Speculative Execution

Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
M.I.T.

http://www.csg.csail.mit.edu/6.823

Speculative Execution Recipe

1. Proceed ahead despite unresolved dependencies using a prediction for an architectural or micro-architectural value

2. Maintain both old and new values on updates to architectural (and often micro-architectural) state.

3. After sure that there was no mis-speculation and there will be no more uses of the old values, discard old values and just use new values.

OR

3. In event of mis-speculation dispose of all new values, restore old values and re-execute from point before mis-speculation

Why might one use old values? O-O-O WAR hazards

Value Management Strategies

Greedy (or Eager) Update:

- Update value in place, and
- Maintain a log of old values to use for recovery.

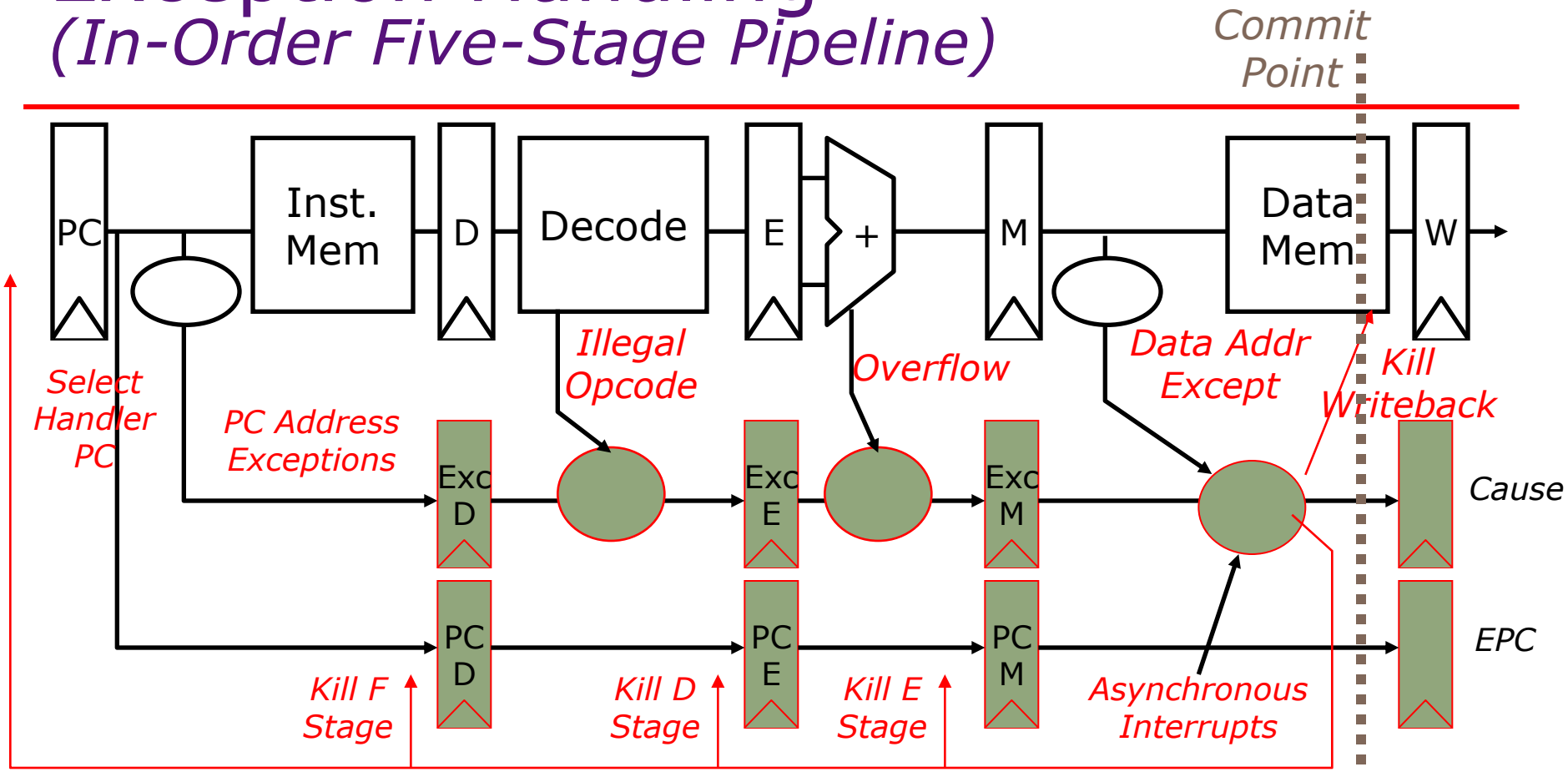
Lazy Update:

- Buffer new value leaving old value in place.
- Replace old value only at 'commit' time.

Why leave an old value in place?

- Old value can be used after new value is generated
- Simplified recovery

Exception Handling
(In-Order Five-Stage Pipeline)



Strategy for PC? Greedy – update immediately

Strategy for Registers? Lazy – update at commit

Misprediction Recovery

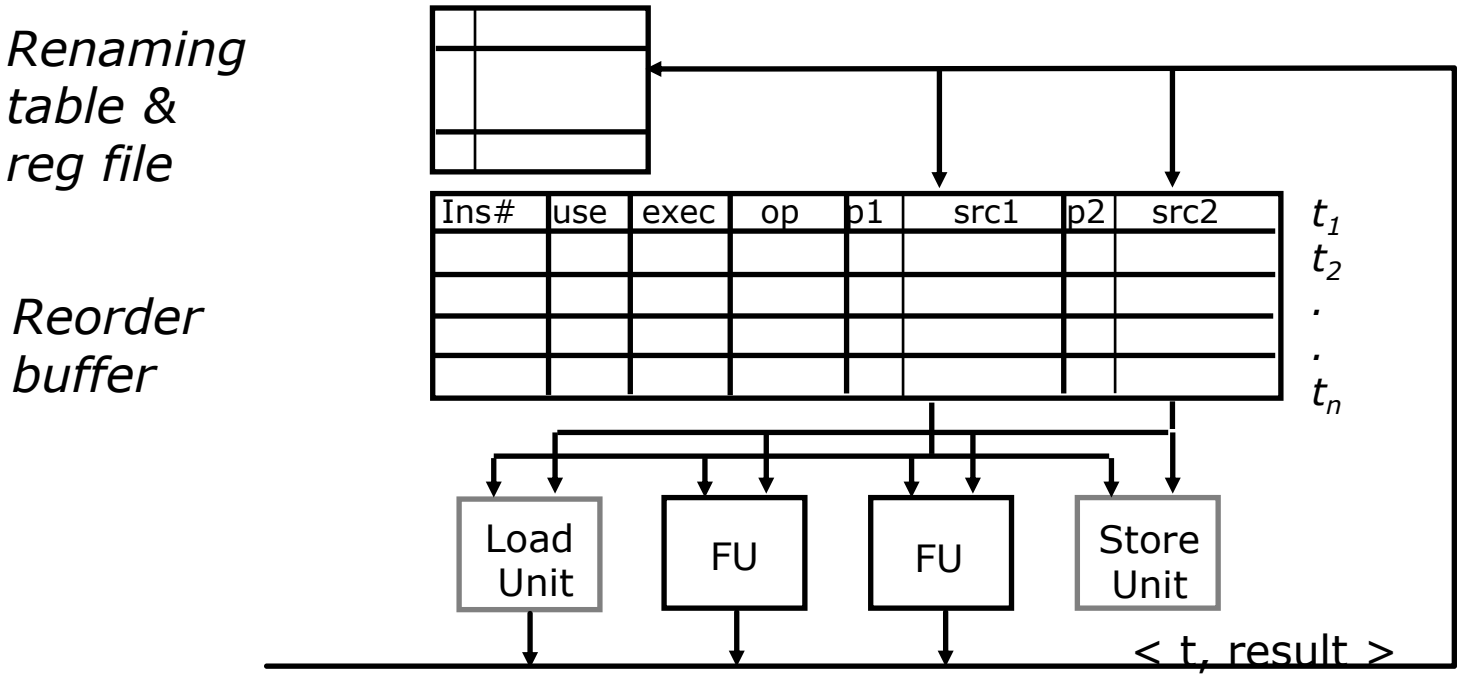
In-order execution machines:

- Guarantee no instruction issued after branch can write-back before branch resolves by keeping values in the pipeline
- Kill all values from all instructions in pipeline behind mispredicted branch

Out-of-order execution?

- Multiple instructions following branch in program order can generate new values before branch resolves

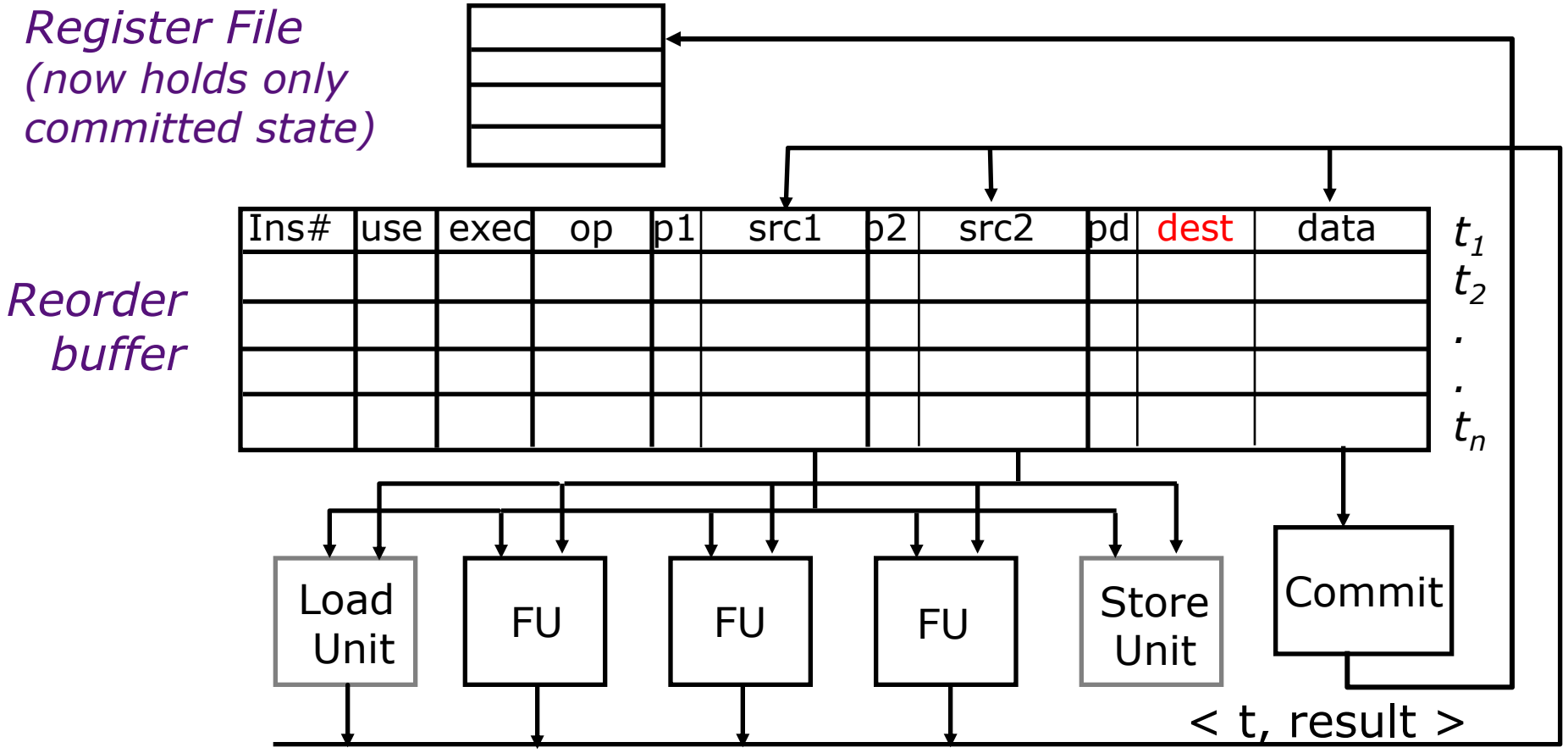
Data-Driven Execution



Basic Operation:

- Enter op and tag or data (if known) for each source
- Replace tag with data as it becomes available
- Issue instruction when all sources are available
- Save dest data when operation finishes

Rollback and Renaming

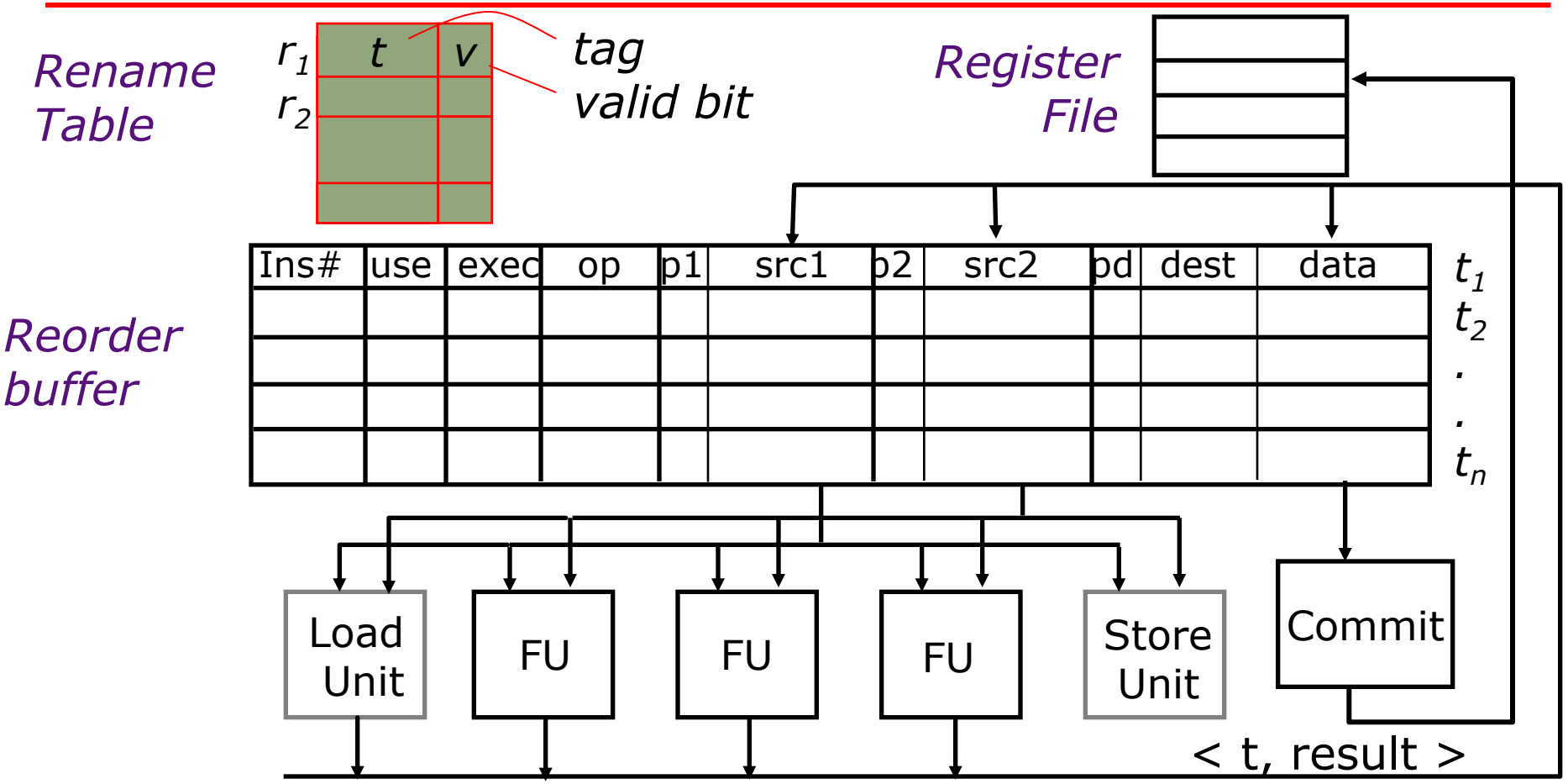


Register file does not contain renaming tags any more.

How does the decode stage find the tag of a source register?

Search the "dest" field in the reorder buffer

Renaming Table

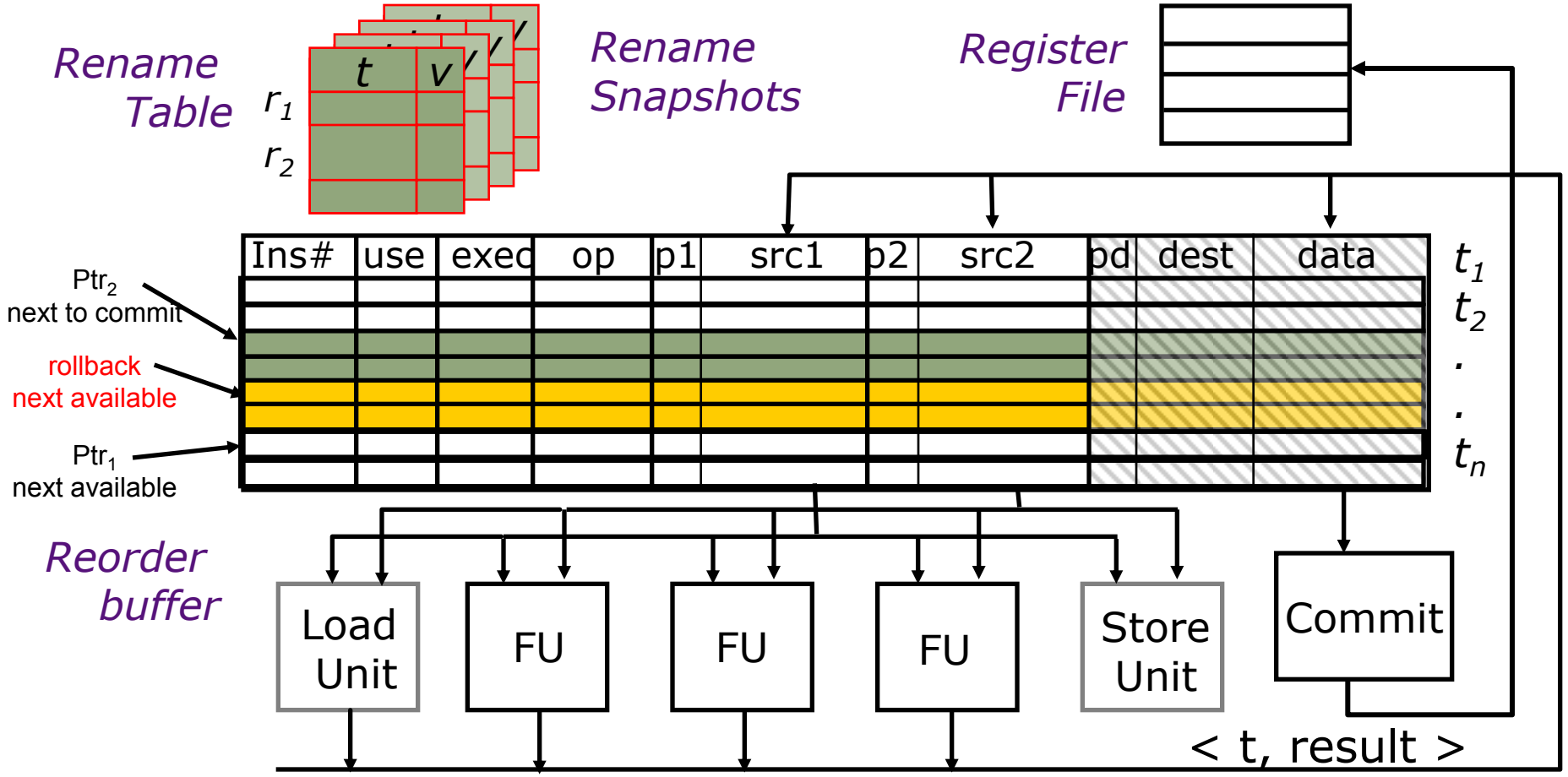


Renaming table is a cache to speed up register name look up.

Valid bits are cleared on exceptions and when else? *Branch mispredicts*

After being cleared, when can instructions be added to ROB? *After drain*

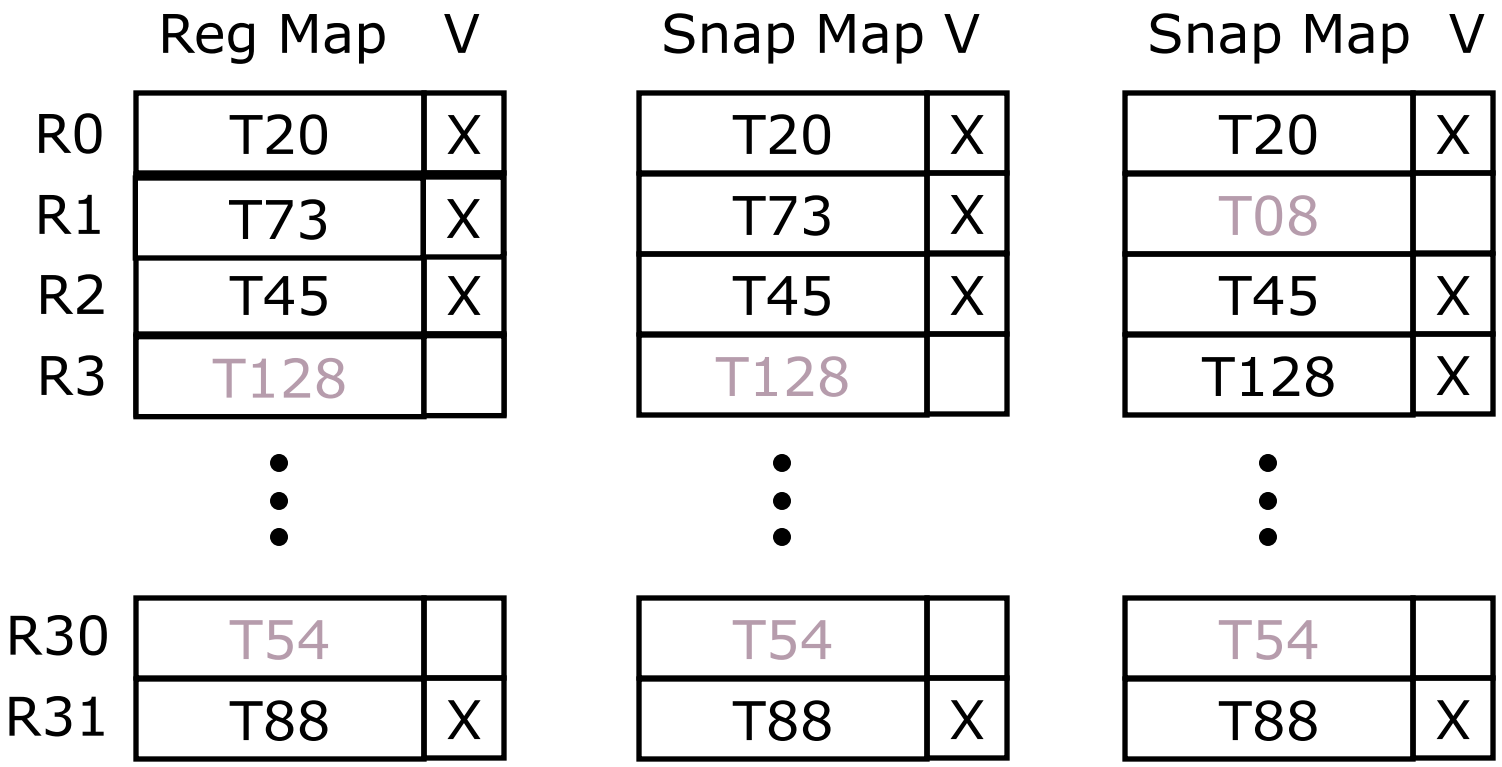
Recovering ROB/Renaming Table



Take snapshot of register rename table at each predicted branch, recover earlier snapshot if branch mispredicted

Map Table Recovery - Snapshots

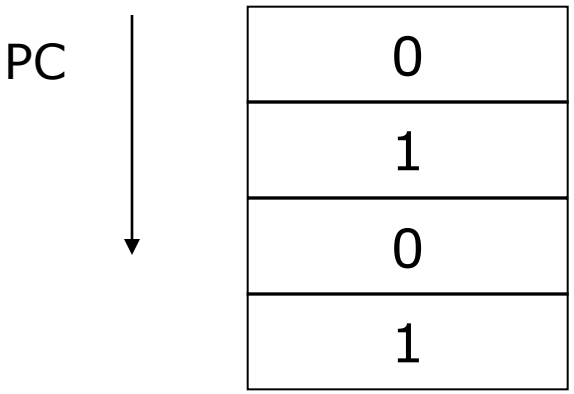
Speculative value management of microarchitectural state



What kind of value management is this? **Greedy!!**

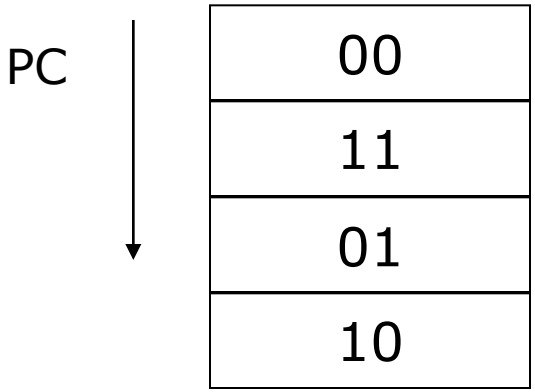
Branch Predictor Recovery

- 1-Bit Counter Recovery



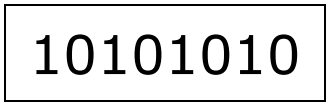
Lazy

- 2-Bit Counter Recovery



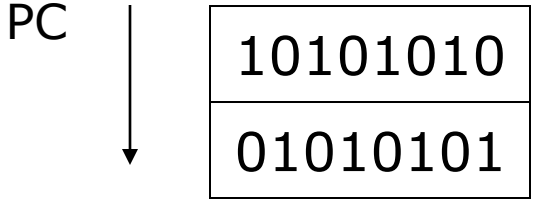
Lazy

- Global History Recovery



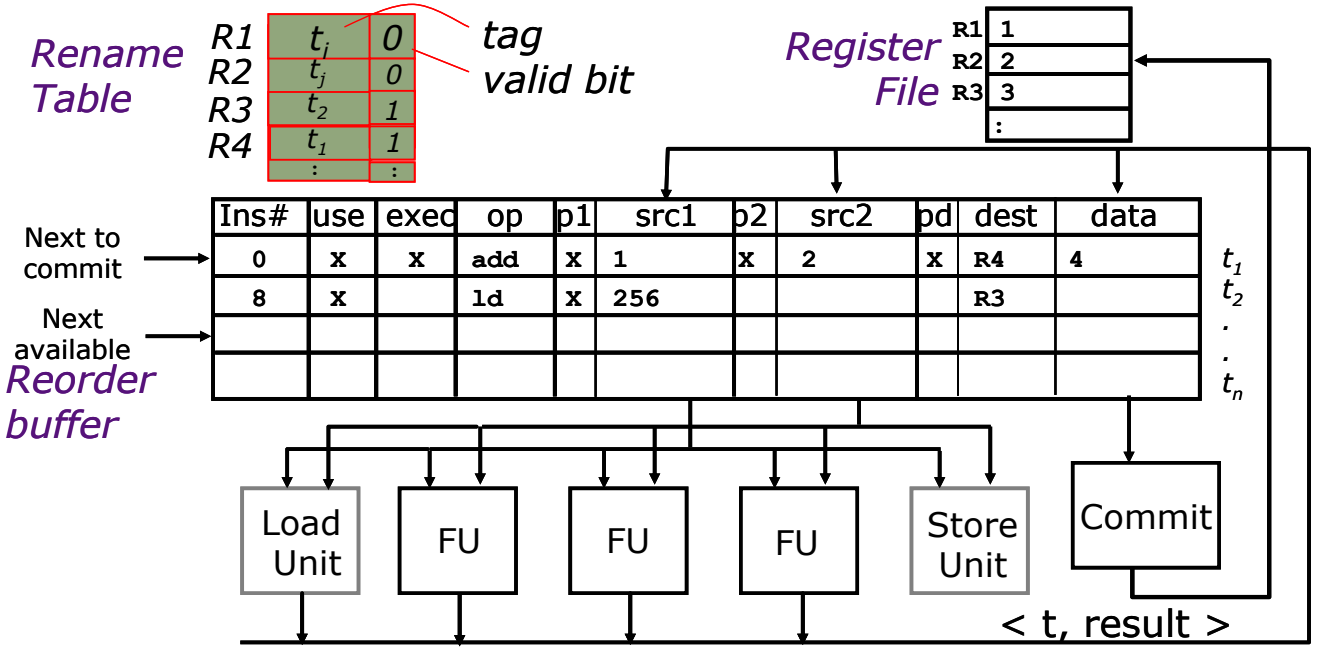
Greedy

- Local History Recovery



Greedy!!

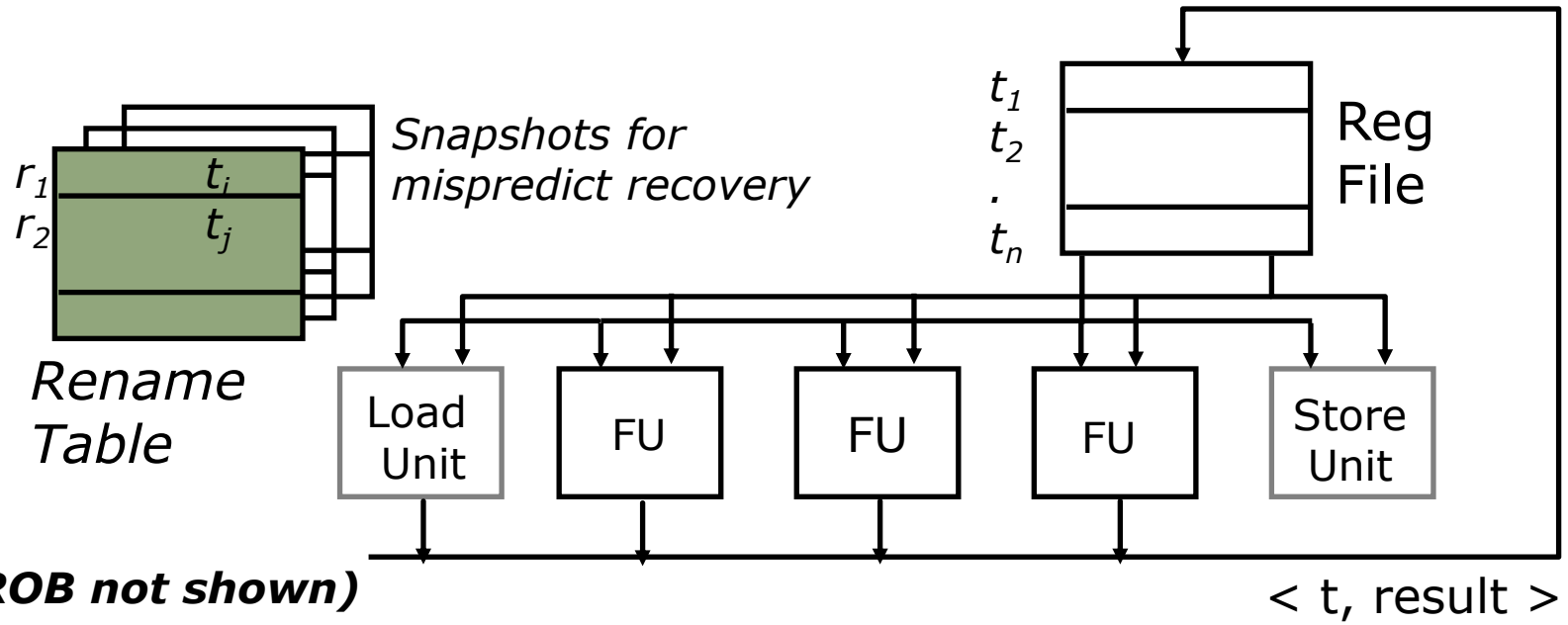
O-o-O Execution with ROB



Basic Operation:

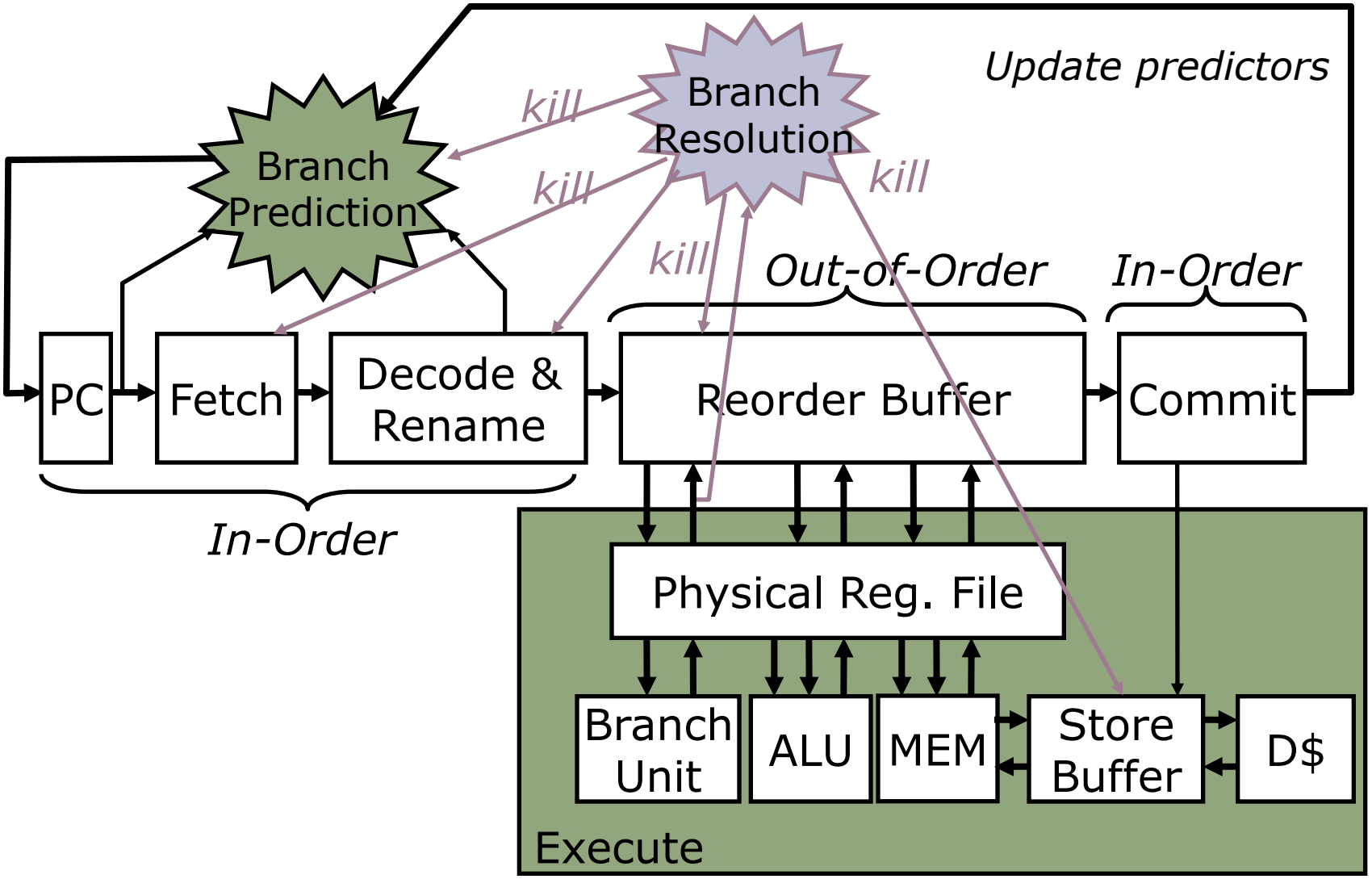
- Enter op and tag or data (if known) for each source
- Replace tag with data as it becomes available
- Issue instruction when all sources are available
- Save dest data when operation finishes
- Commit saved dest data when instruction commits

Unified Physical Register File
(MIPS R10K, Alpha 21264, Pentium 4)



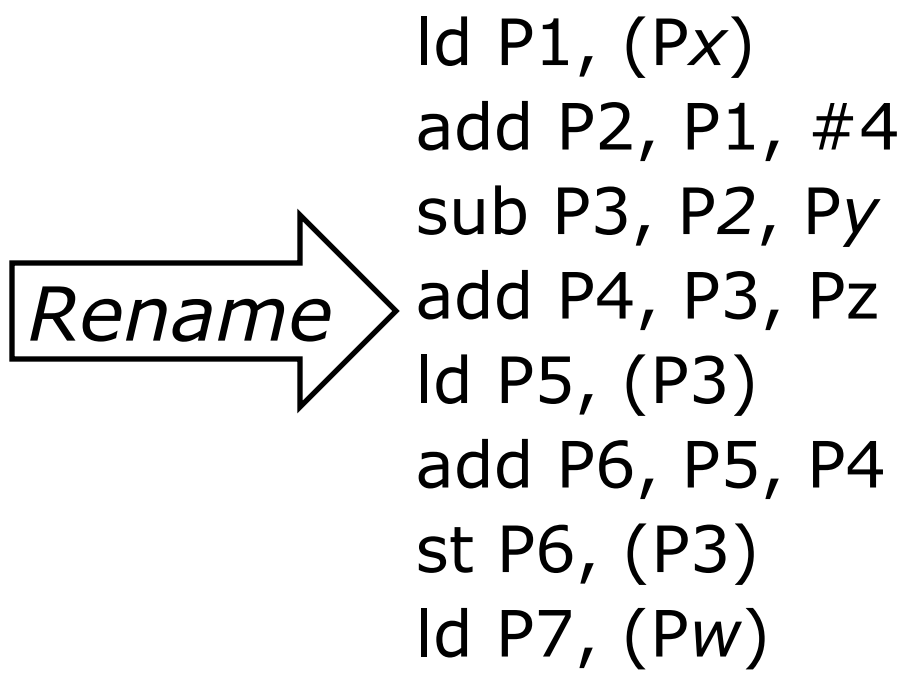
- One regfile for both *committed* and *speculative* values (no data in ROB)
- During decode, instruction result allocated new physical register, source regs translated to physical regs through rename table
- Instruction reads data from regfile at start of execute (not in decode)
- Write-back updates reg. busy bits on instructions in ROB (assoc. search)
- Snapshots of rename table taken at every branch to recover mispredicts
- On exception, renaming undone in reverse order of issue *(MIPS R10000)*

Speculative & Out-of-Order Execution



Lifetime of Physical Registers

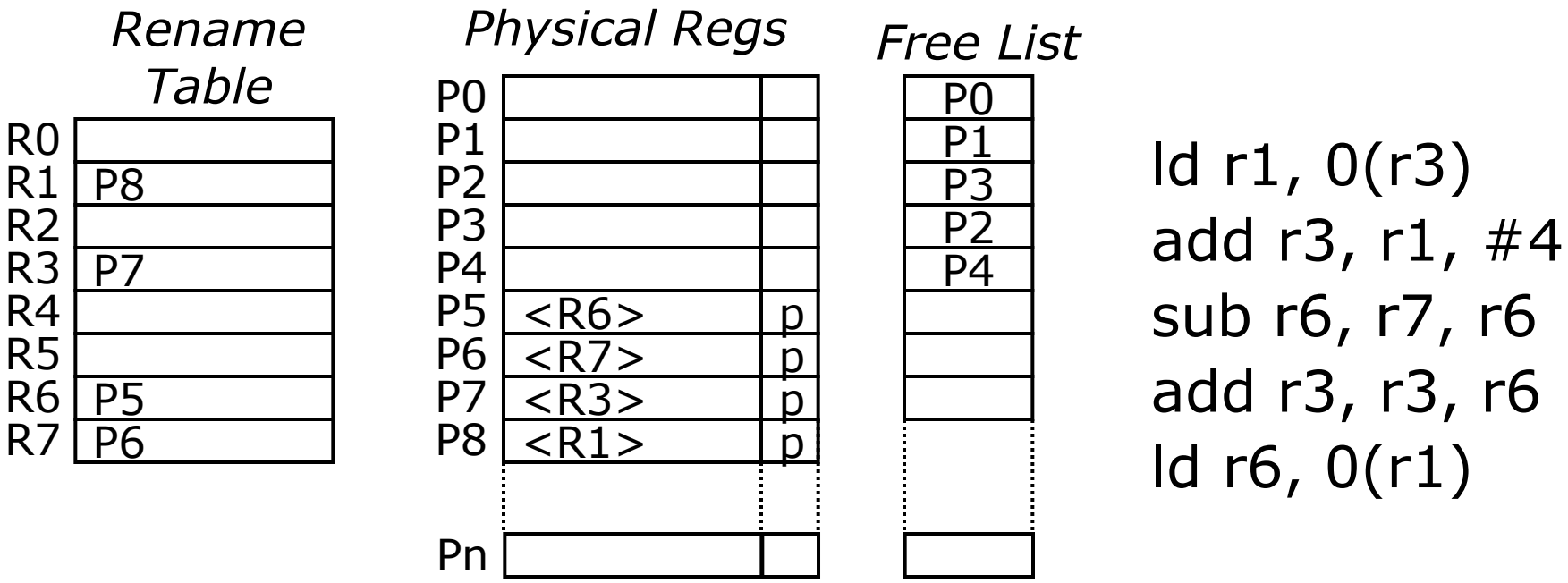
- Physical regfile holds committed and speculative values
- Physical registers decoupled from ROB entries *(no data in ROB)*



When can we reuse a physical register?

When next write to same architectural register commits

Physical Register Management



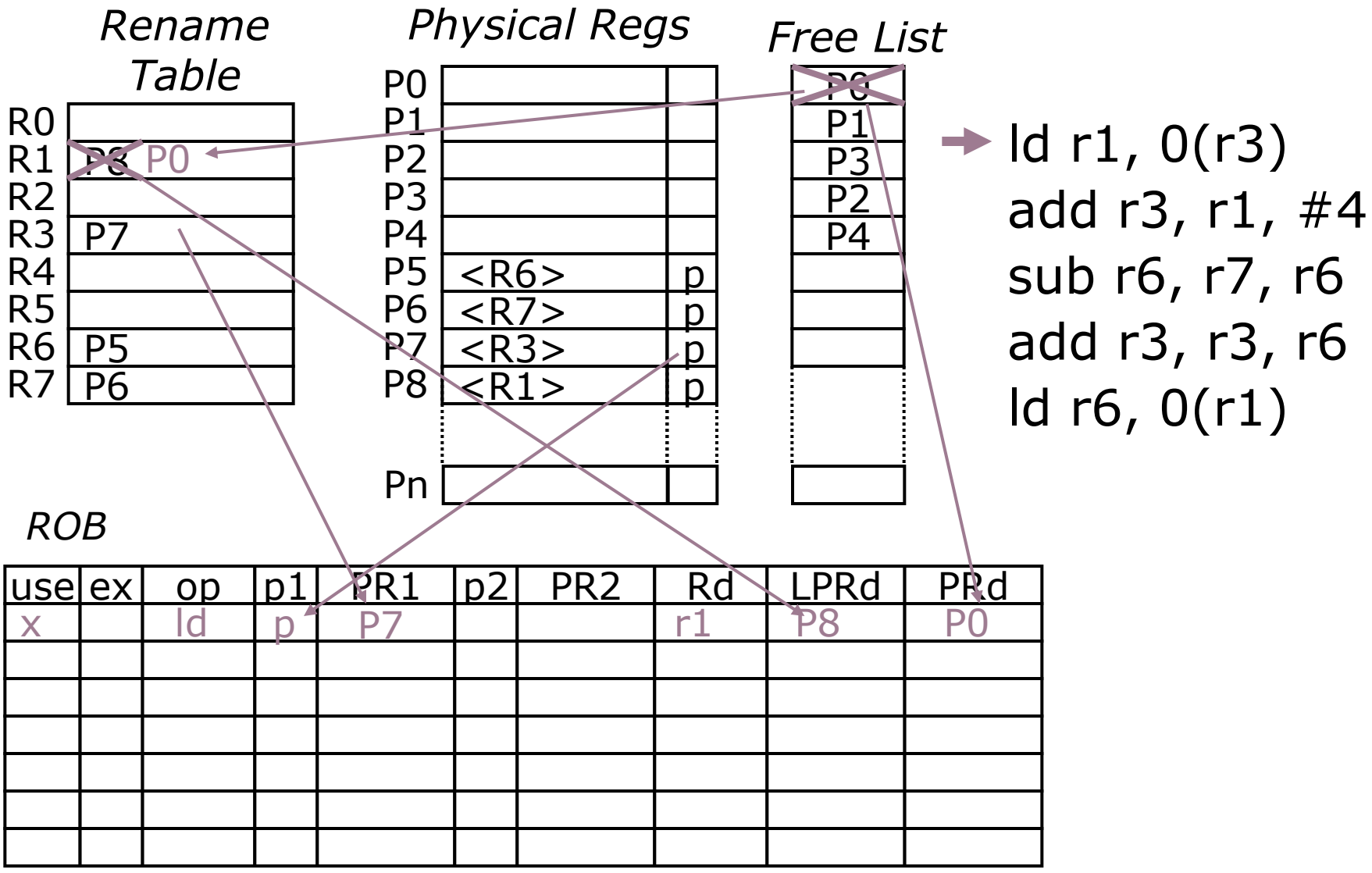

```
ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)
```

ROB

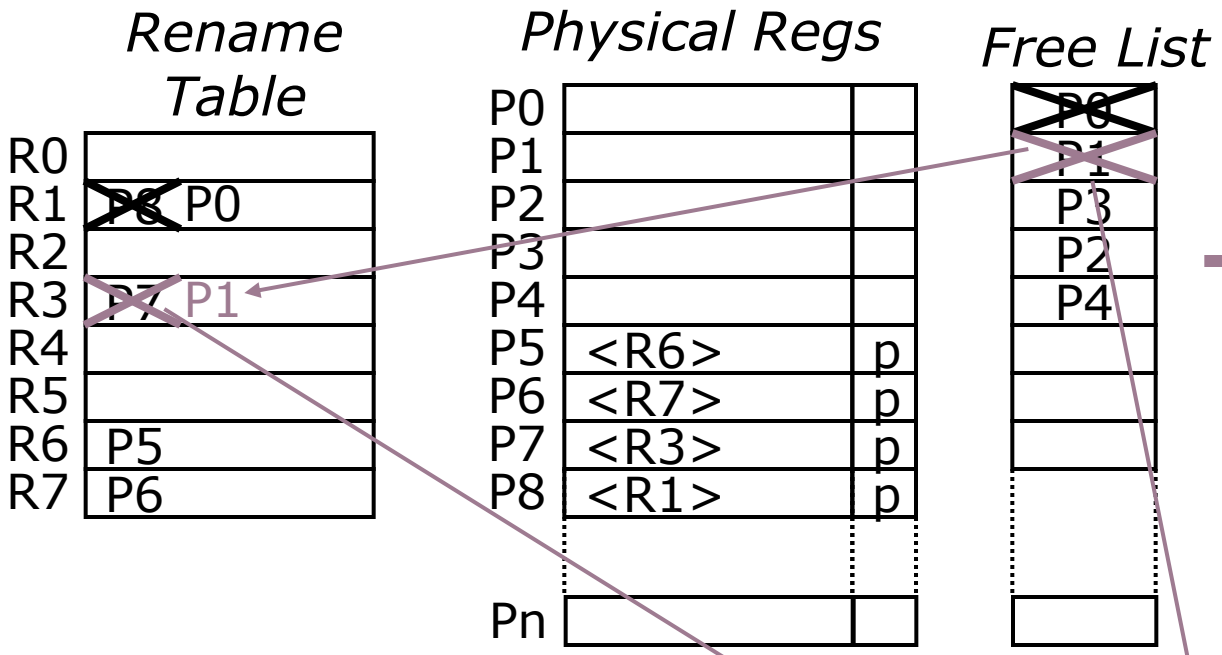
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd

(LPRd requires third read port on Rename Table for each instruction)

Physical Register Management



Physical Register Management

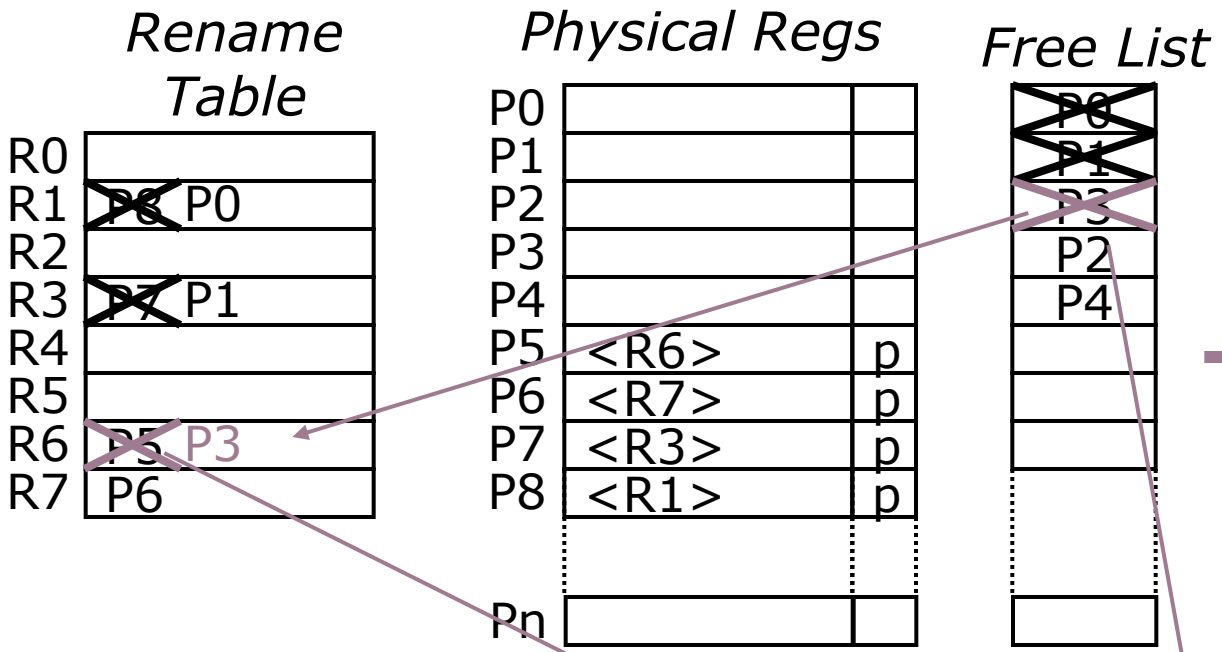


ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			r1	P8	P0
x		add		P0			r3	P7	P1

Physical Register Management

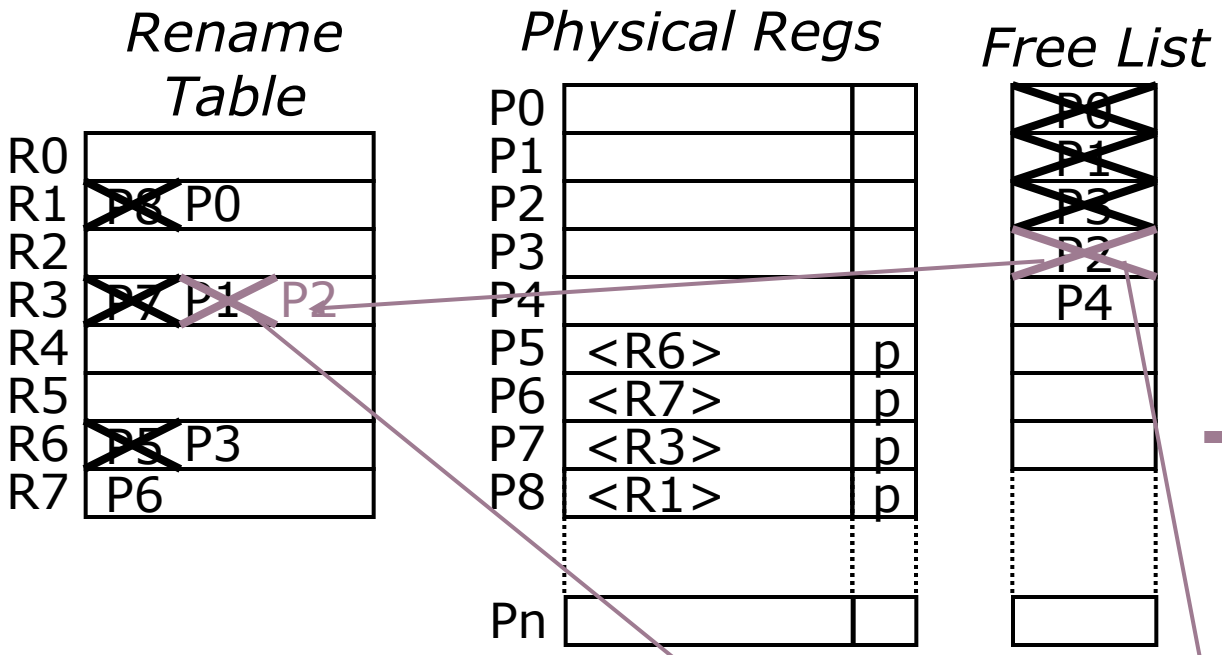



```
ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)
```

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			r1	P8	P0
x		add		P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3

Physical Register Management

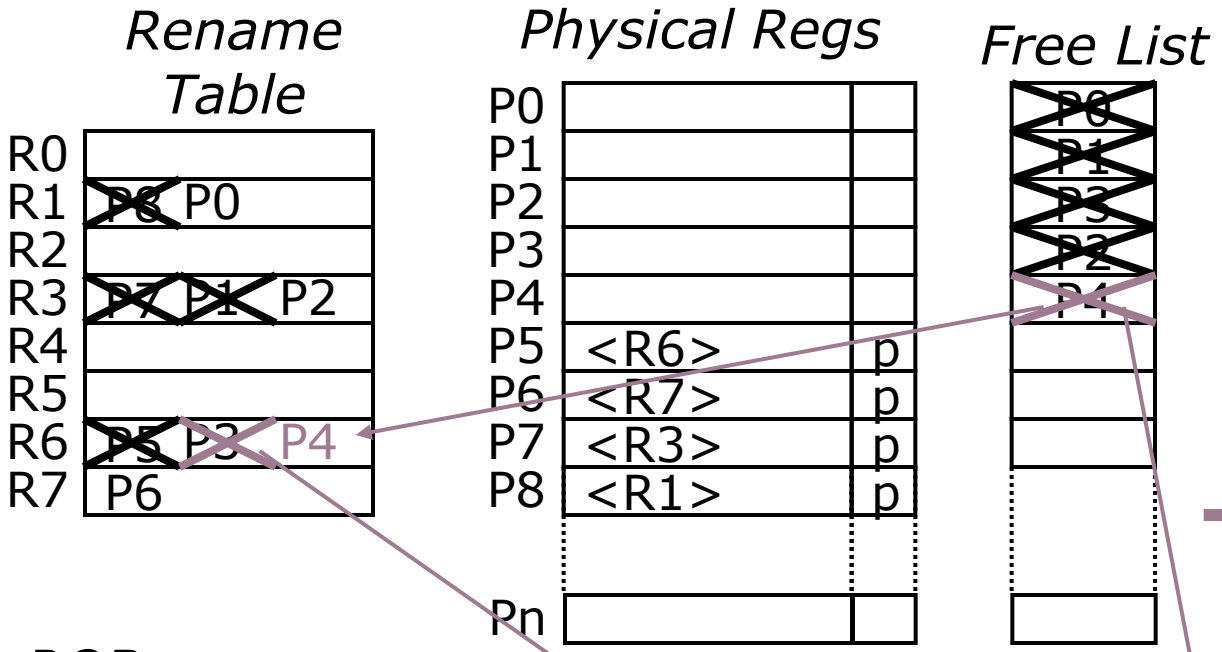


ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
→ add r3, r3, r6
ld r6, 0(r1)

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			r1	P8	P0
x		add		P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3
x		add		P1		P3	r3	P1	P2

Physical Register Management

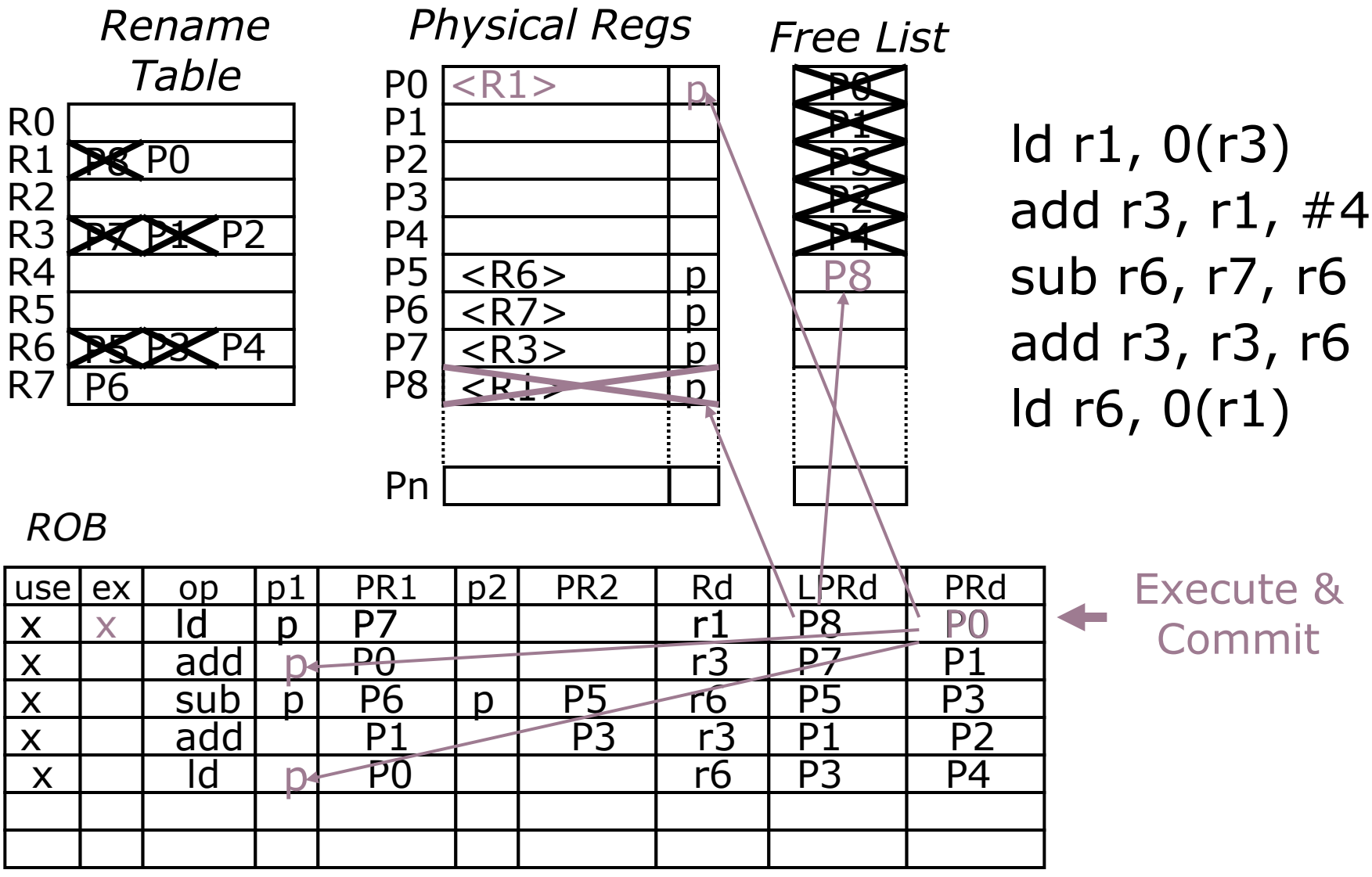


ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
→ ld r6, 0(r1)

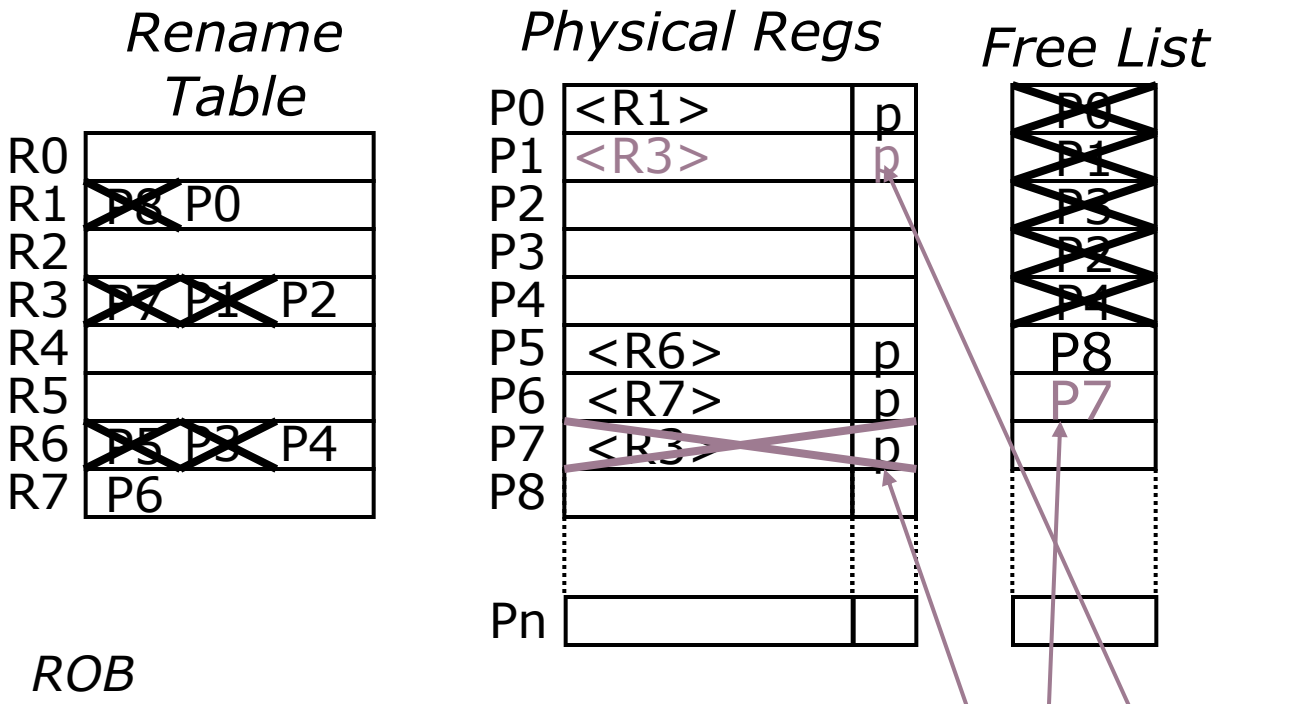
ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			r1	P8	P0
x		add		P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3
x		add		P1		P3	r3	P1	P2
x		ld		P0			r6	P3	P4

Physical Register Management



Physical Register Management



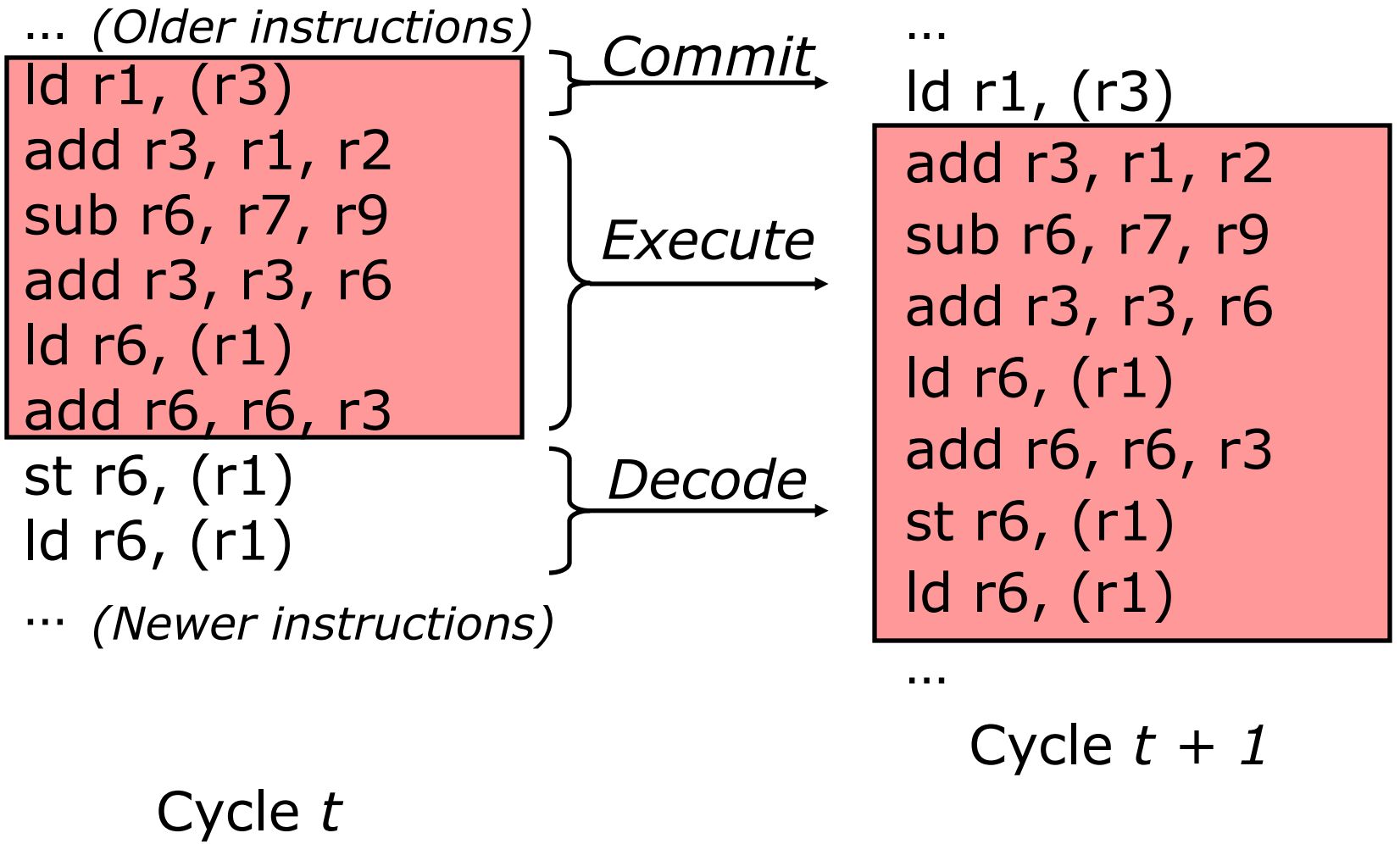

```
ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)
```

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x	x	ld	p	P7			r1	P8	P0
x	x	add	p	P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3
x		add	p	P1		P3	r3	P1	P2
x		ld	p	P0			r6	P3	P4

Execute & Commit

Reorder Buffer Holds Active Instruction Window



Issue Timing

i1	Add R1,R1,#1	$Issue_1$	$Execute_1$		
i2	Sub R1,R1,#1			$Issue_2$	$Execute_2$

How can we issue earlier?

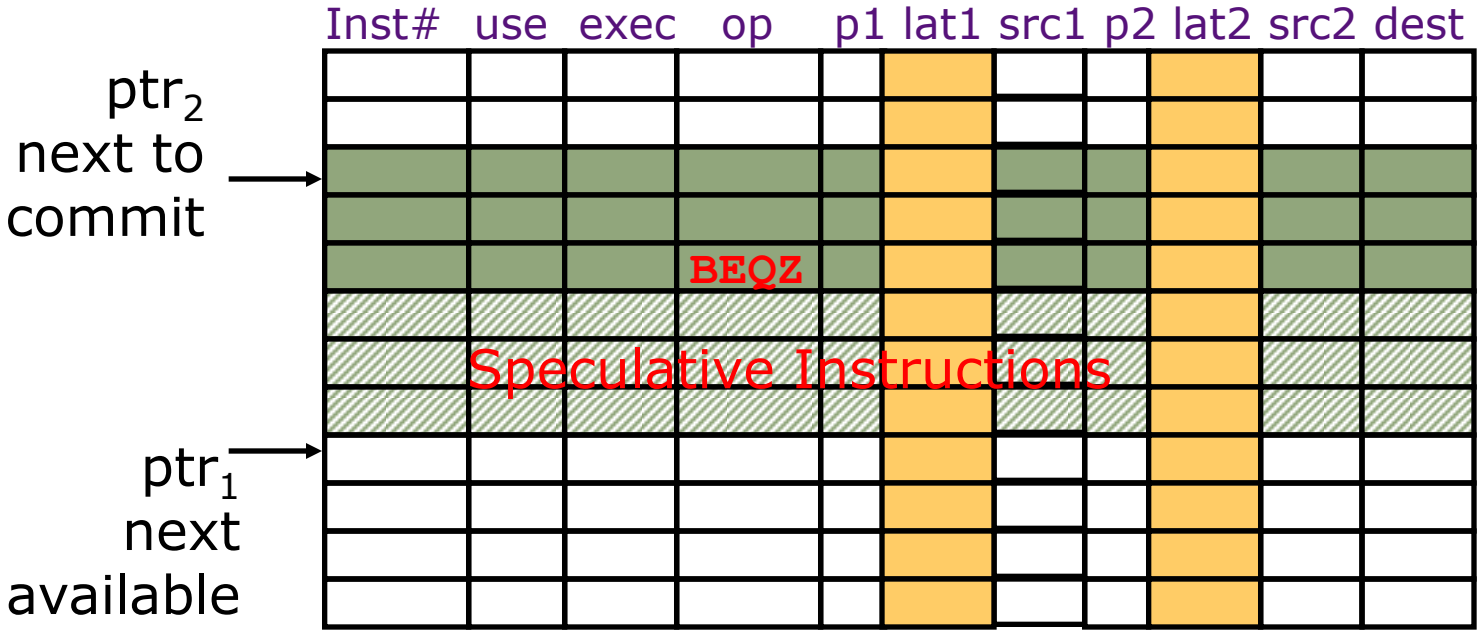
Using knowledge of execution latency (bypass)

i1	LD R1, (R3)	$Issue_1$	$Execute_1$		
i2	Sub R1,R1,#1		$Issue_2$	$Execute_2$	

What might make this schedule fail?

If execution latency wasn't as expected

Issue Queue with latency prediction

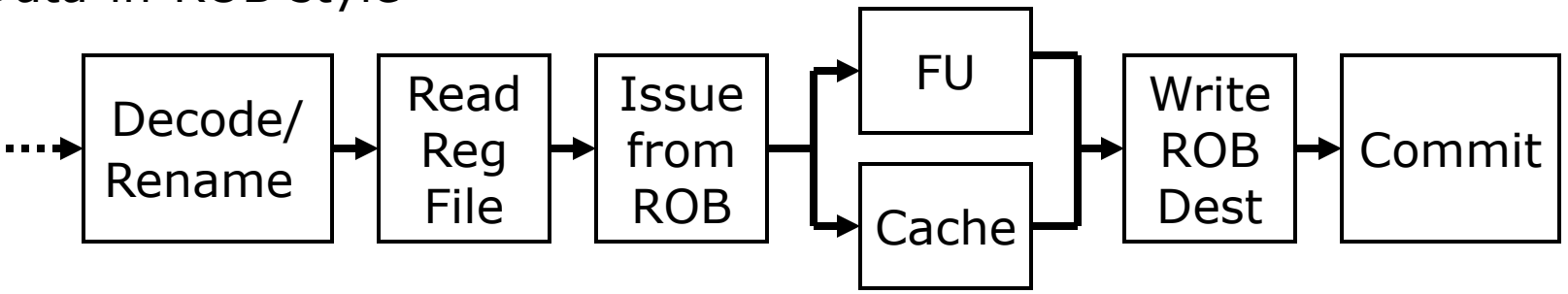


Issue Queue (Reorder buffer)

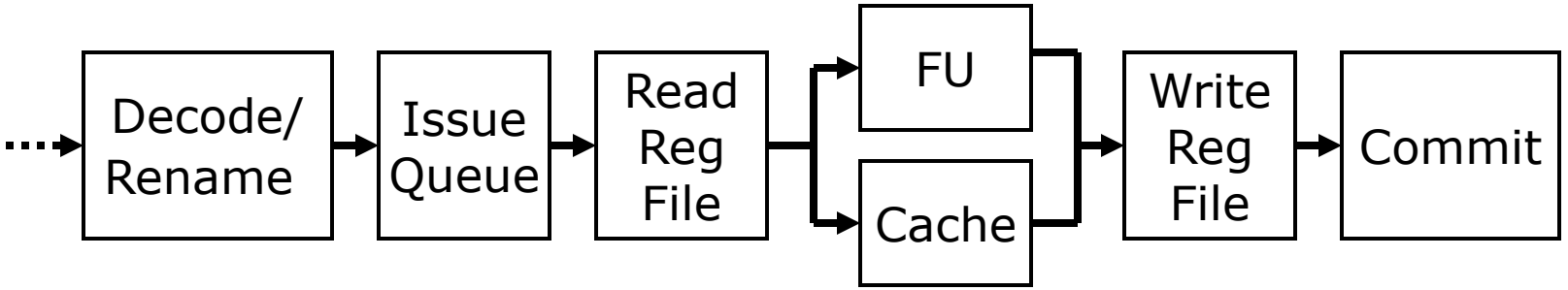
- Fixed latency: latency included in queue entry ('bypassed')
- Predicted latency: latency included in queue entry (speculated)
- Variable latency: wait for completion signal (stall)

Data-in-ROB vs. Single Register File

Data-in-ROB style



Single-register-file style

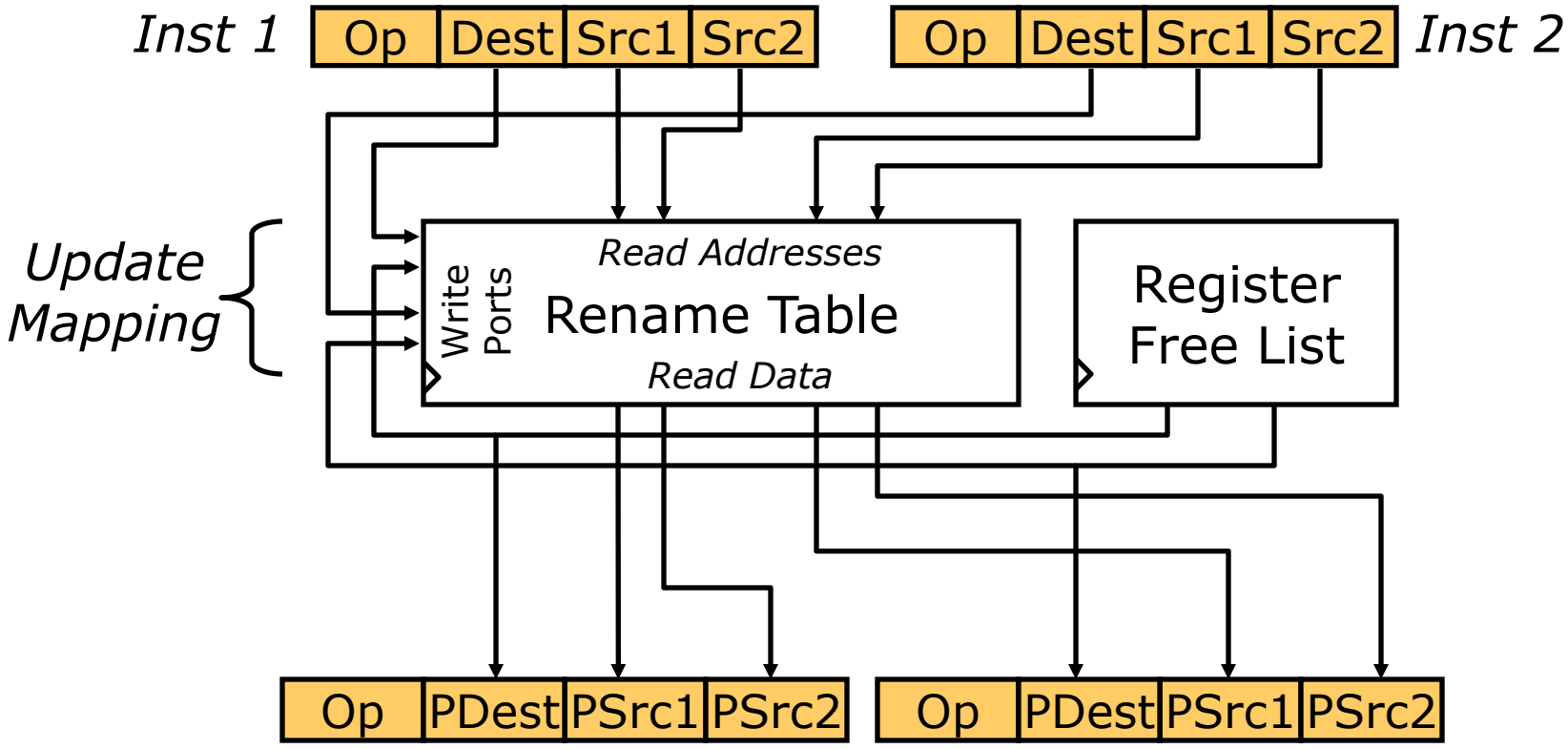


How does issue speculation differ, e.g., on cache miss?

Dependency loop shorter for data-in-ROB style

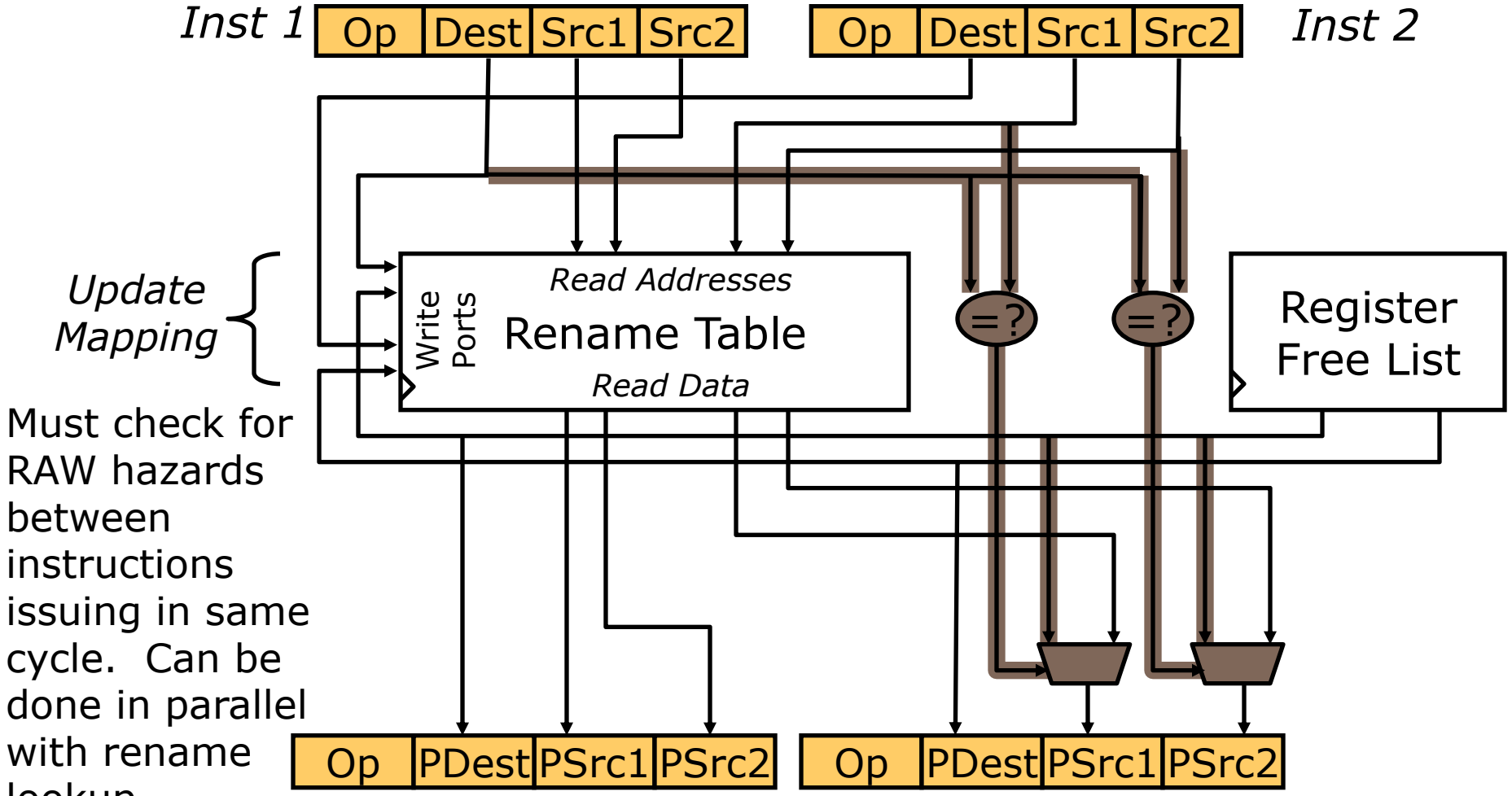
Superscalar Register Renaming

- During decode, instructions allocated new physical destination register
- Source operands renamed to physical register with newest value
- Execution unit only sees physical register numbers



Does this work?

Superscalar Register Renaming



Must check for RAW hazards between instructions issuing in same cycle. Can be done in parallel with rename lookup.

MIPS R10K renames 4 serially-RAW-dependent insts/cycle)

Split Issue and Commit Queues

- How large should the ROB be?
  - Think Little's Law…
- Can split ROB into issue and commit queues

Issue Queue

use	op	p1	PR1	p2	PR2	tag

Commit Queue

ex	Rd	LPRd	PRd

- Commit queue: Allocate on decode, free on commit
- Issue queue: Allocate on decode, free on dispatch
- Pros: Smaller issue queue → simpler dispatch logic
- Cons: More complex mis-speculation recovery

Speculating Both Directions

An alternative to branch prediction is to execute both directions of a branch *speculatively*

- resource requirement is proportional to the number of concurrent speculative executions
- only half the resources engage in useful work when both directions of a branch are executed speculatively
- branch prediction takes less resources than speculative execution of both paths

With accurate branch prediction, it is more cost effective to dedicate all resources to the predicted direction

Thank you !

http://www.csg.csail.mit.edu/6.823