CS 152 Computer Architecture and Engineering

Lecture 14 - Advanced Superscalars

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

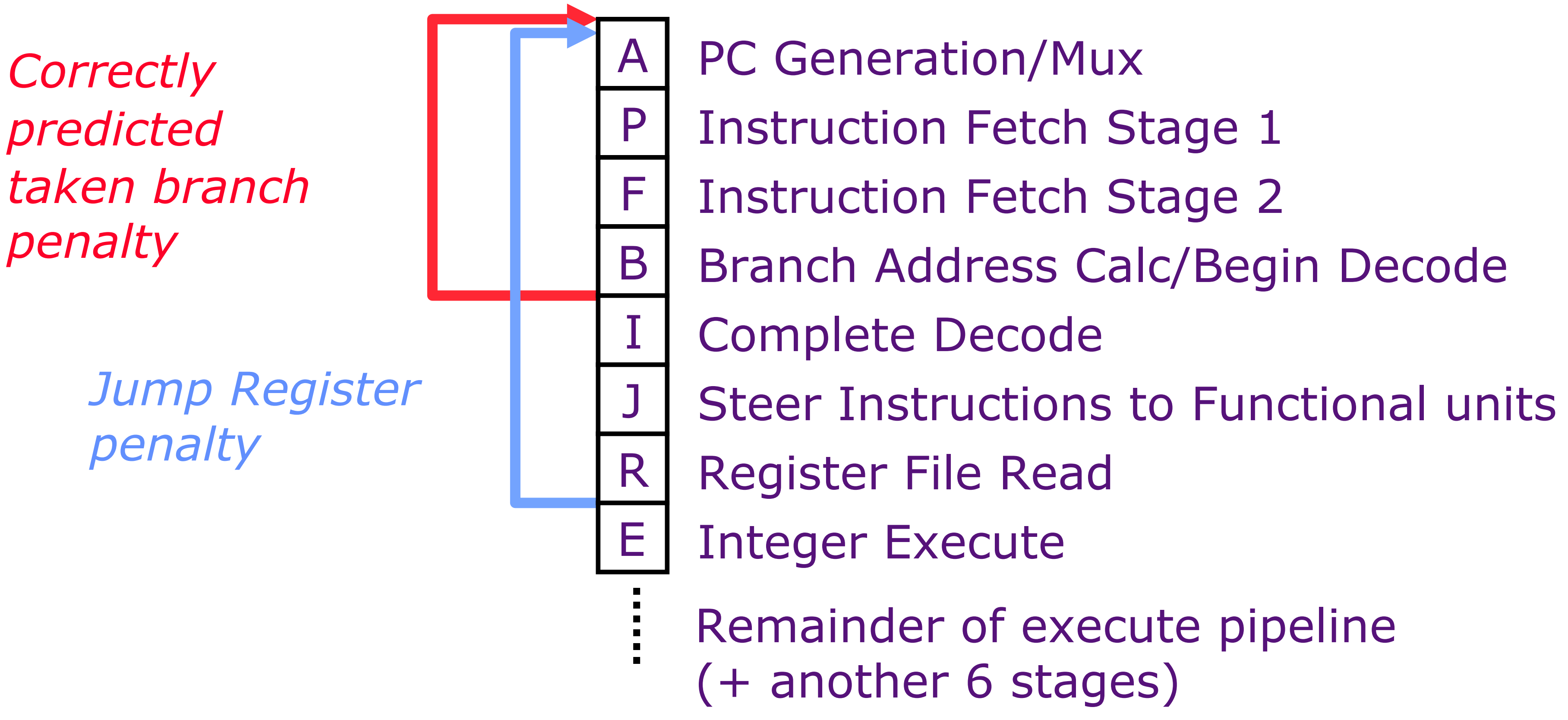
http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152

Last time in Lecture 13

- Register renaming removes WAR, WAW hazards
- Instruction execution divided into four major stages:
  - Instruction Fetch, Decode/Rename, Execute/Complete, Commit
- Control hazards are serious impediment to superscalar performance
- Dynamic branch predictors can be quite accurate (>95%) and avoid most control hazards
- Branch History Tables (BHTs) just predict direction (later in pipeline)
  - Just need a few bits per entry (2 bits gives hysteresis)
  - Need to decode instruction bits to determine whether this is a branch and what the target address is

Limitations of BHTs

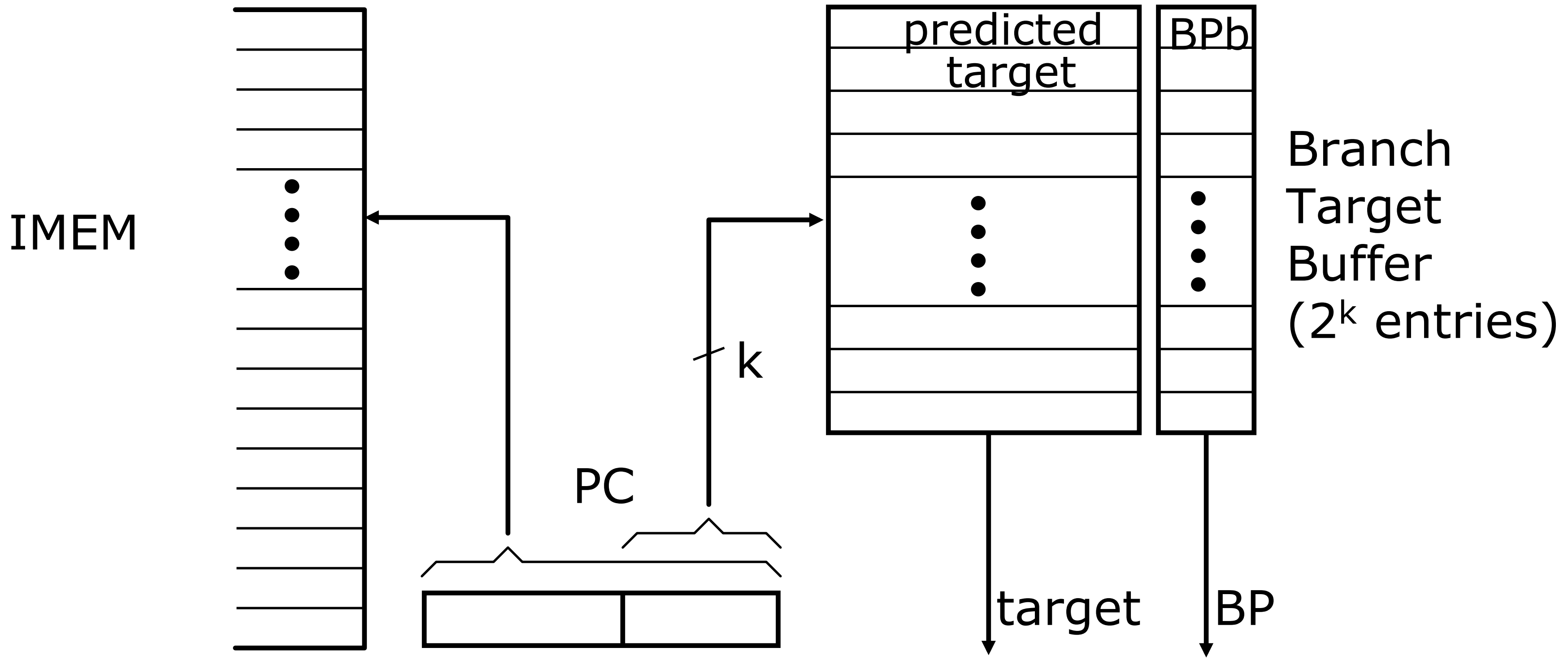
Only predicts branch direction. Therefore, cannot redirect fetch stream until after branch target is determined.



Correctly predicted taken branch penalty
Jump Register penalty
A
PC Generation/Mux
P
Instruction Fetch Stage 1
F
Instruction Fetch Stage 2
B
Branch Address Calc/Begin Decode
I
Complete Decode
J
Steer Instructions to Functional units
R
Register File Read
E
Integer Execute
Remainder of execute pipeline (+ another 6 stages)

UltraSPARC-III fetch pipeline

Branch Target Buffer

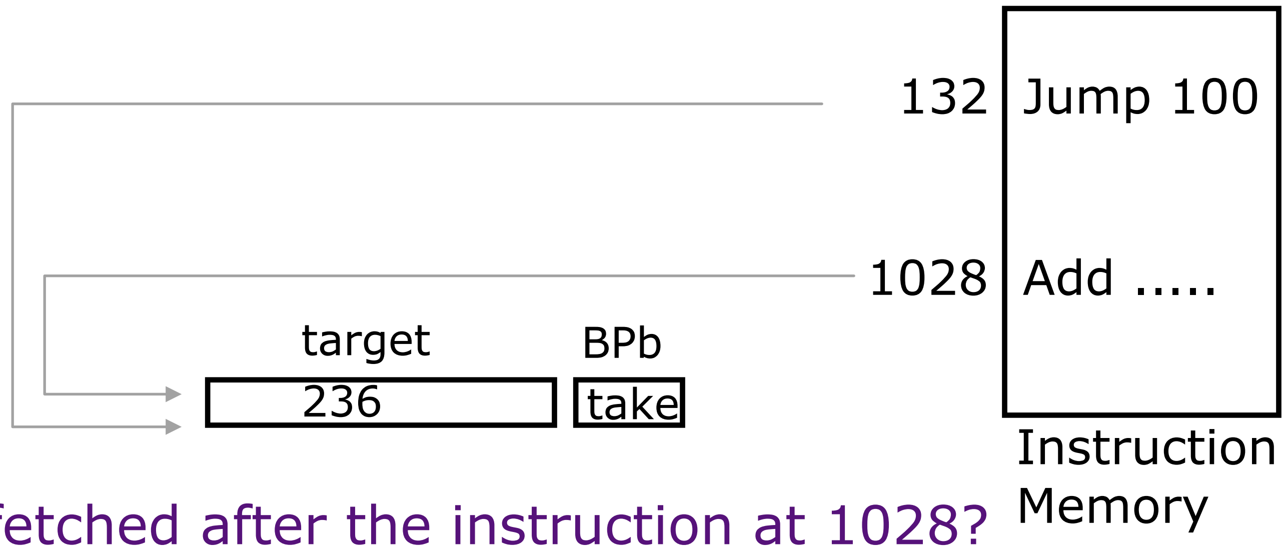


BP bits are stored with the predicted target address.

IF stage: *If (BP=taken) then nPC=target else nPC=PC+4*
later: *check prediction, if wrong then kill the instruction and update BTB & BPb else update BPb*

Address Collisions

Assume a 128-entry BTB



What will be fetched after the instruction at 1028?

BTB prediction = 236
Correct target = 1032

⇒ *kill* PC=236 and *fetch* PC=1032

Is this a common occurrence?
Can we avoid these bubbles?

BTB is only for Control Instructions

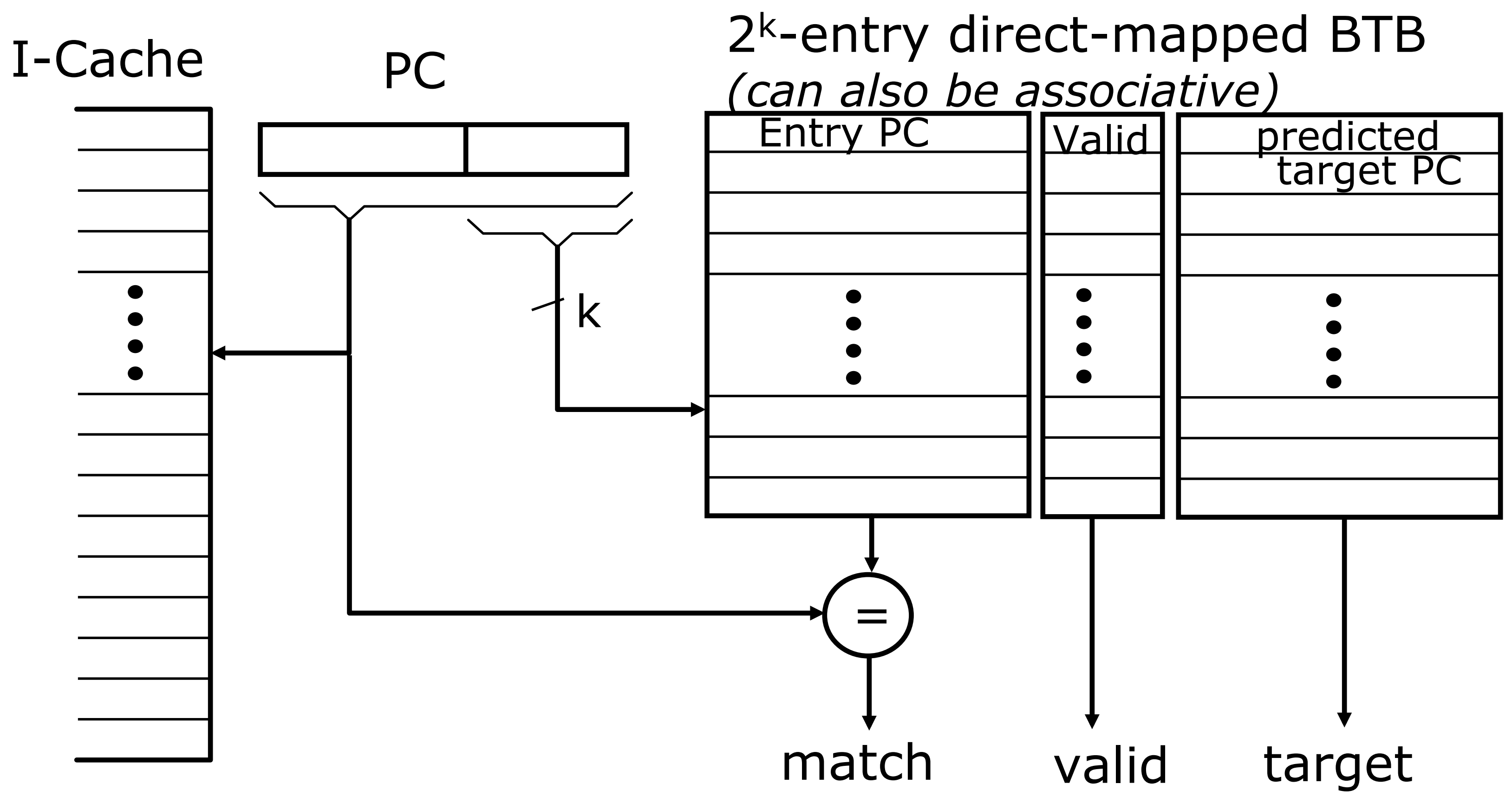
BTB contains useful information for branch and jump instructions only

$\Rightarrow$ Do not update it for other instructions

For all other instructions the next PC is PC+4 !

How to achieve this effect without decoding the instruction?

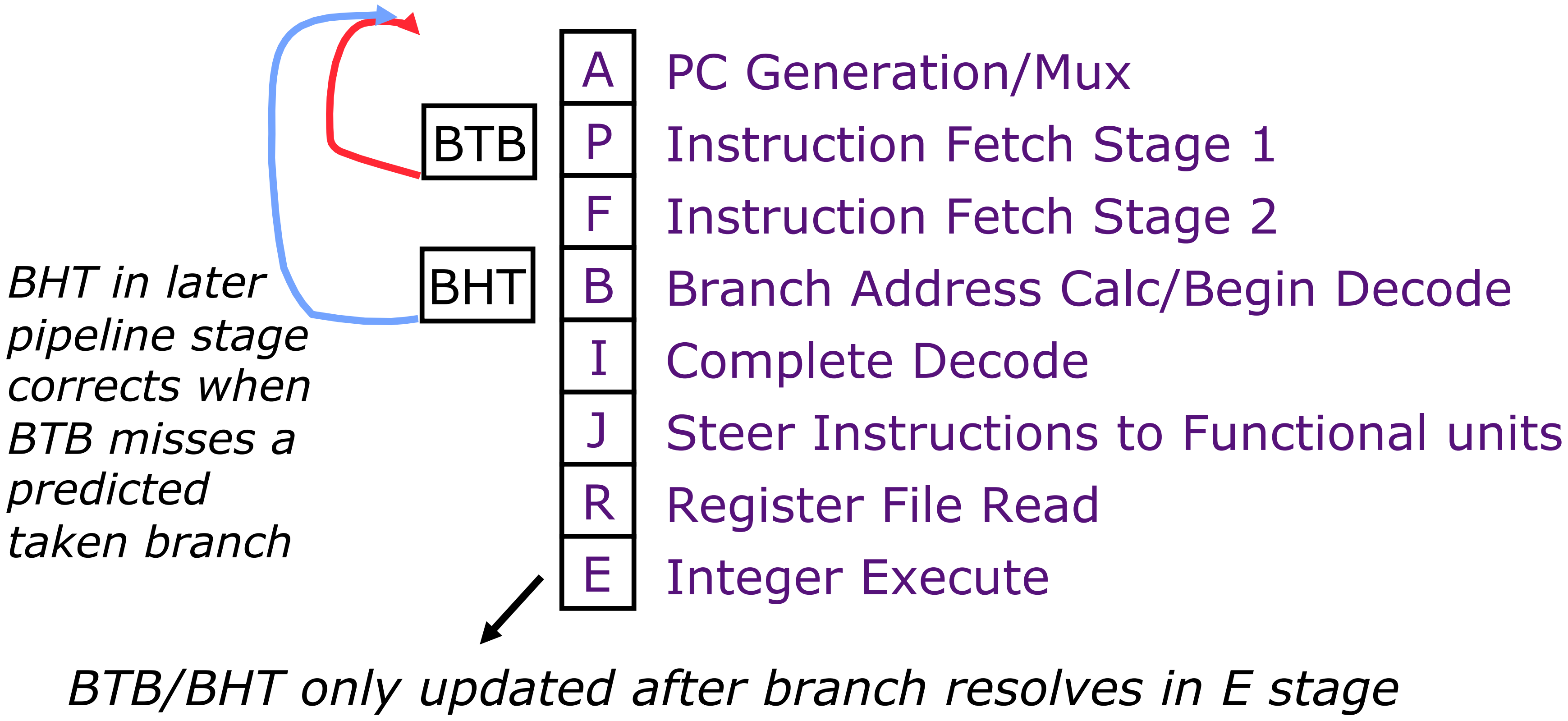
Branch Target Buffer (BTB)



- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only *taken* branches and jumps held in BTB
- Next PC determined *before* branch fetched and decoded

Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JR)
- BHT can hold many more entries and is more accurate



BTB
BHT
A
P
F
B
I
J
R
E
PC Generation/Mux
Instruction Fetch Stage 1
Instruction Fetch Stage 2
Branch Address Calc/Begin Decode
Complete Decode
Steer Instructions to Functional units
Register File Read
Integer Execute
BHT in later pipeline stage corrects when BTB misses a predicted taken branch
BTB/BHT only updated after branch resolves in E stage

Uses of Jump Register (JR)

- Switch statements (jump to address of matching case)

  BTB works well if same case used repeatedly

- Dynamic function call (jump to run-time function address)

  BTB works well if same function usually called, (e.g., in C++ programming, when objects have same type in virtual function call)

- Subroutine returns (jump to return address)

  BTB works well if usually return to the same place

  ⇒ *Often one function called from many distinct call sites!*

How well does BTB work for each of these cases?

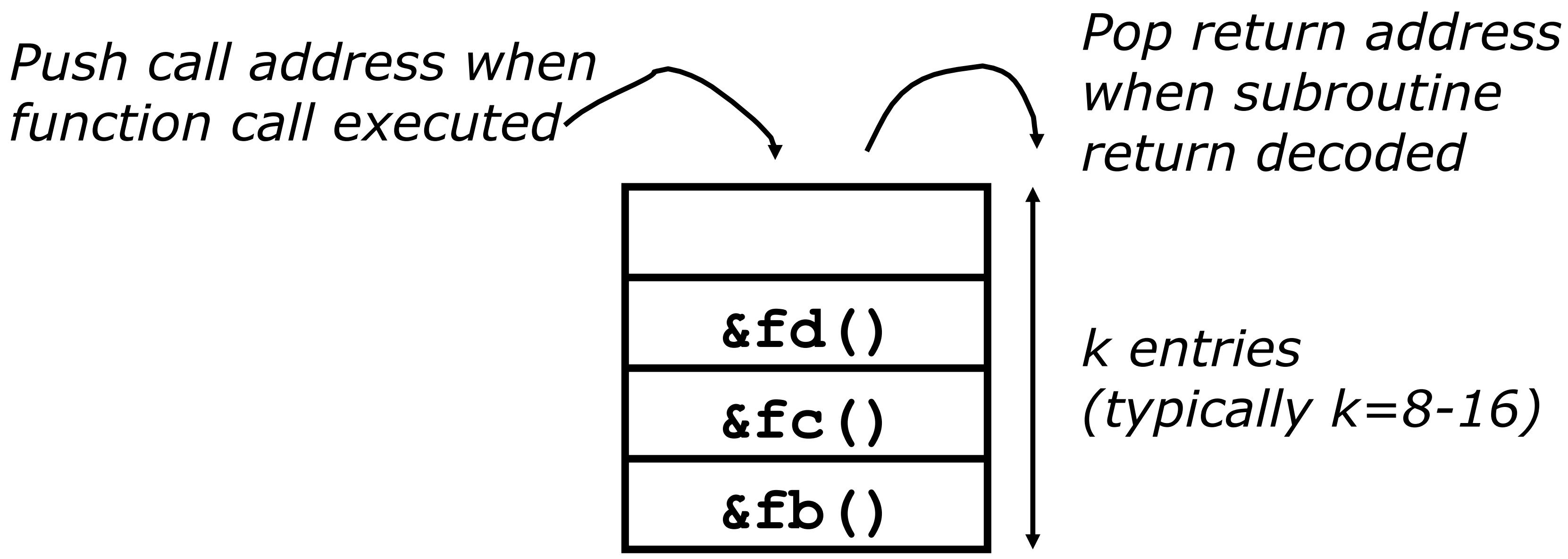
Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```
fa() { fb(); }

fb() { fc(); }

fc() { fd(); }
```



Push call address when function call executed
Pop return address when subroutine return decoded
&fd()
&fc()
&fb()
k entries (typically k=8-16)

Mispredict Recovery

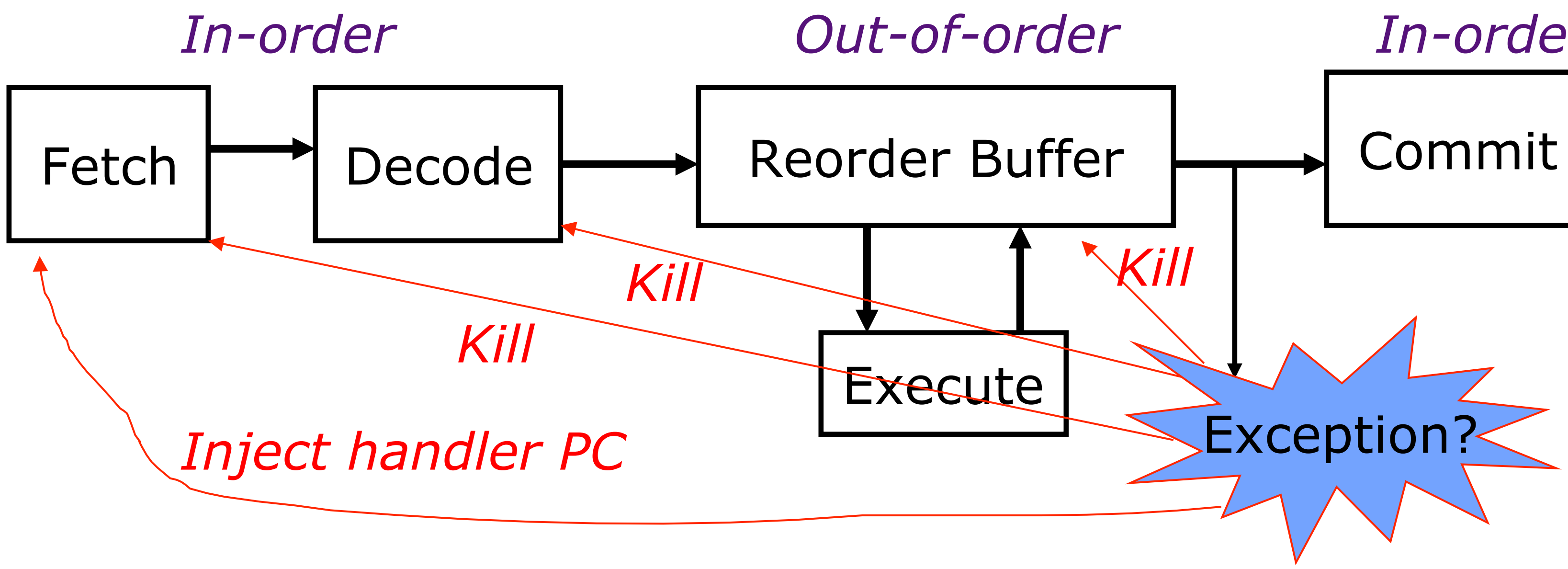
In-order execution machines:

- 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

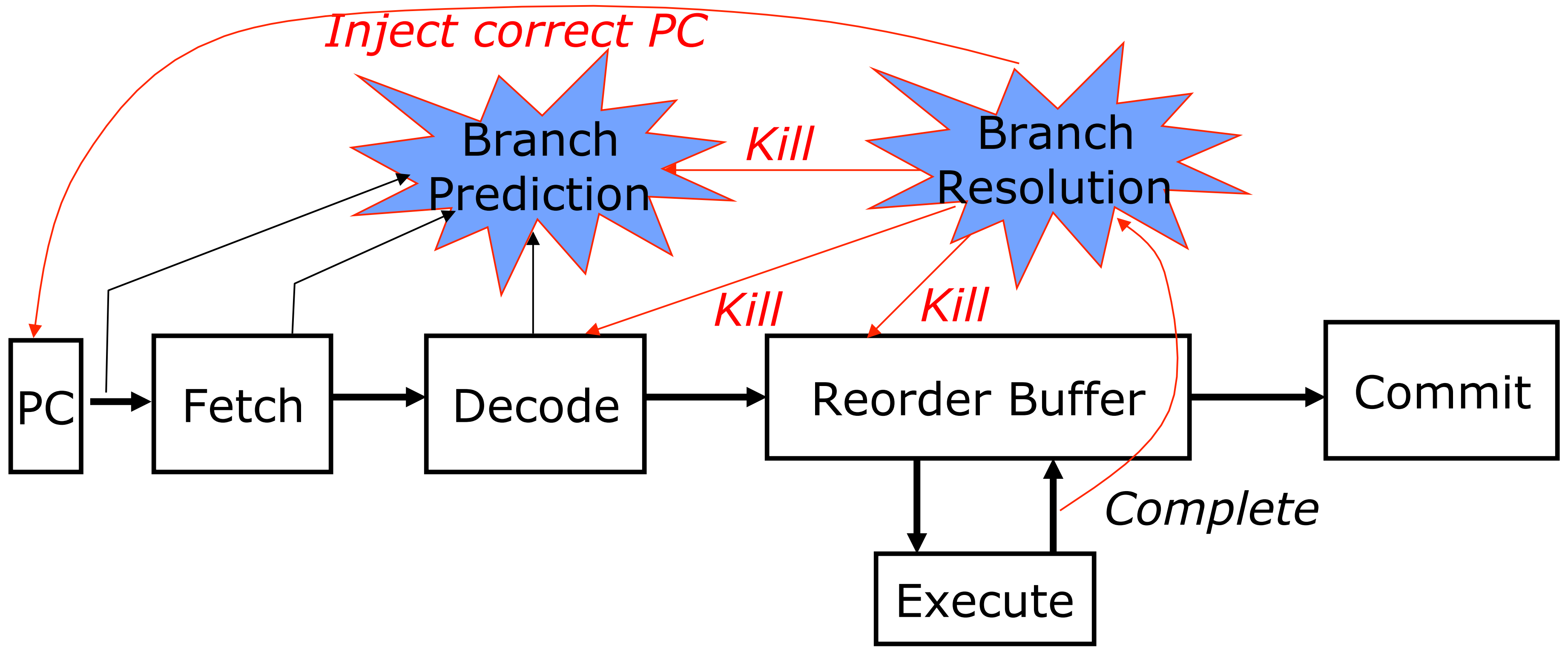
In-Order Commit for Precise Exceptions



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

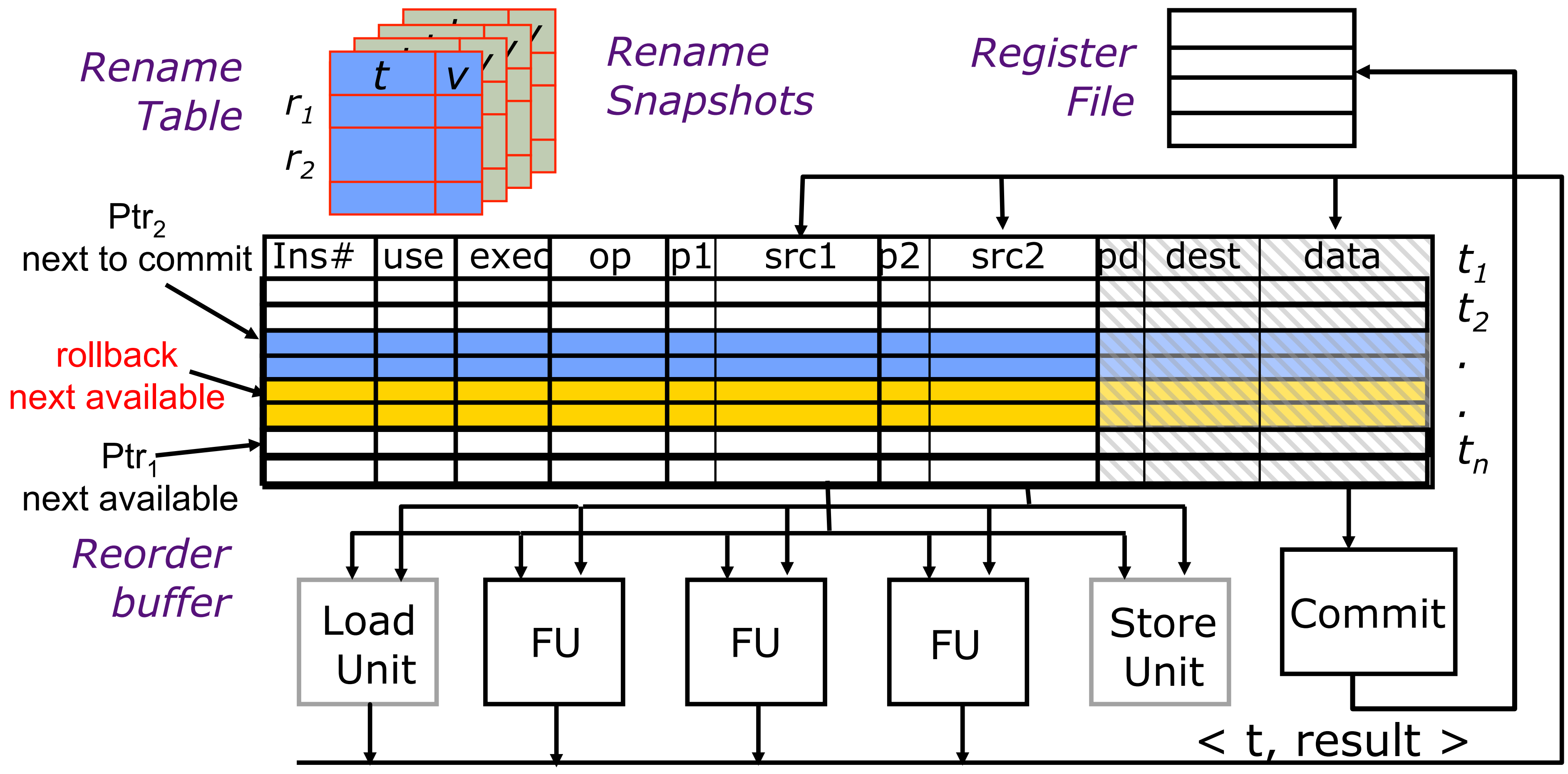
Temporary storage needed in ROB to hold results before commit

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

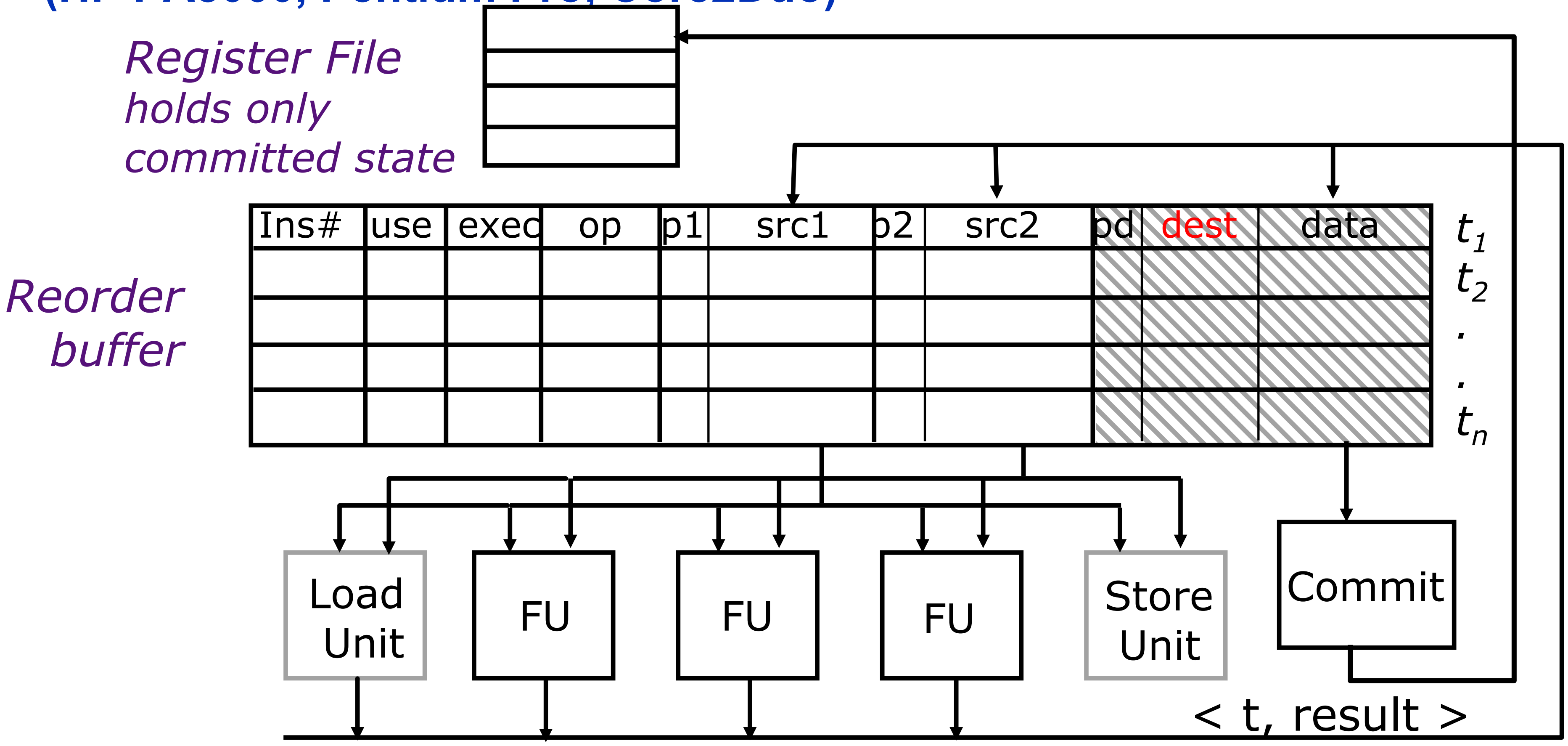
Speculating Both Directions

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

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

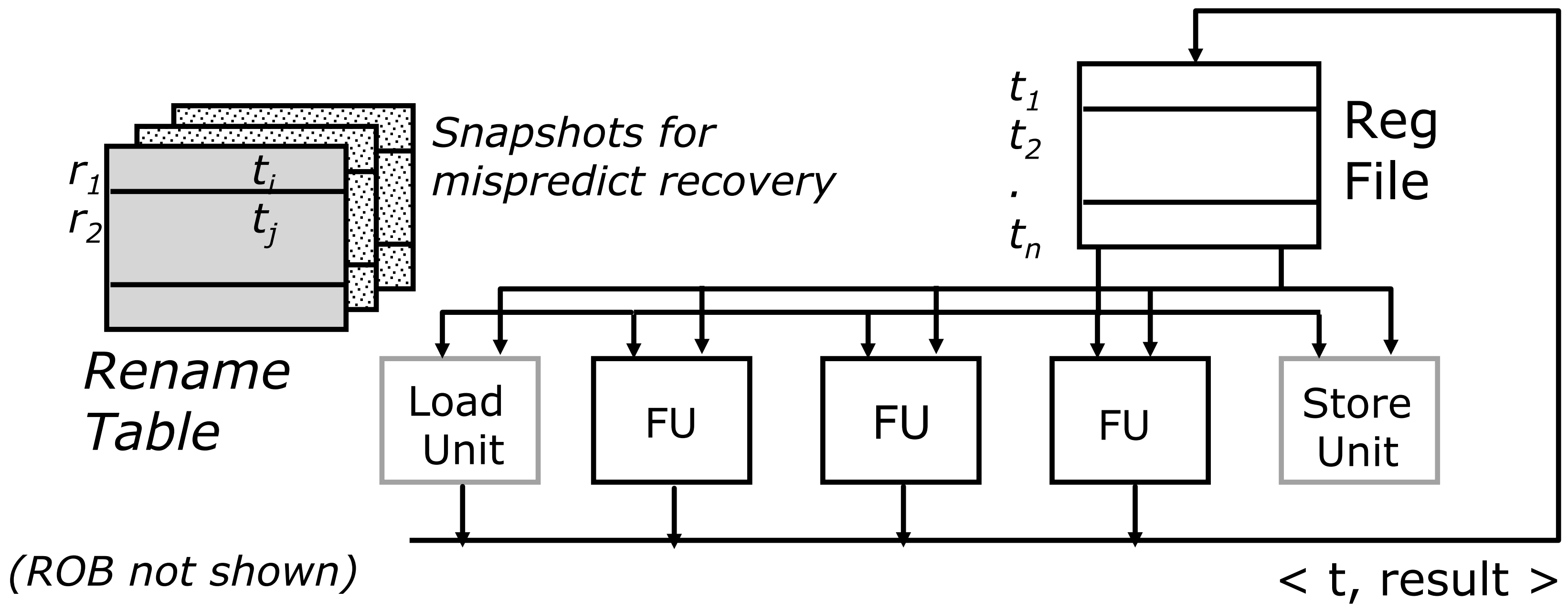


- On dispatch into ROB, ready sources can be in regfile or in ROB dest (copied into src1/src2 if ready before dispatch)
- On completion, write to dest field and broadcast to src fields.
- On issue, read from ROB src fields

CS152 Administrivia

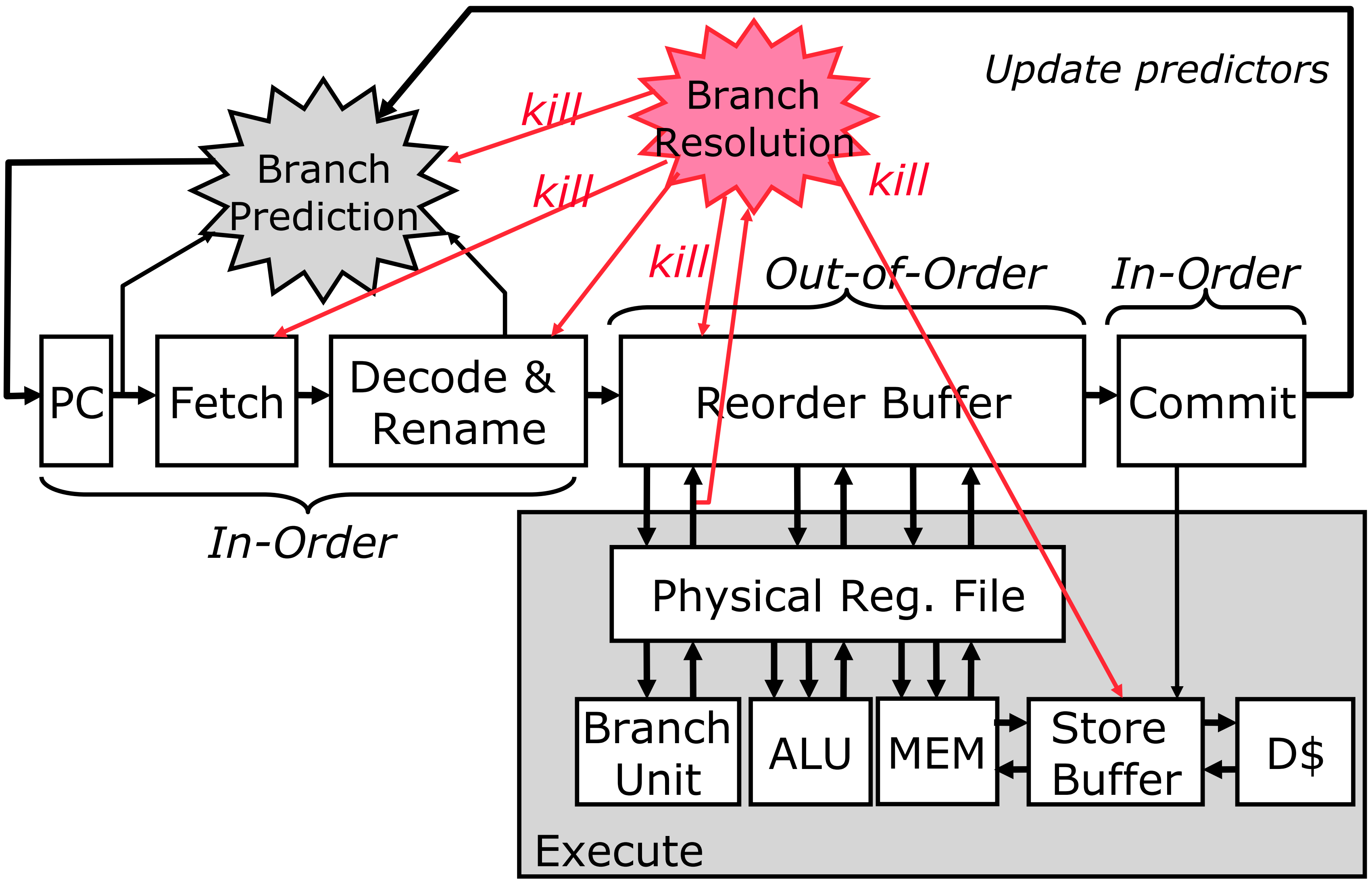
- Quiz 2 results

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)*

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 r1, (r3)
add r3, r1, #4
sub r6, r7, r9
add r3, r3, r6
ld r6, (r1)
add r6, r6, r3
st r6, (r1)
ld r6, (r11)
```

Rename

```
ld P1, (Px)
add P2, P1, #4
sub P3, Py, Pz
add P4, P2, P3
ld P5, (P1)
add P6, P5, P4
st 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

R0	
R1	P8
R2	
R3	P7
R4	
R5	
R6	P5
R7	P6

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5	<R6>	p
P6	<R7>	p
P7	<R3>	p
P8	<R1>	p
Pn		

Free List

P0
P1
P3
P2
P4

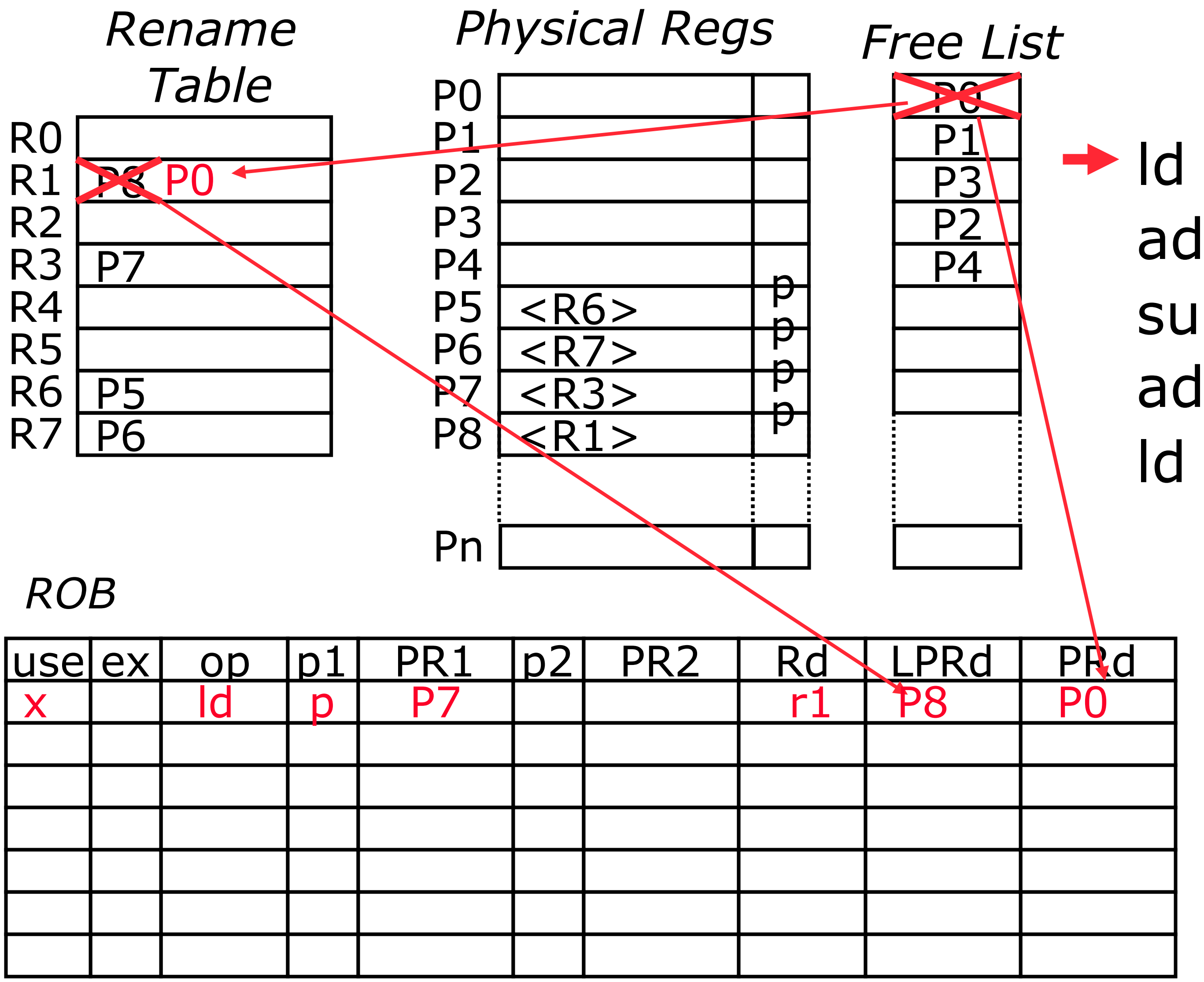
```
ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)
```

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd

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

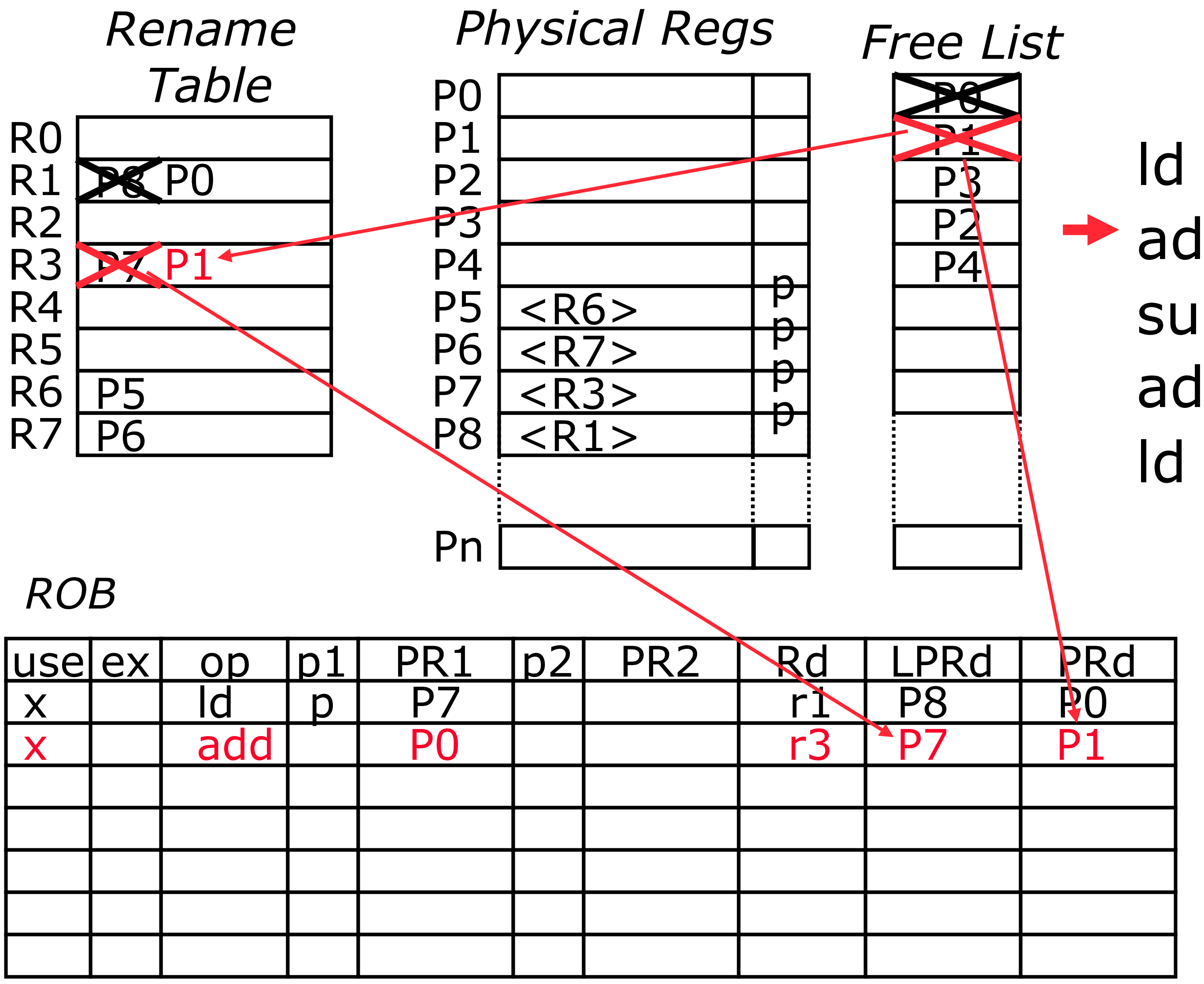
Physical Register Management



use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			r1	P8	P0

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

Physical Register Management



Rename Table

R0	
R1	~~P8~~ P0
R2	
R3	~~P7~~ P1
R4	
R5	
R6	P5
R7	P6

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5	<R6>	p
P6	<R7>	p
P7	<R3>	p
P8	<R1>	p
Pn		

Free List

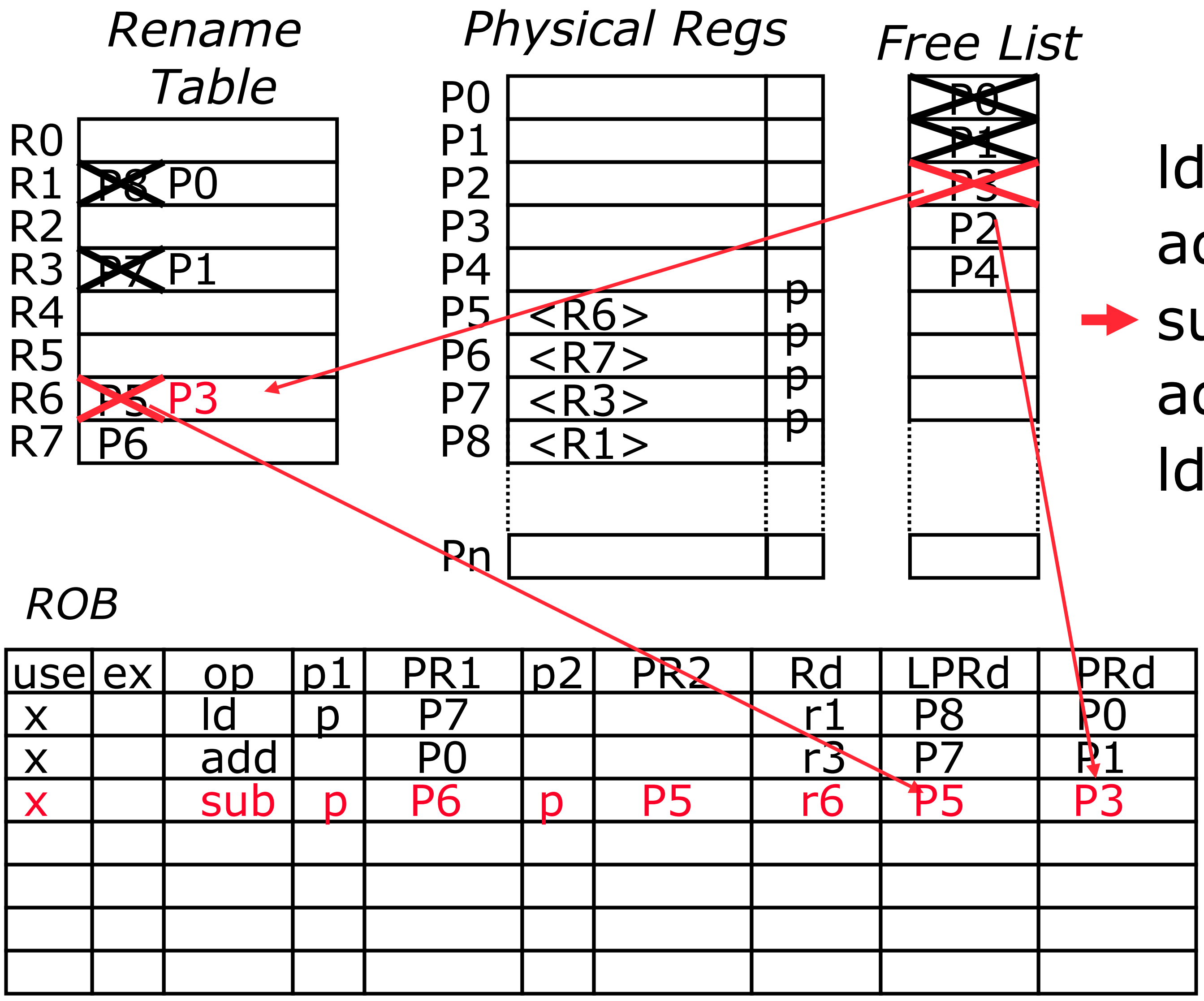
~~P0~~
~~P1~~
P3
P2
P4

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			r1	P8	P0
x		add		P0			r3	P7	P1

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

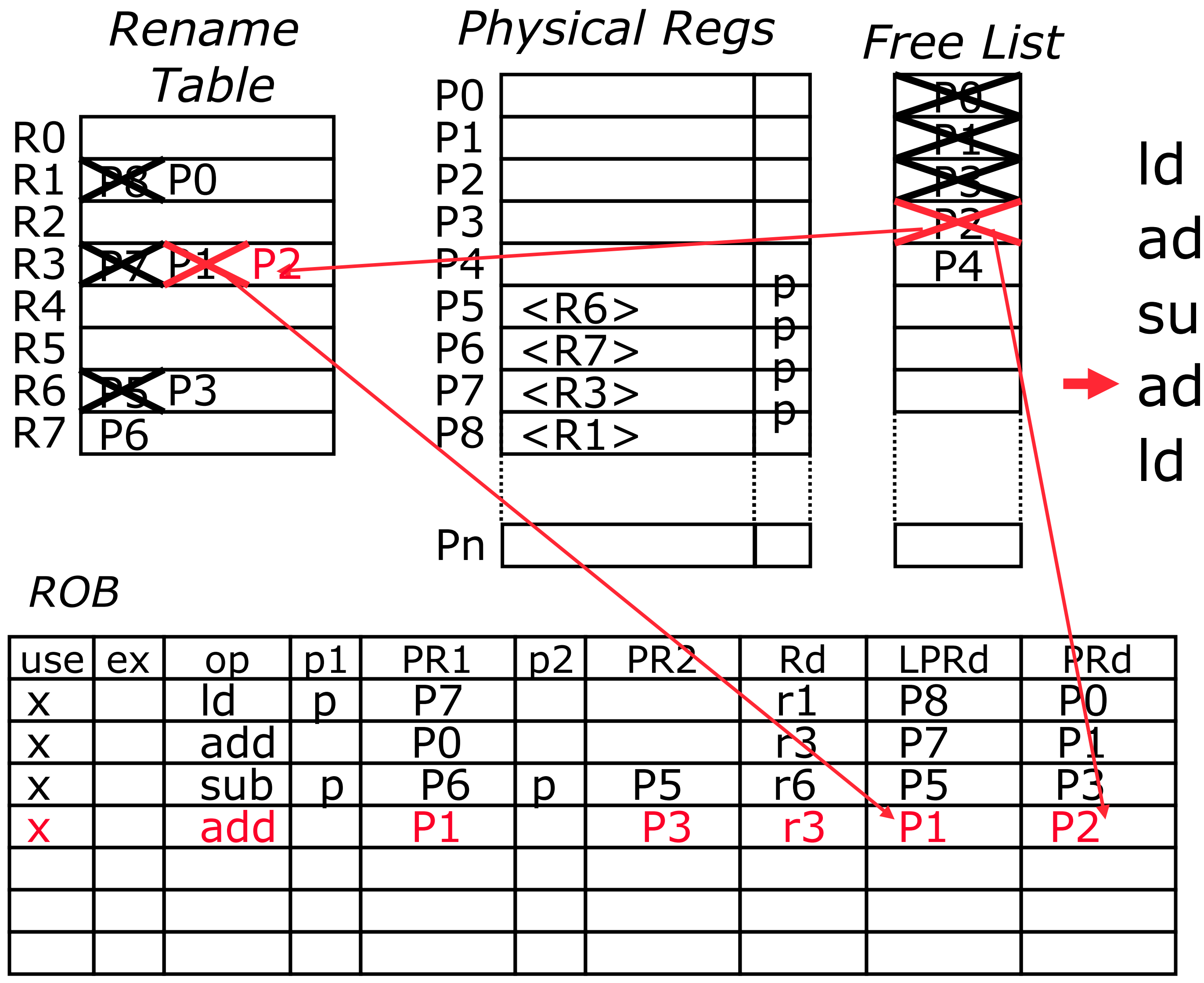
Physical Register Management



use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			r1	P8	P0
x		add		P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3

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

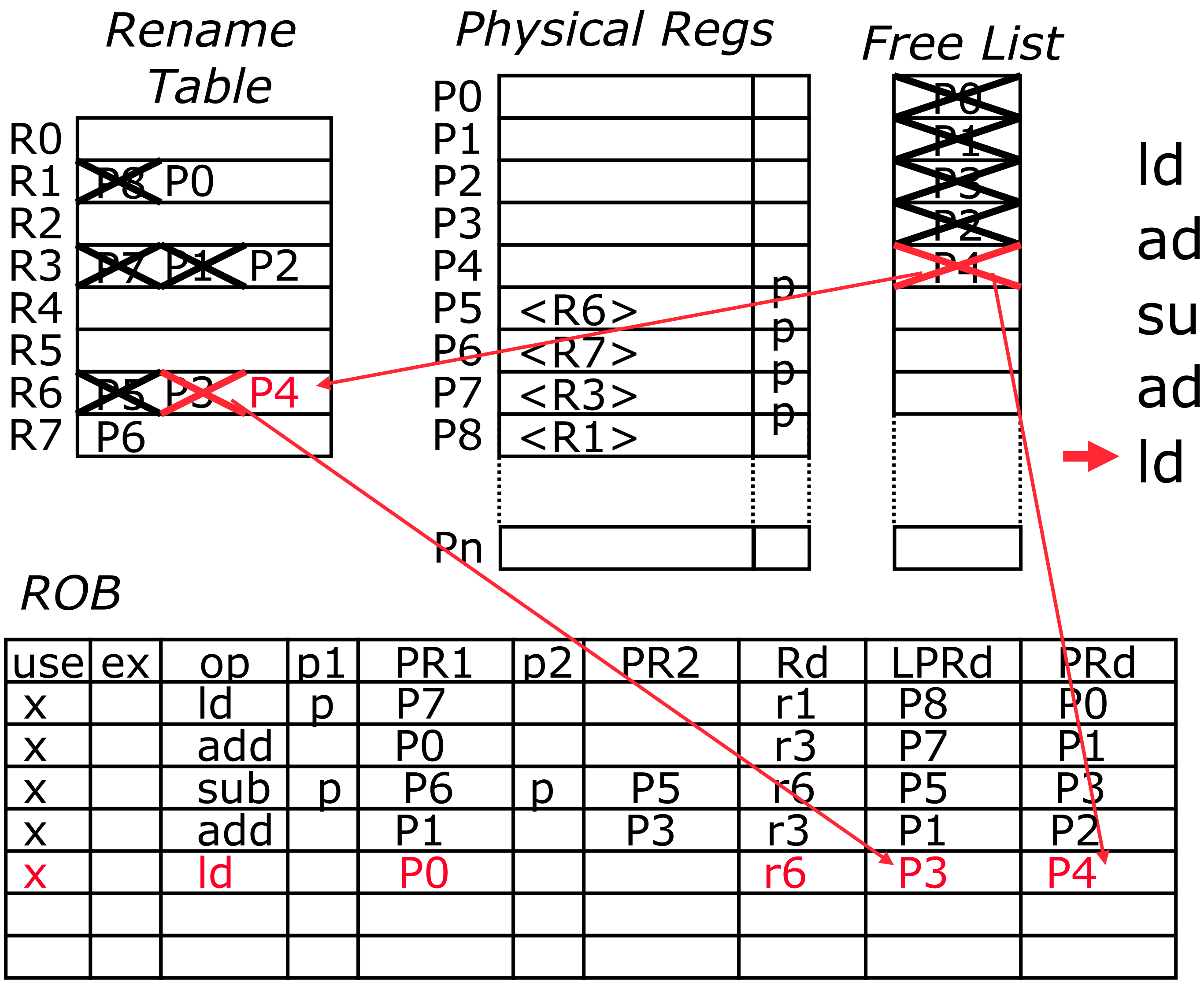
Physical Register Management



use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			r1	P8	P0
x		add		P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3
x		add		P1		P3	r3	P1	P2

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

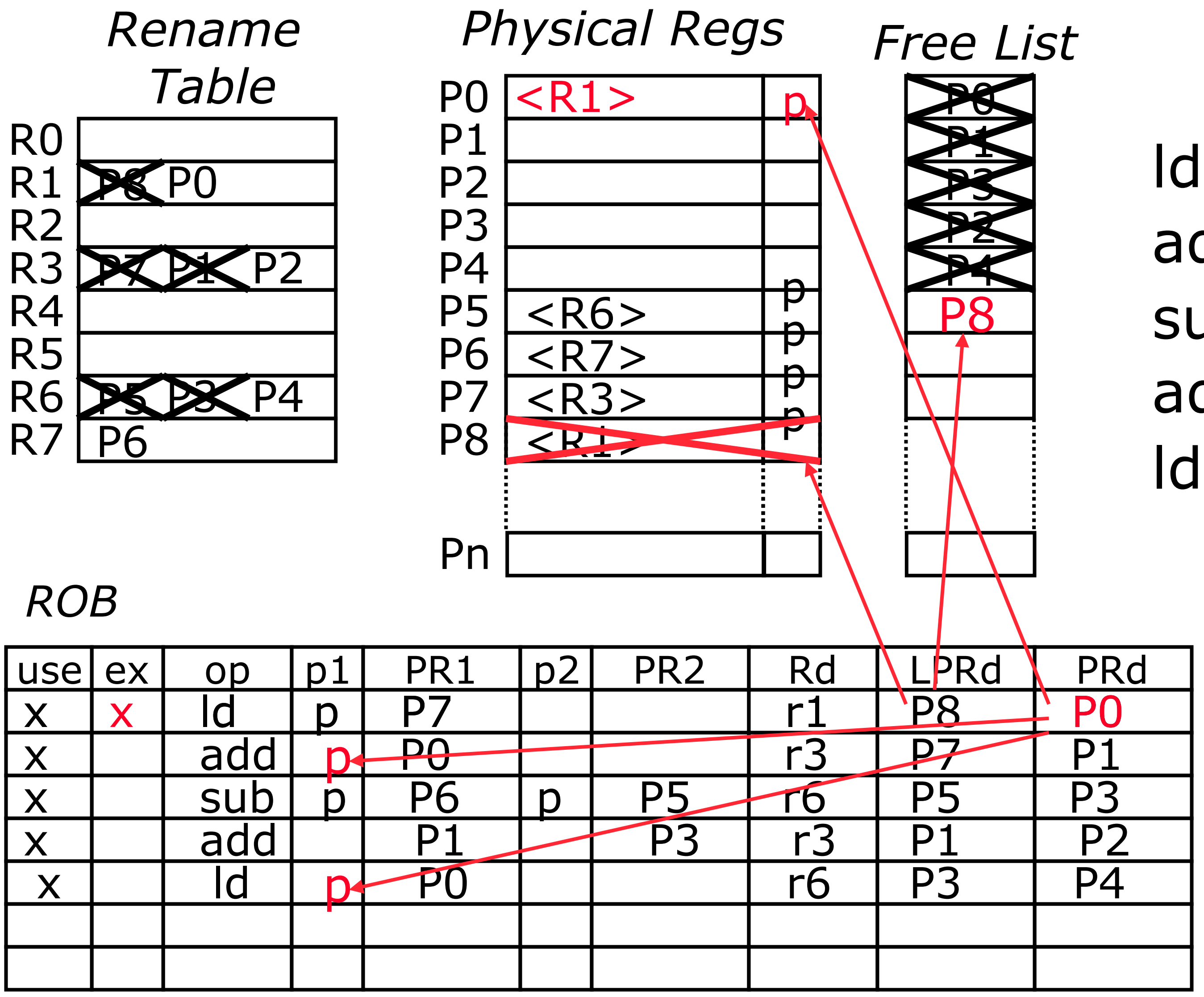
Physical Register Management



use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			r1	P8	P0
x		add		P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3
x		add		P1		P3	r3	P1	P2
x		ld		P0			r6	P3	P4

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

Physical Register Management



Rename Table

R0	
R1	~~P8~~ P0
R2	
R3	~~P7~~ ~~P1~~ P2
R4	
R5	
R6	~~P5~~ ~~P3~~ P4
R7	P6

Physical Regs

P0	<R1>	p
P1		
P2		
P3		
P4		
P5	<R6>	p
P6	<R7>	p
P7	<R3>	p
P8	~~<R1>~~	~~p~~
Pn		

Free List

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

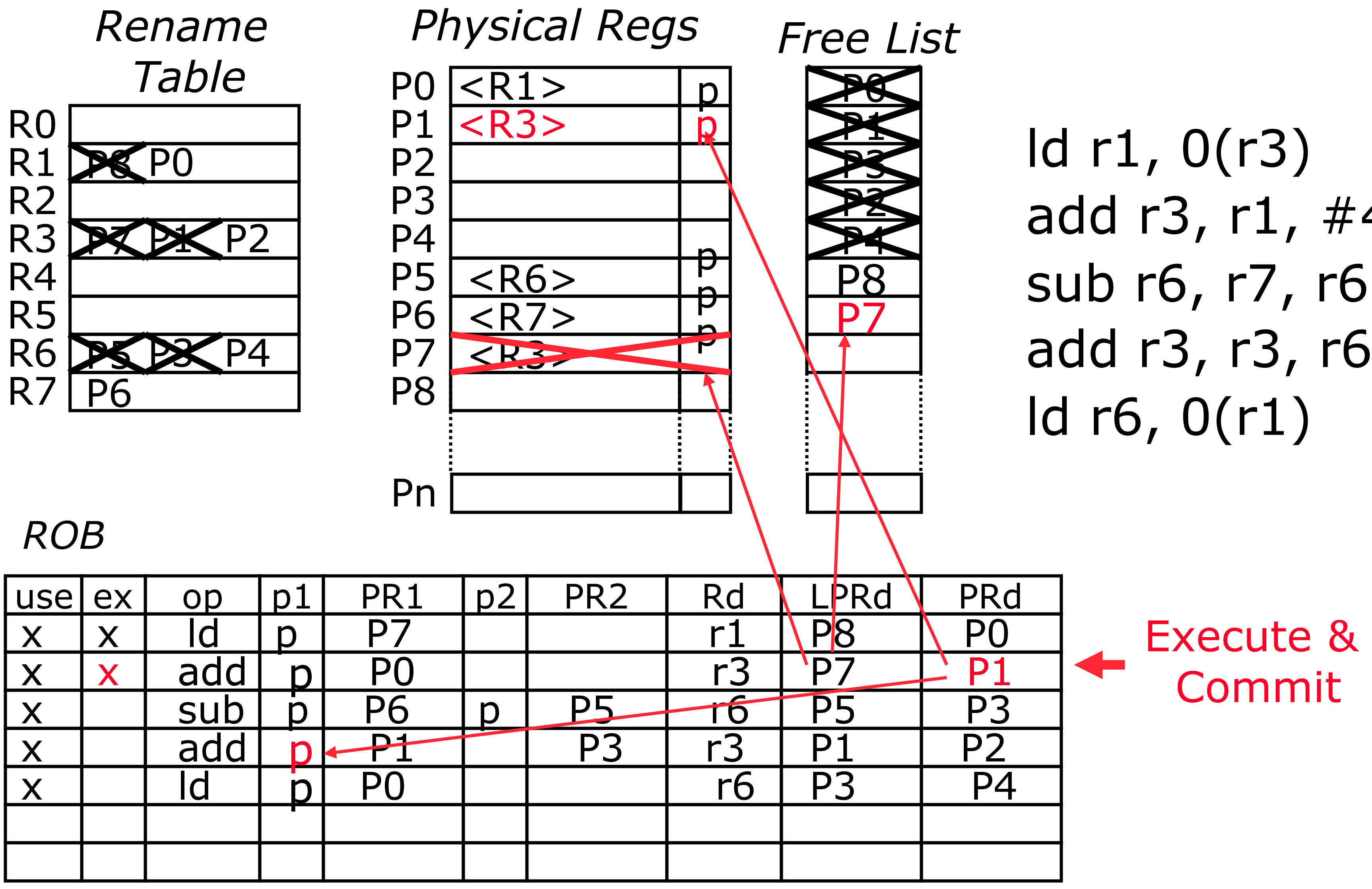
ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x	x	ld	p	P7			r1	P8	P0
x		add	p	P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3
x		add		P1		P3	r3	P1	P2
x		ld	p	P0			r6	P3	P4

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

Execute & Commit

Physical Register Management



Rename Table

R0	
R1	~~P8~~ P0
R2	
R3	~~P7~~ ~~P1~~ P2
R4	
R5	
R6	~~P5~~ ~~P3~~ P4
R7	P6

Physical Regs

P0	<R1>	p
P1	<R3>	p
P2		
P3		
P4		
P5	<R6>	p
P6	<R7>	p
P7	~~<R3>~~	~~p~~
P8		
Pn		

Free List

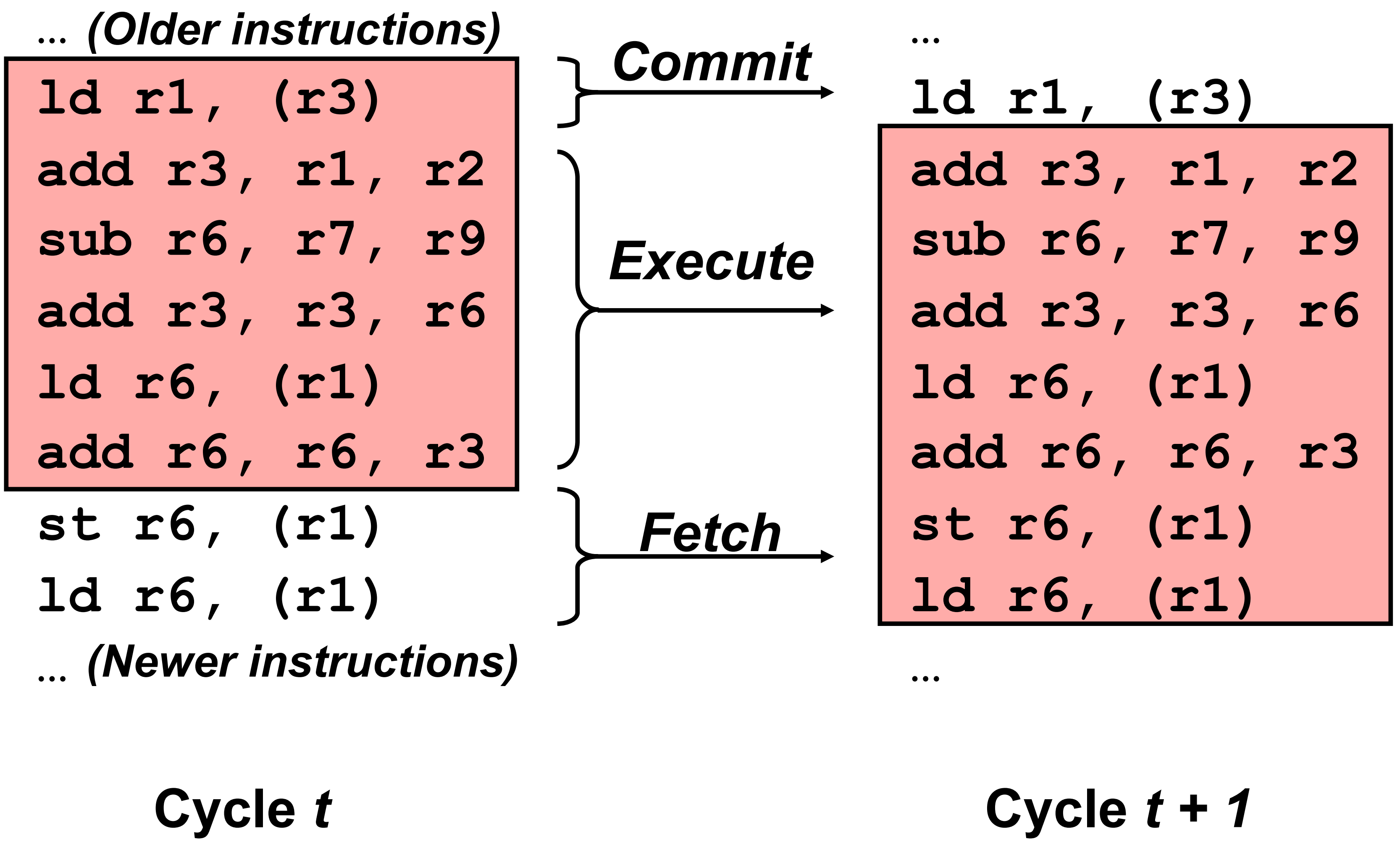
~~P0~~
~~P1~~
~~P3~~
~~P2~~
~~P4~~
P8
P7

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

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

Reorder Buffer Holds Active Instruction Window



… *(Older instructions)*

```
ld r1, (r3)
add r3, r1, r2
sub r6, r7, r9
add r3, r3, r6
ld r6, (r1)
add r6, r6, r3
st r6, (r1)
ld r6, (r1)
```

… *(Newer instructions)*

Cycle *t*

…

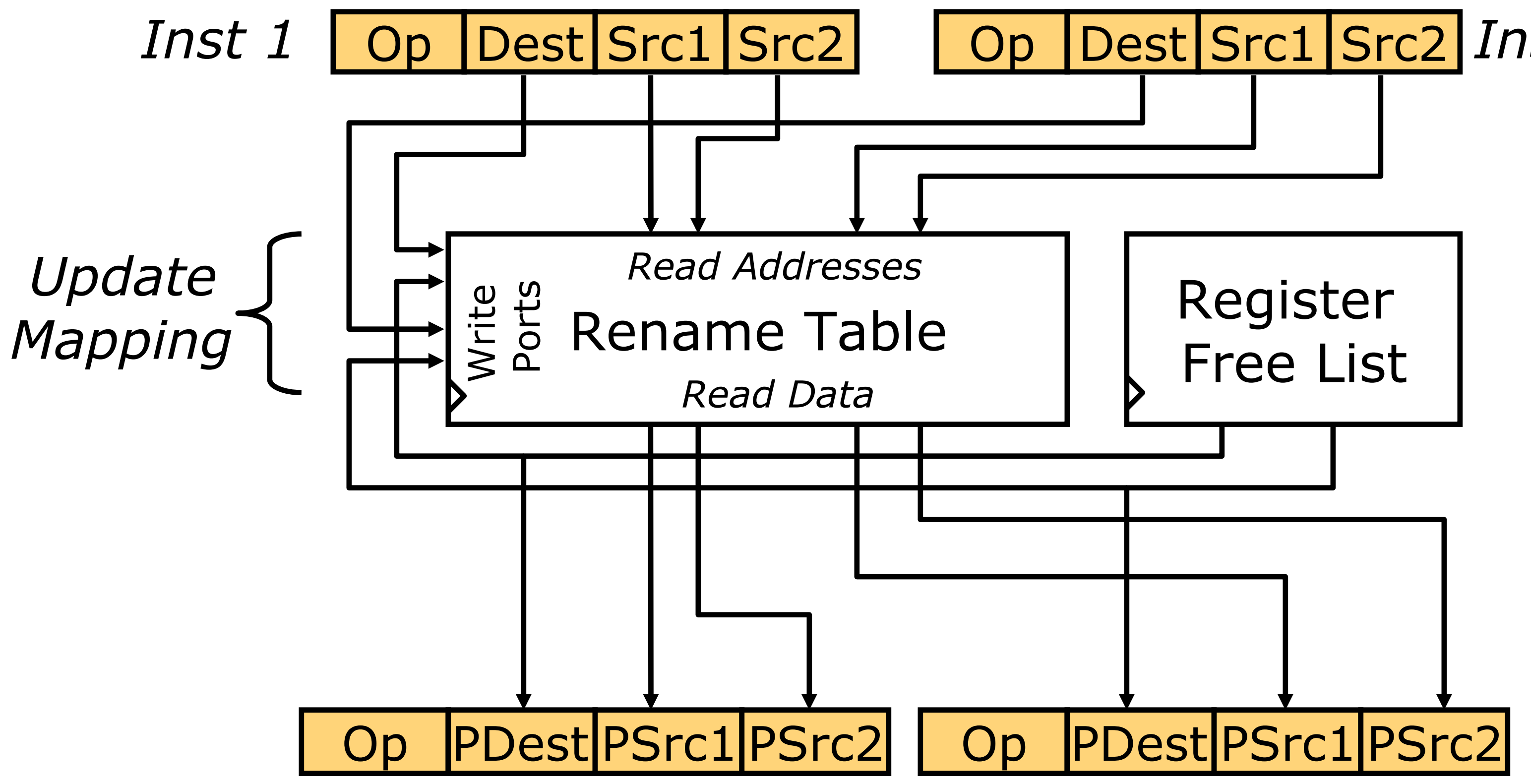
```
ld r1, (r3)
add r3, r1, r2
sub r6, r7, r9
add r3, r3, r6
ld r6, (r1)
add r6, r6, r3
st r6, (r1)
ld r6, (r1)
```

…

Cycle *t + 1*

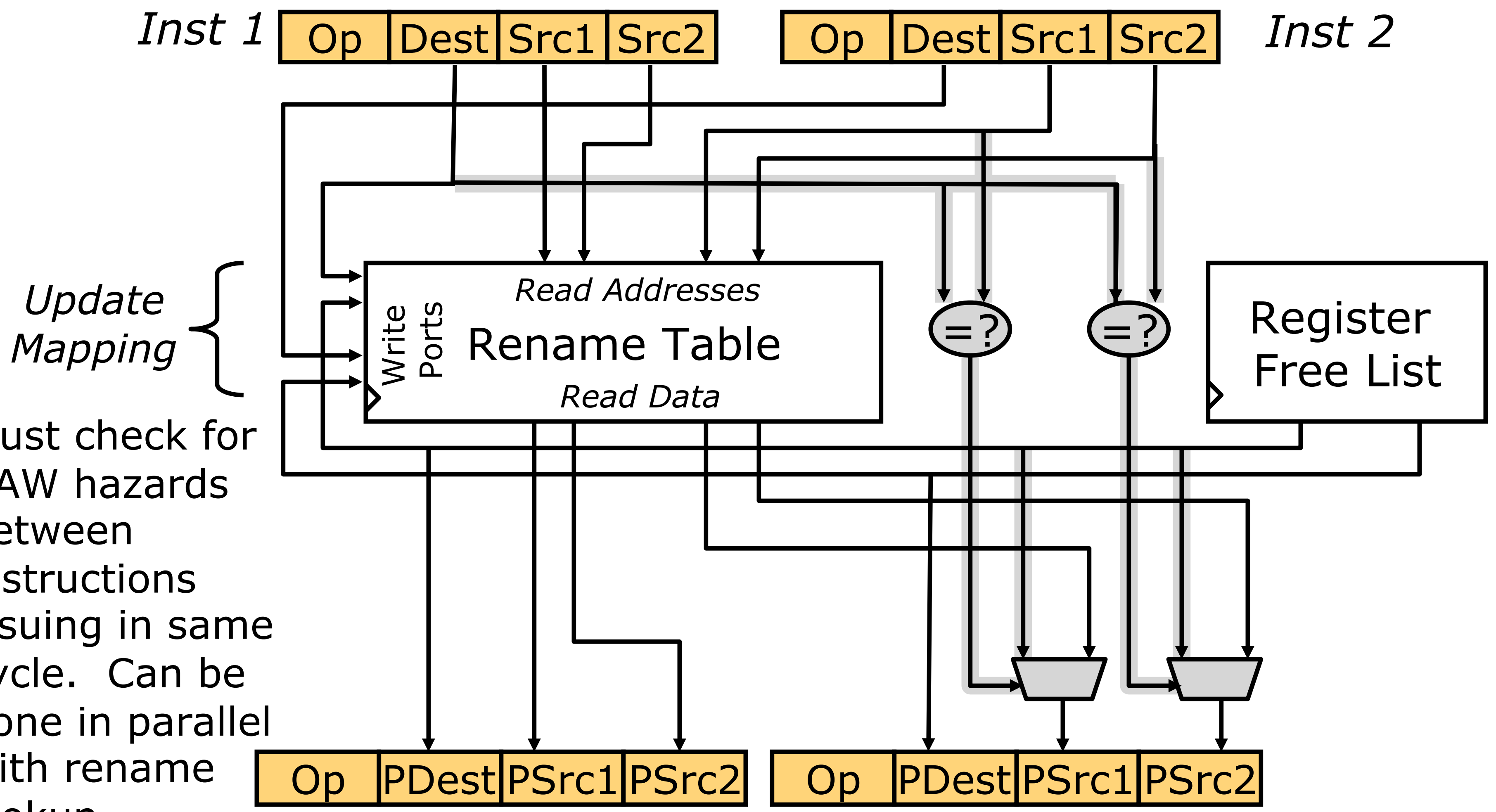
Superscalar Register Renaming

- 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



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

MIPS R10K renames 4 serially-RAW-dependent insts/cycle

Memory Dependencies

```
st r1, (r2)
ld r3, (r4)
```

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

```
st r1, (r2)
ld r3, (r4)
```

- Split execution of store instruction into two phases: address calculation and data write
- Can execute load before store, if addresses known and r4 != r2
- Each load address compared with addresses of all previous uncommitted stores *(can use partial conservative check i.e., bottom 12 bits of address)*
- Don’t execute load if any previous store address not known

(MIPS R10K, 16 entry address queue)

Address Speculation

```
st r1, (r2)
ld r3, (r4)
```

- Guess that r4 != r2
- Execute load before store address known
- Need to hold all completed but uncommitted load/ store addresses in program order
- If subsequently find r4==r2, squash load and *all* following instructions

=> Large penalty for inaccurate address speculation

Memory Dependence Prediction
(Alpha 21264)

```
st r1, (r2)
ld r3, (r4)
```

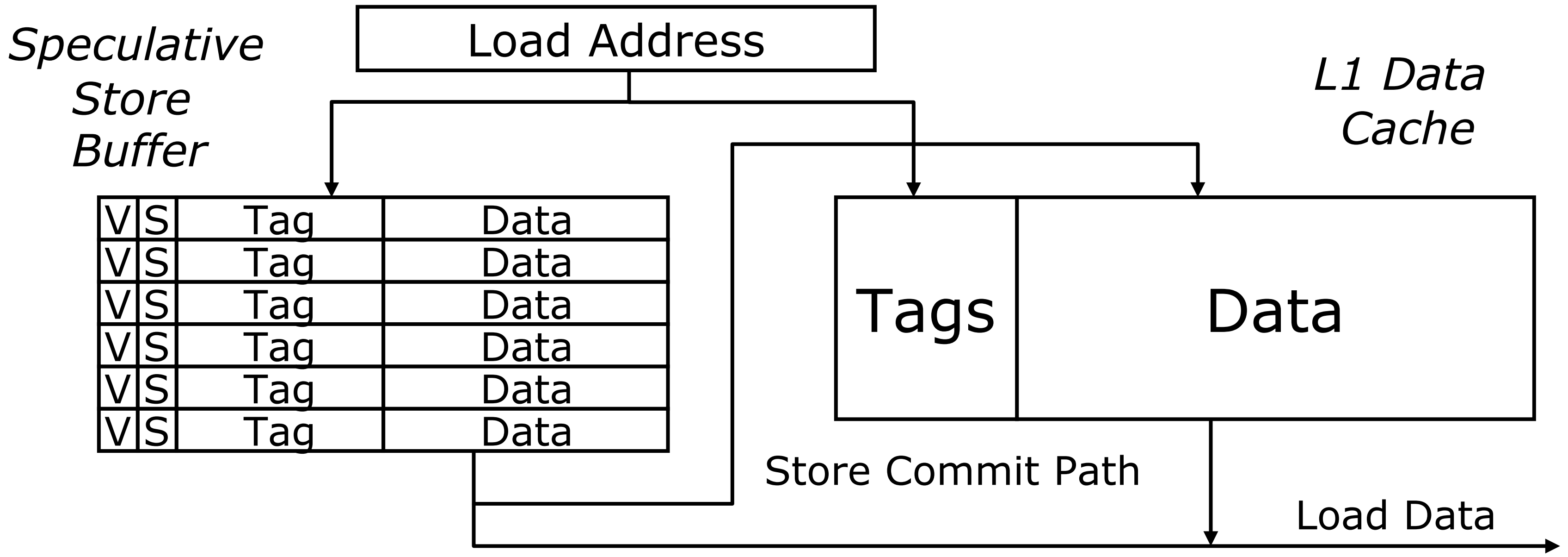
- Guess that r4 != r2 and execute load before store
- If later find r4==r2, squash load and all following instructions, but mark load instruction as *store-wait*
- Subsequent executions of the same load instruction will wait for all previous stores to complete
- Periodically clear *store-wait* bits

Speculative Loads / Stores

Just like register updates, stores should not modify the memory until after the instruction is committed

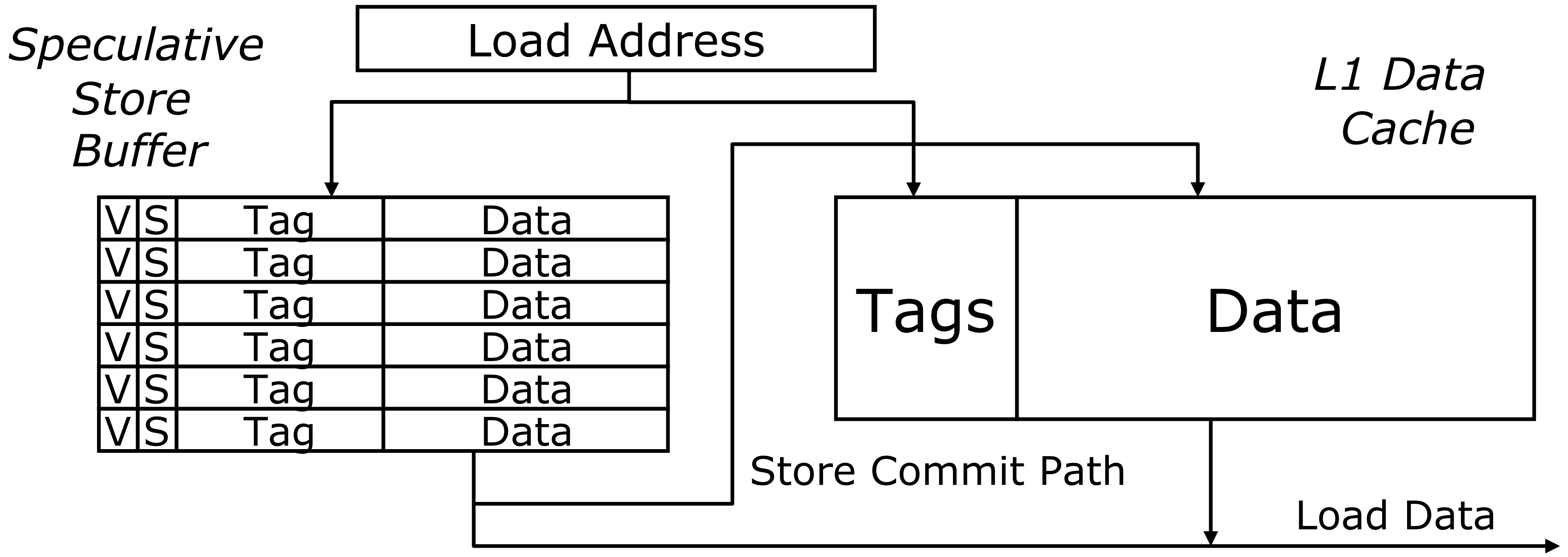
- A speculative store buffer is a structure introduced to hold speculative store data.

Speculative Store Buffer



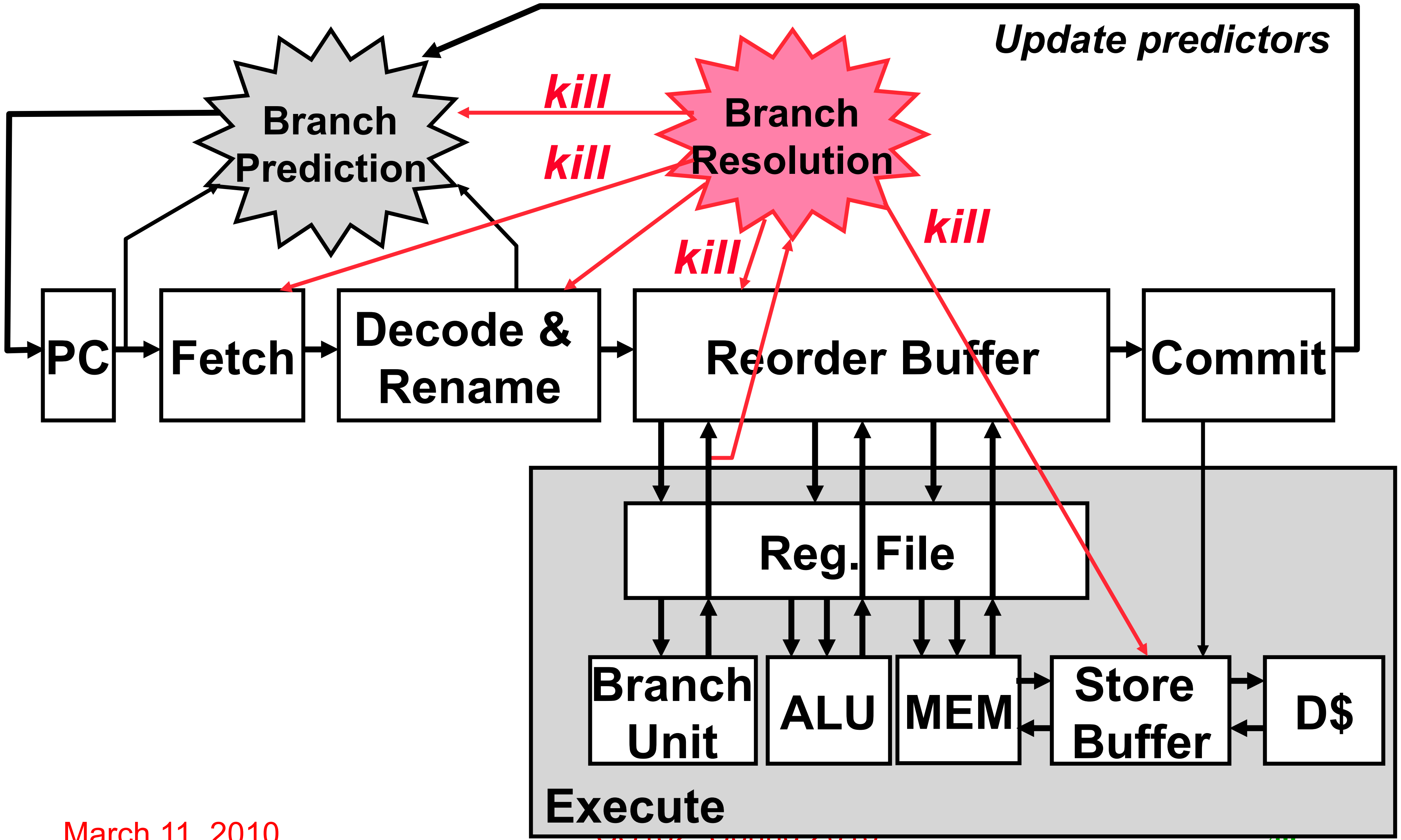
- On store execute:
  - mark entry valid and speculative, and save data and tag of instruction.
- On store commit:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit

Speculative Store Buffer



- 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

Datapath: Branch Prediction and Speculative Execution



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