

Complex Pipelining

MBT

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

Complex Pipeline Control Issues

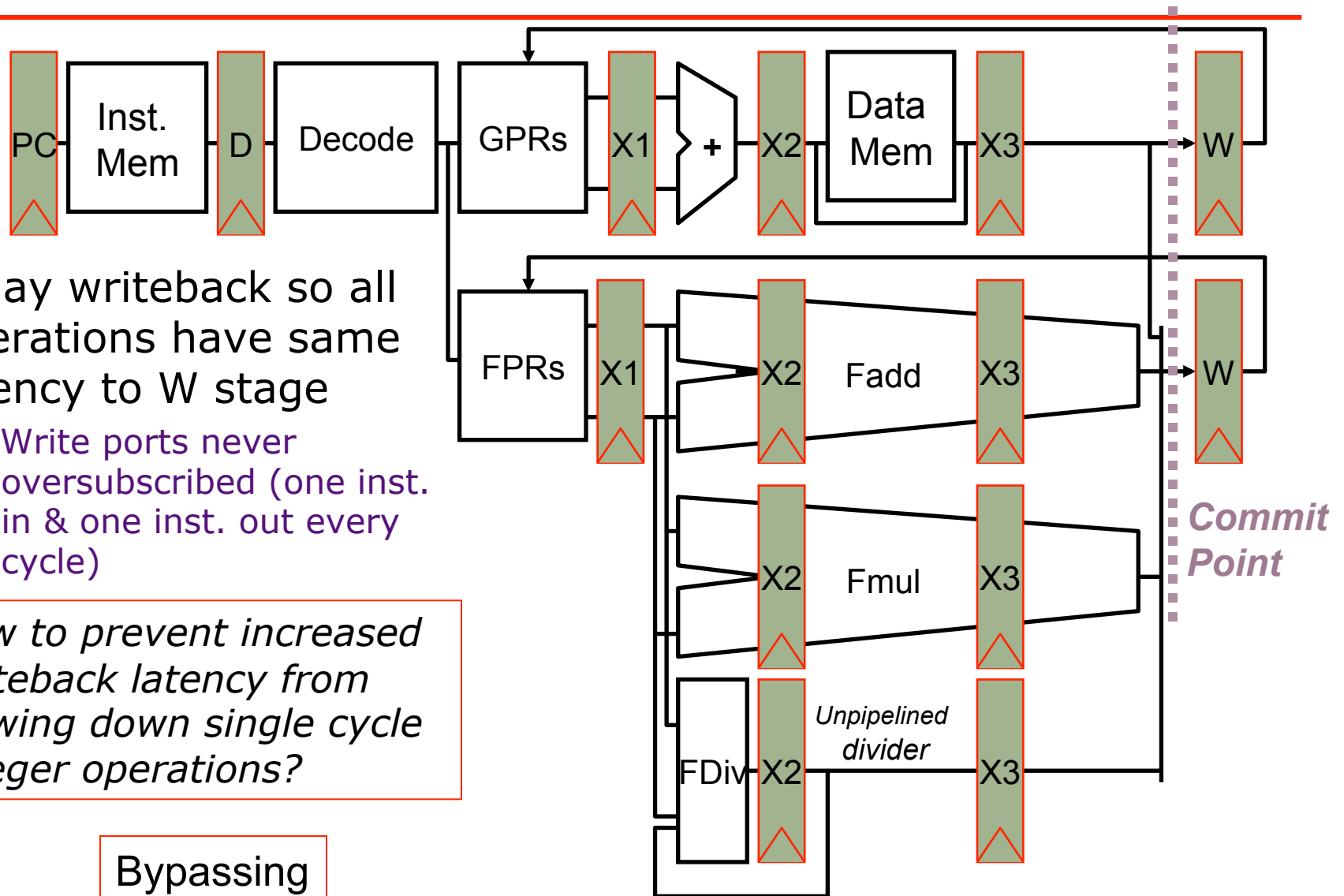
- Structural conflicts at the write-back stage due to variable latencies of different function units
- Structural conflicts at the execution stage if some FPU or memory unit is not pipelined and takes more than one cycle
- Out-of-order write hazards due to variable latencies of different function units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

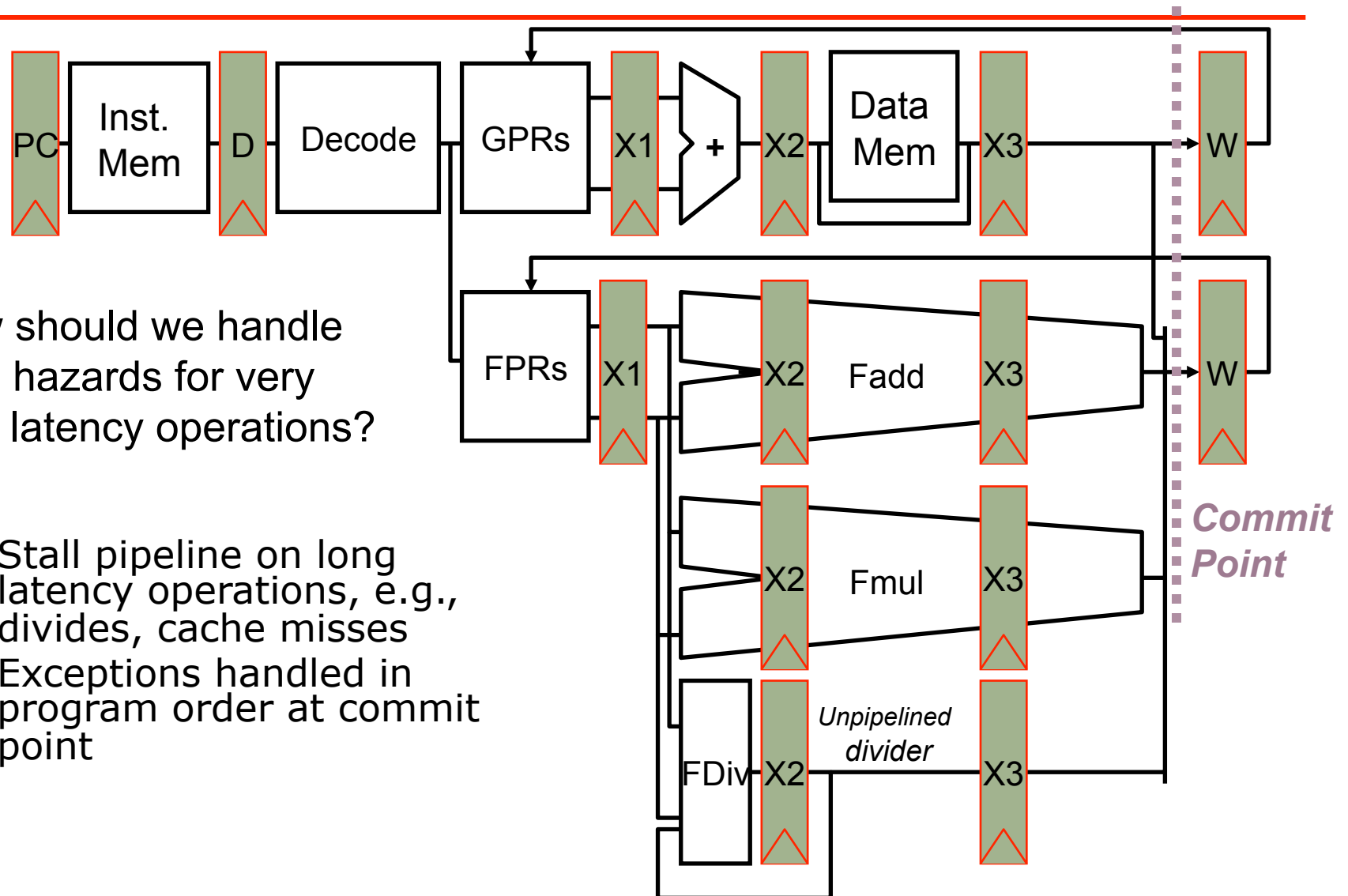
Bypassing



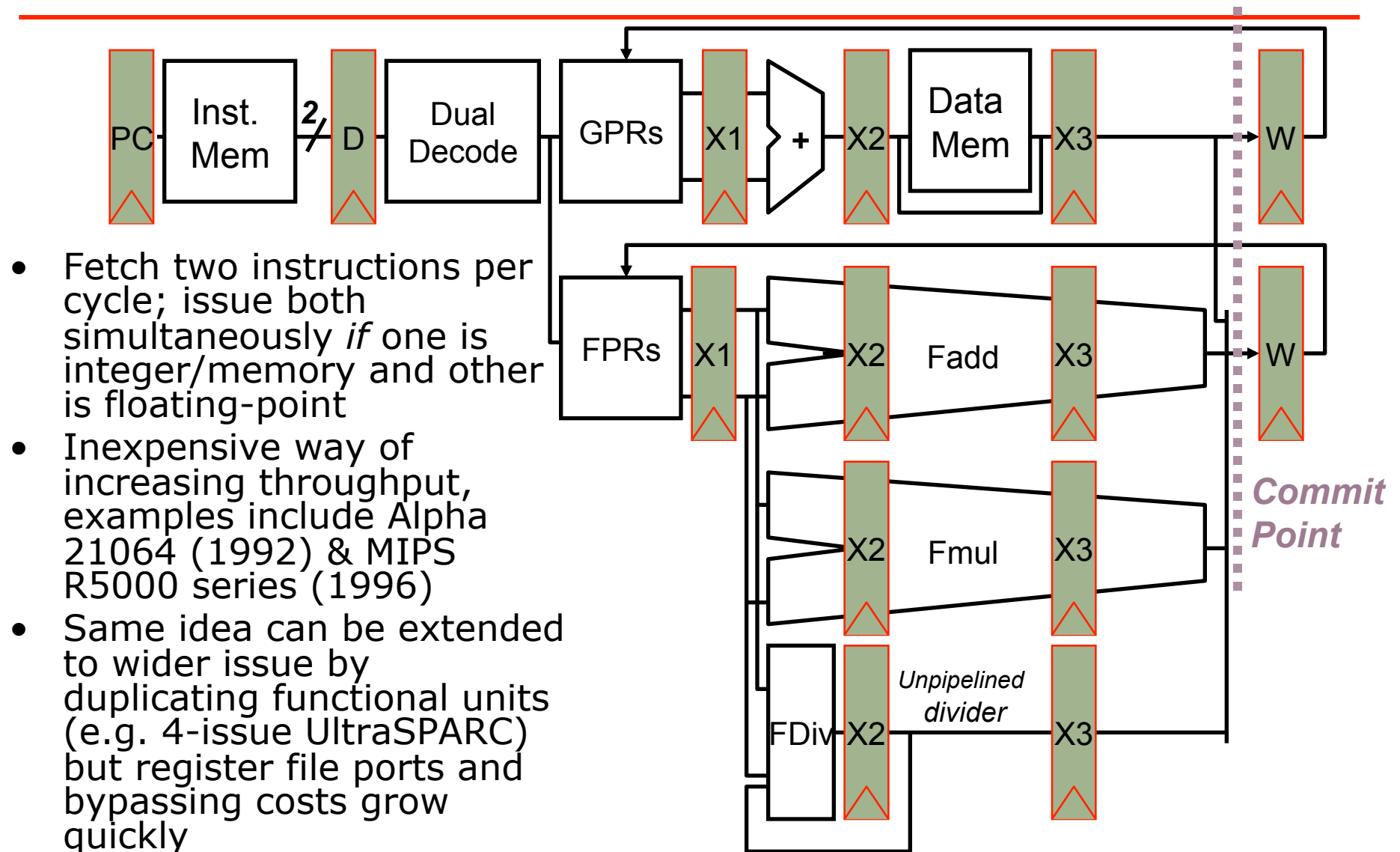
Complex In-Order Pipeline

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



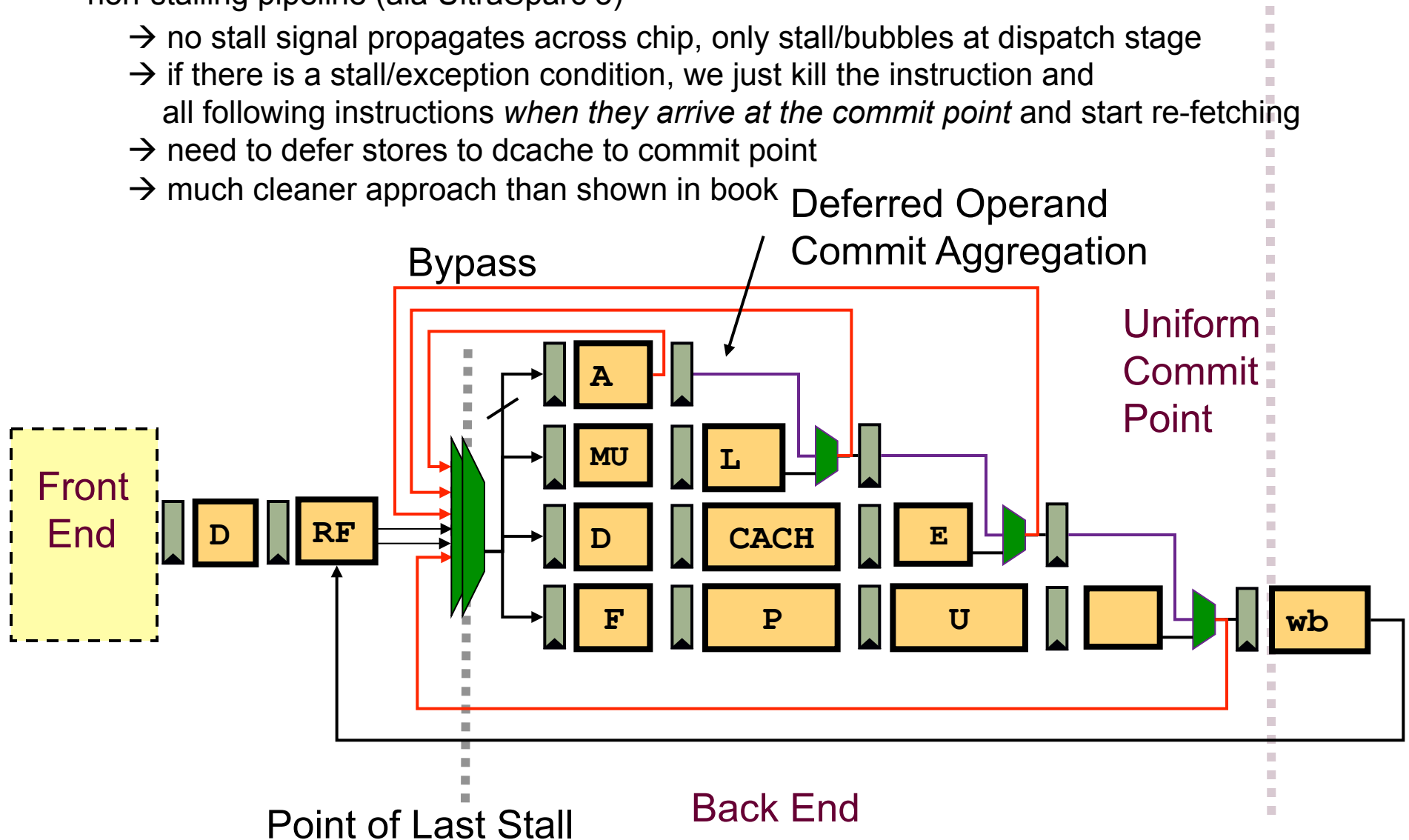
- Fetch two instructions per cycle; issue both simultaneously *if* one is integer/memory and other is floating-point
- Inexpensive way of increasing throughput, examples include Alpha 21064 (1992) & MIPS R5000 series (1996)
- Same idea can be extended to wider issue by duplicating functional units (e.g.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

Modern High-Frequency In-Order Pipeline

- with deferred commit & precise exceptions

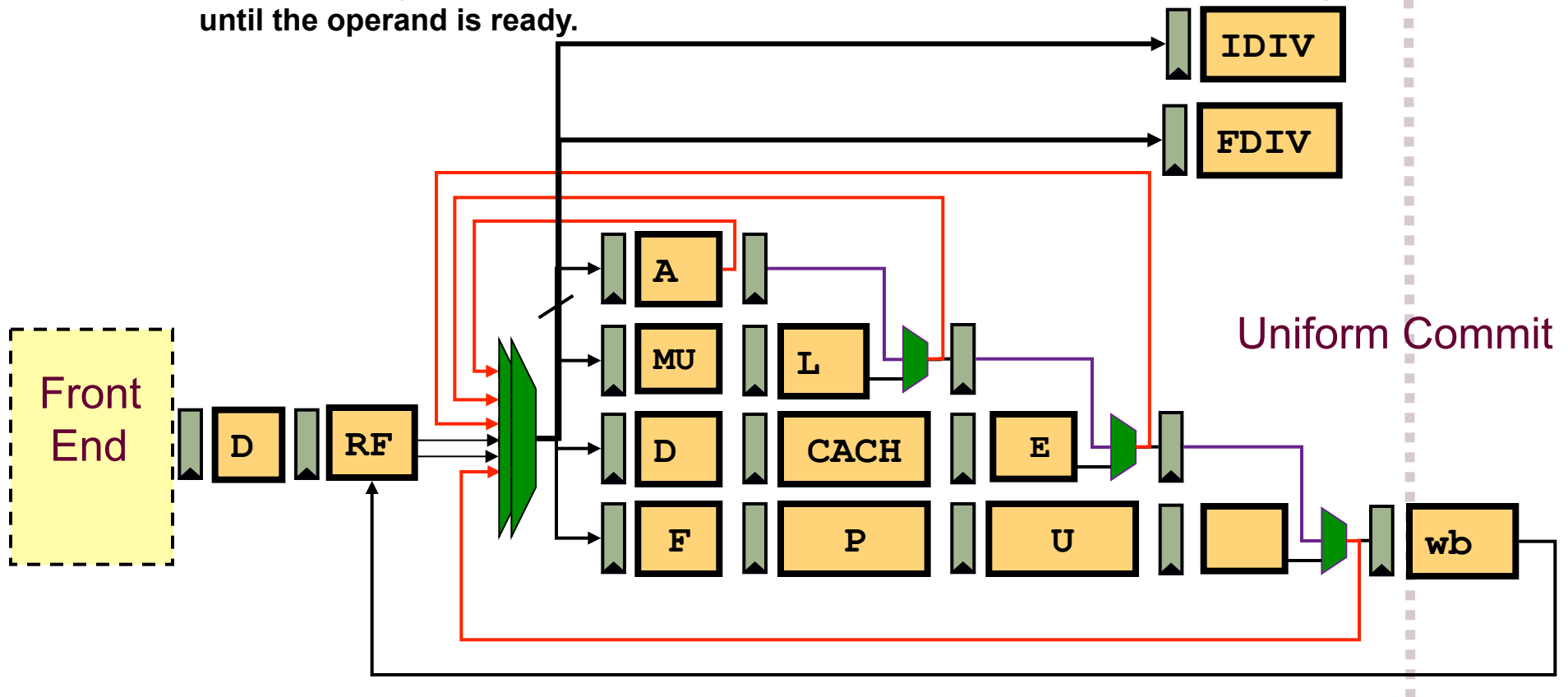
- non-stalling pipeline (ala UltraSparc 3)

- no stall signal propagates across chip, only stall/bubbles at dispatch stage
- if there is a stall/exception condition, we just kill the instruction and all following instructions *when they arrive at the commit point* and start re-fetching
- need to defer stores to dcache to commit point
- much cleaner approach than shown in book



Modern In-Order Pipeline – Long Occupancy Instructions

- today, mostly only matters for divides/sqrt
- for MIPS-style HI/LO registers [which do not cause exceptions (ala MIPS MUL/DIV)]
 - a signal tells the unit to squash its result if it followed killed instructions
 - unit runs in parallel
- for register-to-register operations [ala FDIV]; or non-blocking loads:
 - a) do not issue successive instructions until P cycles before we find out about whether instruction causes an exception, where P is distance to commit → we can often detect exceptions earlier than the operand is available
 - b) Once we are sure they will not cause exceptions, we can perform *out-of-order completion*: issue any instructions that do not read/write to the dest register (prevent RAW / WAW) and defer the write-back of the output operand (steal a cycle from the register file when it is free) until it is ready. If there is an interrupt, the exception handler's read to the register will stall until the operand is ready.



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

$$\begin{array}{l} r_3 \leftarrow (r_1) \text{ op } (r_2) \\ r_5 \leftarrow (r_3) \text{ op } (r_4) \end{array}$$


Read-after-Write
(RAW) hazard

Anti-dependence

$$\begin{array}{l} r_3 \leftarrow (r_1) \text{ op } (r_2) \\ r_1 \leftarrow (r_4) \text{ op } (r_5) \end{array}$$


Write-after-Read
(WAR) hazard

Output-dependence

$$\begin{array}{l} r_3 \leftarrow (r_1) \text{ op } (r_2) \\ r_3 \leftarrow (r_6) \text{ op } (r_7) \end{array}$$


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 $R(i) \cap D(j) \neq \emptyset$

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

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

Register vs. Memory Data Dependence

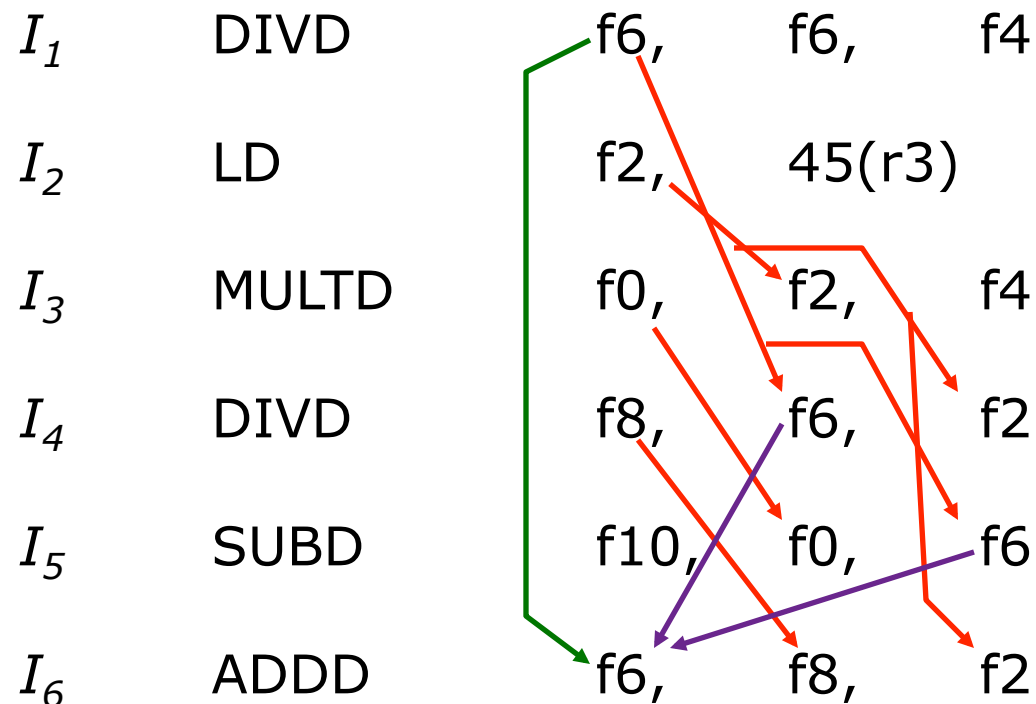
Data hazards due to register operands can be determined at the decode stage *but*

data hazards due to memory operands can be determined only after computing the effective address

<i>store</i>	$M[(r_1) + \text{disp1}] \leftarrow (r_2)$
<i>load</i>	$r_3 \leftarrow M[(r_4) + \text{disp2}]$

Does $(r_1 + \text{disp1}) = (r_4 + \text{disp2})$?

Data Hazards: An Example

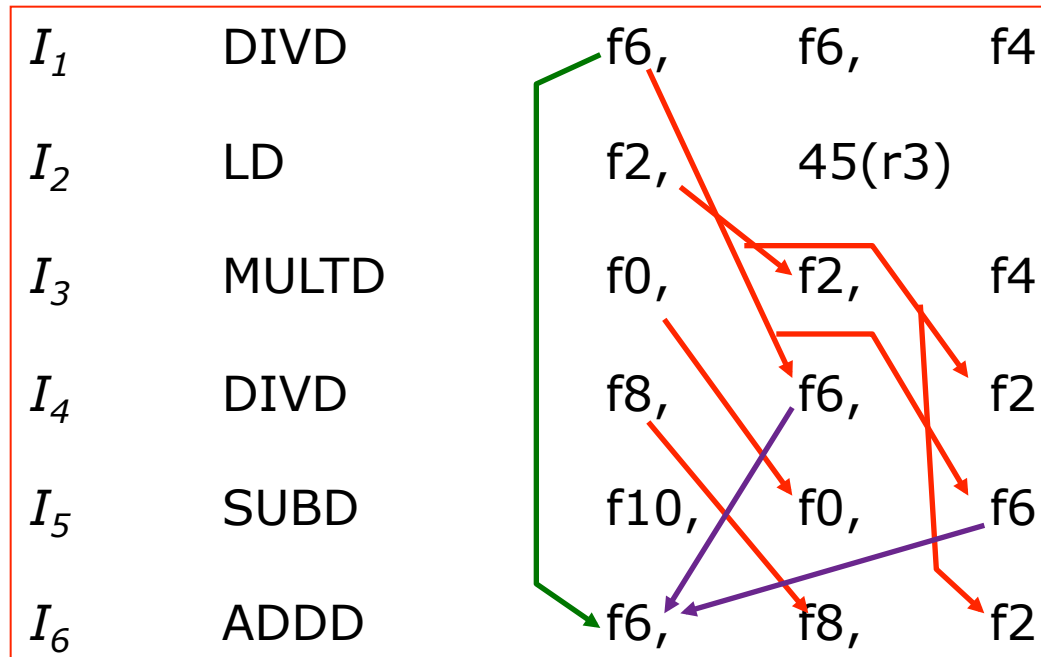


RAW Hazards

WAR Hazards

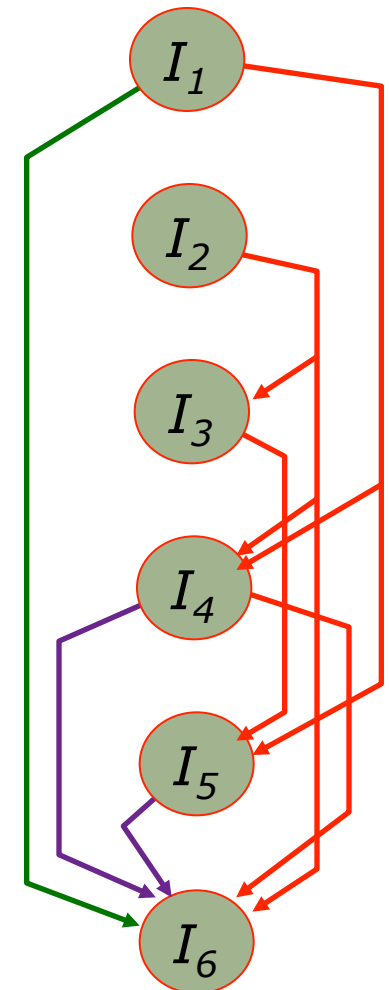
WAW Hazards

Instruction Scheduling



Valid orderings:

<i>in-order</i>	I_1	I_2	I_3	I_4	I_5	I_6
<i>out-of-order</i>	I_2	I_1	I_3	I_4	I_5	I_6
<i>out-of-order</i>	I_1	I_2	I_3	I_5	I_4	I_6



Out-of-order Completion

In-order Issue

					<i>Latency</i>
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

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

Scoreboard:

A Hardware Data Structure to Detect Hazards Dynamically

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 (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)

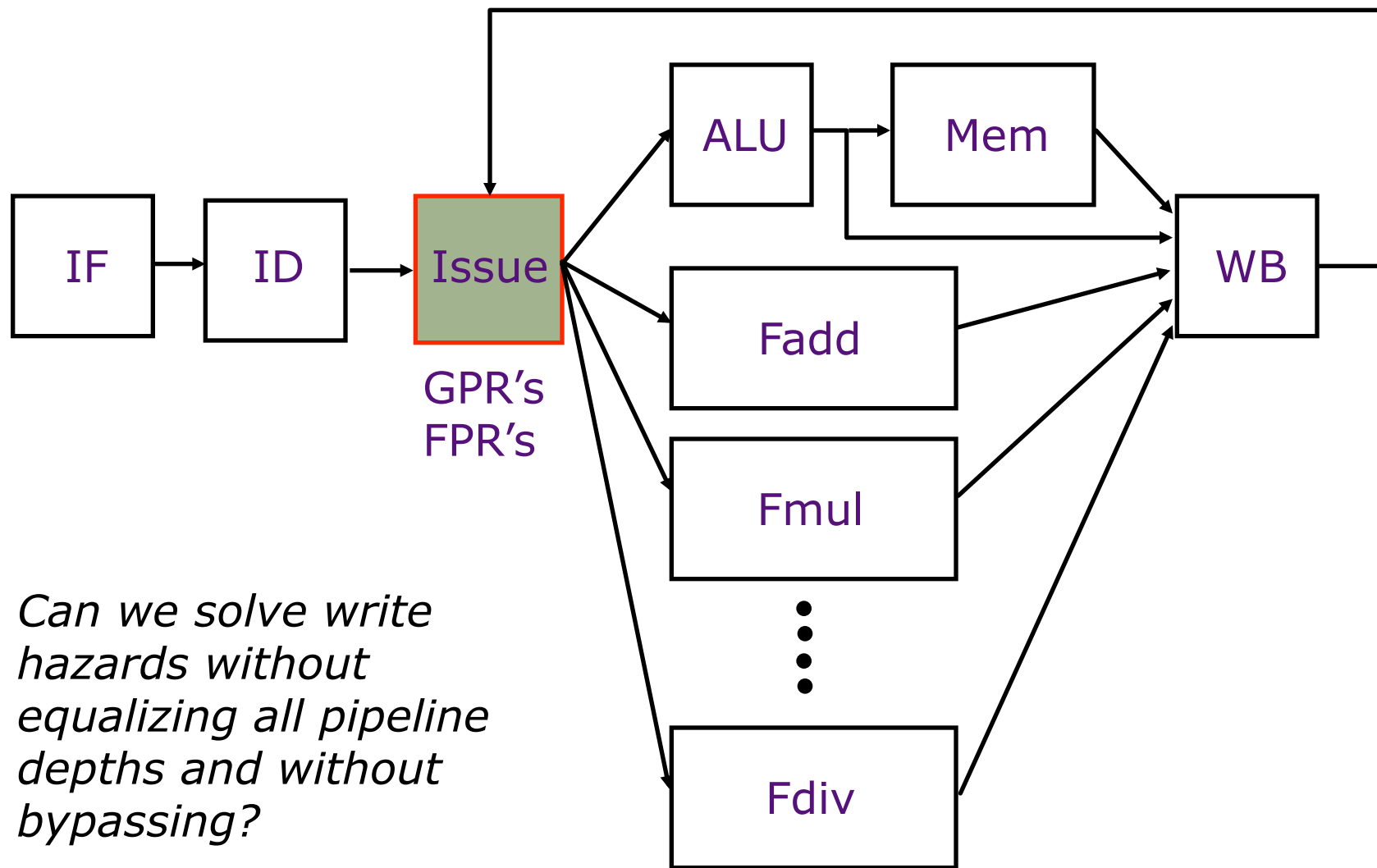
IBM Memo on CDC6600

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?

Non-blocking loads / Long latency instrs

No WAR hazard

⇒ (in-order scoreboard) no need to keep *src1* and *src2*

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

⇒ register names are reserved by uncommitted instrs

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”