

Speculative Execution

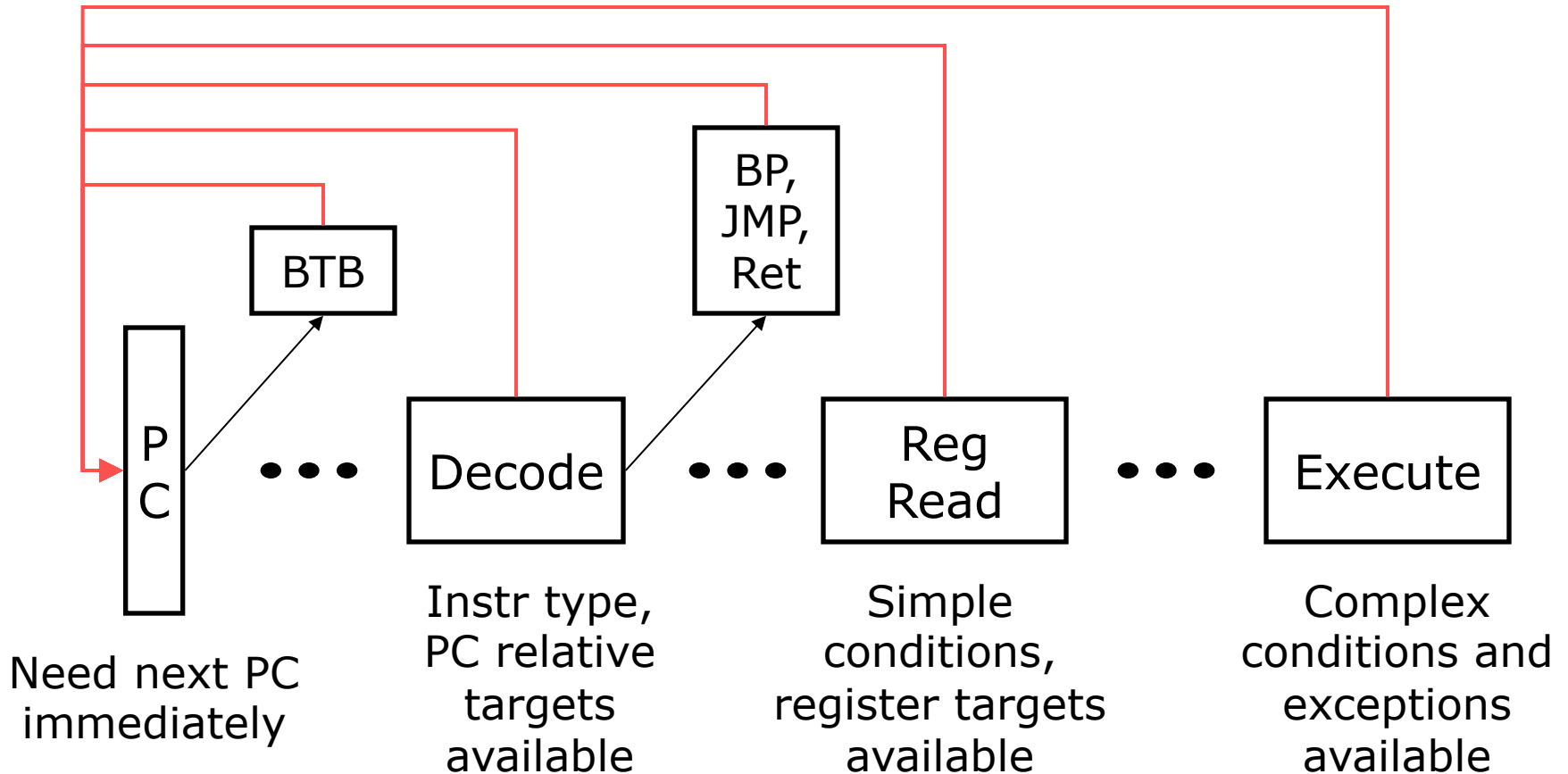
Joel Emer

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

What does this
WW2 poster
have in common
with pipelined
processors?



Overview of branch prediction



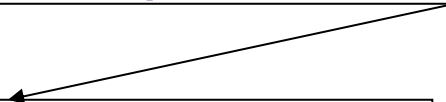
Must speculation check always be correct?

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?

Value Management Strategies

Greedy (or Eager) Update:

- Update value in place, and
- Provide means to reconstruct old values for recovery
 - often this is a log of old values

Lazy Update:

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

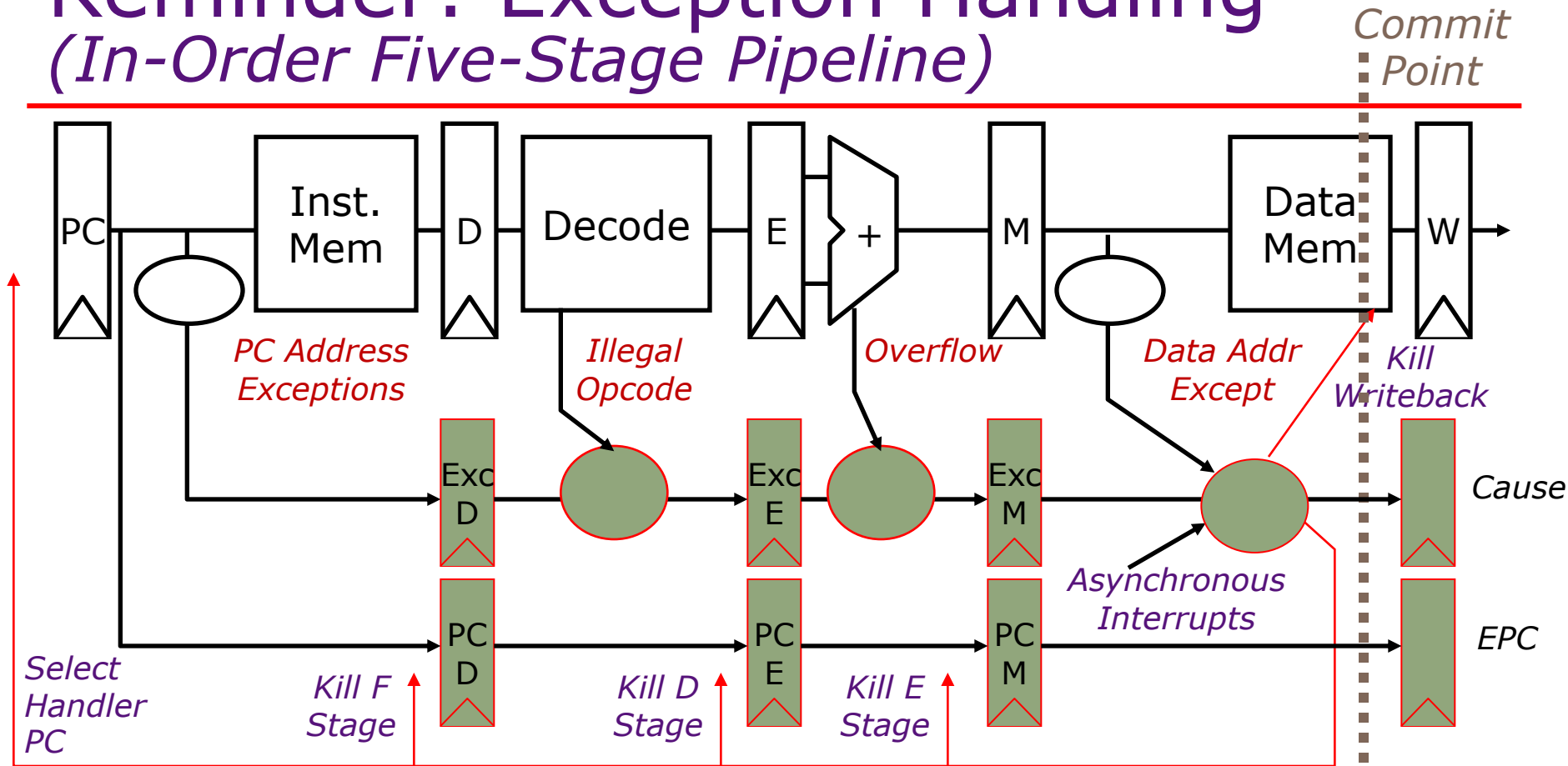
Why leave an old value in place?

Exceptions

- Exceptions create a dependence on the value of the next PC
- Options for handling this dependence:
 - Stall
 - Bypass
 - Find something else to do
 - Change the architecture
 - Speculate!
- How can we handle rollback on mis-speculation?

- Note: earlier exceptions must override later ones

Reminder: Exception Handling (In-Order Five-Stage Pipeline)



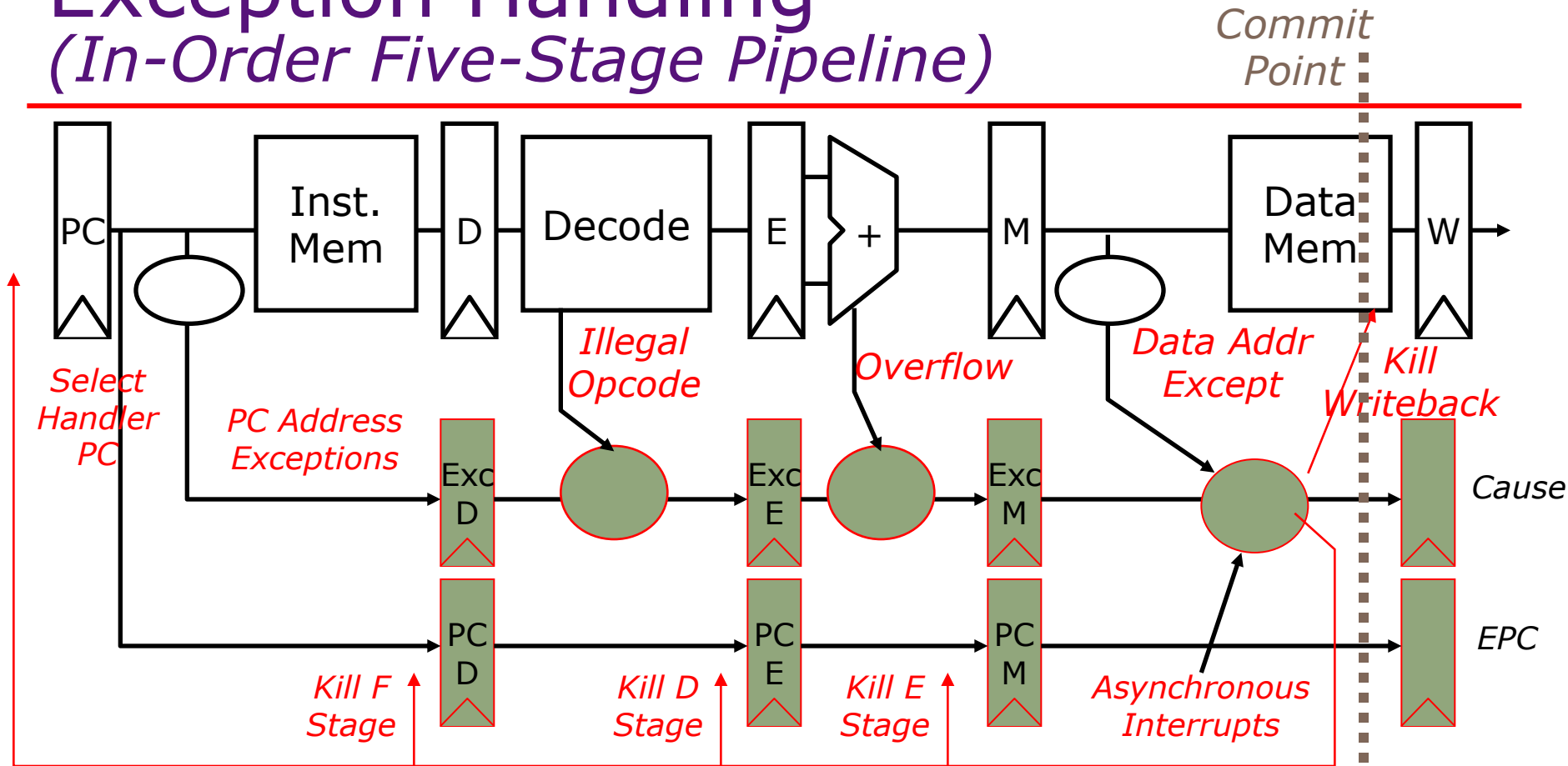
Hold exception flags in pipeline until commit point (M stage)

- If exception at commit:
 - update Cause/EPC registers
 - kill all stages
 - fetch at handler PC

Inject external interrupts at commit point

Exception Handling

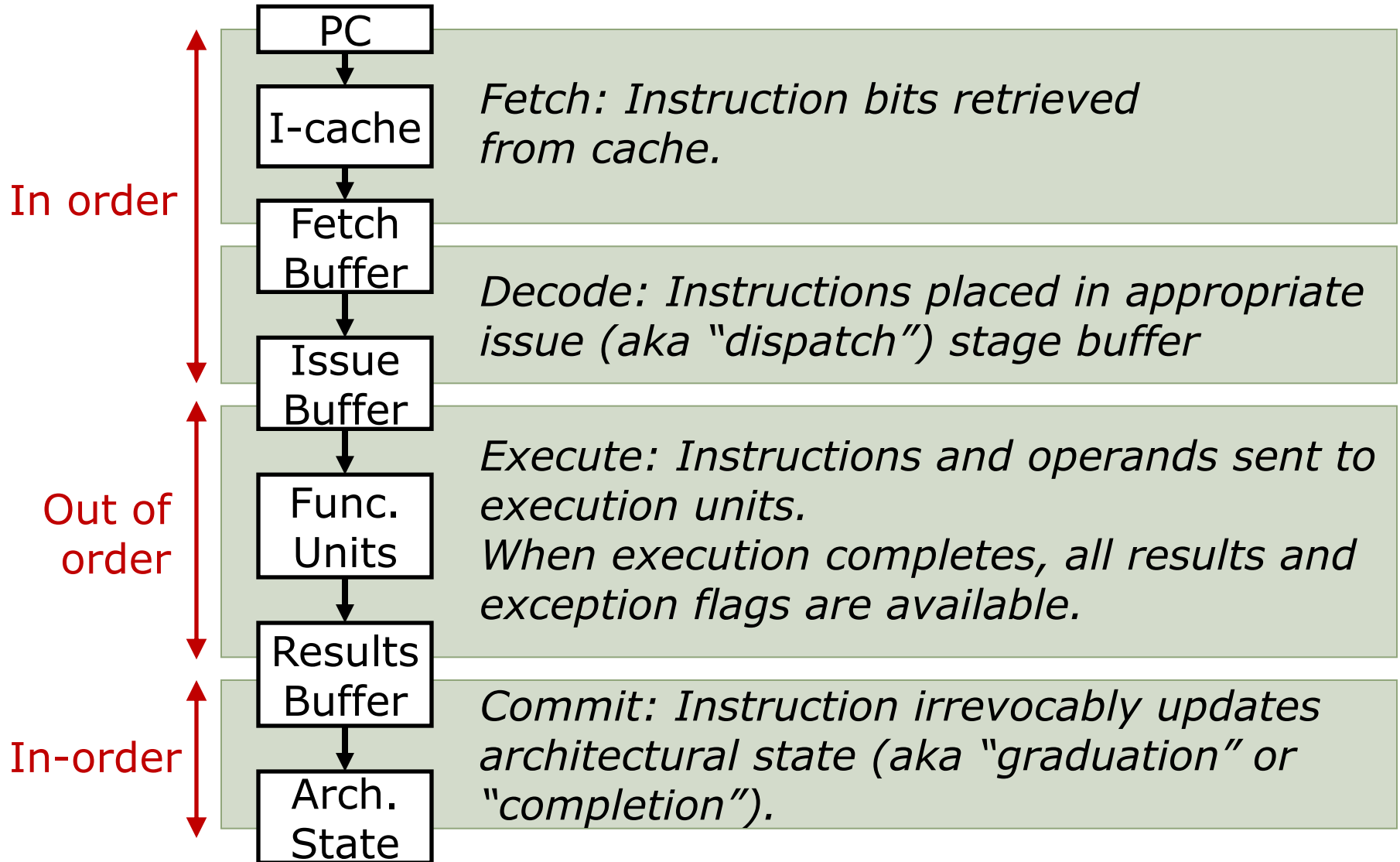
(In-Order Five-Stage Pipeline)



Strategy for Registers?

Strategy for PC?

Phases of Instruction Execution



Misprediction Recovery

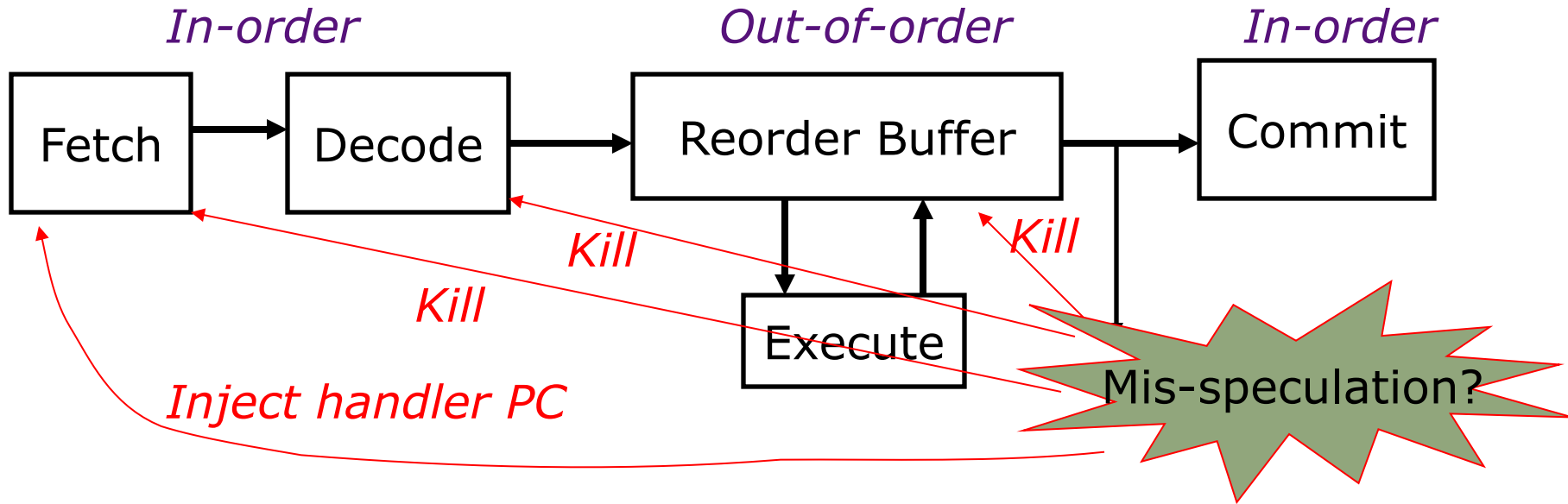
In-order execution machines:

- 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 exception in program order can generate new values before exception resolves

In-Order Commit for Mis-speculation Recovery



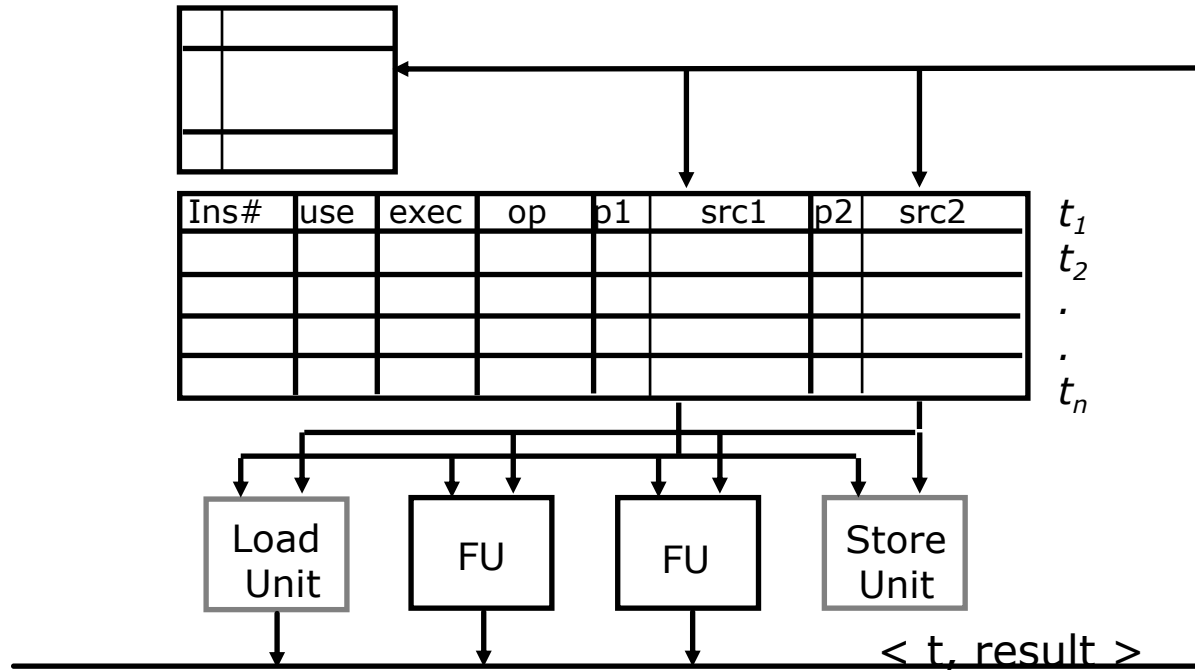
- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order (\Rightarrow out-of-order completion)
- *Commit* (write-back to architectural state, i.e., regfile & memory) is in-order

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

Data-Driven Execution

*Renaming
table &
reg file*

*Reorder
buffer*

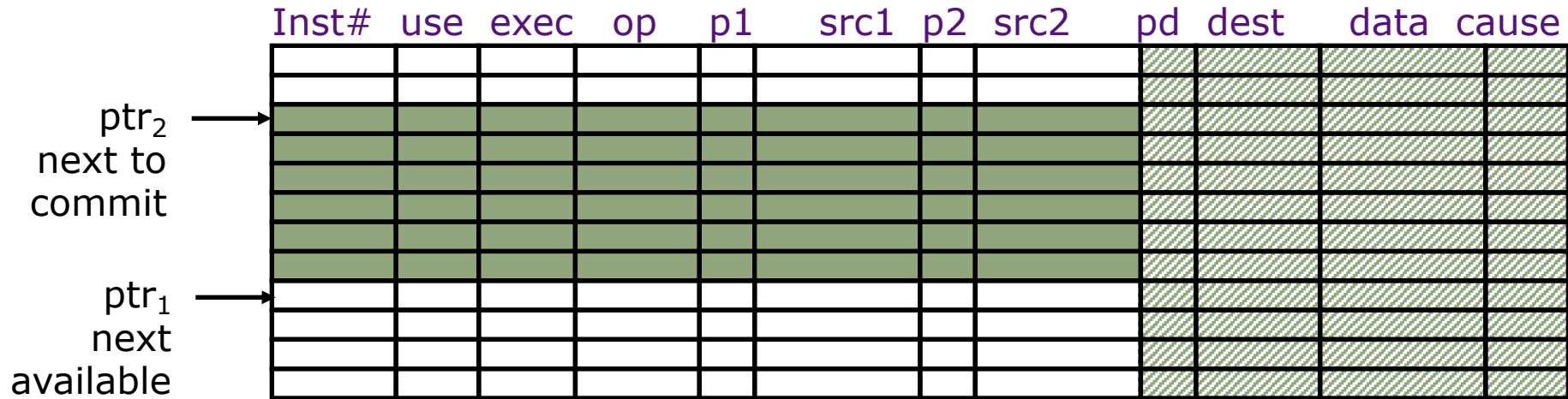


Basic Operation:

- 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

Update strategy?

Extensions for Mis-speculation Recovery



Reorder buffer

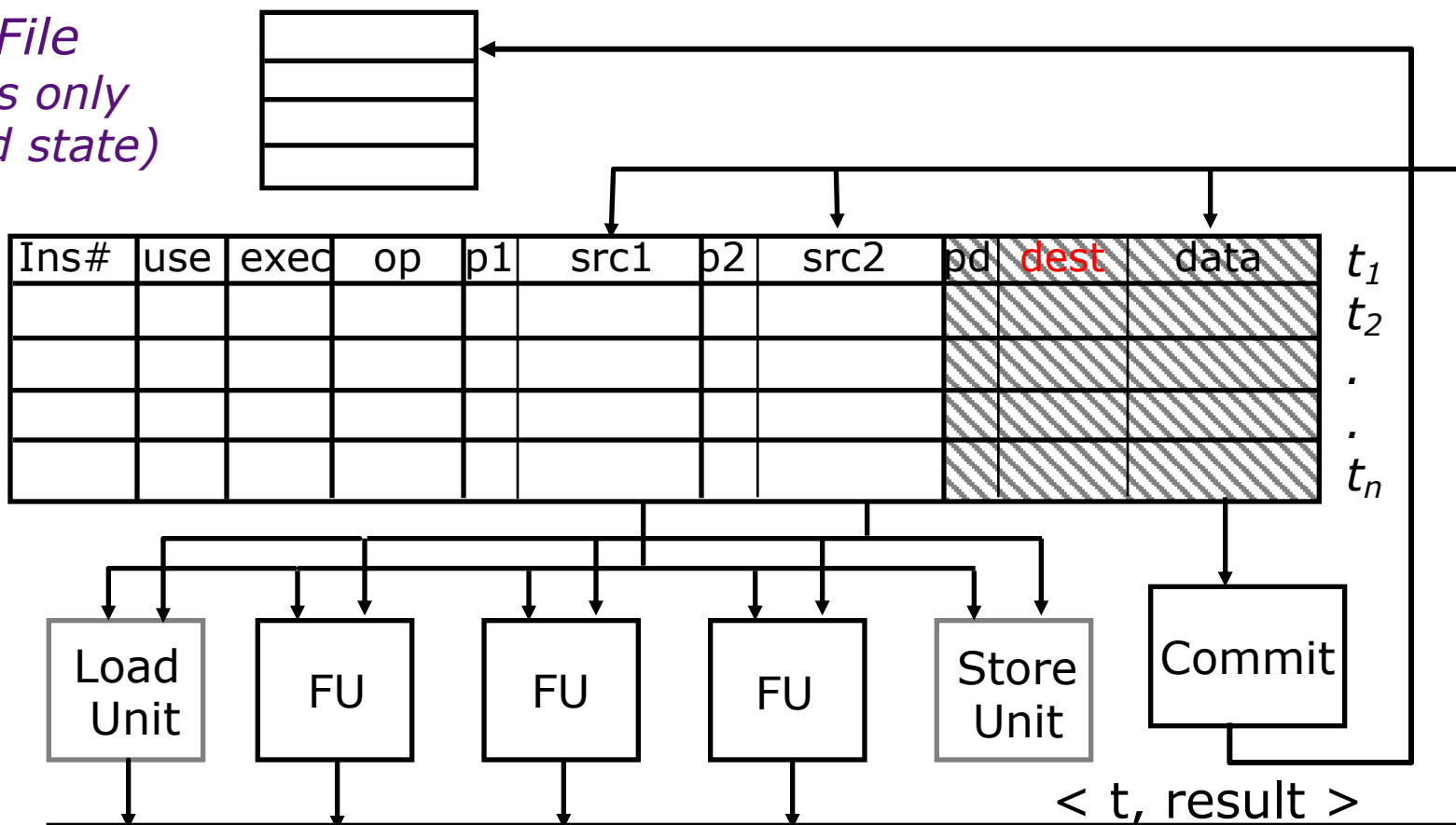
- add $\langle \text{pd}, \text{dest}, \text{data}, \text{cause} \rangle$ fields in the instruction template
- commit instructions to reg file and memory in program order \Rightarrow buffers can be maintained circularly
- on exception, clear reorder buffer by resetting $\text{ptr}_1 = \text{ptr}_2$
(stores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

What is the update policy of registers?

Rollback and Renaming

Register File
(now holds only committed state)

Reorder
buffer

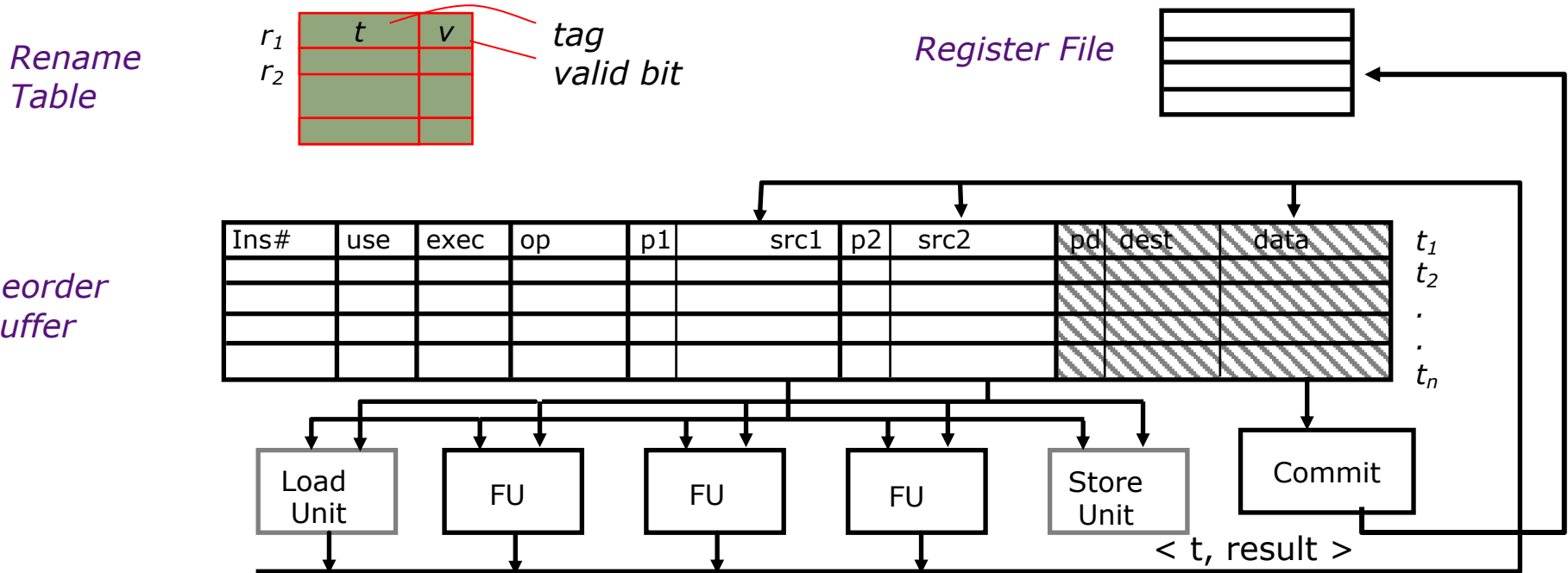


Convert to lazy by holding data in ROB.

But how do we find values before they are committed?

Renaming Table

Micro-architectural speculative cache to speed up tag look up.



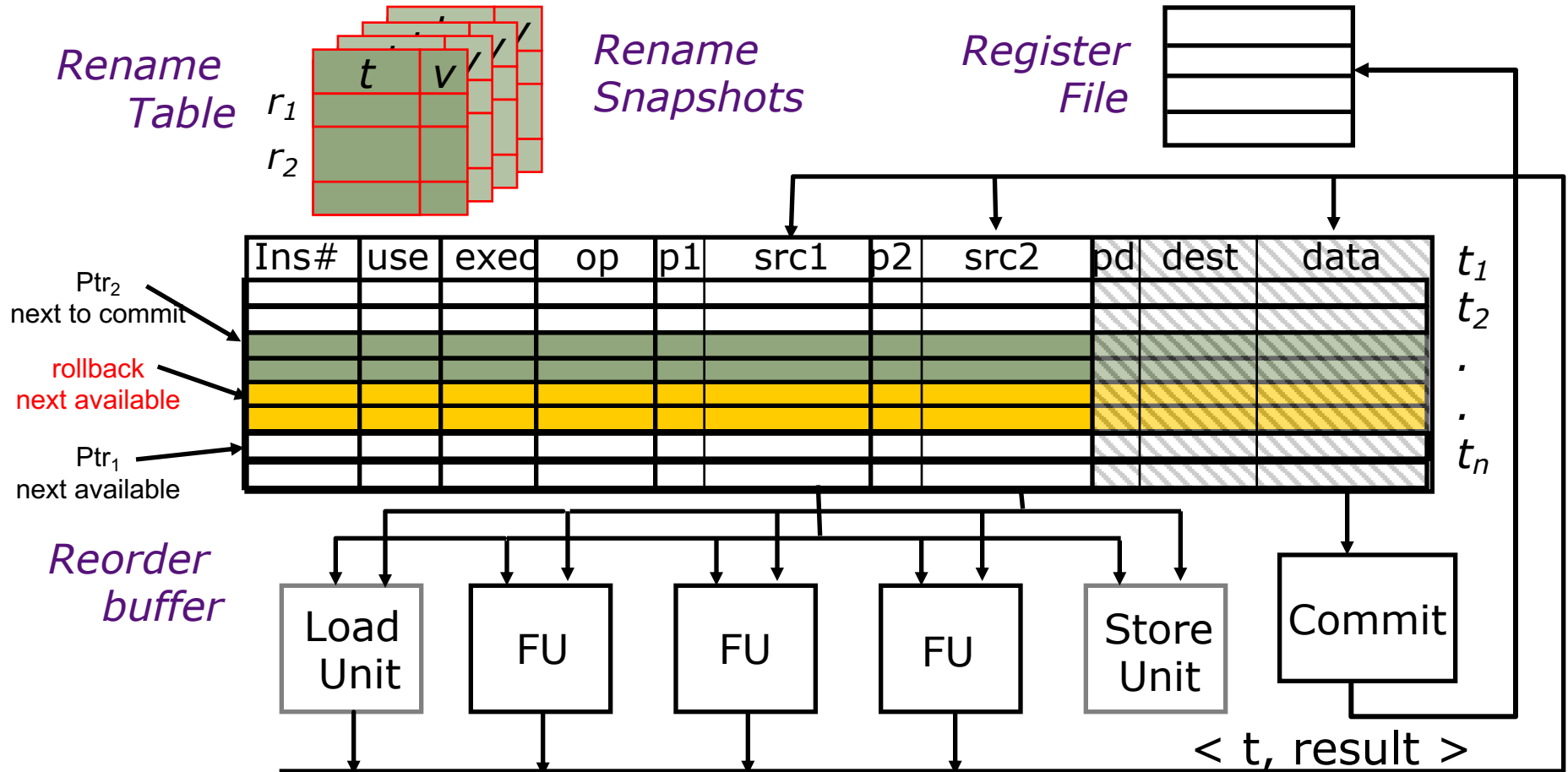
What is the update policy of rename table?

What events cause mis-speculation?

How can we respond to mis-speculation?

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

Recovering ROB/Renaming Table



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

Map Table Recovery - Snapshots

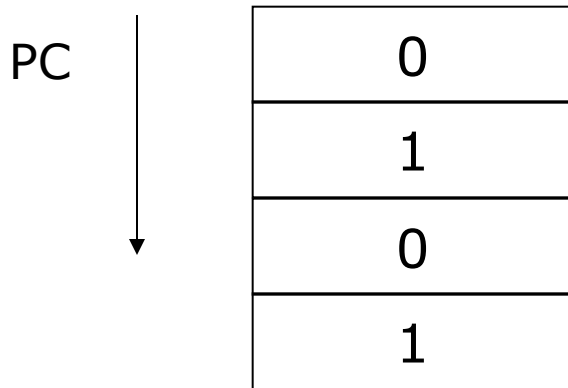
Speculative value management of microarchitectural state

	Reg Map	V	Snap Map	V	Snap Map	V
R0	T20	X	T20	X	T20	X
R1	T73	X	T73	X	T08	
R2	T45	X	T45	X	T45	X
R3	T128		T128		T128	X
	⋮		⋮		⋮	
R30	T54		T54		T54	
R31	T88	X	T88	X	T88	X

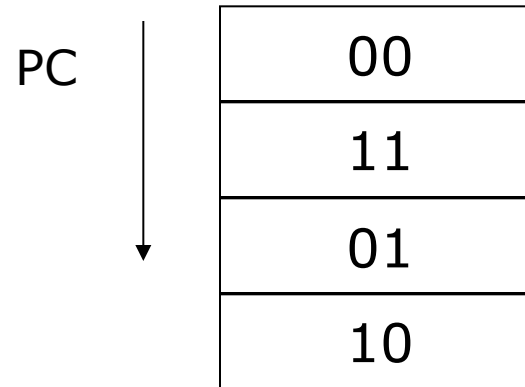
What kind of value management is this?

Branch Predictor Recovery

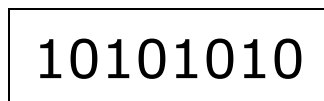
- 1-Bit Counter Recovery



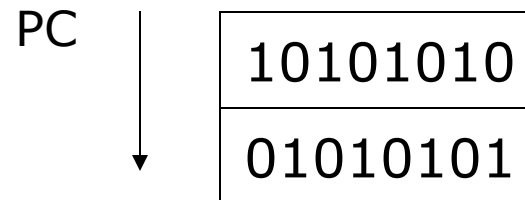
- 2-Bit Counter Recovery



- Global History Recovery

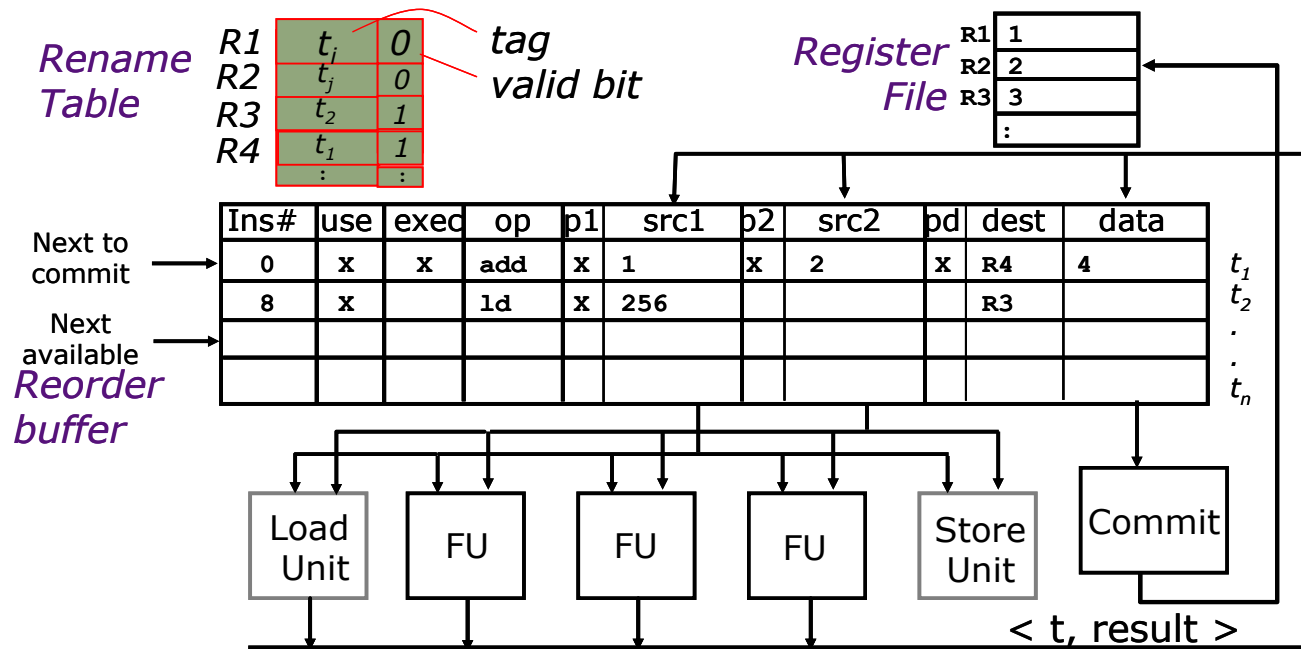


- Local History Recovery



O-o-O Execution with ROB

Data-in-ROB design

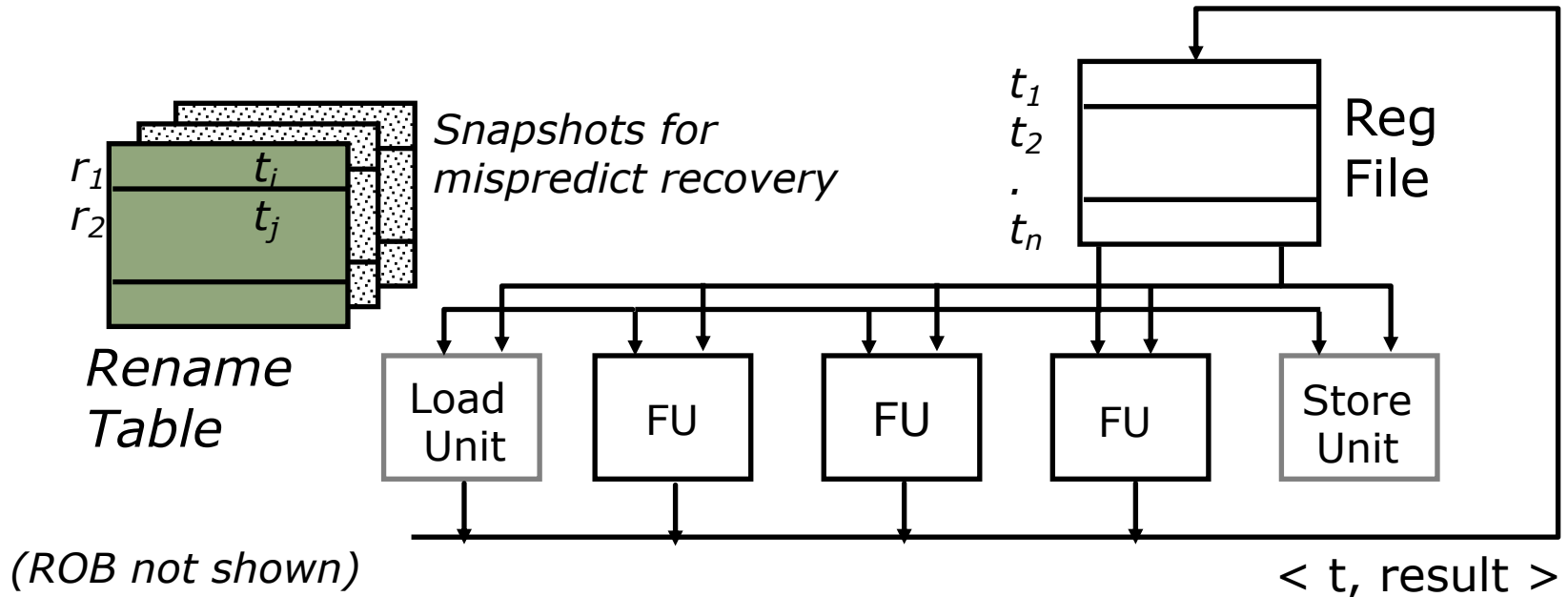


Basic Operation:

- 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

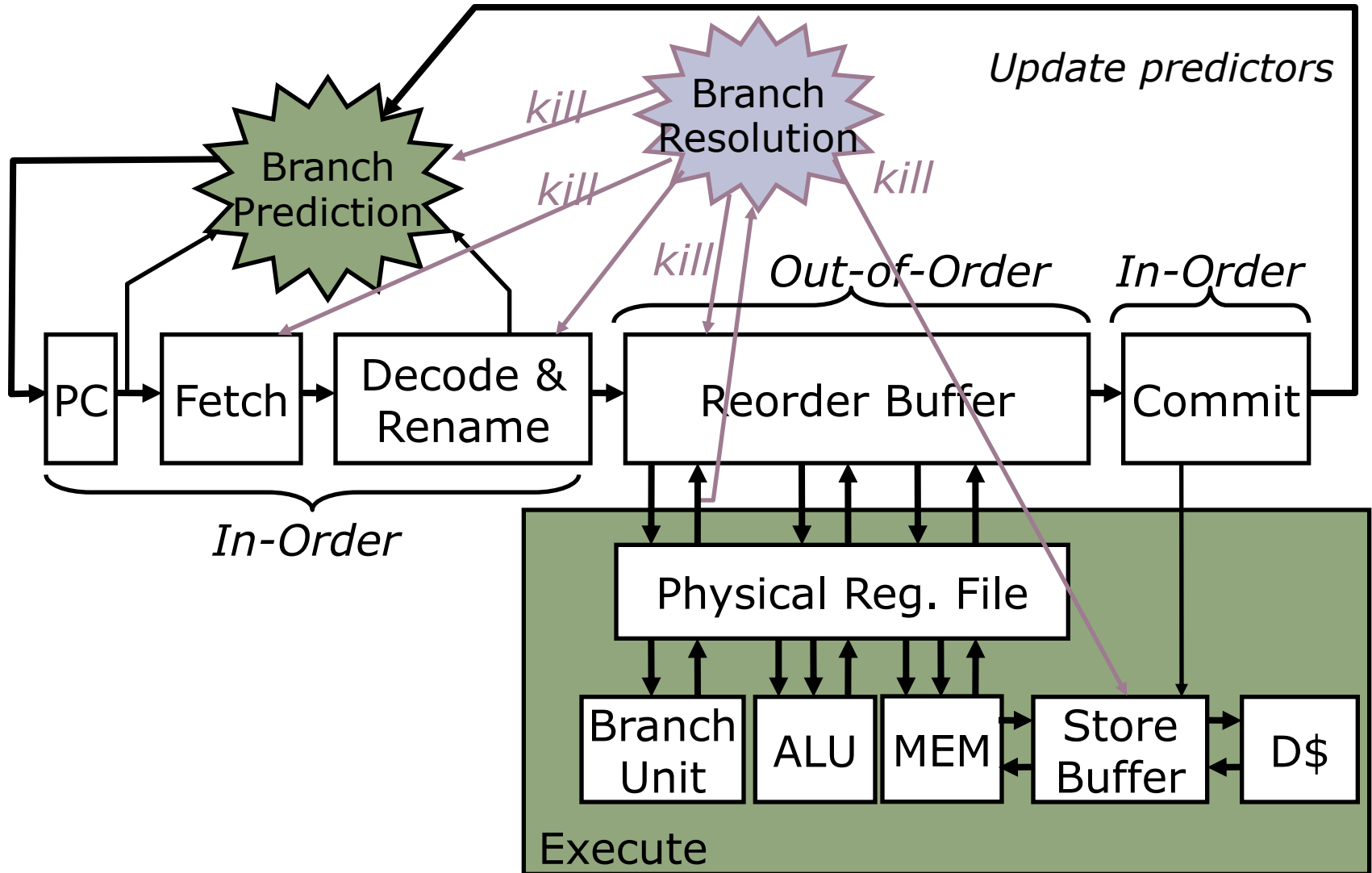
Unified Physical Register File

(MIPS R10K, Alpha 21264, Pentium 4)



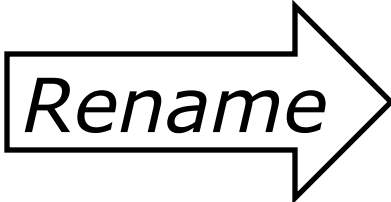
- One regfile for both *committed* and *speculative* values (no data in ROB)
- During decode, instruction result allocated new physical register, source regs translated to physical regs through rename table
- Instruction reads data from regfile at start of execute (not in decode)
- Write-back updates 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



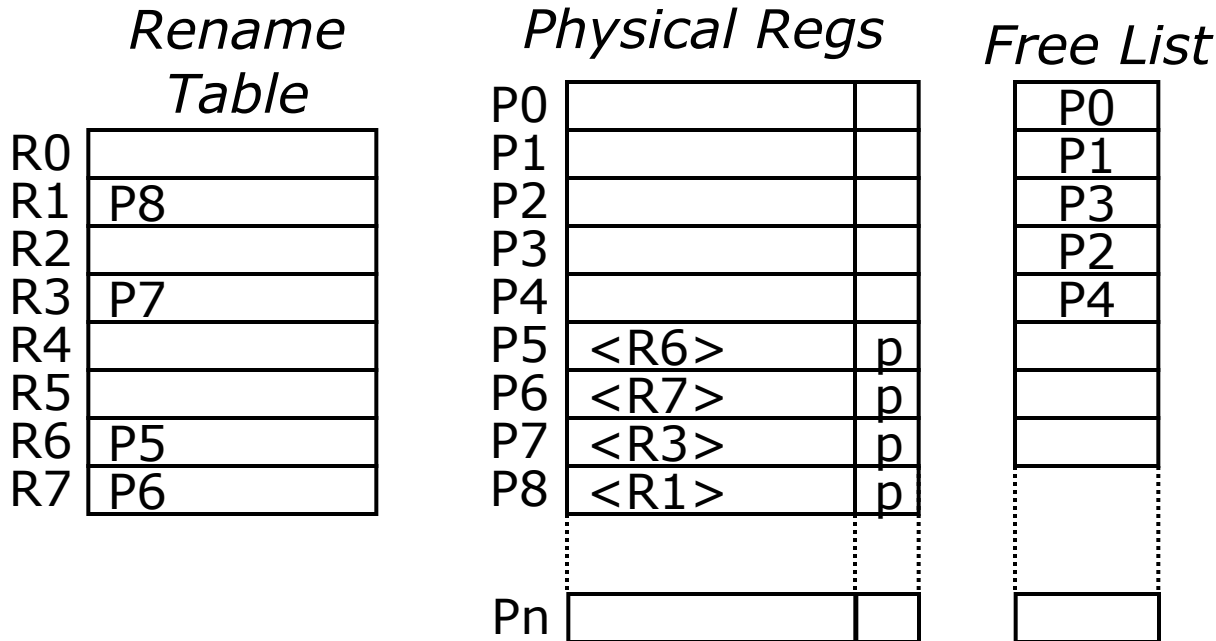
Lifetime of Physical Registers

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

a)	ld r1 , (r3)		ld P1, (Px)
b)	add r3, r1, #4		add P2, P1, #4
c)	sub r1 , r3, r9		sub P3, P2, Py
d)	add r3 , r1, r7		add P4, P3, Pz
e)	ld r6, (r1)		ld P5, (P3)
f)	add r8, r6, r3		add P6, P5, P4
g)	st r8, (r1)		st P6, (P3)
h)	ld r3 , (r11)		ld P7, (Pw)

When can we reuse a physical register?

Physical Register Management



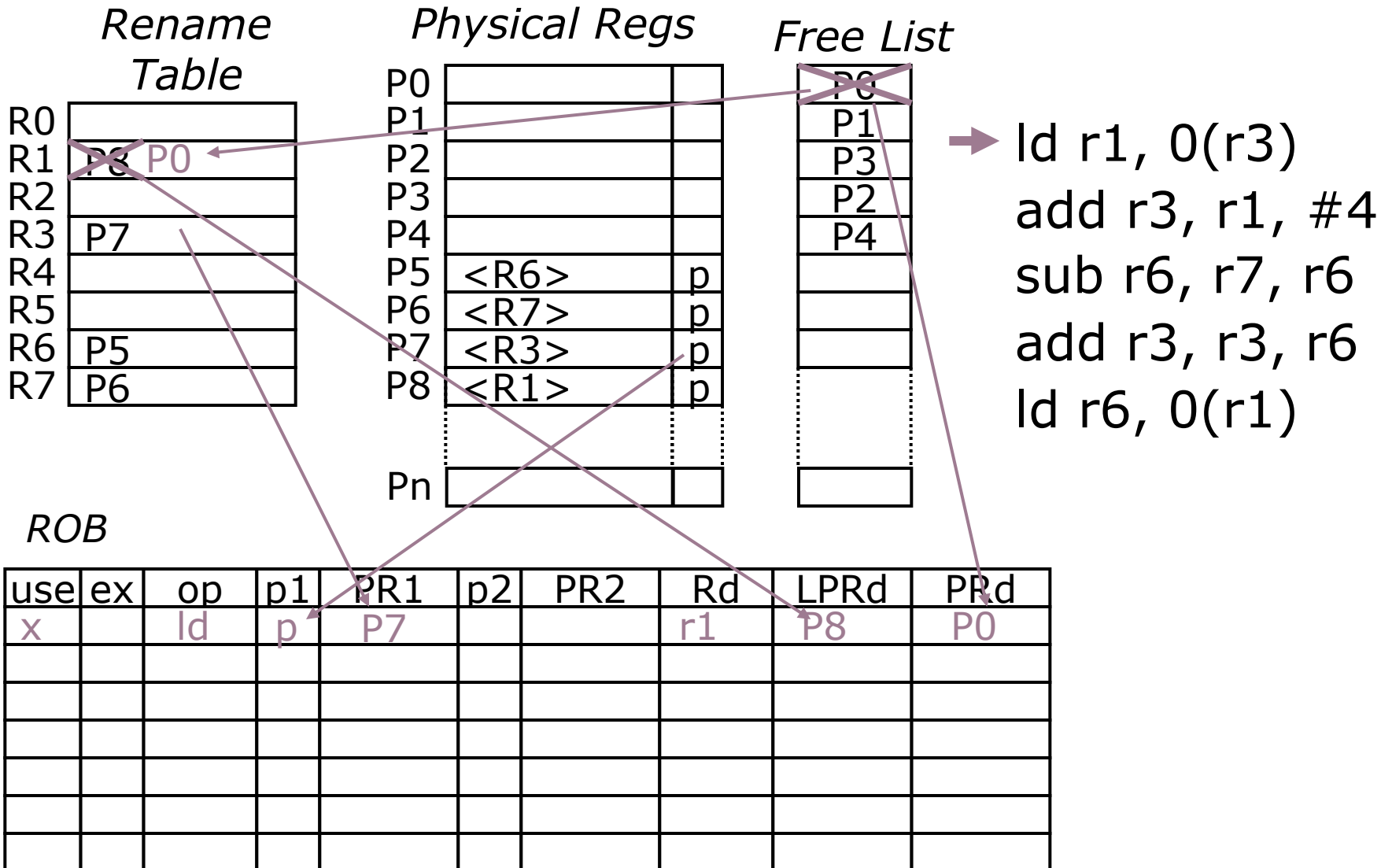
```
ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)
```

ROB

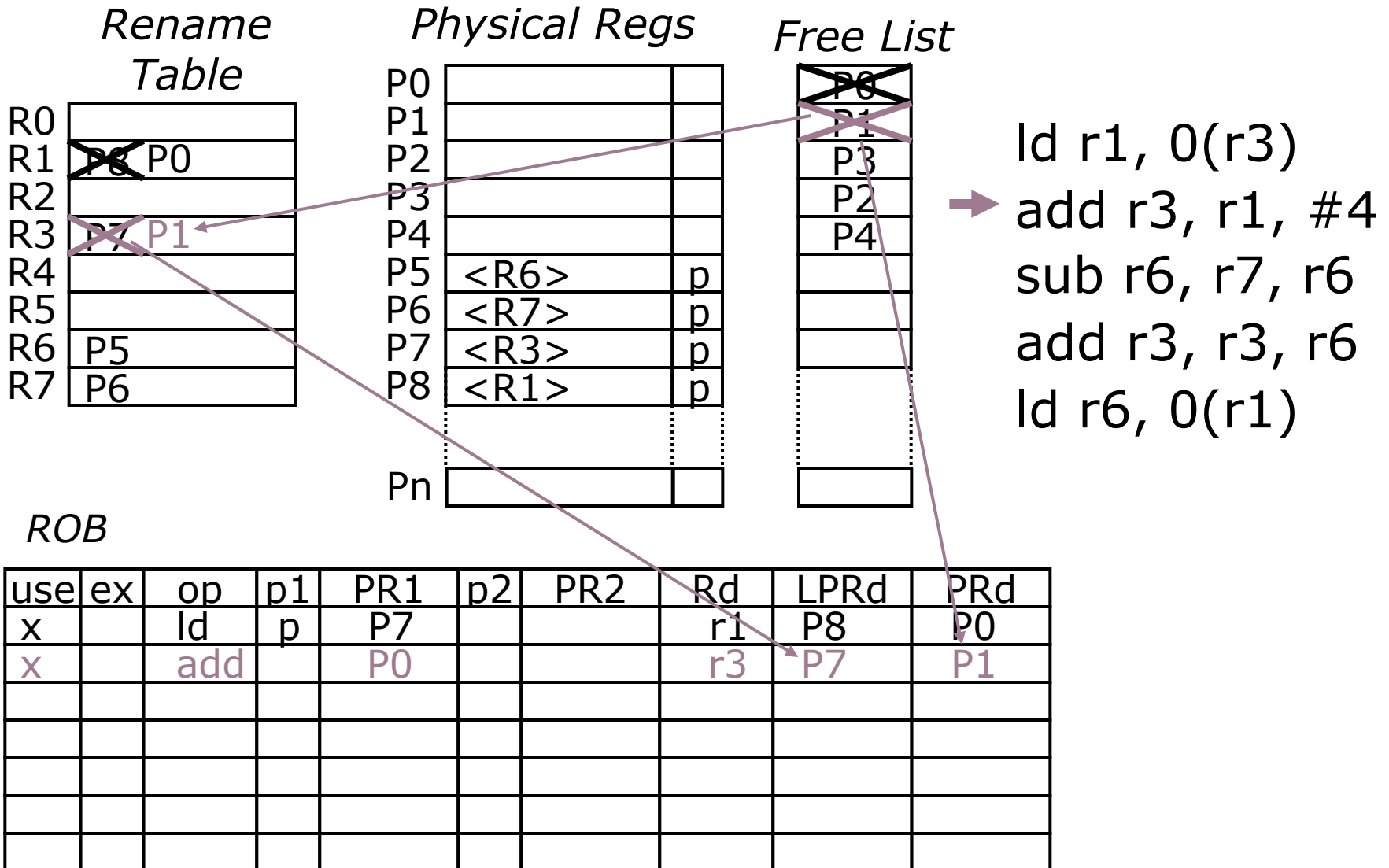
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd

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

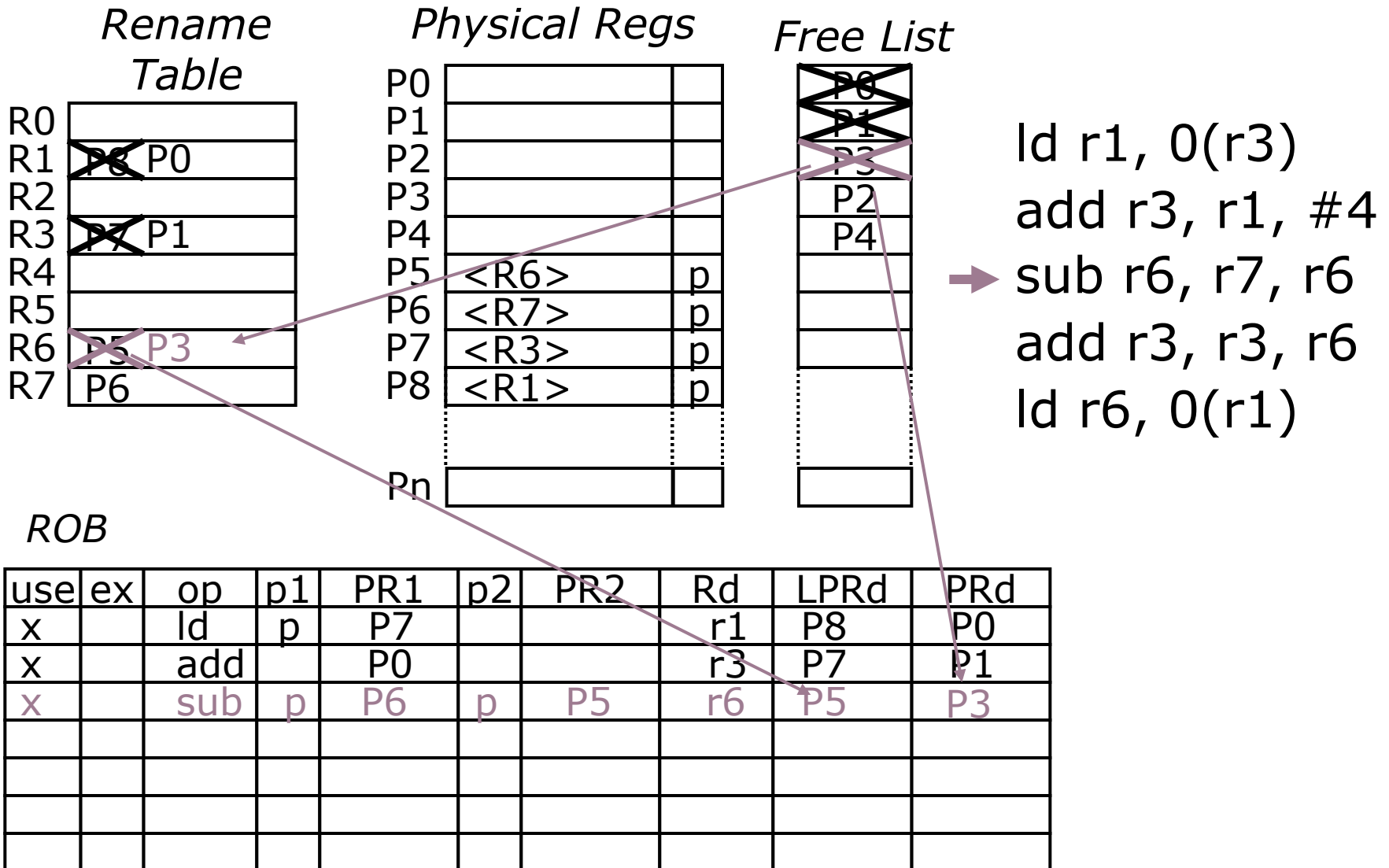
Physical Register Management



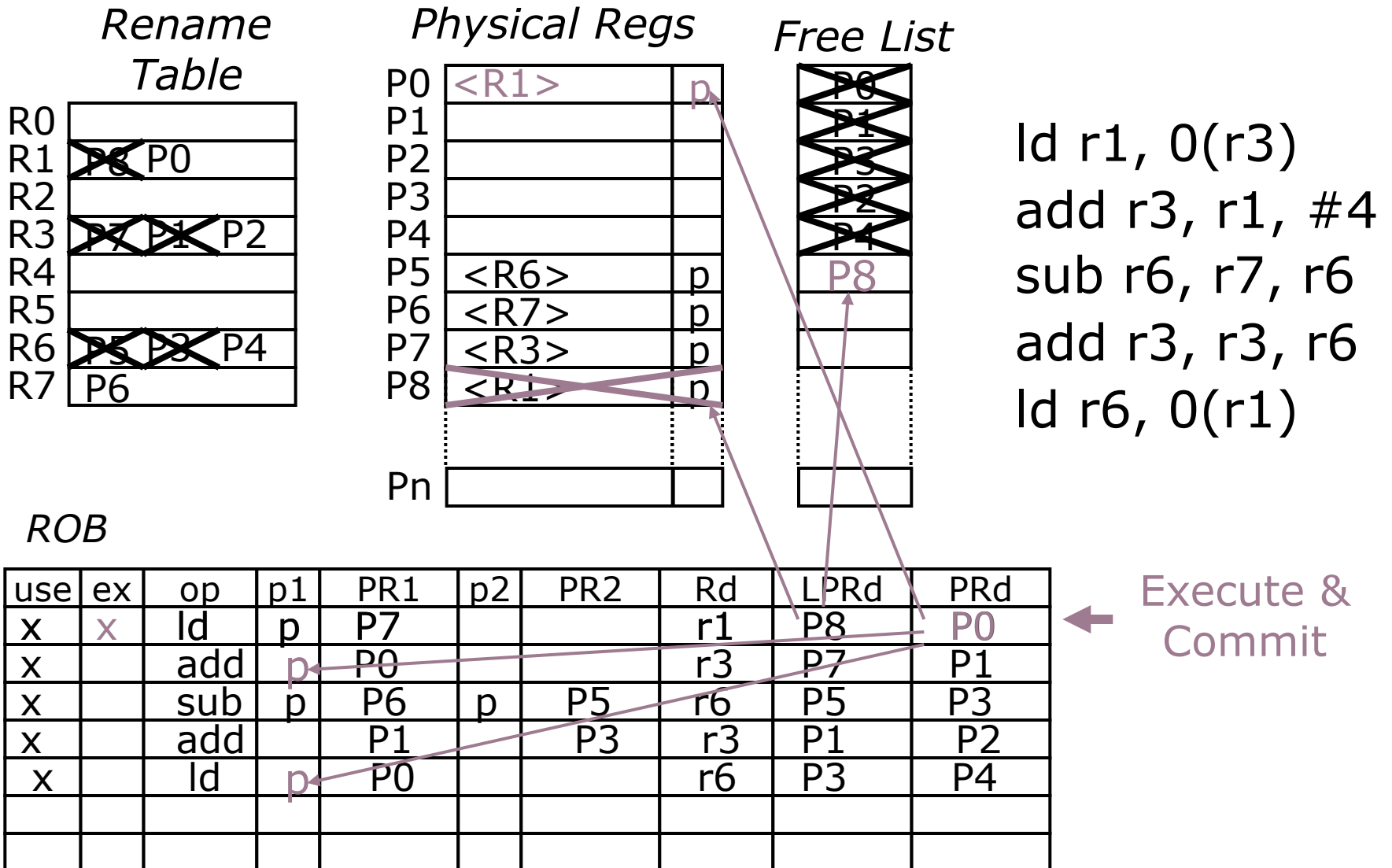
Physical Register Management



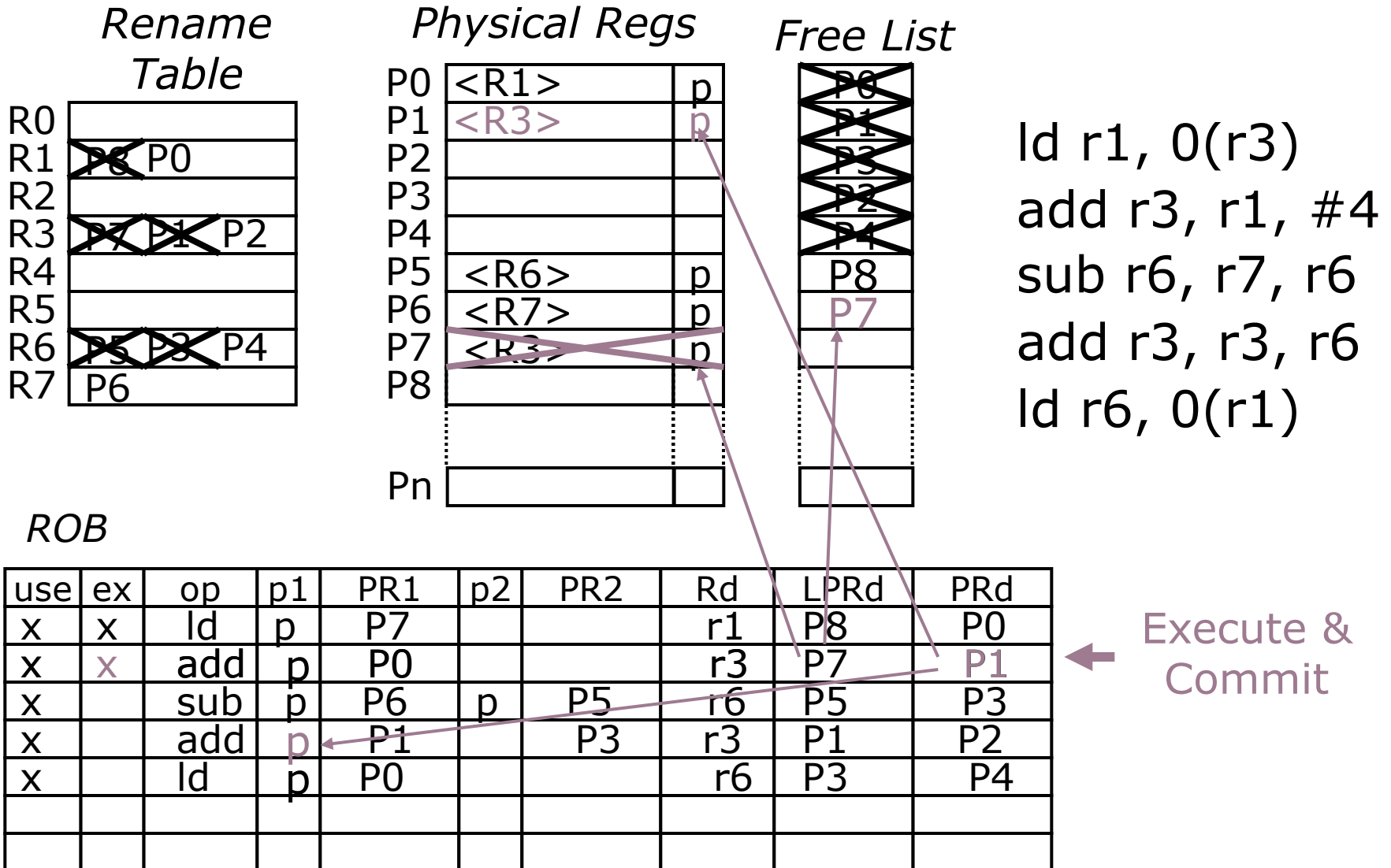
Physical Register Management



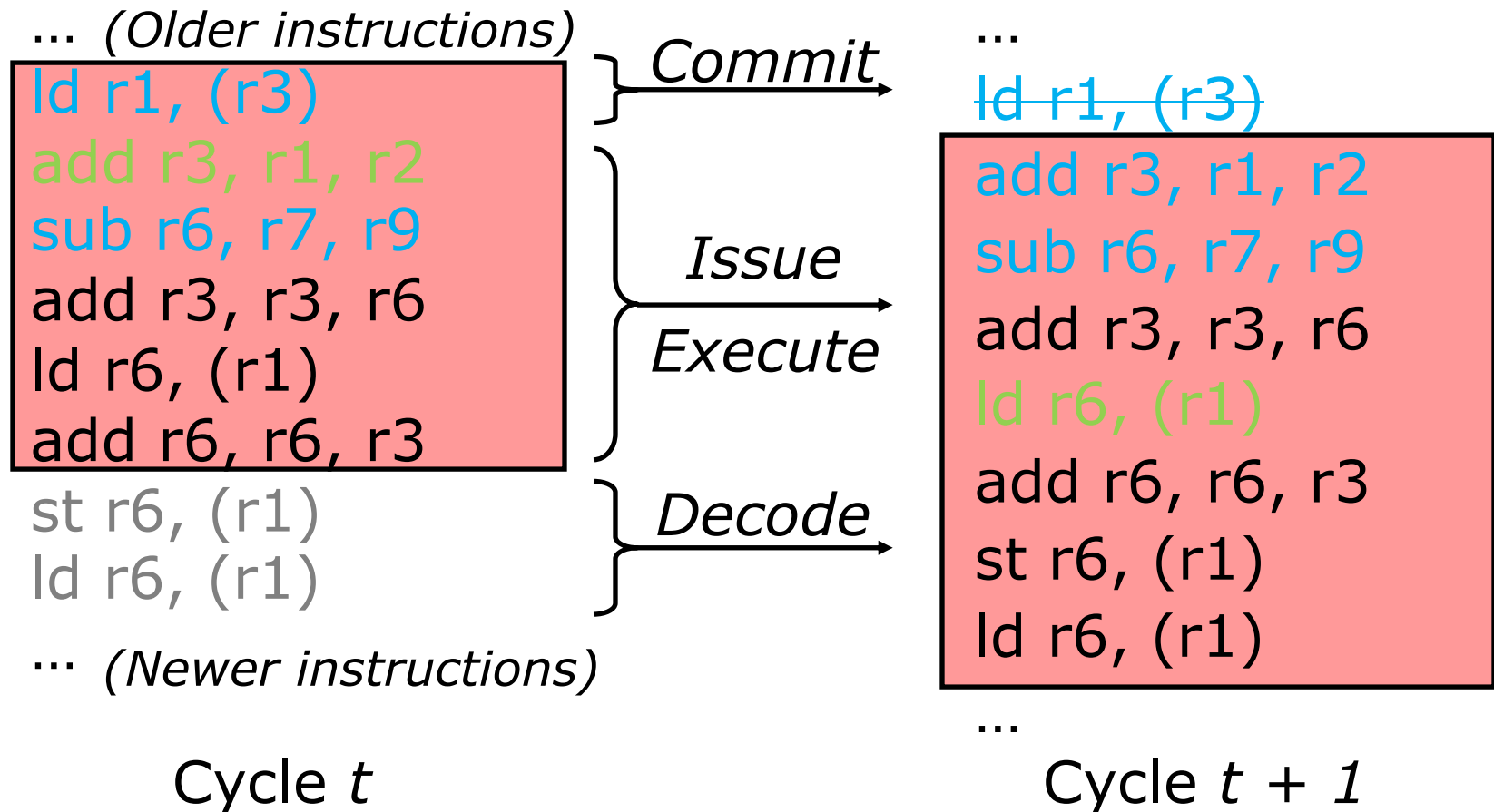
Physical Register Management



Physical Register Management



Reorder Buffer Holds Active Instruction Window



Key: predecode, decoded, issued, executed, committed

Issue Timing

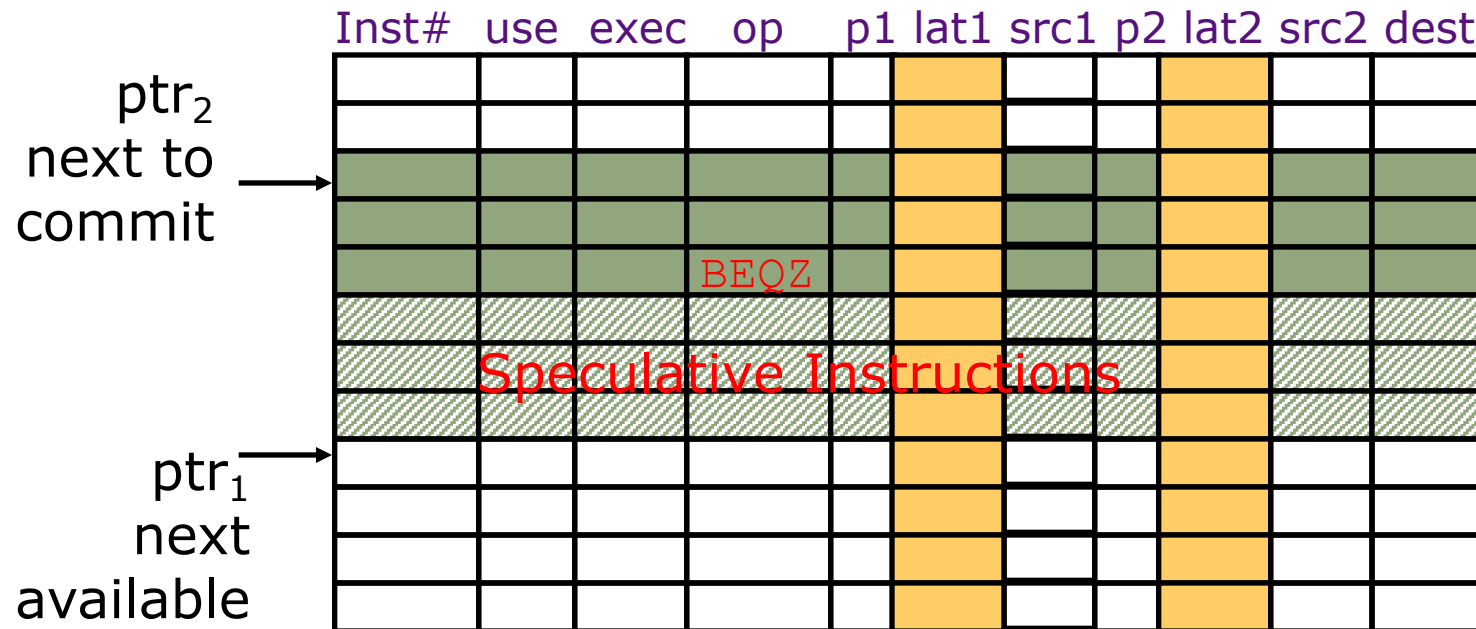
i1	Add R1,R1,#1	Issue ₁	Execute ₁		
i2	Sub R1,R1,#1			Issue ₂	Execute ₂

How can we issue earlier?

i1	LD R1, (R3)	Issue ₁	Execute ₁		
i2	Sub R1,R1,#1		Issue ₂	Execute ₂	

What might make this schedule fail?

Issue Queue with latency prediction

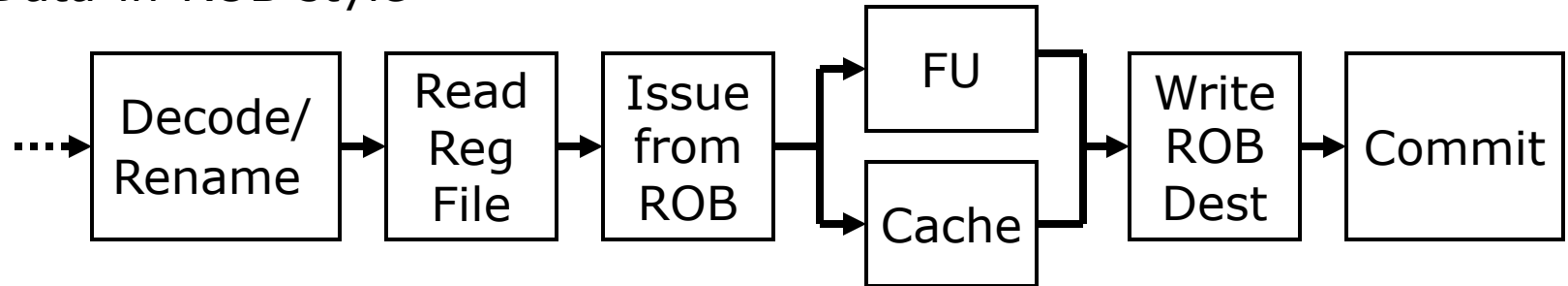


Issue Queue (Reorder buffer)

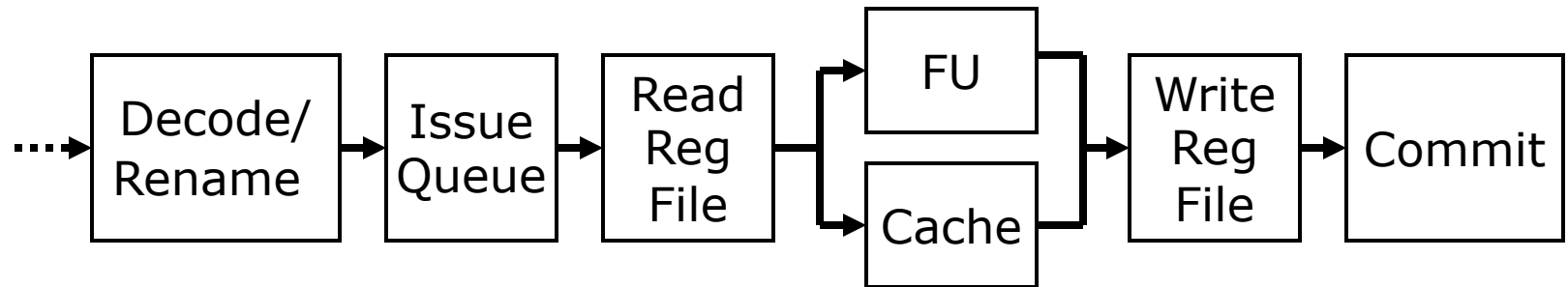
- Fixed latency: latency included in queue entry ('bypassed')
- Predicted latency: latency included in queue entry (speculated)
- Variable latency: wait for completion signal (stall)

Data-in-ROB vs. Unified RegFile

Data-in-ROB style



Unified-register-file style



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

Thank you!