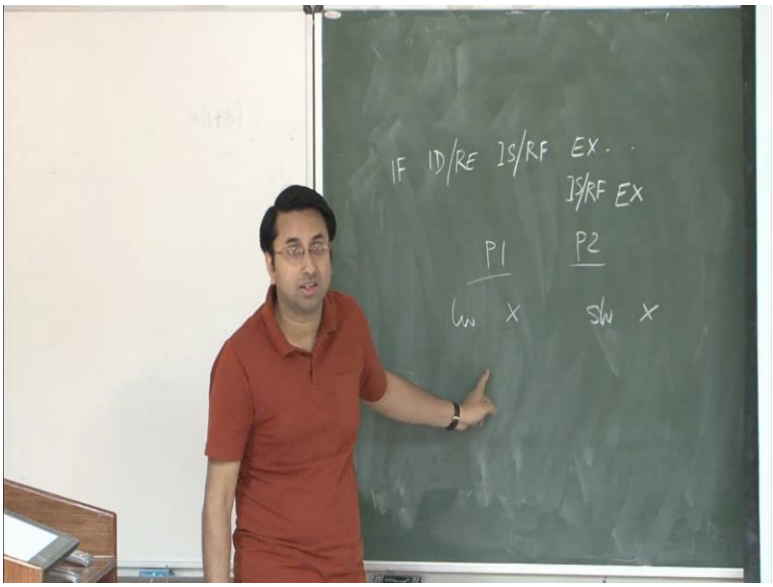
.

What they will issue, but that still earlier than you are delaying this little louder what is the reason.

So you do not know the answer actually which is enough. So, the reason why it is here if that is is a purely multiprocessor reason what we have that is that some other processor. So, let us say your your loading from address x some other processor may modify address x. So, what we have happened is that you have no need the value for address x because the load has completed, but you haven't yet done it because we haven't yet move to the head of the active list.

So, why you are sitting the active list waiting to be retired some other processor modify that. So, we will do that. So, the question why is it to retire this load without any concern because remember this this load must have loaded the previous value the value would not require the value that has been produced by the other processor.

(Refer Slide Time: 34:10)

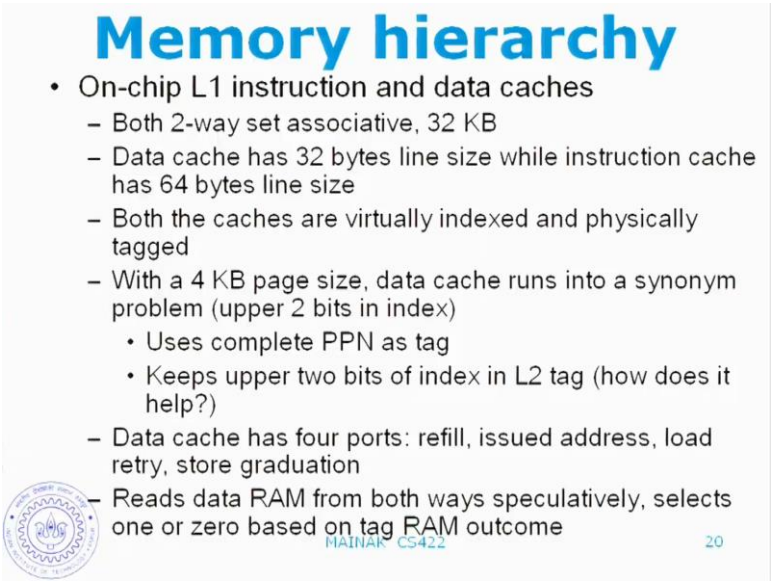


So, essentially I have 2 processors p 1 and p 2 this 1 have a load from address x this 1 have a store to address x this load happen before the store, but it retire after the store. So, in certain situations it is not correct to retire this load you have to retry it even after that alright. So, to cache that you maintain this queue entry. So, the other processor can

actually grow when when the store actually pints out in this processor go even risky actually does not happen in that way, but roughly speaking that is what happens.

And any processors queue holding this address would actually retry the load in all. So, the load must dependence must be executed alright. So, this is the reason why you need to hold address queue entries until retirement alright. And and the reason why you cannot do this search on the active list because active list is not a searchable entry it is not designing the alright and finally, you free the active list into. So, that completes the line of an instruction yes any question.

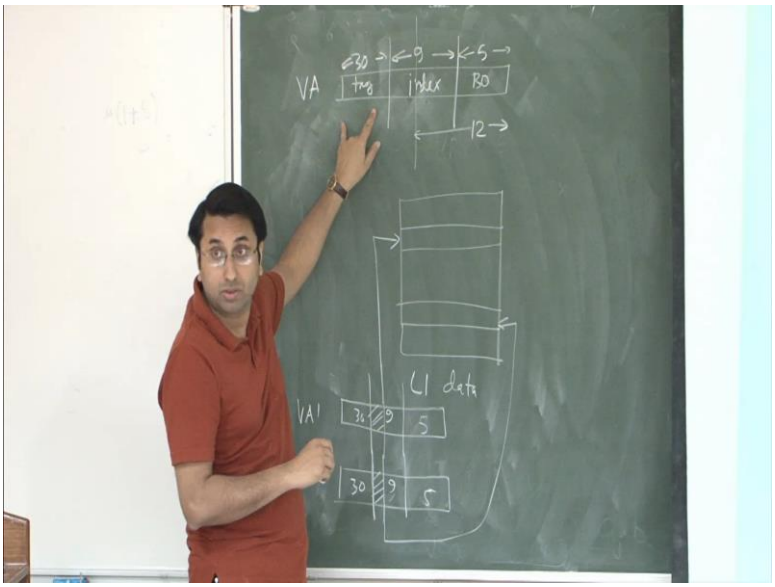
(Refer Slide Time: 35:30)



So, all we are left to with this is the memory hierarchy which we have discussed. So, there are 1 chip l 1 instruction data caches both are 2 way set associative thirty 2 kilo byte in size data cache has thirty 2 byte line size while instruction cache has sixty 4 byte line size why. Right exactly instructions usually have higher special locality that is because usually executes sequentially let us make counter branch which take to somewhere else. So, this is the reason why you have the longer cache block size instructions both the caches are virtually indexed and physically tagged we need that the cache index is derived from the virtual address and the tag is derived from the physical address.

So, this has some implication on how do we actually design the cache this. So, let us try to have the that. So, need a 4 kilo byte date size data cache actually before we discussed with the class problem is... So, just to remind you quickly.

(Refer Slide Time: 36:35)



Imagine that this is my data cache which is virtually indexed meaning that if I this is the virtual address. So, let us try to figure out the the the parts of the address. So, data cache has thirty byte line size. So, block offset is five bits how many bits is the index here is a l 1 data cache. So, you can verify that my index and the remaining things are tagged ok

So, in saying that I can. So, probably 4 bit virtual address yeah forty 4 virtual address translates to the forty bit physical address, so 30 bits. So, page size is 4 kilo bytes. So, my page offset is somewhere here twelve bits ok.

So, now imagine 2 virtual addresses b a 1 and b a 2 belonging to 2 processes and they want to share these these 2 virtual addresses. So, it essentially what it means is that the pair is containing these 2 virtual addresses will actually point to the same physical page in memory alright ok

Now, So, what this means is that. So, remember that the translation where you translate the virtual address to physical address these twelve bits remains unchanged remember because it is a offset within a page alright so; that means, if I take v a 1 can translate it to the physical address what will happen is that this twelve bits will remain unchanged right

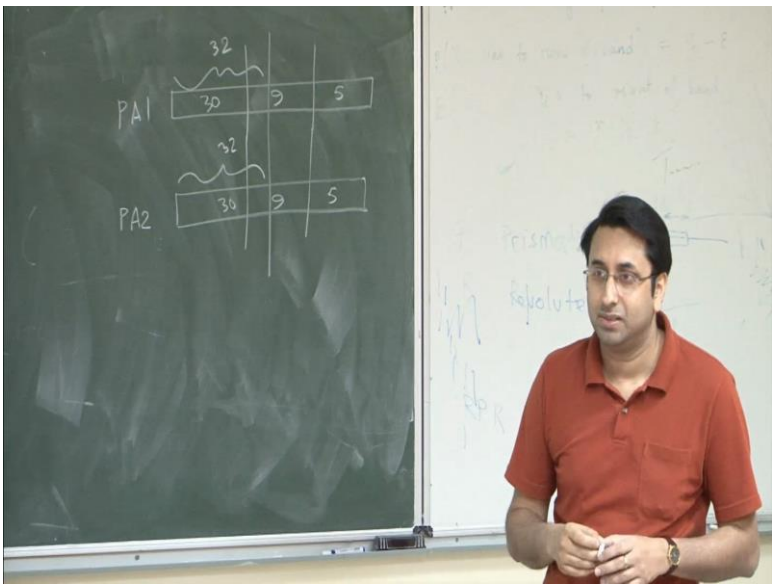
and since these 2 addresses share the same physical page is guaranteed that whenever they refer it to the same cache block same physical cache block these twelve bits will be identical right that is guarantee ok

What is what guarantee this about these 2 bits they may be or may not be right. So, let us continue the situation when they are not identical what will happen is that v a 1 will have 2 some cache index right in v a 2 it rarely have some other caches alright. So, now, the first process modified this particular cache block to v a 1 and gets contact switched off.

The second process reads from this cache 1 it is a wrong value. So, essentially sharing in of loading. So, this is called the synonym problem. So, essentially these are this should be synonyms which was not guaranteed is that clear to everybody. So, so that is that is the problem is mentioned here upper 2 bits for the index that makes the problem actually. So, that is the first problem that we have to worry about.

The second problem is if you think about the tag right the tag of the cache. So, he is saying that it is a physically tagged cache. So, each tag of the cache block comes from the physical address. So, this virtual address gets translated to the physical address and essentially what you do is that you take the upper thirty bits of the physical address for the tag right.

(Refer Slide Time: 40:55)



So,. So, let us see the active need to some problem. So, even into physical addresses p a 1 and p a 2 that of the upper thirty bits are integer. So, these 2 cache blocks are going to have identical tags right now I will have a real problem with with these fourteen bits in the control address are identical because there what will happen is that the same cache index will have 2 cache blocks with exactly identical caches have nobody to distinguish him they map to the same index they have the same tags is that clear to everybody ok

So, I am talking about 2 2 addresses 2 cache blocks that have upper thirty bits identical in the physical address and the lower fourteen bits identical to the virtual address that that happen alright. So, there are 2 problems. So, how to exit. So, it talks about the solution to the second problem it says this this gives us the complete physical page number as a tag he thinks as this part for the tag thirty 2 bits can you solve the problem does not solve the second problem what do you think.

If it does not imagine that there are 2 physical addresses that have identical thirty 2 bits upper thirty 2 bits in a physical address and identical lower twelve bits in the virtual address right that is what we are essentially saying now, but there there are identical addresses right [your] translation load change the lower to the bits right

So, this 1 solves the second problem if we use the complete physical page number as a tag only extra 2 bits for tag to a (( )). So, what is the what is the usual solution to avoid this synonyms does anybody do not know.

karan yeah so.

On on what.

All the synonyms.

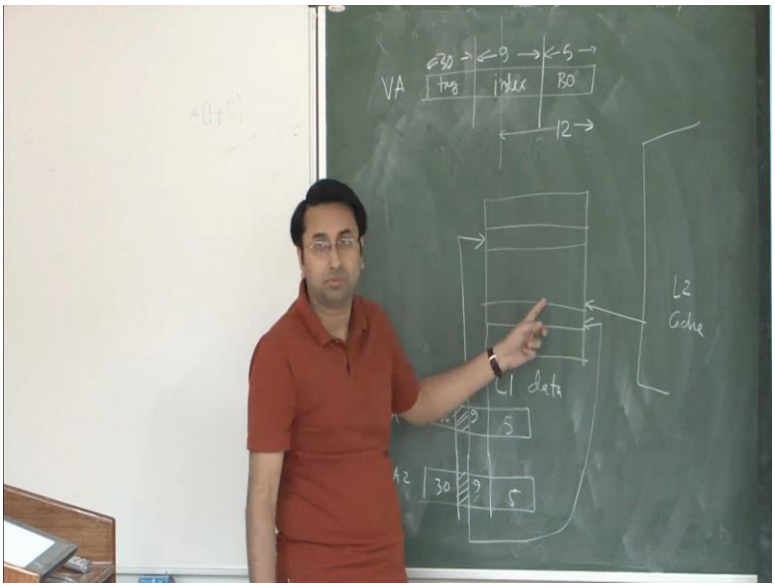
Same.

Same.

No anyway explain the what is the what is the page gallery what does it mean. So, we have this 2 bits here in the virtual address right.

So, you divide your virtual pages into 4 pages based on these 2 bits right and wherever 2 processes request for 2 virtual pages to share they picked up from the same that guarantees that this 2 bits will be identical alright yeah.

(Refer Slide Time: 44:34)

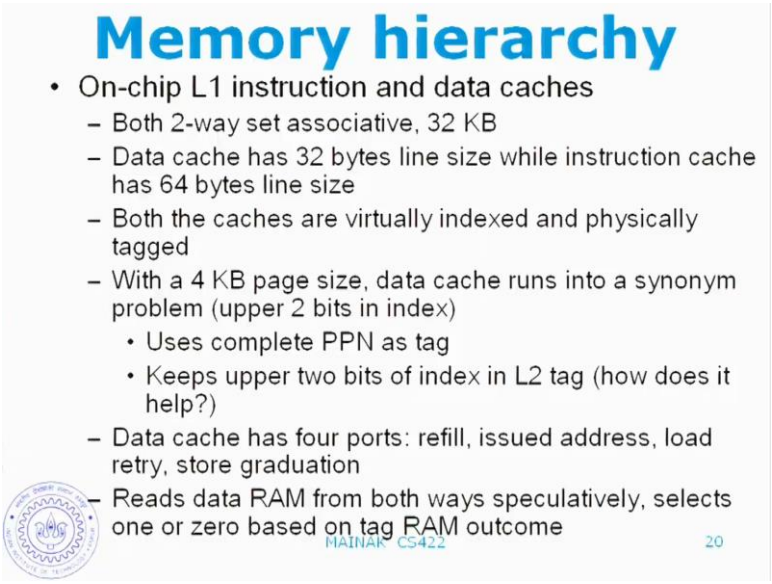


So, that does take care of this problem, but they are in the additional issue that is. So, this is our l 1 cache right and after this we have an l 2 cache and we state that when we have inclusive cache hierarchy, if you replace the block from l 2 cache you must invalid the block in the l 1 cache right to make an inclusion and until [k] has an inclusive hierarchy

So, now imagine a imagine the problem your l 2 cache is physically indexed physically tagged alright. So, when you replace the block from the l 2 cache you can its physical address without any problem from the index in the tag. So, that you send the physical address here to l 1 cache asking that invalidate this block l 1 cache has no way of figuring out what is block is for the physical address because it is indexed by the virtual address.

What if I give you these 2 bits how the virtual address you can actually derive the virtual index because these twelve bits are going to be same in the physical address as the virtual address you take these 2 bits and prepare the index of the index in l 1 cache and then you can invalid the block.

(Refer Slide Time: 45:52)



So, this is the reason why it maintains these 2 bits of index in l 2 tag. So, that for inclusion these declarations can be performed to multiprocessors there will be many other cache to. So, in addition to this you have to do page colour its always a little problem this does not allow for.

However this takes are of designing an inclusive hierarchy where your lowest level cache or the inner most level caching actually is it clear to everybody. Yes right it is to mix up. So, data cache has 4 ports what are these 4 ports 1 is used by reference 1 is used by the issued address 1 is used by the load retry and the 1 is used for store graduation. So, remember that when the store commits it will to the cache it require more the retry unit has a dedicated port and these are the traditional addresses that come to the cache from the address generation unit. So, I will erase that and refills are coming from the next level of the hierarchy ok

Ah it reads the data ram from both the ways speculatively. So, what it does is that is what talking about the l 1 cache it has 2 ways right has it says. So, it actually reads the data from both the miss in parallel and selects 1 or 0 based on the tag ram outcome what was the other option the other option would be that we first do the tag ram look up compare figure out if you actually have a hit and then look up the data ram based on the hit or miss taken.

So, you say time by using this in parallel. So, look up the data ram when the tag ram is parallel both come out together check on the tag ram and then figure out whether to pick 1 or not to pick anything from the from the 2 outcomes in data a

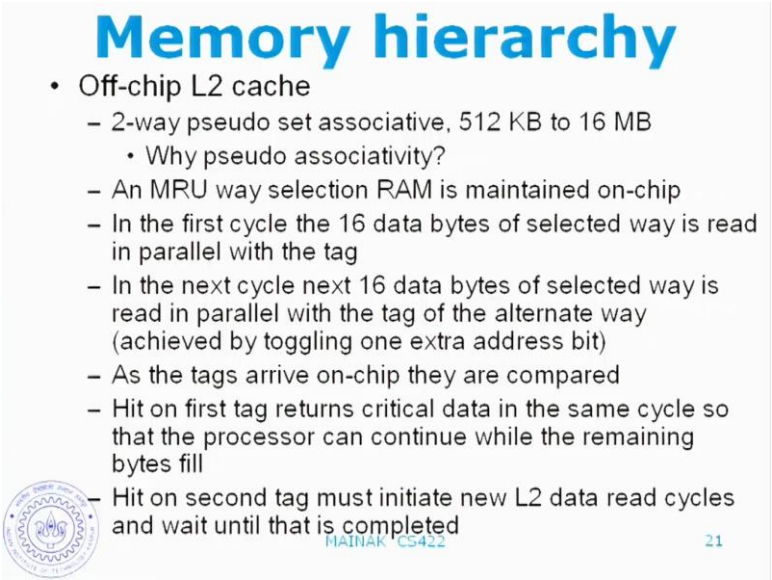
So, we save time what you sacrifice yeah.

Power.

You consume more power exactly I definitely. So, that. So, power consumption would be actually more than double right because in many cases I would not have access any of the data and in a way hit I will be accessing only 1 here in all cases I will be accessing two.

So, this is often called parallel tag data look up and traditionally used in l 1 caches to save time alright. So, that is the main purpose and since l 1 cache is normally have small associative within the energy consumption is still within limit alright, but you are really targeting a low power design they will probably wouldn't be doing this with the sacrificing downs of latency that will be saving the transfer alright. I think we saw the last slide which I will cover next time or may be yeah ill I will probably decorate them, because let us try let me see if I can finish it off.

(Refer Slide Time: 49:30)



So, it is a 2 way pseudo set associative cache configuring to the 2 time right for can do 16 m b. So, what is 2 way pseudo set associative pseudo set associative does anybody remember. You look up the look up 1 of the ways first.

Yes.

Yeah.

Right exactly. So, there are. So, you serialize the look ups to the 2 ways right and. So, there is a fast hit and a slow hit right. So, why was it done normally why it is done pseudo set associative.

Sorry

Right.

Yeah. So, what you think.

Time.

Time exactly.

So, the the past hit latency is seen as the direct map cache in this alright. So, that is what you say did not do tag can you guess why it has an off chip l 2 cache that is my hit. So, imagine what you have to do if you had to look up all the tags compared to looking up 1 tag at a time 1 to 1 to.

So, I think in…

Yes, but they did not have for the reason of having direct map cache it is in fast (( )) there is a separate reason for that why it did not have 2 way pseudo set associative cache [want] to share with the 2 way pseudo set associative cache they do gain latency if it hits in the in the first way, but that was not the primary reason and the different campaigning reason that has to do with the offchip part of the l 2 cache.

What is offchip

So, a main processor does not have the l 2 caches inside it its outside the. So, there is a separate package which has the l 2 cache. So, you read the tag bring it here.

Yes.

Right.

Yeah. So, why is that important.

32 bits tag right sorry this is the l 2 cache. So, it will be different. So, yeah those are save it saves what.

Why is that important?

More way a side.

More way as outside the chip of the why is that let me think about the reason why we multiplex trash and cache in memory what was the reason you need pins to make the data right because it is the object. So, we need more pins on the chip to make the direct 2 tags in parallel that was the same reason why you multiplex the trash and cache addresses because the is normally exactly same as here it is just a limitation of bits ardently cannot afford. So, many pins to have 2 tags in parallel again alright.

So, there is an m I r way selection ram that is maintained on chip. So, its its kind of a predictor you can say what it maintains is that for each intel cache set which says which 1 is the m I r way and they predict that this is the way where you are lightly to hit this is time because it was the m I r way last time alright.

Ah. So, in the first cycle the 16 data bytes of selected way is right in parallel read the tag the next title next 16 data bytes of the selected bit is read in parallel with the tag of the ordinary bit alright actually by talking 1 extra address bit. So, he has the tags that I want it they are compared return first tag returns predictor data in the same title. So, in a thirty 2 byte cache line cycle alright well see that actually because l 1 cache is thirty 2 byte.

So, whenever l 1 cache request requests data to the l 2 cache it always in chunk of 128 data since here and however, many bits you need for the tag that is determined by a capacity of the cache.