# CS 152 Computer Architecture and Engineering
# CS252 Graduate Computer Architecture

## Lecture 3 - Pipelining

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

**http://www.eecs.berkeley.edu/~krste**
**http://inst.eecs.berkeley.edu/~cs152**

## 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
- Load/store, register-rich ISAs (pioneered by Cray, popularized by RISC) perform better in new VLSI technology

## Analyzing Microcoded Machines

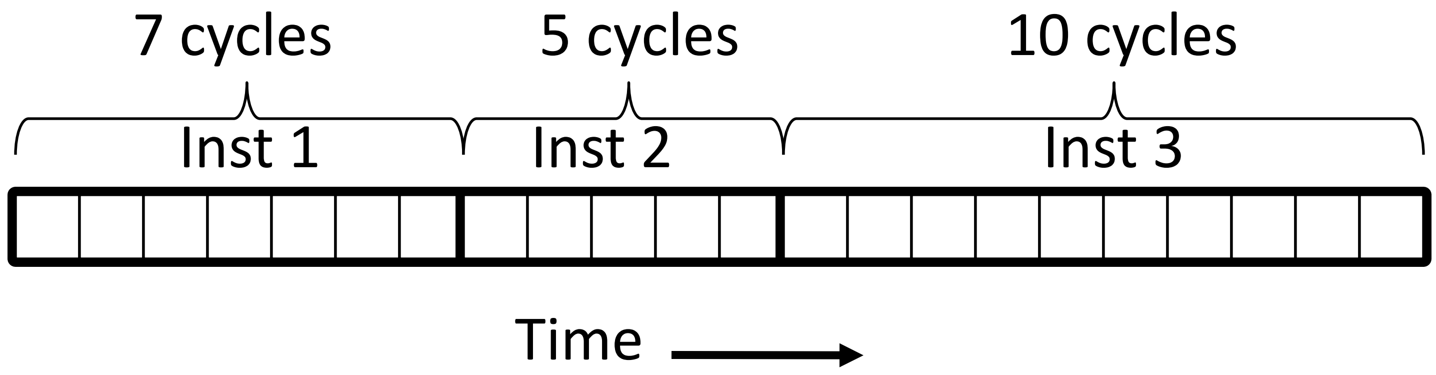
- John Cocke and group at IBM
  - Working on a simple pipelined processor, 801, and advanced compilers inside IBM
  - Ported experimental PL.8 compiler to IBM 370, and only used simple register-register and load/store instructions similar to 801
  - Code ran faster than other existing compilers that used all 370 instructions! (up to 6MIPS whereas 2MIPS considered good before)
- Emer, Clark, at DEC
  - Measured VAX-11/780 using external hardware
  - Found it was actually a 0.5MIPS machine, although usually assumed to be a 1MIPS machine
  - Found 20% of VAX instructions responsible for 60% of microcode, but only account for 0.2% of execution time!
- VAX8800
  - Control Store: 16K*147b RAM, Unified Cache: 64K*8b RAM
  - 4.5x more microstore RAM than cache RAM!

## “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

## CPI for Microcoded Machine



Total clock cycles = 7+5+10 = 22

Total instructions = 3

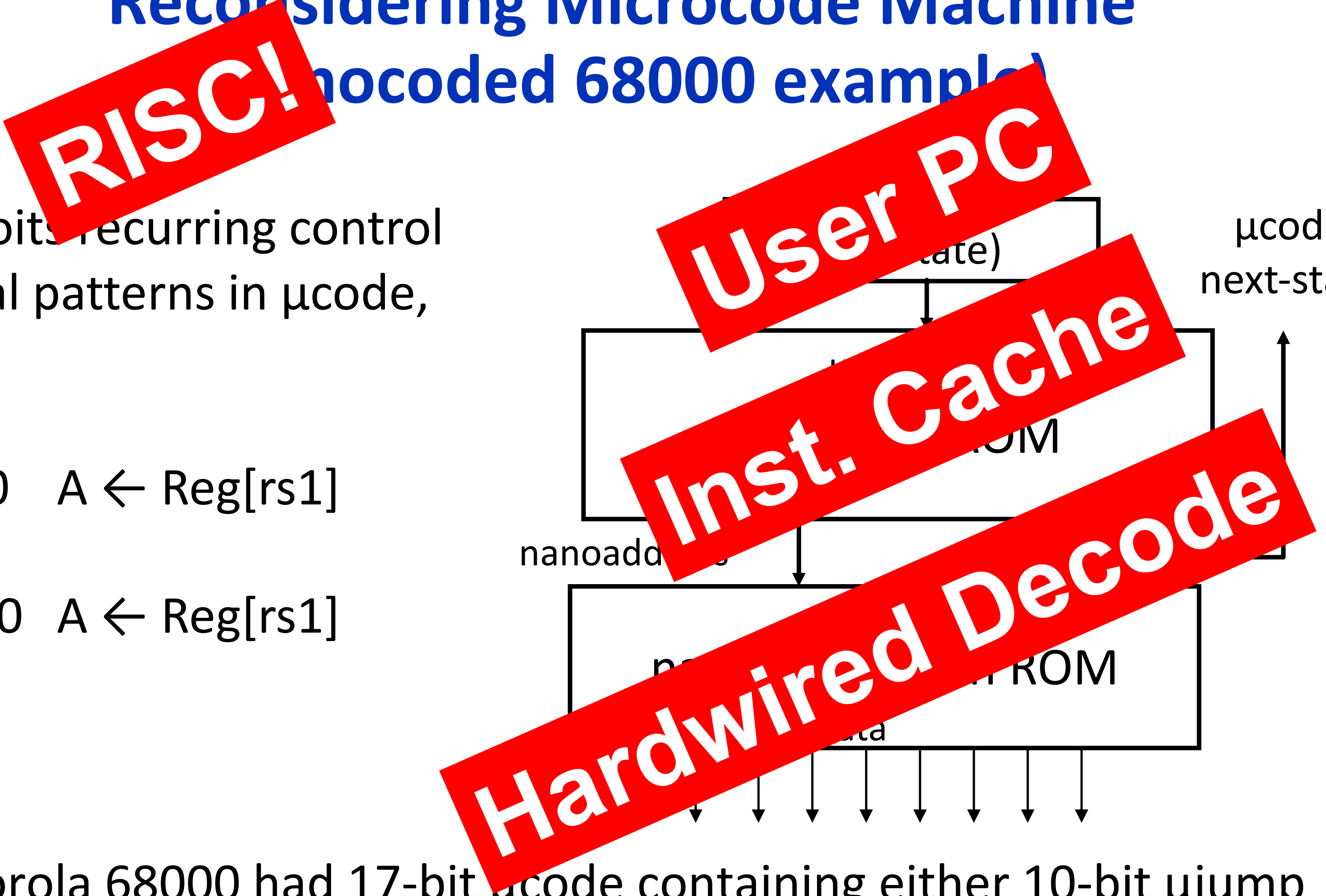
CPI = 22/3 = 7.33

CPI is always an average over a large number of instructions.

## IC Technology Changes Tradeoffs

- Logic, RAM, ROM all implemented using MOS transistors
- Semiconductor RAM ~ same speed as ROM

# Reconsidering Microcode Machine (Nanocoded 68000 example)



Exploits recurring control signal patterns in µcode, e.g.,

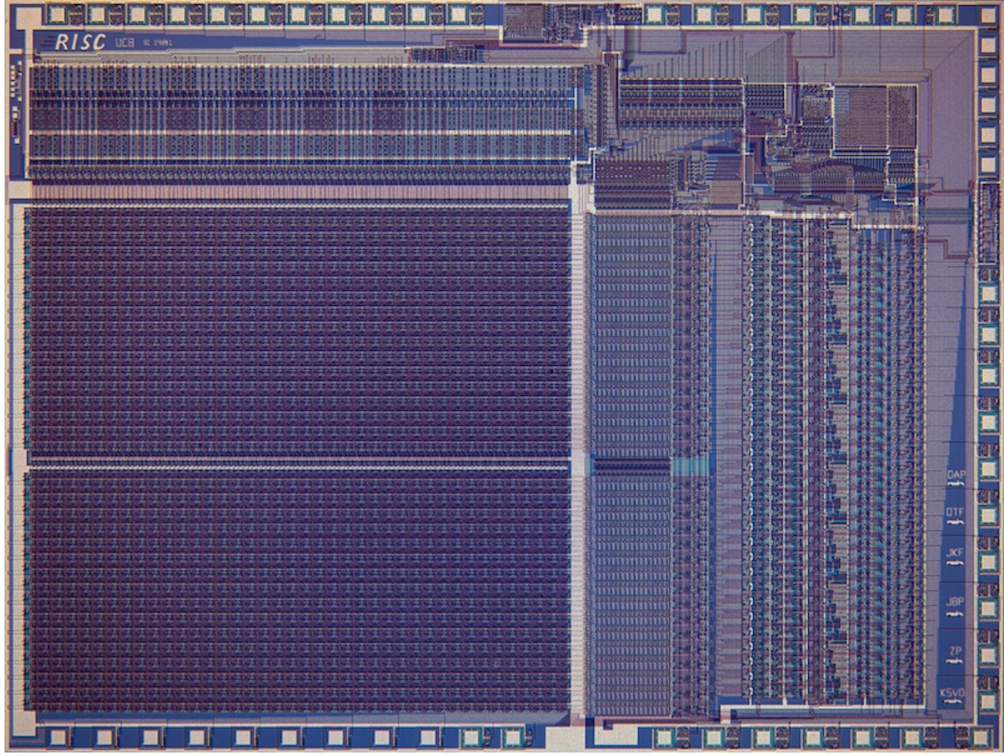
ALU0  A ← Reg[rs1]
...
ALUI0  A ← Reg[rs1]
...

- Motorola 68000 had 17-bit µcode containing either 10-bit µjump or 9-bit nanoinstruction pointer
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals

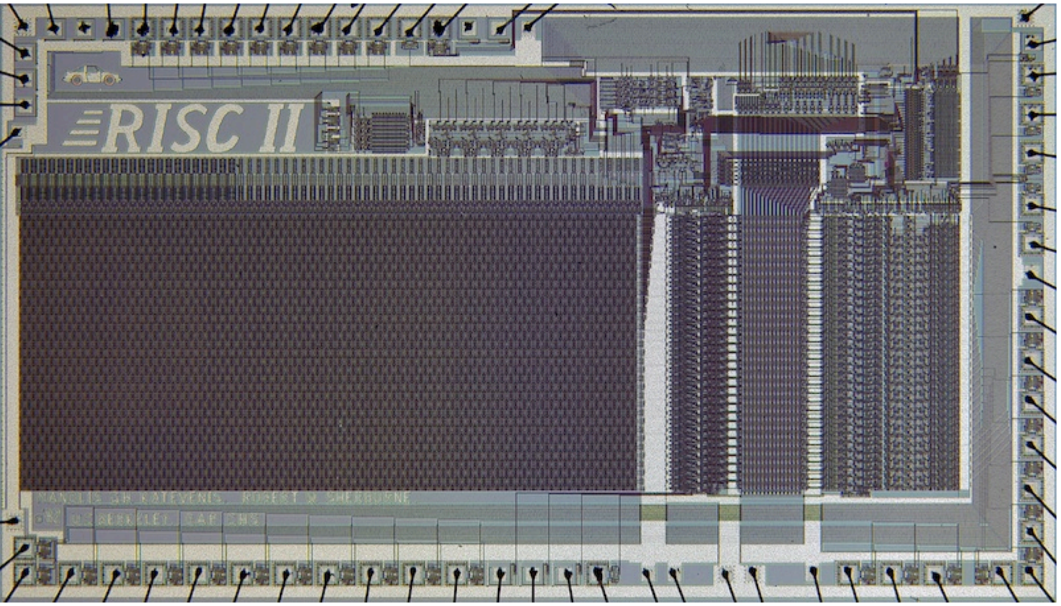
## From CISC to RISC

- Use fast RAM to build fast instruction *cache* of user-visible instructions, not fixed hardware microroutines
  - Contents of fast instruction memory change to fit application needs
- Use simple ISA to enable hardwired pipelined implementation
  - Most compiled code only used few CISC instructions
  - Simpler encoding allowed pipelined implementations
- Further benefit with integration
  - In early ‘80s, finally fit 32-bit datapath + small caches on single chip
  - No chip crossings in common case allows faster operation

# Berkeley RISC Chips



RISC-I (1982) Contains 44,420 transistors, fabbed in 5 µm NMOS, with a die area of 77 mm$^2$, ran at 1 MHz. This chip is probably the first VLSI RISC.



RISC-II (1983) contains 40,760 transistors, was fabbed in 3 µm NMOS, ran at 3 MHz, and the size is 60 mm$^2$.

Stanford built some too…

## Microprogramming is far from extinct

- Played a crucial role in micros of the Eighties
    - DEC uVAX, Motorola 68K series, Intel 286/386
- Plays an assisting role in most modern micros
    - e.g., AMD Zen, Intel Sky Lake, Intel Atom, IBM PowerPC, …
    - Most instructions executed directly, i.e., with hard-wired control
    - Infrequently-used and/or complicated instructions invoke microcode
- Patchable microcode common for post-fabrication bug fixes, e.g. Intel processors load µcode patches at bootup
    - Intel had to scramble to resurrect microcode tools and find original microcode engineers to patch Meltdown/Spectre security vulnerabilites

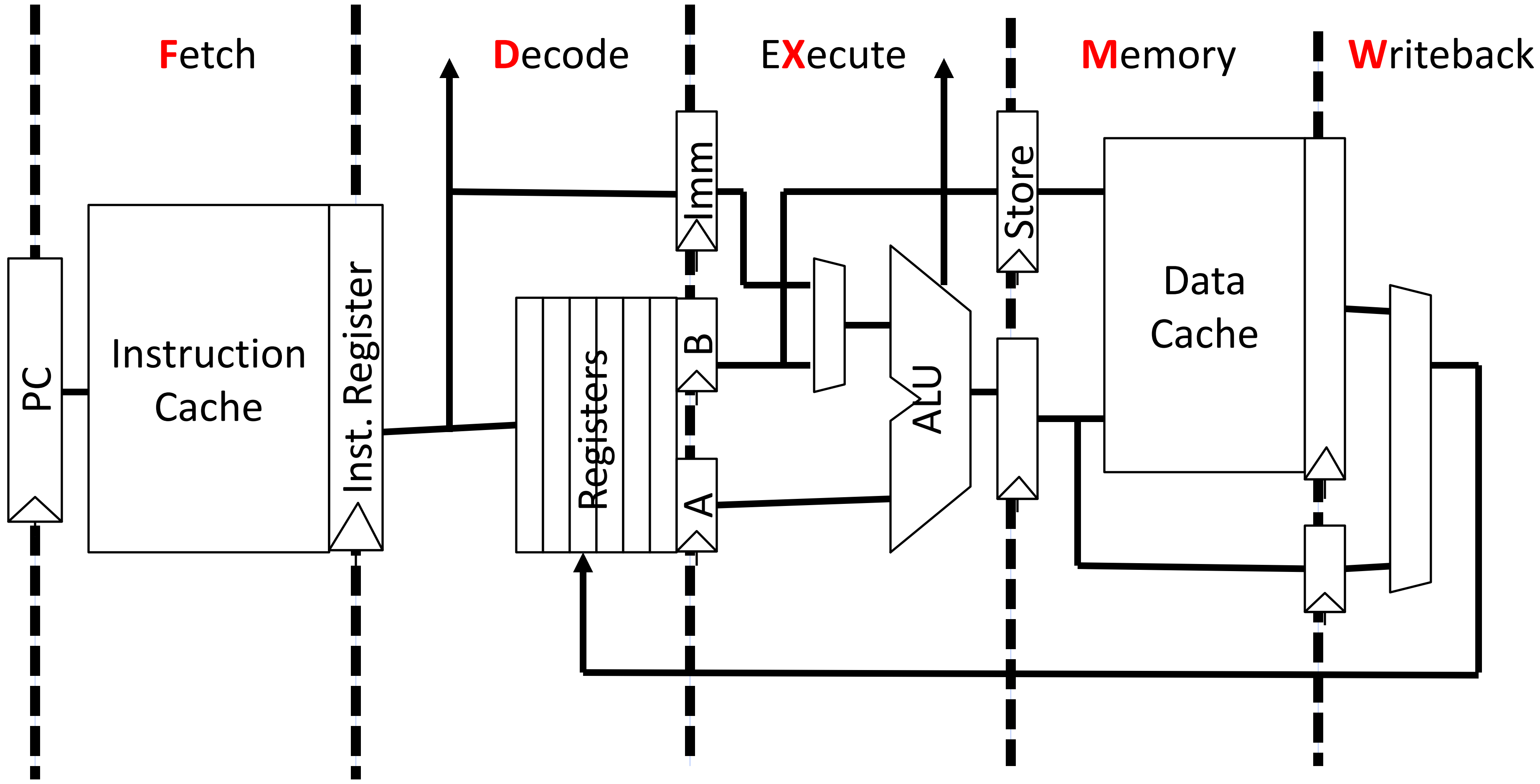
## "Iron Law" of Processor Performance

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

- 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 |

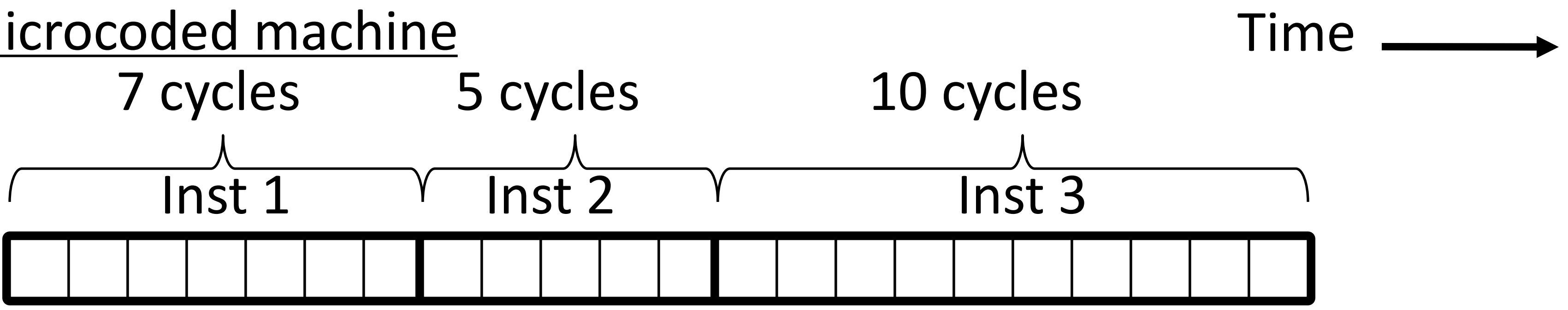
# Classic 5-Stage RISC Pipeline



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

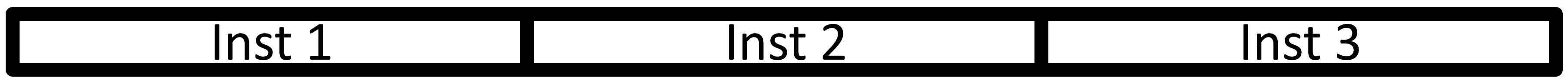
# CPI Examples

Microcoded machine



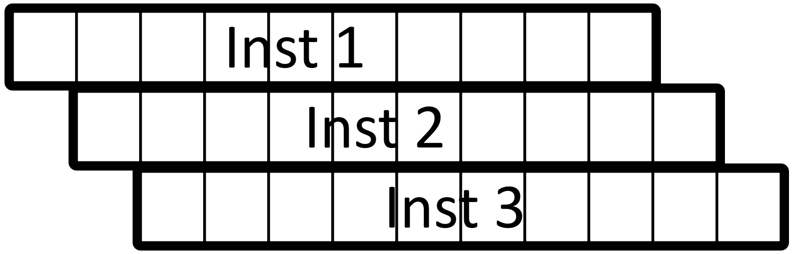
3 instructions, 22 cycles, CPI=7.33

Unpipelined machine



3 instructions, 3 cycles, CPI=1

Pipelined machine



3 instructions, 3 cycles, CPI=1

**5-stage pipeline CPI≠5!!!**

## Instructions interact with each other in pipeline

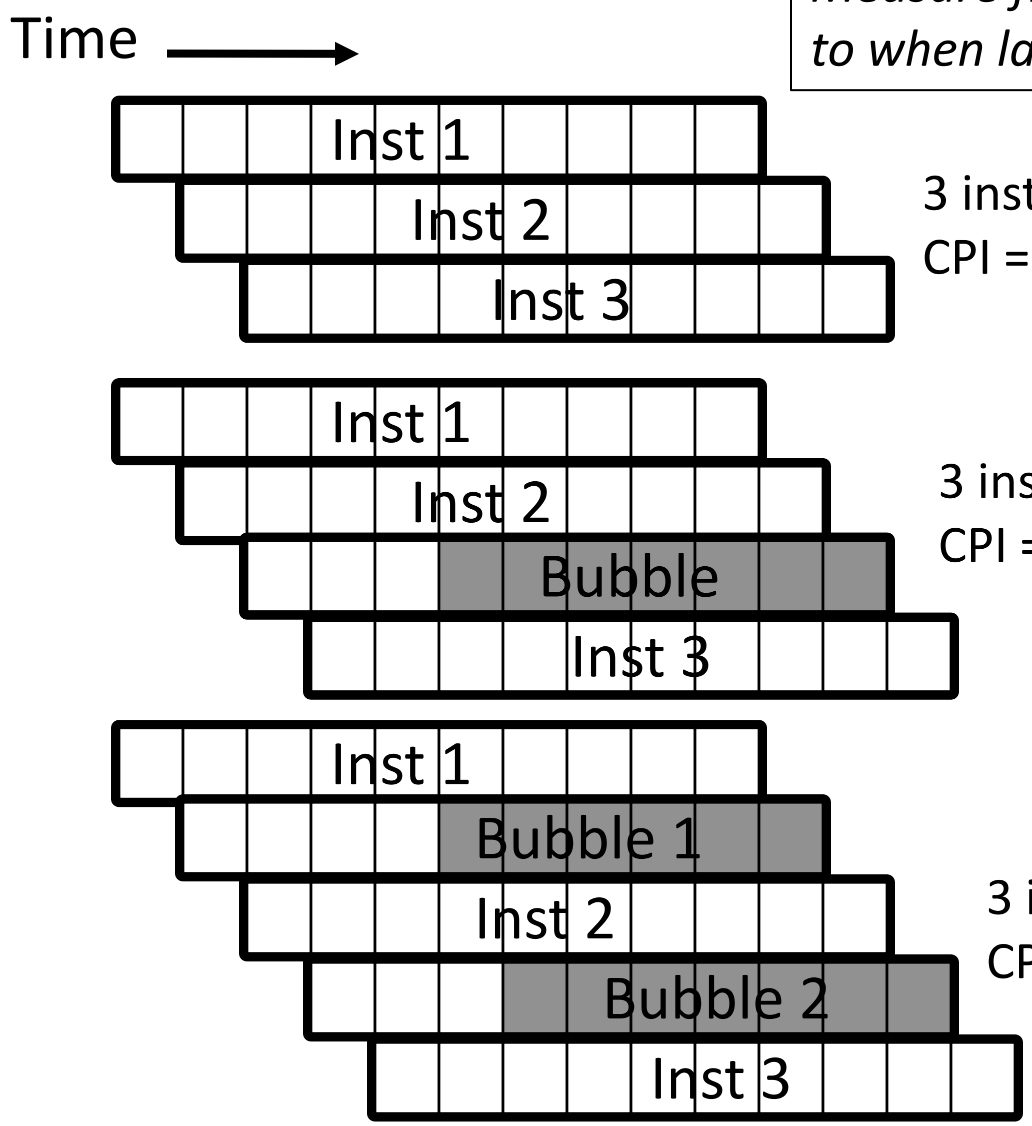
- An instruction in the pipeline may need a resource being used by another instruction in the pipeline → *structural hazard*

- 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

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

Read-after-Write (RAW) hazard

Anti-dependence

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

Write-after-Read (WAR) hazard

Output-dependence

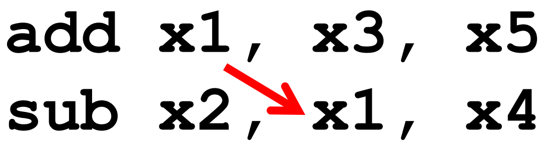
$$\begin{tikzcd}[row sep=small, column sep=tiny]
r_3 \arrow[d, bend right] & \leftarrow & r_1 & \text{op} & r_2 \\
r_3 & \leftarrow & r_6 & \text{op} & r_7
\end{tikzcd}$$

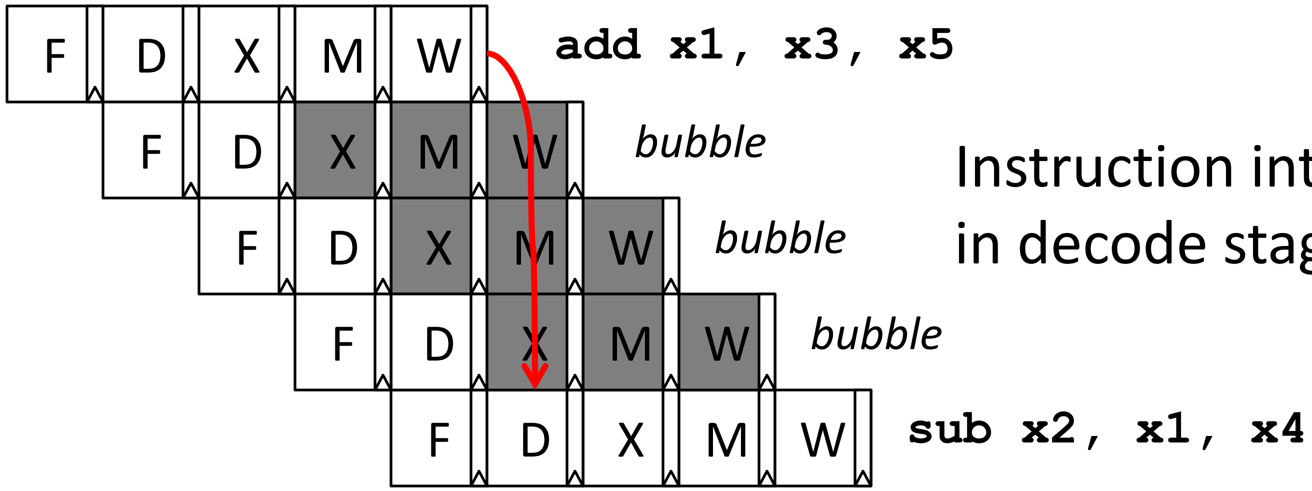
Write-after-Write (WAW) hazard

## Three Strategies for Data Hazards

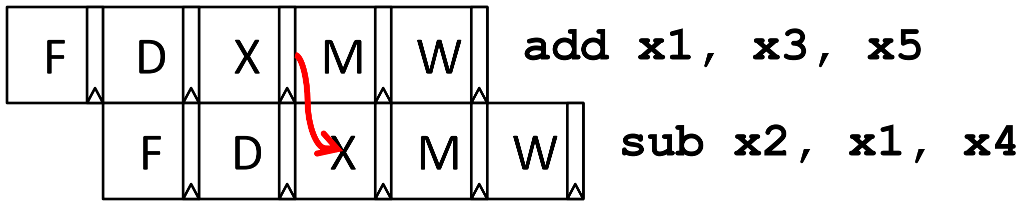
- Interlock
  - Wait for hazard to clear by holding dependent instruction in issue stage
- Bypass
  - Resolve hazard earlier by bypassing value as soon as available
- Speculate
  - Guess on value, correct if wrong

# Interlocking Versus Bypassing



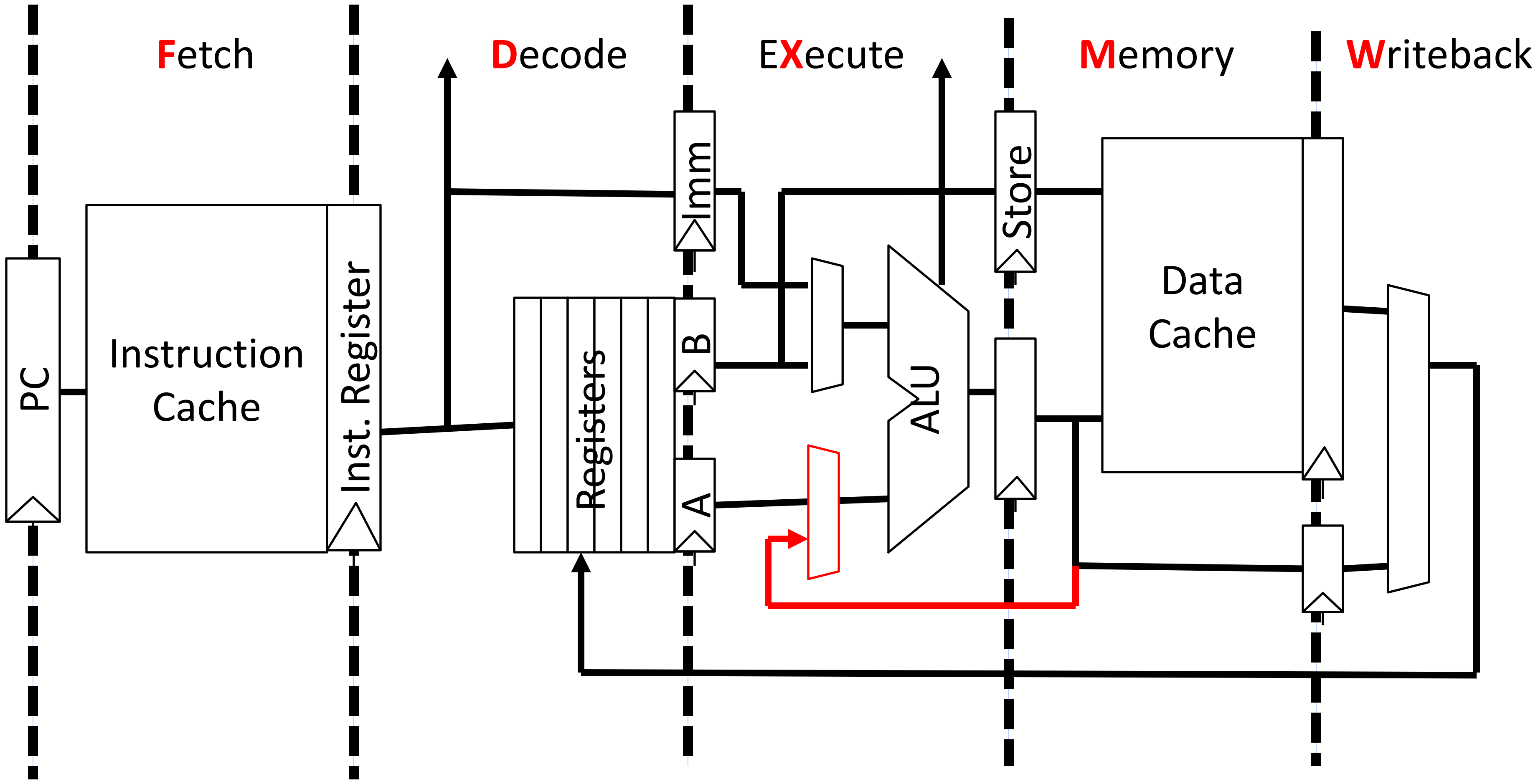


Instruction interlocked in decode stage

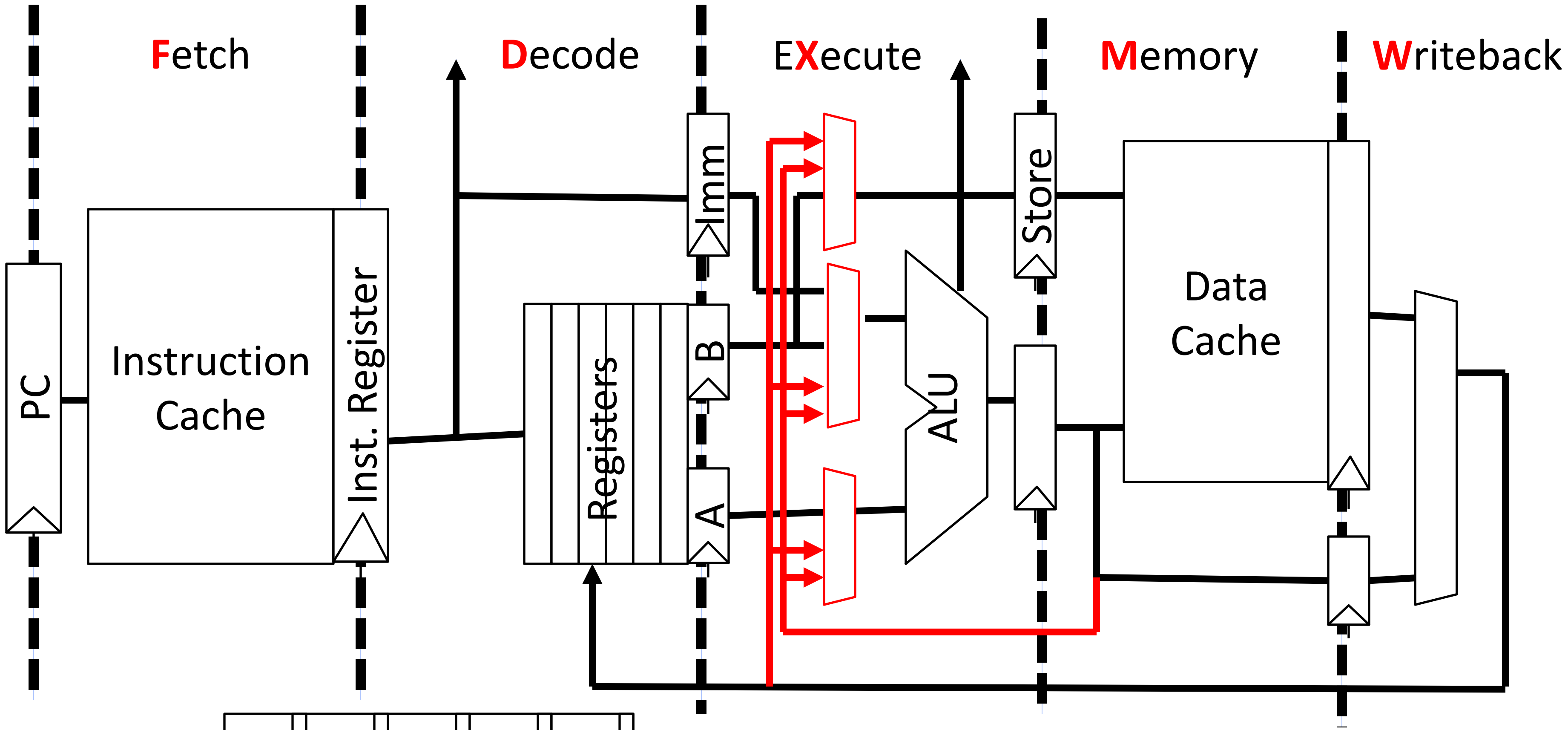


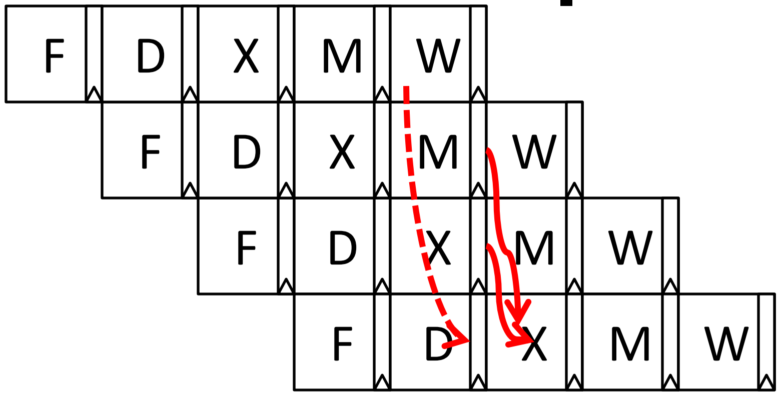
Bypass around ALU with no bubbles

## 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

## CS152 Administrivia

- PS 1 is posted
- PS 1 is due at start of class on Monday Feb 11

- Lab 1 out on Friday
- Lab 1 overview in Section Friday,
  - 1-2pm DIS 101 3113 Etcheverry
  - 2-3pm DIS 102 3107 Etcheverry

## CS252 Administrivia

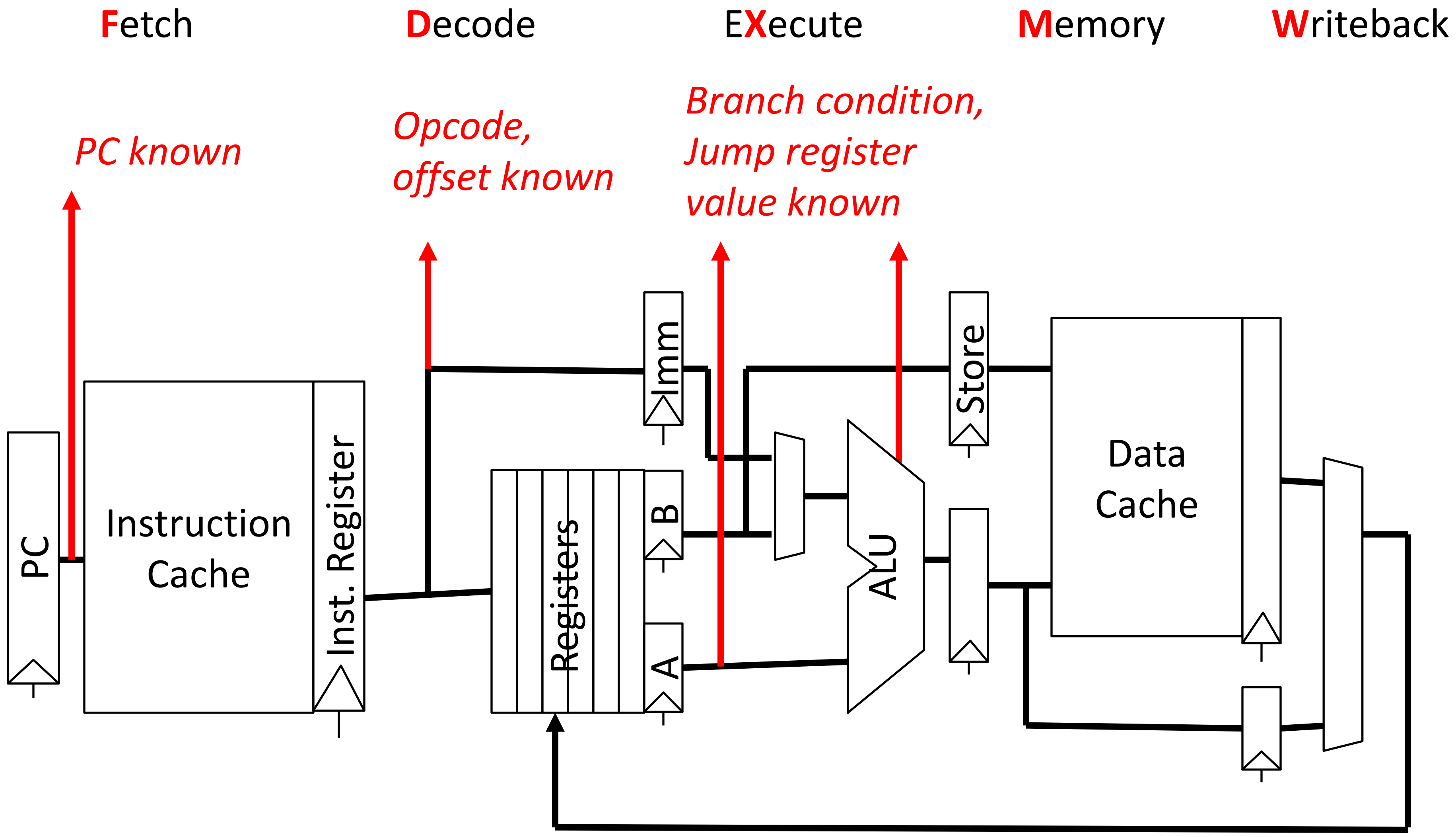
- CS252 discussions grading policy
  - We'll ignore your two lowest scores in grading, which includes absences
  - Send in summary even if you can't attend discussion
- CS252 Piazza class has been created
  - Sign up for this as well as CS152 Piazza
- Each CS252 paper has dedicated thread
  - Post your response as private note to instructors
  - Due 6AM Monday before Monday discussion section

## Control Hazards

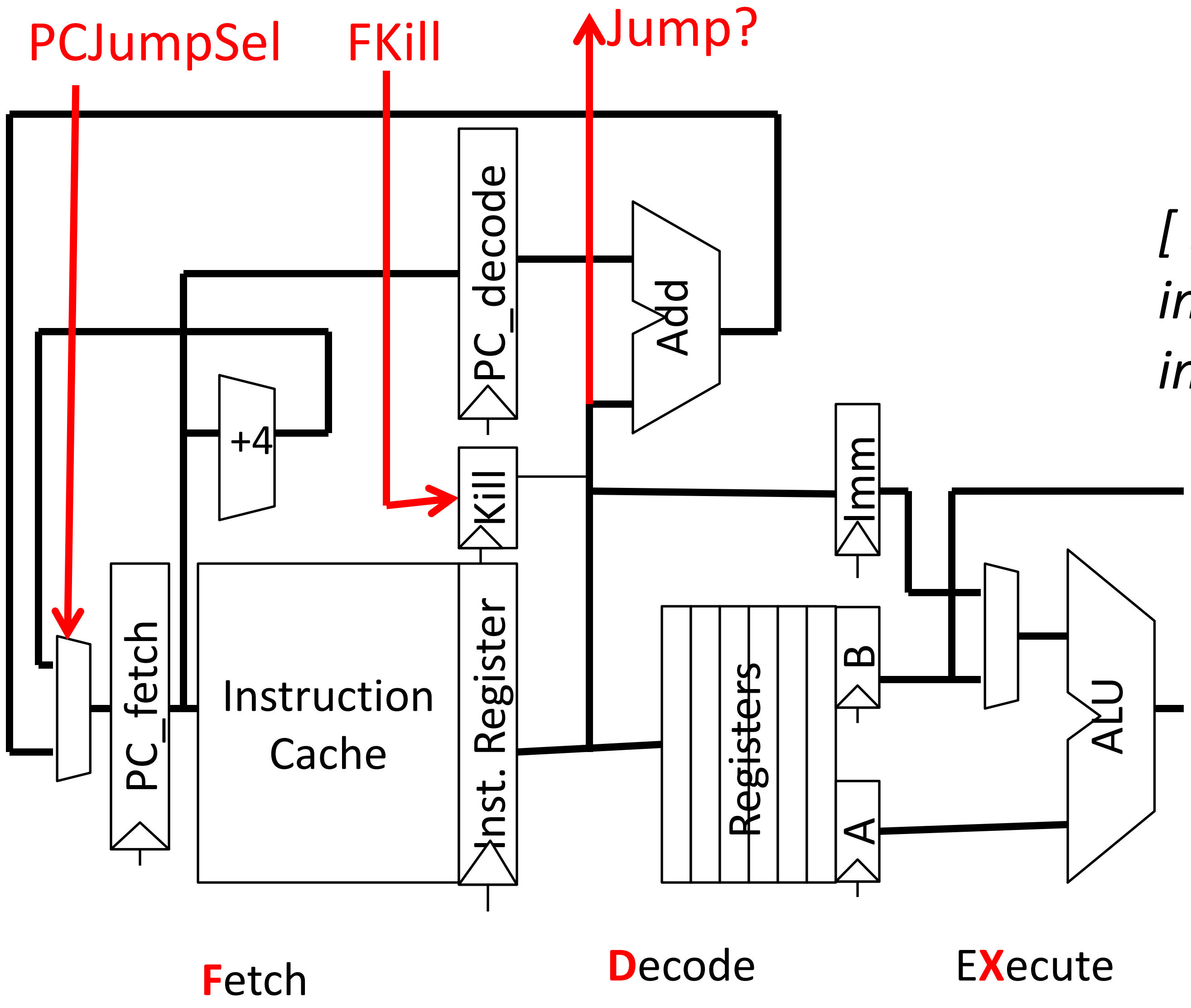
What do we need to calculate next PC?

- 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

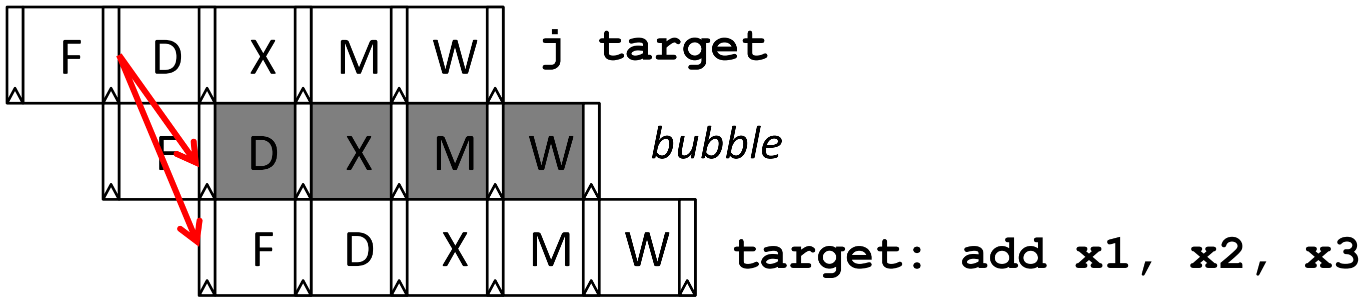


# RISC-V Unconditional PC-Relative Jumps



*[ Kill bit turns instruction into a bubble ]*

# Pipelining for Unconditional PC-Relative Jumps

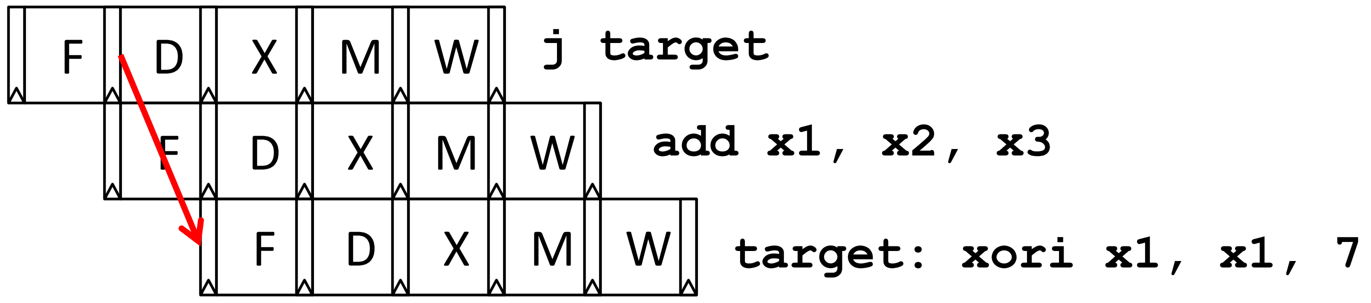


# Branch Delay Slots

- 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
```

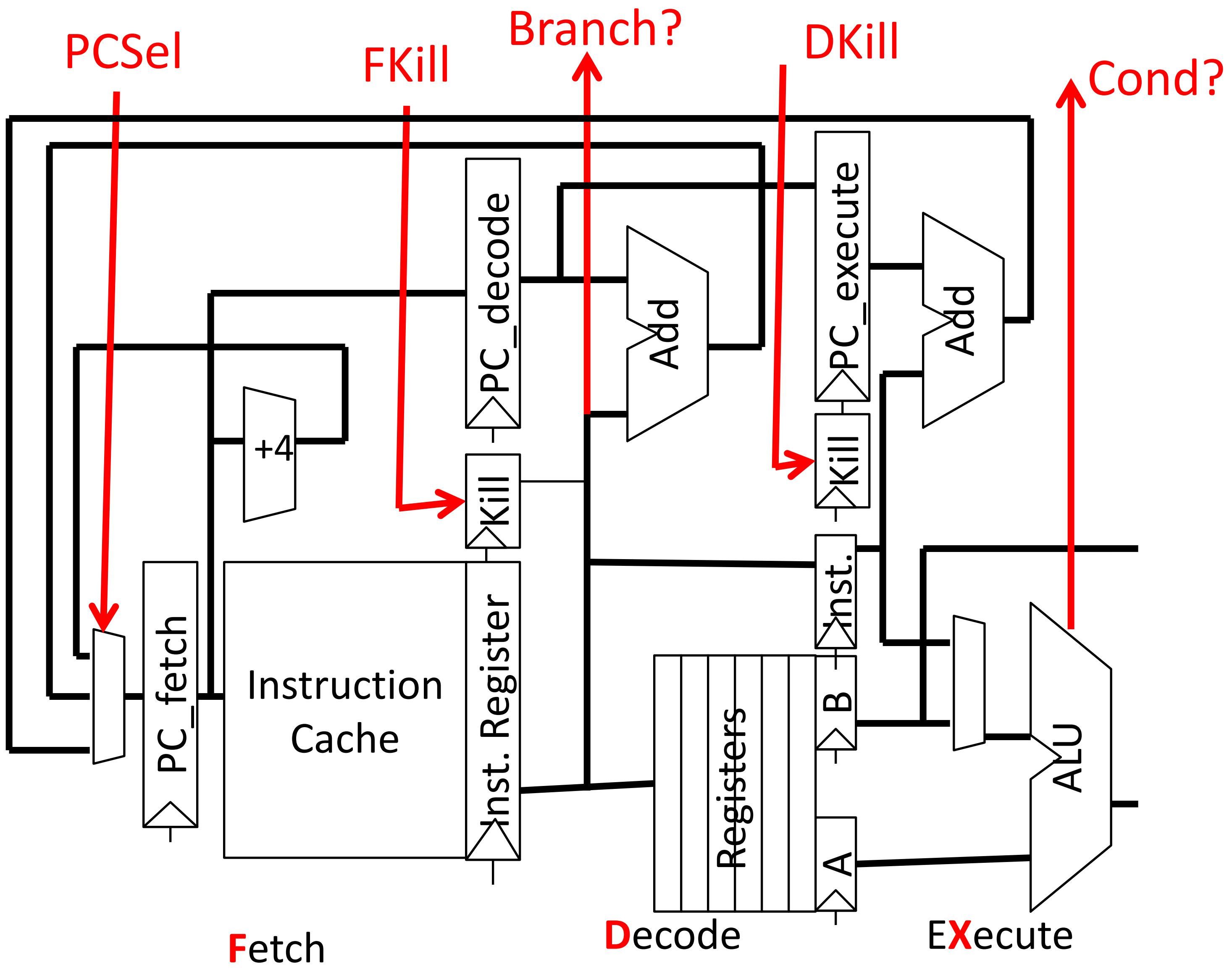
- Software has to fill delay slot with useful work, or fill with explicit NOP instruction



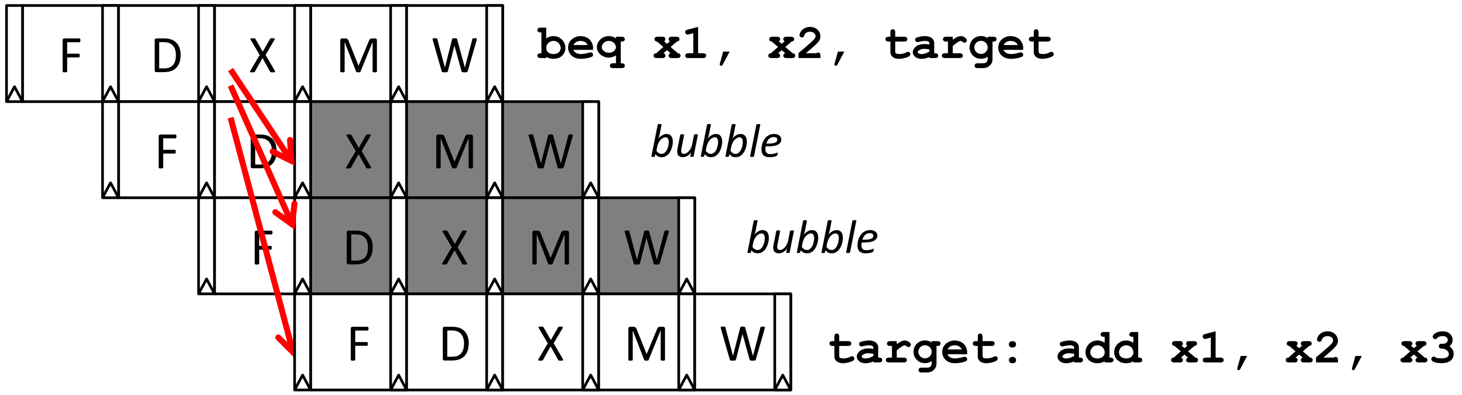
## Post-1990 RISC ISAs don’t have delay slots

- 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

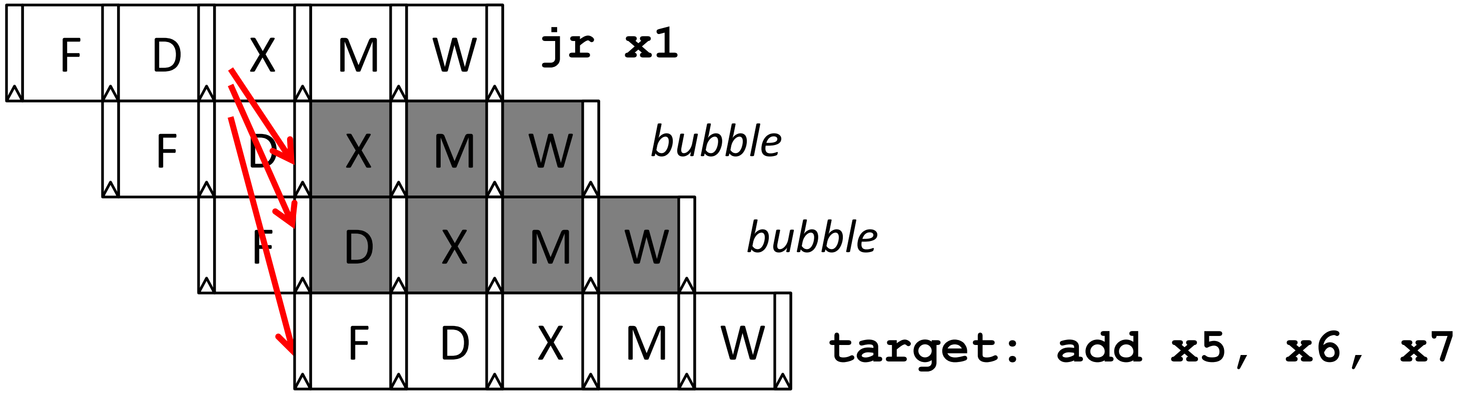


## Pipelining for Conditional Branches



# 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

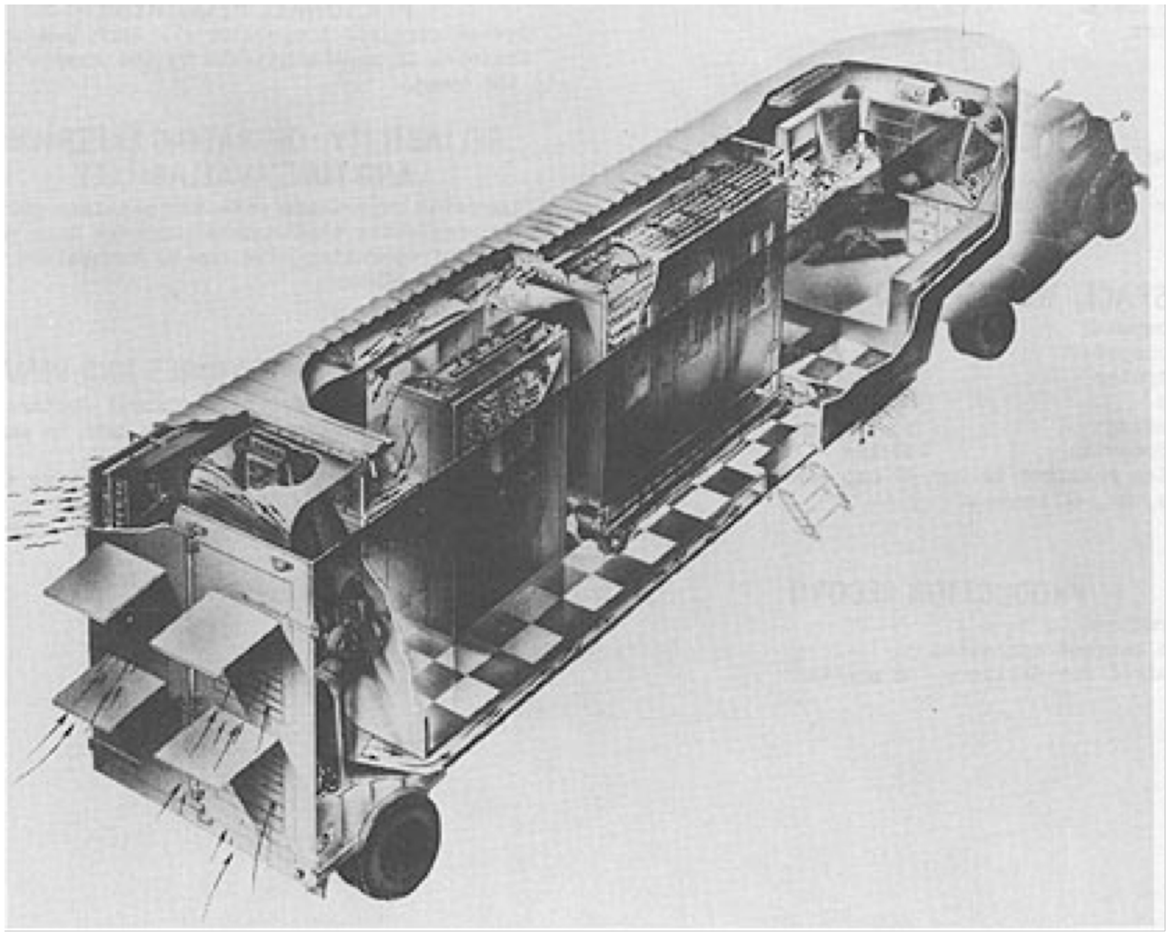
In class, we'll use following terminology

- ***Exception***: An unusual internal event caused by program during execution
  - E.g., page fault, arithmetic underflow
- ***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!



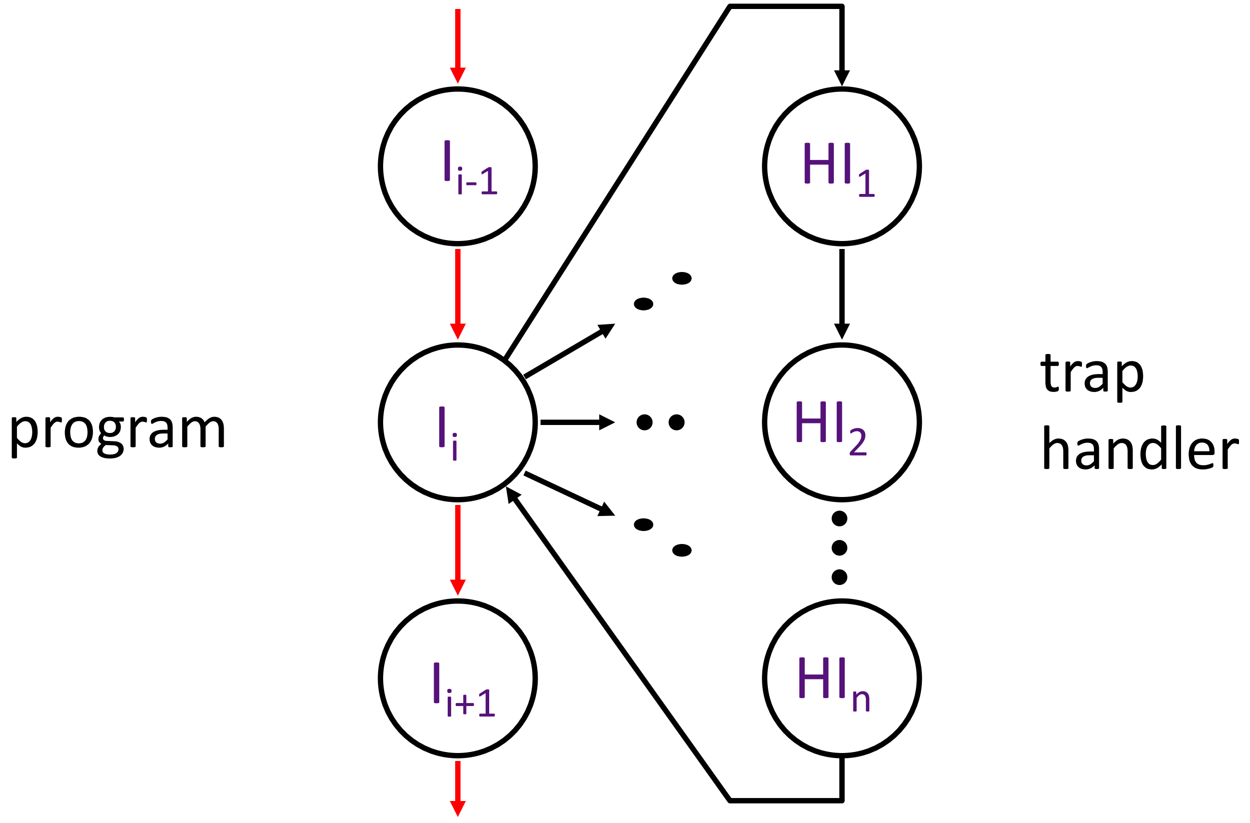
- Carried in two tractor trailers, 12 tons + 8 tons
- Built for US Army Signal Corps

*[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.

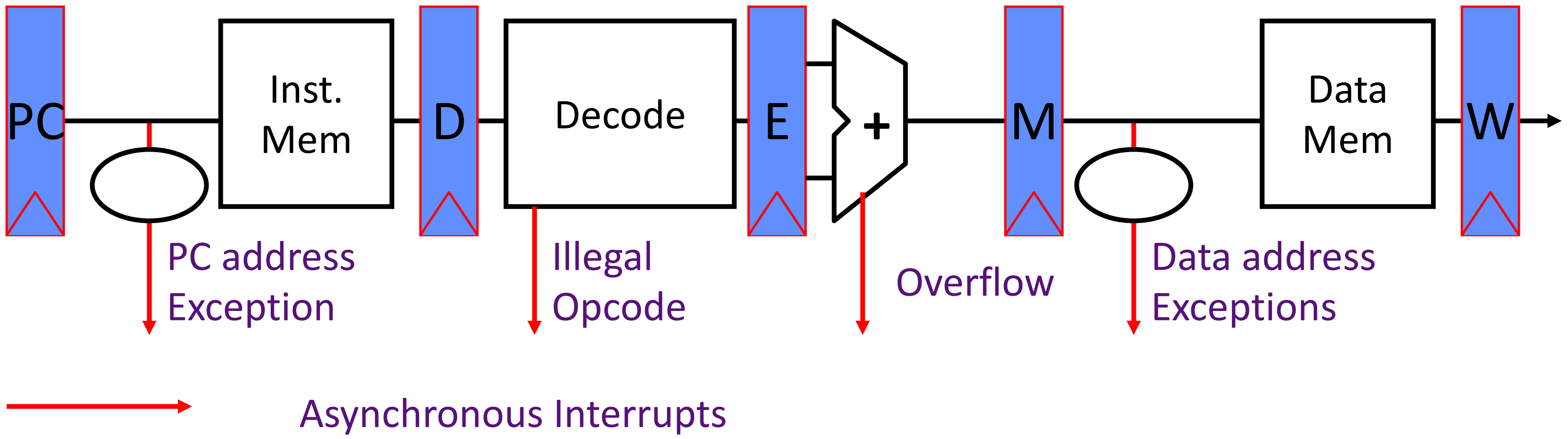
## Trap Handler

- 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

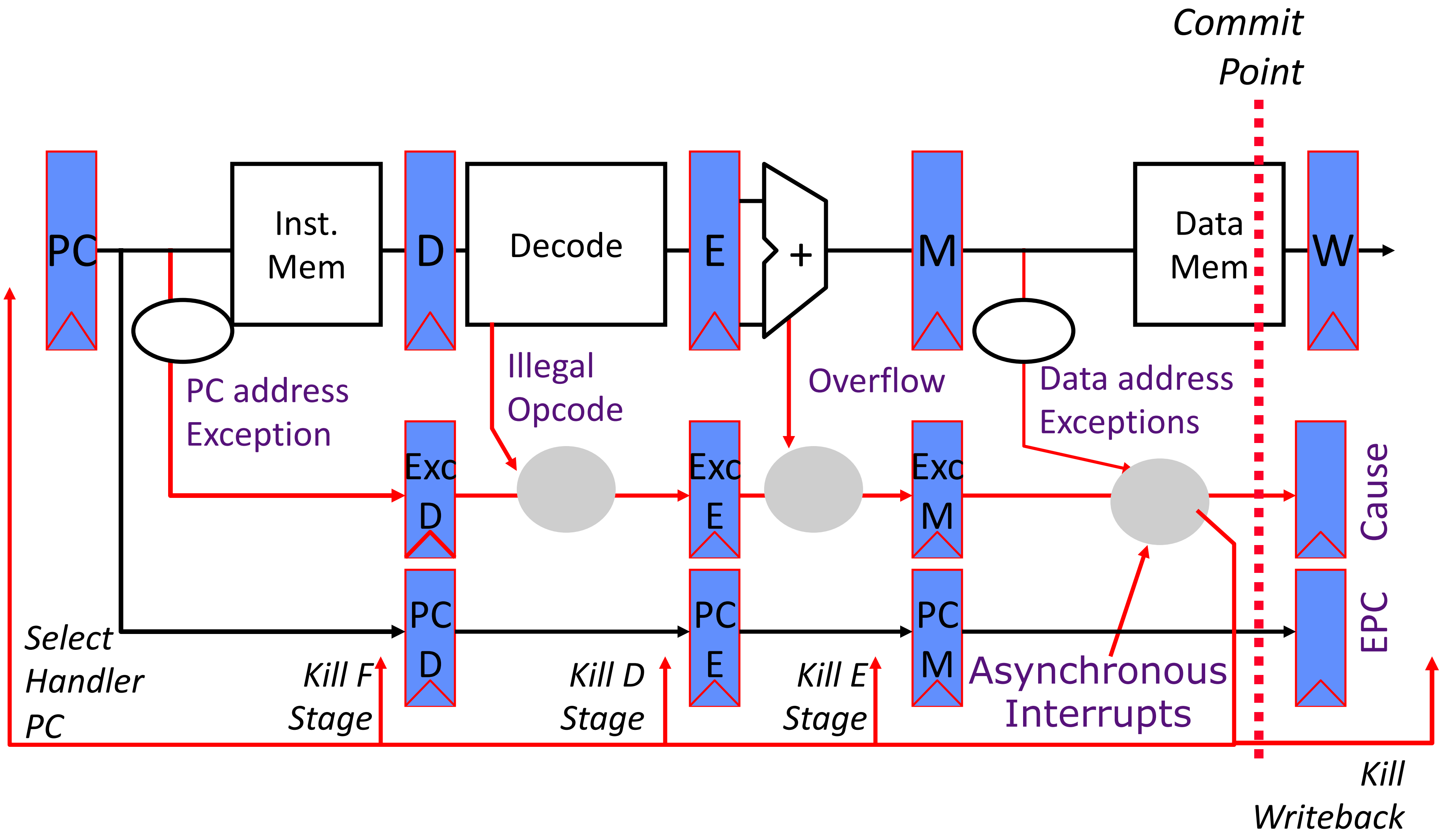
- A synchronous trap is caused by an exception on a *particular instruction*
- In general, the instruction cannot be completed and needs to be *restarted* after the exception has been handled
  - requires undoing the effect of one or more partially executed instructions
- In the case of a system call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

- 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)