ECE 252 / CPS 220 Advanced Computer Architecture I

Lecture 8 Instruction-Level Parallelism – Part 1

Benjamin Lee
Electrical and Computer Engineering
Duke University

www.duke.edu/~bcl15
www.duke.edu/~bcl15/class/class_ece252fall11.html

ECE252 Administrivia

29 September – Homework #2 Due

- Use blackboard forum for questions
- Attend office hours with questions
- Email for separate meetings

4 October – Class Discussion

Roughly one reading per class. Do not wait until the day before!

1. Srinivasan et al. "Optimizing pipelines for power and performance"
2. Mahlke et al. "A comparison of full and partial predicated execution support for ILP processors"
3. Palacharla et al. "Complexity-effective superscalar processors"
4. Yeh et al. "Two-level adaptive training branch prediction"

Complex Pipelining

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

MIPS Floating Point

- Interaction between floating-point (FP), integer datapath defined by ISA
- Architect separate register files for floating point (FPR) and integer (GPR)
- Define separate load/store instructions for FPR, GPR
- Define move instructions between register files
- Define FP branches in terms of FP-specific condition codes

Floating-Point Unit (FPU)

FPU requires much more hardware than integer unit

Single-cycle FPU a bad idea

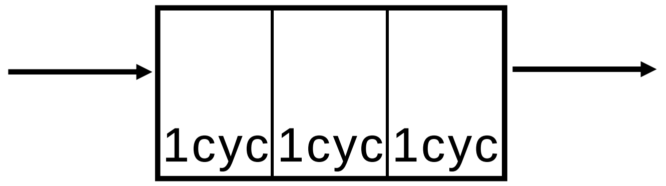
- Why?
- It is common to have several, different types of FPUs (Fadd, Fmul, etc.)
- FPU may be pipelined, partially pipelined, or not pipelined

Floating-point Register File (FPR)

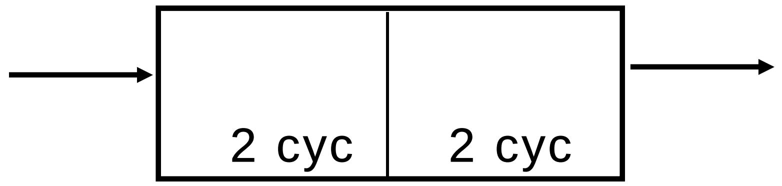
- To operate several FPUs concurrently, FPR requires several read/write ports

Pipelining FPUs

fully pipelined



partially pipelined



Functional units have internal pipeline registers

- Inputs to a functional unit (e.g., register file) can change during a long latency operation
- Operands are latched when an instruction enters the functional unit

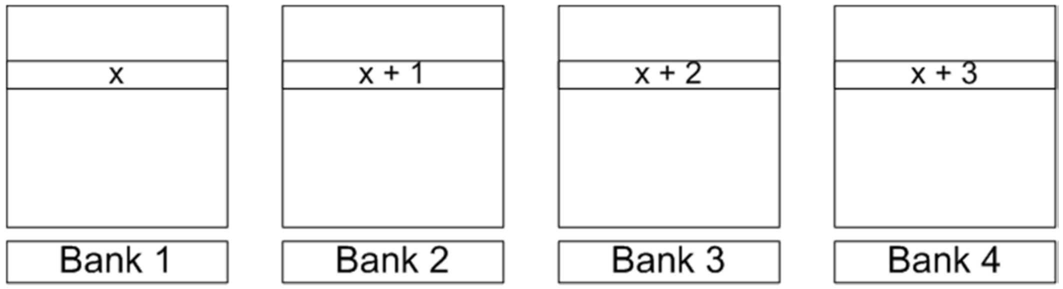
Realistic Memory Systems

Latency of main memory access usually greater than one cycle and often unpredictable

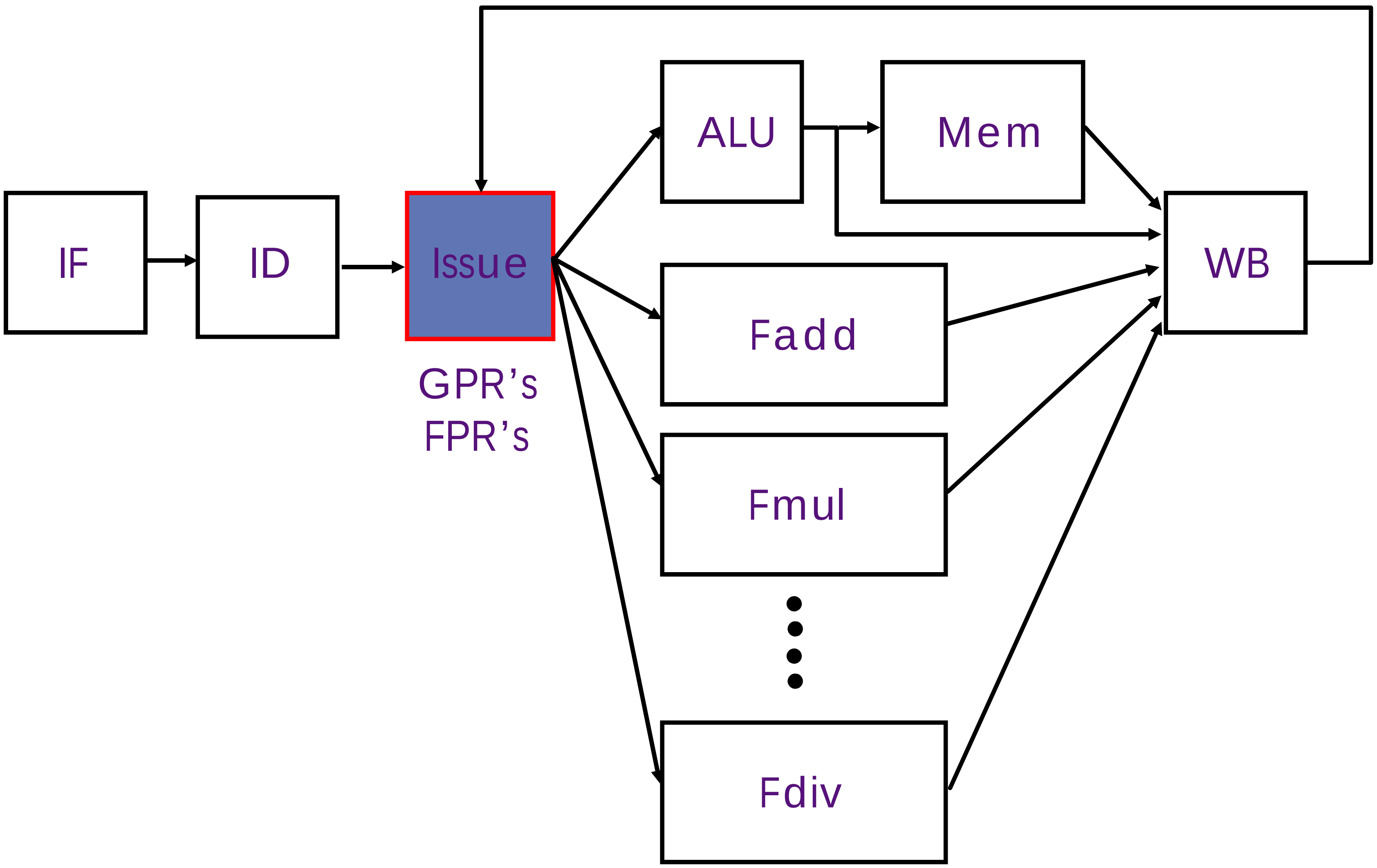
- Solving this problem is a central issue in computer architecture

Improving memory performance

- Separate instruction and data memory ports, no self-modifying code
- Caches -- size L1 cache for single-cycle access
- Caches -- L1 miss stalls pipeline
- Memory – interleaving memory allows multiple simultaneous access
- Memory – bank conflicts stall the pipeline



Multiple Functional Units



Complex Pipeline Control

Implications of multi-cycle instructions

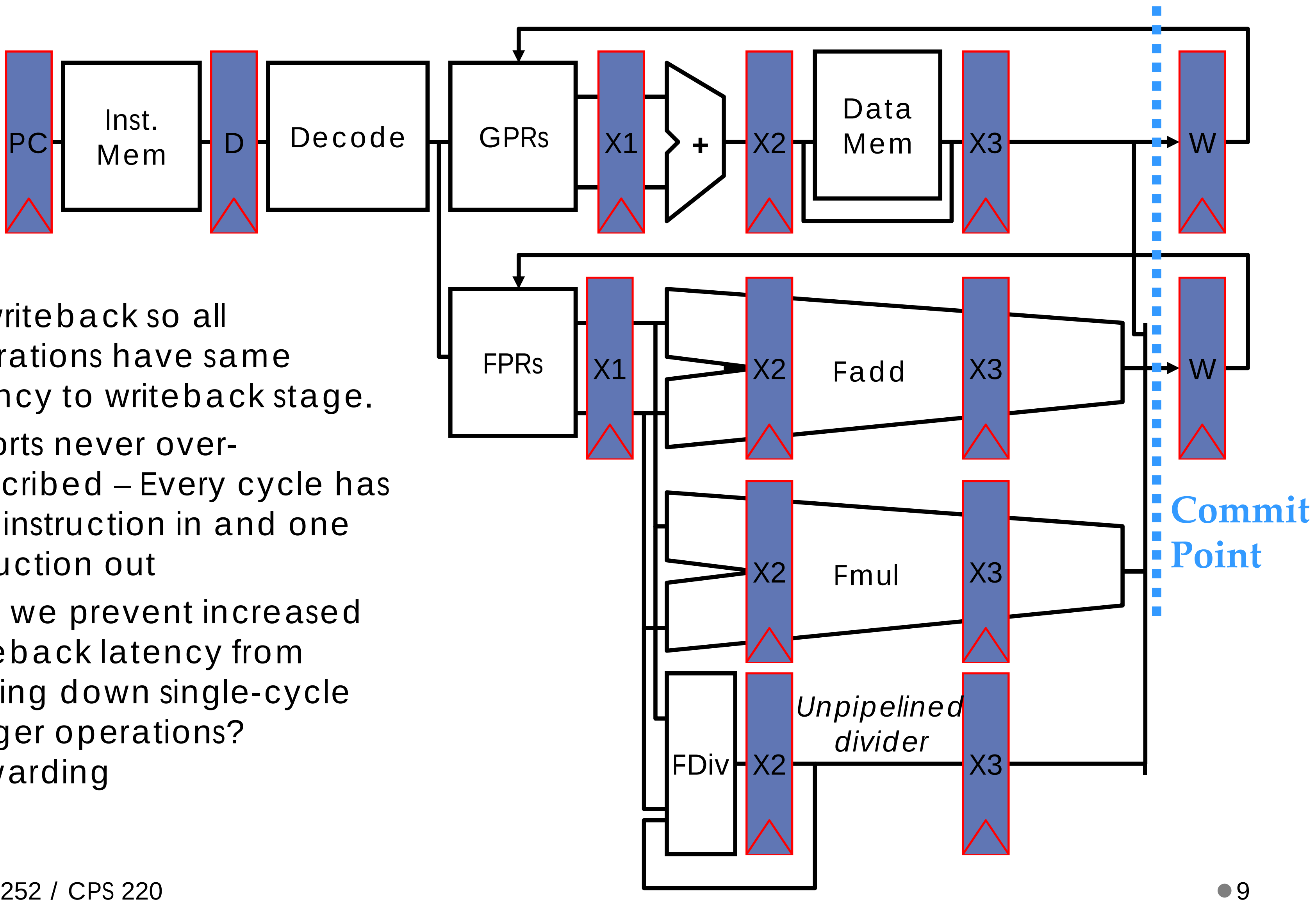
- FPU or memory unit requires more than one cycle
- Structural conflict in execution stage, if FPU or memory unit is not pipelined

Different functional unit latencies

- Structural conflict in writeback stage due to different latencies
- Out-of-order write conflicts due to variable latencies

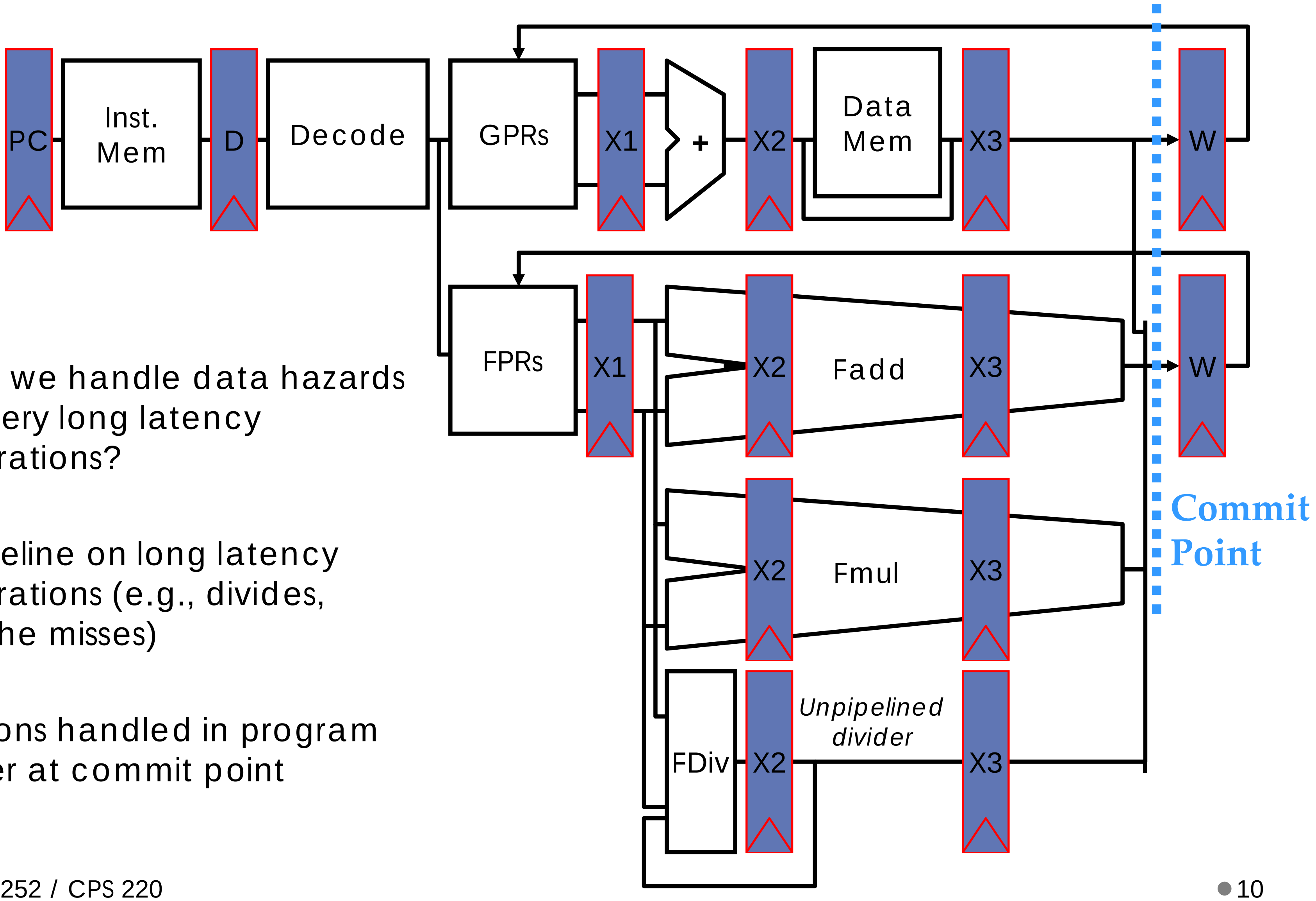
How to handle exceptions?

Complex In-Order Pipeline



- Delay writeback so all operations have same latency to writeback stage.
- Write ports never over-subscribed – Every cycle has one instruction in and one instruction out
- How do we prevent increased writeback latency from slowing down single-cycle integer operations? Forwarding

Complex In-Order Pipeline

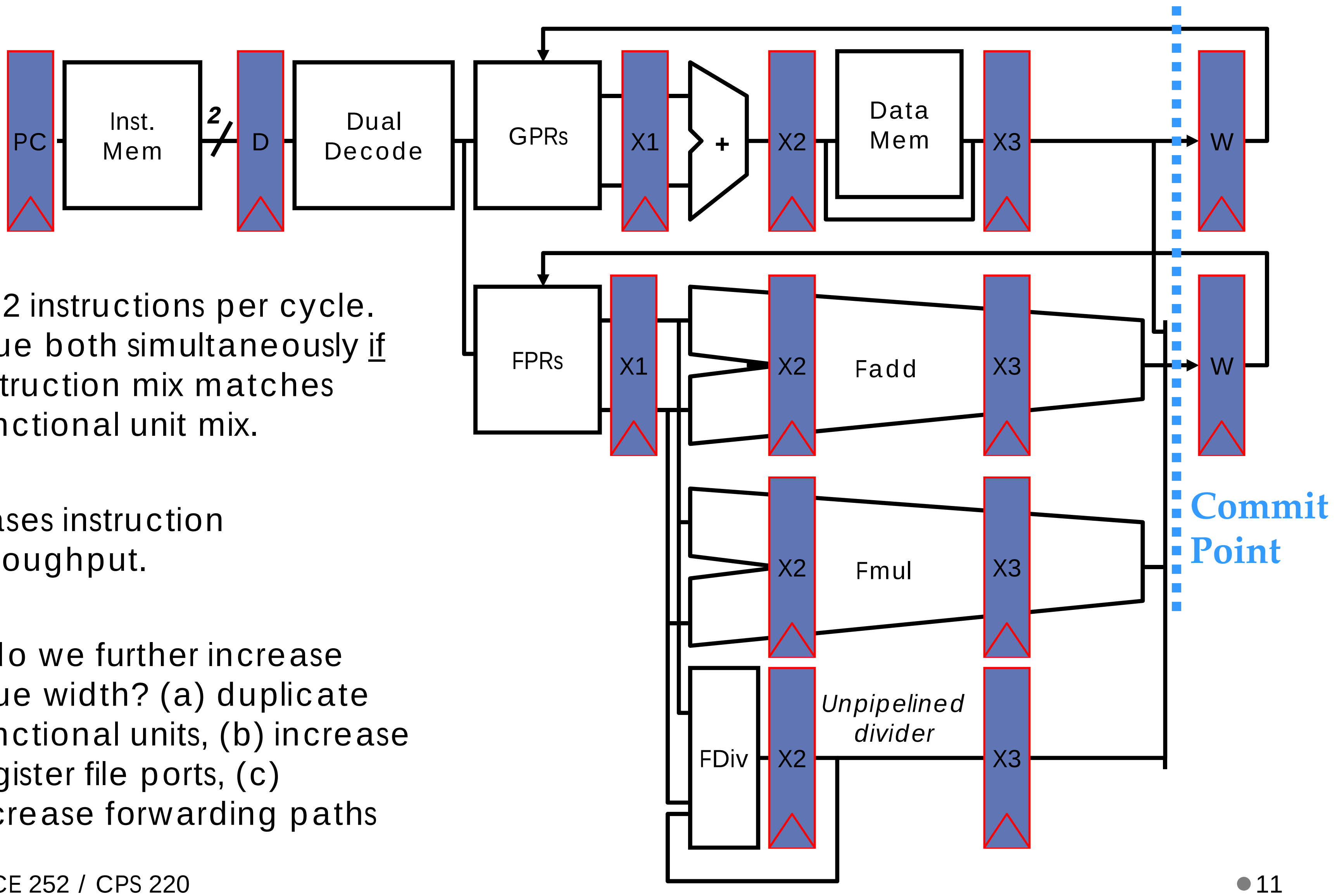


How do we handle data hazards for very long latency operations?

Stall pipeline on long latency operations (e.g., divides, cache misses)

Exceptions handled in program order at commit point

Superscalar In-Order Pipeline



Fetch 2 instructions per cycle.
Issue both simultaneously if instruction mix matches functional unit mix.

Increases instruction throughput.

How do we further increase issue width? (a) duplicate functional units, (b) increase register file ports, (c) increase forwarding paths

Dependence Analysis

Consider executing a sequence of instructions of the form: Rk ← (Ri) op (Rj)

Data Dependence

R3 ← (R1) op (R4) # RAW hazard (R3)

R5 ← (R3) op (R4)

Anti-dependence

R3 ← (R1) op (R2) # WAR hazard (R1)

R1 ← (R4) op (R5)

Output-dependence

R3 ← (R1) op (R2) # WAW hazard (R3)

R3 ← (R6) op (R7)

Detecting Data Hazards

Range and Domain of Instruction (j)

R(j) = registers (or other storage) modified by instruction j

D(j) = registers (or other storage) read by instruction j

Suppose instruction k follows instruction j in program order. Executing instruction k before the effect of instruction j has occurred can cause…

RAW hazard if	$R(j) \cap D(k) \neq \varnothing$	# j modifies a register read by k
WAR hazard if	$D(j) \cap R(k) \neq \varnothing$	# j reads a register modified by k
WAW hazard if	$R(j) \cap R(k) \neq \varnothing$	# j, k modify the same register

Registers vs Memory Dependence

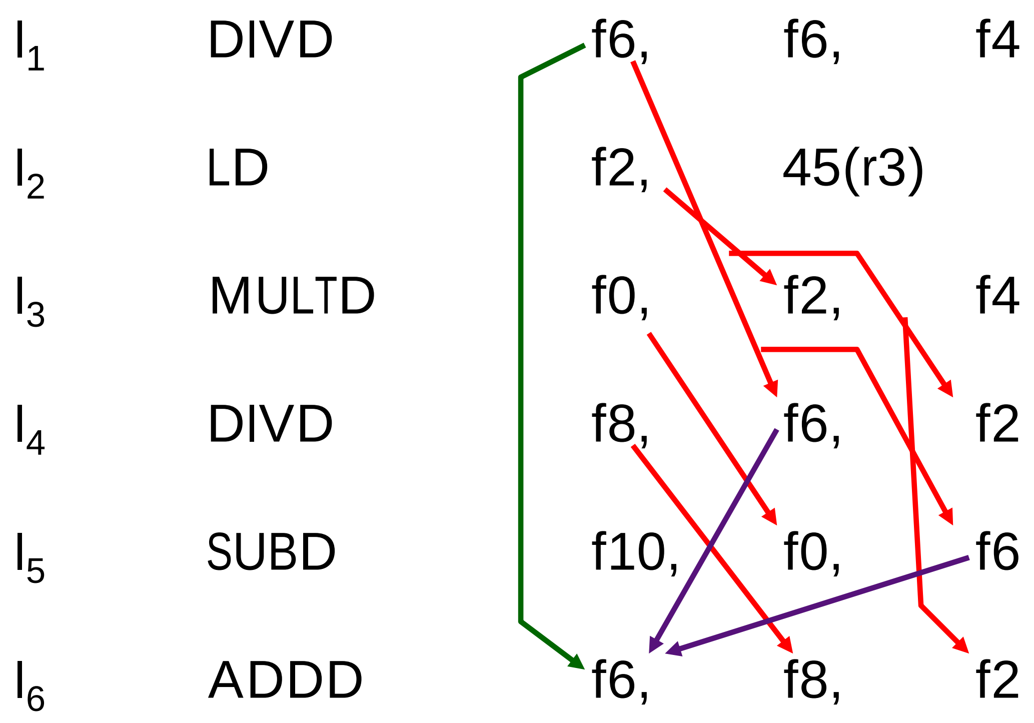
Data hazards due to register operands can be determined at decode stage

Data hazards due to memory operands can be determined <u>only after computing effective address</u> in execute stage

store	M[R1 + disp1] ← R2
load	R3 ← M[R4 + disp2]

(R1 + disp1) == (R4 + disp2)?

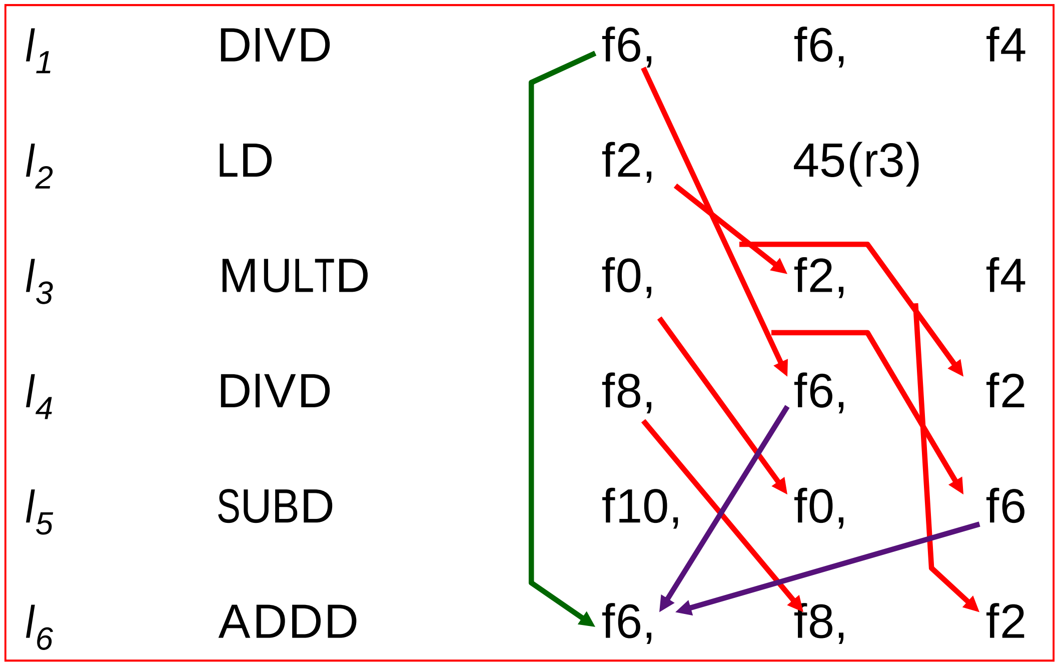
Data Hazards Example



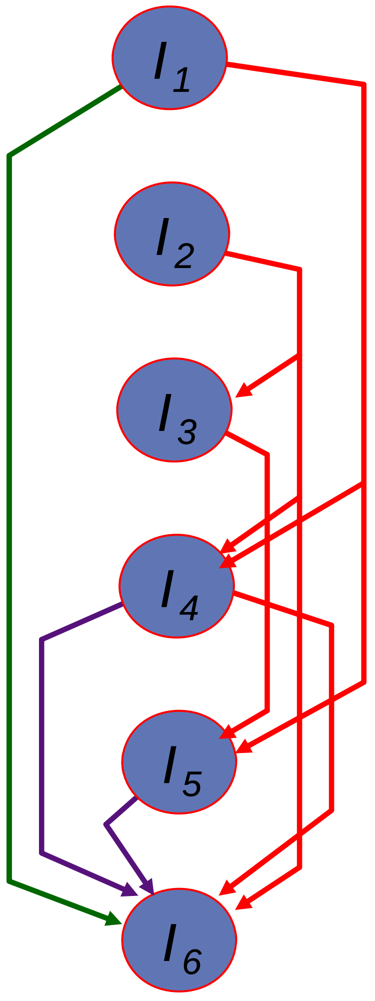
I_1	DIVD	f6,	f6,	f4
I_2	LD	f2,	45(r3)	
I_3	MULTD	f0,	f2,	f4
I_4	DIVD	f8,	f6,	f2
I_5	SUBD	f10,	f0,	f6
I_6	ADDD	f6,	f8,	f2

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

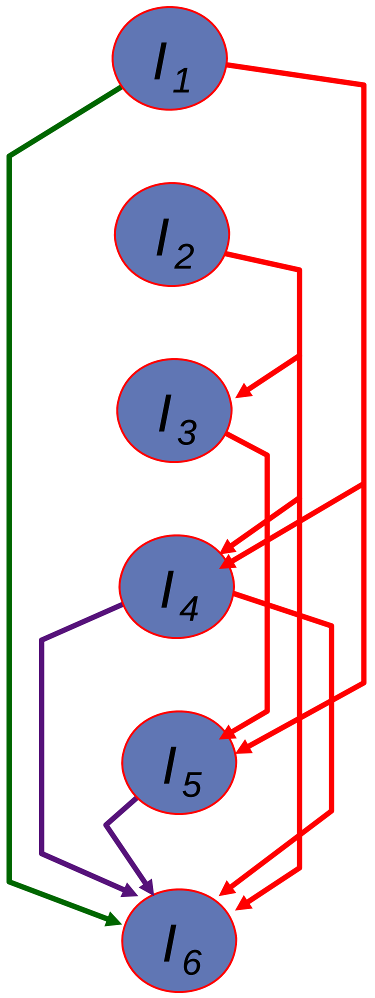


Valid Instruction Orderings

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

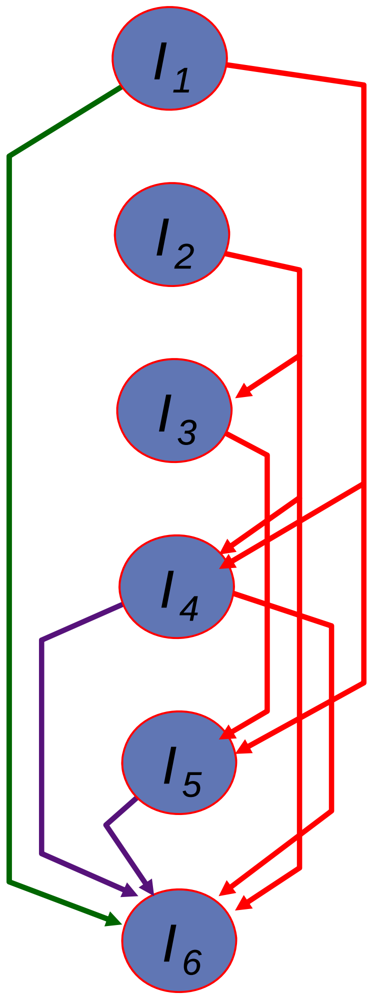
					Latency
I_1	DIVD	f6,	f6,	f4	4
I_2	LD	f2,	45(r3)		1
I_3	MULTD	f0,	f2,	f4	3
I_4	DIVD	f8,	f6,	f2	4
I_5	SUBD	f10,	f0,	f6	1
I_6	ADDD	f6,	f8,	f2	1



Let k indicate when instruction k is issued.
Let <u>k</u> denote when instruction k is completed.

Out-of-Order Completion

					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 $\underline{1}$ $\underline{2}$ 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 $\underline{6}$

Scoreboard

Up until now, we assumed user or compiler <u>statically</u> examines instructions, detecting hazards and scheduling instructions

Scoreboard is a hardware data structure to <u>dynamically</u> detect hazards

Cray CDC6600

<u>Seymour Cray, 1963</u>

- 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
- <u>Dynamic instruction scheduling with scoreboard</u>

- Ten peripheral processors for I/O

-More than 400K transistors, 750 sq-ft, 5 tons, 150kW with novel Freon-based cooling

- Very fast clock, 10MHz (FP add in 4 clocks)
- Fastest machine in world for 5 years
- Over 100 sold ($7-10M each)

IBM Memo on CDC6600

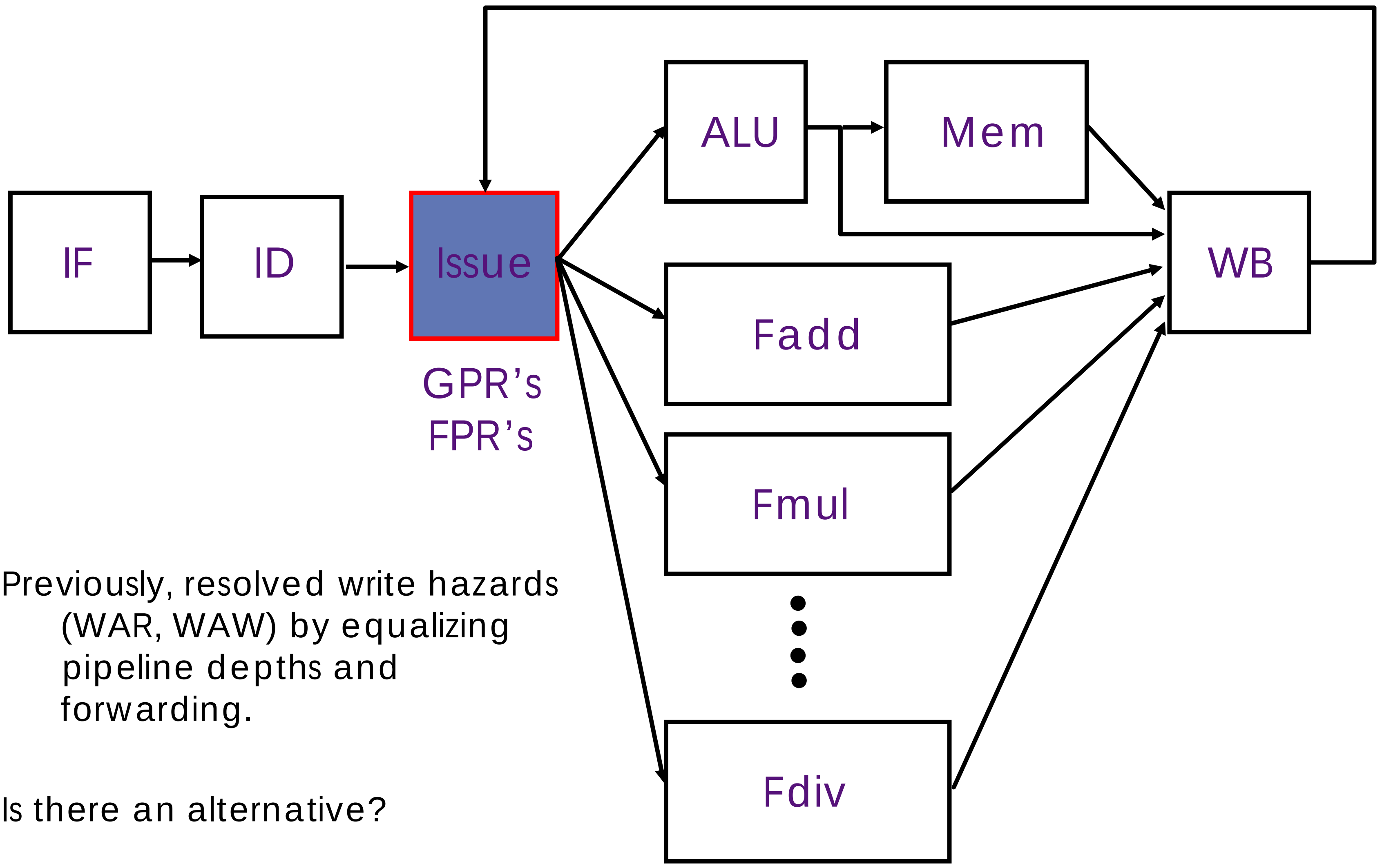
Thomas Watson Jr., IBM CEO, August 1963

“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 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.”

Multiple Functional Units



Previously, resolved write hazards (WAR, WAW) by equalizing pipeline depths and forwarding.

Is there an alternative?

Conditions for Instruction Issue

When is it safe to issue an instruction?

- Suppose a data structure tracks all instructions in all functional units

Before issuing instruction, issue logic must check:

- Is the required functional unit available? Check for structural hazard.
- Is the input data available? Check for RAW hazard.
- Is it safe to write the destination? Check for WAR, WAW hazard
- Is there a structural hazard at the write back stage?

Issue Logic and Data Structure

In issue stage, instruction j consults the table

- Functional unit available?	Check the busy column
- RAW?	Search the dest column for j's sources
- WAR?	Search the source column for j's destination
- WAW?	Search the dest column for j's destination

Add entry if no hazard detected, instruction issues

Remove entry when instruction writes back

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

Simplifying the Data Structure

Assume instructions issue in-order

Assume issue logic does not dispatch instruction if it detects RAW hazard or busy functional unit

Assume functional unit latches operands when the instruction is issued

Simplifying the Data Structure

Can the dispatched instruction cause WAR hazard?

- No, because operands are read at issue and instructions issue in-order

No WAR Hazards

- No need to track source-1 and source-2

Can the dispatched instruction cause WAW hazard?

- Yes, because instructions may complete out-of-order

Do not issue instruction in case of WAW hazard

- In scoreboard, a register name occurs at most once in ‘dest’ column

Scoreboard

Busy[FU#]: a bit-vector to indicate functional unit availability (FU = Int, Add, Mutl, Div)

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

- Bits are set to true by issue logic
- Bits are set to false by writeback stage
- Each functional unit's pipeline registers must carry 'dest' field and a flag to indicate if it's valid: "the (we, ws) pair"

Issue logic checks instruction (opcode, dest, src1, src2) against scoreboard (busy, wp) to dispatch

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

	Busy-Functional Units Status											Writes Pending (WP)	
	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	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

<ins>Instruction Issue Logic</ins>
FU available? Busy[FU#]
RAW? WP[src1] or WP[src2]
WAR? Cannot arise
WAW? WP[dest]

Scoreboard

Detect hazards dynamically

Issue instructions in-order

Complete instructions out-of-order

Increases instruction-level-parallelism by

- More effectively exploiting multiple functional units
- Reducing the number of pipeline stalls due to hazards

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)
- Alvin Lebeck (Duke)
- David Patterson (UCB)
- Daniel Sorin (Duke)