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

Great Idea #1: Abstraction (Levels of Representation/Interpretation)

```
lw    t0, t2, 0
lw    t1, t2, 4
sw    t1, t2, 0
sw    t0, t2, 4
```

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., RISC-V)

Anything can be represented as a *number*, i.e., data or instructions

Assembler

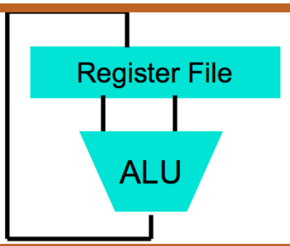
Machine Language Program (RISC-V)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

We are here!

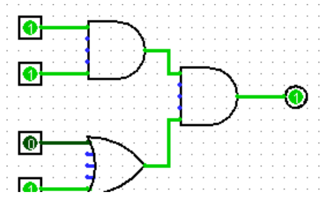
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Instructions interact with each other in pipeline

- An instruction in the pipeline may need a resource being used by another instruction in the pipeline → *structural hazard*

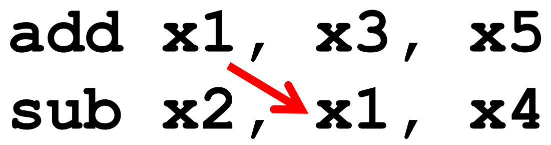
- An instruction may depend on something produced by an earlier instruction
  - Dependence may be for a data value
    → *data hazard*
  - Dependence may be for the next instruction's address
    → *control hazard (branches, exceptions)*

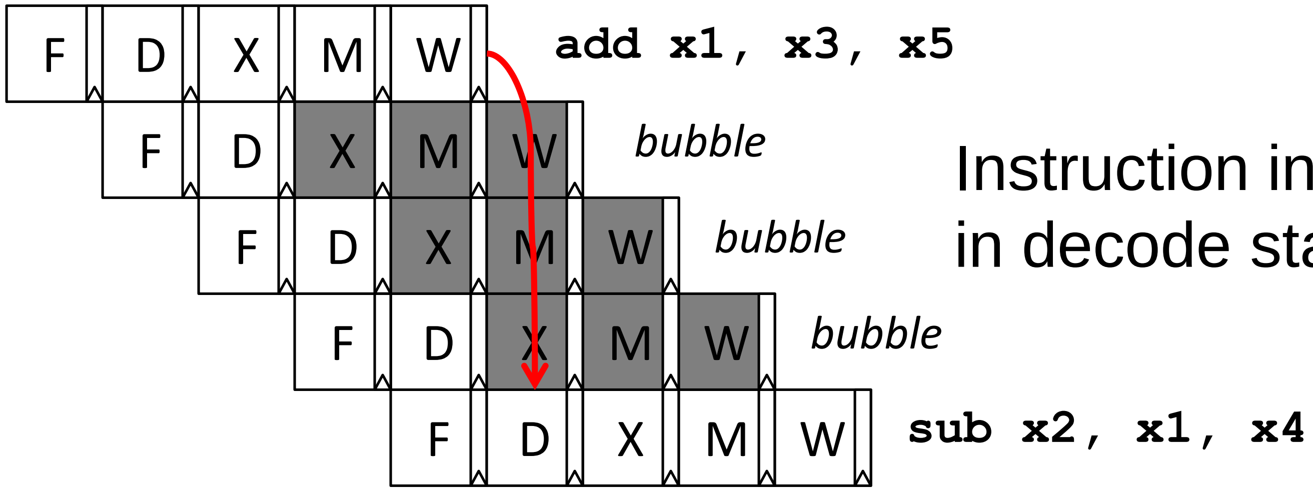
- Handling hazards generally introduces bubbles into pipeline and reduces ideal CPI > 1

Review: Three Strategies for Data Hazards

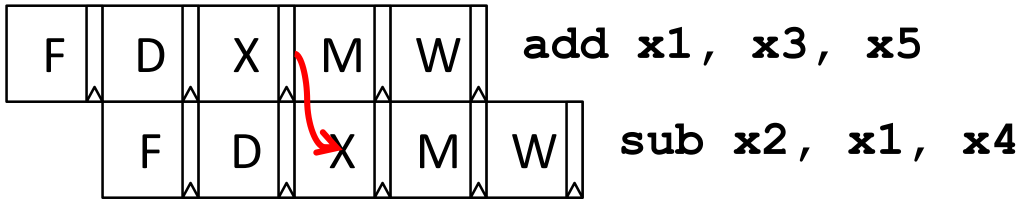
- ***Interlock***
  - Wait for hazard to clear by holding dependent instruction in issue stage
- ***Bypass***
  - Resolve hazard earlier by bypassing value as soon as available
- ***Speculate***
  - Guess on value, correct if wrong

Interlocking Versus Bypassing



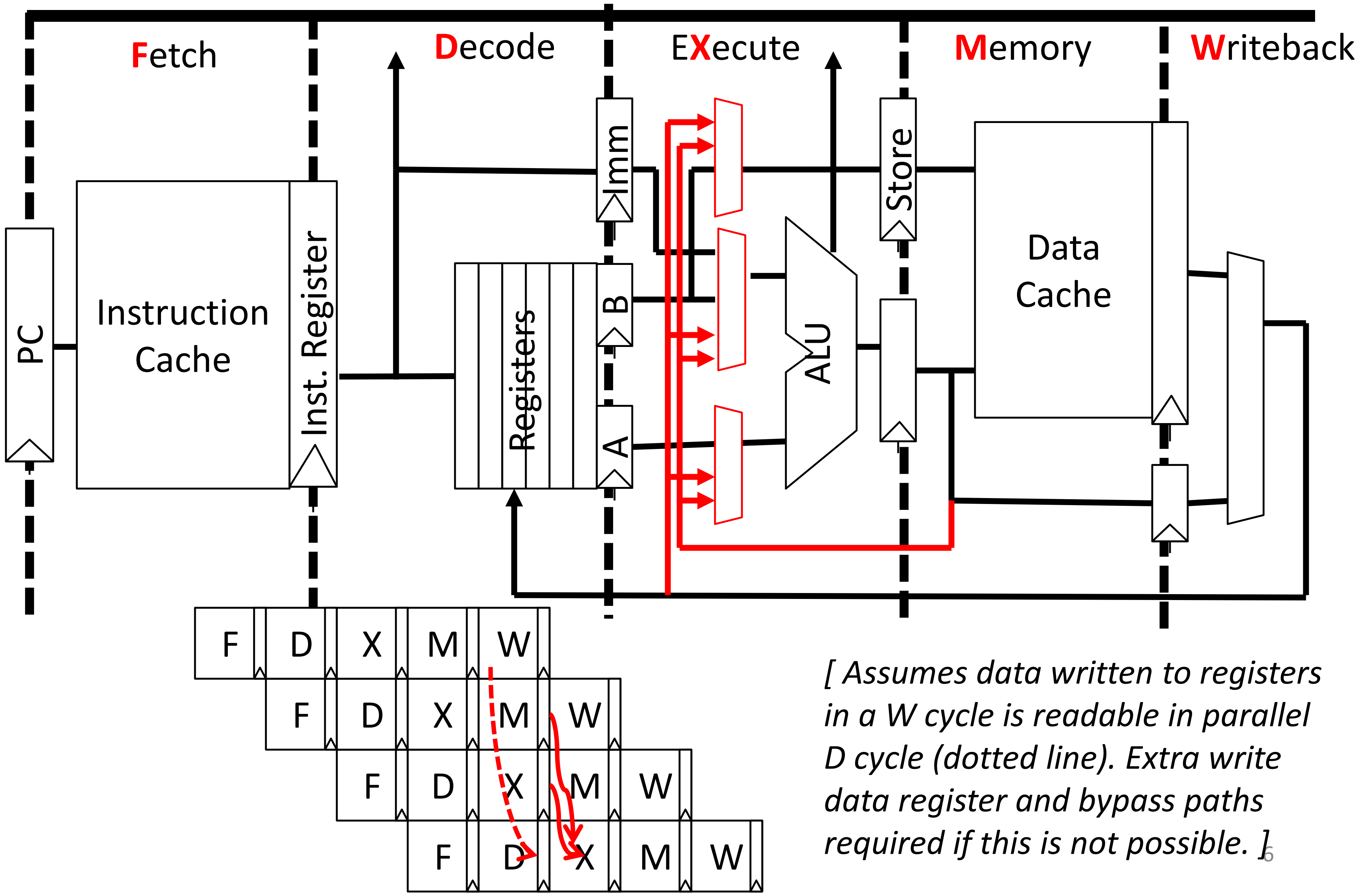


Instruction interlocked in decode stage



Bypass around ALU with no bubbles

Fully Bypassed Data Path



[Assumes data written to registers in a W cycle is readable in parallel D cycle (dotted line). Extra write data register and bypass paths required if this is not possible.]

Register vs. Memory Dependence

- Data hazards due to register operands can be determined at the decode stage, but data hazards due to memory operands can be determined only after computing the effective address

- Store: **`M[r1 + disp1] ← r2`**
- Load: **`r3 ← M[r4 + disp2]`**

- Does **`(r1 + disp1) = (r4 + disp2) ?`**

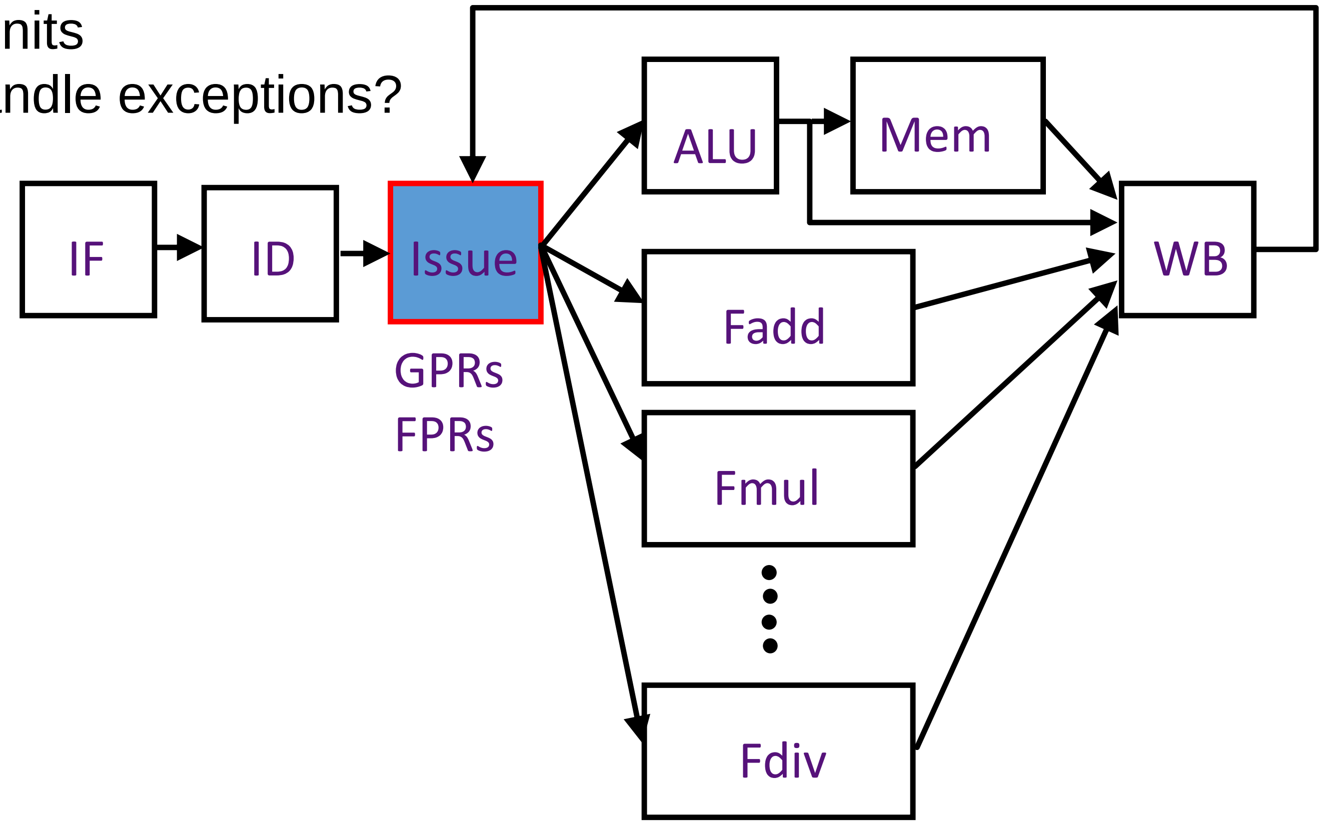
Complex Pipelining: Motivation

Pipelining becomes complex when we want high performance in the presence of:

- Long latency or partially pipelined floating-point units
- Memory systems with variable access time
- Multiple arithmetic and memory units

Issues in Complex Pipeline Control

- Structural conflicts at the execution stage if some FPU or memory unit is not pipelined and takes more than one cycle
- Structural conflicts at the write-back stage due to variable latencies of different functional units
- Out-of-order write hazards due to variable latencies of different functional units
- How to handle exceptions?



Types of Data Hazards

Consider executing a sequence of

$$r_k \leftarrow r_i \text{ op } r_j$$

type of instructions

Data-dependence

$$\begin{tikzcd}
r_3 \arrow[drr] & \leftarrow & r_1 & \text{op} & r_2 \\
r_5 & \leftarrow & r_3 & \text{op} & r_4
\end{tikzcd}$$

Read-after-Write (RAW) hazard

Anti-dependence

$$\begin{tikzcd}
r_3 & \leftarrow & r_1 \arrow[dll] & \text{op} & r_2 \\
r_1 & \leftarrow & r_4 & \text{op} & r_5
\end{tikzcd}$$

Write-after-Read (WAR) hazard

Output-dependence

$$\begin{tikzcd}
r_3 \arrow[d, bend right] & \leftarrow & r_1 & \text{op} & r_2 \\
r_3 & \leftarrow & r_6 & \text{op} & r_7
\end{tikzcd}$$

Write-after-Write (WAW) hazard

Data Hazard Examples

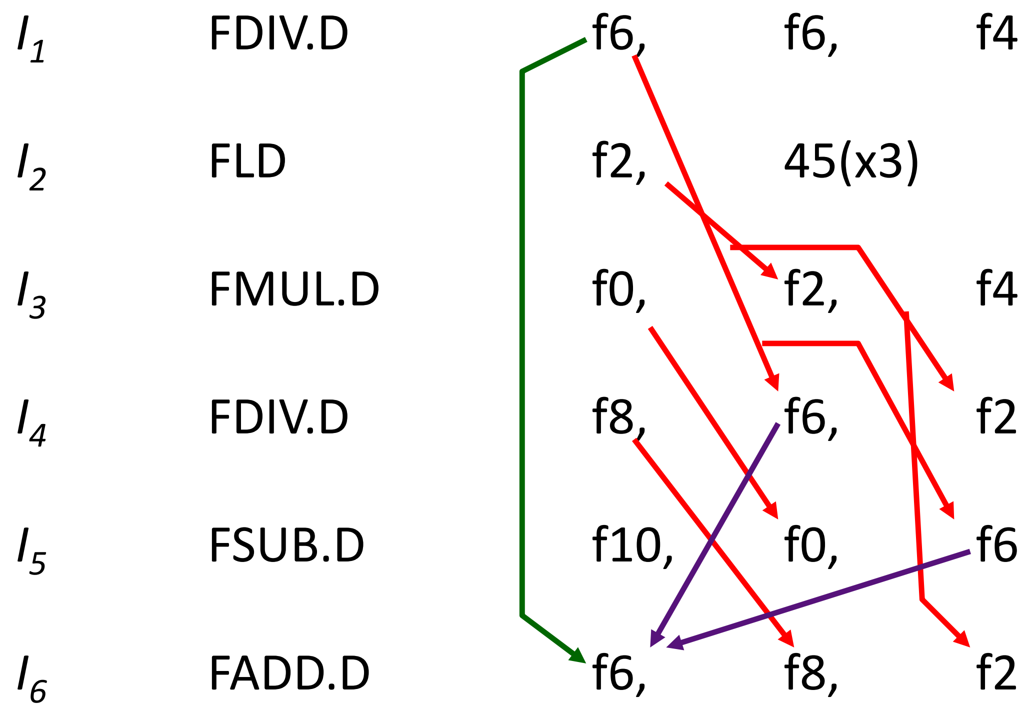
WAW (write after write) *-* ***j*** *tries to write an operand before it is written by* ***i****. The writes end up being performed in the wrong order, leaving the value written by* ***i*** *rather than the value written by* ***j*** *in the destination.*

LW R1, 0(R2)	IF	ID	EX	MEM1	MEM2	**WB**
ADD R1, R2, R3		IF	ID	EX	**WB**	

WAR (write after read) *-* ***j*** *tries to write a destination before it is read by* ***i*** *, so* ***i*** *incorrectly gets the new value.*

SW R1, 0(R2)	IF	ID	EX	MEM1	**MEM2**	WB
ADD R2, R3, R4		IF	ID	EX	**WB**	

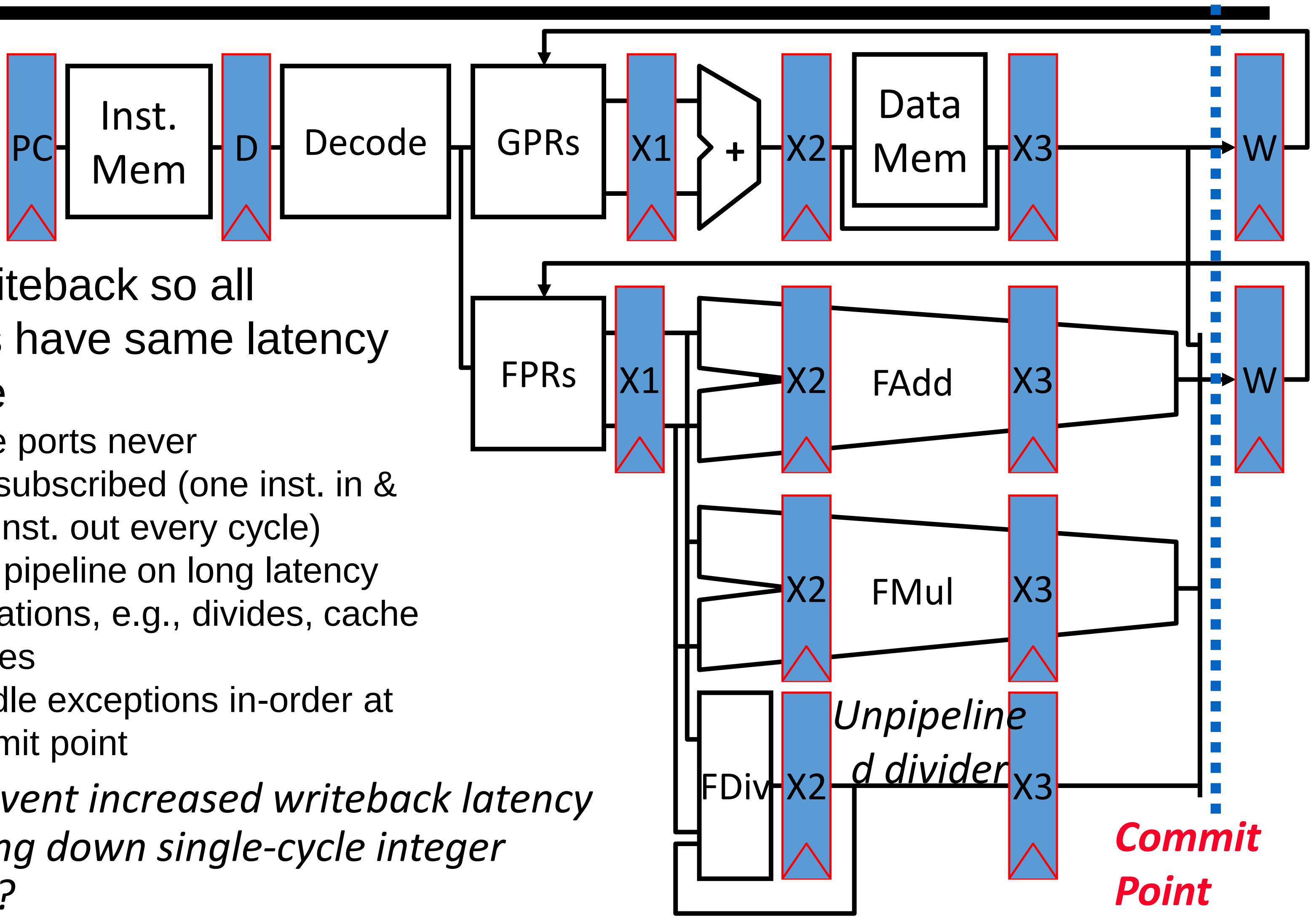
Data Hazards: An Example



I_1	FDIV.D	f6,	f6,	f4
I_2	FLD	f2,	45(x3)	
I_3	FMUL.D	f0,	f2,	f4
I_4	FDIV.D	f8,	f6,	f2
I_5	FSUB.D	f10,	f0,	f6
I_6	FADD.D	f6,	f8,	f2

RAW Hazards
WAR Hazards
WAW Hazards

Recap: Complex In-Order Pipeline

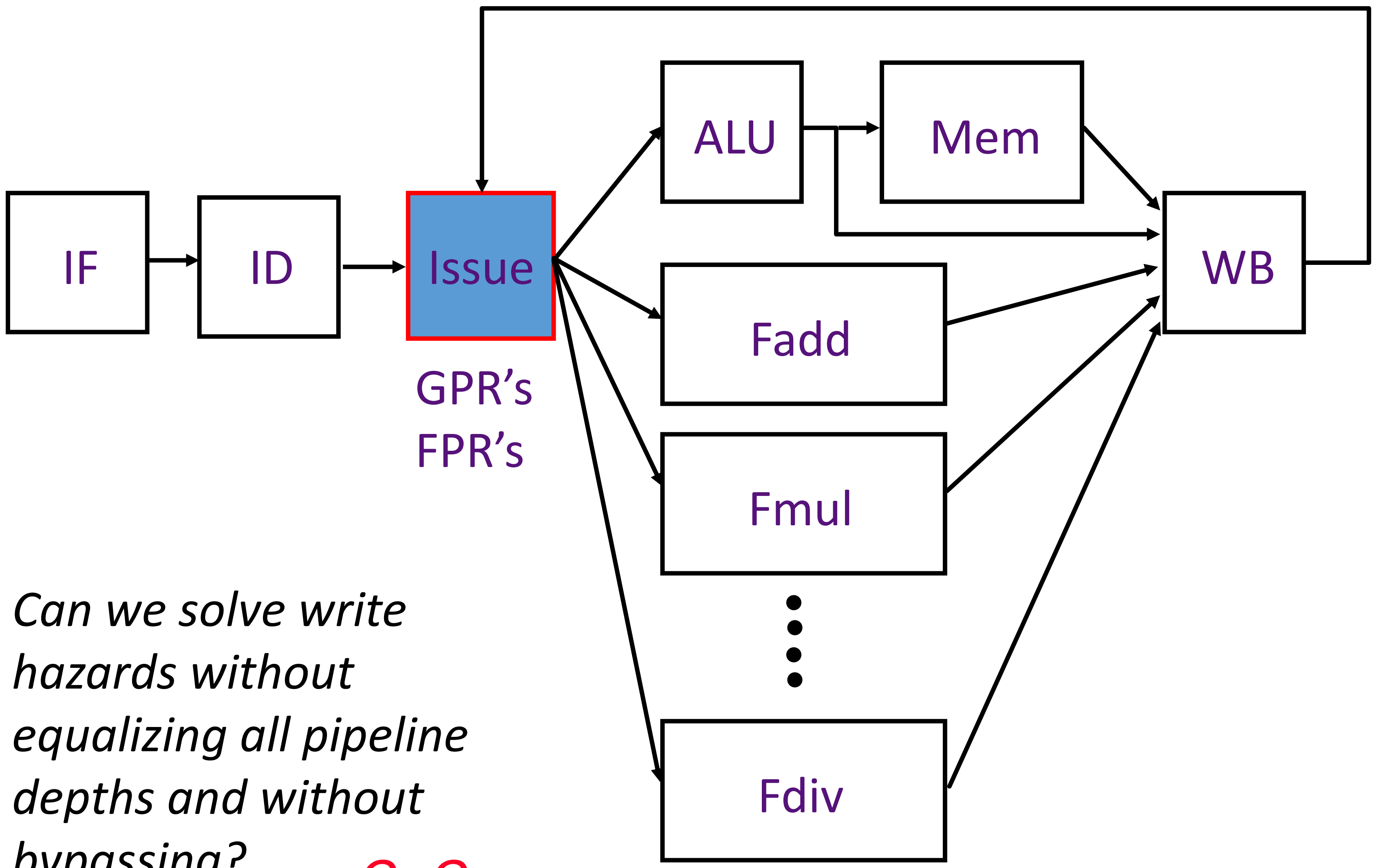


- Delay writeback so all operations have same latency to W stage
  - Write ports never oversubscribed (one inst. in & one inst. out every cycle)
  - Stall pipeline on long latency operations, e.g., divides, cache misses
  - Handle exceptions in-order at commit point

How to prevent increased writeback latency from slowing down single-cycle integer operations?

Bypassing

Complex Pipeline



Can we solve write hazards without equalizing all pipeline depths and without bypassing?

OoO

What is OOO?

- OOO execution is a type of processing where the instructions can begin execution as soon as operands are ready
- Instructions are issued in order however execution proceeds out of order
- Evolution

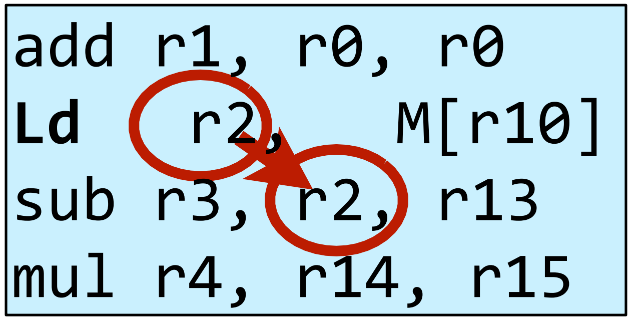
1964	CDC 6600
1966	IBM 360/91 Tomasulo's algorithm
1993	IBM/Motorola PowerPC 601
1995	Fujitsu/HAL SPARC64, Intel Pentium Pro
1996	MIPS R10000, AMD K5
1998	DEC Alpha 21264
2011	Sandy Bridge

Why Out-of-Order?

- Great for ...
  - tolerating variable latencies
  - finding ILP in code (instruction-level parallelism)
  - complex method for fine-grain data **prefetching**
  - plays nicely with poor compilers and lazily written code

- Downsides
  - complicated
  - expensive (area & power)

- Performance! (and easy to program!)



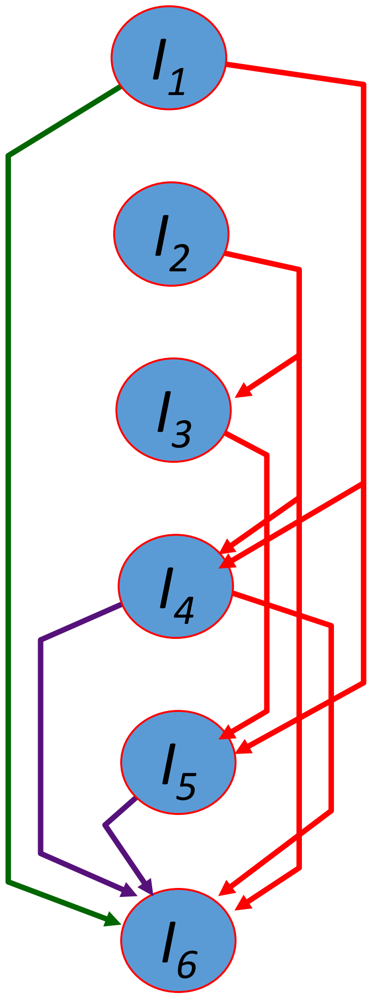
```
add r1, r0, r0
Ld   r2,   M[r10]
sub r3, r2, r13
mul r4, r14, r15
```

OoO is widely used in industry

- Intel Xeon/i-series (10-100W)
- ARM Cortex mobile chips (1W)
- Sun/Oracle Niagra UltraSPARC
- Intel Atom
- Play Station

Instruction Scheduling

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



Valid orderings:

in-order	I_1	I_2	I_3	I_4	I_5	I_6
out-of-order	I_2	I_1	I_3	I_4	I_5	I_6
out-of-order	I_1	I_2	I_3	I_5	I_4	I_6

When is it Safe to Issue an Instruction?

- Suppose a data structure keeps track of all the instructions in all the functional units

- The following checks need to be made before the Issue stage can dispatch an instruction
  - Is the required function unit available?
  - Is the input data available?   (RAW?)
  - Is it safe to write the destination? (WAR?WAW?)
  - Is there a structural conflict at the WB stage?

A Data Structure for Correct Issues

Keeps track of the status of Functional Units

Name	*Busy*	*Op*	*Dest*	*Src1*	*Src2*
Int					
Mem					
Add1					
Add2					
Add3					
Mult1					
Mult2					
Div					

The instruction i at the Issue stage consults this table

FU available?	check the busy column
RAW?	search the dest column for i's sources
WAR?	search the source columns for i's destination
WAW?	search the dest column for i's destination

An entry is added to the table if no hazard is detected;
An entry is removed from the table after Write-Back

Simplifying the Data Structure Assuming In-order Issue

- Suppose the instruction is not dispatched by the Issue stage if a RAW hazard exists or the required FU is busy, and that operands are latched by functional unit on issue:

- Can the dispatched instruction cause a
  - WAR hazard ?

    NO: Operands read at issue

  - WAW hazard ?

    YES: Out-of-order completion

Simplifying the Data Structure ...

- No WAR hazard
  - →no need to keep src1 and src2

- The Issue stage does not dispatch an instruction in case of a WAW hazard
  - → a register name can occur at most once in the dest column

- WP[reg#] : a bit-vector to record the registers for which writes are pending
  - These bits are set by the Issue stage and cleared by the WB stage
  - → Each pipeline stage in the FU's must carry the register destination field and a flag to indicate if it is valid

Scoreboard for In-order Issues

- Busy[FU#] : a bit-vector to indicate FU's availability.
  `(FU = Int, Add, Mult, Div)`
  - These bits are hardwired to FU's.
- WP[reg#] : a bit-vector to record the registers for which writes are pending.
  - These bits are set by Issue stage and cleared by WB stage
- Issue checks the instruction (opcode dest src1 src2)
  - against the scoreboard (Busy & WP) to dispatch

– **FU available?**	Busy[FU#]
– **RAW?**	WP[src1] or WP[src2]
– **WAR?**	*cannot arise*
– **WAW?**	WP[dest]

Scoreboard Dynamics

		Functional Unit Status										Registers Reserved for Writes	
		Int(1)	Add(1)	Mult(3)			Div(4)				WB		
t0	I_1						f6					f6	
t1	I_2	f2						f6				f6, f2	
t2									f6		f2	f6, f2	I_2
t3	I_3			f0						f6		f6, f0	
t4					f0						f6	f6, f0	I_1
t5	I_4					f0	f8					f0, f8	
t6								f8			f0	f0, f8	I_3
t7	I_5		f10						f8			f8, f10	
t8										f8	f10	f8, f10	I_5
t9											f8	f8	I_4
t10	I_6		f6									f6	
t11											f6	f6	I_6

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

Out-of-order Completion
In-order Issue

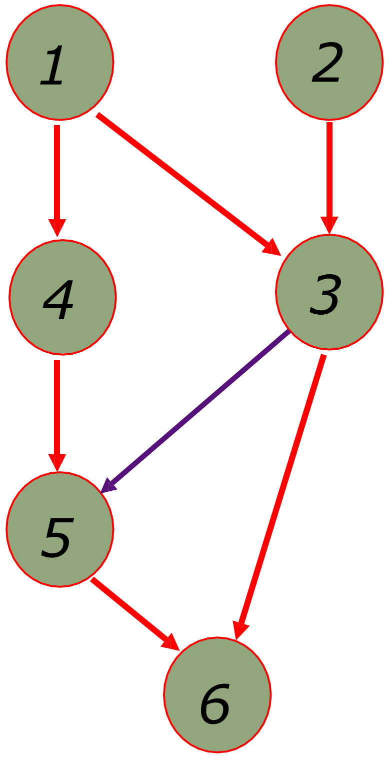
					Latency
I_1	FDIV.D	f6,	f6,	f4	*4*
I_2	FLD	f2,	45(x3)		*1*
I_3	FMULT.D	f0,	f2,	f4	*3*
I_4	FDIV.D	f8,	f6,	f2	*4*
I_5	FSUB.D	f10,	f0,	f6	*1*
I_6	FADD.D	f6,	f8,	f2	*1*

in-order comp 1 2 $\underline{1}$ $\underline{2}$ 3 4 $\underline{3}$ 5 $\underline{4}$ 6 $\underline{5}$ $\underline{6}$

out-of-order comp 1 2 $\underline{2}$ 3 $\underline{1}$ 4 $\underline{3}$ 5 $\underline{5}$ $\underline{4}$ 6 $\underline{6}$

In-Order Issue Limitations: an example

					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*

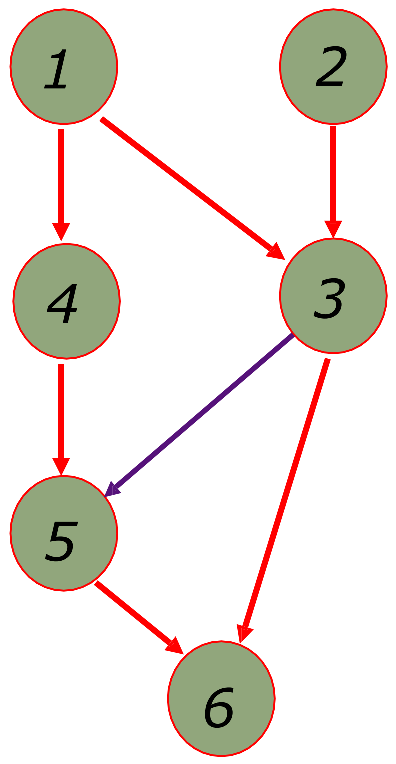

$$\begin{tikzcd}
1 \arrow[d] \arrow[dr] & 2 \arrow[d] \\
4 \arrow[d] & 3 \arrow[dl] \arrow[dd] \\
5 \arrow[dr] & \\
 & 6
\end{tikzcd}$$

In-order: 1 (2,<u>1</u>) . . . . . . <u>2</u> 3 4 <u>4</u> <u>3</u> 5 . . . <u>5</u> 6 <u>6</u>

In-order issue restriction prevents instruction 4 from being dispatched

In-Order Issue Limitations: an example

					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*



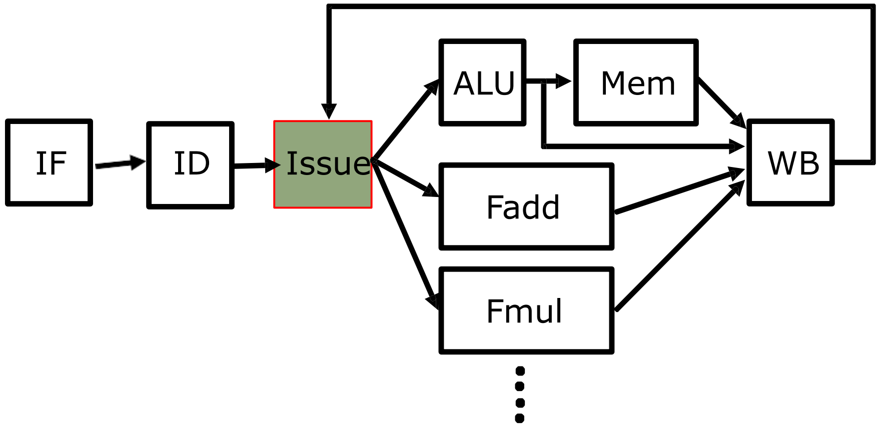
In-order: 1 (2,$\underline{1}$) . . . . . . $\underline{2}$ 3 4 $\underline{4}$ $\underline{3}$ 5 . . . $\underline{5}$ 6 $\underline{6}$

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 did not allow any significant improvement!

Out-of-Order Issue

- Issue stage buffer holds multiple instructions waiting to issue.
- Decode adds next instruction to buffer if there is space and the instruction does not cause a WAR or WAW hazard.
  - Note: WAR possible again because issue is out-of-order (WAR not possible with in-order issue and latching of input operands at functional unit)
- Any instruction in buffer whose RAW hazards are satisfied can be issued (for now, at most one dispatch per cycle). On a write back (WB), new instructions may get enabled.



How many Instructions can be in the pipeline

Which features of an ISA limit the number of instructions in the pipeline?

Number of Registers

Which features of a program limit the number of instructions in the pipeline?

Control transfers

Out-of-order dispatch by itself does not provide any significant performance improvement !

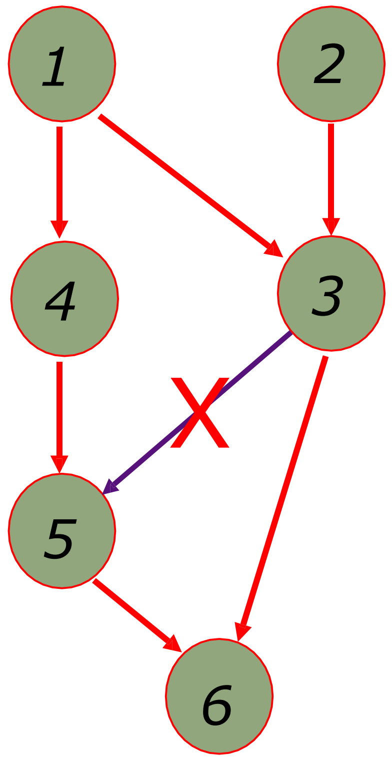
Overcoming the Lack of Register Names

- Floating Point pipelines often cannot be kept 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 ?
    - Robert Tomasulo of IBM suggested an ingenious solution in 1967 based on on-the-fly register renaming

Issue Limitations: In-Order and Out-of-Order

					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,$\underline{1}$) . . . . . . $\underline{2}$ 3 4 $\underline{4}$ $\underline{3}$ 5 . . . $\underline{5}$ 6 $\underline{6}$

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

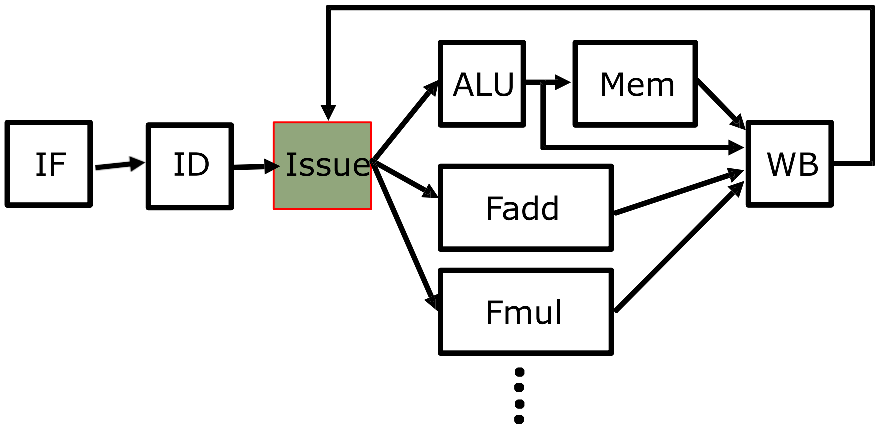
Any antidependence can be eliminated by renaming.
(renaming $\Rightarrow$ additional storage)
Can it be done in hardware?

yes!

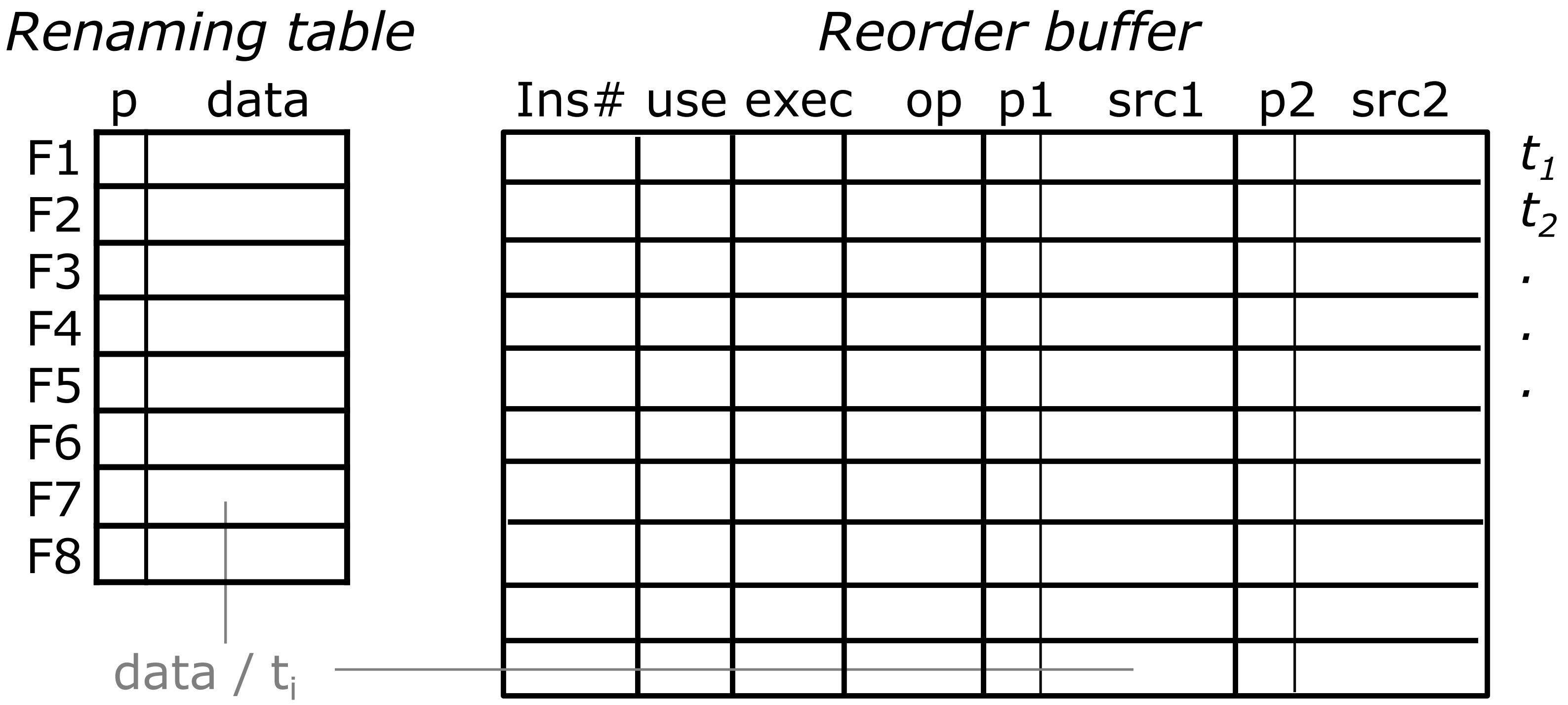
Register Renaming

- Decode does register renaming and adds instructions to the issue stage reorder buffer (ROB)
  - renaming makes WAR or WAW hazards impossible

- Any instruction in ROB whose RAW hazards have been satisfied can be dispatched.
  - Out-of-order or dataflow execution



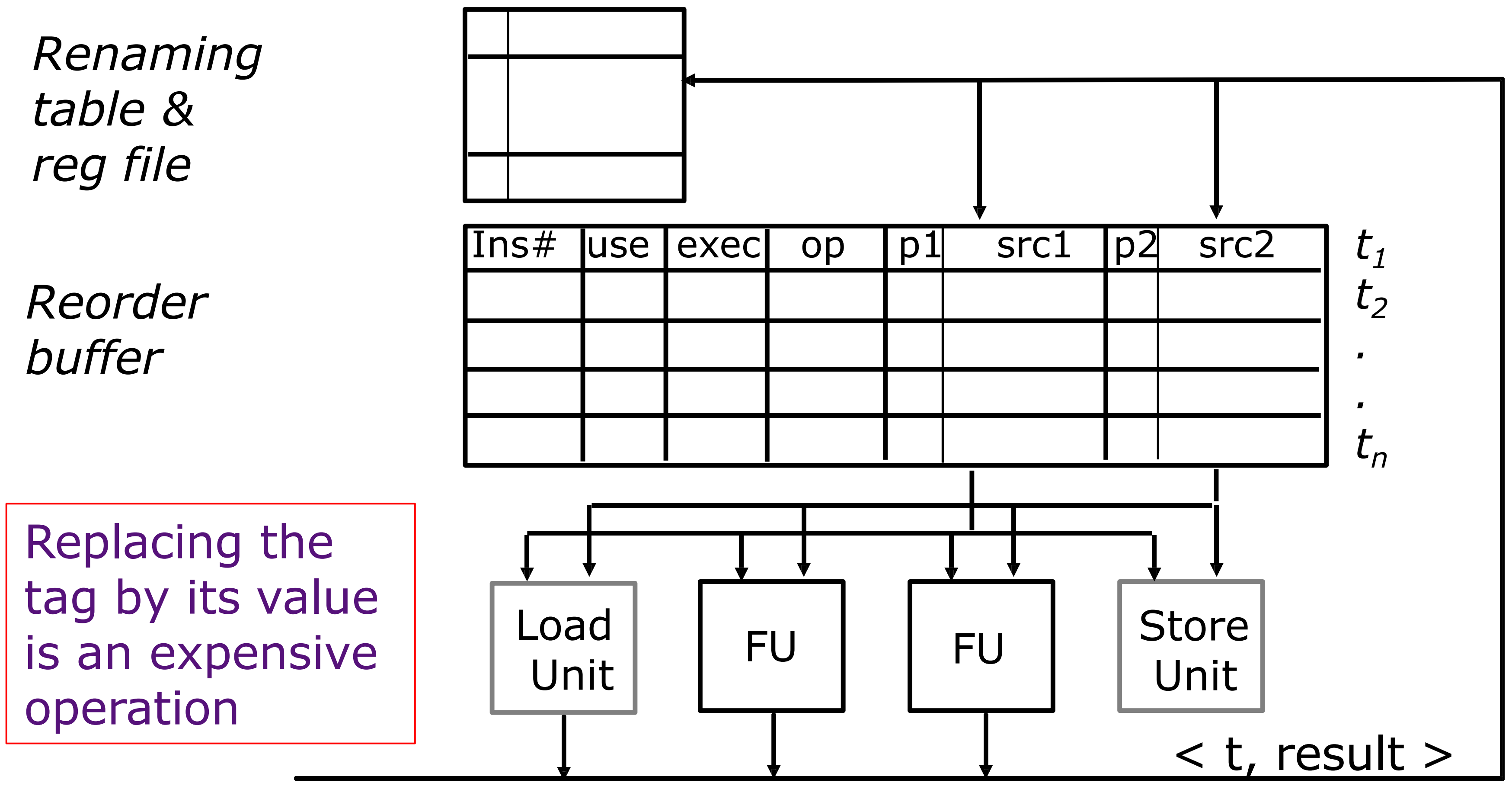
Renaming & Out-of-order Issue An example



1	LD	F2,	34(R2)	
2	LD	F4,	45(R3)	
3	MULTD	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?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

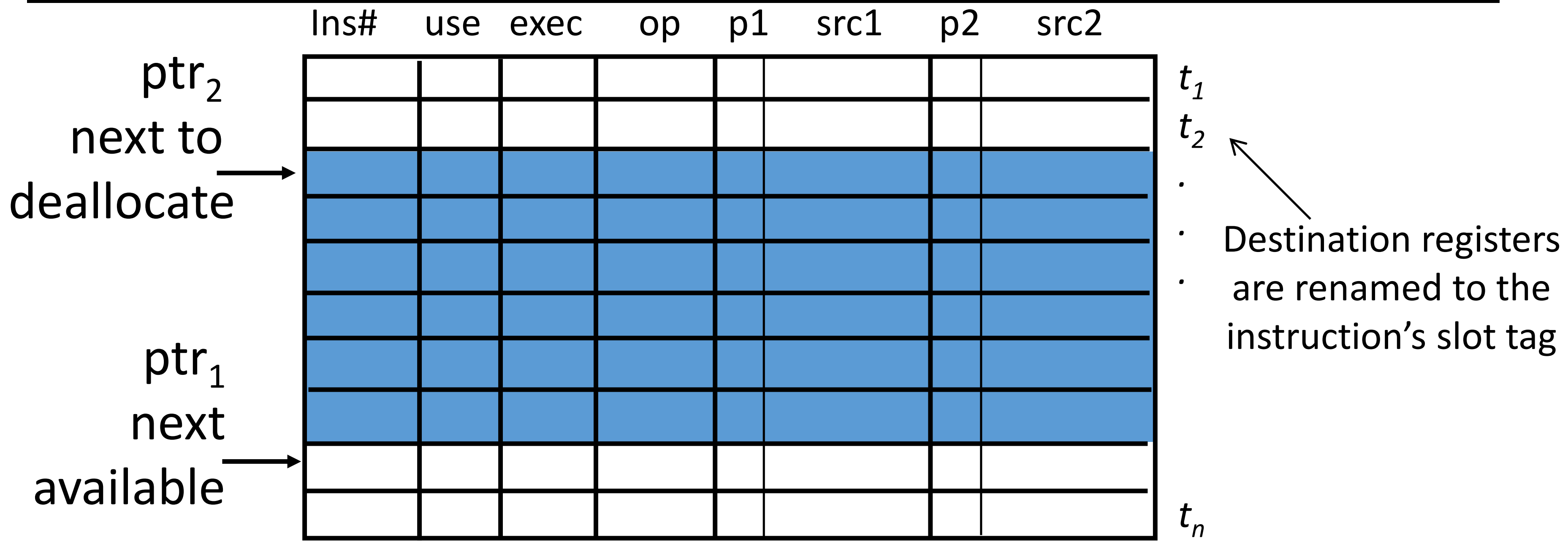
Data-Driven Execution



Replacing the tag by its value is an expensive operation

- Instruction template (i.e., tag t) is allocated by the Decode stage, which also stores the tag in the reg file
- When an instruction completes, its tag is deallocated

Reorder Buffer Management



Ins#
use
exec
op
p1
src1
p2
src2
ptr$_2$ next to deallocate
ptr$_1$ next available
t_1
t_2
t_n
Destination registers are renamed to the instruction's slot tag

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)

Renaming & Out-of-order Issue

An example

Renaming table

	p	data
f1		
f2		t1 v1
f3		
f4		t2 t5
f5		
f6		t3
f7		
f8		t4 v4

data / t_i

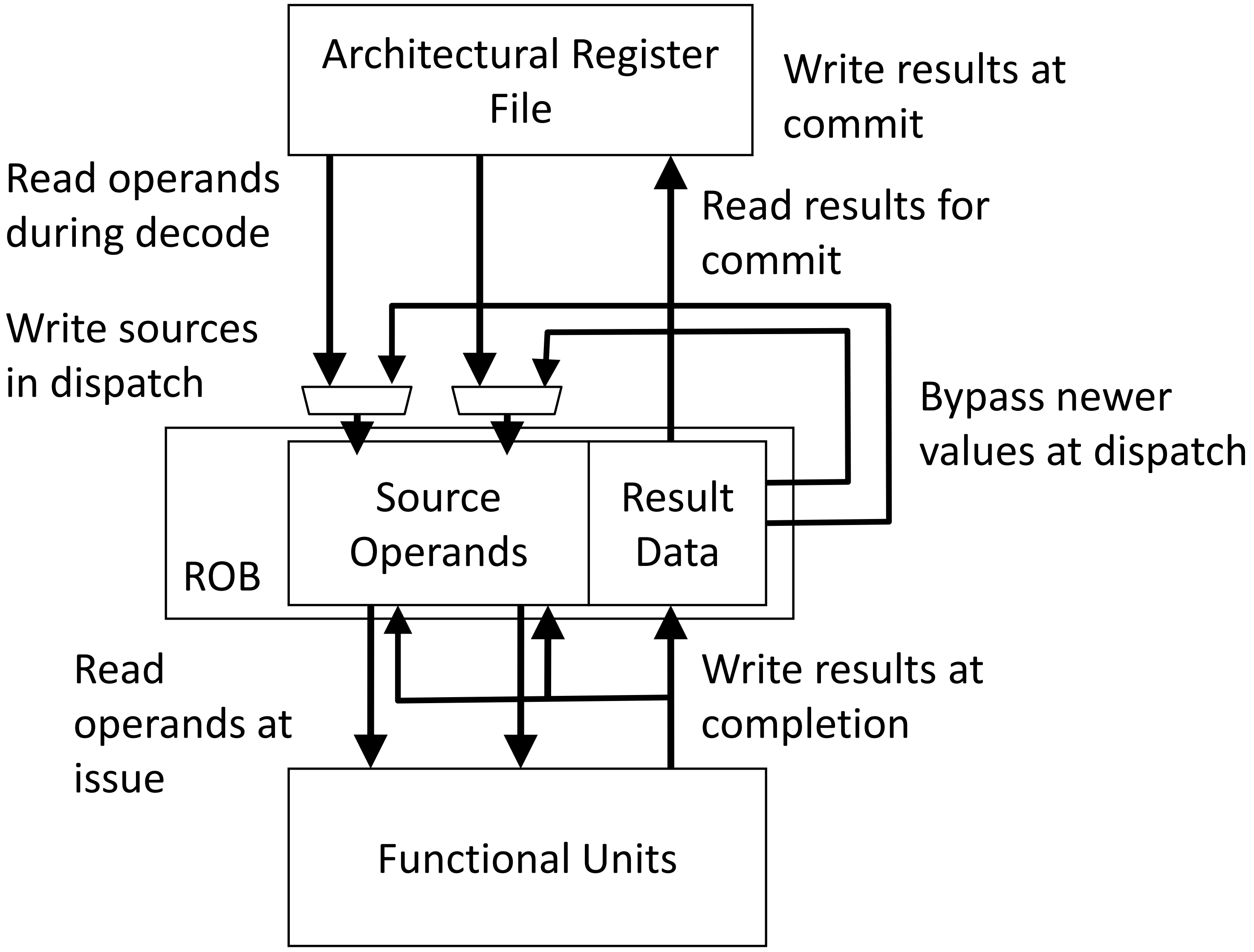
Reorder buffer

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

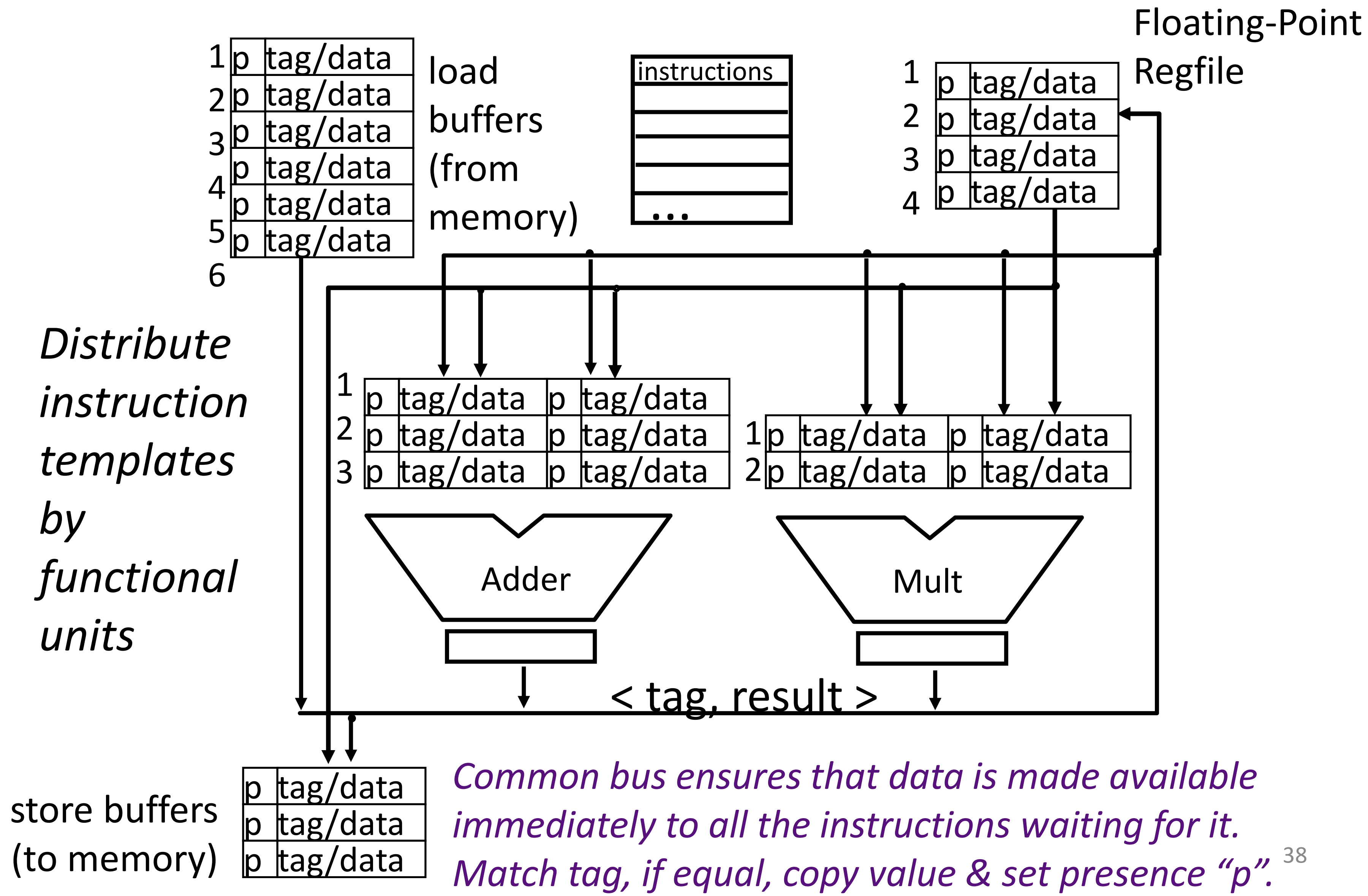
1 FLD	f2,	34(x2)	
2 FLD	f4,	45(x3)	
3 FMULT.D	f6,	f4,	f2
4 FSUB.D	f8,	f2,	f2
5 FDIV.D	f4,	f2,	f8
6 FADD.D	f10,	f6,	f4

- *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


Architectural Register File
Write results at commit
Read operands during decode
Read results for commit
Write sources in dispatch
Bypass newer values at dispatch
Source Operands
Result Data
ROB
Read operands at issue
Write results at completion
Functional Units

IBM 360/91 Floating-Point Unit
R. M. Tomasulo, 1967



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

One more problem needed to be solved

Control transfers

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

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

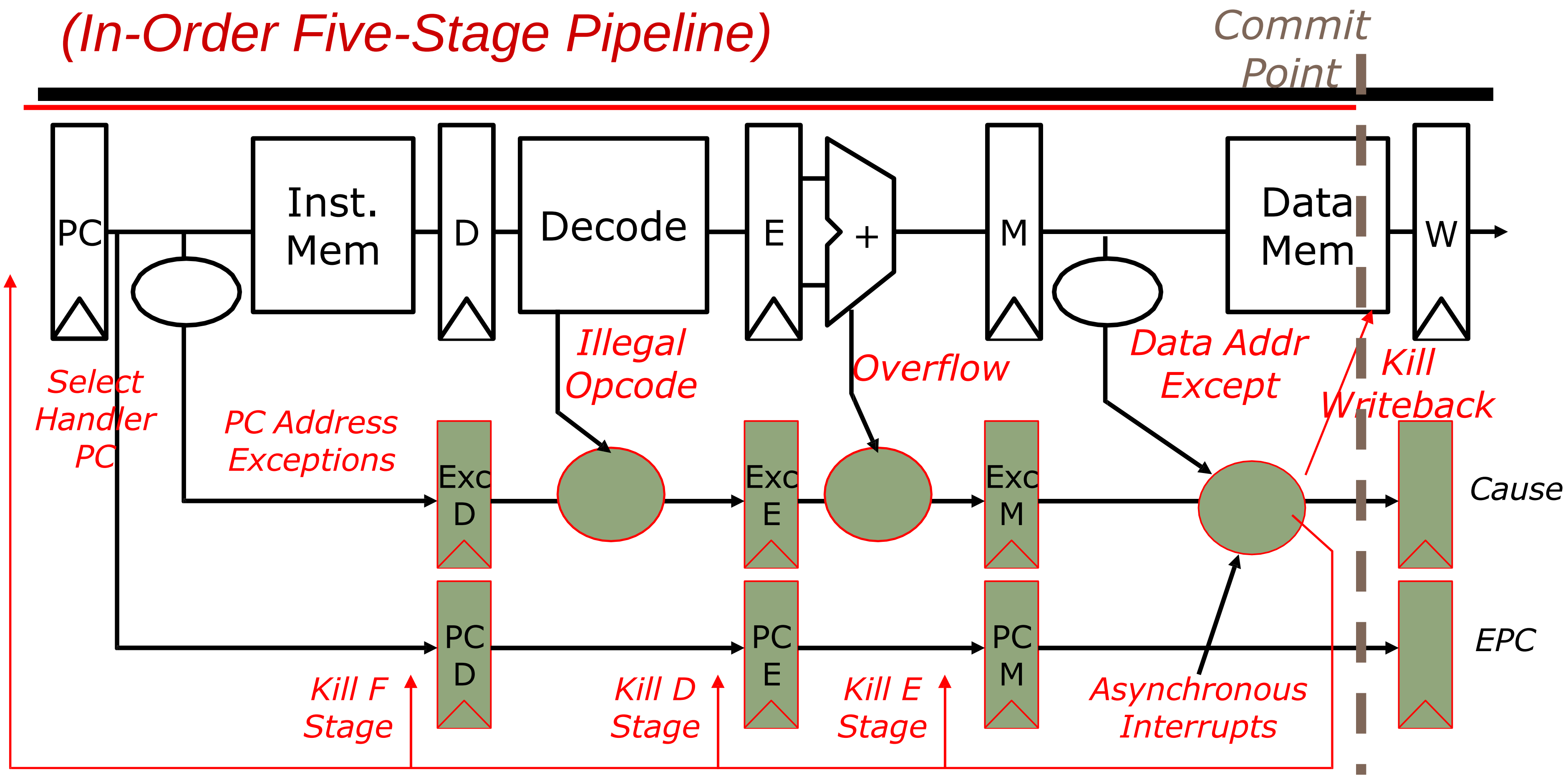
out-of-order comp 1 2 $\underline{2}$ 3 $\underline{1}$ 4 $\underline{3}$ 5 $\underline{5}$ $\underline{4}$ 6 $\underline{6}$

restore f2 *restore f10*

Consider interrupts

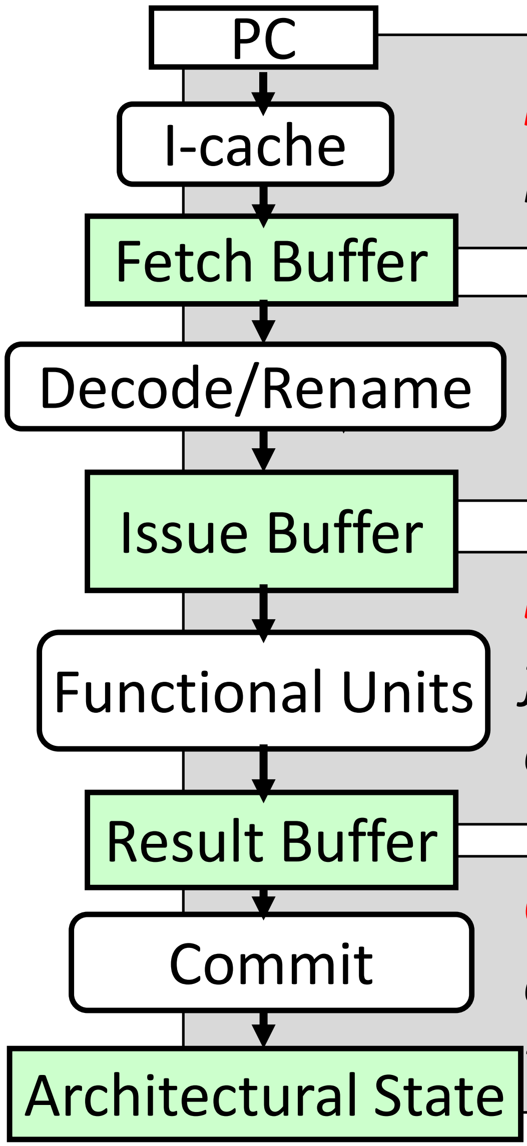
Precise interrupts are difficult to implement at high speed
- want to start execution of later instructions before exception checks finished on earlier instructions

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

Phases of Instruction Execution



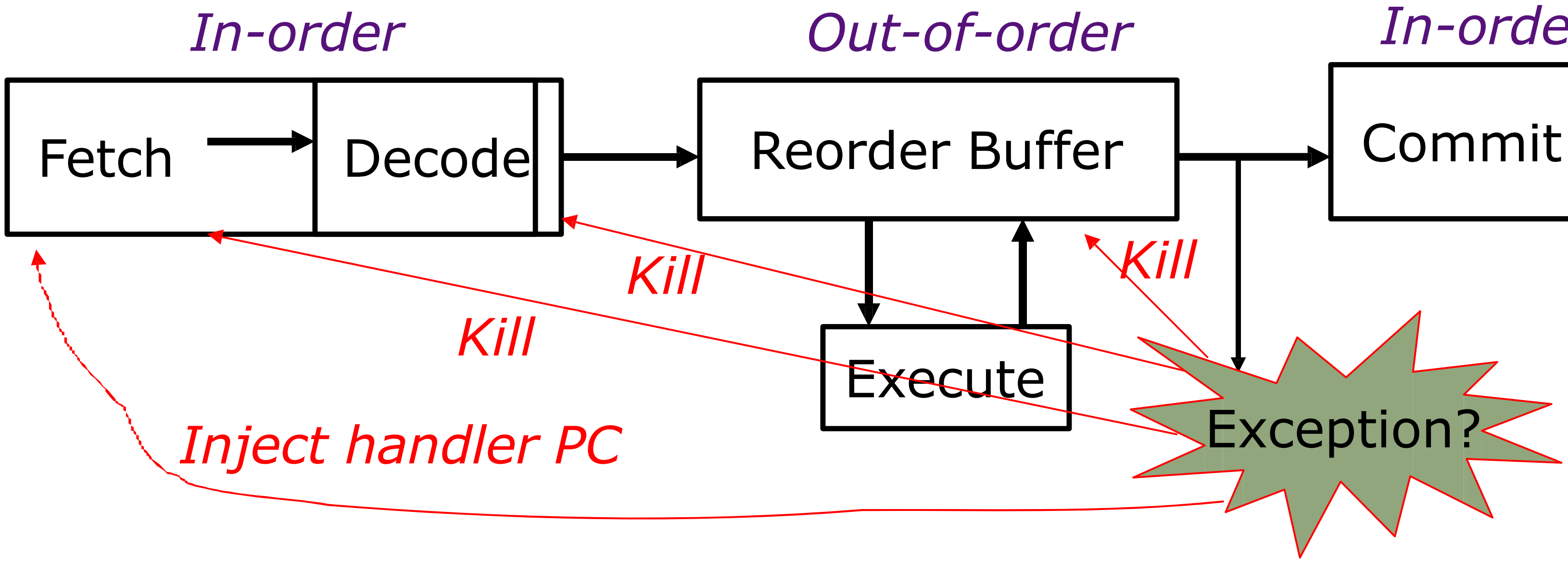
***Fetch**: Instruction bits retrieved from instruction cache.*

***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.*

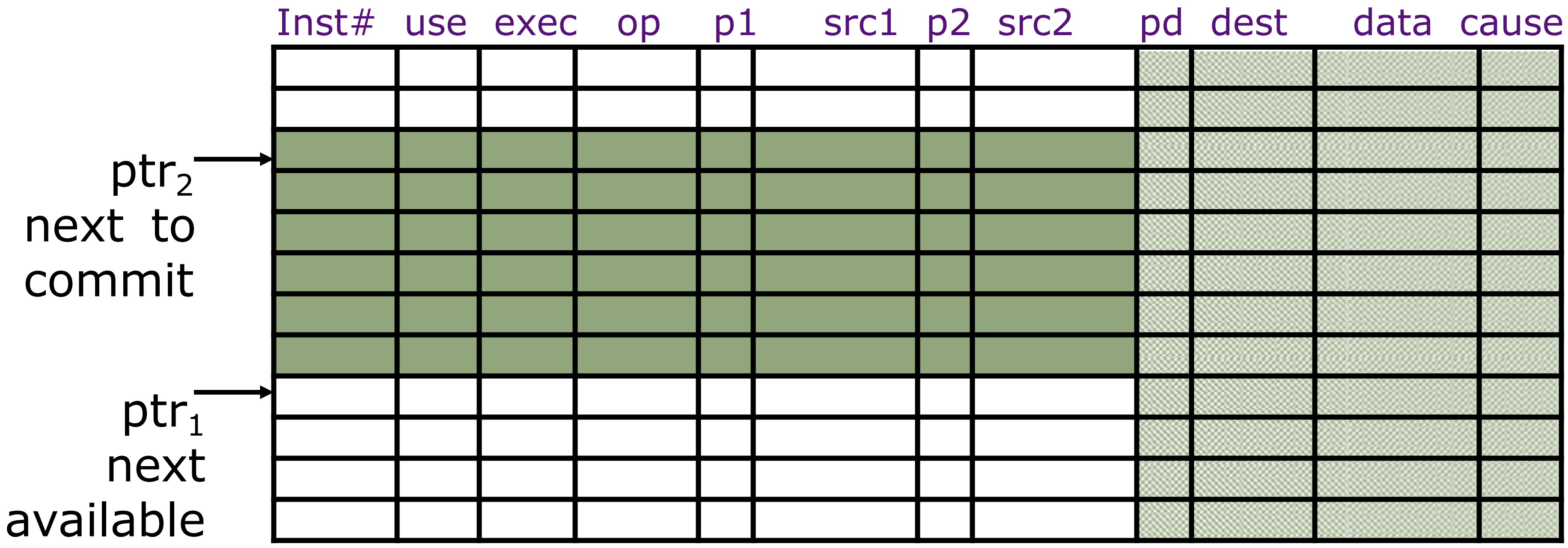
In-Order Commit for Precise Exceptions



- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order (⇒ 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

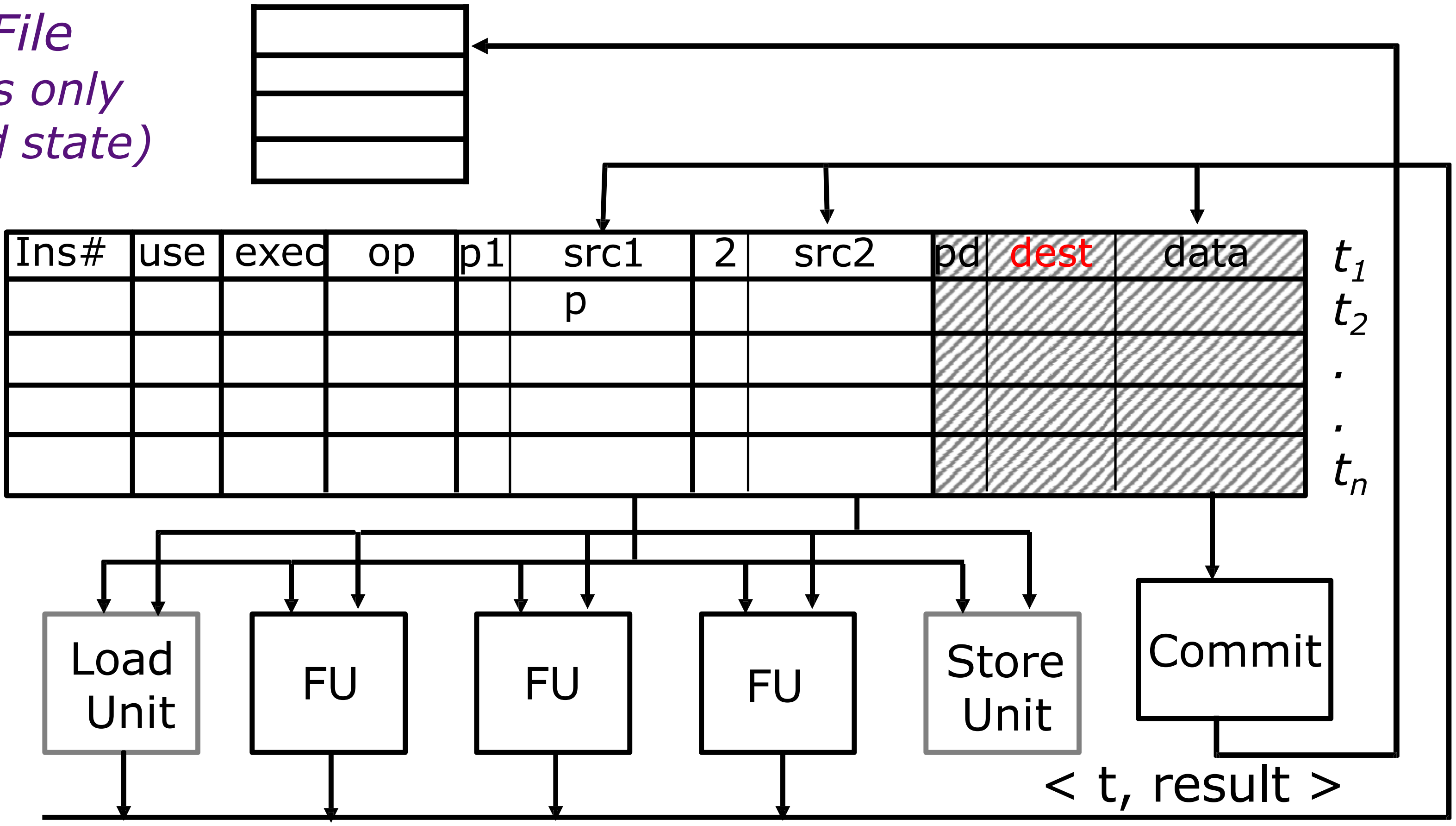
- add <pd, dest, data, cause> fields in the instruction template
- commit instructions to reg file and memory in program order ⇒ buffers can be maintained circularly
- on exception, clear reorder buffer by resetting $ptr_1=ptr_2$

  (stores must wait for commit before updating memory)

Rollback and Renaming
(HP PA8000, Pentium Pro, Core2Duo, Nehalem)

Register File
(now holds only committed state)

Reorder buffer

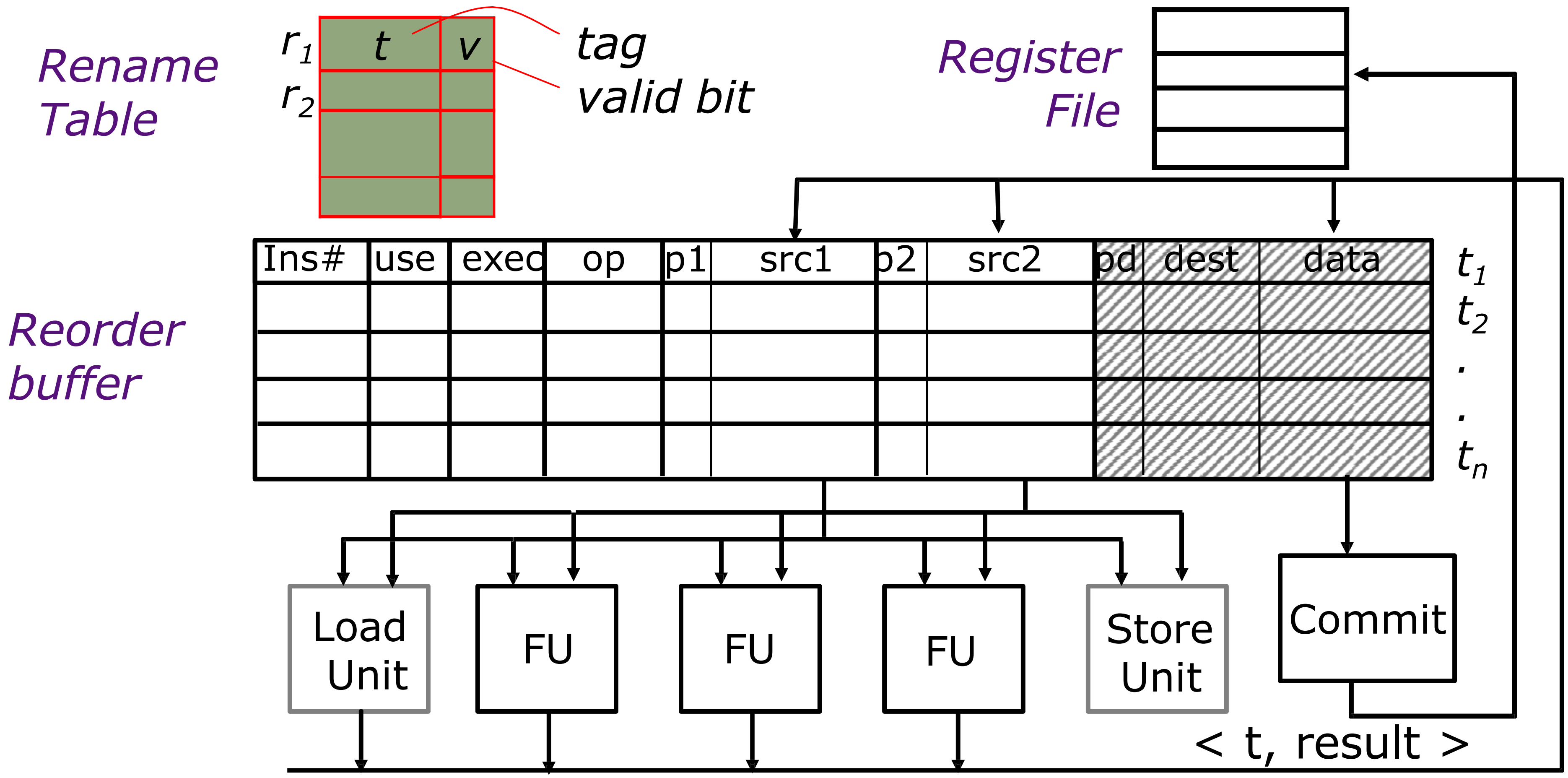


Register file does not contain renaming tags any more.

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

Search the "dest" field in the reorder buffer

Renaming Table



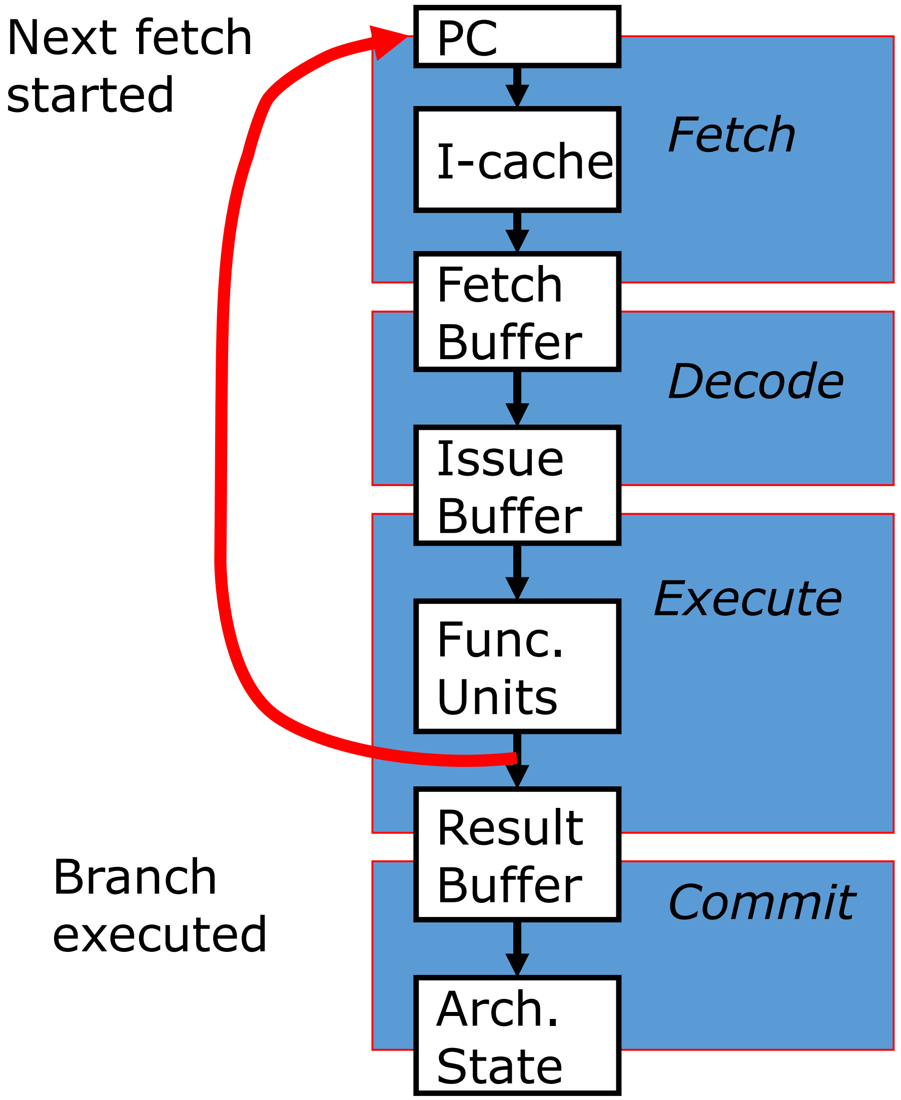
Renaming table is a cache to speed up register name look up.
It needs to be cleared after each exception taken.
When else are valid bits cleared? *Control transfers*

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 + buffers



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

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

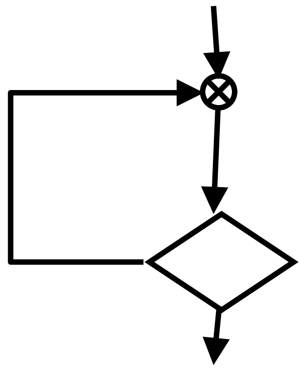
- On a mispredict, could throw away 8*4+(80-1)=111 instructions

- 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

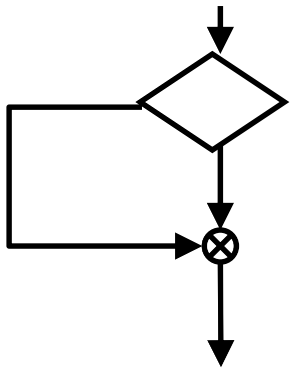
Static Branch Prediction

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)

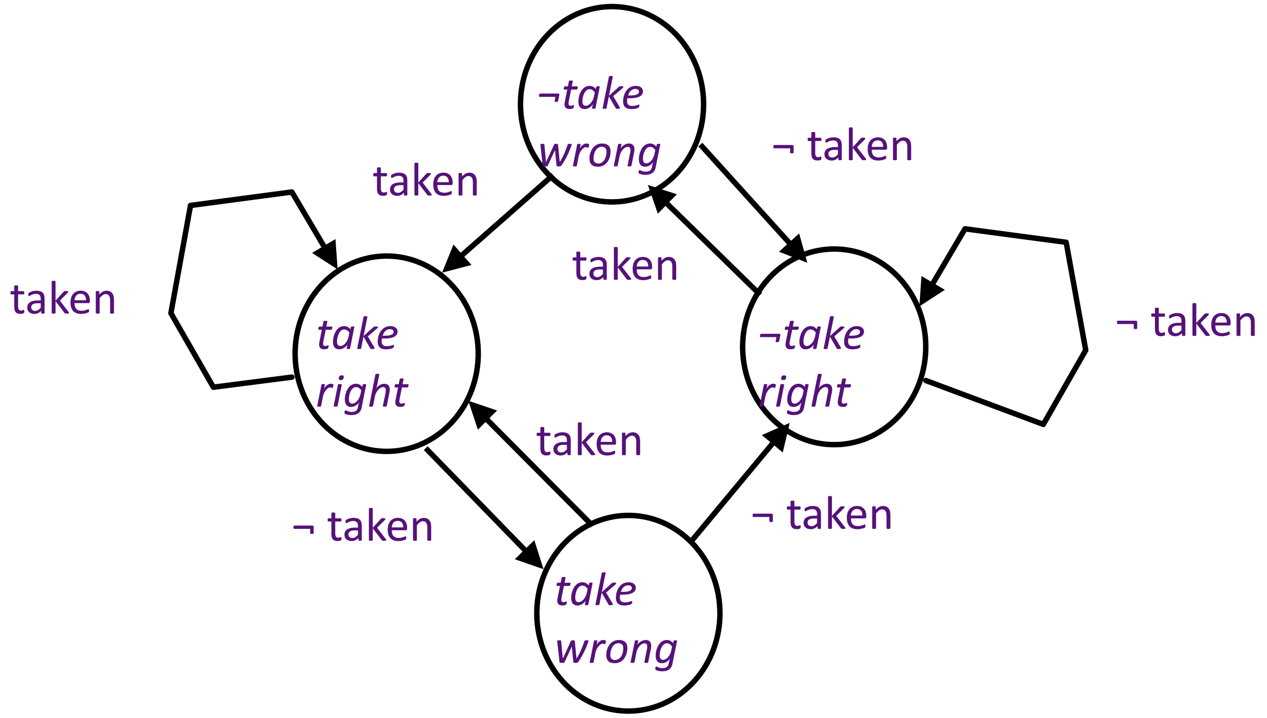
One-Bit Branch History Predictor

- 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

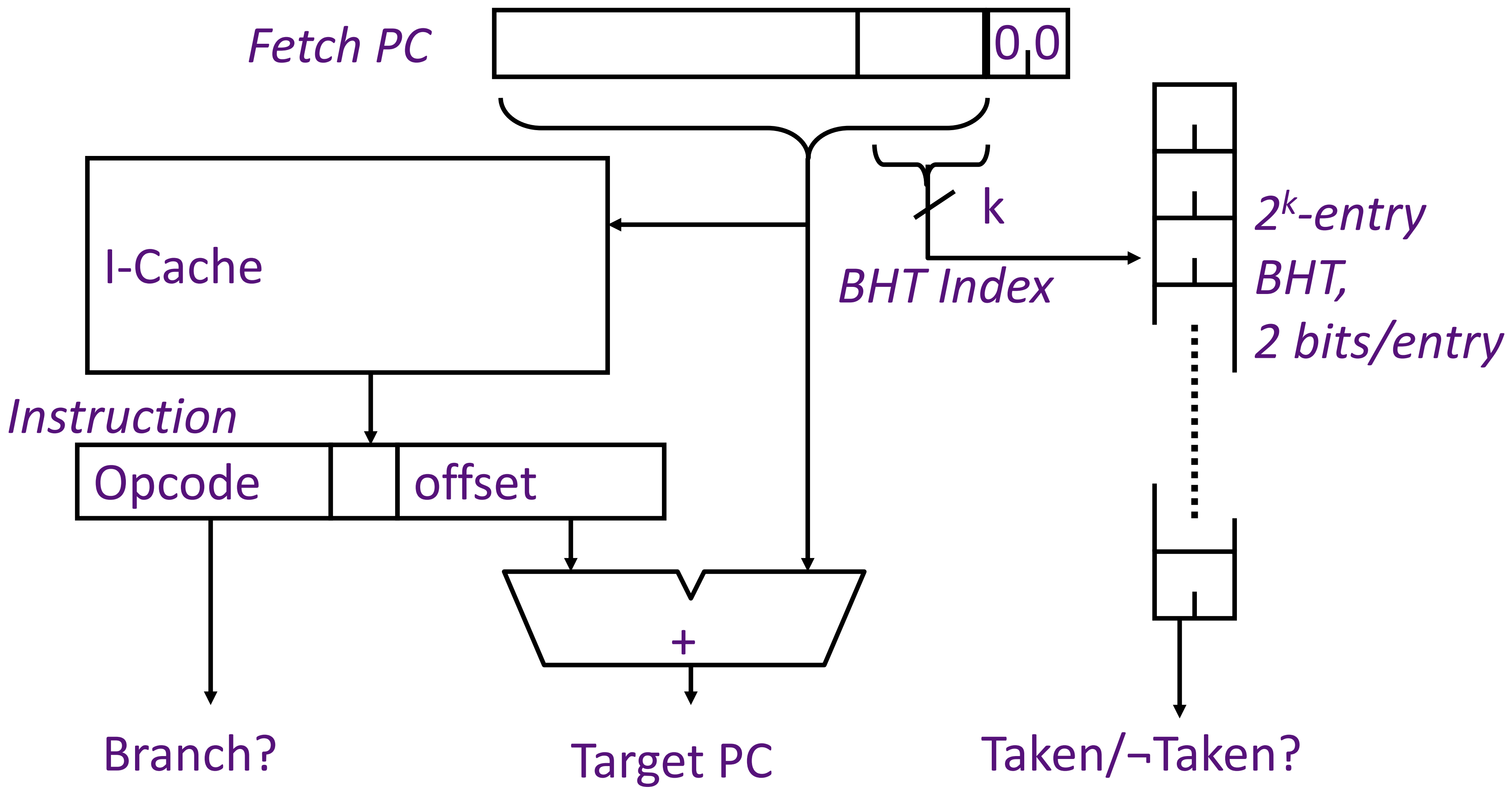
- Assume 2 BP bits per instruction
- Change the prediction after two consecutive mistakes!



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

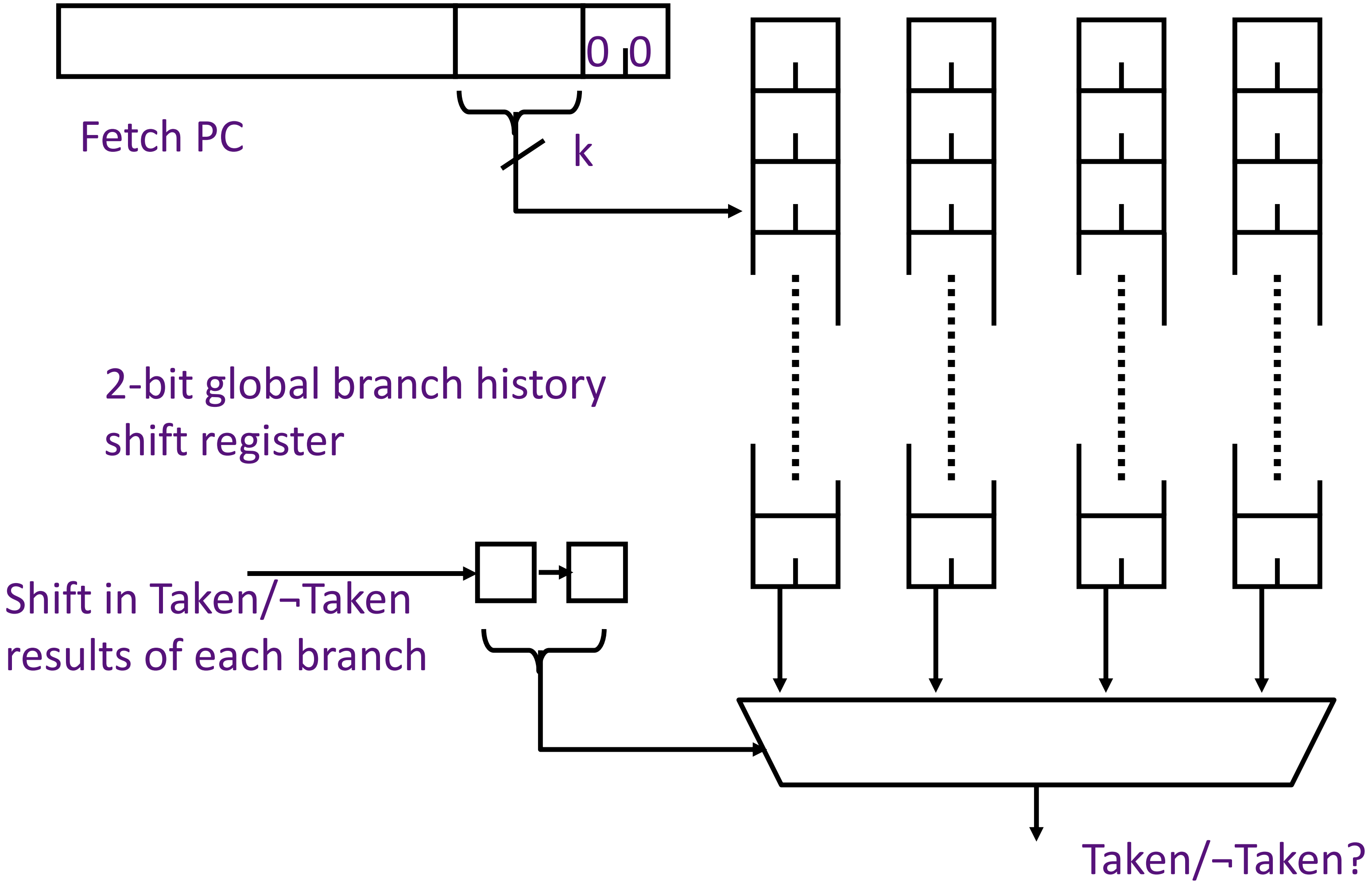
```
if (x[i] < 7) then
      y += 1;
if (x[i] < 5) then
      c -= 4;
```

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)



0 0
Fetch PC
k
2-bit global branch history shift register
Shift in Taken/¬Taken results of each branch
Taken/¬Taken?

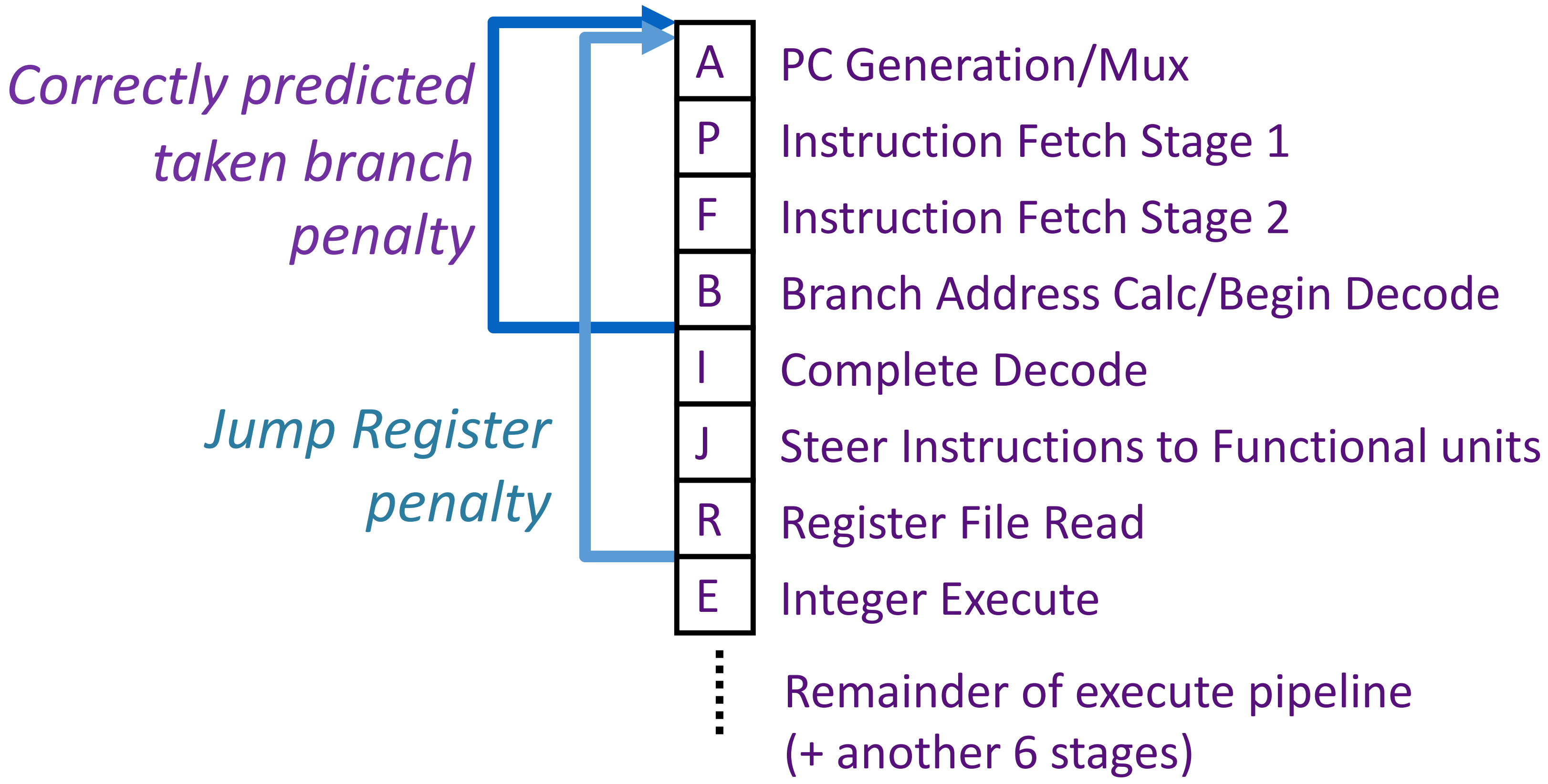
Speculating Both Directions?

- 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!

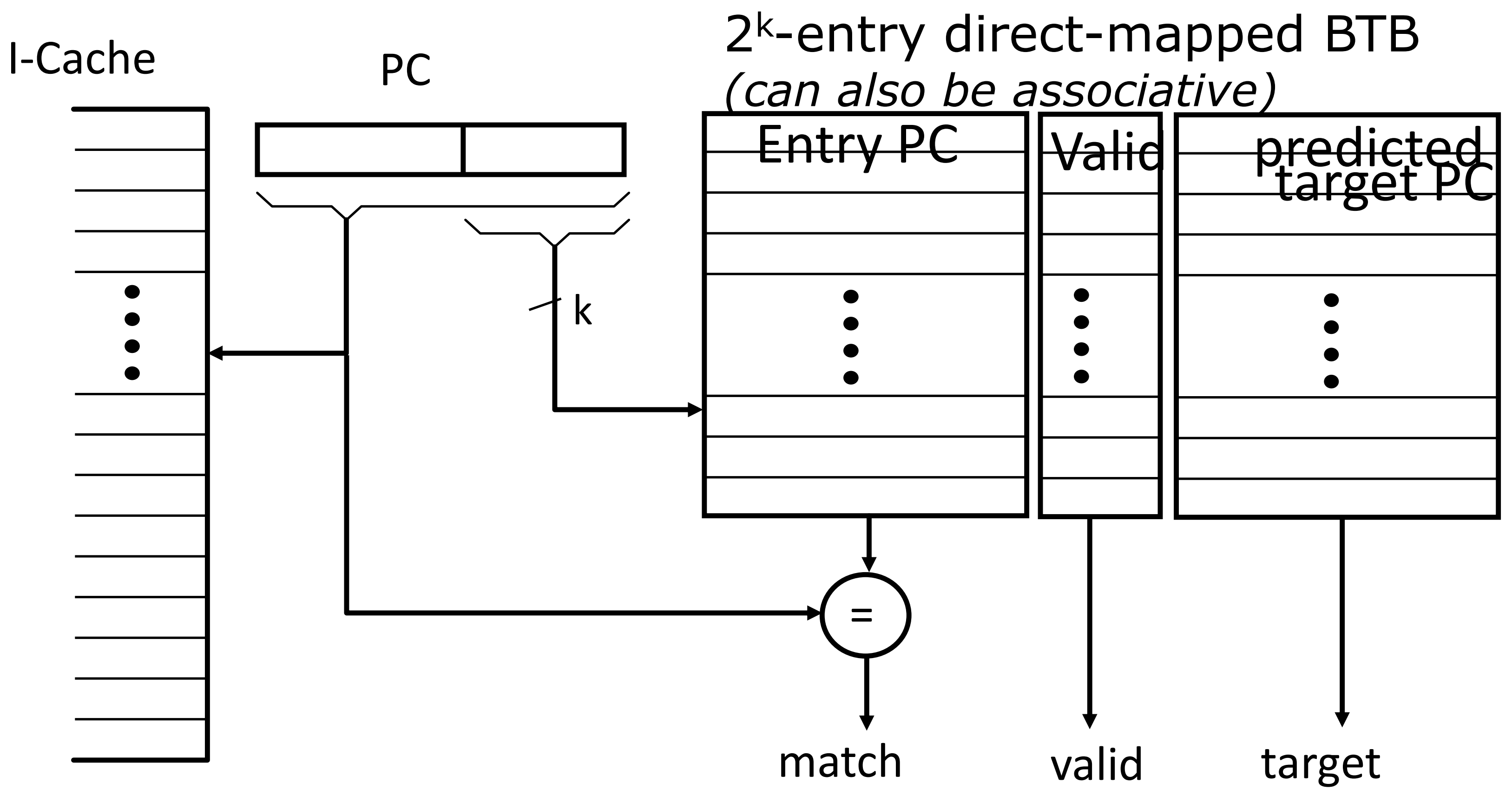
Limitations of BHTs

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


Correctly predicted taken branch penalty
Jump Register penalty
A
P
F
B
I
J
R
E
PC Generation/Mux
Instruction Fetch Stage 1
Instruction Fetch Stage 2
Branch Address Calc/Begin Decode
Complete Decode
Steer Instructions to Functional units
Register File Read
Integer Execute
Remainder of execute pipeline (+ another 6 stages)

UltraSPARC-III fetch pipeline

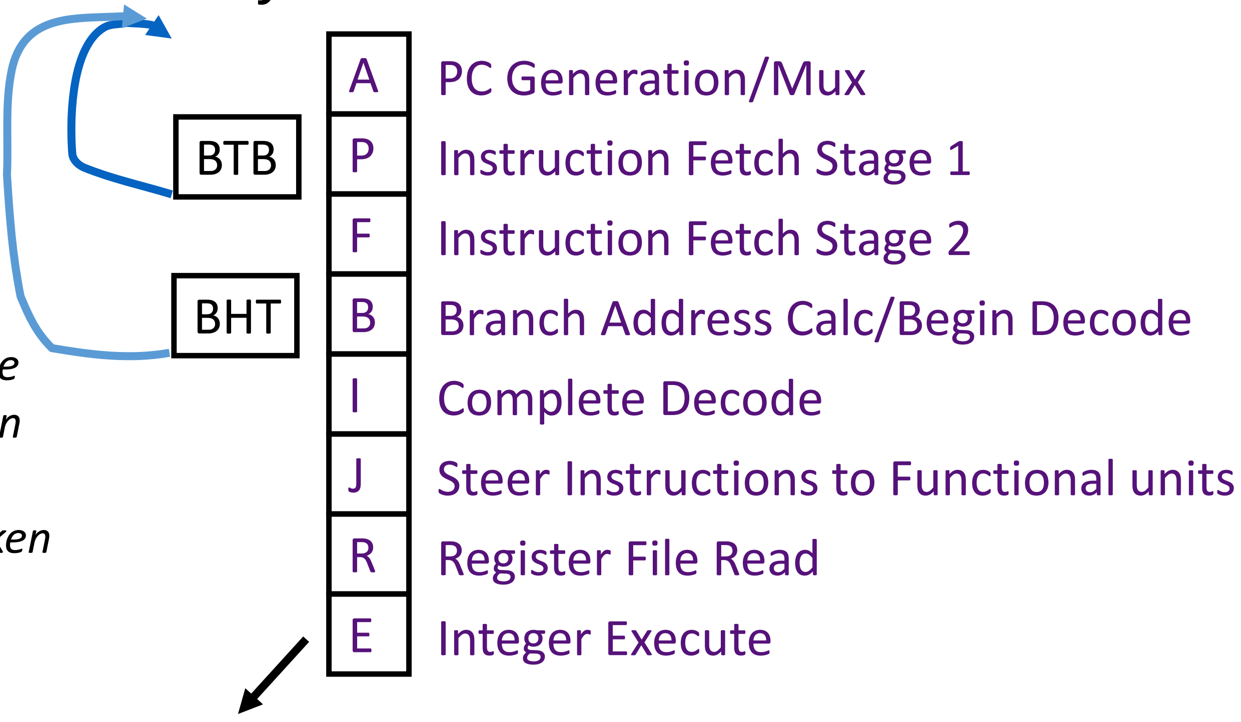
Branch Target Buffer (BTB)



- 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
A
P
F
B
I
J
R
E
PC Generation/Mux
Instruction Fetch Stage 1
Instruction Fetch Stage 2
Branch Address Calc/Begin Decode
Complete Decode
Steer Instructions to Functional units
Register File Read
Integer Execute

BHT in later pipeline stage corrects when BTB misses a predicted taken branch

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

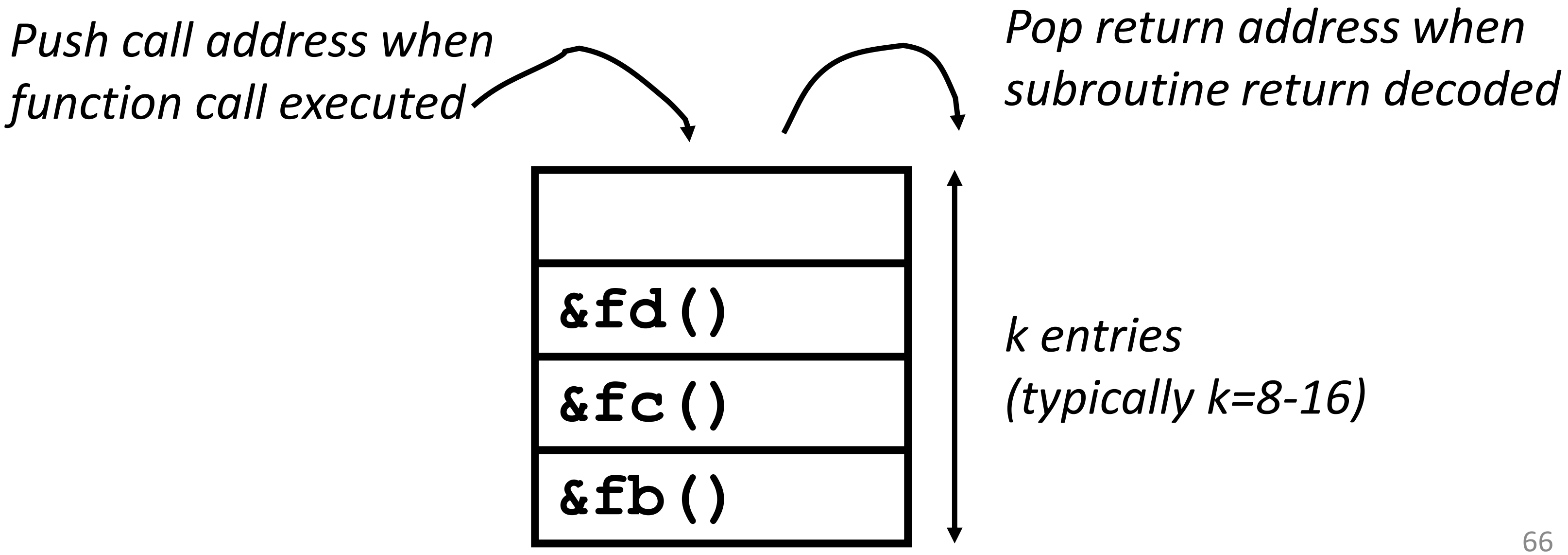
  $\Rightarrow$ *Often one function called from many distinct call sites!*

  How well does BTB work for each of these cases?

Subroutine Return Stack

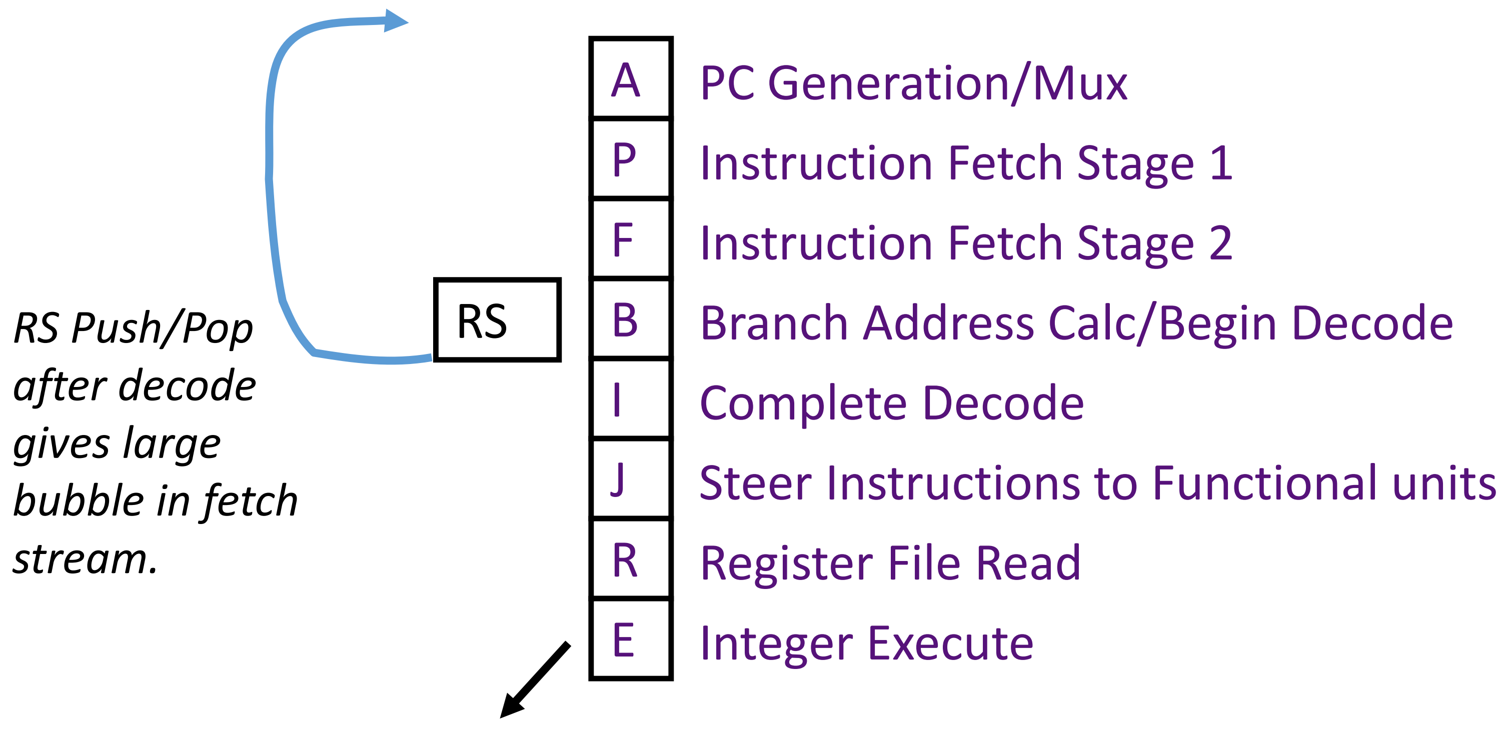
Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

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


Push call address when function call executed
Pop return address when subroutine return decoded
&fd()
&fc()
&fb()
k entries (typically k=8-16)

Return Stack in Pipeline

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

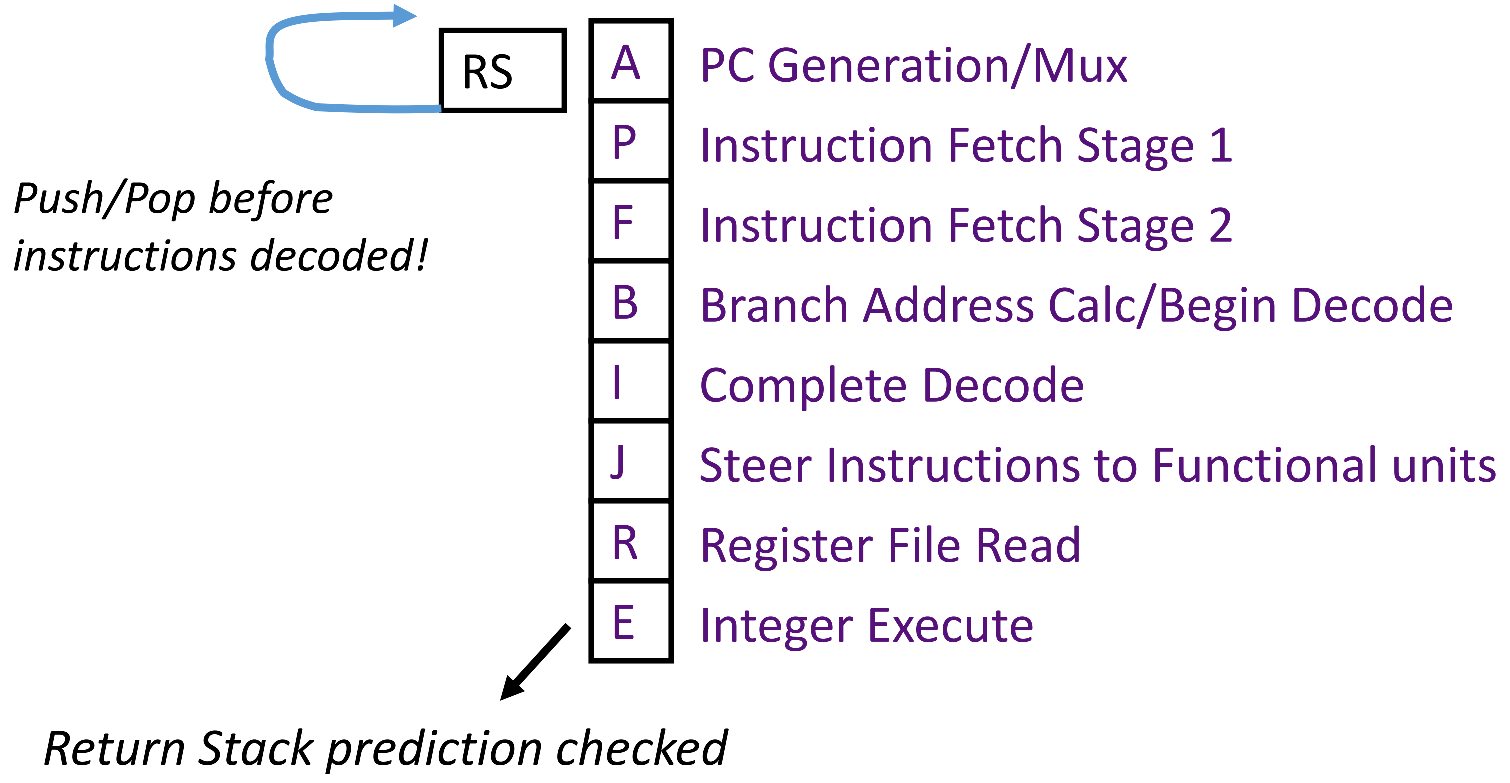


RS Push/Pop after decode gives large bubble in fetch stream.
RS
A PC Generation/Mux
P Instruction Fetch Stage 1
F Instruction Fetch Stage 2
B Branch Address Calc/Begin Decode
I Complete Decode
J Steer Instructions to Functional units
R Register File Read
E Integer Execute

Return Stack prediction checked

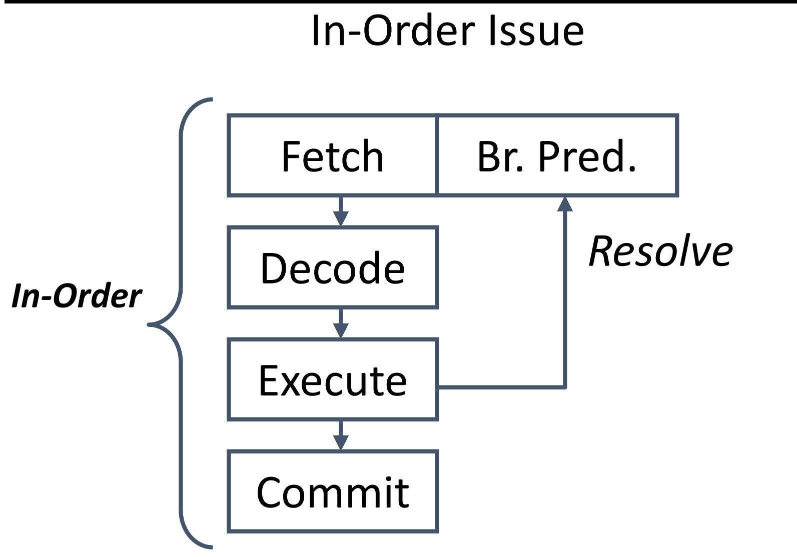
Return Stack in Pipeline

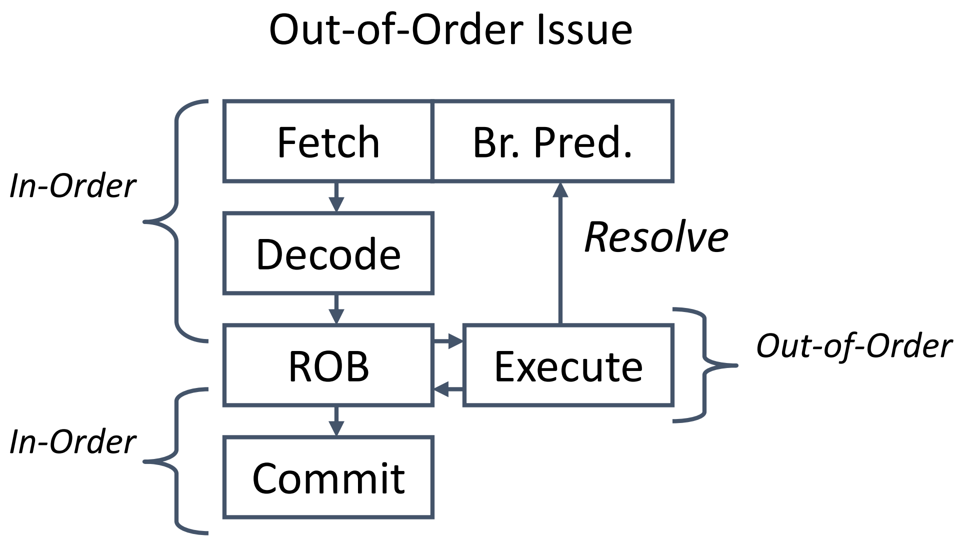
- Can remember whether PC is subroutine call/return using BTB-like structure
- Instead of target-PC, just store push/pop bit



RS
A
PC Generation/Mux
P
Instruction Fetch Stage 1
F
Instruction Fetch Stage 2
B
Branch Address Calc/Begin Decode
I
Complete Decode
J
Steer Instructions to Functional units
R
Register File Read
E
Integer Execute
Push/Pop before instructions decoded!
Return Stack prediction checked

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

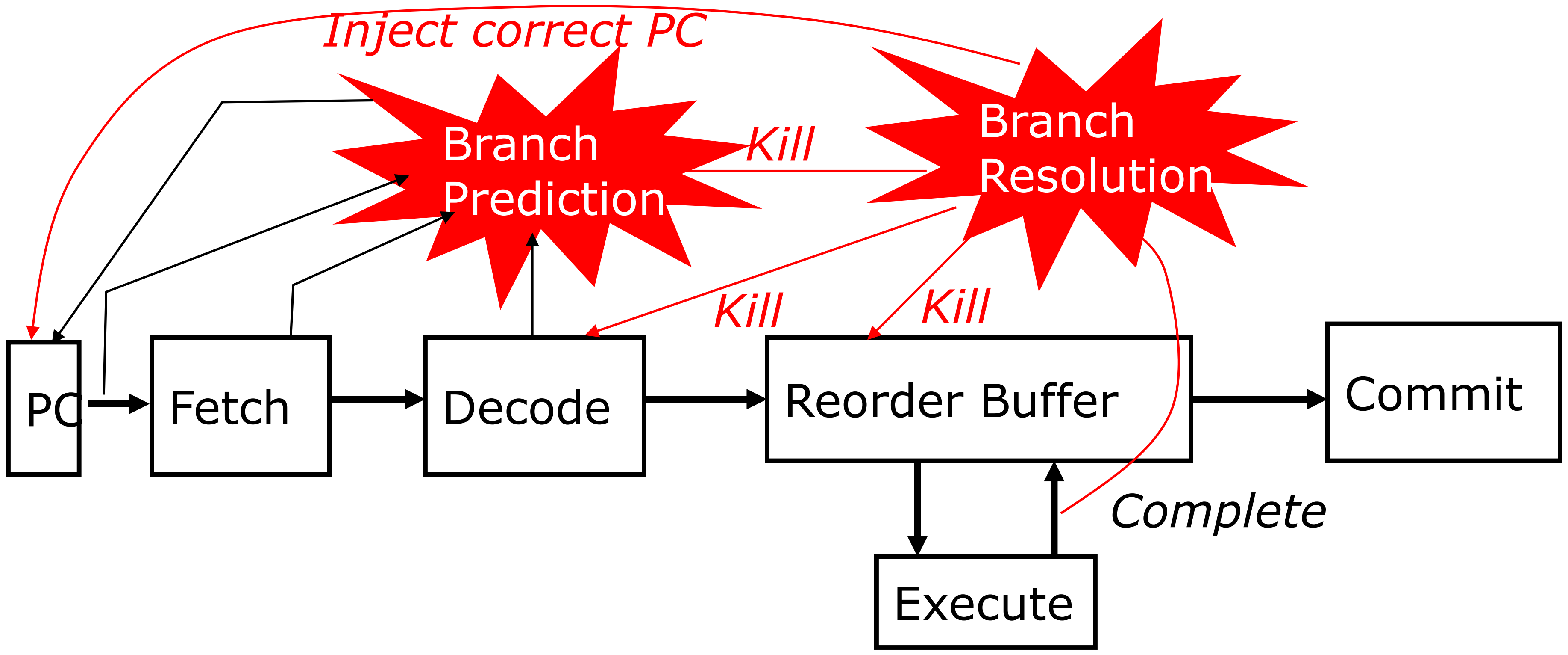
- Both styles of machine can use same branch predictors in front-end fetch pipeline, and both can execute multiple instructions per cycle
- Common to have 10-30 pipeline stages in either style of design

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

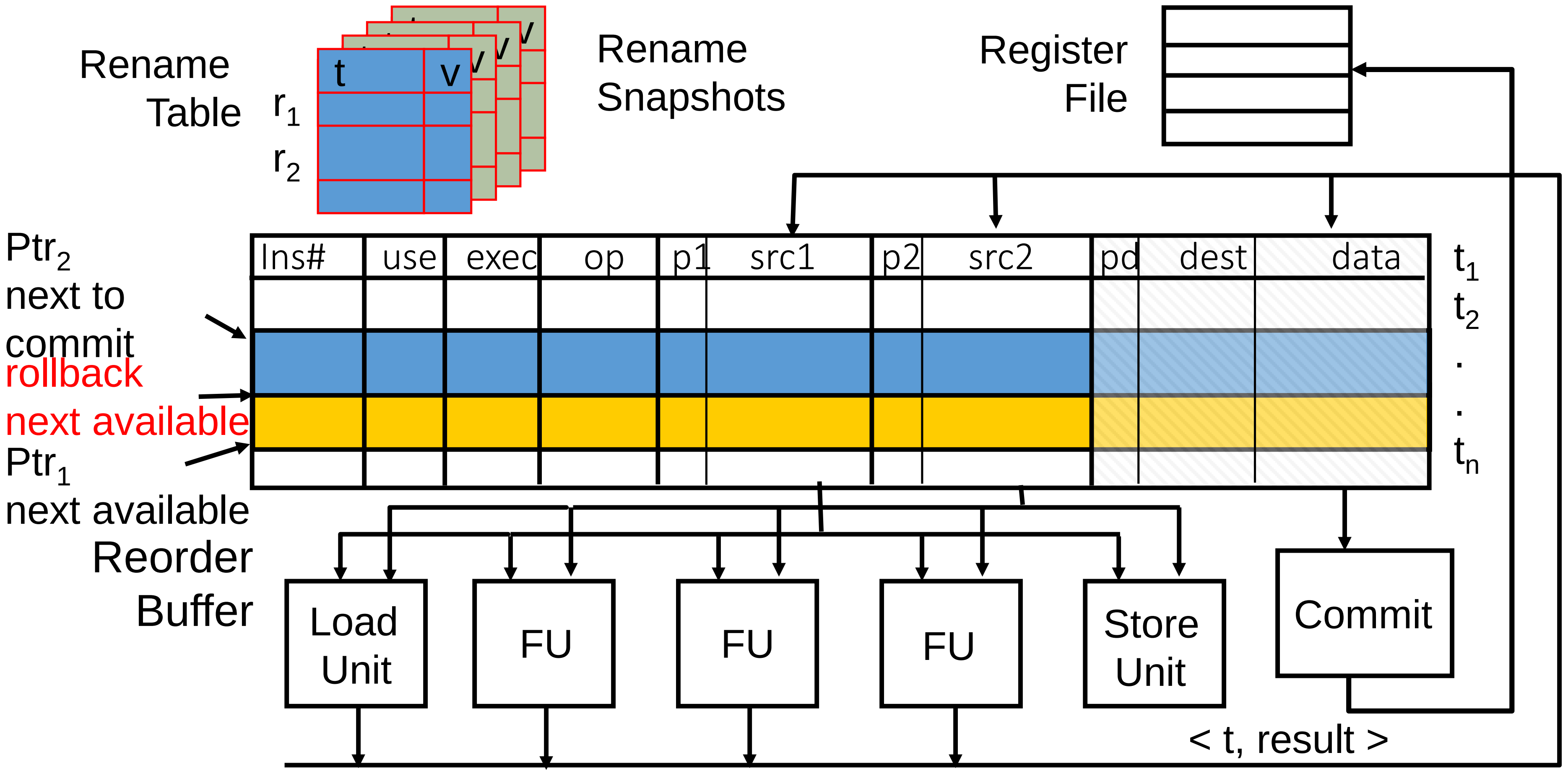
Rename Table Recovery

- 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

Mispredict Recovery



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

OoO Design Choices

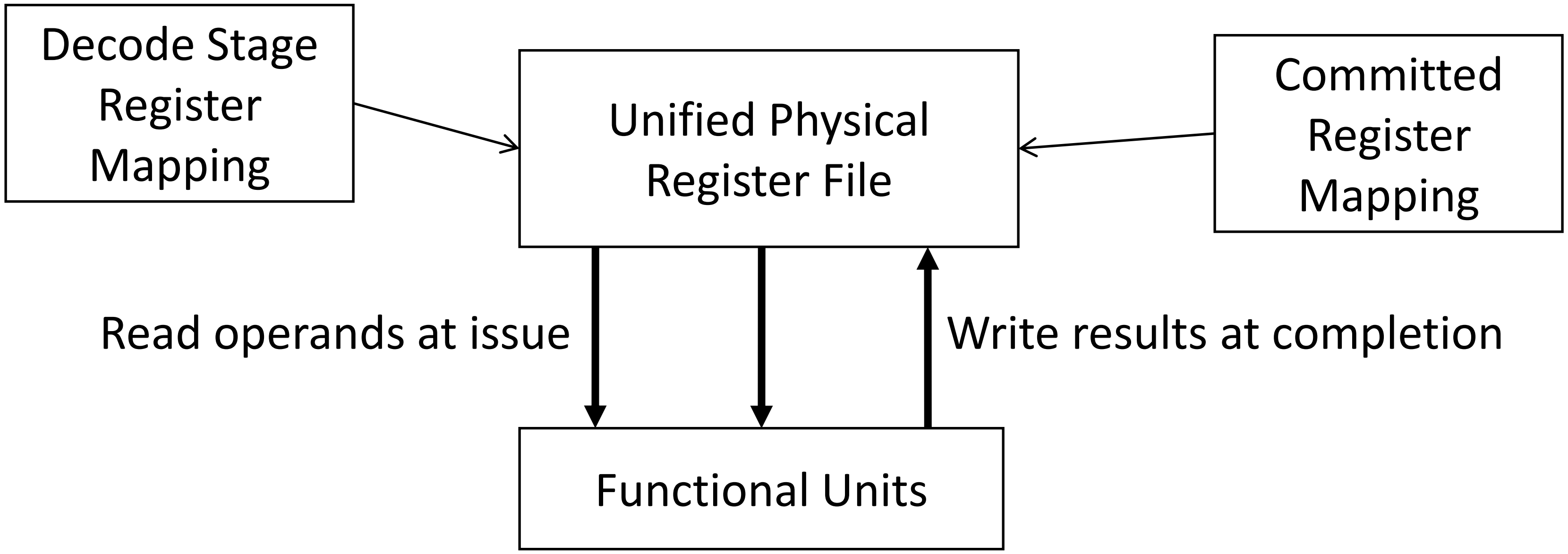
- Where are reservation stations?
  - Part of reorder buffer, or in separate issue window?
  - Distributed by functional units, or centralized?

- 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)

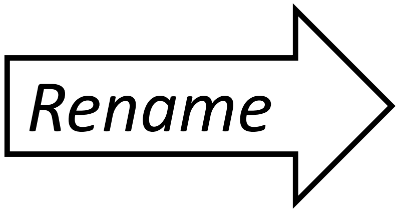
- Rename all architectural registers into a single physical register file during decode, no register values read
- Functional units read and write from single unified register file holding committed and temporary registers in execute
- Commit only updates mapping of architectural register to physical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


Decode Stage Register Mapping
Unified Physical Register File
Committed Register Mapping
Read operands at issue
Write results at completion
Functional Units

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

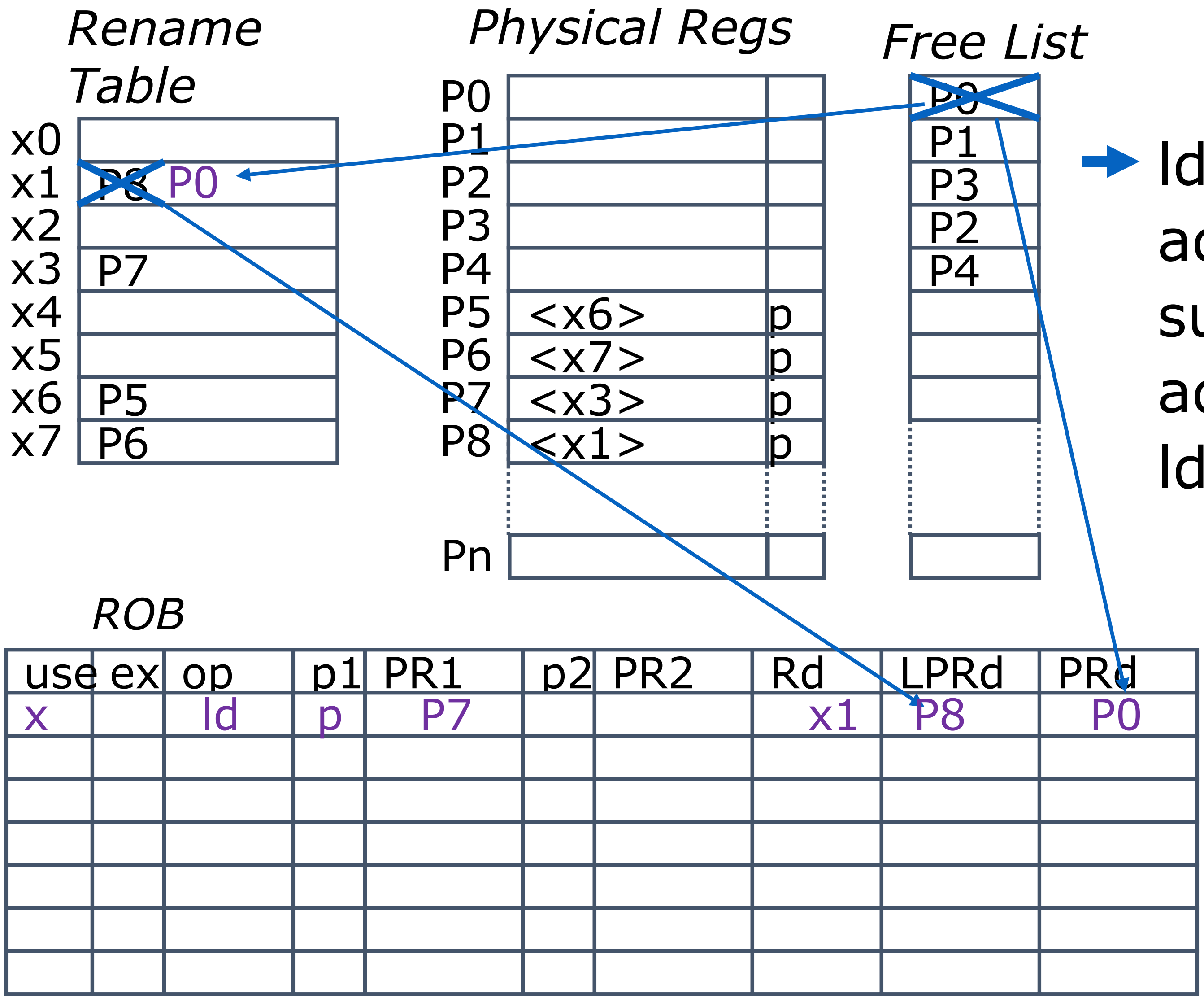
```
ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)
```

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



Rename Table

x0	
x1	~~P8~~ P0
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

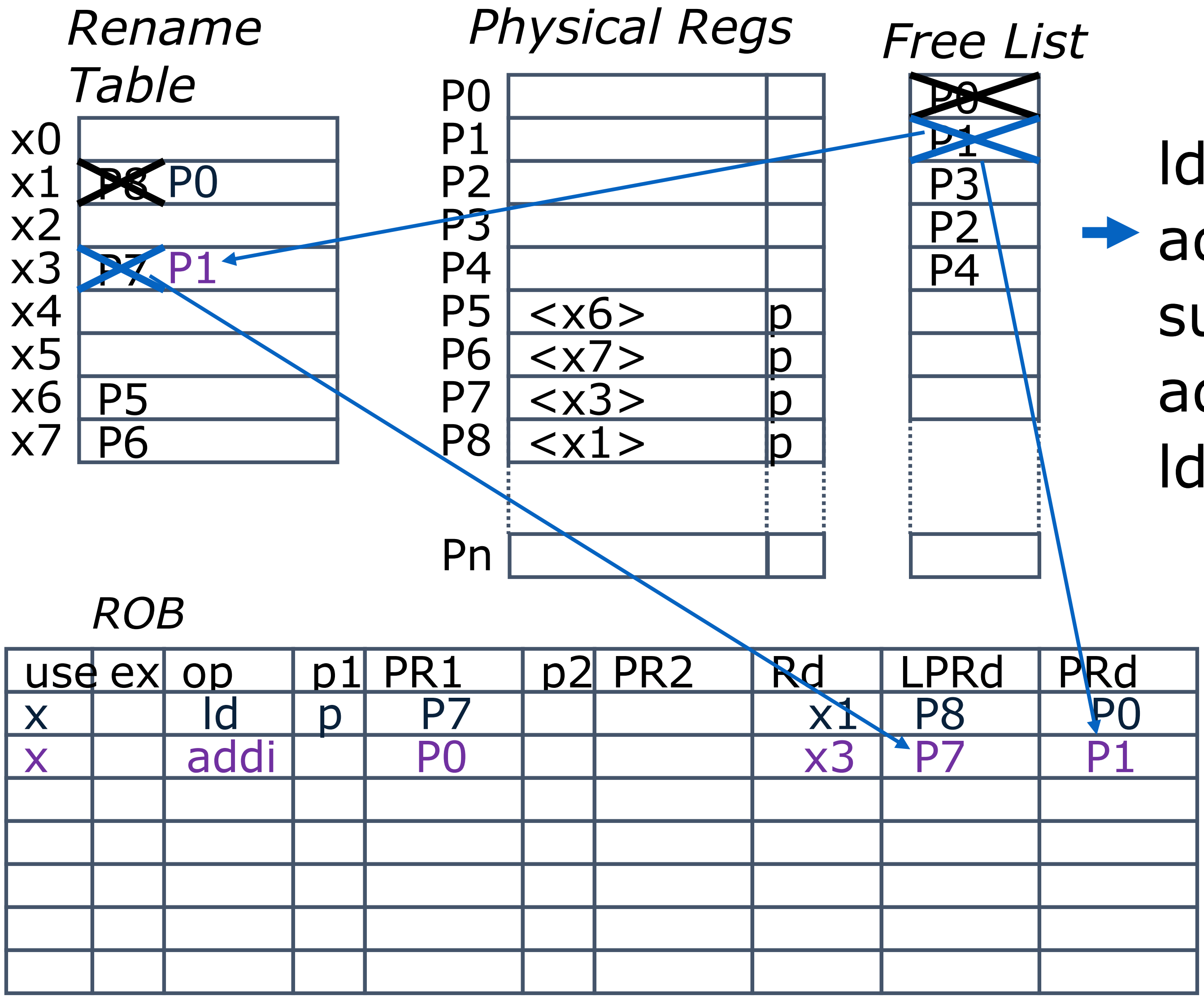
~~P0~~
P1
P3
P2
P4

ROB

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

```
➡ 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	<x1>	p
Pn		

Free List

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

ROB

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

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~~ P3
x7	P6

Physical Regs

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

Free List

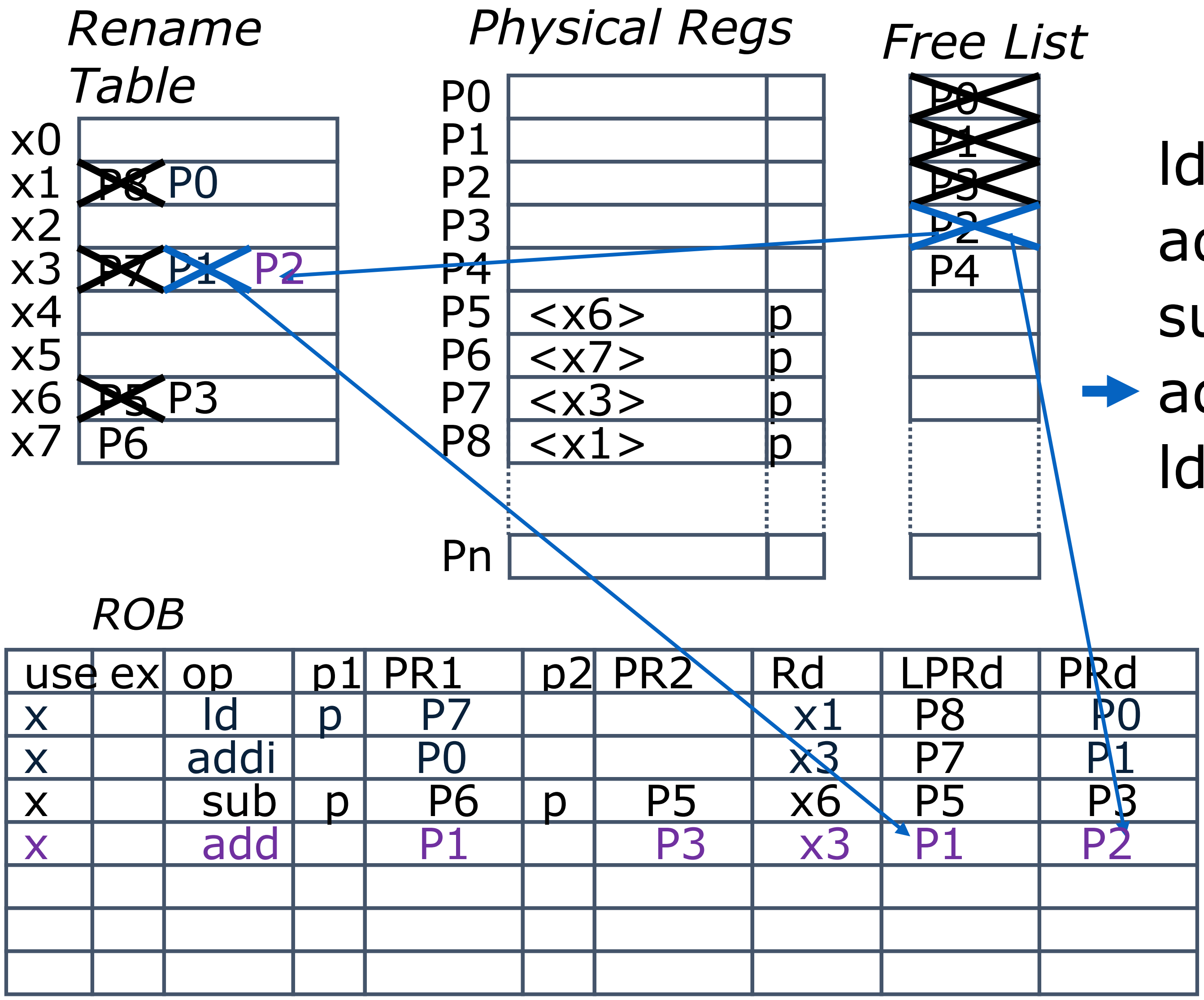
~~P0~~
~~P1~~
~~P3~~
P2
P4

```
  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
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



Rename Table

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

Physical Regs

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

Free List

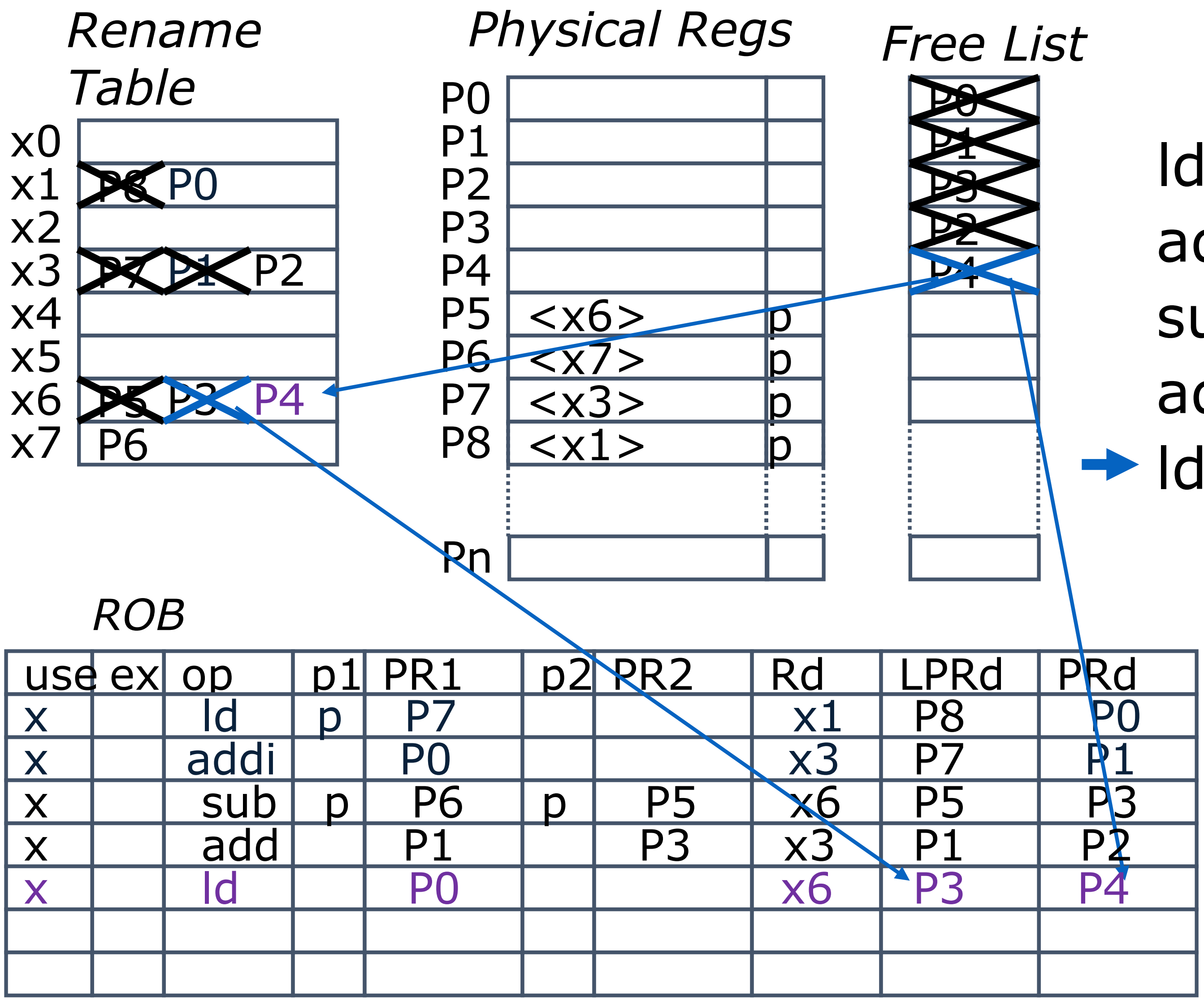
~~P0~~
~~P1~~
~~P3~~
~~P2~~
P4

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



Rename Table

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

Physical Regs

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

Free List

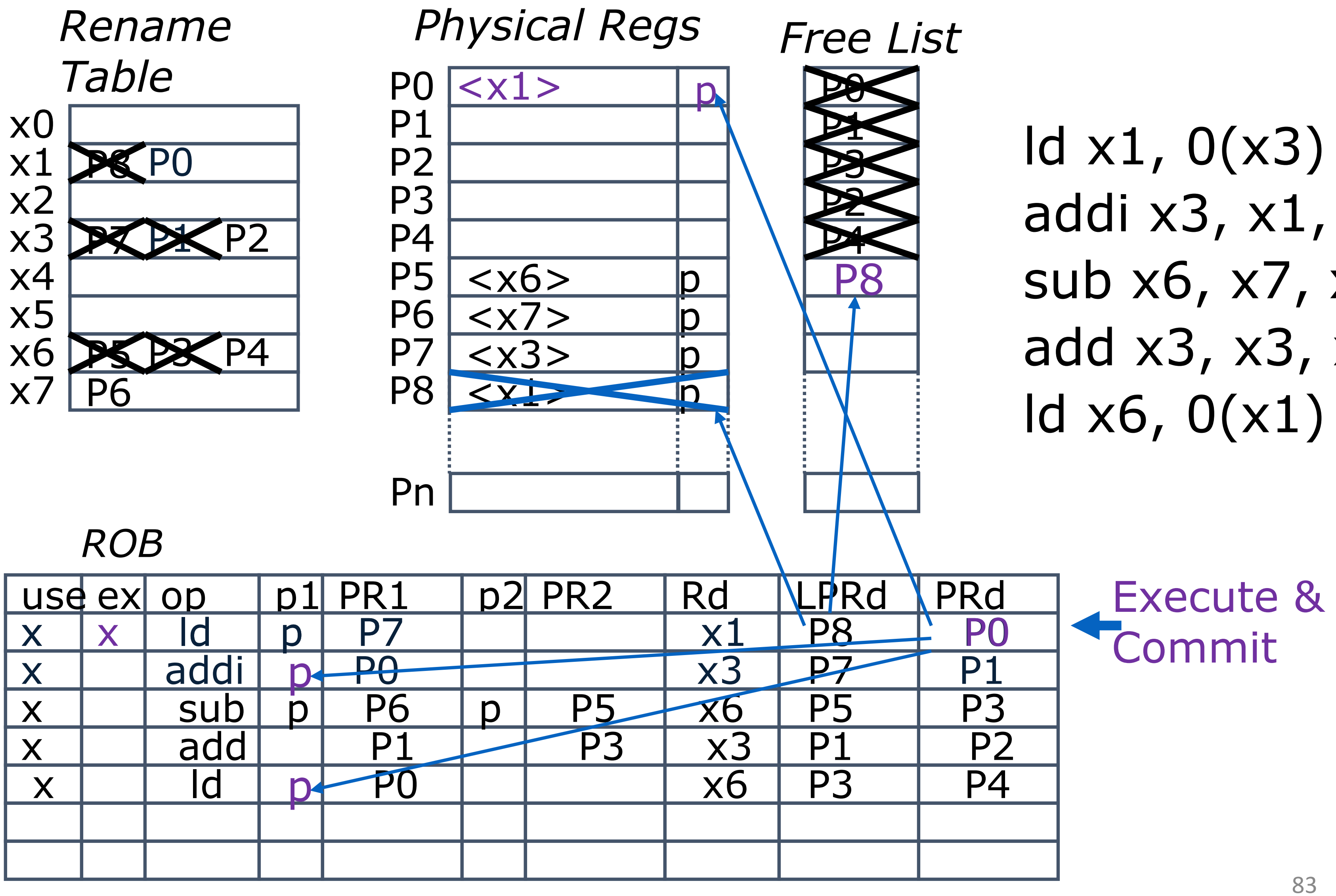
~~P0~~
~~P1~~
~~P3~~
~~P2~~
~~P4~~

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

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~~ 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

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

Free List

~~P0~~
~~P1~~
~~P3~~
~~P2~~
~~P4~~
P8
P7

```
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

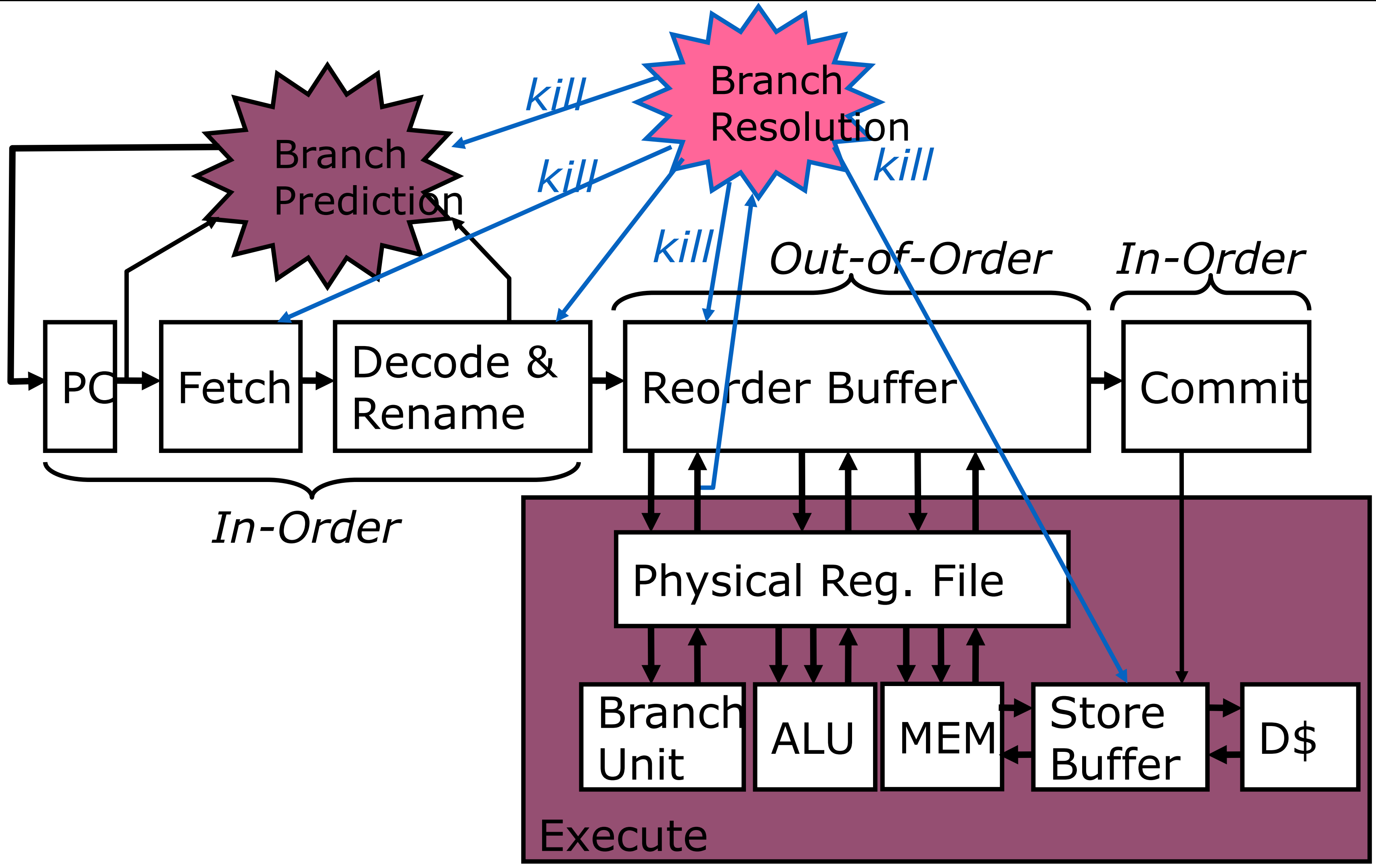
Execute & Commit

Repairing Rename at Traps

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

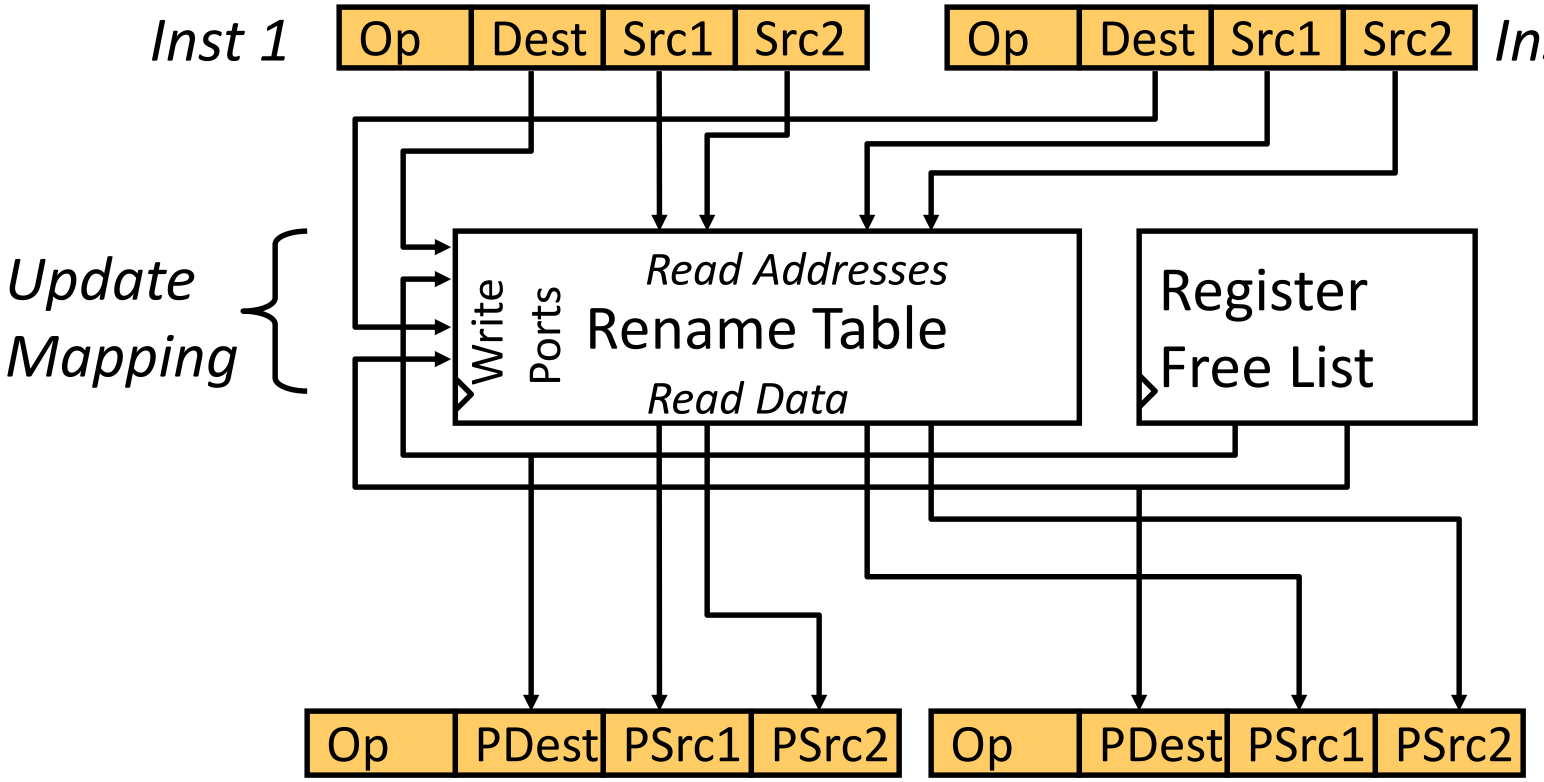
- Alpha 21264 had similar physical register file scheme, but kept complete rename table snapshots for each instruction in ROB (80 snapshots total)
  - Flash copy all bits from snapshot to active table in one cycle

Pipeline Design with Physical Regfile



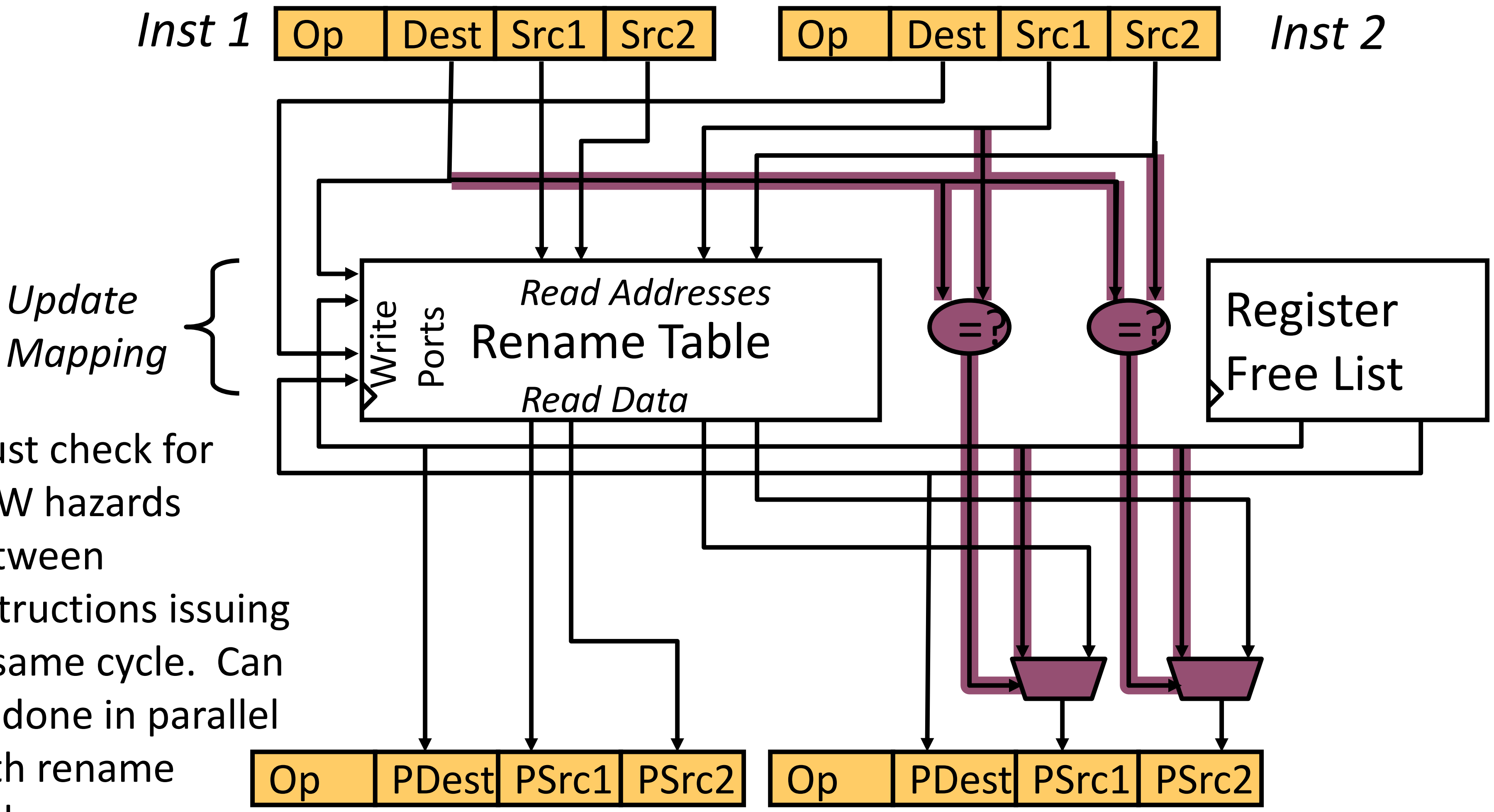
Superscalar Register Renaming

- 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.

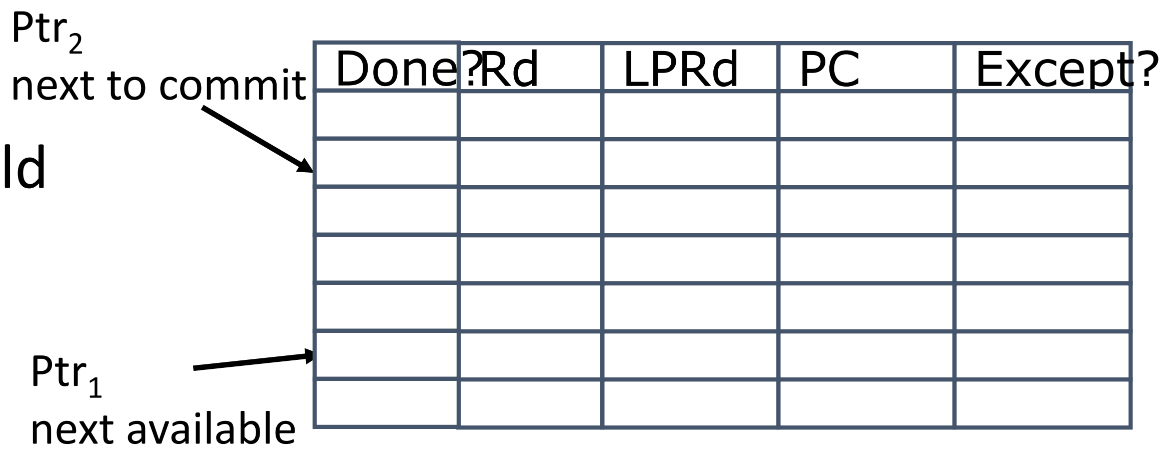
MIPS R10K renames 4 serially-RAW-dependent insts/cycle

Separate Issue Window from ROB

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

Instructions that committed are not in the instruction window any more

us	ex	op	p1	PR1	p2	PR2	PRd	ROB#



Reorder buffer used to hold exception information for commit.

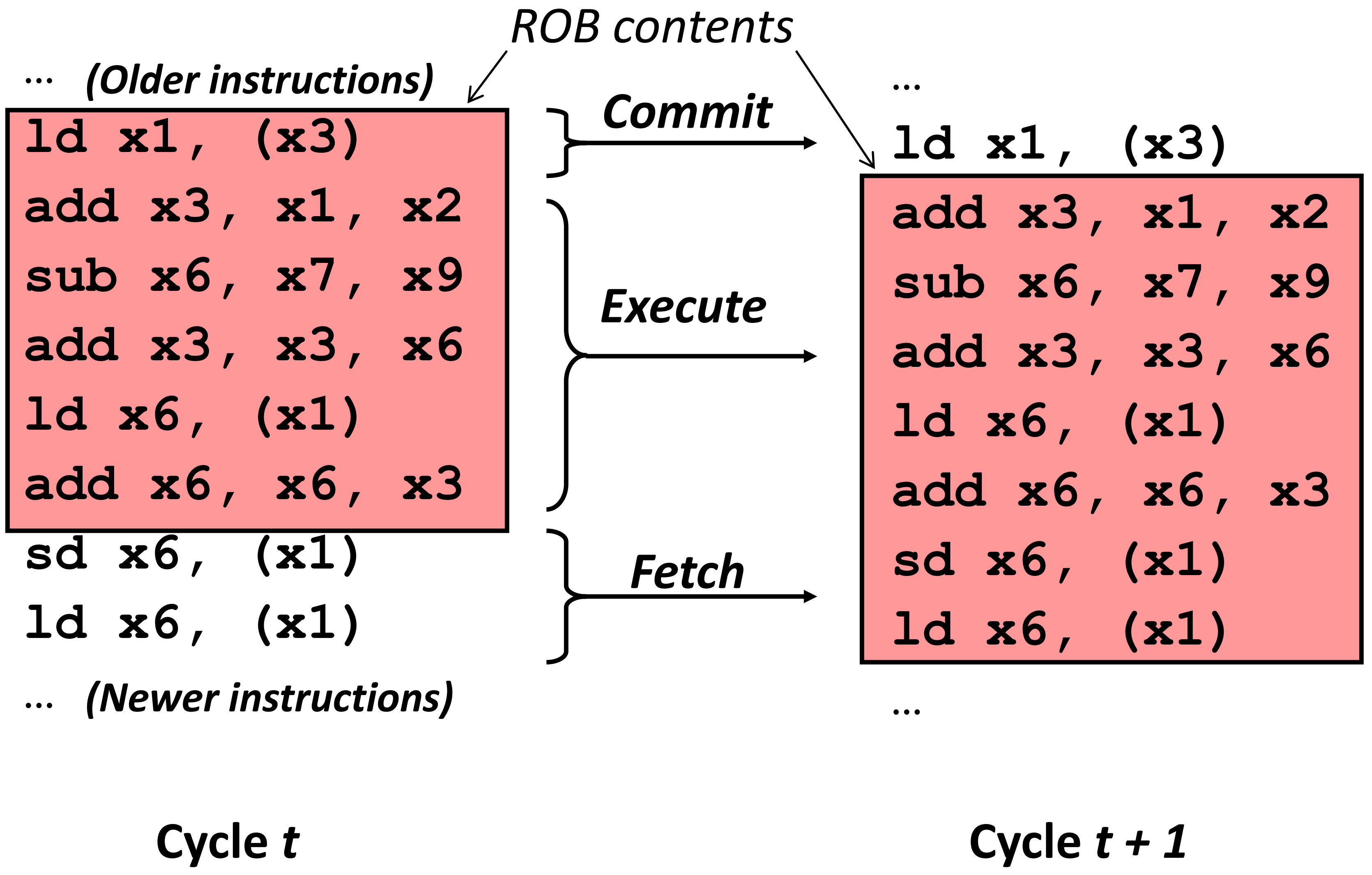
ROB is usually several times larger than instruction window – why?

Separating Completion from Commit

- Re-order buffer holds register results from completion until commit
  - Entries allocated in program order during decode
  - Buffers completed values and exception state until in-order commit point
  - Completed values can be used by dependents before committed (bypassing)
  - Each entry holds program counter, instruction type, destination register specifier and value if any, and exception status (info often compressed to save hardware)

- Memory reordering needs special data structures
  - Speculative store address and data buffers
  - Speculative load address and data buffers

Reorder Buffer Holds Active Instructions (Decoded but not Committed)



Issue Timing

i1	Add R1,R1,#1	Issue_1	Execute_1		
i2	Sub R1,R1,#1			Issue_2	Execute_2

How can we issue earlier?

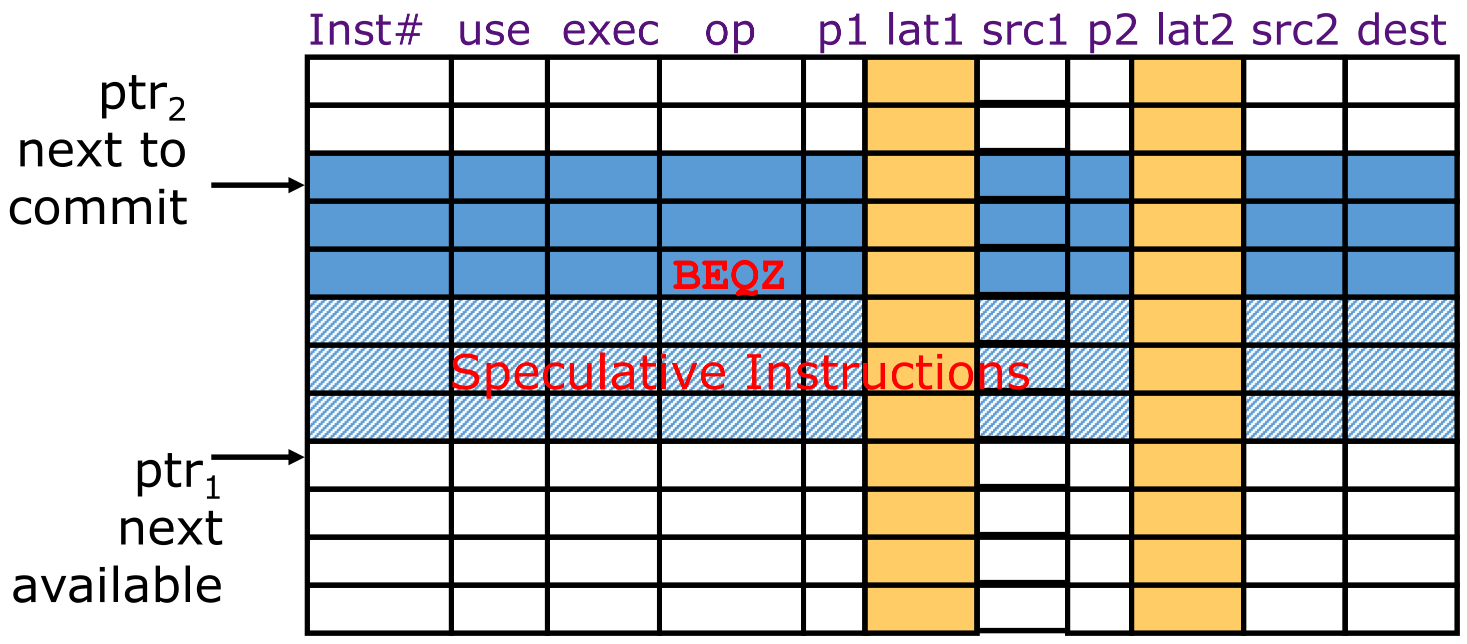
Using knowledge of execution latency (bypass)

i1	Add R1,R1,#1	Issue_1	Execute_1		
i2	Sub R1,R1,#1		Issue_2	Execute_2	

What makes this schedule fail?

If execution latency wasn't as expected

Issue Queue with latency prediction



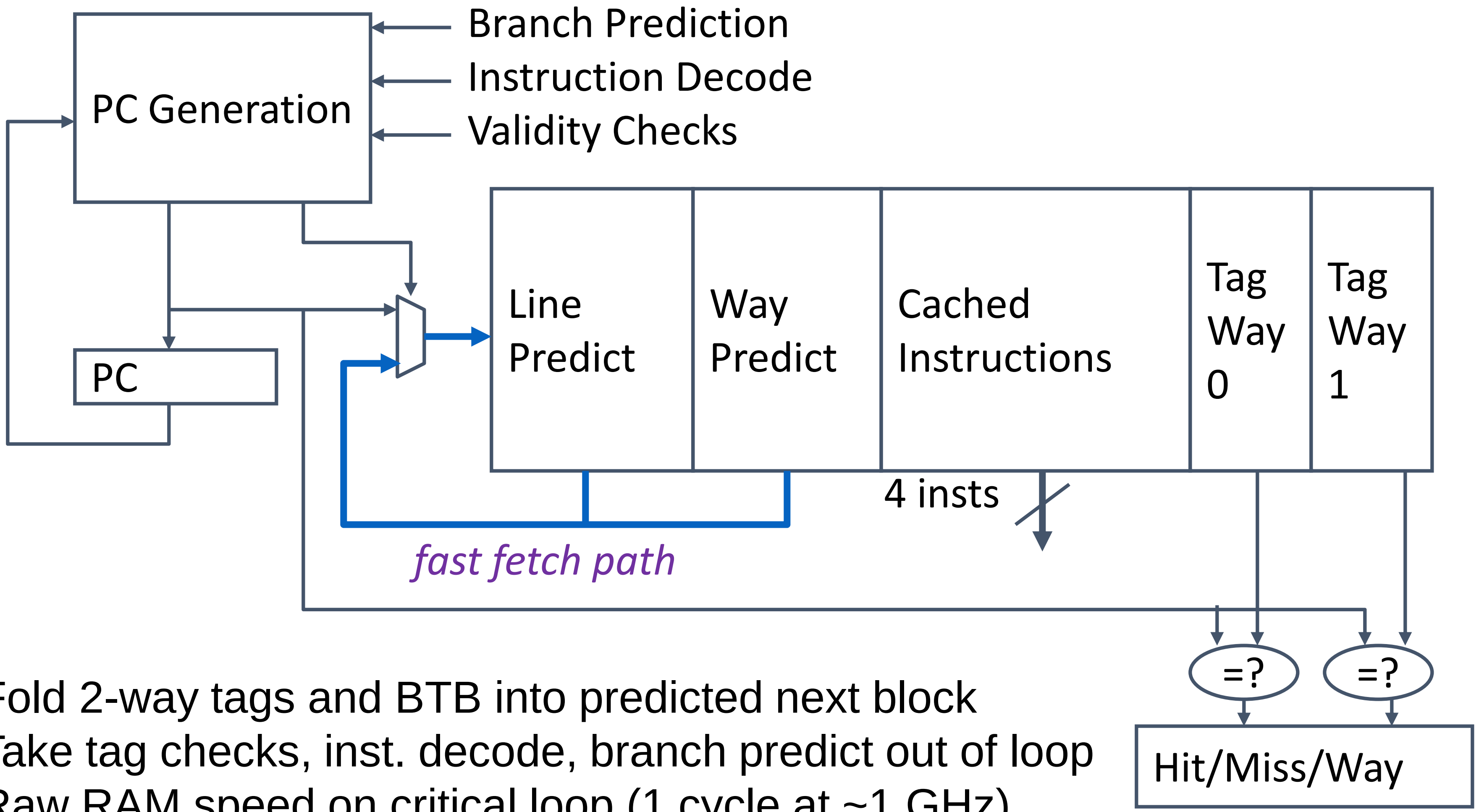
Issue Queue (Reorder buffer)

- Fixed latency: latency included in queue entry (‘bypassed’)
- Predicted latency: latency included in queue entry (speculated)
- Variable latency: wait for completion signal (stall)

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

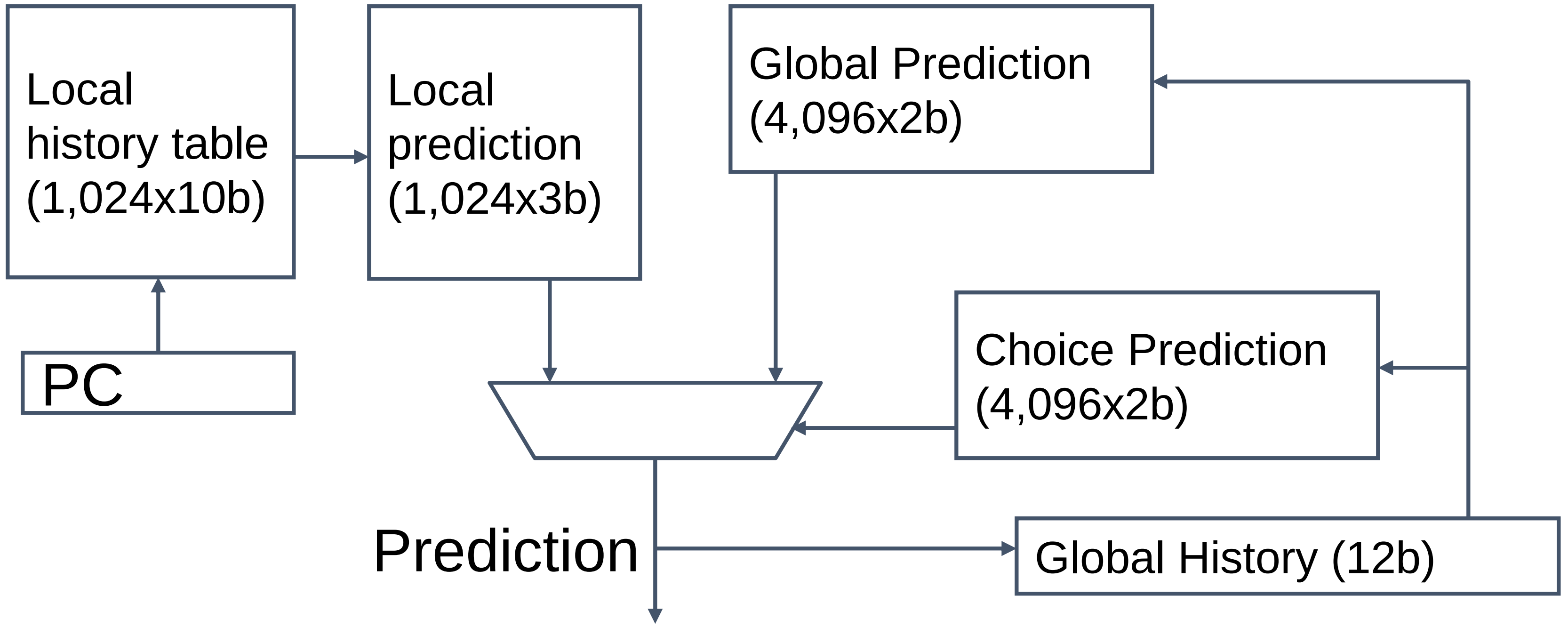
Increasing Taken Branch Bandwidth (Alpha 21264 I-Cache)



- 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


Local history table (1,024x10b)
Local prediction (1,024x3b)
Global Prediction (4,096x2b)
Choice Prediction (4,096x2b)
PC
Prediction
Global History (12b)

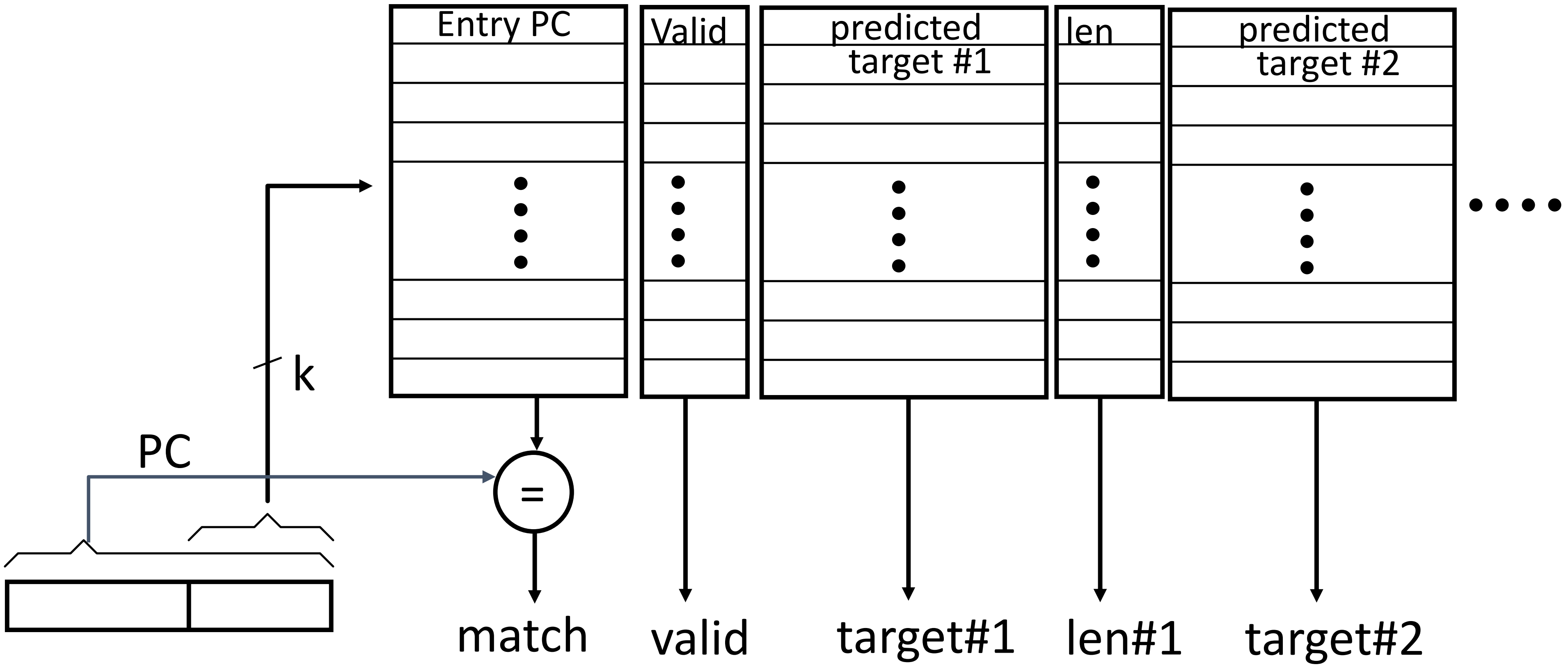
Taken Branch Limit

- 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

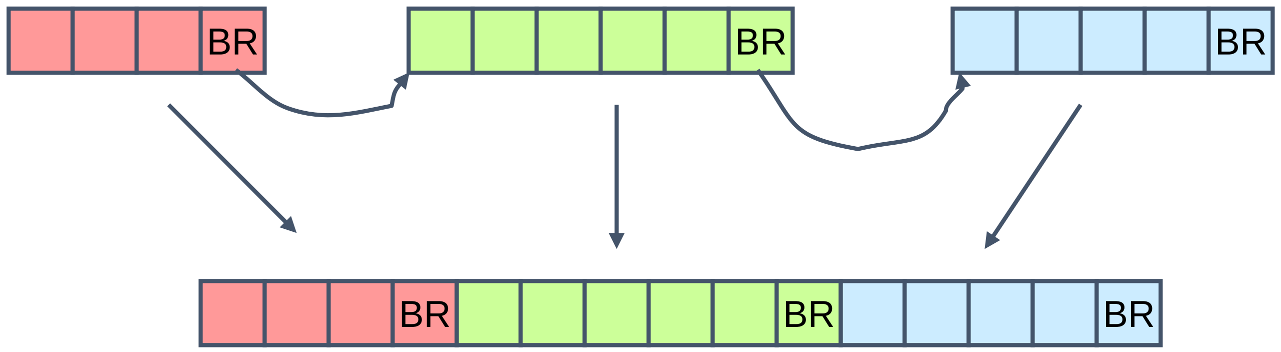
Fetching Multiple Basic Blocks

- 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



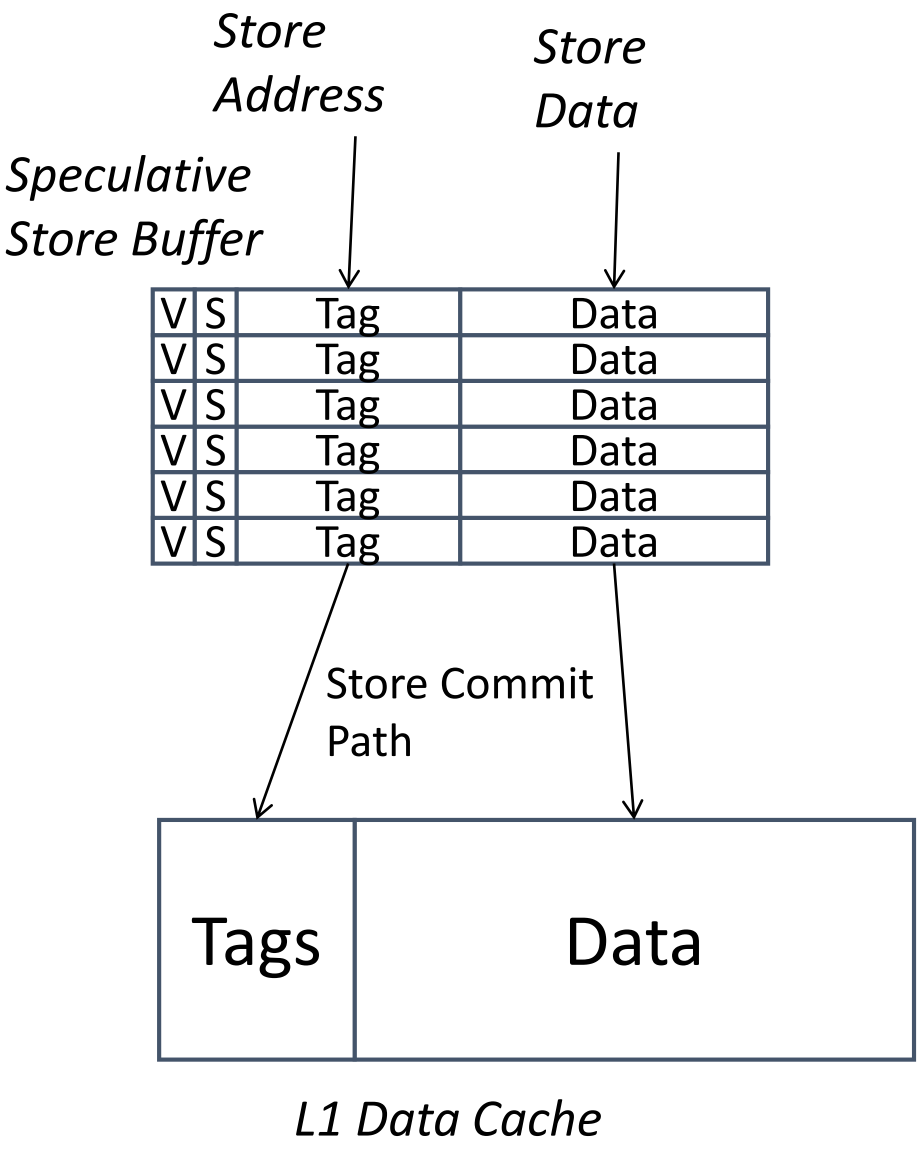
- Single fetch brings in multiple basic blocks
- Trace cache indexed by start address and next n branch predictions
- Used in Intel Pentium-4 processor to hold decoded uops

Load-Store Queue Design

- After control hazards, data hazards through memory are probably next most important bottleneck to superscalar performance

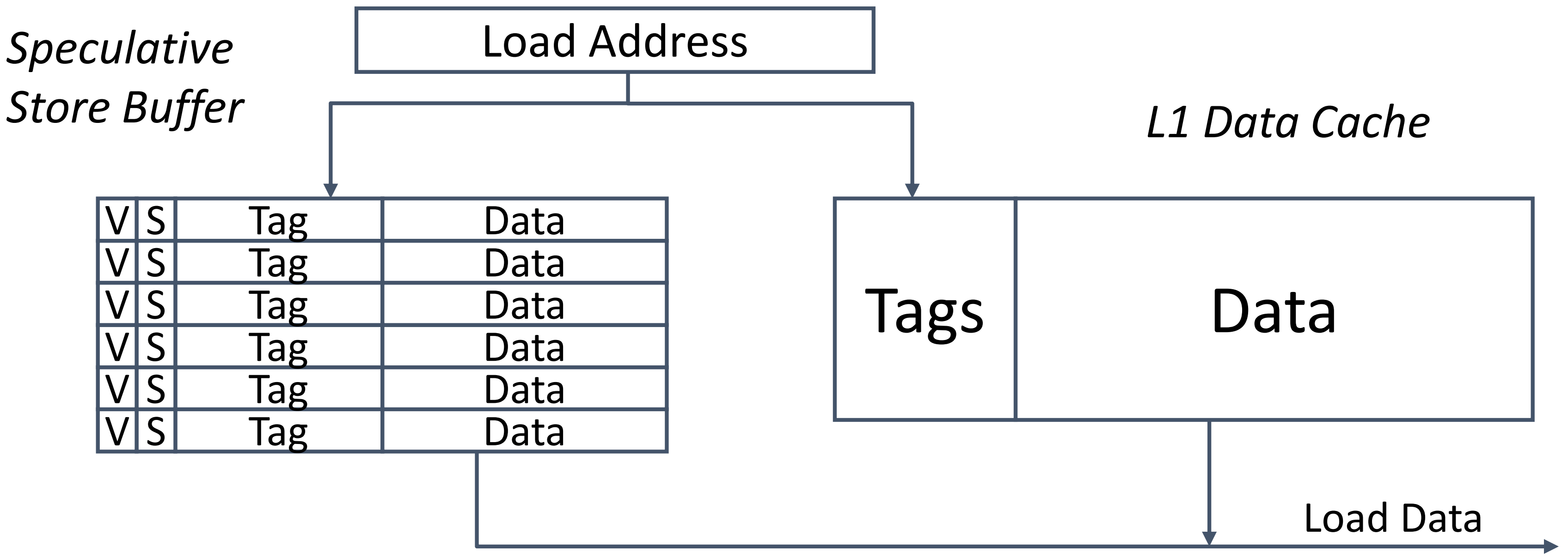
- Modern superscalars use very sophisticated load-store reordering techniques to reduce effective memory latency by allowing loads to be speculatively issued

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

Instruction Flow in Unified Physical Register File Pipeline

- Fetch
  - Get instruction bits from current guess at PC, place in fetch buffer
  - Update PC using sequential address or branch predictor (BTB)
- Decode/Rename
  - Take instruction from fetch buffer
  - Allocate resources to execute instruction:
    - Destination physical register, if instruction writes a register
    - Entry in reorder buffer to provide in-order commit
    - Entry in issue window to wait for execution
    - Entry in memory buffer, if load or store
  - Decode will stall if resources not available
  - Rename source and destination registers
  - Check source registers for readiness
  - Insert instruction into issue window+reorder buffer+memory buffer

Memory Instructions

- Split store instruction into two pieces during decode:
  - Address calculation, store-address
  - Data movement, store-data
- Allocate space in program order in memory buffers during decode
  - Store instructions:
    - Store-address calculates address and places in store buffer
    - Store-data copies store value into store buffer
    - Store-address and store-data execute independently out of issue window
    - Stores only commit to data cache at commit point
- Load instructions:
  - Load address calculation executes from window
  - Load with completed effective address searches memory buffer
  - Load instruction may have to wait in memory buffer for earlier store ops to resolve

Issue Stage

- Writebacks from completion phase “wakeup” some instructions by causing their source operands to become ready in issue window
  - In more speculative machines, might wake up waiting loads in memory buffer

- Need to “select” some instructions for issue
  - Arbiter picks a subset of ready instructions for execution
  - Example policies: random, lower-first, oldest-first, critical-first

- Instructions read out from issue window and sent to execution

Execute Stage

- Read operands from physical register file and/or bypass network from other functional units
- Execute on functional unit
- Write result value to physical register file (or store buffer if store)
- Produce exception status, write to reorder buffer
- Free slot in instruction window

Commit Stage

- Read completed instructions in-order from reorder buffer
  - (may need to wait for next oldest instruction to complete)
- If exception raised
  - flush pipeline, jump to exception handler
- Otherwise, release resources:
  - Free physical register used by last writer to same architectural register
  - Free reorder buffer slot
  - Free memory reorder buffer slot

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

Open-source Berkeley RISC-V Processors

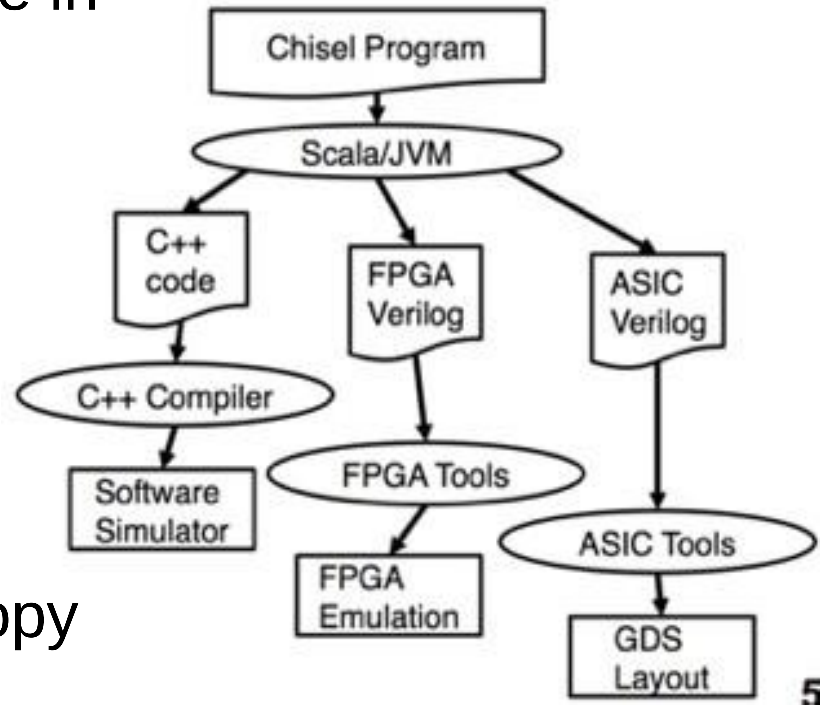
- Sodor Collection
  - RV32I - Entry, educational, not synthesizable
- Rocket-chip SoC generator
- Z-scale
  - RV32IM - micro-controller
- Rocket
  - RV64G - in-order, single-issue application core
- BOOM
  - RV64G - out-of-order, superscalar application core

What is BOOM?

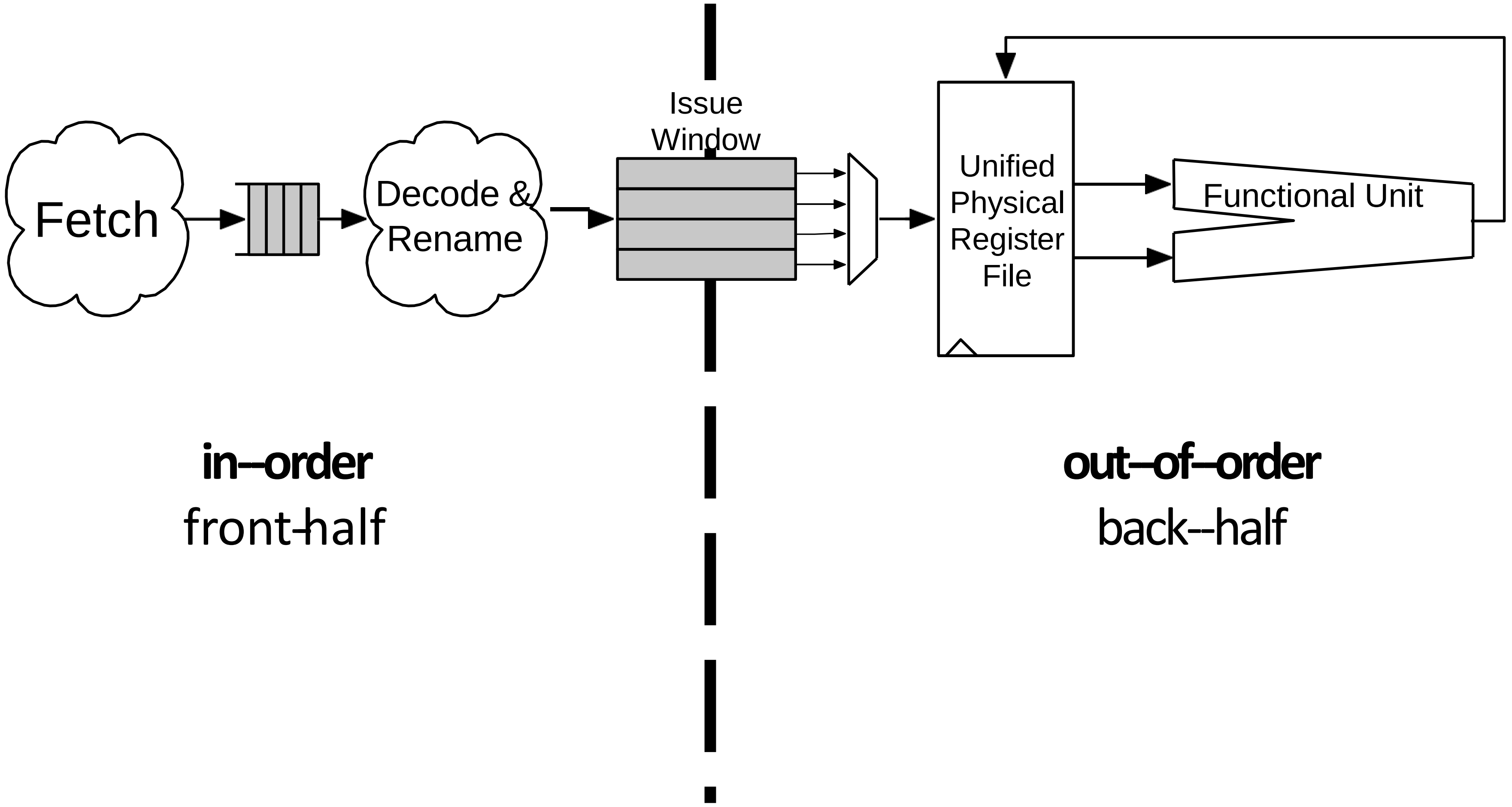
- superscalar, out-of-order processor written in Berkeley’s Chisel RTL
- It is synthesizable
- It is parameterizable
- We hope to use it as a plaOorm for architecture research
- It is open-source!
  - BOOM is a work-in-progress

Chisel

- Hardware Construction Language embedded in Scala
- ***not*** a high-level synthesis language hardware module is a data structure in Scala
- Full power of Scala for writing generators
  - object-oriented programming
  - factory objects, traits, overloading
  - functional programming
  - high-order funs, anonymous funcs, currying
- generated C++ simulator is 1:1 copy of Verilog designs
- version 3.0 coming soon!
  - uses an IR called FIRRTL



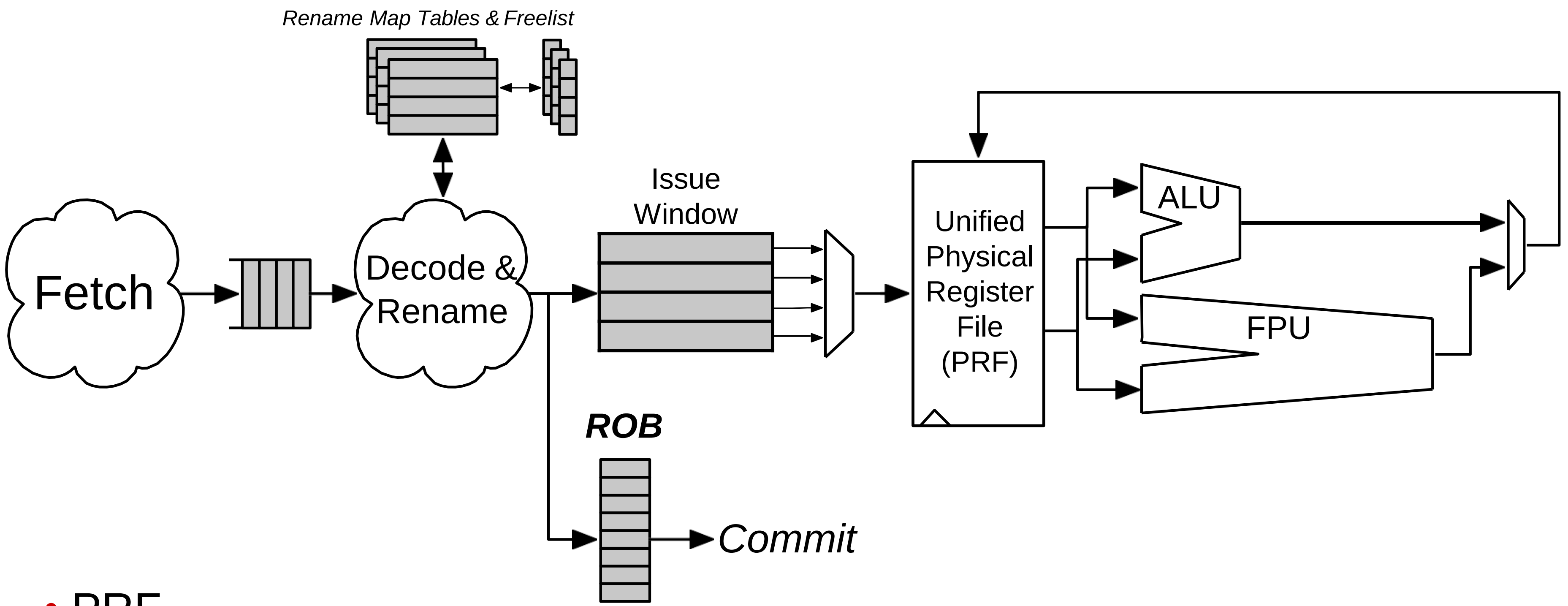
BOOM Pipeline



in--order
front-half

out--of--order
back--half

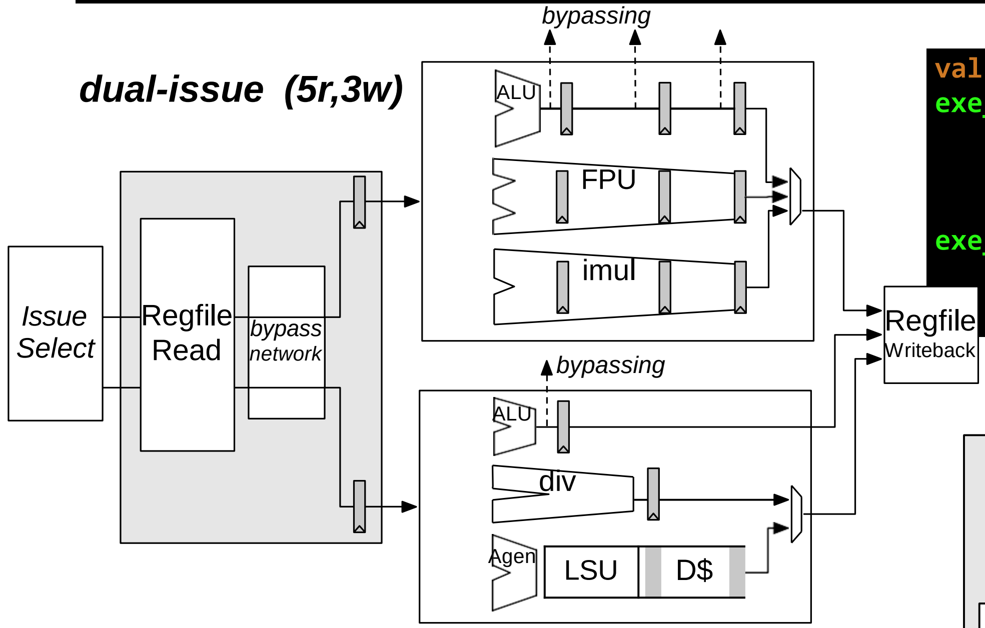
BOOM Pipeline


Rename Map Tables & Freelist
Fetch
Decode & Rename
Issue Window
ROB
Commit
Unified Physical Register File (PRF)
ALU
FPU

- PRF
  - explicit renaming
  - holds speculative and commi6ed data
  - holds both x-regs, f-regs
- split ROB/issue window design
- Unified Issue Window
  - holds all instructions

Parameterized Superscalar

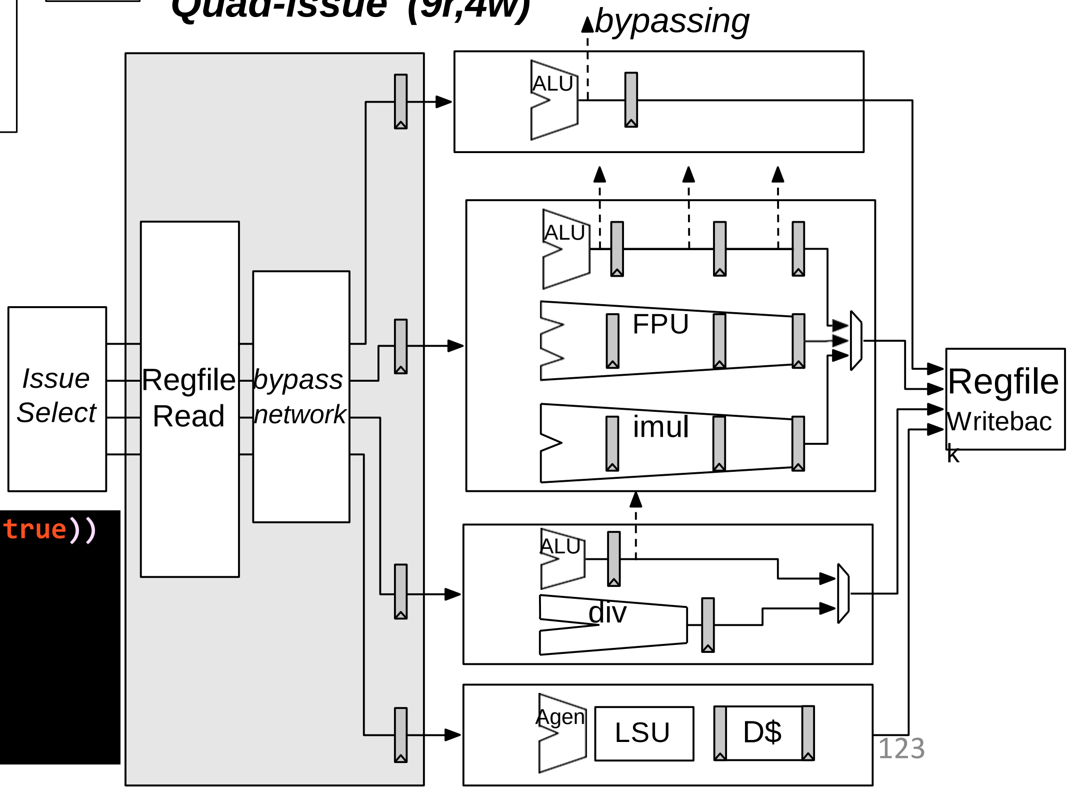
dual-issue (5r,3w)



```
val exe_units = ArrayBuffer[ExecutionUnit]()
exe_units += Module(new ALUExeUnit(is_branch_unit     = true
                                   , has_fpu          = true
                                   , has_mul          = true
                                   ))
exe_units += Module(new ALUMemExeUnit(fp_mem_support = true
                                   , has_div          = true
                                   ))
```

OR

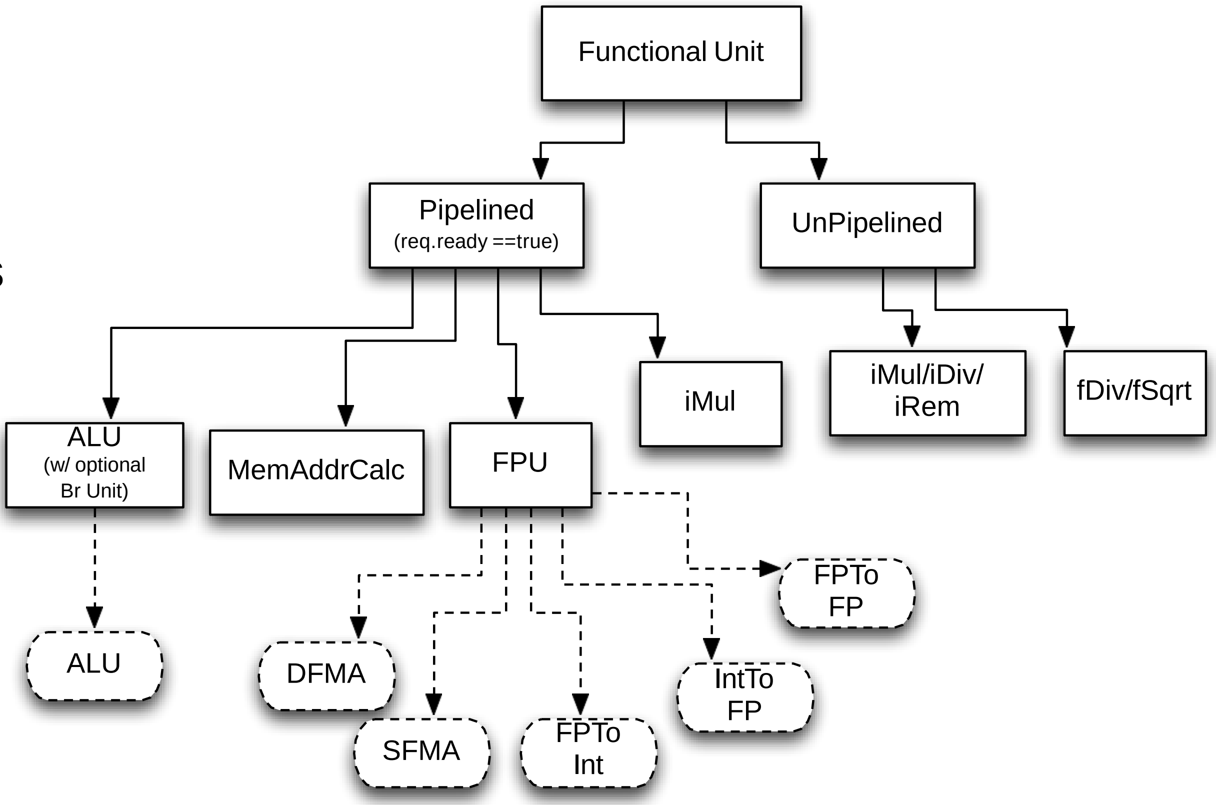
Quad-issue (9r,4w)



```
exe_units += Module(new ALUExeUnit(is_branch_unit = true))
exe_units += Module(new ALUExeUnit(has_fpu = true
                                  , has_mul = true
                                  ))
exe_units += Module(new ALUExeUnit(has_div = true))
exe_units += Module(new MemExeUnit())
```

Open--source Berkeley RISC-V Processors

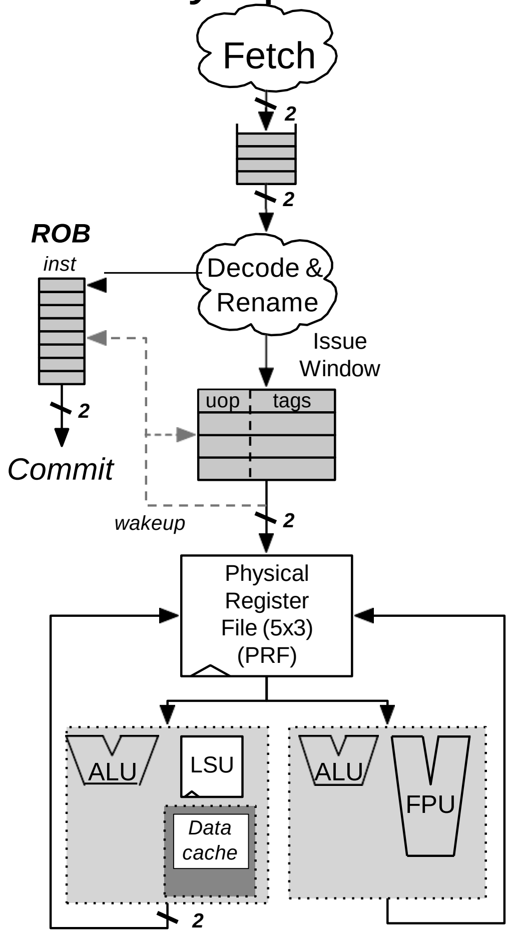
- Abstract FunctionalUnit
  - describes common IO
- Pipelined/Unpipelined
  - handles storing uop metadata, branch resolution, branch kills
- Concrete Subclasses
  - instantiates the actual expert-written FU
  - no modifications required to get FU working with speculative OoO
  - allows easy "stealing" of external code



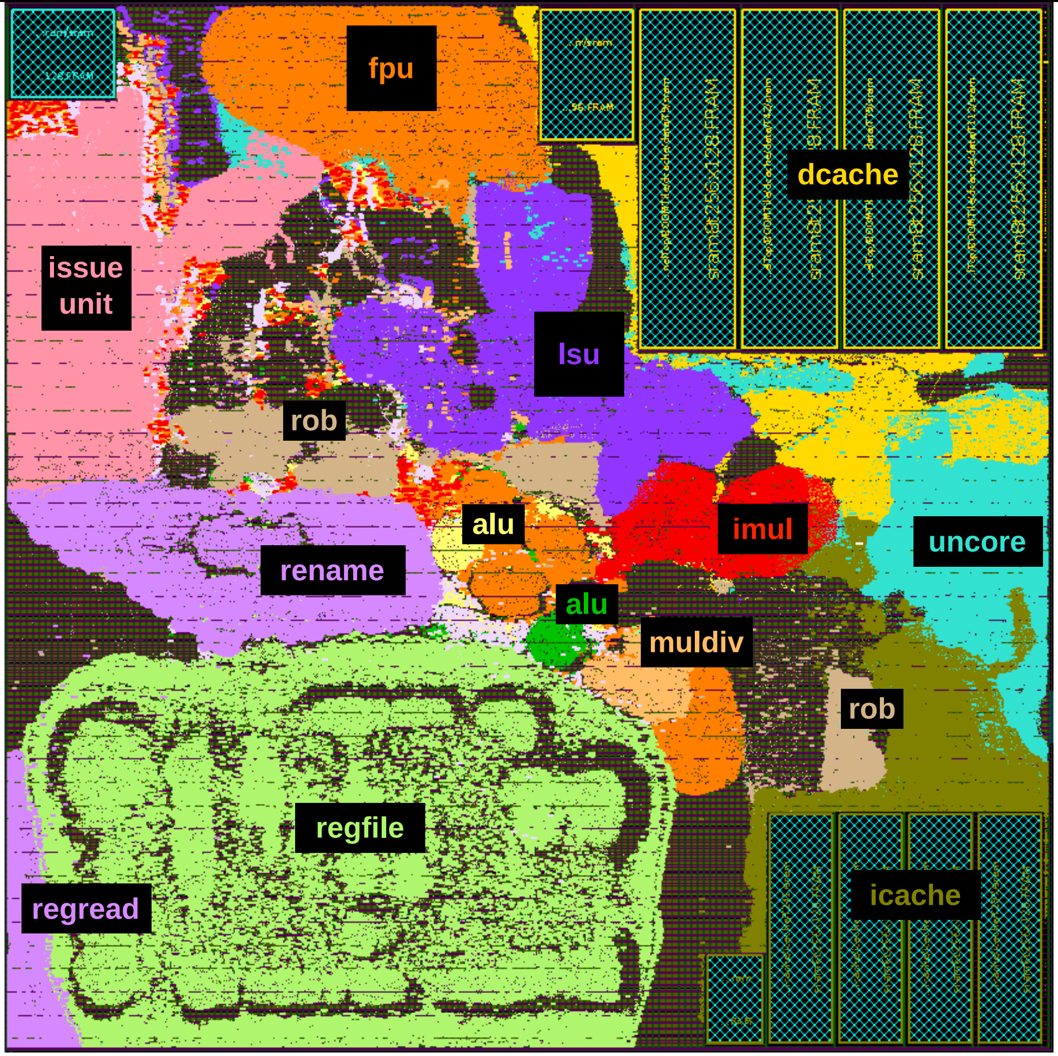
Synthesizable

preliminary results

- Targeting ASIC
- Runs on FPGA
  - Zynq zc706



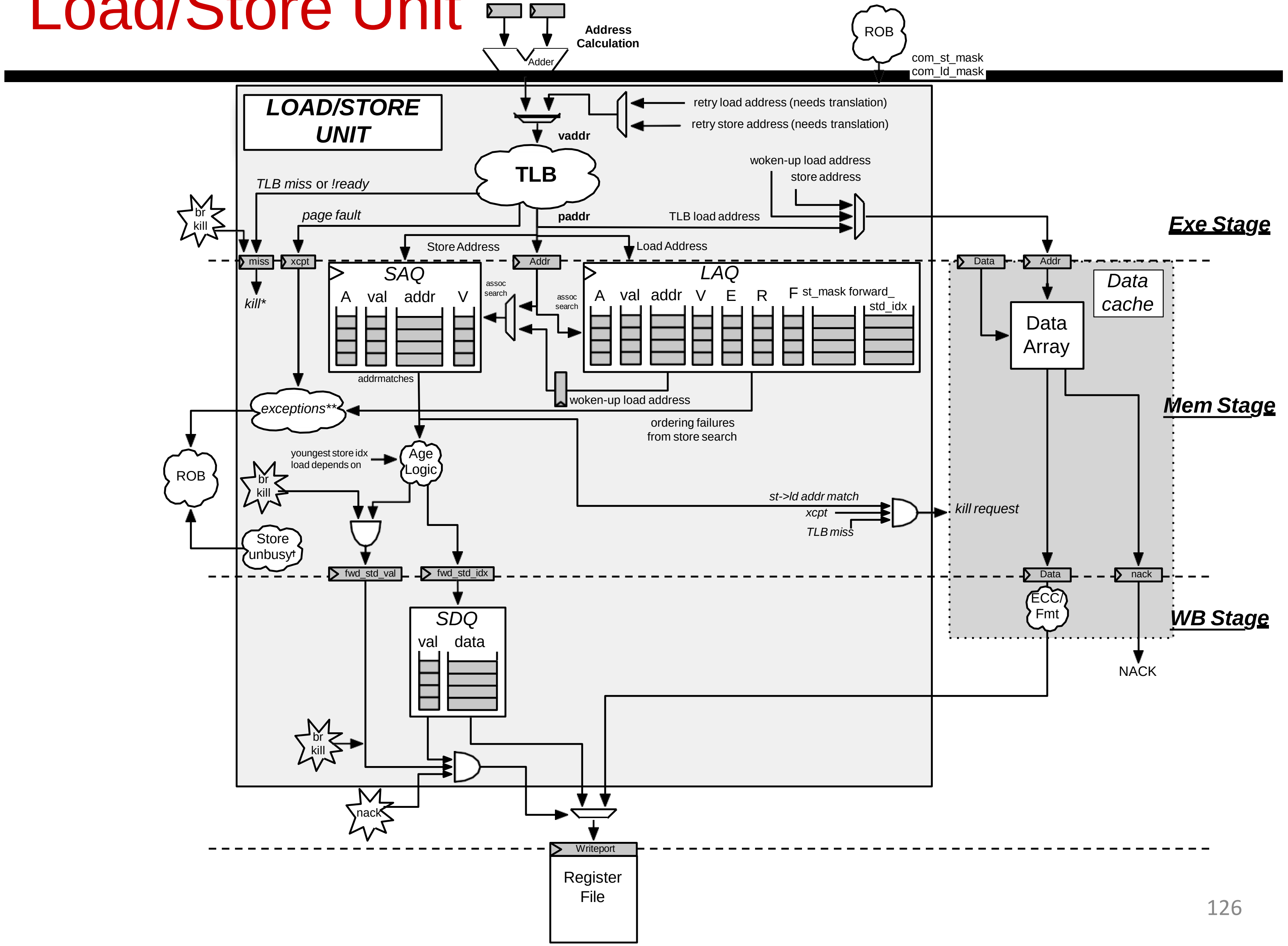
TSMC 45nm floorplan



2-wide BOOM, 16kB L1 caches

1.2 mm^2

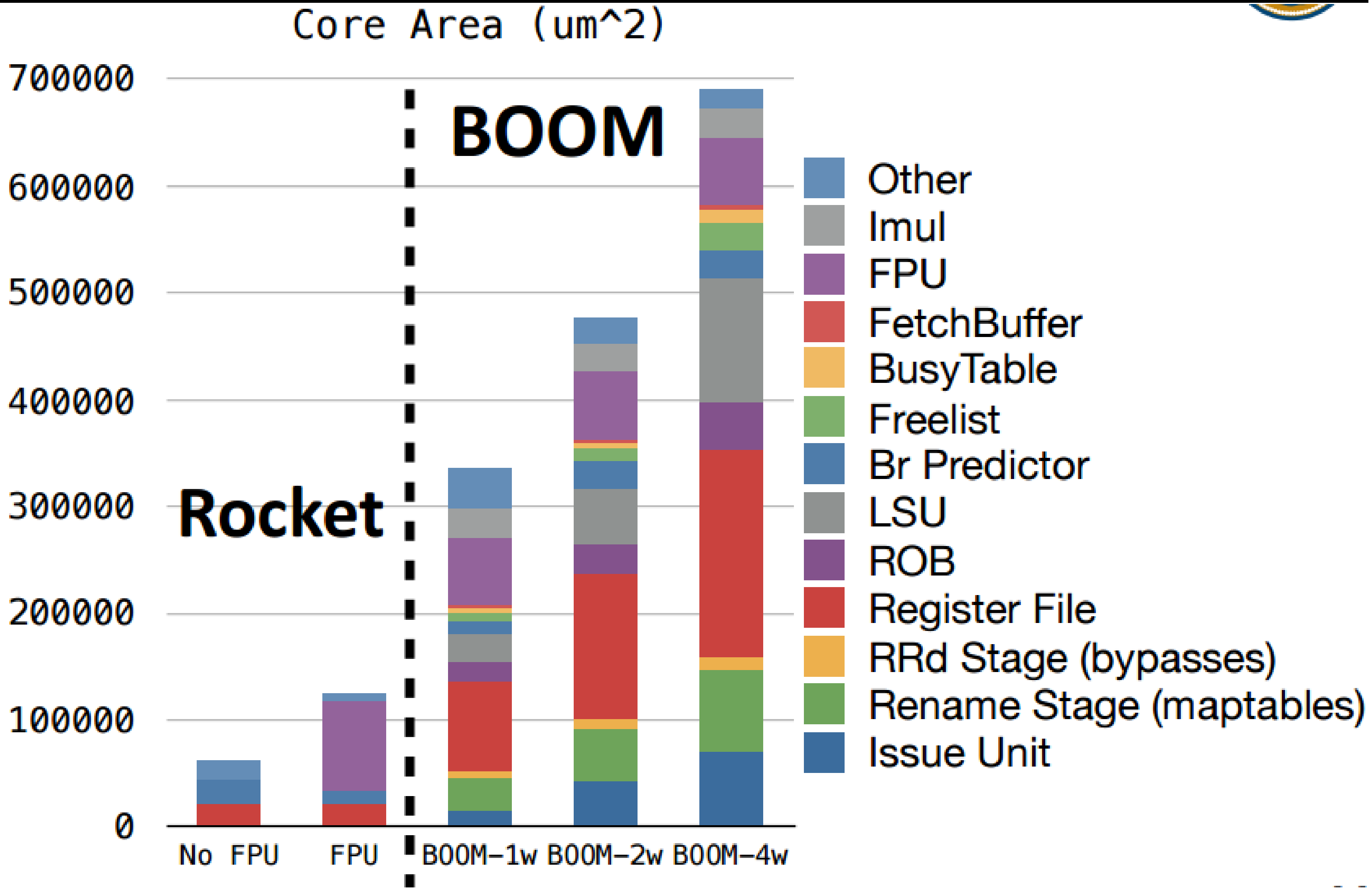
Load/Store Unit



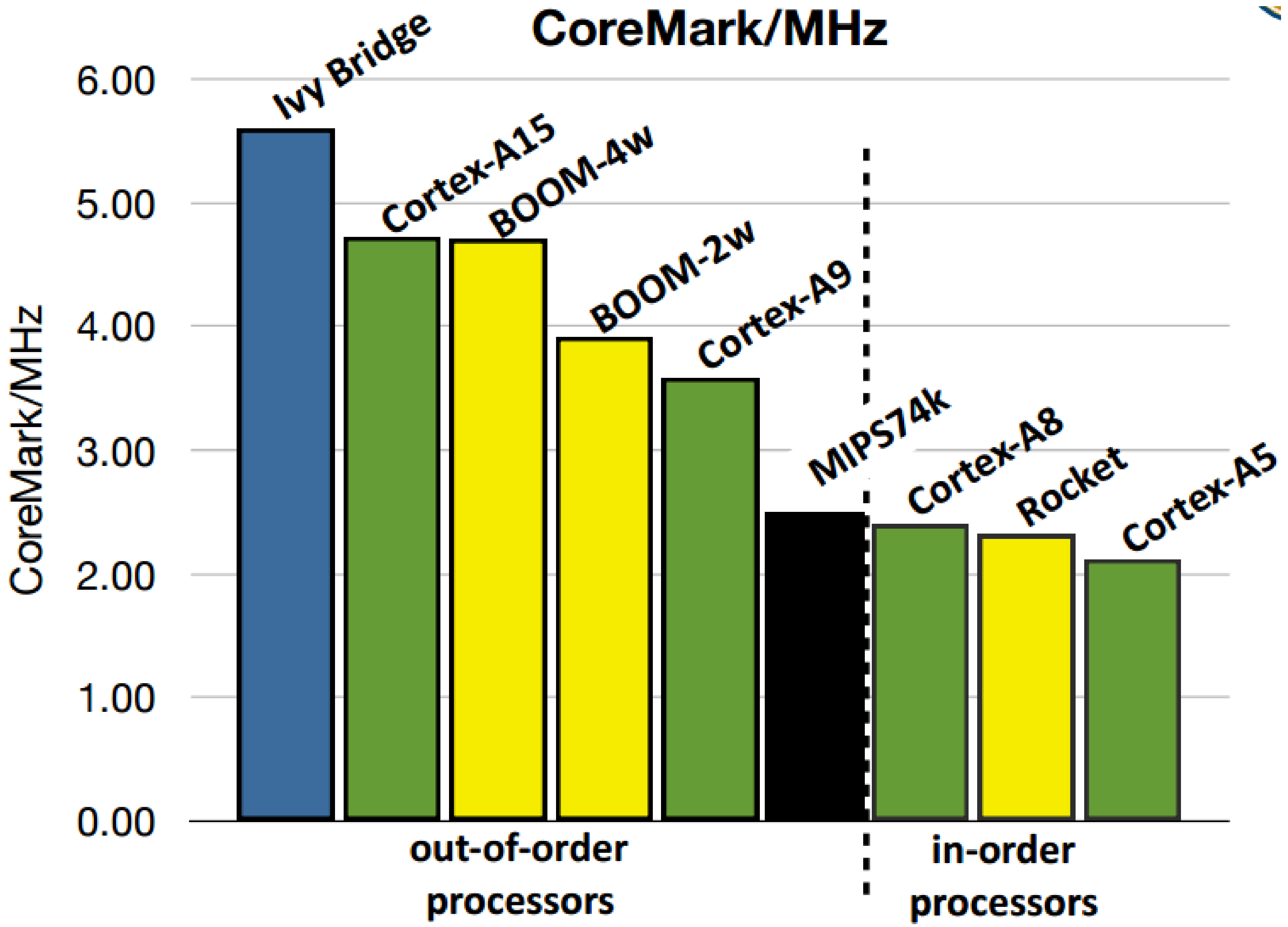
Question of the Day

- How many in-order cores do you think take up the same area as an out-of-order core?

How Many In-Order In The Same Area?



Performance?

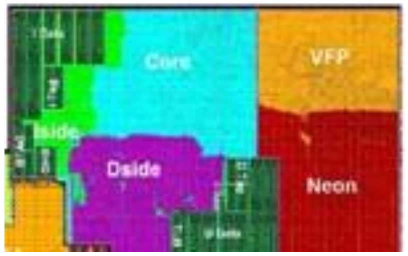


Comparison against ARM

Category	ARM Cortex-A9	RISC-V BOOM-2w
ISA	32-bit ARM v7	64-bit RISC-V v2 (RV64G)
Architecture	2 wide, 3+1 issue Out-of-Order 8-stage	2 wide, 3 issue Out-of-Order 6-stage
Performance	**3.59** CoreMarks/MHz	**3.91** CoreMarks/MHz
Process	TSMC 40GPLUS	TSMC 40GPLUS
Area with 32K caches	~2.5 mm^2	~1.00 mm^2
Area efficiency	**1.4** CoreMarks/MHz/mm^2	**3.9** CoreMarks/MHz/mm^2
Frequency	1.4 GHz	1.5 GHz
Power	**0.5-1.9 W** (2 cores + L2) @ TSMC 40nm, 0.8-2.0 GHz	**0.25 W** (1 core + L1) @ TSMC 45nm, 1 GHz

+9%!

note:
not to scale



Repositories

- BOOM
  - https://github.com/ucb-bar/riscv-boom
  - just the BOOM core code

- Rocket-chip
  - https://github.com/ucb-bar/rocket-chip
  - the rest of the SoC
  - chisel, rocket, uncore, junctions, riscv-tools, fpga-zynq, etc.
  - generates C++ emulator, verilog for FPGA
  - http://riscv.org/tutorial-hpca2015/riscv-rocket-chip-tutorial-bootcamp-hpca2015.pdf (for more information)

BOOM Quick-start

```
$ export ROCKETCHIP_ADDONS="boom"
$ git clone https://github.com/ucb-bar/rocket-chip.git
$ cd rocket-chip
$ git checkout boom
$ git submodule update --init
$ cd riscv-tools
$ git submodule update --init --recursive riscv-tests
$ cd ../emulator; make run CONFIG=BOOMCPPConfig
```

- "BOOM-chip" is (currently) a branch of rocket-chip
- "make run" builds and runs riscv-tests suite
- there are many different CONFIGs available!
  - rocket-chip/src/main/scala/PrivateConfigs.scala
  - rocket-chip/boom/src/main/scala/configs.scala
  - slightly different configs for different targets (CPP, FPGA, VLSI)

How do you verify and debug BOOM?

- parameterization makes this tough!
- tested with:
  - riscv-tests
    - (assembly functional tests + bare-metal micro-benchmarks)
  - CoreMark + riscv-pk
  - SPEC + Linux
  - riscv-torture

riscv-torture

- now open-source!
  - https://github.com/ucb-bar/riscv-torture
- torture generates random tests to stress the core pipeline
- runs test on Spike and your processor
  - architectural register state dumped to memory on program termination
  - diff state
  - if error found, finds smallest version of test that exhibits an error

riscv-torture quick-start

Run Rocket-chip (BOOM-chip?)

```
$ git clone https://github.com/ucb-bar/rocket-chip.git
$ cd rocket--chip
$ git submodule update --init
$ cd riscv--tools
$ git submodule update --init --recursive riscv-tests
$ cd ../emulator; make run CONFIG=BOOMCPPConfig
```

Run Torture

```
$ cd rocket-chip
$ git clone https://github.com/ucb-bar/riscv-torture.git
$ cd riscv-torture
$ git submodule update --init
$ vim Makefile  # change RTL_CONFIG=BOOMCPPConfig
$ make igentest # test that torture works, gen a single test
$ make cnight   # run C++ emulator overnight
```

Commit Logging

- BOOM can generate commit logs
  - (priv level, PC, inst, wb raddr, wb data)

- Spike supports generating commit logs!
  - configure Spike with "--enable-commitlog" flag
  - outputs to stderr

- only semi-automated
  - valid differences from Spike and hw
  - e.g., spinning on LR+SC

```
# modify the build.sh in riscv-tools
build_project riscv-isa-sim --prefix=$RISCV --with-fesvr=$RISCV --enable-commitlog
```

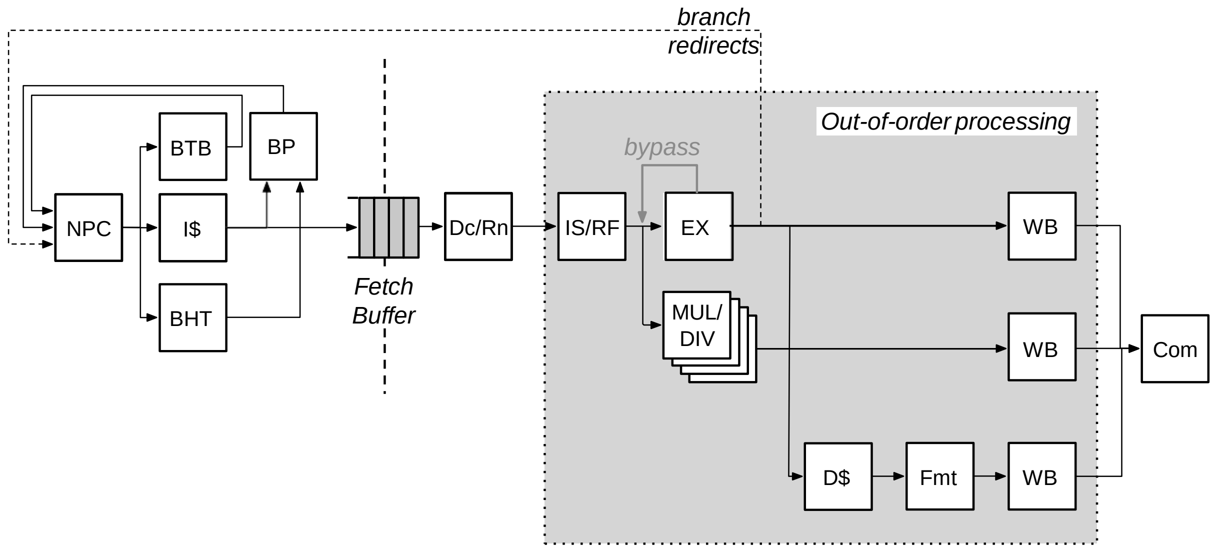
What documentation is available?

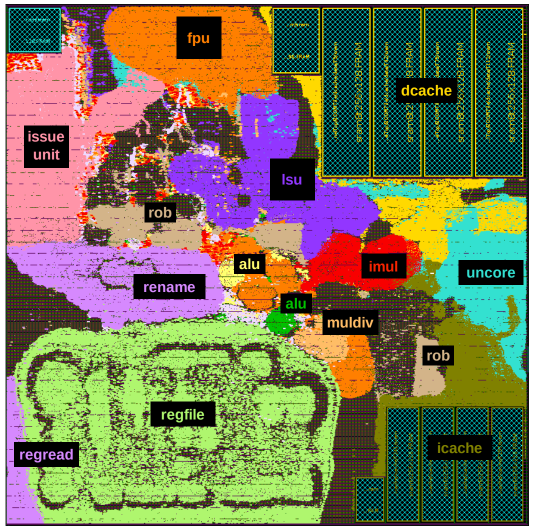
- A design document is in progress
  - https://github.com/ccelio/riscv-boom-doc

- Wiki
  - https://github.com/ucb-bar/riscv-boom/wiki

Conclusion

- BOOM is RV64G, runs SPEC on Linux on an FPGA
- BOOM is ~10k loc and 4 person-years of work
- excellent platform for prototyping new ideas
- now open-source!


branch redirects
Out-of-order processing
bypass
BTB
BP
NPC
I$
BHT
Fetch Buffer
Dc/Rn
IS/RF
EX
MUL/ DIV
WB
WB
WB
Com
D$
Fmt


fpu
dcache
issue unit
lsu
rob
alu
imul
uncore
rename
alu
muldiv
rob
regfile
regread
icache