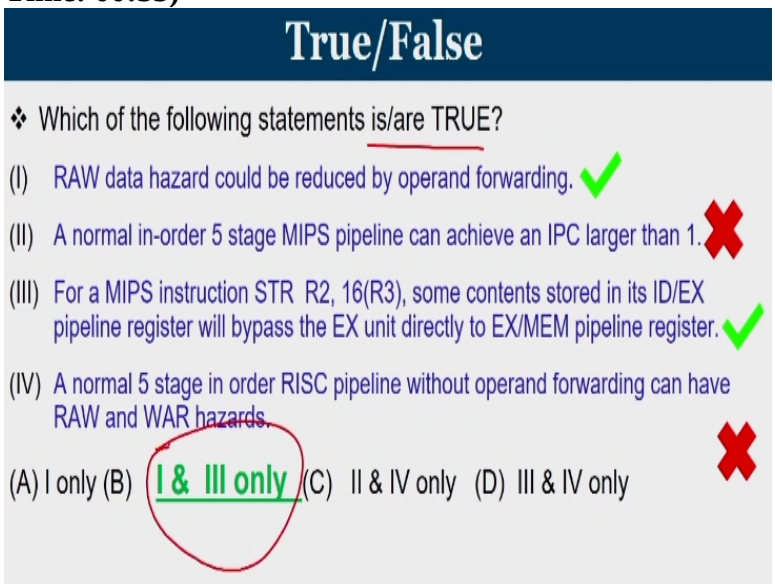
Advanced Computer Architecture
Dr. John Jose
Department of Computer Science & Engineering
Indian Institute of Technology-Guwahati

Tutorial - 2
Pipeline Hazard Analysis

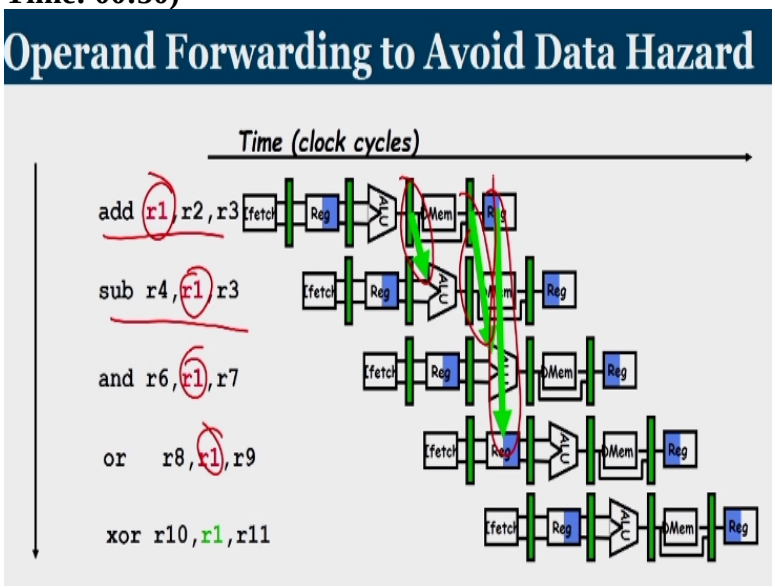
Welcome to the tutorial session of the second week. In tutorial two, we are focusing your attention on understanding pipeline hazards.

(Refer Slide Time: 00:35)



Let us move into the first question. It is basically a set of statements, we have to tell whether it is true or false. Which of the following statements is or are true. RAW data hazard could be reduced by operand forwarding. So that is the first one.

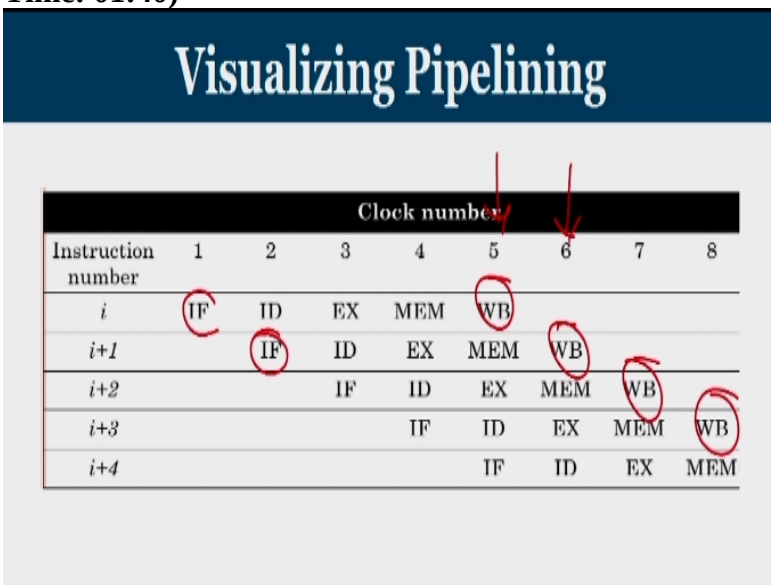
(Refer Slide Time: 00:50)



So consider the set of the instructions that has been given here. This ADD instruction then sub. And we know that the add produces a result which has to be written in r1 and all other subsequent instructions are going to read from r1. So there exist a RAW dependency and we can see that by the concept of operand forwarding we are forwarding the result from one functional unit to another.

That is from output of ALU to input of ALU and from output of MEM stage to the input of a ALU. So because of this, I am not having any stall. So in the statement RAW data hazard could be reduced by operand forwarding is true. The second one is a normal in-order 5 stage MIPS pipeline can achieve an IPC larger than one.

(Refer Slide Time: 01:40)

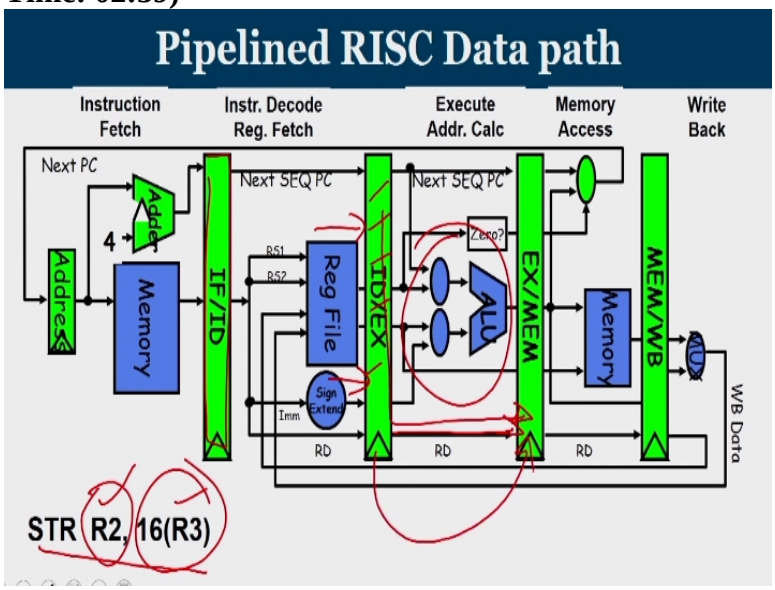


So this is how we visualize a pipeline. In every cycle, I am going to fetch an instruction and in every cycle in ideal case, we are going to complete one instruction. So on an average if you look at every cycle, I am completing one instruction. So the statement a normal in-order 5 stage MIPS pipeline can have an IPC, instructions per cycle larger than 1. I cannot complete more than one instruction per cycle in a normal in-order 5 stage MIPS pipeline. So an IPC larger than 1 is impossible. So the second statement is false.

Moving on to the third statement for a MIPS instruction store R2, 16(R3) some contents stored in its ID/EX pipeline register will bypass the EX unit directly to the EX/MEM pipeline register. So let us try to understand what is the store instruction.

The contents in the register R2 has to be stored into memory and the address to which the storage happen is 16 plus content of R3.

(Refer Slide Time: 02:39)



Let us see how this happens.

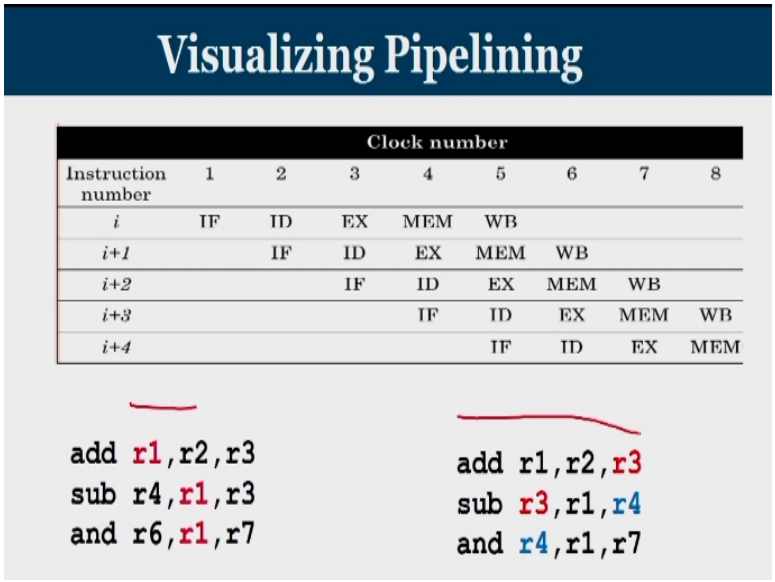
Store R2, 16(R3)

so your instruction is being fetched. So, at the end of the fetching stage your instruction is available here. You fetch the contents of R2 and R3. So R2 and R3 is been now going into it ID/EX register. Now 16 and R3 is added in the ALU to get effective address. In the meantime, the contents of R2 which is already available in the ID/EX register that moves to EX/MEM register.

So all the contents that is available in the ID/EX register are not used by the ALU. Through this line some of the content, it is basically the content of R2 that is the value to be stored in the memory which is already locating in our R2, we move from ID/EX register to the EX MEM register. So the statement for a MIPS instruction store R2, 16(R3) some contents stored in its ID/EX pipeline register will bypass the EX directly to the EX/MEM register is true. It is the content of R2.

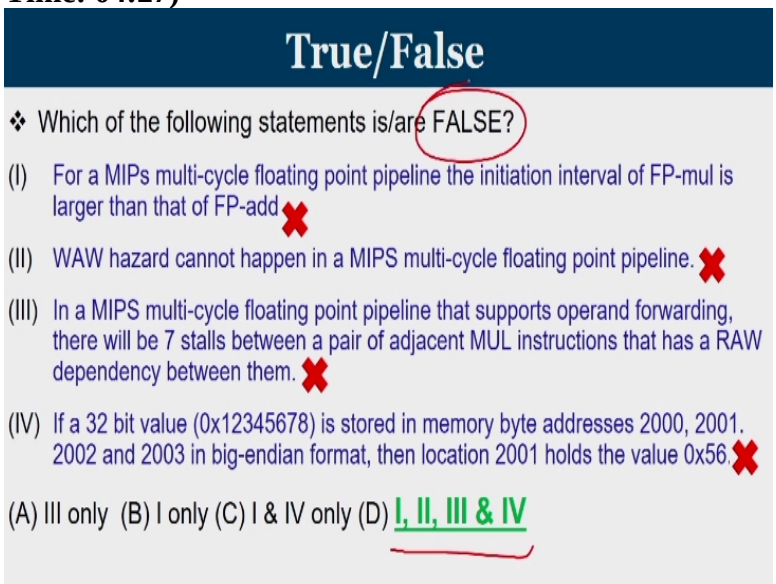
The fourth one is a normal 5 stage in-order RISC pipeline without operand forwarding can have RAW and WAR hazards.

(Refer Slide Time: 03:53)



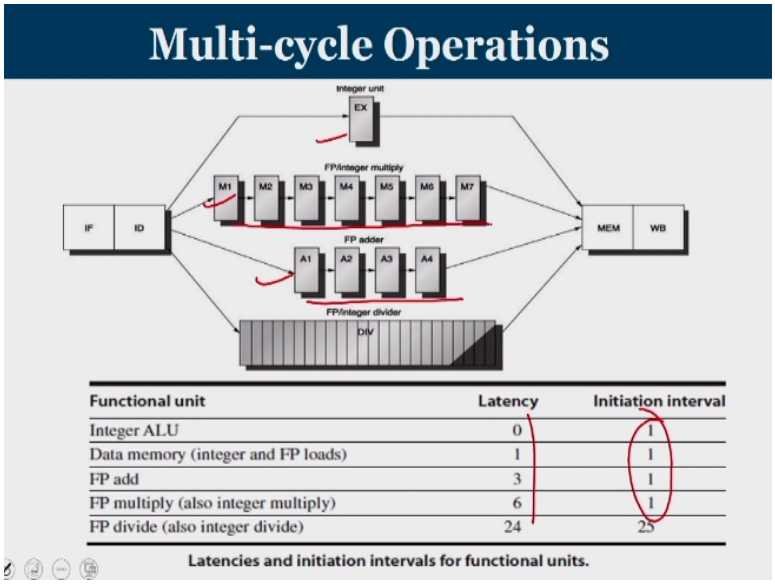
See this is the general pipeline that you have seen and this is RAW hazard. And here we have the WAR hazards. So the statement, the last statement, a normal 5 stage in-order RISC pipeline without operand forwarding can have RAW and WAR hazards ,that is false. So here which are the following statements are true, the answer is one and three.

(Refer Slide Time: 04:27)



So now we will look into the statements like this, the next set of true or false statement. Which of the following statement is or are false? For a MIPS multi-cycle floating point pipeline, the initiation interval of floating point mul is larger than that of floating point add.

(Refer Slide Time: 04:43)



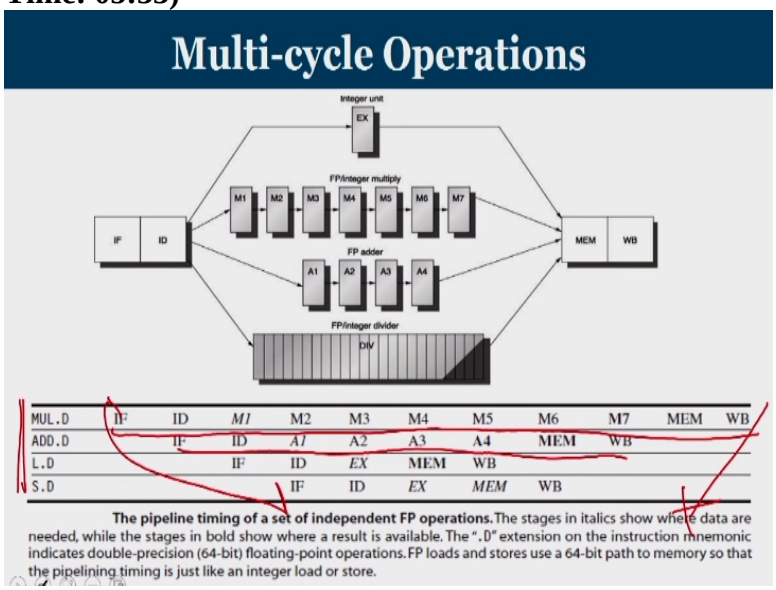
Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

Latencies and initiation intervals for functional units.

Let us see what is there. So the initiation interval is been defined as how much cycle has to be elapsed if you wanted to use the functional unit again. So the integer ALU the floating point multiplier and the floating point adder has an initiation interval of 1. But they differ in latency. There are 4 stages in the FP adder and there are 7 stages in the integer of floating point multiplier.

So the statement for a MIPS multi-cycle floating point pipeline, the initiation interval of floating point mul is larger than that of a floating point add is wrong. They both have the same initiation interval, but however, FP mul has a larger latency than an FP-add. Now WAW hazard cannot happen in a MIPS multi-cycle floating pipeline.

(Refer Slide Time: 05:33)



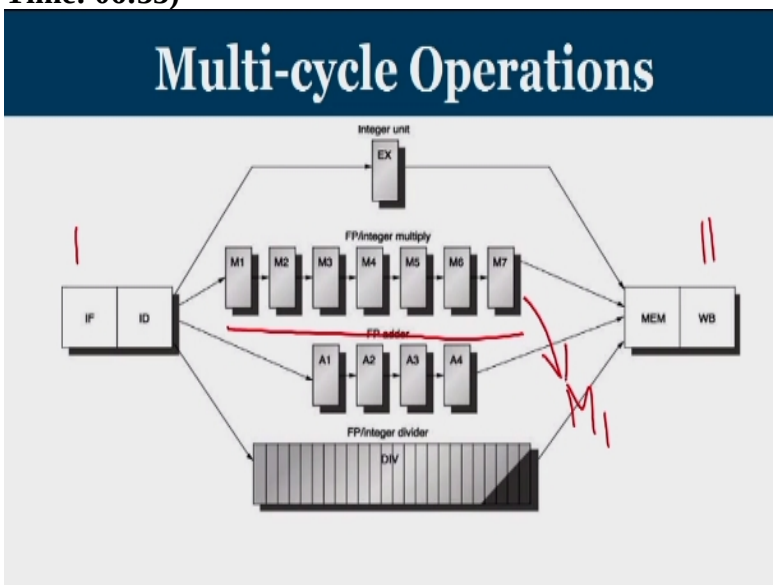
The pipeline timing of a set of independent FP operations. The stages in italics show where data are needed, while the stages in bold show where a result is available. The ".D" extension on the instruction mnemonic indicates double-precision (64-bit) floating-point operations. FP loads and stores use a 64-bit path to memory so that the pipelining timing is just like an integer load or store.

So WAW hazard is write after write hazard. So you consider the case that when you have a MIPS multi-cycle floating point pipeline where some instructions are longer. The first instruction is a multiplication. So it takes 11 cycles to complete. Let us say the second one is add which will take only eight cycles to complete.

So surely even though we are fetching instruction in order the execution is out of order or the completion is out of order. So when the completion is out of order, there is always a possibility that we can have write after write hazards. So the statement, in a MIPS a WAW hazard cannot happen is false. WAW hazard will happen because instructions are completed out of order.

Moving on to the third one, in a MIPS multi-cycle floating point pipeline that supports operand forwarding there will be 7 stalls between a pair of adjacent multiplication instructions that has a RAW dependency between them.

(Refer Slide Time: 06:33)

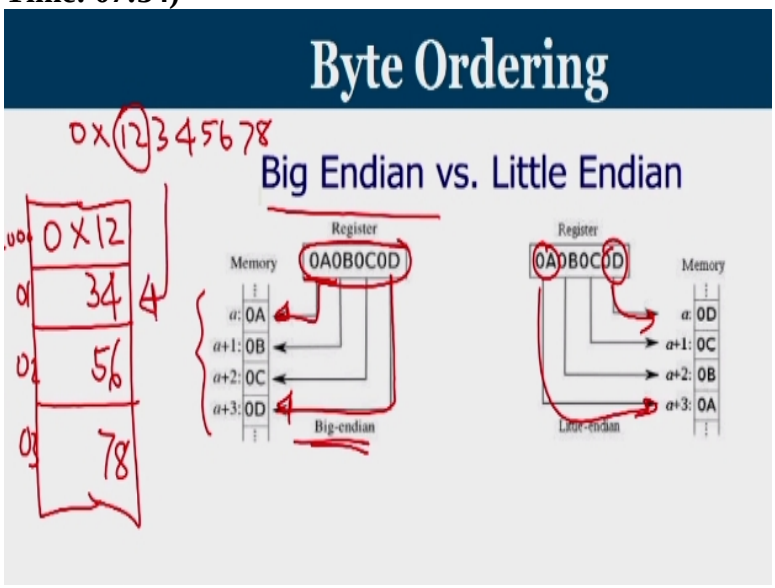


So this is what the general structure of this multiplication unit. You have the first instruction which starts at 1 and then it completes only at 11. If the second instruction is going to have a RAW dependency on it, then the M1 of the second instruction can happen only after M7 of the previous instruction. So there will not be 7 stalls, there will be only 6 stalls between them.

So the statement that there are 7 starts between a pair of adjacent multiplication instruction that has a RAW dependency is wrong. If a 32 bit value 0x12345678 is

stored in memory byte addresses 2000, 2001, 2002, and 2003 in big-endian format, then the location 2001 holds the value 56.

(Refer Slide Time: 07:34)



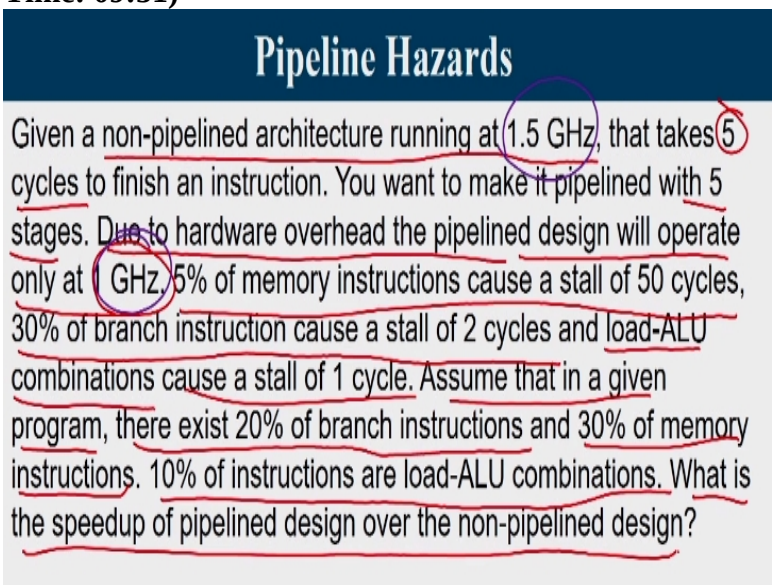
Let us try to understand what you mean by big-endian and little-endian. In big-endian when you store a 4 bite data into 4 continuous memory location, if MSB is stored in the lower address and LSB is stored in the higher address then that is known as big-endian. If the LSB is stored in lower address and MSB is stored in the higher address that is called little-endian.

So the statement when you have a 32 bit value, 12345678 is going to be stored in locations 2000. So these are the locations 2000, 2001, 2002, and 2003. So this is 2000, 2001, 2002, and 2003. And our value is 0x12345678. Then we are going to store them in big-endian format. So once you store in big-endian format, the MSB is stored in the lower address.

So 0x12 get stored here; 3, 4 gets stored here; 5, 6 gets stored here; and 7, 8 will get stored here If it is in big-endian format. So what they are telling is 2001 holds 56. So 2001 is holding 34. So, the statement that 2001 holds 56 is wrong. So all the 4 statements are wrong. So the answer would be 123 and 4, they all are false. So in the first two questions in today's tutorial, there were 4 statements given and we were asked to check whether those statements are true or false.

And we have now seen by correlating with the theory concept that we learned in the lectures to ascertain whether these statements are true or false. So to answer these kinds of questions, you require a thorough reading of the material, the video that has been covered and supplementary reading of the textbook material in which these topics are discussed.

(Refer Slide Time: 09:51)



Let us now move into pipeline hazards. Let us try to understand this question. Given a non-pipelined architecture running at 1.5 GHz which will take 5 cycles to finish an instruction. You want to make it pipelined with 5 stages. When I am having pipeline interface register due to the hardware overhead, the pipeline design will operate only at 1 GHz.

So you have to understand your non- pipelined design will operate at 1.5 GHz. So the clock is 1 divided by 1.5 GHz whereas the pipeline design is operating at 1 GHz. Now 5% of the memory instruction cause the stall of 50 cycles. 30% of branch instruction cause a stall of 2 cycles and load-ALU combinations, we have seen that if there is an ALU operation immediately after the load operation, they cause a stall of 1 cycle.

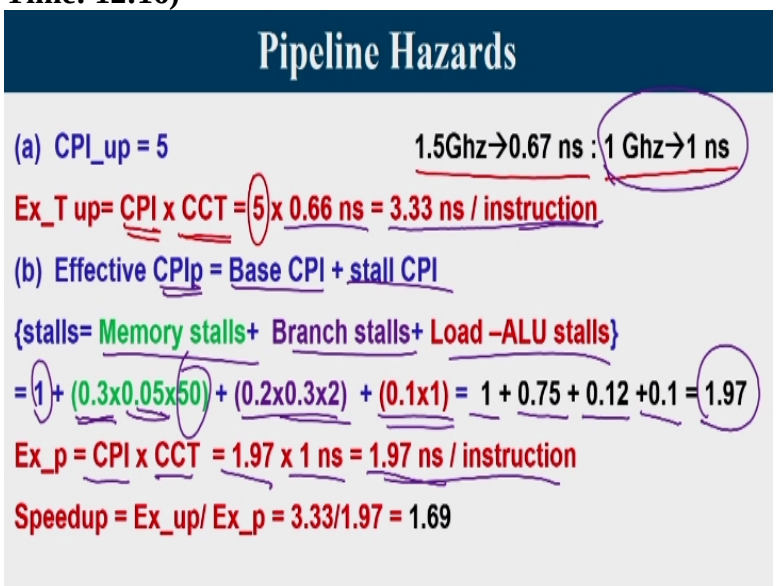
Assume that in a given program there exist 20% of branch instructions and 30% of memory instructions. 10% of instructions are load-ALU combinations. What is the speedup of the pipelined design over the non-pipelined design?

This is the question that has been given. So let us try to understand what the question is all about. We have an unpipelined design wherein the timing is given 1.5 GHz of clock.

We have a pipeline design which can operate only at 1 GHz. But idea of pipeline is the base CPI is 1. Every cycle 11 instruction is getting completed. In the case of a unpipelined version it takes 5 cycles to complete an instruction. Now in the pipeline, we will be having hazards. The first kind of hazard is called the memory issue wherein you are trying to fetch something from the instruction from memory, you are not getting.

So for certain percentage of memory instructions it will be hit in the memory. For certain percentage it may be a miss in the memory. So whenever you miss you are going to have 50 cycles of stall. Similarly, when you come to branches, there also there are hazards for certain fraction of branches. And there is a data hazard for load-ALU combinations that is for certain percentage. Now we have to find out what is the overall execution time in this case.

(Refer Slide Time: 12:16)



So the

CPI unpipelined = 5

So it takes 5 cycles to complete an instruction. The unpipelined version is working at 1.5 GHz, roughly 0.667 nanosecond and the pipelined version is running at 1 nanosecond time. The execution time of the unpipelined is CPI into clock cycle

execution time for 1 instruction. So it takes 5 cycles to complete 1 instruction and each cycle is 0.6667 nanoseconds.

So roughly it is 3.33 nanosecond it will take to complete 1 instruction in the unpipelined design. So the effective CPI in the case of pipelined instruction is the base CPI that is 1, in every cycle we complete 1 instruction plus whatever stalls that is happening. Now what are the kind of stalls? We have memory stalls, we have branch stores and we have load-ALU stalls.

The memory stalls, so what is the base CPI, base CPI is 1. That means every cycle 11 instruction is getting over. Memory, you have 30% of instruction that are memory but 5% of memory instruction will incur 50 cycle stall. So it is :

$$0.3 \times 0.05 \times 50$$

Coming into branch, we have 20% of the instruction that are branch but 30% of the branch instruction only will create hazards. It will create 2 cycle of stalls.

Coming into load-ALU 10% of load-ALU instruction combinations are there and they are going to create 1 cycle stall. So it is

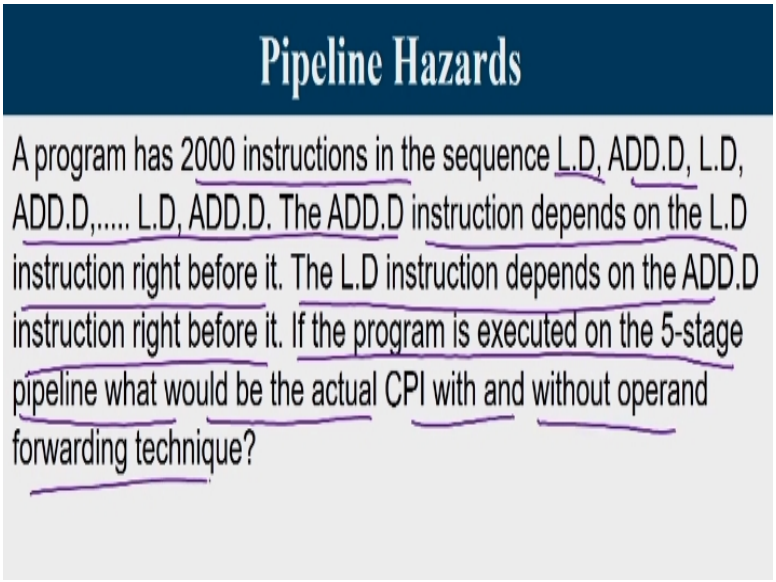
$$\text{Effective CPI (pipelined)} = 1+0.75+0.12+0.1 = 1.97 \text{ (roughly)}$$

So execution time in the case of pipelined CPI into clock cycle time CPI is 1.97. And it is going to operate, the pipelined version is going to operate at 1 nanosecond clock.

So it is 1.97 nanoseconds per instruction. Now what is a speed up? Speed up is execution time of unpipelined divided by execution time of pipeline. So

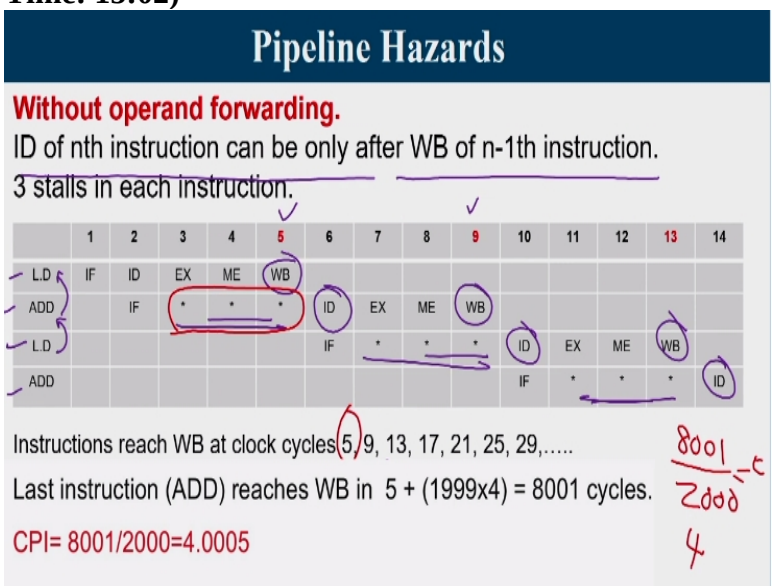
$$\text{Speed up} = 3.33/1.97 = 1.69$$

(Refer Slide Time: 14:23)



The pipeline hazards, next question. A program has 2000 instruction in the sequence load add, load add like that. So the first, third, fifth, seventh, all old number instructions are load and all even number instructions are ADD. The ADD instruction depends on the load instruction right before it. The load instruction depends on the ADD instruction right before it. If the program is executed on a 5-stage pipeline, what would be the actual CPI with and without operand forwarding?

(Refer Slide Time: 15:02)



So without operand forwarding technique let us try to understand how the instruction will flow. So this is the sequence of instruction. You have load, ADD, load, ADD like that. Now the peculiarity is this ADD is dependent on the load. So the result of load will happen only at the writeback stage and since there is no operand forwarding, the

ADD can get its data only after the previous load has written the result because there is a dependency between them.

That means the ID of and ADD statement can happen only after the writeback stage of previous load. So this ADD is going to write the result here. And this load is dependent on the previous ADD. So the ID stage of a load can happen only after the writeback stage of the previous ADD is over. Similarly, you can see that the ID of every instruction can happen only after the writeback stage of the previous instruction.

So ID of nth instruction can be only after the WB of the n – 1th instruction. So there are 3 stalls. You can see there are 3 stalls in each of the instruction if you follow this pattern. So instructions reaches the writeback stage. The first instruction reach writeback stage at 5, second reach at 9, third reaches at 13, fourth reaches at 17. So it is a pattern that you can see; 5, 9, 13 like that.

So the last ADD instruction will reach the WB stage, so the first instruction reaches at 5, then I have another 1999 instructions more. Each will be completed at a difference of 4 clock cycles. The first instruction will reach writeback stage at 5 and then

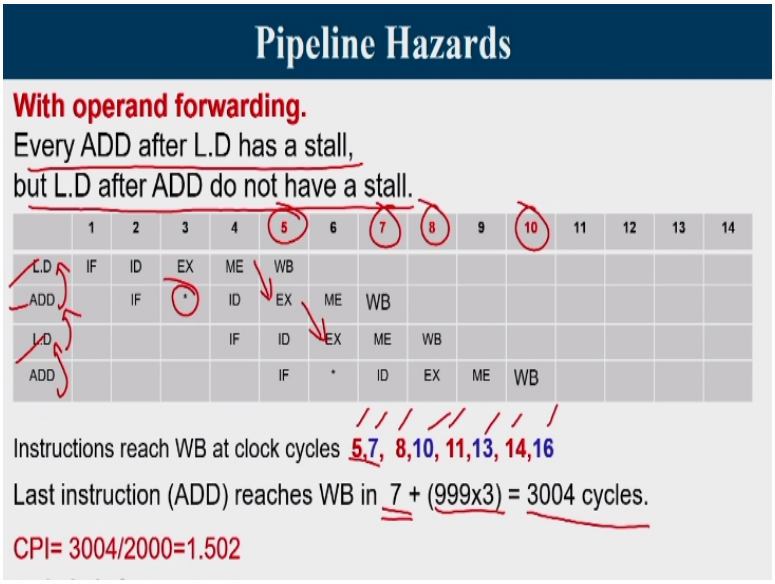
$$5 + 1999$$

more instructions which will take 4. So the answer is going to be 8001 clock cycles. So if you look at the CPI value, then it will be

$$CPI = 8001/2000.$$

So it is roughly going to be 4. Because you know that every cycle one-one instruction is getting over and you have 3 stalls extra. So the average CPI is going to be 4 in this case.

(Refer Slide Time: 17:13)



Now we will see what is with operand forwarding. So if it is with operand forwarding, we can see that load and dependency. So this ADD instruction is dependent on the load. So ADD instruction we get the data only at the M(MEM). So MEM is forwarding to EX. So that is going to incur a 1 cycle stall here. Similarly, this load is dependent on the ADD.

But there it is not a problem. Again, this is dependent on the previous load. So every ADD after load has a stall. But load after ADD do not have a stall, That is the peculiarity. The first instruction complete at 5, second at 7, third at 8, fourth at 10 like that. So if you look at the pattern instructions reads writeback at clock cycles 5, 7. So all the red color markings, they are the load instruction.

So load instruction getting completed at 5, 8, 11, 14 like that. ADD instruction is getting completed at 7, 10, 13, 16 etc. The last instruction is ADD. So as we look into the pattern of ADD only, we look into only the blue. So first ADD instruction is getting completed at 7. We have another 999 ADD instructions more. Each will be completing at a clock cycle shift of 3 cycles.

So at 3004 we have the last ADD instruction completing. So what is CPI?

$$CPI = 3004/2000 = 1.502 \text{ (roughly)}$$

So given in this question, what we have seen is if there is a sequence of load and ADD combinations, then if it is without operand forwarding then every instruction

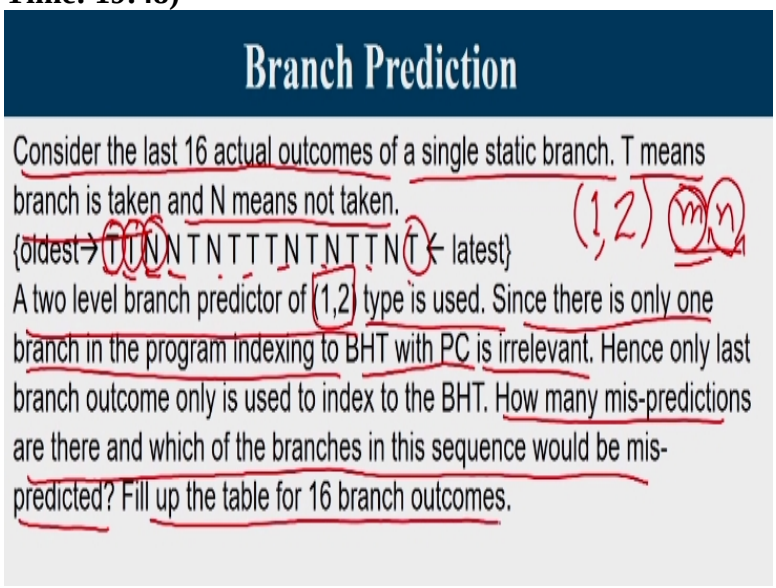
has a stall of 3. So normal an instruction take 1 cycle to complete plus 3 more cycles. So on an average the CPI is 4.

And when you come to the operand forwarding mechanism, the second instruction is dependent on the first instruction. Every ADD is dependent on the previous load and there is a 1 cycle stall. But every load that is dependent on an ADD, since there is operand forwarding, there is no stall that happens. So every alternate instruction we have 1 stall. So when you have 2000 instruction you have 1000 stalls.

So close to 3000 would be the total number of cycles to complete. So

average CPI = 3000/2000 = 1.5

(Refer Slide Time: 19:48)



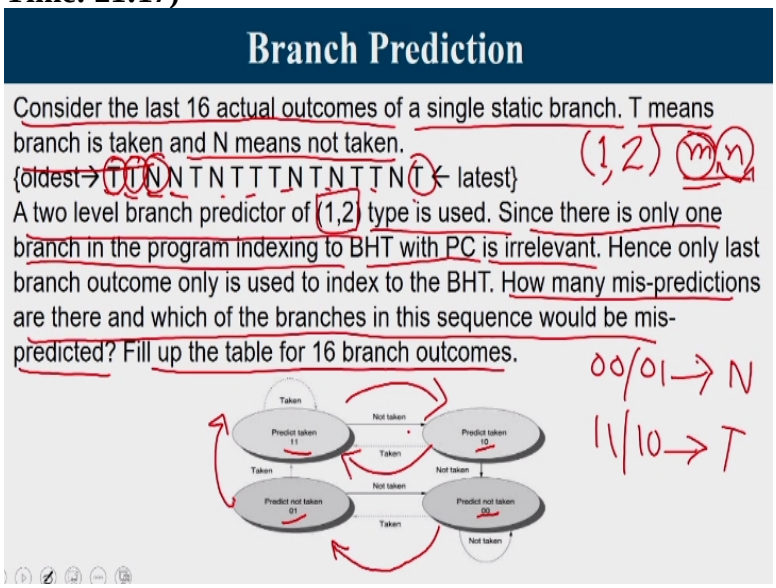
Now let us move into another question which is a branch prediction question. Consider the last 16 actual outcomes of a single branch where T means branch is taken and N means the branch is not taken. So this is the pattern. So if the statistics of what is the outcome of a branch has been collected. So T means taken. So it is taken, taken, not taken, not taken like that for the last 16 outcome. This is the latest one.

The latest one was taken. The first one was taken. Second time the iteration was taken. Third was not taken. That is the way how we have to interpret the sequence. A two level branch predictor of 1-2 type is used. Since there is only one branch in the program indexing to be BHT, branch history table with the PC value is irrelevant. So

1-2 predictor means, last one outcome of a branch is being used to index into the table and you are using a 2 bit predictor.

So MN predictor, basically if a branch has been mentioned as MN predictor, M stands for last M occurrences of the branch is being used to index into the branch history table and each entry has an N bit branch predictor. So they are asking how many mis-predictions are there and which of the branches in the sequence would be mis-predicted. Fill up a table for 16 branch outcomes.

(Refer Slide Time: 21:17)



So we are basically using a 2 bit predictor. So when the value of the predictor is 00 or 01, we predict that the branch is not taken. When the value is 11 or 10, we predict that the branch is taken. So we have to keep this in mind, whenever the value is 00 or 01 the prediction is branch will not take and whenever the value is 11 or 10, the prediction is branch will be taken.

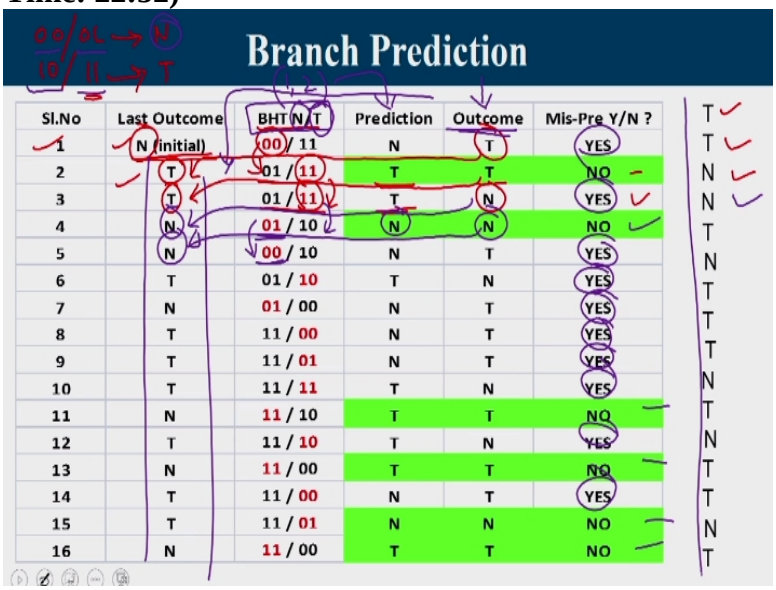
Now let us see at the transition. When we are in 00 we predict that branch is not taken. But if actually the branch is take then from 00 state I how to correct my state as 01. When I am in 01 then if branch is taken actually if it is taken then I change my prediction value to 11. Similarly, when I am in 11, and if the actual outcome is not taken, then I am moving into 10.

Similarly, when I am in 10, when the branch is actually taken, I move into state 11. So this is the transition diagram that has to be familiar with us.

(Refer Slide Time: 22:30)

Now what happens here is, here it has been mentioned that what will happen to the last 16 outcomes of this branch. So this will tell you what happen to the branch. So the first iteration the branch was taken, second iteration branch was not taken, like that the 16th iteration branch was taken.

(Refer Slide Time: 22:52)



Now let us see this table. This table looks a bit clumsy when you try to work out this question for the first time. I will make it more simple. This is your branch, branch history tables entry. So the branch history table has two columns. The first column tells when the branch entry is not taken, you have to consult the first column. When the branch entry is taken, then you have to look into the second entry.

So it is basically a two entry table. So this is the two entries. And I mentioned that we are mentioning a 12 predictor. So we look into what is the outcome of the last branch. If the outcome of the last branch is not taken, then refer this. The last branch is taken then refer the second entry. So to interpret into the table. If the last branch was actually not taken, refer this entry.If the last branch was actually taken then refer the second entry.

So referring into first and second entry is the most crucial thing as far as working with this branch table is concerned. Now the initial configuration of the branch is 00/11. So my first one is my initial condition is N last outcome. Now I am referring into this one. Since the last outcome for the initial case to start with let us assume that it is not taken.

Since it is not taken I refer into the first entry 00. Since the entry is 00 we have to understand that if the entry in the table is 00 or 01 then the prediction is not taken. If the entry is 10 or 11, then the prediction is taken. So since the entry is 00, I predict it has not taken but what actually happened with the first branch? The first branch was taken. So my prediction was not taken, but the actual outcome was taken.

That is a mis-prediction. I predict that branch will not take but actually the branch was taken and that is known as mis-prediction yes or no, this is a mis-predicted case. Now I move into the next entry, I have to make some changes. My state was 00. But the prediction went wrong. When prediction went wrong from 00, I move to 01. That is what has been the change that you see.

So at the end of the first row, the table gets itself updated to 01/11. Why this is 11? I am not going to make any change to that entry which I have not referred. I have referred only the first entry 00 and first entry 00 based upon that I predicted it has not taken actual outcome was taken. So it is a mis-prediction. Whenever there is a mis-prediction, I have to change the state. Now the second row works with the new updated table.

So what is the new table? The new table is 01/11 and this is already taken. Now the last outcome of the branch was taken. So this will get repeated. So these both will be

same. Whatever was the last outcome that has been entered here. So the last outcome was taken. Since the last outcome was taken I look into the second portion of the table. When the last outcome is not taken look into the first portion of table. Last outcome is taken, look into the second portion of the table.

The second portion is 11, that means the prediction will be taken. This stores that. Whenever the entry is 11 the prediction is taken. The actual outcome is also taken. That means it is not a mis-prediction. So in this case the third row is same as this. Now the last outcome was taken How will they get this? This was the last outcome. Last outcome was taken. I refer into this.

It is predicting that the branch will be taken but actually the third time the branch was not taken. So that is a mis-prediction. Once it is a mis-prediction, from 11 when I mis-predict I move to 10. That is what is happening here. So, that is the change that is being reflected. I move into 10. Now I continue with the fourth iteration, the previous one was not taken. So I go here, this is not taken.
Since it is not taken I refer to the first half. The first half is 01. 01 indicates not taken. Actual outcome was also not taken. So there is a perfect match there. So when at 01 when the branch was not taken then we have to understand that it will go back to 00 state. So this is our table. When you are in 01 when the branch is not taken, I move into 00. And then the previous outcome was not taken. So I continue.

So to fill up this table, whatever is here, the same thing will get repeated in this in the next row. This is the outcome. That outcome is used for the subsequent rows. So if you continue like that, we have to understand I have to update only that entry which I have referred. If the entry that I have referred is correct, then I make the predictions, I look into what is the outcome.

Based on the outcome I make appropriate changes and update the entries downward. This is the way how it is to be flourished. So wherever this green color that has been shown these are not these are cases where the prediction and the outcome was matching. In all other cases where the yes is marked, these are places where I got mis-predictions. So I got total of 10 mis-predictions from this entire setup.

So what has been given here is you have given a structure of a table. This is a 12 predictor. So last one occurrence only is being checked and based upon last one occurrence, if the last occurrence is not taken you refer into the first portion of the table. If the last occurrence is taken, you refer into the second portion of the table. Now the contents of this table you look into 00 or 01, then the prediction is not taken.

If the content of the table is 10 or 11, then the prediction will become taken. Now you see correlate with what is actually happening. This is a sequence which is telling what is the outcome of the branch. If they are perfectly matching, then there is no mis-prediction. If there is a mismatch, so the value that we predicted and the actual outcome is different then there is a mismatch and then we have to correlate that.

So with this we come to the end of the tutorial. In this tutorial, we have seen about how to handle with the hazards, few true or false statements were there. And then we had few numerical problems in handling with the hazards and then some dependency issues with and without operand forwarding and towards the end we had branch predictor. This week also we will put up some short exercise questions also for you to work on with respect to branches. Thank you.