CS252 Graduate Computer Architecture
Fall 2015
Lecture 4: Pipelining

Krste Asanovic
`krste@berkeley.edu`
`http://inst.eecs.berkeley.edu/~cs252/fa15`

Last Time in Lecture 3

- Microcoding, an effective technique to manage control unit complexity, invented in era when logic (tubes), main memory (magnetic core), and ROM (diodes) used different technologies
- Difference between ROM and RAM speed motivated additional complex instructions
- Technology advances leading to fast SRAM made technology assumptions invalid
- Complex instructions sets impede parallel and pipelined implementations
- Load/store, register-rich ISAs (pioneered by Cray, popularized by RISC) perform better in new VLSI technology

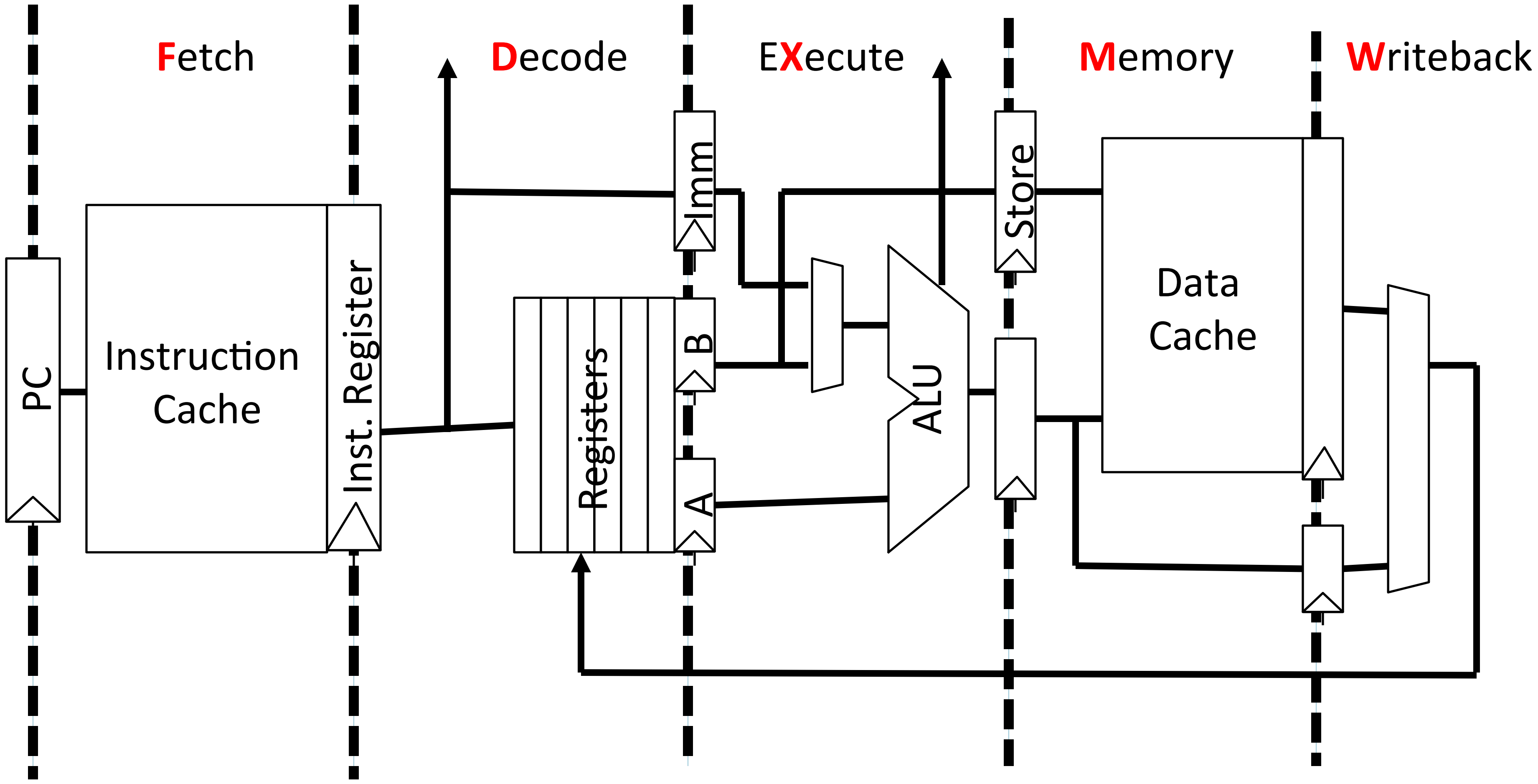
“Iron Law” of Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

- Instructions per program depends on source code, compiler technology, and ISA
- Cycles per instructions (CPI) depends on ISA and μarchitecture
- Time per cycle depends upon the μarchitecture and base technology

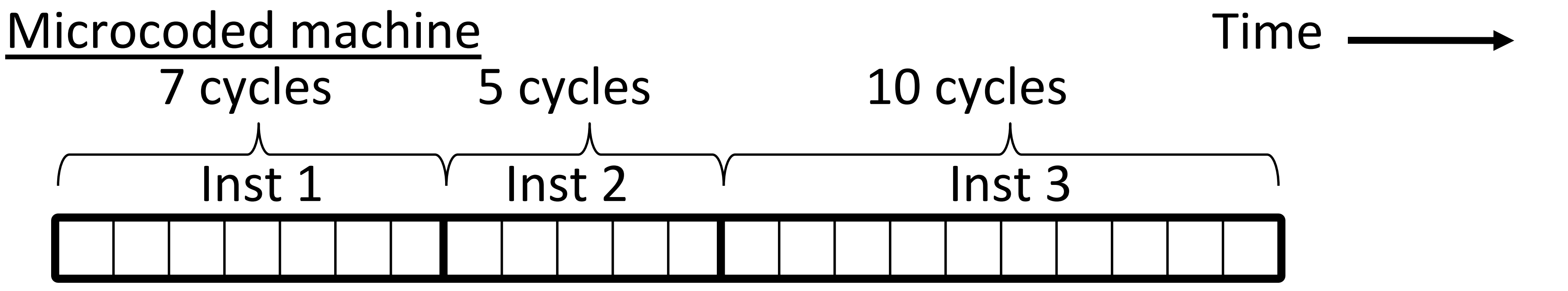
Microarchitecture	CPI	cycle time
Microcoded	>1	short
Single-cycle unpipelined	1	long
Pipelined	1	short

Classic 5-Stage RISC Pipeline

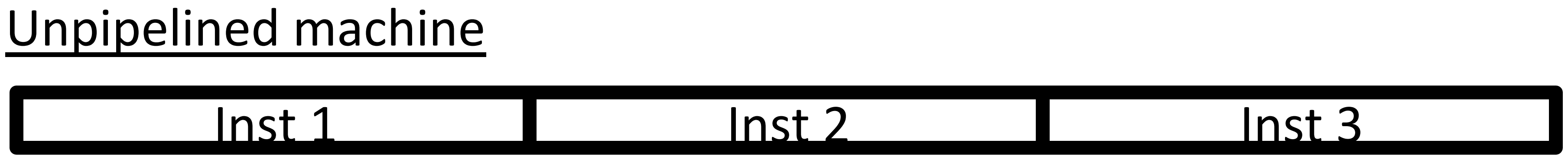


This version designed for regfiles/memories with synchronous reads and writes.

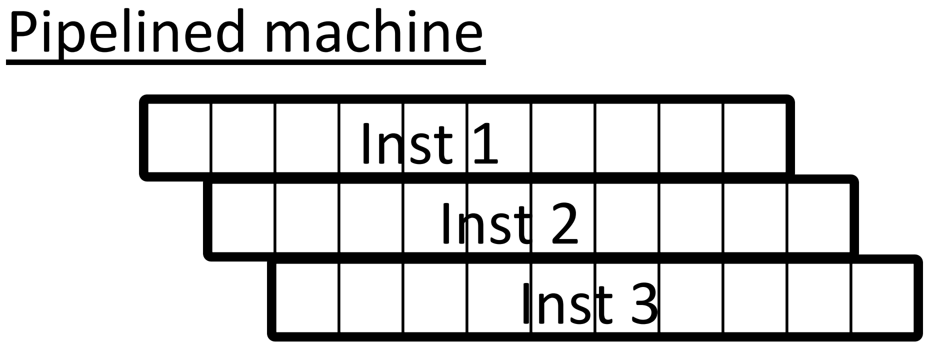
CPI Examples



3 instructions, 22 cycles, CPI=7.33



3 instructions, 3 cycles, CPI=1



3 instructions, 3 cycles, CPI=1

5-stage pipeline CPI≠5!!!

Instructions interact with each other in pipeline

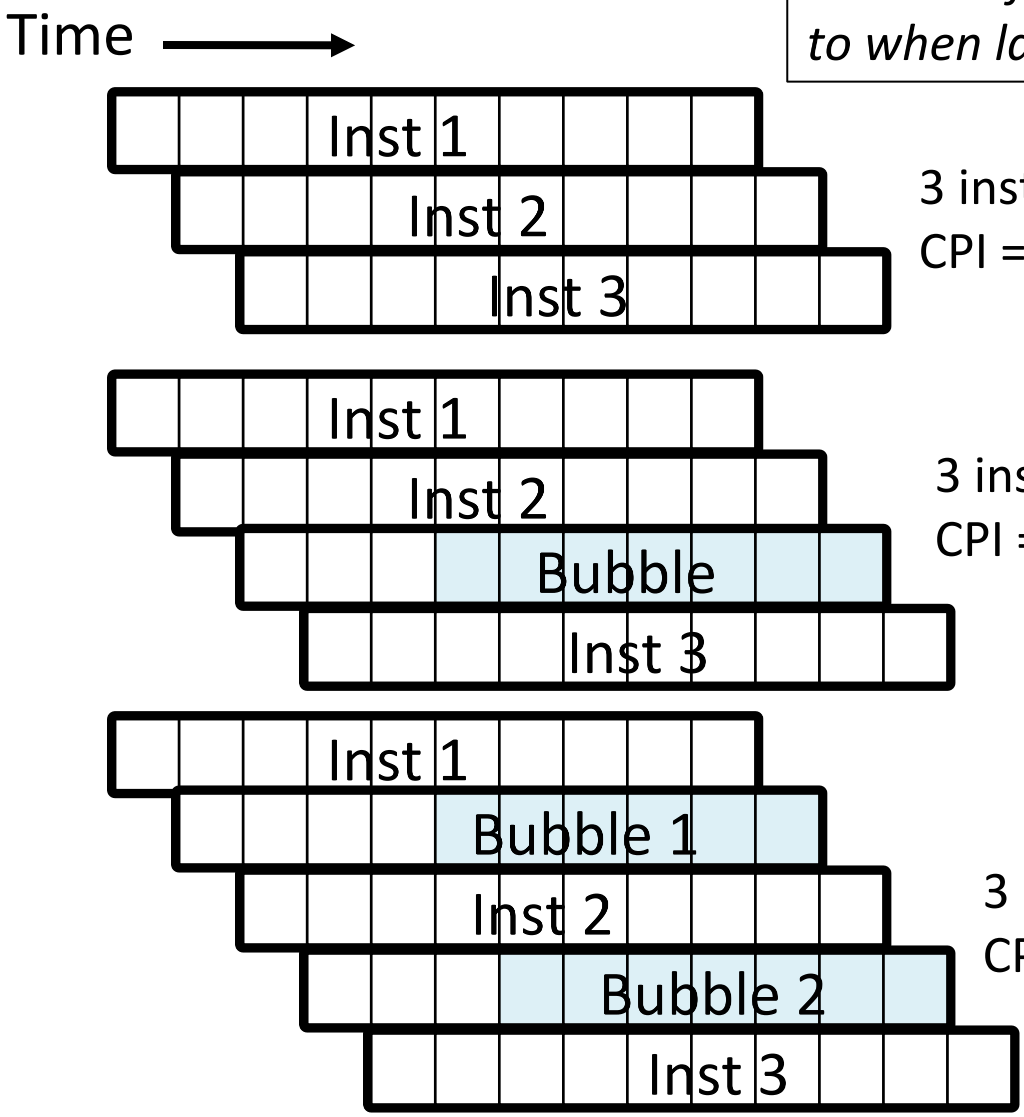
- An instruction in the pipeline may need a resource being used by another instruction in the pipeline
  → *structural hazard*

- An instruction may depend on something produced by an earlier instruction
  - Dependence may be for a data value
    → *data hazard*
  - Dependence may be for the next instruction's address
    → *control hazard (branches, exceptions)*

- Handling hazards generally introduces bubbles into pipeline and reduces ideal CPI > 1

Pipeline CPI Examples

Measure from when first instruction finishes to when last instruction in sequence finishes.



3 instructions finish in 3 cycles
CPI = 3/3 =1

3 instructions finish in 4 cycles
CPI = 4/3 = 1.33

3 instructions finish in 5cycles
CPI = 5/3 = 1.67

Resolving Structural Hazards

- Structural hazard occurs when two instructions need same hardware resource at same time
  - Can resolve in hardware by stalling newer instruction till older instruction finished with resource
- A structural hazard can always be avoided by adding more hardware to design
  - E.g., if two instructions both need a port to memory at same time, could avoid hazard by adding second port to memory
- Classic RISC 5-stage integer pipeline has no structural hazards by design
  - Many RISC implementations have structural hazards on multi-cycle units such as multipliers, dividers, floating-point units, etc., and can have on register writeback ports

Types of Data Hazards

Consider executing a sequence of register-register instructions of type:

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

Data-dependence

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

Read-after-Write (RAW) hazard

Anti-dependence

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

Write-after-Read (WAR) hazard

Output-dependence

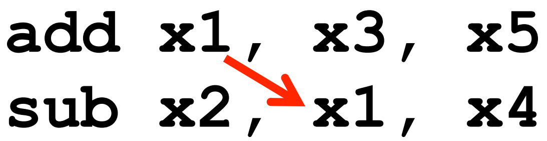
$$\begin{tikzcd}[row sep=small, column sep=small]
r_3 \arrow[d, bend right] & \leftarrow & r_1 & \text{op} & r_2 \\
r_3 & \leftarrow & r_6 & \text{op} & r_7
\end{tikzcd}$$

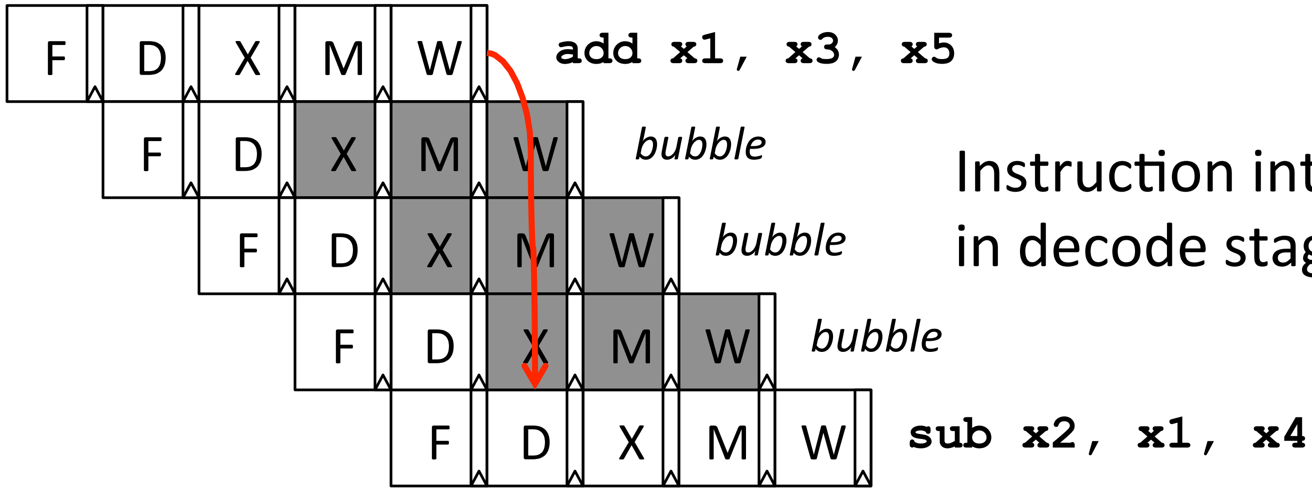
Write-after-Write (WAW) hazard

Three Strategies for Data Hazards

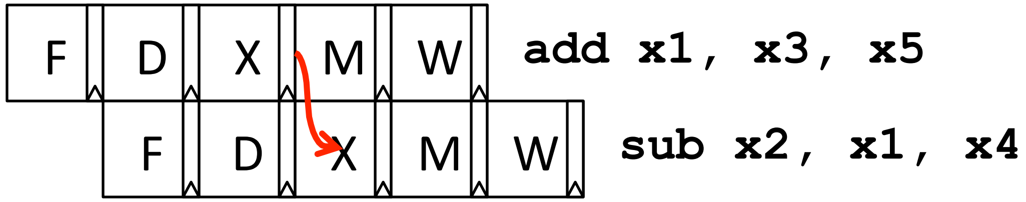
- Interlock
  - Wait for hazard to clear by holding dependent instruction in issue stage
- Bypass
  - Resolve hazard earlier by bypassing value as soon as available
- Speculate
  - Guess on value, correct if wrong

Interlocking Versus Bypassing



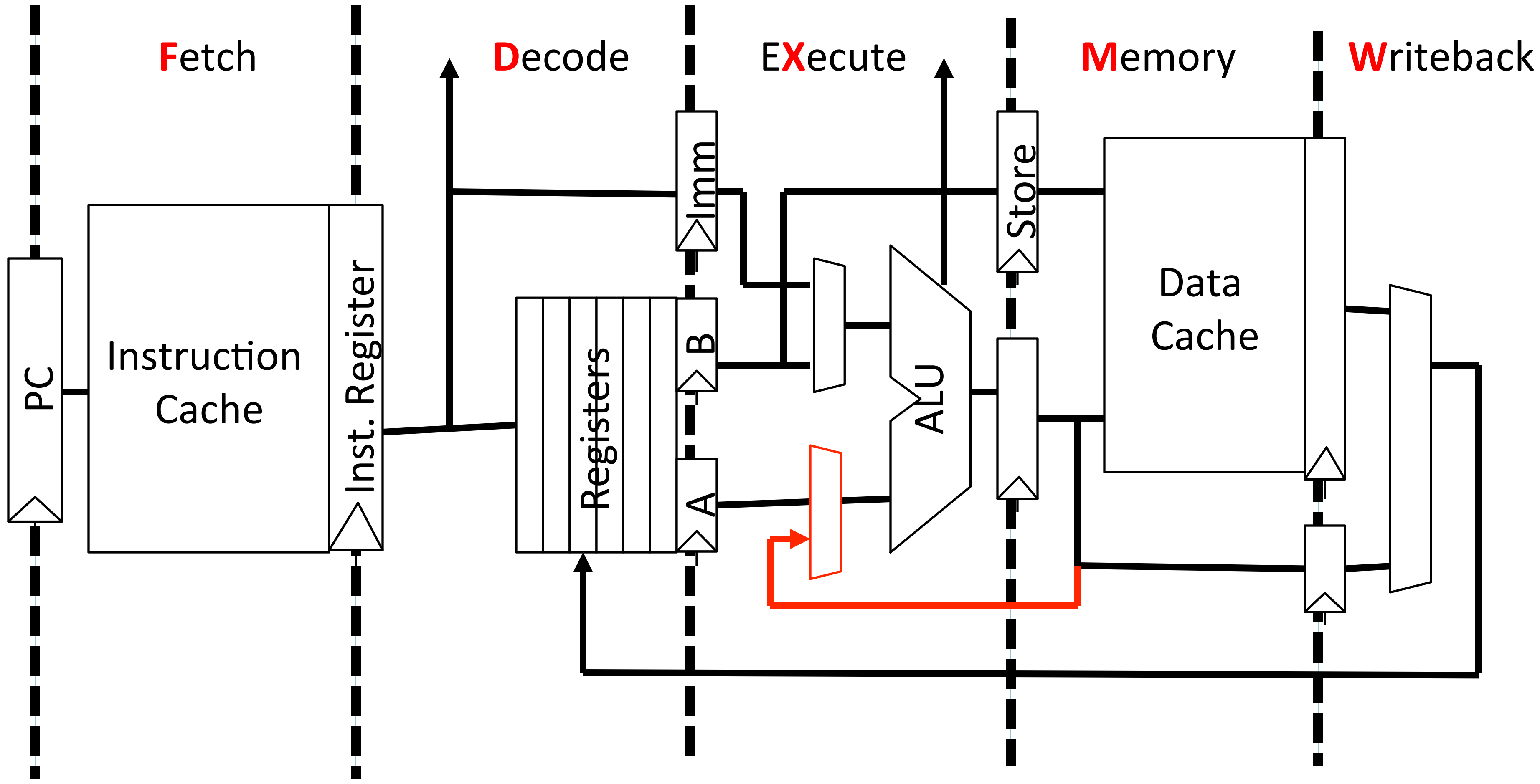


Instruction interlocked in decode stage

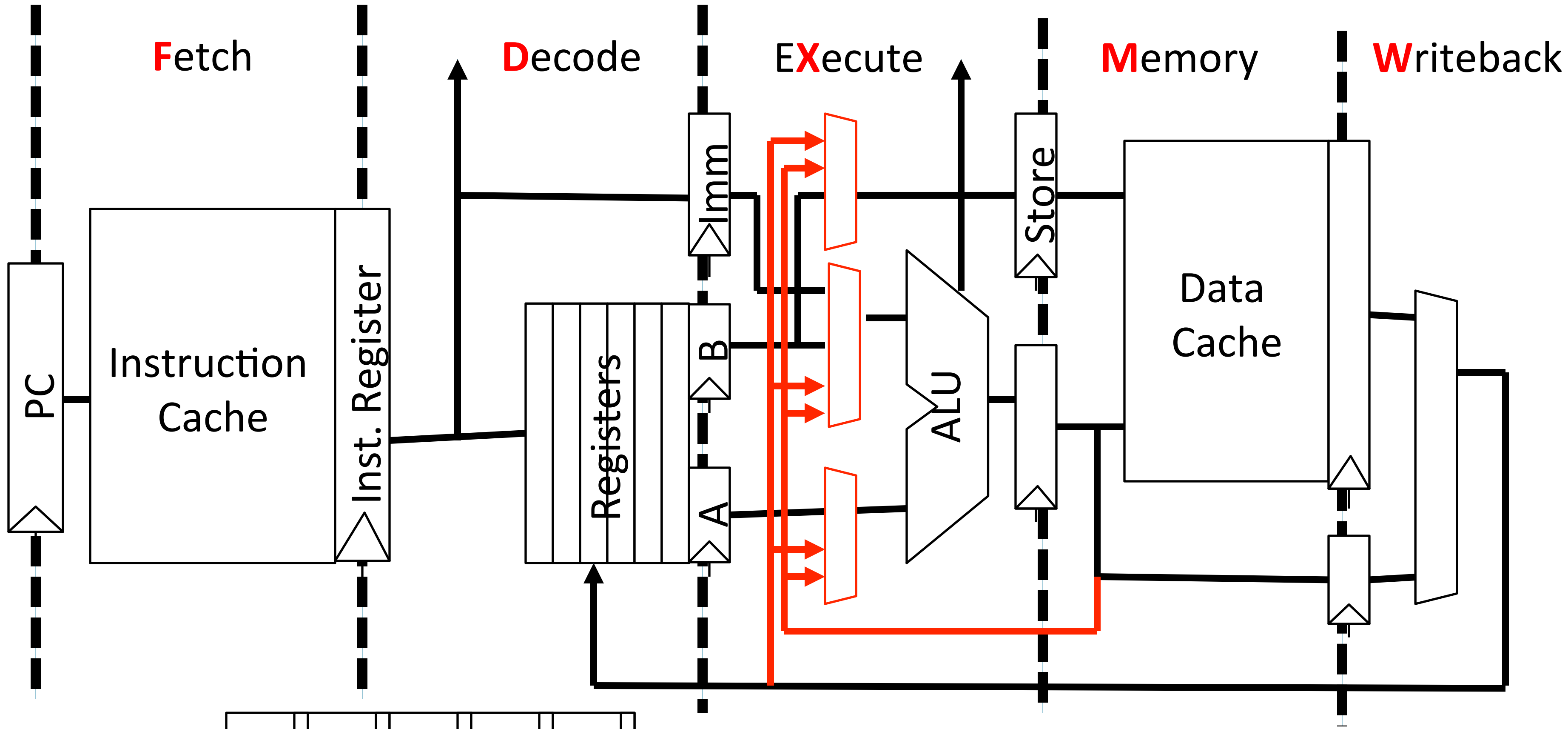


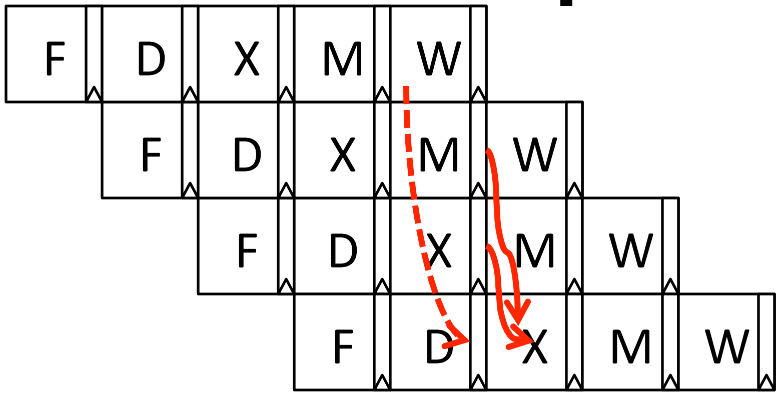
Bypass around ALU with no bubbles

Example Bypass Path



Fully Bypassed Data Path





[Assumes data written to registers in a W cycle is readable in parallel D cycle (dotted line). Extra write data register and bypass paths required if this is not possible.]

Value Speculation for RAW Data Hazards

- Rather than wait for value, can guess value!

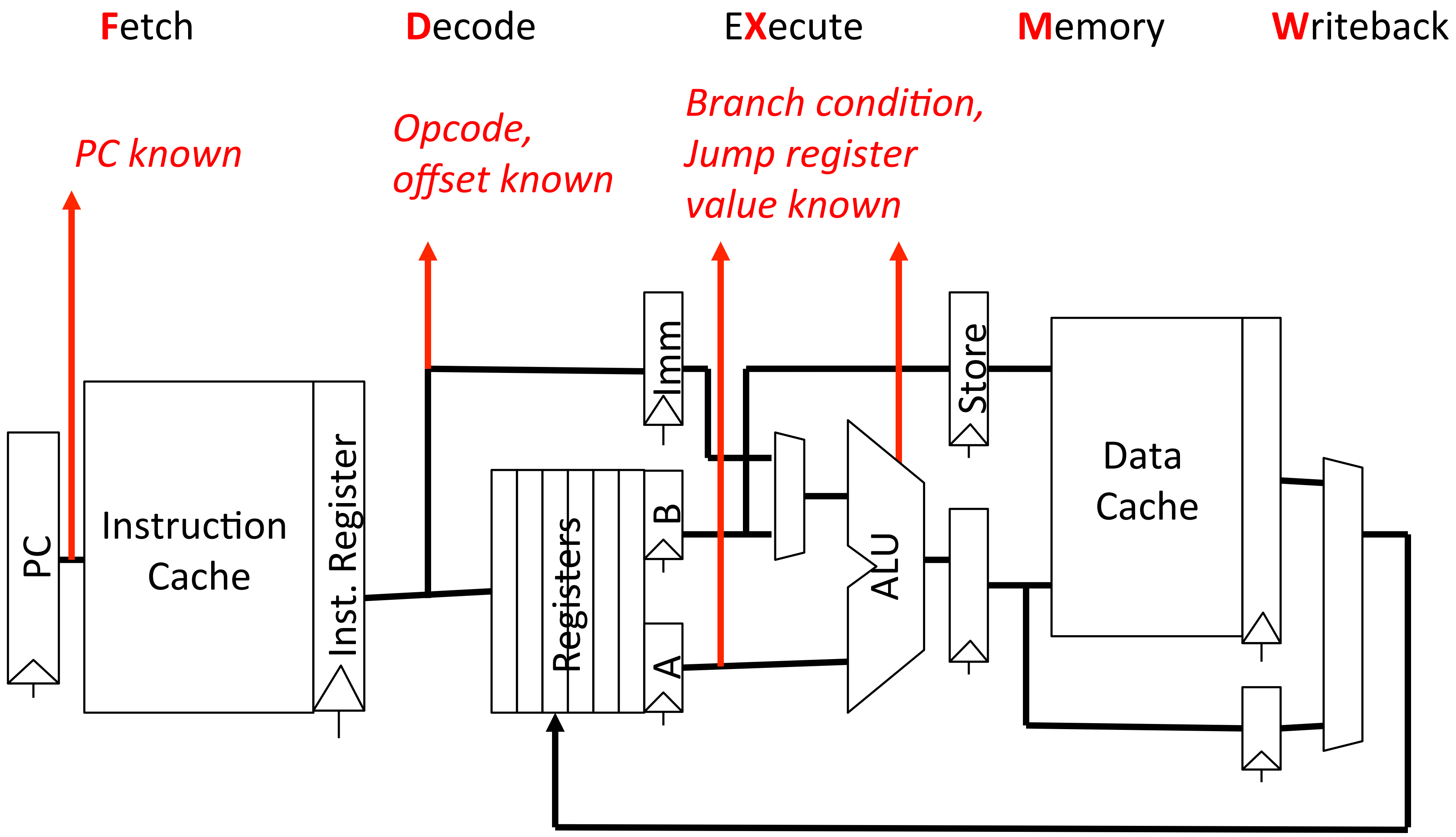
- So far, only effective in certain limited cases:
  - Branch prediction
  - Stack pointer updates
  - Memory address disambiguation

Control Hazards

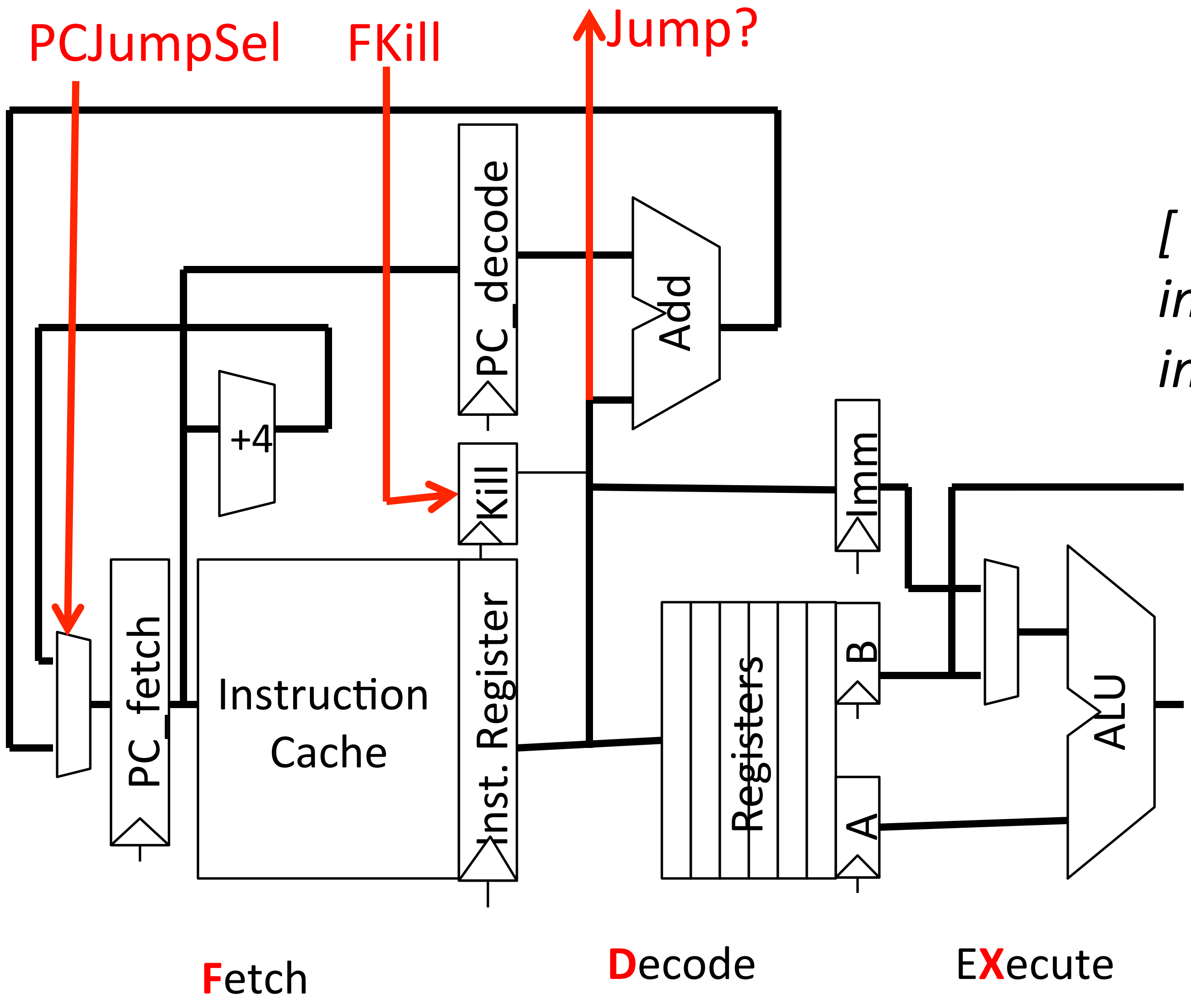
What do we need to calculate next PC?

- For Unconditional Jumps
  - Opcode, PC, and offset
- For Jump Register
  - Opcode, Register value, and offset
- For Conditional Branches
  - Opcode, Register (for condition), PC and offset
- For all other instructions
  - Opcode and PC (and have to know it's not one of above)

Control flow information in pipeline

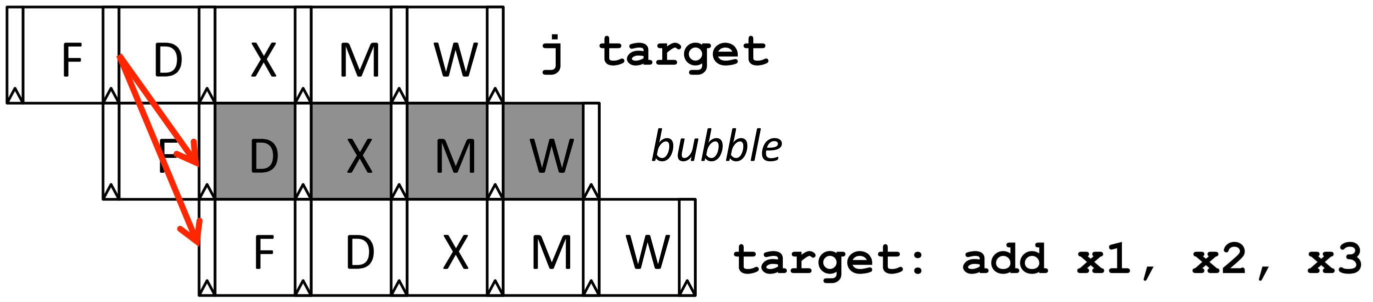


RISC-V Unconditional PC-Relative Jumps



[Kill bit turns instruction into a bubble]

Pipelining for Unconditional PC-Relative Jumps

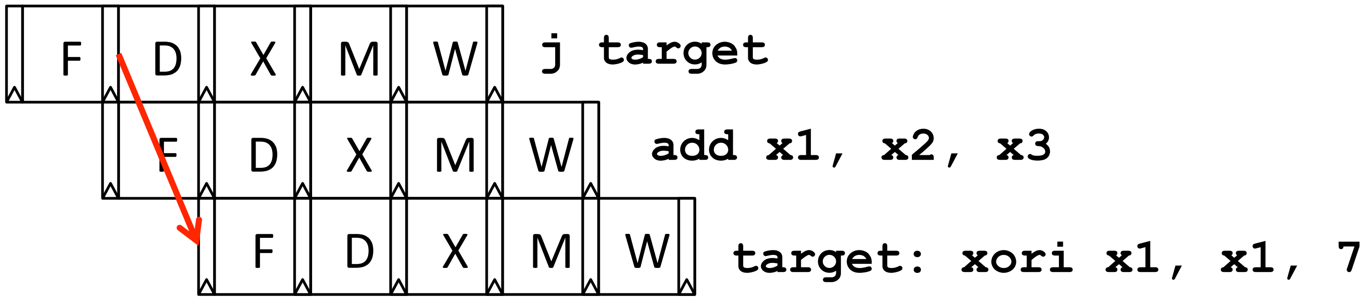


Branch Delay Slots

- Early RISCs adopted idea from pipelined microcode engines, and changed ISA semantics so instruction *after* branch/jump is always executed before control flow change occurs:

```
0x100 j target
0x104 add x1, x2, x3 // Executed before target
…
0x205 target: xori x1, x1, 7
```

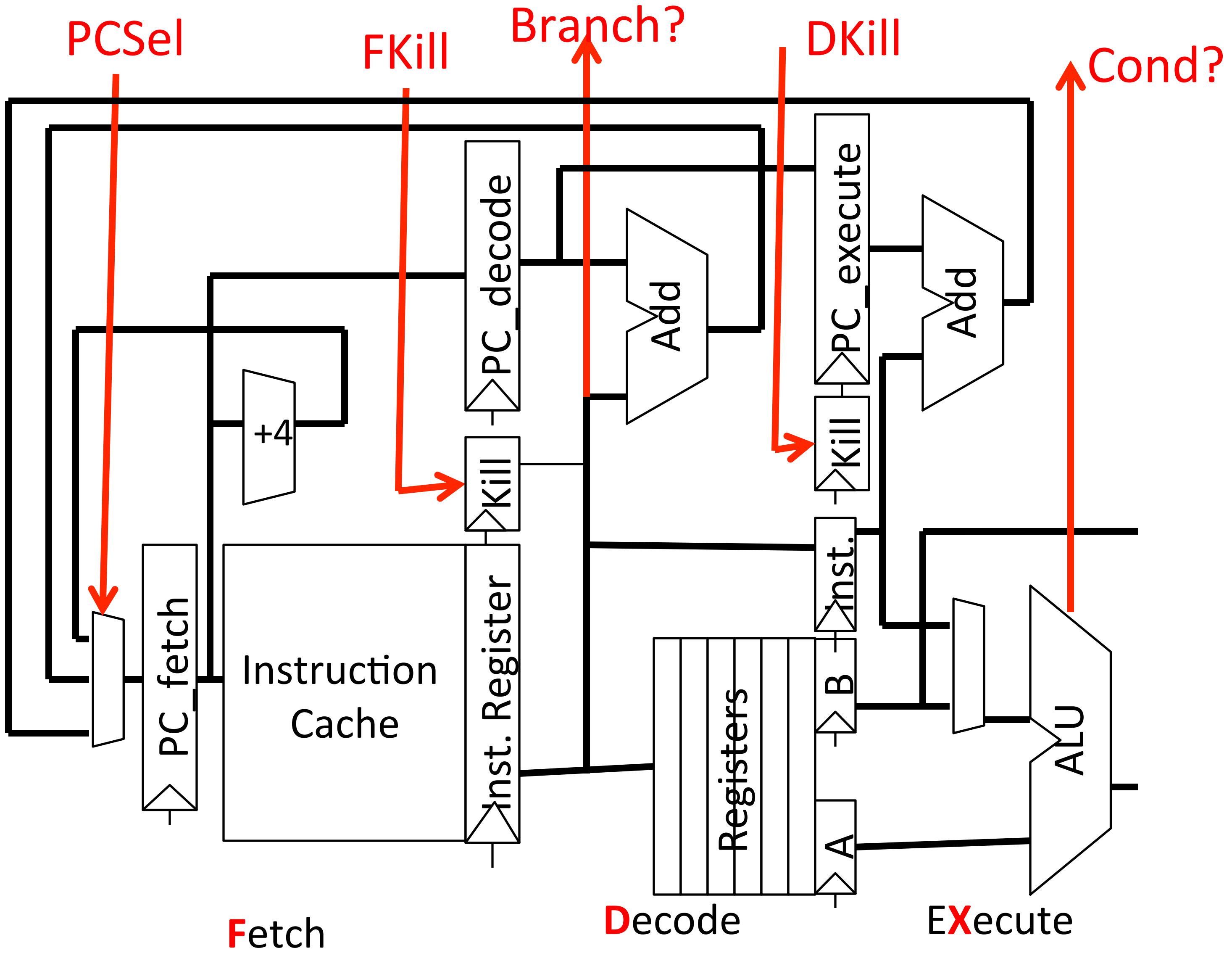
- Software has to fill delay slot with useful work, or fill with explicit NOP instruction



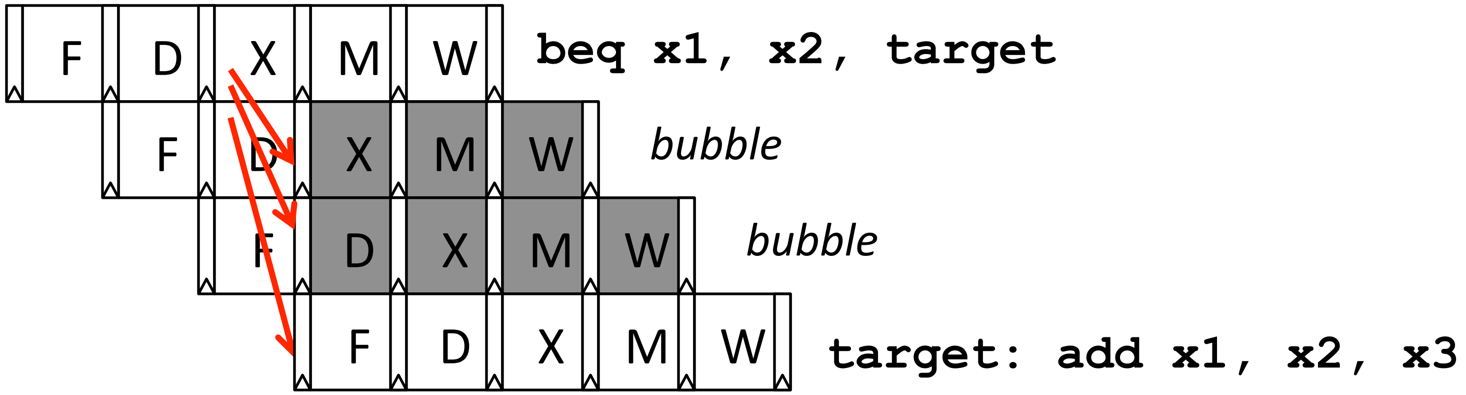
Post-1990 RISC ISAs don't have delay slots

- Encodes microarchitectural detail into ISA
  - c.f. IBM 650 drum layout
- Performance issues
  - Increased I-cache misses from NOPs in unused delay slots
  - I-cache miss on delay slot causes machine to wait, even if delay slot is a NOP
- Complicates more advanced microarchitectures
  - Consider 30-stage pipeline with four-instruction-per-cycle issue
- Better branch prediction reduced need
  - Branch prediction in later lecture

RISC-V Conditional Branches

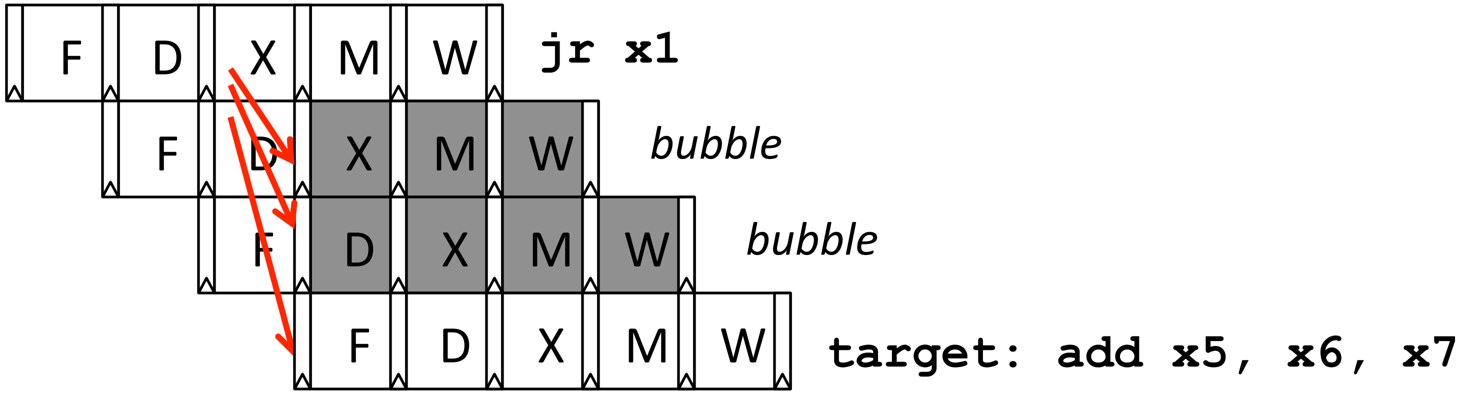


Pipelining for Conditional Branches



Pipelining for Jump Register

- Register value obtained in execute stage



Why instruction may not be dispatched every cycle in classic 5-stage pipeline (CPI>1)

- Full bypassing may be too expensive to implement
  - typically all frequently used paths are provided
  - some infrequently used bypass paths may increase cycle time and counteract the benefit of reducing CPI
- Loads have two-cycle latency
  - Instruction after load cannot use load result
  - MIPS-I ISA defined *load delay slots*, a software-visible pipeline hazard (compiler schedules independent instruction or inserts NOP to avoid hazard). Removed in MIPS-II (pipeline interlocks added in hardware)
    - MIPS:"**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages"
- Jumps/Conditional branches may cause bubbles
  - kill following instruction(s) if no delay slots

Machines with software-visible delay slots may execute significant number of NOP instructions inserted by the compiler.
NOPs reduce CPI, but increase instructions/program!

Traps and Interrupts

In class, we'll use following terminology

- ***Exception***: An unusual internal event caused by program during execution
  - E.g., page fault, arithmetic underflow
- ***Trap***: Forced transfer of control to supervisor caused by exception
  - Not all exceptions cause traps (c.f. IEEE 754 floating-point standard)
- ***Interrupt***: An external event outside of running program, which causes transfer of control to supervisor
- Traps and interrupts usually handled by same pipeline mechanism

History of Exception Handling

- (Analytical Engine had overflow exceptions)
- First system with traps was Univac-I, 1951
  - Arithmetic overflow would either
    - 1. trigger the execution a two-instruction fix-up routine at address 0, or
    - 2. at the programmer's option, cause the computer to stop
  - Later Univac 1103, 1955, modified to add external interrupts
    - Used to gather real-time wind tunnel data
- First system with I/O interrupts was DYSEAC, 1954
  - Had two program counters, and I/O signal caused switch between two PCs
  - Also, first system with DMA (direct memory access by I/O device)
  - And, first mobile computer (two tractor trailers, 12 tons + 8 tons)

Asynchronous Interrupts

- An I/O device requests attention by asserting one of the *prioritized interrupt request lines*
- When the processor decides to process the interrupt
  - It stops the current program at instruction I_i, completing all the instructions up to I_{i-1} (*precise interrupt)*
  - It saves the PC of instruction I_i in a special register (EPC)
  - It disables interrupts and transfers control to a designated interrupt handler running in the kernel mode

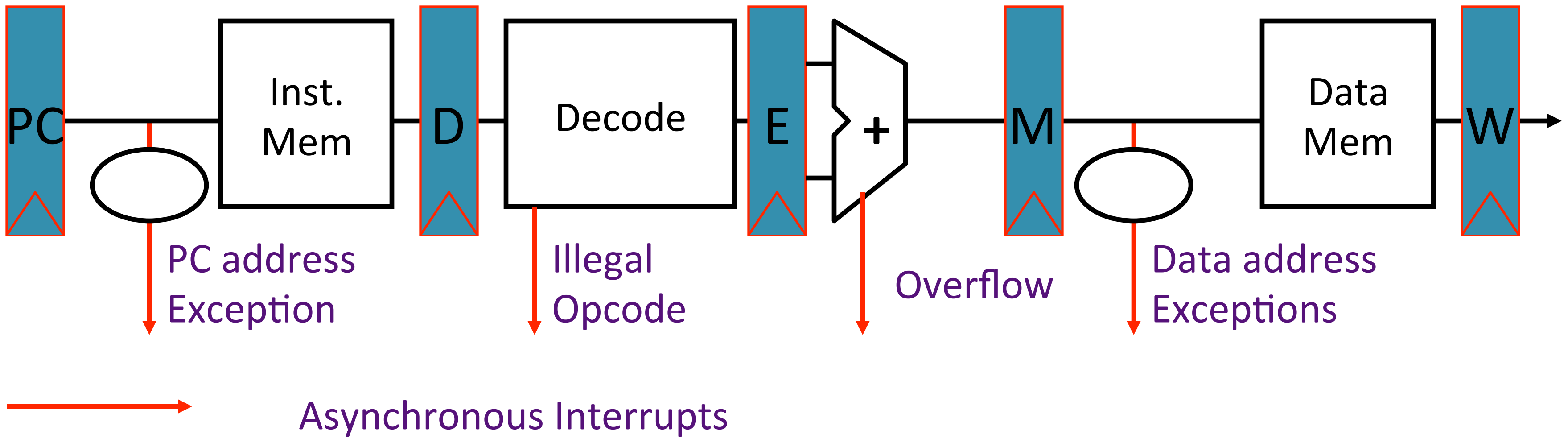
Interrupt Handler

- Saves EPC before enabling interrupts to allow nested interrupts $\Rightarrow$
  - need an instruction to move EPC into GPRs
  - need a way to mask further interrupts at least until EPC can be saved
- Needs to read a *status register* that indicates the cause of the interrupt
- Uses a special indirect jump instruction ERET (*return-from-environment*) which
  - enables interrupts
  - restores the processor to the user mode
  - restores hardware status and control state

Synchronous Trap

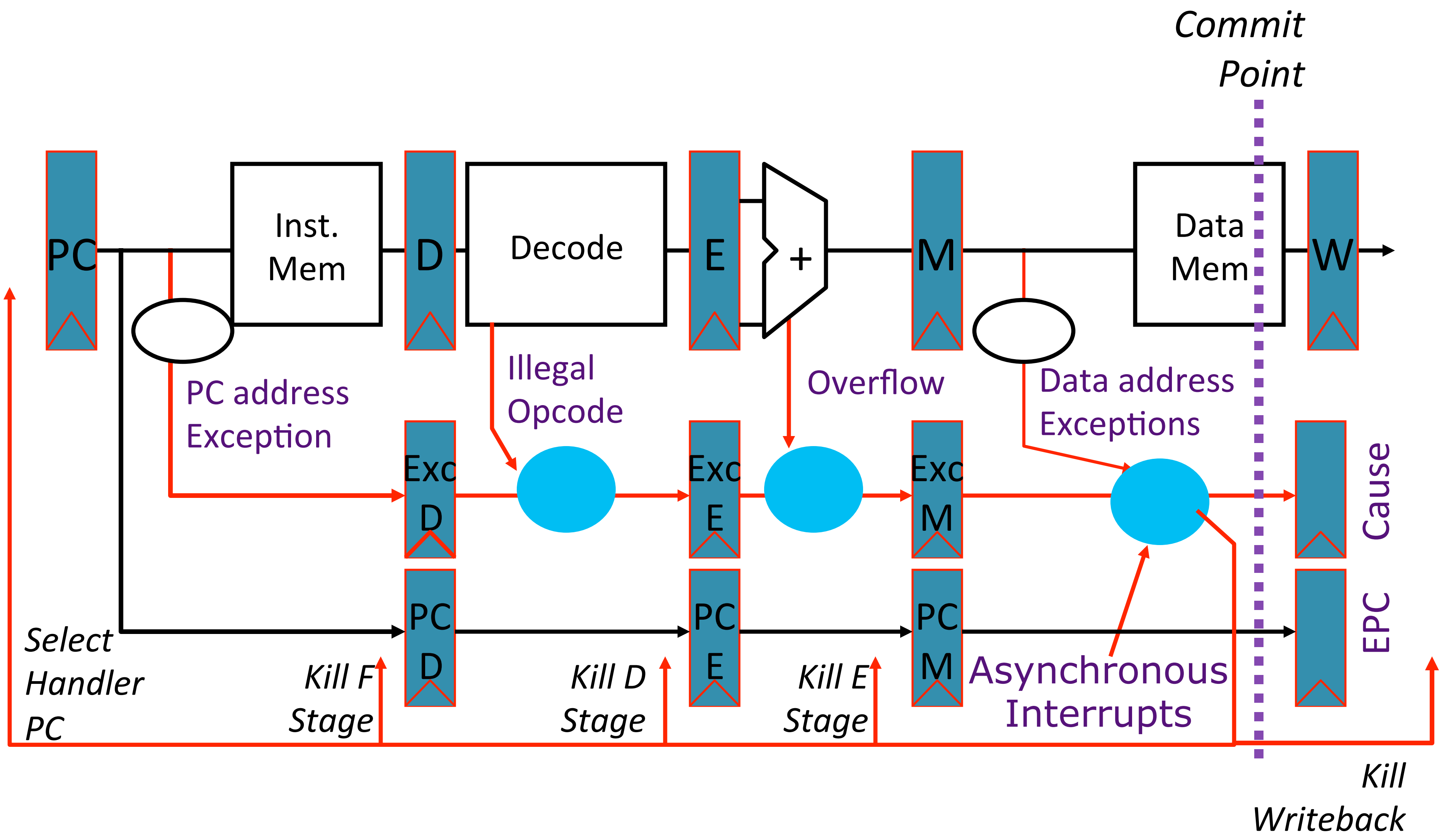
- A synchronous trap is caused by an exception on a *particular instruction*
- In general, the instruction cannot be completed and needs to be *restarted* after the exception has been handled
  - requires undoing the effect of one or more partially executed instructions
- In the case of a system call trap, the instruction is considered to have been completed
  - a special jump instruction involving a change to a privileged mode

Exception Handling 5-Stage Pipeline



- How to handle multiple simultaneous exceptions in different pipeline stages?
- How and where to handle external asynchronous interrupts?

Exception Handling 5-Stage Pipeline



Exception Handling 5-Stage Pipeline

- Hold exception flags in pipeline until commit point (M stage)
- Exceptions in earlier pipe stages override later exceptions *for a given instruction*
- Inject external interrupts at commit point (override others)
- If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage

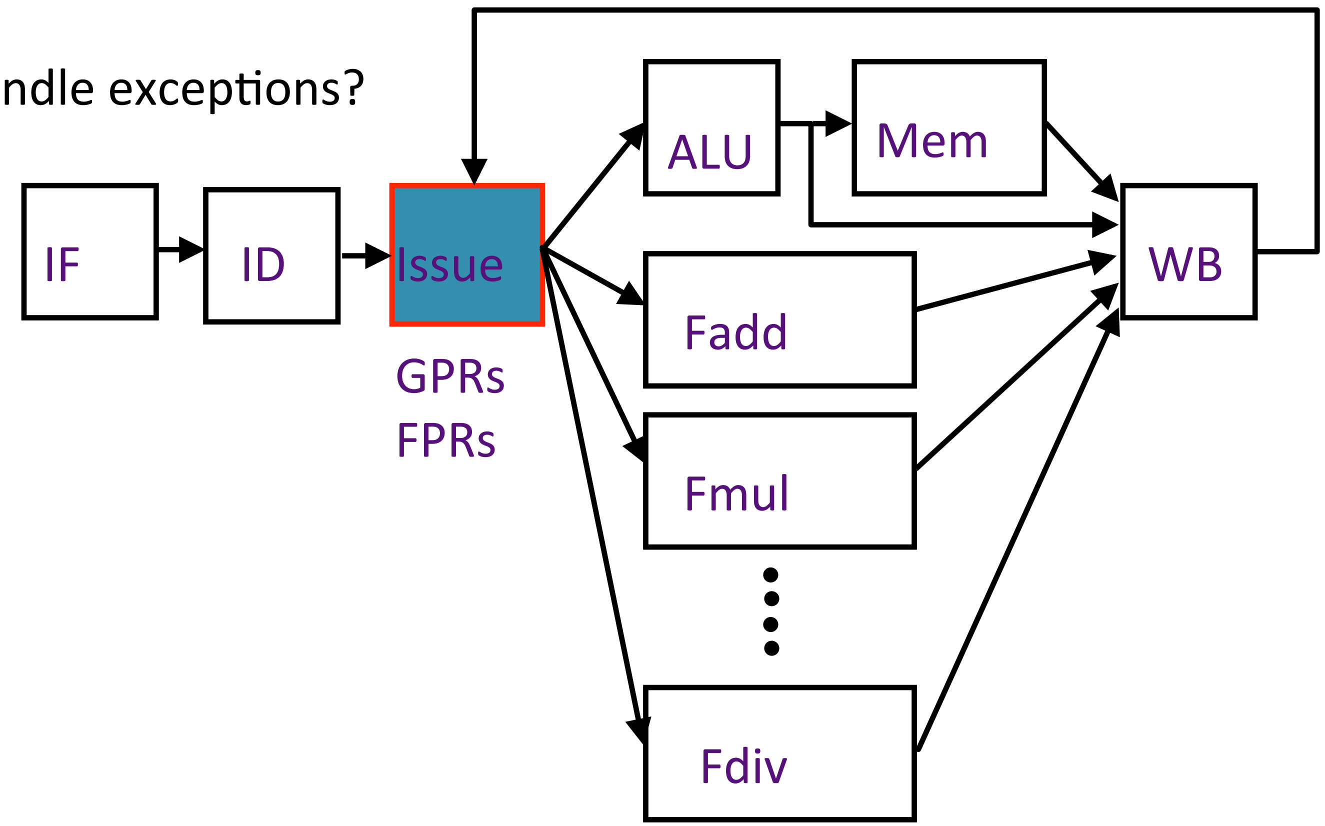
Speculating on Exceptions

- Prediction mechanism
  - Exceptions are rare, so simply predicting no exceptions is very accurate!
- Check prediction mechanism
  - Exceptions detected at end of instruction execution pipeline, special hardware for various exception types
- Recovery mechanism
  - Only write architectural state at commit point, so can throw away partially executed instructions after exception
  - Launch exception handler after flushing pipeline

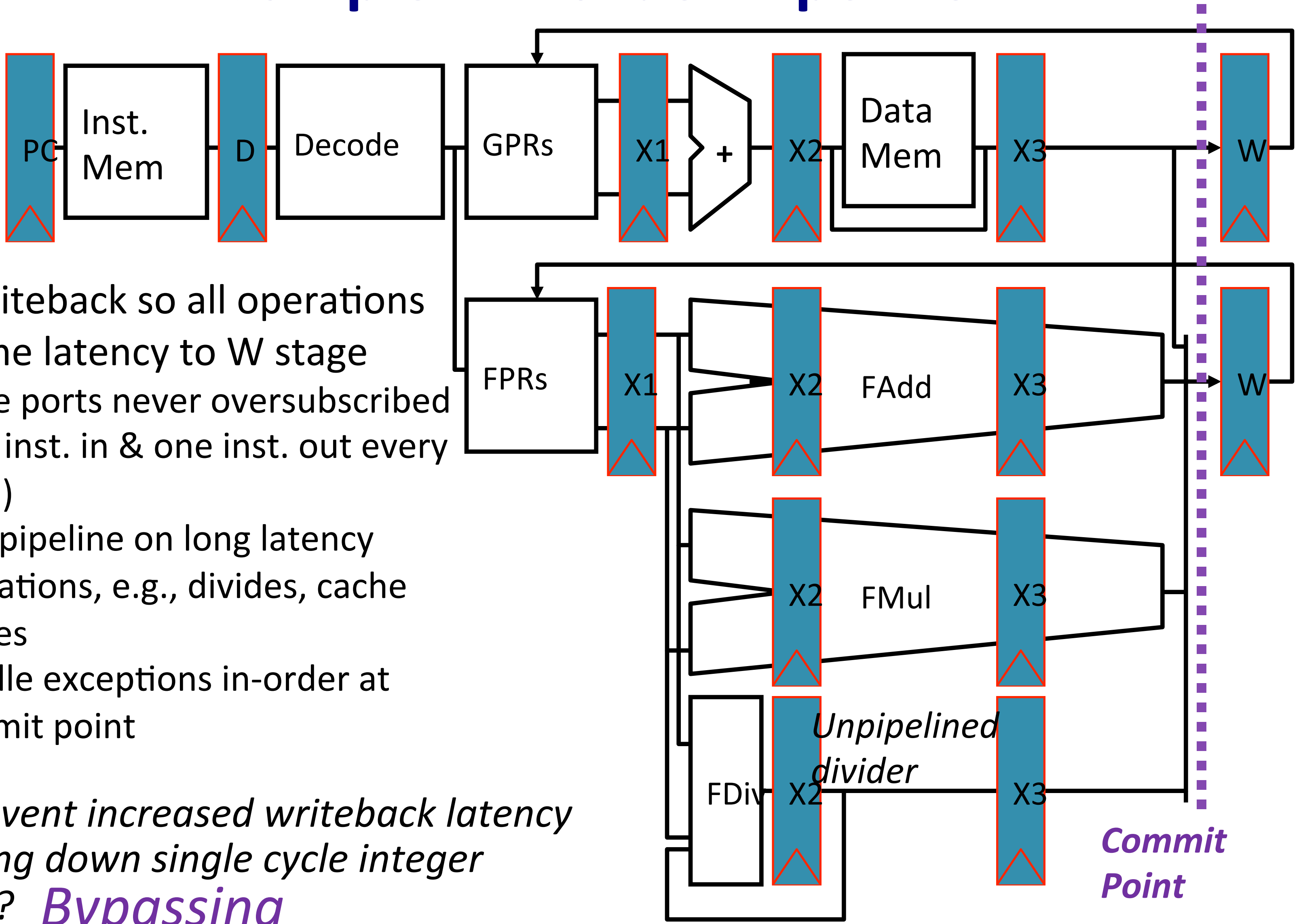
- Bypassing allows use of uncommitted instruction results by following instructions

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?



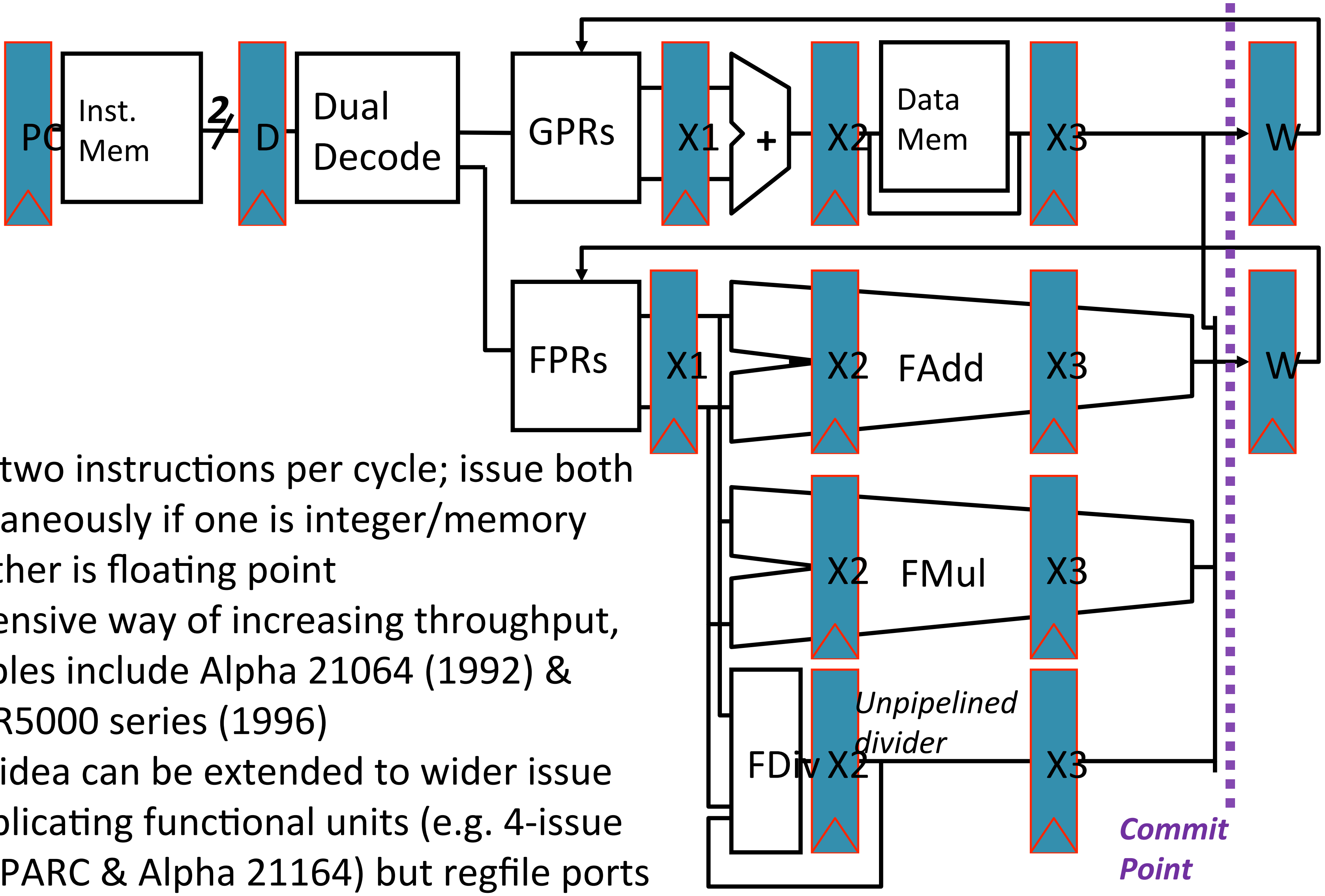
Complex In-Order Pipeline



- 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*

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

Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)