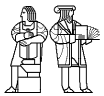
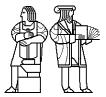


Advanced Superscalar Architectures

Krste Asanovic
Laboratory for Computer Science
Massachusetts Institute of Technology

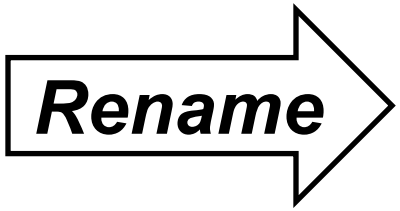


Physical Register Renaming

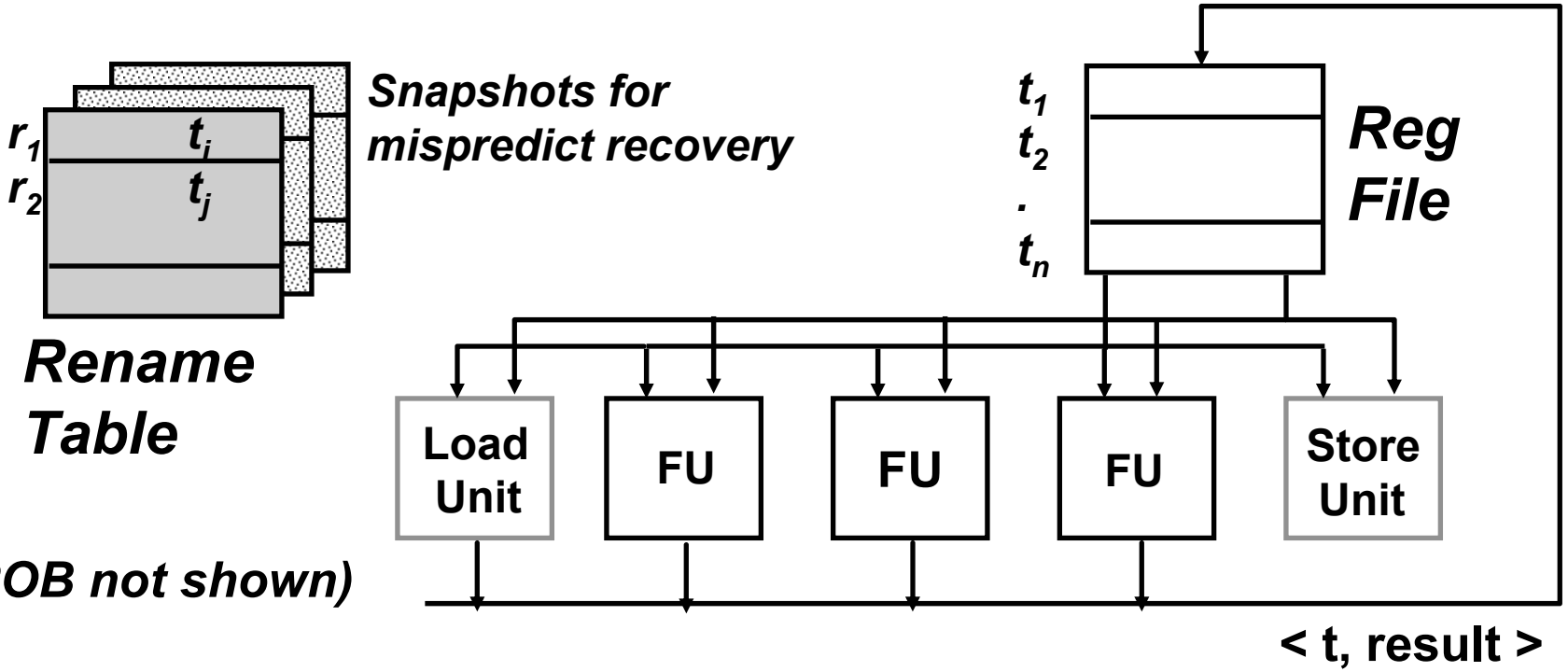
(single physical register file: MIPS R10K, Alpha 21264, Pentium-4)

- During decode, instructions allocated new physical destination register
- Source operands renamed to physical register with newest value
- Execution unit only sees physical register numbers

```
ld r1, (r3)              ld P1, (Px)
add r3, r1, #4           add P2, P1, #4
sub r6, r7, r9           sub P3, Py, Pz
add r3, r3, r6           add P4, P2, P3
ld r6, (r1)              ld P5, (P1)
add r6, r6, r3           add P6, P5, P4
st r6, (r1)              st P6, (P1)
ld r6, (r11)             ld P7, (Pw)
```

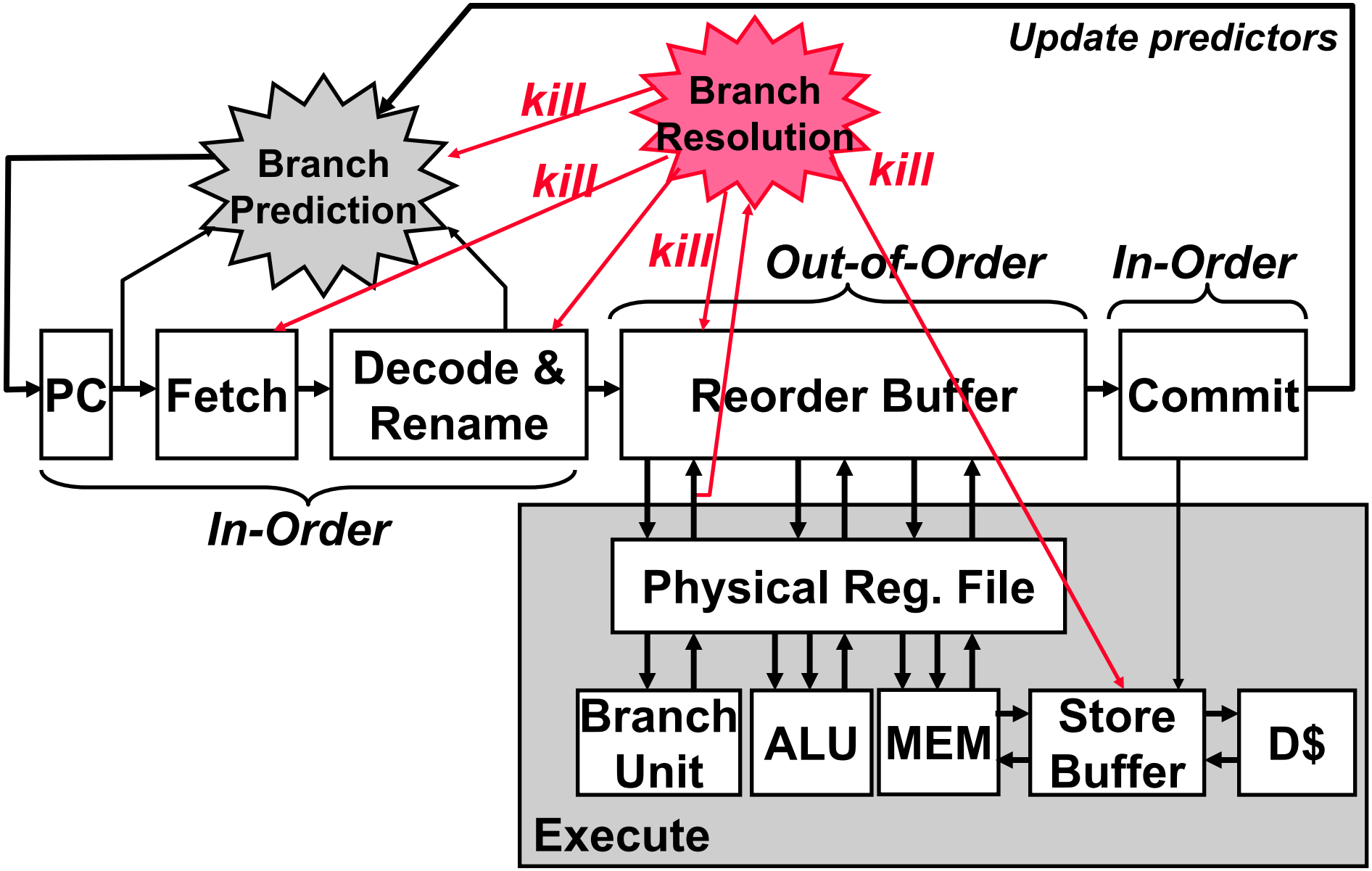


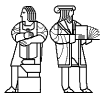
Physical Register File



- 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 and Out-of-Order Execution

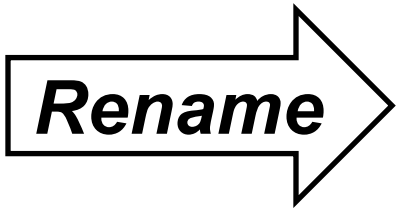




Lifetime of Physical Registers

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

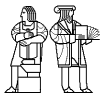
```
ld r1, (r3)
add r3, r1, #4
sub r6, r7, r9
add r3, r3, r6
ld r6, (r1)
add r6, r6, r3
st r6, (r1)
ld r6, (r11)
```




```
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

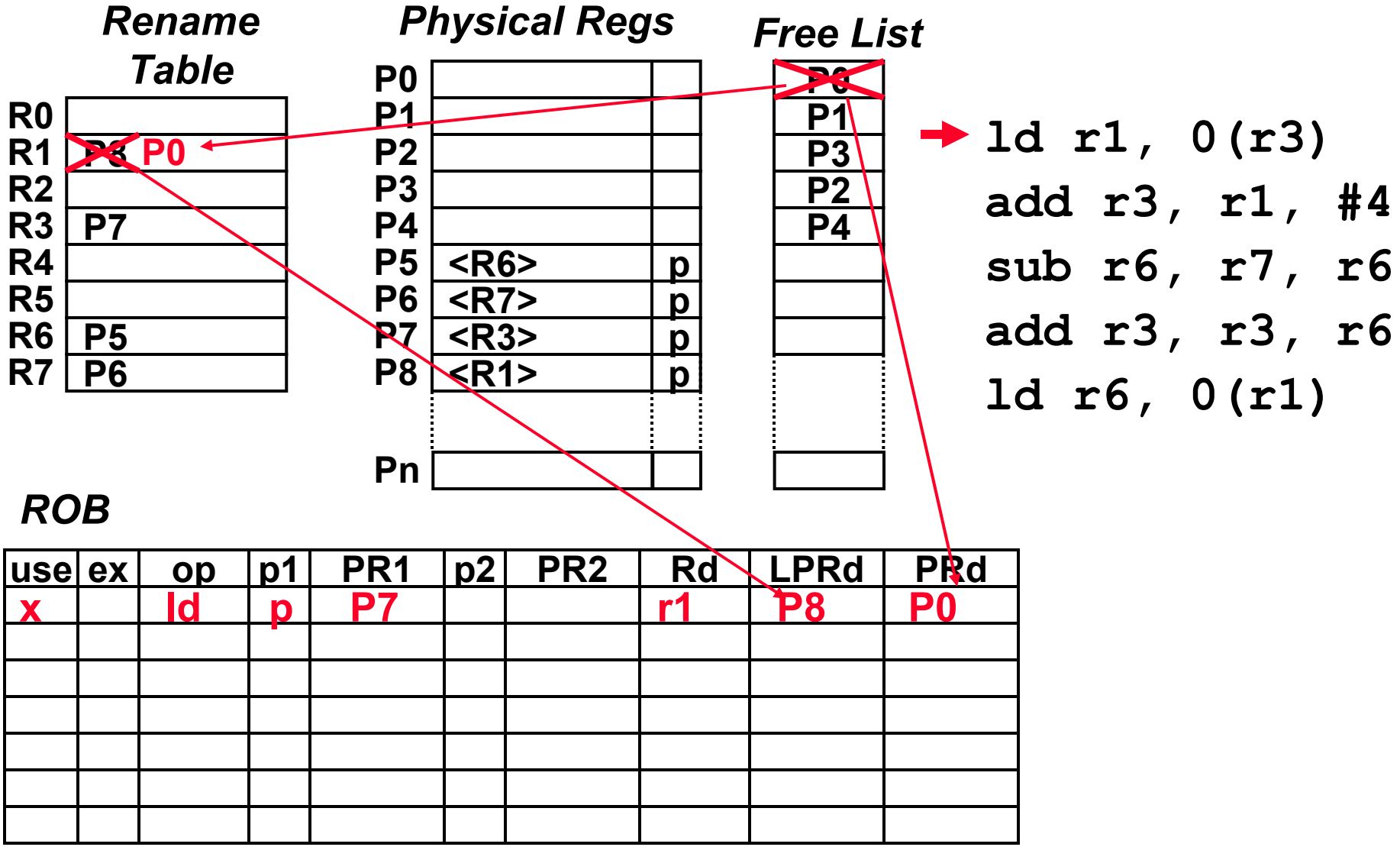
```
ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)
```

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd

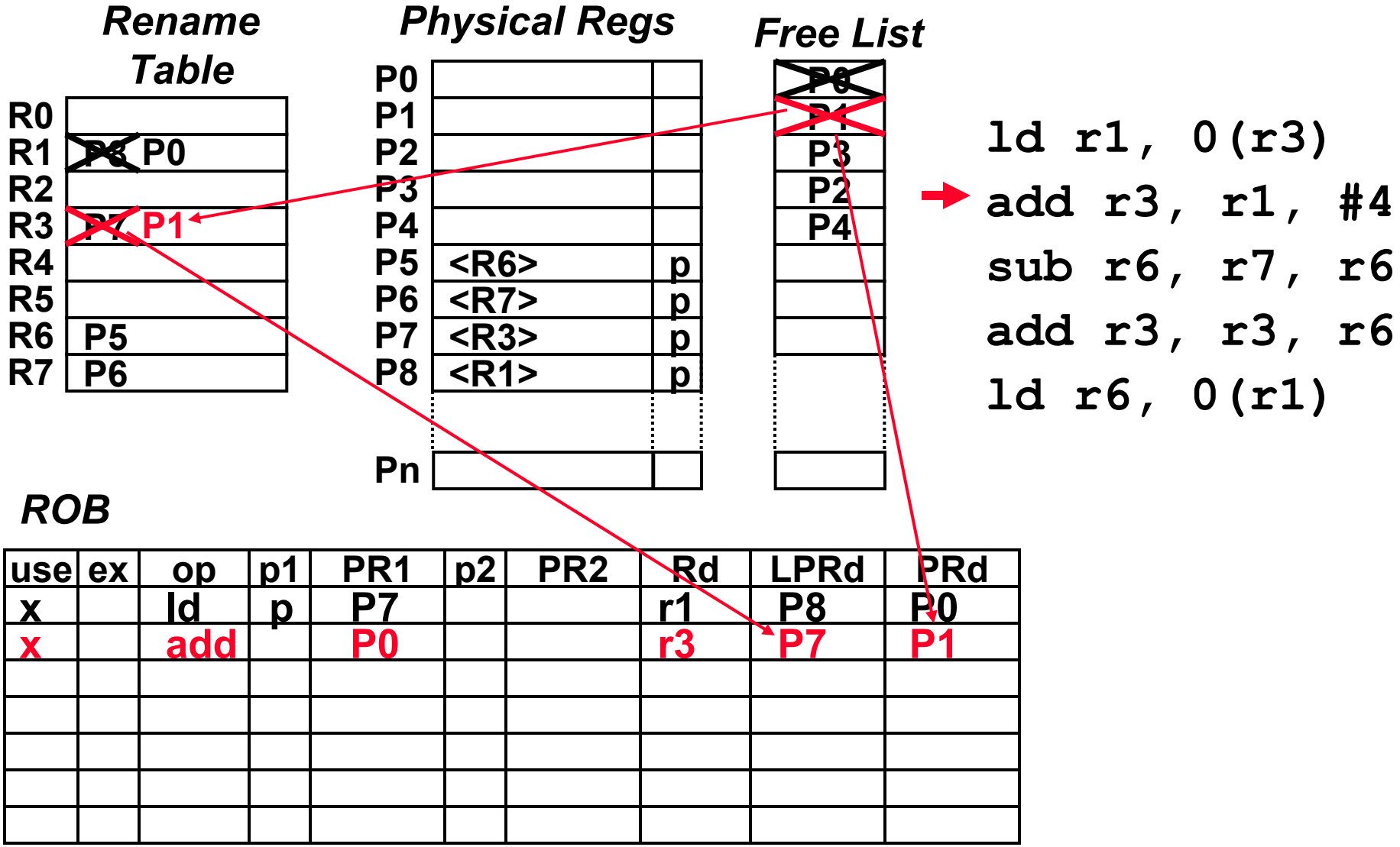
(LPRd requires third read port on Rename Table for each instruction)

Physical Register Management



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

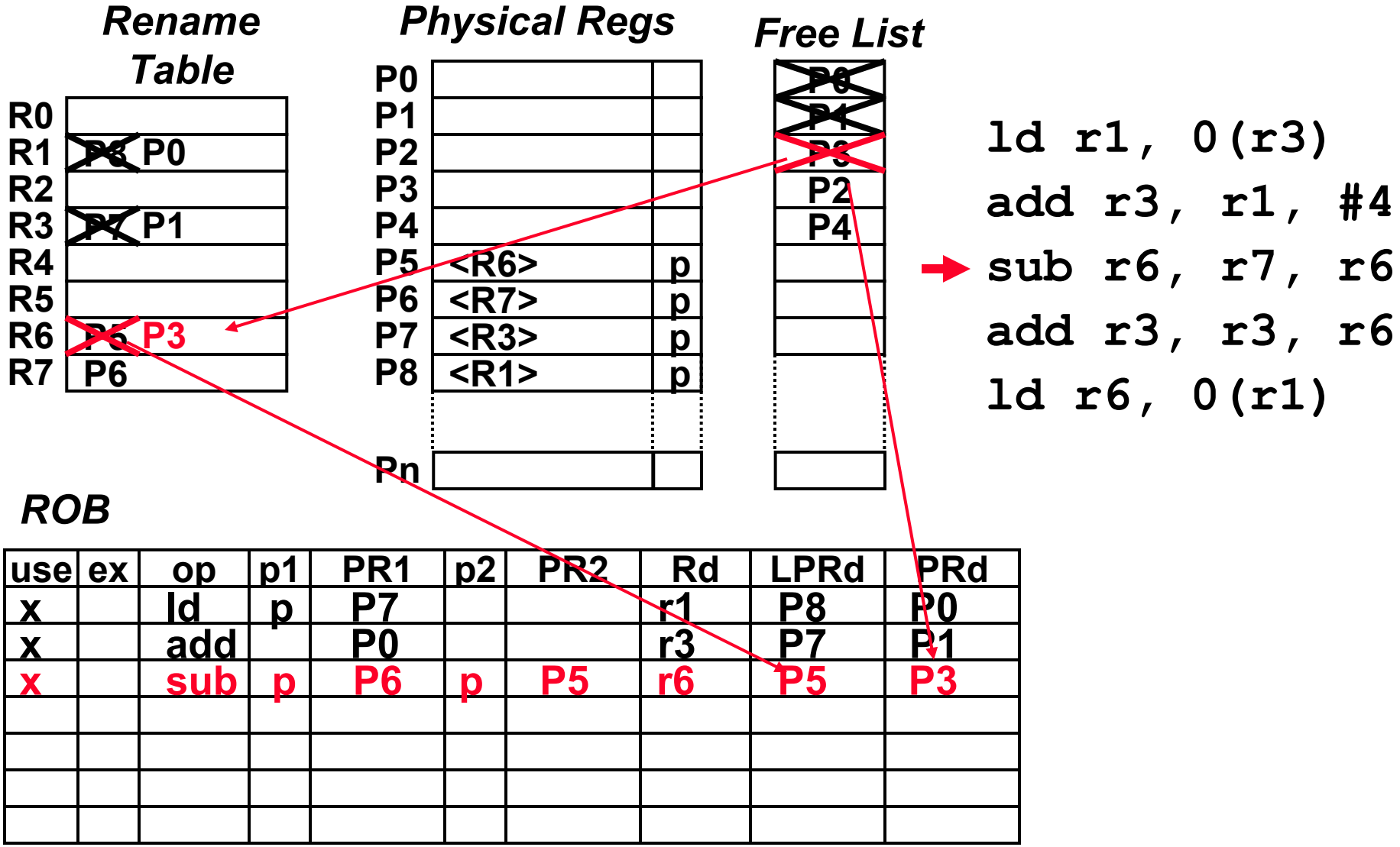
Physical Register Management



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

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

Physical Register Management

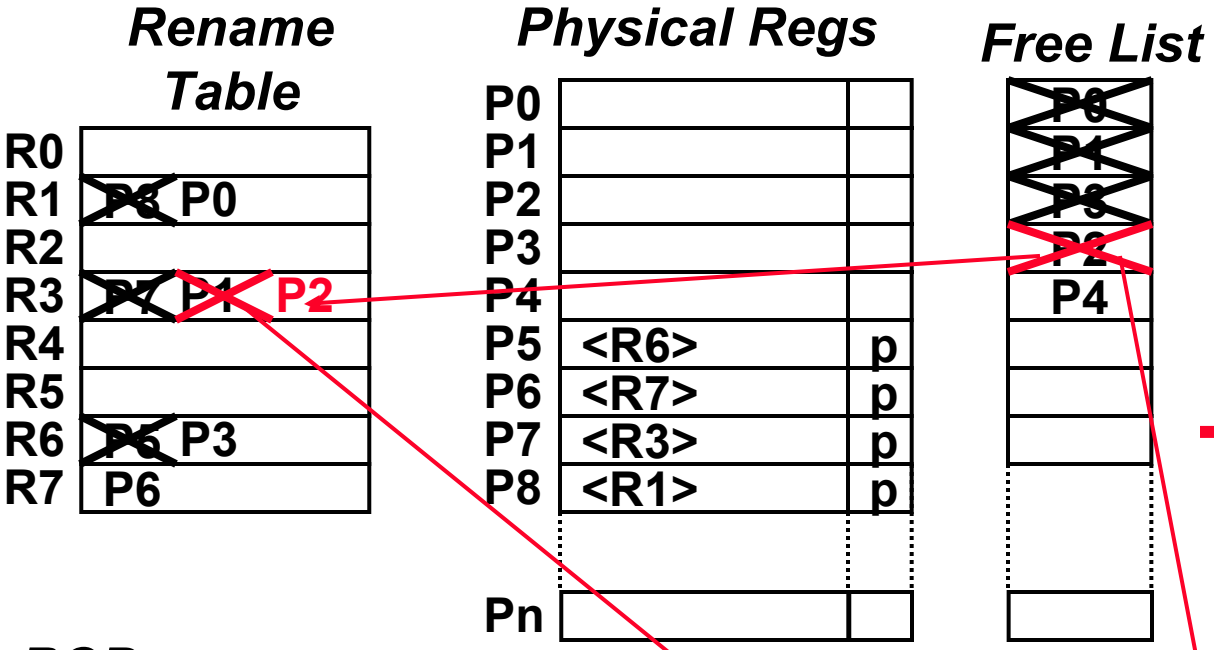


ROB

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

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

Physical Register Management



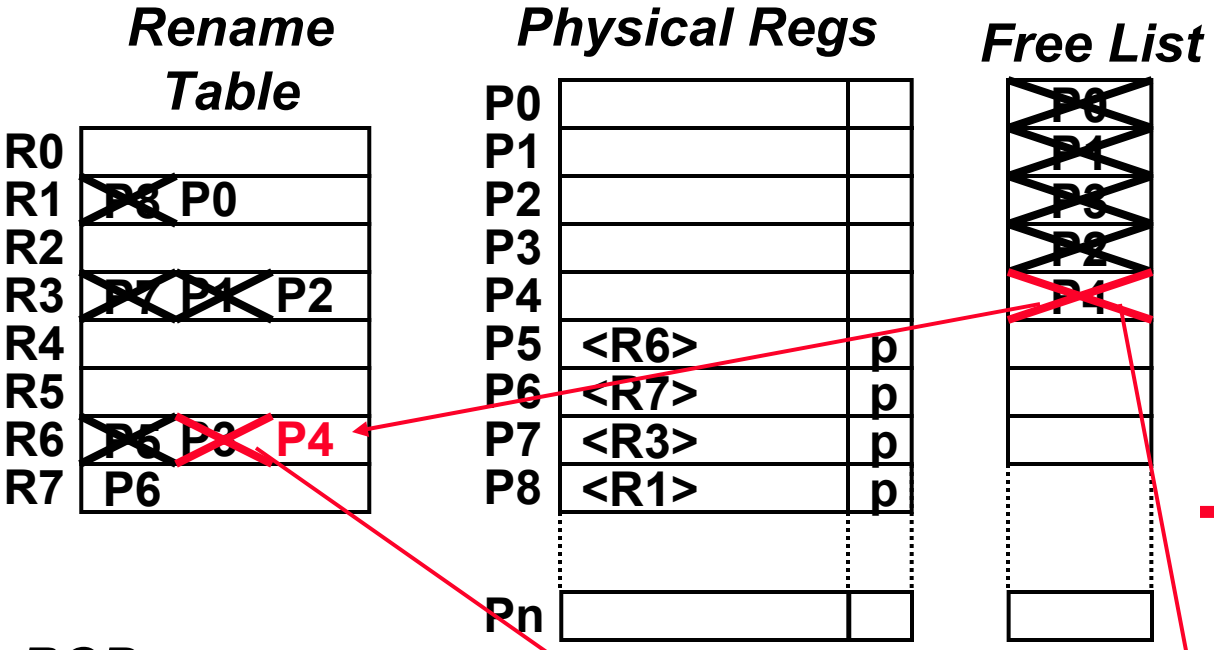

```
ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)
```

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
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

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

Physical Register Management



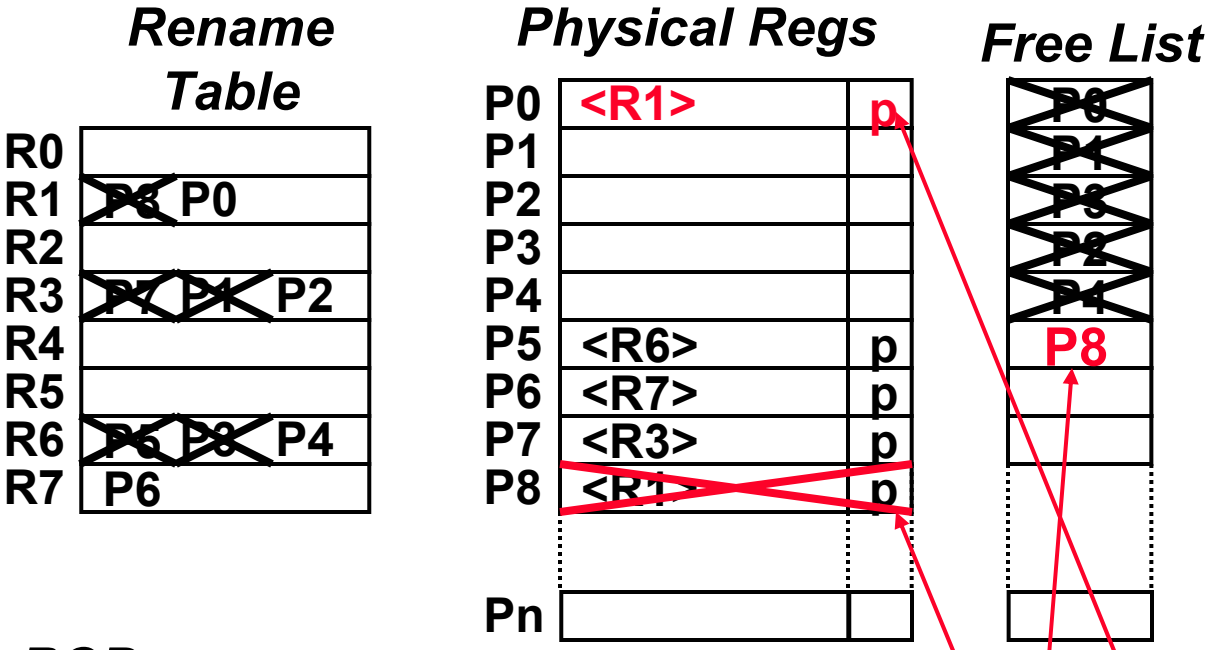

```
ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)
```

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
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

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

Physical Register Management




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

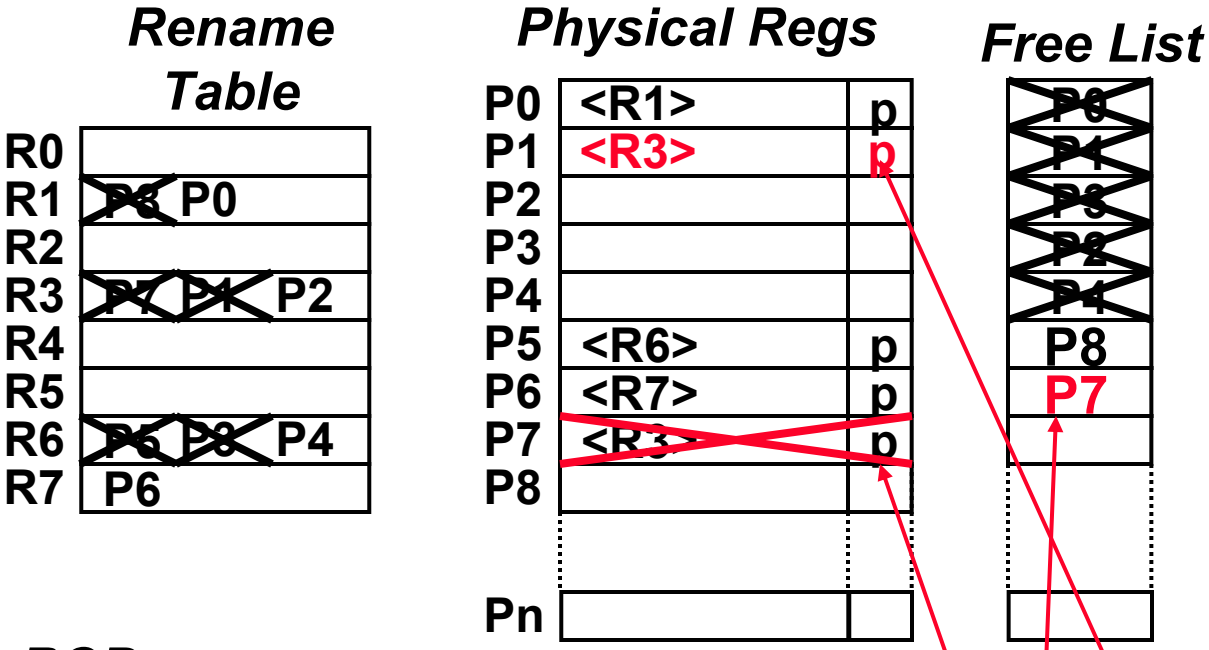
ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
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

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

Physical Register Management




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

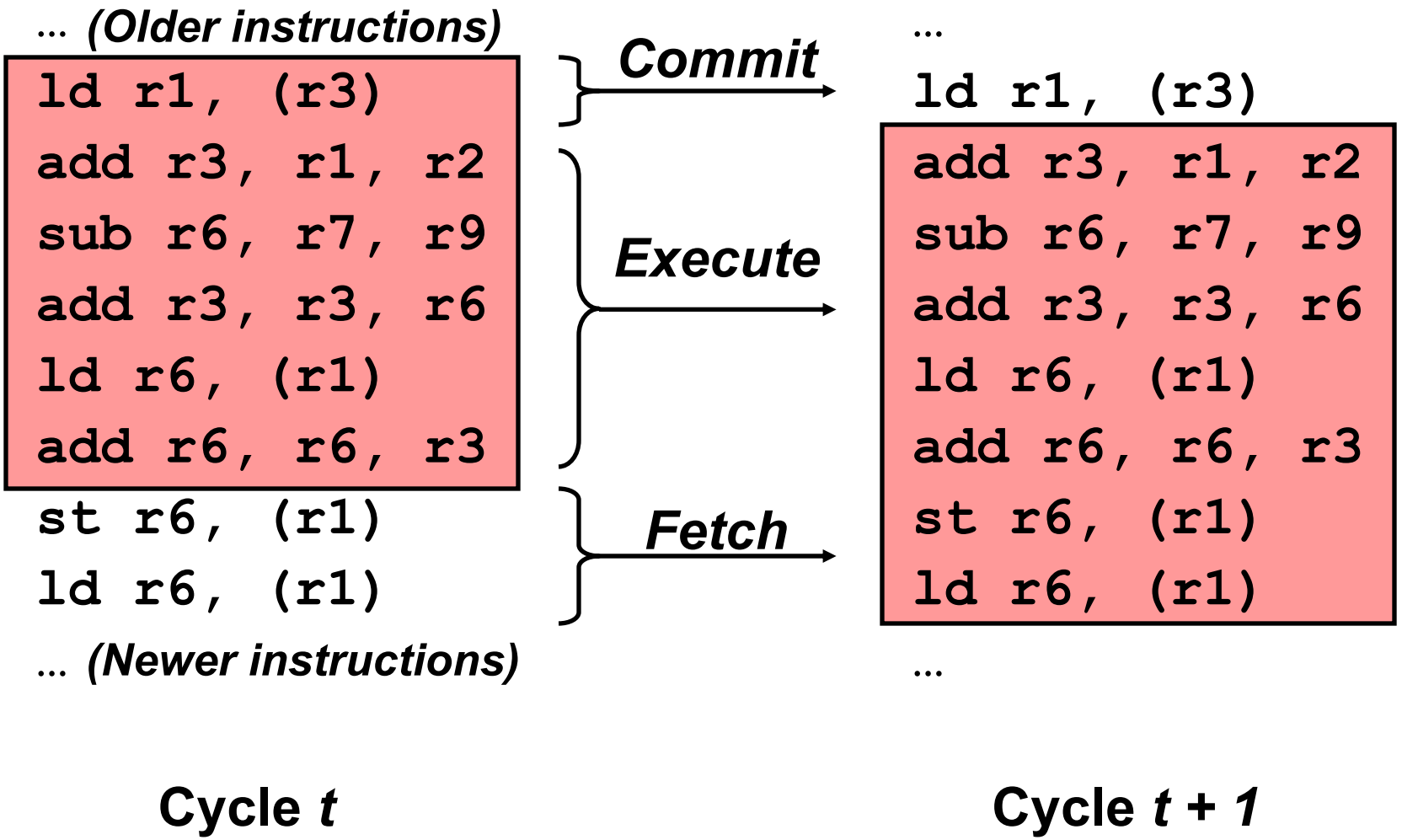
ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
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

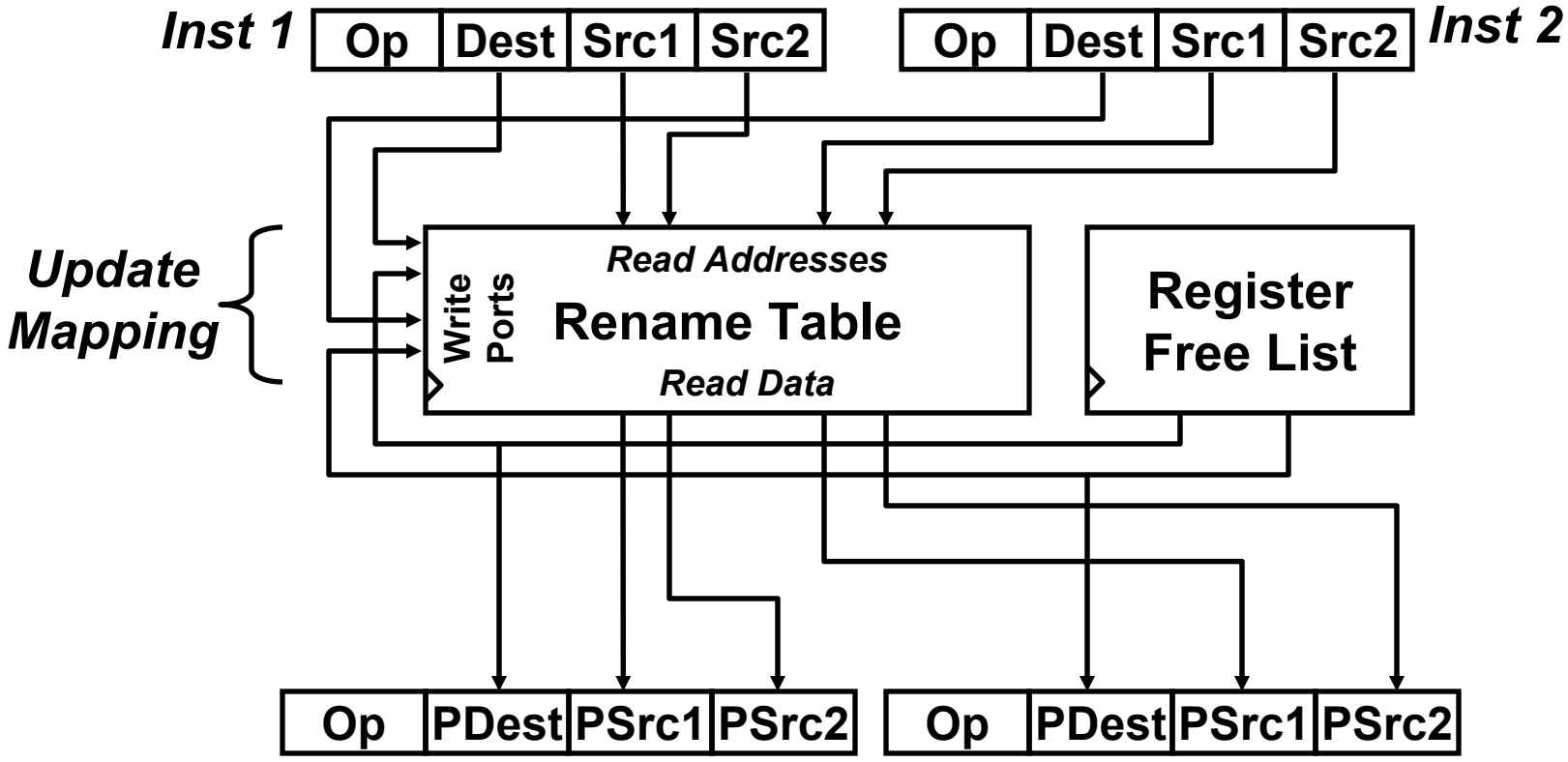
(LPRd requires third read port on Rename Table for each instruction)

Reorder Buffer Holds Active Instruction Window



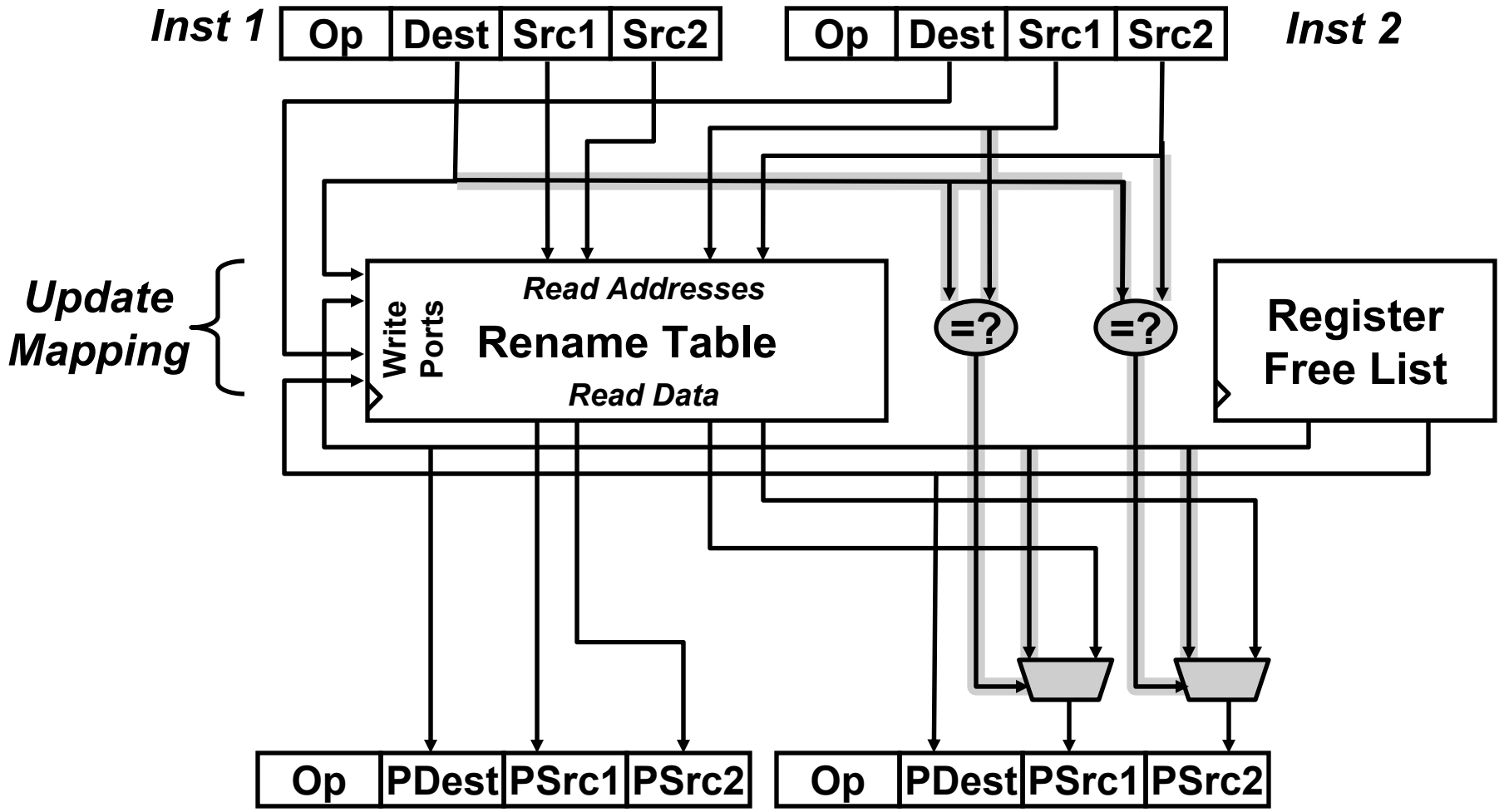
Superscalar Register Renaming

- During decode, instructions allocated new physical destination register
- Source operands renamed to physical register with newest value
- Execution unit only sees physical register numbers



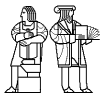
Does this work?

Superscalar Register Renaming



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

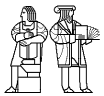
(MIPS R10K renames 4 serially-RAW-dependent insts/cycle)



Memory Dependencies

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

When can we execute the load?



Speculative Loads / Stores

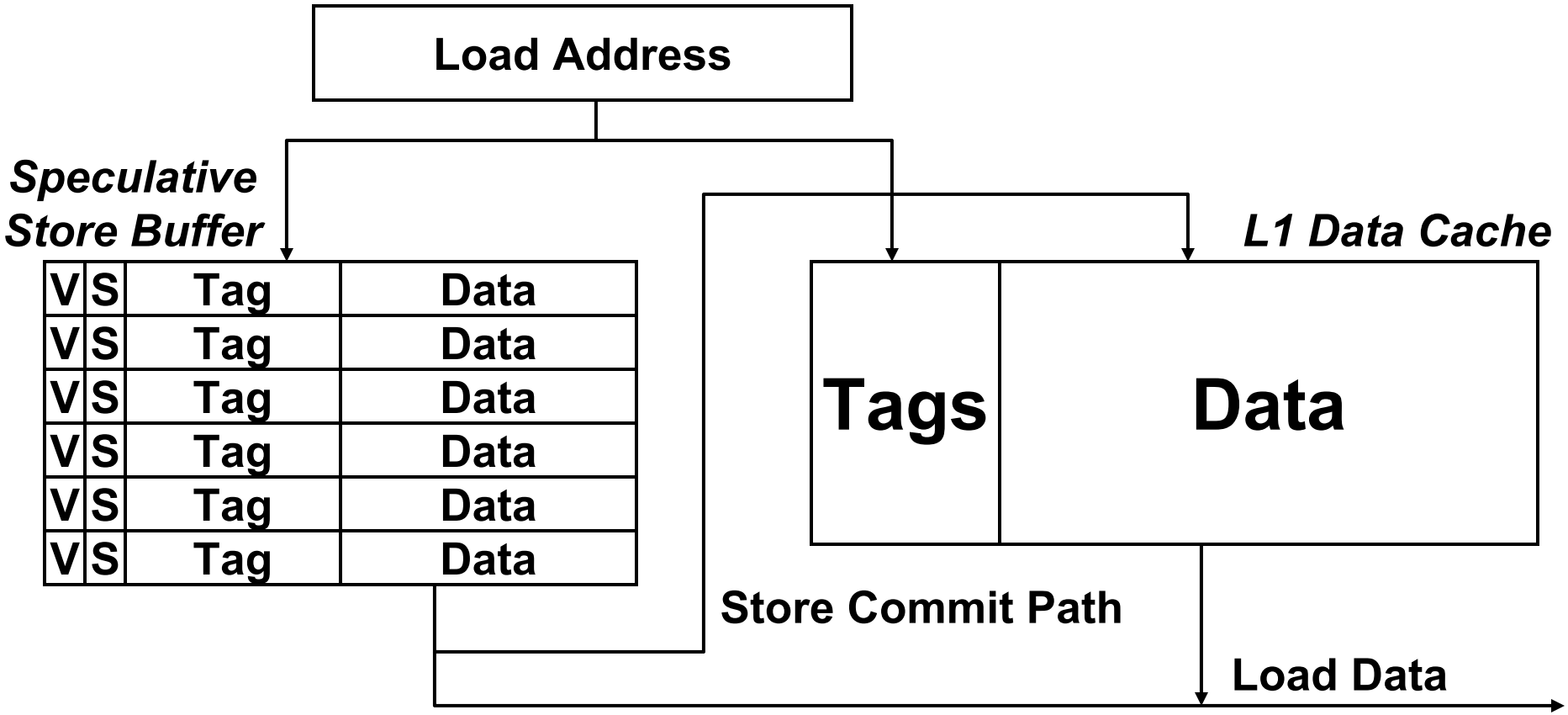
Just like register updates, stores should not modify the memory until after the instruction is committed

$\Rightarrow$ store buffer entry must carry a speculation bit and the tag of the corresponding store instruction

- If the instruction is committed, the speculation bit of the corresponding store buffer entry is cleared, and store is written to cache
- If the instruction is killed, the corresponding store buffer entry is freed

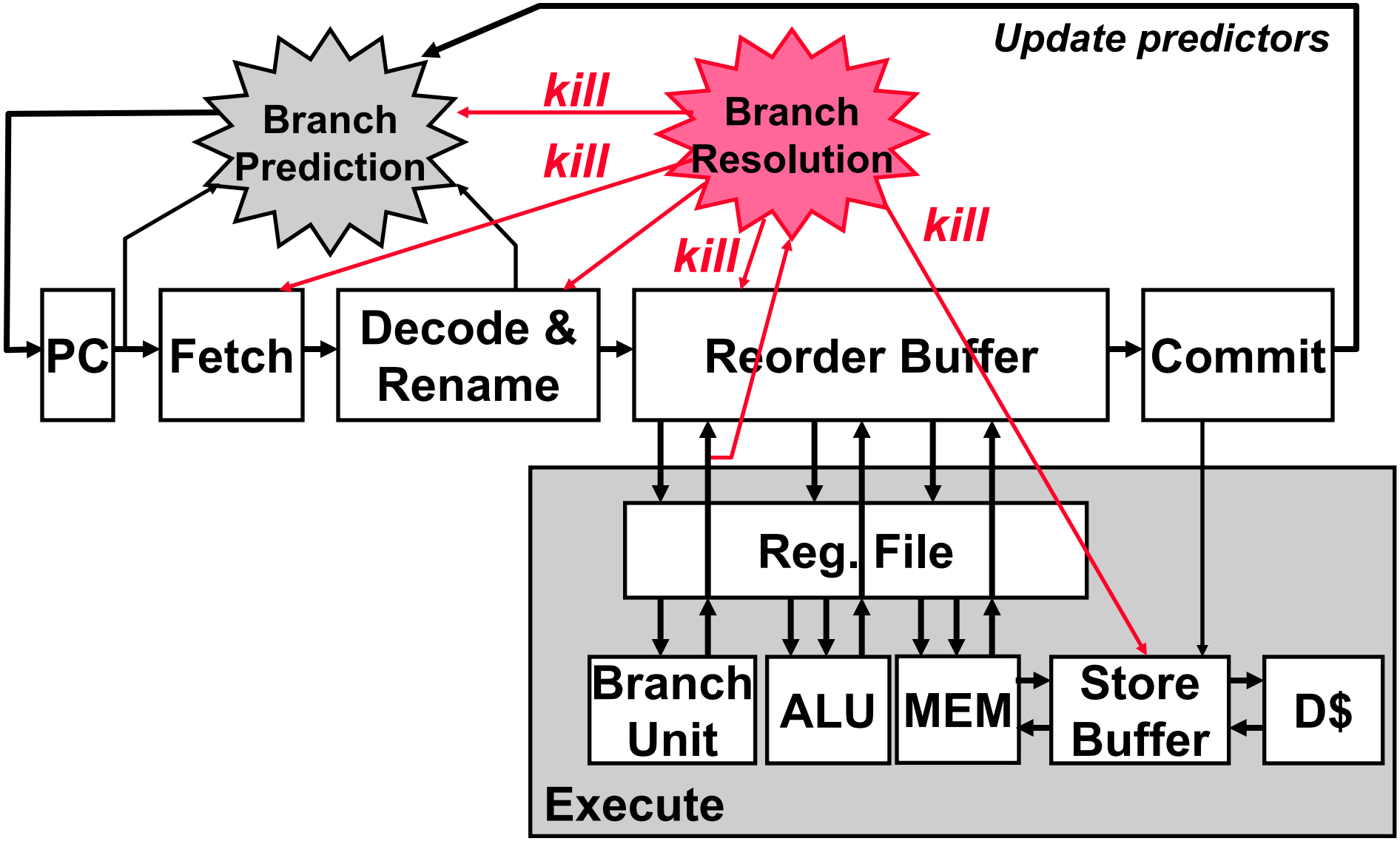
Loads work normally -- “older” store buffer entries needs to be searched before accessing the memory or the cache

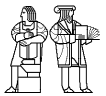
Load Path



- Hit in speculative store buffer has priority over hit in data cache
- Hit to newer store has priority over hits to older stores in speculative store buffer

Datapath: Branch Prediction and Speculative Execution





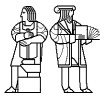
In-Order Memory Queue

- 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

- Stores held in store buffer until commit

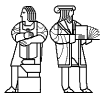


Conservative Out-of-Order Load Execution

```
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)

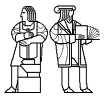


Address Speculation

```
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

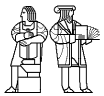
=> Large penalty for inaccurate address speculation



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

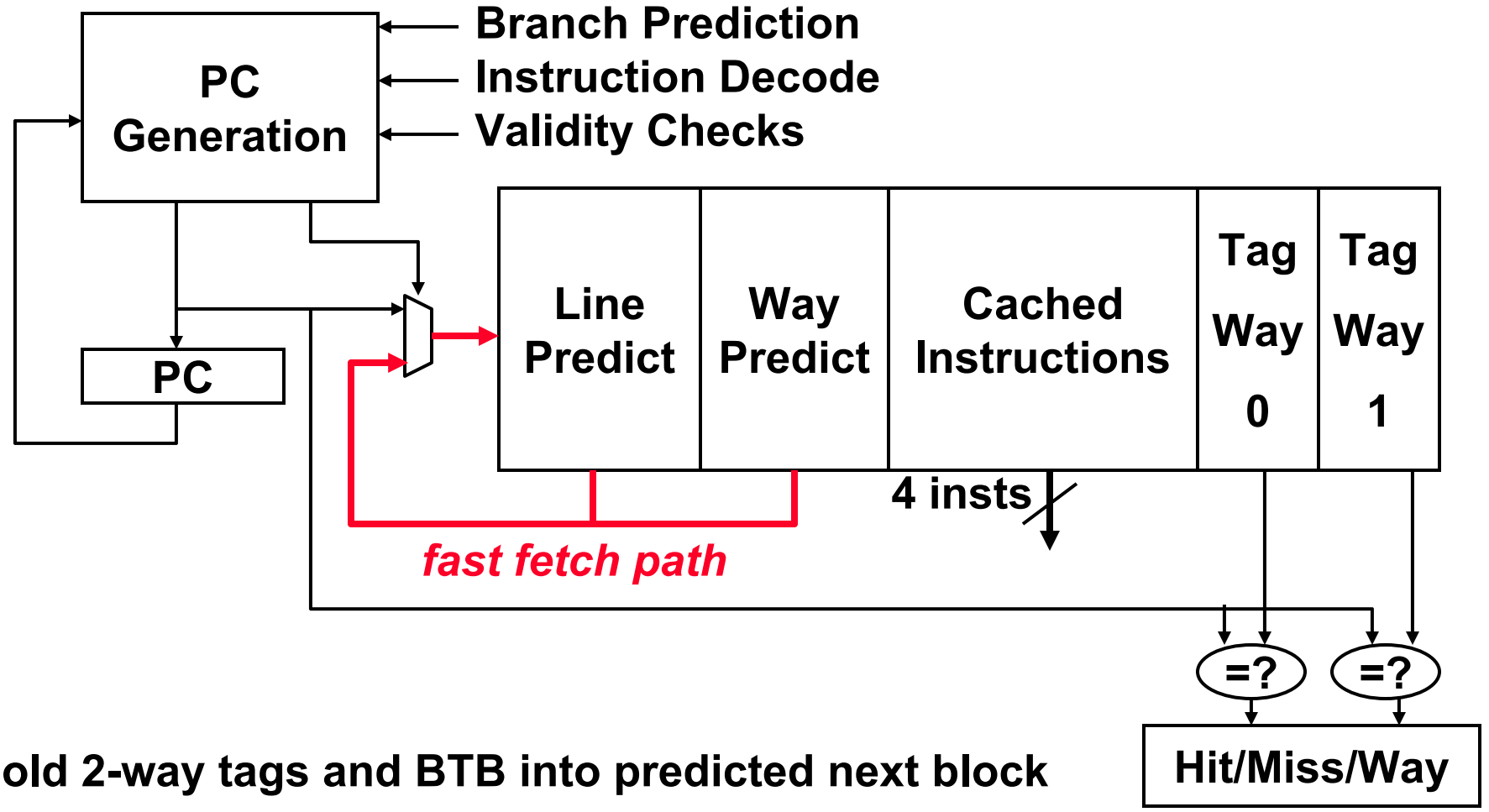


Improving Instruction Fetch

Performance of speculative out-of-order machines often limited by instruction fetch bandwidth

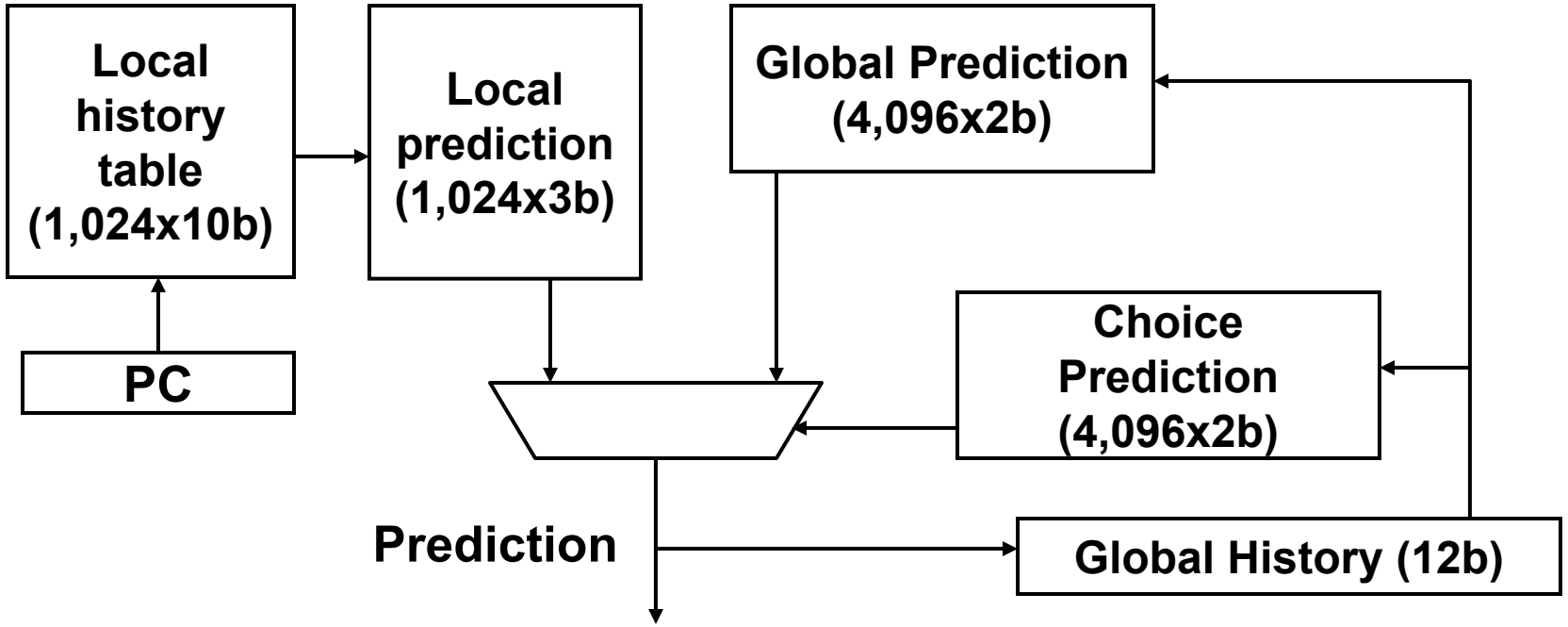
- speculative execution can fetch 2-3x more instructions than are committed
- mispredict penalties dominated by time to refill instruction window
- *taken branches* are particularly troublesome

Increasing Taken Branch Bandwidth
(Alpha 21264 I-Cache)

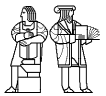


- Fold 2-way tags and BTB into predicted next block
- Take tag checks, inst. decode, branch predict out of loop
- Raw RAM speed on critical loop (1 cycle at ~1 GHz)
- 2-bit hysteresis counter per block prevents overtraining

Tournament Branch Predictor
(Alpha 21264)



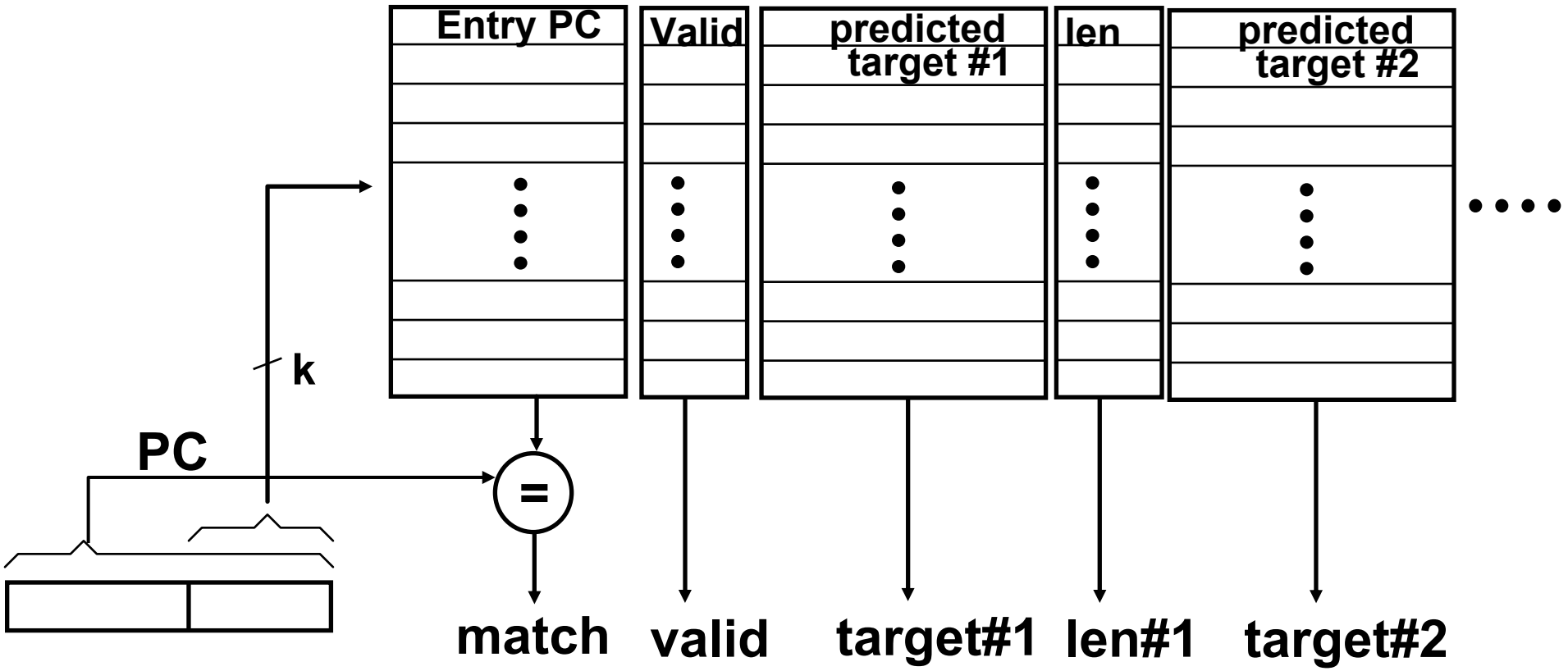
- Choice predictor learns whether best to use local or global branch history in predicting next branch
- Global history is speculatively updated but restored on mispredict
- Claim 90-100% success on range of applications



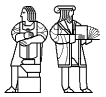
Taken Branch Limit

- Integer codes have a taken branch every 6-9 instructions
- To avoid fetch bottleneck, must execute multiple taken branches per cycle when increasing performance
- This implies:
  - predicting multiple branches per cycle
  - fetching multiple non-contiguous blocks per cycle

Branch Address Cache
(Yeh, Marr, Patt)



Extend BTB to return multiple branch predictions per cycle



Fetching Multiple Basic Blocks

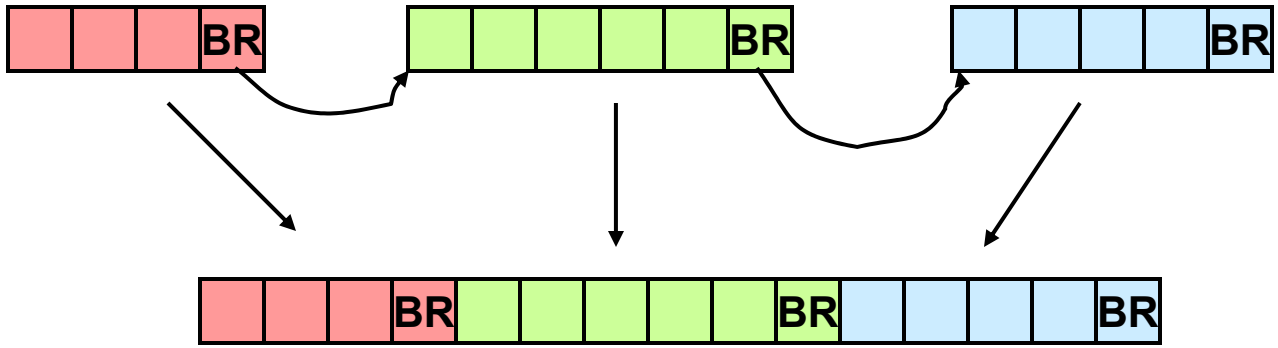
Requires either

- multiported cache: expensive
- interleaving: bank conflicts will occur

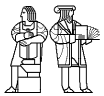
Merging multiple blocks to feed to decoders adds latency increasing mispredict penalty and reducing branch throughput

Trace Cache

Key Idea: Pack multiple non-contiguous basic blocks into one contiguous trace cache line



- Single fetch brings in multiple basic blocks
- Trace cache indexed by start address *and* next *n* branch predictions
- Used in Intel Pentium-4 processor to hold decoded uops



MIPS R10000 (1995)

- 0.35μm CMOS, 4 metal layers
- Four instructions per cycle
- Out-of-order execution
- Register renaming
- Speculative execution past 4 branches
- On-chip 32KB/32KB split I/D cache, 2-way set-associative
- Off-chip L2 cache
- Non-blocking caches

Compare with simple 5-stage pipeline (R5K series)

- ~1.6x performance SPECint95
- ~5x CPU logic area
- ~10x design effort