

CDC Programing Card

CONTROL DATA 6400/6500/6600/6700 COMPUTER SYSTEMS CENTRAL PROCESSOR INSTRUCTIONS			
BRANCH UNIT		MULTIPLY UNIT	
00	PS Program Stop	40	FXi Xj * Xk Floating Product
010	RJ Return Jump to K	41	RXi Xj * Xk Round Floating Product
013	XJ Central Exchange Jump	42	DXi Xj * Xk Floating DP Product
02	JP Go to K + Bi	DIVIDE UNIT	
030	ZR Go to K if Xj = zero	44	FXi Xj/Xk Floating Divide
031	NZ Go to K if Xj ≠ zero	45	RXi Xj/Xk Round Floating Divide
032	PL Go to K if Xj = positive	46	NO No Operation
033	NG Go to K if Xj = negative	47	CXi Xk Sum of ones in Xk to Xi
034	IR Go to K if Xj is in range	INCREMENT UNIT	
035	OR Go to K if Xj is out of range	50	SAi Aj + K Sum of Aj & K to Ai
036	DF Go to K if Xj is definite	51	SAi Bj + K Sum of Bj & K to Ai
037	ID Go to K if Xj is indefinite	52	SAi Xj + K Sum of Xj & K to Ai
04	EQ Go to K if Bi = Bj	53	SAi Xj + Bk Sum of Xj & Bk to Ai
05	NE Go to K if Bi ≠ Bj	54	SAi Aj + Bk Sum of Aj & Bk to Ai
06	GE Go to K if Bi > Bj	55	SAi Aj - Bk Diff of Aj & Bk to Ai
07	LT Go to K if Bi < Bj	56	SAi Bj + Bk Sum of Bj & Bk to Ai
		57	SAi Bj - Bk Diff of Bj & Bk to Ai
BOOLEAN UNIT		60	SBi Aj + K Sum of Aj & K to Bi
10	BXi Xj Xmit Xj to Xi	61	SBi Bj + K Sum of Bj & K to Bi
11	BXi Xj * Xk Logical Product	62	SBi Xj + K Sum of Xj & K to Bi
12	BXi Xj + Xk Logical Sum	63	SBi Xj + Bk Sum of Xj & Bk to Bi
13	BXi Xj - Xk Logical Difference	64	SBi Aj + Bk Sum of Aj & Bk to Bi
14	BXi - Xk Xmit Xk comp to Xi	65	SBi Aj - Bk Diff of Aj & Bk to Bi
15	BXi - Xk * Xj Log Prod Xj & Xk comp	66	SBi Bj + Bk Sum of Bj & Bk to Bi
16	BXi - Xk + Xj Log Sum Xj & Xk comp	67	SBi Bj - Bk Diff of Bj & Bk to Bi
17	BXi - Xk - Xj Log Diff Xj & Xk comp	70	SXi Aj + K Sum of Aj & K to Xi
SHIFT UNIT		71	SXi Bj + K Sum of Bj & K to Xi
20	LXi jk Left Shift Xi by jk	72	SXi Xj + K Sum of Xj & K to Xi
21	AXi jk Right Shift Xi by jk	73	SXi Xj + Bk Sum of Xj & Bk to Xi
22	LXi Bj Xk Left Shift Xk by Bj to Xi	74	SXi Aj + Bk Sum of Aj & Bk to Xi
23	AXi Bj Xk Right Shift Xk by Bj to Xi	75	SXi Aj - Bk Diff of Aj & Bk to Xi
24	NXi Bj Xk Normalize Xk to Xi & Bj	76	SXi Bj + Bk Sum of Bj & Bk to Xi
25	ZXi Bj Xk Round & Normalize Xk	77	SXi Bj - Bk Diff of Bj & Bk to Xi
26	UXi Bj Xk Unpack Xk to Xi & Bj	EXTENDED CORE STORAGE	
27	PXi Bj Xk Pack Xi from Xk & Bj	011	RE Bj + K Read ECS
43	MXi jk Form jk mask in Xi	012	WE Bj + K Write ECS
ADD UNIT		EXIT MODE	
30	FXi Xj + Xk Floating Sum	Bit	
31	FXi Xj - Xk Floating Diff	0	Address out of range
32	DXi Xj + Xk Floating DP Sum	1	Operand out of range (Infinite)
33	DXi Xj - Xk Floating DP Diff	2	Indefinite operand
34	RXi Xj + Xk Round Floating Sum	SYMBOLS	
35	RXi Xj - Xk Round Floating Diff	Comp	= Complement
LONG ADD UNIT		DP	= Double Precision
36	IXi Xj + Xk Integer Sum	Xmit	= Transmit
37	IXi Xj - Xk Integer Difference		
Pub. No. 60164500			
Rev C			

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

Joel Emer

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

CDC 6600-style Scoreboard

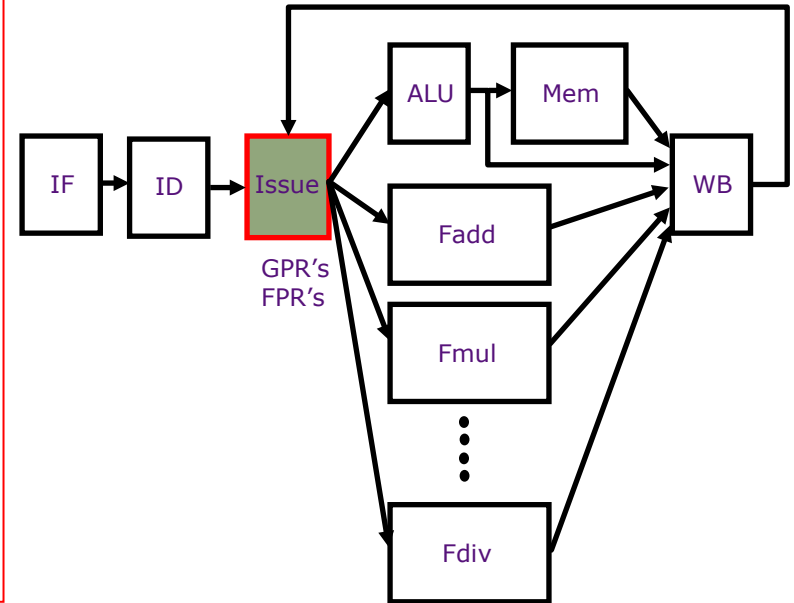
Instructions are issued in order.

An instruction is issued only if

- It cannot cause a RAW hazard
 - It cannot cause a WAW hazard
- ⇒ There can be at most instruction in the execute phase that can write to a particular register*

WAR hazards are not possible

- Due to in-order issue + operands read immediately



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

Scoreboard Dynamics

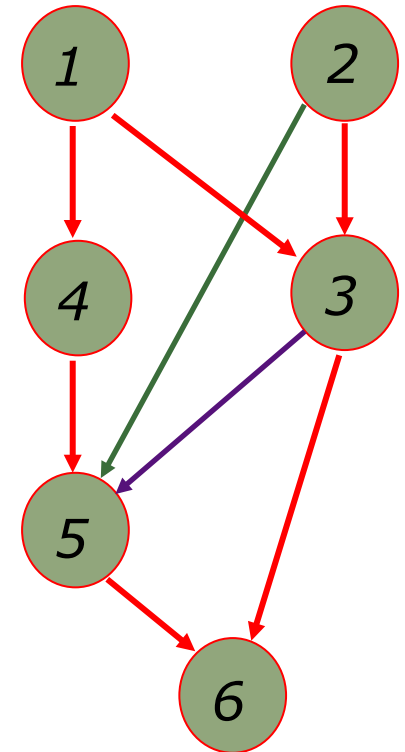
Issue time	Functional Unit Status										Registers Reserved for Writes (WP)	WB time
	Int(1)	Add(1)	Mult(3)			Div(4)			WB			
t0 <i>I</i> ₁						f6					f6	
t1 <i>I</i> ₂	f2						f6				f6, f2	
t2								f6	f2		f6, f2	<u>I</u> ₂
t3 <i>I</i> ₃			f0					f6			f6, f0	
t4				f0					f6		f6, f0	<u>I</u> ₁
t5 <i>I</i> ₄					f0	f8					f0, f8	
t6							f8		f0		f0, f8	<u>I</u> ₃
t7 <i>I</i> ₅		f10						f8			f8, f10	
t8									f8	f10	f8, f10	<u>I</u> ₅
t9									f8		f8	<u>I</u> ₄
t10 <i>I</i> ₆		f6									f6	
t11									f6		f6	<u>I</u> ₆

<i>I</i> ₁	DIVD	f6,	f6,	f4
<i>I</i> ₂	LD	f2,	45(r3)	
<i>I</i> ₃	MULTD	f0,	f2,	f4
<i>I</i> ₄	DIVD	f8,	f6,	f2
<i>I</i> ₅	SUBD	f10,	f0,	f6
<i>I</i> ₆	ADDD	f6,	f8,	f2

In-Order Issue Limitations

An example

					<i>latency</i>
1	LD	F2,	34(R2)		1
2	LD	F4,	45(R3)		<i>long</i>
3	MULTD	F6,	F4,	F2	3
4	SUBD	F8,	F2,	F2	1
5	DIVD	F4,	F2,	F8	4
6	ADDD	F10,	F6,	F4	1

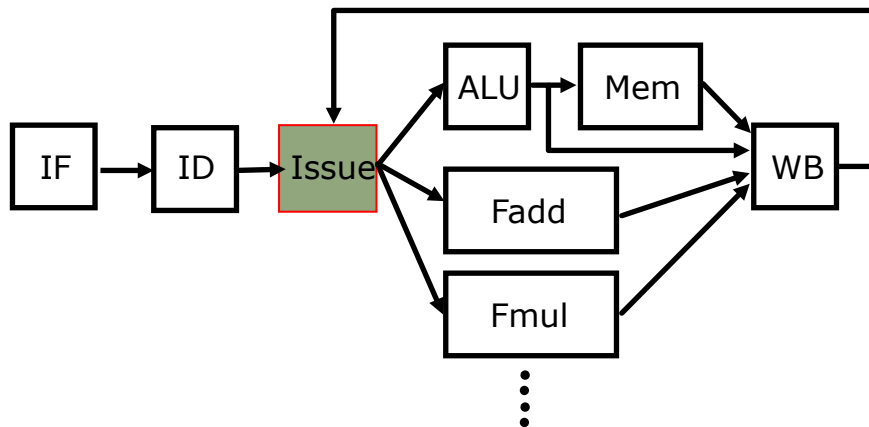


In-order: 1 (2,1) 2 3 4 4 3 5 . . . 5 6 6

In-order restriction 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!

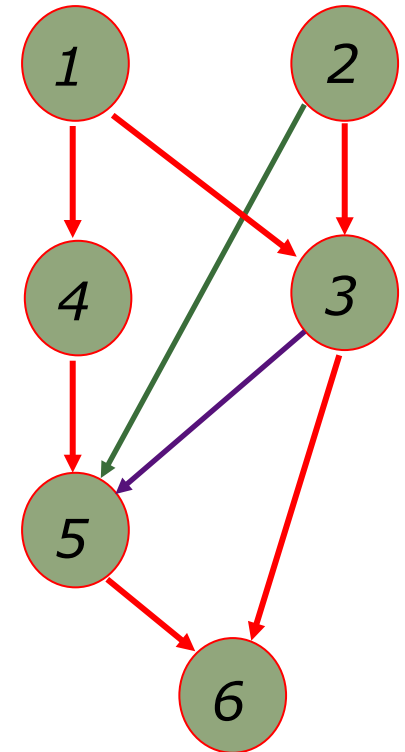
- Issue stage buffer holds multiple instructions waiting to issue.
- Decode adds next instruction to buffer if there is space and the instruction does not cause a WAR or WAW hazard.
- Can issue any instruction in buffer whose RAW hazards are satisfied (*for now at most one dispatch per cycle*).

Note: A writeback (WB) may enable more instructions.

In-Order Issue Limitations

An example

					<i>latency</i>
1	LD	F2,	34(R2)		1
2	LD	F4,	45(R3)		<i>long</i>
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 2 3 5 . 3 . 5 6 6

WAR/WAW hazards prevent instruction 5 from being dispatched

Out-of-order execution did not produce a significant improvement!

How many Instructions can be in the pipeline

Throughput is limited by number of instructions in flight, but which feature of an ISA limits the number of instructions in the pipeline?

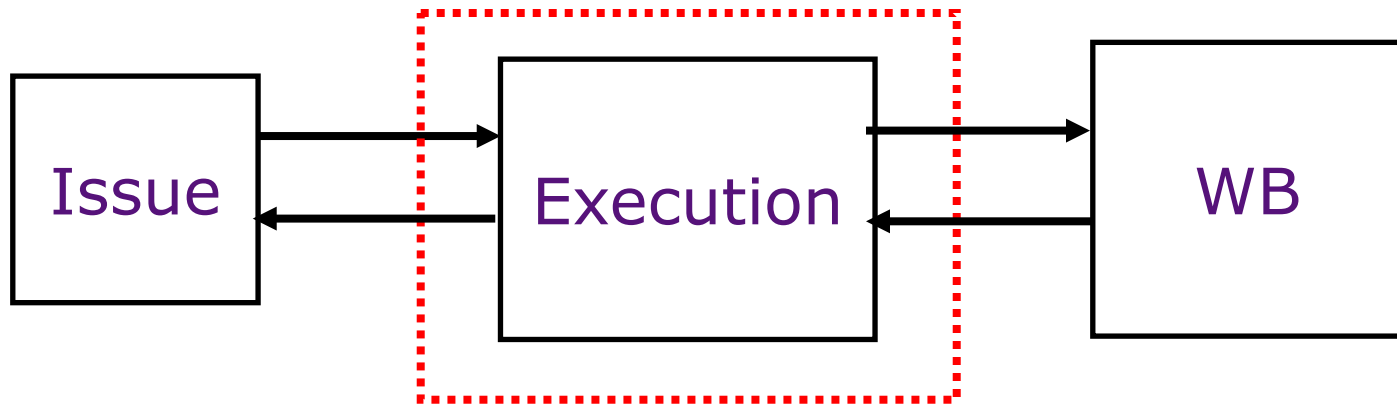
Number of Registers

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

How can we better understand the impact of number of registers on throughput?

Little's Law

Throughput ($\bar{T}$) = Number in Flight ($\bar{N}$) / Latency ($\bar{L}$)



Example:

4 floating point registers

8 cycles per floating point operation

⇒ ½ issues per cycle!

Overcoming the Lack of Register Names

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

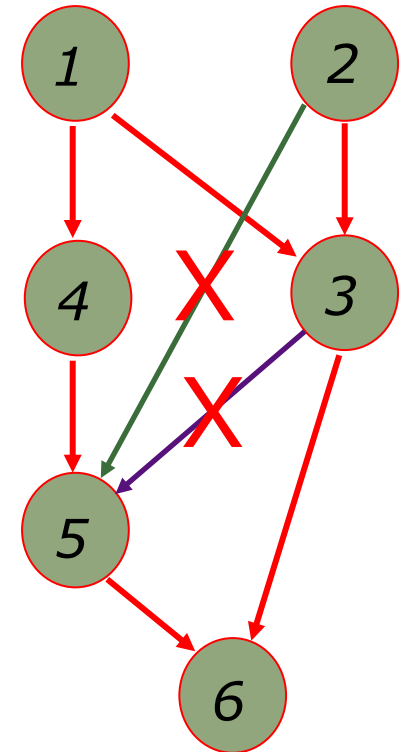
IBM 360 had only 4 Floating Point Registers

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*

Instruction-level Parallelism via *Renaming*

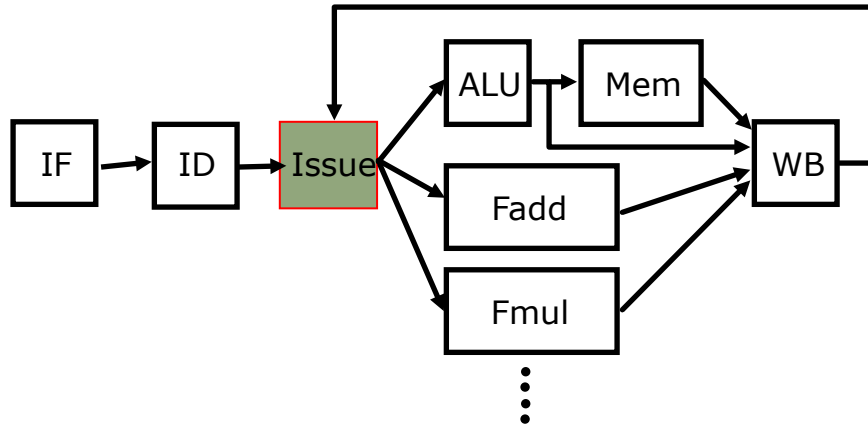
					<i>latency</i>
1	LD	F2,	34(R2)		1
2	LD	F4,	45(R3)		<i>long</i>
3	MULTD	F6,	F4,	F2	3
4	SUBD	F8,	F2,	F2	1
5	DIVD	F4' ,	F2,	F8	4
6	ADDD	F10,	F6,	F4'	1



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

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

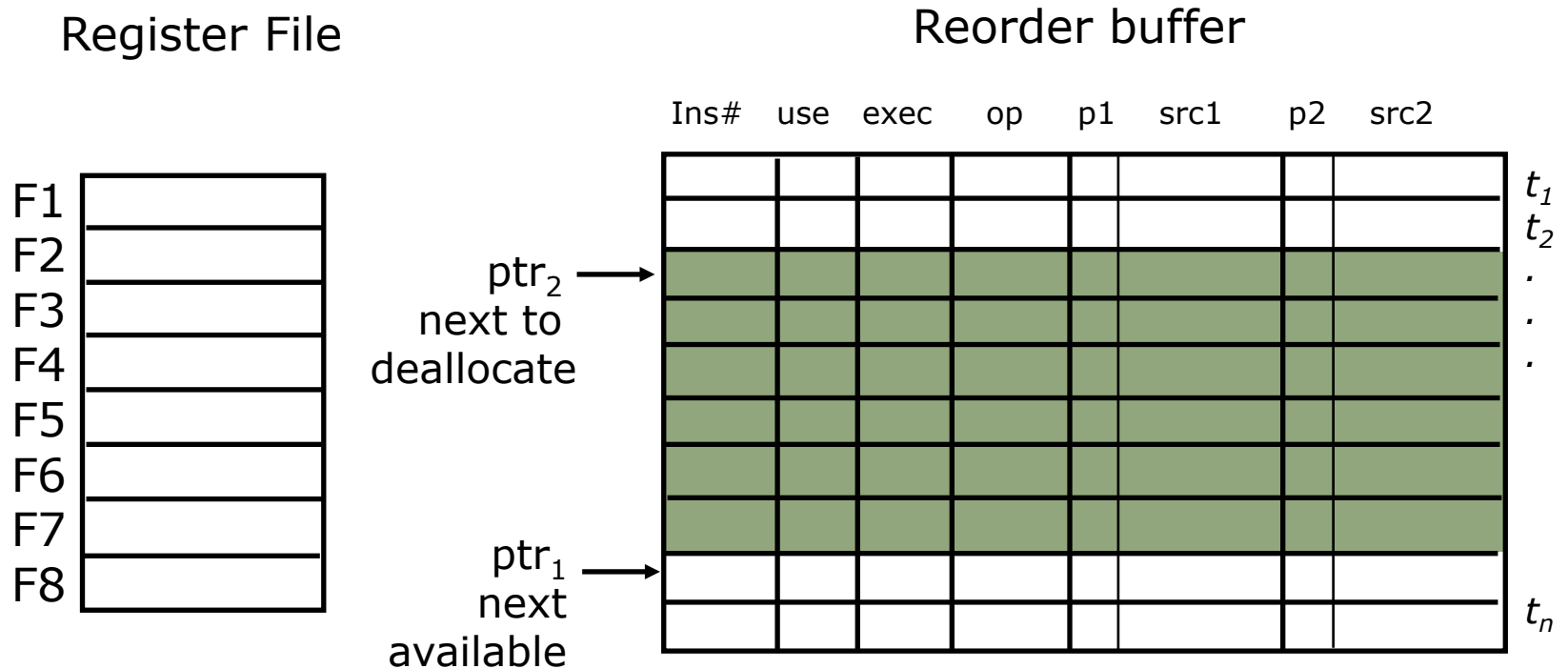
Handling register dependencies



- Decode does register renaming, providing a new spot for each register write
 - Renaming eliminates WAR and WAW hazards by allowing use of more storage space
- Renamed instructions added to an issue stage structure, called the **reorder buffer (ROB)**. Any instruction in the ROB whose RAW hazards have been satisfied can be dispatched
 - Out-of-order or dataflow execution handles RAW hazards

Reorder Buffer

Smith and Pleszkun, 1985



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 ("present" bits p1 and p2 are set)

Is it obvious where an architectural register value is? **No**

Renaming & Out-of-order Issue

Renaming table & reg file

	p	data
F1		
F2		
F3		
F4		
F5		
F6		
F7		
F8		

Holds data (v_i)
or tag(t_i)

Reorder buffer

Ins#	use	exec	op	p1	src1	p2	src2

t_1
 t_2
 t_3
 t_4
 t_5
.
.

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

Renaming & Out-of-order Issue

An example

Renaming table & reg file

Reorder buffer

	p	data
F1		
F2	0	v1
F3		
F4	0	t2
F5		
F6	0	t3
F7		
F8	0	v4

data (v_i) / tag(t_i)

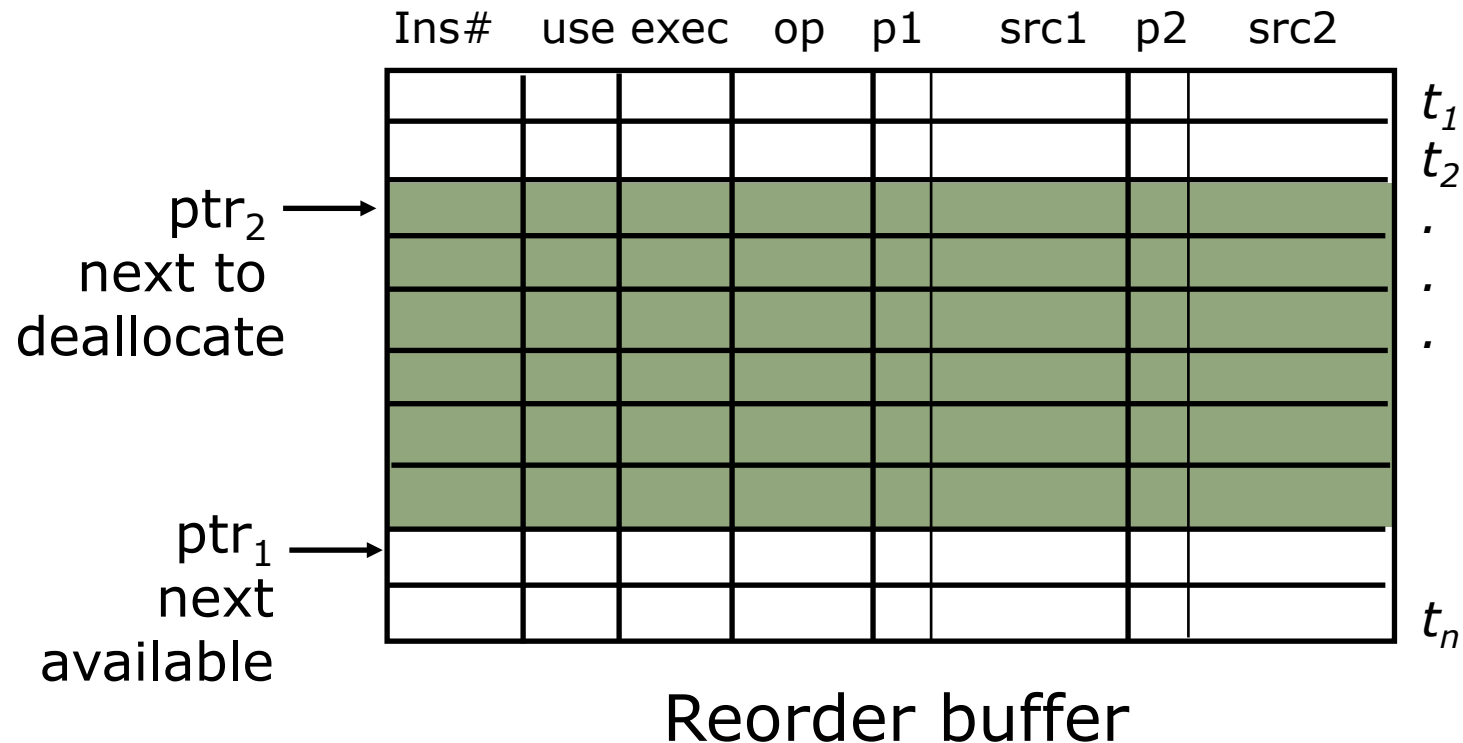
Ins#	use	exec	op	p1	src1	p2	src2
1	0	0	LD				
2	0	0	LD				
3	1	0	MUL	0	v2	1	v1
4	0	0	SUB	1	v1	1	v1
5	1	0	DIV	1	v1	0	v4

$$\begin{matrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ . \\ . \end{matrix}$$

- *Insert instruction in ROB*
- *Issue instruction from ROB*
- *Complete instruction*
- *Empty ROB entry*

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

Simplifying Allocation/Deallocation



Instruction buffer is managed circularly

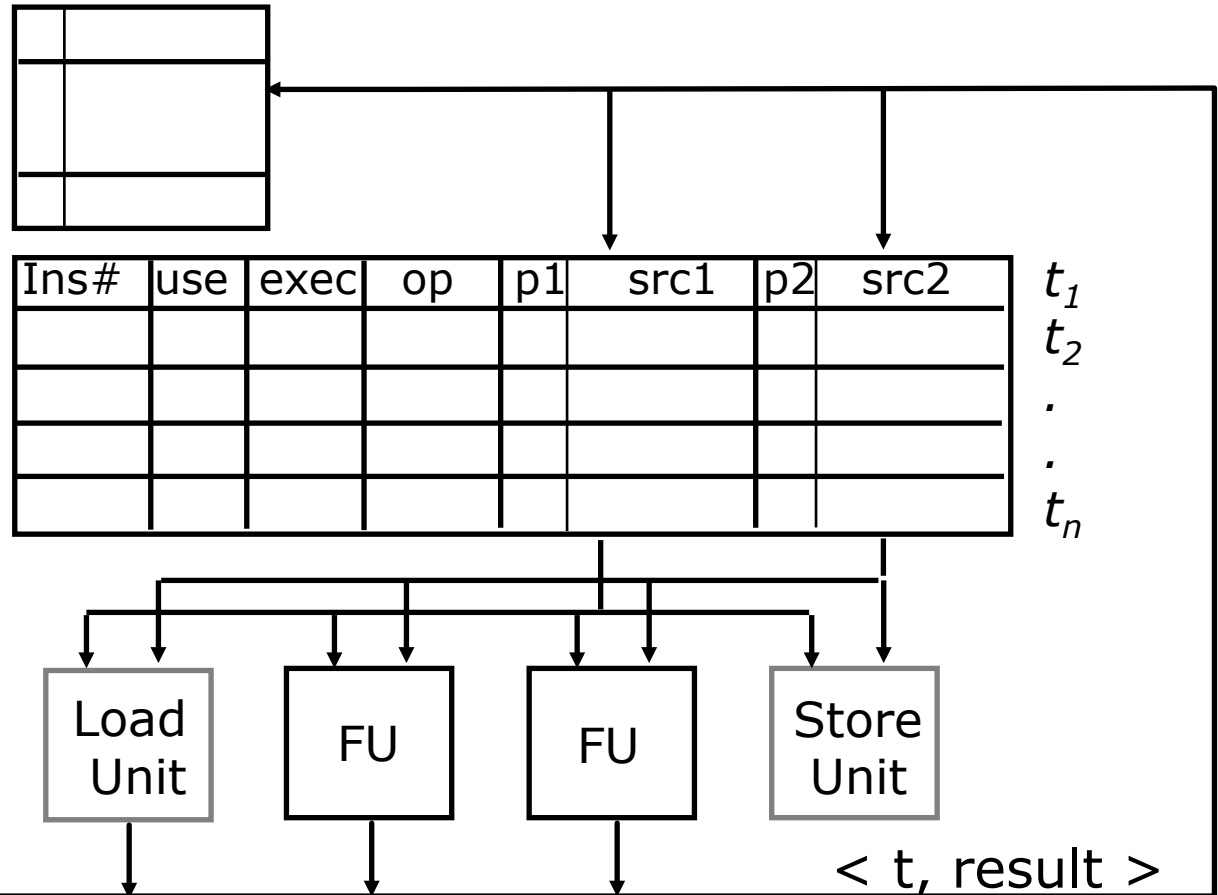
- Set "exec" bit when instruction begins execution
- When an instruction completes its "use" bit is marked free
- Increment ptr_2 only if the "use" bit is marked free

Data-Driven Execution

*Renaming
table &
reg file*

*Reorder
buffer*

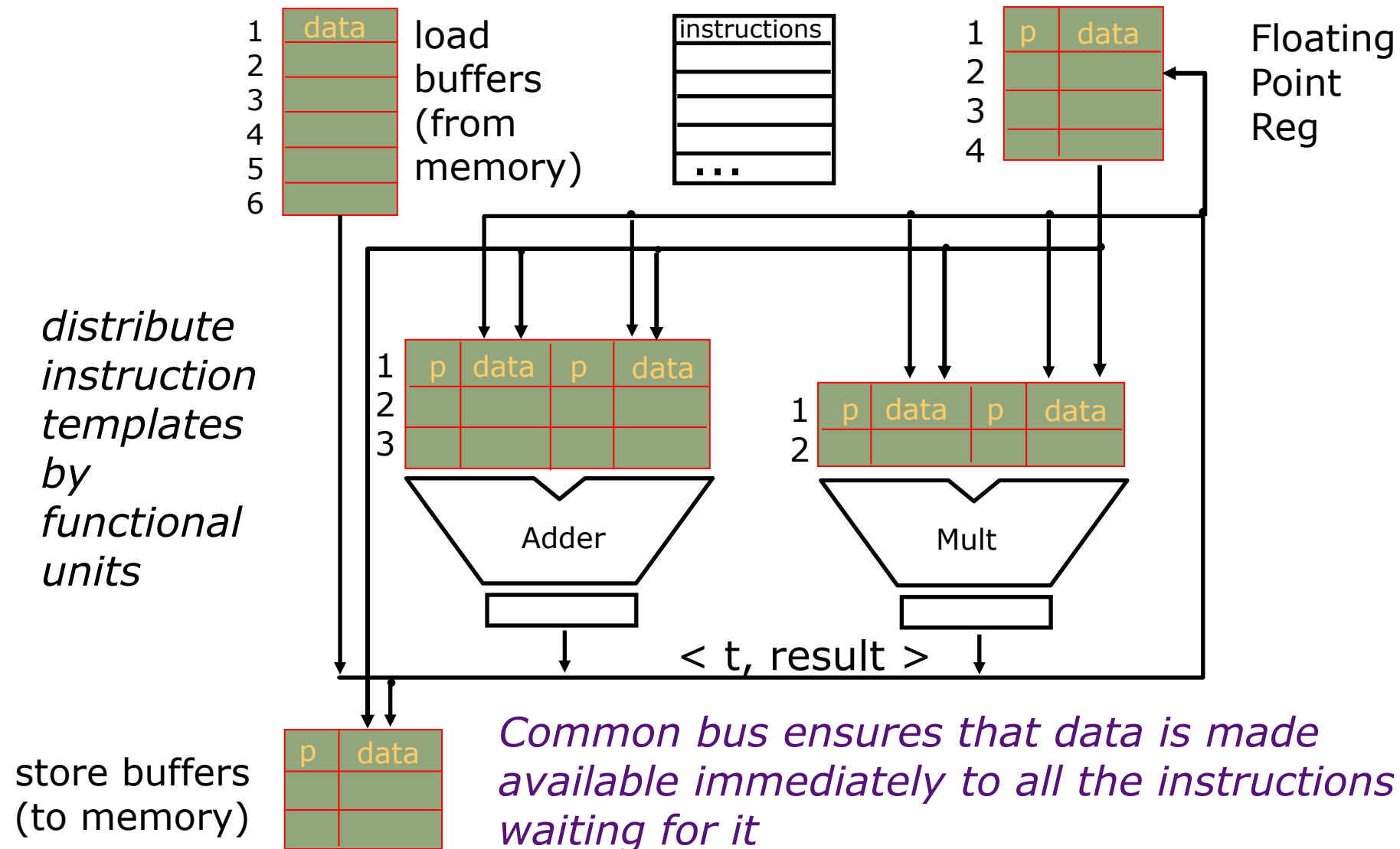
Replacing the
tag by its value
is an expensive
operation



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

IBM 360/91 Floating Point Unit

R. M. Tomasulo, 1967



Effectiveness?

Renaming and Out-of-order execution was first implemented in 1969 in IBM 360/91 but was effective only on a very small class of problems and thus did not show up in the subsequent models until mid-nineties. *Why?*

1. Did not address the memory latency problem which turned out be a much bigger issue than FU latency
2. Made exceptions imprecise

One more problem needed to be solved

Branch/jump penalties

More on this in the next lecture

Reminder: 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})

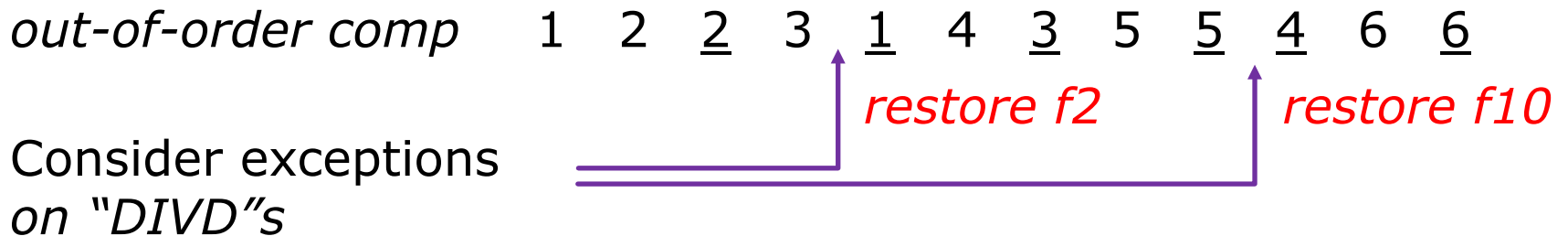
- the effect of all instructions up to and including I_i is complete
- no effect of any instruction after I_i has taken place

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



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

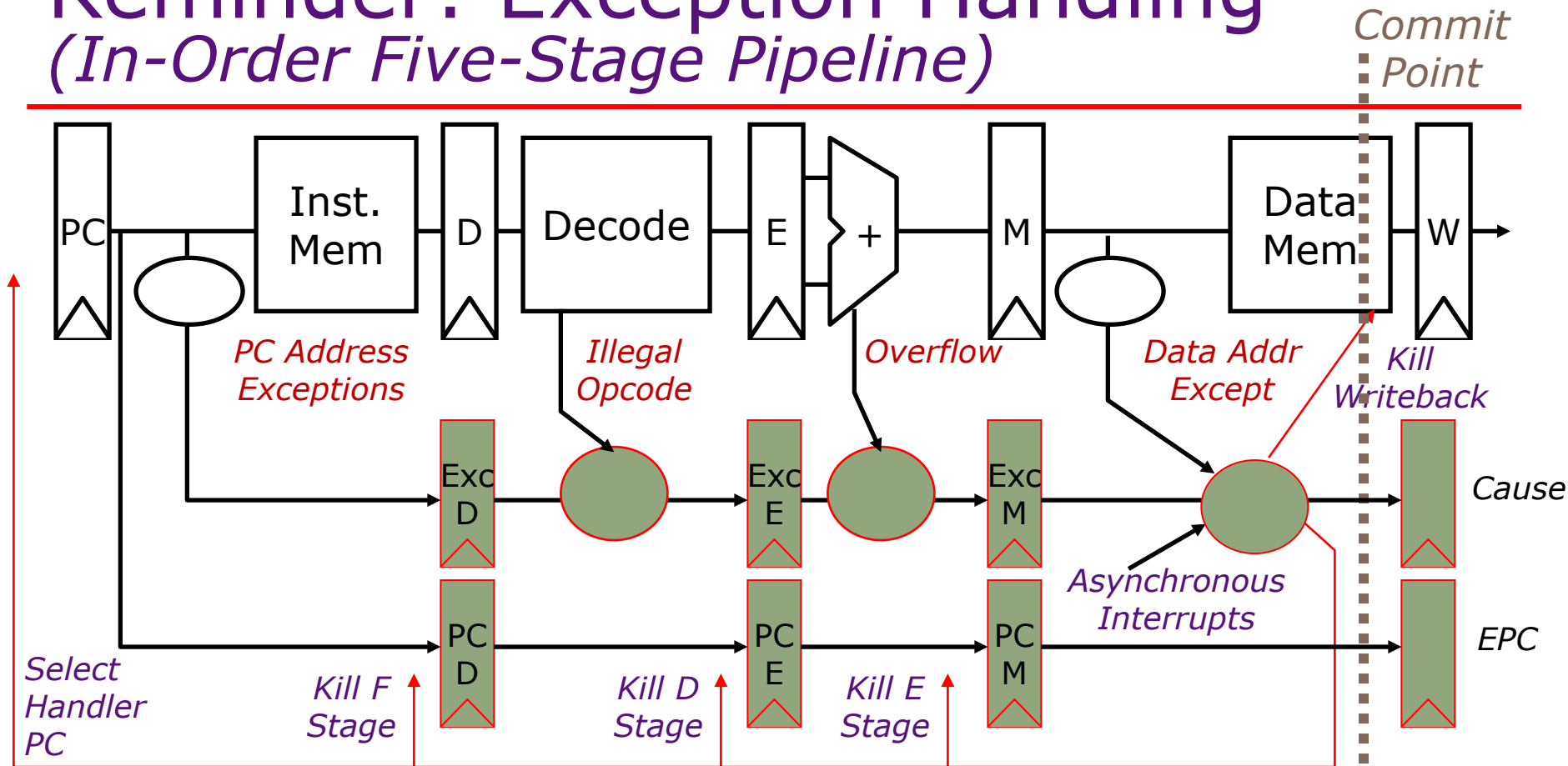
Exceptions

- 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

Reminder: Exception Handling

(In-Order Five-Stage Pipeline)

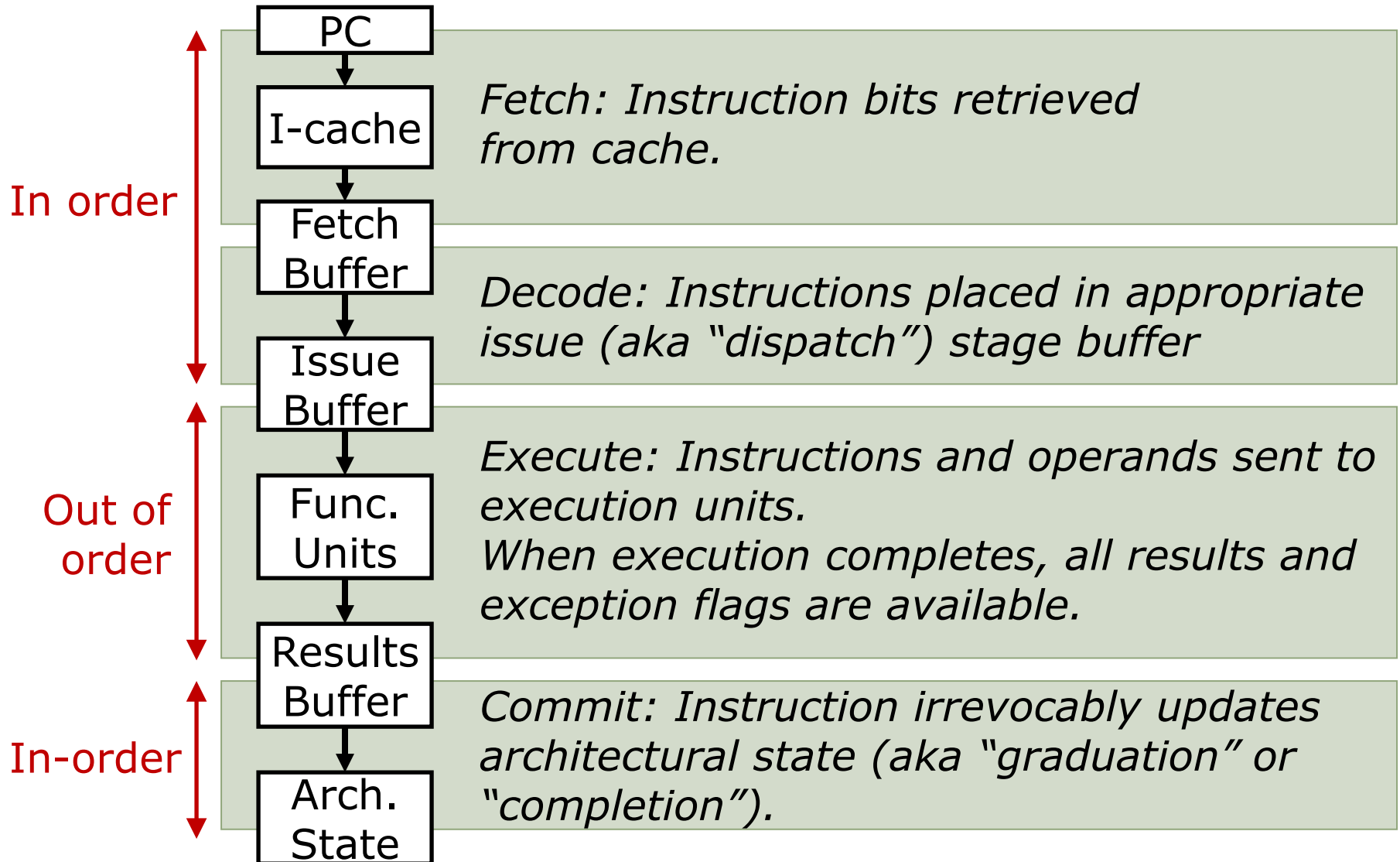


Hold exception flags in pipeline until commit point (M stage)

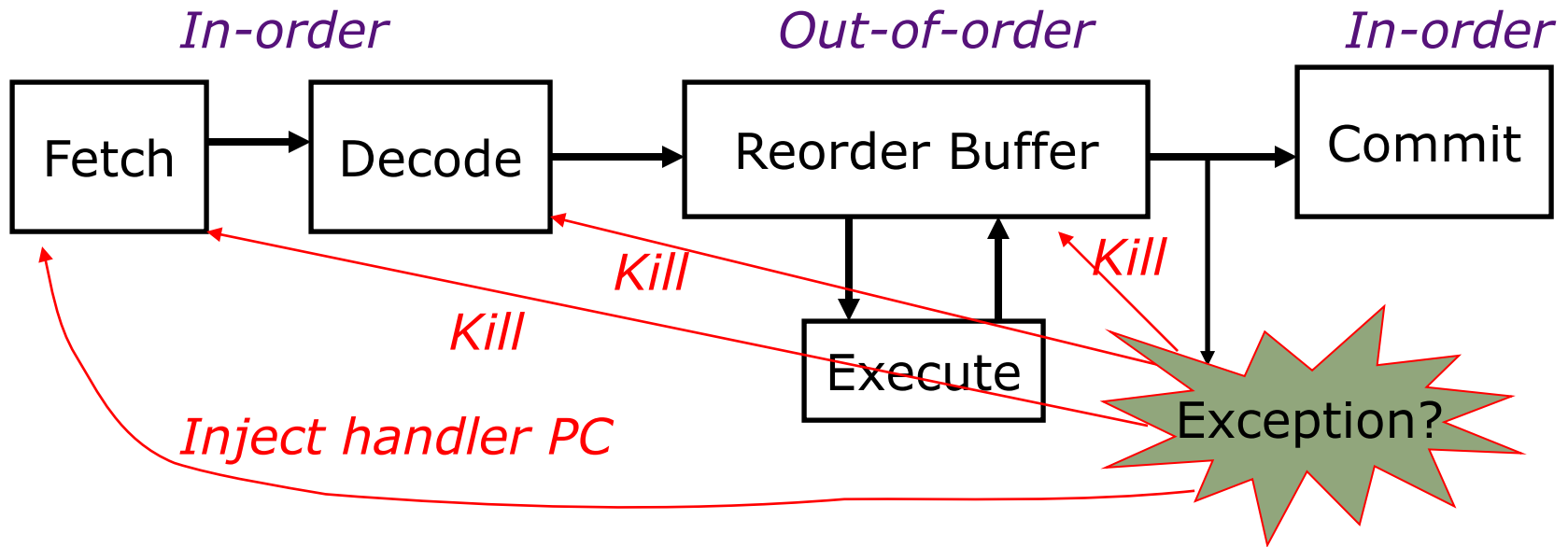
- If exception at commit:
 - update Cause/EPC registers
 - kill all stages
 - fetch at handler PC

Inject external interrupts at commit point

Phases of Instruction Execution



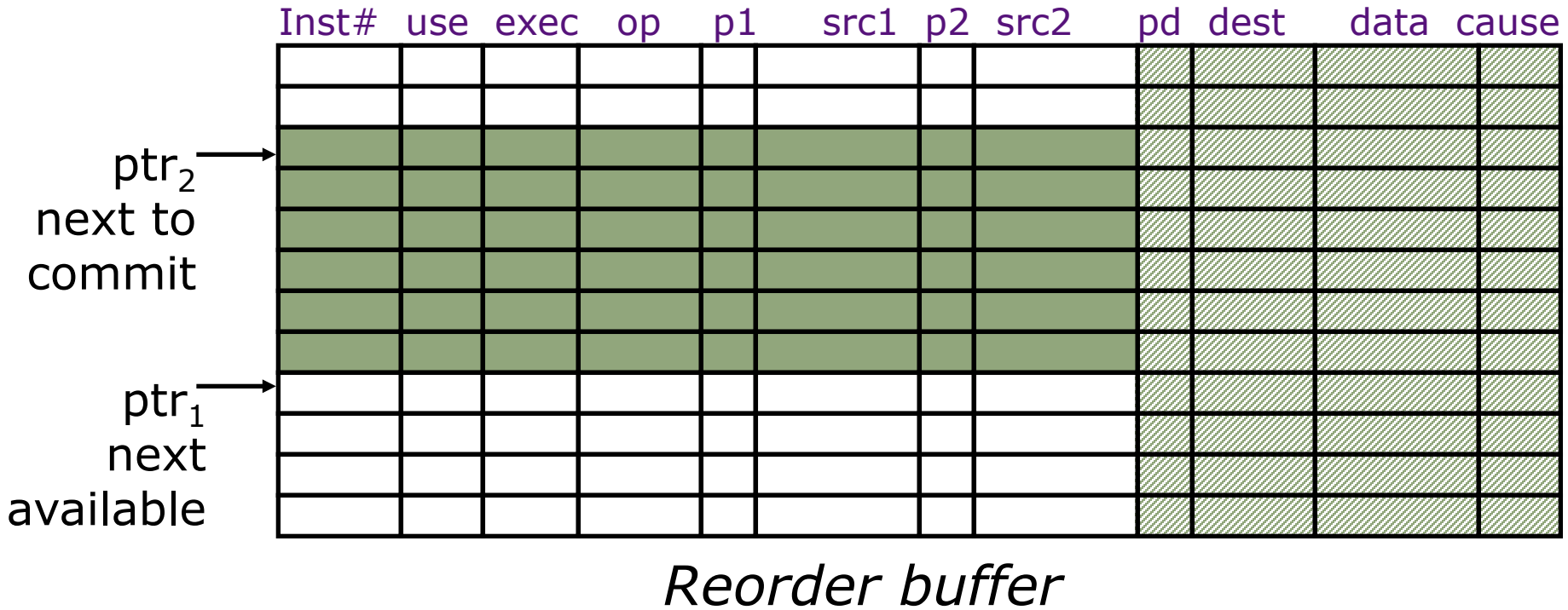
In-Order Commit for Precise Exceptions



- 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

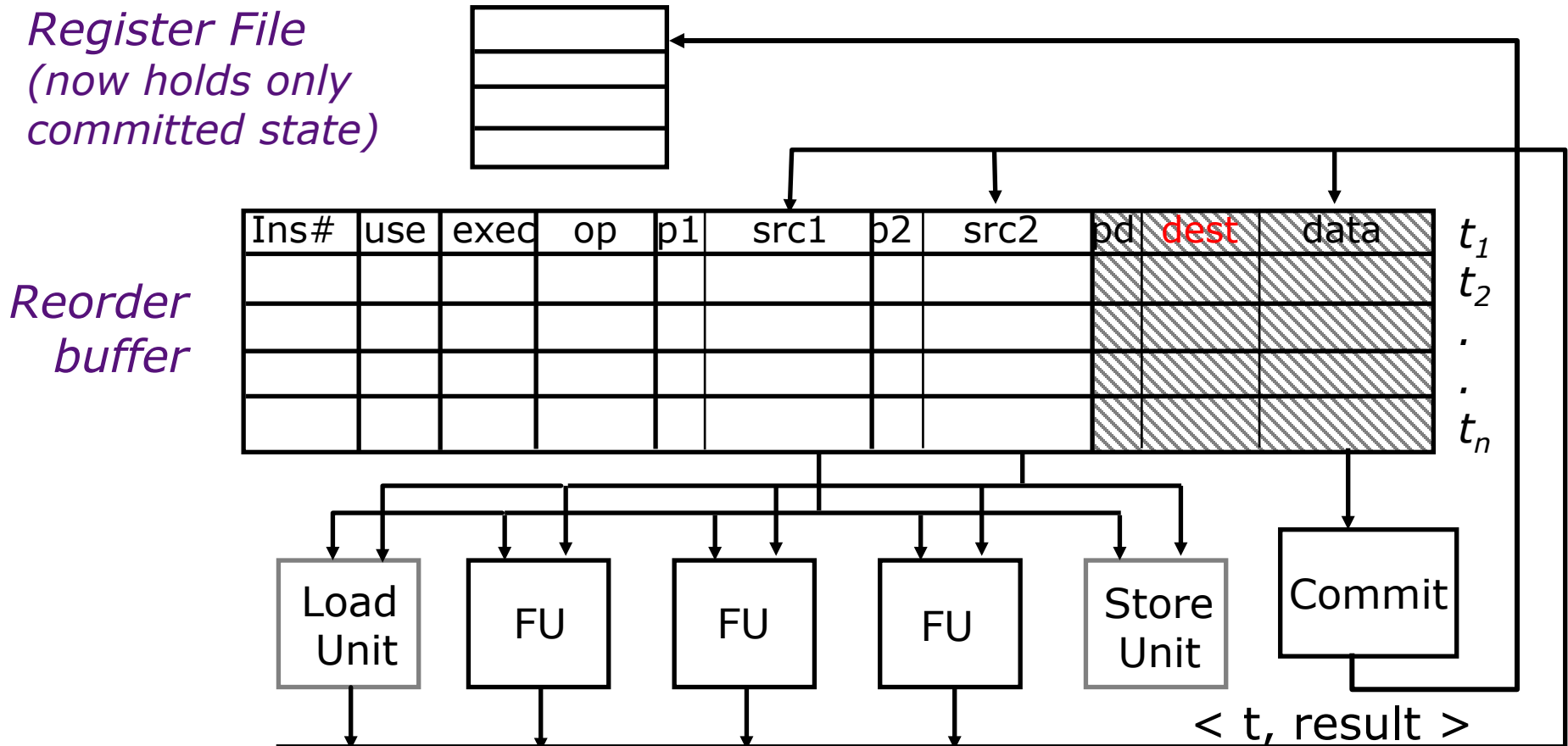
Temporary storage needed to hold results before commit (shadow registers and store buffers)

Extensions for Precise Exceptions



- add <pd, dest, data, cause> fields in the instruction template
- commit instructions to reg file and memory in program order $\Rightarrow$ buffers can be maintained circularly
- on exception, clear reorder buffer by resetting $\text{ptr}_1 = \text{ptr}_2$
(stores must wait for commit before updating memory)

Rollback and Renaming

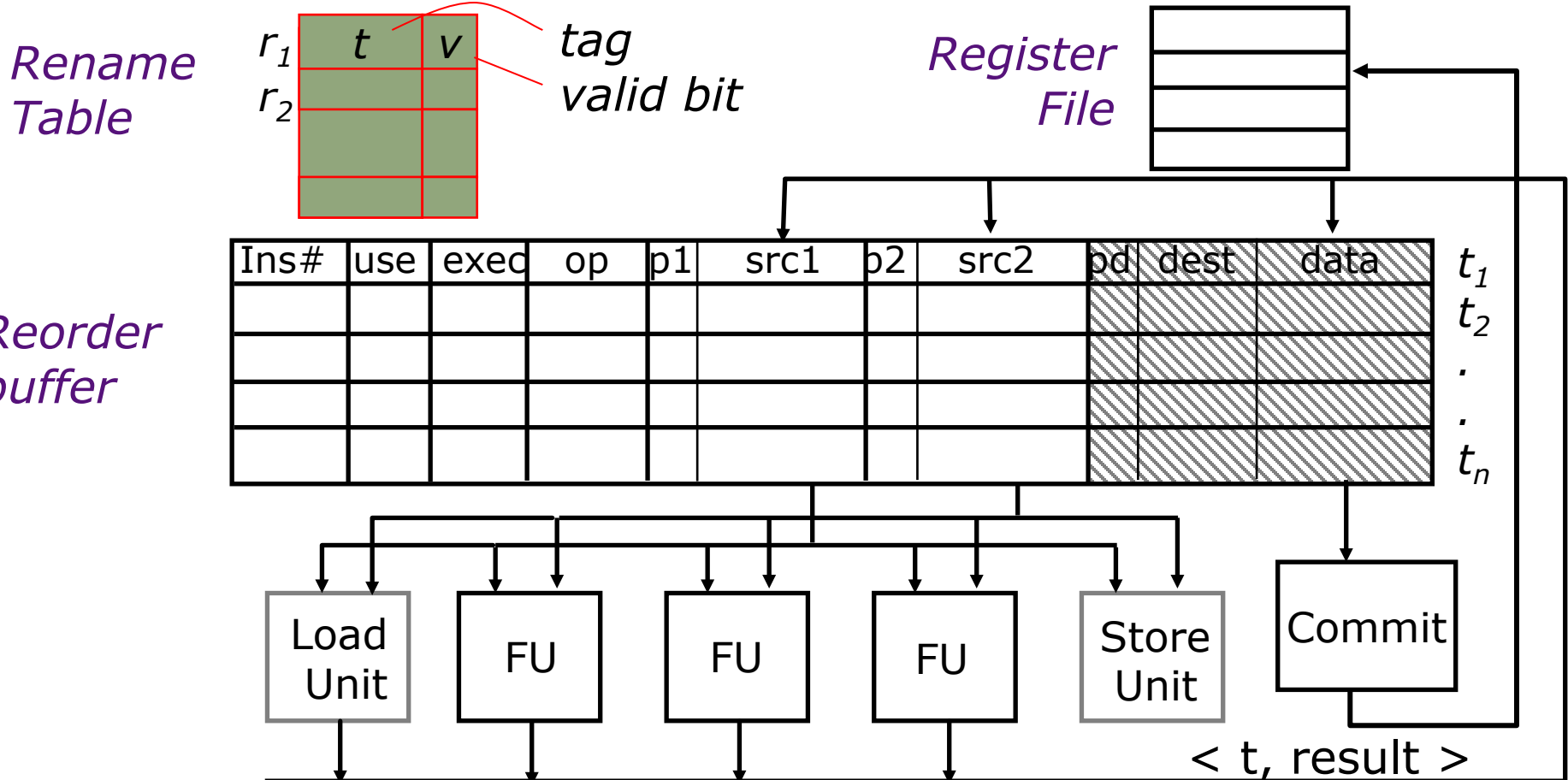


Register file does not contain renaming tags any more.

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-bit 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

