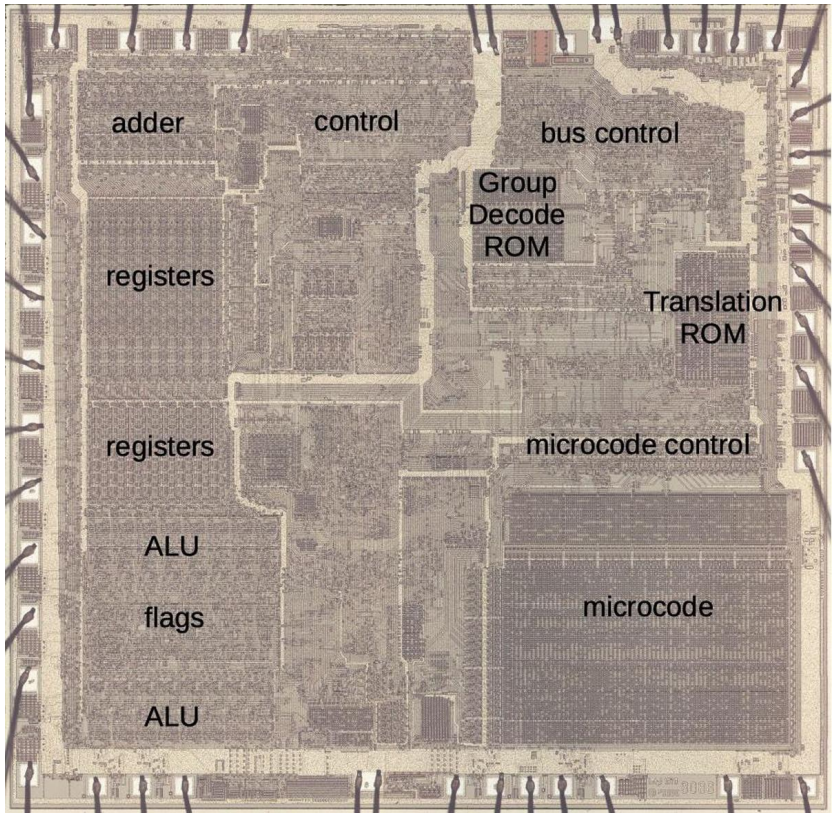
CS 152/252A Computer Architecture and Engineering

Sophia Shao

Lecture 12: Out-of-Order Execution I

How the 8086 processor's microcode engine works

The 8086 microprocessor was a groundbreaking processor introduced by Intel in 1978. It led to the x86 architecture that still dominates desktop and server computing. The 8086 chip uses microcode internally to implement its instruction set. I've been reverse-engineering the 8086 from die photos and this blog post discusses how the chip's microcode engine operated.

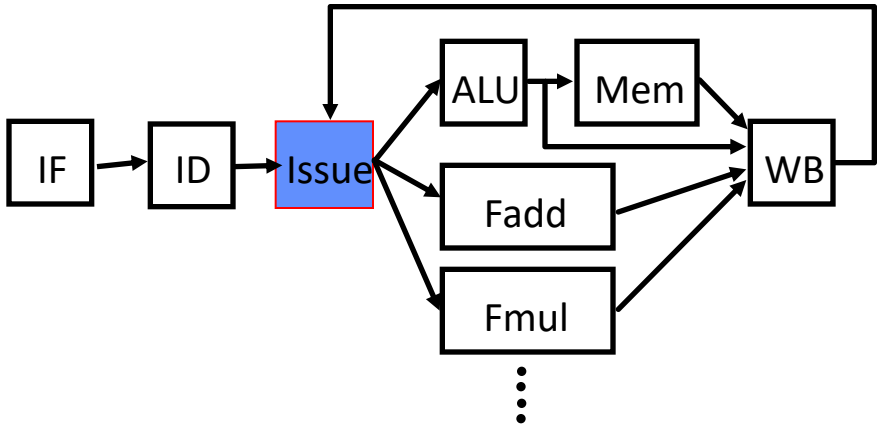


https://www.righto.com/2022/11/how-8086-processors-microcode-engine.html

Last Time in Lecture

- 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

Register Renaming



- Decode does register renaming and adds instructions to the issue-stage instruction reorder buffer (ROB)

  ➔ renaming makes WAR or WAW hazards impossible

- Any instruction in ROB whose RAW hazards have been satisfied can be issued

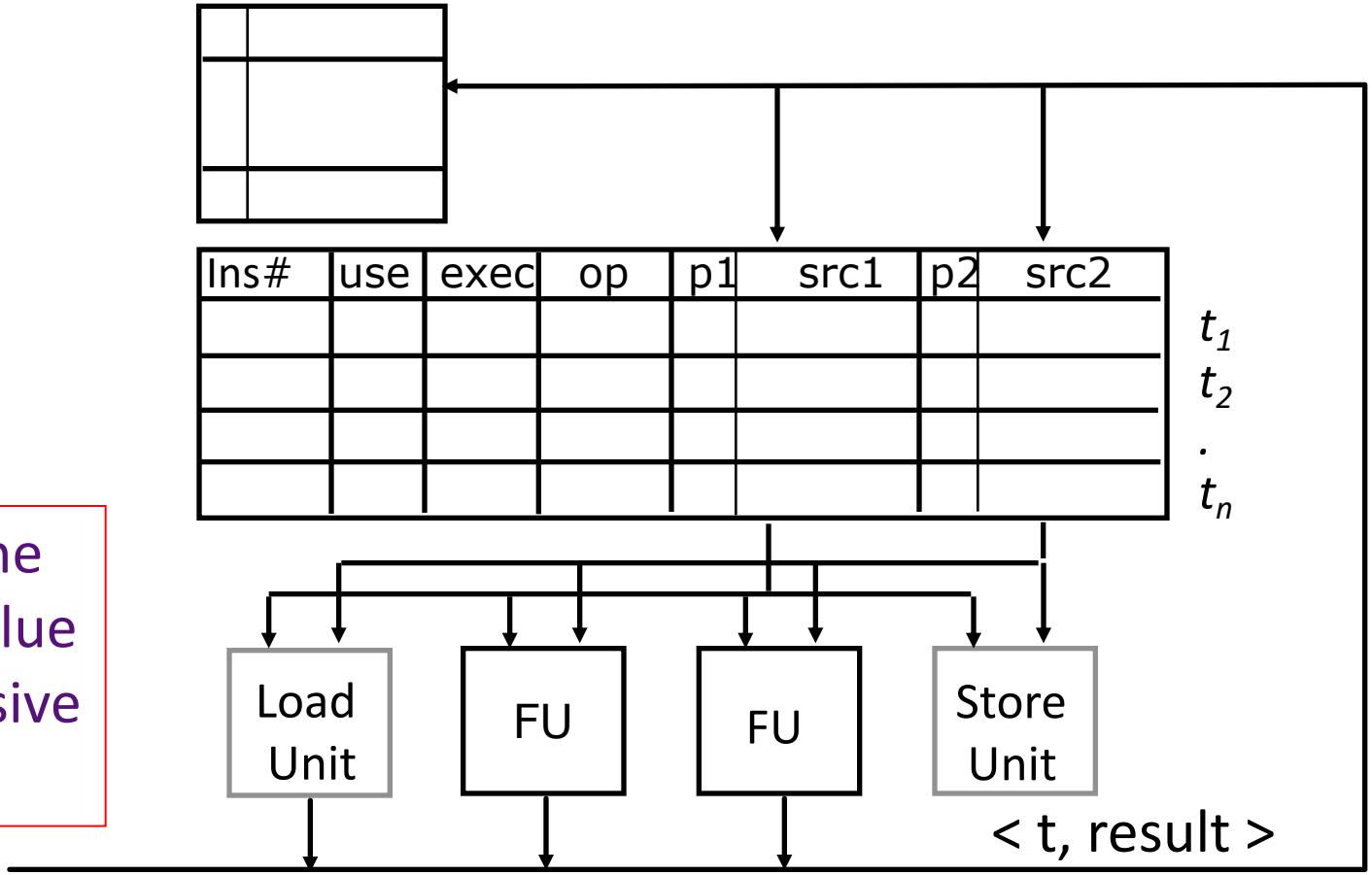
  ➔ Out-of-order or dataflow execution

Renaming Structures

Renaming table & regfile

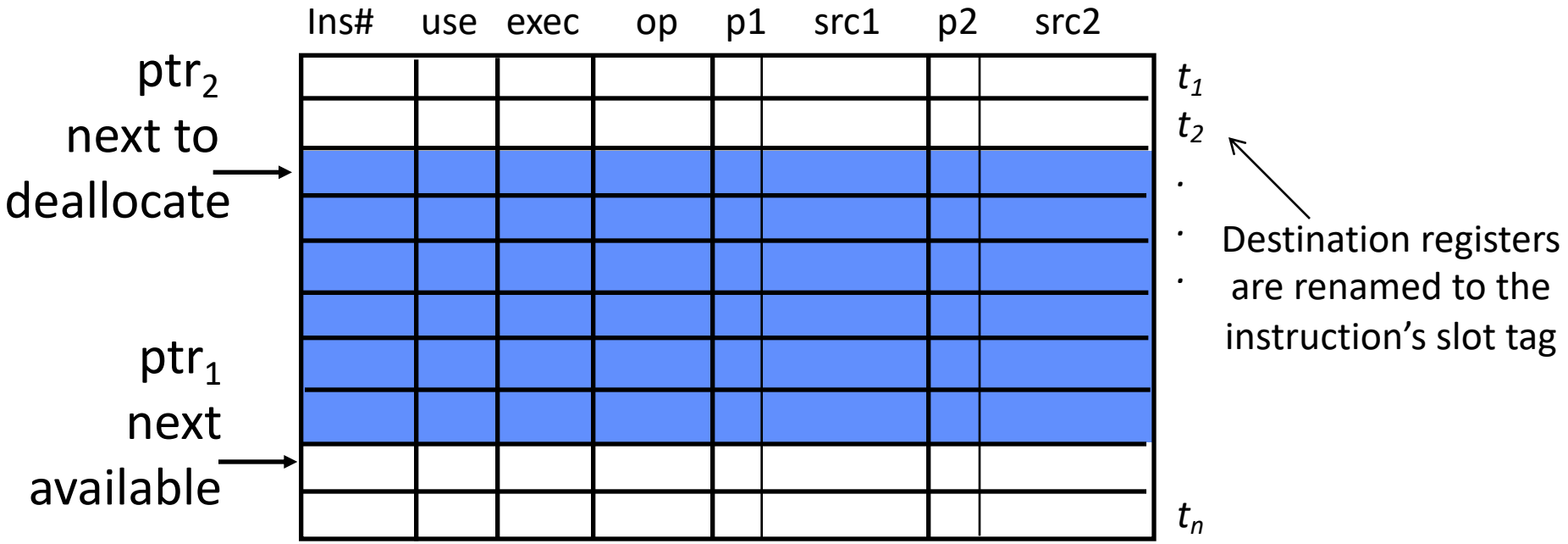
Reorder buffer

Replacing the tag by its value is an expensive operation



- Instruction template (i.e., tag t) is allocated by the Decode stage, which also associates tag with register in regfile
- When an instruction completes, its tag is deallocated

Reorder Buffer Management



ROB managed circularly

- "exec" bit is set when instruction begins execution
- When an instruction completes its "use" bit is marked free
- ptr_2 is incremented only if the "use" bit is marked free

Instruction slot is candidate for execution when:

- It holds a valid instruction ("use" bit is set)
- It has not already started execution ("exec" bit is clear)
- Both operands are available (p1 and p2 are set)

Renaming & Out-of-order Issue

An example

Renaming table

	p	data
f1		
f2		t1
f3		
f4		t5
f5		
f6		t3
f7		
f8		t4

Reorder buffer

	Ins#	use	exec	op	p1	src1	p2	src2	
	1	0	0	LD					t_1
	2	1	0	LD					t_2
	3	1	0	MUL	0	t2	1	v1	t_3
	4	1	0	SUB	1	v1	1	v1	t_4
	5	1	0	DIV	1	v1	0	t4	t_5
									.
									.

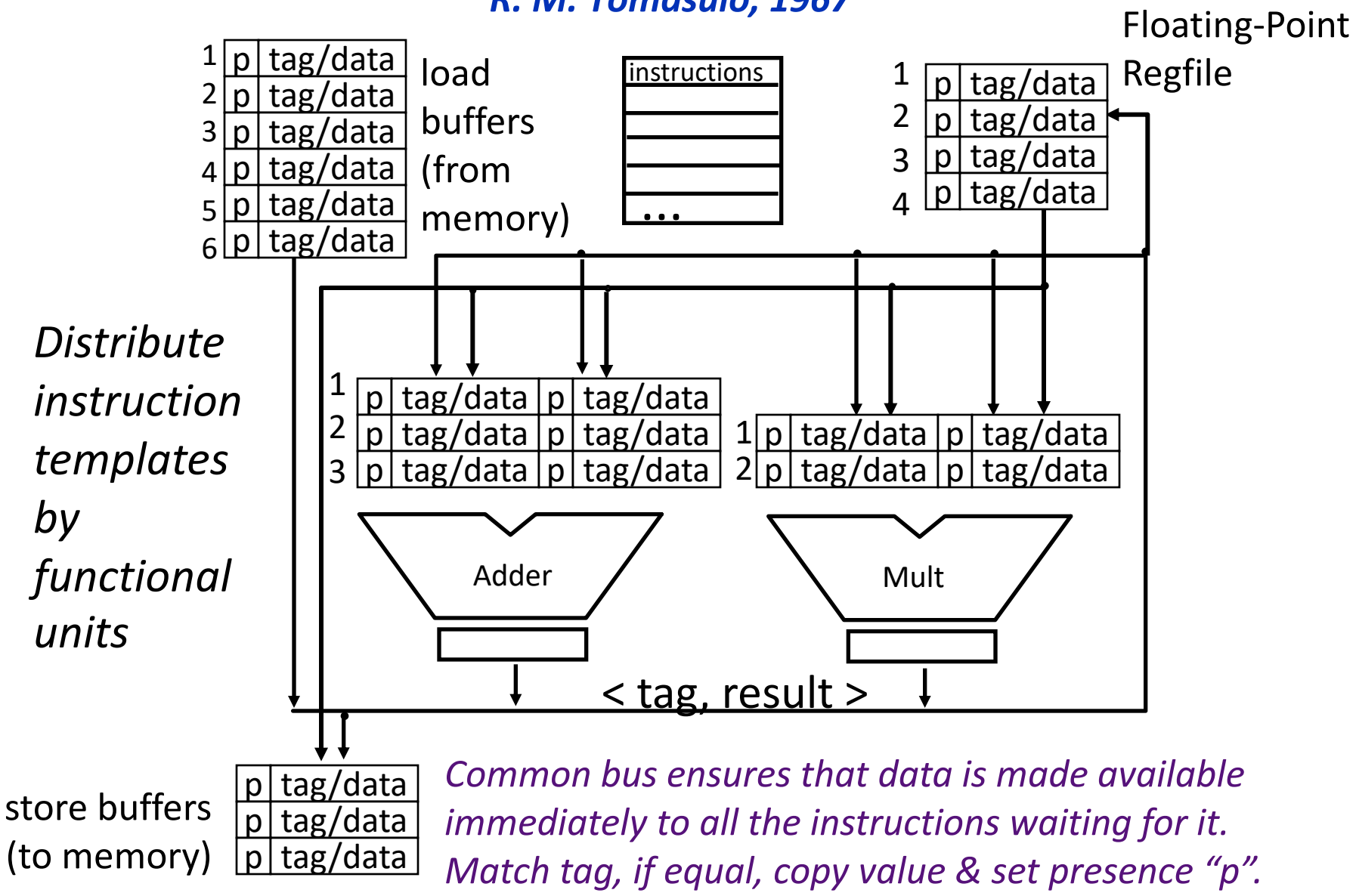
data / t_i

1 FLD	f2,	34(x2)	
2 FLD	f4,	45(x3)	
3 FMULT.D	f6,	f4,	f2
4 FSUB.D	f8,	f2,	f2
5 FDIV.D	f4,	f2,	f8
6 FADD.D	f10,	f6,	f4

- *When are tags in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

IBM 360/91 Floating-Point Unit

R. M. Tomasulo, 1967



Out-of-Order Fades into Background

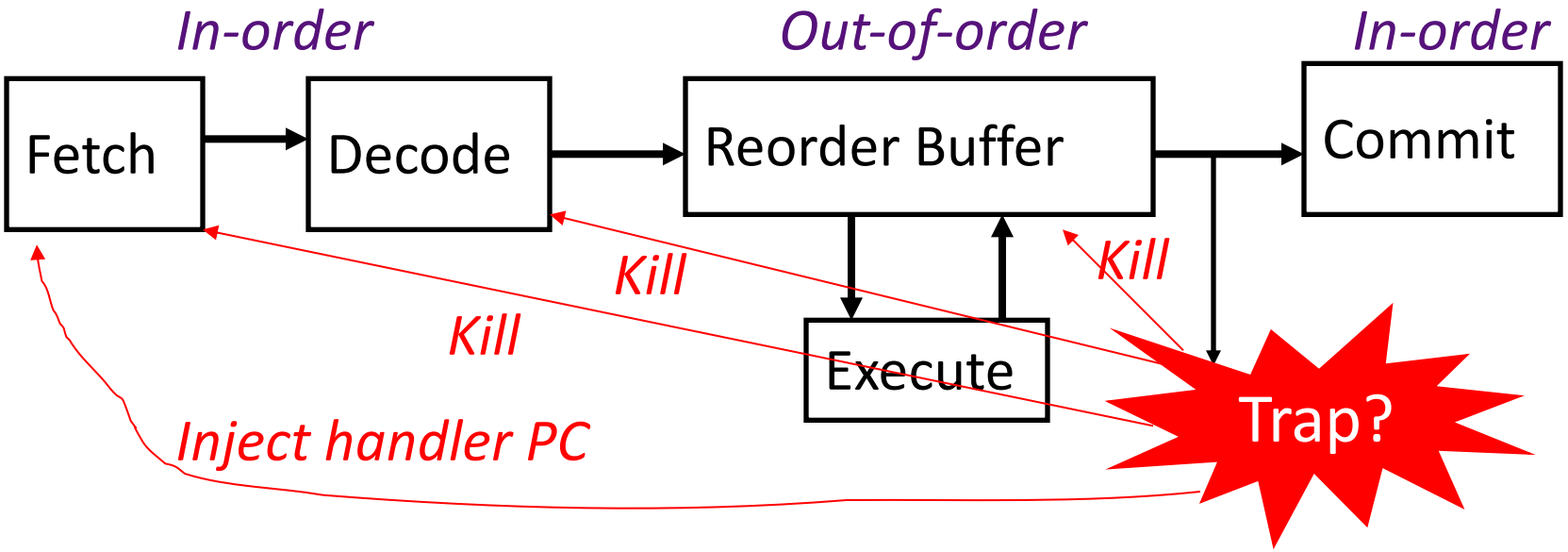
Out-of-order processing implemented commercially in 1960s, but disappeared again until 1990s as two major problems had to be solved:

- Precise traps
  - Imprecise *traps* complicate debugging and OS code
  - Note, precise *interrupts* are relatively easy to provide
- Branch prediction
  - Amount of exploitable instruction-level parallelism (ILP) limited by control hazards

Also, simpler machine designs in new technology beat complicated machines in old technology

  - Big advantage to fit processor & caches on one chip
  - Microprocessors had era of 1%/week performance scaling

In-Order Commit for Precise Traps

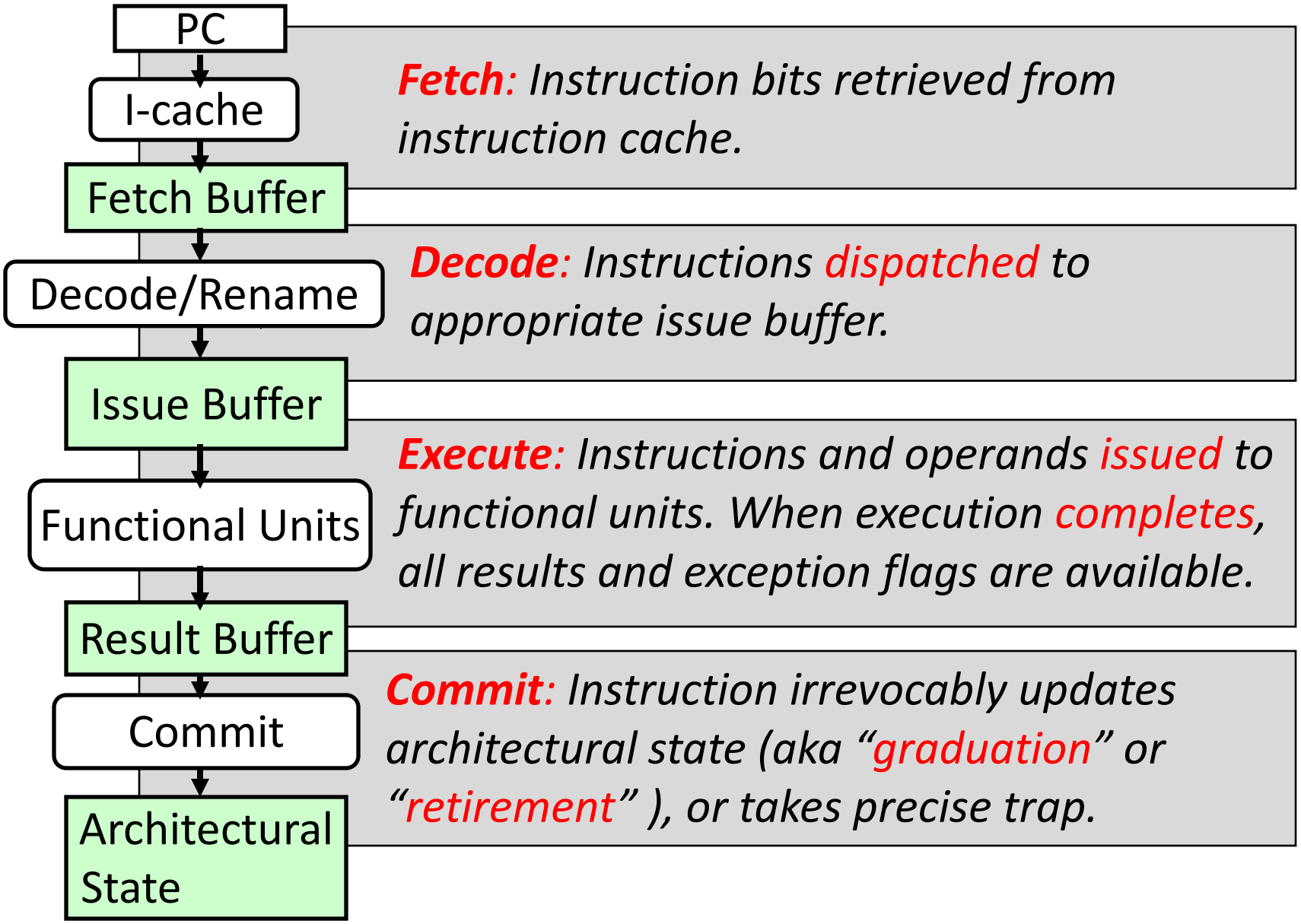


- In-order instruction fetch and decode, and dispatch to reservation stations inside reorder buffer
- Instructions issue from reservation stations out-of-order
- Out-of-order completion, values stored in temporary buffers
- Commit is in-order, checks for traps, and if none updates architectural state

Separating Completion from Commit

- Re-order buffer holds register results from completion until commit
  - Entries allocated in program order during decode
  - Buffers completed values and exception state until in-order commit point
  - Completed values can be used by dependents before committed (bypassing)
  - Each entry holds program counter, instruction type, destination register specifier and value if any, and exception status (info often compressed to save hardware)
- Memory reordering needs special data structures
  - Speculative store address and data buffers
  - Speculative load address and data buffers

Phases of Instruction Execution



In-Order versus Out-of-Order Phases

- Instruction fetch/decode/rename always in-order
  - Need to parse ISA sequentially to get correct semantics
- 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
- 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

CS152 Administrivia

- Midterm 7-9pm Tuesday 2/28
  - Covers lectures 1 – 10, plus assigned problem sets, labs, book readings
  - Excludes this week
  - 155 Dwinelle
  - Midterm review this week in Discussions.
- HW3 out this week
  - Due 3/14
- Lab 2
  - Due 3/02

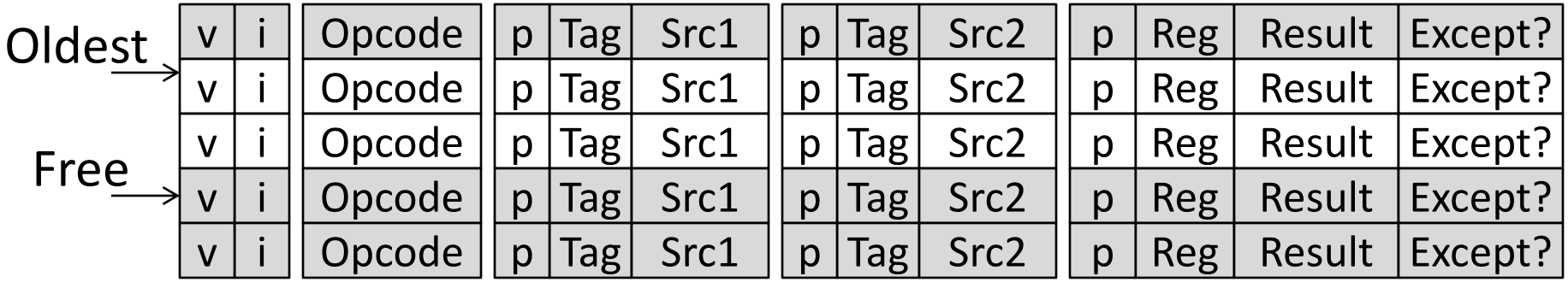
CS252 Administrivia

- Project proposal presentations next week and the week after

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

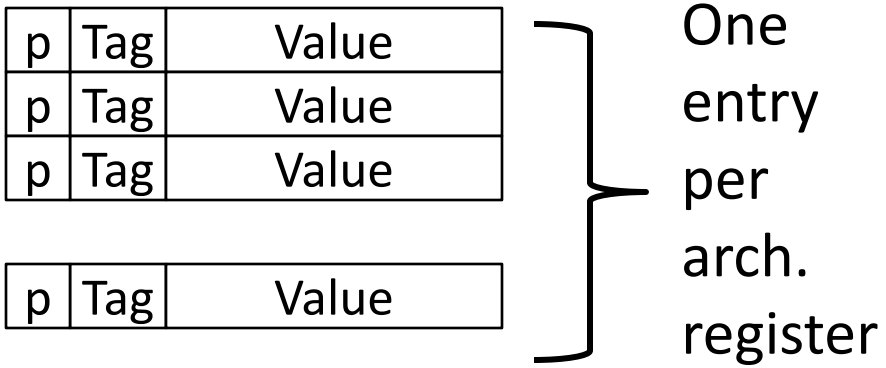
"Data-in-ROB" Design
(HP PA8000, Pentium Pro, Core2Duo, Nehalem)



- 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

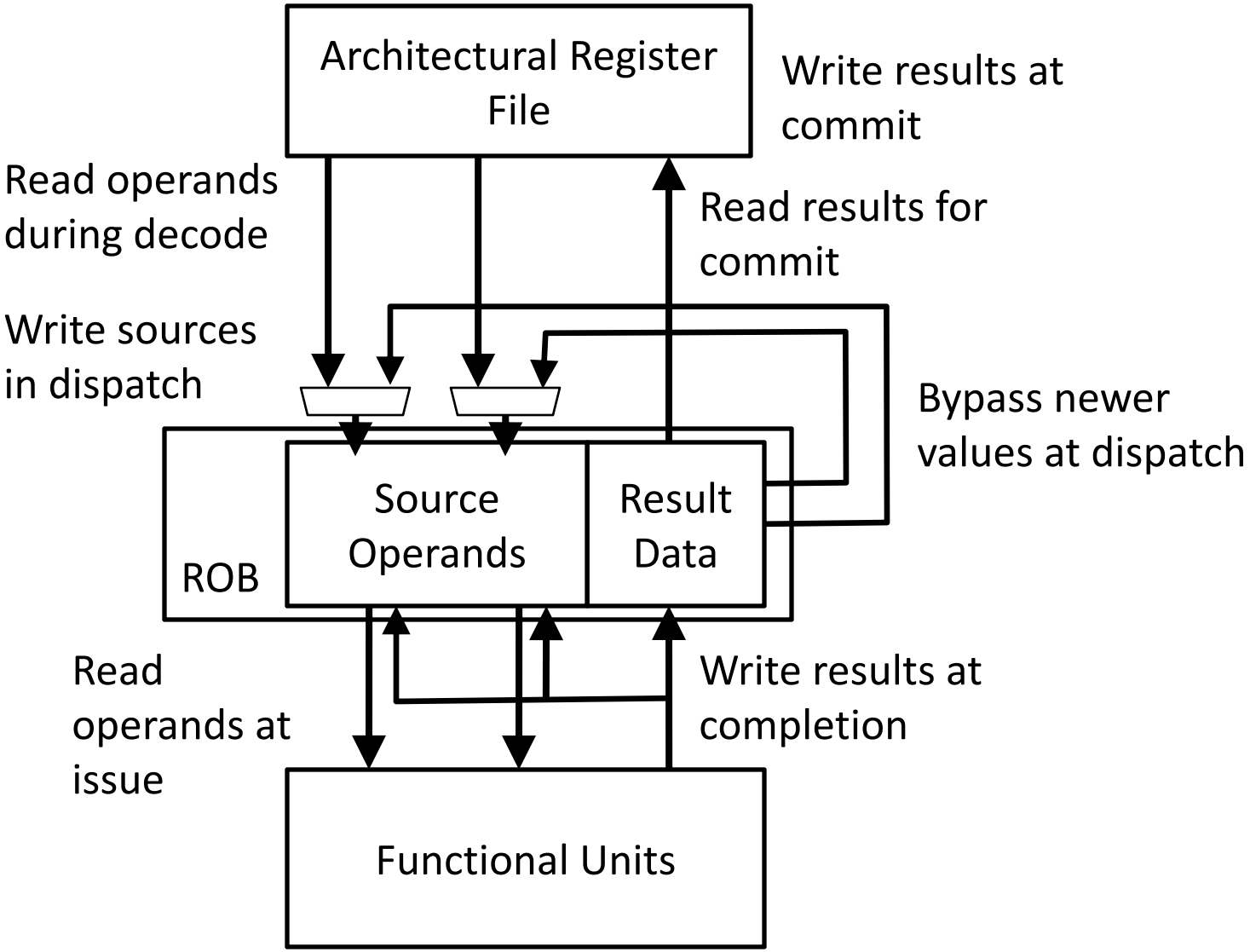
Managing Rename for Data-in-ROB

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



- 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)

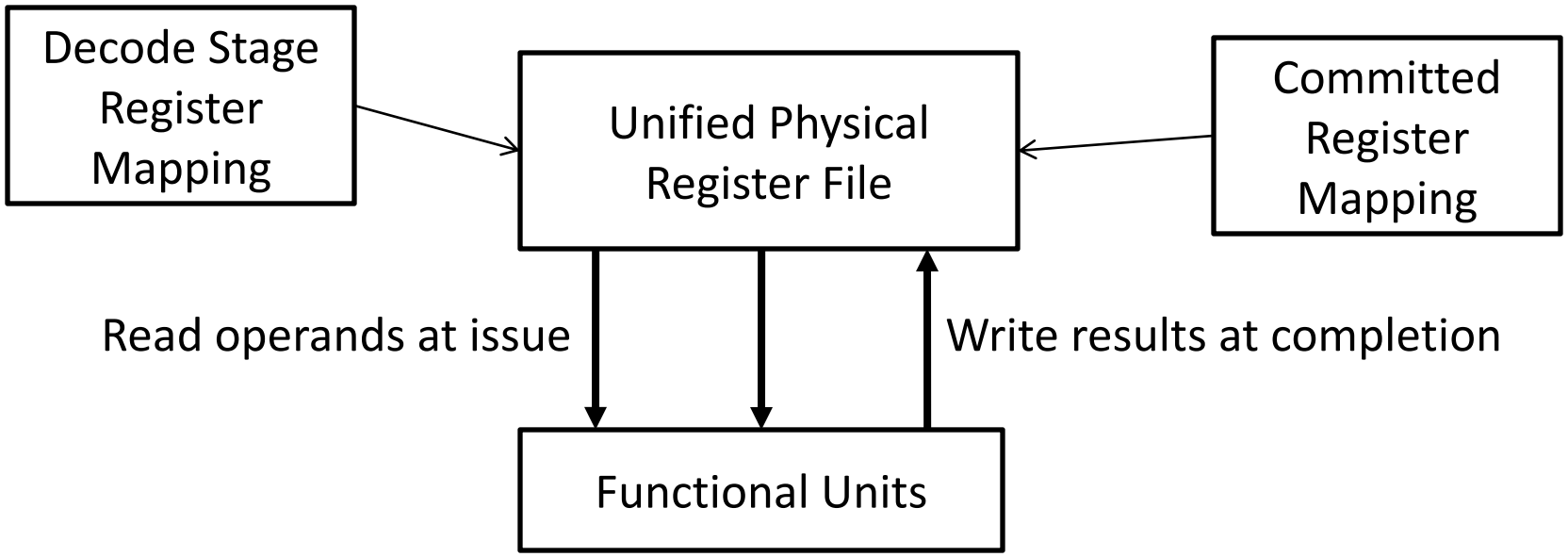
Data Movement in Data-in-ROB Design



Unified Physical Register File

(MIPS R10K, Alpha 21264, Intel Pentium 4 & Sandy/Ivy Bridge)

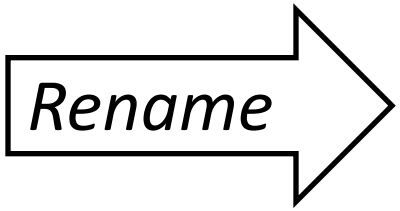
- Rename all architectural registers into a single *physical* register file during decode, no register values read
- Functional units read and write from single unified register file holding committed and temporary registers in execute
- Commit only updates mapping of architectural register to physical register, no data movement



Lifetime of Physical Registers

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

```
ld x1, (x3)
addi x3, x1, #4
sub x6, x7, x9
add x3, x3, x6
ld x6, (x1)
add x6, x6, x3
sd x6, (x1)
ld x6, (x11)
```




```
ld P1, (Px)
addi P2, P1, #4
sub P3, Py, Pz
add P4, P2, P3
ld P5, (P1)
add P6, P5, P4
sd P6, (P1)
ld P7, (Pw)
```

When can we reuse a physical register?

When next writer of same architectural register commits

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

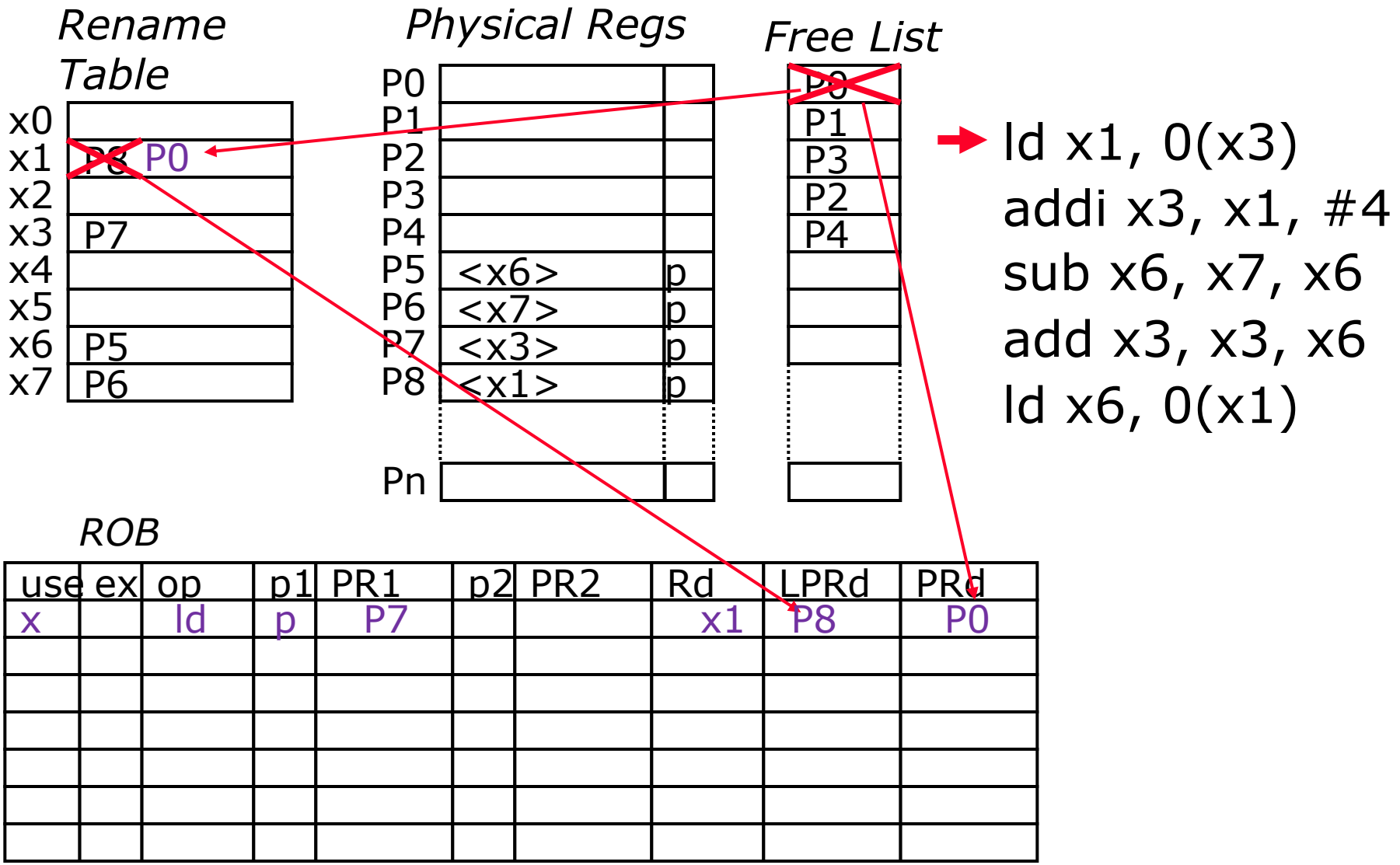
ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd

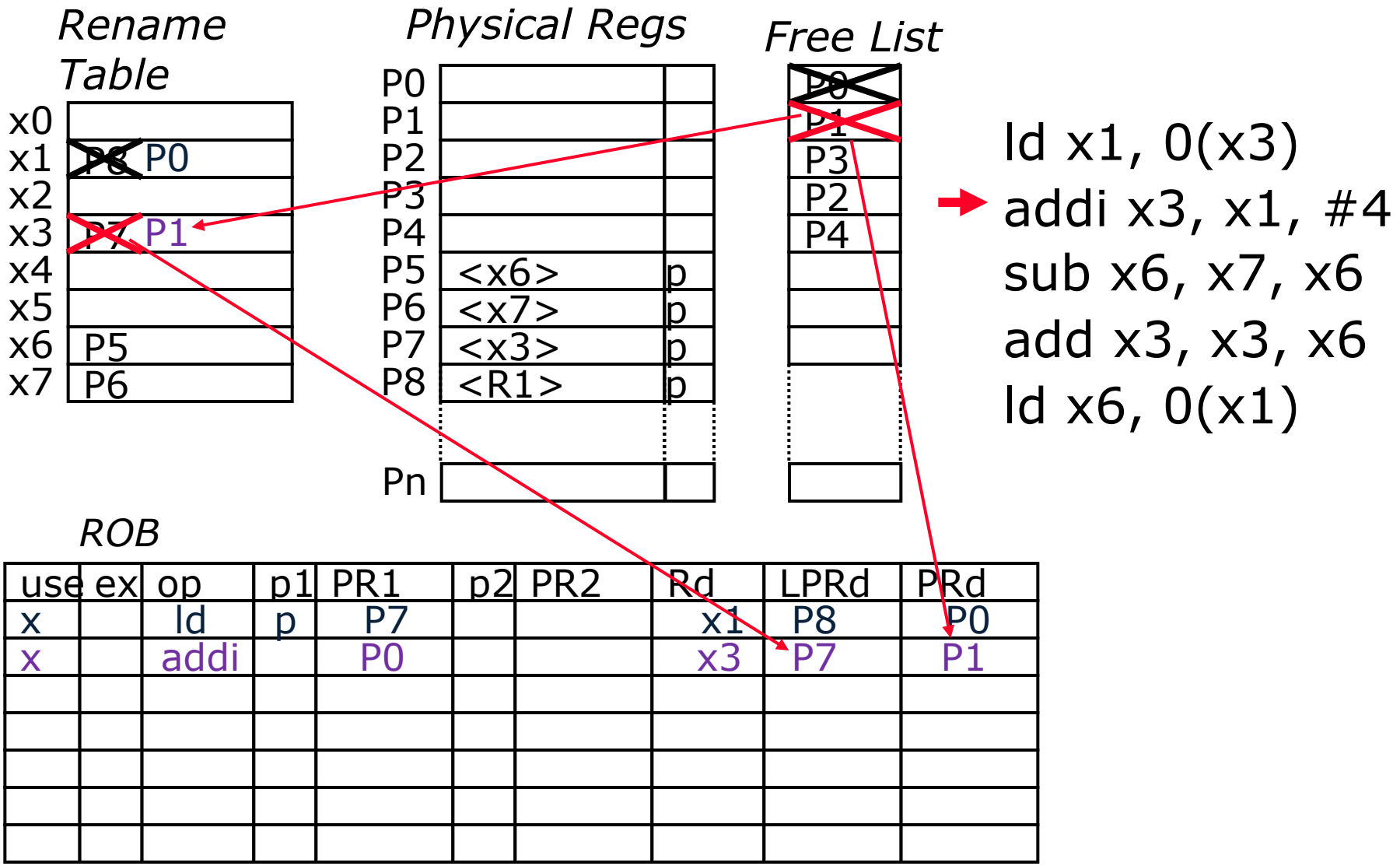
(LPRd requires third read port on Rename Table for each instruction)

Physical Register Management



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

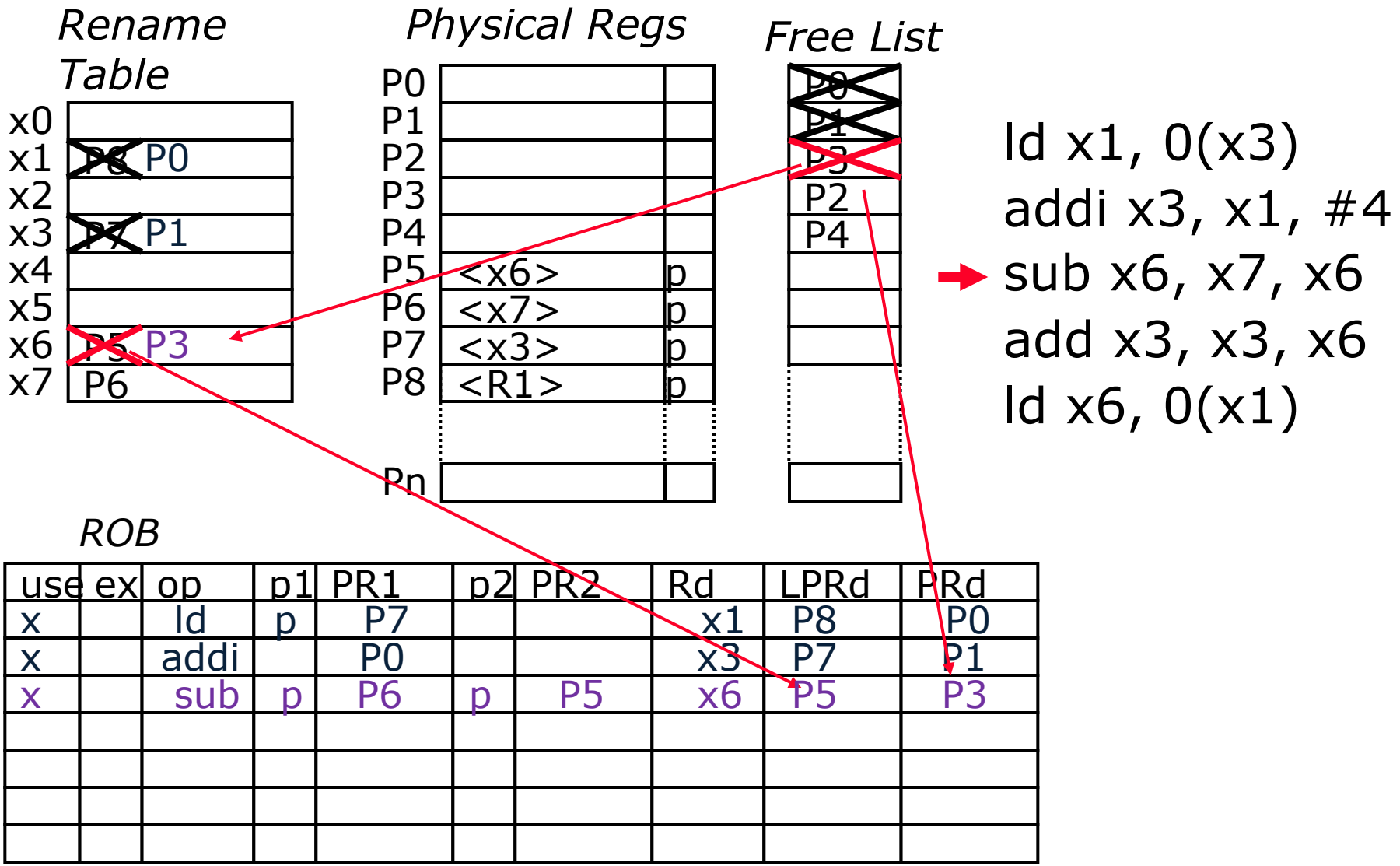
Physical Register Management



ROB

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

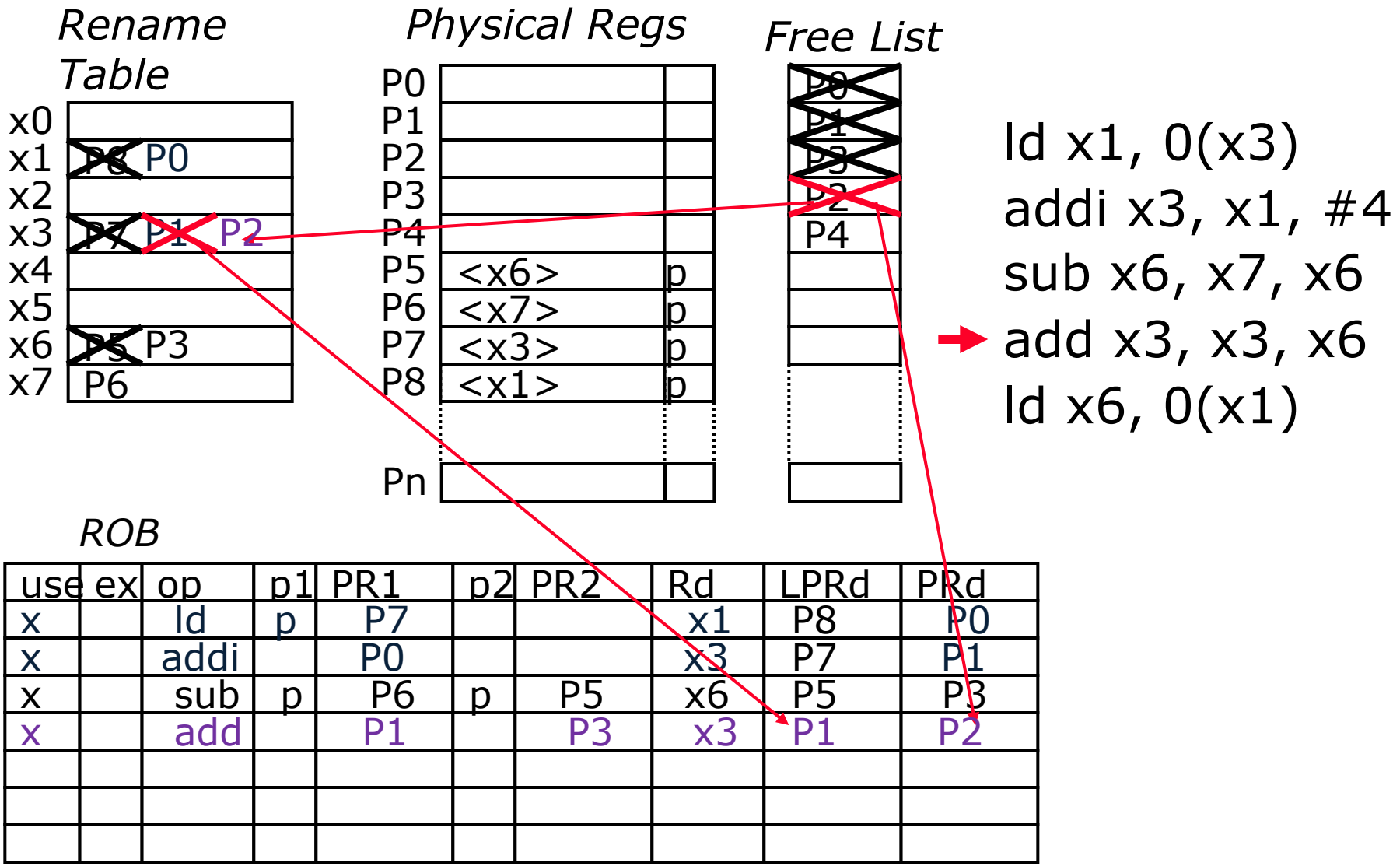
Physical Register Management



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

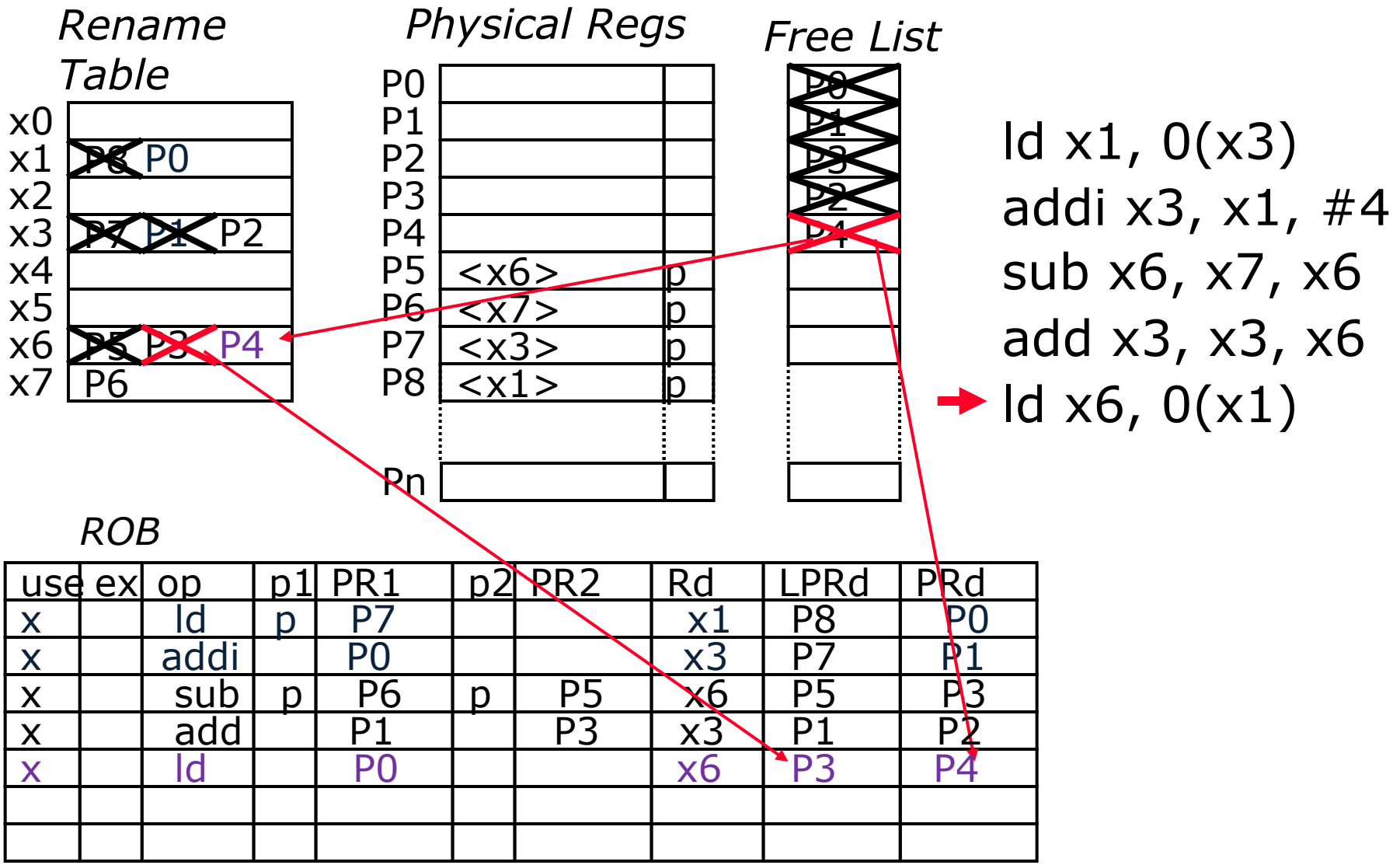
Physical Register Management



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

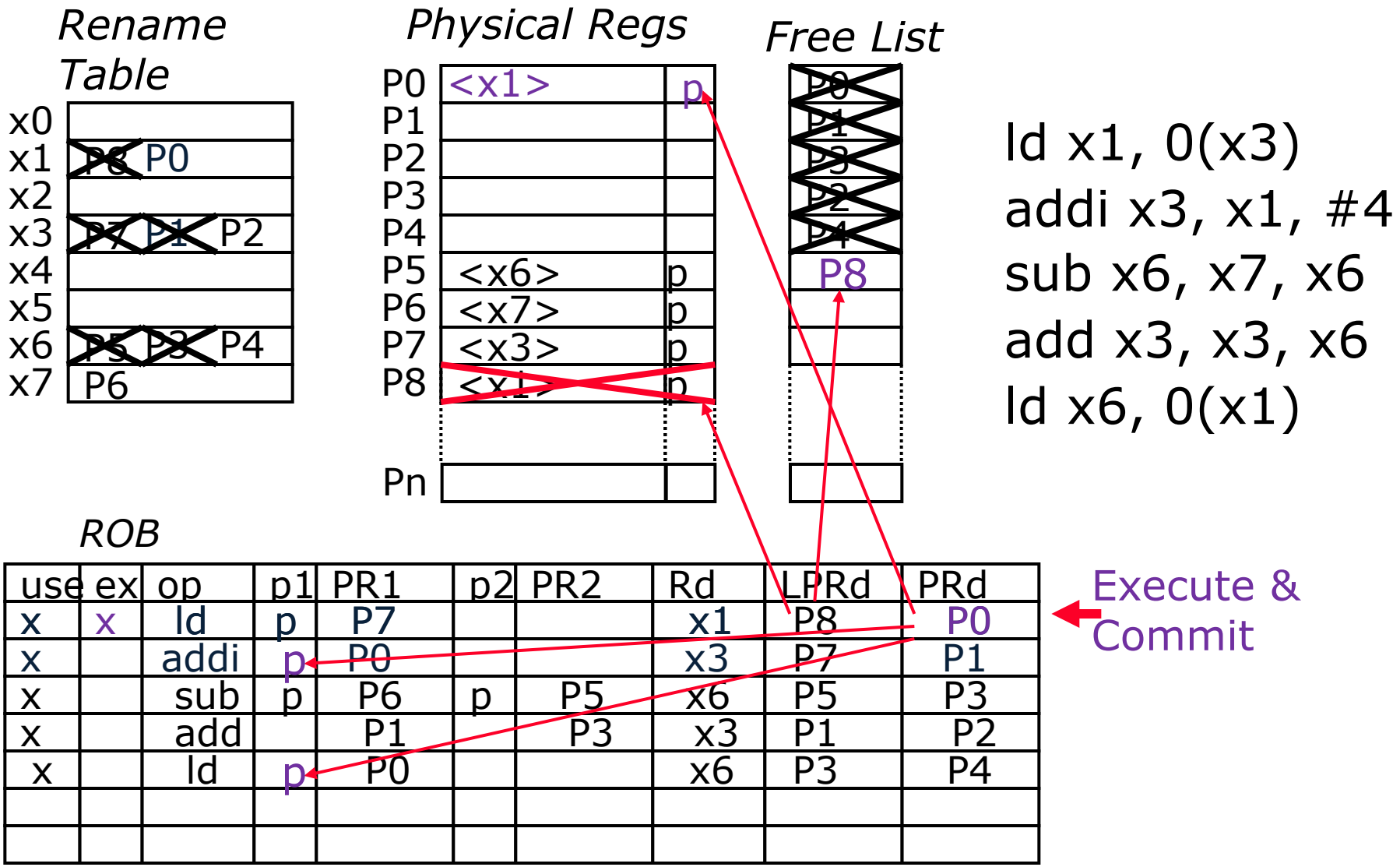
Physical Register Management



ROB

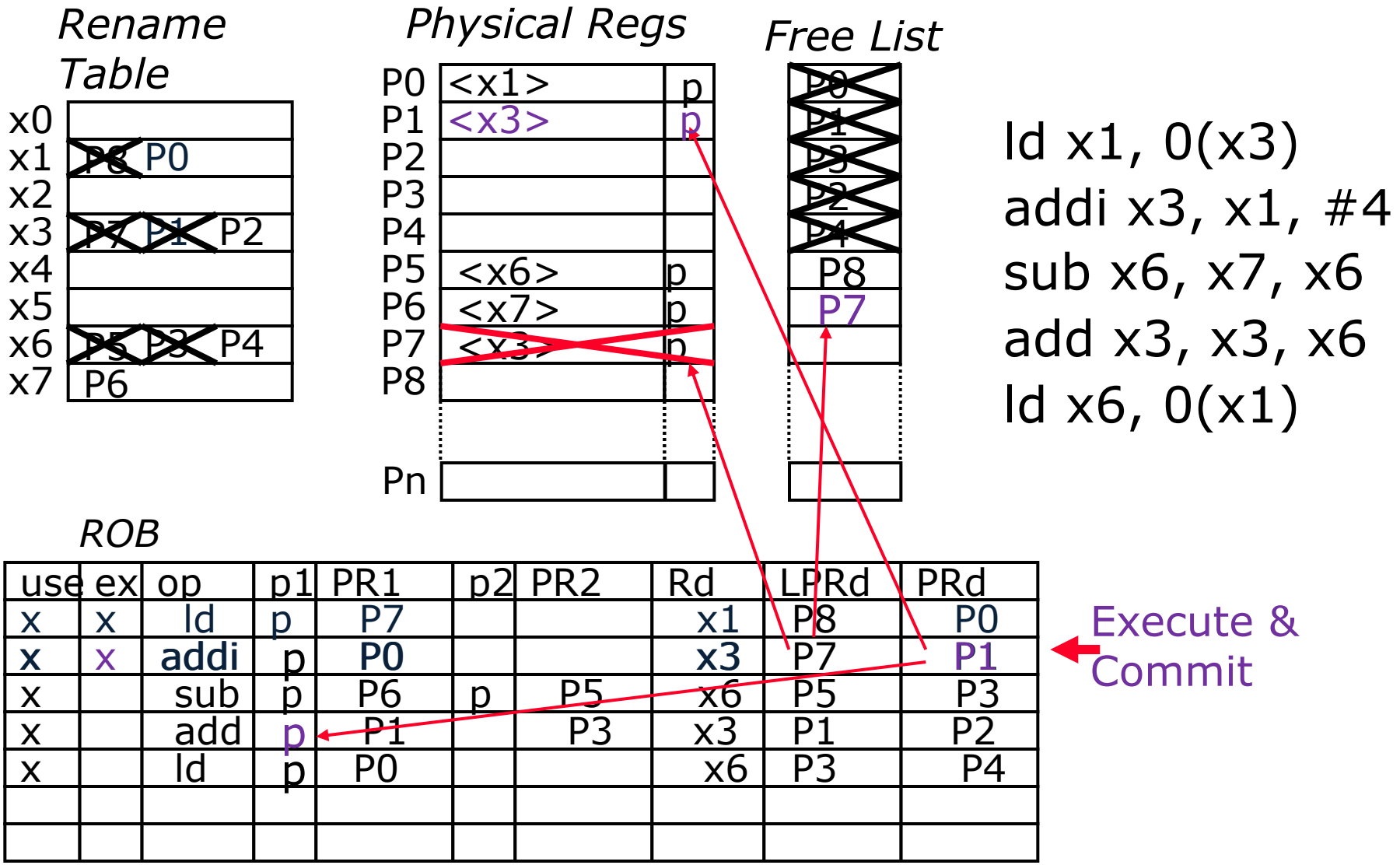
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add		P1		P3	x3	P1	P2
x		ld		P0			x6	P3	P4

Physical Register Management



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

Physical Register Management

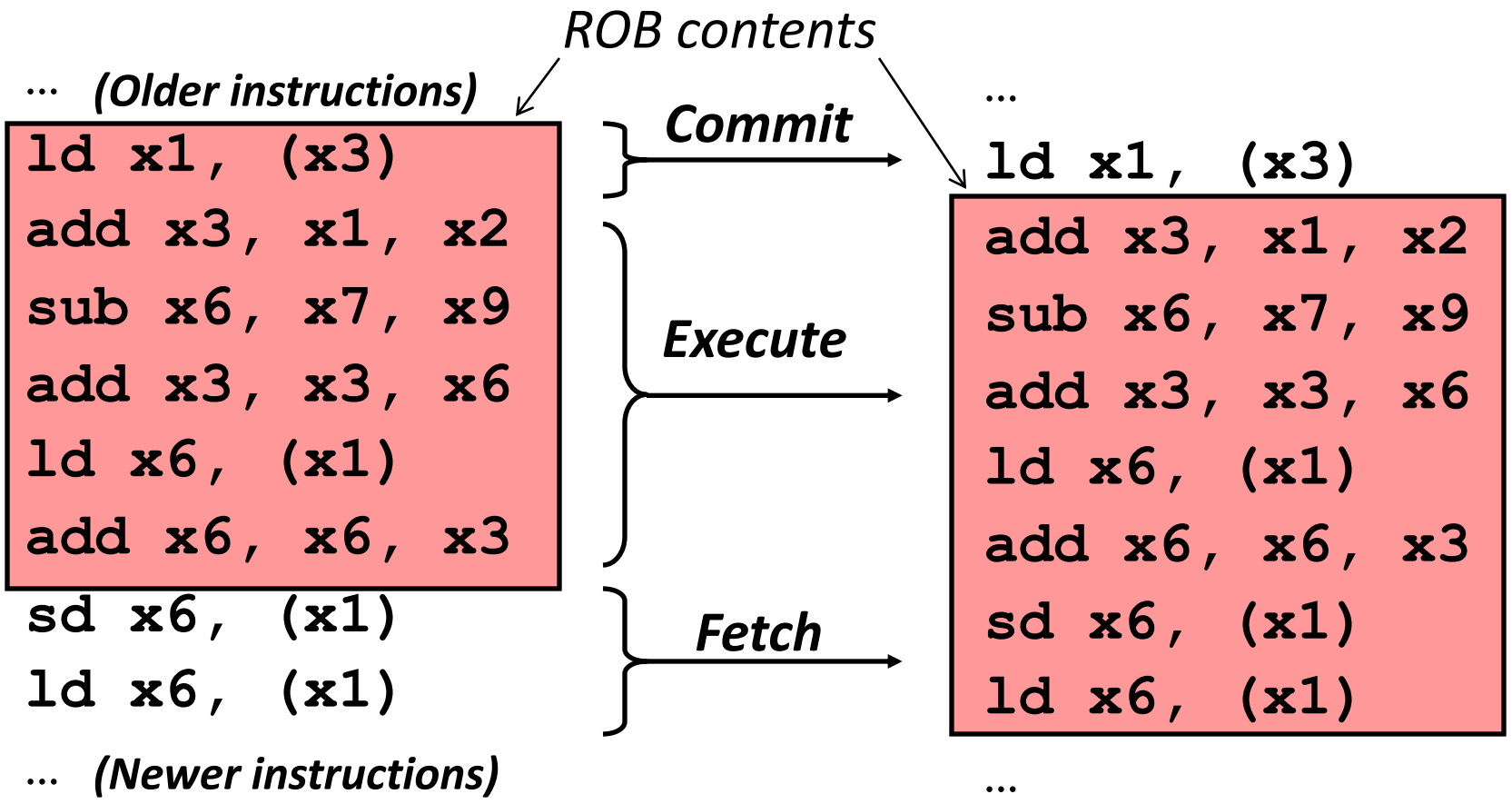


use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x	x	ld	p	P7			x1	P8	P0
x	x	addi	p	P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add	p	P1		P3	x3	P1	P2
x		ld	p	P0			x6	P3	P4

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)

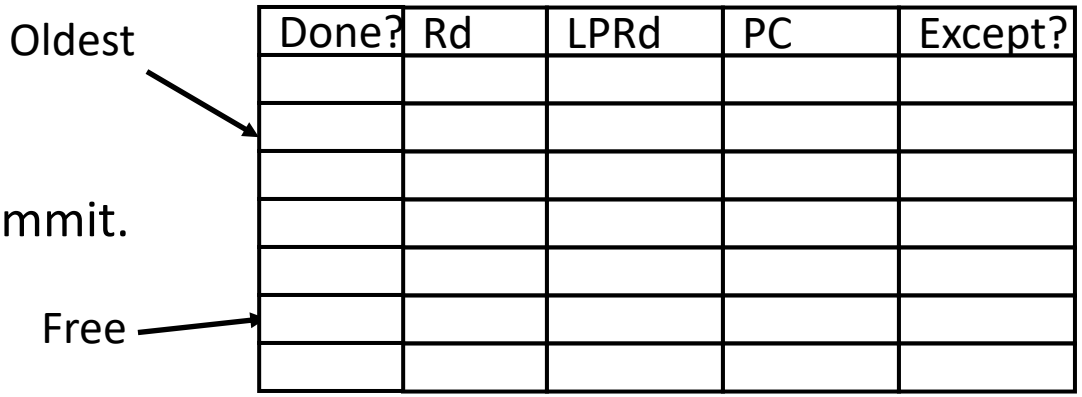


Separate Issue Window from ROB

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.



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

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)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)