CS 152 Computer Architecture and Engineering
CS252 Graduate Computer Architecture

Lecture 11 – Out-of-Order Execution

Krste Asanovic

Electrical Engineering and Computer Sciences
University of California at Berkeley

`http://www.eecs.berkeley.edu/~krste`
`http://inst.eecs.berkeley.edu/~cs152`
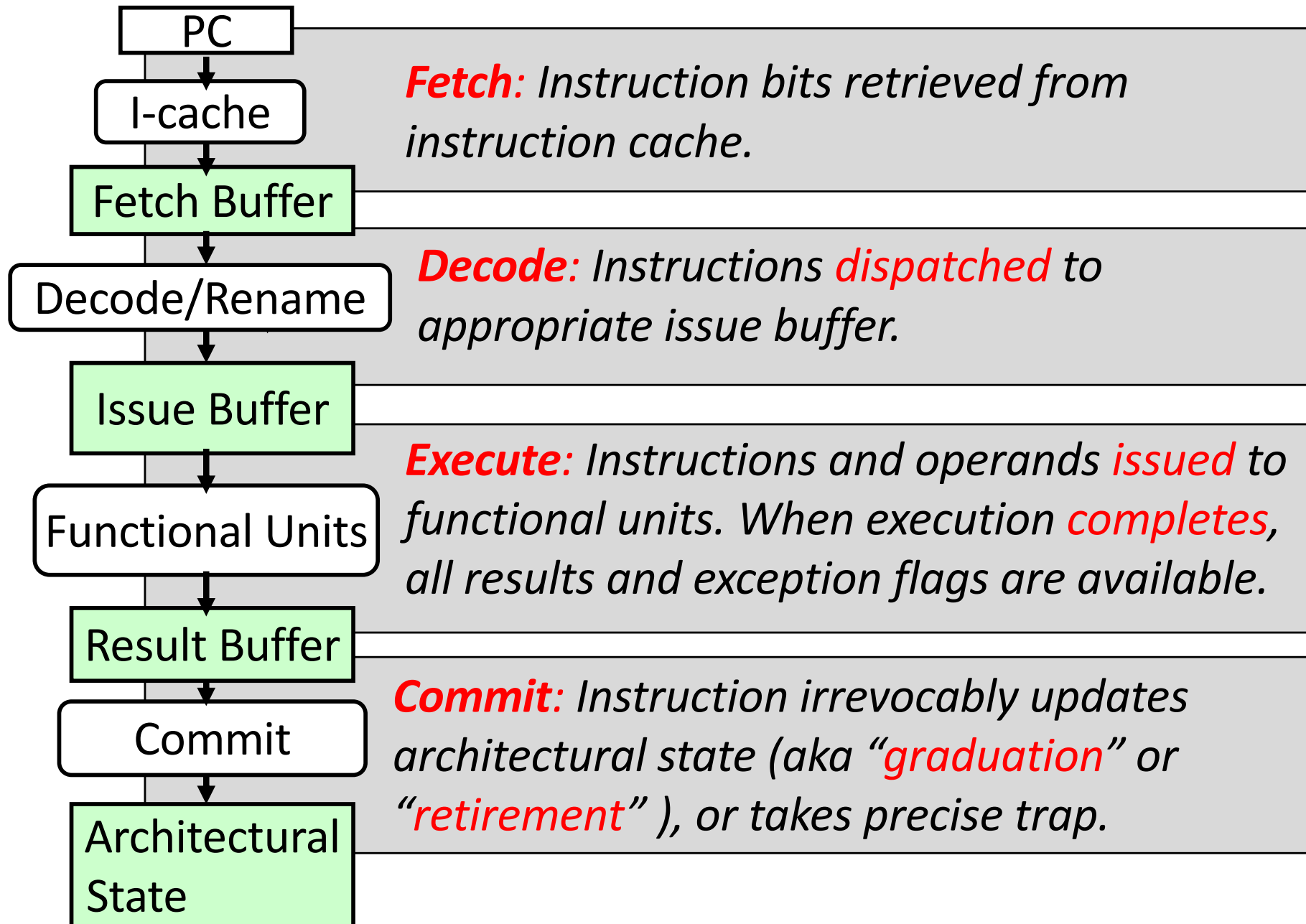
Last Time in Lecture 10

- Pipelining is complicated by multiple and/or variable latency functional units

- Out-of-order and/or pipelined execution requires tracking of dependencies (RAW, WAR, WAW)

- OoO issue limited by WAR and WAW hazards caused by reuse of architectural register names, removed by register renaming

- OoO issue and register renaming invented in mid-1960s but disappeared in practice until 1990s, as simpler architecture approaches (pipelining, caches) could more easily take advantage of technology scaling

- Also, two important problems had to be solved:
  - Control hazards
  - Precise traps and interrupts

2

Recap: In-Order Commit for Precise Traps
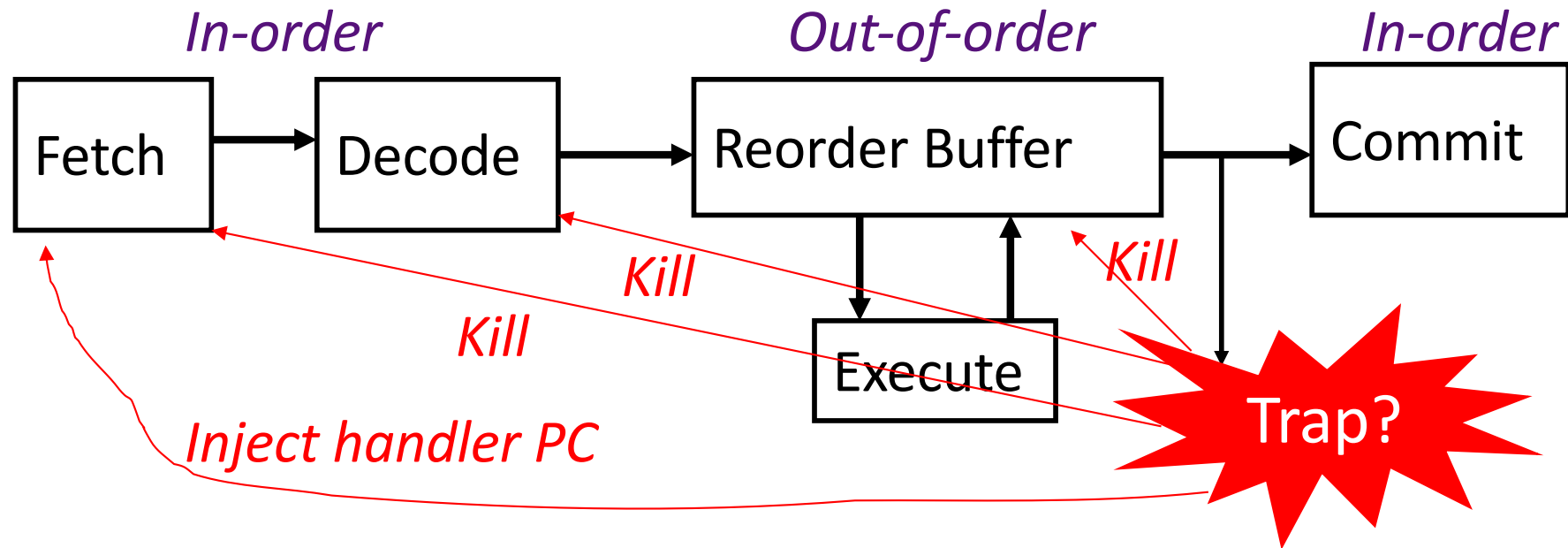
- In-order instruction fetch and decode, and dispatch to reservation stations inside reorder buffer

- Instructions issue from reservation stations out-of-order

- Out-of-order completion, values stored in temporary buffers

- Commit is in-order, checks for traps, and if none updates architectural state

Phases of Instruction Execution

PC → I-cache → Fetch Buffer → Decode/Rename → Issue Buffer → Functional Units → Result Buffer → Commit → Architectural State

Fetch*: Instruction bits retrieved from instruction cache.*

Decode*: Instructions dispatched to appropriate issue buffer.*

Execute*: Instructions and operands issued to functional units. When execution completes, all results and exception flags are available.*

Commit*: Instruction irrevocably updates architectural state (aka "graduation" or "retirement"), or takes precise trap.*

4

In-Order versus Out-of-Order Phases

- **Instruction fetch/decode/rename always in-order**
  - Need to parse ISA sequentially to get correct semantics
  - *CS252:Proposals for speculative OoO instruction fetch, e.g., Multiscalar. Predict control flow and data dependencies across sequential program segments fetched/decoded/executed in parallel, fixup if prediction wrong*

- **Dispatch (place instruction into machine buffers to wait for issue) also always in-order**
  - Some use "Dispatch" to mean "Issue", but not in these lectures

In-Order Versus Out-of-Order Issue

- In-order (InO) issue:

    - Issue stalls on RAW dependencies or structural hazards, or possibly WAR/WAW hazards

    - Instruction cannot issue to execution units unless all preceding instructions have issued to execution units

- Out-of-order (OoO) issue:

    - Instructions dispatched in program order to *reservation stations (or other forms of instruction buffer)* to wait for operands to arrive, or other hazards to clear

    - While earlier instructions wait in issue buffers, following instructions can be dispatched and issued out-of-order

In-Order versus Out-of-Order Completion

- All but simplest machines have out-of-order completion, due to different latencies of functional units and desire to bypass values as soon as available

- Classic RISC 5-stage integer pipeline just barely has in-order completion

  - Load takes two cycles, but following one-cycle integer op completes at same time, not earlier

  - Adding pipelined FPU immediately brings OoO completion

In-Order versus Out-of-Order Commit

- In-order commit supports precise traps, standard today
  - *CS252: Some proposals to reduce the cost of in-order commit by retiring some instructions early to compact reorder buffer, but this is just an optimized in-order commit*

- Out-of-order commit was effectively what early OoO machines implemented (imprecise traps) as completion irrevocably changed machine state
  - i.e., complete == commit in these machines

OoO Design Choices

- Where are reservation stations?
  - Part of reorder buffer, or in separate issue window?
  - Distributed by functional units, or centralized?

- How is register renaming performed?
  - Tags and data held in reservation stations, with separate architectural register file
  - Tags only in reservation stations, data held in unified physical register file

"Data-in-ROB" Design
(HP PA8000, Pentium Pro, Core2Duo, Nehalem)

Oldest →

Free →

v	i	Opcode	p	Tag	Src1	p	Tag	Src2	p	Reg	Result	Except?
v	i	Opcode	p	Tag	Src1	p	Tag	Src2	p	Reg	Result	Except?
v	i	Opcode	p	Tag	Src1	p	Tag	Src2	p	Reg	Result	Except?
v	i	Opcode	p	Tag	Src1	p	Tag	Src2	p	Reg	Result	Except?
v	i	Opcode	p	Tag	Src1	p	Tag	Src2	p	Reg	Result	Except?

- Managed as circular buffer in program order, new instructions dispatched to free slots, oldest instruction committed/reclaimed when done ("p" bit set on result)

- Tag is given by index in ROB (Free pointer value)

- In dispatch, non-busy source operands read from architectural register file and copied to Src1 and Src2 with presence bit "p" set. Busy operands copy tag of producer and clear "p" bit.

- Set valid bit "v" on dispatch, set issued bit "i" on issue

- On completion, search source tags, set "p" bit and copy data into src on tag match. Write result and exception flags to ROB.

- On commit, check exception status, and copy result into architectural register file if no trap.

- On trap, flush machine and ROB, set free=oldest, jump to handler

10

Managing Rename for Data-in-ROB

Rename table associated with architectural registers, managed in decode/dispatch

p	Tag	Value
p	Tag	Value
p	Tag	Value
p	Tag	Value

One entry per arch. register

- If "p" bit set, then use value in architectural register file
- Else, tag field indicates instruction that will/has produced value
- For dispatch, read source operands <p,tag,value> from arch. regfile, then also read <p,result> from producing instruction in ROB at tag index, bypassing as needed. Copy operands to ROB.
- Write destination arch. register entry with <0,Free,_>, to assign tag to ROB index of this instruction
- On commit, update arch. regfile with <1,_,Result> if tag matches, otherwise update with <0,_,Result>.   (Tag value is not updated)
- On trap, reset table (All p=1)

Data Movement in Data-in-ROB Design

Architectural Register File

Write results at commit

Read operands during decode

Read results for commit

Write sources in dispatch

Source Operands

Result Data

Bypass newer values at dispatch

ROB

Read operands at issue

Write results at completion

Functional Units

12
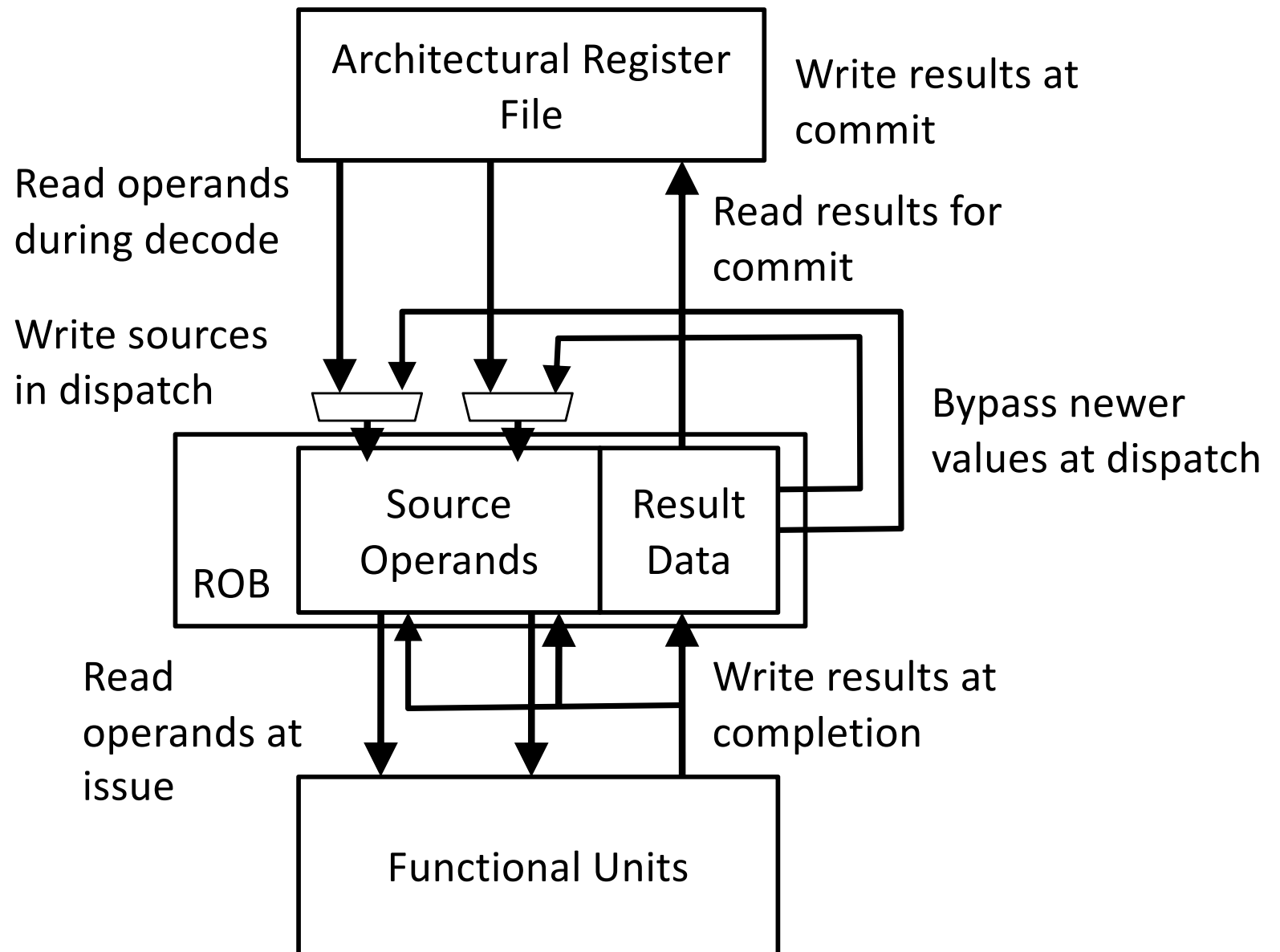
CS152 Administrivia

- PS 3 out today, due Monday March 16

- Lab 2 due Monday March 8
  - Lab 2 takes time to run, so please get started ASAP
  - Don't wait till Sunday!
  - Can share directed simulation results but must do own analysis.
  - Look at pinned README on Piazza

- Midterms being graded, within a week
  - Regrade requests up to one week after exams returned

- Please respond to survey of midterm technical issues
  - This is tied to your email address

- Please also respond to course progress survey
  - Please give feedback on course so far
  - This survey is anonymous

CS252 Administrivia

- Project proposal presentations March 4 (tomorrow) in discussion section
  - 10-minute per project including Q&A with rest of class
  - Schedule on Piazza

Unified Physical Register File
(MIPS R10K, Alpha 21264, Intel Pentium 4 & Sandy/Ivy Bridge)

- Rename all architectural registers into a single *physical* register file during decode, no register values read

- Functional units read and write from single unified register file holding committed and temporary registers in execute

- Commit only updates mapping of architectural register to physical register, no data movement

Decode Stage Register Mapping		Unified Physical Register File		Committed Register Mapping

Read operands at issue   Write results at completion

Functional Units
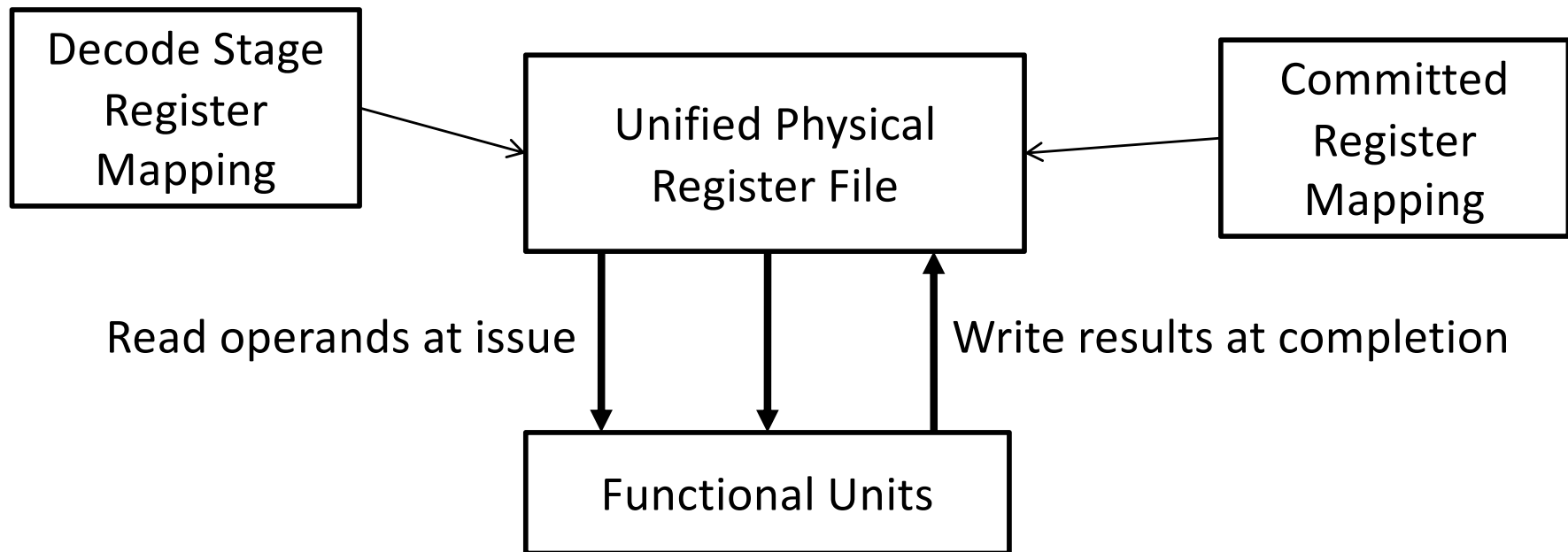
Lifetime of Physical Registers

- Physical regfile holds committed and speculative values
- Physical registers decoupled from ROB entries *(no data in ROB)*
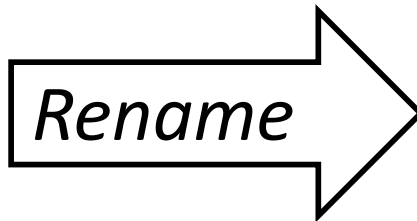
```
ld x1, (x3)                    ld P1, (Px)
addi x3, x1, #4                addi P2, P1, #4
sub x6, x7, x9                 sub P3, Py, Pz
add x3, x3, x6     Rename →    add P4, P2, P3
ld x6, (x1)                    ld P5, (P1)
add x6, x6, x3                 add P6, P5, P4
sd x6, (x1)                    sd P6, (P1)
ld x6, (x11)                   ld P7, (Pw)
```

When can we reuse a physical register?
When next writer of same architectural register commits

16

Physical Register Management

Rename Table

x0	
x1	P8
x2	
x3	P7
x4	
x5	
x6	P5
x7	P6

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5	<x6>	p
P6	<x7>	p
P7	<x3>	p
P8	<x1>	p
...		
Pn		

Free List

P0
P1
P3
P2
P4
...

ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)

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

Rename Table

x0	
x1	P8 P0
x2	
x3	P7
x4	
x5	
x6	P5
x7	P6

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5	<x6>	p
P6	<x7>	p
P7	<x3>	p
P8	<x1>	p
Pn		

Free List

P0
P1
P3
P2
P4

ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0

Physical Register Management

Rename Table

x0	
x1	~~P8~~ P0
x2	
x3	~~P7~~ P1
x4	
x5	
x6	P5
x7	P6

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5	<x6>	p
P6	<x7>	p
P7	<x3>	p
P8	<R1>	p
Pn		

Free List

~~P0~~
~~P1~~
P3
P2
P4

ld x1, 0(x3)
→ addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1

Physical Register Management

Rename Table

x0		
x1	~~P8~~	P0
x2		
x3	~~P7~~	P1
x4		
x5		
x6	~~P5~~	P3
x7	P6	

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5	\<x6\>	p
P6	\<x7\>	p
P7	\<x3\>	p
P8	\<R1\>	p
⋮		
Pn		

Free List

~~P0~~
~~P1~~
~~P3~~
P2
P4

ld x1, 0(x3)
addi x3, x1, #4
→ sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3

Physical Register Management

Rename Table

x0	
x1	~~P8~~ P0
x2	
x3	~~P7~~ ~~P1~~ P2
x4	
x5	
x6	~~P5~~ P3
x7	P6

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5	\<x6\>	p
P6	\<x7\>	p
P7	\<x3\>	p
P8	\<x1\>	p
Pn		

Free List

~~P0~~
~~P1~~
~~P3~~
~~P2~~
P4

ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
→ add x3, x3, x6
ld x6, 0(x1)

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add		P1		P3	x3	P1	P2

Physical Register Management

Rename Table

x0	
x1	~~P8~~ P0
x2	
x3	~~P7~~ ~~P1~~ P2
x4	
x5	
x6	~~P5~~ ~~P3~~ P4
x7	P6

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5	\<x6\>	p
P6	\<x7\>	p
P7	\<x3\>	p
P8	\<x1\>	p
Pn		

Free List

~~P0~~
~~P1~~
~~P3~~
~~P2~~
~~P4~~

ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
→ ld x6, 0(x1)

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add		P1		P3	x3	P1	P2
x		ld		P0			x6	P3	P4

Physical Register Management

Rename Table

x0	
x1	~~P8~~ P0
x2	
x3	~~P7~~ ~~P1~~ P2
x4	
x5	
x6	~~P5~~ ~~P3~~ P4
x7	P6

Physical Regs

P0	<x1>	p
P1		
P2		
P3		
P4		
P5	<x6>	p
P6	<x7>	p
P7	<x3>	p
P8	<x1>	p
Pn		

Free List

~~P0~~
~~P1~~
~~P3~~
~~P2~~
~~P4~~
P8

```
ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)
```

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x	x	ld	p	P7			x1	P8	P0
x		addi	p	P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add		P1		P3	x3	P1	P2
x		ld	p	P0			x6	P3	P4

Execute & Commit

23

Physical Register Management

Rename Table

x0		
x1	P8 P0	
x2		
x3	P7 P1 P2	
x4		
x5		
x6	P5 P3 P4	
x7	P6	

Physical Regs

P0	<x1>	p
P1	<x3>	p
P2		
P3		
P4		
P5	<x6>	p
P6	<x7>	p
P7	<x3>	p
P8		
⋮		
Pn		

Free List

P0
P1
P3
P2
P4
P8
P7
⋮

ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x	x	ld	p	P7			x1	P8	P0
x	x	addi	p	P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add	p	P1		P3	x3	P1	P2
x		ld	p	P0			x6	P3	P4

Execute & Commit

24

Repairing Rename at Traps

- MIPS R10K rename table is repaired by unrenaming instructions in reverse order using the PRd/LPRd fields

- Alpha 21264 had similar physical register file scheme, but kept complete rename table snapshots for each instruction in ROB (80 snapshots total)
  - Flash copy all bits from snapshot to active table in one cycle

Reorder Buffer Holds Active Instructions (Decoded but not Committed)

26
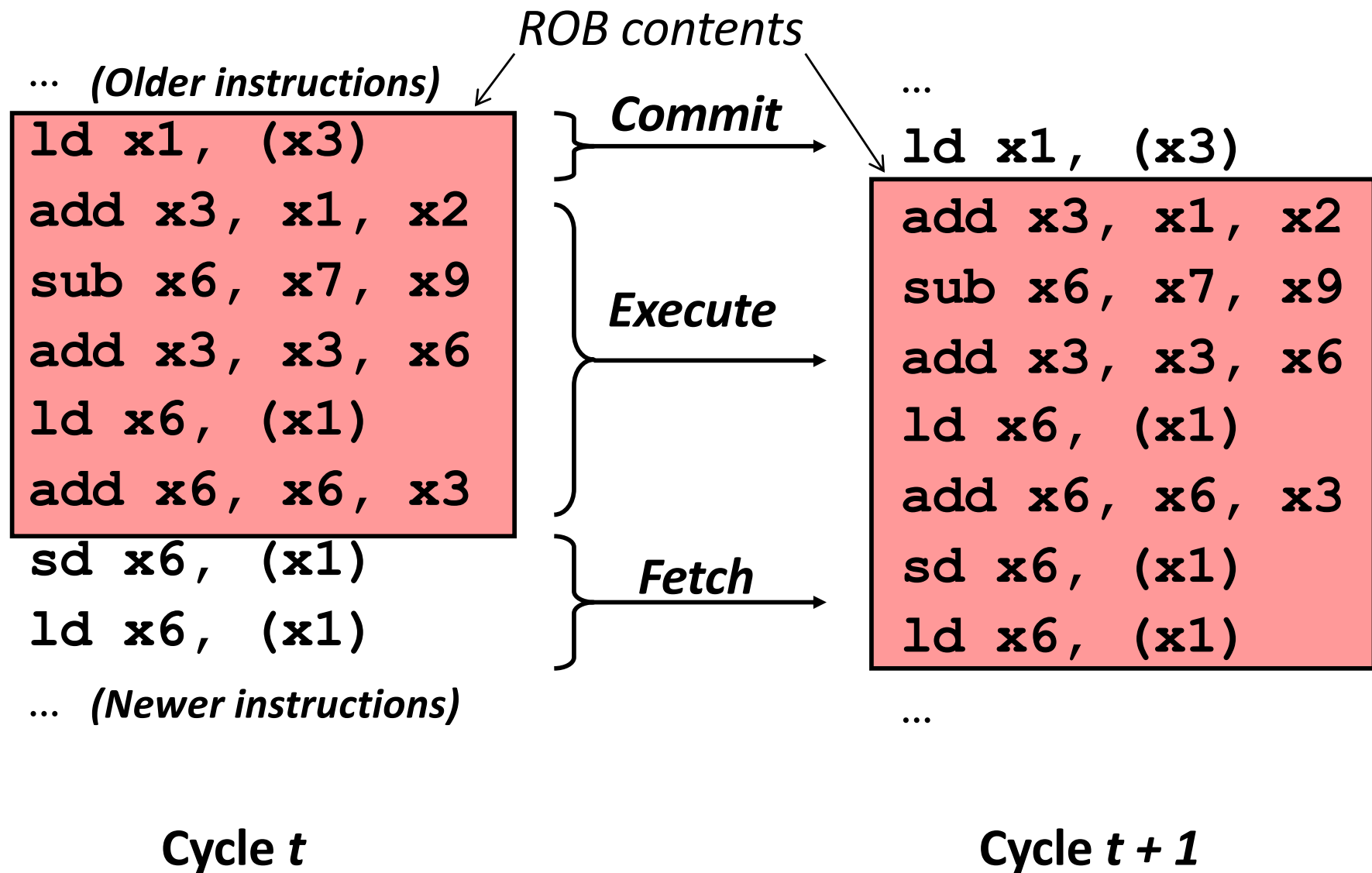
Separate Issue Window from ROB

The issue window holds only instructions that have been decoded and renamed but not issued into execution. Has register tags and presence bits, and pointer to ROB entry.

use	ex	op	p1	PR1	p2	PR2	PRd	ROB#

Reorder buffer used to hold exception information for commit.

Oldest

Done?	Rd	LPRd	PC	Except?

Free

ROB is usually several times larger than issue window – why?

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
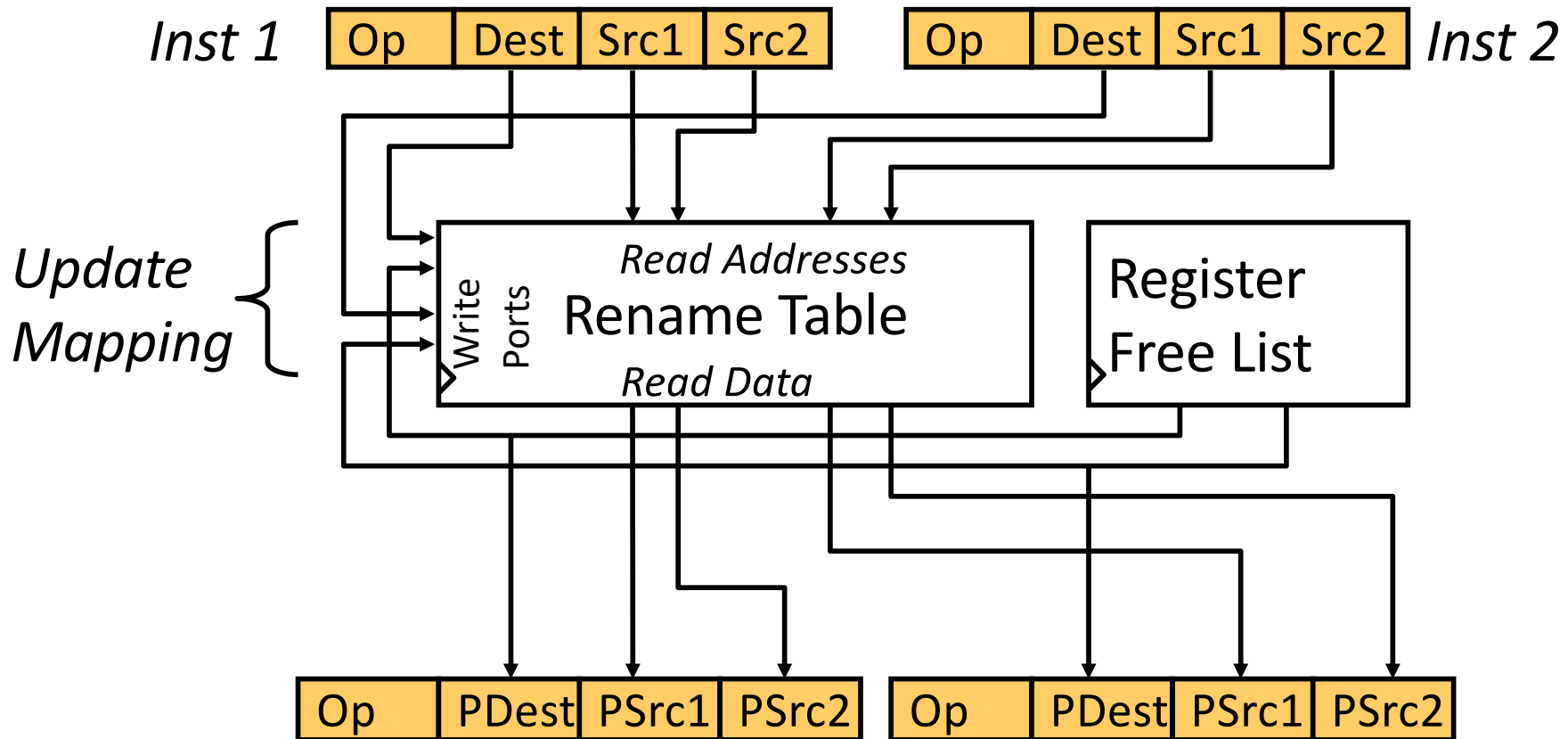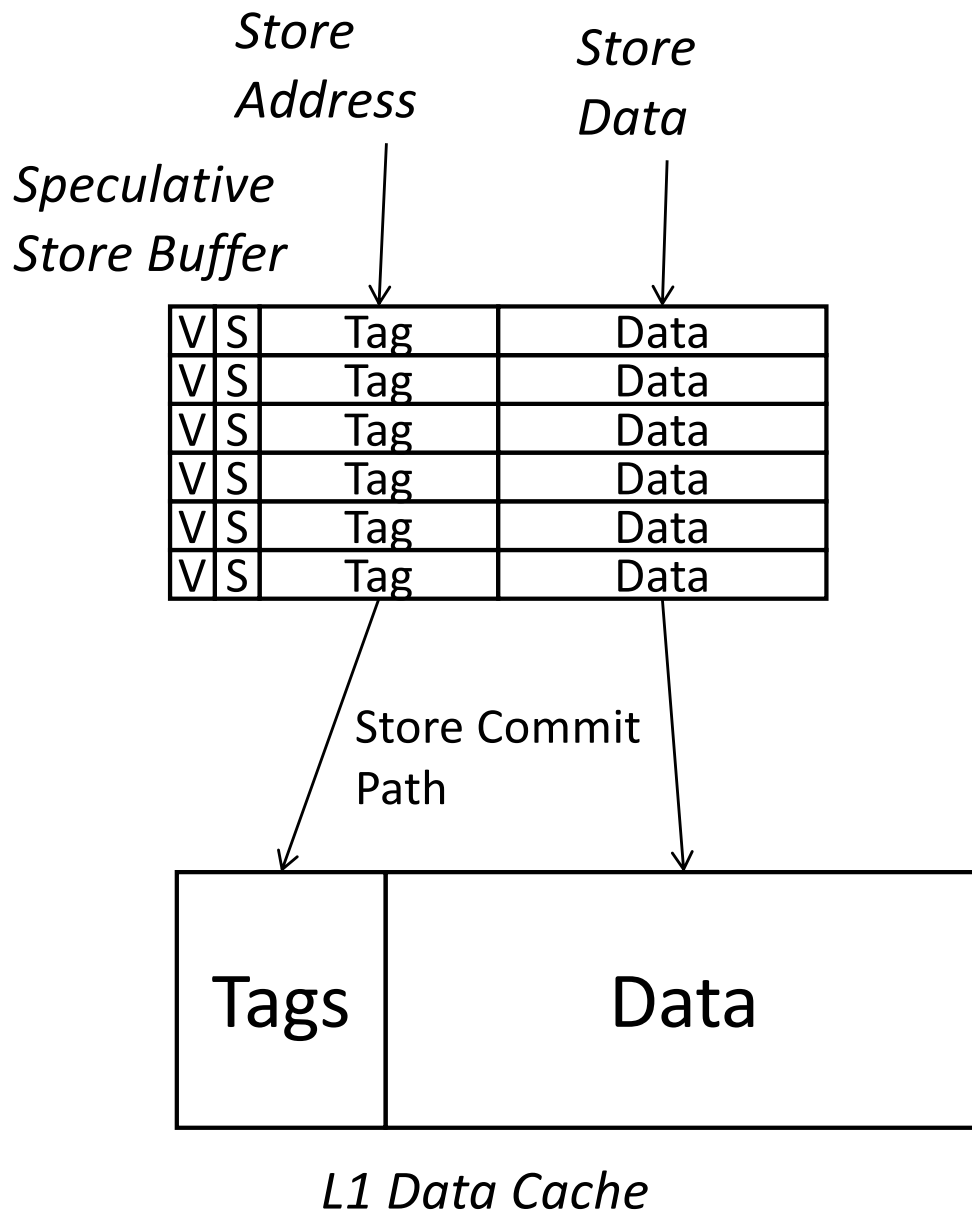

Does this work?

Superscalar Register Renaming

Inst 1 | Op | Dest | Src1 | Src2 | | Op | Dest | Src1 | Src2 | *Inst 2*

Update Mapping

Write Ports

Read Addresses
Rename Table
Read Data

=? =?

Register Free List

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

| Op | PDest | PSrc1 | PSrc2 | | Op | PDest | PSrc1 | PSrc2 |
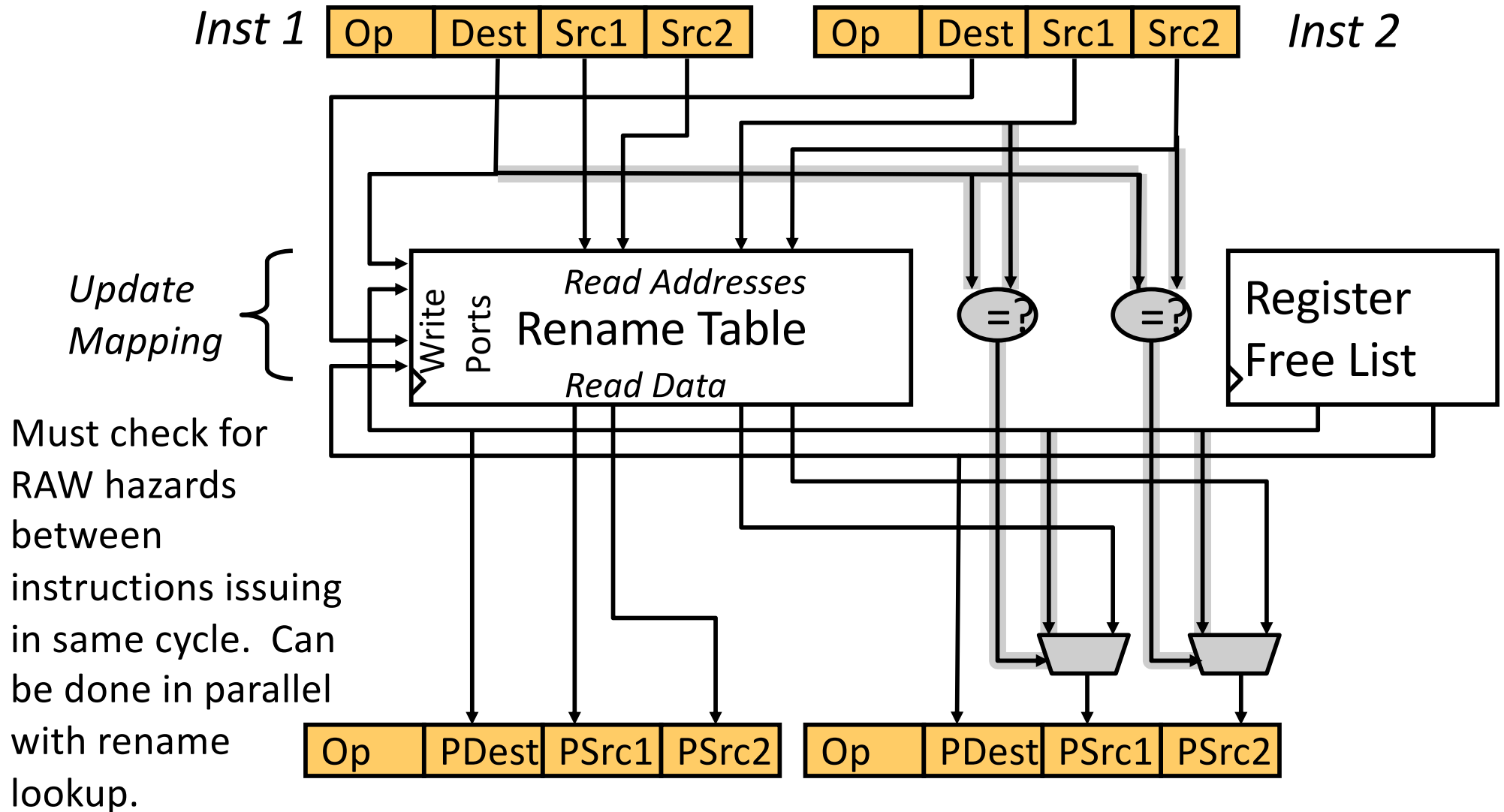
MIPS R10K renames 4 serially-RAW-dependent insts/cycle

Load-Store Queue Design

- After control hazards (branch prediction in next lecture), data hazards through memory are probably next most important bottleneck to superscalar performance

- Modern superscalars use very sophisticated load-store reordering techniques to reduce effective memory latency by allowing loads to be speculatively issued

Speculative Store Buffer

Store Address   *Store Data*

Speculative Store Buffer

V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data

Store Commit Path

Tags	Data

L1 Data Cache

- Just like register updates, stores should not modify the memory until after the instruction is committed. A speculative store buffer is a structure introduced to hold speculative store data.

- During decode, store buffer slot allocated in program order

- Stores split into "store address" and "store data" micro-operations

- "Store address" execution writes tag

- "Store data" execution writes data

- Store commits when oldest instruction and both address and data available:
  - clear speculative bit and eventually move data to cache

- On store abort:
  - clear valid bit

31

Load bypass from speculative store buffer

Speculative Store Buffer

Load Address

L1 Data Cache

V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data

Tags	Data

Load Data
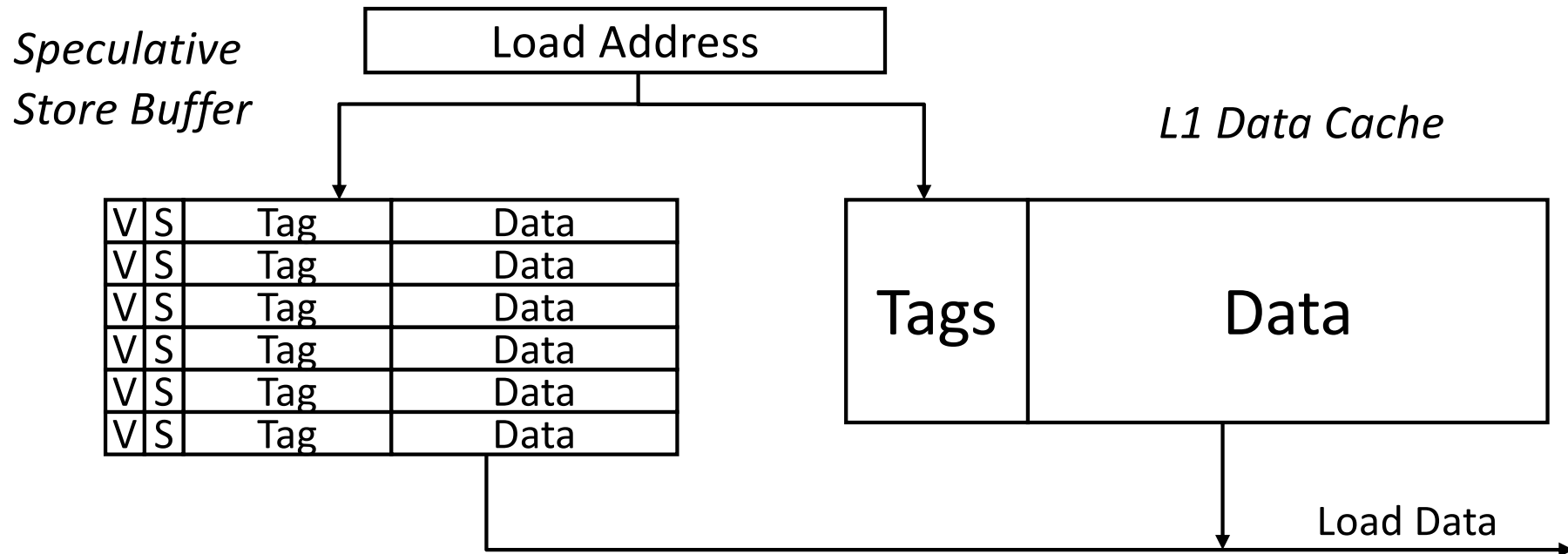
- If data in both store buffer and cache, which should we use?

  Speculative store buffer

- If same address in store buffer twice, which should we use?

  Youngest store older than load

Memory Dependencies

```
sd x1, (x2)
ld x3, (x4)
```

- When can we execute the load?

In-Order Memory Queue

- Execute all loads and stores in program order

=> Load and store cannot leave ROB for execution until all previous loads and stores have completed execution

- Can still execute loads and stores speculatively, and out-of-order with respect to other instructions

- Need a structure to handle memory ordering…

Conservative O-o-O Load Execution

```
sd x1, (x2)
ld x3, (x4)
```

- Can execute load before store, if addresses known and **x4** != **x2**

- Each load address compared with addresses of all previous uncommitted stores
  - can use partial conservative check i.e., bottom 12 bits of address, to save hardware

- Don't execute load if any previous store address not known

- (MIPS R10K, 16-entry address queue)

Address Speculation

```
sd x1, (x2)
ld x3, (x4)
```

- Guess that **x4** != **x2**

- Execute load before store address known

- Need to hold all completed but uncommitted load/store addresses in program order

- If subsequently find **x4==x2**, squash load and all following instructions

- => Large penalty for inaccurate address speculation

36

Memory Dependence Prediction
(Alpha 21264)

```
sd x1, (x2)
ld x3, (x4)
```

- Guess that **x4** != **x2** and execute load before store

- If later find **x4**==**x2**, squash load and all following instructions, but mark load instruction as store-wait

- Subsequent executions of the same load instruction will wait for all previous stores to complete

- Periodically clear store-wait bits

Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
    - Arvind (MIT)
    - Joel Emer (Intel/MIT)
    - James Hoe (CMU)
    - John Kubiatowicz (UCB)
    - David Patterson (UCB)