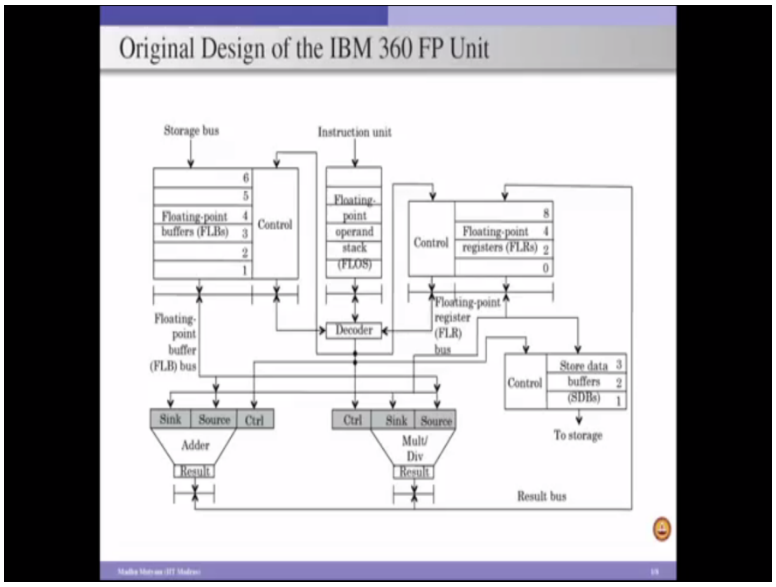
Computer Architecture

Prof. Madhu Mutyam
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Module – 07
Lecture – 25
Tomasulo Algorithm

So, as part of this module we are going to discuss Tomasulo algorithm. And this Tomasulo algorithm is the basis for the modern Superscalar processes. In other words the Tomasulo algorithm provides the basis for dynamic scheduling and so before discussing Tomasulo algorithm, we first discuss the floating point unit design considered in IBM 360.

(Refer Slide Time: 00:44)


Original Design of the IBM 360 FP Unit

Storage bus
Instruction unit
Floating-point buffers (FLBs)
6
5
4
3
2
1
Control
Floating-point operand stack (FLOS)
Control
Floating-point registers (FLRs)
8
4
2
0
Floating-point buffer (FLB) bus
Decoder
Floating-point register (FLR) bus
Control
Store data buffers (SDBs)
3
2
1
To storage
Sink
Source
Ctrl
Adder
Result
Ctrl
Sink
Source
Mult/Div
Result
Result bus

So, the original design of IBM 360 floating point unit is like this. It consists of 2 floating point units - one is floating point adder, the other one is floating point multiply and div unit. And it consists of the set of register files. So, one is FLB - floating point buffers the other one is floating point registers - FLR and the third one is store data buffers - SDB. So here, they have considered the memory addressing modes. So, as a result floating point instruction can take operands as memory operands or it can write the data or the destination operand as a memory operand.

So, that is a reason why, so, they have considered this floating point buffers which reads the value from the memory, using this storage bus and whenever instruction produces the result and if the result needs to be written to the memory, then they use this storage, the store data buffer SDB to write to the storage or the memory. And in this particular design, so, there is a result bus connecting to these 2 floating point units and as and when the floating point unit computes its operation then the result will be placed on the result bus.

This result bus is connected to the FLR - the floating point registers. So, the data will be updated in the floating point register and from there if the data needs to be written to the memory, then it will be sent to the store data buffer using the separate connection and similarly, if the instruction wants to read the data from the memory, because from the memory we are going to read the data onto this FLBs. So, from FLB we are going to read the data to the corresponding functional units. So, as a result by using these, the set of registers, the FLBs, FLRs and SDBs, so we can implement the memory addressing modes using register-register type of the instructions.

So, all instructions which have memory addressing modes or memory operands can be translated into the register operands. So, because like by exploiting these set of register files, we always read the data from these register files or write the data to these register files. Effectively our memory addressing modes are implemented internally by using these registers. This instruction unit reads instructions from the program, in the program order, one at a time, but whenever there is a floating point instruction, then those floating point instructions will be placed in this separate stack, that is called as a floating point operand stack.

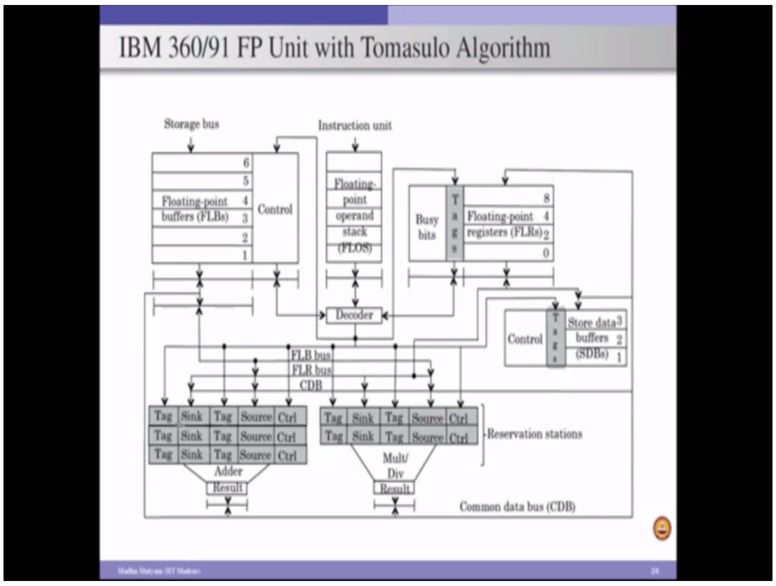
So, this will have the floating point instructions and we read the instructions from this floating point operand stack one at a time and by using this decoder we identify whether the floating point operation is a add operation or MUL or div and accordingly we steer the instruction to the corresponding functional unit. So, whenever, if the instruction requires the data from the memory, this decoder gives a signal to the FLB, so that, the FLB will supply the operand to the corresponding functional unit or if the operand needs to be read from the register that is FLR, then there will be a signal to that so that FLR will supply the data to the functional units.

And also here, in this particular design the floating point units are non-pipelined and they consider the 2 cycle delay for adder and 3 cycle delay for multiply and div unit. So, now once we have this design, we can easily see that because the floating point units are not pipelined, so, as a result, while one instruction is executed on the floating point unit that unit will not be available for next few cycles, if it is an adder unit it will not be available for next 2 cycles, if it is a multiply unit then it is not available for next 3 cycles.

So, as a result, because of this structural hazard, so subsequent floating point instructions cannot be issued to this the floating point units and also as and when any floating point unit completes the execution, then it will place the data on the result bus and through this result bus the data will be updated in the floating point register - FLR. So that any dependent instruction which wants the data then it can read the data from this floating point unit and issue that instruction to the floating point unit.

So, but, because these functional units are not pipelined and also each functional unit at any point of time can accommodate only one instruction, so as a result it is going to degrade the performance significantly. So, in order to improve the overall performance of this floating point unit, IBM had assigned a task to Tomasulo and he implemented a new design for this floating point unit and that actually became the basis for all the modern super scalar processors because it has a concept of reservation station, it has a concept of operant forwarding and it also has a concept of using common bus which will be connected to various register files so that operands can be forwarded to the consumer instructions and that improves overall performance. So, in order to improve the performance of this floating point unit or in order to improve the throughput of this floating point unit, so Tomasulo has suggested set of changes to this basic design and these changes are represented in this design.

(Refer Slide Time: 07:11)



IBM 360/91 FP Unit with Tomasulo Algorithm

The first change he made to this basic floating point unit is considering multiple buffers with each of the functional unit. Here he considers 3 buffers for adder unit and 2 buffers for multiply unit and each of these buffers actually working like a virtual functional unit. And he named these, the buffers, as reservation stations. Operands for instructions can be available either from the FLBs, that is floating point buffers or these operands may be supplied by the instructions which are sitting in the reservation stations.

So, we have 6 registers in the FLB, the floating point buffer register file and we have 5 entries in the reservation station. So, effectively we have total 11 entries from which we can get the values for the instructions whose operands are not ready. So, we can consider a 4 bit tag associated with each of these things. So, that by using this tag, we know whether we are going to get the data from the FLB or the data is available from the reservation stations.

And as and when the data is read from the memory to the FLB then we can forward that information through this common data bus connected to this register file. We will supply both the data as well as the tag associated with that particular entry. So, that this tag will be compared here and if there is a matching tag in the FLR, then we write the data onto the corresponding entry. And similarly, whenever any instruction which completes its execution, here, then we supply both the data as well as the tag of that particular entry onto this common bus.

So, that this common data bus is connected back to this reservation stations. So, we compare the tag of this computed data with the tags of all these entries and whenever there is a matching entry we write the corresponding data into the corresponding field in this reservation station. And also for example, if an instruction is writing some data to the memory, we cannot directly write to the memory because this is very costly if you are directly writing to the memory.

So, we first write to a register and from the register we can take the data and write it to the memory whenever memory is free or whenever we have no other operations to be performed. So, as a result whenever any value is produced and this value needs to be written to the memory through this common bus we actually send it to this another register file that is store data buffer and from there, so, we write the data into that particular thing.

So, as a result using this common data bus, we can ensure that operand forwarding can happen perfectly. And with the tags we can know the data is available either from the floating point buffers or from the instructions which are executed recently. And also once you have these tags, so whenever the data is available either from this FLB or from these functional units, we can forward those values to this register file - FLR as well as to the reservation stations.

And in this particular design, they have considered 2 operand instructions, where one operand will be a source operand, but the other operand will act as both the source as well as the destination. So, this operand which is specified as a sink operand, this acts as both the source and the destination, but whereas this second operand which is a source operand it will be acting only as a source, it will not be used as a destination. And we provided the necessary control signals for all these reservation stations to this FLB as well as to this SDB to ensure that the proper operations are performed.

Here, we can clearly see that there is a separate bus connecting the outputs of these functional units to this reservation station by using this CDB. And there is a bus connecting this FLB to the reservation stations by using this FLB bus and there is a separate bus connecting this FLR to the reservation station by using this FLR bus. So, once we have this design we can clearly see that we have set of reservation stations which act as virtual functional units. And we have the tags associated with the FLR and SDB. So, that operand forwarding can happen and we have a common data bus through which actual operand forwarding is taking place.

Also because of this FLB and SDB register files we are performing all our memory operations in register mode operations. So, that we are separating our memory access from the actual the computation. So, as and when the data is available in these registers then only the instruction will be issued to the functional unit and the functional unit is going to execute the instruction. In other words if the operands are not available for an instruction then the instruction will not be issued to the functional unit. It will be just waiting in the reservation station and it will wait until the operands both operands are ready. And we are now going to discuss an example to illustrate the working of this Tomasulo mechanism.

(Refer Slide Time: 13:29)

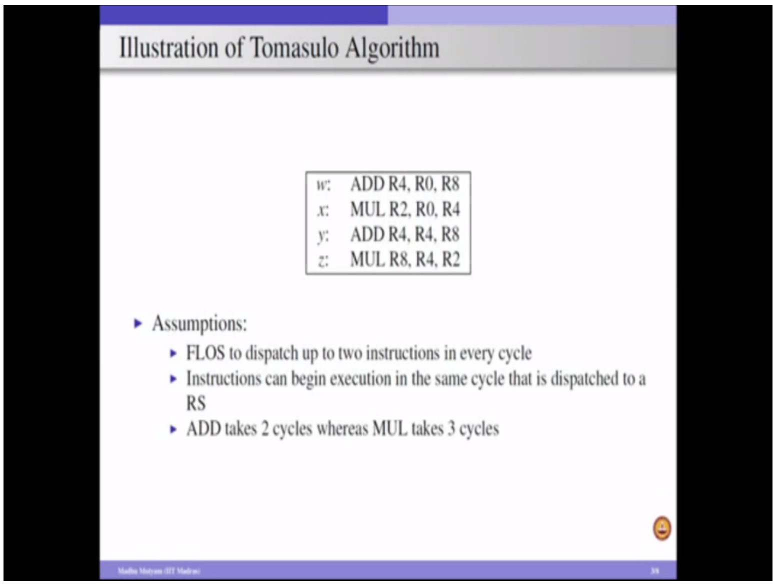


Illustration of Tomasulo Algorithm

w:	ADD R4, R0, R8
x:	MUL R2, R0, R4
y:	ADD R4, R4, R8
z:	MUL R8, R4, R2

- Assumptions:
  - FLOS to dispatch up to two instructions in every cycle
  - Instructions can begin execution in the same cycle that is dispatched to a RS
  - ADD takes 2 cycles whereas MUL takes 3 cycles

So, just consider an example consists of 4 instructions ADD R4 R0 R8, MUL R2 R0 R4, ADD R4 R4 R8 and MUL R8 R4 R2. So, though in the original design they have considered 2 operand instructions, but here we are considering 3 operand instructions. Here the first operand is the destination operand and the other 2 are the source operands. So, we have 4 instructions, here the second instruction that is a MUL instruction is having a true dependence with the add instruction because it is waiting for R4 to be available, where add is going to write to the R4 after add operation is performed.

And there is false independence between these 2 add instructions because they are writing to the same register R4. And there is a dependence between add and the multiply for this R8, but this is a false dependence, but there is a true dependence between add and multiply because of this R4. And also there is a true dependence between this multiply and the previous multiply because of R2.

And we also assume that, at any point of time we can dispatch 2 instructions from the FLOS the floating point operand stack to the functional units. And the instructions can begin execution in the same cycle in which the dispatched to a reservation station. And we also assume that add takes 2 cycles where as MUL takes 3 cycles in later stage. And we consider that even in the optimized design the add and multiply are actually not pipelined. So, now given this example we will see how the Tomasulo algorithm can be implemented or how the Tomasulo algorithm is working to compute this simple piece of code.

(Refer Slide Time: 15:36)

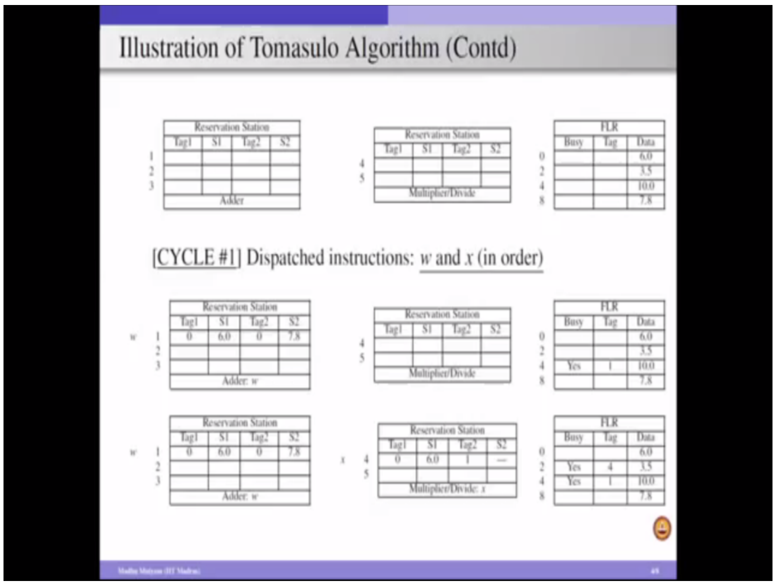


Illustration of Tomasulo Algorithm (Contd)

	Reservation Station			
	Tag1	S1	Tag2	S2
1				
2				
3				
	Adder			

	Reservation Station			
	Tag1	S1	Tag2	S2
4				
5				
	Multiplier/Divide			

	FLR		
	Busy	Tag	Data
0			6.0
2			3.5
4			10.0
8			7.8

[CYCLE #1] Dispatched instructions: *w* and *x* (in order)

		Reservation Station			
		Tag1	S1	Tag2	S2
w	1	0	6.0	0	7.8
	2				
	3				
		Adder: *w*			

	Reservation Station			
	Tag1	S1	Tag2	S2
4				
5				
	Multiplier/Divide			

	FLR		
	Busy	Tag	Data
0			6.0
2			3.5
4	Yes	1	10.0
8			7.8

		Reservation Station			
		Tag1	S1	Tag2	S2
w	1	0	6.0	0	7.8
	2				
	3				
		Adder: *w*			

		Reservation Station			
		Tag1	S1	Tag2	S2
x	4	0	6.0	1	—
	5				
		Multiplier/Divide: *x*			

	FLR		
	Busy	Tag	Data
0			6.0
2	Yes	4	3.5
4	Yes	1	10.0
8			7.8

So, we have 3 reservation stations associated with the adder, 2 reservation stations associated with the multiplier. And we have floating point register file FLR which has 4 registers. And also we assume that for the simplicity sake all our instructions are requiring the data from the register file only. So, there is no instruction which involves writing to the memory or reading from the memory. So, as a result we just consider only one register file that is FLR. And these registers are registers in the floating point register file are initialized with some values. So, register 0 has value 6, register 2 has value of 3.5, register 4 has a value of 10 and register 8 has a value 7.8.

So, now in the first cycle we are going to, we know that every cycle we are going to dispatch 2 instructions. So, in cycle one we are dispatching the first add and the first multiply instructions. So, we dispatch instructions w and x in the program order. So, w is issued, but we know that w instruction is nothing but ADD R4 R0 R8. So, it reads values from 2 registers R0 and R8, but R0 is having value 6 and R8 is having the value 7.8.

So, whenever the registers have the value then we can supply that value directly to the reservation station when we are dispatching this instruction into the reservation station. And to indicate whether the value stored in this S1 S2 are the actual value or the tag we are going to use this tag bits. If the tag bit indicates 0 that indicates that the value whatever is stored in the corresponding source operand will be the actual value. So, here because for the instruction w R0 R8 already having the value in the register file.

So, as a result we reset the tag bits here tag one and tag 2 are reset to 0. So, that this 6 and 7.8 actually indicates the correct values for this instruction. And now this instruction is now going to write to the register R4. So, at the end of the computation of this add instruction, our R4 is going to have the new value, the computed value that will be 13.8. So, in order to indicate that R4 is currently under updation. So, we have to go to this FLR floating point register file and we have to indicate that this value is stale value.

And that will be indicated by a tag 1. This 1 is nothing but the entry of the reservation station because we consider 5 reservation stations in total 3 for the add functional unit and 2 for the floating functional unit. So, we have 5 reservation station entries. So, new value for this R4 will be computed by an instruction that is stored in a reservation station entry. And that will be indicated. Here this one indicates that, this entry is going to produce a new value to this register R4. And after that we also set the busy bit to yes. Once the busy bit is yes or busy bit is 1 in the floating point register file that indicates that we should not read the value from the data field of that particular entry. And it also indicates that when the busy bit is 1. So, currently the instruction is under execution or the instruction is currently waiting in the reservation station. So, that the value in that corresponding entry is a stale value. And we know that we are dispatching 2 instructions in the same cycle, the cycle 1.

Now, we will look at what happens with the instruction x. We know that the instruction x is MUL R2 R0 R4. So, because this is a multiply instruction we are dispatching this instruction to the reservation stations associated with the multiply unit. So, here dispatching here. And here the multiply instruction requires 2 operands one is R0 and R0 is available readily from the floating point register file because when it goes to register file to read the value R0, the busy bit is not set tag bit is not set.

So, as a result we can directly read the value from this the corresponding data field of R0 register. That is a 6 and we are supply the 6 here, but the second operand for the multiply instruction is R4. So, to read the value stored in register R4, we go to the floating point register file, but the corresponding entry in the floating point register file indicates that, there is some other instruction which is actually going to compute a new value for this R4.

So, that is indicated by this busy bit. And also it says that the instruction that is stored in reservation station entry 1 will produce this value. So, as a result we supply this 1 to the tag of reservation station entry 4. So, once we have this 1 in the tag field. So, the value in S2 will be void because it indicates that second operand is not available at this point of time and it is going to be produced by an instruction which is stored in reservation station entry 1.

So, that is indicated here. And now because this multiply instruction is going to produce the new value to register R2. So, we will go to this floating point register file to the second entry and then we will set the busy bit. And also we set the tag field to 4 because this new value will be computed by an instruction that is stored in the reservation station entry 4. So, that is what is indicated here. So, remember here the tag field one indicates that the instruction that is stored in the reservation station entry 1 will produce this value new value.

Four indicates that the instruction that is stored in the reservation station 4 is going to produce a new value into register 2. So, with that the first instruction dispatch is completed in cycle 1, but we know that add is going to take 2 cycle latency, multiply is going to take 3 cycle latency. So, as a result add will not produce the value in the next cycle and we have to wait for the cycle 3 to get the value. And where as for the multiply instruction because multiply is dependent on the add instruction. So, that the operands for the multiply instruction will be available only in cycle 3 and after that it is going to take next 3 cycles and in cycle 6 only, it is going to compute its value.

(Refer Slide Time: 23:21)

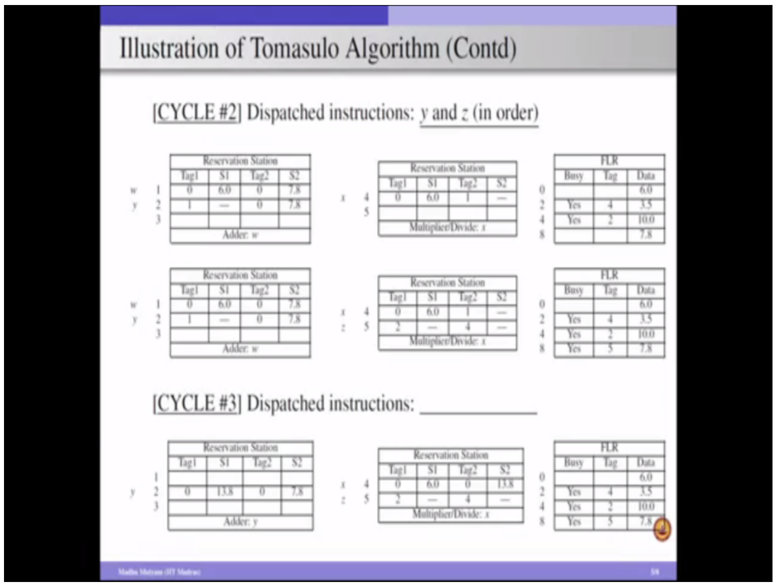


Illustration of Tomasulo Algorithm (Contd)

[CYCLE #2] Dispatched instructions: *y* and *z* (in order)

		Reservation Station			
		Tag1	S1	Tag2	S2
w	1	0	6.0	0	7.8
y	2	1	—	0	7.8
	3				
		Adder: w			

		Reservation Station			
		Tag1	S1	Tag2	S2
x	4	0	6.0	1	—
	5				
		Multiplier/Divide: x			

	FLR		
	Busy	Tag	Data
0			6.0
2	Yes	4	3.5
4	Yes	2	10.0
8			7.8

		Reservation Station			
		Tag1	S1	Tag2	S2
w	1	0	6.0	0	7.8
y	2	1	—	0	7.8
	3				
		Adder: w			

		Reservation Station			
		Tag1	S1	Tag2	S2
x	4	0	6.0	1	—
z	5	2	—	4	—
		Multiplier/Divide: x			

	FLR		
	Busy	Tag	Data
0			6.0
2	Yes	4	3.5
4	Yes	2	10.0
8	Yes	5	7.8

[CYCLE #3] Dispatched instructions: ____________

		Reservation Station			
		Tag1	S1	Tag2	S2
	1				
y	2	0	13.8	0	7.8
	3				
		Adder: y			

		Reservation Station			
		Tag1	S1	Tag2	S2
x	4	0	6.0	0	13.8
z	5	2	—	4	—
		Multiplier/Divide: x			

	FLR		
	Busy	Tag	Data
0			6.0
2	Yes	4	3.5
4	Yes	2	10.0
8	Yes	5	7.8

Madhu Mutyam (IIT Madras)

So, in the next cycle, because every cycle we can dispatch new set of instructions, because we have multiple reservation stations associated with each of the functional units. As long as there is a free entry in the reservation station we can dispatch new instructions. So, now we are going to dispatch 2 more instructions those are y and z. And y instruction is, it is an ADD R4 R4 and R8. So, we are going to read contents from register R4 and the register R8 and we compute the, we perform the add operation and we will write the computed value into register R4.

The z instruction is MUL R8 R4 R2. We are going to read the values from R4 and R2 registers and write the computed value to register R8. So, because the y instruction is an add instruction. So, effectively we are going to dispatch this instruction into reservation station associated with the adder and where as the z instruction is multiply instruction we are going to dispatch this instruction into the reservation station associated with the multiply functional unit. And we know that instruction y requires 2 operands, one is from R4 other one is from R8.

So, now we will go to the register file to see whether R4 and R8 are available. Of course, R8 is available. So, we supply this value directly. So, 7.8 is stored here and the tag is reset that indicates that this is the value required by that instruction, but R4 is the other operand for this add instruction, but we can clearly see here R4 is supplied by the instruction that is stored in the reservation station entry number 1.

So, we supply 1 to this particular tag field. So, this 1 indicates that this instruction is going to produce the value and that value will be required here in this S1 field of second entry of this reservation station. And second operand for the instruction y is already available that is 7.8 we already supplied from the floating point register file FLR, but now this instruction y is actually writing the new value to the register R4.

So, now we update the tag field for this entry 4 from 1 to 2. Previously the value stored in the tag field of register 4 is 1, but now it will be 2. The reason is previously instruction that was stored in reservation station entry 1 will be updating R4, but now the second instruction that is instruction y is going to update this R4 value. We know that there is an output dependency between instruction w and instruction y. So, as a result both the instructions are writing to the same register that is R4. So, as and when we dispatch instruction y to the reservation station.

So, we have to update the corresponding reservation station entry in the tag field of R4 register in the register file. So, that is what we have done here. So, from now on R4 will have 2 in the tag field and coming to the second instruction that is dispatch in the cycle 2. So, this is z instruction which requires 2 operands that is R4 and R2 and writes the value to register

R8. So, but R4 is not available according to this register file. It is going to be supplied by instruction that is stored in reservation station entry 2. So, we are going to supply this 2 to the tag field of the source operand of instruction z.

Similarly, the other operand for instruction z is R2, but R2 is also not available as indicated by this register file and it is going to be produced by an instruction that is stored in the reservation station entry 4. We are going to supply 4 to the tag filed of second operand of this instruction. By the way because this instruction z is writing the value to register R8. So, we update the R8 entry in the register file, by setting the busy bit and also by setting the tag field. Tag field now is having the value 5 for R8 entry in the register file and this 5 indicates that the instruction that is stored in the fifth entry in the register file is going to supply the data required for this R8.

So, that is what is indicated here. Effectively at the end of second cycle the state of FLR is like this. So, R0 is not going to be updated by any of the instructions, but whereas R2 is going to be updated by the instruction that is stored in reservation station 4 and R4 register is going to be updated by the instruction that is stored in the reservation station entry 2. And R8 is going to be updated by the instruction that is stored in the reservation station entry 5.

Now, in the third cycle because there are no more instructions to dispatch. So, we are not dispatching any more instructions, but our first add instruction is issued in cycle 1 and adder is taking 2 cycle latencies. So, as a result in the third cycle add operation is completed. And because the add operation is computing the add on 2 operands. One is 6.0 and the other one is 7.8. So, we computed the value as 13.8 and this13.8 needs to be forwarded to all the pending instructions which are requiring this value. As well as we have to send it to the register file if the register file is waiting for this value to be updated in one of the registers.

So, here as an when this add instruction is completed. So, we are forwarding this value through the common data bus, while we are forwarding this value on the common data bus we also forward the tag of this reservation station entry. So, we are forwarding both 13.8 as well as the tag 1. So, when we forward this value through the common data bus and since that common data bus is connected to all the reservation station entries. And we check all the entries in the reservation station to see if there is any instruction which is waiting for this value to be available.

We can clearly see here in this 2 reservation stations. So, here the second instruction in the add reservation station requires the value from this add instruction. Similarly, the first entry in the reservation station associated with the multiply functional unit also requires the value from this add instruction. So, we are forwarding the value to these 2 entries. So, as a result we can clearly see here, previously this entry was having 1 in the tag field and there is no value in the S1. Now, it becomes 0 in the tag field and 13.8 in the source field. Similarly, here the tag field has one and the source field is having stale data or do not care data.

Now, the tag is going to have a 0 and the source field of this instruction is having 13.8. So, remember whenever the tag field has 0 that indicates the data associated with that particular tag field is the actual data. So, that data forwarded through the operand forwarding by using this common data bus. And to enable this forwarding we use this tag fields and by checking the proper tags in the reservation stations, we have written this computed value to the appropriate locations.

Since there is no tag field that is matching with the tag of this 1 as a result we have not updated anything in the register file. Now, we can clearly see here this second add instruction that is dispatched to this reservation station associated with the add functional unit has 2 operands ready 13.8 and 7.8. So, as a result now this instruction is ready to be executed and also we know that adder is currently idle. So, there is no instruction is executed on this adder. So, as a result we can dispatch this instruction directly onto the functional unit adder. And similarly, in the case of multiply reservation stations, the first instruction that is dispatched in the reservation station has both operands ready 6 and 13.8. So, as a result this multiply operation also can be performed in this cycle.

(Refer Slide Time: 33:17)

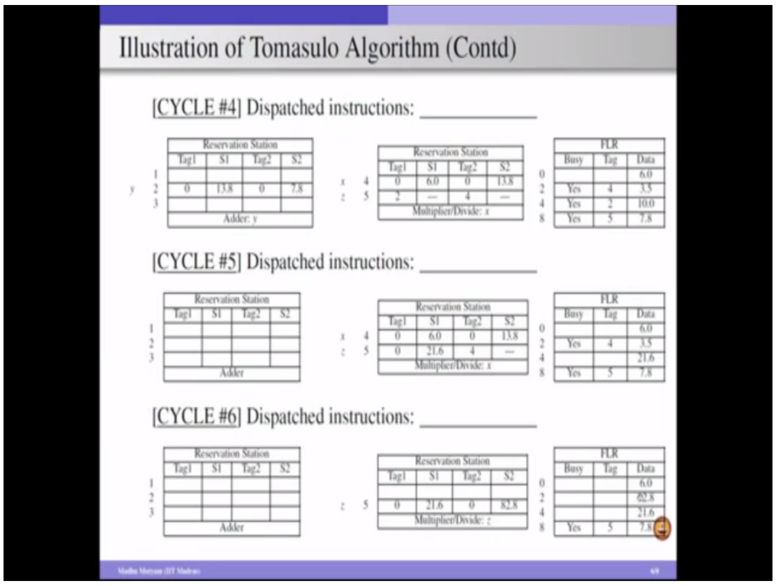


Illustration of Tomasulo Algorithm (Contd)

[CYCLE #4] Dispatched instructions: ____________

		Reservation Station			
		Tag1	S1	Tag2	S2
	1				
y	2	0	13.8	0	7.8
	3				
		Adder: y			

		Reservation Station			
		Tag1	S1	Tag2	S2
x	4	0	6.0	0	13.8
z	5	2	—	4	—
		Multiplier/Divide: x			

	FLR		
	Busy	Tag	Data
0			6.0
2	Yes	4	3.5
4	Yes	2	10.0
8	Yes	5	7.8

[CYCLE #5] Dispatched instructions: ____________

	Reservation Station			
	Tag1	S1	Tag2	S2
1				
2				
3				
	Adder			

		Reservation Station			
		Tag1	S1	Tag2	S2
x	4	0	6.0	0	13.8
z	5	0	21.6	4	—
		Multiplier/Divide: x			

	FLR		
	Busy	Tag	Data
0			6.0
2	Yes	4	3.5
4			21.6
8	Yes	5	7.8

[CYCLE #6] Dispatched instructions: ____________

	Reservation Station			
	Tag1	S1	Tag2	S2
1				
2				
3				
	Adder			

		Reservation Station			
		Tag1	S1	Tag2	S2
z	5	0	21.6	0	82.8
		Multiplier/Divide: z			

	FLR		
	Busy	Tag	Data
0			6.0
2			82.8
4			21.6
8	Yes	5	7.8

Madhu Mutyam (IIT Madras) 68

Because the add instruction is started executing in cycle 3. So, it is going to take 2 cycles to complete. So, as a result in the 4th cycle we cannot get any output. So, because of that reason the contents of reservation stations as well as the register file are same in both cycle 3 and cycle 4. Now, in cycle 5 because this add instruction is completed in the cycle 5. And we have value 21.6 and we have to forward this value to all the pending instructions as well as to the register file.

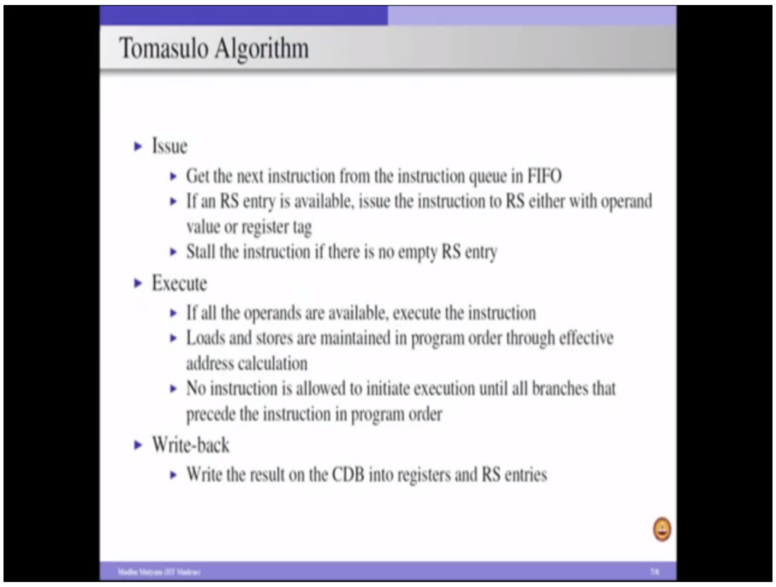
Now, we can clearly see here the first operand of this instruction z is actually waiting for this value to be available. As well as register 4 in the register file is actually waiting for this value. So, we forward this value that is 21.6 to these 2 entries and that will be updated. So, we can clearly see here this 21.6 is updated for this field in the fifth entry of the register file. And also it will be updated in the register file for register 4. So, 21.6 is updated here.

Apart from that there is no other operations performed here. And now in cycle 6 we have to see what happens because we know that multiply instruction is started executing in cycle 3 and it takes 3 cycles. So, at the end in cycle 6 this multiply operation is completed and so, this 6 and 13.8. So, this is computed and the value is 82.8 and this value needs to be forwarded to the pending instructions as well as to the register file. So, we know that the second operand of instruction z is actually waiting for this multiply operation to be completed.

So, we forward that value 82.8 to this. As well as the second register in our register file is actually waiting for this value to be available. So, we update in the register file also for R2.

So, 2 operands for this last multiply operation are available. So, as a result we start executing this multiply operation using this multiplier functional unit in cycle 6. And it is going to take 3 cycles to complete. As a result in cycle 9 we will get the value computed by this multiply operation and accordingly we will update register R8 in the register file. So, this is how instructions are going to be executed by using this Tomasulo algorithm. And from this Tomasulo algorithm we can clearly see that.

(Refer Slide Time: 36:12)



Tomasulo Algorithm

- Issue
  - Get the next instruction from the instruction queue in FIFO
  - If an RS entry is available, issue the instruction to RS either with operand value or register tag
  - Stall the instruction if there is no empty RS entry
- Execute
  - If all the operands are available, execute the instruction
  - Loads and stores are maintained in program order through effective address calculation
  - No instruction is allowed to initiate execution until all branches that precede the instruction in program order
- Write-back
  - Write the result on the CDB into registers and RS entries

We fetch the instructions in the program order from FLOS, that is floating point operand stack. So, when there is an free entry in the reservation station, we issue the instruction to the reservation station either using the operands or by sending the register tags. And if there is no free entry in the reservation station then we are not going to dispatch the instruction to the reservation station. So, that will take care of the structural hazard. So, in the execute phase if all the operands of an instruction are ready then only we issue this instruction to the functional unit. So, that the functional unit is going to execute this instruction.

This takes care of our read after write hazard and for all the loads and stores we are going to maintain the program order because for this loads and stores we are actually computing the effective address calculation, but all these instructions are following the in order. So, effectively loads and stores are maintain the program order because of this effective orders calculation we perform. And in this particular algorithm so they have considered that no instruction is allowed to initiate the execution until all the previous branches are resolved.

So, as a result there is no speculative execution involved in this particular design, but if we want to add the speculation. So, we have to take care of the small modifications to this optimized design by considering speculations and other things. And finally, once the instruction is executed then we write the result on to this common data bus. And through that these values will be forwarded to the register file as well as to the reservation station entries. So, with that I am concluding this module and in the next module we are going to see the dynamic execution core and remaining components in the Superscalar processor design.

Thank you.