CPEN 411: Computer Architecture

Slide Set #9: Reorder Buffer, Exceptions, Multiple Instruction Issue

original slides by Tor Aamodt

aamodt@ece.ubc.ca

Background: PowerPC 604, one of the first microprocessors with a reorder buffer

Introduction to Slide Set 9

- Out-of-order execution with Tomasulo’s algorithm allows an instruction to execute when its source operands are available. Branch prediction enables us to fetch and decode instructions after a branch before that branch is “resolved”.

- To increase performance (reduce CPI) we would like to start executing the instructions fetched and decoded following the predicted branch *before* that branch is resolved. However, we have a problem: out-of-order execution (e.g., using Tomasulo’s algorithm) allows an instruction to update the register file before the branch is “resolved”.

- An elegant solution is a hardware structure known as the “re-order buffer”. One reason for calling it an “elegant solution” is it also enables “virtual memory” (Slide Set 12) by supporting an execution model known as “precise exceptions”.

Learning Objectives

After we finish this set of slides you should be able to:

- Explain the motivation for using a reorder buffer and how a reorder buffer operates.
- Define multiple instruction issue and explain the motivation for using it.
- List two approaches to achieving multiple instruction issue
- Evaluate the timing of a processor using multiple issue and Tomasulo's algorithm.
- Evaluate the timing of a processor using multiple issue, Tomasulo's algorithm and a reorder buffer.
- Define what an exception is and list several categories of exceptions.
- Explain what is meant by a precise exception.
- Explain how the reorder buffer supports precise exceptions.

Speculative Execution

```
       DIVD     R3,R1,R2 ; F:1, D:2, I:3, X:4, W:104
       BEQZ     R3,Label       ; F:2, D:3, I:4, X:105 (“not taken”)
       …                 branch predicted “taken” on cycle 2
Label: DMUL    R4,R4,R2       ; F:3, D:4, I:5, X:6, W:?
```

Goal: Support execution of instructions fetched following a branch prediction before we know if the prediction is correct. Above: Want to execute and broadcast result of DMUL “speculatively” long before branch resolved on cycle 105.

Problem: Such instructions should not update register file because branch might have been predicted incorrectly. Above: cannot let DMUL write to register file until control hazard resolved.

The Reorder Buffer (ROB)

- An elegant solution to this problem is a hardware structure known as a “Reorder Buffer” (ROB).

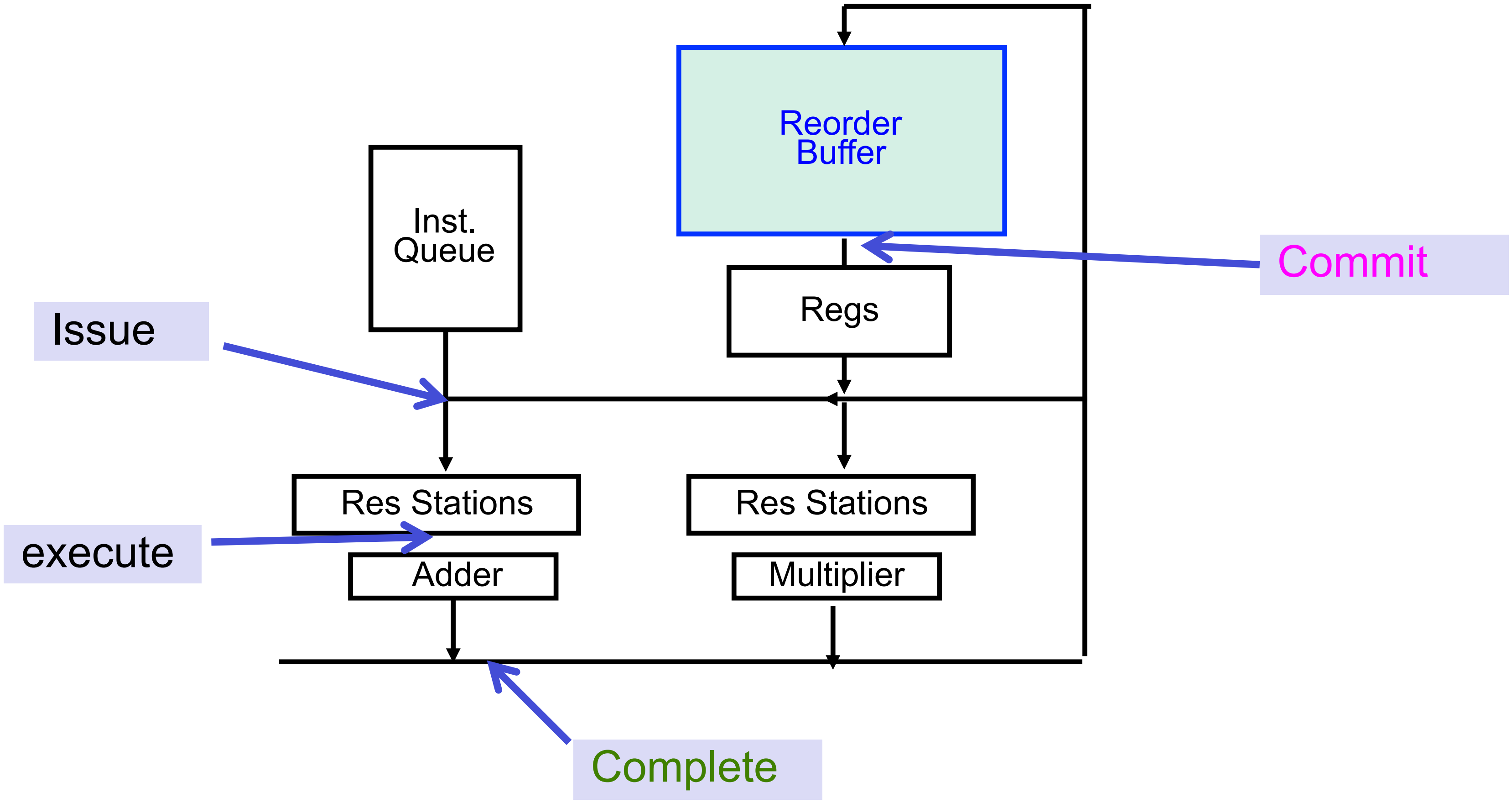
- The reorder buffer is used to divide the “writeback” step into two separate steps: “instruction completion” and “instruction commit”.

- An instruction enters “instruction completion” when it finishes execution.
  - Instruction broadcasts result on the CDB so dependent instructions can begin execution.
  - However, register file not updated yet.
  - Instead, the result is “buffered” in the reorder buffer to be written into the register file later.

- An instruction enters “instruction commit” when it is the oldest instruction that has completed (and is free from “exceptions”).
  - Writes results to register file

The Reorder Buffer (ROB)

Pipeline stages: fetch, decode, issue, execute, complete, commit



The Reorder Buffer (ROB)

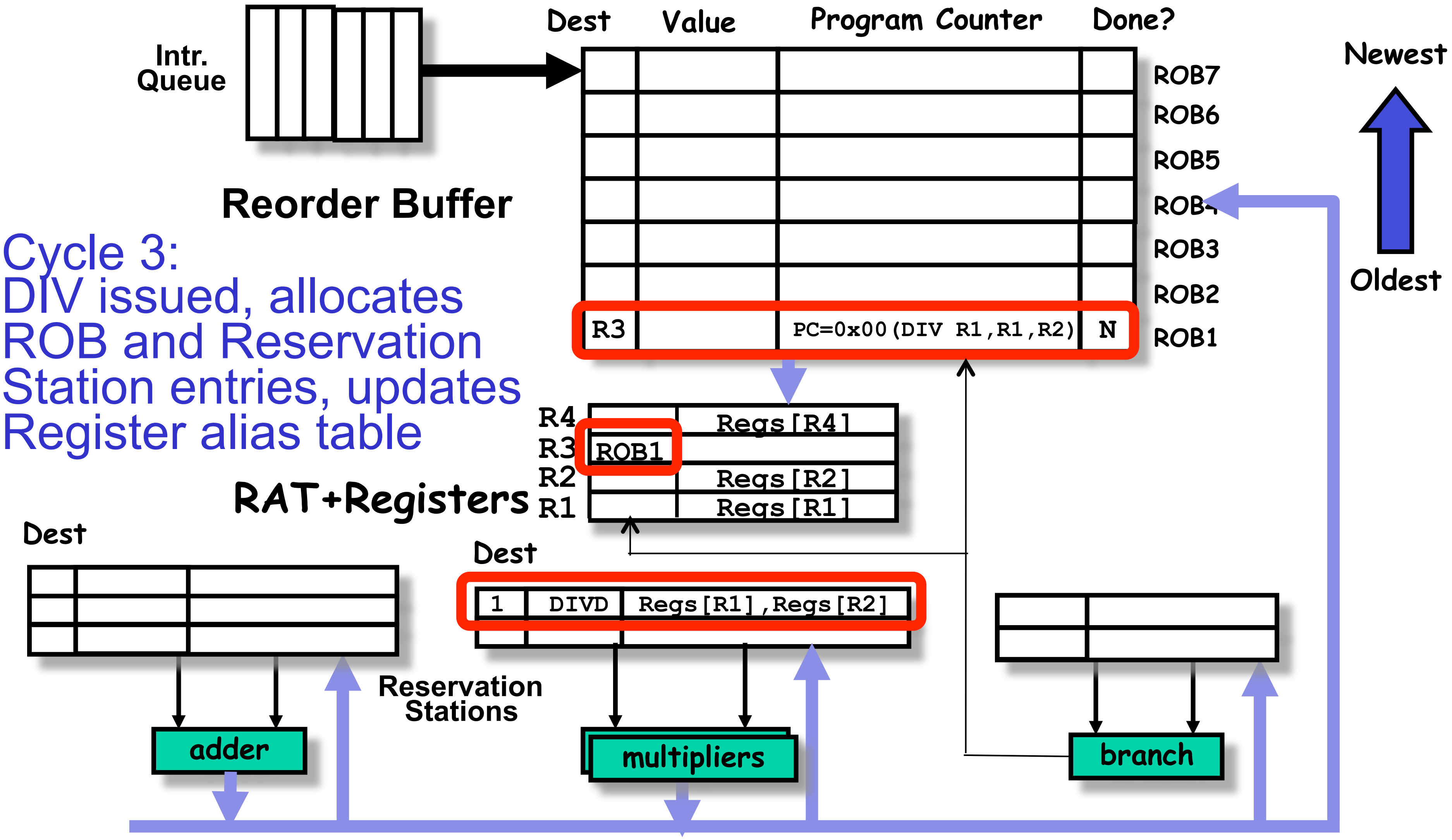
- The Reorder Buffer (ROB) maintains a “window” of dynamic instructions currently in the pipeline.

- The instructions are stored in “first-in first-out” order. That is, the order originally specified by the programmer). Entry allocated in ROB during issue at same time as a reservation station is allocated.

- Each ROB entry contains the instruction’s PC, destination register number, destination register value, and an execution status flag.

- To make it easier to update the ROB after an instruction executes and to enable easy lookup of results buffered in the ROB but not yet written to the register file we use the ROB entry of the instruction as the tag for renaming instead of reservation station ID.

Tomasulo With Reorder Buffer



Cycle 3:
DIV issued, allocates ROB and Reservation Station entries, updates Register alias table

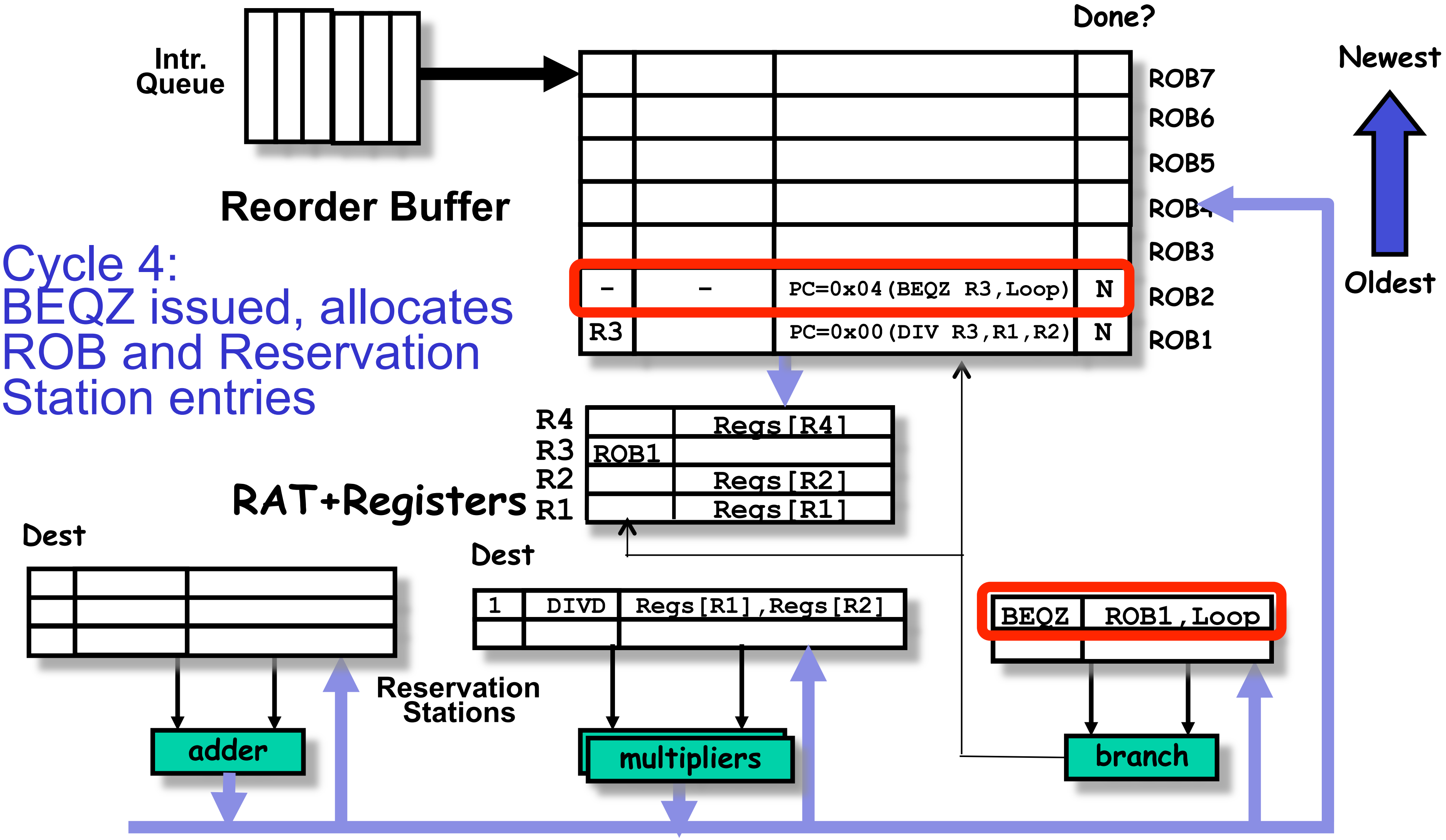
	Dest	Value	Program Counter	Done?
ROB7				
ROB6				
ROB5				
ROB4				
ROB3				
ROB2				
ROB1	R3		PC=0x00(DIV R1,R1,R2)	N

R4		Regs[R4]
R3	ROB1	
R2		Regs[R2]
R1		Regs[R1]

Dest		
1	DIVD	Regs[R1],Regs[R2]

Tomasulo With Reorder Buffer

Cycle 4:
BEQZ issued, allocates ROB and Reservation Station entries



			Done?	
				ROB7
				ROB6
				ROB5
				ROB4
				ROB3
-	-	PC=0x04(BEQZ R3,Loop)	N	ROB2
R3		PC=0x00(DIV R3,R1,R2)	N	ROB1

R4		Regs[R4]
R3	ROB1	
R2		Regs[R2]
R1		Regs[R1]

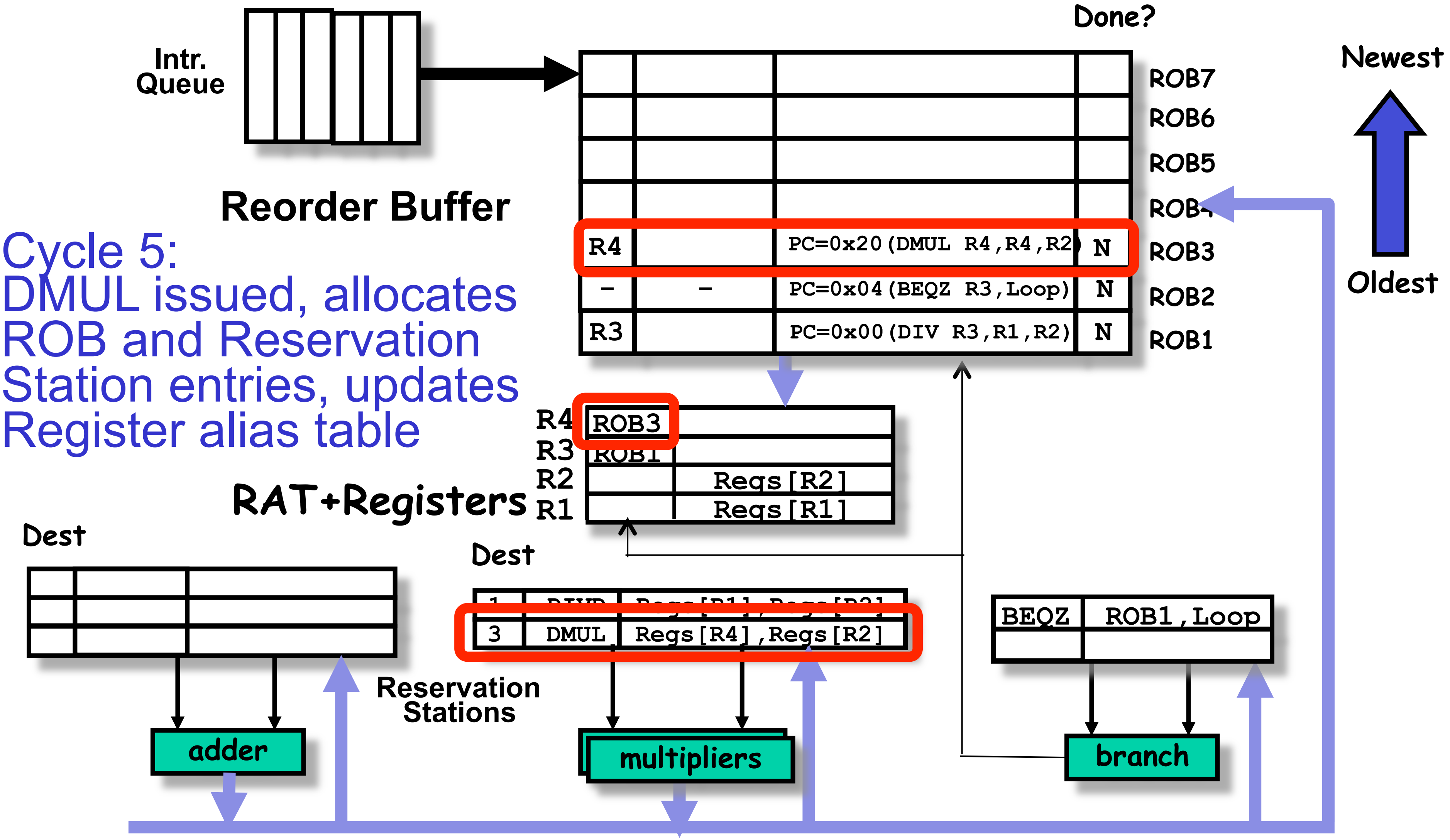
Dest		

Dest		
1	DIVD	Regs[R1],Regs[R2]

BEQZ	ROB1,Loop

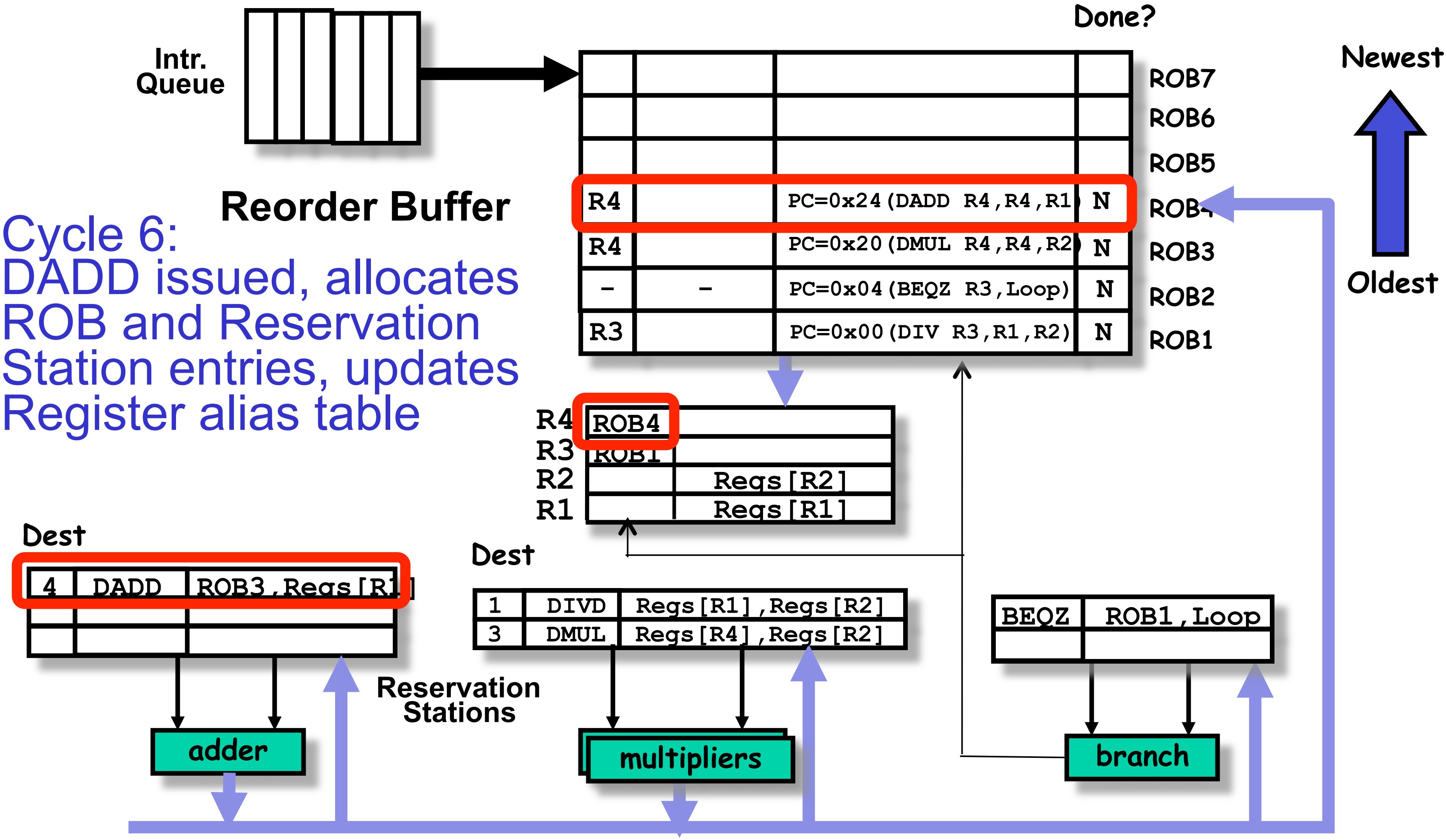
Tomasulo With Reorder Buffer

Cycle 5:
DMUL issued, allocates ROB and Reservation Station entries, updates Register alias table



Tomasulo With Reorder Buffer

Cycle 6:
DADD issued, allocates ROB and Reservation Station entries, updates Register alias table



Reorder Buffer:

			Done?	
				ROB7
				ROB6
				ROB5
R4		PC=0x24 (DADD R4,R4,R1)	N	ROB4
R4		PC=0x20 (DMUL R4,R4,R2)	N	ROB3
-	-	PC=0x04 (BEQZ R3,Loop)	N	ROB2
R3		PC=0x00 (DIV R3,R1,R2)	N	ROB1

Register alias table:

R4	ROB4	
R3	ROB1	
R2		Regs[R2]
R1		Regs[R1]

Adder reservation stations:

Dest		
4	DADD	ROB3,Regs[R1]

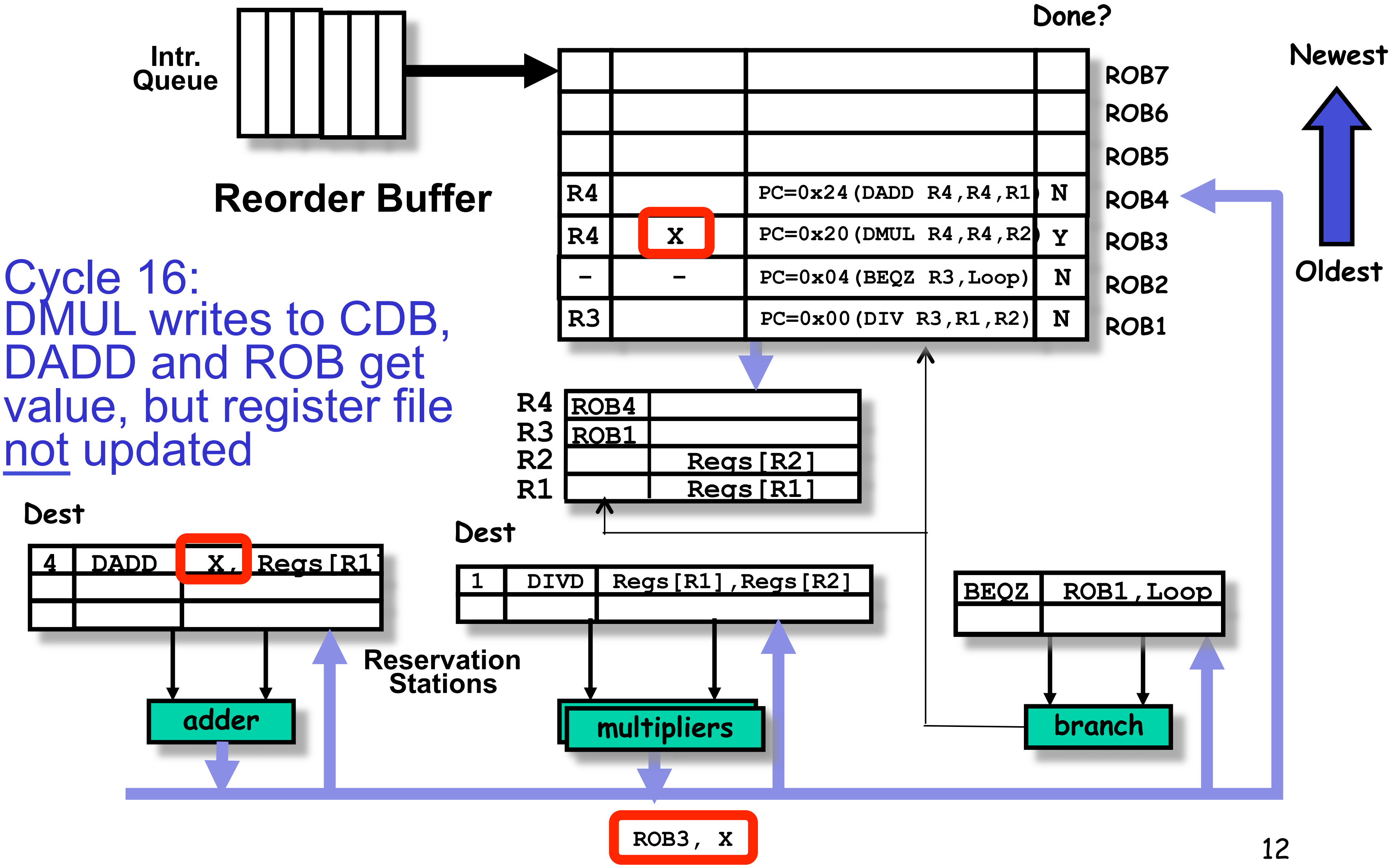
Multiplier reservation stations:

Dest		
1	DIVD	Regs[R1],Regs[R2]
3	DMUL	Regs[R4],Regs[R2]

Branch reservation stations:

BEQZ	ROB1,Loop

Tomasulo With Reorder Buffer



				ROB7
				ROB6
				ROB5
R4		PC=0x24 (DADD R4,R4,R1)	N	ROB4
R4	X	PC=0x20 (DMUL R4,R4,R2)	Y	ROB3
-	-	PC=0x04 (BEQZ R3,Loop)	N	ROB2
R3		PC=0x00 (DIV R3,R1,R2)	N	ROB1

R4	ROB4	
R3	ROB1	
R2		Regs[R2]
R1		Regs[R1]

4	DADD	X, Regs[R1]

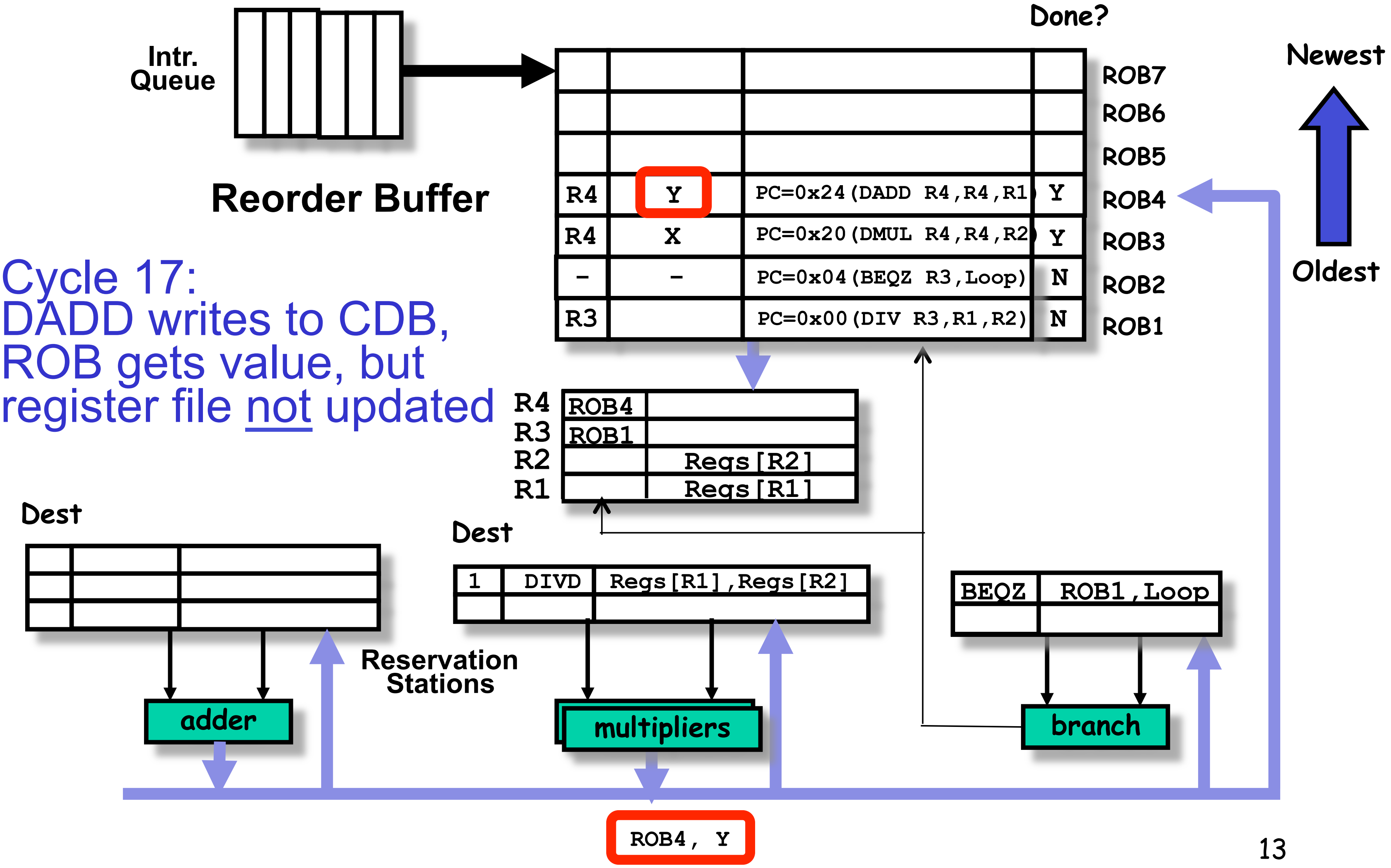
1	DIVD	Regs[R1],Regs[R2]

BEQZ	ROB1,Loop

Cycle 16:
DMUL writes to CDB, DADD and ROB get value, but register file not updated

Tomasulo With Reorder Buffer

Cycle 17:
DADD writes to CDB, ROB gets value, but register file not updated



			Done?	
				ROB7
				ROB6
				ROB5
R4	Y	PC=0x24 (DADD R4,R4,R1)	Y	ROB4
R4	X	PC=0x20 (DMUL R4,R4,R2)	Y	ROB3
-	-	PC=0x04 (BEQZ R3,Loop)	N	ROB2
R3		PC=0x00 (DIV R3,R1,R2)	N	ROB1

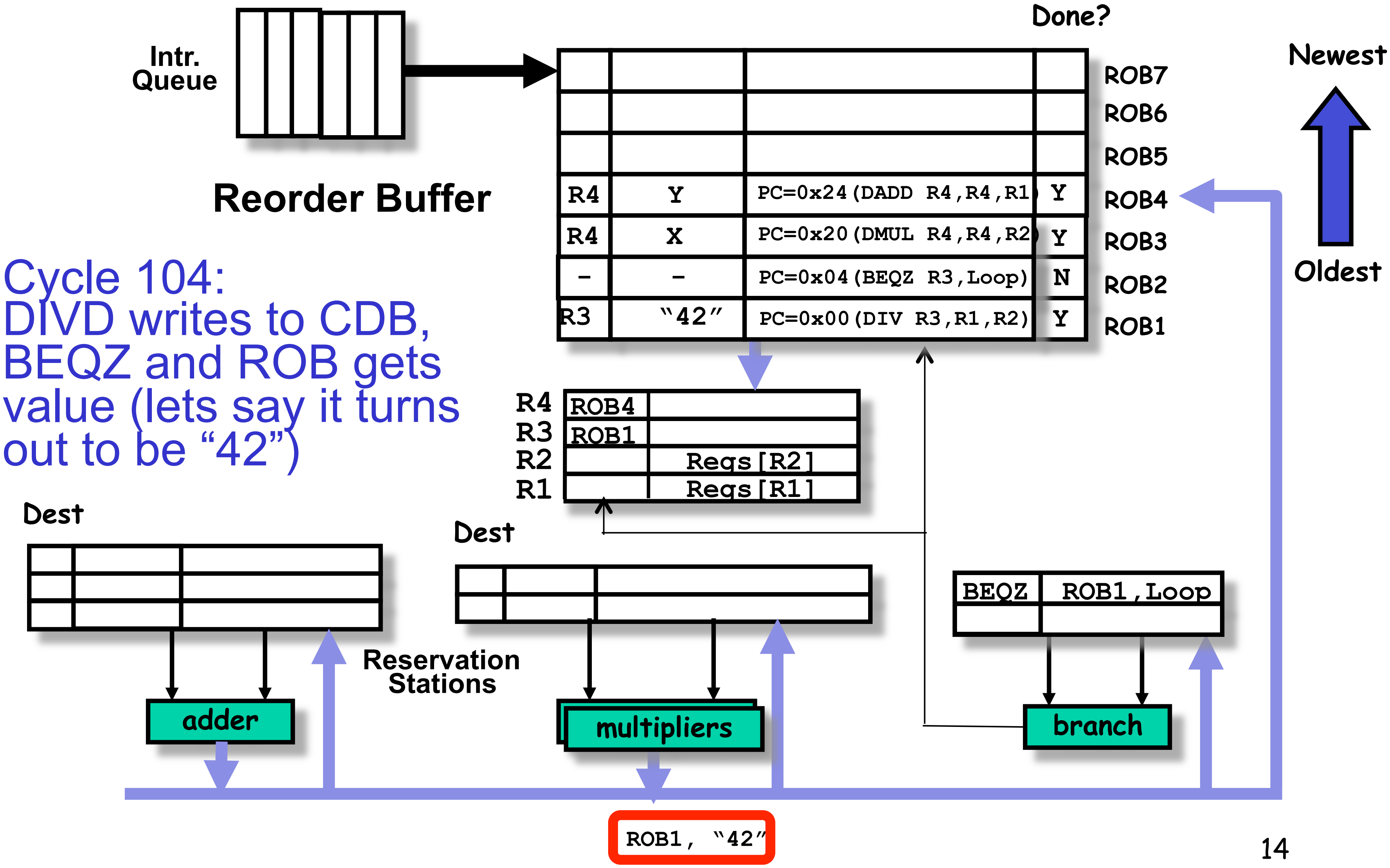
R4	ROB4	
R3	ROB1	
R2		Regs[R2]
R1		Regs[R1]

Dest		
1	DIVD	Regs[R1],Regs[R2]

BEQZ	ROB1,Loop

Tomasulo With Reorder Buffer

Cycle 104:
DIVD writes to CDB, BEQZ and ROB gets value (lets say it turns out to be “42”)

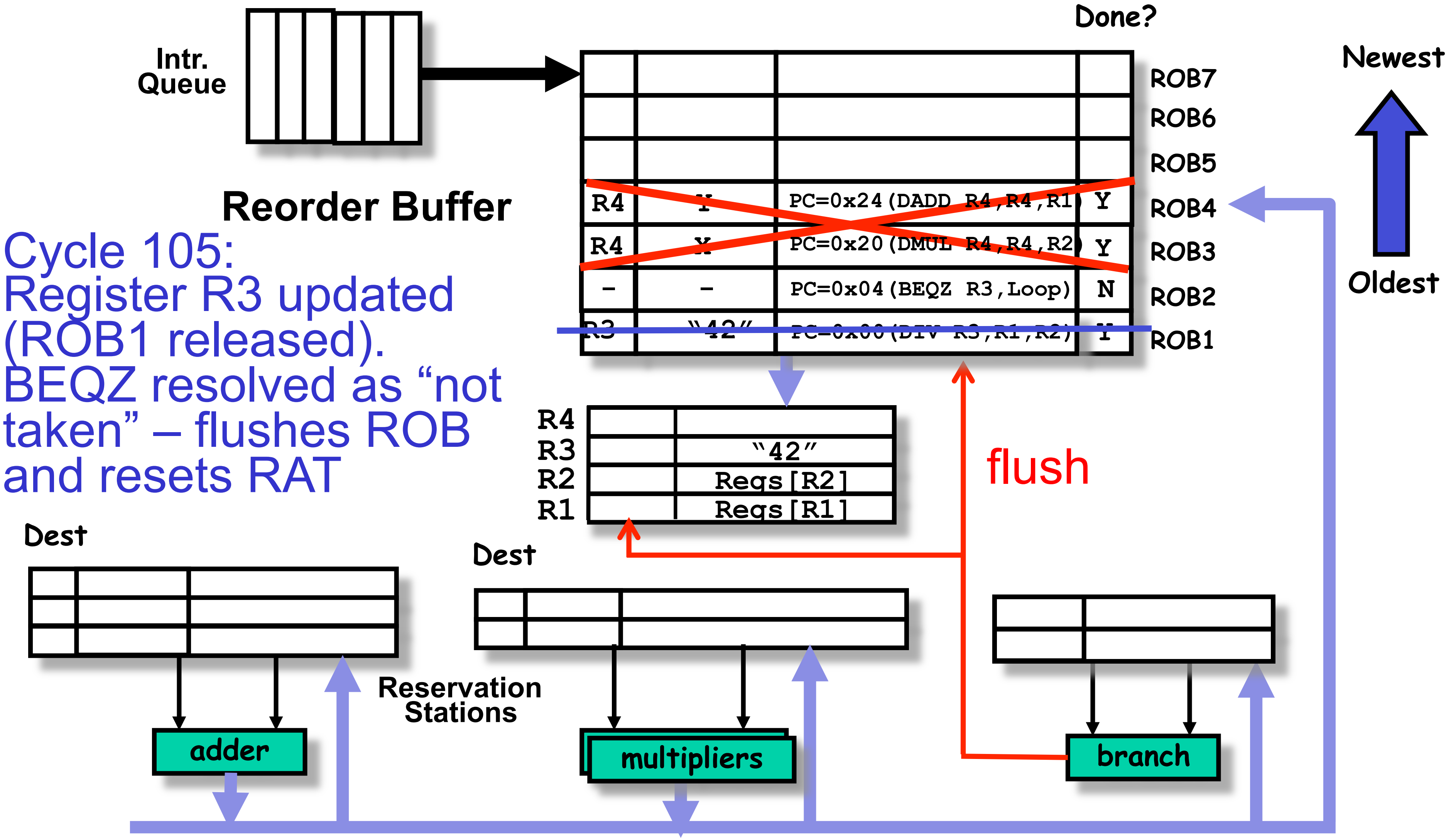


			Done?	
				ROB7
				ROB6
				ROB5
R4	Y	PC=0x24(DADD R4,R4,R1)	Y	ROB4
R4	X	PC=0x20(DMUL R4,R4,R2)	Y	ROB3
-	-	PC=0x04(BEQZ R3,Loop)	N	ROB2
R3	“42”	PC=0x00(DIV R3,R1,R2)	Y	ROB1

R4	ROB4	
R3	ROB1	
R2		Regs[R2]
R1		Regs[R1]

BEQZ	ROB1,Loop

Tomasulo With Reorder Buffer



Cycle 105:
Register R3 updated (ROB1 released).
BEQZ resolved as “not taken” – flushes ROB and resets RAT

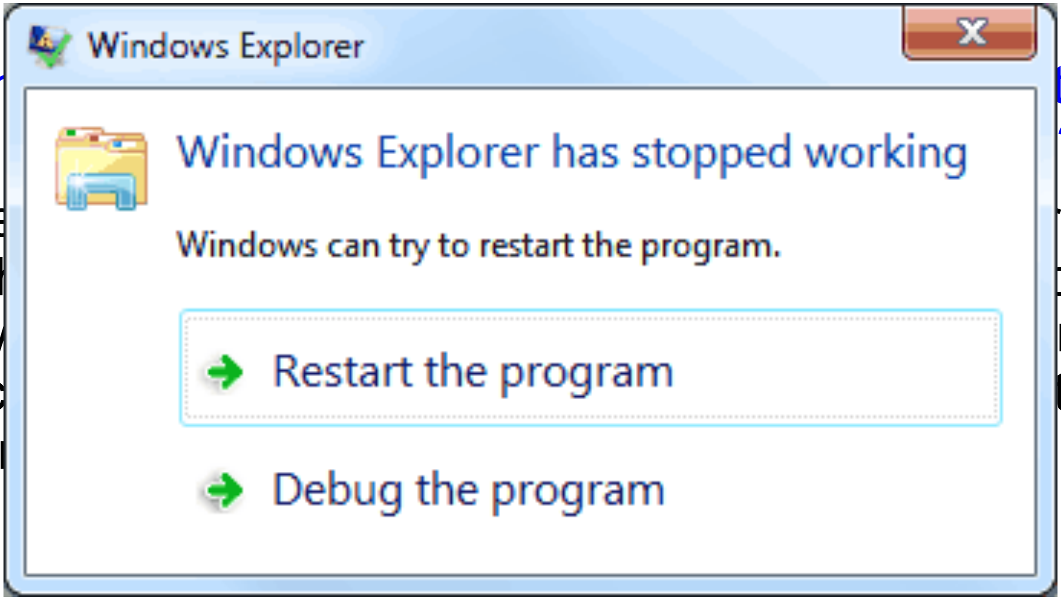
			Done?	
				ROB7
				ROB6
				ROB5
~~R4~~	~~Y~~	~~PC=0x24 (DADD R4,R4,R1)~~	~~Y~~	ROB4
~~R4~~	~~X~~	~~PC=0x20 (DMUL R4,R4,R2)~~	~~Y~~	ROB3
-	-	PC=0x04 (BEQZ R3,Loop)	N	ROB2
~~R3~~	~~“42”~~	~~PC=0x00 (DIV R3,R1,R2)~~	~~Y~~	ROB1

R4		
R3		“42”
R2		Regs[R2]
R1		Regs[R1]

Exceptions

- So far, we have considered how to design hardware to support higher performance execution of a program, but we assumed we can always execute an instruction.

- What h … the program
  needs … ?
- If there … e to do
  someth … y zero, or
  read/w … reason an
  instruc … to virtual
  memo
- We ca



Windows Explorer has stopped working

Windows can try to restart the program.

- The hardware detects exceptions and has a special program run that either handles the exception, or passes control to the operating system, which may handle the exception or end the program.

Exceptions

- Exceptions are also sometimes referred to “interrupts”, or “faults”. We’ll just call them all “exceptions”.
- Generally they require transfer of control to the operating system (OS) or other software “handler” routine.
- Causes:
  - I/O device request
  - Invoking an OS service from a user program
  - Page fault
  - Memory protection violation
  - FP arithmetic anomaly
  - Integer arithmetic overflow
  - etc...

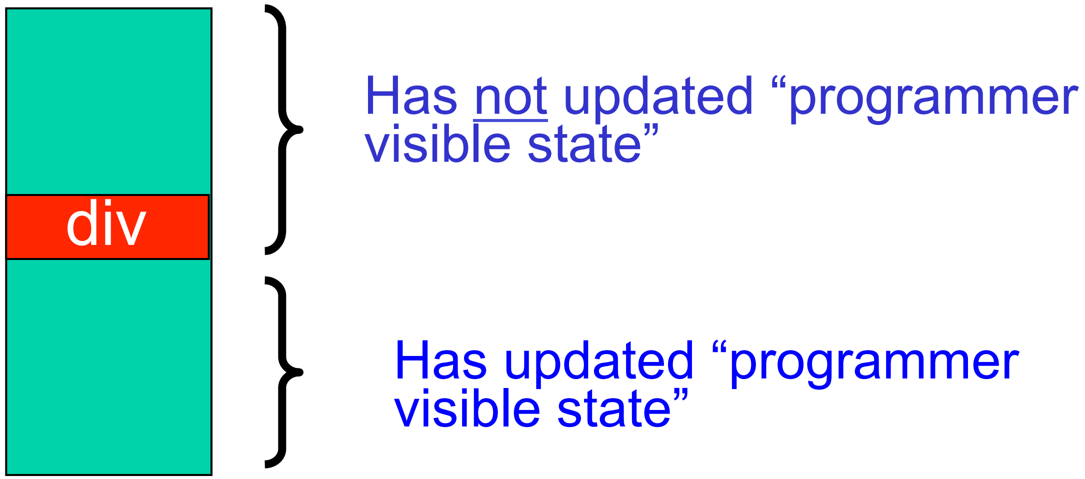
Categories of Exceptions

- Synchronous versus Asynchronous
  - Async: external device or hardware failure.
- User requested versus coerced
  - user requested => handle after instruction
- User maskable versus user nonmaskable
  - e.g., overflow can be masked (ignored) by user.
- Within versus between instructions
  - within => hard
- Resume versus terminate
  - terminate => easier

Most challenging, but very important => required for supporting “virtual memory” (which we will talk about later).

Precise Exceptions

- If hardware guarantees that ALL instructions before faulting instruction have updated programmer visible state (i.e., R1...R31, memory) and NONE after (and including) faulting instruction have updated programmer visible state, then pipeline is said to support **precise exceptions**.



div

Has not updated “programmer visible state”

Has updated “programmer visible state”

Precise Exceptions?

- Both Scoreboard and Tomasulo have:
  - In-order issue, out-of-order execution, out-of-order completion

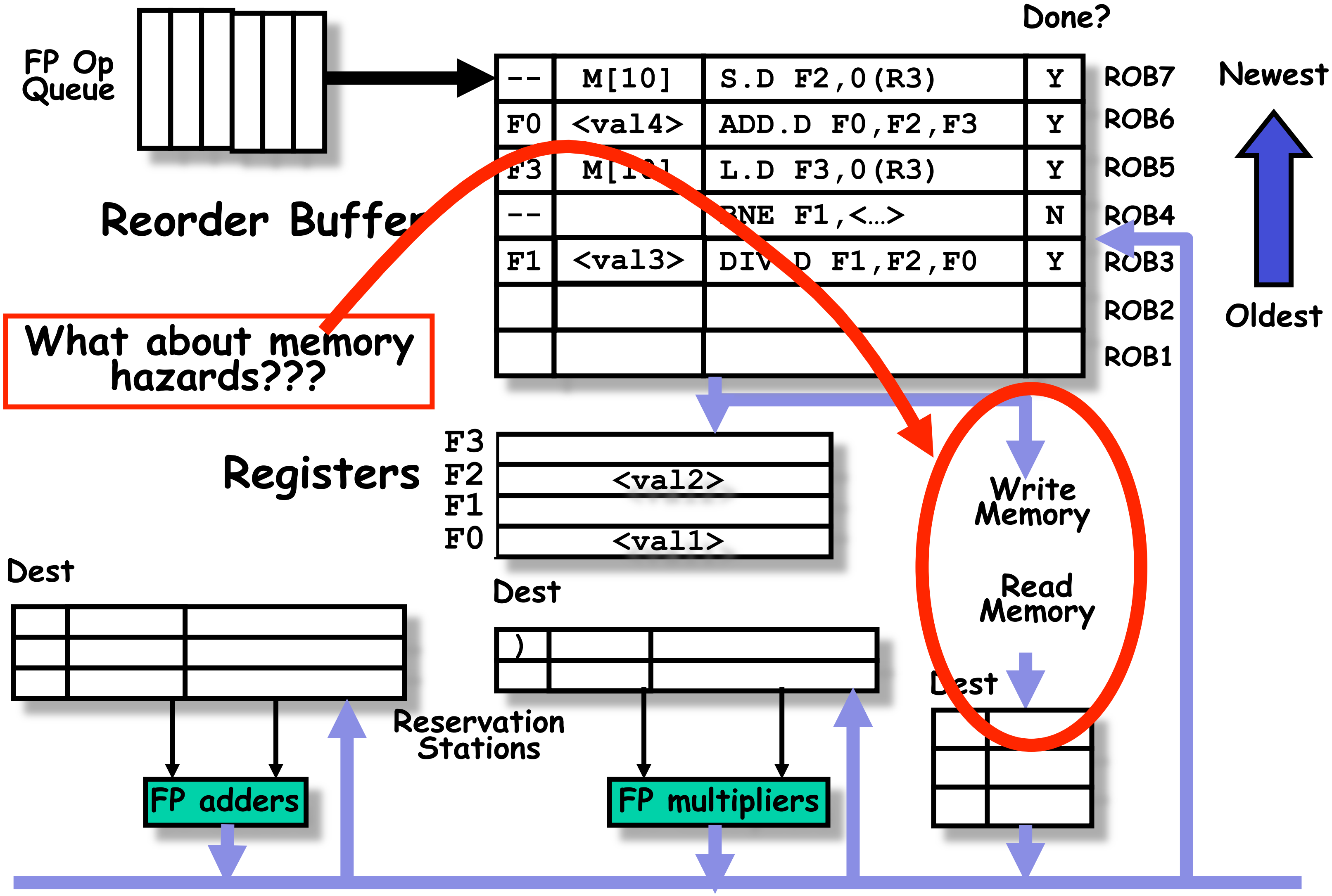
- Out-of-order completion means that interrupts are not precise

- Need way to “resynchronize” execution with instruction stream
  - Easiest way is with reorder buffer

Consider the following sequence of instructions:

```
    SD R5,0(R2)
LD R6,0(R3)
```

Memory Dependencies



			Done?	
--	M[10]	S.D F2,0(R3)	Y	ROB7
F0	<val4>	ADD.D F0,F2,F3	Y	ROB6
F3	M[10]	L.D F3,0(R3)	Y	ROB5
--		BNE F1,<…>	N	ROB4
F1	<val3>	DIV.D F1,F2,F0	Y	ROB3
				ROB2
				ROB1

Registers	
F3	
F2	<val2>
F1	
F0	<val1>

What about memory hazards???

Memory Disambiguation

- Example:

```
          SD R5,0(R2)
       LD R6,0(R3)
```

- If LD wants to read from a different location than the SD wants to write to, it is safe for the LD to read memory *before* the SD writes memory.

- Is this useful? Yes! Turns out to be very useful in out-of-order processors (e.g., Core i7 tries to do this).

- When can LD read memory? Naïve answer is that we are not allowed to start load until we know that address 0(R2) ≠ 0(R3). More details next slide.

- More sophisticated answer: “Predict” whether or not they are dependent and use reorder buffer to fixup if we are wrong.

Hardware Support for Memory Disambiguation

- Add “store queue” to keep track of all outstanding stores to memory, in program order.
  - Keep track of address (when becomes available) and value (when becomes available)
  - FIFO ordering: retire stores from this buffer in program order
- When issuing a load, record current head of store queue (know which stores are ahead of you).
- When have address for load, check store queue:
  - If *any* store prior to load is waiting for its address, stall load.
  - If load address matches earlier store address (associative lookup), then we have a *memory-induced RAW hazard*:
    - store value available ⇒ return value
    - store value not available ⇒ return ROB number of source
  - Otherwise, send out request to memory
- Stores modify memory during commit, which is “in order”, and this prevents WAR/WAW hazards through memory.

Limits to Performance

Pipeline CPI = Ideal CPI
+ Structural stalls
+ Data hazard stalls
+ Control hazard stalls

Have looked at how to remove/reduce stalls: forwarding, out-of-order execution, branch prediction, speculative execution.

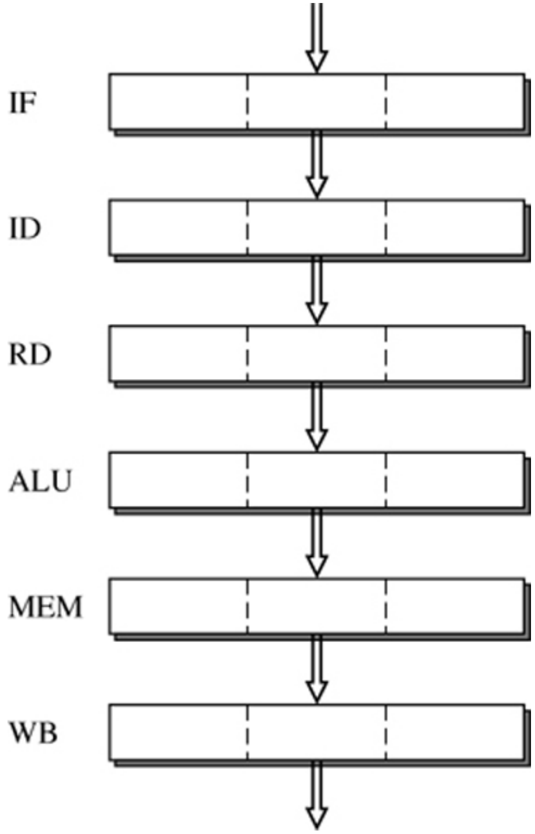
Can we reduce "Ideal CPI"?

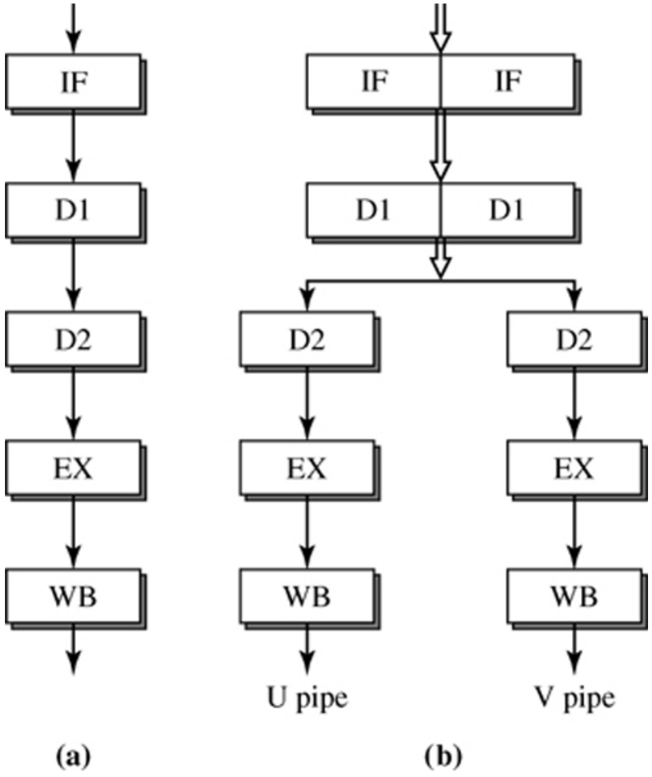
- Ideal CPI limited by rate at which instructions issued: 1 per cycle.
- Modern microprocessors can achieve CPI < 1 by issuing more than one instruction per cycle.
- We will start to speak about average *instructions per cycle (IPC)*
- IPC = (1/CPI)

Multiple Instruction Issue

- We will just call it “multiple issue”
- Two types of multiple issue processor
  - Superscalar
    - Dynamic issue - varying number of instructions per cycle.
    - Statically or dynamically scheduled execution order
      - i.e., in order, out of order w/ or w/o speculative execution
  - VLIW (very long instruction word)
    - Static issue - fixed number of instructions per cycle.
    - Statically scheduled execution

Inorder Superscalar Pipeline





3-wide Superscalar

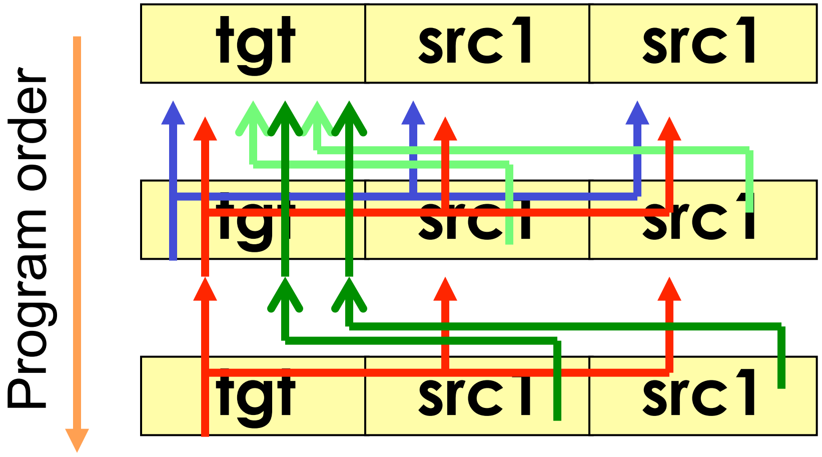
486

Pentium

(2-wide superscalar)

Complexity?

- Issue: Check each instruction for hazards with earlier instructions in program order issued this cycle, as well as against all earlier instructions still in execution.



15 register identifier comparisons to avoid RAW, WAR, and WAW hazards in a 3-wide <u>in-order</u> superscalar versus “only” 5 for 2-wide issue. Number of comparisons grows very quickly with issue width.

2-Issue In-order Superscalar

	Clock Number							
	1	2	3	4	5	6	7	8
integer instruction	IF	ID	EX	MEM	WB			
FP instruction	IF	ID	EX	EX	EX	WB		
integer instruction		IF	ID	EX	MEM	WB		
FP instruction		IF	ID	EX	EX	EX	WB	
integer instruction			IF	ID	EX	MEM	WB	
FP instruction			IF	ID	EX	EX	EX	WB

- Issue restriction: At most one Integer and one Floating Point operation can issue per cycle (this organization was used in the HP 7100 processor).
- Challenge maintaining peak throughput: Next 3 instructions cannot use result of load.

Very Long Instruction Word (VLIW)

- In a superscalar processor, we need to check for dependencies between instructions before they can be issued in parallel.
- Each time we fetch and decode the same group of instructions we will find the same dependencies (through registers).
- Why not let the compiler search for these dependencies and groups of instructions that are known to be independent? This is how a VLIW processor works
- Drawbacks:
  - Need to explicitly encode "no ops" for units that do not have an operation.
  - Hard to predict which memory access instructions miss in cache
  - Need to use profile feedback to find "hot traces" in program to extract instruction level parallelism
  - Memory dependencies hard to determine in compiler

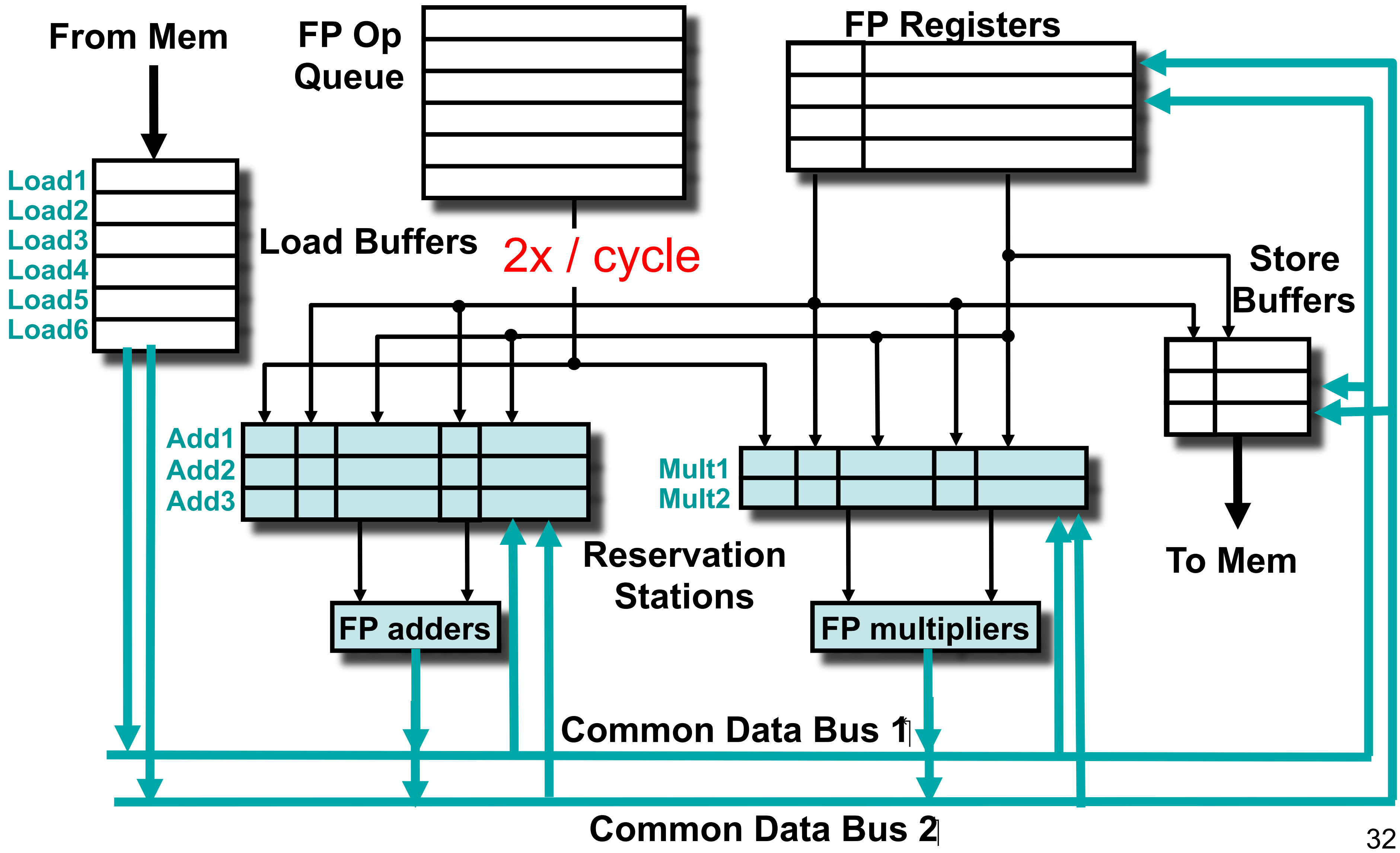
Inst N:	ADD.D F1,F1,F4	**NOP**	L.D F7, 0(R1)	DSUBI R2,R2,#1	**NOP**
Inst N+1:	ADD.D F1,F1,F7	MUL.D F4,F5,F6	**NOP**	DSUBI R2,R2,#1	BEQZ R2, Loop

Superscalar w/ Out of Order Execution

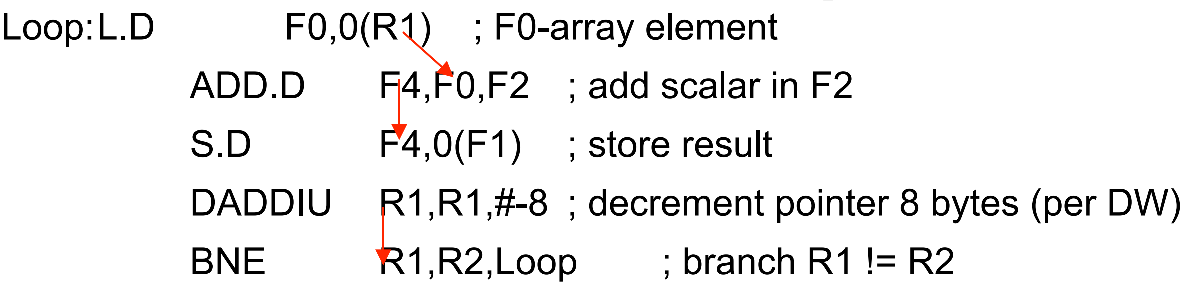
- Use extension of Tomasulo's algorithm.
- Benefits:
  - Overcome data hazards
  - Overcome issue restrictions

- Key challenge: for each instruction in issue packet, in program order
  - Assign reservation station and update pipeline control tables
- Two approaches to overcome this challenge:
  - Run this step at higher clock frequency (i.e., do in half a clock cycle for 2-wide superscalar)
  - Add logic to perform operation on multiple (e.g., 2) instructions at once.

Tomasulo Organization (with two-wide issue)



Example



```
Loop:L.D        F0,0(R1)    ; F0-array element
        ADD.D   F4,F0,F2    ; add scalar in F2
        S.D     F4,0(F1)    ; store result
        DADDIU  R1,R1,#-8  ; decrement pointer 8 bytes (per DW)
        BNE     R1,R2,Loop        ; branch R1 != R2
```

Assumptions:

(1) 2-wide superscalar issue

(2) Infinite number of reservation stations

(3) Perfect branch prediction (let’s do three iterations of loop); separate branch unit.

(4) Issue instruction from target of taken branch 1 cycle after branch (due to fetch restrictions)

(5) One Integer Unit (handles load/store effective address calculation)

(6) Separate FP Function Unit for each type of FP operation

(7) Issue and Write Results take one cycle

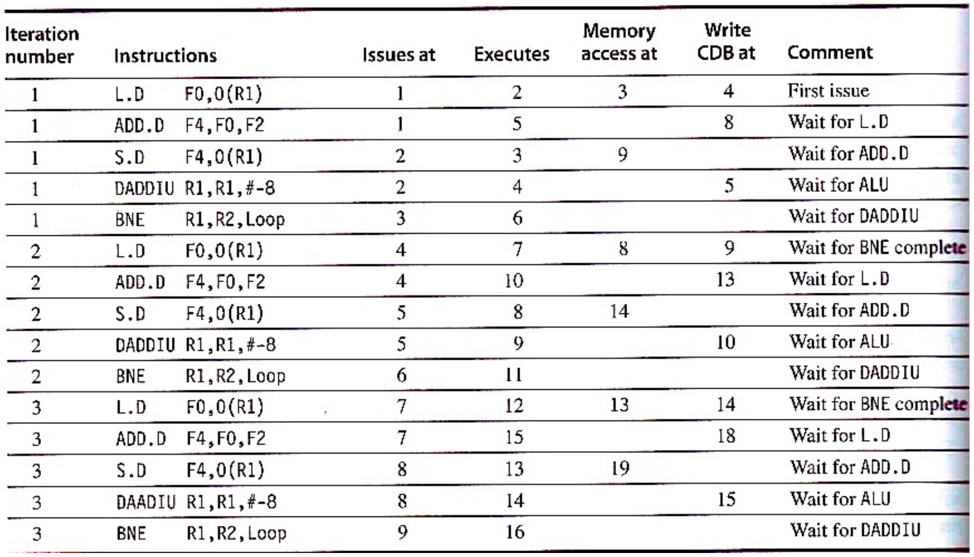
(8) Latencies: One cycle for integer ALU; two cycles for loads; three cycles for FP adds

(9) Two CDBs

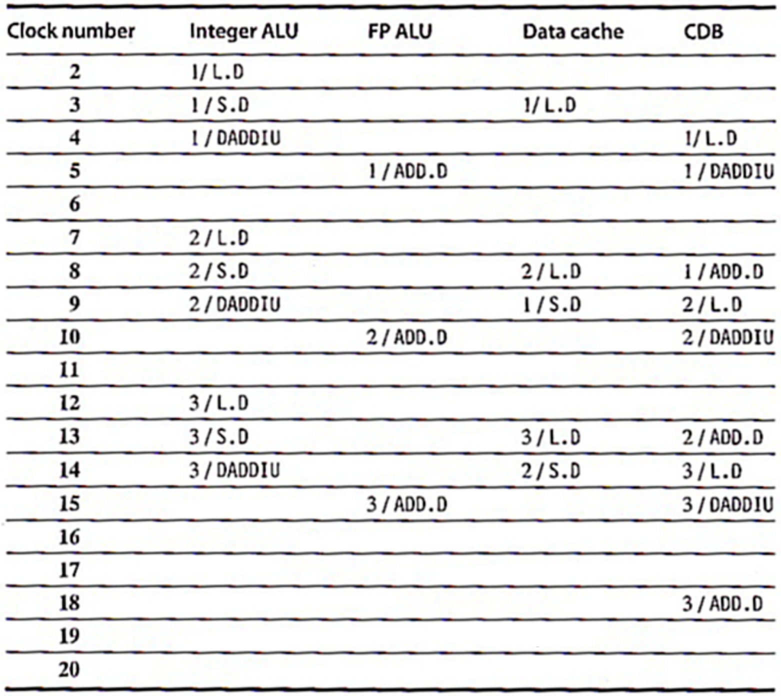
(10) Load/Store effective address calculation “decoupled” from memory access (e.g., compute effective address as soon as possible, even if data to store to memory is not available)

(11) NO Reorder Buffer

Example: 2-issue w/ Tomasulo



Iteration number	Instructions	Issues at	Executes	Memory access at	Write CDB at	Comment
1	L.D F0,0(R1)	1	2	3	4	First issue
1	ADD.D F4,F0,F2	1	5		8	Wait for L.D
1	S.D F4,0(R1)	2	3	9		Wait for ADD.D
1	DADDIU R1,R1,#-8	2	4		5	Wait for ALU
1	BNE R1,R2,Loop	3	6			Wait for DADDIU
2	L.D F0,0(R1)	4	7	8	9	Wait for BNE complete
2	ADD.D F4,F0,F2	4	10		13	Wait for L.D
2	S.D F4,0(R1)	5	8	14		Wait for ADD.D
2	DADDIU R1,R1,#-8	5	9		10	Wait for ALU
2	BNE R1,R2,Loop	6	11			Wait for DADDIU
3	L.D F0,0(R1)	7	12	13	14	Wait for BNE complete
3	ADD.D F4,F0,F2	7	15		18	Wait for L.D
3	S.D F4,0(R1)	8	13	19		Wait for ADD.D
3	DAADIU R1,R1,#-8	8	14		15	Wait for ALU
3	BNE R1,R2,Loop	9	16			Wait for DADDIU



Clock number	Integer ALU	FP ALU	Data cache	CDB
2	1/L.D			
3	1/S.D		1/L.D	
4	1/DADDIU			1/L.D
5		1/ADD.D		1/DADDIU
6				
7	2/L.D			
8	2/S.D		2/L.D	1/ADD.D
9	2/DADDIU		1/S.D	2/L.D
10		2/ADD.D		2/DADDIU
11				
12	3/L.D			
13	3/S.D		3/L.D	2/ADD.D
14	3/DADDIU		2/S.D	3/L.D
15		3/ADD.D		3/DADDIU
16				
17				
18				3/ADD.D
19				
20				

1. Mark relevant dependencies (true deps)
2. Each clock cycle: Start from oldest instruction, working toward younger instructions and attempt to apply scheduling algorithm + assumptions

- Fetch Issue 15 instructions/9 cycles = 1.67 IPC
- Execution Completion rate = 15 inst./16 cycles = 0.94 IPC (much less than 2)

Modified Example (extra Int ALU)

```
Loop:   L.D     F0,0(R1)      ; F0-array element
        ADD.D   F4,F0,F2      ; add scalar in F2
        S.D     F4,0(F1)      ; store result
        DADDIU  R1,R1,#-8     ; decrement pointer 8 bytes (per DW)
        BNE     R1,R2,Loop    ; branch R1 != R2
```

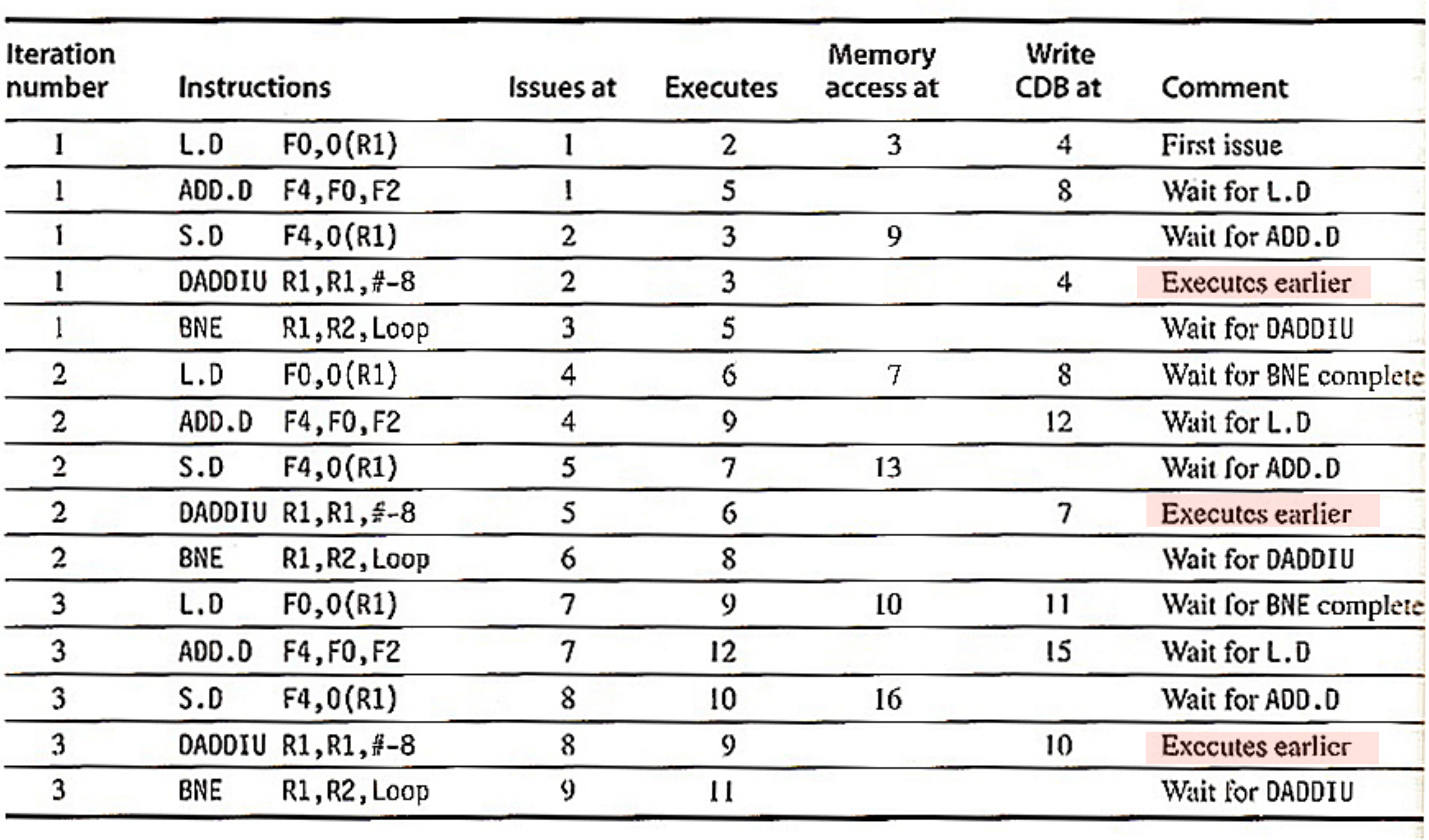
Changed Assumptions:

…

(5) **Address Adder & Integer Unit**

…

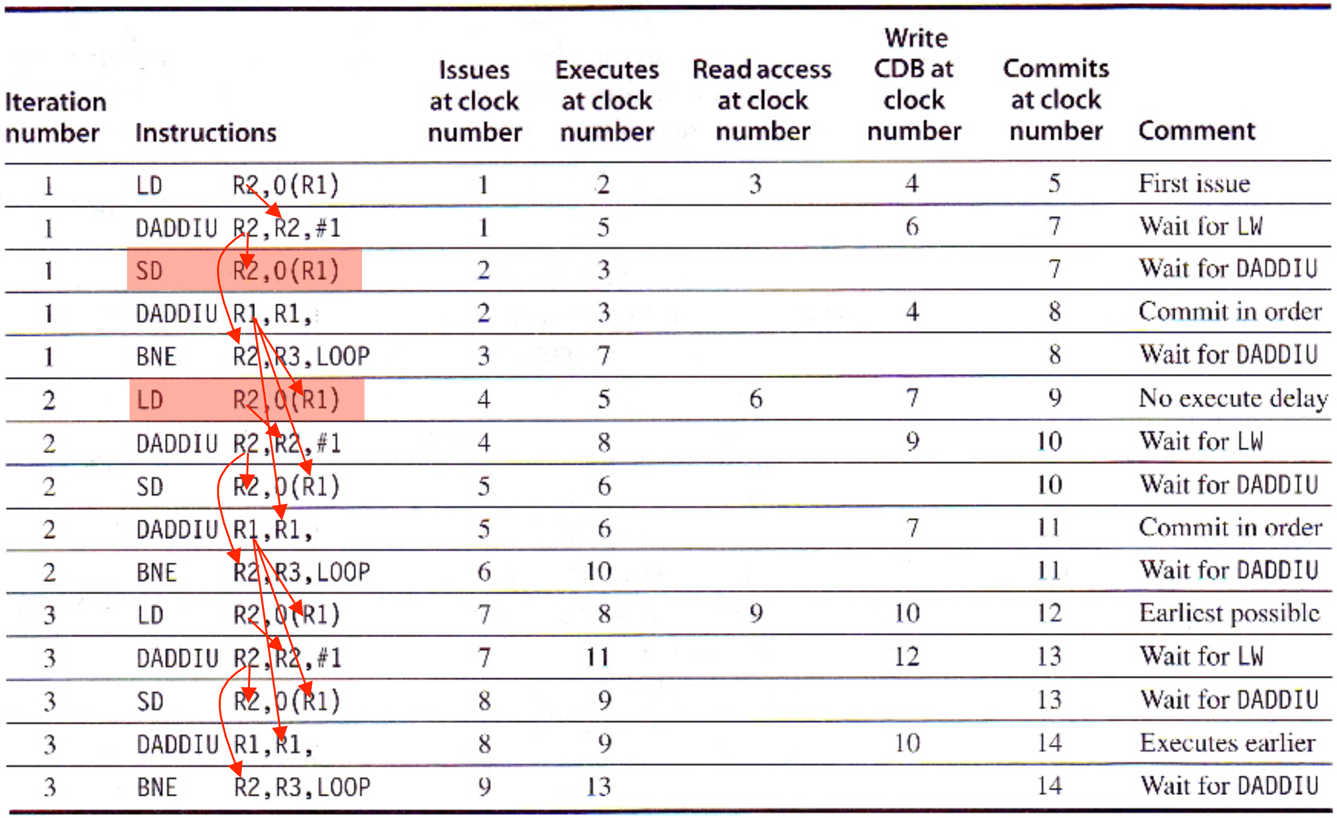
Modified Example, cont'd...



Iteration number	Instructions		Issues at	Executes	Memory access at	Write CDB at	Comment
1	L.D	F0,0(R1)	1	2	3	4	First issue
1	ADD.D	F4,F0,F2	1	5		8	Wait for L.D
1	S.D	F4,0(R1)	2	3	9		Wait for ADD.D
1	DADDIU	R1,R1,#-8	2	3		4	Executes earlier
1	BNE	R1,R2,Loop	3	5			Wait for DADDIU
2	L.D	F0,0(R1)	4	6	7	8	Wait for BNE complete
2	ADD.D	F4,F0,F2	4	9		12	Wait for L.D
2	S.D	F4,0(R1)	5	7	13		Wait for ADD.D
2	DADDIU	R1,R1,#-8	5	6		7	Executes earlier
2	BNE	R1,R2,Loop	6	8			Wait for DADDIU
3	L.D	F0,0(R1)	7	9	10	11	Wait for BNE complete
3	ADD.D	F4,F0,F2	7	12		15	Wait for L.D
3	S.D	F4,0(R1)	8	10	16		Wait for ADD.D
3	DADDIU	R1,R1,#-8	8	9		10	Executes earlier
3	BNE	R1,R2,Loop	9	11			Wait for DADDIU

- Execution completion rate: 3 cycles less (3/S.D vs 1/L.D: 16-4+1 = 13 vs. 16 cycles before) IPC = 15/13 = 1.15 (vs. 0.94)
- **Performance limited by waiting for branch to resolve.**

Example of Multiple Issue, Tomasulo & Reorder Buffer



Iteration number	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LW
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LW
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LW
3	SD R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,	8	9		10	14	Executes earlier
3	BNE R2,R3,LOOP	9	13			14	Wait for DADDIU

1. Mark true dependencies
2. Each clock cycle: Start from oldest instruction, working toward younger instructions and attempt to apply <whichever OoO> algorithm rules

- Commit rate: 15 instructions in 14-5+1 = 10 cycle
- Speculative execution helped use commit 1.5 instructions per cycle

Summary of Slide Set 9

In this slide set, we learned about the following:

- Support for precise exceptions and speculative execution with a single mechanism: The reorder buffer.
- Multiple issue lowers the “ideal CPI” to below 1.
- Two forms of multiple issue: Superscalar, VLIW

We now know how a single processor “core” works.