Complex Pipelining: Out-of-Order Execution, Register Renaming and Exceptions

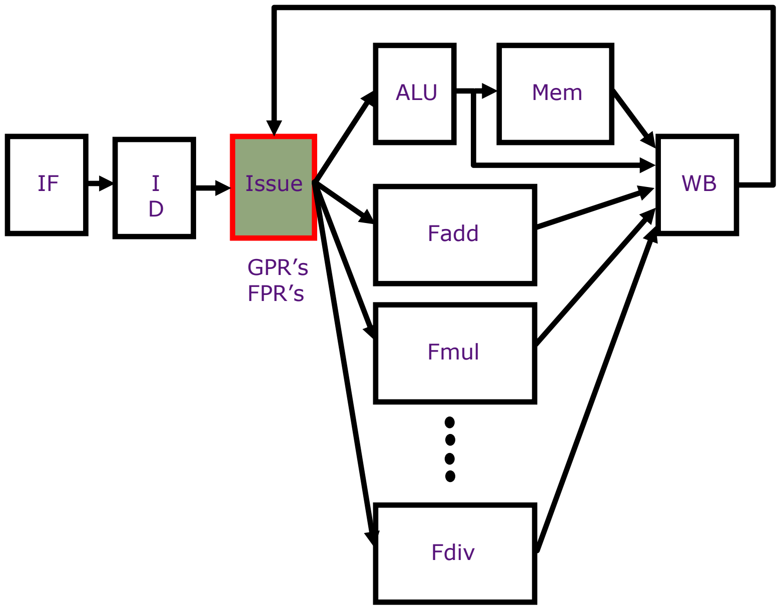
Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
M.I.T.

http://www.csg.csail.mit.edu/6.823

CDC 6600-style Scoreboard

Instructions are issued in order;
An instruction is issued only if

- It cannot cause a RAW hazard
  - ⇒*if operands are read immediately then no need to remember sources of instructions in the execute phases*
- It cannot cause a WAW hazard
  - ⇒*There can be at most instruction in the execute phase that can write in a particular register*



Scoreboard:
Two bit-vectors

Busy[FU#]: Indicates FU's availability
These bits are hardwired to FU's.

WP[reg#]: Records if a write is pending for a register
Set to true by the Issue stage and set to false by the WB stage

Reminder: Scoreboard Dynamics

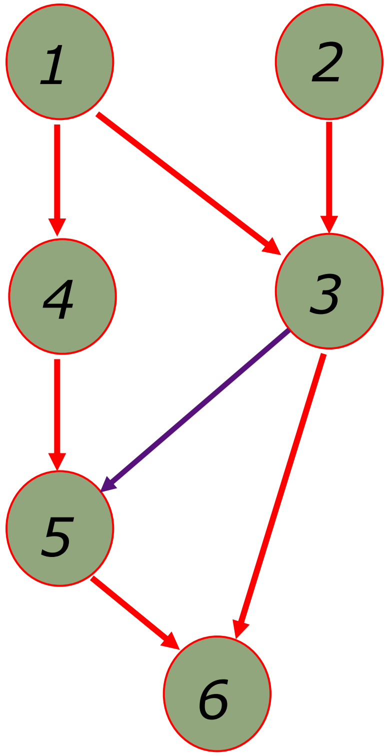
Issue time		Functional Unit Status										WP	WB time
		Int(1)	Add(1)	Mult(3)			Div(4)				WB	**WP**	
t0	I_1						f6					f6	
t1	I_2	f2						f6				f6, f2	
t2									f6		f2	f6, f2	I_2
t3	I_3			f0						f6		f6, f0	
t4					f0						f6	f6, f0	I_1
t5	I_4					f0	f8					f0, f8	
t6								f8			f0	f0, f8	I_3
t7	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	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

Issue checks:
WP[dest]?
WP[src1] or WP[src2]?
Busy[FU#]?

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*

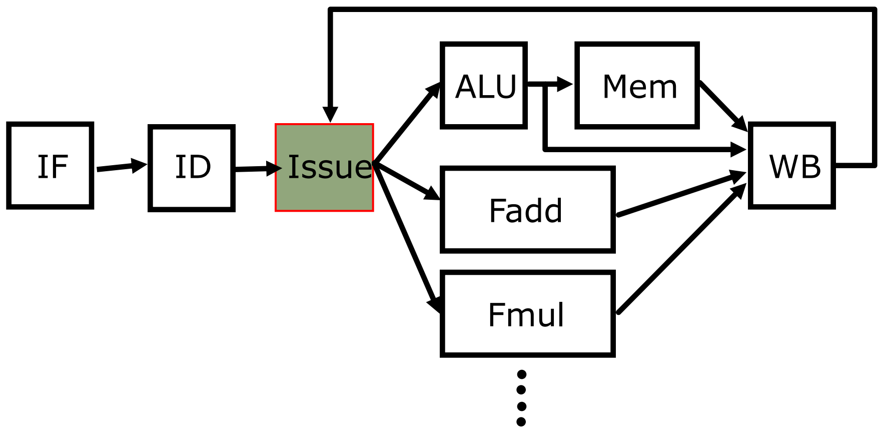


In-order: $1\ (2,\underline{1})\ .\ .\ .\ .\ .\ .\ \underline{2}\ 3\ 4\ \underline{4}\ \ \underline{3}\ 5\ .\ .\ .\ \underline{5}\ 6\ \underline{6}$

In-order restriction prevents instruction 4 from being dispatched

Out-of-Order Issue

How can we address the delay caused by a RAW dependence associated with the next in-order instruction?



Find something else to do!

- 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.
- Can issue any instruction in buffer whose RAW hazards are satisfied *(for now at most one dispatch per cycle).* A writeback (WB) may enable more instructions.

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*

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

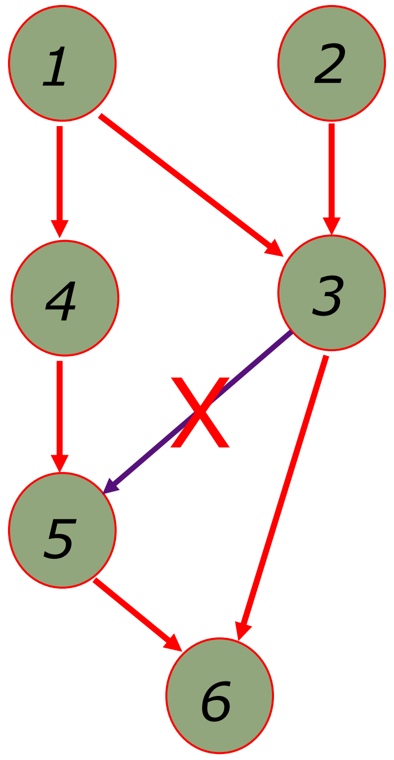
In-order: 1 (2,$\underline{1}$) . . . . . . $\underline{2}$ 3 4 $\underline{4}$ $\underline{3}$ 5 . . . $\underline{5}$ 6 $\underline{6}$

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 did not allow any significant improvement!

Instruction-level Parallelism via *Renaming*

					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 $\underline{6}$

Renaming eliminates WAR and WAW hazards
(renaming $\Rightarrow$ additional storage)

How many Instructions can be in the pipeline

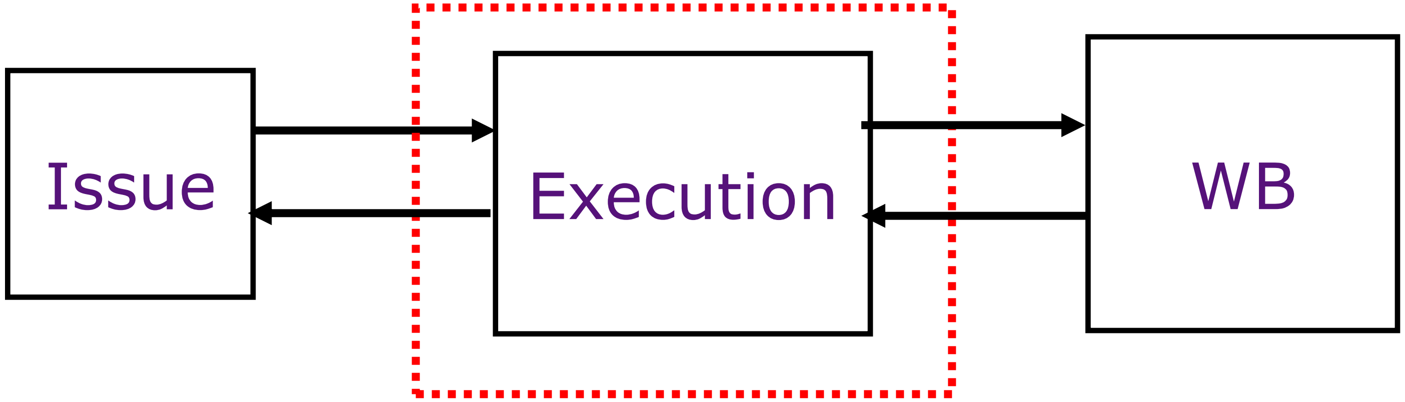
Which feature of an ISA limits the number of instructions in the pipeline?

____

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

Little’s Law

Throughput (T) = Number in Flight (N) / Latency (L)



Example:

4 floating point registers
8 cycles per floating point operation

$\Rightarrow$ *½ issues per cycle!*

Overcoming the Lack of Register Names

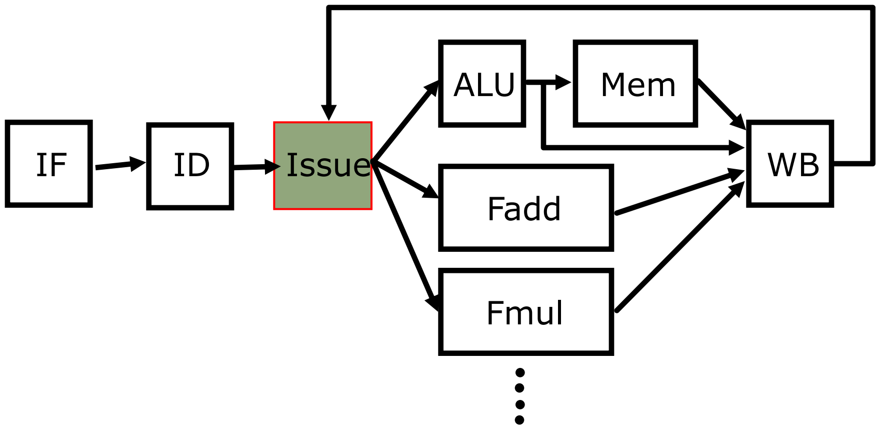
Floating Point pipelines often cannot be kept filled with small number of registers.

IBM 360 had only 4 Floating Point Registers

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

Yes, Robert Tomasulo of IBM suggested an ingenious solution in 1967 based on on-the-fly *register renaming*

Register Renaming



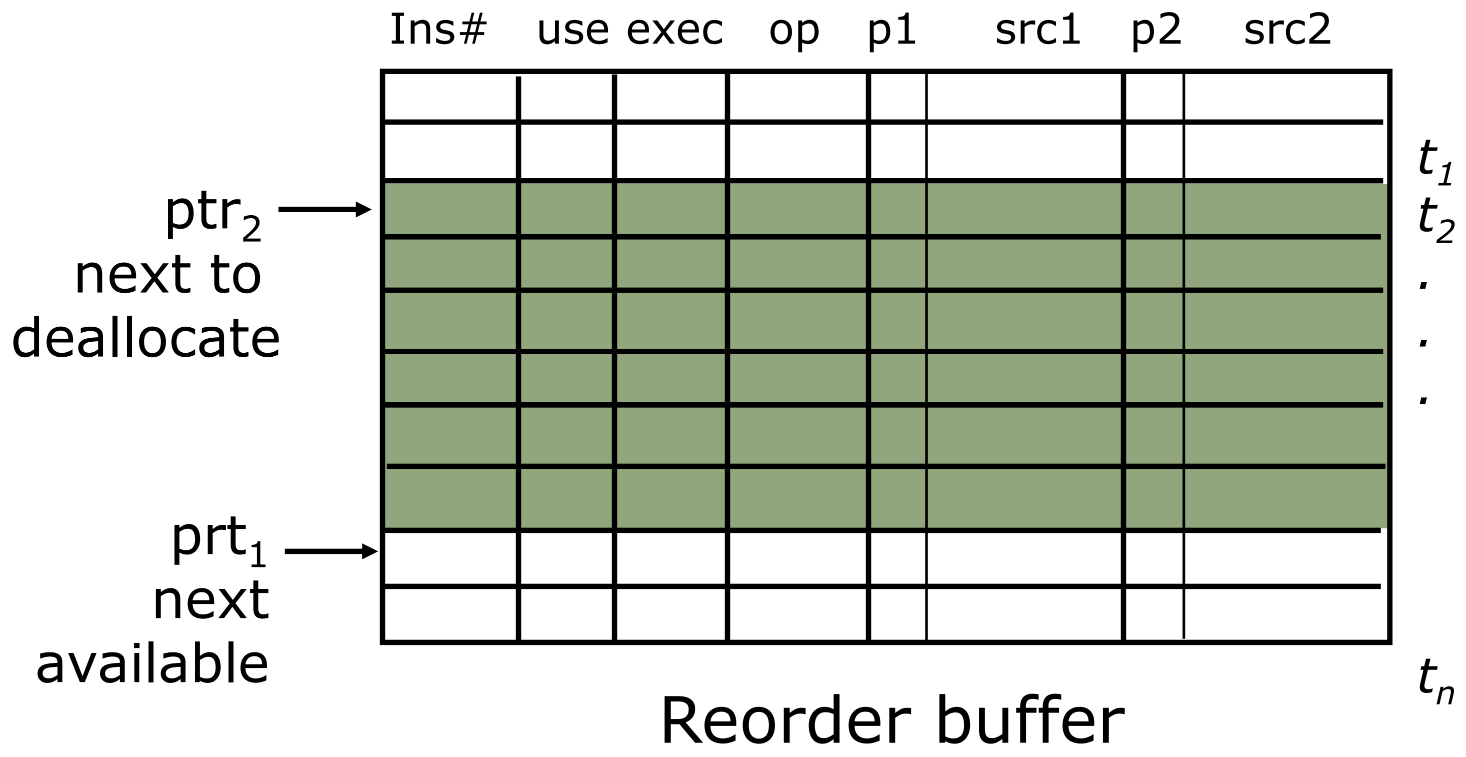
- Decode does register renaming and adds instructions to the issue stage reorder buffer (ROB)

  ⇒ 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

Dataflow execution



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		v1
F3		
F4		t5
F5		
F6		t3
F7		
F8		v4

Reorder buffer

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

data (v_i) / tag(t_i)

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

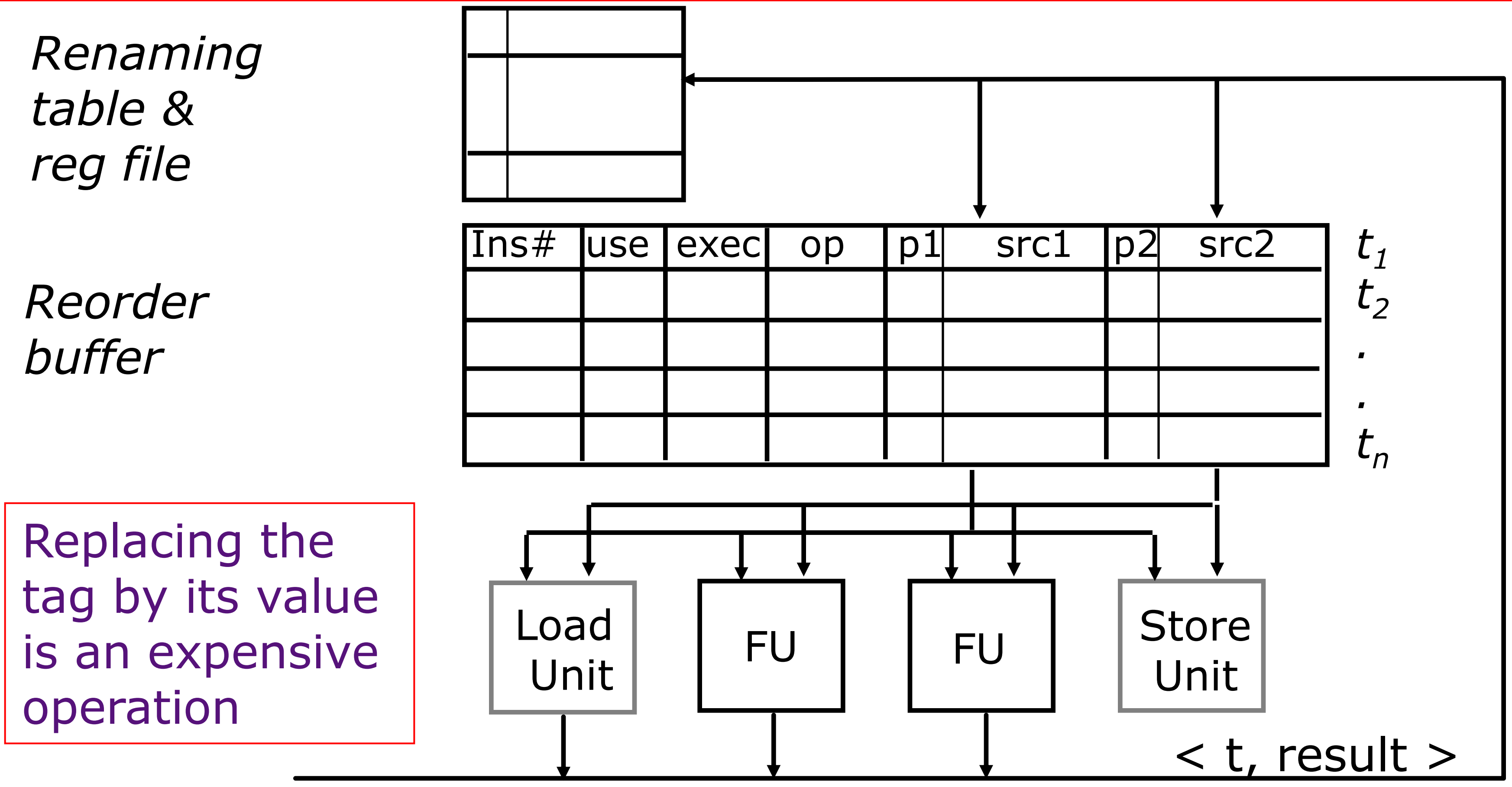
- *When are names in sources replaced by data?*

  Whenever an FU produces data

- *When can a name be reused?*

  Whenever an instruction completes

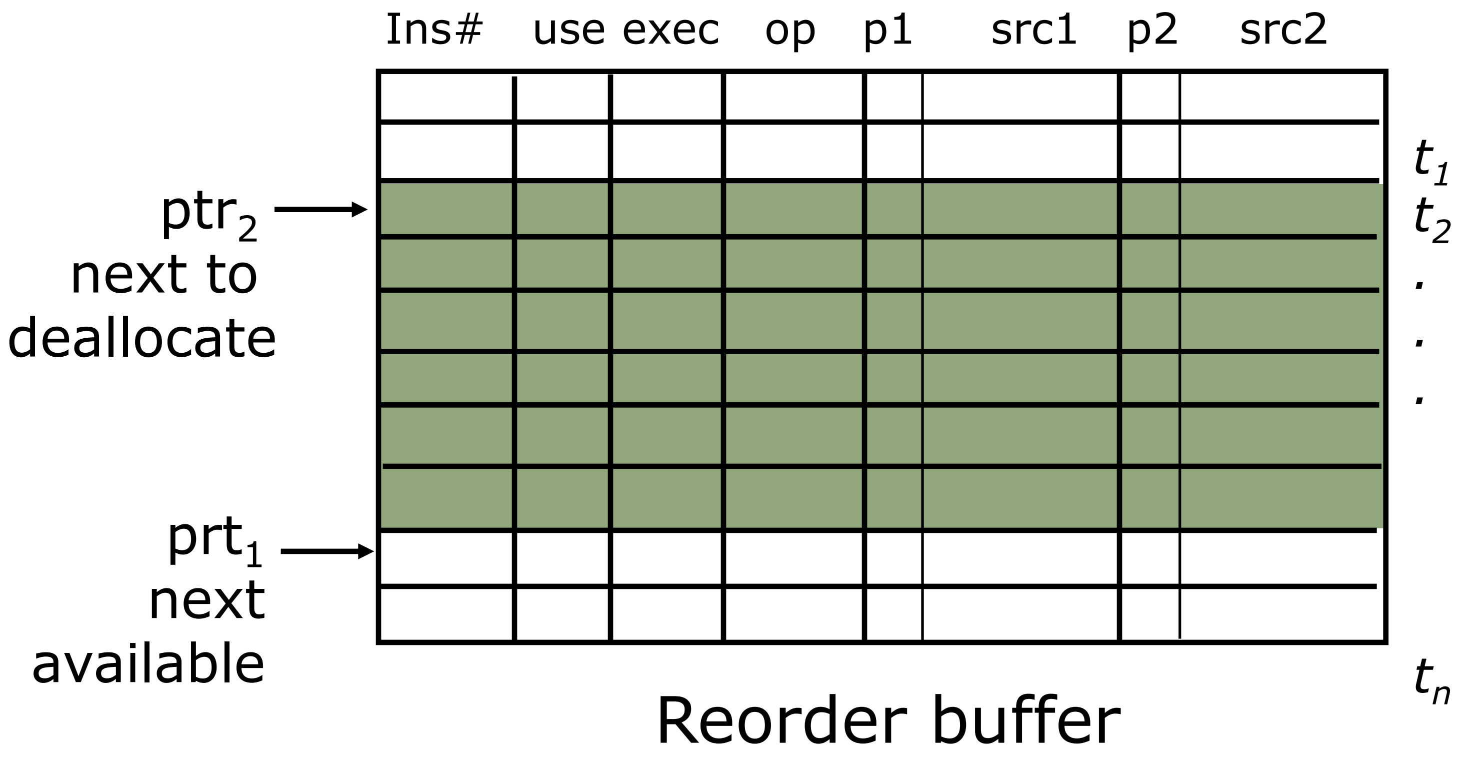
Data-Driven Execution



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

Simplifying Allocation/Deallocation

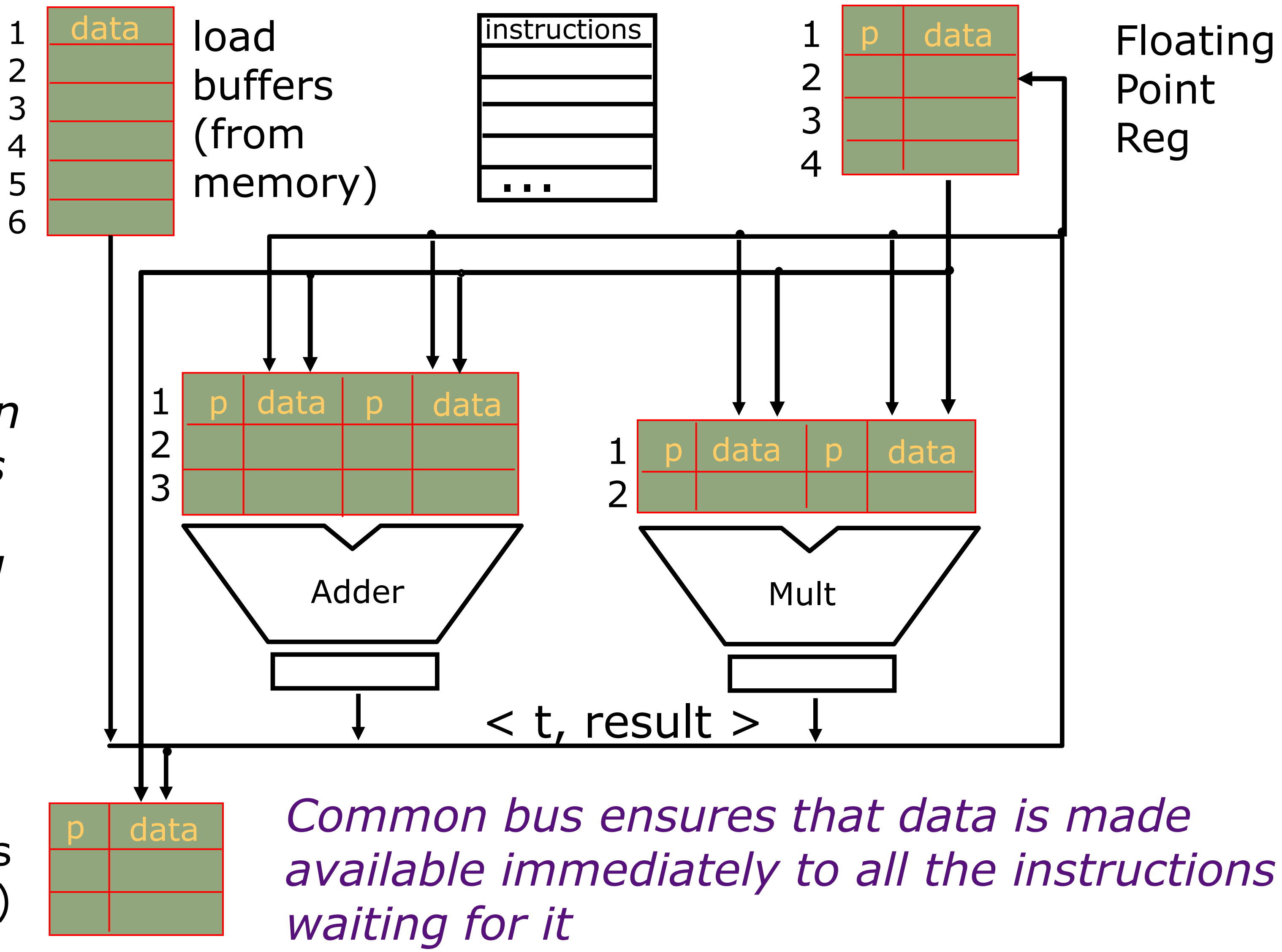


Instruction buffer is managed circularly

- "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

IBM 360/91 Floating Point Unit

R. M. Tomasulo, 1967



distribute instruction templates by functional units

store buffers (to memory)

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

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?

1. Effective on a very small class of programs
2. Did not address the memory latency problem which turned out be a much bigger issue than FU latency
3. Made exceptions imprecise

One more problem needed to be solved

Control transfers

More on this in the next lecture

Precise Exceptions

Exceptions are relatively unlikely events that need special processing, but where adding explicit control flow instructions is not desired, e.g., divide by 0, page fault

Exceptions can be viewed as an implicit conditional subroutine call that is inserted between two instructions.

Therefore, it must appear as if the exception is taken between two instructions (say I_i and I_{i+1})

- the effect of all instructions up to and including I_i is complete
- no effect of any instruction after I_i has taken place

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

Effect on Exceptions
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 $1 \quad 2 \quad \underline{2} \quad 3 \quad \underline{1} \quad 4 \quad \underline{3} \quad 5 \quad \underline{5} \quad \underline{4} \quad 6 \quad \underline{6}$

restore f2 *restore f10*

Consider exceptions

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

Exceptions

- Exceptions create a dependence on the value of the next PC

- Options for handling this dependence:

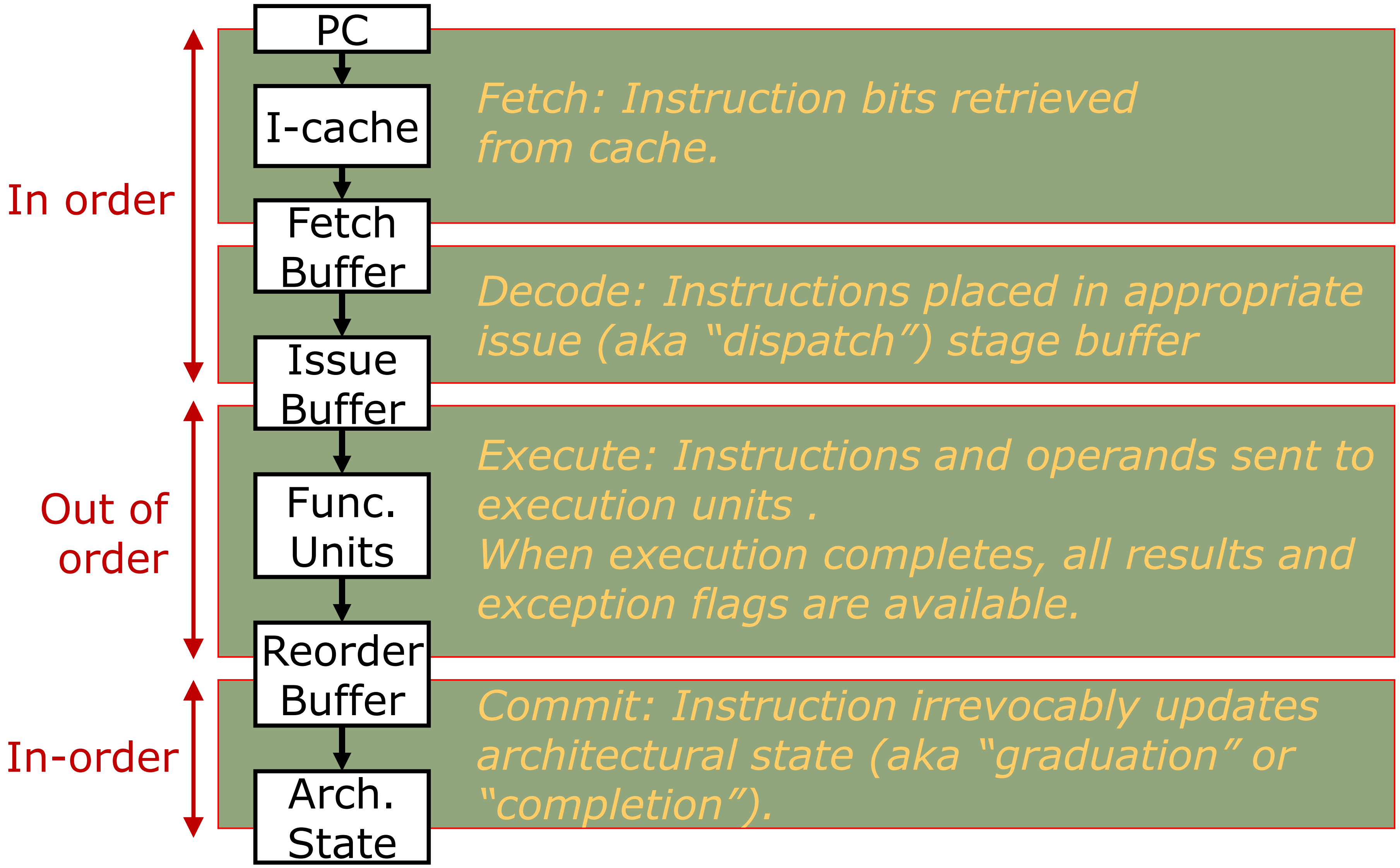
• Stall	No
• Bypass	No
• Find something else to do	No
• Change the architecture	Sometimes: Alpha, Multiflow
• Speculate!	Most common approach!

- How can we handle rollback on mis-speculation

  Delay state update until commit on speculated instructions

- Note: earlier exceptions must override later ones

Phases of Instruction Execution



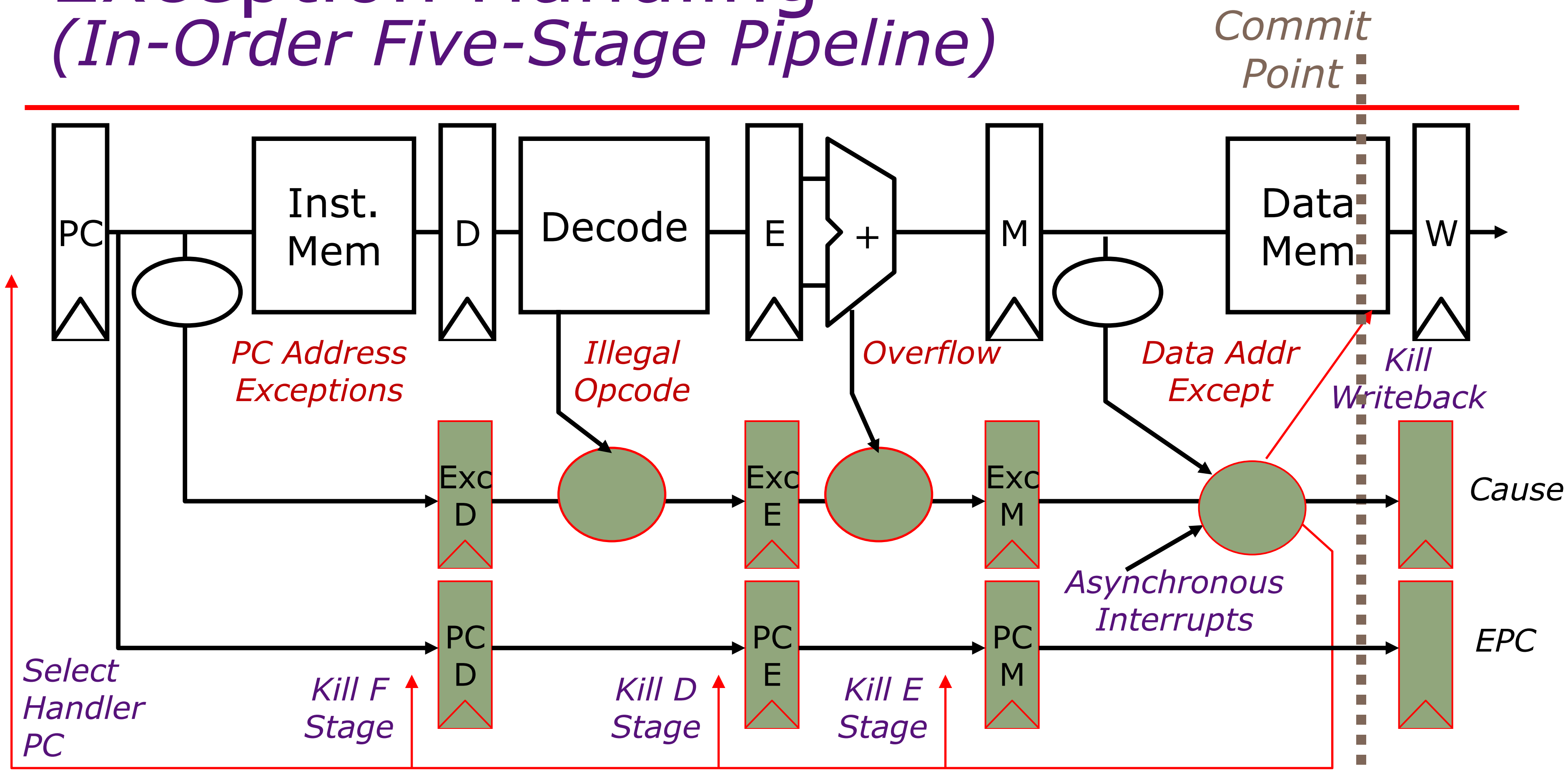
Fetch: Instruction bits retrieved from cache.

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").

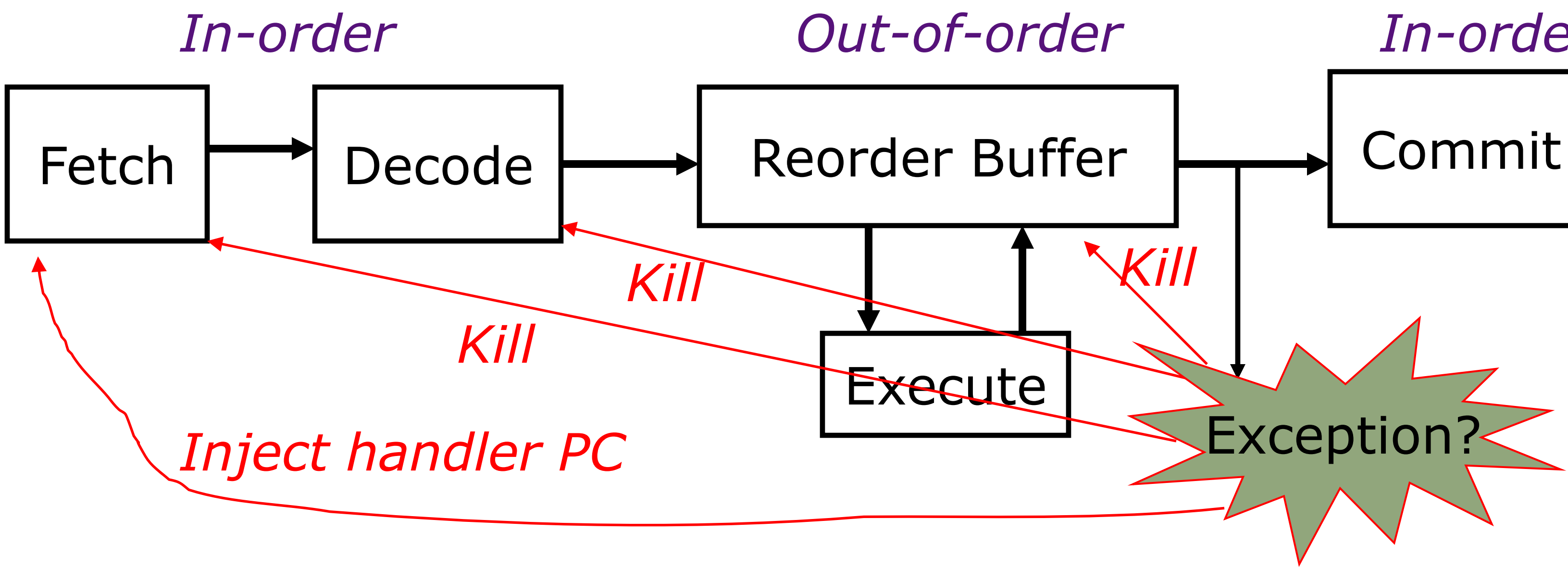
Exception Handling
(In-Order Five-Stage Pipeline)



Hold exception flags in pipeline until commit point (M stage)
- If exception at commit:
  - update Cause/EPC registers
  - kill all stages
  - fetch at handler PC

Inject external interrupts at commit point

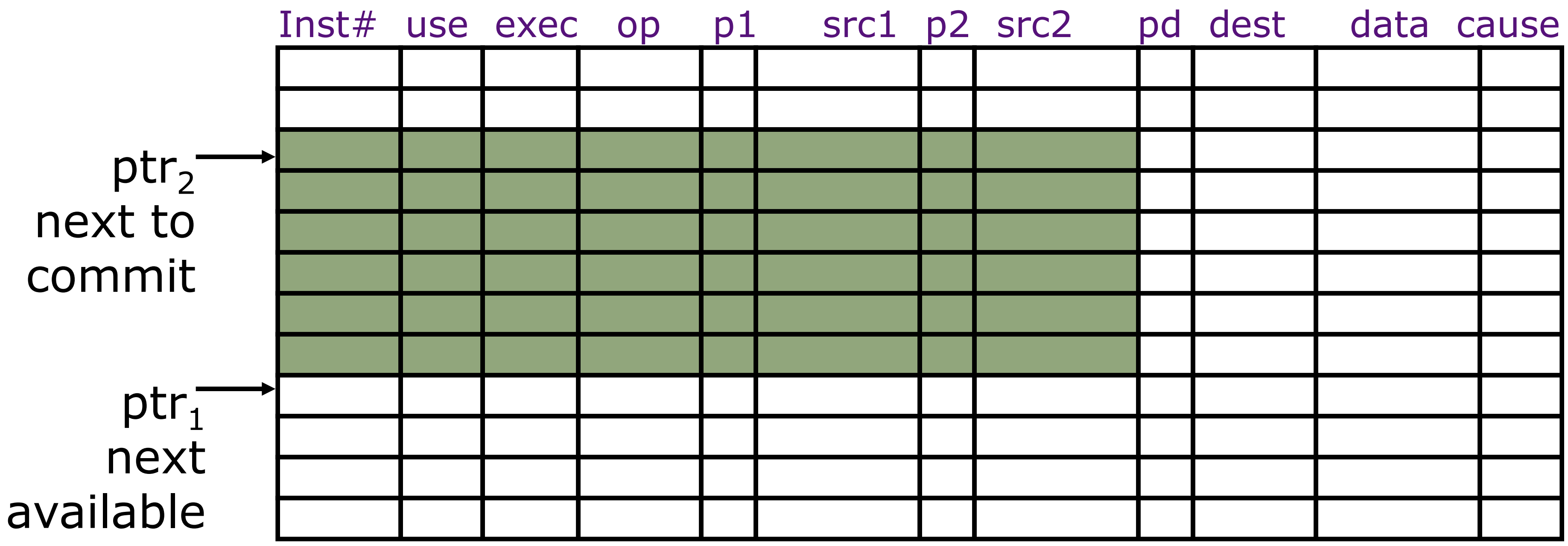
In-Order Commit for Precise Exceptions



- 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)

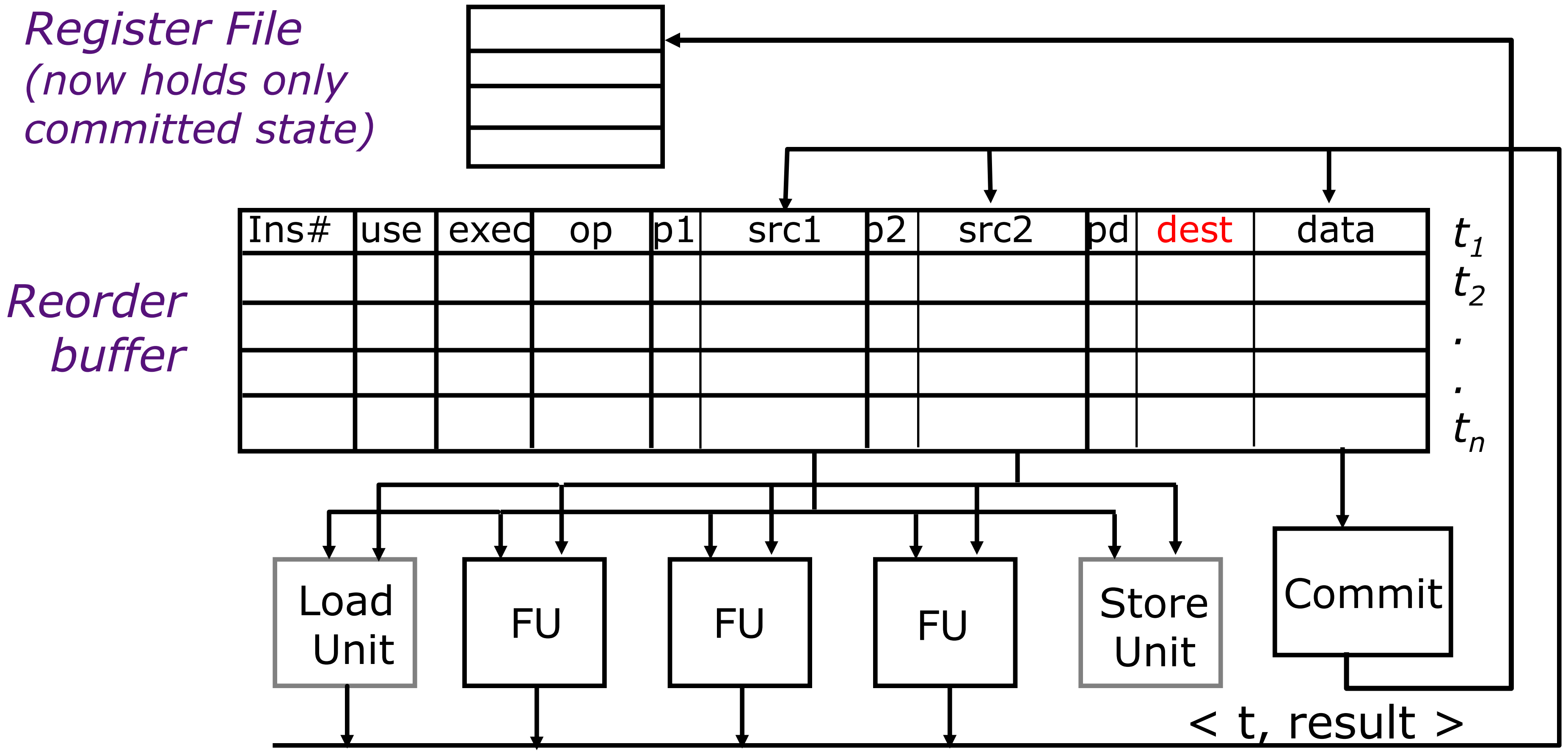
Extensions for Precise Exceptions



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

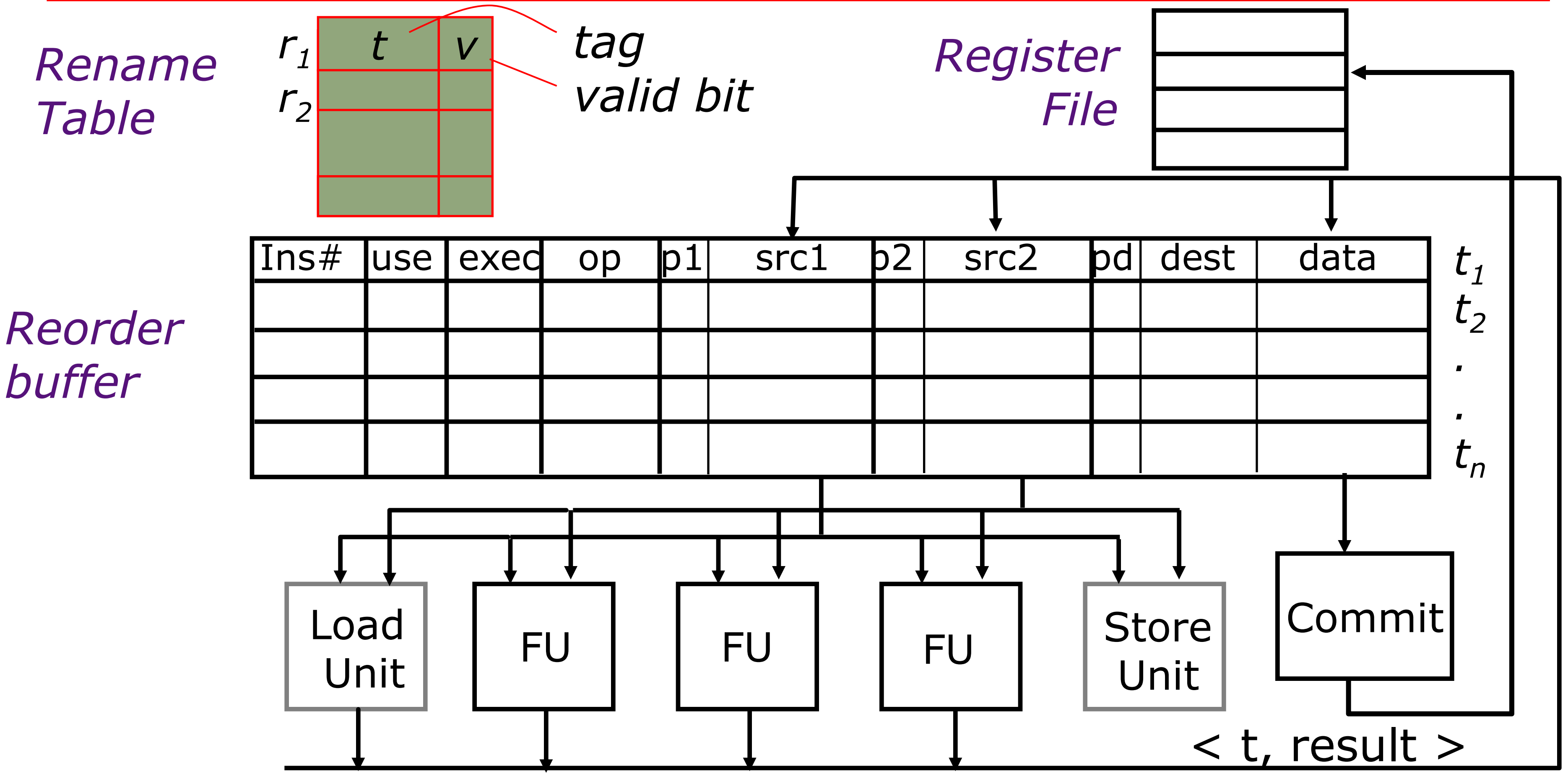


Register file does not contain renaming tags any more.

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 lookup.
It needs to be cleared after each exception taken.
When else are valid bits cleared? *Control transfers*

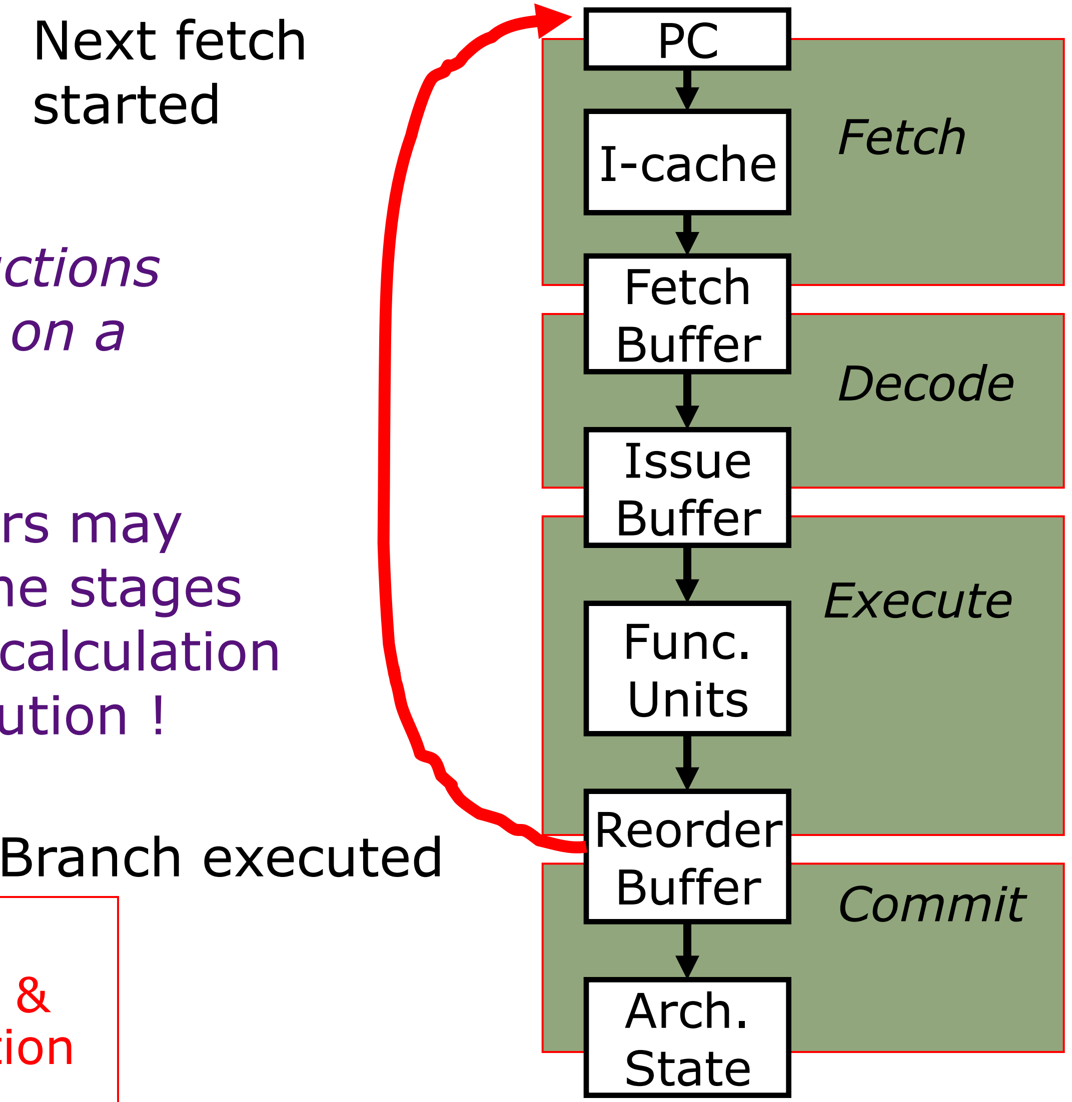
Physical Register Files

- Reorder buffers are space inefficient – a data value may be stored in multiple places in the reorder buffer
- idea – keep all data values in a physical register file
  - Tag represents the name of the data value and name of the physical register that holds it
  - Reorder buffer contains only tags

Thus, 64 data values may be replaced by 8-bit tags for a 256 element physical register file

More on this in later lectures …

Branch Penalty



How many instructions need to be killed on a misprediction?

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

next lecture:
Branch prediction & Speculative execution