CS 152 Computer Architecture and Engineering

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

Dr. George Michelogiannakis
EECS, University of California at Berkeley
CRD, Lawrence Berkeley National Laboratory

`http://inst.eecs.berkeley.edu/~cs152`

Administrivia

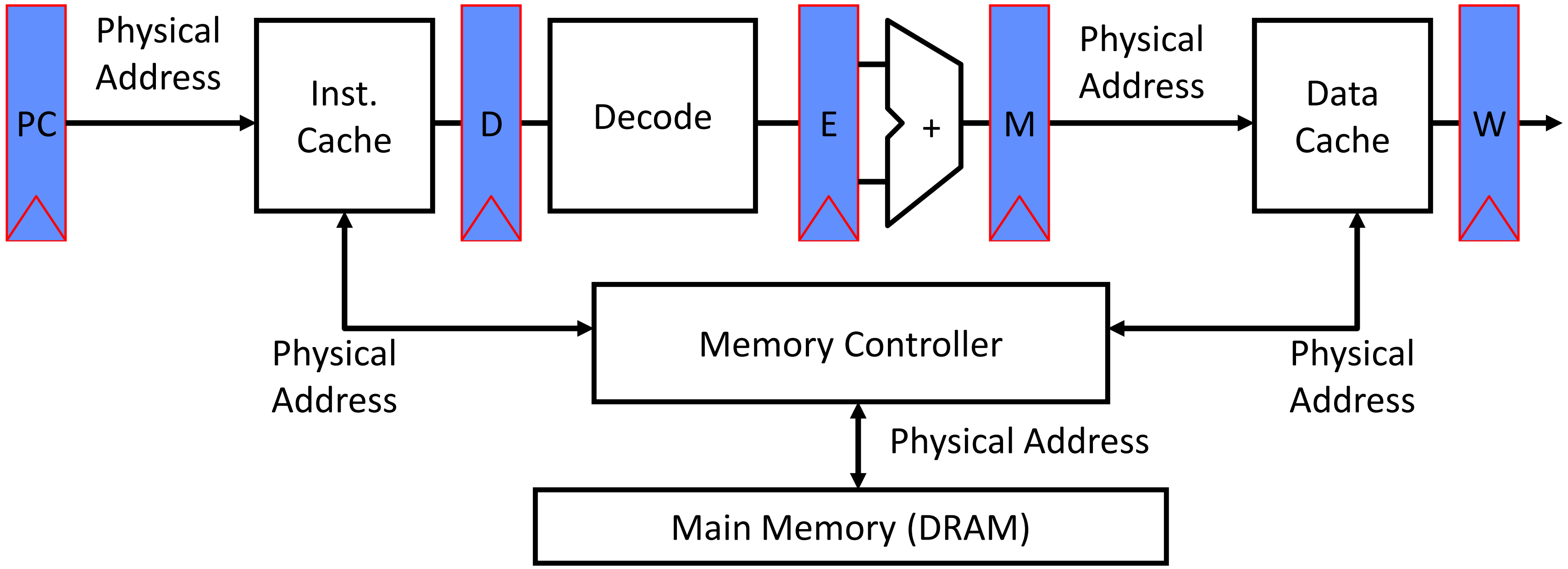
- Graded PS 1 is back
- PS 2 and lab 2 are due Wednesday
  - During class
  - Unless you use a two-day lab extension
- Quiz 2 on module 2 (until last lecture) Monday in a week from now

Last Time in Lecture 9 (End of Module 2)

- Modern page-based virtual memory systems provide:
  - Translation, Protection, Virtual memory.
- 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).

Complex Pipelining: Motivation

- Why would we want more than our in-order pipeline?



Complex Pipelining: Motivation

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

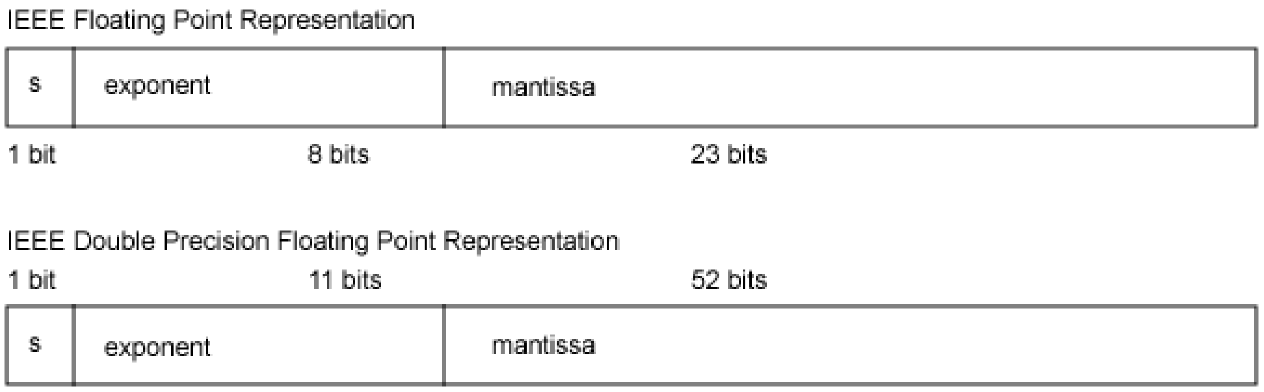
- Long latency or partially pipelined floating-point units
  - Not all instructions are floating point
- Memory systems with variable access time
  - For example cache misses
- Multiple arithmetic and memory units

Floating Point Representation

- IEEE standard 754

Value = $(-1)^s * 1.\text{mantissa} * 2^{(exp-127)}$

Exponent = 0 has special meaning

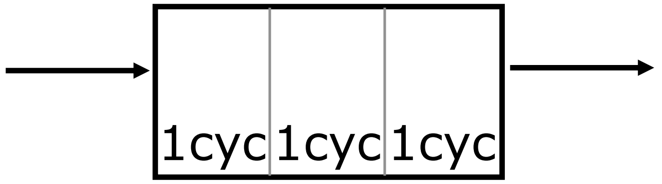

IEEE Floating Point Representation
s
exponent
mantissa
1 bit
8 bits
23 bits
IEEE Double Precision Floating Point Representation
1 bit
11 bits
52 bits
s
exponent
mantissa

Floating-Point Unit (FPU)

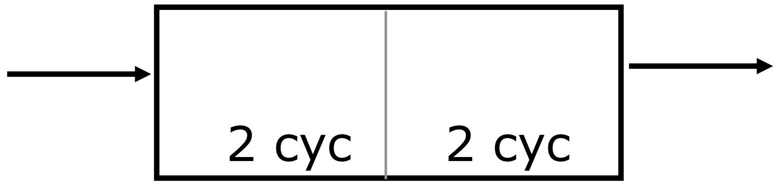
- Much more hardware than an integer unit
  - Single-cycle FPU is a bad idea – why?
  - A simple FPU takes 150,000 gates. Verification complex. Some exceptions specific to floating point.
  - Integer FU to the order of thousands
- Common to have several FPU's
  - Some integer, some floating point
- Common to have different types of FPU's: Fadd, Fmul, Fdiv, …
- An FPU may be pipelined, partially pipelined or not pipelined
- To operate several FPU's concurrently the FP register file needs to have more read and write ports

Functional Unit Characteristics

fully pipelined



partially pipelined



Functional units have internal pipeline registers

⇒ operands are latched when an instruction enters a functional unit
⇒ following instructions are able to write register file during a long-latency operation

Floating-Point ISA

- Interaction between floating-point datapath and integer datapath is determined by ISA
- RISC-V ISA
  - separate register files for FP and Integer instructions
    - the only interaction is via a set of move/convert instructions (some ISA's don't even permit this)
  - separate load/store for FPR's and GPR's (general purpose registers) but both use GPR's for address calculation
  - FP compares write integer registers, then use integer branch

Realistic Memory Systems

Common approaches to improving memory performance:

- Caches - single cycle except in case of a miss

  =>stall

- Banked memory - multiple memory accesses

  => bank conflicts

- split-phase memory operations (separate memory request from response), many in flight

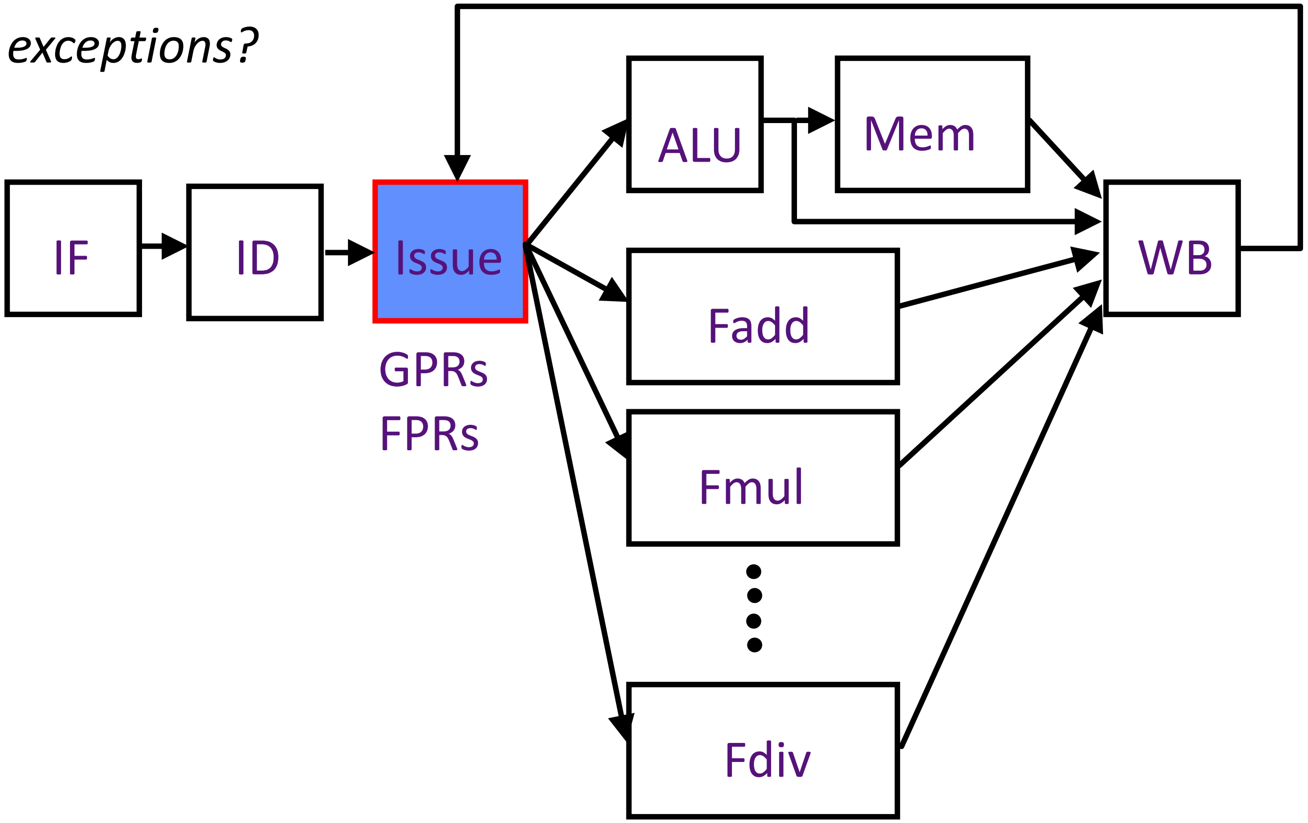
  => out-of-order responses

Latency of access to the main memory is usually much greater than one cycle and often unpredictable

Solving this problem is a central issue in computer architecture

Issues in Complex Pipeline Control

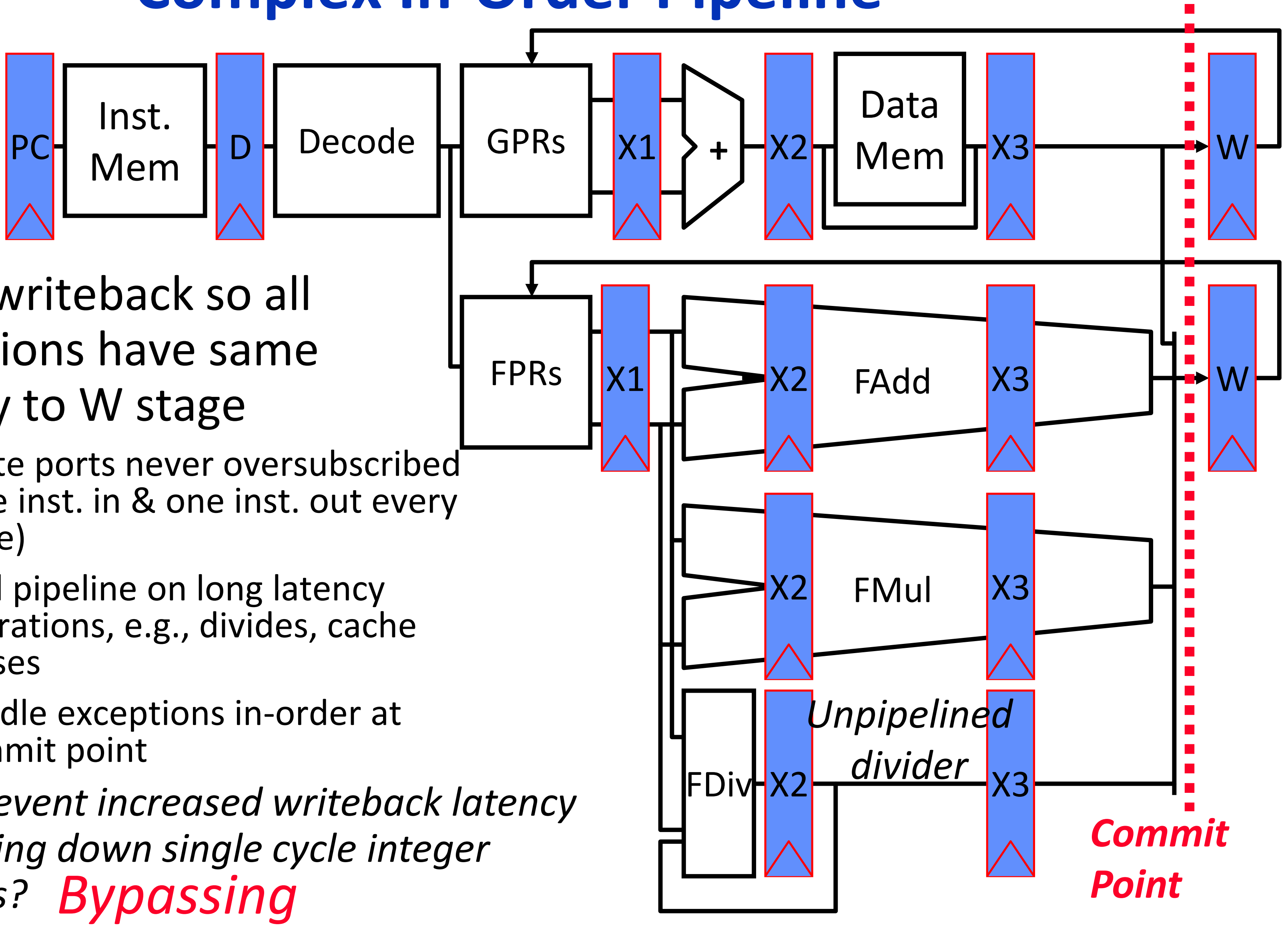
- Structural conflicts at the execution stage
    - If some FPU or memory unit is not pipelined and takes more than one cycle
- Structural conflicts at the write-back stage
    - Due to variable latencies of different functional units
- Out-of-order write hazards
    - Due to variable latencies of different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



Question

- If we issue one instruction per cycle, how can we avoid structural hazards at the writeback stage and out-of-order writeback issues?

Complex In-Order Pipeline



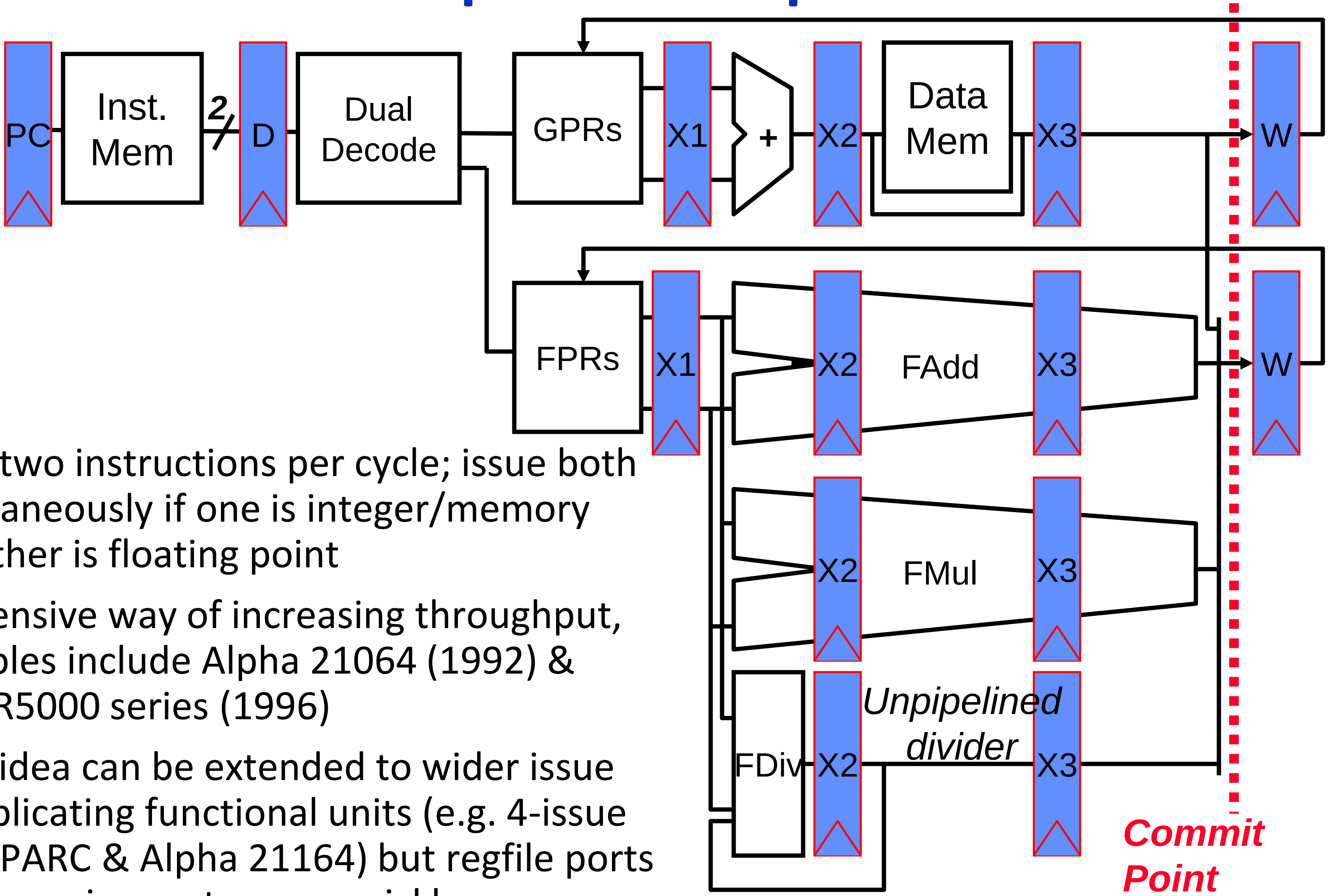
- Delay writeback so all operations have same latency to W stage
  - Write ports never oversubscribed (one inst. in & one inst. out every cycle)
  - Stall pipeline on long latency operations, e.g., divides, cache misses
  - Handle exceptions in-order at commit point

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

Question

- How can we decrease CPI to less than 1?

In-Order Superscalar Pipeline



- Fetch two instructions per cycle; issue both simultaneously if one is integer/memory and other is floating point
- Inexpensive way of increasing throughput, examples include Alpha 21064 (1992) & MIPS R5000 series (1996)
- Same idea can be extended to wider issue by duplicating functional units (e.g. 4-issue UltraSPARC & Alpha 21164) but regfile ports and bypassing costs grow quickly

Types of Data Hazards

Consider executing a sequence of

$$r_k \ <= r_i \ \text{op} \ r_j$$

type of instructions

Data-dependence

$$\begin{tikzcd}[row sep=small, column sep=small]
r_3 \arrow[drr] & <= & r_1 & \text{op} & r_2 \\
r_5 & <= & r_3 & \text{op} & r_4
\end{tikzcd}$$

Read-after-Write (RAW) hazard

Anti-dependence

$$\begin{tikzcd}[row sep=small, column sep=small]
r_3 & <= & r_1 \arrow[dll] & \text{op} & r_2 \\
r_1 & <= & r_4 & \text{op} & r_5
\end{tikzcd}$$

Write-after-Read (WAR) hazard

Output-dependence

$$\begin{tikzcd}[row sep=small, column sep=small]
r_3 \arrow[d, bend right] & <= & r_1 & \text{op} & r_2 \\
r_3 & <= & r_6 & \text{op} & r_7
\end{tikzcd}$$

Write-after-Write (WAW) hazard

Register vs. Memory Dependence

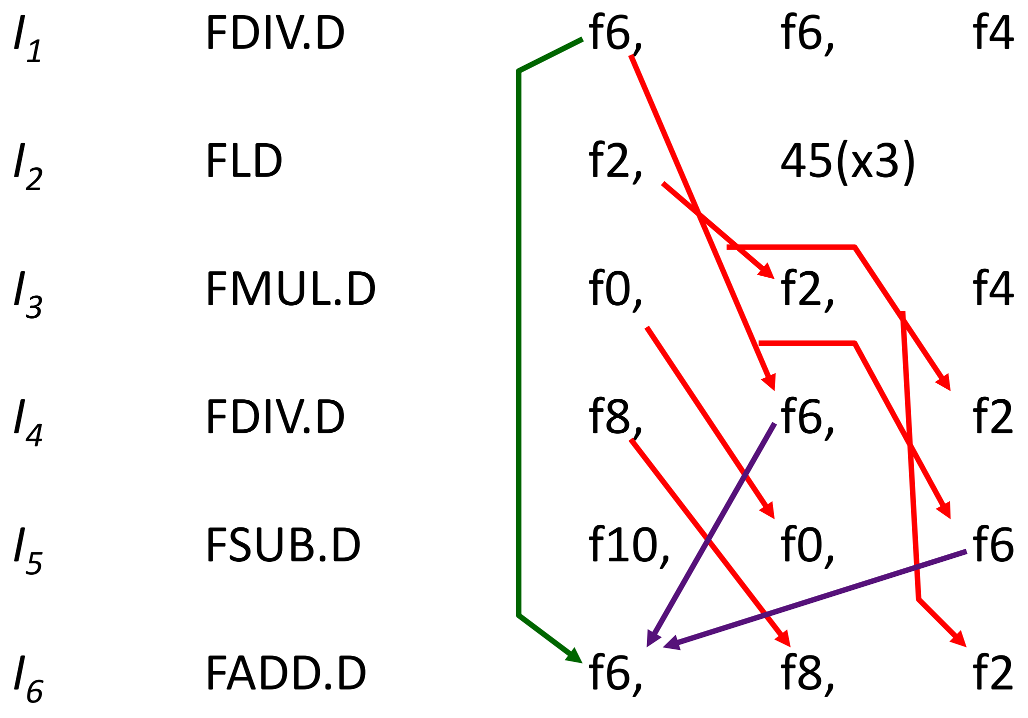
Data hazards due to register operands can be determined at the decode stage, but data hazards due to memory operands can be determined only after computing the effective address

Store: **`M[r1 + disp1] <= r2`**

Load: **`r3 <= M[r4 + disp2]`**

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

Data Hazards: An Example



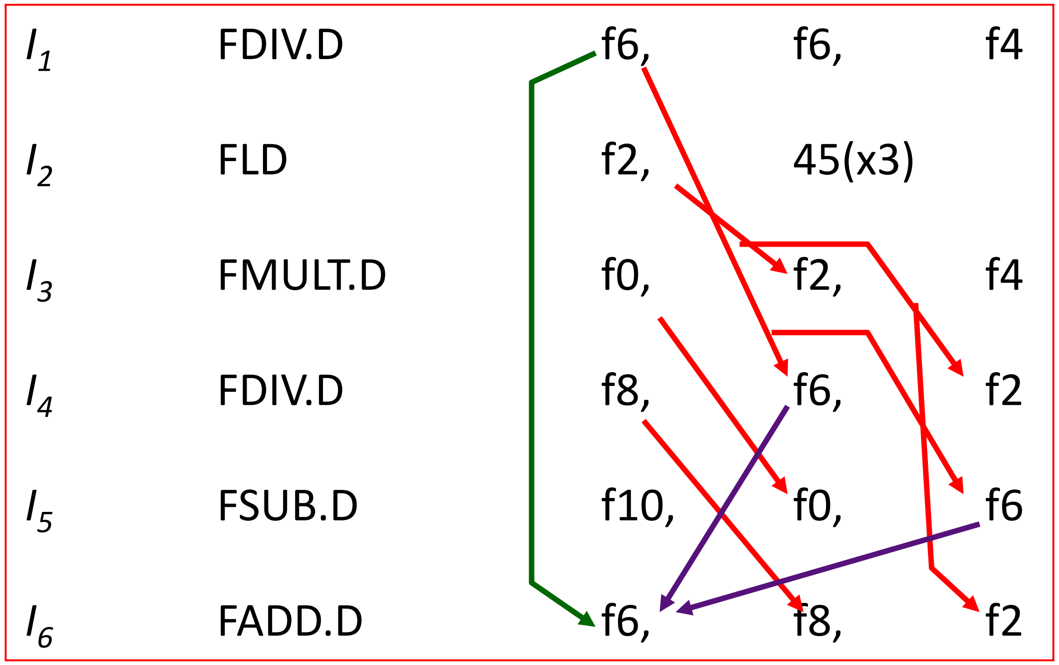
I_1	FDIV.D	f6,	f6,	f4
I_2	FLD	f2,	45(x3)	
I_3	FMUL.D	f0,	f2,	f4
I_4	FDIV.D	f8,	f6,	f2
I_5	FSUB.D	f10,	f0,	f6
I_6	FADD.D	f6,	f8,	f2

RAW Hazards

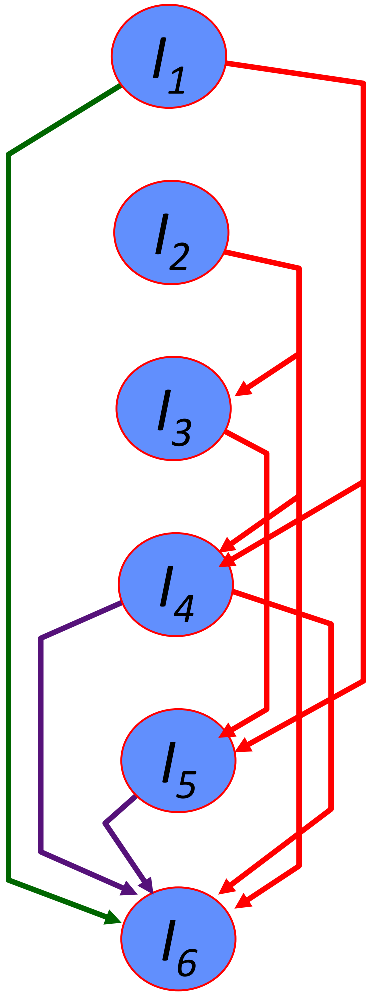
WAR Hazards

WAW Hazards

Instruction Scheduling



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

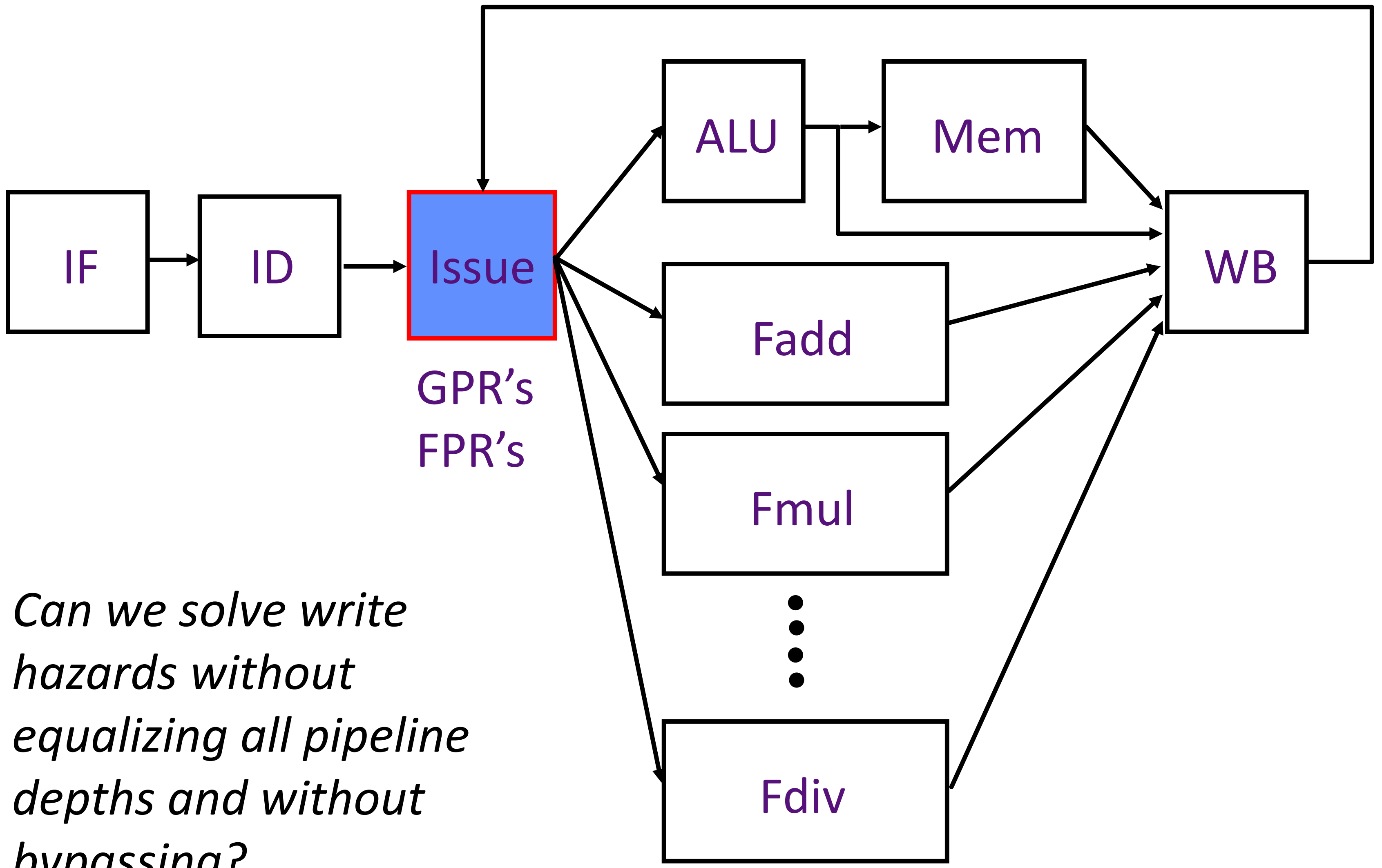
					Latency
I_1	FDIV.D	f6,	f6,	f4	*4*
I_2	FLD	f2,	45(x3)		*1*
I_3	FMULT.Df0,	f2,	f4		*3*
I_4	FDIV.D	f8,	f6,	f2	*4*
I_5	FSUB.D	f10,	f0,	f6	*1*
I_6	FADD.D	f6,	f8,	f2	*1*

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 $\underline{6}$

Underlines are completes

Complex Pipeline



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

When is it Safe to Issue an Instruction?

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

The following checks need to be made before the Issue stage can dispatch an instruction

- Is the required function unit available?
- Is the input data available? => RAW?
- Is it safe to write the destination? => WAR? WAW?
- Is there a structural conflict at the WB stage?

A Data Structure for Correct Issues

Keeps track of the status of Functional Units

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

The instruction i at the Issue stage consults this table

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

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

Simplifying the Data Structure Assuming In-order Issue

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

Can the dispatched instruction cause a

WAR hazard ?

NO: Operands read at issue

WAW hazard ?

YES: Out-of-order completion

Simplifying the Data Structure

- No WAR hazard

  => no need to keep src1 and src2

- The Issue stage does not dispatch an instruction in case of a WAW hazard

  => a register name can occur at most once in the dest column

- WP[reg#] : a bit-vector to record the registers for which writes are pending
  - These bits are set by the Issue stage and cleared by the WB stage

  => Each pipeline stage in the FU's must carry the dest field and a flag to indicate if it is valid "the (we, ws) pair"

Scoreboard for In-order Issues

Busy[FU#] : a bit-vector to indicate FU's availability.
(FU = Int, Add, Mult, Div)

These bits are hardwired to FU's.

WP[reg#] : a bit-vector to record the registers for which writes are pending.

These bits are set by Issue stage and cleared by WB stage

Issue checks the instruction (opcode dest src1 src2) against the scoreboard (Busy & WP) to dispatch

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

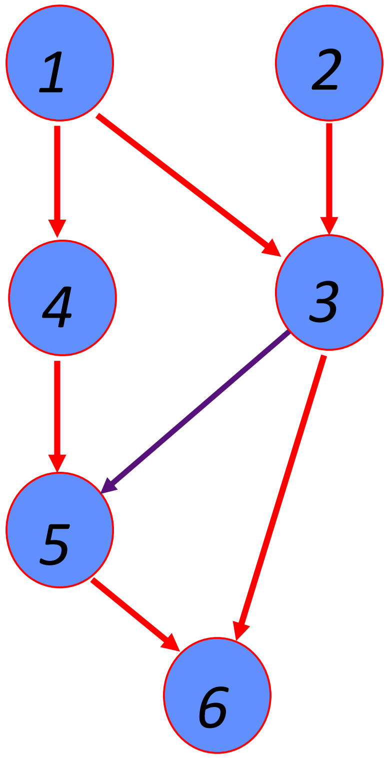
Scoreboard Dynamics

	Functional Unit Status										Registers Reserved for Writes	
	Int(1)	Add(1)	Mult(3)			Div(4)				WB		
t0	I_1					f6					f6	
t1	I_2 f2						f6				f6, f2	
t2								f6		f2	f6, f2	$\underline{I_2}$
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*

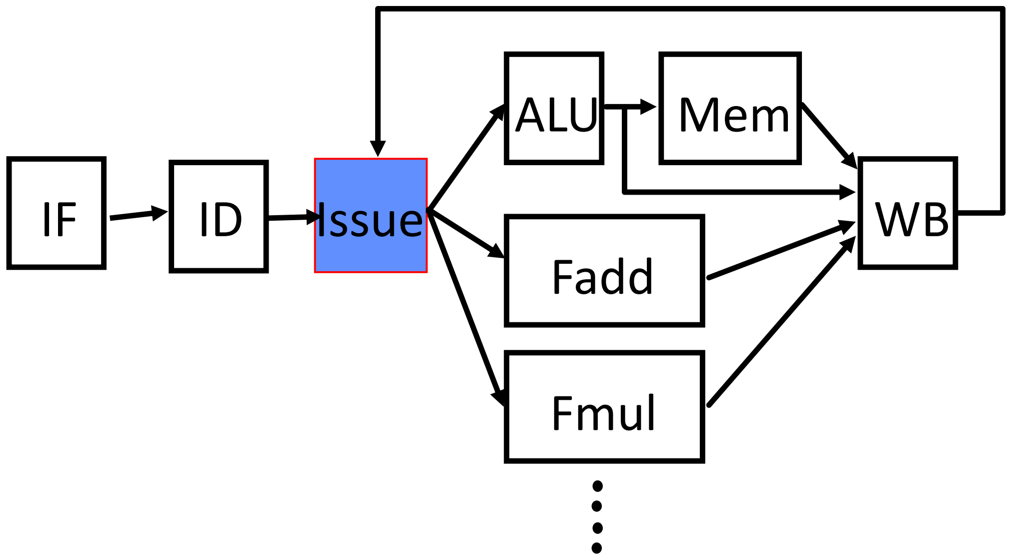
					latency
1	FLD	f2,	34(x2)		*1*
2	FLD	f4,	45(x3)		*long*
3	FMULT.D	f6,	f4,	f2	*3*
4	FSUB.D	f8,	f2,	f2	*1*
5	FDIV.D	f4,	f2,	f8	*4*
6	FADD.D	f10,	f6,	f4	*1*



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

In-order issue restriction prevents instruction 4 from being dispatched

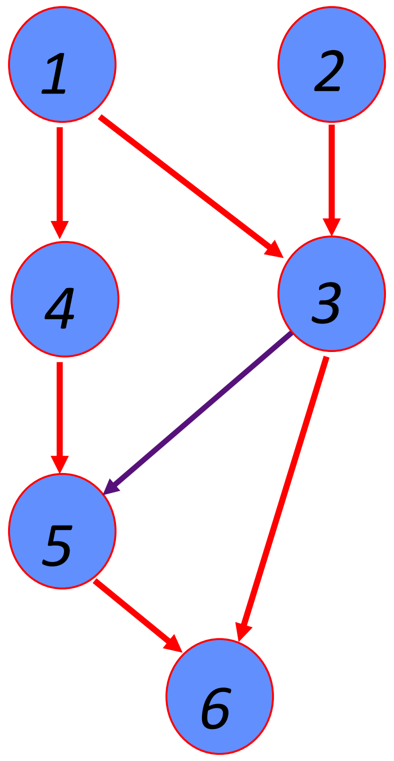
Out-of-Order Issue



- 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 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?

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.

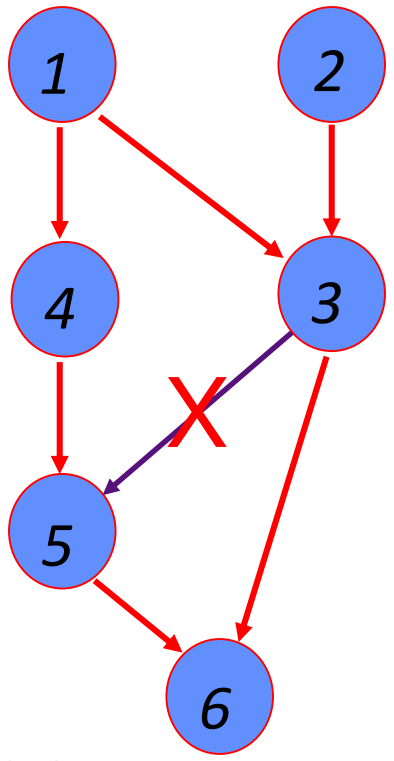
IBM 360 had only 4 floating-point registers

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 and Out-of-Order

					latency
1	FLD	f2,	34(x2)		*1*
2	FLD	f4,	45(x3)		*long*
3	FMULT.Df6,	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 $\underline{6}$

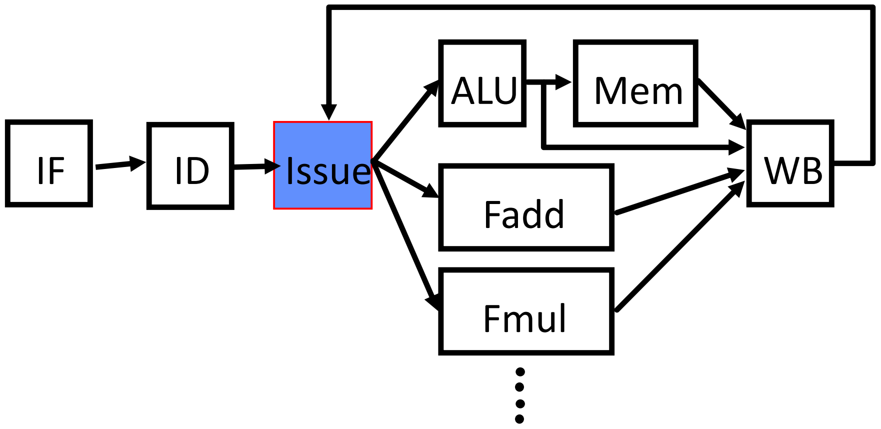
Out-of-order: 1 (2,$\underline{1}$) 4 $\underline{4}$ 5 . . . $\underline{2}$ (3,$\underline{5}$) $\underline{3}$ 6 $\underline{6}$

Any antidependence can be eliminated by renaming.

(renaming => additional storage)

Can it be done in hardware? *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

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252