Complex Pipelining

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

Complex Pipelining: Motivation

Instruction pipelining becomes complex when we want high performance in the presence of

- Multi-cycle operations, for example:
  - Full or partially pipelined floating-point units, or
  - Long-latency operations, e.g., divides

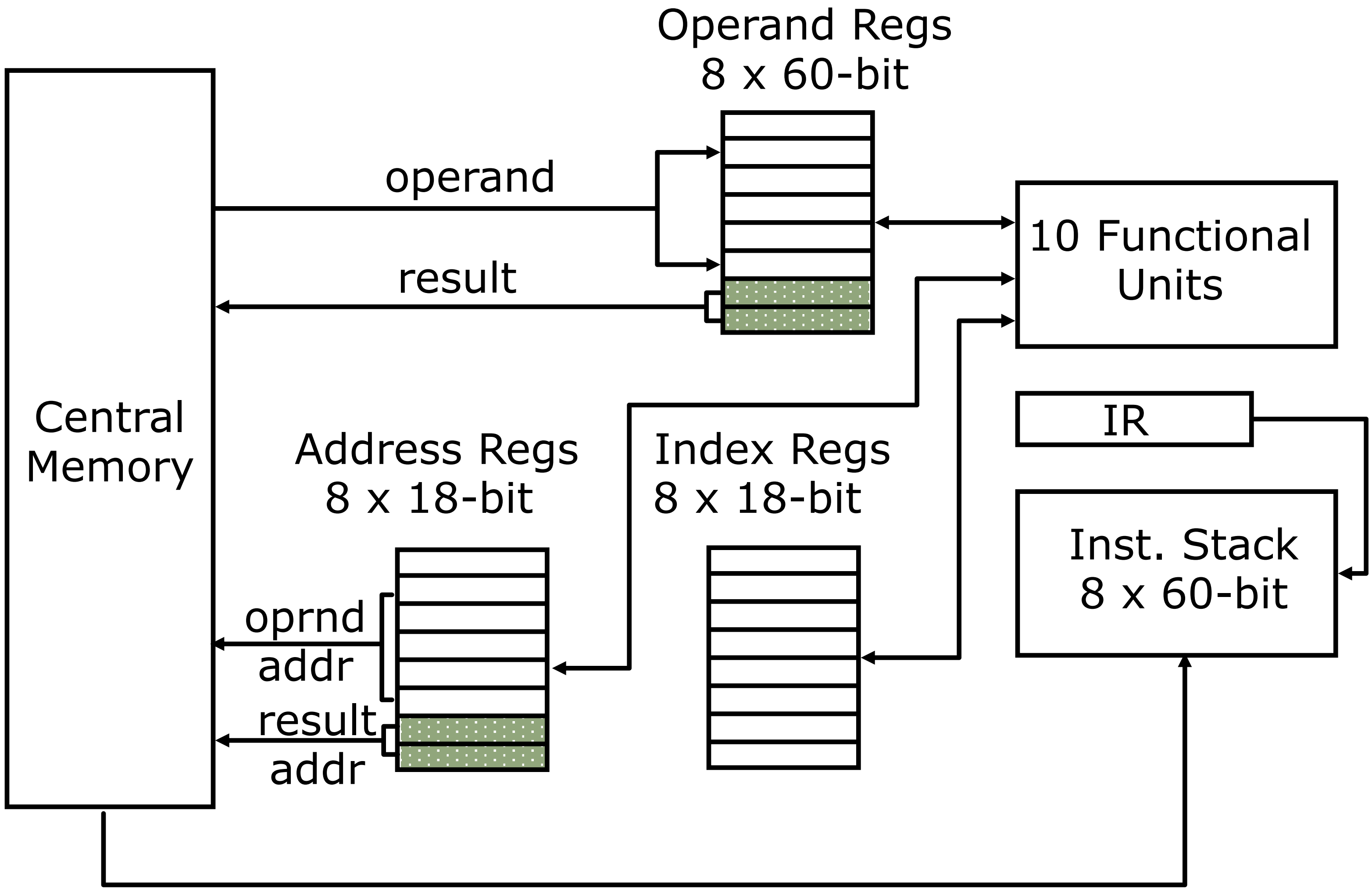
- Variable-latency operations, for example:
  - Memory systems with variable access time

- Replicated functional units, for example:
  - Multiple floating-point or memory units

CDC 6600
Seymour Cray, 1963

- A fast pipelined machine with 60-bit words
  - 128 Kword main memory capacity, 32 banks
- Ten functional units (parallel, unpipelined)
  - Floating Point: adder, 2 multipliers, divider
  - Integer: adder, 2 incrementers, ...
- Hardwired control
- Dynamic scheduling of instructions using a scoreboard
- Ten Peripheral Processors for Input/Output
  - A fast multi-threaded 12-bit integer ALU
- Very fast clock, 10 MHz (FP add in 4 clocks)
- >400,000 transistors, 750 sq. ft., 5 tons, 150 kW, new freon-based cooling technology
- Fastest machine in world for 5 years (until CDC 7600)
  - Over 100 sold ($7-10M each)

CDC 6600: Datapath



CDC 6600: A Load/Store Architecture

- Separate instructions to manipulate three types of registers
  - 8 60-bit data registers (X)
  - 8 18-bit address registers (A)
  - 8 18-bit index registers (B)

- All arithmetic and logic instructions are reg-to-reg

6	3	3	3
opcode	i	j	k

Ri ← (Rj) op (Rk)

- Only Load and Store instructions refer to memory!

6	3	3	18
opcode	i	j	disp

Ri ← M[(Rj) + disp]

  - Touching address registers 1 to 5 initiates a load
    6 to 7 initiates a store
    - *very useful for vector operations*

CDC6600: Vector Addition

```
        B1  ←  - n
loop:   JZE   B1, exit
        A1  ←  B1 + a1      load into X1
        A2  ←  B1 + b1      load into X2
        X6  ←  X1 + X2
        A6  ←  B1 + c1      store X6
        B1  ←  B1 + 1
        jump loop
```

Ai = address register
Bi = index register
Xi = data register

more on vector processing later…

We will present complex pipelining issues more abstractly …

Floating Point ISA

Interaction between the Floating point datapath and the Integer datapath is determined largely by the ISA

RISC-V ISA (with F/D extensions)

- separate register files for FP and Integer instructions
  the only interaction is via a set of move instructions (some ISAs don't even permit this)
- separate load/store for FPRs and GPRs, but both use GPRs for address calculation
- branches only take GPRs as sources; FP compare instructions write result to a GPR, which can then be used in a branch

Floating Point Unit

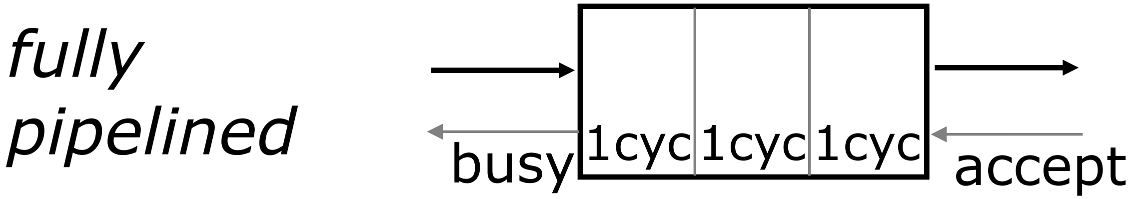
Much more hardware than an integer unit

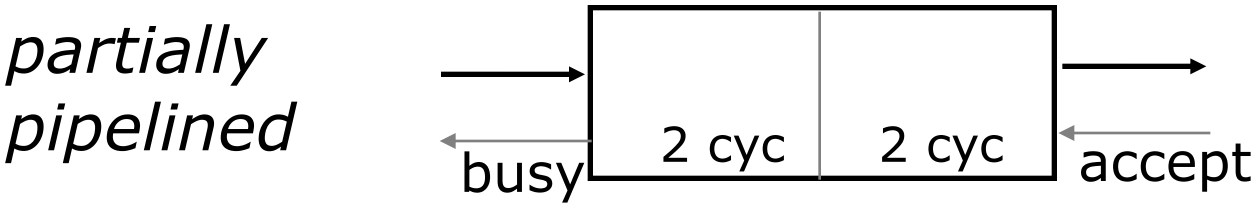
Single-cycle floating point unit is a bad idea - *why?*

- it is common to have several floating point units
- it is common to have different types of FPUs
  Fadd, Fmul, Fdiv, ...
- an FPU may be pipelined, partially pipelined or not pipelined

To operate several FPUs concurrently, the register file needs to have more read and write ports

Functional Unit Characteristics





Functional units have internal pipeline registers

⇒ operands are latched when an instruction enters a functional unit

⇒ inputs to a functional unit (e.g., register file) can change during a long latency operation

Realistic Memory Systems

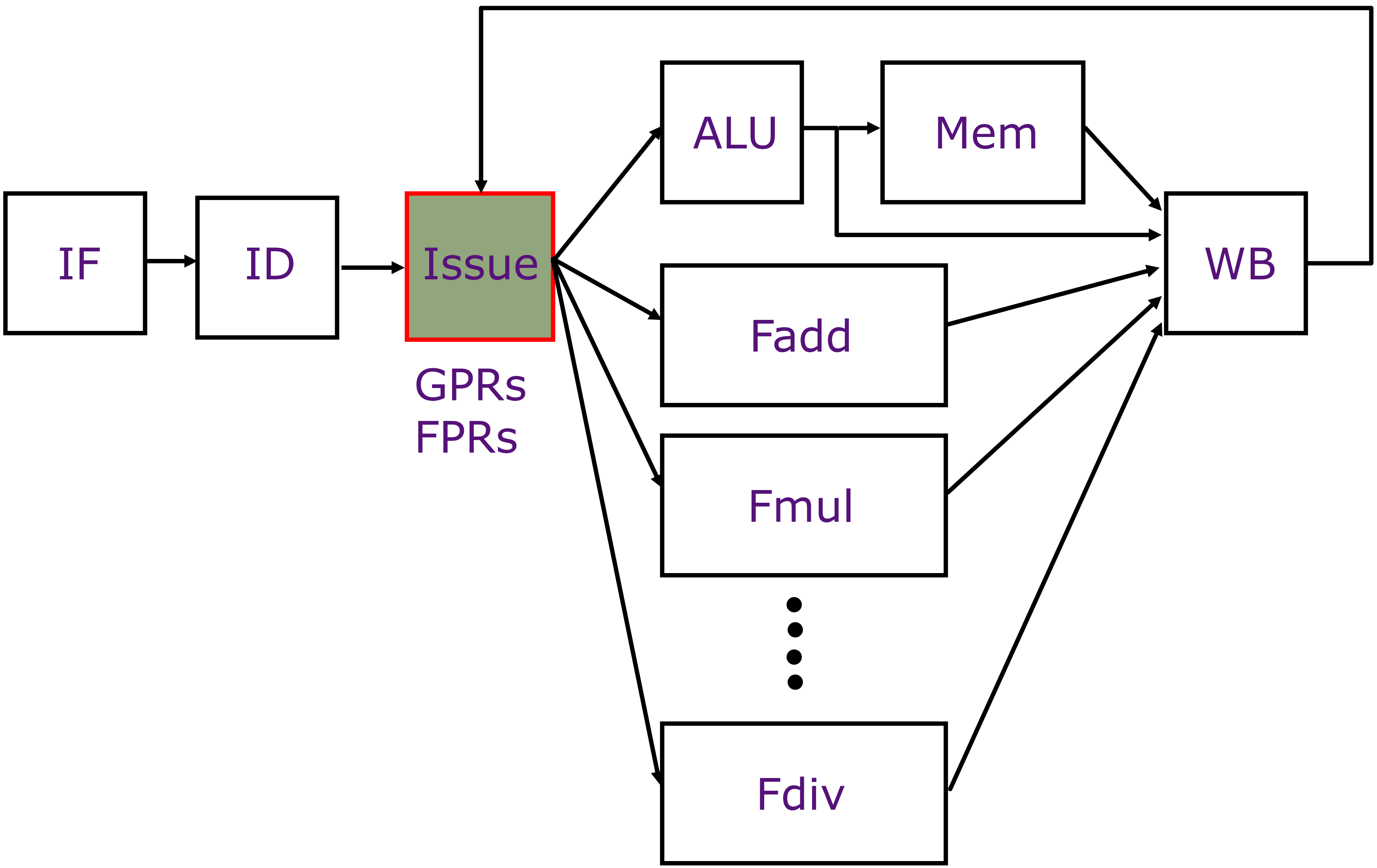
Latency of access to the main memory is usually much higher than one cycle and often unpredictable

Solving this problem is a central issue in computer architecture

Common approaches to improving memory performance

- separate instruction and data memory ports
  - $\Rightarrow$ *no self-modifying code*
- caches
  - *single cycle except in case of a miss* $\Rightarrow$ *stall*
- interleaved memory
  - *multiple memory accesses* $\Rightarrow$ *bank conflicts*
- split-phase memory operations
  - $\Rightarrow$ *out-of-order responses*

Complex Pipeline Structure



Complex Pipeline Control Issues

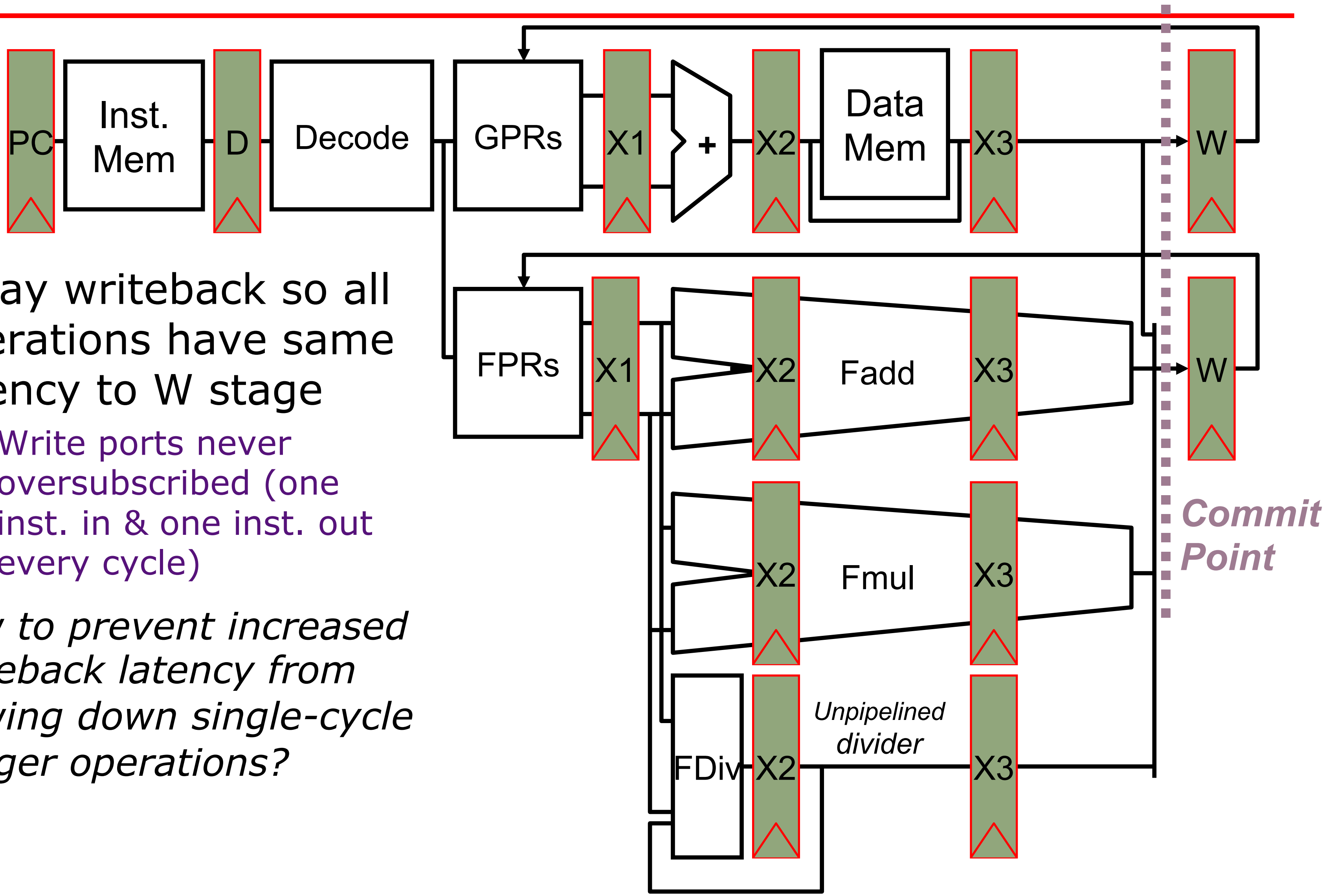
- Structural hazards at the execution stage if some FPU or memory unit is not pipelined and takes more than one cycle

- Structural hazards at the write-back stage due to variable latencies of different function units

- Out-of-order write hazards due to variable latencies of different function units

- How to handle exceptions?

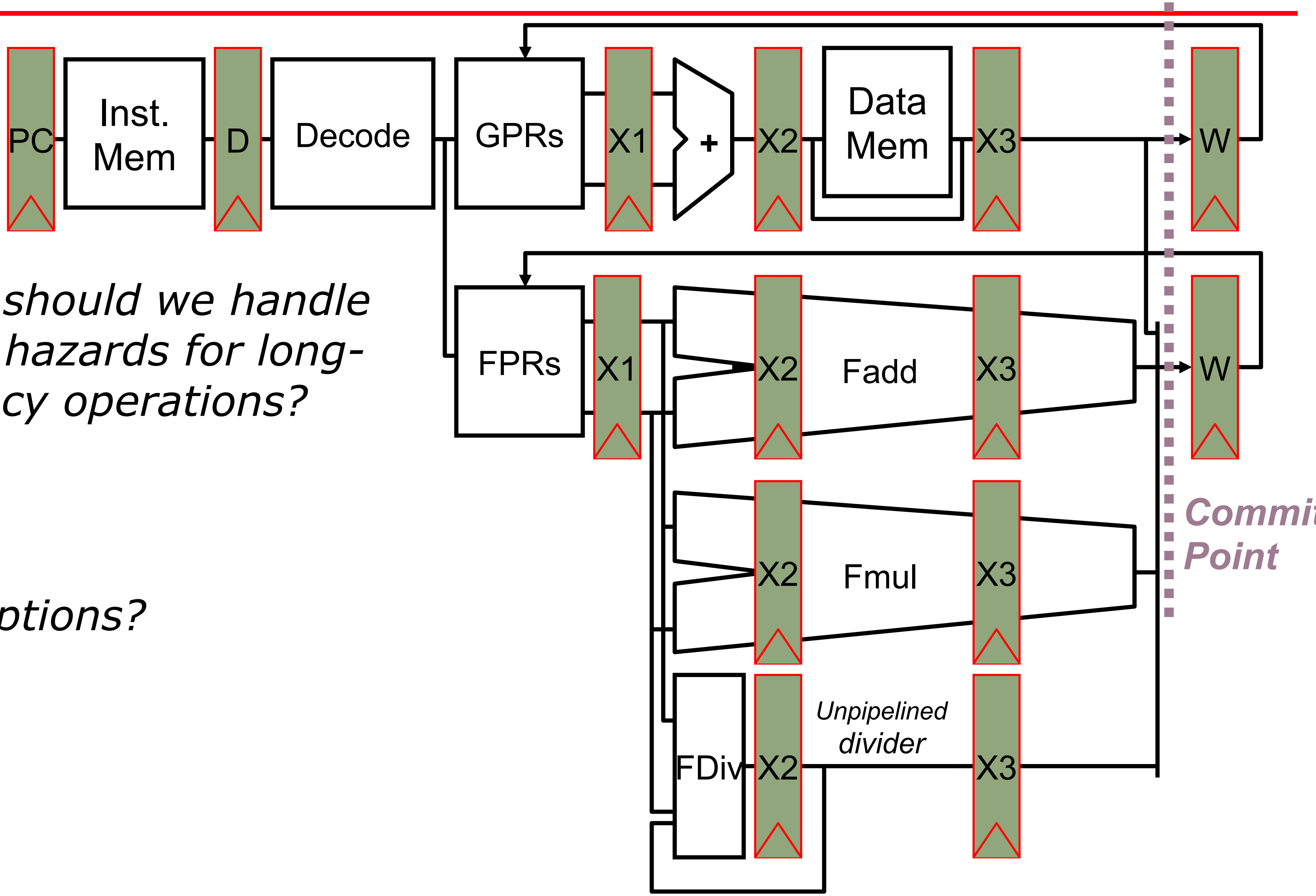
Complex In-Order Pipeline



- Delay writeback so all operations have same latency to W stage
  - Write ports never oversubscribed (one inst. in & one inst. out every cycle)

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

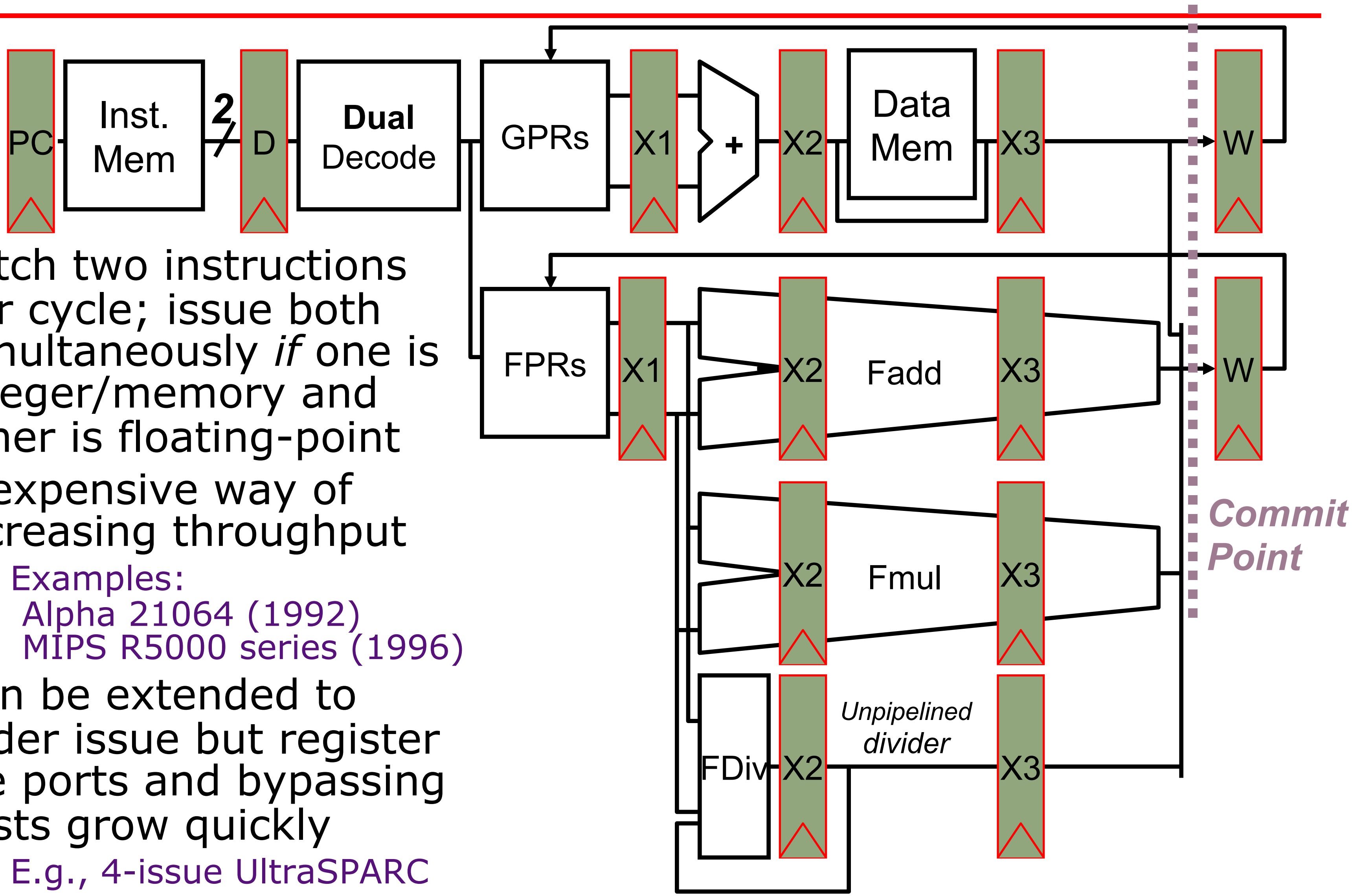
Complex In-Order Pipeline



How should we handle data hazards for long-latency operations?

Exceptions?

Superscalar In-Order 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:
    Alpha 21064 (1992)
    MIPS R5000 series (1996)
- Can be extended to wider issue but register file ports and bypassing costs grow quickly
  - E.g., 4-issue UltraSPARC

Dependence Analysis

Needed to Exploit Instruction-level Parallelism

Types of Data Hazards

Consider executing a sequence of

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

type of instructions

Data-dependence

$r_3 \leftarrow (r_1) \text{ op } (r_2)$ — Read-after-Write
$r_5 \leftarrow (r_3) \text{ op } (r_4)$ — (RAW) hazard

Anti-dependence

$r_3 \leftarrow (r_1) \text{ op } (r_2)$ — Write-after-Read
$r_1 \leftarrow (r_4) \text{ op } (r_5)$ — (WAR) hazard

Output-dependence

$r_3 \leftarrow (r_1) \text{ op } (r_2)$ — Write-after-Write
$r_3 \leftarrow (r_6) \text{ op } (r_7)$ — (WAW) hazard

Detecting Data Hazards

Range and Domain of instruction i

R(i) = Registers (or other storage) modified by instruction i

D(i) = Registers (or other storage) read by instruction i

Suppose instruction j follows instruction i in the program order. Executing instruction j before the effect of instruction i has taken place can cause a

RAW hazard if $R(i) \cap D(j) \neq \emptyset$

WAR hazard if $D(i) \cap R(j) \neq \emptyset$

WAW hazard if $R(i) \cap R(j) \neq \emptyset$

Register vs. Memory Data Dependences

- 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[(a1) + offset1] $\leftarrow$ (a2)
load	a3 $\leftarrow$ M[(a4) + offset2]

Does (a1) + offset1 == (a4) + offset2 ?

In lecture 10, we'll see how to handle memory dependencies
For now, we focus only on register dependencies

Data Hazards: An Example

I_1	FDIV.D	f6,	f6,	f4
I_2	FLD	f2,	45(r3)	
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

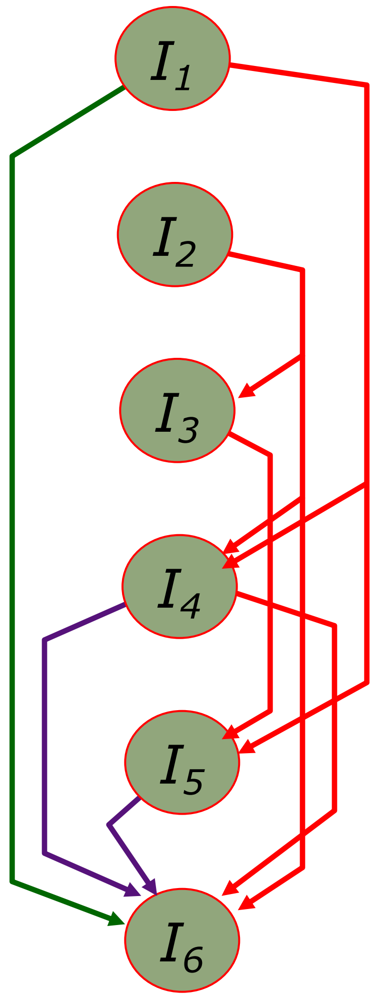
RAW Hazards

WAR Hazards

WAW Hazards

Instruction Scheduling

I_1	FDIV.D	f6,	f6,	f4
I_2	FLD	f2,	45(r3)	
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



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

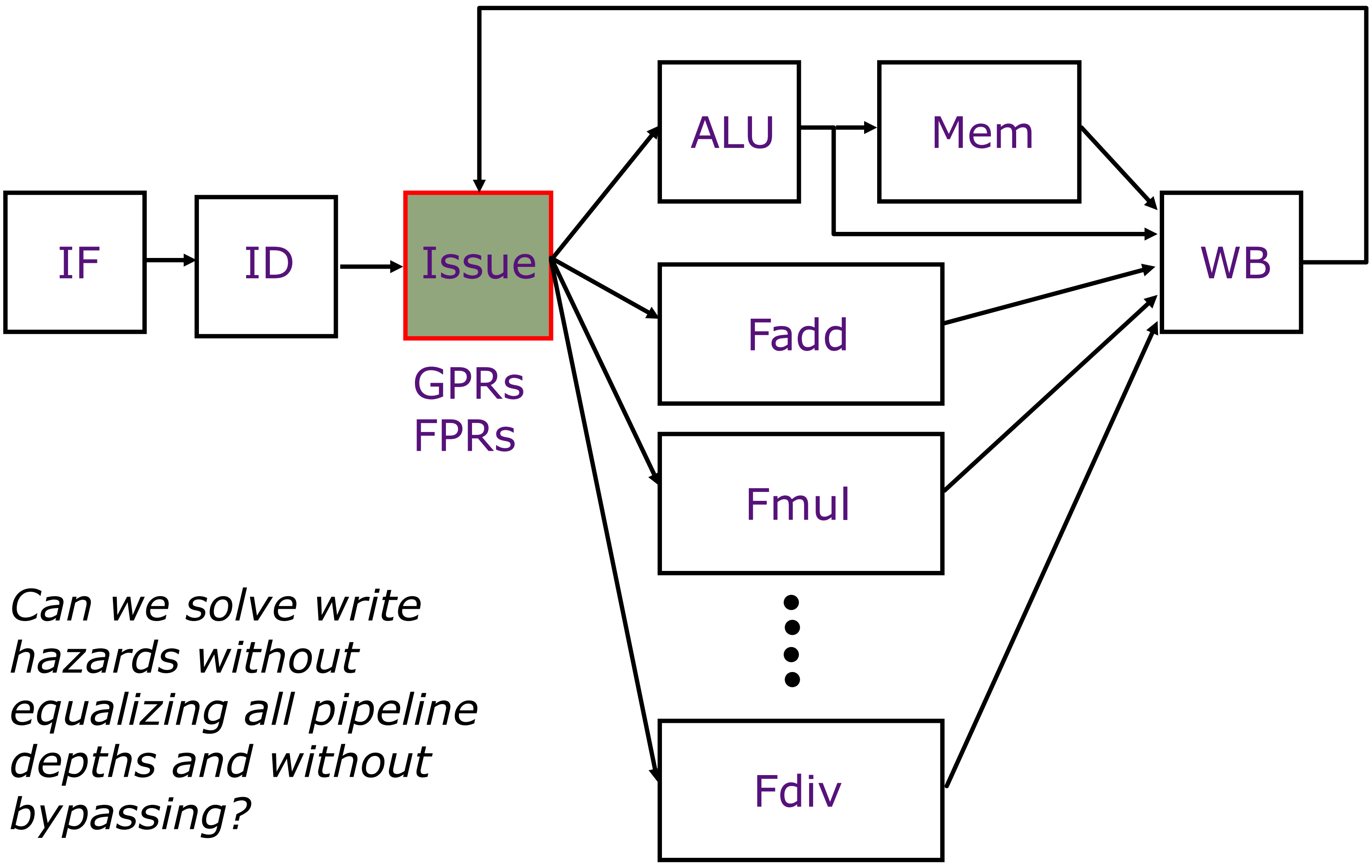
					Latency
I_1	FDIV.D	f6,	f6,	f4	*4*
I_2	FLD	f2,	45(r3)		*1*
I_3	FMUL.D	f0,	f2,	f4	*3*
I_4	FDIV.D	f8,	f6,	f2	*4*
I_5	FSUB.D	f10,	f0,	f6	*1*
I_6	FADD.D	f6,	f8,	f2	*1*

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
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			

What problems can out-of-order comp cause?

Scoreboard: A Hardware Data Structure to Detect Hazards Dynamically

Complex Pipeline



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

When is it Safe to Issue an Instruction?

- Approach: Stall issue until sure that issuing will cause no dependence problems…

- 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? $\Rightarrow$ RAW?
  - Is it safe to write the destination? $\Rightarrow$ 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?
RAW?
WAR?
WAW?

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 if the required FU is busy
- Suppose operands are latched by the functional unit on issue

Can the dispatched instruction cause a

WAR hazard?

WAW hazard?

Name	*Busy*	*Op*	*Dest*	*Src1*	*Src2*
Int					
Mem					
Add1					
......					

Write Pending

No read is pending after issue

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
  - Can be encoded as a bit vector

Name	*Busy*	*Op*	*Dest*	*Src1*	*Src2*
Int					
Mem					
Add1					
……					

Write Pending

No read is pending after issue

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 to true by the Issue stage and set to false by the WB stage

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

FU available?
RAW?
WAR?
WAW?

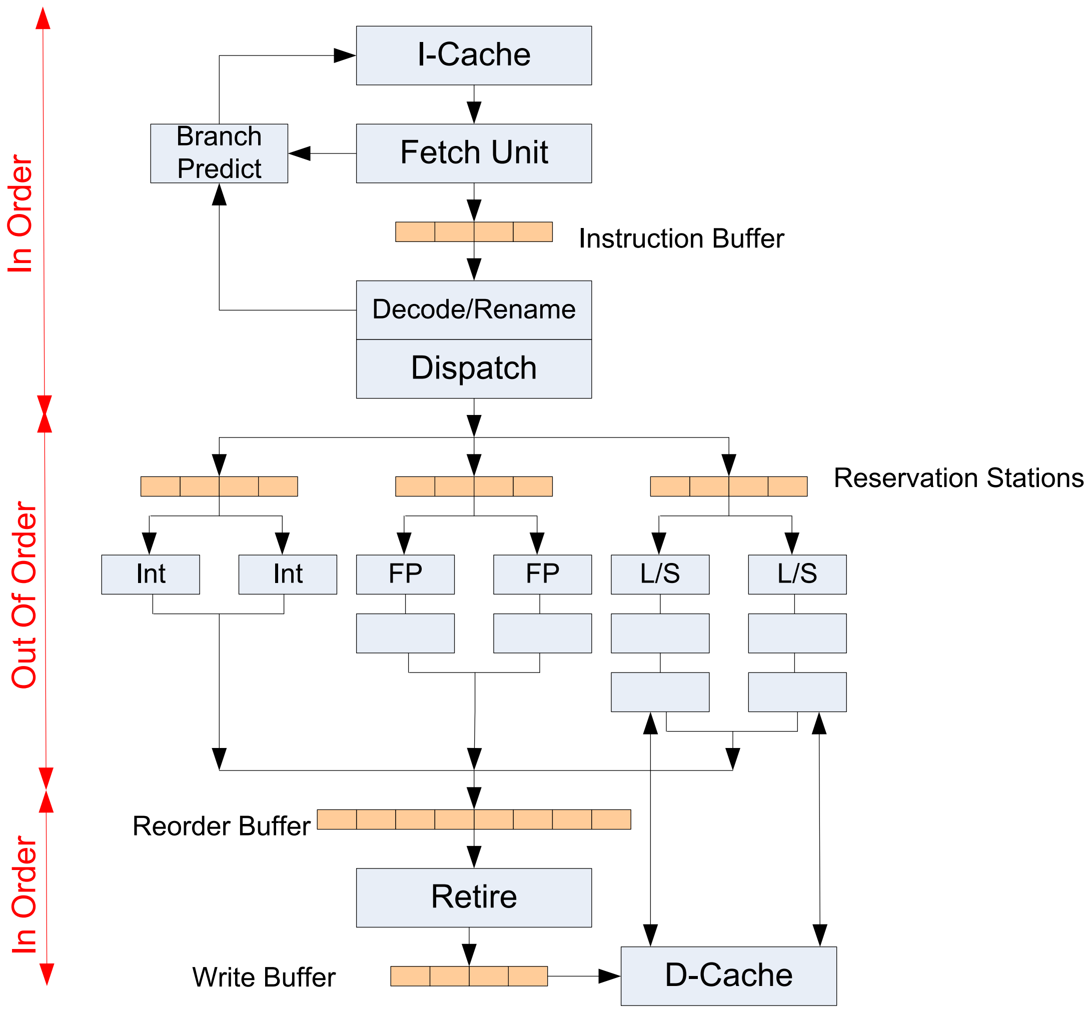
Scoreboard Dynamics

	Issue time	Functional Unit Status: Int(1)	Add(1)	Mult(3)			Div(4)				WB	Registers Reserved for Writes	WB time
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(r3)	
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

Check Busy[Fu#]
Check WP[src1, src2]
Check WP[dest]

Preview: Anatomy of a Modern Out-of-Order Superscalar Core



- L06 (Today): Complex pipes w/ in-order issue
- L07: Out-of-order exec & renaming
- L08: Branch prediction
- L09: Speculative execution and recovery
- L10: Advanced Memory Ops