CS 152 Computer Architecture and Engineering
CS252 Graduate Computer Architecture

Lecture 10 – Complex Pipelines, Out-of-Order Issue, Register Renaming

John Wawrzynek
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~johnw
http://inst.eecs.berkeley.edu/~cs152

Last time in Lecture 9

- Modern page-based virtual memory systems provide:
  - Translation, Protection, independent or shared address spaces, decoupling size of physical memory from process address space size.
- Translation and protection information stored in page tables, held in main memory
- Translation and protection information cached in "translation-lookaside buffer" (TLB) to provide single-cycle translation+protection check in common case
- Virtual memory interacts with cache design
  - Physical cache tags require address translation before tag lookup, or use untranslated offset bits to index cache.
  - Virtual tags do not require translation before cache hit/miss determination, but need to be flushed or extended with ASID to cope with context swaps. Also, must deal with virtual address aliases (usually by disallowing copies in cache).

CS152 Administrivia

- PS2 due today Feb 17
- Lab2 due on Tuesday March 1

- Midterm in class time slot next Tuesday Feb 22
  - In-person in 306 Soda Hall
  - Covers lectures 1 – 9, plus assigned problem sets, labs, book readings
  - closed book/notes
  - no cheat-sheet, no calculators

CS252 Administrivia

- Project Proposal due today
- Proposal should be one page PDF including:
  - Title
  - Team member names
  - What are you trying to do?
  - How is it done today?
  - What is your idea for improvement and why do you think you'll be successful
  - What infrastructure are you going to use for your project?
  - Project timeline with milestones
- Mail PDF of proposal to instructors

Complex Pipelining: Motivation

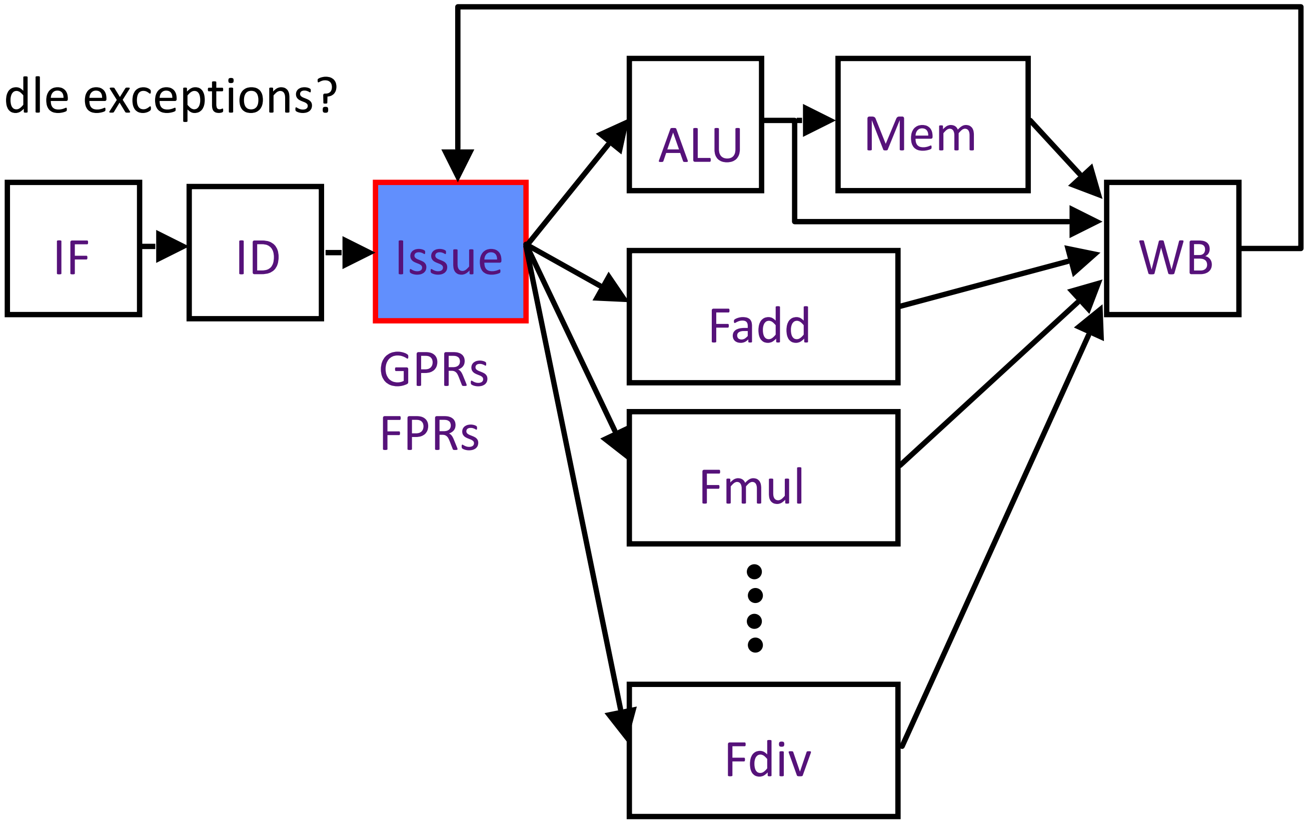
Pipelined instruction execution improves performance. Generally, instruction level parallelism (ILP) is exploited to boost performance.

Pipelining becomes complex in the presence of:

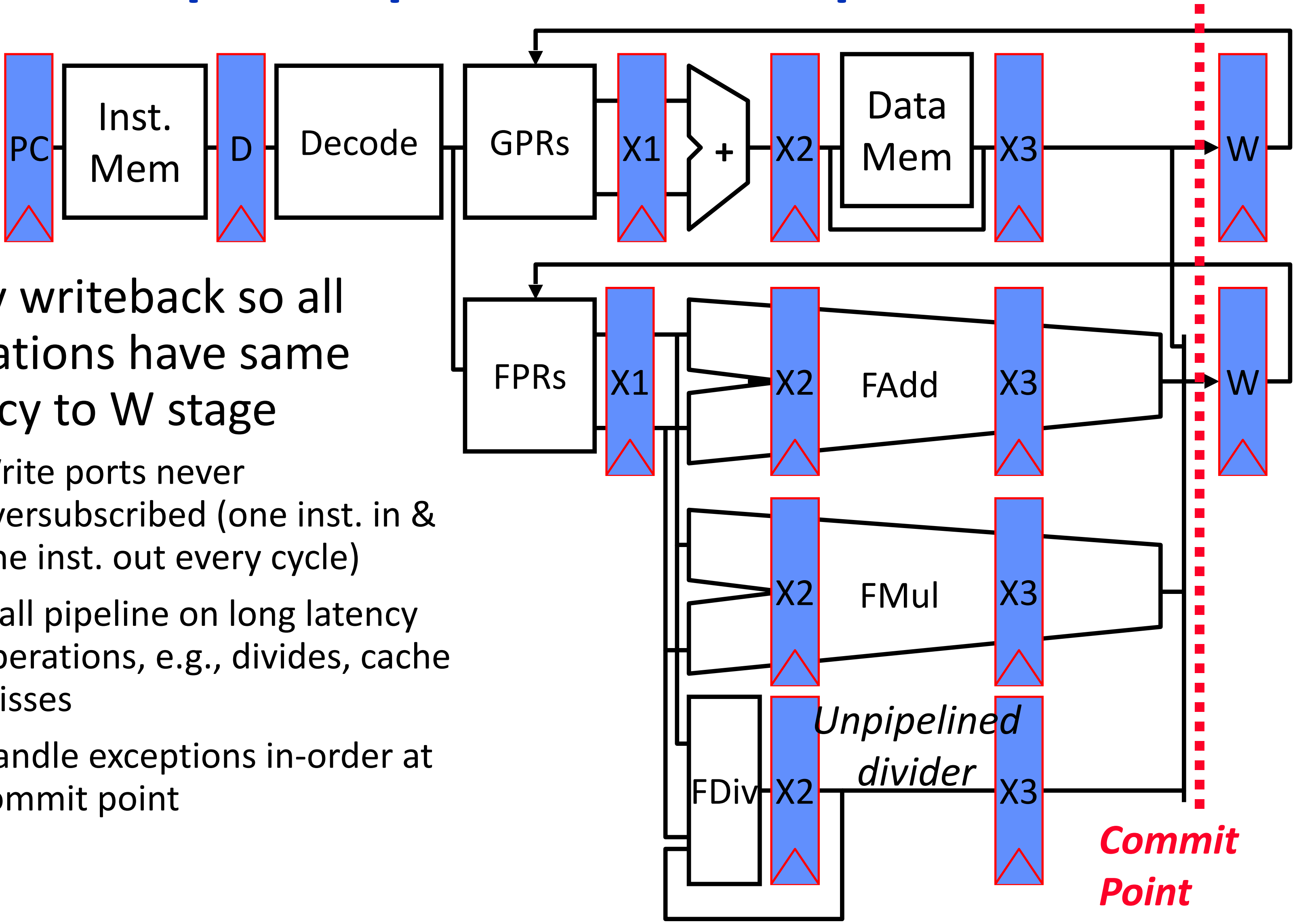
- Long latency or partially pipelined floating-point units
- Memory systems with variable access time
- Multiple arithmetic and memory units

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?

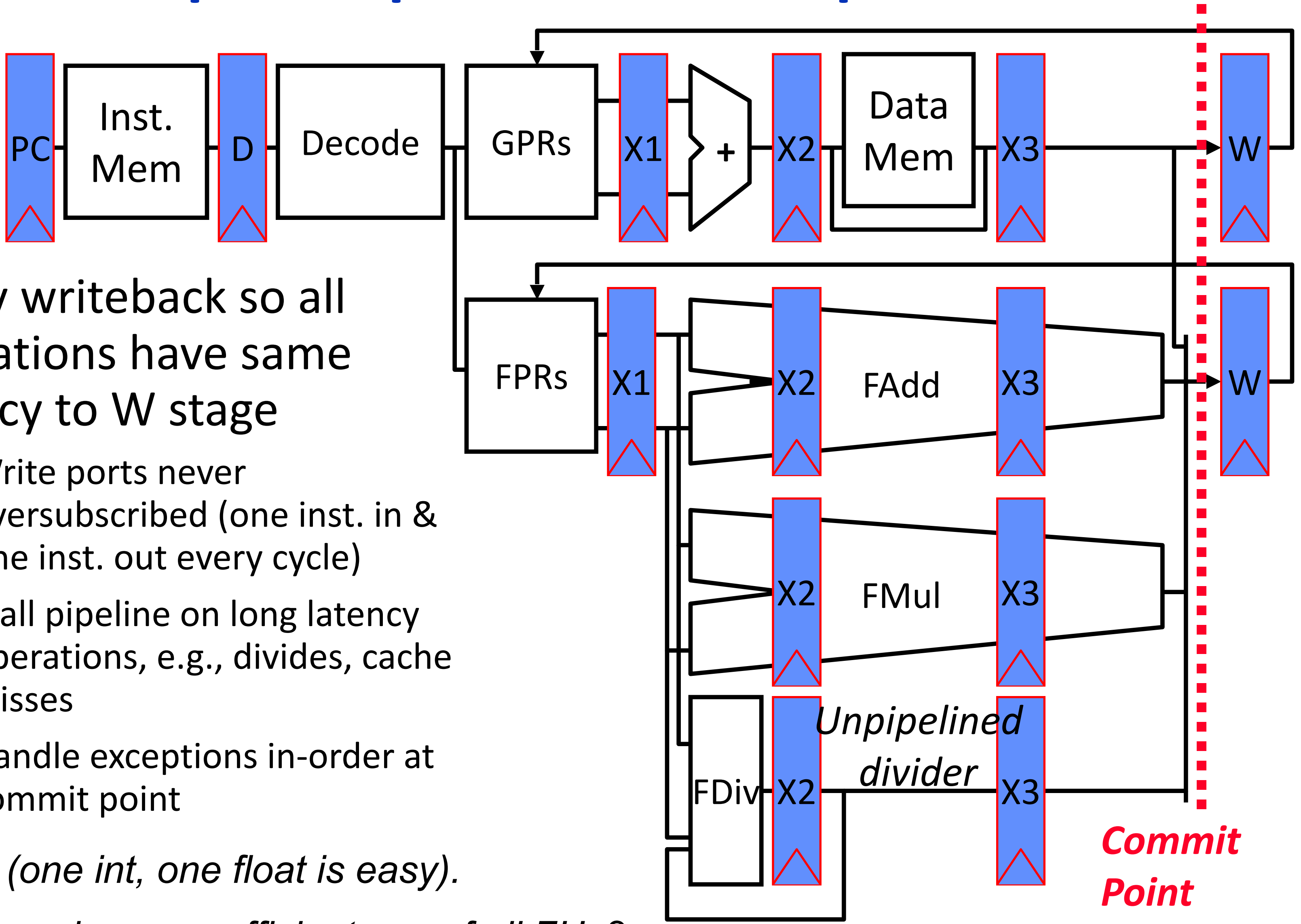


Recap: Complex In-Order Pipeline



- 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

Recap: Complex In-Order Pipeline



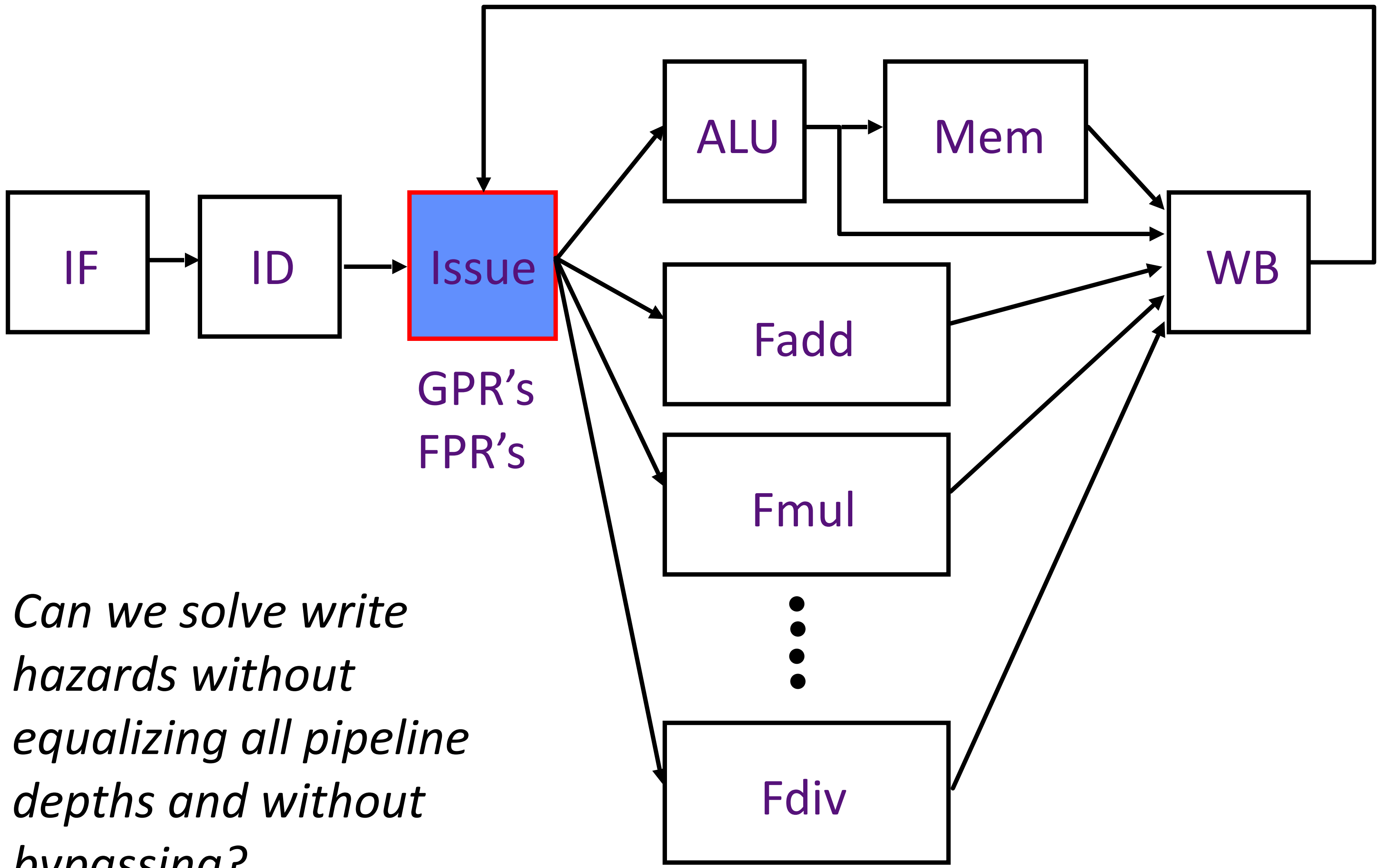
- Delay writeback so all operations have same latency to W stage
  - Write ports never oversubscribed (one inst. in & one inst. out every cycle)
  - Stall pipeline on long latency operations, e.g., divides, cache misses
  - Handle exceptions in-order at commit point

Dual issue (one int, one float is easy).

How do we make more efficient use of all FUs?

Also …

Complex Pipeline



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

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

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

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

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

Data Hazards: An Example

```
I1    FDIV.D    f6,    f6,       f4
I2    FLD       f2,    45(x3)
I3    FMUL.D    f0,    f2,       f4
I4    FDIV.D    f8,    f6,       f2
I5    FSUB.D    f10,   f0,       f6
I6    FADD.D    f6,    f8,       f2
```

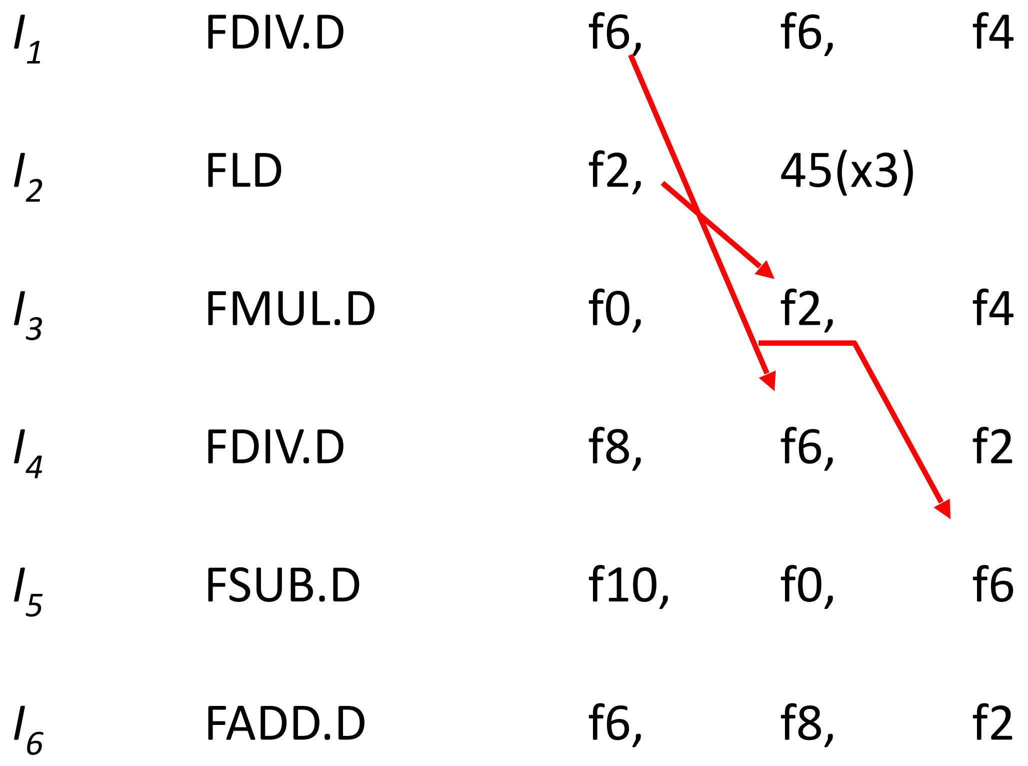
RAW Hazards

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

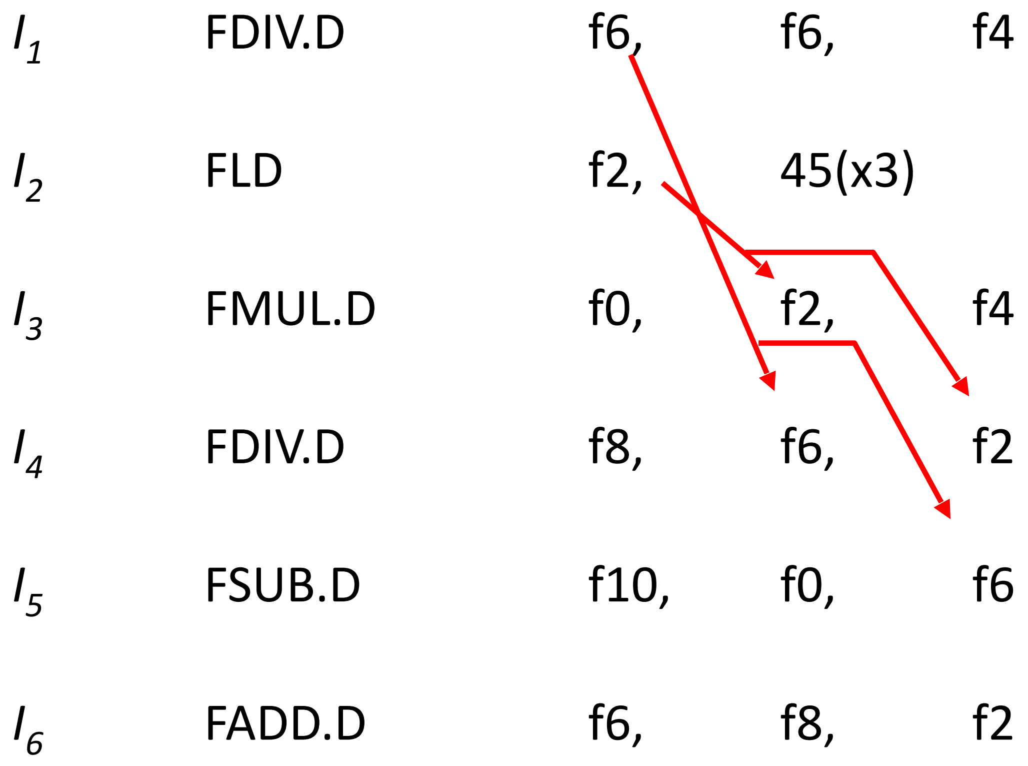
Data Hazards: An Example



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

Data Hazards: An Example

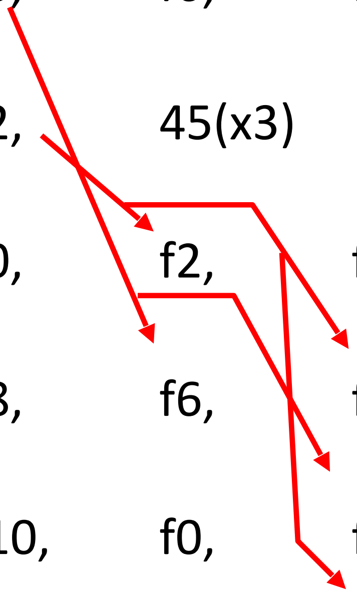


I_1	FDIV.D	f6,	f6,	f4
I_2	FLD	f2,	45(x3)	
I_3	FMUL.D	f0,	f2,	f4
I_4	FDIV.D	f8,	f6,	f2
I_5	FSUB.D	f10,	f0,	f6
I_6	FADD.D	f6,	f8,	f2

RAW Hazards

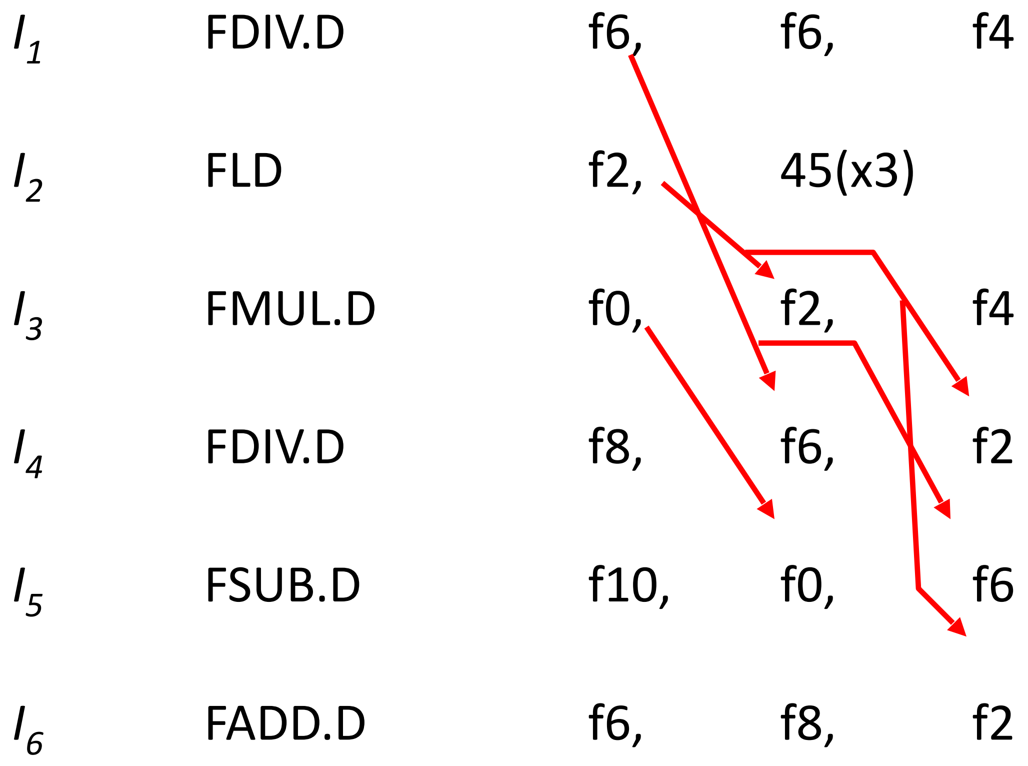
Data Hazards: An Example

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

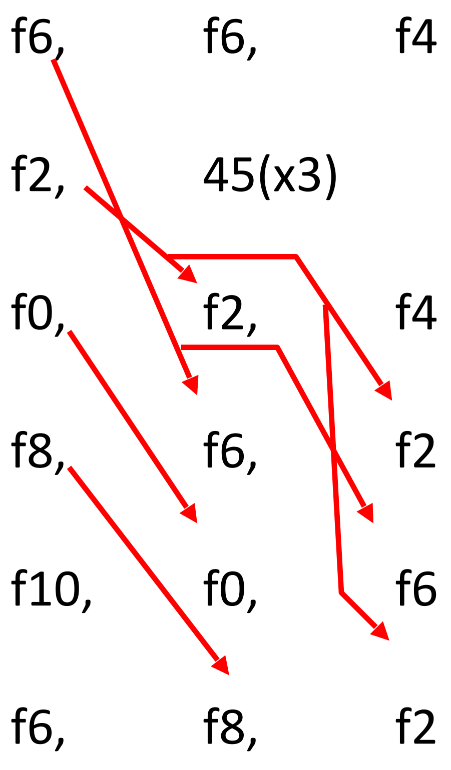
Data Hazards: An Example



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

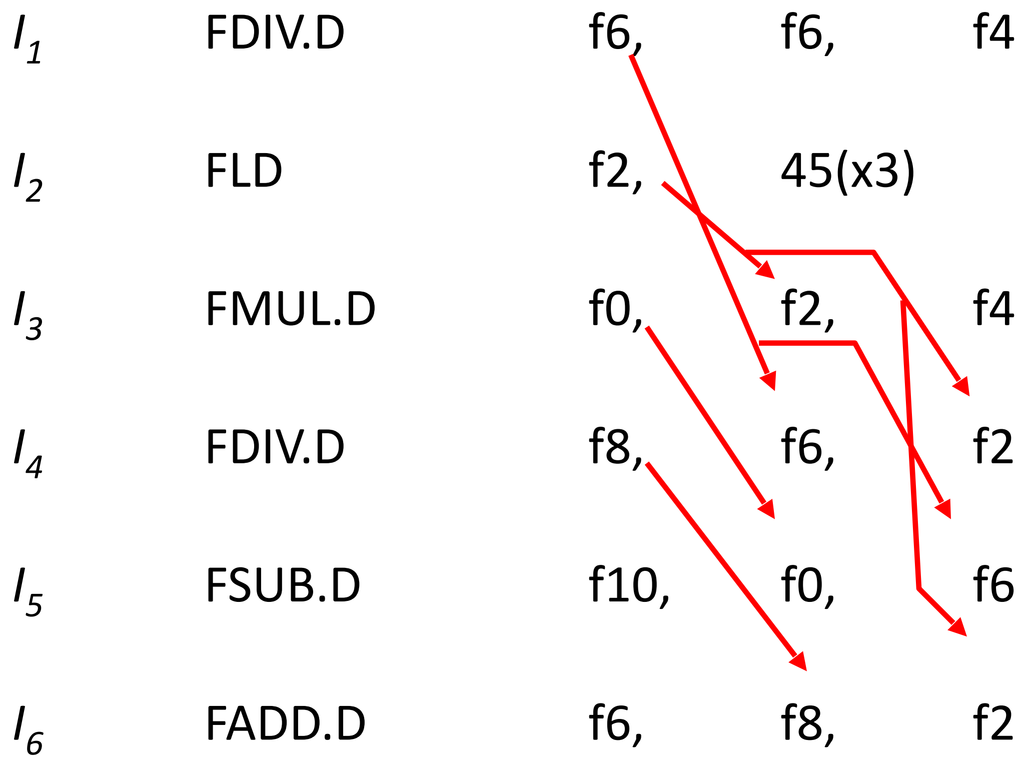
Data Hazards: An Example



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

Data Hazards: An Example

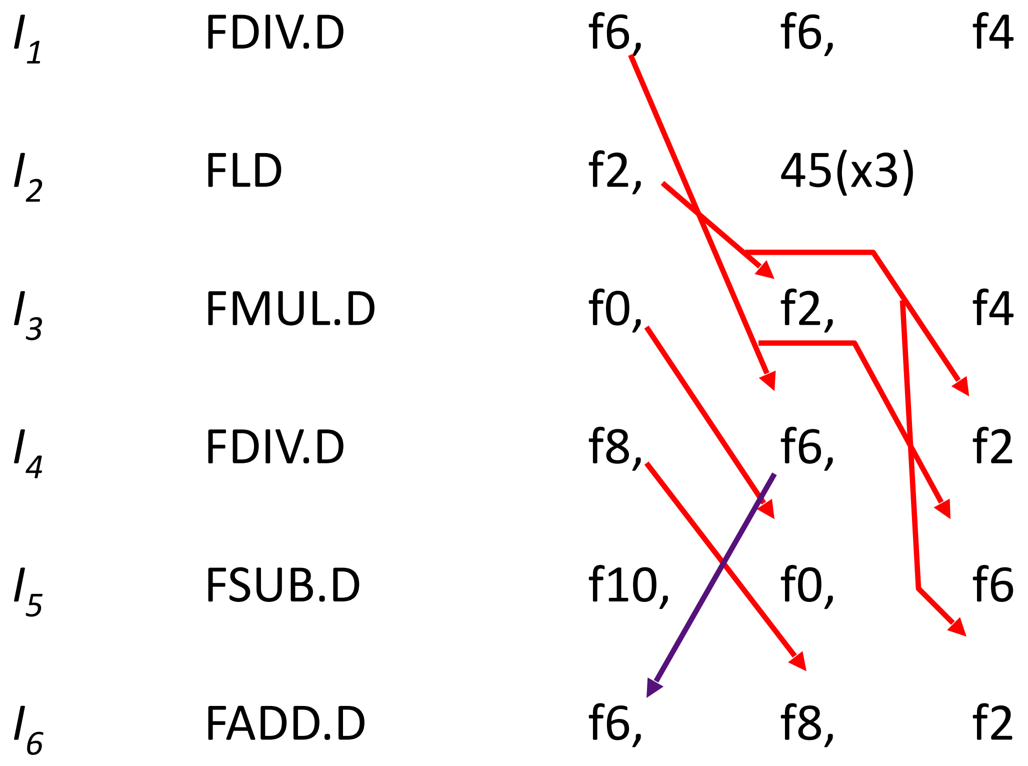


I_1	FDIV.D	f6,	f6,	f4
I_2	FLD	f2,	45(x3)	
I_3	FMUL.D	f0,	f2,	f4
I_4	FDIV.D	f8,	f6,	f2
I_5	FSUB.D	f10,	f0,	f6
I_6	FADD.D	f6,	f8,	f2

RAW Hazards

WAR Hazards

Data Hazards: An Example

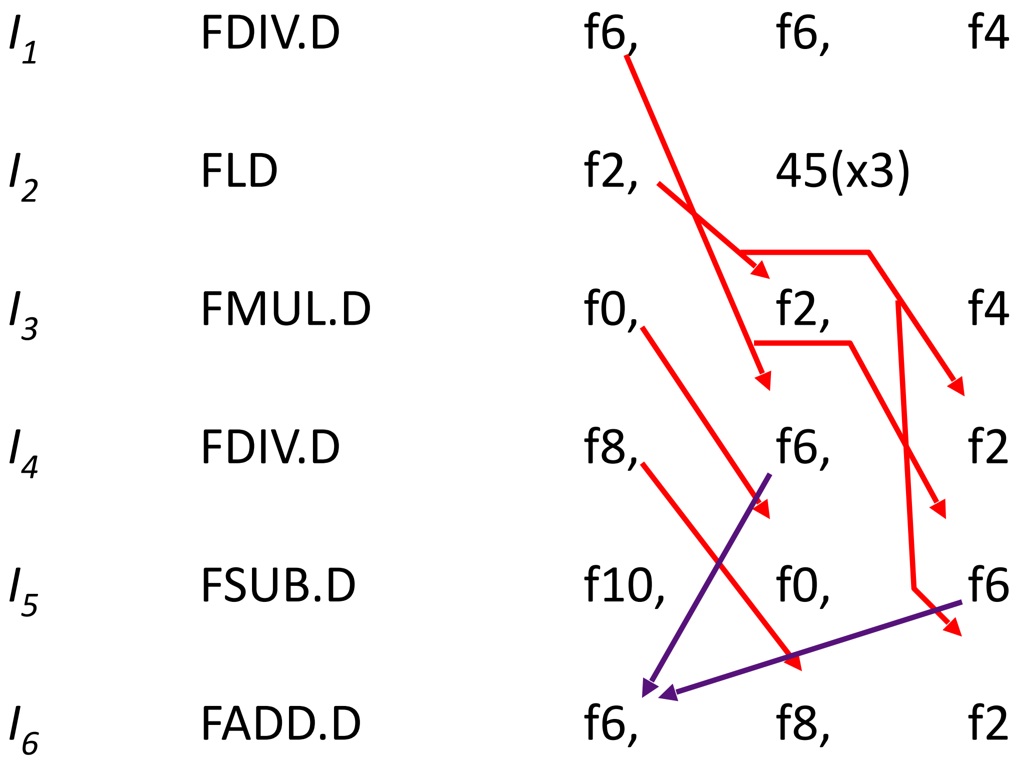


I_1	FDIV.D	f6,	f6,	f4
I_2	FLD	f2,	45(x3)	
I_3	FMUL.D	f0,	f2,	f4
I_4	FDIV.D	f8,	f6,	f2
I_5	FSUB.D	f10,	f0,	f6
I_6	FADD.D	f6,	f8,	f2

RAW Hazards

WAR Hazards

Data Hazards: An Example

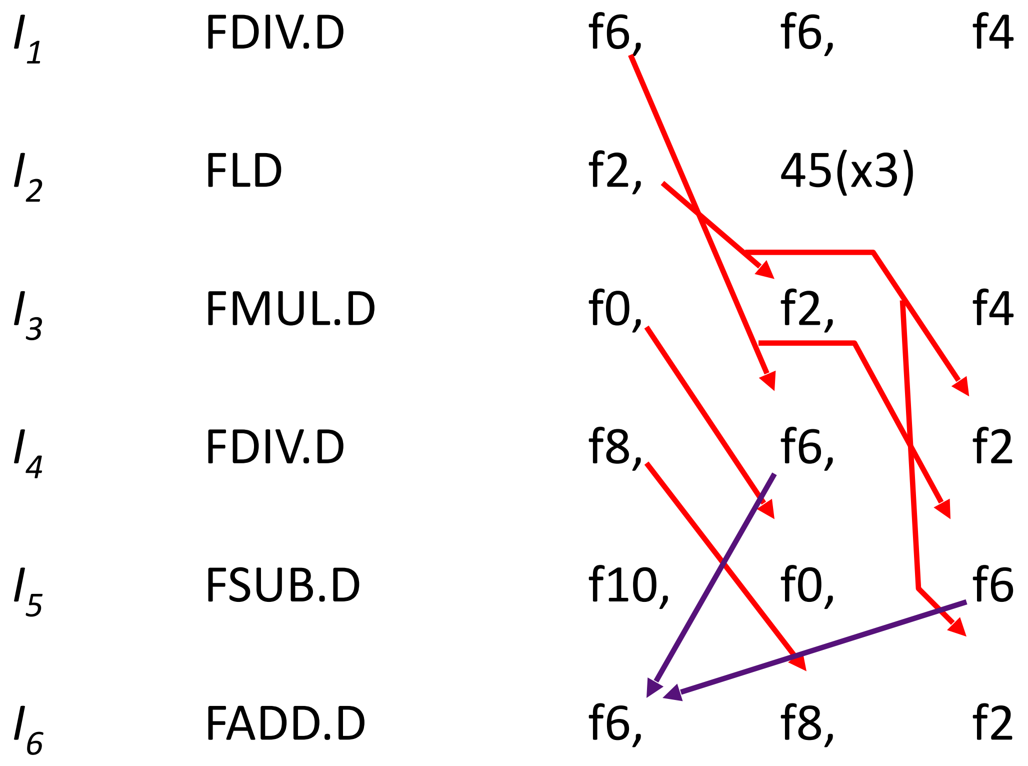


I_1	FDIV.D	f6,	f6,	f4
I_2	FLD	f2,	45(x3)	
I_3	FMUL.D	f0,	f2,	f4
I_4	FDIV.D	f8,	f6,	f2
I_5	FSUB.D	f10,	f0,	f6
I_6	FADD.D	f6,	f8,	f2

RAW Hazards

WAR Hazards

Data Hazards: An Example



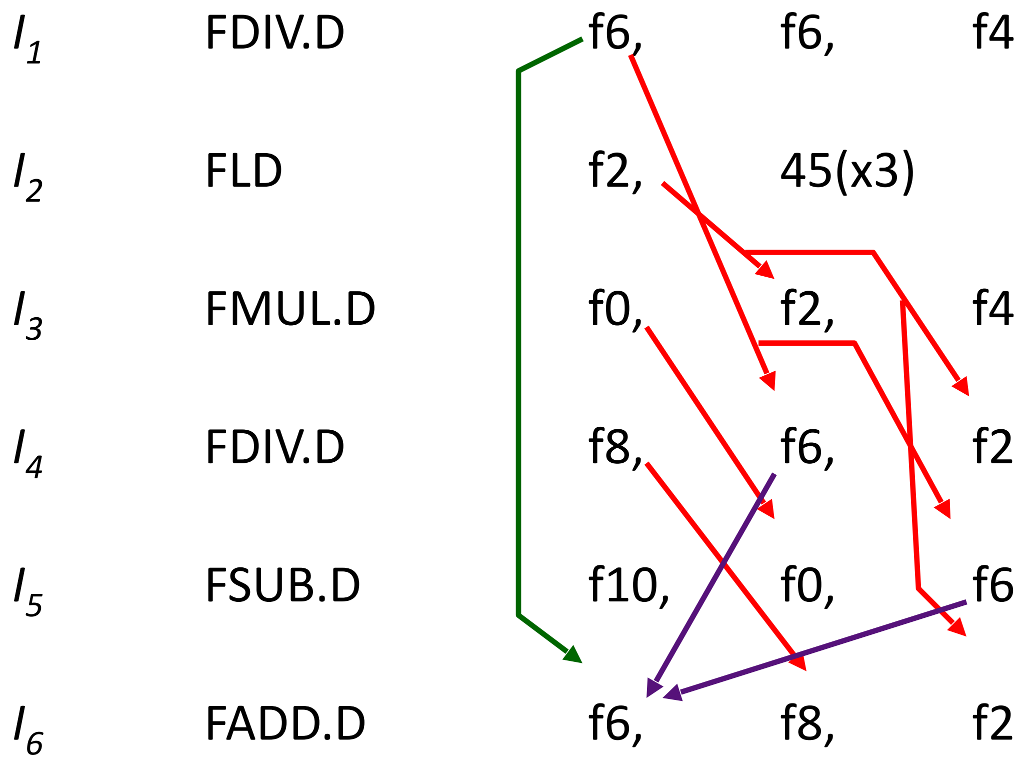
I_1	FDIV.D	f6,	f6,	f4
I_2	FLD	f2,	45(x3)	
I_3	FMUL.D	f0,	f2,	f4
I_4	FDIV.D	f8,	f6,	f2
I_5	FSUB.D	f10,	f0,	f6
I_6	FADD.D	f6,	f8,	f2

RAW Hazards

WAR Hazards

WAW Hazards

Data Hazards: An Example



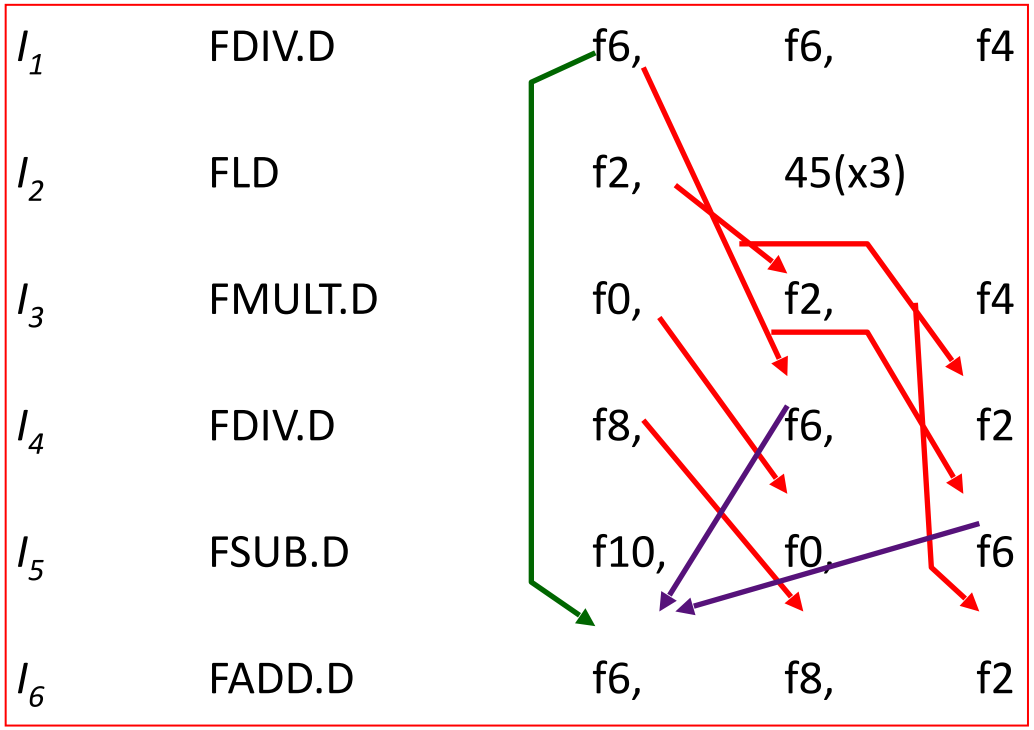
I_1	FDIV.D	f6,	f6,	f4
I_2	FLD	f2,	45(x3)	
I_3	FMUL.D	f0,	f2,	f4
I_4	FDIV.D	f8,	f6,	f2
I_5	FSUB.D	f10,	f0,	f6
I_6	FADD.D	f6,	f8,	f2

RAW Hazards

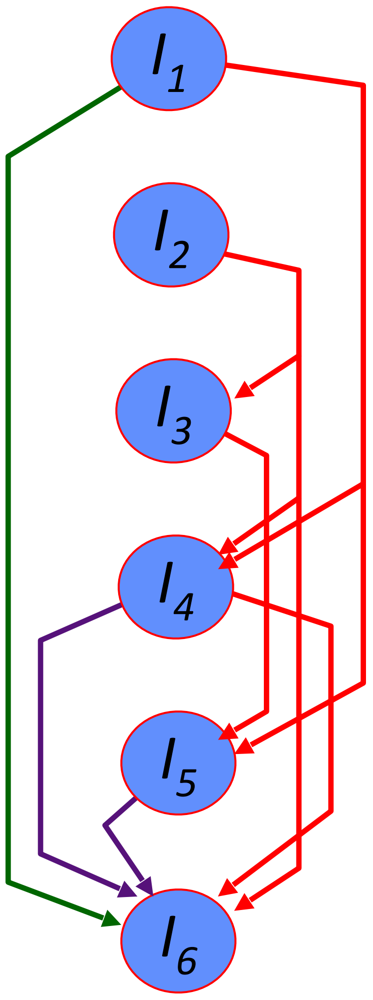
WAR Hazards

WAW Hazards

Single Issue Instruction Scheduling

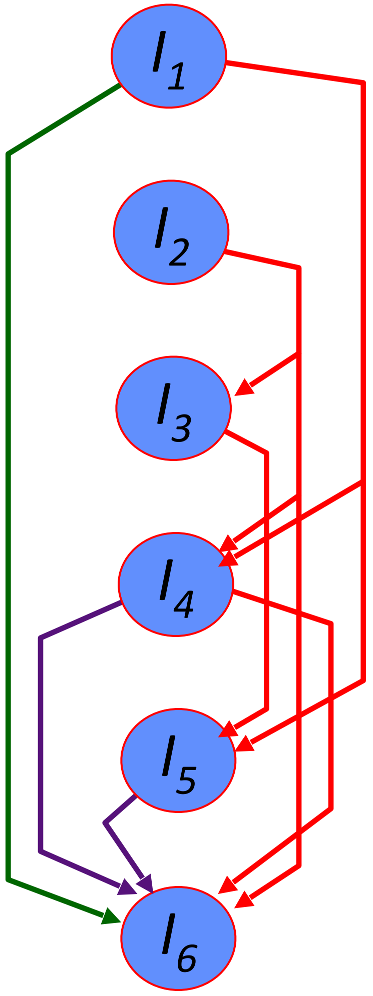


I_1	FDIV.D	f6,	f6,	f4
I_2	FLD	f2,	45(x3)	
I_3	FMULT.D	f0,	f2,	f4
I_4	FDIV.D	f8,	f6,	f2
I_5	FSUB.D	f10,	f0,	f6
I_6	FADD.D	f6,	f8,	f2



Single Issue 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:

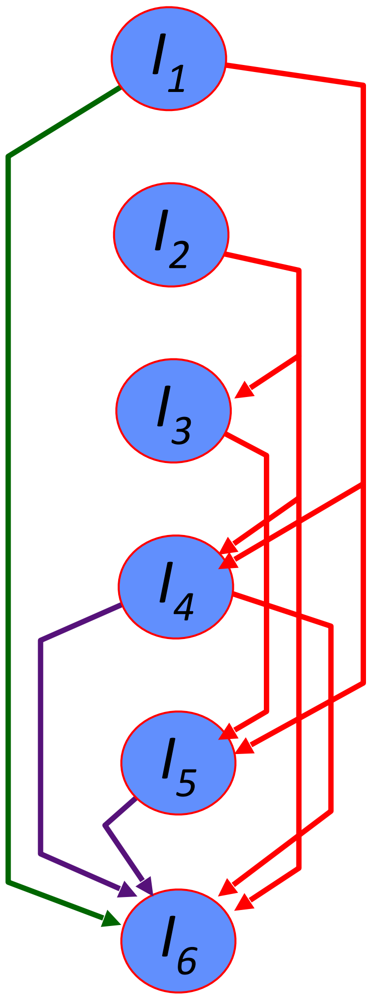
in-order I_1 I_2 I_3 I_4 I_5 I_6

out-of-order

out-of-order

Single Issue 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

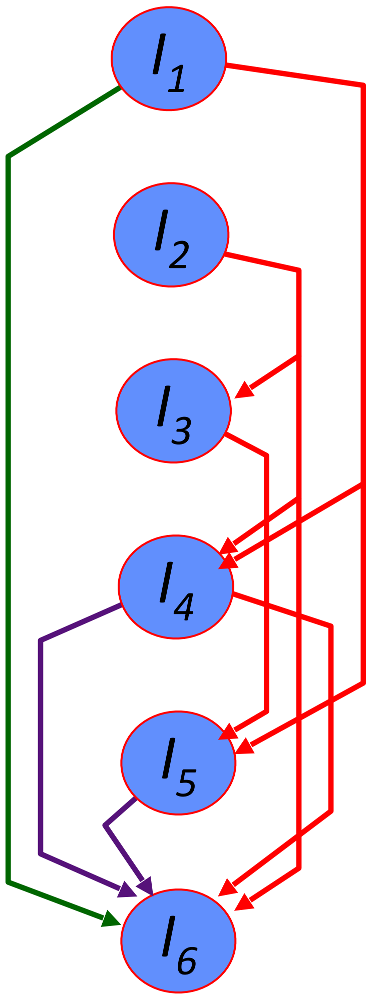


Valid orderings:

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						

Single Issue 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

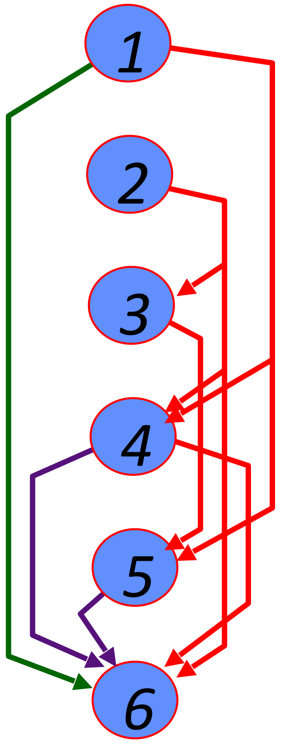


Valid orderings:

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
	I_1	I_2	I_3	I_5	I_4	I_6
out-of-order						

Out-of-order Completion

In-order Issue



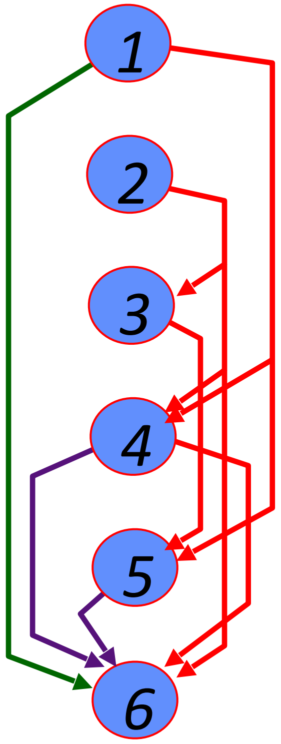
					Latency
I_1	FDIV.D	f6,	f6,	f4	4
I_2	FLD	f2,	45(x3)		1
I_3	FMULT.D	f0,	f2,	f4	3
I_4	FDIV.D	f8,	f6,	f2	4
I_5	FSUB.D	f10,	f0,	f6	1
I_6	FADD.D	f6,	f8,	f2	1

in-order comp 1 2

out-of-order comp 1 2

Out-of-order Completion

In-order Issue



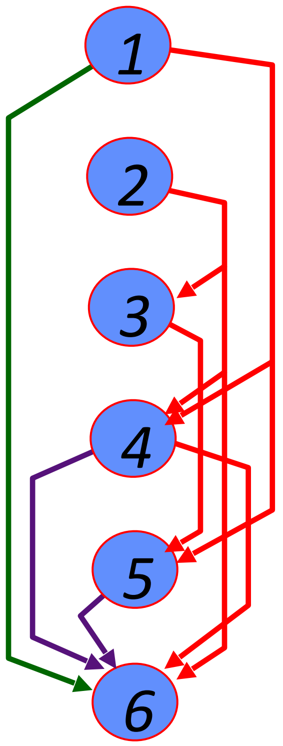
					Latency
I_1	FDIV.D	f6,	f6,	f4	*4*
I_2	FLD	f2,	45(x3)		*1*
I_3	FMULT.D	f0,	f2,	f4	*3*
I_4	FDIV.D	f8,	f6,	f2	*4*
I_5	FSUB.D	f10,	f0,	f6	*1*
I_6	FADD.D	f6,	f8,	f2	*1*

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

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 $\underline{6}$

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 issue an instruction into execution

- 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 Issue

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 issued by the Issue stage if a RAW hazard exists or the required FU is busy, and that operands are registered by functional unit on issue:

Can the issued instruction cause a

WAR hazard ?

WAW hazard ?

Simplifying the Data Structure Assuming In-order Issue

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

Can the issued instruction cause a

WAR hazard ?

NO: Earlier instructions read their operands at issue

WAW hazard ?

Simplifying the Data Structure Assuming In-order Issue

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

Can the issued instruction cause a

WAR hazard ?

NO: Earlier instructions read their operands 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 issue 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 Issue

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) before issue

FU available?
RAW?
WAR?
WAW?

Scoreboard for In-order Issue

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) before issue

FU available? Busy[FU#]
RAW?
WAR?
WAW?

Scoreboard for In-order Issue

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) before issue

FU available?	Busy[FU#]
RAW?	WP[src1] or WP[src2]
WAR?	
WAW?	

Scoreboard for In-order Issue

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) before issue

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

Scoreboard for In-order Issue

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) before issue

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	

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

Scoreboard Dynamics

	Functional Unit Status										Registers Reserved for Writes
	Int(1)	Add(1)	Mult(3)			Div(4)				WB	
t0	I_1					f6					f6

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

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

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

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 $\underline{I}_2$

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

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 $\underline{I_2}$
t3	I_3		f0						f6		f6, f0

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

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 $\underline{I_2}$
t3	I_3		f0						f6		f6, f0
t4				f0						f6	f6, f0 $\underline{I_1}$

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

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	$\underline{I_2}$
t3	I_3		f0						f6		f6, f0	
t4				f0						f6	f6, f0	$\underline{I_1}$
t5	I_4				f0	f8					f0, f8	

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

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	$\underline{I_2}$
t3	I_3		f0						f6		f6, f0	
t4				f0						f6	f6, f0	$\underline{I_1}$
t5	I_4				f0	f8					f0, f8	
t6							f8			f0	f0, f8	$\underline{I_3}$

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

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	$\underline{I_2}$
t3	I_3		f0						f6		f6, f0	
t4				f0						f6	f6, f0	$\underline{I_1}$
t5	I_4				f0	f8					f0, f8	
t6							f8			f0	f0, f8	$\underline{I_3}$
t7	I_5	f10						f8			f8, f10	

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

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	$\underline{I_2}$
t3	I_3		f0						f6		f6, f0	
t4				f0						f6	f6, f0	$\underline{I_1}$
t5	I_4				f0	f8					f0, f8	
t6							f8			f0	f0, f8	$\underline{I_3}$
t7	I_5	f10						f8			f8, f10	
t8									f8	f10	f8, f10	$\underline{I_5}$

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

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	$\underline{I_2}$
t3	I_3		f0						f6		f6, f0	
t4				f0						f6	f6, f0	$\underline{I_1}$
t5	I_4				f0	f8					f0, f8	
t6							f8			f0	f0, f8	$\underline{I_3}$
t7	I_5	f10						f8			f8, f10	
t8									f8	f10	f8, f10	$\underline{I_5}$
t9										f8	f8	$\underline{I_4}$

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

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	$\underline{I_2}$
t3	I_3		f0						f6		f6, f0	
t4				f0						f6	f6, f0	$\underline{I_1}$
t5	I_4				f0	f8					f0, f8	
t6							f8			f0	f0, f8	$\underline{I_3}$
t7	I_5	f10						f8			f8, f10	
t8									f8	f10	f8, f10	$\underline{I_5}$
t9										f8	f8	$\underline{I_4}$
t10	I_6	f6									f6	

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

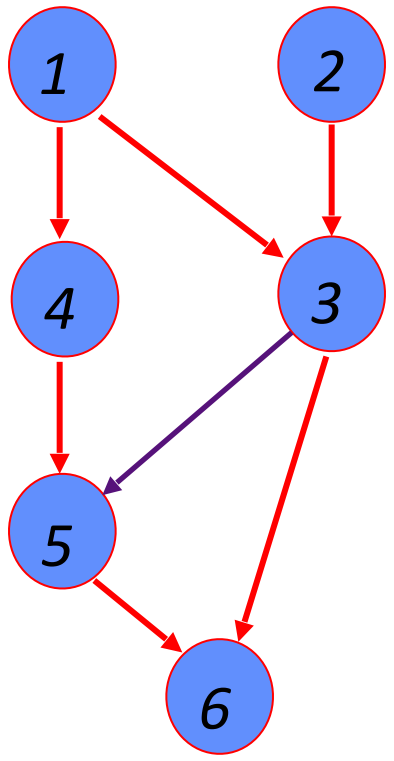
Scoreboard Dynamics

	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	$\underline{I_2}$
t3	I_3		f0						f6		f6, f0	
t4				f0						f6	f6, f0	$\underline{I_1}$
t5	I_4				f0	f8					f0, f8	
t6							f8			f0	f0, f8	$\underline{I_3}$
t7	I_5	f10						f8			f8, f10	
t8									f8	f10	f8, f10	$\underline{I_5}$
t9										f8	f8	$\underline{I_4}$
t10	I_6	f6									f6	
t11										f6	f6	$\underline{I_6}$

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*

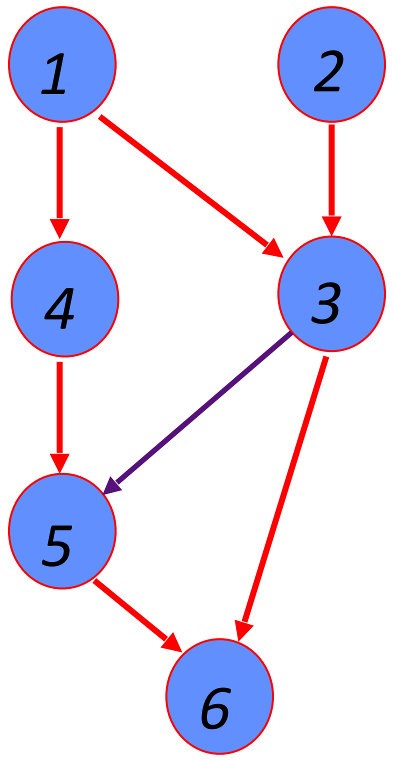
					latency
1	FLD	f2,	34(x2)		*1*
2	FLD	f4,	45(x3)		*long*
3	FMULT.D	f6,	f4,	f2	*3*
4	FSUB.D	f8,	f2,	f2	*1*
5	FDIV.D	f4,	f2,	f8	*4*
6	FADD.D	f10,	f6,	f4	*1*


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

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

In-Order Issue Limitations: *an example*

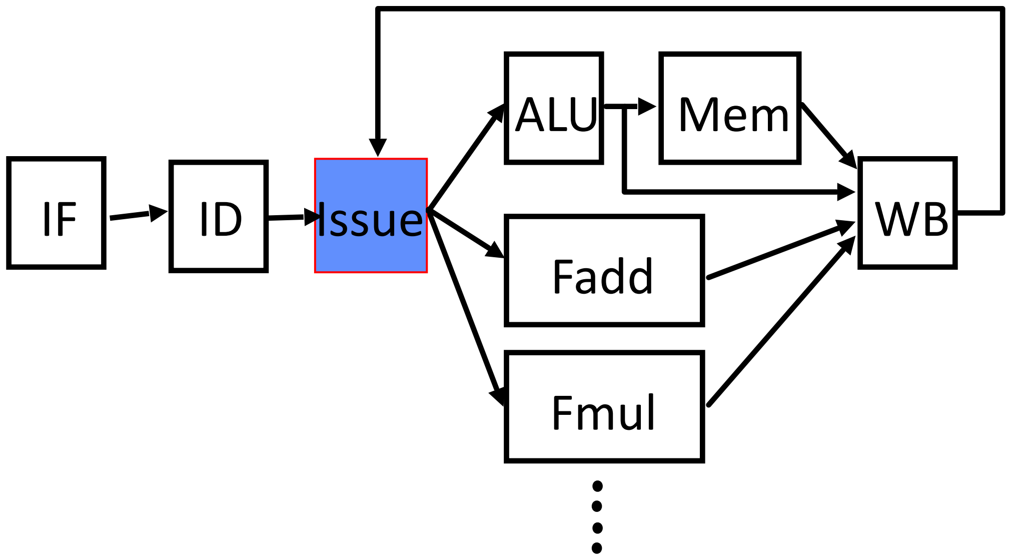
					latency
1	FLD	f2,	34(x2)		*1*
2	FLD	f4,	45(x3)		*long*
3	FMULT.D	f6,	f4,	f2	*3*
4	FSUB.D	f8,	f2,	f2	*1*
5	FDIV.D	f4,	f2,	f8	*4*
6	FADD.D	f10,	f6,	f4	*1*



In-order: 1 (2,<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 issued

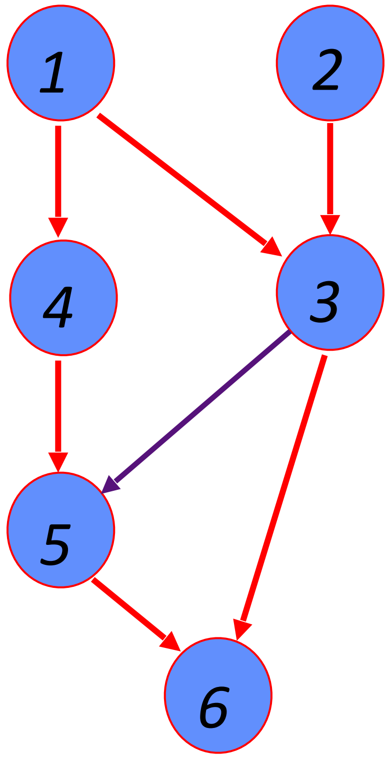
Out-of-Order Issue



- 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 registering of input operands at functional unit)
- Any instruction in buffer whose RAW hazards are satisfied can be issued (for now, at most one issue per cycle). On a write back (WB), new instructions may get enabled.

Issue Limitations: In-Order and 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*


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

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

Issue Limitations: In-Order and 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*

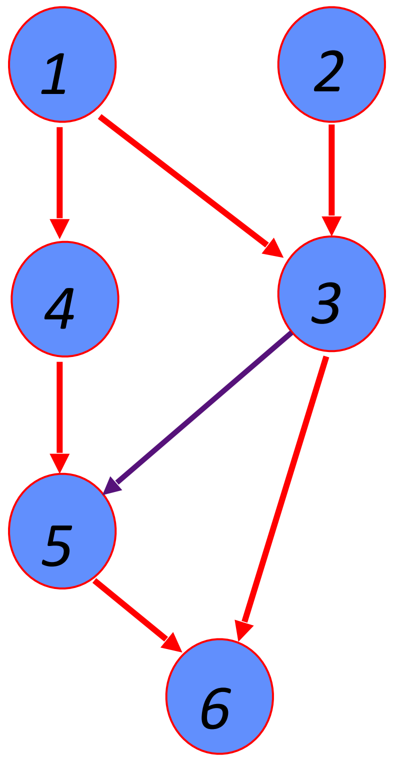
$$\begin{tikzcd}
1 \arrow[d] \arrow[drr] & & 2 \arrow[d] \\
4 \arrow[d] & & 3 \arrow[dll] \arrow[ddl] \\
5 \arrow[dr] & & \\
 & 6 &
\end{tikzcd}$$

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 $\underline{6}$

Issue Limitations: In-Order and 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,$\underline{1}$) . . . . . . $\underline{2}$ 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!

How many instructions can be in the pipeline?

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

How many instructions can be in the pipeline?

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

Number of Registers

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 issue 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.

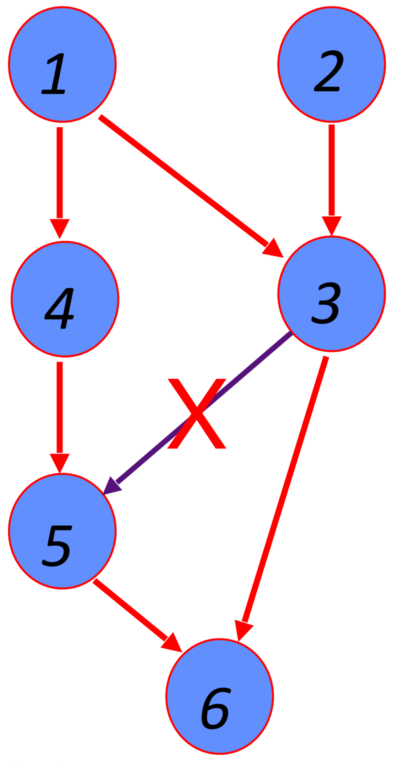
IBM 360 had only 4 floating-point registers

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

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

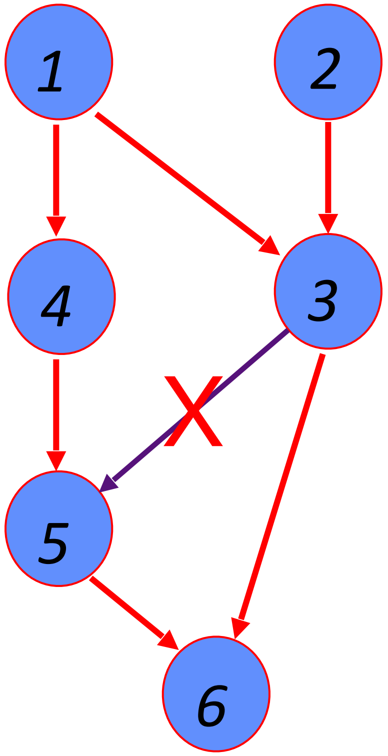
Any antidependence can be eliminated by renaming.

(renaming ➔ additional storage)

Can it be done in hardware?

Issue Limitations: In-Order and 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,<u>1</u>) . . . . . . <u>2</u> 3 4 <u>4</u> <u>3</u> 5 . . . <u>5</u> 6 <u>6</u>

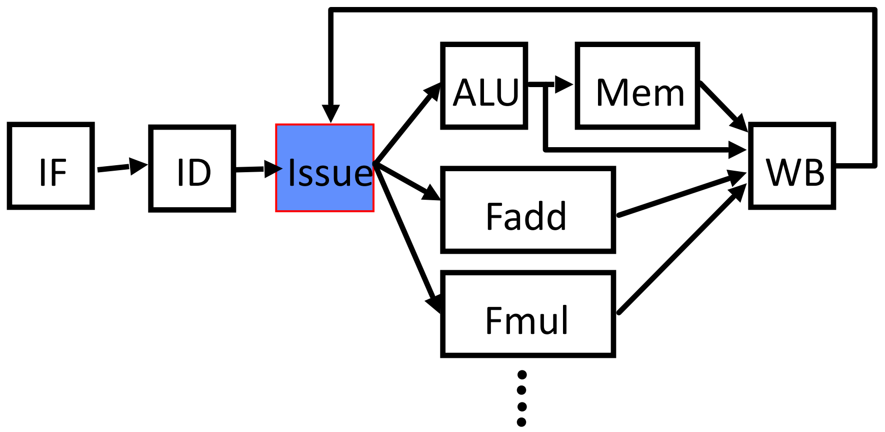
Out-of-order: 1 (2,<u>1</u>) 4 <u>4</u> 5 . . . <u>2</u> (3,<u>5</u>) <u>3</u> 6 <u>6</u>

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

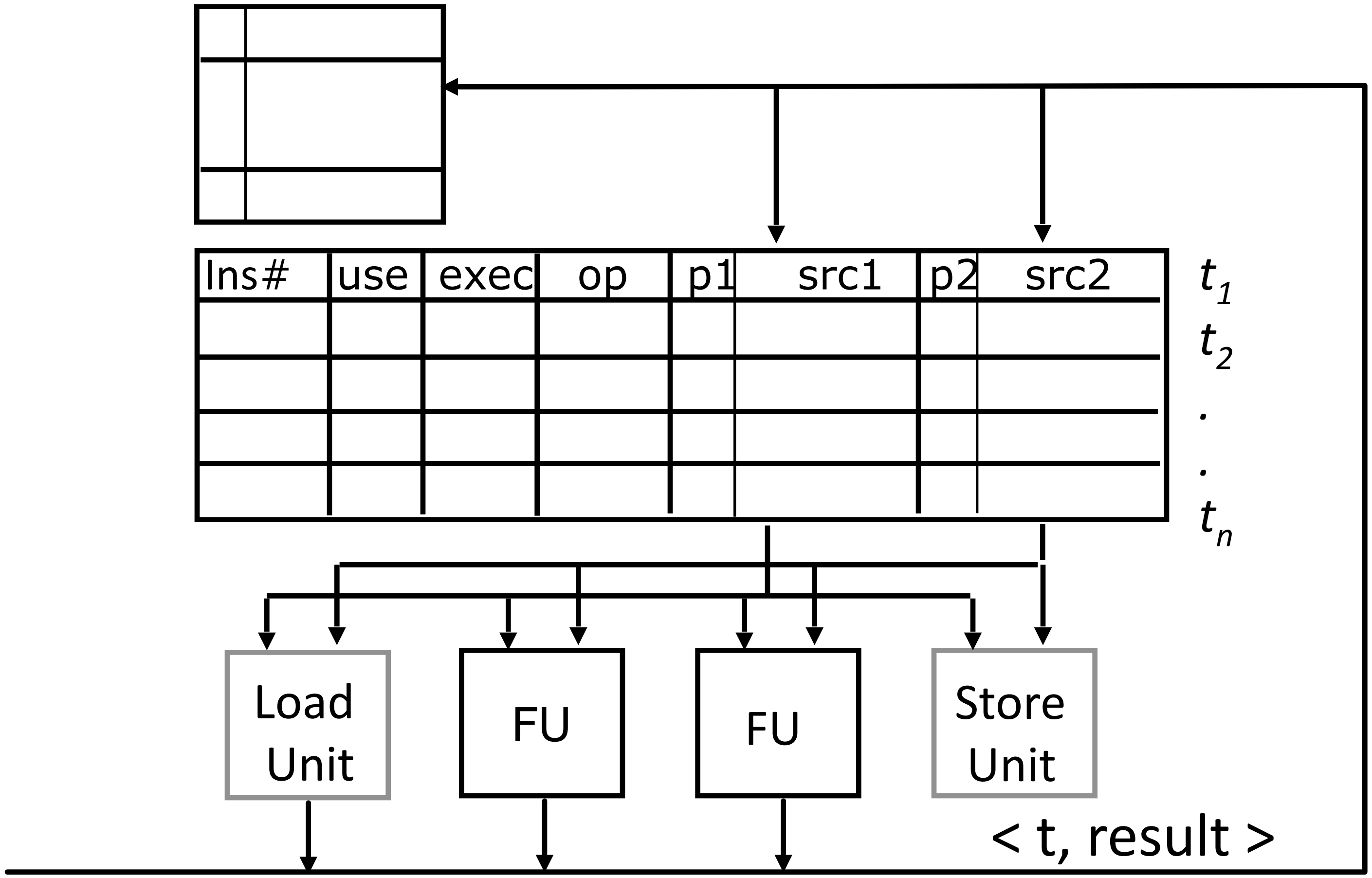
- Any instruction in ROB whose RAW hazards have been satisfied can be issued

  ➔ Out-of-order or dataflow execution

Renaming Structures

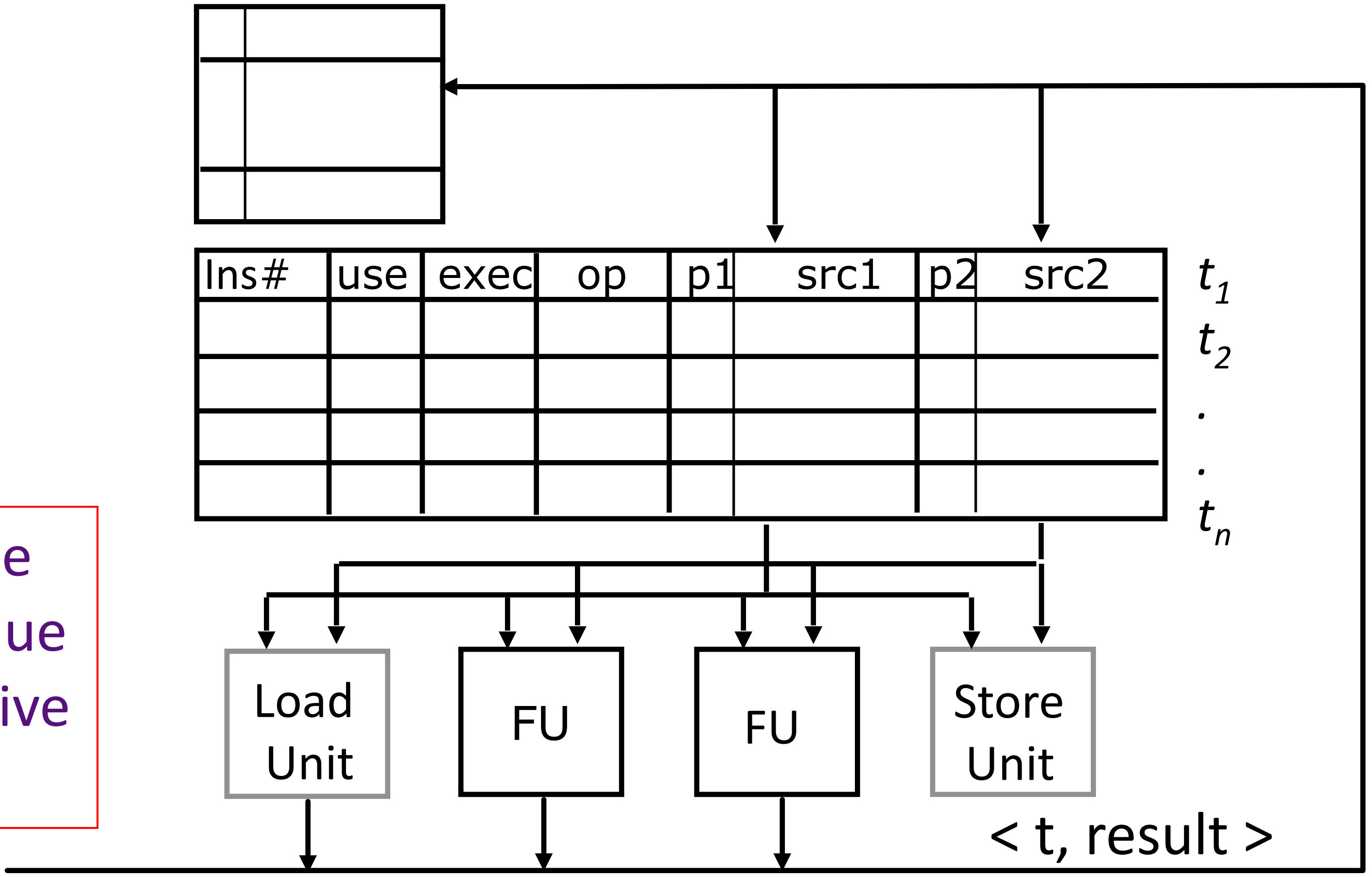
Renaming table & regfile

Reorder buffer



- 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

Renaming Structures



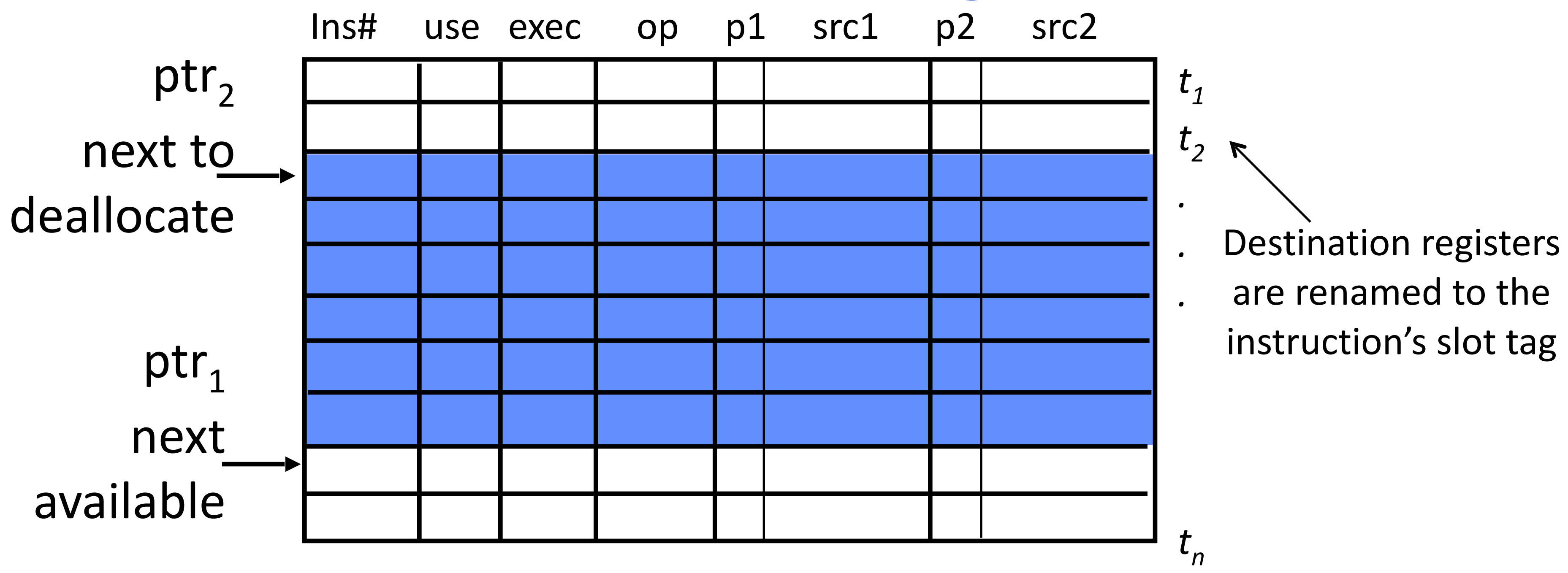
Renaming table & regfile

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



ROB managed circularly

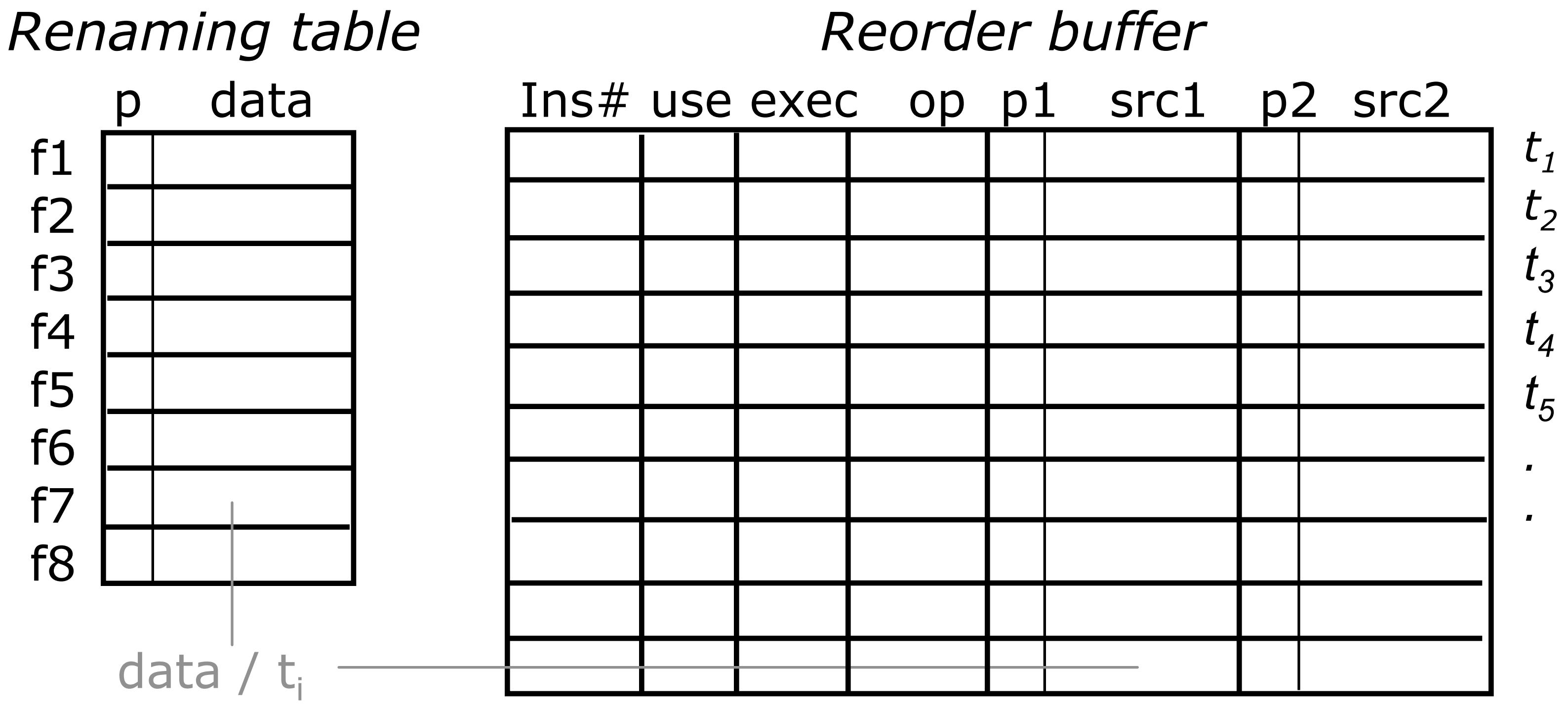
- "exec" bit is set when instruction begins execution
- When an instruction completes its "use" bit is marked free
- ptr$_2$ is incremented only if the "use" bit is marked free

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

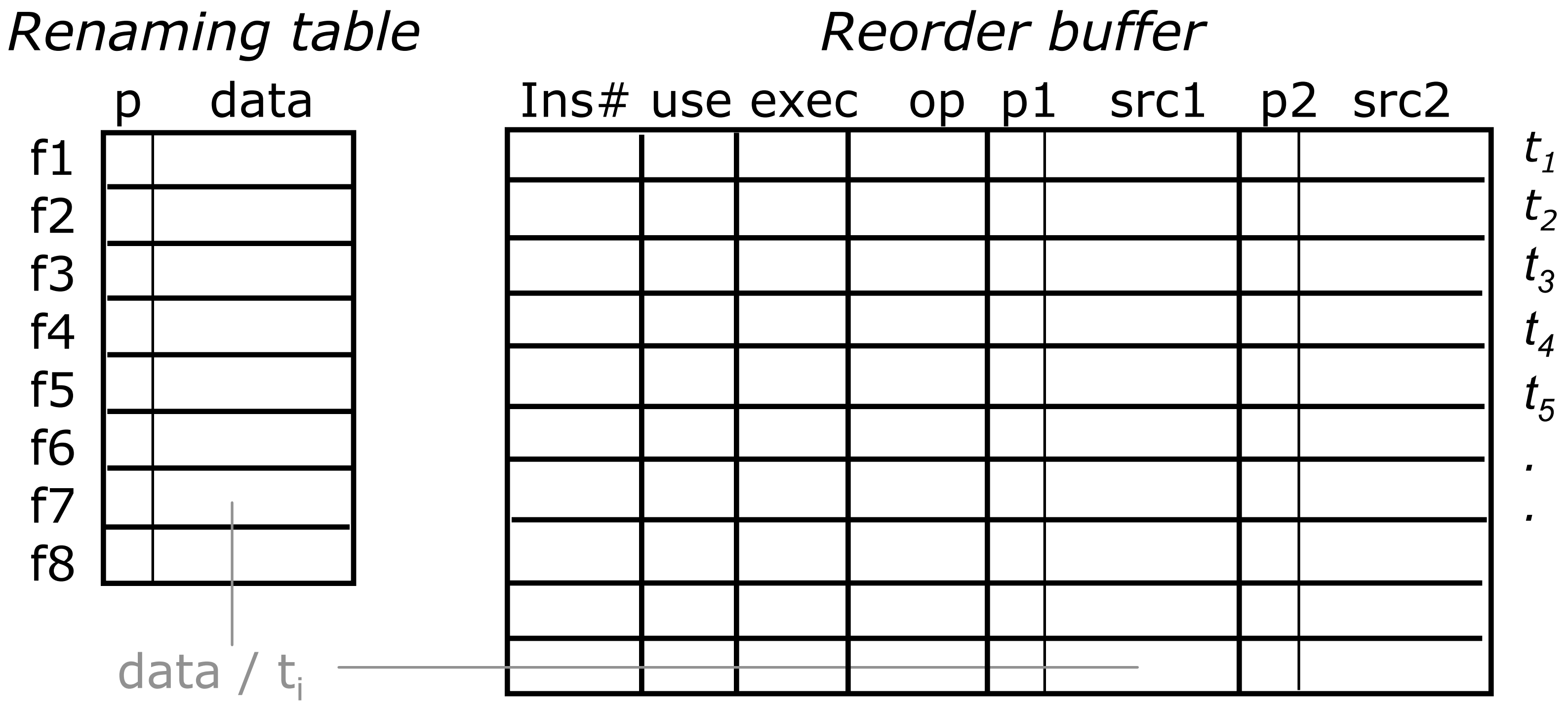


1 FLD	f2,	34(x2)	
2 FLD	f4,	45(x3)	
3 FMULT.D	f6,	f4,	f2
4 FSUB.D	f8,	f2,	f2
5 FDIV.D	f4,	f2,	f8
6 FADD.D	f10,	f6,	f4

- *When are tags in sources replaced by data?*
- *When can a name be reused?*

Renaming & Out-of-order Issue

An example

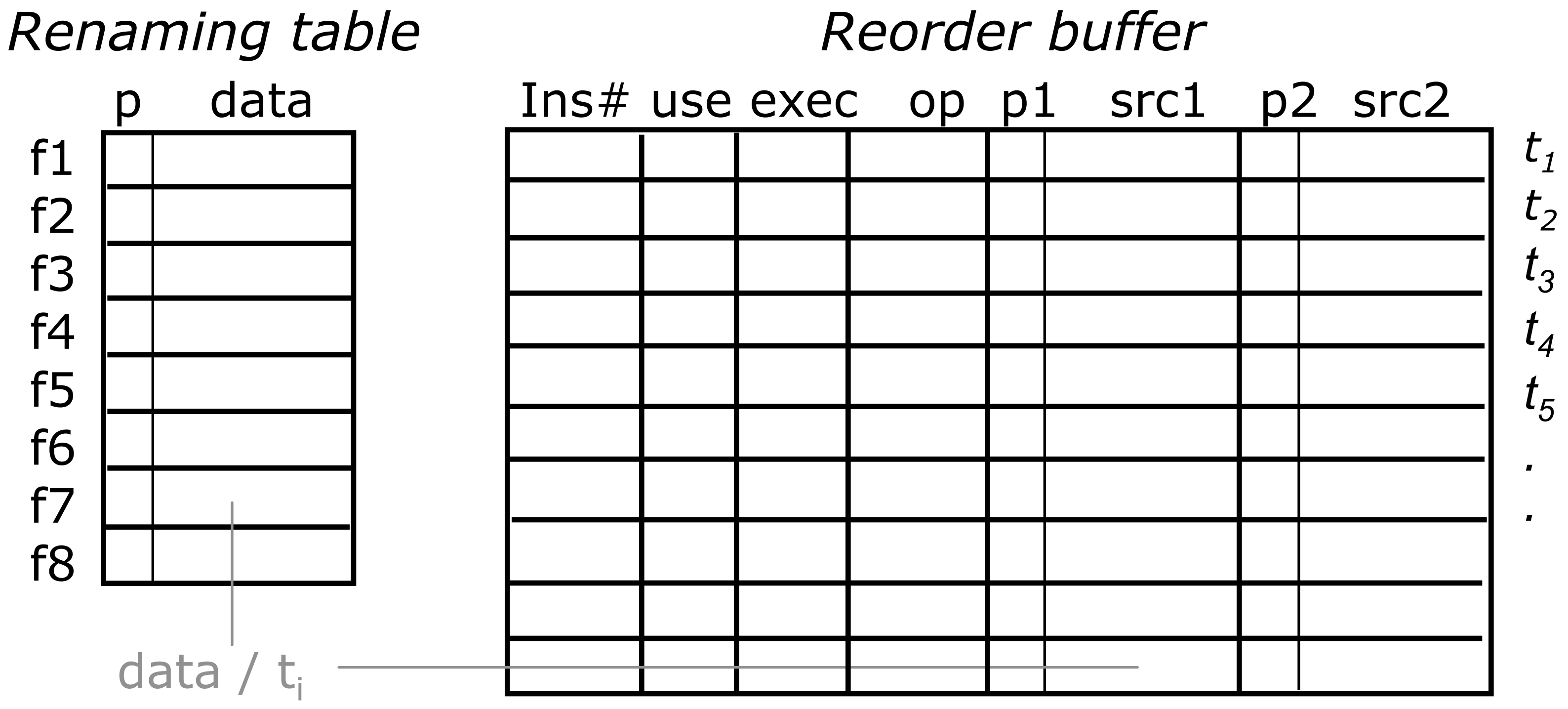


1 FLD	f2,	34(x2)	
2 FLD	f4,	45(x3)	
3 FMULT.D	f6,	f4,	f2
4 FSUB.D	f8,	f2,	f2
5 FDIV.D	f4,	f2,	f8
6 FADD.D	f10,	f6,	f4

- *When are tags in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*

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		
f3		
f4		
f5		
f6		
f7		
f8		

Reorder buffer

Ins#	use	exec	op	p1	src1	p2	src2	
1	1	0	LD					t_1
								t_2
								t_3
								t_4
								t_5
								.
								.

data / 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

- *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		t1
f3		
f4		
f5		
f6		
f7		
f8		

Reorder buffer

Ins#	use	exec	op	p1	src1	p2	src2	
1	1	0	LD					t_1
								t_2
								t_3
								t_4
								t_5
								.
								.

data / 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

- *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		t1
f3		
f4		
f5		
f6		
f7		
f8		

Reorder buffer

Ins#	use	exec	op	p1	src1	p2	src2	
1	1	1	LD					t_1
								t_2
								t_3
								t_4
								t_5
								.
								.

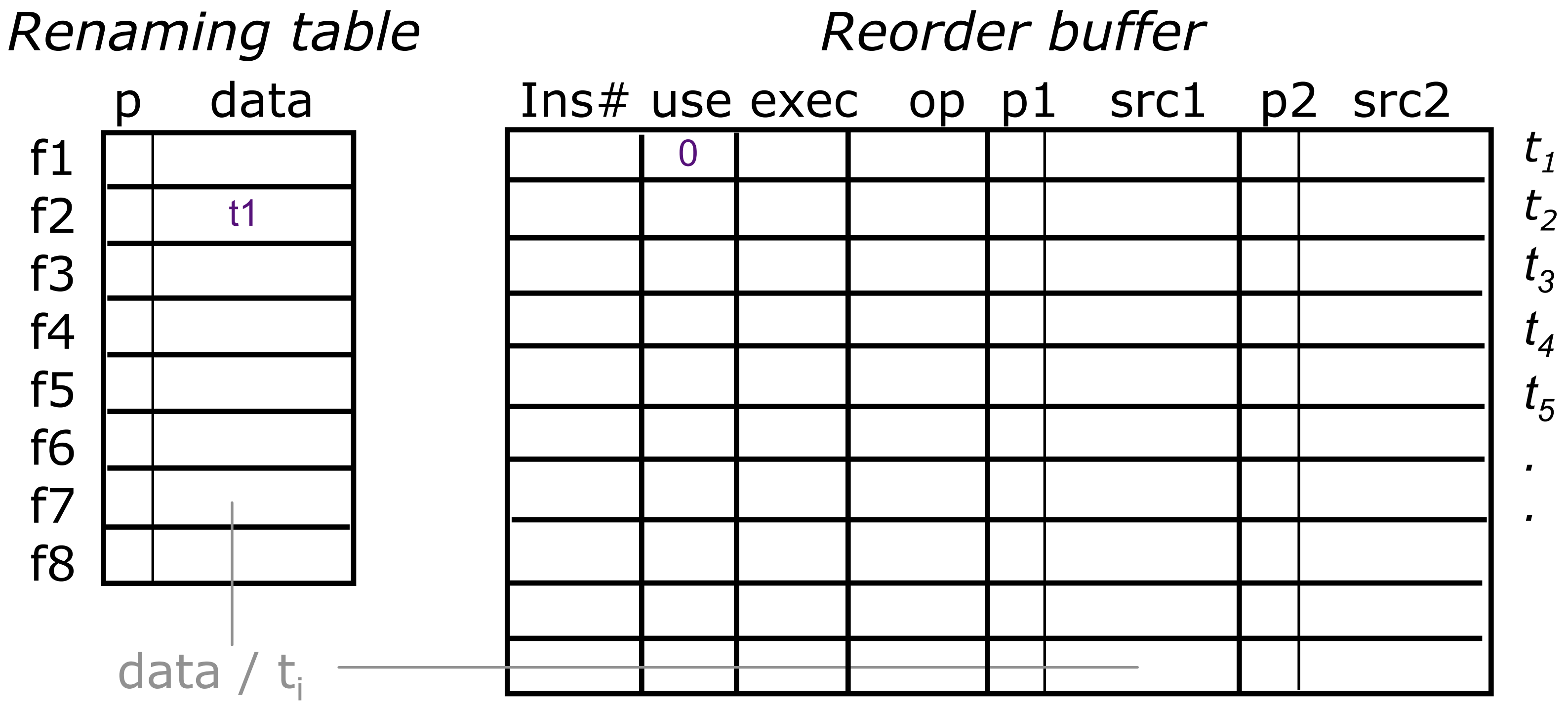
data / t_i

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

- *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

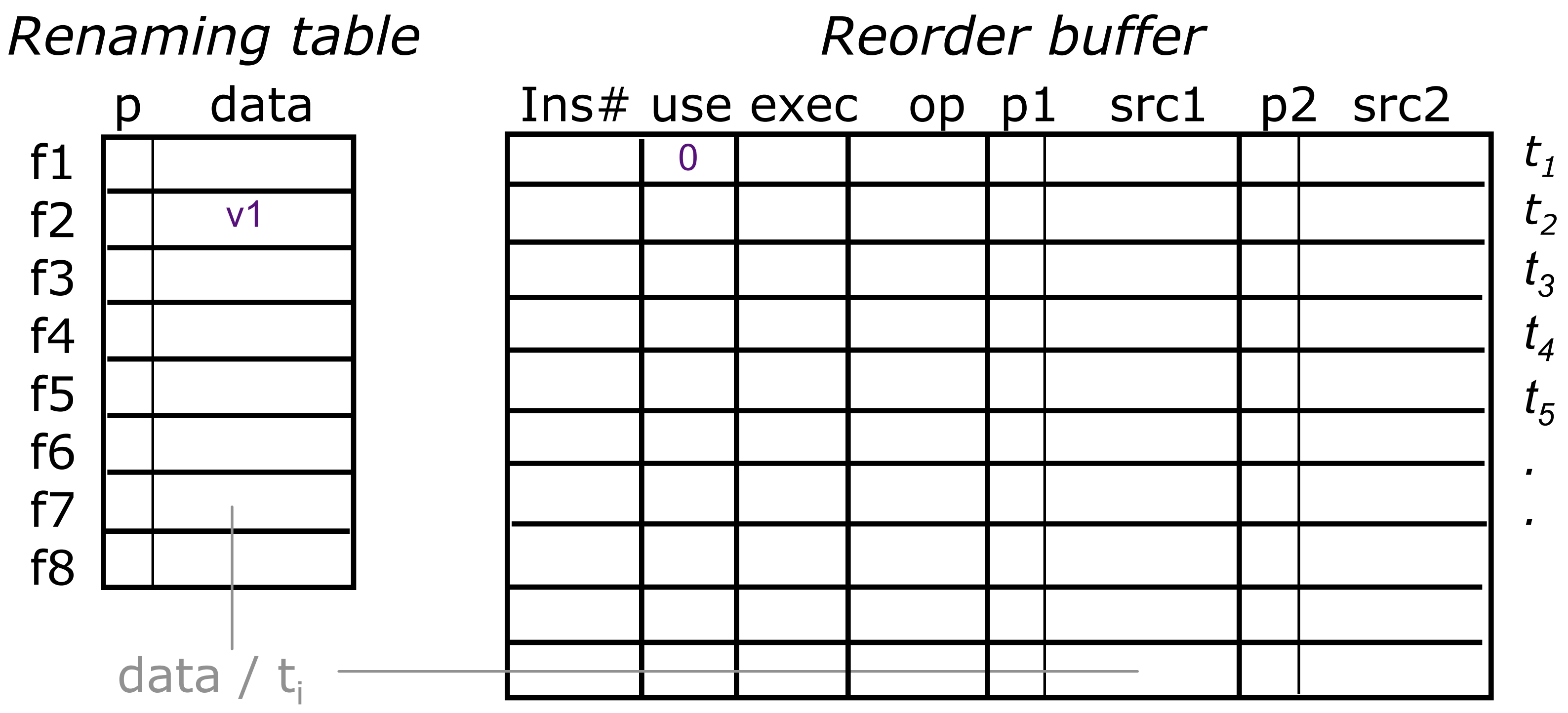


1 FLD	f2,	34(x2)	
2 FLD	f4,	45(x3)	
3 FMULT.D	f6,	f4,	f2
4 FSUB.D	f8,	f2,	f2
5 FDIV.D	f4,	f2,	f8
6 FADD.D	f10,	f6,	f4

- *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



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		t2
f5		
f6		
f7		
f8		

Reorder buffer

Ins#	use	exec	op	p1	src1	p2	src2	
	0							t_1
2	1	0	LD					t_2
								t_3
								t_4
								t_5
								.
								.

data / 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

- *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		t2
f5		
f6		
f7		
f8		

Reorder buffer

Ins#	use	exec	op	p1	src1	p2	src2	
	0							t_1
2	1	1	LD					t_2
								t_3
								t_4
								t_5
								.
								.

data / 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

- *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		t2
f5		
f6		t3
f7		
f8		

data / t_i

Reorder buffer

Ins#	use	exec	op	p1	src1	p2	src2	
	0							t_1
2	1	1	LD					t_2
3	1	0	MUL	0	t2	1	v1	t_3
								t_4
								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

Renaming & Out-of-order Issue

An example

Renaming table

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

Reorder buffer

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

data / 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

- *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		t2
f5		
f6		t3
f7		
f8		t4

data / t_i

Reorder buffer

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

Renaming & Out-of-order Issue

An example

Renaming table

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

Reorder buffer

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

data / 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

- *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		t4

Reorder buffer

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

data / 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

- *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		t4

data / t_i

Reorder buffer

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

Renaming & Out-of-order Issue

An example

Renaming table

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

data / t_i

Reorder buffer

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

Renaming & Out-of-order Issue

An example

Renaming table

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

data / t_i

Reorder buffer

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

Renaming & Out-of-order Issue

An example

Renaming table

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

data / t_i

Reorder buffer

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

Renaming & Out-of-order Issue

An example

Renaming table

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

data / t_i

Reorder buffer

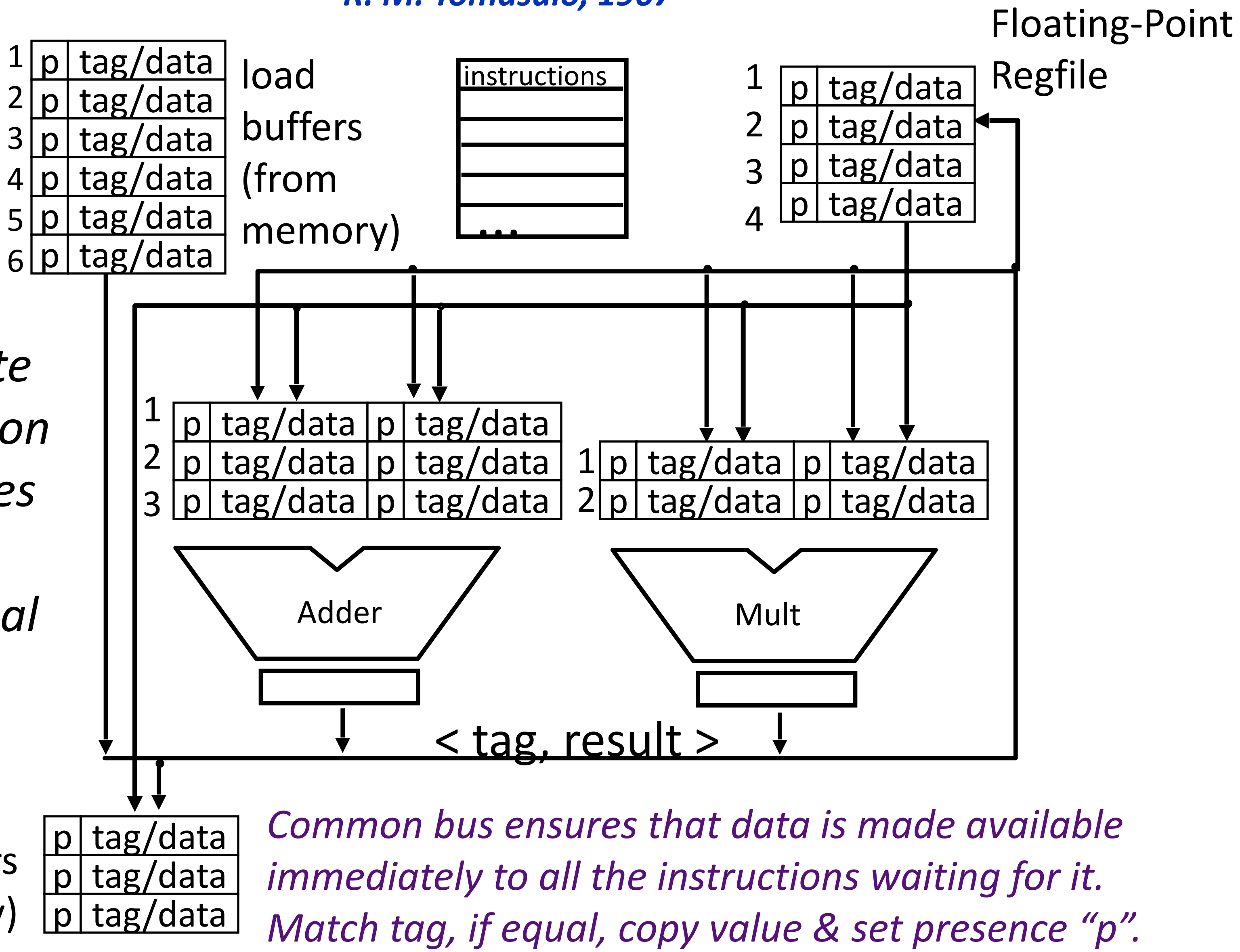
Ins#	use	exec	op	p1	src1	p2	src2	
	0							t_1
2	0							t_2
3	1	0	MUL	1	v2	1	v1	t_3
4	0							t_4
5	1	0	DIV	1	v1	1	v4	t_5
								.
								.

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

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".

IBM ACS

- Second supercomputer project (Y) started at IBM in response to CDC6600
- Multiple Dynamic instruction Scheduling (DIS) invented by Lynn Conway for ACS
  - Used unary encoding of register specifiers and wired-OR logic to detect any hazards (similar design used in Alpha 21264 in 1995!)
- Seven-issue, out-of-order processor
  - Two decoupled streams, each with DIS
- Cancelled in favor of IBM360-compatible machines

Out-of-Order Fades into Background

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:

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

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

Out-of-Order Fades into Background

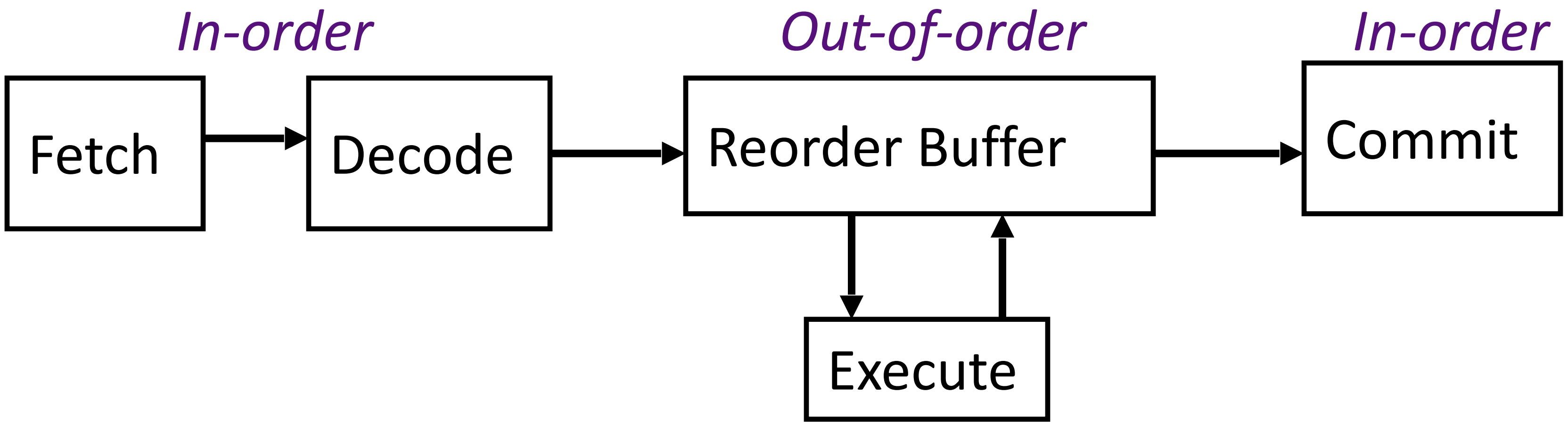
Out-of-order processing implemented commercially in 1960s, but disappeared again until 1990s as two major problems had to be solved:

- 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

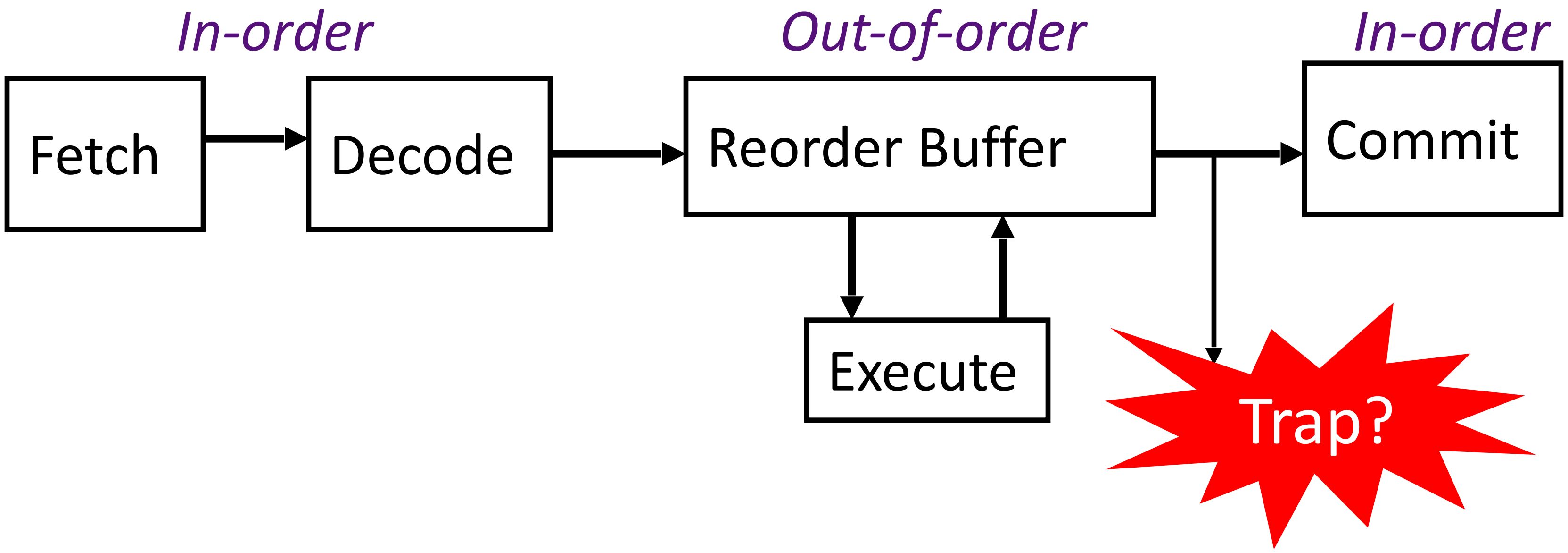
- Big advantage to fit processor & caches on one chip
- Microprocessors had era of 1%/week performance scaling

In-Order Commit for Precise Traps



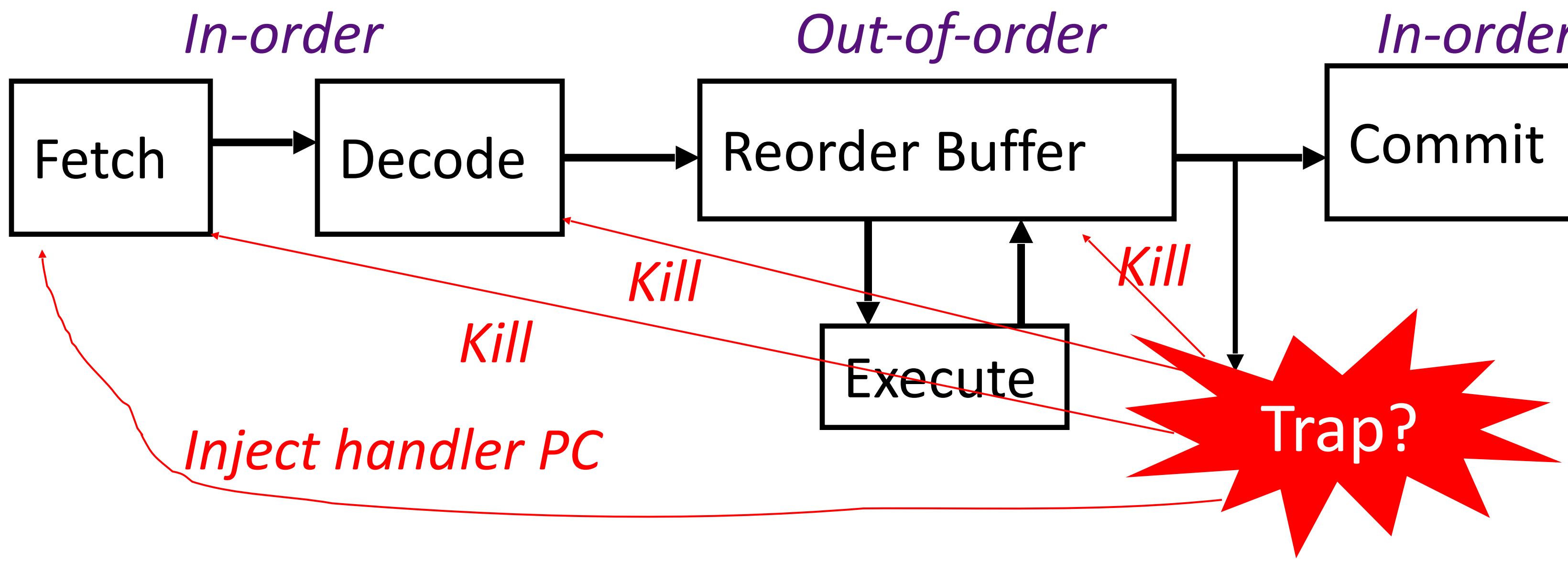
- In-order instruction fetch and decode, and dispatch to reservation stations inside reorder buffer
- Instructions issue from reservation stations out-of-order
- Out-of-order completion, values stored in temporary buffers
- Commit is in-order, checks for traps, and if none updates architectural state

In-Order Commit for Precise Traps



- In-order instruction fetch and decode, and dispatch to reservation stations inside reorder buffer
- Instructions issue from reservation stations out-of-order
- Out-of-order completion, values stored in temporary buffers
- Commit is in-order, checks for traps, and if none updates architectural state

In-Order Commit for Precise Traps

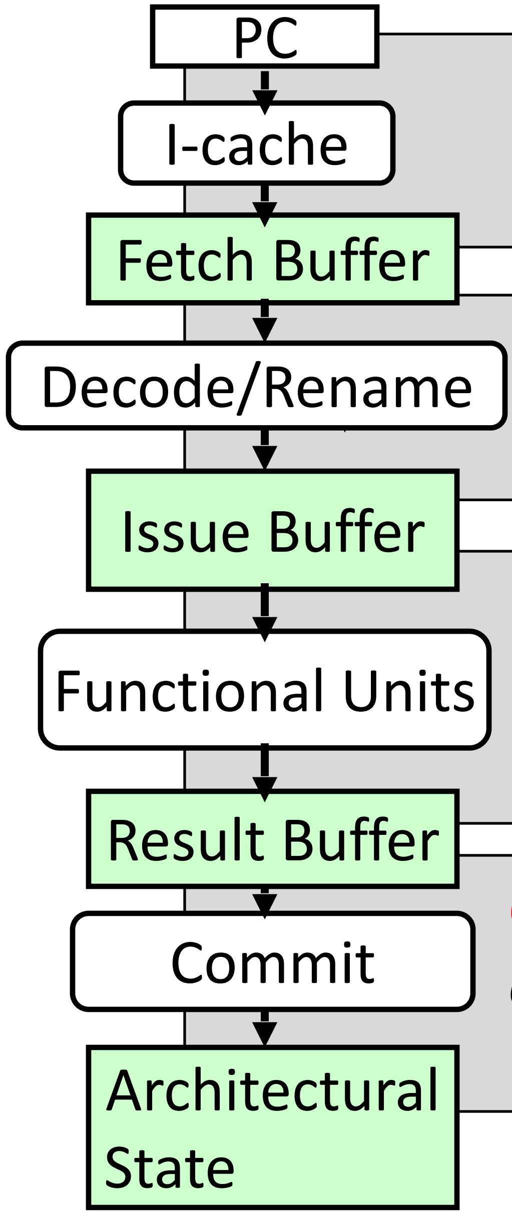


- In-order instruction fetch and decode, and dispatch to reservation stations inside reorder buffer
- Instructions issue from reservation stations out-of-order
- Out-of-order completion, values stored in temporary buffers
- Commit is in-order, checks for traps, and if none updates architectural state

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



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 "retirement"), or takes precise trap.*

Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Krste Asanovic (UCB)
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)