ECE 550D
Fundamentals of Computer Systems and Engineering

Fall 2025

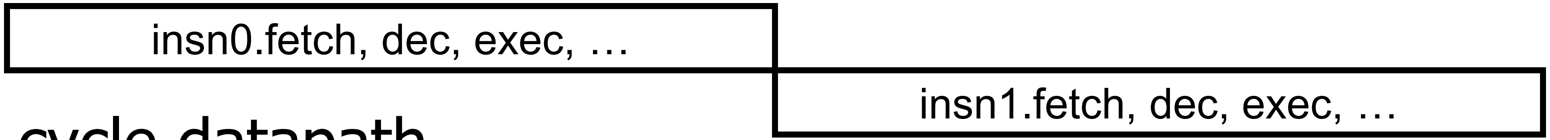
Pipelining

Yiran Chen and Hai “Helen” Li
Duke University

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

Clock Period/Frequency and CPI/IPC

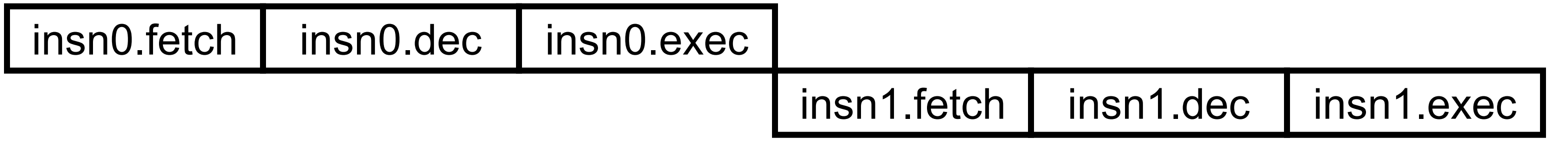
- Throughput = IPC x Frequency = 1/(CPI x Period)
- Single-cycle datapath
  - Low CPI: 1
  - Long clock period: to accommodate slowest insn



insn0.fetch, dec, exec, …

insn1.fetch, dec, exec, …

- Multi-cycle datapath
  - Short clock period (high frequency)
  - High CPI (low IPC)

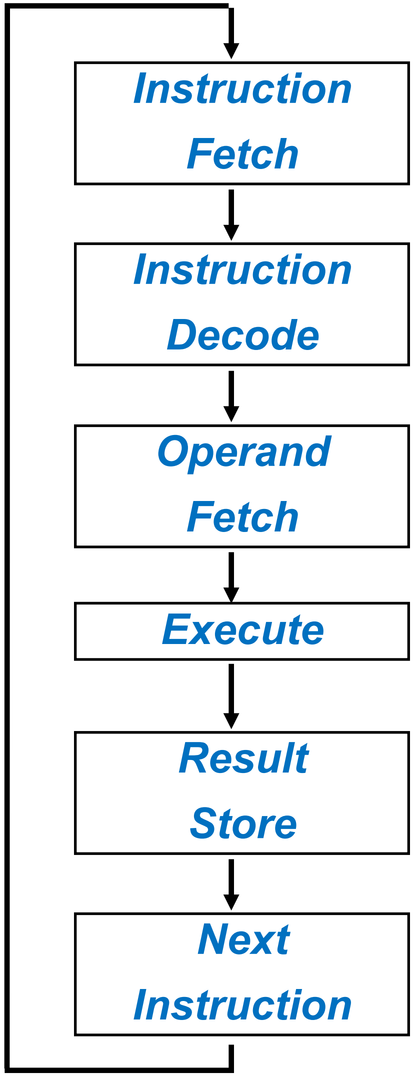


insn0.fetch | insn0.dec | insn0.exec

insn1.fetch | insn1.dec | insn1.exec

- 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 numbers)

- Result Store:

  Write result to register or memory

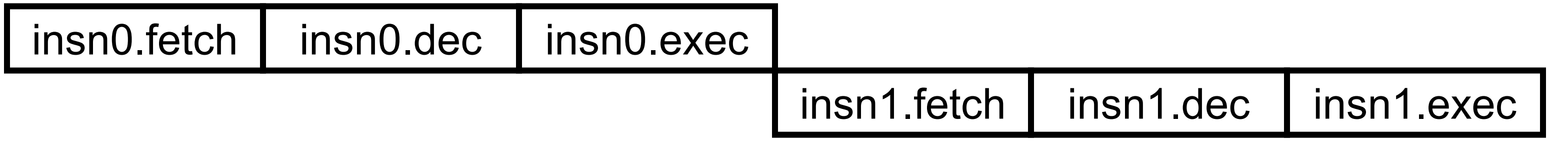
- Next Instruction:

  Figure out mem addr of next insn, repeat

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

Pipelining

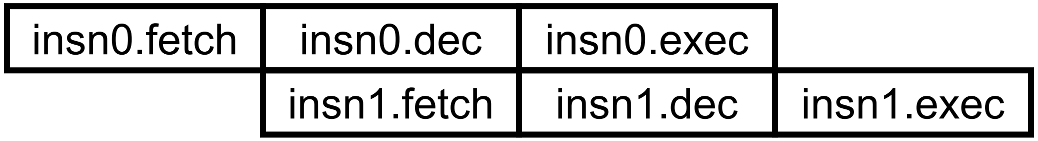
- In multi-cycle design:



insn0.fetch | insn0.dec | insn0.exec

insn1.fetch | insn1.dec | insn1.exec

- In a pipelined design:

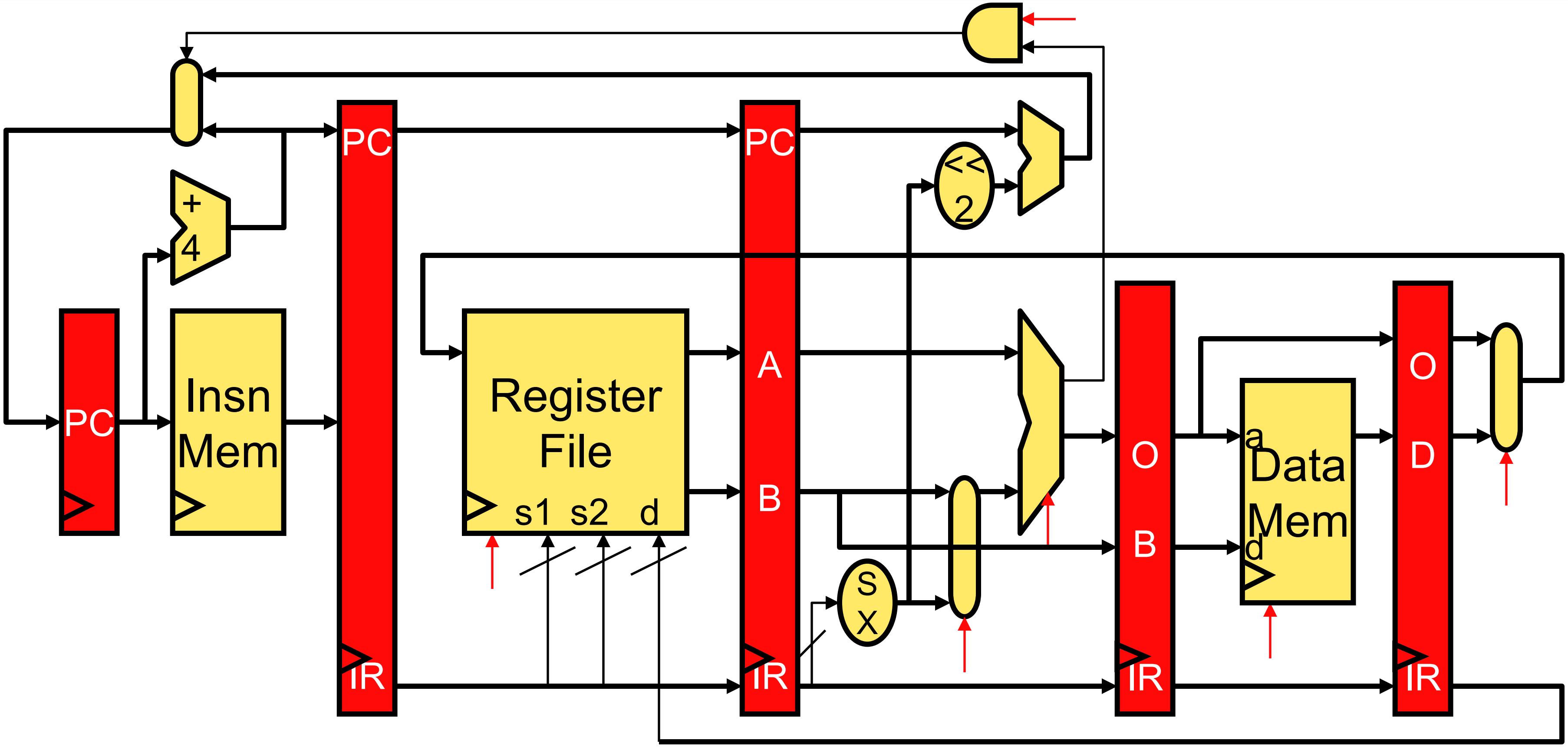


insn0.fetch | insn0.dec | insn0.exec

insn1.fetch | insn1.dec | insn1.exec

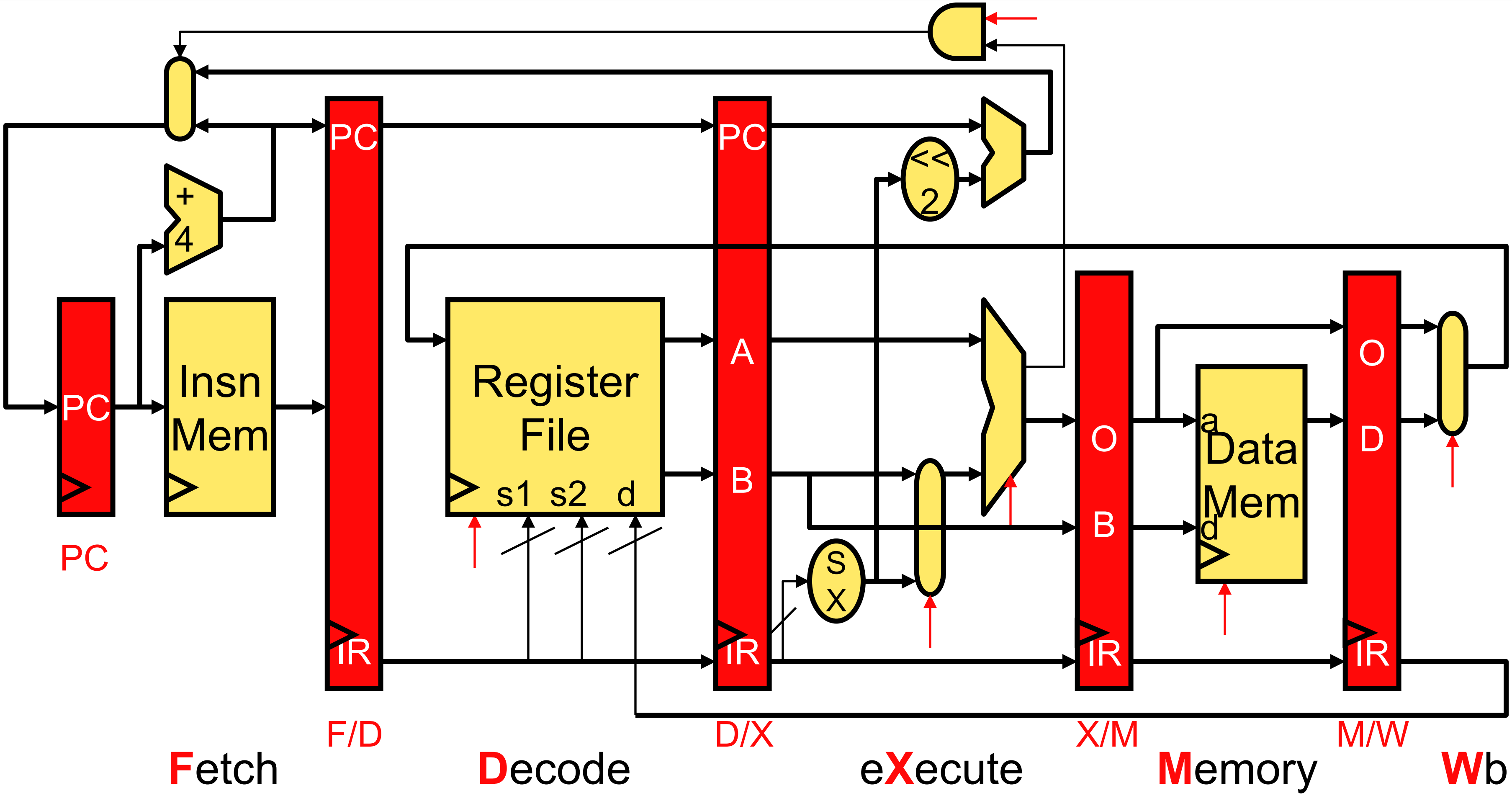
  - **Improves insn throughput rather than insn latency**
  - **Exploits parallelism at insn-stage level to do so**
  - **Individual insns take same number of stages**
  + **But insns enter and leave at a much faster rate**
  - Breaks “atomic” VN loop, but maintains the illusion
- *Analogy: automotive assembly line*
- Challenges?

5-Stage-Pipelined MIPS Datapath



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

Pipeline Terminology



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

Some More Terminology

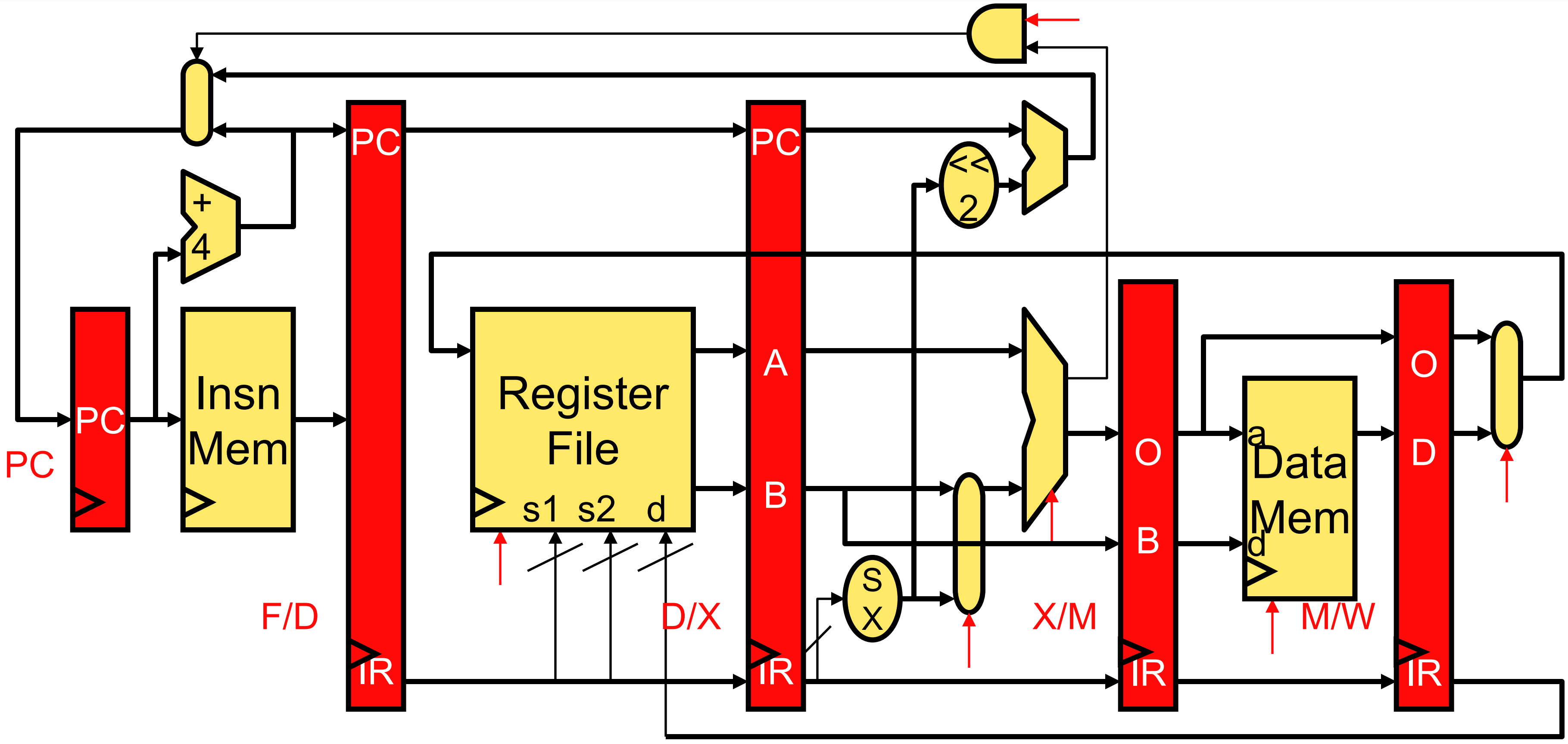
- **Scalar pipeline:** one insn per stage per cycle
  - Alternative: “superscalar” (check ECE 552)

- **In-order pipeline:** insns enter execute stage in VN order
  - Alternative: “out-of-order” (check ECE 552)

- **Pipeline depth:** number of pipeline stages
  - Nothing magical about 5
  - Trend has been to have deeper pipelines

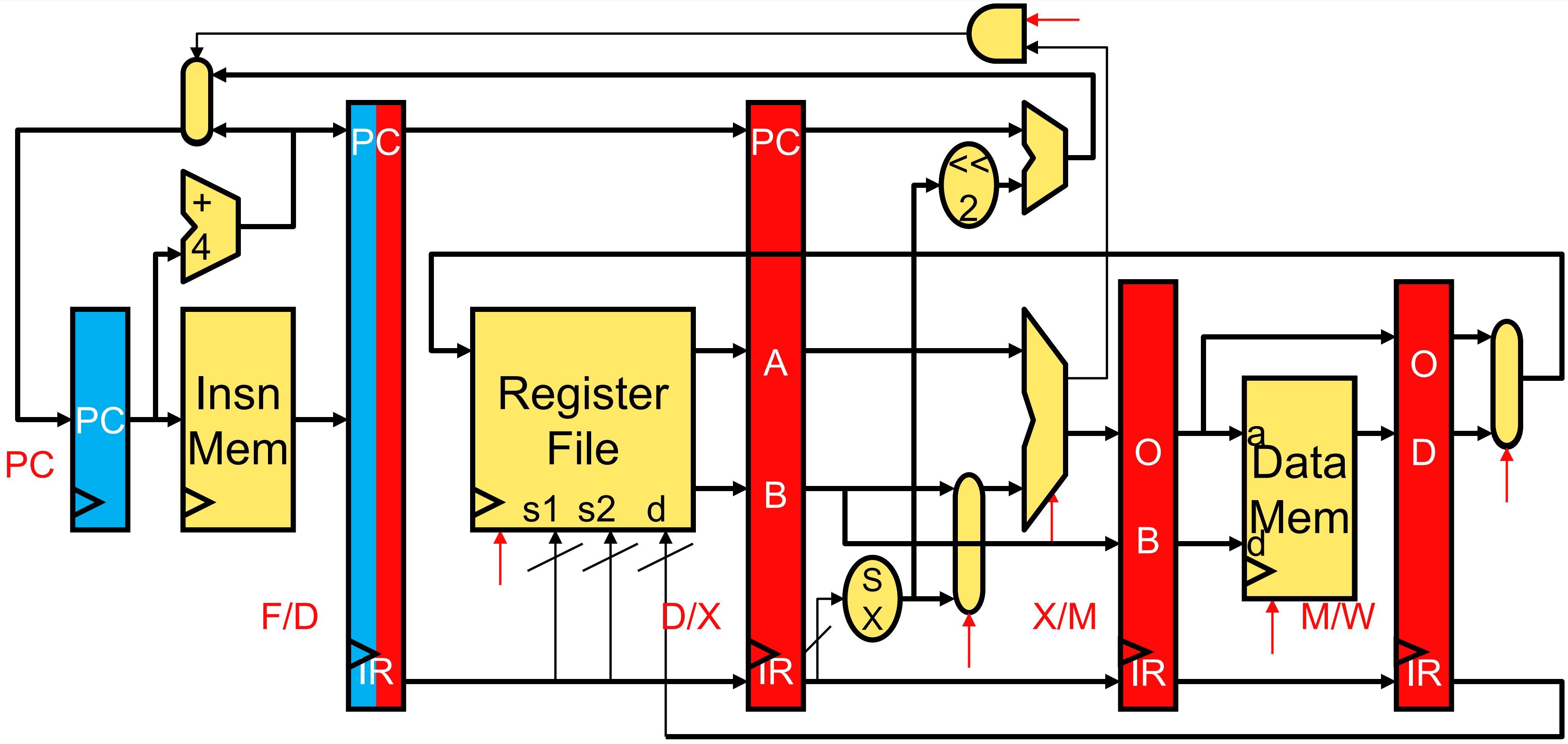
- → our MIPS pipeline is a scalar in-order 5-stage pipeline

Pipeline Example: Cycle 1



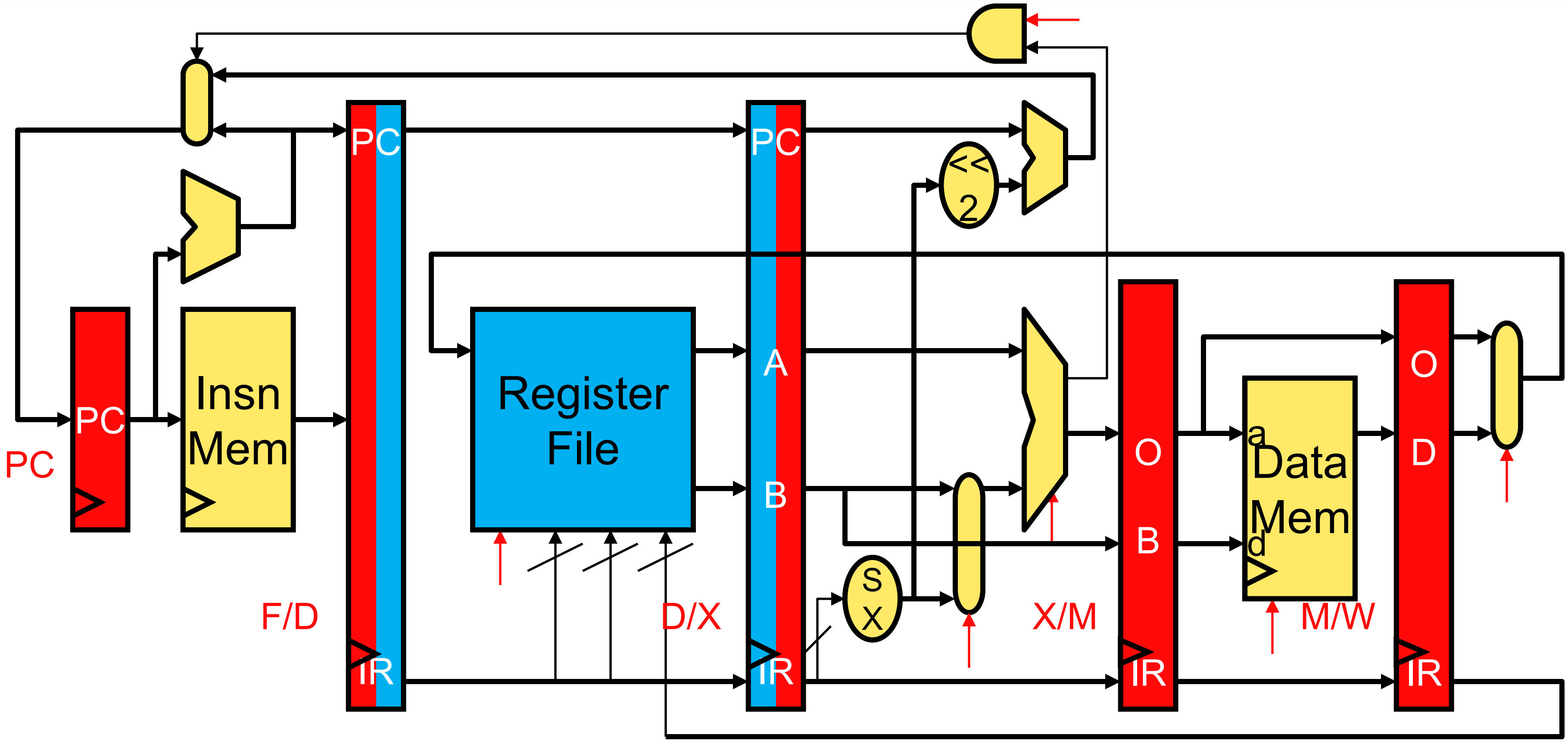
add $3,$2,$1

Pipeline Example: Cycle 2



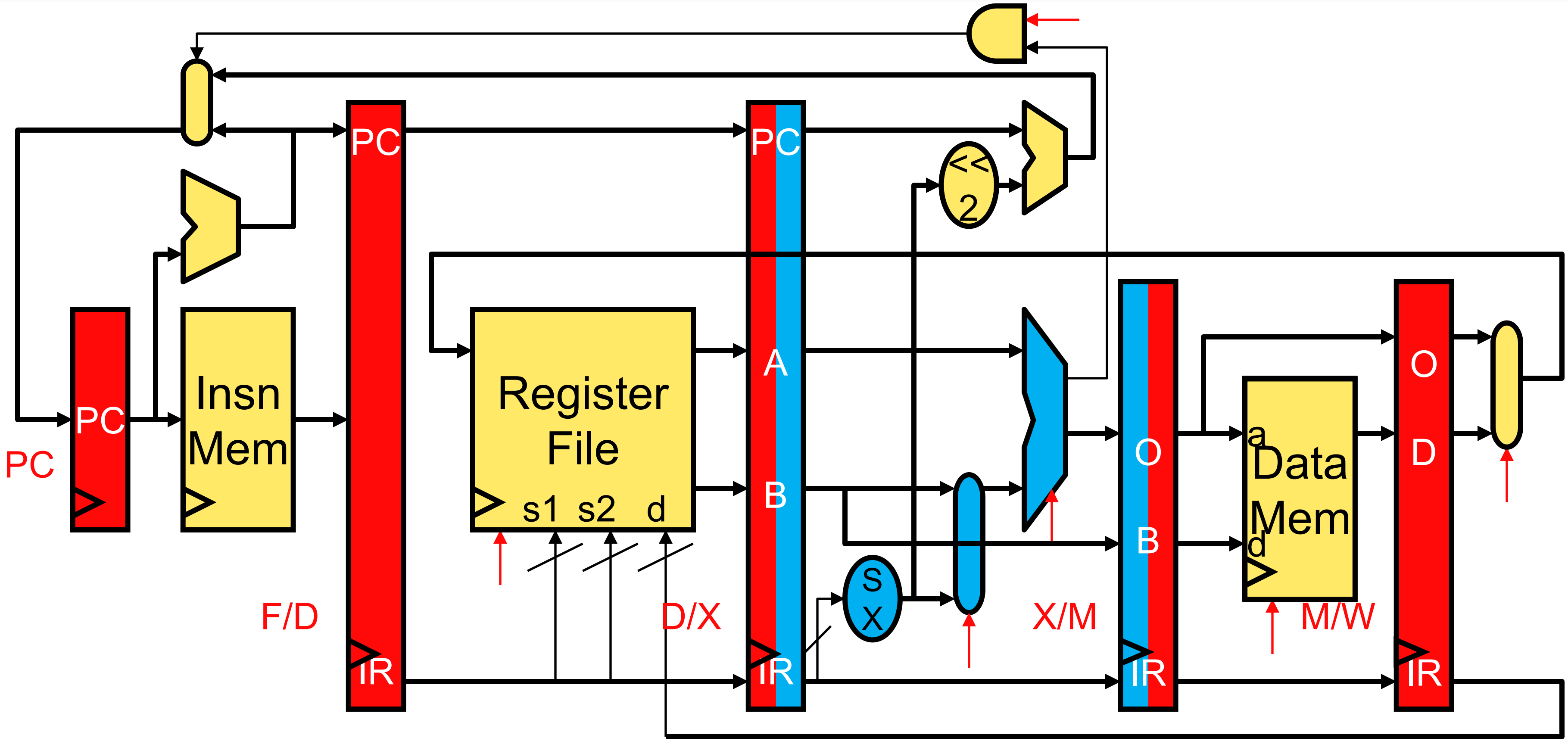
lw $4,0($5)   add $3,$2,$1

Pipeline Example: Cycle 3



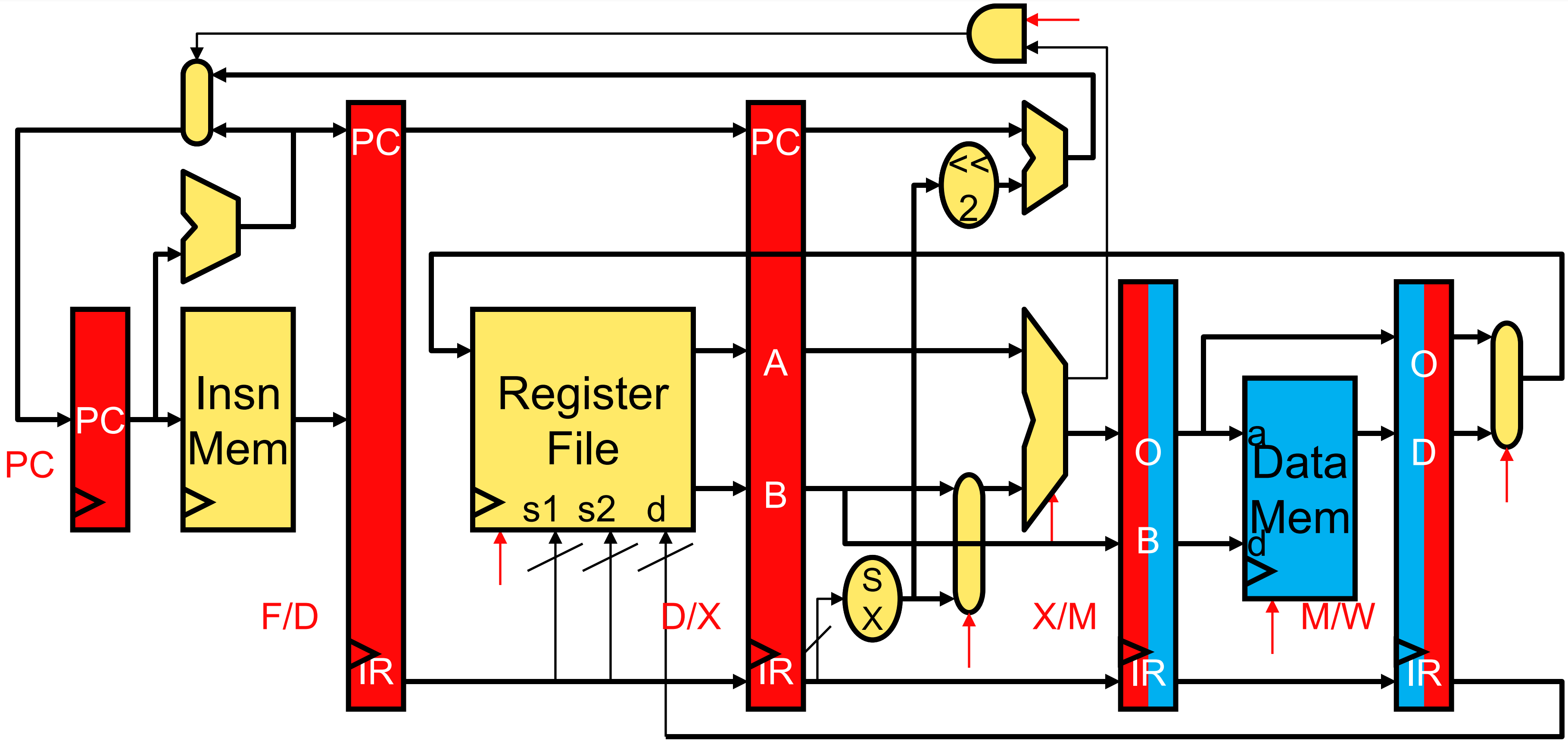
sw $6,4($7) lw $4,0($5) add $3,$2,$1

Pipeline Example: Cycle 4



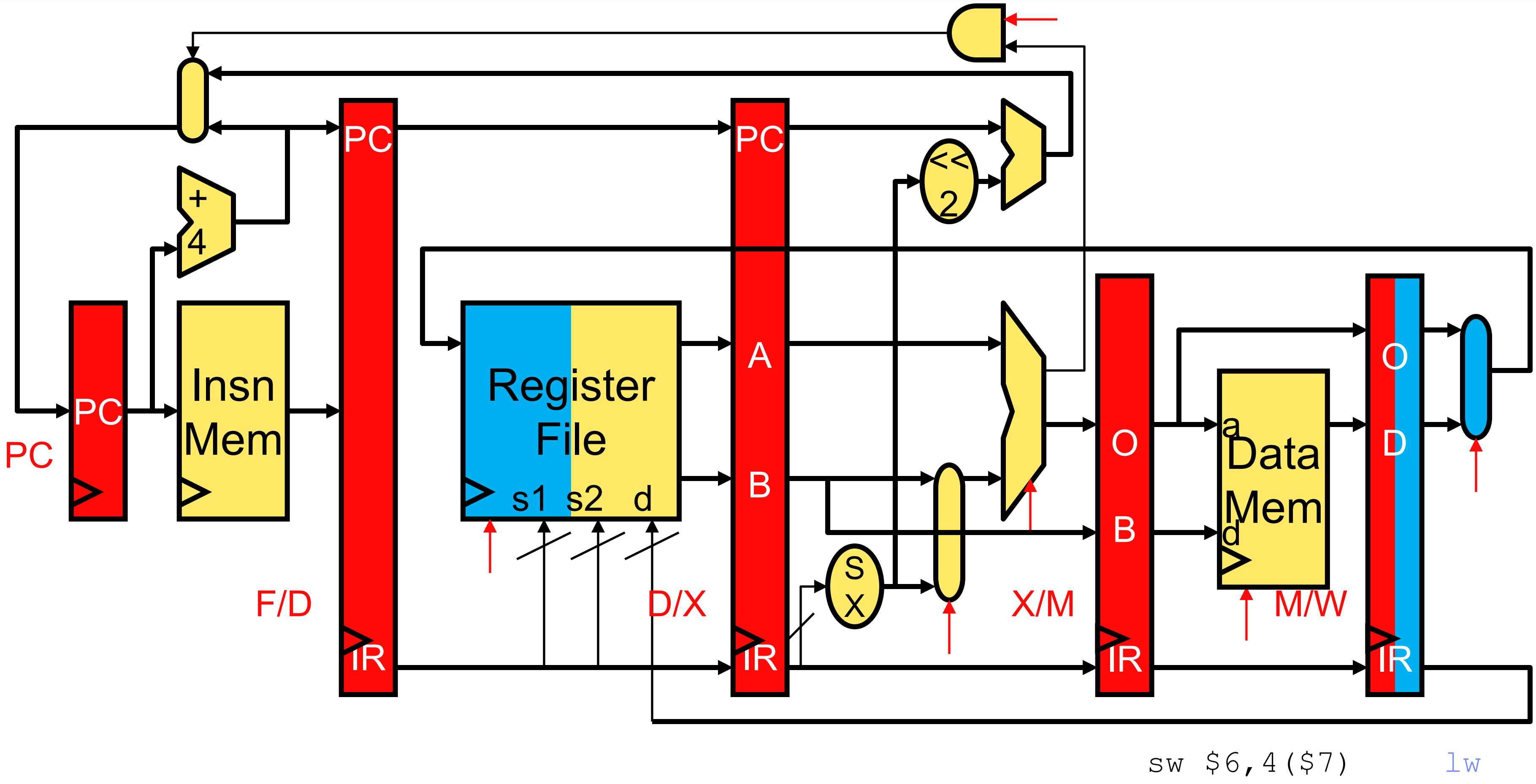
sw $6,4($7) lw $4,0($5) add $3,$2,$1

Pipeline Example: Cycle 5

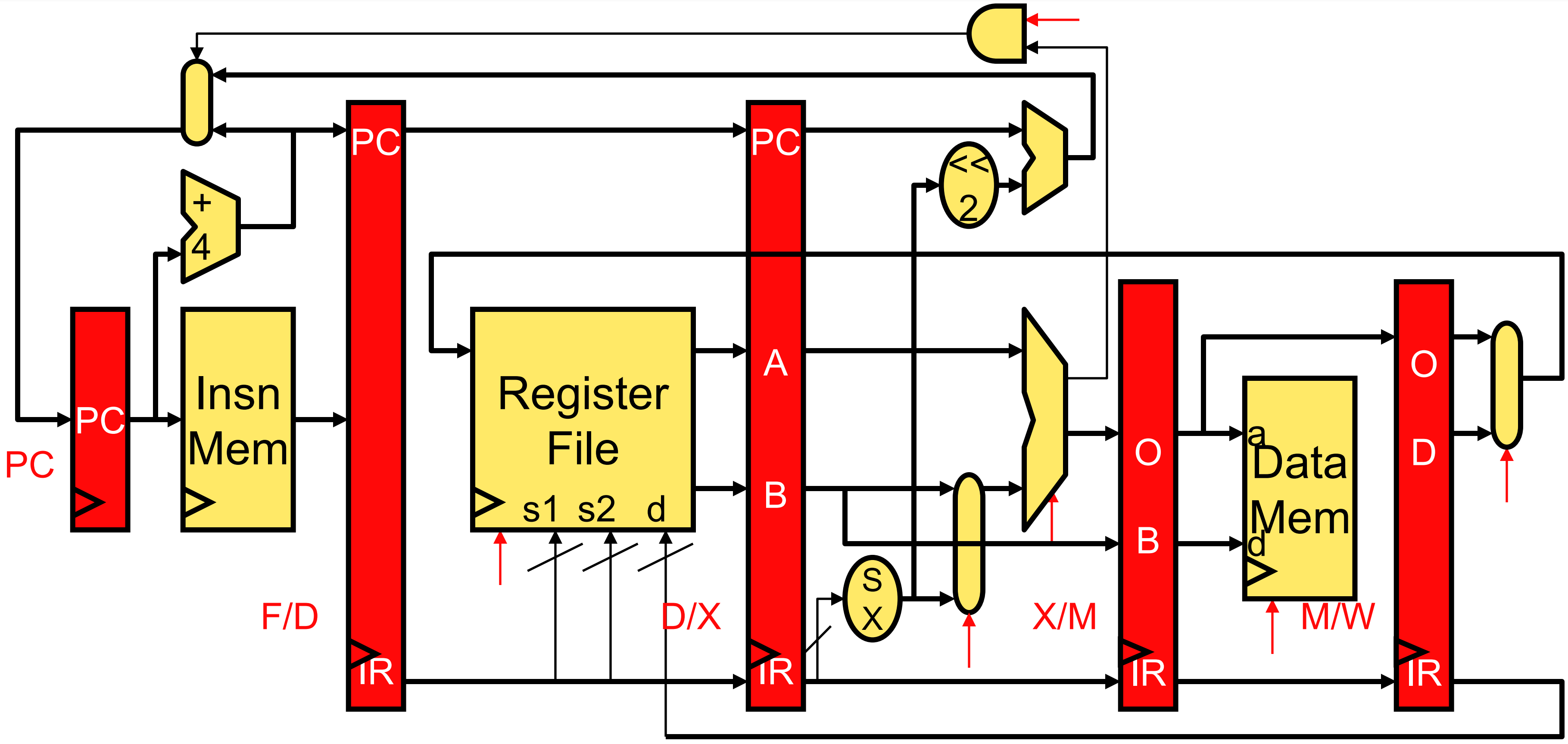


sw $6,4($7)   lw $4,0($5)   add

Pipeline Example: Cycle 6



Pipeline Example: Cycle 7



sw

Pipeline Diagram

- Shorthand for what we just saw

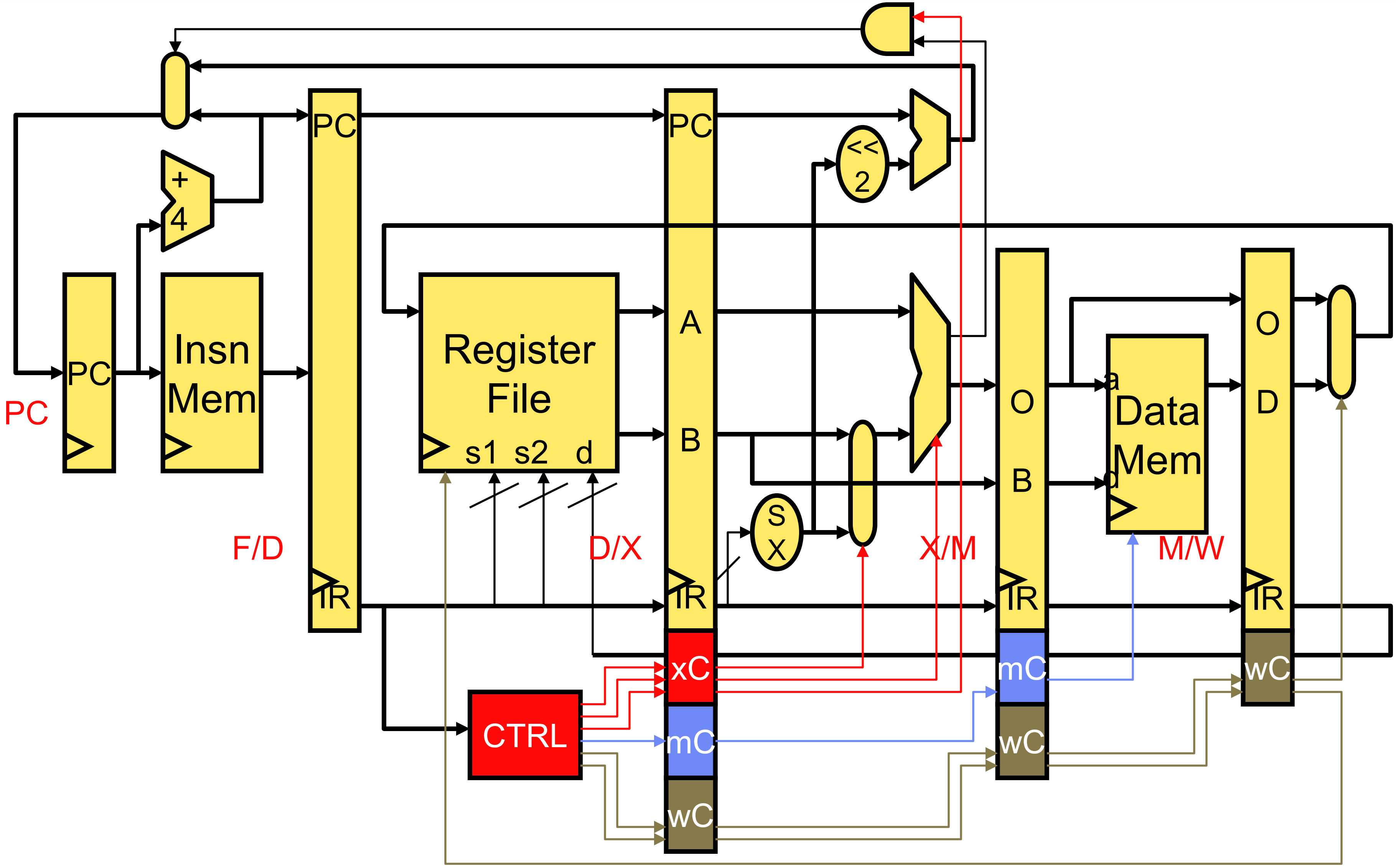
	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,0($5)`		F	D	**X**	M	W			
`sw $6,4($7)`			F	D	X	M	W		

- Columns: clock cycle number
- Rows: insn
- Cells: stage
- Convention: **X** means `lw $4,0($5)` finishes execute stage and writes into X/M latch at the end of cycle 4

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 the control signals with the insn through the pipeline

Pipelined Control



Pipeline Performance Example

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

From previous lecture

- 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**
    - *this will be even worse when accounting for a clock period that accommodates for the slowest stage

- Pipelined
  - Clock period = **12ns**
  - CPI = **1.5** (on average insn completes every 1.5 cycles)
  - Performance = **18ns/insn**

Some Questions…

- **Why is pipeline clock period > (delay through single-cycle dp / # stages)?**

  - Registers add delay
  - Pipeline stages have different delays → clock period should accommodate for the slowest stage

  - Note that both factors have implications on the ideal number of pipeline stages

Some Questions…

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

pronounced “no-op”

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

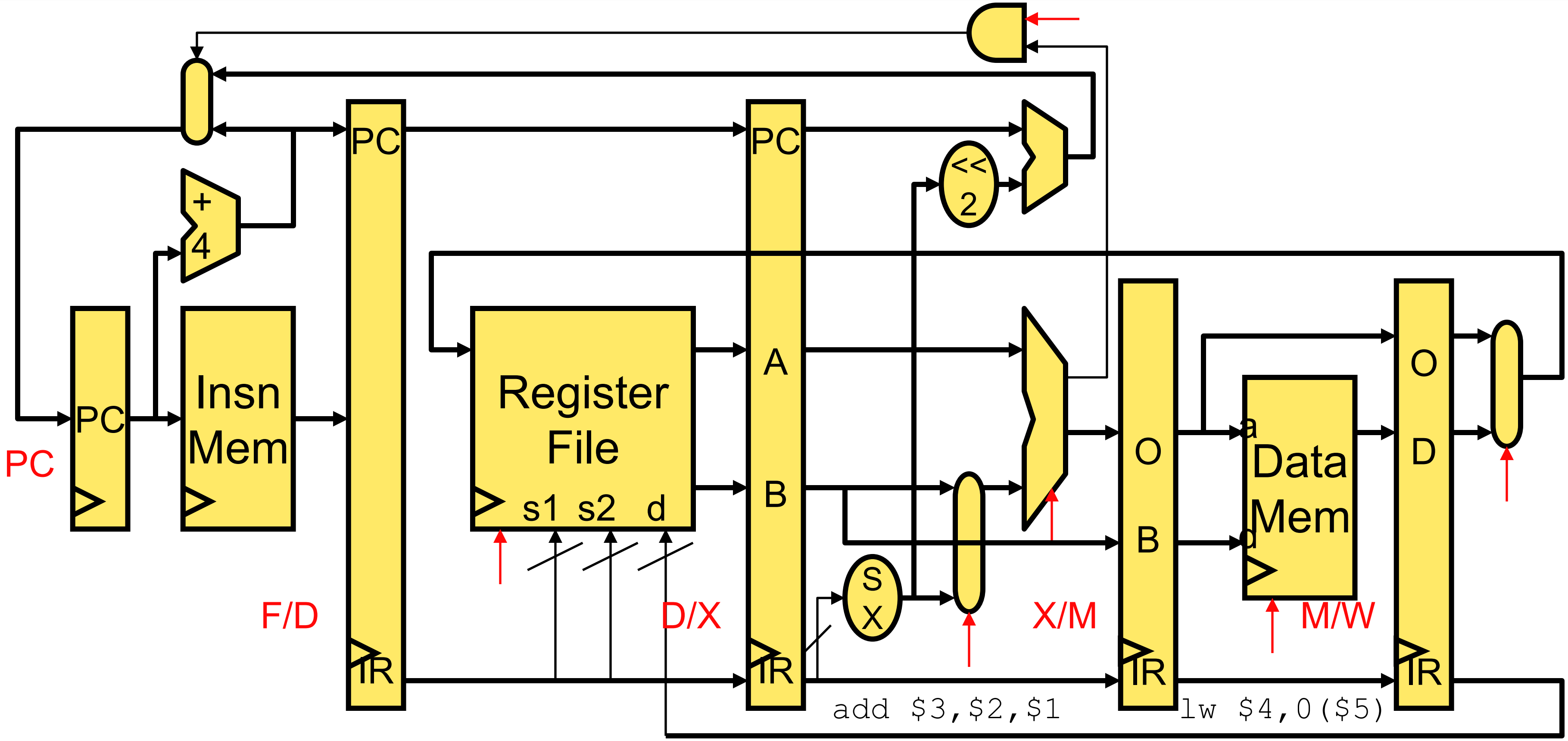
Dependences and Hazards

- **Dependence**: relationship between two insns
  - **2 types:**
    - **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.
  - Not a problem in single-/multi-cycle designs
  - But we must account for it in a pipeline

- **Hazard**: dependence leading to a wrong outcome/state
  - Leads to stalling the pipeline → reduce performance
  - **3 types of hazards:**
    - **Structural**: due to datapath restrictions
    - **Data**: due to specific data dependences
    - **Control**: due to specific control dependences

Structural Hazards

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 is still only 1 insn per cycle
  - **→ Structural hazard**

Structural Hazards

- **Structural hazards**
  - Two insns trying to use same circuit at same time

- **To fix structural hazards**: proper ISA/pipeline design
  - Each insn uses every structure exactly once for at most one cycle
  - If we want to accommodate for a different behavior, we should somehow duplicate hardware

Data Hazards

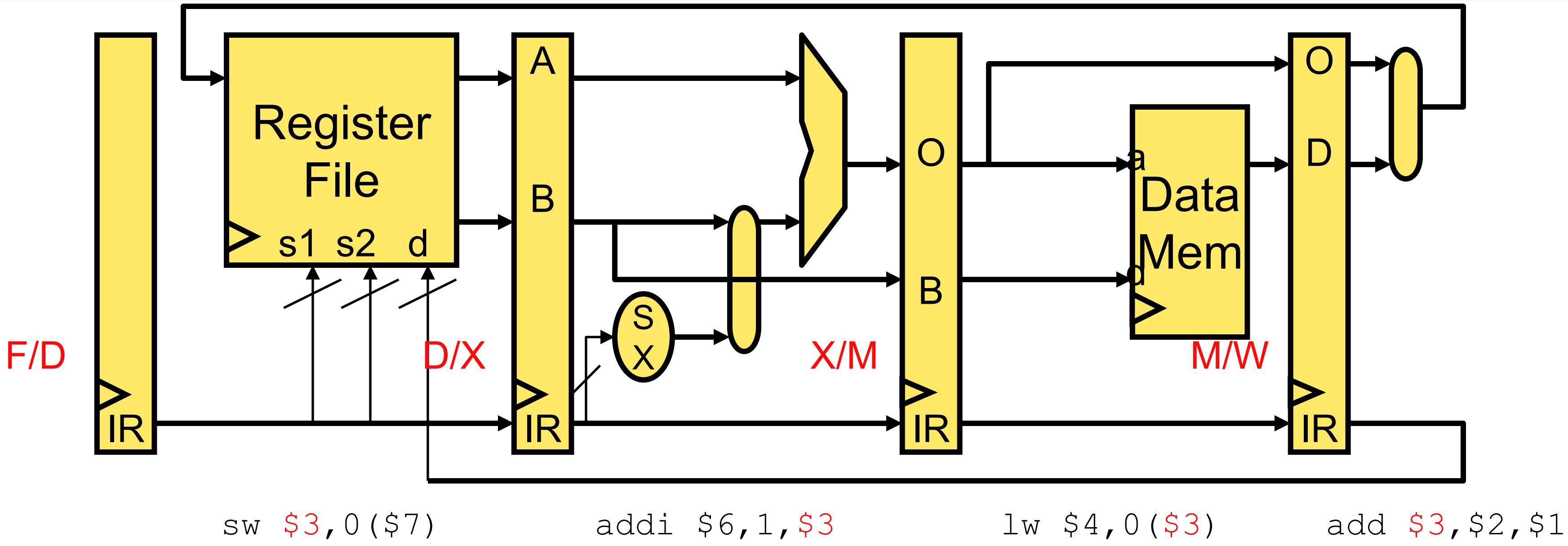
Data Hazards

- *Let's forget about branches and the control for a little bit…*

- Programs have **data dependences**
  - They pass values via registers and memory
  - 4 types of data dependencies
    - RAR (Read After Read)
    - RAW (Read After Write)
    - WAR (Write After Read)
    - WAW (Write After Write)

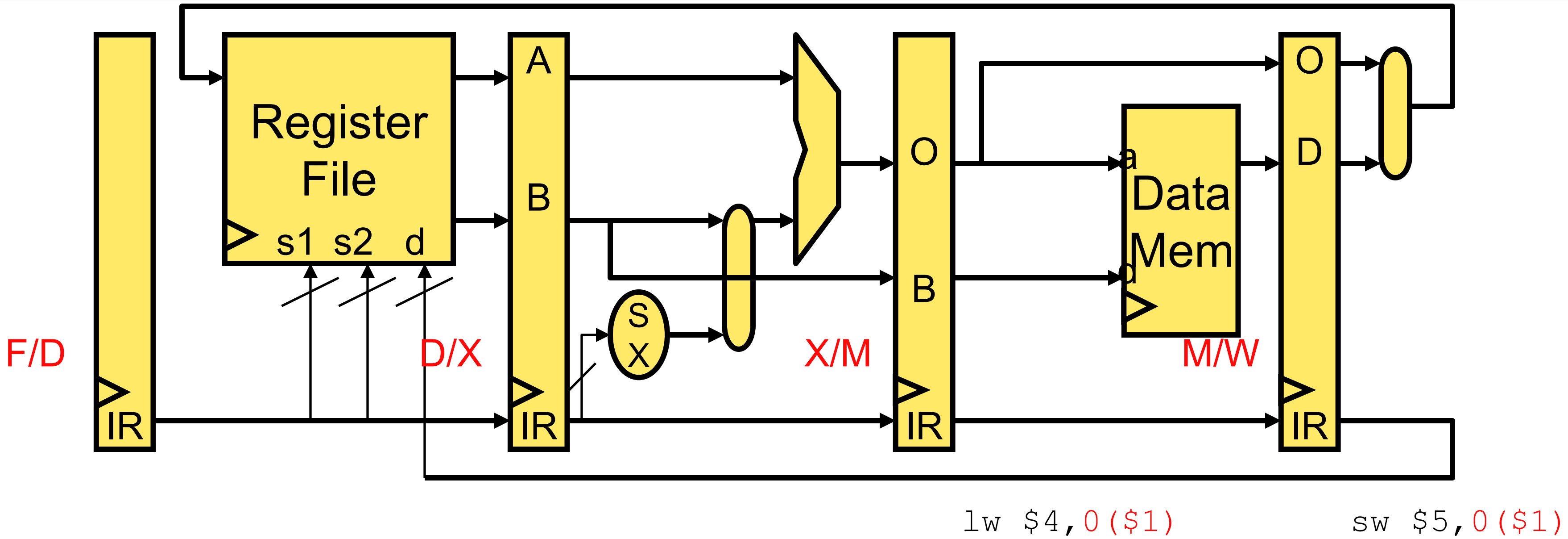
- Do all data dependences lead to hazards?
  - We should just check for RAW-related hazards in this case

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 (if regfile writes first, reads second)

Memory & RegFile 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 the same stage
- Data hazards through registers?
  - Yes (previous slide)
  - Occur because regfile write is 3 stages after regfile read

Fixing RegFile Data Hazards

- Make a new rule, which is:
  - We 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 the reg in question"
    - Compiler moves around existing insns to do this
      - Called **code scheduling**
    - If none can be found, insert **NOPs** between data-dependent insns
    - This is called **software interlocks**

Software Interlock Example

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

Problem(s)?

- Insns 1 and 2 need to be separated by 2 insns
- Can any of the last 3 insns be scheduled between the first two?
  - **`sw $7,0($3)`**? No, because it creates hazard with **`sub $3,$2,$1`**
  - **`add $6,$2,$8`**? Yes
  - **`addi $3,$5,4`**? Yes
    - This one isn't that straight forward
    - When in the pipeline, **`lw`** and **`sw`** read their "old" **`$3`** value before **`addi`** writes its "new" **`$3`** value in the W stage, so the code's behavior is still the same as originally intended

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

Software Interlock Performance

- Using same values of earlier example:
  - 20% of insns require insertion of 1 **`nop`**
  - 5% of insns require insertion of 2 **`nops`**

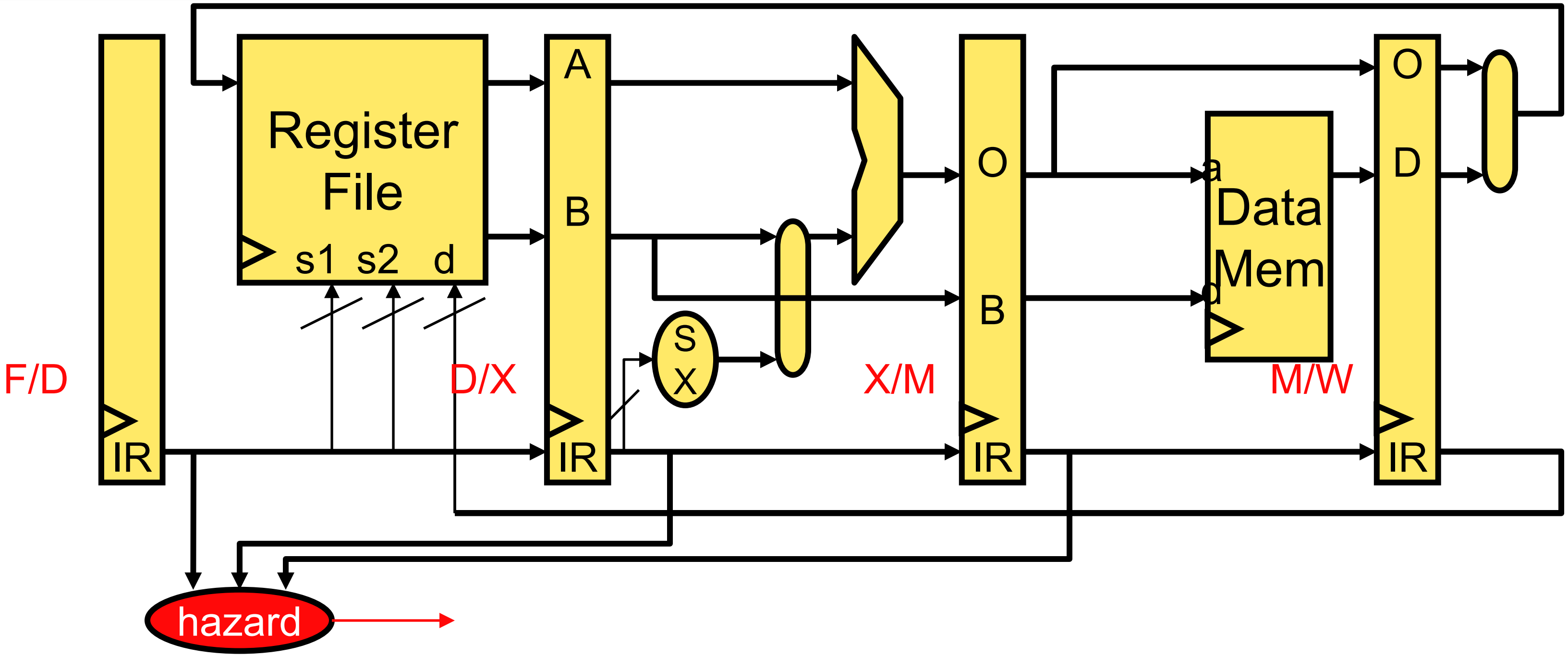
  - CPI is still 1 technically
  - But now there are more insns
  - #insns = 1 + 0.20*1 + 0.05*2 = **1.3**
  - **30% more insns (30% slowdown) due to data hazards**

Hardware Interlocks

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

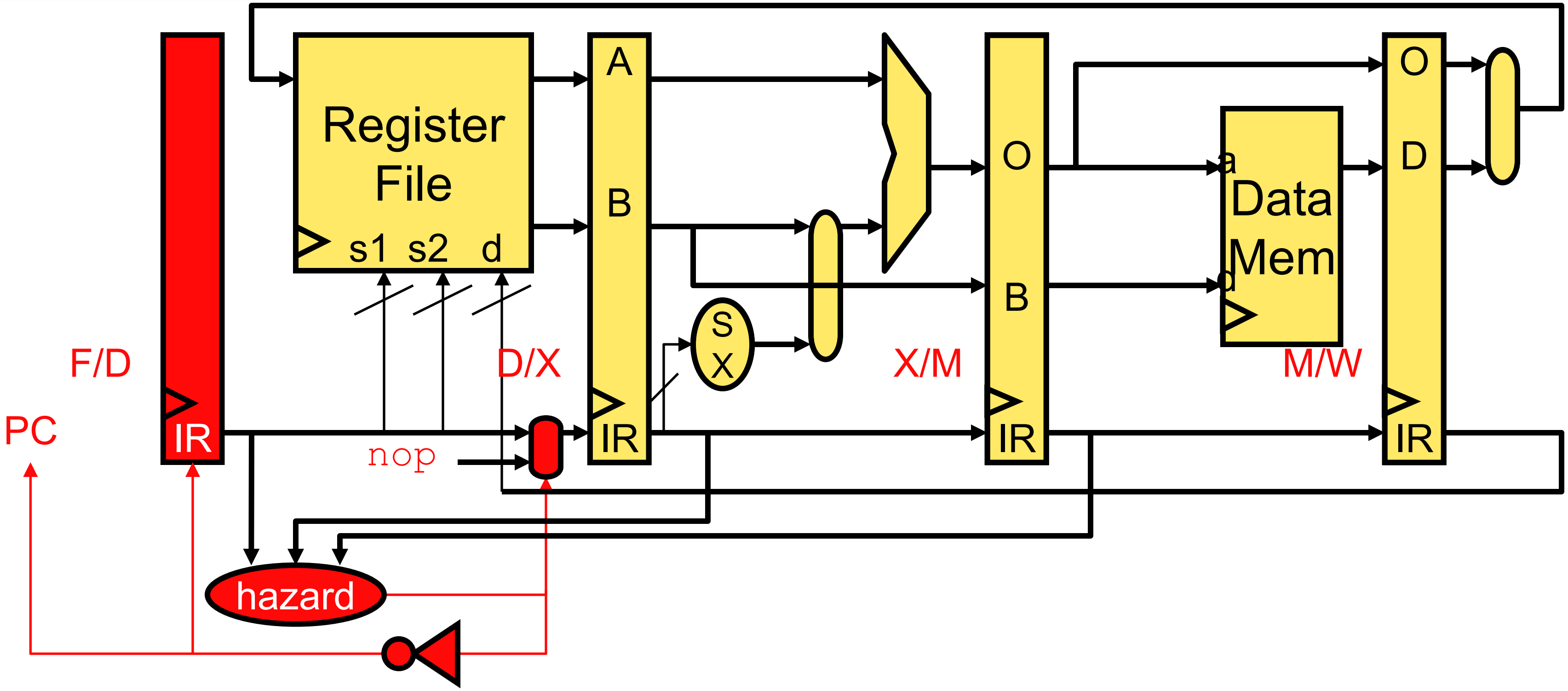
- A better (more compatible) way: **hardware interlocks**
  - Processor detects data hazards and fixes them
  - Two aspects to this:
    1. Detecting hazards
    2. Fixing hazards

Detecting Data Hazards



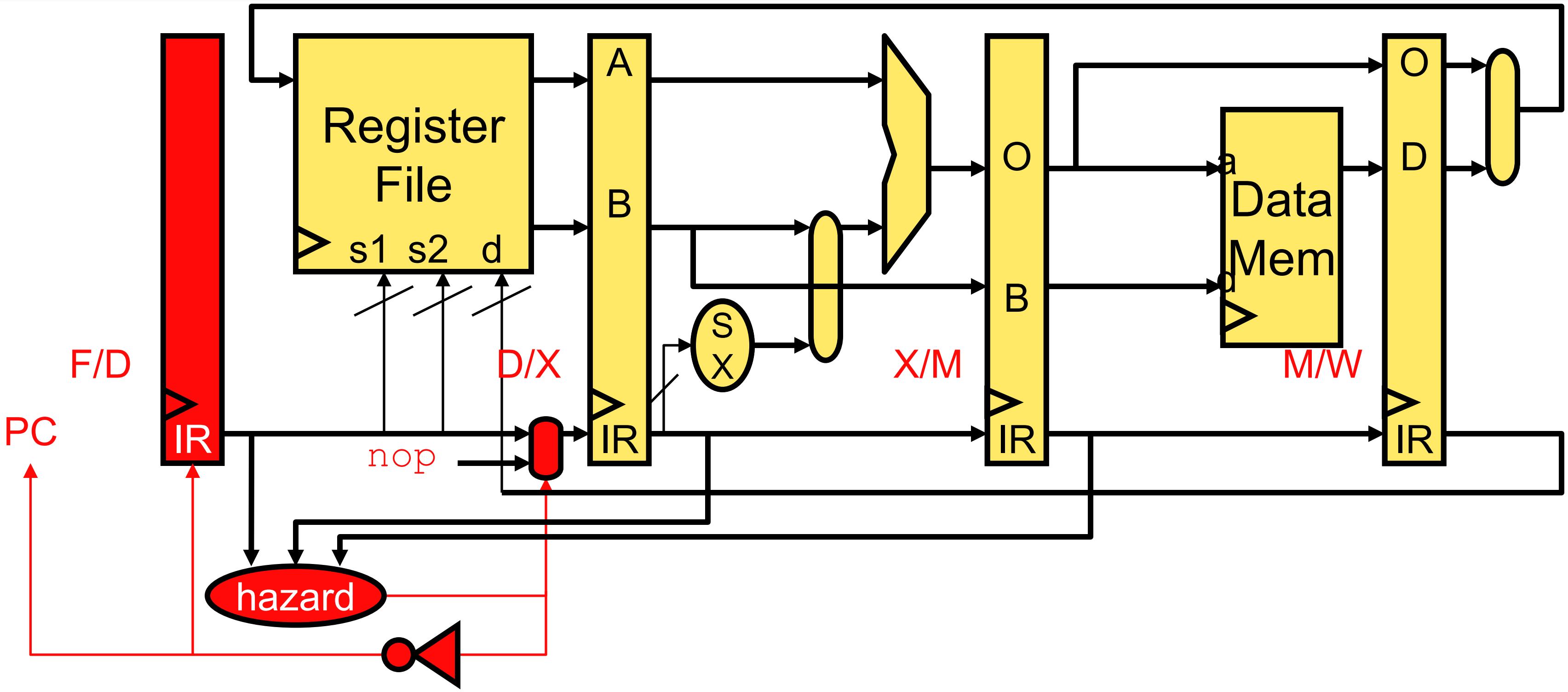
- Compare F/D insn input register names with output register names of older insns in pipeline
  - Hazard =
    - (F/D.IR.RS1 == D/X.IR.RD) || (F/D.IR.RS2 == D/X.IR.RD) ||
    - (F/D.IR.RS1 == X/M.IR.RD) || (F/D.IR.RS2 == X/M.IR.RD)

Fixing Data Hazards



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

Hardware Interlock Example: cycle 1



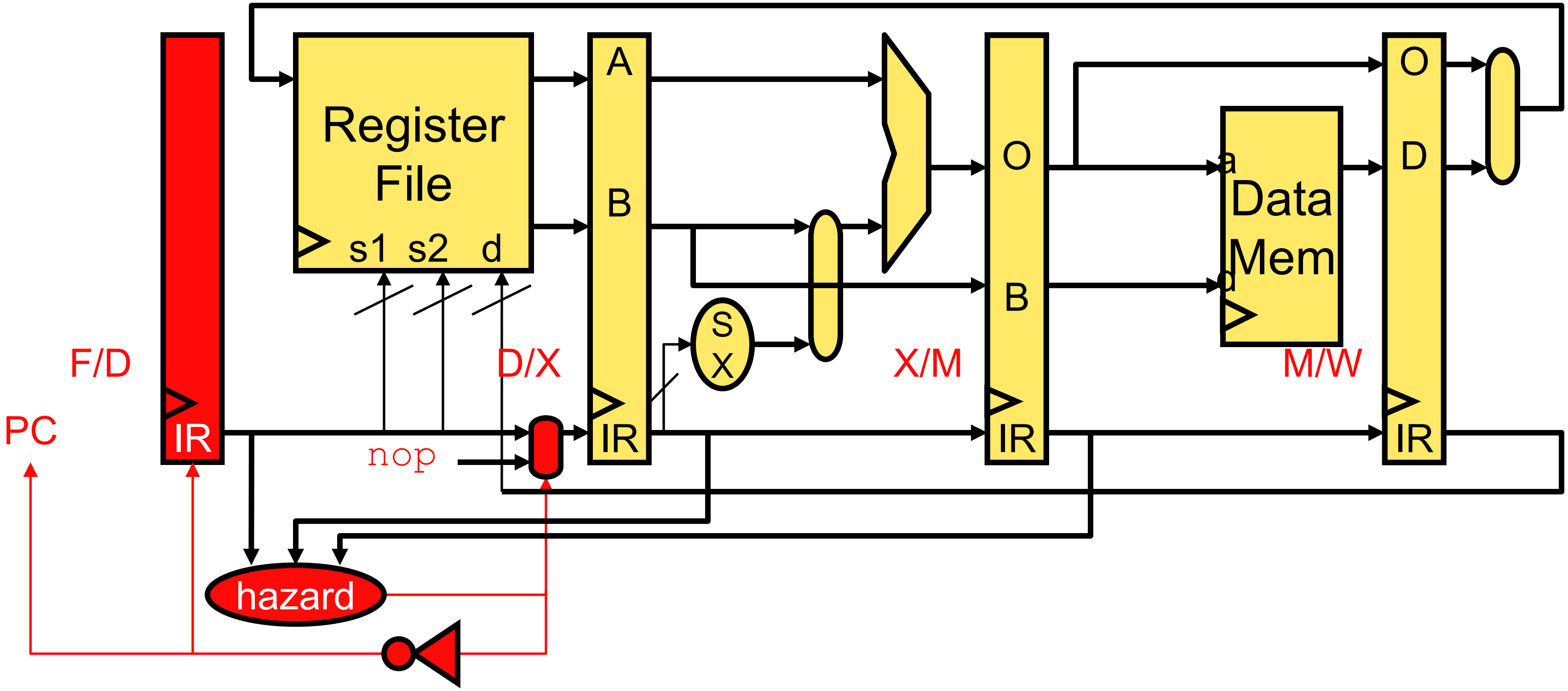
`lw $4,0($3)`

`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**

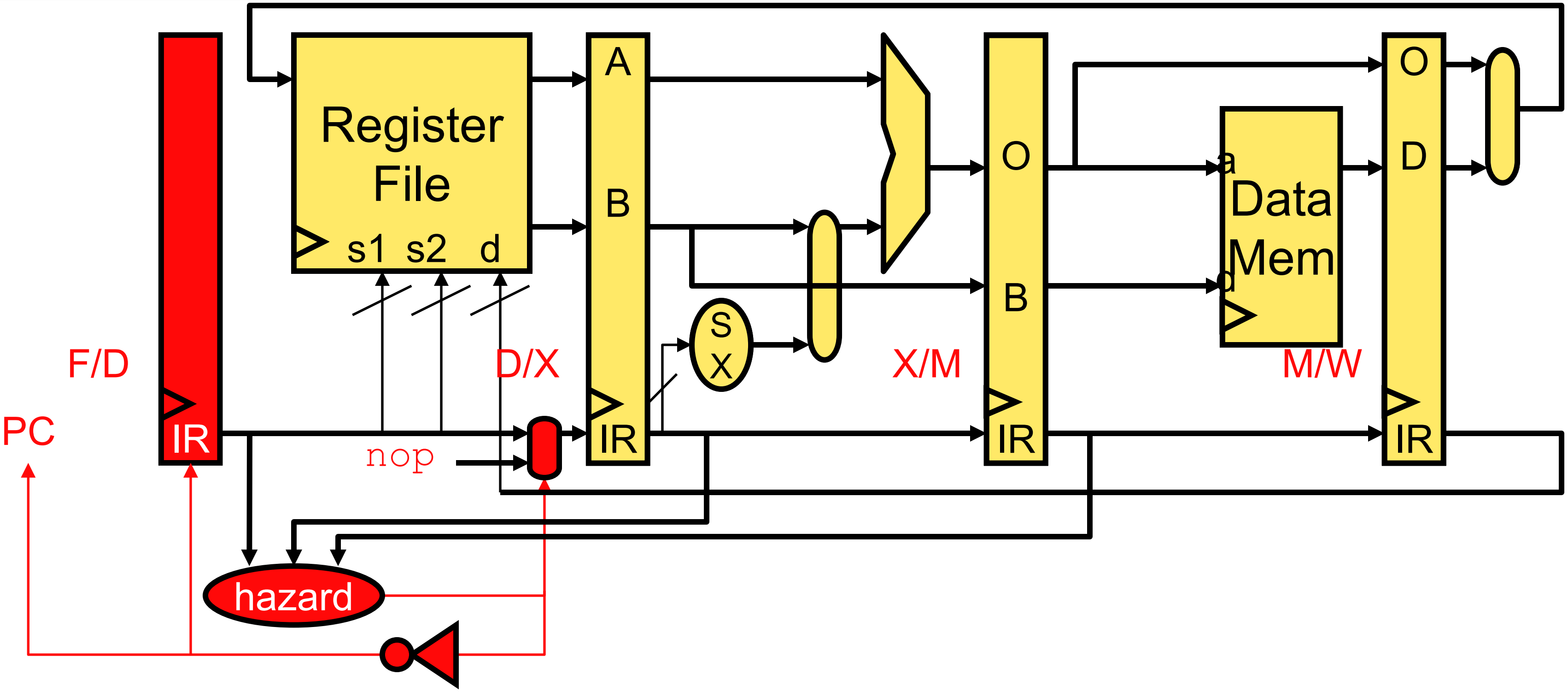
Hardware Interlock Example: cycle 2



`lw $4,0($3)` `nop` `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)` nop nop `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 (inserting a NOP) is called **stall** or **bubble**
- Mechanism is called **stall logic**
- Part of a 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	F	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~~		

Pipeline Diagram with Data Hazards

- Can also 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	D	D	X	M	W	
`sw $6,4($7)`			F	F	F	D	X	M	W

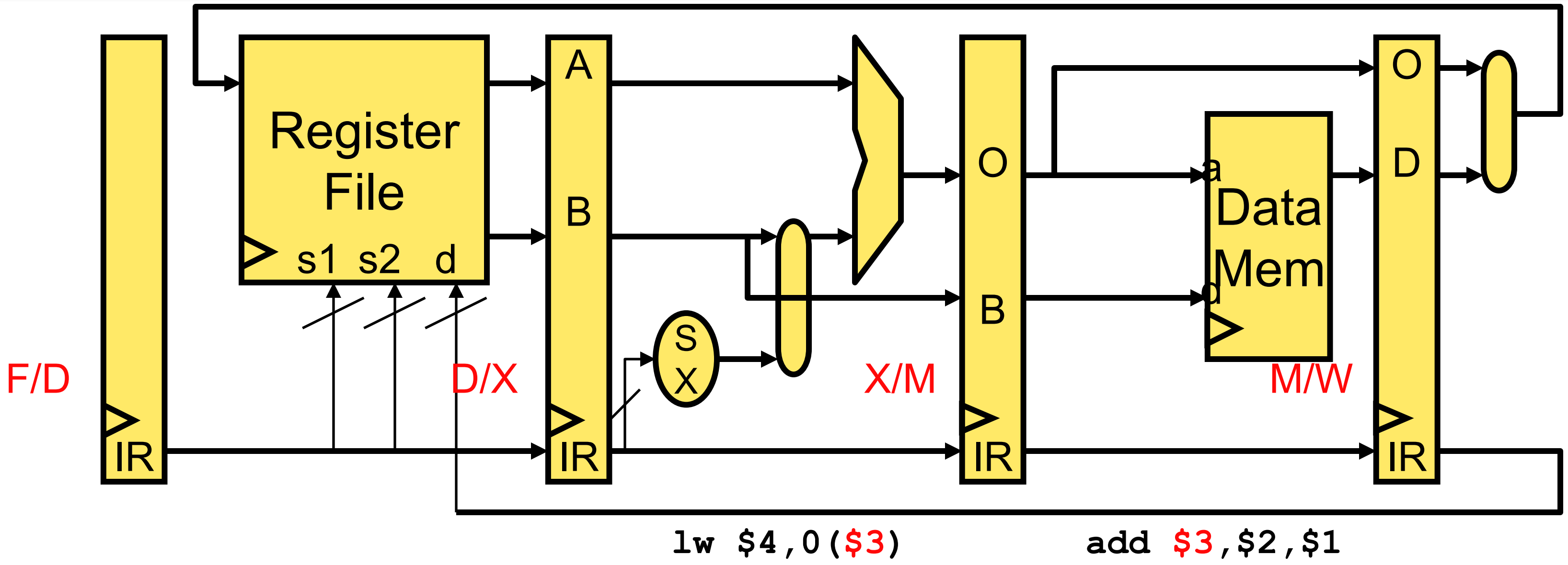
Hardware Interlock Performance

- 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 **`nops`**)

  - 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

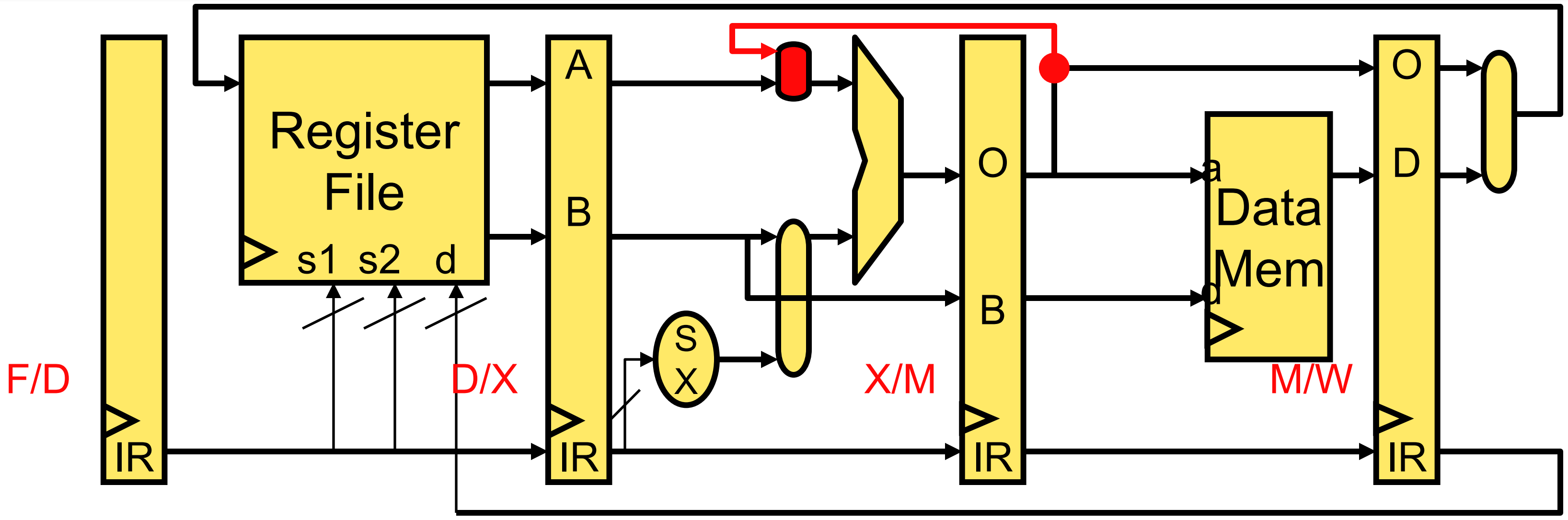
- We can do better!

Observe



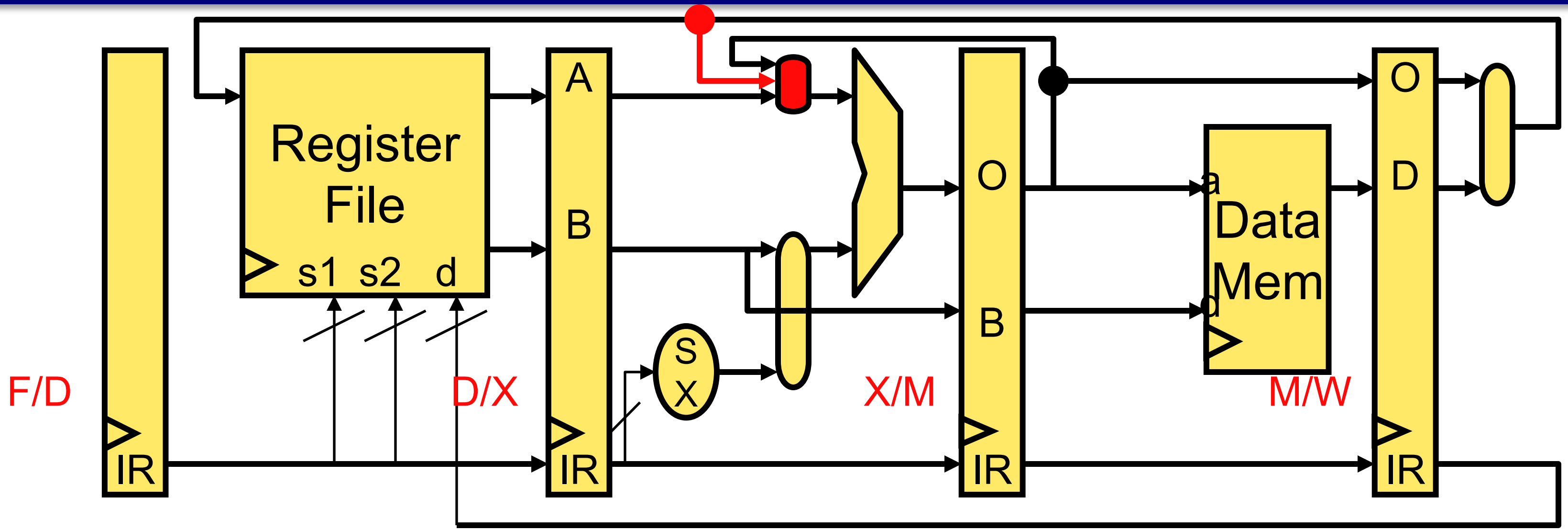
- This situation seems broken
  - `lw $4,0($3)` has already read `$3` from regfile
  - `add $3,$2,$1` hasn't yet written `$3` to regfile
- But fundamentally, everything is still OK
  - `lw $4,0($3)` hasn't actually used `$3` yet (nothing written yet)
  - `add $3,$2,$1` has already computed `$3`

Bypassing



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

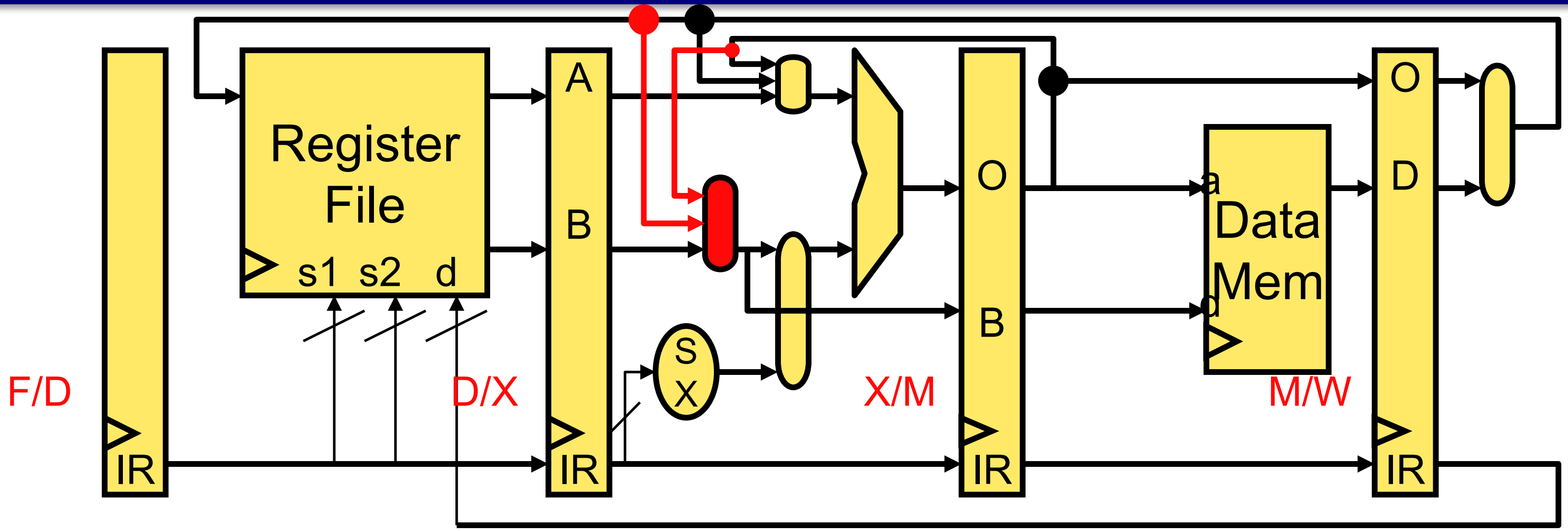
WX Bypassing



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

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

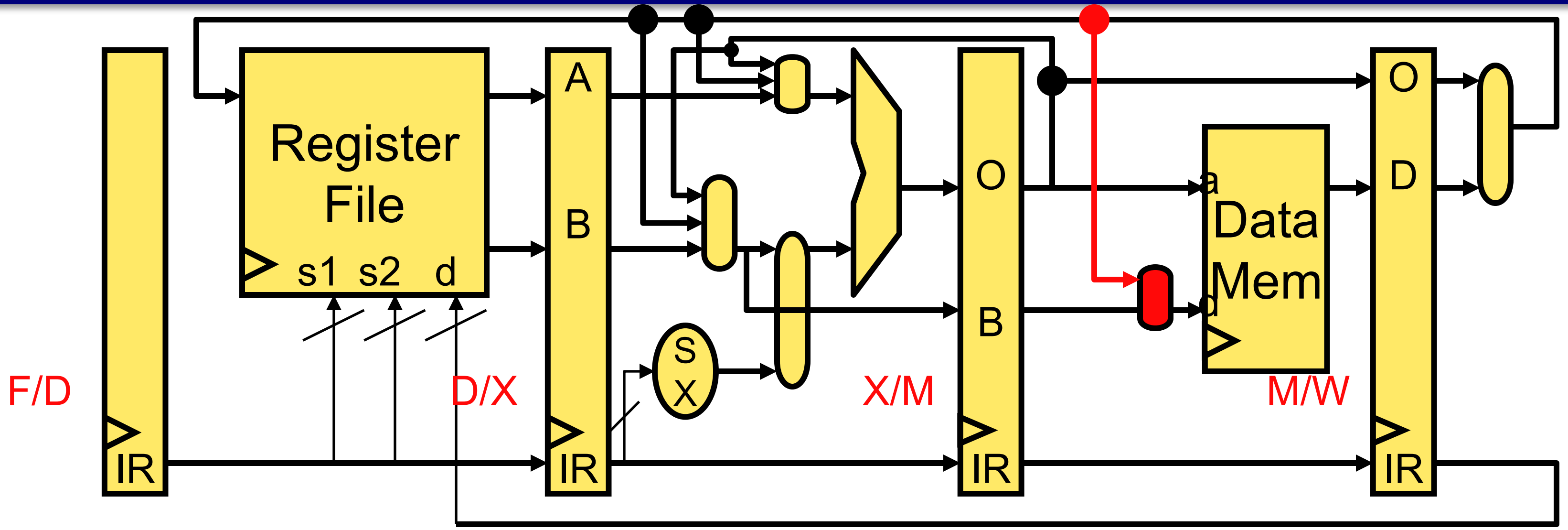
ALUinB Bypassing



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

- Can also bypass to ALU input B

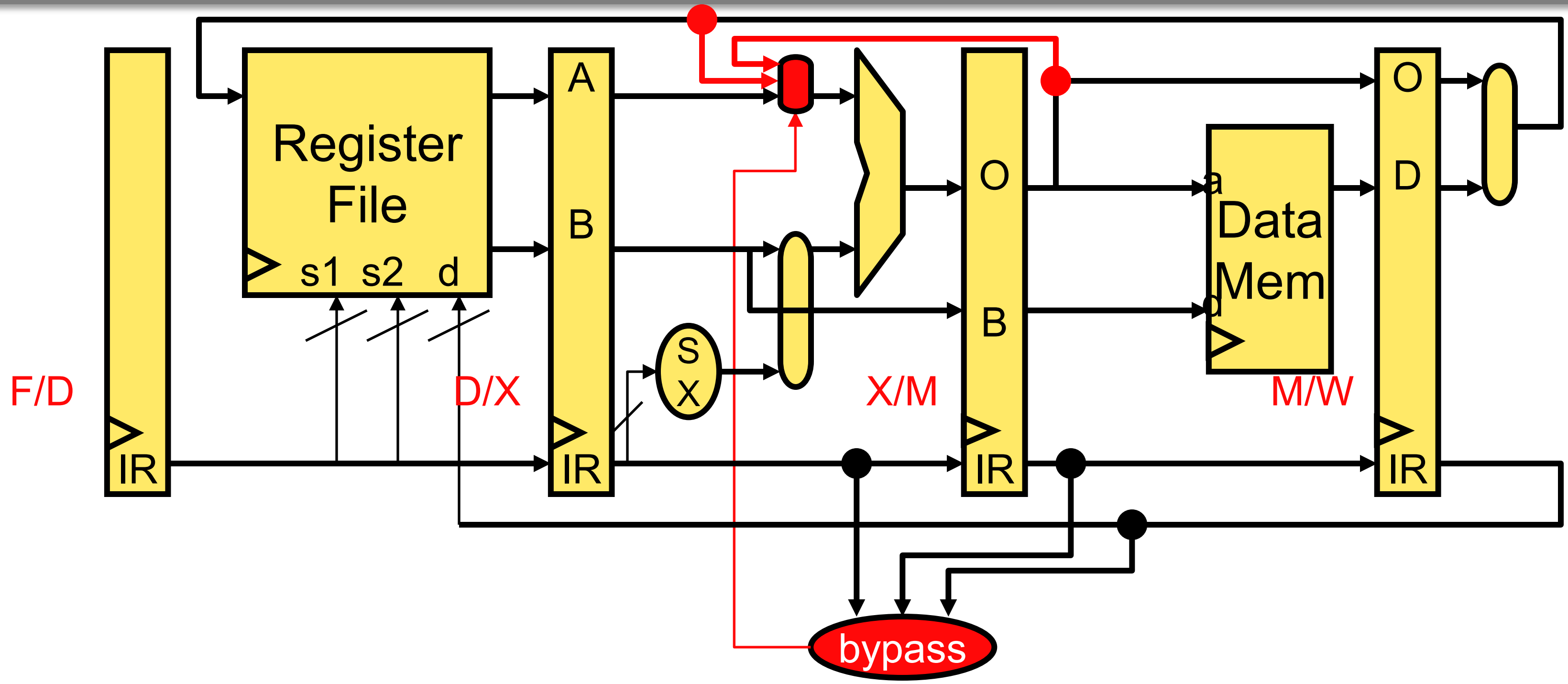
WM Bypassing?



```
sw $3,0($4)        lw $3,0($2)
```

- Does WM bypassing make sense?
  - To the data input?
    - Yes
  - What about to the address input?
    - No. Address is **computed** at X stage (reg value + immediate)

Bypass Logic



- Each mux has its own control
- E.g., for ALUinA mux:

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

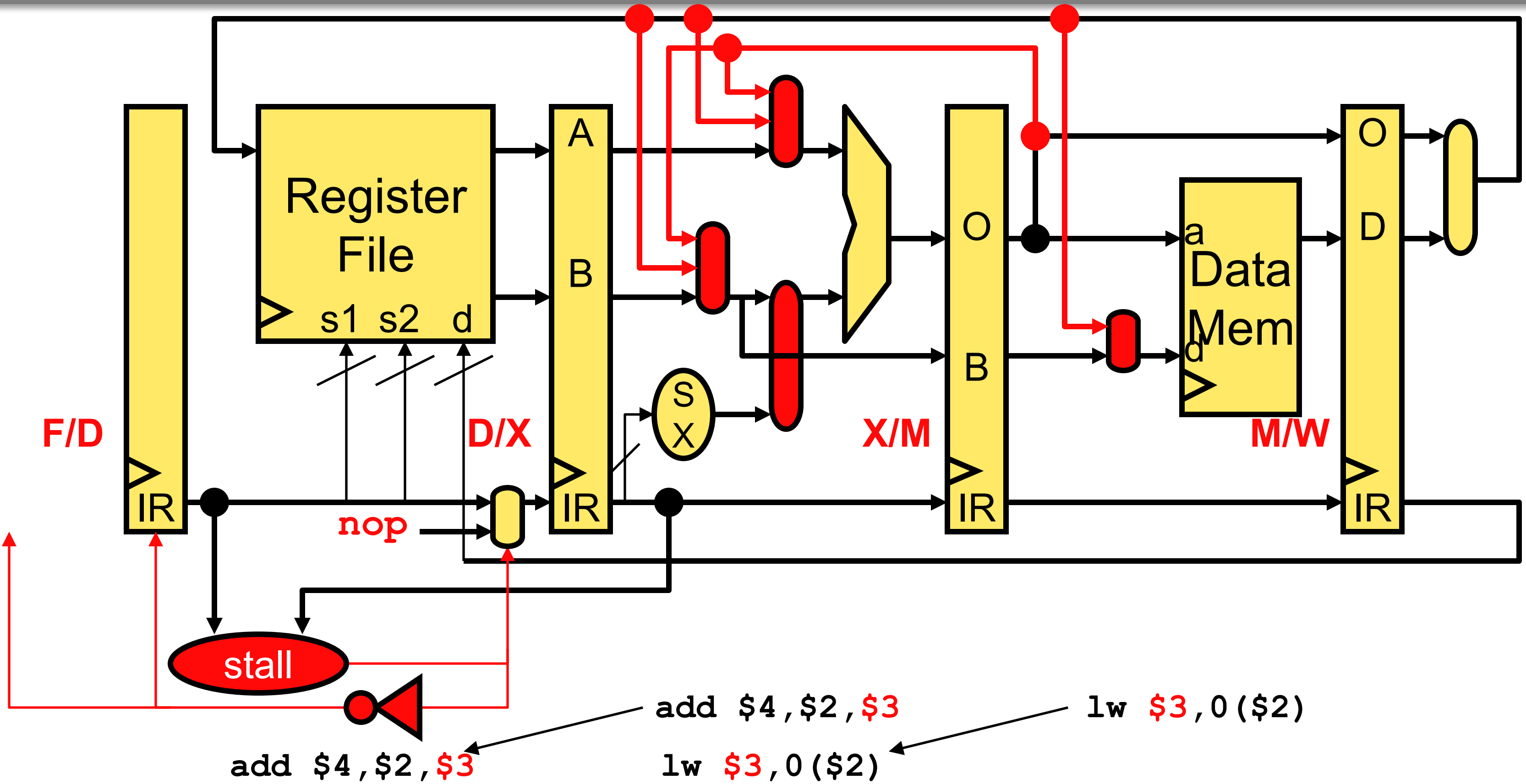
(D/X.IR.RS1 == M/W.IR.RD) → mux select = 1

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 we can't bypass → must stall
- Slide #46 shows **full bypassing**: all possible bypasses
  - Is stall logic still necessary?
    - Yes

Yes, Load Output to ALU Input

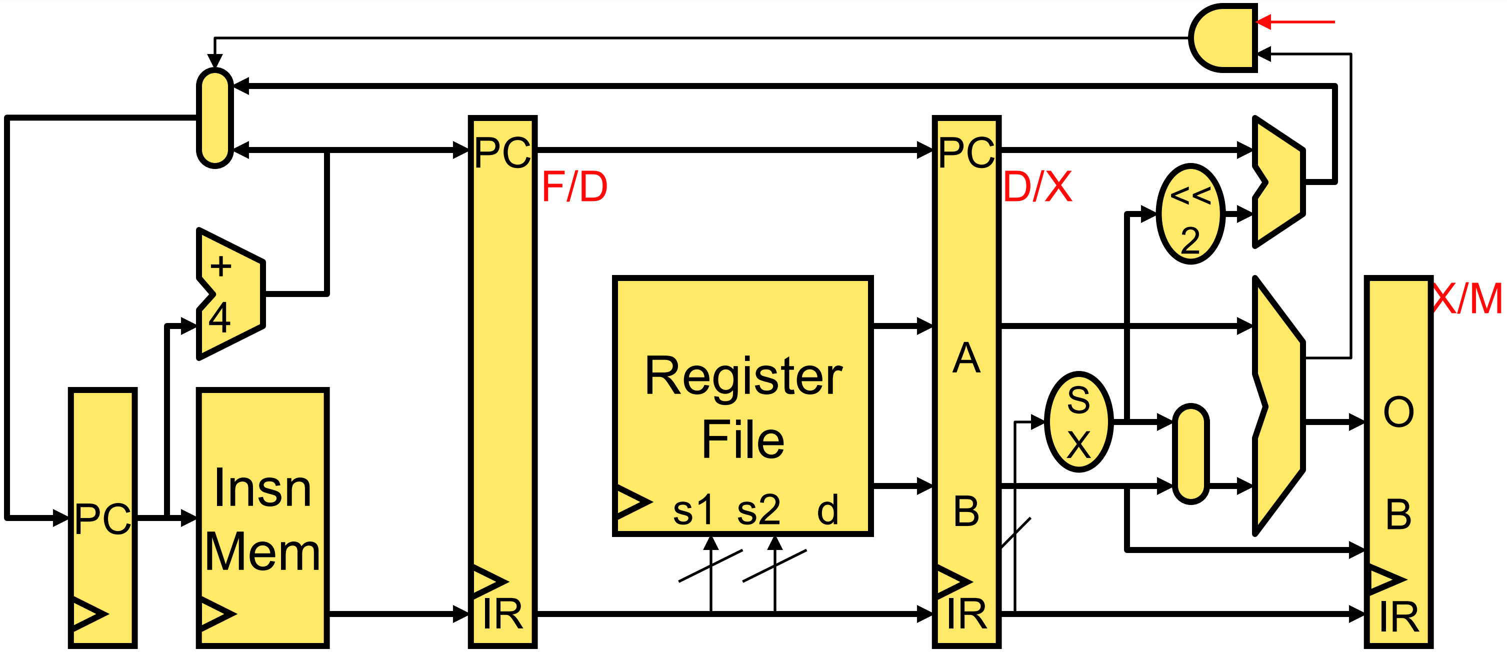


Stall = (D/X.IR.OP==LOAD) && (
(F/D.IR.RS1==D/X.IR.RD) ||
((F/D.IR.RS2==D/X.IR.RD) && (F/D.IR.OP!=STORE))
)

Intuition: "Stall if it's a load where rs1 is a data hazard for the next instruction, or where rs2 is a data hazard in a *non-store* next instruction". This is because rs2 is safe in a store instruction, because it doesn't use the X stage, and can be M/W bypassed.

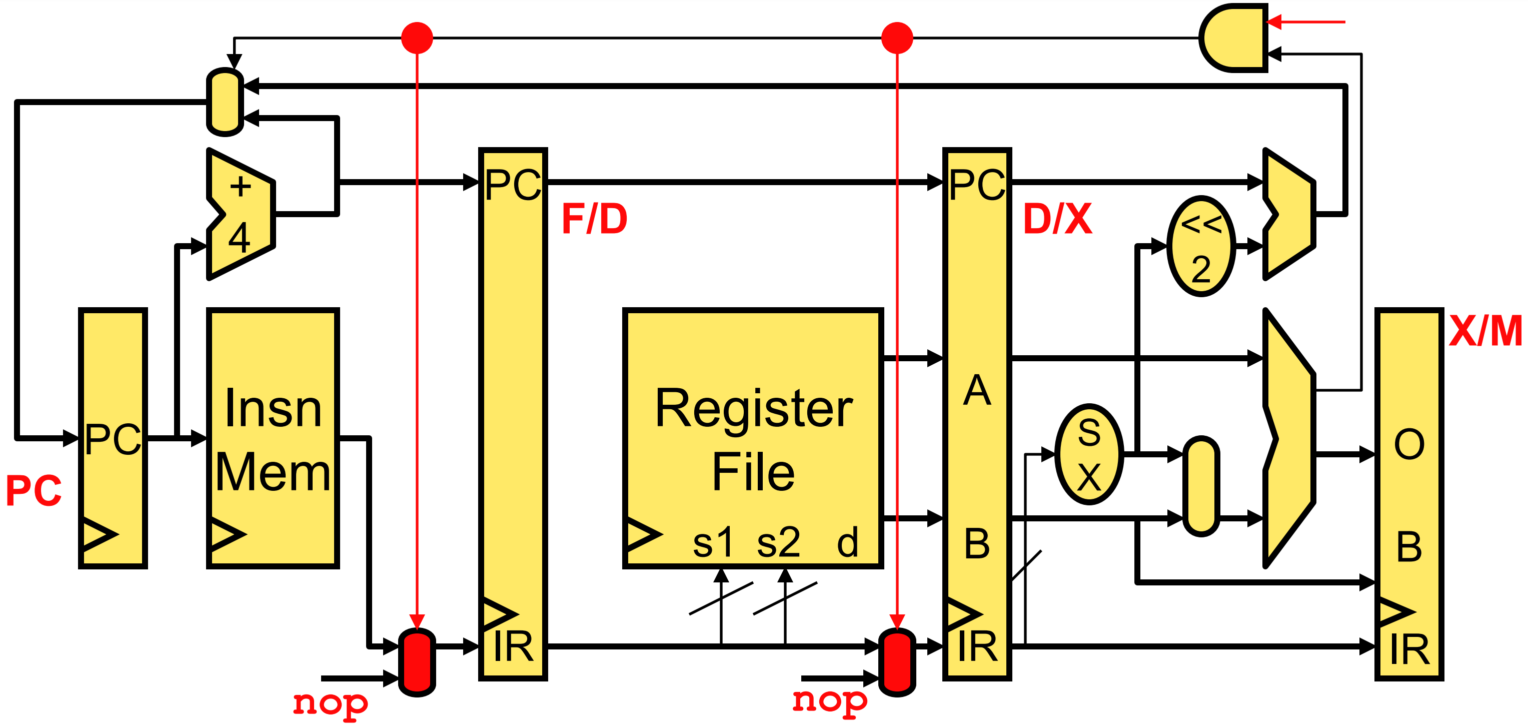
Control Hazards

Control Hazards



- **Control hazards**
  - Must fetch post-branch insns before branch outcome is known
    - Since we don’t know whether or not we’re branching until the insn reaches the **X stage** (ALU)
  - Default: assume “branch **not taken**” (at fetch, can’t tell 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 RegFile or DMem

Branch Recovery Pipeline Diagram

	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)`~~			~~F~~	~~D~~					
~~`addi $8,$7,1`~~				~~F~~					
`targ:` `sw $6,4($7)`					**F**	D	X	M	W

- Control hazards sometimes indicated with **c***
  - 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

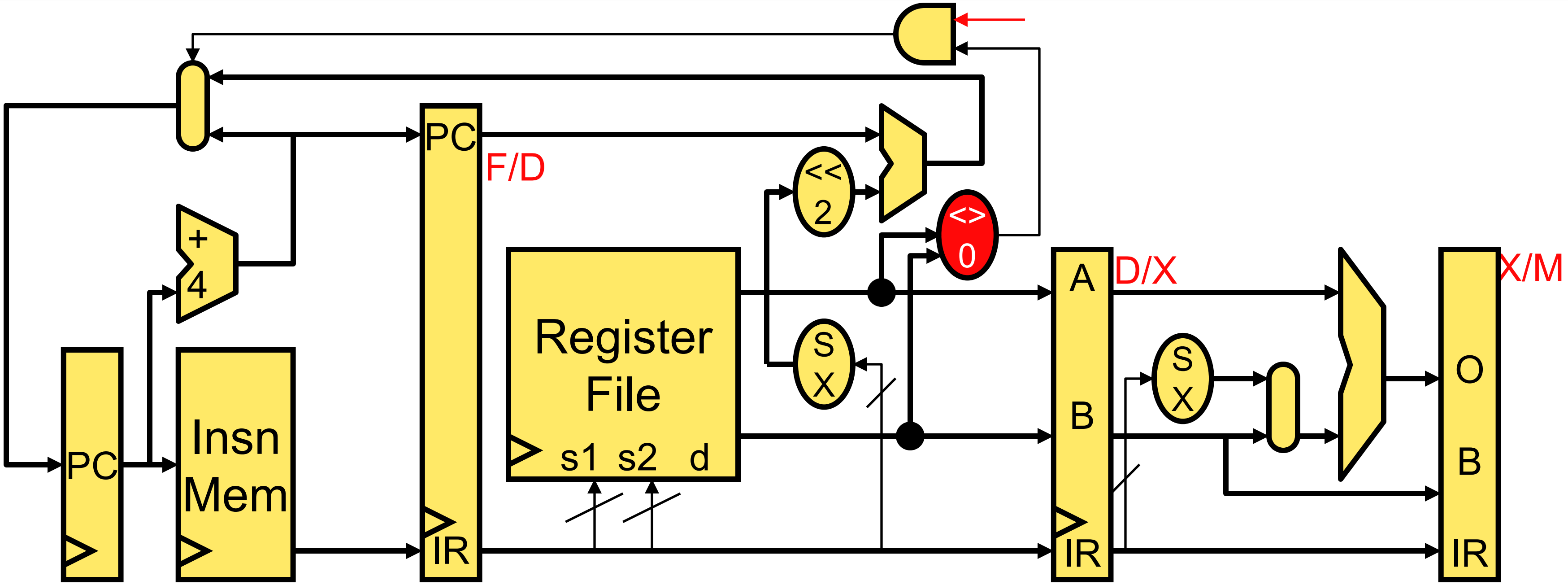
- Again, measuring effect on CPI (clock period is fixed)

- Approximate calculation
  - **Branch: 20%**, load: 20%, store: 10%, other: 50%
  - **75% of branches are taken**
    - **How come?**

- 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?

Can We Perform Better w.r.t. Control Hazards?

Fast Branch



- **Fast branch**: can decide at D instead of X
  - Duplicate comparison logic only, not the whole ALU
  - + New taken branch penalty is now **1** stall instead of 2
  - – Additional insns (`slt`) for more complex tests, must bypass to D too
  - 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

Speculative Execution

- Speculation: "risky transactions on chance of profit"

- **Speculative execution**
  - Execute before all parameters known with certainty
  - **Correct speculation**
    - + Avoid stall, improve performance
  - **Incorrect speculation (mis-speculation)**
    - – Must abort/flush/squash incorrect insns
    - – Must undo incorrect changes (recover pre-speculation state)
  - The "game": **[$\%_{correct}$ * gain] – [(1–$\%_{correct}$) * penalty]**

- **Control speculation**: speculation aimed at control hazards
  - Unknown parameter: are these the correct insns to execute next?

Control Speculation Mechanics

- Guess branch target, start fetching at guessed position
  - Doing nothing is implicitly guessing target is PC+4
  - Can actively guess other targets: **dynamic branch prediction**

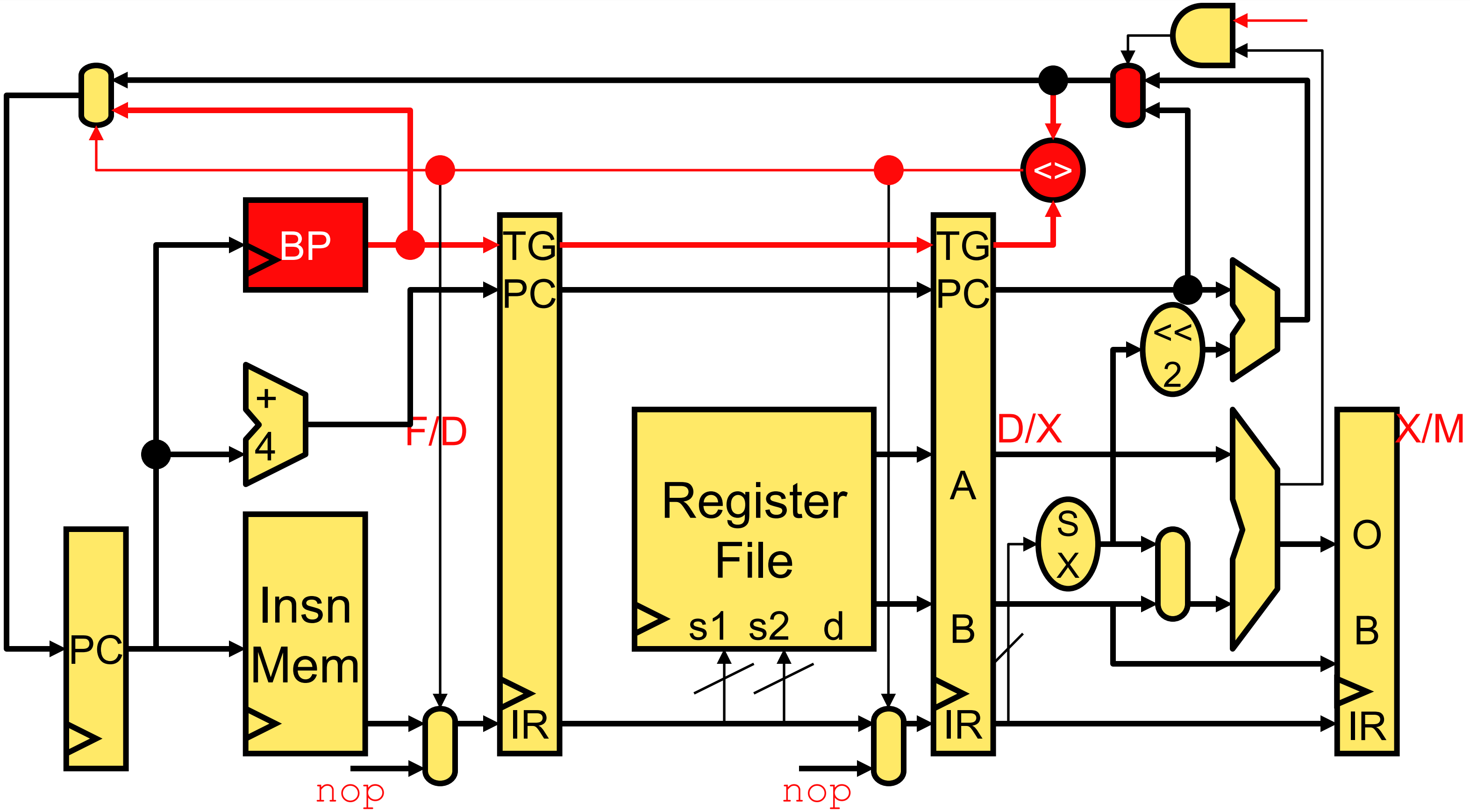
- Execute branch to verify (check) guess
  - Correct speculation? keep going
  - Mis-speculation? Flush mis-speculated insns
    - Hopefully haven't modified permanent state (Regfile, DMem)

    + Happens naturally in in-order 5-stage pipeline

- "Game" for in-order 5 stage pipeline
  - $\%_{\text{correct}}$ = ?
  - Gain = 2 cycles

  + Penalty = 0 cycles → **mis-speculation no worse than stalling**

Dynamic Branch Prediction



- **Dynamic branch prediction**: guess outcome
  - Start fetching from guessed address
  - Flush on **mis-prediction** (notice new recovery circuit)

Branch Prediction: Short Summary

- Key principle of micro-architecture:
  - Programs do the same thing over and over (why?)
  - Exploit for performance:
    - Learn what a program did before
    - Guess that it will do the same thing again
- Inside a branch predictor: the short version
  - Use some of the PC bits as an **index** to a separate RAM
  - This RAM contains (a) branch destination and (b) whether we predict the branch will be taken
  - RAM is updated with results of past executions of branches
  - Algorithm for predictions can be simple (“assume it’s same as last time”), or get quite fancy

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 (so 25% are penalized)
  - 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

- → take ECE 552

Pipeline Depth

- No magic about 5 stages, trend had been to deeper pipelines
  - 486: 5 stages (50+ gate delays / clock)
  - Pentium: 7 stages
  - Pentium II/III: 12 stages
  - Pentium 4: 22 stages (~10 gate delays / clock) **"super-pipelining"**
  - Core1/2: 14 stages

- Increasing **pipeline depth**
  + Increases clock frequency (reduces period)
  – But decreases IPC (increases CPI)
  - Branch mis-prediction penalty becomes longer
  - Non-bypassed data hazard stalls become longer
  - At some point, CPI losses offset clock gains, question is when?
    - 1GHz Pentium 4 was slower than 800 MHz PentiumIII
    - What was the point?
      - People buy frequency, not frequency*IPC (throughput)

Real pipelines…

- Real pipelines fancier than what we have seen
  - Superscalar: multiple instructions in a stage at once
  - Out-of-order: re-order instructions to reduce stalls
  - SMT: execute multiple threads at once on processor
    - Side by side, sharing pipeline resources
  - Multi-core: multiple pipelines on chip
    - Cache coherence: No stale data