

Microprogramming thrived in '60s and '70s

- Significantly faster ROMs than DRAMs were available
- For complex instruction sets, datapath and controller were cheaper and simpler
- New instructions , e.g., floating point, could be supported without datapath modifications
- Fixing bugs in the controller was easier
- ISA compatibility across various models could be achieved easily and cheaply

Except for the cheapest and fastest machines, all computers were microprogrammed

Microprogramming: early 1980s

- Evolution bred more complex micro-machines
 - Complex instruction sets led to need for subroutine and call stacks in μcode
 - Need for fixing bugs in control programs was in conflict with read-only nature of μROM
 - → Writable Control Store (WCS) (B1700, QMachine, Intel i432, ...)
- With the advent of VLSI technology assumptions about ROM & RAM speed became invalid → more complexity
- Better compilers made complex instructions less important.
- Use of numerous micro-architectural innovations, e.g., pipelining, caches and buffers, made multiple-cycle execution of reg-reg instructions unattractive

VAX 11-780 Microcode

; P1WFUD,1 [600,1205]
; CALL2 .MIC [600,1205]

MICRO2 1F(12)
Procedure call

26-May-81 14:58:1
: CALLG, CALLS

VAX11/780 Microcode : PCS 01, FPLA 0D, WCS122

Page 771

```

;29744 ;HERE FOR CALLG OR CALLS, AFTER PROBING THE EXTENT OF THE STACK
;29745
;29746 =0 ;-----;CALL SITE FOR MPUSH
;29747 CALL.7: D_Q,AND,RC[T2], ;STRIP MASK TO BITS 11-0
;29748 CALL,J/MPUSH ;PUSH REGISTERS
;-----;RETURN FROM MPUSH
;29750 CACHE_D[LONG], ;PUSH PC
;29751 LAB_R[SP] ; BY SP
;-----;UPDATE SP FOR PUSH OF PC &
;29752 CACHE_D[LONG], ;PUSH PC
;29753 LAB_R[SP] ; BY SP
;-----;READY TO PUSH FRAME POINTER
;29754 CALL.8: R[SP]&VA_LA-K[,8]
;29755 ;-----;UPDATE SP FOR PUSH OF PC &
;29756 D_R[FPP]
;29757 ;-----;READY TO PUSH FRAME POINTER
;29758 =0 ;-----;CALL SITE FOR PSHSP
;29759 CACHE_D[LONG], ;STORE FP,
;29760 LAB_R[SP], ; GET SP AGAIN
;29761 SC_K[.FFF0], ;16 TO SC
;29762 CALL,J/PSHSP
;29763 ;-----;READY TO PUSH AP
;29764 D_R[AP], ;READY TO PUSH AP
;29765 Q_ID[PSL] ; AND GET PSW FOR COMBINATIO
;29766 ;-----;STORE OLD AP
;29767 CACHE_D[LONG], ;CLEAR PSW<T,N,Z,V,C>
;29768 Q_Q,ANDNOT,K[.1F], ;GET SP INTO LATCHES AGAIN
;29769 LAB_R[SP]
;-----;LOAD NEW PC AND CLEAR OUT
;29770 PC&VA_RC[T1], FLUSH,IB
;29771 ;-----;PSW TO D<31:16>
;29772 D_DAL,SC, ;RECOVER MASK
;29773 Q_RC[T2], ;PUT -13 IN SC
;29774 SC_SC+K[.3], ;START FETCHING SUBROUTINE I
;29775 LOAD,IB, PC_PC+1
;-----;MASK AND PSW IN D<31:03>
;29776 D_DAL,SC, ;GET LOW BITS OF OLD SP TO Q<1:0>
;29777 Q_RC[T4], ;PUT -3 IN SC
;29778 SC_SC+K[.A]
;-----;
```

Writable Control Store (WCS)

- Implement control store in RAM not ROM
 - MOS SRAM memories now almost as fast as control store (core memories/DRAMs were 2-10x slower)
 - Bug-free microprograms difficult to write
- User-WCS provided as option on several minicomputers
 - Allowed users to change microcode for each processor
- User-WCS failed
 - Little or no programming tools support
 - Difficult to fit software into small space
 - Microcode control tailored to original ISA, less useful for others
 - Large WCS part of processor state - expensive context switches
 - Protection difficult if user can change microcode
 - Virtual memory required restartable microcode

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 vulnerabilities

Analyzing Microcoded Machines

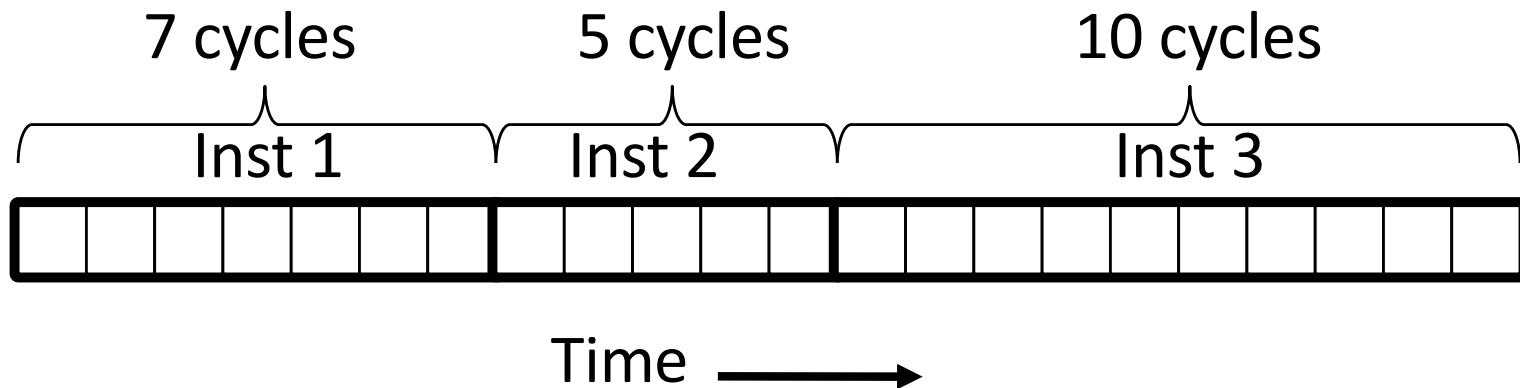
- John Cocke and group at IBM
 - Working on a simple pipelined processor, 801, and advanced compilers inside IBM
 - Ported experimental PL.8 compiler to IBM 370, and only used simple register-register and load/store instructions similar to 801
 - Code ran faster than other existing compilers that used 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



$$\text{Total clock cycles} = 7+5+10 = 22$$

$$\text{Total instructions} = 3$$

$$\text{CPI} = 22/3 = 7.33$$

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 (microcoded 68000 example)

RISC!

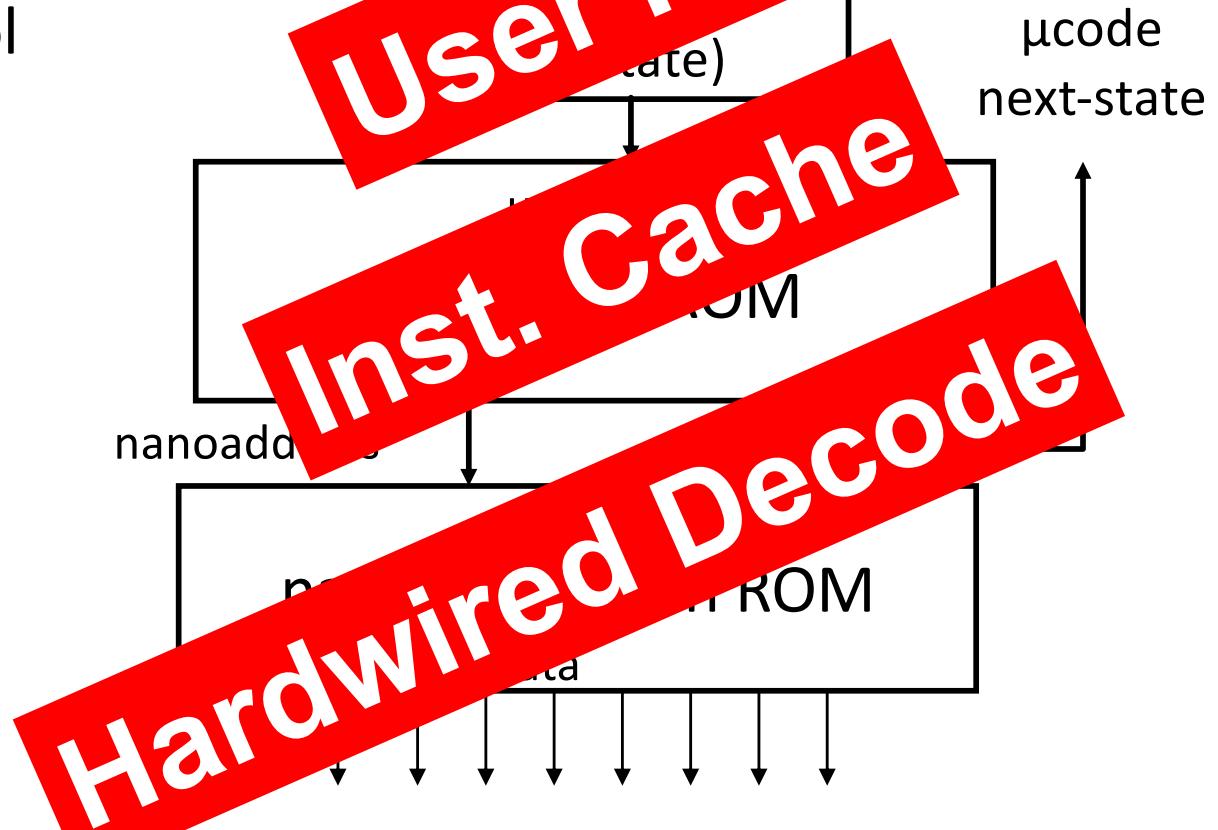
Exploits recurring control
signal patterns in μcode,
e.g.,

ALU0 A ← Reg[rs1]

...

ALUI0 A ← Reg[rs1]

...



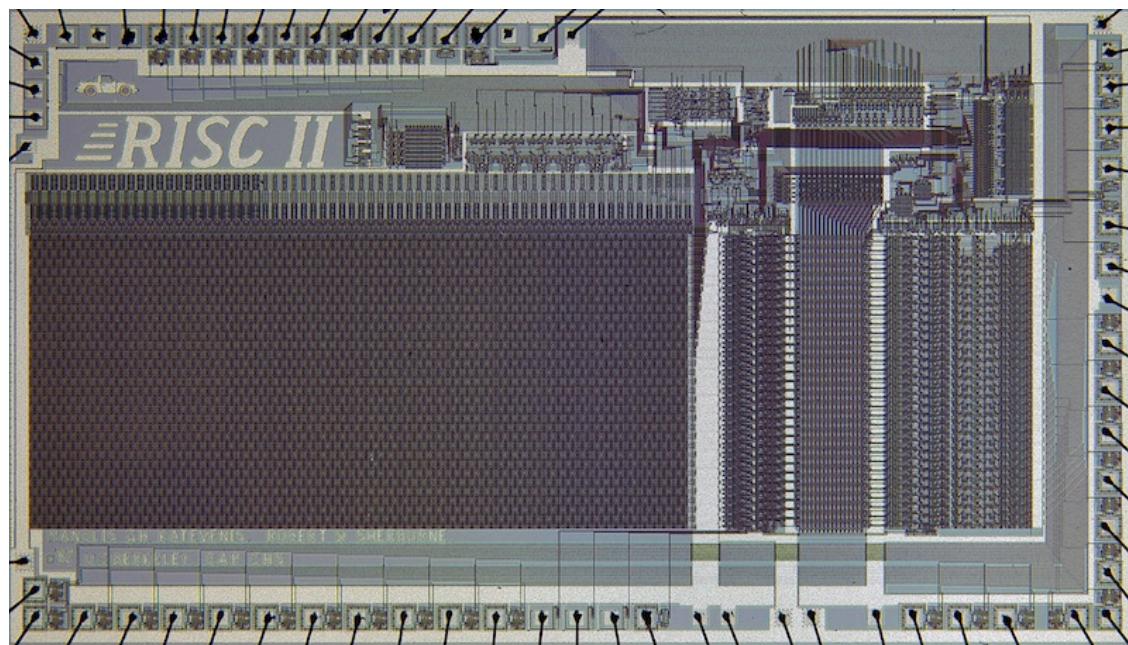
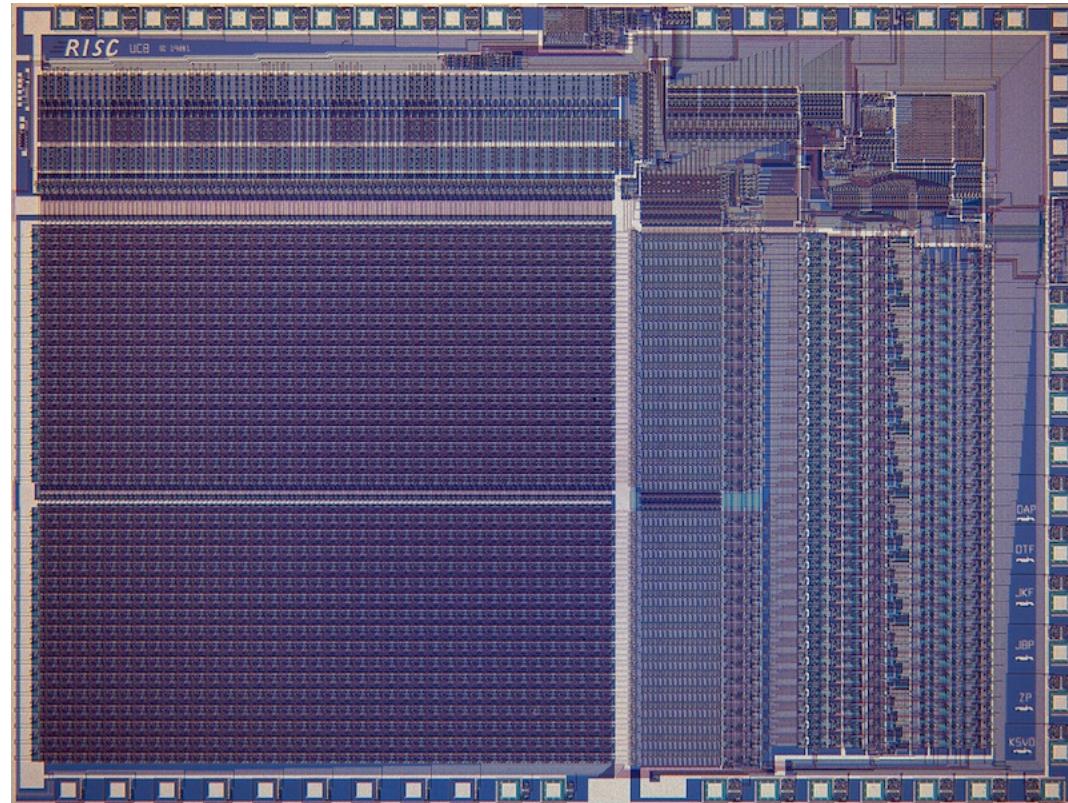
- Motorola 68000 had 17-bit μcode containing either 10-bit μjump or 9-bit nanoinstruction pointer
 - Nanoinstructions were 68 bits wide, decoded to give 196 control signals

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
 - RISC ISA comparable to vertical microcode
- 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...

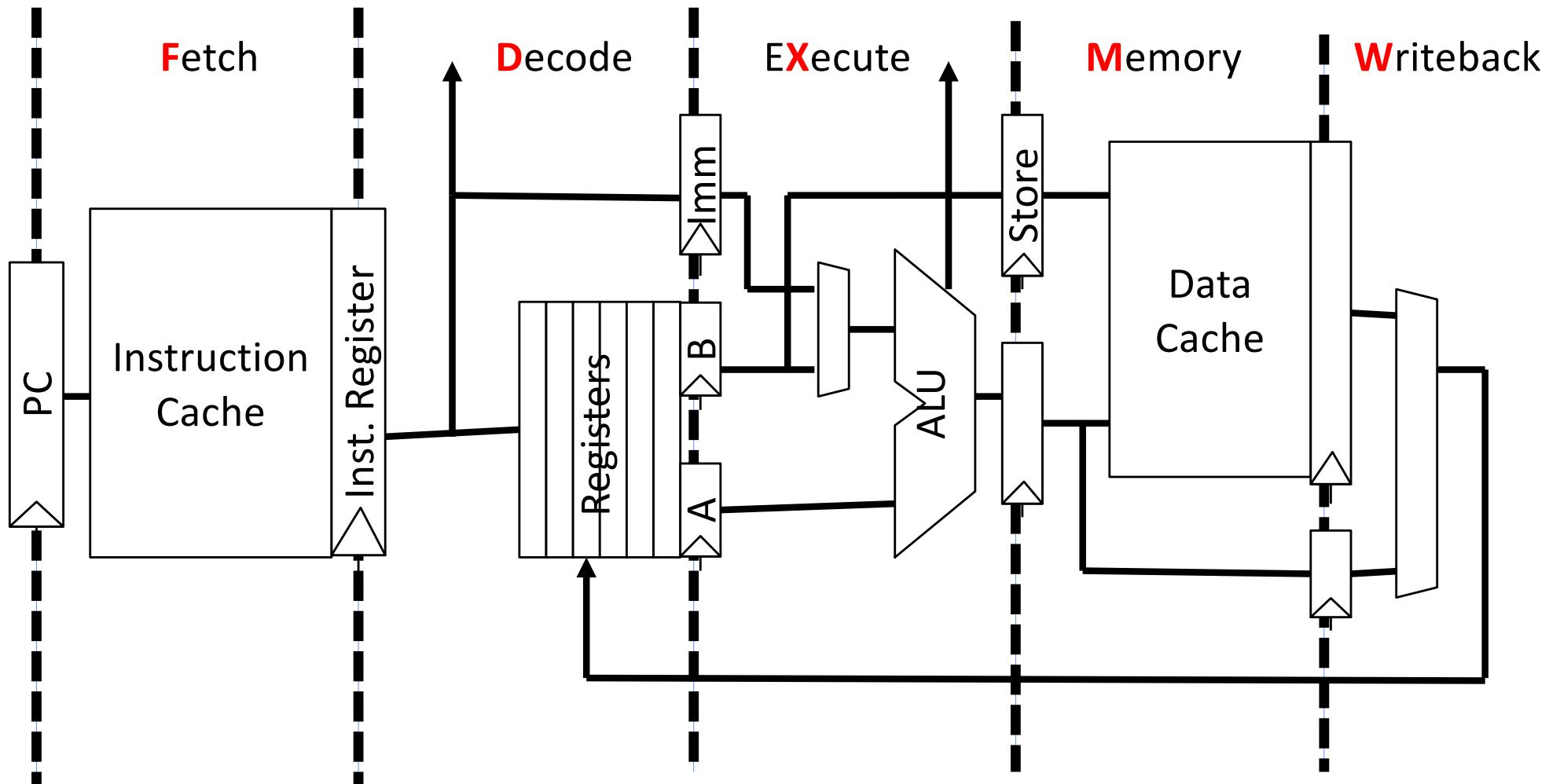
“Iron Law” of Processor Performance

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

- 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

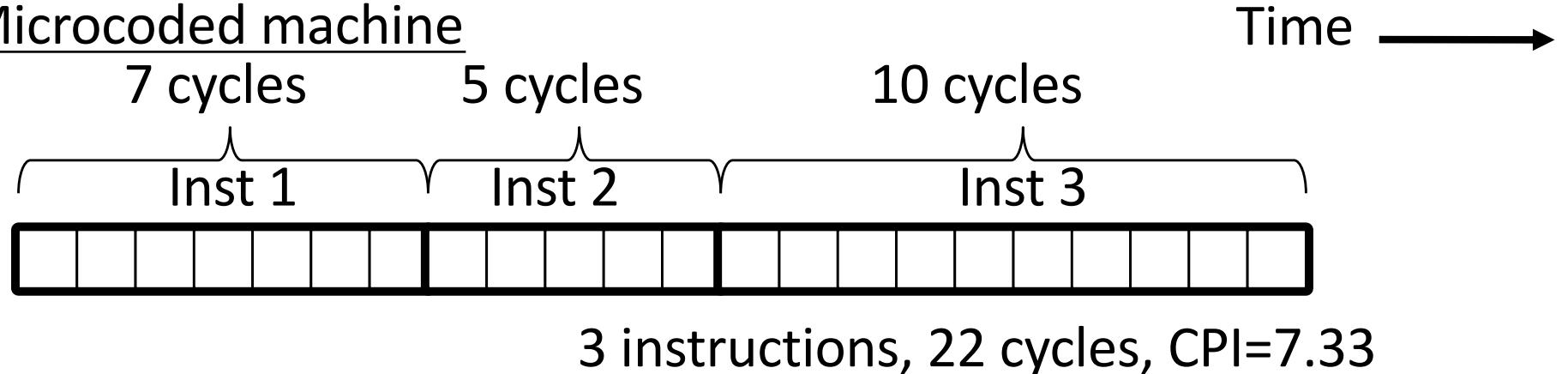
Classic 5-Stage RISC Pipeline



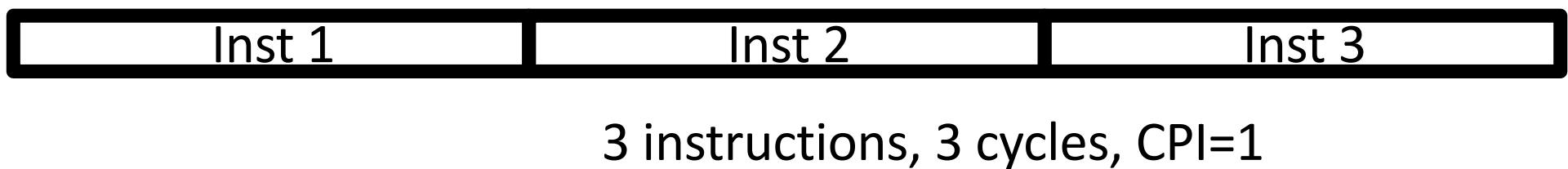
*This version designed for regfiles/memories
with synchronous reads and writes.*

CPI Examples

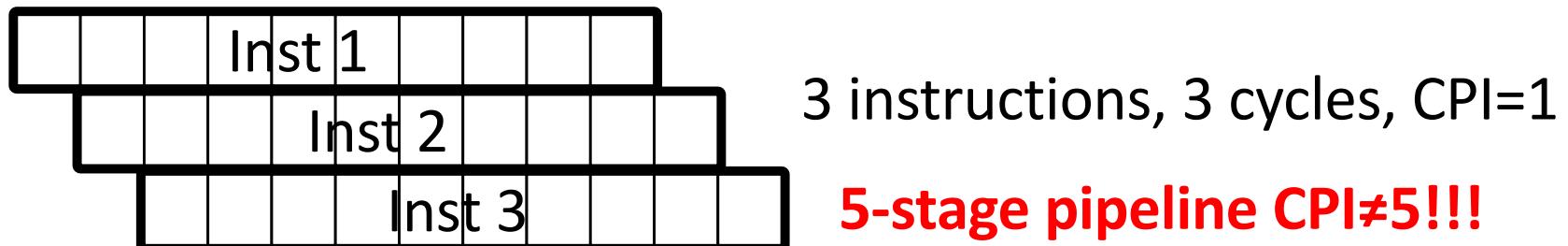
Microcoded machine



Unpipelined machine



Pipelined machine



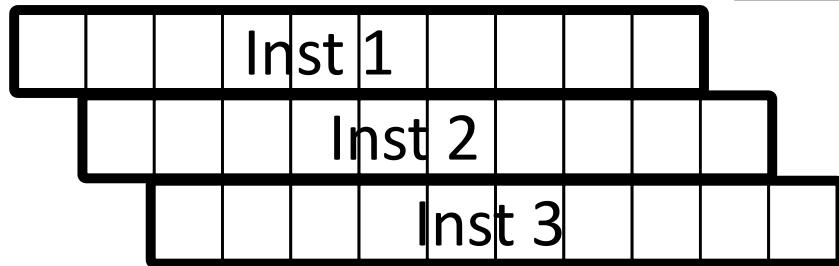
Instructions interact with each other in pipeline

- 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

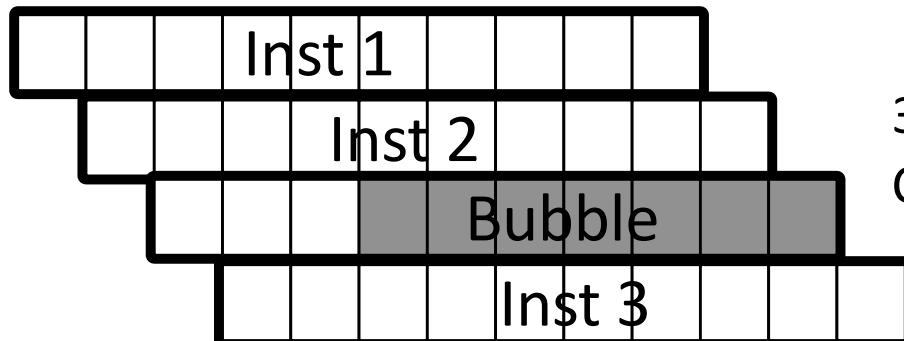
Pipeline CPI Examples

Time →

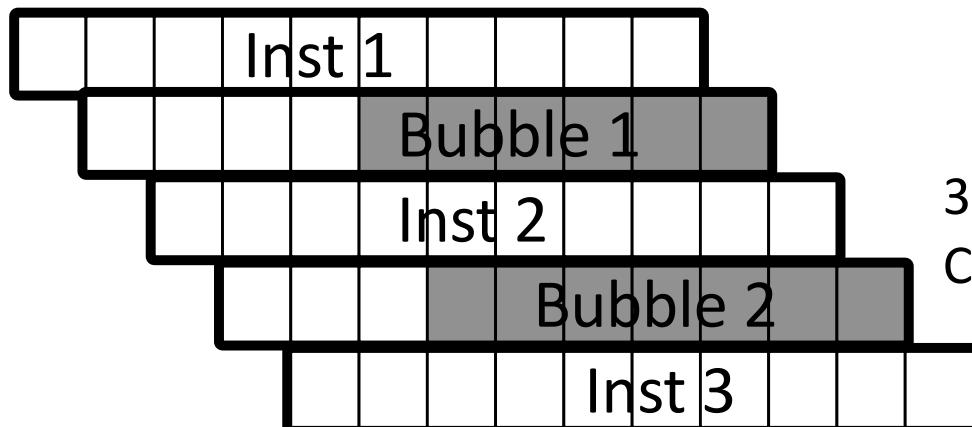
Measure from when first instruction finishes to when last instruction in sequence finishes.



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

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

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

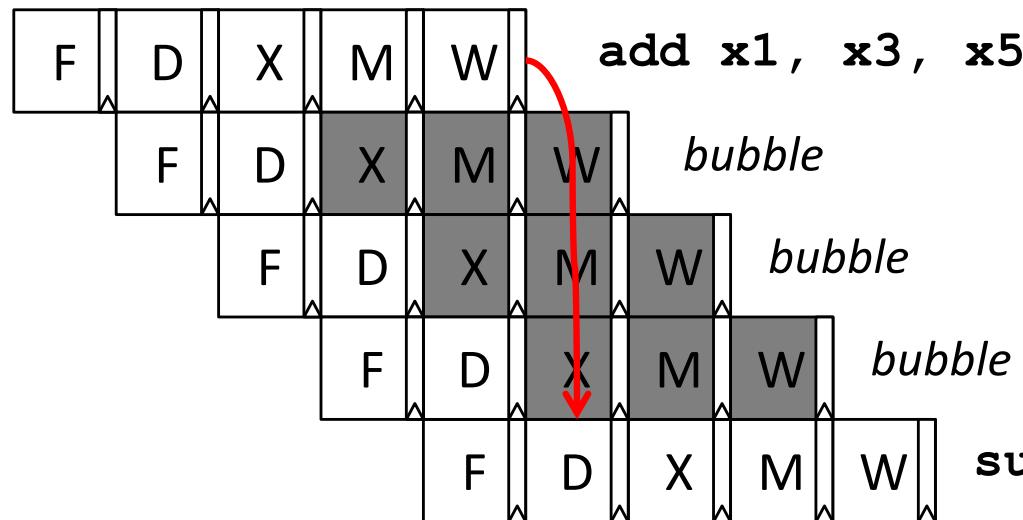
- Resolve hazard earlier by bypassing value as soon as available

- Speculate

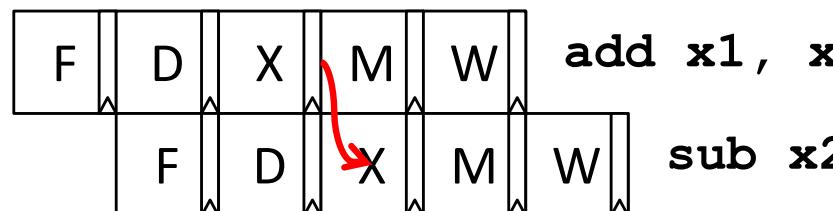
- Guess on value, correct if wrong

Interlocking Versus Bypassing

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

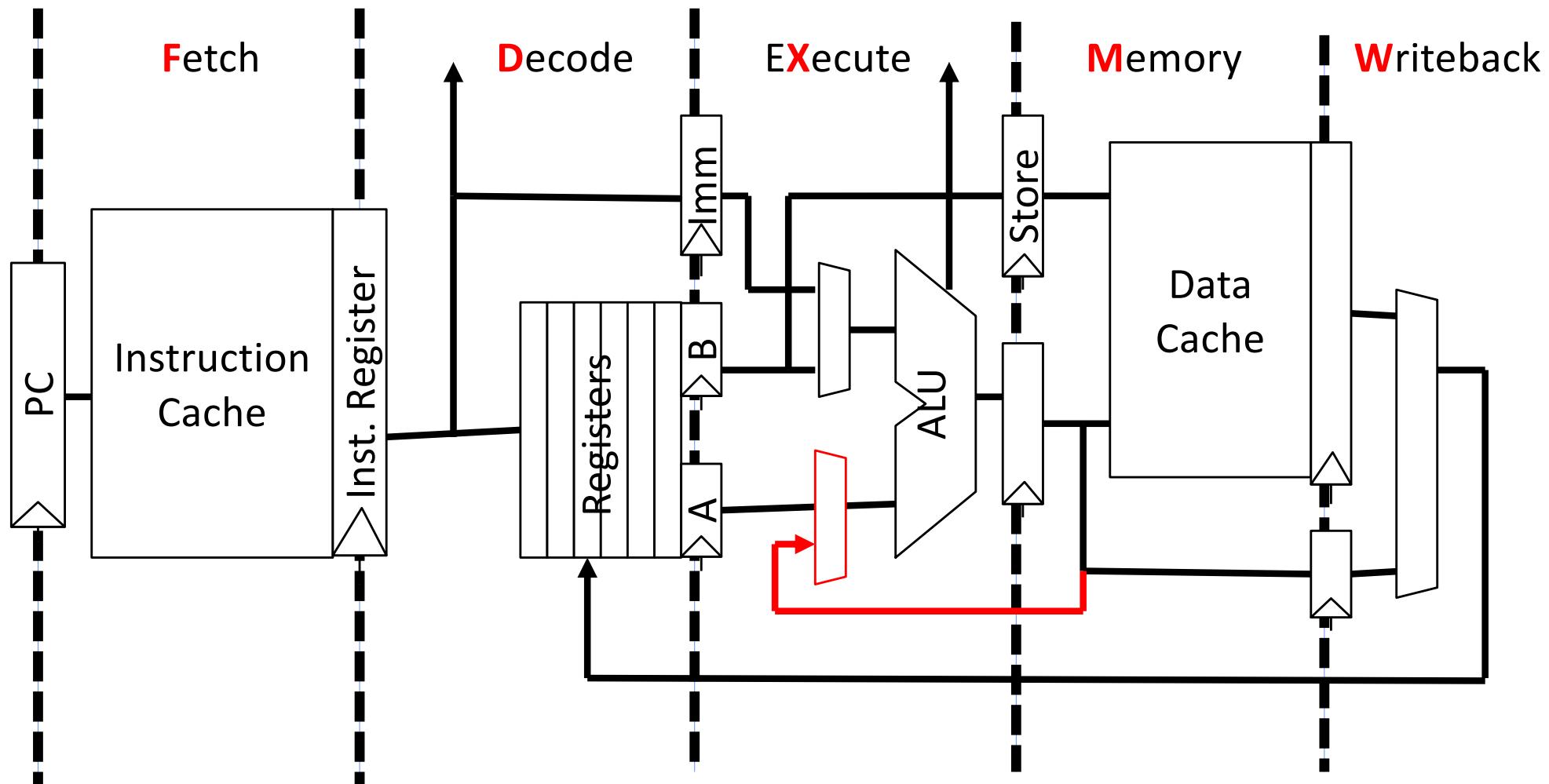


Instruction interlocked
in decode stage

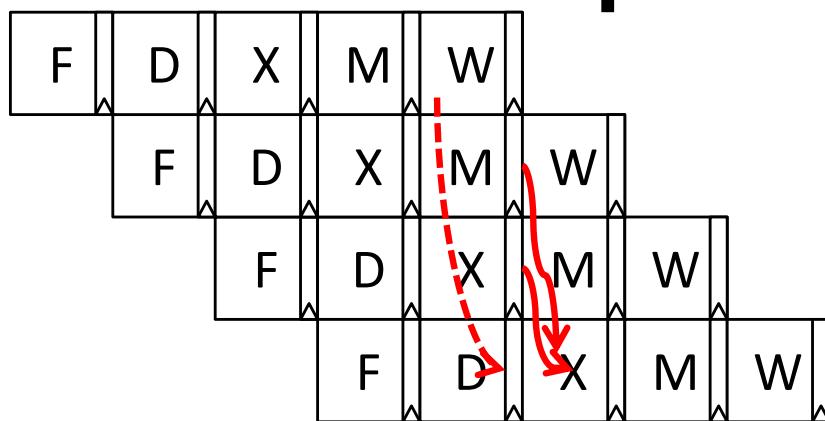
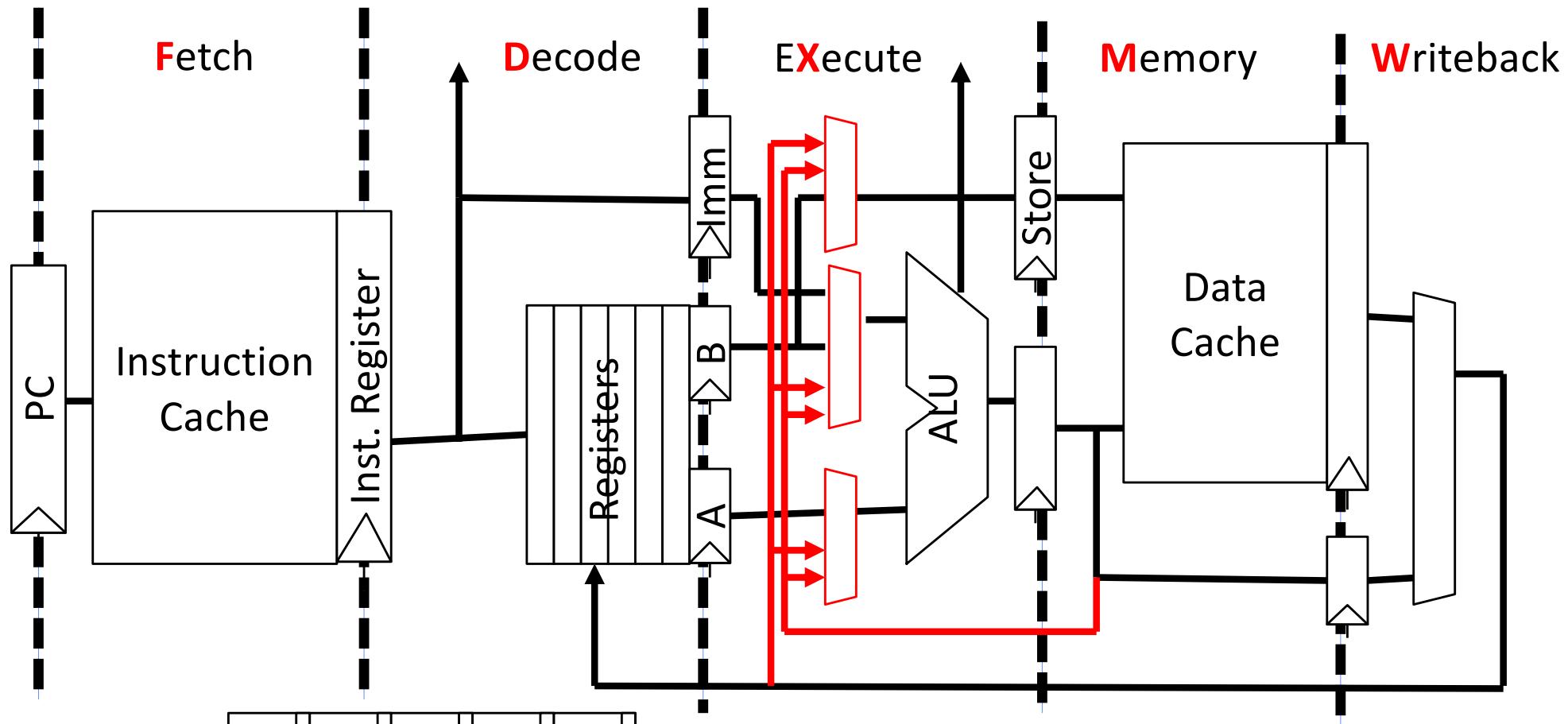


Bypass around ALU
with no bubbles

Example Bypass Path



Fully Bypassed Data Path



[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

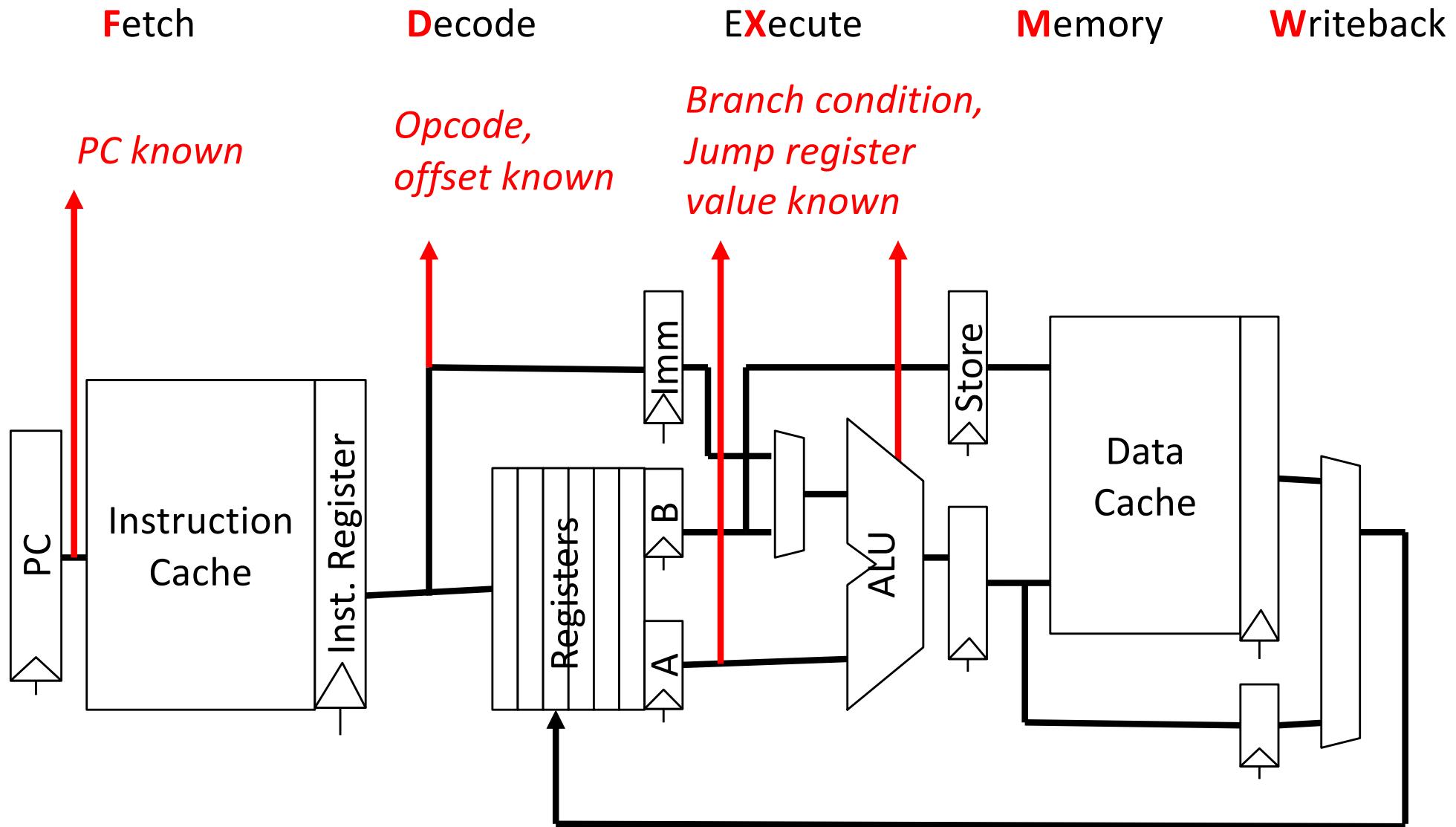
- Rather than wait for value, can guess value!
- So far, only effective in certain limited cases:
 - Branch prediction
 - Stack pointer updates
 - Memory address disambiguation

Control Hazards

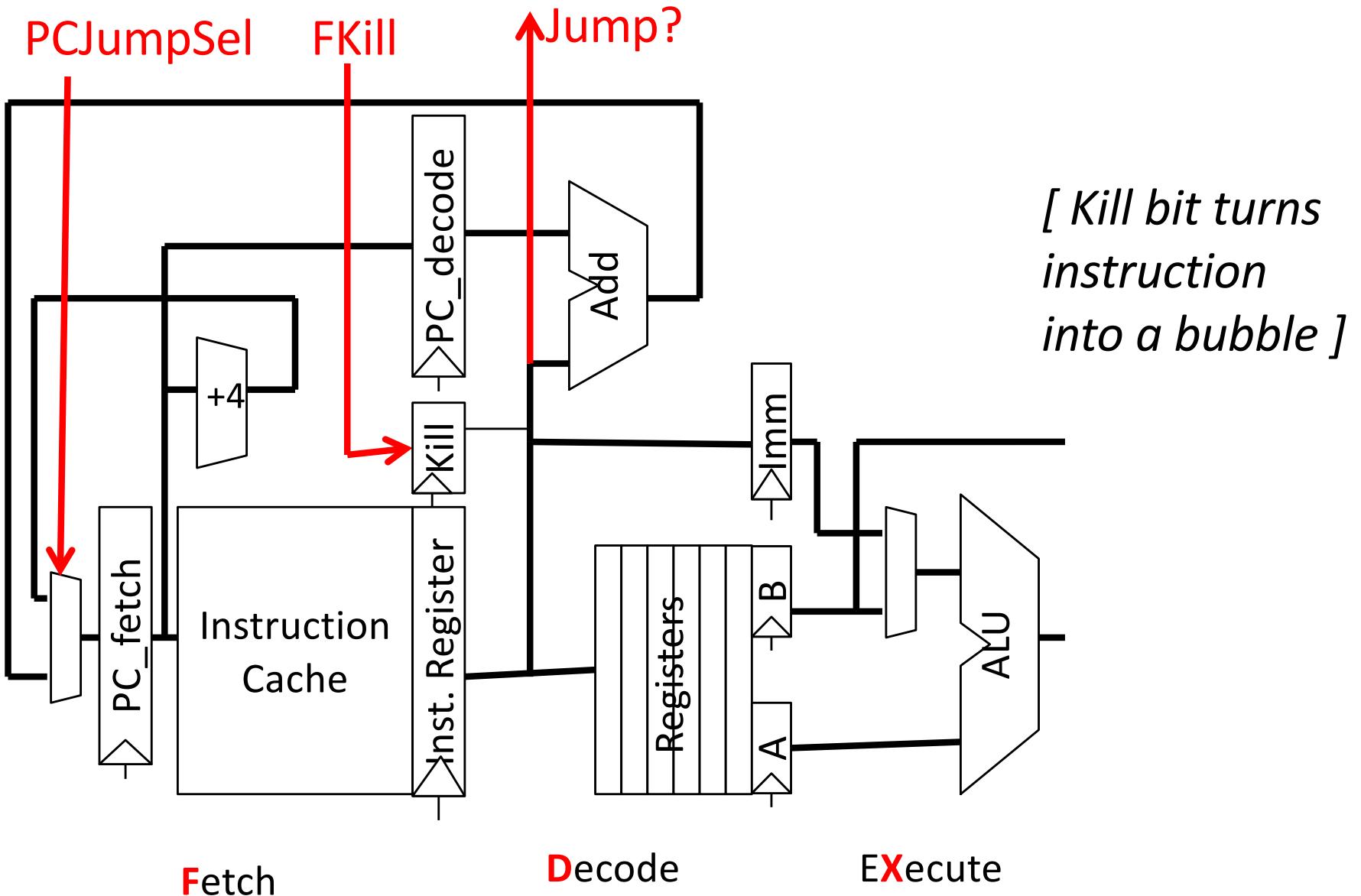
What do we need to calculate next PC?

- 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)

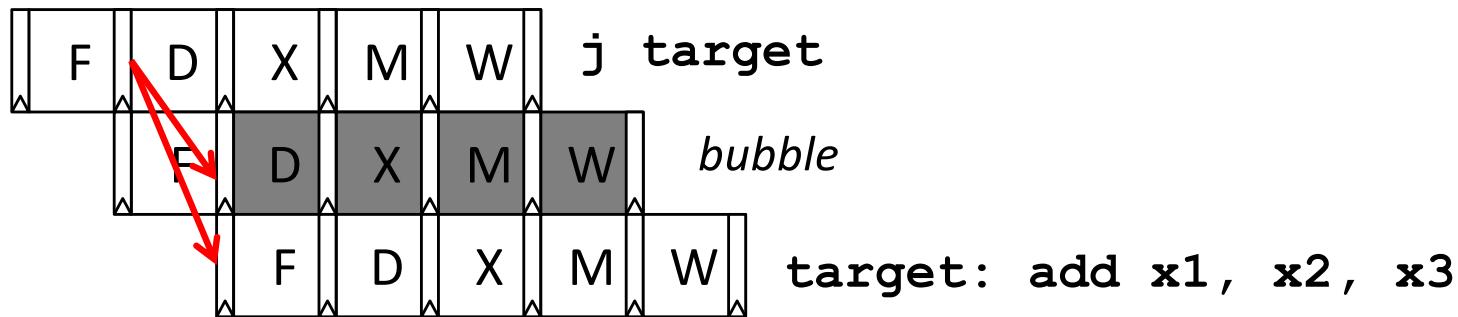
Control flow information in pipeline



RISC-V Unconditional PC-Relative Jumps



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:

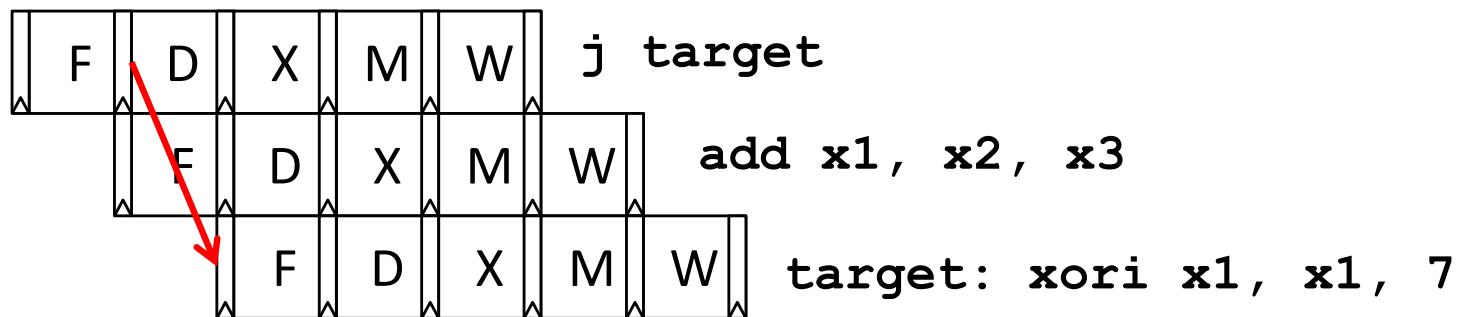
```
0x100 j target
```

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

...

```
0x205 target: xori x1, x1, 7
```

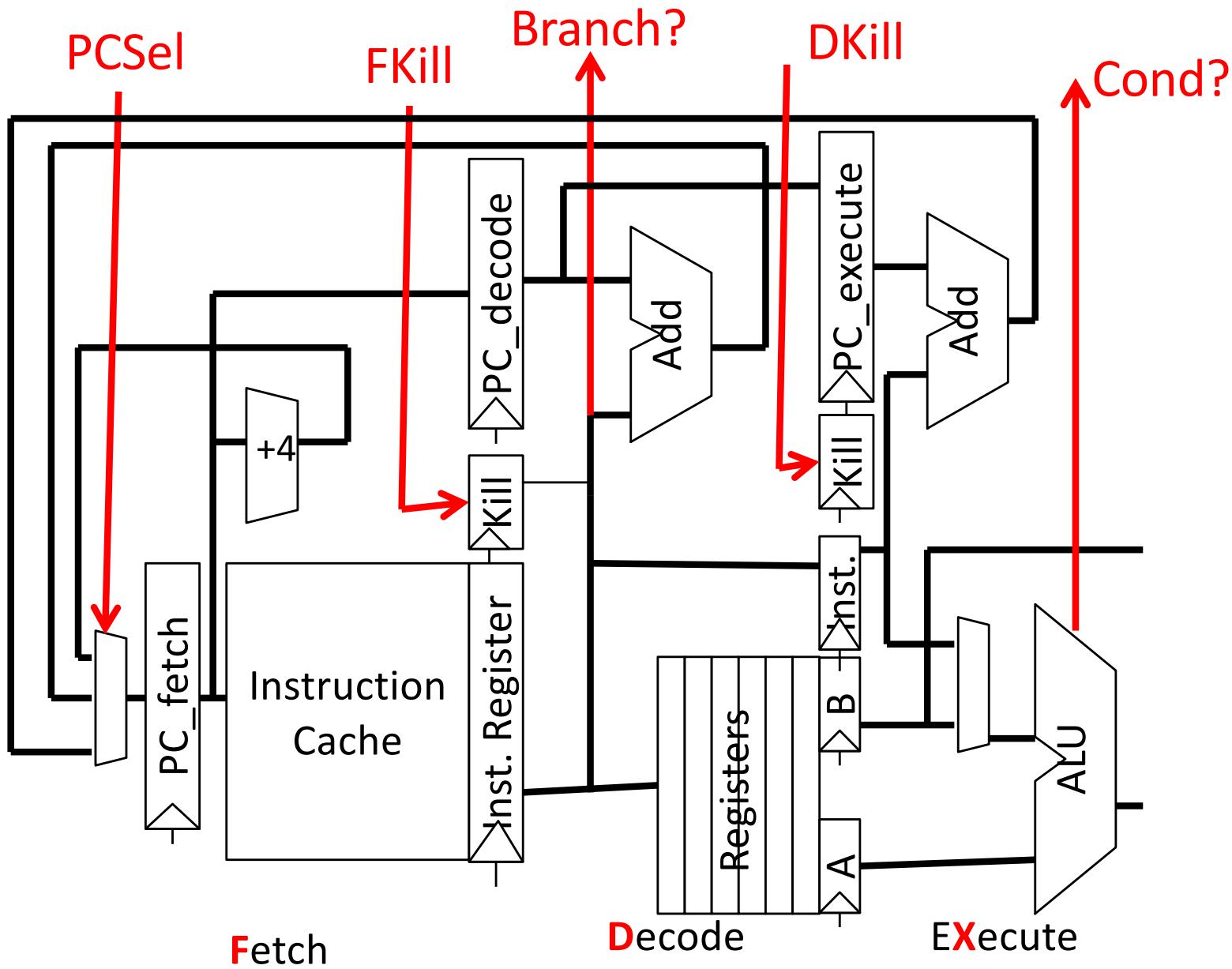
- Software has to fill delay slot with useful work, or fill with explicit NOP instruction



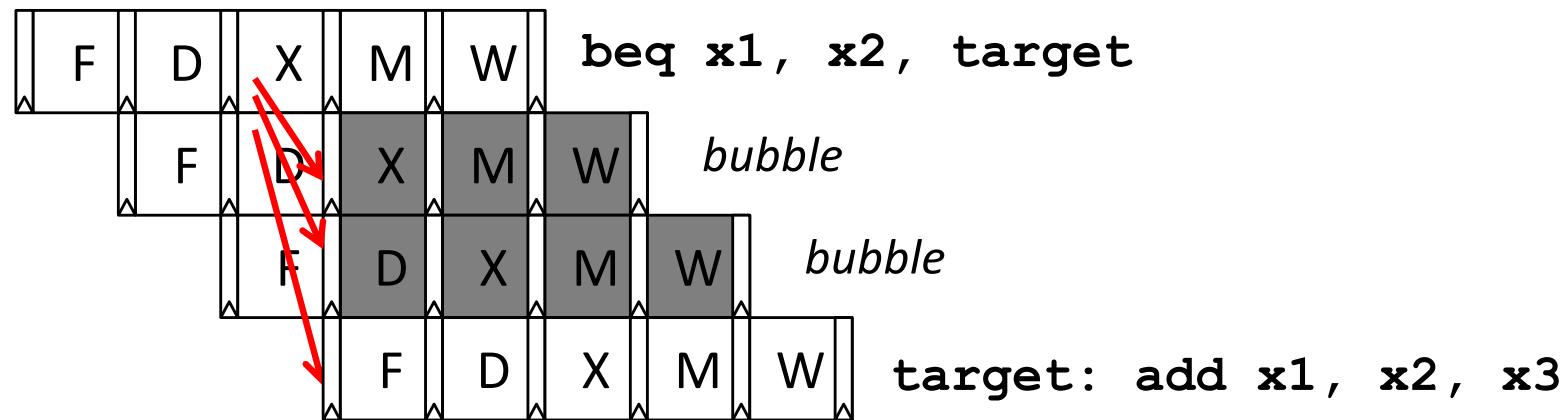
Post-1990 RISC ISAs don't have delay slots

- 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

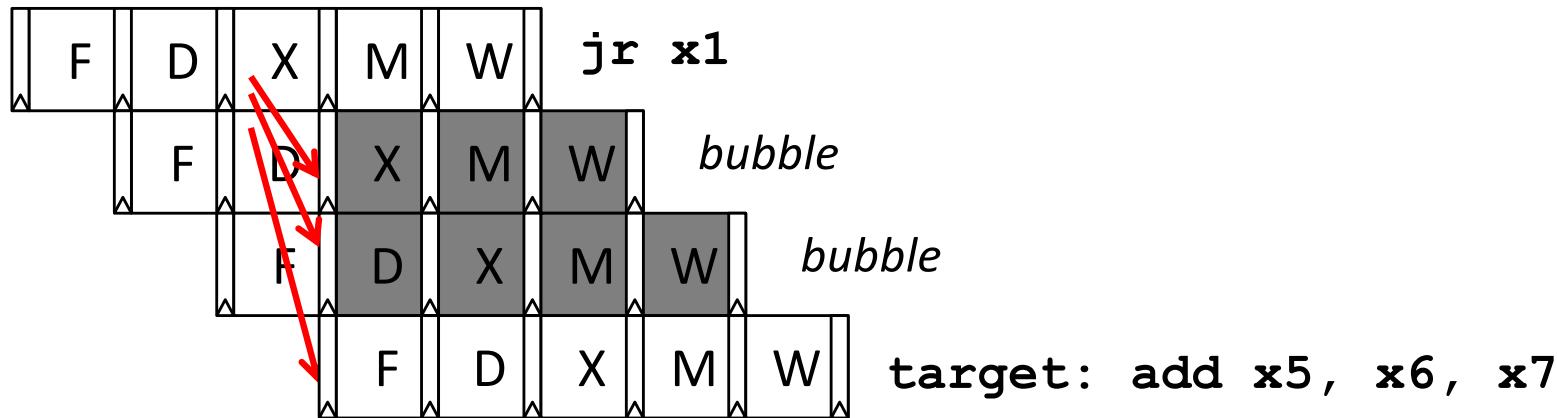


Pipelining for Conditional Branches



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: “Microprocessor without Interlocked Pipeline Stages”
- 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)