CS211
Advanced Computer Architecture

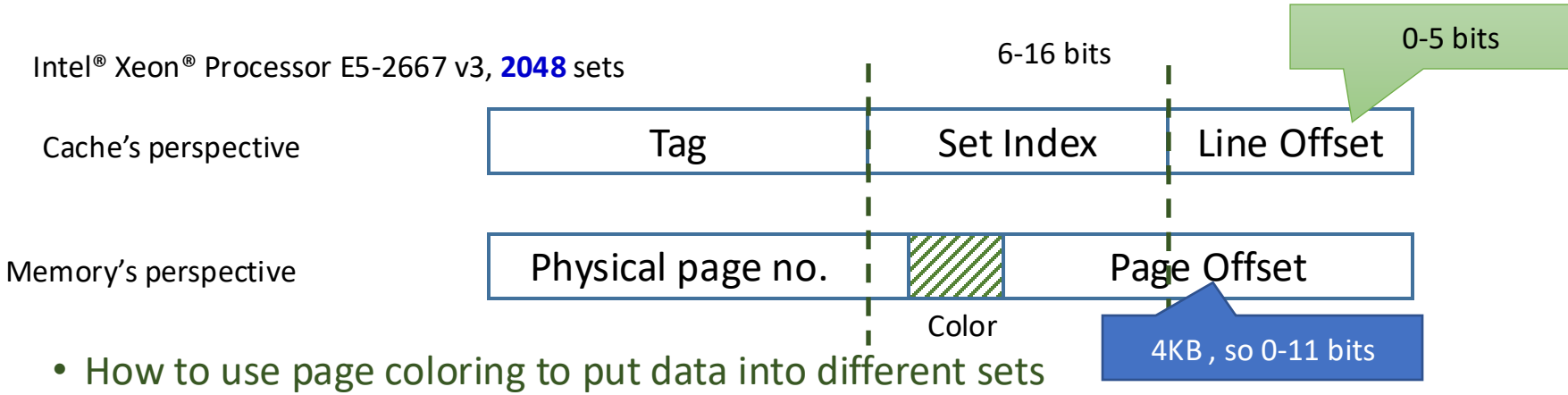
L10 Out of Order Execution

Chundong Wang

October 24th, 2025

Cache coloring for high performance

- Page coloring (cache coloring)
  - Different colors to physical pages
    - Same color → same cache set

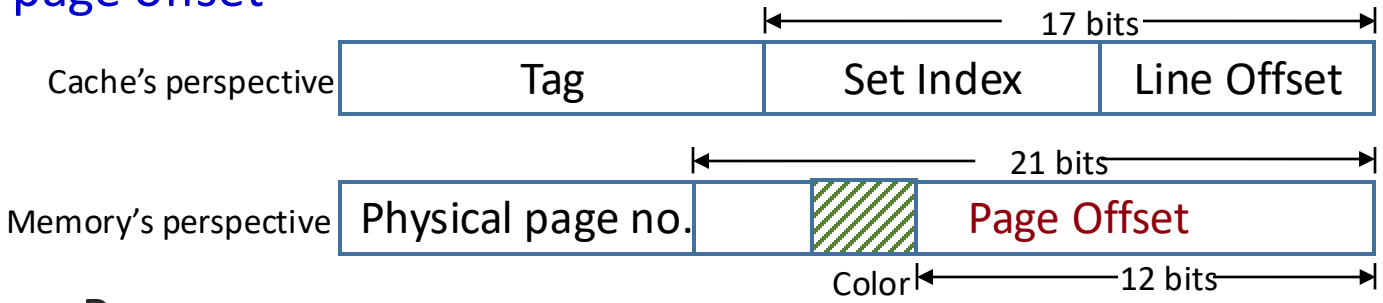


  - How to use page coloring to put data into different sets

Kefei Wang, Jian Liu, and Feng Chen. 2020. Put an elephant into a fridge: optimizing cache efficiency for in-memory key-value stores. Proc. VLDB Endow. 13, 9 (May 2020), 1540–1554. DOI:https://doi.org/10.14778/3397230.3397247

Cache coloring for high performance

- Virtual page no → physical page no
  - In the charge of OS
- In-page offset



- Huge Page
  - 2MB, 1GB, …

With huge page, programmers customize in-page layout
➔data in a page not contending one set, less conflict

Kefei Wang, Jian Liu, and Feng Chen. 2020. Put an elephant into a fridge: optimizing cache efficiency for in-memory key-value stores. Proc. VLDB Endow. 13, 9 (May 2020), 1540–1554. DOI:https://doi.org/10.14778/3397230.3397247

Cache is changing

- ADR → eADR
  - A super-capacitor installed to flush all cache lines to memory (if pesistent) in case of power fail

Taiyu Zhou, Yajuan Du, Fan Yang, Xiaojian Liao, and Youyou Lu. 2023. Efficient Atomic Durability on eADR-Enabled Persistent Memory. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '22). Association for Computing Machinery, New York, NY, USA, 124–134. https://doi.org/10.1145/3559009.3569676

Chongnan Ye, Meng Chen, Qisheng Jiang, and Chundong Wang. 2023. Hercules: Enabling Atomic Durability for Persistent Memory with Transient Persistence Domain. ACM Trans. Embed. Comput. Syst. Just Accepted (July 2023). https://doi.org/10.1145/3607473

Types of Data Hazards

Consider executing a sequence of

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

type of instructions

Data-dependence

$r_3 \leftarrow r_1$ op r_2
$r_5 \leftarrow r_3$ op r_4

Read-after-Write (RAW) hazard

Anti-dependence

$r_3 \leftarrow r_1$ op r_2
$r_1 \leftarrow r_4$ op r_5

Write-after-Read (WAR) hazard

Output-dependence

$r_3 \leftarrow r_1$ op r_2
$r_3 \leftarrow r_6$ op r_7

Write-after-Write (WAW) hazard

Register vs. Memory Dependence

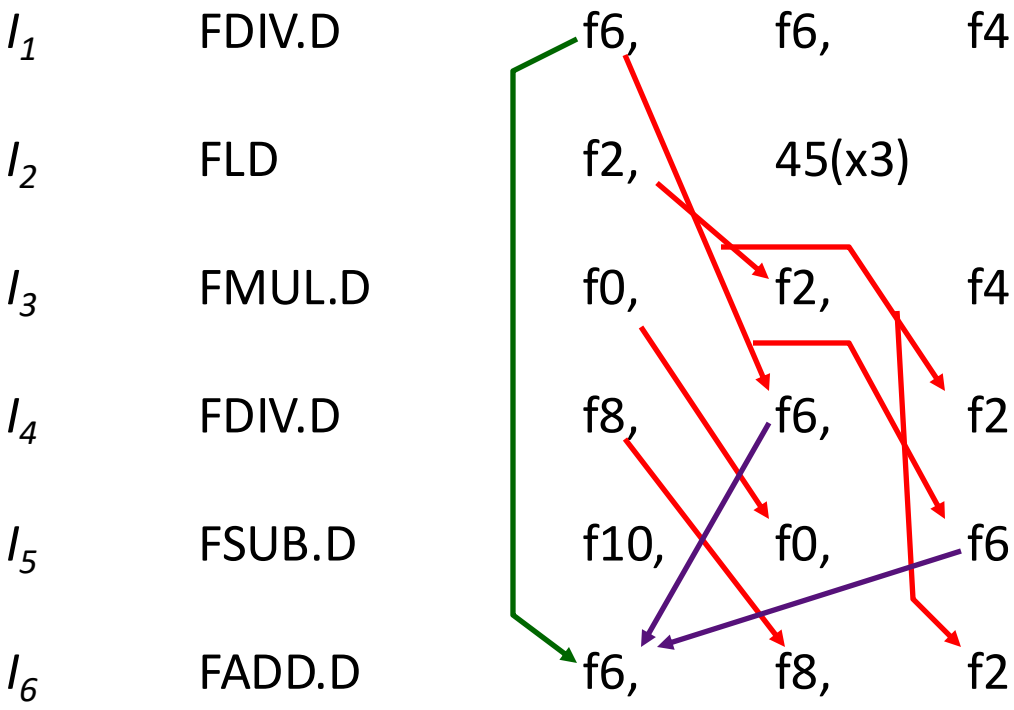
Data hazards due to register operands can be determined at the decode stage, but data hazards due to memory operands can be determined only after computing the effective address

Store: **M[r1 + disp1] ← r2**

Load: **r3 ← M[r4 + disp2]**

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

Data Hazards: An Example



RAW Hazards
WAR Hazards
WAW Hazards

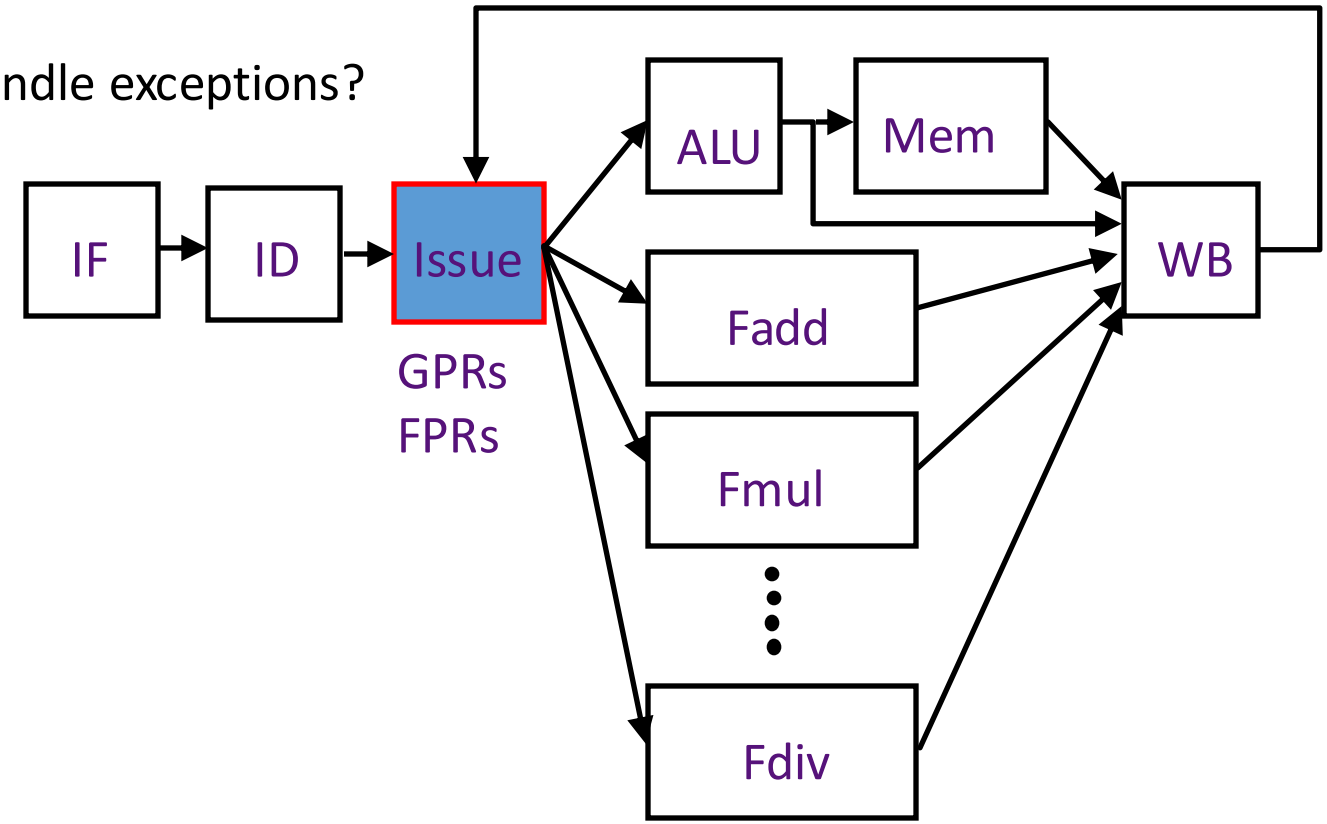
Complex Pipelining: Motivation

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

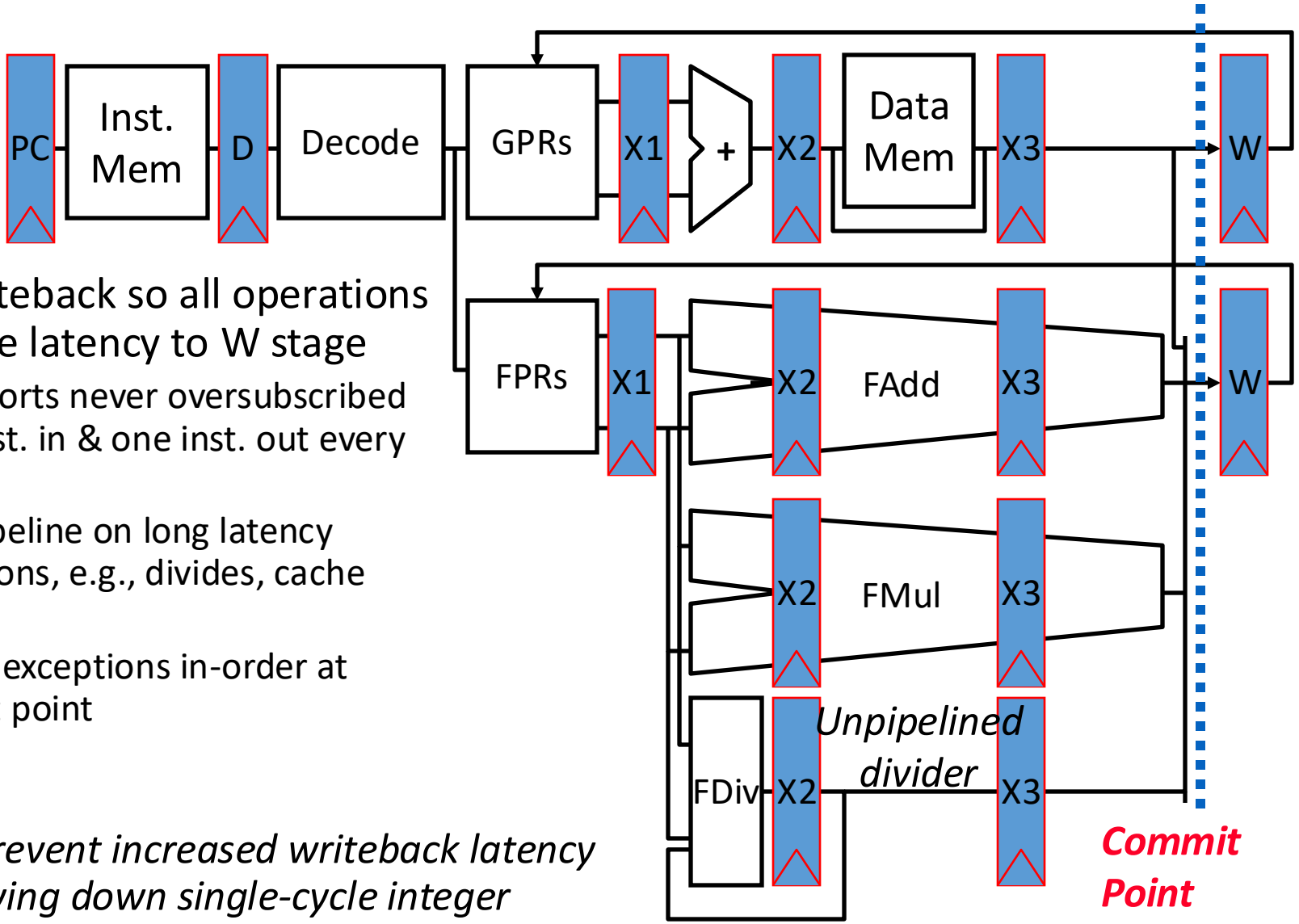
- Long latency or partially pipelined floating-point units
- Memory systems with variable access time
- Multiple arithmetic and memory units

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?



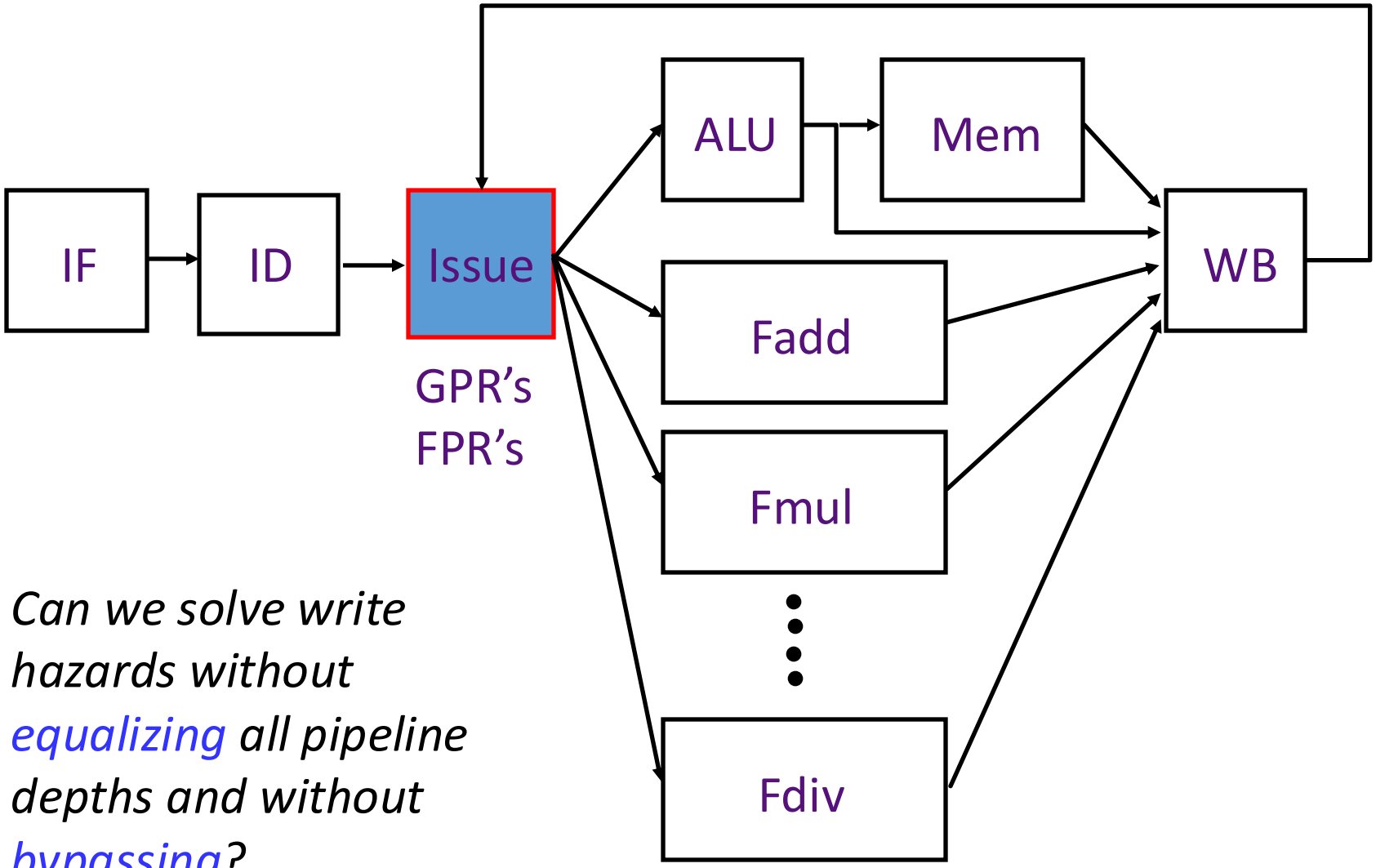
Recap: Complex In-Order Pipeline



- 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?

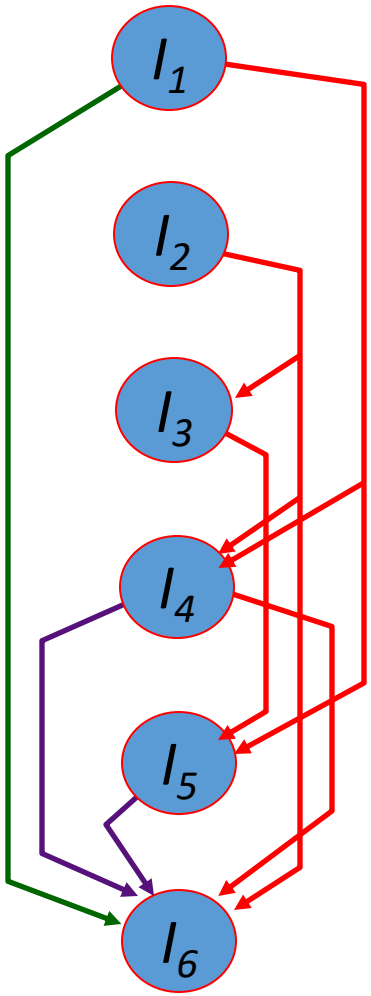
Complex Pipeline



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

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

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 ☕ ☕ 1 2 3 4 ☕ 3 5 4 6 5 6

out-of-order comp 1 2 2 3 1 4 3 5 5 4 6 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 functional unit (FU) is busy, and that operands are latched by FU on issue:

Can the dispatched instruction cause a

WAR hazard ?

NO: Operands read at issue

WAW hazard ?

YES: Out-of-order completion

Anti-dependence

$r_3 \leftarrow r_1$ op r_2

$r_3 \leftarrow r_4$ op r_5

Write-after-Write (WAW) hazard

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

上海科技大学 ShanghaiTech University

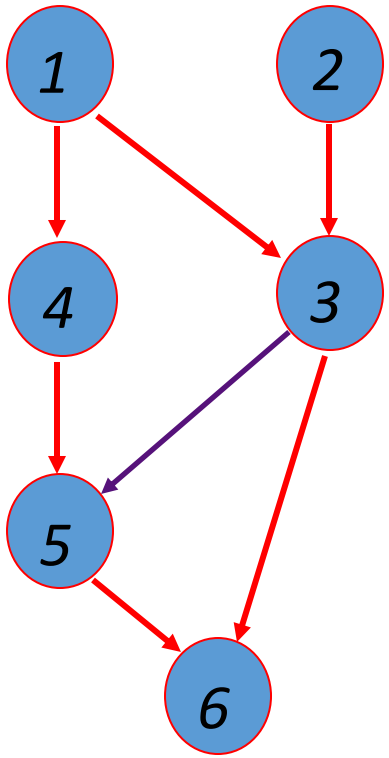
	Functional Unit Status										Registers Reserved for Writes	
	Int(1)		Add(1)	Mult(3)			Div(4)			WB		
t0	I_1						f6				f6	
t1	I_2	f2						f6			f6, f2	
t2									f6	f2	f6, f2	I_2
t3	I_3			f0						f6	f6, f0	
t4					f0					f6	f6, f0	I_1
t5	I_4					f0	f8				f0, f8	
t6								f8		f0	f0, f8	I_3
t7	I_5	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

In-Order Issue Limitations: *an example*

上海科技大学 ShanghaiTech University

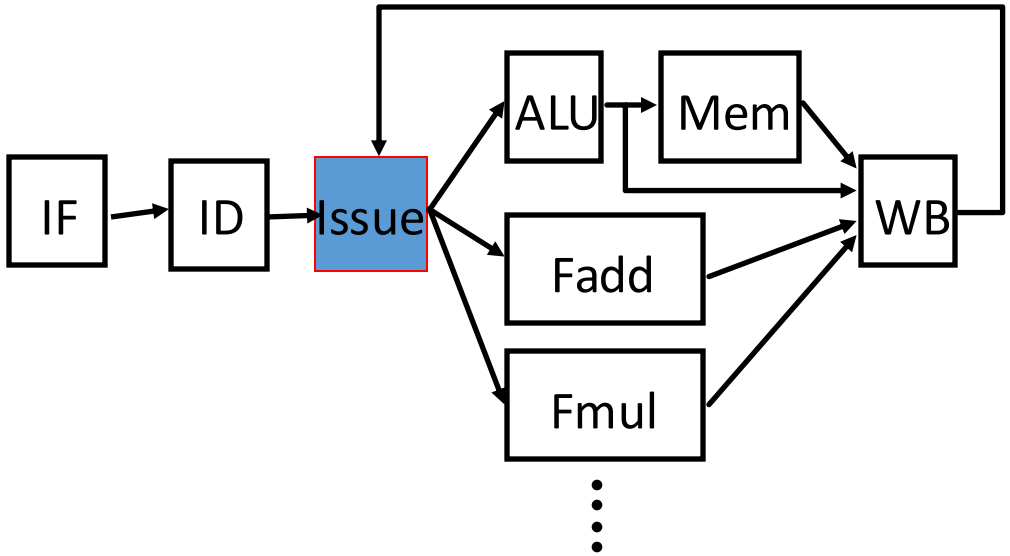
					latency
1	FLD	f2,	34(x2)		*1*
2	FLD	f4,	45(x3)		*long*
3	FMULT.D	f6,	f4,	f2	*3*
4	FSUB.D	f8,	f2,	f2	*1*
5	FDIV.D	f4,	f2,	f8	*4*
6	FADD.D	f10,	f6,	f4	*1*



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

In-order issue restriction prevents instruction 4 from being dispatched

Out-of-Order Issue

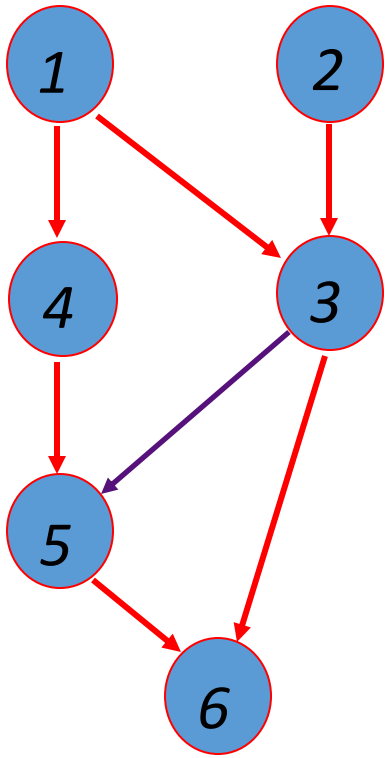


- Issue stage buffer holds multiple instructions waiting to issue.
- Decode adds next instruction to buffer if there is space and the instruction does not cause a WAR or WAW hazard.
  - Note: WAR possible again because issue is out-of-order (WAR not possible with in-order issue and latching of input 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.

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

上海科技大学 ShanghaiTech University

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

Out-of-order execution did not allow any significant improvement!

How many instructions can be in the pipeline?

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

Number of Registers

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.

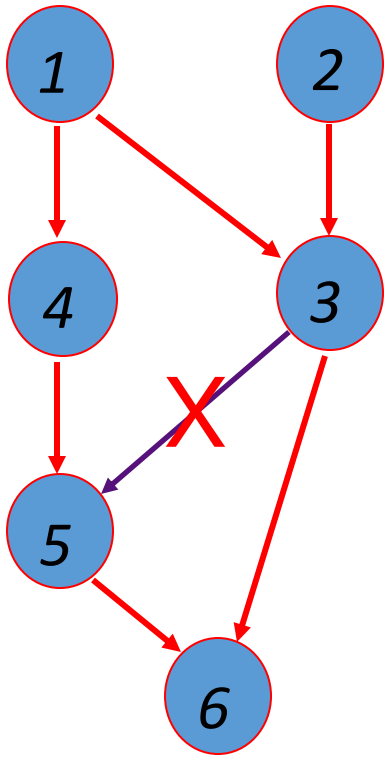
IBM 360 had only 4 floating-point registers

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 using on-the-fly *register renaming*

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

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



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

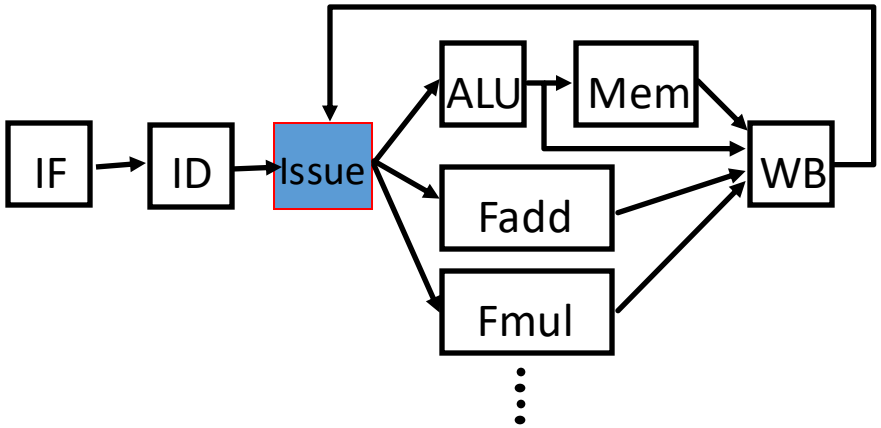
Out-of-order: 1 (2,1) 4 4 5 . . . 2 (3,5) 3 6 6

Any antidependence can be eliminated by renaming.

(renaming ➔ additional storage)

Can it be done in hardware? yes!

Register Renaming



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

  ➔ renaming makes WAR or WAW hazards impossible

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

  ➔ Out-of-order or dataflow execution

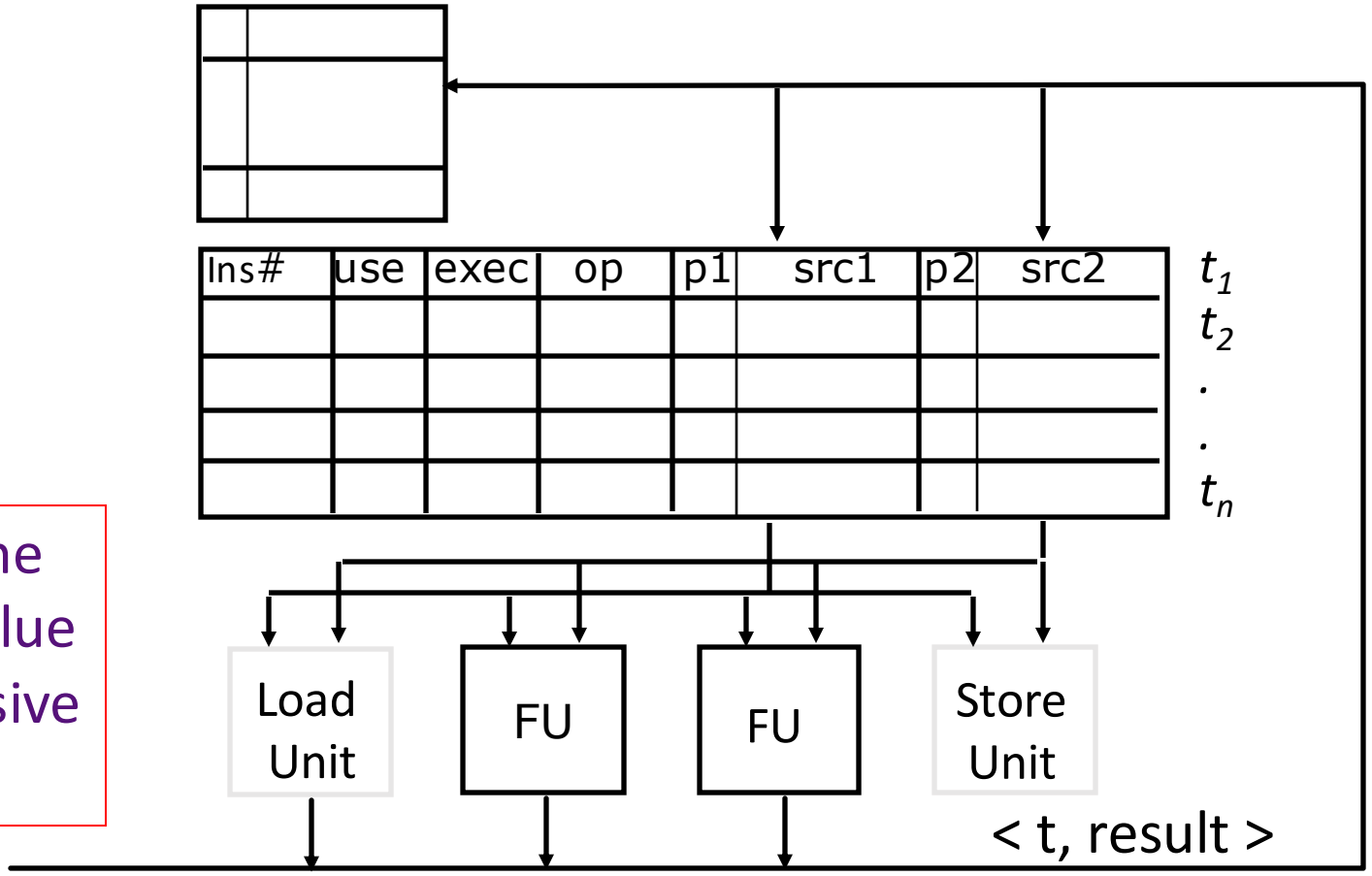
Renaming Structures

上海科技大学 ShanghaiTech University

Renaming table & regfile

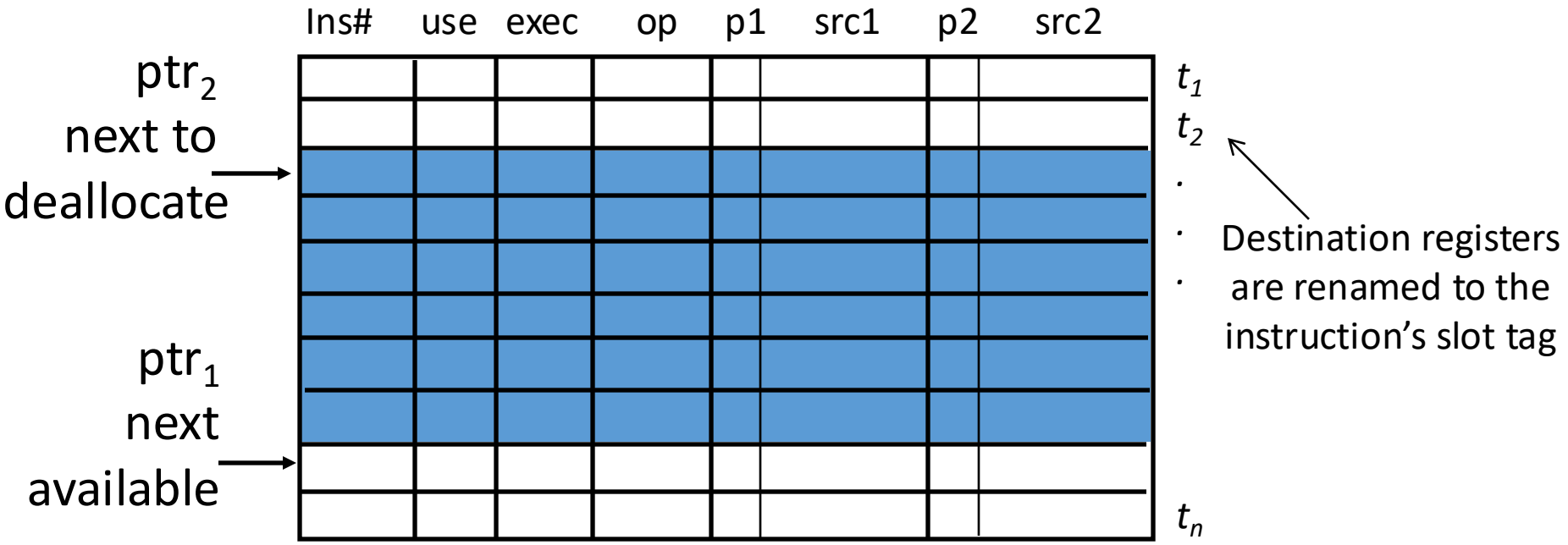
Reorder buffer

Replacing the tag by its value is an expensive operation



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

Reorder Buffer Management



Instruction slot is candidate for execution when:

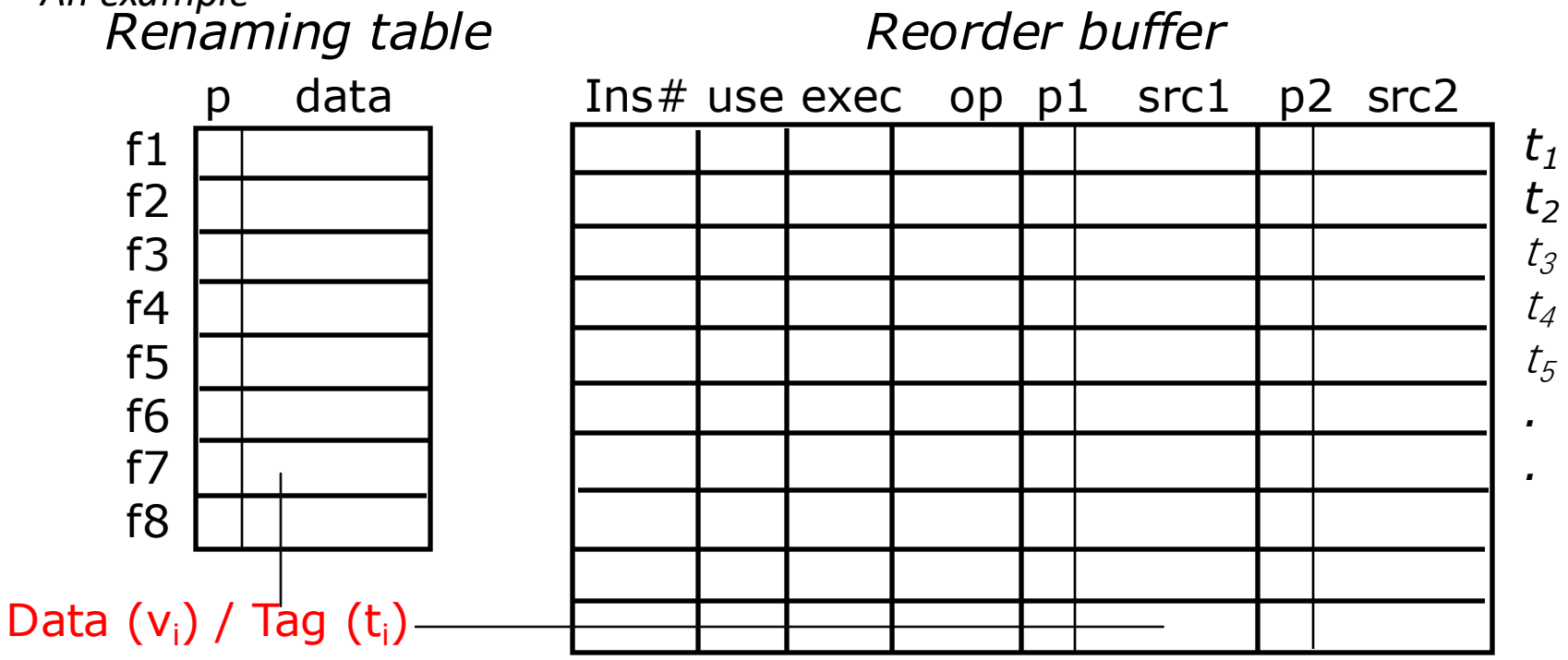
- It holds a valid instruction ("use" bit is set)
- It has not already started execution ("exec" bit is clear)
- Both operands are available (p1 and p2 are set)

Is it obvious where an architectural register value is?

No

Renaming & Out-of-order Issue

An example



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

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

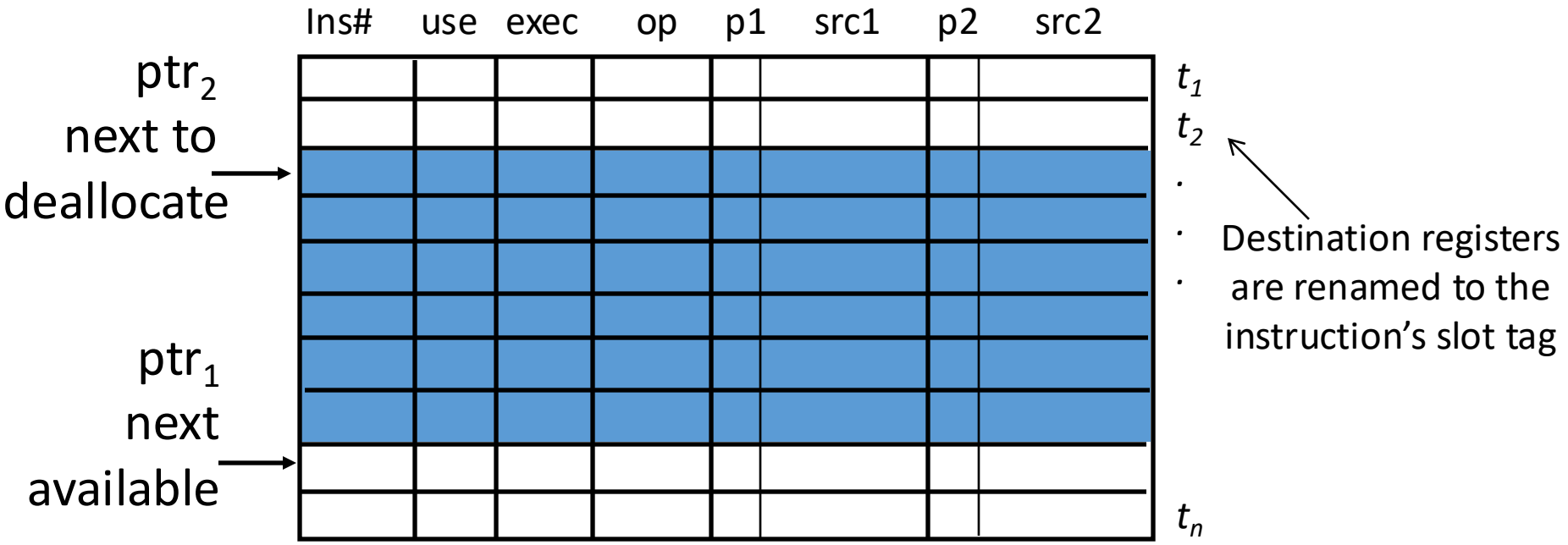
	Ins#	use	exec	op	p1	src1	p2	src2	
	1	0	0	LD					t_1
	2	0	0	LD					t_2
	3	1	0	MUL	0	v2	1	v1	t_3
	4	0	0	SUB	1	v1	1	v1	t_4
	5	1	0	DIV	1	v1	0	v4	t_5
									.
									.

Data (v_i) / Tag (t_i)

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

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

Reorder Buffer Management

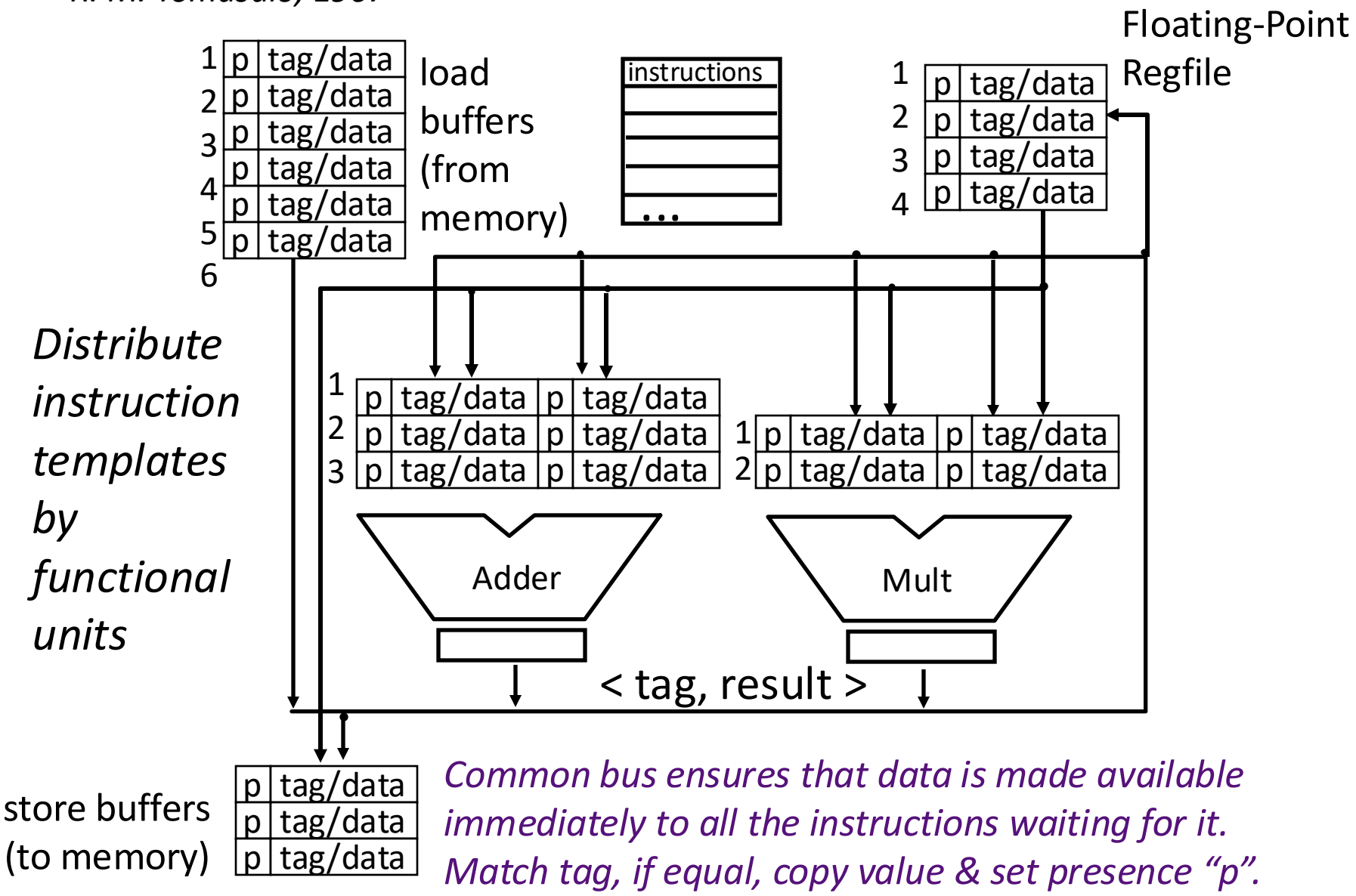


ROB managed circularly

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

IBM 360/91 Floating-Point Unit

R. M. Tomasulo, 1967



Out-of-Order Fades into Background

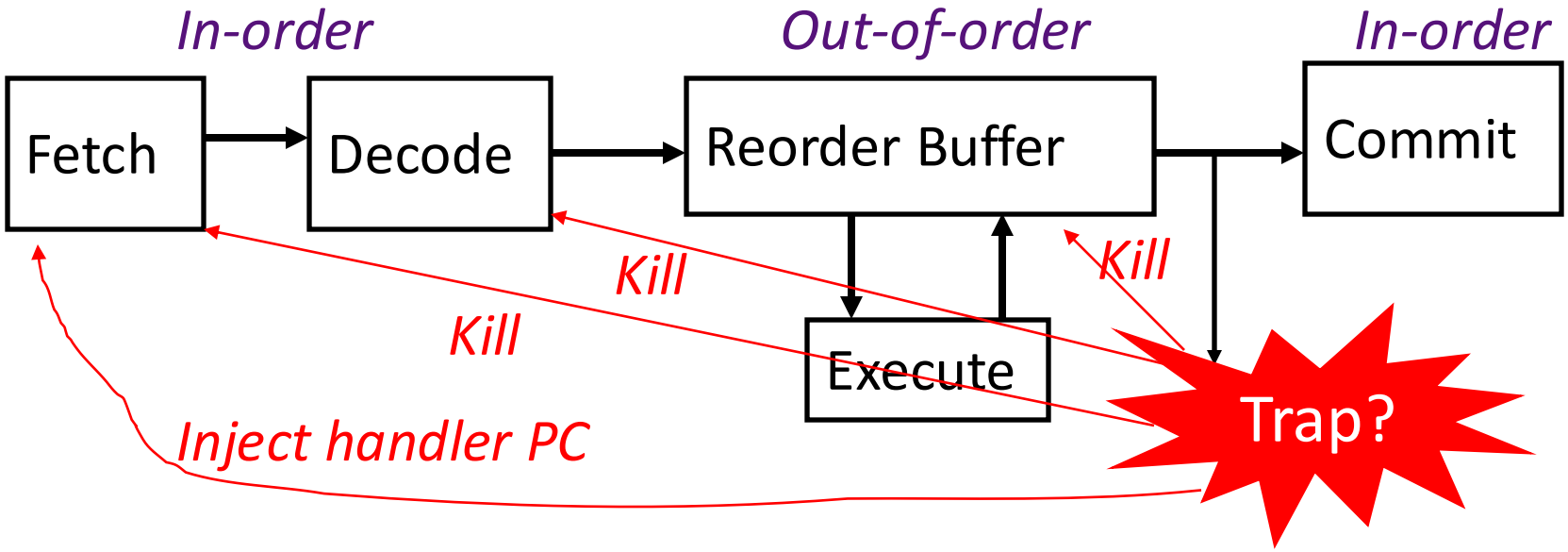
Renaming and Out-of-order execution was first implemented in 1969 in IBM 360/91 but was effective only on a very small class of problems and thus did not show up in the subsequent models until mid-nineties.

- Did not address the memory latency problem which turned out be a much bigger issue than FU latency
- 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

In-Order Commit for Precise Traps

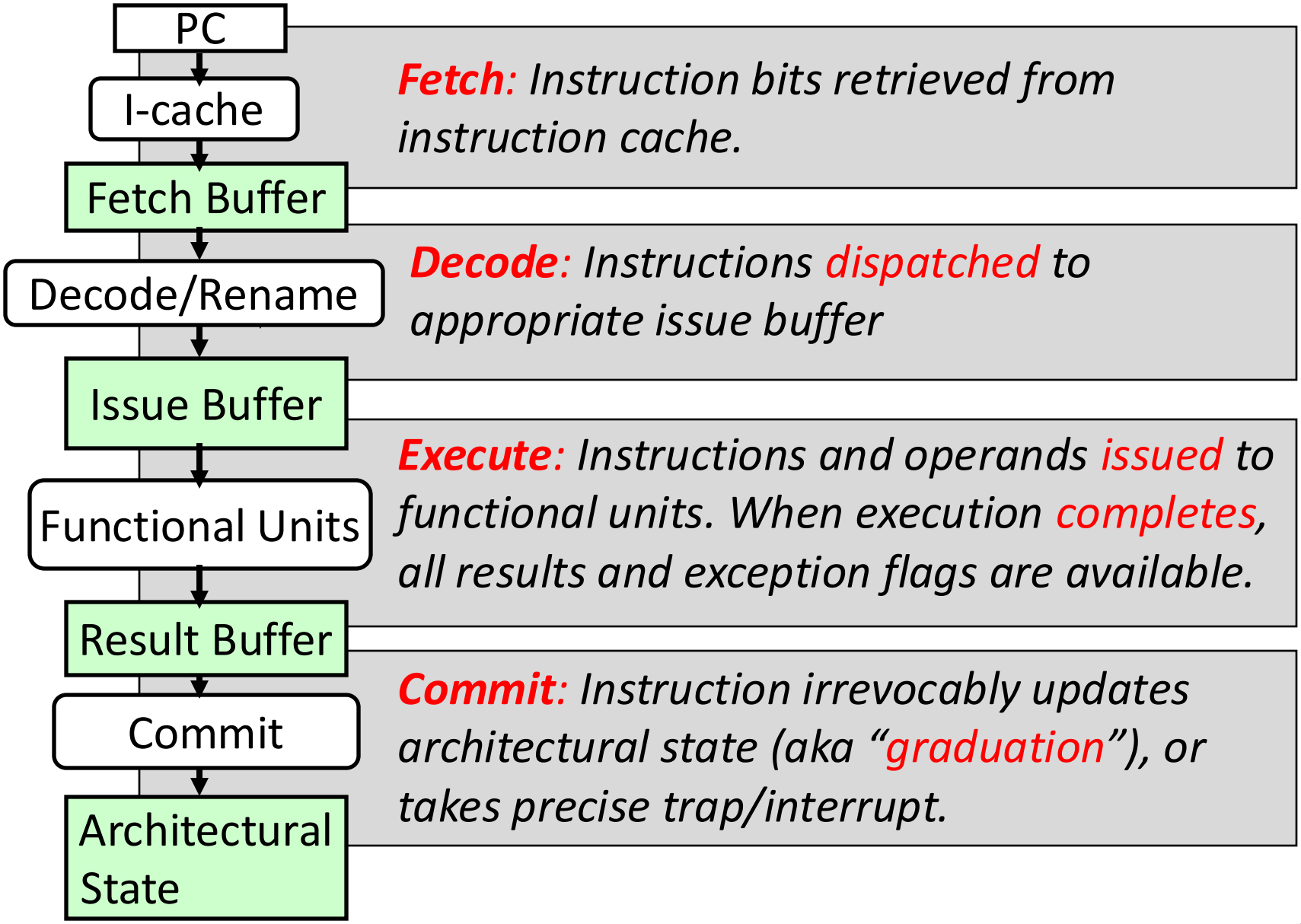


- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order (➔ out-of-order completion)
- Commit (write-back to architectural state, i.e., regfile & memory) is in-order
- *Temporary storage* needed to hold results before commit (shadow registers and store buffers)

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

Phases of Instruction Execution



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 CS211 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

Conclusion

- In-order completion
- Out-of-order completion

- In-order issue
- Out-of-order issue

- In-order commit
- Out-of-order commit

Acknowledgements

- These slides contain materials developed and copyright by:
  - Prof. Krste Asanovic (UC Berkeley)
  - Prof. Yonghong Yan (UNC Charlotte)
  - Prof. Daniel Sanchez (MIT)