ECE 252 / CPS 220
Advanced Computer Architecture I

Lecture 9
Instruction-Level Parallelism – Part 2

Benjamin Lee
Electrical and Computer Engineering
Duke University

www.duke.edu/~bcl15
www.duke.edu/~bcl15/class/class_ece252fall11.html

ECE252 Administrivia

29 September – Homework #2 Due

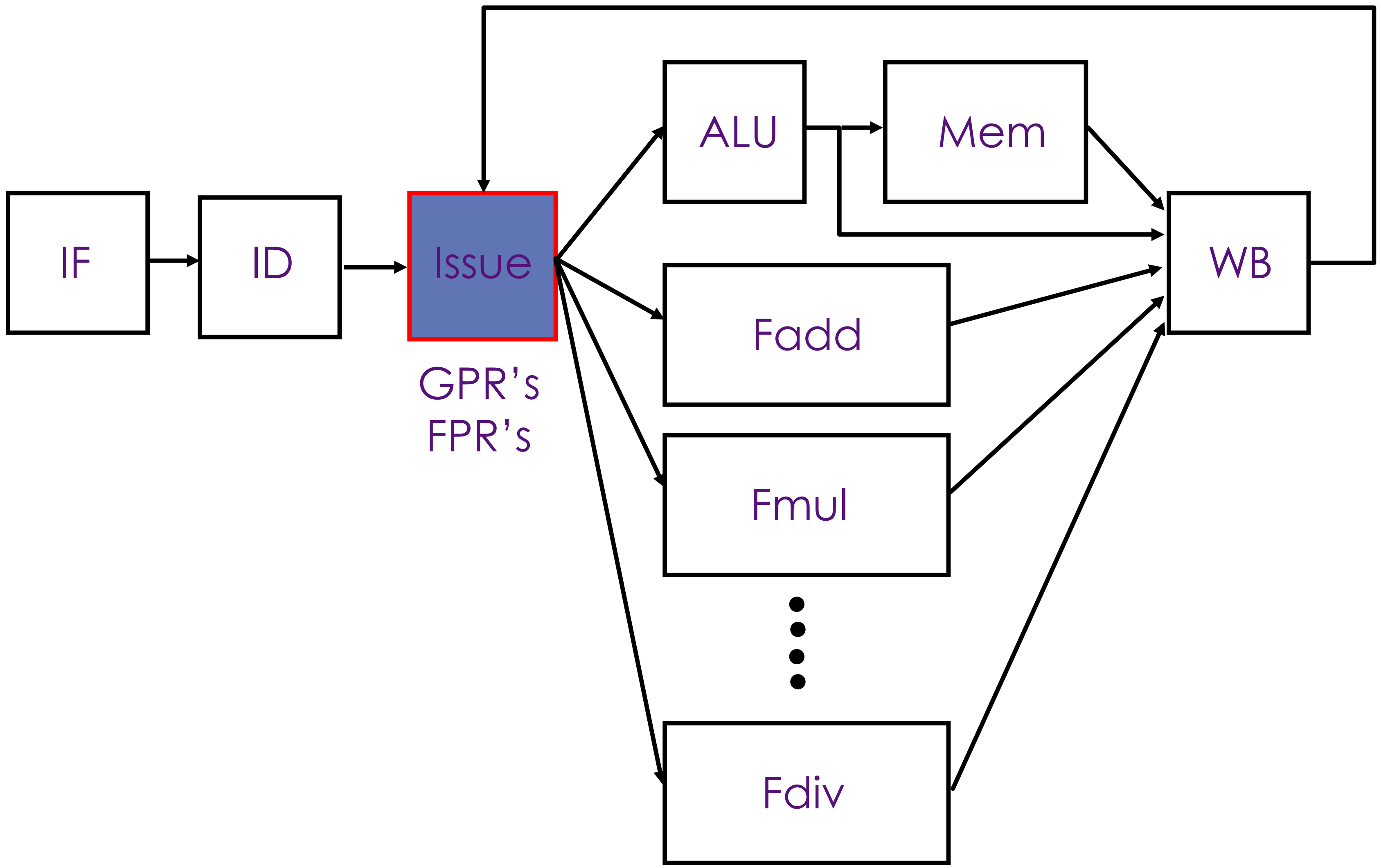
- Use blackboard forum for questions
- Attend office hours with questions
- Email for separate meetings

4 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"

In-Order Issue Pipeline



Scoreboard

Busy[FU#]: a bit-vector to indicate functional unit availability (FU = Int, Add, Mutl, Div)

WP[#regs]: a bit-vector to record the registers to which writes are pending

- Bits are set to true by issue logic
- Bits are set to false by writeback stage
- Each functional unit's pipeline registers must carry 'dest' field and a flag to indicate if it's valid: "the (we, ws) pair"

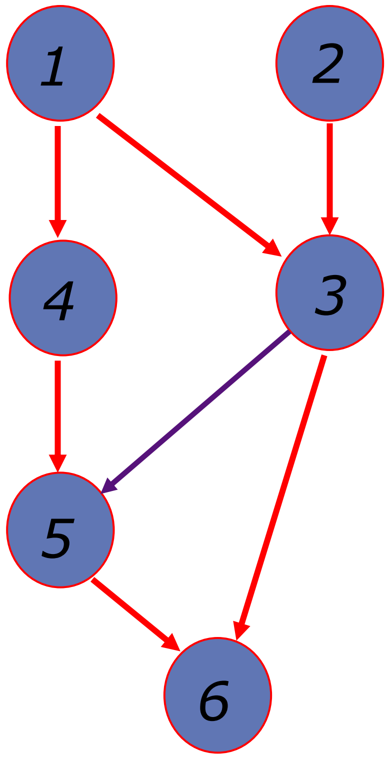
Issue logic checks instruction (opcode, dest, src1, src2) against scoreboard (busy, wp) to dispatch

- FU available?	Busy[FU#]
- RAW?	WP[src1] or WP[src2]
- WAR?	Cannot arise
- WAW?	WP[dest]

Limitations of In-Order Issue

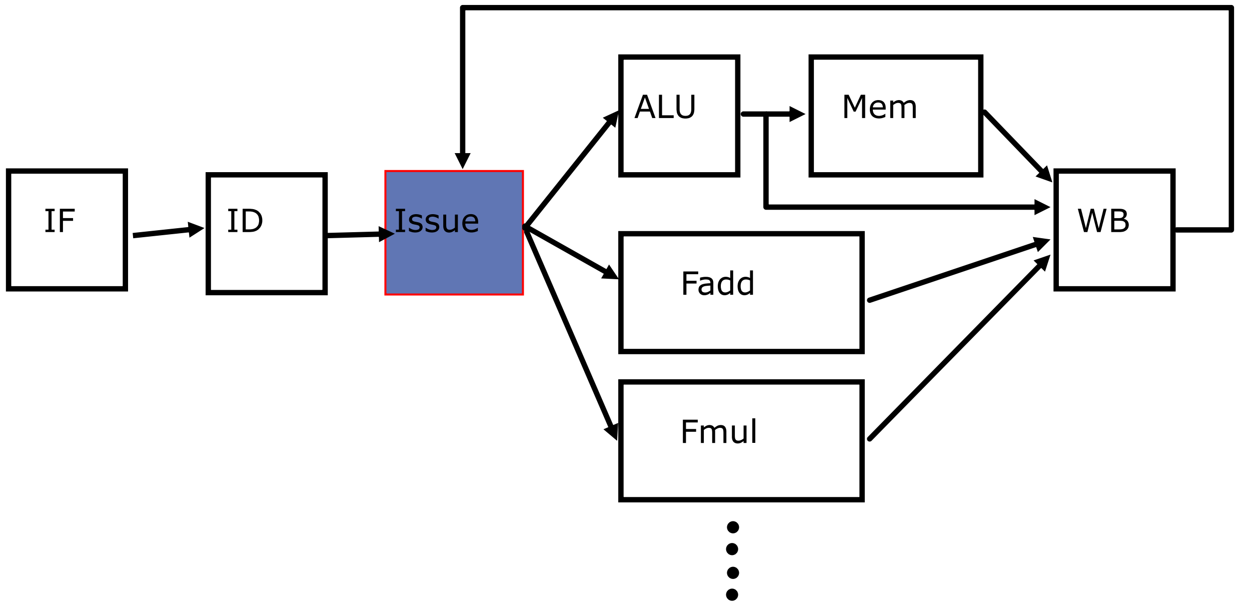
Instruction	Operands	Latency
1: LD	F2, 34(R2)	1
2: LD	F4, 45(R3)	long
3: MULTD	F6, F4, F2	3
4: SUBD	F8, F2, F2	1
5: DIVD	F4, F2, F8	4
6: ADDD	F10, F6, F4	1

In-order: 1 (2 1) ....2 3 4 4 3 5 ....5 6 6



In-order restriction keeps instruction 4 from issuing

Out-of-Order Issue



- Issue stage buffer holds multiple instructions waiting to issue
- Decode stage adds next instruction to buffer if there is space and the instruction does not cause a WAR or WAW hazard
- Any instruction in buffer whose RAW hazards are satisfied can issue
- When instruction commits in write-back stage, a new instruction can issue

Limitations of Out-of-Order Issue

Instruction	Operands	Latency
1: LD	F2, 34(R2)	1
2: LD	F4, 45(R3)	long
3: MULTD	F6, F4, F2	3
4: SUBD	F8, F2, F2	1
5: DIVD	F4, F2, F8	4
6: ADDD	F10, F6, F4	1

$$\begin{tikzcd}
1 \arrow[d] \arrow[drr] & & 2 \arrow[d] \\
4 \arrow[d] & & 3 \arrow[dll] \arrow[ddl] \\
5 \arrow[dr] & & \\
& 6 &
\end{tikzcd}$$

In-order:	1 (2 $\underline{1}$) ....$\underline{2}$ 3 4 $\underline{4}$ $\underline{3}$ 5 ....$\underline{5}$ 6 $\underline{6}$
Out-of-order:	1 (2 $\underline{1}$) 4 $\underline{4}$ ....$\underline{2}$ 3...... $\underline{3}$ 5 ....$\underline{5}$ 6 $\underline{6}$

Out-of-order execution has no gain.

Why did we not issue instruction 5?

Instructions In-Flight

What features of an ISA limit the number of instructions in the pipeline? Number of registers

What features of a program limit the number of instructions in the pipeline? Control transfers

Out-of-order issue does not address these other limitations.

Mitigating Limited Register Names

Floating point pipelines often cannot be filled with small number of registers

- IBM 360 had only 4 floating-point registers

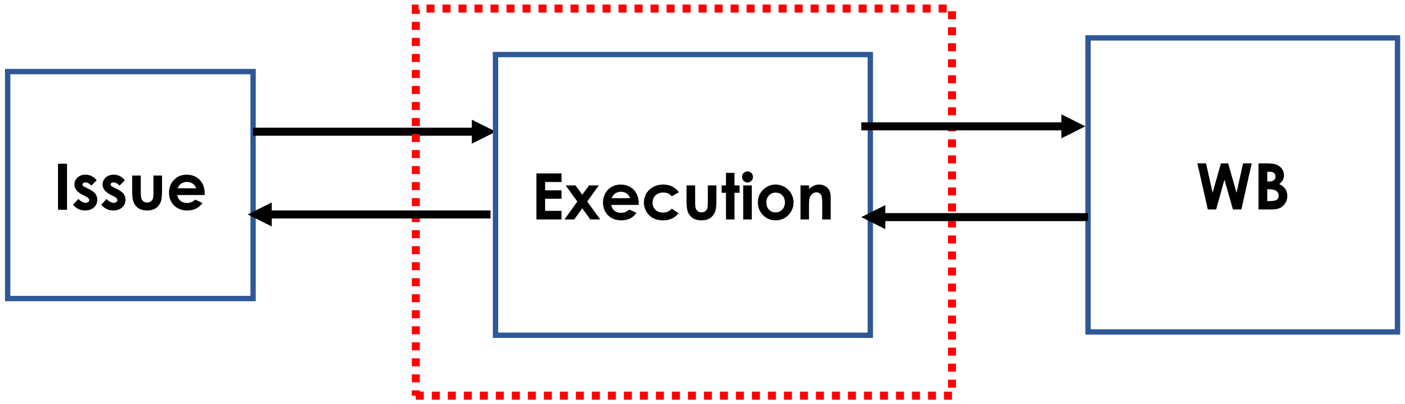
Can a microarchitecture use more registers than specified by the ISA without loss of ISA compatibility?

- In 1967, Robert Tomasulo's solution was dynamic register renaming.

Little's Law

Throughput (T) = Number-in-flight (N) / Latency (L)

- Example: 4 floating-point registers, 8 cycles per floating-point op
- Little's Law → ½ issue per cycle

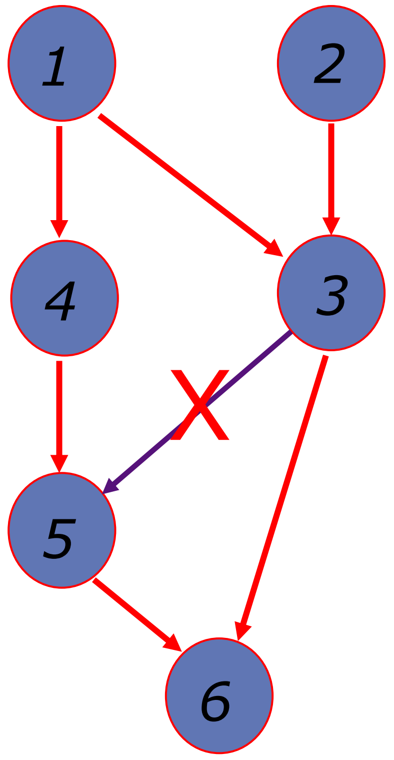


ILP via Renaming

Instruction	Operands	Latency
1: LD	F2, 34(R2)	1
2: LD	F4, 45(R3)	long
3: MULTD	F6, F4, F2	3
4: SUBD	F8, F2, F2	1
5: DIVD	F4, F2, F8	4
6: ADDD	F10, F6, F4	1

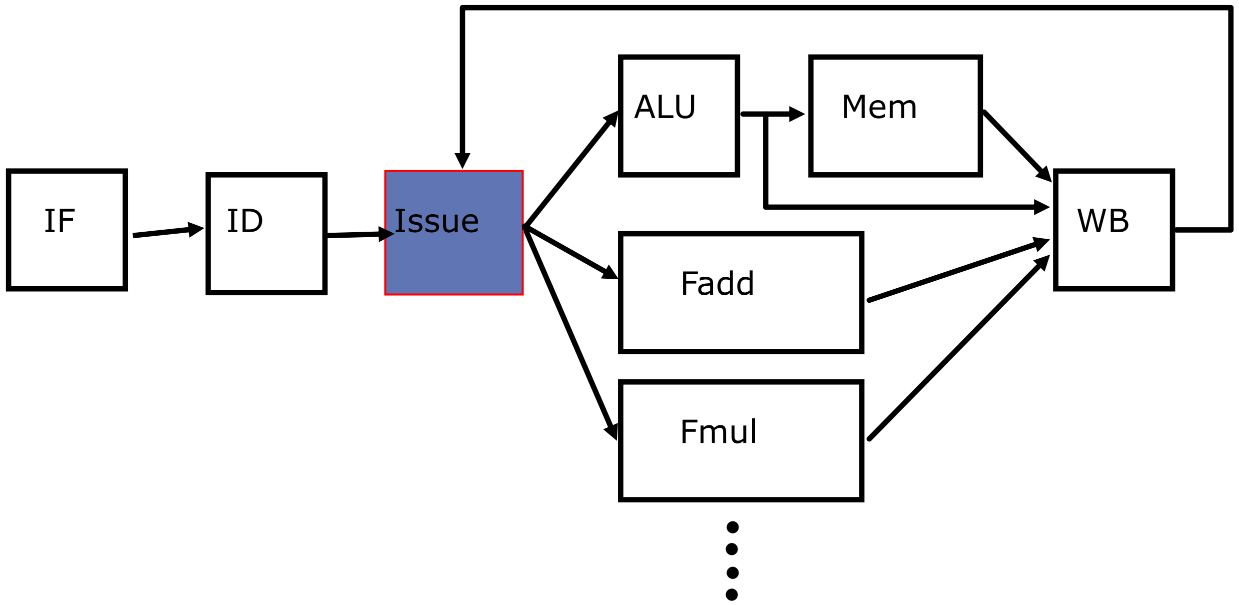
In-order: 1 (2 1) ....2 3 4 4 3 5 ....5 6 6

Out-of-order: 1 (2 1) 4 4 5 ....2 (3, 5) 3 6 6



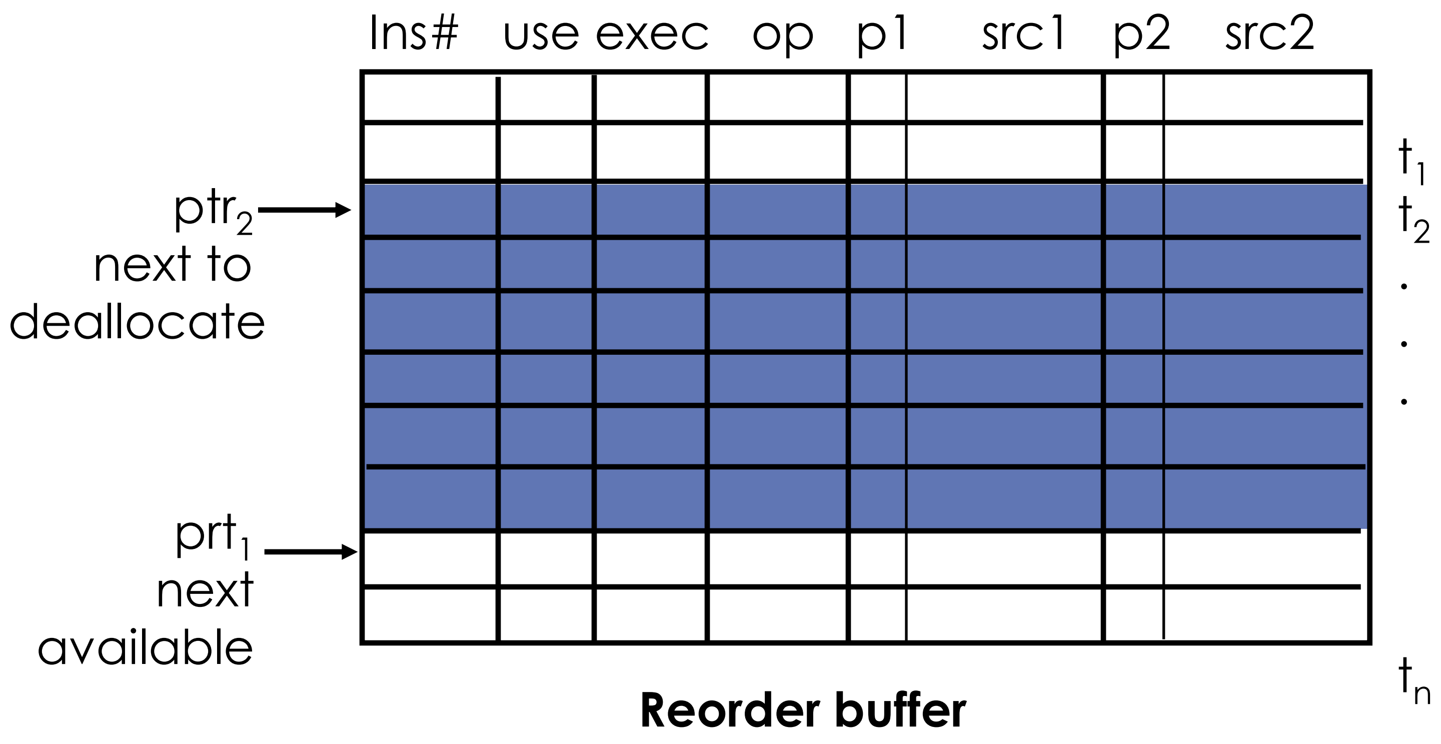
Any anti-dependence can be eliminated by renaming (requires additional storage). Renaming can be done in hardware!

Register Renaming



- Decode stage renames registers and adds instructions to the reorder buffer (ROB).
- Renaming eliminates WAR or WAW hazards
- Instructions in ROB whose RAW hazards are resolved can issue
- Out-of-order or dataflow execution

Dataflow Execution



Instruction slot is candidate for execution when…

- Instruction is valid ("use" bit is set)
- Instruction is not already executing ("exec" bit is clear)
- Operands are available ("p1" and "p2" are set for "src1" and "src2")

Renaming and the ROB

1. Insert instruction into ROB (after decoding it)
   i. ROB entry is used, set “use=1”
   ii. Instruction is not yet executing, set “exec=0”
   iii. Specify operation in ROB entry
2. Update renaming table
   i. Identify instruction’s destination register (e.g., F1)
   ii. Look up register (e.g., F1) in renaming table
   iii. Insert pointer from renaming table to instruction’s ROB entry
3. When instruction executes, update “exec=1”
4. When instruction writes-back, replace pointer to ROB with value produced by instruction

Example

Renaming table

	p	data
F1		
F2		t1 v1
F3		
F4		t2 t5
F5		
F6		t3
F7		
F8		t4 v4

data / t_i

Reorder buffer

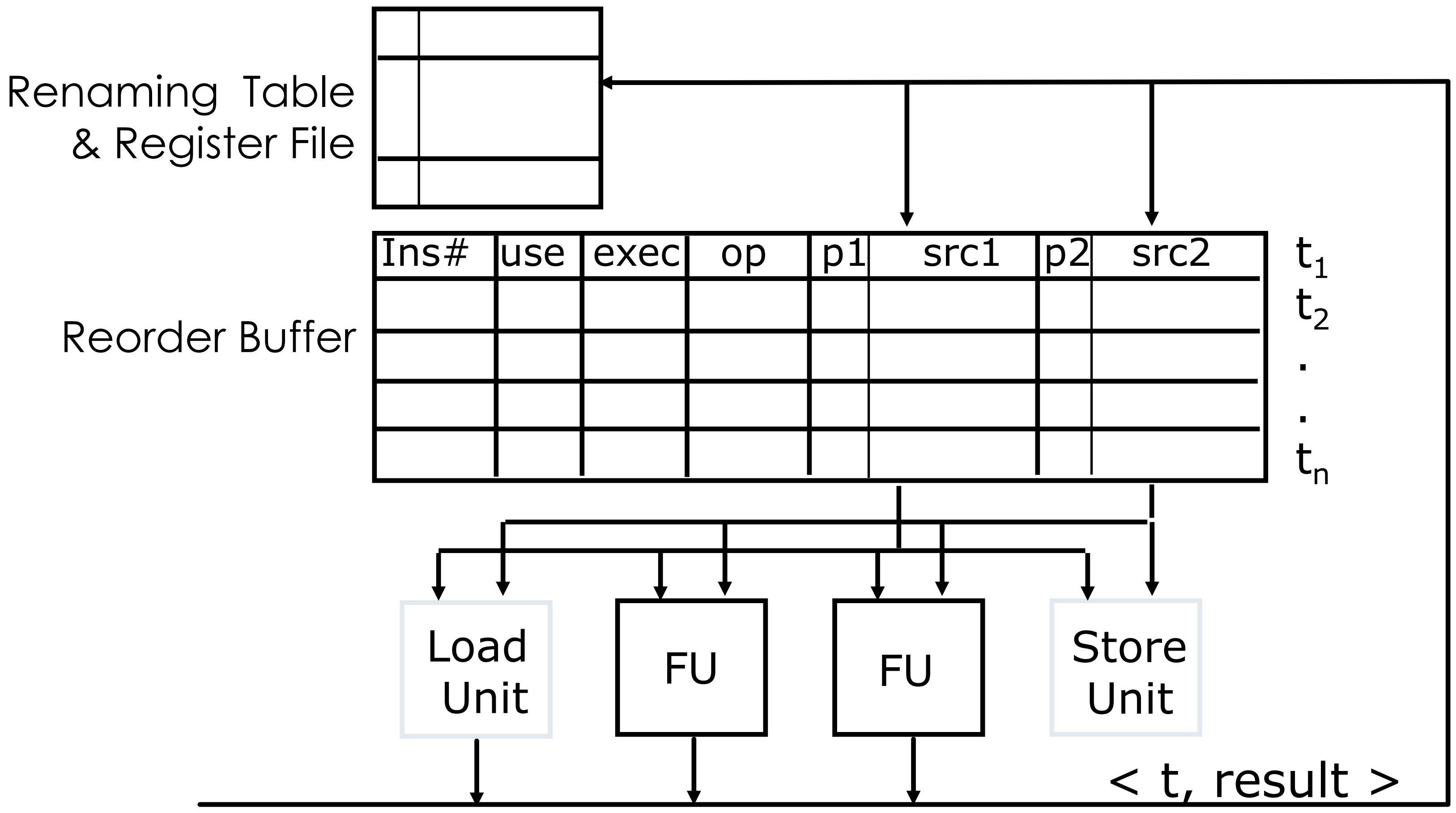
Ins#	use	exec	op	p1	src1	p2	src2	
1	1 0	1 0	LD					t_1
2	1 0	1 0	LD					t_2
3	1	0	MUL	0	t2 v2	1	v1	t_3
4	1 0	1 0	SUB	1	v1	1	v1	t_4
5	1	0	DIV	1	v1	0 1	t4 v4	t_5
								.
								.

1: LD F2, 34 (R2)
2: LD F4, 45 (R3)
3: MUTLD F6, F4, F2
4: SUBD F8, F2, F2
5: DIVD F4, F2, F8
6: ADDD F10, F6, F4

When are names in sources replaced by data? When a functional unit produces data

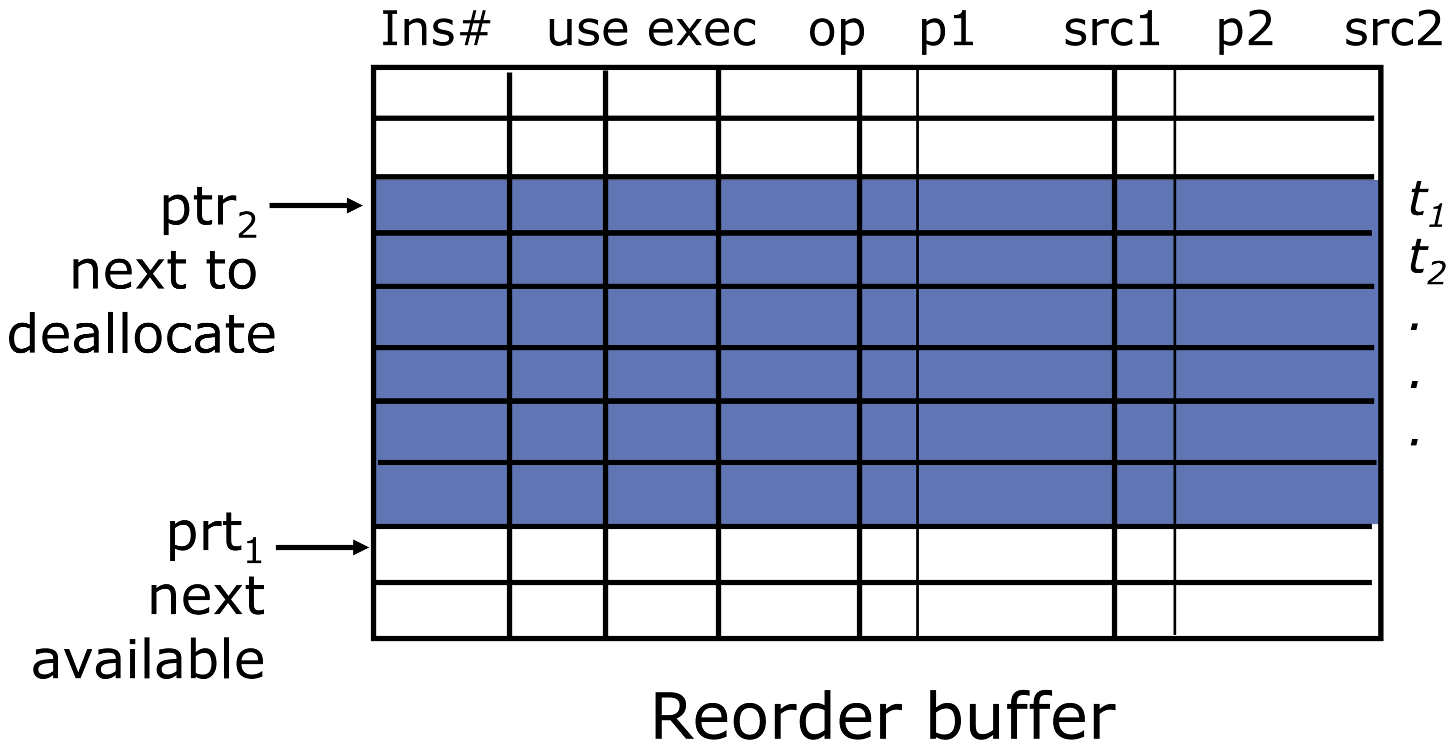
When can a name be re-used? When an instruction completes

Register Renaming



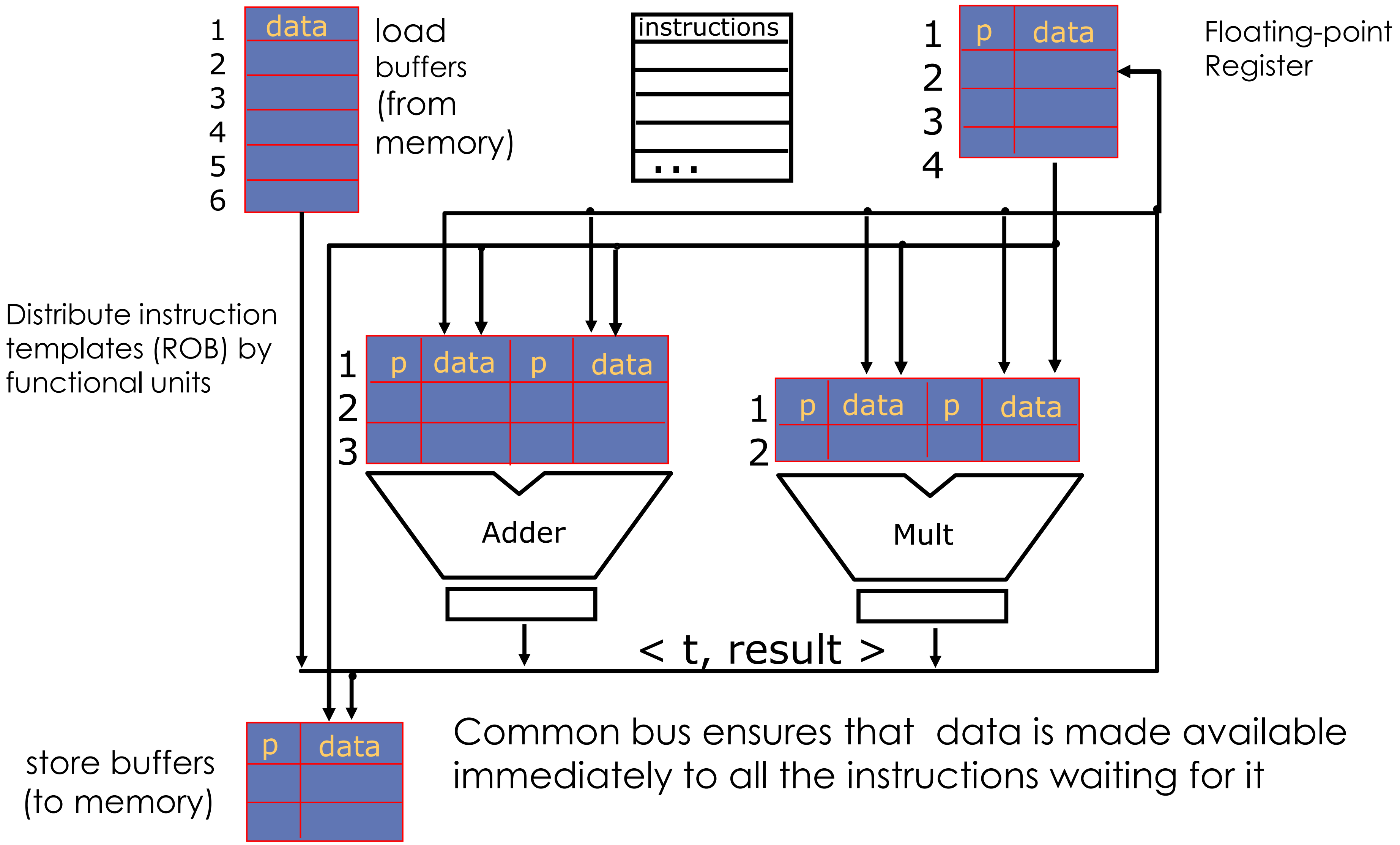
- Decode stage allocates instruction template (i.e., tag t) and stores tag in register file.
- When instruction completes, tag is de-allocated.

Allocating/Deallocating Templates



- Reorder buffer is managed circularly.
- Field "exec" is set when instruction begins execution.
- Field "use" is cleared when instruction completes
- Ptr2 increments when "use" bit is cleared.

IBM 360/91 Floating-Point Unit


1
2
3
4
5
6
data
load buffers (from memory)
instructions
...
1
2
3
4
p
data
Floating-point Register
Distribute instruction templates (ROB) by functional units
1
2
3
p
data
p
data
1
2
p
data
p
data
Adder
Mult
< t, result >
store buffers (to memory)
p
data

Common bus ensures that data is made available immediately to all the instructions waiting for it

Effectiveness

History

- Renaming/out-of-order execution first introduction in 360/91 in 1969
- However, implementation did not re-appear until mid-90s
- Why?

Limitations

- Effective on a very small class of problems
- Memory latency was a much bigger problem in the 1960s
- Problem-1: Exceptions were not precise
- Problem-2: Control transfers

Precise Interrupts

Definition

- It must appear as if an interrupt is taken between two instructions
- Consider instructions k, k+1
- Effect of all instructions up to and including k is totally complete
- No effect of any instruction after k has taken place

Interrupt Handler

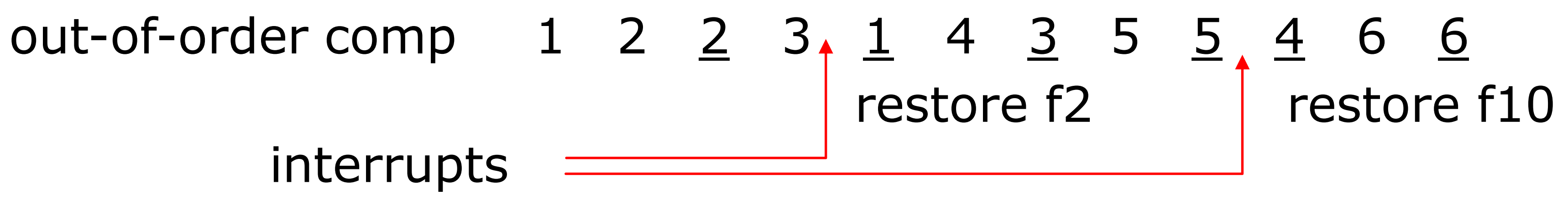
- Aborts program or restarts at instruction k+1

Out-of-Order & Interrupts

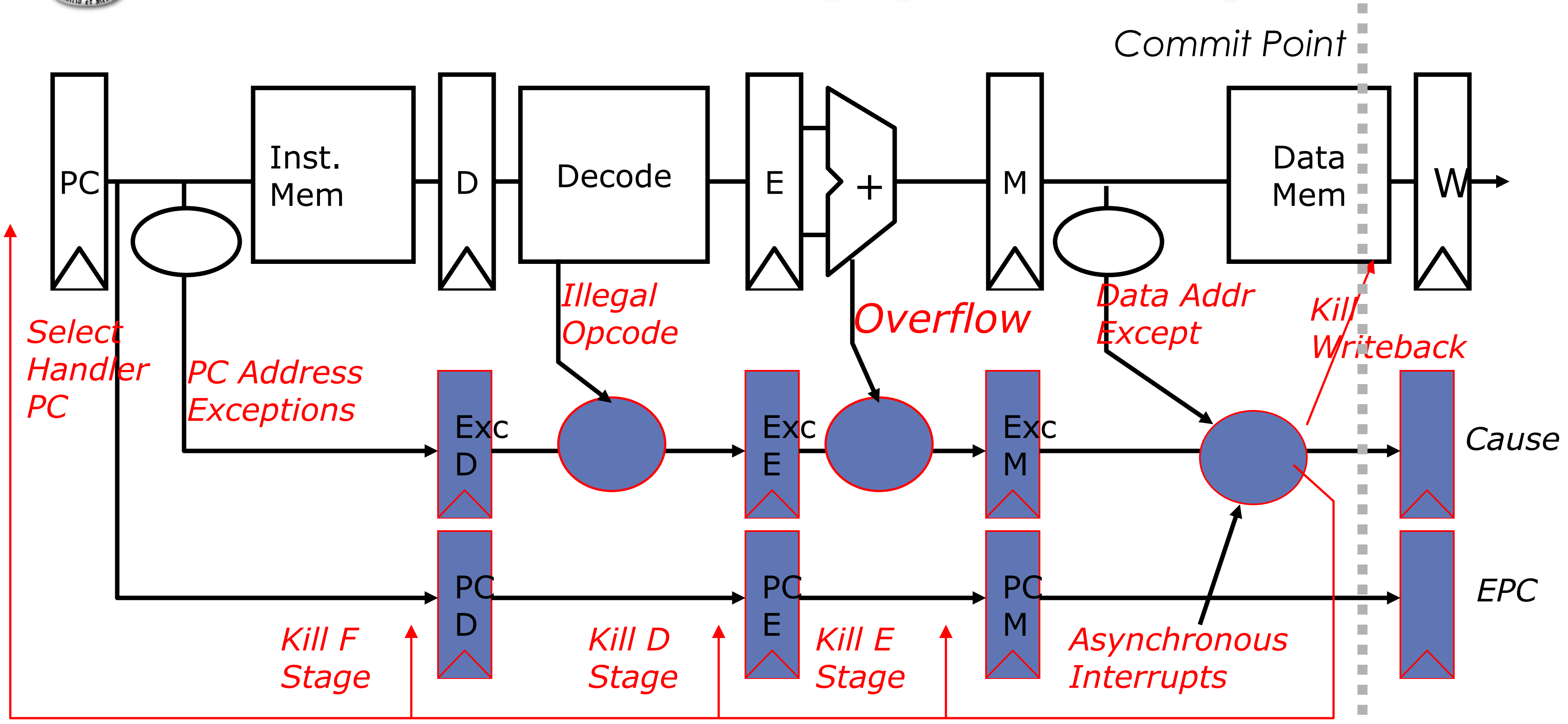
Out-of-order Completion

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

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

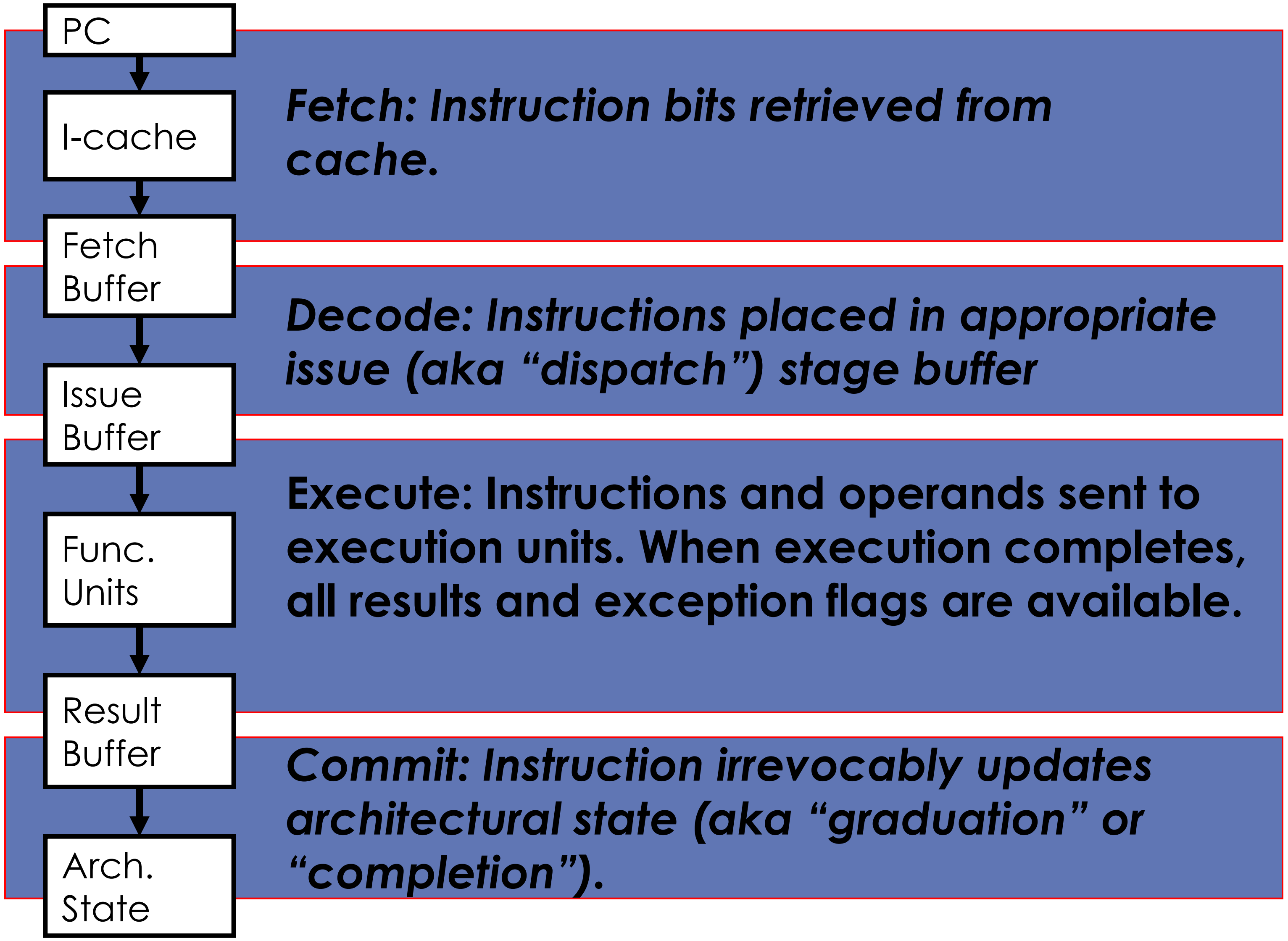


Interrupt Handling (in-order)



-- Hold exception flags in pipeline until commit point
-- Exceptions in earlier pipe stages override later exceptions
-- Inject external interrupts, which over-ride others, at commit point
-- If exception at commit: (1) update Cause and EPC registers, (2) kill all stages, (3) inject handler PC into fetch stage

Phases of Instruction Execution



Fetch: Instruction bits retrieved from cache.

Decode: Instructions placed in appropriate issue (aka "dispatch") stage buffer

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

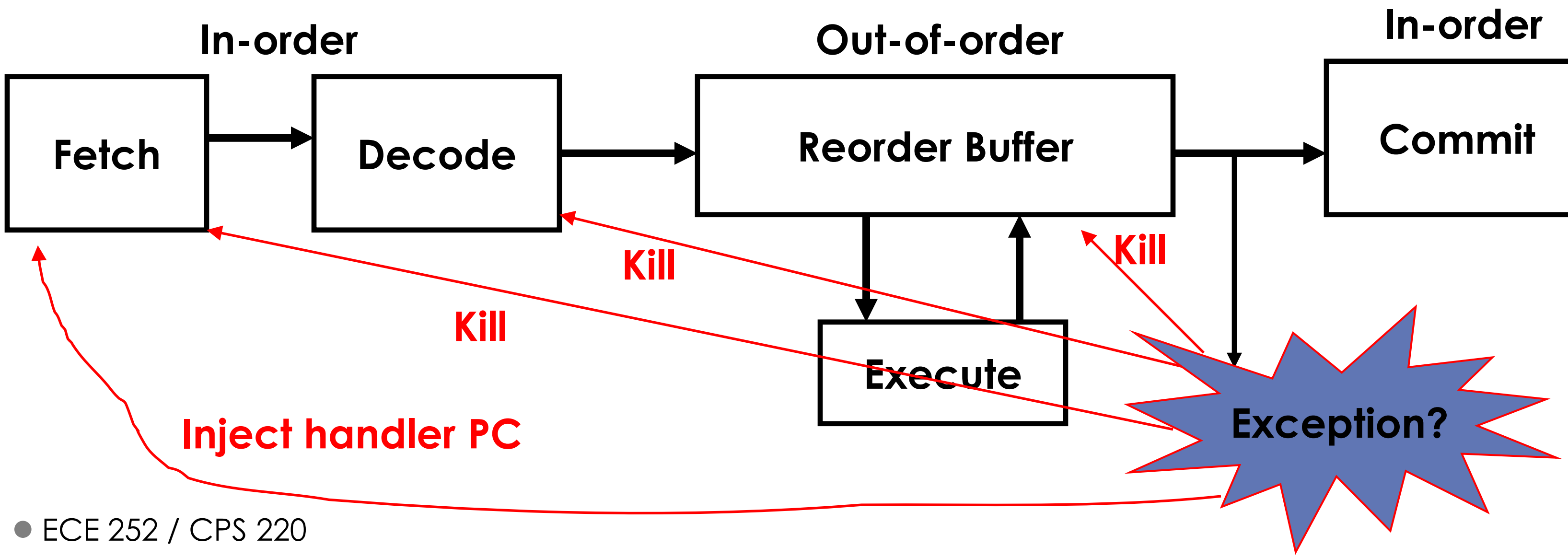
Commit: Instruction irrevocably updates architectural state (aka "graduation" or "completion").

Supporting Precise Exceptions

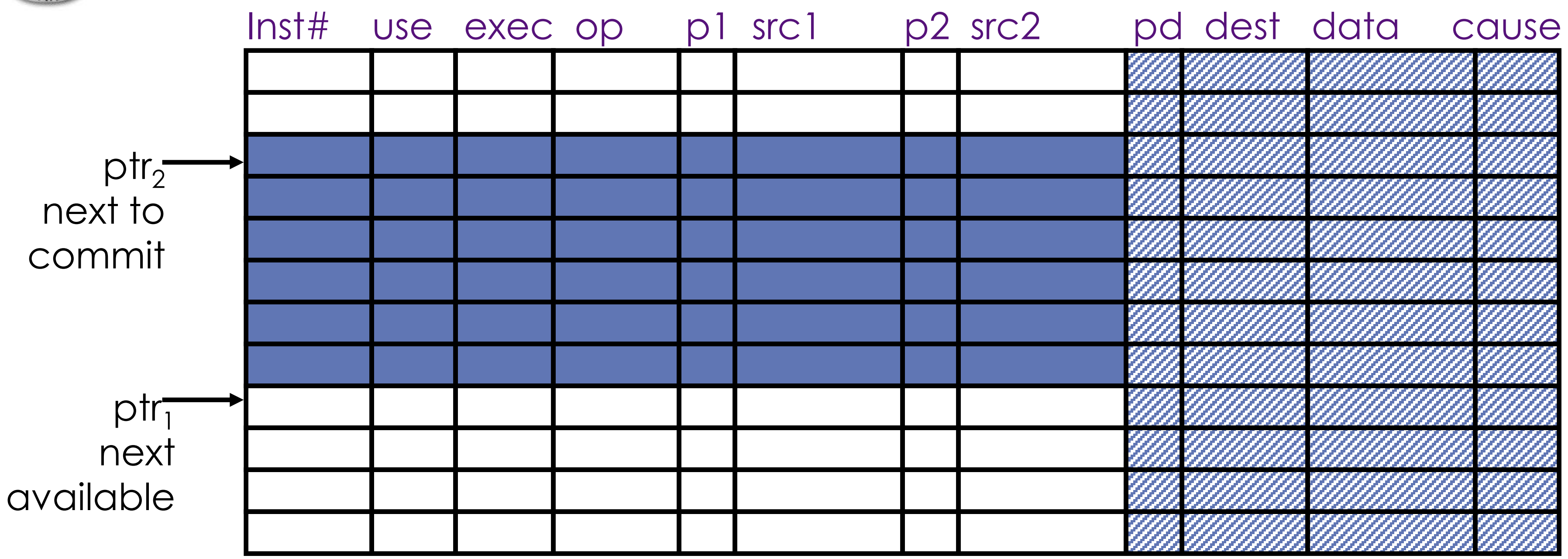
In-Order Commit

- Instructions fetched, decoded into re-order buffer (ROB) in-order
- Instructions executed, completed out-of-order
- Instructions committed in-order
- Instruction commit writes to architectural state (e.g., register file, memory)

Temporary storage needed to hold results before commit (e.g., shadow registers, store buffers)

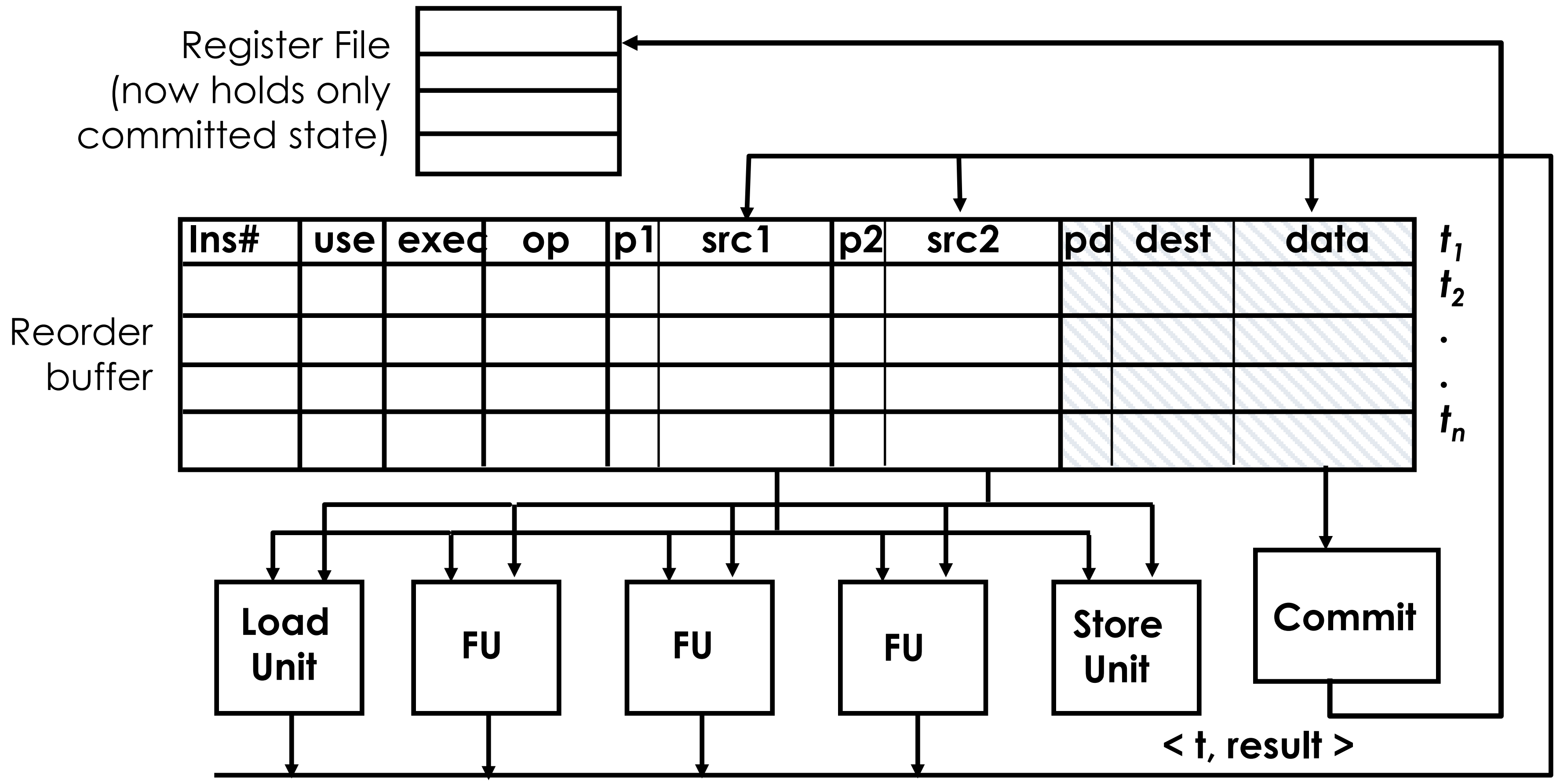


Supporting Precise Exceptions



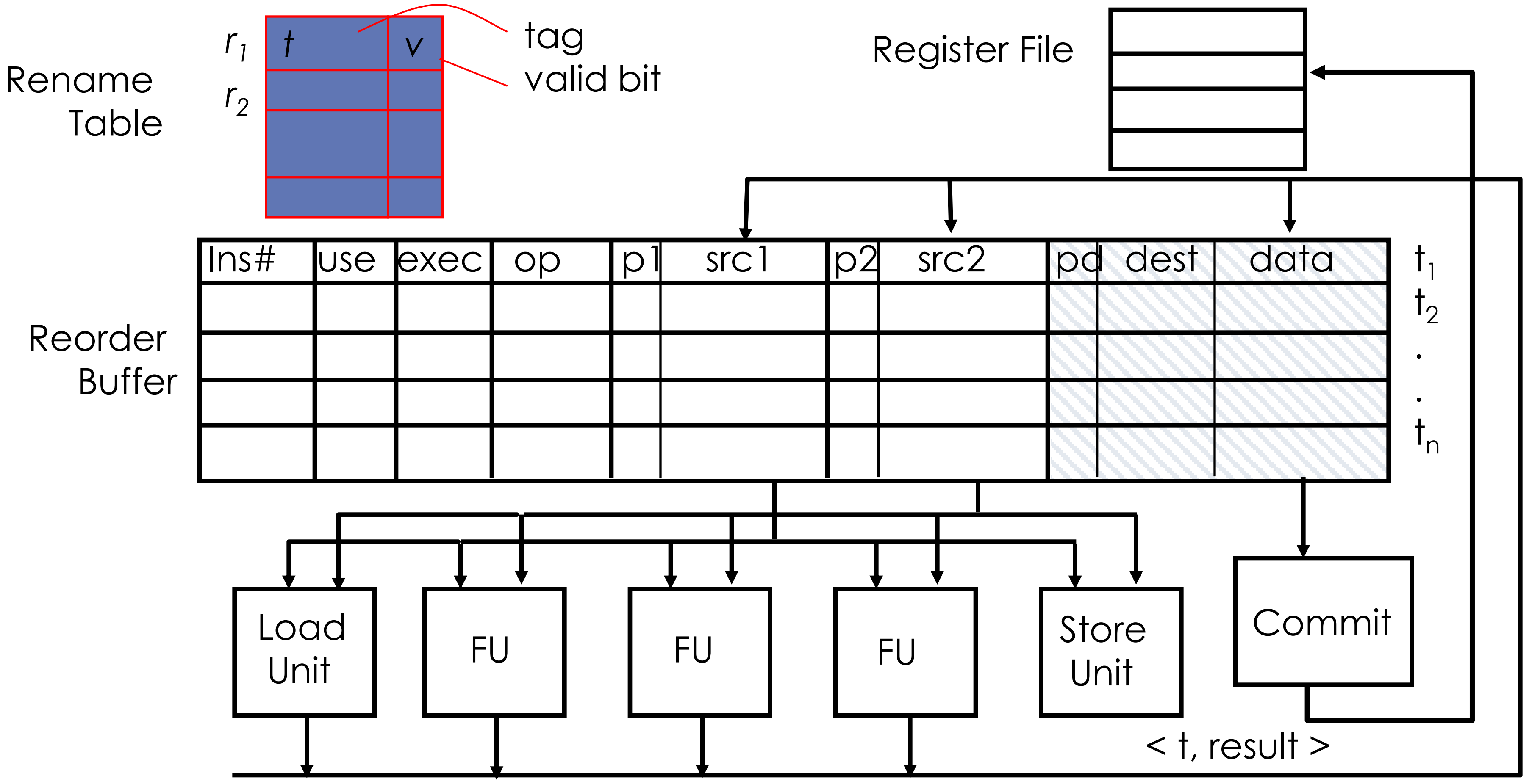
- Add <pd, dest, data, cause> fields to instruction template
- Commit instructions to register file and memory in-order
- On exception, clear re-order buffer (reset ptr-1 = ptr-2)
- Store instructions must commit before modifying memory

Renaming and Rollbacks



Register file no longer contains renaming tags. How does decode stage find tag of a source register? Search the 'dest' field in reorder buffer (ROB).

Renaming Table



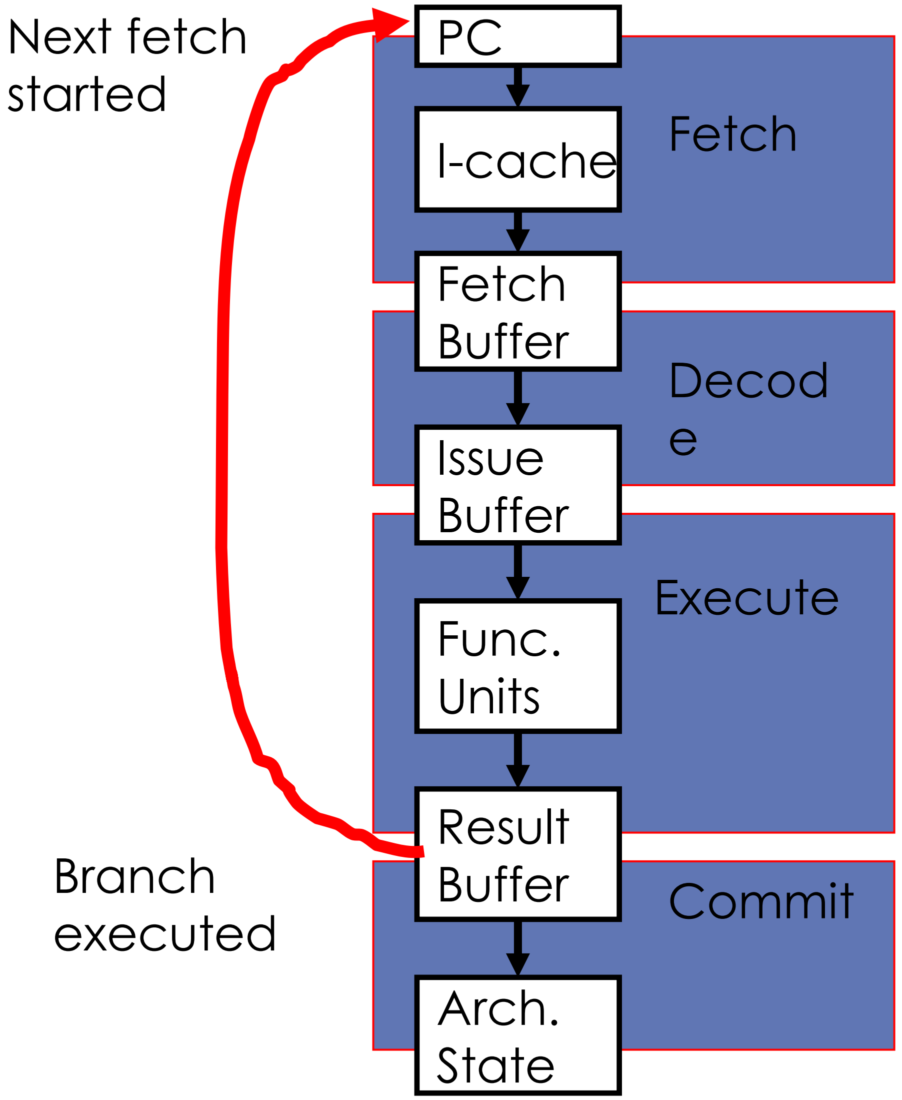
Renaming table is a cache, speeds up register name look-up. Table is cleared after each exception. When else are valid bits cleared? Control transfers.

Control Transfer Penalty

Modern processors may have >10 pipeline stages between next PC calculation and branch resolution.

How much work is lost if pipeline does not follow correct instruction flow?

[Loop Length] x [Pipeline Width]



Branches and Jumps

Each instruction fetch depends on 1-2 pieces of information from preceding instruction:

1. Is preceding instruction a branch?
2. If so, what is the target address?

Instruction	Taken known?	Target known?
J	after decode	after decode
JR	after decode	after fetch
BEQZ/BNEZ	after fetch*	after decode

*assuming zero? detect when register read

Reducing Control Flow Penalty

Software Solutions

1. Eliminate branches -- loop unrolling increases run length before branch
2. Reduce resolution time – instruction scheduling moves instruction that produces condition earlier (of limited value)

Hardware Solutions

1. Find other work – delay slots and software cooperation
2. Speculate – predict branch result and execute instructions beyond branch

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:

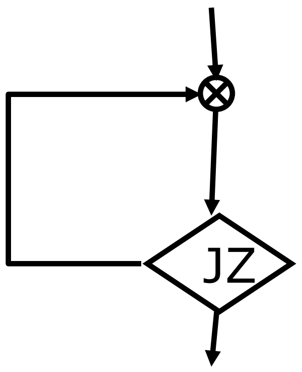
-- Separate instruction execution and instruction commit

-- Kill instructions following branch in pipeline

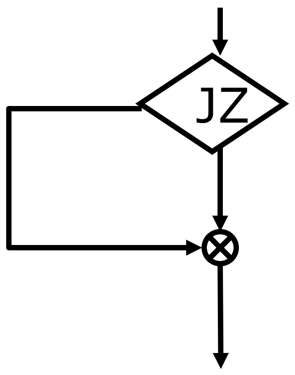
-- Restore architectural state to correct path of execution

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.

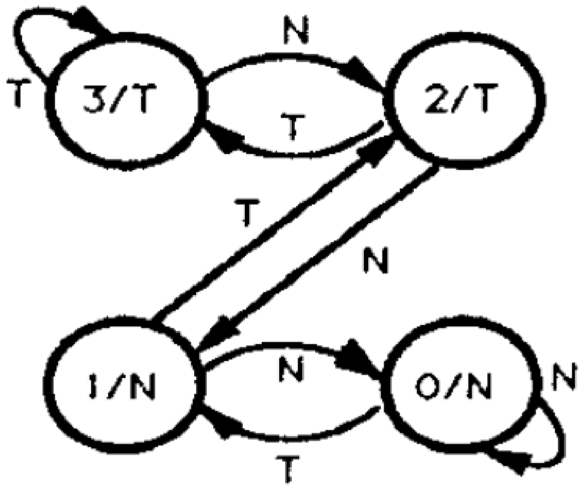
Dynamic Branch Prediction

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

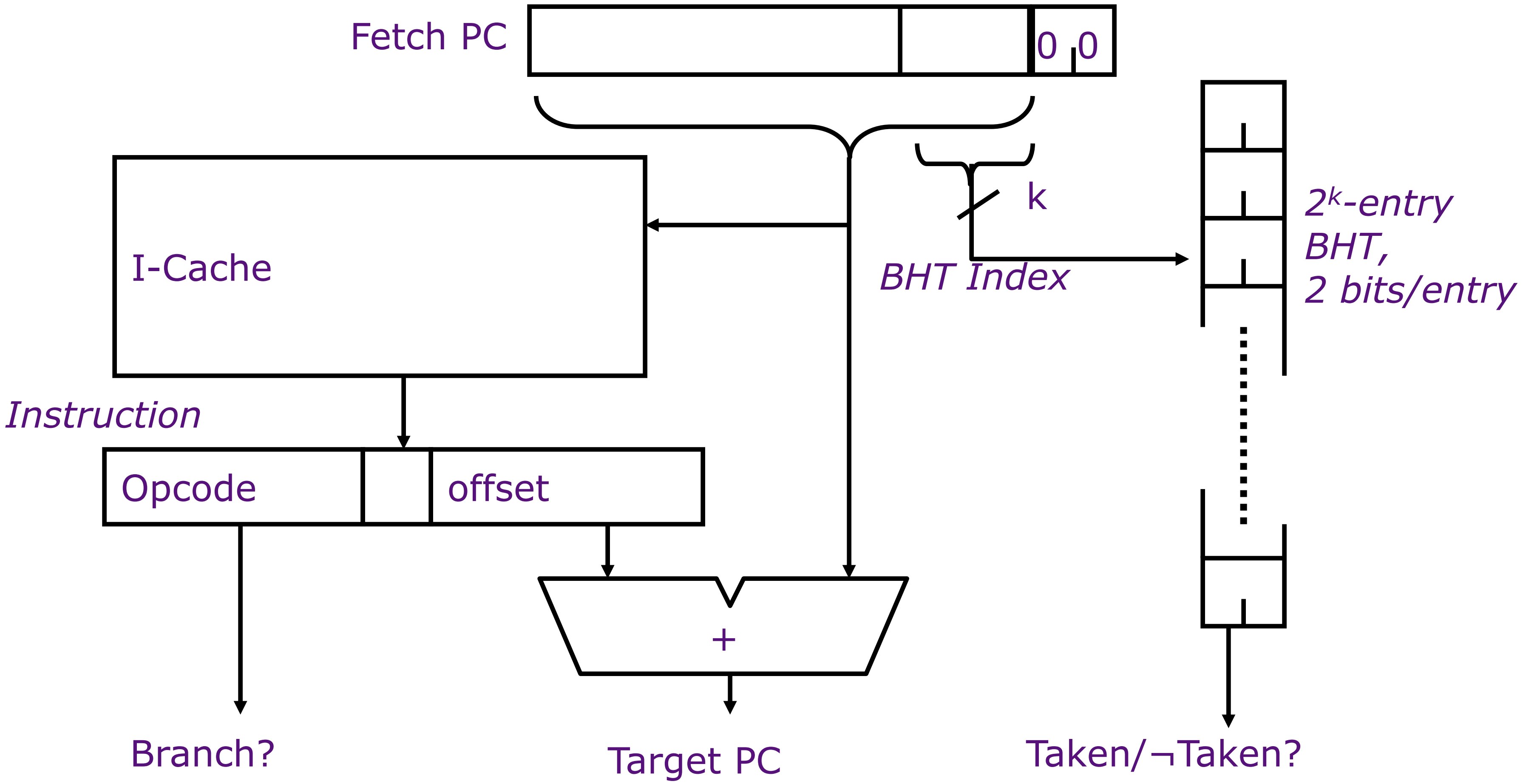


Automaton A2
(2-bit Saturating Up-down Counter)

Use two-bit saturating counter.

Changes prediction after two consecutive mistakes.

Branch History Table (BHT)

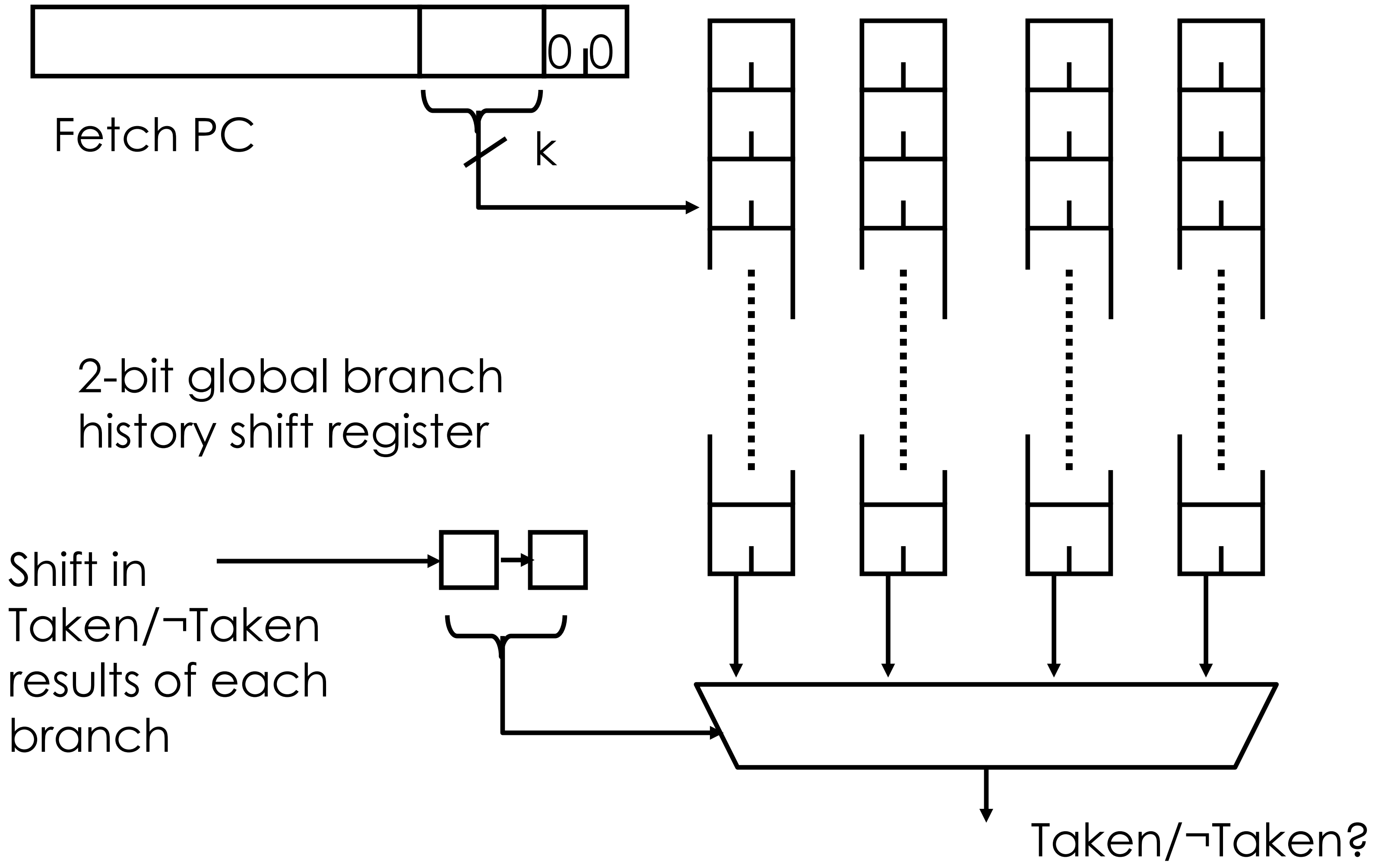


BHT is an array of 2-bit branch predictors, indexed by branch PC

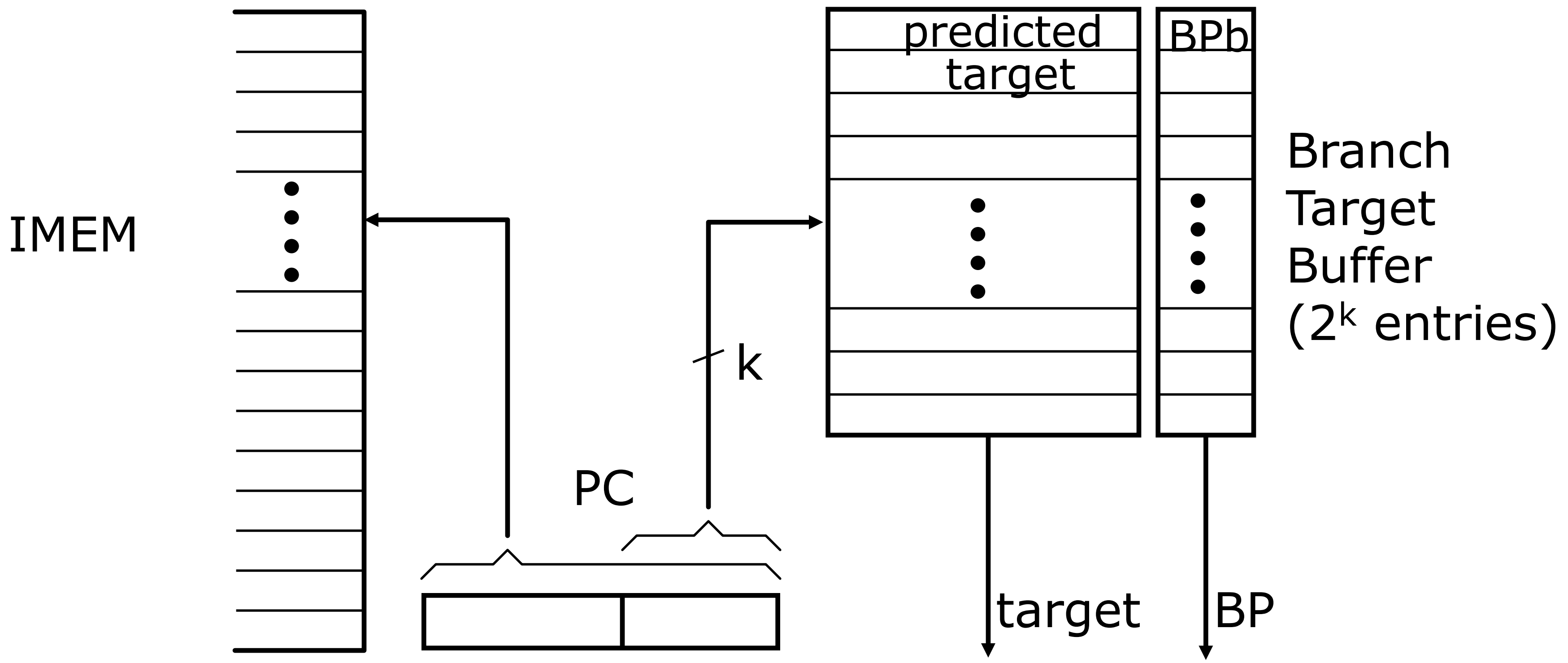
4K-entry branch history table, 80-90% accurate

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)



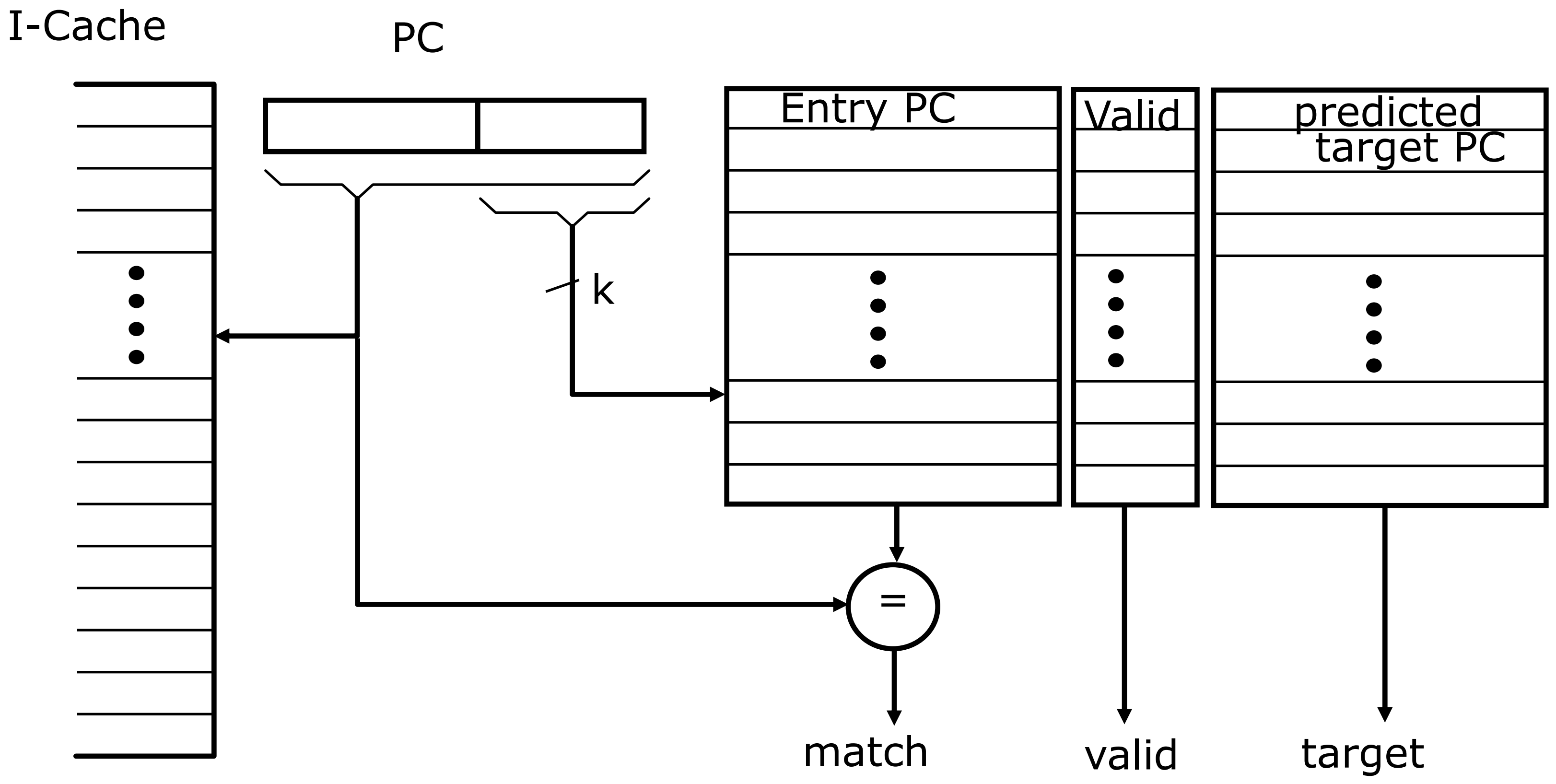
BHT only predicts branch direction (taken, not taken). Cannot redirect instruction flow until after branch target determined.

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

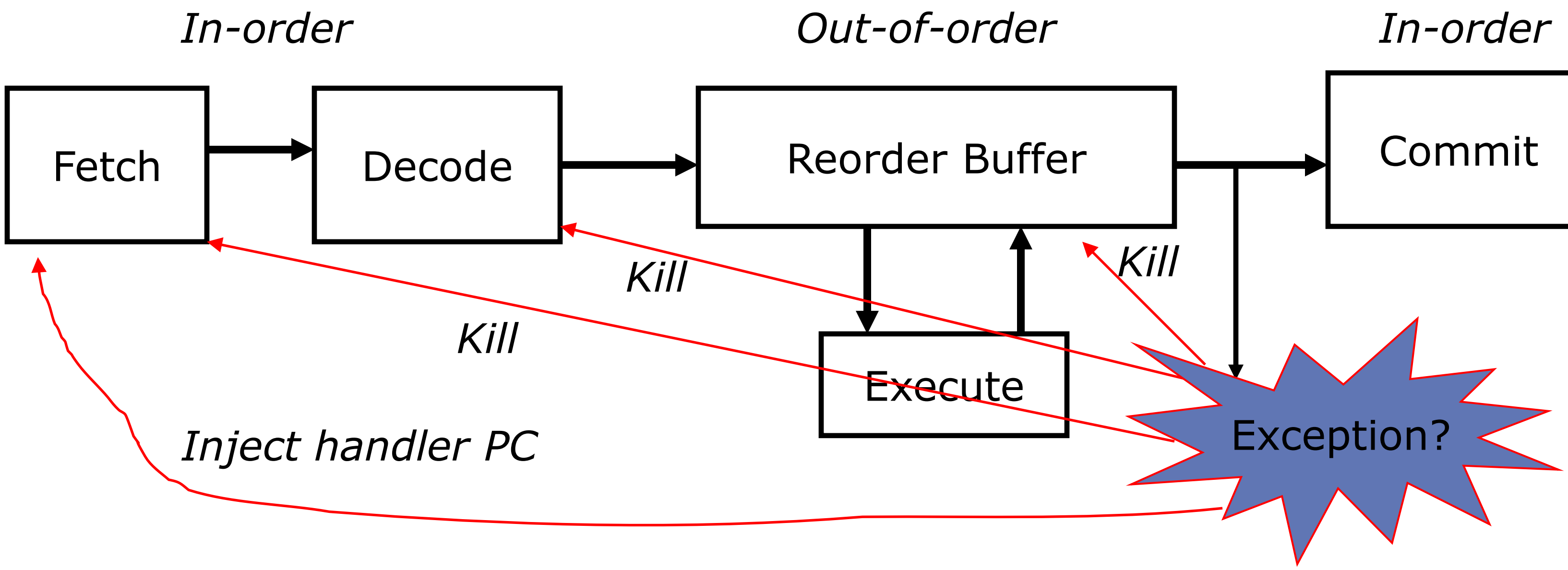
In-order execution

- 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



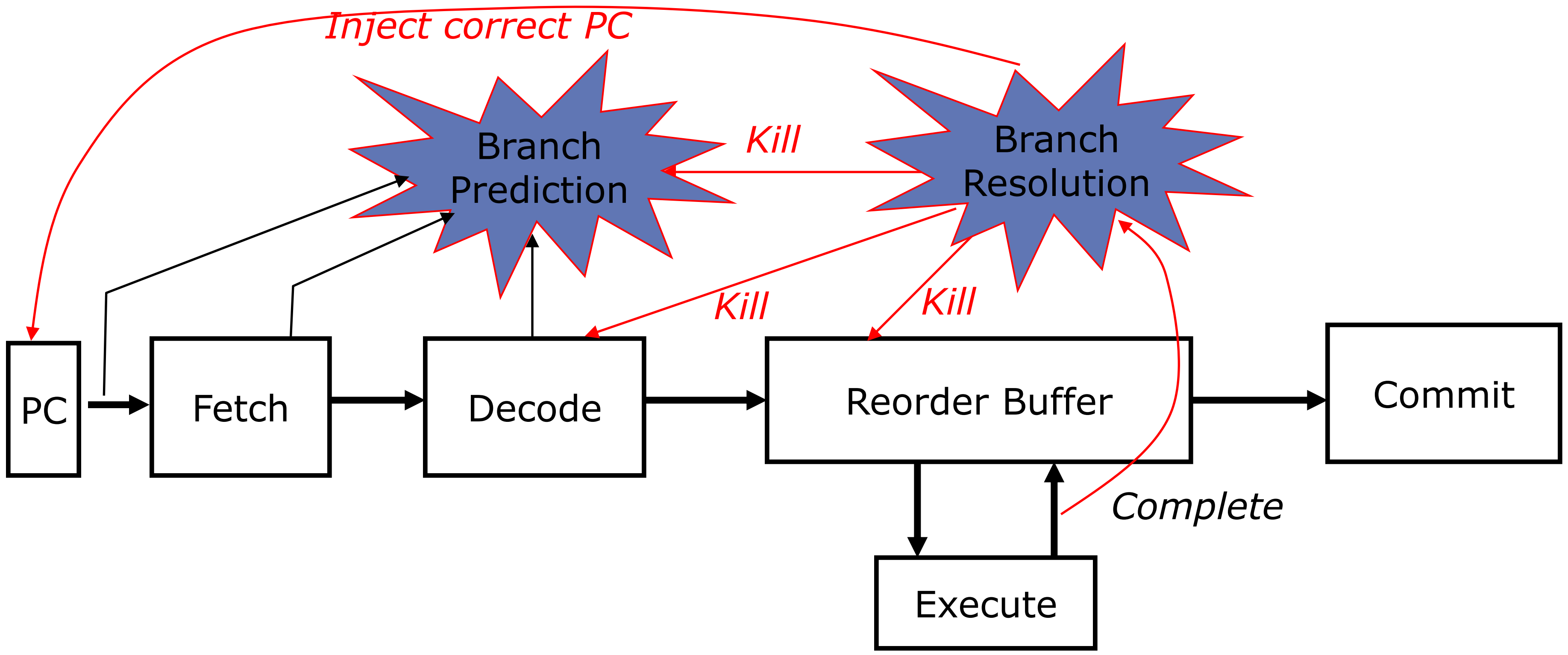
-- Instructions fetched, decoded in-order (entering the reorder buffer -- ROB)

-- Instructions executed out-of-order

-- Instructions commit in-order (write back to architectural state)

-- Temporary storage needed in ROB to hold results before commit

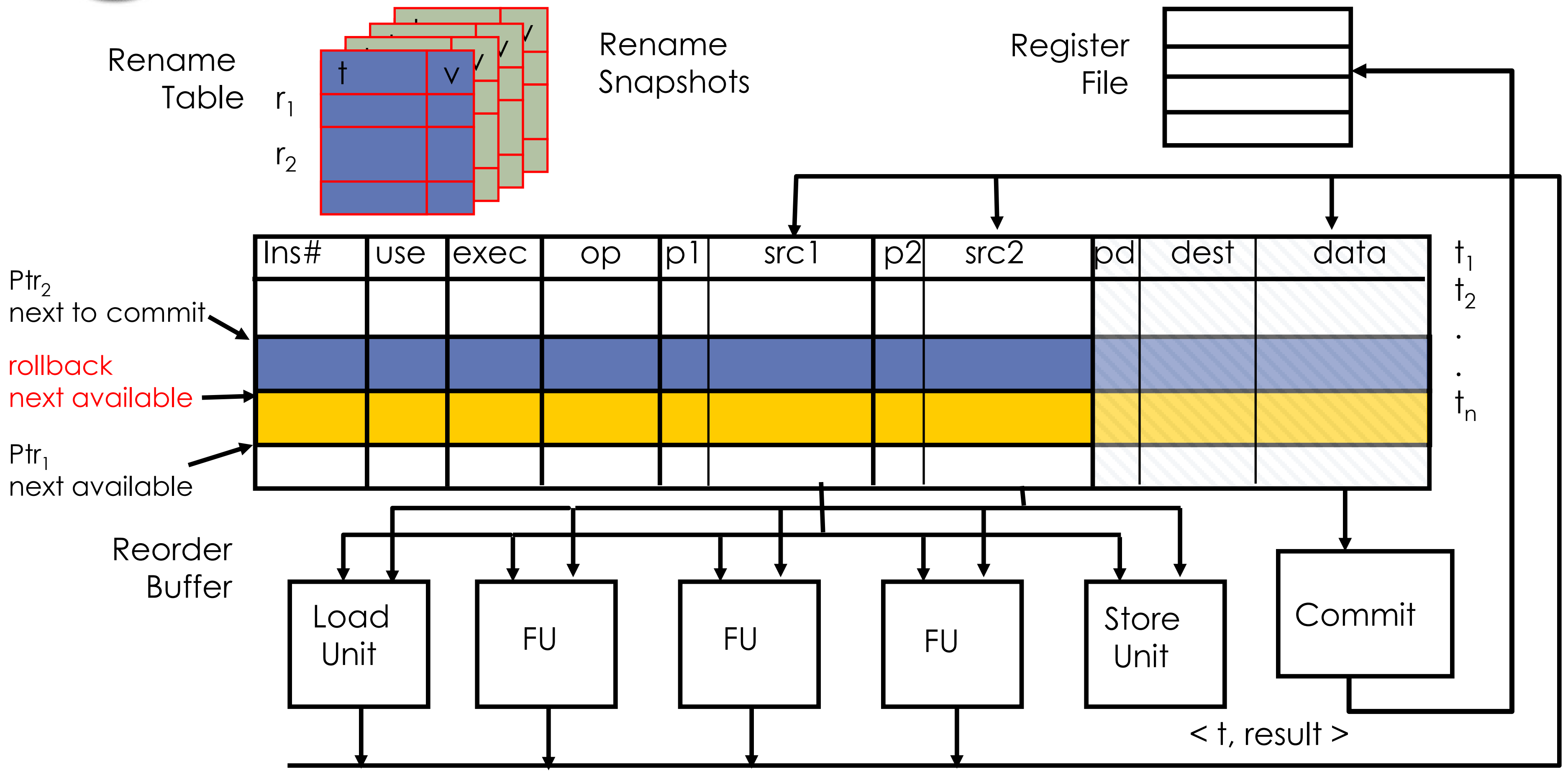
Branch Misprediction in Pipeline



-- 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)