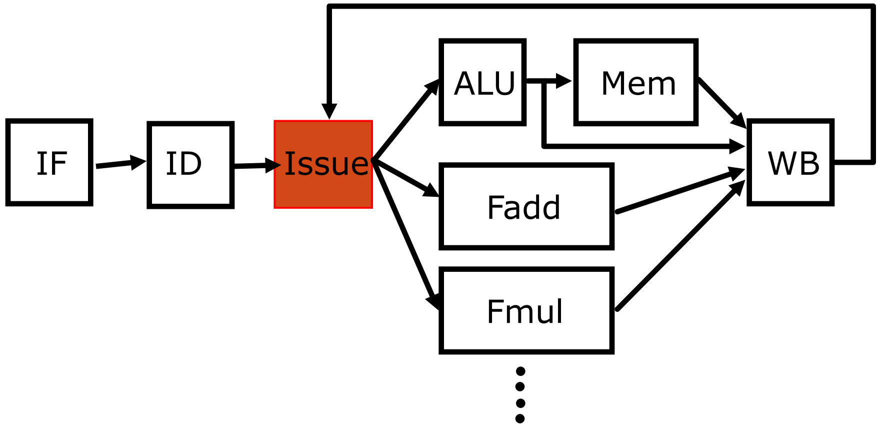
ARCHITECTURE OF COMPUTER SYSTEMS LECTURE 11 - OUT-OF-ORDER ISSUE, REGISTER RENAMING, & BRANCH PREDICTION

LAST TIME IN LECTURE 12

- Pipelining is complicated by multiple and/or variable latency functional units
- Out-of-order and/or pipelined execution requires tracking of dependencies
  - RAW
  - WAR
  - WAW
- Dynamic issue logic can support out-of-order execution to improve performance
  - Last time, looked at simple scoreboard to track out-of-order completion
- Hardware register renaming can further improve performance by removing hazards.

REGISTER RENAMING



- Decode does register renaming and adds instructions to the issue-stage instruction reorder buffer (ROB)

  ⇒ renaming makes WAR or WAW hazards impossible

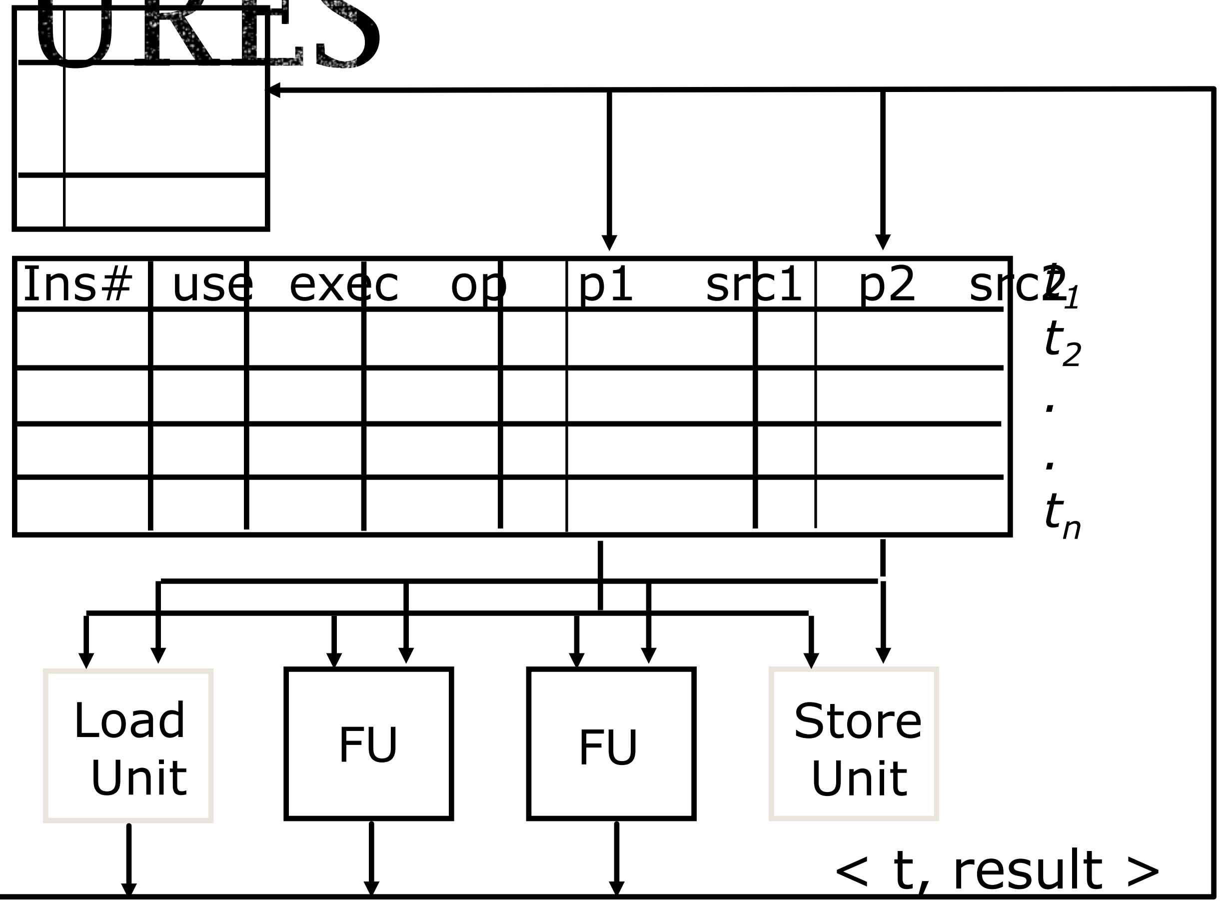
- Any instruction in ROB whose RAW hazards have been satisfied can be issued.

  ⇒ Out-of-order or dataflow execution

RENAMING STRUCTURES

Renaming table & regfile

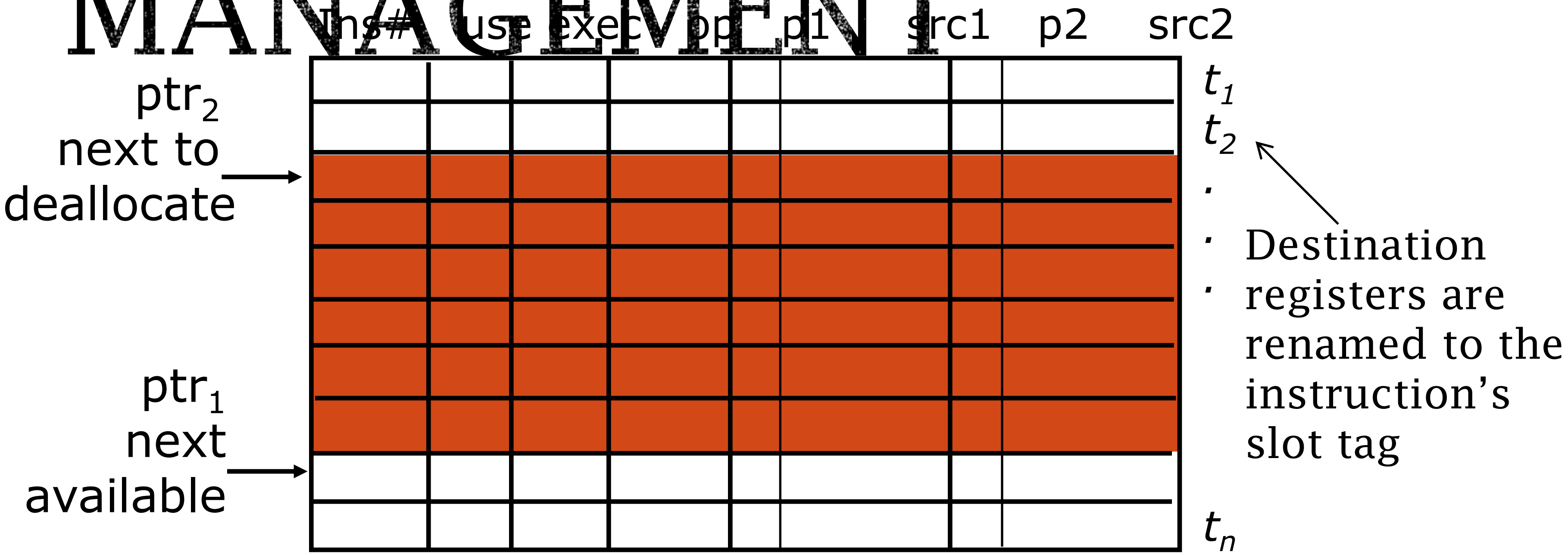
Reorder buffer



Replacing the tag by its value is an expensive operation

- Instruction template (i.e., tag t) is allocated by the Decode stage, which also associates tag with register in regfile
- When an instruction completes, its tag is deallocated

REORDER BUFFER MANAGEMENT



ROB managed circularly

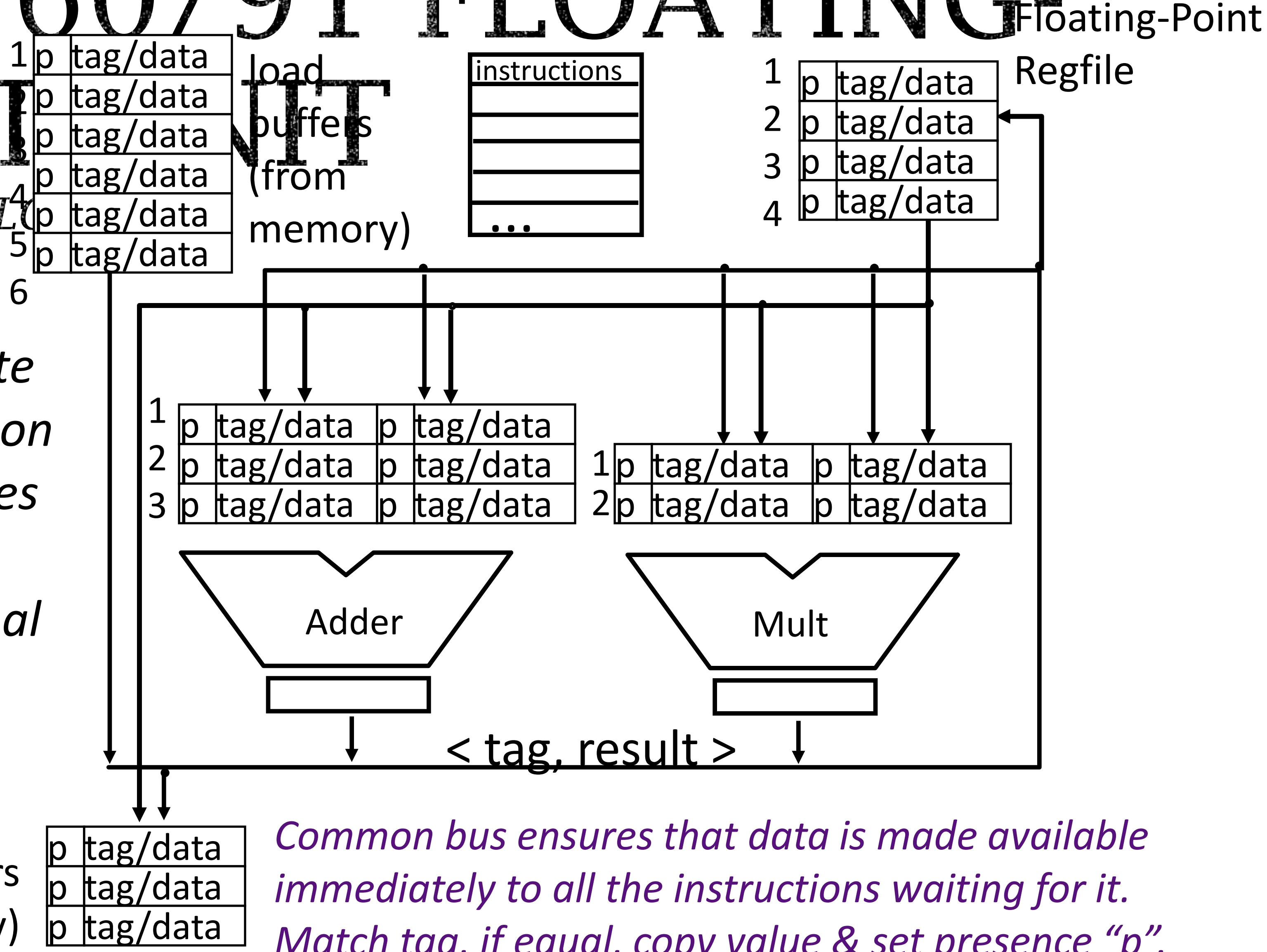
- "exec" bit is set when instruction begins execution
- When an instruction completes its "use" bit is marked free
- ptr_2 is incremented only if the "use" bit is marked free

Instruction slot is candidate for execution when:

- It holds a valid instruction ("use" bit is set)
- It has not already started execution ("exec" bit is clear)
- Both operands are available (p1 and p2 are set)

IBM 360/91 FLOATING-POINT UNIT

R. M. TOMASULO



Distribute instruction templates by functional units

Common bus ensures that data is made available immediately to all the instructions waiting for it. Match tag, if equal, copy value & set presence "p".

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. Effective on a very small class of programs
2. Memory latency a much bigger problem
3. Exceptions not precise!

One more problem needed to be solved

Control transfers

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
I_2	LD	f2, 45(r3)
I_3	MULTD	f0, f2, f4
I_4	DIVD	f8, f6, f2
I_5	SUBD	f10, f0, f6
I_6	ADDD	f6, f8, f2

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

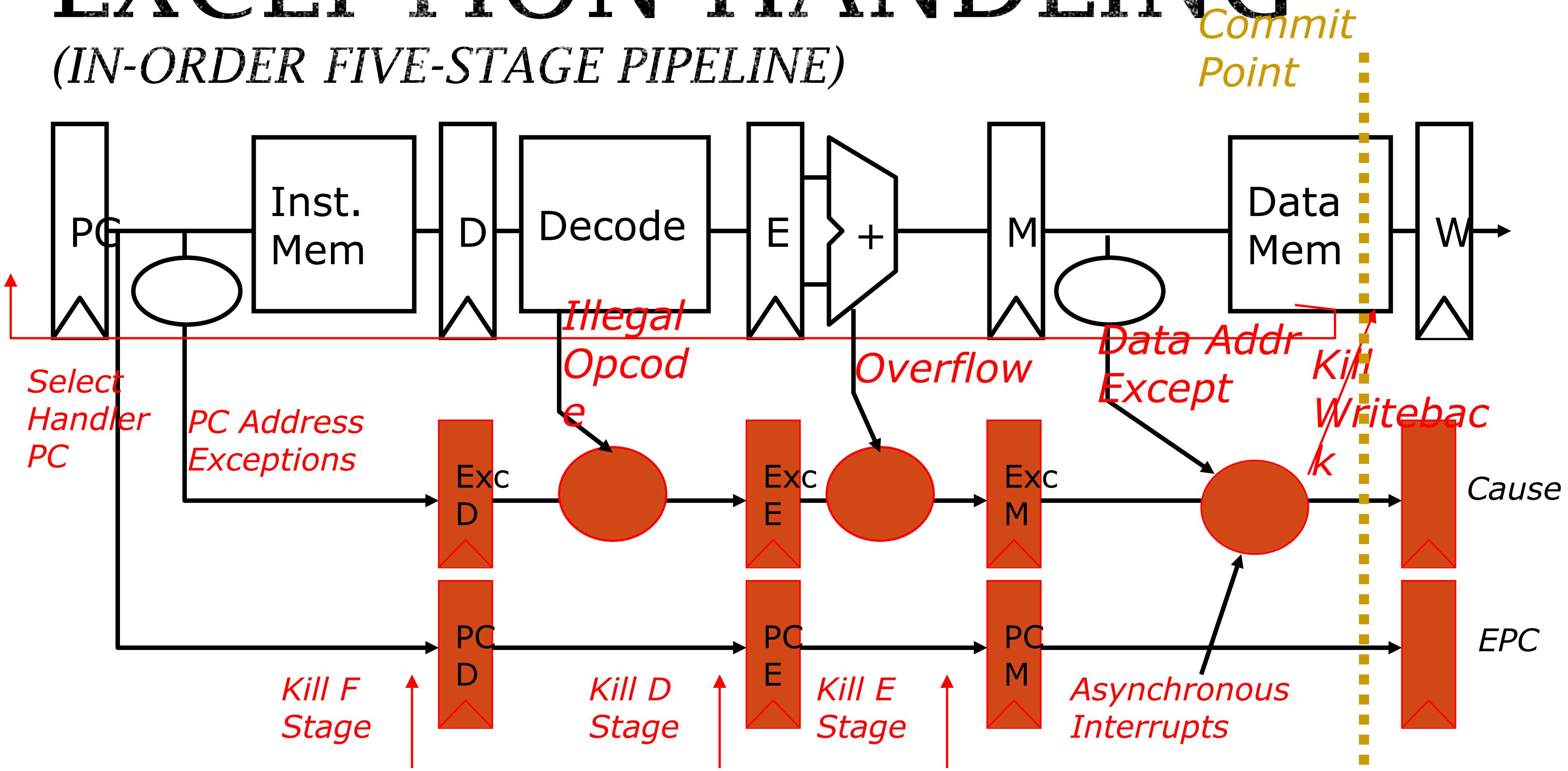
restore f2 *restore f10*

Consider interrupts

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

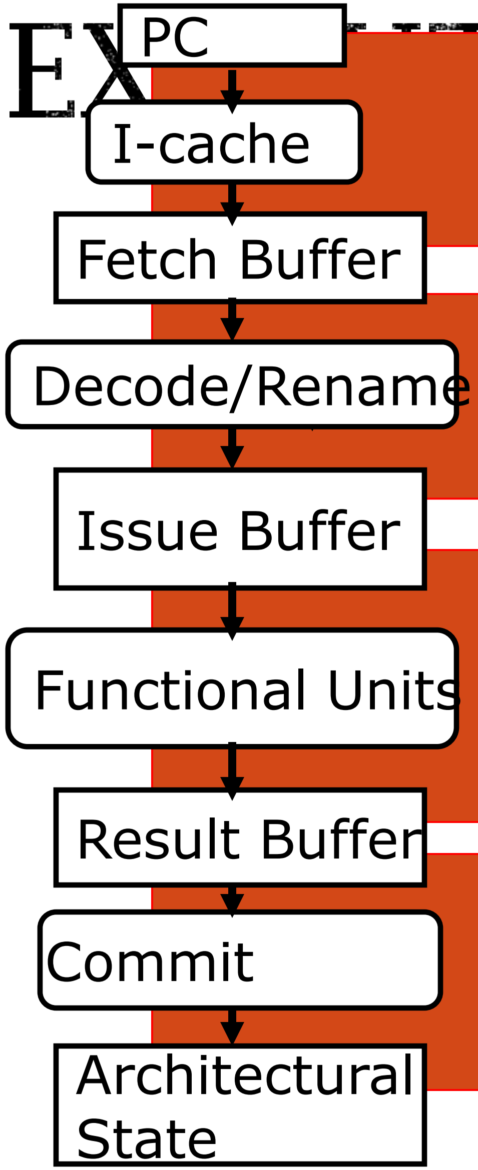
EXCEPTION HANDLING

(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

INSTRUCTION EX



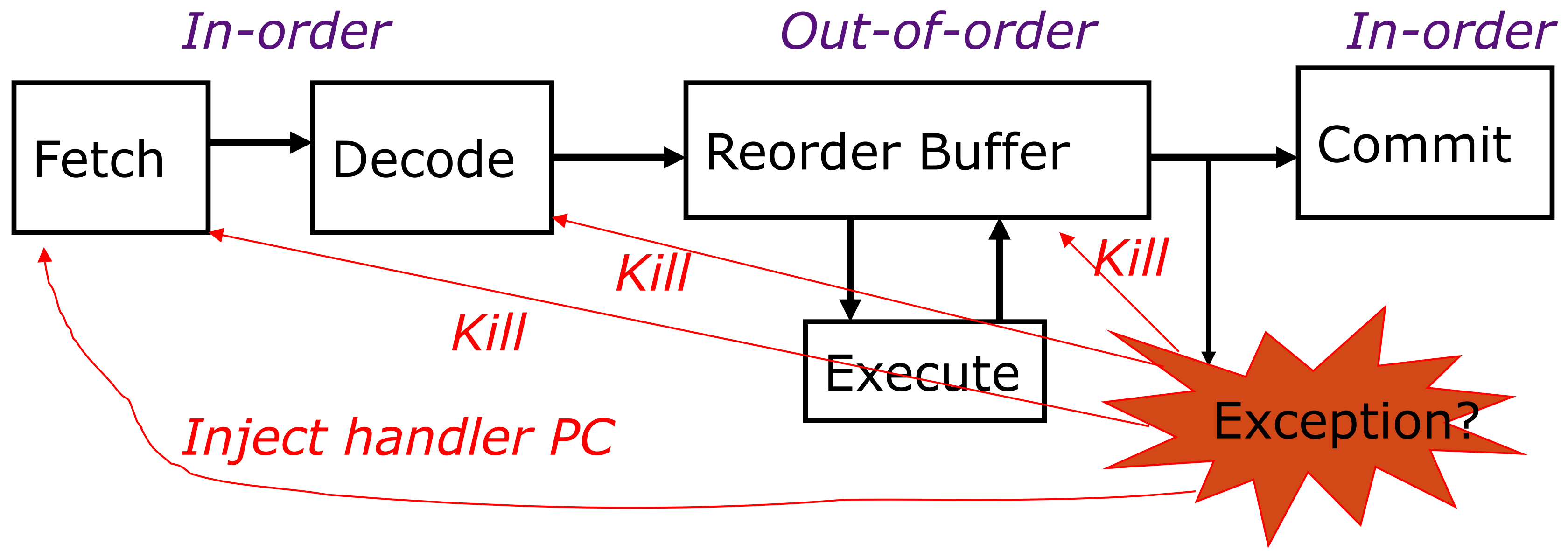
Fetch: Instruction bits retrieved from cache.

Decode: Instructions dispatched to appropriate issue-stage buffer

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

Commit: Instruction irrevocably updates architectural state (aka "graduation").

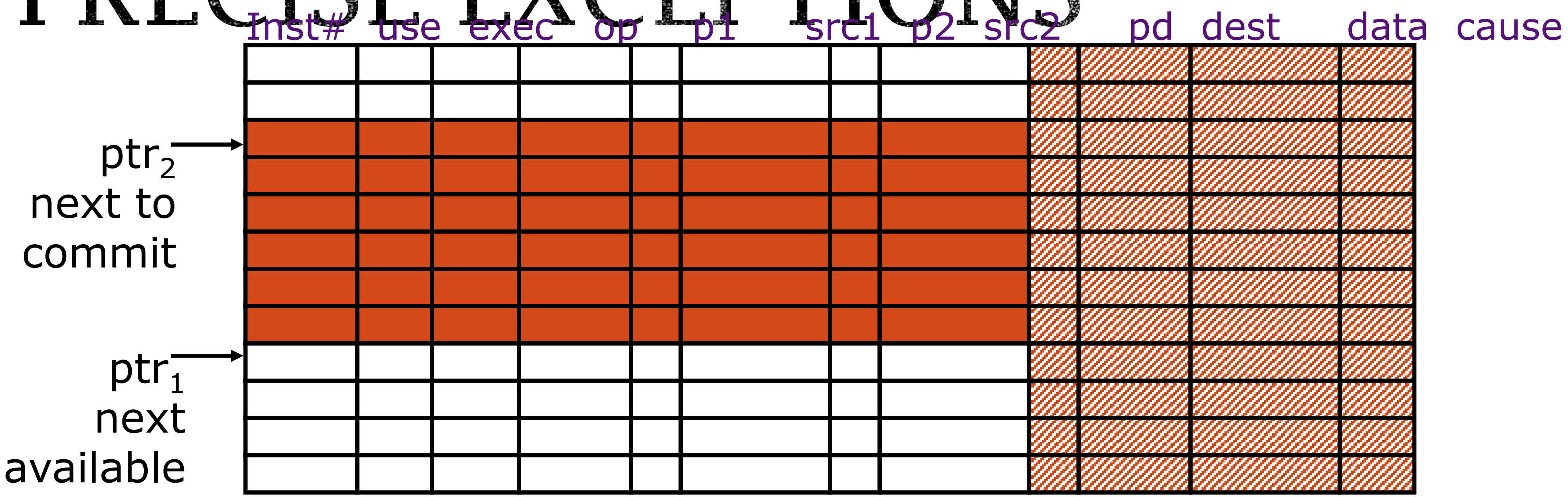
IN-ORDER COMMIT FOR PRECISE EXCEPTIONS



- 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)

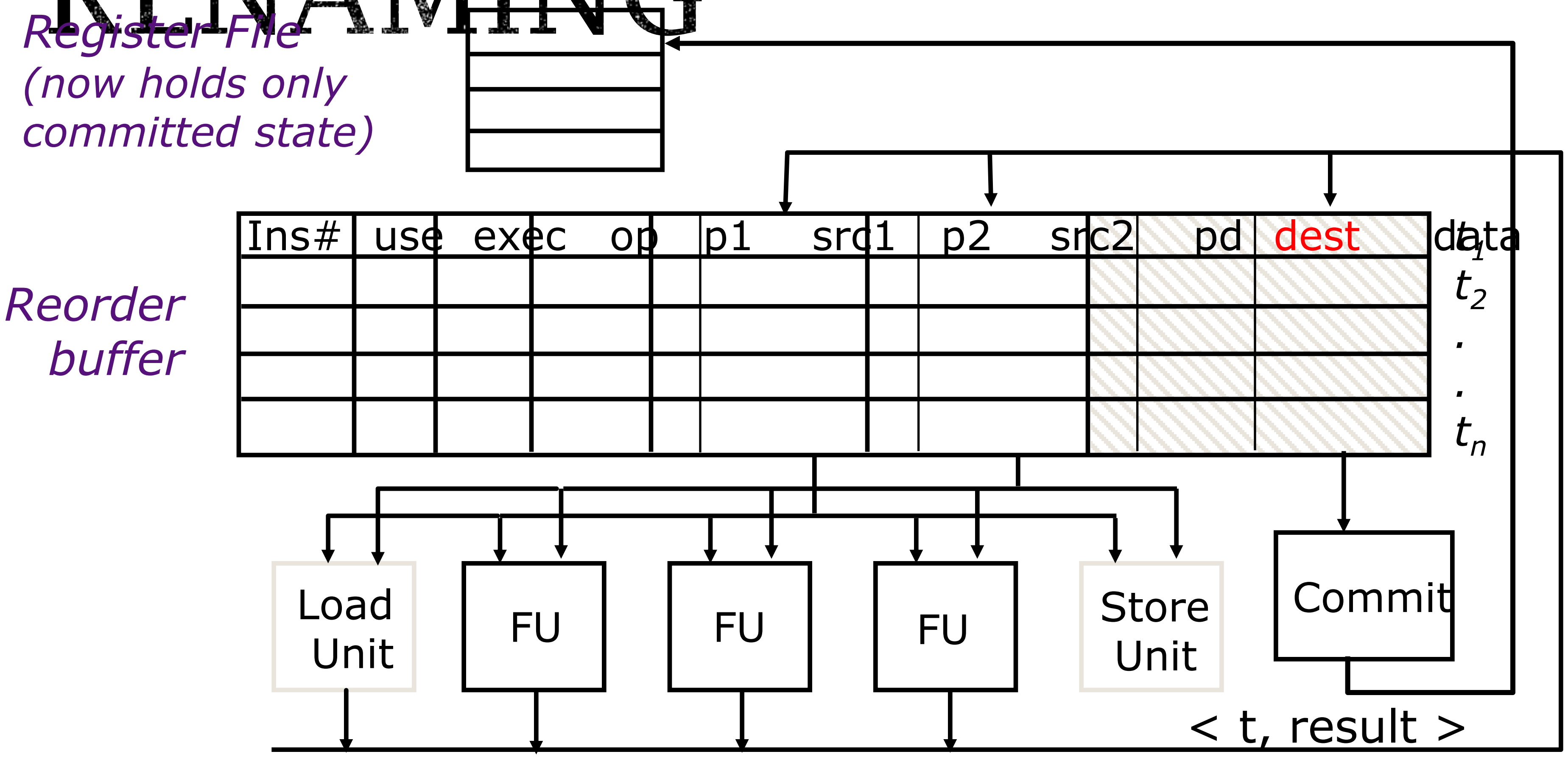
EXTENSIONS FOR PRECISE EXCEPTIONS



Reorder buffer

- add <pd, dest, data, cause> fields in the instruction template
- commit instructions to reg file and memory in program order ⇒ buffers can be maintained circularly
- on exception, clear reorder buffer by resetting ptr_1=ptr_2
  (stores must wait for commit before updating memory)

ROLLBACK AND RENAMING

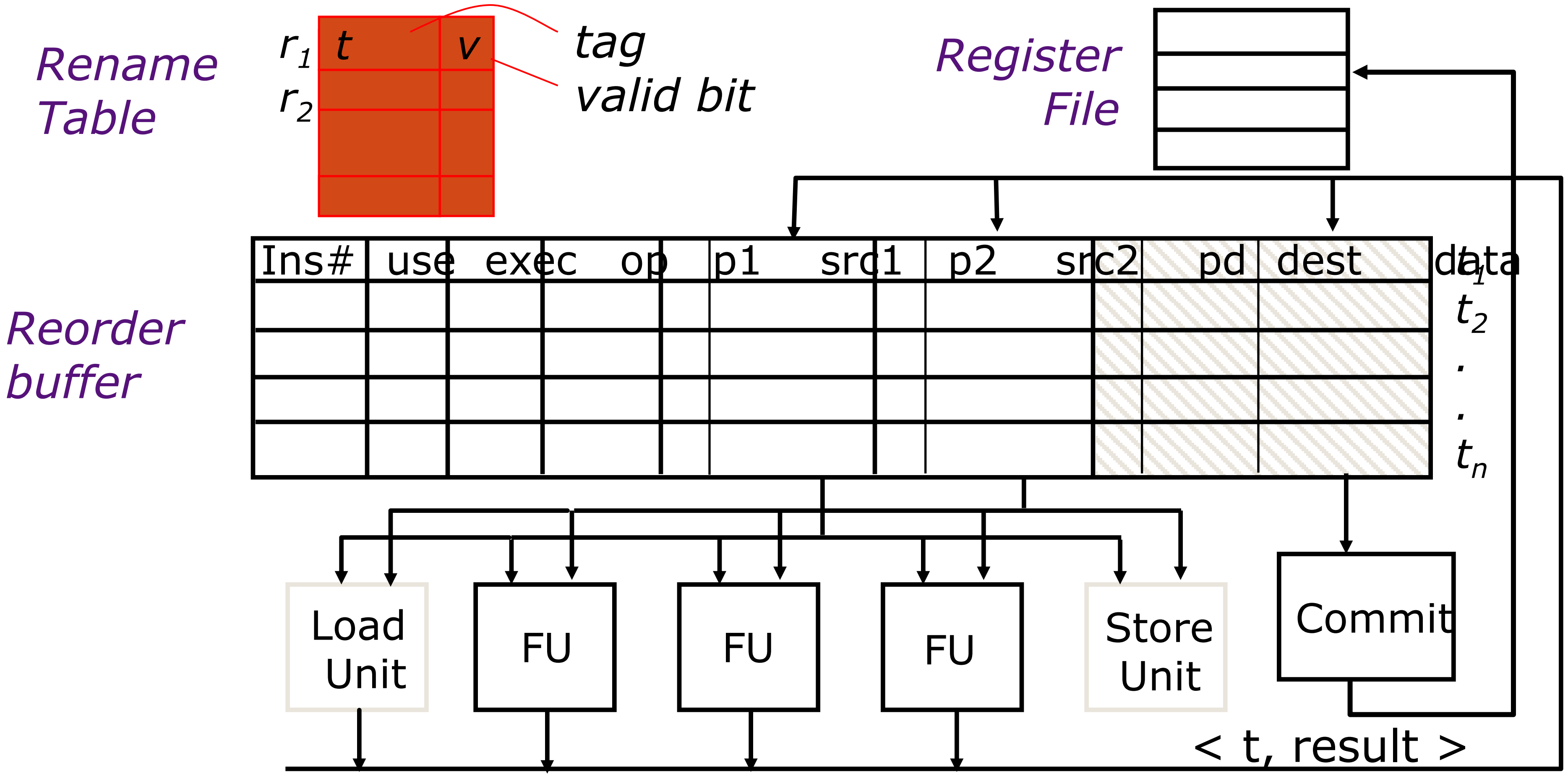


Register file does not contain renaming tags any more.

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

Search the "dest" field in the reorder buffer

RENAMING TABLE



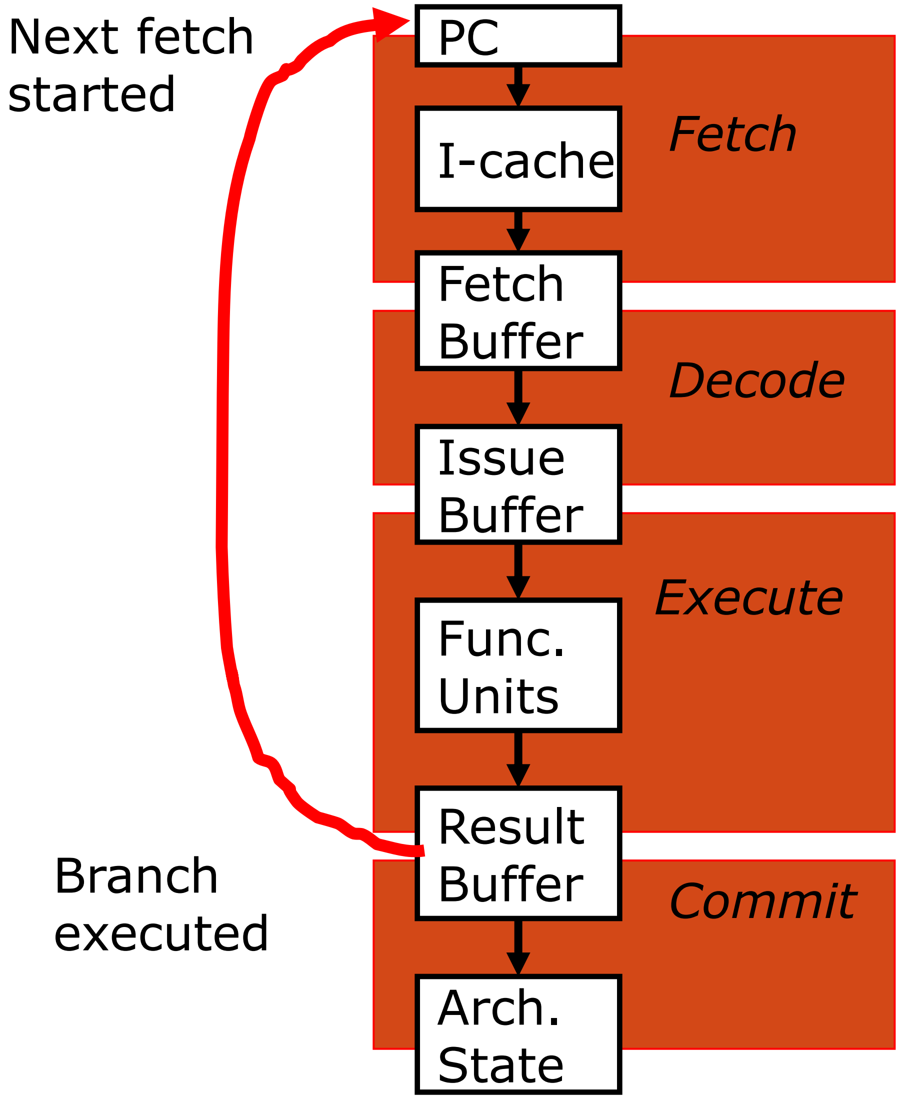
Renaming table is a cache to speed up register name look up.
It needs to be cleared after each exception taken.
When else are valid bits cleared? *Control transfers*

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



CS152 ADMINISTRIVIA

- Quiz 2, Tuesday March 5
  - Caches and Virtual memory L6 – L9, PS 2, Lab 2, readings

MISPREDICT RECOVERY

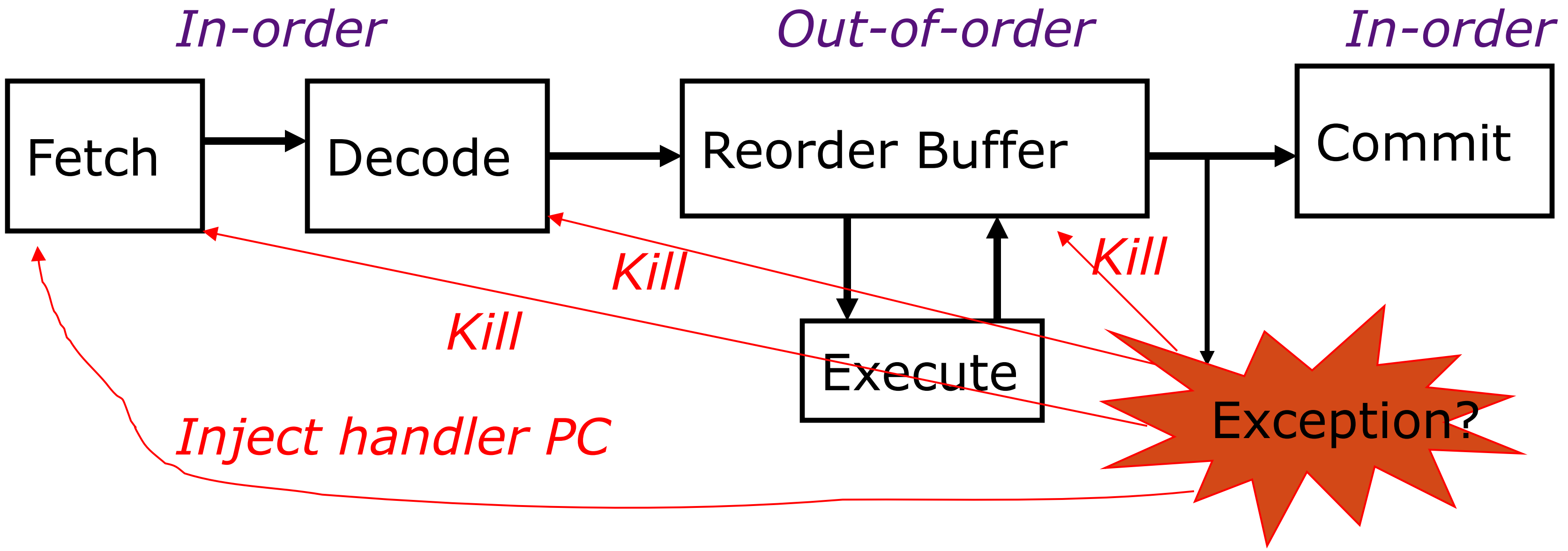
In-order execution machines:

- 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

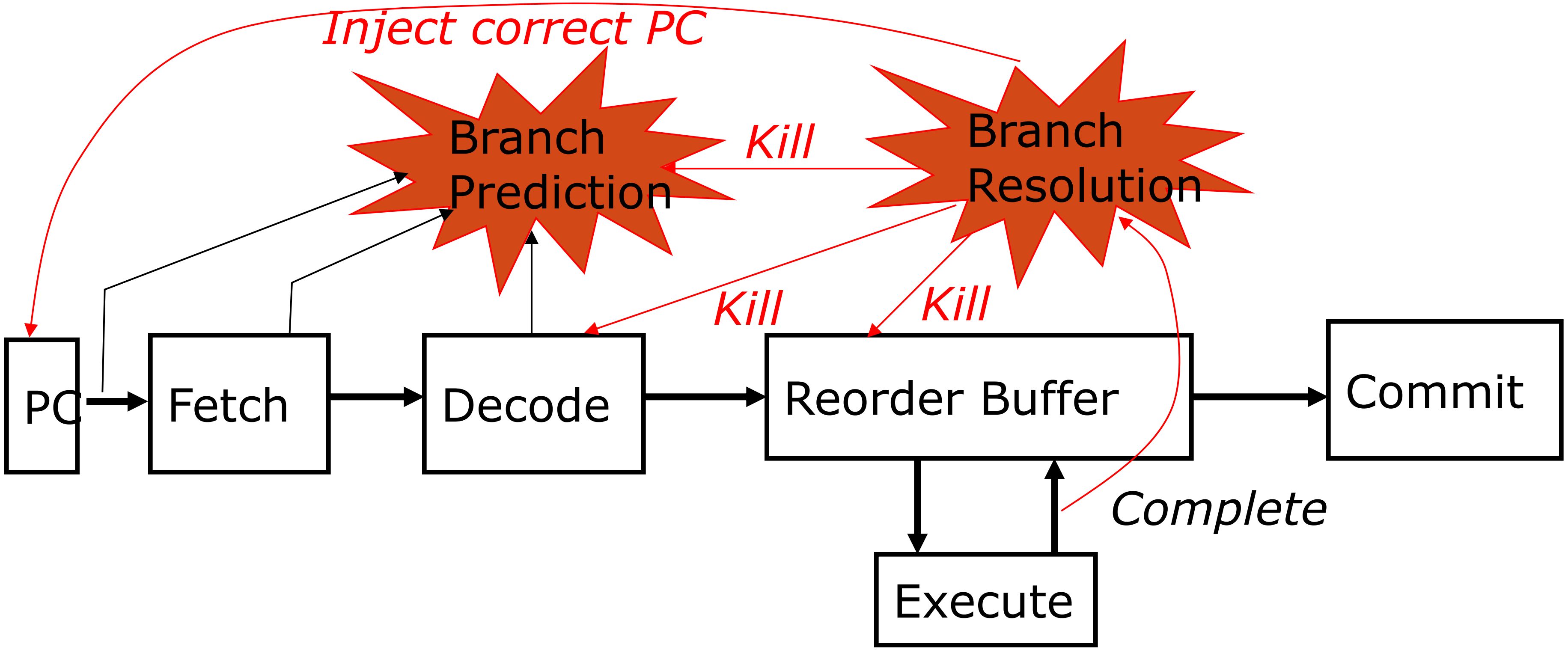
IN-ORDER COMMIT FOR PRECISE EXCEPTIONS



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

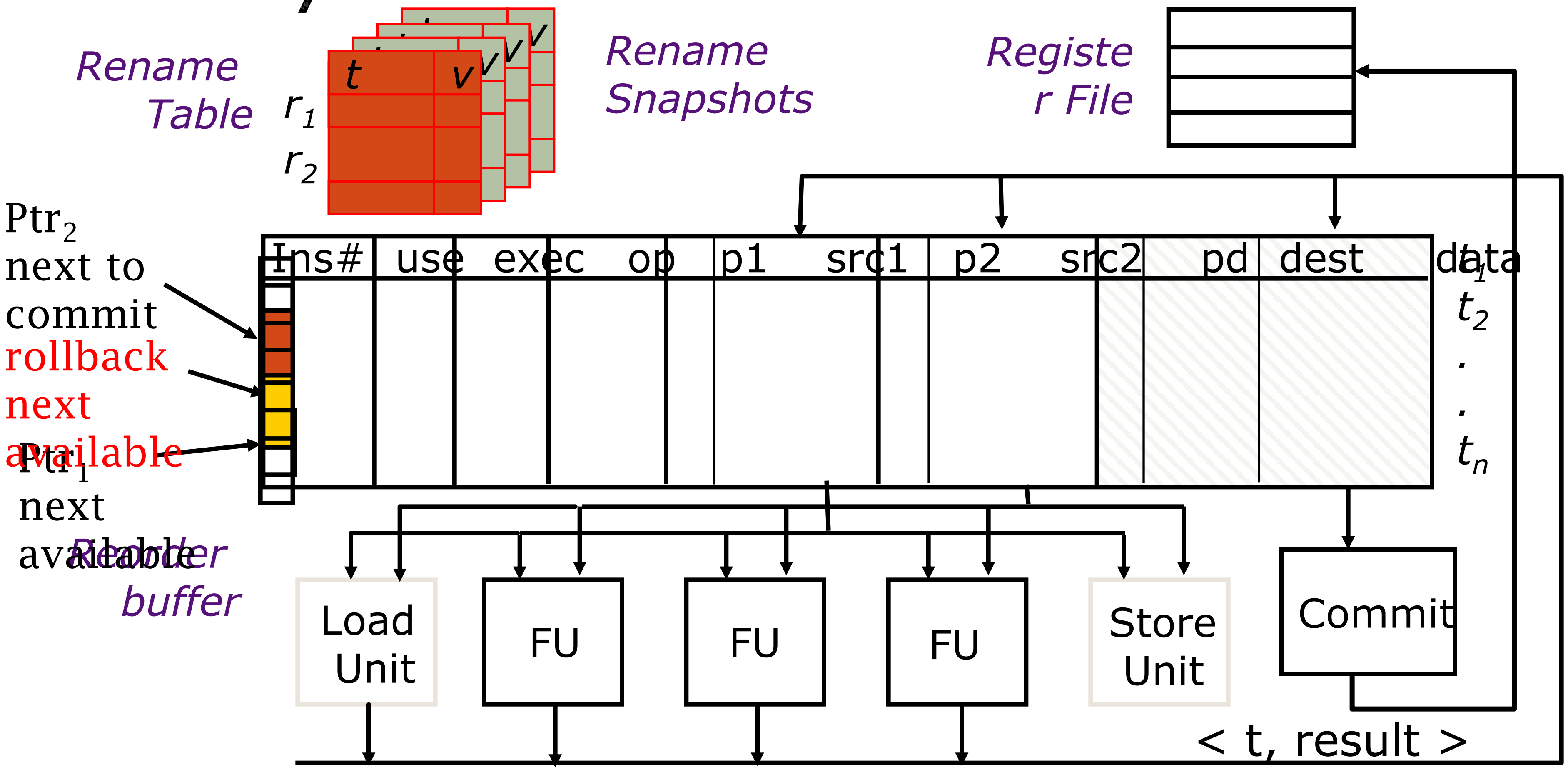
Temporary storage needed in ROB to hold results before commit

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

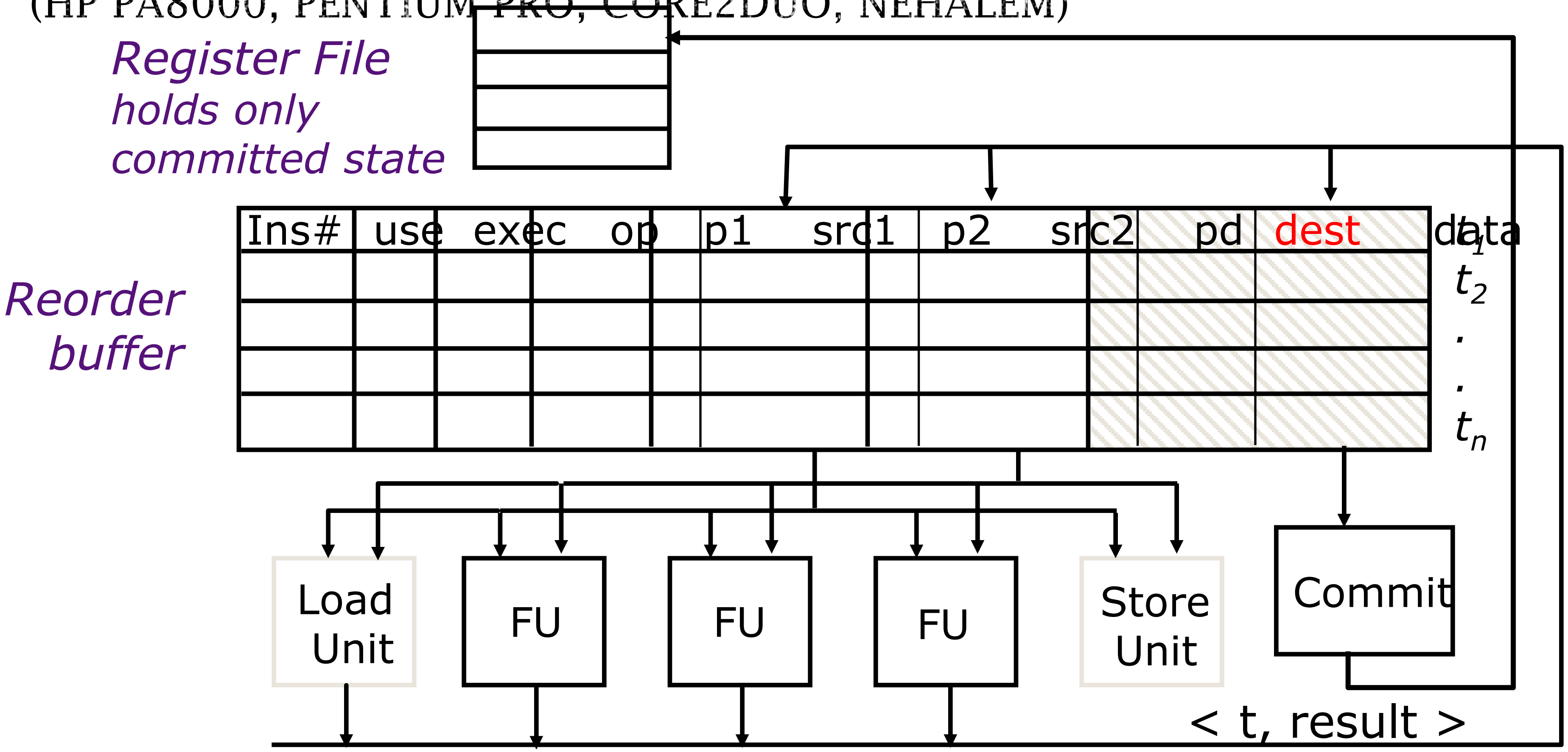
RECOVERING ROB/RENAMING TABLE



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

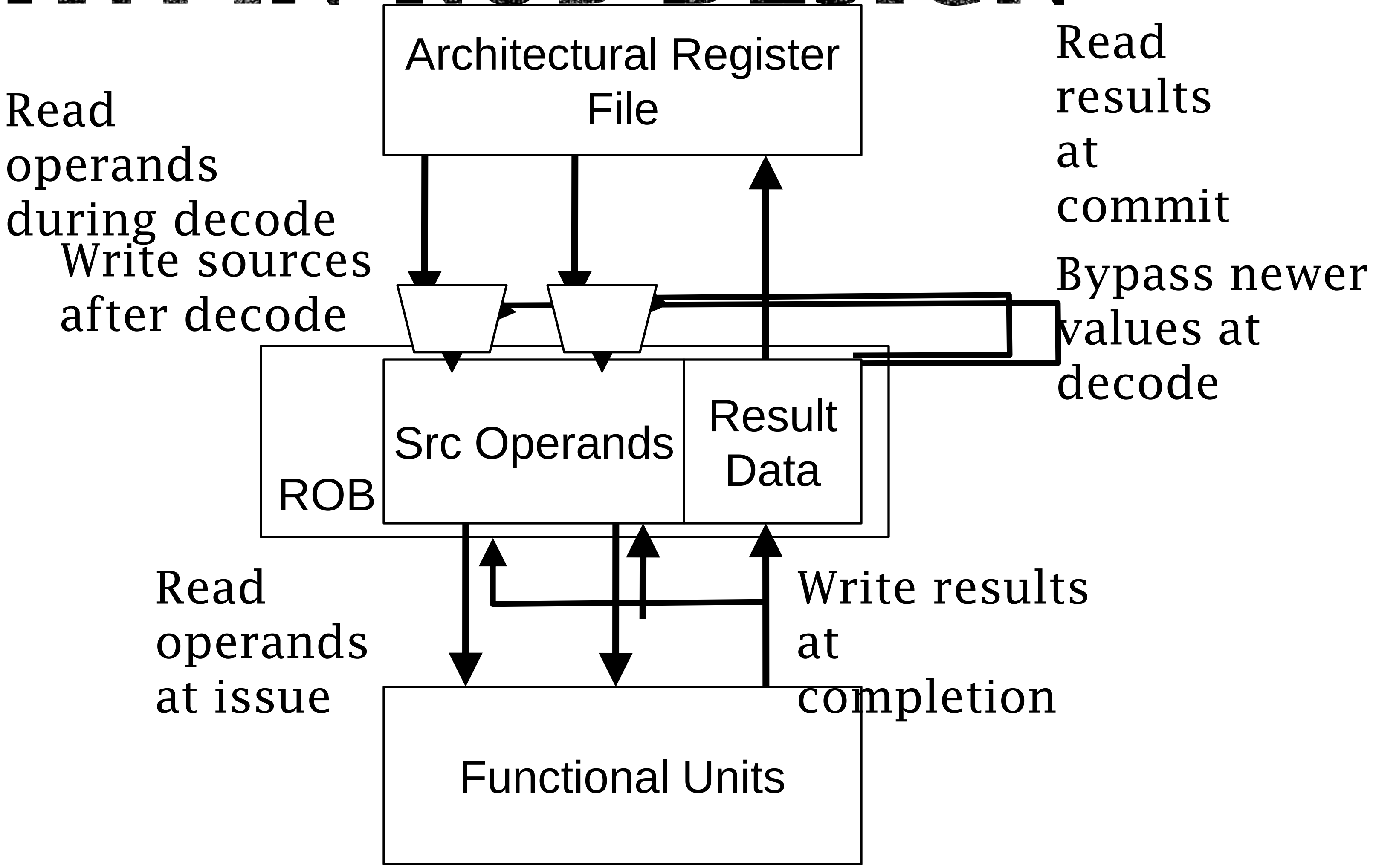
"DATA-IN-ROB" DESIGN

(HP PA8000, PENTIUM PRO, CORE2DUO, NEHALEM)



- 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

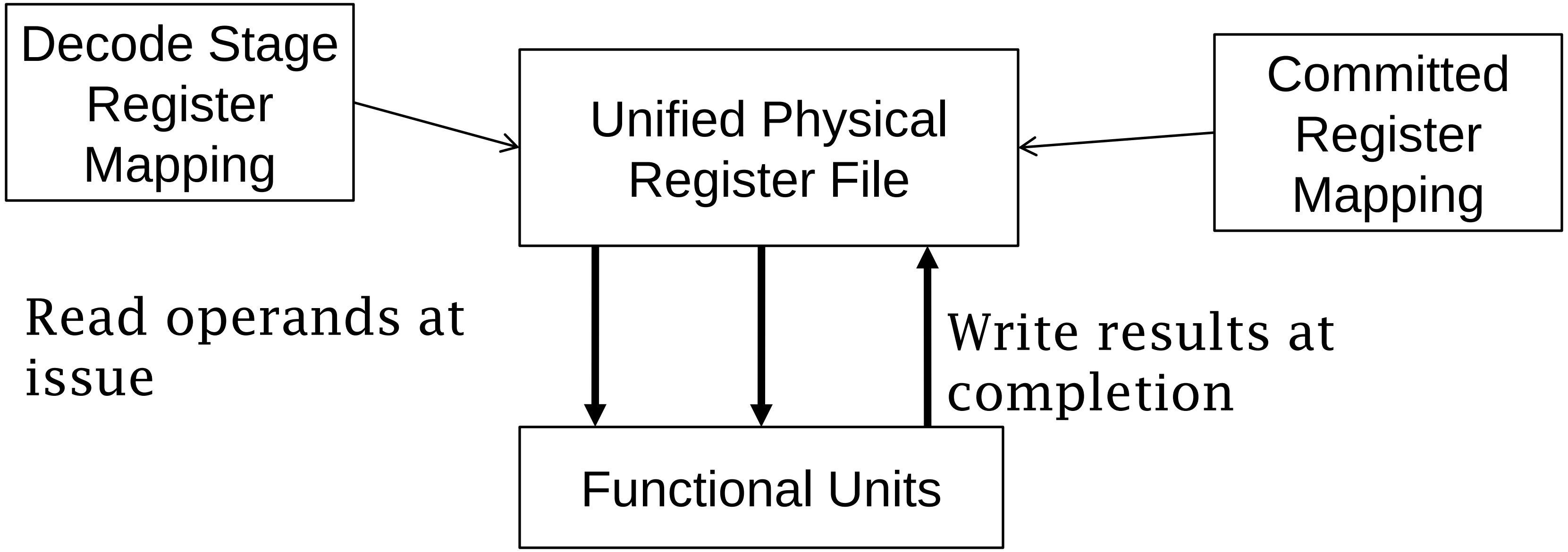
DATA MOVEMENT IN DATA-IN-ROB DESIGN



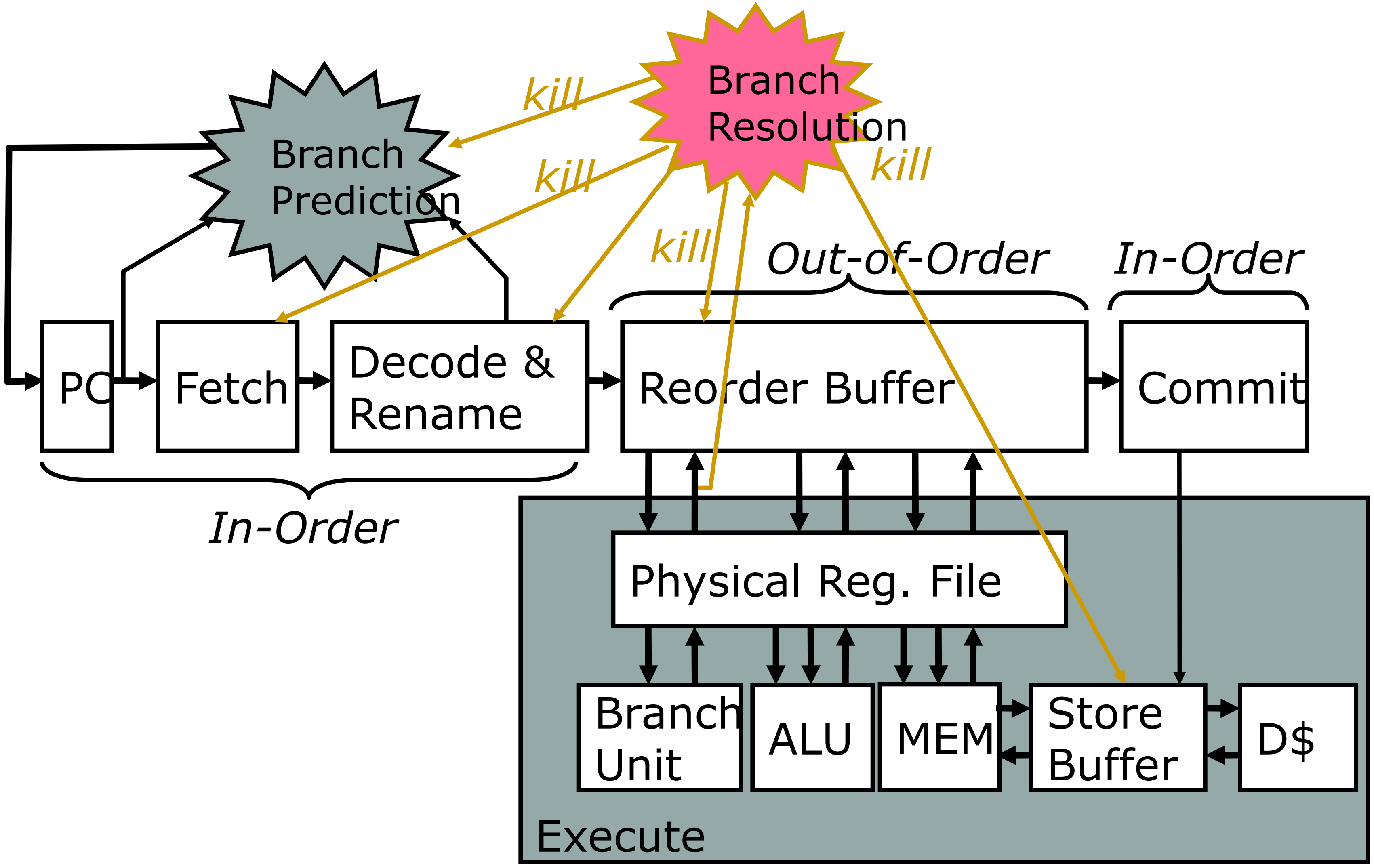
UNIFIED PHYSICAL REGISTER FILE

(MIPS R10K, ALPHA 21264, INTEL PENTIUM 4 & SANDY 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



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 x1, (x3)		ld P1, (P*x*)
addi x3, x1, #4		addi P2, P1, #4
sub x6, x7, x9		sub P3, P*y*, P*z*
add x3, x3, x6	*Rename* →	add P4, P2, P3
ld x6, (x1)		ld P5, (P1)
add x6, x6, x3		add P6, P5, P4
sd x6, (x1)		sd P6, (P1)
ld x6, (x11)		ld P7, (P*w*)

When can we reuse a physical register?

When next write 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

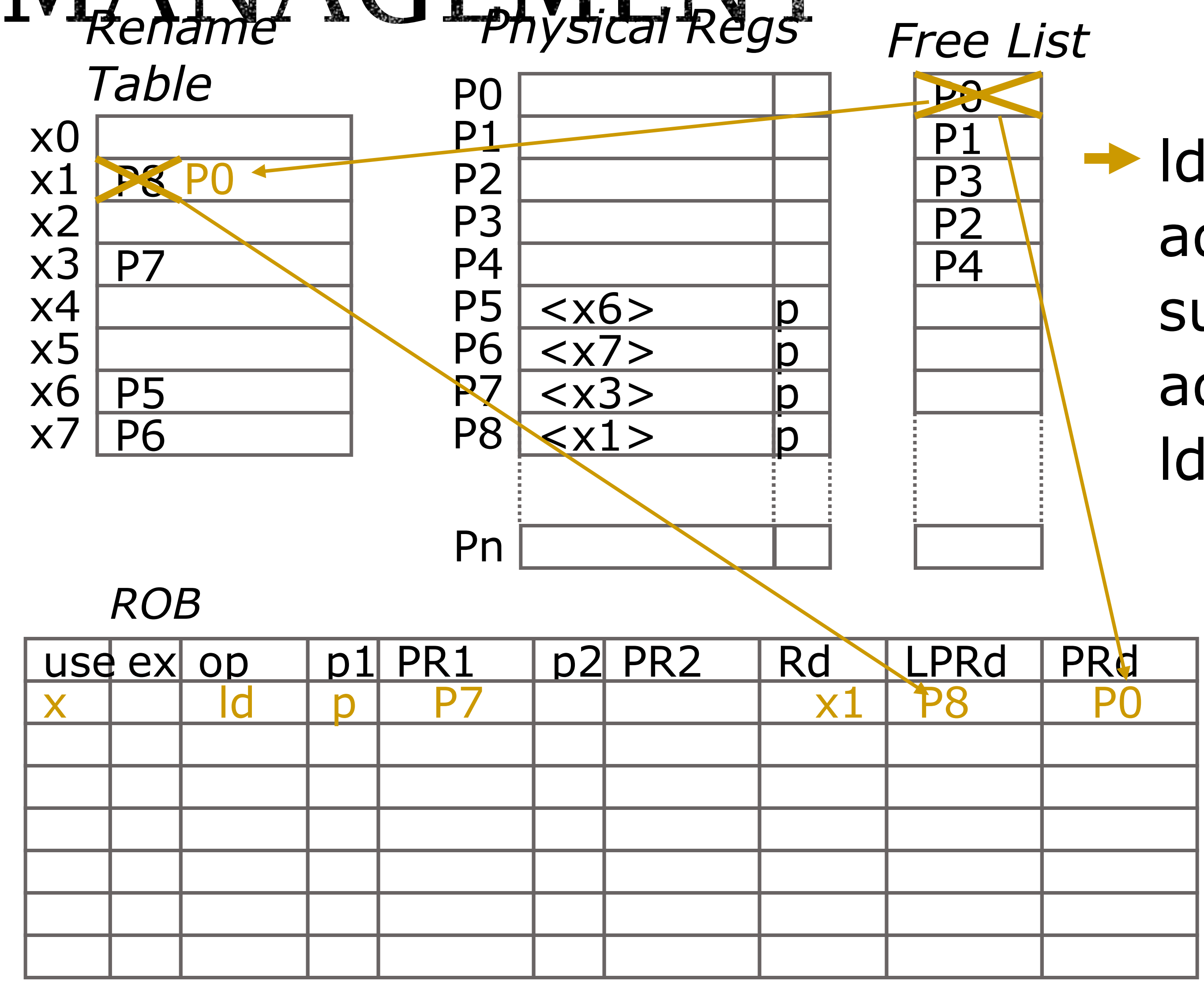
ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)

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



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

PHYSICAL REGISTER MANAGEMENT

Rename Table

x0	
x1	~~P8~~ P0
x2	
x3	~~P7~~ P1
x4	
x5	
x6	P5
x7	P6

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5	<x6>	p
P6	<x7>	p
P7	<x3>	p
P8	<R1>	p
⋮		
Pn		

Free List

~~P0~~
~~P1~~
P3
P2
P4
⋮

ld x1, 0(x3)

→ addi x3, x1, #4

sub x6, x7, x6

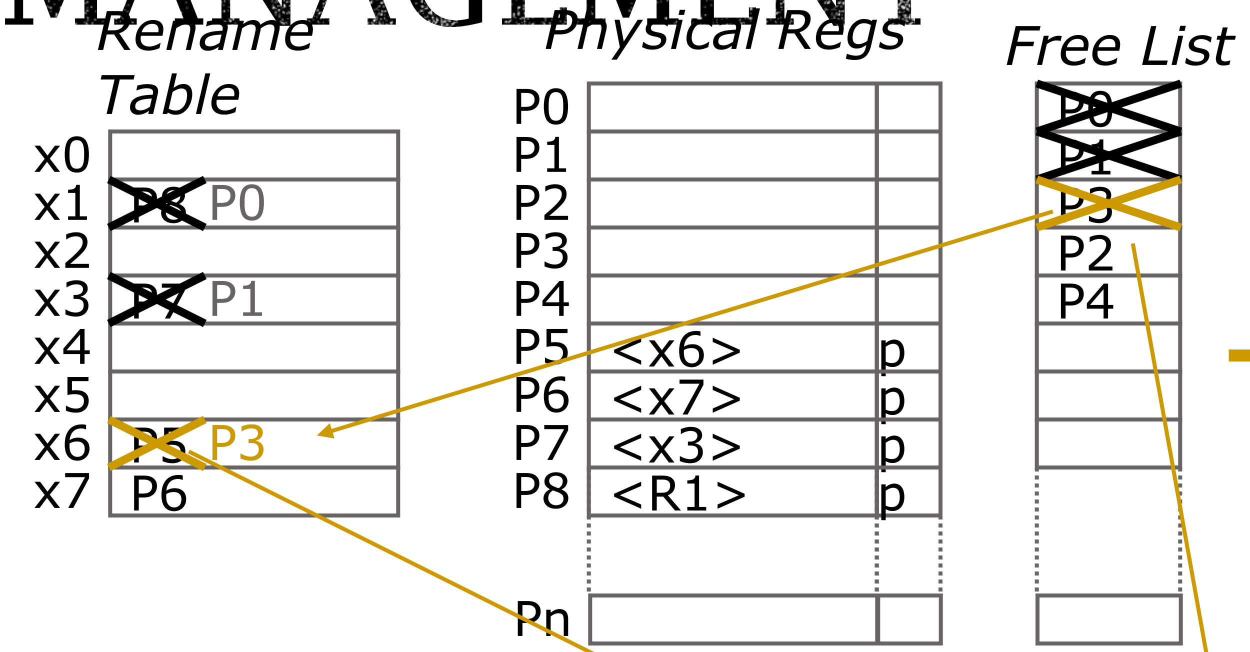
add x3, x3, x6

ld x6, 0(x1)

ROB

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

PHYSICAL REGISTER MANAGEMENT

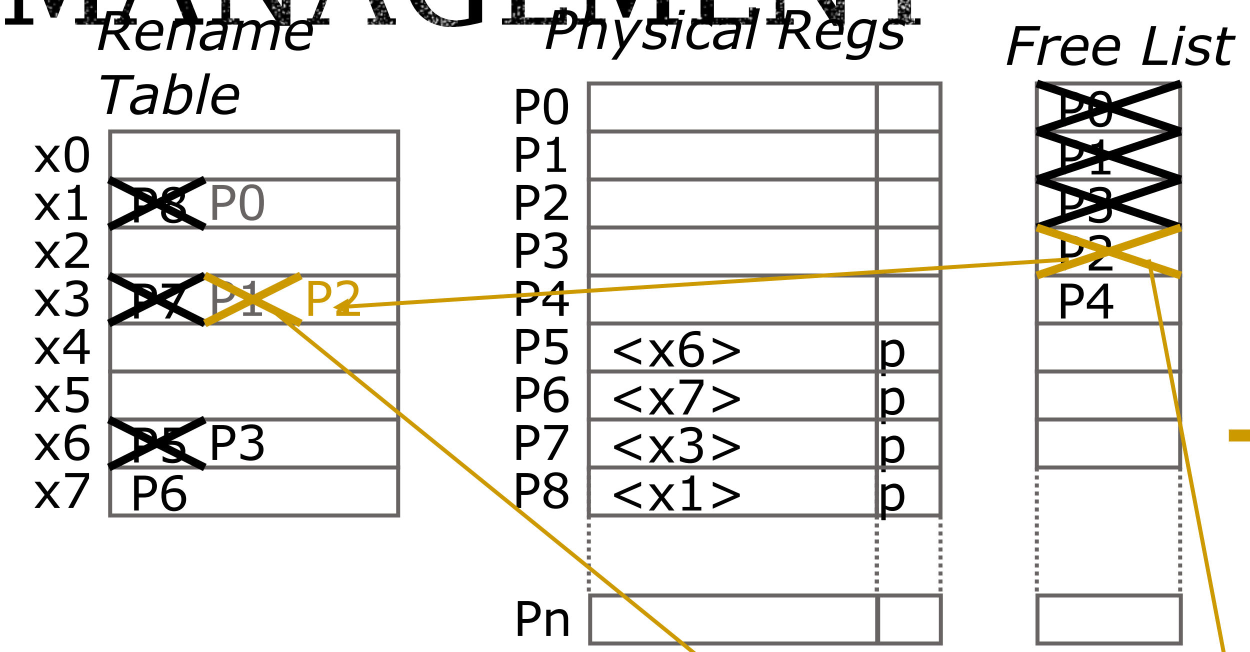


ld x1, 0(x3)
addi x3, x1, #4
→ sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)

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

PHYSICAL REGISTER MANAGEMENT

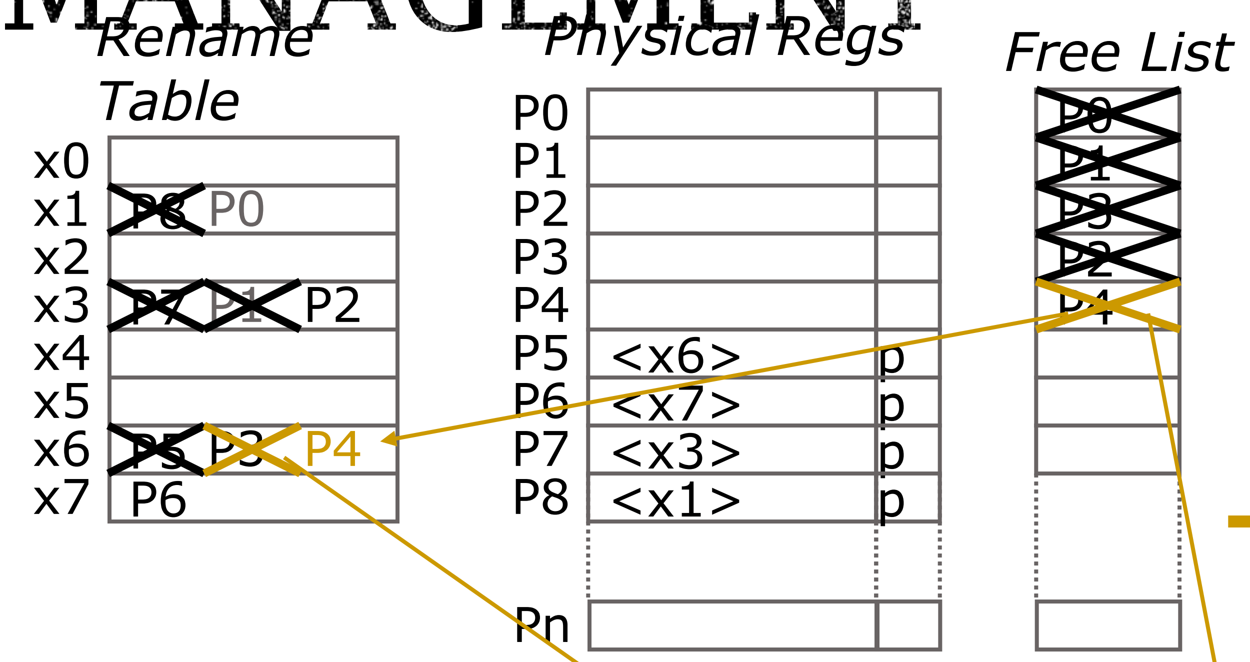


ROB

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

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

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
x		ld		P0			x6	P3	P4

PHYSICAL REGISTER MANAGEMENT

Rename Table

x0	
x1	~~P8~~ P0
x2	
x3	~~P7~~ ~~P1~~ P2
x4	
x5	
x6	~~P5~~ ~~P3~~ P4
x7	P6

Physical Regs

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

Free List

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

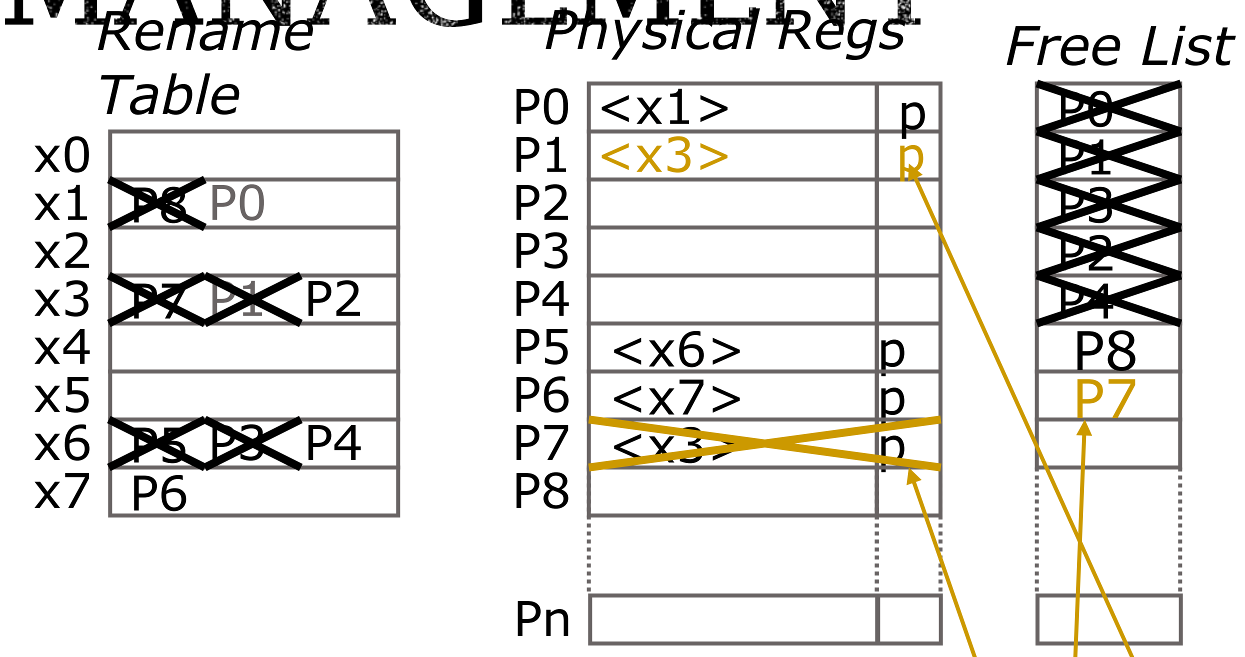
```
ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)
```

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

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