Computer Architecture

Prof. Anshul Kumar
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

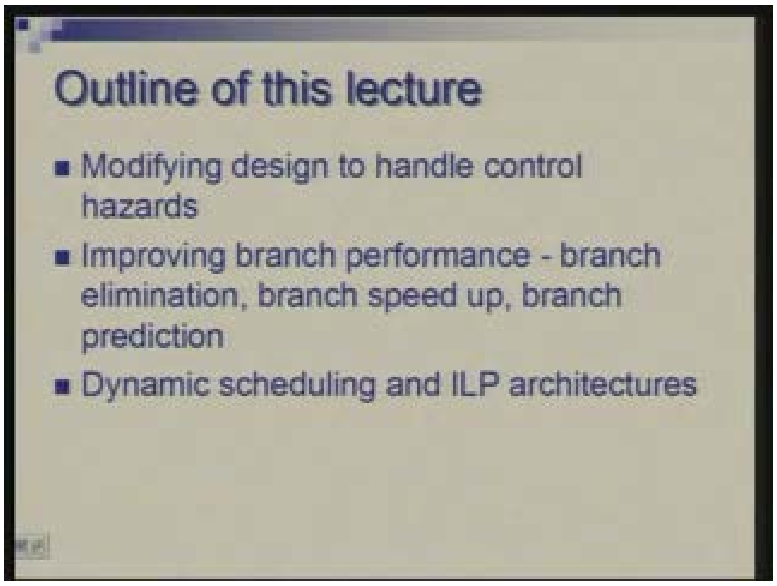
Lecture - 27

Pipelined Processor Design: Handling Control Hazards

We have been discussing pipelined design for MIPS processor. Last time we had seen how we can handle data hazards by introducing stalls if necessary and we also saw how we can introduce bypass paths or forwarding paths so that delay can be cut down.

We noticed that in some cases delay may still be required or the stall cycle may simply be required and we worked out the logic which is required to be put in the controller to take care of this situation. So you need to detect when hazards are occurring; you need to basically find out the dependency and then introduce appropriate control signals. Today we are going to look at the issue of handling control hazards.

(Refer Slide Time: 00:02:04)



Outline of this lecture

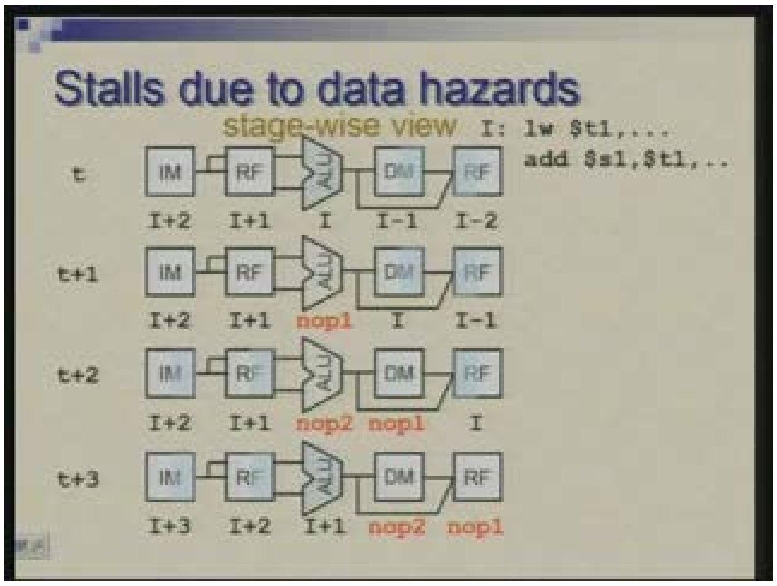
- Modifying design to handle control hazards
- Improving branch performance - branch elimination, branch speed up, branch prediction
- Dynamic scheduling and ILP architectures

So we will take the design which we have discussed so far and modify it to handle the control hazards, then we will see how we can improve performance in view of control hazards or branch hazards and a couple of techniques we will look at briefly and not go into too much of detail.

We would look at the possibility of eliminating branches altogether in some special cases or trying to speed up the execution of branches or introduce something what is called prediction of branch. So you do branch prediction and try to take action accordingly.

Finally I will make a brief mention about dynamic scheduling which is used for instruction level parallel architecture; that is another way which is used to keep the pipeline full in view of branch hazards and data hazards.

(Refer Slide Time: 00:03:07)



Stalls due to data hazards

stage-wise view

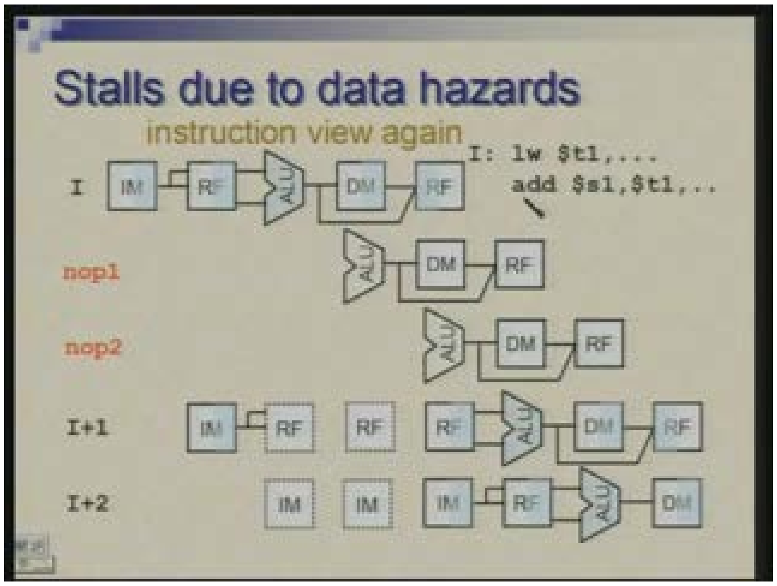
```
I: lw $t1,...
   add $s1,$t1,..
```

	IM	RF	ALU	DM	RF
t	I+2	I+1	I	I-1	I-2
t+1	I+2	I+1	nop1	I	I-1
t+2	I+2	I+1	nop2	nop1	I
t+3	I+3	I+2	I+1	nop2	nop1

So just to recollect what was the effect of data hazards on execution of instructions. We had seen that when data hazards are there you need to introduce null instruction or nop instructions or bubbles in between instructions.

So, for example, looking at a sequence of instructions in the stage wise view where you are showing various stages and vertically you are looking at various cycles or time instants you would notice that nops are getting introduced when there are two instructions which are dependent on each other.

(Refer Slide Time: 00:03:43)



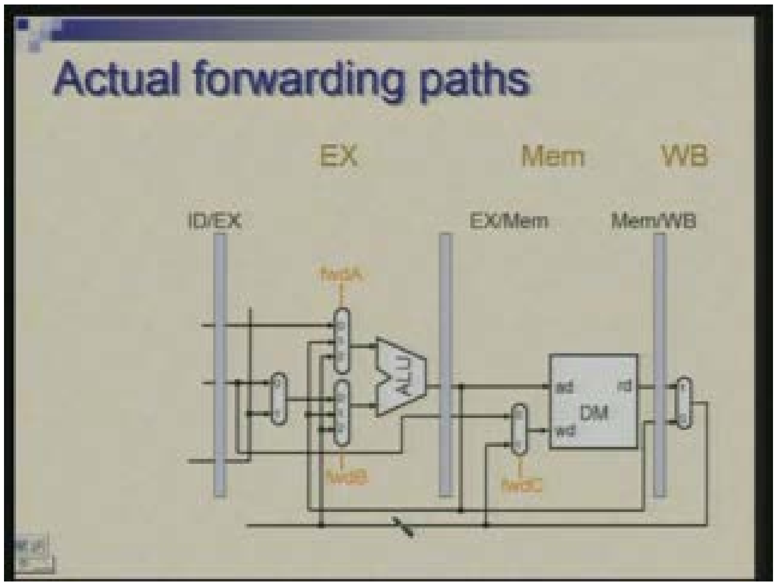
Stalls due to data hazards

instruction view again

```
I: lw $t1,...
   add $s1,$t1,..
```

The same thing is seen in another view where you are looking at instruction by instruction and here are the two nop instructions because the instruction which is dependent is getting delayed and this delay is passed onto subsequent instructions also so everything behind this instruction in the pipeline gets delayed.

(Refer Slide Time: 00:04:08)

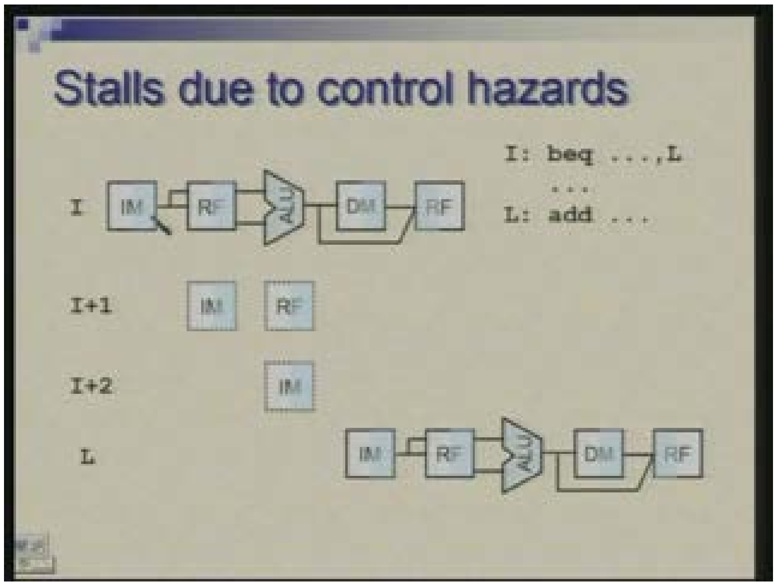


Actual forwarding paths

So, to improve things we had identified where we can have forwarding paths. So this diagram, for example, shows that you can have paths going from output of ALU through this inter stage register back to ALU or you could have output of the memory stage after this register and possibly after multiplexer going back to memory and also back too ALU.

So now you need to derive control signals which guide these multiplexers. So you need to detect which source address is matching with the destination address in the two instructions and accordingly enable these paths.

(Refer Slide Time: 00:05:02)



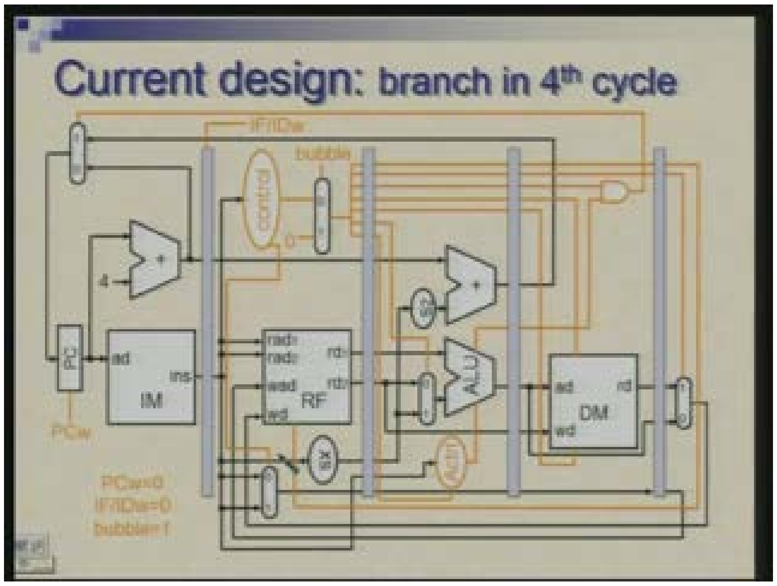
Stalls due to control hazards

```
I: beq ...,L
    ...
L: add ...
```

Now, coming back to the control hazards we now imagine a branch instruction like this (Refer Slide Time: 5:05) and these are the stages it is going through and here we take some decision and either continue sequentially or branch to an instruction label L.

So we assume that if the processor continues getting instructions in sequence; at some point suppose we come to know that there is a branch and the condition has become true then the address change is the sequence is broken and you can fetch the target instruction here but now you realize that the two instructions which had entered the pipeline are not the one you intended and therefore they need to be flushed out so you need to actually generate a control signal which will do this.

(Refer Slide Time: 00:05:53)

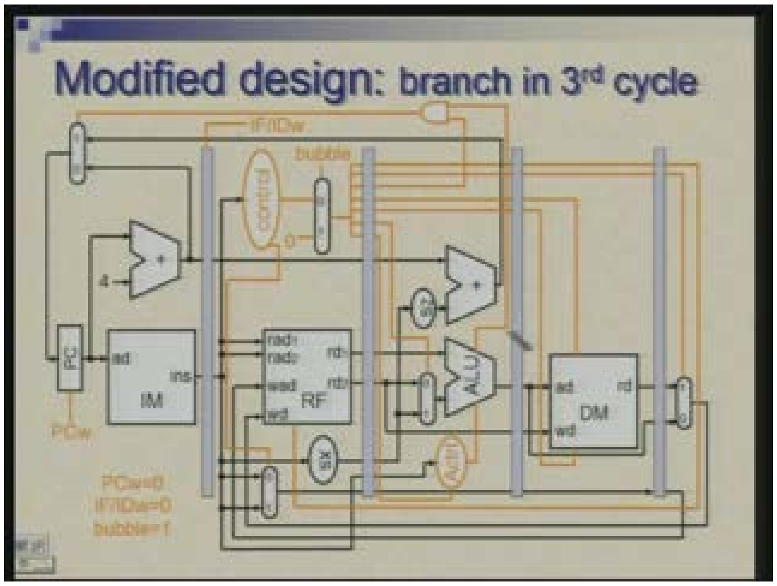


Here is the datapath and controller which we have designed so far. Now there is one thing which we need to notice here is that here actually the way we have designed the branch will take place in fourth cycle unlike what we expected here; we thought that the condition regarding tested in ALU and the next instruction can be fetched in this cycle. But what is happening in this datapath is that we are looking at the target address after this register and we are also looking at the condition which is tested by ALU after this register so that means it is in memory stage only. We are looking at this outcome of the branch and here is it the control signal is being generated for feeding to this multiplexer.

So basically if this is how it is done then there will be one more additional delay. So this instruction actually can start only.... the instruction will be fetched and be available from this cycle onwards because at the end of fourth cycle you are latching the new address into program counter and therefore the fetch cycle shifts to this. So we will try to improve this by tapping this output the new address and the control signal before the register (Refer Slide Time: 7:34).

So this is the change which is shown here.

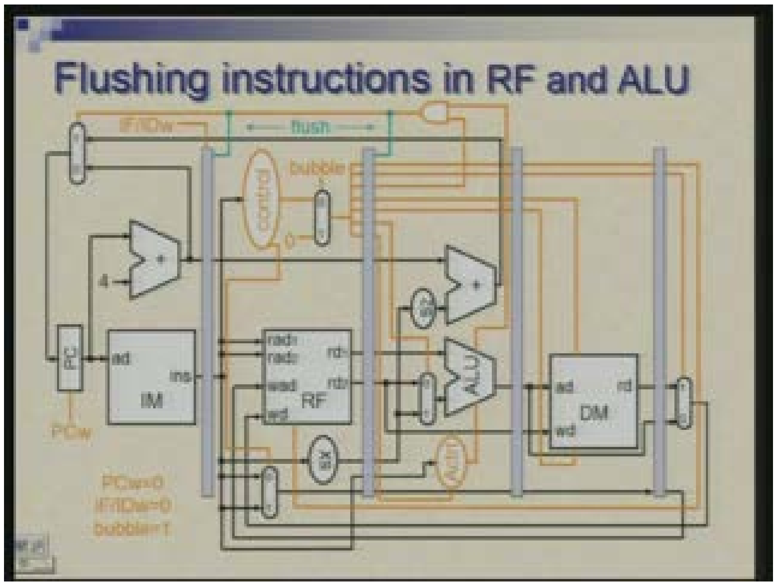
(Refer Slide Time: 7:37)



We are not making these pass through this register (Refer Slide Time: 7:42). The consequences of this are that the delay calculation which ultimately determines the clock period would undergo a change. Now you need to account for the multiplexer delay of this which we have ignored so far along with this adder and also with this ALU to look at the to consider the path delays.

So now if you look at the paths going from this register to some registers in this case PC so you need to account for this ALU, this AND gate, (Refer Slide Time: 8:28) this multiplexer and then to PC so this is one path. Similarly, the path you need to consider is that going through S2, adder, then AND gate and multiplexer and so on. So the delays get redistributed and whatever is the influence of this change on the clock has to be borne in mind. but what we The objective here is that we should be able to decide at the end of third cycle what the next instruction is so this is the change which will enable that.

(Refer Slide Time: 9:06)

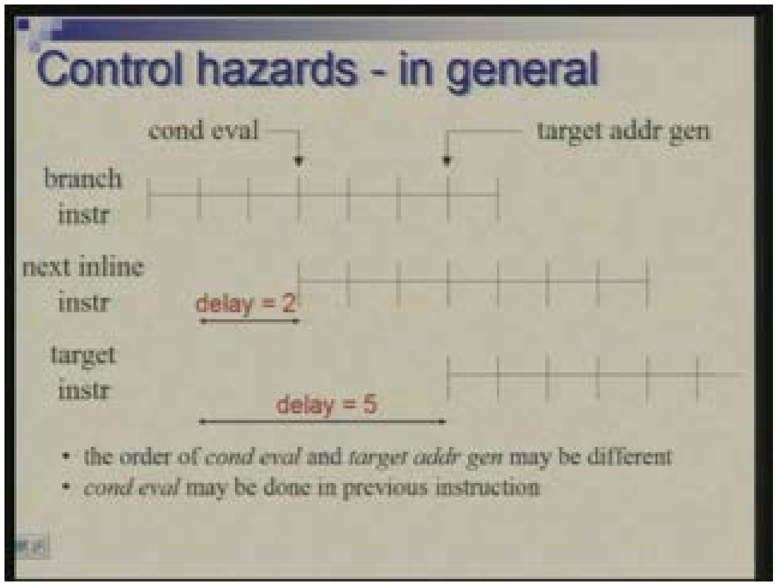


Now having done that we need to flush the instructions which have entered the pipeline in anticipation but are not required. So when we are loading the new value in PC, when we are loading the target address in PC that is the signal which is actually making it possible. We need to use the signal to flush out the instructions which are in these two stages. So the instruction which would have normally gone from this stage to this stage and the instruction which would have gone from this stage to this stage (Refer Slide Time: 9:50) would be flushed if you make the contents of these two registers as 0. So effectively you have made these as nop instructions. What we had realized is that if you make everything in this register 0 it acts like a nop instruction that is what we were trying to do when we were introducing nops in the case of data hazards.

So, similar action somewhat similar action is required that we need to make contents of this register and this register 0 when the signal is active. So effectively we have flushed away these two instructions and do not allow them to proceed forward. Is that alright? Is there any question about this?

So this is how we can handle the branches in the most simple manner. In general, now we realize that there are two things which are happening in a branch instruction in general that there is a condition which is being evaluated and there is an address which is being evaluated so two things would typically happen which would happen in some particular cycle.

(Refer Slide Time: 00:11:03)



Control hazards - in general

cond eval
target addr gen
branch instr
next inline instr
delay = 2
target instr
delay = 5

- the order of *cond eval* and *target addr gen* may be different
- *cond eval* may be done in previous instruction

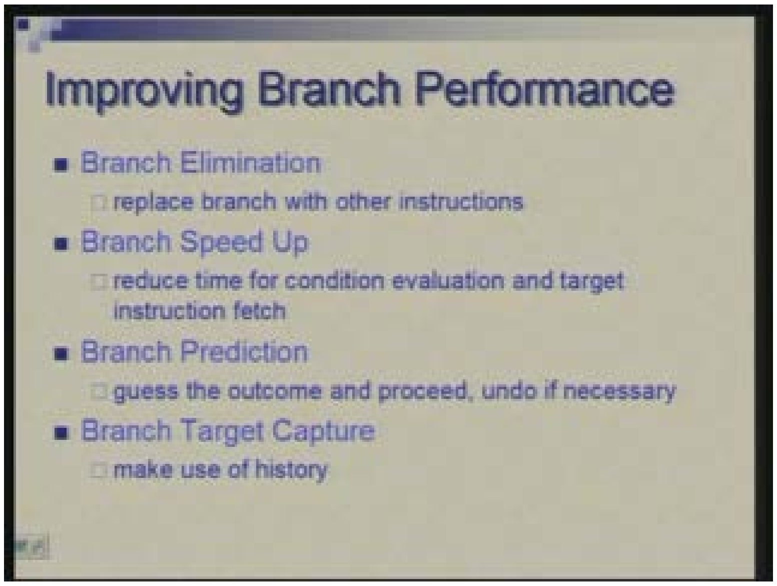
Now suppose there was a deep pipeline; somewhere there is a condition evaluation which is getting completed somewhere, target address of the machine is completed. So, what is the earliest you can have next instruction if you are going inline that means you are not branching and what is the earliest you can have the next instruction if you are going to the target that is the branch is taking place. So the particular cycle is where these two activities are happening would determine how you can place the next instruction. So whatever technique you are applying if you are predicting one of the two alternatives and going that way then we need to keep in mind these delays. So to make things better we this is what we need to cut down.

In our example here this was happening. We started with this (Refer Slide Time: 12:05) where basically the computation of next address and condition evaluation were getting completed in the third cycle and in fourth cycle we were using it. So if you can reduce this for example we finished this in the third cycle itself (Refer Slide Time: 00:12:20) and took decision there so thereby we reduced the delay. If we can reduce it further as we will see later on we can make things even better. So this is what has to be kept in mind.

Another factor is that there are architectures where condition evaluation and branching are split into two instructions. So you would have for example a subtract instruction and the result of that will be used to set some flags. The branch instruction will simply test the flag; it will not actually do the comparison it will only test the flag and carry out branch. So condition evaluation there is trivial. In fact the real evaluation takes place in an instruction which is earlier. So it could be immediately preceding instruction or it could be an instruction which is occurring one or two instructions earlier. So we need to, therefore we need to see where exactly in the previously instruction the condition is getting evaluated.

Now, there are several ways in which you can improve the performance in view of branch hazards.

(Refer Slide Time: 00:13:34)



Improving Branch Performance

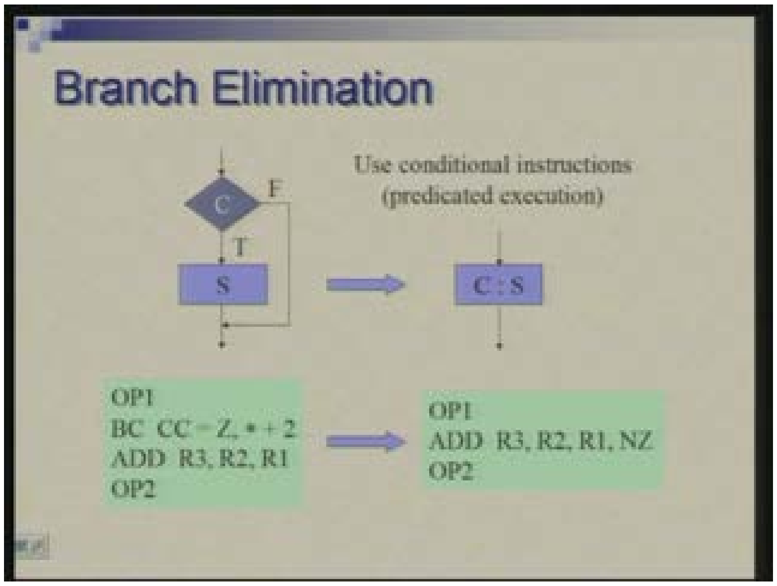
- Branch Elimination
  - replace branch with other instructions
- Branch Speed Up
  - reduce time for condition evaluation and target instruction fetch
- Branch Prediction
  - guess the outcome and proceed, undo if necessary
- Branch Target Capture
  - make use of history

There are cases when you can eliminate branches altogether and therefore get rid of the problem. You can speed up branch execution this is what I was discussing in the previous slide that you can take these decisions earlier or you can move these events as early as possible or increase the gap between decision making and actual branching so we will see that in some more detail.

Then third thing is branch prediction where you do something in anticipation in a manner that most of the time you are correct and when you are correct you are saving a lot of time. If you in the rare event of not getting it correctly you lose time but in probabilistic sense if you take average then on the whole you would have improved the situation. So this branch prediction can be done statically or dynamically; I will distinguish between these two how that is done and lastly particularly when you are doing dynamic prediction you are looking at what happened in the past so when you are doing that you can also remember the target address and thereby also speed up not only the decision but calculation of the target address.

Here is the small illustration of branch elimination.

(Refer Slide Time: 00:15:02)


Branch Elimination

Use conditional instructions (predicated execution)

```
OP1
BC  CC = Z, * + 2
ADD  R3, R2, R1
OP2
```

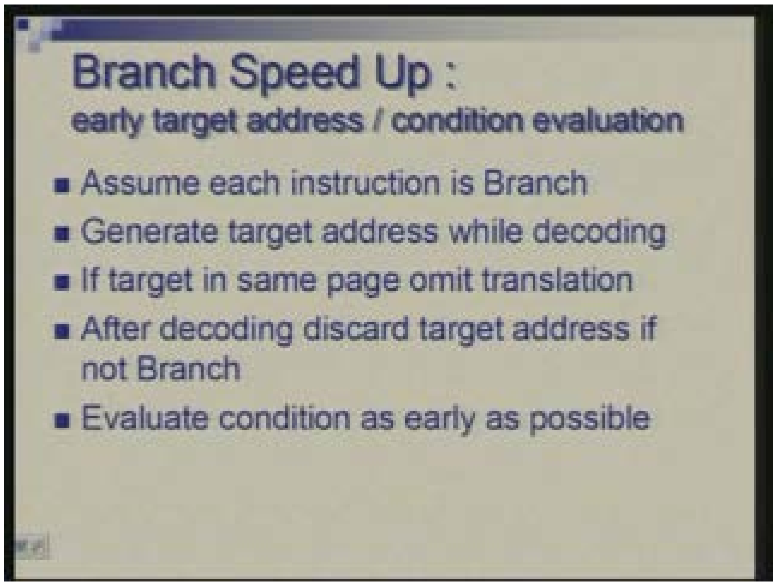
```
OP1
ADD  R3, R2, R1, NZ
OP2
```

Suppose you are testing some condition and if the condition is true you want to do one step or one instruction and if the condition is false you want to skip that. So it is a very small if or conditional structure so there may be one instruction or two instructions not too many. These can be replaced by what is called conditional instruction or predicated instruction. So, some processors have this feature that with almost every instruction you can attach a condition so you would have some field which will indicate that this instruction needs to be done only under certain conditions. So essentially you have fused these two; you have attached the condition with the instruction itself and if the condition is true the instruction gets done otherwise it is like nop.

So, suppose there was conditional branch here so this is not MIPS language it is testing if the condition...... this specifies which condition is being tested so condition being tested is the zero flag which indicates if result of previous instruction was 0. So if z flag is set then you branch to current instruction plus 2 that means you skip ADD. So, if ADD instruction had a provision of specifying a condition you could put that together. So here we are saying that you do this if z flag is not set; NZ means nonzero. So we are removing this branch instruction explicit branch instruction and putting that condition with the add instruction itself.

So now you might wonder that whatever you do the condition has to be somewhere or the other tested so yes that is true but now condition testing is part of the ADD instruction so you could start preparation for doing the addition and depending upon the condition you may store the result or you may not store the result. So if you look at this as a whole you would have saved a few cycles.

(Refer Slide Time: 00:17:30)



Branch Speed Up :

early target address / condition evaluation

- Assume each instruction is Branch
- Generate target address while decoding
- If target in same page omit translation
- After decoding discard target address if not Branch
- Evaluate condition as early as possible

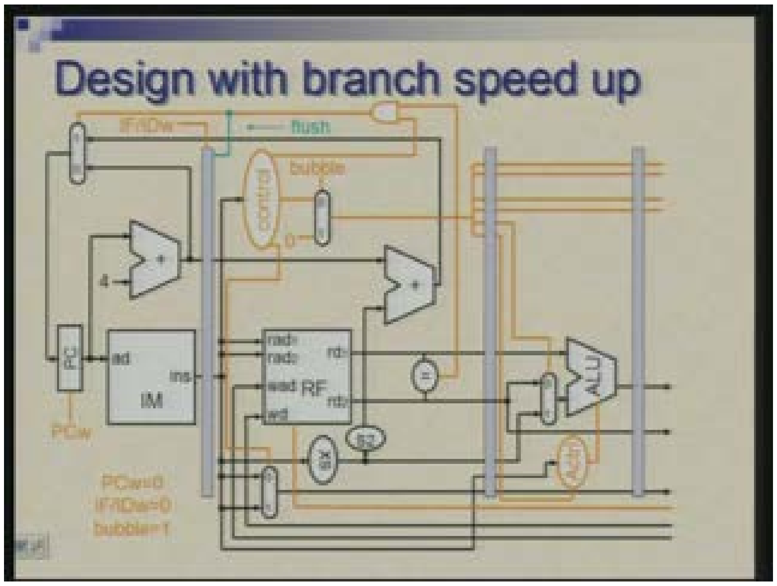
How do you speed up execution of branch?
Now you can speed up; target is calculation and condition evaluation. So for calculating the target address early what you can assume is that each instruction is a branch instruction and then generate target address. What may be done normally is that you may test, you may check, you may decode the instruction, find out if it is branch instruction then do the target address calculation because now you know that it is a branch instruction. But we can calculate target address in anticipation. Actually in the design we have done this is already happening.

If you recall, in the second cycle when you are fetching the registers A and B we are also calculating the target address and we keep it in a register and use it if necessary. So this technique we have not mentioned explicitly but we have already used this. Situation could be more complex if virtual memory was involved. So address calculation also involves page table look up and doing that translation from virtual address to real address. So it is not simply address calculation does not simply involve making doing an addition but it may involve something much more.

So, starting early in anticipation like this is definitely advantageous and you can also omit page (Refer Slide Time: 19:16) this translation if the target address is in same page in which you have the current instruction so all that check could be performed early and if the instruction is not found to be branch instruction you can discard this so no harm done.

Now secondly, condition evaluation. Again you try to move this as early as possible in the sequence of cycles and we will see an example of that. So I am showing a part of the design; last is I have just stretch things horizontally and the last stages have been thrown out just to make space.

(Refer Slide Time: 19:53)



What we have done is instead of checking for equality in the ALU we have introduced a comparator in the second stage itself in the second cycle itself and also the target address calculation which was actually happening here in the third cycle; although in the multi cycle design we had described earlier we had done this in the second cycle because ALU was free at that time so our motivation there for doing that in the second cycle was that utilize the ALU while it is free. But now we know that it has advantage it is advantageous if we do all these as early as possible. So I am moving that adder here I am moving and introducing a comparator here (Refer Slide Time: 20:55).

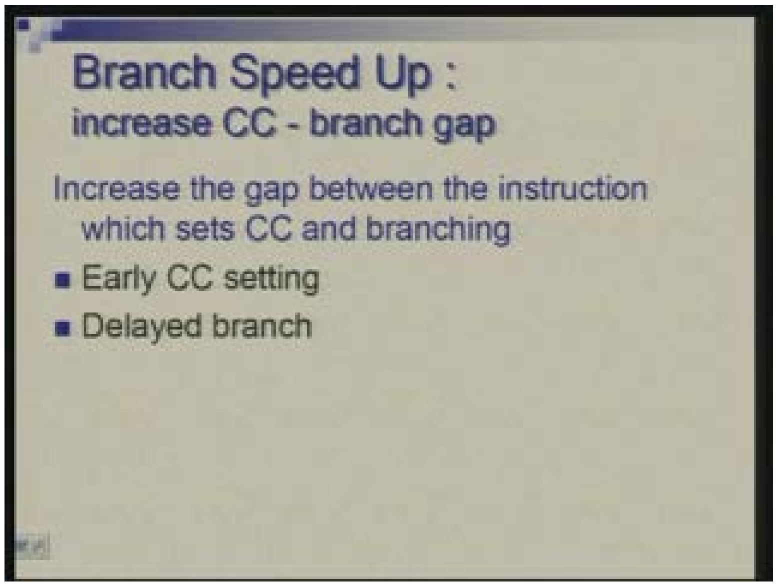
Now what is the implication of this in terms of clock cycle?
It is that this path may not be making so much of difference because whether it is here or there it is just coming in series with AND gate and multiplexer these do not cause any delay but the main effect would be that the delay of register file and the delay of this comparator they are coming in series within a clock cycle so it will definitely have some adverse effect on the clock cycle.

Now remember that testing for equality is much simpler as compared to testing for less than or greater than. In this case you need to do bit by bit comparison and then take AND of all those; you do not need any carry to be propagated. So it may be possible to afford equality or inequality comparison; less than greater than comparison may still be done within ALU the fast ALU which is sitting in third cycle so at least beq bna kind of instruction can be speeded up by doing this. So we are putting a simpler comparison here comparator here which will add to the delay but only marginally.

Now with this done we are ready to get the target instruction in the third cycle itself so we need to only flush this one we do not need to flush instruction which is going into this. So this is a this is a design with slight improvement where we are losing one cycle basically when branch actually occurs instead of two cycles.

Now, another way of speeding up is to increase the gap between condition code evaluation and actual branching.

(Refer Slide Time: 00:23:03)



Branch Speed Up :
increase CC - branch gap

Increase the gap between the instruction which sets CC and branching

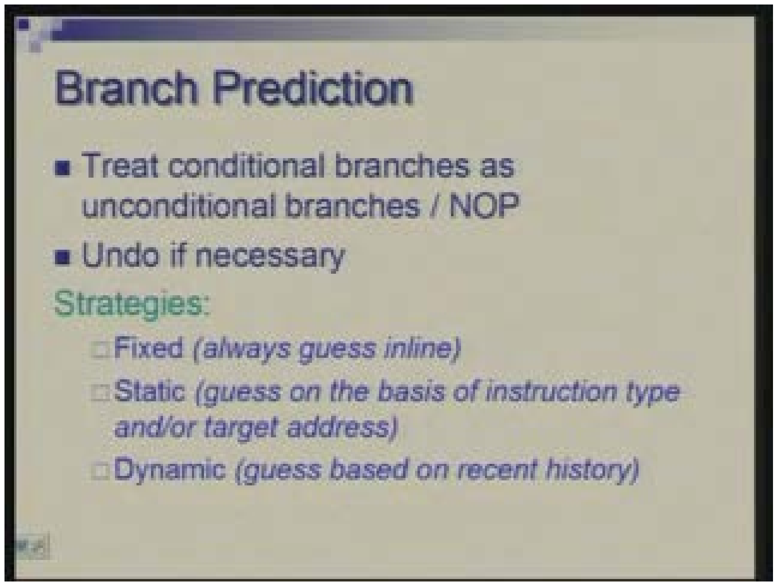
- Early CC setting
- Delayed branch

Now imagine that you were having separate instructions to evaluate condition and the branch instruction was only looking at some flags so flag can be easily looked at in a cycle as soon as you have the instruction. So what can be done is that you can increase the gap between the instruction which is actually setting the condition code and the instruction which is testing so that the situation is somewhat similar to data hazard situation that one instruction is dependent upon the previous one. So here one instruction is setting cc the condition code, other instruction is testing and if you increase the gap between these that means have other useful instructions in between then the waiting is cut down. So, as soon as the branch instruction can calculate its target address you can branch; condition testing is done early.

The other approach was to do delayed branch. That means do the target address calculation and assume that branch has to be effective whether it is true or false after a few cycles. So let us just say that you want to delay by one instruction. That means after a branch instruction there will always be an unconditional instruction which has to be done whether condition is true or false. So the role of compiler or the code generator is that it tries to find suitable independent instruction which has to be done under both conditions and keep it after the branch instruction and the role of the hardware here is that you take a decision about branching, well, you take a decision but actually make branch effective one cycle later.

Next is the technique of branch prediction where effectively we try to treat branches either as unconditional branches that means do not worry about the condition and try to branch or as no operation that means do not branch but continue sequentially and when you know whether you have gone wrong or if necessary you can undo.

(Refer Slide Time: 00:25:16)



Branch Prediction

- Treat conditional branches as unconditional branches / NOP
- Undo if necessary

Strategies:

- Fixed *(always guess inline)*
- Static *(guess on the basis of instruction type and/or target address)*
- Dynamic *(guess based on recent history)*

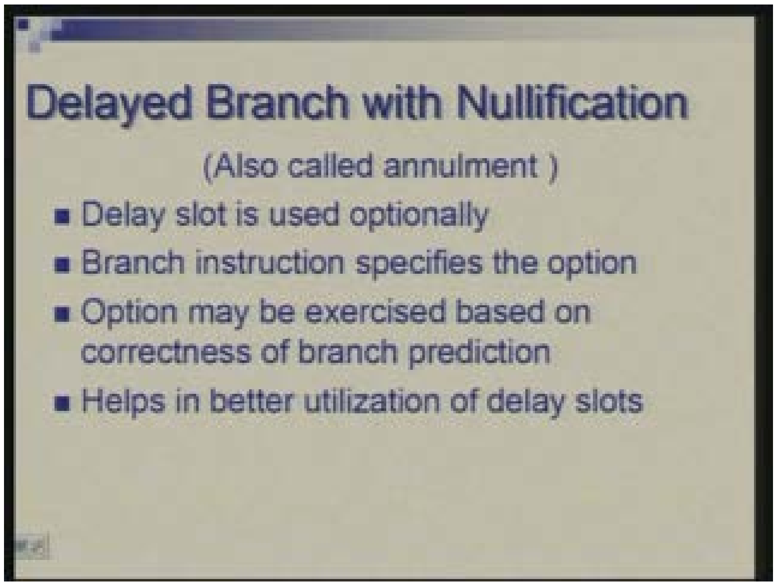
Now this is the basic idea. Question is how do you do the prediction?
There are three ways of doing it: one is fixed prediction that you always guess inline so effectively this is what we were trying to do; we allow the pipeline to get subsequent instructions into the pipeline and if you find that branch condition is true and branch is to be taken then you undo. So this is called fixed prediction.

Second is static prediction. Here you make choice between going inline or going for the target by using some criteria and the criteria is fixed it does not depend upon situation which is created at the run time you should know ahead of time whether for this particular branch you need to predict inline or you need to predict the target. So the basis of such prediction could be opcode or the target address; you might say that some kind of branches are more likely to be taken and some kind of branches are less likely to be taken

Or it could be based on the target address. For example, if you are branching to an address which is earlier that is branch back which typically indicates and of a loop then you may predict that yes branch is more likely to be taken because loops are often iterated several times. Otherwise if it is a forward jump you might say that it is either equally likely or less likely to be taken; it is an exception condition normally you go through and the condition checking is to take care of situations which are occurring less occasionally and less likely. So depending upon any such feature you can decide.

The dynamic branch prediction tries to take into account what has happened when the program has been executing. So there you keep a record of what happened in the past that if there was a branch instruction which occurs several times inside a loop then you see that on previous occurrences what happened to this; whether condition was true or false. So a simple minded approach there would be to think that the pattern which was there last time is going to repeat so you could just look at the last one and say that this is what is going to happen next time or a more sophisticated decision would look at last several occurrences and based on that you may decide.

(Refer Slide Time: 00:28:39)



Delayed Branch with Nullification

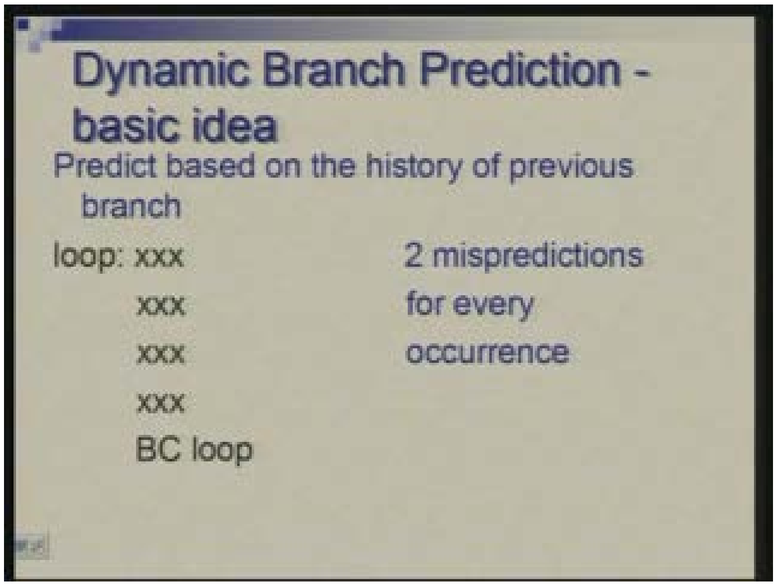
(Also called annulment)

- Delay slot is used optionally
- Branch instruction specifies the option
- Option may be exercised based on correctness of branch prediction
- Helps in better utilization of delay slots

When you look at branch prediction and delay slots then together you can make the design little more intelligent by giving the programmer flexibility of selectively bypassing the delay slots so that is called delayed branch with nullification or annulment. So the delay slot is not something which is fixed by the hardware it is possible and the programmer has the choice to use it or not to use it. With the branch instruction you could have another field which will indicate that you have to enable the delay slot or you do not have to enable. It would depend upon how you are predicting. Depending upon your prediction you may actually exercise this option of using a delay slot or not using the delay slot.

A simple branch prediction which is based on just looking at the last occurrence of the same branch may sometime work poorly whereas some very straightforward logic can do better. So just look at a simple loop like this that you have a few instructions and at that end you have instructions which loops back.

(Refer Slide Time: 00:29:55)



Dynamic Branch Prediction - basic idea

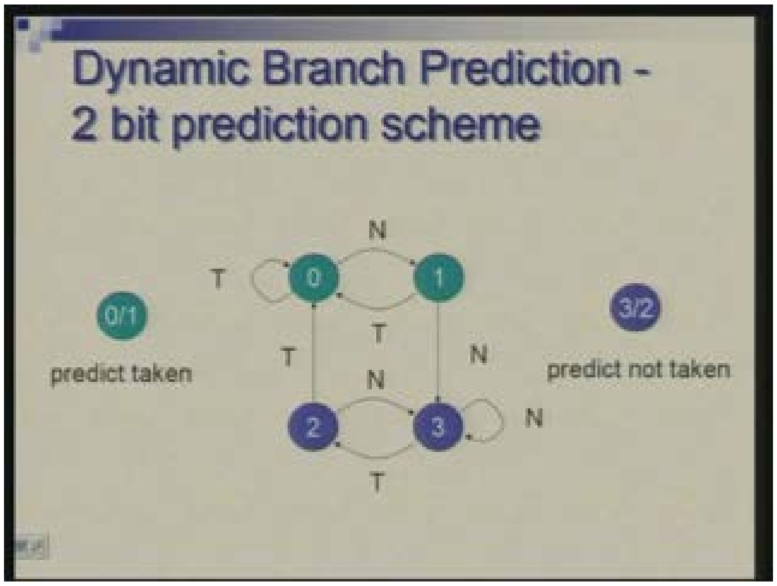
Predict based on the history of previous branch

```
loop: xxx          2 mispredictions
      xxx          for every
      xxx          occurrence
      xxx
      BC loop
```

So now suppose this itself is in a larger loop an outer loop where every time you come here you do a few iterations and then come out then you will come again here do few iterations and then come out. So now suppose your strategy is to predict based on the previous outcome. So, if last time we had last time the condition had become true you think that it will be true this time and vice versa. So, with every instance of the loop you will have two mispredictions that is when you enter the loop first time when you come to this for the first time in some loop instance you will find that last time the condition was false you will think it is false again but now it is the beginning of the loop so you will iterate you might go wrong at this point and last time when the condition becomes false you might still think loop is going to continue and you will make a mistake. So twice you will make a mistake: when you are entering the loop and when you are exiting the loop.

But on the other hand, if you had chosen a static branch policy here saying that always you predict that loop will be taken or the branch will be taken you will make one mistake per loop. So apparently the dynamic prediction strategy does worse than the static prediction strategy in this case. To make branch dynamic branch prediction more effective we need to modify our dynamic prediction strategy and one possible scheme which is commonly used is shown here.

(Refer Slide Time: 00:31:54)


Dynamic Branch Prediction - 2 bit prediction scheme

So instead of just remembering what happened in the last branch you try to remember little more. Here I am showing a scheme where you have where you actually imagine that there is a state machine which can be in one of the four states and these states can be represented by 2 bits. So instead of remembering 1-bit information you are remembering effectively 2 bits of information. What this machine remembers is some summary of last several outcomes. What we are trying to do here is that you do not change your you do not change your decision just by looking at one change. So, for example, suppose you are in state 0 and you are predicting that branch is taken; suppose branch keeps on getting taken you remain in that state and you keep on predicting that branch is taken. At some point branch does not get taken so N means branch is not taken, T means branch is taken.

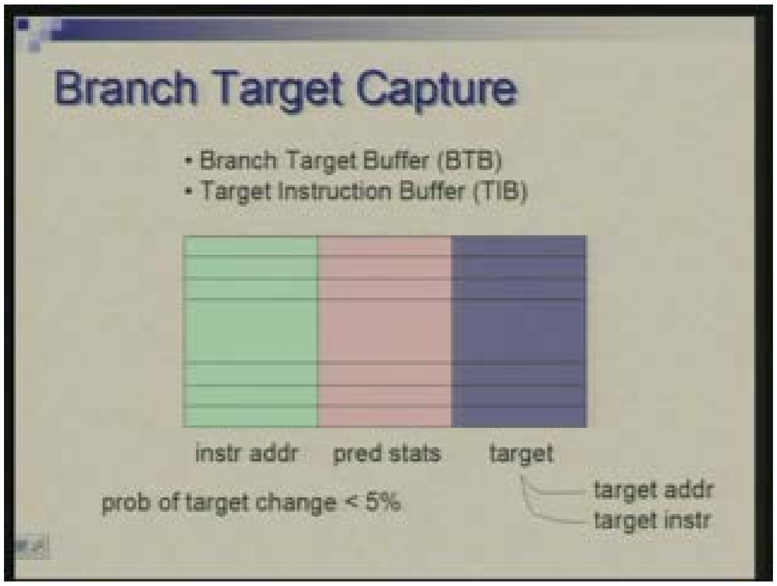
Now these arcs (Refer Slide Time: 33:10) are indicating how you are transiting the state; the label on the arc indicates what was the actual outcome and based on the actual outcome you are going to some next state. While you are in state 0 and 1 you predict that branch is taken, while you are in state 2 and 3 you predict that branch is not taken. So the situation I am trying to depict here is that when you are comfortably in this state and continuously branch is being taken and you are predicting that branch is being taken if once the condition becomes false you are not taking the branch you are still in a state where you will continue to predict that branch is taken unless you get another N and then you go to a different state.

So in 1 also you will predict that branch is taken and if it turns out to be true you get back to 0. If branch is not taken then you go to state 3 where you now start predicting that branch is not taken.

So the idea here is that in a situation like what I just depicted with a single loop you will actually continue in state 0 or 1 you will not come to state 2 and 3. So this is a general mechanism which can be used for dynamic branch prediction and you will avoid making double mistakes when there is a simple loop.

Lastly we are trying not only to know the decision early either as anticipation or by actual computation; we also want target address to be available to us as early as possible.

(Refer Slide Time: 00:34:57)



Branch Target Capture

- Branch Target Buffer (BTB)
- Target Instruction Buffer (TIB)

So if you are keeping the history; if you are looking at what happened in the past we can also recall what was the target address in the previous occurrence of the same branch and if that is available in a buffer we can simply pick up the address from there rather than calculating. So, actually speaking the address calculation will still occur just to ensure that what you have picked up from the buffer is same as what you actually get and if the two are same it is fine otherwise you will try to undo what was done in anticipation.

So here the assumption is that the address is not going to change from one occurrence of the branch to the next occurrence of same branch. There are situations where this may not hold. For example, if you take JR instruction, the branch address is coming from a register and you cannot be sure that what comes next time will be same. But if you take a jump instruction with a constant address or a beq type of instruction where the address is obtained by adding a constant to the program counter so none of these two things are changing and therefore the address next time is bound to be the same. So keeping in mind that JR will occur much less often than beq, bne and J instructions this scheme will work.

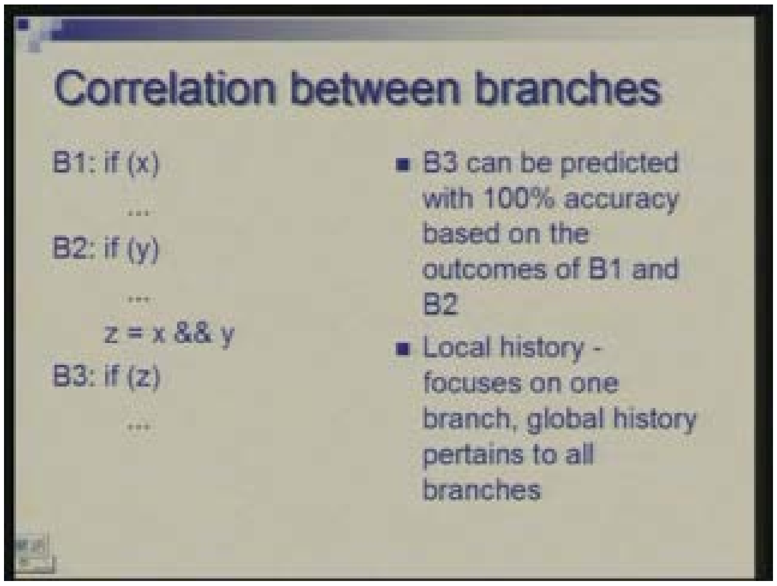
So here is the picture (Refer Slide Time: 36:35) of how this buffer could be organized; this is one way there are many different organizations. It could be organized as an associative or contented decibel memory where you look at the current instruction address and each word here has three fields: one field carries instruction address; second field will carry prediction statistics like those 2 bits of the finite state machine and the target address. So, given the address of the current instruction which is the branch instruction you try to look up in this table; looking up in this table is that in parallel it will search and try to match that address with all the addresses in the first field, if any match occurs you pick up the prediction statistics and you pick up the target address.

Again here there is a variation possible that you may store target address or you may store target instruction directly. So instruction to which you are branching can be directly fetched from there so you go a step further here. The only thing you need to keep in mind is that if you have target instruction here the instruction following that is not available here so for that you need to calculate the address but it is assumed that it will be calculated in the due course. So, in due course of time instruction branch instruction carries out all that it has to do; all these things we are doing in anticipation and we of course have to make sure that what we anticipated or what we predicted or really correct. But it allows us a way of doing it early most of the time.

Typically, for example, probability of changing of target could be less than 5 percent and therefore at least 95 percent of the time you are doing it fast and you are doing it correctly.

Now, so far we assumed that we look at repeated occurrences of same branch. So you have branch somewhere in the program and because of loop you are coming to this again and again. Sometimes there is some coupling or some correlation between various branches.

(Refer Slide Time: 00:38:51)



Correlation between branches

```
B1: if (x)
      ...
B2: if (y)
      ...
    z = x && y
B3: if (z)
      ...
```

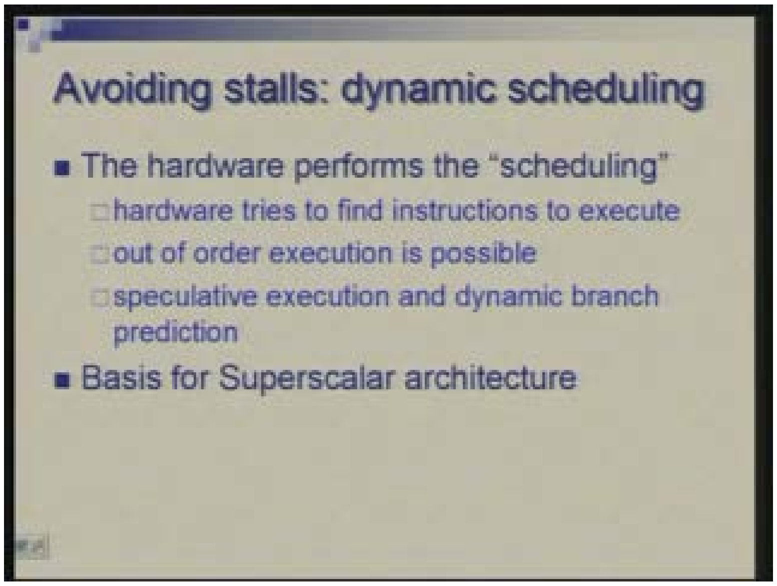
- B3 can be predicted with 100% accuracy based on the outcomes of B1 and B2
- Local history - focuses on one branch, global history pertains to all branches

For example, look at this sequence on the left; you are testing some condition here, you are testing some other condition there then you are looking at something which is dependent on both these. So suppose z which is being tested here was x and y so now if you once you have gone through this and you have gone through this and the value of x and y have not changed in this course then it varies it to predict what z may be because you have evaluated x and y; if both are true for example then you can predict that the branch will be taken here.

So B3 can be predicted with 100 percent accuracy on the basis of outcomes of B1 and B2. So what this means is that there is there may be often this is a very simple case but in general what this points out to is that there could be some correlation between various branches. So, if you are looking at not just the history of this branch but global history that means history of branches which have recently occurred in time there may not be recent occurrences of just the same branch but other branches.

For example, if you encode outcome of this branch by 0,1; outcome of this branch by 0,1 (Refer Slide Time: 40:24) you look at the string of 1s and 0s which represent last few branches and looking at that pattern it may be possible to predict with much more accuracy the outcome of the branch you have currently at hand. So, by going for global history you can improve your prediction. Of course more and more of these things you do you are incurring cost somewhere; you are putting in more and more control hardware to carry out these things and also more hardware to undo the effect of wrong decisions. So, finally one very powerful method of trying to avoid stalls whether they are due to data hazards or branch hazards is to do what is called dynamic scheduling.

(Refer Slide Time: 00:41:20)



Avoiding stalls: dynamic scheduling

- The hardware performs the "scheduling"
  - hardware tries to find instructions to execute
  - out of order execution is possible
  - speculative execution and dynamic branch prediction
- Basis for Superscalar architecture

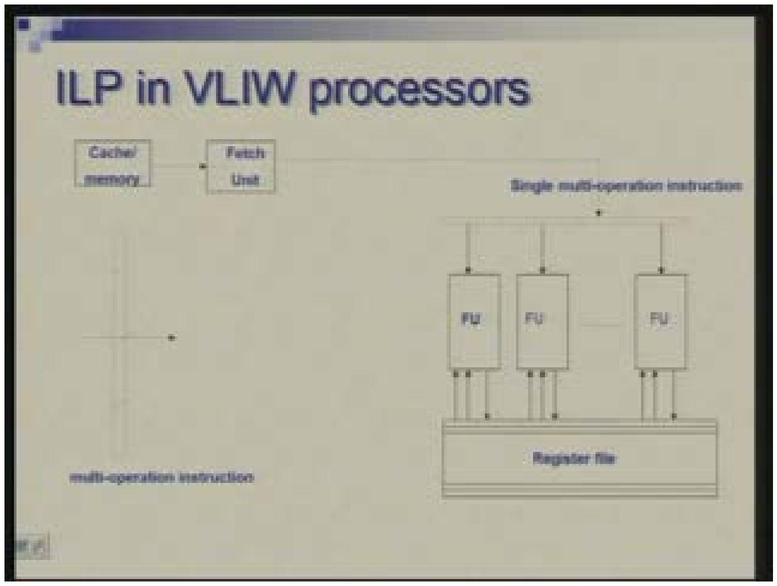
Instead of putting the instruction in the pipeline in the order in which they are occurring you try to analyze the instruction and push those instructions against the pipeline which can go through without causing any stalls so that is dynamic scheduling. What we are doing now, for example, when we are doing data forwarding we were trying to check the dependency between instructions and trying to take correct action.

On the other hand, similar mechanism could be built in hardware which will if one instruction is producing result which is to be used by next instruction the next instruction could be pushed in pipeline a little later; you can put something else in the pipeline. Some of these things can be done by compiler also but compiler does not have complete picture of what is happening dynamically. So, dynamic pipelining is an expensive thing; you need to have lot of extra hardware but it tries to find instructions which can keep the pipeline busy as far as possible; if necessary it can change the order of instruction; out of order instruction and it can nicely support speculative execution and dynamic branch prediction the kind of stuff we have been talking of. This also forms the basis of what is called superscalar architecture.

In superscalar architecture you try to fetch, decode and execute several instructions at the same time and the pipeline there has to be necessarily a dynamic pipeline where you look at a window of instructions in your stream of instructions and pick up something which can be pushed in the pipeline so the same idea of dynamic pipeline actually is getting extended to multiple instruction and this is called instruction level parallelism. You are not talking of parallel parallelism in terms of multiple processors doing multiple instructions but same processor is capable of fetching and decoding and executing several instructions at a time. The key here is the hardware which looks at a set of instructions and finds out which instruction can be initiated. Of course the hardware which executes these instructions in parallel also has to make sure that the results are obtained consistently.

Coming to ILP or Instruction Level Parallelism there is a different approach to this also that is we rely entirely on compiler to identify what instructions can be done together in parallel and do not leave this worry to the hardware.

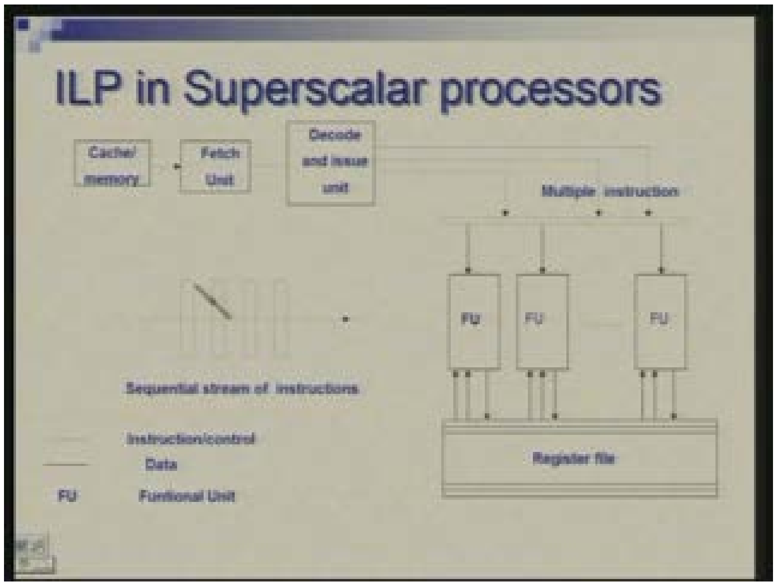
(Refer Slide Time: 00:44:06)



Here each instruction could possibly be carrying multiple operations. It is like a long instruction word where you have coded multiple instructions put them together so that is where the term VLIW comes into picture Very Large Instruction Word it is the meaning of VLIW. So whether we are having a superscalar architecture with dynamic scheduling or a compiler driven VLIW approach the basic thing is that you have multiple functional units so multiple ALUs which can handle multiple instructions so that is the basic requirement and all these have to access a register file so register file also must support multiple read write ports so that all these can actually do in parallel.

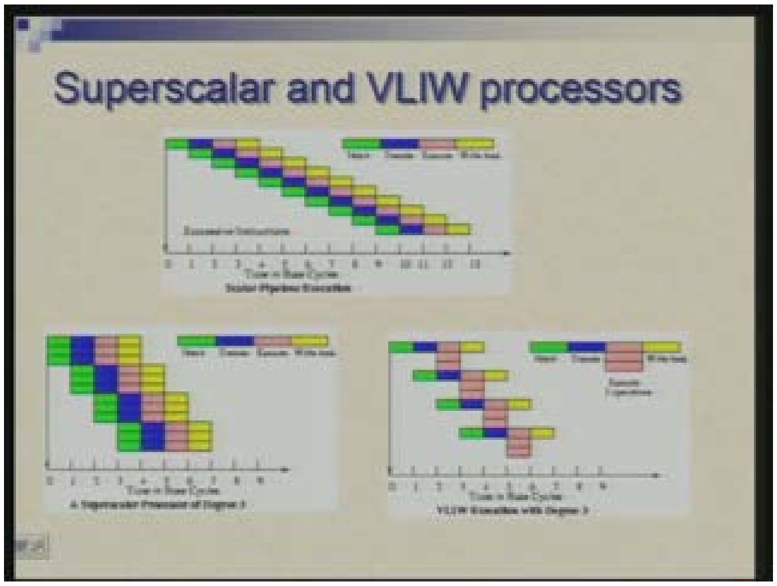
Between VLIW and superscalar the difference will be the way instructions are fetched and pushed in the pipeline. In case of VLIW we will expect that compiler forms long instructions carrying multiple operations and the hardware will simply take them one by one and execute them. On the other hand, superscalar architecture will have a complicated decode and issue unit which will look at many instructions which are fetched simultaneously; pick up the one out of these and then assign to different functional units to do them in parallel.

(Refer Slide Time: 46:14)


ILP in Superscalar processors

Here we are showing that multiple instructions are there but they are in terms of stream of instructions where they are following each other. Each instruction as you see in the program is like a scalar instruction but in case of VLIW each instruction is a special instruction which has many operational fields.

(Refer Slide Time: 00:46:30)



Superscalar and VLIW processors

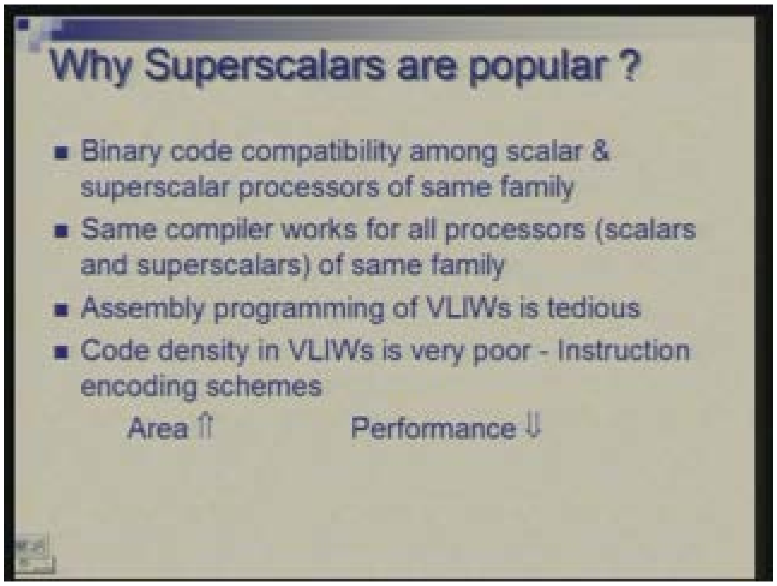
Scalar Pipeline Execution
A Superscalar Processor of Degree 3
VLIW Execution with Degree 3

So, if you look at the timings of these two alternative architecture versus timing of a simple pipeline this is how it will look like. This top picture shows a four stage pipeline; let us say the four stages are: instruction fetch, decode, execute and write back. Then in ideal case you have instructions which are overlapping in this manner. So at any given time you have up to four instructions in flight.

A superscalar, suppose again in ideal case that the degree of parallelism is 3 here so in every cycle it is fetching three instructions, decoding three instructions, executing three instructions, doing write back for three instructions. BIW, on the other hand, will look at these as single instruction being fetched, single instruction being decoded but each instruction will actually carry out multiple operations. So on the same scale it does three execute operations. This is how one could place three architecture in a common perspective.

Now, most of the modern processors are actually superscalar processors whether you take Pentium or power PC or IBs power 4 or PA-RISC and so on all these are high performance desktop computing machines; all are superscalar processors. VLIWs are used only in specialized embedded applications.

(Refer Slide Time: 00:48:24)



Why Superscalars are popular ?

- Binary code compatibility among scalar & superscalar processors of same family
- Same compiler works for all processors (scalars and superscalars) of same family
- Assembly programming of VLIWs is tedious
- Code density in VLIWs is very poor - Instruction encoding schemes

  Area ⇑ Performance ⇓

So question is why are superscalar popular for general purpose computing?
The main reason there is of binary code compatibility. If you take for example Intel's 486 processor and then Pentium a superscalar version of the same thing the code compatibility exists; the code which was available for older machines can be made to work on the new machines without any change in the code. Code can directly be taken from the old machine and run on the new machine in the same series as long as instruction set is same. The difference comes only here in the hardware.

As far as programmer or user is concerned it is just another machine doing executing the same code faster; architecturally it is different fundamentally whereas if you were to achieve speed through VLIW technique you would need to regenerate the code you have to have a specialized compiler which can detect parallelism at instruction level, pack the instructions appropriately and then you can run the code. So there could be source level compatibility but no object level compatibility. Actually this code compatibility from a commercial point of view is a very very major issue.

So these VLIW processors require very specialize compilers. It is almost impossible to code them by hand whereas if you can code normal scalar machine by hand you can code superscalar and there it does not require anything extra. Of course a compiler which is designed for superscalar processor would try to take into account some features and produce better code.

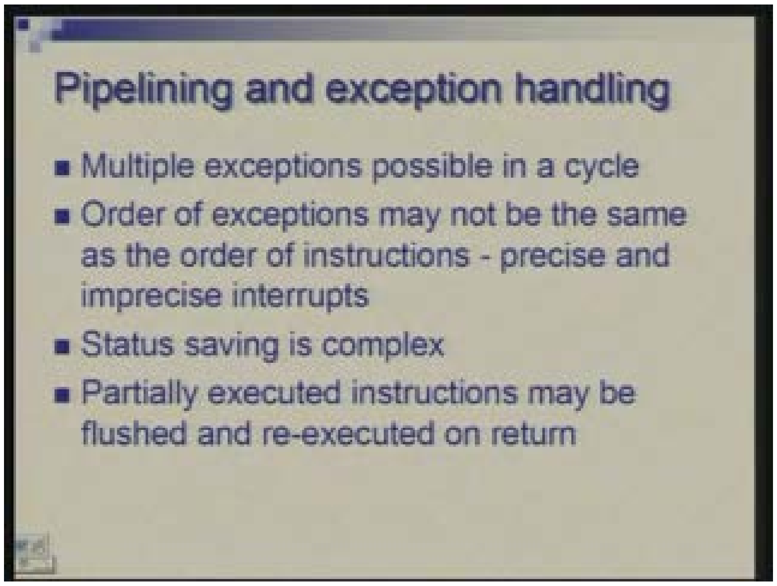
If you know that the machine is superscalar; how many functional units it has in parallel and how it will be better to keep the pipeline busy the compiler can produce better code but even the earlier code which is not generated specially for superscalar will still run correctly. Sometimes code density in VLIW can be very poor. You may not find many instructions to be packed together and they could be you have to fill them with nops.

In terms of area or the cost superscalars are more expensive. in terms of performance it is possible to achieve a higher degree of performance in VLWI technique provided you are able to have a good compiler. Finally before I conclude let me say few words about exception handling.

We discussed exceptional handling in case of multi cycle-design; how is it placed in case of pipeline design. In fact in pipeline design exception handling becomes more complex. The main problem is that since there are many instructions in processing you could have many exceptions which could be generated at the same time. In the same cycle you may find that one instruction is showing overflow, another instruction is showing page fault or illegal opcode. And knowing that different types of exceptions get detected in different cycles it is also possible that an instruction which is coming later may show an exception earlier.

So, for example, let us say the there was an instruction which results in overflow; overflow will be detected only when you have done the arithmetic operation whereas another instruction which is coming behind it has wrong opcode so immediately you will come to know. So it may happen that you may find one exception earlier which was actually supposed to occur later in the instruction sequence. This makes things very difficult and you have a concept of precise interrupt or imprecise interrupt. In case of Precise interrupt means that you detect interrupts or exception in the same order in which instructions are occurring. Somehow you have to time your detections so that the order in which you detect is not inconsistent with the order in which instructions are occurring

(Refer Slide Time: 00:53:08)



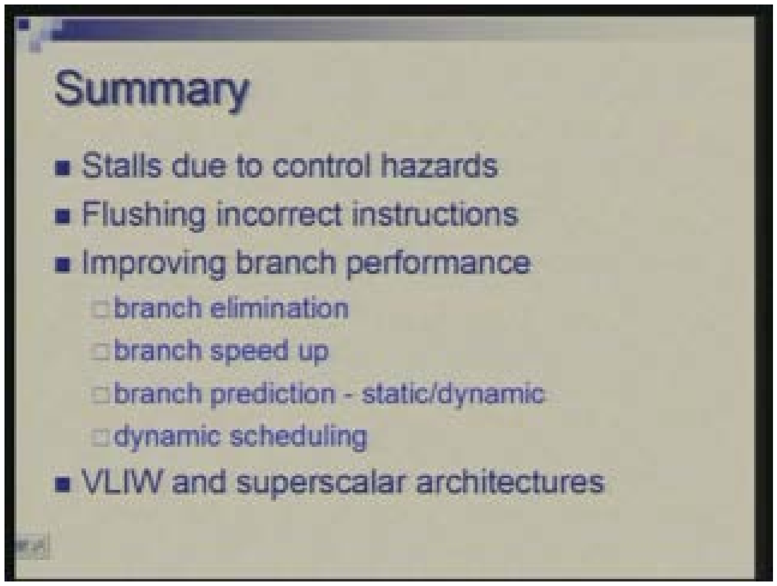
Pipelining and exception handling

- Multiple exceptions possible in a cycle
- Order of exceptions may not be the same as the order of instructions - precise and imprecise interrupts
- Status saving is complex
- Partially executed instructions may be flushed and re-executed on return

Some machines insist on precise interrupt; some do not worry about it and allow imprecise interrupts to happen. The saving of status is naturally very complex here because if you need to handle the exception and come back and resume the instruction you need to know how far the instructions were executed and with many instructions in flight each having done to some stage you need to remember all that that has happened so that you can continue.

some time what you do is that instruction which are partially executed you may simply flush them rather than trying to remember how far they have done you may simply flush them out and restart them all over again when you come back. So there are a lot of possibilities and we will leave this at this point realizing that there are many complexities involved in exceptional handling when there is pipelining.

(Refer Slide Time: 00:54:11)



Summary

- Stalls due to control hazards
- Flushing incorrect instructions
- Improving branch performance
  - branch elimination
  - branch speed up
  - branch prediction - static/dynamic
  - dynamic scheduling
- VLIW and superscalar architectures

Finally to summarize we looked at the stalls which occurred due to branch hazards and we saw how we can flush the instructions which may come up wrongly into the pipeline. We looked at several techniques to improve branch performance including branch elimination, branch speed up, branch prediction in a static or dynamic manner and dynamic scheduling. So, from dynamic scheduling it was led to superscalar and VLIW architecture which are basically instruction level parallel architectures and try to improve performance beyond CPI of 1. So pipelining ideally tries to make CPI as 1 but in real case it will be slightly worse than 1. To cross this CPI 1 barrier you need to do instruction level parallelism either in a VLIW manner or in a superscalar manner. I will stop at that, thank you.