CSE 548: Computer Systems Architecture

Pipelining Review

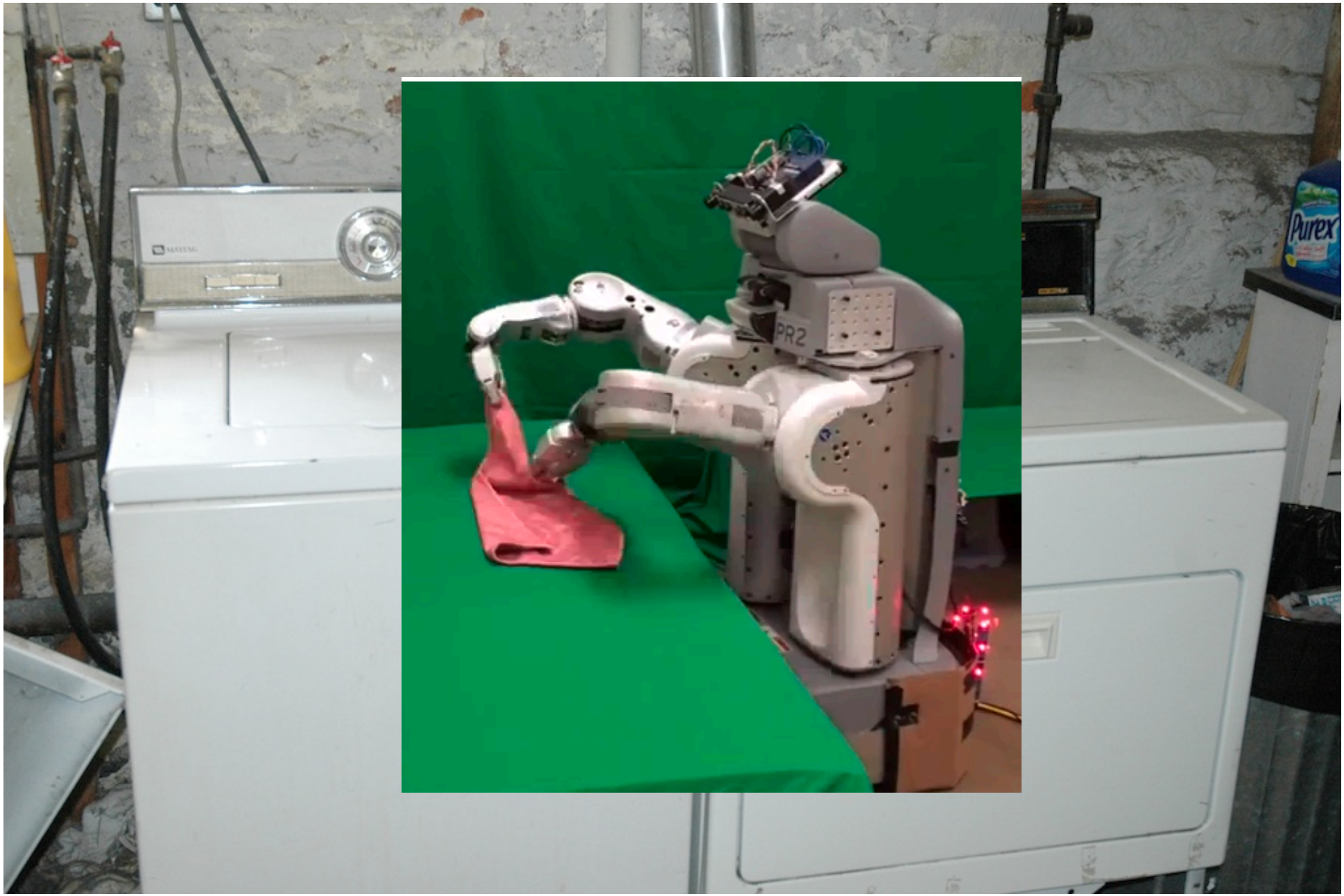
Luis Ceze, Spring 2017

(based on slides lifted from friends at UPenn, UIUC, UW, MIT).

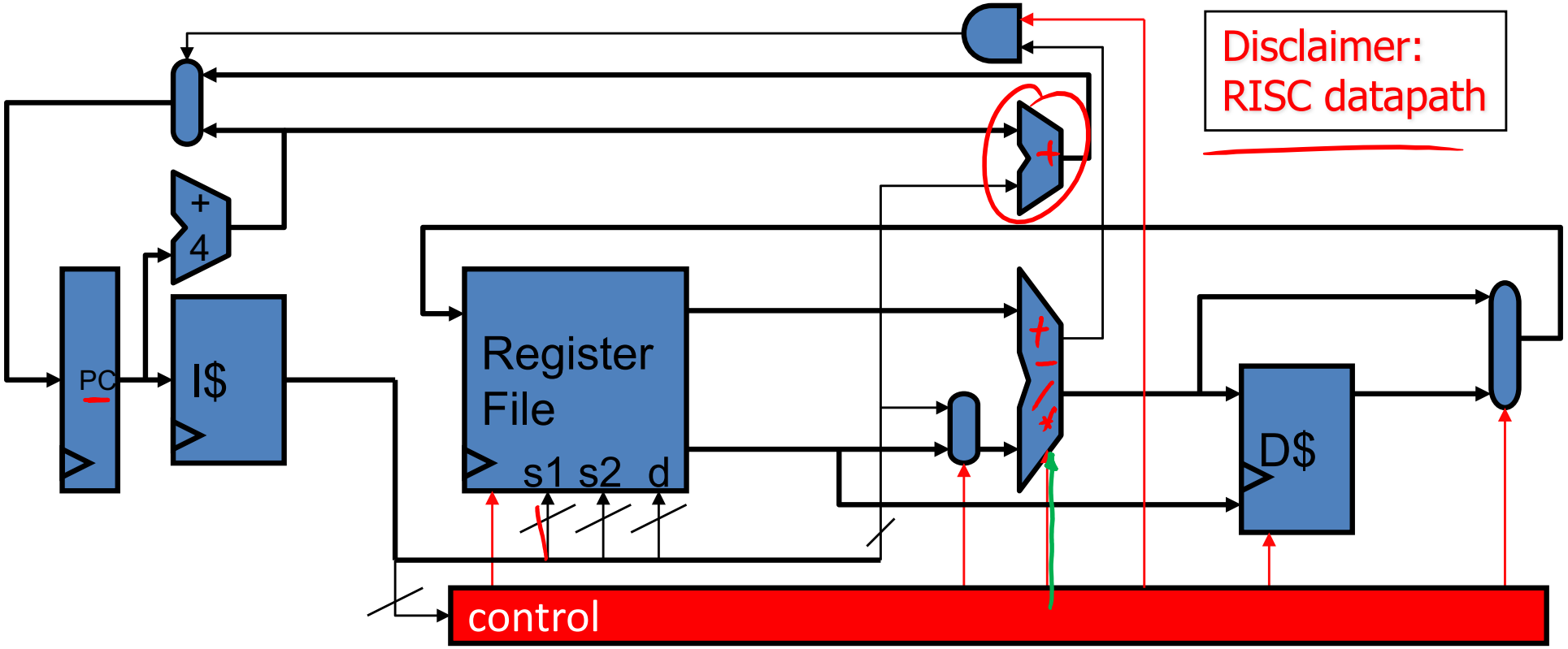
(Scalar In-Order) Pipelining

- Basic Pipelining
  - Pipeline control
- Data Hazards
  - Software interlocks and scheduling
  - Hardware interlocks and stalling
  - Bypassing
- Control Hazards
  - Branch prediction

The eternal pipelining metaphor

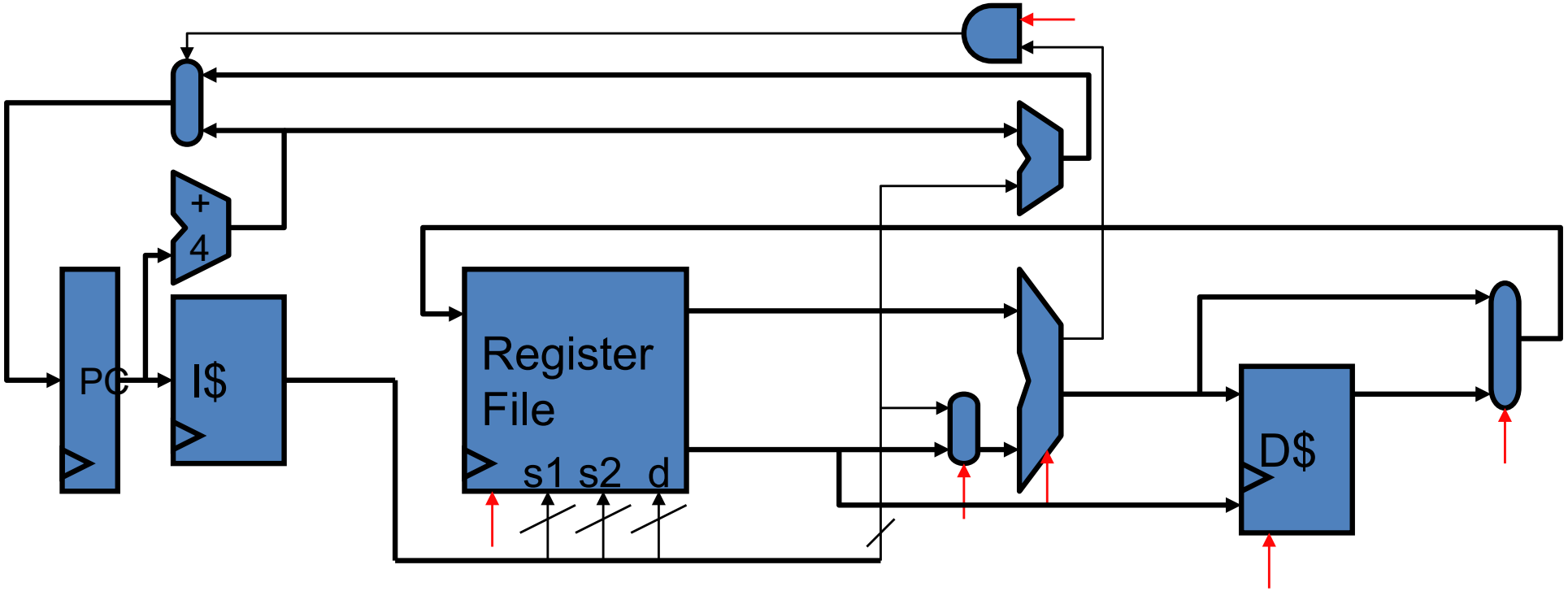


Datapath and Control



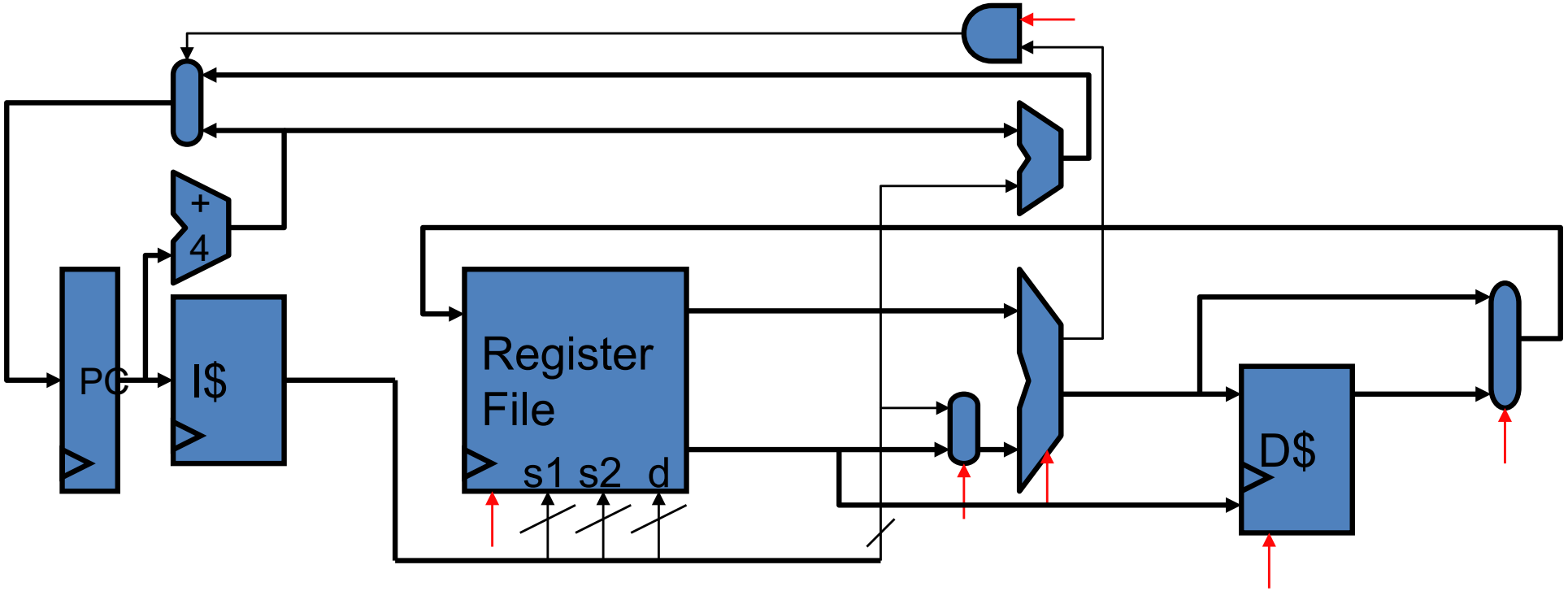
- **Datapath**: implements execute portion of fetch/exec. loop
  - Functional units (ALUs), registers, memory interface
- **Control**: implements decode portion of fetch/execute loop
  - Mux selectors, write enable signals regulate flow of data in datapath
  - Part of decode involves translating insn opcode into control signals

Single-Cycle Datapath



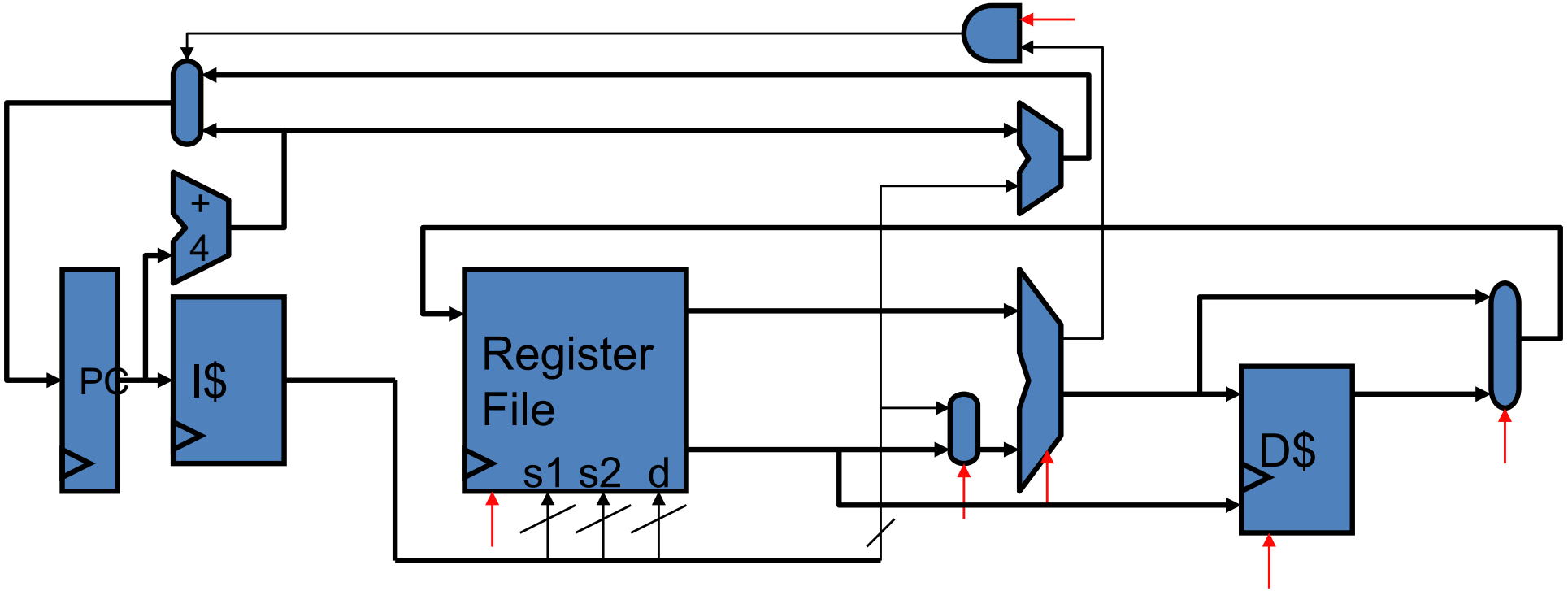
- **Single-cycle datapath**: true "atomic" VonNeuman loop
  - Fetch, decode, execute one complete insn every cycle
  - **"Hardwired control"**: opcode to control signals ROM
  - *What is the CPI? What happens to the clock cycle time?*

Single-Cycle Datapath

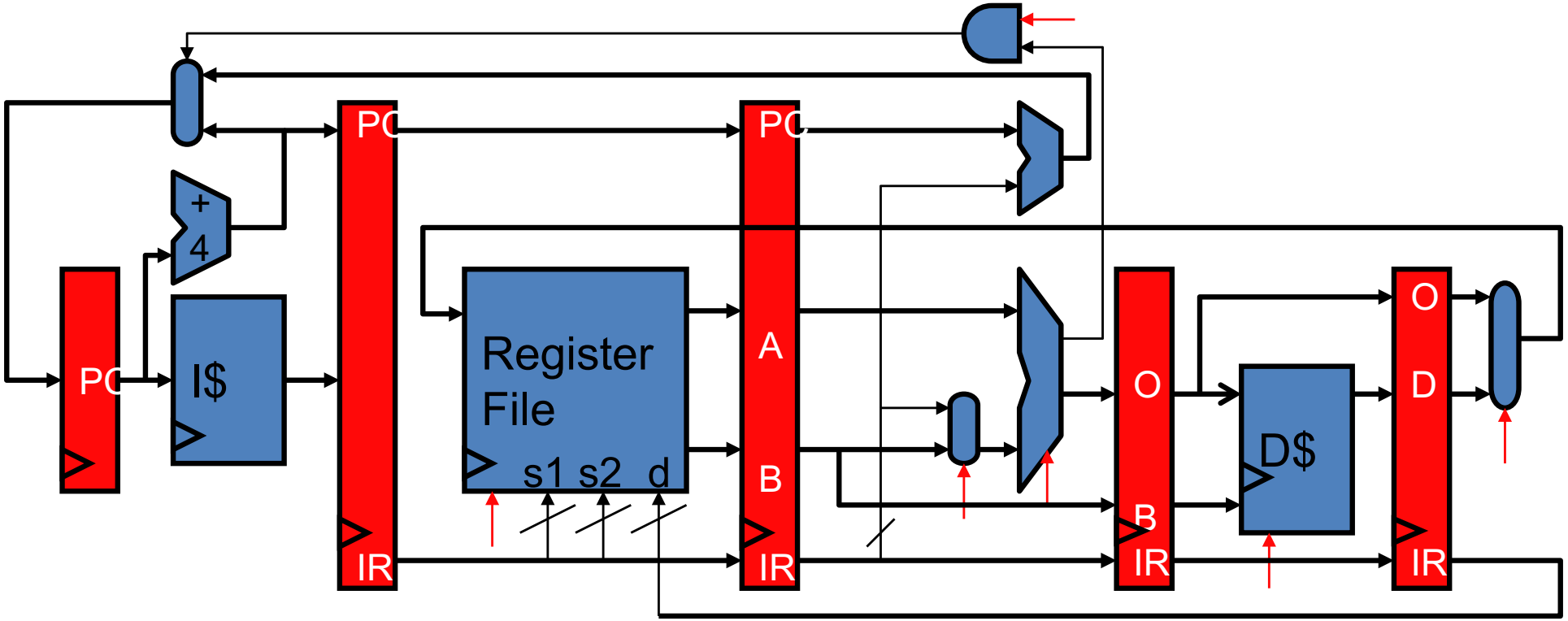


- **Single-cycle datapath**: true "atomic" VonNeuman loop
  - Fetch, decode, execute one complete insn every cycle
  - **"Hardwired control"**: opcode to control signals ROM
  + Low CPI: 1 by definition
  - Long clock period: to accommodate longest insn
- Does all this work need to be done in one shot?

Can we just chop this up?

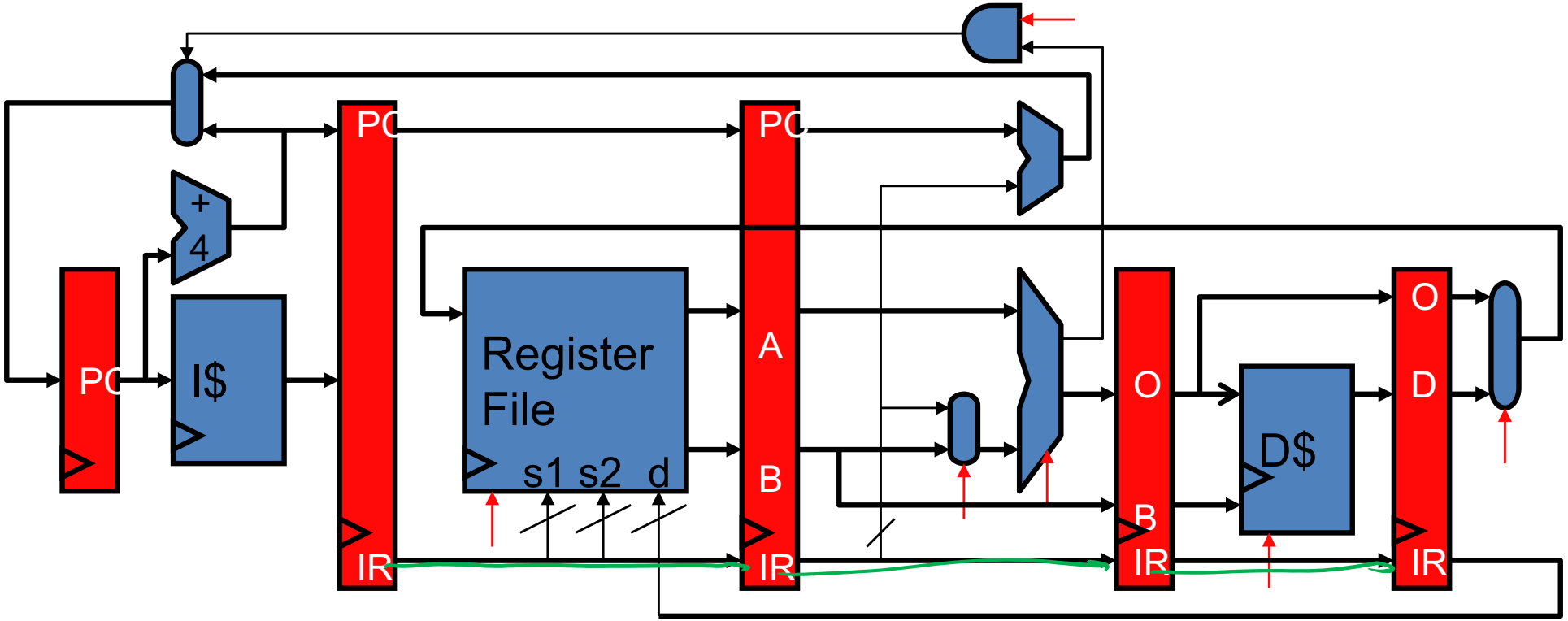


5 Stage Pipelined Datapath



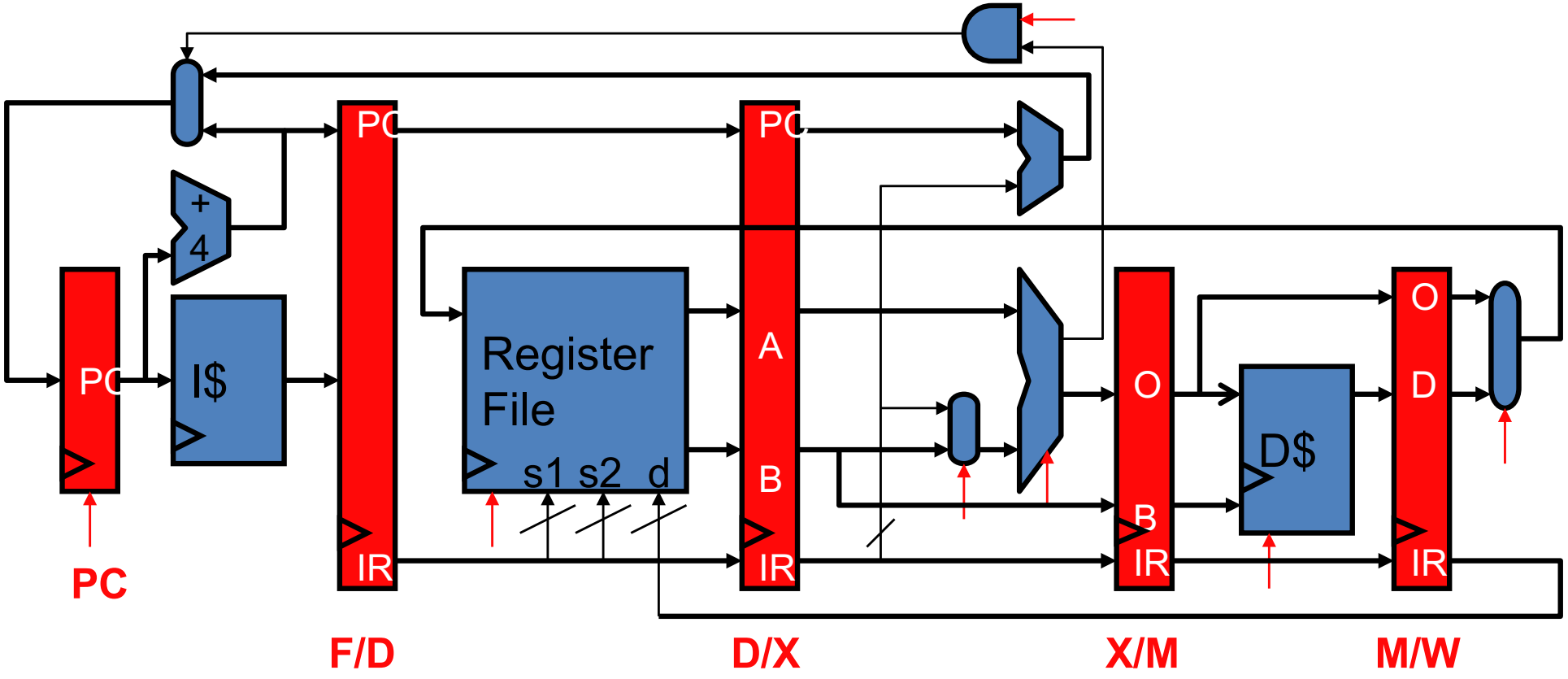
- Temporary values (PC,IR,A,B,O,D) re-latched every stage
  - Why?

5 Stage Pipelined Datapath



- Temporary values (PC,IR,A,B,O,D) re-latched every stage
  - Why? 5 insns may be in pipeline at once with different PCs
  - **Pipelined control**: one single-cycle controller
    - Control signals themselves pipelined

Pipeline Terminology

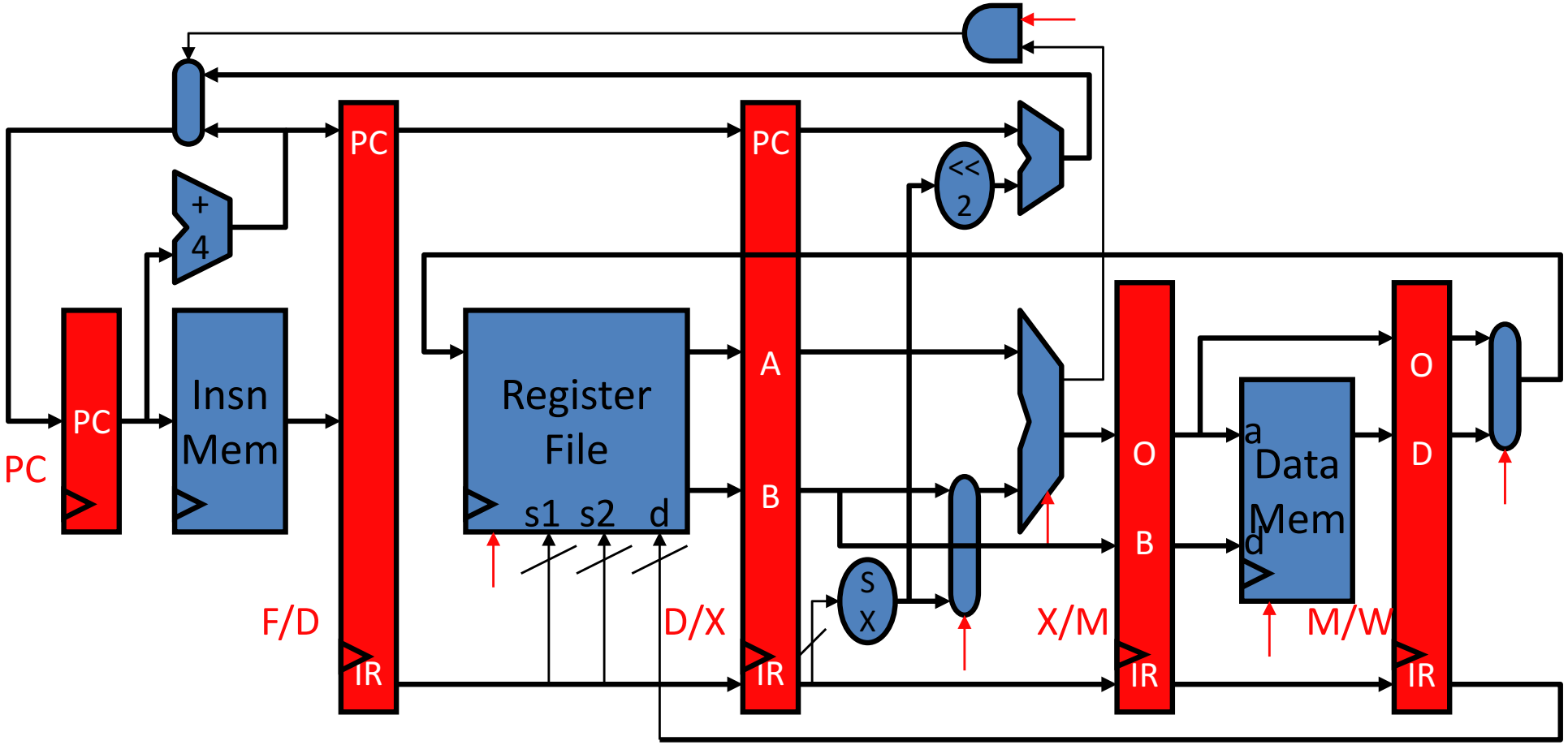


- Five stage: **F**etch, **D**ecode, e**X**ecute, **M**emory, **W**riteback
  - Nothing magical about the number 5 (Pentium 4 has 22 stages)
- Latches (pipeline registers) named by stages they separate
  - **PC**, **F/D**, **D/X**, **X/M**, **M/W**

More Terminology & Foreshadowing

- **Scalar pipeline**: one insn per stage per cycle
  - Alternative: "superscalar" (later)
- **In-order pipeline**: insns enter execute stage in program order
  - Alternative: "out-of-order" (later)
- **Pipeline depth**: number of pipeline stages
  - Nothing magical about five
  - Trend has been to deeper pipelines

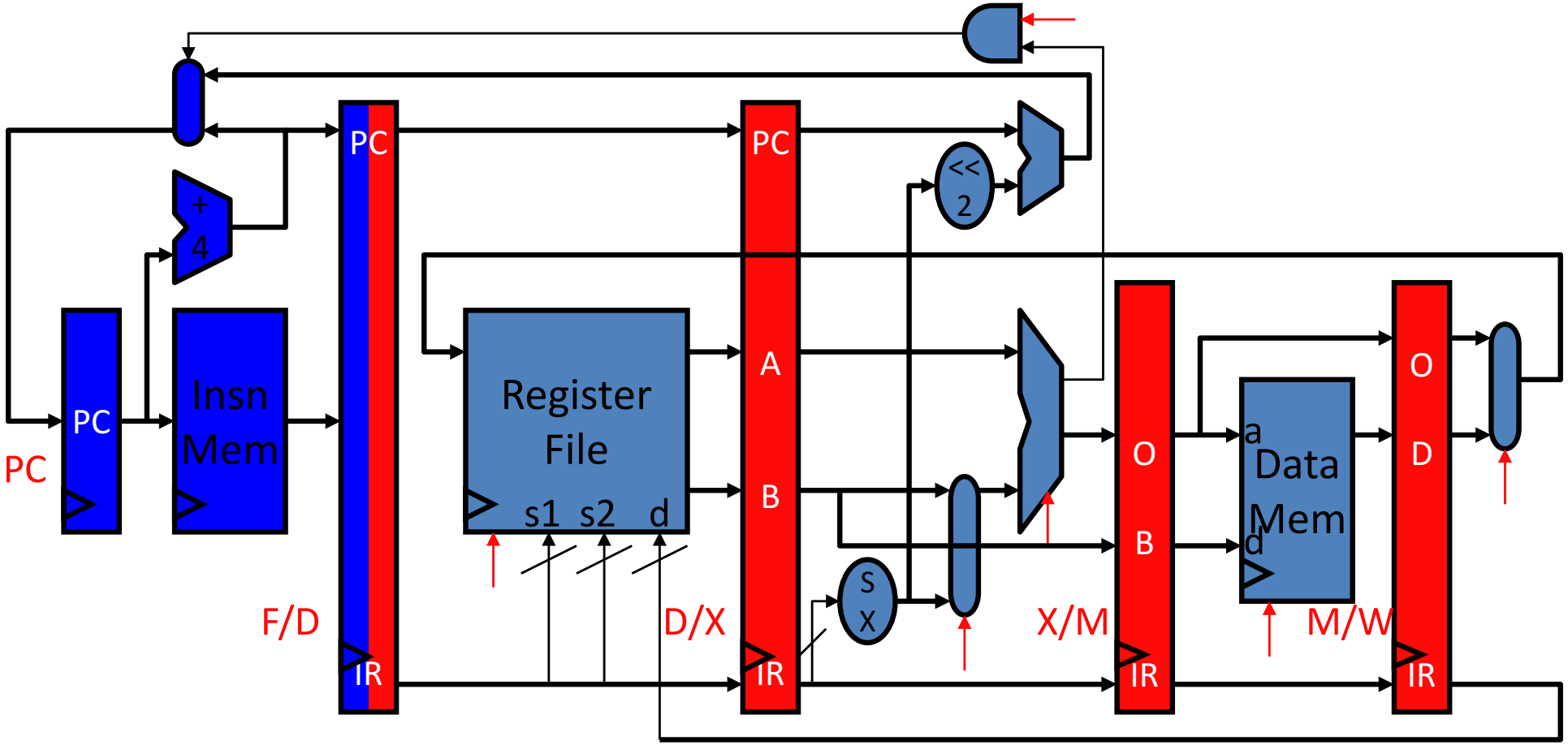
Pipeline Example: Cycle 1




```
add $3,$2,$1
```

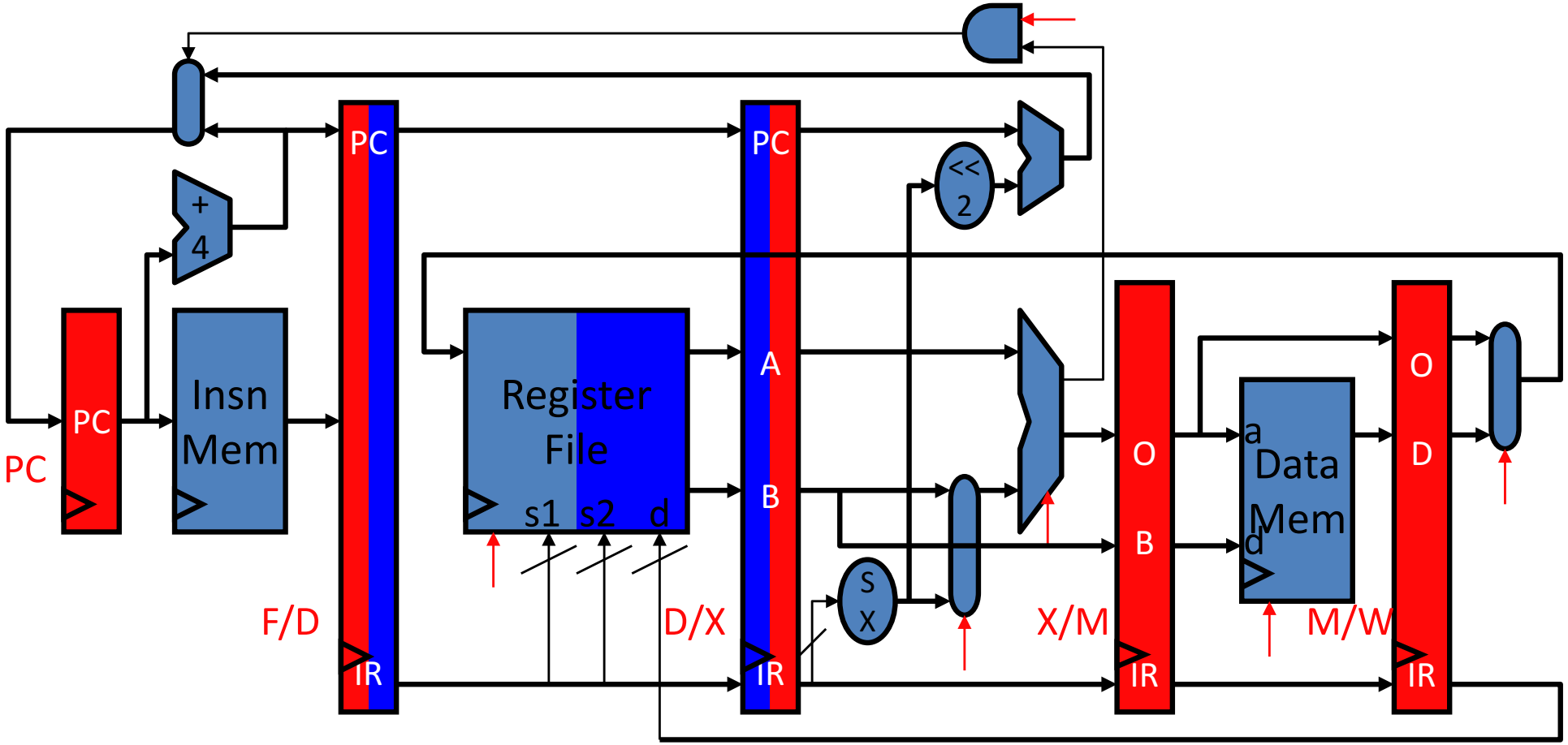
- 3 instructions

Pipeline Example: Cycle 2

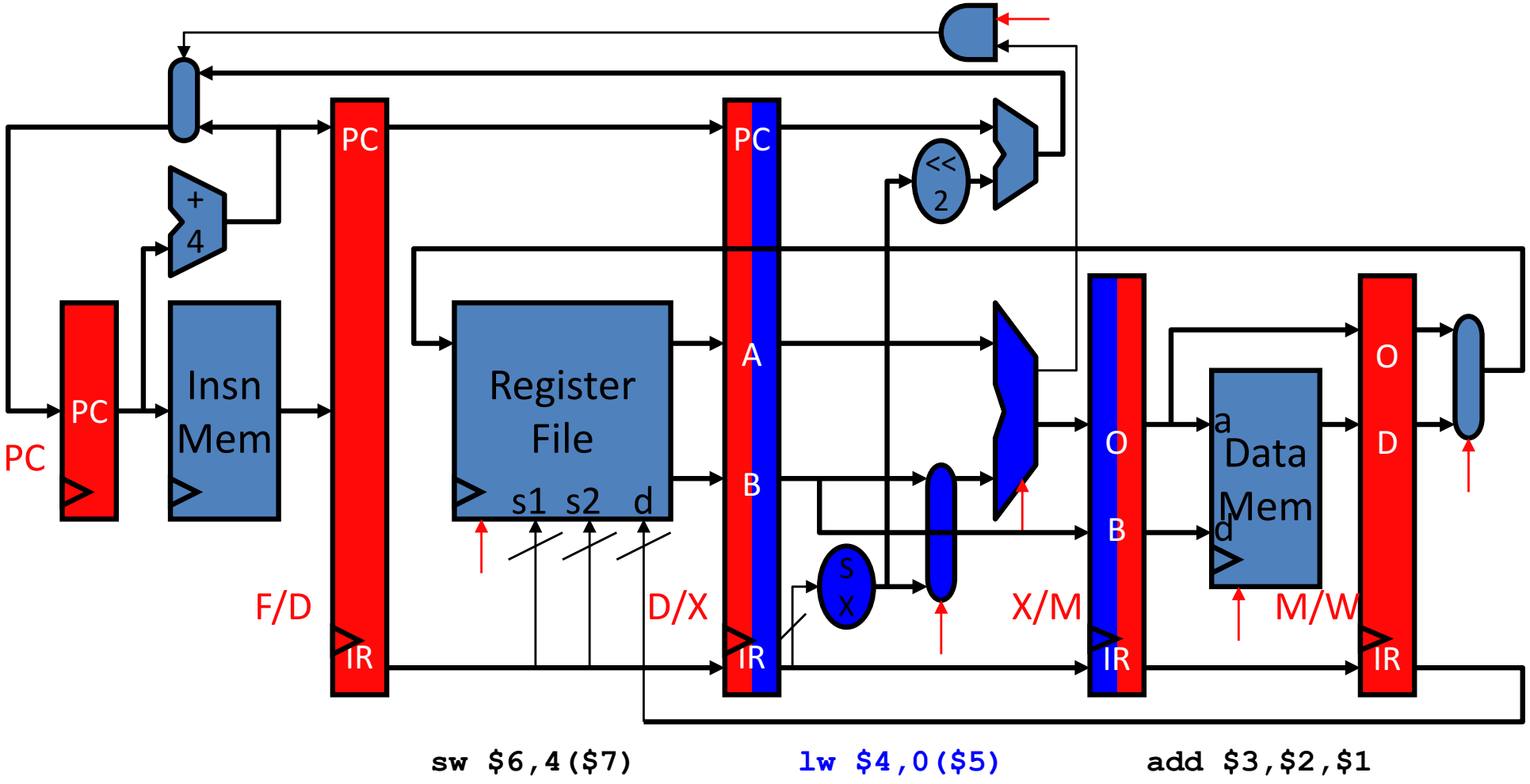


`lw $4,0($5)` `add $3,$2,$1`

Pipeline Example: Cycle 3

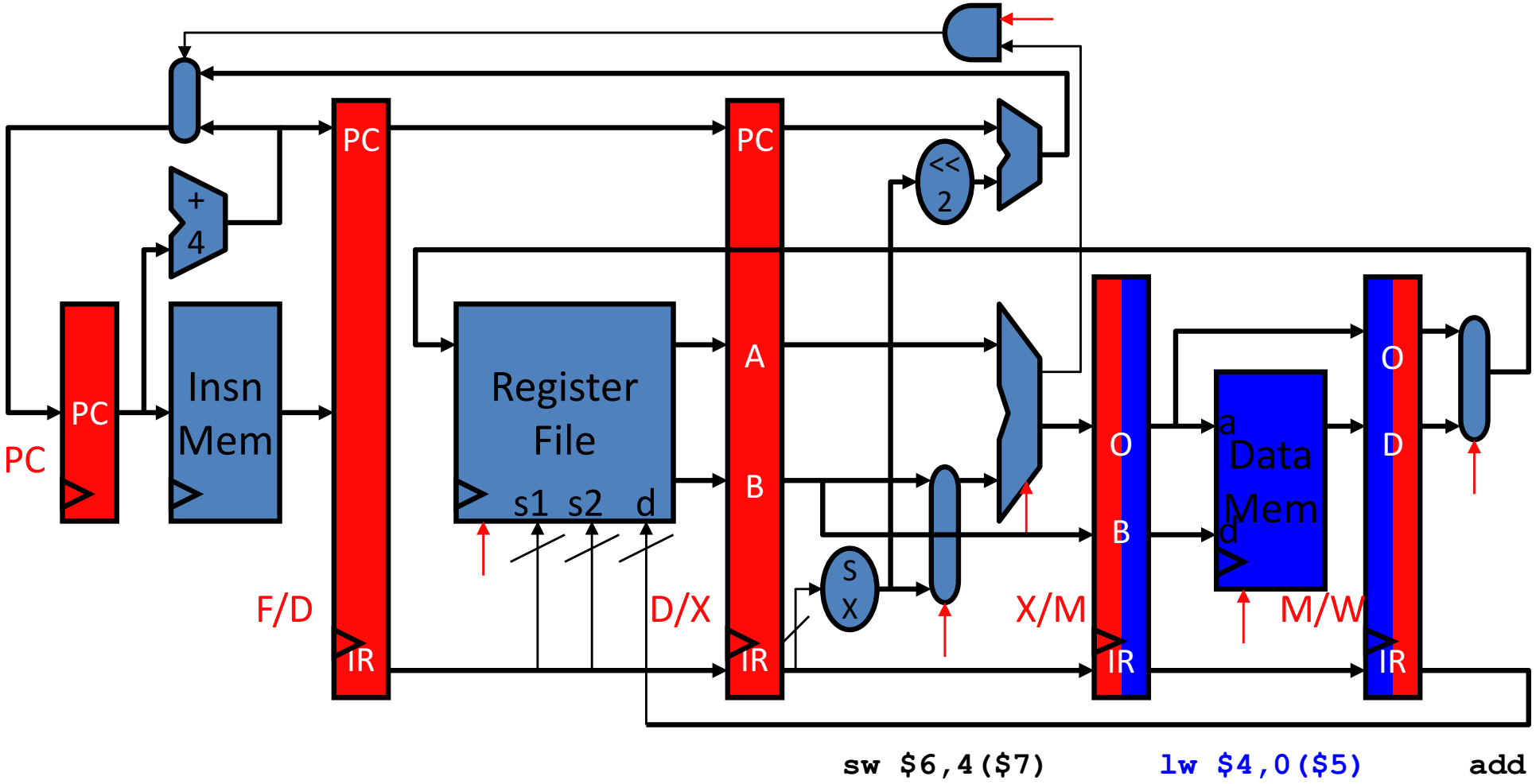


Pipeline Example: Cycle 4

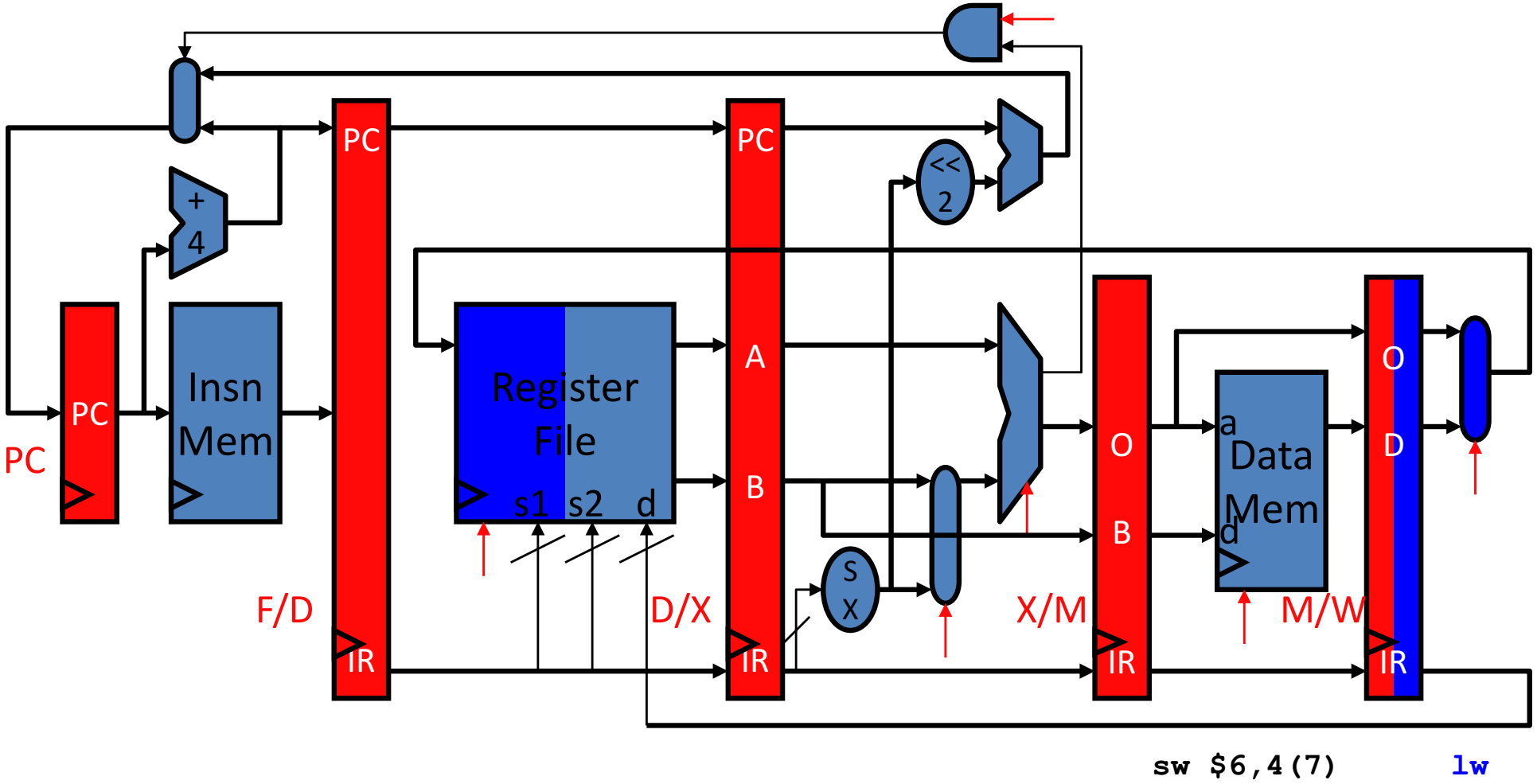


- 3 instructions

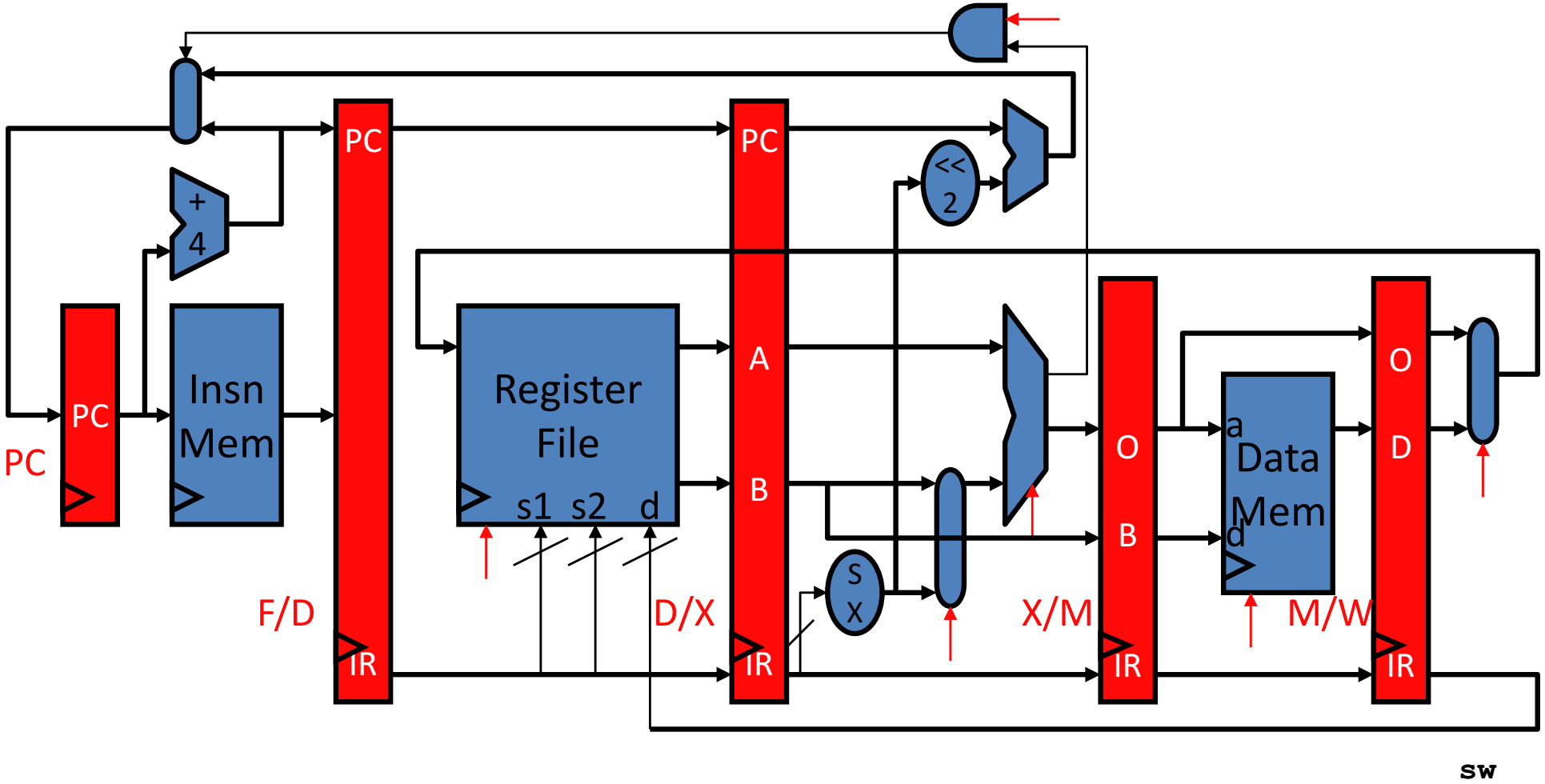
Pipeline Example: Cycle 5



Pipeline Example: Cycle 6



Pipeline Example: Cycle 7



Pipeline Diagram

- **Pipeline diagram**: shorthand for what we just saw
  - Across: cycles
  - Down: insns
  - Convention: **X** means **`lw $4,0($5)`** finishes execute stage and writes into X/M latch at end of cycle 4

	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,0($5)`		F	D	**X**	M	W			
`sw $6,4($7)`			F	D	X	M	W		

Example Pipeline Perf. Calculation

- Single-cycle
  - Clock period = 50ns, CPI = 1
  - Performance = 50ns/insn
- 5-stage pipelined?
  - Clock period = **12ns** (approx. (50ns / 5 stages) + overheads)
  - \+ CPI ?
  - \+ Performance? ___ns/insn

Example Pipeline Perf. Calculation

- Single-cycle
  - Clock period = 50ns, CPI = 1
  - Performance = 50ns/insn
- 5-stage pipelined
  - Clock period = **12ns** (approx. (50ns / 5 stages) + overheads)
  - \+ CPI = **1** (each insn takes 5 cycles, but 1 completes each cycle)
  - \+ Performance = **12ns/insn**
  - Well actually … CPI = 1 + some penalty for pipelining (next)
    - CPI = **1.5** (on average insn completes every 1.5 cycles)
    - Performance = **18ns/insn**

    - Latches add delay
    - Extra "bypassing" logic adds delay
    - Pipeline stages have different delays, clock period is max delay

Q1: Why Is Pipeline ***CPI***…

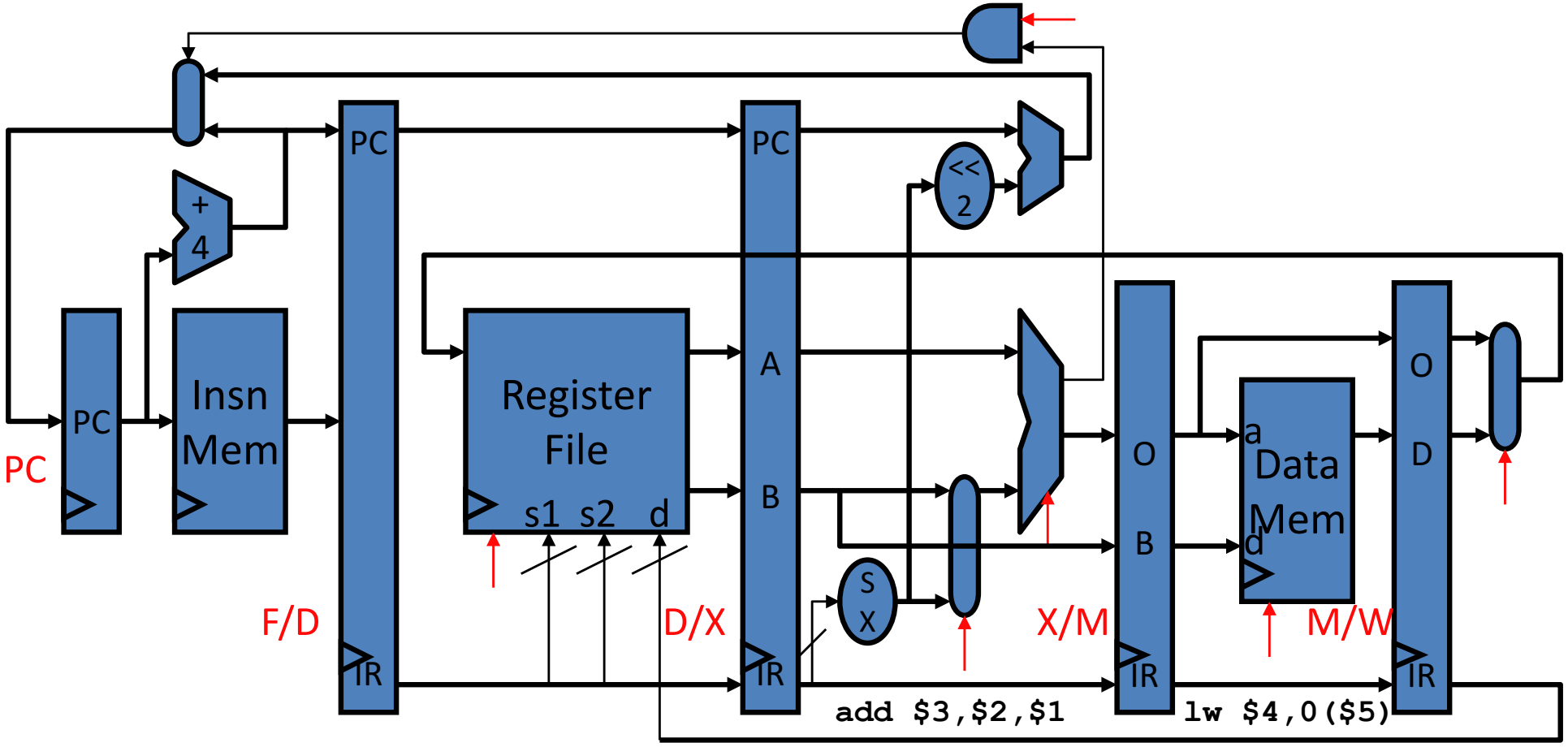
- … > 1?
  - CPI for scalar in-order pipeline is 1 **+ stall penalties**
  - Pipelining is not always smooth…
  - Stalls used to resolve hazards
    - **Hazard**: condition that jeopardizes pipeline flow
    - **Stall**: pipeline delay introduced to restore pipeline flow
- Calculating pipeline CPI
  - **Frequency of stall * stall cycles**
  - Penalties add
  - 1 + stall-freq$_1$*stall-cyc$_1$ + stall-freq$_2$*stall-cyc$_2$ + …
- Correctness/performance/make common case fast (MCCF)
  - Long penalties OK if they happen rarely, e.g., 1 + 0.01 * 10 = 1.1
  - Stalls also have implications for ideal number of pipeline stages

What can go *wrong* in pipelined execution?

Dependences and Hazards

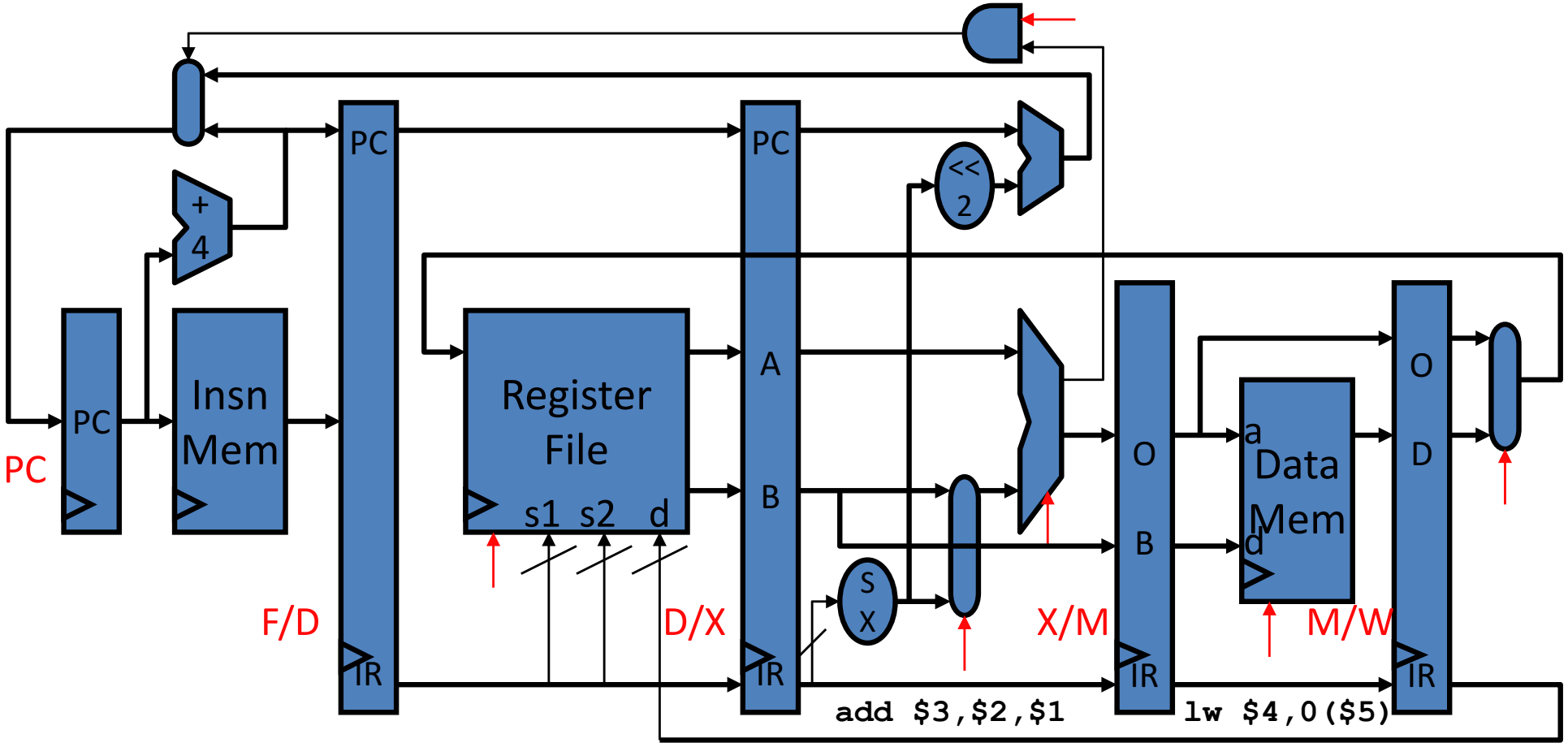
- **Dependence**: relationship between two insns
  - **Data**: two insns use same storage location
  - **Control**: one insn affects whether another executes at all
  - Not a bad thing, programs would be boring without them
  - Enforced by making older insn go before younger one
    - Happens naturally in single-/multi-cycle designs
    - But not in a pipeline
- **Hazard**: dependence & possibility of wrong insn order
  - Effects of wrong insn order cannot be externally visible
    - **Stall**: for order by keeping younger insn in same stage
  - Hazards are a bad thing: stalls reduce performance

Why Does Every Insn Take 5 Cycles?



- Could/should we allow **`add`** to skip M and go to W?

Why Does Every Insn Take 5 Cycles?



- Could/should we allow **`add`** to skip M and go to W? No
  - It wouldn't help: peak fetch still only 1 insn per cycle
  - **Structural hazards**: **`add`** and **`lw`** would use same circuit

Structural Hazards

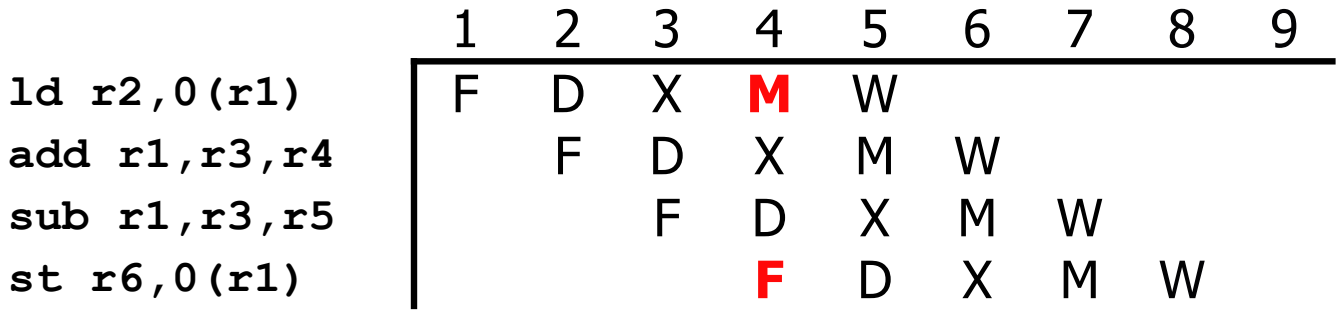
- **Structural hazards**
  - Two insns trying to use same circuit at same time
    - E.g., structural hazard on regfile write port

- How do we solve this issue?

Structural Hazards

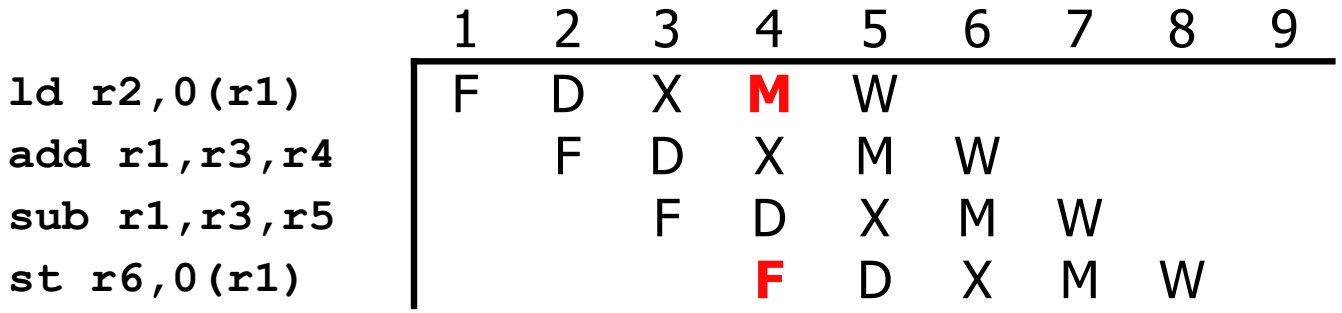
- **Structural hazards**
  - Two insns trying to use same circuit at same time
    - E.g., structural hazard on regfile write port
- **To fix structural hazards**: proper ISA/pipeline design
  - Each insn uses every structure exactly once
  - For at most one cycle
- **Tolerate structure hazards**
  - Add stall logic to stall pipeline when hazards would occur

Example Structural Hazard



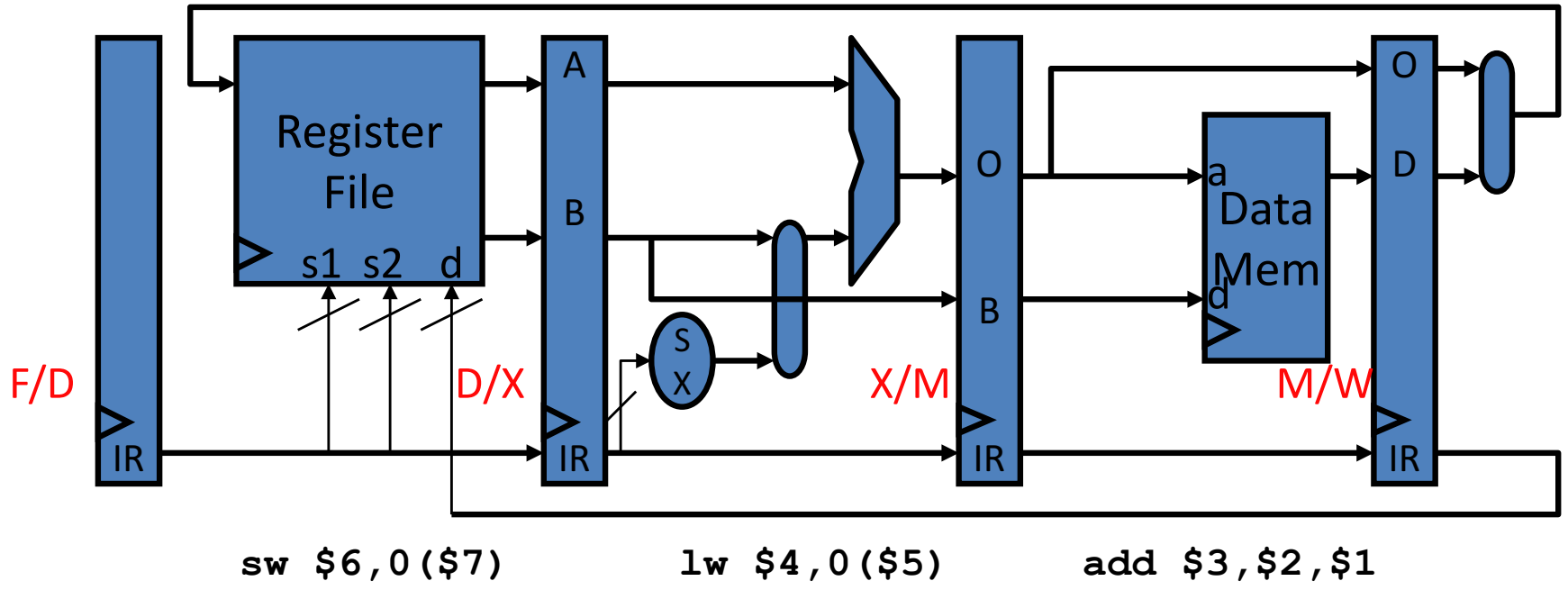
- **Structural hazard**: resource needed twice in one cycle
  - Example: unified instruction & data cache
  - Solutions?

Example Structural Hazard



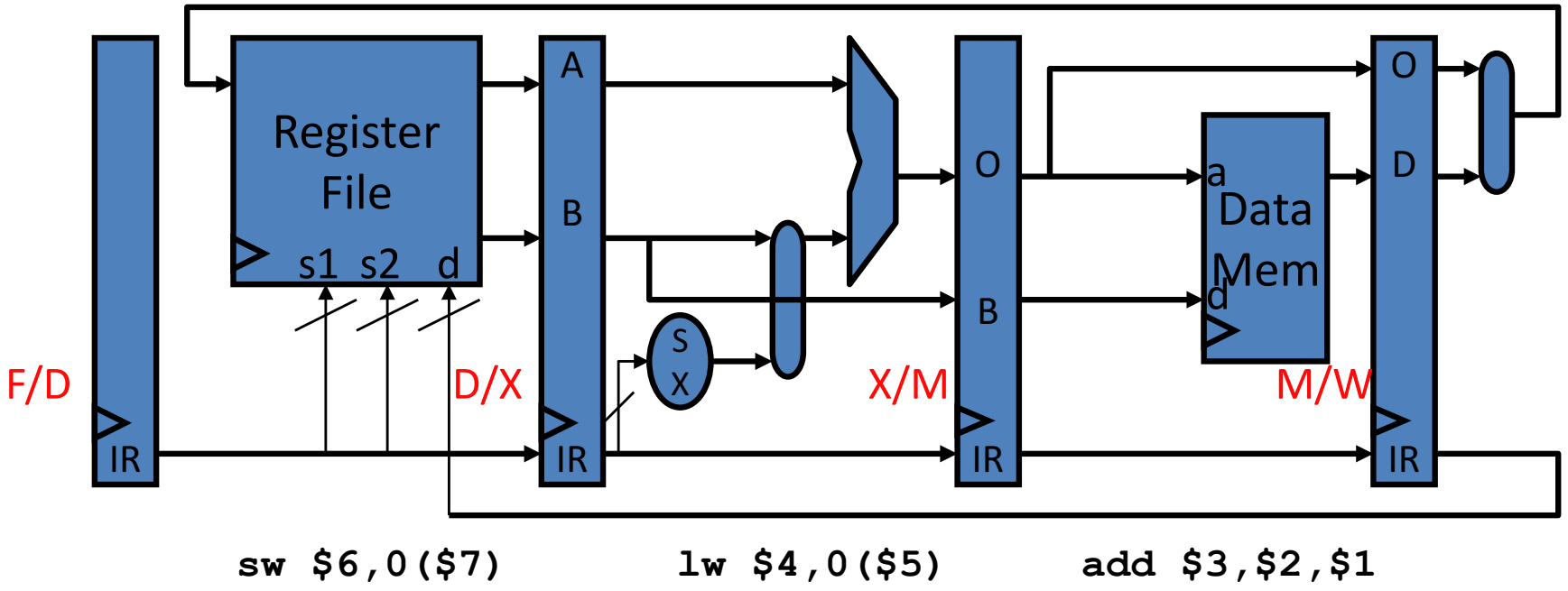
- **Structural hazard**: resource needed twice in one cycle
  - Example: unified instruction & data cache
  - Solutions?
    - Separate instruction/data caches
    - Redesign cache to allow 2 accesses per cycle (slow, expensive)
    - Stall pipeline

Data Hazards



- Let's forget about branches and the control for a while
- The three insn sequence we saw earlier executed fine…
  - *Can you imagine situations when it will **not** be fine?*

Data Hazards



- Let's forget about branches and the control for a while
- The three insn sequence we saw earlier executed fine…
  - But it wasn't a real program
  - Real programs have **data dependences**
    - They pass values via registers and memory

Dependent Operations

- Independent operations

```
add $3,$2,$1
add $6,$5,$4
```

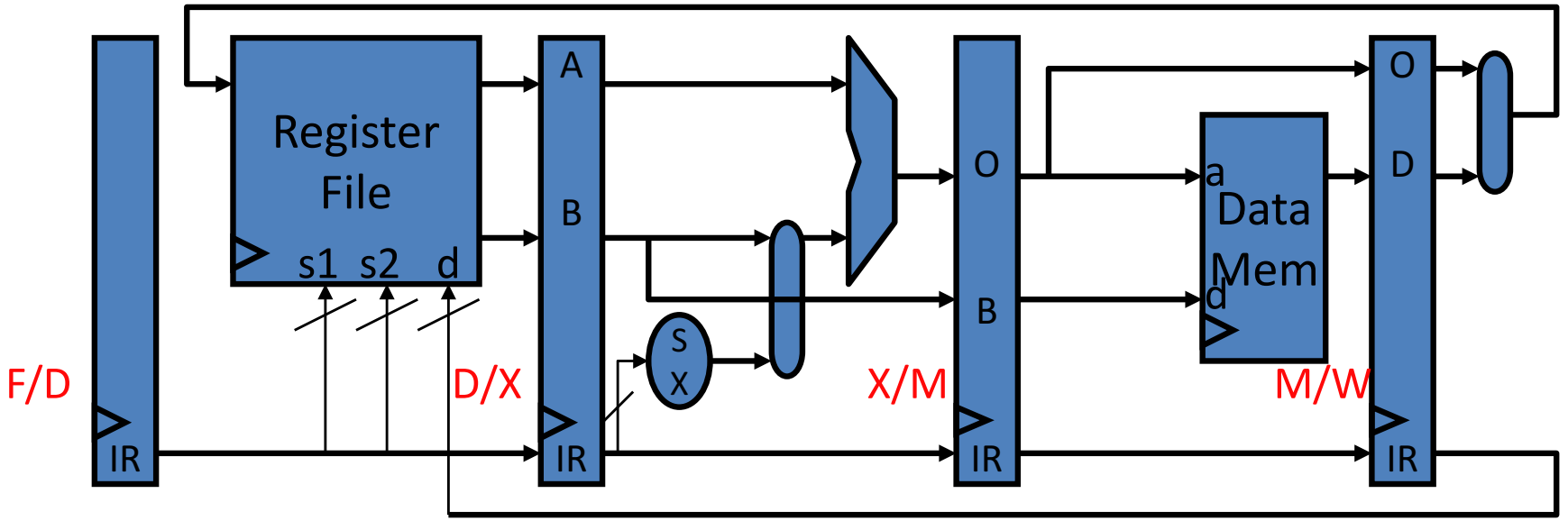
- Would this program execute correctly on a pipeline?

```
add $3,$2,$1
add $6,$5,$3
```

- What about this program?

```
add $3,$2,$1
lw $4,0($3)
addi $6,1,$3
sw $3,0($7)
```

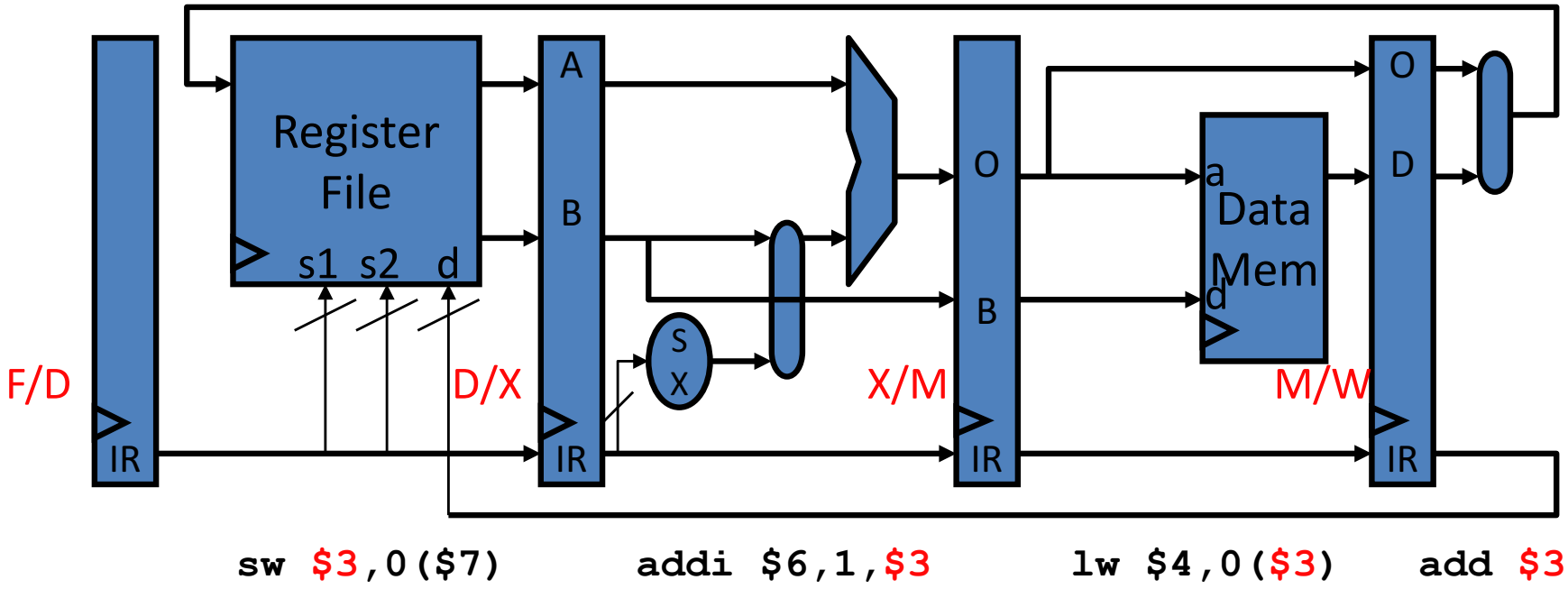
Data Hazards




```
sw $3,0($7)      addi $6,1,$3      lw $4,0($3)      add $3,$2,$1
```

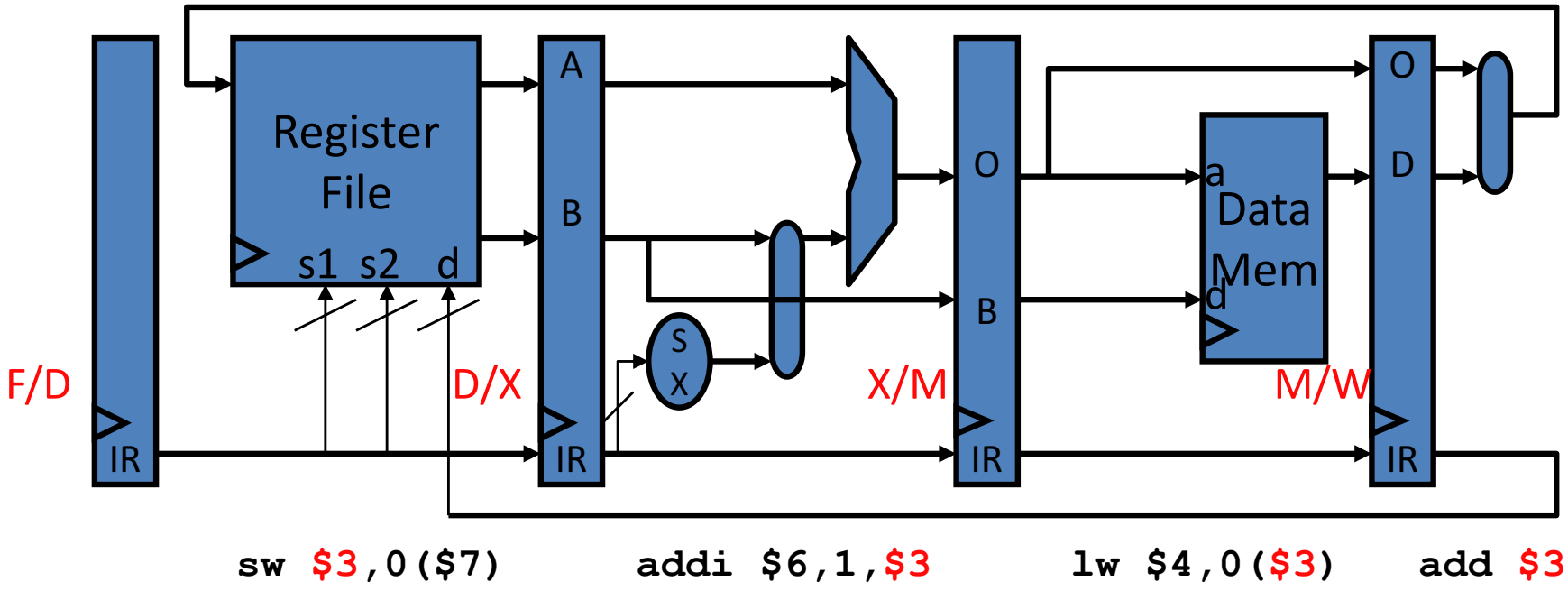
- Would this "program" execute correctly on this pipeline?

Data Hazards



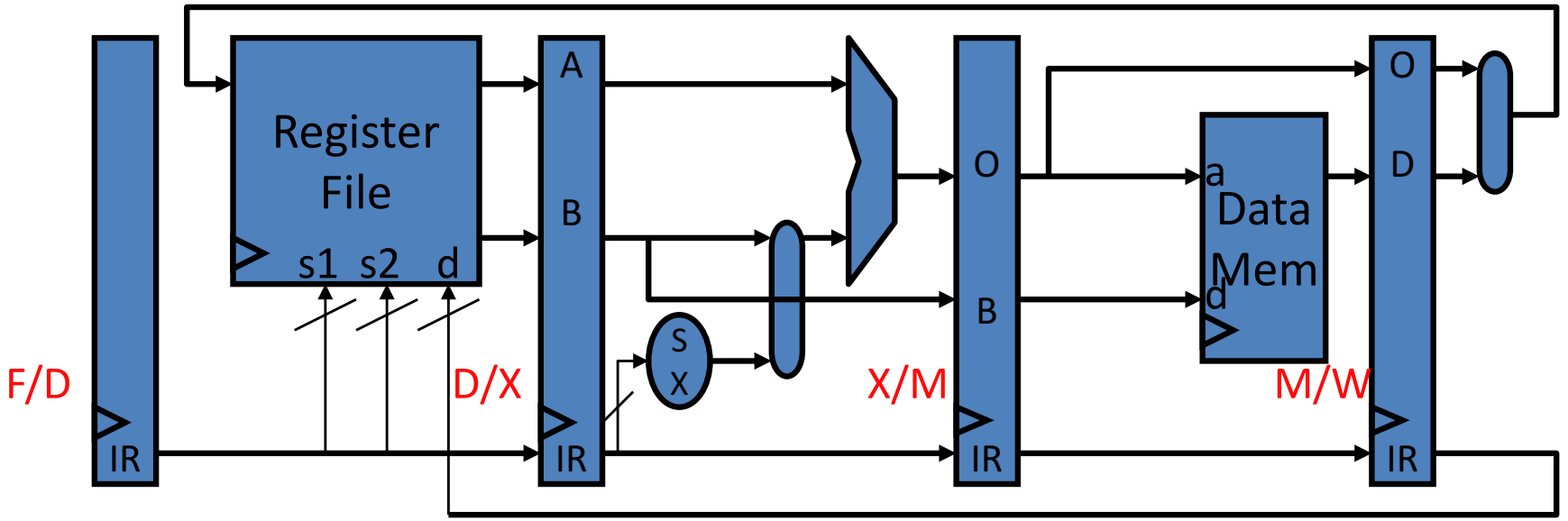
- Would this "program" execute correctly on this pipeline?
  - Which insns would execute with correct inputs?

Data Hazards



- Would this "program" execute correctly on this pipeline?
  - Which insns would execute with correct inputs?
  - **add** is writing its result into **$3** in current cycle
  - **lw** read **$3** 2 cycles ago → got wrong value
  - **addi** read **$3** 1 cycle ago → got wrong value
  - **sw** is reading **$3** this cycle → maybe (depending on regfile design)

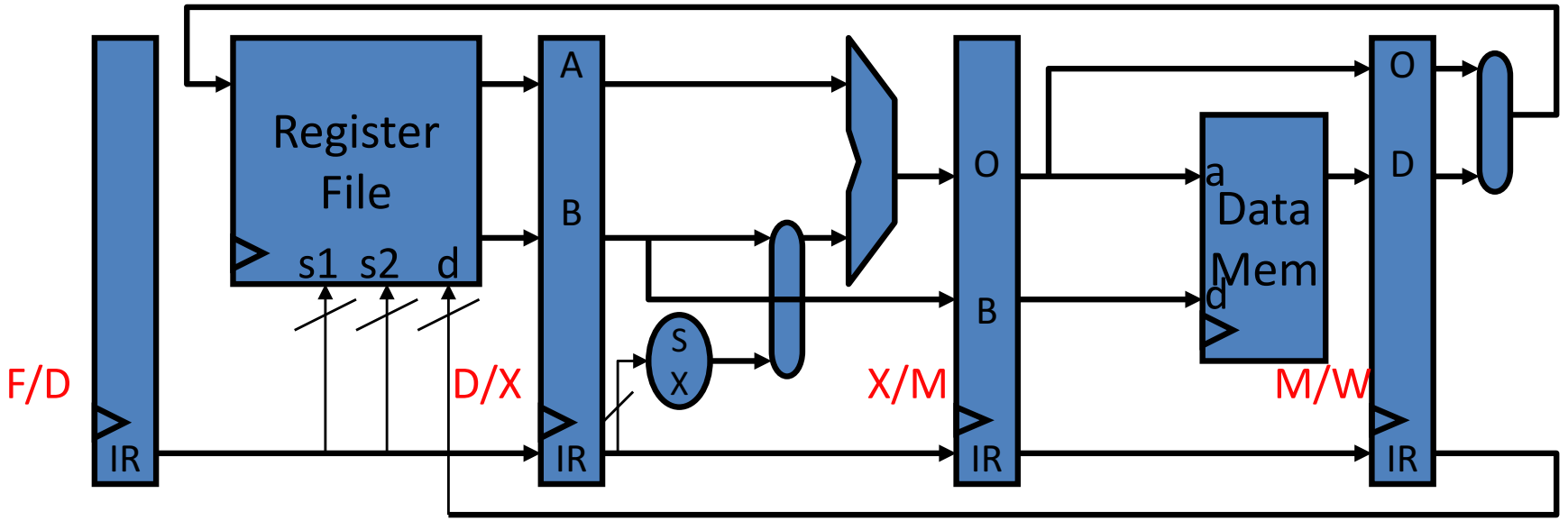
Memory Data Hazards




```
lw $4,0($1)        sw $5,0($1)
```

- What about data hazards through memory, is that a hazard?

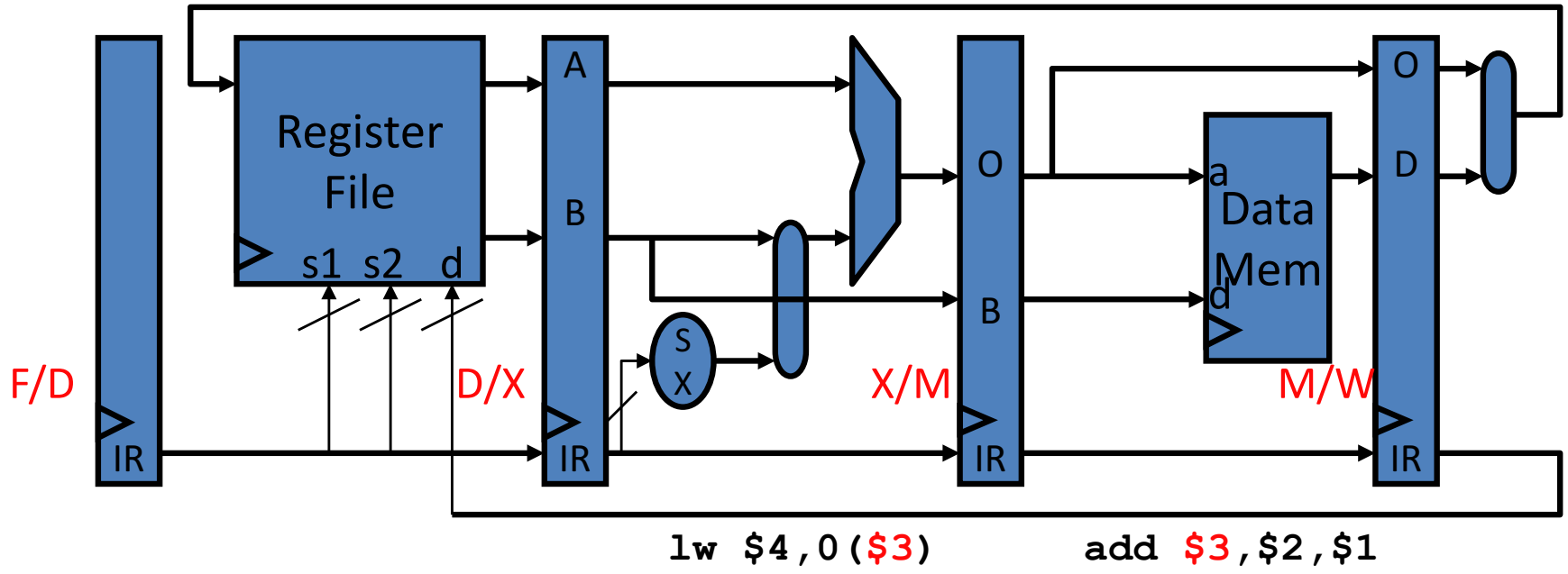
Memory Data Hazards



`lw $4,0($1)` `sw $5,0($1)`

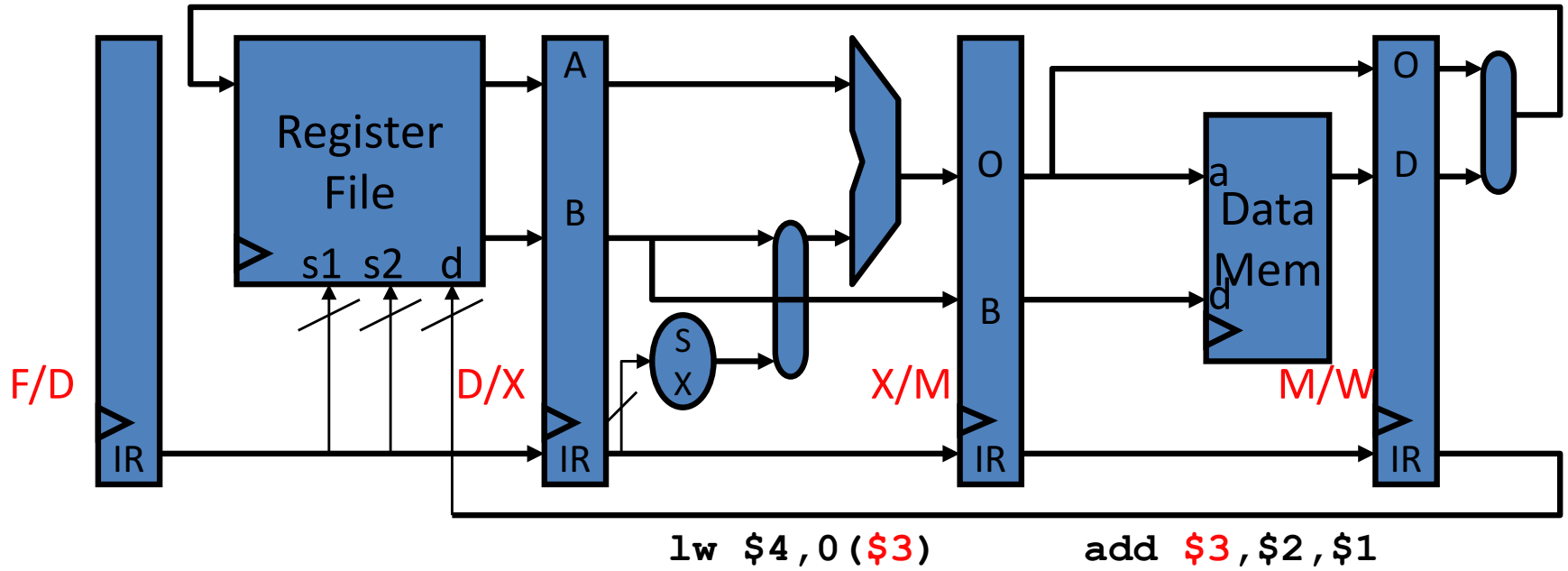
- What about data hazards through memory? No
  - **`lw`** following **`sw`** to same address in next cycle, gets right value
  - Why? Data mem read/write always take place in same stage

Observation!



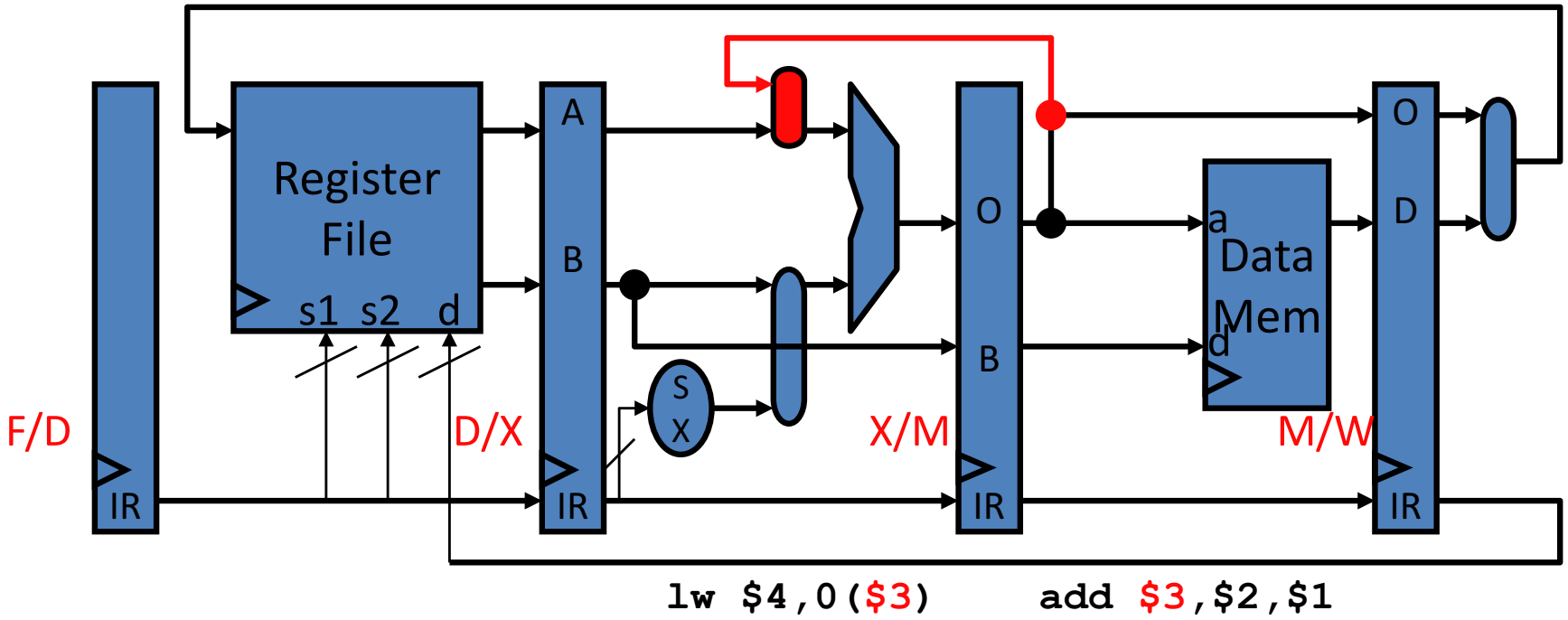
- Technically, this situation is broken
  - `lw $4,0($3)` has already read `$3` from regfile
  - `add $3,$2,$1` hasn't yet written `$3` to regfile
- But …☺

Observation!



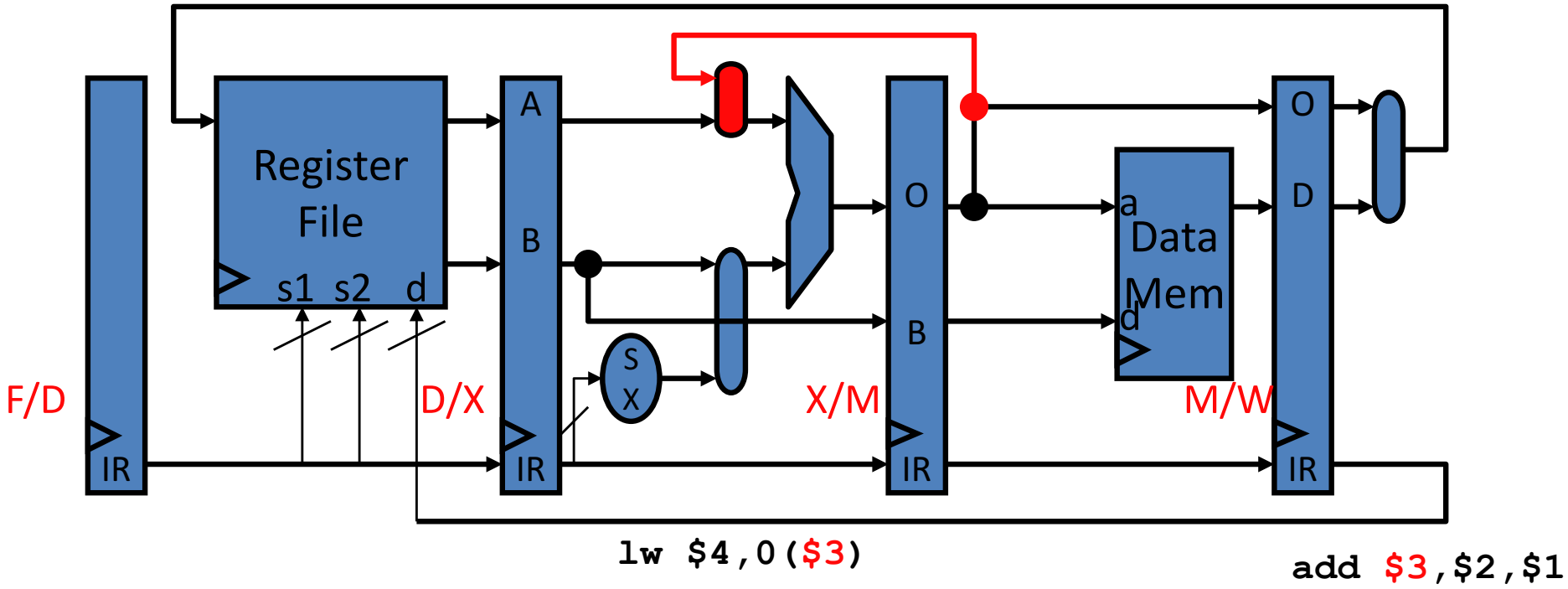
- Technically, this situation is broken
  - **`lw $4,0($3)`** has already read **`$3`** from regfile
  - **`add $3,$2,$1`** hasn't yet written **`$3`** to regfile
- But fundamentally, everything is OK
  - **`lw $4,0($3)`** hasn't actually used **`$3`** yet
  - **`add $3,$2,$1`** has already computed **`$3`**
- *How can we take advantage of this?*

Reducing Data Hazards: Bypassing



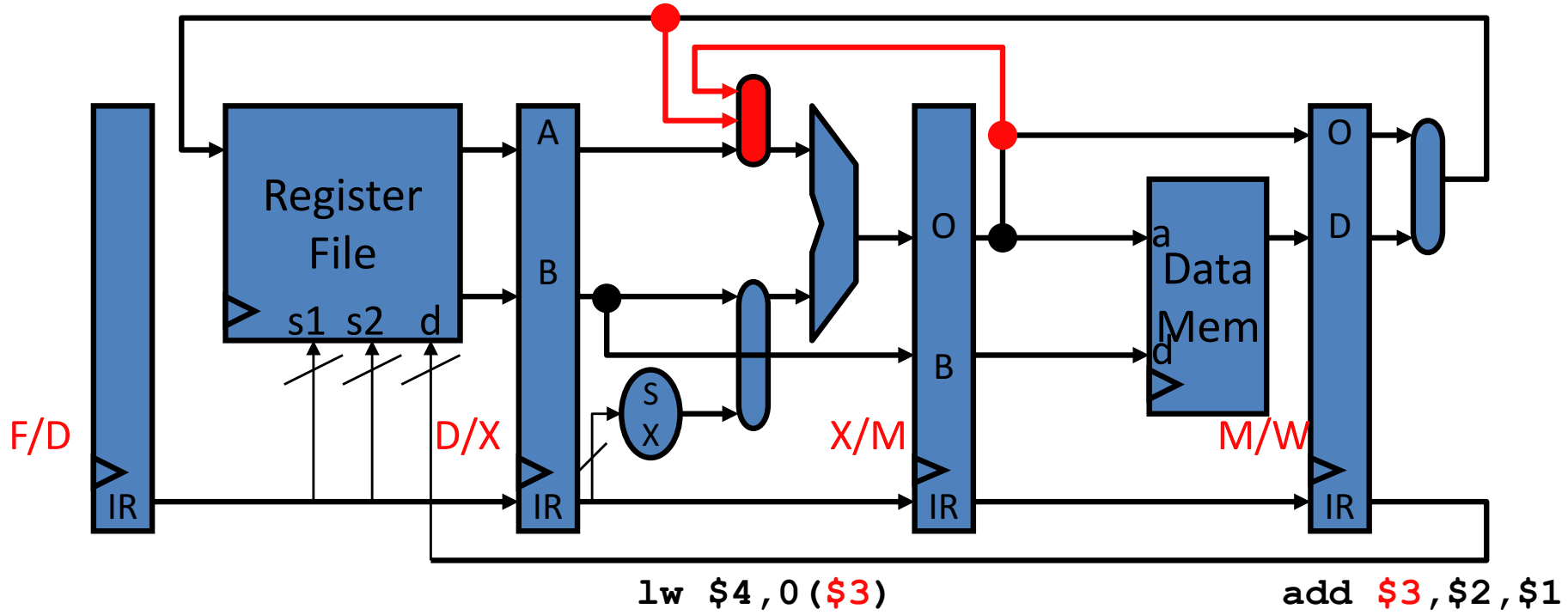
- **Bypassing**
  - Reading a value from an intermediate (μarchitectural) source
  - Not waiting until it is available from primary source
  - Here, we are bypassing the register file
  - Also called **forwarding**

What about this situation?



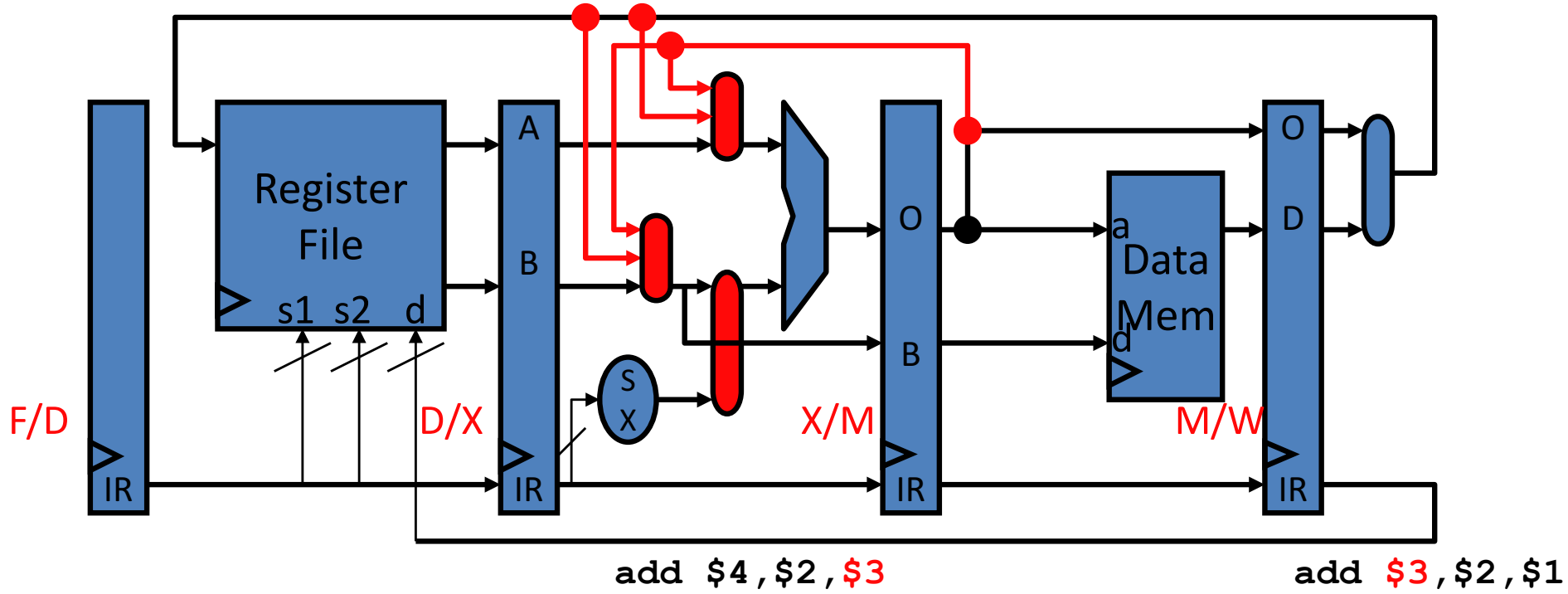
- Would the bypassing above work?

WX Bypassing



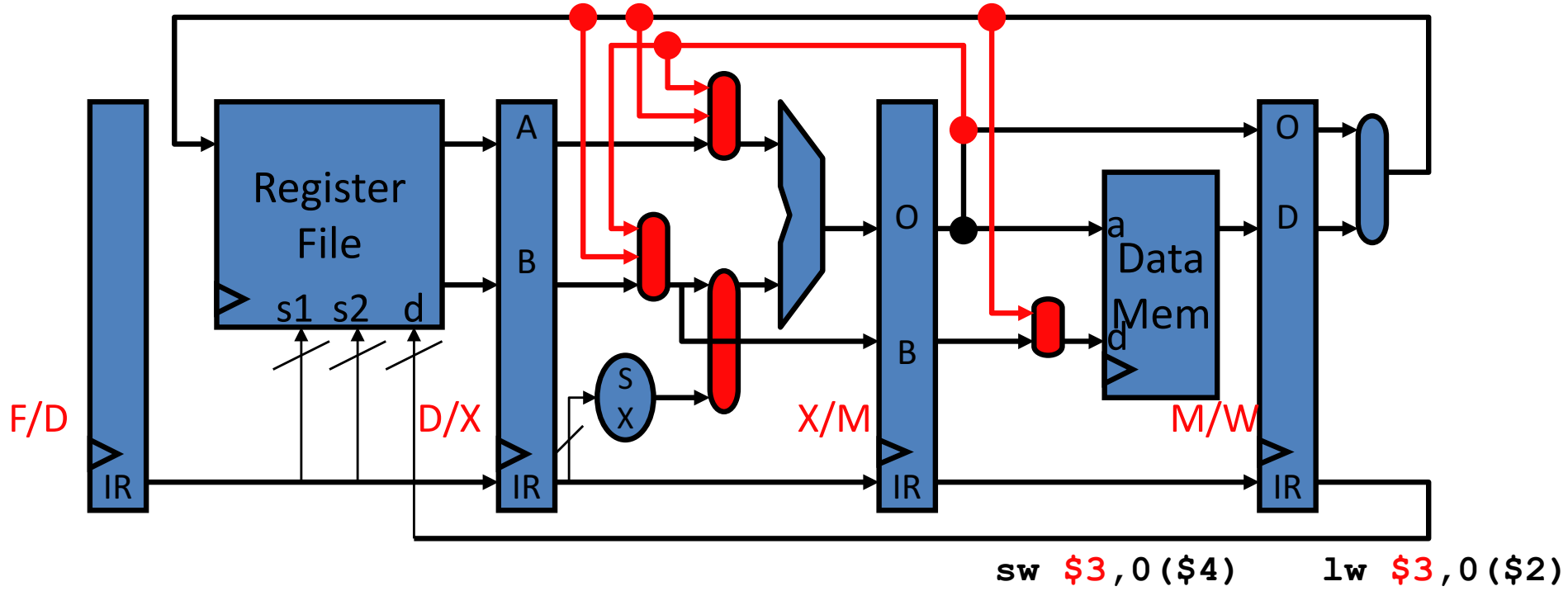
- Add another bypass path and MUX input
- First one was an **MX** bypass
- This one is a **WX** bypass

ALUinB Bypassing



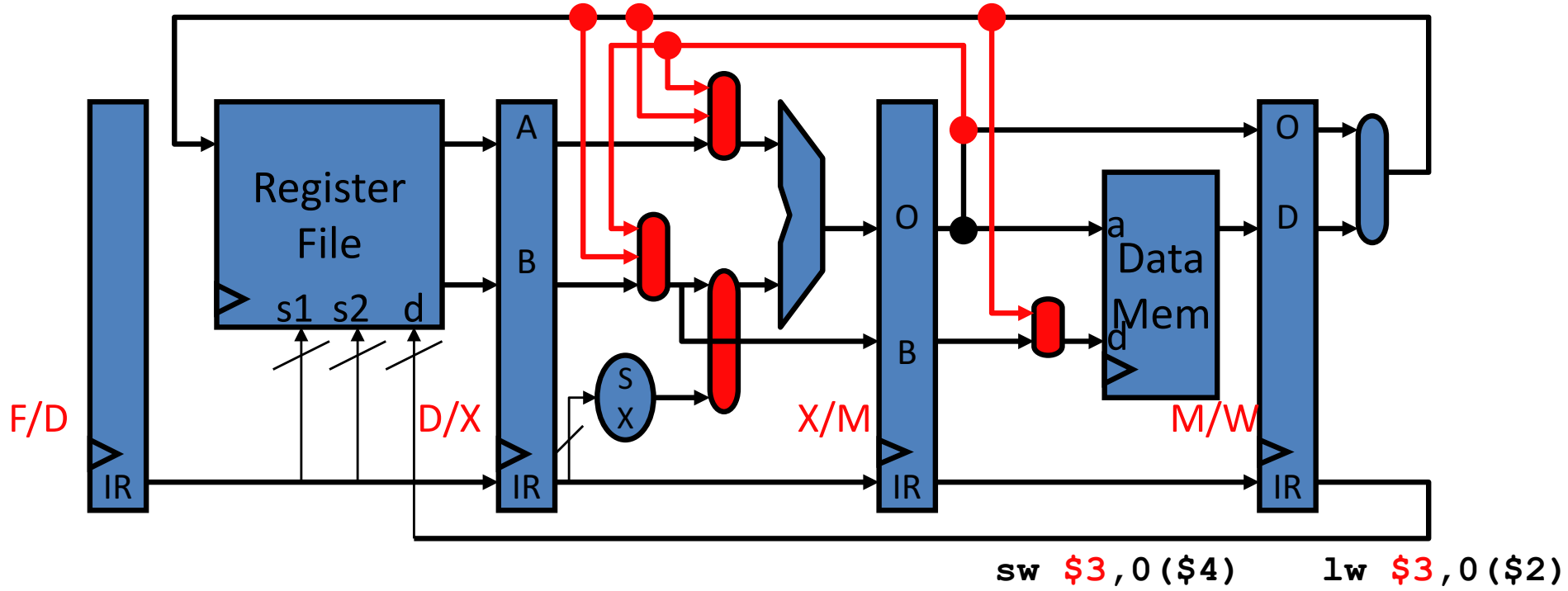
- Can also bypass to ALU input B

WM Bypassing?



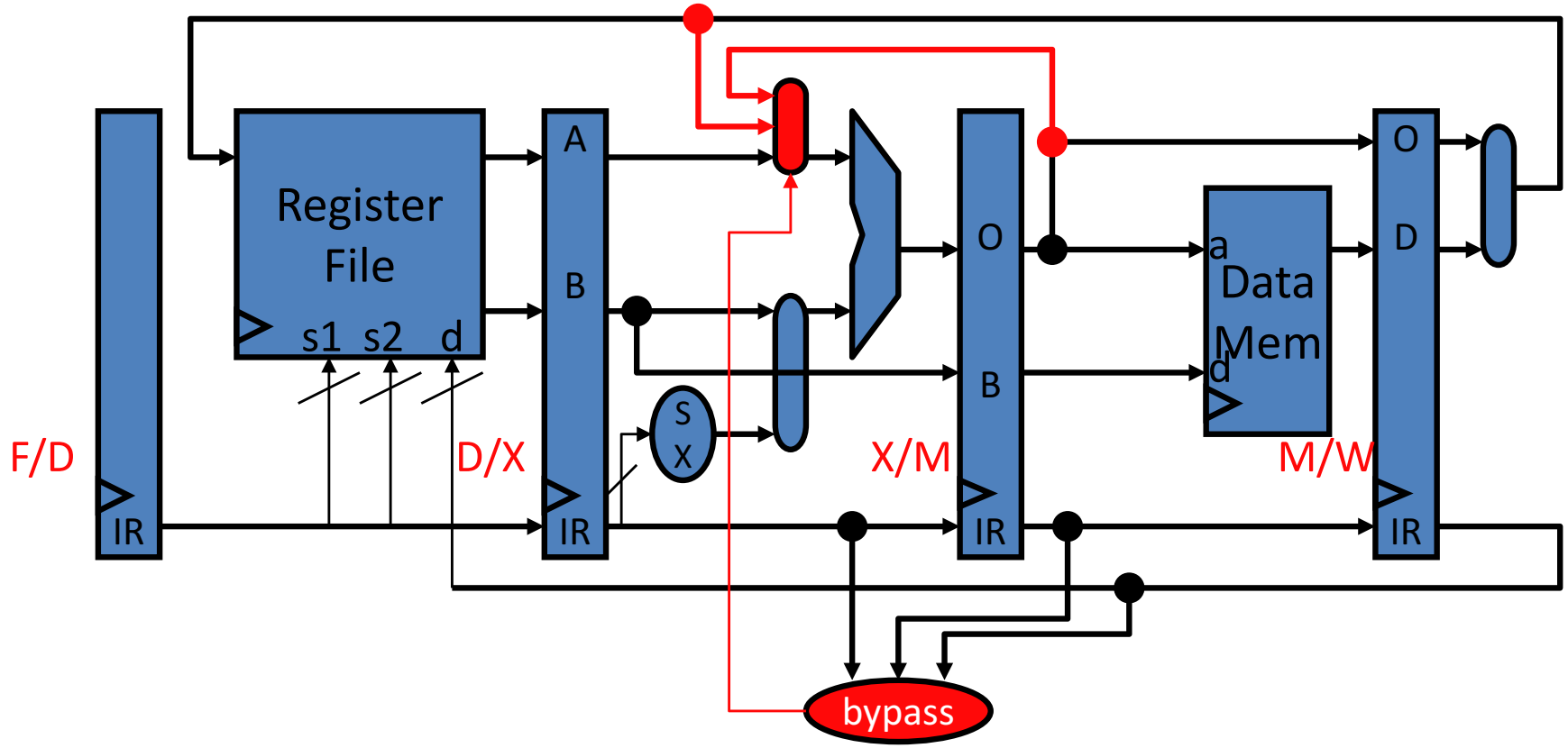
- Does WM bypassing make sense?

WM Bypassing?



- Does WM bypassing make sense?
  - Not to the address input (why not?)
  - But to the store data input, yes

Bypass Logic



- Each MUX has its own, here it is for MUX ALUinA

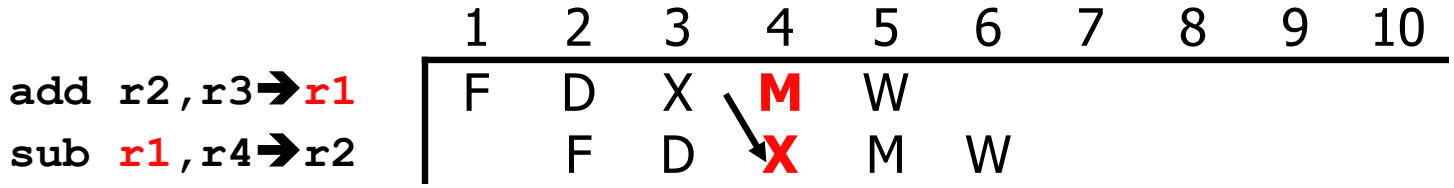
  (D/X.IR.RegSource1 == X/M.IR.RegDest) => 0

  (D/X.IR.RegSource1 == M/W.IR.RegDest) => 1

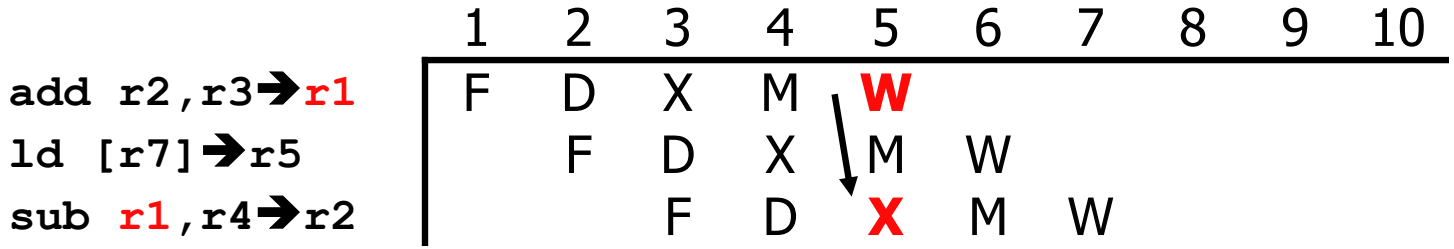
  Else => 2

Pipeline Diagrams with Bypassing

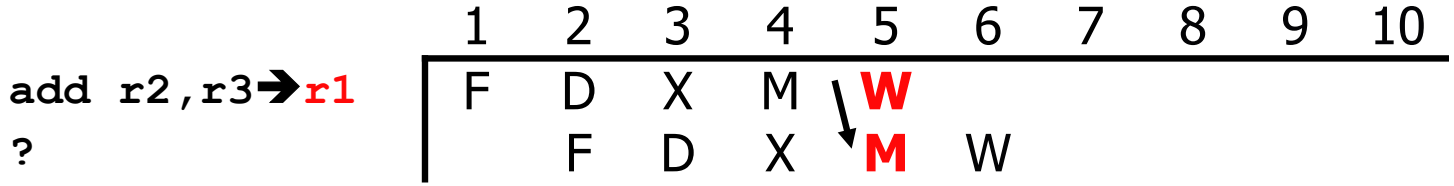
- If bypass exists, "from"/"to" stages execute in same cycle
  - Example: full bypassing, use MX bypass



  - Example: full bypassing, use WX bypass

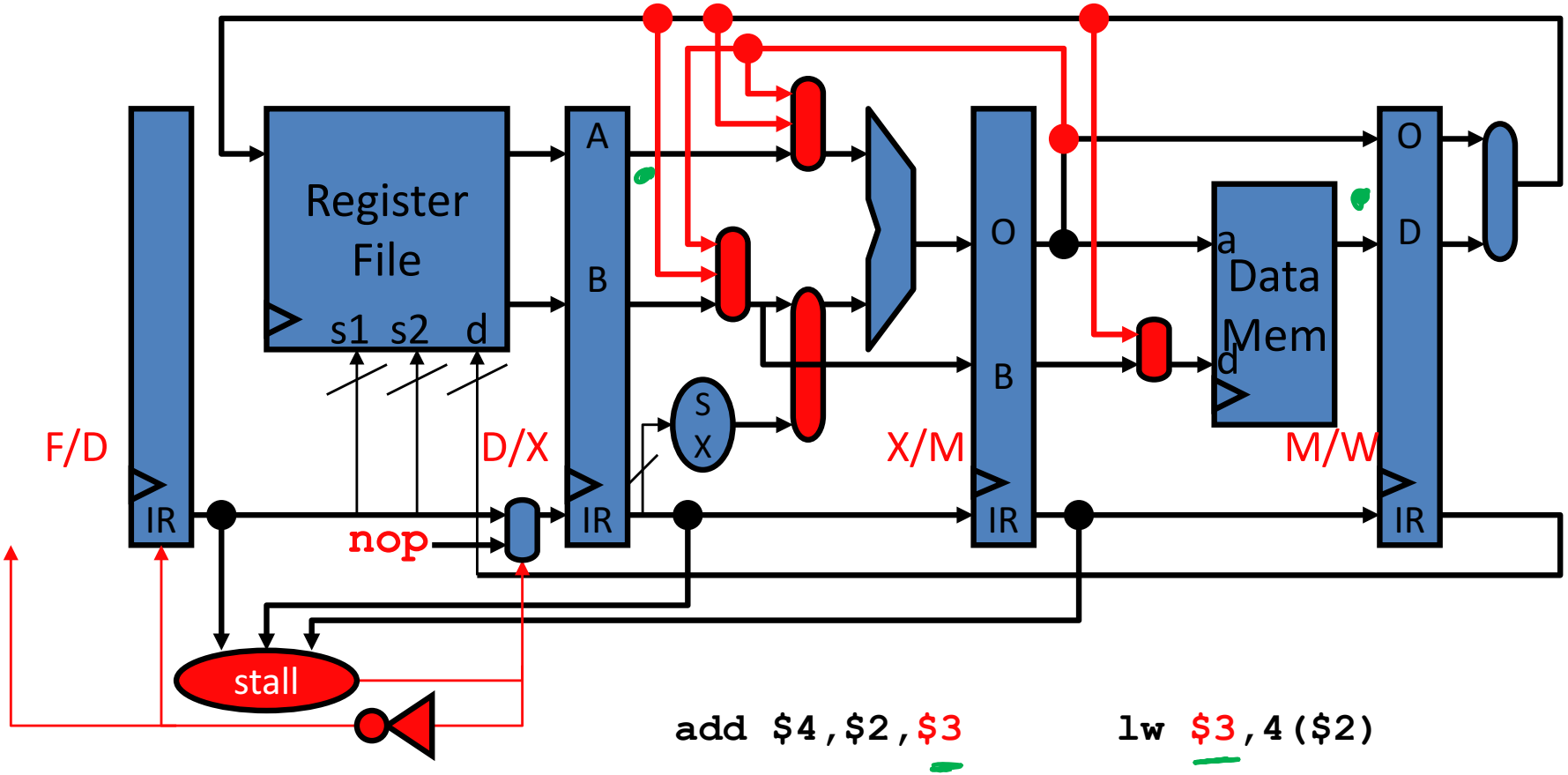


  - Example: WM bypass

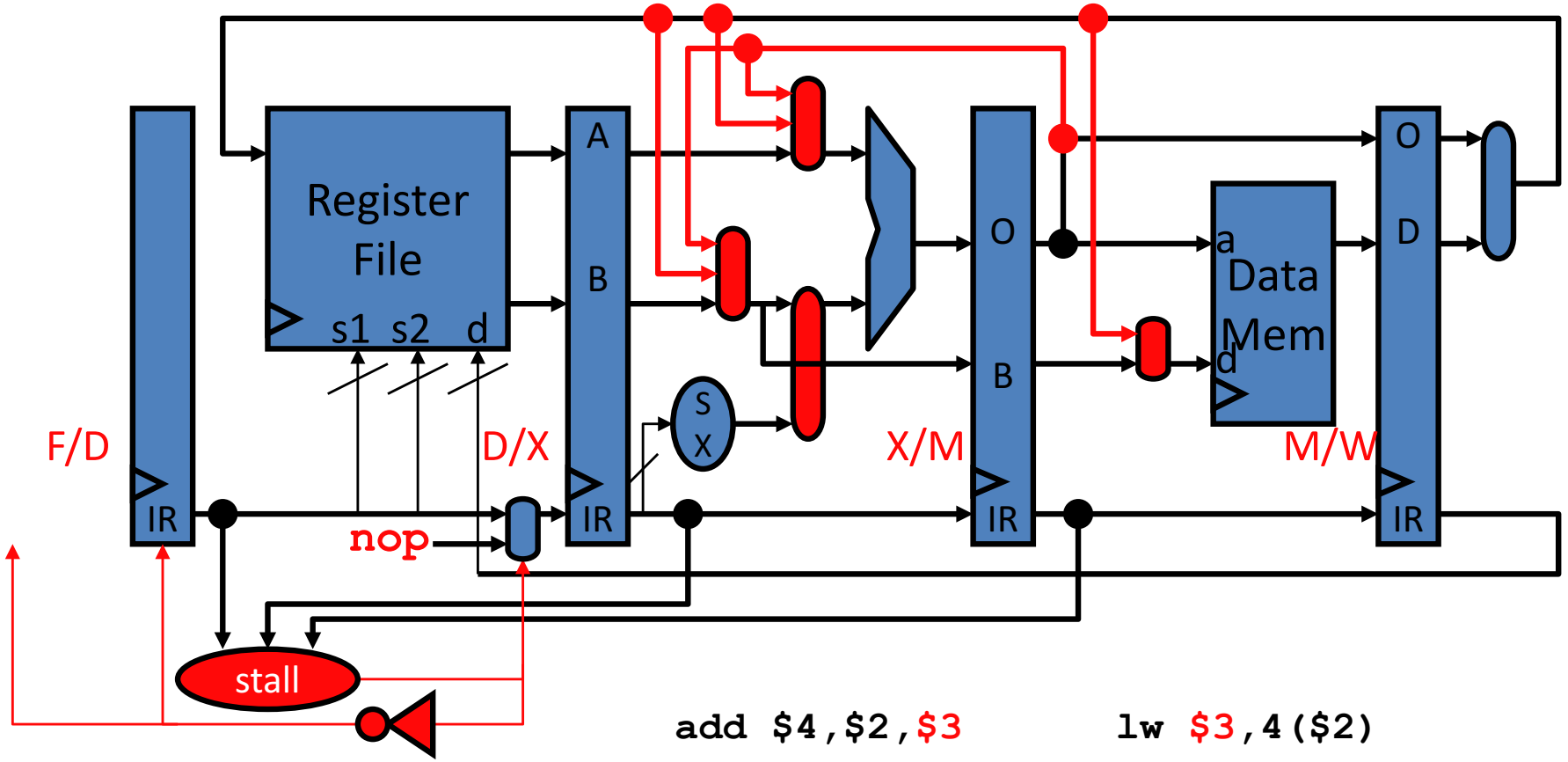


    - Can you think of a code example that uses the WM bypass?

Have We Prevented All Data Hazards?

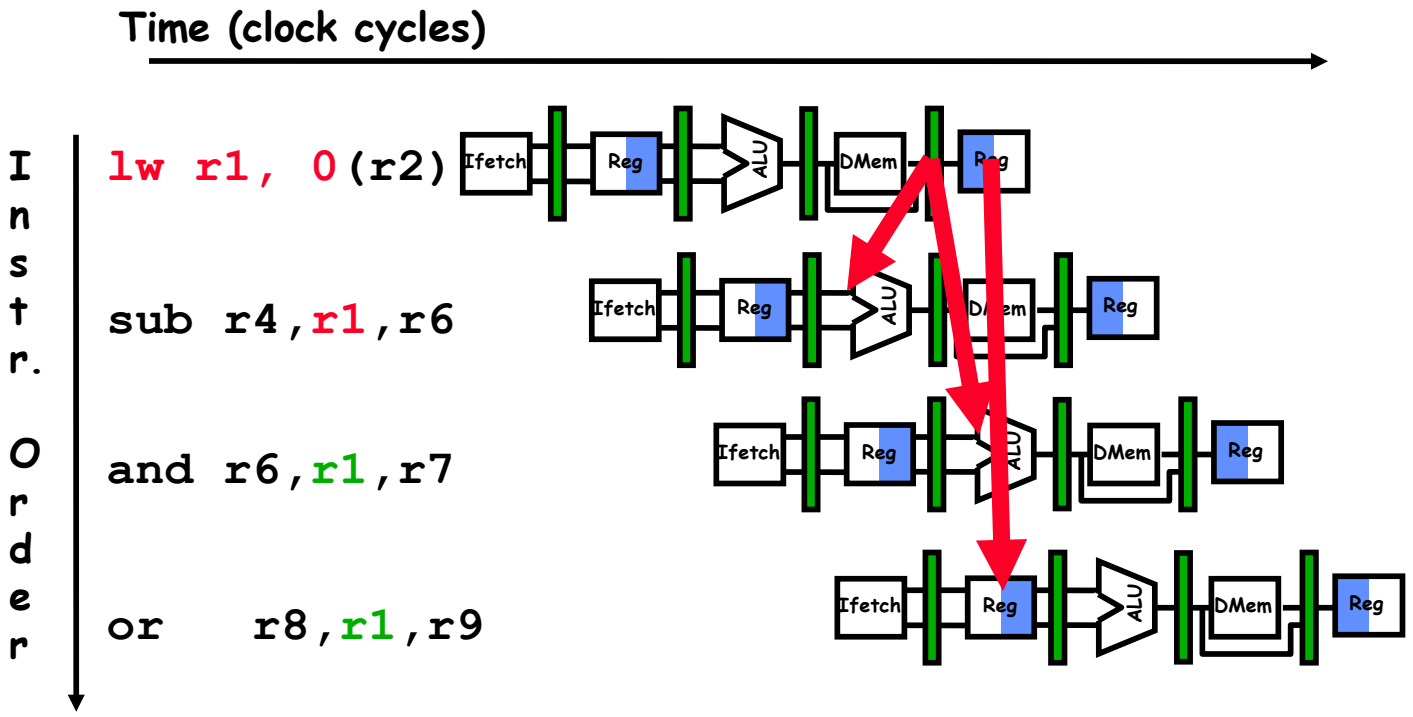


Have We Prevented All Data Hazards?

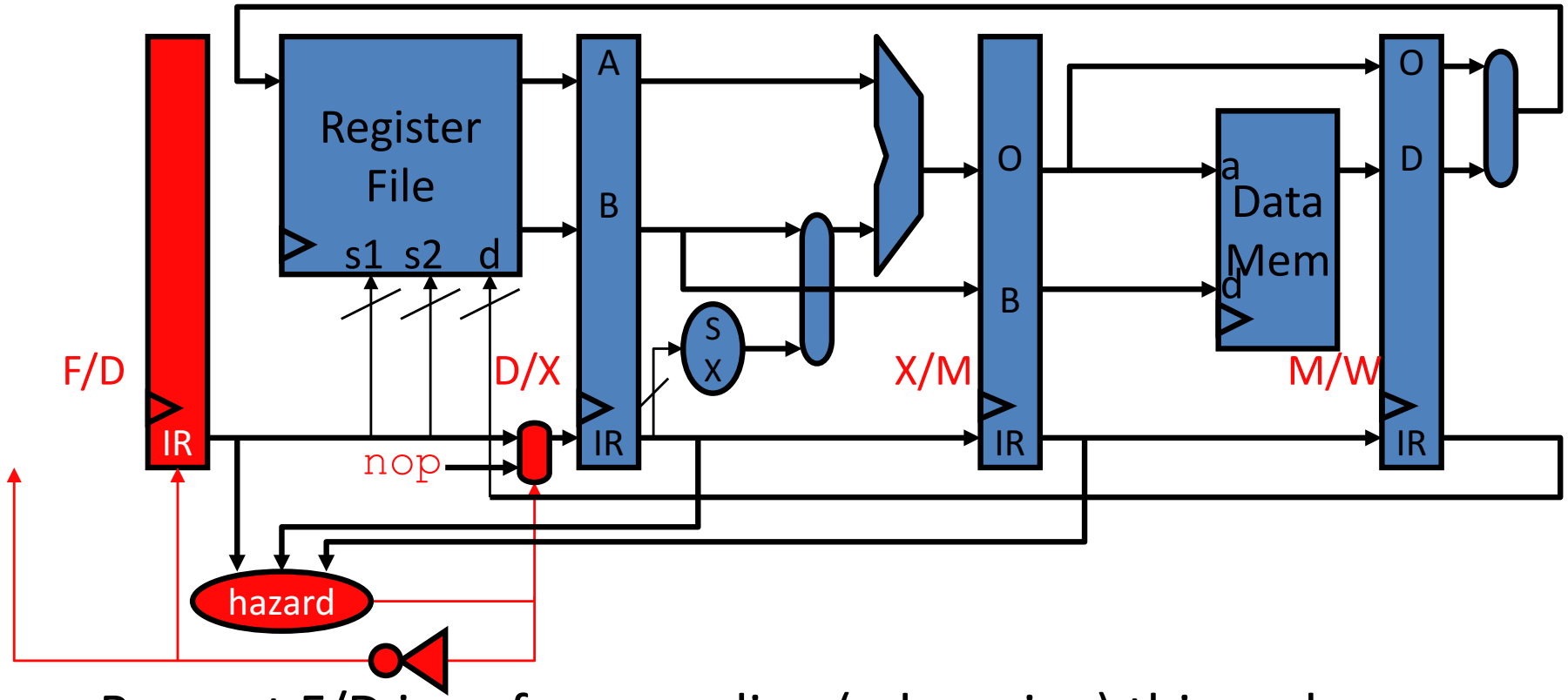


- No. Consider a "load" followed by a dependent "add" insn
- Bypassing alone isn't sufficient
- Solution? Detect this, and then stall the "add" by one cycle

Loads followed by use

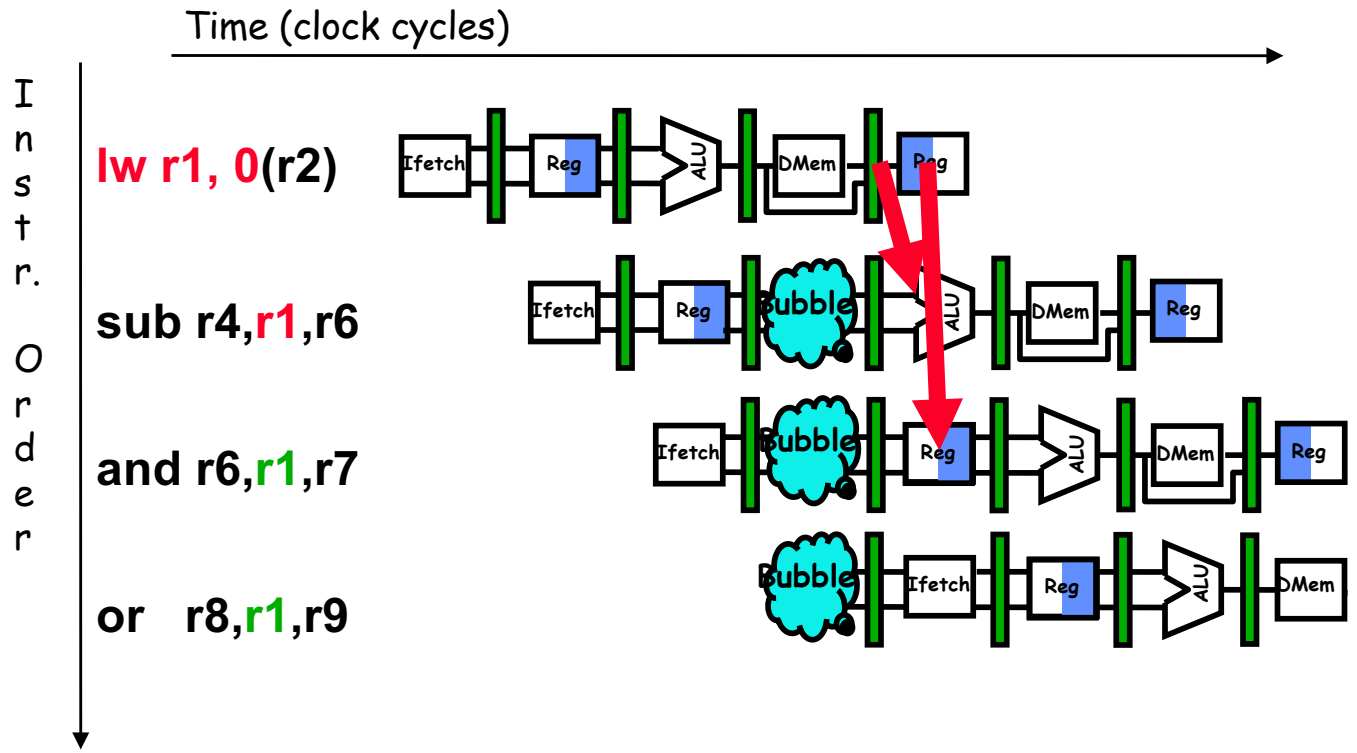


Stalling to Avoid Data Hazards



- Prevent F/D insn from reading (advancing) this cycle
  - Write `nop` into D/X.IR (effectively, insert `nop` in hardware)
  - Also reset (clear) the datapath control signals
  - Disable F/D latch and PC write enables (why?)
- Re-evaluate situation next cycle

Loads followed by use stall



The ARM1136JF-S processor features:

- an integer unit with integral EmbeddedICE-RT logic
- an eight-stage pipeline
- branch prediction with return stack
- low interrupt latency
- external coprocessor interface and coprocessors 14 and 15
- Instruction and Data *Memory Management Units* (MMUs), managed using MicroTLB structures backed by a unified Main TLB
- Instruction and data caches, including a non-blocking data cache with *Hit-Under-Miss* (HUM)
- the caches are virtually indexed and physically addressed
- 64-bit interface to both caches
- a bypassable write buffer
- level one *Tightly-Coupled Memory* (TCM) that can be used as a local RAM with DMA, or as SmartCache
- high-speed *Advanced Microprocessor Bus Architecture* (AMBA) level two interfaces supporting prioritized multiprocessor implementations
- *Vector Floating-Point* (VFP) coprocessor support
- external coprocessor support
- trace support
- JTAG-based debug.