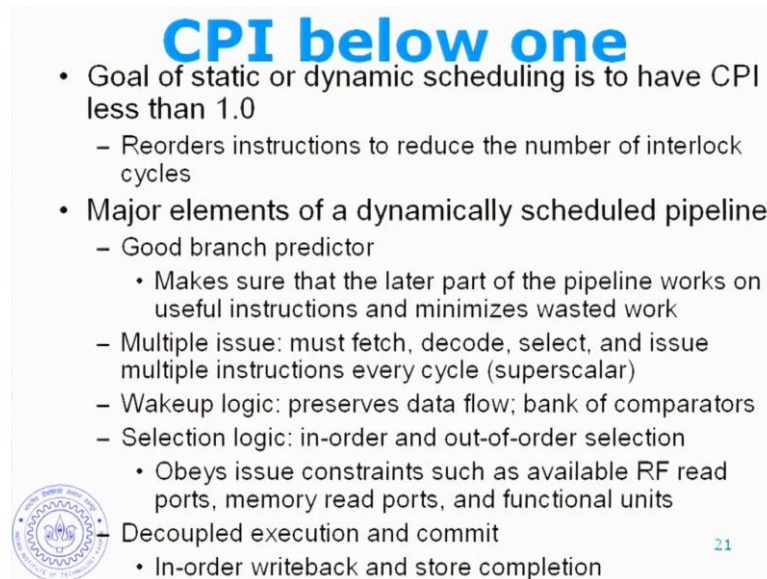



**Lecture - 23**  
**Dynamic Scheduling, Speculative Execution**

(Refer Slide Time: 00:22)



**CPI below one**

- Goal of static or dynamic scheduling is to have CPI less than 1.0
  - Reorders instructions to reduce the number of interlock cycles
- Major elements of a dynamically scheduled pipeline
  - Good branch predictor
    - Makes sure that the later part of the pipeline works on useful instructions and minimizes wasted work
  - Multiple issue: must fetch, decode, select, and issue multiple instructions every cycle (superscalar)
  - Wakeup logic: preserves data flow; bank of comparators
  - Selection logic: in-order and out-of-order selection
    - Obeys issue constraints such as available RF read ports, memory read ports, and functional units
  - Decoupled execution and commit
    - In-order writeback and store completion

 21

We are discussing goal of static or dynamic scheduling is to have CPI less than; 1 and the primary way to achieve that is to reorder instructions; to reduce the number of interlock cycles. So, we have discussed main the technique of dynamic scheduling. So, second is the summary of the major elements of dynamic rescheduled pipeline, you need a good branch predictor to make sure that the later part of the pipeline works on useful instructions and minimizes wasted work otherwise, you do not have a good predictor you will be fetching wrong instructions in most of the time and you will be doing waste wood work which will have to be thrown away later.

Second element is multiple issue you must fetch decode select and issue multiple instructions every cycle and that is what is called a superscalar processor. Third element is wake up logic. So, this logic is it is of responsible for waking up instruction that are waiting for values. So, it preserves data flow and the implementation of the wakeup logic it essentially a bank of comparators. Next 1 is selection logic which; essentially selects subsets of instruction that can execute and there could be 2 possibilities of selection you could select in order or you could select out of order itself. So, we will look at both of

this very soon. In fact, we have already looked at out of order selection the only constraint the extra constraint for out of order selection is using a priority mechanism or often called a time breaker for example, if you have decided that you design hardware such that in a cycle maximum you can issue 10 instructions.

Now, it could be possible that we have 20 instructions waiting, now out of order selection of course, does not impose any constraint on which instructions you can pick you can pick any 10 of this. So, then has to be time breaking point whereas, in-order already gives you a restriction that you have to go in order you cannot violate the order which means; many queue have to be different instructions and different places which are ready you may be only into select high order.

So, we will soon look at this 1 actually it is a deconstruction of out of order selection much simpler lower performance. So, essentially the idea here is that the selection logic obeys the issue constraints such as available register file read ports memory read ports and functional units. So, I its example just I mentioned about this 10 instructions maximum that comes from these constraints usually because 10 instructions would possibly mean that your register file ports has 20 read ports memory read ports may be something depending on how many of these 10 instructions can be memory operations and so on and so, forth. And finally, you have decoupled execution and commit.

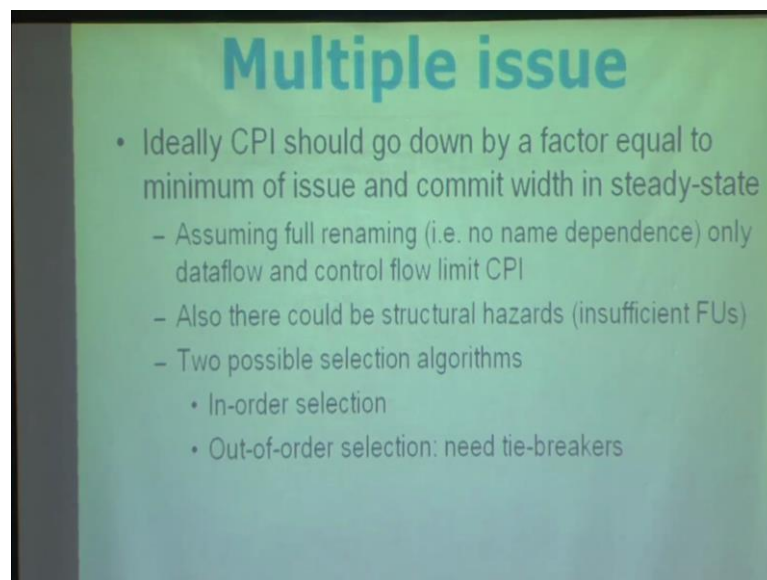
So, this is very different from what we have discussed, for single issue in order pipelines; that is you execute an instruction. And then after some time the result will be return back to the register file or memory. So, essentially this particular logic preserves in order write back and store completion. So, these are the major elements of dynamic scheduling that we have discussed which tries to achieve this goal that is CPI as low as possible all right. So, at each stage if you if you if you observed carefully, you will introduce certain constraints for implementation purpose.

For example, here you are saying that I do not have an oracle predictor; I know realistic predictor which will make mistakes so, many times that will lower that, will actually increases CPI by some amount  $\alpha$ . Here you are saying that I cannot really fetch decode and select an issue and infinite unbounded number of instructions and its some limit there that will further reduce increase your CPI  $\alpha$ .

Here you are saying that I can wakeup arbitrary of some instructions, but I cannot read select on all issue all right. So, there also imposing certain latency and here of course, you are saying that I cannot really, commit in modern number of instructions there also you have some limits. So, all these taken together along, with your data flows will give you some CPI which is hopefully better than; whatever we have discussed and whatever you are doing in your home all right. Any question on this basic scheme of dynamic scheduling.

So, this actually with this description I have searched away all the implementation devious how exactly you go and implement all this things it is a basically normally. So, this 1 also we have discussed. So, just to recap a little bit so, when you are issuing multiple instructions in the cycle ideally the CPI should go down by a fraction equal to minimum of issue and commit in the steady state all right. So, how many instructions I can execute in the cycle and how many instructions I can commit in the cycle minimum of these 2 decide by how much my CPI should go down compared to a single issue in order pipeline.

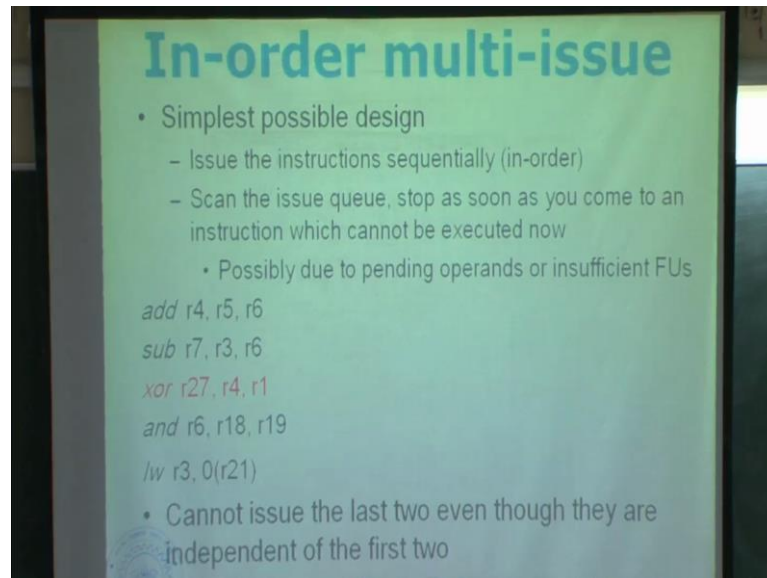
(Refer Slide Time: 05:53)



So, assuming full renaming that is suppose you have no name dependence, you have no false dependence all right; you have only true data flow dependence and the control flow. So, data flow and control flow will only limit the CPI all right; also there could be structural hazards that is insufficient functional units are the same. And there are 2

possible selection algorithms in order selection and out of order selection this 1, we have already discussed ok. So, let us take a look at this 1 what really is in order selection.

(Refer Slide Time: 06:24)



So, this is a simplest possible design of our simplest processor by the way the term simplest scanner means; that your executing multiple instructions in cycle that is what it really means. So, in this case what you do is you issue the instructions sequentially in order all right. So, you scan the issue queue stop as soon as you come to instruction which cannot be executed. Now it which could be possibly due to pending operands or insufficient function units.

So, here is an example let us see what is happening here. So, these 2 are independent instructions and let us assume that their operands are now ready r5, r6, and r3 r18 all right. This 1 depends on add all right. So, it is clear that this 1 cannot execute together with this 1 that is pretty of this all right these are independent and these 2 are also independent all right. So, at out of order scheduler would actually, pick these 4 instructions except the red 1 and would actually execute them in a cycle and in order scheduler would actually issue only these 2 in this cycle, because it cannot violate the oracle execute the would all right.

In the next cycle it would probably pickup these 3 and issue them all right; into the different step. So, you cannot issue the last 2 even though they are independent of the first 2. So, that is what you do certain do in order issue and what you gain by doing this

Single design.

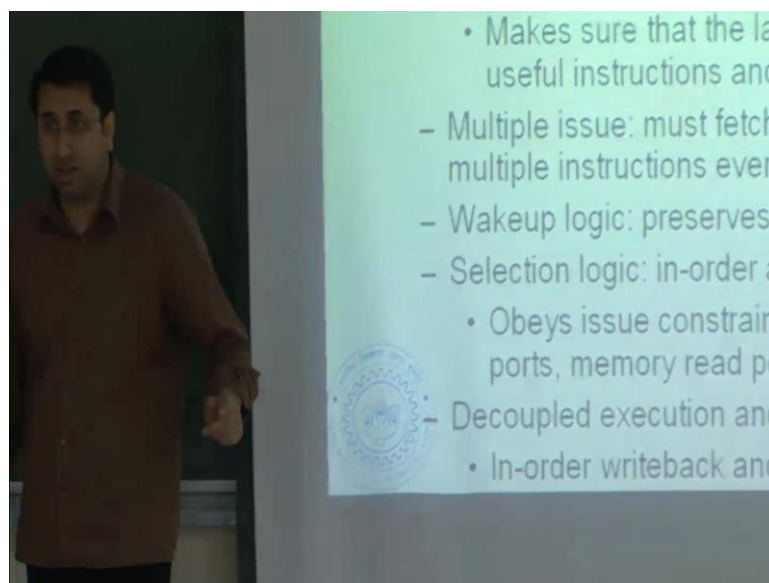
Single design in what terms. So, from this list can you tell me what gets simplified

Wakeup logic and...

Wakeup logic gets simplified why you still have to wake up the waiting instruction.

That is wakeup logic is not required.

(Refer Slide Time: 08:17)



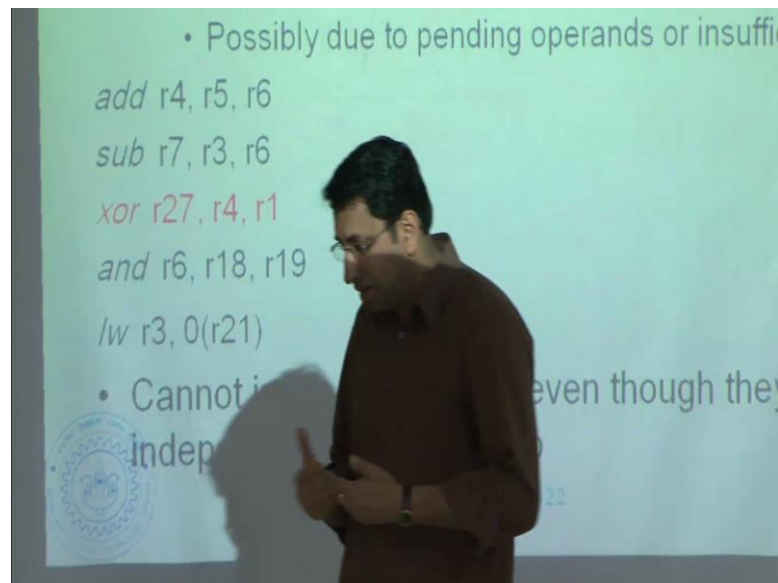
So, basically when the selection logic it is the logic which instruction will stop at the first waiting 1. So, it is a simpler logic that that is not depend on.

What is do I need this 1 in order commit or written.

Sir in selection logic you have already chosen like. So, do not.

Can I verify your registers immediately. So, here we were actually putting the results back to the queue entry and later moving on can I quit here.

(Refer Slide Time: 09:06)



Can I move r4 and r7 direct to registers immediately is that possible which means; bypasses are available.

You saw a bypasses are available whether it goes to or queue slot.

Yes you can.

I can anybody else. So, intuitively it seems that in order issue should reading of this particular logic in order commit later is that right.

Sir if the by pipeline stages are not each instructions are not passing throughout the pipeline stages.

Now let us assume them. Let us assume them there is single pipeline that is it every instructions goes to that pipe yes your right, otherwise; there would be problem yeah, because I essentially talking about now for a completion problem.

Yeah.

No let us not get into that so...

Sir, different execute.

So, essentially I am saying that in that case whatever, we have seen in the multi-cycle execution right. So, how many instructions they complete early and they late no. Let us not assume that we have a single let us say otherwise simple pipe stage pipe or whatever you know every instructions goes to same pipe.

Pipe structure pipe stage same for there are 2 stages cycle in the same register.

Ok.

And if the order of them gets changed

Ok .

The order of fetch...

All right. So, essentially you are saying that in the register file light code I should have some mechanism to maintain all in which; there are is that understandable for example, these 2 these instructions could be writing go to r4, I must reserve this order when they write to register file which make sense anything else any other issue in this.

We have rank.

Yes, but remember that instruction execute in order which means if there is a branch before it has already executed and involve the protocol by now.

You cannot put anything in the...

No, you can say that I probably cannot execute anything together with the branch that is only what that I say yes true because anything that goes in parallel of the branch we have possibility of producing the wrong value and register value yes that is true any other issue.

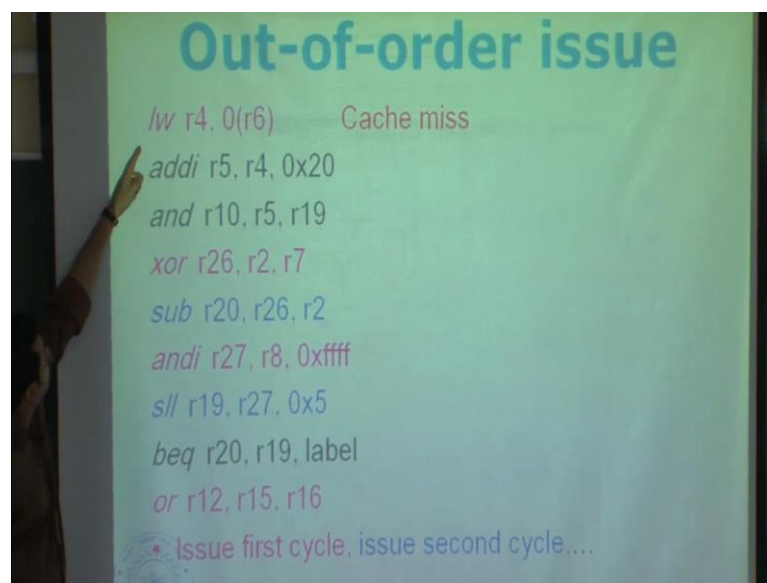
Independence of value cannot be figure out at time out.

Sure yes. So, we follow the same protocol as it make sense yeah; there here the things are much similar, because they will go in order you know that the execution in order.

Sir, but the next there is store and the next statement is the row configured out there that is what is the independent or dependent.

So, you are saying that if I have a store in a load I cannot execute them parallel in the cycle that is what you are saying yes agree right yeah. So, many we still have certain issue constraints as that coming out here and we also have to maintain register write code order to detect write after write hazards any problem, with write after read hazards and that create an issue write after read dependencies all right. So, the point here is that its indeed similar it pretty much reads you of the in order extra pipe stage or in order commit provide to maintain certain points that that you here you guys have raised ok. And here is an example of an out of order issue.

(Refer Slide Time: 13:03)



We have already discussed this also. So, colors are hopefully clear right we have 3 colors here. So, depends on your test and how you recede, I will call it a pink this particular color. So, the pink instructions can go with the first cycle, because they are all independent and you can figure out why the rest cannot go this 1 depend on the load this 1 depends on the add this is independent this 1 depends on the x or this is also independent this 1 depends on the add and this 1 depends on the shift this 1 is independent all right ok.

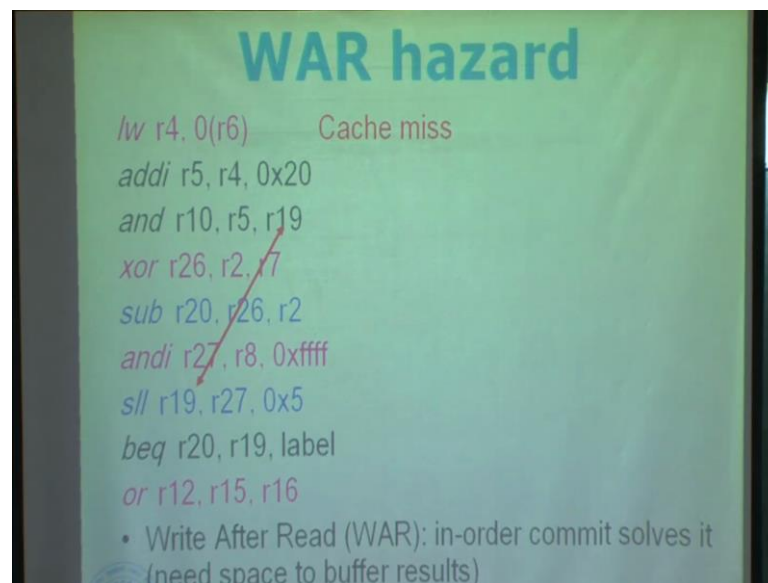
So, the pink instructions are independent they can go in a first cycle all right. So, that is that is the basic idea of out of order issue I do not clear about the order right; we have discussed it already the blue ones can go in the second cycle. Because, the black 1 still cannot go because, I am send read a cache miss which is essentially, meaning that it is



going to take long time to complete all right. So, these 3 instructions are arithmetic instructions actually logic instructions; they will complete in the cycle probably. So, there dependence can issue, in the next cycle like this 1 can issue in the next cycle this 1 can issue in the next cycle.

So, that is the second issue cycle and finally, when the load completes let us see ok. In the third cycle these this can go we are in problem, because both these operands will get ready from these 2 all right. It needs 20 and 90 they come from here and here and then finally, when the load completes this will go and in the next cycle this will go all right. So, that is the out of order issue and then my commit logic will guaranty that they update registers in the order specified in this program, in this particular sequential order that is the essence of essence of out of order right clear any question on this ok.

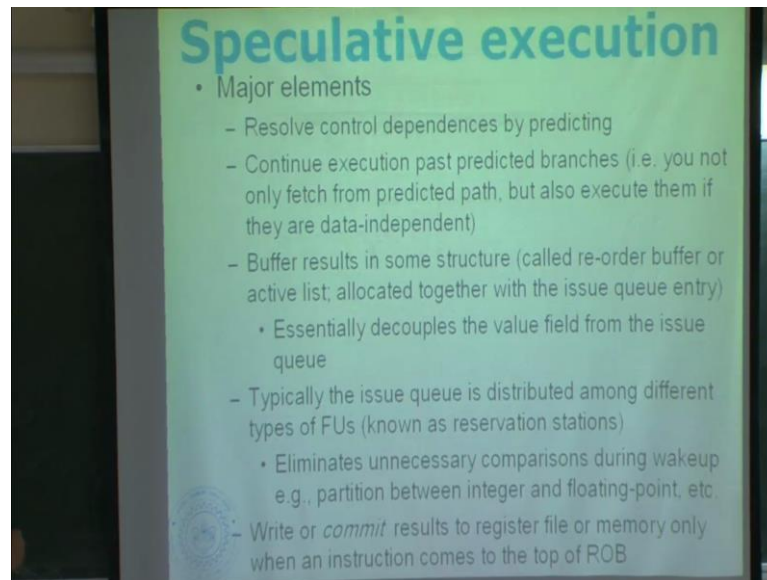
(Refer Slide Time: 15:21)



So, and we also discussed the presence of this write after read problem; which we have to worry about as you can see here right. So, we said that the shift can issue in the second cycle, but you have to be sure that it does not overwrite r19; which is been controlled by the end instruction much later and in order commit solves a problem, because we will make sure that 19 will not be return until all these instructions get a chance to write all right. So, in order commit will make sure that first r4 is written then r5 then r10 then r20 then r20 7 then r19 all right.

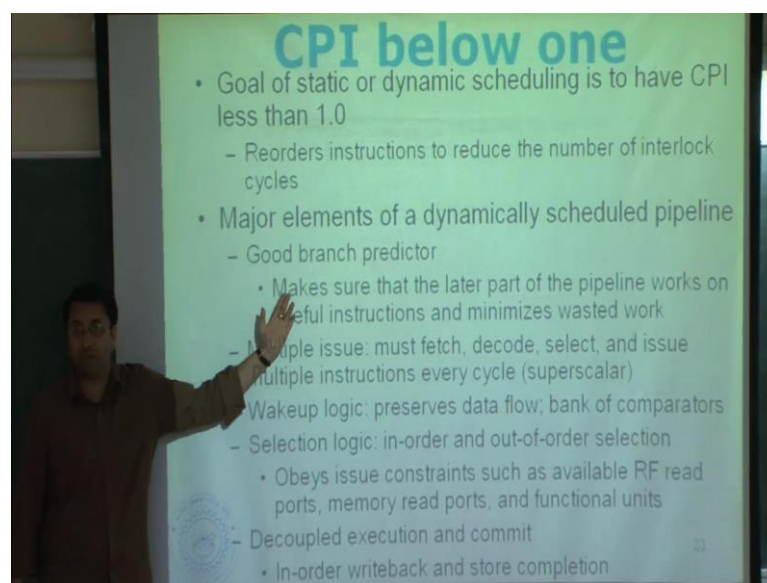
So, there is there is no name dependence problem here it will be it will be commit order, but the down side is that you need space to buffer results essentially you have to make sure that r19 is what made visible to the to the other instructions all right ok.

(Refer Slide Time: 16:23)



So, often people call dynamics scheduling also form a speculative execution, because the point here is that your combining. So, you can you can see it as. So, if you go back this list.

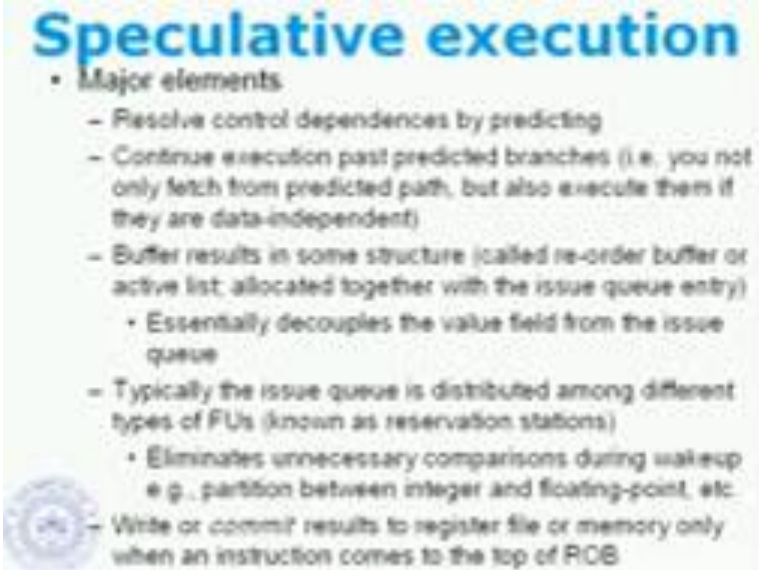
(Refer Slide Time: 16:30)



So, often this part except the branch predictor can be used independent of what is happening in fetch whether, you have predictor or not you can independently use this part of the logic all right ok. So, essentially you can say that I will keep on fetching in order no branch predictor on a branch whenever, you observe a branch and it is all completed. I can still do this actually even in order branch predictor nothing stop you from using all these logic.

So, that is the pure dynamic scheduling there is a no speculative execution as soon as you introduce a branch predictor your waiting to the speculation in the pipeline your now saying that there are certain instructions; which are speculatively fetched which may be actually on the wrong path ok. So, eventually the branch predictors predict prediction will requested and I will get to know whether the instructions are correct or not. So, these 2 taken together is often called speculative execution. So, that just a different name of what we have already discussed.

(Refer Slide Time: 17:35)



## Speculative execution

- Major elements
  - Resolve control dependences by predicting
  - Continue execution past predicted branches (i.e. you not only fetch from predicted path, but also execute them if they are data-independent)
  - Buffer results in some structure (called re-order buffer or active list; allocated together with the issue queue entry)
    - Essentially decouples the value field from the issue queue
  - Typically the issue queue is distributed among different types of FUs (known as reservation stations)
    - Eliminates unnecessary comparisons during wakeup e.g., partition between integer and floating-point, etc.
  - Write or commit results to register file or memory only when an instruction comes to the top of ROB

So, resolve control dependences by predicting continue execution past predicted that is you not only fetch from predicted path, but also execute them if they are data independent. So, we have already discussed this also buffer results in some structure is called a reorder buffer or active list depending on the processor architecture. So, active list is that a use y bit architectures otherwise, in most other processor you will see that all reorder buffer it essentially a separate structure which maintains; the results of the

instructions and the order of the instructions all right allocated together with the issue queue entry.

So, in the issue queue entry we put everything, because 1 that you saw here I am just decoupling a few fields from issue queue entry putting, in the similar structure already out of order all right. So, essentially decouples the value field from the issue queue that is it typically the issue queue is distributed among different types of function units. So, these are known as a reservation station that is what we have discussed also eliminates unnecessary comparisons during wakeup.

For example, it would now have a partition between, integer and floating point we should use integer results; would be broadcast only on the integer queue the floating point results would be broadcast only floating point queue and so, on all right. So, you can say some of the comparisons, because 1 to 1 comparison is actually not needed from all this all right and finally, you write a commit results register file or memory only when an instruction comes to the top of the ROB So, ROB is a is an acronym used for reorder buffer.

So, reorder buffer is essentially a free code queue all right; it maintains the instructions and it has the value field which stores the results produced by an instruction and it will be drained in order. So, that is essentially that is what we have discussed it is just that I am I am just introducing a new structure called ROB to decouple the value from issue queue and distributing the issue queue across function units; which how it have an issue queue a read point n it have an issue queue number of store you need the cache will have an issue queue will have an issue queue and. So, on any question this is clear ok.

(Refer Slide Time: 19:53)

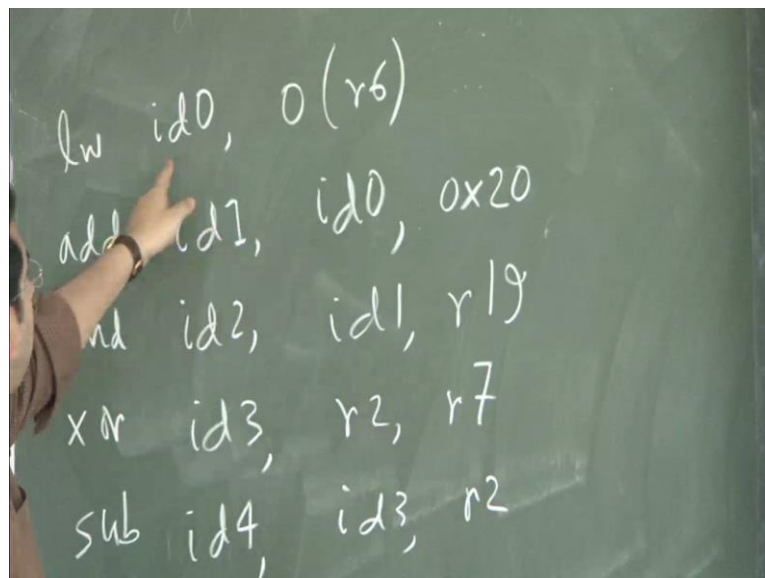
## Register renaming

- Recall the problems with our naïve register renaming through issue queue slots (or ROB slots)
- Registers visible to the compiler
  - Logical or architectural registers
  - 32 in number for MIPS, 8 for x86, and is fixed by the ISA
- Physical registers inside the processor
  - Much larger in number, not visible to compiler
- The destination logical register of every instruction is assigned a new physical register number
- The dependencies are tracked based on the namespace of physical registers

MIPS R10000 has 32 logical and 64 physical regs  
Intel Pentium 4 has 8 logical and 128 physical regs

So, now just 1 small element has left, in this architecture something called register renaming we have already seen this essentially, what we were doing was that we were renaming our registers into issue queue slots right. So, whenever instruction produces the register just go back to this example. So, here essentially r4 will be renamed to slot id0 all right r5 will be renamed to slot id1 r10 will be renamed to slot id2 all right. So, whenever slot id0 is done which are the dependence on slot id0 it will be open r4 is.

(Refer Slide Time: 20:40)



So, essentially what I am doing is I could actually, now rewrite the instruction as followed. I would say load word id0 comma 0 r6 addi id1 comma id0 comma 0 x 20 so, on and so, forth I can actually, translate the instructions to different name space all right. Similarly I would say and id2 whenever, I get a new instruction and a new target I assign to new slot right; id1 r19 x or id3 r2 r7 sub id4 id3 r2 etcetera all right is this clear this renaming mechanism on translating the registers from 1 namespace to another.

The only thing observe here is that; it is a mix of 2 namespaces some of the names are coming from the register architecture register some of the registers coming from my queue slot id. And there would be a table we should maintain this mapping for example, whenever you do this it would say r4 is now mapped to id0. So, the next instruction will comes in you look up the table that row that found I should replace r4 by id0. So, that is all we will do the renaming.

So, that is what we have done in now. And here since we have 2 namespaces we have problem that is we need to figure out whenever instruction issues where from it should read the bank all right. So, there are 2 possible places it can read the value problem. So, for example, this instruction will read 1 value from the queue slot id1 the other value from register 19, in the register file and because of these 2 different sources of values we get a problem that we have to a wakeup cycle at the time of commit actually.

So, make sure to let this instruction know that for example, suppose this instruction some of these instruction issue got delayed for some reason, whatever; it may be and by the time it issues this instruction is already return back ok. So, when it tries back it has to tell this instruction and oh now you should not take the value from id0, you should actually take the value from r4 right that has to be not change back. So, this becomes a huge problem because of as a primary reason for this problem is that we have 2 namespaces doing stuff here we would not want this case that is it we want to all values to come from.

So, that is exactly what today clusters do is and that is called register renaming. So, let us try to define the few terms is the problem clear any question, before I move on. So, registers visible to the compiler at logical or architecture registers for example, 32 in number for MIPS the 1 that you are using for the homework, there are 8 for x86. And this is fixed by the instruction set architecture how much how many registers should

made to be made visible to the compiler or the program programmer ok. And there are physical registers inside the processor.

Usually today this is much larger in number and not visible to compiler; however, the requirement is that your physical register file size should be at least as large as the logical register file size all right. So, that in the minimum case we should be able to have a 1 to 1 mapping for 1 logical register 1 physical register ar. And that is what the processor your using for your homework actually does which has 32 logical registers; which has 32 physical registers and there is always a fixed 1 to 1 mapping for the logical register and physical register ok.

But today's concerns actually have much larger number of physical registers inside the processor which are not visible to the compiler and if you have that essentially; then you have to establish the mapping and the mapping actually change all the time which, logical register maps to which physical register actually it is some change and that is exactly algorithm that that defines register alone.

The destination logical register of every instruction is assigned a new physical register number just like this except of except assigning a queue slot id I will pick up the free physical register and give it to the extraction target the dependencies are tracked based on the namespace of physical registers only there is no queue slot I d name spacing all right. So, MIPS r10 k has 32 logical and 64 physical registers Intel Pentium 4 has 8 logical and 128 physical registers today physical processor have many more physical registers. So, we will see exactly if what it means to have more physical registers why you want that and so on.



(Refer Slide Time: 26:00)

## Register renaming

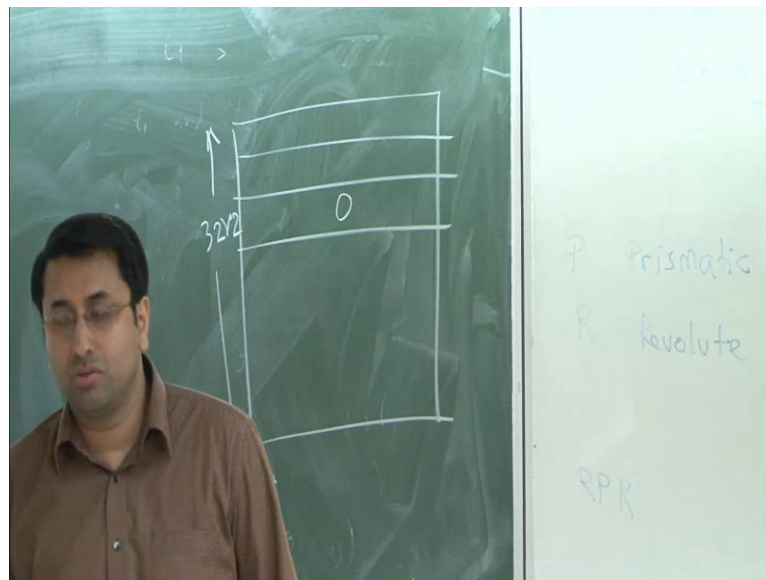
- Assume 64 physical regs and already renamed registers:  
r6=p54, r19=p38, r2=p0, r7=p20, r15=p3, r16=p23.

<i>lw</i> r4, 0(r6)	<i>lw</i> p15, 0(p54) [r4 renamed to p15]
<i>addi</i> r5, r4, 0x20	<i>addi</i> p40, p15, 0x20 [r5 renamed to p40]
<i>and</i> r10, r5, r19	<i>and</i> p39, p40, p38 [r10 renamed to p39]
<i>xor</i> r26, r2, r7	<i>xor</i> p62, p0, p20 [r26 renamed to p62]
<i>sub</i> r20, r26, r2	<i>sub</i> p8, p62, p0 [r20 renamed to p8]
<i>andi</i> r27, r8, 0xffff	<i>andi</i> p19, p25, 0xffff [r27 renamed to p19]
<i>sll</i> r19, r27, 0x5	<i>sll</i> p45, p19, 0x5 [r19 renamed to p45]
<i>beq</i> r20, r19, label	<i>beq</i> p8, p45, label
<i>or</i> r12, r15, r16	<i>or</i> p59, p3, p23 [r12 renamed to p59]

MAINAK CS422 28

So, next we visit our last example let us see how renaming works. So, let us assume that there are 64 physical registers and this is the currently already renamed registers all right. So, r6 currently points to physical register 54 r19 points 2p 38 r 2 points to p0 r7 is p 20 r5 is p 3 r6 is p20. So, there will be a table which will maintain this maps ok.

(Refer Slide Time: 26:32)



So, I will have a tables for MIPS I will have table with 32 entries actually 31, because r0 is actually fixed this r1 to 0. So, in this case for example, we are let us see r2 So, this row is for r2 it will basically holds 0 which, means; that currently any instruction requiring r2



should read physical register 0 all right and etcetera. So, the first instruction comes which has the target r four. So, it should get a new register which register should it get. So, there is a free list of registers that is the predictor maintain which tells which register is free I pick up the first register which is free.

So, let us suppose I pick p15 all right. So, r4 gets renamed to p15 and then the instruction why is that r6 is the source of a register. So, it looks up the map table 1 is that r6 is currently mapped to p54 ok. So, it gets renamed to this. So, why the instruction it finally, issue an execute it will go and read p54 to get the value and put the result in p15 all right. So, the next instruction comes it has r4 at the source looks up the table finds that r4 is renamed to p15. So, it changes to p15 and assigns the new register to r5 p40 all right. And the process continues now you can see that the registers which had a more problem now get 2 different names in that problem actually goes a right.

So, this instruction will read from p38 because at this point r19 was mapped to p38 and here this instruction will get a new register for r19 which is a p point here it cannot be p38 why is that.

The horizontal.

Sorry

The horizontal

It is not about the hallow it is just that p38 is not available at this point which have already allocated to a register. So, we will soon see how to recycle register all right. So, there will be a mechanism whereby will actually free cycle registers, but currently the point is that whenever; something gets renamed to a physical register that physical register is occupied the subsequent instruction cannot use that register until this freed and we will see when an instruction when an when a register gets freed. So, it is a basic algorithm figure it is very simple yes.

Why

Yeah. So, yes I need to a new register

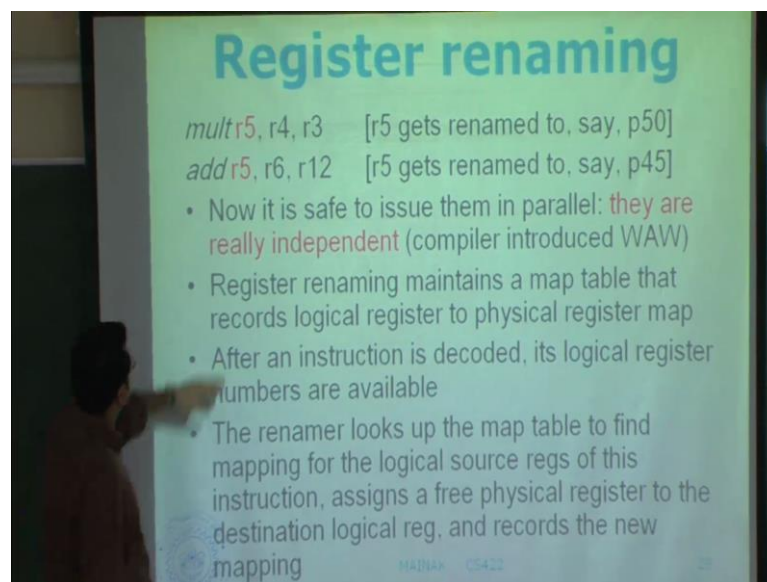
Yeah

Yeah. So, every target gets a new register and the new register is the 1 which is free out of many free registers you will pick up right. So, does this example tell you that you probably need at least 1 more register 1 more physical register than the number of logical registers if you had exactly same ok.

Then there is any problems actually I do not know right; because at any point and time all registers would probably be mapped this only I get a new instruction nothing is free I cannot really allocate. So, to make all progress I need at least 1 extra physical register. So, its formalizes portion that what exactly how it how it rates to execution time and performance any question on the algorithm clear all right.

So, branch instructions do not have a target. So, they do not get a new register they only rename their sources. So, r19 becomes p45 because p45 was assigned r19 here and r20 is taken from here which is p8 r20 was renamed to p8 all right. But there is nothing there is no new register assigned to branch. So, you just rename the sources. So, this 1 now make sure that everything is in a single namespace and instruction issue it knows where to read from the value it reads the value from there and that is it ok.

(Refer Slide Time: 30:55)



So, here is another example how renaming solves a problem of write after write hazard. So, here 2 instruction that write to the same register r five. So, in this instruction r5 will get renamed to some registers at p50 and here it will get renamed to some other register

let say p45 whatever is freed all right. So, now, they can actually execute concurrently they will map to different registers all right.

So, now it is safe to issue them in parallel because they are actually independent ok just the compiler give to shortage of registers architecture registers that introduces the write after write hazard. So, essentially what I am saying here is that well actually you could answer question that since, the compiler was holds to do this why not expose all the registers to the compiler right why to have this complicated hardware inside to do the remaining if the compiler of the registers that it do a better job of register location right what you think why setting done in this way is that question clear right yeah.

The scheduling is...

Yeah yes. So, what I am saying that see, but what I am trying to say is why we are renaming, because you want to get into the name renaming right that is the only paths and why do we have independence because the compiler was probably short of registers that is why you have to reduce this name dependences like here you would very well ask why the compiler choose r19 here it could have chosen a different register right and the reason is that we have an different arrays you figure out that r19 is available which is picked up r19 then the located the panel here. So, it give you give more registers it should why essentially done in this way why Intel processors are start with 8 logical register forever.

Sorry say again...

You do not need to know, but if an expenses of the compiler will get into more bit itself give anymore registers they are still be name dependences it is not guaranty that all name dependences will go away, but probably it would be solo in number that it does not matter anymore, but it is never done you will not be done ever what is that yes.

So, the I s a would change because of adding new registers

Exactly. So, instruction say the architecture have to change

Yeah.

Sorry.

Back out compatibility.

Back out compatibility yes. So, the binary is that combine today will not work tomorrow as soon as you do this ok.

Sir

Yes

Old registers will be assigned

Sorry

Old registers will be assigned

In ISA code used to change right

Yeah

You have to recompile your all you know all applications which is possible

Sorry say again

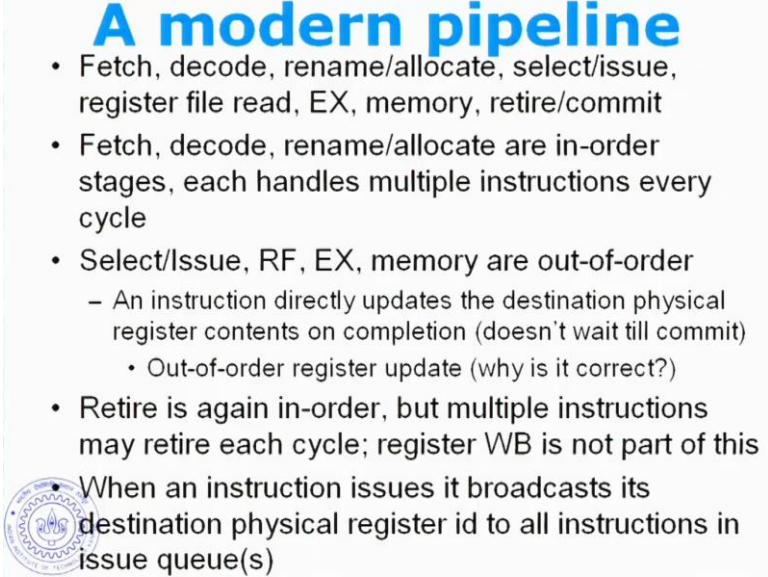
No here is a problem you have bought Microsoft office suite which is compiled for yesterday's 686 Microsoft will issue license to do actually, to all the customers it is a huge aided for the software industry. So, it is primarily business angle why this is not done theoretically there is no problem we could do actually all right. So, just to hide this problem processors are doing which inside the in the pipeline this is clear ok.

Although, I was mentioned actually a and b took a took an interesting step when the design obtained they actually introduced 16 logical registers and shows that it actually works. So, of course, they came with the lie that if you want to make use of this you have to decompile binders, but your old binders will continue to work with lower performance that is all right. So, then of course, the d clearly all right.

So, register renaming maintains some map table that records a logical register to physical register map after an instruction is decoded its logical register numbers are available the renamer looks up the map table to find mapping for the logical source register of this

instruction assigns a free physical register to the destination logical register and records the new mapping.

(Refer Slide Time: 36:05)



### A modern pipeline

- Fetch, decode, rename/allocate, select/issue, register file read, EX, memory, retire/commit
- Fetch, decode, rename/allocate are in-order stages, each handles multiple instructions every cycle
- Select/Issue, RF, EX, memory are out-of-order
  - An instruction directly updates the destination physical register contents on completion (doesn't wait till commit)
    - Out-of-order register update (why is it correct?)
- Retire is again in-order, but multiple instructions may retire each cycle; register WB is not part of this
- When an instruction issues it broadcasts its destination physical register id to all instructions in issue queue(s)

So, that is the renaming procedure. So, how the pipeline does would like. So, these are my pipeline stages, I am not saying really that each stage is a cycle each stage is a cycle So, fetch decode then rename allocate. So, in the rename allocate stage you rename your logical registers source registers assign a new physical register destination you allocate ROB entry you allocate issue queue entries then you do a selection issue whenever; the instruction is ready when should it be ready then you read the register file. So, when you read this you read the physical register of course, all right.

Then execute lookup memory period and then retire commit. Let us not come to this particular stage, because now the question is what this stage is really it mean because previously we were actually copying values from ROB to the register file now that the mean is gone, because I can execute I can write the register immediately because each instruction vomits the different registers. So, there is no question of write after write hazard or write after read hazard I can write the register immediately.

So, you see what that particular stage does. So, fetch decode rename allocate are in order stages each handles multiple instructions every cycle meaning that here you go sequentially fetch multiple instruction decode 1 of them rename all of them and locate all of them all right; select issue register file read execution memory lookup are out of order

they can go you can execute instructions as an when they become ready you do not have to obey the order all right.

An instruction directly updates the destination register file destination physical register contents on completion does not wait till commit we will soon argue that this is correct although intuitively it is correct because every instruction gets a new physical register. So, there is no question of complete and also there is no question of write after read hazard yeah. So, out of order register updates that happens actually now and we have to argue that why is it correct. So, next we will come to that point and retire is again in order; so, by the way.


So, this is the technical term of a news for from it; that means, an instruction of the time which means it was born whenever fetch and finally, it is done we right. So, retire is again in order, but multiple instructions may retire each cycle and register write back is not part of this anymore all right it happens that is part of the execution fetch all right; when an instruction issues it broadcast its destination physical register I d to all instructions in issue queues and that is essentially the wakeup logic. So, every instruction will compare its source physical registers with the destination of the broadcast register.

(Refer Slide Time: 39:02)

## Register renaming

- Assume 64 physical regs and already renamed registers:  
r6=p54, r19=p38, r2=p0, r7=p20, r15=p3, r16=p23.

<i>lw</i> r4, 0(r6)	<i>lw</i> p15, 0(p54) [r4 renamed to p15]
<i>addi</i> r5, r4, 0x20	<i>addi</i> p40, p15, 0x20 [r5 renamed to p40]
<i>and</i> r10, r5, r19	<i>and</i> p39, p40, p38 [r10 renamed to p39]
<i>xor</i> r26, r2, r7	<i>xor</i> p62, p0, p20 [r26 renamed to p62]
<i>sub</i> r20, r26, r2	<i>sub</i> p8, p62, p0 [r20 renamed to p8]
<i>andi</i> r27, r8, 0xffff	<i>andi</i> p19, p25, 0xffff [r27 renamed to p19]
<i>sll</i> r19, r27, 0x5	<i>sll</i> p45, p19, 0x5 [r19 renamed to p45]
<i>beq</i> r20, r19, label	<i>beq</i> p8, p45, label
<i>or</i> r12, r15, r16	<i>or</i> p59, p3, p23 [r12 renamed to p59]

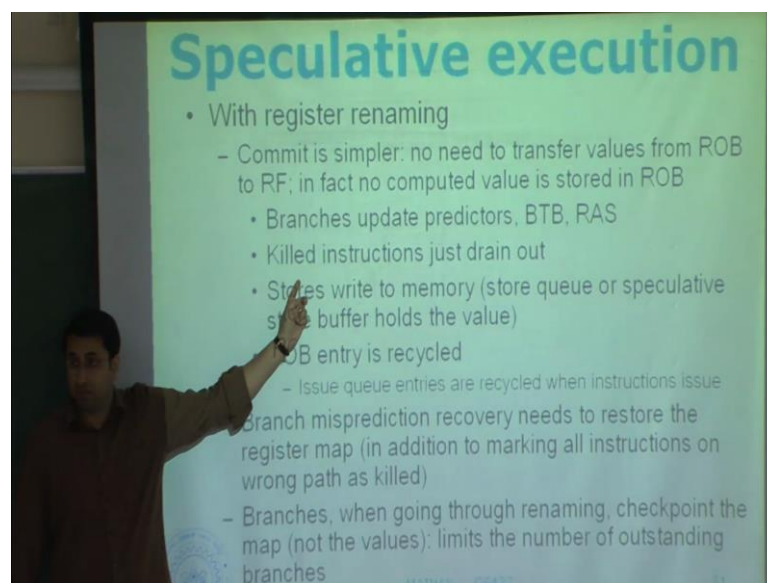

MAINAK CS422
28

So, for example, here when this instruction issues it will broadcast p15 to all the instructions and this guy will have a match and we know that now the next cycle probably an issue. So, in this case probably cannot issue in the next cycle with the

probably wait for up to more cycles. Similarly, when this instruction issues it will broadcast p62 and this instruction will magic source it will actually wakeup and know that next cycle it control all right. So, that is how the wakeup time any question.

So, with register renaming what does speculative execution now look like commit is much simpler because there is no need to transfer values from ROB to register file. In fact, no competitor value stored in ROB now the ROB only stores a few step is not the instruction and just maintains the order all right.

(Refer Slide Time: 40:13)

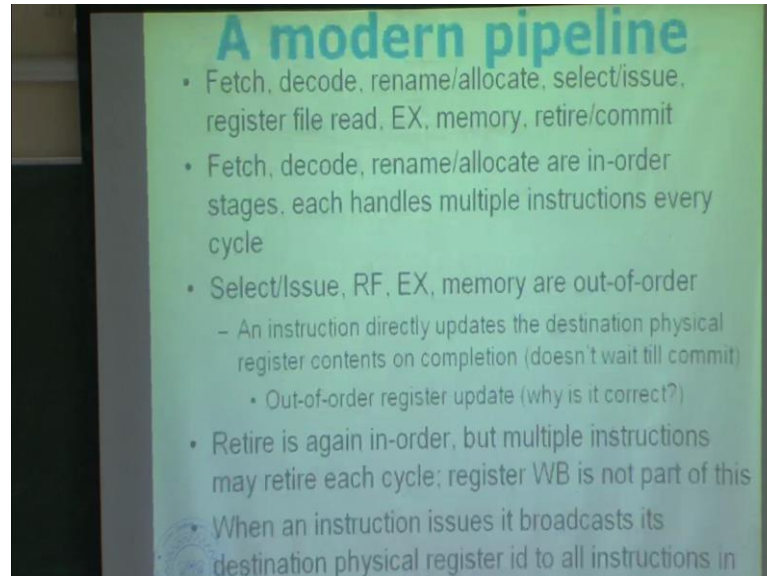


Branches update predictors branch target buffers and return address stacks when they commit killed instructions just drain out. So, these are the instructions that are on the wrong path later discover because of misprediction. So, they just get killed and ROB we just remove them from the from the from the processors also stores write to memory at this time store queue or speculative store buffer holds the value. So, remember that we said that we distribute the issue queue into functional units right.

So, memory subsystem will get 2 issue queues 1 is the load queue another 1 is the store queue. So, the load instructions will go to the load queue store instruction will go to store queue and the store queue will also have the value that it needs to store. So, when the store comes to the head of the ROB the value would be moved from the store queue to memory and finally, the ROB entries recycled. So, that it can normally used all right and

remember that issue queue entries are recycled when instructions issue. So, they are issue they are recycled much earlier in the ROB entry.

(Refer Slide Time: 41:26)



So, ROB entry is held if you go back to the pipeline here. So, ROB entry gets allocated in the allocate pipe stage and it remains held until the instruction moves to this stage all right. So, it is a pretty long time actually whereas an issue queue entry is allocated here in allocates stage, but gets read as soon as an issue, because there is no need for this issue queue entry anymore instruction and ambition all right. So, issue queue entries actually get remain occupy for short of period whereas, ROB entries are extra very long period.

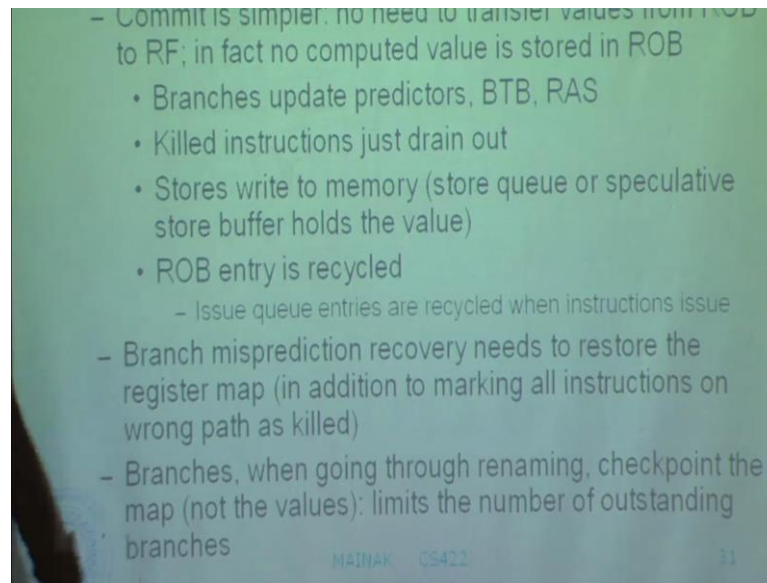
So, what is the implication on that relative sizes of these 2 things ROB in issue queue the ROB is in issue queues ROB entries remain occupy for longer period of time where issue queue entries remain occupy for shorter period of time. So, which 1 should be bigger ROB or issue queue.

ROB

ROB right that is pretty of this right branch misprediction recovery is to rest of the register map. So, we have talked about this rest of do not checkpoint the same thing will continue whenever a branch is renamed you take a checkpoint of the of the map table and if it is discontinue rest over the map table.



(Refer Slide Time: 42:52)



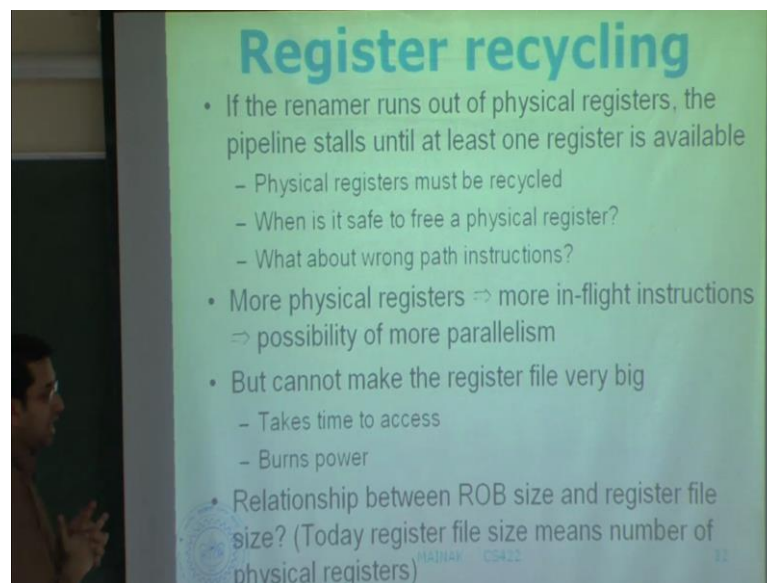
In addition to marking all instructions on wrong path as killed. So, these are these instructions a killed instructions ok. Branches when going through renaming checkpoint the register map remember that these are not branches is only the map that is checkpointed. So, here for example, when this branch instruction shows up when this renamed it would only checkpoint the map meaning that it remembers that at this point of time r19 was holding the map p45 r 27 was holding p19 and so, on and so, forth the value of p45 is not checkpoint value of p45 can be whatever; we does not care.

So, we will soon argue why it is correct, because it is not make obvious that that we will not check pointing the value of p45 we are only check pointing the map that r19 corresponds to p45 ok. So, could it be that some instruction actually updates p45 in a wrong way it may a later of course, I can recover in that the only observation here to make is why it is correct is that any instruction that counts. Let us suppose this branch was actually mispredicted any instruction that comes after the branch will not be allocated p45 it will be allocate some other register.

So, p45 can only be updated by this instruction that cannot be anything else updating p45 So, you do not need to checkpoint the values you only need checkpoint the map all right. Second of course, we are continuously make an interesting assumption that there is algorithm which recycles registers in a same way we will come to that actually the recycling algorithm of registers ok.

So, this is the implication of this is that this also we talked about limits the number of outstanding branches, because it depends on how many checkpoints you can accommodate we will solve all right. If your processor can accommodate hundred checkpoints then your process sheet will support hundred and resolve outside in branches it quite all right usually the number is much smaller than hundred which may be tens of branches any question ok.

(Refer Slide Time: 45:22)



## Register recycling

- If the renamer runs out of physical registers, the pipeline stalls until at least one register is available
  - Physical registers must be recycled
  - When is it safe to free a physical register?
  - What about wrong path instructions?
- More physical registers  $\Rightarrow$  more in-flight instructions
  - $\Rightarrow$  possibility of more parallelism
- But cannot make the register file very big
  - Takes time to access
  - Burns power
- Relationship between ROB size and register file size? (Today register file size means number of physical registers)

MAJNAH CS422 12


So, the last point how to recycle registers any cycle why should I let us go back to the example.

(Refer Slide Time: 45:39).

## Register renaming

- Assume 64 physical regs and already renamed registers:  
r6=p54, r19=p38, r2=p0, r7=p20, r15=p3, r16=p23.

<i>lw</i> r4, 0(r6)	<i>lw</i> p15, 0(p54) [r4 renamed to p15]
<i>addi</i> r5, r4, 0x20	<i>addi</i> p40, p15, 0x20 [r5 renamed to p40]
<i>and</i> r10, r5, r19	<i>and</i> p39, p40, p38 [r10 renamed to p39]
<i>xor</i> r26, r2, r7	<i>xor</i> p62, p0, p20 [r26 renamed to p62]
<i>sub</i> r20, r26, r2	<i>sub</i> p8, p62, p0 [r20 renamed to p8]
<i>andi</i> r27, r8, 0xffff	<i>andi</i> p19, p25, 0xffff [r27 renamed to p19]
<i>sll</i> r19, r27, 0x5	<i>sll</i> p45, p19, 0x5 [r19 renamed to p45]
<i>beq</i> r20, r19, label	<i>beq</i> p8, p45, label
<i>or</i> r12, r15, r16	<i>or</i> p59, p3, p23 [r12 renamed to p59]

 MAINAK CS422 28

So, when should I free p15 when can another instruction use p15

When add I has finished

When add I has finished why no that there is no instruction in future that is the reason that is use p15.

There is cone from that into r4

Why.

And either instruction write to r4.

Sorry say again

Write any other instruction write to r4 we

Do I have any such situation here?

I do not have any common target here you are saying that if there is an instruction which writes to r4.

When. So, an instruction now has several in it is not really a point in that right its spans over certain cycles. So, you can fetch.

That that is instruction which is

So, we have instruction which tries to r4 ok

When other instruction maintains we commits there

At that time entry obtain

So, essentially what you are saying is that let us suppose an instruction here writes to r4 all right. So, at the time r4 will be given a new map which is some register whatever the that the register is free whatever, whichever register is free I can point and he saying that when this instruction finally, commits I am guaranteed that p15 can be recycled it make sense because from now on r4 has a new committed name right whatever was assigned here. So, r4 to p50 map now must expire and I should be able to reuse p15 for some other register make sense does everybody see that it is a conservative policy I could have done better why.

Because I think r4 is not used for a...

We can just expire the queue this queue

Other registers.

Right.

So, do you see any difficulty out of that...

You are right, yes that is we search in the problems why it is conservative do you see how will do that let me answer this question I really cannot see the future that is the problem. So, what do I need to determine I need to determine if beyond the certain instruction p15 is dead there would not be any instruction will be increase in thing right that is all execution my add r deserving that I could have done that actually right yeah. So, there are there are research papers which have looked at this problem you can actually design predictors which can predict last starch to a certain register all right. So, that is actually.

Need to...

Sorry

Store that which is store that information.

Is this your static information starting from point you could follow different control path right and in different control paths register may become get a different points yeah you could remember the whole thing on this path the register becomes led here on this path yeah of course, because we have storage you could remember the whole thing actually yes.

Sir when 1 instruction wakeup the other instruction

Yeah

So, we can check we can drag this thing.

Overall in instructions are getting later

All right.

So, whenever old instruction as said we committed...

Wakeup instruction when its different to...

No does everybody see. So, what he is suggesting is that when this instruction issues it will broadcast p15. So, all the instructions waiting in the issue queue we will we will compare and you would know what instruction waiting for p15 and when all of them actually have committed you can.

Further wakeup are in the...

How do you know there will not be any further improbability.

We should wakeup...

It is not about the wakeup there may be some instruction which is not any fetched actually which require r4 right. So, yeah; so, as such it is not an easy problem which is why there is processor actually do what he has suggested we wait until r4 is overwritten and that instruction when commits I know that the new map has not committed. So, I can now free p15 ok.

Now, the easy way to do that. So, how do you really implement that is now the question is and the difficult is that when this instruction commits you cannot really say that well; I look up the map table and find out what r4 maps when I free that register that we know because r4, now point to some totally different register. Because, by the time this instruction commits there will be many more instruction to the pipeline which already have overwritten R4. So, how do you really free p15 how do you find out that actually that p15 is the 1 to be freed when this instruction finally, commits that user r4 that that overwritten example yes.

The show me ROB get...

Yes exactly. So, you remember in the ROB. So, when you rename this instruction you when you assign r4 a new map you look up the table there finally, oh r4 now maps to p15 that this 1 you just remember fifteen instead of ROB entry when you find the commit you know what is looking as table all right. So, that is why register recycling algorithm. So, now, it should tell you why I do not need to checkpoint value for branches because there is no way p45 is going to be freed and going to be reassign before this branch commits because any instruction that overwrites p45 will have to come later and it cannot commit actually all right ok.

So, yeah; so, if the renamer runs out of physical registers the pipeline stalls until at least 1 register is available. So, physical registers must be recycled when is it safe to free a physical register we have just discussed what about wrong path instructions. So, yeah; so, that is an interesting question actually. So, think about this instruction this 1 right. So, here r12 gets mapped to p59 all right and the pipeline continues moving eventually, you figure out that this branch 1 mispredicted which means; this instruction should not happen to fetch actually. So, now, what we will do with p59 when does p59 get freed.

Should be free now only...

Should be free; now only right does everybody see that as soon as this instruction comes to the head of the ROB in the killed state of course, I can free p59 immediately because it is actually r1 committed map. So, there is no problem in free P59 and using it because whatever value that was put in p59 is a garbage value that we ignore until you ignore it and overwrite all right ok.

So, more physical registers means more in-flight instructions that should be now clear actually because as soon as I valued a registers my pipeline has to strong. So, if have more registers I can keep on renaming registers and then keep on moving my pipeline right and then also it is a more possibility of more parallelism. So, number of registers has great connection with how much identity you can expose.

But, cannot make the register file very big because there are down sides it takes time to access and it will be gets bigger you know there is a there is a thumb rule that you have to remember that is smaller and faster and usually branches structures are more all right. So, as you make it bigger and bigger it is going to be slower and slower yeah. So, you start using c b I or in some other place and also it burns power which is very important the r structures usually take energy what is the is there any relationship between ROB size and register file size. So, so today register file size means the number of physical registers what you think should directly mechanically it gives me size independent.

Which reduce size of maximum less than or equal to.

Wait a second what is less than or equal to

The ROB size is less than or equal to the register file size

Why

Mapping of ROB register

Mapping a physical register. So, at match is a map always physical register we have look at the when we are using all these register we have a mapping for register.

You are saying ROB size is less than or equal to physical register size right sorry.

All the definitions are...

(Refer Slide Time: 55:09)



Between 2 between

Right ok

Whatever the size of this physical register file

Yes

So, that may be increase

Why

There is 1 more entry

There is

And it has the same destination register

There is no corresponding mapping to the register

If all the instructions have the same destination register then; I will be actually recycling registers right quietly good.

Commit

Yeah



Because cannot be commutative.

The worst case would actually, be that all right fine. So, what I am suggesting. So, less than equal to what is it. So, you are saying that you are right. So, you are saying as many physical registers I have, I could I should remember rename those many instructions. So, usually is the equality what make sense.

Less than equal to

Sorry

Less than equal to

Less than equal to yes, because if I have a bigger ROB some of the slots will remain empty all the time, because I do not rename this instruction all right; in the worst case can I title this a little bit if I tell you that I have eight logical registers.

1 less than 2

Sorry

1 less than it remains on

Remains on what it remains on

ROB size

No this 1 we have already. So, you are saying ROB size not bigger than  $m - 1$  why is independent of this size is not when logical registers.

Yes

Lookups how do you have them

Sir, basically like your using 2 names per single register like if a register is between see r4. So, add I after the source and then other instruction will using r4 as a r4 vector r4 register. So, we can.

Right

So, we cannot would not use overwritten for a single logical.

It would be 348 vectors instead will not free until that instruction is vomited right you do not know when is the return.

Because, they are not after the road instruction you would use a same the same physical register.

No you are saying that there are 2 instruction, with the same target.

No

No sorry yeah

Sir, 1 instruction is using

Ah

So, 1 instruction is using say 1 has. So, much 1 after the source. So, it has 1 happen we really 1 into the and 1 one is using it as a destination. So, we use the 1

Which comes first.

Sir the destination 1 and after the

Fine

So, we have to use second entry of this

Right ok.

And we should we should not require more than

What you require another instruction that uses the same registers until

Another name

Any another name

We can read a

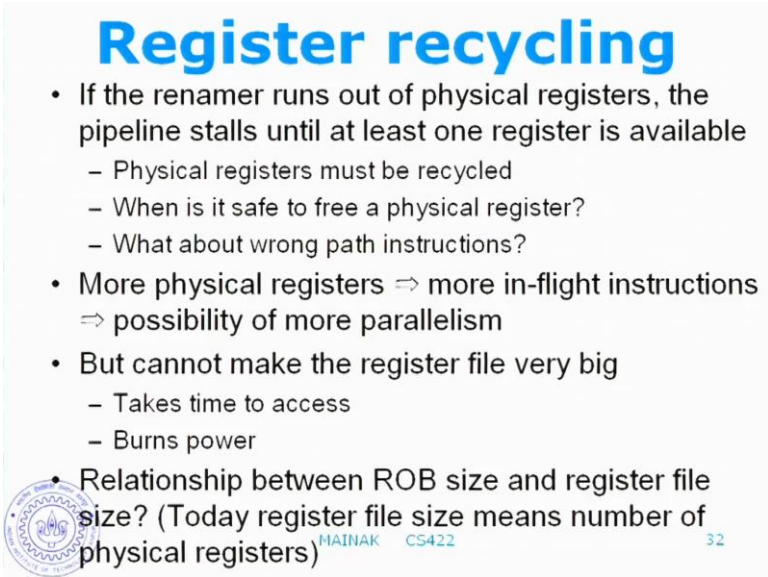
Only 1 that commits

1 that

You do not know why you written by the way you may have anymore information of the same register. So, if I had an ROB size.

Greater than or equal to  $n + 1$ .

(Refer Slide time: 58:49)



## Register recycling

- If the renamer runs out of physical registers, the pipeline stalls until at least one register is available
  - Physical registers must be recycled
  - When is it safe to free a physical register?
  - What about wrong path instructions?
- More physical registers  $\Rightarrow$  more in-flight instructions  $\Rightarrow$  possibility of more parallelism
- But cannot make the register file very big
  - Takes time to access
  - Burns power
- Relationship between ROB size and register file size? (Today register file size means number of physical registers)

ROB size should be

Greater than or equal to  $n + 1$

Greater

Greater than

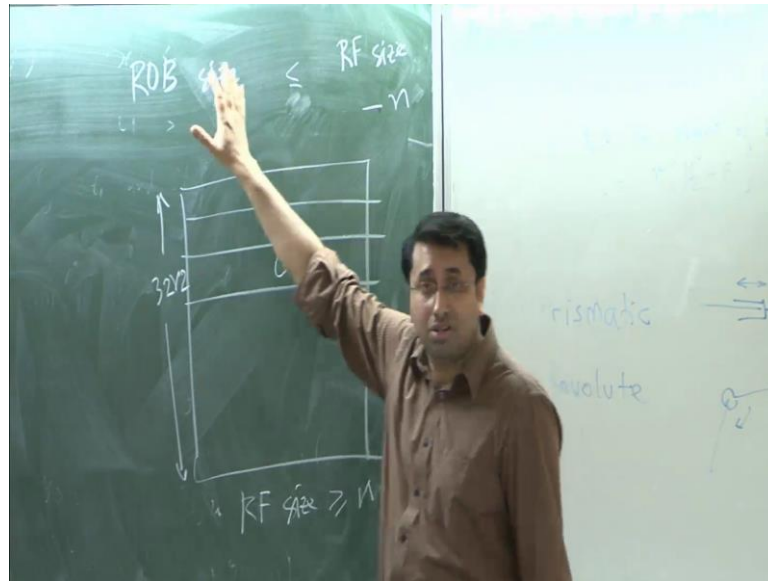
Greater than why

Number

You know we have register worse than; I do not know how your bonding structure you find and you will not find what is the problem all; I am saying is that the minimum number of physical register occupying within 10 at any point and time.

Register file size should be greater than  $n$

(Refer Slide Time: 59:22)



So, we have that we have already discussed right that ROB size should be greater than equal to  $n + 1$  great forward problems. So, can somebody put  $n$  on some side here. So, I have already  $n$  physical registers occupied now I start fetching instructions and putting in ROB can I say like this  $n$  always occupied whatever I have left with only that much I can allocated in the worst case right of course, eventually they will get freed, but I do not know when they will be freed. So, I should not happen ROB size bigger than; this ROB size should less than or equal to  $r f$  size register for the cycle is that understandable.