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)

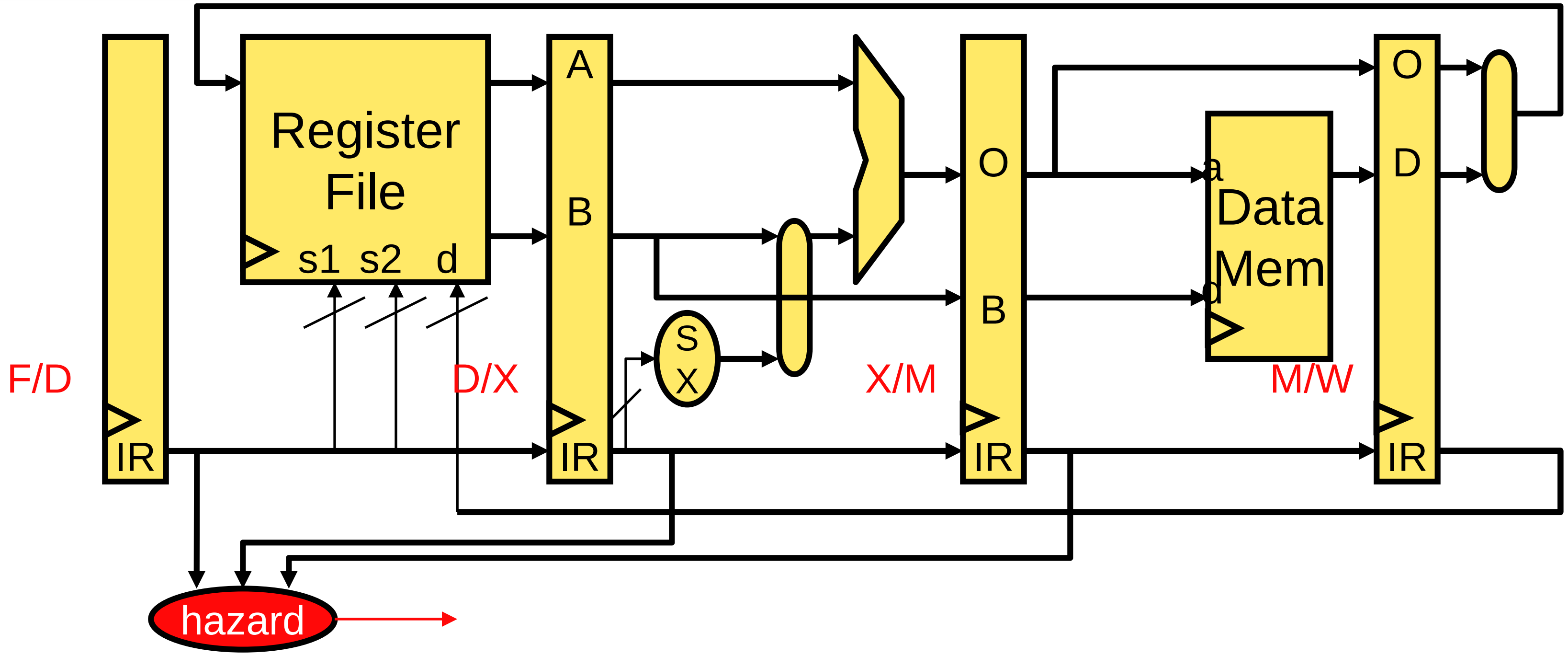
Last Time

- What did we do last time?
- Pipelining
  - Pipelined datapath
  - Pipeline control
  - Dependences and hazards

Hardware Interlocks

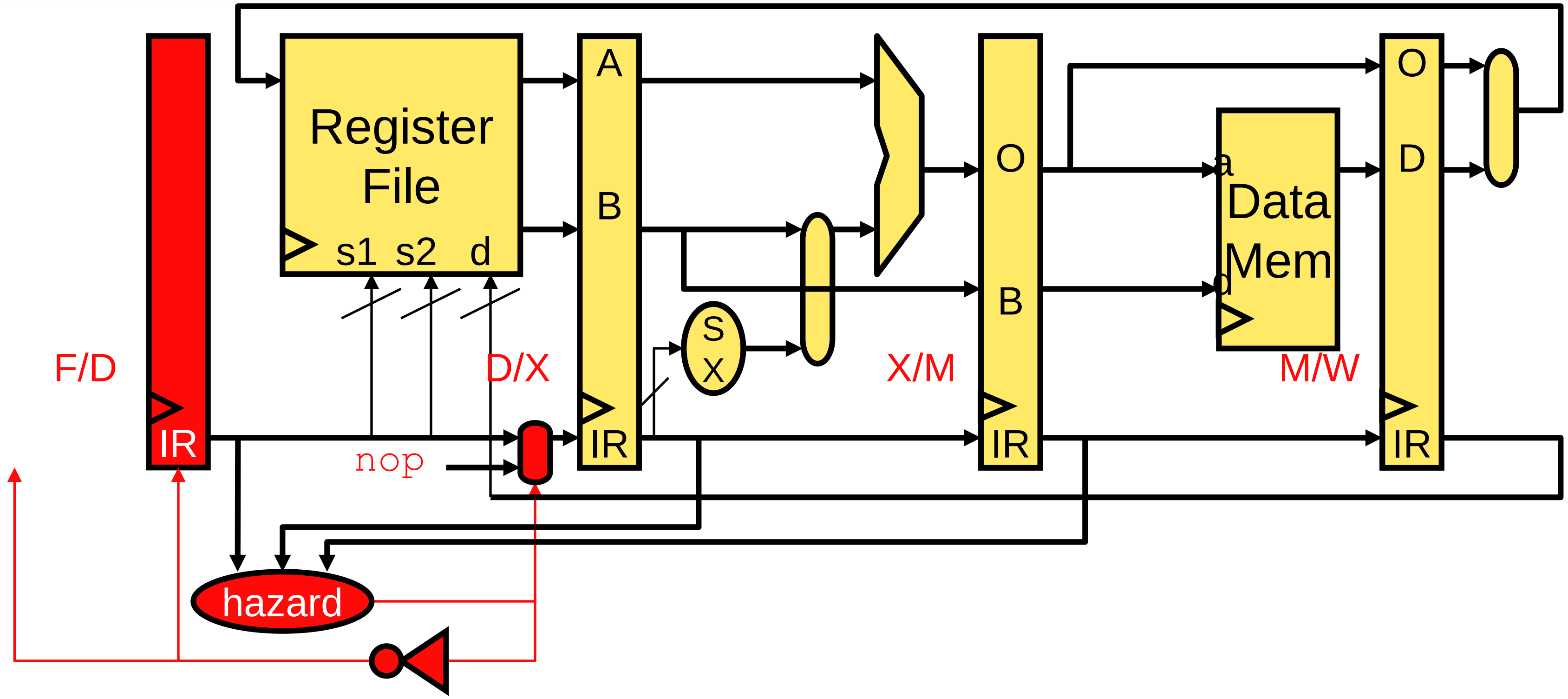
- A better (more compatible) way: **hardware interlocks**
  - Processor detects data hazards and fixes them
  - Two aspects to this
    - Detecting hazards
    - Fixing hazards

Detecting Data Hazards



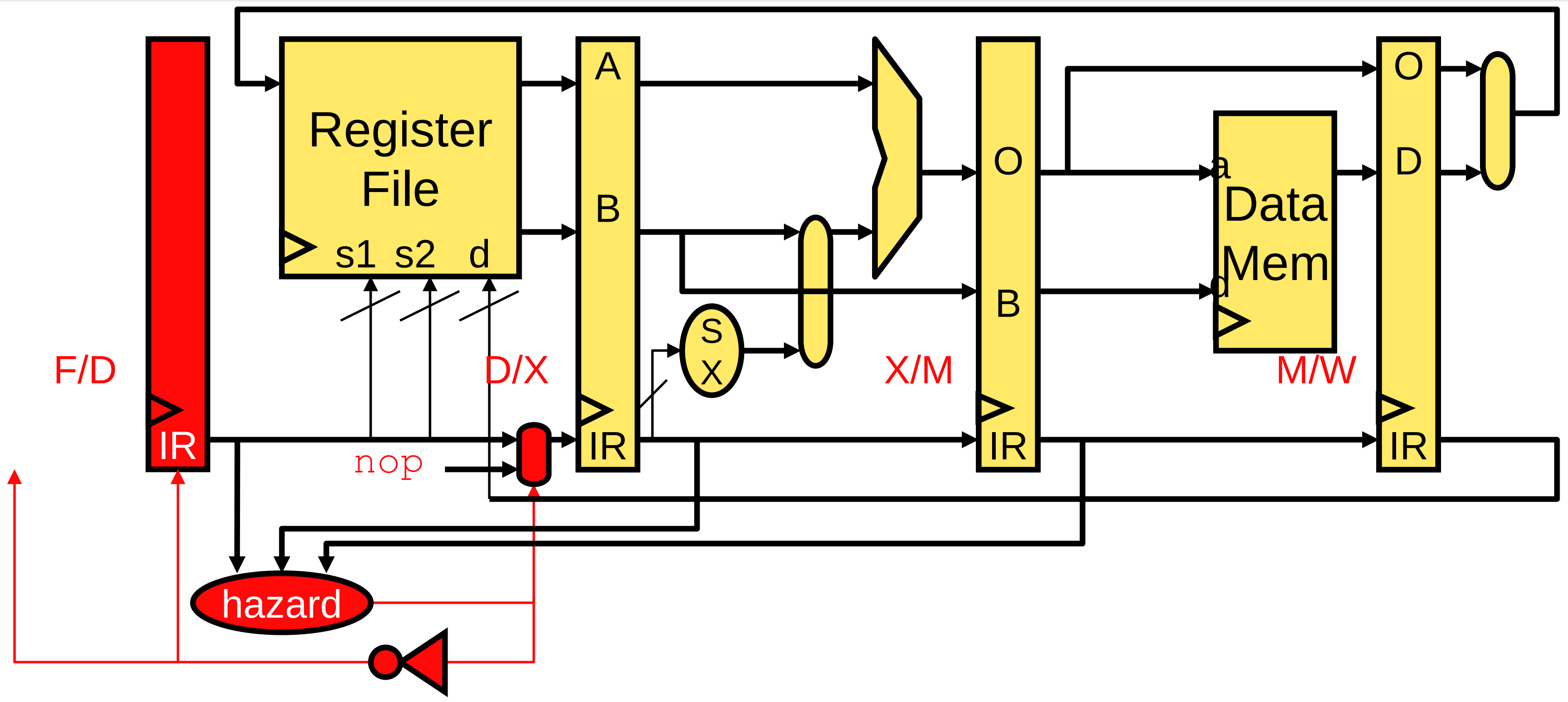
- Compare F/D insn input register names with output register names of older insns in pipeline
  - Hazard =
    - (F/D.IR.RS1 == D/X.IR.RD) || (F/D.IR.RS2 == D/X.IR.RD) ||
    - (F/D.IR.RS1 == X/M.IR.RD) || (F/D.IR.RS2 == X/M.IR.RD)

Fixing Data Hazards



- Prevent F/D insn from reading (advancing) this cycle
  - Write **nop** into D/X.IR (effectively, insert **nop** in hardware)
  - Also 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

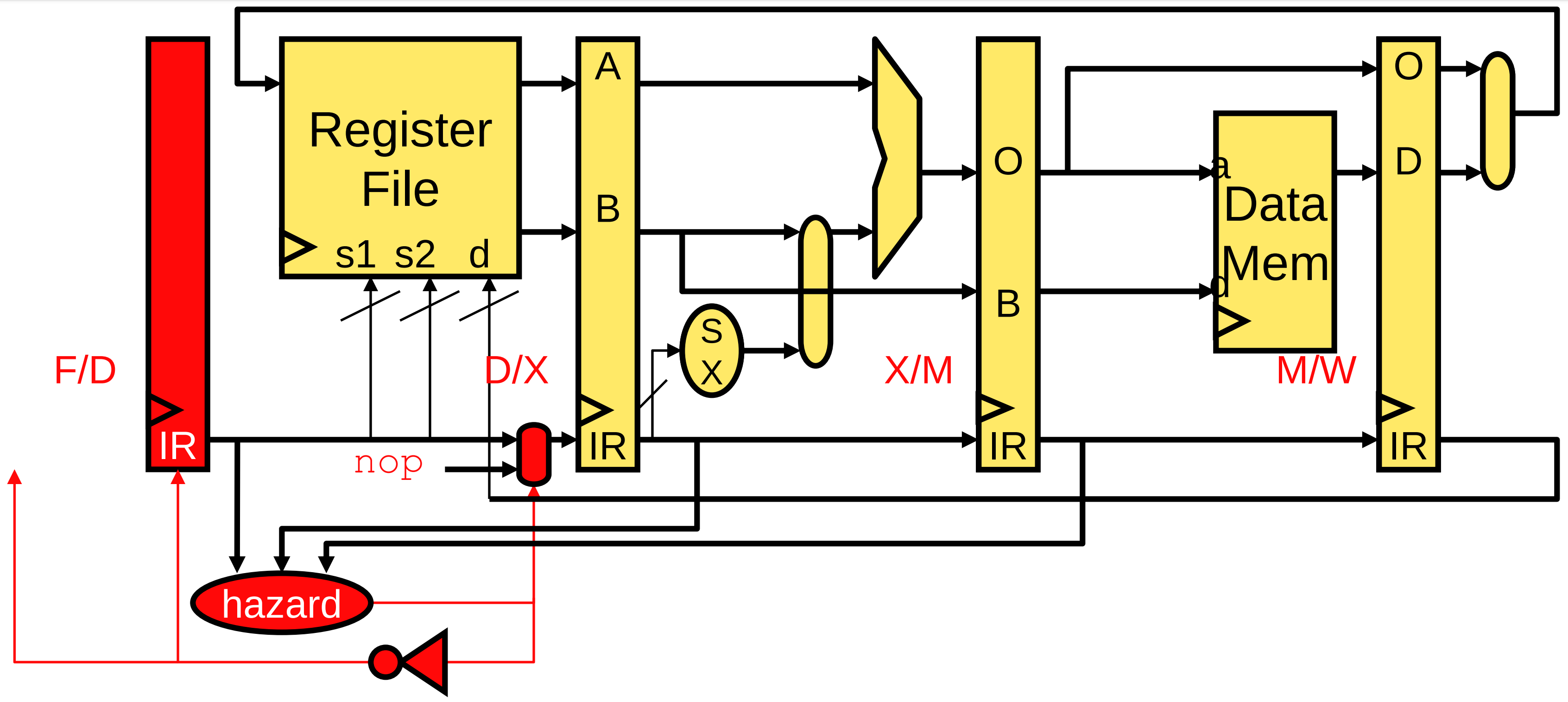


`lw $4,0($3)` `add $3,$2,$1`

(**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

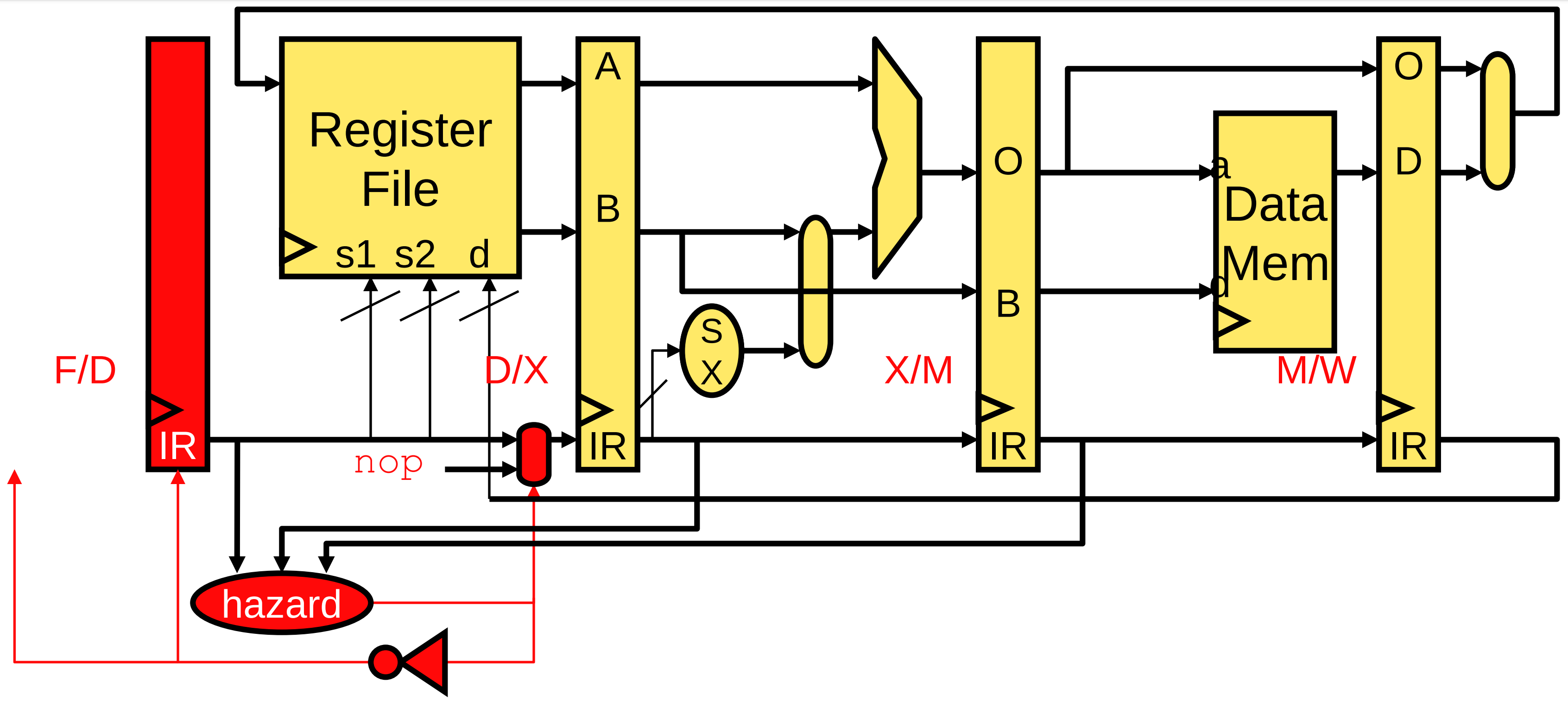


`lw $4,0($3)`   `add $3,$2,$1`

(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

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~~		

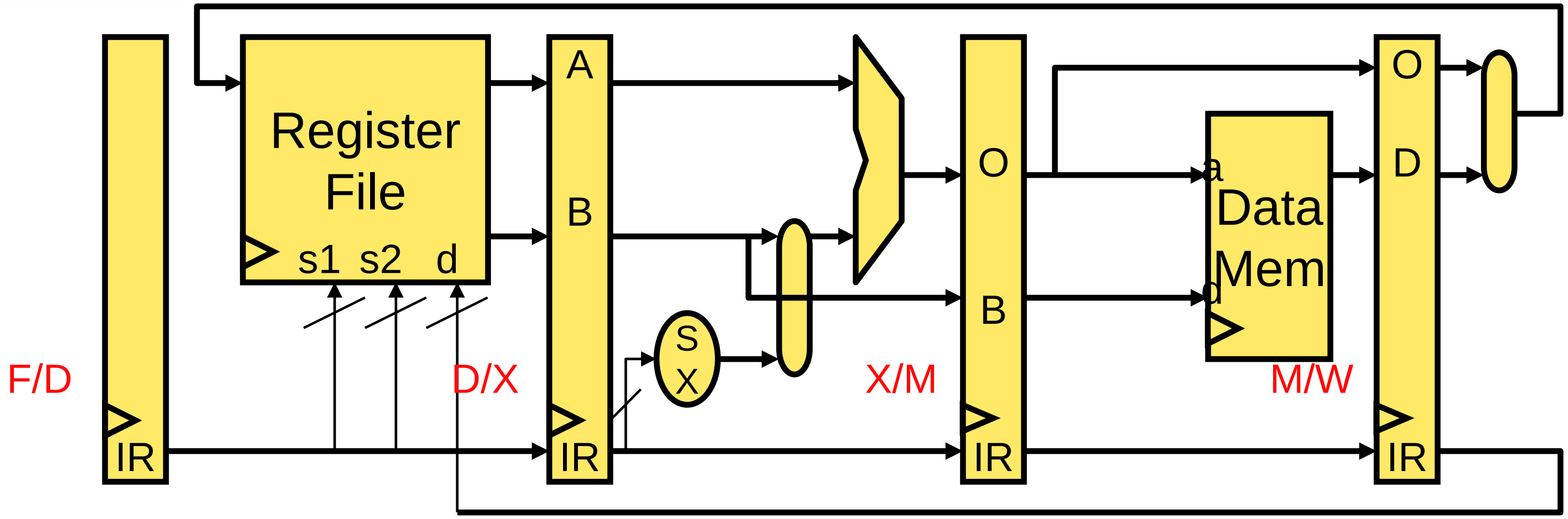
Hardware Interlock Performance

- 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 **`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

- Anyway, we can do better

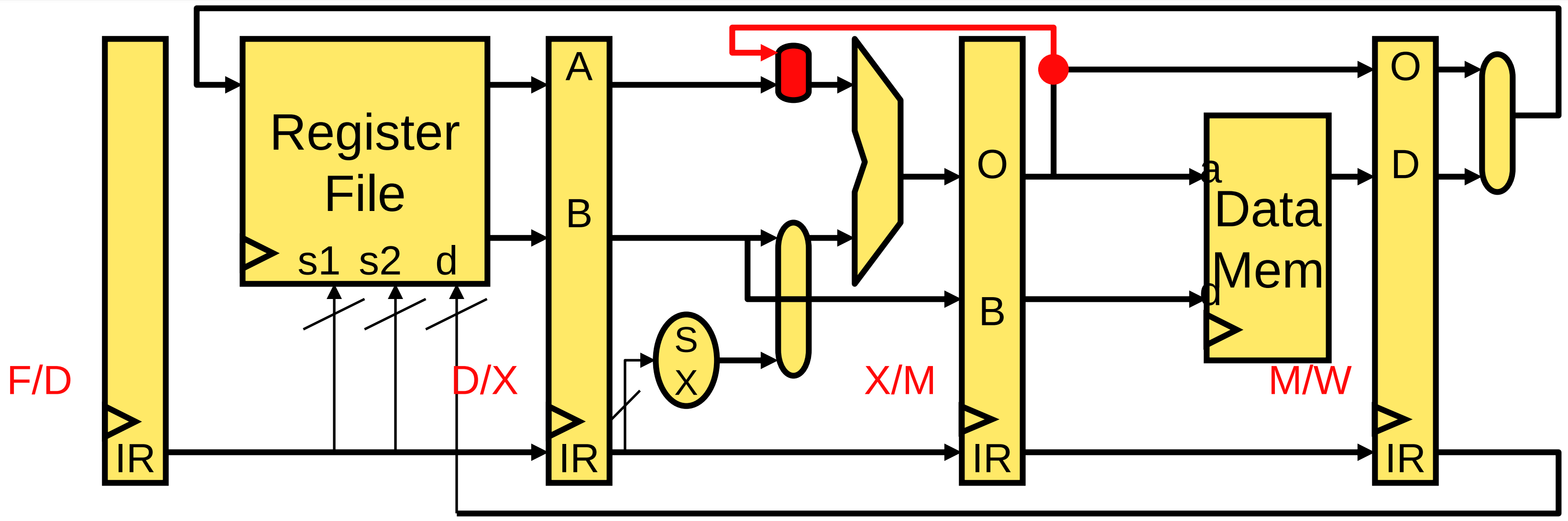
Observe



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

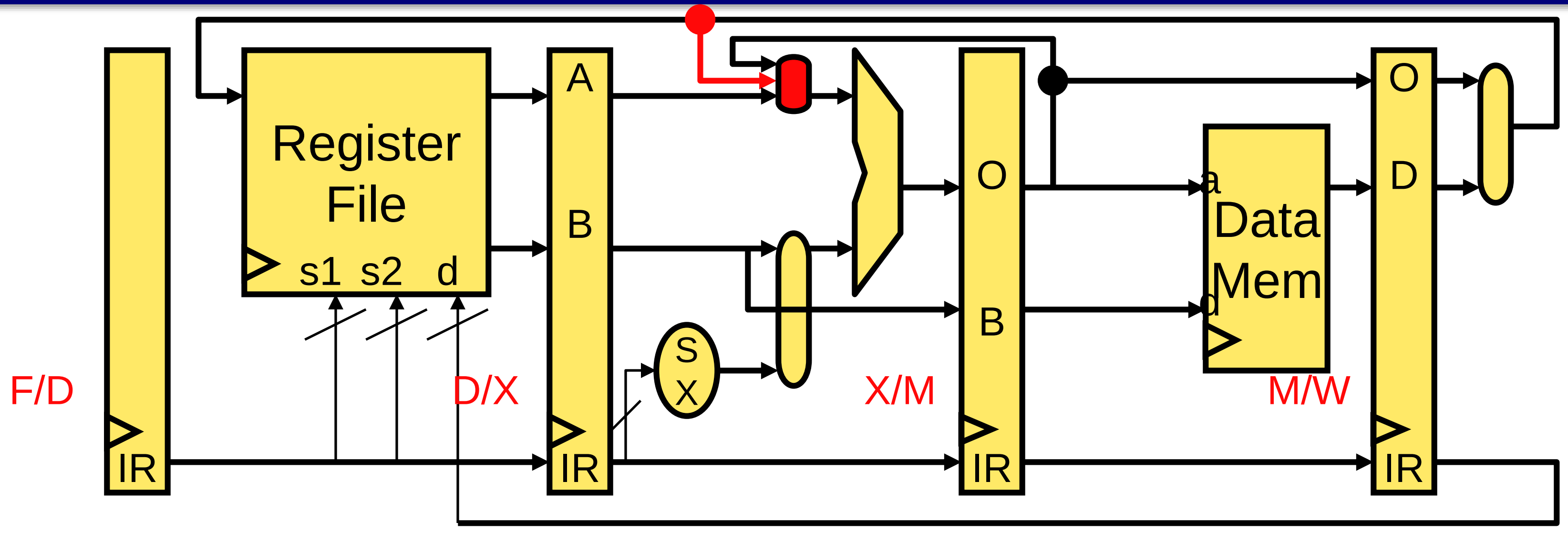
- This situation seems broken
  - `lw $4,0($3)` has already read `$3` from regfile
  - `add $3,$2,$1` hasn't yet written `$3` to regfile
- But fundamentally, everything is still OK
  - `lw $4,0($3)` hasn't actually **used** `$3` yet
  - `add $3,$2,$1` has already **computed** `$3`

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**

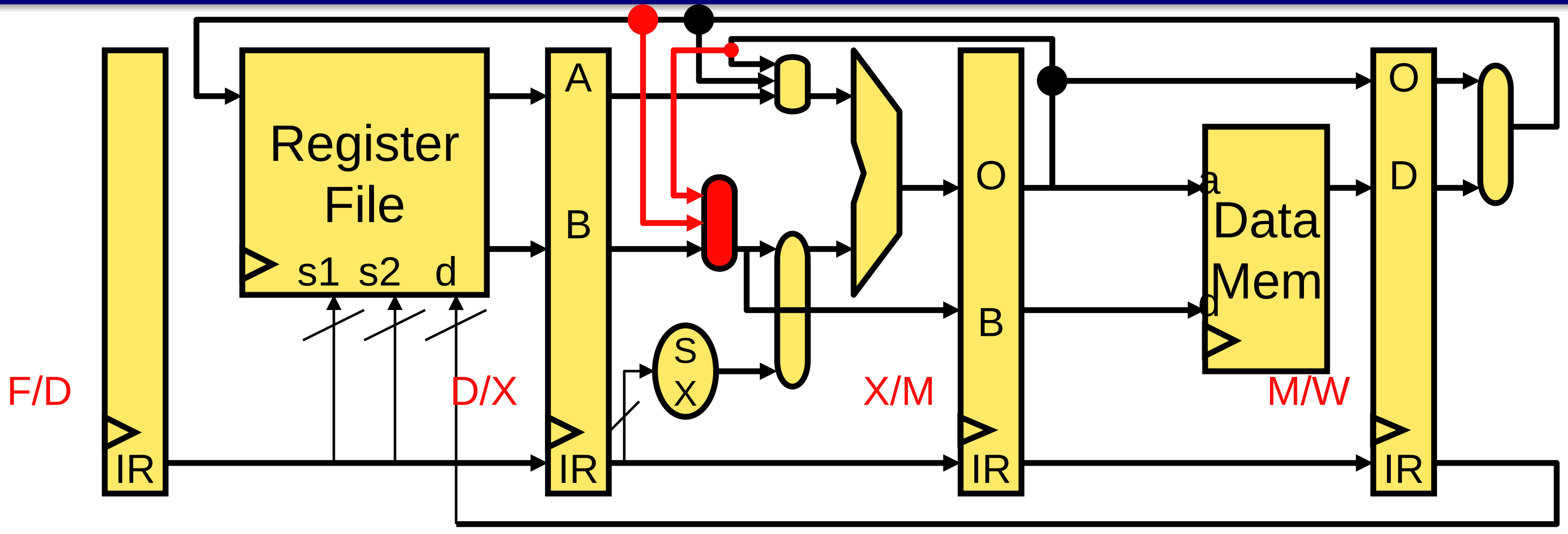
WX Bypassing



`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

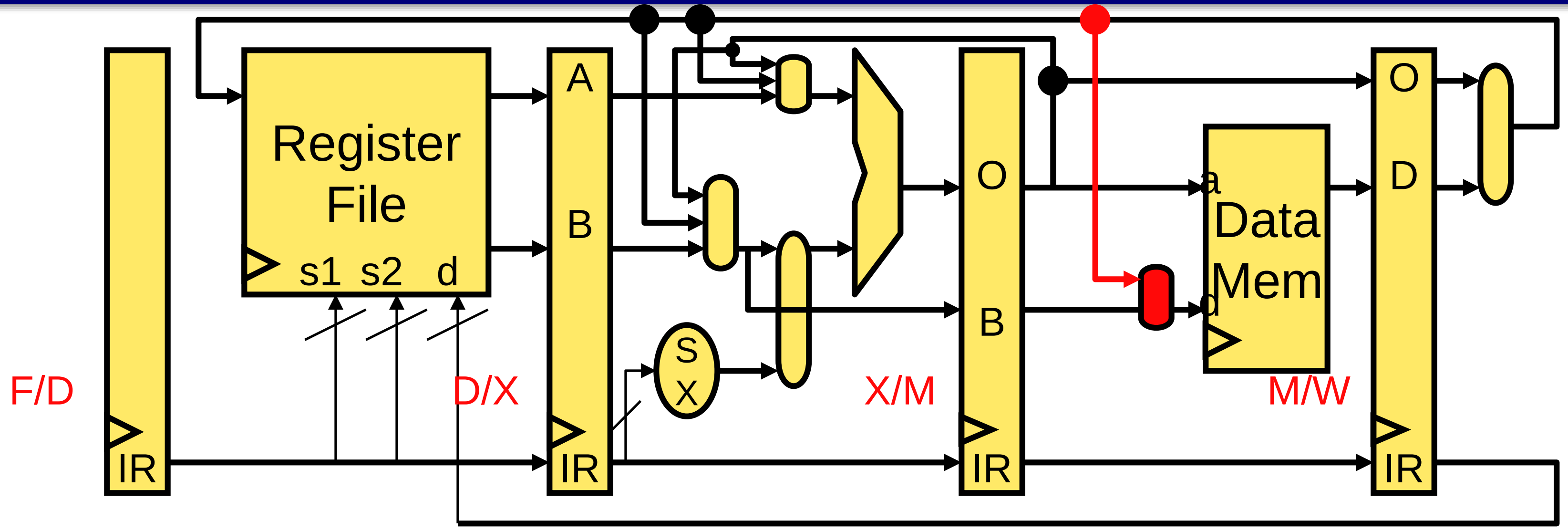
ALUinB Bypassing



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

- Can also bypass to ALU input B

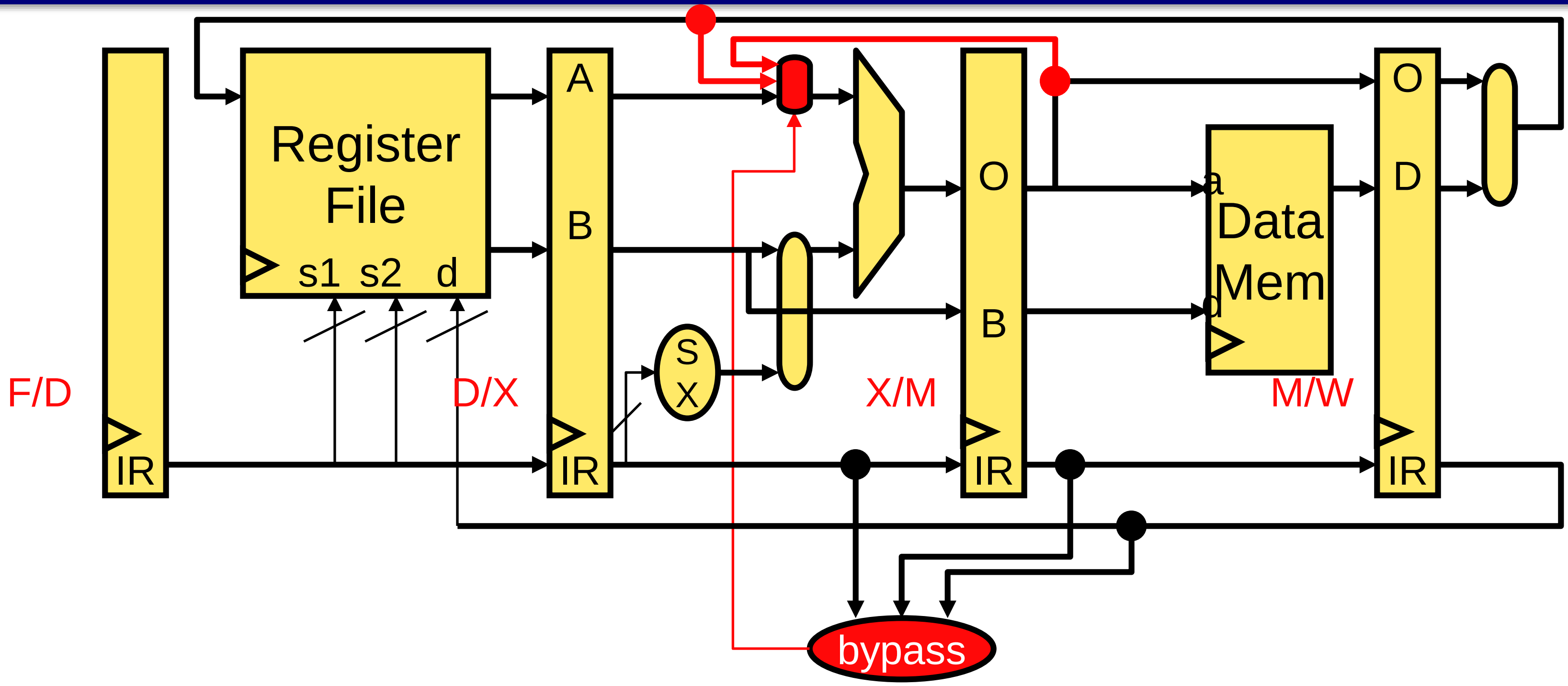
WM Bypassing?



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

- Does WM bypassing make sense?
  - Yes, for the store data input

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

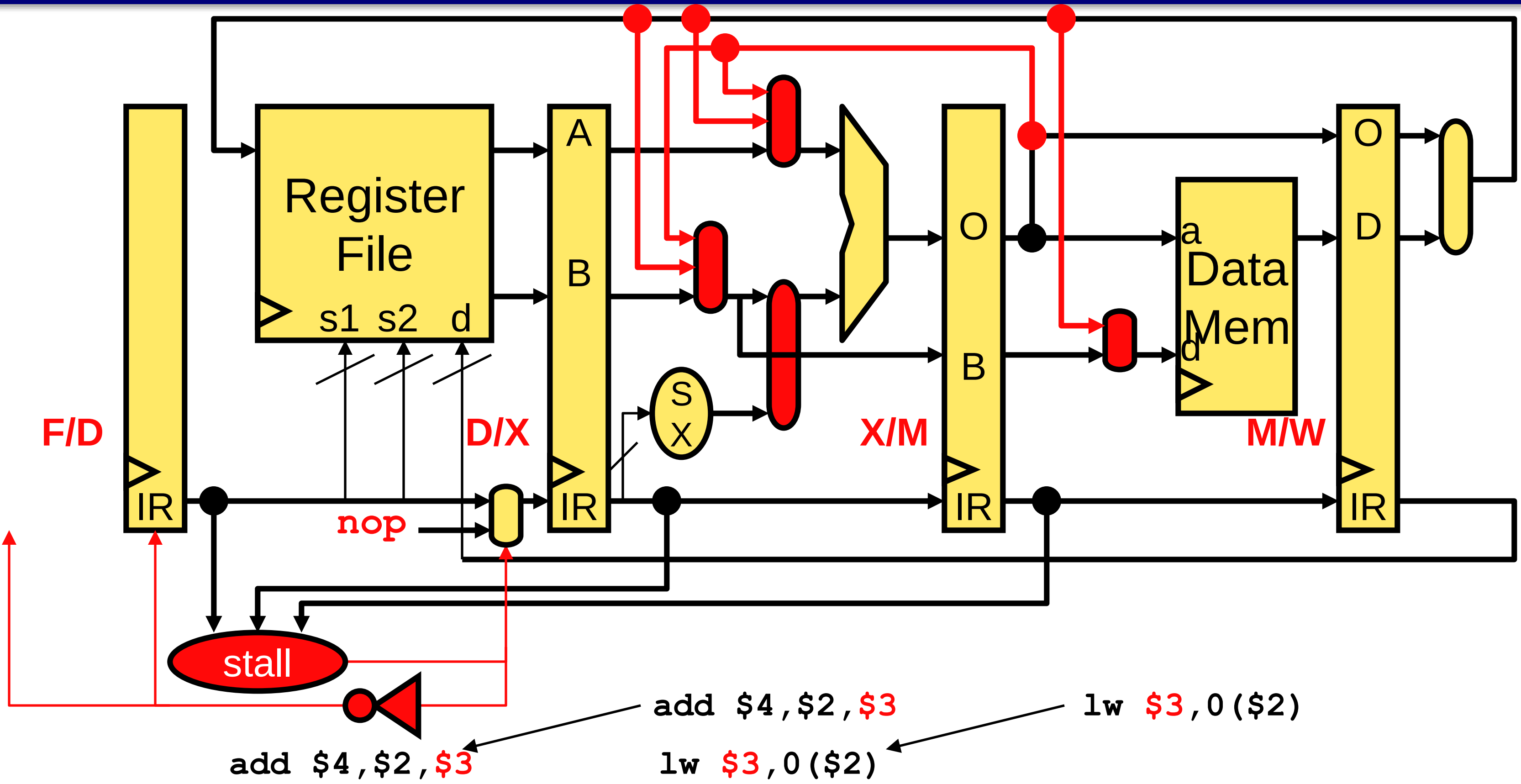
  Else → mux select = 2

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

- **Full bypassing**: all bypasses possible
  - 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))
)

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))
        )
```

```
     F/D              D/X
add $4,$3,$2     lw $3,0($2)
```

```
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))
        )
```

```
     F/D              D/X
add $4,$2,$3     lw $3,0($2)
```

```
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))
        )
```

```
     F/D              D/X
sw $3,0($4)      lw $3,0($2)
```

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 WM bypassed.

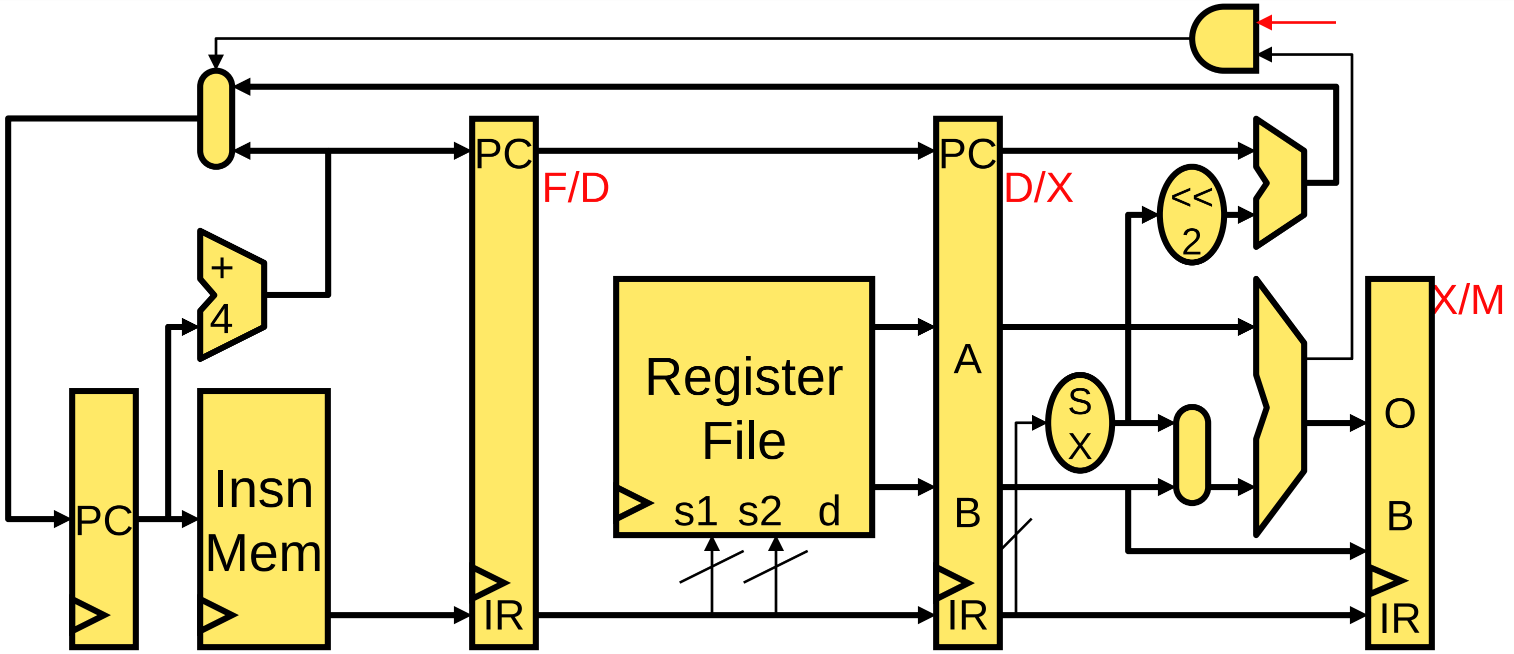
Pipeline Diagram With Bypassing

	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

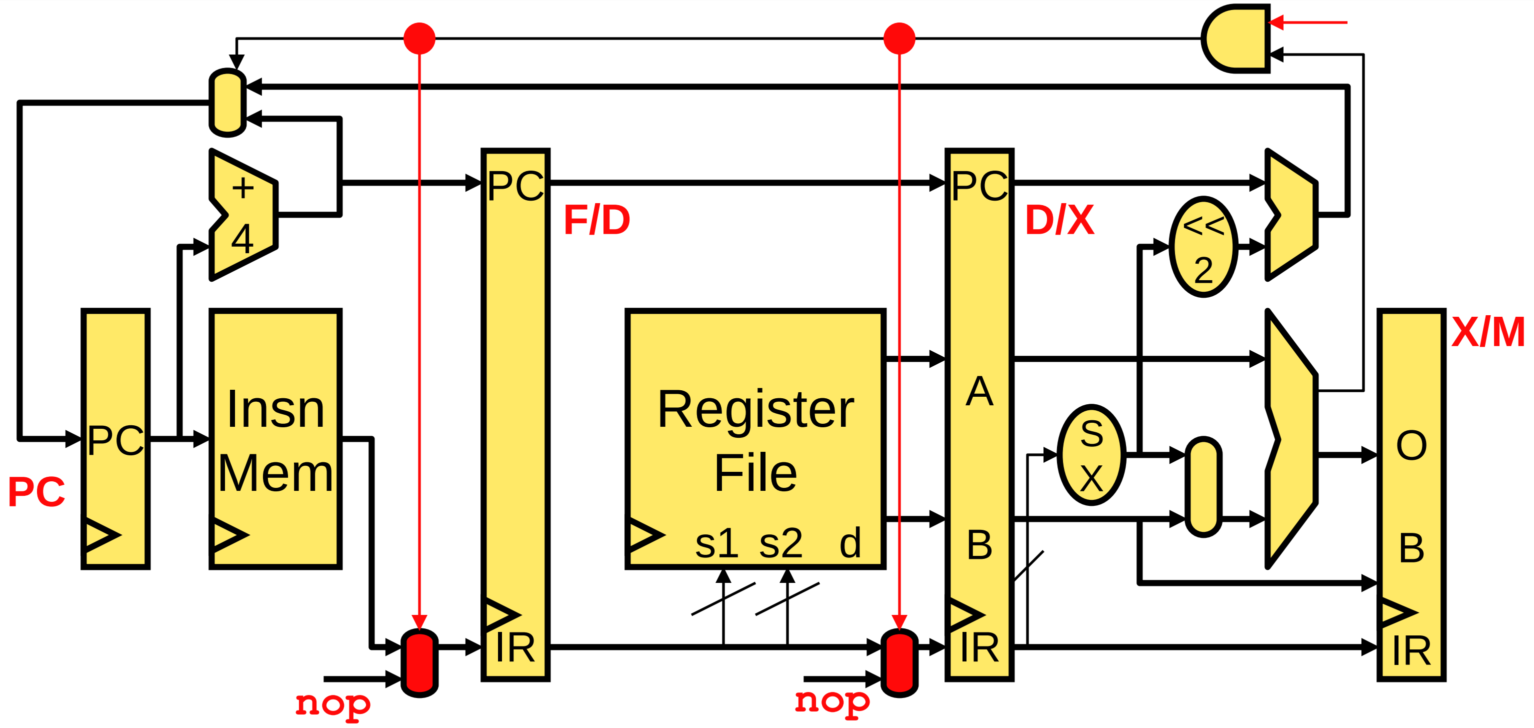
	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,0($3)`		F	D	X	M	W			
`addi $6,$4,1`			F	D	**d***	X	M	W	

Control Hazards



- **Control hazards**
  - Must fetch post branch insns before branch outcome is known
  - Default: assume "**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 permanent state (RegFile, 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 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

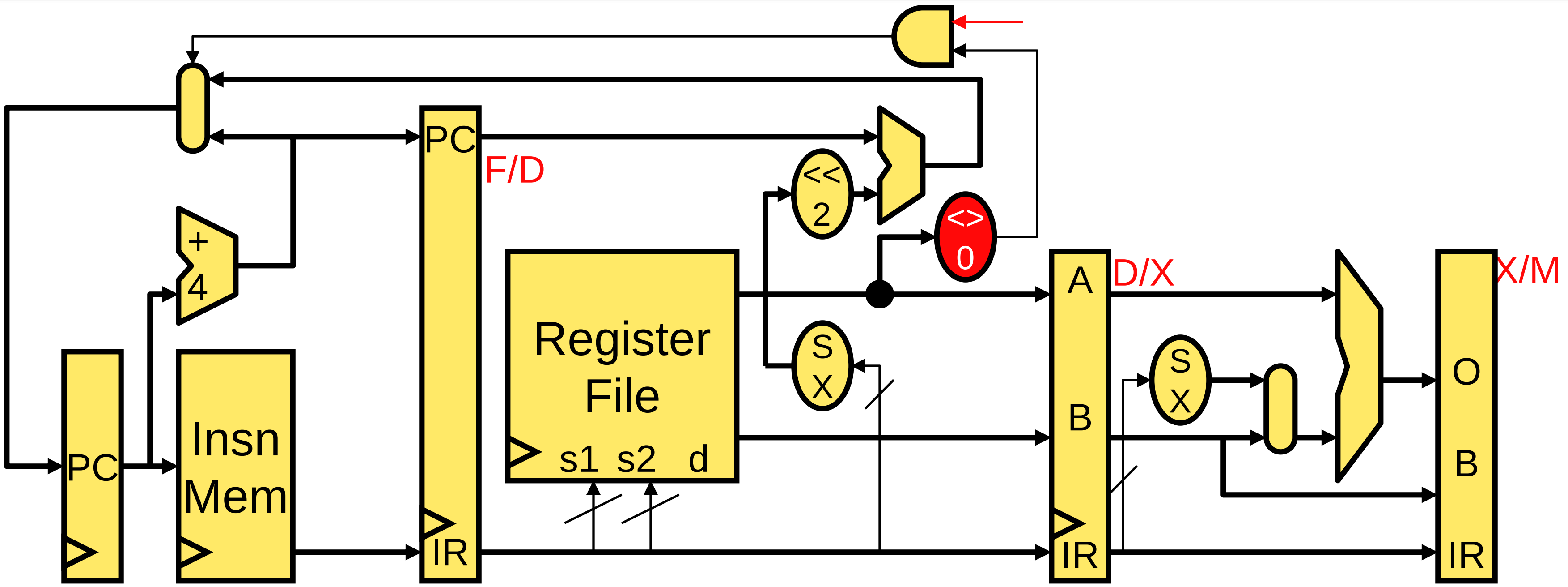
Branch Performance

- Again, 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**

- 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?

Fast Branch



- **Fast branch**: can decide at D, not X
  - Test must be comparison to zero or equality, no time for ALU
  - + New taken branch penalty is 1
  - – Additional insns (`slt`) for more complex tests
  - 25% of branches have complex tests that require extra insn
  - CPI = 1 + 0.20*0.75***1**(branch) + 0.20*0.75*0.25*1(extra insn) = **1.19**

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”: **$[\%_{correct} * \text{gain}] – [(1–\%_{correct}) * \text{penalty}]$**

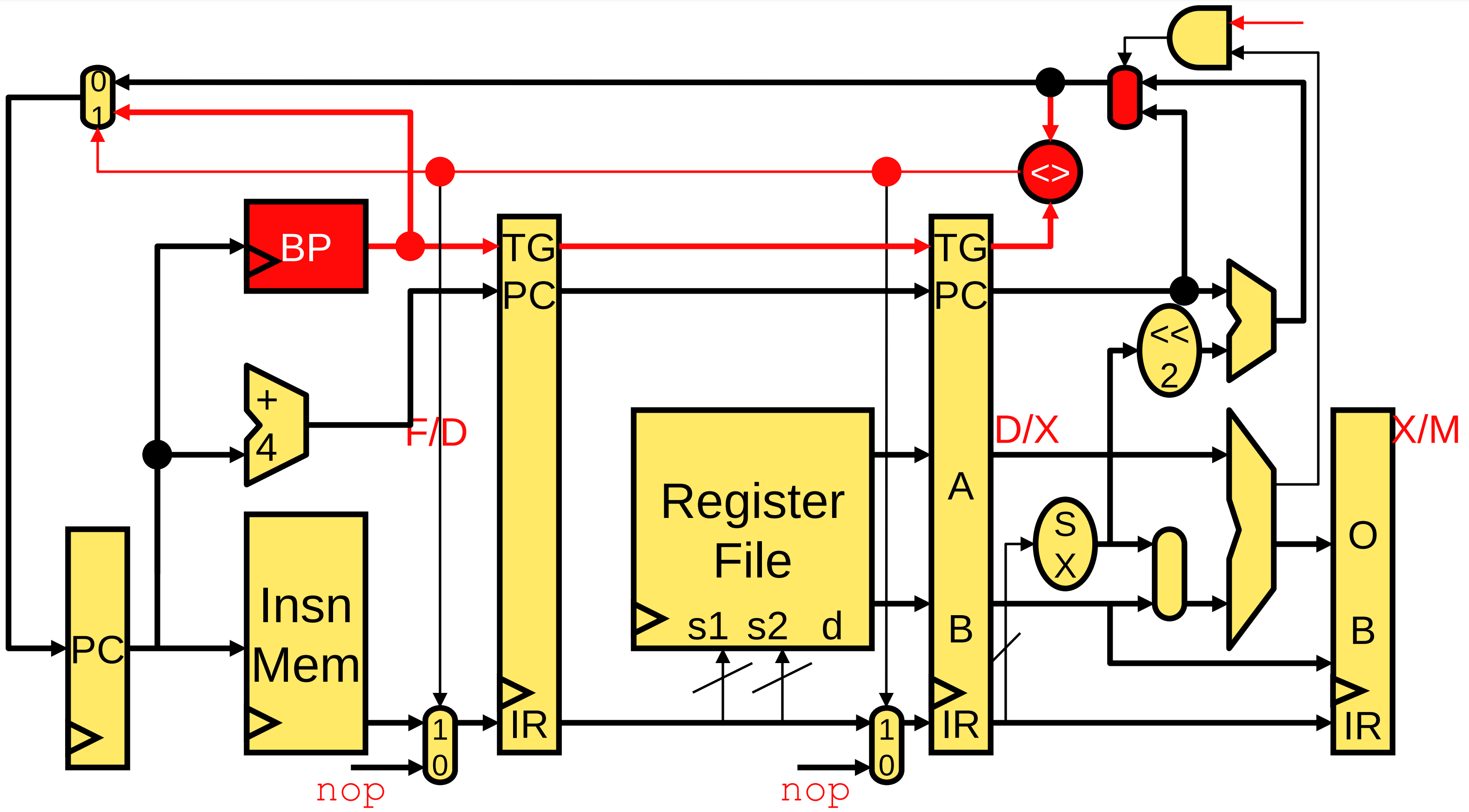
- **Control speculation**: speculation aimed at control hazards

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
    - $\%_{correct}$ = ?
    - Gain = 2 cycles
    - Penalty = 2 cycles

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%

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 by frequency, not frequency * IPC

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 (future lectures)

Summary

- Pipelining
    - Pipeline control