ECE 550D
Fundamentals of Computer Systems and Engineering

Fall 2023

Pipelining

Xin Li & Dawei Liu
Duke Kunshan University

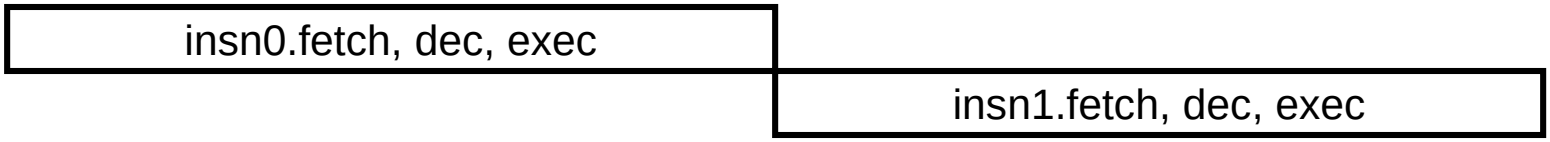
Slides are derived from work by
Andrew Hilton, Tyler Bletsch and Rabih Younes (Duke)

Last time

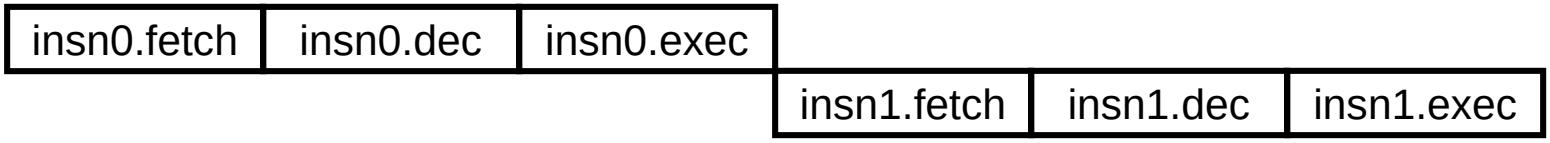
- What did we do last time?
- Datapaths:
  - Multi-cycle
    - Faster clock (yay!)
    - Worse CPI (boooo)
  - Performance:
    - IPC
    - Performance / Watt
    - CPU Performance Equation

Clock Period and CPI

- Single-cycle datapath
  - Low CPI: 1
  - Long clock period: to accommodate slowest insn

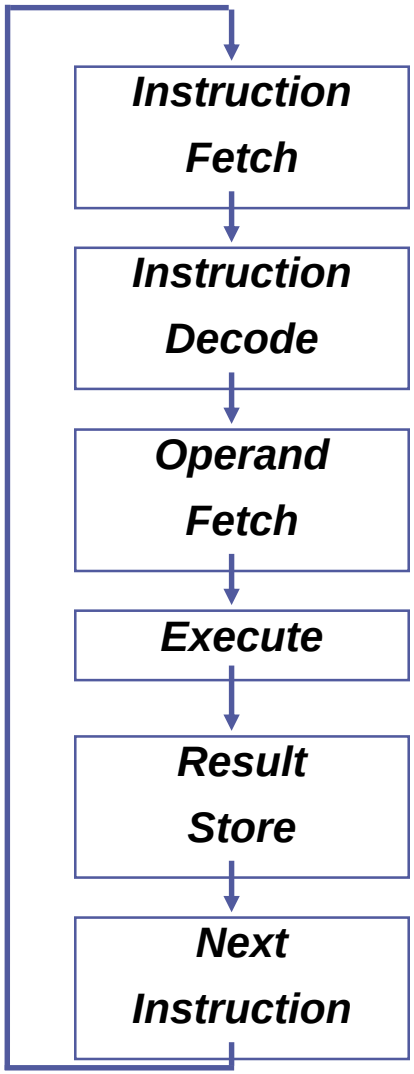


- Multi-cycle datapath
  - Short clock period
  - High CPI



- Can we have both low CPI and short clock period?
  - No good way to make a single insn go faster
  - Insn latency doesn't matter anyway … insn throughput matters
  - Key: exploit inter-insn parallelism

Remember The von Neumann Model?

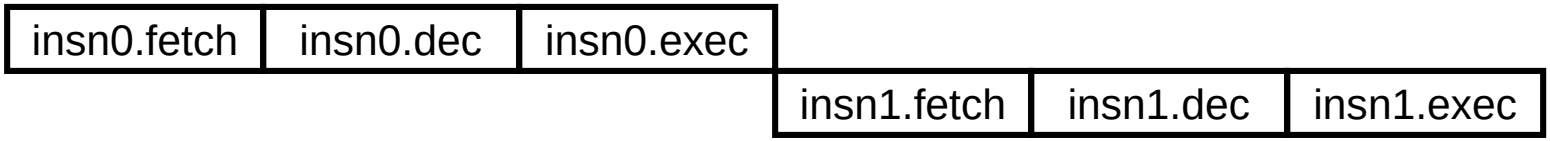


- Instruction Fetch:
  Read instruction bits from memory
- Decode:
  Figure out what those bits mean
- Operand Fetch:
  Read registers (+ mem to get sources)
- Execute:
  Do the actual operation (e.g., add the #s)
- Result Store:
  Write result to register or memory
- Next Instruction:
  Figure out mem addr of next insn, repeat

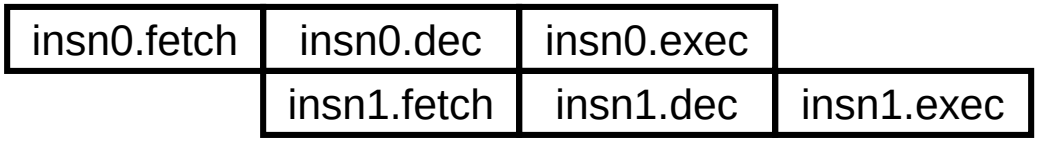
We'll call this the "**VN loop**"

Pipelining

- **Pipelining**: important performance technique
  - **Improves insn throughput rather than insn latency**
  - **Exploits parallelism at insn-stage level to do so**
  - Begin with multi-cycle design

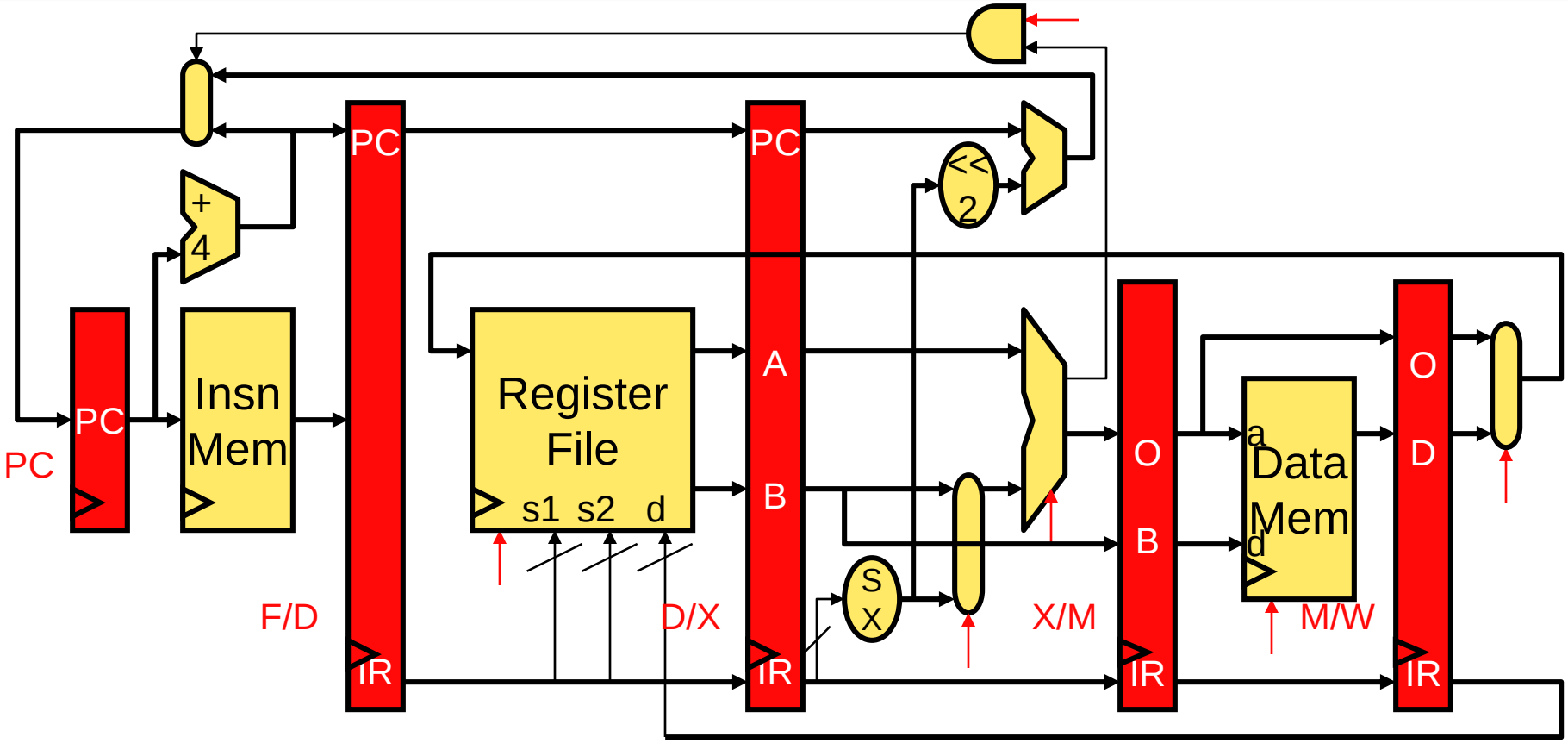


  - When insn advances from stage 1 to 2, next insn enters stage 1



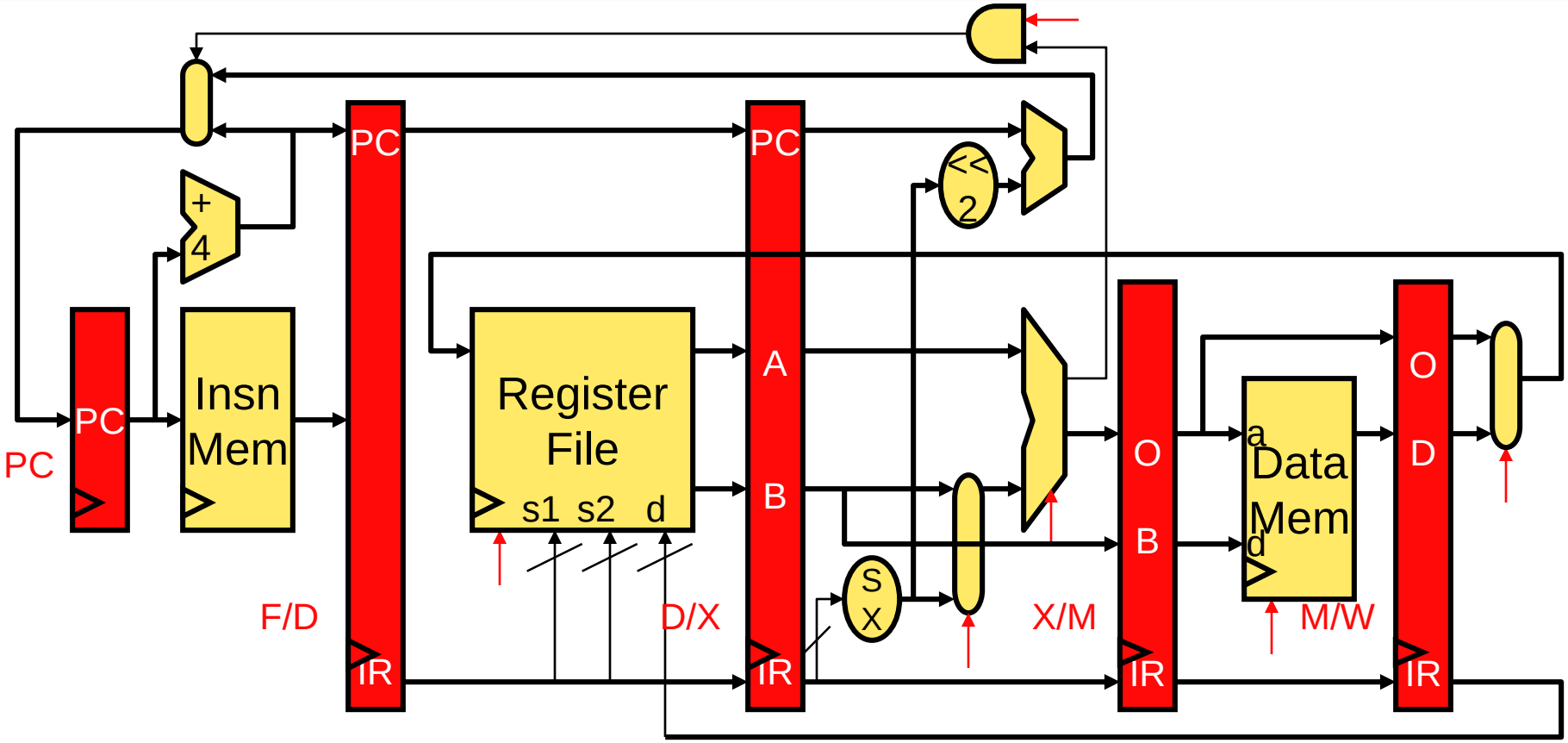
  - Individual insns take same number of stages
  + **But insns enter and leave at a much faster rate**
  - Physically breaks "atomic" VN loop ... but must maintain illusion
- Automotive assembly line analogy

5 Stage Pipelined Datapath



- Stages: **F**etch, **D**ecode, e**X**ecute, **M**emory, **W**riteback
- Latches (pipeline registers): **PC**, **F/D**, **D/X**, **X/M**, **M/W**

5 Stage Pipelined Datapath

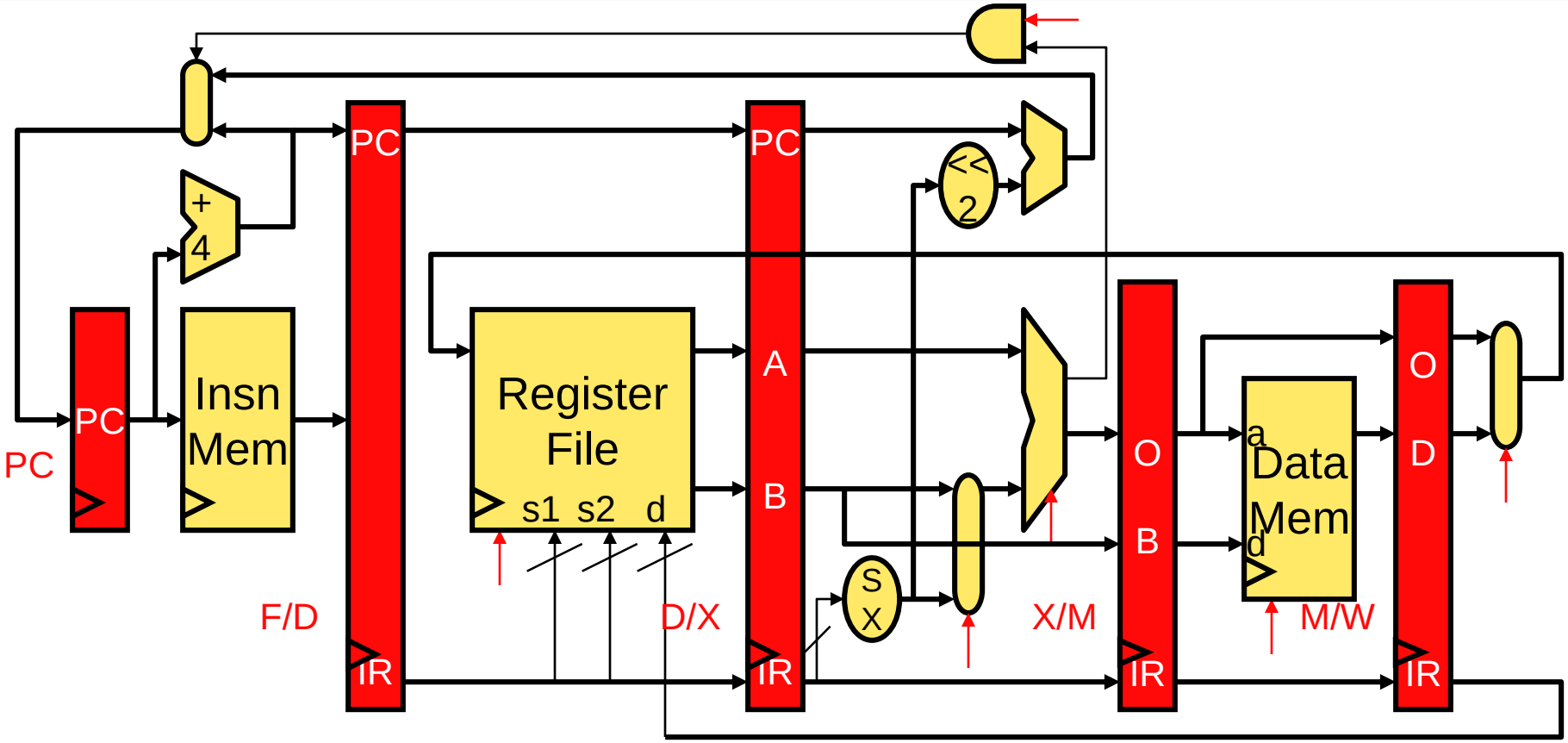


- Temporary values (PC,IR,A,B,O,D) re-latched every stage
  - Notice, PC not latched after ALU stage (why not?)

Pipeline Terminology

- **Scalar pipeline**: one insn per stage per cycle
  - Alternative: "superscalar" (take ECE 552)
- **In-order pipeline**: insns enter execute stage in VN order
  - Alternative: "out-of-order" (take ECE 552)
- **Pipeline depth**: number of pipeline stages
  - Nothing magical about five
  - Trend has been to deeper pipelines

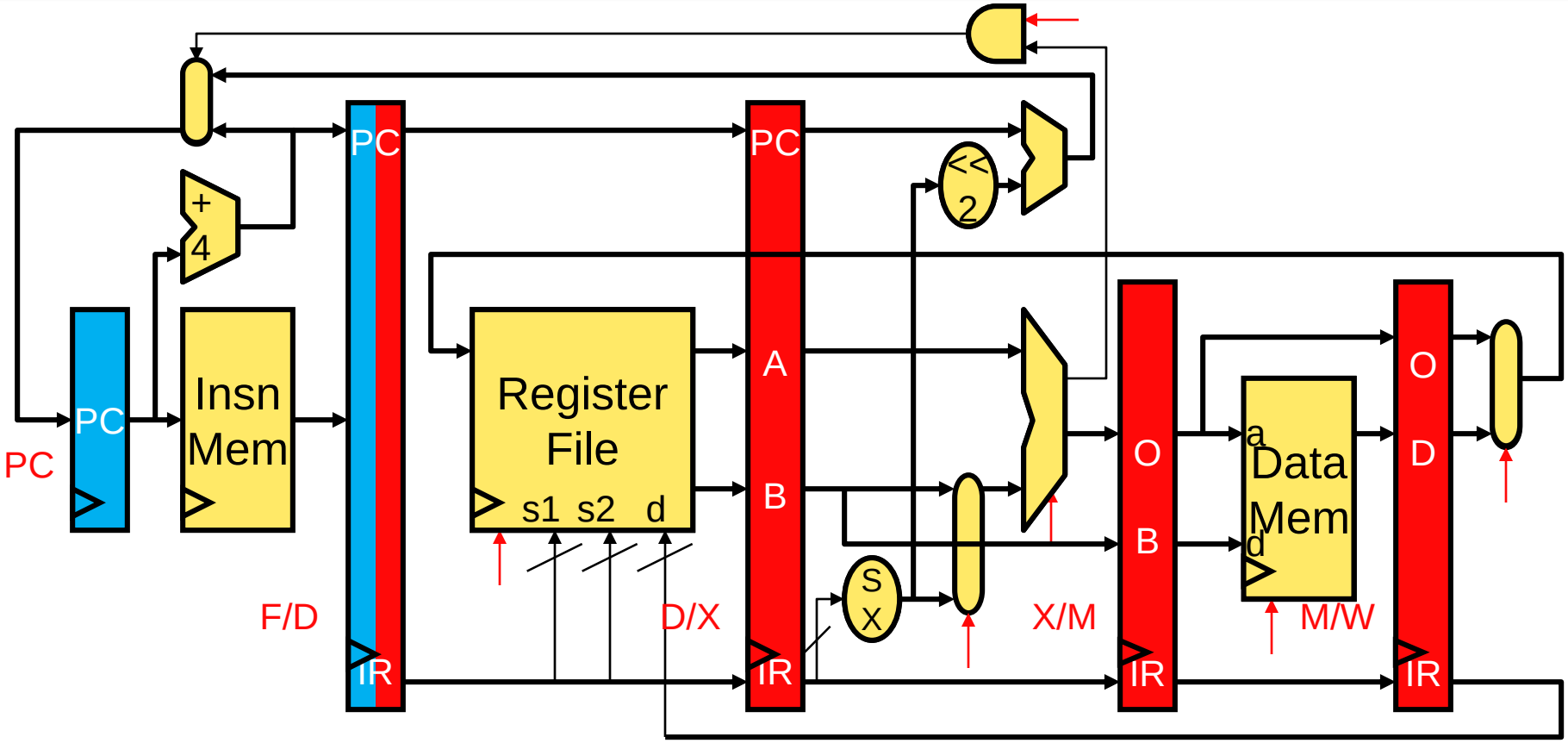
Pipeline Example: Cycle 1




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

- 3 instructions

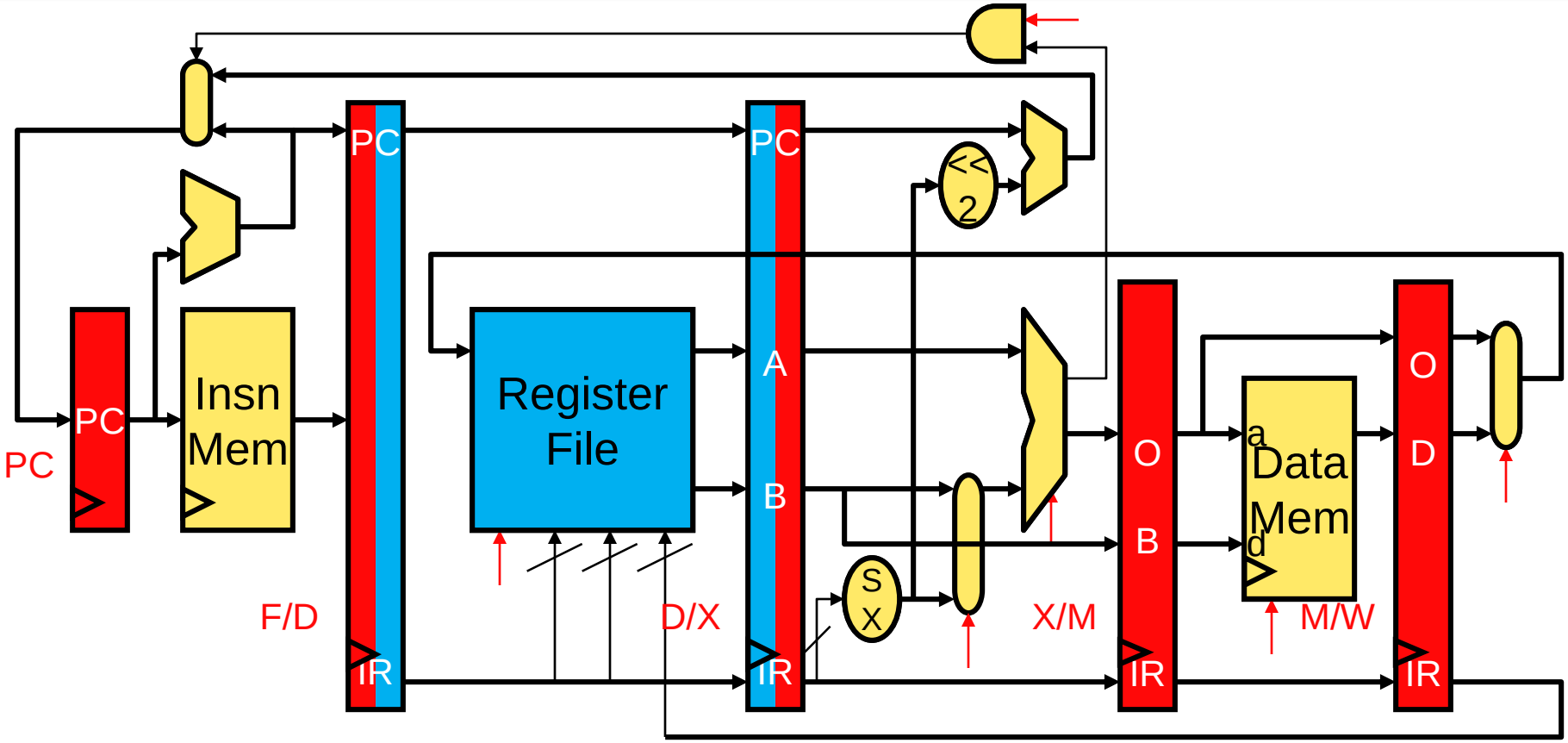
Pipeline Example: Cycle 2




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

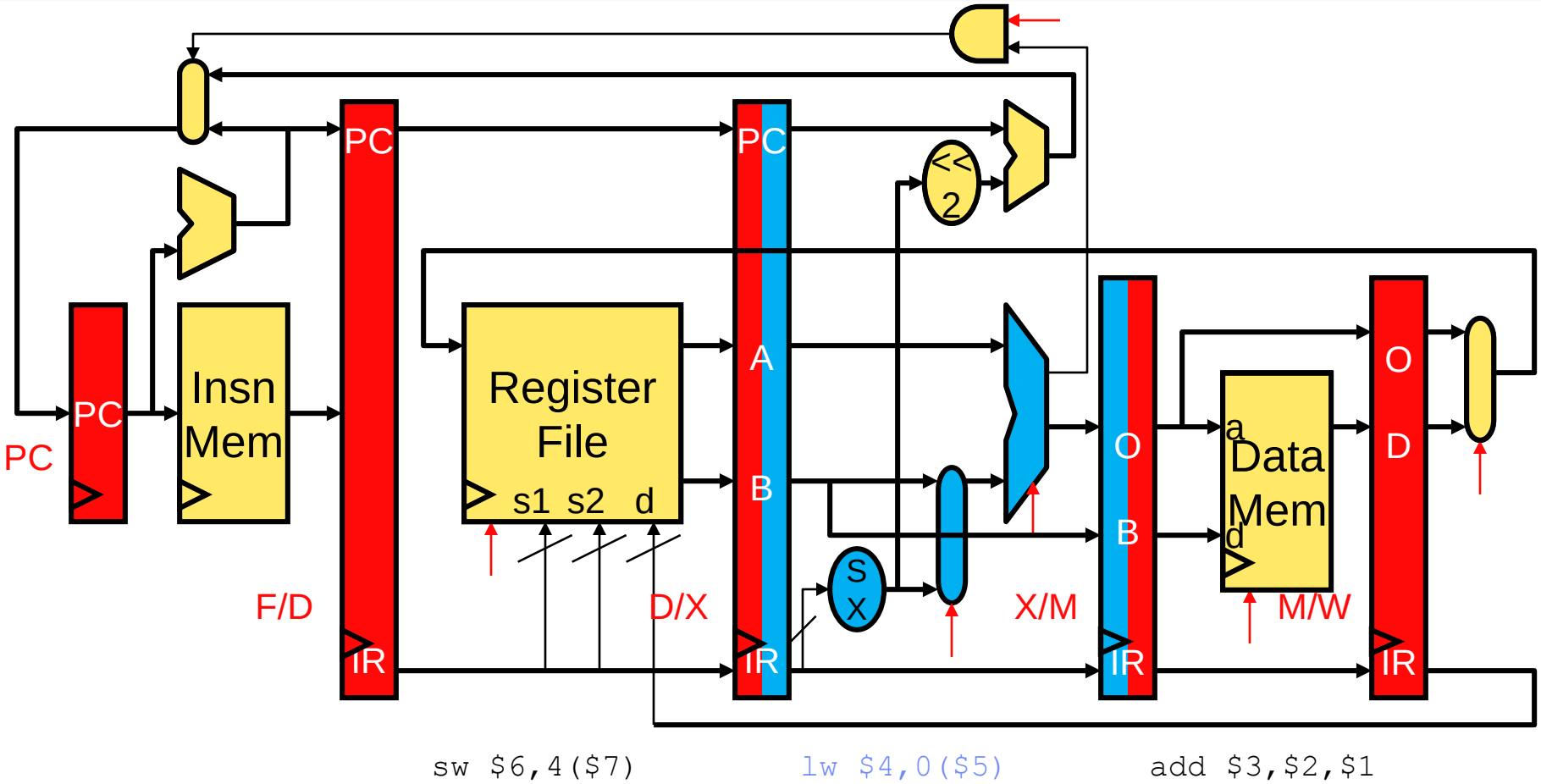
- 3 instructions

Pipeline Example: Cycle 3



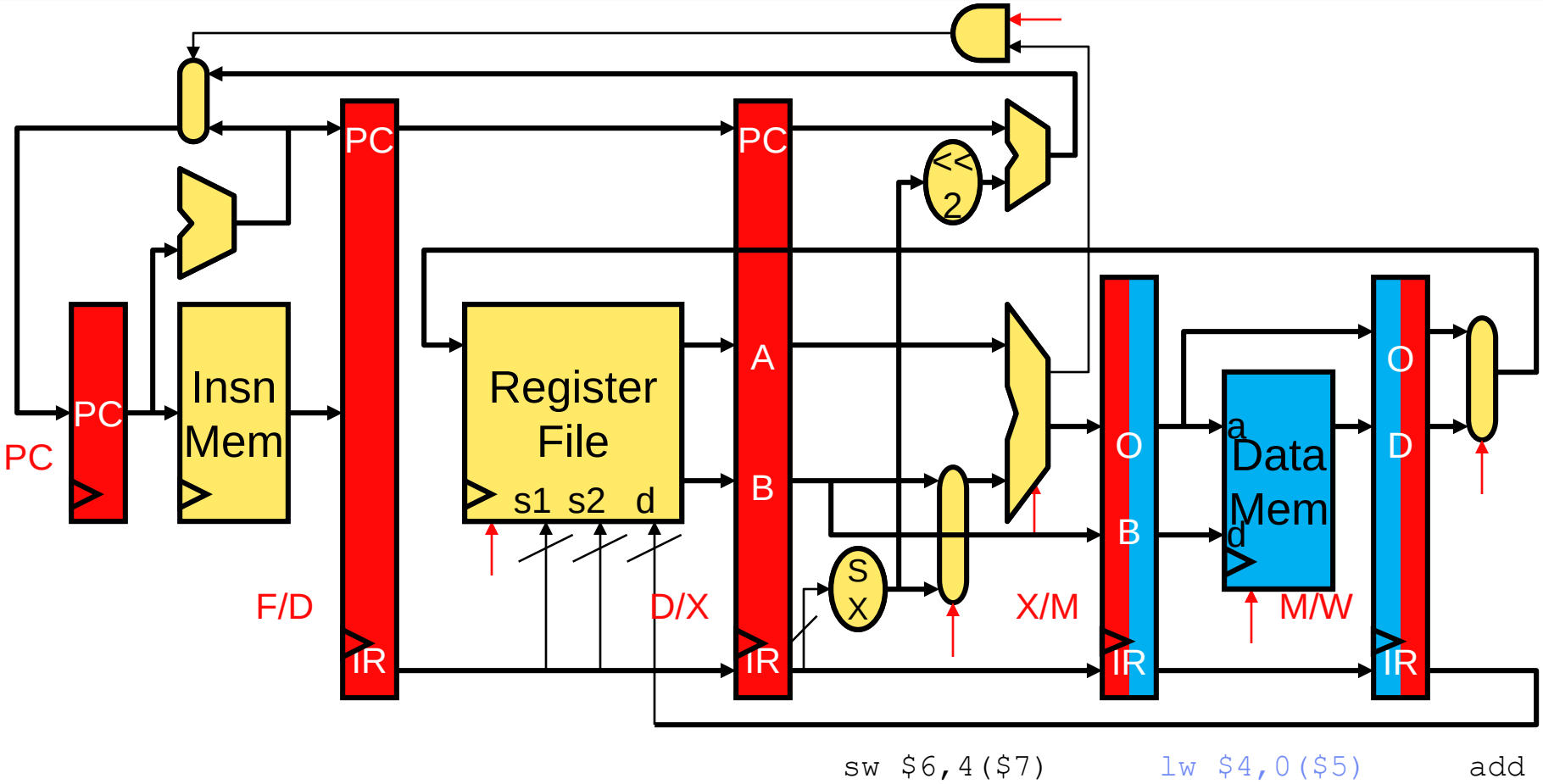
- 3 instructions

Pipeline Example: Cycle 4



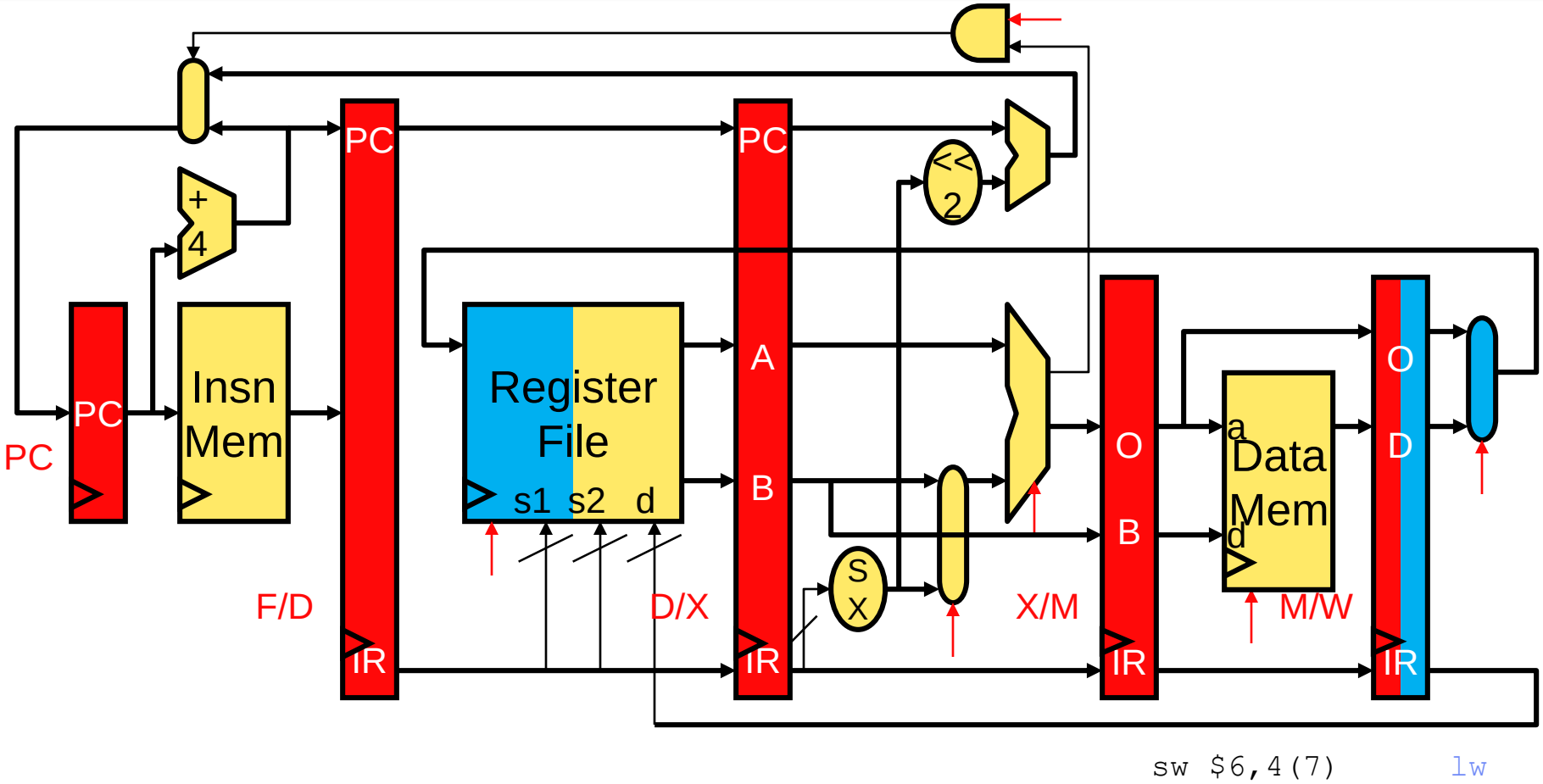
- 3 instructions

Pipeline Example: Cycle 5



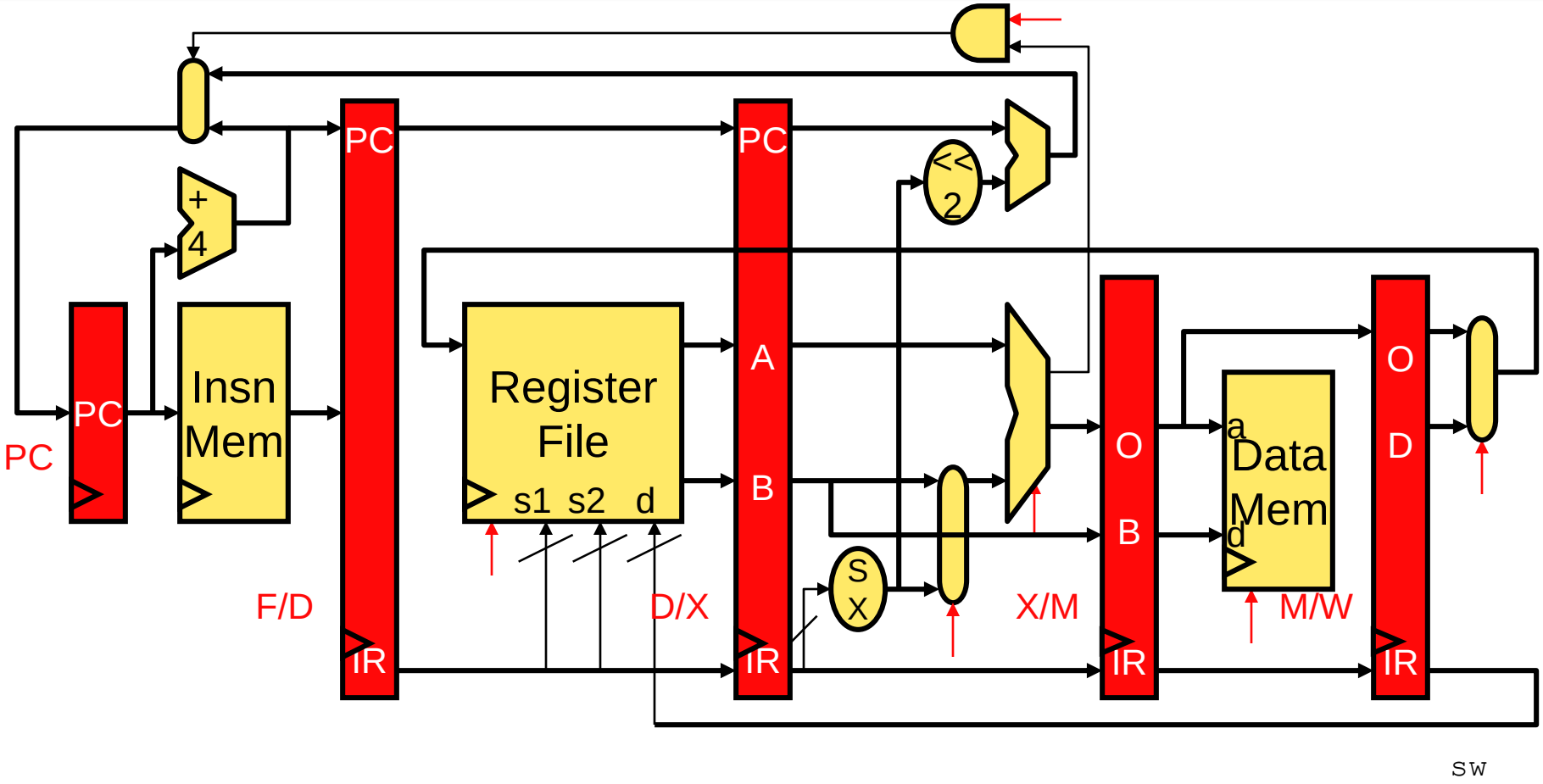
- 3 instructions

Pipeline Example: Cycle 6



- 3 instructions

Pipeline Example: Cycle 7



- 3 instructions

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

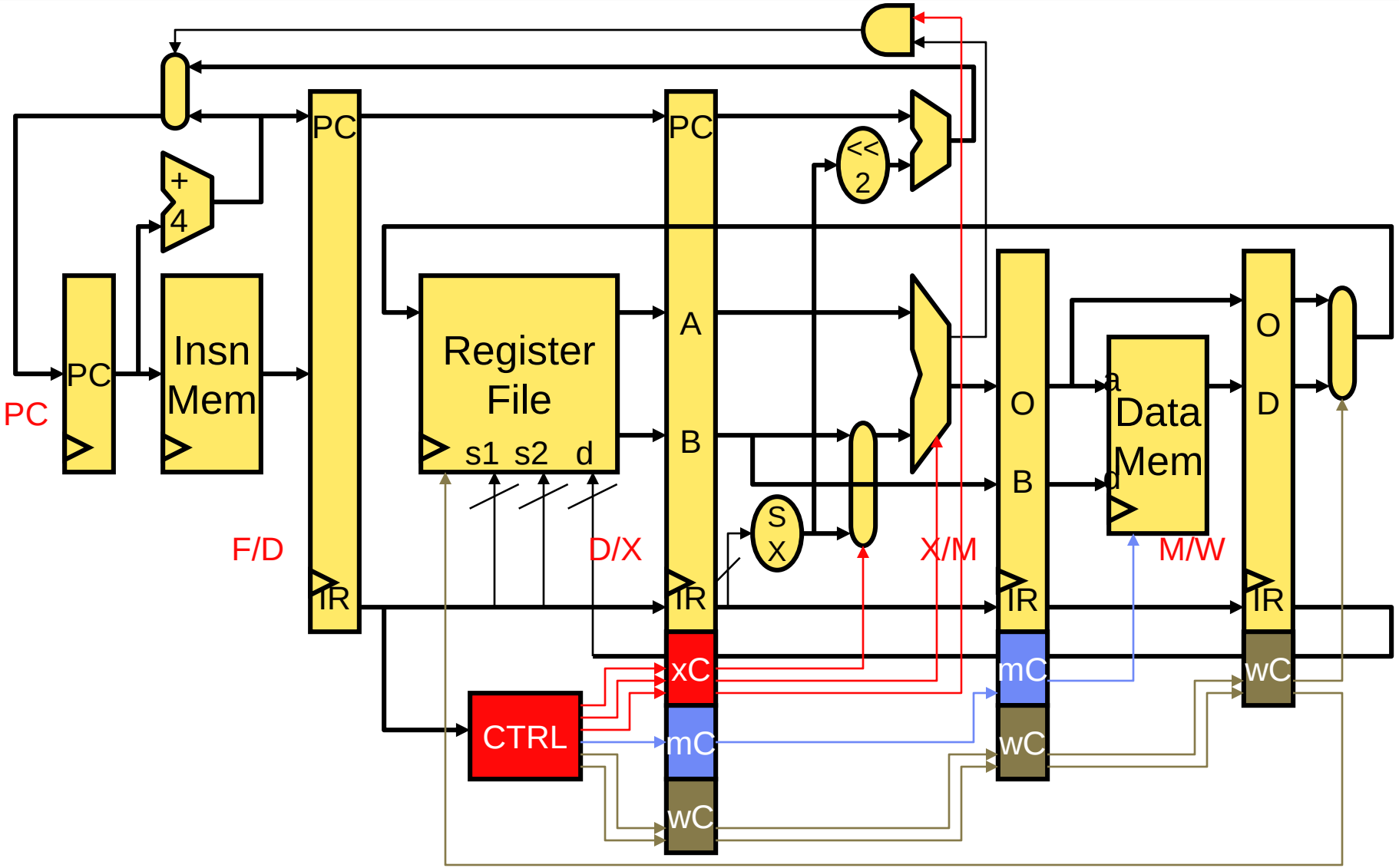
	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,0($5)`		F	D	**X**	M	W			
`sw $6,4($7)`			F	D	X	M	W		

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

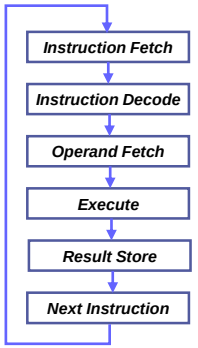
- Single-cycle
  - Clock period = 50ns, CPI = 1
  - Performace = 50ns/insn
- Multi-cycle
  - Branch: 20% (3 cycles), load: 20% (5 cycles), other: 60% (4 cycles)
  - Clock period = **12ns**, CPI = (0.2*3+0.2*5+0.6*4) = 4
    - Remember: latching overhead makes it 12, not 10
  - Performance = **48ns/insn**
- Pipelined
  - Clock period = **12ns**
  - CPI = **1.5** (on average insn completes every 1.5 cycles)
  - Performance = **18ns/insn**

Some questions (1)

- **Why Is Pipeline Clock Period > delay thru datapath / number of pipeline stages?**
  - Latches (FFs) add delay
  - Pipeline stages have different delays, clock period is max delay
  - Both factors have implications for ideal number pipeline stages

Some questions (2)

- **Why Is Pipeline CPI > 1?**
  - CPI for scalar in-order pipeline is 1 **+ stall penalties**
  - Stalls used to resolve hazards
    - **Hazard**: condition that jeopardizes VN illusion
    - **Stall**: artificial pipeline delay introduced to restore VN illusion



VN loop

(What we have to pretend we're doing)

- Calculating pipeline CPI
  - **Frequency of stall** * **stall cycles**
  - Penalties add (stalls generally don't overlap in in-order pipelines)
  - 1 + stall-freq$_1$*stall-cyc$_1$ + stall-freq$_2$*stall-cyc$_2$ + …

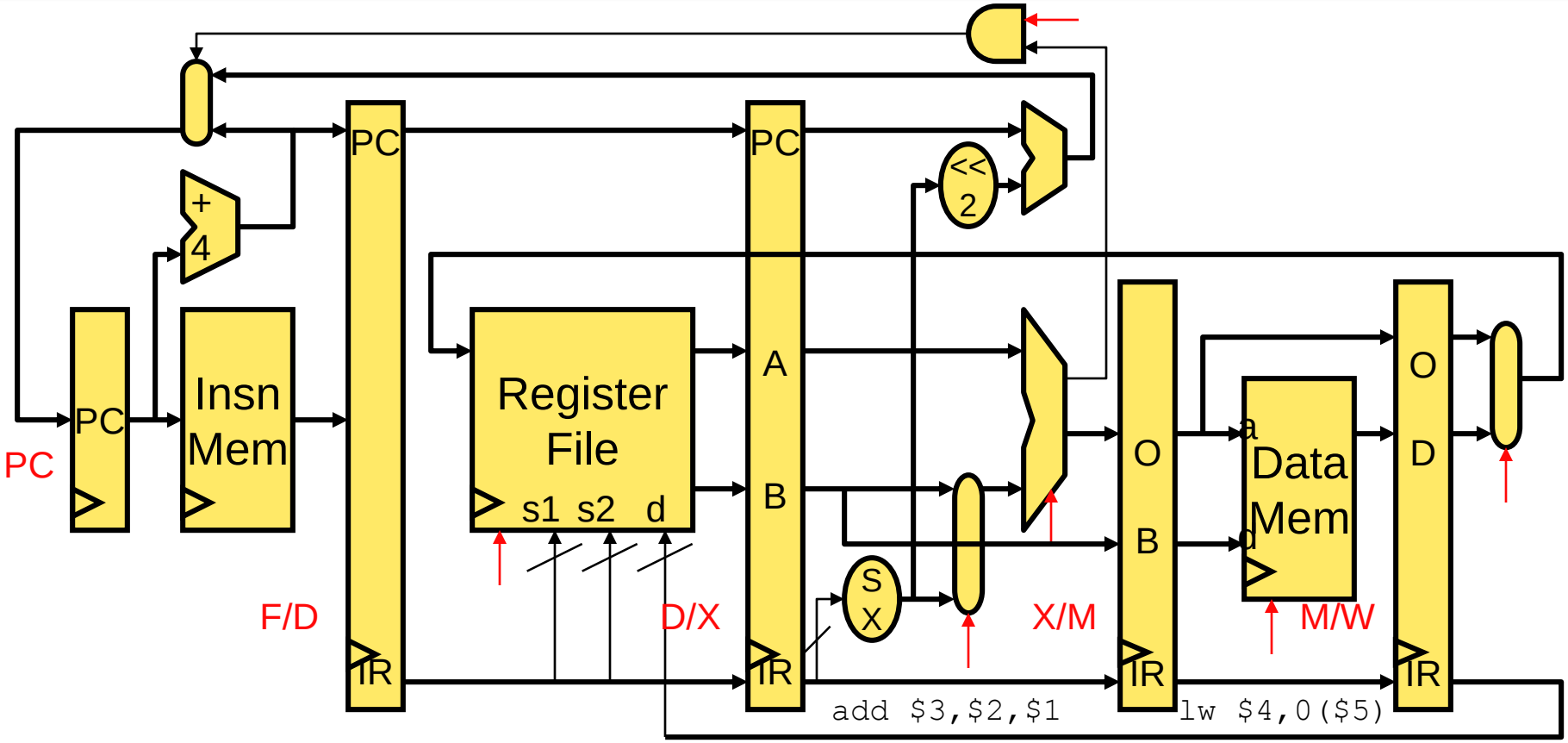
- Correctness/performance
  - Long penalties OK if they happen rarely, e.g., 1 + 0.01 * 10 = 1.1
  - Stalls also have implications for ideal number of pipeline stages

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?



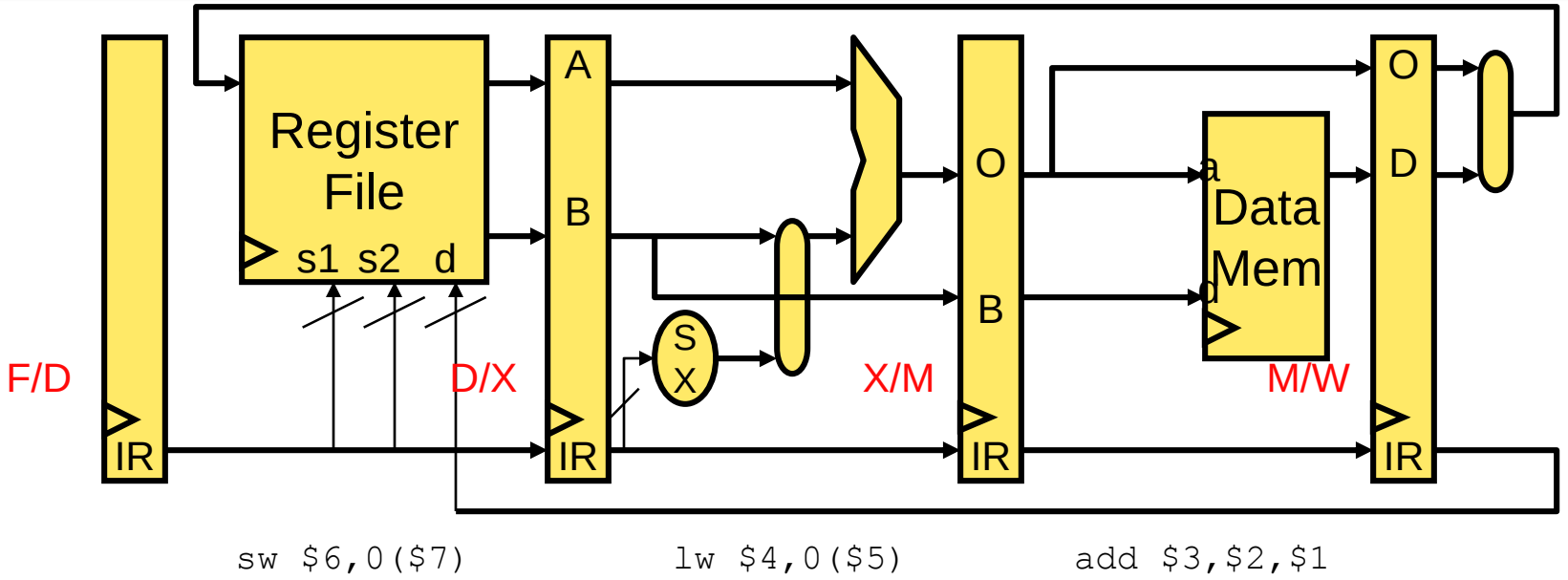
- Could /should we allow **`add`** to skip M and go to W? No
  - It wouldn't help: peak fetch still only 1 insn per cycle
  - **Structural hazards**: imagine **`add`** follows **`lw`**

Structural Hazards

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

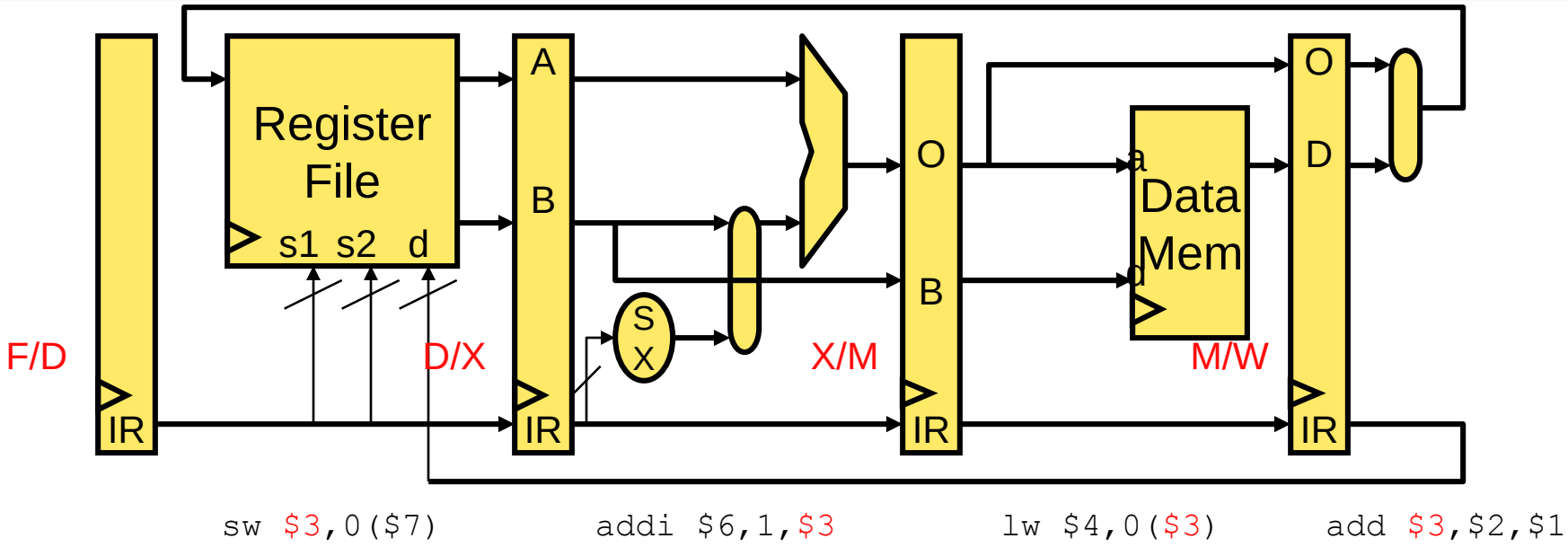
- **To fix structural hazards**: proper ISA/pipeline design
  - Each insn uses every structure exactly once for at most one cycle
  - Always at same stage relative to F

Data Hazards



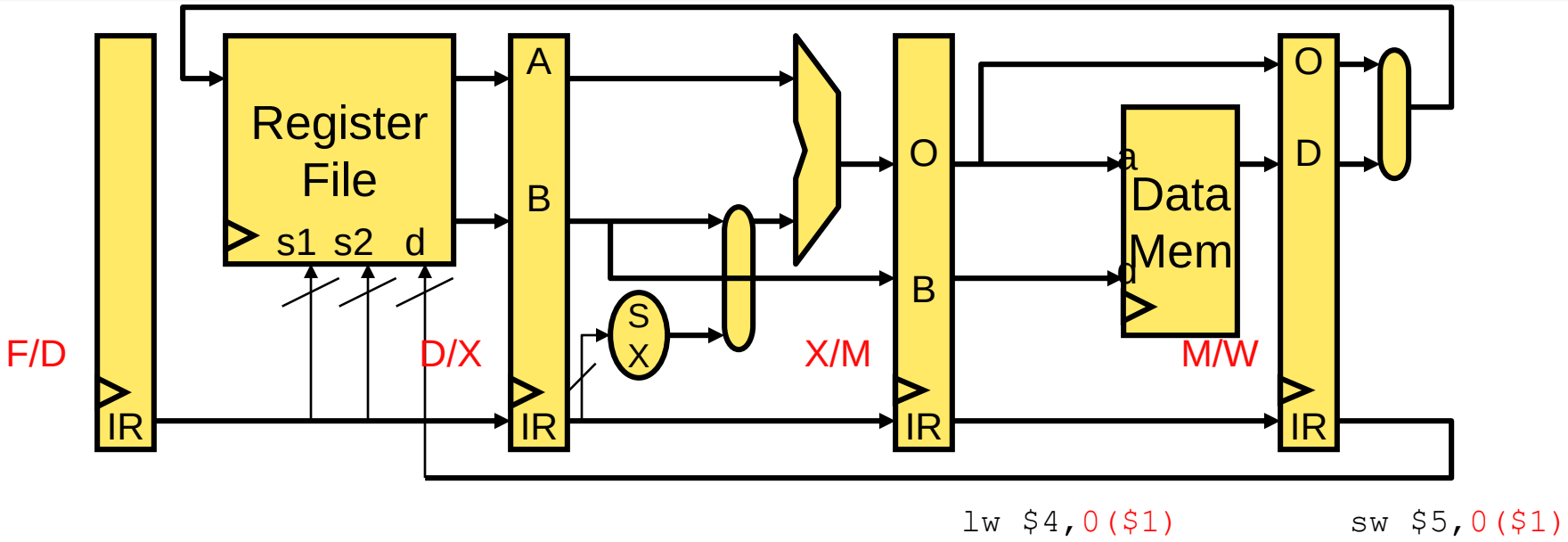
- Let's forget about branches and the control for a while
- The three insn sequence we saw earlier executed fine…
  - But it wasn't a real program
  - Real programs have **data dependences**
    - They pass values via registers and memory

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 (M)
- 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 don'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"
  - **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
    - No hardware interlocks in MIPS

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`

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

Software Interlock Performance

- Same deal
  - Branch: 20%, load: 20%, store: 10%, other: 50%

- 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

Summary

- Pipelining
  - Pipelined datapath
  - Pipeline datapath control
  - Dependences and hazards