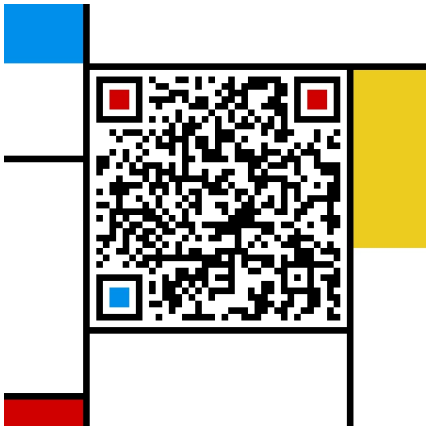
计算机组成与系统结构
Computer Organization & System Architecture

Huang Kejie (黄科杰) 百人计划研究员
Office: 玉泉校区老生仪楼 304
Email address: huangkejie@zju.edu.cn
HP: 17706443800



Renaming & Out-of-order Issue

An example

Renaming table

	p	data
f1		
f2		v1
f3		
f4		t2
f5		
f6		t3
f7		
f8		v4

v1

data / t_i

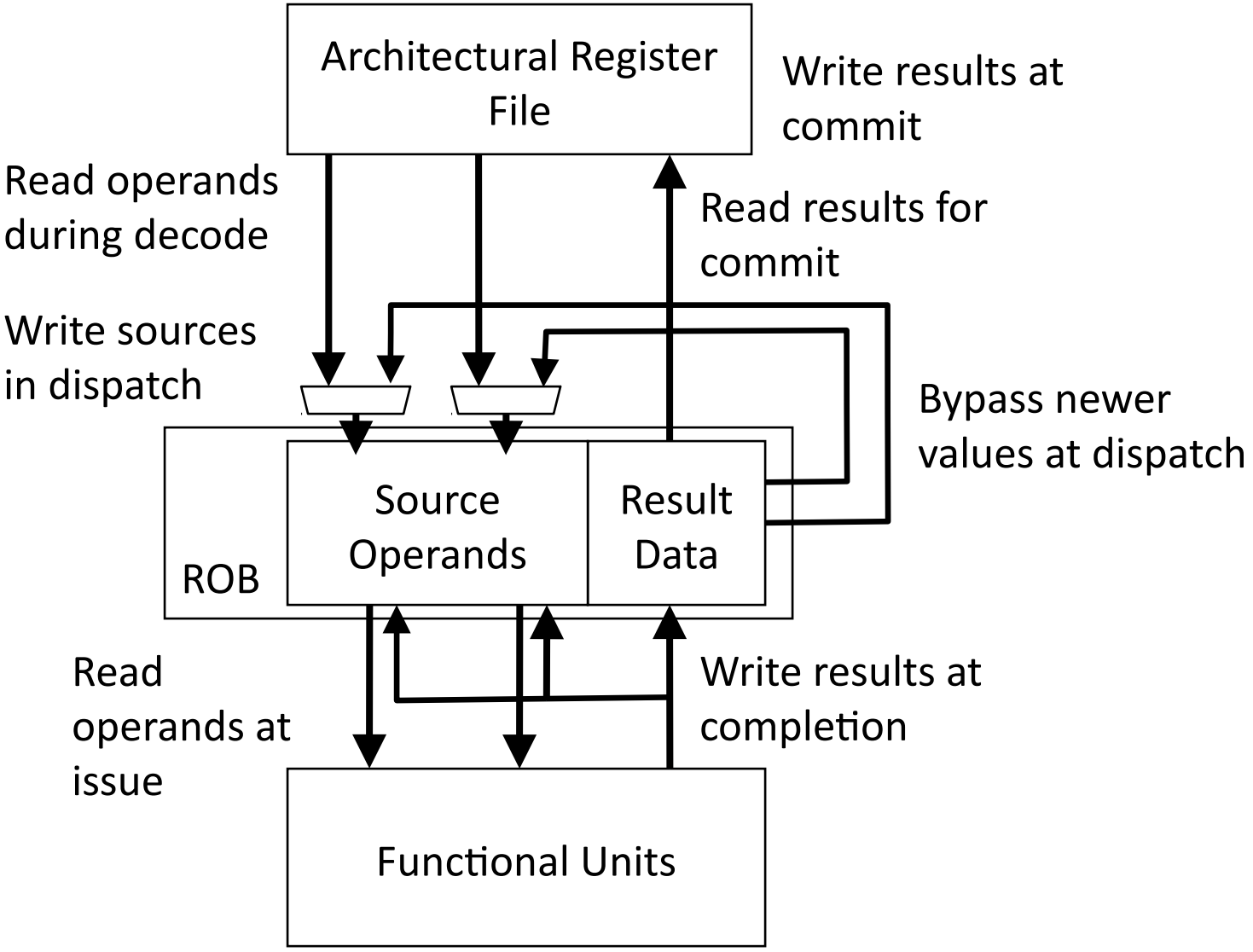
Reorder buffer

	Ins#	use	exec	op	p1	src1	p2	src2	
	1	0	0	LD					t_1
	2	0	0	LD					t_2
	3	1	0	MUL	0	t2	1	v1	t_3
	4	0	0	SUB	1	v1	1	v1	t_4
	5	1	0	DIV	1	v1	0	t4	t_5
									.
									.

```
1 FLD      f2,  34(x2)
2 FLD      f4,  45(x3)
3 FMULT.D  f6,  f4,  f2
4 FSUB.D   f8,  f2,  f2
5 FDIV.D   f4,  f2,  f8
6 FADD.D   f10, f6,  f4
```

- *When are tags in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

Data Movement in Data-in-ROB Design



OoO Design Choices

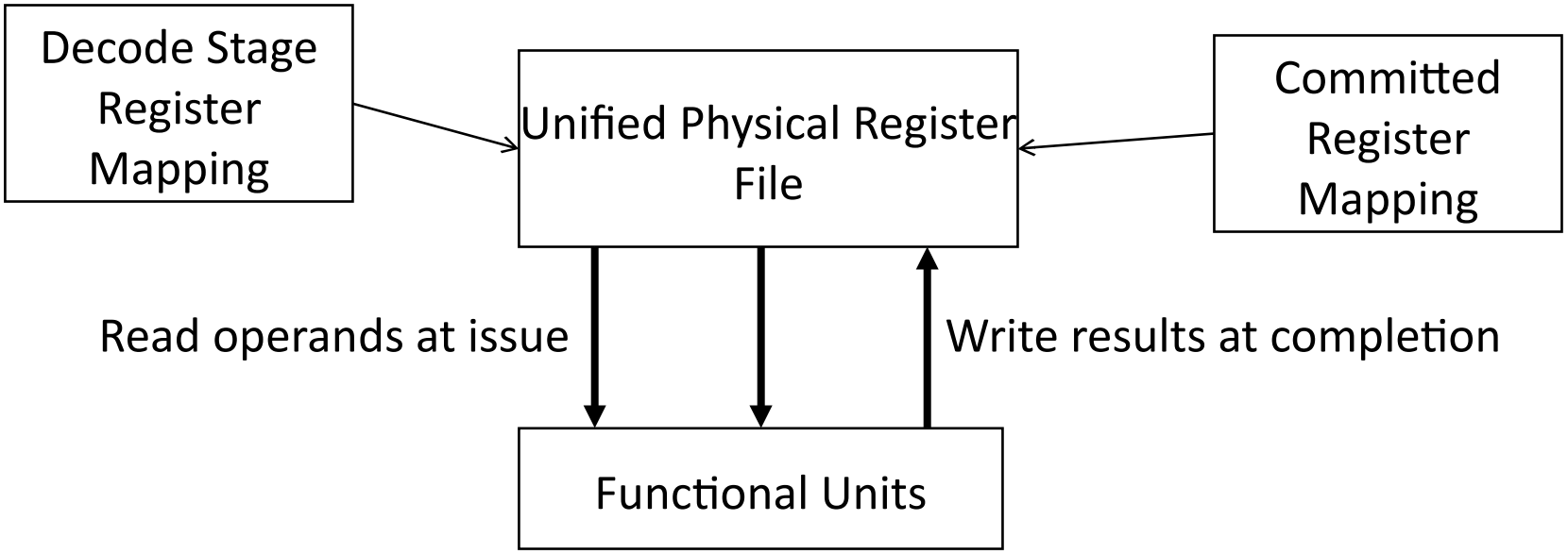
- Where are reservation stations?
  - Part of reorder buffer, or in separate issue window?
  - Distributed by functional units, or centralized?

- How is register renaming performed?
  - Tags and data held in reservation stations, with separate architectural register file
  - Tags only in reservation stations, data held in unified physical register file

Unified Physical Register File

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

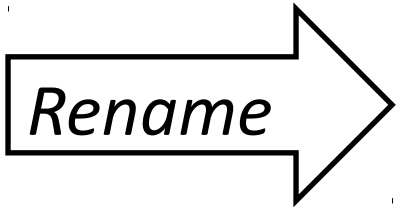
- Rename all architectural registers into a single physical register file during decode, no register values read
- Functional units read and write from single unified register file holding committed and temporary registers in execute
- Commit only updates mapping of architectural register to physical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

Rename Table

x0	
x1	P8
x2	
x3	P7
x4	
x5	
x6	P5
x7	P6

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5	<x6>	p
P6	<x7>	p
P7	<x3>	p
P8	<x1>	p
Pn		

Free List

P0
P1
P3
P2
P4

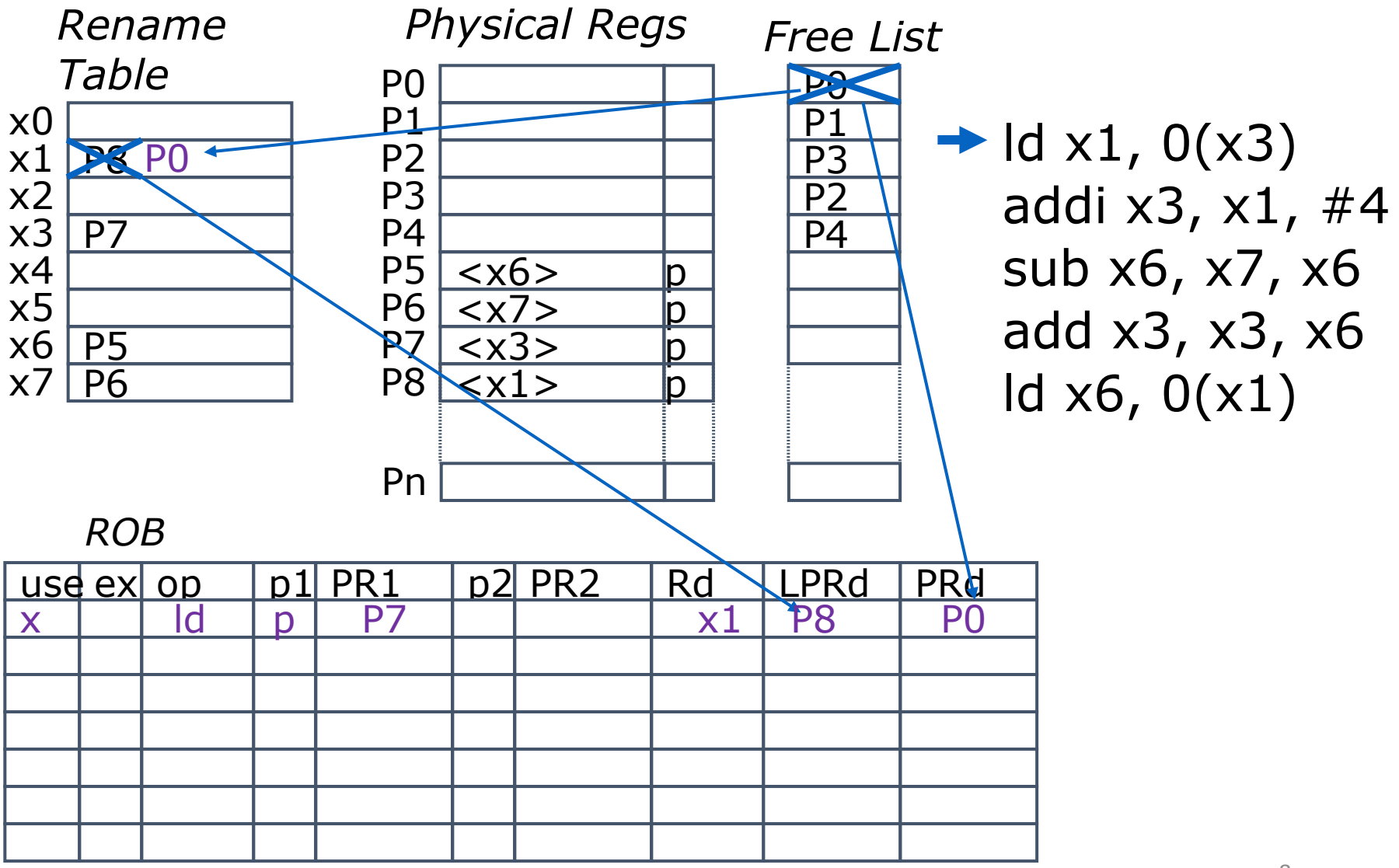
```
ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)
```

ROB

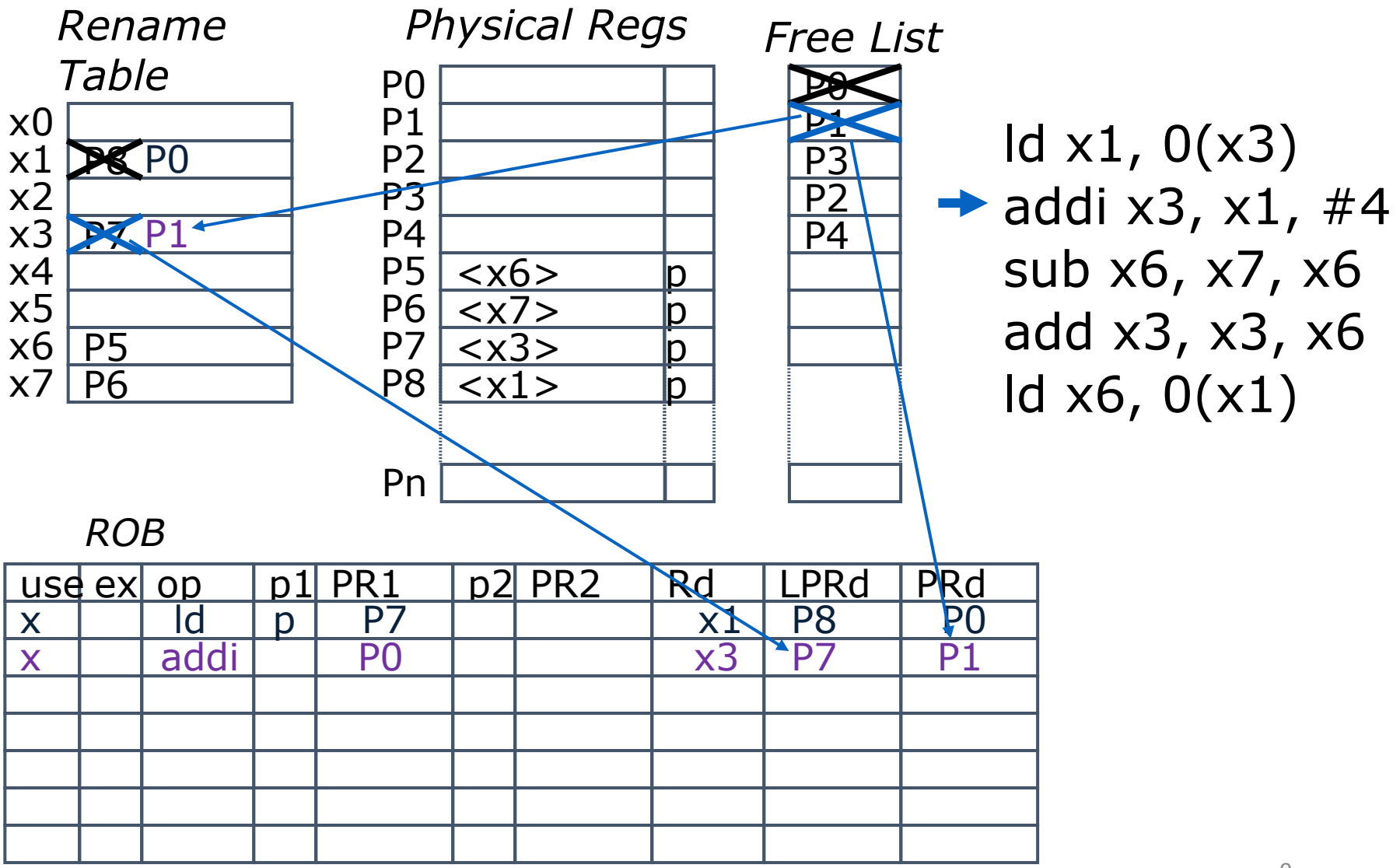
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd

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

Physical Register Management

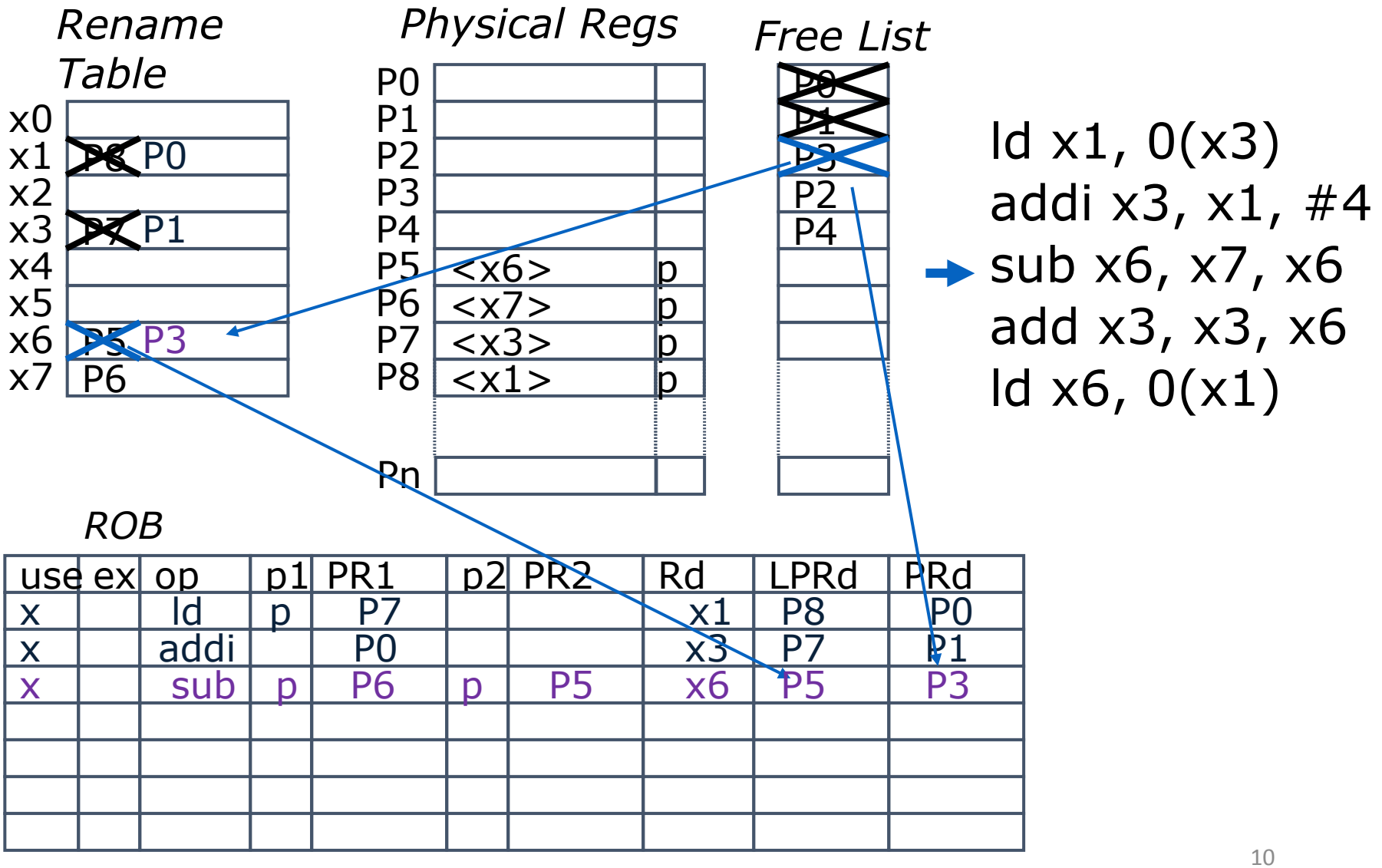


Physical Register Management

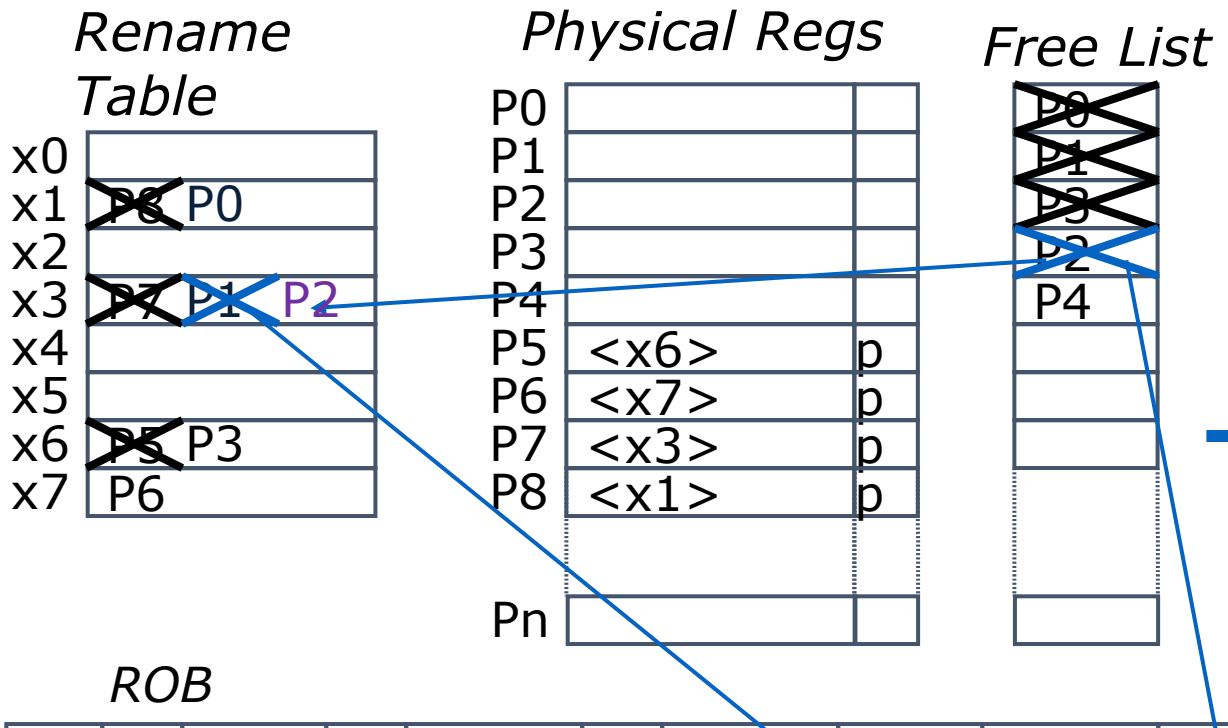


use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1

Physical Register Management



Physical Register Management

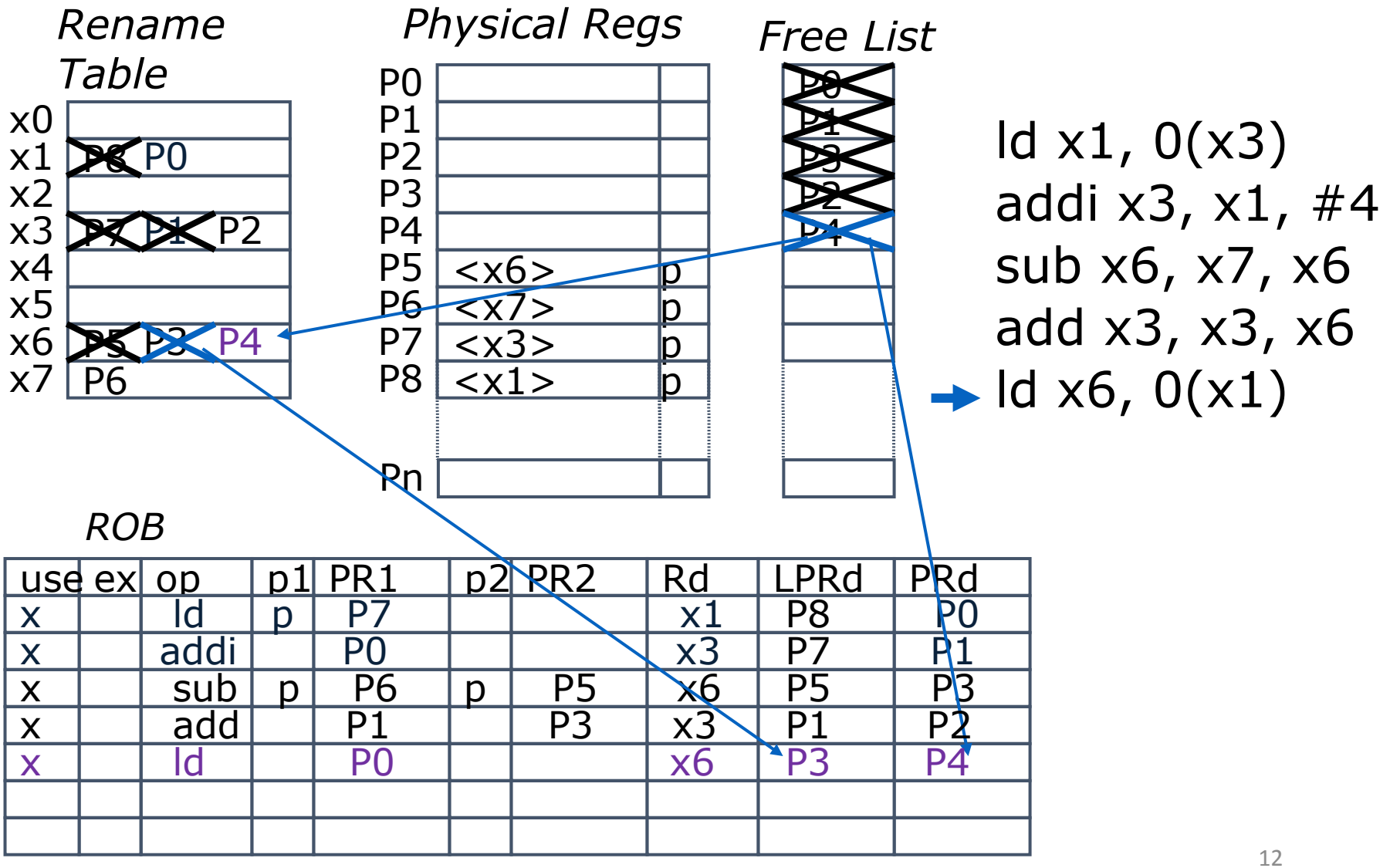


ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
→ add x3, x3, x6
ld x6, 0(x1)

ROB

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

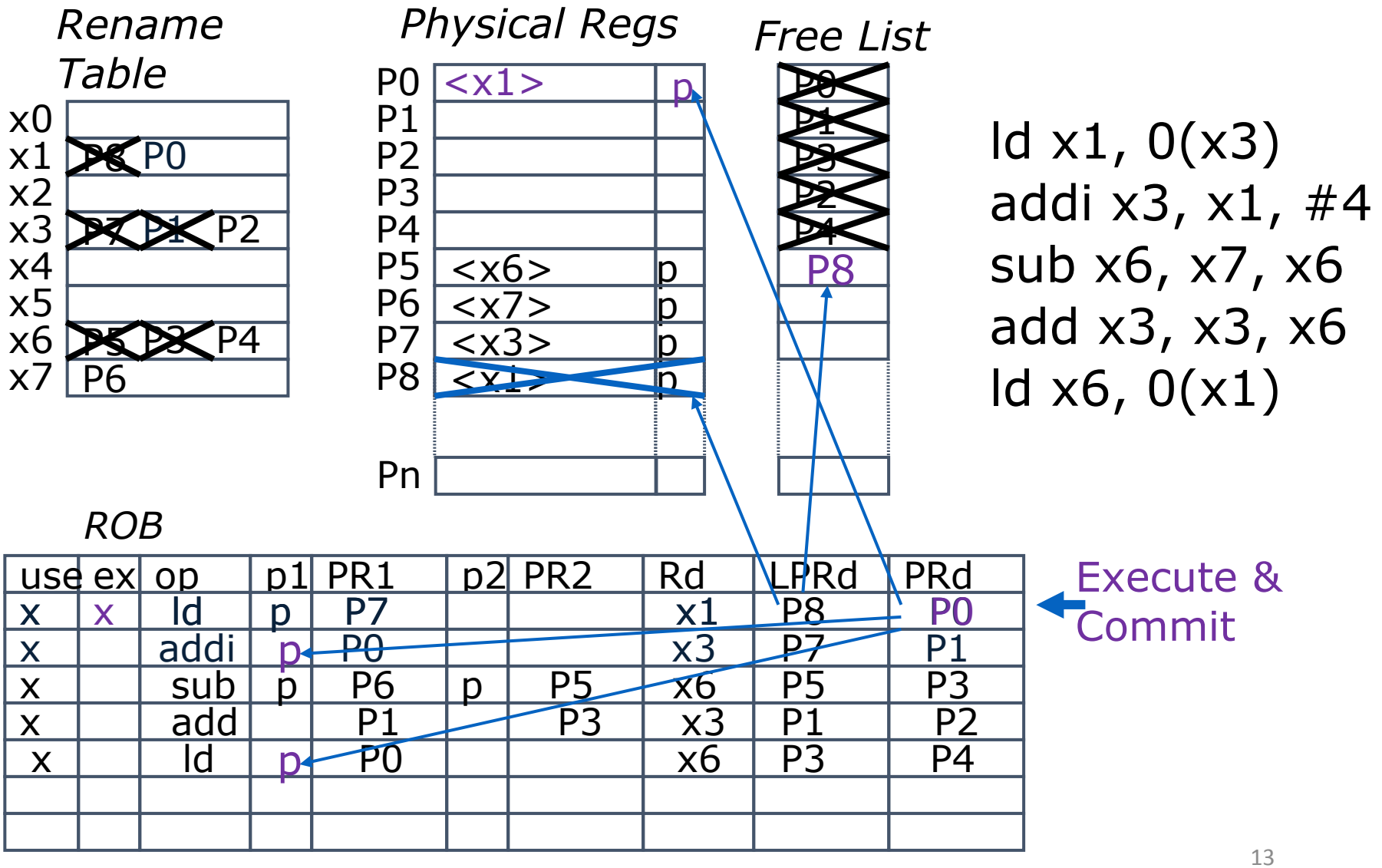
Physical Register Management



ROB

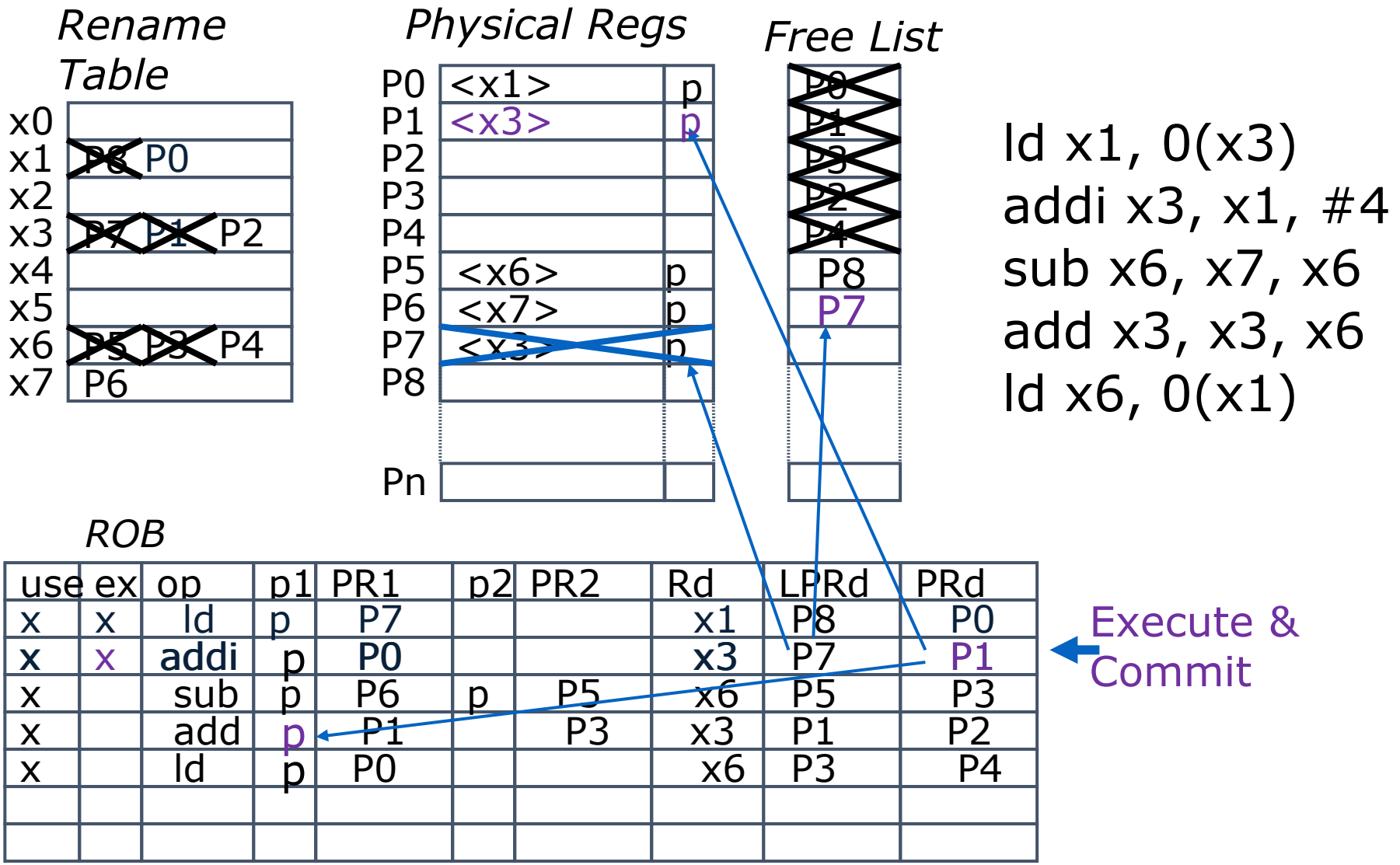
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add		P1		P3	x3	P1	P2
x		ld		P0			x6	P3	P4

Physical Register Management



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

Physical Register Management



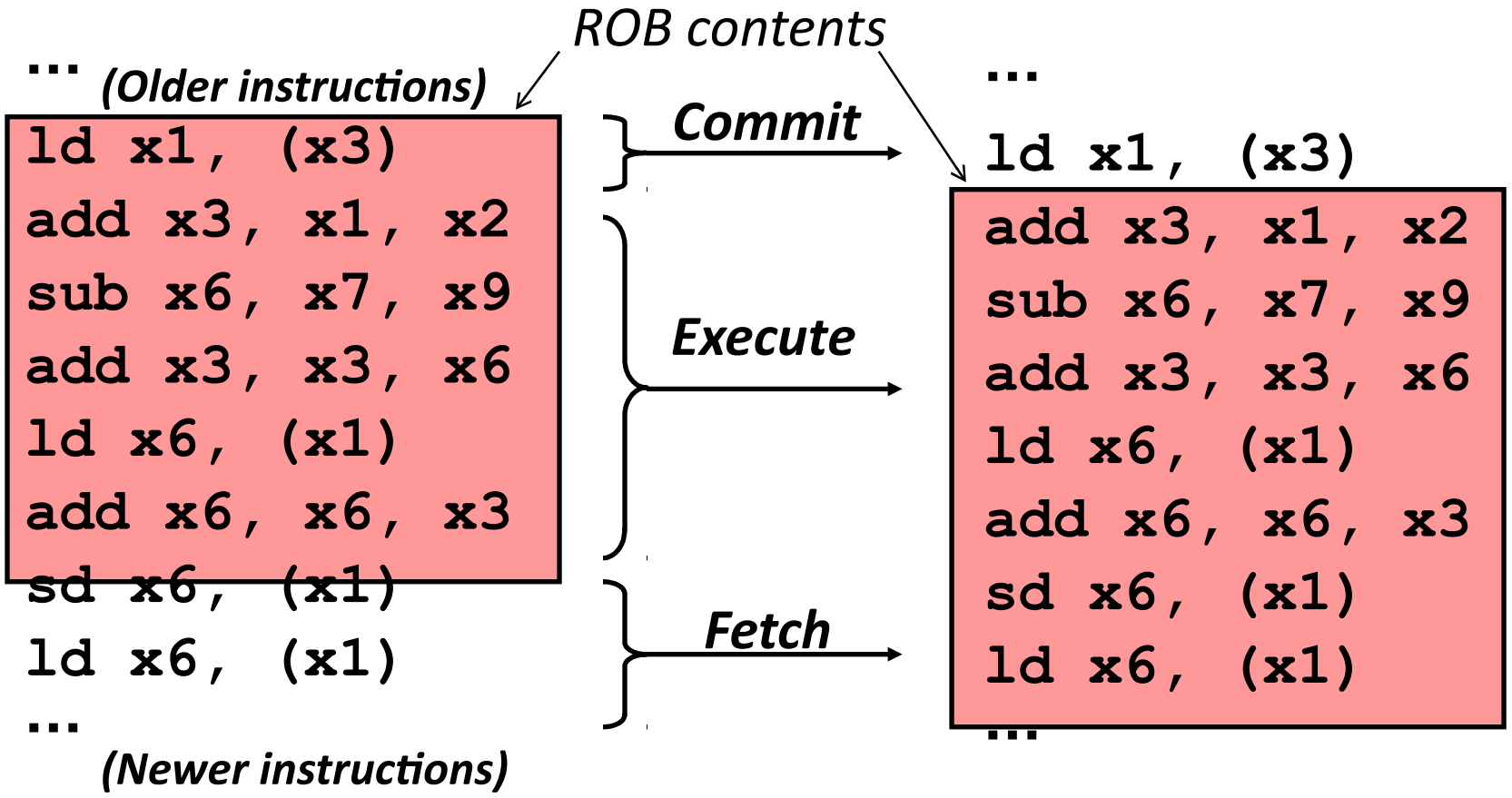
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x	x	ld	p	P7			x1	P8	P0
x	x	addi	p	P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add	p	P1		P3	x3	P1	P2
x		ld	p	P0			x6	P3	P4

Repairing Rename at Traps

- MIPS R10K rename table is repaired by unrenaming instructions in reverse order using the PRd/LPRd fields

- 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)

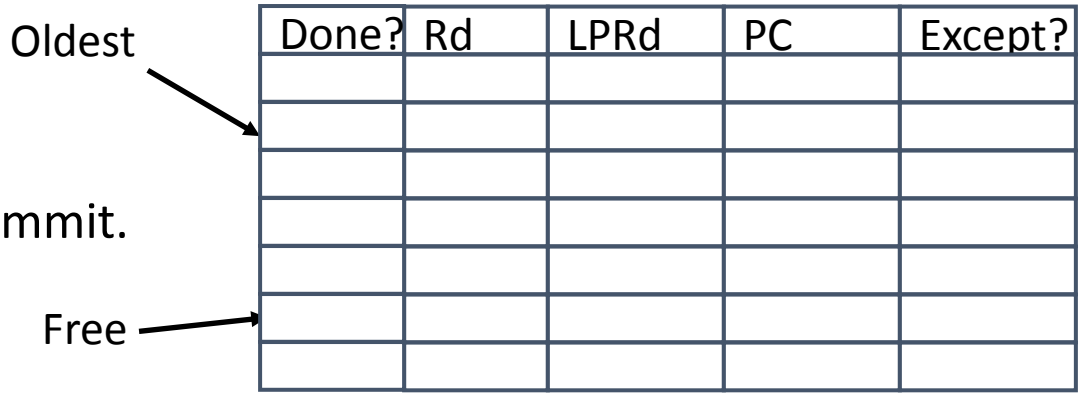


Separate Issue Window from ROB

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#

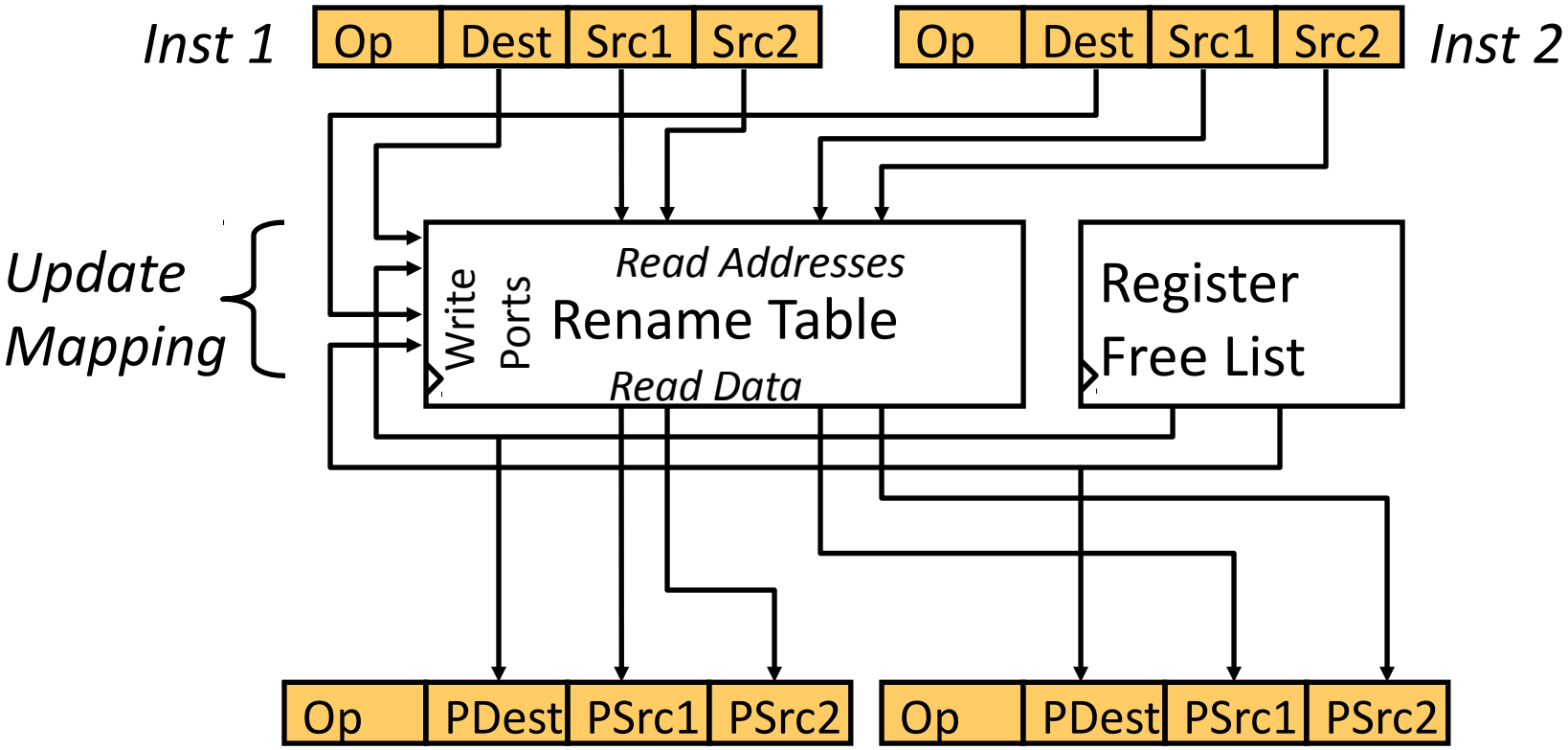
Reorder buffer used to hold exception information for commit.



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

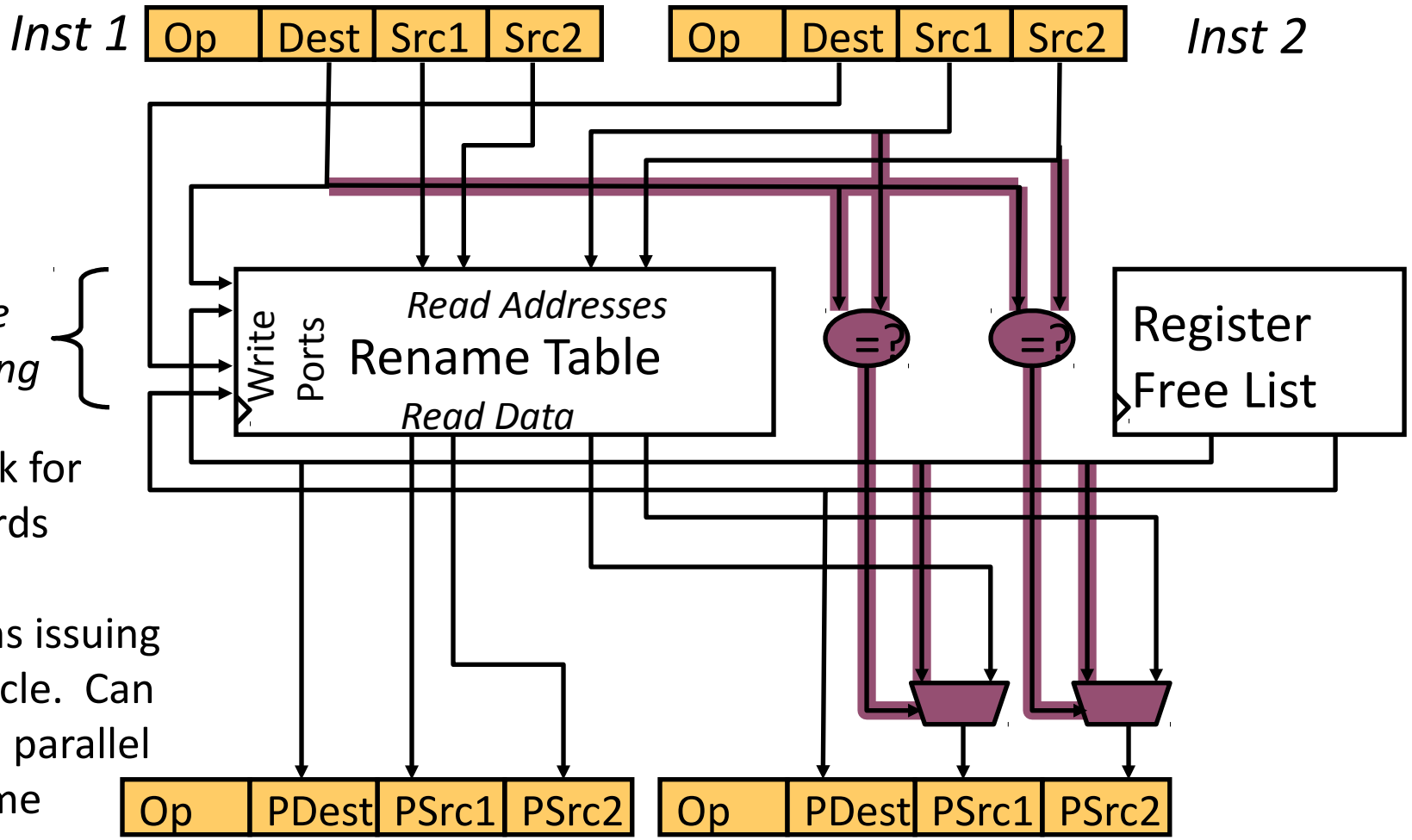
Superscalar Register Renaming

- 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

Out-of-Order Fades into Background

- Out-of-order processing implemented commercially in 1960s, but disappeared again until 1990s as two major problems had to be solved:
- Precise traps
  - Imprecise traps complicate debugging and OS code
  - Note, precise interrupts are relatively easy to provide
- Branch prediction
  - Amount of exploitable instruction-level parallelism (ILP) limited by control hazards
- Also, simpler machine designs in new technology beat complicated machines in old technology
  - Big advantage to fit processor & caches on one chip
  - Microprocessors had era of 1%/week performance scaling

Effectiveness?

Renaming and Out-of-order execution was first implemented in 1969 in IBM 360/91 but did not show up in the subsequent models until mid-Nineties.

Why ?

Reasons

1. Exceptions not precise!
2. Effective on a very small class of programs

One more problem needed to be solved _____

Precise Interrupts

- It must appear as if an interrupt is taken between two instructions (say I_i and I_{i+1})
  - the effect of all instructions up to and including I_i is totally complete
  - no effect of any instruction after I_i has taken place

- The interrupt handler either aborts the program or restarts it at I_{i+1}.

Effect on Interrupts

Out-of-order Completion

I_1	DIVD	f6,	f6,	f4
	LD	f2,	45(r3)	
I_2	MULTD	f0,	f2,	f4
	DIVD	f8,	f6,	
I_3	SUBD	f10,	f0,	f2
	ADDD	f6,	f8,	
I_4				f6

out-of-order comp 1 2 2 3 1 4 3 5 5 4 6 6

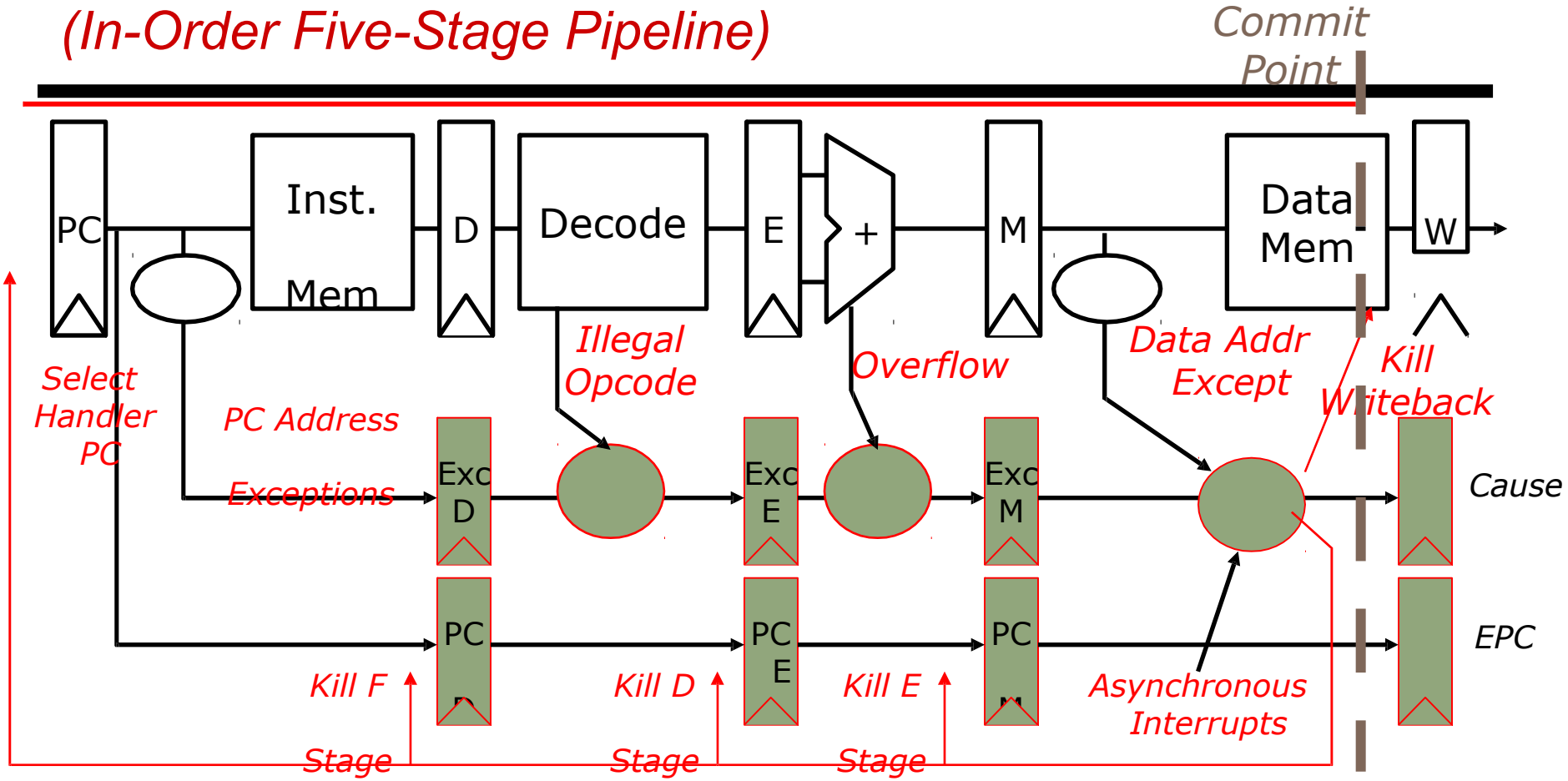
I_5 *restore f2* f2 *restore f10*

Consider interrupts

I_6

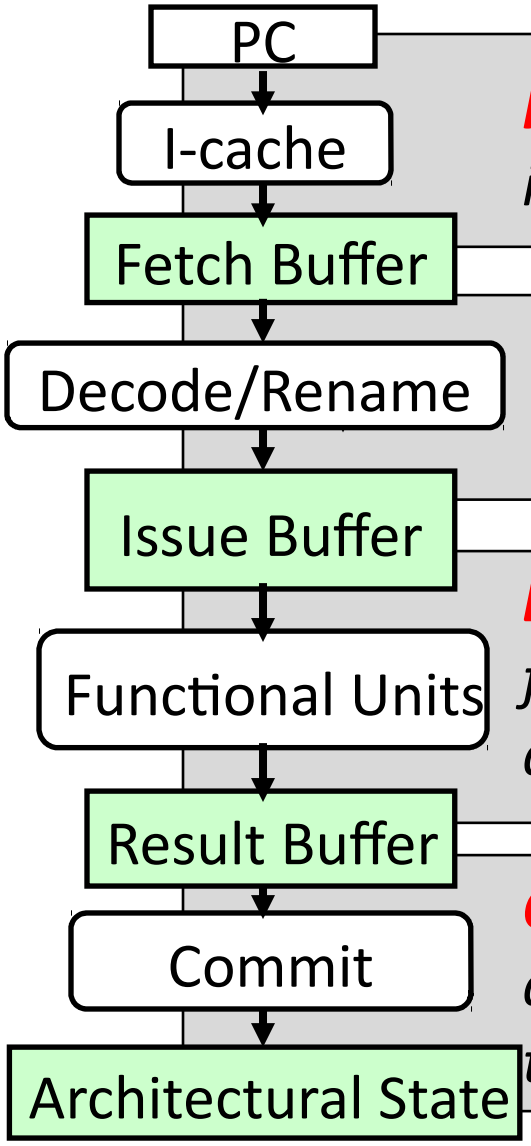
Precise interrupts are difficult to implement at high speed
- want to start execution of later instructions before exception checks finished on earlier instructions

Exception Handling
(In-Order Five-Stage Pipeline)



- Hold exception flags in pipeline until commit point (M stage)
- Exceptions in earlier pipe stages override later exceptions
- Inject external interrupts at commit point (override others)
- If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage

Phases of Instruction Execution



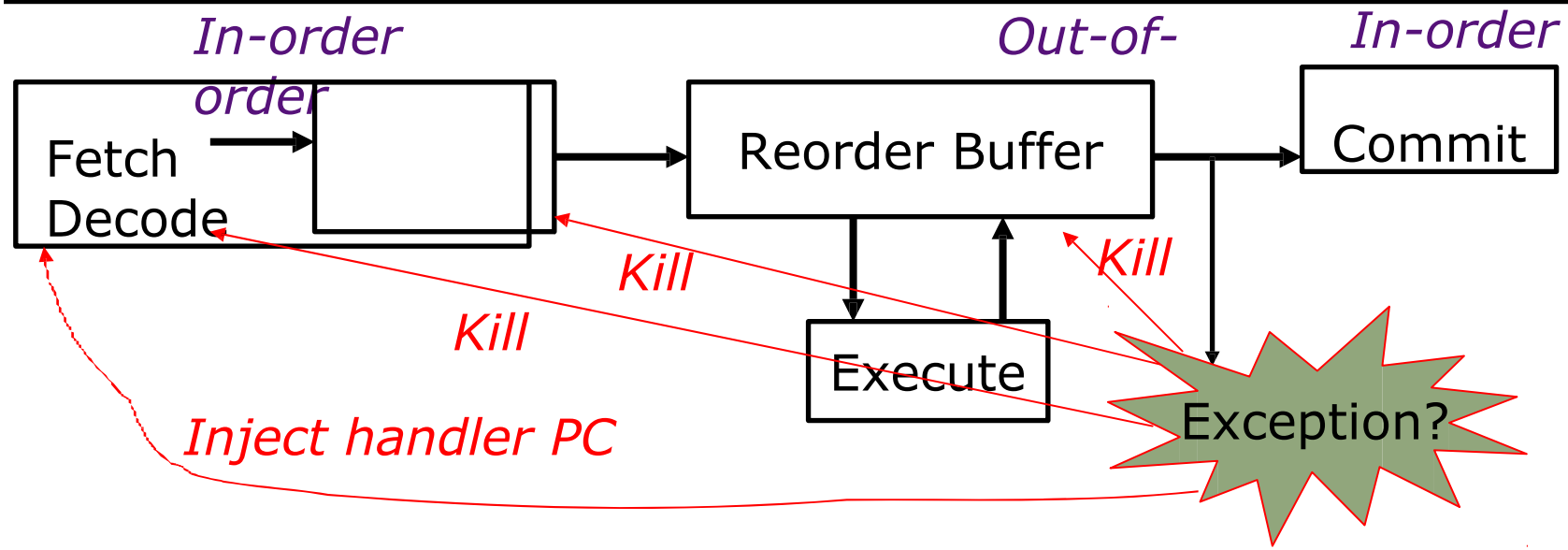
Fetch: *Instruction bits retrieved from instruction cache.*

Decode: *Instructions dispatched to appropriate issue buffer*

Execute: *Instructions and operands issued to functional units. When execution completes, all results and exception flags are available.*

Commit: *Instruction irrevocably updates architectural state (aka "graduation"), or takes precise trap/interrupt.*

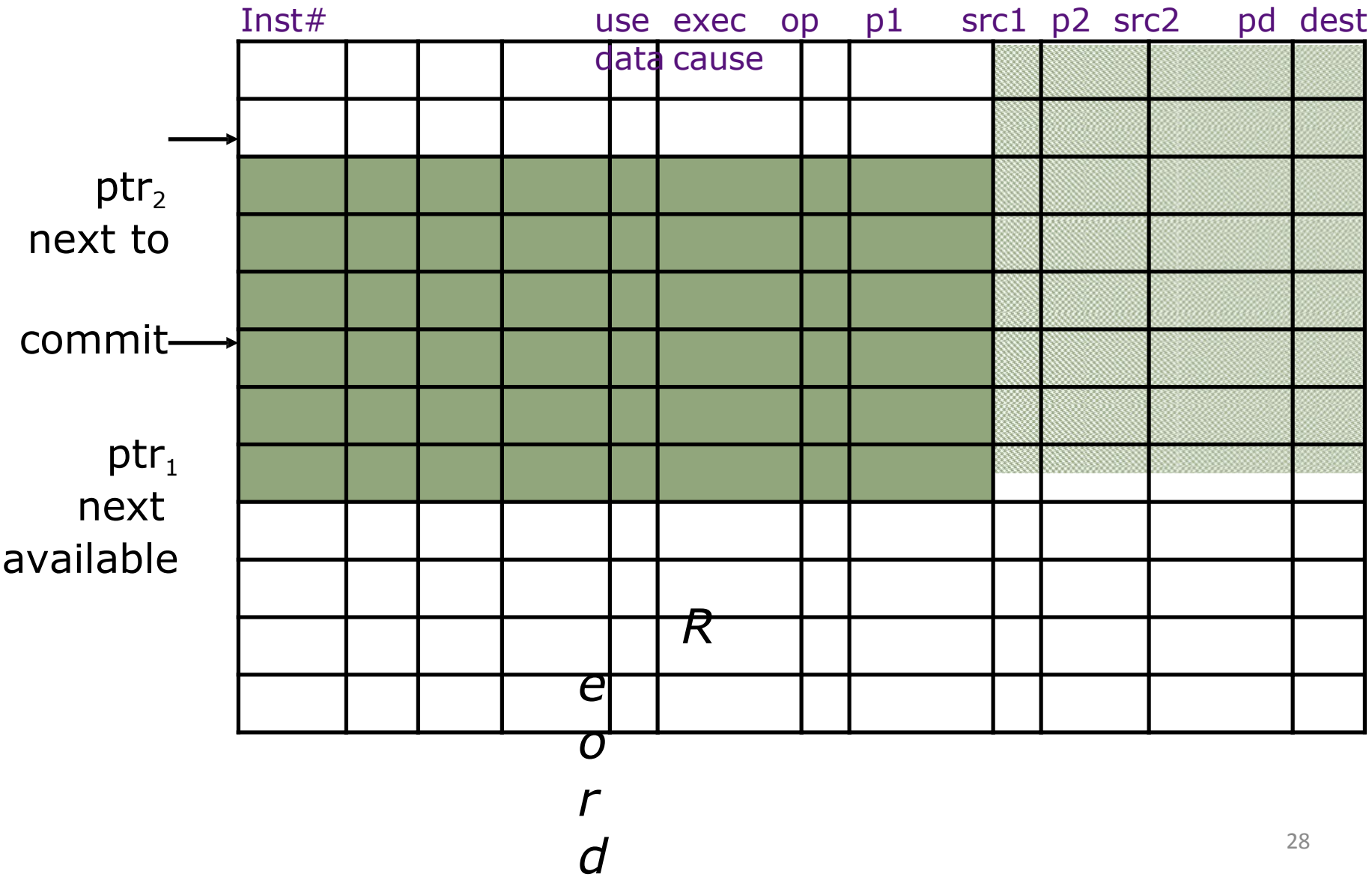
In-Order Commit for Precise Exceptions



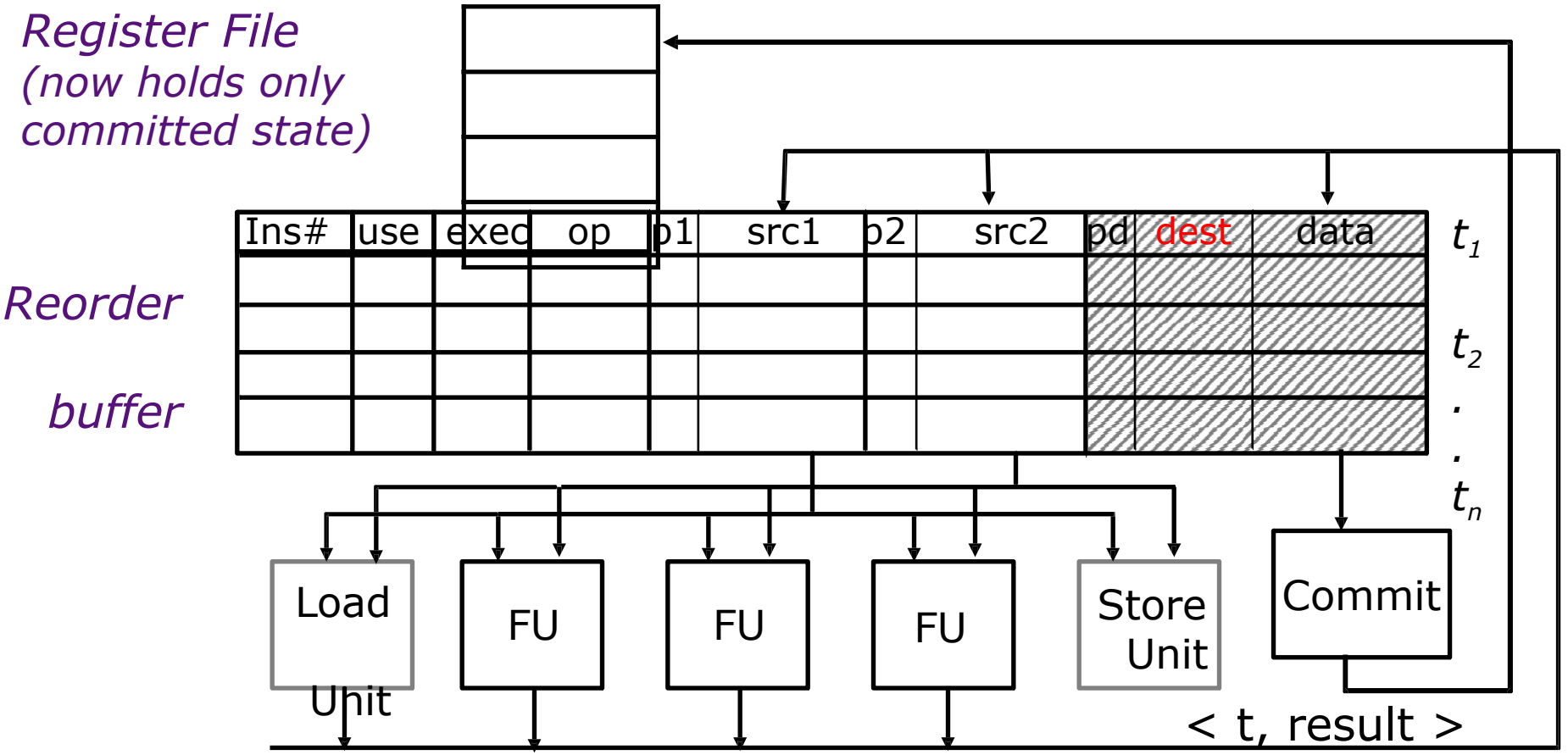
- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order ($\Rightarrow$ out-of-order completion)
- *Commit* (write-back to architectural state, i.e., regfile & memory, is in-order

Temporary storage needed to hold results before commit (shadow registers and store buffers)

Extensions for Precise Exceptions



Rollback and Renaming

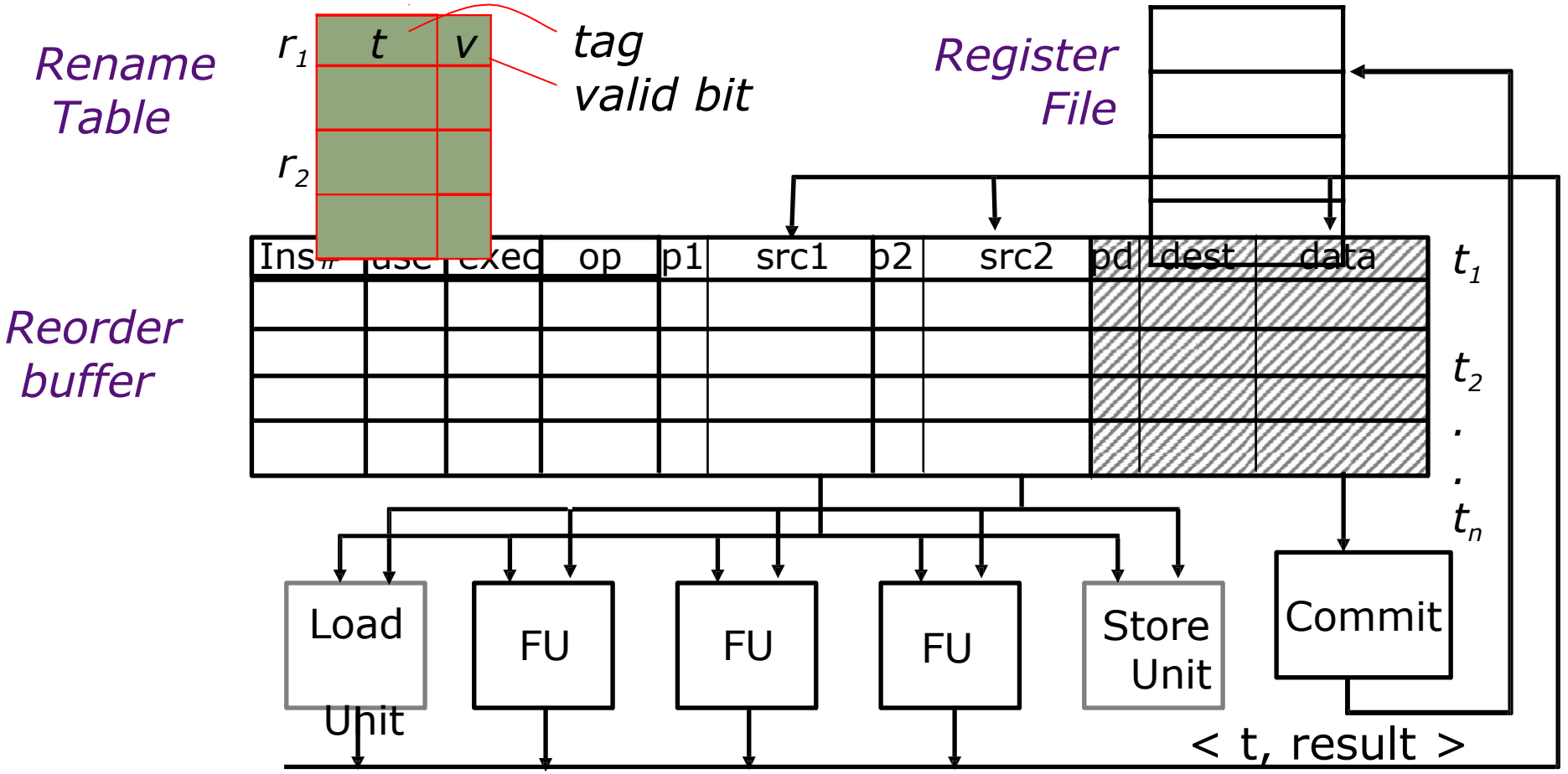


Register file does not contain renaming tags any more.

How does the decode stage find the tag of a source register?

Search the "dest" field in the reorder buffer

Renaming Table



Renaming table is a cache to speed up register name look up. It needs to be cleared after each exception taken.

When else are valid bits cleared?

Control transfers

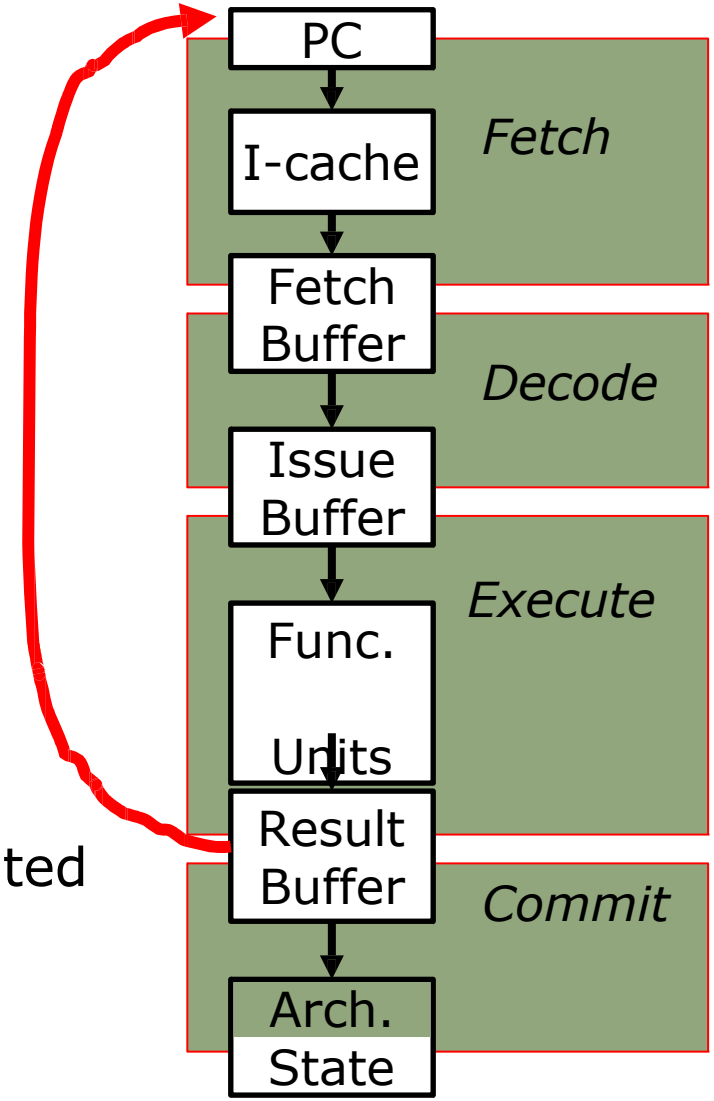
Branch Penalty

Next fetch started

How many instructions need to be killed on a misprediction?

Modern processors may have > 10 pipeline stages between next pc calculation and branch resolution !

Branch executed



Average Run-Length between Branches

Average dynamic instruction mix from SPEC92:

	SPECint92	SPECfp92
ALU	39 %	13 %
FPU Add		20 %
FPU Mult		13 %
load	26 %	23 %
store	9 %	9 %
branch	16 %	8 %
other	10 %	12 %

SPECint92: *compress, eqntott, espresso, gcc , li*
SPECfp92: *doduc, ear, hydro2d, mdijdp2, su2cor*

What is the average *run length* between branches

Out-of-order Completion
In-order Issue

					Latency
I_1	FDIV.D	f6,	f6,	f4	*4*
I_2	FLD	f2,	45(x3)		*1*
I_3	FMULT.D	f0,	f2,	f4	*3*
I_4	FDIV.D	f8,	f6,	f2	*4*
I_5	FSUB.D	f10,	f0,	f6	*1*
I_6	FADD.D	f6,	f8,	f2	*1*

in-order comp 1 2 1 2 3 4 3 5 4 6 5 6

out-of-order comp 1 2 2 3 1 4 3 5 5 4 6 6

In-Order versus Out-of-Order Phases

- Instruction fetch/decode/rename always in-order
  - Need to parse ISA sequentially to get correct semantics
  - Proposals for speculative OoO instruction fetch, e.g., Multiscalar. Predict control flow and data dependencies across sequential program segments fetched/decoded/executed in parallel, fixup if prediction wrong

- Dispatch (place instruction into machine buffers to wait for issue) also always in-order
  - Some use “Dispatch” to mean “Issue”, but not in these lectures

In-Order Versus Out-of-Order Issue

- In-order issue:
  - Issue stalls on RAW dependencies or structural hazards, or possibly WAR/WAW hazards
  - Instruction cannot issue to execution units unless all preceding instructions have issued to execution units

- Out-of-order issue:
  - Instructions dispatched in program order to reservation stations (or other forms of instruction buffer) to wait for operands to arrive, or other hazards to clear
  - While earlier instructions wait in issue buffers, following instructions can be dispatched and issued out-of-order

In-Order versus Out-of-Order Completion

- All but the simplest machines have out-of-order completion, due to different latencies of functional units and desire to bypass values as soon as available

- Classic RISC 5-stage integer pipeline just barely has in-order completion
  - Load takes two cycles, but following one-cycle integer op completes at same time, not earlier
  - Adding pipelined FPU immediately brings OoO completion

In-Order versus Out-of-Order Commit

- In-order commit supports precise traps, standard today
  - Some proposals to reduce the cost of in-order commit by retiring some instructions early to compact reorder buffer, but this is just an optimized in-order commit

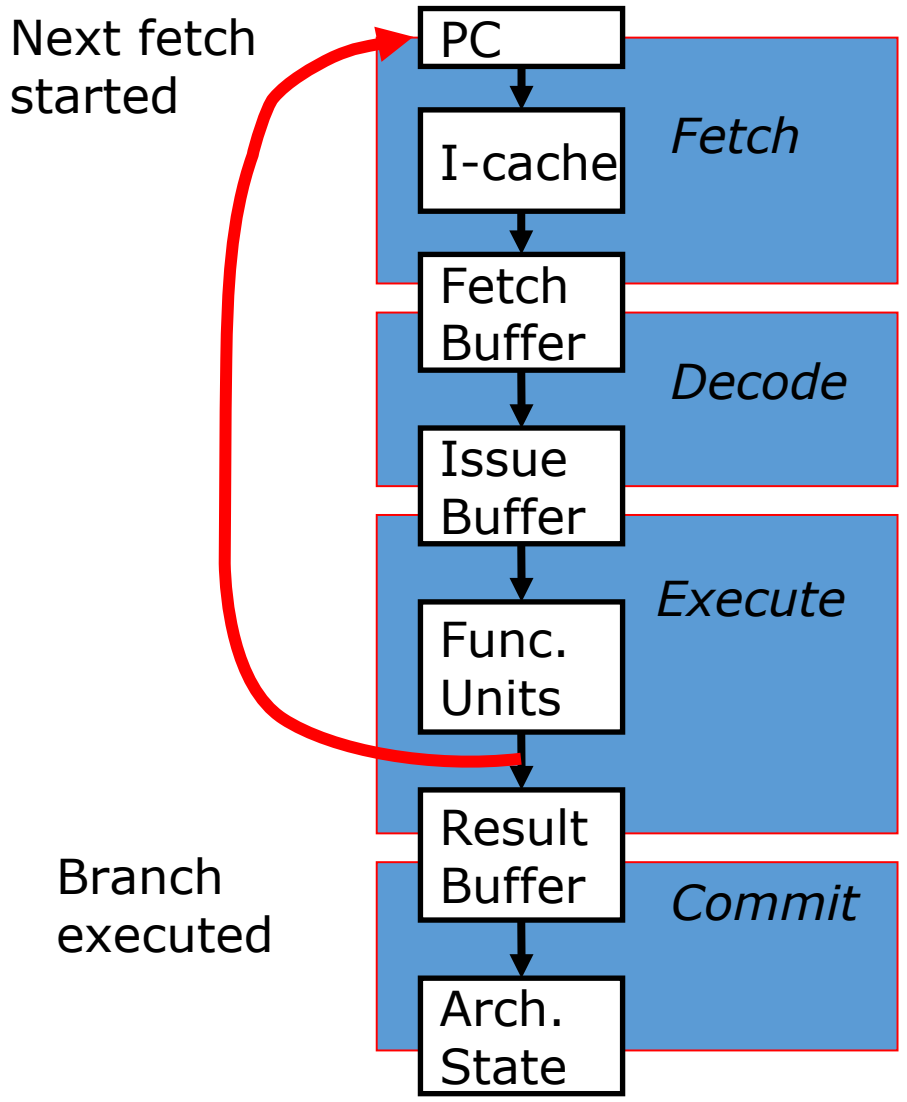
- Out-of-order commit was effectively what early OoO machines implemented (imprecise traps) as completion irrevocably changed machine state
  - i.e., complete == commit in these machines

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, i.e., branch prediction
    - Speculative execution of instructions beyond the branch
    - Many advances in accuracy, widely used

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

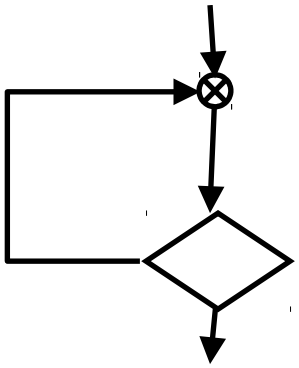
- On a mispredict, could throw away 8*4+(80-1)=111 instructions

- 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

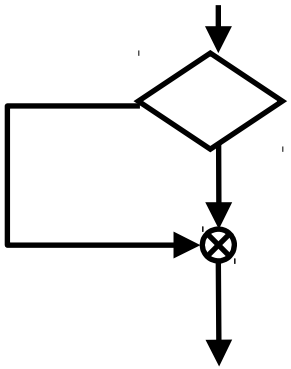
Static Branch Prediction

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)

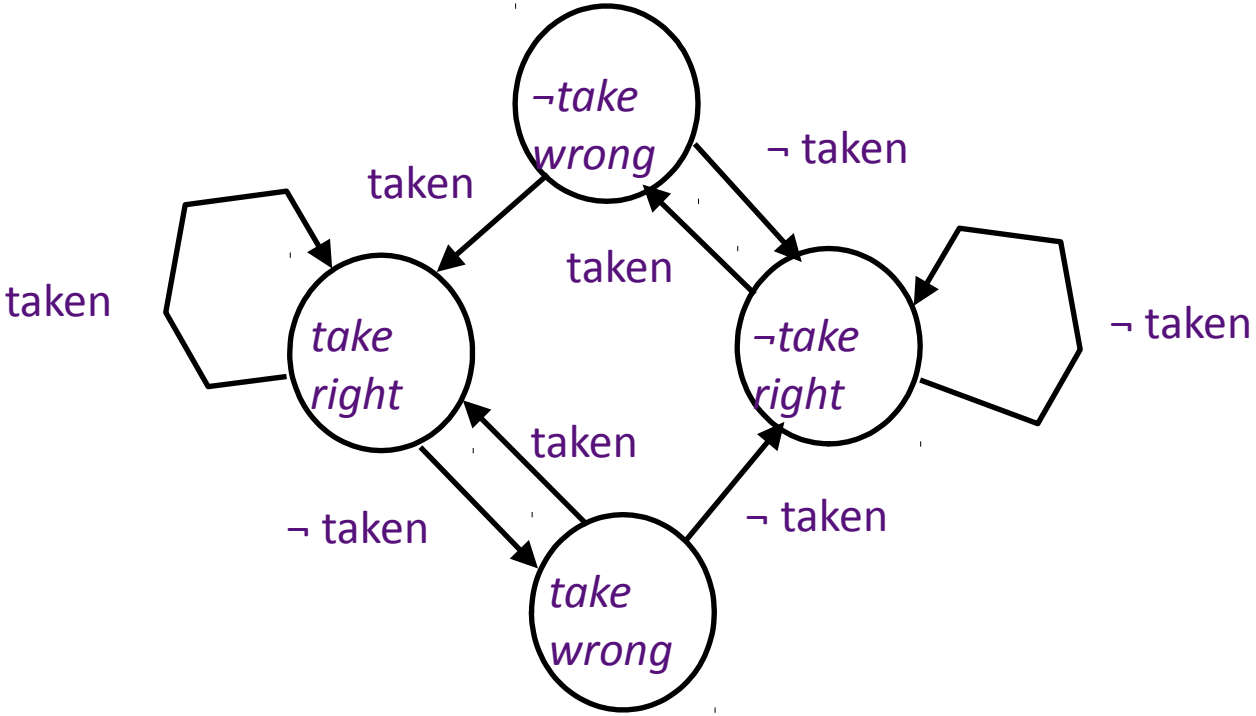
One-Bit Branch History Predictor

- For each branch, remember last way branch went

- Has problem with loop-closing backward branches, as two mispredicts occur on every loop execution
  - first iteration predicts loop backwards branch not-taken (loop was exited last time)
  - last iteration predicts loop backwards branch taken (loop continued last time)

Branch Prediction Bits

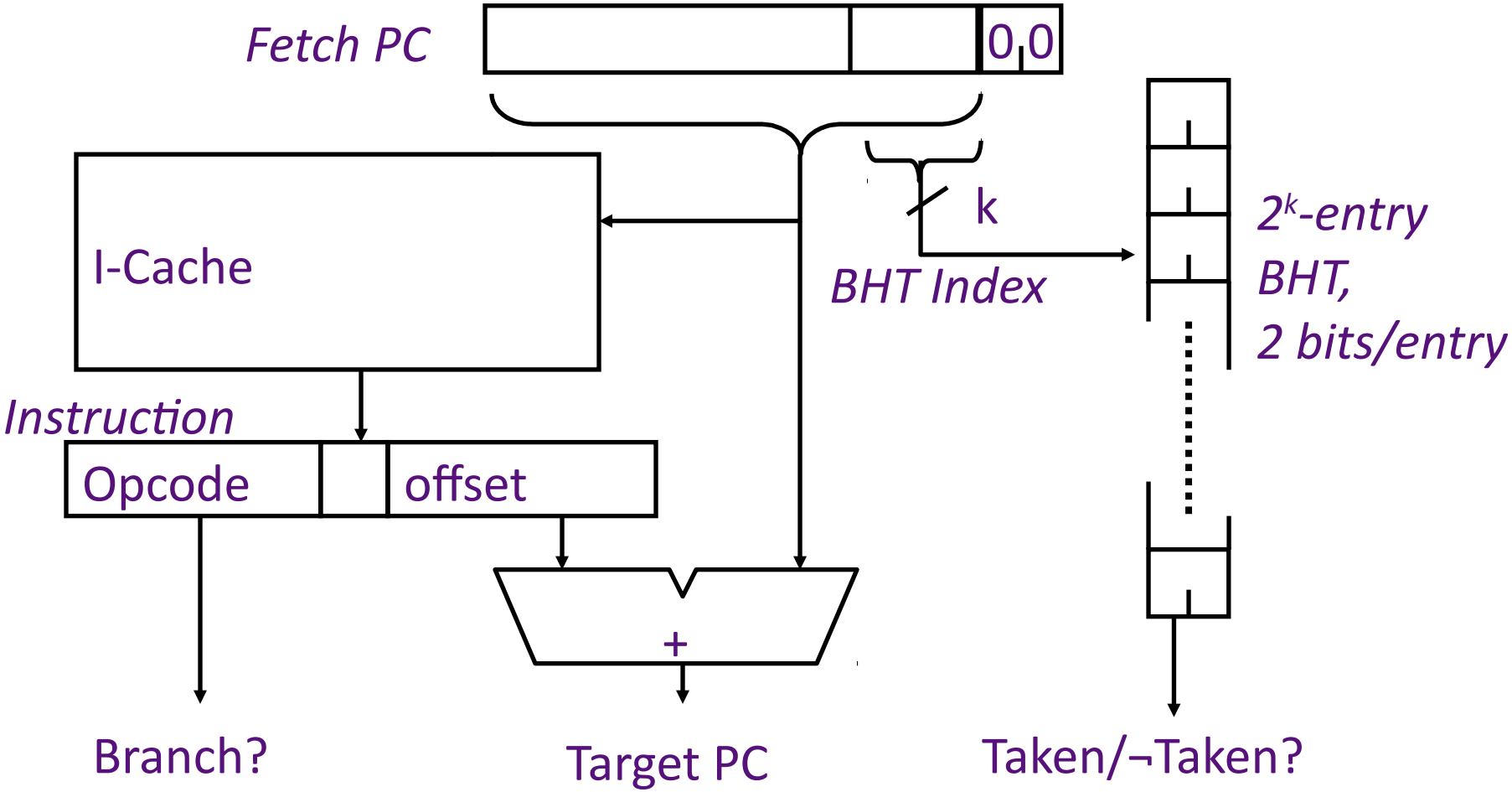
- Assume 2 BP bits per instruction
- Change the prediction after two consecutive mistakes!



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

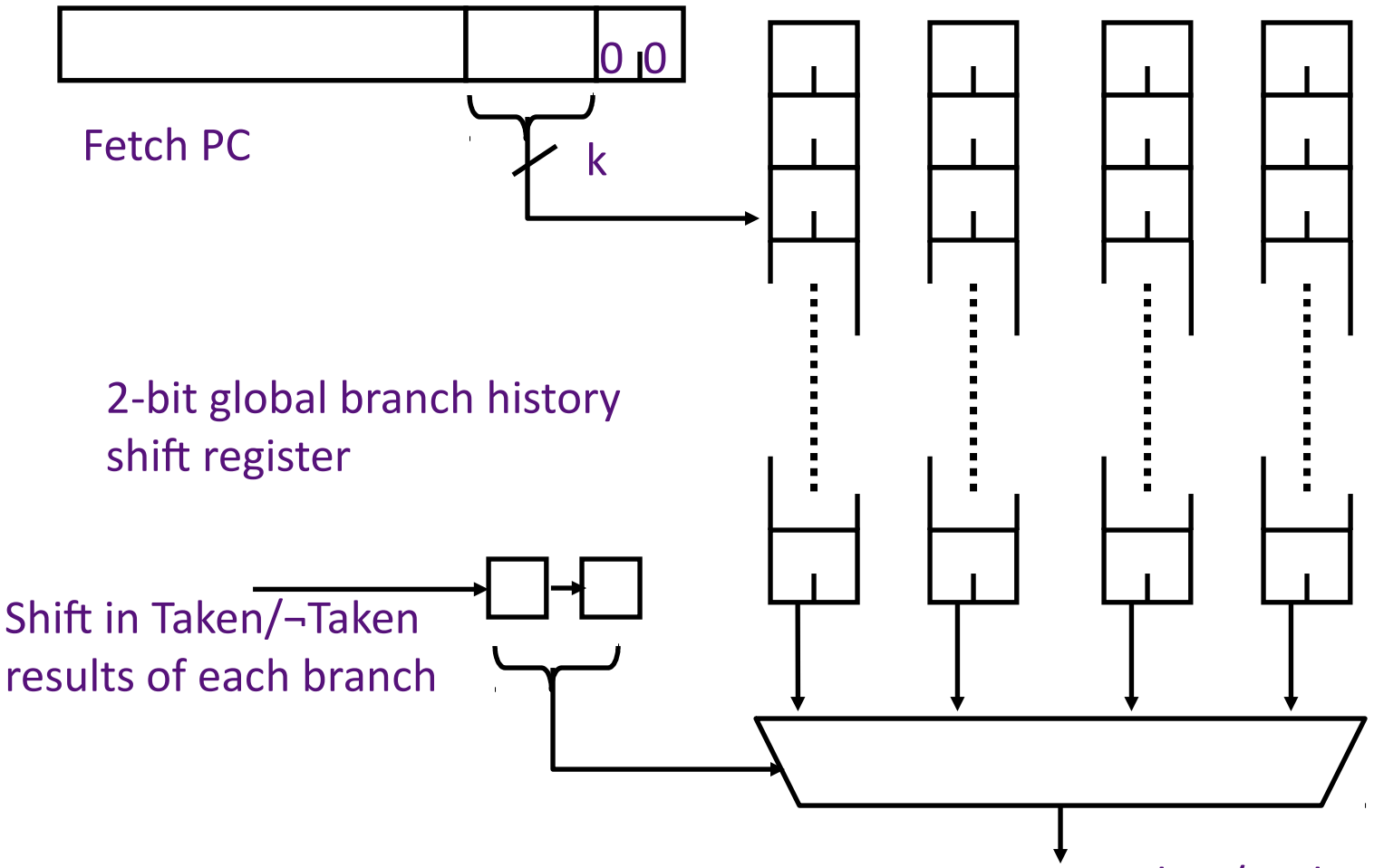
```
if (x[i] < 7) then
   y += 1;
if (x[i] < 5) then
   c -= 4;
```

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)



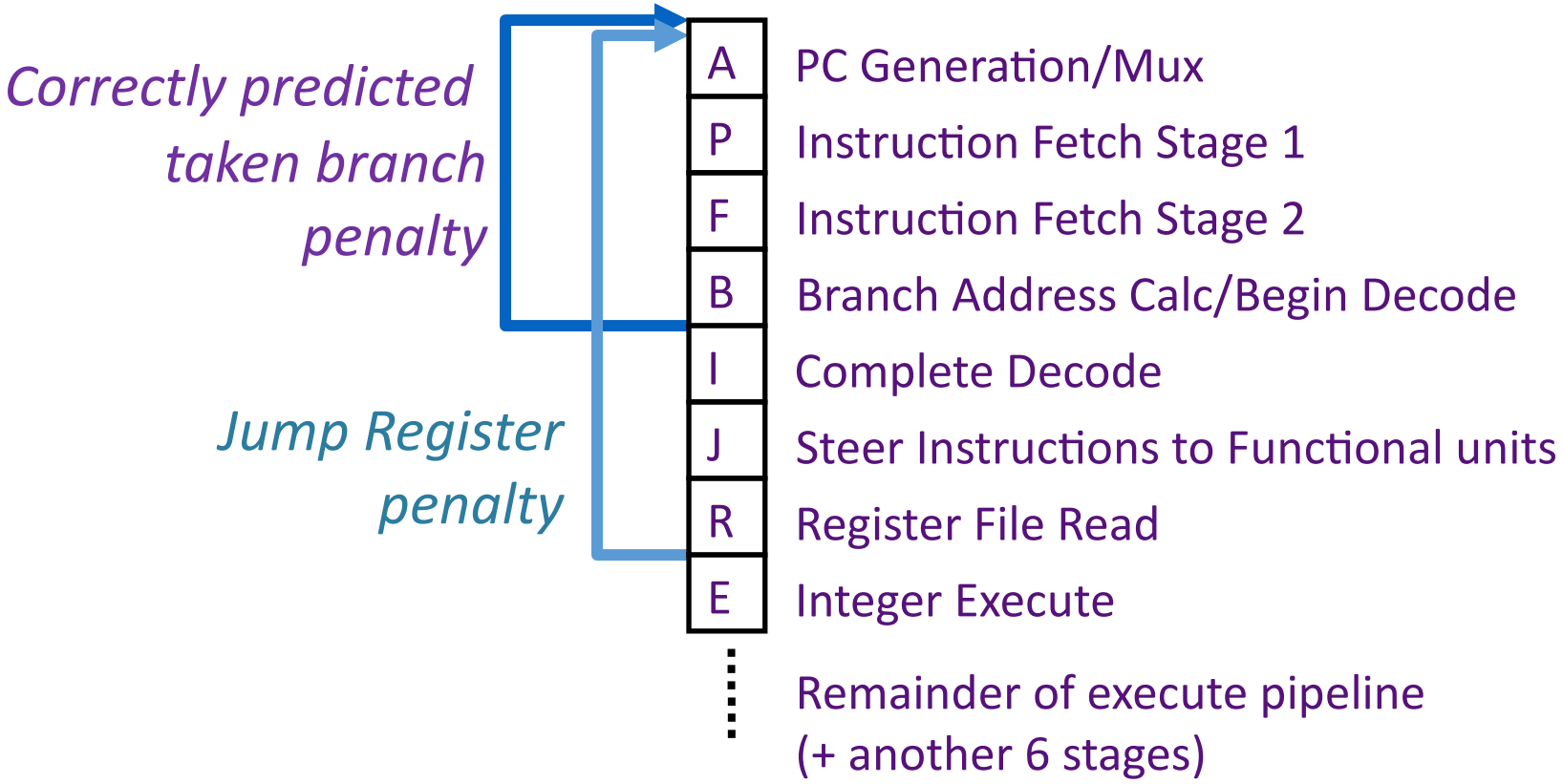
Speculating Both Directions?

- 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!

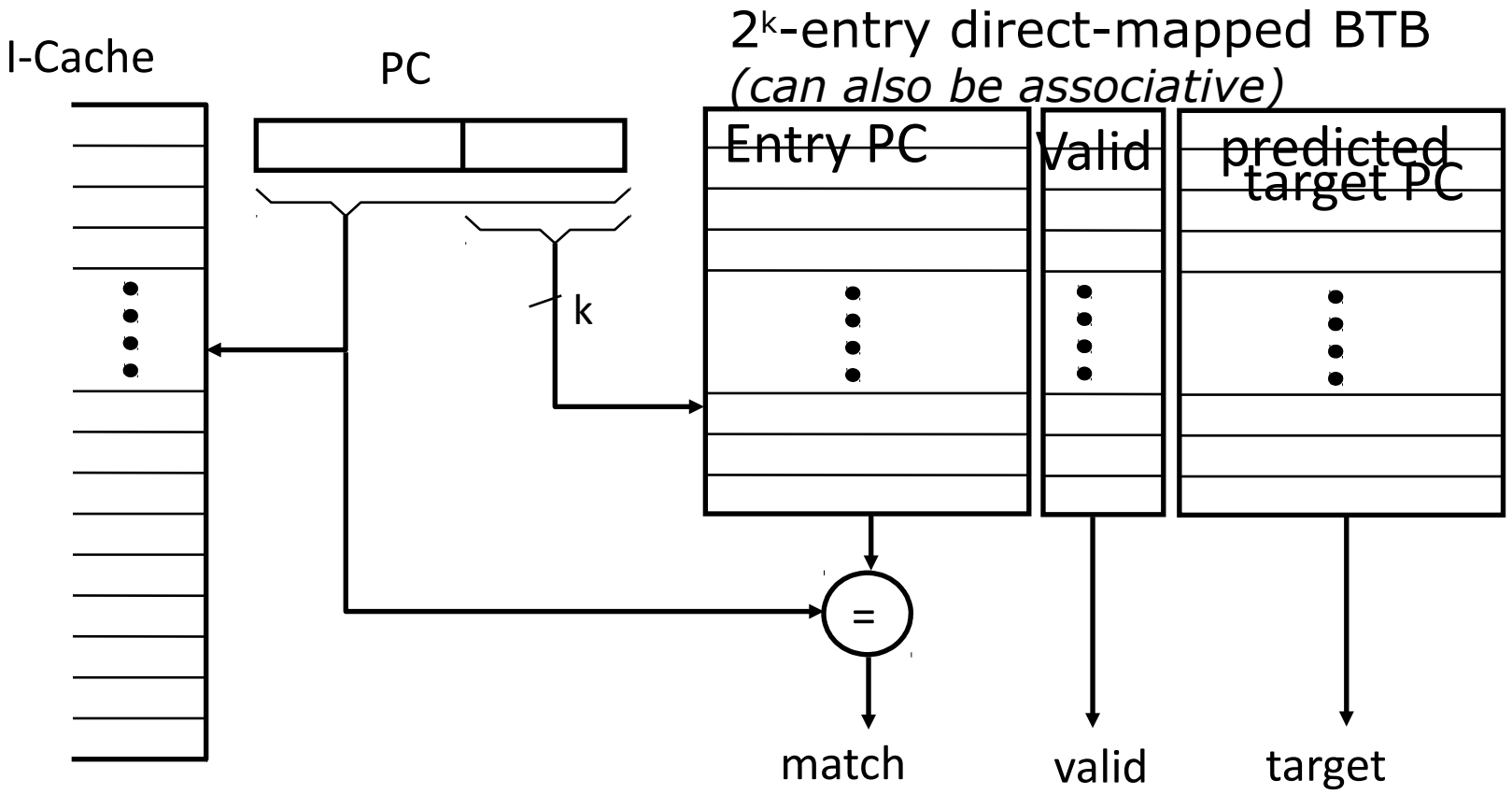
Limitations of BHTs

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



UltraSPARC-III fetch pipeline

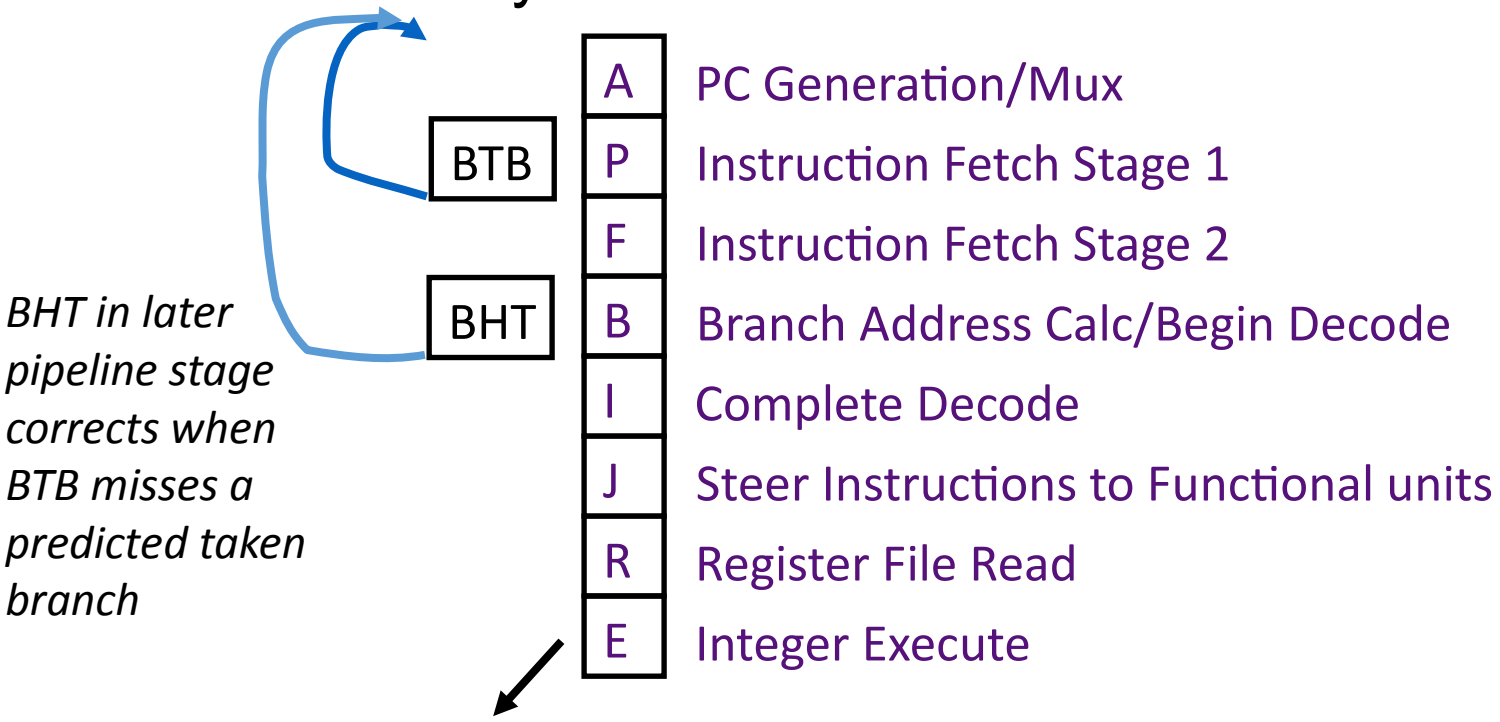
Branch Target Buffer (BTB)



- 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



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

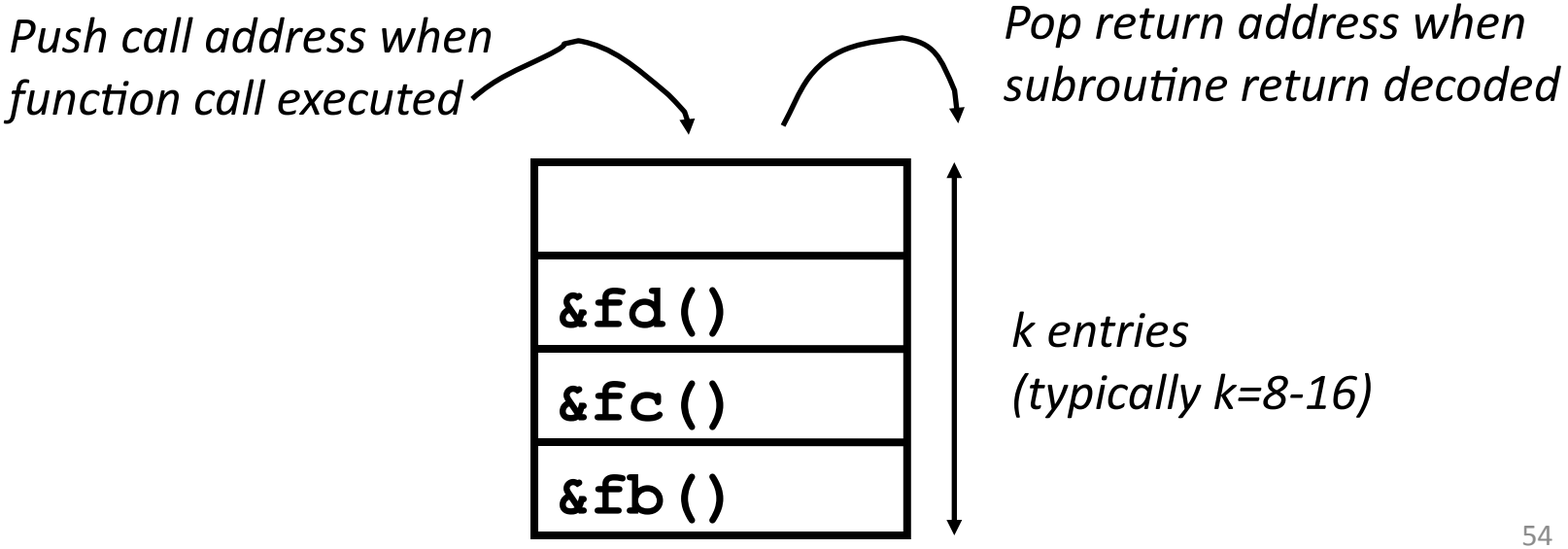
  $\Rightarrow$ *Often one function called from many distinct call sites!*

  How well does BTB work for each of these cases?

Subroutine Return Stack

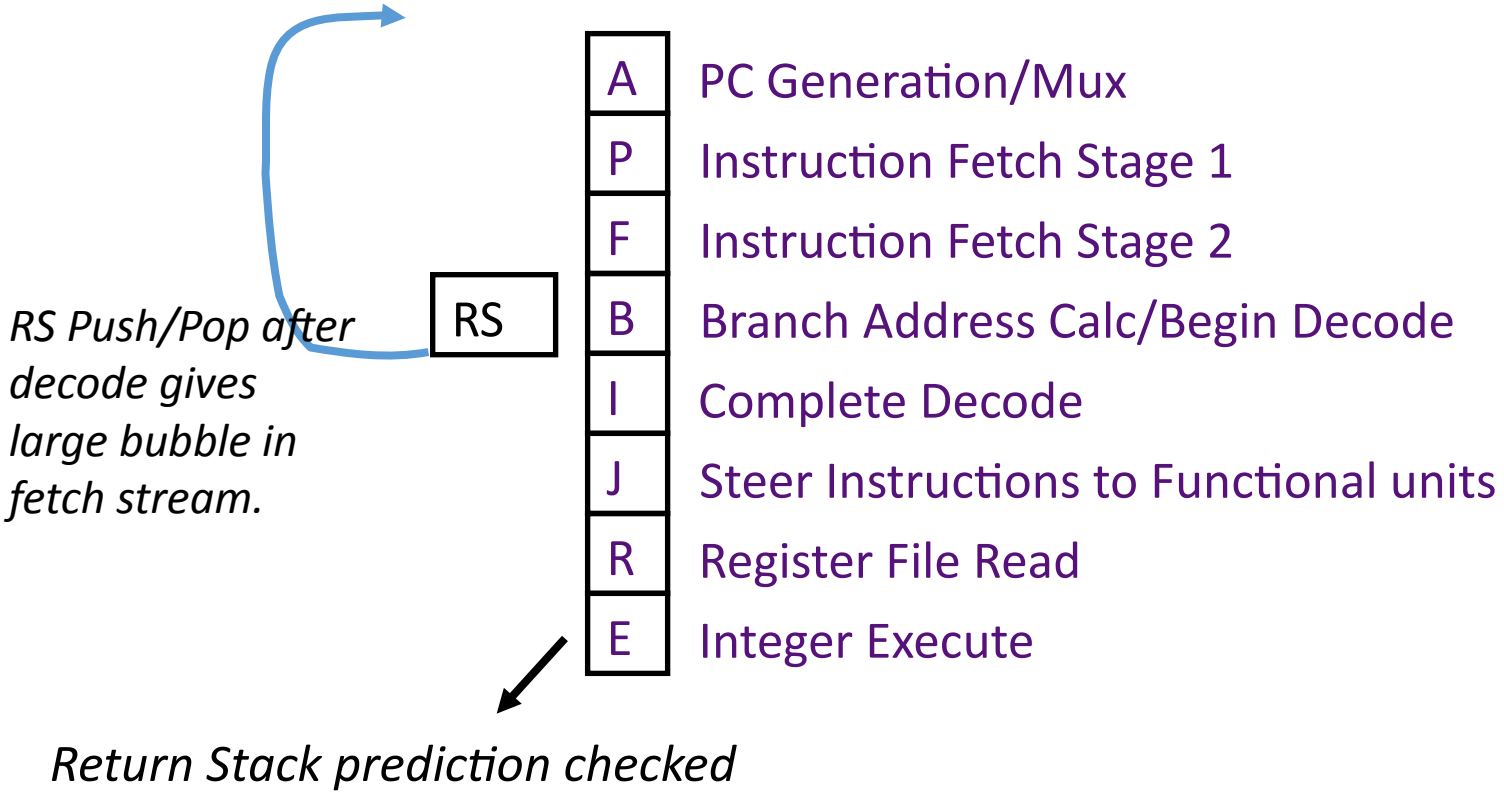
Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

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



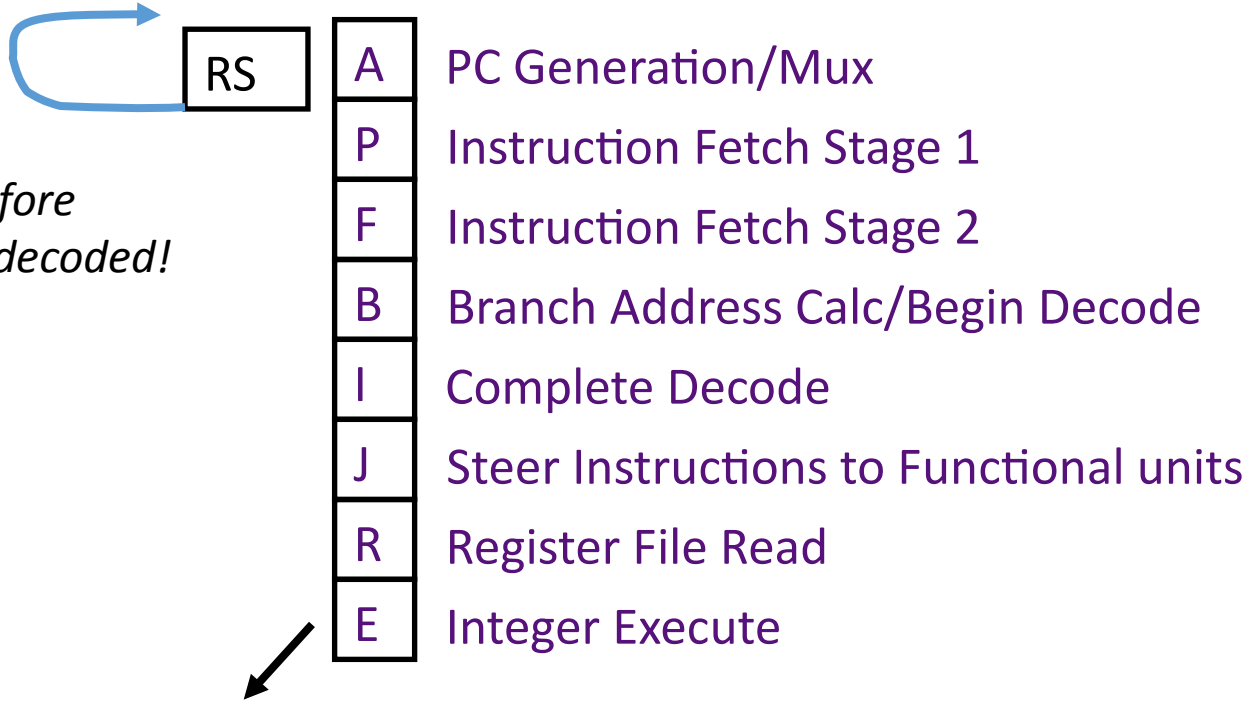
Return Stack in Pipeline

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

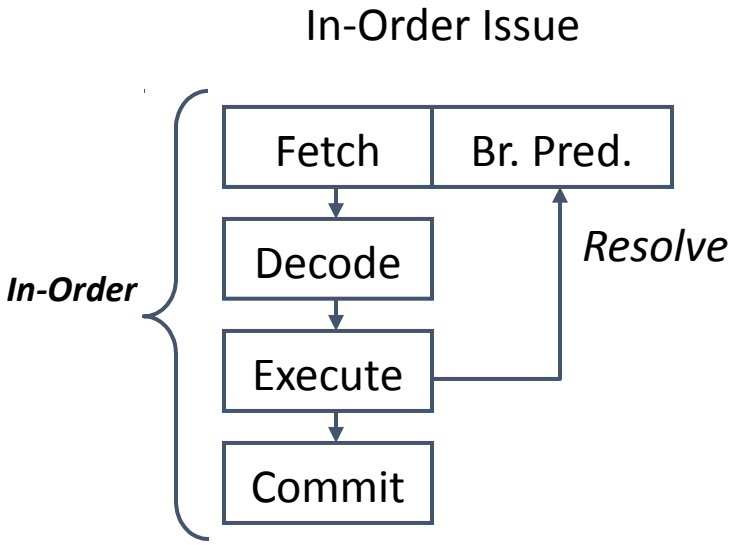


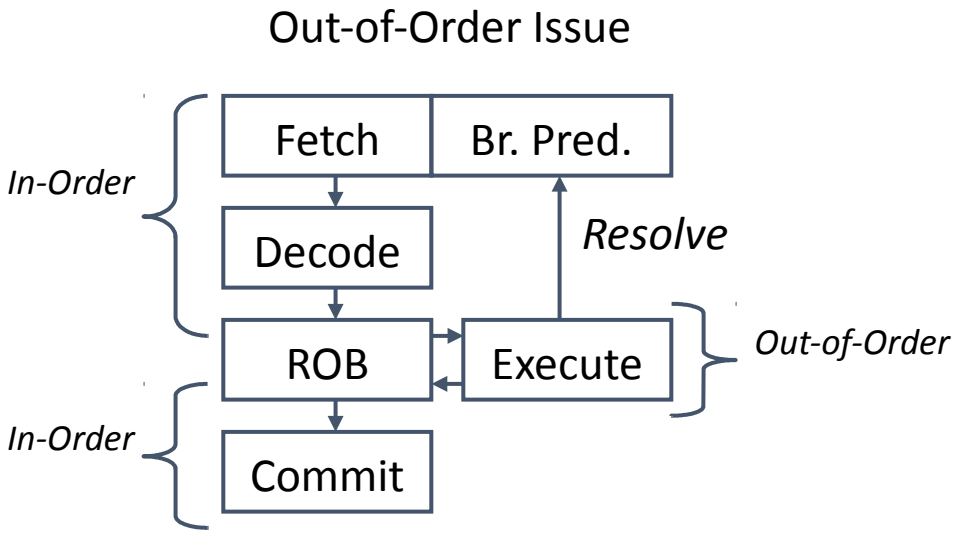
Return Stack in Pipeline

- 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

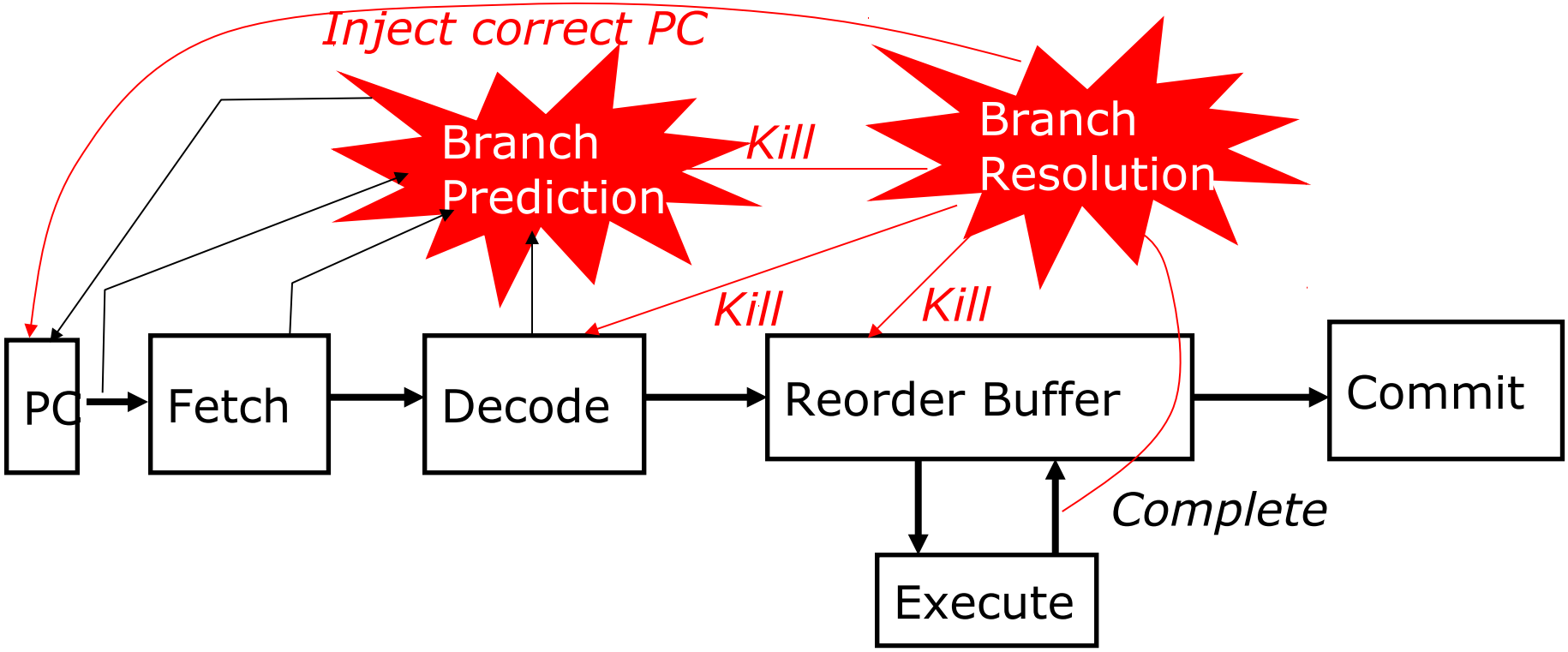
- Both styles of machine can use same branch predictors in front-end fetch pipeline, and both can execute multiple instructions per cycle
- Common to have 10-30 pipeline stages in either style of design

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