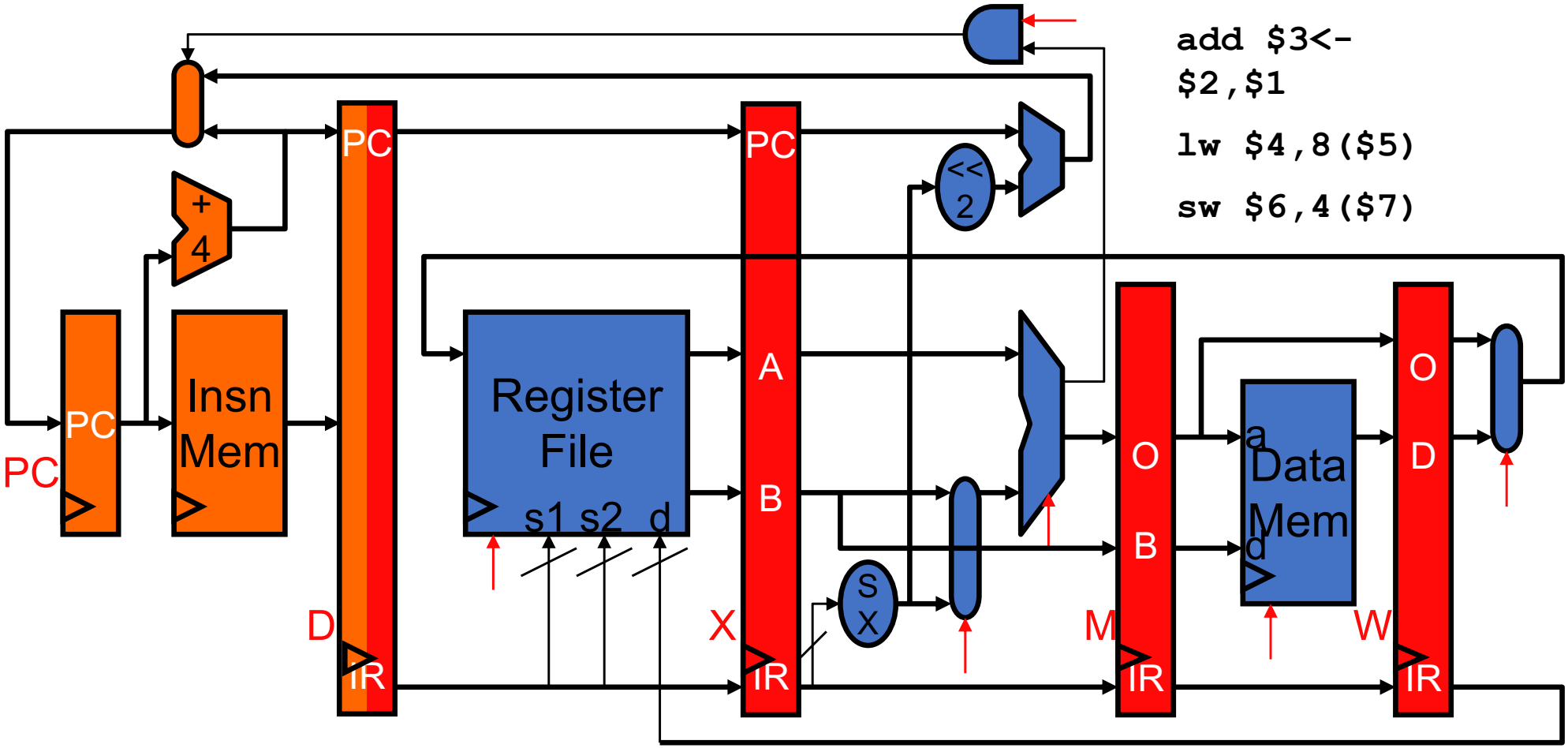
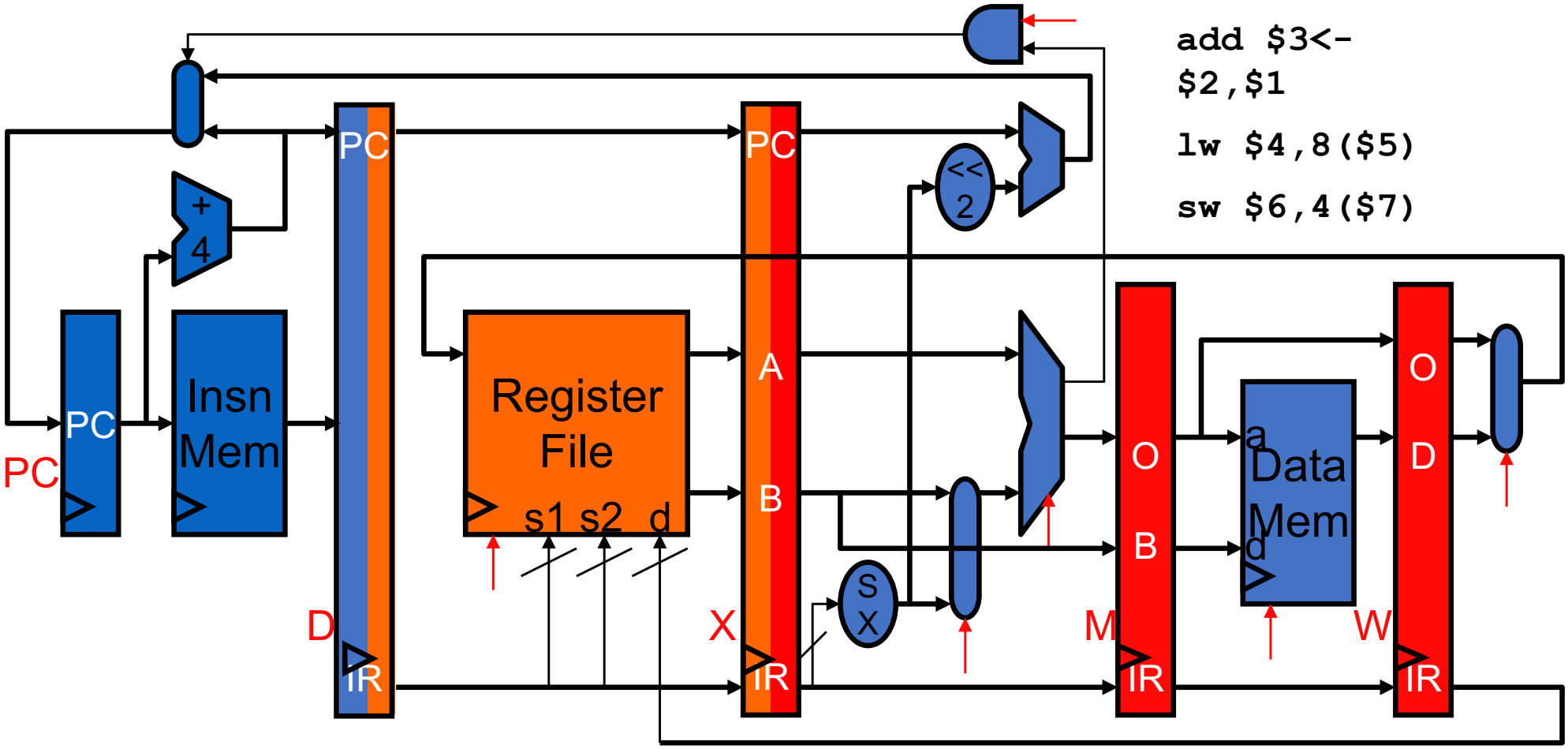
Pipeline Example: Cycle 1



add $3<-$2,$1

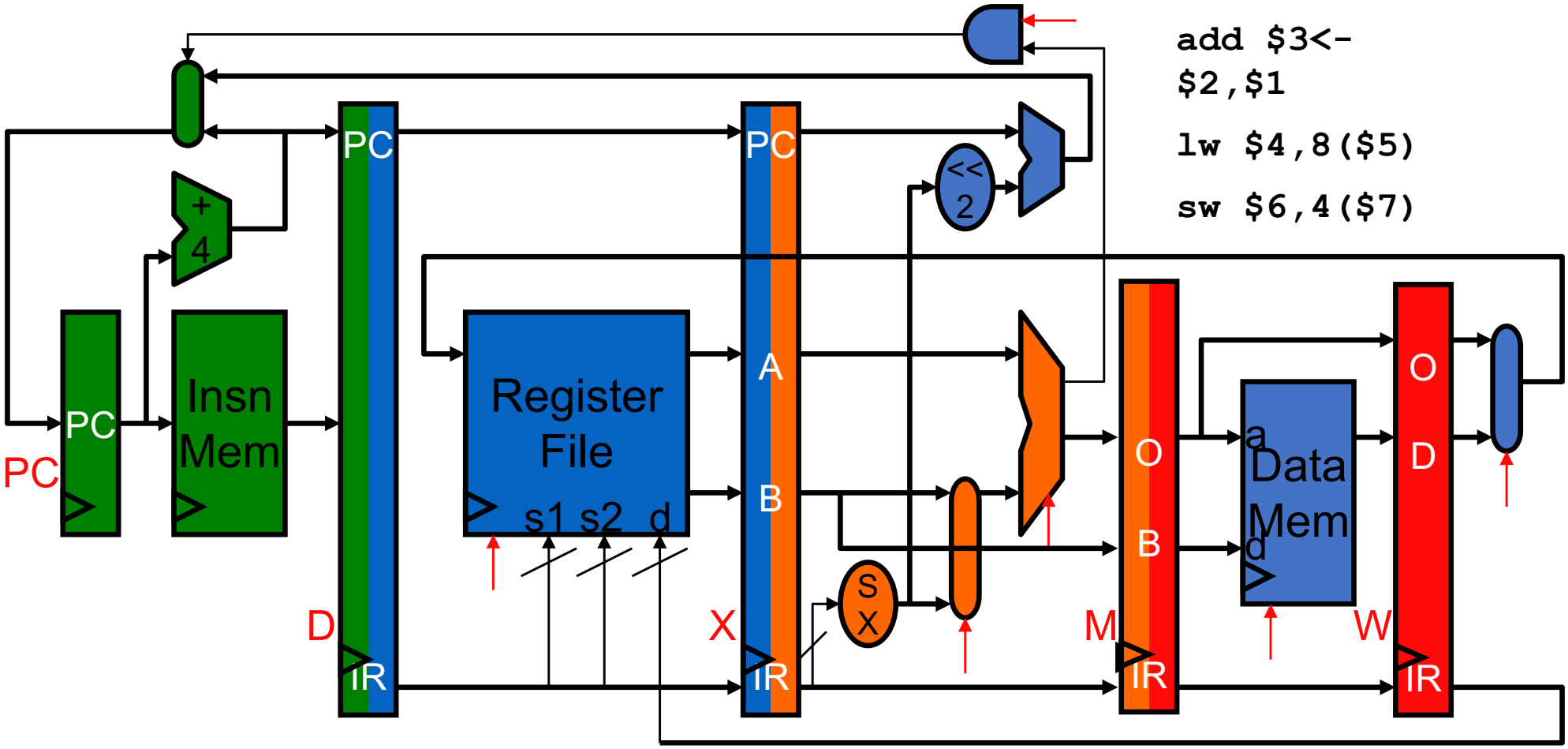
- 3 instructions

Pipeline Example: Cycle 2



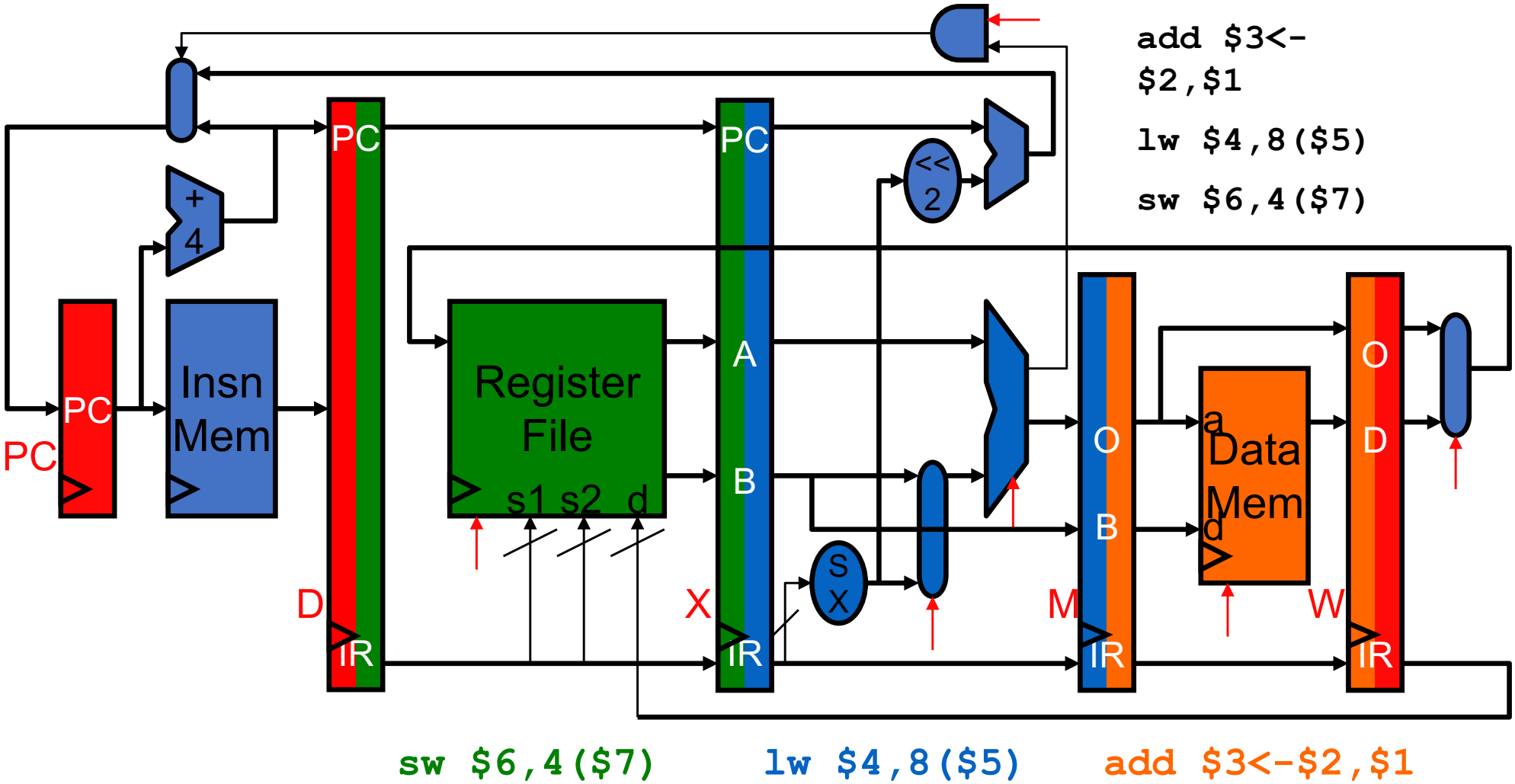
lw $4,8($5) add $3<-$2,$1

Pipeline Example: Cycle 3



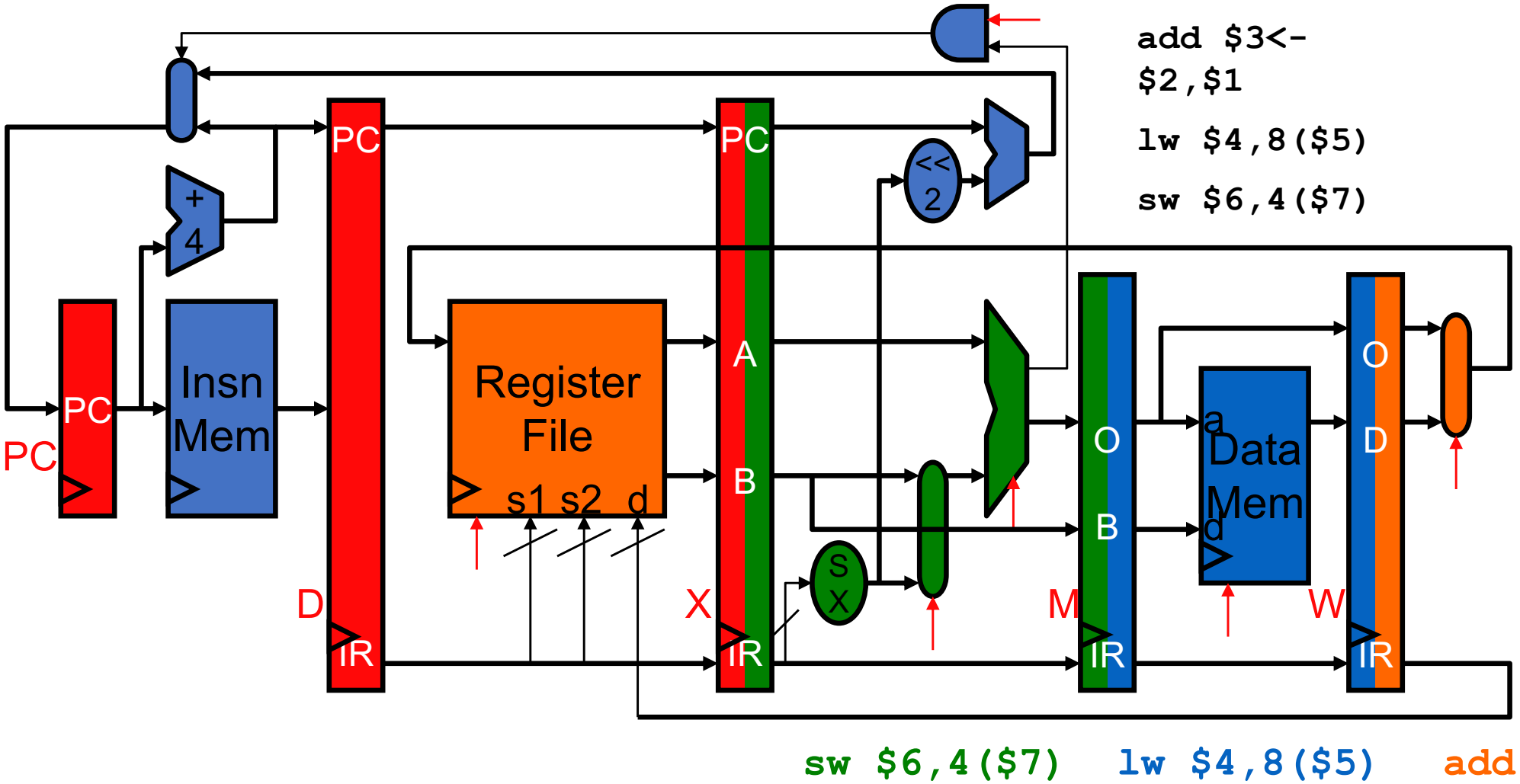
sw $6,4($7) lw $4,8($5) add $3<-$2,$1

Pipeline Example: Cycle 4

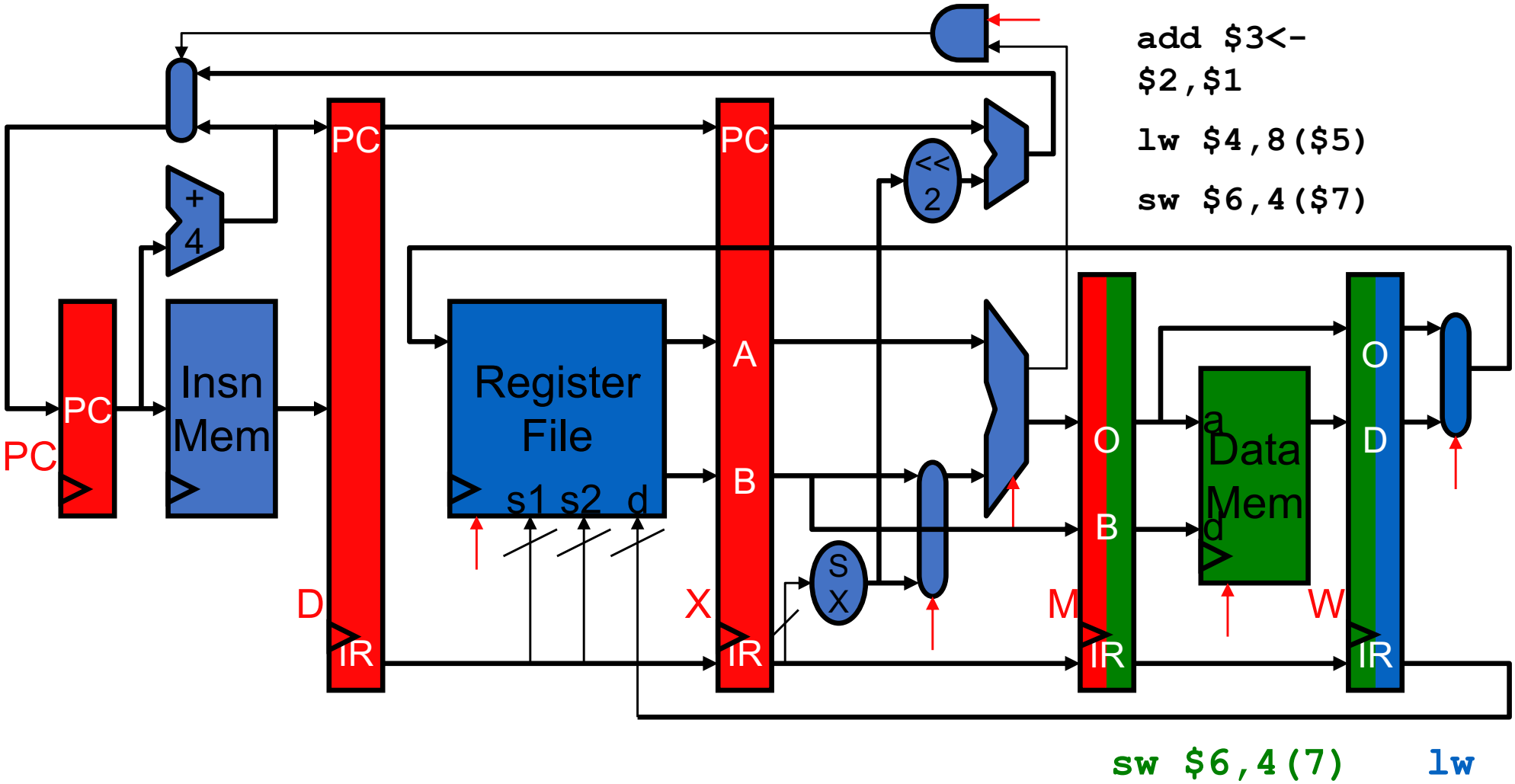


- 3 instructions

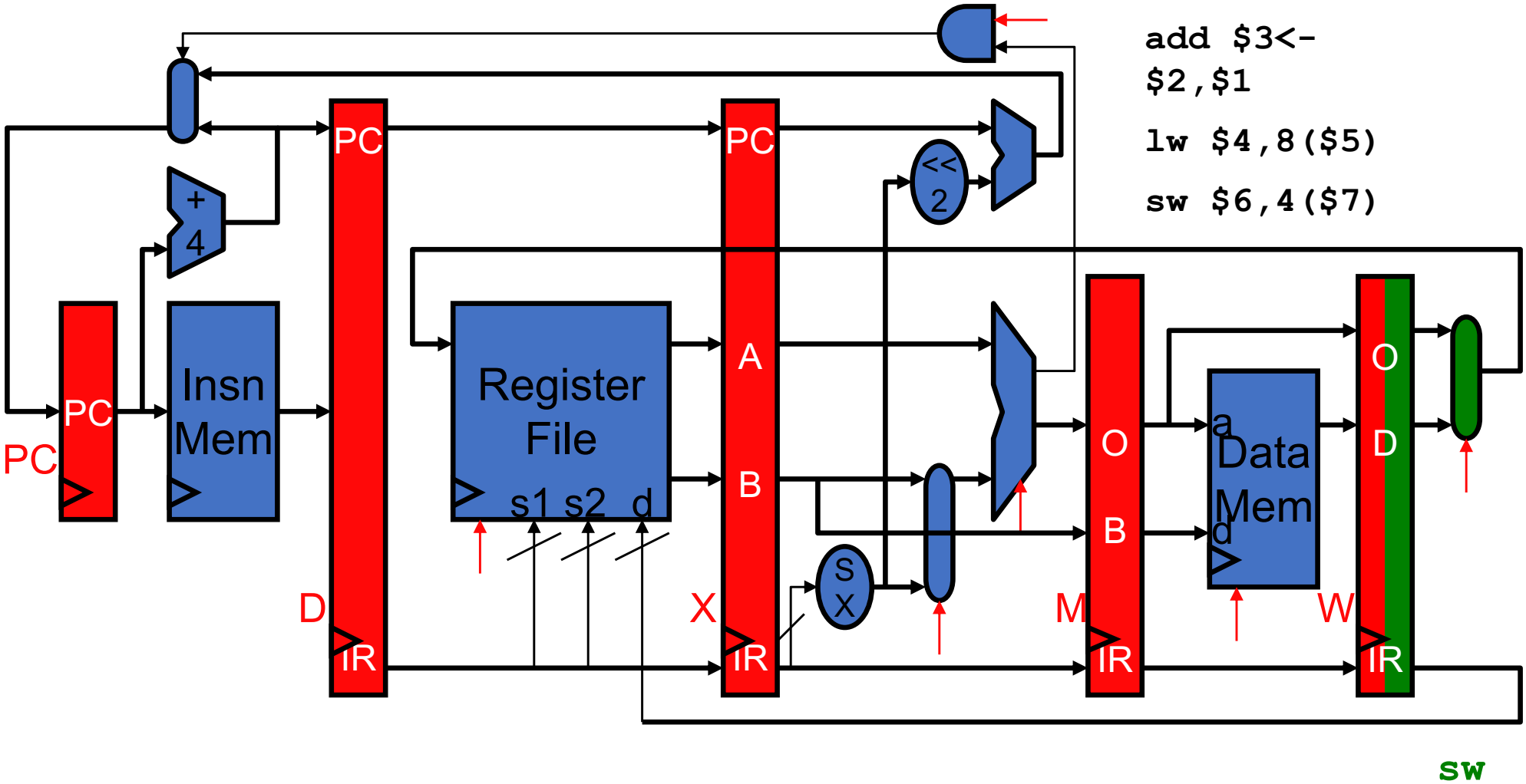
Pipeline Example: Cycle 5



Pipeline Example: Cycle 6



Pipeline Example: Cycle 7



Pipeline Diagram

- **Pipeline diagram**: shorthand for what we just saw
  - Across: cycles
  - Down: insns
  - Convention: e.g., **X** means `lw $4,8($5)` finishes e**X**ecute stage and writes into 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 $4,8($5)`		F	D	**X**	M	W			
`sw $6,4($7)`			F	D	X	M	W		

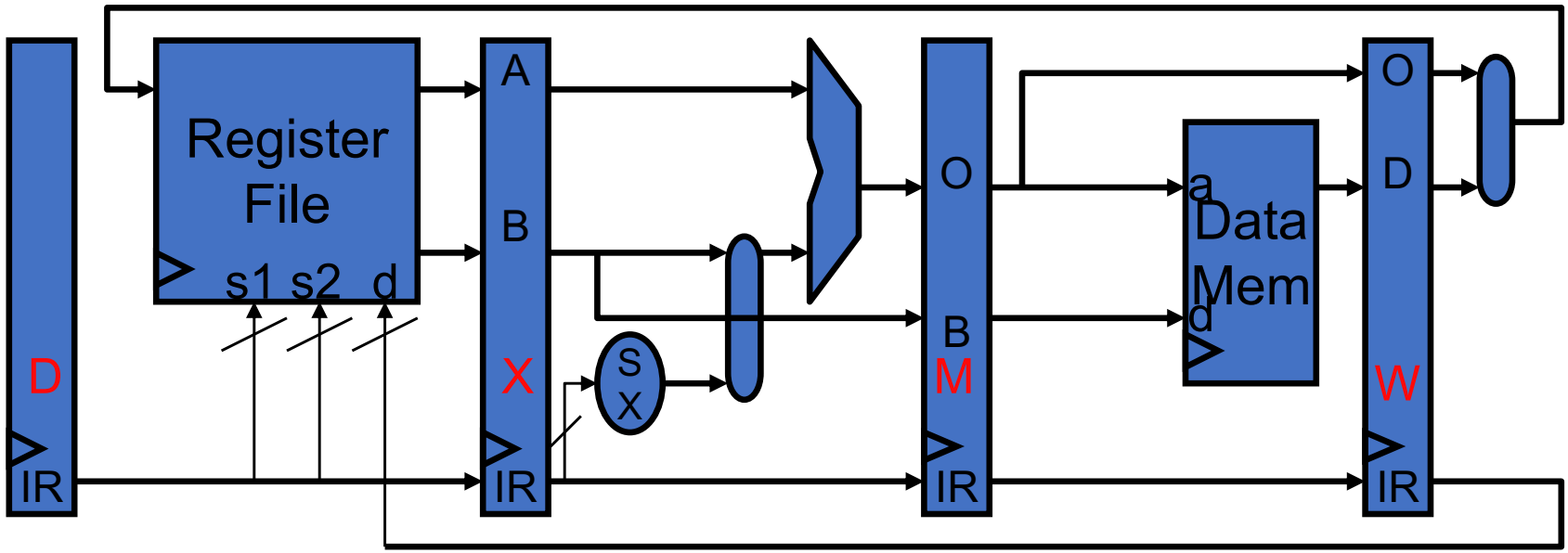
Data Dependences, Pipeline Hazards, and Bypassing

Dependences and Hazards

- **Dependence**: relationship between two insns
  - **Data dep.**: two insns use same storage location
  - **Control dep.**: one insn affects whether another executes at all
  - **Enforced** by making older insn go before younger one
    - Happens naturally in single-cycle designs
    - But not in a pipeline!

- **Hazard**: dependence & possibility of wrong insn order
  - **Stall**: for order by keeping younger insn waiting in same stage
  - Hazards are a bad thing: stalls reduce performance

Data Hazards




```
sw $6,4($7)    lw $4,8($5)   add $3<-$2,$1
```

- 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
 lw  $7,($9)
```

Dependent Operations

Data hazard

	1	2	3	4	5	6	7	8	9
`add $3<-$2,$1`	F	D	X	M	W				
`add $6<-$5,$3`		F	D	**X**	M	W			
`Lw $7,($9)`			F	D	X	M	W		

	1	2	3	4	5	6	7	8	9
`add $3<-$2,$1`	F	D	X	M	W				
`add $6<-$5,$3`		F	D	D	D	**X**	M	W	
`Lw $7,($9)`									

Stall for 2 cycles

Dependent Operations

	1	2	3	4	5	6	7	8	9
`add $3<-$2,$1`	F	D	X	M	W				
`add $6<-$5,$3`		F	D	**X**	M	W			
`Lw  $7,($9)`			F	D	X	M	W		

	1	2	3	4	5	6	7	8	9
`add $3<-$2,$1`	F	D	X	M	W				
`add $6<-$5,$3`		F	D	D	D	**X**	M	W	
`Lw  $7,($9)`			F	D	X	M	W		

Structural Hazard

Dependent Operations

	1	2	3	4	5	6	7	8	9
`add $3<-$2,$1`	F	D	X	M	W				
`add $6<-$5,$3`		F	D	X	M	W			

Hazards → solution 1: STALL

	1	2	3	4	5	6	7	8	9
`add $3<-$2,$1`	F	D	X	M	W				
`add $6<-$5,$3`		F	D	D	D	X	M	W	
`Lw $7,($9)`			F	F	F	D	X	M	W

Stall for 2 cycles as well

Structural Hazards

- **Structural hazards**
  - Two insns trying to use same circuit at same time
    - E.g., structural hazard on register file write port
- **Tolerate structure hazards**
  - Stall: Add stall logic to stall pipeline when hazards occur
- **To avoid structural hazards**
  - Avoided if:
    - Each insn uses every structure exactly once
    - For at most one cycle
    - All instructions travel through all stages
  - Add more resources:
    - Example: two memory accesses per cycle (Fetch & Memory)
    - Split instruction & data memories allows simultaneous access

Note: Why Does Every Insn Take 5 Cycles?

	1	2	3	4	5	6	7	8	9
`Lw $7,($9)`		F	D	X	M	W			
`add $6<-$5,$3`			F	D	X	M	W		
`add $6<-$5,$3`			F	D	X	W			

- 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**: imagine **`add`** after **`lw`** (only 1 reg. write port)

Solve data and structural hazards -- Stalls

	1	2	3	4	5	6	7	8	9
`add $3<-$2,$1`	F	D	X	M	W				
`add $6<-$5,$3`		F	D	**X**	M	W			
`Lw  $7,($9)`			F	D	X	M	W		

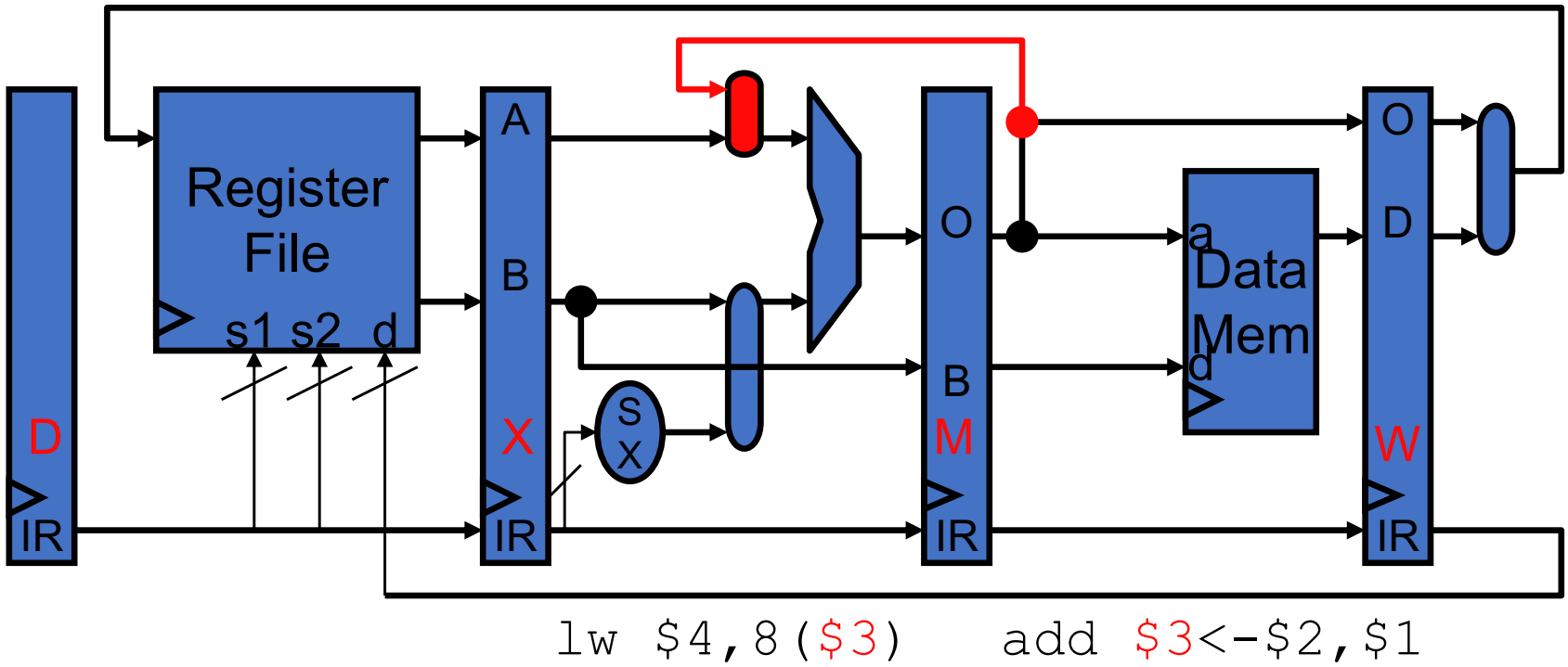
Stalls reduce performance

`add $6<-$5,$3`		F	D	D	D	**X**	M	W	
`Lw  $7,($9)`			F	F	F	D	X	M	W

Stall for 2 cycles as well

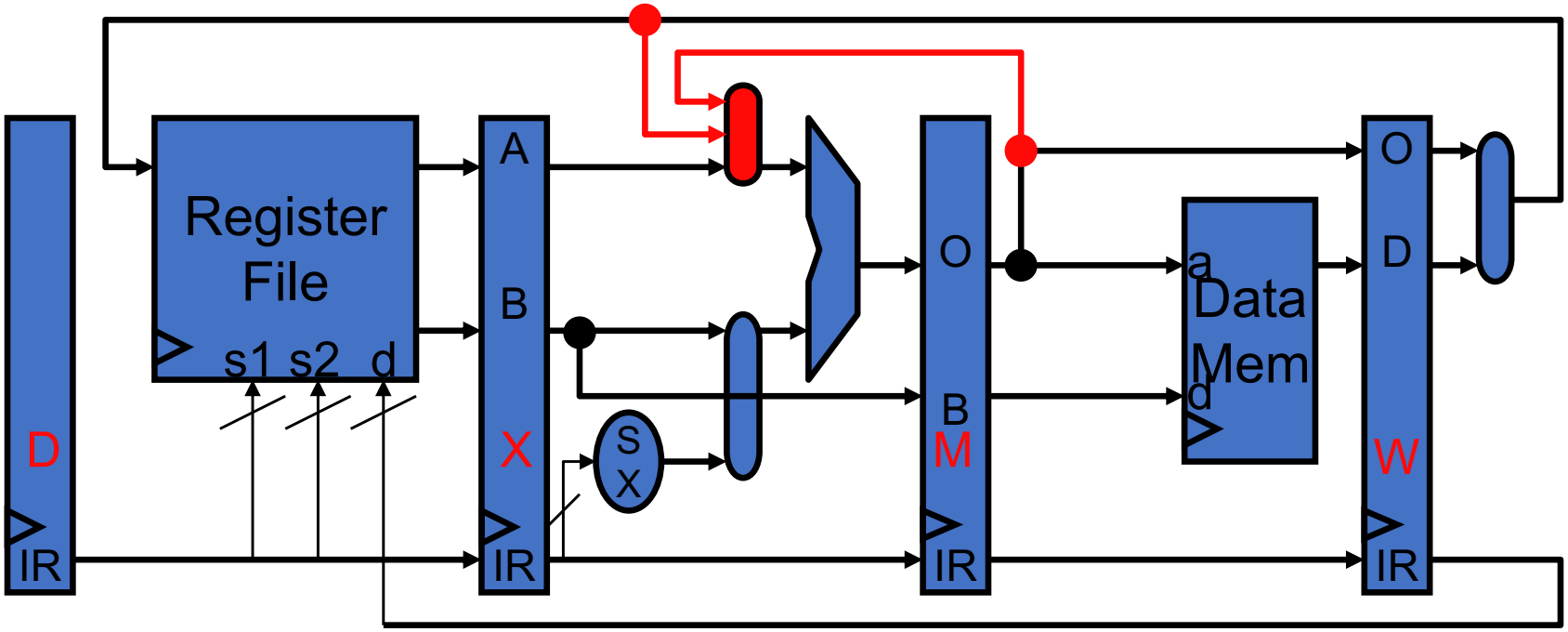
Can we do better than STALLs?

Solving data hazards w/o stalls: Bypassing



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

WX Bypassing




```
add $4<-$3,$2                    add $3<-$2,$1
                Some other insn
```

- What about this combination?
  - Add another bypass path and MUX (multiplexor) input
  - This one is a **WX** bypass

Pipeline Diagrams with Bypassing

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

	1	2	3	4	5	6	7	8	9	10
`add r1<-r2,r3`	F	D	X	**M**	W					
`sub r2<-r1,r4`		F	D	**X**	M	W				

  - Example: WX bypass

	1	2	3	4	5	6	7	8	9	10
`add r1<-r2,r3`	F	D	X	M	**W**					
`ld r5,[r7+4]`		F	D	X	M	W				
`sub r2<-r1,r4`			F	D	**X**	M	W			

  - Example: WM bypass

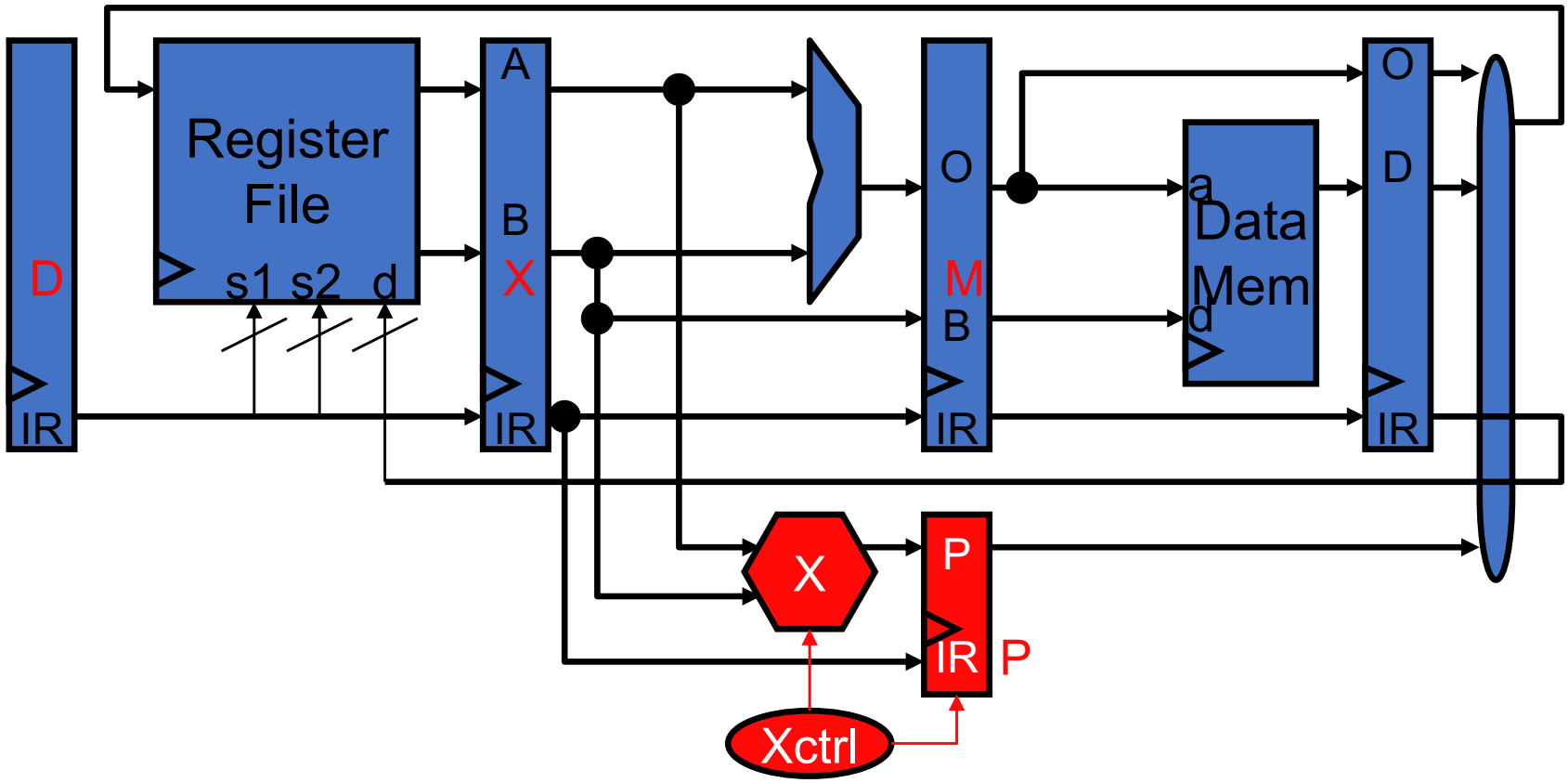
	1	2	3	4	5	6	7	8	9	10
`add r1<-r2,r3`	F	D	X	M	**W**					
`?`		F	D	X	**M**	W				

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

Answer: st r1, 8(r4)

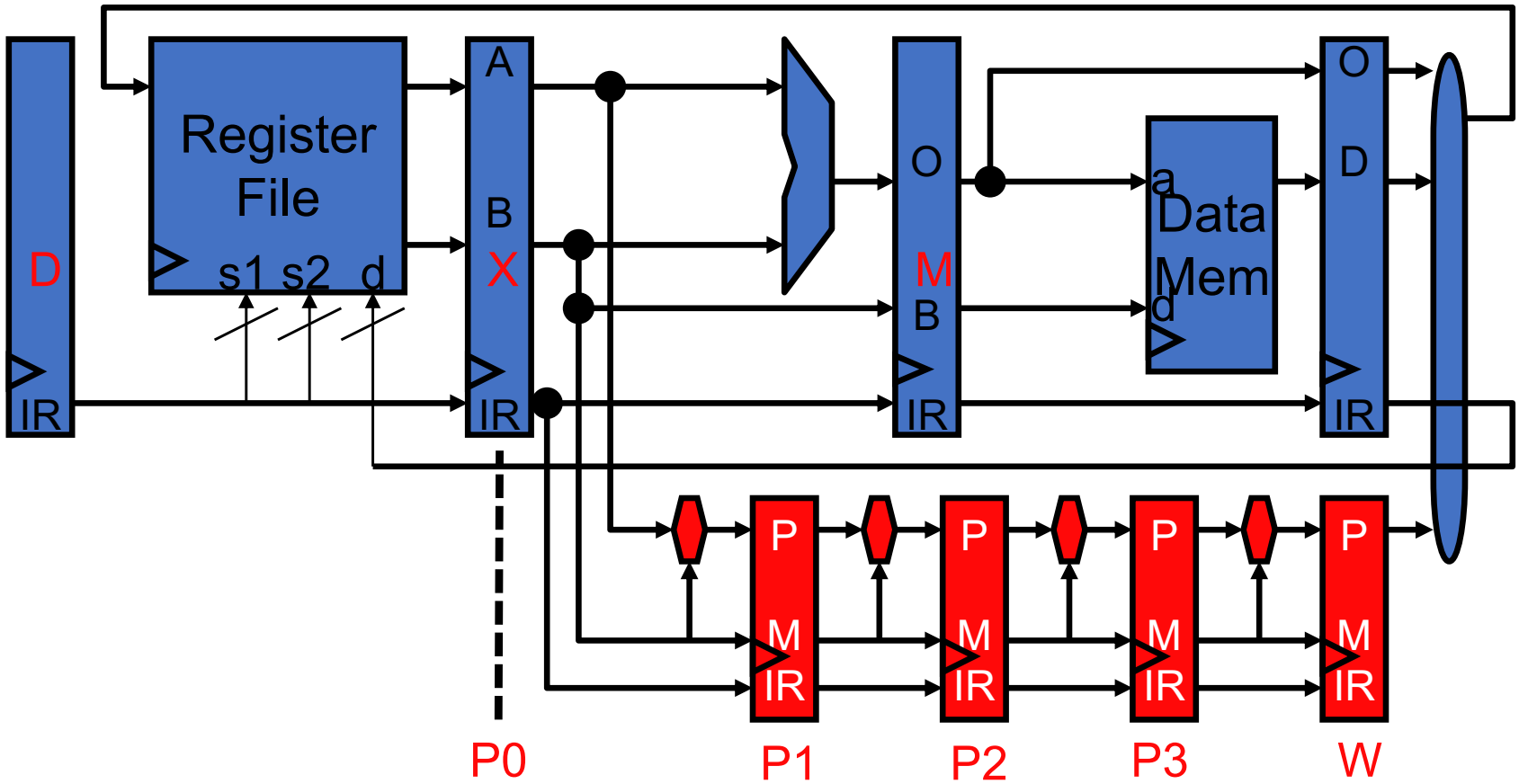
Multi-Cycle Operations

Pipelining and Multi-Cycle Operations



- What if we wanted to add an operation that takes multiple cycles to execute?
  - E.g., 4-cycle multiply
  - **P**: separate output latch connects to W stage
  - Controlled by pipeline control finite state machine (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
  - **But still takes 4 cycles to generate output value**

Pipeline Diagram with Multiplier

- Allow **independent** instructions

	1	2	3	4	5	6	7	8	9
`mul $4<-$3,$5`	F	D	P0	P1	P2	P3	W		
`addi $6<-$7,1`		F	D	X	M	W			

- Even allow **independent multiply** instructions

	1	2	3	4	5	6	7	8	9
`mul $4<-$3,$5`	F	D	P0	P1	P2	P3	W		
`mul $6<-$7,$8`		F	D	P0	P1	P2	P3	W	

- But must stall subsequent **dependent** instructions:

	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

Multiplier Write Port Structural Hazard

- What about…
  - Two instructions trying to write register file in same cycle?
  - Structural hazard!
- Must prevent:

	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**		

- Solution? stall the subsequent instruction

	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	**d***	X	M	**W**	

More Multiplier Nasties

- What about…
  - Mis-ordered writes to the same register
  - Software 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`									

- Common? Not for a 4-cycle multiply with 5-stage pipeline
  - More common with deeper pipelines
  - In any case, must be correct

Corrected Pipeline Diagram

- With the correct stall logic
  - Prevent mis-ordered writes to the same register
  - Why two cycles of delay?

	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	**d***	**d***	X	M	**W**	
…									
…									
`add $10,$4,$6`									

- **Multi-cycle operations complicate pipeline logic**

Pipelined Functional Units

- Almost all multi-cycle functional units are pipelined
  - Each operation takes *N* cycles
  - But can start initiate a new (independent) operation every cycle
  - Requires internal latching and some hardware replication
  - \+ A cheaper way to improve throughput than multiple non-pipelined units

	1	2	3	4	5	6	7	8	9	10	11
`mulf f0,f1,f2`	F	D	P0	P1	P2	P3	W				
`mulf f3,f4,f5`		F	D	P0	P1	P2	P3	W			

  - One exception: int/FP divide: difficult to pipeline and not worth it

	1	2	3	4	5	6	7	8	9	10	11
`divf f0,f1,f2`	F	D	P	P	P	P	W				
`divf f3,f4,f5`		F	D	**s***	**s***	**s***	P	P	P	P	W

  - **s*** = stall for structural hazard