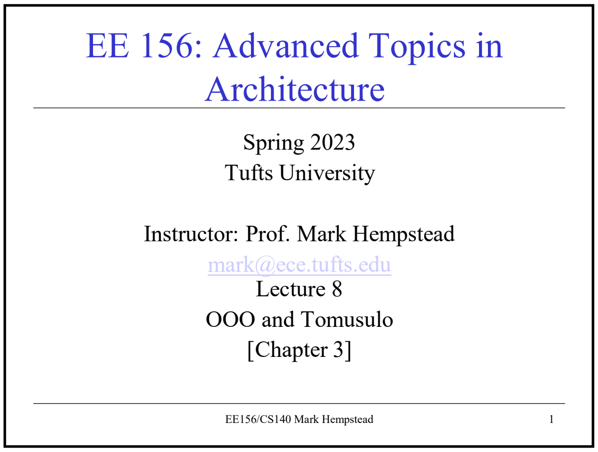
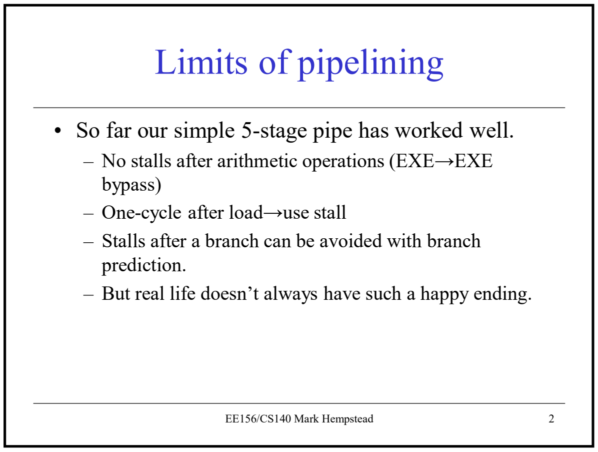


EE 156: Advanced Topics in Architecture

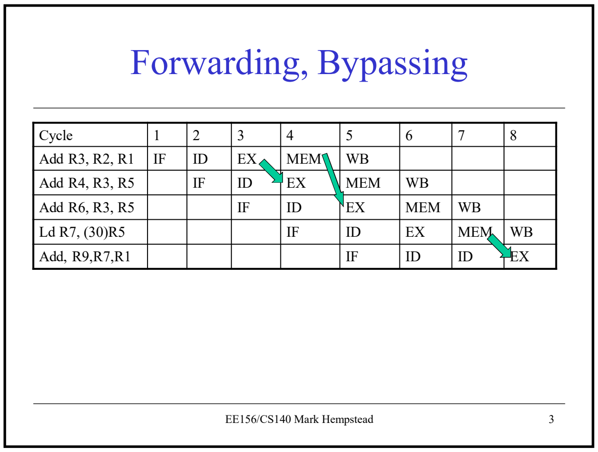
Spring 2023
Tufts University

Instructor: Prof. Mark Hempstead
mark@ece.tufts.edu
Lecture 8
OOO and Tomusulo
[Chapter 3]



Limits of pipelining

- So far our simple 5-stage pipe has worked well.
  - No stalls after arithmetic operations (EXE→EXE bypass)
  - One-cycle after load→use stall
  - Stalls after a branch can be avoided with branch prediction.
  - But real life doesn't always have such a happy ending.



Forwarding, Bypassing

Cycle	1	2	3	4	5	6	7	8
Add R3, R2, R1	IF	ID	EX	MEM	WB			
Add R4, R3, R5		IF	ID	EX	MEM	WB		
Add R6, R3, R5			IF	ID	EX	MEM	WB	
Ld R7, (30)R5				IF	ID	EX	MEM	WB
Add, R9,R7,R1					IF	ID	ID	EX

Long pipes = lots of stalls

- Our 5-stage pipe may be a 15-stage pipe. Why?
  - Smaller stages with less logic → higher frequency
  - Dependencies now imply more stalls.
  - Loads & branches similarly get worse.

Cycle	1	2	3	4	5	6	7	8					
Add R3, R2, R1	F1	F2	D1	D2	E1	E2	E3	M1	M2	W1	W2		
Add R4, R3, R5		F1	F2	D1	D2	D2	D2	E1	E2	E3	M1	M2	W1	W2

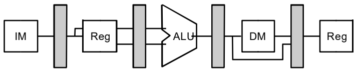
Load misses = even worse

- Even with a 5-stage pipe, load misses can cause numerous stalls:

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
Ld R3, 10(R1)	IF	ID	EX	M	M	M	M	M	M	WB			
Add R4, R3, R5		IF	ID	ID	ID	ID	ID	ID	ID	EX	M	WB	
Sub R7, R8, R9			IF	IF	IF	IF	IF	IF	IF	ID	EX	M	WB

- What's really dumb about this stall?
  - The final Sub is stalled, but it doesn't use R3 or R4, and the ALU is sitting idle.

Not so easy to fix this



Sub R7, R8,R9

Add R4, R3,R5

Ld R3, 10(R1)

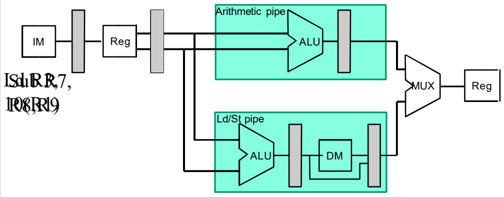
- Problems:
  - The ID and Mem pipe stages are full. The Sub cannot just hop over them (not without lots of new wires)
  - Remember when we talked about exceptions? What if the Sub finishes before the Ld and Add, but then the Ld or Add takes an exception?

Baby steps to fix it

- Let’s take one step at a time, and ignore the other daunting problems ahead of us ☺.
- So first let’s try to get our independent instructions into the pipeline without bouncing. But how?

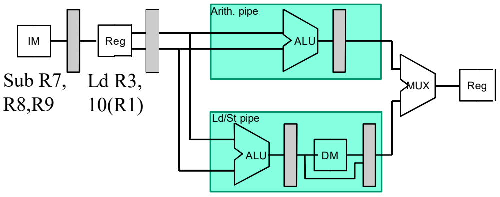
Multiple small pipes

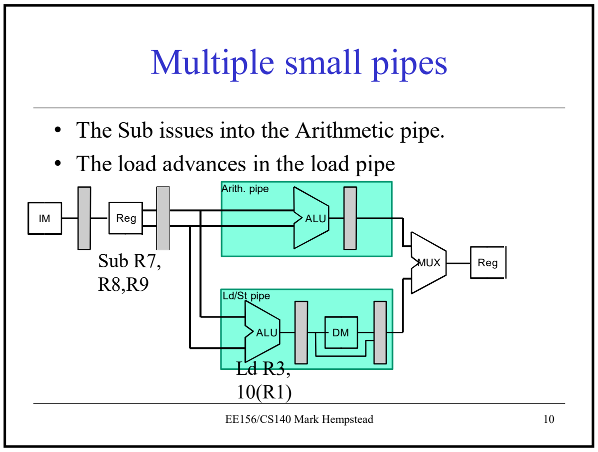
- How about one pipe for Ld/St and one for arithmetic (add, sub, and, or, etc.)?



Multiple small pipes

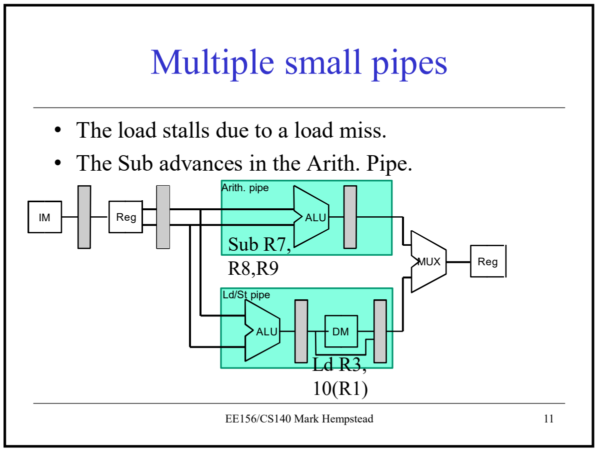
- The sub enters ID
- The load issues into the load pipe





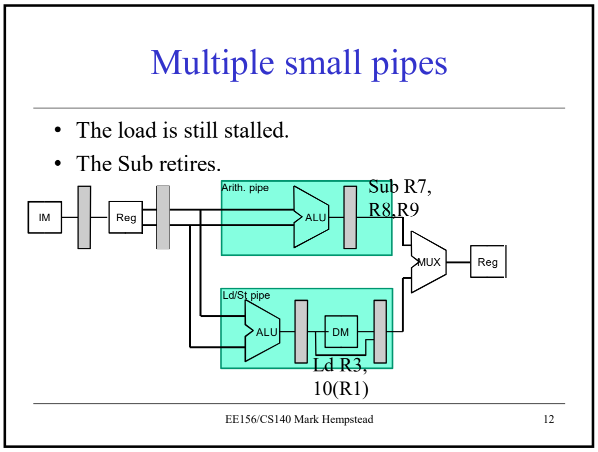
Multiple small pipes

- The Sub issues into the Arithmetic pipe.
- The load advances in the load pipe



Multiple small pipes

- The load stalls due to a load miss.
- The Sub advances in the Arith. Pipe.



Multiple small pipes

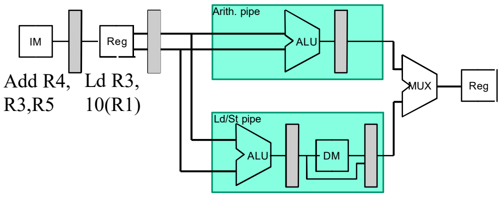
- The load is still stalled.
- The Sub retires.

Unintended consequences

- We've had an unrelated instruction that would have been stuck behind a stalled load.
  - We let it advance past the load into its own mini-pipe
  - But it retired OOO. Won't that cause problems with exceptions?   Yes!
  - And it still doesn't solve all of our problems ☹
  - Consider this sequence:
    Ld R3, 10(R1)
    Add R4, R3,R5
    Sub R7, R8,R9

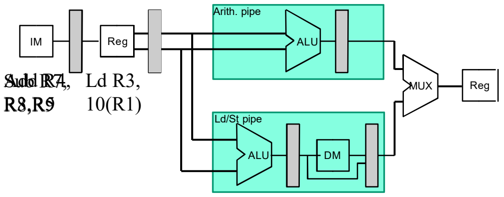
Multiple small pipes

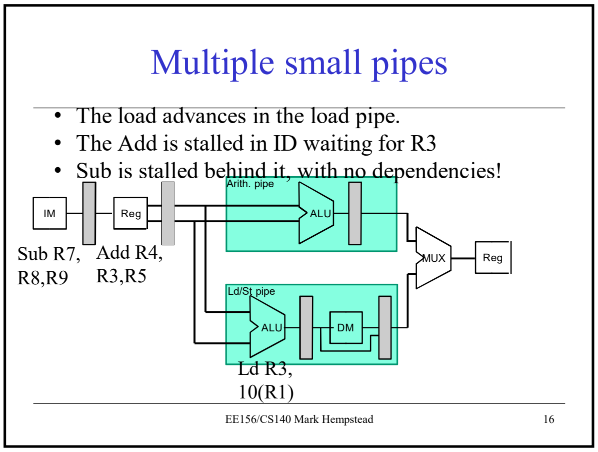
- Start with the Add and Ld in the pipe.



Multiple small pipes

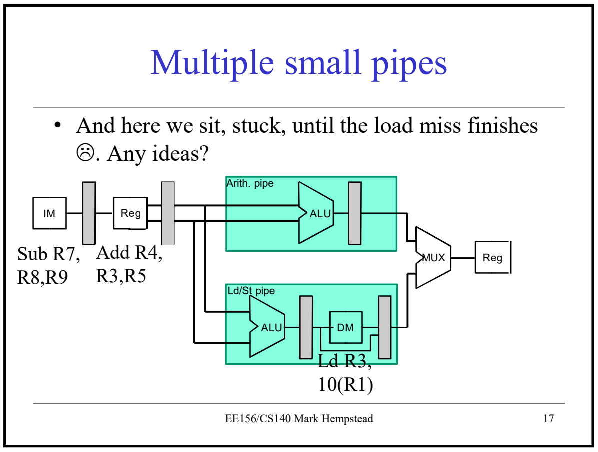
- The load issues into the load pipe, the Add advances and the Sub is fetched.





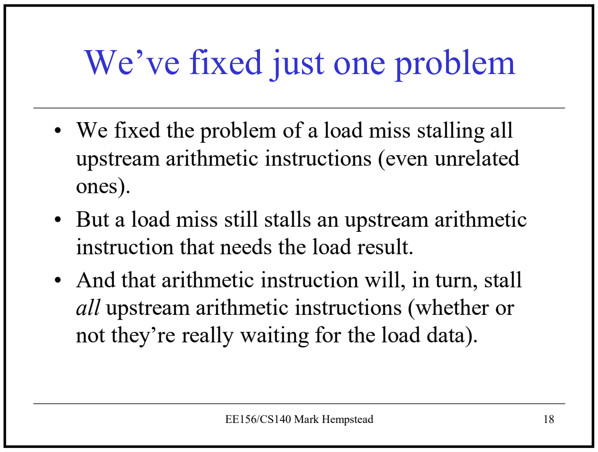
Multiple small pipes

- The load advances in the load pipe.
- The Add is stalled in ID waiting for R3
- Sub is stalled behind it, with no dependencies!



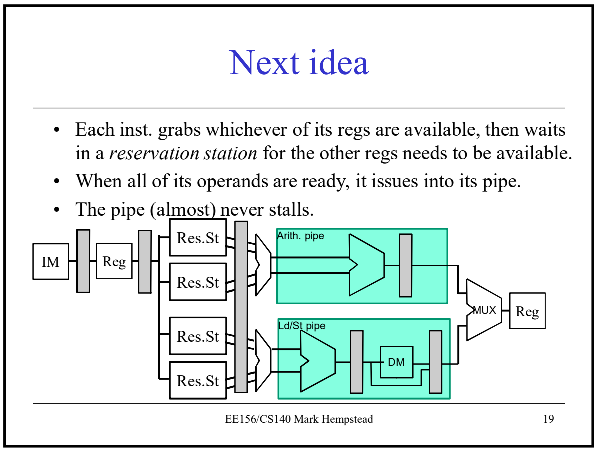
Multiple small pipes

- And here we sit, stuck, until the load miss finishes ☹. Any ideas?



We’ve fixed just one problem

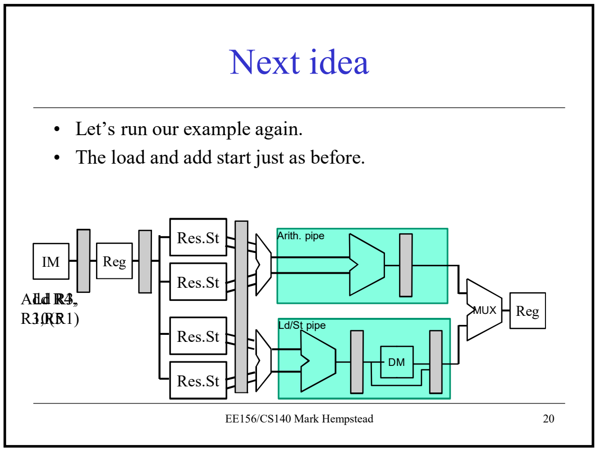
- We fixed the problem of a load miss stalling all upstream arithmetic instructions (even unrelated ones).
- But a load miss still stalls an upstream arithmetic instruction that needs the load result.
- And that arithmetic instruction will, in turn, stall *all* upstream arithmetic instructions (whether or not they’re really waiting for the load data).



Next idea

- Each inst. grabs whichever of its regs are available, then waits in a *reservation station* for the other regs needs to be available.
- When all of its operands are ready, it issues into its pipe.
- The pipe (almost) never stalls.

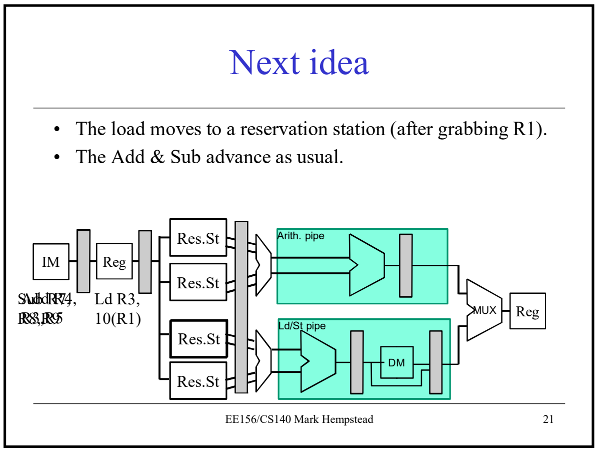
EE156/CS140 Mark Hempstead 19



Next idea

- Let’s run our example again.
- The load and add start just as before.

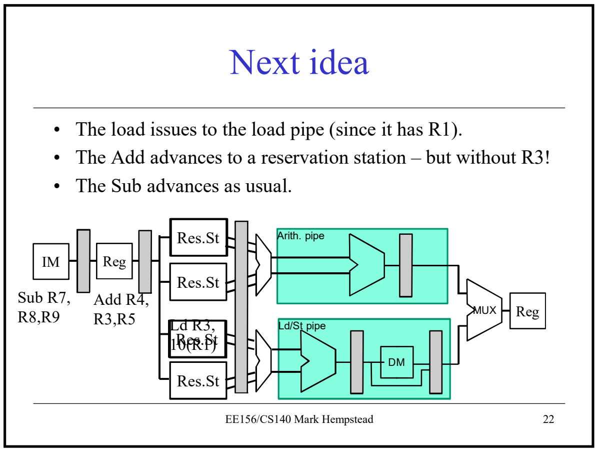
EE156/CS140 Mark Hempstead 20



Next idea

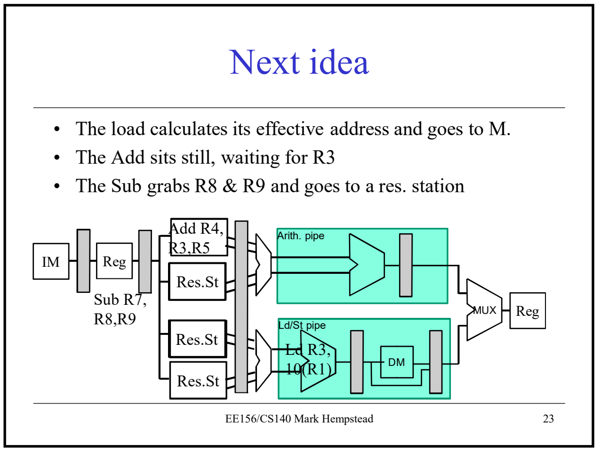
- The load moves to a reservation station (after grabbing R1).
- The Add & Sub advance as usual.

EE156/CS140 Mark Hempstead 21



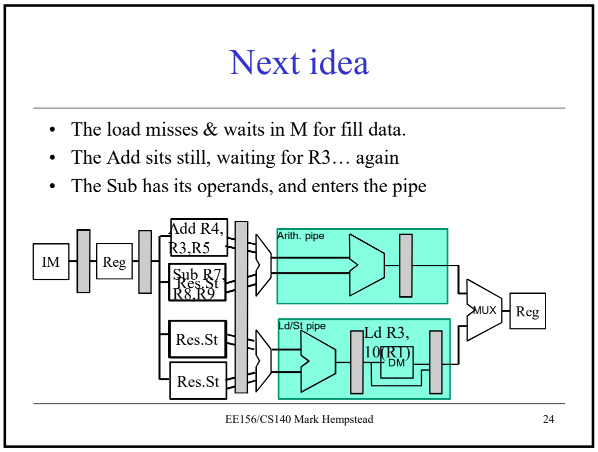
Next idea

- The load issues to the load pipe (since it has R1).
- The Add advances to a reservation station – but without R3!
- The Sub advances as usual.



Next idea

- The load calculates its effective address and goes to M.
- The Add sits still, waiting for R3
- The Sub grabs R8 & R9 and goes to a res. station

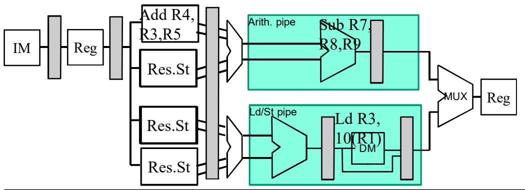


Next idea

- The load misses & waits in M for fill data.
- The Add sits still, waiting for R3… again
- The Sub has its operands, and enters the pipe

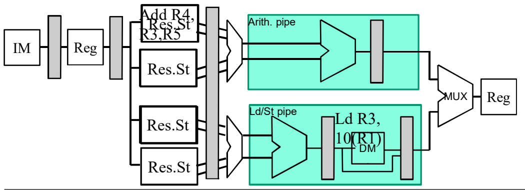
Next idea

- The load is still waiting in M for fill data.
- The Add is still waiting for R3… again
- The Sub retires to the register file and is done.
- And here we wait… until the load returns data.



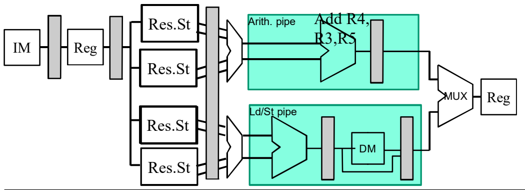
Next idea

- Now assume the load has finally gotten its data (e.g., from L2).
- The load advances to the RF and is done.
- The Add grabs the R3 bypass data & enters the pipe.



Next idea

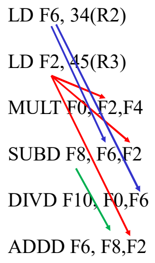
- The Add finishes, goes to the RF and is done.



That’s (mostly) Tomusulo

- We’ve (more or less) built Tomusulo
  - It’s a *dataflow* machine that can execute out of order.
  - Instructions mostly enter the appropriate pipe as soon as their operands are ready
  - “Bypassing” is officially called the Common Data Bus, and loads/store are a bit different than we drew.
- We’ve made precise exceptions even harder: why?
  - The SUB (in principle the 3rd instruction) finished before the ADD; but what if the ADD had an exception?
- And we’ve added two new problems: WAW and WAR hazards.

What is data flow


LD F6, 34(R2)

LD F2, 45(R3)

MULT F0, F2,F4

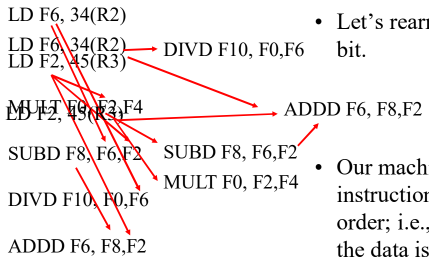
SUBD F8, F6,F2

DIVD F10, F0,F6

ADDD F6, F8,F2

- Here’s a short program.
- Here are the dependencies

Data flow


LD F6, 34(R2)

LD F6, 34(R2)

LD F2, 45(R3)

MULT F0, F2,F4

LD F2, 45(R3)

SUBD F8, F6,F2

DIVD F10, F0,F6

ADDD F6, F8,F2

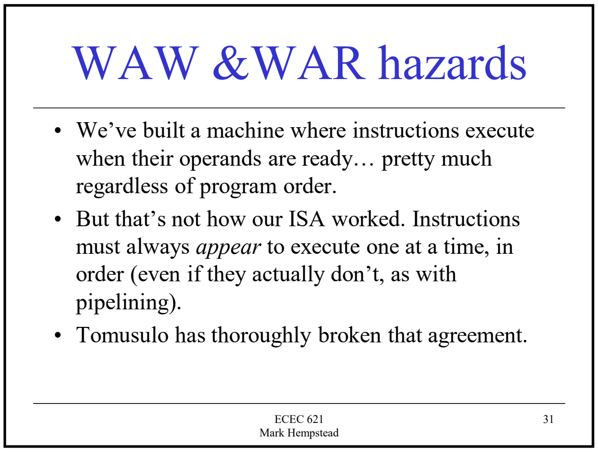
DIVD F10, F0,F6

ADDD F6, F8,F2

SUBD F8, F6,F2

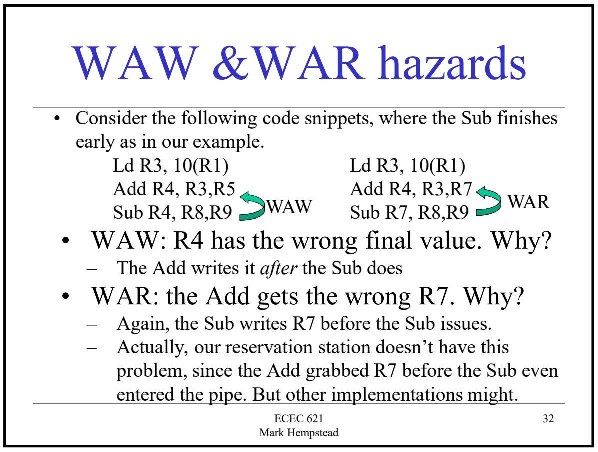
MULT F0, F2,F4

- Let’s rearrange them a bit.
- Our machine executes instructions in this order; i.e., as soon as the data is ready.



WAW &WAR hazards

- We’ve built a machine where instructions execute when their operands are ready… pretty much regardless of program order.
- But that’s not how our ISA worked. Instructions must always *appear* to execute one at a time, in order (even if they actually don’t, as with pipelining).
- Tomusulo has thoroughly broken that agreement.

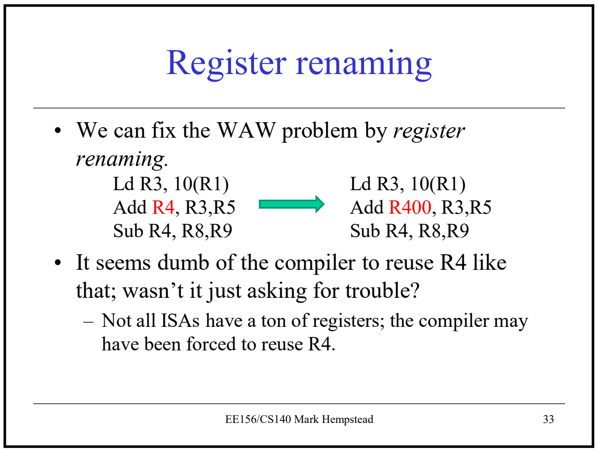


WAW &WAR hazards

- Consider the following code snippets, where the Sub finishes early as in our example.

```
Ld R3, 10(R1)                Ld R3, 10(R1)
Add R4, R3,R5                Add R4, R3,R7
Sub R4, R8,R9   WAW          Sub R7, R8,R9   WAR
```

- WAW: R4 has the wrong final value. Why?
  - The Add writes it *after* the Sub does
- WAR: the Add gets the wrong R7. Why?
  - Again, the Sub writes R7 before the Sub issues.
  - Actually, our reservation station doesn’t have this problem, since the Add grabbed R7 before the Sub even entered the pipe. But other implementations might.

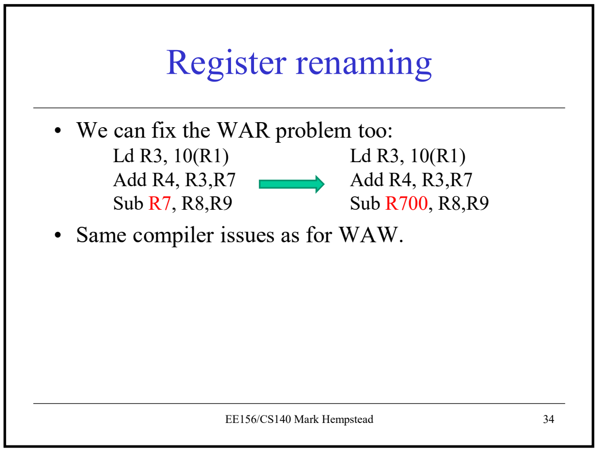


Register renaming

- We can fix the WAW problem by *register renaming.*

```
Ld R3, 10(R1)           Ld R3, 10(R1)
Add R4, R3,R5    →      Add R400, R3,R5
Sub R4, R8,R9           Sub R4, R8,R9
```

- It seems dumb of the compiler to reuse R4 like that; wasn’t it just asking for trouble?
  - Not all ISAs have a ton of registers; the compiler may have been forced to reuse R4.

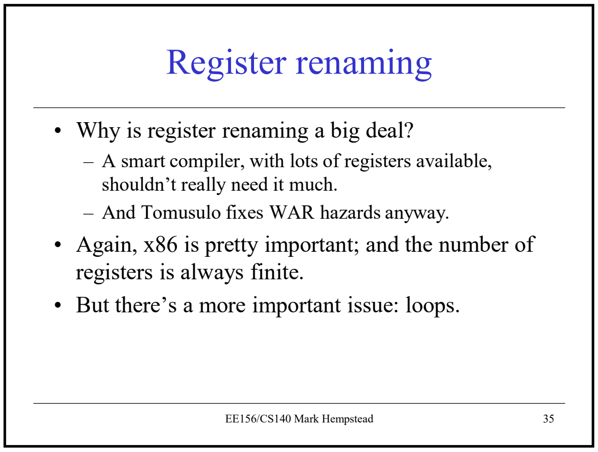


Register renaming

- We can fix the WAR problem too:

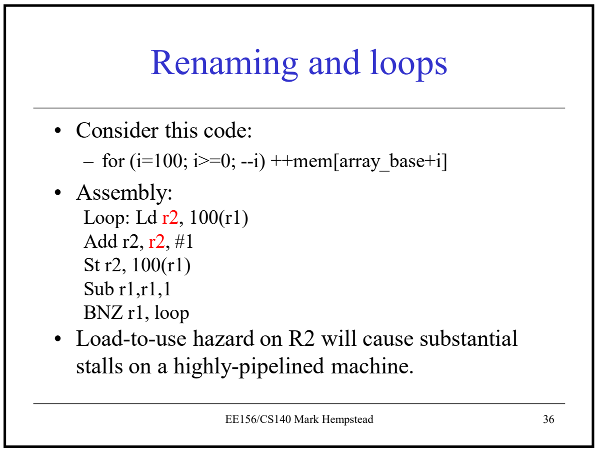
```
Ld R3, 10(R1)                 Ld R3, 10(R1)
Add R4, R3,R7        →        Add R4, R3,R7
Sub R7, R8,R9                 Sub R700, R8,R9
```

- Same compiler issues as for WAW.



Register renaming

- Why is register renaming a big deal?
  - A smart compiler, with lots of registers available, shouldn't really need it much.
  - And Tomusulo fixes WAR hazards anyway.
- Again, x86 is pretty important; and the number of registers is always finite.
- But there's a more important issue: loops.

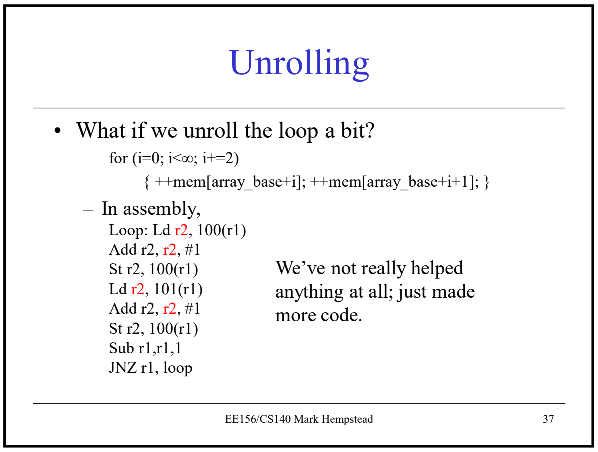


Renaming and loops

- Consider this code:
  - for (i=100; i>=0; --i) ++mem[array_base+i]
- Assembly:

```
Loop: Ld r2, 100(r1)
Add r2, r2, #1
St r2, 100(r1)
Sub r1,r1,1
BNZ r1, loop
```

- Load-to-use hazard on R2 will cause substantial stalls on a highly-pipelined machine.



Unrolling

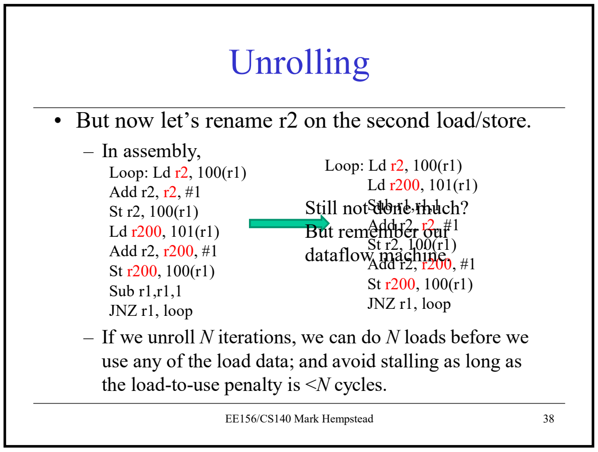
- What if we unroll the loop a bit?

  ```
  for (i=0; i<∞; i+=2)
        { ++mem[array_base+i]; ++mem[array_base+i+1]; }
  ```

  – In assembly,

  ```
  Loop: Ld r2, 100(r1)
  Add r2, r2, #1
  St r2, 100(r1)
  Ld r2, 101(r1)
  Add r2, r2, #1
  St r2, 100(r1)
  Sub r1,r1,1
  JNZ r1, loop
  ```

We've not really helped anything at all; just made more code.



Unrolling

- But now let's rename r2 on the second load/store.

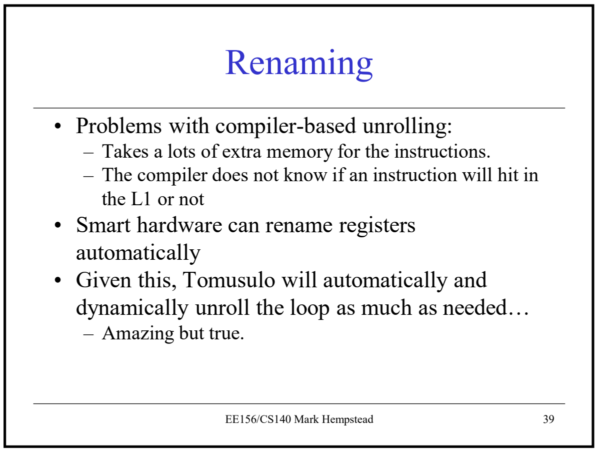
  – In assembly,

  ```
  Loop: Ld r2, 100(r1)
  Add r2, r2, #1
  St r2, 100(r1)
  Ld r200, 101(r1)
  Add r2, r200, #1
  St r200, 100(r1)
  Sub r1,r1,1
  JNZ r1, loop
  ```

  ```
  Loop: Ld r2, 100(r1)
        Ld r200, 101(r1)
        Sub r1,r1,1
        Add r2, r2, #1
        St r2, 100(r1)
        Add r2, r200, #1
        St r200, 100(r1)
        JNZ r1, loop
  ```

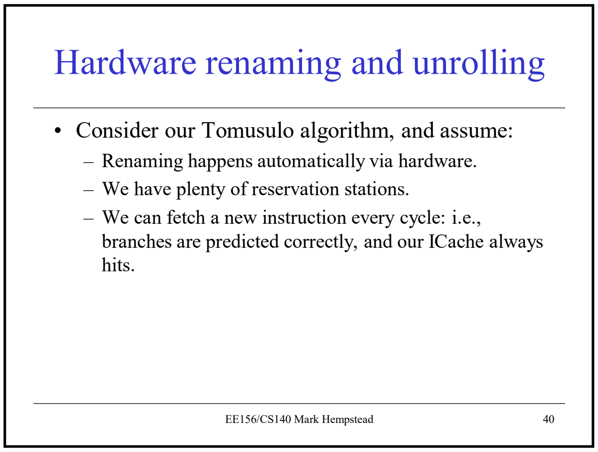
Still not done much? But remember our dataflow machine.

  – If we unroll *N* iterations, we can do *N* loads before we use any of the load data; and avoid stalling as long as the load-to-use penalty is <*N* cycles.



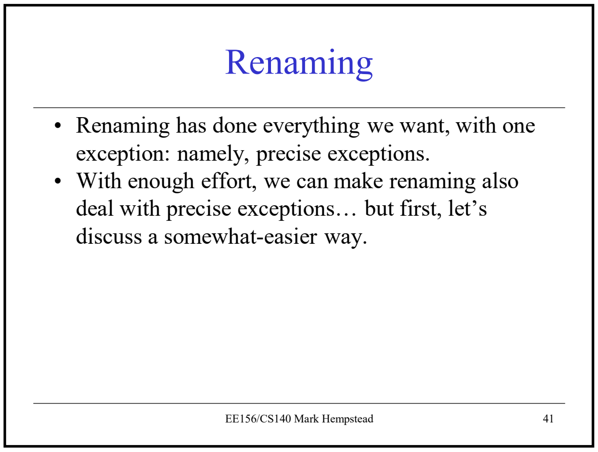
Renaming

- Problems with compiler-based unrolling:
  – Takes a lots of extra memory for the instructions.
  – The compiler does not know if an instruction will hit in the L1 or not
- Smart hardware can rename registers automatically
- Given this, Tomusulo will automatically and dynamically unroll the loop as much as needed…
  – Amazing but true.



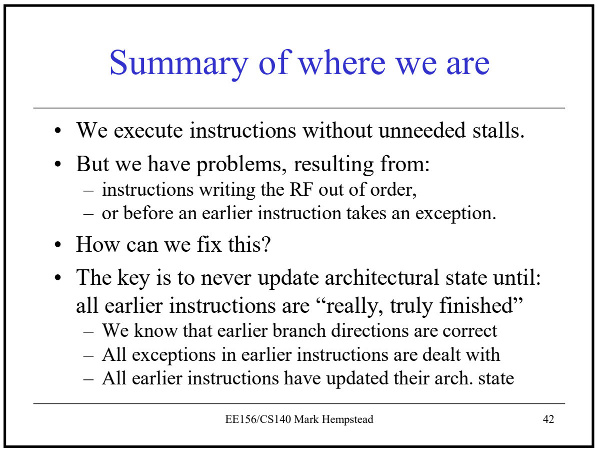
Hardware renaming and unrolling

- Consider our Tomusulo algorithm, and assume:
  - Renaming happens automatically via hardware.
  - We have plenty of reservation stations.
  - We can fetch a new instruction every cycle: i.e., branches are predicted correctly, and our ICache always hits.



Renaming

- Renaming has done everything we want, with one exception: namely, precise exceptions.
- With enough effort, we can make renaming also deal with precise exceptions… but first, let's discuss a somewhat-easier way.



Summary of where we are

- We execute instructions without unneeded stalls.
- But we have problems, resulting from:
  - instructions writing the RF out of order,
  - or before an earlier instruction takes an exception.
- How can we fix this?
- The key is to never update architectural state until: all earlier instructions are "really, truly finished"
  - We know that earlier branch directions are correct
  - All exceptions in earlier instructions are dealt with
  - All earlier instructions have updated their arch. state

Reorder buffer

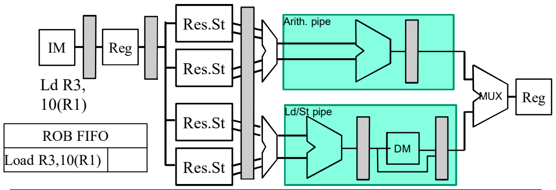
- Definition: an instruction *commits* when we update architectural state with its results.
  - The key is to execute out of order, but commit in order.
- So we must somehow:
  - Even as we put instructions into multiple pipes in dataflow order, still remember which instruction came first in the program.
  - After an instruction finishes, save its results until it commits & we can thus write the RF.
- A *reorder buffer* (ROB) is the magic trick to do both of these
  - It's really not magic, or even very hard.

ROB details

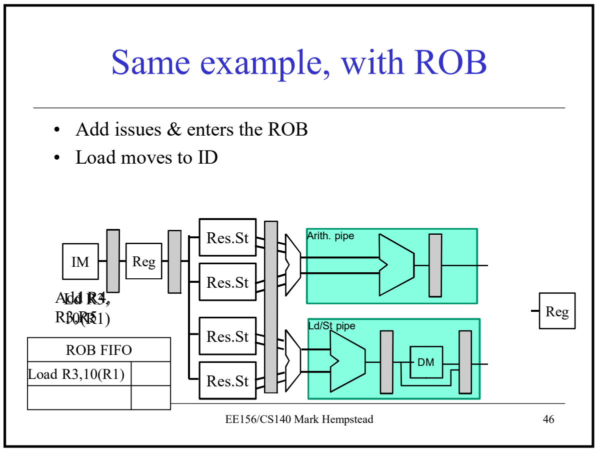
- The ROB is a FIFO.
- An instruction will:
  - Enter the ROB when it gets fetched
  - Store its result in its ROB entry when the result is calculated (i.e., when it exits the appropriate pipe)
  - Leave the ROB when it commits. Then move its results to the RF.
- When can it commit?
  - Its results must be in the ROB with it
  - All earlier instructions must be committed (i.e., it must be at the head of the FIFO).
- The FIFO is usually a circular buffer with head/tail pointers.
- OK, let's do our simple example yet again!

Same example, with ROB

- Load issues. It also enters the ROB.



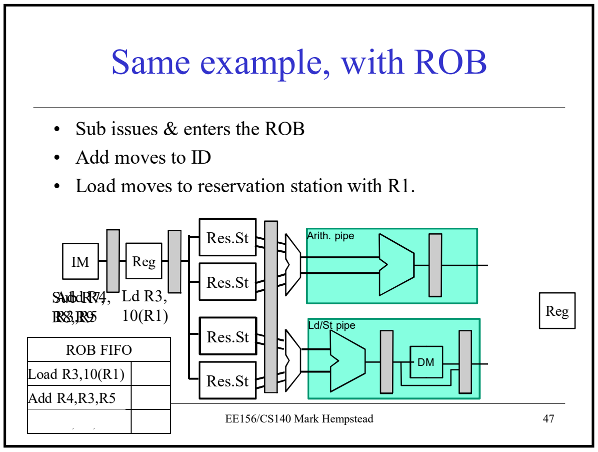
ROB FIFO	
Load R3,10(R1)	



Same example, with ROB

- Add issues & enters the ROB
- Load moves to ID

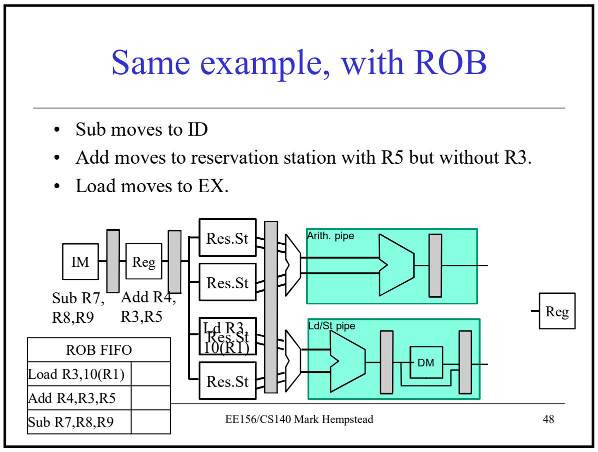
ROB FIFO	
Load R3,10(R1)	



Same example, with ROB

- Sub issues & enters the ROB
- Add moves to ID
- Load moves to reservation station with R1.

ROB FIFO	
Load R3,10(R1)	
Add R4,R3,R5	



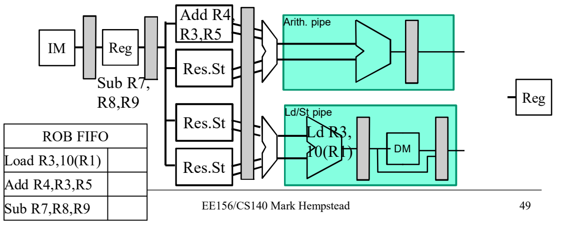
Same example, with ROB

- Sub moves to ID
- Add moves to reservation station with R5 but without R3.
- Load moves to EX.

ROB FIFO	
Load R3,10(R1)	
Add R4,R3,R5	
Sub R7,R8,R9	

Same example, with ROB

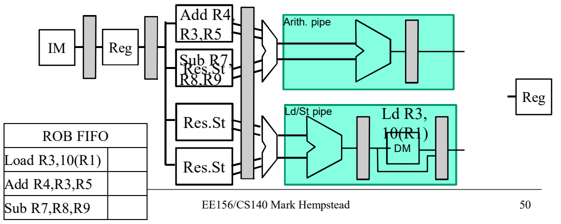
- Sub grabs R8,R9 and moves to a reservation station
- Add is stuck waiting for R3.
- Load moves to Mem and has an L1 miss.



ROB FIFO	
Load R3,10(R1)	
Add R4,R3,R5	
Sub R7,R8,R9	

Same example, with ROB

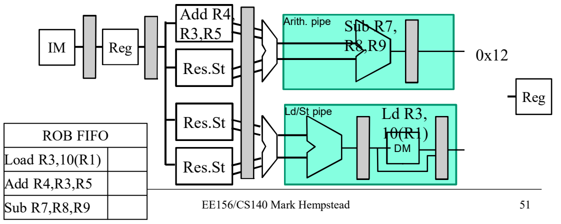
- Sub has both operands; it enters the arith. pipe
- Add is still stuck waiting for R3.
- Load waits in Mem for the L2 to return data.



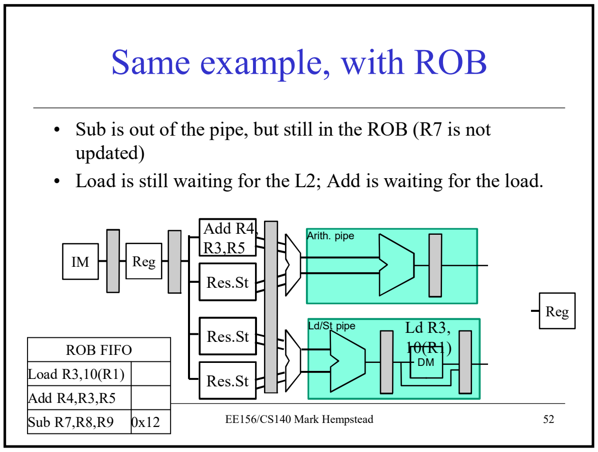
ROB FIFO	
Load R3,10(R1)	
Add R4,R3,R5	
Sub R7,R8,R9	

Same example, with ROB

- Sub exits the pipe, but cannot commit until the add does; its result is saved in the ROB.
- Load is still waiting for the L2; Add is waiting for the load.



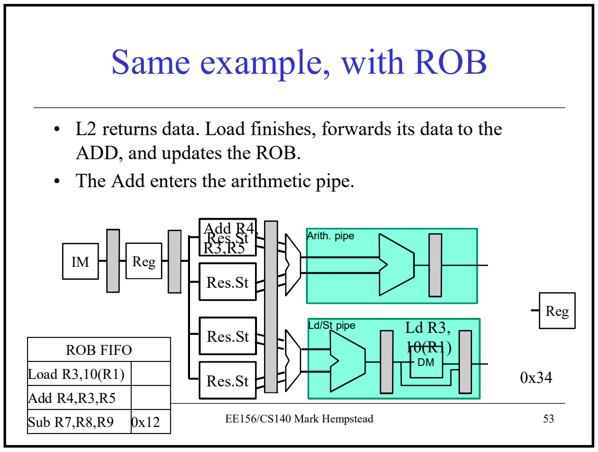
ROB FIFO	
Load R3,10(R1)	
Add R4,R3,R5	
Sub R7,R8,R9	



Same example, with ROB

- Sub is out of the pipe, but still in the ROB (R7 is not updated)
- Load is still waiting for the L2; Add is waiting for the load.

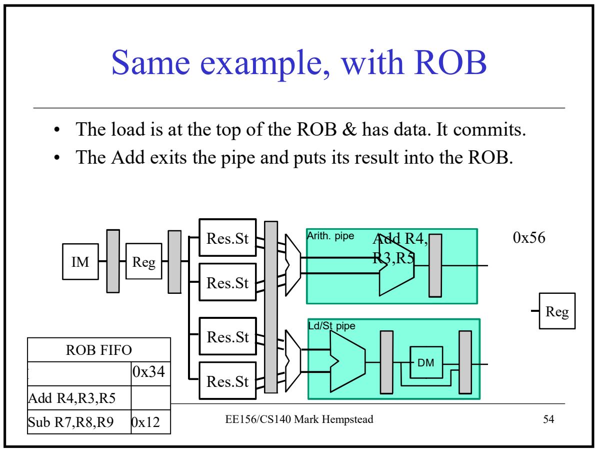
ROB FIFO	
Load R3,10(R1)	
Add R4,R3,R5	
Sub R7,R8,R9	0x12



Same example, with ROB

- L2 returns data. Load finishes, forwards its data to the ADD, and updates the ROB.
- The Add enters the arithmetic pipe.

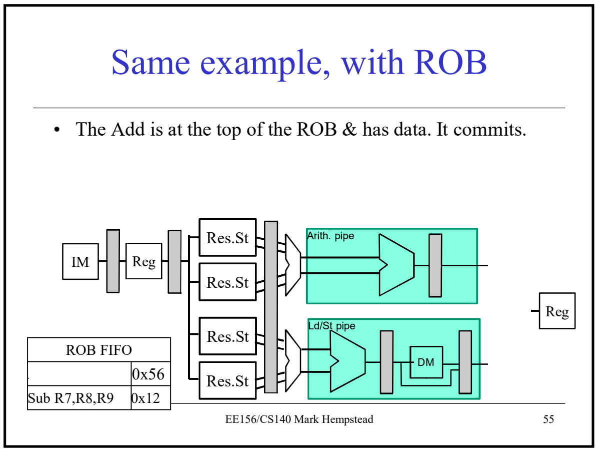
ROB FIFO	
Load R3,10(R1)	
Add R4,R3,R5	
Sub R7,R8,R9	0x12



Same example, with ROB

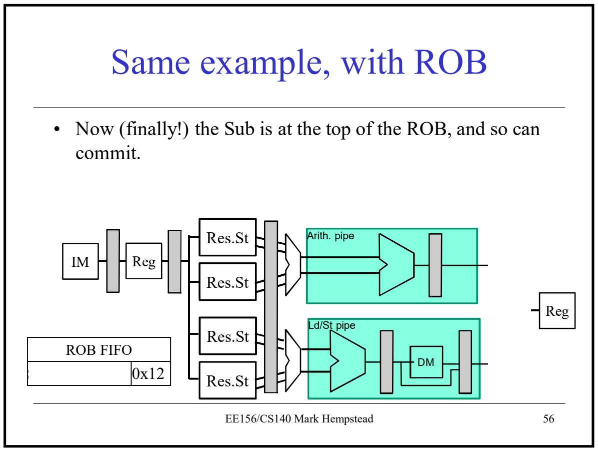
- The load is at the top of the ROB & has data. It commits.
- The Add exits the pipe and puts its result into the ROB.

ROB FIFO	
	0x34
Add R4,R3,R5	
Sub R7,R8,R9	0x12



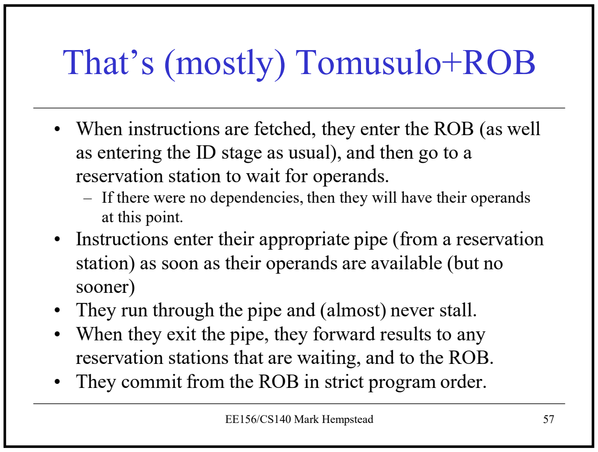
Same example, with ROB

- The Add is at the top of the ROB & has data. It commits.



Same example, with ROB

- Now (finally!) the Sub is at the top of the ROB, and so can commit.



That’s (mostly) Tomusulo+ROB

- When instructions are fetched, they enter the ROB (as well as entering the ID stage as usual), and then go to a reservation station to wait for operands.
  - If there were no dependencies, then they will have their operands at this point.
- Instructions enter their appropriate pipe (from a reservation station) as soon as their operands are available (but no sooner)
- They run through the pipe and (almost) never stall.
- When they exit the pipe, they forward results to any reservation stations that are waiting, and to the ROB.
- They commit from the ROB in strict program order.

What’s left in Tomusulo?

- We’re almost there, except:
  - We haven’t dealt with exceptions, or branch prediction.
  - There’s one more hazard left: hazards through memory.
- But those are all easy, now that we’ve built the infrastructure.

Exceptions

- Precise exceptions are easy.
  - We’ve already ensured that instruction #n does not change architectural state until instruction #n-1 is really truly done.
  - We’ve already talked about not letting an instruction signal an exception until the WB stage
  - Those two guarantee precise exceptions.
- All that’s left: flush the ROB and all reservation stations after an exception, and then restart the PC afterwards.

Branch prediction

- Branch prediction is easy too.
- No instruction can commit until we know it should really execute.
  - all earlier branch predictions are guaranteed to be correct.
- The details:
  - Do branch prediction and branch-target buffer as usual.
  - After any mispredict, flush the ROB of all instructions after the mispredicted branch
  - Restart the PC with the correct branch target

Loads and stores

- Consider the following code:

```
DIVD F3,F1,F2
ST F3, 0(R1)
LD F10, 0(R1)
```

- Problem:
  - The divide is slow
  - The store depends on the divide (via R3), so it waits
  - The load happens right away – before the store.
  - We load the wrong value!
- Is this a new kind of dependency that we've ignored?

Loads and stores

- Is R1 the dependency?

```
DIVD F3,F1,F2
ST F3, 0(R1)
LD F10, 0(R1)
```

- Well, perhaps, but that's not quite it. Consider

```
DIVD F3,F1,F2
ST F3, 0(R1)
LD F10, 10(R1)
```

No problem, even though R1 looks the same

- And consider

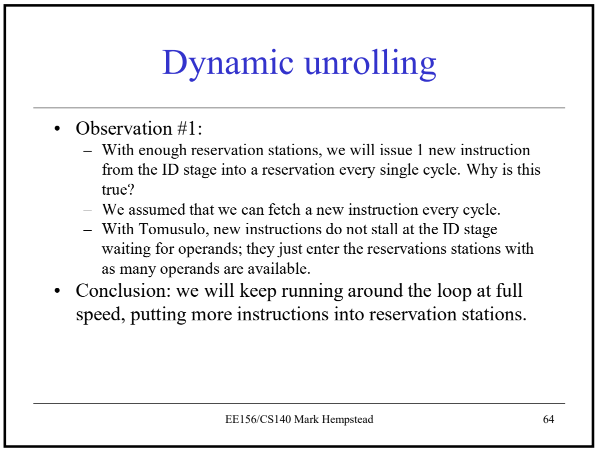
```
DIVD F3,F1,F2
ST F3, 0(R1)
LD F10, 0(R10)
```

A big problem if R1==R10

- Actually, the dependency is the *address*.

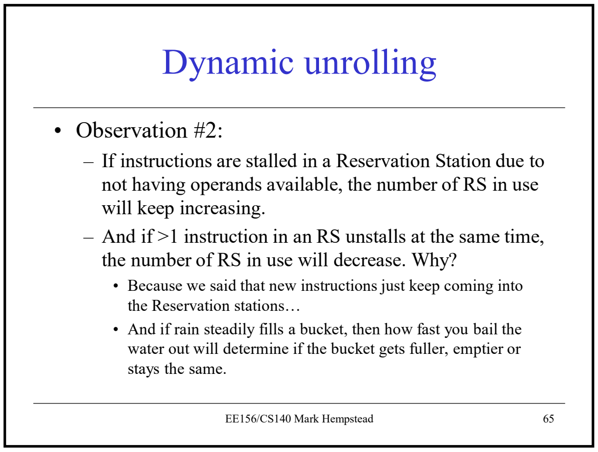
We need more rules

- Memory is architectural state, just like the RF
  - I.e., once you write it, it's hard to undo.
  - And it's a way to pass values from one inst. to another.
  - So put in similar interlocks as for registers.
- The new rules:
  - Stores do not actually happen (i.e., do not go to any cache) until they commit. So they happen in order.
  - Loads cannot leave their reservation station if there is a store with the same address ahead of them.
  - In practice, after computing the effective address, stores are kept in the ROB and not in any reservation station or pipe.



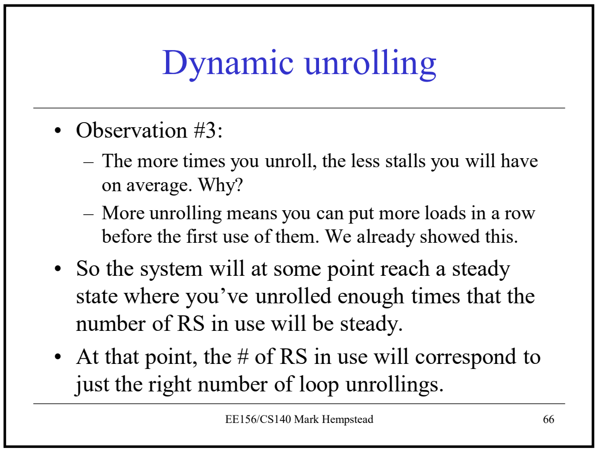
Dynamic unrolling

- Observation #1:
  - With enough reservation stations, we will issue 1 new instruction from the ID stage into a reservation every single cycle. Why is this true?
  - We assumed that we can fetch a new instruction every cycle.
  - With Tomusulo, new instructions do not stall at the ID stage waiting for operands; they just enter the reservations stations with as many operands are available.
- Conclusion: we will keep running around the loop at full speed, putting more instructions into reservation stations.



Dynamic unrolling

- Observation #2:
  - If instructions are stalled in a Reservation Station due to not having operands available, the number of RS in use will keep increasing.
  - And if >1 instruction in an RS unstalls at the same time, the number of RS in use will decrease. Why?
    - Because we said that new instructions just keep coming into the Reservation stations…
    - And if rain steadily fills a bucket, then how fast you bail the water out will determine if the bucket gets fuller, emptier or stays the same.



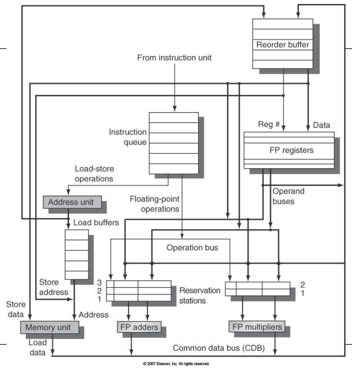
Dynamic unrolling

- Observation #3:
  - The more times you unroll, the less stalls you will have on average. Why?
  - More unrolling means you can put more loads in a row before the first use of them. We already showed this.
- So the system will at some point reach a steady state where you've unrolled enough times that the number of RS in use will be steady.
- At that point, the # of RS in use will correspond to just the right number of loop unrollings.

Extra

TOMOSULO’S ALGORITHM DETAILED EXAMPLE

How this looks physically



Four Steps of Speculative Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue

   If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called “dispatch”)

2. Execution—operate on operands (EX)

   When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW (sometimes called “issue”)

3. Write result—finish execution (WB)

   Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available. (tags are now ROB #s not RS #s)

4. Commit—update register with reorder result

   When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called “graduation”)

Reservation Station Components

- Op: Operation to perform in the unit
- Qj, Qk: Reservation stations producing source registers (value to be written)
  - Note: No ready flags needed as in Scoreboard
  - Qj,Qk=0 => ready
  - Store buffers only have Qi for RS producing result
- Vj, Vk: Value of Source operands
  - Store buffers has V field, result to be stored
- Busy: Indicates reservation station or FU are occupied
- Register Result Status: Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

Tomasulo With Reorder Buffer - Cycle 0

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	No						
0	Mult2	No						

Reservation Stations

	Busy	Address
Load1		
Load2		
Load3		

Entry	Busy	Instruction	State	Destination	Value
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					

Reorder Buffer

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #									
Busy	no	no	no	no	no	no	no		no

Tomasulo With Reorder Buffer - Cycle 1

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	No						
0	Mult2	No						

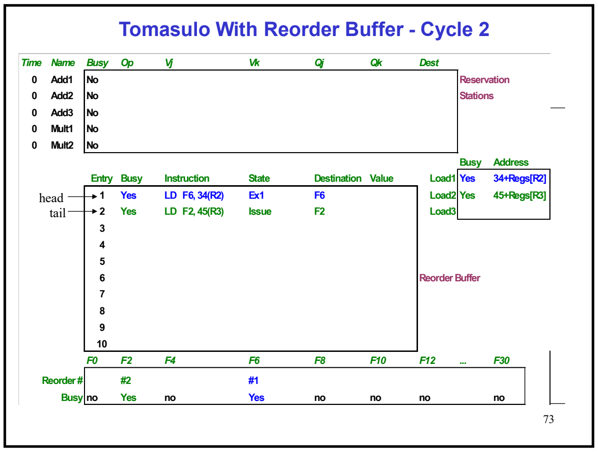
Reservation Stations

	Busy	Address
Load1	Yes	34+Regs[R2]
Load2		
Load3		

Entry	Busy	Instruction	State	Destination	Value
1	Yes	LD F6, 34(R2)	Issue	F6	
2					
3					
4					
5					
6					
7					
8					
9					
10					

Reorder Buffer

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #				#1					
Busy	no	no	no	Yes	no	no	no		no



Tomasulo With Reorder Buffer - Cycle 2

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	No						
0	Mult2	No						

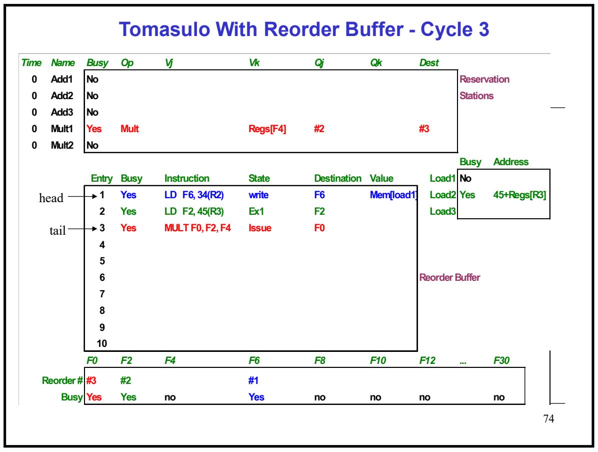
Reservation Stations

	Busy	Address
Load1	Yes	34+Regs[R2]
Load2	Yes	45+Regs[R3]
Load3		

	Entry	Busy	Instruction	State	Destination	Value
head	1	Yes	LD F6, 34(R2)	Ex1	F6	
tail	2	Yes	LD F2, 45(R3)	Issue	F2	
	3					
	4					
	5					
	6					
	7					
	8					
	9					
	10					

Reorder Buffer

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #		#2		#1					
Busy	no	Yes	no	Yes	no	no	no		no



Tomasulo With Reorder Buffer - Cycle 3

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	Yes	Mult		Regs[F4]	#2		#3
0	Mult2	No						

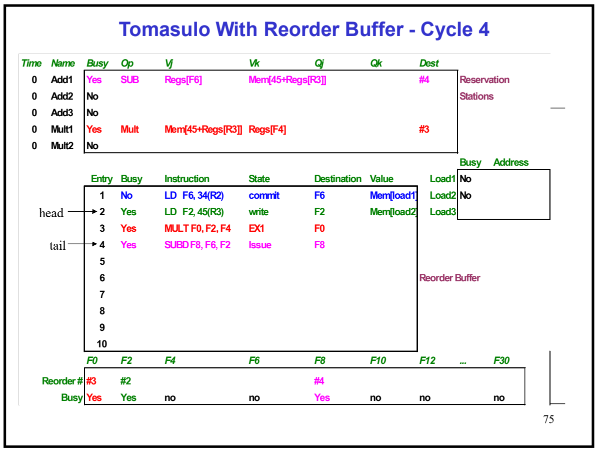
Reservation Stations

	Busy	Address
Load1	No	
Load2	Yes	45+Regs[R3]
Load3		

	Entry	Busy	Instruction	State	Destination	Value
head	1	Yes	LD F6, 34(R2)	write	F6	Mem[load1]
	2	Yes	LD F2, 45(R3)	Ex1	F2	
tail	3	Yes	MULT F0, F2, F4	Issue	F0	
	4					
	5					
	6					
	7					
	8					
	9					
	10					

Reorder Buffer

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3	#2		#1					
Busy	Yes	Yes	no	Yes	no	no	no		no



Tomasulo With Reorder Buffer - Cycle 4

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	Yes	SUB	Regs[F6]	Mem[45+Regs[R3]]			#4
0	Add2	No						
0	Add3	No						
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3
0	Mult2	No						

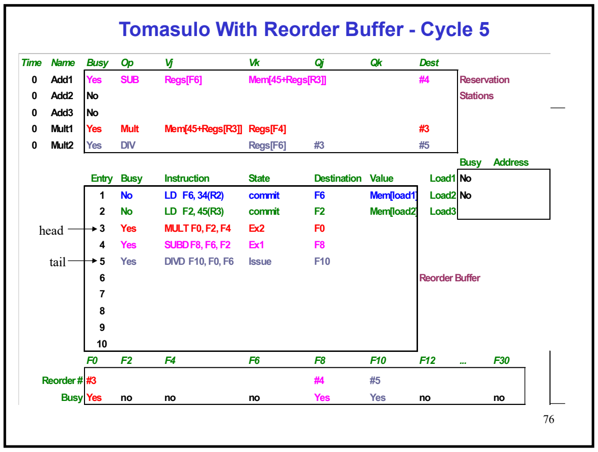
Reservation Stations

	Busy	Address
Load1	No	
Load2	No	
Load3		

	Entry	Busy	Instruction	State	Destination	Value
	1	No	LD F6, 34(R2)	commit	F6	Mem[load1]
head	2	Yes	LD F2, 45(R3)	write	F2	Mem[load2]
	3	Yes	MULT F0, F2, F4	EX1	F0	
tail	4	Yes	SUBD F8, F6, F2	Issue	F8	
	5					
	6					
	7					
	8					
	9					
	10					

Reorder Buffer

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3	#2			#4				
Busy	Yes	Yes	no	no	Yes	no	no		no



Tomasulo With Reorder Buffer - Cycle 5

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	Yes	SUB	Regs[F6]	Mem[45+Regs[R3]]			#4
0	Add2	No						
0	Add3	No						
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3
0	Mult2	Yes	DIV		Regs[F6]	#3		#5

Reservation Stations

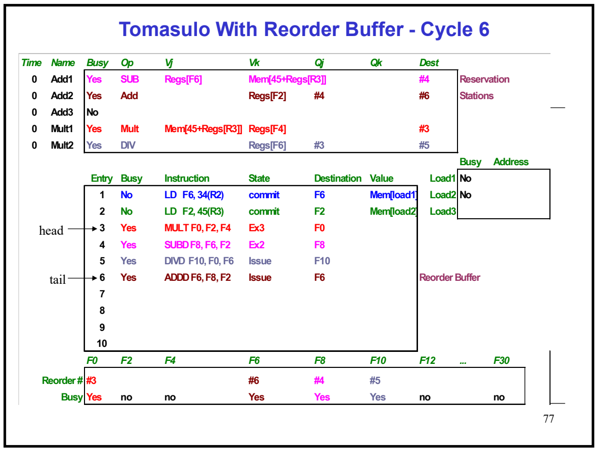
	Busy	Address
Load1	No	
Load2	No	
Load3		

	Entry	Busy	Instruction	State	Destination	Value
	1	No	LD F6, 34(R2)	commit	F6	Mem[load1]
	2	No	LD F2, 45(R3)	commit	F2	Mem[load2]
head →	3	Yes	MULT F0, F2, F4	Ex2	F0	
	4	Yes	SUBD F8, F6, F2	Ex1	F8	
tail →	5	Yes	DIVD F10, F0, F6	Issue	F10	
	6					
	7					
	8					
	9					
	10					

Reorder Buffer

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3				#4	#5			
Busy	Yes	no	no	no	Yes	Yes	no		no

76



Tomasulo With Reorder Buffer - Cycle 6

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	Yes	SUB	Regs[F6]	Mem[45+Regs[R3]]			#4
0	Add2	Yes	Add		Regs[F2]	#4		#6
0	Add3	No						
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3
0	Mult2	Yes	DIV		Regs[F6]	#3		#5

Reservation Stations

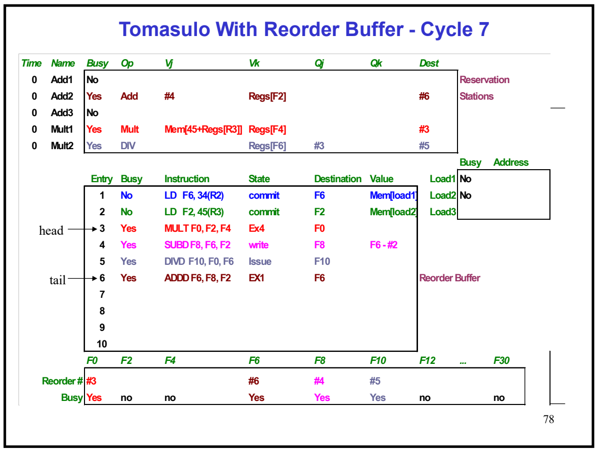
	Busy	Address
Load1	No	
Load2	No	
Load3		

	Entry	Busy	Instruction	State	Destination	Value
	1	No	LD F6, 34(R2)	commit	F6	Mem[load1]
	2	No	LD F2, 45(R3)	commit	F2	Mem[load2]
head →	3	Yes	MULT F0, F2, F4	Ex3	F0	
	4	Yes	SUBD F8, F6, F2	Ex2	F8	
	5	Yes	DIVD F10, F0, F6	Issue	F10	
tail →	6	Yes	ADDD F6, F8, F2	Issue	F6	
	7					
	8					
	9					
	10					

Reorder Buffer

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3			#6	#4	#5			
Busy	Yes	no	no	Yes	Yes	Yes	no		no

77



Tomasulo With Reorder Buffer - Cycle 7

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	Yes	Add	#4	Regs[F2]			#6
0	Add3	No						
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3
0	Mult2	Yes	DIV		Regs[F6]	#3		#5

Reservation Stations

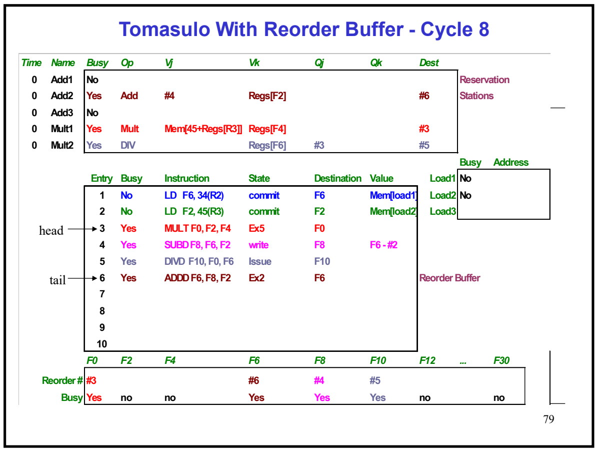
	Busy	Address
Load1	No	
Load2	No	
Load3		

	Entry	Busy	Instruction	State	Destination	Value
	1	No	LD F6, 34(R2)	commit	F6	Mem[load1]
	2	No	LD F2, 45(R3)	commit	F2	Mem[load2]
head →	3	Yes	MULT F0, F2, F4	Ex4	F0	
	4	Yes	SUBD F8, F6, F2	write	F8	F6 - #2
	5	Yes	DIVD F10, F0, F6	Issue	F10	
tail →	6	Yes	ADDD F6, F8, F2	EX1	F6	
	7					
	8					
	9					
	10					

Reorder Buffer

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3			#6	#4	#5			
Busy	Yes	no	no	Yes	Yes	Yes	no		no

78



Tomasulo With Reorder Buffer - Cycle 8

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	Yes	Add	#4	Regs[F2]			#6
0	Add3	No						
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3
0	Mult2	Yes	DIV		Regs[F6]	#3		#5

Reservation Stations

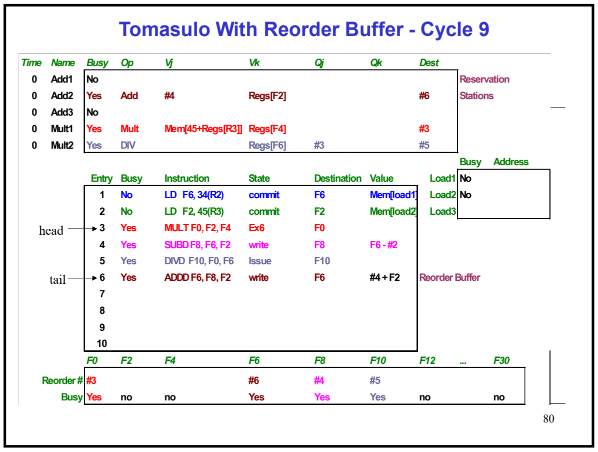
	Busy	Address
Load1	No	
Load2	No	
Load3		

	Entry	Busy	Instruction	State	Destination	Value
	1	No	LD F6, 34(R2)	commit	F6	Mem[load1]
	2	No	LD F2, 45(R3)	commit	F2	Mem[load2]
head →	3	Yes	MULT F0, F2, F4	Ex5	F0	
	4	Yes	SUBD F8, F6, F2	write	F8	F6 - #2
	5	Yes	DIVD F10, F0, F6	Issue	F10	
tail →	6	Yes	ADDD F6, F8, F2	Ex2	F6	
	7					
	8					
	9					
	10					

Reorder Buffer

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3			#6	#4	#5			
Busy	Yes	no	no	Yes	Yes	Yes	no		no

79



Tomasulo With Reorder Buffer - Cycle 9

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	Yes	Add	#4	Regs[F2]			#6
0	Add3	No						
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3
0	Mult2	Yes	DIV		Regs[F6]	#3		#5

Reservation Stations

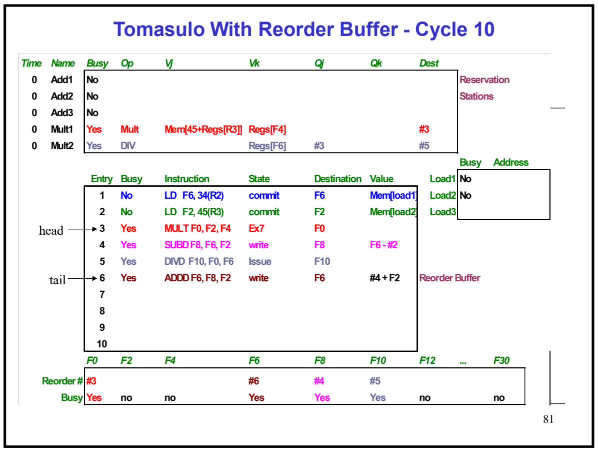
	Busy	Address
Load1	No	
Load2	No	
Load3		

	Entry	Busy	Instruction	State	Destination	Value
	1	No	LD F6, 34(R2)	commit	F6	Mem[load1]
	2	No	LD F2, 45(R3)	commit	F2	Mem[load2]
head →	3	Yes	MULT F0, F2, F4	Ex6	F0	
	4	Yes	SUBD F8, F6, F2	write	F8	F6 - #2
	5	Yes	DIVD F10, F0, F6	Issue	F10	
tail →	6	Yes	ADDD F6, F8, F2	write	F6	#4 + F2
	7					
	8					
	9					
	10					

Reorder Buffer

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3			#6	#4	#5			
Busy	Yes	no	no	Yes	Yes	Yes	no		no

80



Tomasulo With Reorder Buffer - Cycle 10

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3
0	Mult2	Yes	DIV		Regs[F6]	#3		#5

Reservation Stations

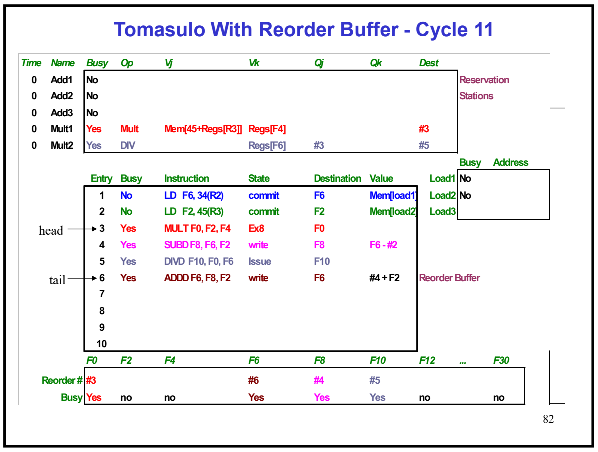
	Busy	Address
Load1	No	
Load2	No	
Load3		

	Entry	Busy	Instruction	State	Destination	Value
	1	No	LD F6, 34(R2)	commit	F6	Mem[load1]
	2	No	LD F2, 45(R3)	commit	F2	Mem[load2]
head →	3	Yes	MULT F0, F2, F4	Ex7	F0	
	4	Yes	SUBD F8, F6, F2	write	F8	F6 - #2
	5	Yes	DIVD F10, F0, F6	Issue	F10	
tail →	6	Yes	ADDD F6, F8, F2	write	F6	#4 + F2
	7					
	8					
	9					
	10					

Reorder Buffer

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3			#6	#4	#5			
Busy	Yes	no	no	Yes	Yes	Yes	no		no

81



Tomasulo With Reorder Buffer - Cycle 11

Reservation Stations

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3
0	Mult2	Yes	DIV		Regs[F6]	#3		#5

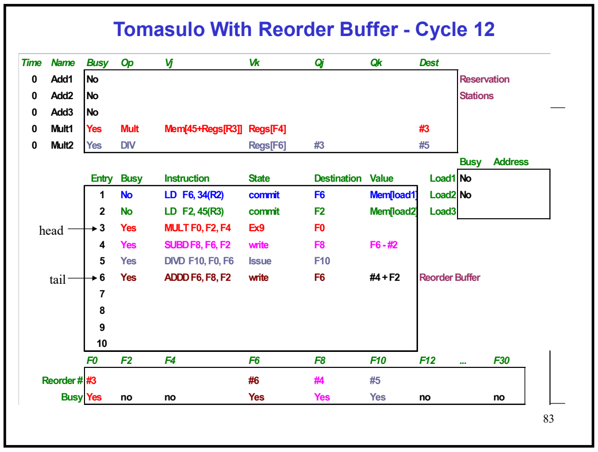
	Busy	Address
Load1	No	
Load2	No	
Load3		

Reorder Buffer

	Entry	Busy	Instruction	State	Destination	Value
	1	No	LD F6, 34(R2)	commit	F6	Mem[load1]
	2	No	LD F2, 45(R3)	commit	F2	Mem[load2]
head →	3	Yes	MULT F0, F2, F4	Ex8	F0	
	4	Yes	SUBD F8, F6, F2	write	F8	F6 - #2
	5	Yes	DIVD F10, F0, F6	Issue	F10	
tail →	6	Yes	ADDD F6, F8, F2	write	F6	#4 + F2
	7					
	8					
	9					
	10					

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3			#6	#4	#5			
Busy	Yes	no	no	Yes	Yes	Yes	no		no

82



Tomasulo With Reorder Buffer - Cycle 12

Reservation Stations

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3
0	Mult2	Yes	DIV		Regs[F6]	#3		#5

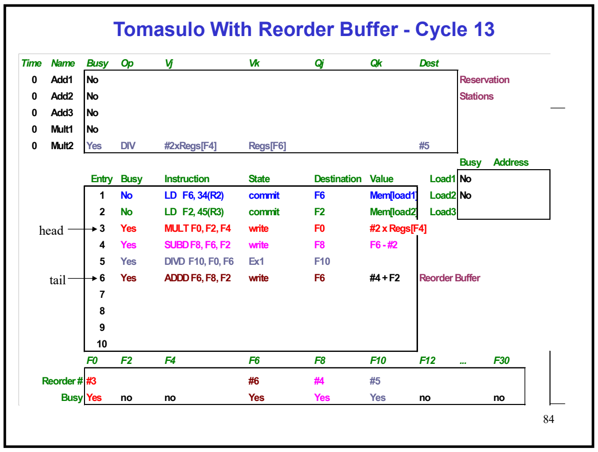
	Busy	Address
Load1	No	
Load2	No	
Load3		

Reorder Buffer

	Entry	Busy	Instruction	State	Destination	Value
	1	No	LD F6, 34(R2)	commit	F6	Mem[load1]
	2	No	LD F2, 45(R3)	commit	F2	Mem[load2]
head →	3	Yes	MULT F0, F2, F4	Ex9	F0	
	4	Yes	SUBD F8, F6, F2	write	F8	F6 - #2
	5	Yes	DIVD F10, F0, F6	Issue	F10	
tail →	6	Yes	ADDD F6, F8, F2	write	F6	#4 + F2
	7					
	8					
	9					
	10					

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3			#6	#4	#5			
Busy	Yes	no	no	Yes	Yes	Yes	no		no

83



Tomasulo With Reorder Buffer - Cycle 13

Reservation Stations

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	No						
0	Mult2	Yes	DIV	#2xRegs[F4]	Regs[F6]			#5

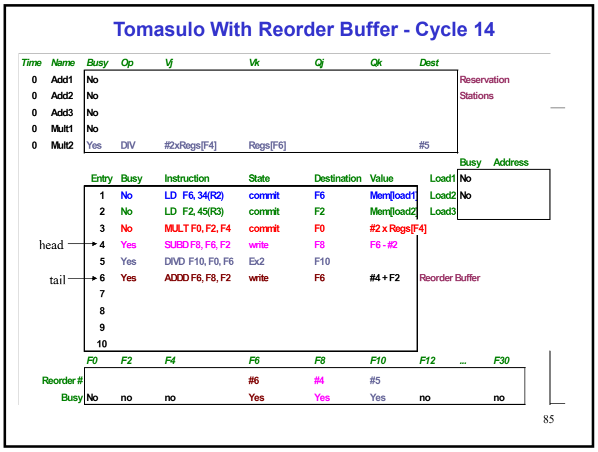
	Busy	Address
Load1	No	
Load2	No	
Load3		

Reorder Buffer

	Entry	Busy	Instruction	State	Destination	Value
	1	No	LD F6, 34(R2)	commit	F6	Mem[load1]
	2	No	LD F2, 45(R3)	commit	F2	Mem[load2]
head →	3	Yes	MULT F0, F2, F4	write	F0	#2 x Regs[F4]
	4	Yes	SUBD F8, F6, F2	write	F8	F6 - #2
	5	Yes	DIVD F10, F0, F6	Ex1	F10	
tail →	6	Yes	ADDD F6, F8, F2	write	F6	#4 + F2
	7					
	8					
	9					
	10					

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3			#6	#4	#5			
Busy	Yes	no	no	Yes	Yes	Yes	no		no

84



Tomasulo With Reorder Buffer - Cycle 14

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	No						
0	Mult2	Yes	DIV	#2xRegs[F4]	Regs[F6]			#5

Reservation Stations

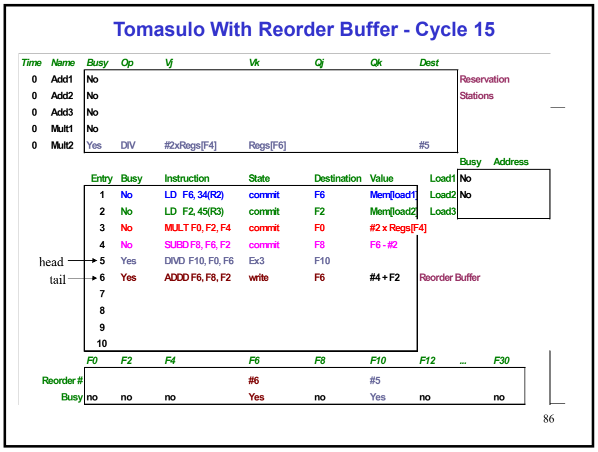
	Busy	Address
Load1	No	
Load2	No	
Load3		

	Entry	Busy	Instruction	State	Destination	Value
	1	No	LD F6, 34(R2)	commit	F6	Mem[load1]
	2	No	LD F2, 45(R3)	commit	F2	Mem[load2]
	3	No	MULT F0, F2, F4	commit	F0	#2 x Regs[F4]
head →	4	Yes	SUBD F8, F6, F2	write	F8	F6 - #2
	5	Yes	DIVD F10, F0, F6	Ex2	F10	
tail →	6	Yes	ADDD F6, F8, F2	write	F6	#4 + F2
	7					
	8					
	9					
	10					

Reorder Buffer

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #				#6	#4	#5			
Busy	No	no	no	Yes	Yes	Yes	no		no

85



Tomasulo With Reorder Buffer - Cycle 15

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	No						
0	Mult2	Yes	DIV	#2xRegs[F4]	Regs[F6]			#5

Reservation Stations

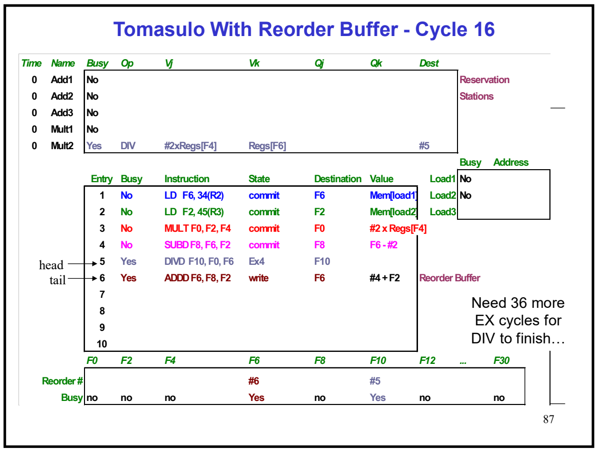
	Busy	Address
Load1	No	
Load2	No	
Load3		

	Entry	Busy	Instruction	State	Destination	Value
	1	No	LD F6, 34(R2)	commit	F6	Mem[load1]
	2	No	LD F2, 45(R3)	commit	F2	Mem[load2]
	3	No	MULT F0, F2, F4	commit	F0	#2 x Regs[F4]
	4	No	SUBD F8, F6, F2	commit	F8	F6 - #2
head →	5	Yes	DIVD F10, F0, F6	Ex3	F10	
tail →	6	Yes	ADDD F6, F8, F2	write	F6	#4 + F2
	7					
	8					
	9					
	10					

Reorder Buffer

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #				#6		#5			
Busy	no	no	no	Yes	no	Yes	no		no

86



Tomasulo With Reorder Buffer - Cycle 16

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	No						
0	Mult2	Yes	DIV	#2xRegs[F4]	Regs[F6]			#5

Reservation Stations

	Busy	Address
Load1	No	
Load2	No	
Load3		

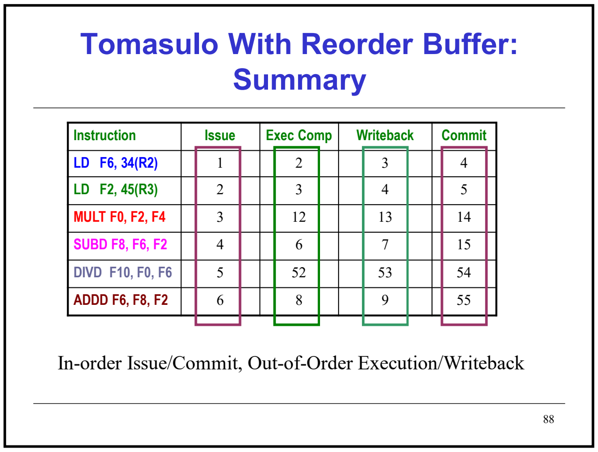
	Entry	Busy	Instruction	State	Destination	Value
	1	No	LD F6, 34(R2)	commit	F6	Mem[load1]
	2	No	LD F2, 45(R3)	commit	F2	Mem[load2]
	3	No	MULT F0, F2, F4	commit	F0	#2 x Regs[F4]
	4	No	SUBD F8, F6, F2	commit	F8	F6 - #2
head →	5	Yes	DIVD F10, F0, F6	Ex4	F10	
tail →	6	Yes	ADDD F6, F8, F2	write	F6	#4 + F2
	7					
	8					
	9					
	10					

Reorder Buffer

Need 36 more EX cycles for DIV to finish…

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #				#6		#5			
Busy	no	no	no	Yes	no	Yes	no		no

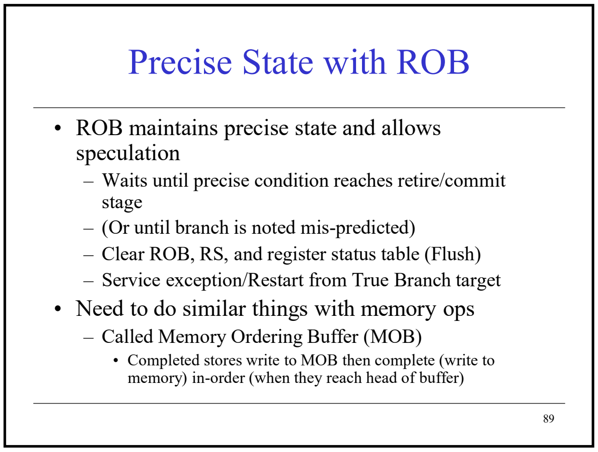
87



Tomasulo With Reorder Buffer: Summary

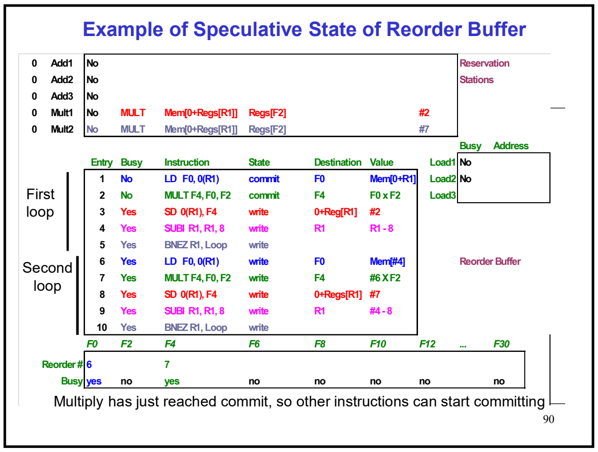
Instruction	Issue	Exec Comp	Writeback	Commit
LD F6, 34(R2)	1	2	3	4
LD F2, 45(R3)	2	3	4	5
MULT F0, F2, F4	3	12	13	14
SUBD F8, F6, F2	4	6	7	15
DIVD F10, F0, F6	5	52	53	54
ADDD F6, F8, F2	6	8	9	55

In-order Issue/Commit, Out-of-Order Execution/Writeback



Precise State with ROB

- ROB maintains precise state and allows speculation
  - Waits until precise condition reaches retire/commit stage
  - (Or until branch is noted mis-predicted)
  - Clear ROB, RS, and register status table (Flush)
  - Service exception/Restart from True Branch target
- Need to do similar things with memory ops
  - Called Memory Ordering Buffer (MOB)
    - Completed stores write to MOB then complete (write to memory) in-order (when they reach head of buffer)



Example of Speculative State of Reorder Buffer

Reservation Stations

0	Add1	No				
0	Add2	No				
0	Add3	No				
0	Mult1	No	MULT	Mem[0+Regs[R1]]	Regs[F2]	#2
0	Mult2	No	MULT	Mem[0+Regs[R1]]	Regs[F2]	#7

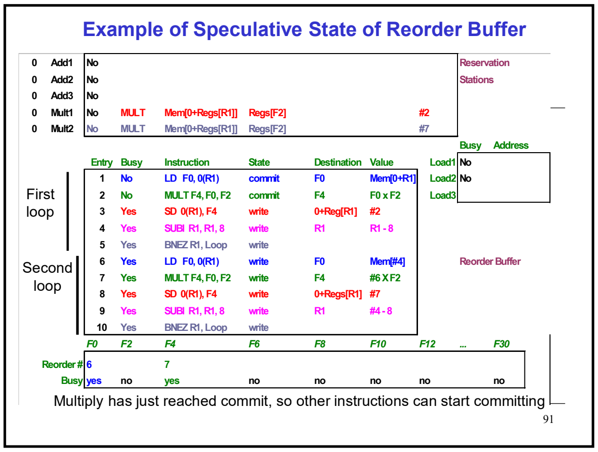
	Busy	Address
Load1	No	
Load2	No	
Load3		

Reorder Buffer

	Entry	Busy	Instruction	State	Destination	Value
First loop	1	No	LD F0, 0(R1)	commit	F0	Mem[0+R1]
	2	No	MULT F4, F0, F2	commit	F4	F0 x F2
	3	Yes	SD 0(R1), F4	write	0+Reg[R1]	#2
	4	Yes	SUBI R1, R1, 8	write	R1	R1 - 8
	5	Yes	BNEZ R1, Loop	write		
Second loop	6	Yes	LD F0, 0(R1)	write	F0	Mem[#4]
	7	Yes	MULT F4, F0, F2	write	F4	#6 X F2
	8	Yes	SD 0(R1), F4	write	0+Regs[R1]	#7
	9	Yes	SUBI R1, R1, 8	write	R1	#4 - 8
	10	Yes	BNEZ R1, Loop	write		

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	6		7						
Busy	yes	no	yes	no	no	no	no		no

Multiply has just reached commit, so other instructions can start committing



Example of Speculative State of Reorder Buffer

							Reservation Stations
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	No	MULT	Mem[0+Regs[R1]]	Regs[F2]	#2	
0	Mult2	No	MULT	Mem[0+Regs[R1]]	Regs[F2]	#7	

	Busy	Address
Load1	No	
Load2	No	
Load3		

Reorder Buffer

	Entry	Busy	Instruction	State	Destination	Value
First loop	1	No	LD F0, 0(R1)	commit	F0	Mem[0+R1]
	2	No	MULT F4, F0, F2	commit	F4	F0 x F2
	3	Yes	SD 0(R1), F4	write	0+Reg[R1]	#2
	4	Yes	SUBI R1, R1, 8	write	R1	R1 - 8
	5	Yes	BNEZ R1, Loop	write		
Second loop	6	Yes	LD F0, 0(R1)	write	F0	Mem[#4]
	7	Yes	MULT F4, F0, F2	write	F4	#6 X F2
	8	Yes	SD 0(R1), F4	write	0+Regs[R1]	#7
	9	Yes	SUBI R1, R1, 8	write	R1	#4 - 8
	10	Yes	BNEZ R1, Loop	write		

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
Reorder #	6		7						
Busy	yes	no	yes	no	no	no	no		no

Multiply has just reached commit, so other instructions can start committing

91