

Lecture - 12

Basic pipelining, Branch Prediction

Life of an instruction

- Combinational implementation (single cycle)

The diagram illustrates the life of an instruction in a single-cycle combinational implementation of a processor. The instruction flow is divided into five stages by vertical dashed blue lines:

- IF (Instruction Fetch):** The instruction is fetched from Memory (Mem.) and loaded into the Instruction Register (IR). The Program Counter (P.C.) is incremented by 4 (indicated by a red arrow labeled '4') and loaded back into the P.C. register. The Clock (CLK) is shown as a blue vertical bar.
- ID/RF (Instruction Decode/Register File Access):** The instruction from the IR is decoded by the Ext. (Extension) block. The Register File (R.F.) is accessed to retrieve the register values specified in the instruction (indicated by red arrows labeled 'W' for write and 'R' for read).
- EX (Execute):** The register values are combined with immediate values from the Ext. block and fed into the ALU (Arithmetic Logic Unit). The ALU performs the operation specified by the instruction. The result is compared with a constant value (indicated by a red arrow labeled '=?') to determine if a branch should be taken.
- DMEM (Data Memory Access):** The ALU result is used to calculate the effective address for data memory access. The data is then read from or written to the Memory (Mem.) block.
- WB (Write Back):** The final result from the ALU or the data from the Memory is written back to the Register File (R.F.) to be stored in the destination register.

Red arrows indicate the flow of data and control signals throughout the stages. Purple arrows highlight specific data paths, such as the register values being read from the R.F. and the ALU result being compared and then used for memory access or write back.

At the bottom, the stages are labeled: IF, ID/RF, EX, DMEM, and WB. A small logo of the University of Technology is visible in the bottom left corner.

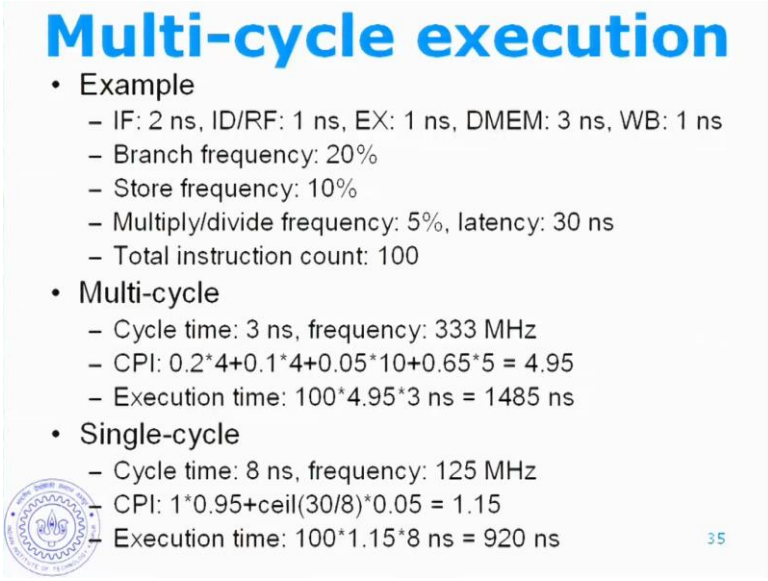
So, you talk about the multi tackling implementation were essentially what we do is you put a pipeline latches, these are not exactly pipeline latches, but these are latches. So, that you can carry the data from one side to the another. So, essentially what happens is that, in the first cycle you will do whatever is needed to be done in the instruction phase stage and instruction register will store the result of that. Next cycle only this particular stage will be active and doing the 0 unsigned extension, and then whichever wires cross these boundary will hold the values of this cycle, whatever is computed. Next cycle only this

stage will be active and the next cycle this stage will be active and the next cycle this stage will be active ok

So, essentially what we are doing is you are now having smaller cycle, but each instruction takes five cycles to execute. And the question arises which one is better the single cycle combinational implementation or multi cycle implementation right. So, so here is a example that we talked about last time. So, suppose you are... So, usually the stage is a account balance, because there are different things happening in stage. So, the time similarity. So, assume that instruction page takes two nano second and your decode register file the takes one nano second, execution takes one nano second, data memory takes 3 nano second and drive back takes one nano second. And let us assume that branch frequency is 20 percent. So, I mentioned this last time also that if you look at this one the branch instructions take four cycles to complete right. So, 1, 2, 3, 4 - these wire your PC ignore the next cycle comes in.

Similarly, store instructions take four cycles to complete, because in this stage your store gets improve the value comes in and the address comes in. So, we do not give five cycles for store everything else will require something to into the register file, we will require the fifth cycle.

(Refer Slide Time: 03:25)



Multi-cycle execution

- Example
 - IF: 2 ns, ID/RF: 1 ns, EX: 1 ns, DMEM: 3 ns, WB: 1 ns
 - Branch frequency: 20%
 - Store frequency: 10%
 - Multiply/divide frequency: 5%, latency: 30 ns
 - Total instruction count: 100
- Multi-cycle
 - Cycle time: 3 ns, frequency: 333 MHz
 - CPI: $0.2 \times 4 + 0.1 \times 4 + 0.05 \times 10 + 0.65 \times 5 = 4.95$
 - Execution time: $100 \times 4.95 \times 3 \text{ ns} = 1485 \text{ ns}$
- Single-cycle
 - Cycle time: 8 ns, frequency: 125 MHz
 - CPI: $1 \times 0.95 + \text{ceil}(30/8) \times 0.05 = 1.15$
 - Execution time: $100 \times 1.15 \times 8 \text{ ns} = 920 \text{ ns}$

So, with this code if your data we can compute, whatever we want to compute. So, a branch frequency 20 percent, store frequency 10 percent; assume that multiply divide

frequency is 5 percent that may take longer, that is 30 nano seconds alright. So, and total instruction count is 100 alright. So, given these particular configuration and given these particular program. So, of course, we are talking about a particular program, here were 20 percent branch store and so on.

We have to compute the multi cycle implementation with a single cycle implementation alright. So, let see how to do that. So, first thing that we calculate for multi cycle implementation is cycle time. So, it is a 3 nano second why is that? So, that is because the longest stage takes 3 nano second right. So, that is why. So, every stage has to be 3 nano second. So, that gives us a frequency of 333 mega hertz alright, and we can calculate CPI. So, branches take 4 cycles, as 20 percent stores take four cycle 10 percent multiply divide take 10 cycles right 3 nano second cycle 30 nano second alright 5 percent and whatever is left will take 5 cycles. So, that keeps a CPI 4.95 alright. So, as I said it should be close to five slightly below, because of store enlarge and as you can see because a multiply divide frequency. So, low it is highly does not much.

So, now you calculate execution time of this program which has 100 instructions. So, 100 times CPI times. So, that gives us 1485 nano seconds. Any question on calculation alright? So, what about the single cycle? So, a single cycle everything will compute the cycle right. So, you add all this things up you get 8 nano second, that is your cycle time that is gives us a frequency of 125 mega hertz. So, first thing will notice is that your multi cycle frequency is not really five times higher, that is the first thing to notice. So, if the ratio is between two and the reason is that as you say your cycle time this case determine by the longest stage that there you lose a lot. Like for example, your decode register file stay wait for two nano second doing nothing, because cycle times alright.

So, what the CPI for your single cycle implement time, you are pretty much everything remains unchanged. So, here everything takes a cycle right, there is even though branches compute early we have nothing to do actually, we cannot really exploit we have to wait till the cycle boundary way, because you block there is nothing really to do with the middle of a clock you can compute or wait for event to happen only at the clock boundary happen only at the clock boundary alright. So, 95 percent instructions take one cycle, these five percent just instructions at going to take longer and how long they will take? Well they request ceiling of 30 over eight cycles alright. So, that gives us the CPI of 1.15. So, you can notice that your multi cycle CPI is not really 5 times of that. So,

ratio depends on lot of this, you just cannot bind seven by multi cycle five times. So, what my execution time? 100 instructions, time CPI then cycle time 920 nano second. So, as I mentioned last time in most cases we should expect that your combination design will be better in terms of ((Refer Time: 07:22)). So, as such there is no reason to do this do a multi cycle implementation, you will do this will see actually compute examples on that, suddenly floating point a make multi cycle, because there you can actually say behavior here you say nothing, you newly perform we do a multi cycle alright is this calculation clear to everybody.

So, here is one more example which takes a balance pipeline. So, here we assume that all your stages take same time, we just see that case what happens right? Everything else is unchanged the only thing that we have changed is the stage two nano seconds alright. So, now a multi cycle cycle times two nano second frequency is 500 mega hertz alright, CPI. So, everything remains unchanged except this particular number is going to change, 30 nano second meets now cycles alright. So, that because 5.2. So, as you can see now actually your multiply divide actually ((Refer Time: 08:23)) it go beyond five and you can calculate with execution time which turns out to be 1040 nano second. What is happening in the single cycle site? My cycle time is 10 nano second alright, frequency is 100 mega hertz.

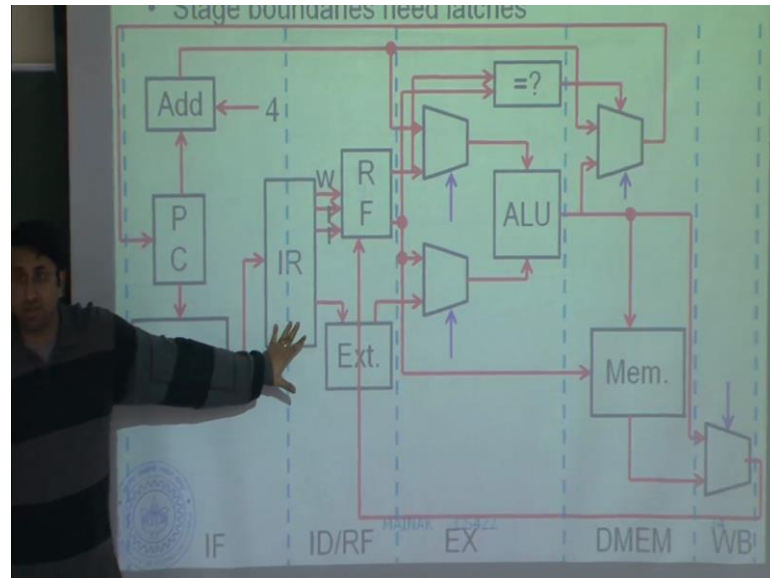
So, now I can see that the ratio is exactly five as you if you are balance five, you will get a five times slower clock in single cycle. You can calculate the CPI 95 percent instructions take one cycle, and 5 percent multiply divide will take 3 cycles right that gives us 1.1, again very close to 5; you can see that the ratio is almost close to 5 and as you calculate execution time you run slower whenever ((Refer Time: 09:09)). So, why where do we actually use in this case missing the cycle yeah.

Student: branching store yeah can you try to image they will run for less time they will become ((Refer Time: 09:25))

Exactly. So, here branch and store instructions to transfer 4 cycles, and how much is that eight times. Here they will be finished by nano second one hand to wait for two nano second, because events will occur only at cycle boundaries, that is why you lose to those two nano seconds. So, you can actually calculate that the come exactly from the this 60 nano second alright. So, it is clear to everybody. So, what happens in unbalanced pipe?

What happens in balanced pipe? No question? So, often multi cycle designs are used as an intermediate design before you go to pipeline alright. So, if we look at the multi cycle design carefully you will find that, in the second cycle I know it is a branch right.

(Refer Slide Time: 10:31)



So, I go back. So, here I decode the instruction in the second stage. So, I know there is a branch instruction.

(Refer Slide Time: 10:39)

Pipelining

- Observations
 - In the second cycle, I know if it is a branch; if not, start fetching the next instruction?
 - When the ALU is doing an addition (say), the decoder is sitting idle; can we use it for some other instruction?
 - In summary, exactly one phase is active at any point in time: wastes hardware resources
- Form a pipeline
 - Process five instructions in parallel
 - Each instruction is in a different stage of processing (called pipe stage)
 - How to synchronize between pipe stages?

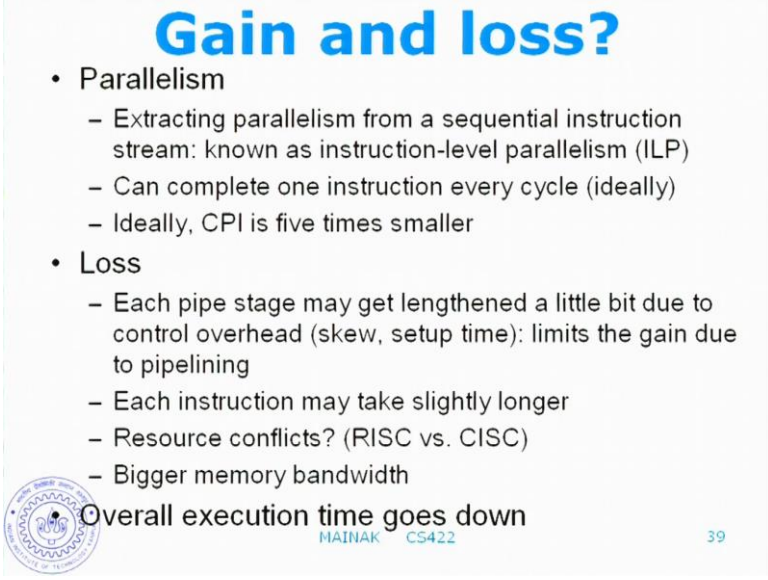
So, this case if I know that it is not a branch, then I can again start fetching the next instruction, because I know that it is the next instruction will be sequentially next right.

So, that is one option also when a new is doing an addition that say the decoder is actually before you say that in a multi cycle design first cycle you will fetch, next cycle you will decode the fetch will stay idle. The next cycle will be execution, the decode will stay idle and so on so forth right.

So, suddenly exactly one phase is active at any pointing time, it wastes the lot of hardware right. So, then from pipeline what you can do is you process five instructions in parallel, each instruction is in a different stage of processing called a pipe stage and how do you synchronize between five stages input pipeline latches right. So, this what ((Refer Time: 11:35)). So, now stage boundaries meet pipeline registers or latches alright. So, wherever you see red wire crossing a blue dotted line, you know that you need a pipeline register their alright. So, for example, here this is what? This is a pipeline register which contains instruction alright. Here for example, you hold the 0 or sign extended operand, here you have to hold the two operands coming out of the register file the pipeline register is 1.

Similarly, here you have to contain the comparison outcome, you have to hold the ALU outcome, you have to hold the stoke value coming out of the register file, and so on right. So, these are the pipeline registers and that is all you synchronize, whenever a clock ticks, ticks with move from here to here, ticks with move from here to here, ticks with move from here to here, ticks with move from here to there alright and so on. So, as I think notice is that although I have put the register file here, the register file right actually happens in this stage alright. So, fix that in mind, it just a physical design, but the actual operation happens here. So, we will pick up whichever value was register file and actually that will be happen in the cycle. So, these are pipelined mix processor, any question?


(Refer Slide Time: 13:07)



Gain and loss?

- Parallelism
 - Extracting parallelism from a sequential instruction stream: known as instruction-level parallelism (ILP)
 - Can complete one instruction every cycle (ideally)
 - Ideally, CPI is five times smaller
- Loss
 - Each pipe stage may get lengthened a little bit due to control overhead (skew, setup time): limits the gain due to pipelining
 - Each instruction may take slightly longer
 - Resource conflicts? (RISC vs. CISC)
 - Bigger memory bandwidth

Overall execution time goes down

 MAINAK CS422 39


So, what we gain what we lose we gain it answer parallelism, because we are extracting parallelism from a sequential instructions stream, this is known as instruction level parallelism, and this is pipelining is the simplest form of ILP, where you essentially form a pipeline of instructions and execute well try to execute five instructions in parallel. If you have a five stage pipeline, and ideally you should be able to complete one instruction every cycle. Ideally one instruction will enter into the pipeline every cycle and one should be... So, ideally your CPI should be five times one, smaller compare to a compare to compare to a single cycle compare to a single cycle or multi cycle design. If we assume that there are there are no multiplied by type of questions. CPI should draw by five times, because it would look like your finishing one instruction every cycle, because we observe into the pipeline one instruction will count every cycle.

What we will lose. So, each pipe stage may get lengthened a little bit due to control over it, for example this latches are not really ideal, they have something called skew time they have set of time. So, whenever the clock ticks the value will does not get transferred from input to output, it takes time alright. So, that limits the gain due to pipelining, because essentially now what is happening is that always shown in this example.

(Refer Slide Time: 14:50)

Multi-cycle execution

- Example
 - IF: 2 ns, ID/RF: 1 ns, EX: 1 ns, DMEM: 3 ns, WB: 1 ns
 - Branch frequency: 20%
 - Store frequency: 10%
 - Multiply/divide frequency: 5%, latency: 30 ns
 - Total instruction count: 100
- Multi-cycle
 - Cycle time: 3 ns, frequency: 333 MHz
 - CPI: $0.2 \times 4 + 0.1 \times 4 + 0.05 \times 10 + 0.65 \times 5 = 4.95$
 - Execution time: $100 \times 4.95 \times 3 \text{ ns} = 1485 \text{ ns}$
- Single-cycle
 - Cycle time: 8 ns, frequency: 125 MHz
 - CPI: $1 \times 0.95 + \text{ceil}(30/8) \times 0.05 = 1.15$
 - Execution time: $100 \times 1.15 \times 8 \text{ ns} = 920 \text{ ns}$

35

Your pipeline latency may not be exactly nano second any more, it will be slightly more than that alright. So, that is one problem. So, each instruction may take slightly longer. So, essentially that will effect to CPI. This will really not be five times smaller would there be resource conflicts. So, that is one problem like, here you can see that my register file is going to be needed in two stages right, here annual read from register file, here annual write to the register file. And now both of the stages are going to be concurrently executive right, one instruction will be reading from register file from other instruction you will be trying to write to a register file right. So, there will be resource conflict process that you have to resolve ((Refer Time: 15:37)).

Similar problem which is memory here right, instruction fetch you require accessing memory and your data memory load you require, and this is where the risc versus cisc debate becomes complicated, because in risc if you look at instruction architecture they are very clear about what stage should require, what resource is very clear. If you look at the Isa, you can figure it out immediately in cisc its not at all clear when looking at instruction, it will be very difficult to figure out what all resource is the instruction will require to execute, it may require five instruction accesses to complete an instruction. So, you maybe actually you really do not know how many times you require in the memory resource to complete that instruction.

So, that is that because very difficult. So, that will rise is much easier to pipeline in this size, you require bigger memory bandwidth, now because you will have to access memory twice every cycle one for instruction; one for data right, because they will happening concurrently. So, over all execution time goes down, that is so over all benefit even though we have these short cuttings alright, any question?

So, pipelining has one major problem and these are called pipeline hazards, and you have already looked at some up when you talking about simple pipelines that. So, that will be two types of hazards right; one is that for you to get a input you really do not know what written on that input alright, we have seen one example of that you might have forgotten, but you can review that in slides. And the second problem was that you want to do a computation on the data maybe not available in time. So, they are two different types of hazards that can happen.

So, if you look at this slide you can actually figure out what kind of hazards to expect here. So, one problem as you can see here is, the first kind that is we do not know what to do that involves a program counter, because your program counter gets updated here. But if it is a branch instruction right, but you have fetch the instruction already here. The next instruction to be fetched should be here should be happening here, but we really do not know what to fetch, because of program counter is not yet updated alright. So, that is called a control hazard alright.

And the second problem arises, because suppose you are having an instruction which produces a value. So, it tries to say register fetch alright, and the very next instruction raise from register 20. So, first instruction will naturally write to register 20 here, right in this particular cycle, but the next, but the next instruction will be reading the register file in this cycle right.

(Refer Slide Time: 18:45)



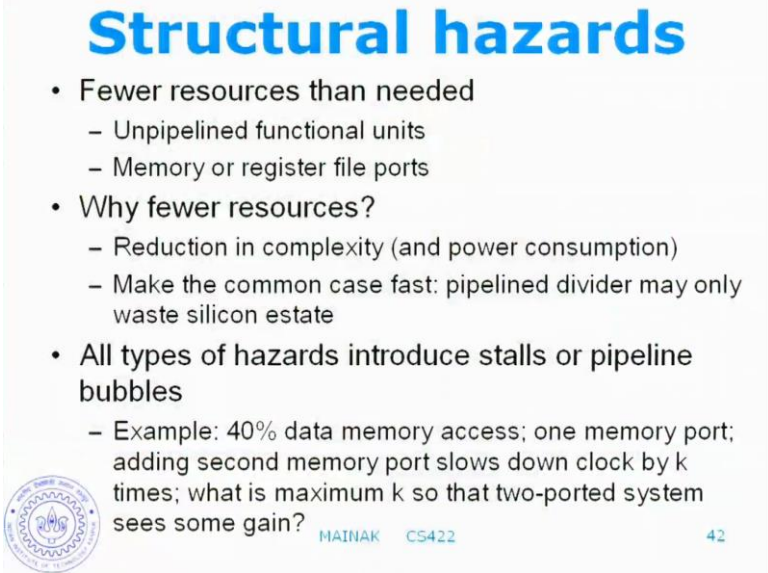
So, if you look at the timing of these two instructions. So, time goes in this direction. So, these are my clock cycles right. So, first instruction that produces register 20 is here, this is the instruction. The next instruction that reads register 20 will be reading from register file here, but this is written only here. So, it is going to get a wrong value from register ((Refer Time: 19:23)) alright is that clear to everybody. So, that is a data hazard. So, we have to do something. We have heard this problem. The first problem is easier to solve with a sense that the solutions are easier, because that well branches are not every instruction. So, I can wait whenever I see a branch, but this is very frequent back to back register uses. I produce the value you continue with the next instruction. So, if you here say that well unknown delay this register read, tick this point that is correct, do not take a very, very large performance loss alright, is it clear? These two types of hazards.

And there is a third type which is called structural hazard that arises due to resource complex, it happens in the same resource access till at least two stages of the pipe like, we all talked about register file or the memory control hazards problem should branches a branch does not resolve immediately after it fetched. So, what to fetch in the next cycle we do not know defines an important parameter called branch, we will talk about that that is how many cycles do you use, if you choose not to not to know not to go ahead and wait until the branch is or will be something that is wrong. So, will define this particular called a branch... Third point is data hazards dependent instructions may not execute back to back if dependence does not resolve. So, what you get is speedup of pipeline is

pipeline depth over one plus stall cycles per instruction, because of this hazards you may have to do stall cycles. So, essentially what happens is that, you really do not get speed of a pipeline equal to pipeline depth, you can divide by certain other over.


So, first let us look at structural hazard, because as such its boring in the sense that there is no smart solution to it. Usually structural hazards are resolved by storing the more resources. So, if you have fewer resources that needed then you normally have structural hazard. For example, if you have an un pipelined function unit that essentially what has means is that if the function unit takes 20 nano second to complete one operation, the next operation cannot be started on that functional unit before 20 nano seconds.

(Refer Slide Time: 21:58)



Structural hazards

- Fewer resources than needed
 - Unpipelined functional units
 - Memory or register file ports
- Why fewer resources?
 - Reduction in complexity (and power consumption)
 - Make the common case fast: pipelined divider may only waste silicon estate
- All types of hazards introduce stalls or pipeline bubbles
 - Example: 40% data memory access; one memory port; adding second memory port slows down clock by k times; what is maximum k so that two-ported system sees some gain?

 MAINAK CS422 42

So essentially every 20 nano seconds, you can start one operation; that is one type of structural hazard, because essentially what I am saying is that even if I have an operation ready to go we cannot send it, because I do not have function units or rather than if I have two of these things I could have you should two such operations in parallel alright. So, that is why it is a structural hazards. And we already talked about these two things memory or register file source, because as I mention we need to read from register file and write to register file. So, unless we have separate read and write ports there is no way resolve this structural hazard

So, essentially what I am saying is that you through in words sources to the ((Refer Time: 22:42)). So, the question now is that well its sounds funny right that we are saying

that well, I deliberately have fewer resources that I need because I know that I need. So, much, but why should I have that fewer resources right. So, why what is the reason? Well, the primary reason is the reduction complex, right. So, you may say that well, I could have gotten rid of touch completely if I had 100, sure but would you have a hundred to hundred not, because the simple reason is that may be once in a while you require 100, but not all it unnecessarily increases your complexity and also increases the power consumption. So, that is why you normally design a processor for the common case as you said. So, loss says right, and in the rare cases you resolve to some other way.

So, make the common case first pipeline divider may only waste. So, divider set on pipeline, even though you know that if you do not pipeline divided. So, get a long... So, if we have two division operation sign back to back, second one we have to wait alright, but you want do this, because that is not a common case. So, look at the frequencies of such occurrences and decide what to do? So, all that so hazard introduces stalls and pipeline bubbles. So, here is an example, suppose your 40 percent data memory accesses in the program, and you have just one memory ((Refer Time: 24:25)), adding the second memory port slows down clock by eight times.

So, do not ask me why this happens? So, this has to be with how memory models are actually built, that why introduce in a new port slow down the memory body. Why should that? So, that has to do with the electrical properties of this particular systems how they are designed. So, I will gonna give details of that, but never the less this is true that we ports to a memory structure, it will slow down and here I am saying is that adding a second memory port slows down the clock by k times.

So, a question now is what is maximum k ? So, that two ported system is sees same gain, because now essentially have to right that I am going to pipeline a processor, I know this statistics. So, now the design has two options either you go with the single port, and memory system and you know take the stalls up that or you have two ported memory system, but actually design processor will slower clock. So, which one is better? So far that I need to know the break even right, what is the maximum k ? So, that two ported systems sees same gain, because then I can go back to your electrical engineer and say that well look can you design a memory module with this ratio k , then I will point. Otherwise if the answer is no, then I have rethink that you know, because I change the design. So, how would I calculate this if I have a single coated memory, what is going to

be my execution time for this program, you can assume a clock frequency of something x how do you calculate this?

See, if I have a single code whenever the data memory is in use, I should not be use that module right, because data instruction memory is same its memory is it? So, whenever I access data from memory, if I have single port I cannot access instruction right. So, what is the application of that what case effected because of that.

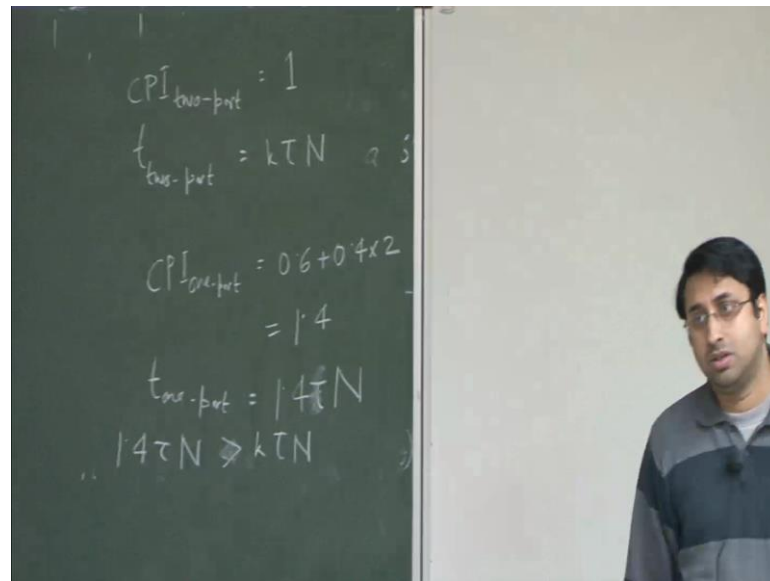
Student: ((Refer Time: 26:40))

(Refer Slide Time: 26:44)



So, if you yeah... So, here is my I can fetch here, I can fetch here, but I cannot fetch here right this has to wait. So, I can only fetch here right. So, can you this how do you this how do is going to be my execution timings. So, for every data memory access what I am adding in the execution time one cycle extra right.

(Refer Slide Time: 27:21)



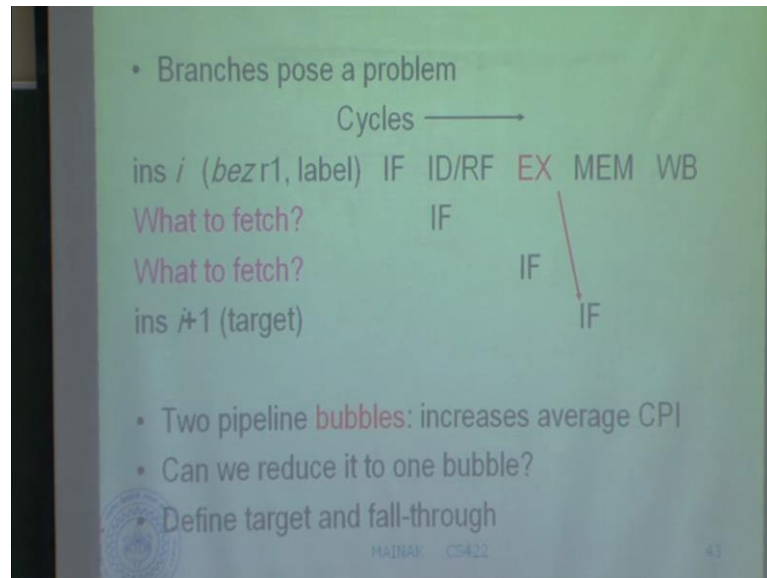
So, I can say that my CPI is going to be what? CPI one port 0.6 plus 0.4, it will be two right; can I do that for a pipelined implementation, that is a approximation right. So, how much is that 1.4 alright. So, my execution time one port, let us assume that I have just one instruction in this program I can normalize that, but if you want any instructions. So, that 1.4 with the frequency times number of instructions N all right, not frequency I am sorry cycle time. So, let us call it out 1.4 tau head alright is that clear to everybody I may get gross approximation, but is going to be more or less.

What about CPI two port? CPI two port, sorry...

Student: will be one.

One right. So, execution time two port is going to be what? My cycle time, sorry k tau head right. So, I want them to be equal right. So, or if I want my two ported system to be better, I want this right, oh I am sorry yeah right. So, k is less than 1.4. So, adding a port should not slow down my clock by more than 40 percent if it does then I cannot go to a two points any question for this. So, keep this in mind that every time you will be throwing some resource to resolve this structural hazard, it will do somewhere else. So, there is always a trade up in most of the most of the things that you come across in computer systems. Usually not a one way to wait loose somewhere in the trade of analysis, and figure out which one which wait should go.

(Refer Slide Time: 30:23)

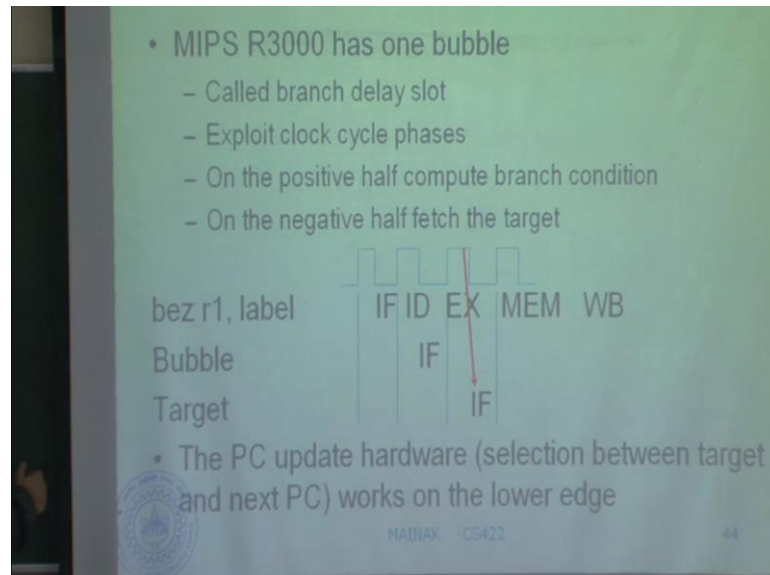


So, let us take a look at control hazard a little more careful. So, this is the problem, that is depicted here cycles go in this direction. So, these are my each cycle stage alright. So, here is a branch instruction which is which will be anything I am put in branch if equal to 0 right and that executes here. So, what we have shown in the diagram actually it resolves here to receive. So, let us assume that we can actually squeeze that multipliers are effects to stage. So, I talk about this particular multipliers, this one. So, let us assume that we can actually squeeze this into this particular cycle. So, at terms below over here. So, I can resolve my PC, the next PC in the problem is that I do not know what to fetch in these two cycles alright, because here the branch is not yet computed, here the branch is not yet computed actually is being computed. And here I know were to fetch now to bypass pass this, tells me that this is my, this is your PC to fetched out, is that clear to everybody.

So, these are called two pipeline bubbles, it increases with average CPI. If you do nothing, if you well and to just wait. So, that way what I saying is that for every branch instruction you are adding two cycle over edge, two extra cycles. So, now the question is can I really reduce it to one bubble, let us go on stay by the time right. So, instead of reducing both the bubbles together, let first ask this simpler question that. Can I solve get rid of one bubble all of this two. So, at this point we need couple of definitions actually Sudhanshu have already defined, these two terms target and fall through. So, in this case when the branch executes whatever label that appears here to do target, that is call a

target fall through is a next instruction. So, if the branch does not want to target it will fall through.

(Refer Slide Time: 32:41)



So, MIPS R3000 has one bubble. So, that is a pipelined MIPS processor very similar to the one way, the question is how do they actually manage to do this one bubble instead of two. So, essentially what they have done is. So, by the way this particular bubble that they had is called a branch delay slot. So, this just instruction just up to the branch which has the boundary. So, how do they are actually getting a one bubble. So, what they do is they actually exploit clock cycle phases; on the positive half they compute the branch condition, and the negative half they fetch the target.

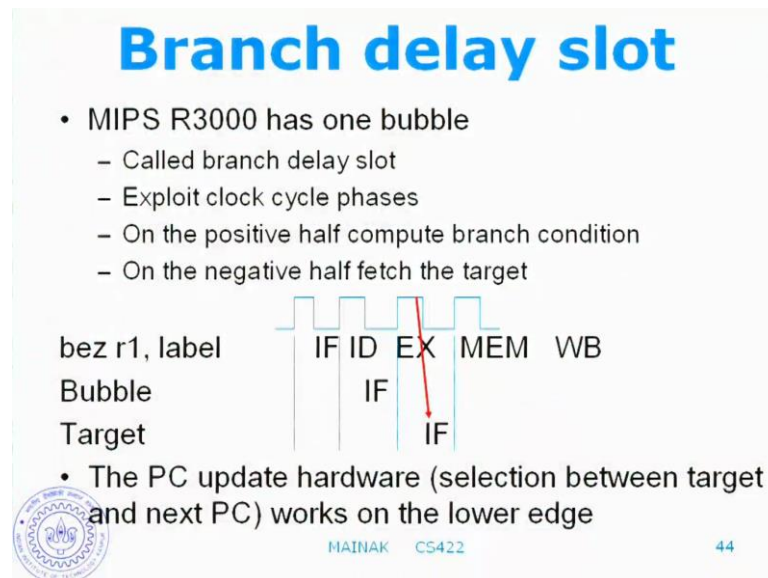
So, if you now look at they are saying that they will always do instruction fetching half cycle, it happens only in the second half of the cycle the instruction fetch and the branch execution will complete within the first half of the cycle. So, now, what you have? If the branch instruction in this here, you have a bubble and now you actually know the fetch target here right here in this cycle, because the branch execution completes the first half it communicates the PC to a fetcher and the fetcher fetch in the in the second half of the cycle ok.

So, essentially can somebodies tell you what do I lose by doing this, of course, I gain back one cycle definitely one instruction I have got rid of one bubble, but I must be losing something what is that?

Student: half should be not enough to...

Exactly, my half cycle should be long enough to accommodate the branch instruction execution and also long enough to complete the fetching actually from memory right.

(Refer Slide Time: 34:25)



So, in the sense you may be sacrifice in terms of cycle you may be running a cross side of slower frequency, but you are gaining back one boundary look at branch.

Student: Sir may be branch instruction execute in the...

Student: Yes

Yes other be know yes that is right yeah. So, branches. So, essentially what does that mean if I go back again. So, refer to this diagram over and over. So, what this means is that, if you look at what the branch is do they do two things right. They will compute the target here in the ALU, and they will carry out the comparison here, they go in parallel. And in most cases you can expect that this is going to be that particular path, this is not this is a simple comparison actually.

So, all I am asking is can I finish this addition in upper cycle, that is what I am alright we have possible depending on how long I have to cycle this, yeah try to know that it is a branch instruction, yes you know it here yeah you know it here. So, we can have a alternative like cycle, yes fetching next instruction itself the third cycle targeting

instruction, and only four cycle when the branch is executed we to know whether the right answer was the next instruction or the target instruction right. So, essentially we have created only one bubble yeah. So, here we do not need half cycle I mean we do not need to increase the clock time, and well here you have two bubbles, one after the branch and the target no I am fetching both the instruction. So, you have to keep one yeah alright that I will know until a that is possible yes. So, you are complicating your PC sequence as well. So, that essentially what we saying is that after the branch I would fetch the fall through all the time yeah. And then I will fetch the then I change my PC to the target, and then I may want to change the PC again back to the right one yeah right yeah of course, yes you can do that.

So, you are saying that here, I am going to fetch the fall through yeah right, here I would fetch the target path which I know from the yeah, you almost know yeah that face one small problem you are can someone tell me what? Extra one thing I need to do, sorry new target may the thing should yeah. So, if you remember the branches come with the offset during their instruction, but we need one more thing to complete the target what is that sorry condition, no condition is differ. So, he say I is into the target. So, there are two things color coming branches one is the powerful otherwise a target the part of target is in the instruction, but I do not know the target yet here I just need know the offset what we have need extra on top of that, sorry adding you need an adder exactly you need an adder here to be target right yeah.

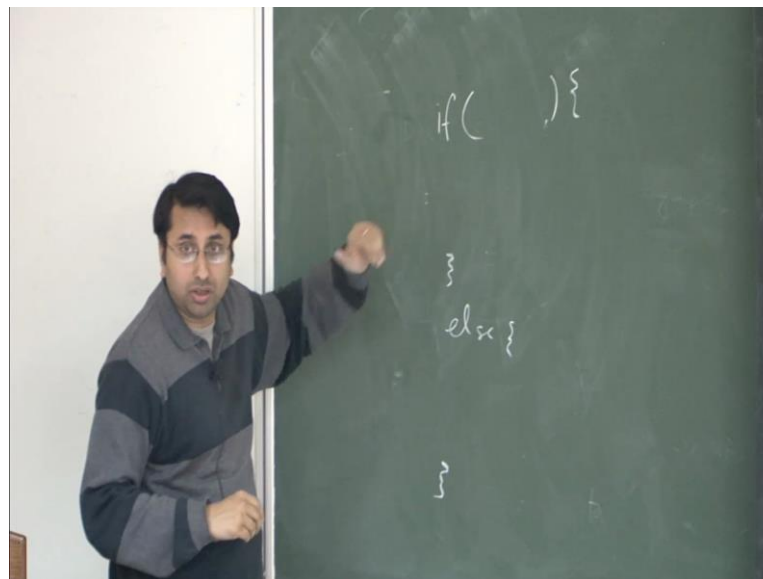
So, if you can afford an adder in the decoder yes, then you can do that MIPS does not have that. So, is it clear to everybody what MIPS has done to solve the problem, I mean not really solve the problem they have reduced it from two bubble to one bubble, the PC update hardware that is the selection between target, and the next PC works on the lower edge. So, yeah from the lower edge you will have the multi flexible could you any question on this clear. How do they actually maintain this constraint the execute branches execute in the first half, and well that also design their hardware.

So, you design the hardware to the same ALU event yeah, but you are saying that the other ALU instructions may still over the other half yeah. So, they have designed the adder to be operating the half cycle the adder is optimizing, yes, yes exactly right. So, ALU has other operations right other than the add operation they can, but yeah I mean if you do this in made a new to operand on the half cycle for most instructions. So, the

adder is often more the longest ones the logical instruction any other question on this. So, this is your branch delay slot this bubble. So, another question is clearly I cannot get rid of this the way, it is given is impossible should get rid of this bubble no way.

So, what can I do here that is still useful can anybody suggest, how can I make good use of this branch delay slot instead of losing the cycle all the time. Can I do something, some other instructions after the what type of instruction can you put there or from, where can you bring this instructions. Can I put any instruction here will that be correct no not any instruction not any instruction, instruction that sufficiently power from the branch instruction yeah, sorry.

(Refer Slide Time: 40:51)



So, let us take a if else control. So, this is my branch, this will be translated to a branch instruction, and in most cases this is going to be the target, this is the fall through. Now yes what you were saying, from where can I bring this instruction which is going to be do. So, and is going to be correct as you have suggested I cannot fill any instruction can you clarify on that is why cannot they put anything here.

Student: Suppose if you insert the only instruction which is... So, branch is taken.

Exactly, exactly. So, that is a very good example, if the branch is taken I should not be executing anything here. So, clearly I cannot put something from here in the branch delay slot. So, what can I then here yeah exactly. So, these are often called the

conversions points. So, branch is diverging and then it is going to converge here, and these are often used for filling up the branch need as not the instructions from here, but you have to be careful why exactly. So, the value that is produced by this instruction like for example, suppose this instruction is writing to register 20. 20 should not be used on the path at the branch is going to execute actually otherwise you will get a wrong value of the register 20. So, you have to be very, very careful on posting something to the branch delay slot. Or any other option yes we can move something above the exactly. So, we can move something from here to the function result. So, similarly again that constant applies that only though instructions that can be moved here, which can be delayed the execution of which can be delayed actually. So, this actually somewhat it easier option to give something from above to the delay slot.

So, it is a job of the compiler today you filled the delay slot appropriately, and the compiler cannot find anything could not no option, this it say that well I have nothing to fill in. So, that essentially amounts to do the one cycle. So, it is still better than losing one cycle all the time, sometimes you will be actually doing some useful in the branch delay slot. So, given the condition at that point in time given the situation, this was consider to be a very good solution, a smart one actually that give the compiler the flexibility to fill in this particular slot as suppose to doing something in hardware.

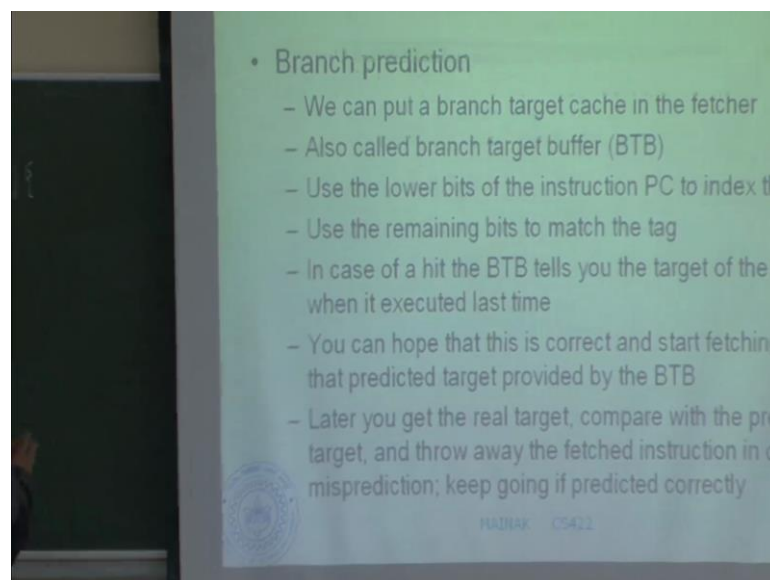
So, you could hardware something actually, but instead of doing that you have given the compiler of flexibility to fill up this particular slot, but later it became a big headache for MIPS to carry forward useful latency, because the problem is that ones you have designed a compiler which emits code which a branch delay slot, suddenly along the line you cannot throw away those code, they assist still have to run correctly on execution. Because later when you go on an a will see techniques to get rid of all boundaries, this becomes a headache big headache actually that, now you could essentially what I saying is that well whatever you do, you have to execute the instruction from the branch, because it compiler is emitting code which may actually be needed to execute here some instruction, you cannot just omit that particular instruction.

So, anyway, so that is your branch delay slot. So, question is can be utilized a delay slot as the compiler guide, the delay slot is always executed with respect of the branch. So, boost instructions common to fall through target paths to a delay slot or from earlier than the branch. So, that is what we have discussed just now, not only responsible to

find you have to be careful also must boost something that does not alter the outcome of fall through or target basic blocks. If the branch delay slot is filled with useful instruction. So, then we do not lose anything in CPI, otherwise you free a branch penalty of one cycle. So, that is the branch penalty number of cycles you lose, if we have alright. So, what else can we do? So, this one we looked at a right in one of the examples earlier. So, you could actually try to predict the outcome of the branch. So, essentially what we are asking is as early as possibly the pipeline right, can you tell me the two question.

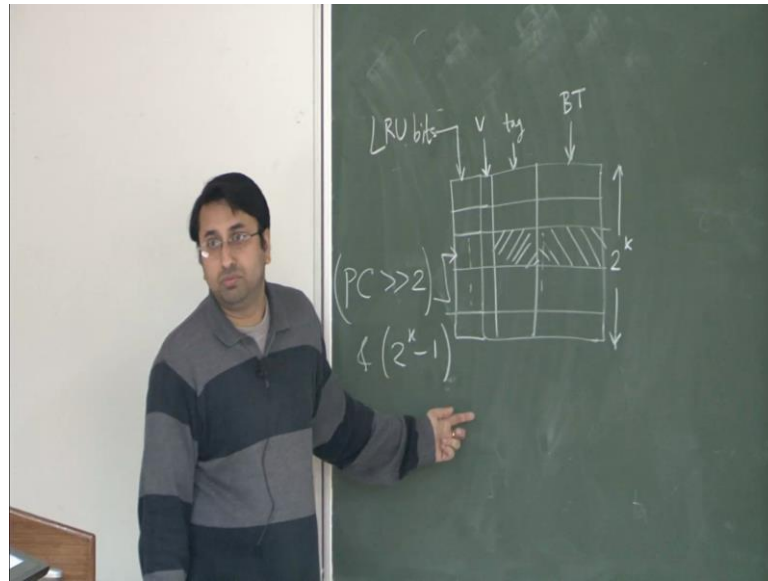
Now, first try to know it is a branch and once I know that it is a branch, can you tell me where the branch is going before the branch is actually executed. So, that is why the importance of prediction.

(Refer Slide Time: 46:19)



So, the simple one of the simplest technique is to put a branch target cache in a fetcher, this is called a branch target. So, essentially what this on store is for a branch instruction, what happen to the branch when the branch is executed last time, that is what it remember? So, in this particular much, much ((Refer Time: 46:41)). So, next time when you fetch the same branch, you can look up this buffer and know where it went last time. Of course, this that could be odd not that, all that the branch may go every time along the same direction. So, how you what is this BTB ((Refer Time: 46:55)).

(Refer Slide Time: 46:57)



So, it is a cache as it mentions. So, it has multiple entries alright, each entry will have a just like your cache, each entry will have a tag, and each entry will have a branch target, and each entry corresponds to some branch in new program. And of course, in addition to these, if you have a set associated with branch target, you will have some bits to carry or replacement. So, there will be certain bits for replacement algorithm, which will not discuss at this point. Like your LRU bits, how many of you have not heard of this recently used replacement policy raise hands, that is wonderful.

So, you gave want to do a LRU replacement in BTB, I will need some ((Refer Time: 48:10)) to remember which one is this recently used which one is most recently used and so on so forth right, but I will not I am not able to this folder. So, first question that lies is ((Refer Time: 48:24)) how do I retakes into the tape, that the first question right. So, what am I doing? I am fetching an instruction right, I am at a particular PC and fetch the instruction. What do I need to know I need to a two things, tell me, if it is a branch, and if it is a branch, tell me where it went last time. I should get answer to both the questions by looking up this table, right. So, what I do is? I take the current PC that I have just fetch from, and suppose I have here these are normally part of two, two to the k entries right. So, what I do is I shift out last two bits, because instructions are hold by long last two which will be 0 in width. So, there are no information and a counter next k bits. So, this same as do that actually. So, I take the large last least significant k bits after moving

out the last two bits of PC use that to index into the tape. So, that keeps you one entry the same right.

So, then I look up this particular tag a, this tag entry stores the remaining base of PC the upper bits, with upper bits of PC match the tag then I know that oh, this is the branch I just fetched which is currently in the table right because a tag actually matched with my upper bits of PC. So, that I got actually the answer that oh, this is the branch, because it seeing the BT. So, then I look up the target and I immediately know where the branch went last time, wait and then I can change my PC and start fetching from that particular instruction in the next cycle. And if I am correct I have that particular bubble that we are told about is it with everyone how I look up the BTB. So, now essentially what I have assumed here there is an invalid that only branches will go into this particular table, well otherwise by just a hit I cannot say that it is a branch, it is actually serving the decoder its decoding that it is this instruction in the branch. So, that phase have to be a little careful when I insert something into the BTB.

So, what do I insert and where do I insert, there are two questions actually. So, what are the answers, any suggestion? What goes into the same, and when does it go into the table.

Student: When you say branch that we get to know from the decoder.

And...

Student: Insert in the decoder in the BT...

((Refer Time: 51:13)) So, branch goes to five stages of execution right.

Student: After execution we get to know the targets.

Right, exactly. So, when then branch finally, executes I know the actual target, and of course, I know its PC, but it was fetched. So, then I can insert into this table, it will the target and the tag. Now to reduce the space over head to BTB or rather to improve the of the capacity of BTB usually the branches which are not taken are not inserted in the BTB, because if you ((Refer Time: 51:56)) in the BTB, that is if you do not find this particular PC in the BTB, it will just fall through. So, you never insert a branch which is not taken, you only insert a taken branch as in the BTB, because that is what matters?

So, in case of a hit, the BTB tells you the target of the branch when it executed last time. You can hope that is correct and start fetching from the predicted target provided by the BTB. Later you get the real target compared with a predicted target and throw away the fetched instructions in case of a misprediction, keep going if predicted correctly. This is the concept clear to everybody what BTB exactly stores, why it stores that and how you look up the BT. So, it close with one small question can somebody tell me for what kind of control transfer instructions, BTB is going to be great it would be 100 percent accurate.

Student: ((Refer Time: 52:58))

What will the point of a call there be sorry unconditional jumps, unconditional jumps exactly unconditional jumps will have 100 percent, not exactly 100 percent last time you will be going to suffer its, but the remaining execution of that unconditional jump you are going to be correct, because every time it goes to the same place. Any else any other instruction said to have 100 percent sorry...

Student: (Refer Time: 53:29))

Say it switch case no, do not yeah I will go to different cases at different times no no, but remember that this branches are going to have the same PC should be over writing the same entry over and over yeah. It will very bad actually what else something that is lying unconditional jump.

Student: ((Refer Time: 53:58))

No, no, we cannot do the same phase every time, you may call the same procedure from different places right, sorry...

Student: Loop

Loops using. So, you will be fairly accurate except the last step, that is right, what else?

Student: ((Refer Time: 54:20))

Something like unconditional jump, what about function calls, because it always going to the same place right, you call function we always go the function. So, these are called direct procedure calls; there are redirect calls also. So, they will be very bad.