



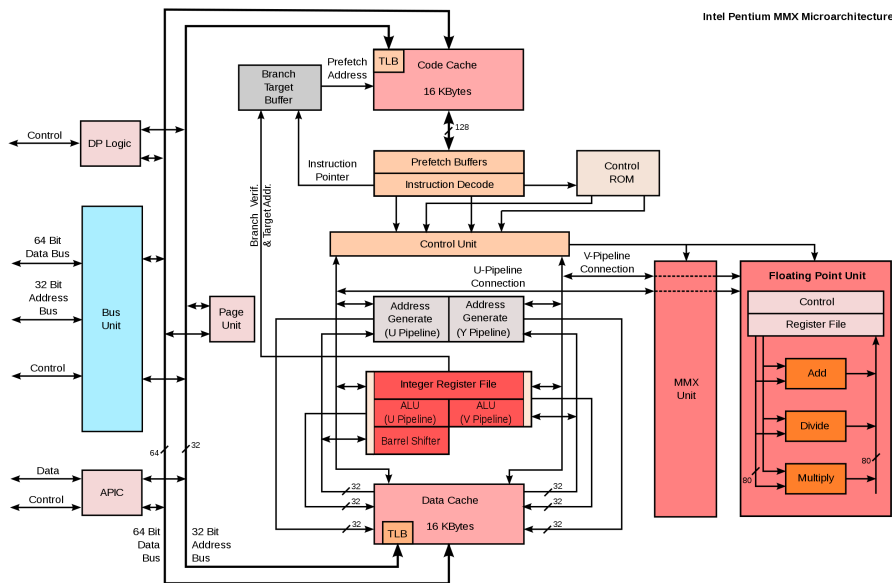
Sophia Shao

# CS 152/252A Computer Architecture and Engineering

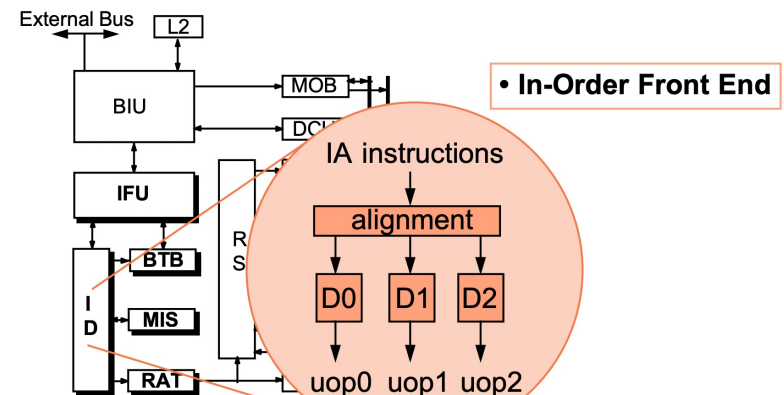
## Lecture 4 – Pipelining

### Intel P5 and P6

CISC, RISC, or CRISC??



### Implementation: Microarchitecture



## Last Time in Lecture 2

- 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

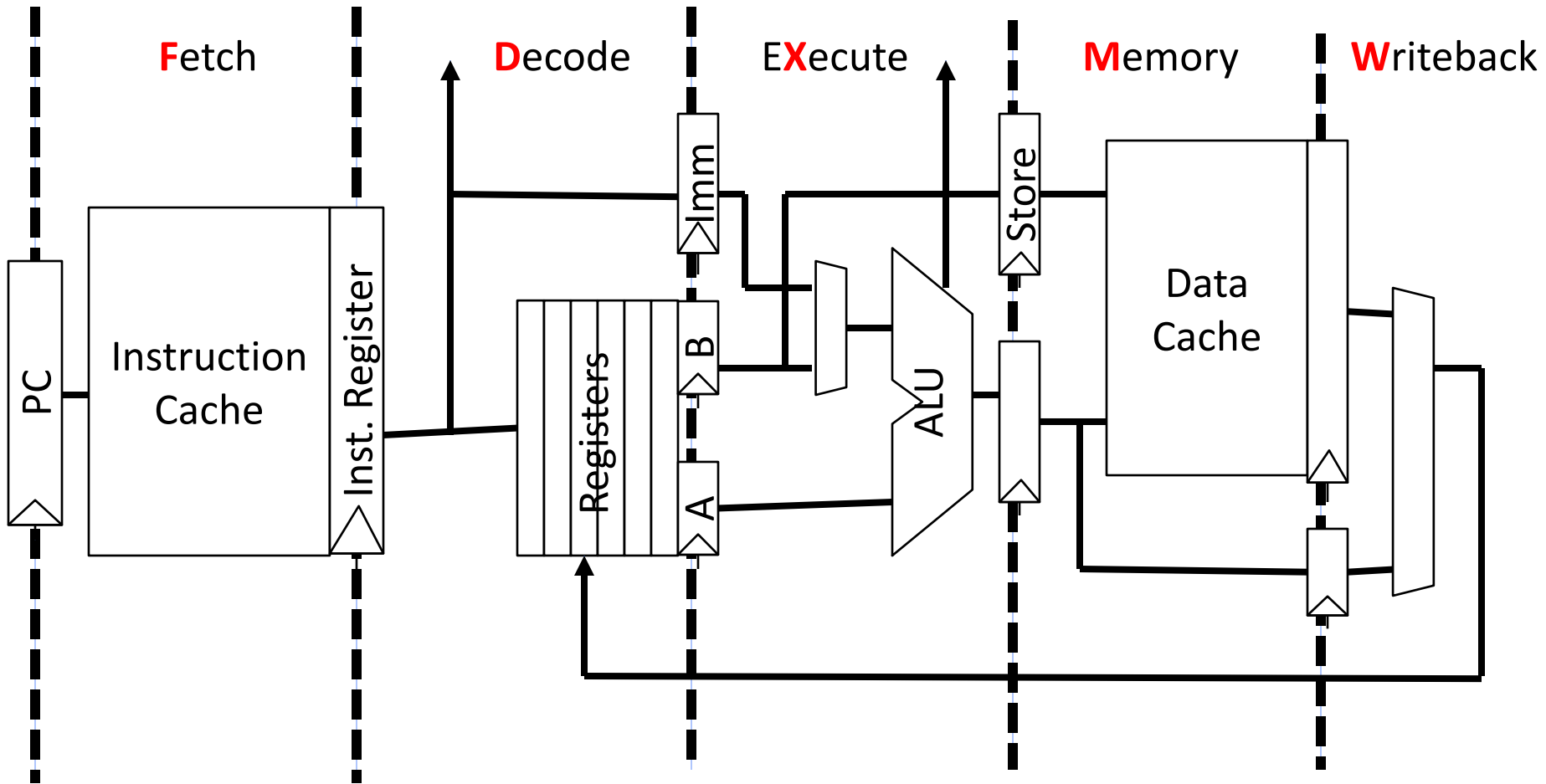
# “Iron Law” of Processor Performance

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

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

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

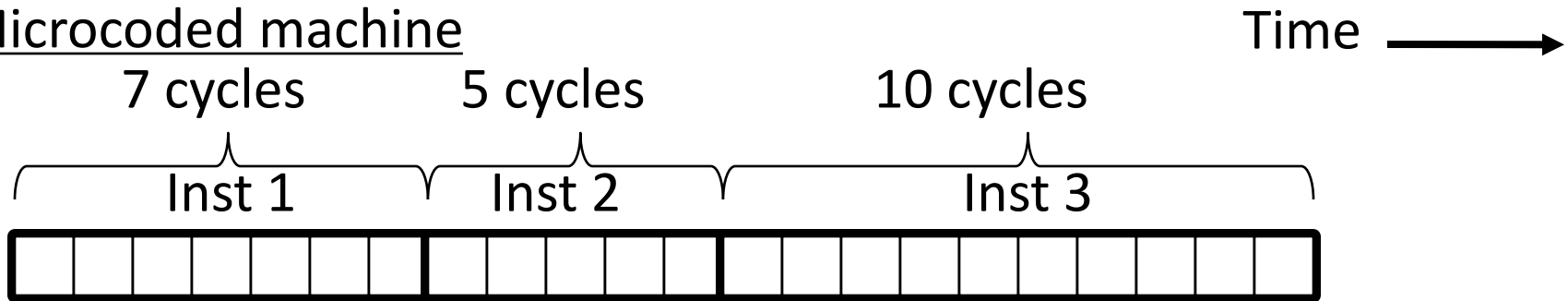
# Classic 5-Stage RISC Pipeline



*This version designed for regfiles/memories with synchronous writes and asynchronous read.*

# CPI Examples

## Microcoded machine



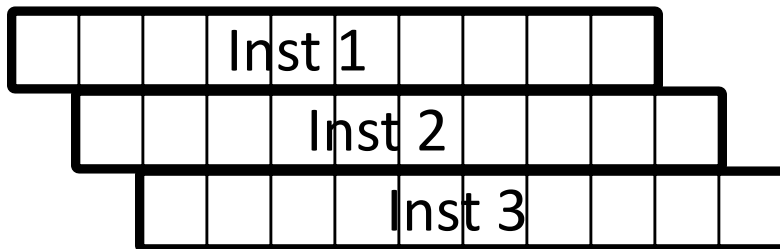
3 instructions, 22 cycles,  $CPI=7.33$

## Unpipelined machine



3 instructions, 3 cycles,  $CPI=1$

## Pipelined machine



3 instructions, 3 cycles,  $CPI=1$

**5-stage pipeline  $CPI \neq 5!!!$**

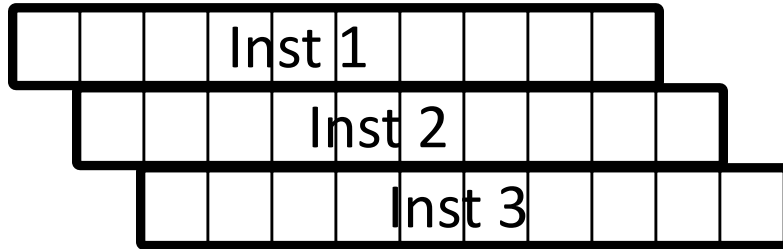
# Instructions interact with each other in pipeline

- 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  $CPI > 1$

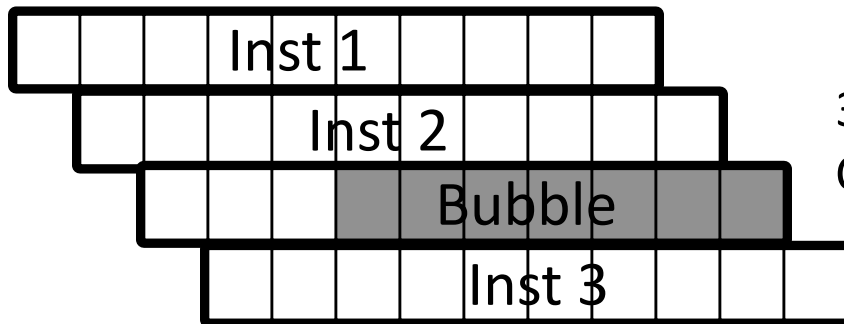
# Pipeline CPI Examples

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

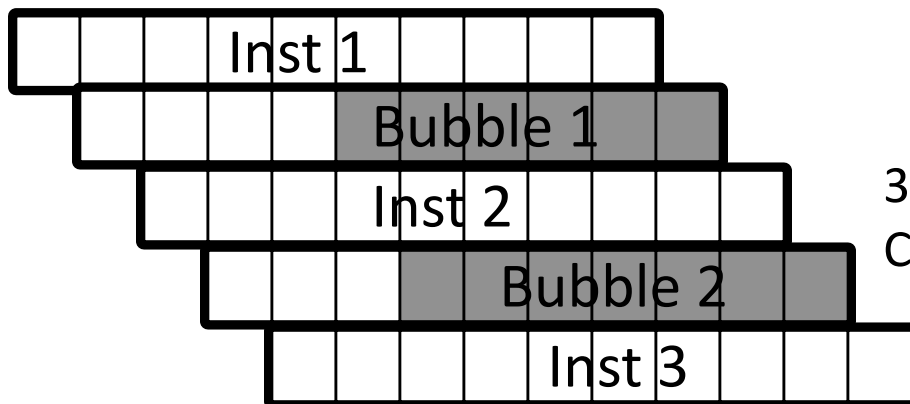
Time →



3 instructions finish in 3 cycles  
 $\text{CPI} = 3/3 = 1$



3 instructions finish in 4 cycles  
 $\text{CPI} = 4/3 = 1.33$



3 instructions finish in 5 cycles  
 $\text{CPI} = 5/3 = 1.67$

# 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

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



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

# Three Strategies for Data Hazards

- Interlock

- Wait for hazard to clear by holding dependent instruction in issue stage

- Bypass

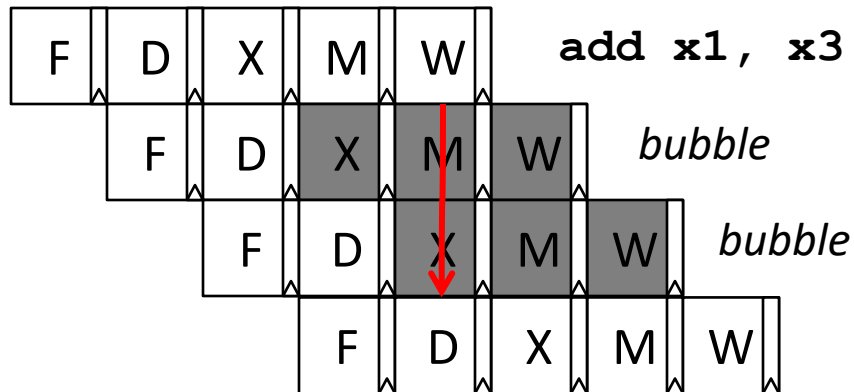
- Resolve hazard earlier by bypassing value as soon as available

- Speculate

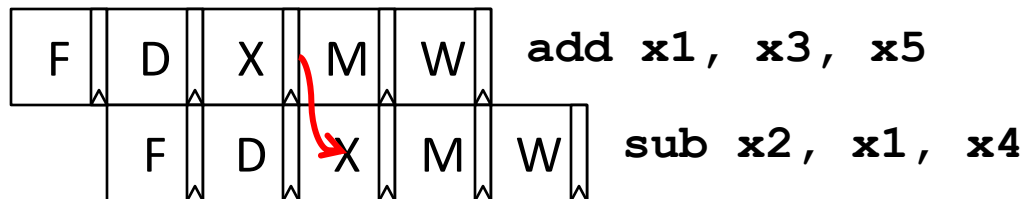
- Guess on value, correct if wrong

# Interlocking Versus Bypassing

add x1, x3, x5  
 sub x2, **x1**, x4

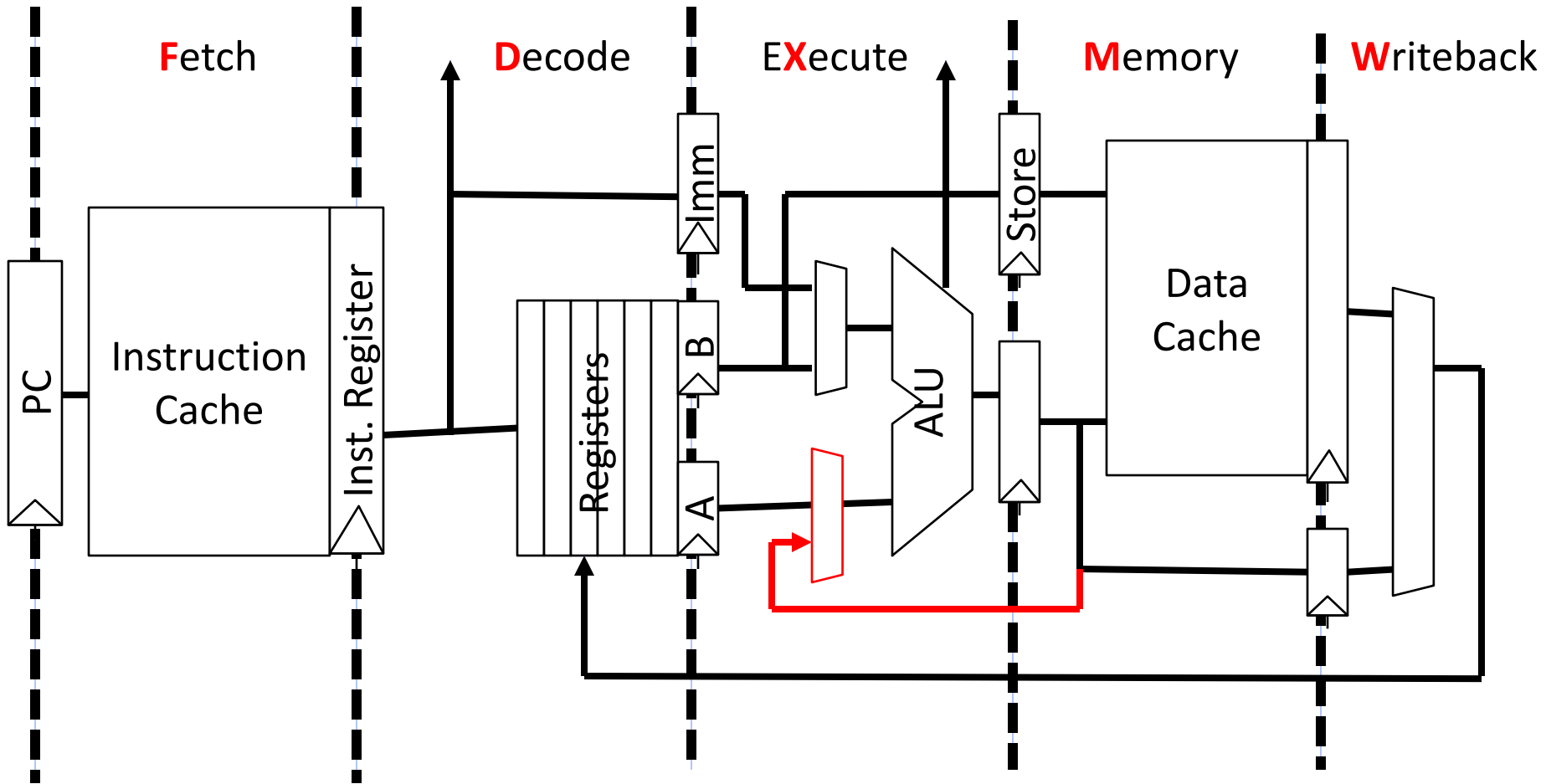


Instruction interlocked  
 in decode stage

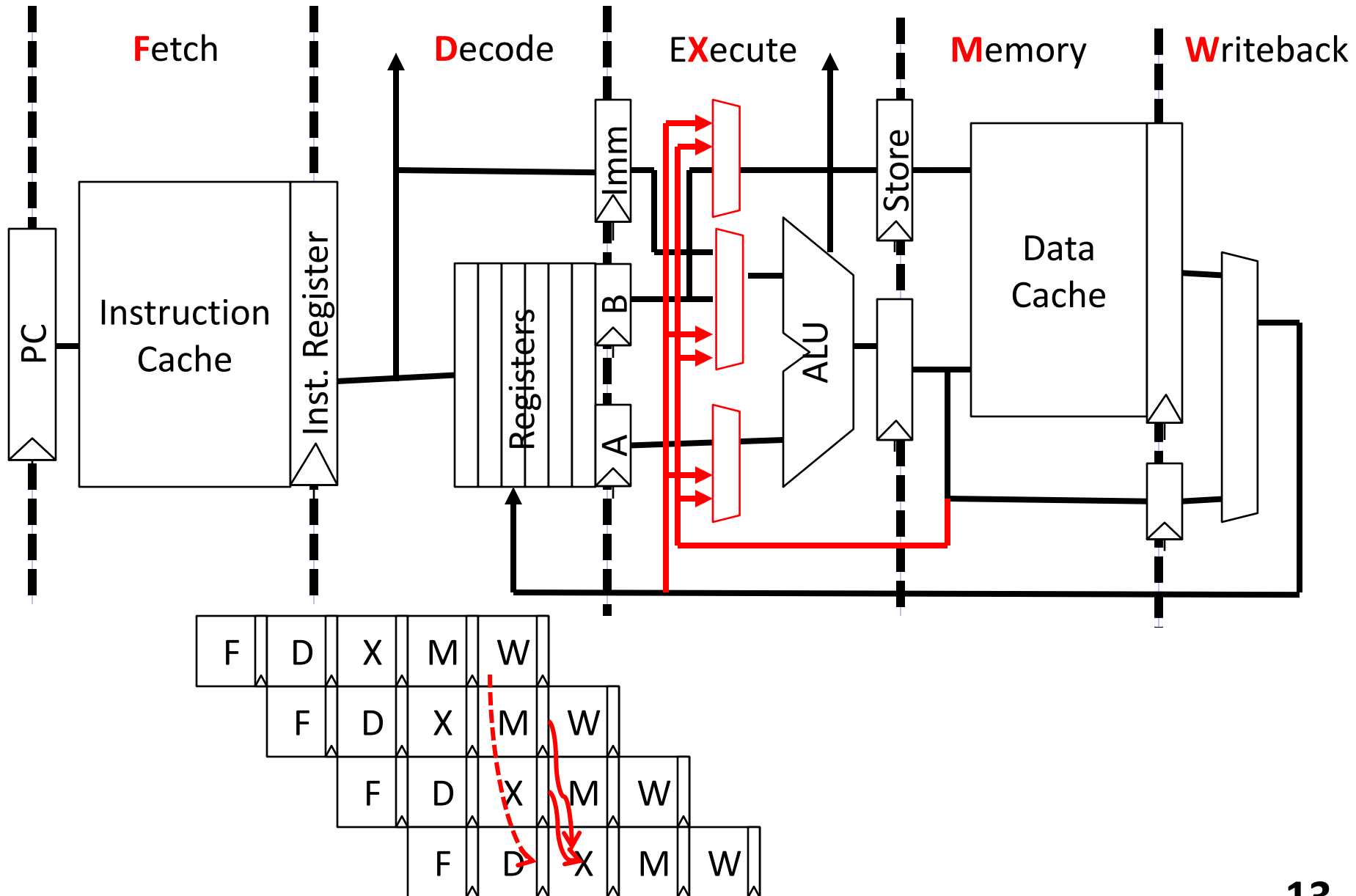


Bypass around ALU  
 with no bubbles

# Example Bypass Path



# Fully Bypassed Data Path



# Value Speculation for RAW Data Hazards

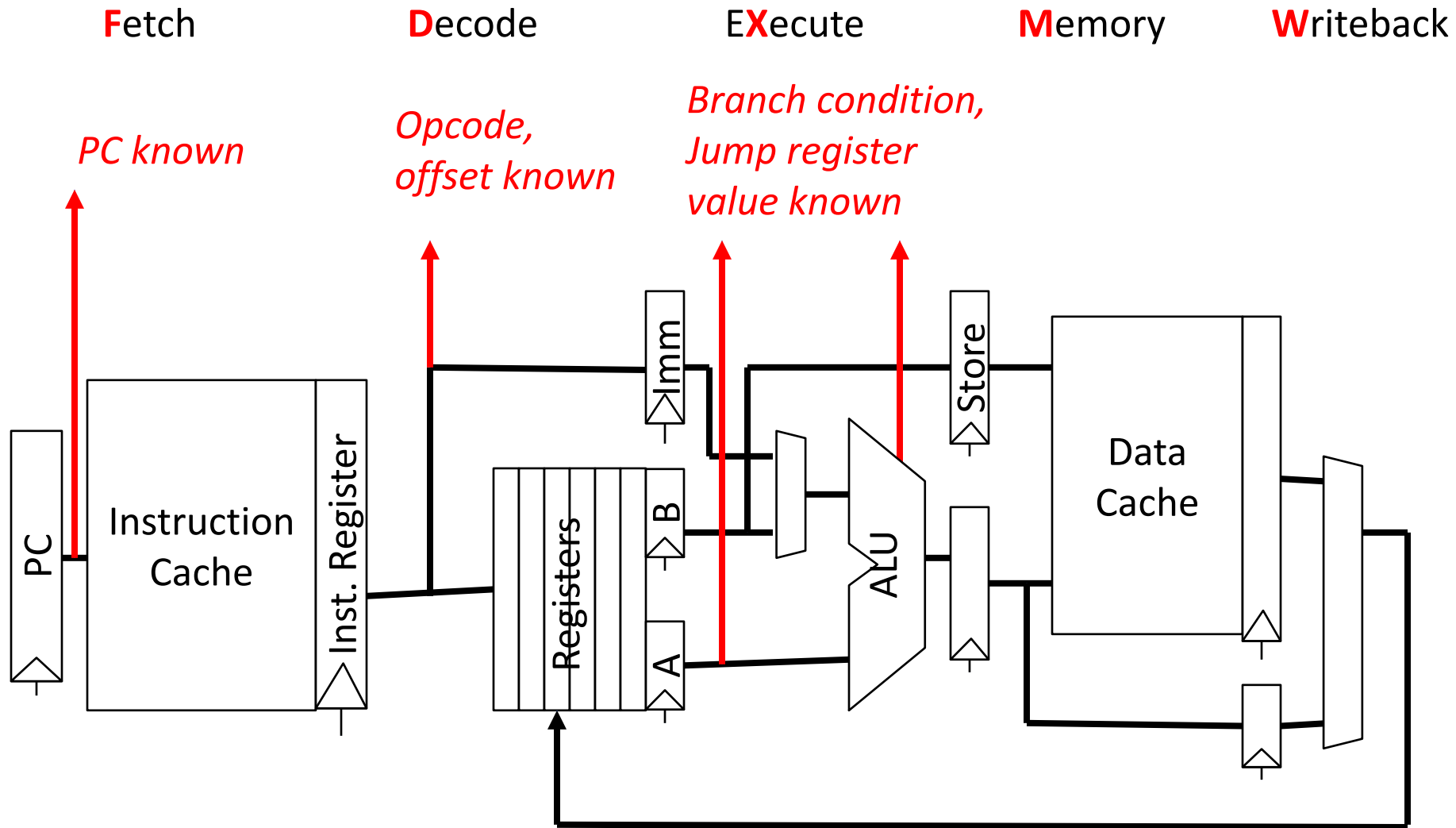
- Rather than wait for value, can guess value!
- So far, only effective in certain limited cases:
  - Branch prediction
  - Stack pointer updates
  - Memory address disambiguation

# Control Hazards

What do we need to calculate next PC?

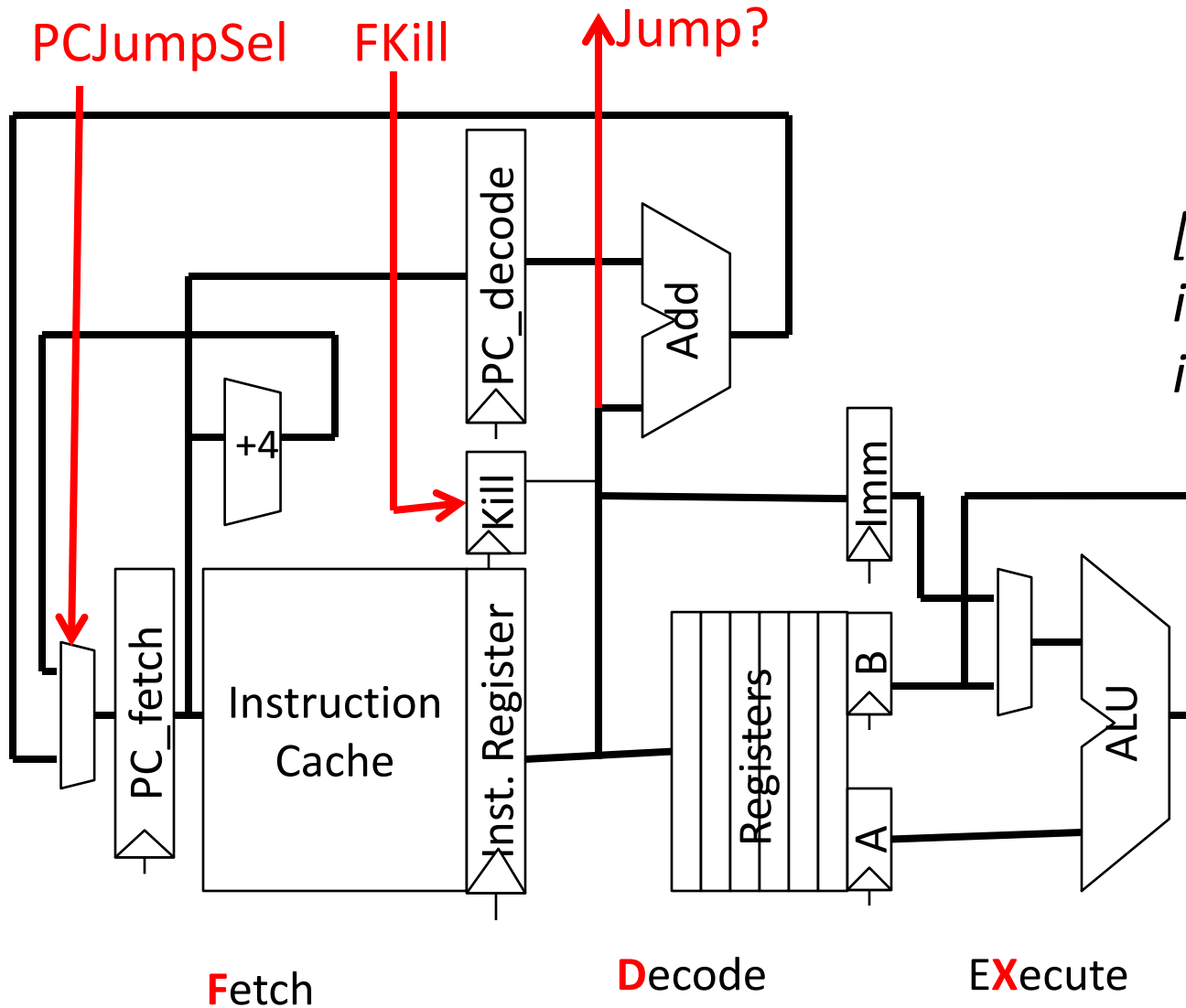
- For Unconditional Jumps
  - Opcode, PC, and offset
- For Jump Register
  - Opcode, Register value, and offset
- For Conditional Branches
  - Opcode, Register (for condition), PC and offset
- For all other instructions
  - Opcode and PC ( and have to know it's not one of above )

# Control flow information in pipeline



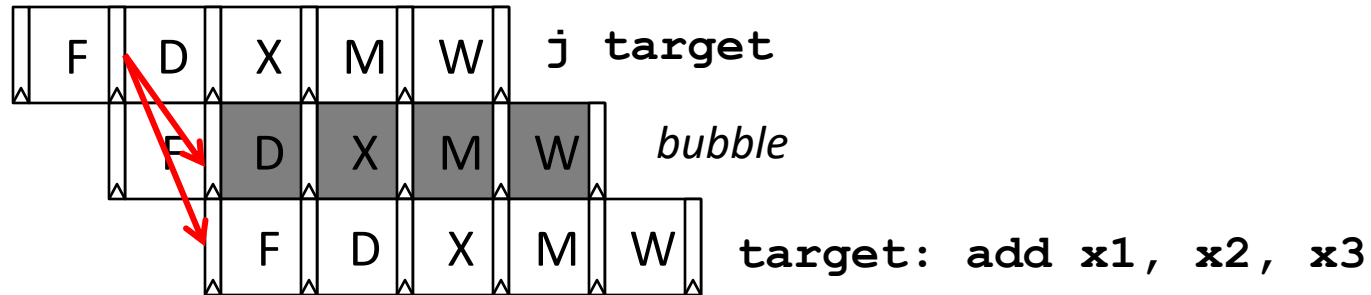


# RISC-V Unconditional PC-Relative Jumps



*[ 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:

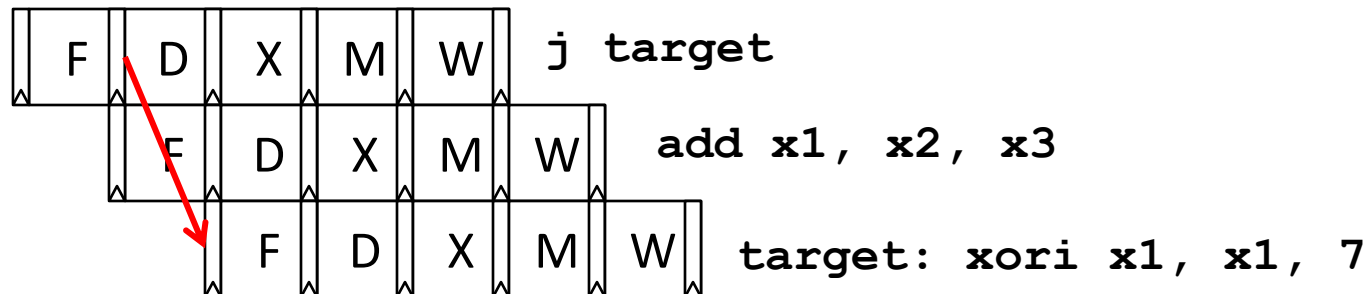
0x100 j target

0x104 add x1, x2, x3 // Executed before target

...

0x205 target: xori x1, x1, 7

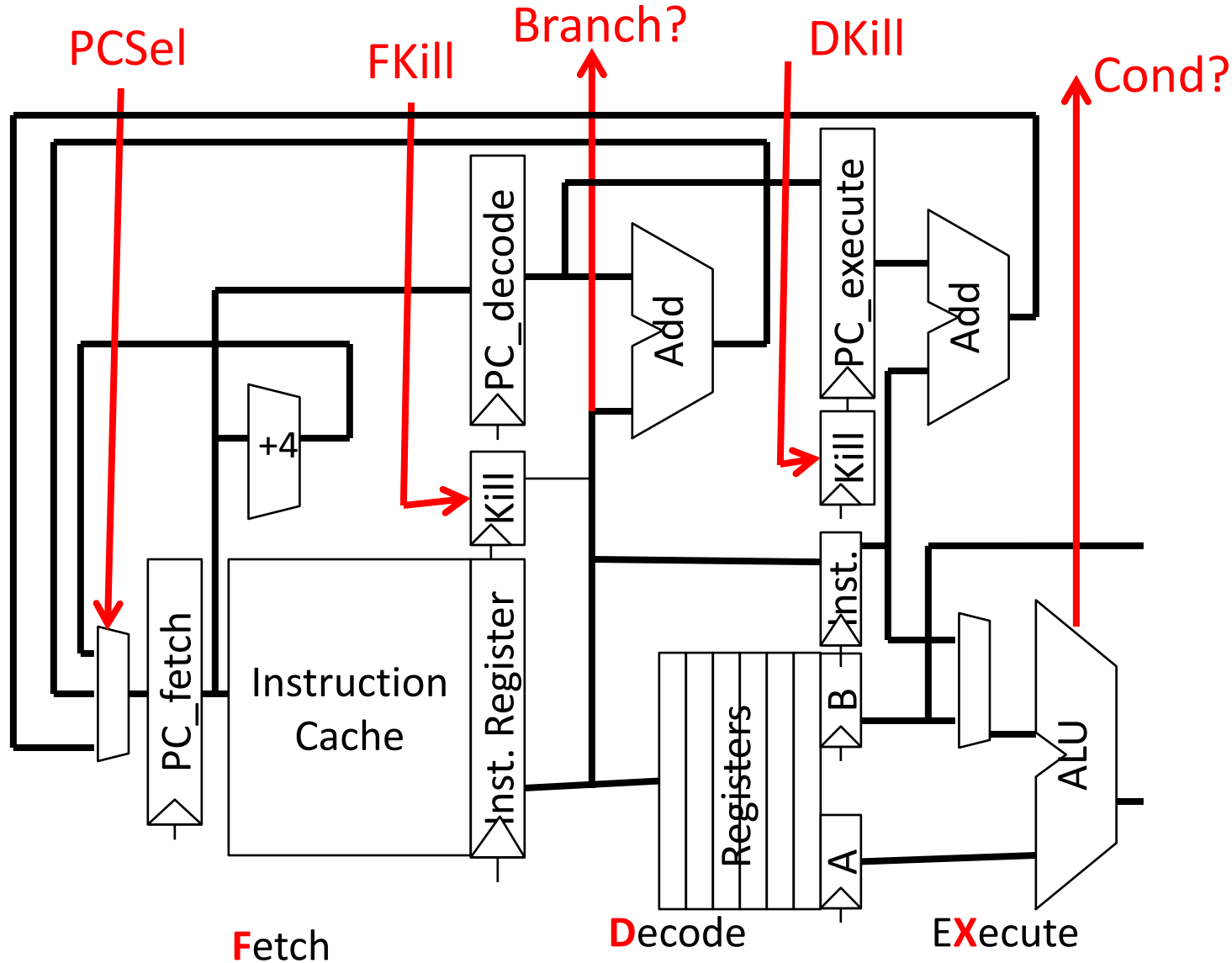
- Software has to fill delay slot with useful work, or fill with explicit NOP instruction



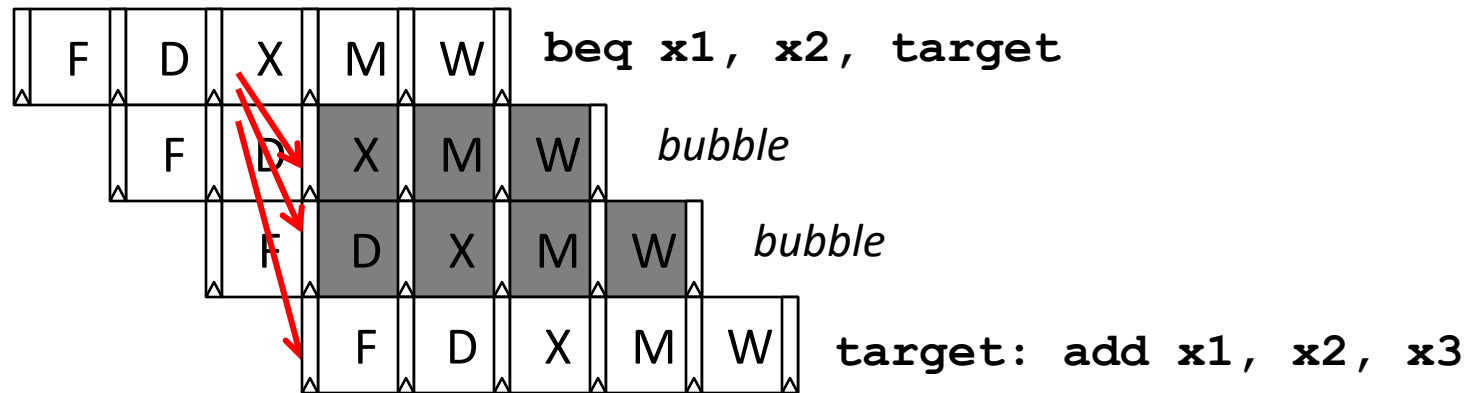
# Post-1990 RISC ISAs don't have delay slots

- 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

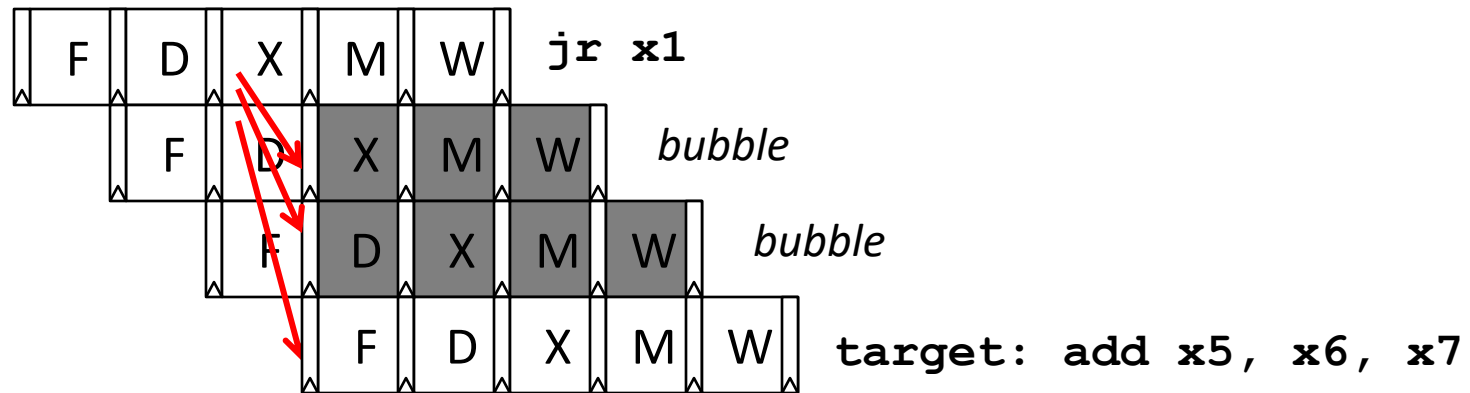


# Pipelining for Conditional Branches



# 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!*



# CS152 Administritivia

- HW1 released
  - Due Feb 02
- Lab1 released
  - Due Feb 09
- Lab group matching
  - [https://docs.google.com/forms/d/e/1FAIpQLSfxXVEwM6a-pR2-RU\\_ntjD1zfskipOWf4e-8eCljxfOhtTiaA/viewform?usp=sf\\_link](https://docs.google.com/forms/d/e/1FAIpQLSfxXVEwM6a-pR2-RU_ntjD1zfskipOWf4e-8eCljxfOhtTiaA/viewform?usp=sf_link)
- Discussions and OHs start this week.
  - Check the course calendar for details.
  - Wednesday 10am-12pm discussion dropped.
  - Discussions will be recorded.

# CS252 Administritivia

## ■ CS252 Readings on

- <https://ucb-cs252-sp23.hotcrp.com/u/0/>
- Use hotcrp to upload reviews before Wednesday:
  - Write one paragraph on main content of paper including good/bad points of paper
  - Also, answer/ask 1-3 questions about paper for discussion
  - First two “360 Architecture”, “VAX11-780”
- 2-3pm Wednesday, Soda 606/Zoom

## ■ CS252 Project Timeline

- Proposal Wed Feb 22
- One page in PDF format including:
  - project title
  - team members (2 per project)
  - what problem are you trying to solve?
  - what is your approach?
  - infrastructure to be used
  - timeline/milestones

# Traps and Interrupts

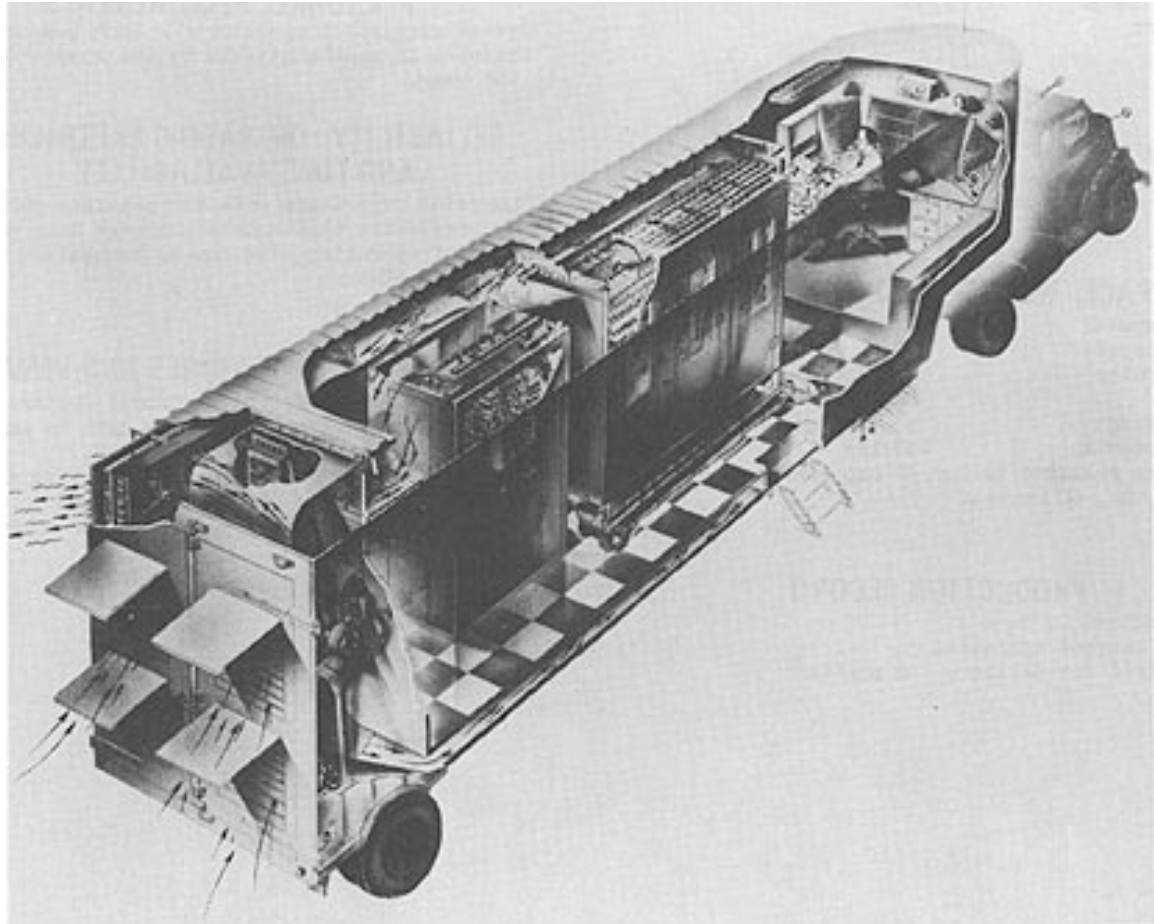
In class, we'll use following terminology

- ***Exception***: An unusual internal event caused by program during execution
  - E.g., page fault, arithmetic underflow
- ***Interrupt***: An external event outside of running program
- ***Trap***: Forced transfer of control to supervisor caused by exception or interrupt
  - Not all exceptions cause traps (c.f. IEEE 754 floating-point standard)

# 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 (**D**irect **M**emory **A**ccess by I/O device)
  - And, first mobile computer!

# DYSEAC, first mobile computer!



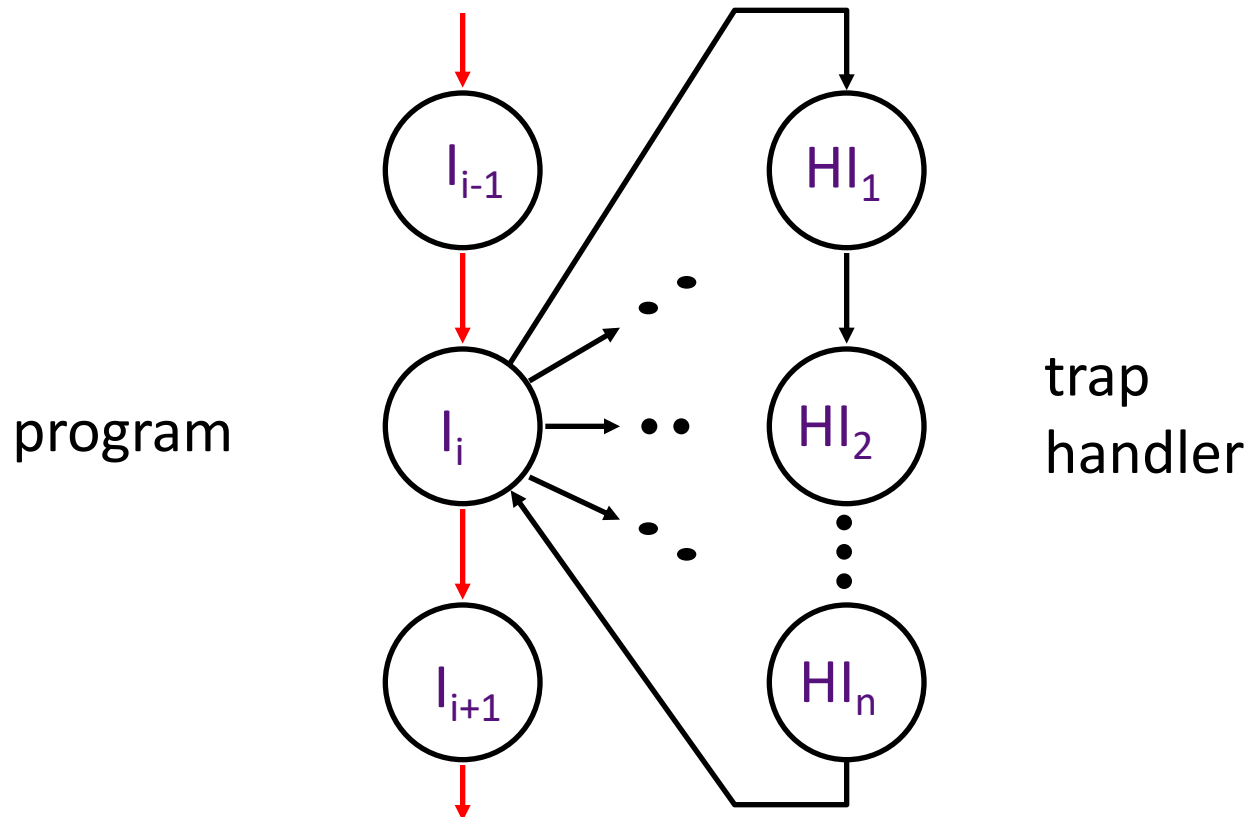
- Carried in two tractor trailers, 12 tons + 8 tons
- Built for US Army Signal Corps

*[Courtesy Mark Smotherman]*

# 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 supervisor mode

# Trap: altering the normal flow of control



An *external or internal event* that needs to be processed by another (system) program. The event is usually unexpected or rare from program's point of view.

# Trap Handler

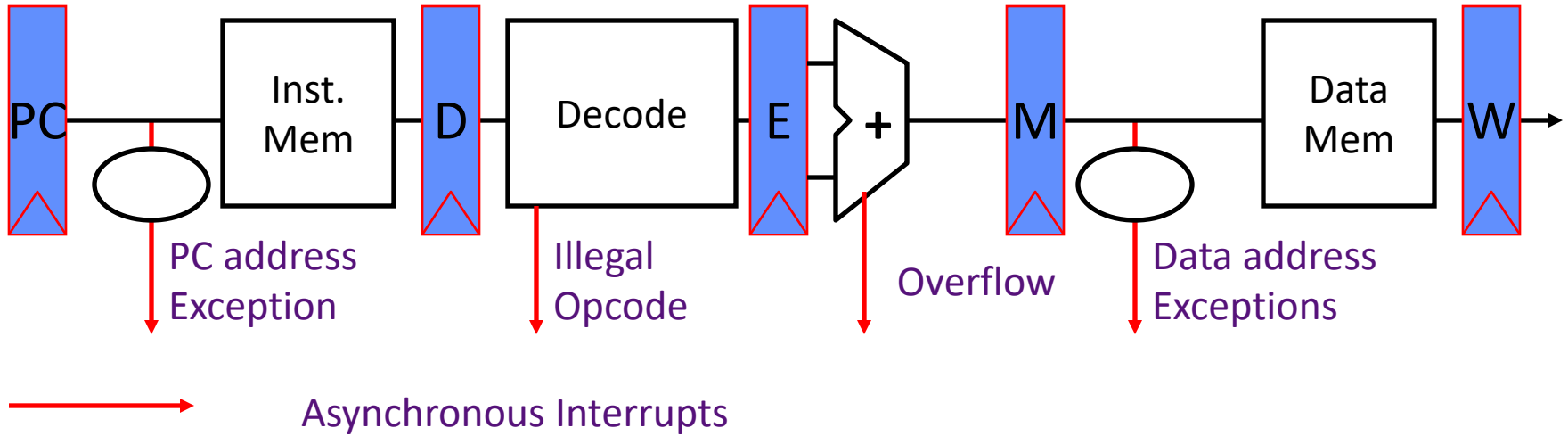
- Saves **EPC** before enabling interrupts to allow nested interrupts  $\Rightarrow$ 
  - need an instruction to move EPC into GPRs
  - need a way to mask further interrupts at least until EPC can be saved
- Needs to read a *status register* that indicates the **cause** of the trap
- Uses a special indirect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



# Synchronous Trap

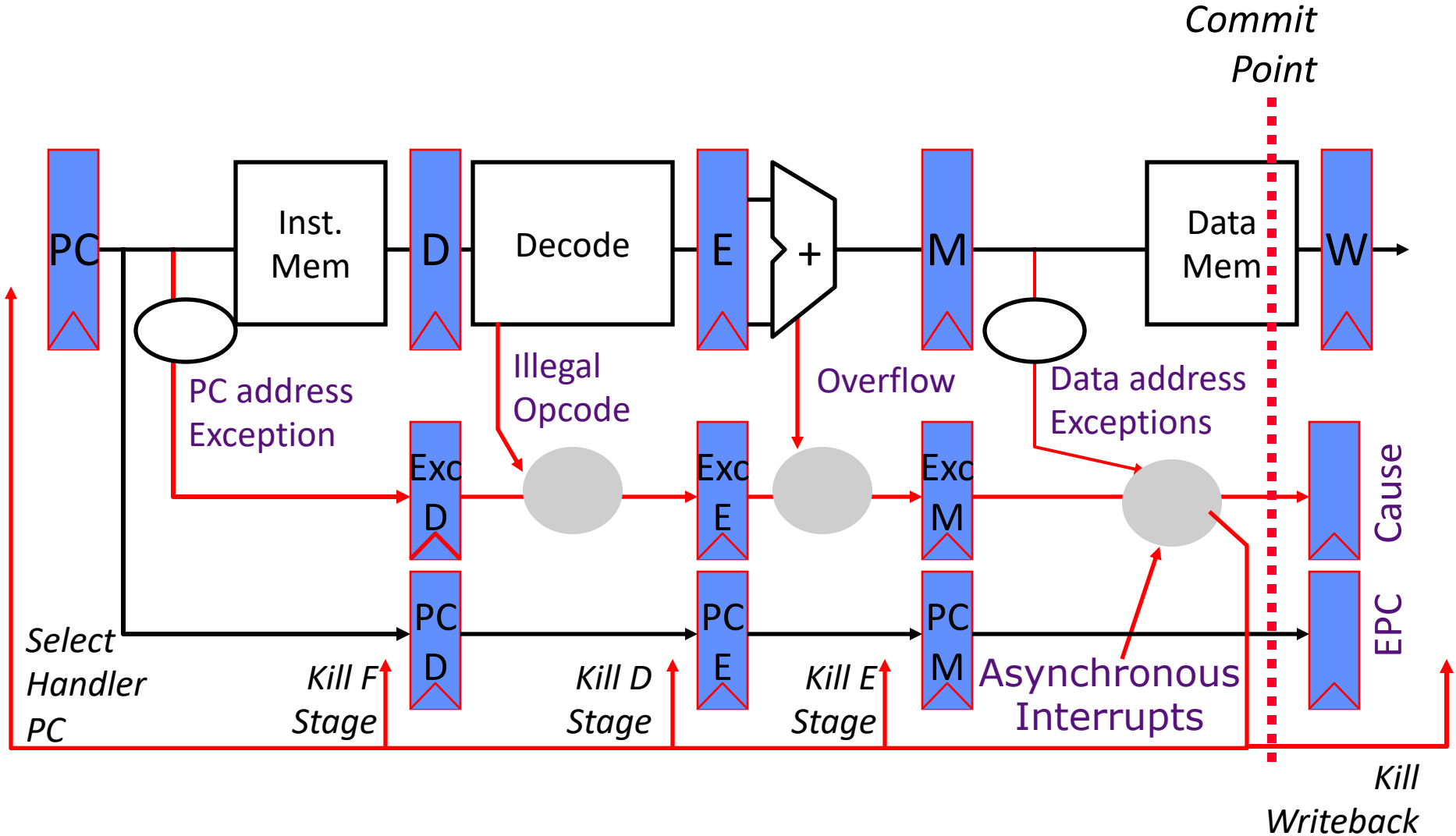
- A synchronous trap is caused by an exception on a *particular instruction*
- In general, the instruction cannot be completed and needs to be *restarted* after the exception has been handled
  - requires undoing the effect of one or more partially executed instructions
- In the case of a system call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

# 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)
  - David Patterson (UCB)
- MIT material derived from course 6.823
- UCB material derived from course CS252