ECE 350: Digital Systems

Implementation of Computer Hardware

Lecture 10: Pipeline

Brief review of ISAs

- MIPS-like ISA (load-store) with displacements (immediates)
- RISCs (Reduced Instruction Set Computers) require compositions of simple instructions to implement interesting control flows
  - Think of a for/while loop in an HLL; the work is broken up among many instructions
  - Pointer manipulations are all-you
  - Data management is again, all-you

Instruction Formats

Inst' Type	Instruction Format						
R	Opcode [31:27]	RD [26:22]	RS [21:17]	RT [16:12]	shiftamt [11:7]	ALUop [6:2]	Zeros [1:0]
I	Opcode [31:27]	RD [26:22]	RS [21:17]	Immediate [16:0]			
J1	Opcode [31:27]	Target [26:0]					
J2	Opcode [31:27]	RD [26:22]					

The ECE 350 ISA

Inst’	Opcode (ALU Op)	Type	Usage	Operation
add	00000 (00000)	R	add $rd, $rs, $rt	$rd = $rs + $rt
addi	00101	I	addi $rd, $rs, N	$rd = $rs + N
sub	00000 (00001)	R	sub $rd, $rs, $rt	$rd = $rs - $rt
and	00000 (00010)	R	and $rd, $rs, $rt	$rd = $rs AND $rt
or	00000 (00011)	R	or $rd, $rs, $rt	$rd = $rs OR $rt
sll	00000 (00100)	R	sll $rd, $rs, shamt	$rd = $rs << shamt
sra	00000 (00101)	R	sra $rd, $rs, shamt	$rd = $rs / 2^shamt
mul	00000 (00110)	R	mul $rd, $rs, $rt	$rd = $rs * $rt (32b X 32b);
div	00000 (00111)	R	div $rd, $rs, $rt	$rd = $rs / $rt (32b ÷ 32b);

The ECE 350 ISA

j	00001	J1	j N	PC = N
bne	00010	I	bne $rd, $rs, N	if($rd != $rs) PC = PC+1+N
jal	00011	J1	jal N	$r31 = PC+1; PC=N
jr	00100	J2	jr $rd	PC = $rd
blt	00110	I	blt $rd, $rs, N	if($rd < $rs) PC=PC+1+N
sw	00111	I	sw $rd, N($rs)	MEM[$rs + N] = $rd
lw	01000	I	lw $rd, N($rs)	$rd = MEM[$rs + N]

ISA Notes

- I-type immediate field [16:0] is signed 2's complement and sign-extended to the full 32-bit word size.
- J-type target field [26:0] is extended to the full 32-bit PC size using the upper bits from the current PC+1.
- Fields that are undefined are filled with zeroes by the assembler.
- Register $r0 always equals zero. Registers $r1 through $r30 are general purpose. Register $r31 stores the link address of a jump-and-link instruction.
- Memory is word-addressed. The instruction and data memory address spaces are separate. Data begins at data memory address zero.
- After a reset, all register values are zero and program execution begins from instruction memory address zero. The memory's contents are not reset.

The "Assembulator"

- The ECE 350 assembler
  - Takes assembly (MIPS-like) and produces machine code
  - Updated each semester to reflect the current ISA
  - Exports to a .mif file which can be used by Quartus to populate a memory block on the FPGA
- General processor procedure:
  1. Write and test an assembly program using the Assembulator
  2. Export the instruction memory and data memory files (.mif)
  3. Load these memory files into your Quartus project
  4. Run your processor (functional simulation) and examine outputs (e.g. sw data and addresses)

HLL/Assembly Comparison

HLL Example

```
x = 1;
y = 5;
while (y>=x)
{
   x = x+1;
   ...
}
```

Assembly Example

```
lw $r1, 0($r0)
lw $r2, 1($r0)
loop:  blt  $r2, $r1, done
       addi $r1, $r1, 1
       ...
       j loop
done: halt
```

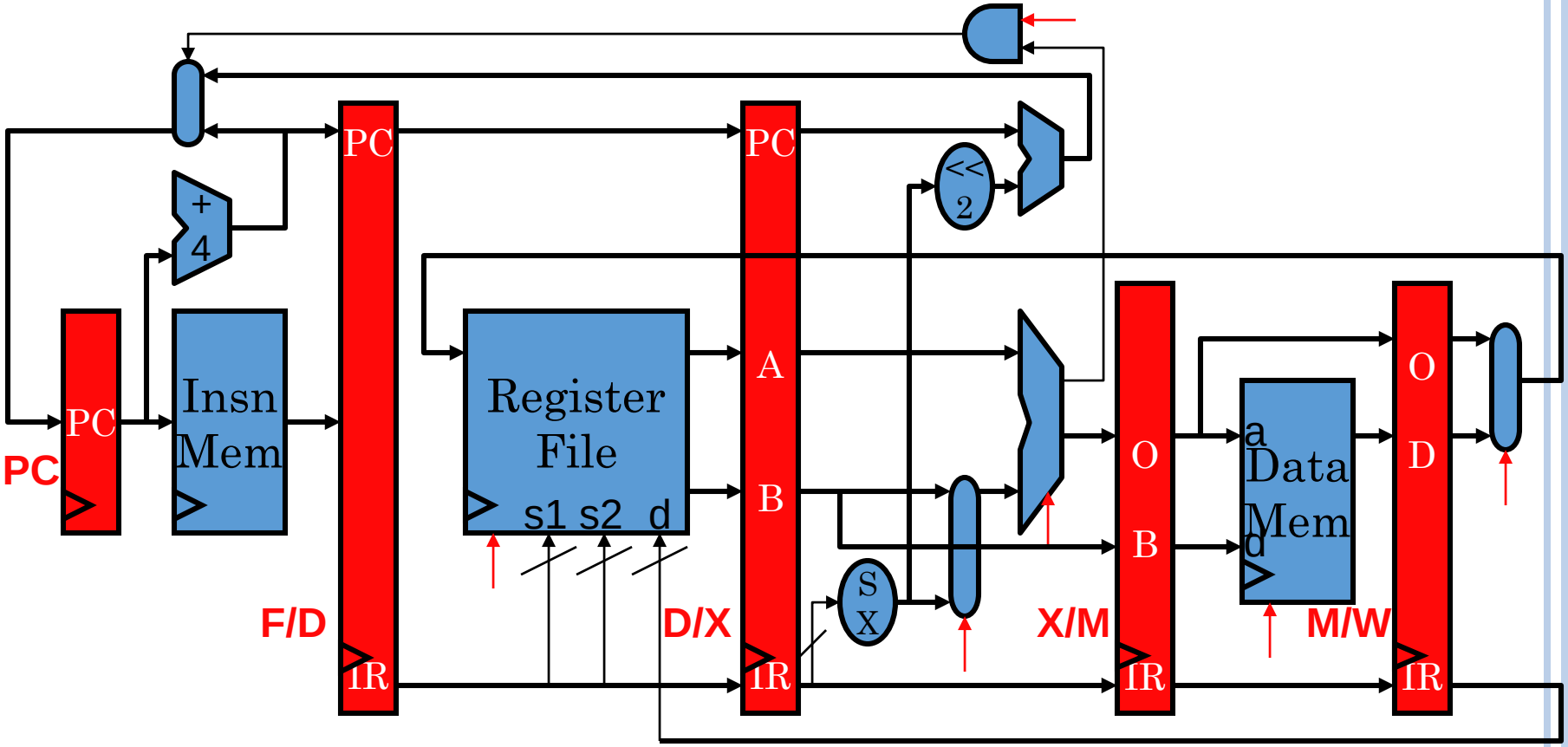
DMEM

Address	Data
0	1
1	5

Concurrency

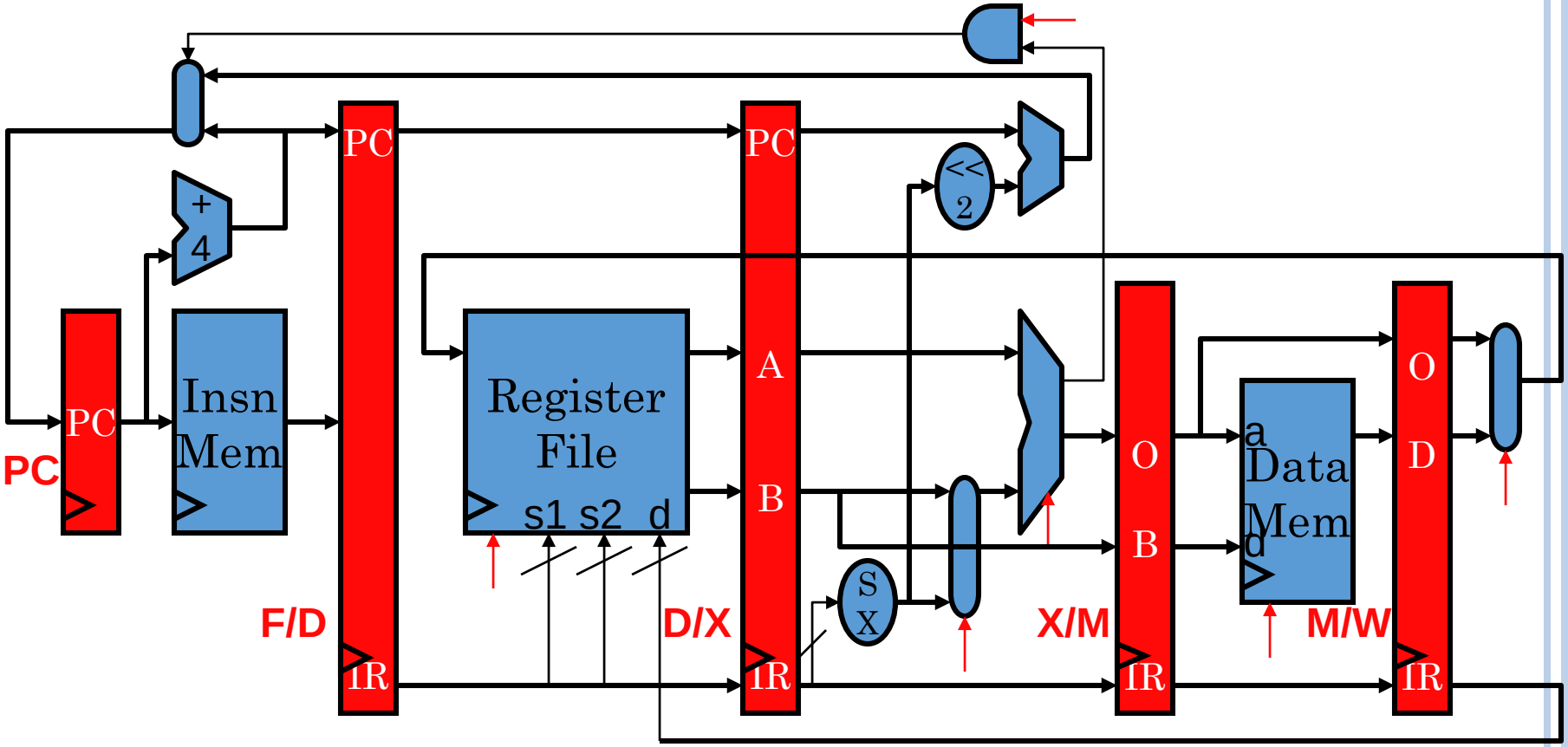
- How do we design machines to perform more work per unit time?
  - Pipelines (aka, assembly lines!)
  - Multiple cores (single threaded, or otherwise)
  - Distributed systems
- Basic idea: divide the work into stages
  - Each stage performs a fixed amount of work
  - Intermediate values are passed along every cycle

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**

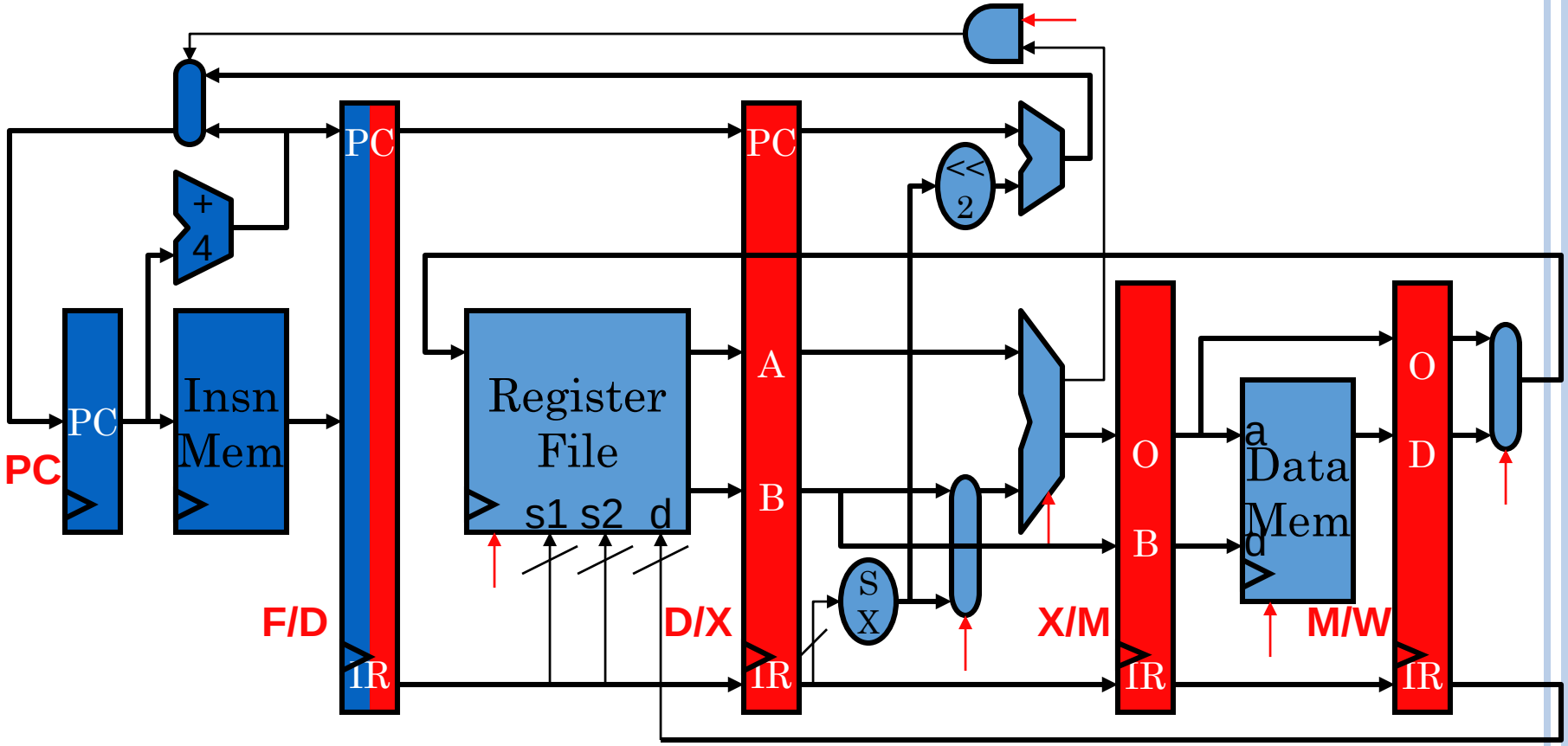
Pipeline Example: Cycle 1




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

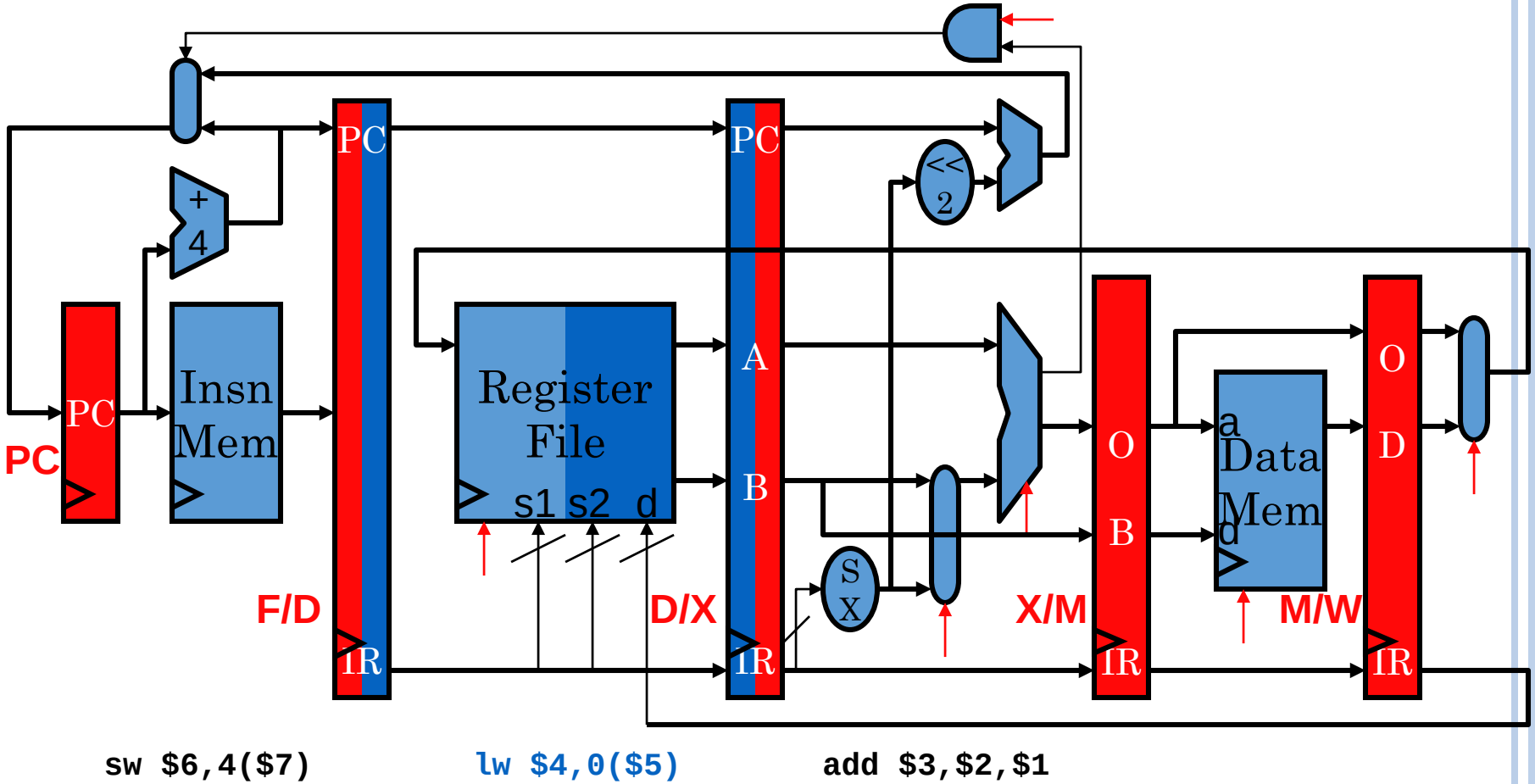
3 instructions: add, lw, sw

Pipeline Example: Cycle 2



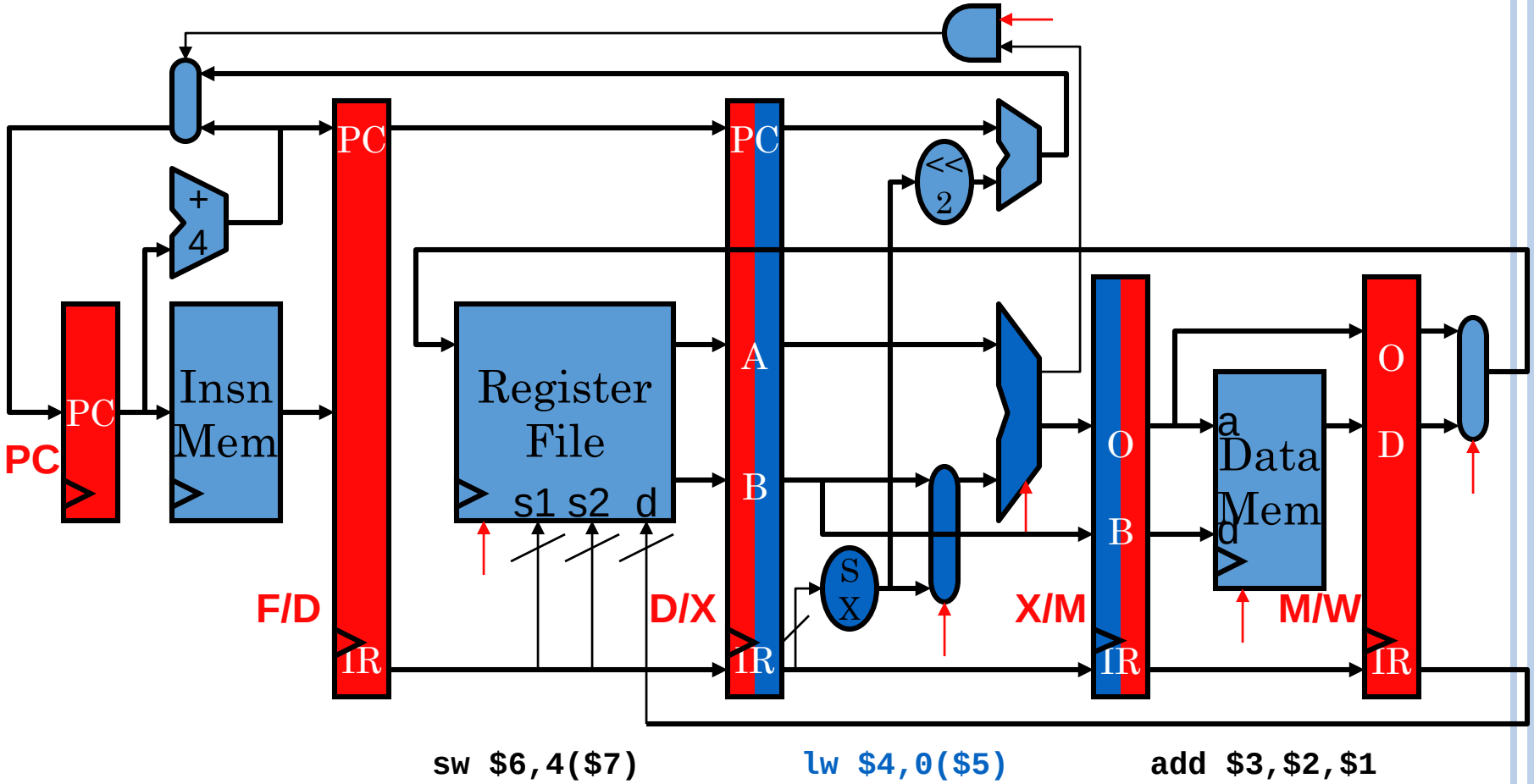
3 instructions: add, **lw**, sw

Pipeline Example: Cycle 3

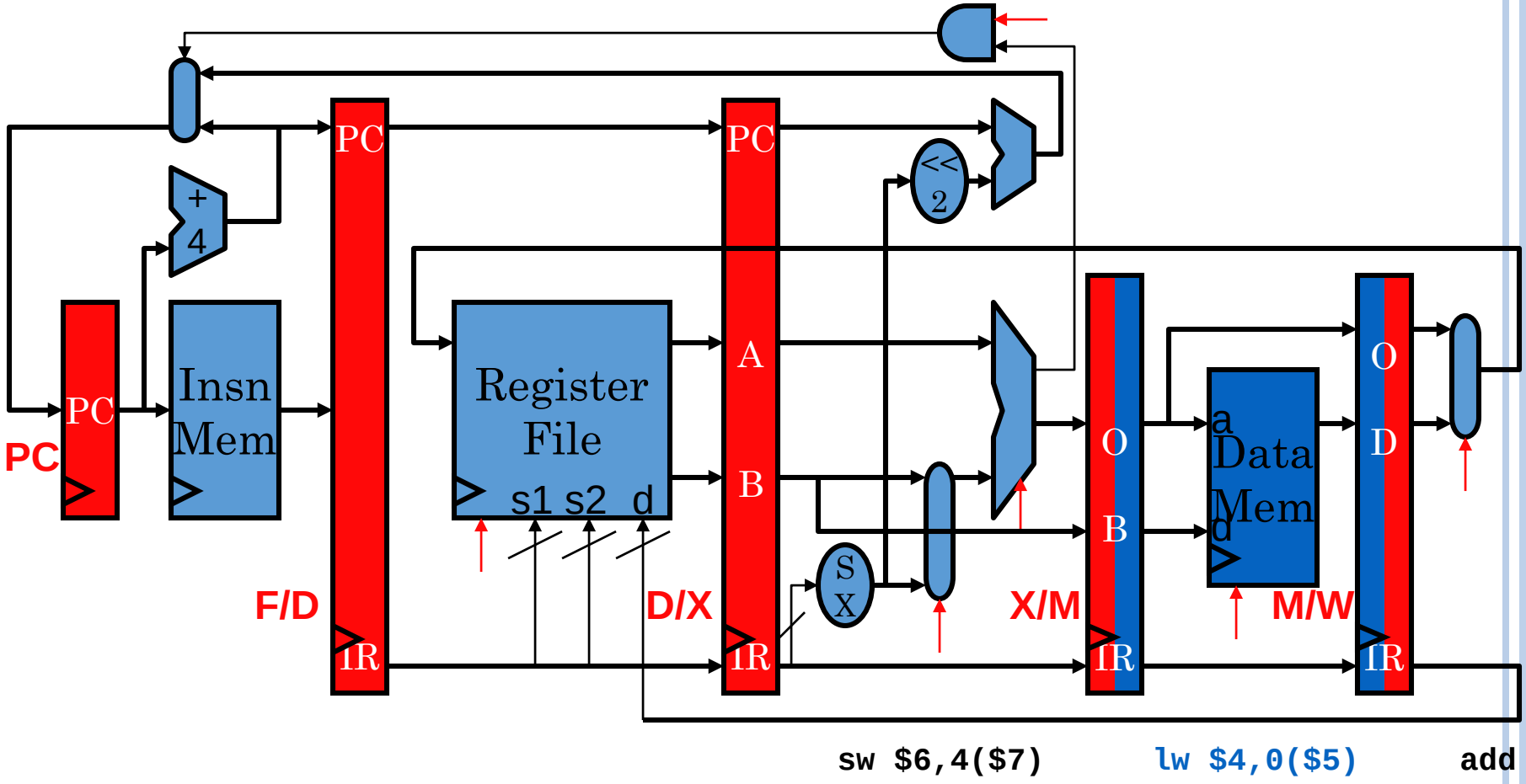


3 instructions: add, lw, **sw**

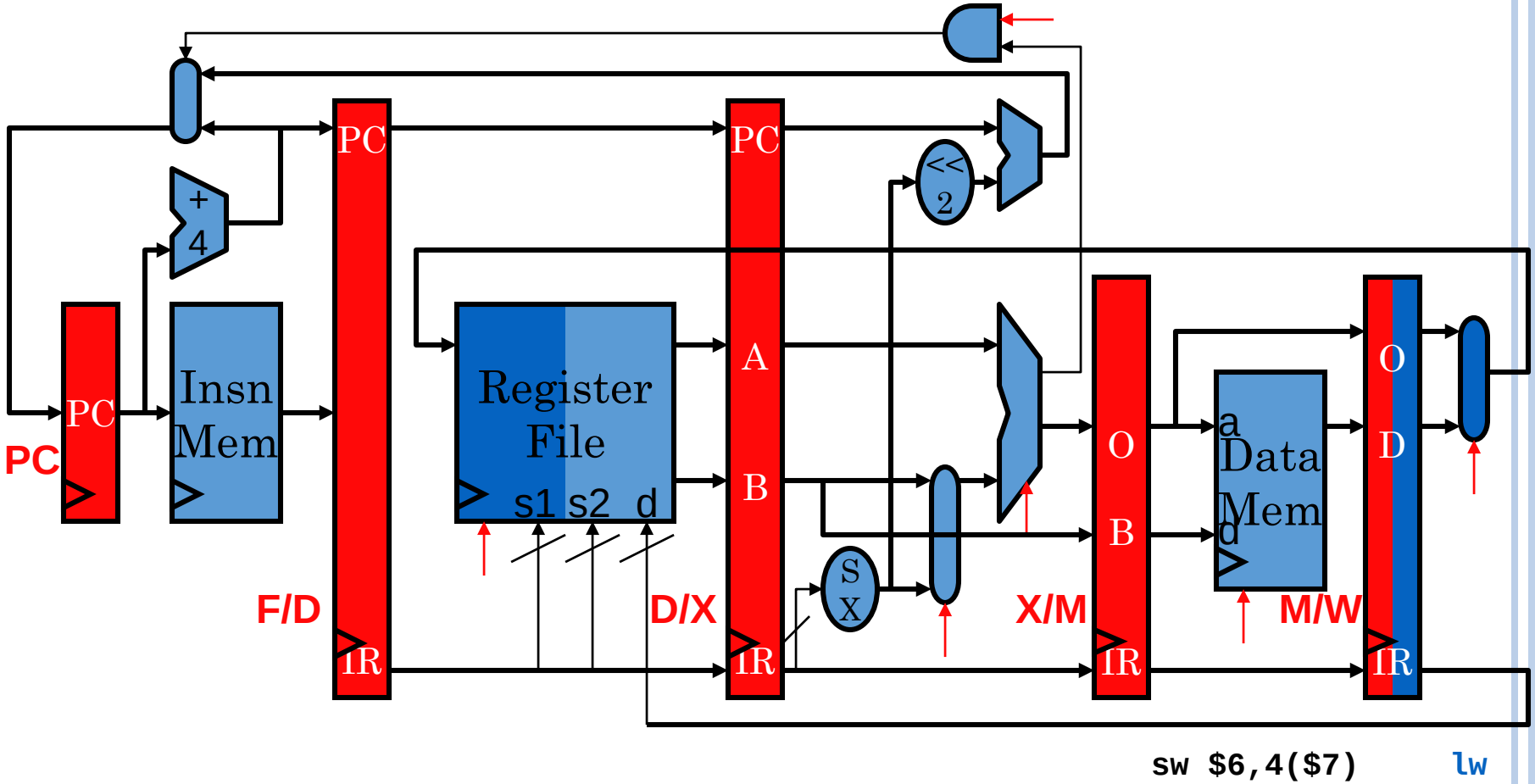
Pipeline Example: Cycle 4



Pipeline Example: Cycle 5

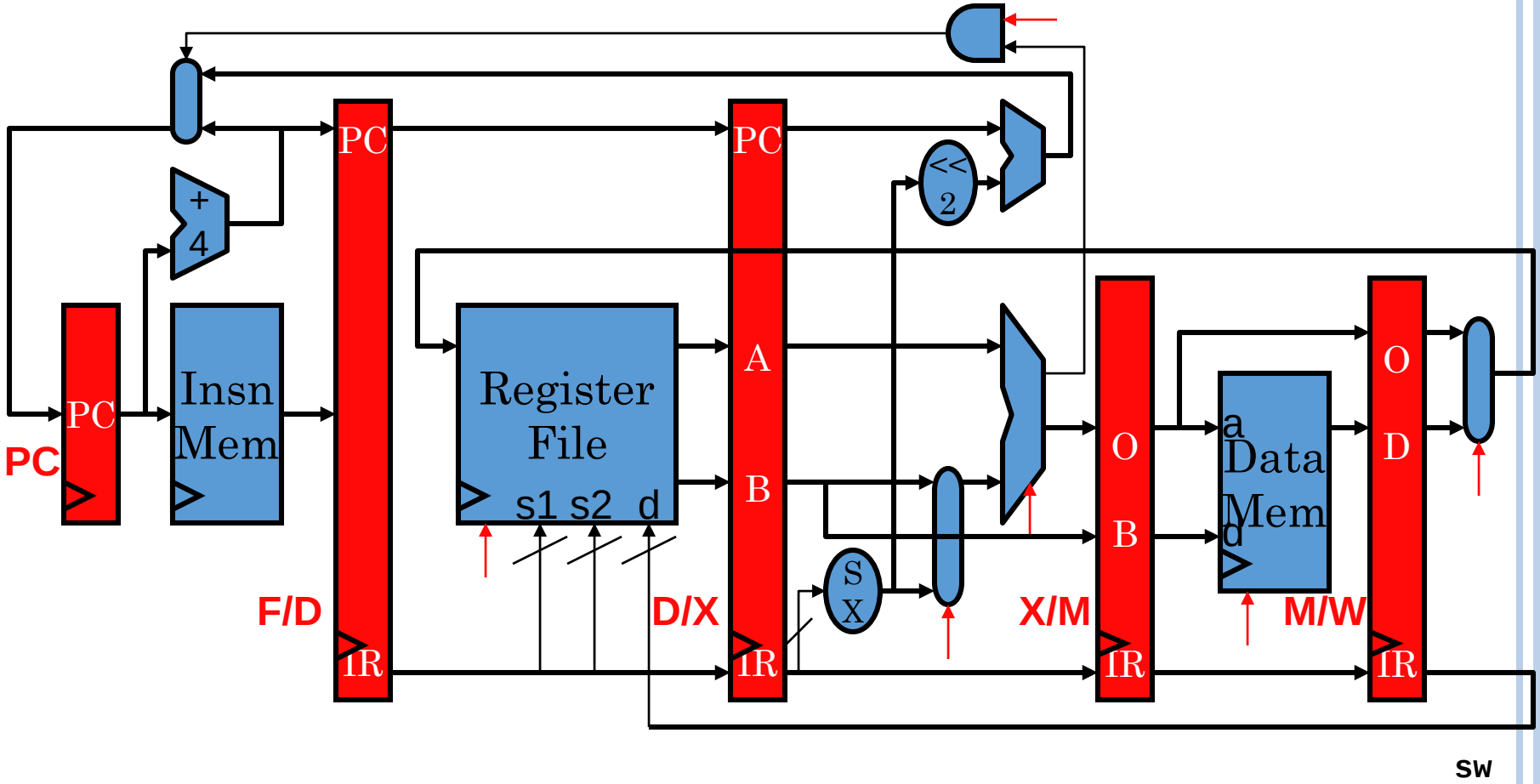


Pipeline Example: Cycle 6

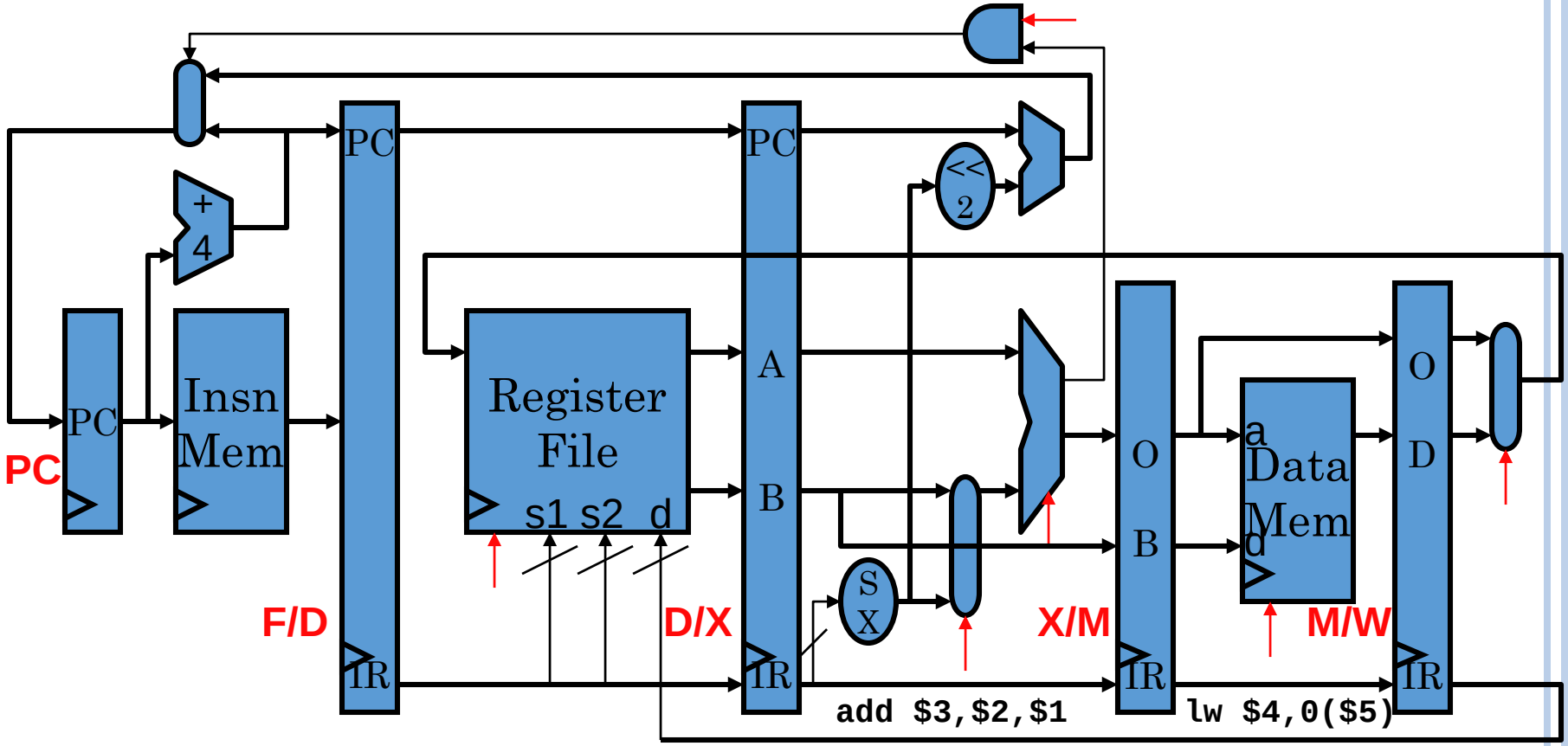


How can we ensure an inst. in DECODE reads a value from WRITEBACK?

Pipeline Example: Cycle 7



Why Does Every INSN Take 5 Cycles?

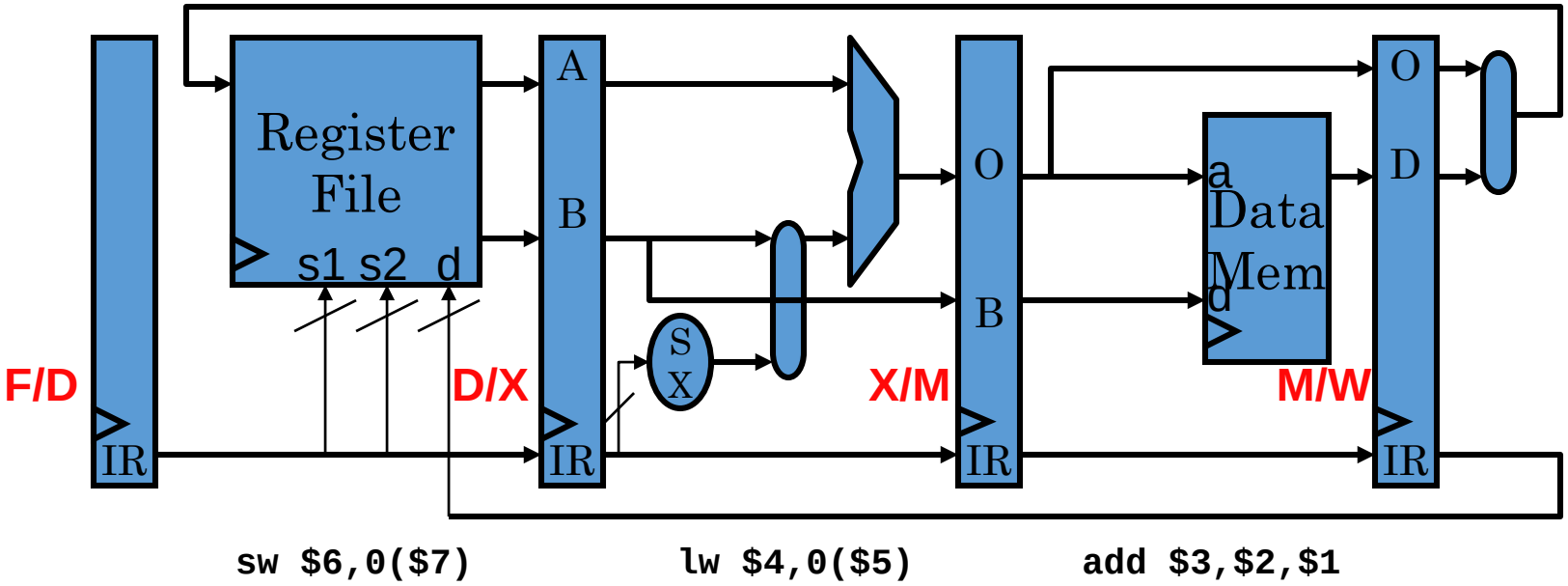


- Why not let **add** skip M and go straight to W?
  - It wouldn't help: peak fetch still only 1 insn per cycle
  - **Structural hazards**: not enough resources per stage for 2 insns

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
- Control hazards

Data Hazards



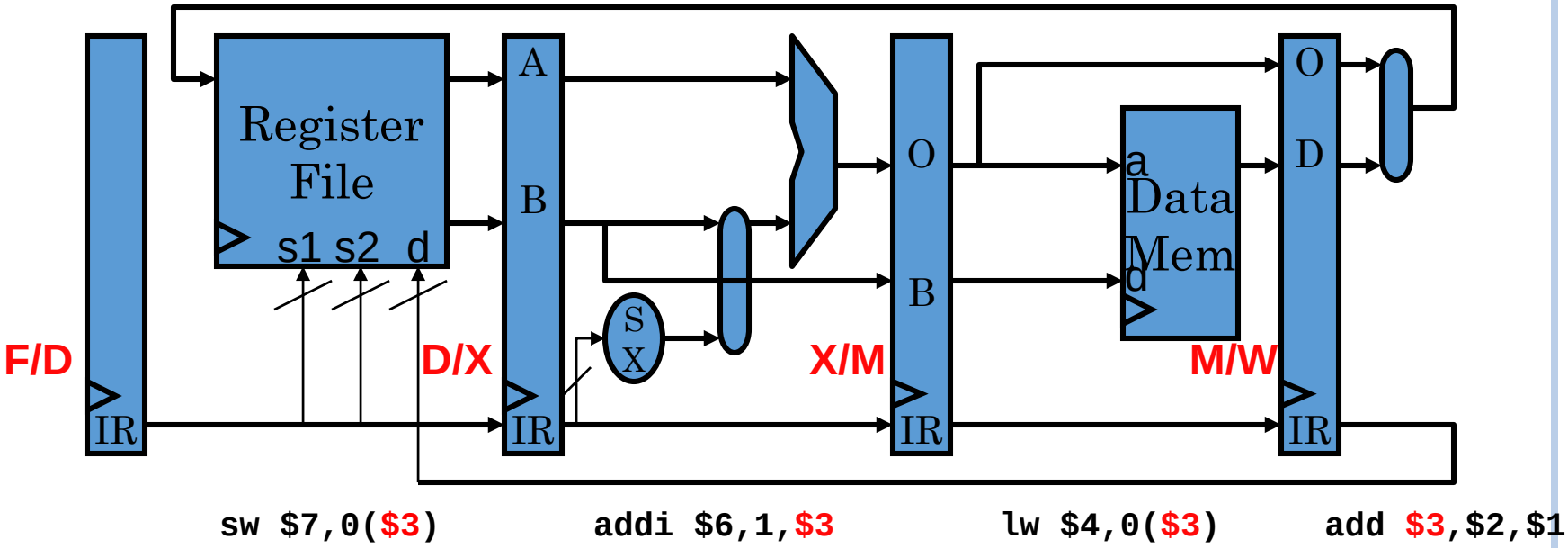
- Let's forget about branches and control for a while
- Real programs have **data dependences**
  - They pass values via registers and memory

Data Hazards

- Consider this program fragment

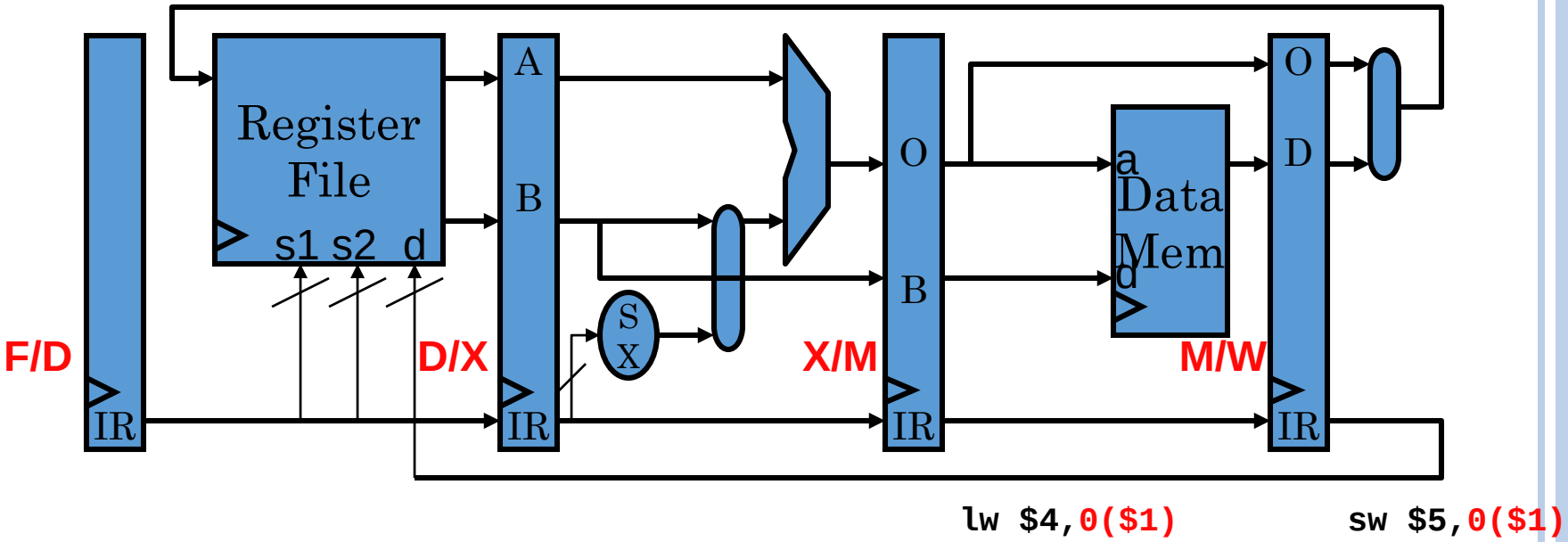
```
add $3,$2,$1
lw $4,0($3)
addi $6,1,$3
sw $3,0($7)
```

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

Data Hazards

- RAW – Read after Write (the reg and mem examples we have just considered)
  - Hazard if write can occur in later pipeline stage than read, as for registers here – stale value read
- WAW – Write after Write
  - If later instruction's write can complete before earlier one, wrong value persists – not possible here, but can happen if more than one stage can write

Data Hazards

- WAR – Write after read
  - Hazard if some instructions can write early in pipeline and others can read late – can't happen here but is real on other architectures
- RAR – Read after Read
  - ??

Fixing Register Data Hazards

- In our architecture, can only legally read register value 3 cycles after writing it
- One way to enforce this: make sure programs can't do it
  - Compiler (or alert ECE350 student programmer) puts two **independent** insns between write/read insn pair
    - If they aren't there already, insert them!
  - Independent means: "do not interfere with register in question"
    - Do not write it: otherwise meaning of program changes
    - Do not read it: otherwise creates a new data hazard
  - **Code scheduling**: compiler moves around existing insns to do this
  - If none can be found, must use **NOPs**
  - This is called a system with **software interlocks**

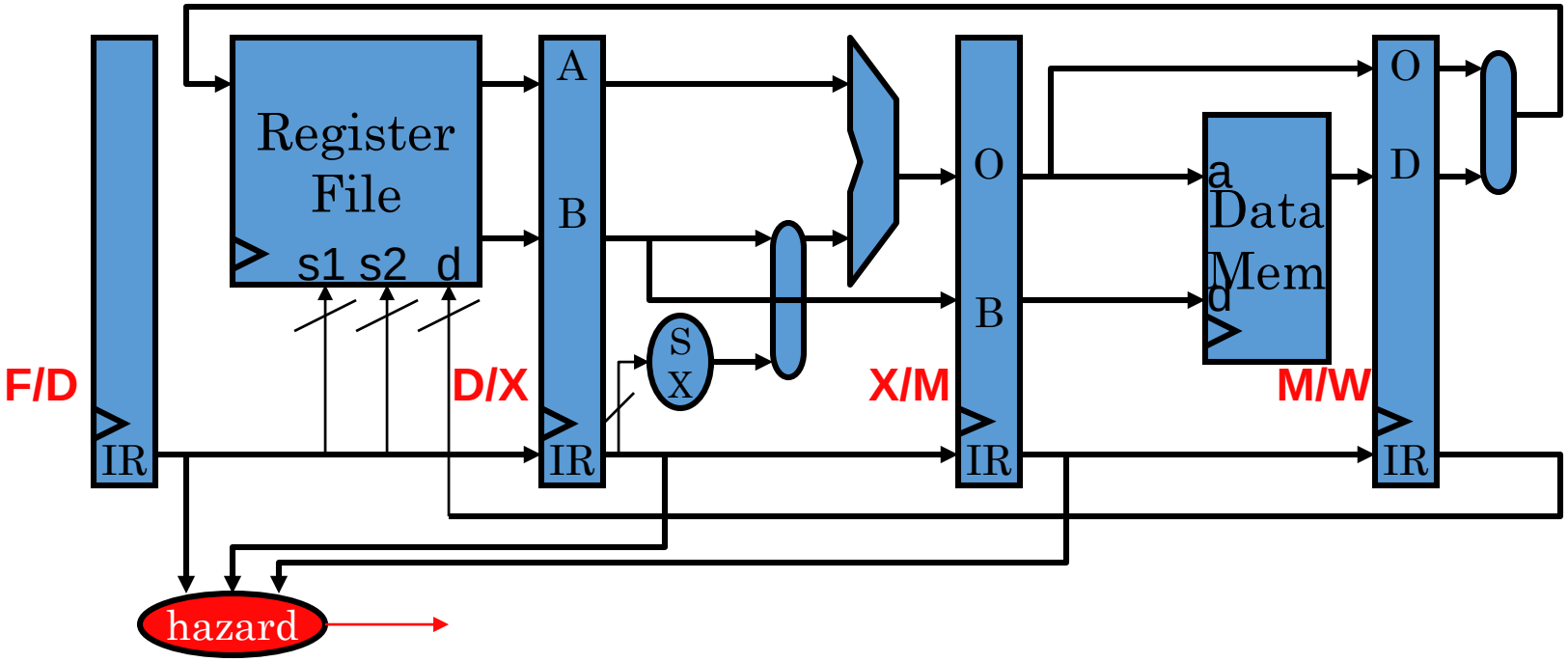
MIPS: **M**icroprocessor w/out **I**nterlocking (hardware) **P**ipeline **S**tages

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, thus, violating backward compatibility requirement!

- 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

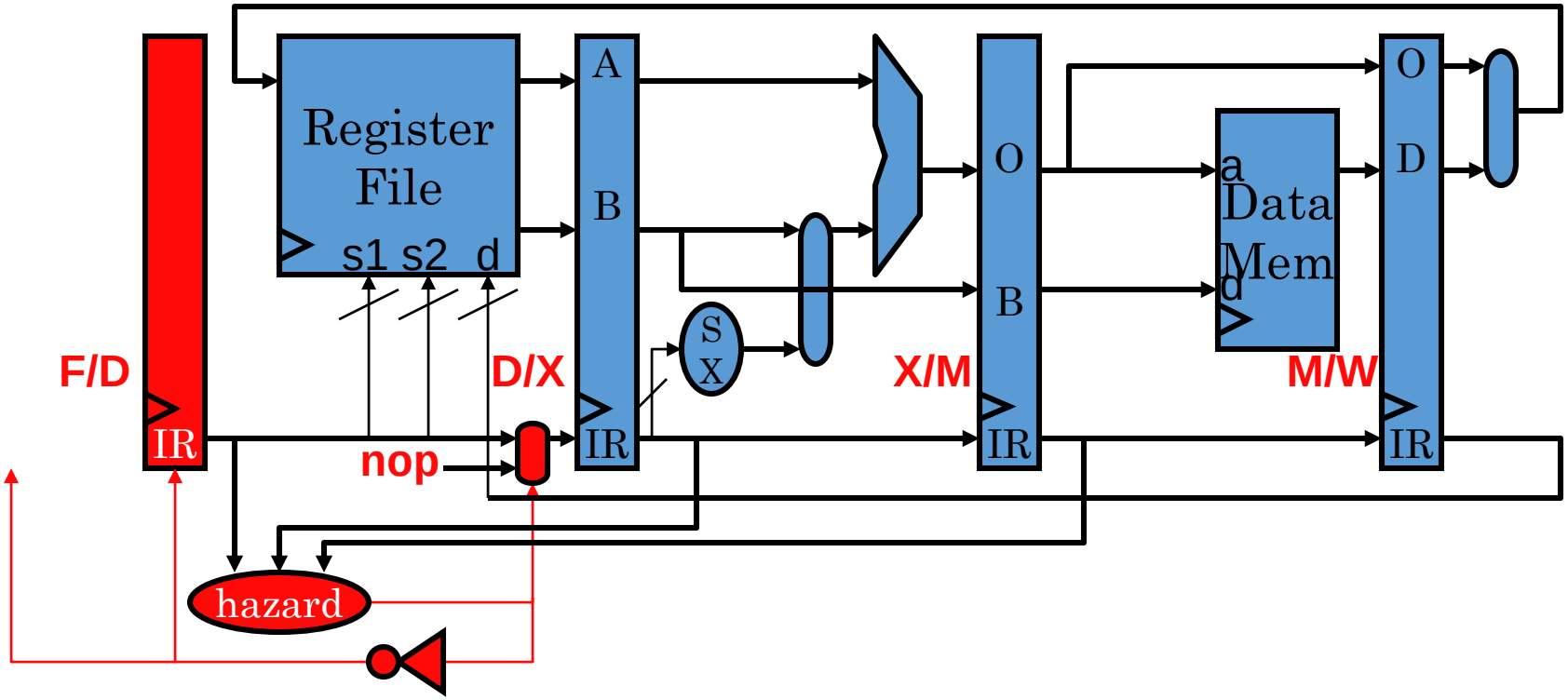


Compare F/D insn input register names with output register names of older insns in pipeline

Hazard =

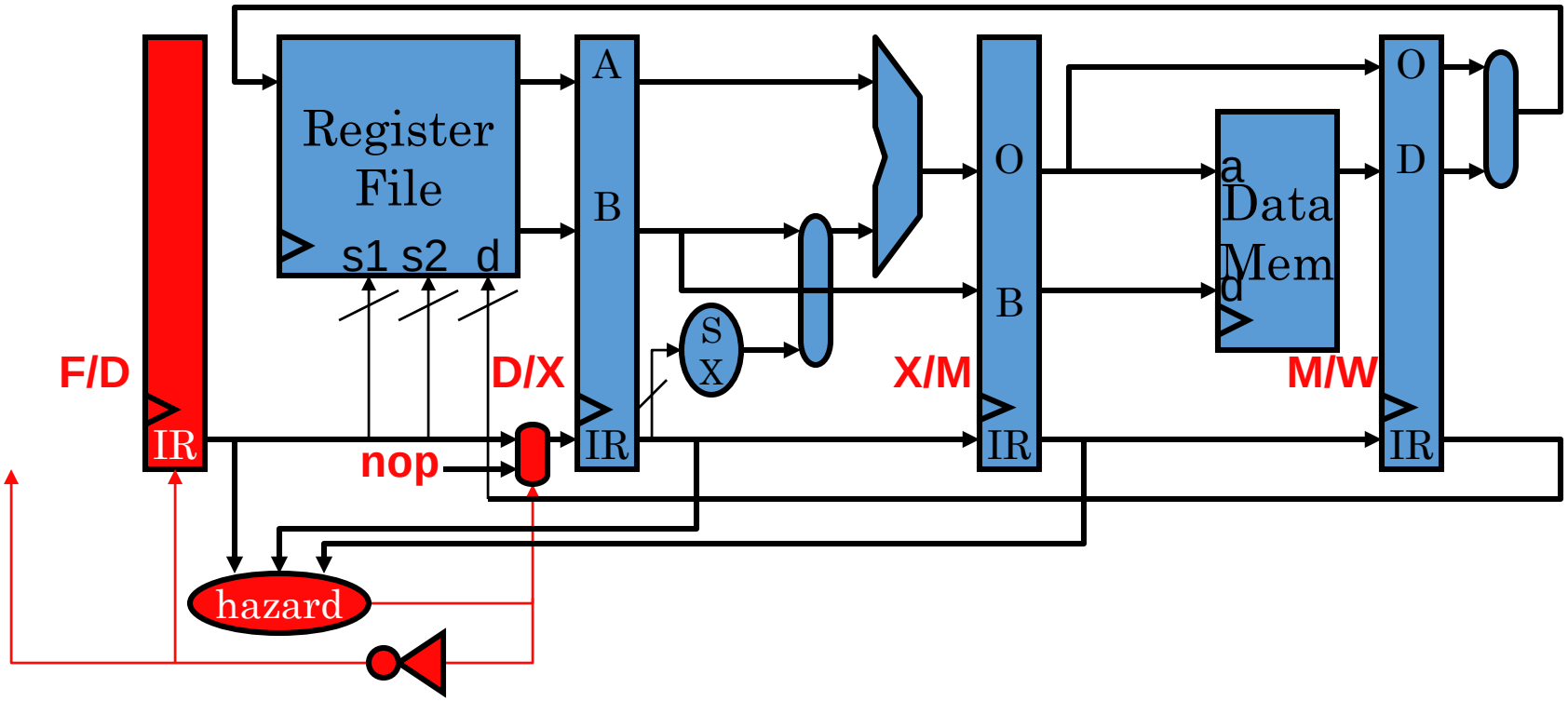
(F/D.IR.RS1 == D/X.IR.RD) || (F/D.IR.RS2 == D/X.IR.RD) ||
(F/D.IR.RS1 == X/M.IR.RD) || (F/D.IR.RS2 == X/M.IR.RD)

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

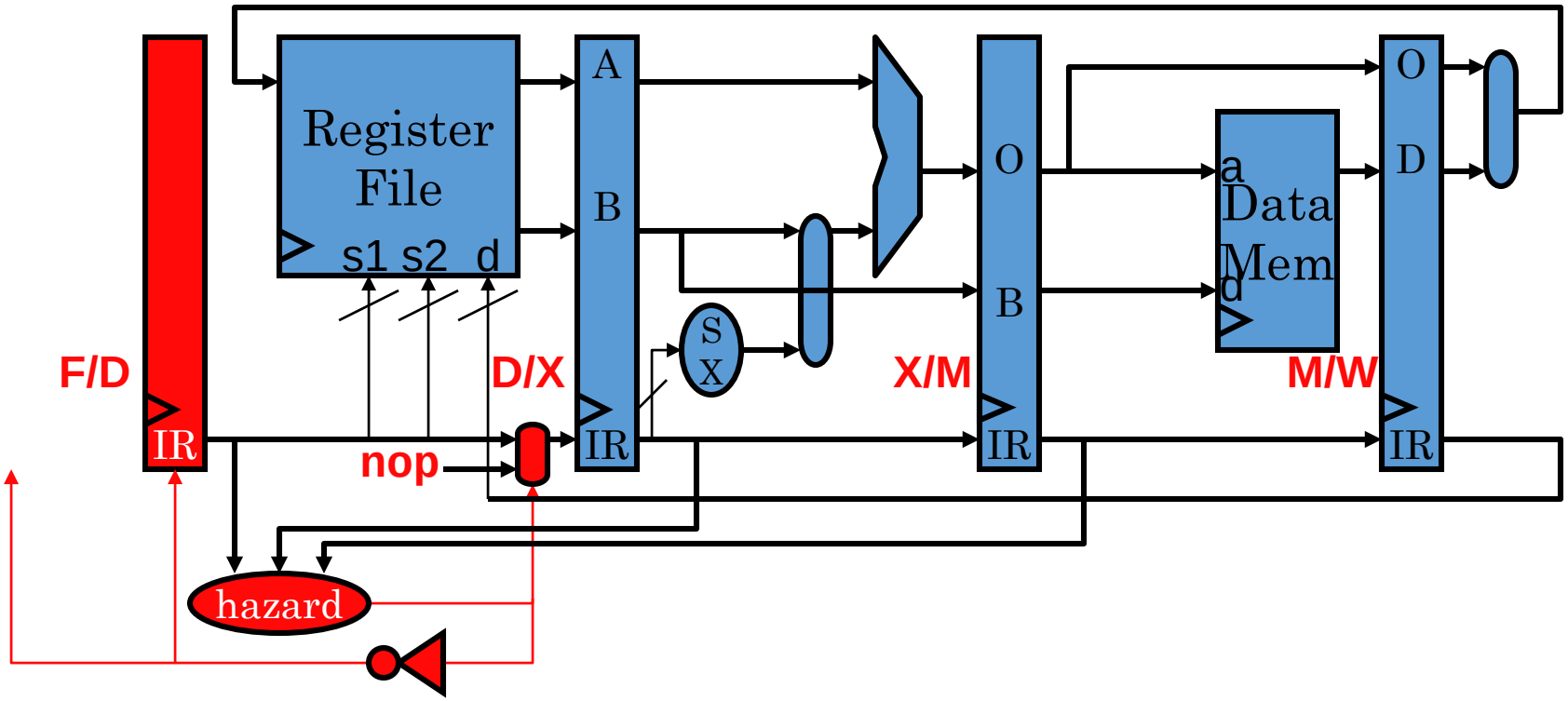
Hardware Interlock Example: cycle 1



(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



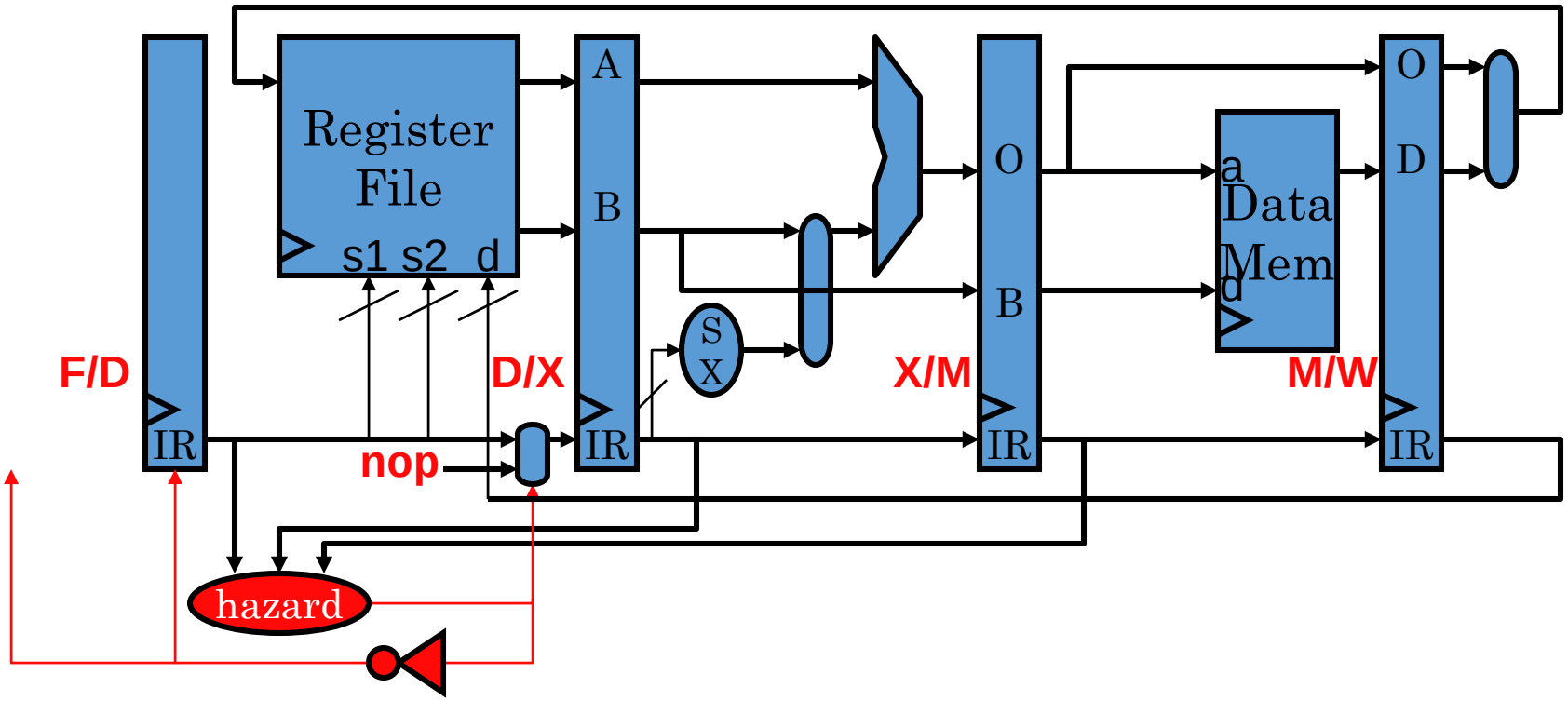
`lw $4,0($3)` `add $3,$2,$1`

(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**)

= **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)

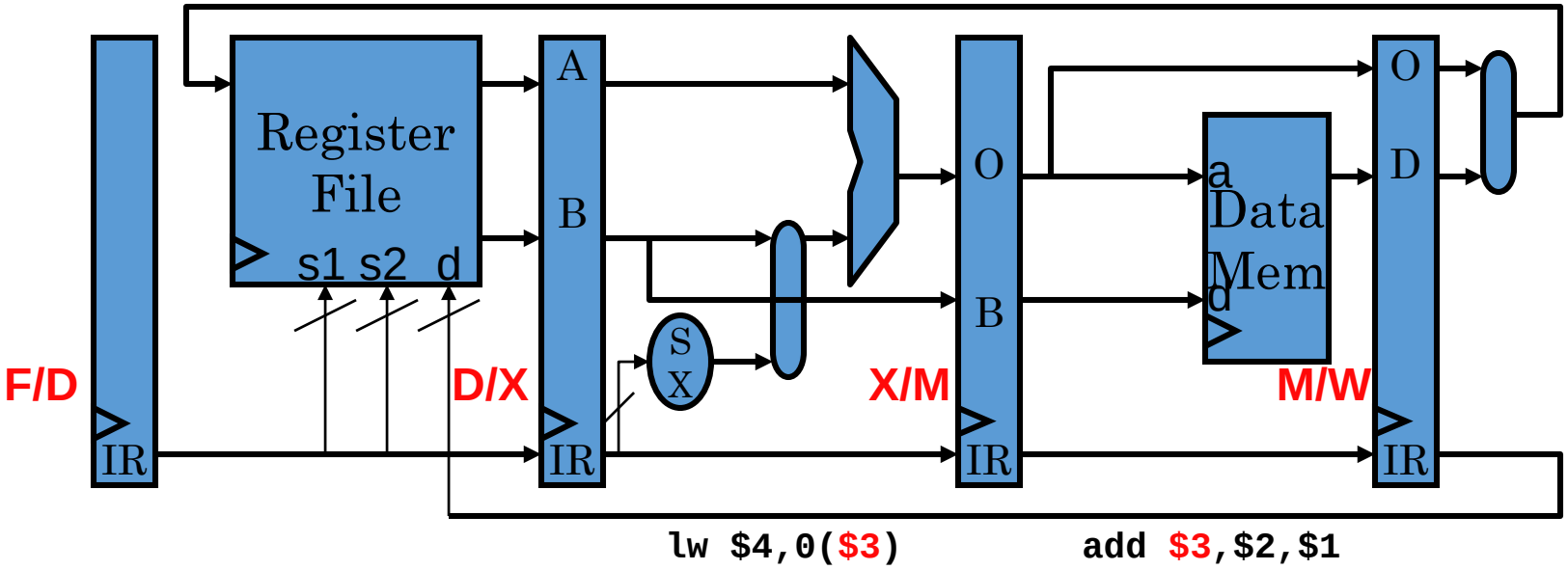
= 0

Pipeline Control Terminology

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

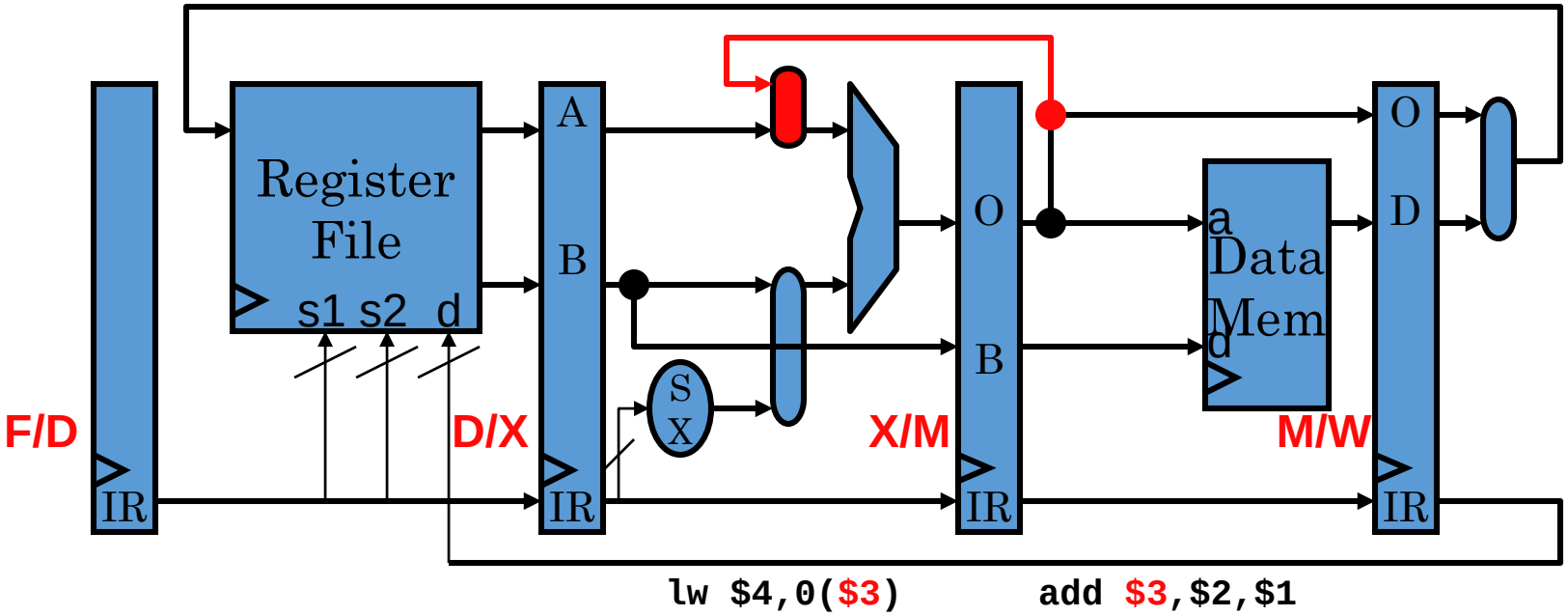
- Mechanism is called **stall logic**
- Part of more general **pipeline control mechanism**
  - Controls advancement of insns through pipeline
- Distinguished from **pipelined datapath control**
  - Controls datapath at each stage
  - Pipeline control controls advancement of datapath control

Improving interlocks…



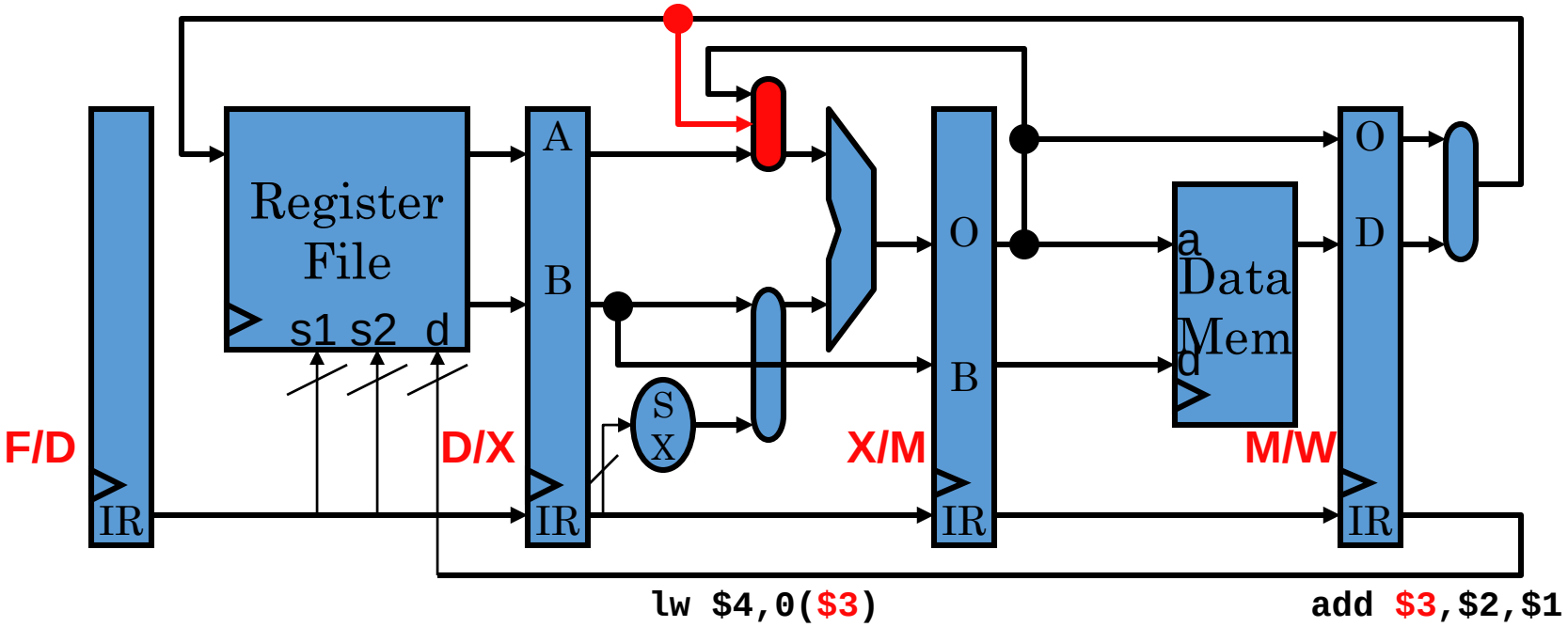
- We know this situation is broken:
  - **`lw $4,0($3)`** has already read **$3** from regfile
  - **`add $3,$2,$1`** hasn't yet written **$3** to regfile
- But fundamentally, everything is still OK
  - **`lw $4,0($3)`** hasn't actually **used $3 yet!**
  - **`add $3,$2,$1`** has already computed **$3**

Bypassing



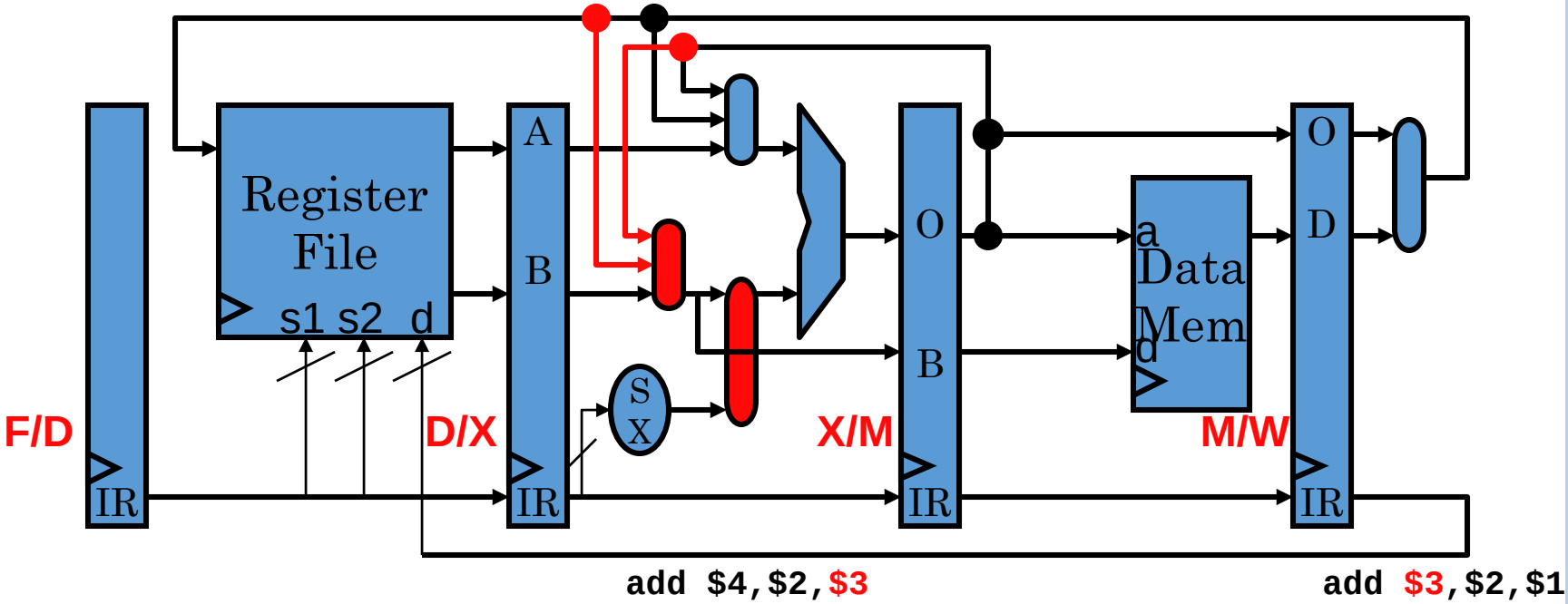
- **Bypassing**
  - Reading a value from an intermediate (m-architectural) source
  - Not waiting until it is available from primary source (RegFile)
  - Here, we are bypassing the register file
  - Also called **forwarding**

WX Bypassing



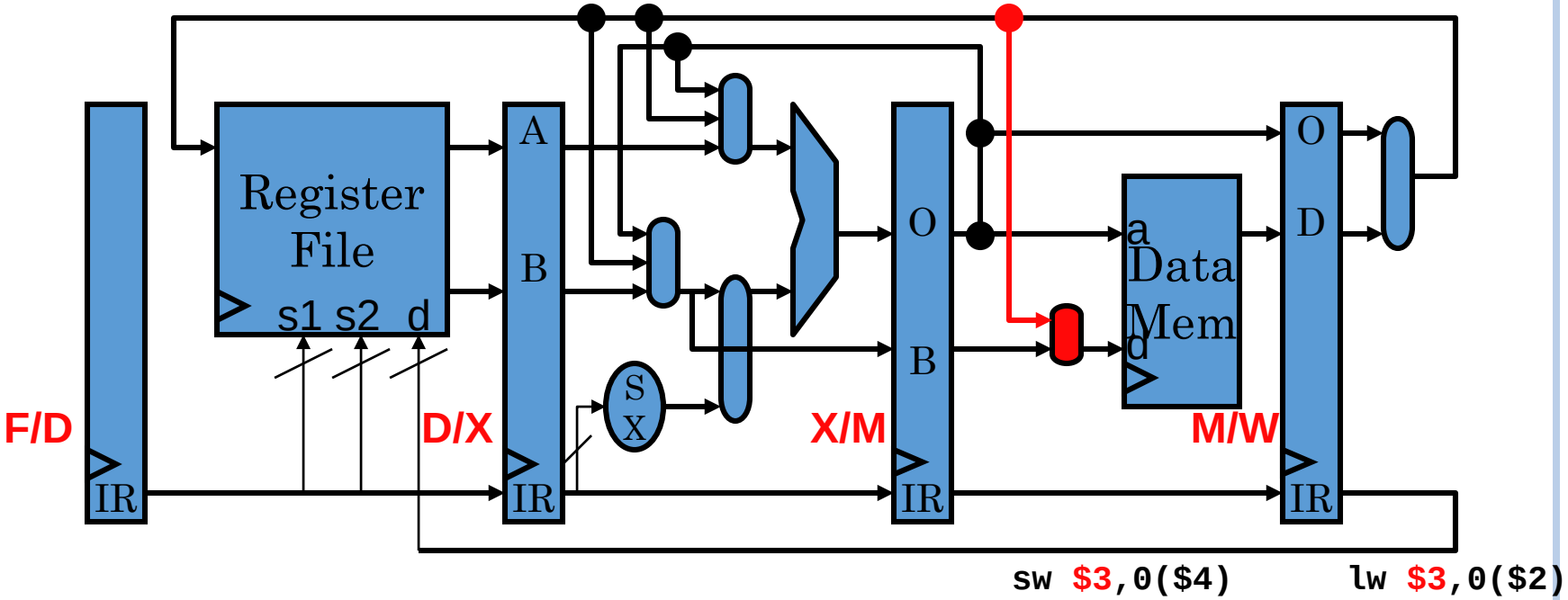
- What about this combination?
  - Add another bypass path and MUX input
  - First one was an **MX** bypass
  - This one is a **WX** bypass

ALUinB Bypassing



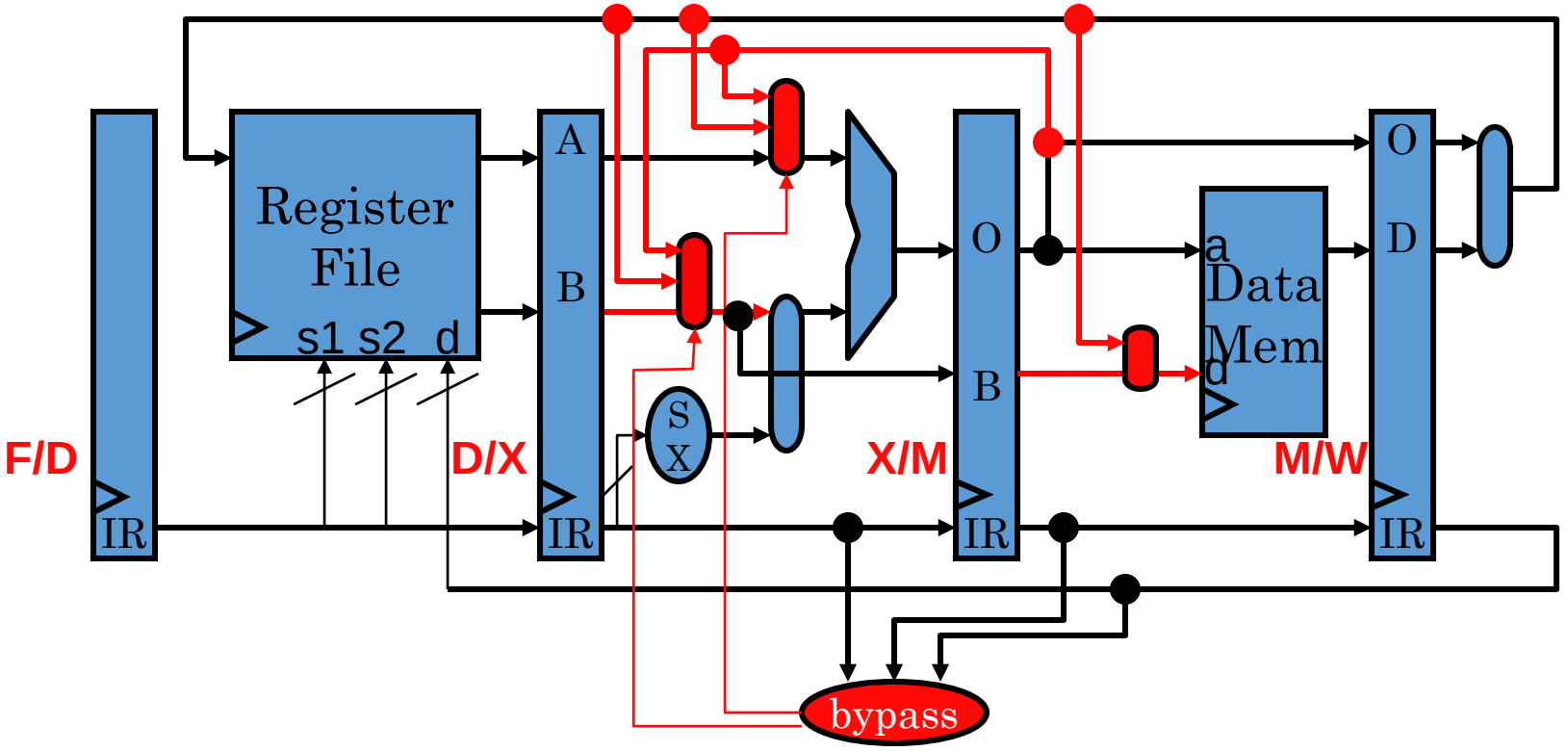
- Can also bypass to ALU input B

WM Bypassing?



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

Bypass Logic



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

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

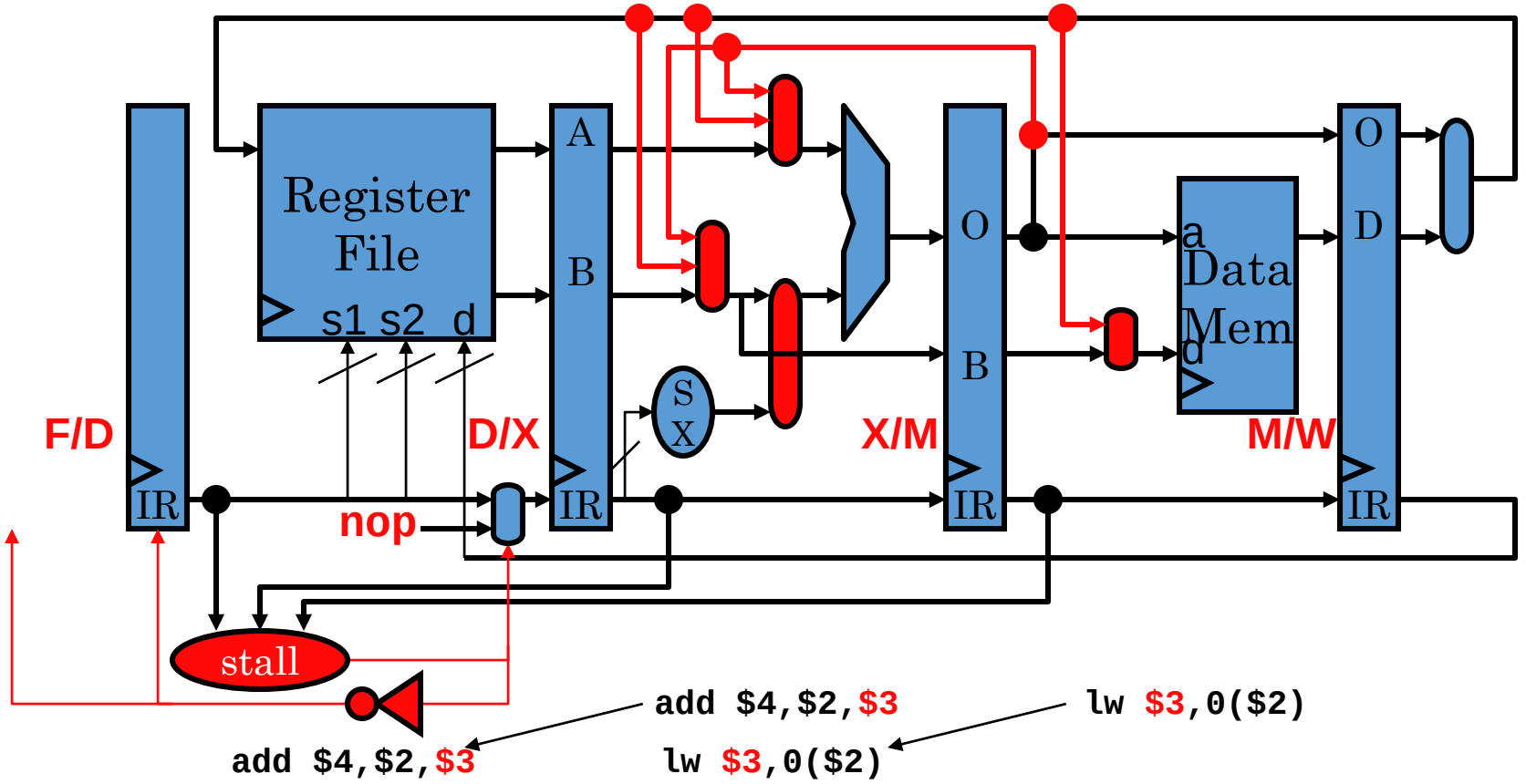
  (D/X.IR.RS1 == M/W.IR.RD) → mux select = 1

  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
- Previous slide shows **full bypassing**: all bypasses possible
  - Is stall logic still necessary?

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) && (F/D.IR.OP != STORE))

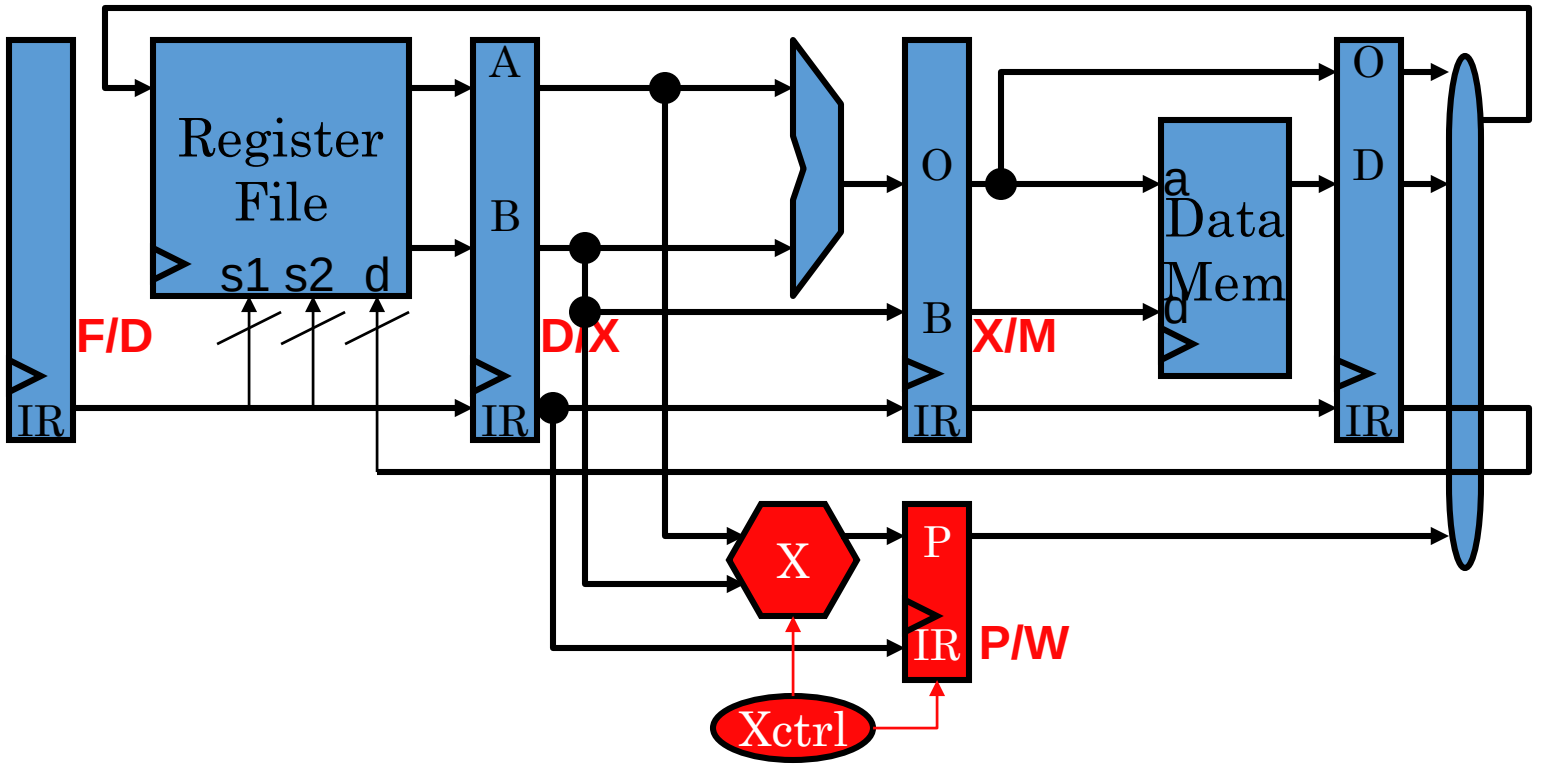
Pipeline Diagram With Bypassing

	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	

- 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

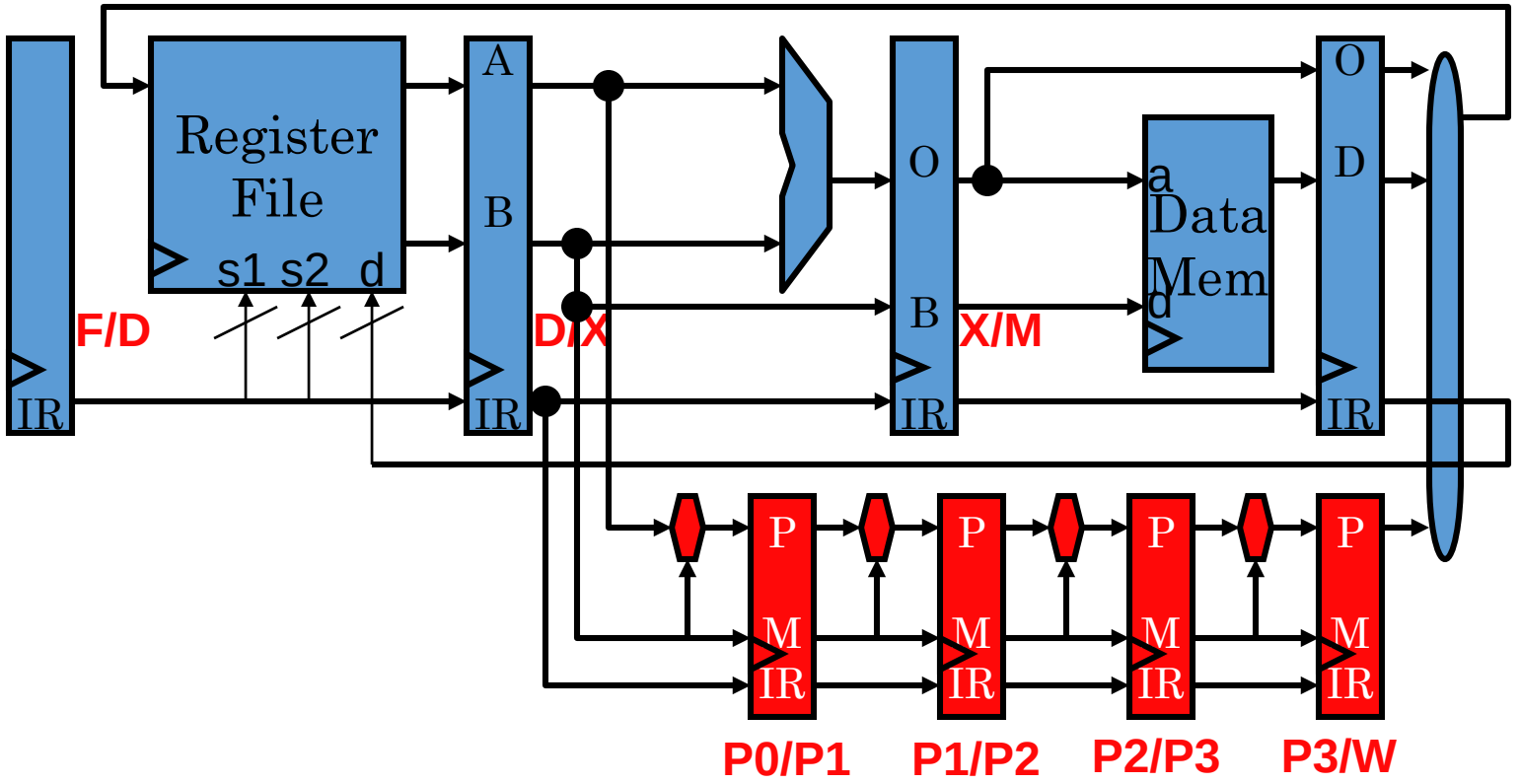
	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,0($3)`		F	D	X	M	W			
`addi $6,$4,1`			F	D	**d***	X	M	W	

Pipelining and Multi-Cycle Operations



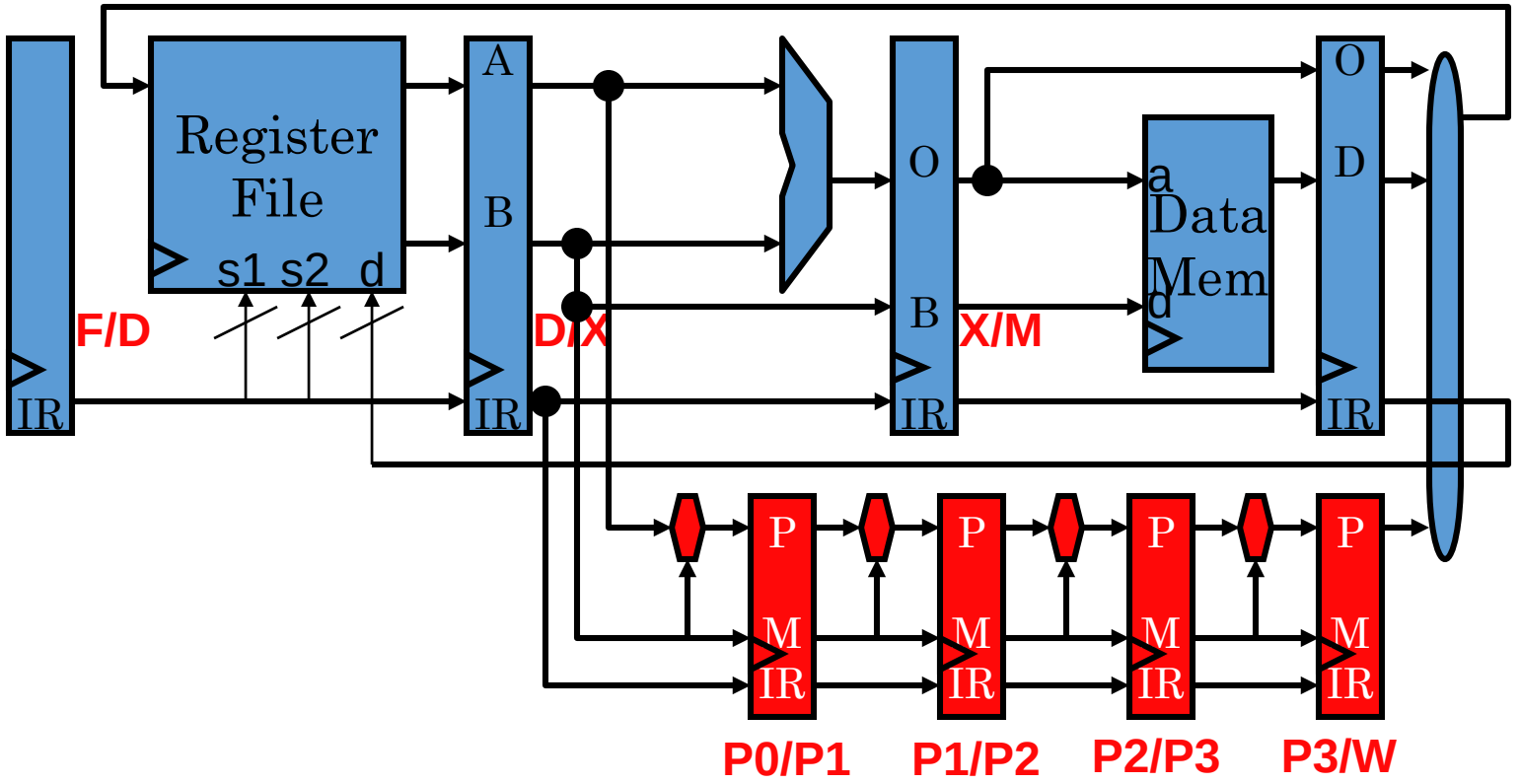
- What if you wanted to add a multi-cycle operation?
  - E.g., 4-cycle multiply (or 32, or a variable number > 1)
  - **P/W**: separate output latch connects to W stage (since output destined for Rfile)
  - Controlled by pipeline control and multiplier FSM
  - STALL All new instructions until Mul/Div completes – fixed count or dynamic
    - Finish prior instructions, Feed NOPs or freeze subsequent stages

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
    - Subject to normal register conflict constraints
    - But not other instructions – smart compiler helps here

What about Stall Logic?



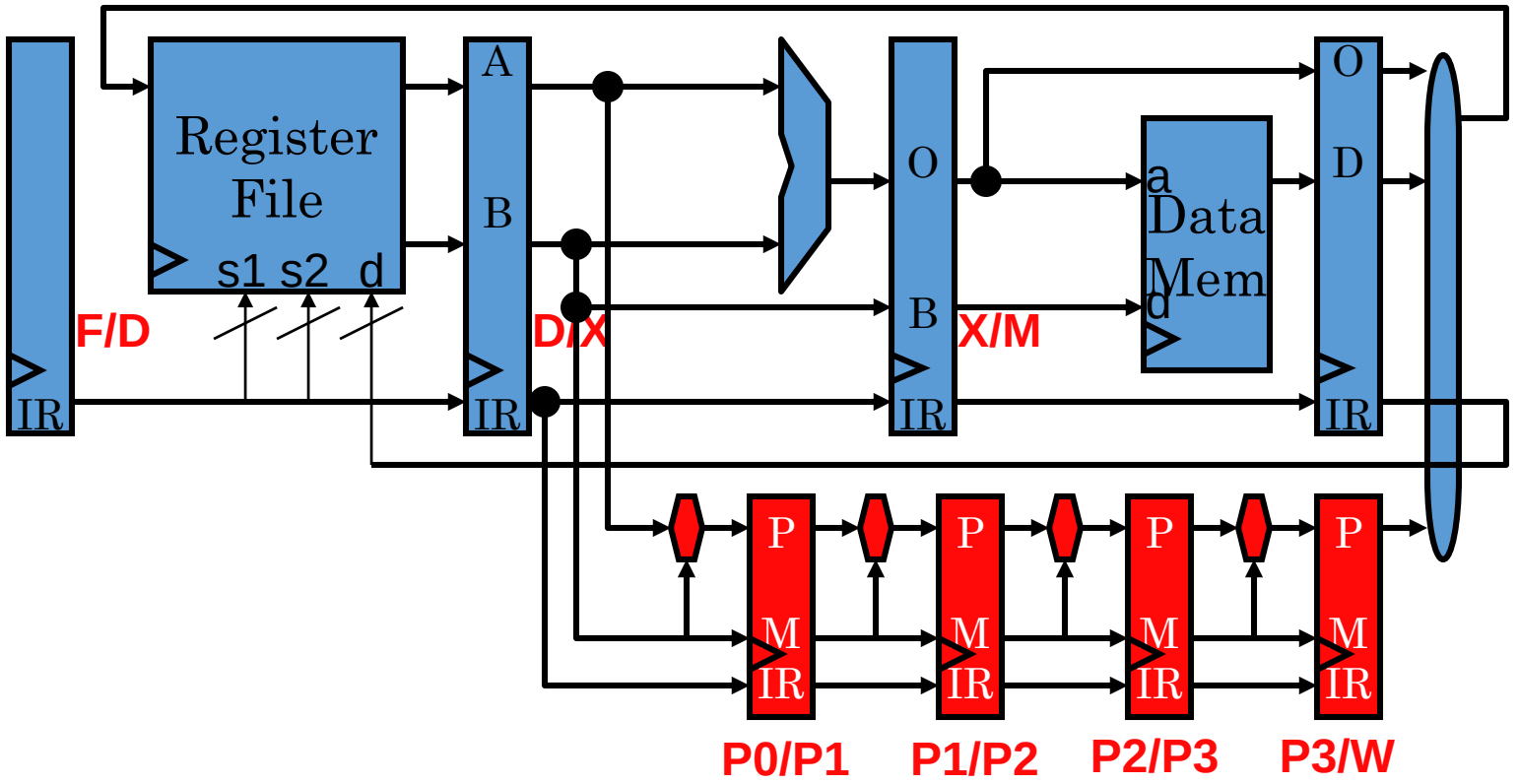
Stall = (OldStallLogic) | |

(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) | |

(F/D.IR.RS1 == P2/P3.IR.RD) | | (F/D.IR.RS2 == P2/P3.IR.RD)

Actually, ugh…



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

  Stall = (OldStallLogic) | |

  (**F/D.IR.RD != null && P0/P1.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		
`addi $6,$4,1`		F	**d***	**d***	**d***	D	X	M	W

- This is the situation 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		
`addi $6,$1,1`		F	D	X	M	W			
`add $5,$6,$10`			F	D	X	M	**W**		

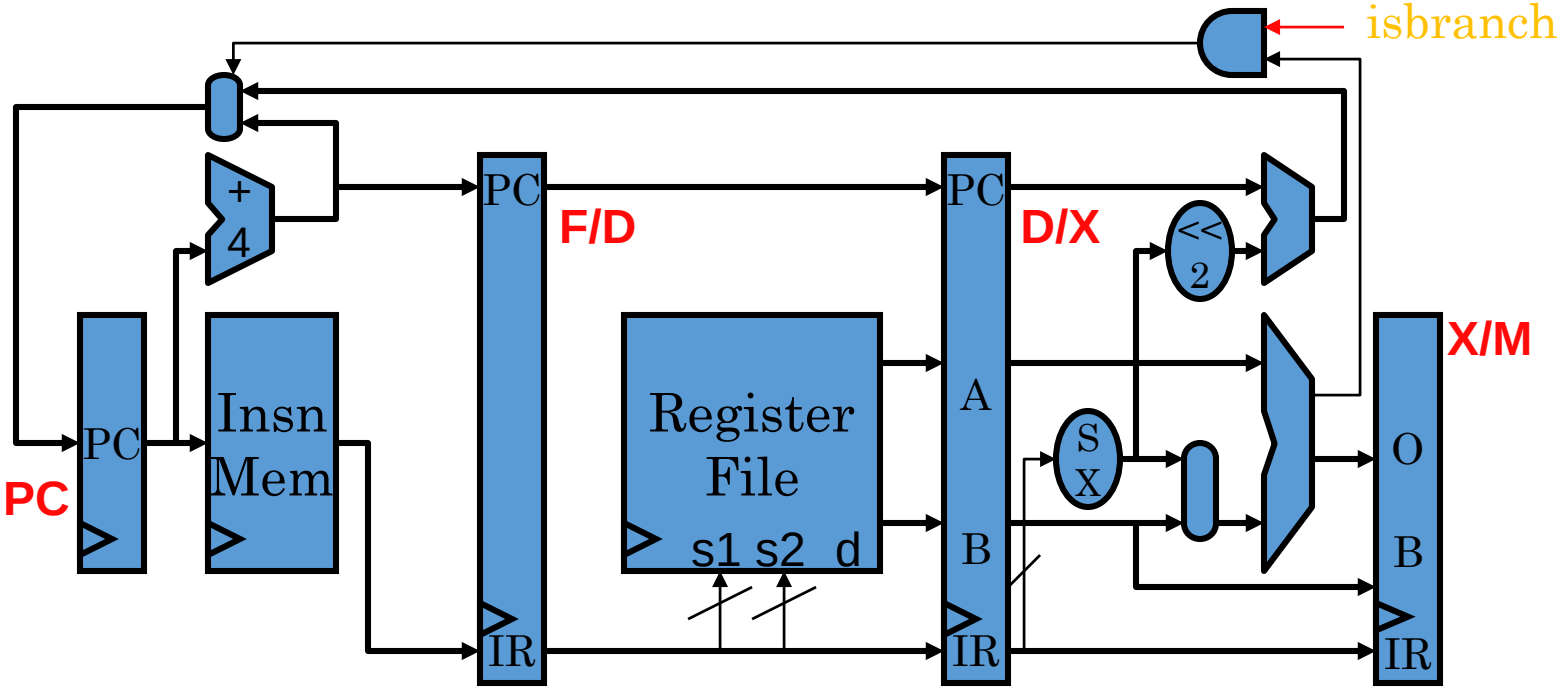
More Multiplier Nasties

- This is another situation we'll need to avoid (WAW hazard)
  - 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	W

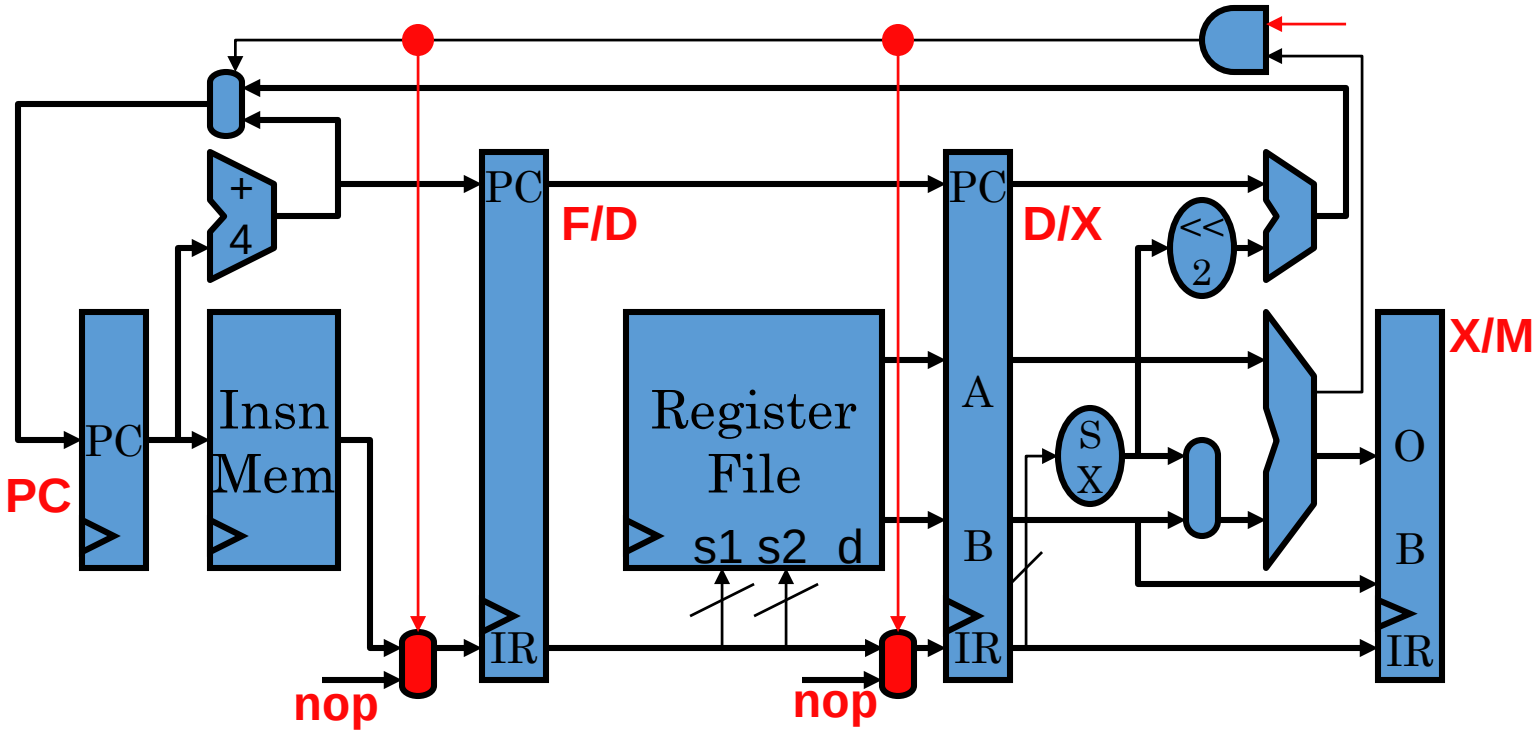
- **Multi-cycle operations complicate pipeline logic**
  - They're not impossible, but they require more complexity

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



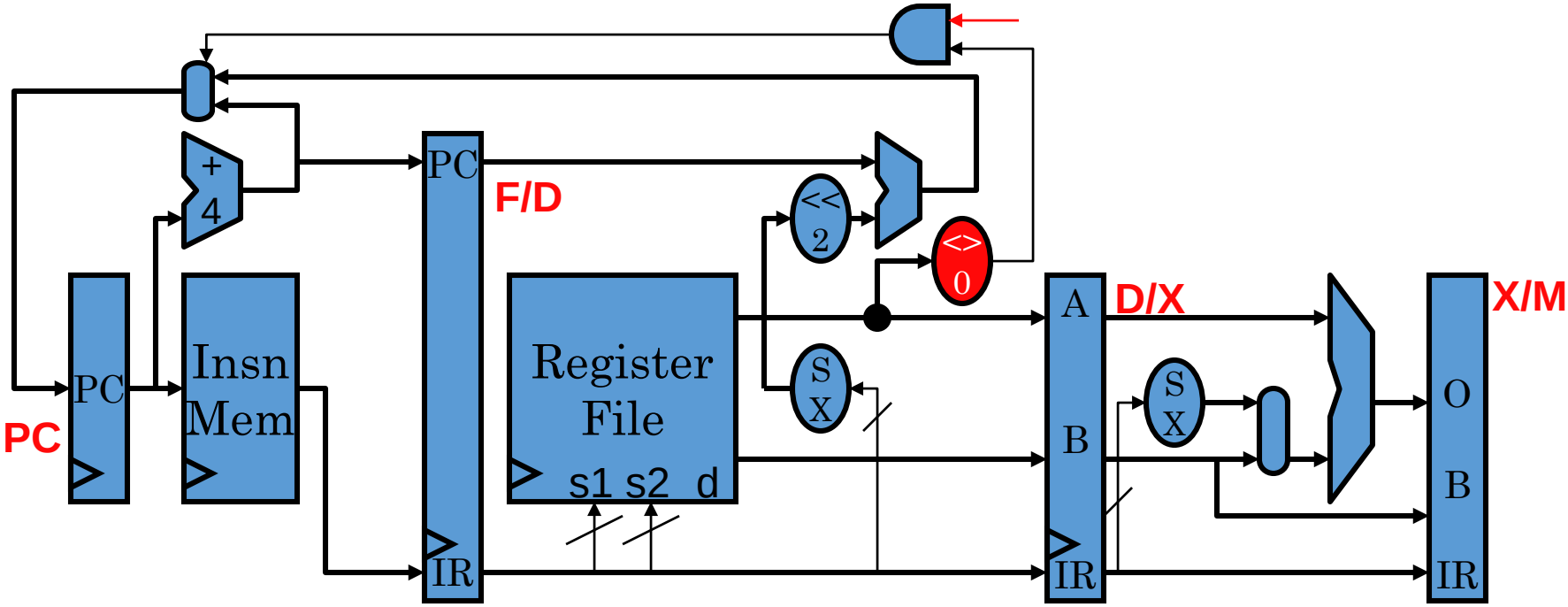
- **Branch recovery**: what to do when branch **is** taken
  - **Flush** insns currently in F/D and D/X (they're wrong)
    - Replace with **NOPs**
    - Haven't yet written to permanent state (RegFile, DMem)

Control Hazard Pipeline Diagram

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

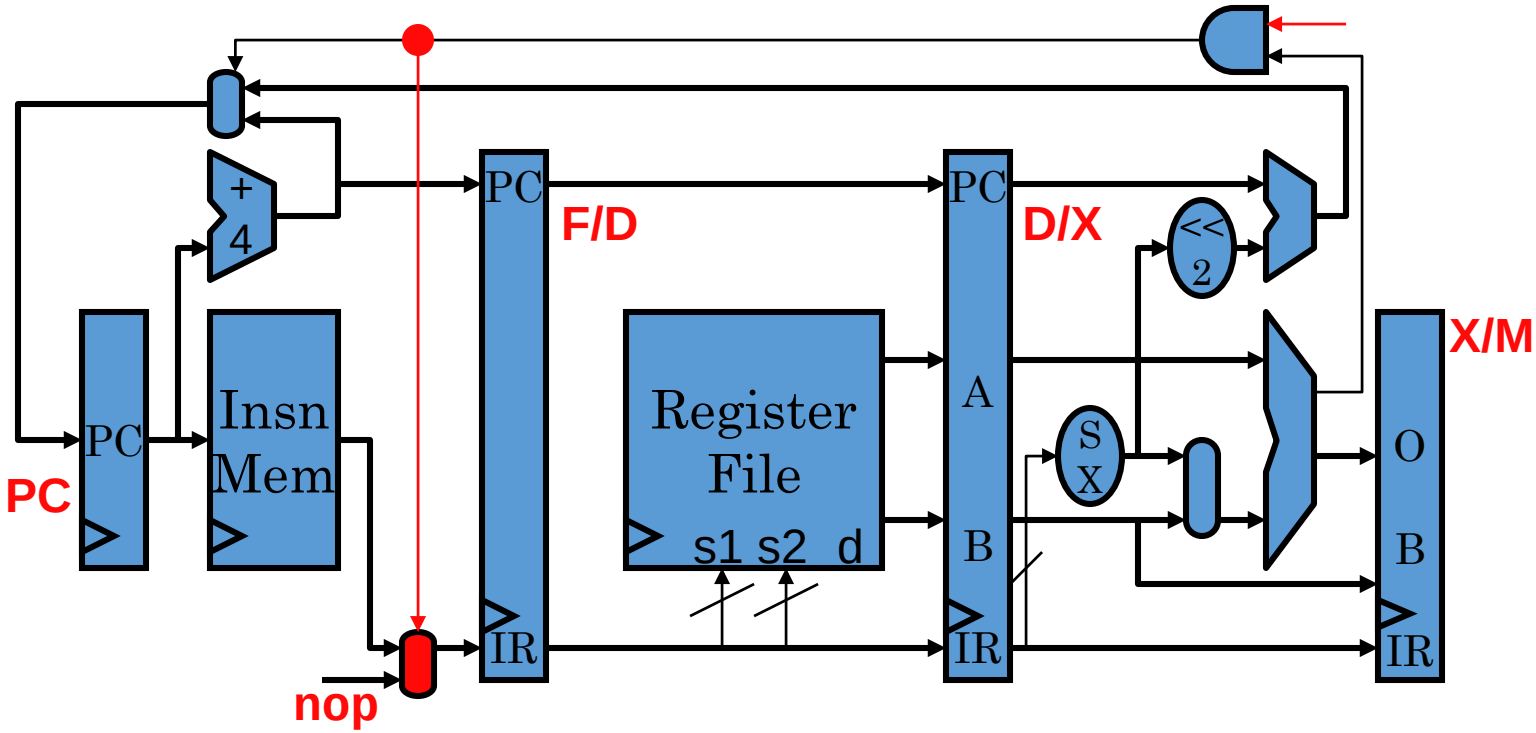
	1	2	3	4	5	6	7	8	9
`addi $3,$0,1`	F	D	X	M	W				
`bnez $3,targ`		F	D	X	M	W			
`sw $6,4($7)`			**c***	**c***	F	D	X	M	W

One Sol'n: Fast Branches



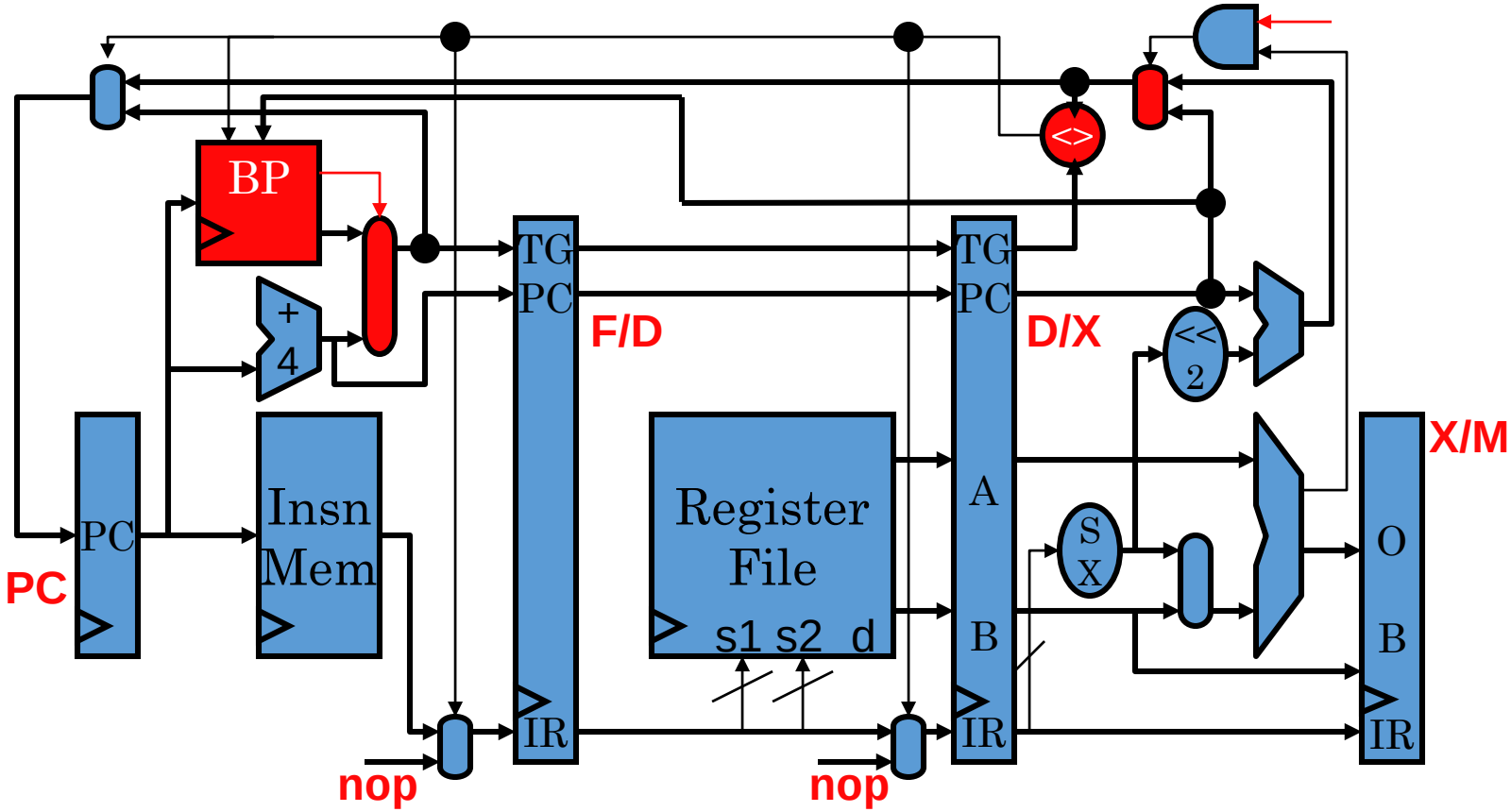
- **Fast branch**: resolves in Decode stage, not Execute
  - Test must be comparison to zero or equality, no time for ALU
  - + New taken branch penalty is only 1
  - − Must be able to bypass into decode now, too

Another Sol'n: Delayed Branches



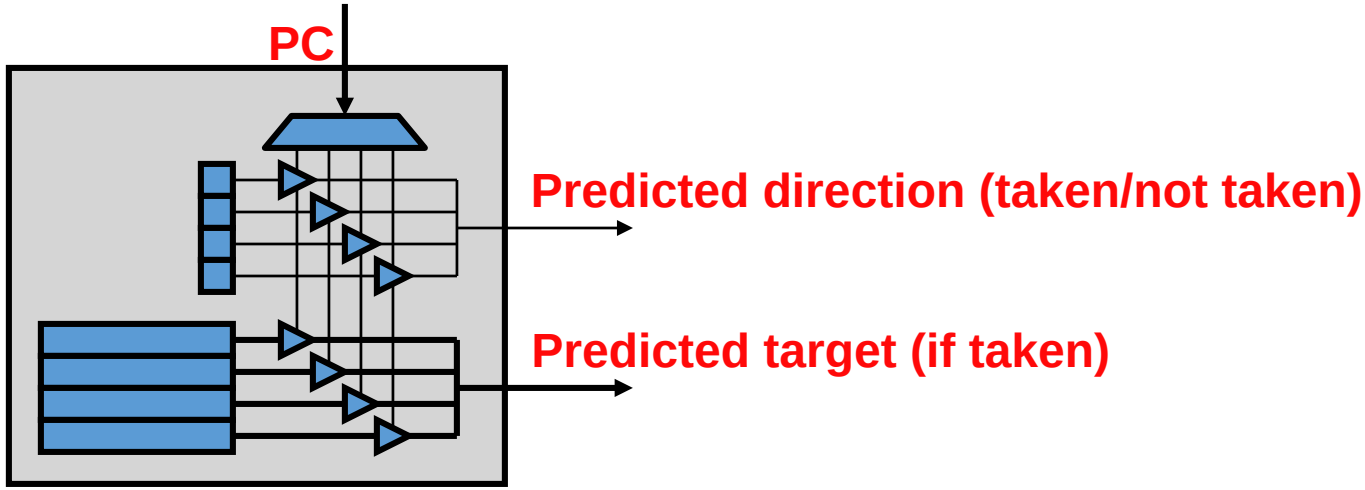
- **Delayed branch**: don't flush insn immediately following
  - As if branch takes effect one insn later
  - ISA modification → compiler accounts for this behavior
  - Insert insns independent of branch into **branch delay slot(s)**

Dynamic Branch Prediction



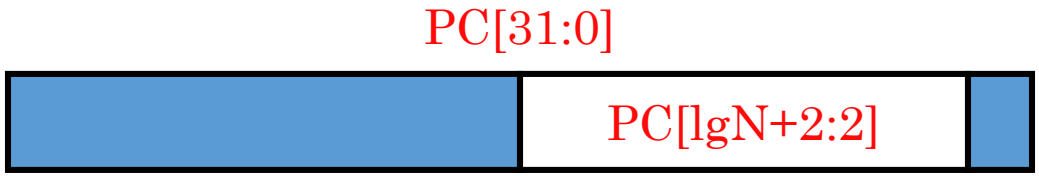
- **Dynamic branch prediction**: guess outcome
  - Start fetching from guessed address
  - Flush on **mis-prediction**

Inside A Branch Predictor



- Two parts
  - **Direction predictor**: maps PC to taken/not-taken
  - **Target buffer**: maps PC to taken target
- 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
  - 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?
- 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?

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
- See ECE 552 for details!