ECE 552 / CPS 550 Advanced Computer Architecture I

Lecture 10 Instruction-Level Parallelism – Part 3

Benjamin Lee
Electrical and Computer Engineering
Duke University

www.duke.edu/~bcl15
www.duke.edu/~bcl15/class/class_ece252fall12.html

ECE552 Administrivia

27 September – Homework #2 Due

- Assignment on web page. Teams of 2-3.
- Submit soft copies to Sakai.
- Use Piazza for questions

2 October – Class Discussion

Roughly one reading per class. Do not wait until the day before!

1. Srinivasan et al. "Optimizing pipelines for power and performance"
2. Mahlke et al. "A comparison of full and partial predicated execution support for ILP processors"
3. Palacharla et al. "Complexity-effective superscalar processors"
4. Yeh et al. "Two-level adaptive training branch prediction"

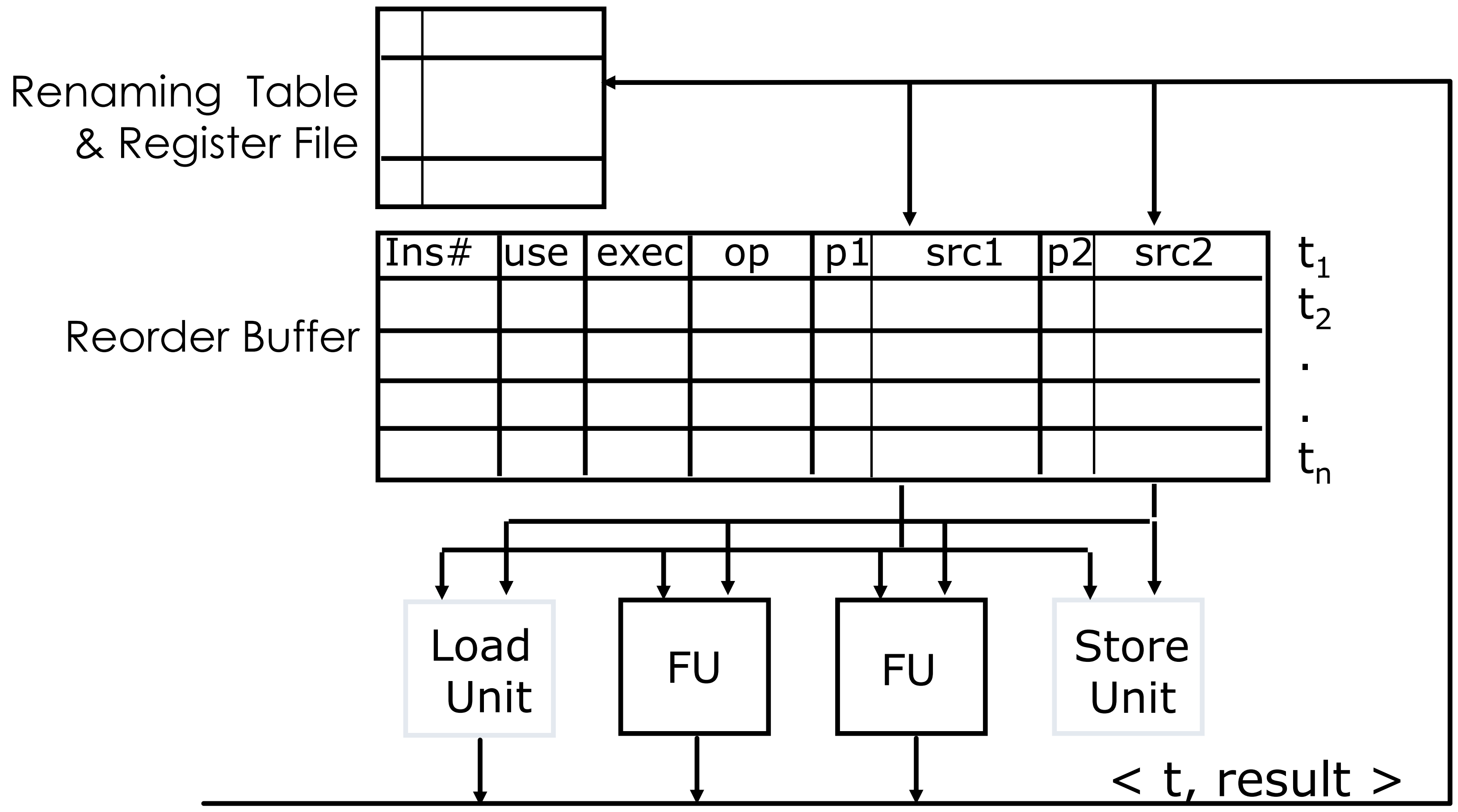
ECE552 Administrivia

4 October – Midterm Exam

- 75 minutes, in-class
- Closed book, closed notes exam

1. **Performance metrics** – performance, power, yield
2. **Technology** – trends that changed architectural design
3. **History** – Instruction sets (accumulator, stack, index, general-purpose)
4. **CISC** – microprogramming, writing microprogram fragments
5. **Pipelining** – Performance, hazards and ways to resolve them
6. **Instruction-level Parallelism** – mechanisms to dynamically detect data dependences and to manage instruction flow (Scoreboard, Tomasulo, Physical Register File)
7. **Speculative Execution** – exception handling, branch prediction
8. **Readings** – High-level questions, not details

Tomasulo's Implementation



- Decode stage allocates instruction template (i.e., tag t) and stores tag in register file.
- When instruction completes, tag is de-allocated.

Tomasulo’s Structures

Reorder Buffer (ROB)	-- buffers in-flight instructions in program order
	-- supports in-order commit, precise exceptions
	-- e.g., instruction#, use, exec, op
Reservation Stations	-- tracks renamed source operands
	-- if operands ready, contains value (e.g., v1)
	-- if operands pending, contains tag (e.g., t1)
	-- may be combined with ROB (e.g., our example)
	-- may be distributed across functional units
Renaming Table	-- if write committed, points to register file (e.g., F1)
	-- if write pending, points to ROB entry (e.g., t1)
	-- Rename registers (e.g., F1) with ROB tags (e.g., t1)
Register File	-- contains architected state, committed values
Common Data Bus (CDB)	-- functional units broadcast computed values
	-- broadcast includes <tag, result>

Tomasulo's Pipeline

1. Fetch	
2. Dispatch	-- Decode instruction
	-- Stall if structural hazard in ROB
	-- Allocate ROB entry and rename using ROB tags
	-- Read source operands when they are ready
3. Execute	-- Issue instruction when all operands ready
	-- Instructions may issue out-of-order
4. Complete	-- Stall if structural hazard on the common data bus
	-- Broadcast tag and completed result
	-- Mark ROB entry as complete
	-- Instructions may complete out-of-order
5. Retire	-- Stall if oldest instruction (head of ROB) not complete
	-- Handle any interrupts
	-- Write-back value for oldest instruction to register file or mem
	-- Free ROB entry
	-- Instructions retire in-order

Physical Register File

Tomasulo Performance Limitations

-- Too much data movement on common data bus

-- Multi-input multiplexors, long buses impact clock frequency

Alternative Approach to Register Renaming

-- Eliminate architectural register file (e.g., R0-R31, F1-F8)

-- Add larger physical register file, which holds all values (e.g., P0-Pn, n>>32)

-- Modify rename table to map architected registers to physical registers

-- Add free list to manage unallocated physical registers

-- Reorder buffer tracks ready operands, supports in-order retire, supports free list management (but does not rename)

Lifetime of Physical Registers

-- Architected registers are those defined by the instruction set architecture

-- Register renaming can be implemented in two ways

-- Rename with buffer tags -- insert speculatively computed values into ROB

-- Rename with physical registers – hold both committed and speculative values

With Architected Registers		**With Physical Registers**	
1. ld	R1, (R3)	ld	P1, (Px)
2. add	**R3**, R1, 4	add	**P2**, P1, 4
3. sub	R6, R7, R9	sub	P3, Py, Pz
4. add	**R3**, R3, R6	add	**P4**, P2, P3
5. ld	R6, (R1)	ld	P5, (P1)
6. add	R6, R6, R3	add	P6, P5, P4
7. st	R6, (R1)	st	P6, (P1)
8. ld	R6, (R11)	ld	P7, (Pw)

-- Every instruction's destination register R renamed to physical register P

-- When do we reuse physical register? When next write of same architected register commits. Example: Reuse **P2** when instruction 4 commits

Physical Register Management

Rename Table

-- Maps architected registers (e.g., R*) to physical registers (e.g., P*)

-- Rename table identifies physical register P* that contains the value of R*

-- Example: MIPS has 32 architected registers so rename table has 32 entries.

-- Microarchitecture might have N >> 32 physical registers.

Physical Registers

-- Contain N>>32 registers that can hold committed data.

-- Committed data is present when "p" flag is set.

Free List

-- List of physical registers available for renaming.

-- Stall pipeline if there are insufficient physical registers

Reorder Buffer

-- Issue logic checks buffer to determine if operand values present

-- Tracks sequence of renamings for the same architected register

Physical Register Management

-- After the fetch stage, instruction enters decode stage.

-- Decode stage (1) extracts architected registers, (2) renames to physical registers, and (3) inserts instruction into reorder buffer.

Every instruction's destination register is renamed! Eliminates WAW/WAR hazards.

Renaming for instruction "op Rd, R1, R2" requires the following steps:

i. Lookup source registers (R1,R2) in rename table.
   Insert corresponding physical register (P<y>,P<z>) into ROB.

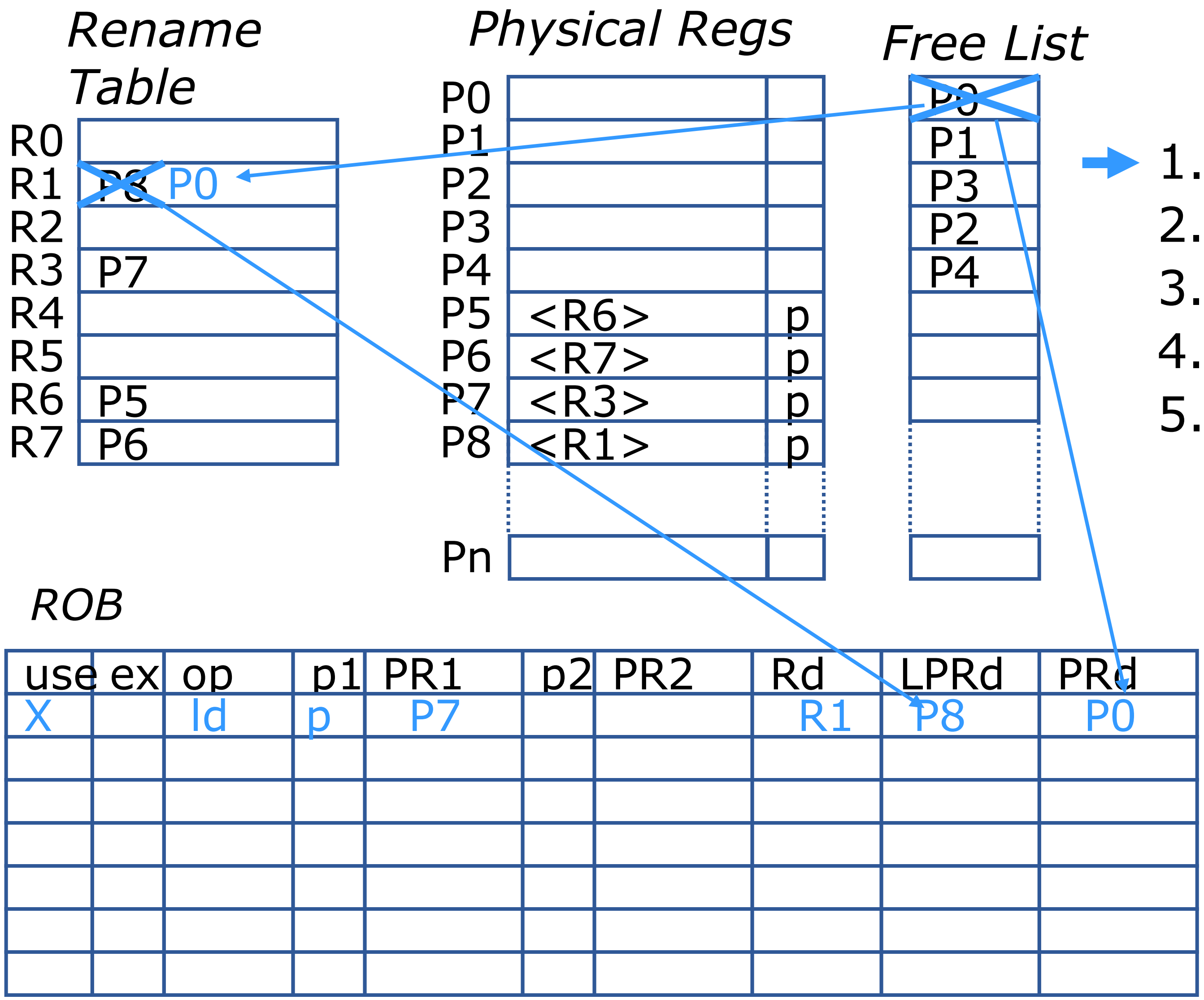
ii. If values for P<y>, P<z> are present, set "p" flag in ROB.

iii. Lookup destination register (Rd) in rename table.
   Suppose Rd already renamed to P<w>.
   Because we rename Rd in step iii, this is the last instruction for which P<w> is valid.
   Denote P<w> as last physical register (LPRd) in ROB. Required for managing free list.

iv. Rename Rd to P<x>, which is next available register from free list.
   Denote as current physical register (PRd) in ROB.

Issue logic sends instruction to execution units when both source registers present

Renaming R1 to P0



Rename Table

R0	
R1	~~P8~~ P0
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

1. ld R1, 0(R3)
2. add R3, R1, 4
3. sub R6, R7, R6
4. add R3, R3, R6
5. ld R6, 0(R1)

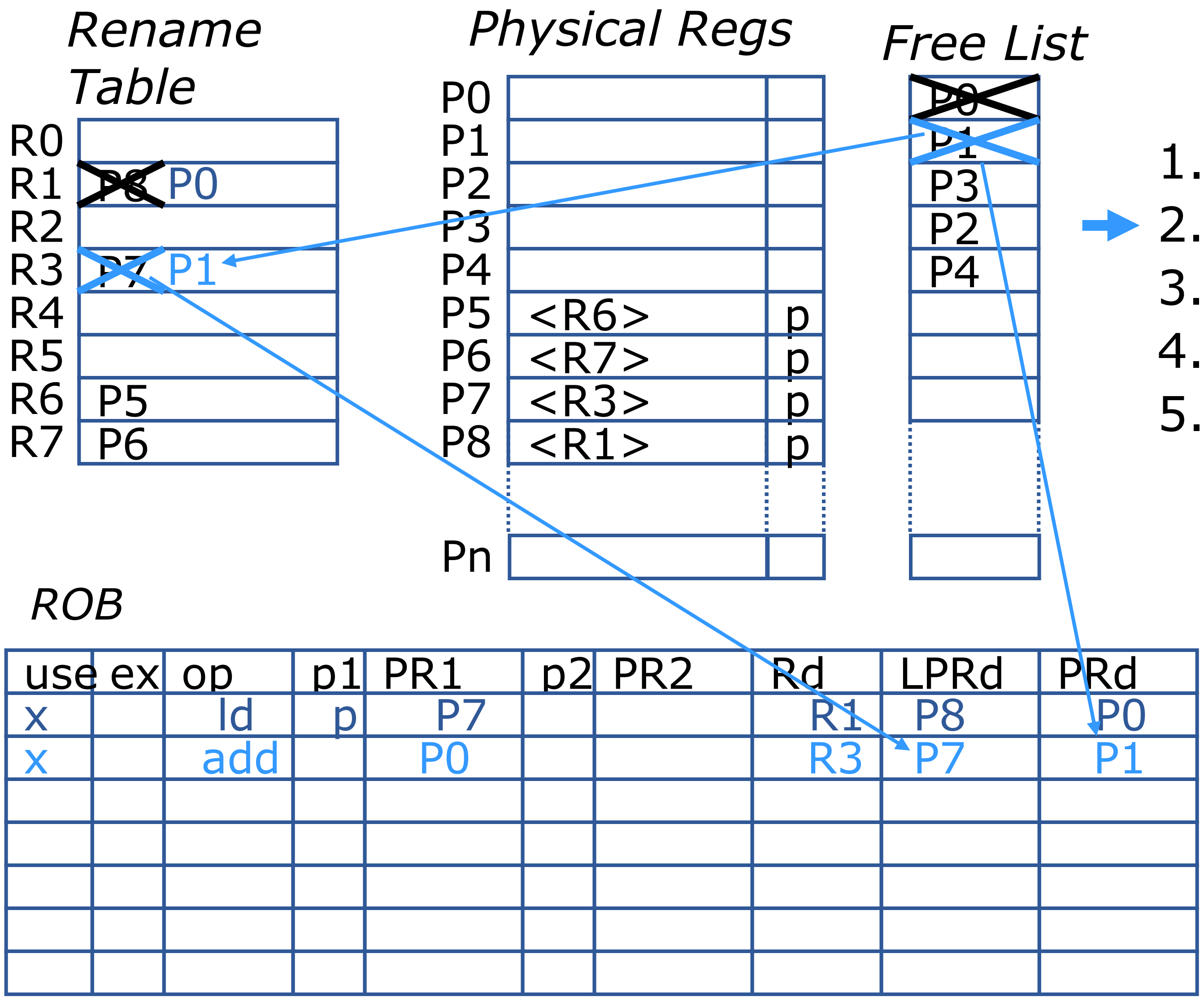
ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
X		ld	p	P7			R1	P8	P0

PR1/2: src physical regs
p1/2: set when physical reg values are present

Rd: dest architected reg
LPRd: last dest physical reg
PRd: new dest physical reg

Renaming R3 to P1



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

1. ld R1, 0(R3)
2. add R3, R1, 4
3. sub R6, R7, R6
4. add R3, R3, R6
5. ld R6, 0(R1)

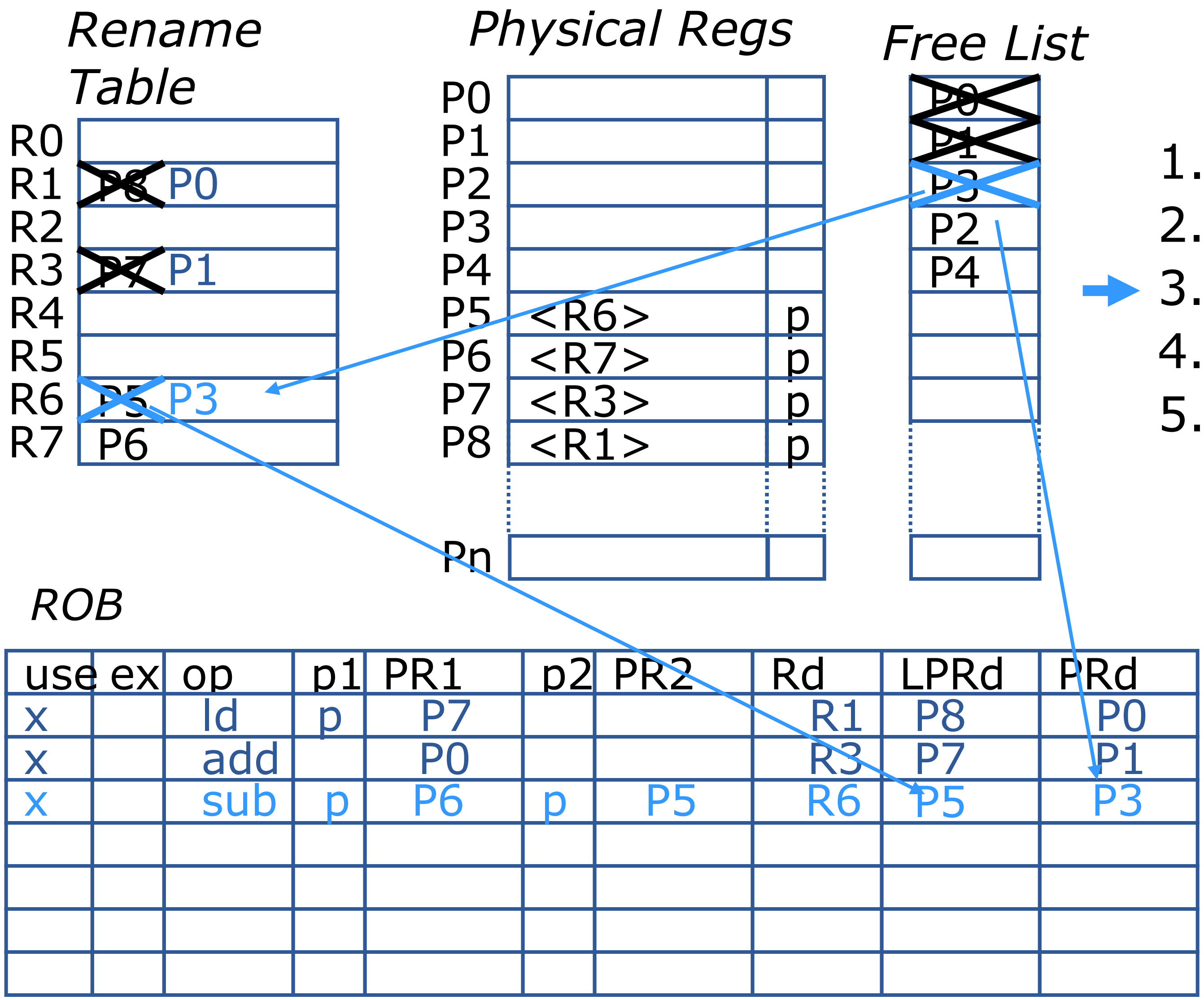
ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			R1	P8	P0
x		add		P0			R3	P7	P1

PR1/2: src physical regs
p1/2: set when physical reg values are present

Rd: dest architected reg
LPRd: last dest physical reg
PRd: new dest physical reg

Renaming R6 to P3



Rename Table

R0	
R1	~~P8~~ P0
R2	
R3	~~P7~~ P1
R4	
R5	
R6	~~P5~~ P3
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

1. ld R1, 0(R3)
2. add R3, R1, 4
3. sub R6, R7, R6
4. add R3, R3, R6
5. ld R6, 0(R1)

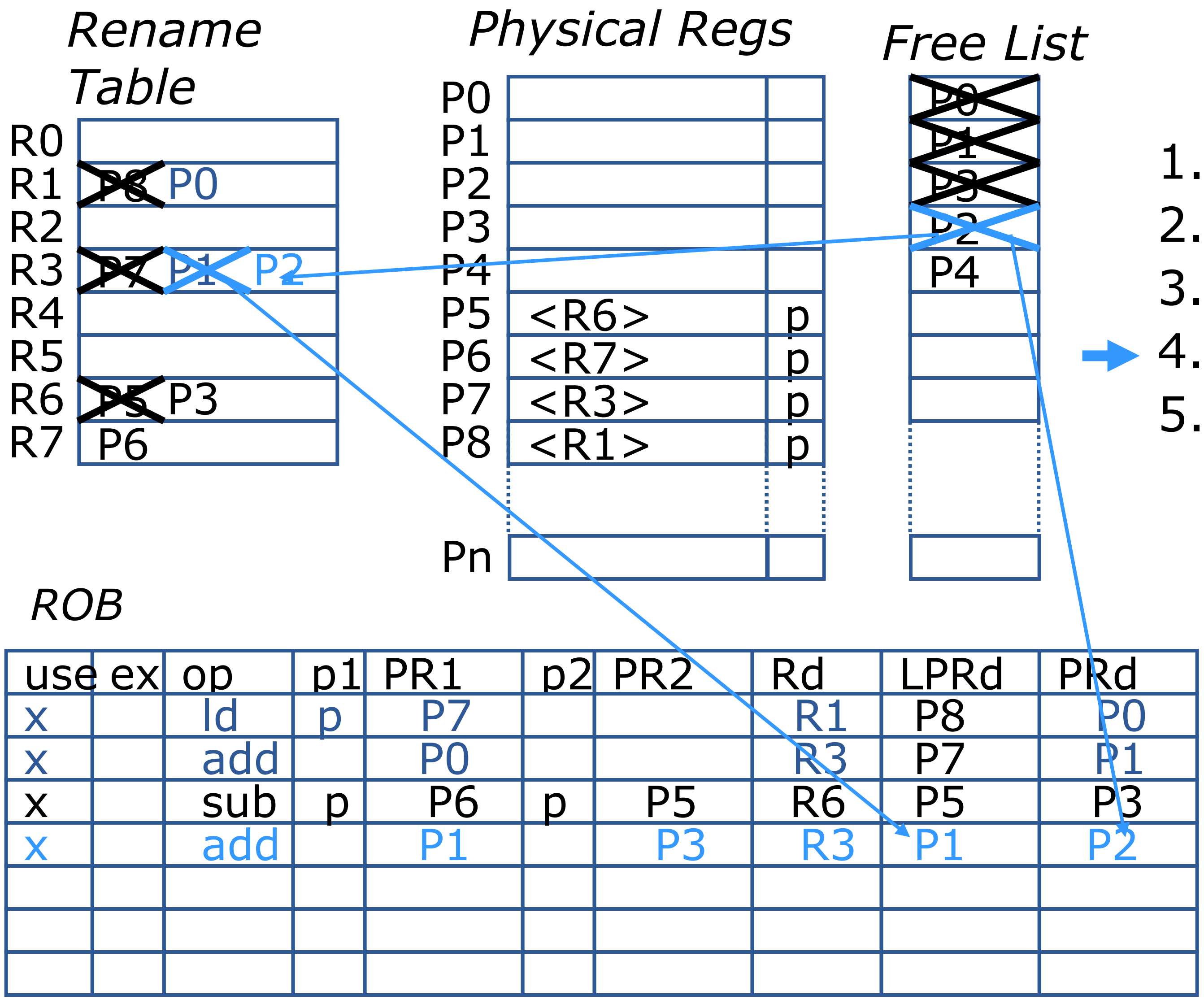
ROB

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

PR1/2: src physical regs
p1/2: set when physical reg values are present

Rd: dest architected reg
LPRd: last dest physical reg
PRd: new dest physical reg

Renaming R3 to P2



Rename Table

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

1. ld R1, 0(R3)
2. add R3, R1, 4
3. sub R6, R7, R6
4. add R3, R3, R6
5. ld R6, 0(R1)

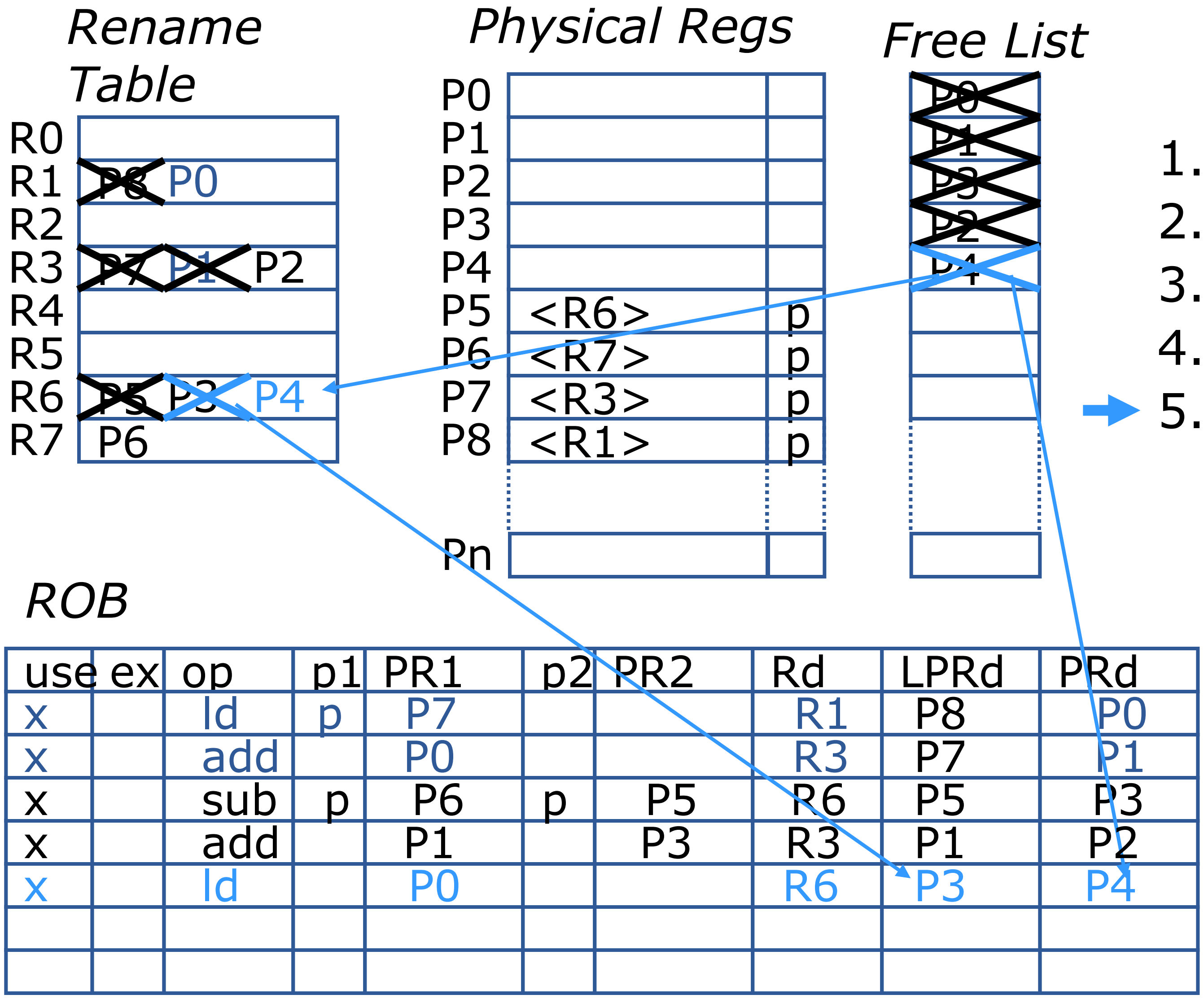
ROB

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

PR1/2: src physical regs
p1/2: set when physical reg values are present

Rd: dest architected reg
LPRd: last dest physical reg
PRd: new dest physical reg

Renaming R6 to P4



Rename Table

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

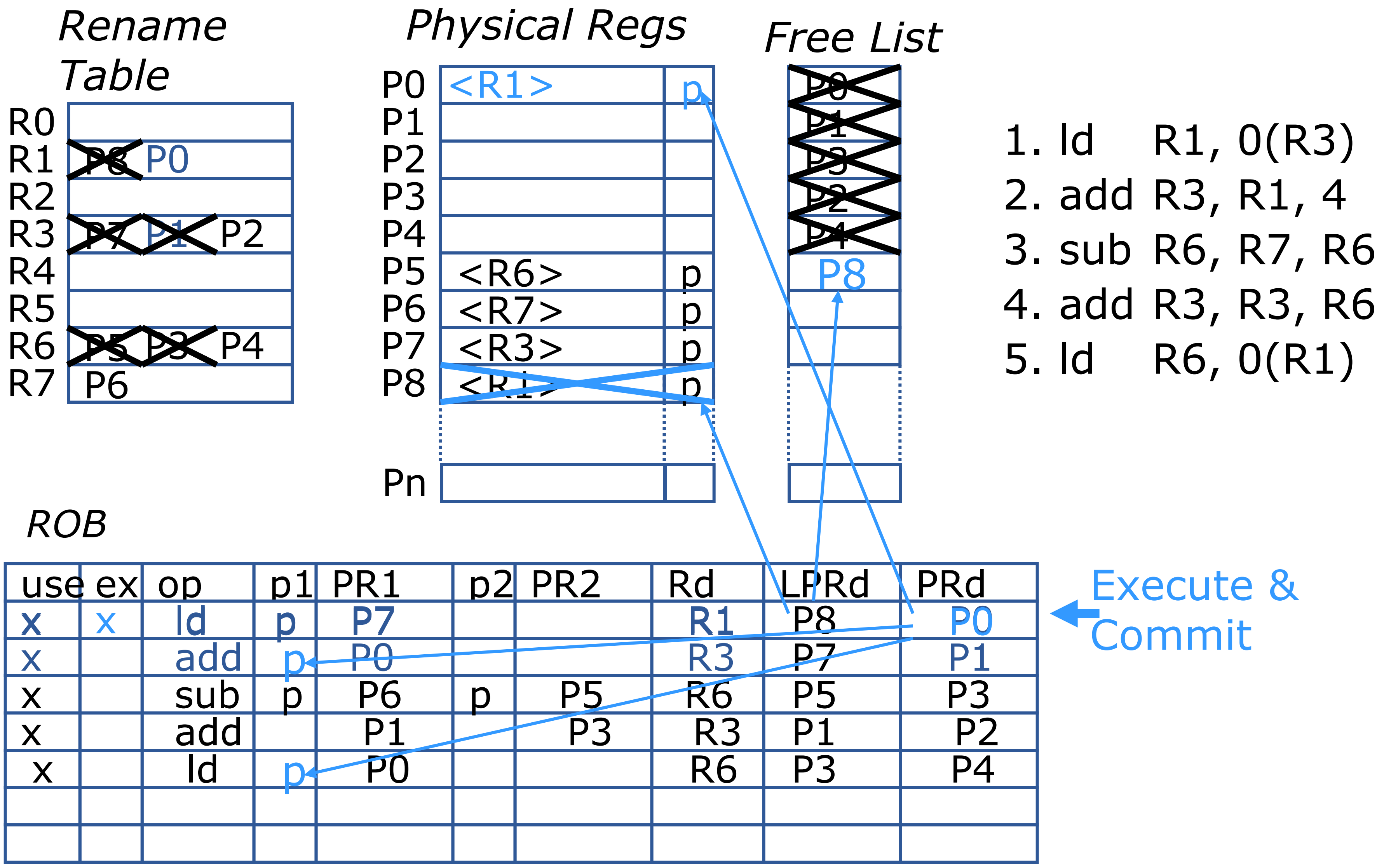
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			R1	P8	P0
x		add		P0			R3	P7	P1
x		sub	p	P6	p	P5	R6	P5	P3
x		add		P1		P3	R3	P1	P2
x		ld		P0			R6	P3	P4

1. ld R1, 0(R3)
2. add R3, R1, 4
3. sub R6, R7, R6
4. add R3, R3, R6
5. ld R6, 0(R1) ←

PR1/2: src physical regs
p1/2: set when physical reg values are present

Rd: dest architected reg
LPRd: last dest physical reg
PRd: new dest physical reg

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

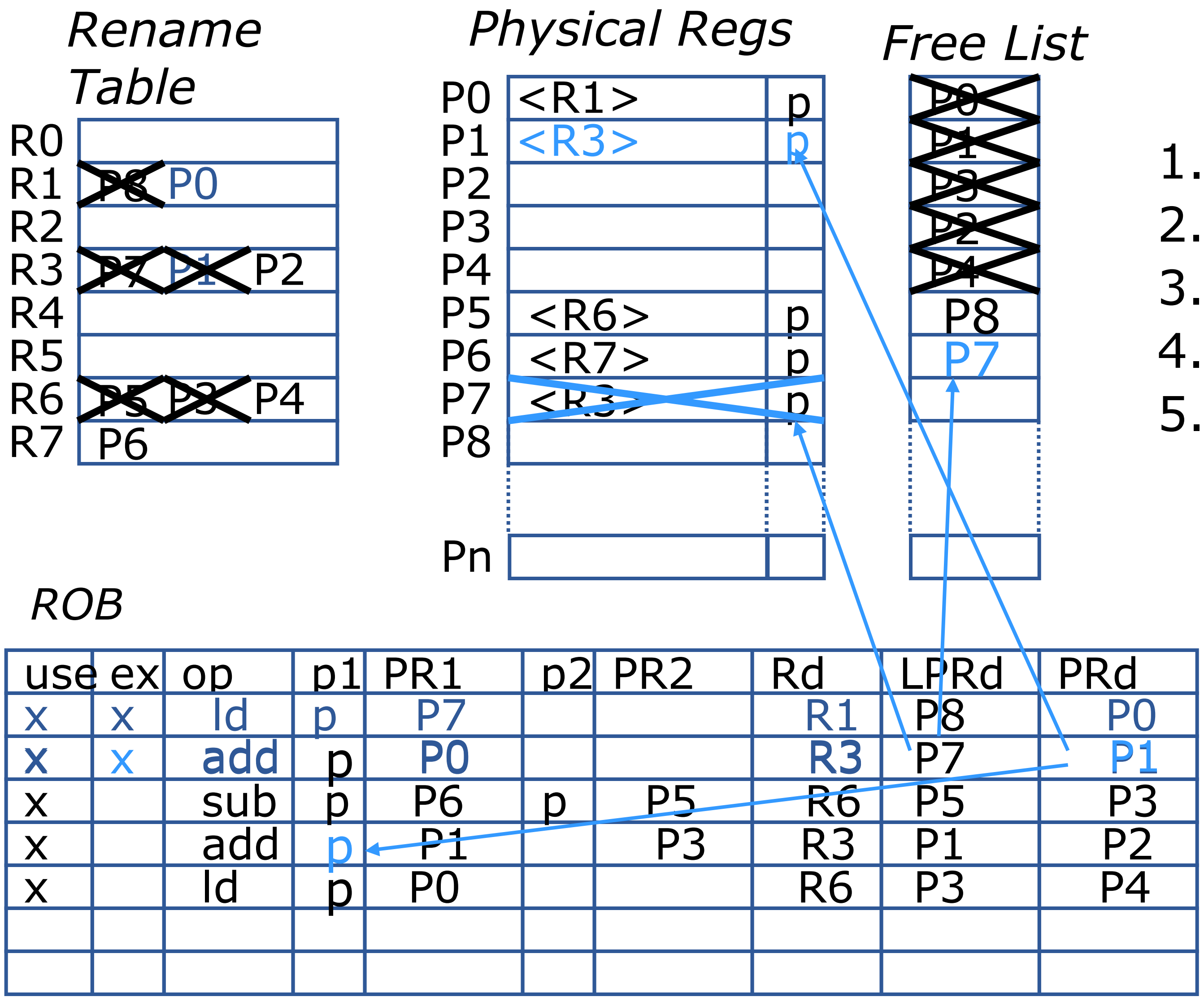
1. ld R1, 0(R3)
2. add R3, R1, 4
3. sub R6, R7, R6
4. add R3, R3, R6
5. ld R6, 0(R1)

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

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

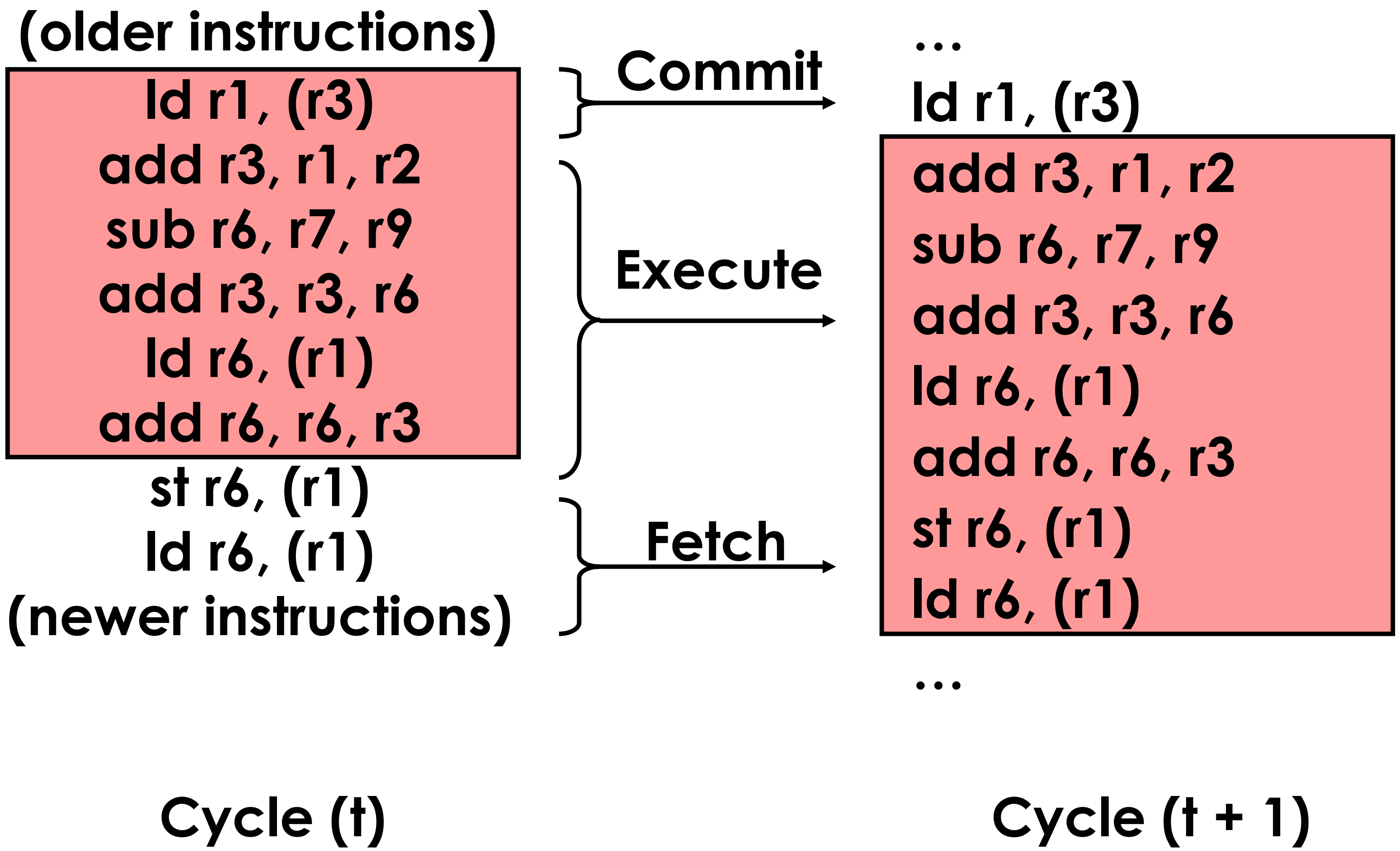
ROB

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

Execute & Commit

1. ld R1, 0(R3)
2. add R3, R1, 4
3. sub R6, R7, R6
4. add R3, R3, R6
5. ld R6, 0(R1)

Active Instruction Window in ROB

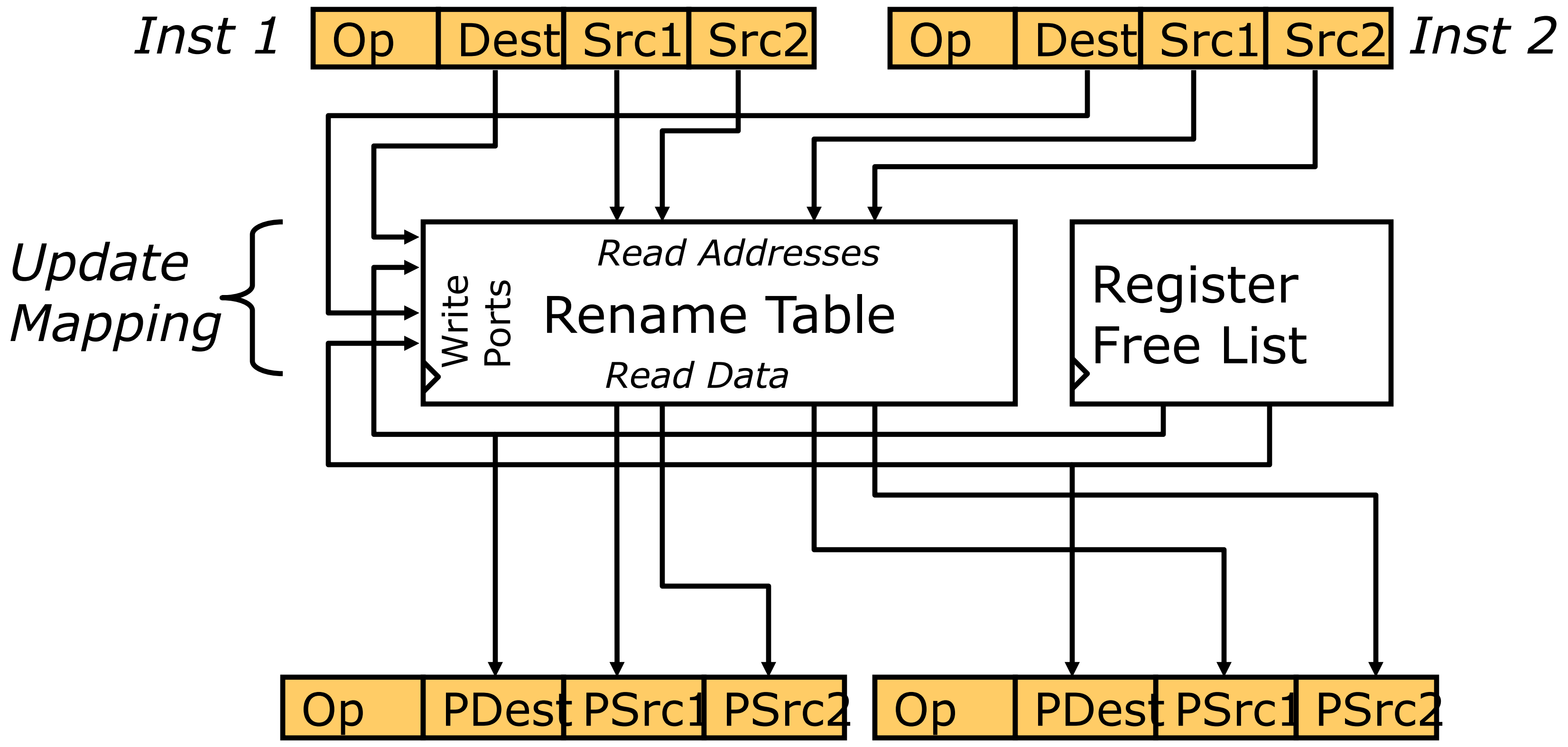


```
(older instructions)
ld r1, (r3)
add r3, r1, r2
sub r6, r7, r9
add r3, r3, r6
ld r6, (r1)
add r6, r6, r3
st r6, (r1)
ld r6, (r1)
(newer instructions)
```

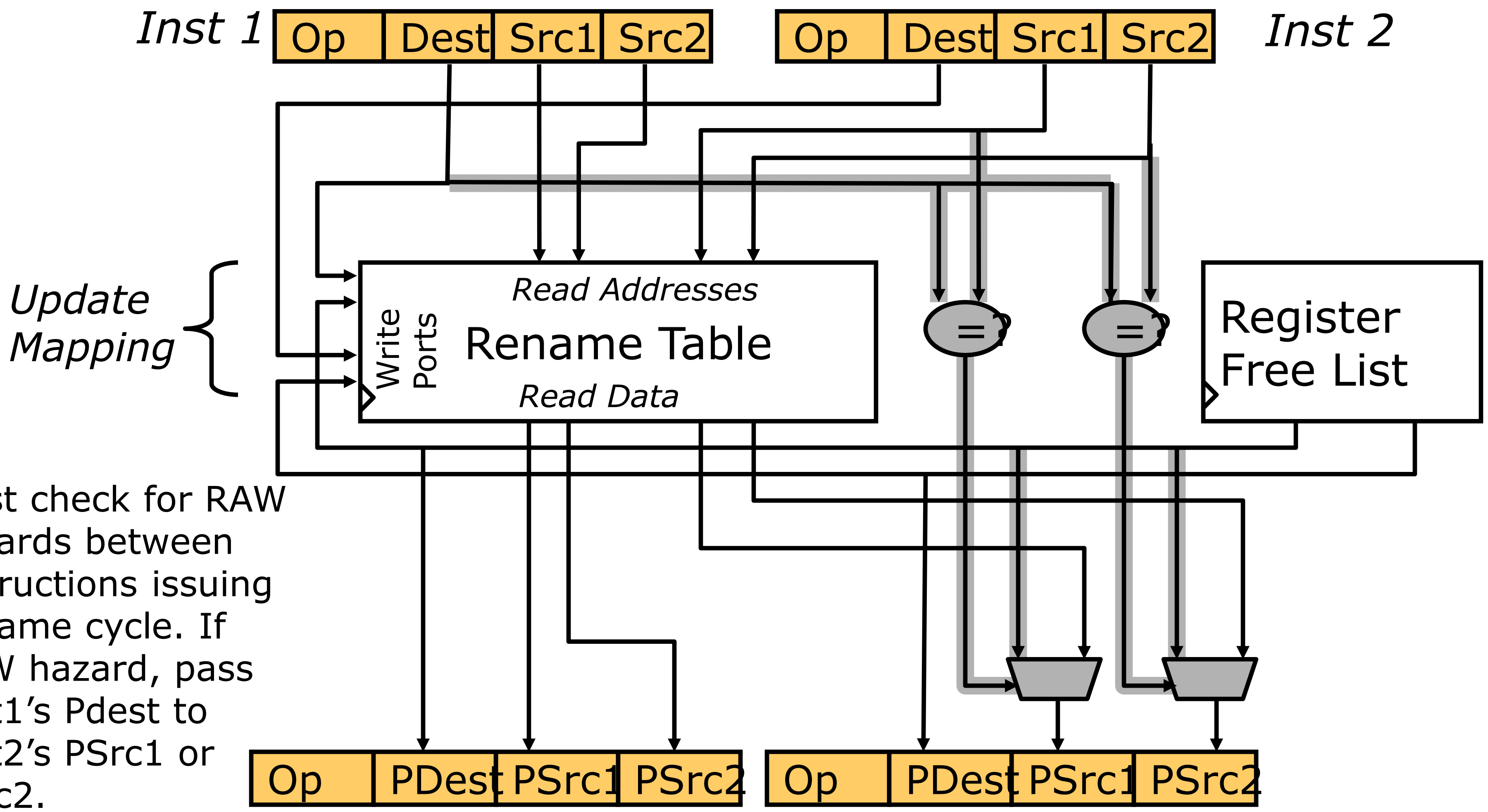
```
…
ld r1, (r3)
add r3, r1, r2
sub r6, r7, r9
add r3, r3, r6
ld r6, (r1)
add r6, r6, r3
st r6, (r1)
ld r6, (r1)
…
```

Superscalar Register Renaming

-- During decode, instruction is allocated new physical register for dest
-- Instruction's source registers 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. If RAW hazard, pass Inst1's Pdest to Inst2's PSrc1 or PSrc2.

Memory Dependencies

```
st    r1, j(r2)
ld    r3, k(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 and commit architected state until all previous loads and stores have completed execution

Can still execute loads speculatively and out-of-order with respect to other instructions.

Out-of-order Loads

Conservative out-of-order load execution

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

-- Split execution of store instruction into two phases
-- Address calculation and data write

-- Can execute load before store if addresses known and j(r2) != k(r4)

-- Each load address compared with addresses of previous uncommitted stores

-- Don't execute load if any previous store address not known

Speculative Loads

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

-- Guess that j(r4) != k(r2)

-- Execute load before store address is known

-- Need to hold all completed but uncommitted load/store addresses in program order

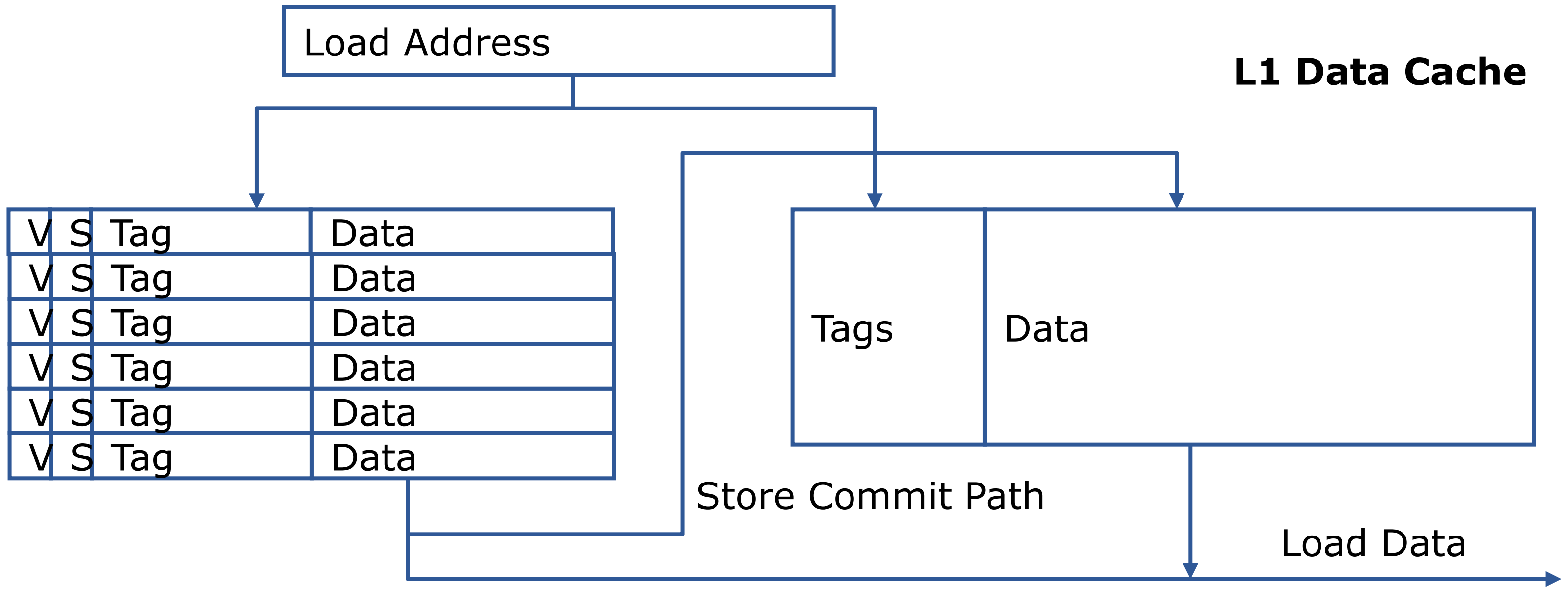
-- Later, if we find r4 == r2, squash load and all following instructions
-- Large penalty for inaccurate address speculation

Speculative 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

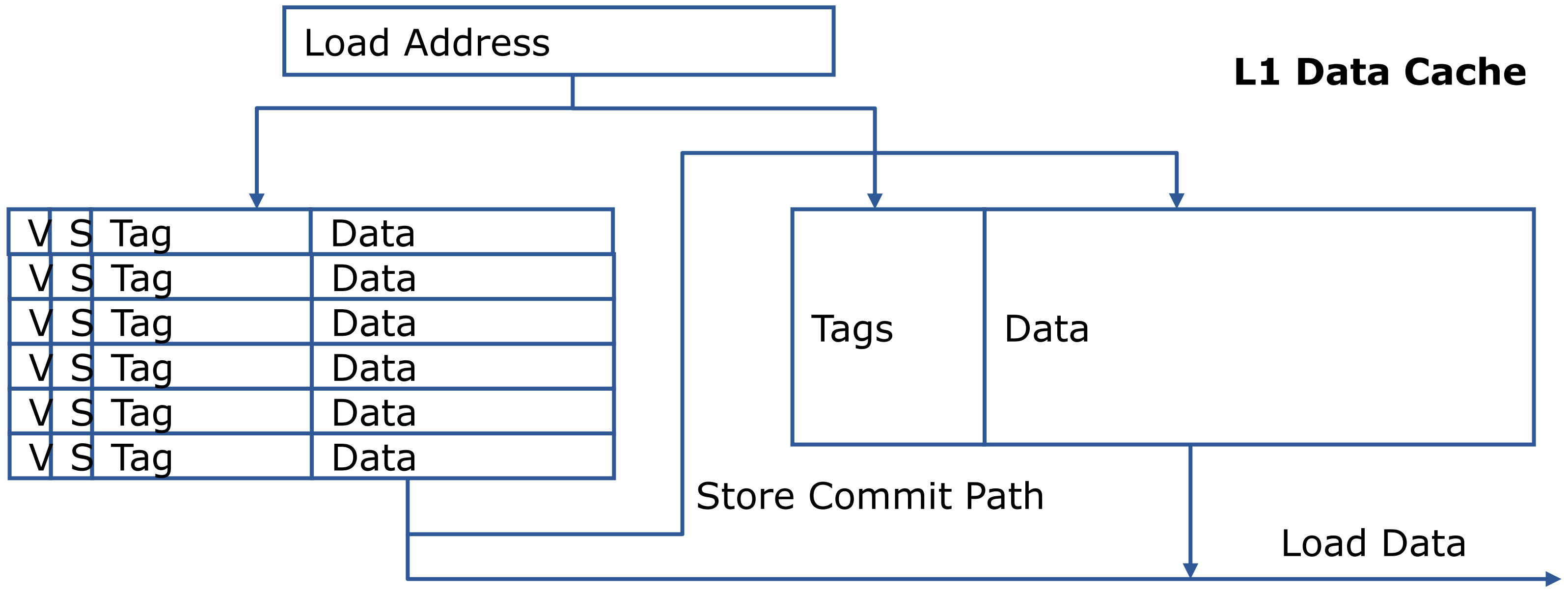


-- On store execute: mark entry valid (V) and speculative (S), save data and tag of instruction

-- On store commit: clear speculative bit and eventually move data to cache

-- On store abort: clear valid bit

Speculative Store Buffer



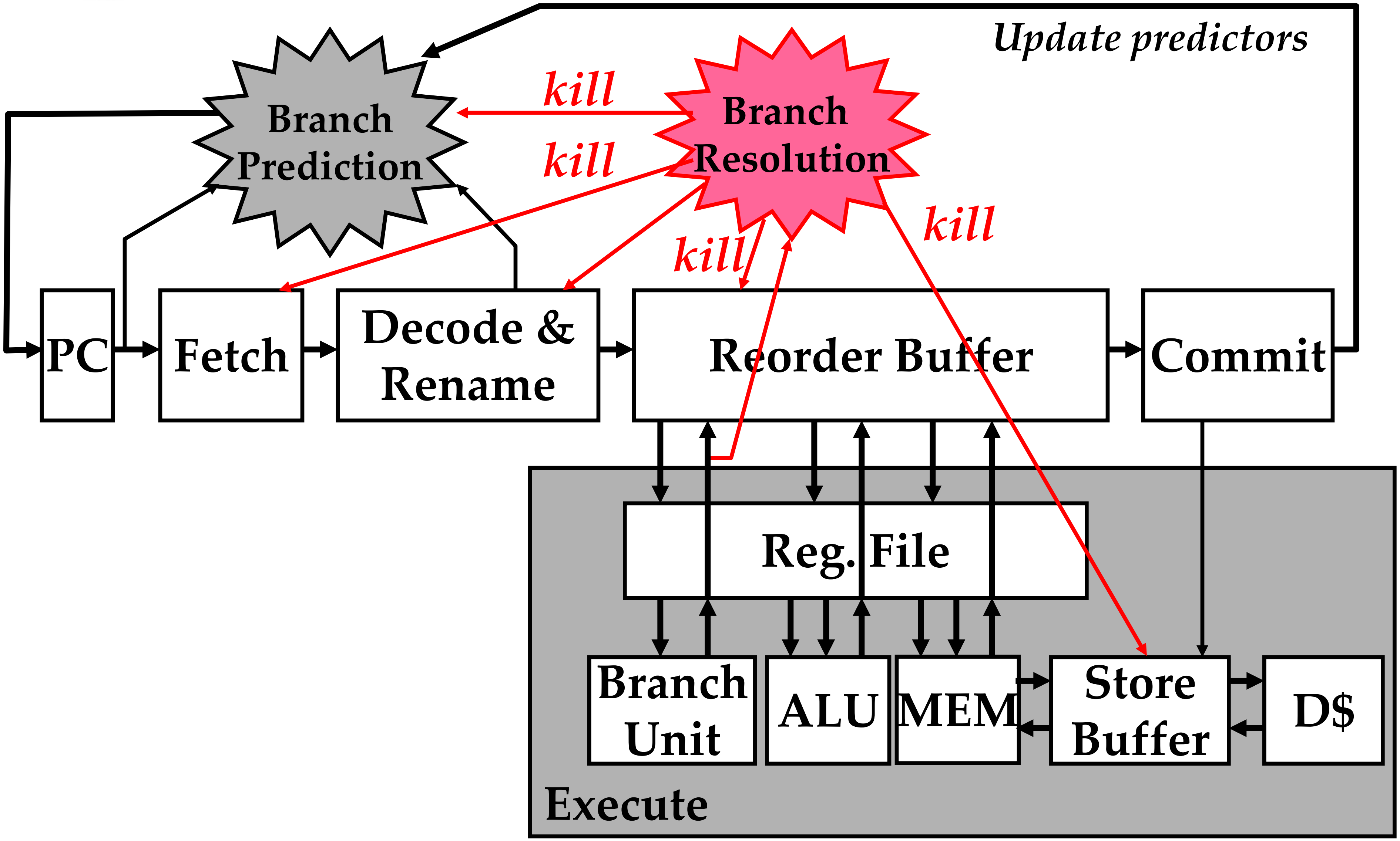
-- If data in both store buffer and cache, which should we use?

-- Speculative store buffer

-- If same address in store buffer twice, which should we use?

-- Youngest store that is older than load

Speculative Datapath



Branch Prediction

Motivation

-- Branch penalties limit performance of deeply pipelined processors

-- Modern branch predictors have high accuracy (>95%) and can significantly reduce branch penalties

Hardware Support

-- Prediction structures: branch history tables, branch target buffer, etc.

-- Mispredict recovery mechanisms:

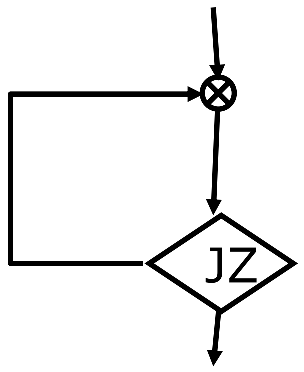
-- Separate instruction execution and instruction commit

-- Kill instructions following branch in pipeline

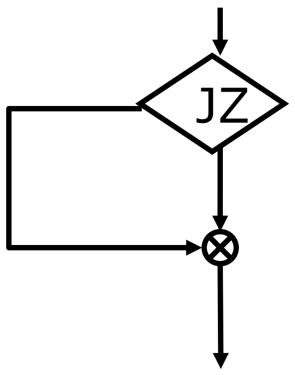
-- Restore architectural state to correct path of execution

Static Branch Prediction

backward
90%



forward
50%



On average, probability a branch is taken is 60-70%.

But branch direction is a good predictor.

ISA can attach preferred direction semantics to branches (e.g., Motorola MC8810, bne0 prefers taken, beq0 prefers not taken).

ISA can allow choice of statically predicted direction (e.g., Intel IA-64). Can be 80% accurate.

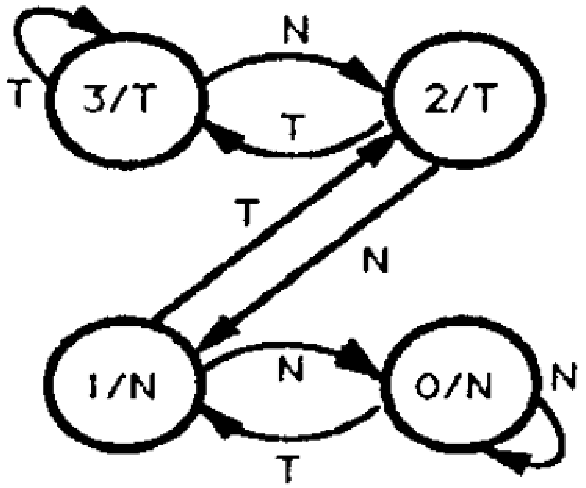
Dynamic Branch Prediction

Learn from past behavior

Temporal Correlation -- The way a branch resolves may be a good predictor of the way it will resolve at the next execution

Spatial Correlation -- Several branches may resolve in a highly correlated manner (preferred path of execution in the application)

2-bit Branch Predictor

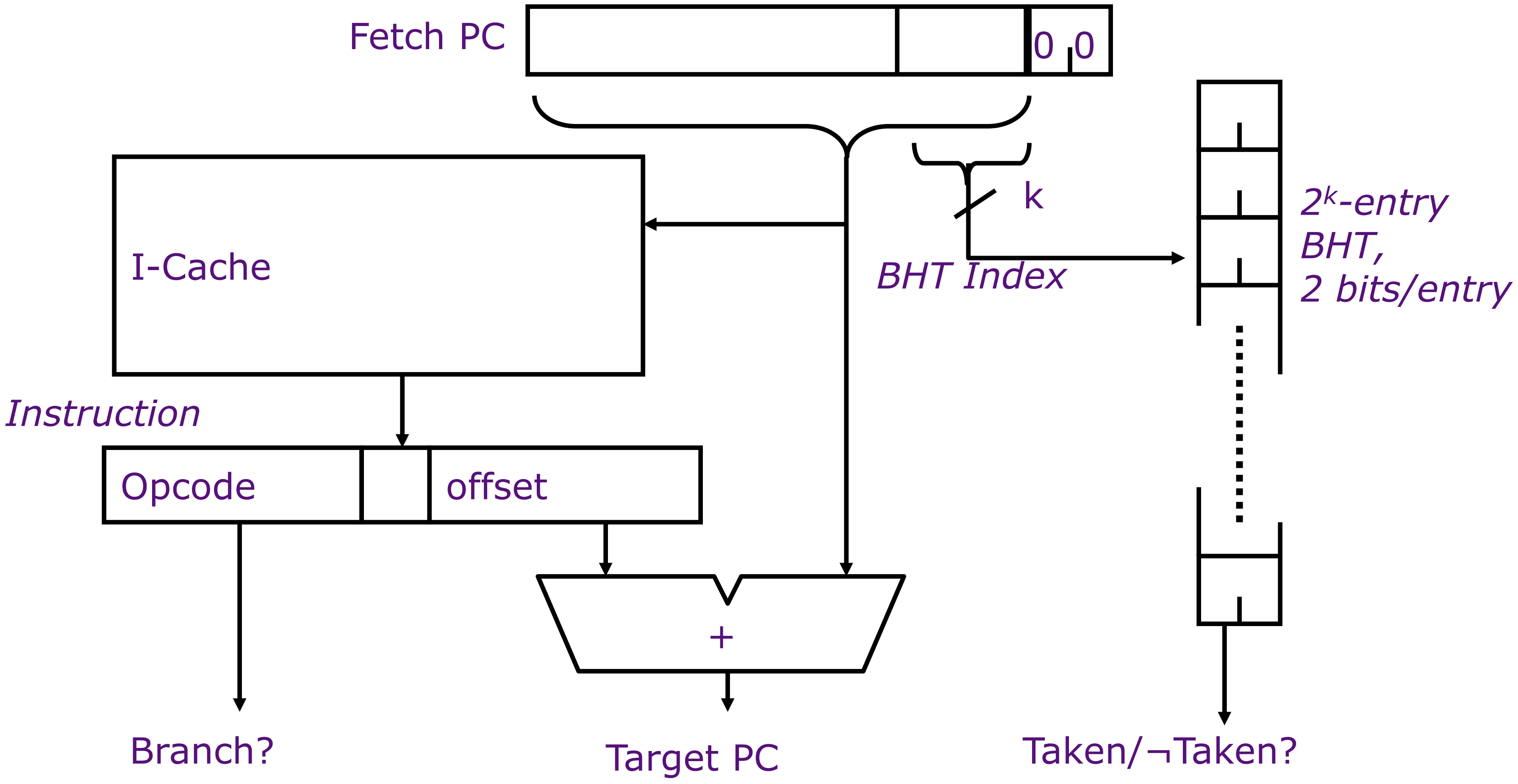


Automaton A2
(2-bit Saturating Up-down Counter)

Use two-bit saturating counter.

Changes prediction after two consecutive mistakes.

Branch History Table (BHT)

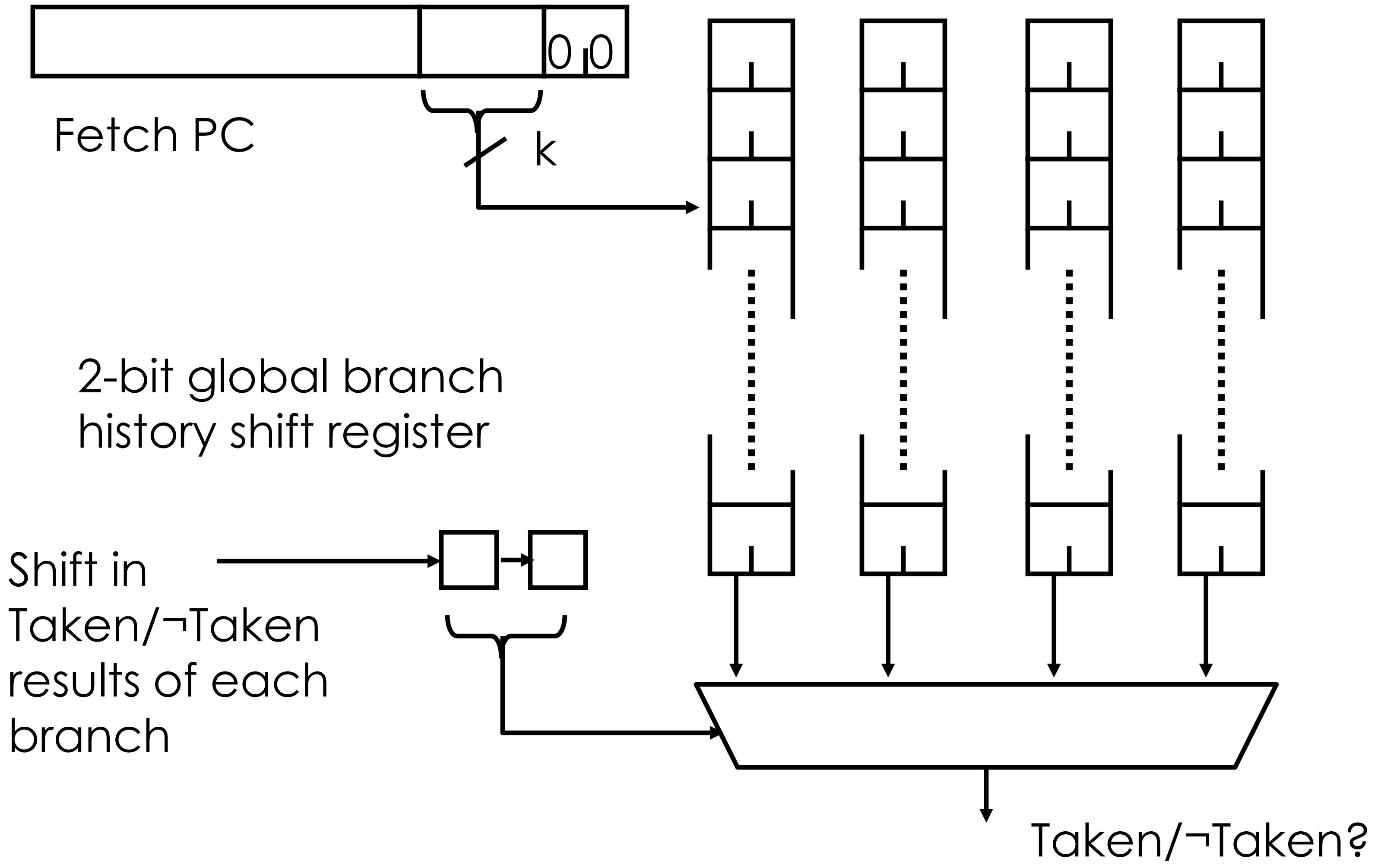


BHT is an array of 2-bit branch predictors, indexed by branch PC

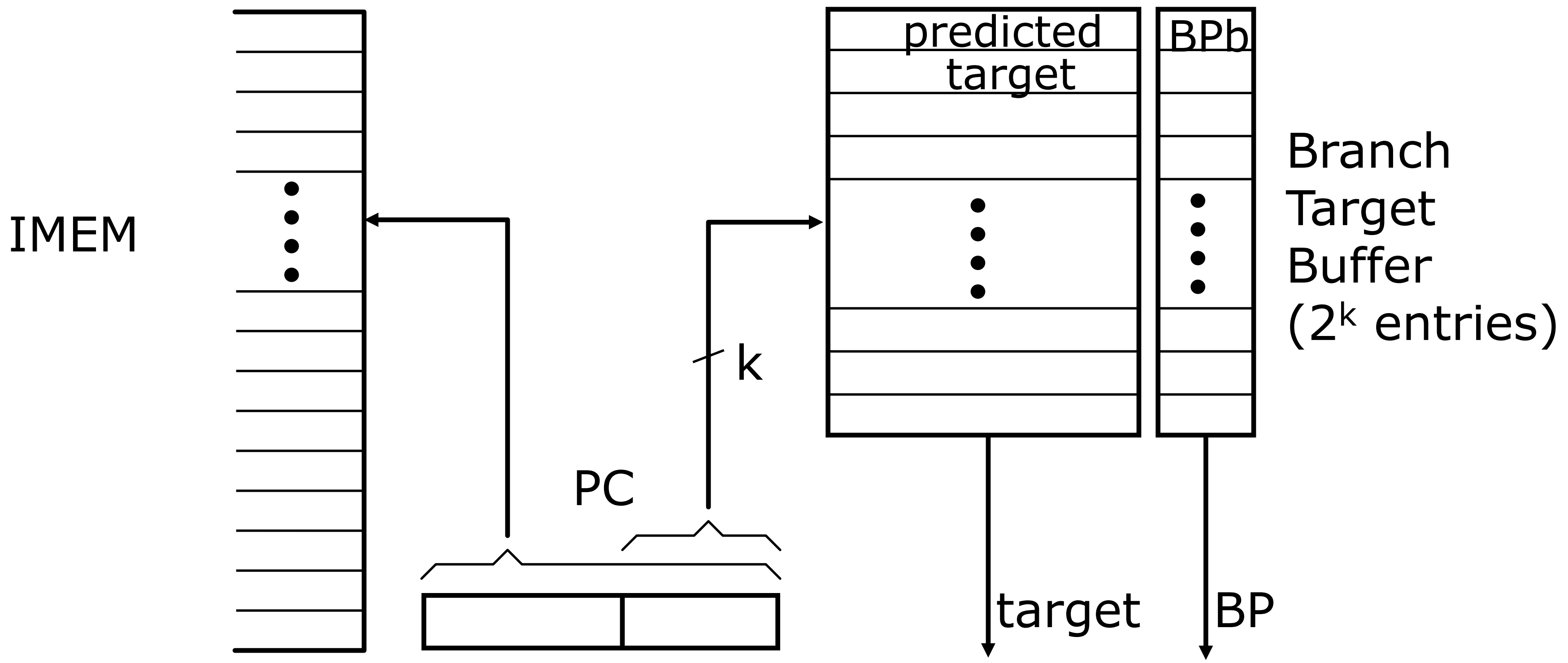
4K-entry branch history table, 80-90% accurate

Two-Level Branch Prediction

Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)


0 0
Fetch PC
k
2-bit global branch history shift register
Shift in Taken/¬Taken results of each branch
Taken/¬Taken?

Branch Target Buffer (BTB)



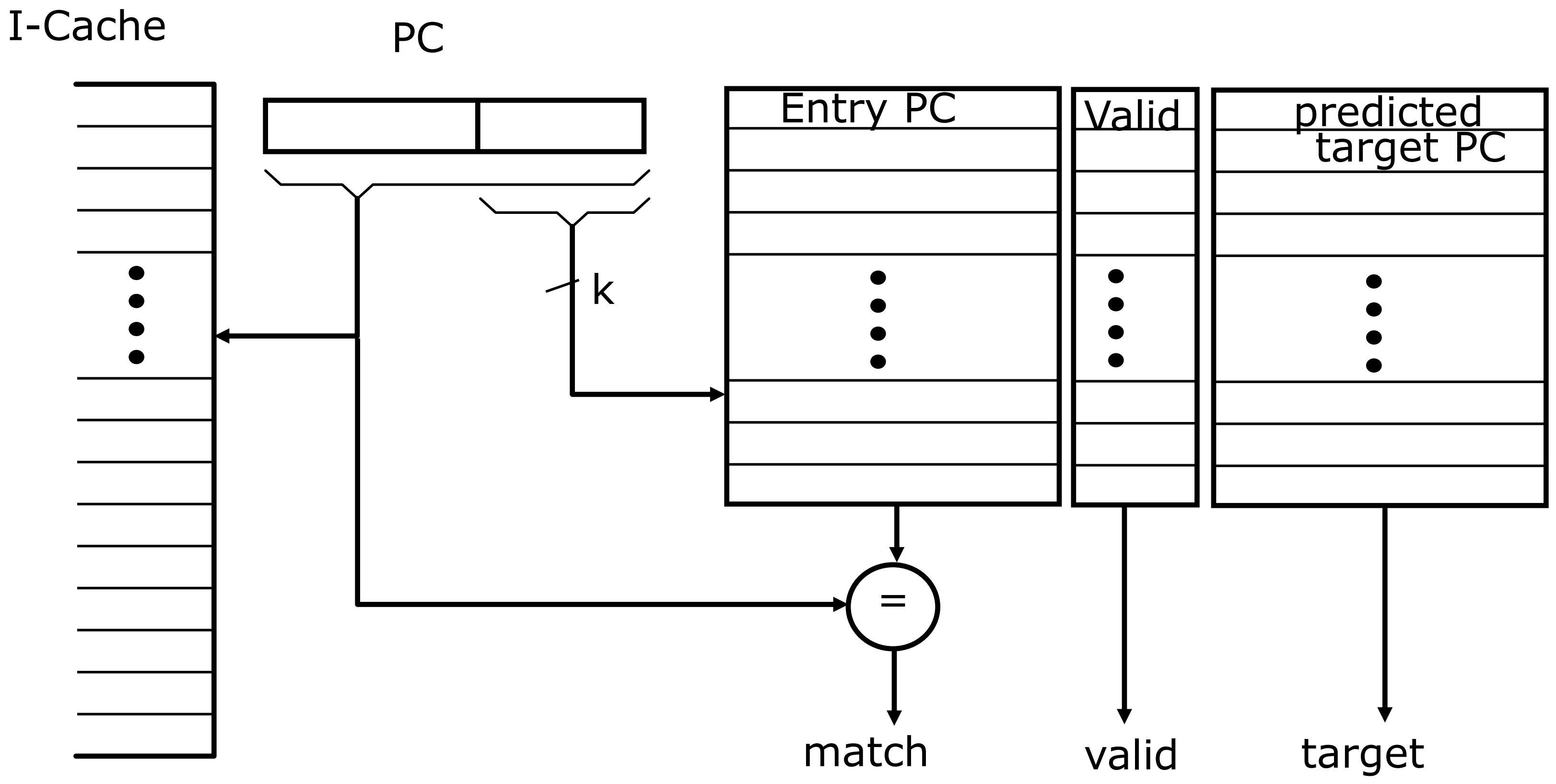
BHT only predicts branch direction (taken, not taken). Cannot redirect instruction flow until after branch target determined.

Store target with branch predictions.

During fetch – if (BP == taken) then nPC=target, else nPC=PC+4

Later – update BHT, BTB

Branch Target Buffer (BTB) – v2



Keep both branch PC and target PC in the BTB

If match fails, PC+4 is fetched

Only taken branches and jumps held in BTB

Mispredict Recovery

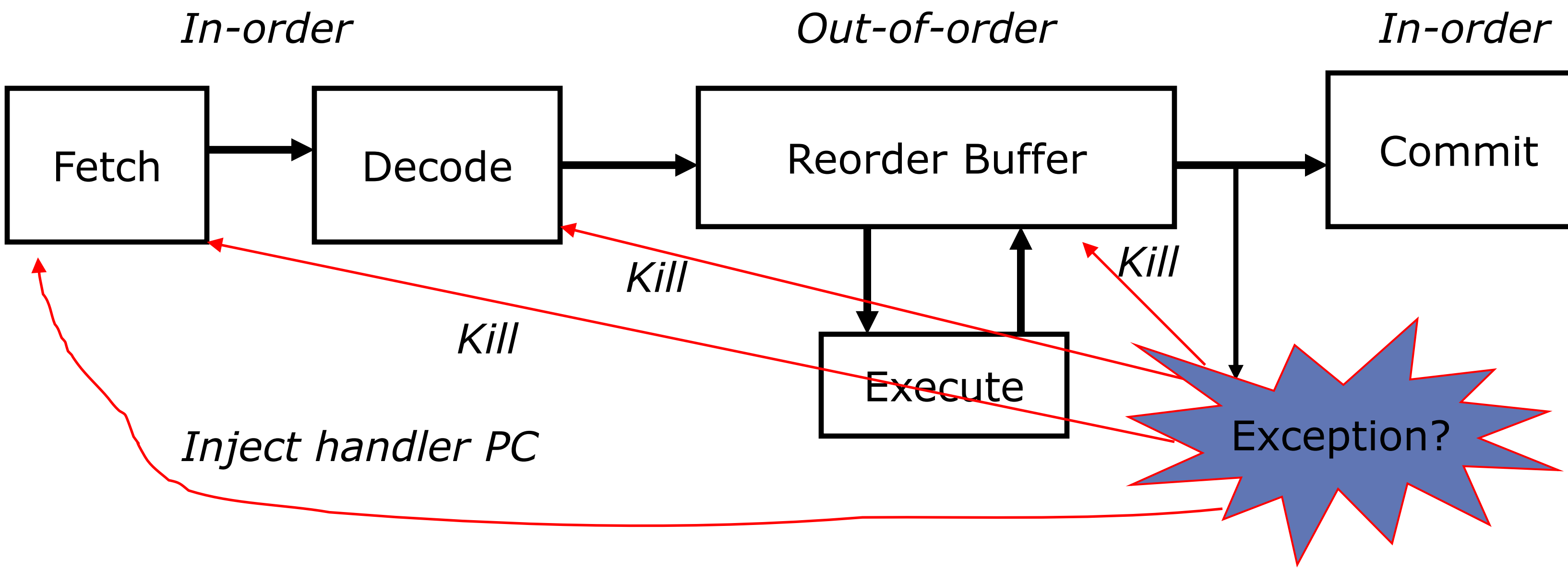
In-order execution

- No instruction following branch can commit before branch resolves
- Kill all instructions in pipeline behind mis-predicted branch

Out-of-order execution

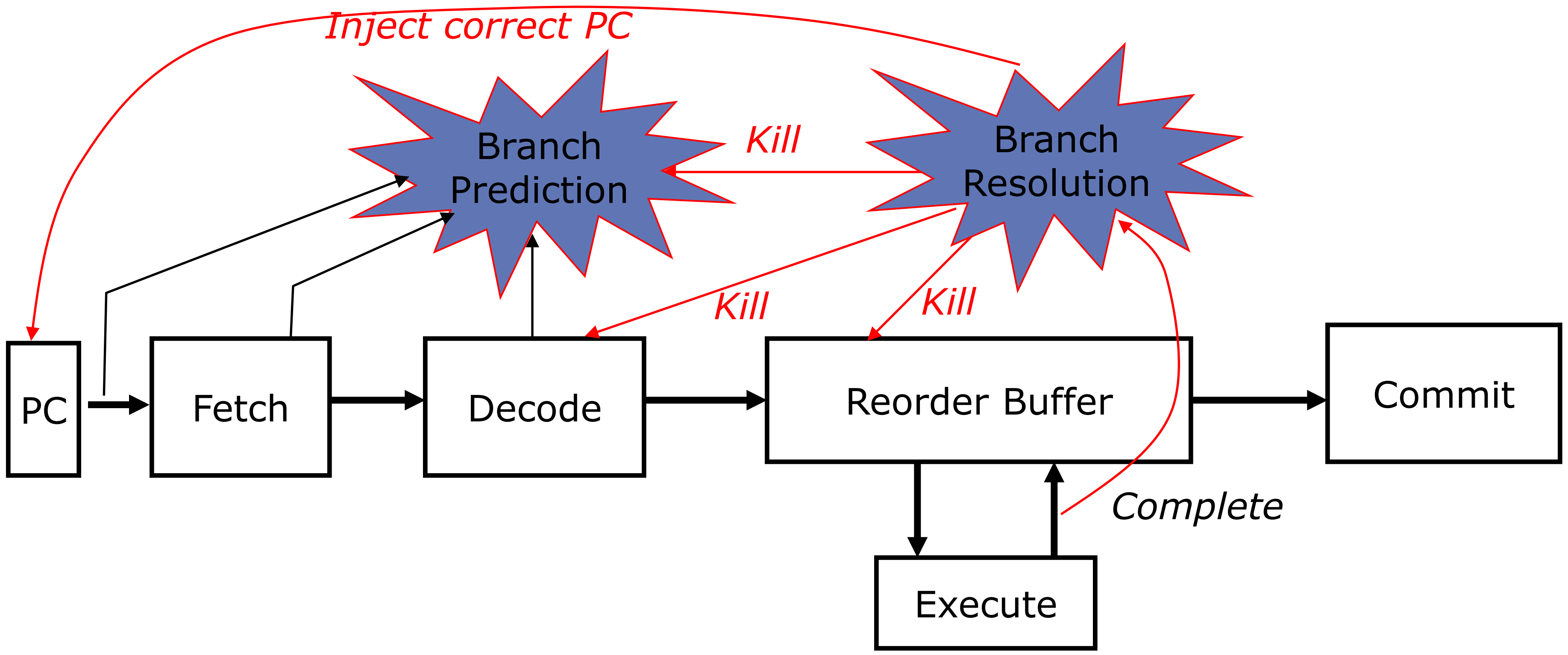
- Multiple instructions following branch can complete before one branch resolves

In-order Commit



-- Instructions fetched, decoded in-order (entering the reorder buffer -- ROB)
-- Instructions executed out-of-order
-- Instructions commit in-order (write back to architectural state)
-- Temporary storage needed in ROB to hold results before commit

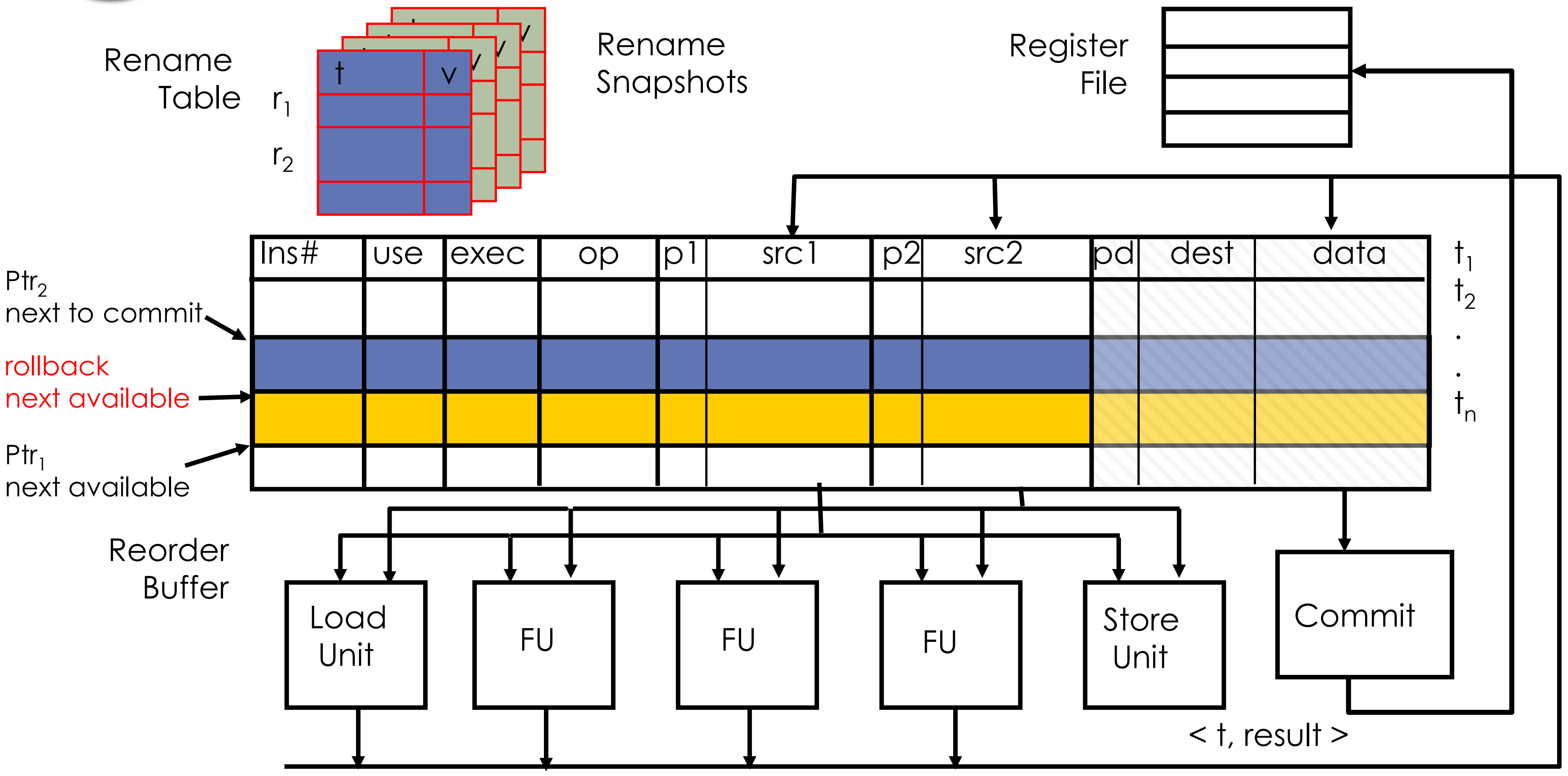
Branch Misprediction in Pipeline



-- Can have multiple unresolved branches in reorder buffer -- ROB

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

Mispredict Recovery



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

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)
- Alvin Lebeck (Duke)
- David Patterson (UCB)
- Daniel Sorin (Duke)