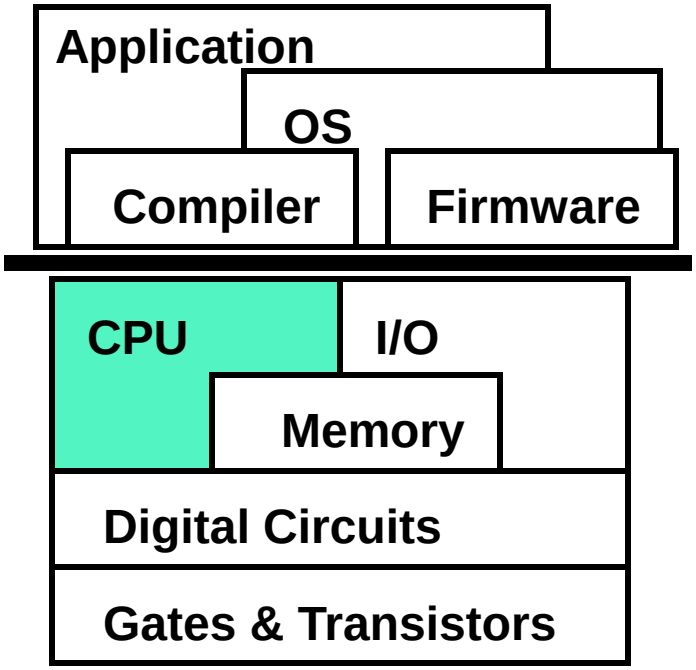
CS/ECE 752: Advanced Computer Architecture I

Professor Matthew D. Sinclair

OOO + Speculation

Slide History/Attribution Diagram:

This Unit: Dynamic Scheduling II



- Previously: dynamic scheduling
  - Insn buffer + scheduling algorithms
  - Scoreboard: no register renaming
  - Tomasulo: register renaming
- Now: add speculation, precise state
  - Re-order buffer
  - PentiumPro vs. MIPS R10000
- Also: dynamic load scheduling
  - Out-of-order memory operations

Superscalar + Out-of-Order + Speculation

- Three great tastes that go great together
  - IPC ≥ 1?
    - Go superscalar
  - Superscalar increases RAW hazards?
    - Go out-of-order (OoO)
  - RAW hazards still a problem?
    - Build a larger window
  - Branches a problem for filling large window?
    - Add control speculation (e.g., TAGE)

Speculation and Precise Interrupts

- Why are we discussing these together?
  - Sequential (vN) semantics for interrupts
    - All insns before interrupt should be complete
    - All insns after interrupt should look as if never started (abort)
    - **Basically want same thing for mis-predicted branch**
  - What makes precise interrupts difficult?
    - OoO completion → must undo post-interrupt writebacks
    - Same thing for branches
    - In-order → branches complete before younger insns writeback
    - OoO → not necessarily

- Precise interrupts, mis-speculation recovery: same problem
- **Same problem → same solution**

Precise State

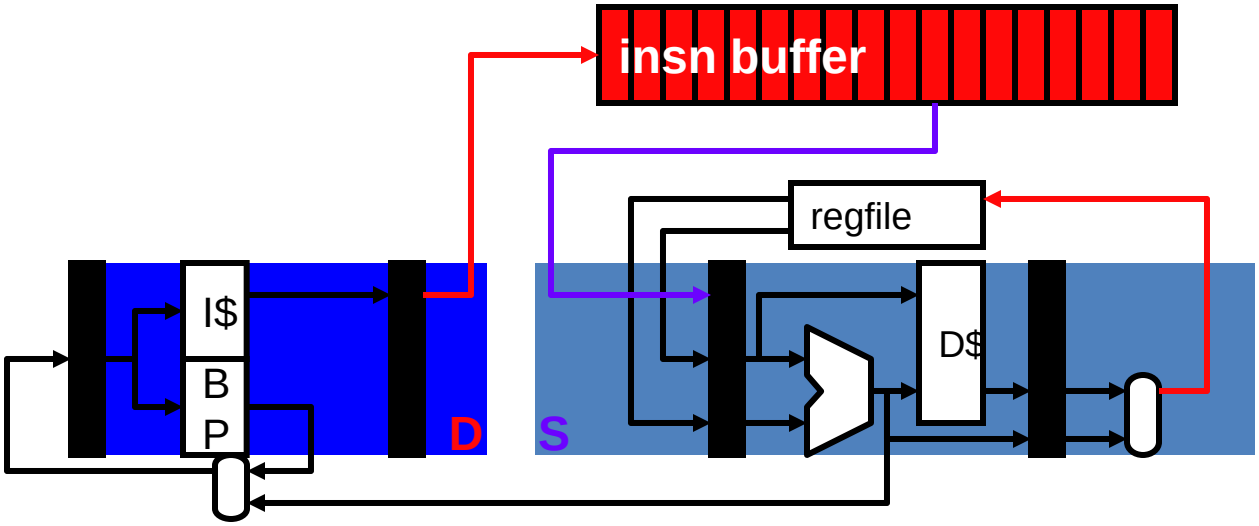
- Speculative execution requires
  - (Ability to) abort & restart at every branch
  - Abort & restart at every load useful for load speculation (later)
    - And for shared memory multiprocessing (much later)
- Precise synchronous (program-internal) interrupts require
  - Abort & restart at every load, store, ??
- Precise asynchronous (external) interrupts require
  - Abort & restart at every ??

- Bite the bullet
  - Implement abort & restart at every insn
  - Called **"precise state"**

Precise State Options

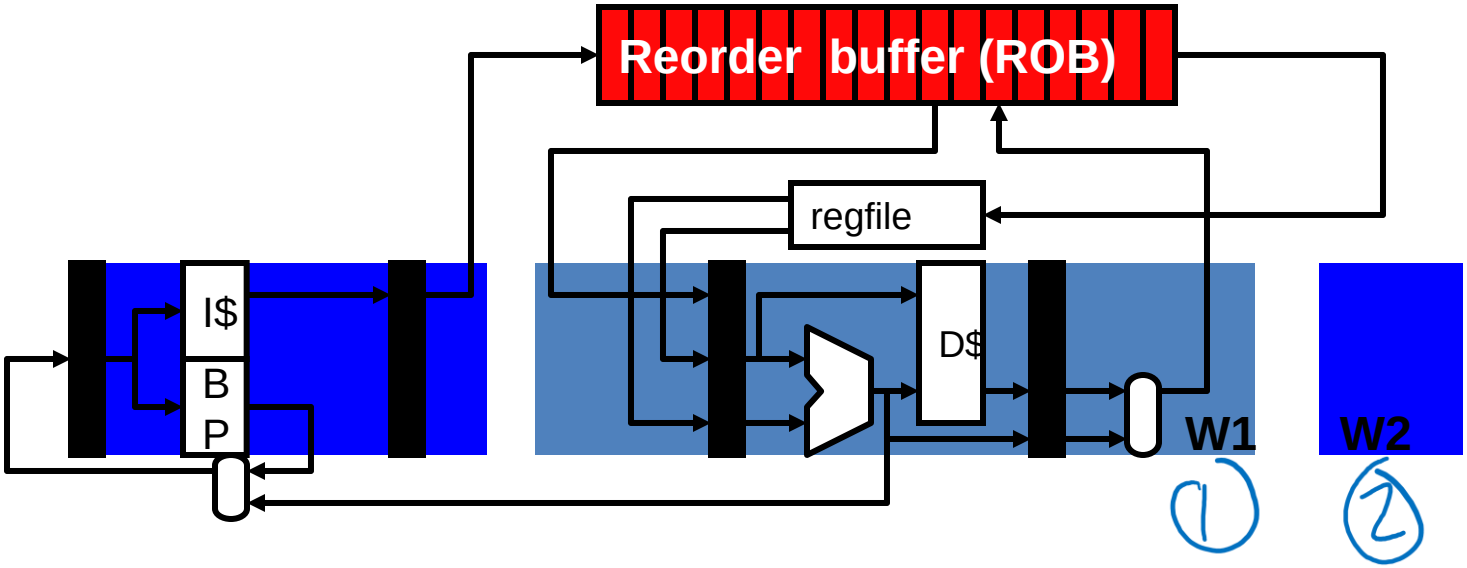
- Imprecise state: ignore the problem!
  - Makes page faults (any restartable exceptions) difficult
  - Makes speculative execution almost impossible
  - Compromise: Alpha implemented precise state only for integer ops
- Force in-order completion (W): stall pipe if necessary
  - Slow
- Precise state in software: trap to recovery routine
  - Implementation dependent
  - Trap on every mis-predicted branch (you must be joking)
- Precise state in hardware
  + Everything is better in hardware (except policy)

The Problem with Precise State



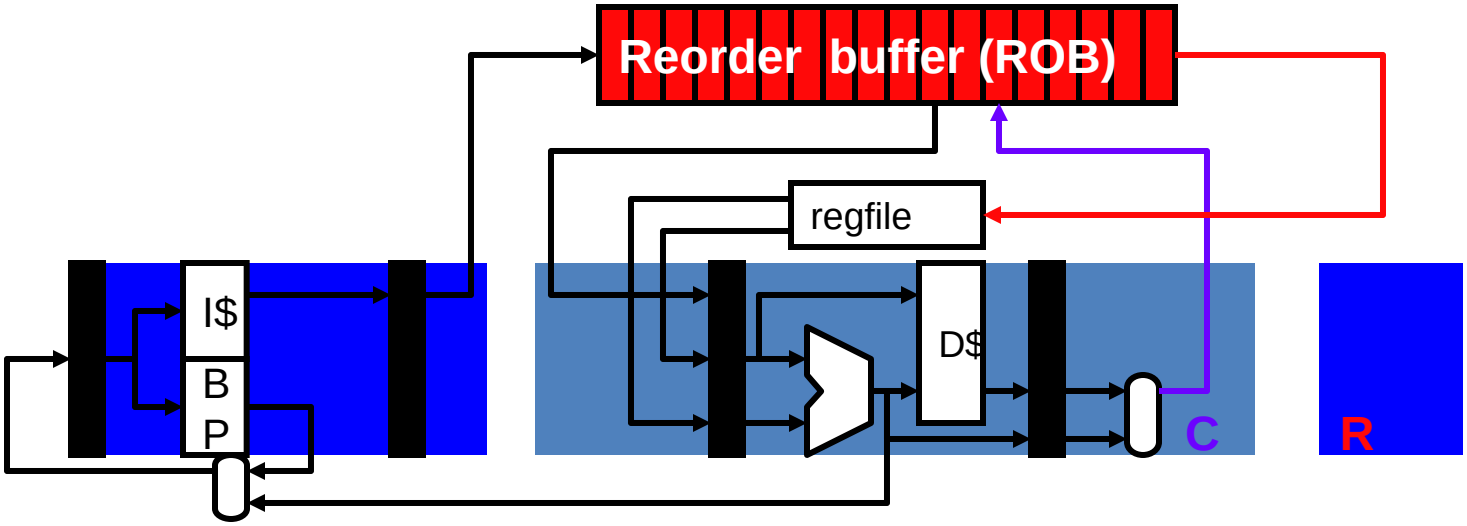
- Problem: **writeback** combines two separate functions
  - Forwards values to younger insns: OK for this to be out-of-order
  - Write values to registers: would like this to be in-order
- Similar problem (decode) for OoO execution: solution?
  - Split decode (D) → in-order dispatch (D) + out-of-order issue (S)
  - Separate using insn buffer: scoreboard or reservation station

Re-Order Buffer (ROB)



- **Insn buffer → re-order buffer (ROB)**
  - Buffers completed results en route to register file
  - May be combined with RS or separate
    - Combined in picture: register-update unit RUU (Sohi's method)
    - Separate (more common today): P6-style
- Split writeback (W) into two stages
  - Why is there no latch between W1 and W2?

Complete and Retire



- **Complete (C)**: first part of writeback (W)
  - Completed insns write results into ROB
  - \+ Out-of-order: **wait** doesn't back-propagate to younger insns
- **Retire (R)**: aka commit, graduate
  - ROB writes results to register file
  - In order: **stall** back-propagates to younger insns

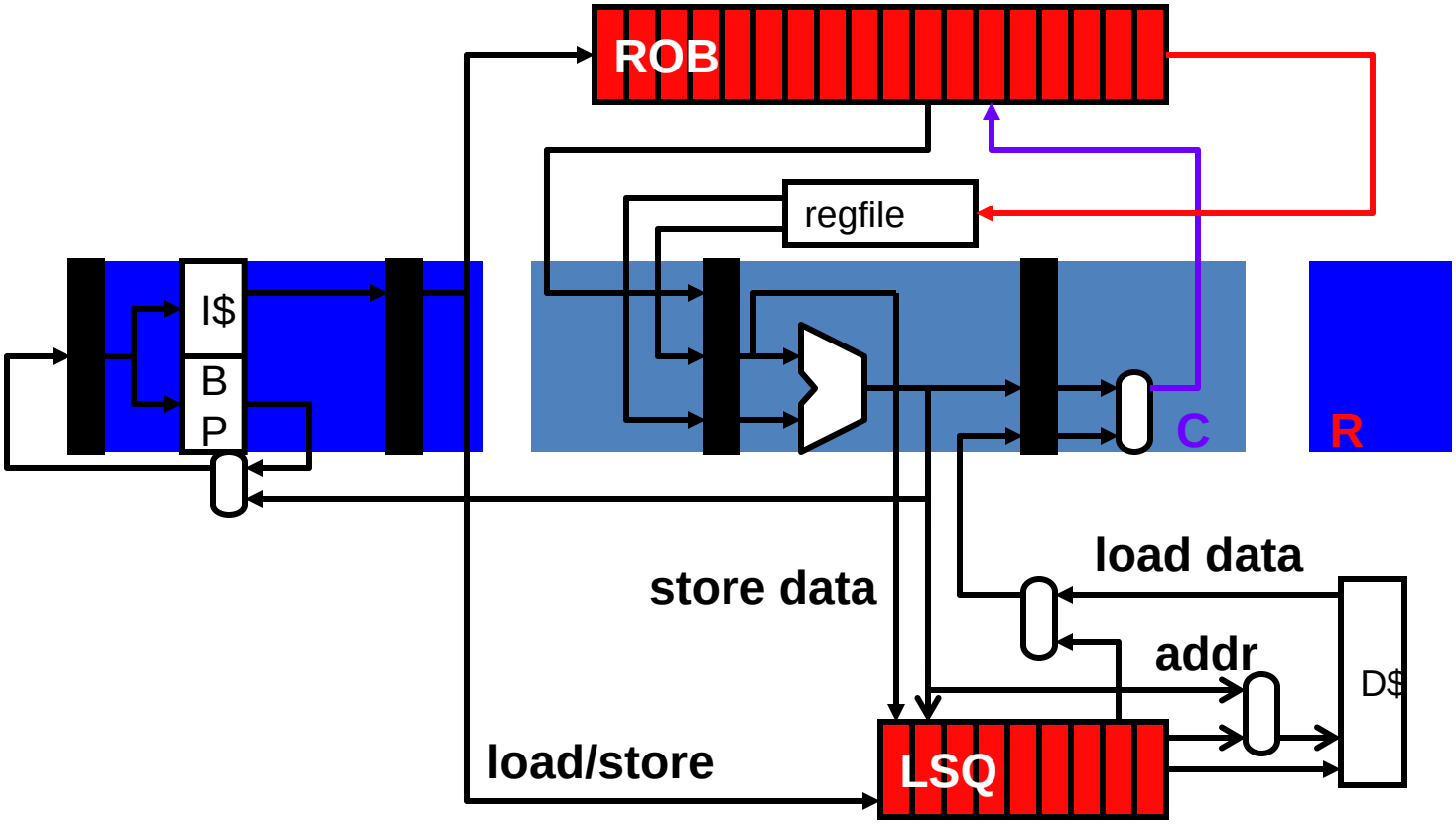
Load/Store Queue (LSQ)

- ROB makes register writes in-order, but what about stores?

- As usual, i.e., store to D$ in X stage?
  - Not even close, imprecise memory worse than imprecise registers

- **Load/store queue (LSQ)**
  - Completed stores write to LSQ
  - When store retires, head of LSQ written to D$
  - When loads execute, access LSQ and D$ in parallel
    - Forward from LSQ if older store with matching address
  - More modern design: loads and stores in separate queues
  - More on this later

ROB + LSQ



- Modulo gross simplifications, this picture is almost realistic!

P6

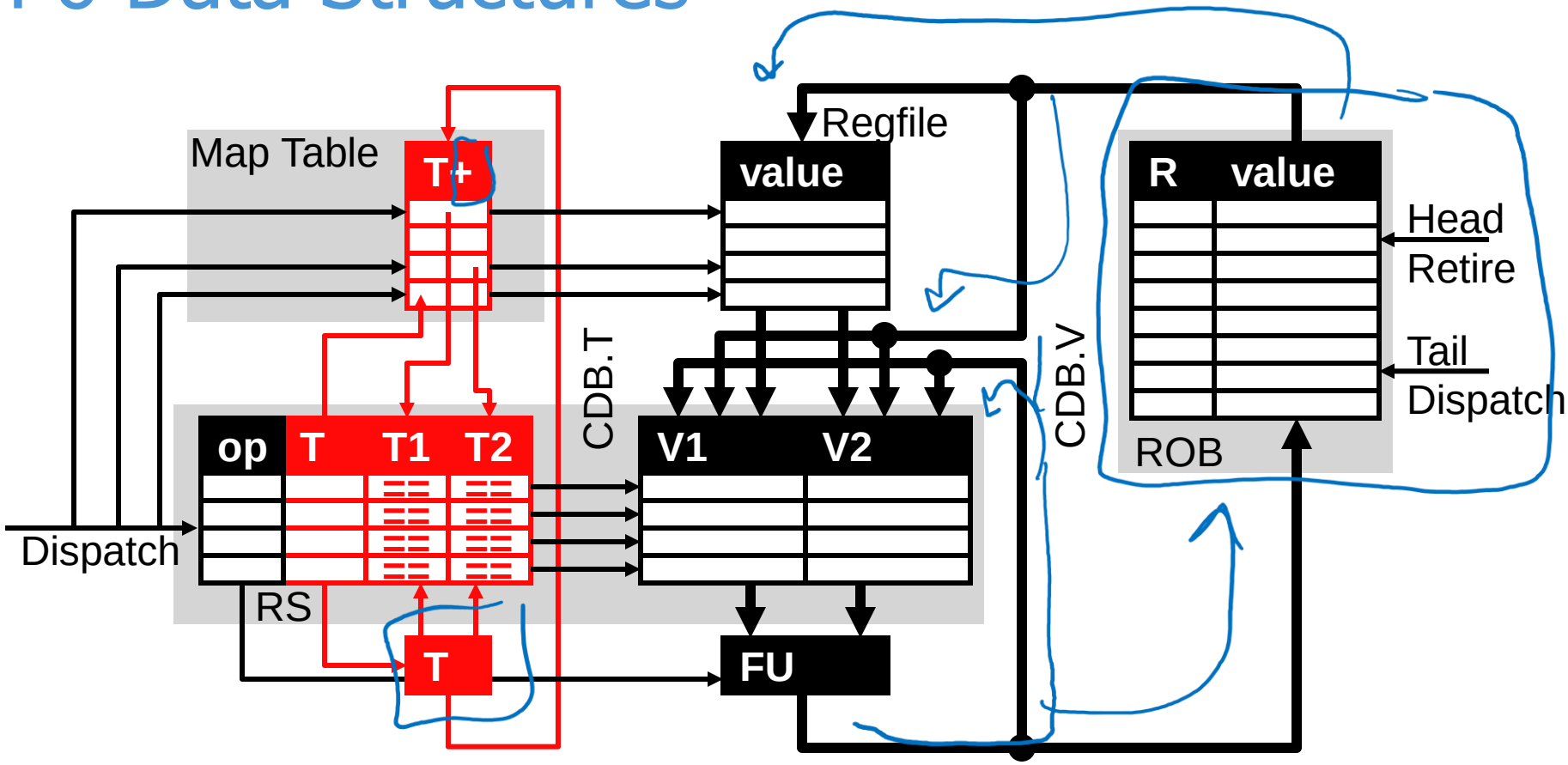
- P6: Start with Tomasulo's algorithm… add ROB
  - Separate ROB and RS

- Simple-P6
  - Our old RS organization: 1 ALU, 1 load, 1 store, 2 3-cycle FP

P6 Data Structures

- Reservation Stations are same as before
- ROB
  - **head**, **tail**: pointers maintain sequential order
  - **R**: insn output register, **V**: insn output value
- Tags are different
  - Tomasulo: RS# → P6: ROB#
- Map Table is different
  - **T+**: tag + "ready-in-ROB" bit
  - T==0 → Value is ready in regfile
  - T!=0 → Value is not ready
  - T!=0+ → Value is ready in the ROB

P6 Data Structures



- Insn fields and status bits
- Tags
- Values

P6 Data Structures

value

ROB							
ht	#	Insn	R	V	S	X	C
	1	`ldf X(r1),f1`					
	2	`mulf f0,f1,f2`					
	3	`stf f2,Z(r1)`					
	4	`addi r1,4,r1`					
	5	`ldf X(r1),f1`					
	6	`mulf f0,f1,f2`					
	7	`stf f2,Z(r1)`					

Map Table	
Reg	T+
f0	
f1	
f2	
r1	

CDB	
T	V

Reservation Stations								
#	FU	busy	op	T	T1	T2	V1	V2
1	ALU	no						
2	LD	no						
3	ST	no						
4	FP1	no						
5	FP2	no						

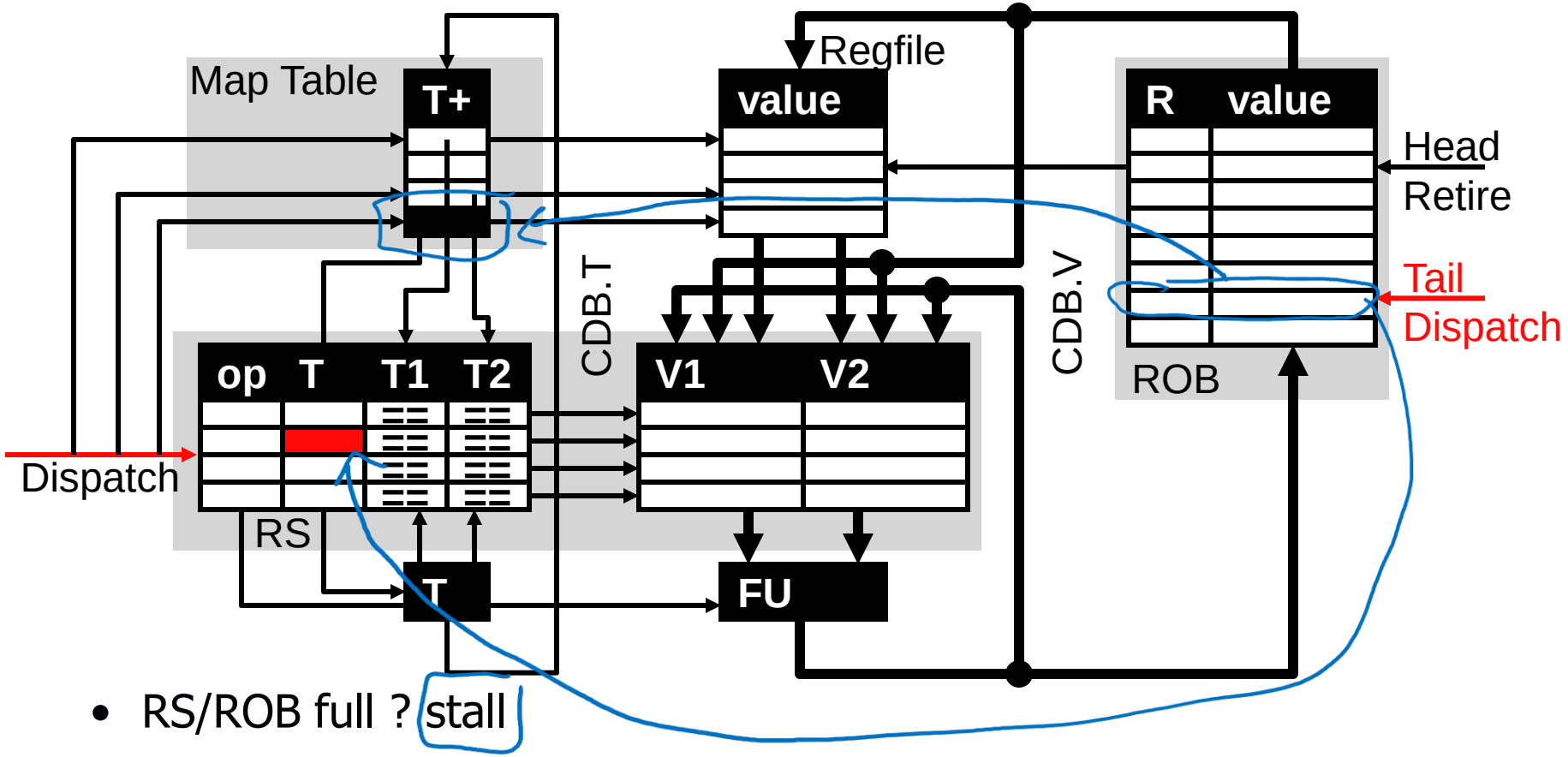
P6 Pipeline

- New pipeline structure: F, **D**, S, **X**, **C**, **R**
  - **D (dispatch)**
    - Structural hazard (ROB/LSQ/RS) ? **Stall**
    - Allocate ROB/LSQ/RS
    - Set RS tag to ROB#
    - Set Map Table entry to ROB# and clear "ready-in-ROB" bit
    - Read ready registers into RS (from either ROB or Regfile)
  - **X (execute)**
    - Free RS entry
    - Use to be at W, can be earlier because RS# are not tags

P6 Pipeline

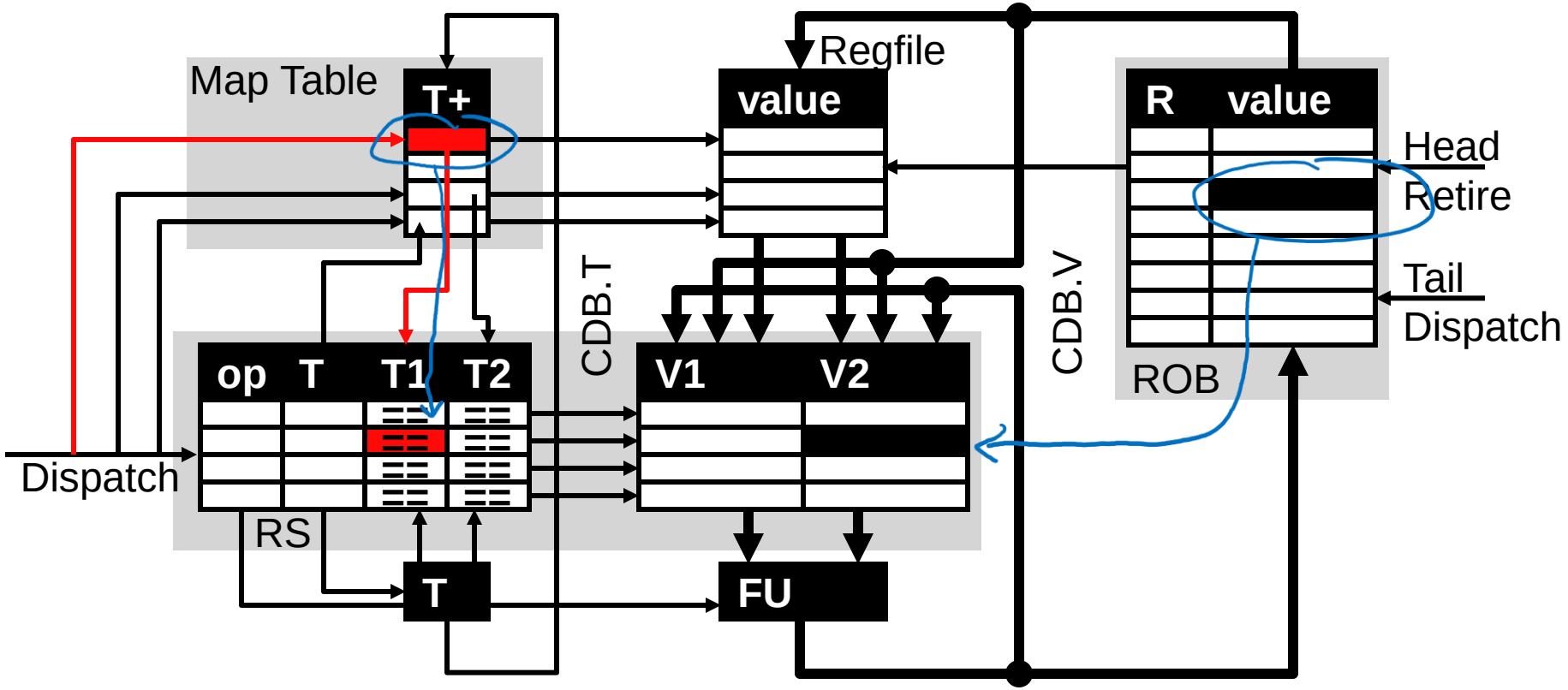
- **C (complete)**
  - Structural hazard (CDB)? **wait**
  - Write value into ROB entry indicated by RS tag
  - Mark ROB entry as complete
  - If not overwritten, mark Map Table entry "ready-in-ROB" bit (+)
- **R (retire)**
  - Insn at ROB head not complete ? **stall**
  - Handle any exceptions
  - Write ROB head value to register file
  - If store, write LSQ head to D$
  - Free ROB/LSQ entries

P6 Dispatch (D): Part I



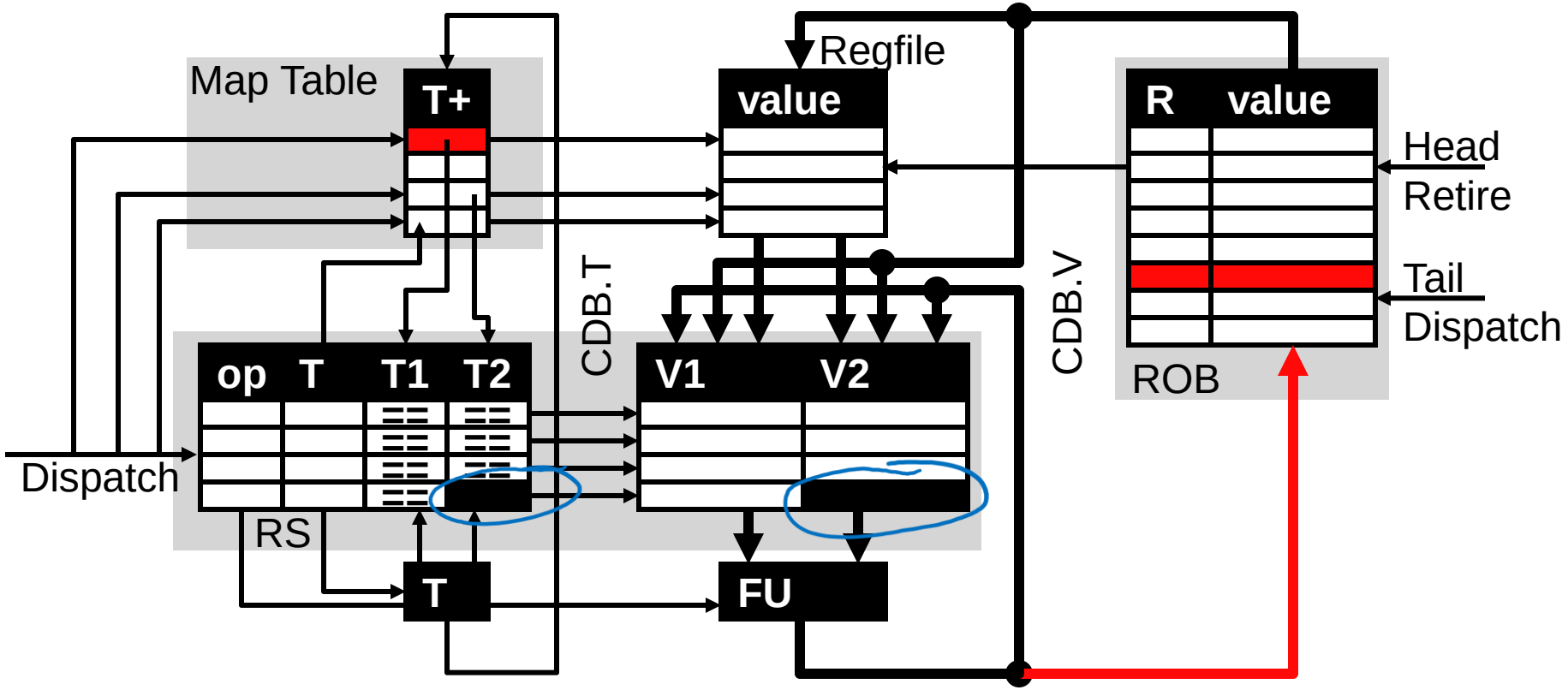
- RS/ROB full ? stall
- Allocate RS/ROB entries, assign ROB# to RS output tag
- Set output register Map Table entry to ROB#, clear "ready-in-ROB"

P6 Dispatch (D): Part II



- Read tags for register inputs from Map Table
  - Tag==0 → copy value from Regfile (not shown)
  - Tag!=0 → copy Map Table tag to RS
  - Tag!=0+ → copy value from ROB

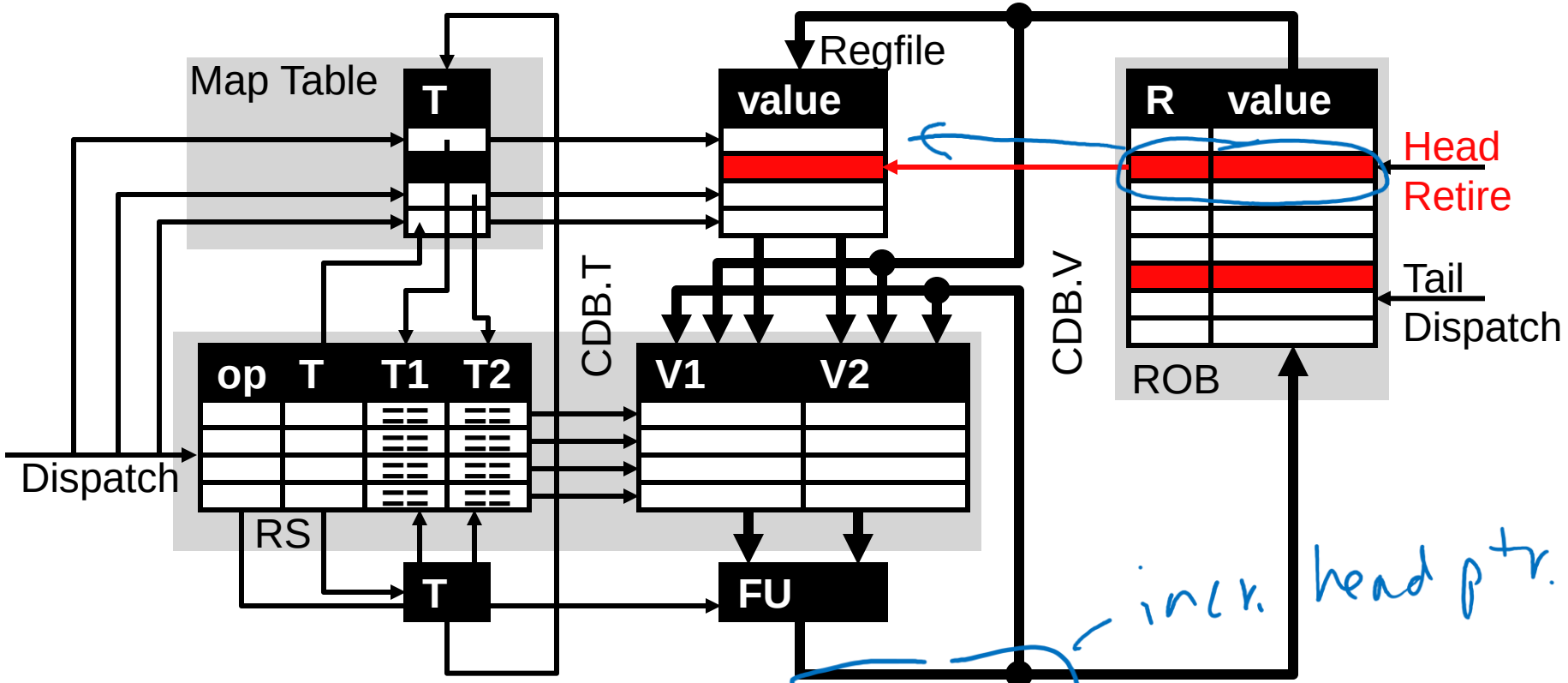
P6 Complete (C)



- Structural hazard (CDB) ? Stall : broadcast <value,tag> on CDB
- Write result into ROB, if still valid set MapTable "ready-in-ROB" bit
- Match tags, write CDB.V into RS slots of dependent insns

most up to date mapping

P6 Retire (R)



- ROB head not complete ? stall : free ROB entry
- Write ROB head result to Regfile
- If still valid, clear Map Table entry → now read from RF for this reg.

P6: Cycle

ROB							
ht	#	Insn	R	V	S	X	C
	1	`ldf X(r1),f1`					
	2	`mulf f0,f1,f2`					
	3	`stf f2,Z(r1)`					
	4	`addi r1,4,r1`					
	5	`ldf X(r1),f1`					
	6	`mulf f0,f1,f2`					
	7	`stf f2,Z(r1)`					

Map Table	
Reg	T+
`f0`	
`f1`	
`f2`	
`r1`	

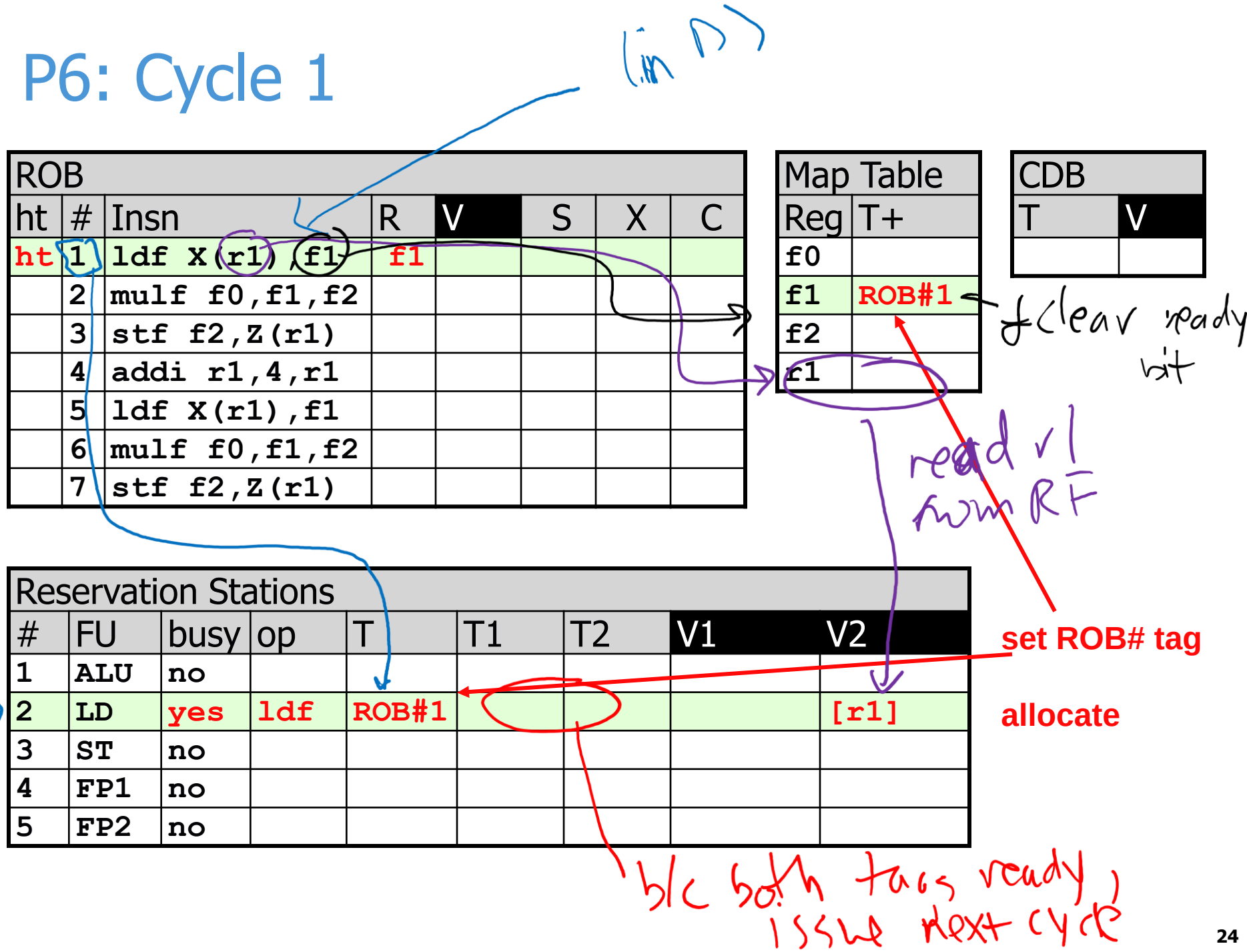
CDB	
T	V

Reservation Stations								
#	FU	busy	op	T	T1	T2	V1	V2
1	`ALU`	`no`						
2	`LD`							
3	`ST`	`no`						
4	`FP1`	`no`						
5	`FP2`	`no`						

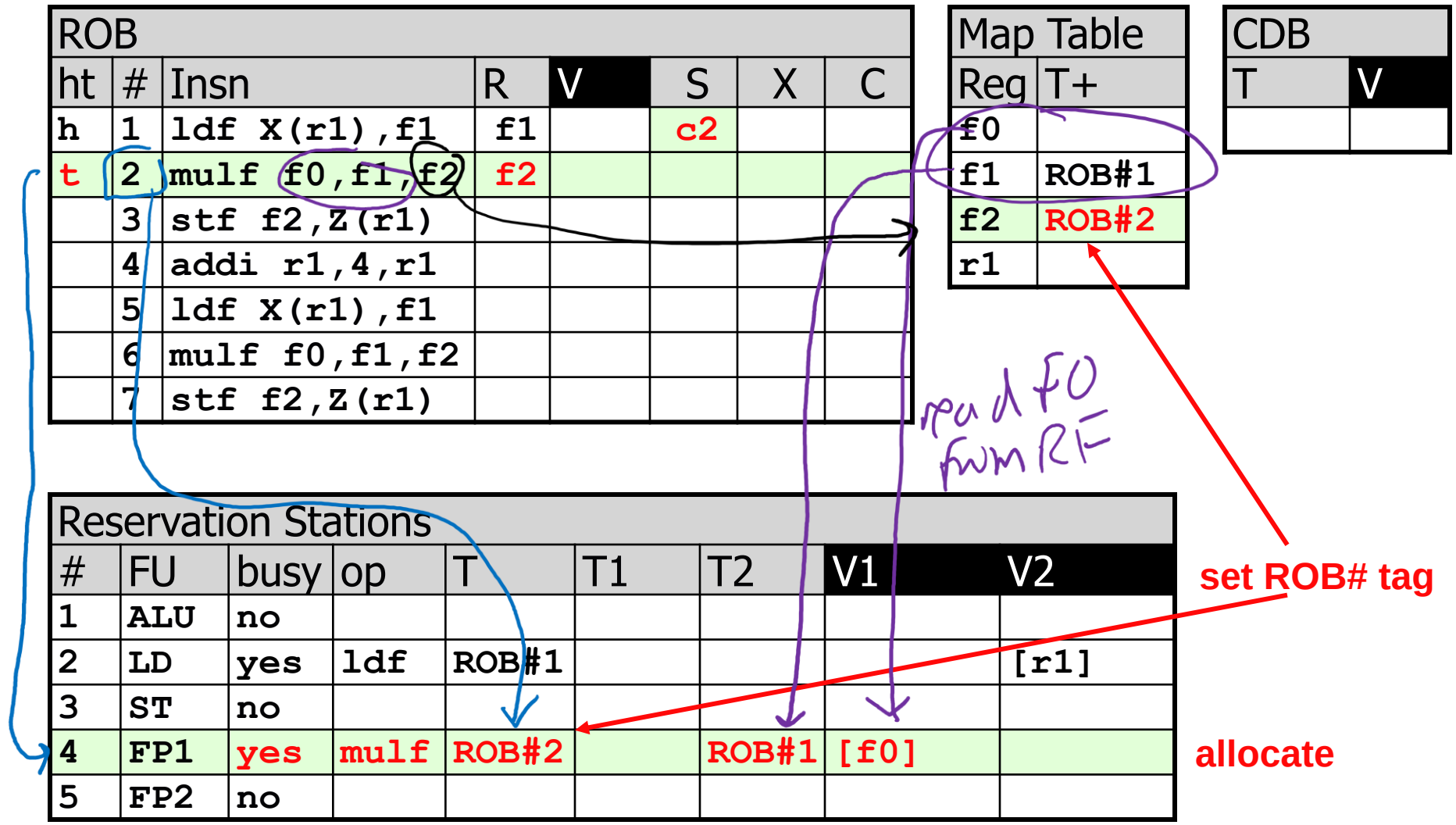
Per Stage Info P6

- Dispatch Part I
  - RS/ROB full ? stall
  - Allocate RS/ROB entries, assign ROB# to RS output tag
  - Set output register Map Table entry to ROB#, clear "ready-in-ROB"
- Dispatch Part II
  - Read tags for register inputs from Map Table
    - Tag==0 → copy value from Regfile (not shown)
    - Tag!=0 → copy Map Table tag to RS
    - Tag!=0+ → copy value from ROB
- Complete
  - Structural hazard (CDB) ? Stall : broadcast <value,tag> on CDB
  - Write result into ROB, if still valid set MapTable "ready-in-ROB" bit
  - Match tags, write CDB.V into RS slots of dependent insns
- Retire
  - ROB head not complete ? stall : free ROB entry
  - Write ROB head result to Regfile
  - If still valid, clear Map Table entry

P6: Cycle 1


(in D)
ROB
ht | # | Insn | R | V | S | X | C
ht | 1 | ldf X(r1),f1 | f1
2 | mulf f0,f1,f2
3 | stf f2,Z(r1)
4 | addi r1,4,r1
5 | ldf X(r1),f1
6 | mulf f0,f1,f2
7 | stf f2,Z(r1)
Map Table
Reg | T+
f0
f1 | ROB#1
f2
r1
CDB
T | V
clear ready bit
read r1 from RF
Reservation Stations
| FU | busy | op | T | T1 | T2 | V1 | V2
1 | ALU | no
2 | LD | yes | ldf | ROB#1 | [r1]
3 | ST | no
4 | FP1 | no
5 | FP2 | no
set ROB# tag
allocate
b/c both tags ready, issue next cycle

P6: Cycle 2


ROB
ht # Insn R V S X C
h 1 ldf X(r1),f1 f1 c2
t 2 mulf f0,f1,f2 f2
3 stf f2,Z(r1)
4 addi r1,4,r1
5 ldf X(r1),f1
6 mulf f0,f1,f2
7 stf f2,Z(r1)
Map Table
Reg T+
f0
f1 ROB#1
f2 ROB#2
r1
CDB
T V
read f0 from RF
Reservation Stations
FU busy op T T1 T2 V1 V2
1 ALU no
2 LD yes ldf ROB#1 [r1]
3 ST no
4 FP1 yes mulf ROB#2 ROB#1 [f0]
5 FP2 no
set ROB# tag
allocate

P6: Cycle 3

ROB							
ht	#	Insn	R	V	S	X	C
h	1	ldf X(r1),f1	f1		c2	c3	
	2	mulf f0,f1,f2	f2				
t	3	stf f2,Z(r1)					
	4	addi r1,4,r1					
	5	ldf X(r1),f1					
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	
f1	ROB#1
f2	ROB#2
r1	

CDB	
T	V

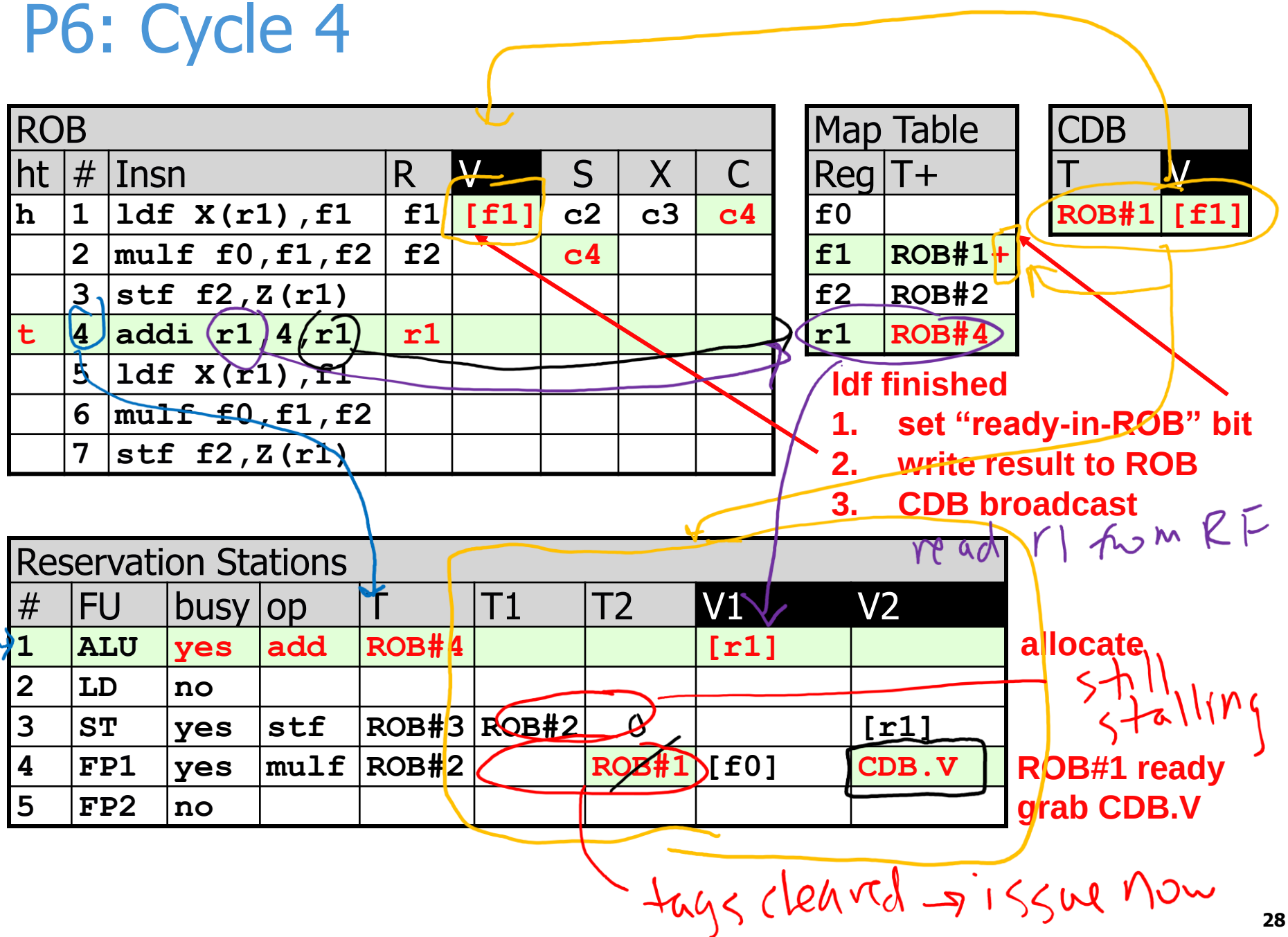
read r1 from RF

Reservation Stations								
#	FU	busy	op	T	T1	T2	V1	V2
1	ALU	no						
2	LD	no						
3	ST	yes	stf	ROB#3	ROB#2			[r1]
4	FP1	yes	mulf	ROB#2		ROB#1	[f0]	
5	FP2	no						

free
allocate

still stalling on ldf

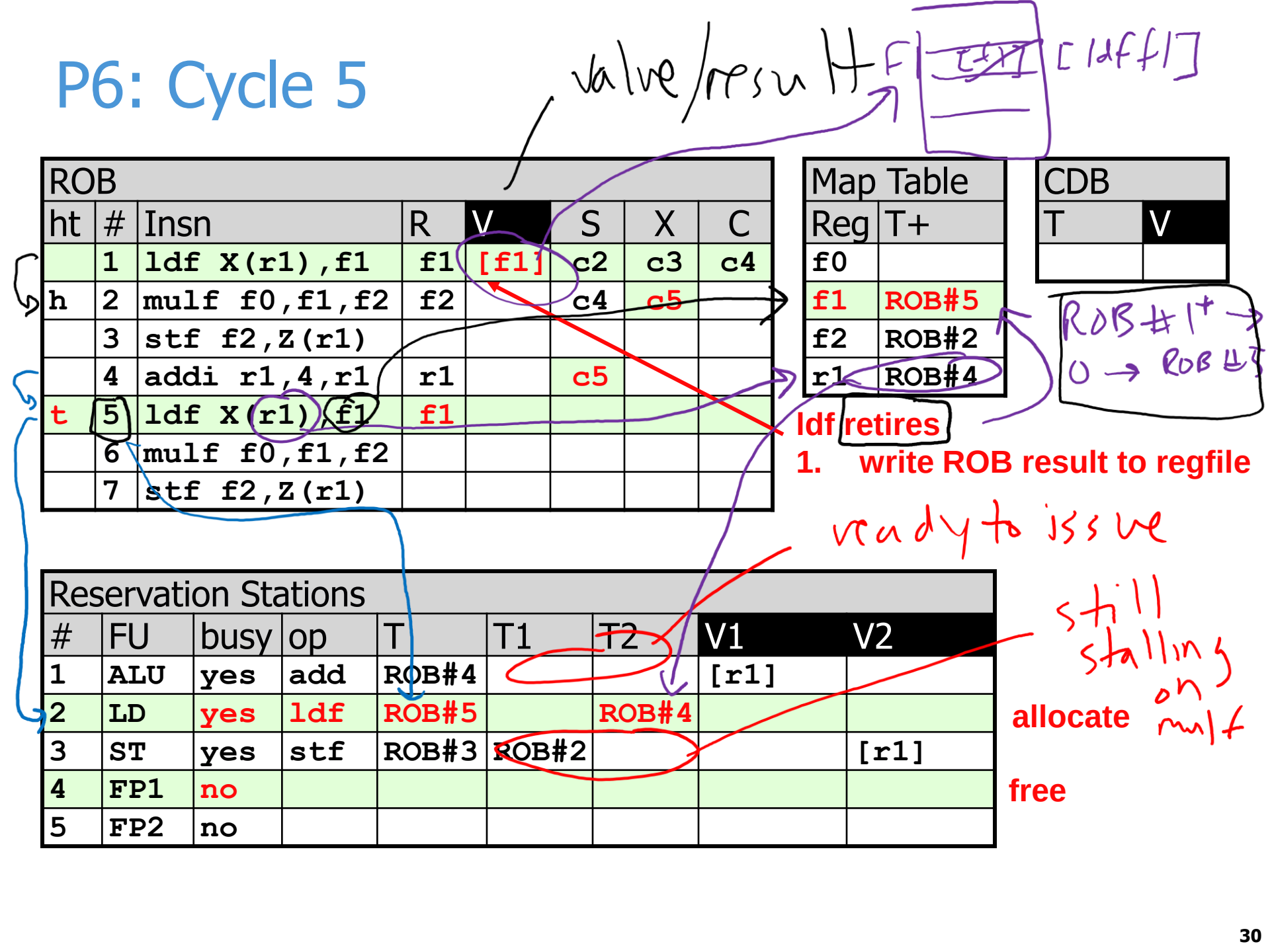
P6: Cycle 4


ROB
ht # Insn R V S X C
h 1 ldf X(r1),f1 f1 [f1] c2 c3 c4
2 mulf f0,f1,f2 f2 c4
3 stf f2,Z(r1)
t 4 addi r1,4,r1 r1
5 ldf X(r1),f1
6 mulf f0,f1,f2
7 stf f2,Z(r1)
Map Table
Reg T+
f0
f1 ROB#1
f2 ROB#2
r1 ROB#4
CDB
T V
ROB#1 [f1]
ldf finished
1. set "ready-in-ROB" bit
2. write result to ROB
3. CDB broadcast
read r1 from RF
Reservation Stations
FU busy op T T1 T2 V1 V2
1 ALU yes add ROB#4 [r1]
2 LD no
3 ST yes stf ROB#3 ROB#2 0 [r1]
4 FP1 yes mulf ROB#2 ROB#1 [f0] CDB.V
5 FP2 no
allocate
still stalling
ROB#1 ready
grab CDB.V
tags cleared → issue now

Announcements 10/8/24

- HTCondor accounts created – **let me know if you did not get an email**
- Project preliminary ideas feedback ongoing (hopefully done tonight)
- HW3 grading ongoing
  - Will release solution tonight to aid in Exam prep
  - Grading unlikely to be done until shortly before exam
- Midterm 1 on Friday
  - Review on Wednesday during lecture – bring questions!

P6: Cycle 5


value/result
F
[ldf f1]
ROB
ht # Insn R V S X C
1 ldf X(r1),f1 f1 [f1] c2 c3 c4
h 2 mulf f0,f1,f2 f2 c4 c5
3 stf f2,Z(r1)
4 addi r1,4,r1 r1 c5
t 5 ldf X(r1),f1 f1
6 mulf f0,f1,f2
7 stf f2,Z(r1)
Map Table
Reg T+
f0
f1 ROB#5
f2 ROB#2
r1 ROB#4
CDB
T V
ROB#1+ → 0 → ROB#1
ldf retires
1. write ROB result to regfile
ready to issue
still stalling on mulf
allocate
free
Reservation Stations
FU busy op T T1 T2 V1 V2
1 ALU yes add ROB#4 [r1]
2 LD yes ldf ROB#5 ROB#4
3 ST yes stf ROB#3 ROB#2 [r1]
4 FP1 no
5 FP2 no

P6: Cycle 6

ROB							
ht	#	Insn	R	V	S	X	C
	1	ldf X(r1),f1	f1	[f1]	c2	c3	c4
h	2	mulf f0,f1,f2	f2		c4	c5+	
	3	stf f2,Z(r1)					
	4	addi r1,4,r1	r1		c5	c6	
	5	ldf X(r1),f1	f1				
t	6	mulf f0,f1,f2	f2				
	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	
f1	ROB#5
f2	ROB#6
r1	ROB#4

CDB	
T	V

ROB#2 → ROB#6

read f0 from RF

still stalling on addi

Reservation Stations								
#	FU	busy	op	T	T1	T2	V1	V2
1	ALU	no						
2	LD	yes	ldf	ROB#5		ROB#4		
3	ST	yes	stf	ROB#3	ROB#2			[r1]
4	FP1	yes	mulf	ROB#6		ROB#5	[f0]	
5	FP2	no						

free

still stalling on allocate mulf

P6: Cycle 7

ROB							
ht	#	Insn	R	V	S	X	C
	1	ldf X(r1),f1	f1	[f1]	c2	c3	c4
h	2	mulf f0,f1,f2	f2		c4	c5	
	3	stf f2,Z(r1)					
	4	addi r1,4,r1	r1	[r1]	c5	c6	c7
	5	ldf X(r1),f1	f1		c7		
t	6	mulf f0,f1,f2	f2				
	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	
f1	ROB#5
f2	ROB#6
r1	ROB#4

CDB	
T	V
ROB#4	[r1]

stall D (no free ST RS)

Reservation Stations								
#	FU	busy	op	T	T1	T2	V1	V2
1	ALU	no						
2	LD	yes	ldf	ROB#5		ROB#4		CDB.V
3	ST	yes	stf	ROB#3	ROB#2			[r1]
4	FP1	yes	mulf	ROB#6		ROB#5	[f0]	
5	FP2	no						

can now issue
ROB#4 ready
grab CDB.V
still stall on ldf
still stalling on mulf

P6: Cycle 8

ROB							
ht	#	Insn	R	V	S	X	C
	1	ldf X(r1),f1	f1	[f1]	c2	c3	c4
h	2	mulf f0,f1,f2	f2	[f2]	c4	c5+	c8
	3	stf f2,Z(r1)			c8		
	4	addi r1,4,r1	r1	[r1]	c5	c6	c7
	5	ldf X(r1),f1	f1		c7	c8	
t	6	mulf f0,f1,f2	f2				
	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	
f1	ROB#5
f2	ROB#6
r1	ROB#4+

CDB	
T	V
ROB#2	[f2]

not head of ROB

stall R for addi (in-order, not h)

ROB#2 invalid in MapTable
don't set "ready-in-ROB"

Reservation Stations								
#	FU	busy	op	T	T1	T2	V1	V2
1	ALU	no						
2	LD	no					CDB.V	
3	ST	yes	stf	ROB#3	ROB#2		[f2]	[r1]
4	FP1	yes	mulf	ROB#6		ROB#5	[f0]	
5	FP2	no						

ready to issue
ROB#2 ready
grab CDB.V

still no ST RS available

still stalling on ld

P6: Cycle 9

still not ROB head

ROB							
ht	#	Insn	R	V	S	X	C
	1	ldf X(r1),f1	f1	[f1]	c2	c3	c4
	2	mulf f0,f1,f2	f2	[f2]	c4	c5+	c8
h	3	stf f2,Z(r1)			c8	c9	
	4	addi r1,4,r1	r1	[r1]	c5	c6	c7
	5	ldf X(r1),f1	f1	[f1]	c7	c8	c9
	6	mulf f0,f1,f2	f2		c9		
t	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	
f1	ROB#5+
f2	ROB#6
r1	ROB#4+

CDB	
T	V
ROB#5	[f1]

ROB#2 ≠ ROB#6, so leave

retire mulf

all pipe stages active at once!

read r1 from ROB#4

Reservation Stations								
#	FU	busy	op	T	T1	T2	V1	V2
1	ALU	no						
2	LD	no						
3	ST	yes	stf	ROB#7	ROB#6		-	ROB#4.V
4	FP1	yes	mulf	ROB#6		ROB#5	[f0]	CDB.V
5	FP2	no						

free, re-allocate

ROB#5 ready

grab CDB.V

can issue now

P6: Cycle 10

still not ROB head

ROB							
ht	#	Insn	R	V	S	X	C
	1	ldf X(r1),f1	f1	[f1]	c2	c3	c4
	2	mulf f0,f1,f2	f2	[f2]	c4	c5+	c8
h	3	stf f2,Z(r1)			c8	c9	c10
	4	addi r1,4,r1	r1	[r1]	c5	c6	c7
	5	ldf X(r1),f1	f1	[f1]	c7	c8	c9
	6	mulf f0,f1,f2	f2		c9	c10	
t	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	
f1	ROB#5+
f2	ROB#6
r1	ROB#4+

CDB	
T	V

nothing on CDB

Reservation Stations								
#	FU	busy	op	T	T1	T2	V1	V2
1	ALU	no						
2	LD	no						
3	ST	yes	stf	ROB#7	ROB#6			ROB#4.V
4	FP1	no						
5	FP2	no						

free

stalling on mulf

P6: Cycle 11

ROB							
ht	#	Insn	R	V	S	X	C
	1	`ldf X(r1),f1`	f1	[f1]	c2	c3	c4
	2	`mulf f0,f1,f2`	f2	[f2]	c4	c5	c8
	3	`stf f2,Z(r1)`			c8	c9	c10
h	4	`addi r1,4,r1`	r1	[r1]	c5	c6	c7
	5	`ldf X(r1),f1`	f1	[f1]	c7	c8	c9
	6	`mulf f0,f1,f2`	f2		c9	c10	
t	7	`stf f2,Z(r1)`					

Map Table	
Reg	T+
f0	
f1	ROB#5+
f2	ROB#6
r1	ROB#4+

CDB	
T	V

retire stf (LSQ)

Reservation Stations								
#	FU	busy	op	T	T1	T2	V1	V2
1	ALU	no						
2	LD	no						
3	ST	yes	stf	ROB#7	ROB#6			ROB#4.V
4	FP1	no						
5	FP2	no						

sth stalling on mult

Precise State in P6

- Point of ROB is maintaining **precise state**
  - How does that work?
  - Easy as 1,2,3
    1. Wait until last good insn retires, first bad insn at ROB head
    2. Clear contents of ROB, RS, and Map Table
    3. Start over
  - Works because zero (0) means the right thing…
    - 0 in ROB/RS → entry is empty
    - Tag == 0 in Map Table → register is in regfile
  - …and because regfile and D$ writes take place at R
  - Example: page fault in first `stf`

P6: Cycle 9 (with precise state)

ROB							
ht	#	Insn	R	V	S	X	C
	1	ldf X(r1),f1	f1	[f1]	c2	c3	c4
	2	mulf f0,f1,f2	f2	[f2]	c4	c5+	c8
h	3	stf f2,Z(r1)			c8	c9	
	4	addi r1,4,r1	r1	[r1]	c5	c6	c7
	5	ldf X(r1),f1	f1	[f1]	c7	c8	c9
	6	mulf f0,f1,f2	f2		c9		
t	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	
f1	ROB#5+
f2	ROB#6
r1	ROB#4+

CDB	
T	V
ROB#5	[f1]

what if here?

PAGE FAULT

Reservation Stations								
#	FU	busy	op	T	T1	T2	V1	V2
1	ALU	no						
2	LD	no						
3	ST	yes	stf	ROB#7	ROB#6			ROB#4.V
4	FP1	yes	mulf	ROB#6		ROB#5	[f0]	CDB.V
5	FP2	no						

P6: Cycle 10 (with precise state)

ROB							
ht	#	Insn	R	V	S	X	C
	1	`ldf X(r1),f1`	f1	[f1]	c2	c3	c4
	2	`mulf f0,f1,f2`	f2	[f2]	c4	c5+	c8
	3	`stf f2,Z(r1)`					
	4	`addi r1,4,r1`					
	5	`ldf X(r1),f1`					
	6	`mulf f0,f1,f2`					
	7	`stf f2,Z(r1)`					

Map Table	
Reg	T+
f0	
f1	
f2	
r1	

CDB	
T	V

faulting insn at ROB head?
CLEAR EVERYTHING

Reservation Stations								
#	FU	busy	op	T	T1	T2	V1	V2
1	ALU	no						
2	LD	no						
3	ST	no						
4	FP1	no						
5	FP2	no						

P6: Cycle 11 (with precise state)

ROB							
ht	#	Insn	R	V	S	X	C
	1	`ldf X(r1),f1`	f1	[f1]	c2	c3	c4
	2	`mulf f0,f1,f2`	f2	[f2]	c4	c5+	c8
ht	3	`stf f2,Z(r1)`					
	4	`addi r1,4,r1`					
	5	`ldf X(r1),f1`					
	6	`mulf f0,f1,f2`					
	7	`stf f2,Z(r1)`					

Map Table	
Reg	T+
f0	
f1	
f2	
r1	

CDB	
T	V

START OVER
(after OS fixes page fault)

Reservation Stations								
#	FU	busy	op	T	T1	T2	V1	V2
1	ALU	no						
2	LD	no						
3	ST	yes	stf	ROB#3			[f4]	[r1]
4	FP1	no						
5	FP2	no						

P6: Cycle 12 (with precise state)

ROB							
ht	#	Insn	R	V	S	X	C
	1	`ldf X(r1),f1`	f1	[f1]	c2	c3	c4
	2	`mulf f0,f1,f2`	f2	[f2]	c4	c5+	c8
h	3	`stf f2,Z(r1)`			c12		
t	4	`addi r1,4,r1`	r1				
	5	`ldf X(r1),f1`					
	6	`mulf f0,f1,f2`					
	7	`stf f2,Z(r1)`					

Map Table	
Reg	T+
f0	
f1	
f2	
r1	ROB#4

CDB	
T	V

Reservation Stations								
#	FU	busy	op	T	T1	T2	V1	V2
1	ALU	yes	addi	ROB#4			[r1]	
2	LD	no						
3	ST	yes	stf	ROB#3			[f4]	[r1]
4	FP1	no						
5	FP2	no						

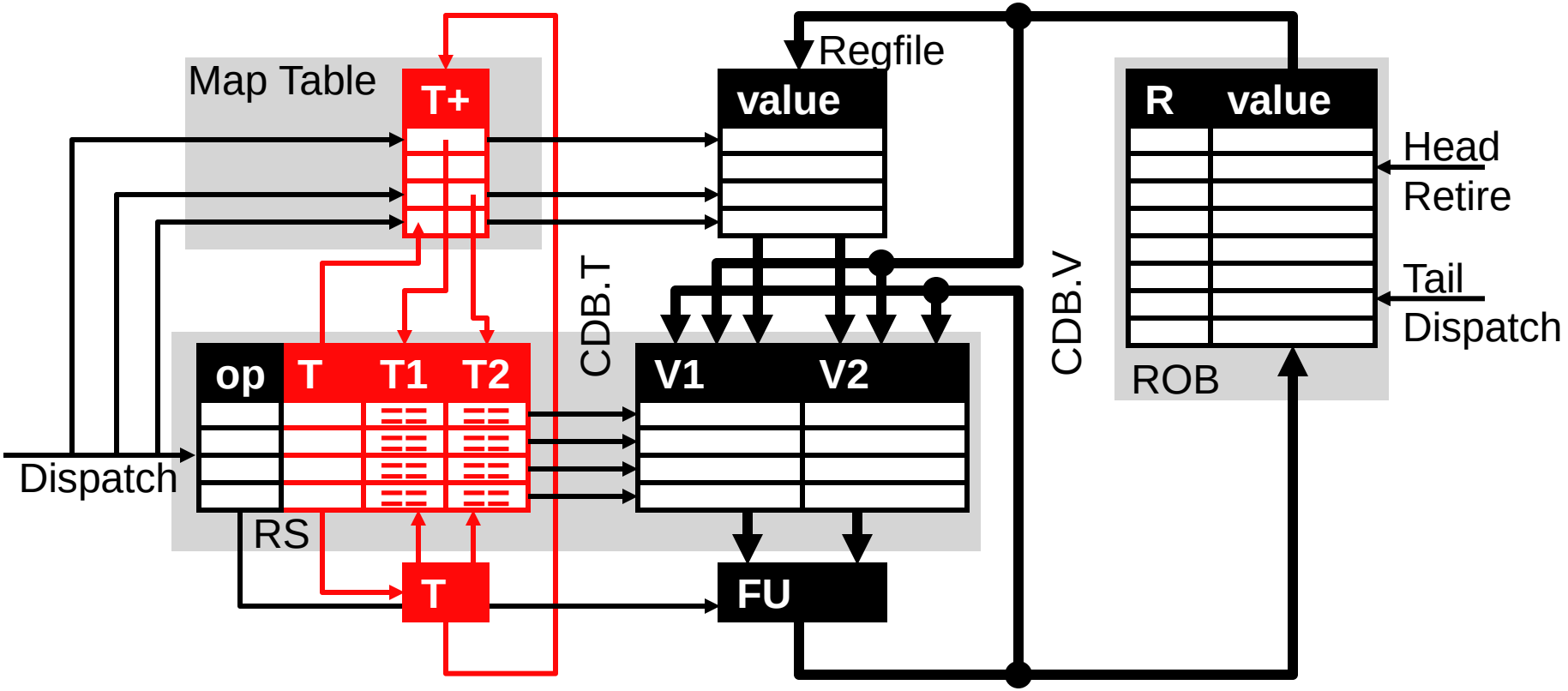
P6 Performance

- In other words: what is the cost of precise state?
  - \+ In general: same performance as "plain" Tomasulo
    - ROB is not a performance device
    - Maybe a little better (RS freed earlier $\rightarrow$ fewer struct hazards)
  - – Unless ROB is too small
    - In which case ROB struct hazards become a problem
  - Rules of thumb for ROB size
    - At least N (width) * number of pipe stages between D and R
    - At least N * $t_{hit\text{-}L2}$
    - Can add a factor of 2 to both if you want
    - What is the rationale behind these?

P6 (Tomasulo+ROB) Redux

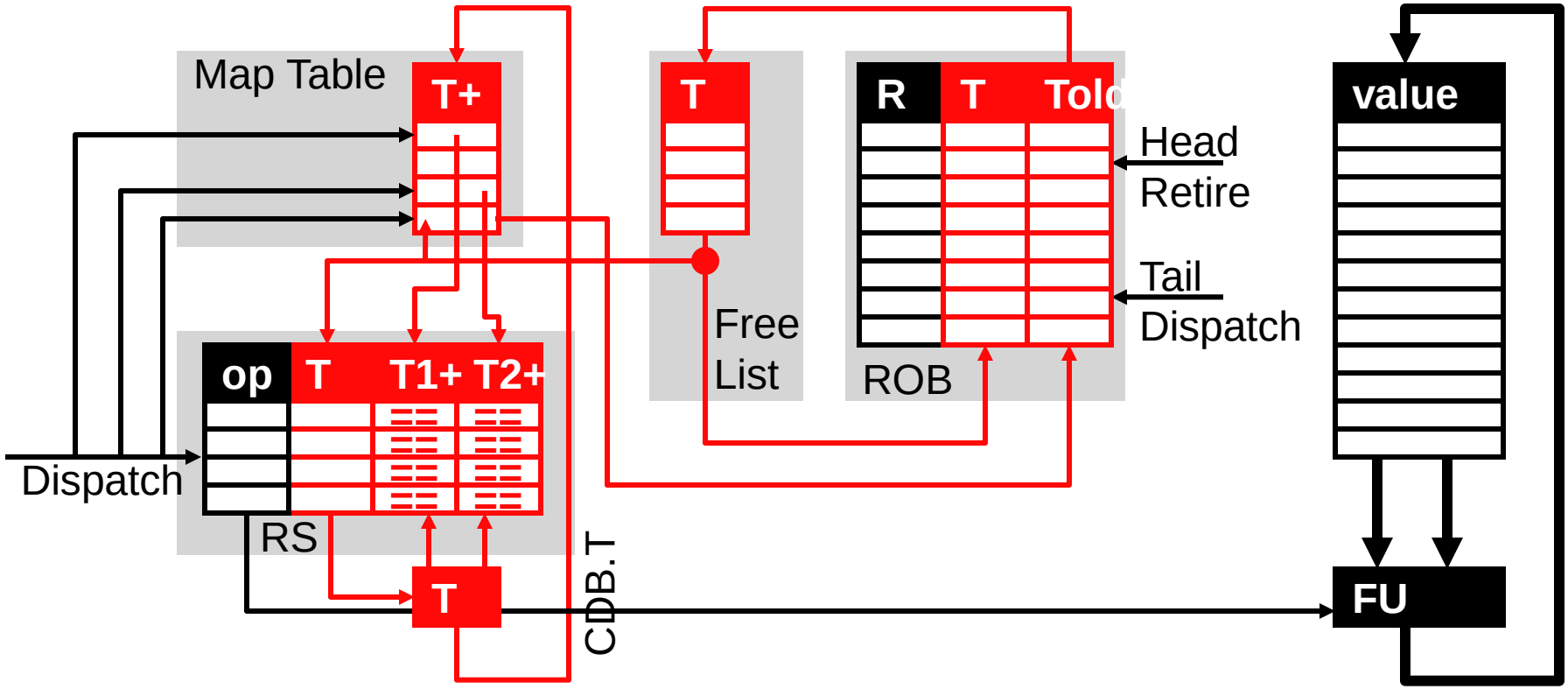
- Popular design for a while
  - (Relatively) easy to implement correctly
    - Anything goes wrong (mispredicted branch, fault, interrupt)?
    - Just clear everything and start again
  - Examples: Intel PentiumPro, IBM/Motorola PowerPC, AMD K6
- Actually making a comeback…
  - Examples: Intel PentiumM
- But went away for a while, why?

The Problem with P6



- Problem for high performance implementations
  - Too much **value movement** (regfile/ROB→RS→ROB→regfile)
  - Multi-input muxes, long buses complicate routing and slow clock

MIPS R10K: Alternative Implementation



- One big **physical register file** holds all data no copies
  - \+ Register file close to FUs → small fast data path
  - ROB and RS "on the side" used only for control and tags

Register Renaming in R10K

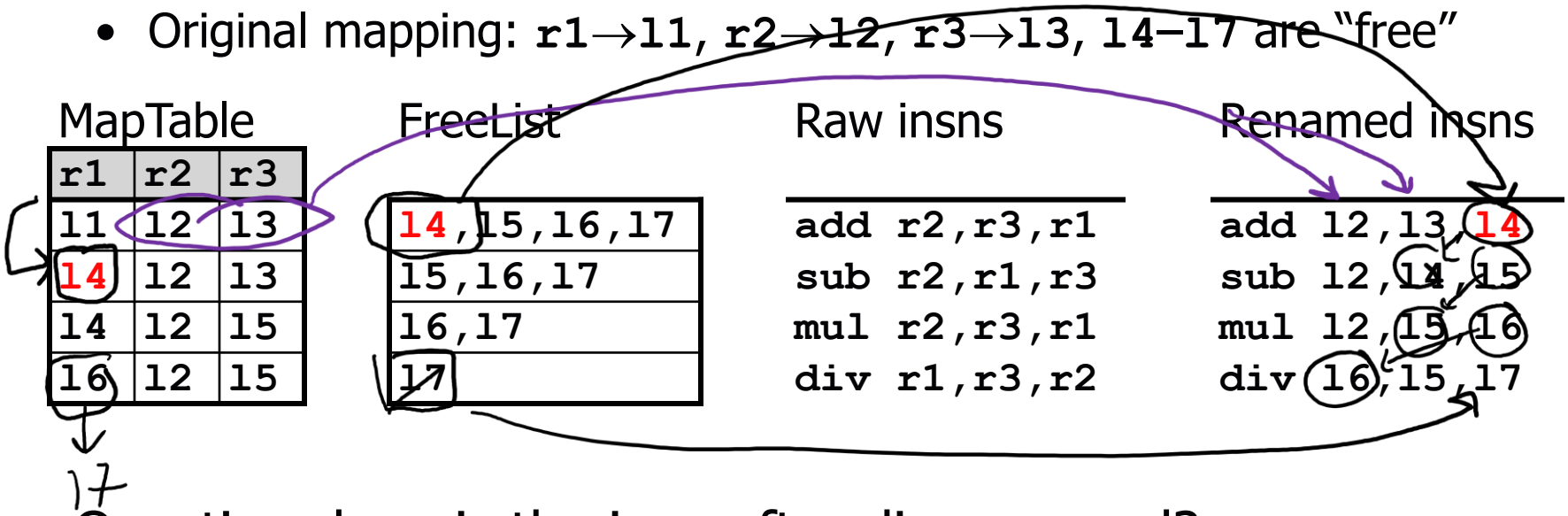
- Architectural register file? Gone
- **Physical register file** holds all values
  - #physical registers = #architectural registers + #ROB entries
  - Map architectural registers to physical registers
  - Removes WAW, WAR hazards (physical registers replace RS copies)
- Fundamental change to **map table**
  - Mappings cannot be 0 (there is no architectural register file)
- **Free list** keeps track of unallocated physical registers
  - ROB is responsible for returning physical registers to free list

Know it's safe to free phys. reg @ retirement

- Conceptually, this is "true register renaming"
  - Have already seen an example

Register Renaming Example

- Parameters
  - Names: `r1,r2,r3`
  - Locations: `l1,l2,l3,l4,l5,l6,l7`
  - Original mapping: `r1`→`l1`, `r2`→`l2`, `r3`→`l3`, `l4`–`l7` are "free"



- Question: how is the insn after div renamed?
  - We are out of free locations (physical registers)
  - Real question: how/when are physical registers freed?

Freeing Registers in P6 and R10K

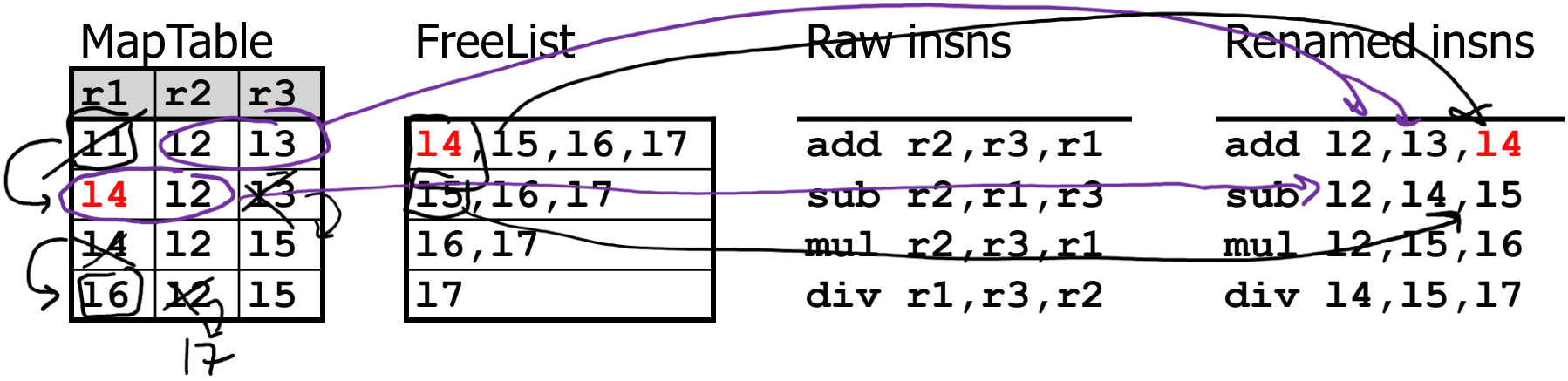
- P6
  - No need to free storage for speculative ("in-flight") values explicitly
  - Temporary storage comes with ROB entry
  - R: copy speculative value from ROB to register file, free ROB entry

- R10K
  - Can't free physical register when insn retires
  - No architectural register to copy value to
  - But…
  - Can free physical register previously mapped to same logical register
  - Why? All insns that will ever read its value have retired

some pending instr. may need

(if my instr is head of ROB)

Freeing Registers in R10K



- When **`add`** retires, free l1
- When **`sub`** retires, free l3
- When **`mul`** retires, free ? l4
- When **`div`** retires, free ? l2
- See the pattern?

R10K Data Structures

- New tags (again)
  - P6: ROB# → R10K: PR# (Phys. Reg.)
- ROB
  - **T**: physical register corresponding to insn's logical output
  - **Told**: physical register previously mapped to insn's logical output
- RS
  - **T**, **T1**, **T2**: output, input physical registers
- Map Table
  - **T+**: PR# (never empty) + "ready" bit
- Free List
  - **T**: PR#

- No values in ROB, RS, or on CDB

R10K Data Structures

ROB							
ht	#	Insn	T	Told	S	X	C
	1	`ldf X(r1),f1`					
	2	`mulf f0,f1,f2`					
	3	`stf f2,Z(r1)`					
	4	`addi r1,4,r1`					
	5	`ldf X(r1),f1`					
	6	`mulf f0,f1,f2`					
	7	`stf f2,Z(r1)`					

Map Table	
Reg	T+
f0	PR#1+
f1	PR#2+
f2	PR#3+
r1	PR#4+

CDB
T

Free List
PR#5,PR#6, PR#7,PR#8

Reservation Stations						
#	FU	busy	op	T	T1	T2
1	ALU	no				
2	LD	no				
3	ST	no				
4	FP1	no				
5	FP2	no				

Notice I: no values anywhere

Notice II: MapTable is never empty

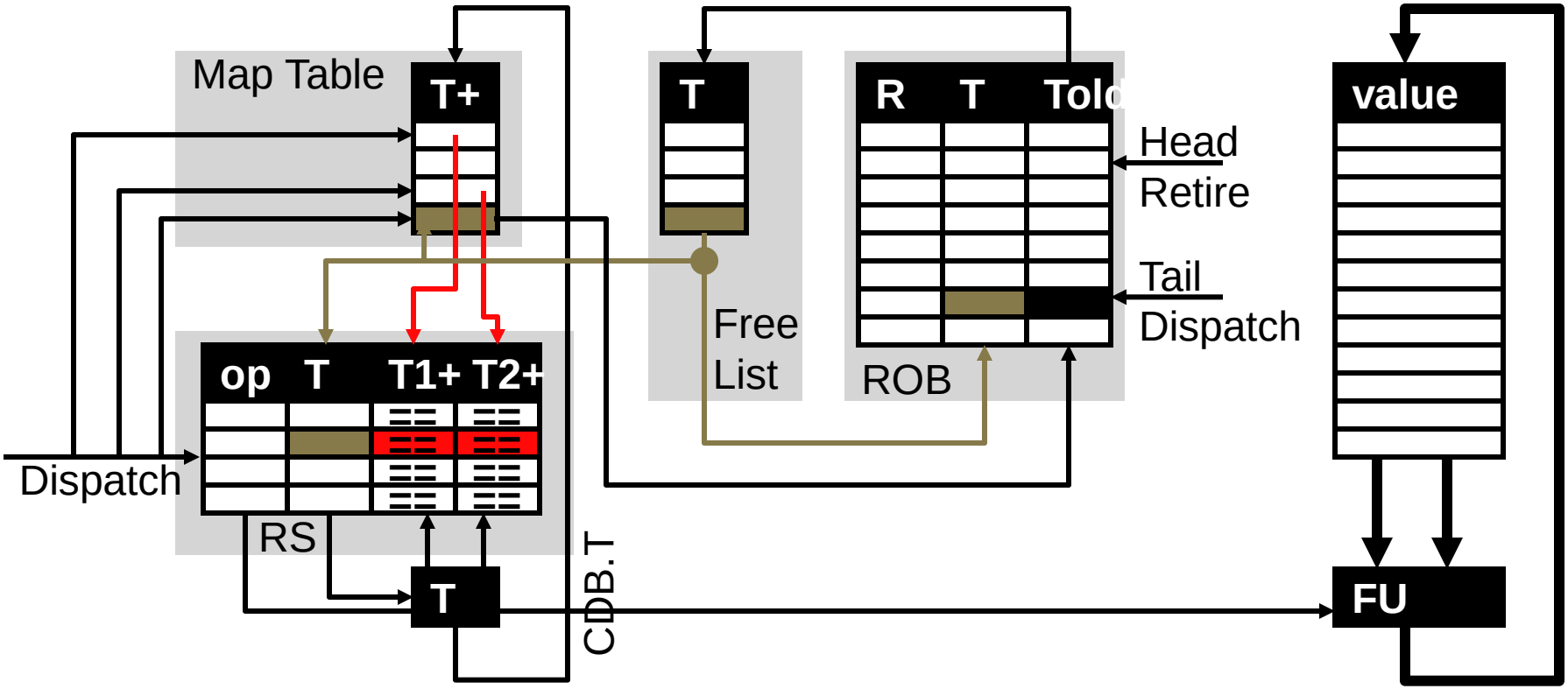
R10K Pipeline

- R10K pipeline structure: F, **D**, S, X, **C**, **R**
  - **D (dispatch)**
    - Structural hazard (RS, ROB, LSQ, **physical registers**) ? stall
    - Allocate RS, ROB, LSQ entries and new physical register (T)
    - **Record previously mapped physical register (Told)**
  - **C (complete)**
    - Write destination physical register
  - **R (retire)**
    - ROB head not complete ? Stall
    - Handle any exceptions
    - Store write LSQ head to D$
    - Free ROB, LSQ entries
    - **Free previous physical register (Told)**

free list is empty?

(update free list)

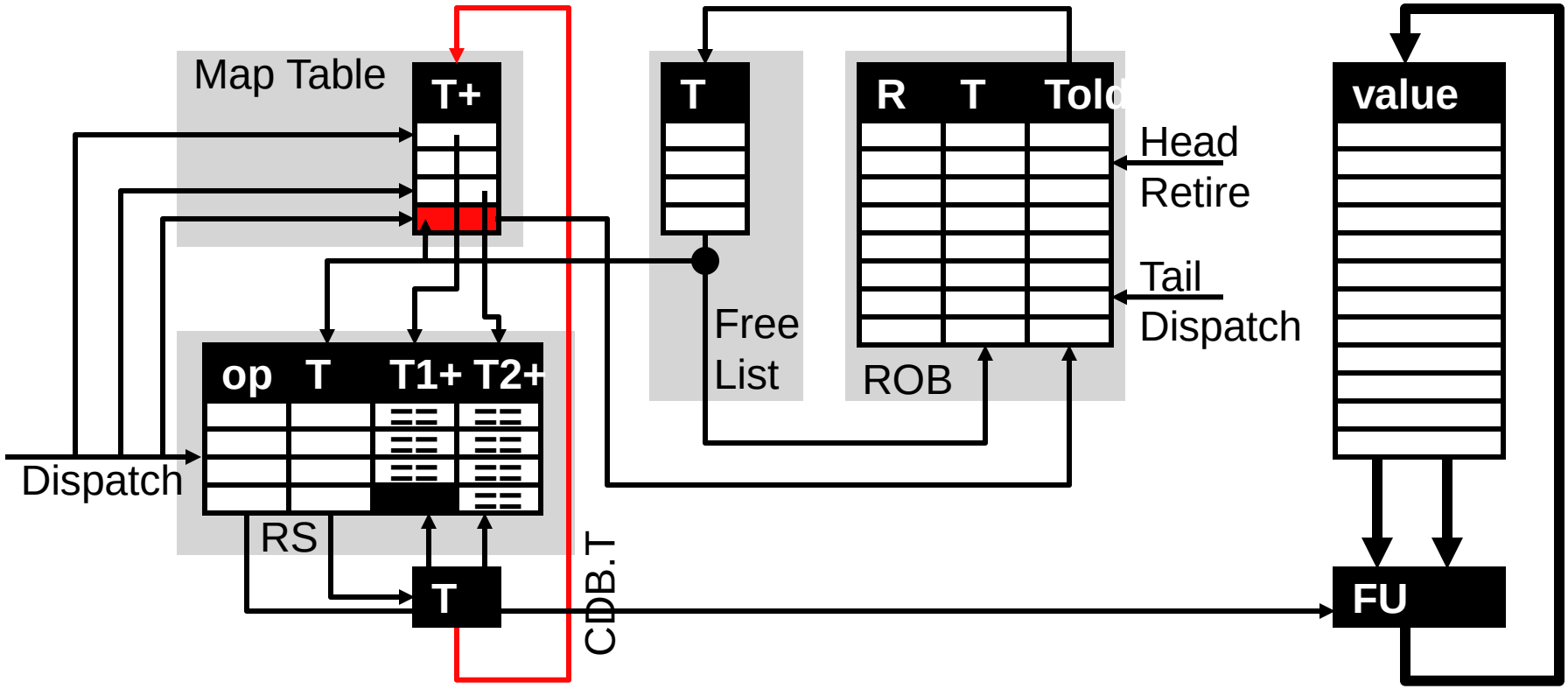
R10K Dispatch (D)



- Read preg (physical register) tags for input registers, store in RS
- Read preg tag for output register, store in ROB (Told)
- Allocate new preg (free list) for output register, store in RS, ROB, Map Table

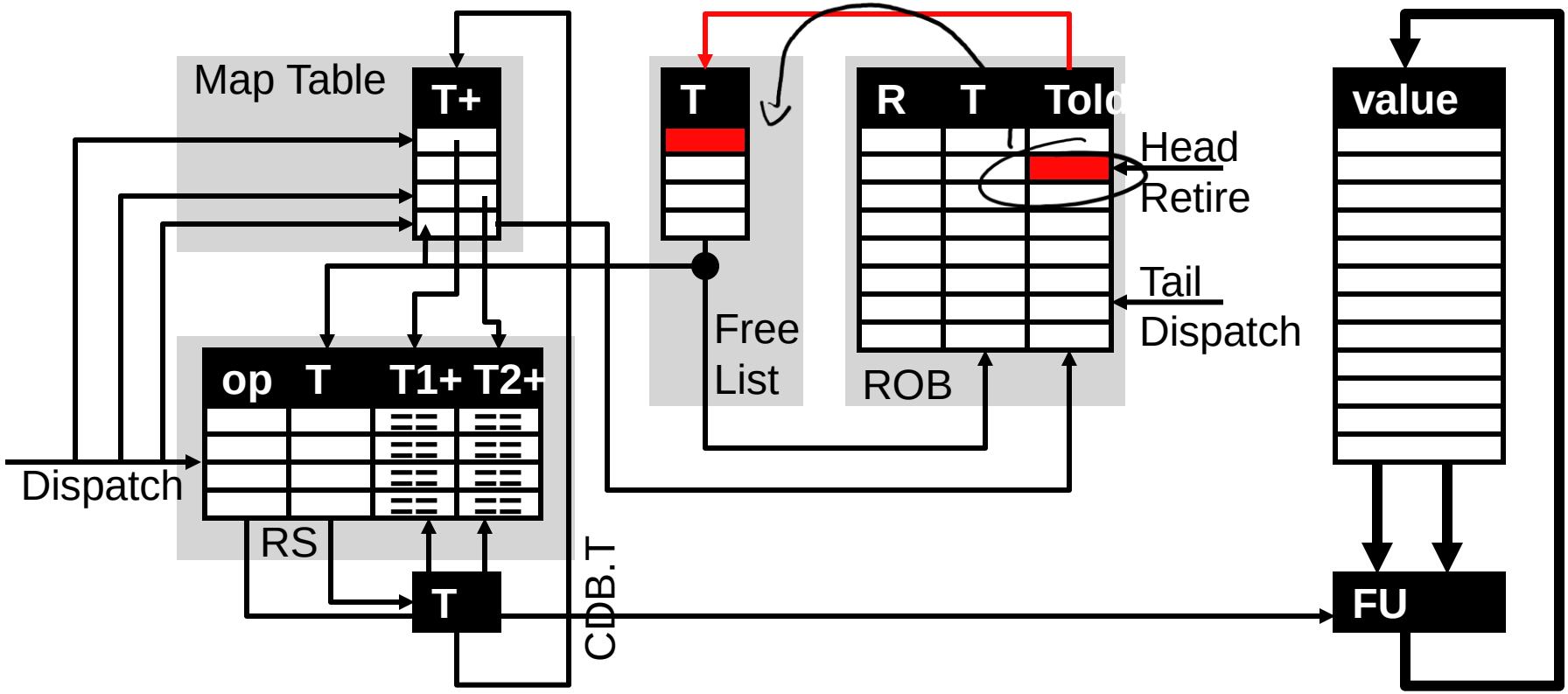
old mapping for dest reg.

R10K Complete (C)



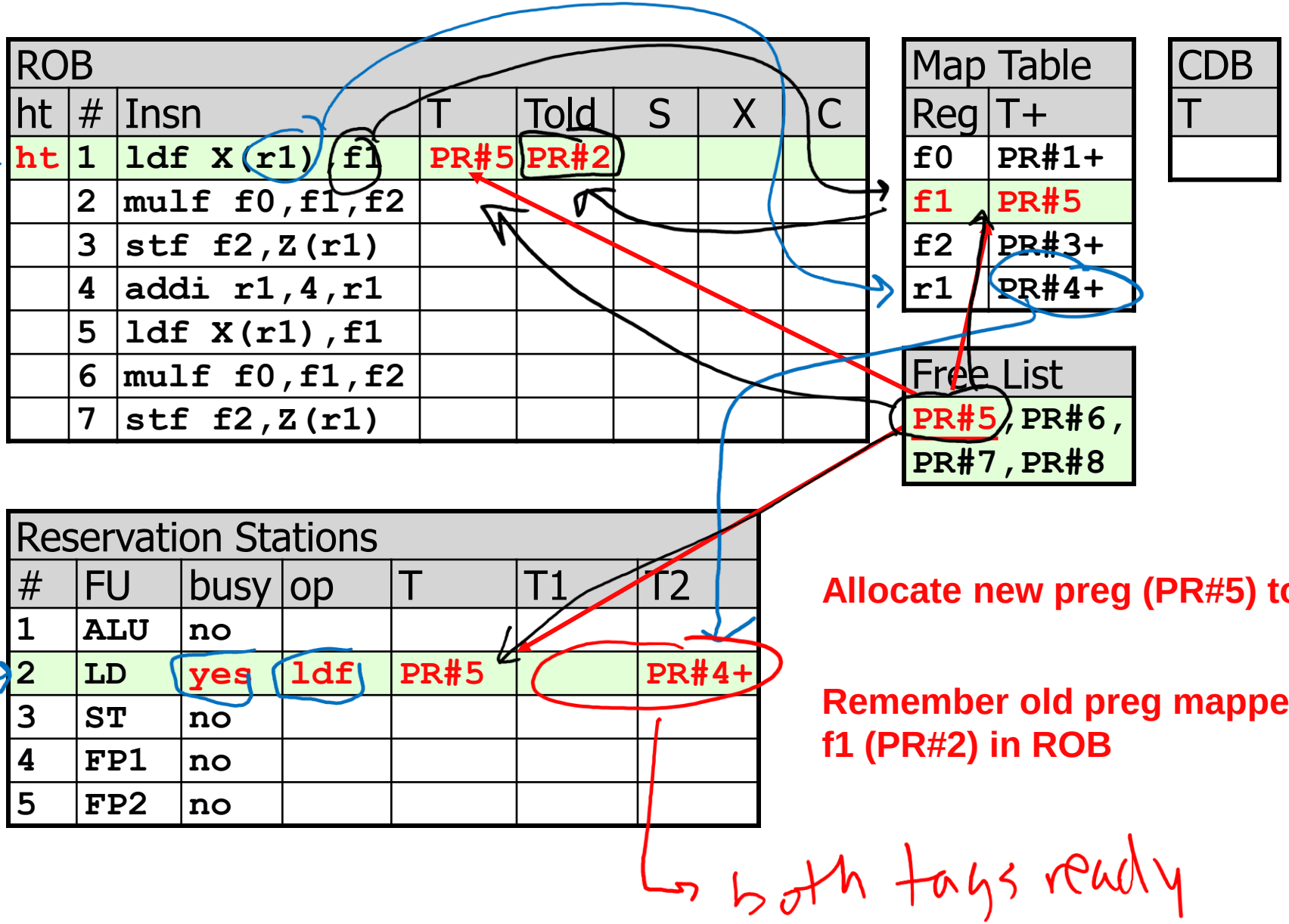
- Set insn's output register ready bit in map table
- Set ready bits for matching input tags in RS

R10K Retire (R)



- Return Told of ROB head to free list

R10K: Cycle 1



ROB							
ht	#	Insn	T	Told	S	X	C
ht	1	ldf X(r1),f1	PR#5	PR#2			
	2	mulf f0,f1,f2					
	3	stf f2,Z(r1)					
	4	addi r1,4,r1					
	5	ldf X(r1),f1					
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	PR#1+
f1	PR#5
f2	PR#3+
r1	PR#4+

CDB
T

Free List
PR#5,PR#6, PR#7,PR#8

Reservation Stations						
#	FU	busy	op	T	T1	T2
1	ALU	no				
2	LD	yes	ldf	PR#5		PR#4+
3	ST	no				
4	FP1	no				
5	FP2	no				

Allocate new preg (PR#5) to f1

Remember old preg mapped to f1 (PR#2) in ROB

both tags ready

R10K: Cycle 2

PR#3 was old mapping for f3

ROB							
ht	#	Insn	T	Told	S	X	C
h	1	ldf X(r1),f1	PR#5	PR#2	c2		
t	2	mulf f0,f1,f2	PR#6	PR#3			
	3	stf f2,Z(r1)					
	4	addi r1,4,r1					
	5	ldf X(r1),f1					
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	PR#1+
f1	PR#5
f2	PR#6
r1	PR#4+

CDB
T

Free List
PR#6, PR#7, PR#8

Reservation Stations						
#	FU	busy	op	T	T1	T2
1	ALU	no				
2	LD	yes	ldf	PR#5		PR#4+
3	ST	no				
4	FP1	yes	mulf	PR#6	PR#1+	PR#5
5	FP2	no				

Allocate new preg (PR#6) to f2

Remember old preg mapped to f3 (PR#3) in ROB

ready (f0)

not ready ⇒ stall

R10K: Cycle 3

ROB							
ht	#	Insn	T	Told	S	X	C
h	1	ldf X(r1),f1	PR#5	PR#2	c2	c3	
	2	mulf f0,f1,f2	PR#6	PR#3			
t	3	stf f2,Z(r1)					
	4	addi r1,4,r1					
	5	ldf X(r1),f1					
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	PR#1+
f1	PR#5
f2	PR#6
r1	PR#4+

CDB
T

Free List
PR#7,PR#8

Reservation Stations						
#	FU	busy	op	T	T1	T2
1	ALU	no				
2	LD	no				
3	ST	yes	stf		PR#6	PR#4+
4	FP1	yes	mulf	PR#6	PR#1+	PR#5
5	FP2	no				

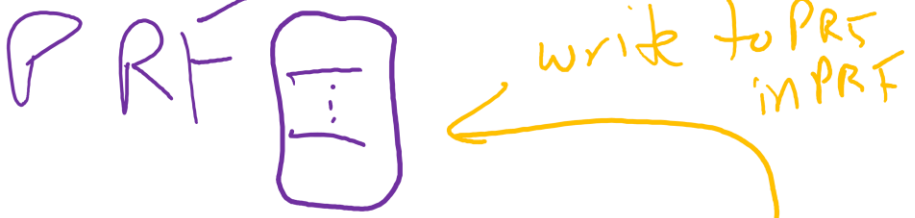
Stores are not allocated pregs

Free

stall on ld f

stall on mulf

R10K: Cycle 4



ROB							
ht	#	Insn	T	Told	S	X	C
h	1	ldf X(r1),f1	PR#5	PR#2	c2	c3	c4
	2	mulf f0,f1,f2	PR#6	PR#3	c4		
	3	stf f2,Z(r1)					
t	4	addi r1,4,r1	PR#7	PR#4			
	5	ldf X(r1),f1					
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	PR#1+
f1	PR#5+
f2	PR#6
r1	PR#7

CDB
T
PR#5

PR#4+ → PR#7

Free List	
PR#7	PR#8

Reservation Stations						
#	FU	busy	op	T	T1	T2
1	ALU	yes	addi	PR#7	PR#4+	
2	LD	no				
3	ST	yes	stf		PR#6	PR#4+
4	FP1	yes	mulf	PR#6	PR#1+	PR#5+
5	FP2	no				

ldf completes
set MapTable ready bit

Match PR#5 tag from CDB & issue

still stalling on mulf

R10K: Cycle 5

ROB							
ht	#	Insn	T	Told	S	X	C
	1	ldf X(r1),f1	PR#5	PR#2	c2	c3	c4
h	2	mulf f0,f1,f2	PR#6	PR#3	c4	c5	
	3	stf f2,Z(r1)					
	4	addi r1,4,r1	PR#7	PR#4	c5		
t	5	ldf X(r1),f1	PR#8	PR#5			
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	PR#1+
f1	PR#8
f2	PR#6
r1	PR#7

CDB
T

Free List
PR#8,PR#2

Reservation Stations						
#	FU	busy	op	T	T1	T2
1	ALU	yes	addi	PR#7	PR#4+	
2	LD	yes	ldf	PR#8		PR#7
3	ST	yes	stf		PR#6	PR#4+
4	FP1	no				
5	FP2	no				

Free

ldf retires
Return PR#2 to free list

Precise State in R10K

- Problem with R10K design? Precise state is more difficult
  - Physical registers are written out-of-order (at C)
  - That's OK, there is no architectural register file
  - We can "free" written registers and "restore" old ones
  - Do this by manipulating the Map Table and Free List, not regfile

- Two ways of restoring Map Table and Free List
  - Option I: serial rollback using T, T_{old} ROB fields
    ±Slow, but simple
  - Option II: single-cycle restoration from some checkpoint
    ±Fast, but checkpoints are expensive
  - Modern processor compromise: **make common case fast**
    - Checkpoint only (low-confidence) branches (frequent rollbacks)
    - Serial recovery for page-faults and interrupts (rare rollbacks)

R10K: Cycle 5 (with precise state)

ROB							
ht	#	Insn	T	Told	S	X	C
	1	ldf X(r1),f1	PR#5	PR#2	c2	c3	c4
h	2	mulf f0,f1,f2	PR#6	PR#3	c4	c5	
	3	stf f2,Z(r1)					
	4	addi r1,4,r1	PR#7	PR#4	c5		
t	5	ldf X(r1),f1	PR#8	PR#5			
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	PR#1+
f1	PR#8
f2	PR#6
r1	PR#7

CDB
T

Free List
PR#8,PR#2

Reservation Stations						
#	FU	busy	op	T	T1	T2
1	ALU	yes	addi	PR#7	PR#4+	
2	LD	yes	ldf	PR#8		PR#7
3	ST	yes	stf		PR#6	PR#4+
4	FP1	no				
5	FP2	no				

undo insns 3-5
(doesn't matter why)
use serial rollback

R10K: Cycle 6 (with precise state)

ROB							
ht	#	Insn	T	Told	S	X	C
	1	ldf X(r1),f1	PR#5	PR#2	c2	c3	c4
h	2	mulf f0,f1,f2	PR#6	PR#3	c4	c5	
	3	stf f2,Z(r1)					
t	4	addi r1,4,r1	PR#7	PR#4	c5		
	5	ldf X(r1),f1	PR#8	PR#5			
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	PR#1+
f1	PR#5+
f2	PR#6
r1	PR#7

CDB
T

Free List
PR#2,PR#8

Reservation Stations						
#	FU	busy	op	T	T1	T2
1	ALU	yes	addi	PR#7	PR#4+	
2	LD	no				
3	ST	yes	stf		PR#6	PR#4+
4	FP1	no				
5	FP2	no				

undo ldf (ROB#5)
1. free RS
2. free T (PR#8), return to FreeList
3. restore MT[f1] to Told (PR#5)
4. free ROB#5

insns may execute during rollback (not shown)

R10K: Cycle 7 (with precise state)

ROB							
ht	#	Insn	T	Told	S	X	C
	1	ldf X(r1),f1	PR#5	PR#2	c2	c3	c4
h	2	mulf f0,f1,f2	PR#6	PR#3	c4	c5	
t	3	stf f2,Z(r1)					
	4	addi r1,4,r1	PR#7	PR#4	c5		
	5	ldf X(r1),f1					
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	PR#1+
f1	PR#5+
f2	PR#6
r1	PR#4+

CDB
T

Free List
PR#2, PR#8, PR#7

Reservation Stations						
#	FU	busy	op	T	T1	T2
1	ALU	no				
2	LD	no				
3	ST	yes	stf		PR#6	PR#4+
4	FP1	no				
5	FP2	no				

undo addi (ROB#4)
1. free RS
2. free T (PR#7), return to FreeList
3. restore MT[r1] to Told (PR#4)
4. free ROB#4

R10K: Cycle 8 (with precise state)

ROB							
ht	#	Insn	T	Told	S	X	C
	1	ldf X(r1),f1	PR#5	PR#2	c2	c3	c4
ht	2	mulf f0,f1,f2	PR#6	PR#3	c4	c5	
	3	stf f2,Z(r1)					
	4	addi r1,4,r1					
	5	ldf X(r1),f1					
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	PR#1+
f1	PR#5+
f2	PR#6
r1	PR#4+

CDB
T

Free List
PR#2,PR#8, PR#7

Reservation Stations						
#	FU	busy	op	T	T1	T2
1	ALU	no				
2	LD	no				
3	ST	no				
4	FP1	no				
5	FP2	no				

undo stf (ROB#3)
1. free RS
2. free ROB#3
3. no registers to restore/free
4. how is D$ write undone?

P6 vs. R10K (Renaming)

Feature	P6	R10K
Value storage	ARF,ROB,RS	PRF
Register read	@D: ARF/ROB → RS	@S: PRF → FU
Register write	@R: ROB → ARF	@C: FU → PRF
Speculative value free	@R: automatic (ROB)	@R: overwriting insn
Data paths	ARF/ROB → RS RS → FU FU → ROB ROB → ARF	PRF → FU FU → PRF
Precise state	Simple: clear everything	Complex: serial/checkpoint

- R10K-style became popular in late 90's, early 00's
  - E.g., MIPS R10K (duh), DEC Alpha 21264, Intel Pentium4
- P6-style could make a comeback
  - Why? Frequency (power) is on the retreat, simplicity is important

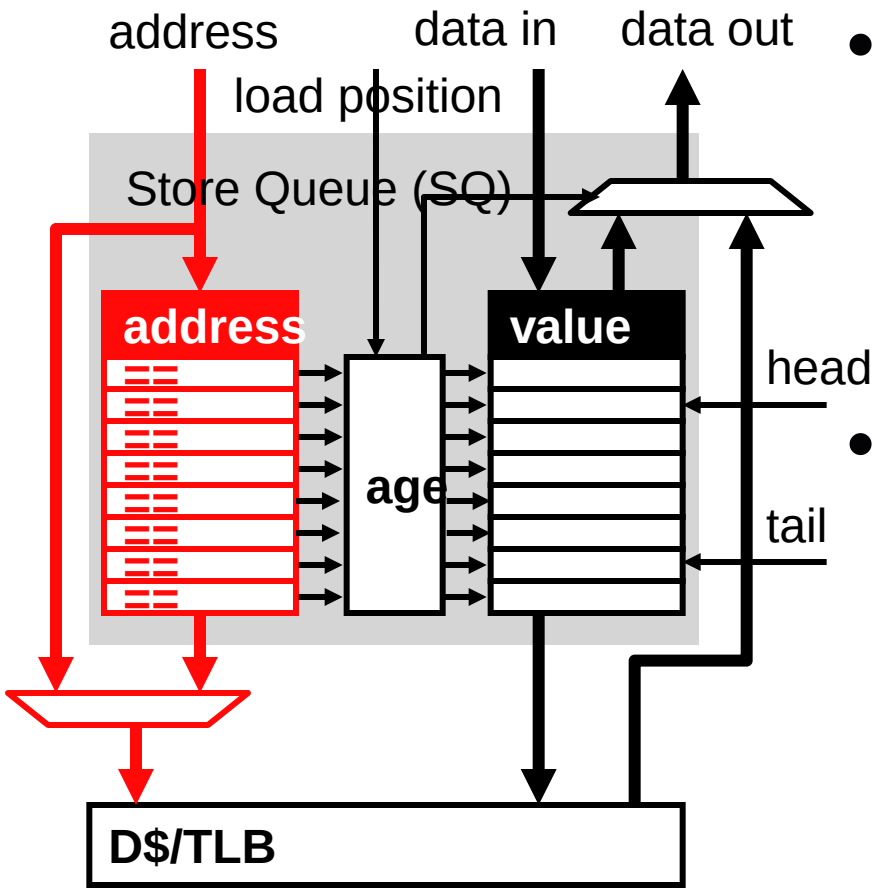
Out of Order Memory Operations

- All insns are easy in out-of-order…
  - Register inputs only
  - Register renaming captures all dependences
  - Tags tell you exactly when you can execute
- … except loads
  - Register and memory inputs (older stores)
  - Register renaming does not tell you all dependences
    - Memory renaming (a little later)
  - How do loads find older in-flight stores to same address (if any)?

Data Memory Functional Unit

- D$/TLB + structures to handle in-flight loads/stores
  - Performs four functions
  - **In-order store retirement**
    - Writes stores to D$ in order
    - Basic, implemented by store queue (SQ)
  - **Store-load forwarding**
    - Allows loads to read values from older un-retired stores
    - Also basic, also implemented by store queue (SQ)
  - **Memory ordering violation detection**
    - Checks load speculation (more later)
    - Advanced, implemented by load queue (LQ)
  - **Memory ordering violation avoidance**
    - Advanced, implemented by dependence predictors

Simple Data Memory FU: D$/TLB + SQ

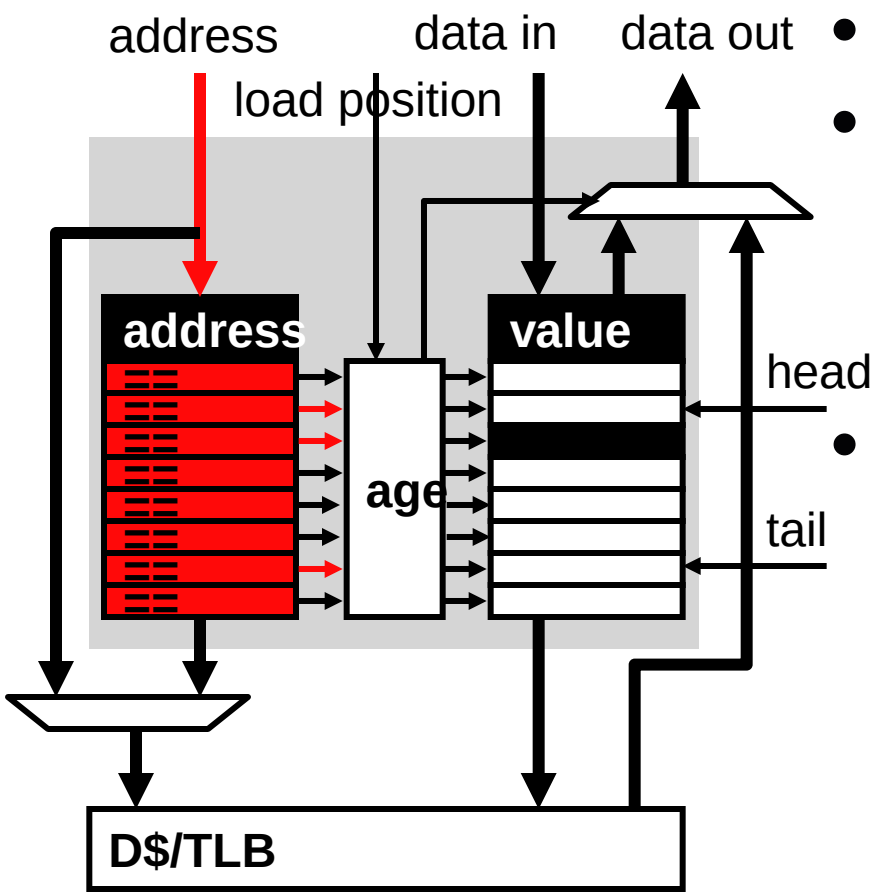


- Just like any other FU
  - 2 register inputs (addr, data in)
  - 1 register output (data out)
  - 1 non-register input (load pos)?
- **Store queue (SQ)**
  - In-flight store address/value
  - In program order (like ROB)
  - Addresses associatively searchable
  - Size heuristic: 15-20% of ROB
- But what does it do?

Data Memory FU "Pipeline"

- Stores
  - **Dispatch (D)**
    - Allocate entry at SQ tail
  - **Execute (X)**
    - Write address and data into corresponding SQ slot
  - **Retire (R)**
    - Write address/data from SQ head to D$, free SQ head
- Loads
  - **Dispatch (D)**
    - Record current SQ tail as "load position"
  - **Execute (X)**
    - Where the good stuff happens

"Out-of-Order" Load Execution



- In parallel with D$ access
- **Send address to SQ**
  - Compare with all store addresses
  - CAM: like FA$, or RS tag match
  - Select all matching addresses
- **Age logic selects youngest store that is older than load**
  - Uses load position input
  - Any? load **"forwards"** value from SQ
  - None? Load gets value from D$

Conservative Load Scheduling

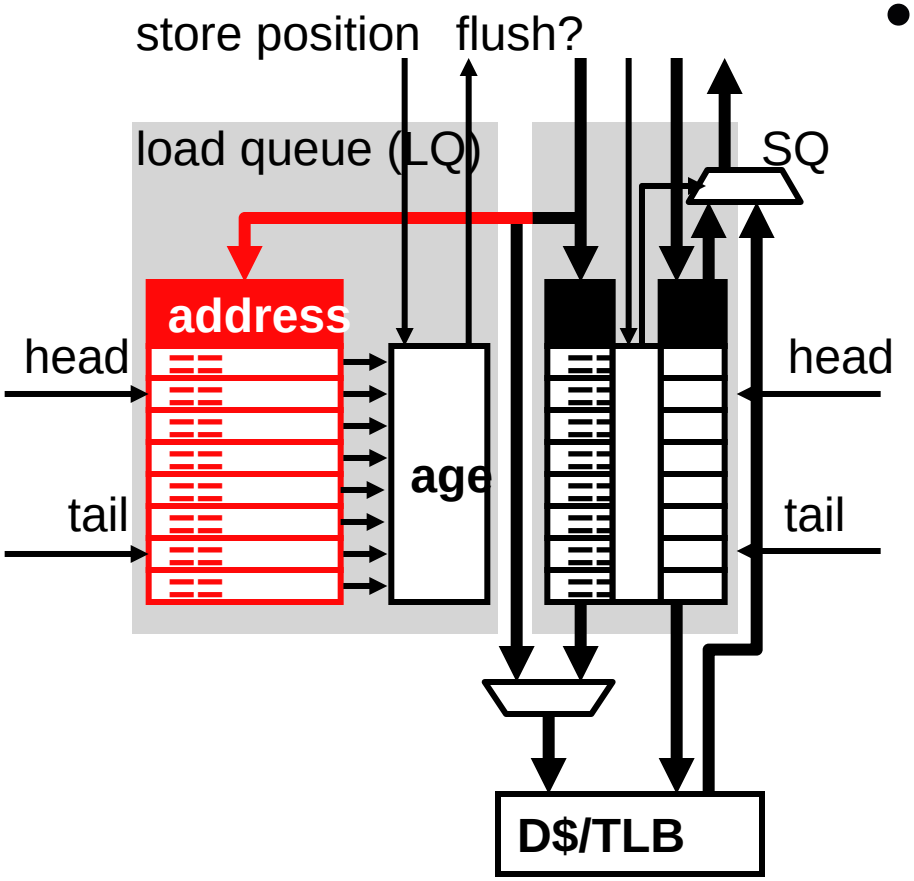
- Why "" in "out-of-order"?
  - + Load can execute out-of-order with respect to (wrt) other loads
  - + Stores can execute out-of-order wrt other stores
  - – **Loads must execute in-order wrt older stores**
    - Load execution requires knowledge of all older store addresses
    - + Simple
    - – Restricts performance
  - Used in P6

Opportunistic Memory Scheduling

- Observe: on average, < 10% of loads forward from SQ
  - Even if older store address is unknown, chances are it won't match
  - Let loads execute in presence of older **"ambiguous stores"**
  + Increases performance
  - But what if ambiguous store *does* match?

- **Memory ordering violation**: load executed too early
  - Must detect…
  - And fix (e.g., by flushing/refetching insns starting at load)

D$/TLB + SQ + LQ

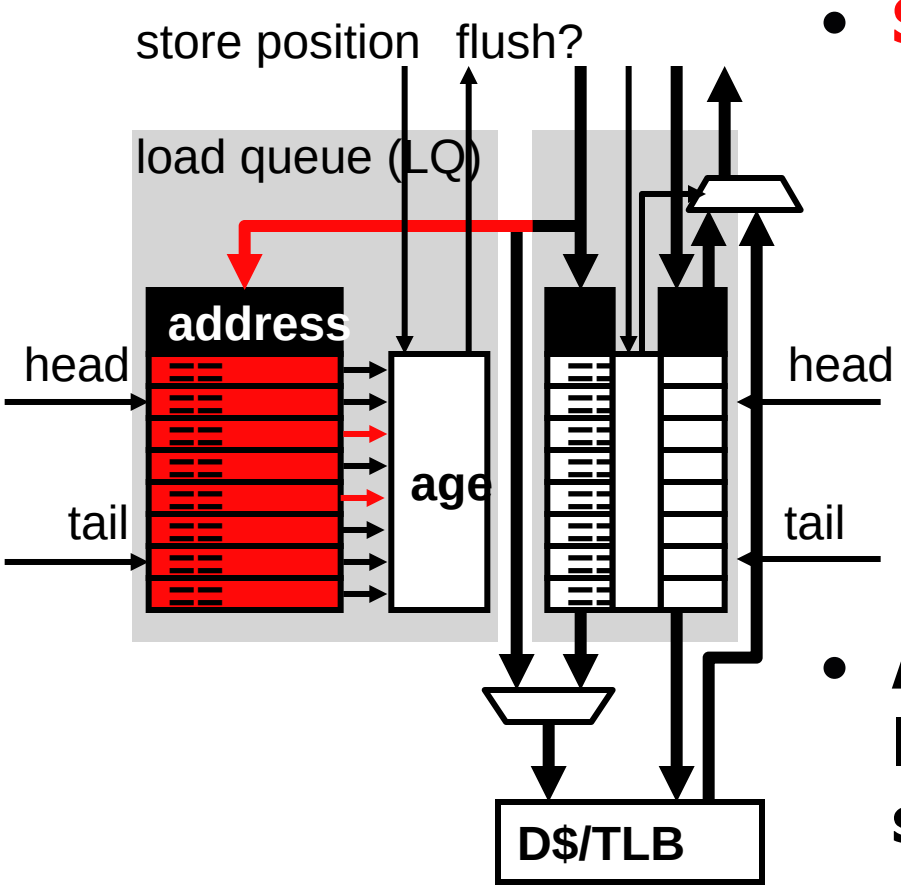


- **Load queue (LQ)**
  - In-flight load addresses
  - In program-order (like ROB,SQ)
  - Associatively searchable
  - Size heuristic: 20-30% of ROB

Advanced Memory "Pipeline" (LQ Only)

- Loads
  - **Dispatch (D)**
    - Allocate entry at LQ tail
  - **Execute (X)**
    - Write address into corresponding LQ slot
- Stores
  - **Dispatch (D)**
    - Record current LQ tail as "store position"
  - **Execute (X)**
    - Where the good stuff happens

Detecting Memory Ordering Violations



- **Store sends address to LQ**
  - Compare with all load addresses
  - Selecting matching addresses
  - Matching address?
    - Load executed before store
    - Violation
    - Fix!
- **Age logic selects oldest load that is younger than store**
  - Use store position
  - Processor flushes and restarts

Intelligent Load Scheduling

- Opportunistic scheduling better than conservative…
  - + Avoids many unnecessary delays
- …but not significantly
  - – Introduces a few flushes, but each is much costlier than a delay

- Observe: loads/stores that cause violations are "stable"
  - Dependences are mostly program based, program doesn't change
  - Scheduler is deterministic

- Exploit: **intelligent load scheduling**
  - Hybridize conservative and opportunistic
  - Predict which loads, or load/store pairs will cause violations
  - Use conservative scheduling for those, opportunistic for the rest

Memory Dependence Prediction

- Store-blind prediction
  - Predict load only, wait for all older stores to execute
  - ± Simple, but a little too heavy handed
  - Example: Alpha 21264

- Store-load pair prediction
  - Predict load/store pair, wait only for one store to execute
  - ± More complex, but minimizes delay
  - Example: Store-Sets
    - Load identifies the right dynamic store in two steps
    - Store-Set Table: load-PC → store-PC
    - Last Store Table: store-PC → SQ index of most recent instance
  - Implemented in next Pentium? (guess)

Limits of Insn-Level Parallelism (ILP)

- Before we build a big superscalar… how much ILP is there?
  - **ILP: instruction-level parallelism** [Fisher`81]
  - Sustainable rate of useful instruction execution
- ILP limit study
  - Assume perfect/infinite hardware, successively add realism
  - Examples: [Wall'88][Wilson+Lam'92]
  - Some surprising results
    - \+ Perfect/infinite "theoretical" ILP: int > 50, FP > 150
      - Sometimes called the **"dataflow limit"**
    - – Real machine "actual" ILP: int ~2, FP ~ 3
    - Fundamental culprits: branch prediction, memory latency
    - Engineering culprits: "window" (RS/SQ/regfile) size, issue width

Clock Rate vs. IPC

- Does frequency vs. width tradeoff actually work?
  - Yes in some places, no in others
  + **Yes**: fetch, decode, rename, retire (all the in-order stages)
  – **No**: issue, execute, complete (all the out-of-order stages)
  - What's the difference?
    - Out-of-order: parallelism doesn't help if insns themselves serial
      - 2 dependent insns execute in 2 cycles, regardless of width
    - In-order: inter-insn parallelism doesn't matter

- Intel Pentium4: **multiple clock domains**
  - In-order stages run at 3.4 GHz, out-of-order stages at 6.8 GHz!
  - Frequency $\propto$ $\text{Power}_{\text{dynamic}}$ $\rightarrow$ high frequency only where necessary

Dynamic Scheduling Redux

- Dynamic scheduling is a performance technique
- But what about…
  - **"Scalability"**: how big can we profitably make it?
  - Power/energy?
  - Reliability?

"Scalability"

- **Scalability**: how big/wide should we make a window?
  - Bigger/wider structures (can) improve IPC, but degrade clock
    - Where is the cross-over?
  - Caveat: scalability is conjunctive (the "Anna Karenina" principle)
    - For a design to be scalable, all components must be scalable

- Non-scalable (and scalable) structures
  - Mostly in execution core (see clock rate vs. IPC)
  - N^2 networks (e.g., bypassing network)
  - Large SRAMs with many read/write ports (e.g., physical regfile)
  - Large multi-ported CAMs (e.g., scheduler or reservation stations)
  - Large age-ordered CAMs (e.g., load and store queues)
  - A lot of current research on scalable versions of these structures
  - \+ ROB is not a problem: few ports, none in "execution core" really

Pentium III vs. Pentium4 (Processors)

Feature	Pentium III	Pentium 4
Peak clock	800 MHz	3.4 GHz (6.8 internal)
Pipeline stages	15	22
Branch prediction	512 local + 512 BTB	2K hybrid + 2K BTB
Primary caches	16KB 4-way	8KB 4-way + 64KB T$
L2	512KB-2MB	256KB-2MB
Fetch width	16 bytes	3 μops (16 bytes on miss)
Rename/retire width	3 μops	3 μops
Execute width	5 μops	7 μops (X2)
Register renaming	P6	R10K
ROB/RS size	40/20	128/60
Load scheduling	Conservative	Intelligent
Anything else?	No	Hyperthreading

Dynamic Scheduling and Power/Energy

- Is dynamic scheduling low-power?
  - Probably not
  - New SRAMs consume a lot of power
    - Re-order buffer, reservation stations, **physical register file**
  - New CAMs consume even more (relatively)
    - Reservation stations, **load/store queue**
- Is dynamic scheduling low-energy?
  - ± Could be
  - Does performance improvement offset power increase?
  - Are there "deep sleep" modes?

Unit Summary

- Modern dynamic scheduling must support precise state
  - A software sanity issue, not a performance issue
- Strategy: Writeback → Complete (OoO) + Retire (iO)
- Two basic designs
  - P6: Tomasulo + re-order buffer, copy based register renaming
    ±Precise state is simple, but fast implementations are difficult
  - R10K: implements true register renaming
    ±Easier fast implementations, but precise state is more complex
- Out-of-order memory operations
  - Store queue: conservative load scheduling (iO wrt older stores)
  - Load queue: opportunistic load scheduling (OoO wrt older stores)
  - Intelligent memory scheduling: hybrid

Dynamic Scheduling Summary

- Out-of-order execution: a performance technique
  - Easier/more effective in hardware than software (isn't everything?)
  - Idea: make scheduling transparent to software
- Feature I: Dynamic scheduling (iO $\rightarrow$ OoO)
  - "Performance" piece: re-arrange insns into high-performance order
  - Decode (iO) $\rightarrow$ dispatch (iO) + issue (OoO)
  - Two algorithms: Scoreboard, Tomasulo
- Feature II: Precise state (OoO $\rightarrow$ iO)
  - "Correctness" piece: put insns back into program order
  - Writeback (OoO) $\rightarrow$ complete (OoO) + retire (iO)
  - Two designs: P6, R10K
- Don't forget about memory scheduling