Advanced Computer Architecture
Dr. John Jose, Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati, Assam.

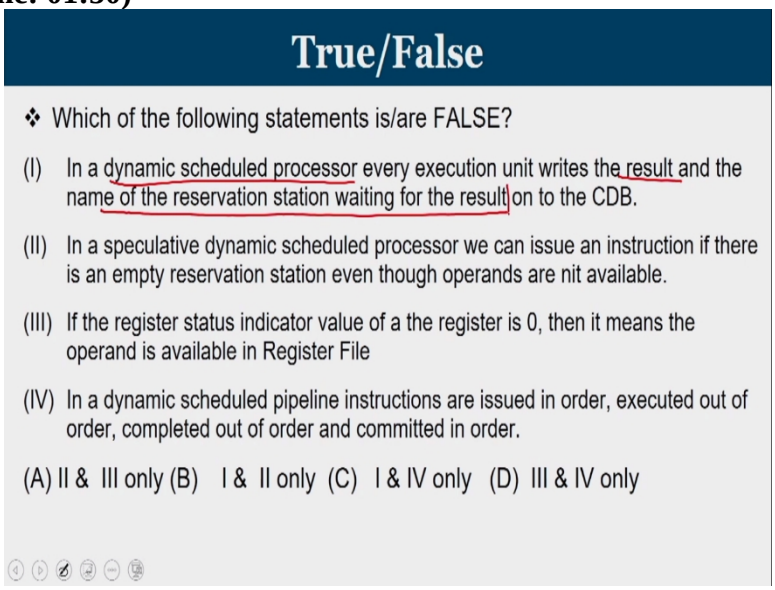
Tutorial 3
Static and Dynamic Scheduling

Welcome all of you to the tutorial of week number 3. In today's tutorial, we will try to understand about little bit deeper concepts in static and dynamic scheduling. Over the lecture videos, you might have already seen how compiler is going to help the hardware in understanding hazards, and then improving the performance of the pipeline, which we broadly called under static scheduling techniques.

And then we have seen about dynamic scheduling techniques with Tomasulo's algorithm and the speculative approach in order to handle branches and controls hazards. So the main objective of this tutorial is to get some experience on what are the kind of ways in which problems can come, such that your knowledge in this domain can be used for those problems. So the first set is been organized as a couple of statements where you are asked to find out whether those statements are true or false.

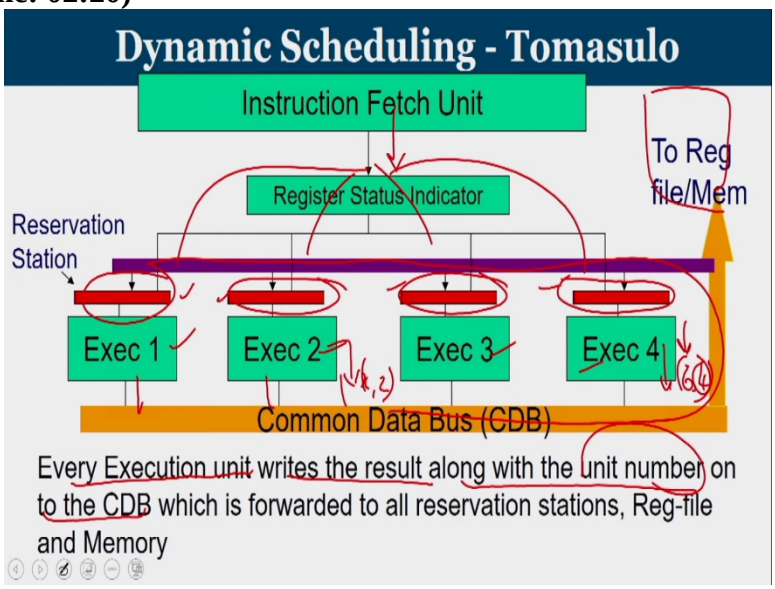
And then we have a multi cycle, MIPS pipeline, where in 14 point units are been involved, and trying to see how operand forwarding has been implemented. And the third question that is been planned in this tutorial is about Tomasulo's algorithm and it is detailed working.

(Refer Slide Time: 01:50)



Consider the first question which of these following statements is/are false. Let us examine one by one in the dynamically scheduled processor, every execution unit writes the result and the name of the reservation station waiting for the result. So, there are 2 things that has been mentioned, it write the result as well as the name of the reservation station waiting for the result on to CDB.

(Refer Slide Time: 02:20)



Let us try to understand how the basic Tomasulo's algorithms implementation structure is there, these are the execution units. Now, whatever you see in this red color, they are the reservation station and then we have the bus. So, after the instruction is being fetched, you are going to issue these instruction into appropriate functional units. Now as and when the operands are ready, so, here it is, the register and memory assignment operands are ready you are going to enter into the execution unit.

Sometimes, when you wait in the reservation station, all the operands may not be available. So, the way by which you are getting operands is, whenever the operands values are ready, you will get it that is the way how it is getting. So you are continuously pinging the common data bus and as and when the value is coming, then you are going to absorb it. So, every execution unit write the result along with the unit number on to CDB, this unit number is suppose if this execution unit 4 is going to produce a result let us say 6, then it is going to tell 6 and 4.

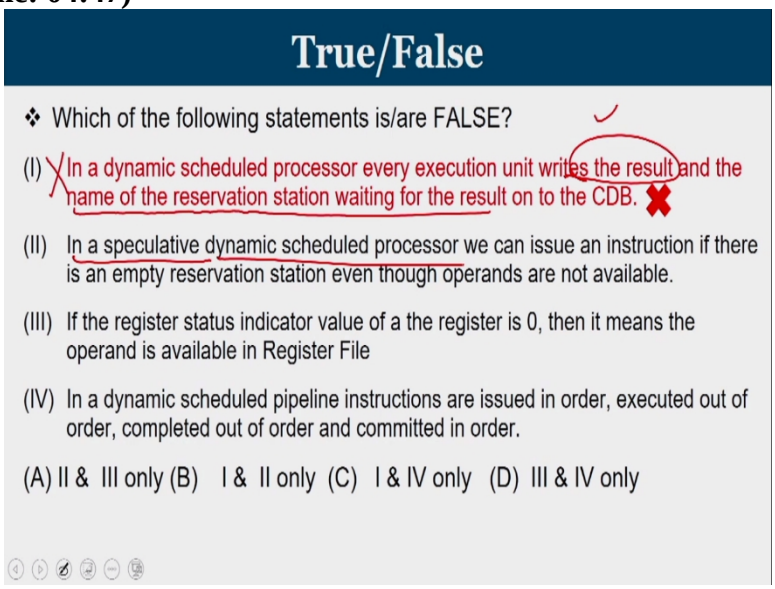
6 indicate the result and 4 indicates the unit that produced. So, that said this is going to produce a result x. So it is written as Exec 2, the 2 is the execution unit. So, why you do like

this, because there may be some operation that will be waiting in the reservation stations, where they are waiting for the updated value of the operands and the operands will be produced by some of these execution unit.

So, when the value is entering into common data bus, those entries which are waiting in the reservation station should know which execution unit has produced the result. So in no way and execution unit knows which are the reservation station waiting for the result that is been produced by these execution unit. It is the other way around the reservation stations are waiting for a result that is been produced to by some execution unit.

Execution unit does not know the otherwise. So, when a result is being produced by execution unit, the result as well as the execution unit number is being entered or broadcasted into the common data bus. So, everybody pinging common data bus will know this is the result produced by execution unit N and whoever is waiting for the result execution unit N and can take the result and then proceed.

(Refer Slide Time: 04:47)



So, the first statement is actually false, we the execution unit will produce the result that is true, but it will produce the result or it is going to write the name of the reservation station waiting for the result, that is false. So statement number 1 is false. Looking at the second statement in a speculative dynamic scheduled processor, we can issue an instruction if there is an empty reservation station, even though operands are not available.

So the moment that term speculative, speculative dynamic scheduled processor means it is a processor, which works with the help of a reorder buffer. Since we are using for speculation and out-of-order execution, the operands are being produced. And as on when the operands are been produced, these operands are temporarily stored the result of the operations are temporarily stored in the register called reorder buffer.

Now, sometimes certain entries in the reorder buffer may be filled early and certain other entries may be filled up early depending upon the order of completion of instructions. Now, once we decide that one instruction is no longer speculative, then we call it as a process called commit. So from the reorder buffer, from the top of the reorder buffer, one one instruction, the result of each of these instructions are being stored either into register or to the memory in-order to reflect the commit operation.

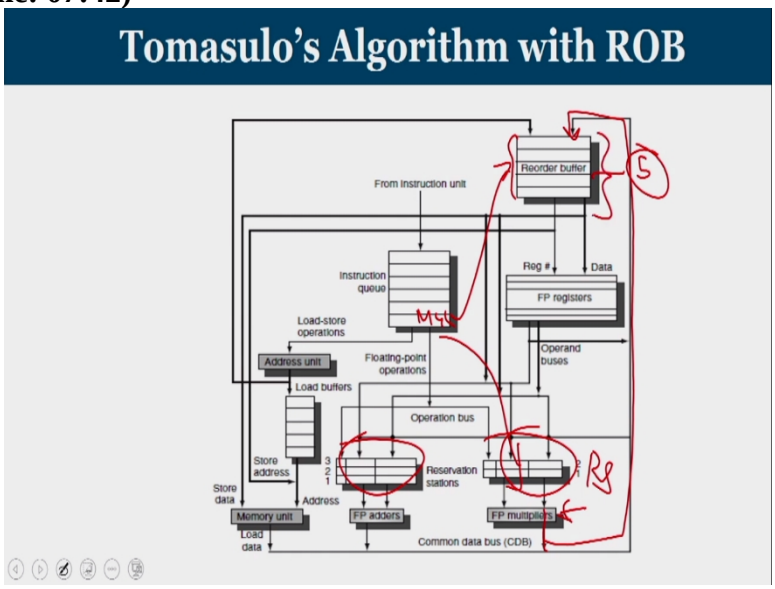
So at the time of issue also we have to make sure that we get an entry in the reorder buffer. So when you go to a speculative processor, after you fetch the instruction, before issuing into the reservation stations, before issuing into functional units, there are 2 things that you have to see, first is whether there is a free reservation station entry, and second aspect is whether there is an entry allotted to this instruction in the reorder buffer, both the things are being required.

Only if an entry you have in reorder buffer as well as an entry you have in the reservation station, then only an instruction can be issued. So at this point of time, whether any operand is available or not available is not a question because you can wait in the reservation station and as on when operands are available. When previous instructions complete, you can get it and then move forward.

But when we are not going to work with a speculative processor, that means if it is a normal dynamically scheduled super scalar processor, then reorder buffers are not there. So the issue is restricted to whenever you have space in the reservation station, then you can issue. Main difference between a speculative processor and a non-speculative processor is in the case of a speculative processor, reorder buffer is been involved.

So when the reorder buffer is been involved first I have to reserve a space in the reorder buffer. So if the reorder buffer is full, I cannot issue. So, issue is dependent on availability of space into the reorder buffer and availability of entry in the reservation station.

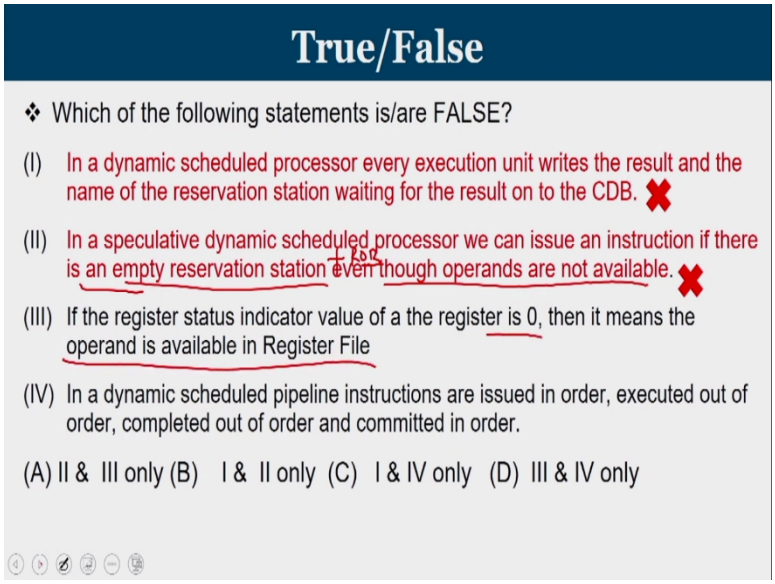
(Refer Slide Time: 07:42)



So this is generally the reorder buffer looks like so you need to have an entry here. At the same time, you need to have an entry in the reservation station. So this is the reservation station. So let us say I am going to have a multiplication instruction. So multiplication instruction has to be given to the multiplier. So, there should be a free to reservation entry as well as there should be an entry in the reorder buffer, why they enter in the reorder buffer is important.

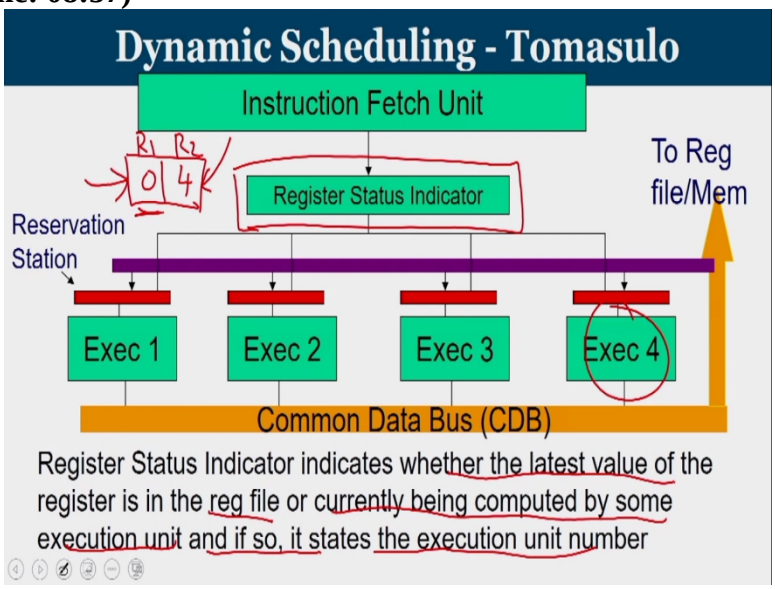
That means let us say my reorder buffer entry is 5 are been assigned to this multiplier as and when the multiplier produces the result, it will go into the fifth location in the reorder buffer. But the reorder buffer entry is not given as on when you produce the result there is no guarantee that there is a space that is made available.

(Refer Slide Time: 08:25)



So the statement that a speculative dynamic scheduled processor, we can issue an instruction if there is an empty reservation station even though operands are not available is not correct, because we can issue, if there is an empty reservation station plus a reorder buffer then only if both the cases are there then only we can say that statement is true. Moving on to the next one, if the register status indicator value of the register is 0 then it means that the operand is available in the register file.

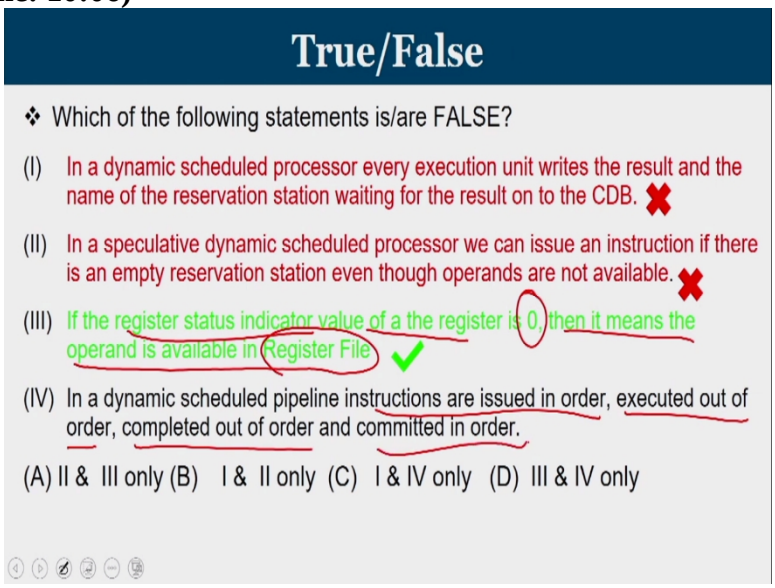
(Refer Slide Time: 08:57)



So, we have to understand what do you mean by register status indicator. So, for every register in a dynamic scheduled processor we have a registered status indicator, it indicates whether the latest value of the register is in the register file or currently being computed by some execution unit. And if that is the case, it says the execution unit number. So, the registered status indicator like what I am mentioned, there are 2 registers R 1 and R 2.

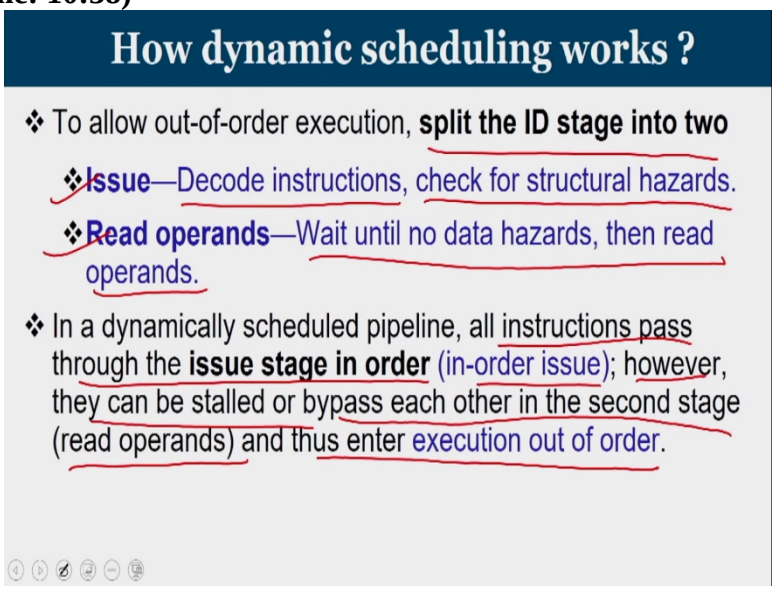
If registered status indicator is 0, that means the latest value of register R 1 is available in the register file. If this is being written as 4 then the latest value of register R 2 will be produced by execution unit 4. So, any non-zero value shows that the updated value of that register is not available in the register file, it is to be produced by some execution unit number. So it is execution unit number that has been written. So, whenever the value is 0, that means the latest value is available in the register file.

(Refer Slide Time: 10:06)



So, the statement if the register status indicator value of a register is 0, then it means the operand is available in the register file that is correct. If it is non-zero, then it is not available in the register file it will be produced by some execution unit. So statement number 3 is true. Moving on to statement number 4 in a dynamic scheduled pipeline instructions are issued in-order, executed out-of-order, completed out-of-order and committed in-order.

(Refer Slide Time: 10:38)



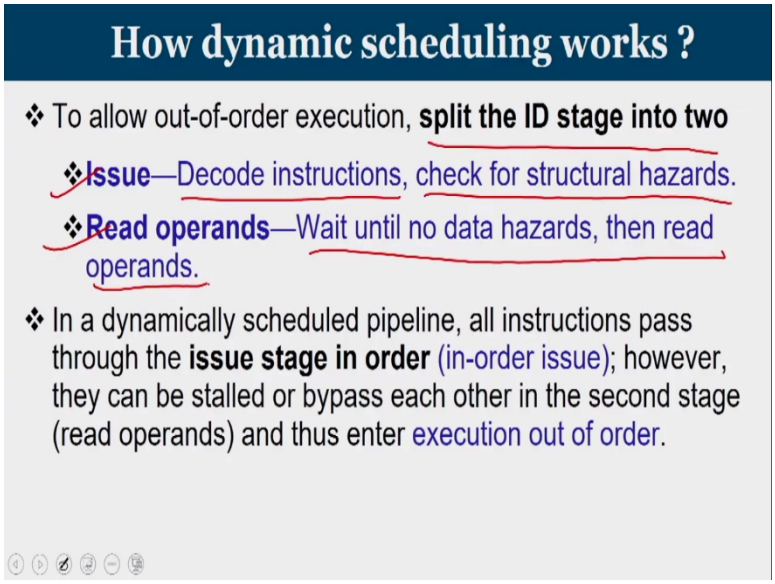
Let us try to see what do you mean by dynamic scheduling. In dynamic scheduling, we are splitting the ID stage into 2 like issue and read operands. So, in the issue, we decode the instruction and check for structural hazards. In the read operand stage you wait until there is no data hazard on the new operand. So in dynamically scheduled pipeline all instructions pass through the issue stage in order that is what is known as in-order issue.

However, they can be stalled or bypass each other in the second stage and the center in execution out of order. So, what we have to understand in the case of a dynamic scheduled processor is we have we are going to bring instructions and then you see whether the functional unit is available or not. If the functional unit is not available that we are stalling. If instruction any stall then all of other subsequent instructions are also stalled.

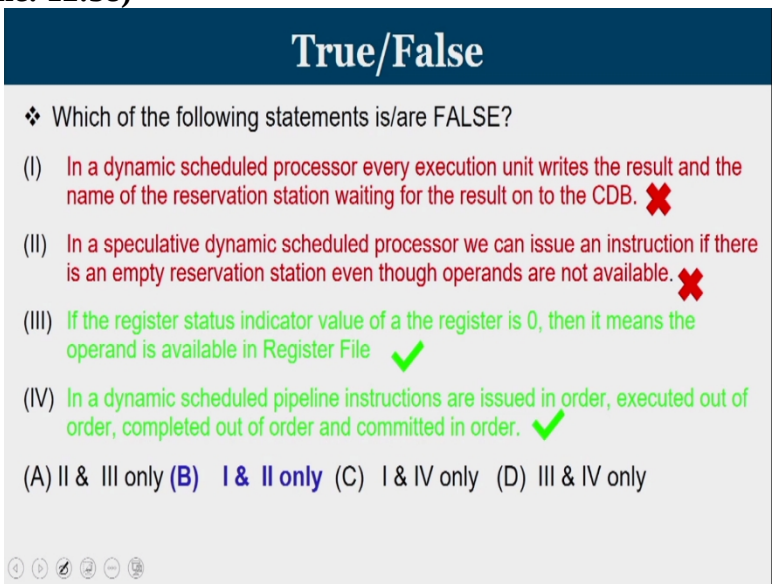
So, the process of assigning into the functional unit or process of making an entry into the reservation station is should be strictly in order and that is what is known as in order issue. So, issue happens in-order and then depending on the availability of the functional unit, depending on the availability of the readiness of the operands, you can enter into execution stage in whatever order that you want. So, that is called execution out-of-order.

When you start execution out-of-order instructions can complete also in out-of-order, because the beginning of execution is not in sequence. So completion of execution also may not be in sequence. But then we are going to write the result in the reorder buffers and from the reorder buffer updating into memory as well as the registers happened in order. So that process is called commit. So to summarize, issue is in order, execution is out-of-order, completion is out-of-order, but commit is in-order.

(Refer Slide Time: 12:37)

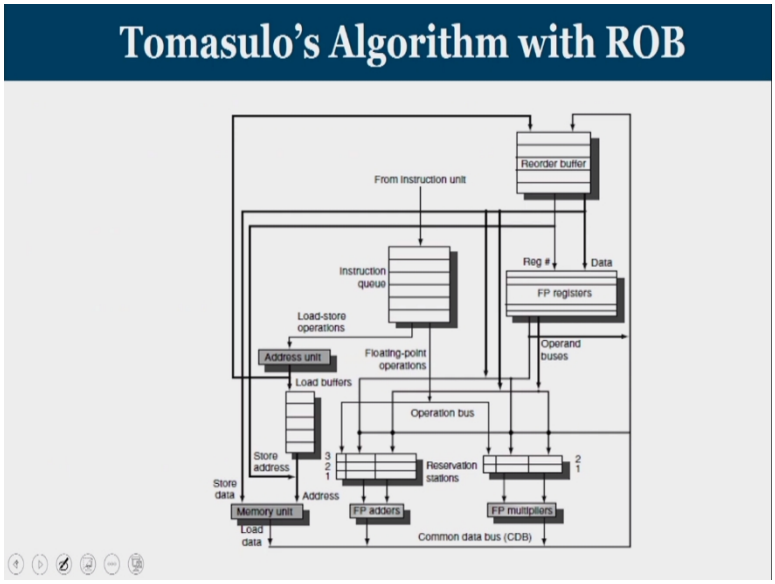


(Refer Slide Time: 12:38)

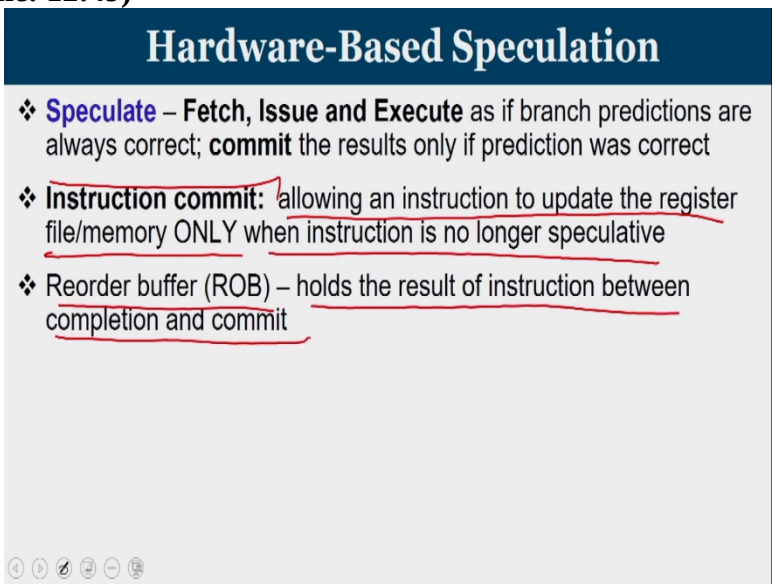


So the statement a dynamically scheduled pipeline instructions are issued in order, executed out-of-order and completed out-of-order, but committed in order is true.

(Refer Slide Time: 12:48)

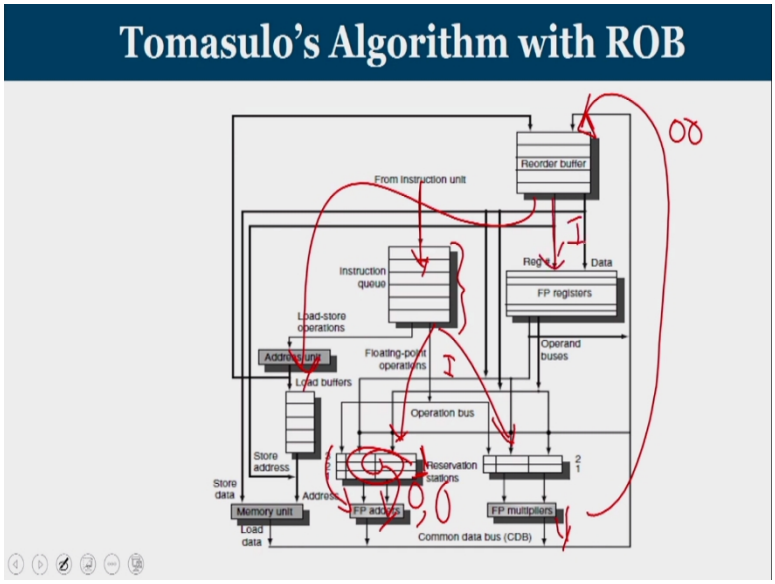


(Refer Slide Time: 12:49)



So when you have to look at instruction commit, it allows an instruction to update the register file or memory only when it is the instruction is no longer speculative, until then, instructions are stored in reorder buffer it holds the result of an instruction between the completion stage and the commit stage.

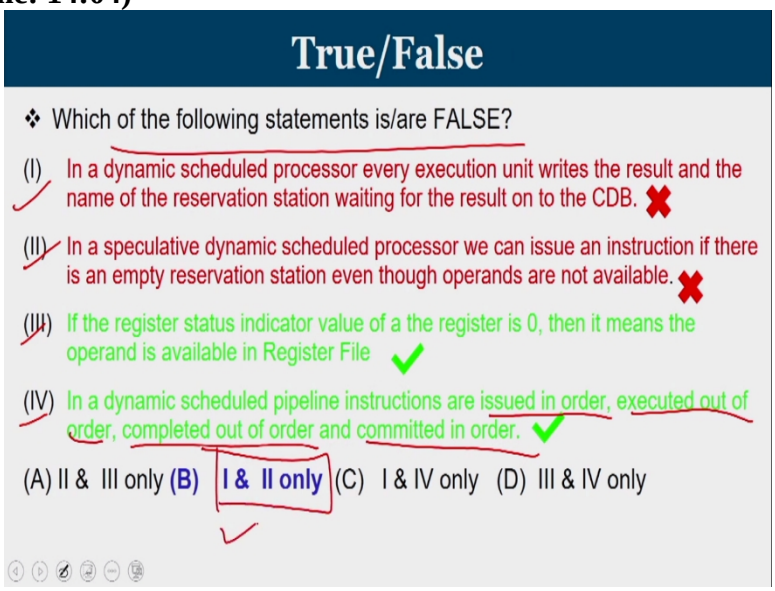
(Refer Slide Time: 13:05)



So, we are going to bring instructions into the instruction queue, entering into the reservation station that is in order. Now, when you are waiting in the reservation station and entering into the functional unit and starting execution that portion is out-of-order, because some of the instruction waiting in the reservation station may have all the operands ready so they may enter into execution. Some of the instructions may be waiting first in the reservation station may not have the operands ready so it is still waiting for the operand.

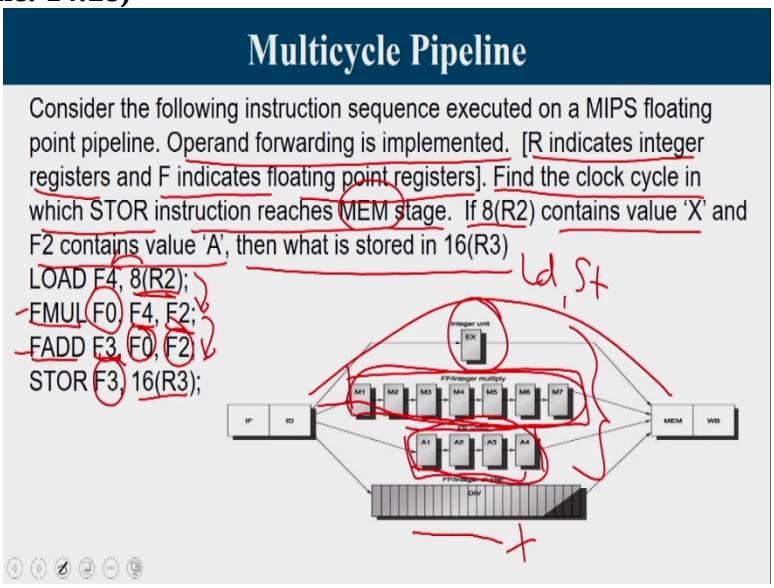
So all others which are after that, in the reservation station can bypass itself to enter into the functional unit. So execution is out-of-order, once execution is out-of-order and our time at which you write into the common data bus that is also out-of-order. So the entry into reorder buffer is out-of-order. But entry from reorder buffer all the way to memory and to the registers that happens in order and that is called commit.

(Refer Slide Time: 14:04)



So the summary that is what has been mentioned. The dynamically scheduled pipeline are issued in order, execute out-of-order, completed out-of-order and committed in order. So from this we can see that statement number 1 and 2 is false. And statement number 3 and 4 is true. So, the correct answer which of the following statements are false in this 1 and 2.

(Refer Slide Time: 14:28)



We now move into the next problem that is a multicycle pipeline. Consider the following instruction sequence executed on a MIPS floating point pipeline. Operand forwarding is implemented. R indicates integer register and F indicates floating point registers. Find the clock cycle in which store instruction reaches its MEM stage. If 8 of R2 contains a value X and F2 contains a value A then what is stored in 16 of R3. Let us try to understand what this instruction is all about.

Load F4, 8(R2)

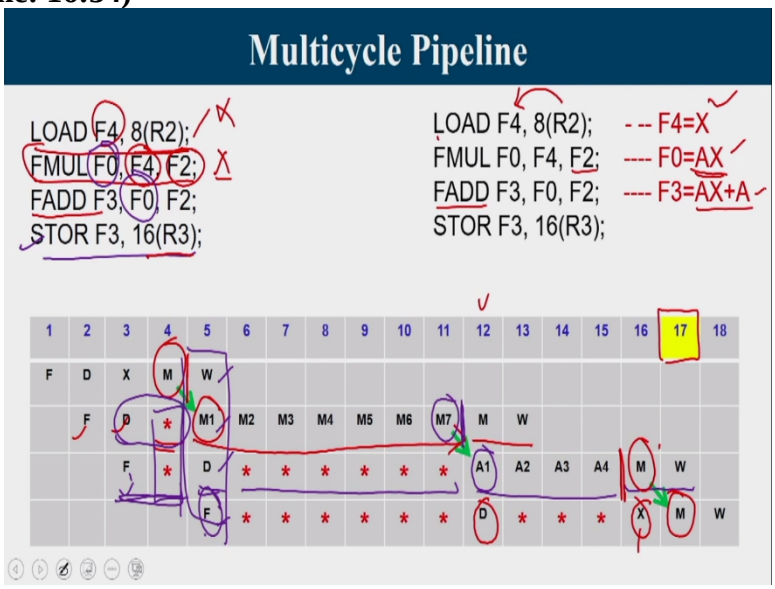
so from 8 of R2 will be giving you a memory location. Contents R2 + 8, that is an address from where I am going to load into F4. Then Fmul multiplication operation F4 whatever you have loaded and then that is multiplied with the value F2. So F4 into F2 has been computed and that result is stored in F0. Now, we have an FADD. So, whatever is F0, again you are going to add the F2 to F0.

So, there existed dependency between the first 2 instructions between the second and the third instruction. Now you have the result that is available in F3. And that result is going to be returned to 16 (R3). We should know that since we are using a MIPS floating point

pipeline, this is the floating point pipeline, where this is the part where in the integer operations are being handled and multiplication floating point.

Multiplication typically takes 7 cycles in the execution unit and the floating point adding will take 4 cycles in the execution unit. In this question anyway division is been not given. So it is not relevant to us. So these are the 3 units that work. So, both the load as well as the store statement as making use of integer unit for computation of effective address. FMUL and FADD are going to use these 7 stage multiplier and the 4 stage adder that is been given in the diagram.

(Refer Slide Time: 16:34)



So, these are the 4 instruction. Now let us try to understand what are the clock cycles in which, these operations are carried out. So first is a load operation it happens through the integer pipeline. So fetch, decode, execute, MEM 1 write back happens in the first 5 cycles of fetch. Now we have a data dependency the multiplication operation will get F4 only at this point, so, this is the point at which the contents to be returned to F4 is been taken from memory.

So, even though I start by fetching in cycle number 2, decoding in cycle number 3, I cannot say this is floating point multiplication. So, rather than integer unit it is going to make use of the 7 stage multiplier M1, M2 etc., up to M7. M1 stage or floating point multiplier can happen only after the MEM stage of the load instruction. So, there is operand forwarding from the output of MEM stage to the input of M1 stage.

So we have a 1 cycle stall, and then it continues all the way up to M7 and MEM and write back. So, FMUL, the second instruction is completing at clock cycle number 12. Now, look at the third instruction, it is a floating point ADD again there is a dependency on this F0. So, this F0 is the result of the floating point multiplication and the second F0 is going to be the source operand. So there is a dependency.

So, it is only at this point the result is available F4 into F2. So, I am going to fetch the third instruction here. Since there is already a stall there, I can enter to decode stage only at this point. And then all the cycles I have to wait because of the raw dependency my previous result is still not ready, the result is ready at the end of the 11th clock cycle. So from output of M7 to the input of A1 there is an operand forwarding and A1, A2, A3 and A4 4 cycles it will take to complete the floating point ADD followed by MEM write back.

So, this instruction will complete at clock cycle number 17. Coming into the last instruction into the store operation, you can do the fetching only at clock cycle number 5, because a status like this shows that the third instruction was fetch at clock cycle number 3, but then it cannot move to decode state. So it will be there in the same state for one more cycle because the decoder is not free, decoder is busy, with the second one due to the stall.

So fetching happens only at this point or else when you look vertically, all the entries should be different, this is write back this, this is one stage of execution, this is decode, this is fetch. That is a way it is. So, if at all I start fetch here, then this decode will come here. So decode and decode is going to clash. So, that should not happen vertically if you look no functional unit has will be repeated.

Now again operand forwarding means you are going to fetch here, but the decode happens only when the previous instructions reaches start execution and when the previous instruction start at MEM stage then only I will perform execution. So, at this point 16(R3) is being computed that is done through the integer unit, and then you will get the result that is to be stored will be available to you from this point onwards.

So, you can forward from this unit, that is from the output of MEM unit that is 1 option to the input of MEM unit. So that the value is available for, for storing. So the question is asking at which cycle the store instruction will perform the MEM operation. So, clock cycle number

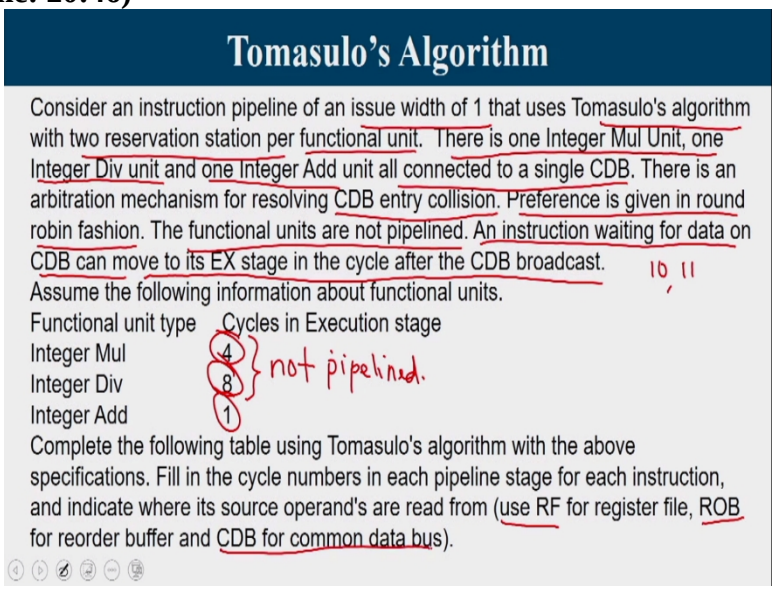
17. Now in the question that has been asked if 8(R2) is X, and the content of F2 is A so the first one you will get the value X into F4 and then you are going to perform a multiplication where F2 is A.

So F4 into F2 that is actually X. And when we perform this ADD operation, then the effective result is

$$X + 8$$

That is the result that you are going to get. So that is all about this question.

(Refer Slide Time: 20:46)



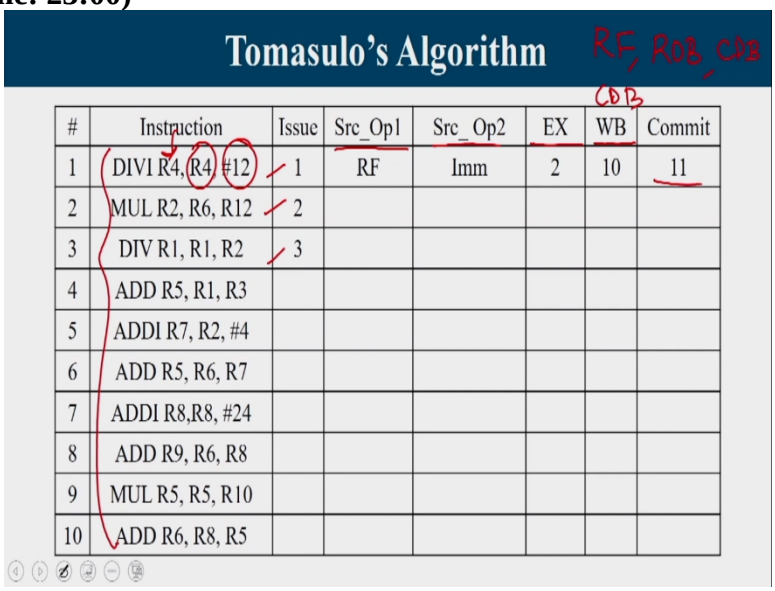
Moving on to the next one. This question is all about Tomasulo's algorithm. Considering an instruction pipeline of an issue width of 1 that means every cycle I can issue only one instruction that uses Tomasulo's algorithm with the 2 reservation station for a functional unit. So, in the input of a functional unit, 2 instructions can read. There is one integer multiplication unit, one integer division unit and one integer add unit all connected to the single common data bus.

There is an arbitration mechanism for resolving CDB entry collision. So when there are multiple functional units that are being connected to the CDB not to functional unit should write the result together. So, if they are write to CDB at the same time, then that create collision. So, there is a mechanism by which it resolve CDB entry collision. Preference is given in round robin fashion. The functional units are not pipeline is very important, even this adder, multiplier and integer unit, none of them are pipeline.

Since they are not pipeline when you have subsequent operation if the same division operation is coming one after another second division operation can start only if the first division is over. But at the functional unit is pipeline than it is not a problem. So here functional units are not pipeline and instruction waiting for data on CDB can move to its EX stage in the cycle after this CDB broadcast. So if somebody writes into CDB at 10 only in 11 the instructions waiting can enter into execution unit.

Assume the following information about the functional units for the integer multiplier will take 4 cycles, integer divider will take 8 cycles, integer adder will take 1 cycle and these 2 are not pipeline that is very important. Complete the following table using Tomasulo's algorithm with the above specification, filling the cycle number in each pipeline stage for each instruction and indicate whether it source operands are read from register file, reorder buffer or CDB.

(Refer Slide Time: 23:00)



#	Instruction	Issue	Src_Op1	Src_ Op2	EX	WB	Commit
1	DIVI R4, R4, #12	1	RF	Imm	2	10	11
2	MUL R2, R6, R12	2					
3	DIV R1, R1, R2	3					
4	ADD R5, R1, R3						
5	ADDI R7, R2, #4						
6	ADD R5, R6, R7						
7	ADDI R8,R8, #24						
8	ADD R9, R6, R8						
9	MUL R5, R5, R10						
10	ADD R6, R8, R5						

So, this is been given these are the set of instructions that are to be executed. Now, majority of them are instructions which carries two operands this division multiplication adding and are so source operand one. So, this is the source operand 1 and the second 1 is source operand 1 and the first one that you see is the destination.

So from where will I get the source operand 1 that can be 3 possibilities you can either get from the register file direct or it is available in the reorder buffer. That means a previous instruction has produced the result, but the previous instruction was not completed or you can

get it from CDB that means the previous instruction is not yet over, previous instruction is under progress. So as on when the result is produced I can get it. So, you will write.

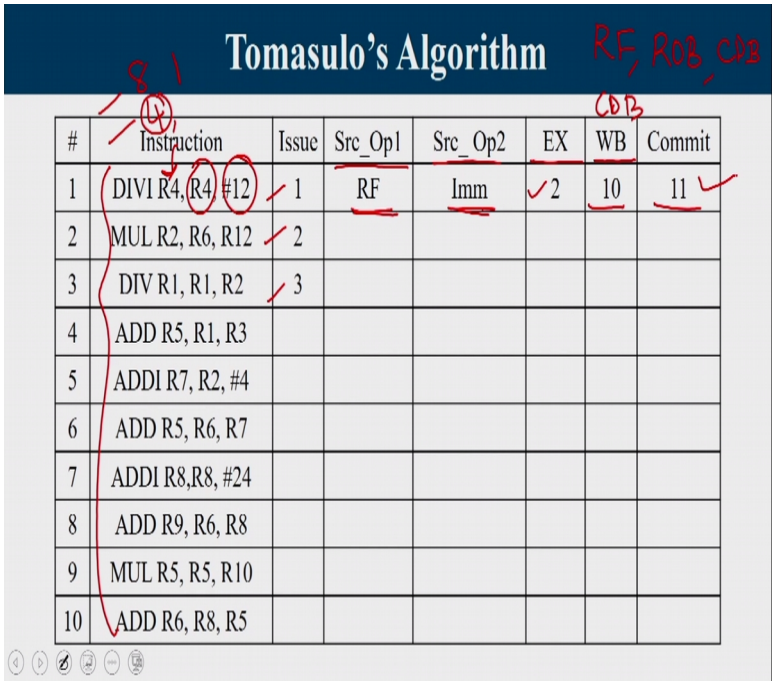
So, this is been showing that the first 3 instructions are been issued one by one in every cycle. So we have an issue bandwidth of 1 and then you have to write to the time at which the execution happens, execution begins rather. When the write back into so this is basically CDB write back and where are you going to commit. We have to understand 2-3 things here. So, the first one is when an instruction is coming, we have to issue the instruction and issue always happen in-order.

So, if an instruction cannot be issued, all of other subsequent instruction also will be stalled. And what are the conditions to issue. We need to have an entry in the reorder buffer and we need to have an entry in the reservation station. Here the size of reorder buffer is not mentioned in the question. So we can assume that reorder buffer is having sufficient space because of lack of space in reorder buffer we are not stalling.

But for each functional unit the number of the reservation station is 2. So at most only 2 instructions can wait in a given functional unit's reservation station. So, if your third instruction is coming, the first instruction is not yet complete, third instruction cannot be issued. So issue will be stalled. And when I issuing N, I can start execution in N + 1 if operands are ready and based upon the length of or the latency of the functional unit and complete the execution and then the commit always has to be in order.

So issue is in order execution can be out-of-order, CDB write back can be out-of-order. But reorder buffer commit should always be in order. Now let us see how the values are being populated in this table.

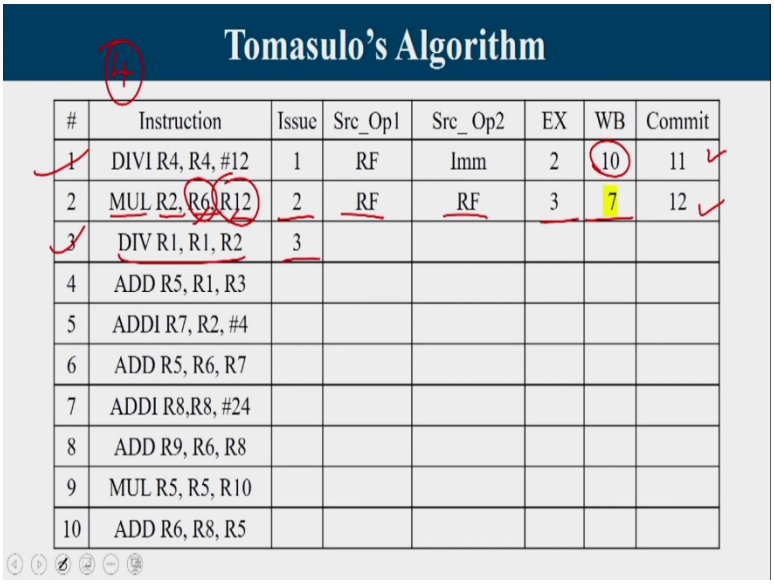
(Refer Slide Time: 25:29)



It is been mentioned that the division unit has a latency of 8, multiplier unit has the latency of 1 and adder has a latency. The multiplier has a latency of 4 and an adder has a latency of 1. Now, the first instruction is issued at 1 and the operand R4 at this point R4 will be available in the register file, because it is very first instruction value is there and second operand is immediate, because the value has told represents immediate.

So when issue one the operands are ready both operands are ready. So I can start execution in cycle number 2, it is a division operation, it takes 8 cycles. So, cycle number 2, 3, 4, 5, 6, 7, 8, 9 at the end of 9 clock cycle the execution is over. So at the 10th clock cycle, it is going to write in to CDB. So, in the 10th clock cycle, when you write something into CDB, it reaches reorder buffer and in 11 you are going to commit.

(Refer Slide Time: 26:30)



#	Instruction	Issue	Src_Op1	Src_ Op2	EX	WB	Commit
1	DIVI R4, R4, #12	1	RF	Imm	2	10	11
2	MUL R2, R6, R12	2	RF	RF	3	7	12
3	DIV R1, R1, R2	3					
4	ADD R5, R1, R3						
5	ADDI R7, R2, #4						
6	ADD R5, R6, R7						
7	ADDI R8,R8, #24						
8	ADD R9, R6, R8						
9	MUL R5, R5, R10						
10	ADD R6, R8, R5						

Now, we will move into the next instruction. The next instruction is

mul R2, R6, R12

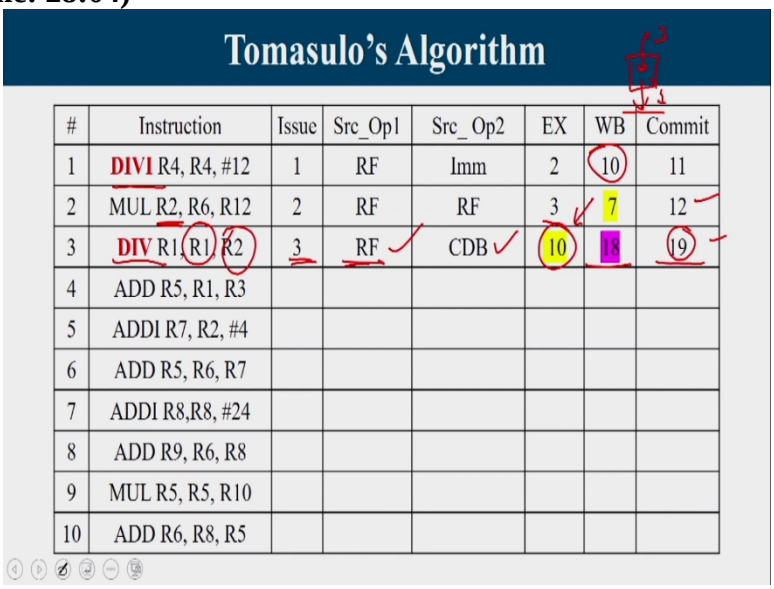
So R6 and R12 are the operands. Both the operands are now in the register file. Because no other instruction before it is going to produce a result either to R6 or to R12. It is issued into, it is a multiplication. So multiplier can directly start because both the source operands are available. Since it is issued into the execution started 3.

Since the execution started 3 cycle number 3, 4, 5 and 6 will be used for the multiplier. Because the latency of the multiplier is 4 in unpipeline unit. So at 6 multiplication is over, at 7 I am going to write into the CDB. Even though the result reaches their reorder buffer at clock cycle number 7, the commit will happen only at 12, because at 11 only the previous instruction is completed. So I can commit only in the next cycle.

There is one more question of how many instructions, I can commit, it is generally, it is equal to the fetch bandwidth. Since I am issuing only one instruction per clock cycle, commit also can be 1. So in this case, we will assume that we are committing only 1 instruction every clock cycle. Now I move to the third instruction you have to understand third instruction is also division and division unit is unpipelined.

So only the previous division when is a previous division getting over previous division is getting over only at the 10. So even though I issue because of, there are 2 reservation station, I can issue the division.

(Refer Slide Time: 28:04)



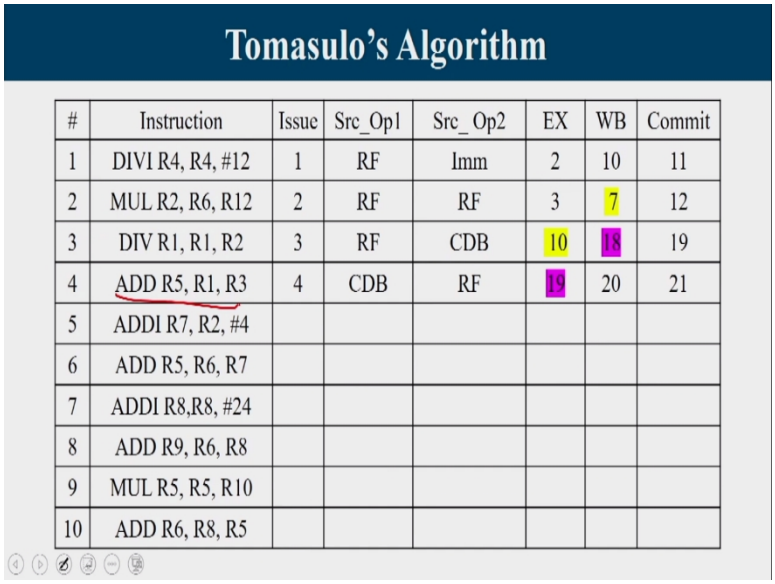
#	Instruction	Issue	Src_Op1	Src_ Op2	EX	WB	Commit
1	DIVI R4, R4, #12	1	RF	Imm	2	10	11
2	MUL R2, R6, R12	2	RF	RF	3	7	12
3	DIV R1, R1, R2	3	RF	CDB	10	18	19
4	ADD R5, R1, R3						
5	ADDI R7, R2, #4						
6	ADD R5, R6, R7						
7	ADDI R8,R8, #24						
8	ADD R9, R6, R8						
9	MUL R5, R5, R10						
10	ADD R6, R8, R5						

But when I issue the division, this is the problem, there is one more division already there in pending. So the first operand for this division is R1 it is in the register file, second operand is R2 but you have to know that R2 value will be produced only by the second instruction. So when I issue that is it clock cycle number 3, second instruction is still in progress. I will get the result only at 7. In 7 the value of R2 is available.

But even though in 7 I have first operand R1 is already available in register file, I am taking it at clock cycle number 7 from CDB I get the second operand R2, but the functional unit is not free. So I have to wait this is the dependency. In 10 only division unit will write the result so in 10 only I can start. So in 10 this let us say this is a division unit, it is going to write into CDB at the same time a new one can end up.

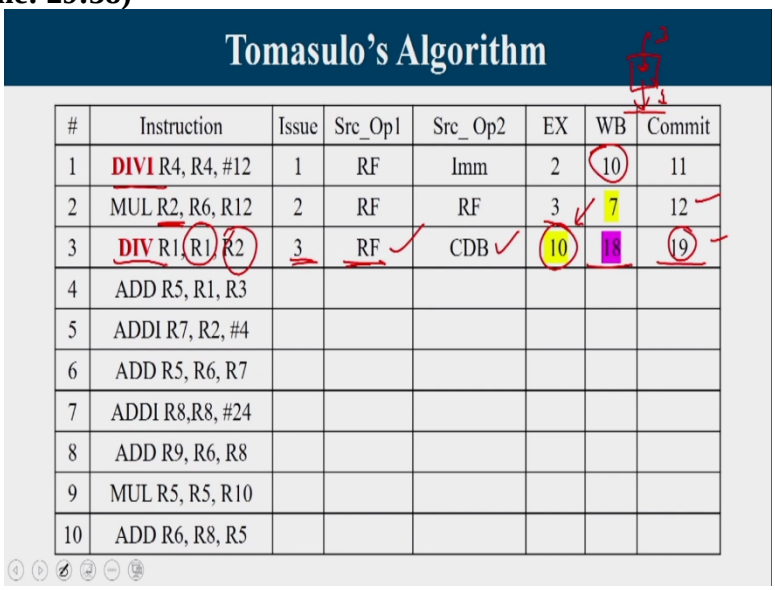
So instruction 1 result is return to CDB and instruction 3 will enter the division unit. So, at 10 the division starts it takes up to clock cycles 17, 8 cycles. So only in clock cycle 18, I can write the result of the division. So naturally in 19 only I can commit. Why this 19, because from 12 to 19 no other instruction is going to reach the reorder buffer. So, in 18 the result of division is return into reorder buffer and 19 we can commit.

(Refer Slide Time: 29:29)

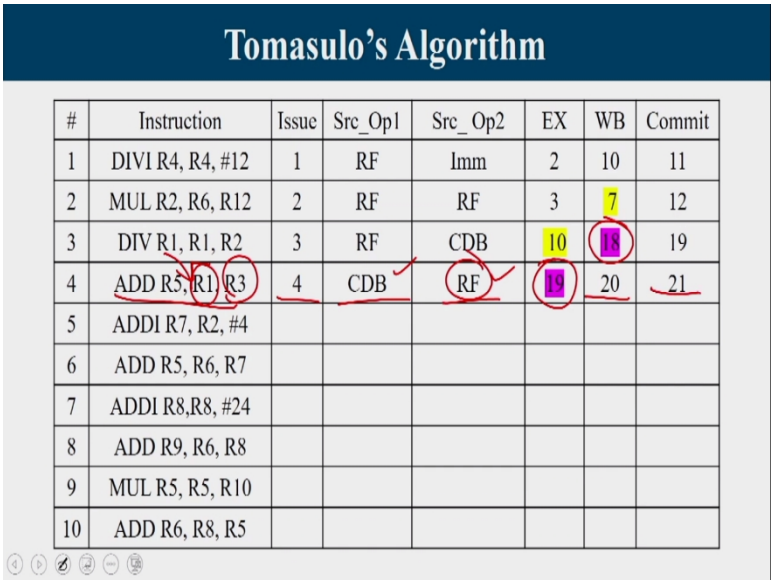


Moving further and going into the next instruction add, there are 2 reservation station entry but I have only one adder. So, in the previous table upto 3 the issue numbers was given now we have to find out when issue can happen.

(Refer Slide Time: 29:38)



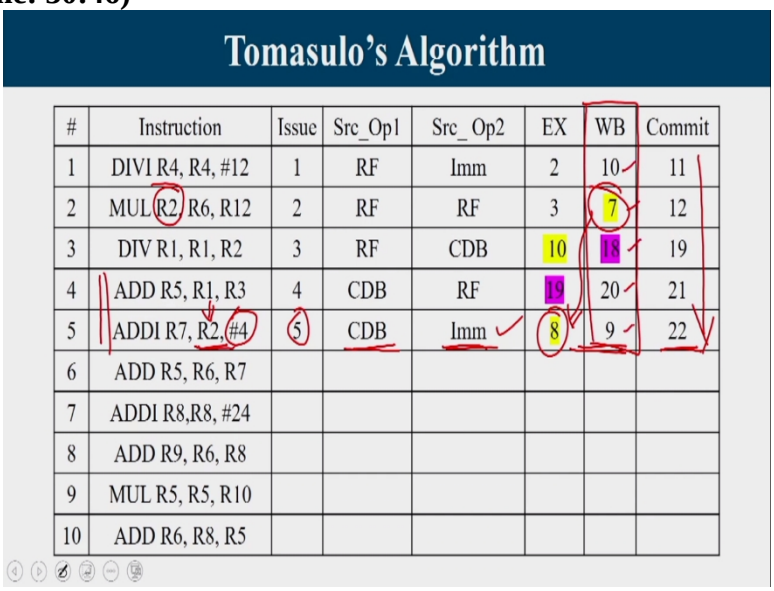
(Refer Slide Time: 29:43)



So, since all the instructions previously are issued, and the reservation station is available in the order, I can issue it at 4, what about R1, R1 is produced only by this division. So, I have to wait in CDB, What about R3, R3 normally, so R3 is there in the register file. So, first operand, I will get from CDB, R1 second operand, I will get from the register file. But when I can I start execution because second operand is available, but first operand is not available.

First operand will be produced from the CDB. So, only at clock cycle number 18, the result is written into CDB for broadcasting. So in 19, I can start execution that has been given in the question. So if a value is returned to CDB broadcast at clock cycle number N. So every instruction waiting for this operand can start execution in N + 1. Since it is an add operation, if you start execution at 19, in 1 cycle execution is over at 20, I can write the result and 21, I can commit.

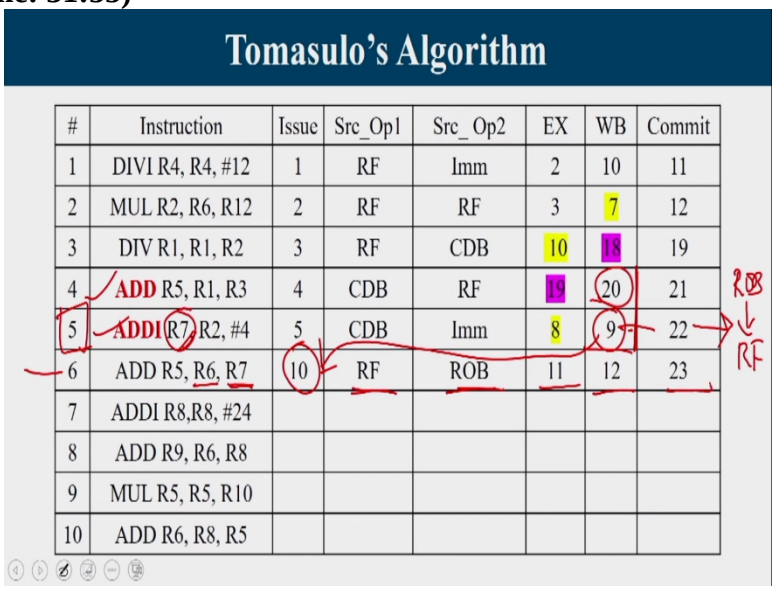
(Refer Slide Time: 30:46)



Now, I moved to the second add operations since I have 2 reservation stations, I can issue the second add also, so I can issue in the very next cycle, issue bandwidth is 1. R2 is the first operand. So, we have to understand that R2 is produced by instruction number 2. So to get R2 I have to wait in CDB, second operand is an immediate value. Now, when will CDB produces the result for R2, we know that R2 value is available only at 7.

So, this is the dependency at 8 I can start execution. So, one of the operand is available at 8 and the second operand is available readily. So, it is waiting for the operand to get. So, at the end of clock cycle number 7 multiplication produces the result and I can start execution at 8 and it takes only 1 cycle. So, at 8 I start execution. At 9 I complete the write back and at 22 I can commit. So you have to understand that this should be strictly in order.

(Refer Slide Time: 31:55)



But if you look at the right back you can see that it is out-of-order some of the instructions are completing early. Now I move into instruction number 6. Ideally I can issue it that clock cycle number 6, but if you see that already these 2 add instructions are there in the reservation station and we know that there is no there is only 2 entry in the reservation station and first add will get over only a 20. Second add will get over only at 9. So, in that case I cannot issue because of lack of reservation station. So I have to wait till it is 10.

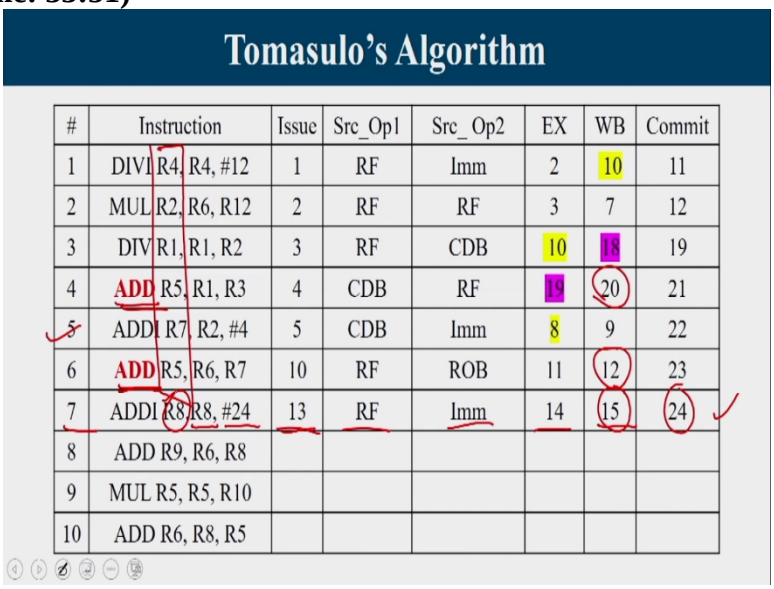
So, how I get this 10 means only at 9, the reservation station entries been free. So till 9 in the set this instruction number 5 is occupying the reservation station and it will be relieved from the reservation station when I write the result into the CDB. So at 10 only I can issue, so

since I can issue instruction number 6 at 10 everybody after that also will be issued after 10 because issue is in order.

Nobody issue at 10, what about R6, R6 will be there in the register file. What about R7, R7 value will be produced by him at clock cycle number 9. So when I issue a clock cycle number 10, I should know that R7 value is available in the reorder buffer. So take it from the reorder buffer, because from clock cycle number 9 up to clock cycle number 22 the value to be returned to R7 is available in reorder buffer at 22, from reorder buffer you are going to write into register files.

Because instruction committed. So only if an instruction is committed, then only the registered file gets updated. So till execution is over. So your execution gets over a clock cycle number 9, from 9 up to 22, it is available in the reorder buffer. So it is issued at 10 it can start execution at 11 and it will complete at 11 itself to write back and 23 you are going to commit.

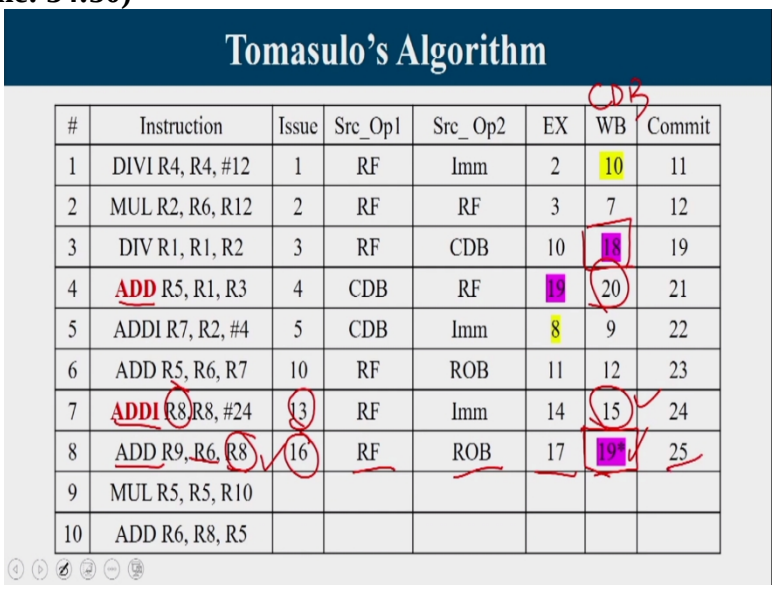
(Refer Slide Time: 33:51)



Now we move to the next one this add instruction. Now we have to know that this add is over, but this add is still pending. So that will consume 1 entry of the reservation station. This add is pending until mid-cycle until clock cycle number 12. So, I have one add which will be completed only at 20, have the second add which will be completed only at 12. So till 12 the reservation station is not free. So I can issue only at 13.

In 13 when I issue my first operand R8, R8 is nowhere coming in the destination of any of the previous instruction. So, R8 is available in register file and second operand is immediate. So, I can issue at 13, execution start at 14 and 15 I complete and then all 24, I can commit. So from clock cycle number 15, till clock cycle number 24, the value of R8 is available in the reorder buffer.

(Refer Slide Time: 34:50)



#	Instruction	Issue	Src_Op1	Src_Op2	EX	WB	Commit
1	DIVI R4, R4, #12	1	RF	Imm	2	10	11
2	MUL R2, R6, R12	2	RF	RF	3	7	12
3	DIV R1, R1, R2	3	RF	CDB	10	18	19
4	ADD R5, R1, R3	4	CDB	RF	19	20	21
5	ADDI R7, R2, #4	5	CDB	Imm	8	9	22
6	ADD R5, R6, R7	10	RF	ROB	11	12	23
7	ADDI R8, R8, #24	13	RF	Imm	14	15	24
8	ADD R9, R6, R8	16	RF	ROB	17	19*	25
9	MUL R5, R5, R10						
10	ADD R6, R8, R5						

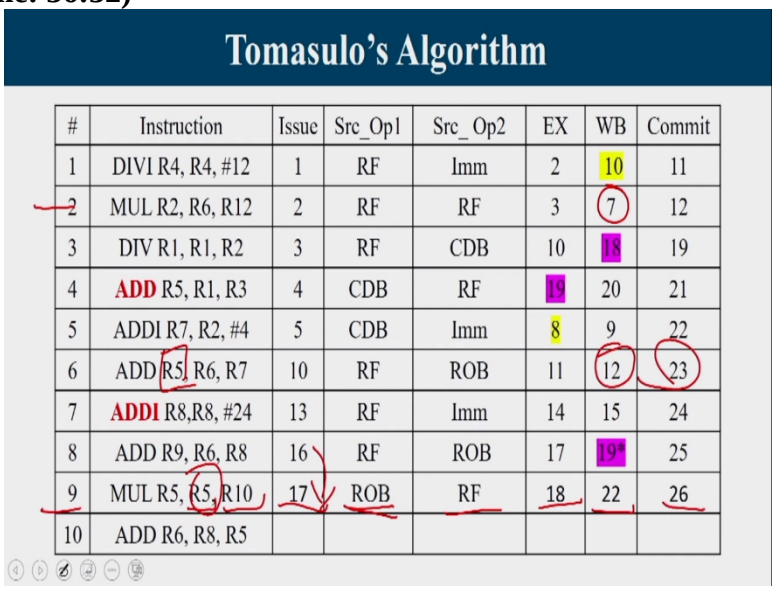
Moving further I am moving to the next add instruction. The next add instruction also I have to understand that we will try it at 14, previously I had issued at 13, can I issue at 14, if you look at clock cycle number 14, you know that this add is still pending, this add is also still pending. So it will complete only at 16, the second add will complete only at 15. So, in 16 only I can issue the new instruction.

So, the concept of issuing is since I have only 2 reservation station, every issue I have to see that whether this 2 instructions are already there in the reservation station. I can issue only if there is an entry that is been free. So in this case still 15 I have to wait. So, in 15 your reservation station is getting free. So in 16 I can start. Now R6 and R8. R6 is there in register file we have to see in R8, R8 is the result of the previous operation, but if you look at clock cycle number 16, you should know that by 15 R8 value is there and reorder buffer.

So you take it from the order buffer. So, 16 I issue 17, I come I execute by end of 17 it should be over, I can write into 18 into CDB. Now, you know that already somebody is writing into CDB at clock cycle number 18. So this also cannot happen at 18, this is called CDB conflict. So it is already mentioned in the question that CDB conflict is been resolved. So I will write

only in the next cycle, will be holding the result till 19. And then I write CDB at 19 committing at 25.

(Refer Slide Time: 36:32)

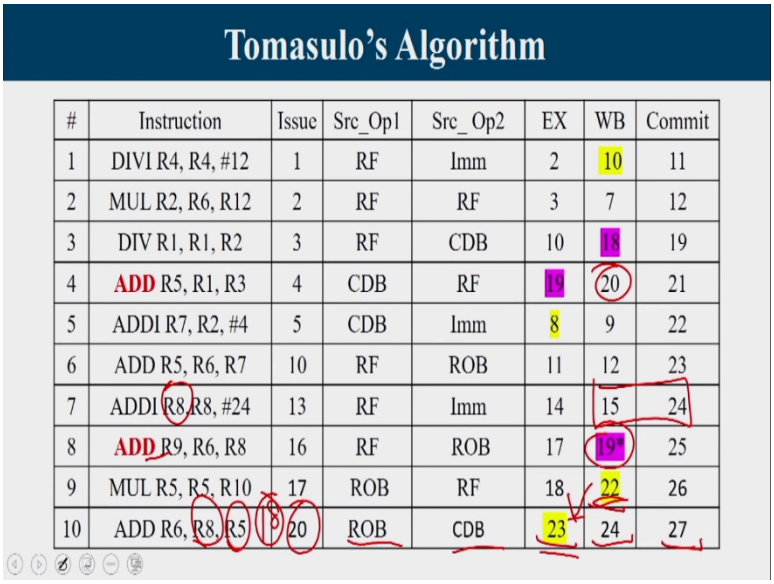


#	Instruction	Issue	Src_Op1	Src_ Op2	EX	WB	Commit
1	DIVI R4, R4, #12	1	RF	Imm	2	10	11
2	MUL R2, R6, R12	2	RF	RF	3	7	12
3	DIV R1, R1, R2	3	RF	CDB	10	18	19
4	**ADD** R5, R1, R3	4	CDB	RF	19	20	21
5	ADDI R7, R2, #4	5	CDB	Imm	8	9	22
6	ADD R5, R6, R7	10	RF	ROB	11	12	23
7	**ADDI** R8,R8, #24	13	RF	Imm	14	15	24
8	ADD R9, R6, R8	16	RF	ROB	17	19*	25
9	MUL R5, R5, R10	17	ROB	RF	18	22	26
10	ADD R6, R8, R5						

The next one is a multiplication instruction, you have to see what is the status of multiplication, the first multiplication get over by 7. So anything after 7 you do not have a structural hazard. So here you're issuing the multiplication at 17. Why just after 16 only I can try, why there is no other check because as far as multiplication is concerned, the reservation station is always available.

What about R5, R5 will be produced that clock cycle number 12. So from 12 to 23, value of R5 is available and reorder buffer. So I am assuming it 17, so at 17 if you look at then the content of R5 is available and reorder buffer what about R10 nobody is producing, so R10 is there in register file. So 17 issue happens, 18 execution happens and then it is a multiplication operation. So 18, 19, 20 and 21- 4 cycle to complete multiplication, 22 the result has been return to CDB and then commit happen from 26.

(Refer Slide Time: 37:29)



Now you come to the last add operation. So ideally, it will try whether it can get issued at 18. But if you look at 18, we can see that this is still pending, it will be happening only at 20. And what about this adder, that is going to write the result only at 19. So at 19 only, you are going to write the result. So in 20, I can issue the new instruction. So adding happens after that, so issue of adding at 20 and then source operand is, reorder buffer is holding the value of R8 because this is the result that is being produced.

So from cycle number 15 all the way to cycle number 24, the value of R8 is in reorder buffer, what about R5, at clock cycle number 20, R5 is still under process. At 22 only I will get it ready. So the second operand is in CDB. So there is a dependency between them. So 22 will produce the results and 23 can start execution, it is an add operation 1 cycle 24 you write the result and 27 you are going to commit.

So with this one, we have filled up the table. It is a pretty lengthy exercise. But once you try to solve similar tables like this, you will get a clear idea of how Tomasulo's algorithm works. What are the ways in which operands are waiting, So operand can wait for a result that is being generated in CDB by some other functional unit. But if in some cases you are going to read from the register file, because already the value is available.

In some cases, the result is already produced by the previous instruction that is completed, but instruction is not committed since the instruction is not committed, then we may have to wait for it. So, with this we come to the end of this tutorial and some lengthy exercises be

solved in this tutorial. So when you move to advanced computer architecture concepts, these kind of problems are needed for deeper understanding.

I hope the tutorial session was useful for you in understanding dynamics scheduled processes, speculative processes and Tomasulo’s algorithm. Try to work out more number of problems like this which is given in the textbook. Thank you.