CS252 Graduate Computer Architecture
Spring 2014
Lecture 7: Advanced Out-of-Order Superscalar Designs

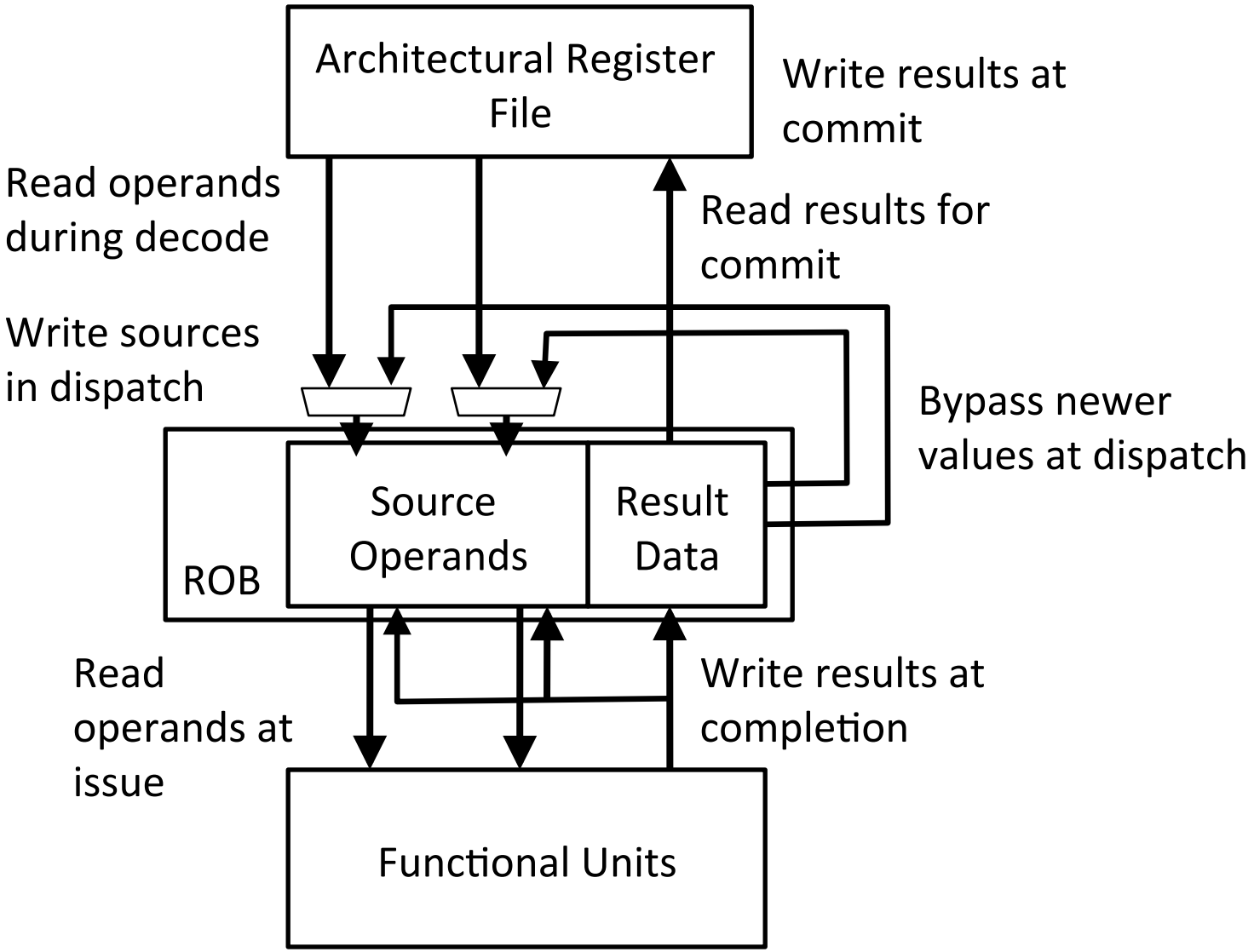
Krste Asanovic
krste@eecs.berkeley.edu
http://inst.eecs.berkeley.edu/~cs252/fa15

Last Time in Lecture 6

Modern Out-of-Order Architectures with Precise Traps

- Data-in-ROB design

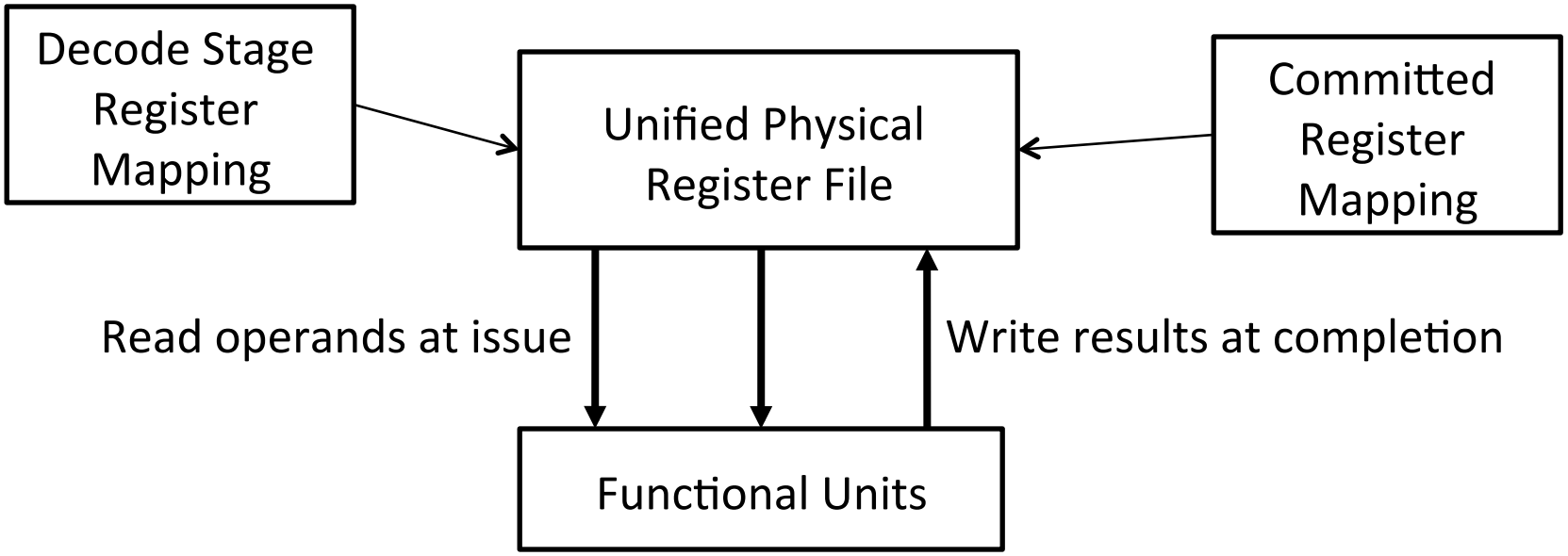
Data Movement in Data-in-ROB Design



Unified Physical Register File

(MIPS R10K, Alpha 21264, Intel Pentium 4 & Sandy/Ivy Bridge)

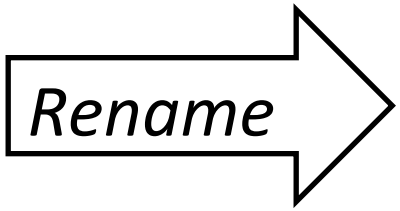
- Rename all architectural registers into a single *physical* register file during decode, no register values read
- Functional units read and write from single unified register file holding committed and temporary registers in execute
- Commit only updates mapping of architectural register to physical register, no data movement



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)
```

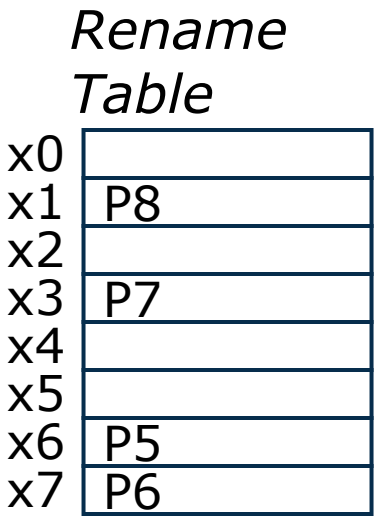


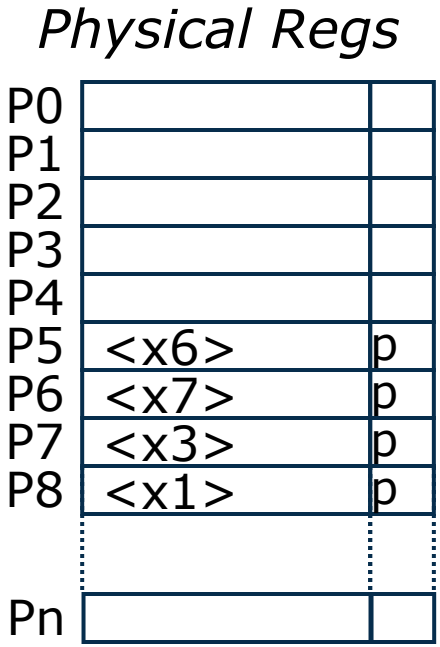

```
ld P1, (Px)
addi P2, P1, #4
sub P3, Py, Pz
add P4, P2, P3
ld P5, (P1)
add P6, P5, P4
sd P6, (P1)
ld P7, (Pw)
```

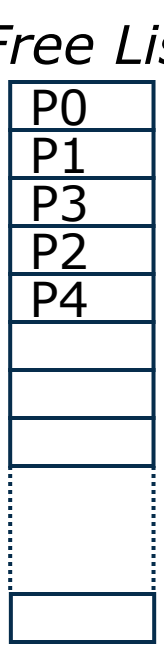
When can we reuse a physical register?

When next writer of same architectural register commits

Physical Register Management







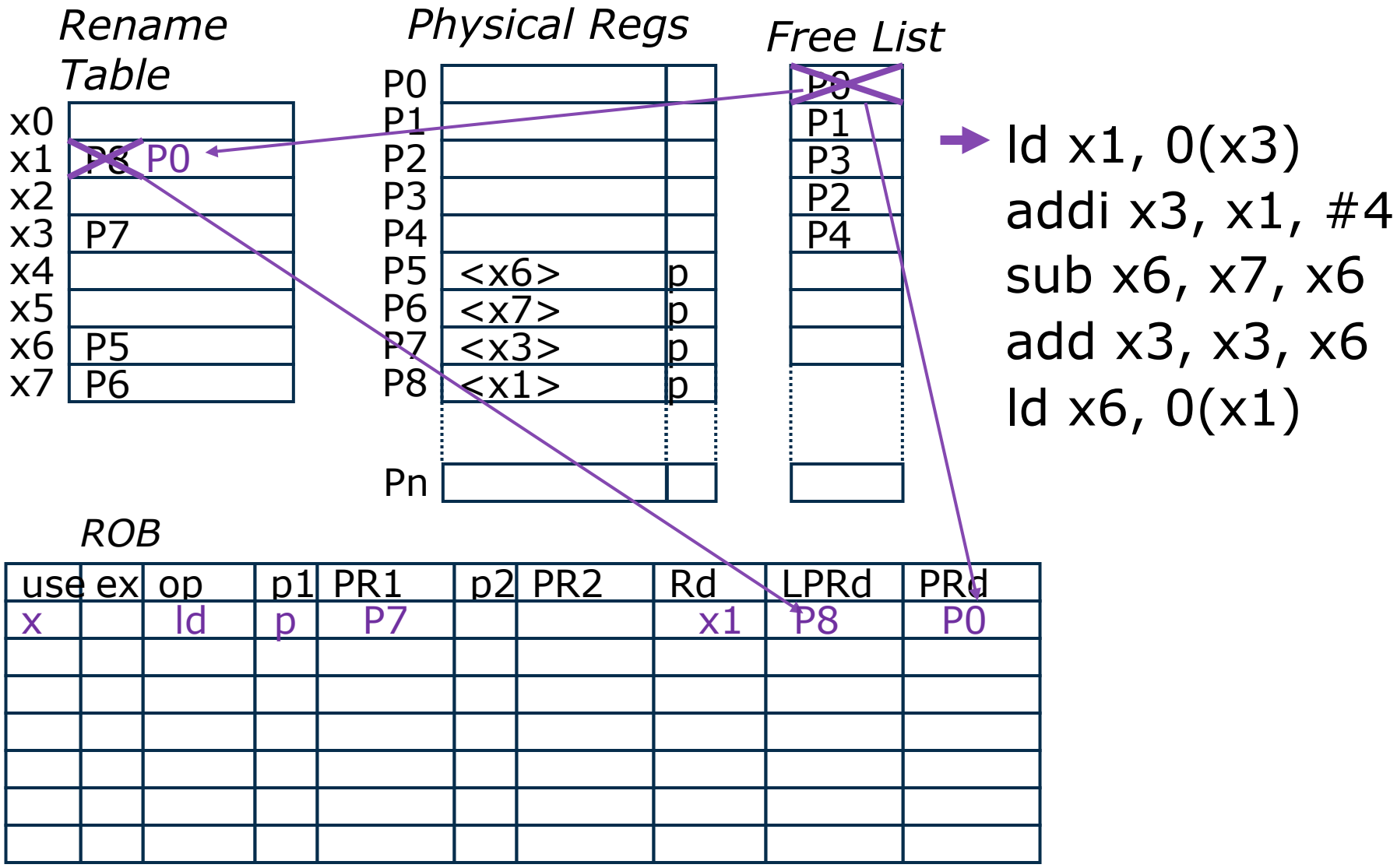

```
ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)
```

ROB

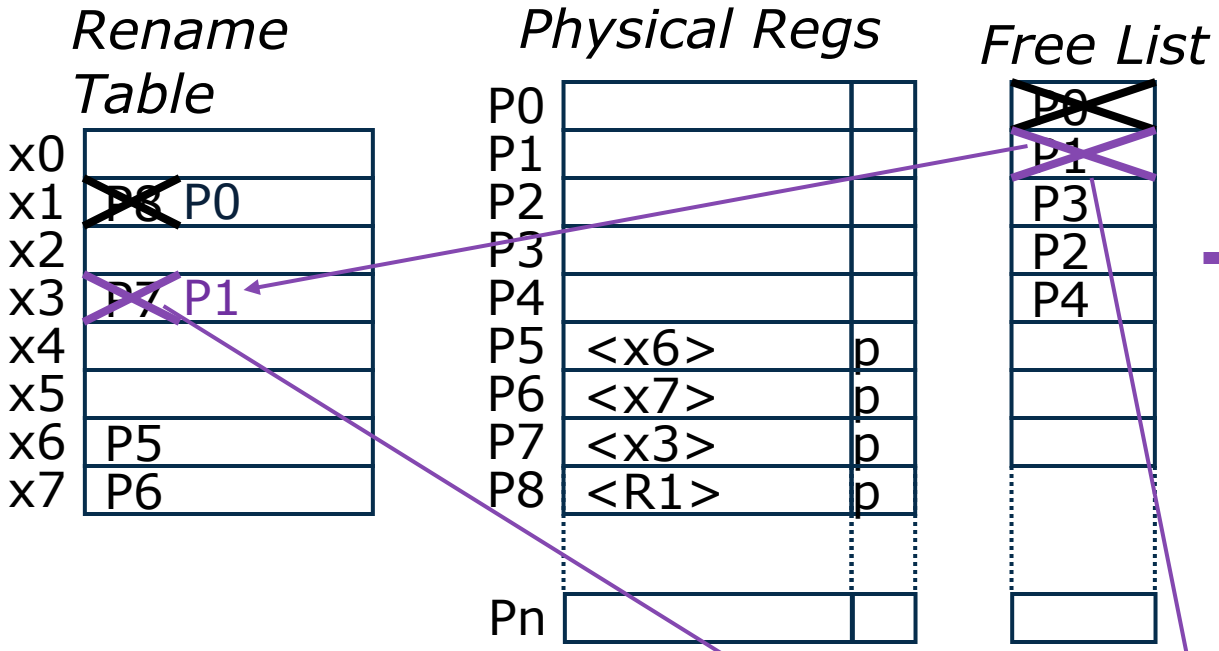
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd

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

Physical Register Management



Physical Register Management

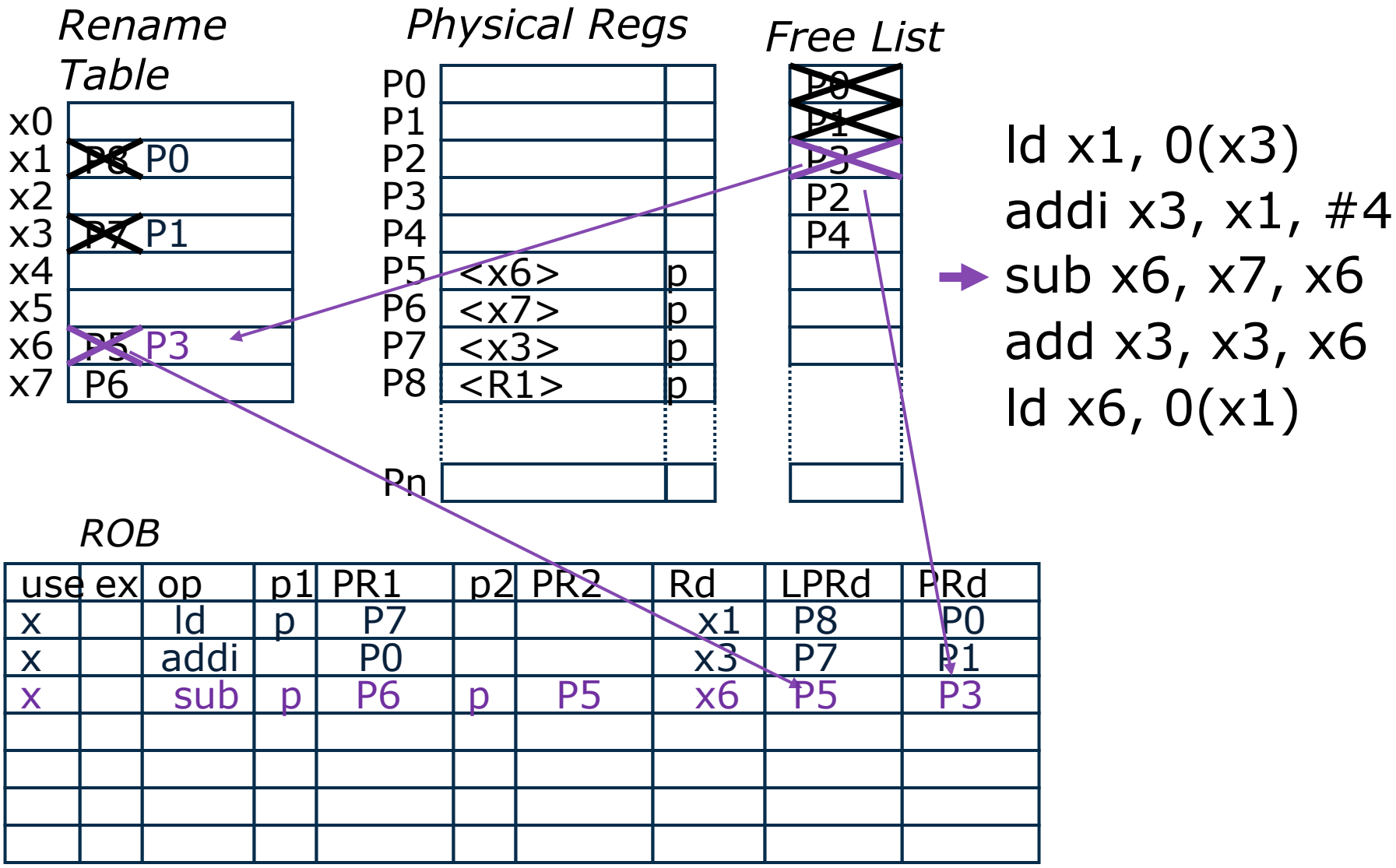



```
ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)
```

ROB

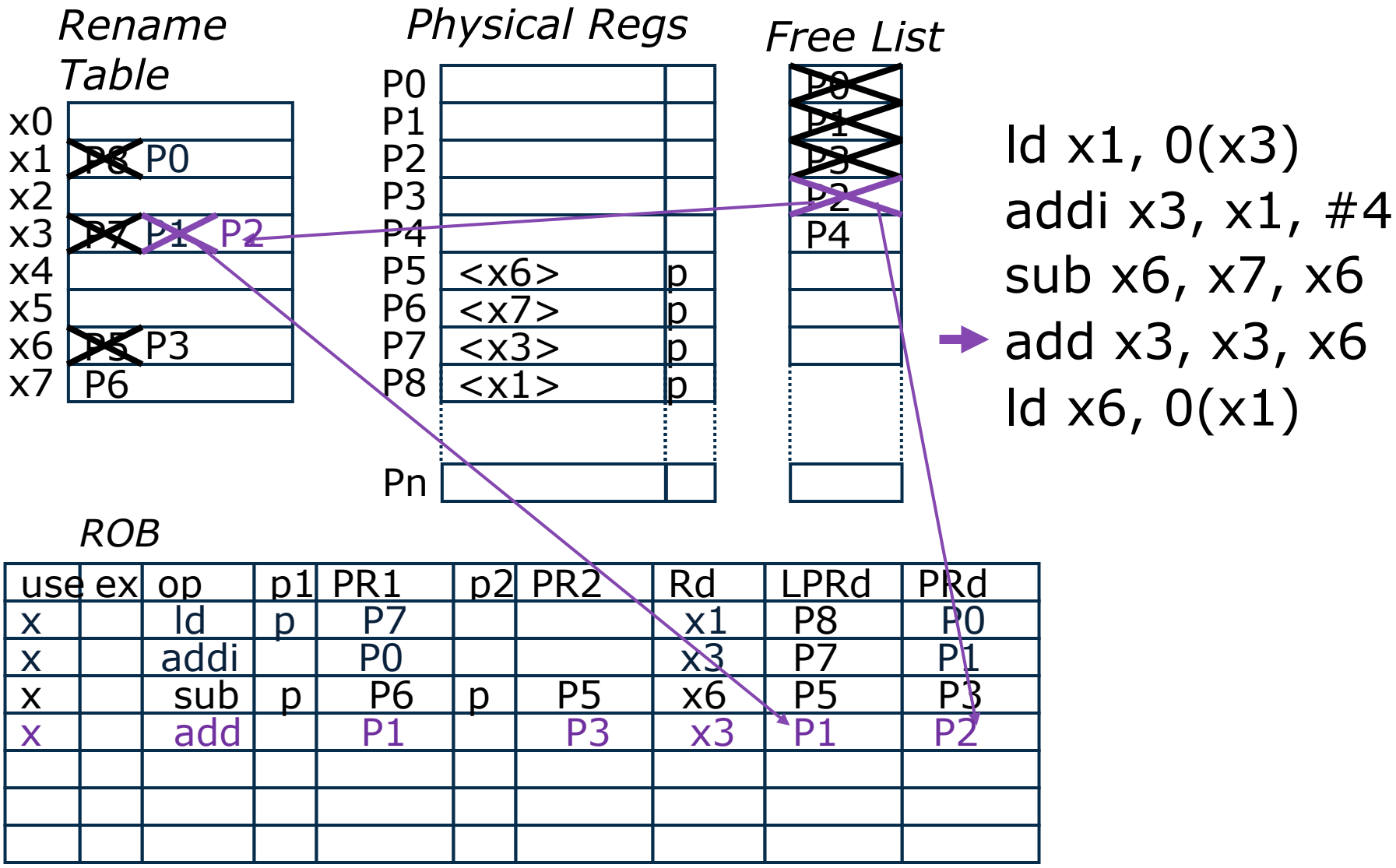
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1	

Physical Register Management



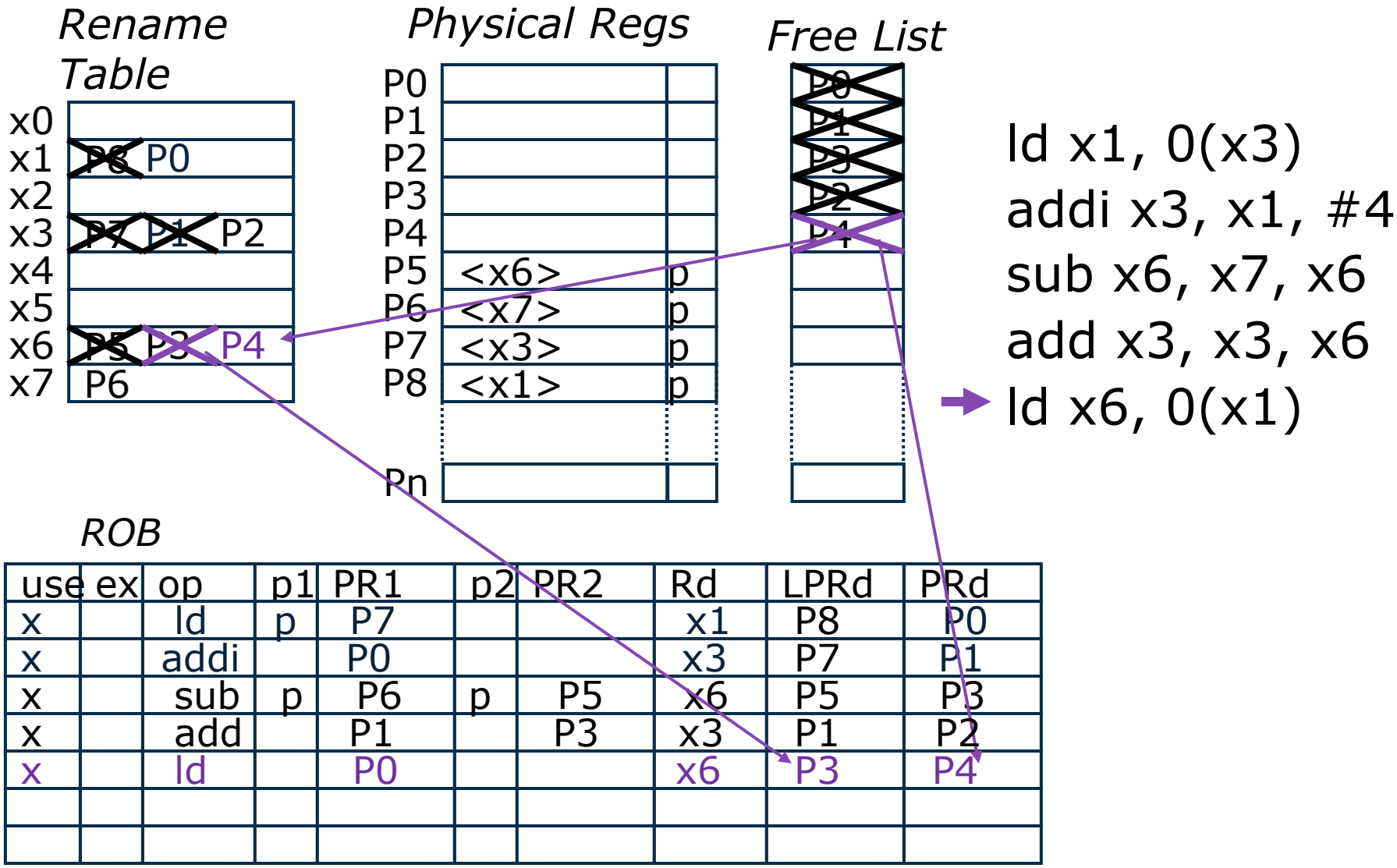
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3

Physical Register Management



use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add		P1		P3	x3	P1	P2

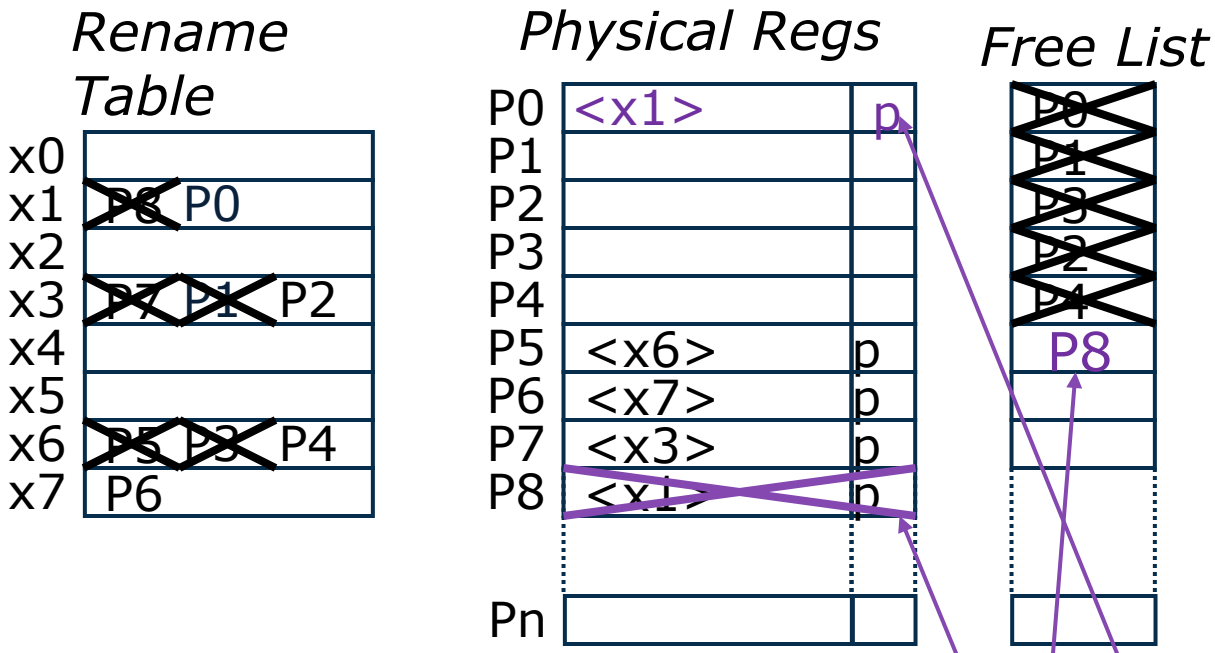
Physical Register Management



ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add		P1		P3	x3	P1	P2
x		ld		P0			x6	P3	P4

Physical Register Management



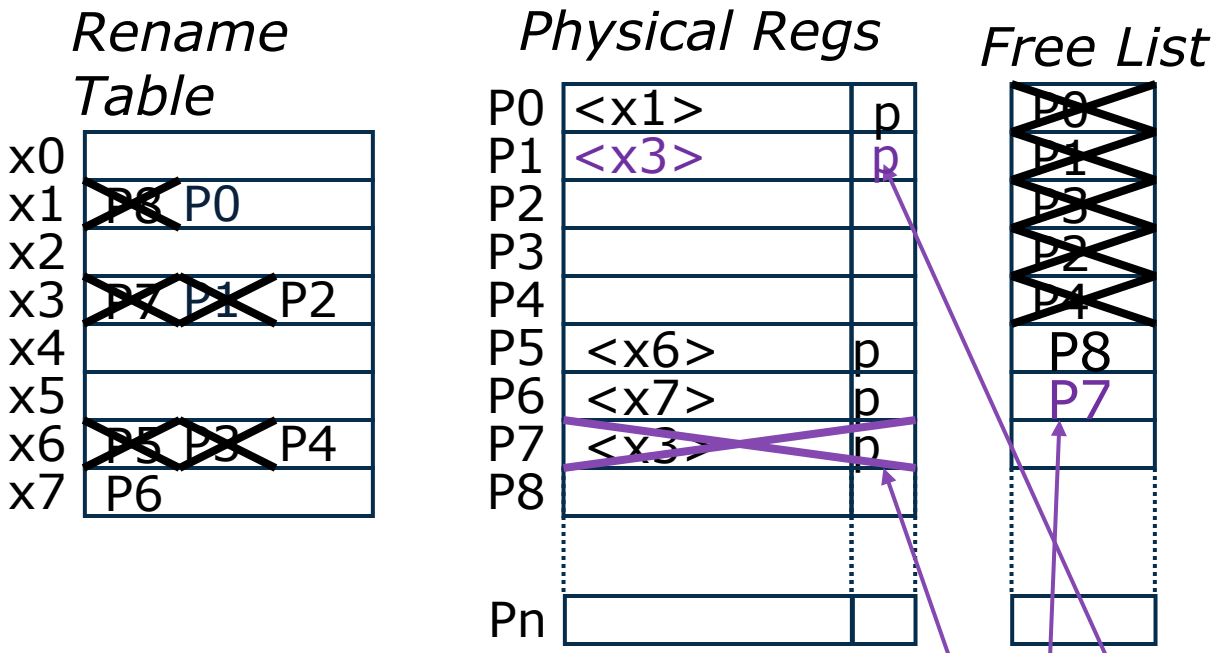

```
ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)
```

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x	x	ld	p	P7			x1	P8	P0
x		addi	p	P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add		P1		P3	x3	P1	P2
x		ld	p	P0			x6	P3	P4

Execute & Commit

Physical Register Management




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

ROB

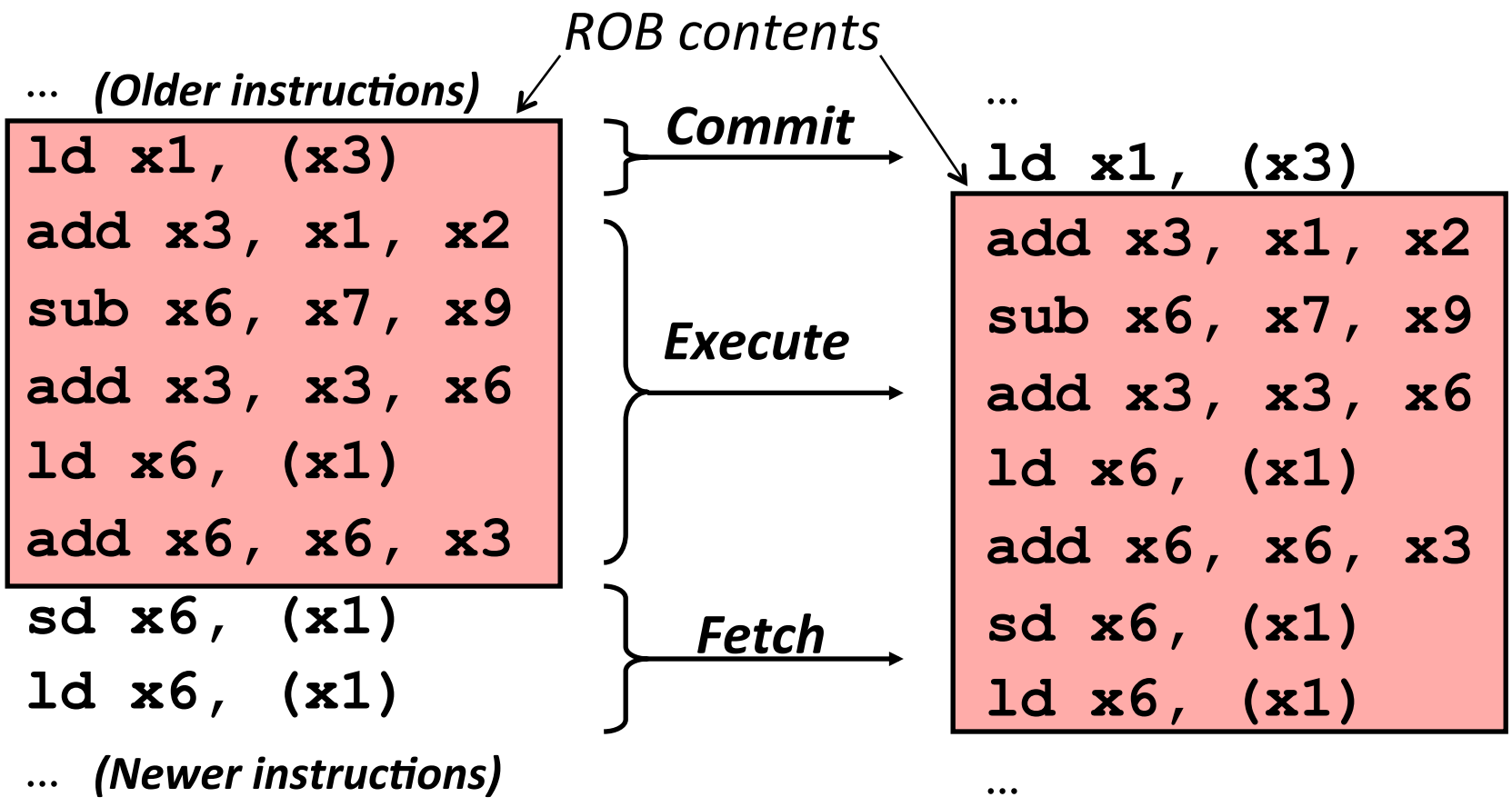
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x	x	ld	p	P7			x1	P8	P0
x	x	addi	p	P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add	p	P1		P3	x3	P1	P2
x		ld	p	P0			x6	P3	P4

Execute & Commit

MIPS R10K Trap Handling

- Rename table is repaired by unrenaming instructions in reverse order using the PRd/LPRd fields
- The Alpha 21264 had similar physical register file scheme, but kept complete rename table snapshots for each instruction in ROB (80 snapshots total)
  - Flash copy all bits from snapshot to active table in one cycle

Reorder Buffer Holds Active Instructions (Decoded but not Committed)

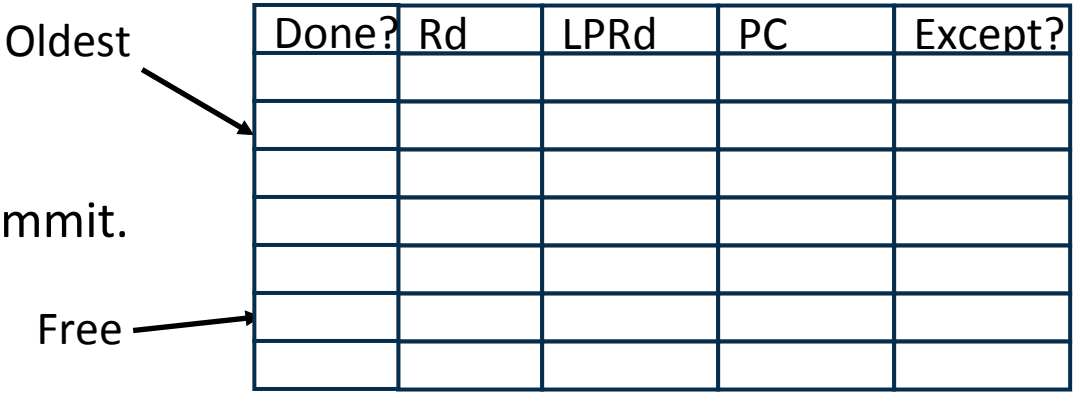


Separate Issue Window from ROB

The issue window holds only instructions that have been decoded and renamed but not issued into execution. Has register tags and presence bits, and pointer to ROB entry.

use	ex	op	p1	PR1	p2	PR2	PRd	ROB#

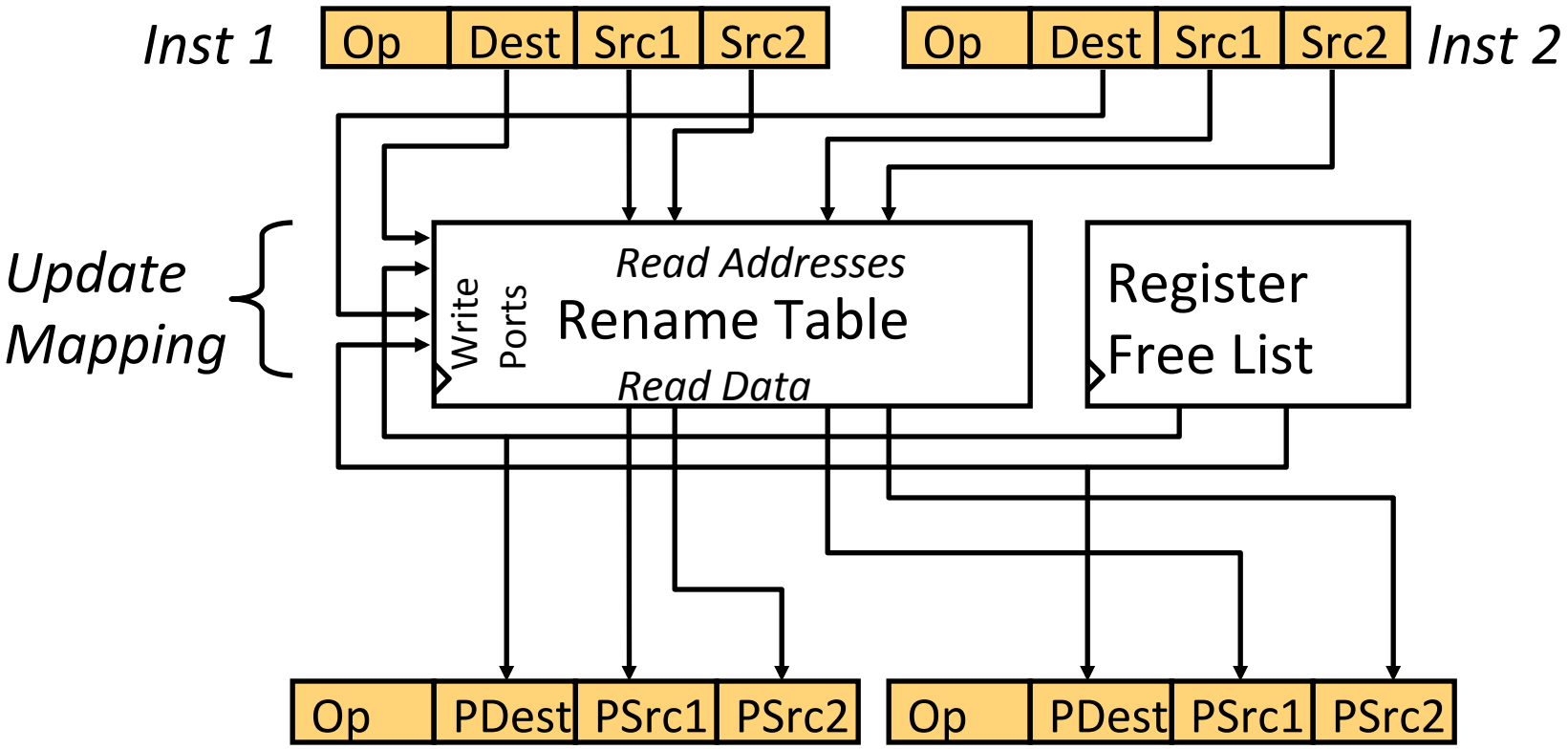
Reorder buffer used to hold exception information for commit.



ROB is usually several times larger than issue window – why?

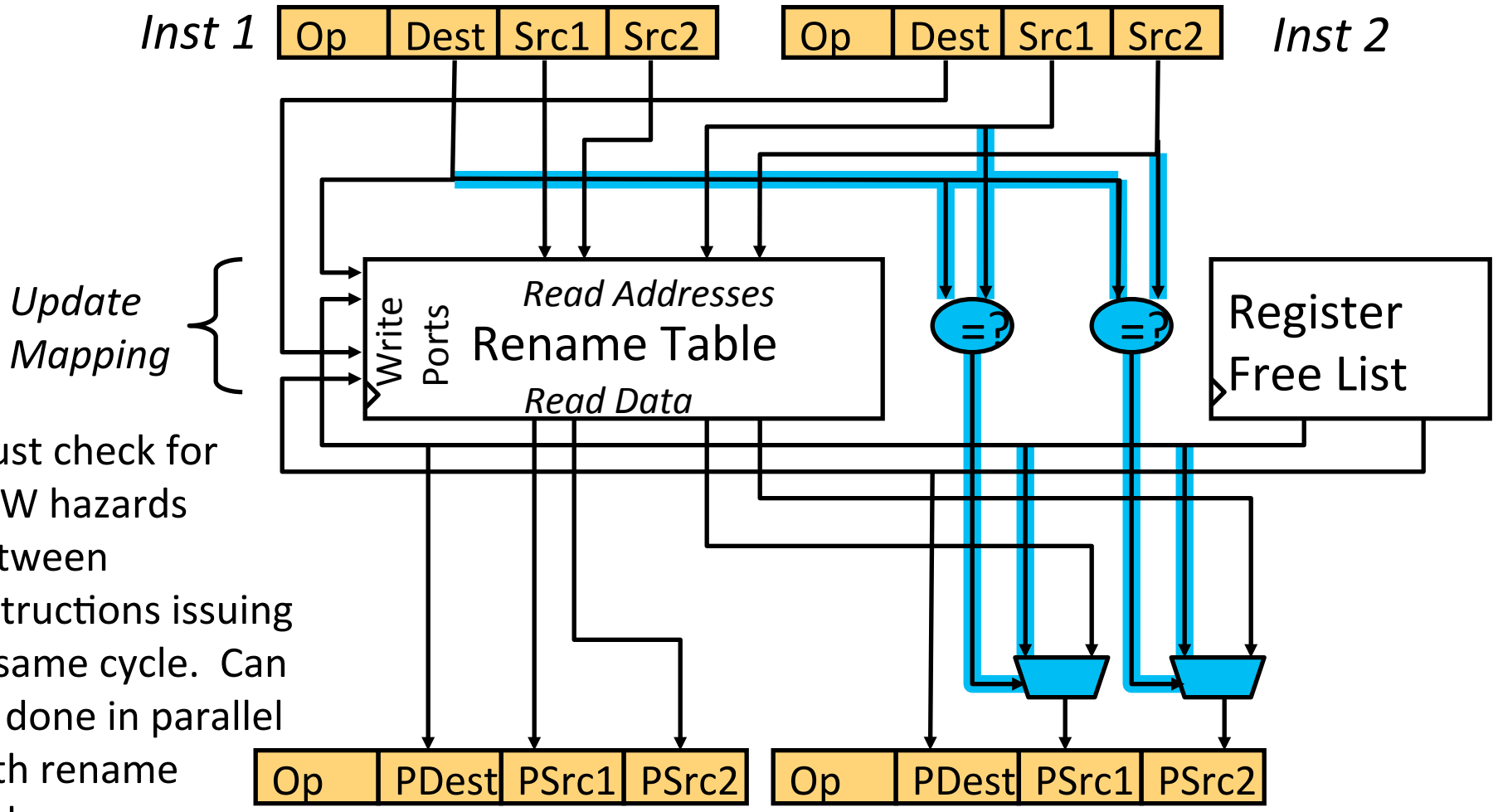
Superscalar Register Renaming

- 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.

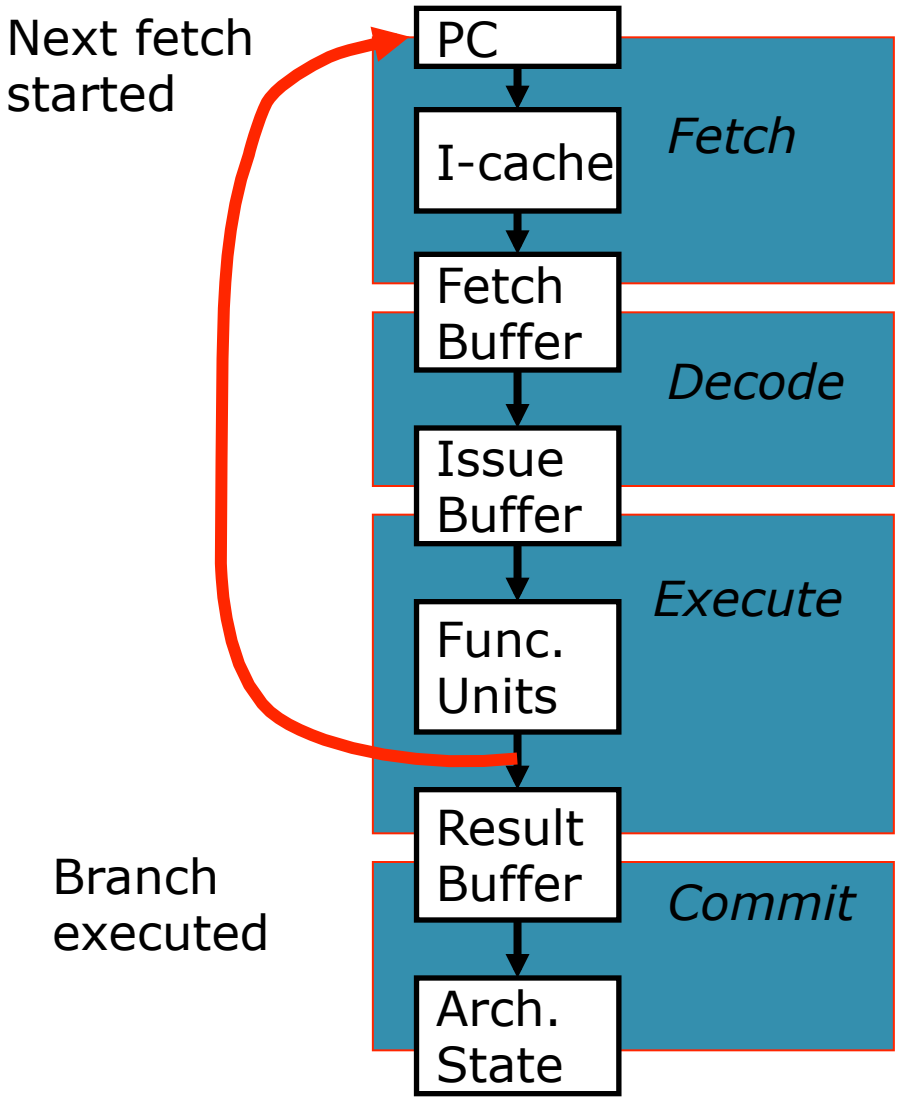
MIPS R10K renames 4 serially-RAW-dependent insts/cycle

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 + buffers



Reducing Control Flow Penalty

- Software solutions
  - Eliminate branches - loop unrolling
    - Increases the run length
  - Reduce resolution time - instruction scheduling
    - Compute the branch condition as early as possible (of limited value because branches often in critical path through code)

- Hardware solutions
  - Find something else to do - delay slots
    - Replaces pipeline bubbles with useful work (requires software cooperation) – quickly see diminishing returns
  - Speculate - branch prediction
    - Speculative execution of instructions beyond the branch
    - Many advances in accuracy

Branch Prediction

Motivation:

Branch penalties limit performance of deeply pipelined processors

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

Required hardware support:

Prediction structures:

- Branch history tables, branch target buffers, etc.

Mispredict recovery mechanisms:

- *Keep result computation separate from commit*
- Kill instructions following branch in pipeline
- Restore state to that following branch

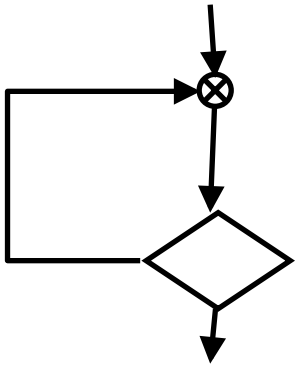
Importance of Branch Prediction

- Consider 4-way superscalar with 8 pipeline stages from fetch to dispatch, and 80-entry ROB, and 3 cycles from issue to branch resolution
- On a mispredict, could throw away 8*4+(80-1)=111 instructions
- Improving from 90% to 95% prediction accuracy, removes 50% of branch mispredicts
  - If 1/6 instructions are branches, then move from 60 instructions between mispredicts, to 120 instructions between mispredicts

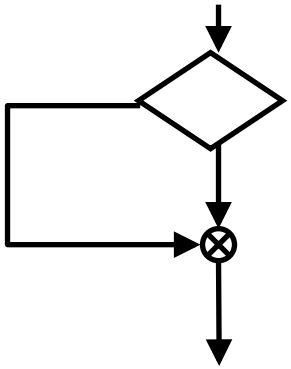
Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:

backward
90%



forward
50%



ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110

bne0 *(preferred taken)* beq0 *(not taken)*

ISA can allow arbitrary choice of statically predicted direction, e.g., HP PA-RISC, Intel IA-64

typically reported as ~80% accurate

Dynamic Branch Prediction
learning based on past behavior

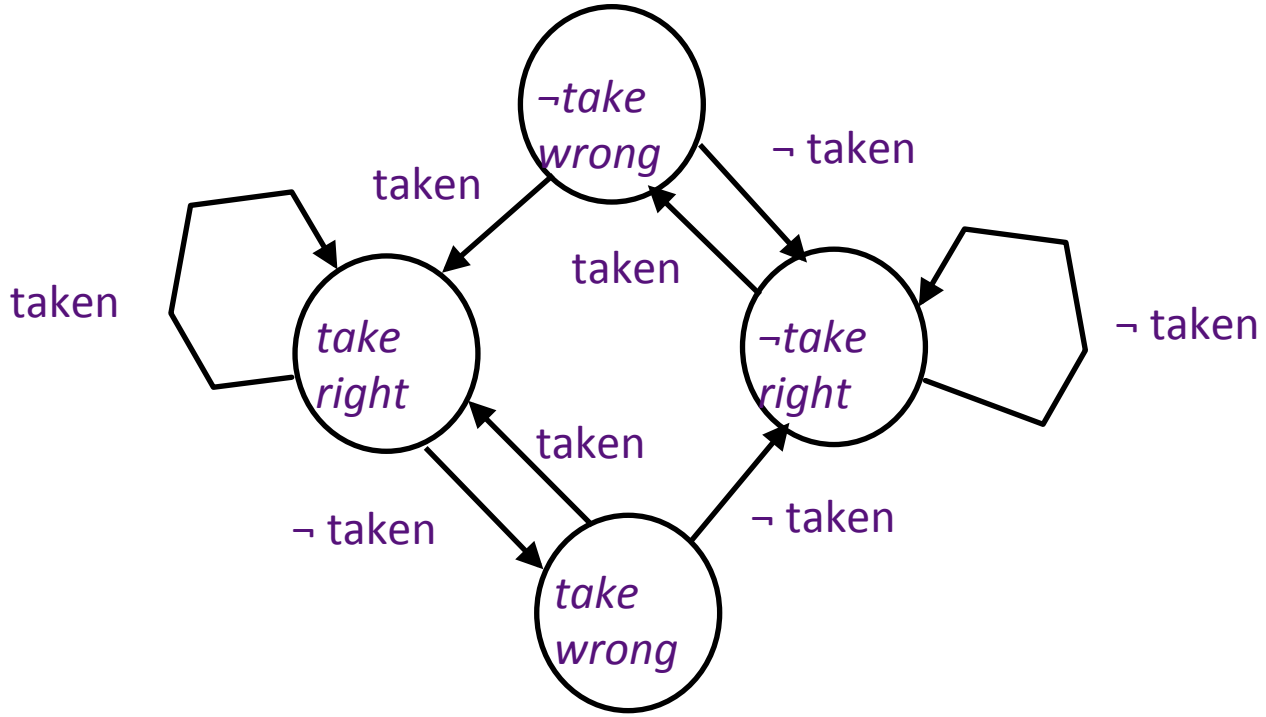
- Temporal correlation
  - The way a branch resolves may be a good predictor of the way it will resolve at the next execution
- Spatial correlation
  - Several branches may resolve in a highly correlated manner (a preferred path of execution)

One-Bit Branch History Predictor

- For each branch, remember last way branch went
- Has problem with loop-closing backward branches, as two mispredicts occur on every loop execution
  1. first iteration predicts loop backwards branch not-taken (loop was exited last time)
  2. last iteration predicts loop backwards branch taken (loop continued last time)

Branch Prediction Bits

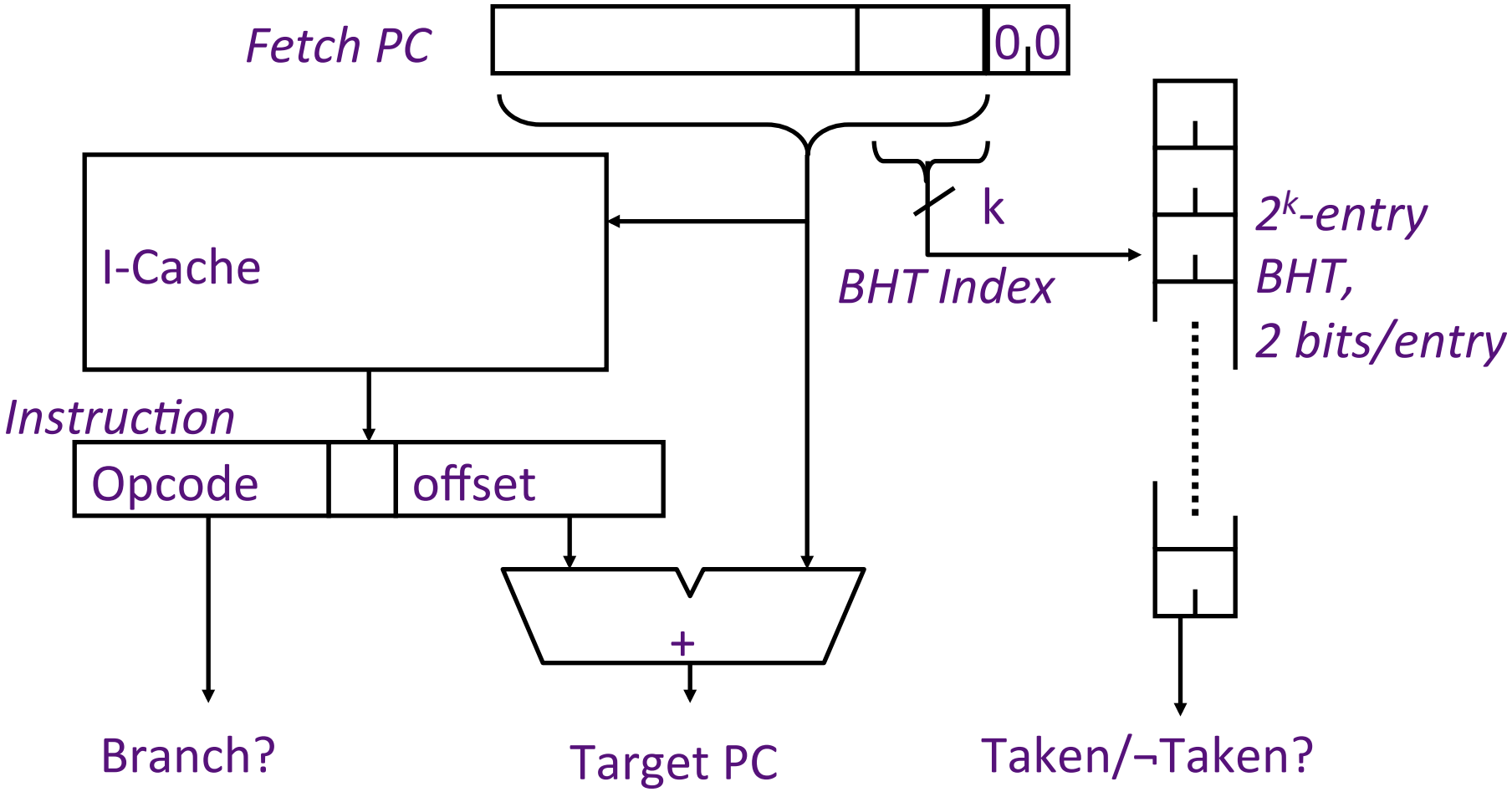
- Assume 2 BP bits per instruction
- Change the prediction after two consecutive mistakes!



BP state:

(*predict* take/¬take) x (*last prediction* right/wrong)

Branch History Table (BHT)



4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

Exploiting Spatial Correlation

Yeh and Patt, 1992

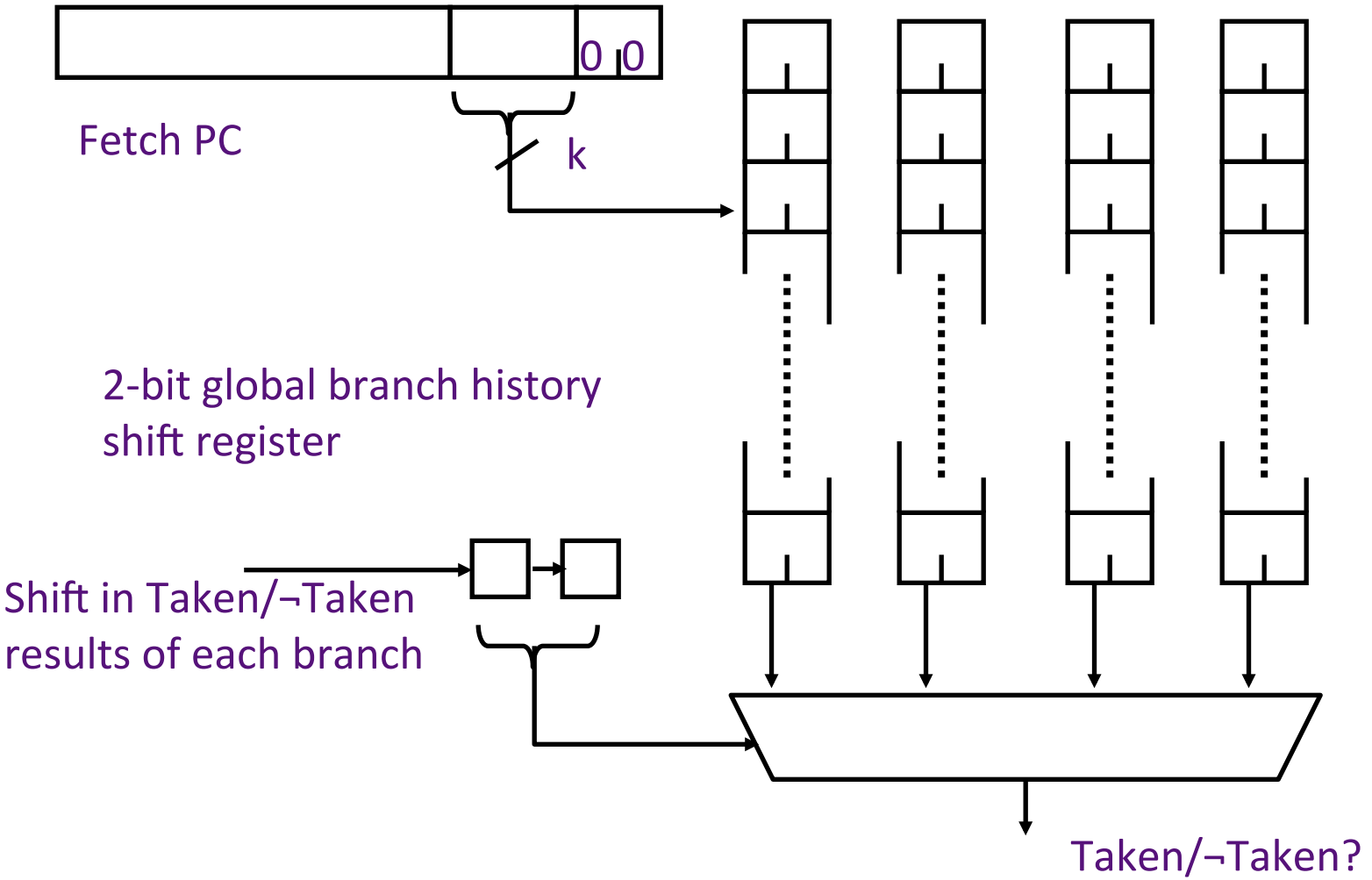
```
if (x[i] < 7) then
     y += 1;
if (x[i] < 5) then
     c -= 4;
```

If first condition false, second condition also false

History register, H, records the direction of the last N branches executed by the processor

Two-Level Branch Predictor

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

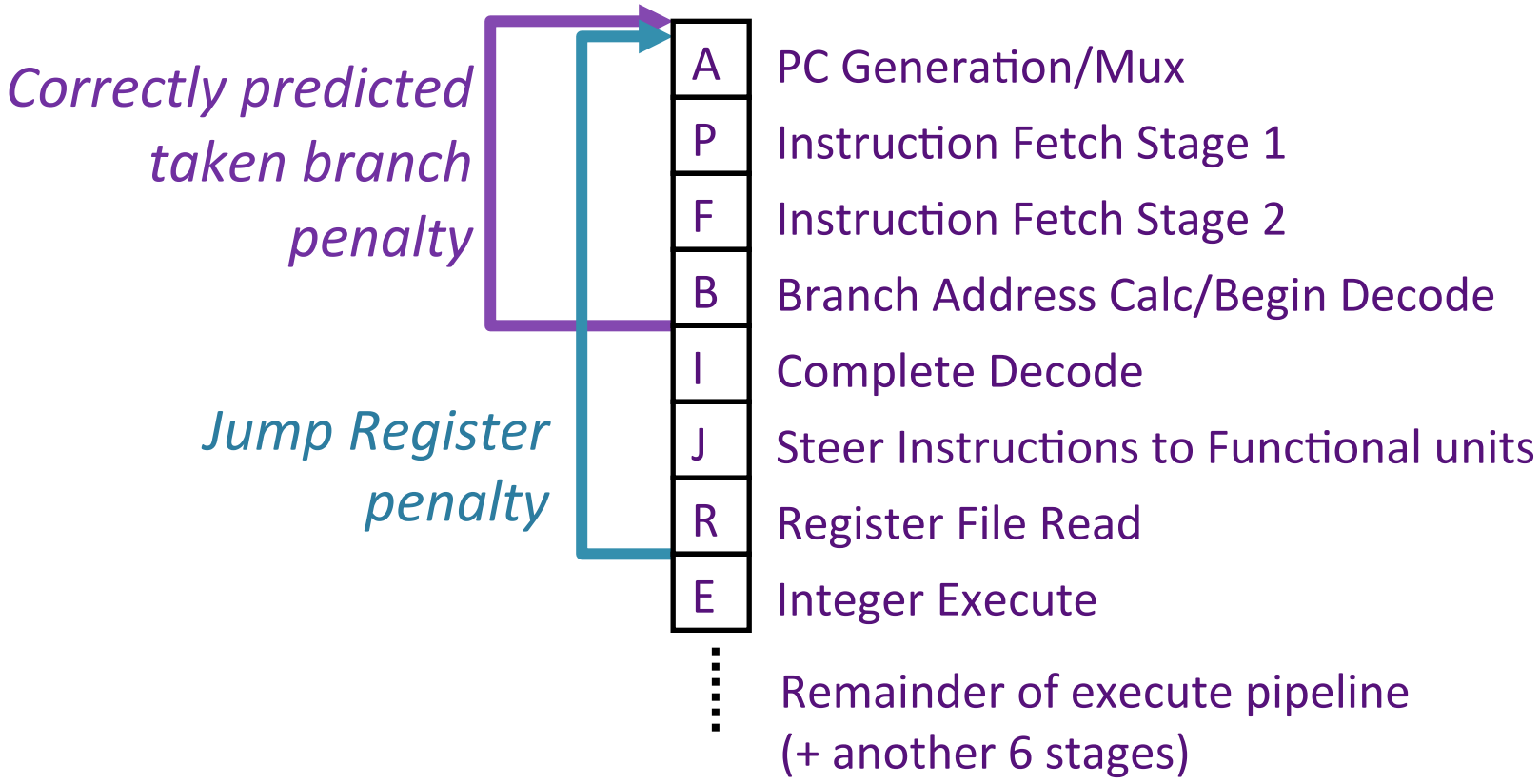


Speculating Both Directions

- 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!

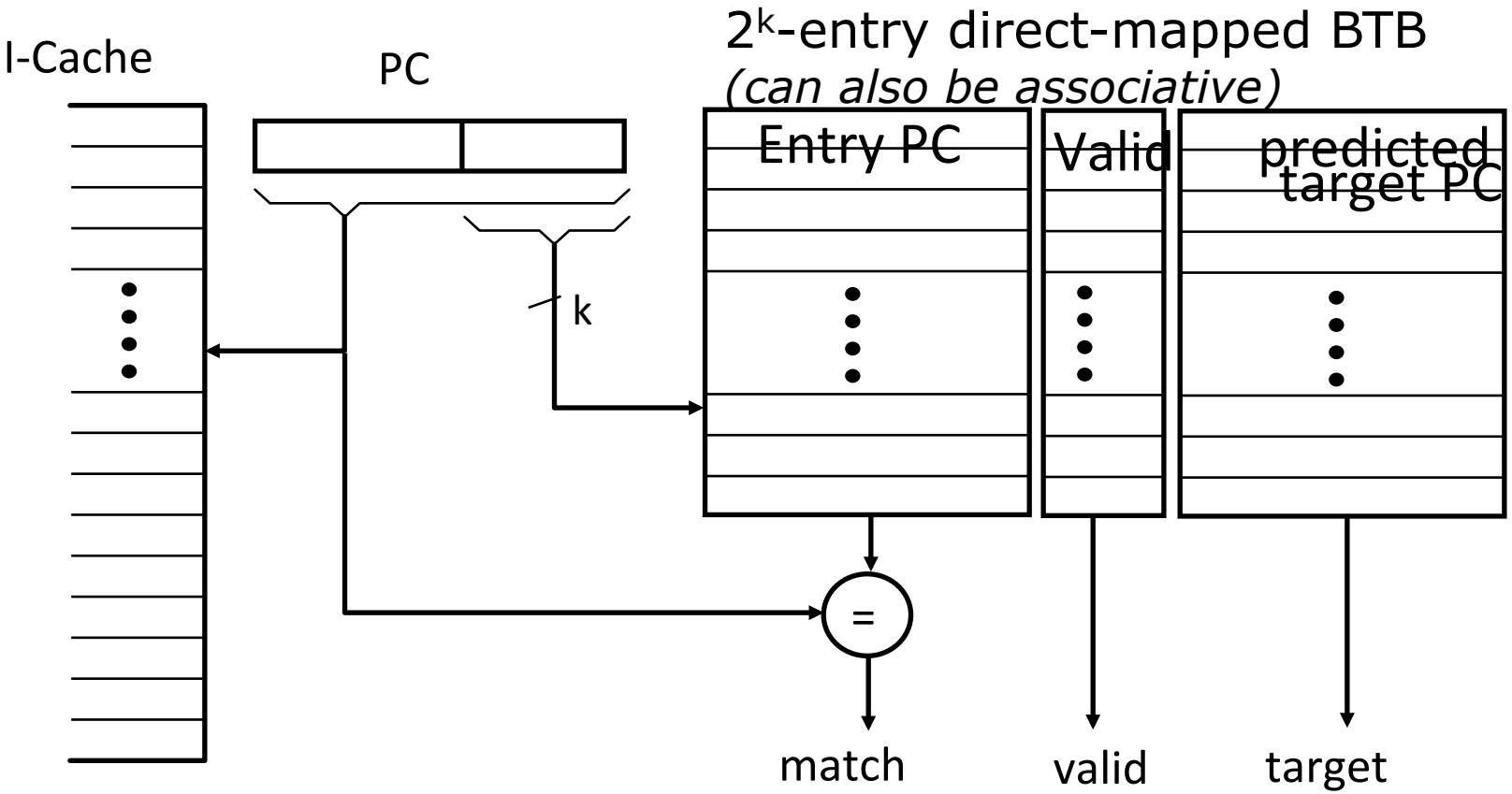
Limitations of BHTs

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



UltraSPARC-III fetch pipeline

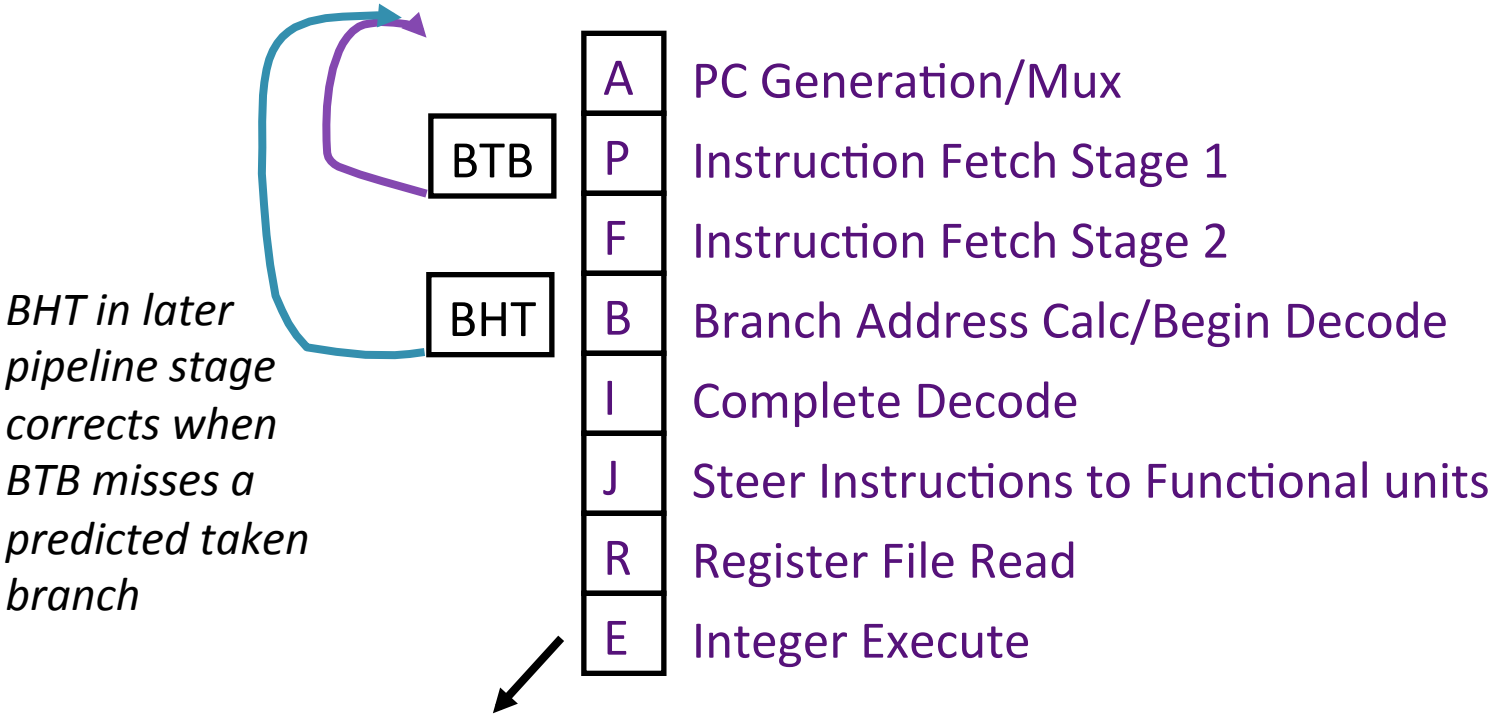
Branch Target Buffer (BTB)



- 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 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

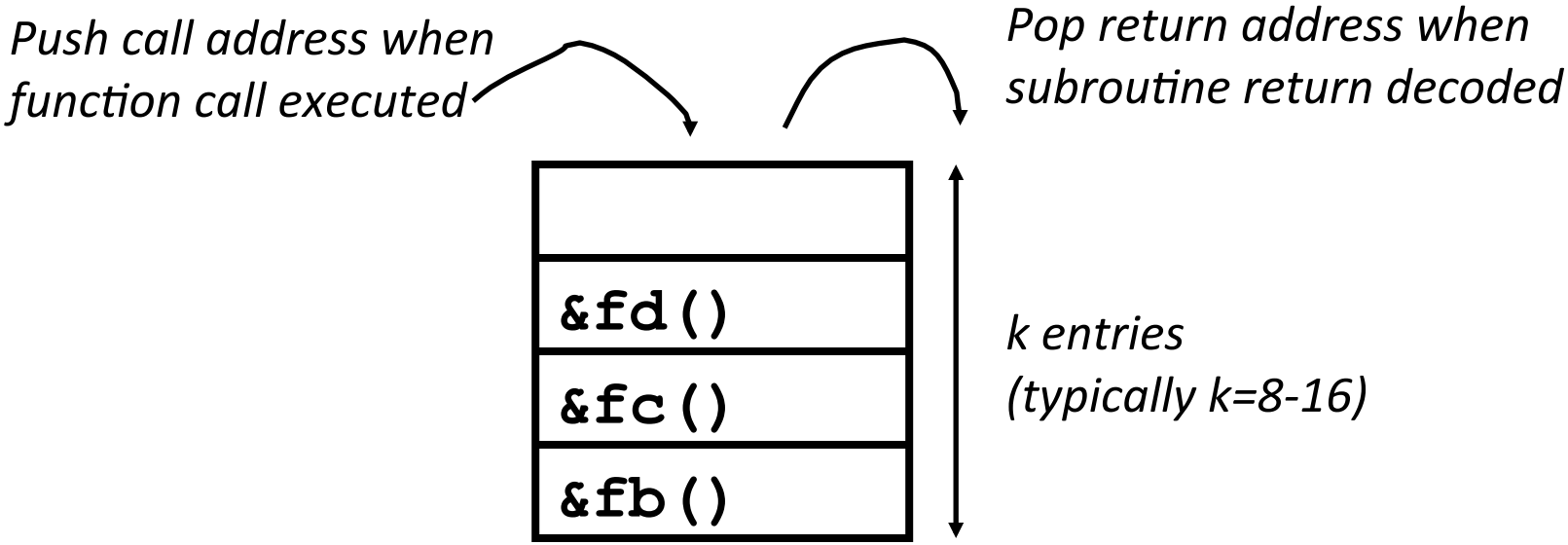
  ⇒ *Often one function called from many distinct call sites!*

How well does BTB work for each of these cases?

Subroutine Return Stack

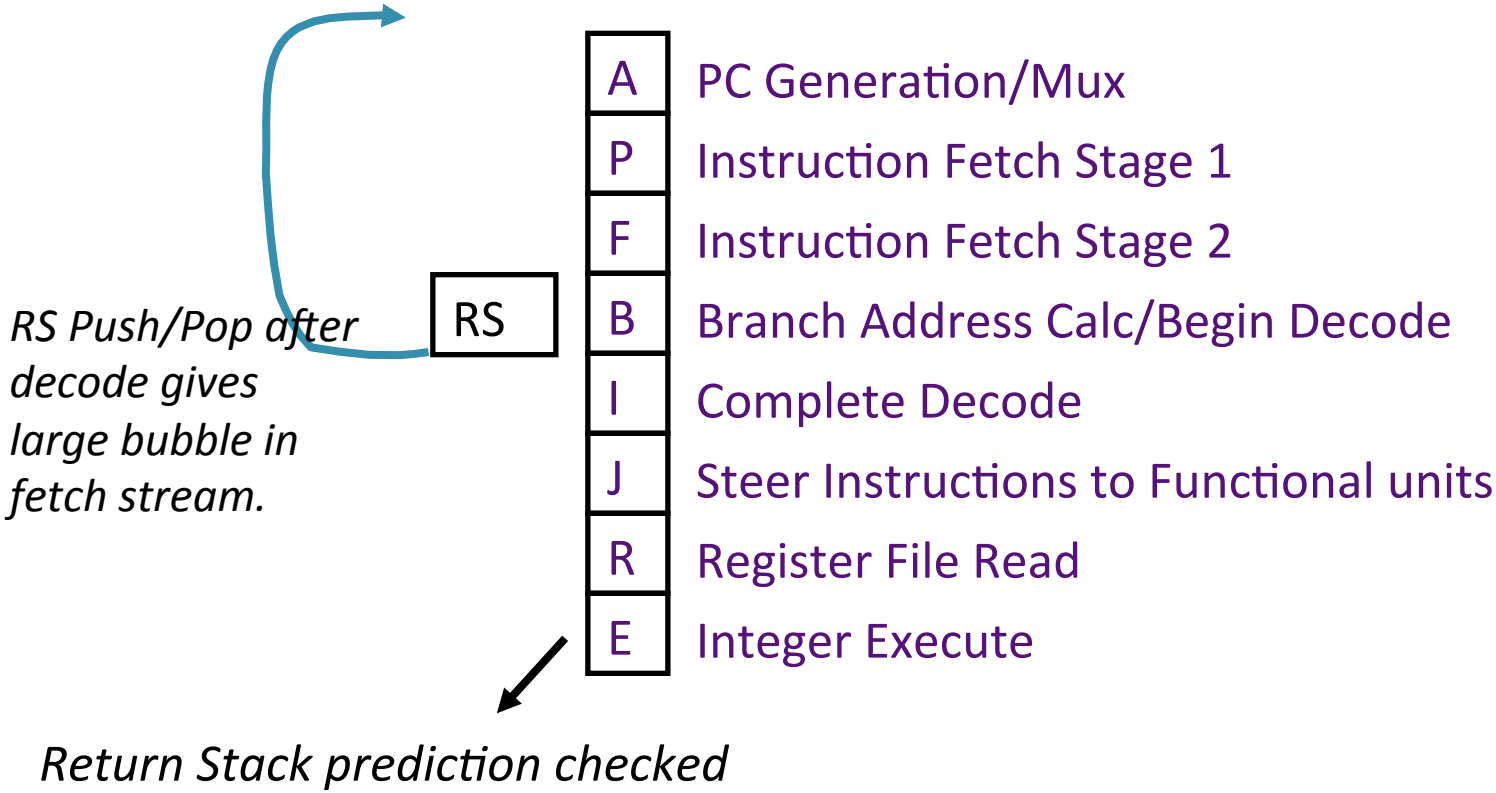
Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

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



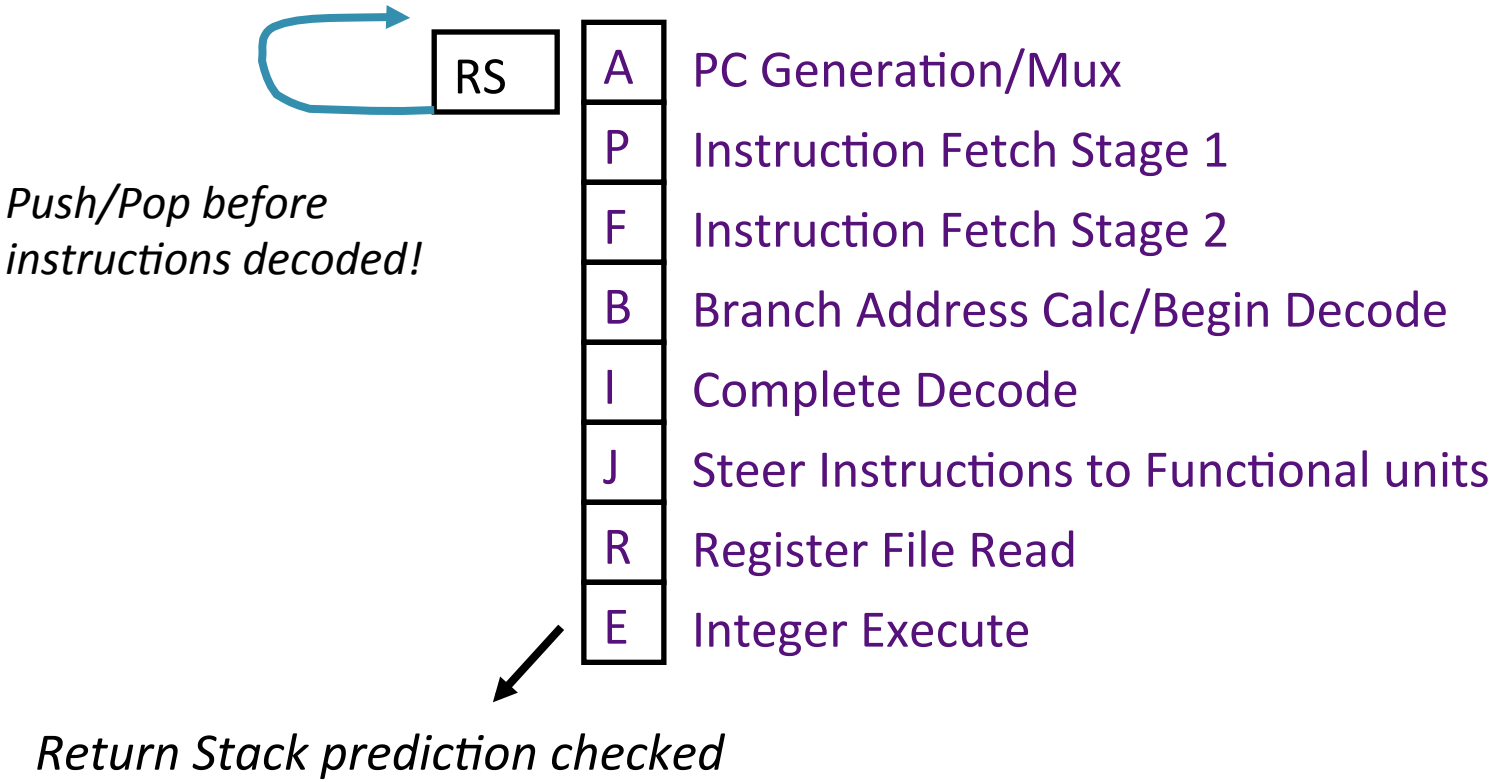
Return Stack in Pipeline

- How to use return stack (RS) in deep fetch pipeline?
- Only know if subroutine call/return at decode

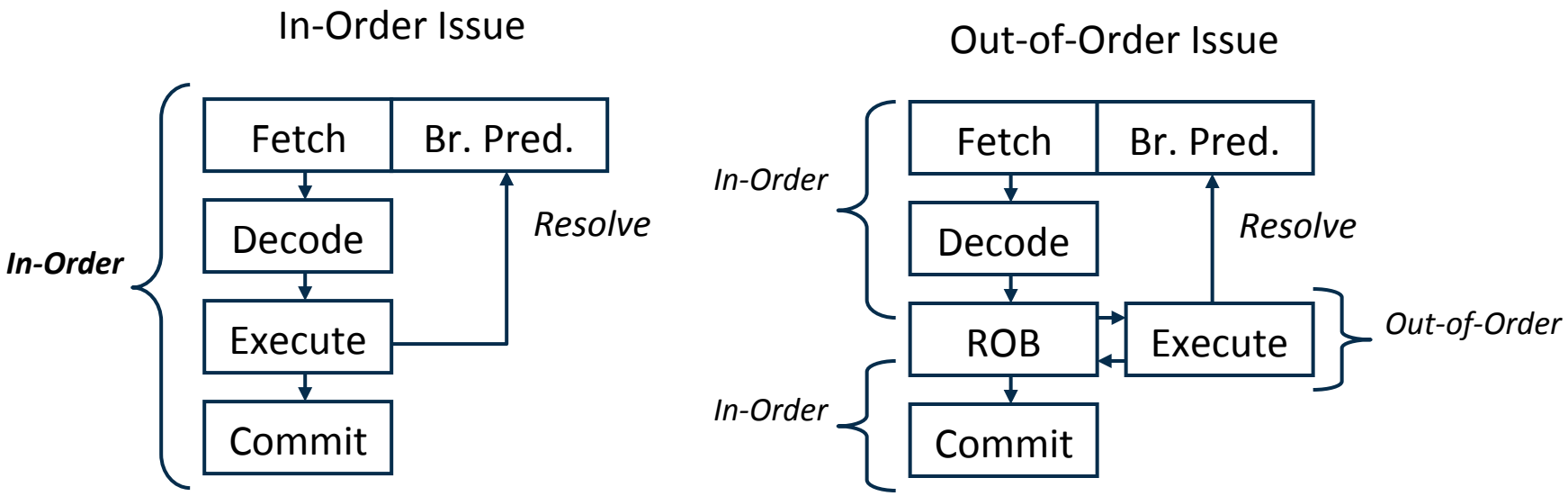


Return Stack in Pipeline

- Can remember whether PC is subroutine call/return using BTB-like structure
- Instead of target-PC, just store push/pop bit



In-Order vs. Out-of-Order Branch Prediction



- Speculative fetch but not speculative execution - branch resolves before later instructions complete
- Completed values held in bypass network until commit

- Speculative execution, with branches resolved after later instructions complete
- Completed values held in rename registers in ROB or unified physical register file until commit

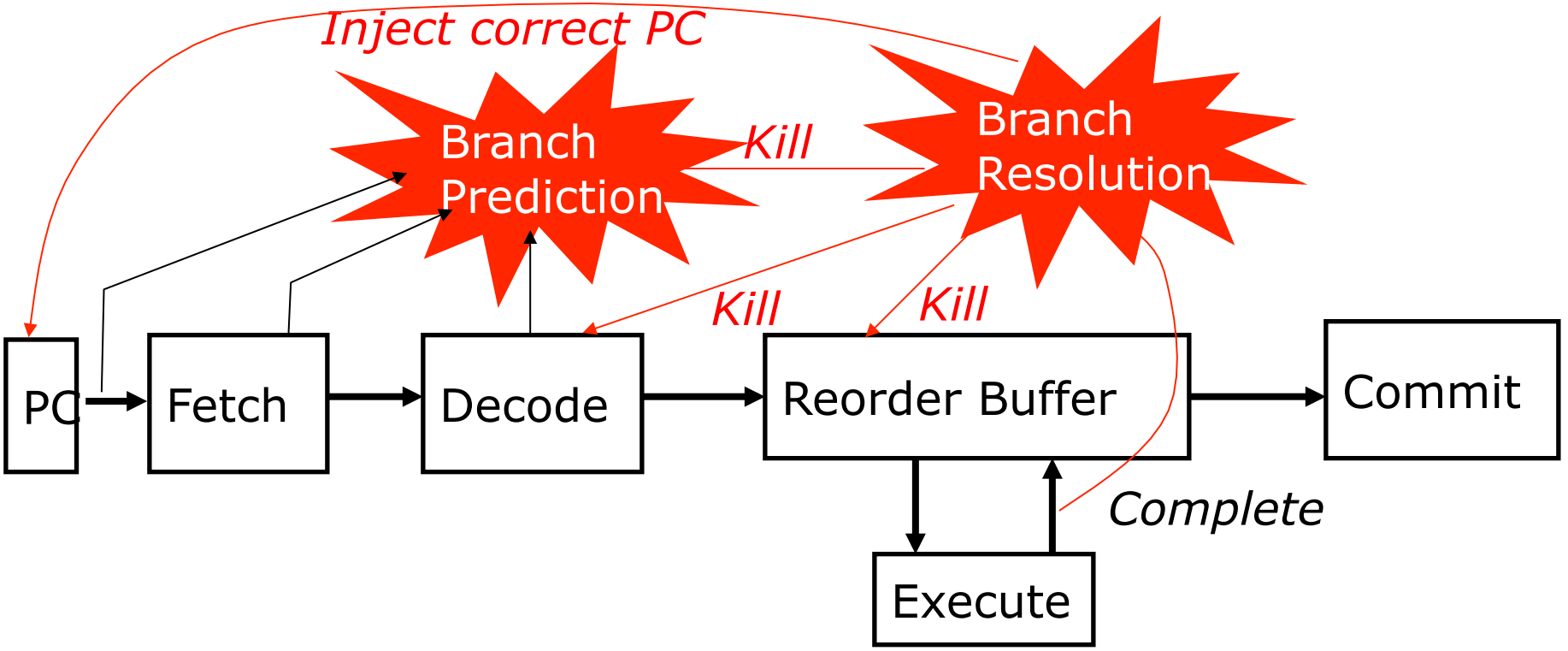
• Both styles of machine can use same branch predictors in front-end fetch pipeline, and both can execute multiple instructions per cycle

• Common to have 10-30 pipeline stages in either style of design

InO vs. OoO Mispredict Recovery

- In-order execution?
  - Design so 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
  - A simple solution would be to handle like precise traps
    - Problem?

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
- MIPS R10K uses four mask bits to tag instructions that are dependent on up to four speculative branches
- Mask bits cleared as branch resolves, and reused for next branch

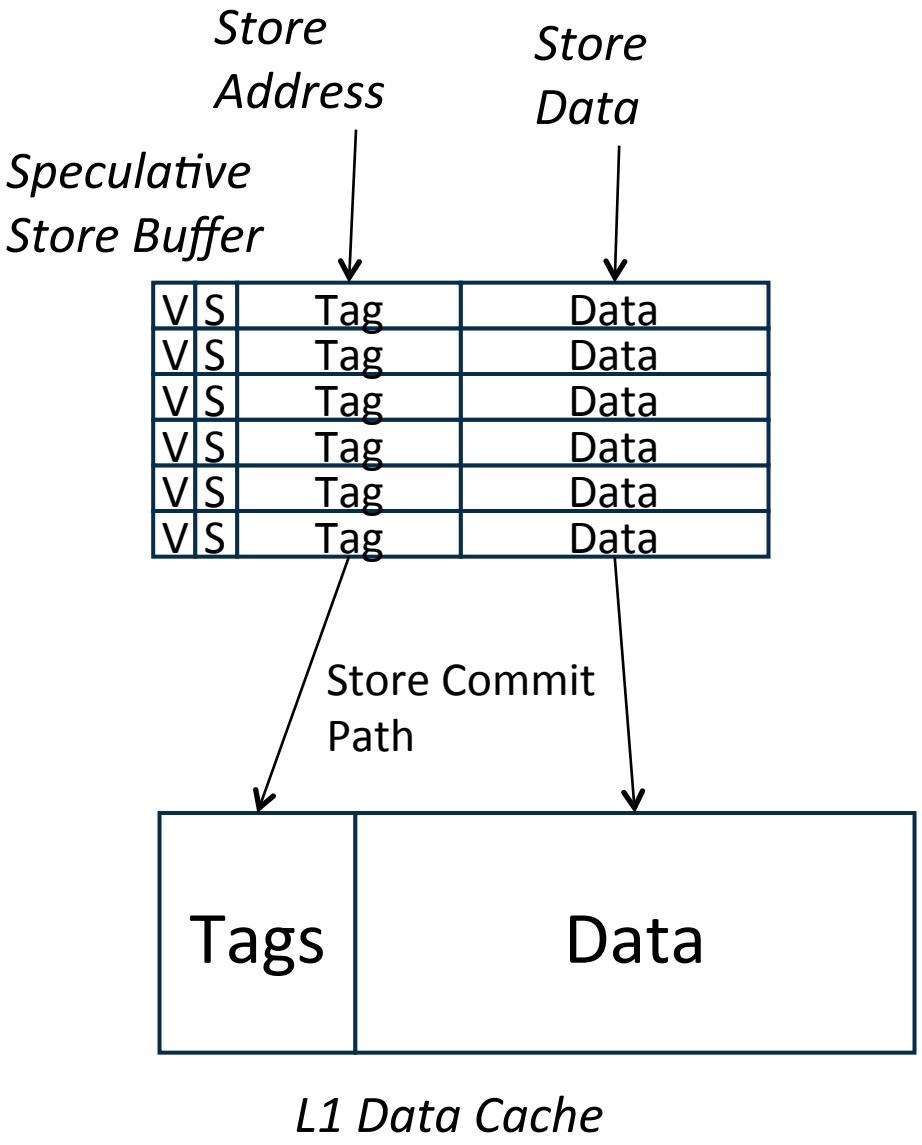
Rename Table Recovery

- Have to quickly recover rename table on branch mispredicts
- MIPS R10K only has four snapshots for each of four outstanding speculative branches
- Alpha 21264 has 80 snapshots, one per ROB instruction

Load-Store Queue Design

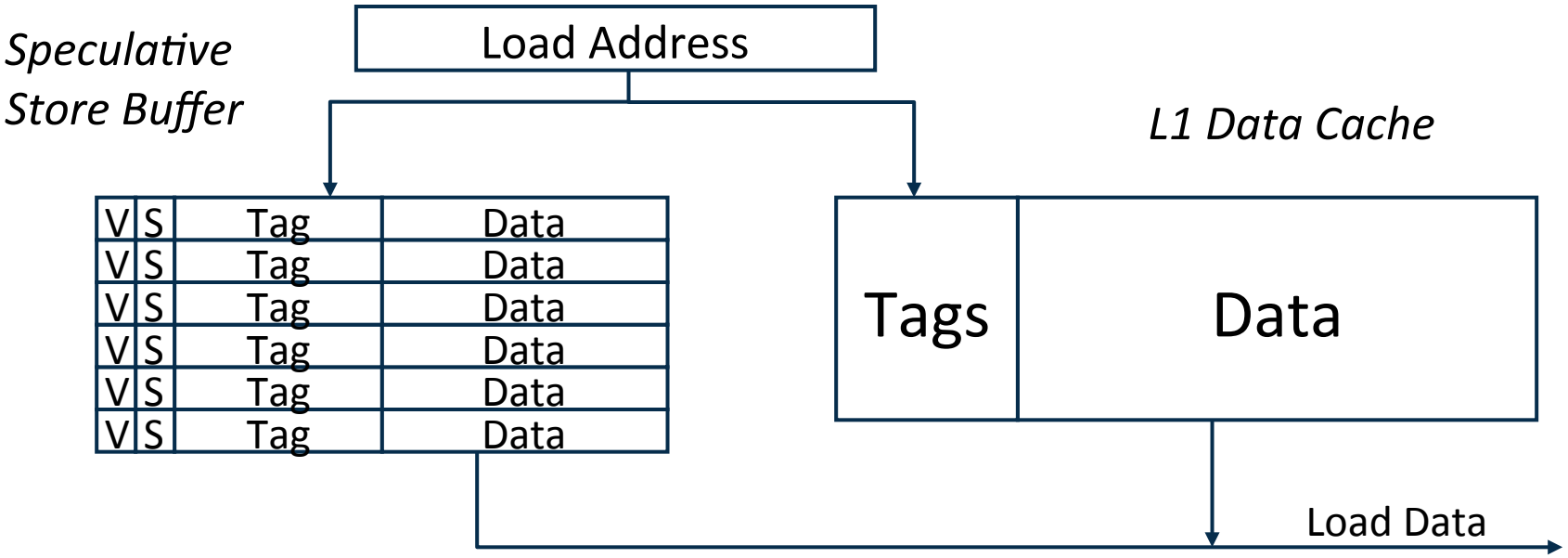
- After control hazards, data hazards through memory are probably next most important bottleneck to superscalar performance
- Modern superscalars use very sophisticated load-store reordering techniques to reduce effective memory latency by allowing loads to be speculatively issued

Speculative Store Buffer



- 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.
- During decode, store buffer slot allocated in program order
- Stores split into "store address" and "store data" micro-operations
- "Store address" execution writes tag
- "Store data" execution writes data
- Store commits when oldest instruction and both address and data available:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit

Load bypass from 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

Memory Dependencies

```
sd x1, (x2)
ld x3, (x4)
```

- 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

```
sd x1, (x2)
ld x3, (x4)
```

- Can execute load before store, if addresses known and **x4** != **x2**
- Each load address compared with addresses of all previous uncommitted stores
  - can use partial conservative check i.e., bottom 12 bits of address, to save hardware
- Don't execute load if any previous store address not known
- (MIPS R10K, 16-entry address queue)

Address Speculation

```
sd x1, (x2)
ld x3, (x4)
```

- Guess that **x4** != **x2**
- Execute load before store address known
- Need to hold all completed but uncommitted load/ store addresses in program order
- If subsequently find **x4==x2**, squash load and all following instructions

- => Large penalty for inaccurate address speculation

Memory Dependence Prediction (Alpha 21264)

```
sd x1, (x2)
ld x3, (x4)
```

- Guess that **x4** != **x2** and execute load before store
- If later find **x4**==**x2**, 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

Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)