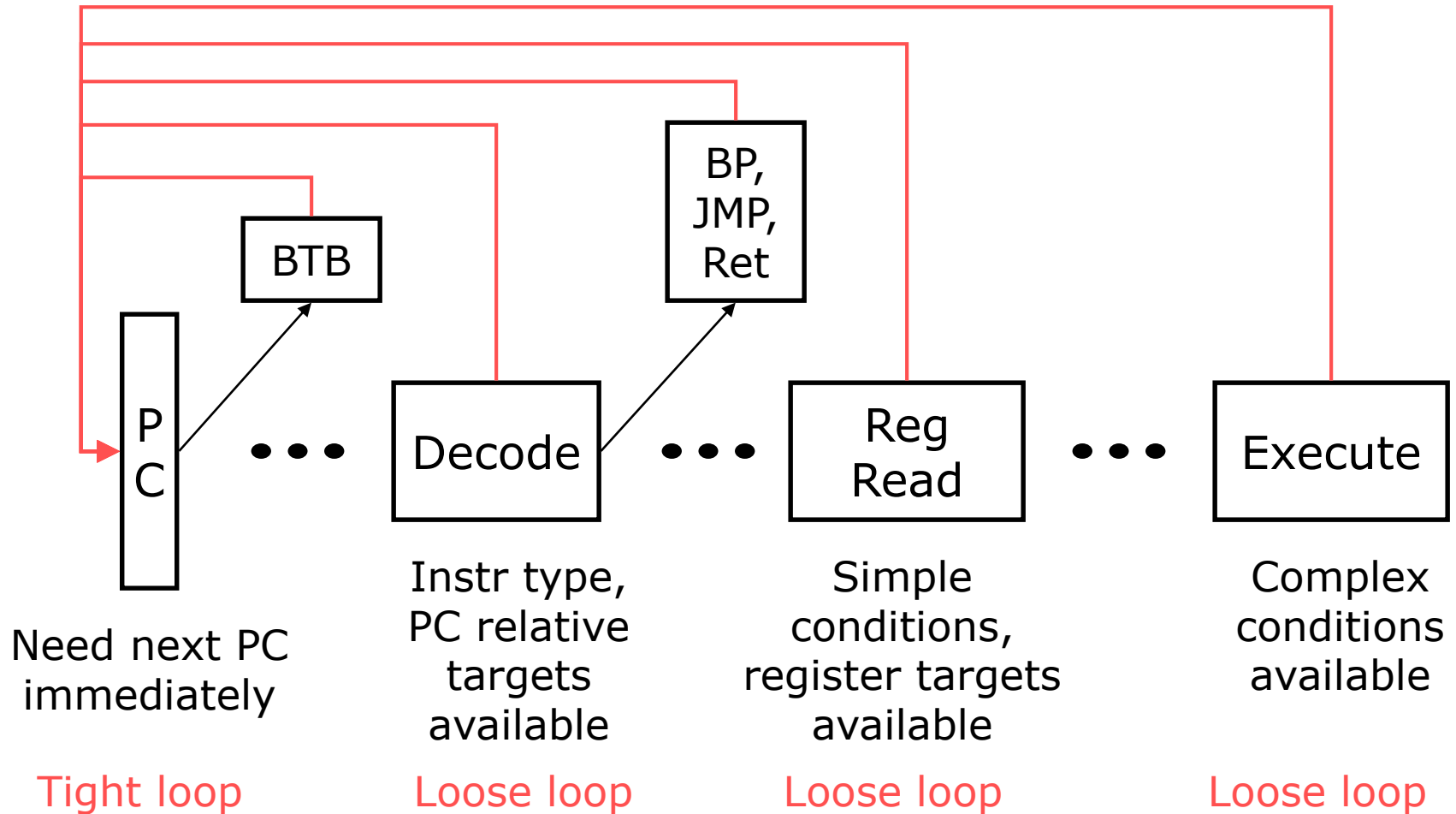


Speculative Execution

Daniel Sanchez

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

Overview of branch prediction



Must speculation check always be correct? No...

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
- Provide means to reconstruct old values for recovery
 - often this is a log of old values

Lazy Update:

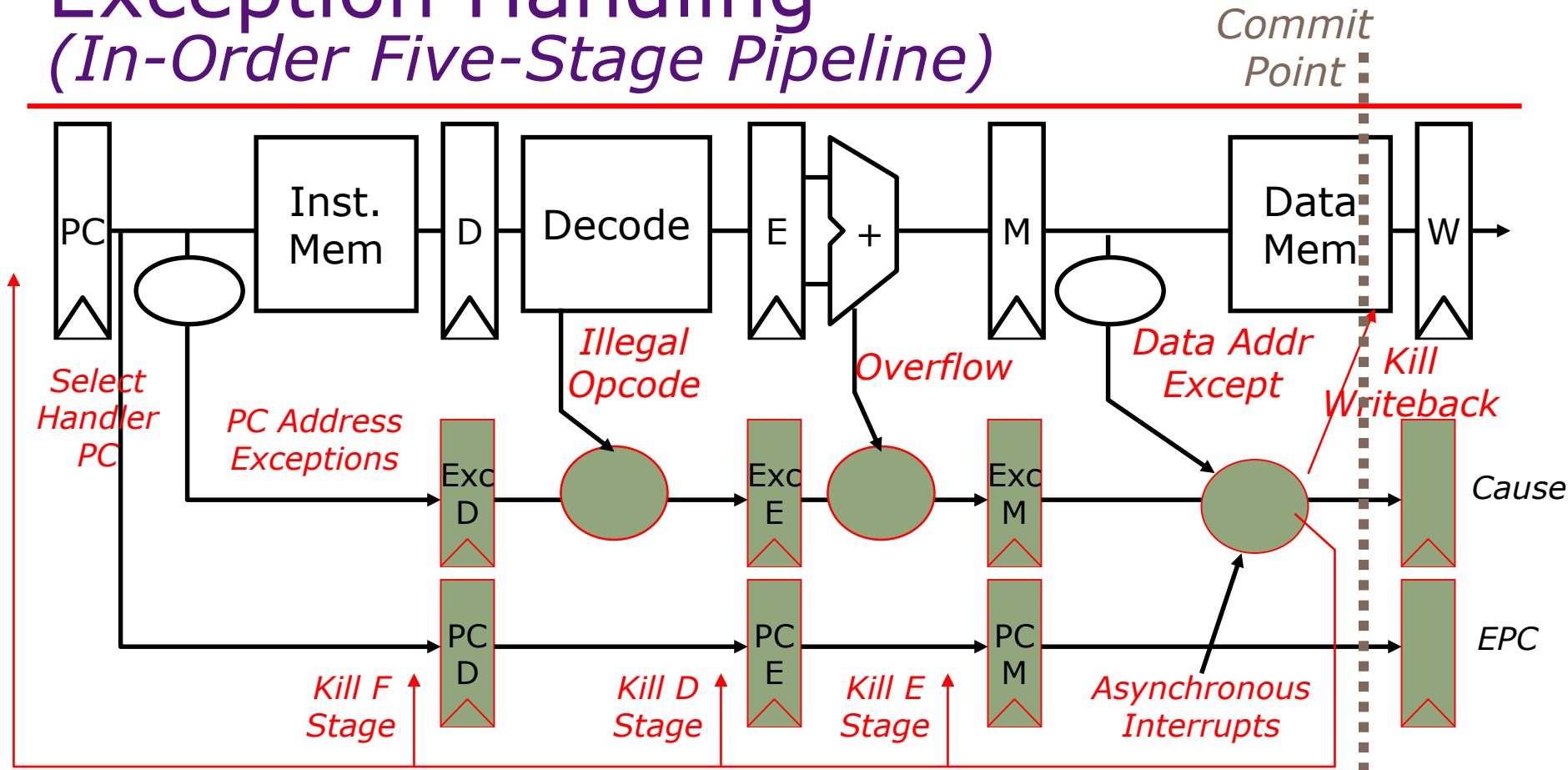
- Buffer new value, leaving old value in place
- Replace old value only at 'commit' time

Why leave an old value in place?

- When there will be limited use of new value
- To make it easy to use old value after new value is generated
- To simplify recovery

Exception Handling

(In-Order Five-Stage Pipeline)



Strategy for Registers?
 Where are 'new' values?
 Strategy for PC?
 Where is 'log'?

Lazy – update at commit
 In execution pipeline
 Greedy – update immediately
 In pipeline of PC latches

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

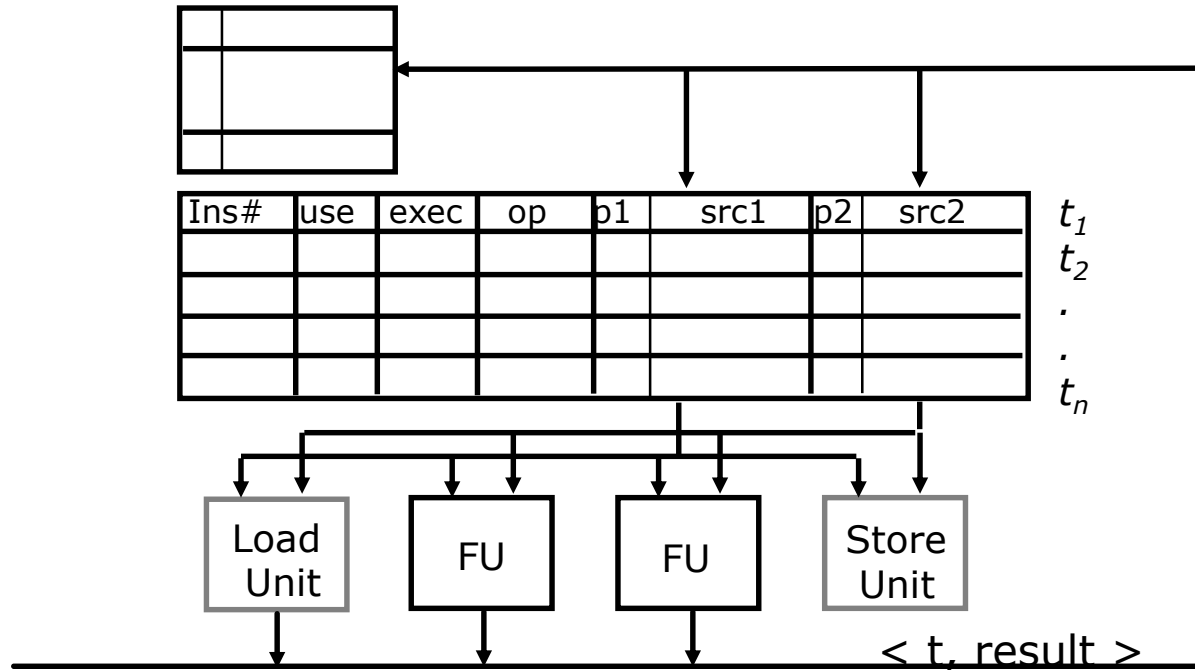
Out-of-order execution?

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

Data-Driven Execution

*Renaming
table &
reg file*

*Reorder
buffer*



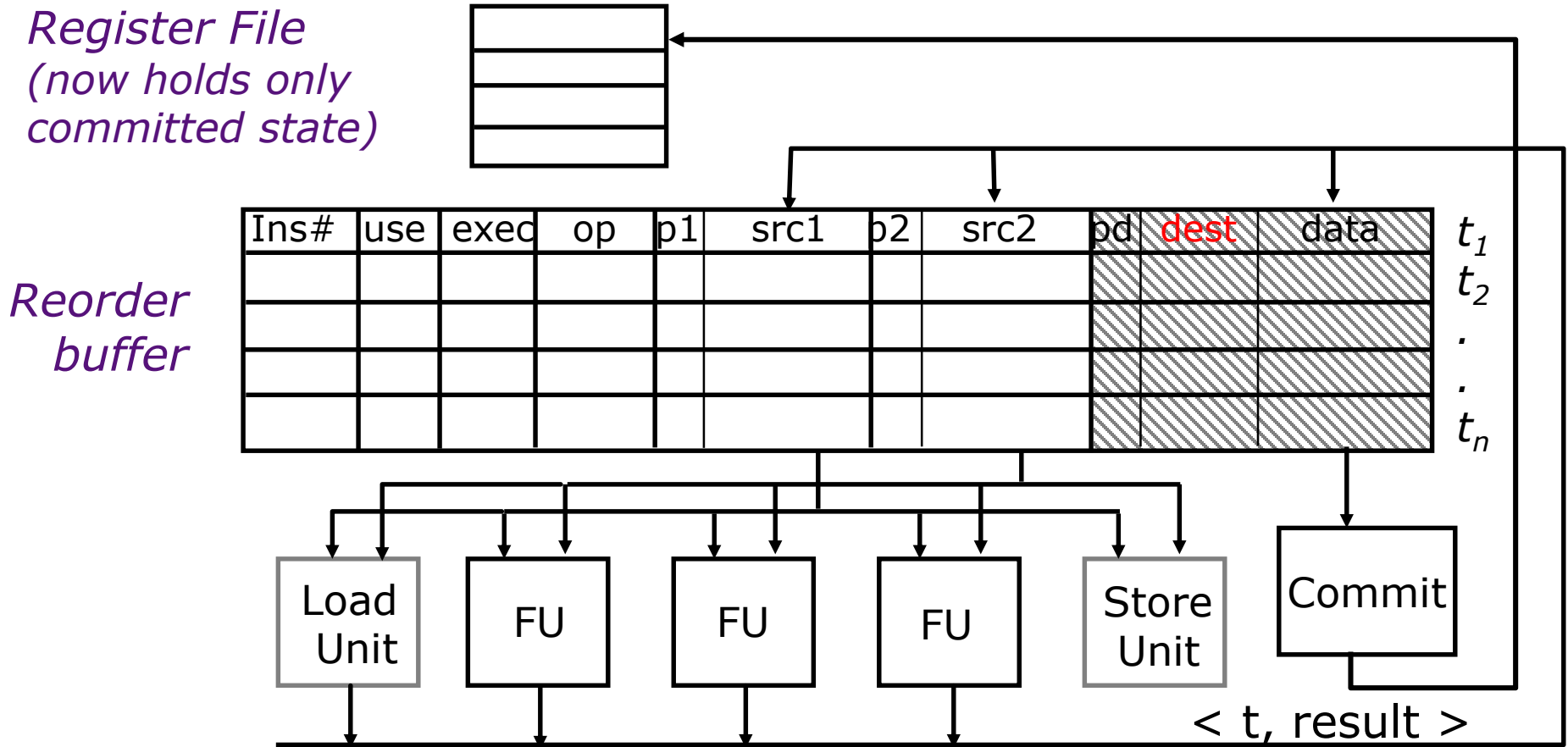
Basic Operation:

- 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?

Greedy – update at execute

Rollback and Renaming



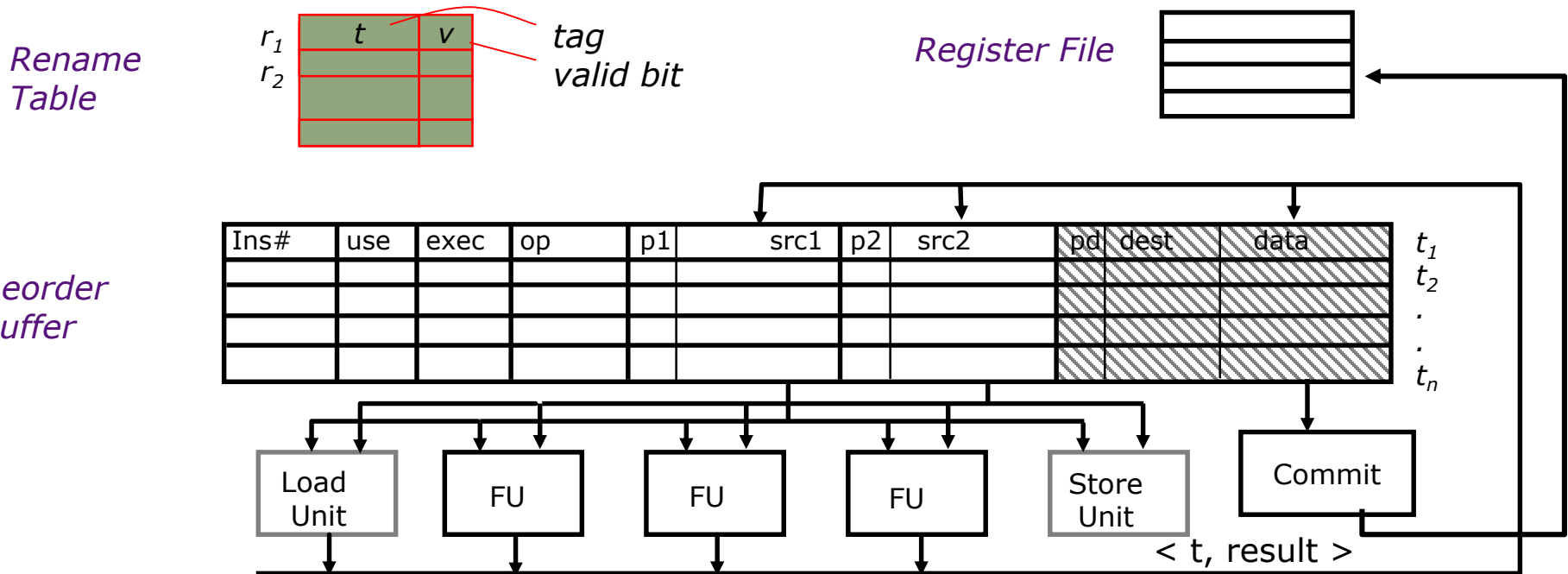
Convert to lazy by holding data in ROB.

But how do we find values before they are committed?

Search the "dest" field in the reorder buffer

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?

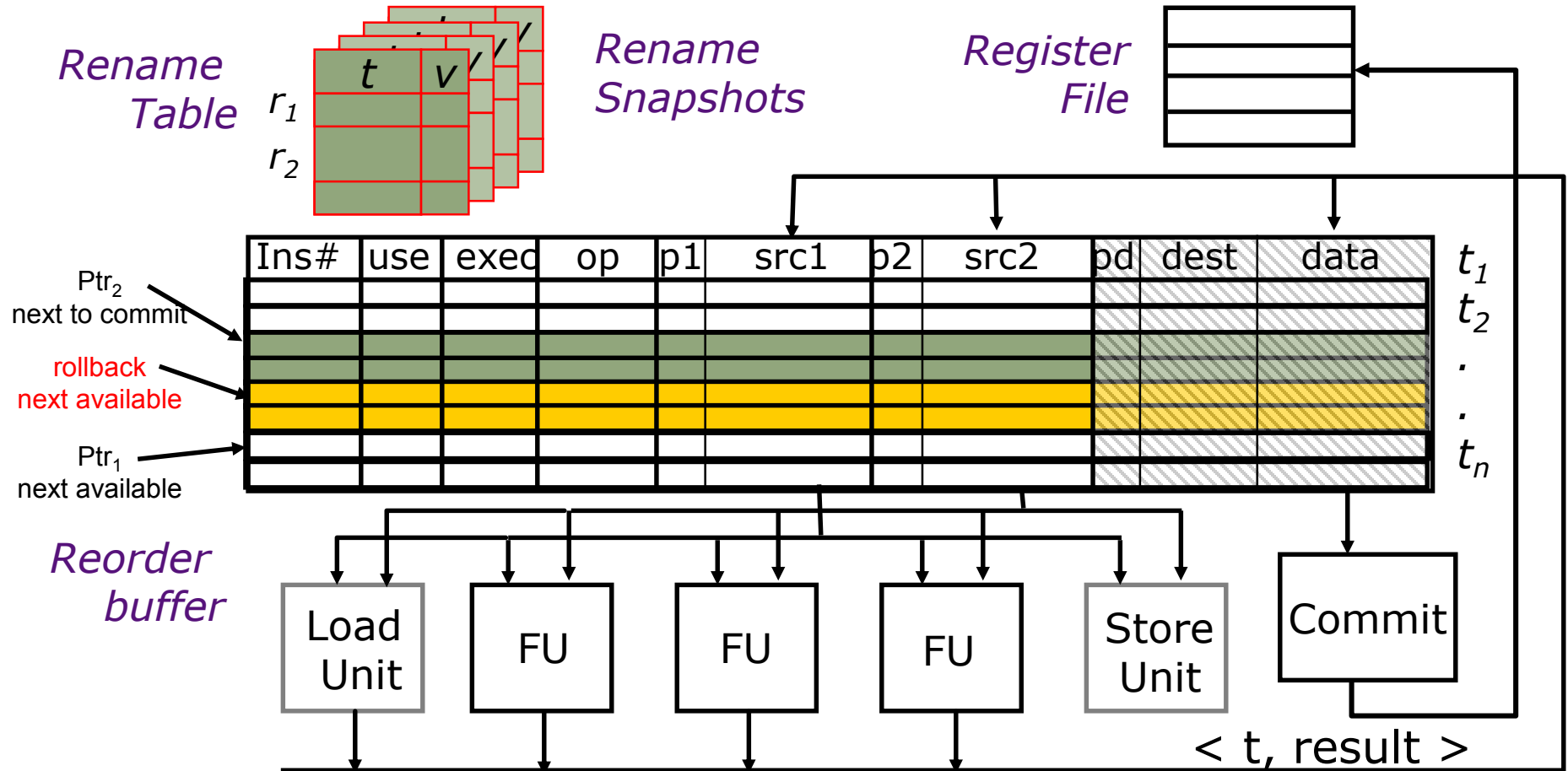
Greedy

Exceptions & branch mispredicts

Clear valid bits

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

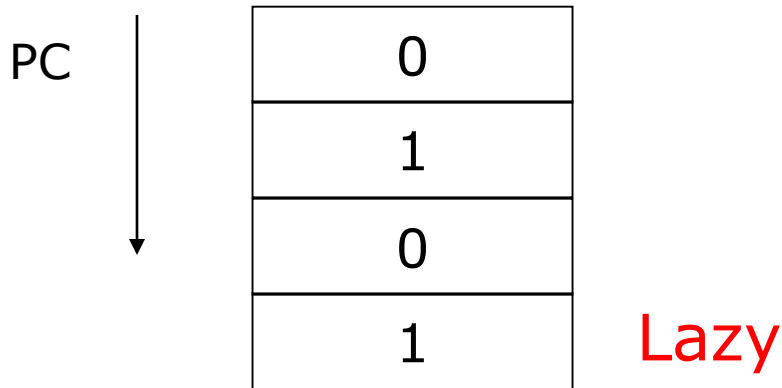
| | Reg Map | V | Snap Map | V | Snap Map | V |
|-----|---------|---|----------|---|----------|---|
| R0 | T20 | X | T20 | X | T20 | X |
| R1 | T73 | X | T73 | X | T08 | |
| R2 | T45 | X | T45 | X | T45 | X |
| R3 | T128 | | T128 | | T128 | X |
| | ⋮ | | ⋮ | | ⋮ | |
| R30 | T54 | | T54 | | T54 | |
|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

What kind of value management is this?

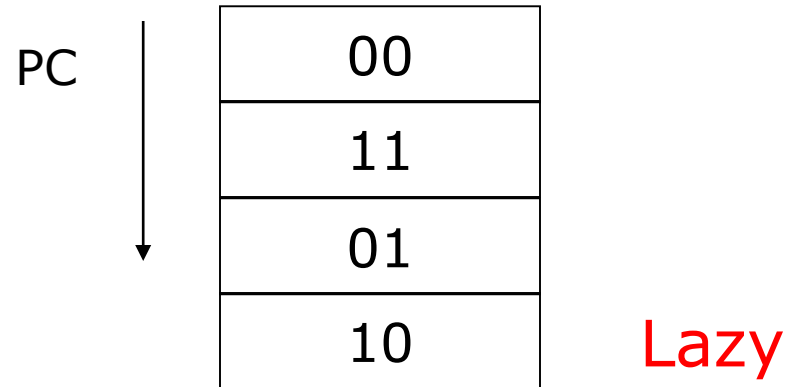
Greedy!!

Branch Predictor Recovery

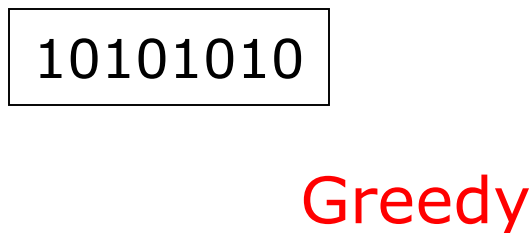
- 1-Bit Counter Recovery



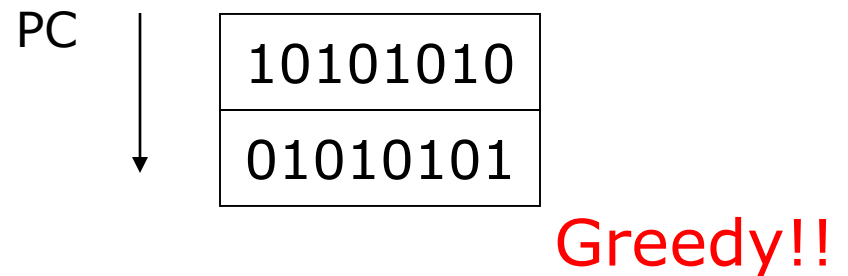
- 2-Bit Counter Recovery



- Global History Recovery

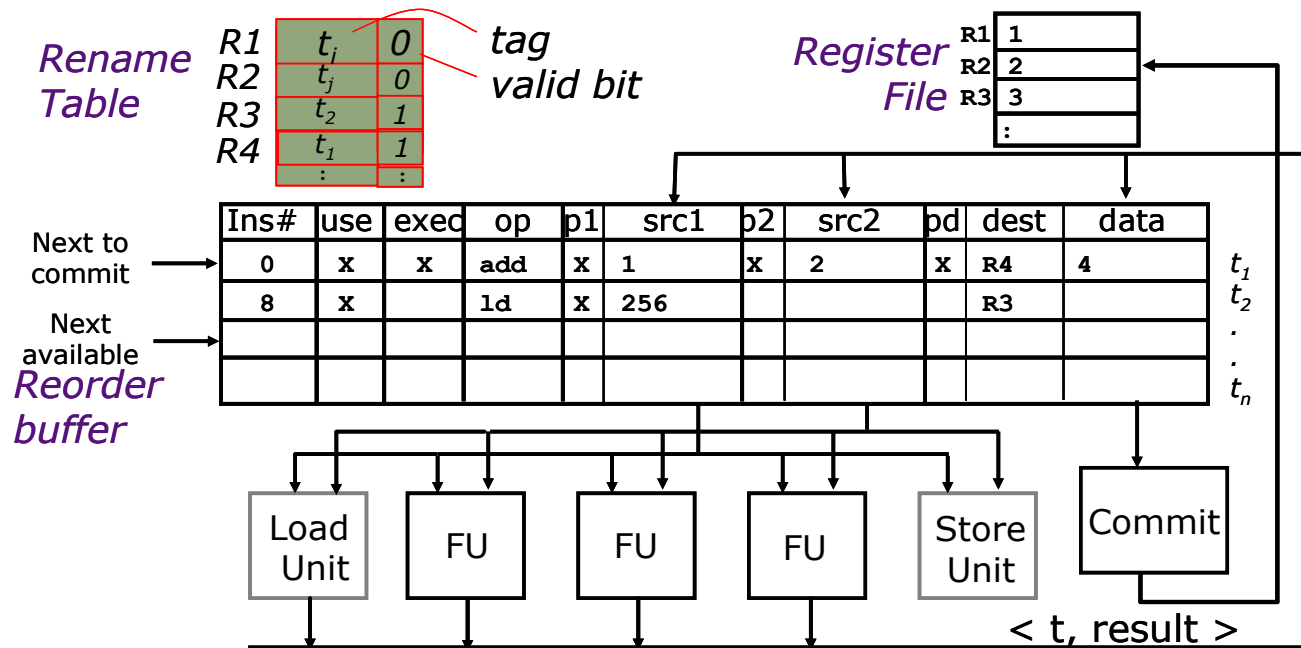


- Local History Recovery



O-o-O Execution with ROB

Data-in-ROB design

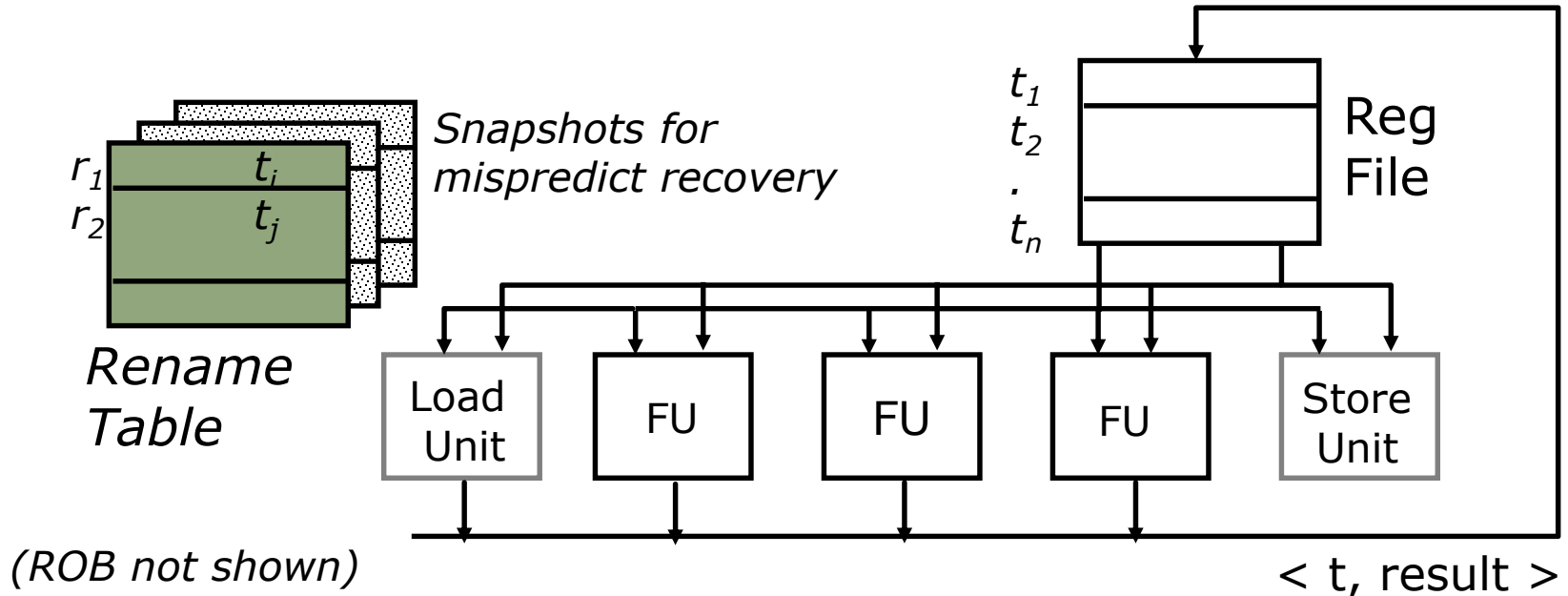


Basic Operation:

- 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

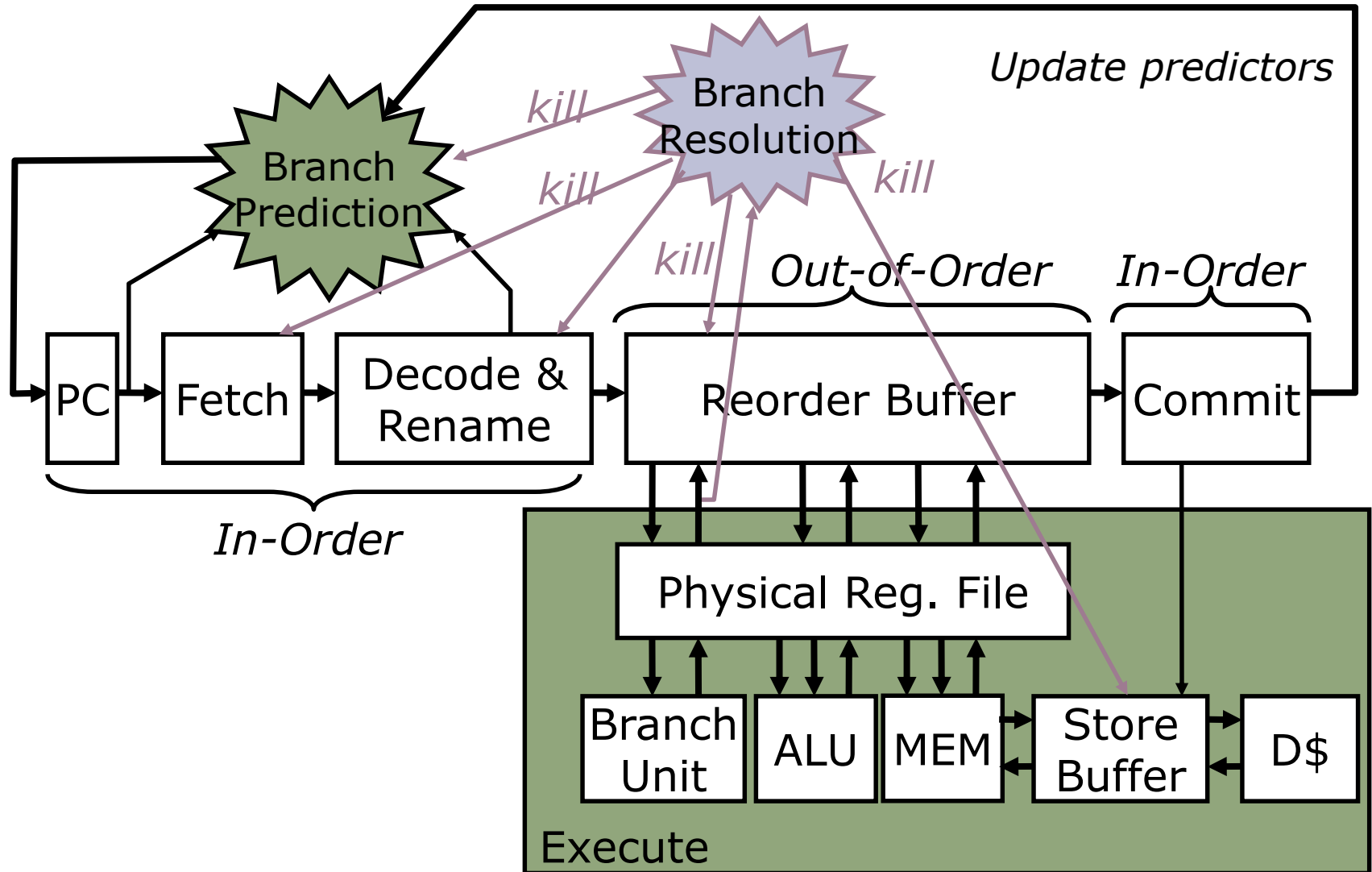
Unified Physical Register File

(MIPS R10K, Alpha 21264, Pentium 4)



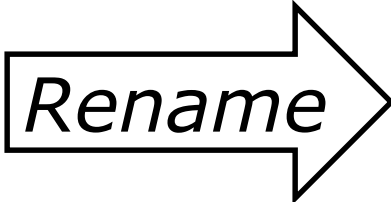
- One regfile for both *committed* and *speculative* values (no data in ROB)
- During decode, instruction result allocated new physical register, source regs translated to physical regs through rename table
- Instruction reads data from regfile at start of execute (not in decode)
- Write-back updates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

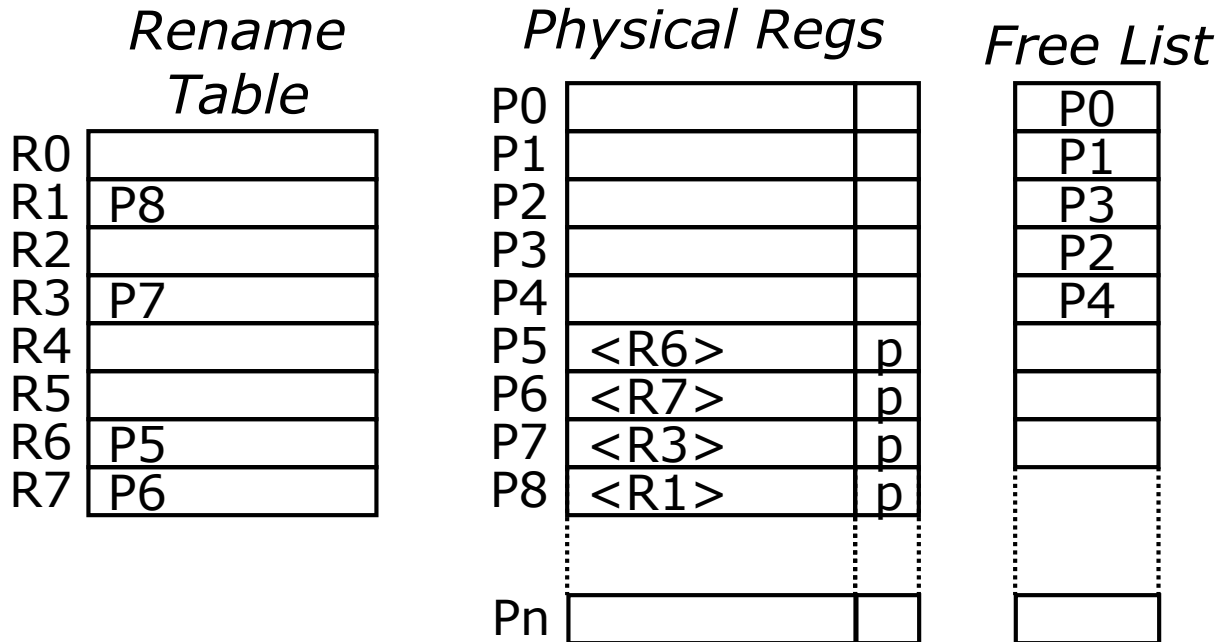
- Physical regfile holds committed and speculative values
- Physical registers decoupled from ROB entries (*no data in ROB*)

| | | | |
|----|----------------|--|----------------|
| 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?

When next write to same architectural register commits

Physical Register Management



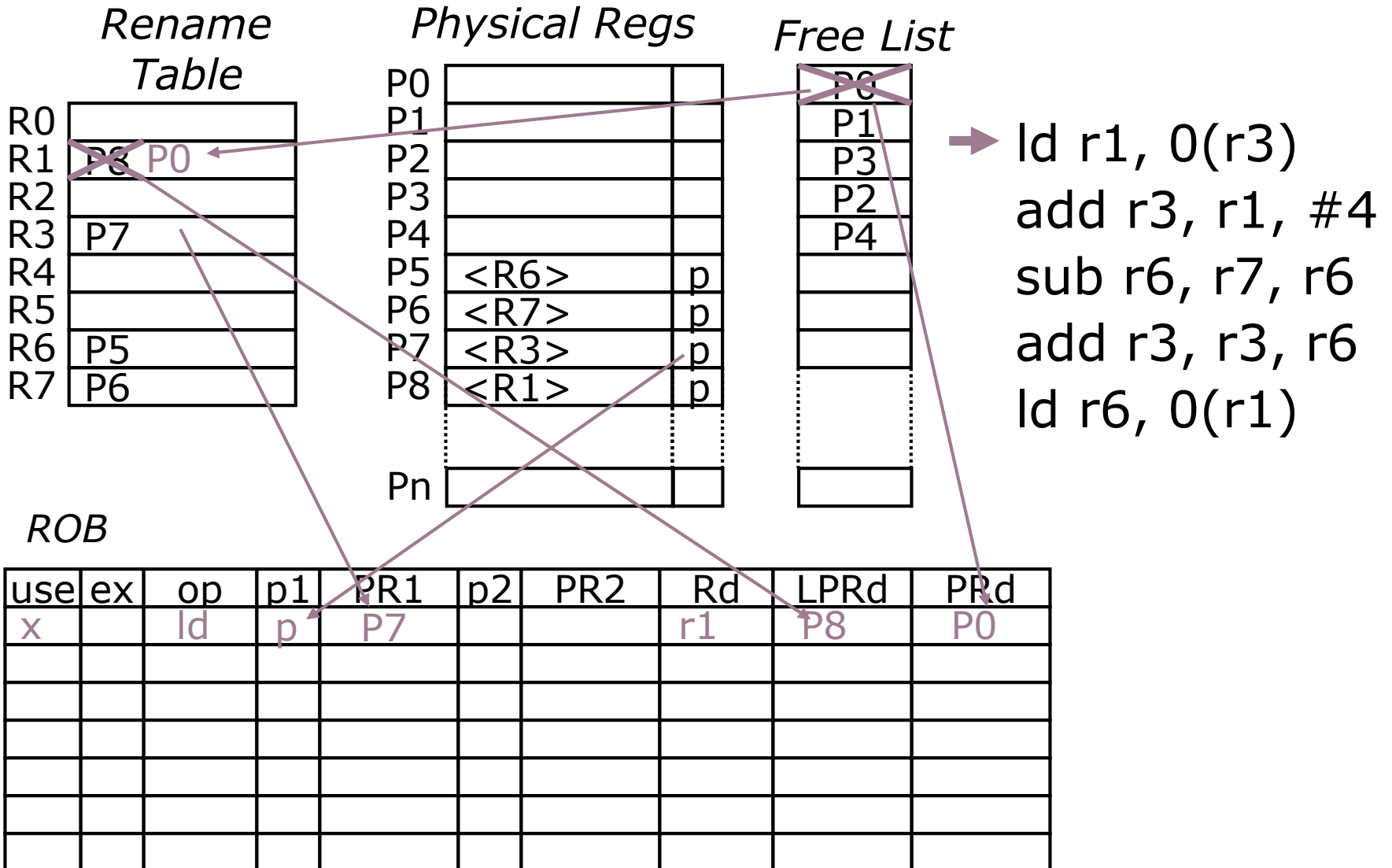
```
ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)
```

ROB

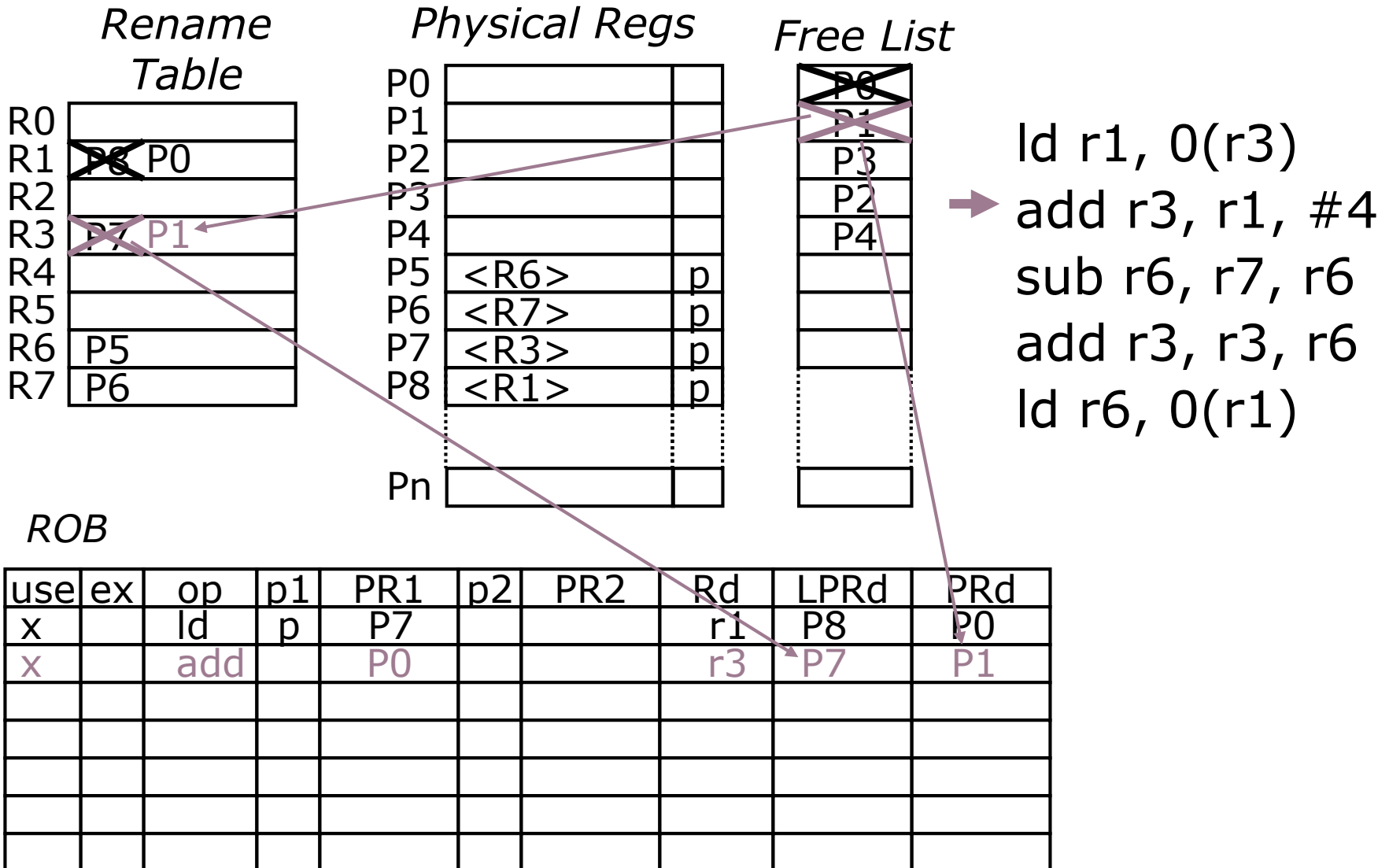
| use | ex | op | p1 | PR1 | p2 | PR2 | Rd | LPRd | PRd |
|-----|----|----|----|-----|----|-----|----|------|-----|
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |

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

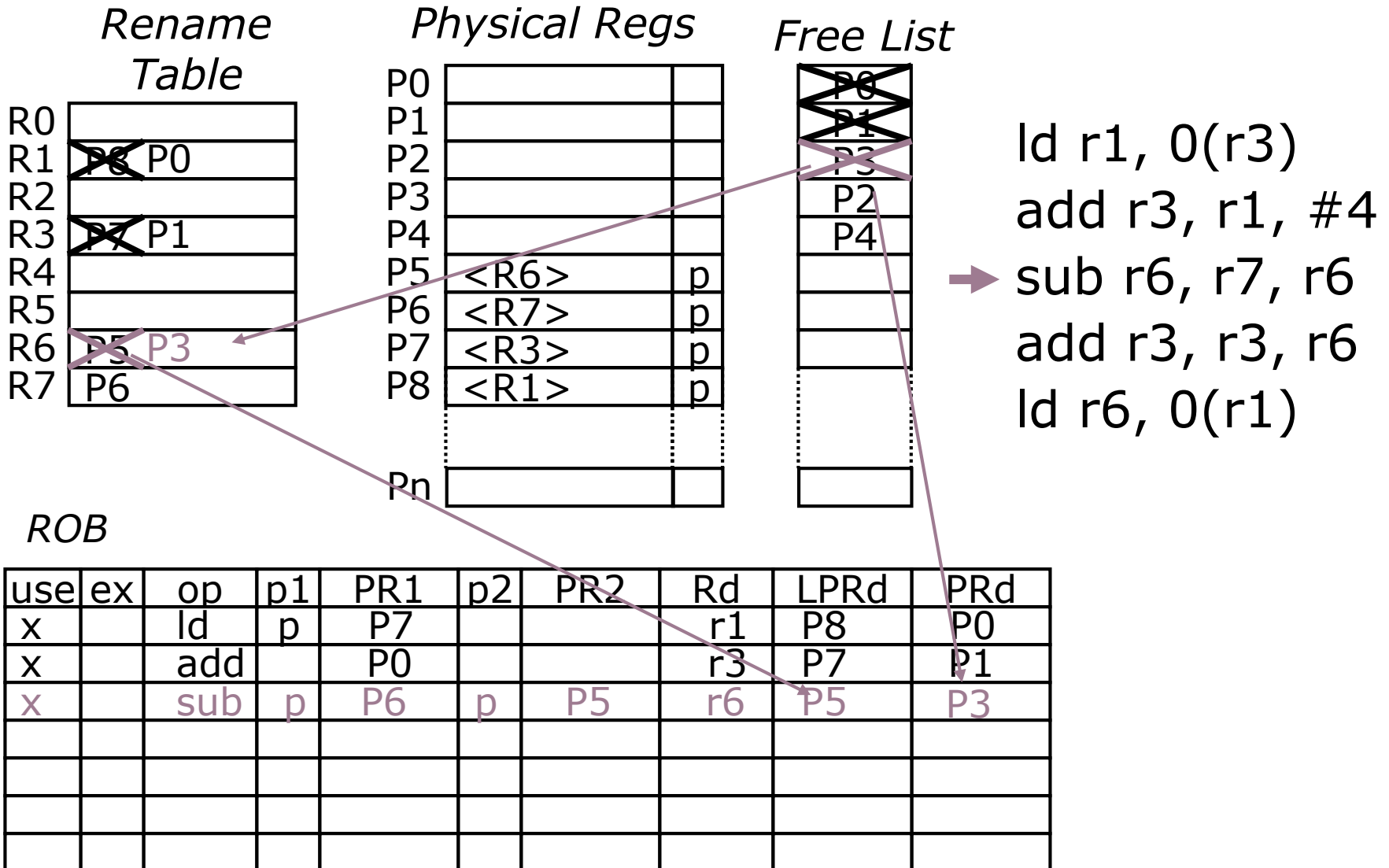
Physical Register Management



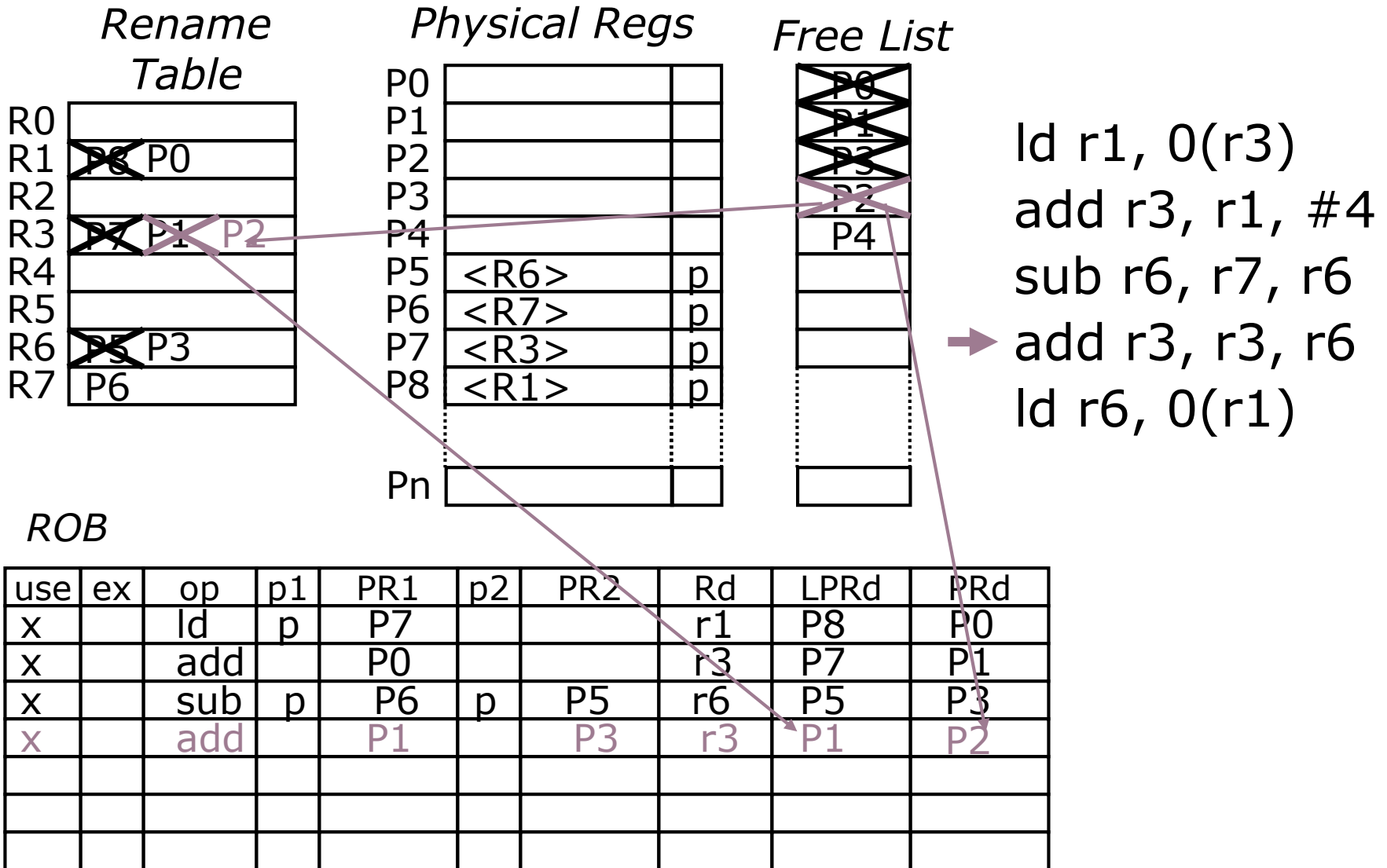
Physical Register Management



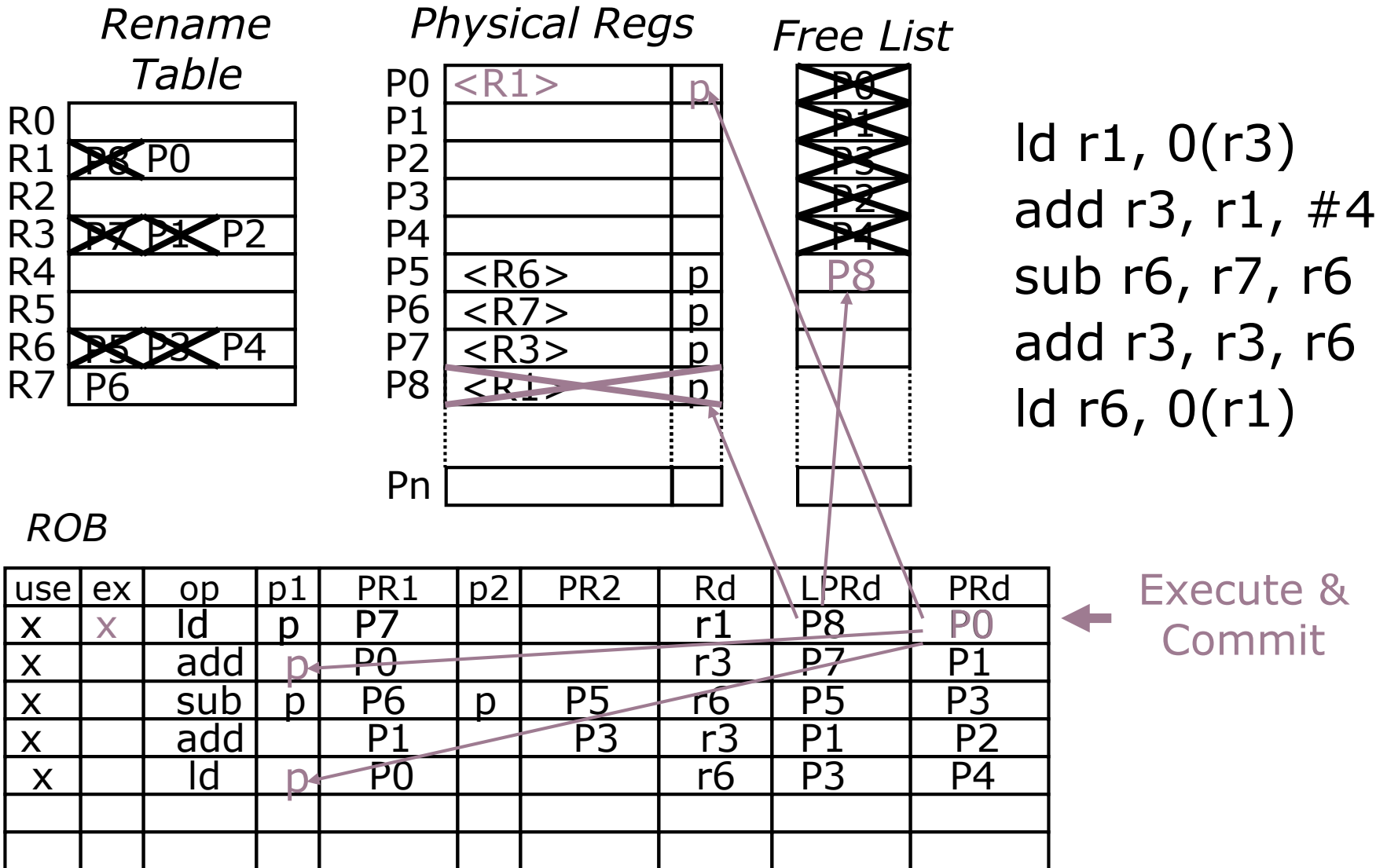
Physical Register Management



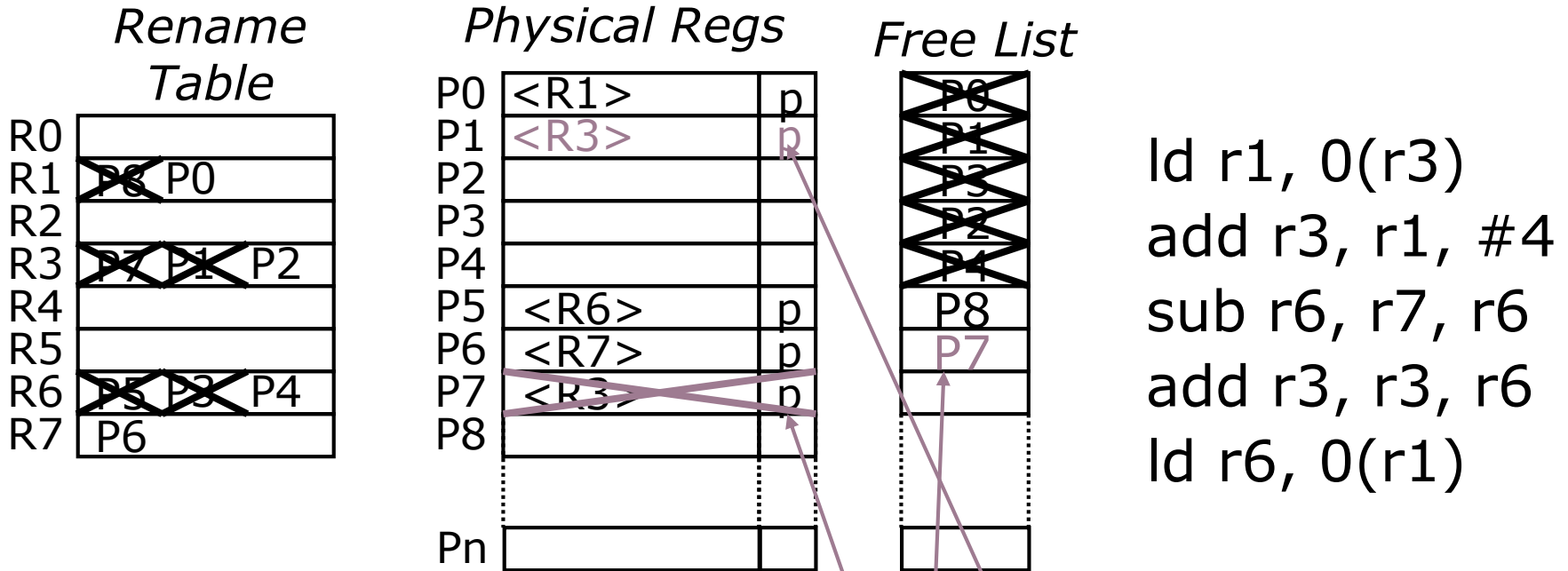
Physical Register Management



Physical Register Management



Physical Register Management

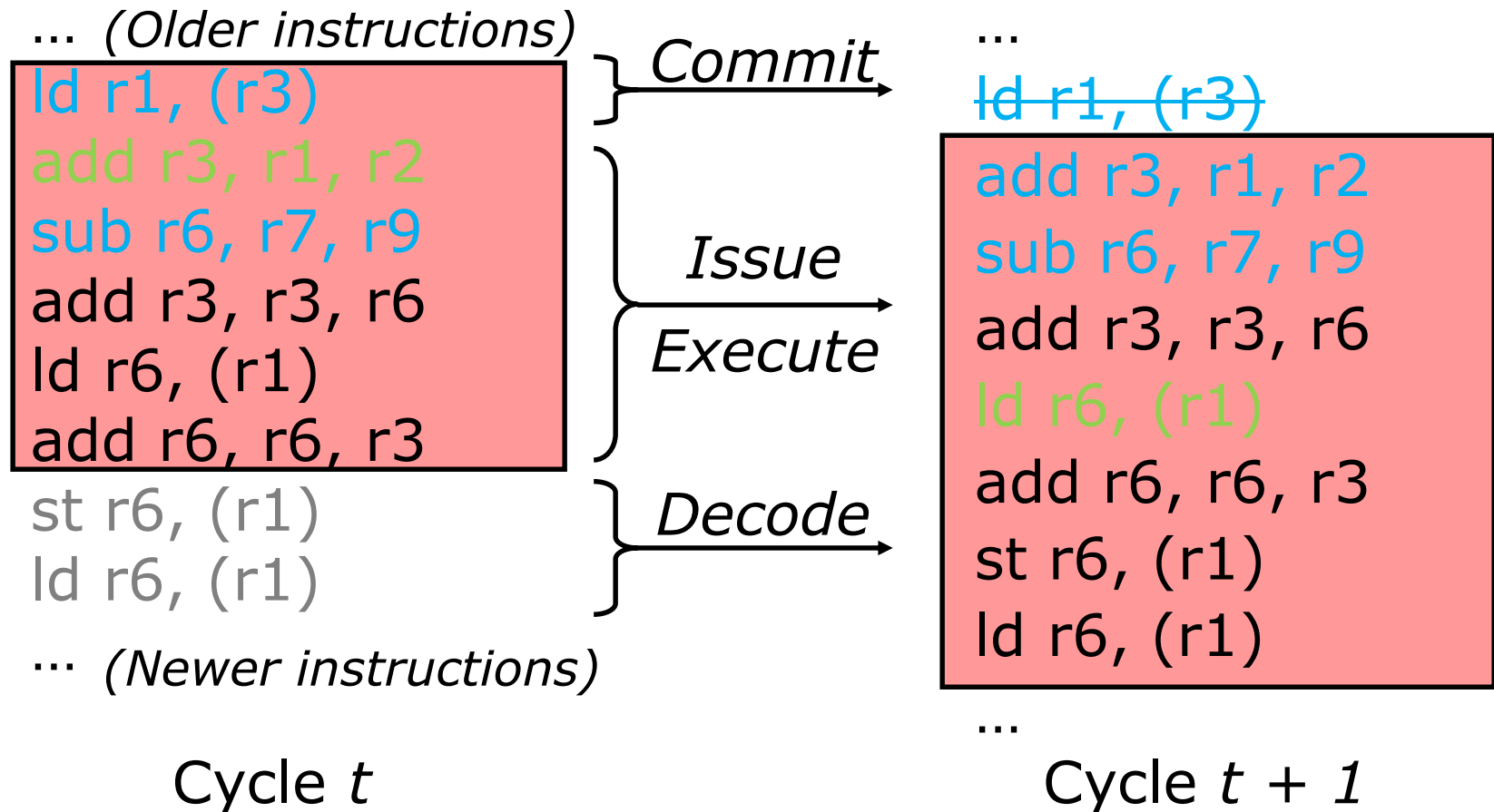


ROB

| 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



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?

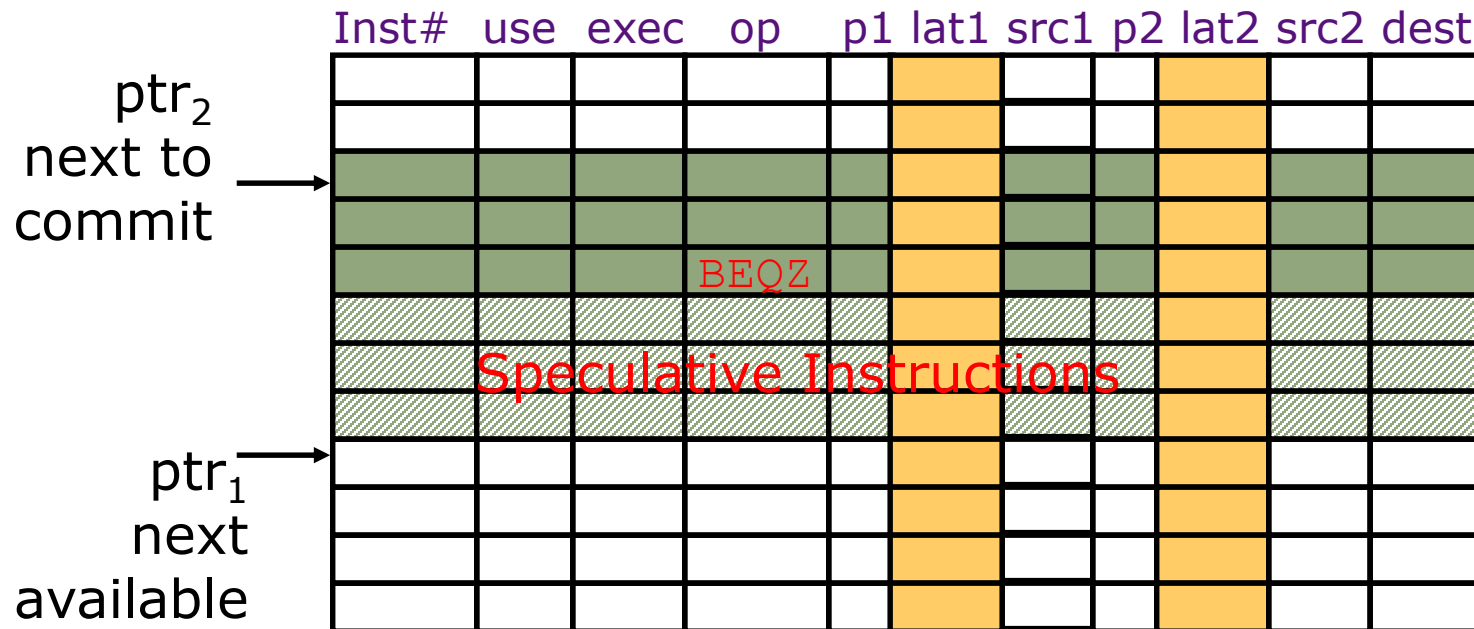
Using knowledge of execution latency (bypass)

| | | | | | |
|----|--------------|--------------------|----------------------|----------------------|--|
| 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

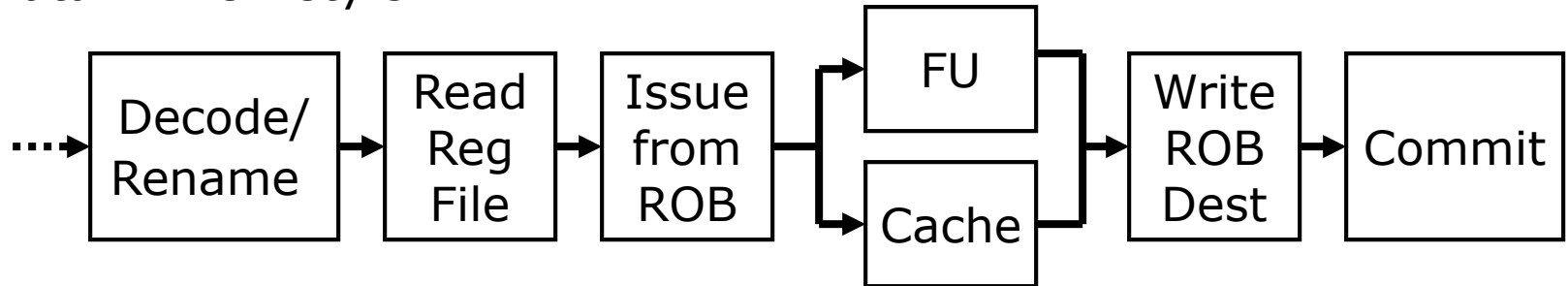


Issue Queue (Reorder buffer)

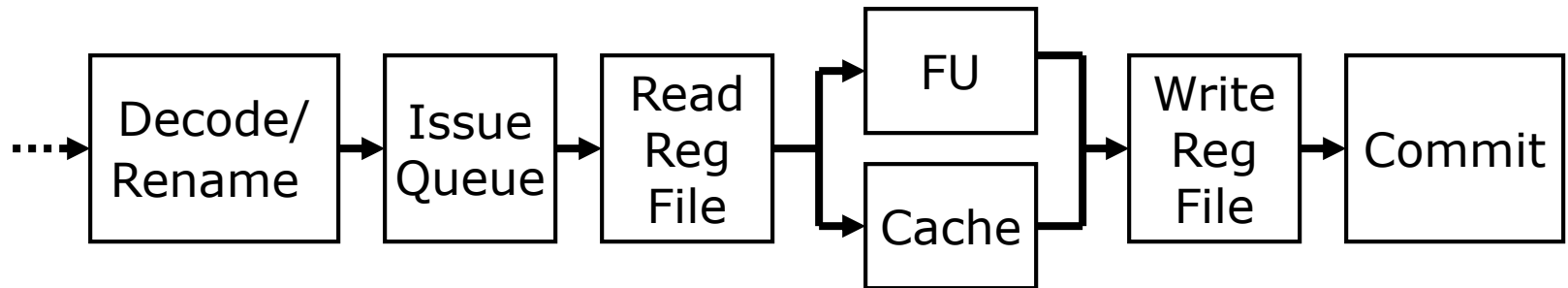
- Fixed latency: latency included in queue entry ('bypassed')
- Predicted latency: latency included in queue entry (speculated)
- Variable latency: wait for completion signal (stall)

Data-in-ROB vs. Unified RegFile

Data-in-ROB style



Unified-register-file style

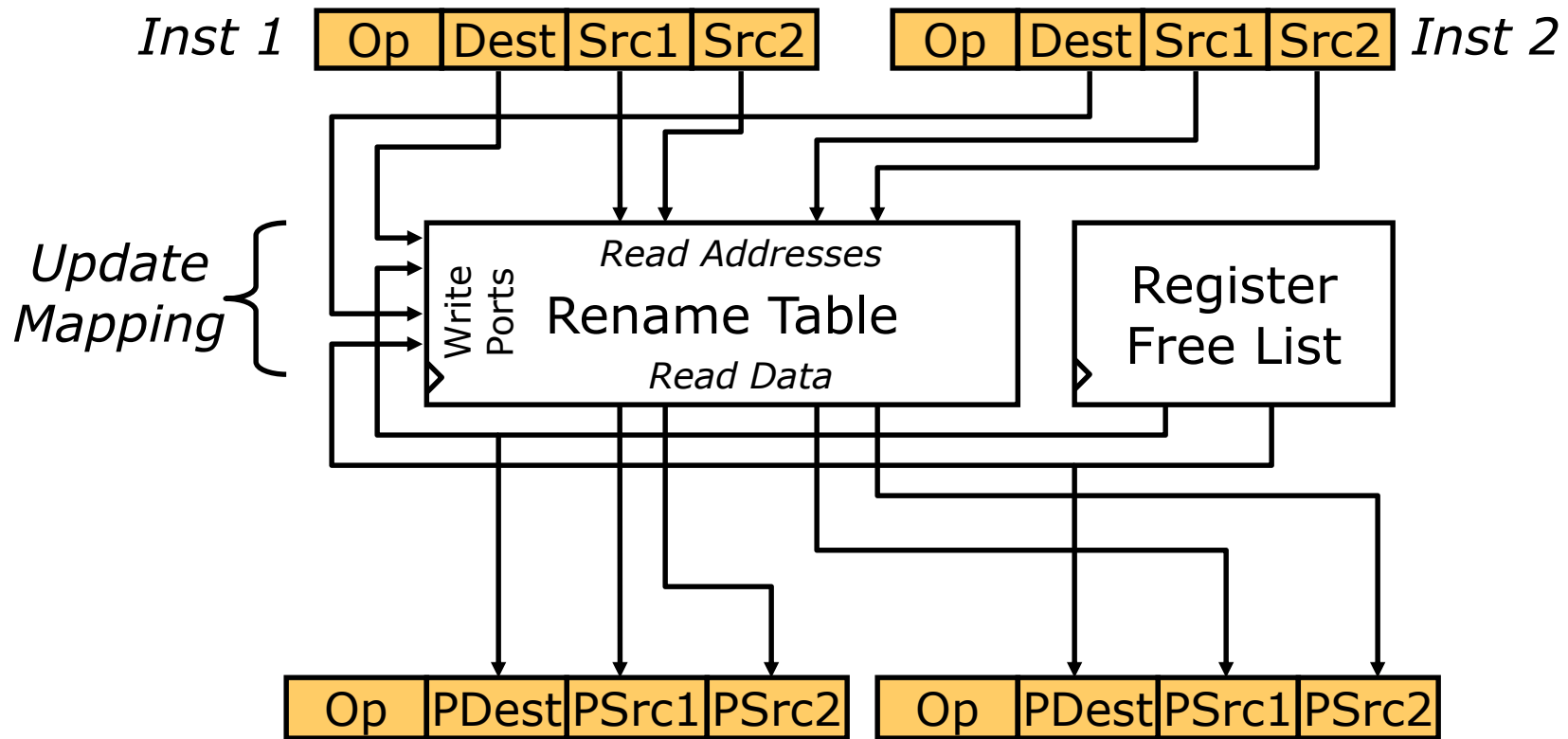


How does issue speculation differ, e.g., on cache miss?

Dependency loop shorter for data-in-ROB style

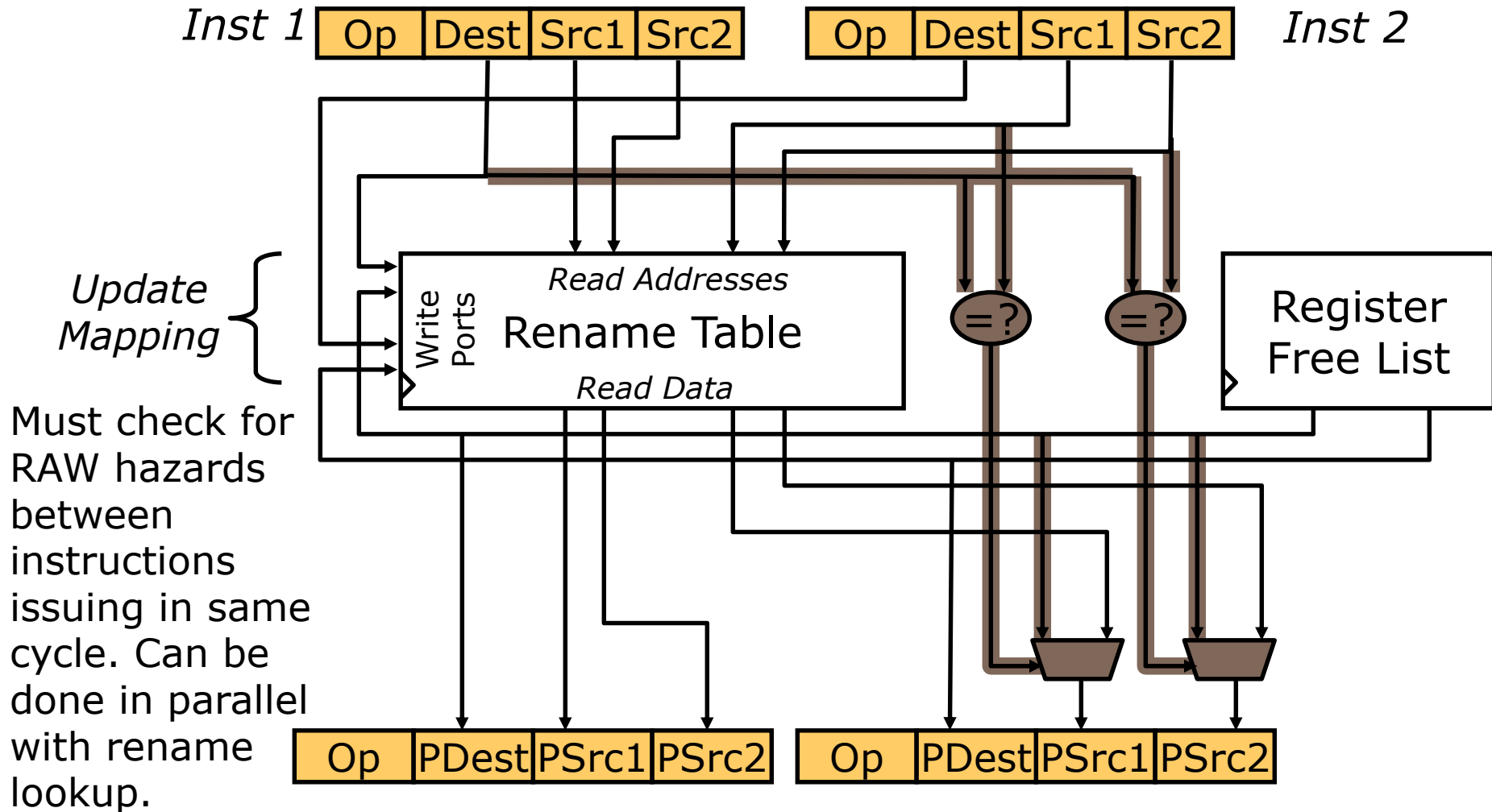
Superscalar Register Renaming

- 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



(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!