Complex Pipelining

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

Based on the material prepared by
Arvind and Krste Asanovic

Complex Pipelining: Motivation

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

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

Floating Point ISA

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

MIPS ISA

- separate register files for FP and Integer instructions
  the only interaction is via a set of move instructions (some ISA's don't even permit this)
- separate load/store for FPR's and GPR's but both use GPR's for address calculation
- separate conditions for branches
  FP branches are defined in terms of condition codes

Floating Point Unit

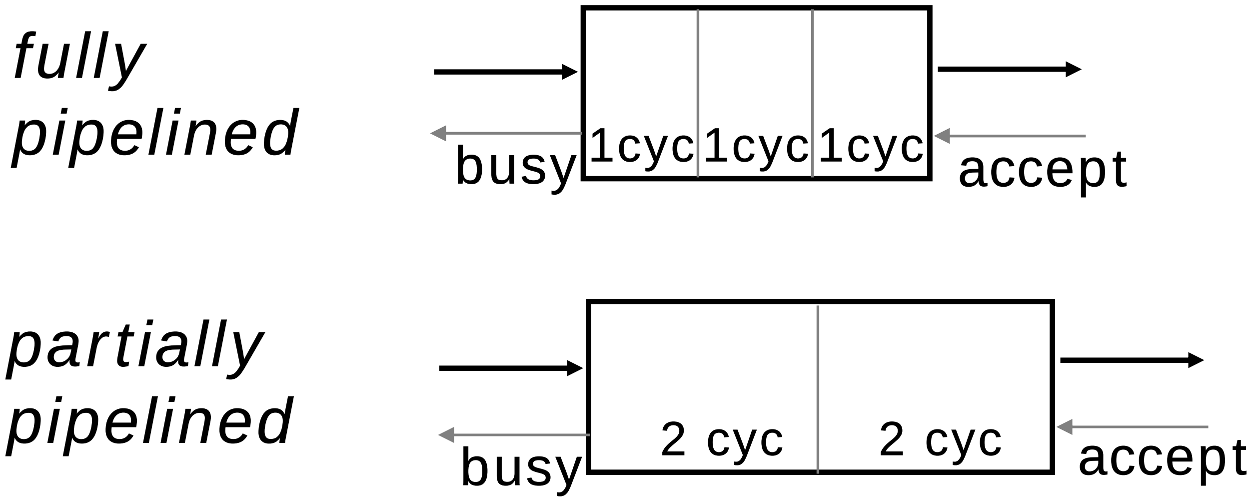
Much more hardware than an integer unit

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 FPU's
  Fadd, Fmul, Fdiv, ...
- an FPU may be pipelined, partially pipelined or not pipelined

To operate several FPU's concurrently the register file needs to have more read and write ports

Function Unit Characteristics



Function units have internal pipeline registers

⇒ operands are latched when an instruction enters a function unit

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

Realistic Memory Systems

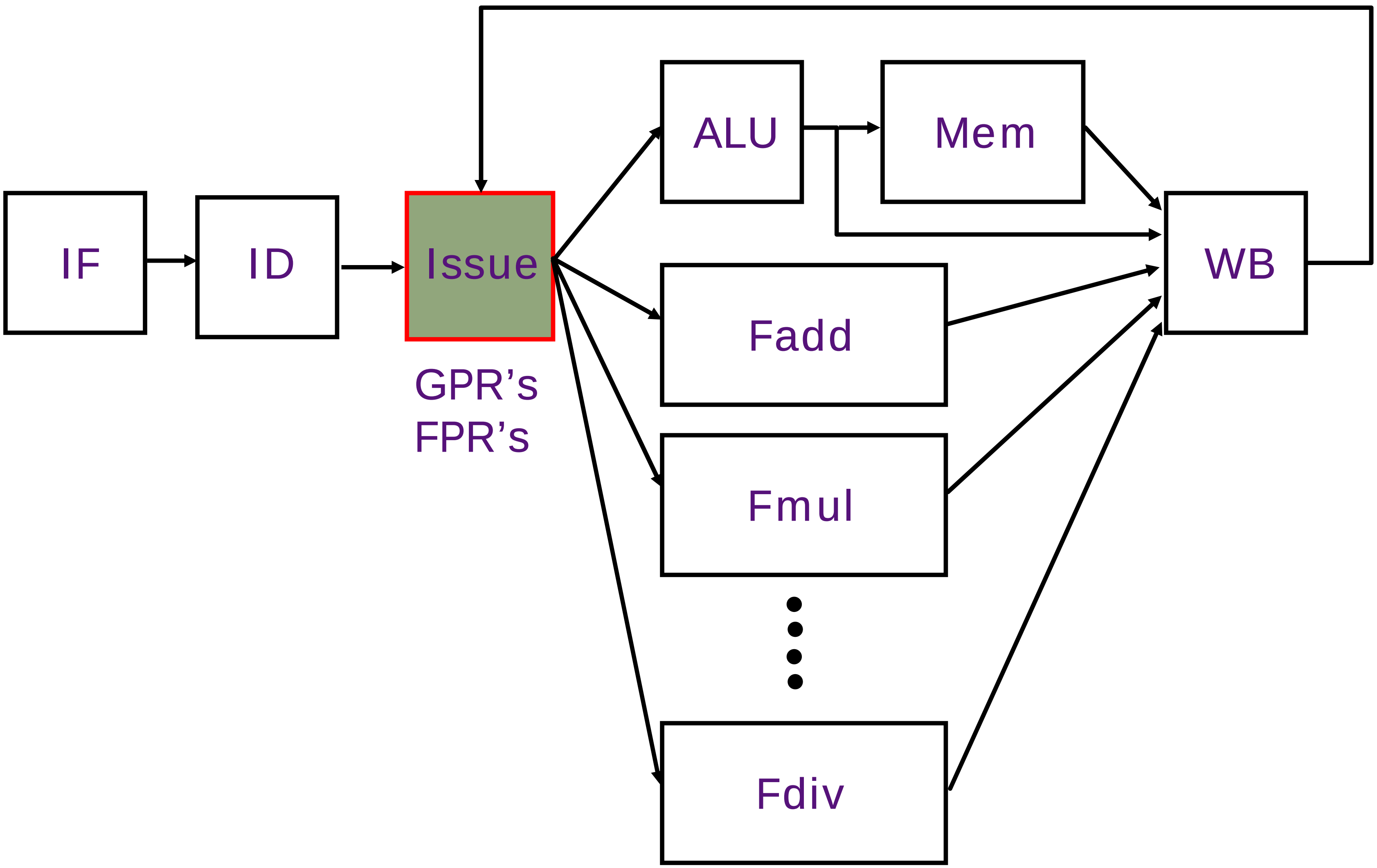
Latency of access to the main memory is usually much greater than one cycle and often unpredictable

Solving this problem is a central issue in computer architecture

Common approaches to improving memory performance

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

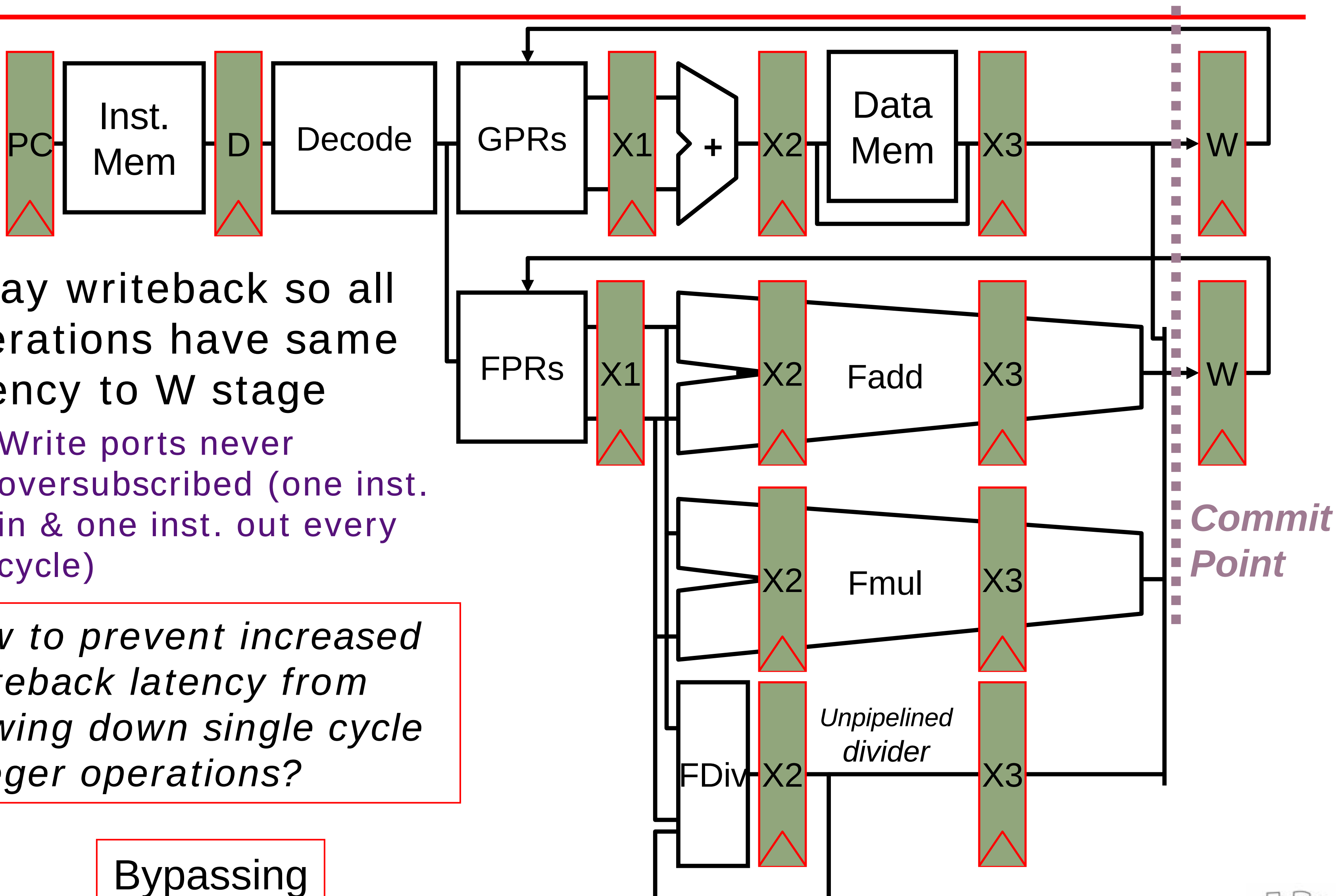
Complex Pipeline Structure



Complex Pipeline Control Issues

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

Complex In-Order Pipeline

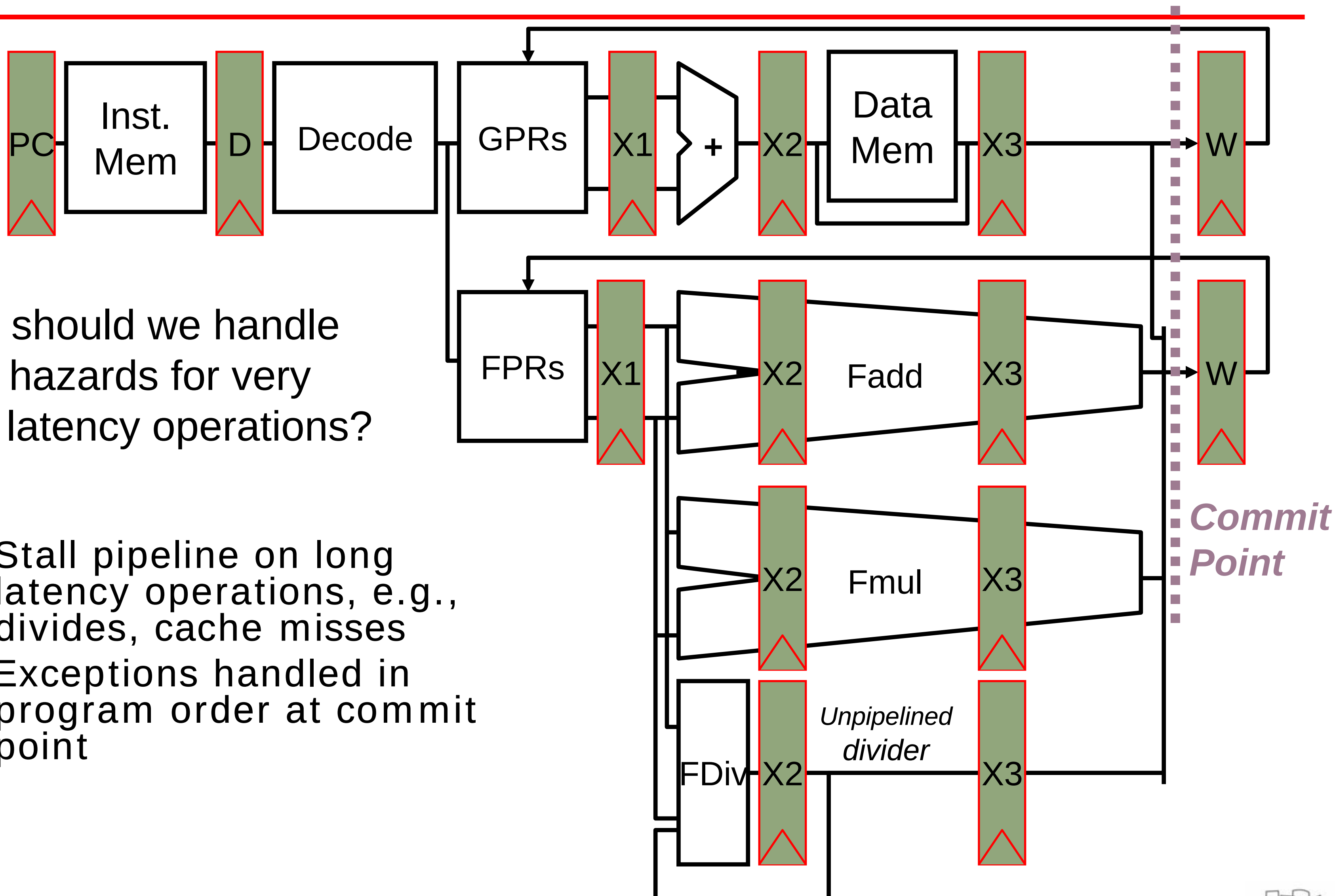


- Delay writeback so all operations have same latency to W stage
  - Write ports never oversubscribed (one inst. in & one inst. out every cycle)

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

Bypassing

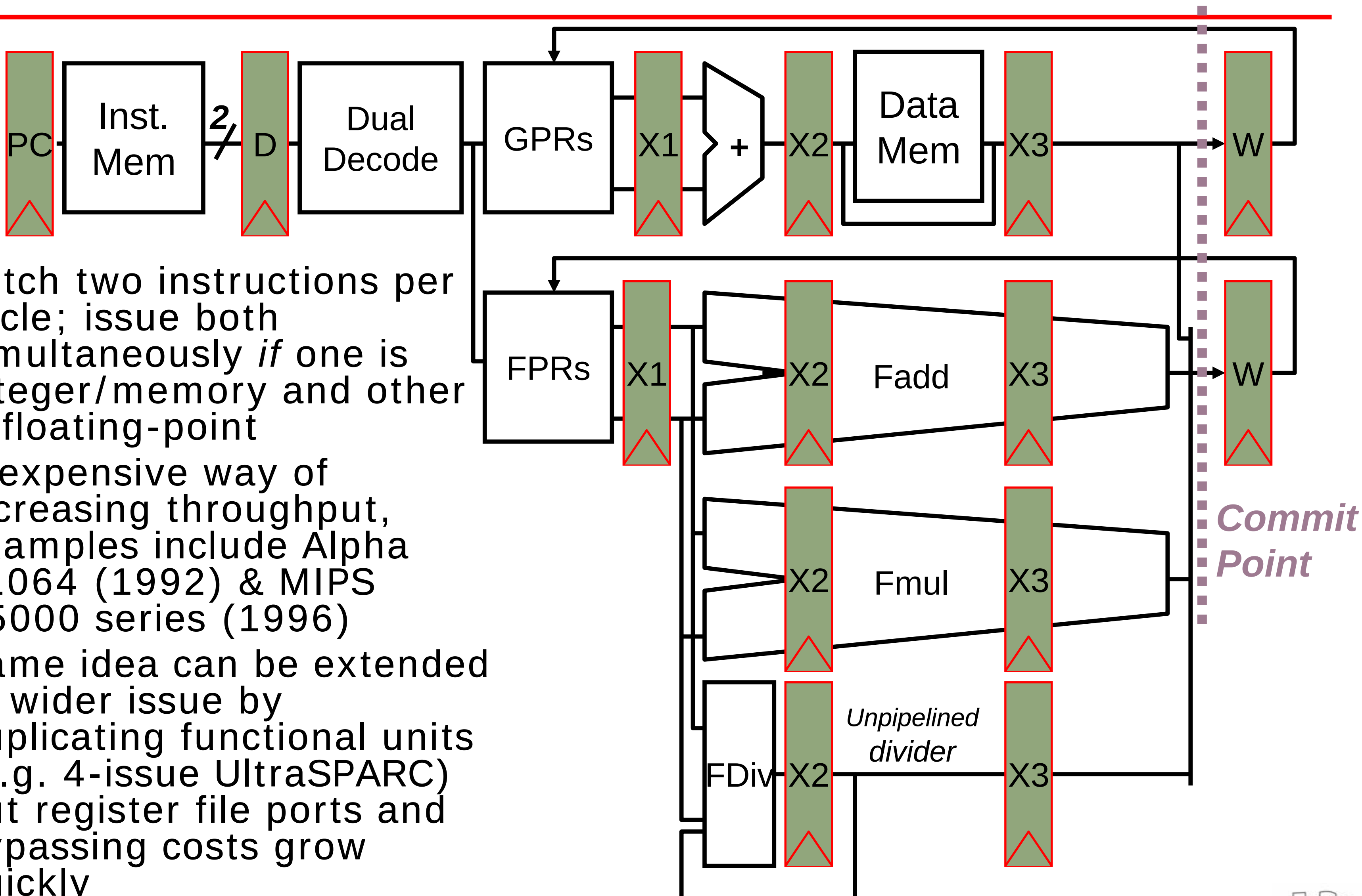
Complex In-Order Pipeline



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

- Stall pipeline on long latency operations, e.g., divides, cache misses
- Exceptions handled in program order at commit point

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 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) but register file ports and bypassing costs grow quickly

Dependence Analysis

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

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

Register vs. Memory Data 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	$M[(r_1) + disp1] \leftarrow (r_2)$
load	$r_3 \leftarrow M[(r_4) + disp2]$

Does $(r_1 + disp1) = (r_4 + disp2)$ *?*

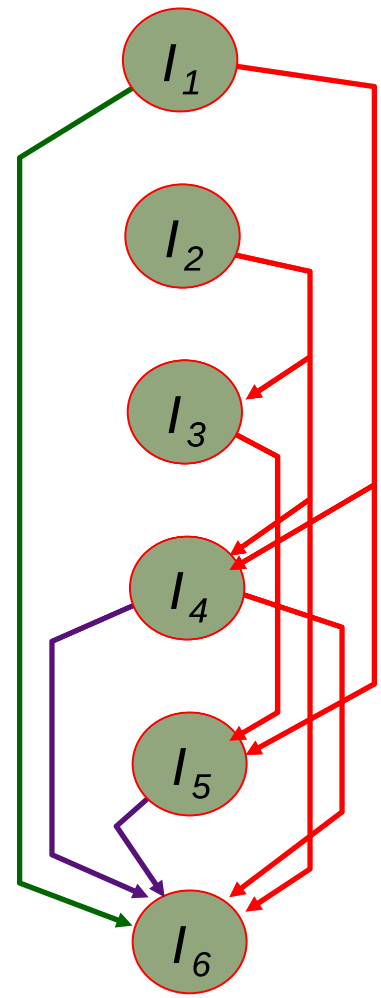
Data Hazards: An Example

I_1	DIVD	f6,	f6,	f4
I_2	LD	f2,	45(r3)	
I_3	MULTD	f0,	f2,	f4
I_4	DIVD	f8,	f6,	f2
I_5	SUBD	f10,	f0,	f6
I_6	ADDD	f6,	f8,	f2

RAW Hazards
WAR Hazards
WAW Hazards

Instruction Scheduling

I_1	DIVD	f6,	f6,	f4
I_2	LD	f2,	45(r3)	
I_3	MULTD	f0,	f2,	f4
I_4	DIVD	f8,	f6,	f2
I_5	SUBD	f10,	f0,	f6
I_6	ADDD	f6,	f8,	f2



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	DIVD	f6,	f6,	f4	*4*
I_2	LD	f2,	45(r3)		*1*
I_3	MULTD	f0,	f2,	f4	*3*
I_4	DIVD	f8,	f6,	f2	*4*
I_5	SUBD	f10,	f0,	f6	*1*
I_6	ADDD	f6,	f8,	f2	*1*

in-order comp	1	2			<u>1</u>	<u>2</u>	3	4		<u>3</u>	5	<u>4</u>	6	<u>5</u>	<u>6</u>
out-of-order comp	1	2	<u>2</u>	3	<u>1</u>	4	<u>3</u>	5	<u>5</u>	<u>4</u>	6	<u>6</u>			

Five-minute break to stretch your legs

Scoreboard:
A Hardware Data Structure to Detect Hazards Dynamically

CDC 6600 *Seymour Cray, 1963*

Image removed due to copyright restrictions.

Image removed due to copyright restrictions.

- 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 (no microcoding)
- 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, novel freon-based technology for cooling
- Fastest machine in world for 5 years (until 7600)
  - over 100 sold ($7-10M each)

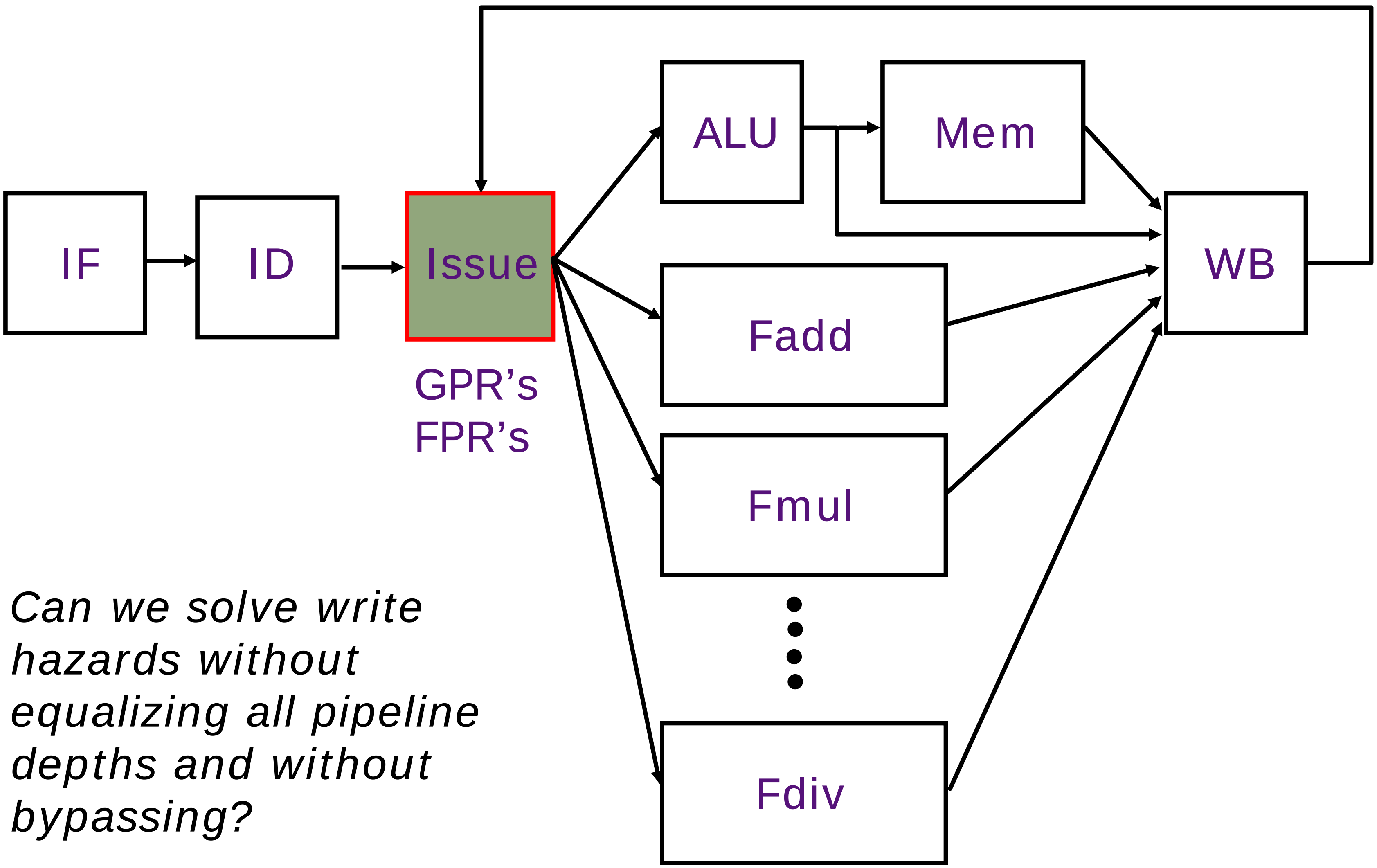
IBM Memo on CDC6600

Thomas Watson Jr., IBM CEO, August 1963:

> *"Last week, Control Data ... announced the 6600 system. I understand that in the laboratory developing the system there are only 34 people including the janitor. Of these, 14 are engineers and 4 are programmers... Contrasting this modest effort with our vast development activities, I fail to understand why we have lost our industry leadership position by letting someone else offer the world's most powerful computer."*

To which Cray replied: *"It seems like Mr. Watson has answered his own question."*

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? $\Rightarrow$ 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?	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 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 to true by the Issue stage and set to false 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 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?	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	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	DIVD	f6,	f6,	f4
I_2	LD	f2,	45(r3)	
I_3	MULTD	f0,	f2,	f4
I_4	DIVD	f8,	f6,	f2
I_5	SUBD	f10,	f0,	f6
I_6	ADDD	f6,	f8,	f2

Thank you !