ECE/CS 250
Computer Architecture

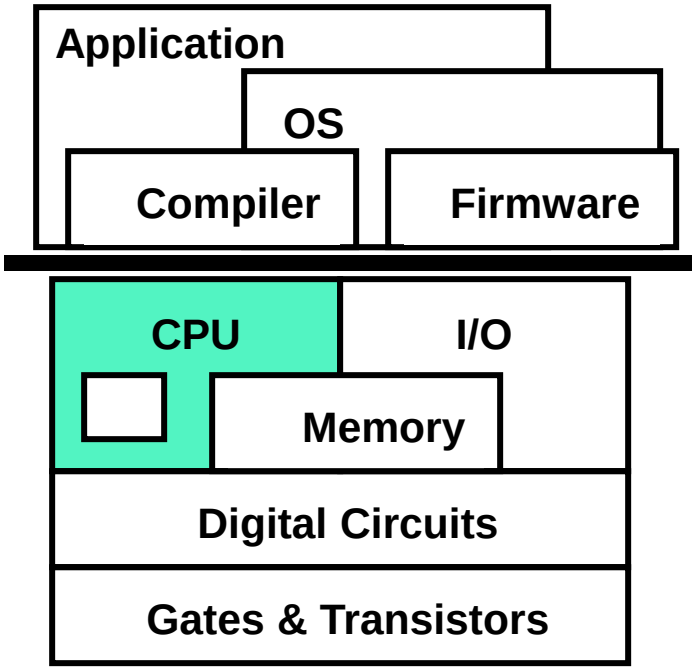
Fall 2022

Pipelining

Tyler Bletsch
Duke University

Includes material adapted from Dan Sorin (Duke) and Amir Roth (Penn).

This Unit: Pipelining



- Basic Pipelining
  - Pipeline control
- Data Hazards
  - Software interlocks and scheduling
  - Hardware interlocks and stalling
  - Bypassing
- Control Hazards
  - Fast and delayed branches
  - Branch prediction
- Multi-cycle operations
- Exceptions

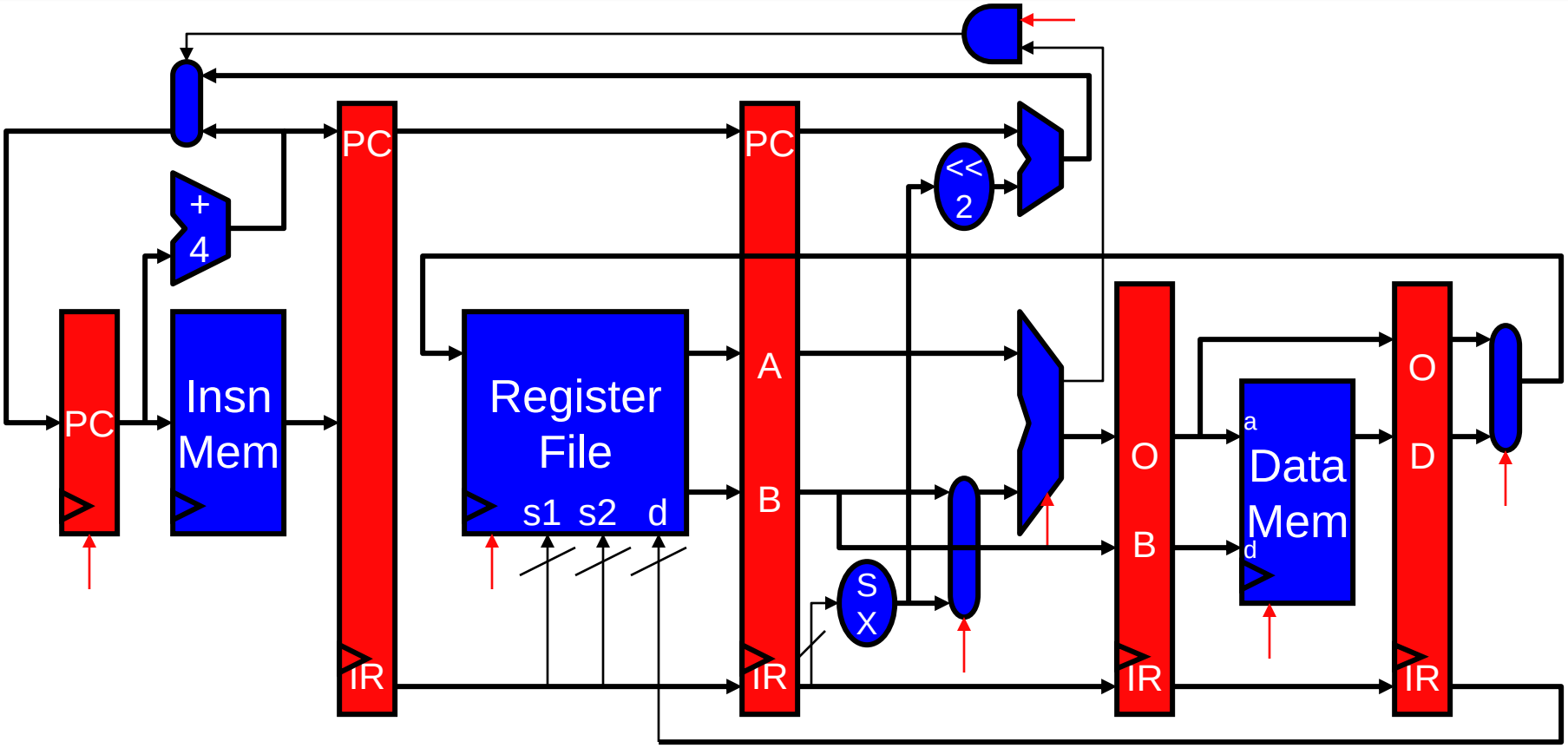
Readings

- P+H
  - Chapter 4: Section 4.5-end of Chapter 4

Pipelining

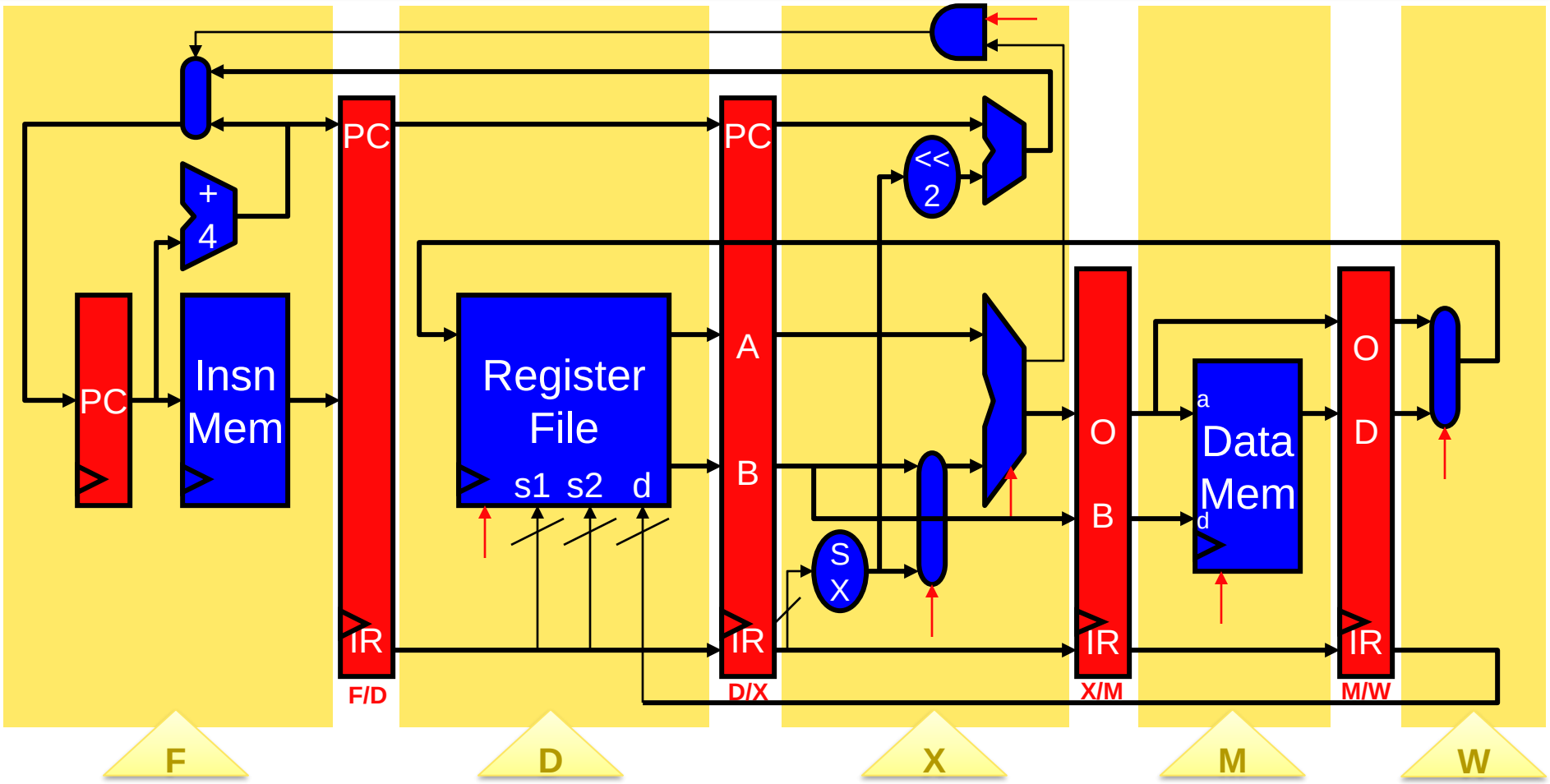
- Important performance technique
  - **Improves insn throughput (rather than insn latency)**
- Laundry / SubWay analogy
- Basic idea: divide instruction's "work" into stages
  - When insn advances from stage 1 to 2
  - Allow next insn to enter stage 1
  - Etc.
- Key idea: each instruction does same amount of work as before
  - + **But insns enter and leave at a much faster rate**

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, they share a single PC?
  - Notice, PC not re-latched after ALU stage (why not?)

Pipeline Terminology

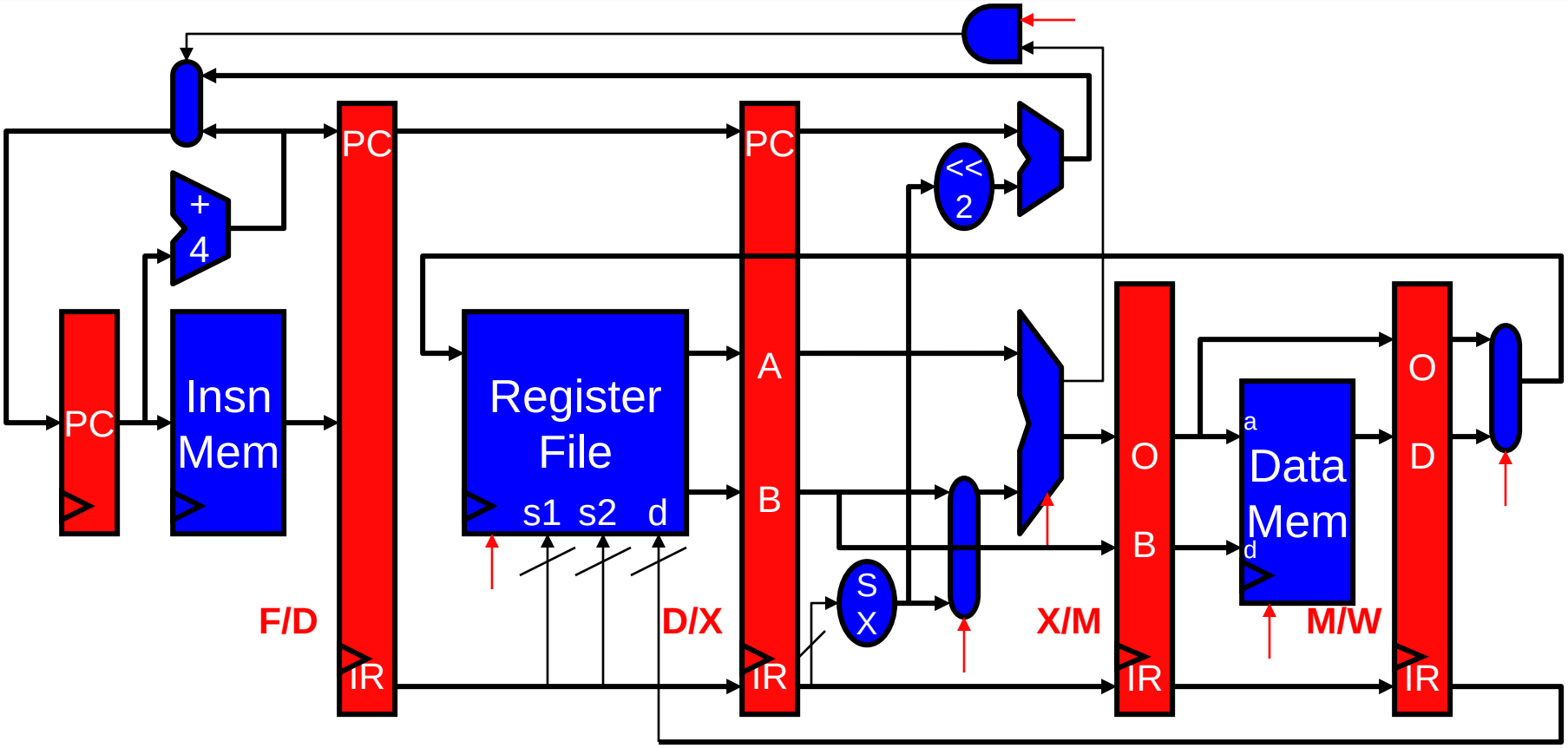


- Five stage: **F**etch, **D**ecode, e**X**ecute, **M**emory, **W**riteback
  - Latches (pipeline registers) named by stages they separate
    - **PC**, **F/D**, **D/X**, **X/M**, **M/W**

Aside: Not All Pipelines Have 5 Stages

- H&P textbook uses well-known 5-stage pipe != all pipes have 5 stages
- Some examples
  - OpenRISC 1200: 4 stages
  - Sun UltraSPARC T1/T2 (Niagara/Niagara2): 6/8 stages
  - AMD Athlon: 10 stages
  - Pentium 4: 20 stages
- ICQ: why does Pentium 4 have so many stages?
- ICQ: how can you possibly break "work" to do single insn into that many stages?
- Moral of the story: in ECE/CS 250, we focus on H&P 5-stage pipe, but don't forget that this is just one example

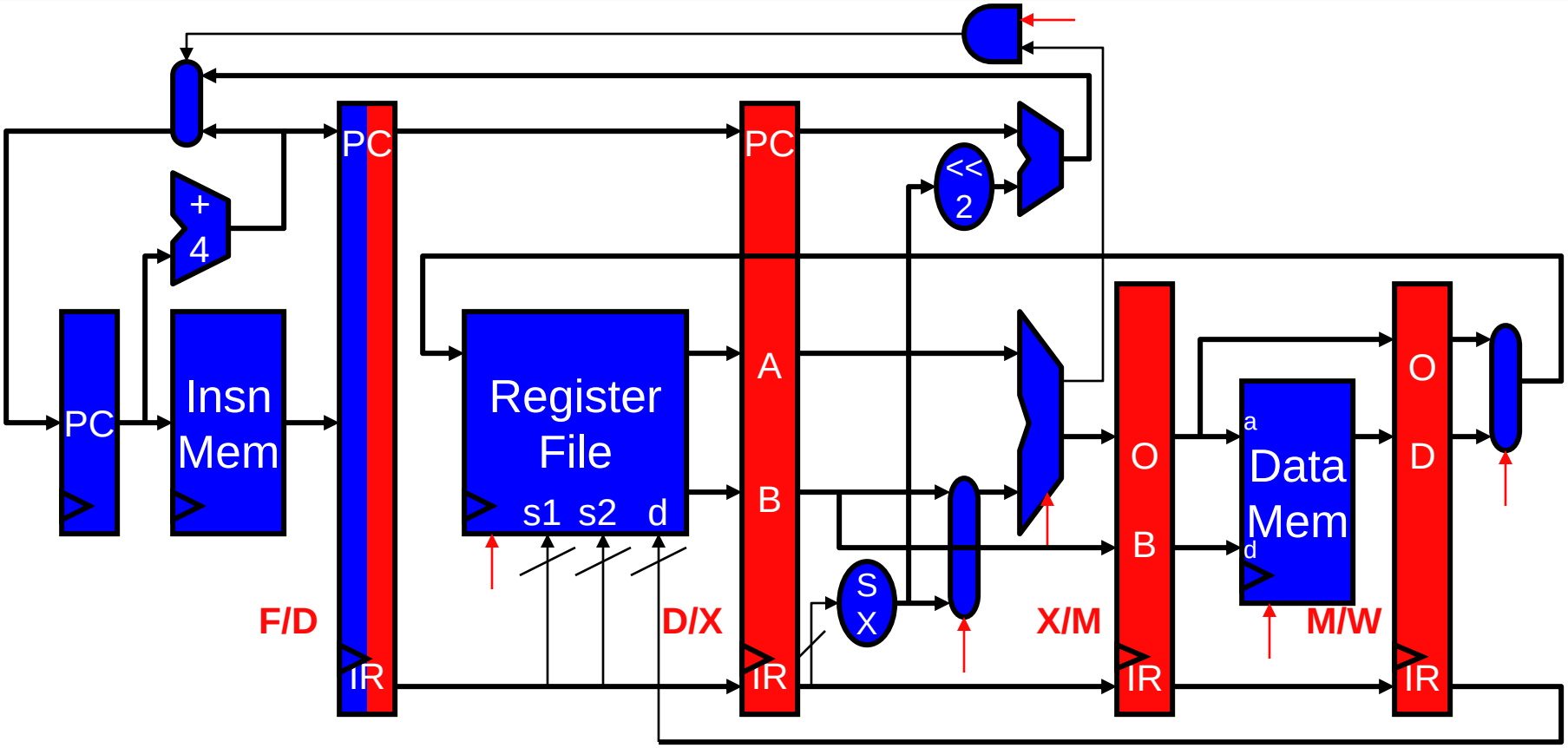
Pipeline Example: Cycle 1




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

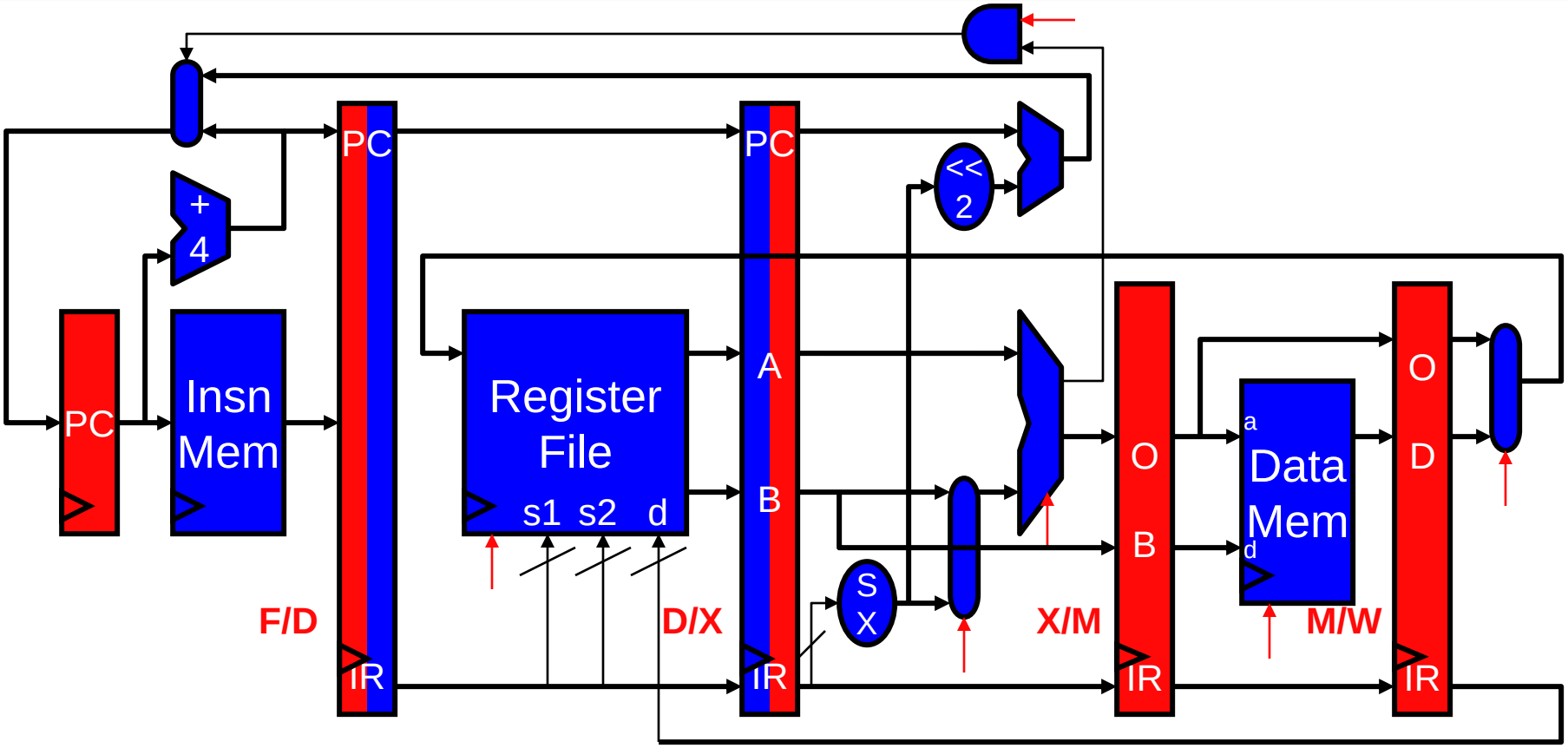
- 3 instructions

Pipeline Example: Cycle 2



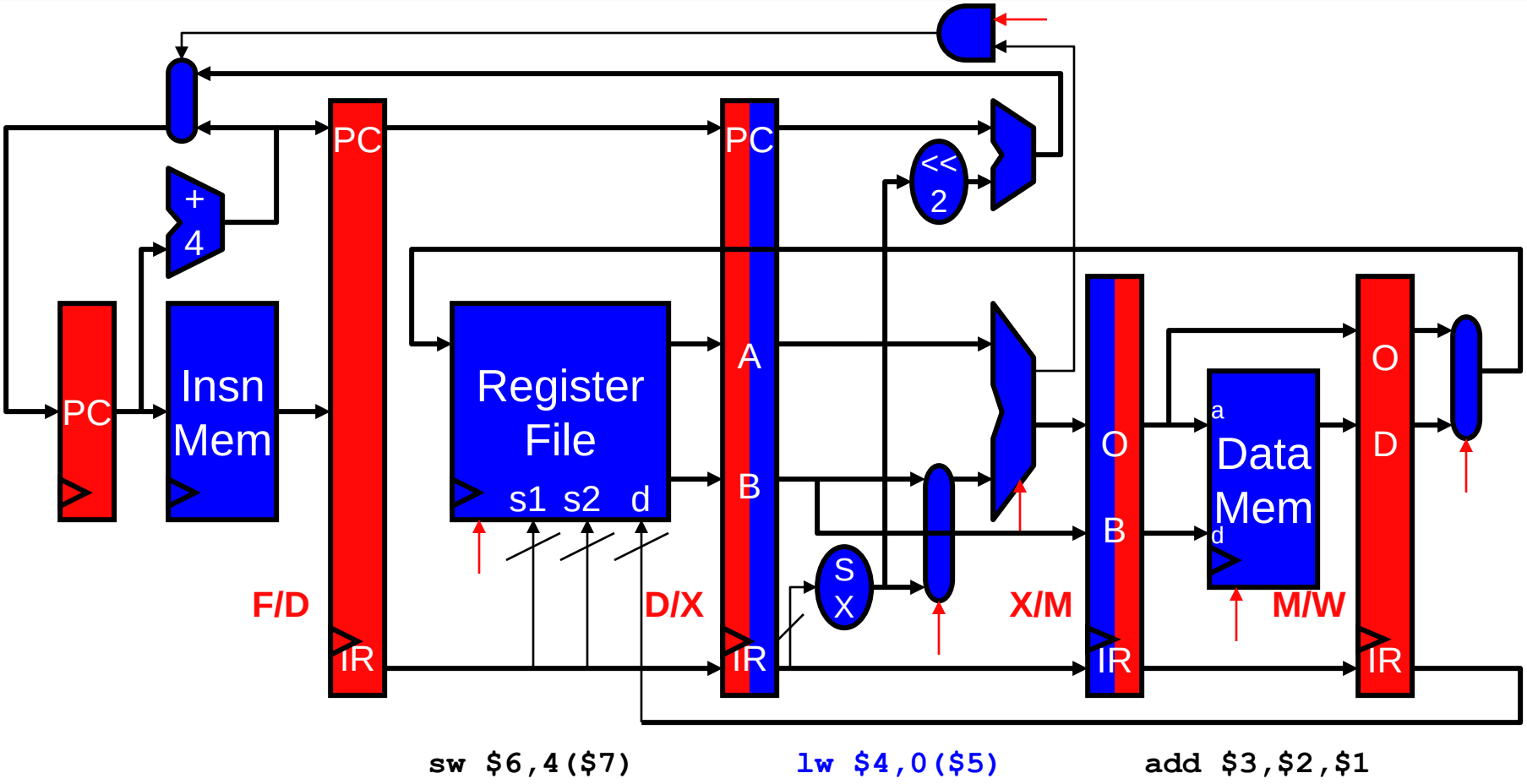
`lw $4,0($5)` `add $3,$2,$1`

Pipeline Example: Cycle 3



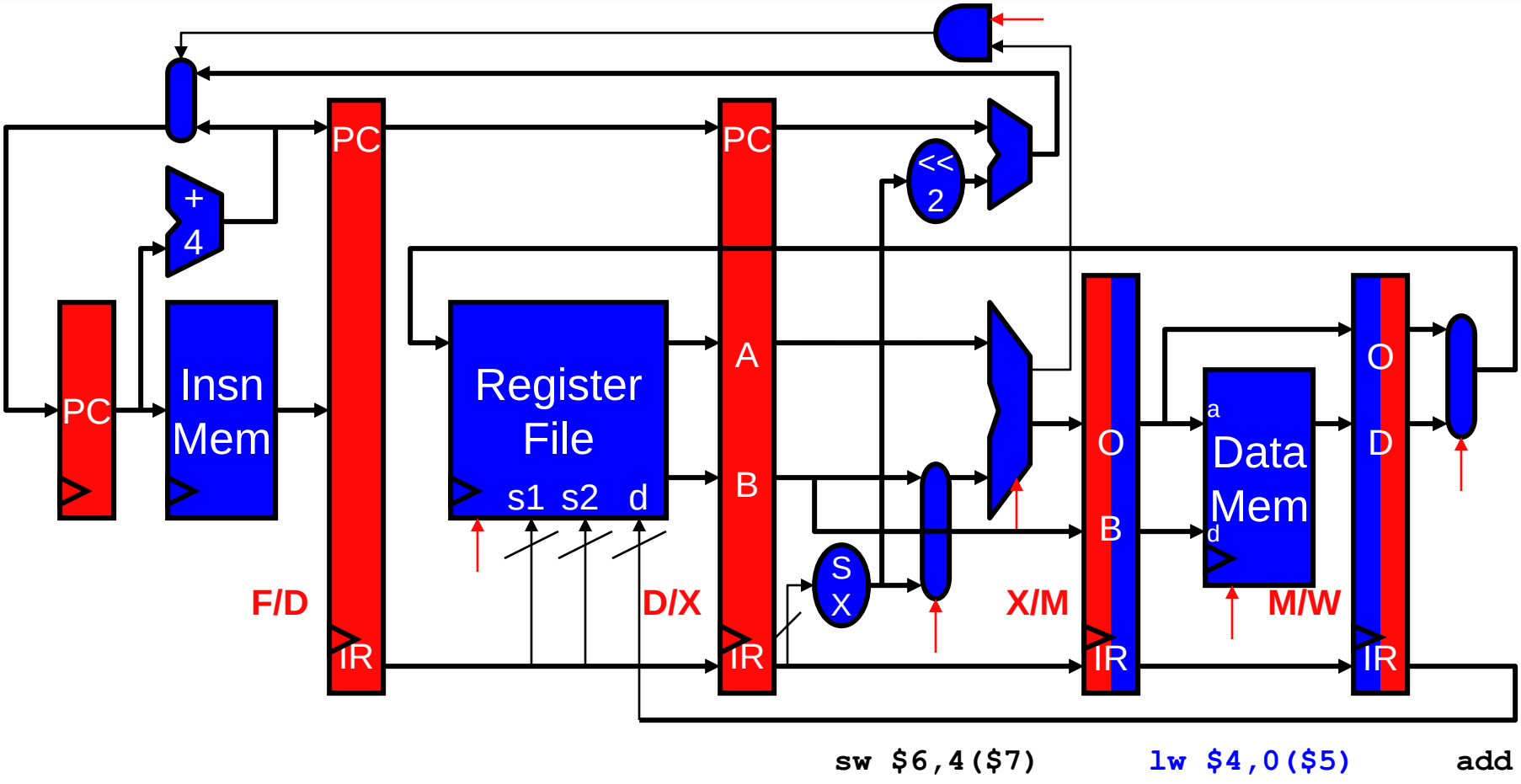
`sw $6,4($7)` `lw $4,0($5)` `add $3,$2,$1`

Pipeline Example: Cycle 4

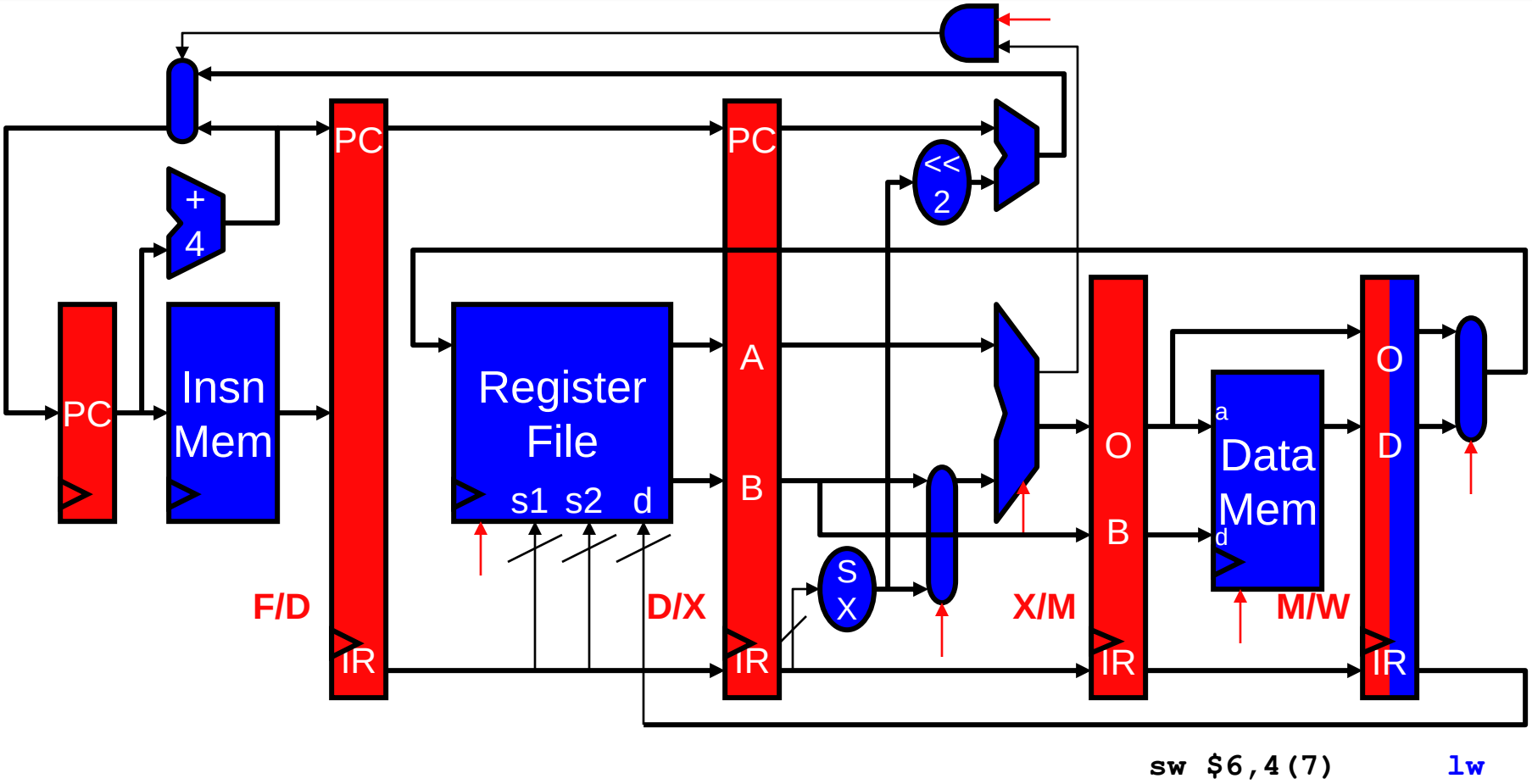


- 3 instructions

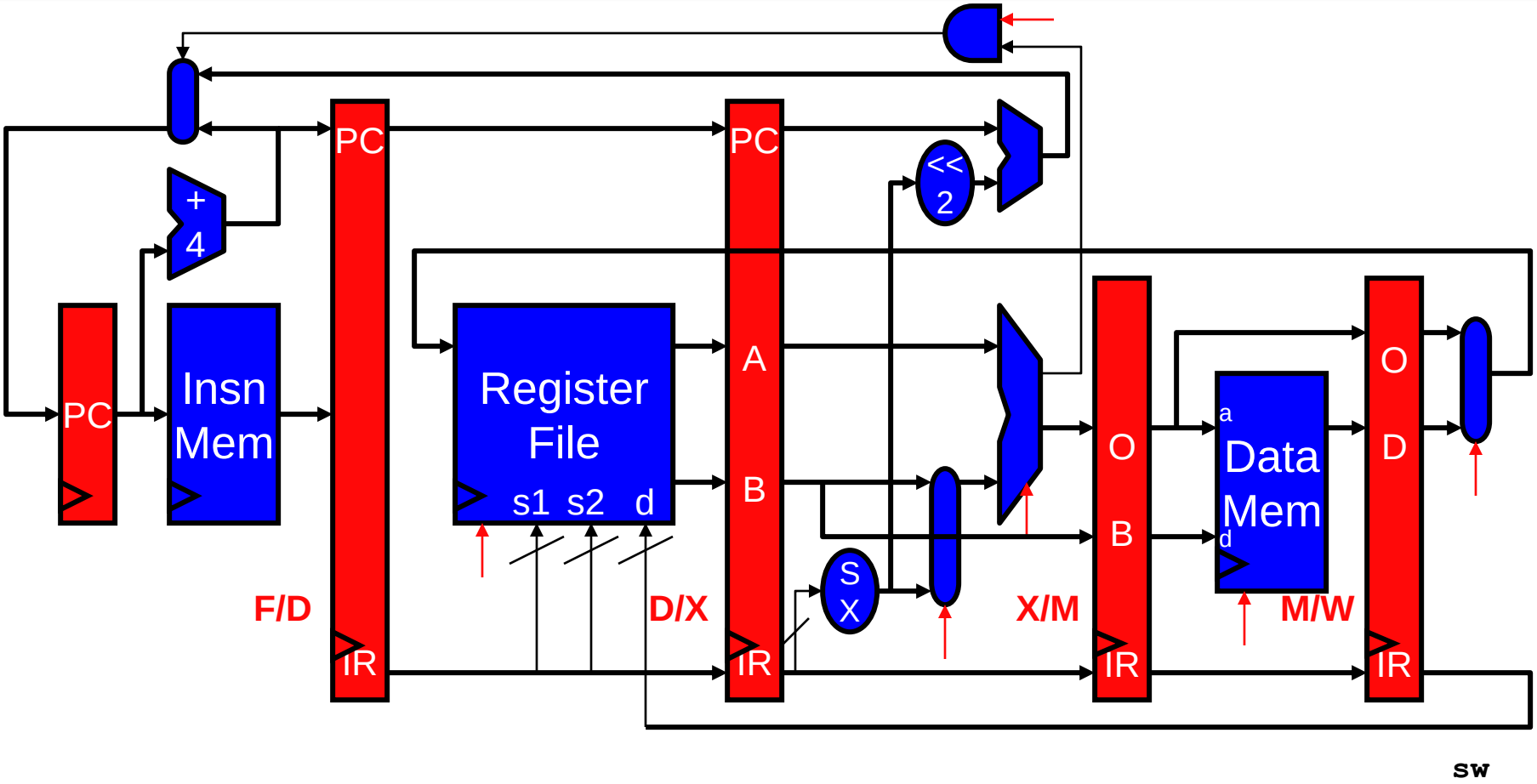
Pipeline Example: Cycle 5



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

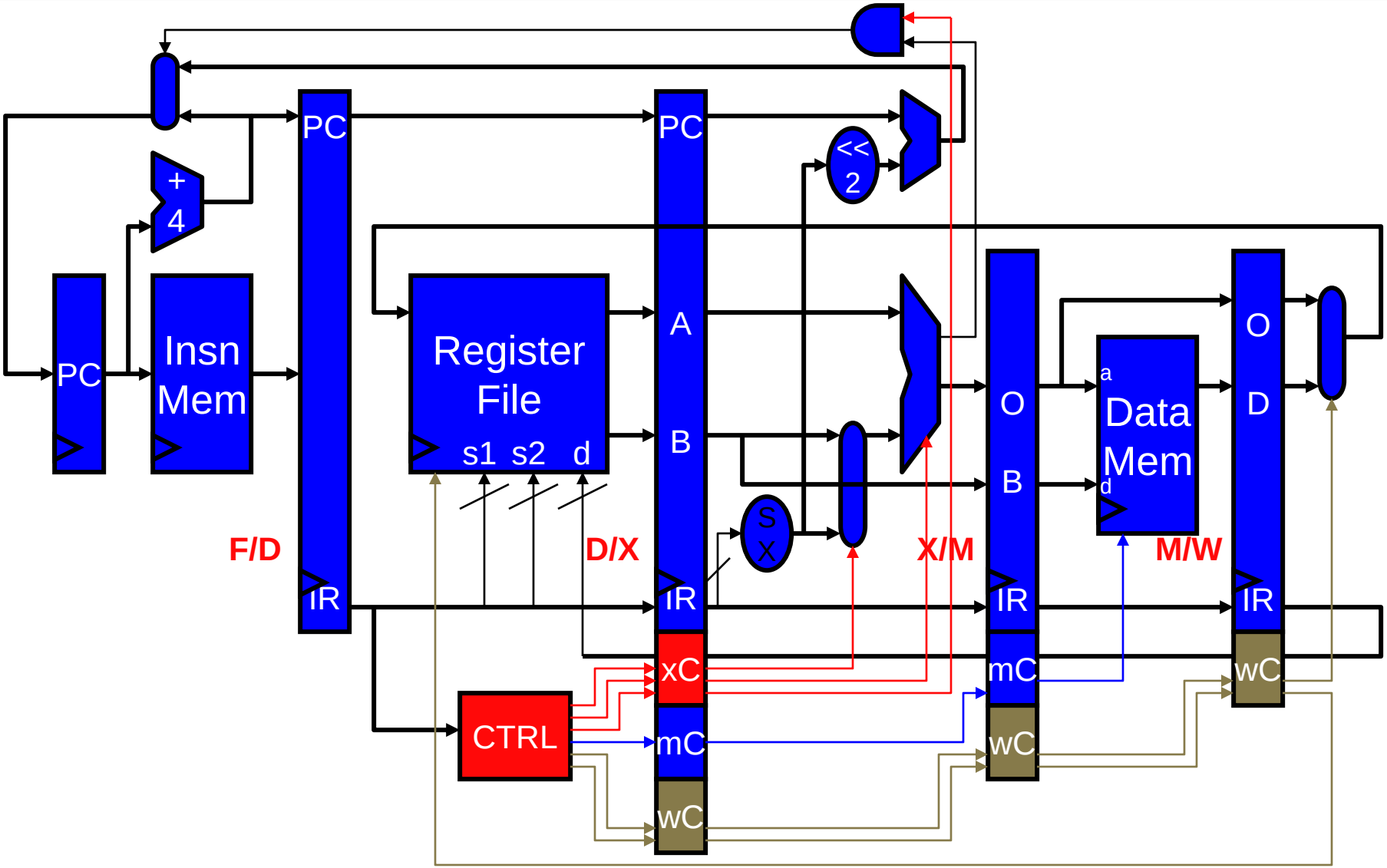
	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,0($5)`		F	D	**X**	M	W			
`sw $6,4($7)`			F	D	X	M	W		

What About Pipelined Control?

- Should it be like single-cycle control?
  - But individual insn signals must be staged
- How many different control units do we need?
  - One for each insn in pipeline?

- Solution: use simple single-cycle control, but pipeline it
  - Single controller
  - Key idea: pass control signals with instruction through pipeline

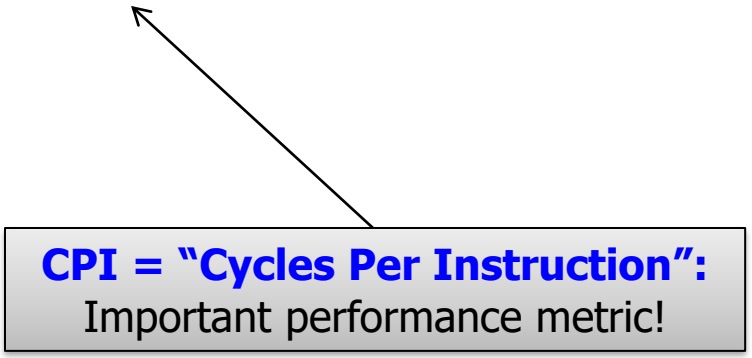
Pipelined Control



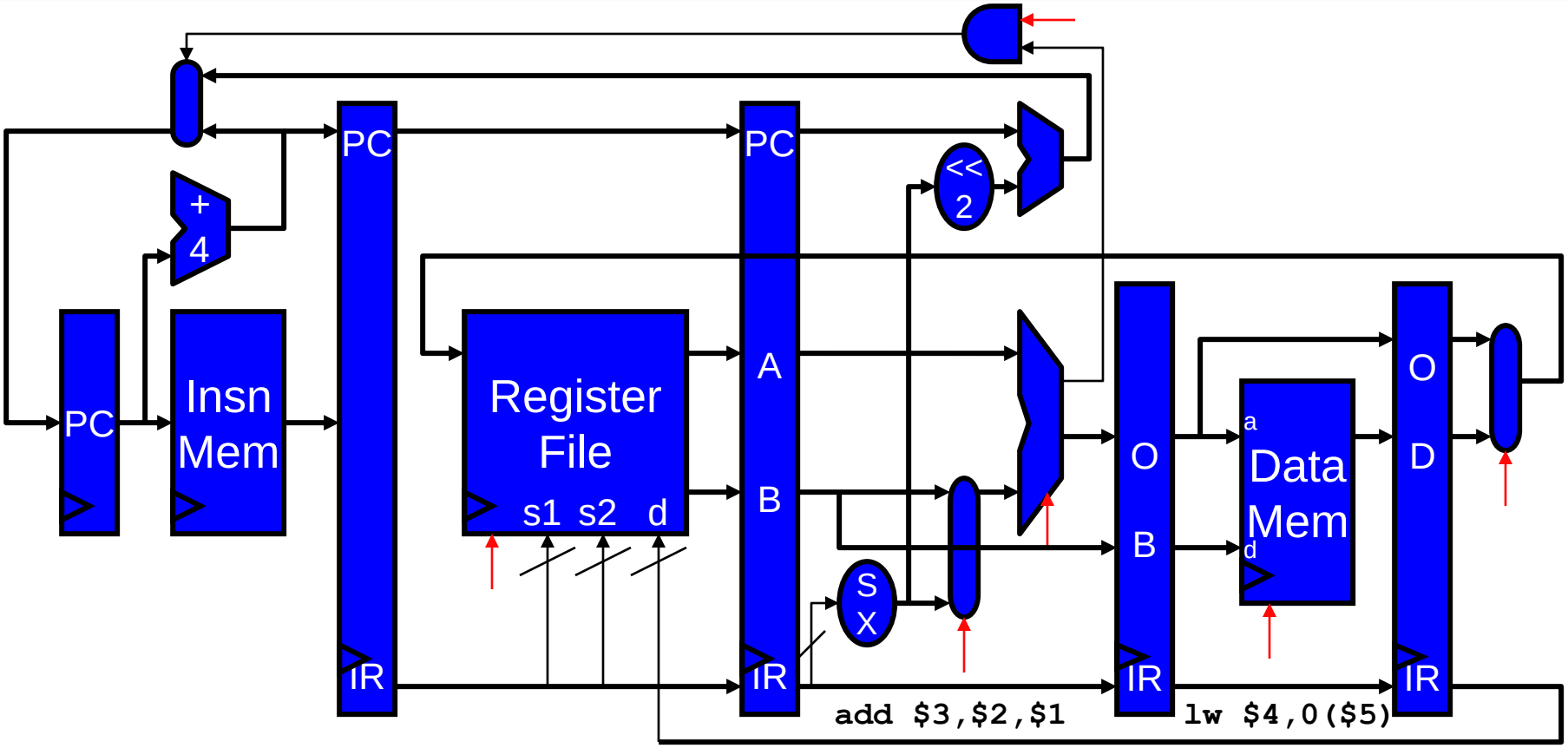
Pipeline Performance Calculation

- Single-cycle
  - Clock period = 50ns, CPI = 1
  - Performance = 50ns/insn

- Pipelined
  - Clock period = **12ns (why not 10ns?)**
  - CPI = **1** (each insn takes 5 cycles, but 1 completes each cycle)
  - Performance = **12ns/insn**



Why Does Every Insn Take 5 Cycles?

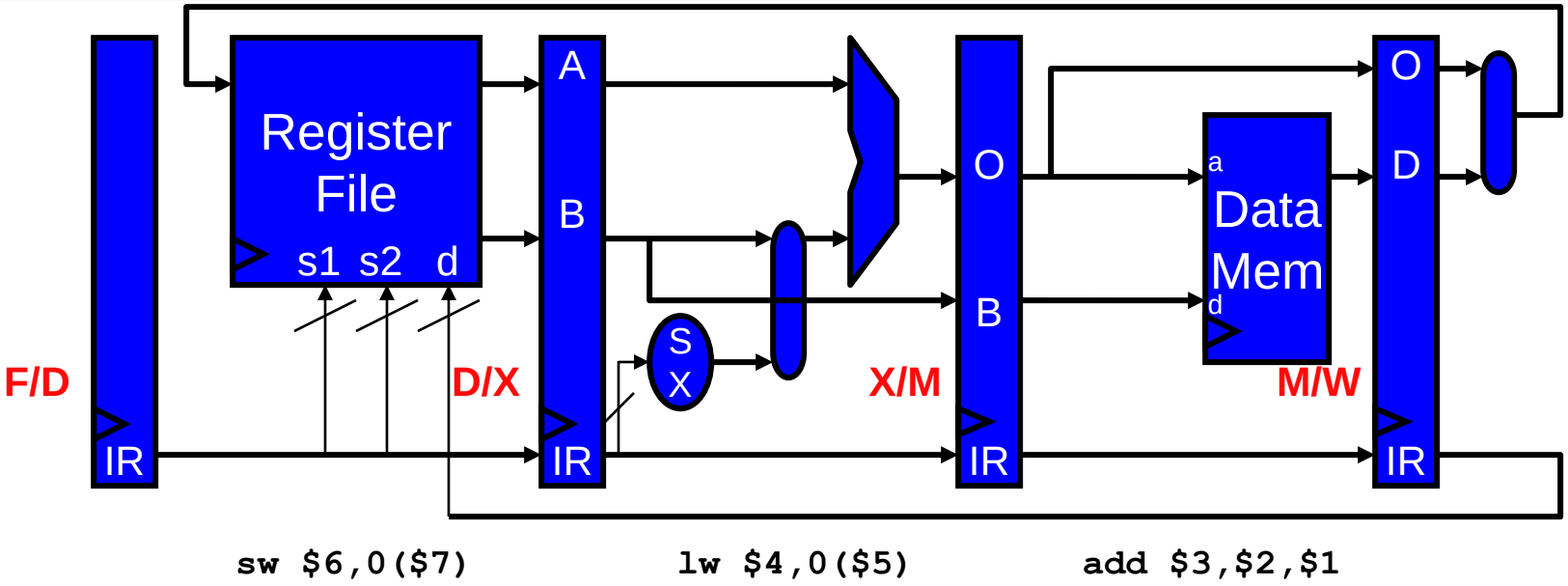


- Why not let **`add`** skip M and go straight to W?
  - It wouldn't help: peak fetch still only 1 insn per cycle
  - **Structural hazards**: not enough resources per stage for 2 insns

Pipeline Hazards

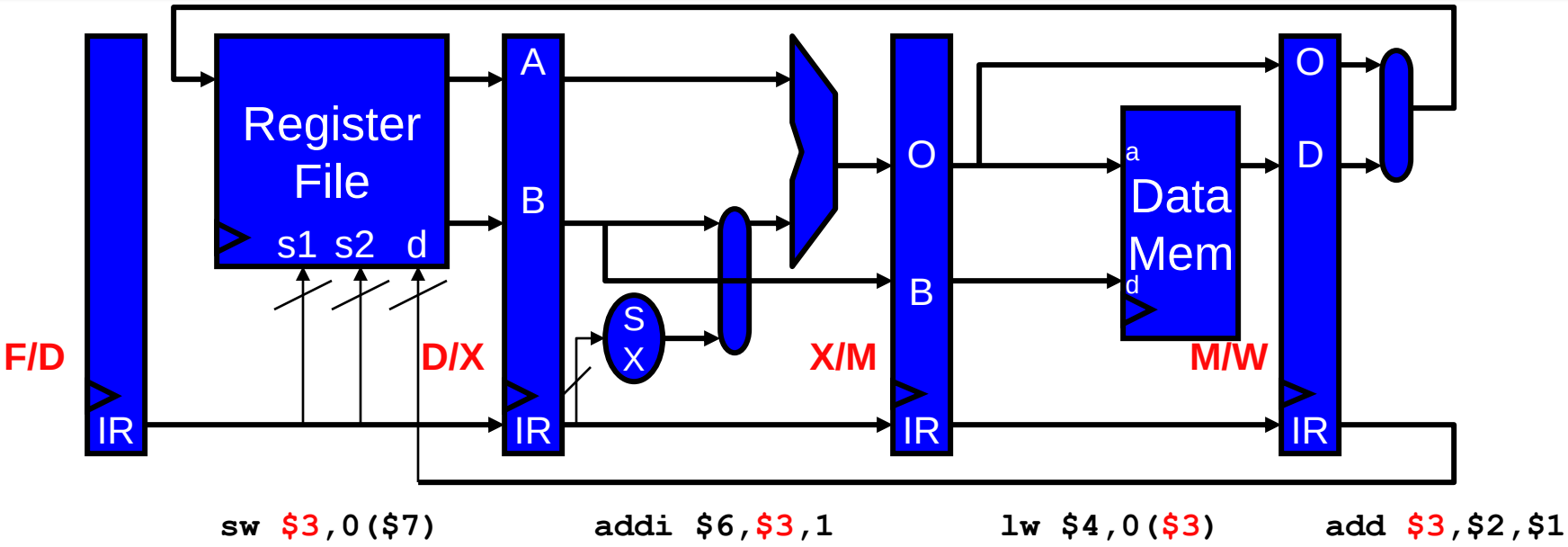
- **Hazard**: condition leads to incorrect execution if not fixed
  - "Fixing" typically increases CPI
  - Three kinds of hazards
- **Structural hazards**
  - Two insns trying to use same circuit at same time
    - E.g., structural hazard on RegFile write port
  - Fix by proper ISA/pipeline design: 3 rules to follow
    - Each insn uses every structure exactly once
    - For at most one cycle
    - Always at same stage relative to F
- **Data hazards** (next)
- **Control hazards** (a little later)

Data Hazards



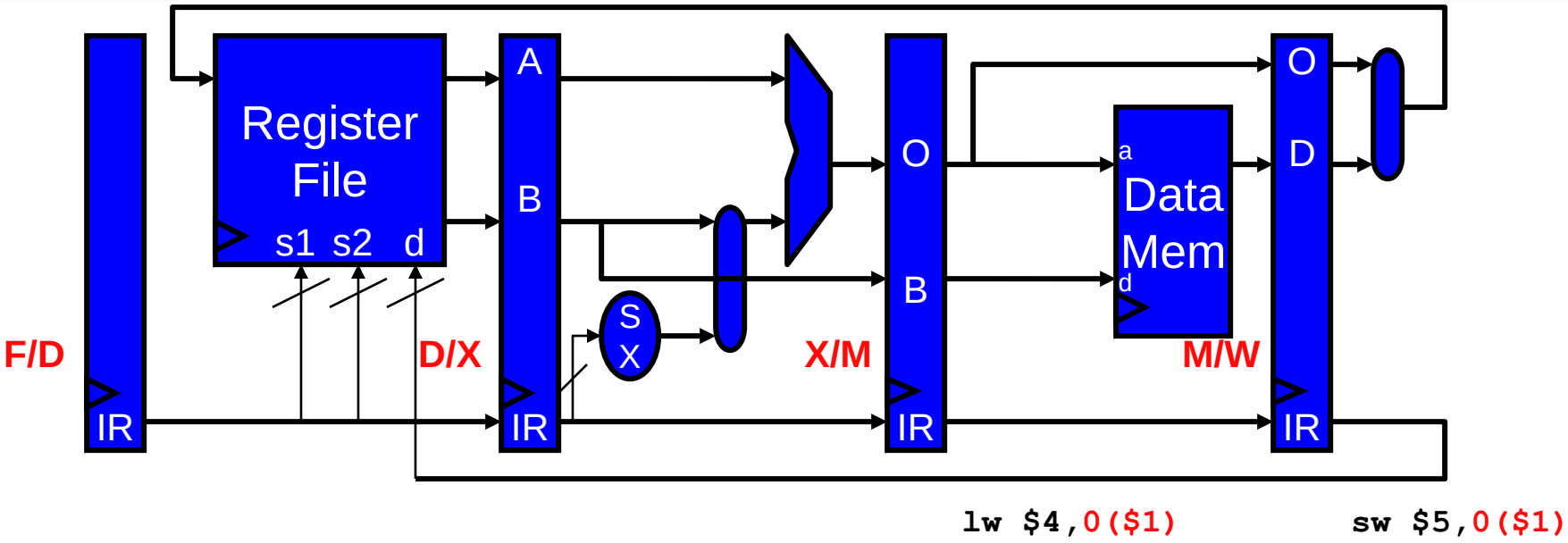
- Let's forget about branches and control for a while
- The sequence of 3 insns we saw earlier executed fine…
  - But it wasn't a real program
  - Real programs have **data dependences**
    - They pass values via registers and memory

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 → OK (regfile timing: write first half)

Memory Data Hazards



- What about data hazards through memory? No
  - `lw` following `sw` to same address in next cycle, gets right value
  - Why? DMem read/write take place in same stage
- Data hazards through registers? Yes (previous slide)
  - Occur because register write is 3 stages after register read
  - Can only read a register value 3 cycles after writing it

Fixing Register Data Hazards

- Can only read register value 3 cycles after writing it

- One way to enforce this: make sure programs can't do it
  - Compiler puts two **independent** insns between write/read insn pair
    - If they aren't there already
  - Independent means: "do not interfere with register in question"
    - Do not write it: otherwise meaning of program changes
    - Do not read it: otherwise create new data hazard
  - **Code scheduling**: compiler moves around existing insns to do this
  - If none can be found, must use **NOPs**

  - This is called **software interlocks**
    - **MIPS**: **M**icroprocessor w/out **I**nterlocking **P**ipeline **S**tages

Software Interlock Example

```
sub $3,$2,$1
lw $4,0($3)
sw $7,0($3)
add $6,$2,$8
addi $3,$5,4
```

- Can any of last 3 insns be scheduled between first two?
  - **`sw $7,0($3)`**? No, creates hazard with **`sub $3,$2,$1`**
  - **`add $6,$2,$8`**? OK
  - **`addi $3,$5,4`**? YES...-ish. Technically. (but it hurts to think about)
    - Would work, since **`lw`** wouldn't get its **`$3`** from it due to delay
    - Makes code REALLY hard to follow – each instruction's effects "happen" at different delays (memory writes "immediate", register writes delayed, etc.)
    - Let's not do this, and just add a **`nop`**s where needed
  - Still need one more insn, use **`nop`**

```
sub $3,$2,$1
add $6,$2,$8
nop
lw $4,0($3)
sw $7,0($3)
addi $3,$5,4
```

Software Interlock Performance

- Software interlocks
  - 20% of insns require insertion of 1 `nop`
  - 5% of insns require insertion of 2 `nops`

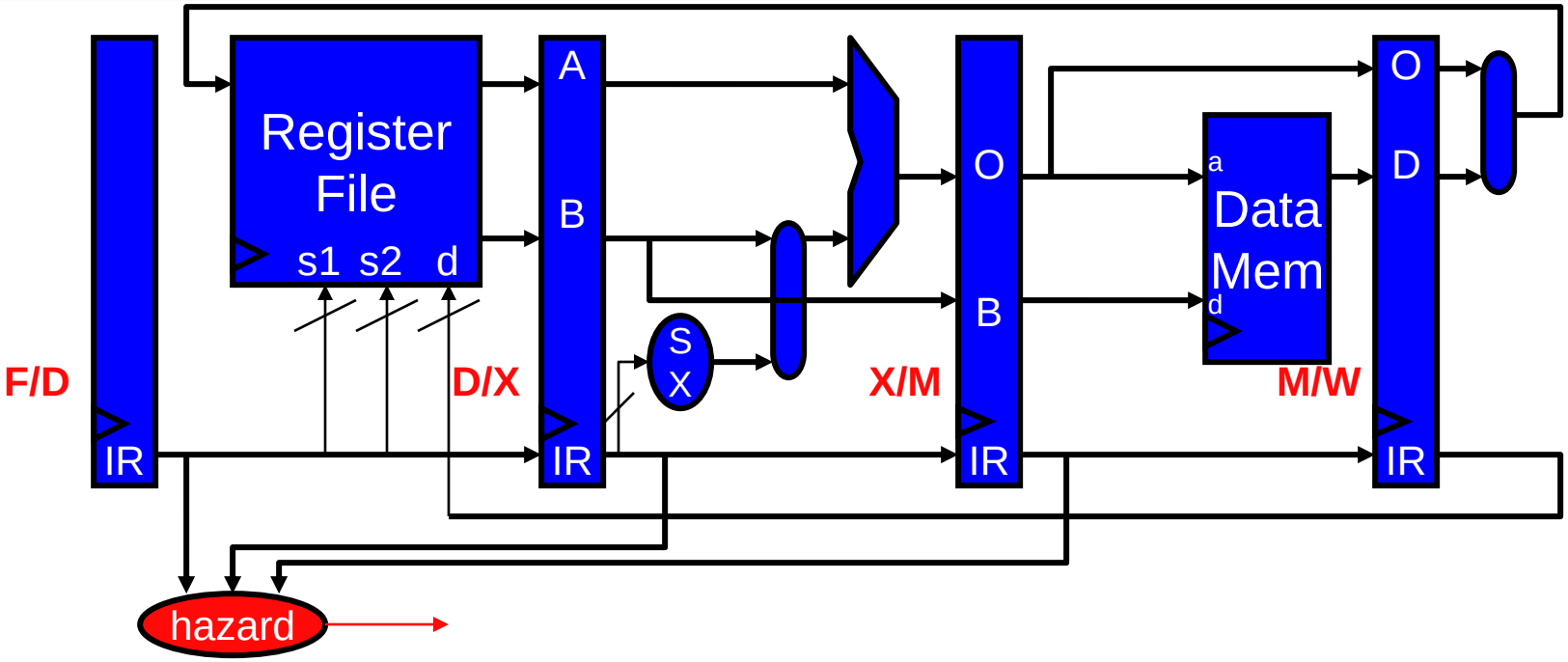
  - CPI is still 1 technically
  - But now there are more insns
  - #insns = 1 + 0.20*1 + 0.05*2 = **1.3**
  - **30% more insns (30% slowdown) due to data hazards**

Hardware Interlocks

- Problem with software interlocks? Not compatible
  - Where does **3** in "read register 3 cycles after writing" come from?
    - From structure (depth) of pipeline
  - What if next MIPS version uses a 7 stage pipeline?
    - Programs compiled assuming 5 stage pipeline will break

- A better (more compatible) way: **hardware interlocks**
  - Processor detects data hazards and fixes them
  - Two aspects to this
    - Detecting hazards
    - Fixing hazards

Detecting Data Hazards

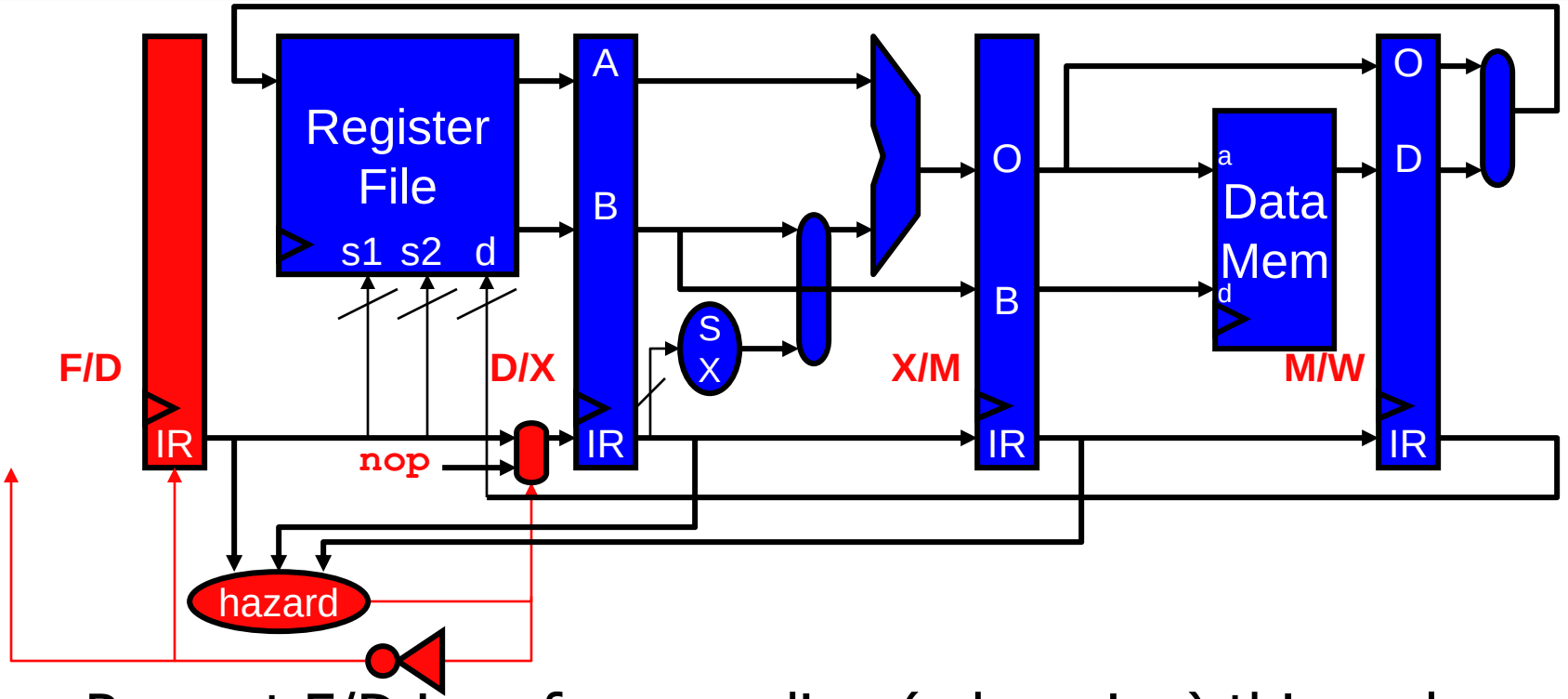


- Compare F/D insn input register names with output register names of older insns in pipeline

  Hazard =

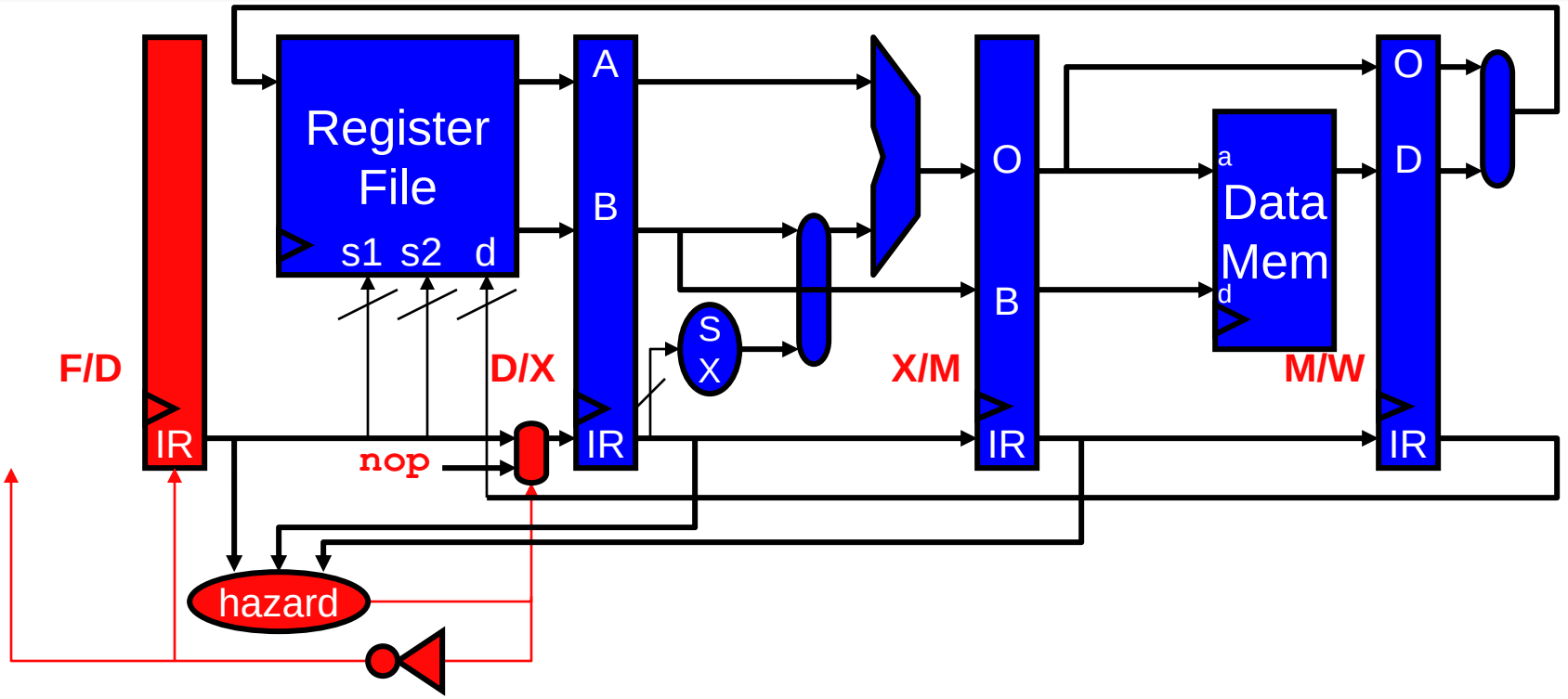
  (F/D.IR.RS1 == D/X.IR.RD) || (F/D.IR.RS2 == D/X.IR.RD) ||
  (F/D.IR.RS1 == X/M.IR.RD) || (F/D.IR.RS2 == X/M.IR.RD)

Fixing Data Hazards



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

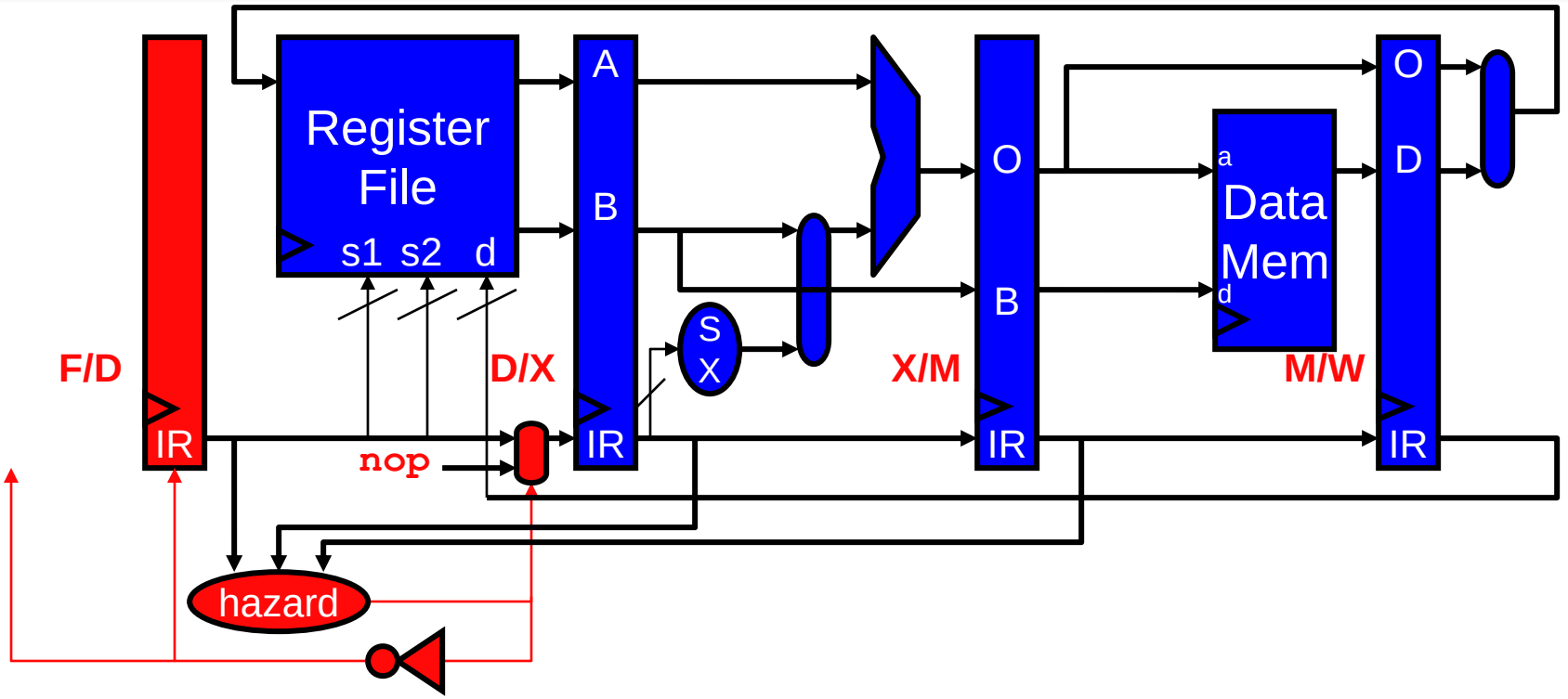
Hardware Interlock Example: cycle 1



(F/D.IR.RS1 == D/X.IR.RD) || (**F/D.IR.RS2 == D/X.IR.RD**) ||
(F/D.IR.RS1 == X/M.IR.RD) || (F/D.IR.RS2 == X/M.IR.RD)

= **1**

Hardware Interlock Example: cycle 2

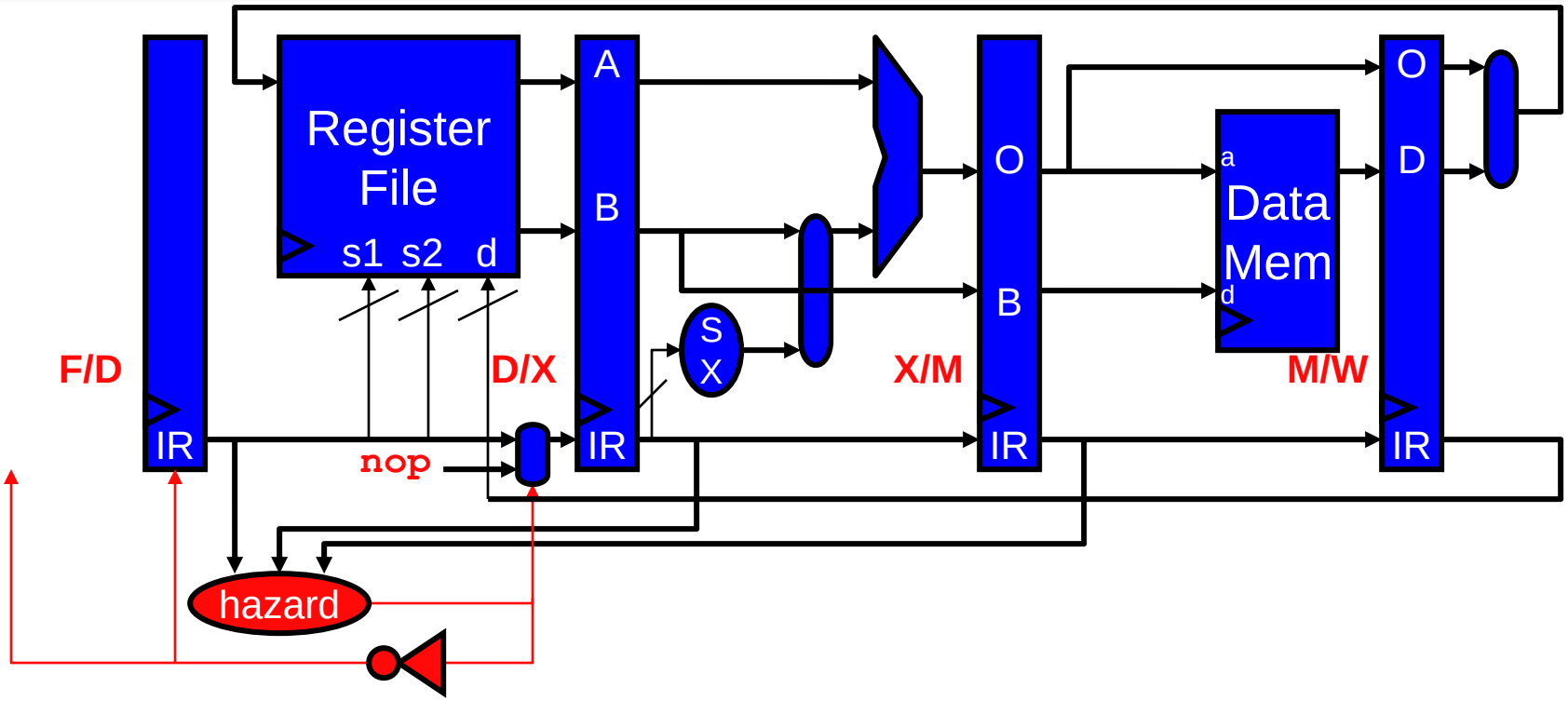


(F/D.IR.RS1 == D/X.IR.RD) || (F/D.IR.RS2 == D/X.IR.RD) ||

(F/D.IR.RS1 == X/M.IR.RD) || (**F/D.IR.RS2 == X/M.IR.RD**)

= **1**

Hardware Interlock Example: cycle 3



`lw $4,0($3)`

`add $3,$2,$1`

(F/D.IR.RS1 == D/X.IR.RD) || (F/D.IR.RS2 == D/X.IR.RD) ||
(F/D.IR.RS1 == X/M.IR.RD) || (F/D.IR.RS2 == X/M.IR.RD)

= **0**

Pipeline Control Terminology

- Hardware interlock maneuver is called **stall** or **bubble**

- Mechanism is called **stall logic**
- Part of more general **pipeline control** mechanism
  - Controls advancement of insns through pipeline
- Distinguished from **pipelined datapath control**
  - Controls datapath at each stage
  - Pipeline control controls advancement of datapath control

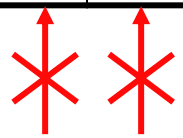
Pipeline Diagram with Data Hazards

- Data hazard stall indicated with **d***
  - Stall propagates to younger insns

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

  - This is not OK (why?)

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



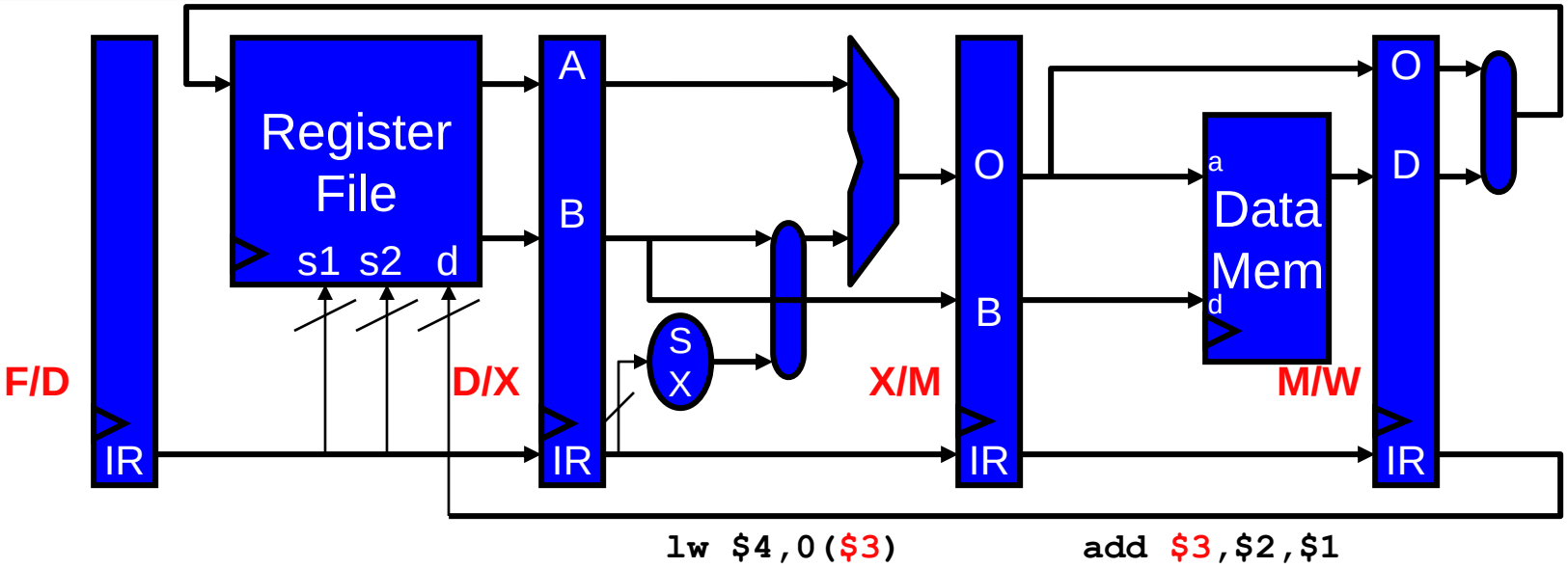
Hardware Interlock Performance

- Hardware interlocks: same as software interlocks
  - 20% of insns require 1 cycle stall (i.e., insertion of 1 **`nop`**)
  - 5% of insns require 2 cycle stall (i.e., insertion of 2 **`nop`**s)

  - CPI = 1 + 0.20*1 + 0.05*2 = **1.3**
  - So, either CPI stays at 1 and #insns increases 30% (software)
  - Or, #insns stays at 1 (relative) and CPI increases 30% (hardware)
  - Same difference

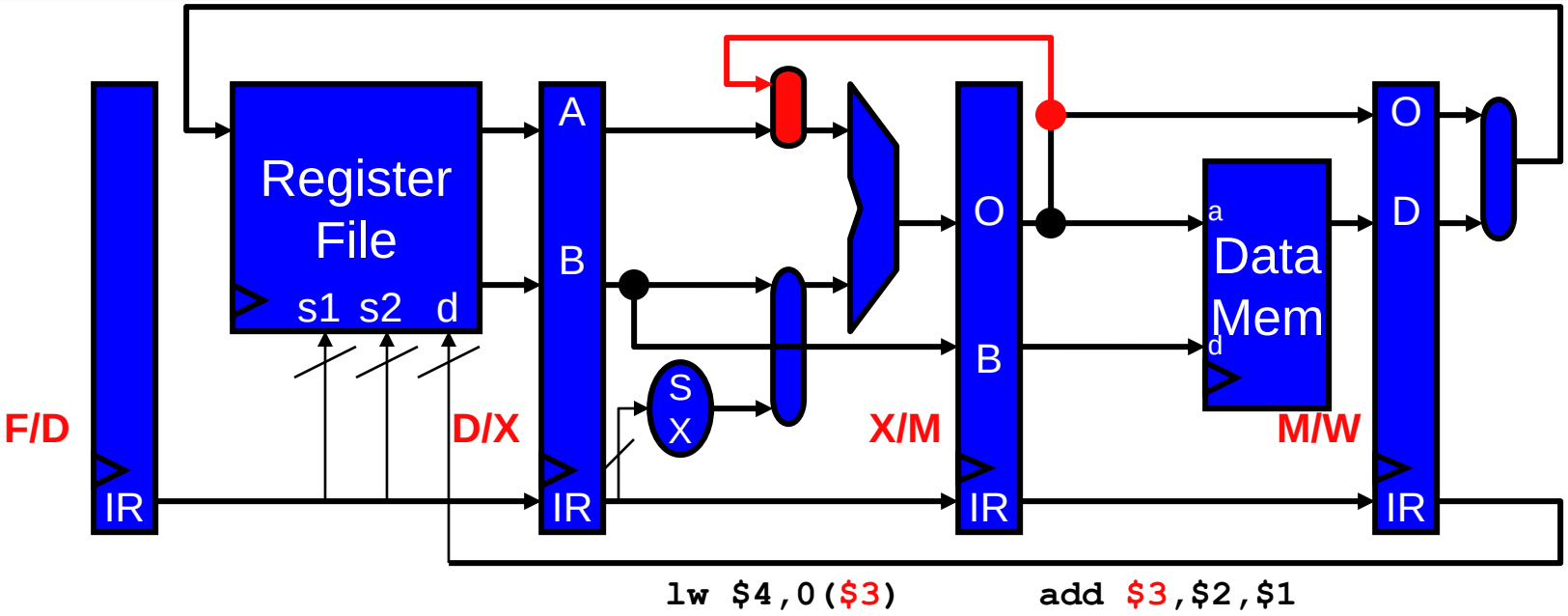
- Anyway, we can do better

Observe



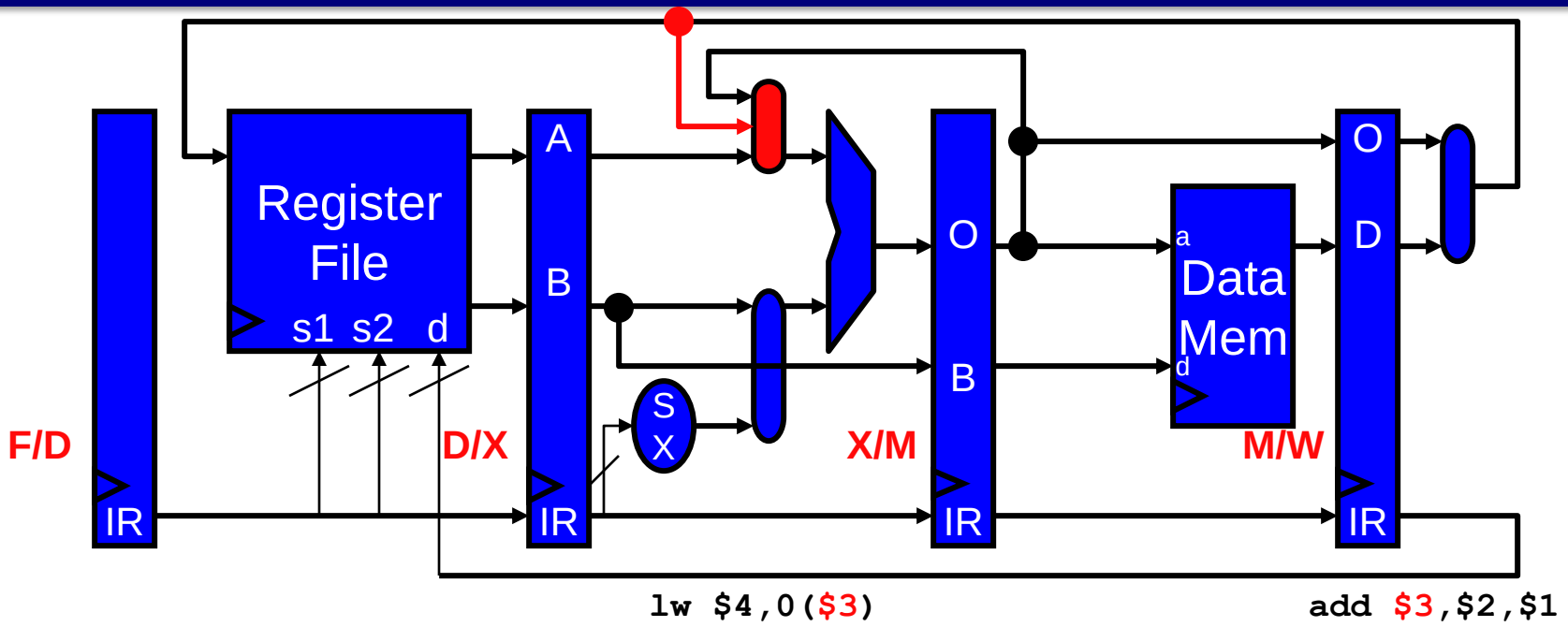
- This situation seems broken
  - `lw $4,0($3)` has already read `$3` from regfile
  - `add $3,$2,$1` hasn't yet written `$3` to regfile
- But fundamentally, everything is still OK
  - `lw $4,0($3)` hasn't actually **used** `$3` yet
  - `add $3,$2,$1` has already computed `$3`

Bypassing



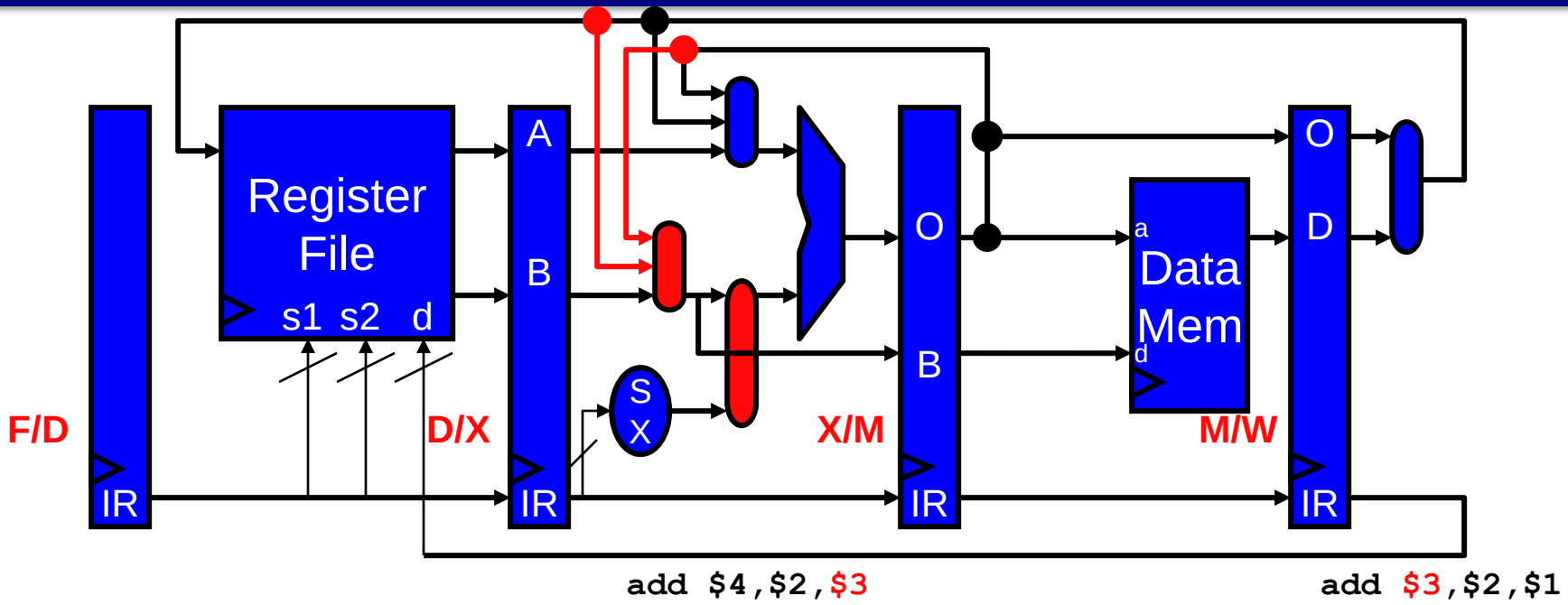
- **Bypassing**
  - Reading a value from an intermediate (μarchitectural) source
  - Not waiting until it is available from primary source (RegFile)
  - Here, we are bypassing the register file
  - Also called **forwarding**

WX Bypassing



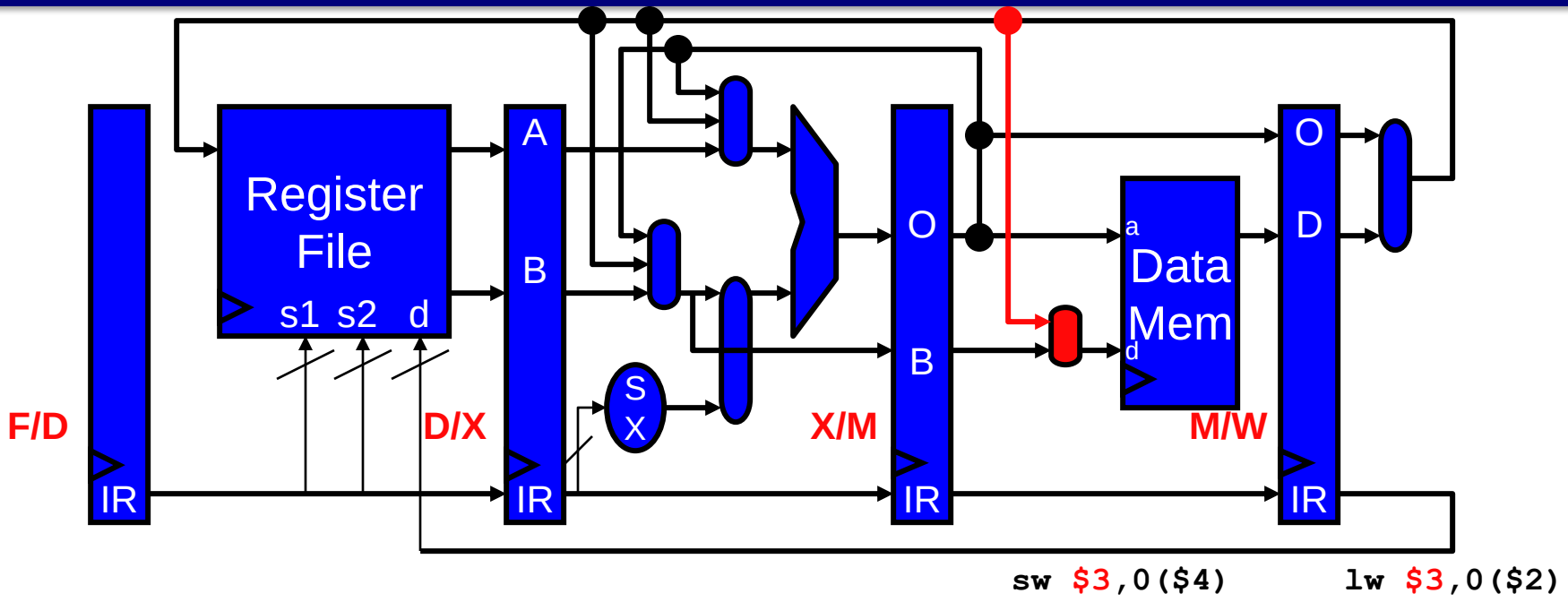
- What about this combination?
  - Add another bypass path and MUX input
  - First one was an **MX** bypass
  - This one is a **WX** bypass

ALUinB Bypassing



- Can also bypass to ALU input B

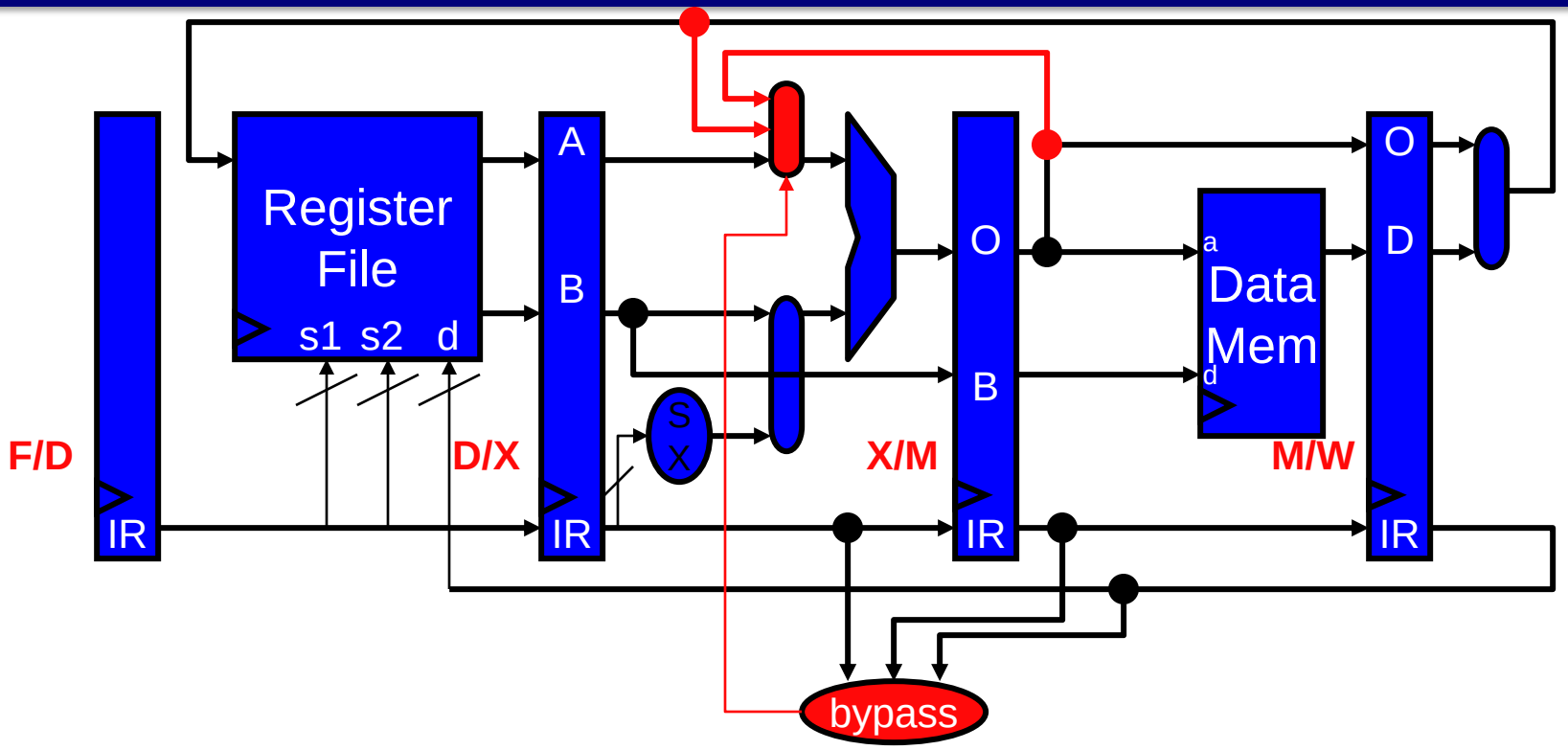
WM Bypassing?



- Does WM bypassing make sense?
  - Not to the address input (why not?)
    - Address input requires the ALU to compute; value is not ready *anywhere* in the CPU
  - But to the store data input, yes

This slide shows **full bypassing** (all bypasses possible in this design).

Bypass Logic



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

  (D/X.IR.RS1 == X/M.IR.RD) → mux select = 0

  (D/X.IR.RS1 == M/W.IR.RD) → mux select = 1

  Else → mux select = 2

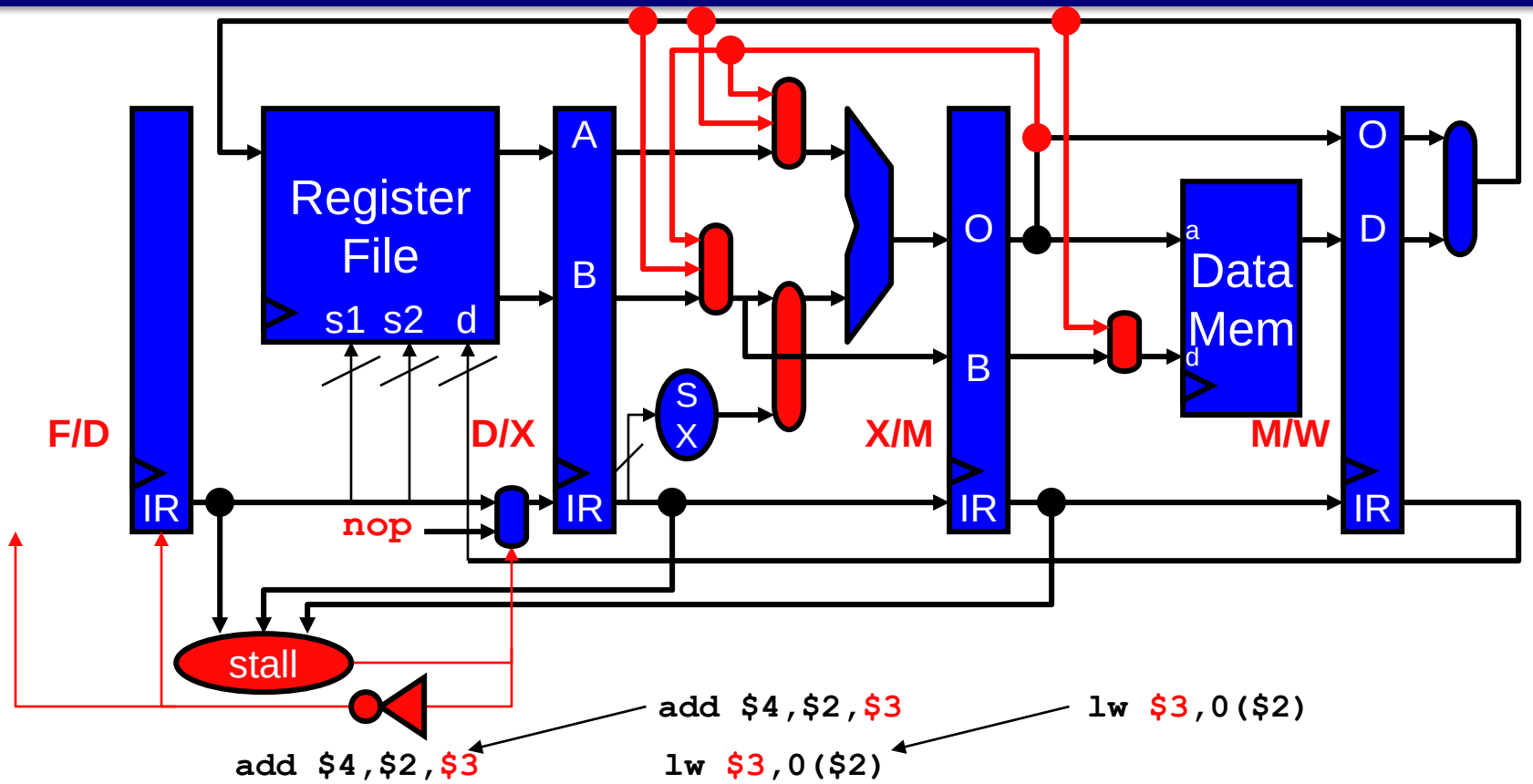
Bypass and Stall Logic

- Two separate things
  - Stall logic controls pipeline registers
  - Bypass logic controls muxes
- But complementary
  - For a given data hazard: if can't bypass, must stall

- Slide #40 shows **full bypassing**: all bypasses possible
  - Is stall logic still necessary?

Our CPU's stall condition!

Yes, Load Output to ALU Input




```
Stall = (D/X.IR.OP==LOAD) && (
          (F/D.IR.RS1==D/X.IR.RD) ||
          (F/D.IR.RS2==D/X.IR.RD)
        )
```

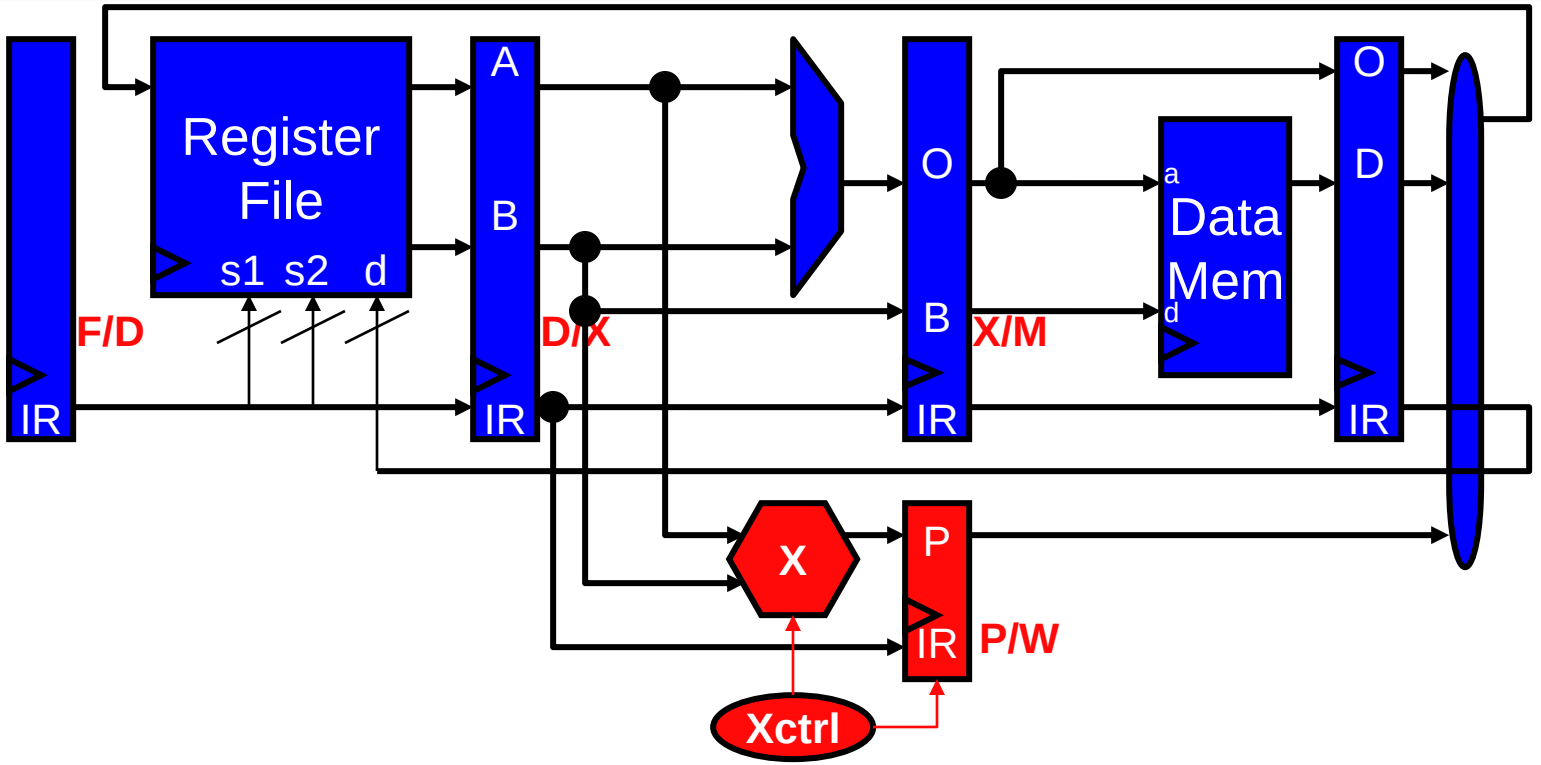
Pipeline Diagram With Bypassing

	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,0($3)`		F	D	X	M	W			
`addi $6,$4,1`			F	**d***	D	X	M	W	
`sub $9,$10,$11`					F	D	X	M	W

- Sometimes you will see it like this
  - Denotes that stall logic implemented at X stage, rather than D
  - Equivalent, doesn't matter when you stall as long as you do

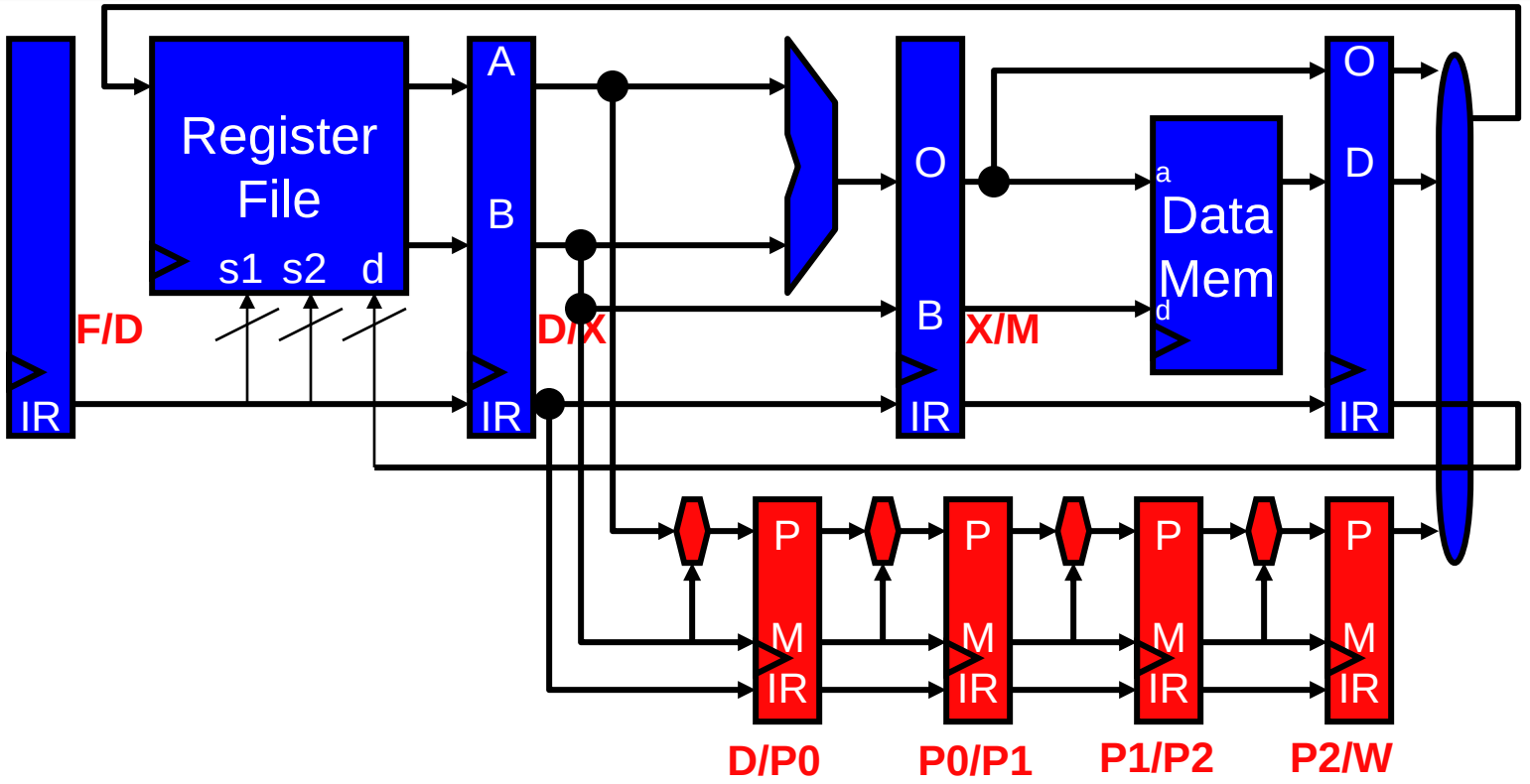
	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,0($3)`		F	D	X	M	W			
`addi $6,$4,1`			F	D	**d***	X	M	W	
`sub $9,$10,$11`					F	D	X	M	W

Pipelining and Multi-Cycle Operations



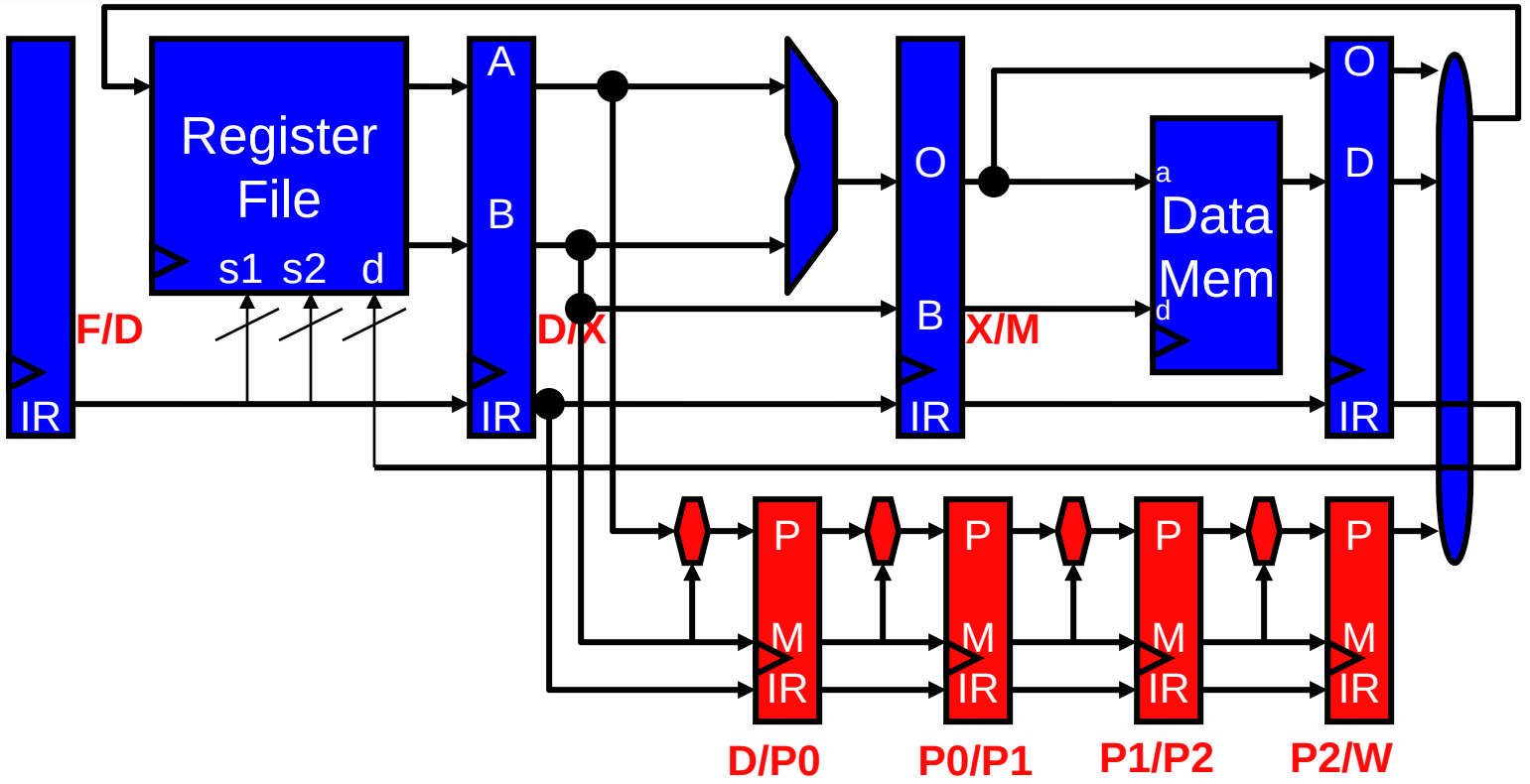
- What if you wanted to add a multi-cycle operation?
  - E.g., 4-cycle multiply
  - **P/W**: separate output latch connects to W stage
  - Controlled by pipeline control and multiplier FSM

A Pipelined Multiplier



- Multiplier itself is often pipelined: what does this mean?
  - Product/multiplicand register/ALUs/latches replicated
  - Can start different multiply operations in consecutive cycles

What about Stall Logic?



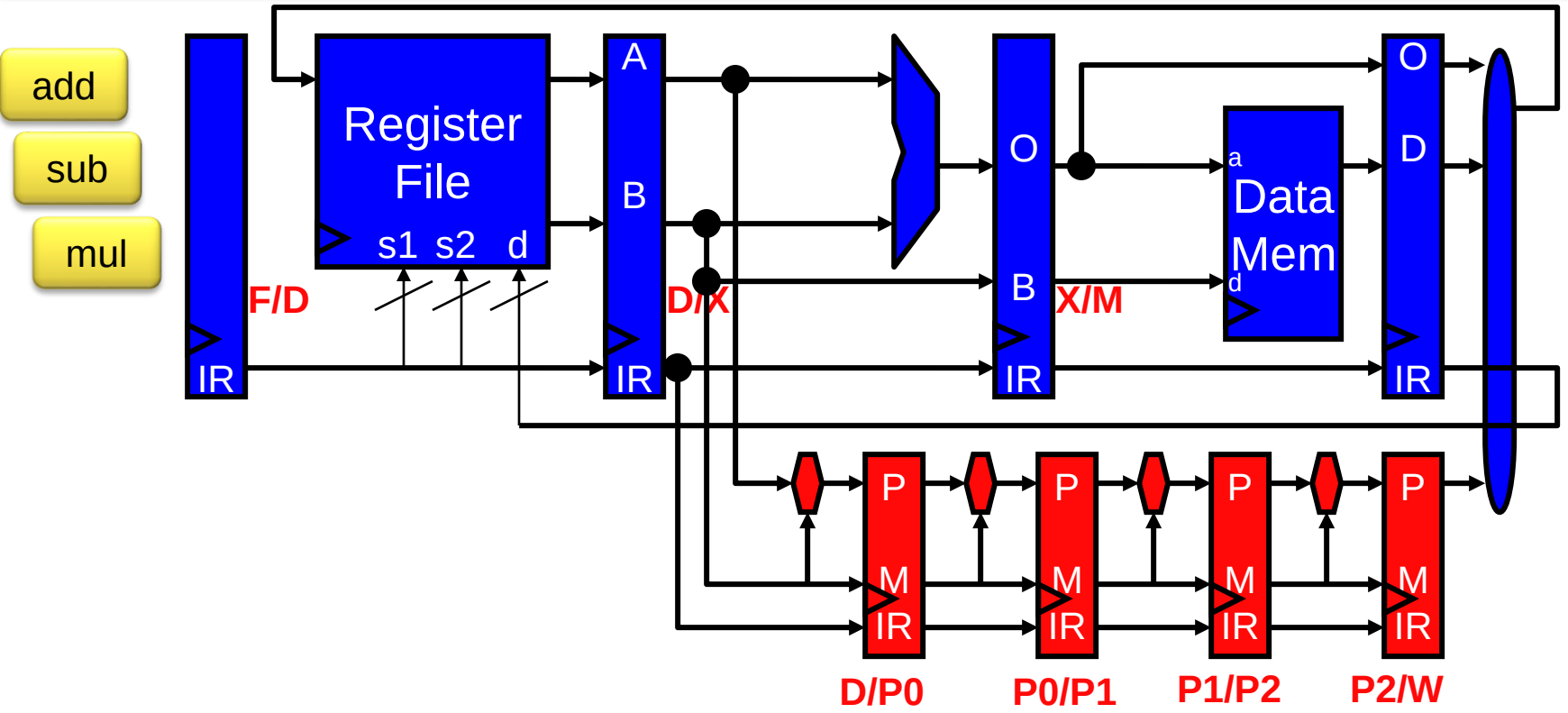
Stall = (OldStallLogic) ||

(F/D.IR.RS1 == D/P0.IR.RD) || (F/D.IR.RS2 == D/P0.IR.RD) ||

(F/D.IR.RS1 == P0/P1.IR.RD) || (F/D.IR.RS2 == P0/P1.IR.RD) ||

(F/D.IR.RS1 == P1/P2.IR.RD) || (F/D.IR.RS2 == P1/P2.IR.RD)

Actually, It's Somewhat Nastier



- What does this do? Hint: think about structural hazards

  Stall = (OldStallLogic) ||

  (**F/D.IR.RD != null && D/P0.IR.RD != null**)

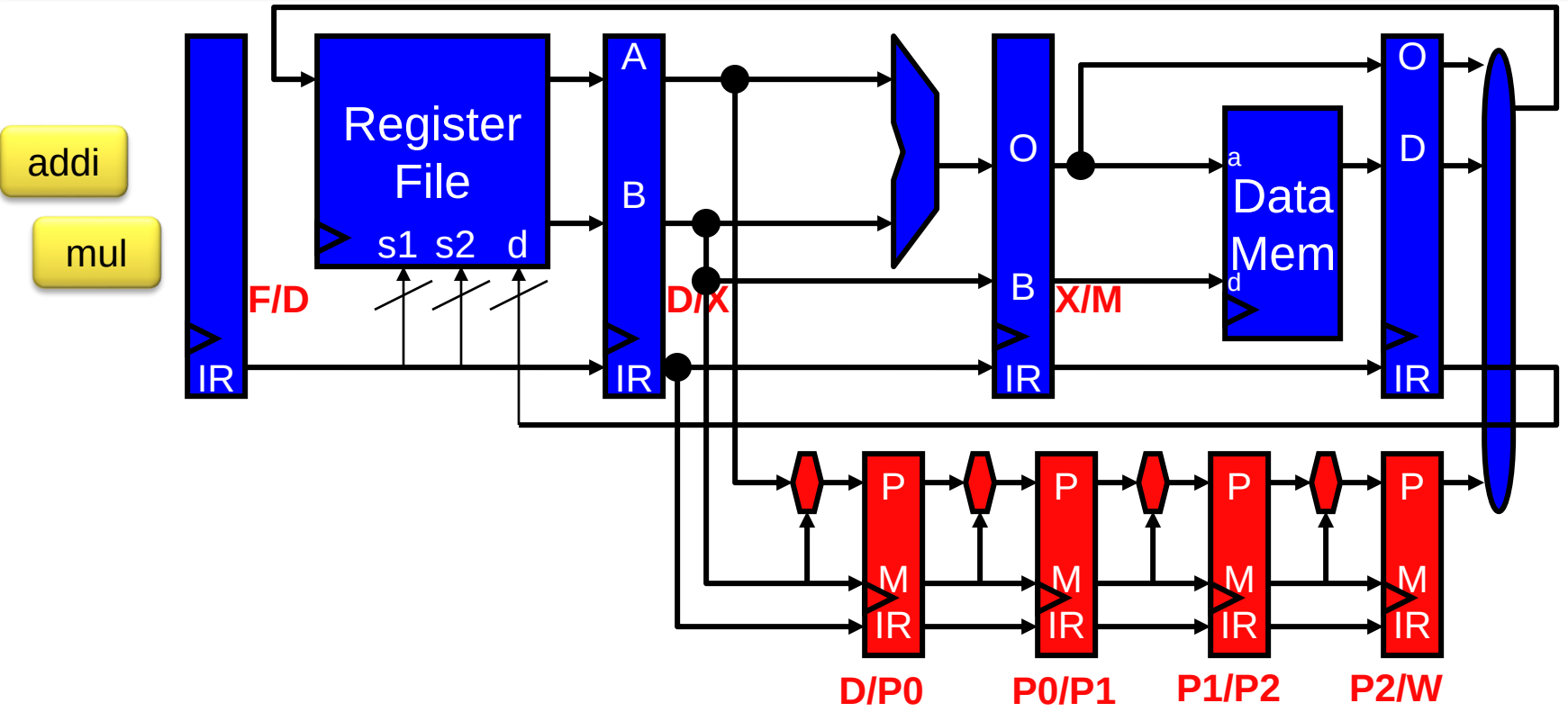
Pipeline Diagram with Multiplier

	1	2	3	4	5	6	7	8	9
`mul $4,$3,$5`	F	D	P0	P1	P2	P3	W		
`sub $6,$1,$8`		F	**d***	**d***	**d***	D	X	M	W

- This is the situation that the previous logic tries to avoid
  - Two instructions trying to write RegFile in same cycle

	1	2	3	4	5	6	7	8	9
`mul $4,$3,$5`	F	D	P0	P1	P2	P3	W		
`sub $6,$1,$8`		F	D	X	M	W			
`add $5,$6,$10`			F	D	X	M	**W**		

Honestly, It's Even Nastier Than That



- And what about this? ("WAR" hazard)

  Stall = (OldStallLogic) ||

  (F/D.IR.RD == D/P0.IR.RD) ||
  (F/D.IR.RD == P0/P1.IR.RD)

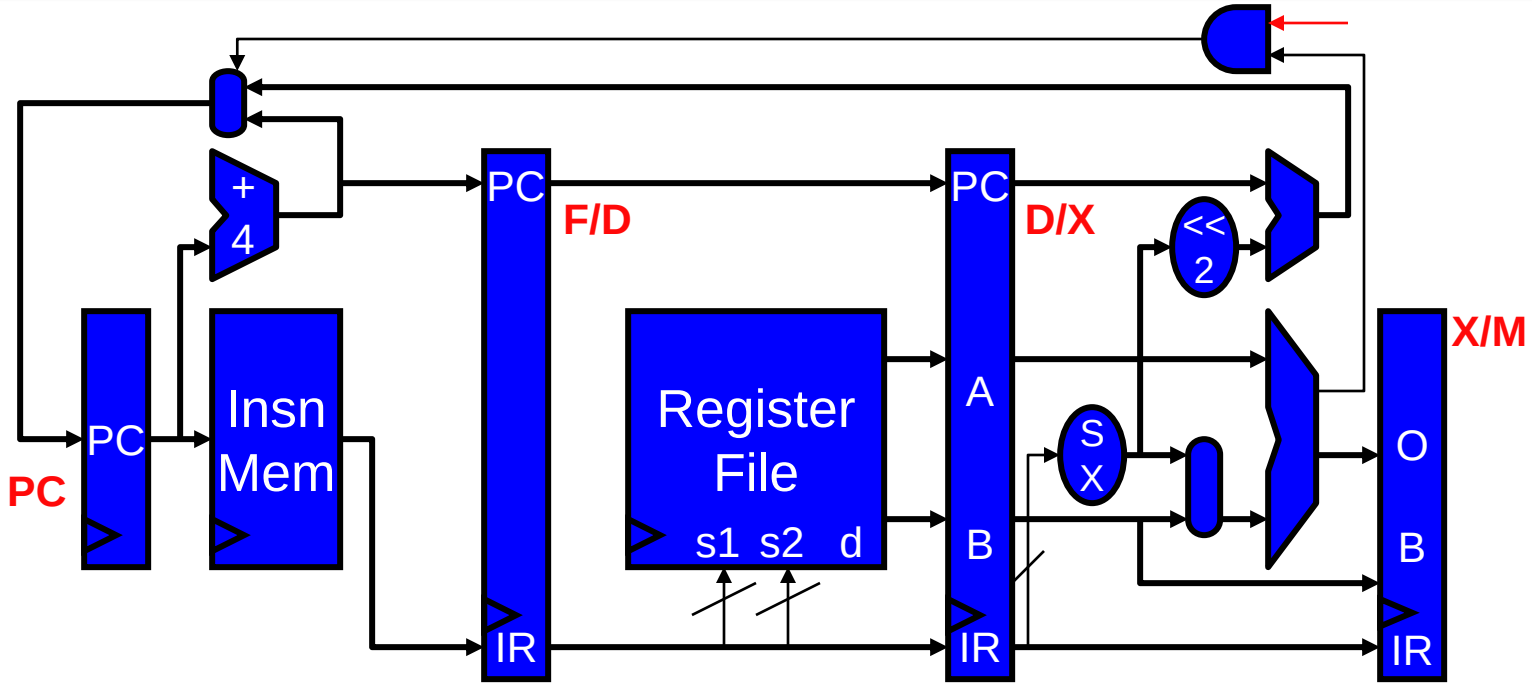
More Multiplier Nasties

- This is the situation that the previous slide tries to avoid
  - Mis-ordered writes to the same register
  - Compiler thinks add gets **`$4`** from **`addi`**, actually gets it from **`mul`**

	1	2	3	4	5	6	7	8	9
`mul $4,$3,$5`	F	D	P0	P1	P2	P3	**W**		
`addi $4,$1,1`		F	D	X	M	W			
…									
…									
`add $10,$4,$6`						F	D	X	M

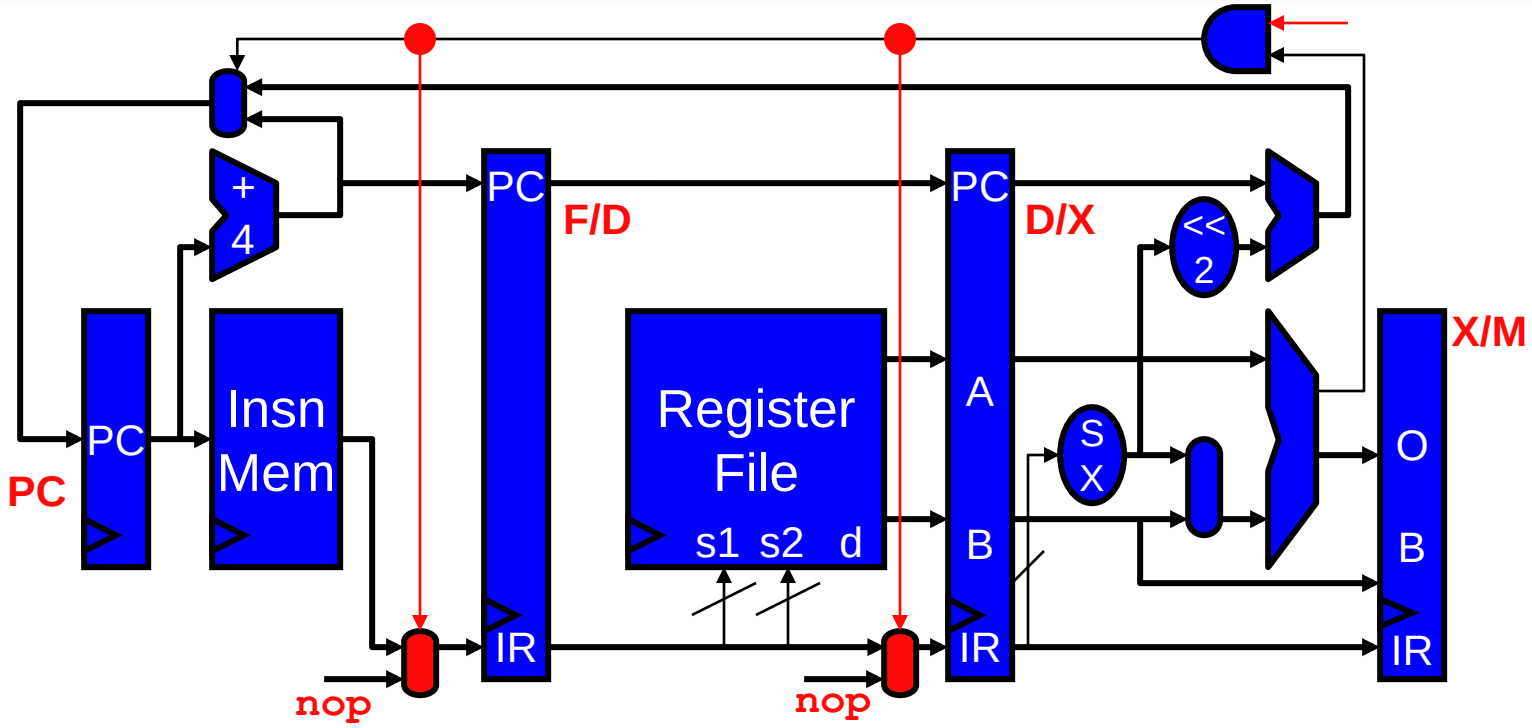
- **Multi-cycle operations complicate pipeline logic**
  - They're not impossible, but they require more complexity

Control Hazards



- **Control hazards**
  - Must fetch post branch insns before branch outcome is known
  - Default: assume "**not-taken**" (at fetch, can't tell if it's a branch)

Branch Recovery



- **Branch recovery**: what to do when branch **is** taken
  - **Flush** insns currently in F/D and D/X (they're wrong)
    - Replace with **`NOPs`**

    + Haven't yet written to permanent state (RegFile, DMem)

Control Hazard Pipeline Diagram

- Control hazards indicated with **c*** (or not at all)
  - Penalty for taken branch is 2 cycles

	1	2	3	4	5	6	7	8	9
`addi $3,$0,1`	F	D	X	M	W				
`bnez $3,targ`		F	D	X	M	W			
`sw $6,4($7)`			**c***	**c***	F	D	X	M	W

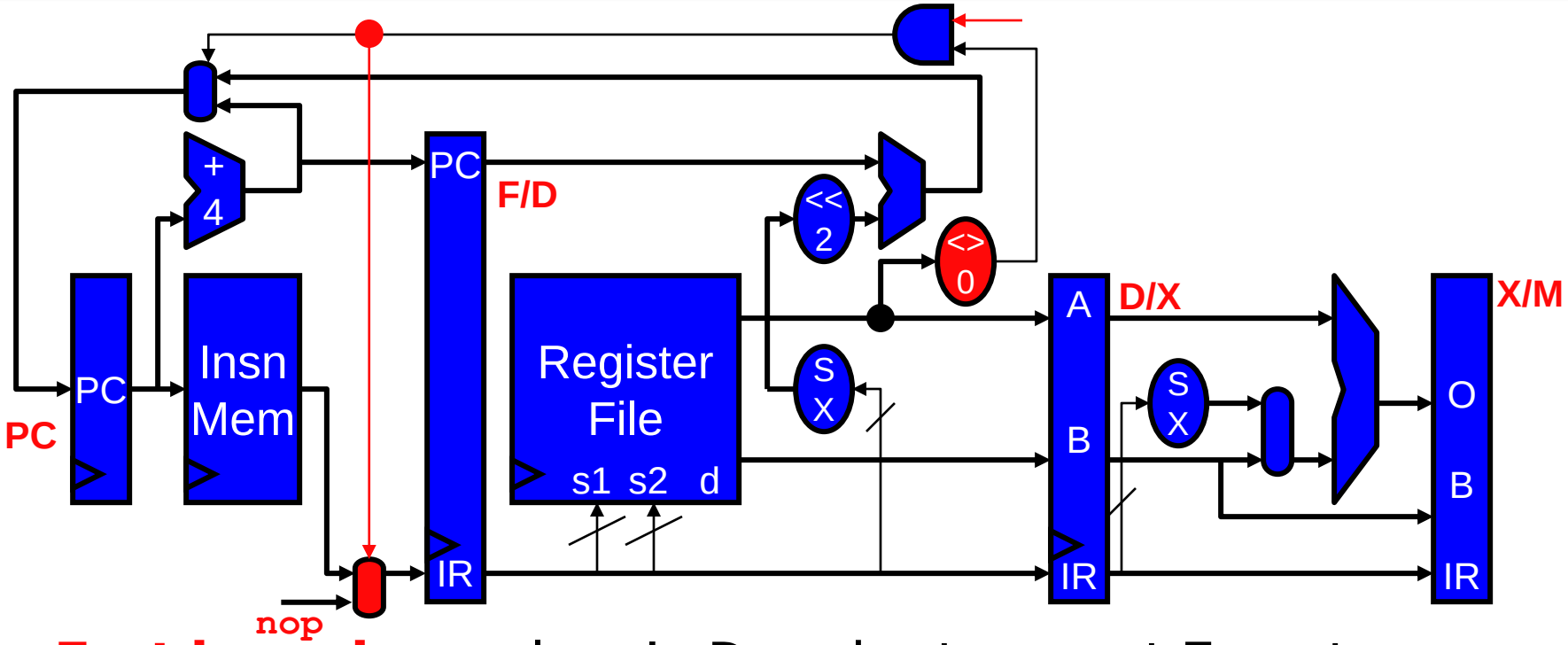
Branch Performance

- Again, measure effect on CPI (clock period is fixed)

- Back of the envelope calculation
  - **Branch: 20%**, load: 20%, store: 10%, other: 50%
  - **75% of branches are taken (why so many taken?)**

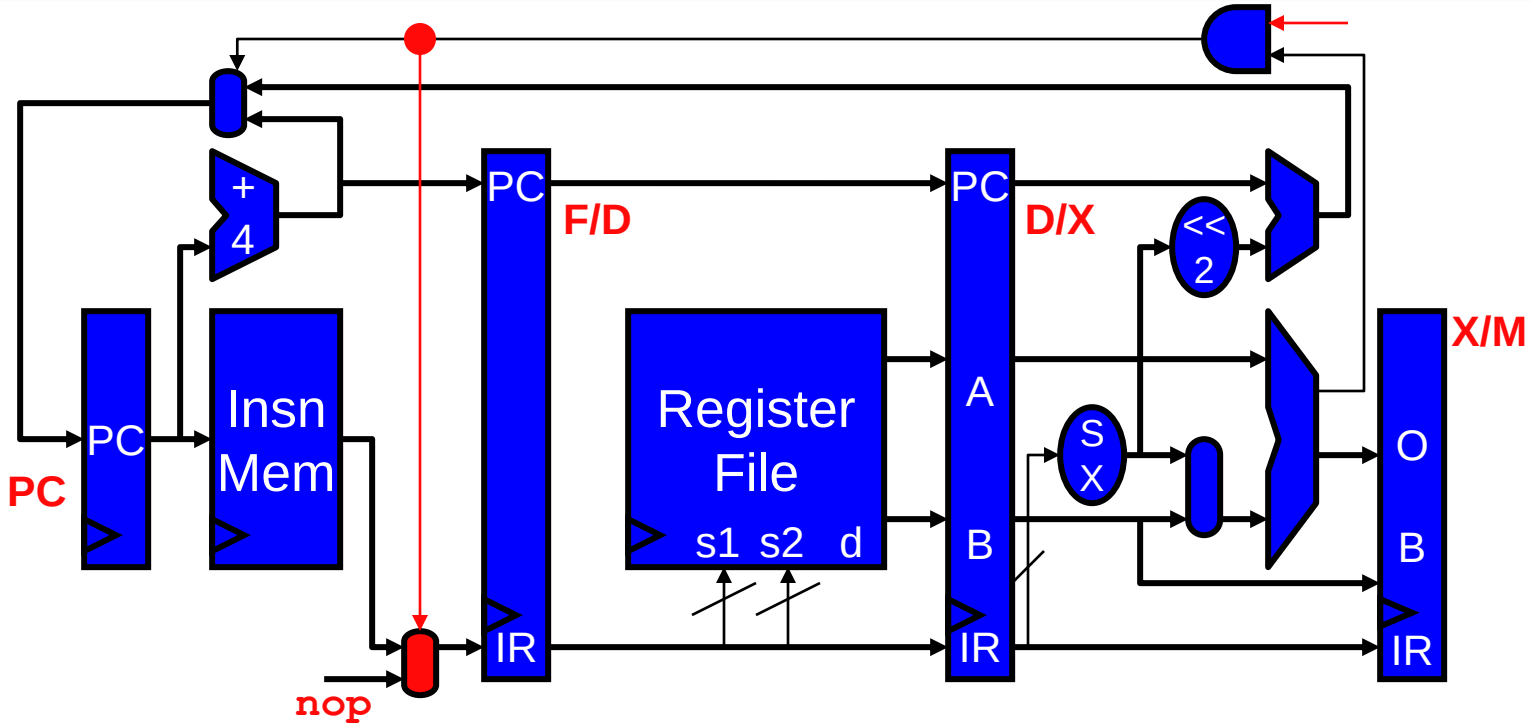
- CPI if no branches = 1
- CPI with branches = 1 + 0.20*0.75*2 = 1.3
  - **Branches cause 30% slowdown**
  - How do we reduce this penalty?

Option 1: Fast Branches



- **Fast branch**: resolves in Decode stage, not Execute
  - Test must be comparison to zero or equality, no time for ALU
  + New taken branch penalty is only 1
  – Need additional comparison insns (`slt`) for complex tests
  – Must be able to bypass into decode now, too

Option 2: Delayed Branches

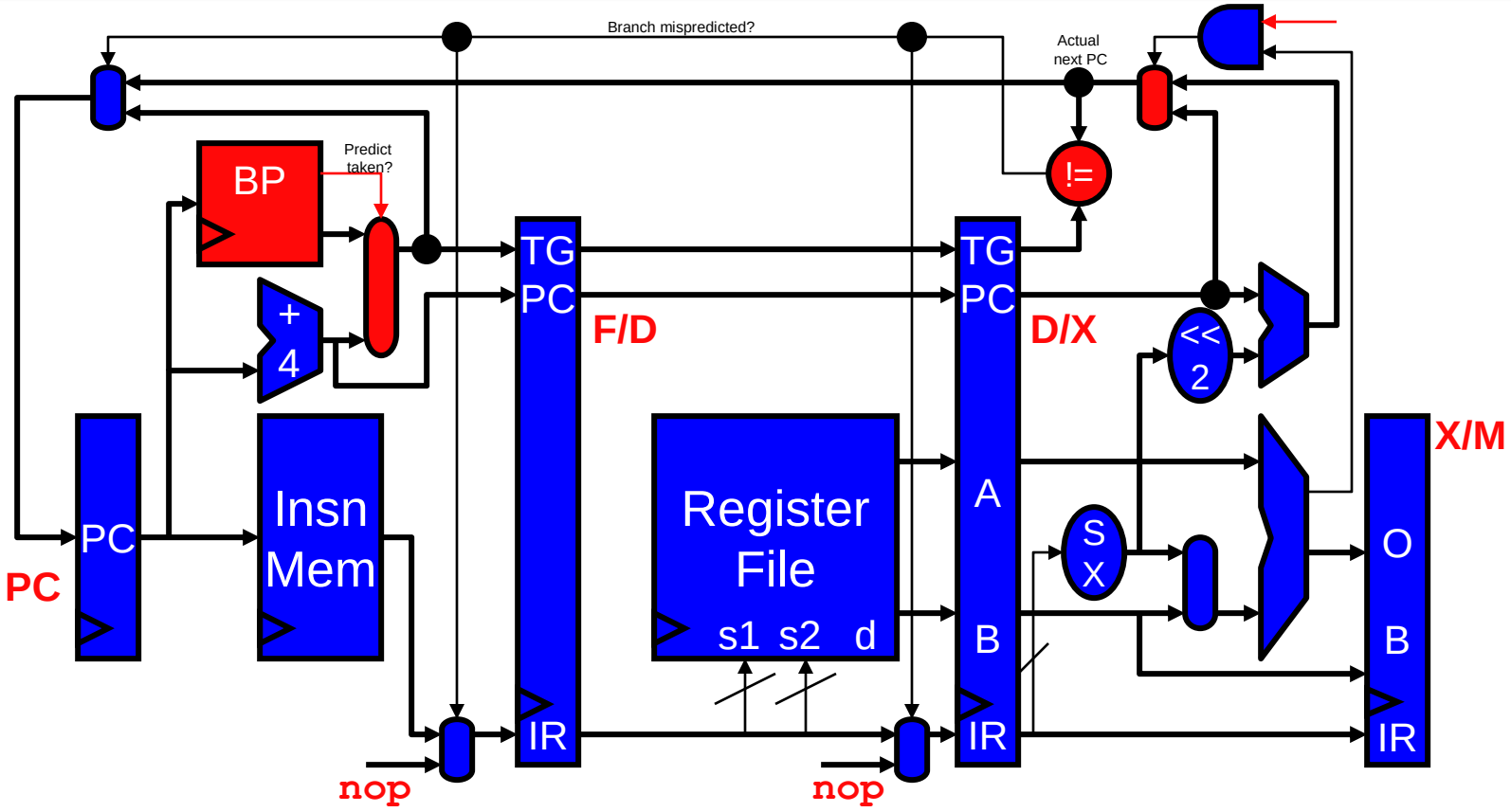


- **Delayed branch**: don't flush insn immediately following
  - As if branch takes effect one insn later
  - ISA modification → compiler accounts for this behavior
  - Insert insns independent of branch into **branch delay slot(s)**

Improved Branch Performance?

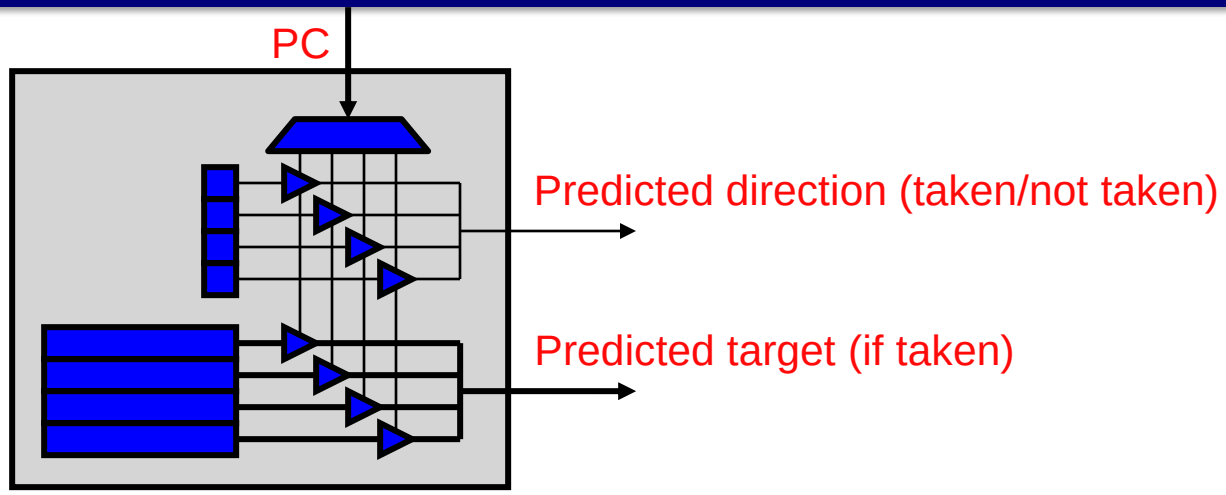
- Same parameters
  - **Branch: 20%**, load: 20%, store: 10%, other: 50%
  - 75% of branches are taken
- Fast branches
  - 25% of branches have complex tests that require extra insn
  - CPI = 1 + 0.20*0.75*1(branch) + 0.20*0.25*1(extra insn) = **1.2**
- Delayed branches
  - 50% of delay slots can be filled with insns, others need nops
  - CPI = 1 + 0.20*0.75*1(branch) + 0.20*0.50*1(extra insn) = **1.25**
  - **Bad idea: painful for compiler, gains are minimal**
  - E.g., delayed branches in SPARC architecture (Sun computers)
    - Also MIPS (but not in SPIM by default)

Option 3: Dynamic Branch Prediction



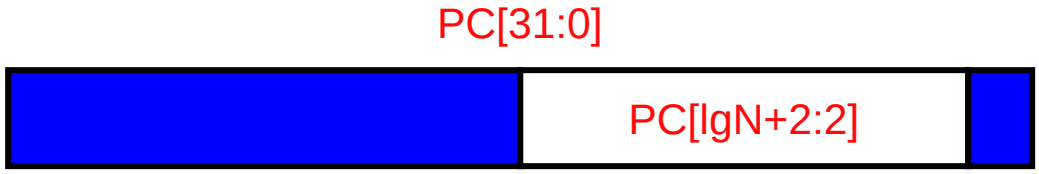
- **Dynamic branch prediction**: guess outcome
  - Start fetching from guessed address
  - Flush on **mis-prediction**

Inside A Branch Predictor



- Two parts
  - **Target buffer**: maps PC to taken target
  - **Direction predictor**: maps PC to taken/not-taken
- What does it mean to "map PC"?
  - Use some PC bits as index into an array of data items (like Regfile)

More About "Mapping PCs"



- If array of data has N entries
  - Need log(N) bits to index it
- Which log(N) bits to choose?
  - Least significant log(N) after the least significant 2, why?
  - LS 2 are always 0 (PCs are aligned on 4 byte boundaries)
  - Least significant change most often → gives best distribution
- What if two PCs have same pattern in that subset of bits?
  - Called **aliasing**
  - We get a nonsense target (intended for another PC)
  - That's OK, it's just a guess anyway, we can recover if it's wrong

Updating A Branch Predictor

- How do targets and directions get into branch predictor?
  - From previous instances of branches
  - Predictor "learns" branch behavior as program is running
    - Branch X was taken last time, probably will be taken next time

- Branch predictor needs a write port, too (not in my ppt)
  - New prediction written only if old prediction is wrong

Types of Branch Direction Predictors

- Predict same as last time we saw this same branch PC
  - 1 bit of state per predictor entry (take or don't take)
  - For what code will this work well? When will it do poorly?
- Use 2-level saturating counter
  - 2 bits of state per predictor entry
    - 11, 10 = take, 01, 00 = don't take
    - Why is this usually better?
- And every other possible predictor you could think of!
  - ICQ: Think of other ways to predict branch direction

- Dynamic branch prediction is one of most important problems in computer architecture

Branch Prediction Performance

- Same parameters
  - **Branch: 20%**, load: 20%, store: 10%, other: 50%
  - 75% of branches are taken
- Dynamic branch prediction
  - Assume branches predicted with 75% accuracy
  - CPI = 1 + 0.20*(0.25)*2 = **1.1**
- Branch (esp. direction) prediction was a hot research topic
  - Accuracies now 90-95%

Pipelining And Exceptions

- Remember exceptions?
  - Pipelining makes them nasty
  - 5 instructions in pipeline at once
  - Exception happens, how do you know which instruction caused it?
    - Exceptions propagate along pipeline in latches
  - Two exceptions happen, how do you know which one to take first?
    - One belonging to oldest insn
  - When handling exception, have to flush younger insns
    - Piggy-back on branch mis-prediction machinery to do this
- Just FYI – we'll solve this problem in ECE 552 (CS 550)

Pipeline Performance Summary

- Base CPI is 1, but hazards increase it
- Remember: nothing magical about a 5 stage pipeline
  - Pentium4 (first batch) had 20 stage pipeline
- Increasing **pipeline depth** (#stages)
  - \+ Reduces clock period (that's why companies do it)
  - – But increases CPI
  - Branch mis-prediction penalty becomes longer
    - More stages between fetch and whenever branch computes
  - Non-bypassed data hazard stalls become longer
    - More stages between register read and write
  - At some point, CPI losses offset clock gains, question is when?

Instruction-Level Parallelism (ILP)

- Pipelining: a form of **instruction-level parallelism (ILP)**
  - Parallel execution of insns from a single sequential program
- There are ways to exploit ILP
  - We'll discuss this a bit more at end of semester, and then we'll really cover it in great depth in ECE 552 (CS 550)
- We'll also talk a bit about thread-level parallelism (TLP) and how it's exploited by multithreaded and multicore processors

Summary

- Principles of pipelining
  - Pipelining a datapath and controller
  - Performance and pipeline diagrams
- Data hazards
  - Software interlocks and code scheduling
  - Hardware interlocks and stalling
  - Bypassing
- Control hazards
  - Branch prediction

Next up: Multicore Processors