

CS 152 Computer Architecture and Engineering

Lecture 11 - Out-of-Order Issue, Register Renaming, & Branch Prediction

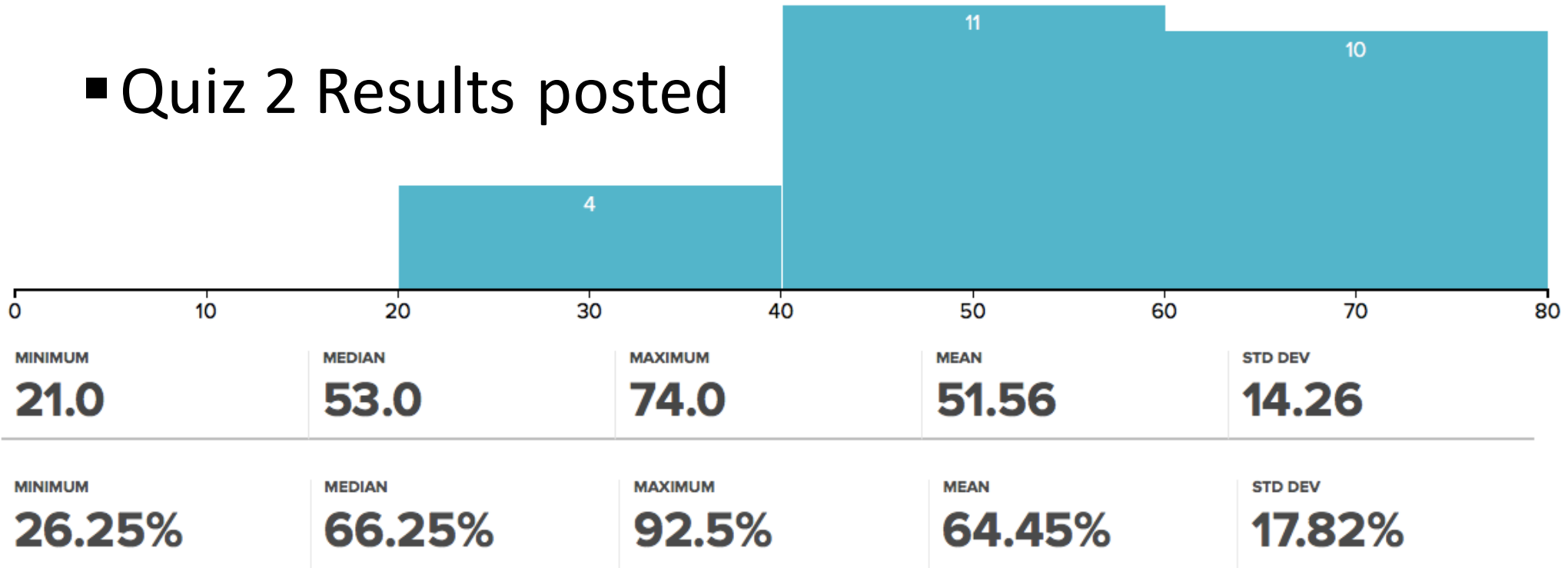
John Wawrzynek

Electrical Engineering and Computer Sciences
University of California at Berkeley

<http://www.eecs.berkeley.edu/~johnw>
<http://inst.eecs.berkeley.edu/~cs152>

CS152 Administrivia

- Quiz 2 Results posted

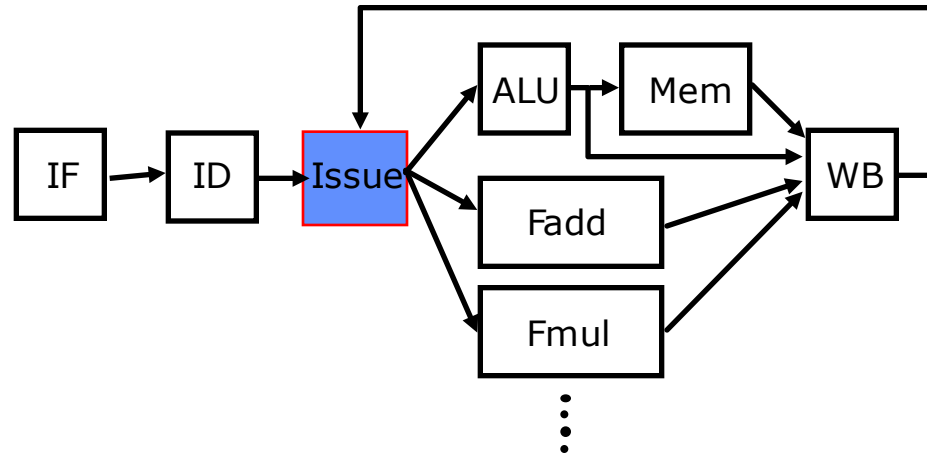


- PS3 posted

Last time in Lecture 12

- 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
- Dynamic issue logic can support out-of-order execution to improve performance
 - Last time, looked at simple scoreboard to track out-of-order completion
- Hardware register renaming can further improve performance by removing hazards.

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

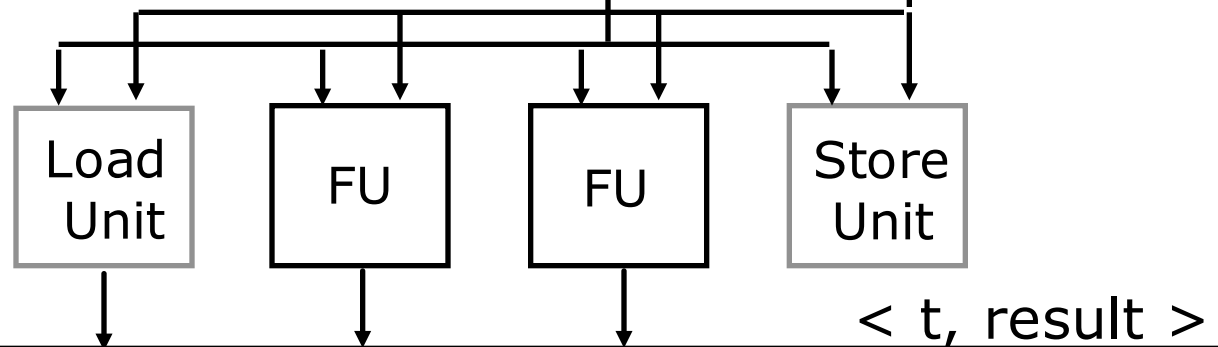
Renaming
table &
regfile

Reorder
buffer

Ins#	use	exec	op	p1	src1	p2	src2

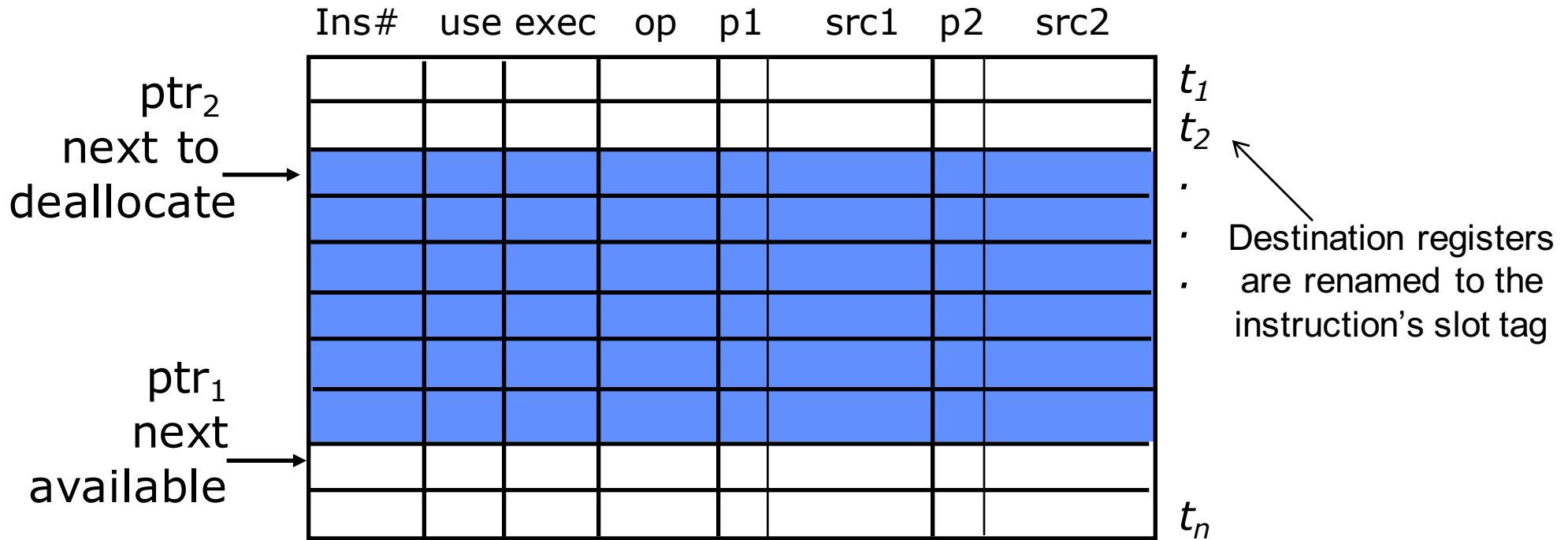
t_1
 t_2
.
.
 t_n

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

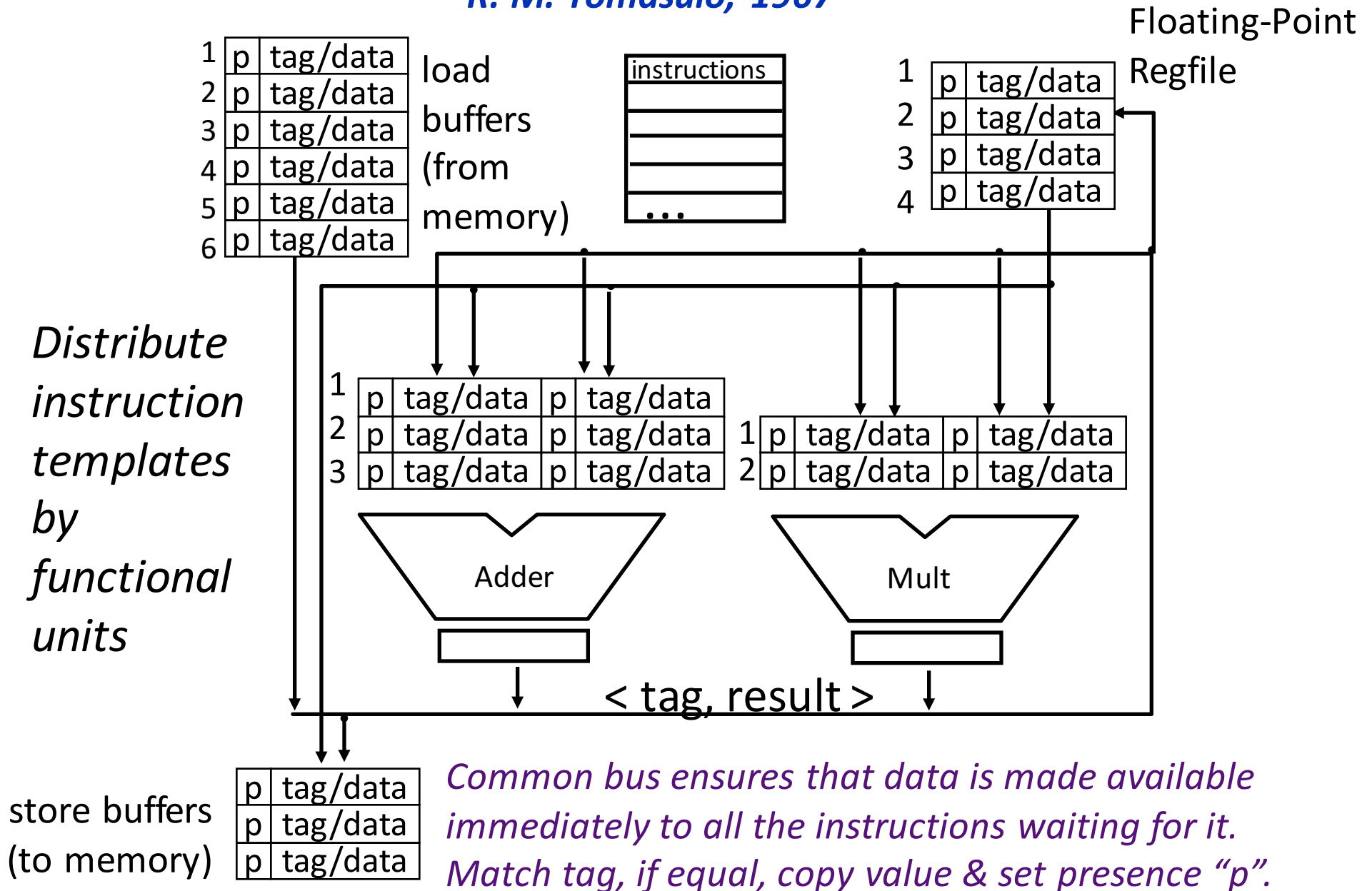
- "exec" bit is set when instruction begins execution
- When an instruction completes its "use" bit is marked free
- ptr₂ is incremented only if the "use" bit is marked free

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)

IBM 360/91 Floating-Point Unit

R. M. Tomasulo, 1967



Effectiveness?

Renaming and Out-of-order execution was first implemented in 1969 in IBM 360/91 but did not show up in the subsequent models until mid-Nineties.

Why ?

Reasons

1. Complex logic / cost
2. Memory latency a much bigger problem
3. Exceptions not precise!

One more problem needed to be solved

Control transfers

Precise Interrupts

It must appear as if an interrupt is taken between two instructions (say I_i and I_{i+1})

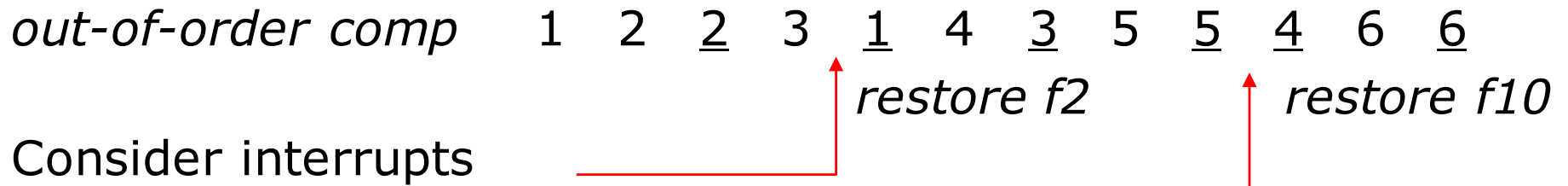
- the effect of all instructions up to and including I_i is totally complete
- no effect of any instruction after I_i has taken place

The interrupt handler either aborts the program or restarts it at I_{i+1} .

Effect on Interrupts

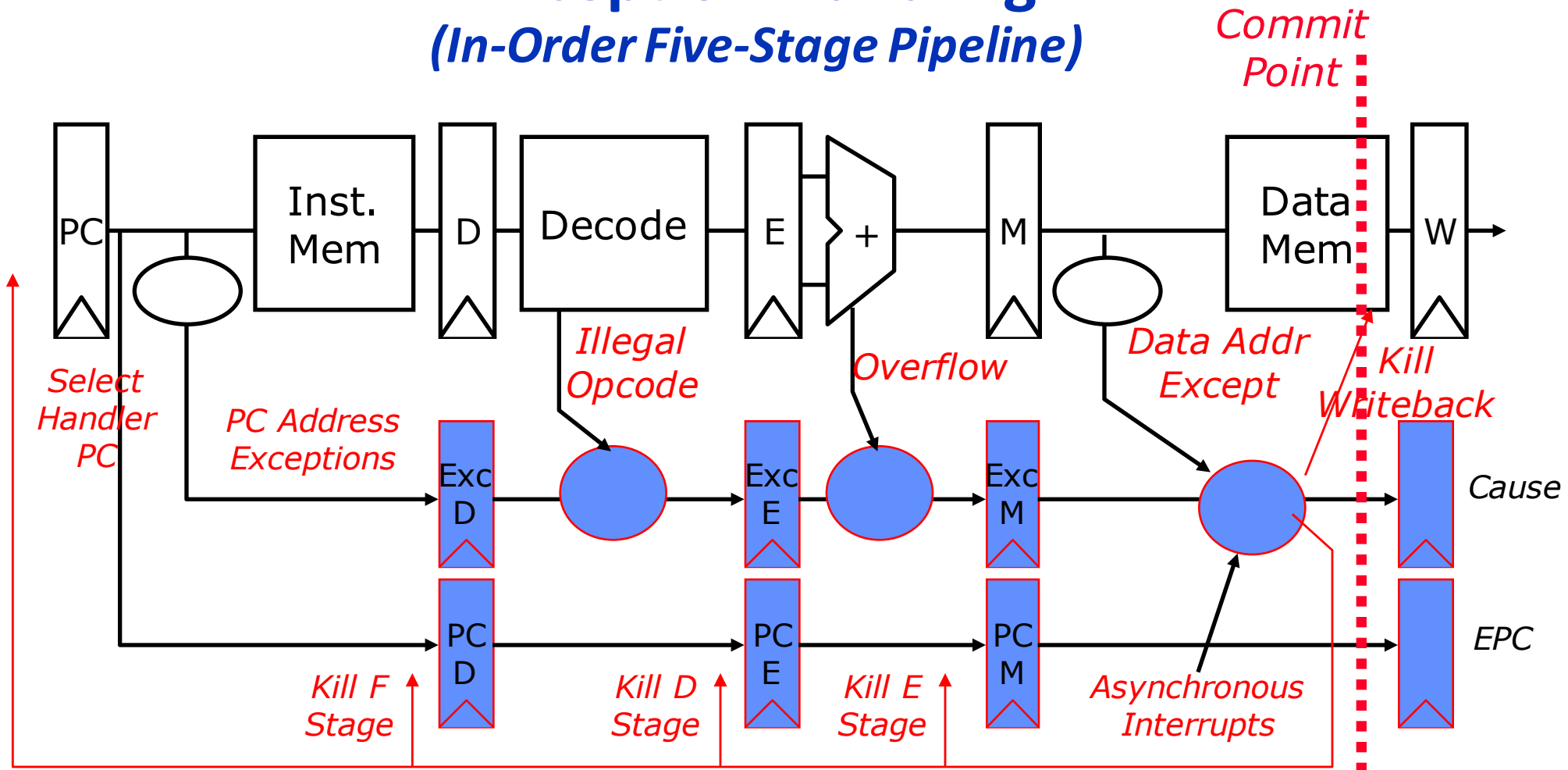
Out-of-order Completion

I_1	DIVD	f6,	f6,	f4
I_2	LD	f2,	45(r3)	
I_3	MULTD	f0,	f2,	f4
I_4	DIVD	f8,	f6,	f2
I_5	SUBD	f10,	f0,	f6
I_6	ADDD	f6,	f8,	f2



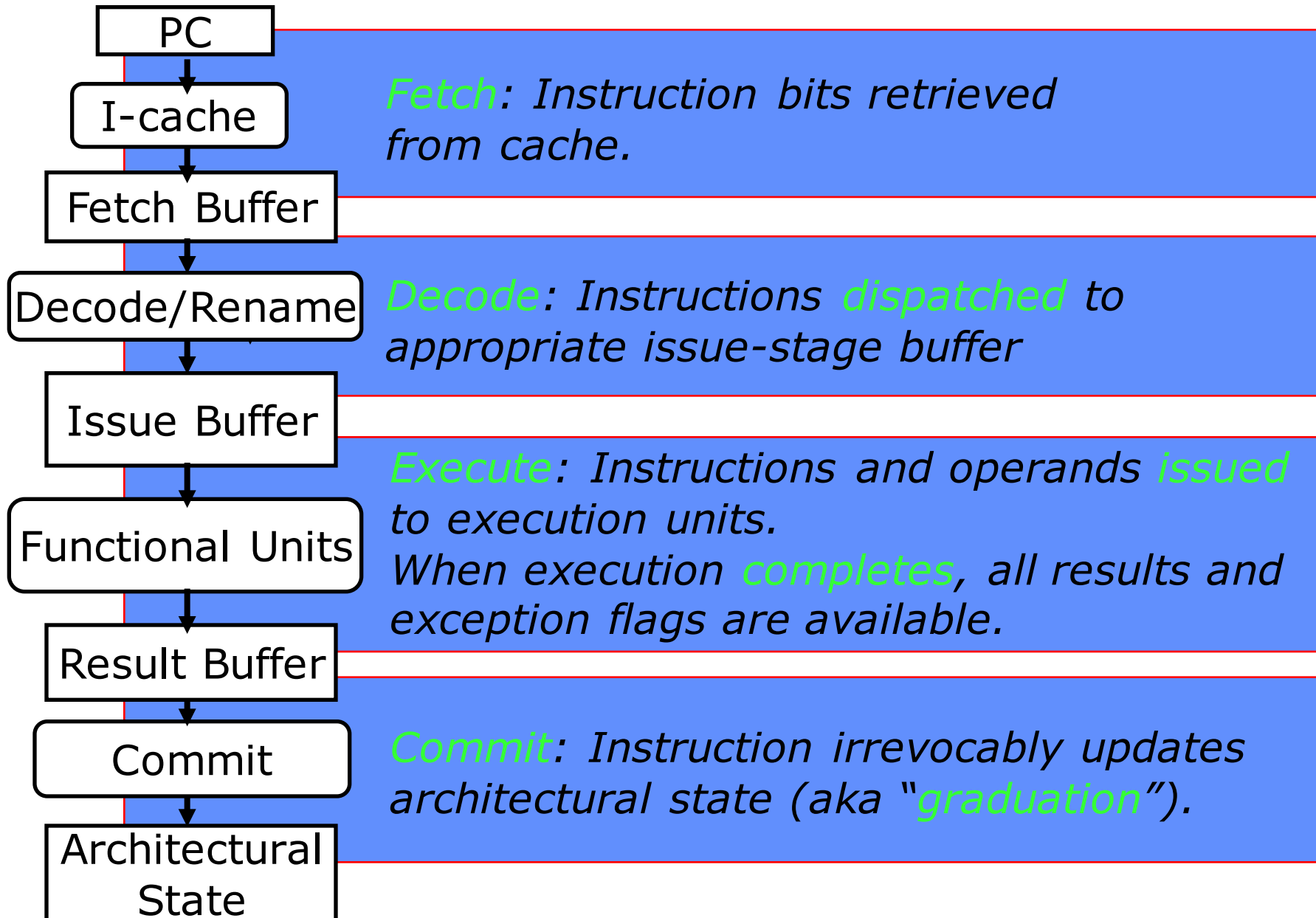
Precise interrupts are difficult to implement at high speed
- want to start execution of later instructions before
exception checks finished on earlier instructions

Exception Handling (In-Order Five-Stage Pipeline)

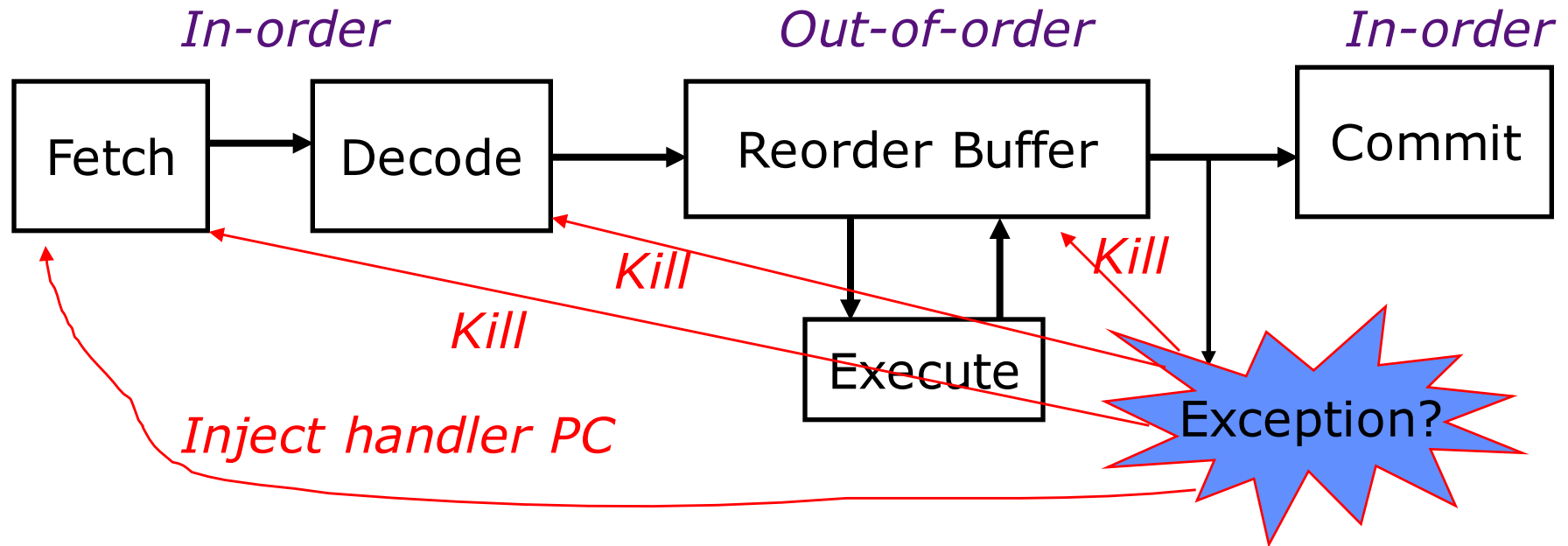


- Hold exception flags in pipeline until commit point (M stage)
- Exceptions in earlier pipe stages override later exceptions
- Inject external interrupts at commit point (override others)
- If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage

Phases of Instruction Execution



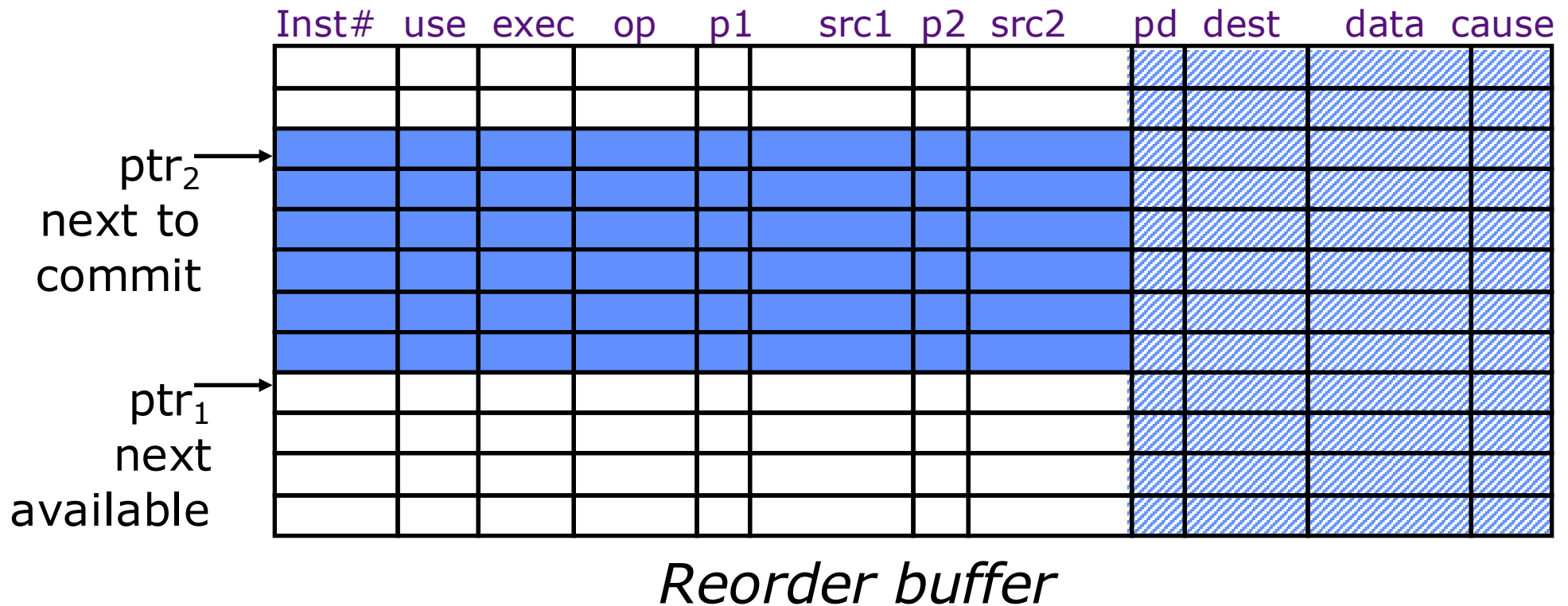
In-Order Commit for Precise Exceptions



- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order (\Rightarrow out-of-order completion)
- *Commit* (write-back to architectural state, i.e., regfile & memory, is in-order)

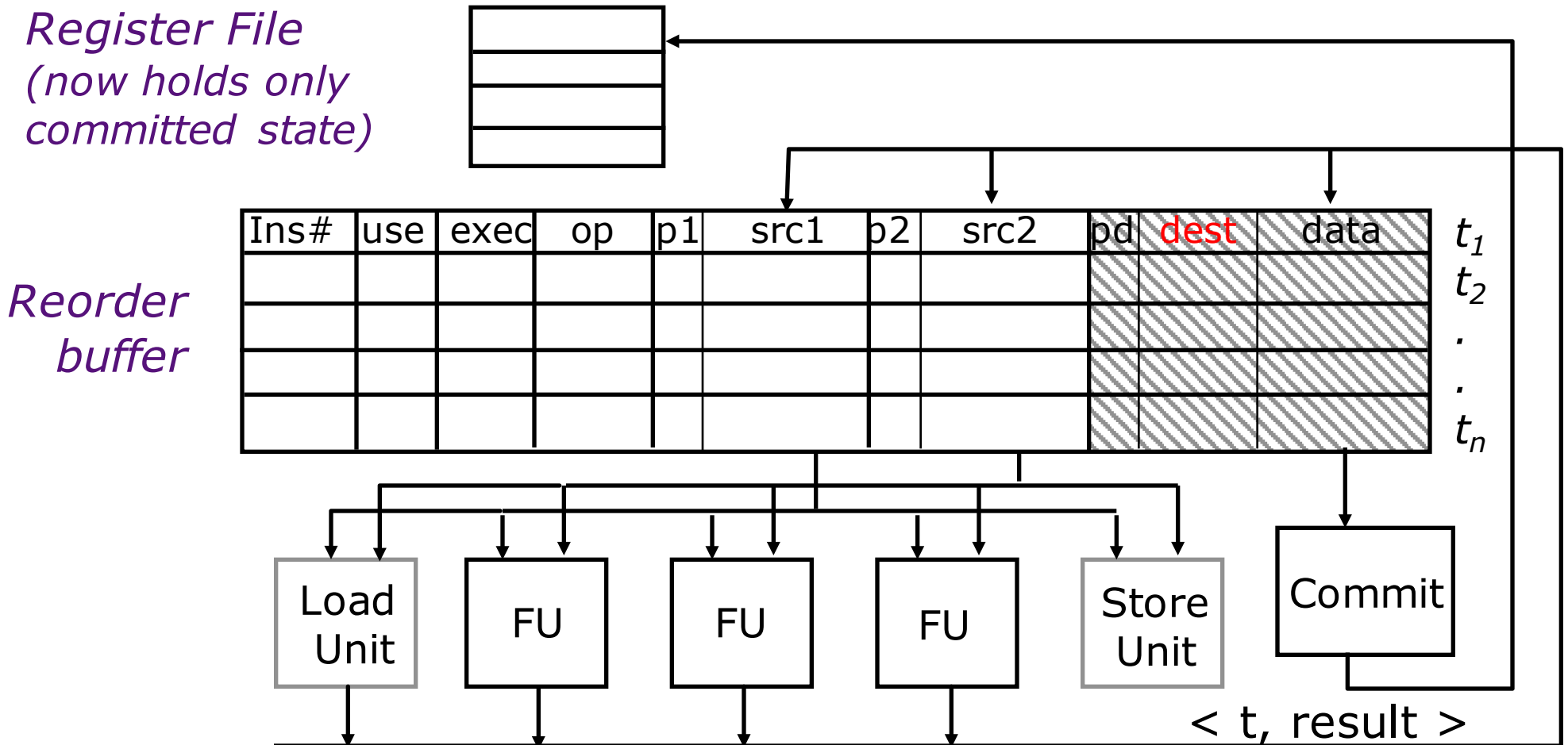
Temporary storage needed to hold results before commit (shadow registers and store buffers)

Extensions for Precise Exceptions



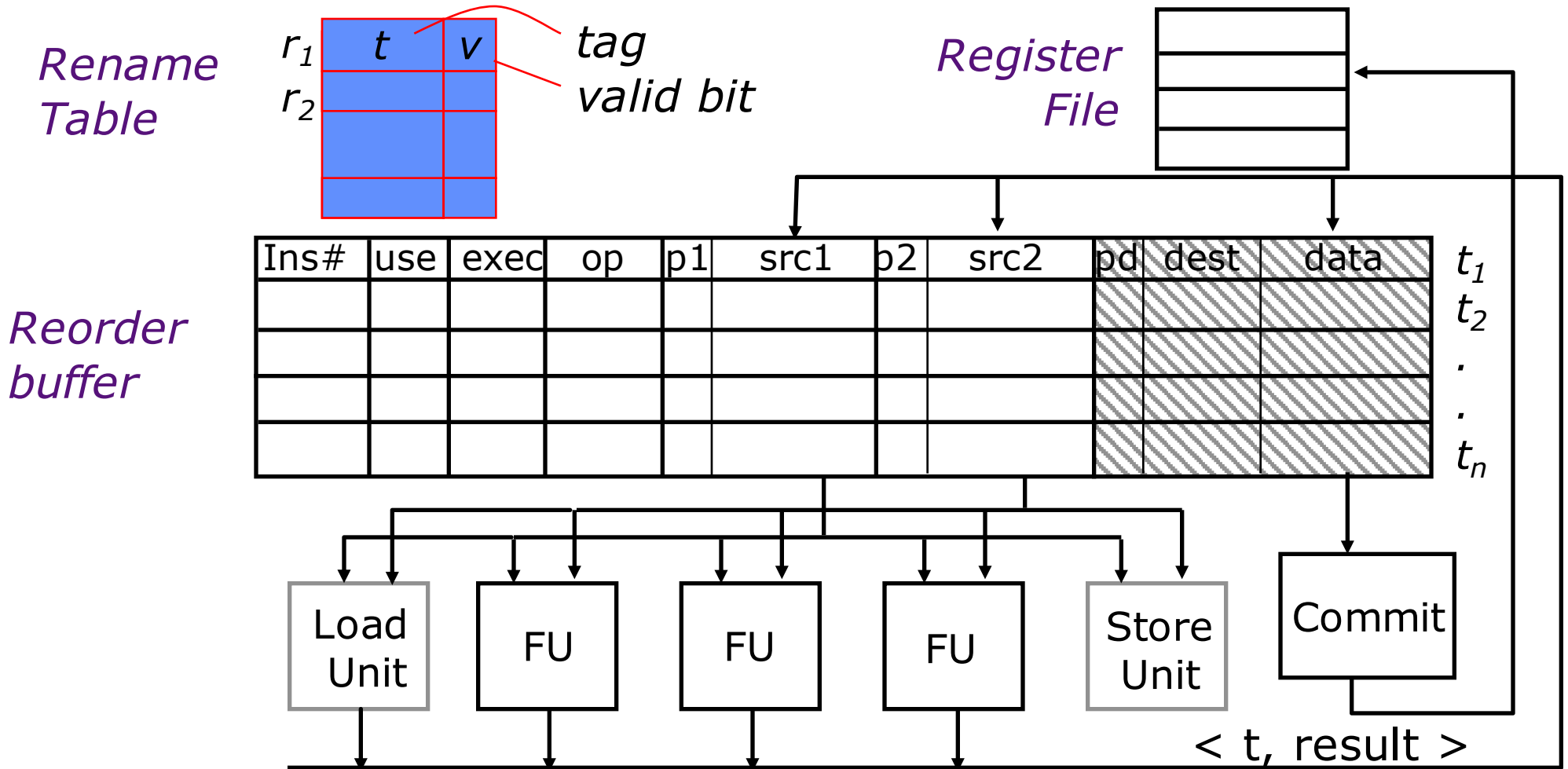
- add $\langle \text{pd}, \text{dest}, \text{data}, \text{cause} \rangle$ fields in the instruction template
- commit instructions to reg file and memory in program order \Rightarrow buffers can be maintained circularly
- on exception, clear reorder buffer by resetting $\text{ptr}_1 = \text{ptr}_2$
(stores must wait for commit before updating memory)

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.
 It needs to be cleared after each exception taken.

When else are valid bits cleared?

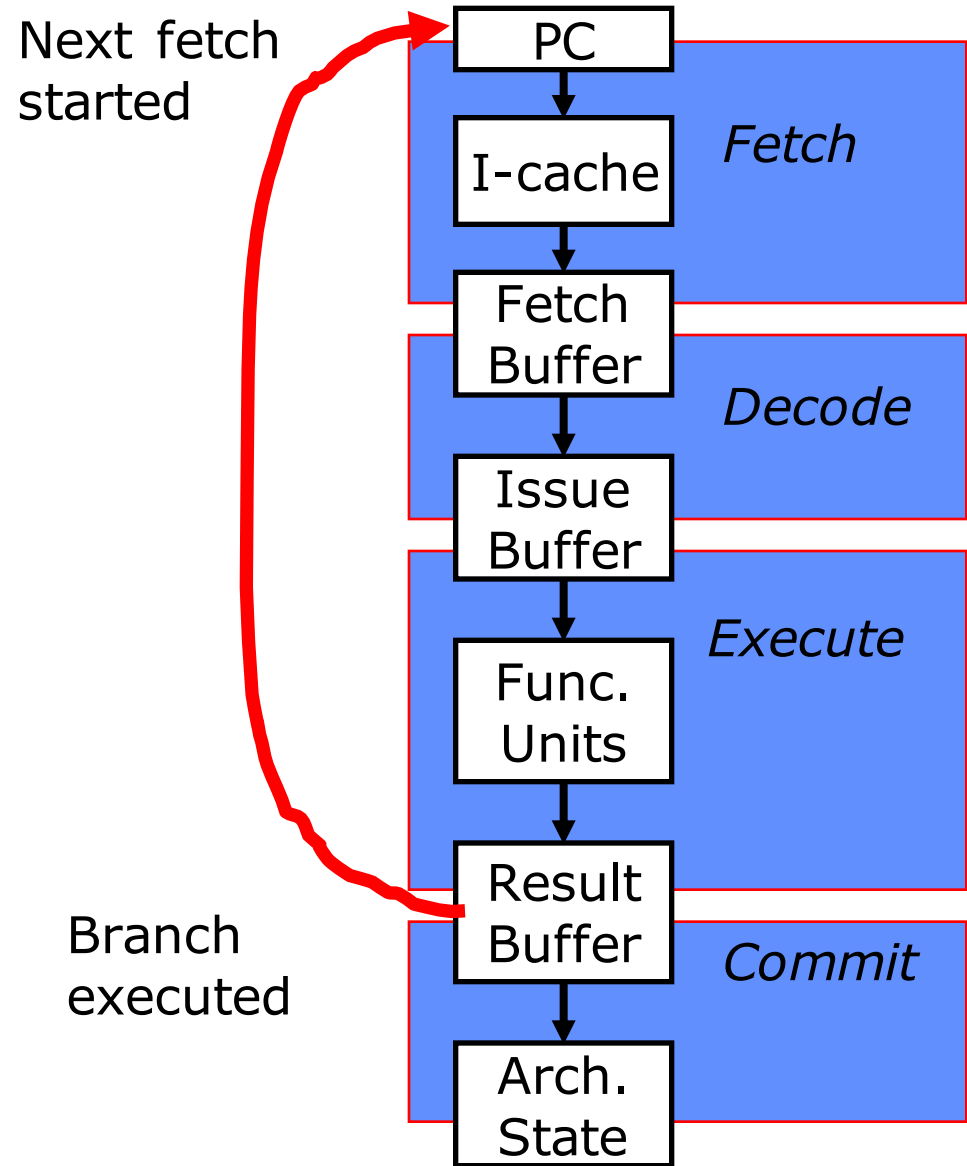
Control transfers

Control Flow Penalty

*Modern processors may have
> 10 pipeline stages between
next PC calculation and branch
resolution !*

*How much work is lost if
pipeline doesn't follow
correct instruction flow?*

~ Loop length x pipeline width



Mispredict Recovery

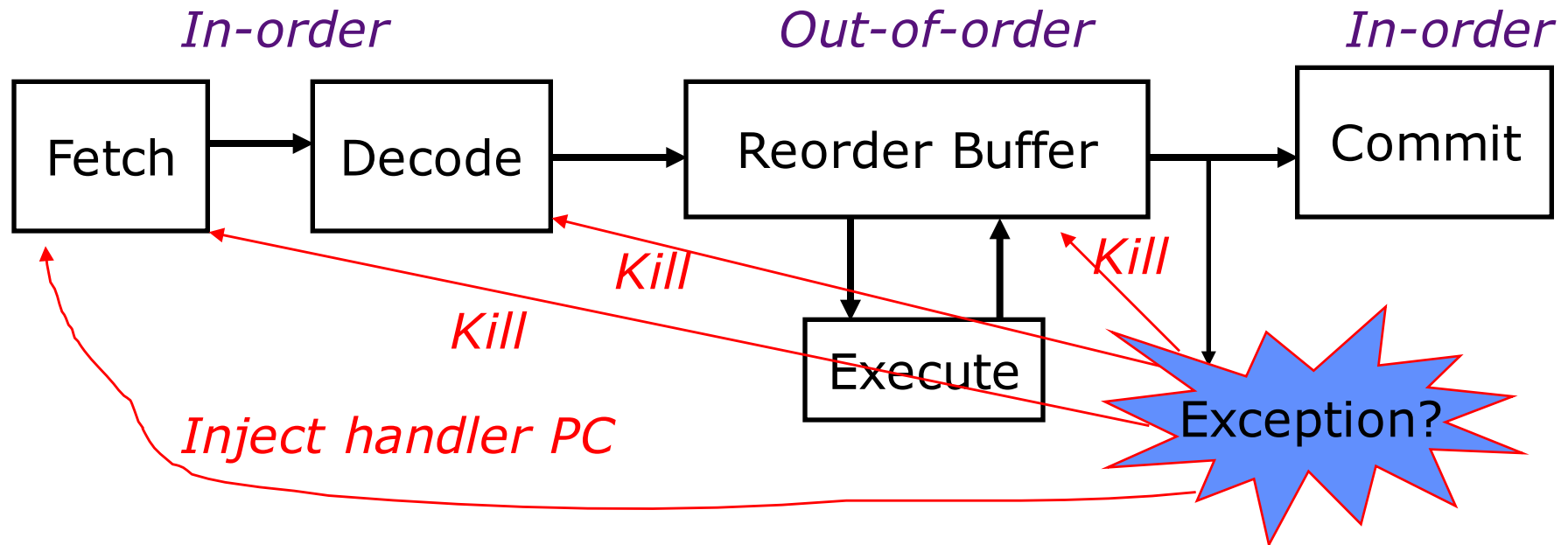
In-order execution machines:

- Assume no instruction issued after branch can write-back before branch resolves
- Kill all instructions in pipeline behind mispredicted branch

Out-of-order execution?

- Multiple instructions following branch in program order can complete before branch resolves

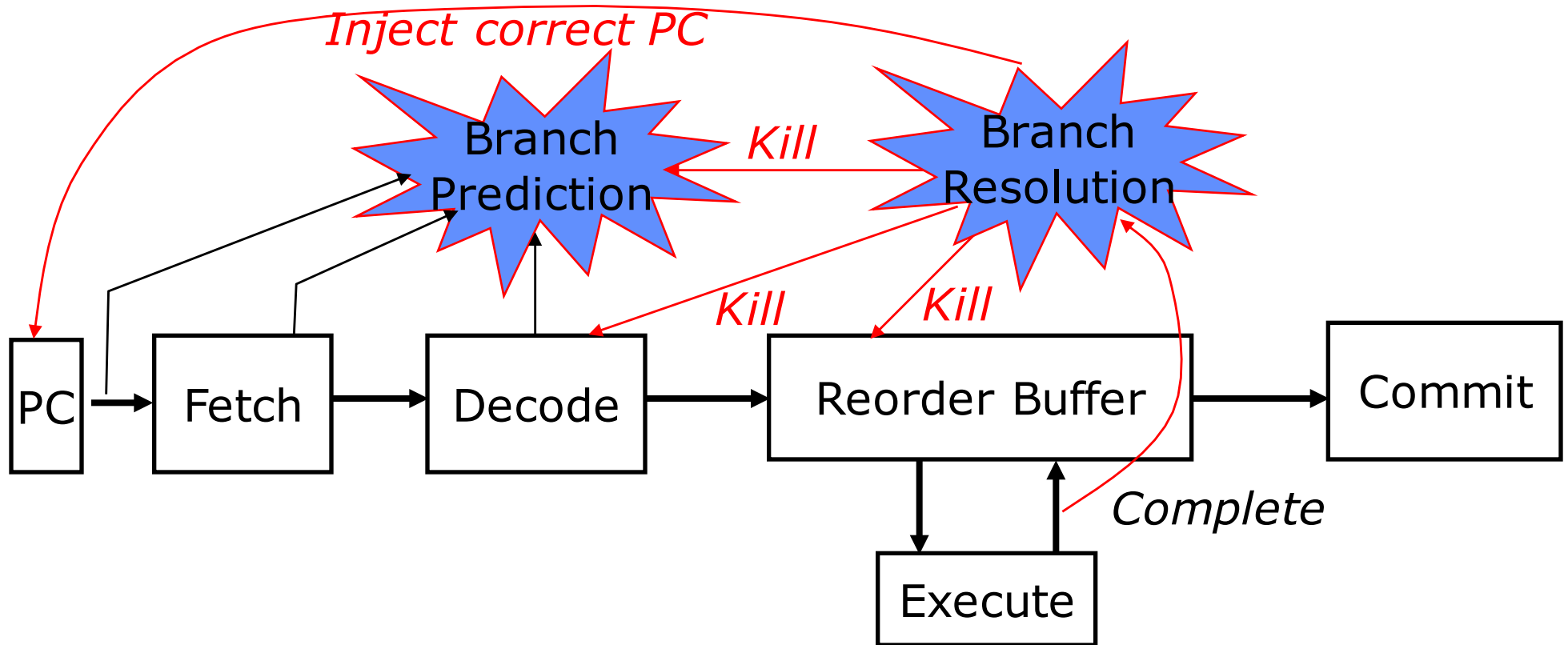
In-Order Commit for Precise Exceptions



- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order (\Rightarrow out-of-order completion)
- *Commit* (write-back to architectural state, i.e., regfile & memory, is in-order)

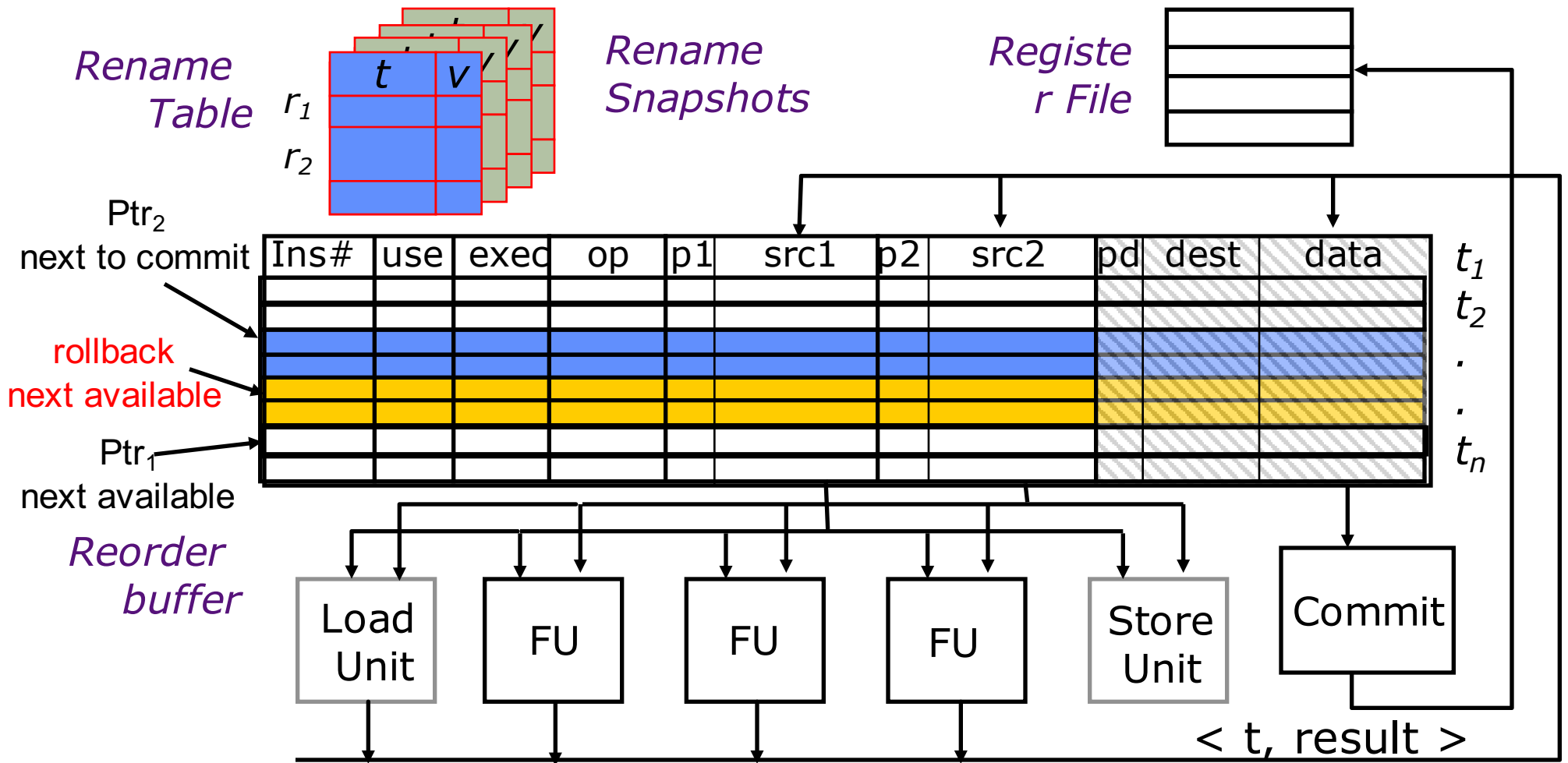
Temporary storage needed in ROB to hold results before commit

Branch Misprediction in Pipeline



- Can have multiple unresolved branches in ROB
- Can resolve branches out-of-order by killing all the instructions in ROB that follow a mispredicted branch

Recovering ROB/Renaming Table



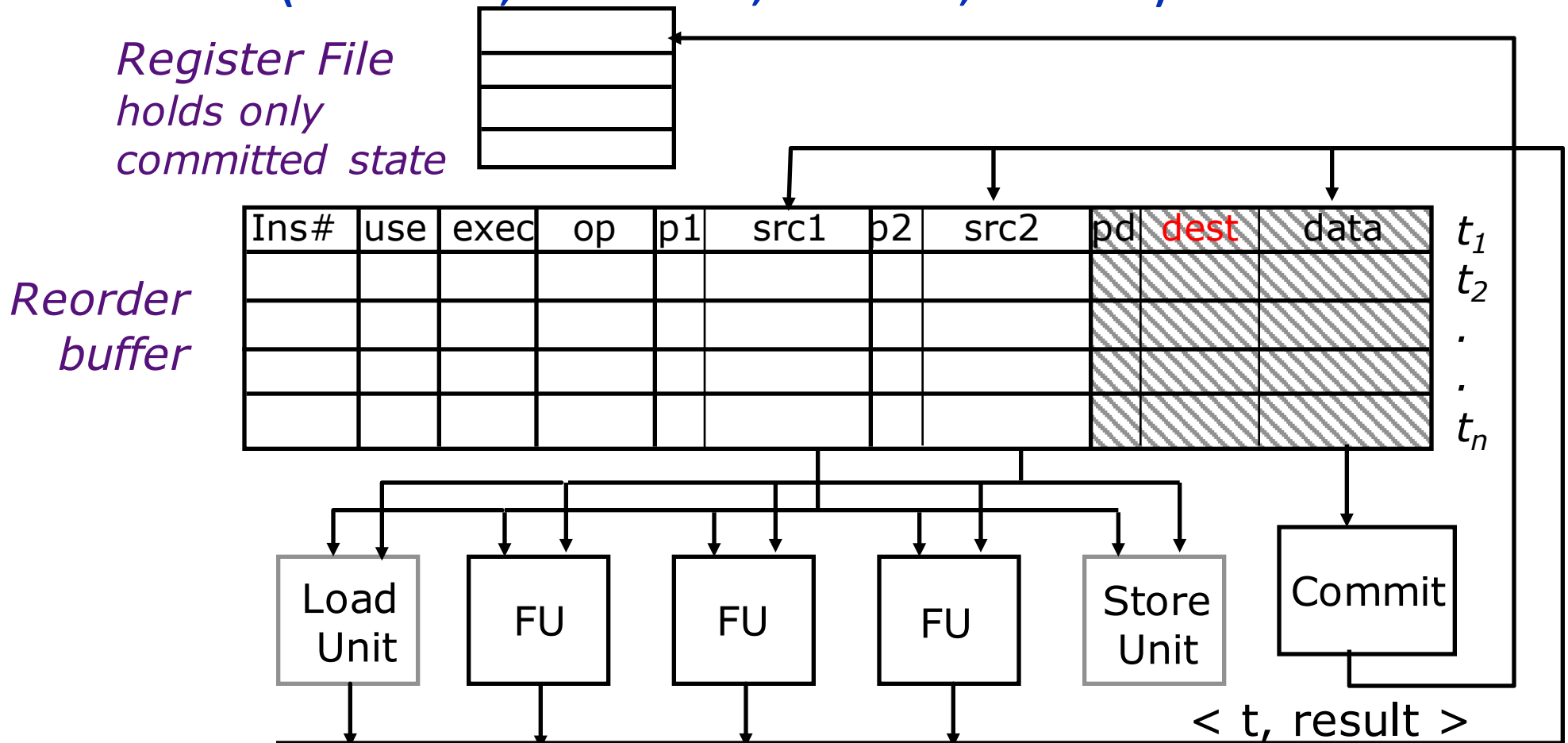
Take snapshot of register rename table at each predicted branch, recover earlier snapshot if branch mispredicted

Important Points on OOO Execution

- Precise interrupts are provided by committing (graduating) instructions in-order.
- Any instruction not yet committed is considered “speculative” because it might be killed because of:
 - Interrupt on an older instruction
 - Branch misprediction
- Instructions (and their result values) are held in ROB waiting to commit

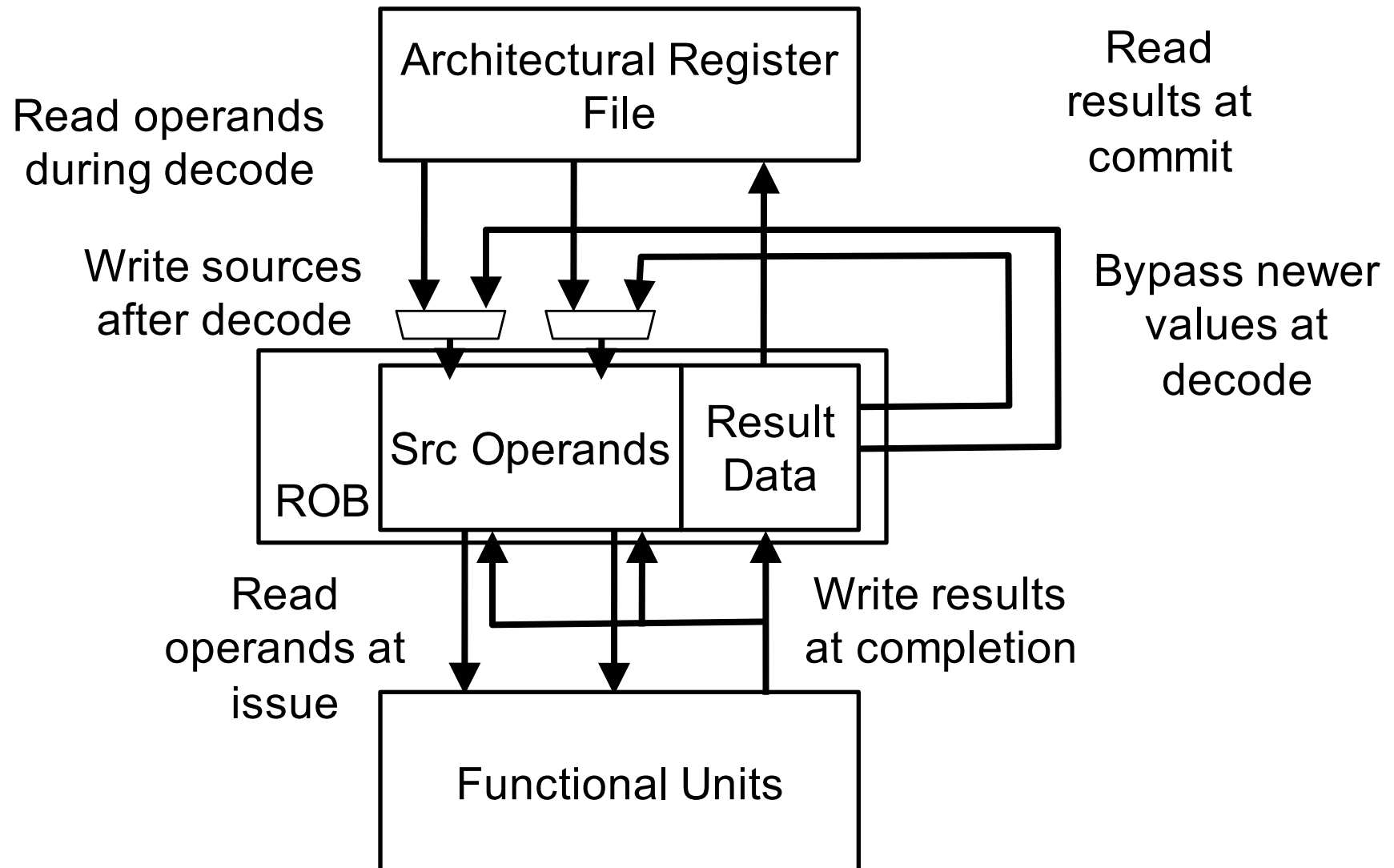
“Data-in-ROB” Design

(HP PA8000, Pentium Pro, Core2Duo, Nehalem)



- On dispatch into ROB, ready sources can be in regfile or in ROB dest (copied into src1/src2 if ready before dispatch)
- On completion, write to dest field and broadcast to src fields.
- On issue, read from ROB src fields
- Instructions are committed to the register file in order
- *Instruction slots (i.e. tags) are recycled after commit*

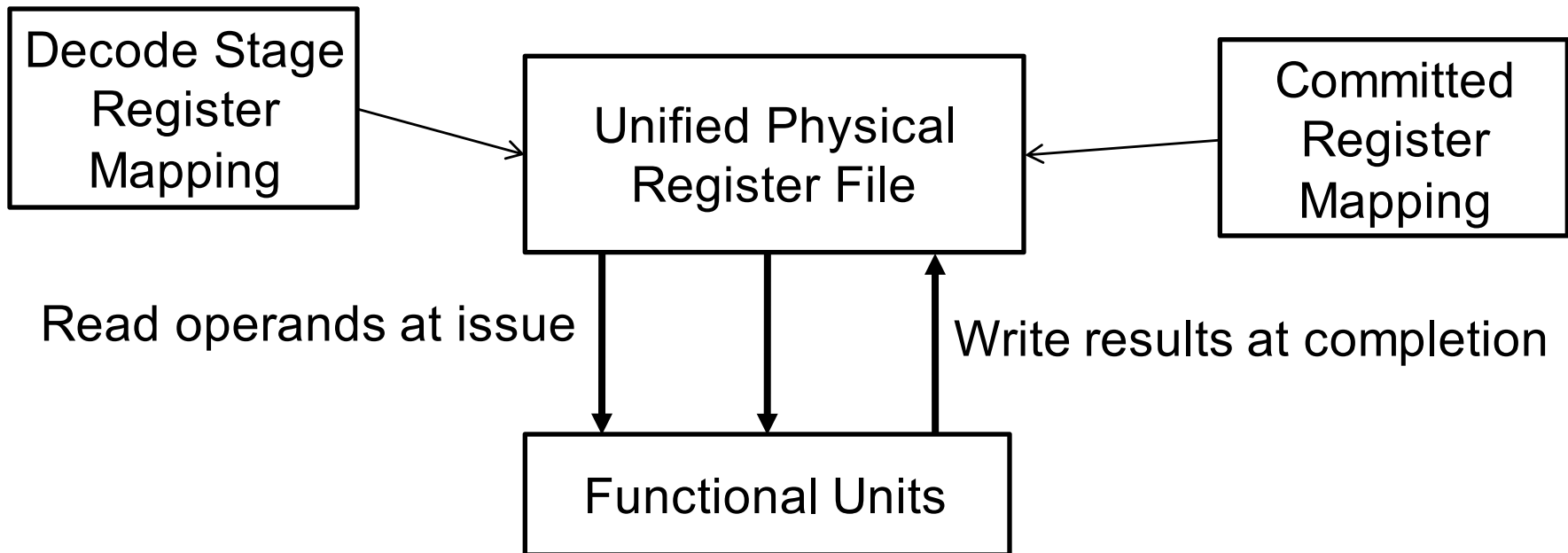
Data Movement in Data-in-ROB Design



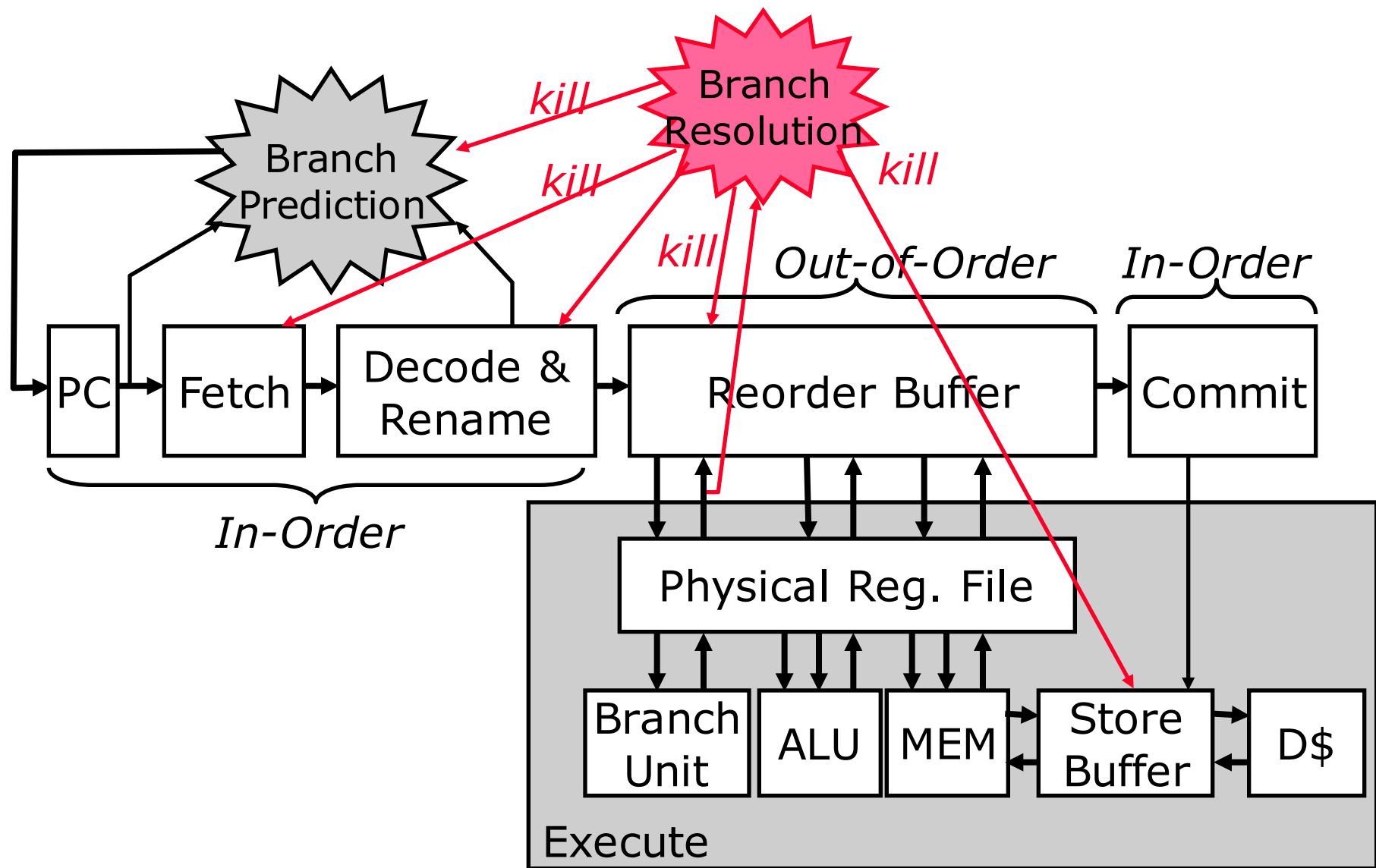
Unified Physical Register File

(MIPS R10K, Alpha 21264, Intel Pentium 4 & Sandy 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 the mapping of architectural register to physical register, no data movement



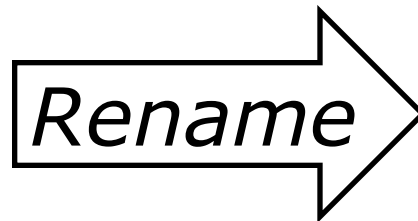
Pipeline Design with Physical Regfile



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 write 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

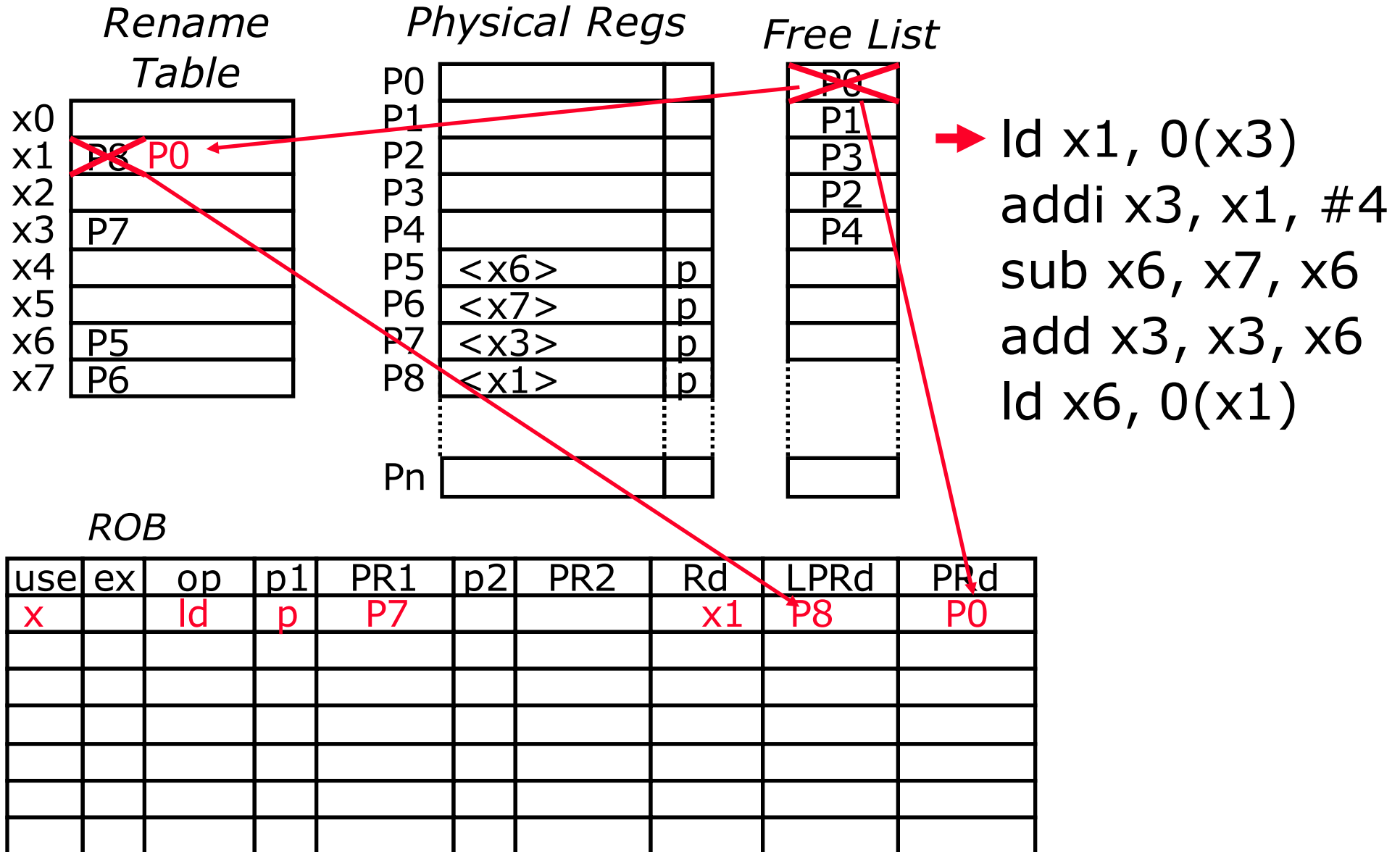
```
ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)
```

ROB

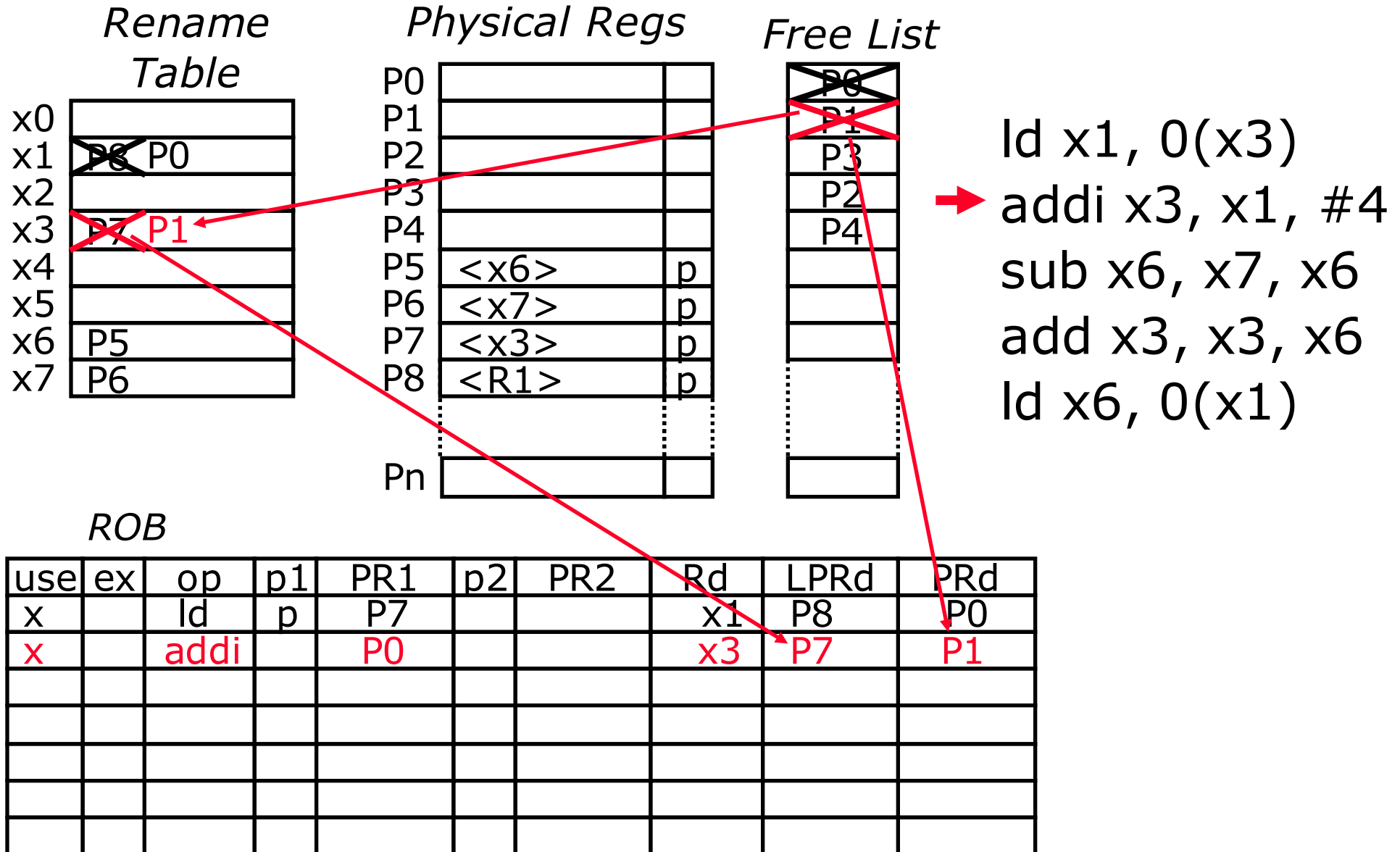
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd

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

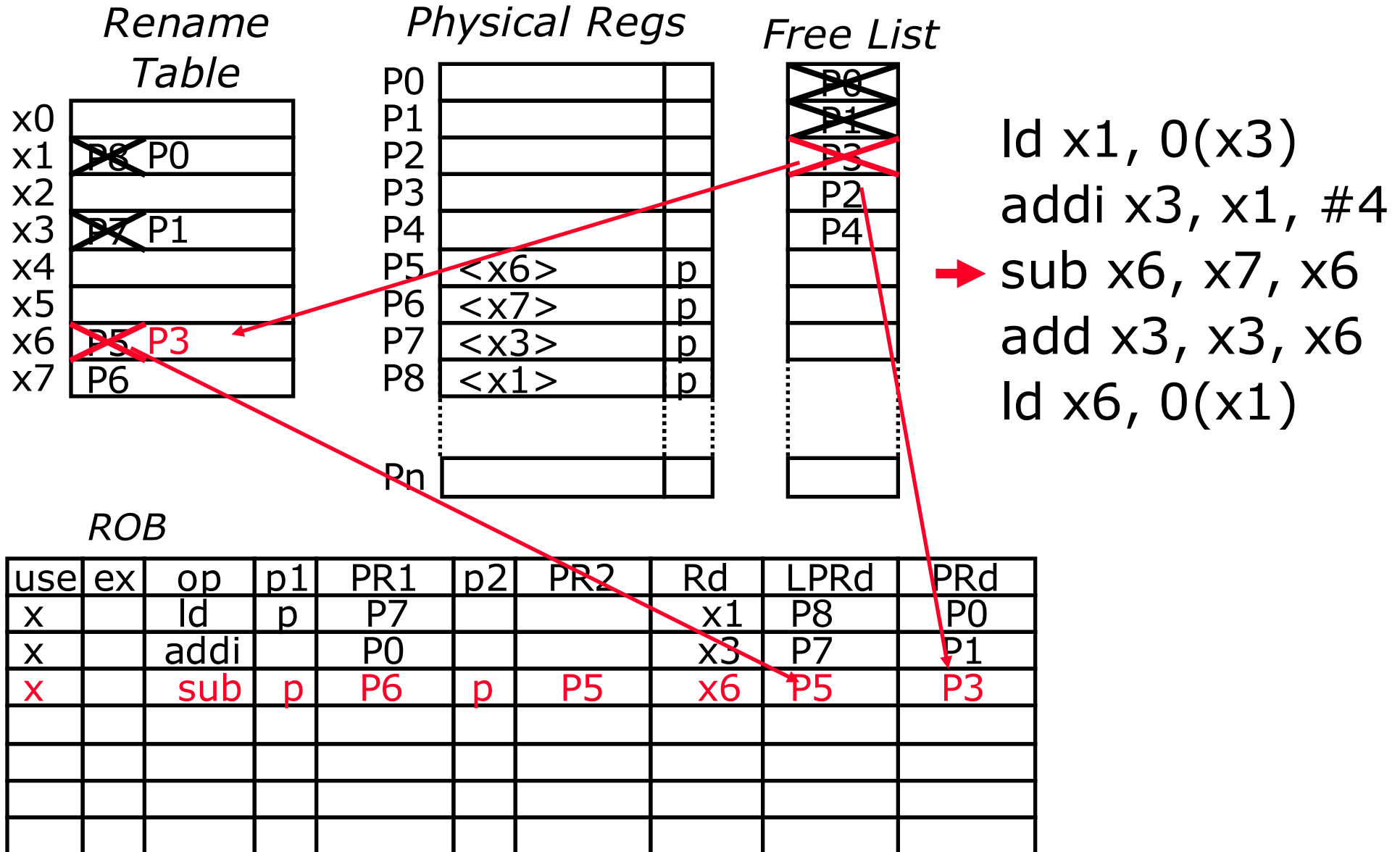
Physical Register Management



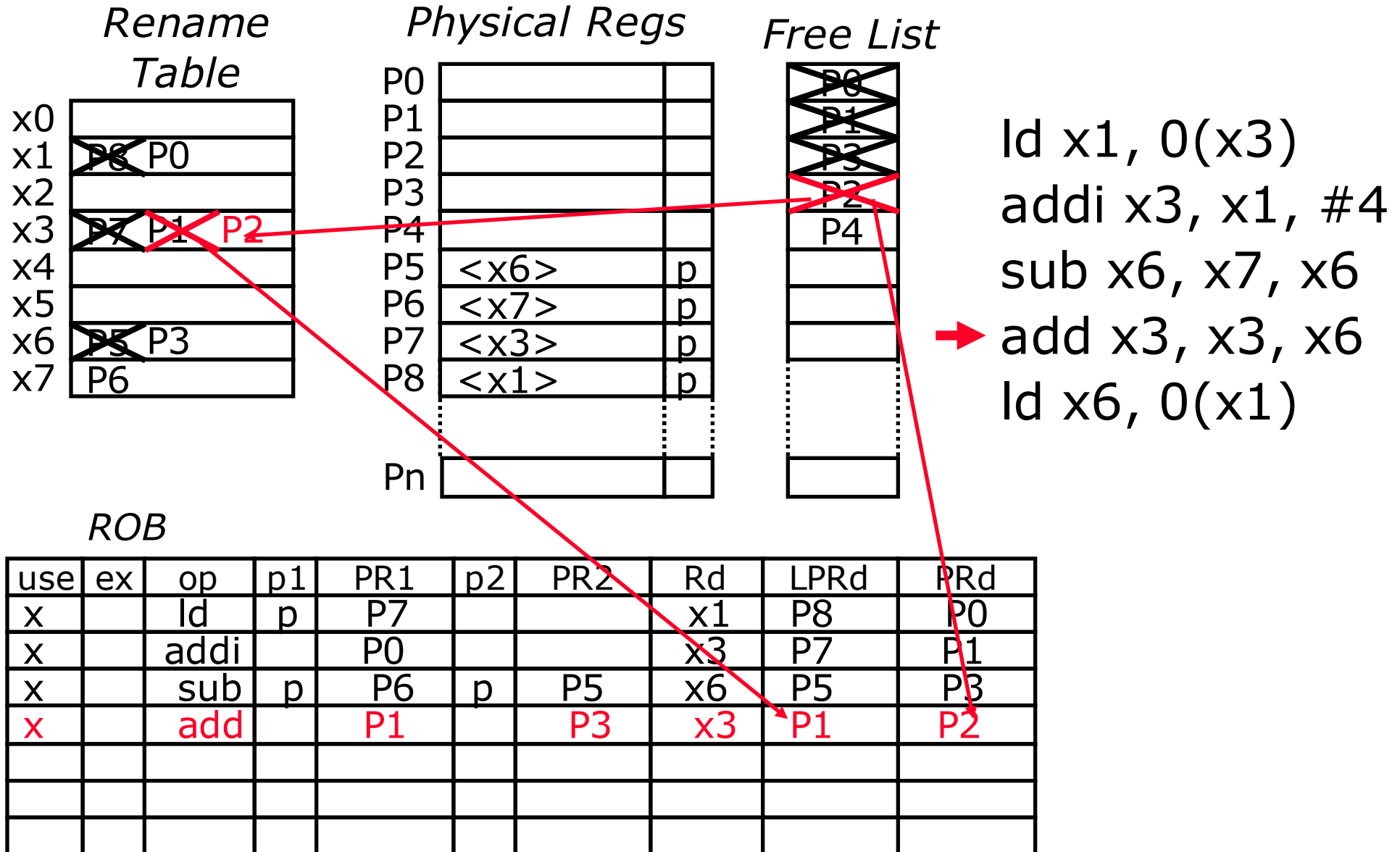
Physical Register Management



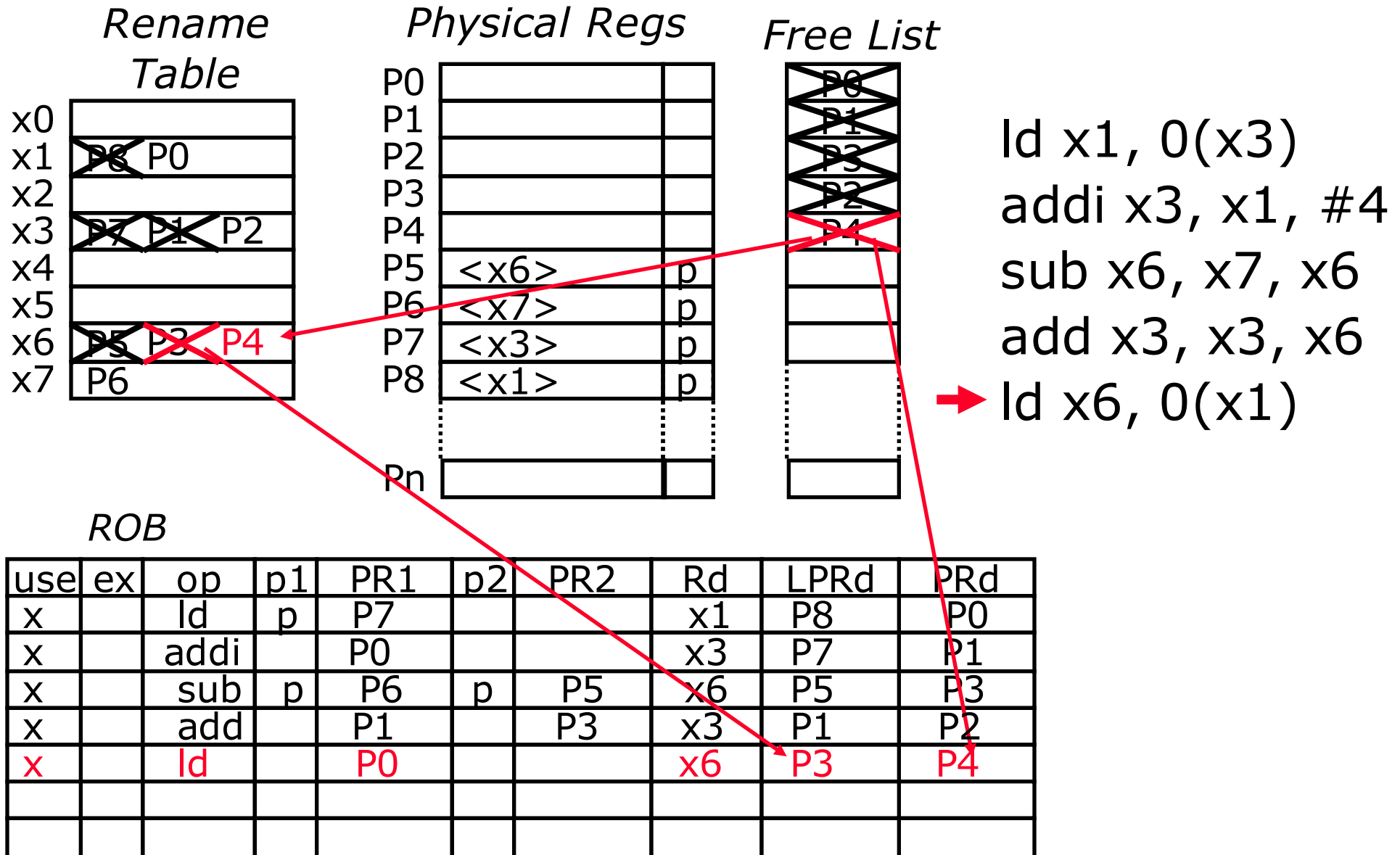
Physical Register Management



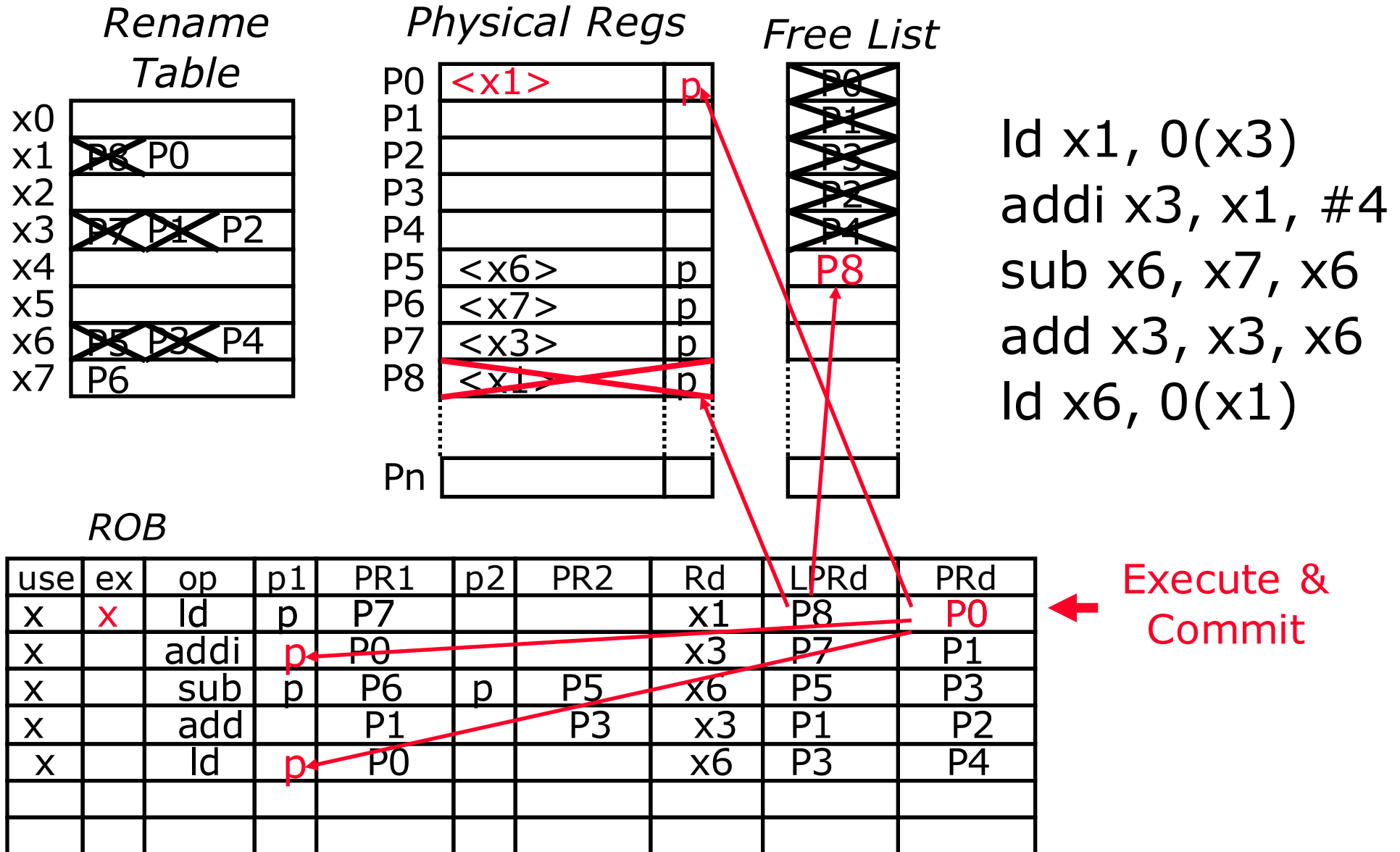
Physical Register Management



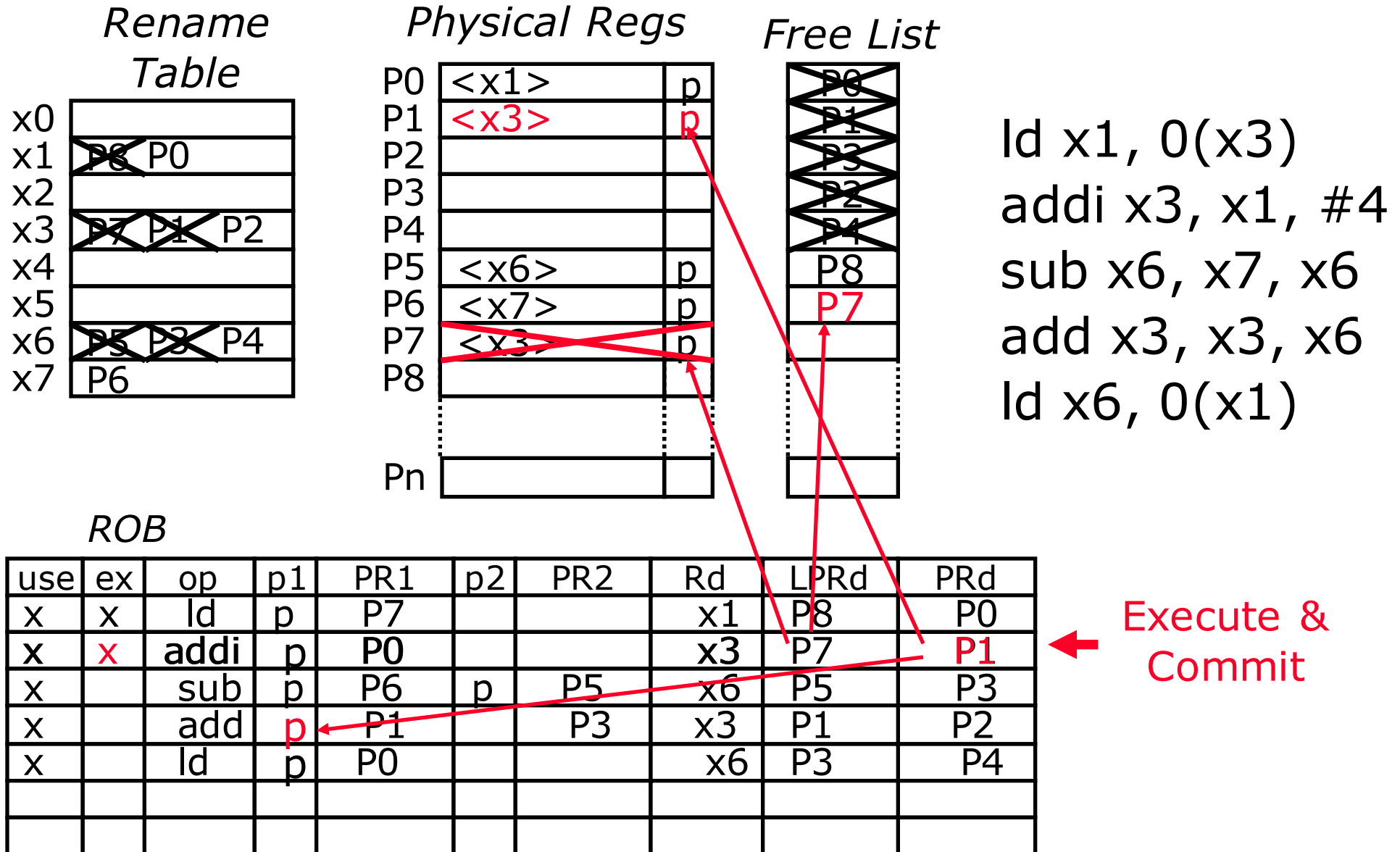
Physical Register Management



Physical Register Management



Physical Register Management



Acknowledgements

- These slides contain material developed and copyright by:
 - Arvind (MIT)
 - Krste Asanovic (MIT/UCB)
 - Joel Emer (Intel/MIT)
 - James Hoe (CMU)
 - John Kubiatowicz (UCB)
 - David Patterson (UCB)
- MIT material derived from course 6.823
- UCB material derived from course CS252