

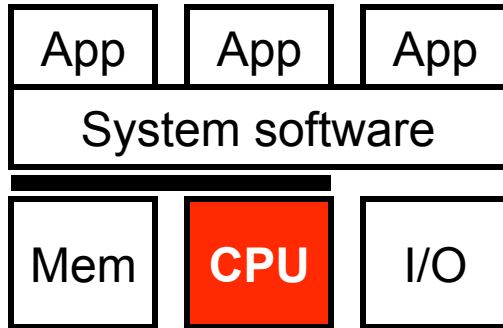
CIS 371

Computer Organization and Design

Unit 5: Pipelining

Based on slides by Prof. Amir Roth & Prof. Milo Martin

This Unit: Pipelining



- Processor performance
 - Latency vs throughput
- Single-cycle & multi-cycle datapaths
- Basic pipelining
- Data hazards
 - Software interlocks and scheduling
 - Hardware interlocks and stalling
 - Bypassing
 - Load-use stalling
- Pipelined multi-cycle operations
- Control hazards
 - Branch prediction



Readings

- P&H
 - Chapter 4 (4.5 – 4.8)

In-Class Exercise

- You have a washer, dryer, and “folder”
 - Each takes 30 minutes per load
 - How long for one load in total?
 - How long for two loads of laundry?
 - How long for 100 loads of laundry?

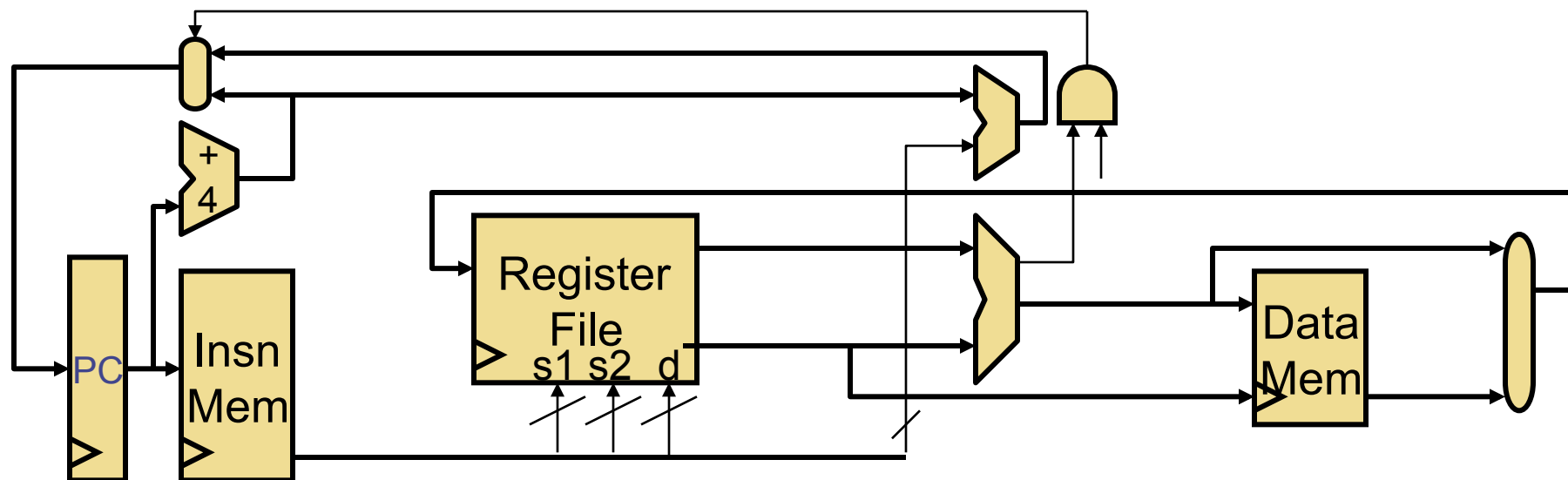
- Now assume:
 - Washing takes 30 minutes, drying 60 minutes, and folding 15 min
 - How long for one load in total?
 - How long for two loads of laundry?
 - How long for 100 loads of laundry?

In-Class Exercise Answers

- You have a washer, dryer, and “folder”
 - Each takes 30 minutes per load
 - How long for one load in total? **90 minutes**
 - How long for two loads of laundry? $90 + 30 = \mathbf{120 \text{ minutes}}$
 - How long for 100 loads of laundry? $90 + 30*99 = \mathbf{3060 \text{ min}}$

- Now assume:
 - Washing takes 30 minutes, drying 60 minutes, and folding 15 min
 - How long for one load in total? **105 minutes**
 - How long for two loads of laundry? $105 + 60 = \mathbf{165 \text{ minutes}}$
 - How long for 100 loads of laundry? $105 + 60*99 = \mathbf{6045 \text{ min}}$

240 → 371



- CIS 240: build something that works
- CIS 371: build something that works “well”
 - “well” means “high-performance” but also cheap, low-power, etc.
 - Mostly “high-performance”
 - So, what is the performance of this?
 - What is performance?

Performance

Processor Performance Equation

$$\text{Execution time} = \text{"seconds per program"} = (\text{instructions/program}) * (\text{seconds/cycle}) * (\text{cycles/instruction})$$

$$(1 \text{ billion instructions}) * (1\text{ns per cycle}) * (1 \text{ cycle per insn}) = 1 \text{ second}$$

- Instructions per program: "dynamic instruction count"
 - Runtime count of instructions executed by the program
 - Determined by program, compiler, instruction set architecture (ISA)
- **Cycles per instruction: "CPI"** (typical range: 2 to 0.5)
 - On average, how many *cycles* does an instruction take to execute?
 - Determined by program, compiler, ISA, micro-architecture
- **Sec. per cycle: "clock period"** (typical range: 2ns to 0.25ns)
 - Reciprocal is frequency: 0.5 Ghz to 4 Ghz (1 Hertz = 1 cycle per sec)
 - Determined by micro-architecture, technology parameters
- For minimum execution time, minimize each term
 - Difficult: ***often pull against one another***

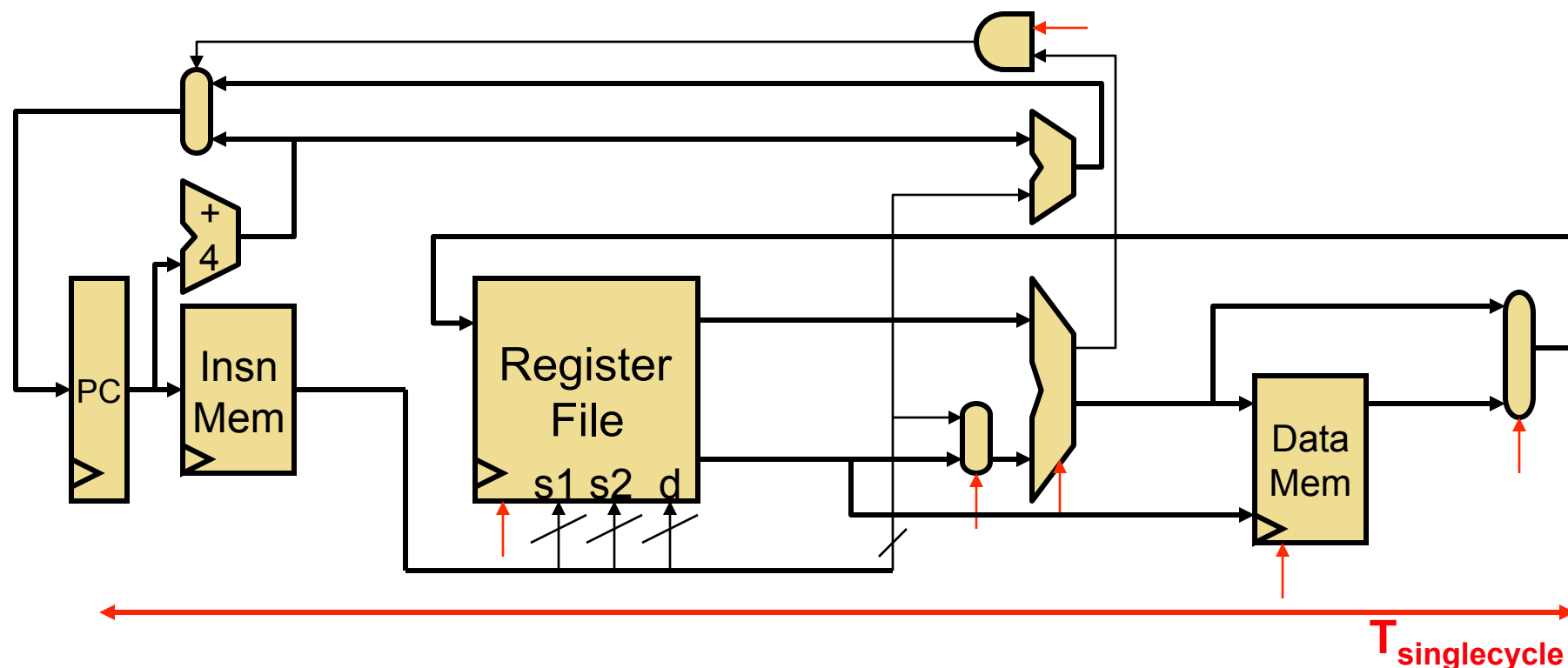
Cycles per Instruction (CPI)

- **CPI**: Cycle/instruction for **on average**
 - **IPC** = $1/\text{CPI}$
 - Used more frequently than CPI
 - Favored because “bigger is better”, but harder to compute with
 - Different instructions have different cycle costs
 - E.g., “add” typically takes 1 cycle, “divide” takes >10 cycles
 - Depends on relative instruction frequencies
- CPI example
 - A program executes equal: integer, floating point (FP), memory ops
 - Cycles per instruction type: integer = 1, memory = 2, FP = 3
 - What is the CPI? $(33\% * 1) + (33\% * 2) + (33\% * 3) = 2$
 - **Caveat**: this sort of calculation ignores many effects
 - Back-of-the-envelope arguments only

Improving Clock Frequency

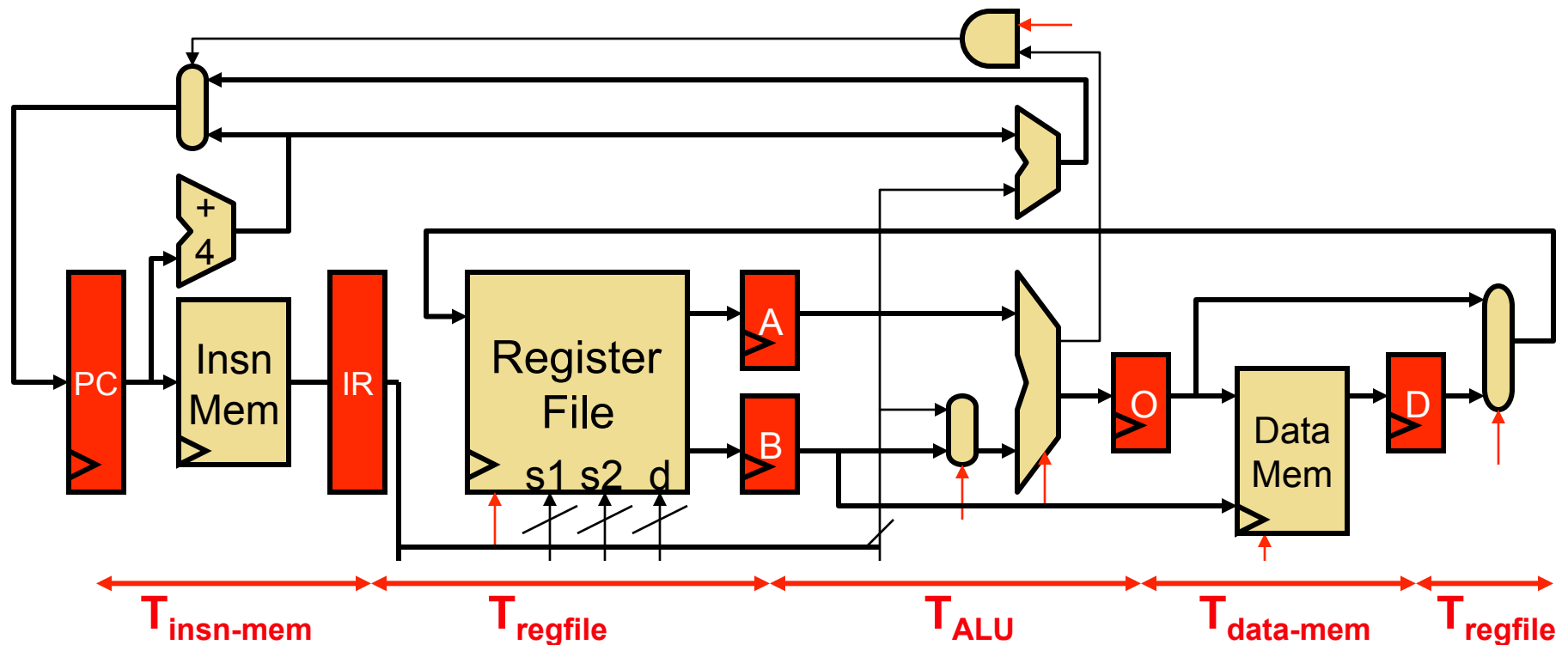
- **Faster transistors**
- Micro-architectural techniques
 - **Multi-cycle processors**
 - Break each instruction into small bits
 - Less logic delay -> improved clock frequency
 - Different instructions take different number of cycles
 - $CPI > 1$
 - **Pipelined processors**
 - As above, but overlap parts of instruction (parallelism!)
 - Faster clock, but CPI can still be around 1

Single-Cycle Datapath



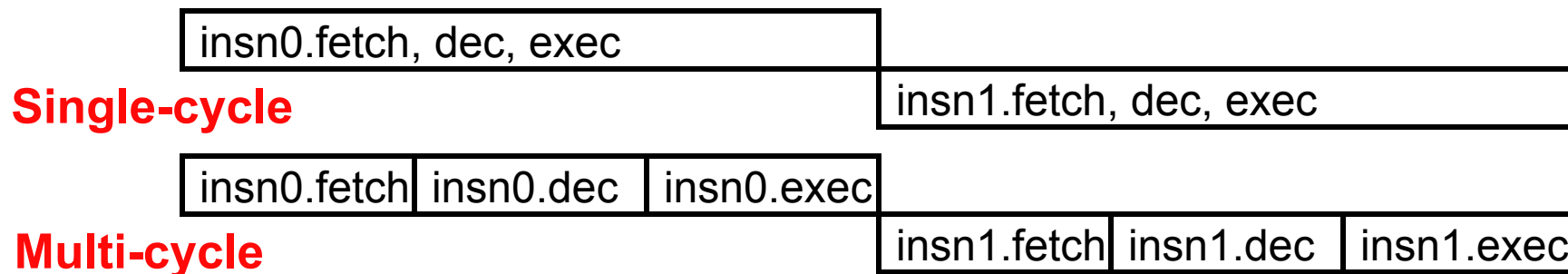
- **Single-cycle datapath:** true “atomic” fetch/execute loop
 - Fetch, decode, execute one complete instruction every cycle
 - + Takes 1 cycle to execution any instruction by definition (“CPI” is 1)
 - Long clock period: to accommodate slowest instruction (worst-case delay through circuit, must wait this long *every* time)

Multi-Cycle Datapath



- **Multi-cycle datapath:** attacks slow clock
 - Fetch, decode, execute one complete insn over multiple cycles
 - **Allows insns to take different number of cycles**
 - + Opposite of single-cycle: short clock period (less "work" per cycle)
 - Multiple cycles per instruction (higher "CPI")

Recap: Single-cycle vs. Multi-cycle



- **Single-cycle datapath:**
 - Fetch, decode, execute one complete instruction every cycle
 - + Low CPI: 1 by definition
 - Long clock period: to accommodate slowest instruction
- **Multi-cycle datapath:** attacks slow clock
 - Fetch, decode, execute one complete insn over multiple cycles
 - **Allows insns to take different number of cycles**
 - + Opposite of single-cycle: short clock period (less "work" per cycle)
 - Multiple cycles per instruction (higher "CPI")

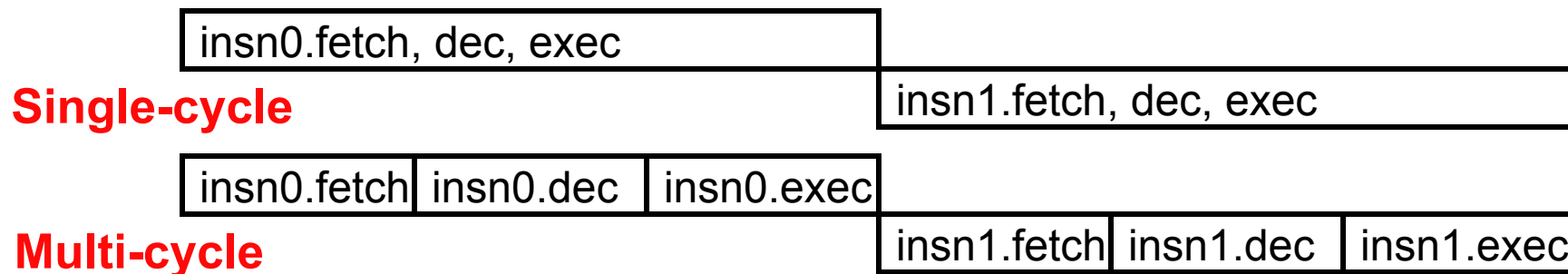
Single-cycle vs. Multi-cycle Performance

- Single-cycle
 - Clock period = 50ns, CPI = 1
 - Performance = **50ns/insn**
- Multi-cycle has opposite performance split of single-cycle
 - + Shorter clock period
 - Higher CPI
- Multi-cycle
 - Branch: 20% (**3** cycles), load: 20% (**5** cycles), ALU: 60% (**4** cycles)
 - Clock period = **11ns**, $CPI = (20\% * 3) + (20\% * 5) + (60\% * 4) = 4$
 - Why is clock period 11ns and not 10ns? overheads
 - Performance = **44ns/insn**

Latency versus Throughput

- **Latency (execution time)**: time to finish a fixed task
- **Throughput (bandwidth)**: number of tasks in fixed time
 - Different: exploit parallelism for throughput, not latency (e.g., bread)
 - Often contradictory (latency **vs.** throughput)
 - Will see many examples of this
 - Choose definition of performance that matches your goals
 - Scientific program? Latency, web server: throughput?
- Example: move people 10 miles
 - Car: capacity = 5, speed = 60 miles/hour
 - Bus: capacity = 60, speed = 20 miles/hour
 - Latency: **car = 10 min**, bus = 30 min
 - Throughput: car = 15 PPH (count return trip), **bus = 60 PPH**
- Fastest way to send 1TB of data? (at 100+ mbits/second)

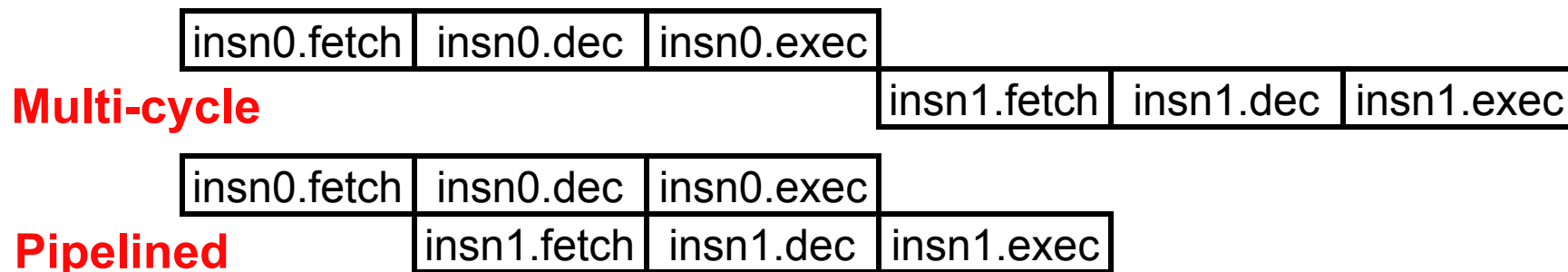
Latency versus Throughput



- Can we have both low CPI and short clock period?
 - Not if datapath executes only one insn at a time
- Latency and throughput: two views of performance ...
 - (1) at the program level and (2) at the instructions level
- Single instruction latency
 - Doesn't matter: programs comprised of billions of instructions
 - Difficult to reduce anyway
- Goal is to make programs, not individual insns, go faster
 - Instruction throughput → program latency
 - Key: **exploit inter-insn parallelism**

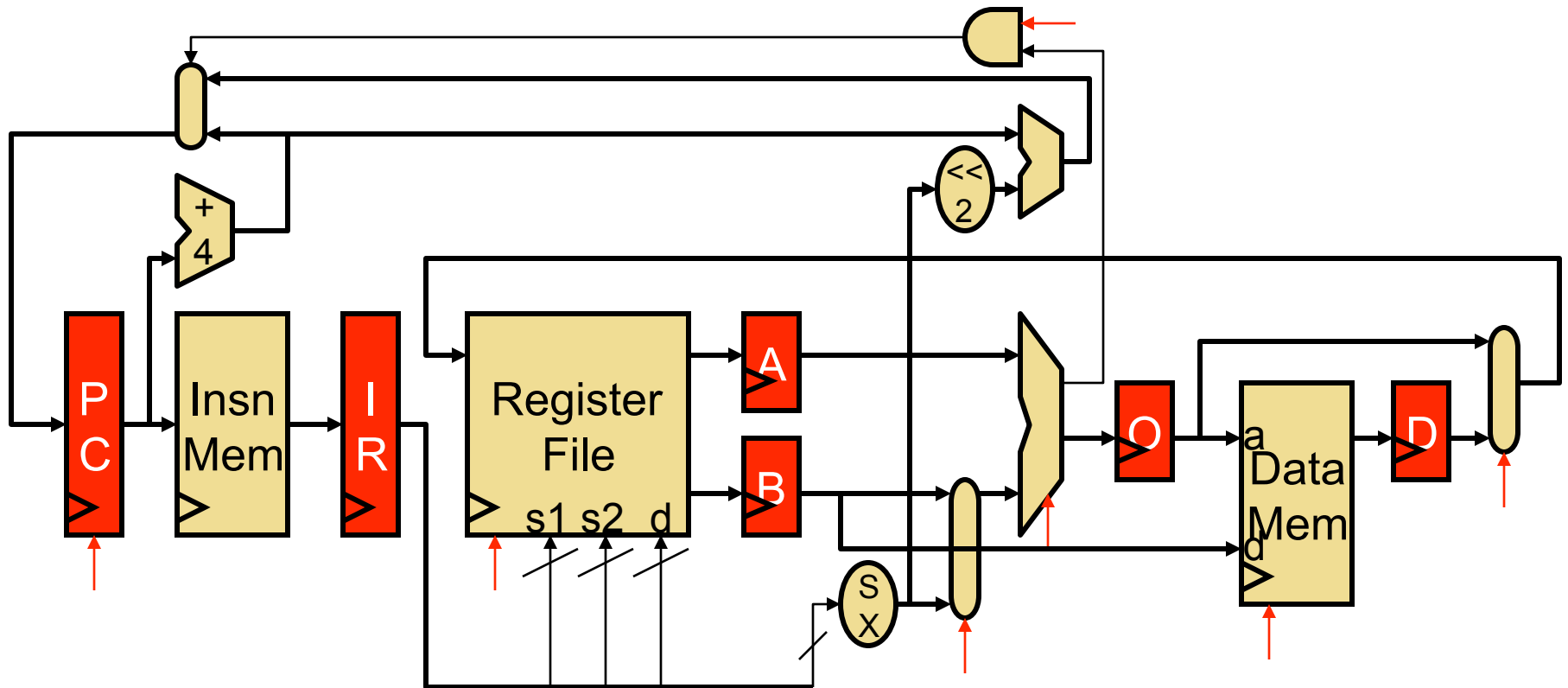
Pipelined Datapath

Pipelining

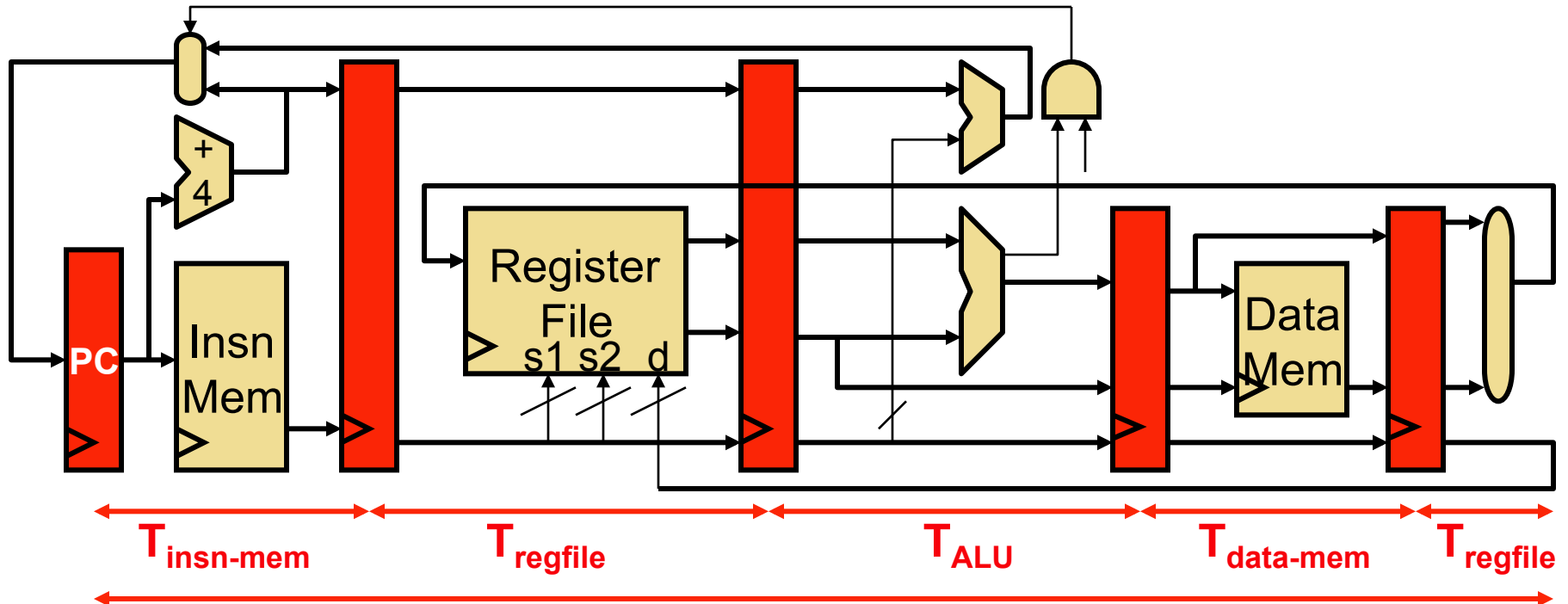


- Important performance technique
 - **Improves instruction throughput rather instruction latency**
- Begin with multi-cycle design
 - When insn advances from stage 1 to 2, next insn enters at stage 1
 - Form of parallelism: “insn-stage parallelism”
 - Maintains illusion of sequential fetch/execute loop
 - Individual instruction takes the same number of stages
 - + **But instructions enter and leave at a much faster rate**
- Laundry analogy

5 Stage Multi-Cycle Datapath

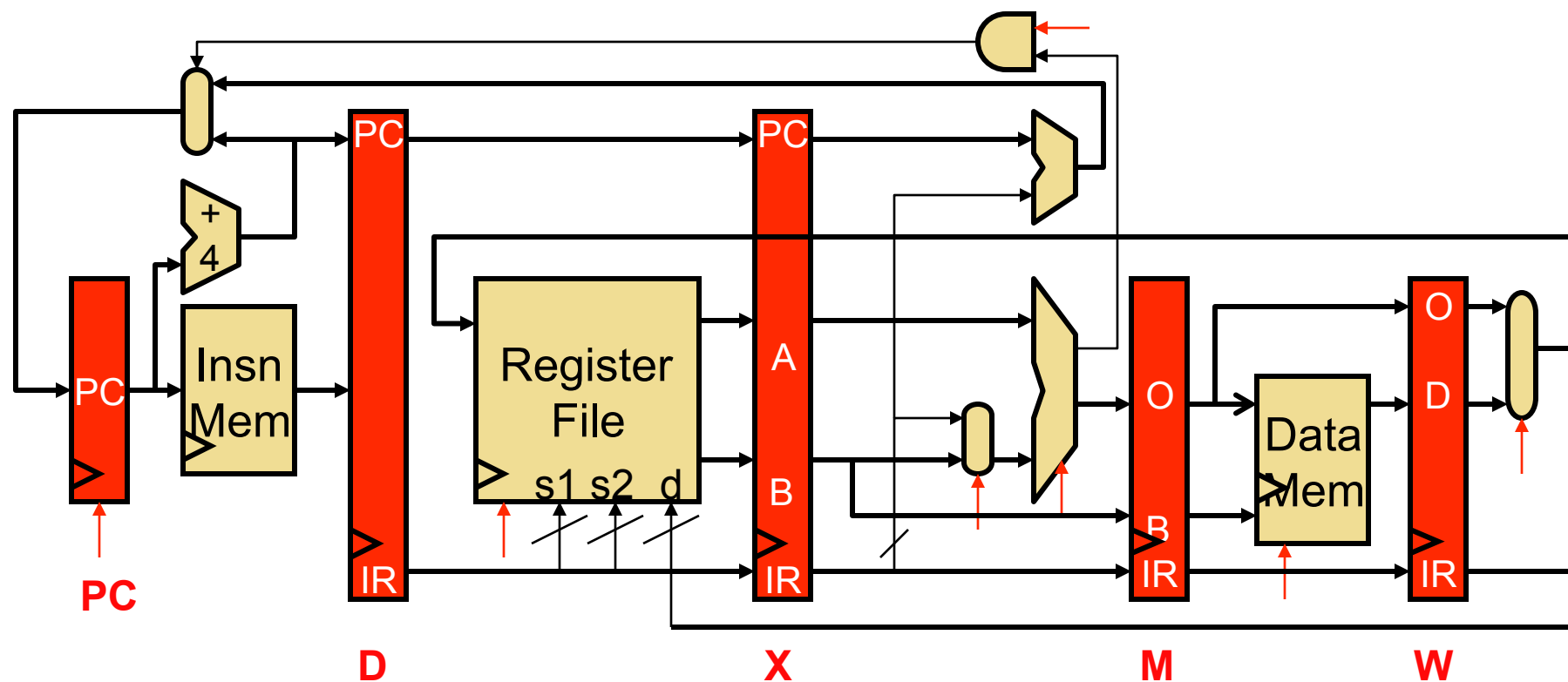


5 Stage Pipeline: Inter-Insn Parallelism



- **Pipelining**: cut datapath into N stages (here 5) $T_{\text{singlecycle}}$
 - One insn in each stage in each cycle
 - + Clock period = $\text{MAX}(T_{\text{insn-mem}}, T_{\text{regfile}}, T_{\text{ALU}}, T_{\text{data-mem}})$
 - + Base CPI = 1: insn enters and leaves every cycle
 - Actual CPI > 1: pipeline must often "stall"
 - Individual insn latency increases (pipeline overhead), not the point

5 Stage Pipelined Datapath



- Five stage: **F**etch, **D**ecode, **eX**ecute, **M**emory, **W**riteback
 - Nothing magical about 5 stages (Pentium 4 had 22 stages!)
- Latches (pipeline registers) named by stages they begin
 - **PC, D, X, M, W**

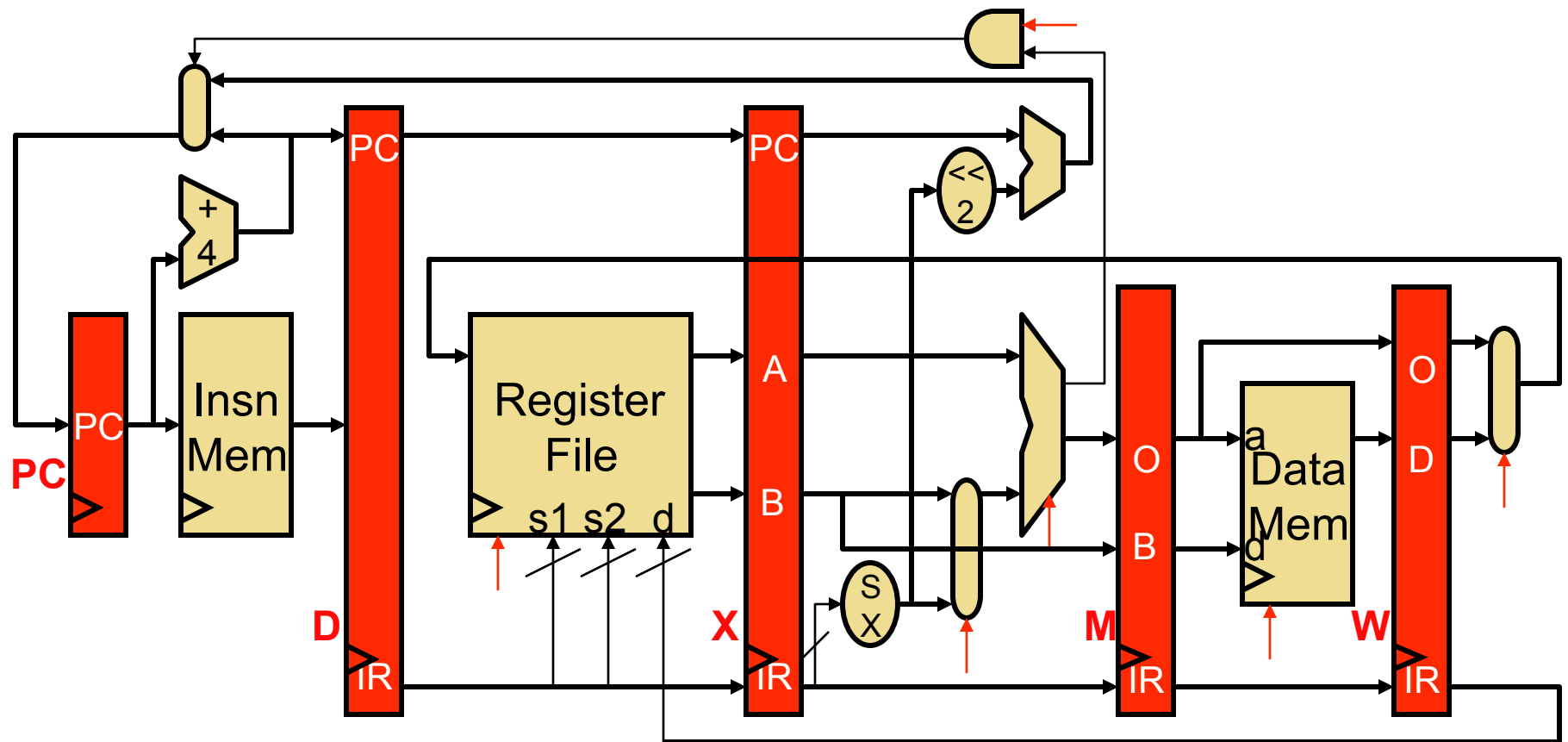
More Terminology & Foreshadowing

- **Scalar pipeline**: one insn per stage per cycle
 - Alternative: “superscalar” (later)
- **In-order pipeline**: insns enter execute stage in order
 - Alternative: “out-of-order” (later)
- **Pipeline depth**: number of pipeline stages
 - Nothing magical about five
 - Contemporary high-performance cores have ~15 stage pipelines

Instruction Convention

- Different ISAs use inconsistent register orders
- Some ISAs (for example MIPS)
 - Instruction destination (i.e., output) **on the left**
 - add \$1, \$2, \$3 means $\$1 \leftarrow \$2 + \$3$
- Other ISAs
 - Instruction destination (i.e., output) **on the right**
 - add r1, r2, r3 means $r1 + r2 \rightarrow r3$
 - ld 8(r5), r4 means $\text{mem}[r5+8] \rightarrow r4$
 - st r4, 8(r5) means $r4 \rightarrow \text{mem}[r5+8]$
- Will try to specify to avoid confusion, next slides MIPS style

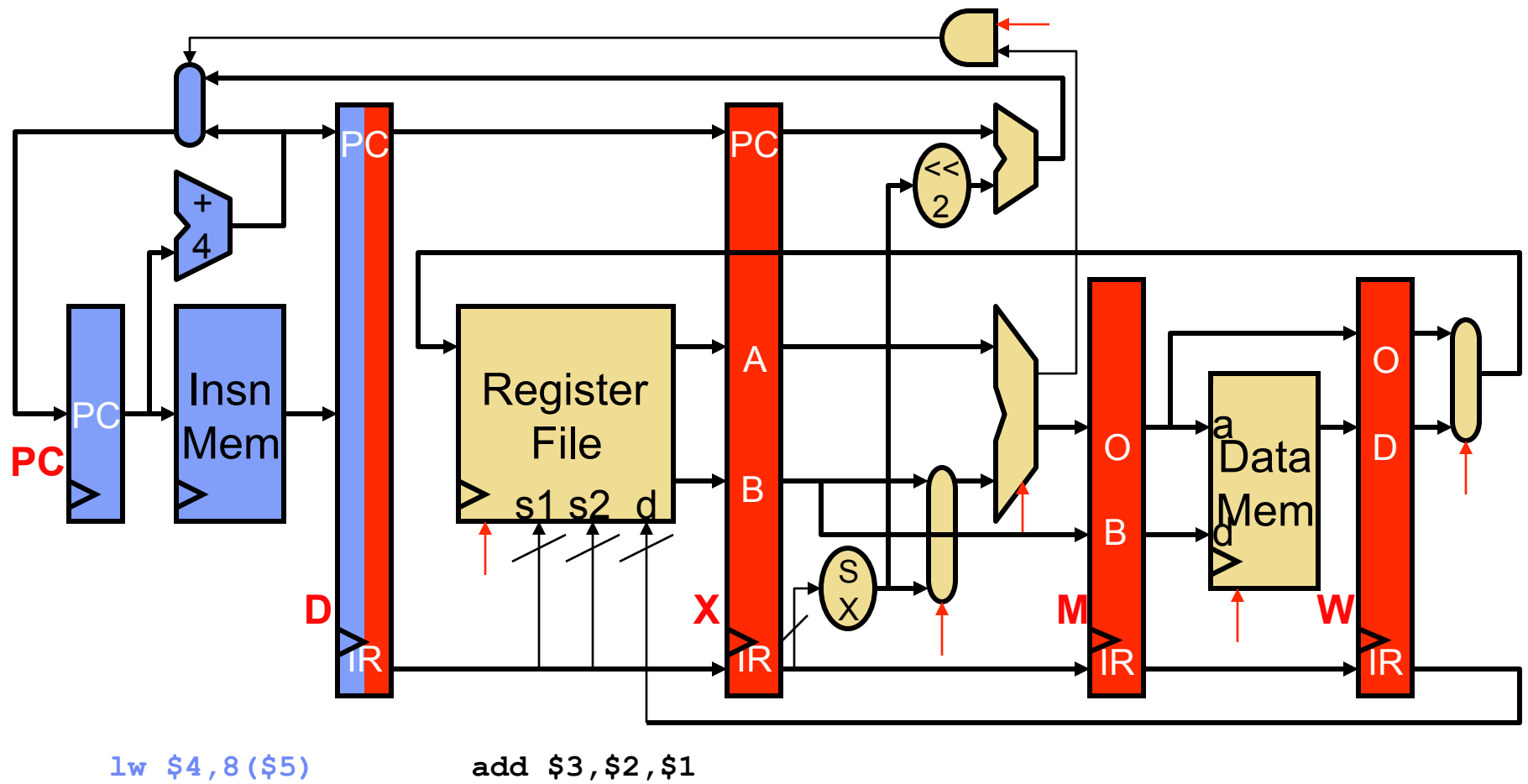
Pipeline Example: Cycle 1



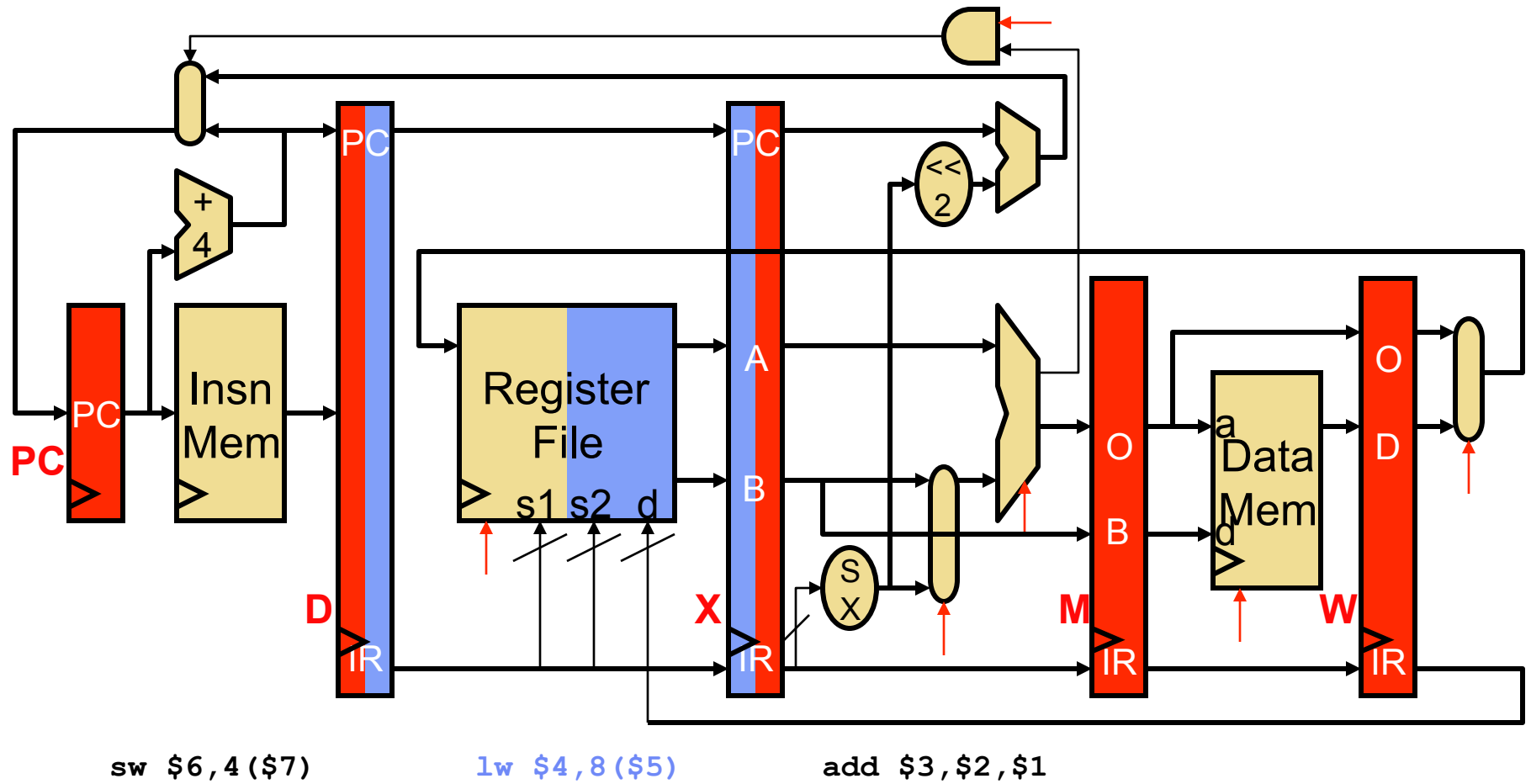
add \$3,\$2,\$1

- 3 instructions

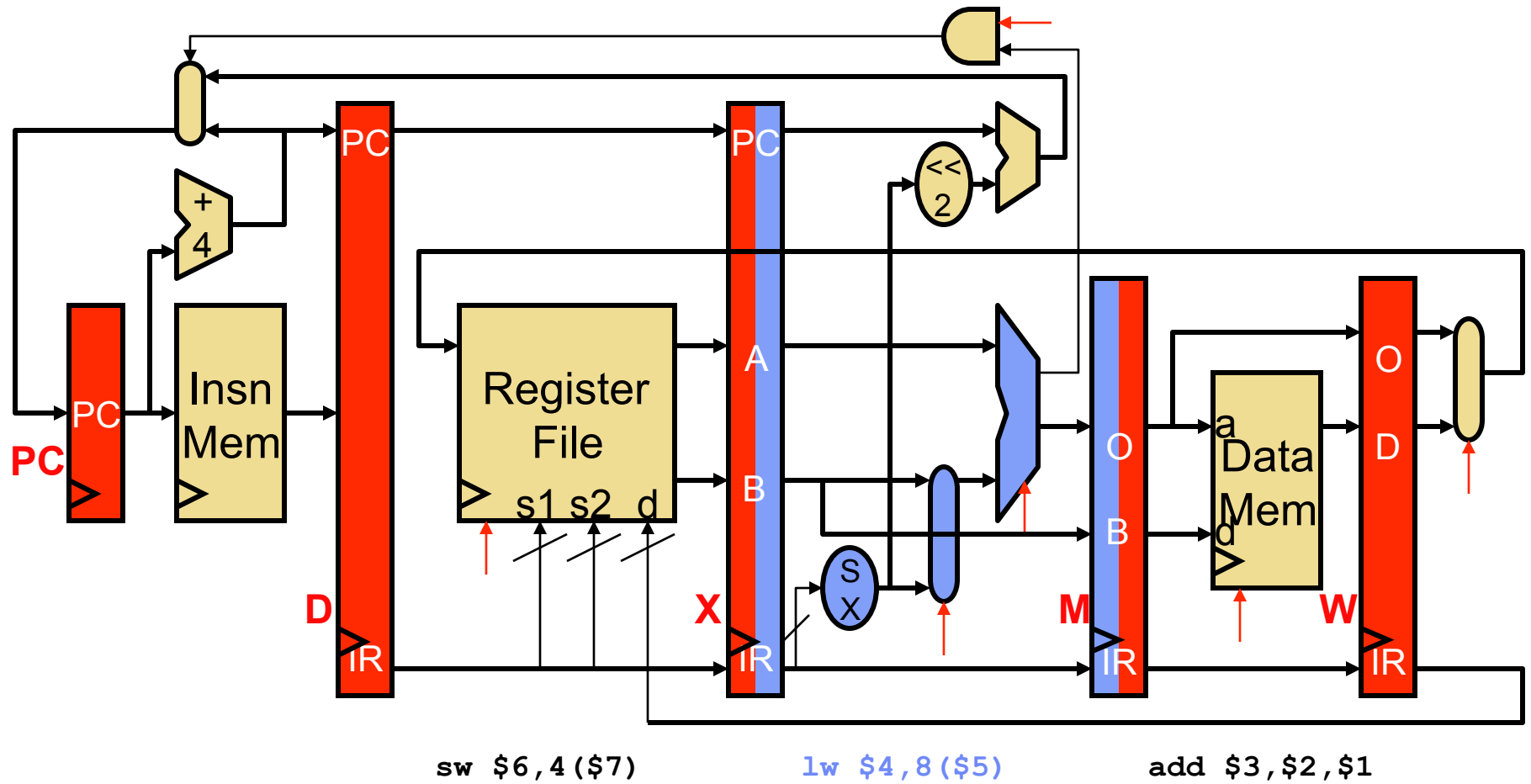
Pipeline Example: Cycle 2



Pipeline Example: Cycle 3

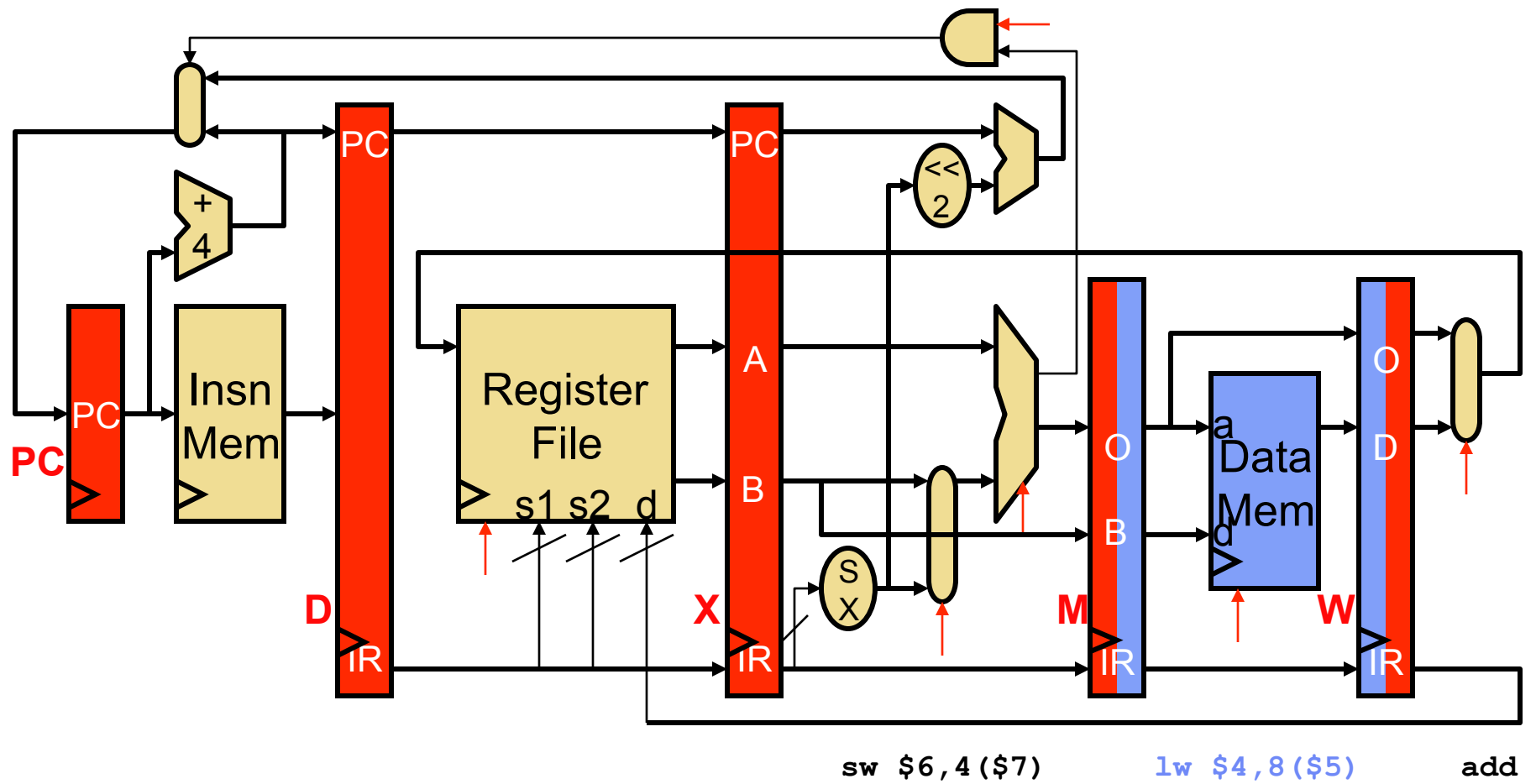


Pipeline Example: Cycle 4

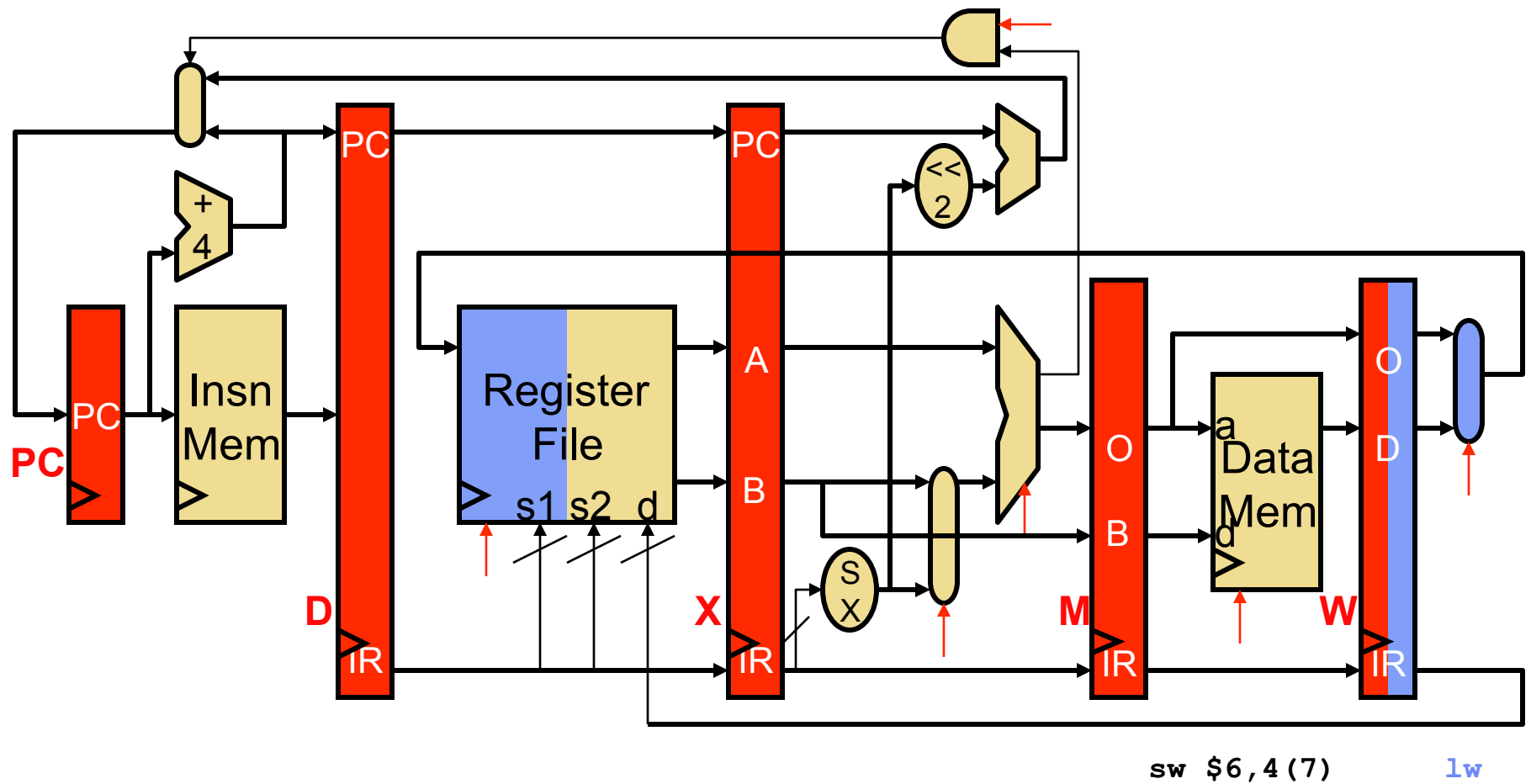


- 3 instructions

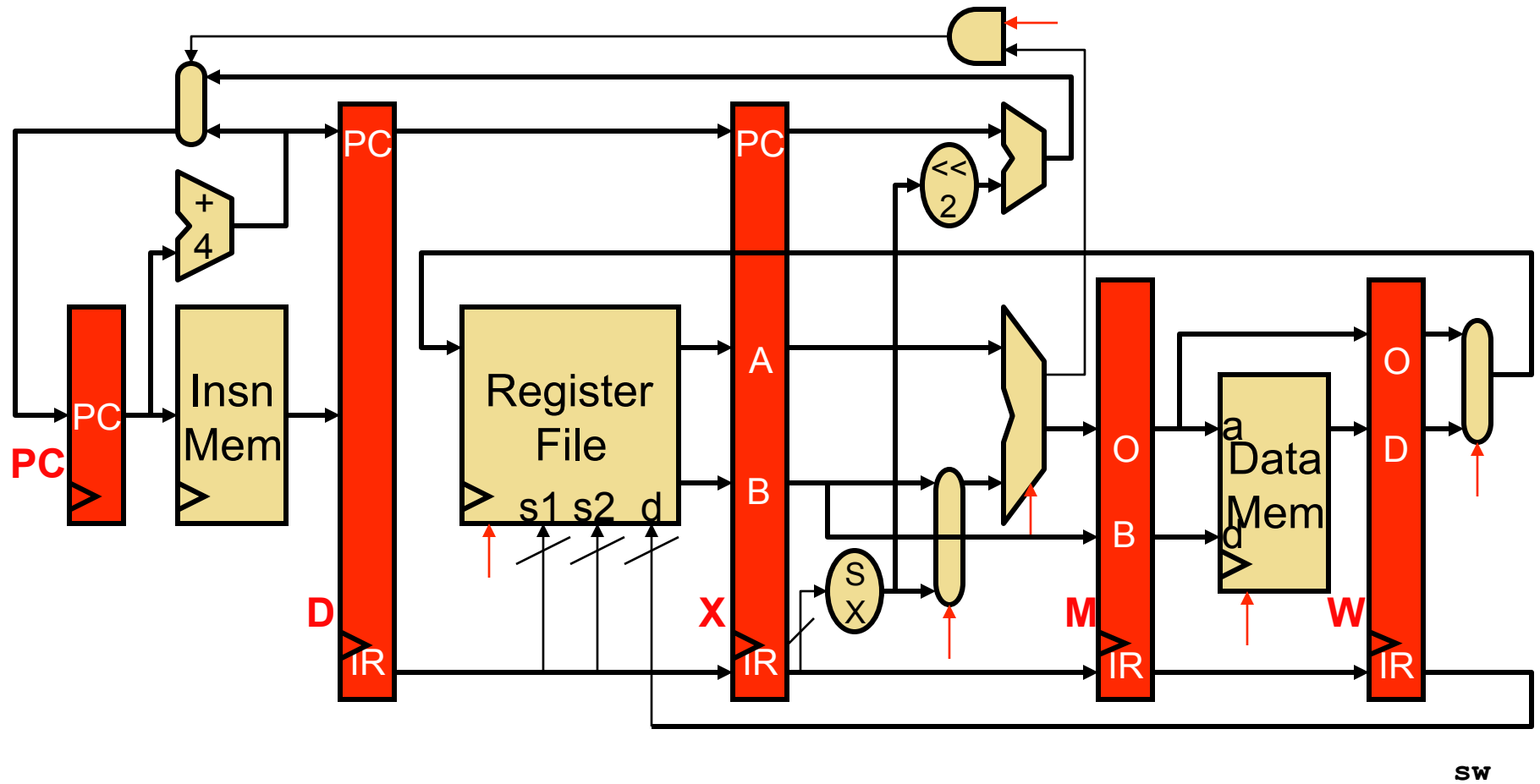
Pipeline Example: Cycle 5



Pipeline Example: Cycle 6



Pipeline Example: Cycle 7



Pipeline Diagram

- **Pipeline diagram**: shorthand for what we just saw
 - Across: cycles
 - Down: insns
 - Convention: **X** means `lw $4, 8($5)` finishes execute stage and writes into M latch at end of cycle 4

	1	2	3	4	5	6	7	8	9
<code>add \$3, \$2, \$1</code>	F	D	X	M	W				
<code>lw \$4, 8(\$5)</code>		F	D	X	M	W			
<code>sw \$6, 4(\$7)</code>			F	D	X	M	W		

Example Pipeline Perf. Calculation

- Single-cycle
 - Clock period = 50ns, CPI = 1
 - Performance = 50ns/insn
- Multi-cycle
 - Branch: 20% (3 cycles), load: 20% (5 cycles), ALU: 60% (4 cycles)
 - Clock period = 11ns, CPI = $(20\% * 3) + (20\% * 5) + (60\% * 4) = 4$
 - Performance = 44ns/insn
- 5-stage pipelined
 - Clock period = **12ns** approx. (50ns / 5 stages) + overheads
 - + CPI = **1** (each insn takes 5 cycles, but 1 completes each cycle)
 - + Performance = **12ns/insn**
 - Well actually ... CPI = 1 + some penalty for pipelining (next)
 - CPI = **1.5** (on average insn completes every 1.5 cycles)
 - Performance = **18ns/insn**
 - Much higher performance than single-cycle or multi-cycle

Q1: Why Is Pipeline Clock Period ...

- ... $>$ (delay thru datapath) / (number of pipeline stages)?
 - Three reasons:
 - Latches add delay
 - Pipeline stages have different delays, clock period is max delay
 - Extra datapaths for pipelining (bypassing paths)
 - These factors have implications for ideal number pipeline stages
 - Diminishing clock frequency gains for longer (deeper) pipelines

Q2: Why Is Pipeline CPI...

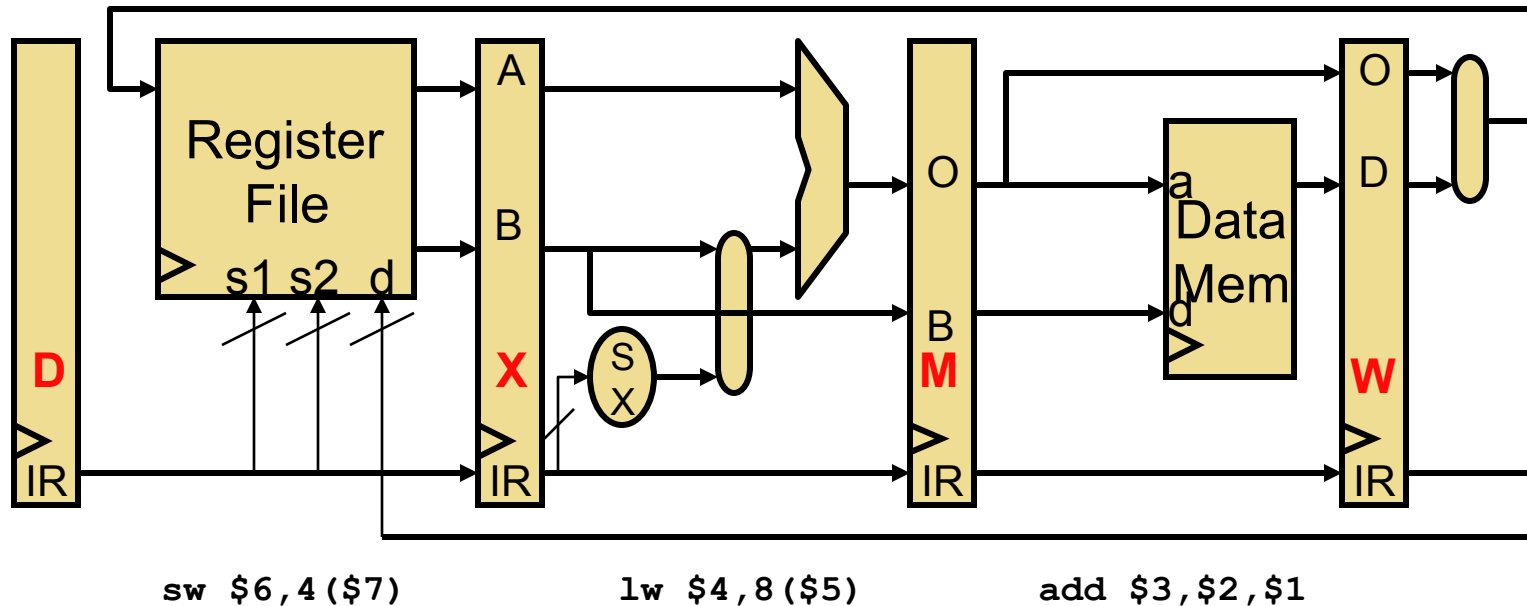
- ... > 1?
 - CPI for scalar in-order pipeline is 1 + **stall penalties**
 - Stalls used to resolve hazards
 - **Hazard**: condition that jeopardizes sequential illusion
 - **Stall**: pipeline delay introduced to restore sequential illusion
- Calculating pipeline CPI
 - **Frequency of stall * stall cycles**
 - Penalties add (stalls generally don't overlap in in-order pipelines)
 - $1 + (\text{stall-freq}_1 * \text{stall-cyc}_1) + (\text{stall-freq}_2 * \text{stall-cyc}_2) + \dots$
- Correctness/performance/make common case fast
 - Long penalties OK if they are rare, e.g., $1 + (0.01 * 10) = 1.1$
 - Stalls also have implications for ideal number of pipeline stages

Data Dependences, Pipeline Hazards, and Bypassing

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

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

Dependent Operations

- Independent operations

```
add $3, $2, $1  
add $6, $5, $4
```

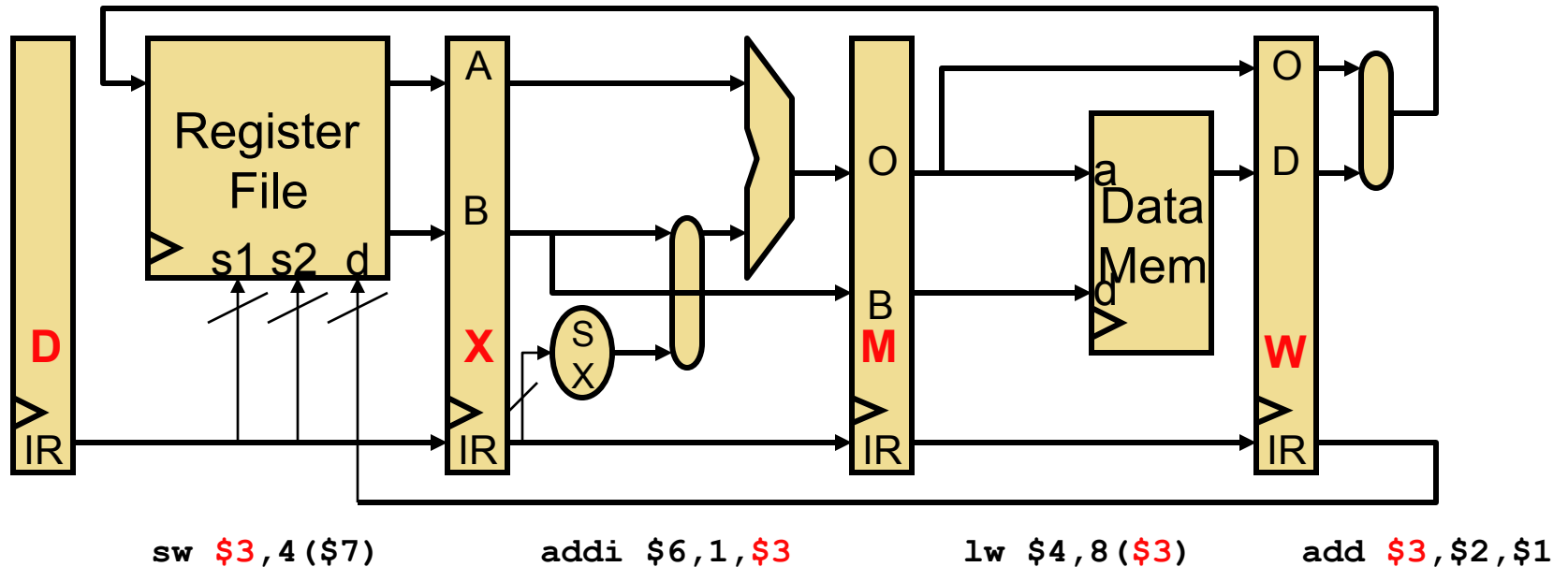
- Would this program execute correctly on a pipeline?

```
add $3, $2, $1  
add $6, $5, $3
```

- What about this program?

```
add $3, $2, $1  
lw $4, 8($3)  
addi $6, 1, $3  
sw $3, 8($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` two cycles ago → got wrong value
 - `addi` read `$3` one cycle ago → got wrong value
 - `sw` is reading `$3` this cycle → maybe (depending on regfile design)

Fixing Register Data Hazards

- Can only read register value three cycles after writing it
- **Option #1: 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"
 - Do not write it: otherwise meaning of program changes
 - Do not read it: otherwise create new data hazard
 - **Code scheduling**: compiler moves around existing insns to do this
 - If none can be found, must use **nops** (no-operation)
- This is called **software interlocks**
 - **MIPS**: **M**icroprocessor w/out **I**nterlocking **P**ipeline **S**tages

Software Interlock Example

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



- Can any of last three insns be scheduled between first two
 - `sw $7, 8($3)`? No, creates hazard with `add $3, $2, $1`
 - `add $6, $2, $8`? Okay
 - `addi $3, $5, 4`? No, `lw` would read \$3 from it
 - Still need one more insn, use `nop`

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

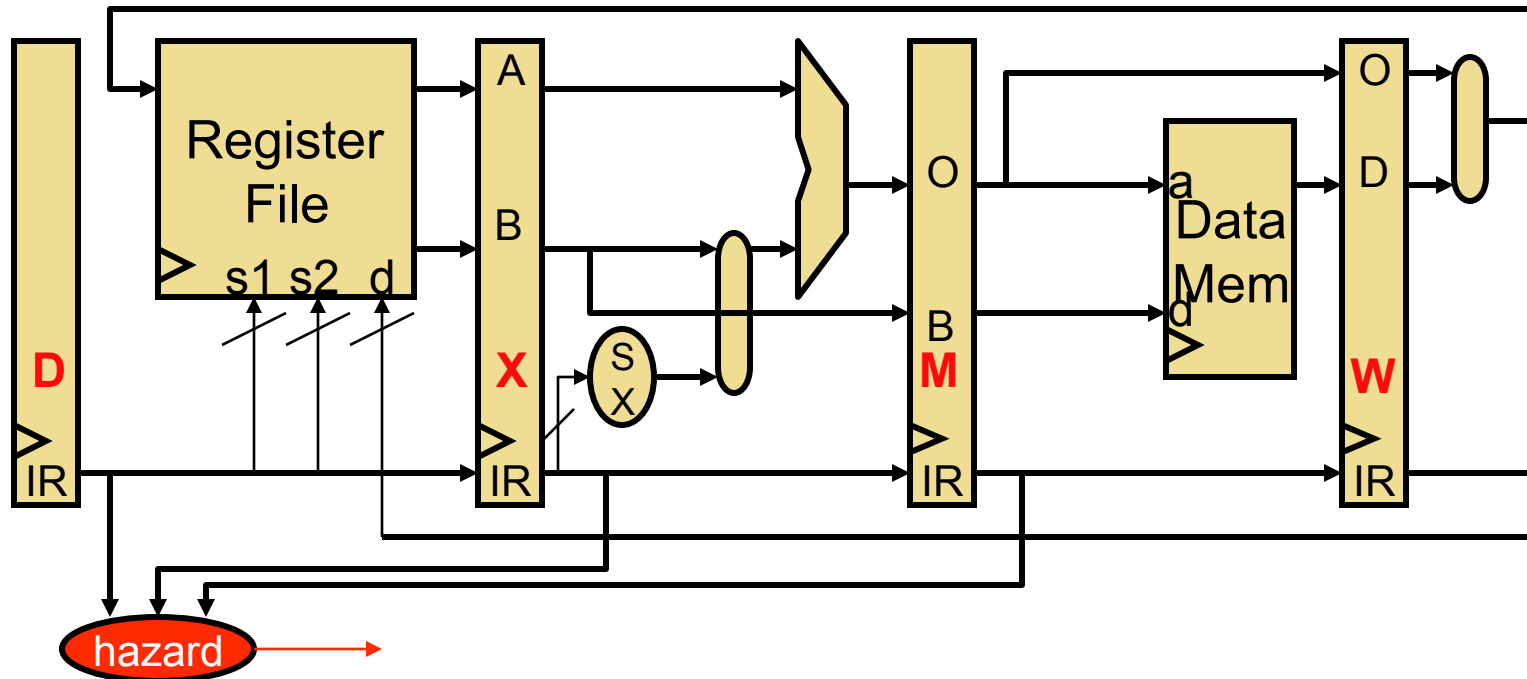
Software Interlock Performance

- Assume
 - Branch: 20%, load: 20%, store: 10%, other: 50%
- For software interlocks, let's assume:
 - 20% of insns require insertion of 1 `nop`
 - 5% of insns require insertion of 2 `nops`
- Result:
 - CPI is still 1 technically
 - But now there are more insns
 - $\#insns = 1 + 0.20*1 + 0.05*2 = \mathbf{1.3}$
 - **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
- **Option #2: hardware interlocks**
 - Processor detects data hazards and fixes them
 - Resolves the above compatibility concern
 - Two aspects to this
 - Detecting hazards
 - Fixing hazards

Detecting Data Hazards

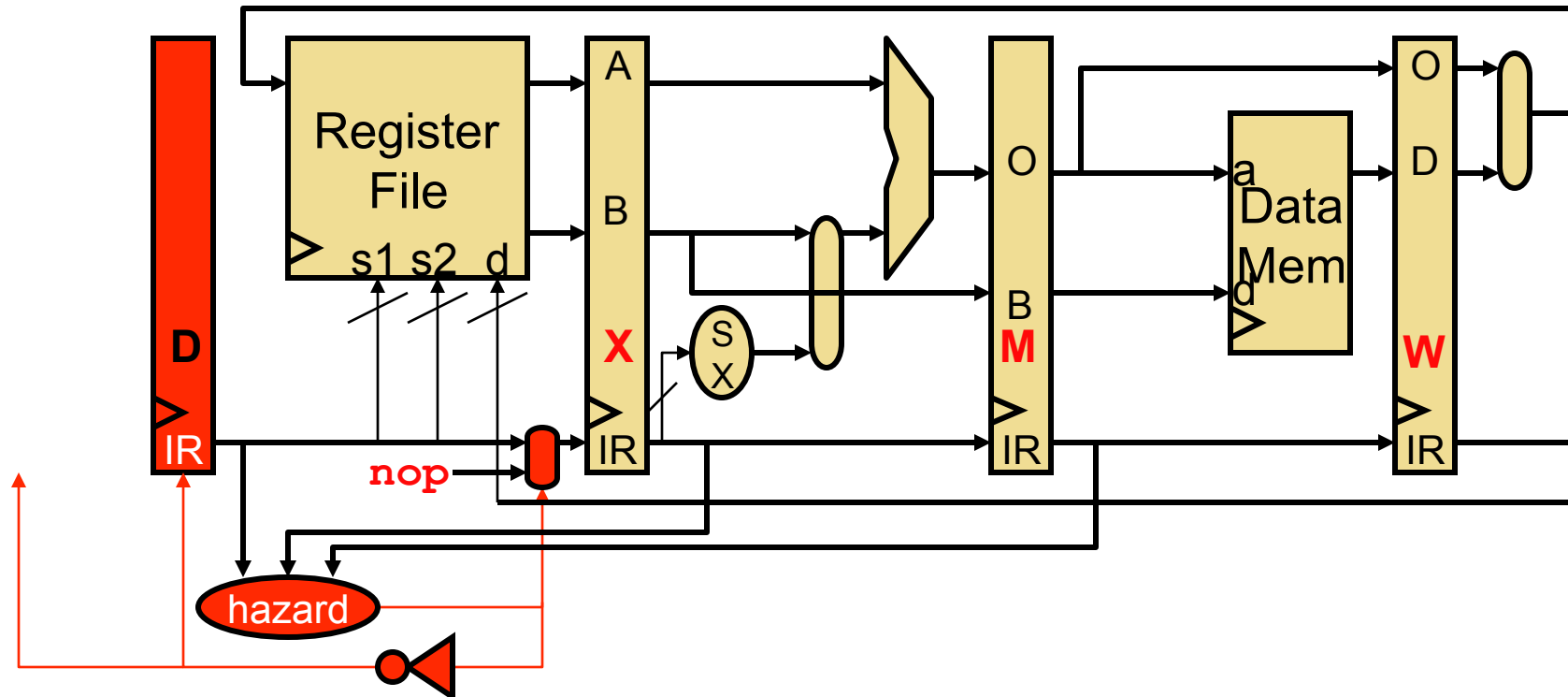


- Compare input register names of insn in D stage with output register names of older insns in pipeline

Stall =

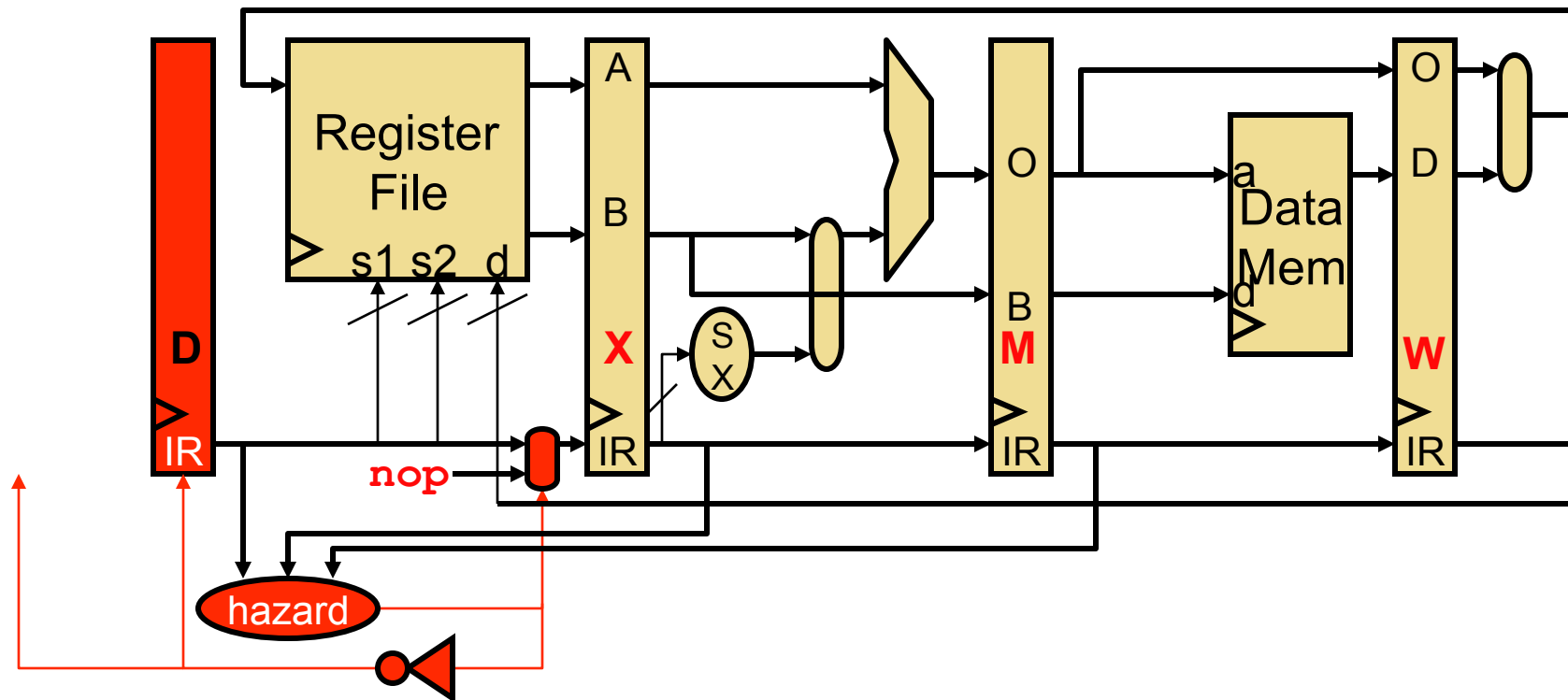
```
(D.IR.RegSrc1 == X.IR.RegDest) ||  
(D.IR.RegSrc2 == X.IR.RegDest) ||  
(D.IR.RegSrc1 == M.IR.RegDest) ||  
(D.IR.RegSrc2 == M.IR.RegDest)
```

Fixing Data Hazards



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

Hardware Interlock Example: cycle 1



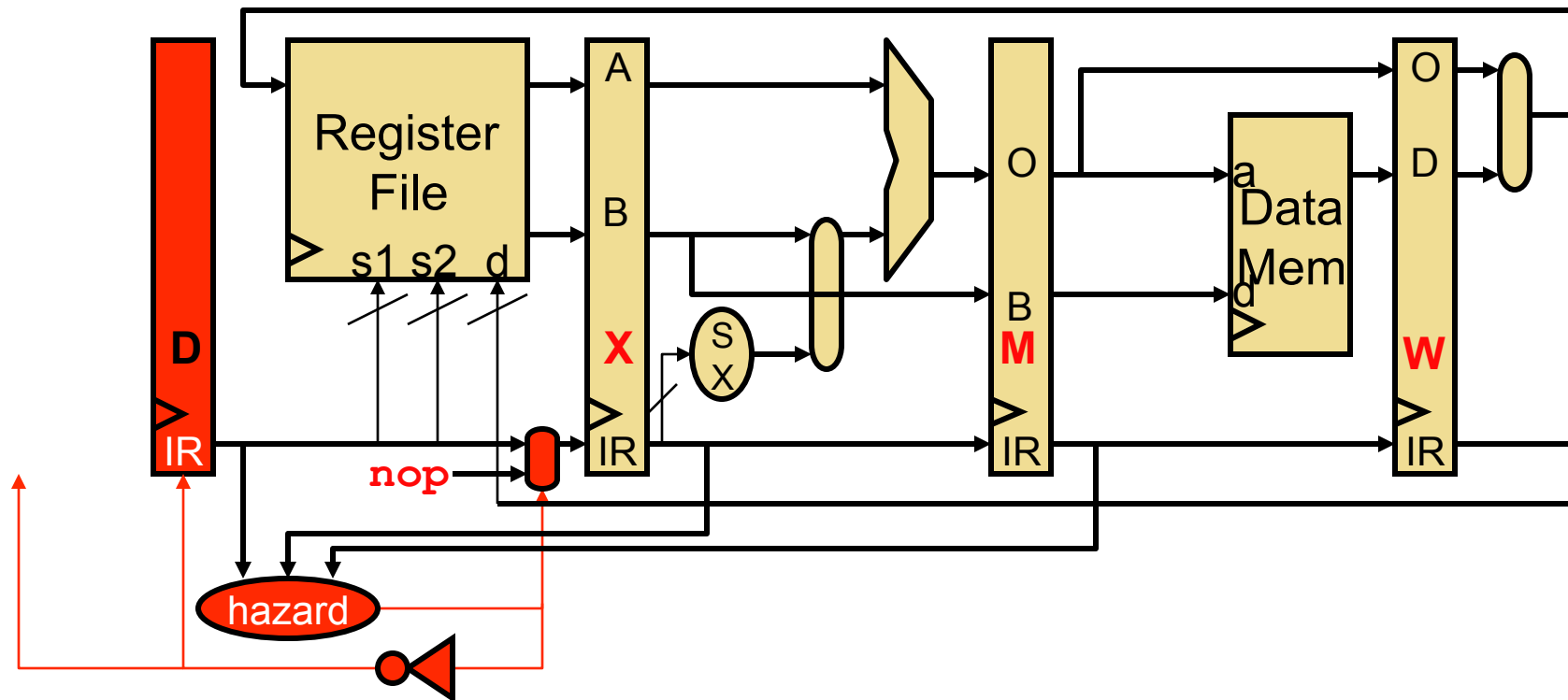
`lw $4, 0($3)`

`add $3, $2, $1`

Stall =

$(D.IR.RegSrc1 == X.IR.RegDest) ||$
 $(D.IR.RegSrc2 == X.IR.RegDest) ||$
 $(D.IR.RegSrc1 == M.IR.RegDest) ||$
 $(D.IR.RegSrc2 == M.IR.RegDest) = 1$

Hardware Interlock Example: cycle 2



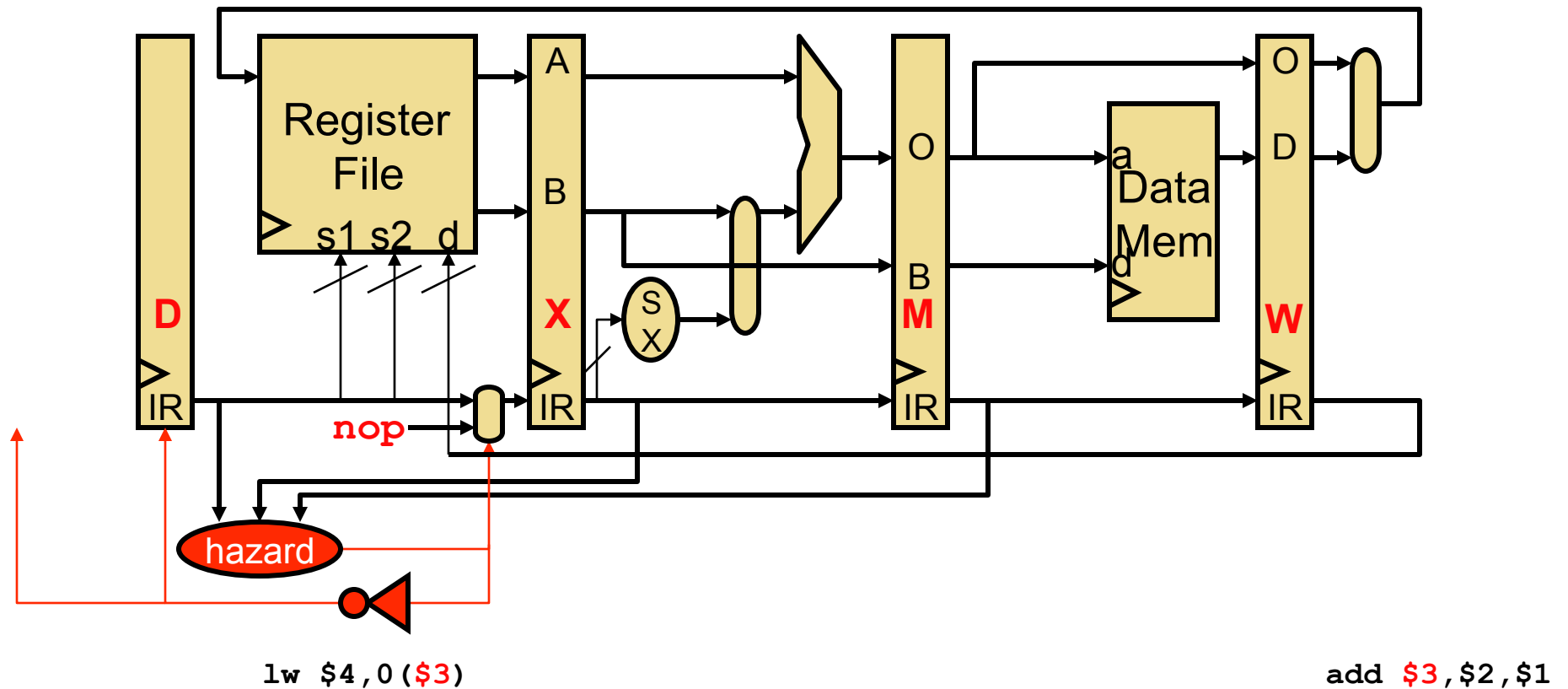
`lw $4, 0($3)`

`add $3, $2, $1`

Stall =

$$\begin{aligned}
 & (D.IR.RegSrc1 == X.IR.RegDest) \parallel \\
 & (D.IR.RegSrc2 == X.IR.RegDest) \parallel \\
 & (D.IR.RegSrc1 == M.IR.RegDest) \parallel \\
 & (\mathbf{D.IR.RegSrc2 == M.IR.RegDest}) = 1
 \end{aligned}$$

Hardware Interlock Example: cycle 3



Stall =

$$\begin{aligned}
 & (D.IR.RegSrc1 == X.IR.RegDest) \parallel \\
 & (D.IR.RegSrc2 == X.IR.RegDest) \parallel \\
 & (D.IR.RegSrc1 == M.IR.RegDest) \parallel \\
 & (D.IR.RegSrc2 == M.IR.RegDest) = 0
 \end{aligned}$$

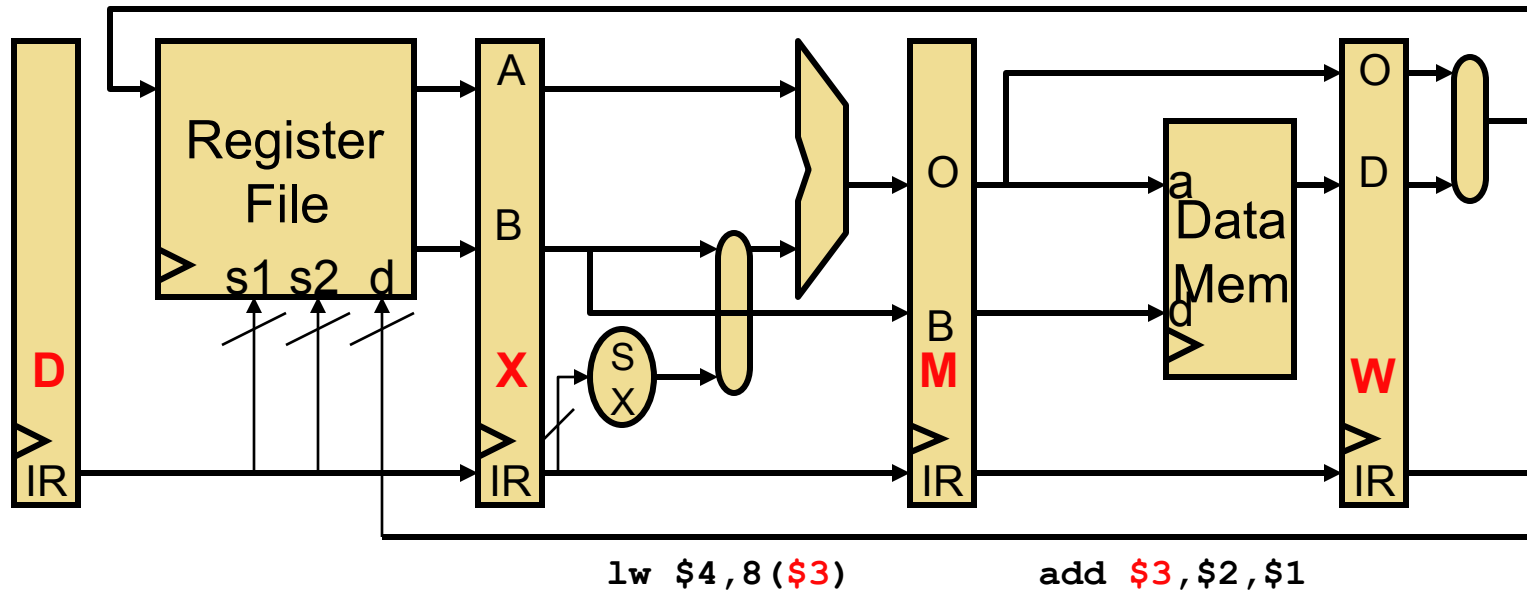
Pipeline Control Terminology

- 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
- Distinguish from **pipelined datapath control**
 - Controls datapath at each stage
 - Pipeline control controls advancement of datapath control

Hardware Interlock Performance

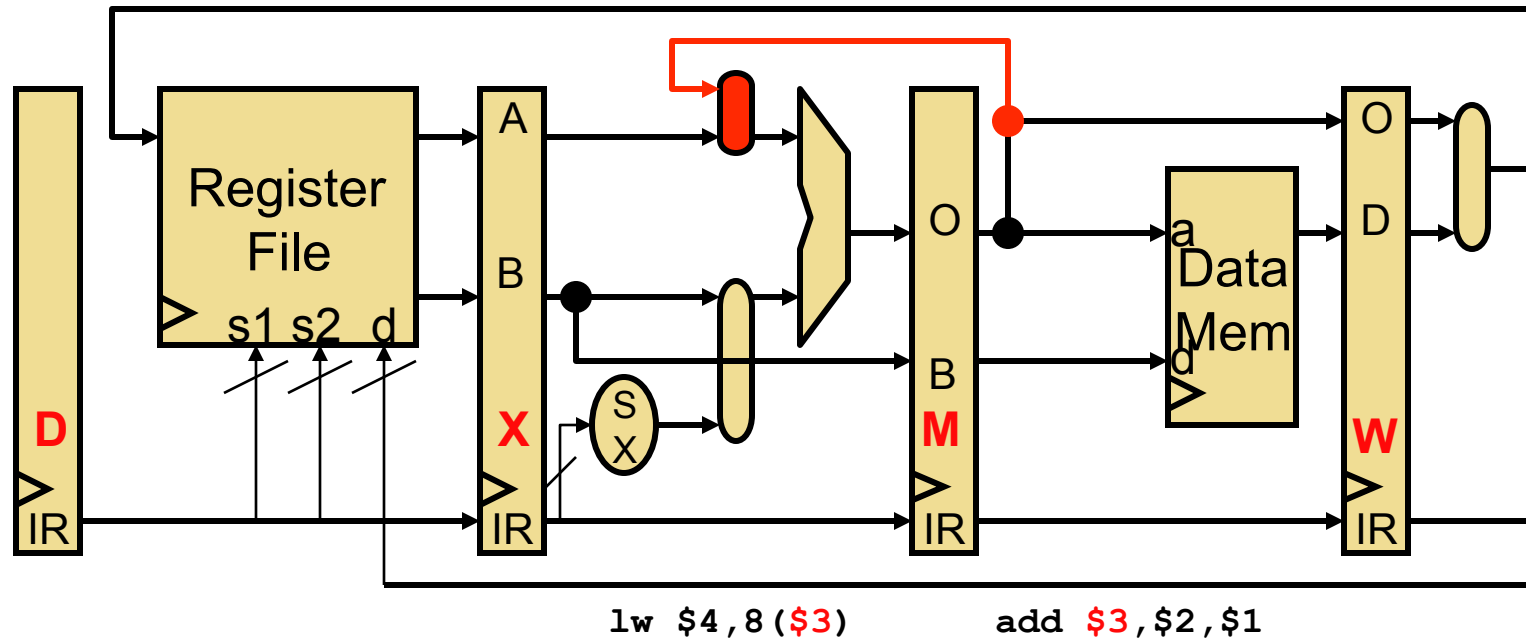
- As before:
 - Branch: 20%, load: 20%, store: 10%, other: 50%
- Hardware interlocks: same as software interlocks
 - 20% of insns require 1 cycle stall (I.e., insertion of 1 `nop`)
 - 5% of insns require 2 cycle stall (I.e., insertion of 2 `nops`)
 - $\text{CPI} = 1 + 0.20 \cdot 1 + 0.05 \cdot 2 = \mathbf{1.3}$
 - 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

Observation!



- Technically, this situation is broken
 - `lw $4, 8($3)` has already read `$3` from regfile
 - `add $3, $2, $1` hasn't yet written `$3` to regfile
- But fundamentally, everything is OK
 - `lw $4, 8($3)` hasn't actually used `$3` yet
 - `add $3, $2, $1` has already computed `$3`

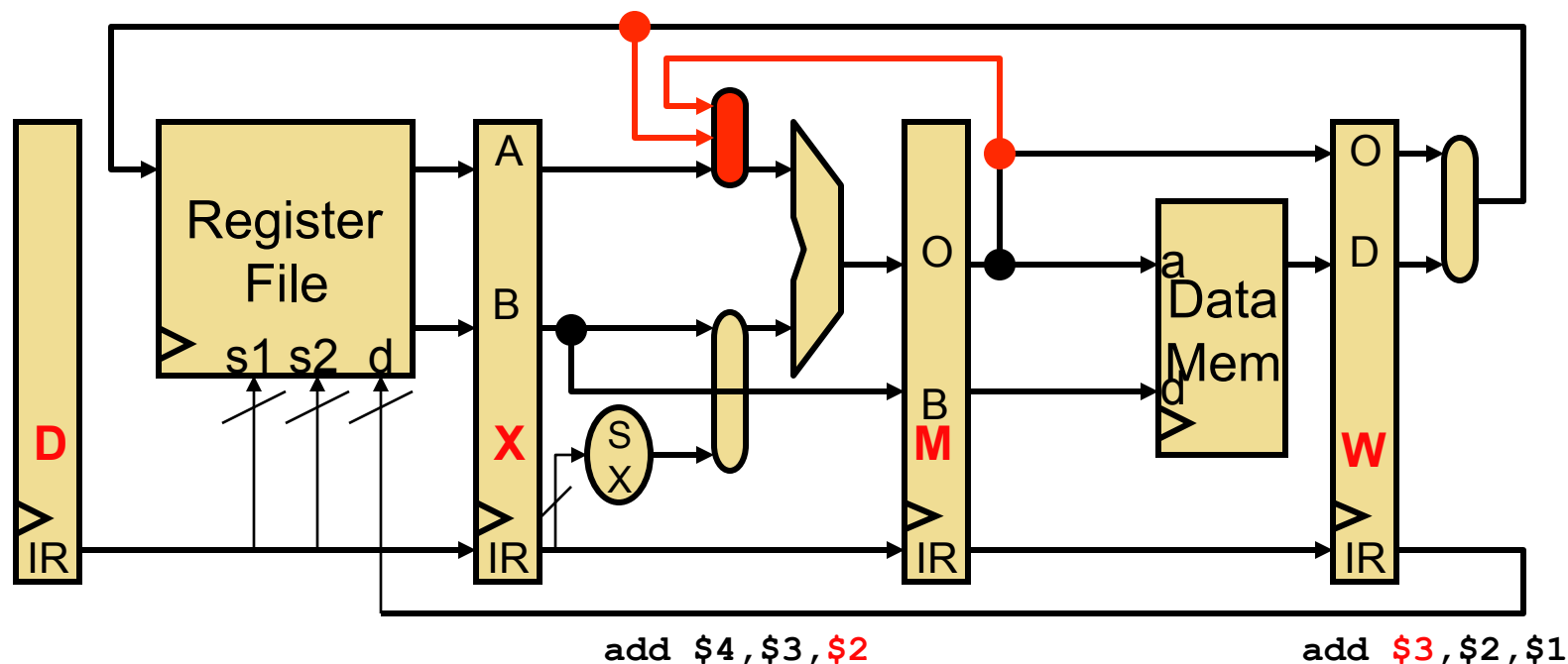
Bypassing



- **Bypassing**

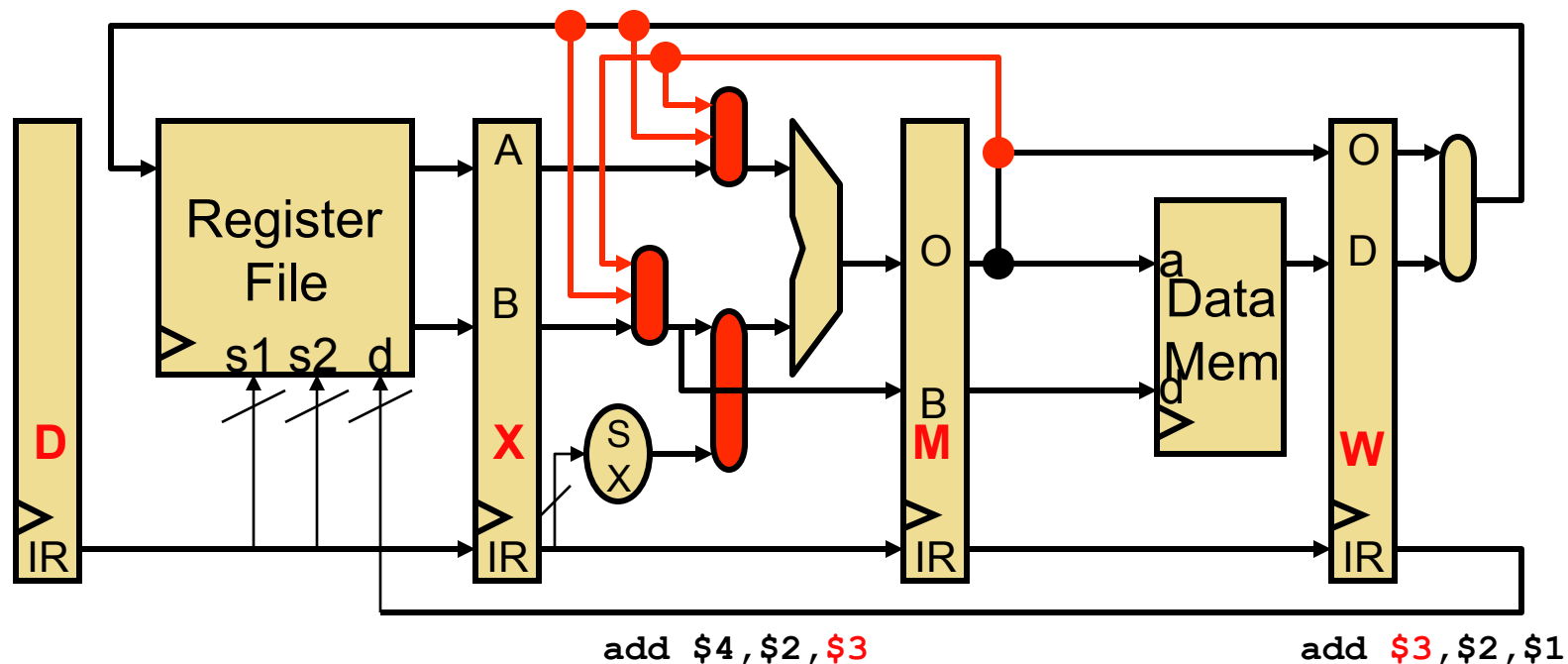
- Reading a value from an intermediate (μ architectural) source
- Not waiting until it is available from primary source
- Here, we are bypassing the register file
- Also called **forwarding**

WX Bypassing



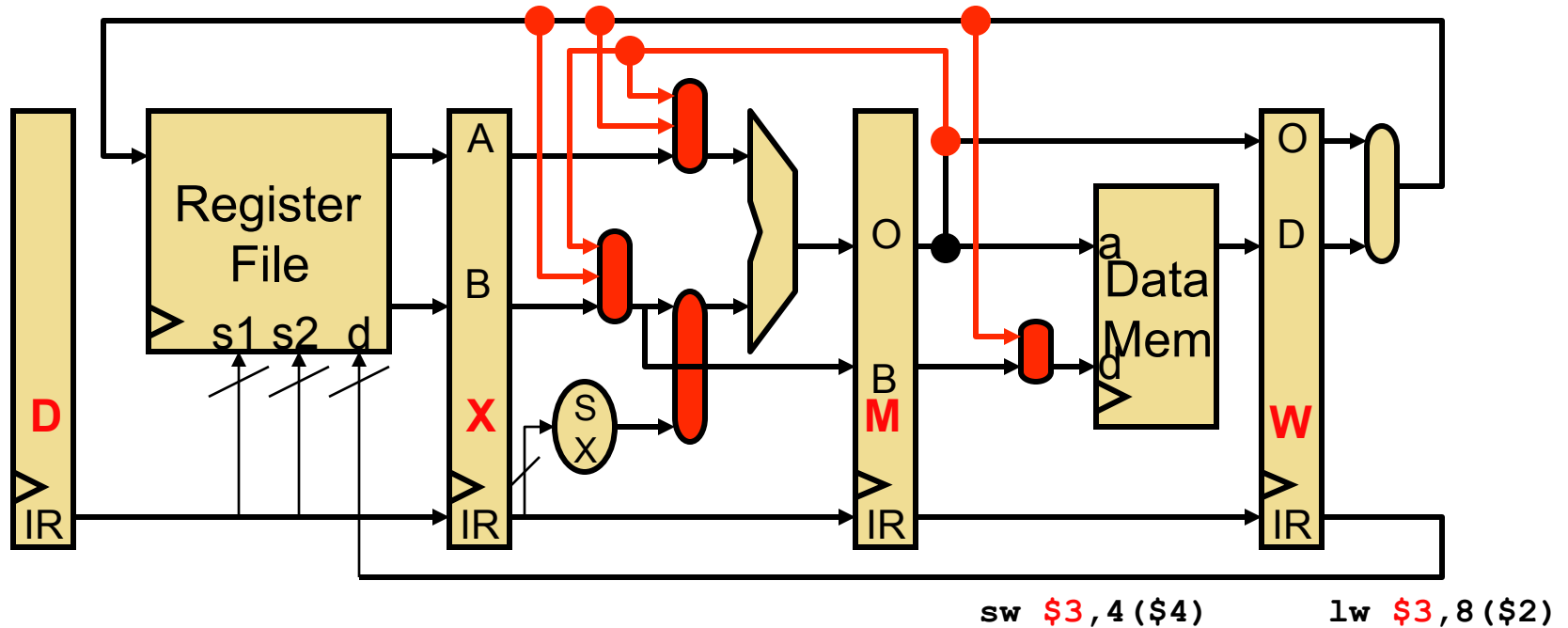
- What about this combination?
 - Add another bypass path and MUX (multiplexor) input
 - First one was an **MX** bypass
 - This one is a **WX** bypass

ALUinB Bypassing



- Can also bypass to ALU input B

WM Bypassing?



- Does WM bypassing make sense?

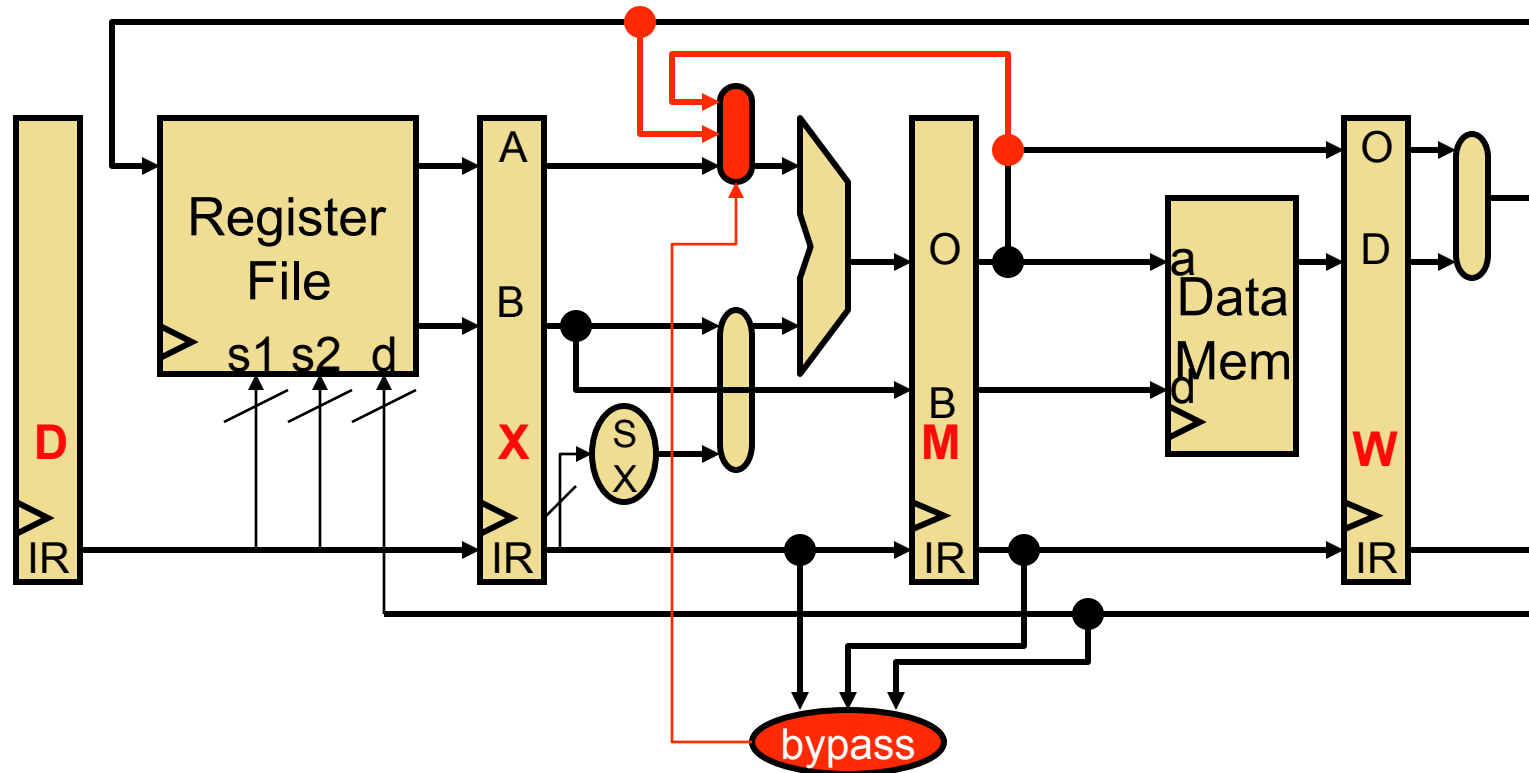
- Not to the address input (why not?)

`sw $4, 4 ($3)` `lw $3, 8 ($2)`

- But to the store data input, yes

`sw $3, 4 ($4)` `lw $3, 8 ($2)`

Bypass Logic



- Each multiplexor has its own, here it is for "ALUinA"
 $(X.IR.RegSrc1 == M.IR.RegDest) \Rightarrow 0$
 $(X.IR.RegSrc1 == W.IR.RegDest) \Rightarrow 1$
Else $\Rightarrow 2$

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 r2,r3→r1	F	D	X	M	W					
sub r1,r4→r2		F	D	X	M	W				

- Example: WX bypass

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

- Example: WM bypass

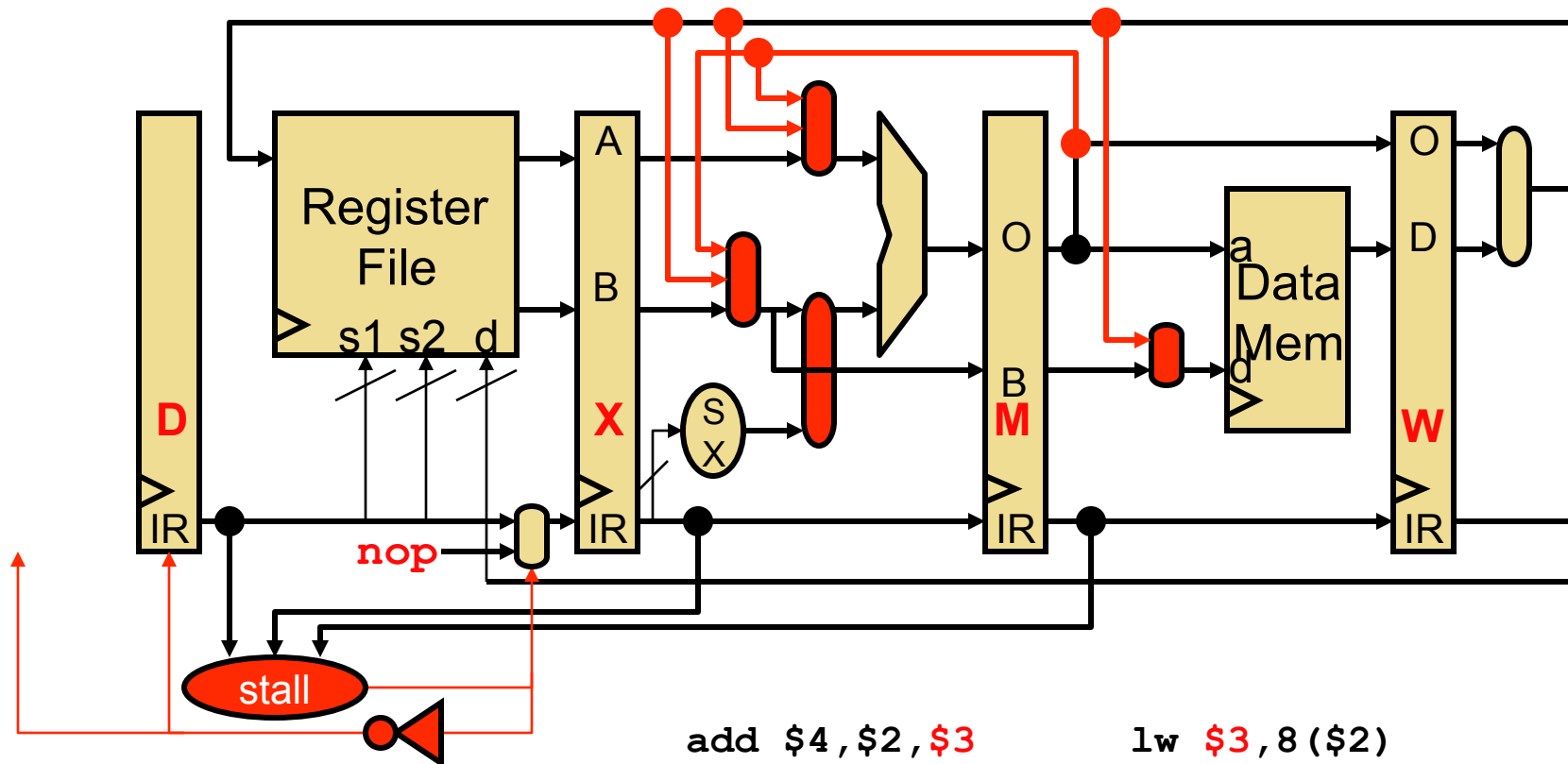
	1	2	3	4	5	6	7	8	9	10
add r2,r3→r1	F	D	X	M	W					
?		F	D	X	M	W				

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

Bypass and Stall Logic

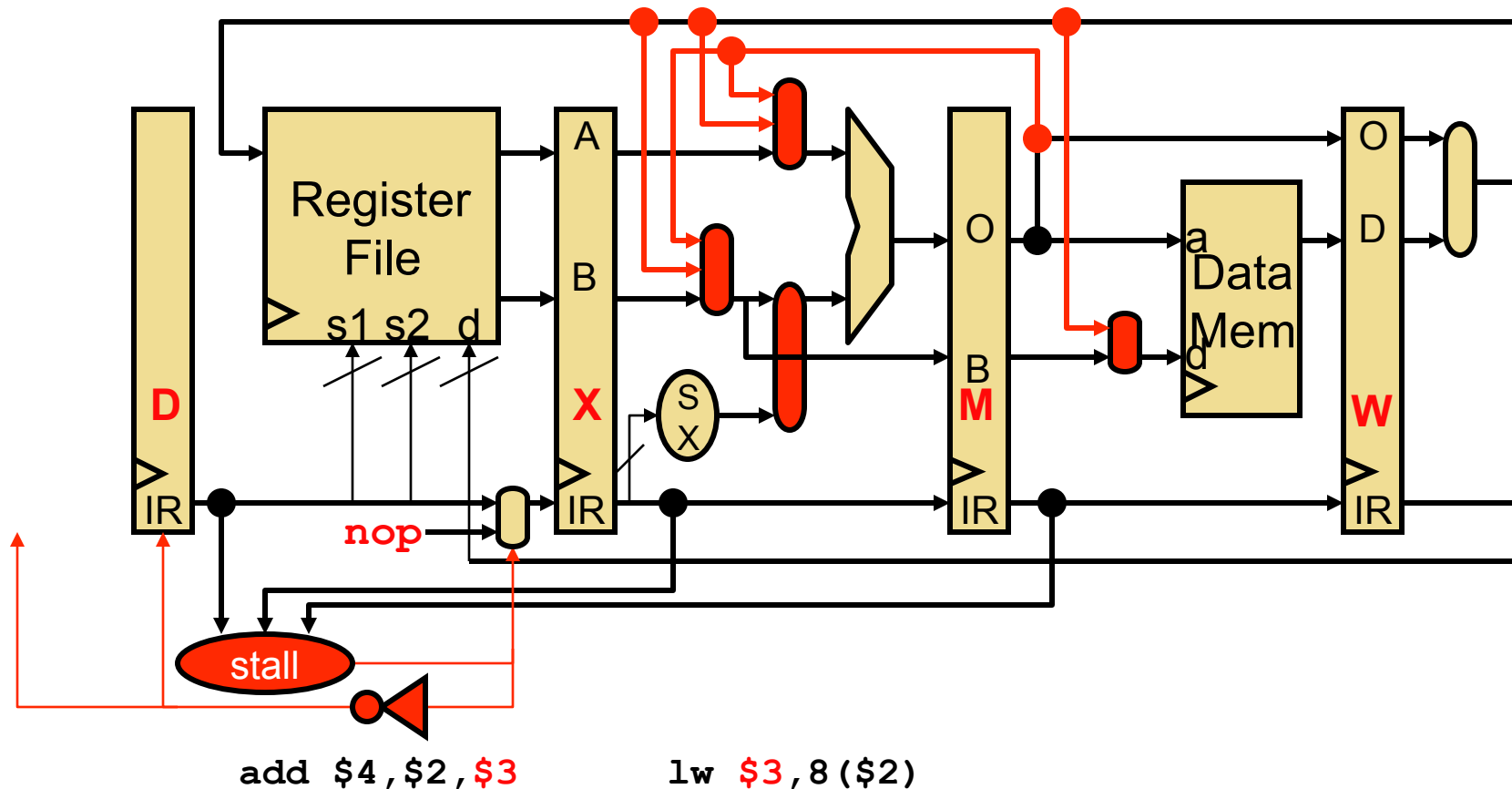
- Two separate things
 - Stall logic controls pipeline registers
 - Bypass logic controls multiplexors
- But complementary
 - For a given data hazard: if can't bypass, must stall
- Previous slide shows **full bypassing**: all bypasses possible
 - Have we prevented all data hazards? (Thus obviating stall logic)

Have We Prevented All Data Hazards?



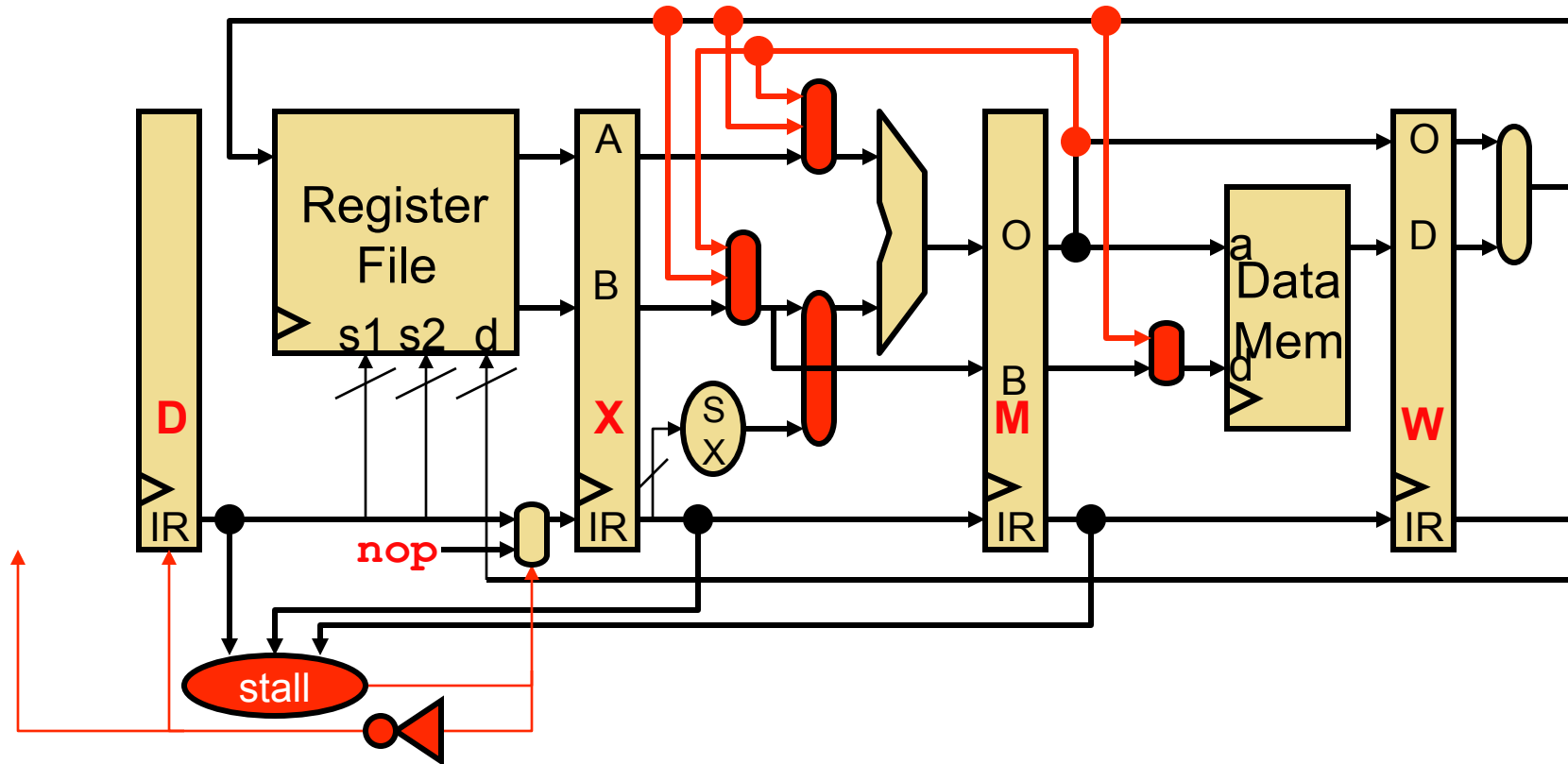
- No. Consider a "load" followed by a dependent "add" insn
- Bypassing alone isn't sufficient!
- **Hardware solution**: detect this situation and inject a stall cycle
- **Software solution**: ensure compiler doesn't generate such code

Stalling on Load-To-Use Dependences



- Prevent "D insn" from advancing this cycle
 - Write **nop** into X.IR (effectively, insert **nop** in hardware)
 - Keep same "D insn", same PC next cycle
- Re-evaluate situation next cycle

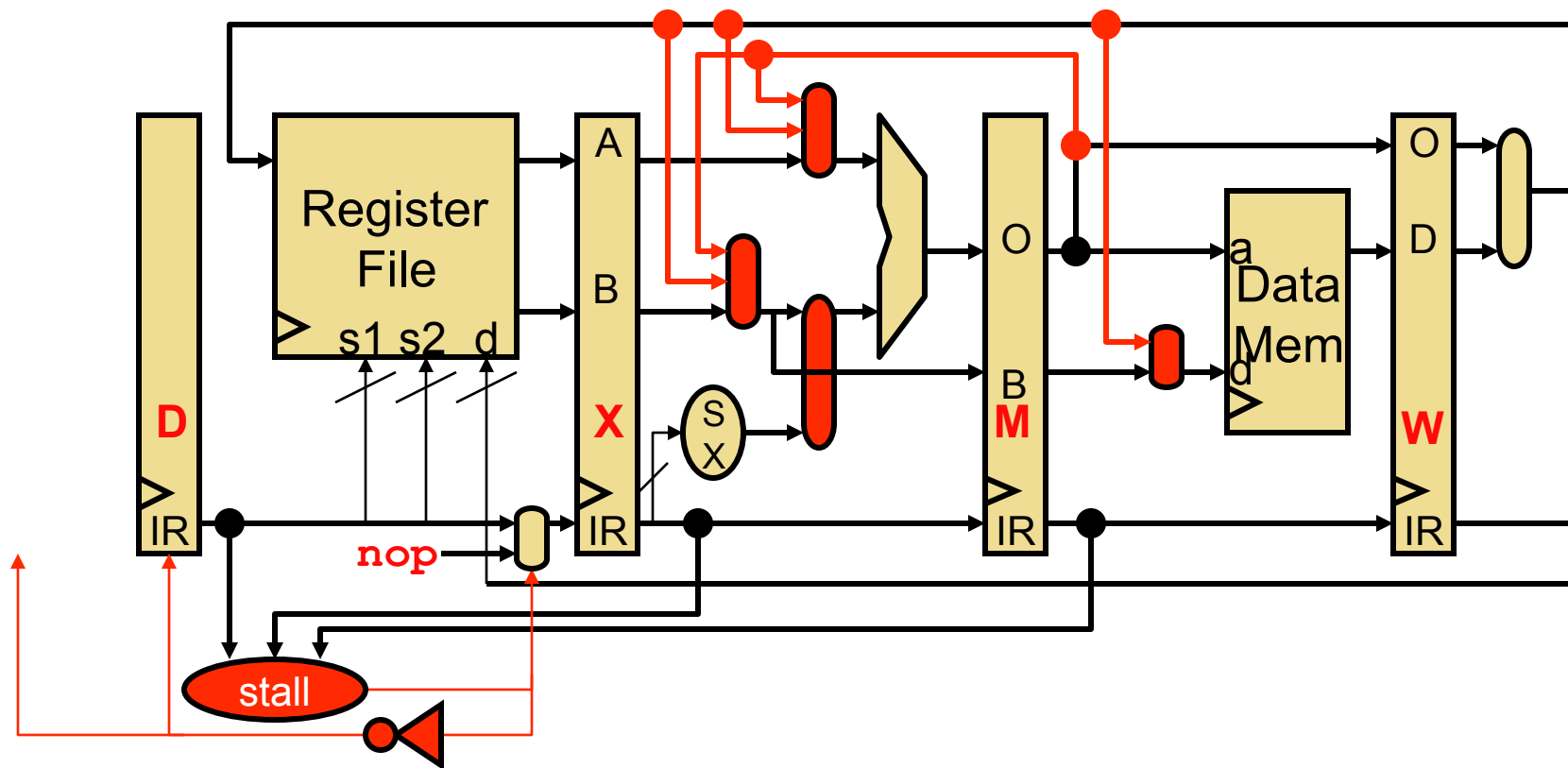
Stalling on Load-To-Use Dependences



add \$4, \$2, \$3 1w \$3, 8(\$2)

```
Stall = (X.IR.Operation == LOAD) &&
        ( (D.IR.RegSrc1 == X.IR.RegDest) ||
          ((D.IR.RegSrc2 == X.IR.RegDest) && (D.IR.Op != STORE))
        )
```

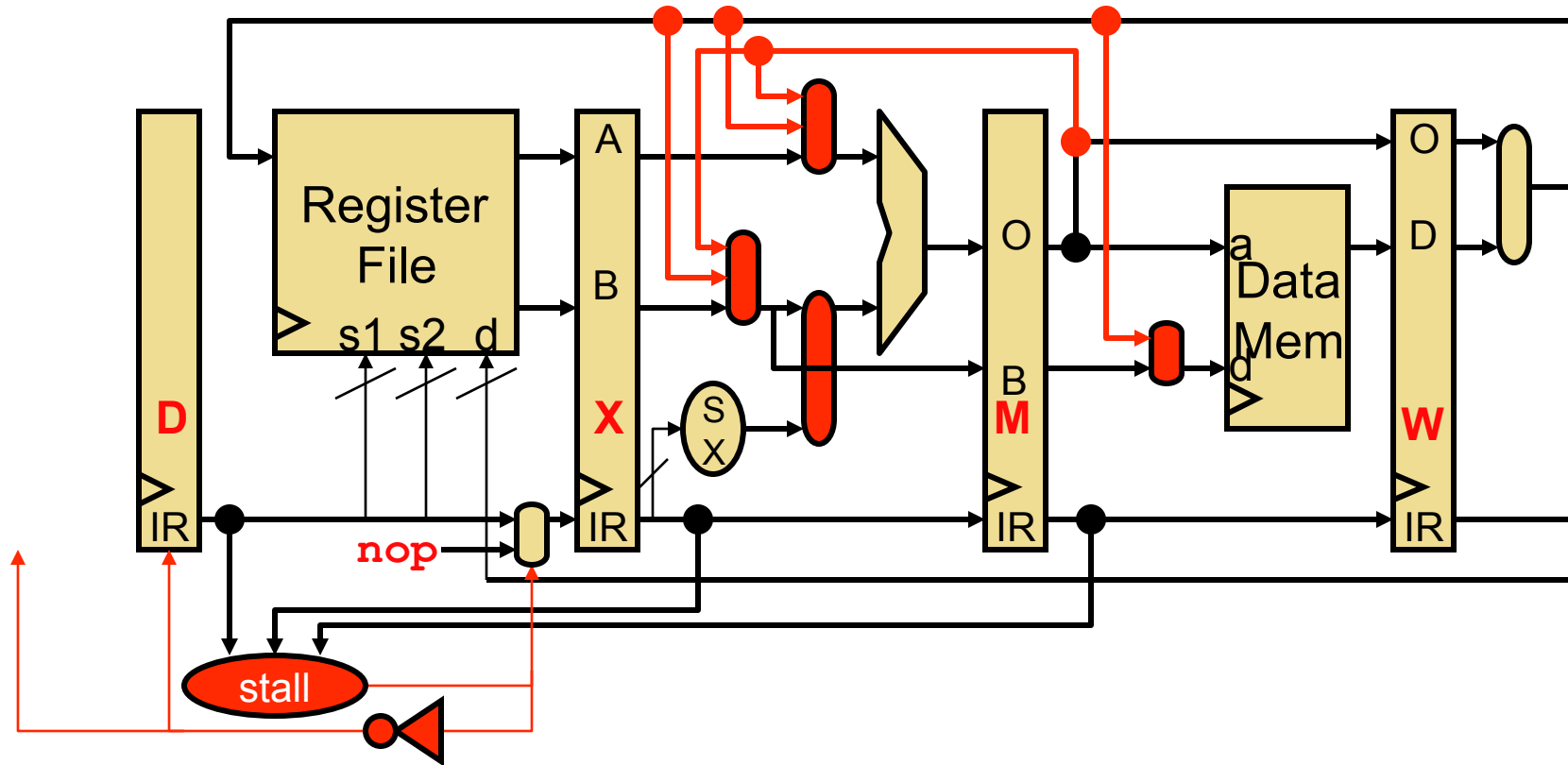
Stalling on Load-To-Use Dependences



add \$4, \$2, \$3 (stall bubble) lw \$3, 8(\$2)

```
Stall = (X.IR.Operation == LOAD) &&
        ( (D.IR.RegSrc1 == X.IR.RegDest) ||
          ((D.IR.RegSrc2 == X.IR.RegDest) && (D.IR.Op != STORE))
        )
```

Stalling on Load-To-Use Dependences



add \$4, \$2, \$3 (stall bubble) lw \$3, ...

```
Stall = (X.IR.Operation == LOAD) &&
        ( (D.IR.RegSrc1 == X.IR.RegDest) ||
          ((D.IR.RegSrc2 == X.IR.RegDest) && (D.IR.Op != STORE))
        )
```

Performance Impact of Load/Use Penalty

- Assume
 - Branch: 20%, load: 20%, store: 10%, other: 50%
 - 50% of loads are followed by dependent instruction
 - require 1 cycle stall (I.e., insertion of 1 `nop`)
- Calculate CPI
 - $\text{CPI} = 1 + (1 * 20\% * 50\%) = \mathbf{1.1}$

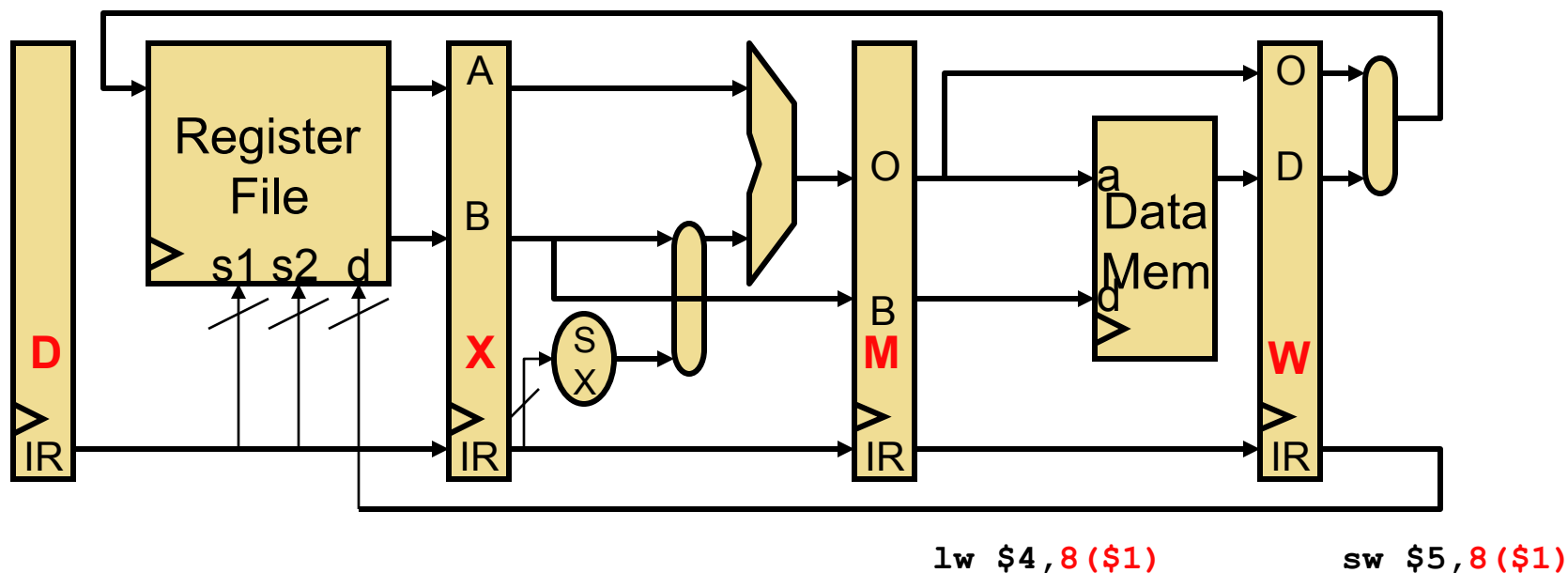
Reducing Load-Use Stall Frequency

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

- Use compiler scheduling to reduce load-use stall frequency
 - As done for software interlocks, but for performance not correctness

	1	2	3	4	5	6	7	8	9
add \$3, \$2, \$1	F	D	X	M	W				
lw \$4, 4(\$3)		F	D	X	M	W			
sub \$8, \$3, \$1			F	D	X	M	W		
addi \$6, \$4, 1				F	D	X	M	W	

Dependencies Through Memory

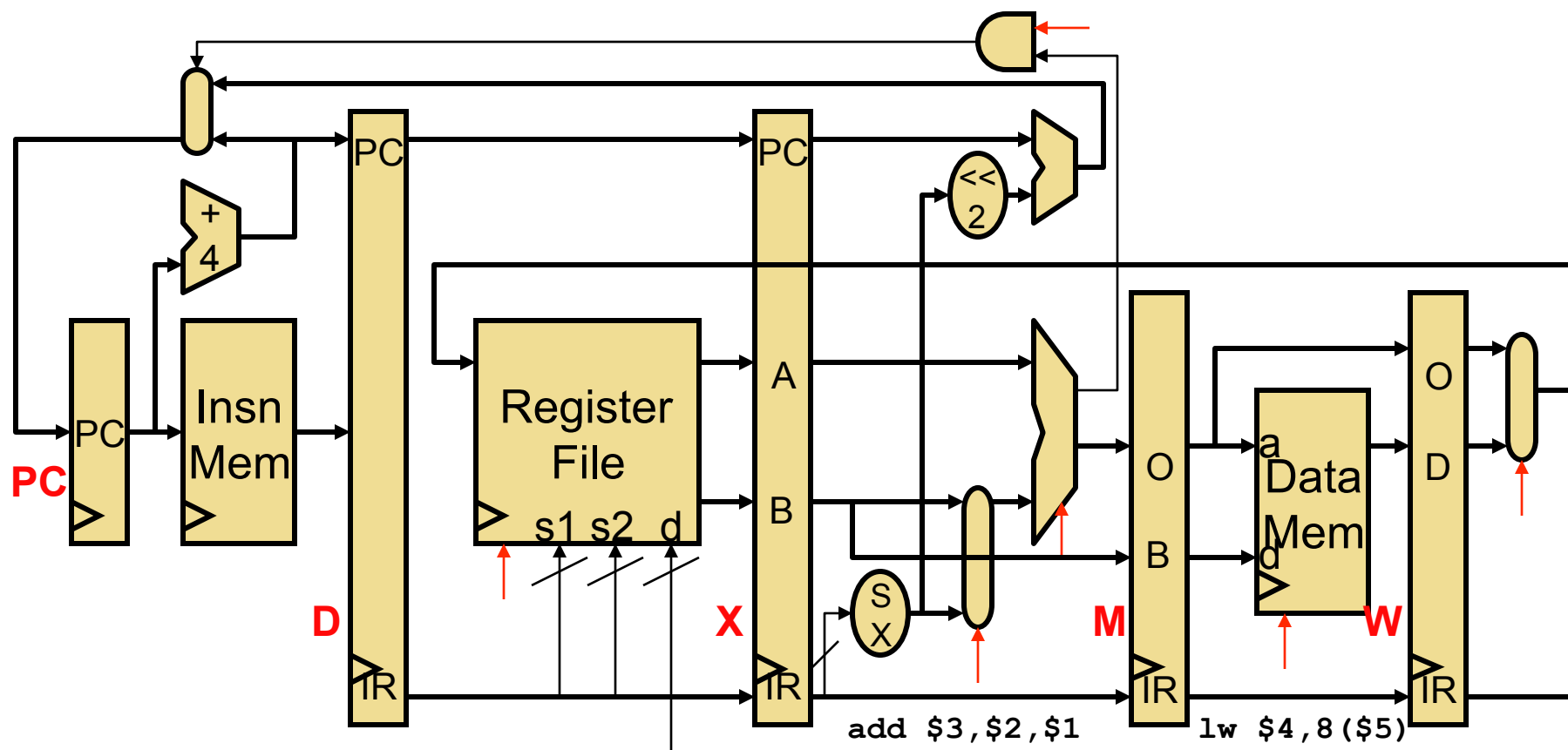


- Are “load to store” memory dependencies a problem? No
 - `lw` following `sw` to same address in next cycle, gets right value
 - Why? Data mem read/write always take place in same stage
- Are there any other sort of hazards to worry about?

Structural Hazards

- **Structural hazards**
 - Two insns trying to use same circuit at same time
 - E.g., structural hazard on register file write port
- **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
- **Tolerate structure hazards**
 - Add stall logic to stall pipeline when hazards occur

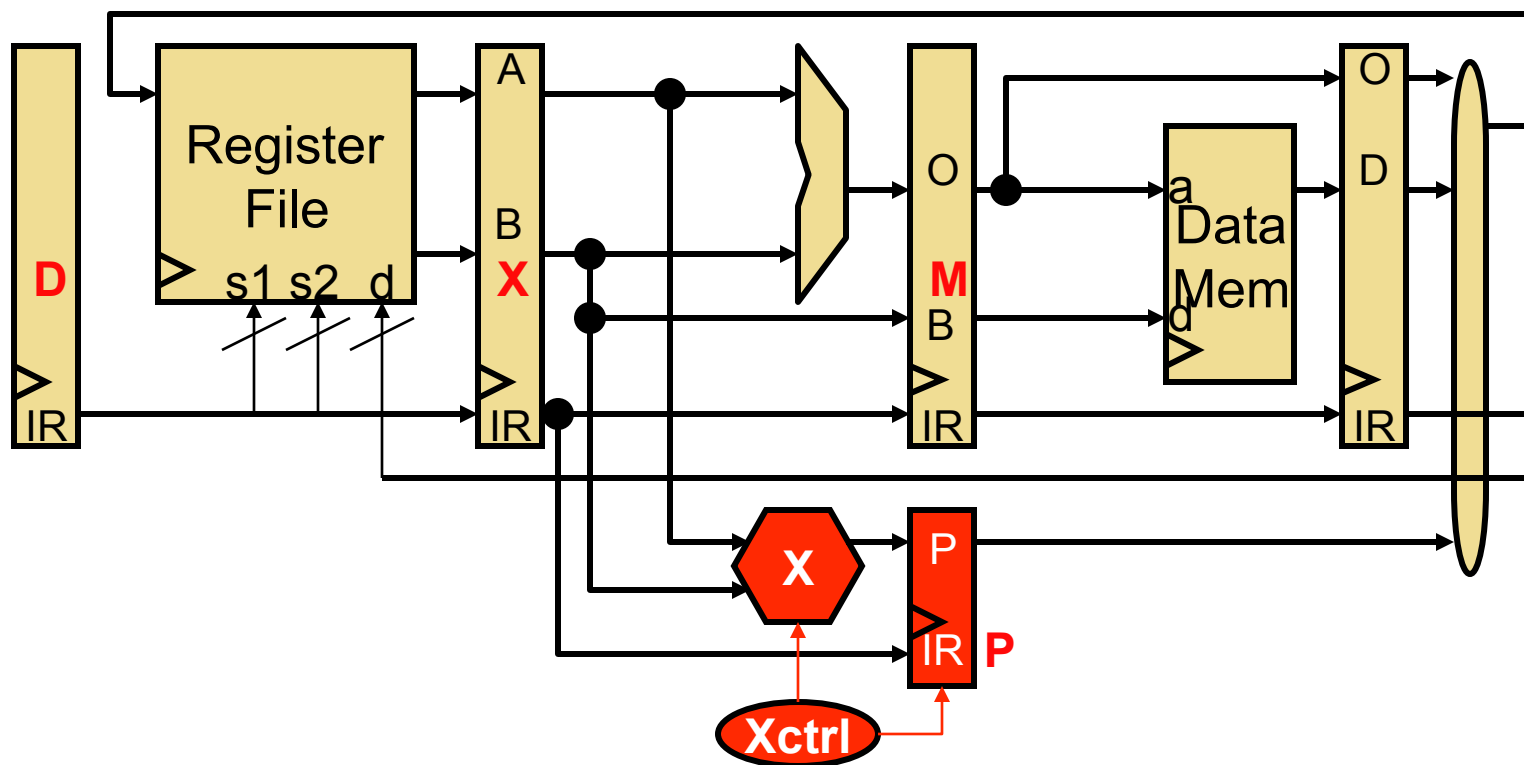
Why Does Every Insn Take 5 Cycles?



- 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` after `lw` (only 1 reg. write port)

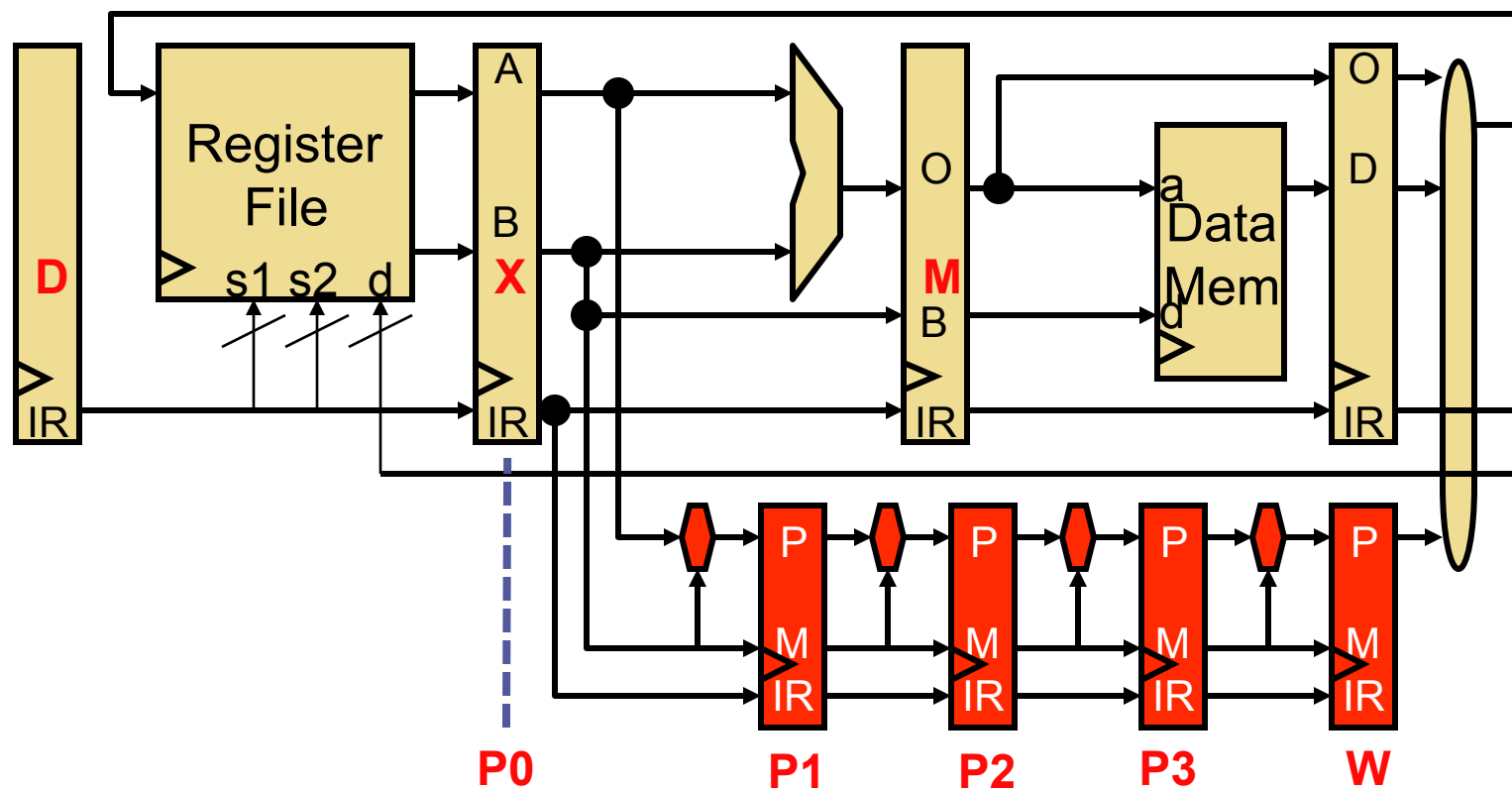
Multi-Cycle Operations

Pipelining and Multi-Cycle Operations



- What if you wanted to add a multi-cycle operation?
 - 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			

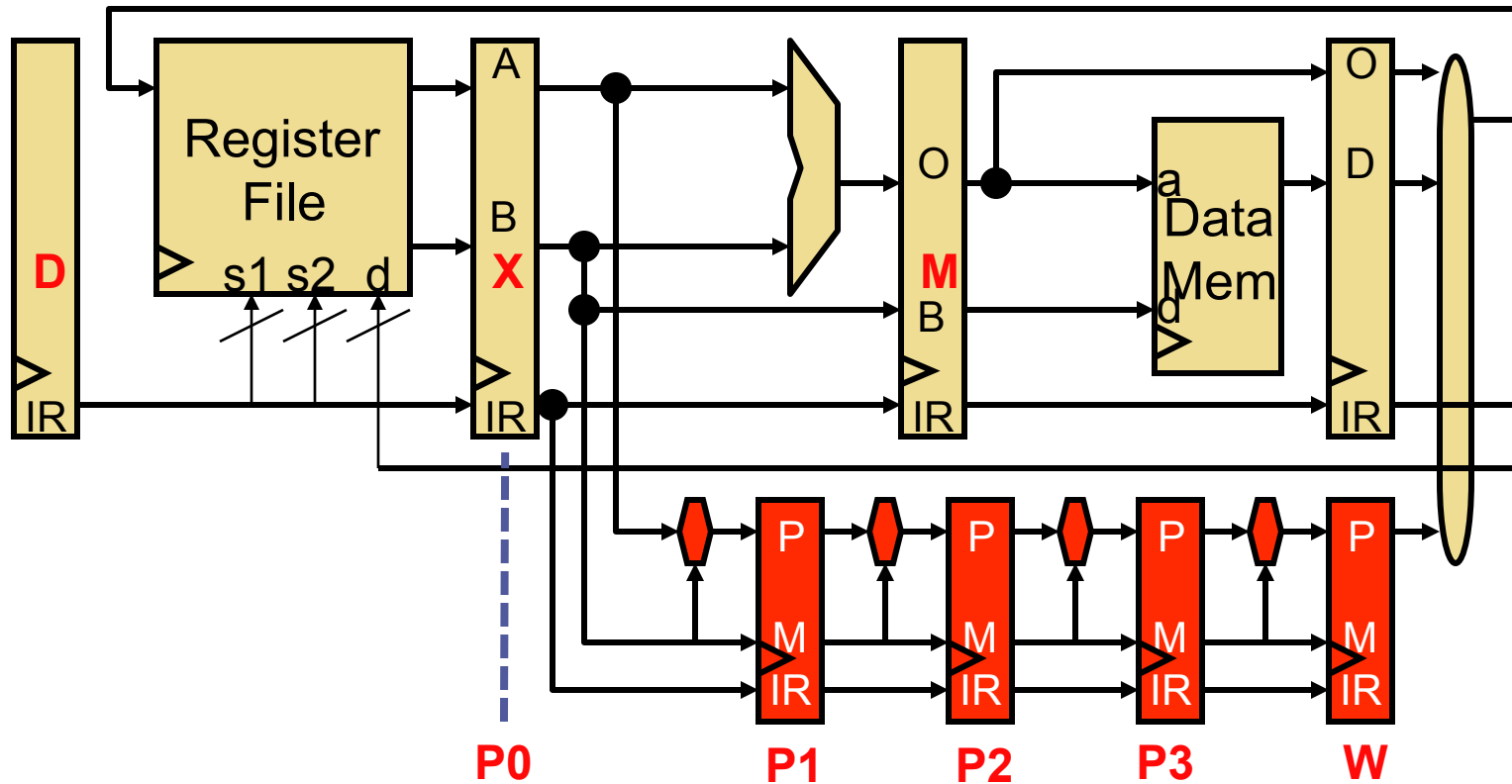
- Even allow **independent multiply** instructions

	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:

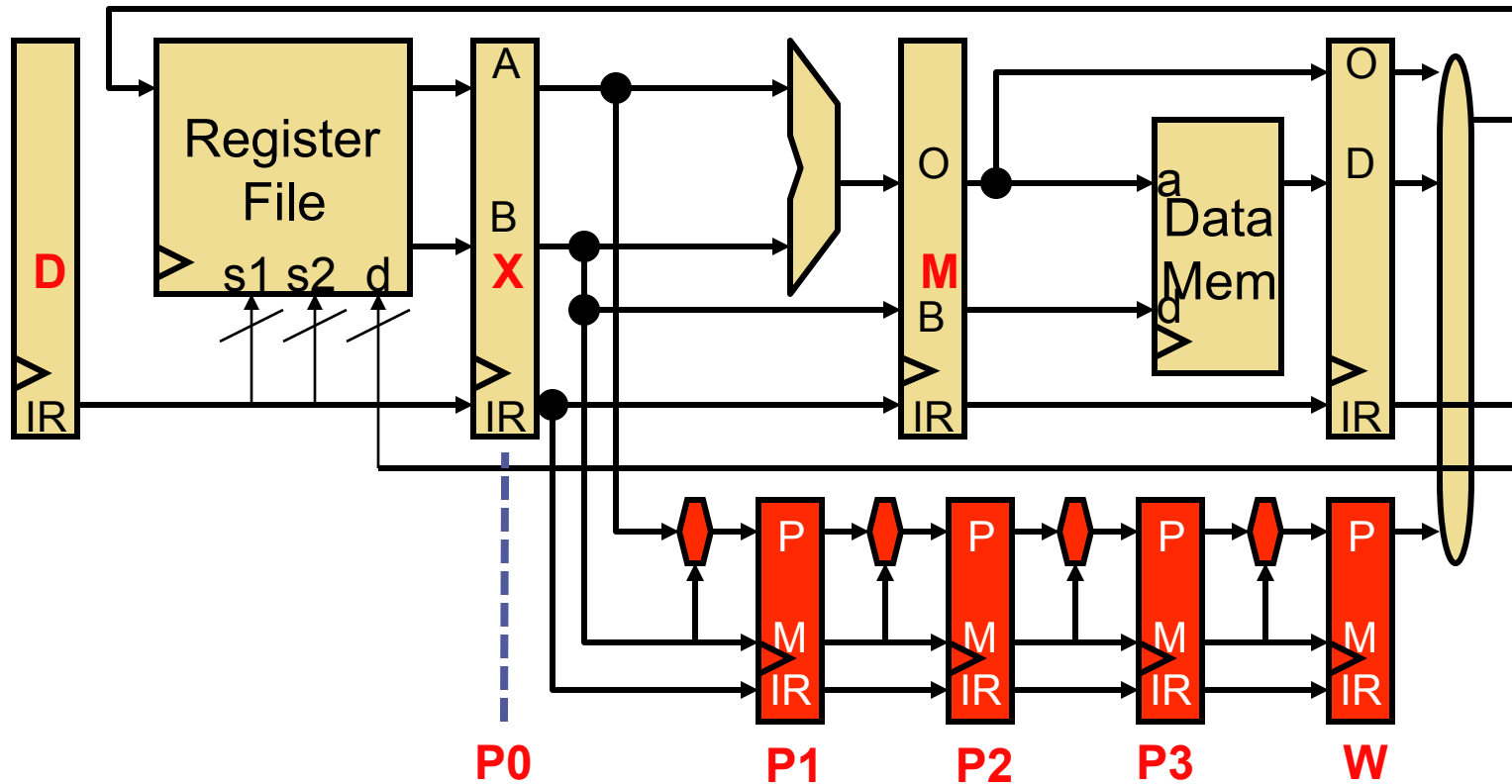
	1	2	3	4	5	6	7	8	9
mul \$4 , \$3, \$5	F	D	P0	P1	P2	P3	W		
addi \$6, \$4 , 1		F	D	d*	d*	d*	X	M	W

What about Stall Logic?



	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

What about Stall Logic?



```

Stall = (OldStallLogic) ||
(D.IR.RegSrc1 == P0.IR.RegDest) || (D.IR.RegSrc2 == P0.IR.RegDest) ||
(D.IR.RegSrc1 == P1.IR.RegDest) || (D.IR.RegSrc2 == P1.IR.RegDest) ||
(D.IR.RegSrc1 == P2.IR.RegDest) || (D.IR.RegSrc2 == P2.IR.RegDest)
    
```

Multiplier Write Port Structural Hazard

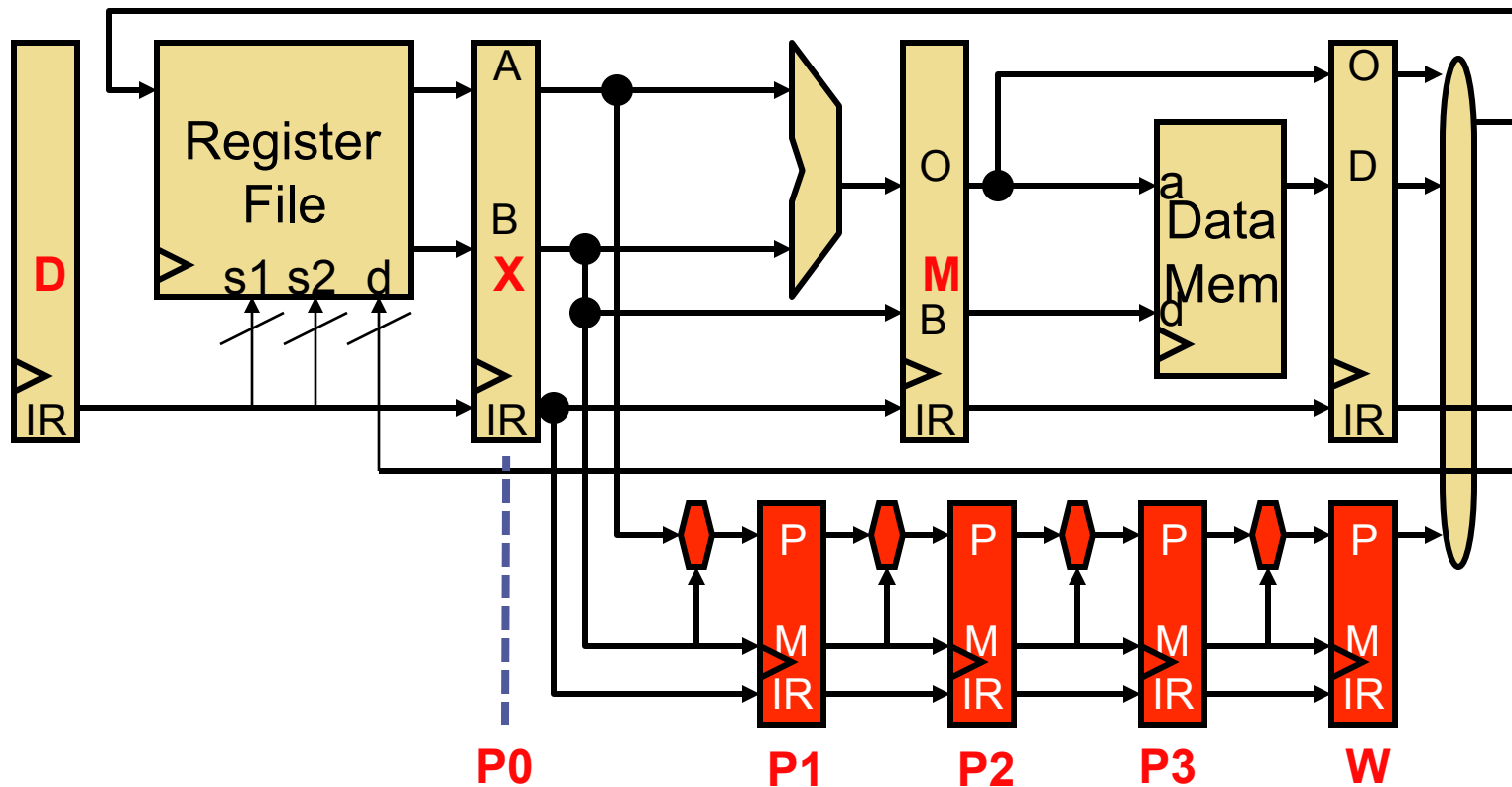
- What about...
 - Two instructions trying to write register file in same cycle?
 - Structural hazard!
- Must prevent:

	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	

Preventing Structural Hazard



- Fix to problem on previous slide:

Stall = (OldStallLogic) ||

(**D.IR.RegDest "is valid" &&**

D.IR.Operation != MULT && P1.IR.RegDest "is valid")

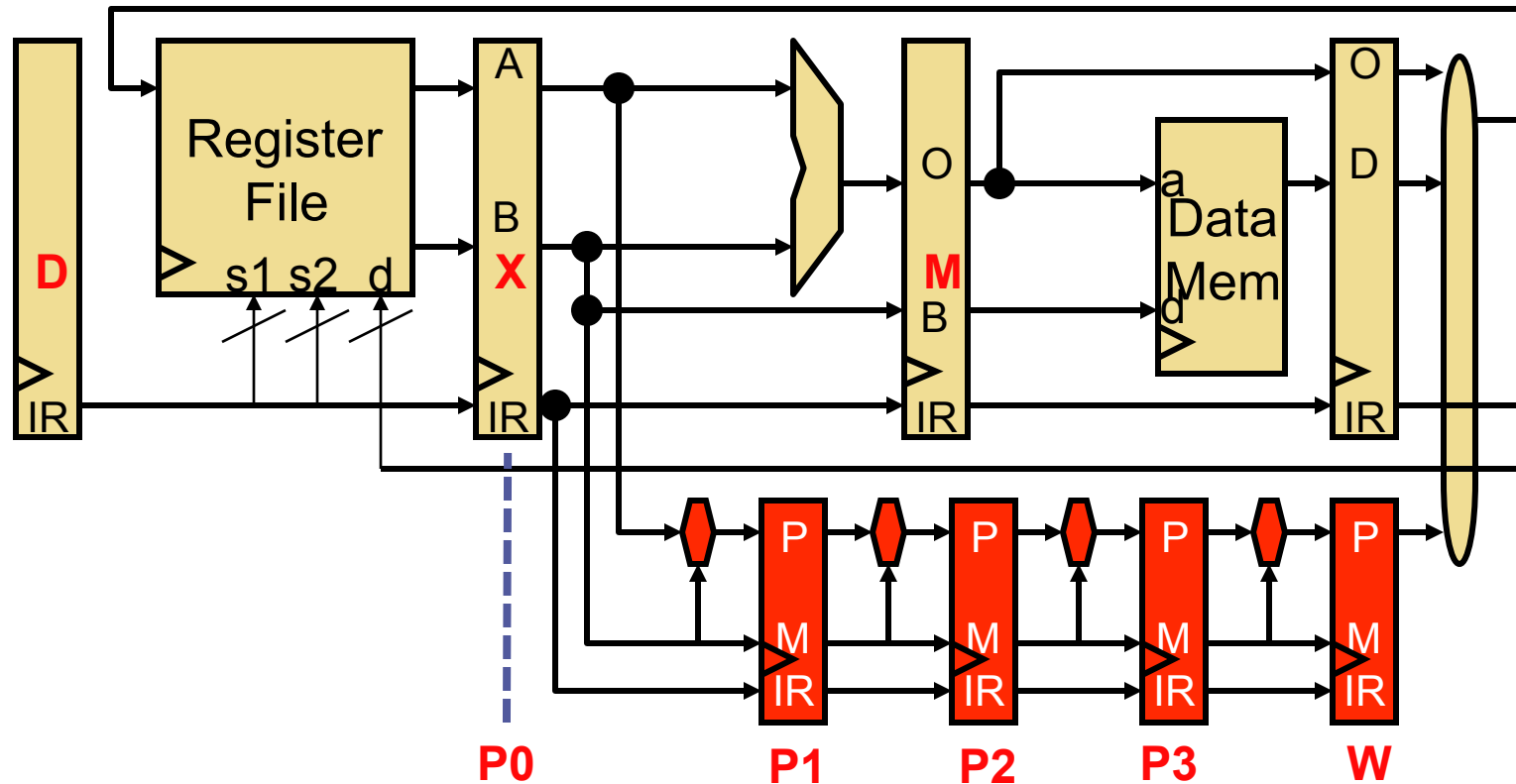
More Multiplier Nasties

- 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
<code>mul \$4, \$3, \$5</code>	F	D	P0	P1	P2	P3	W		
<code>addi \$4, \$1, 1</code>		F	D	X	M	W			
...									
...									
<code>add \$10, \$4, \$6</code>					F	D	X	M	W

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

Preventing Mis-Ordered Reg. Write



- Fix to problem on previous slide:

Stall = (OldStallLogic) ||

((D.IR.RegDest == X.IR.RegDest) && (X.IR.Operation == MULT))

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					F	D	X	M	W

- **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 add bandwidth than multiple non-pipelined units

	1	2	3	4	5	6	7	8	9	10	11
<code>mul f0, f1, f2</code>	F	D	E*	E*	E*	E*	W				
<code>mul f3, f4, f5</code>		F	D	E*	E*	E*	E*	W			

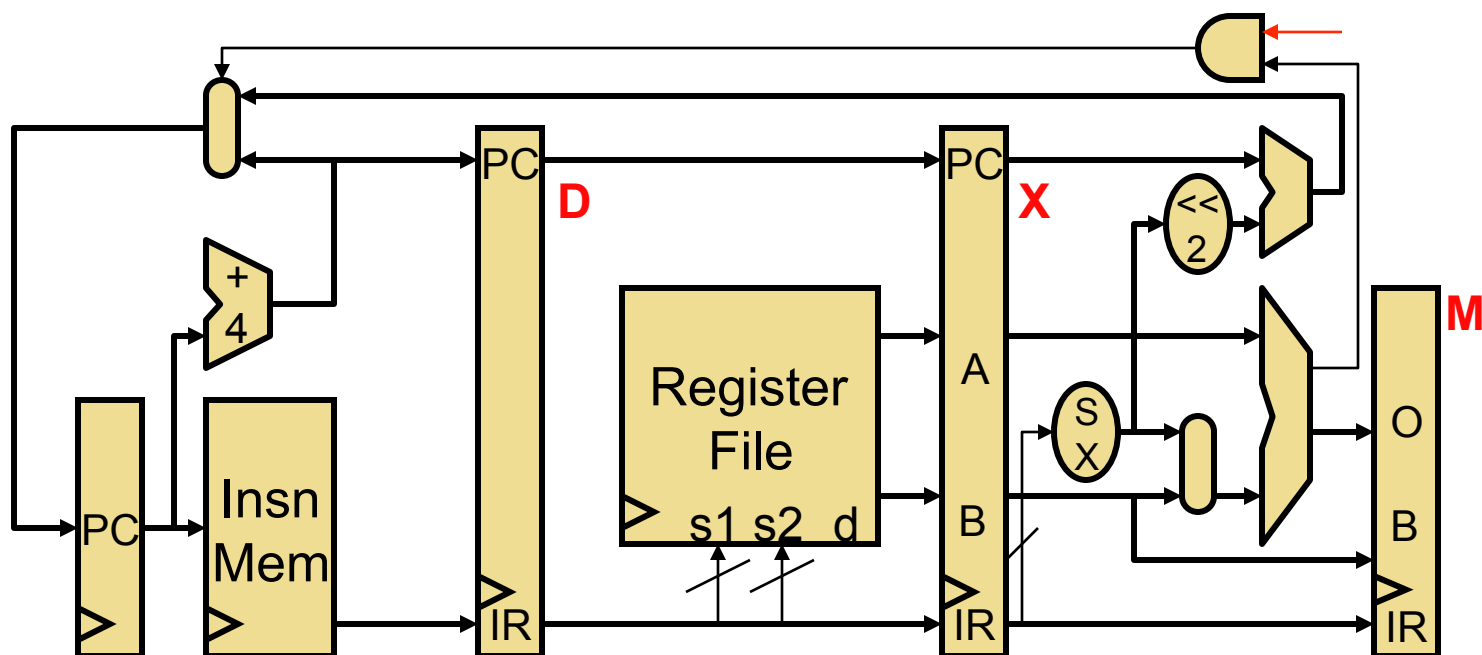
- One exception: int/FP divide: difficult to pipeline and not worth it

	1	2	3	4	5	6	7	8	9	10	11
<code>div f0, f1, f2</code>	F	D	E/	E/	E/	E/	W				
<code>div f3, f4, f5</code>		F	D	s*	s*	s*	E/	E/	E/	E/	W

- **s*** = structural hazard, two insns need same structure
 - ISAs and pipelines designed to have few of these
 - Canonical example: all insns forced to go through M stage

Control Dependences and Branch Prediction

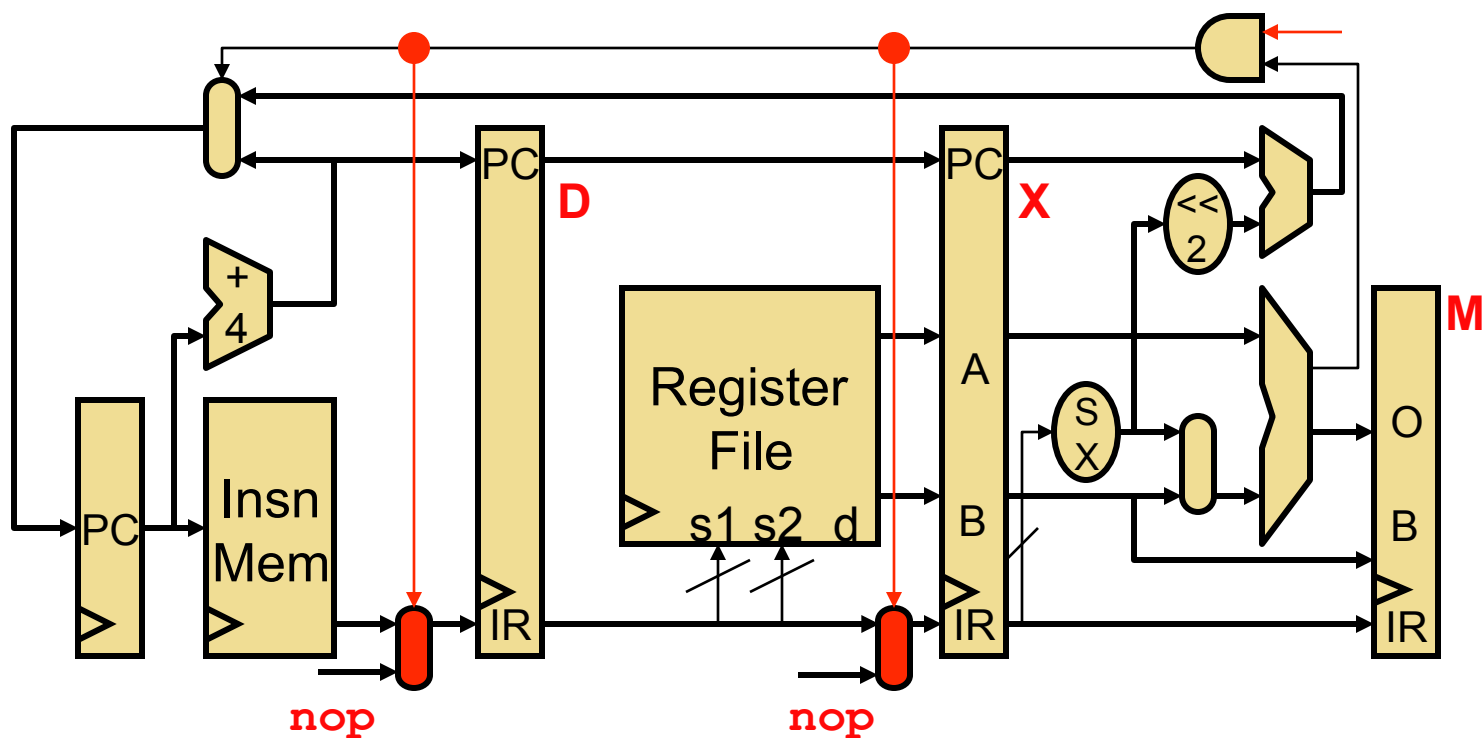
What About Branches?



- **Branch speculation**

- Could just stall to wait for branch outcome (two-cycle penalty)
- **Fetch past 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 actually taken
 - Insns that will be written into D and X are wrong
 - **Flush them**, i.e., replace them with **nops**
 - + They haven't had written permanent state yet (regfile, DMem)
 - Two cycle penalty for taken branches

Branch Speculation and Recovery

Correct:

	1	2	3	4	5	6	7	8	9
<code>addi r1,1→r3</code>	F	D	X	M	W				
<code>bnez r3,targ</code>		F	D	X	M	W			
<code>st r6→[r7+4]</code>			F	D	X	M	W		
<code>mul r8,r9→r10</code>				F	D	X	M	W	

speculative

- **Mis-speculation recovery:** what to do on wrong guess
 - Not too painful in an short, in-order pipeline
 - Branch resolves in X
 - + Younger insns (in F, D) haven't changed permanent state
 - **Flush** insns currently in D and X (i.e., replace with **nops**)

Recovery:

	1	2	3	4	5	6	7	8	9
<code>addi r1,1→r3</code>	F	D	X	M	W				
<code>bnez r3,targ</code>		F	D	X	M	W			
<code>st r6→[r7+4]</code>			F	D	--	--	--		
<code>mul r8,r9→r10</code>				F	--	--	--	--	
<code>targ:add r4,r5→r4</code>					F	D	X	M	W

Branch Performance

- Back of the envelope calculation
 - **Branch: 20%**, load: 20%, store: 10%, other: 50%
 - Say, **75% of branches are taken**
- $\text{CPI} = 1 + 20\% * 75\% * 2 =$
 $1 + \mathbf{0.20 * 0.75 * 2} = 1.3$
 - **Branches cause 30% slowdown**
 - Worse with deeper pipelines (higher mis-prediction penalty)
- Can we do better than assuming branch is not taken?

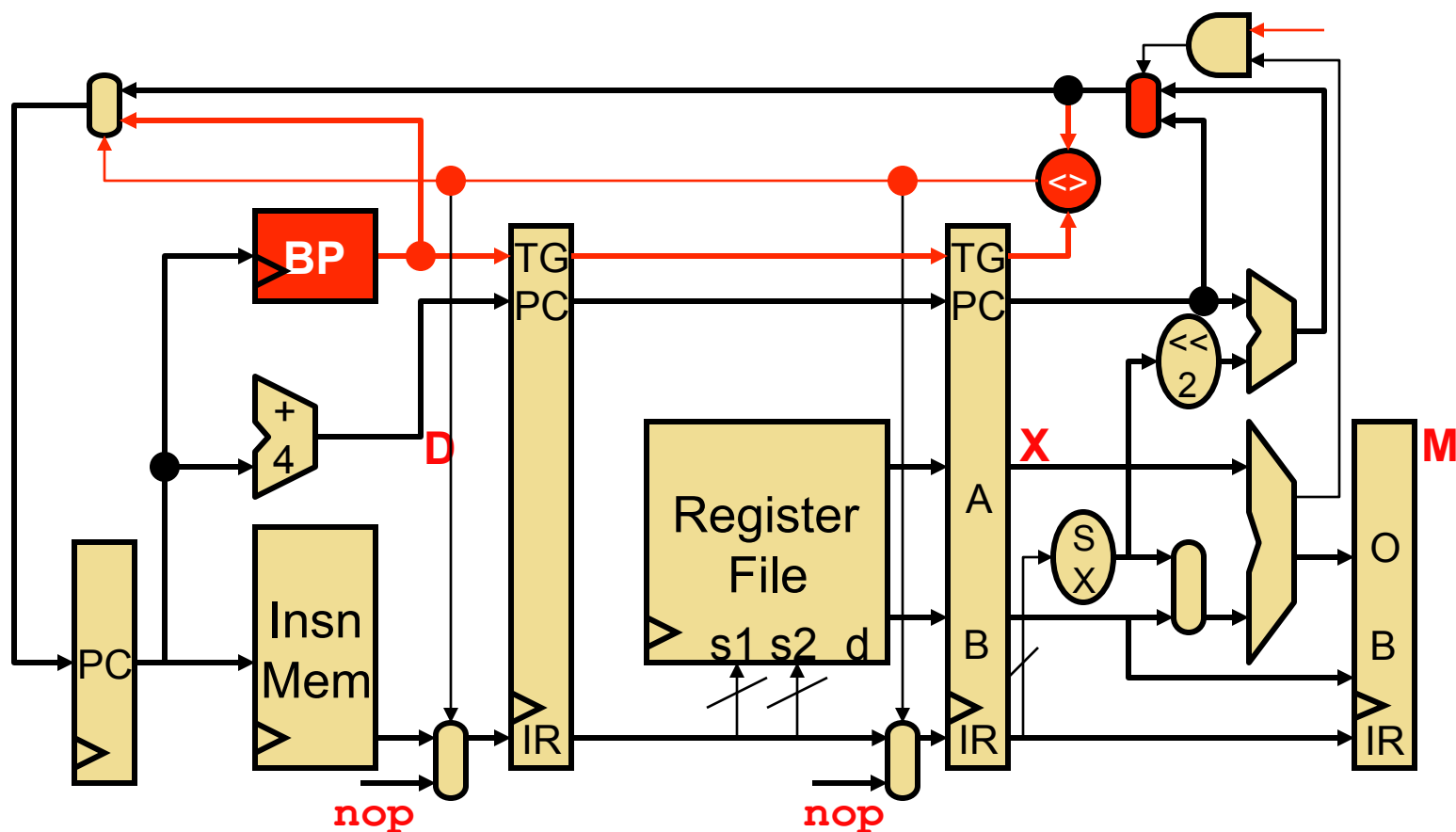
Big Idea: Speculative Execution

- 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)
- **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

Dynamic Branch Prediction

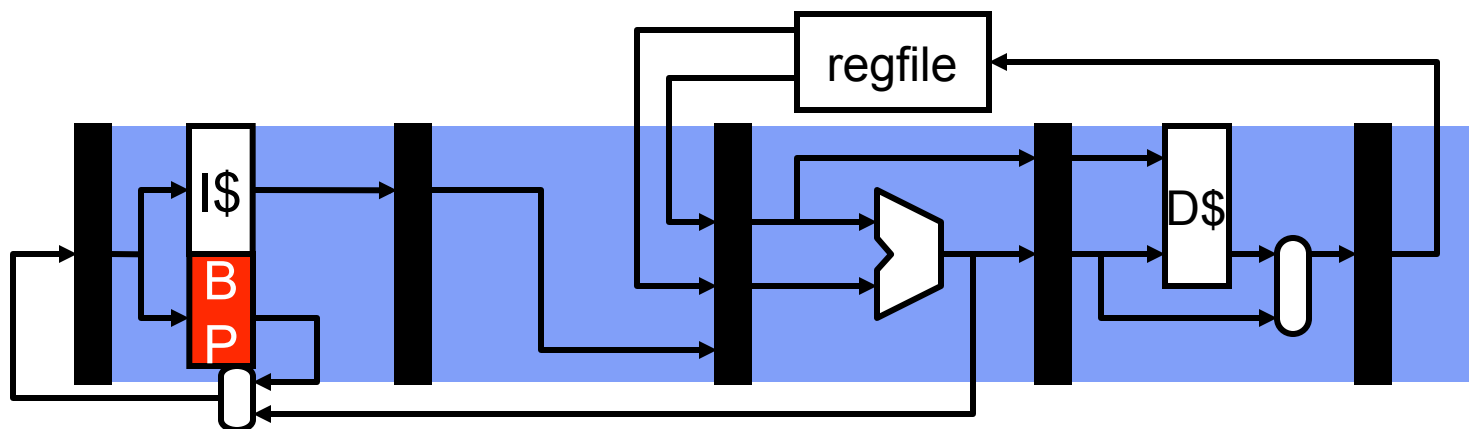


- **Dynamic branch prediction:** hardware guesses outcome
 - Start fetching from guessed address
 - Flush on **mis-prediction**

Branch Prediction Performance

- Parameters
 - **Branch: 20%**, load: 20%, store: 10%, other: 50%
 - 75% of branches are taken
- Dynamic branch prediction
 - Branches predicted with 95% accuracy
 - $\text{CPI} = 1 + 20\% * 5\% * 2 = \mathbf{1.02}$

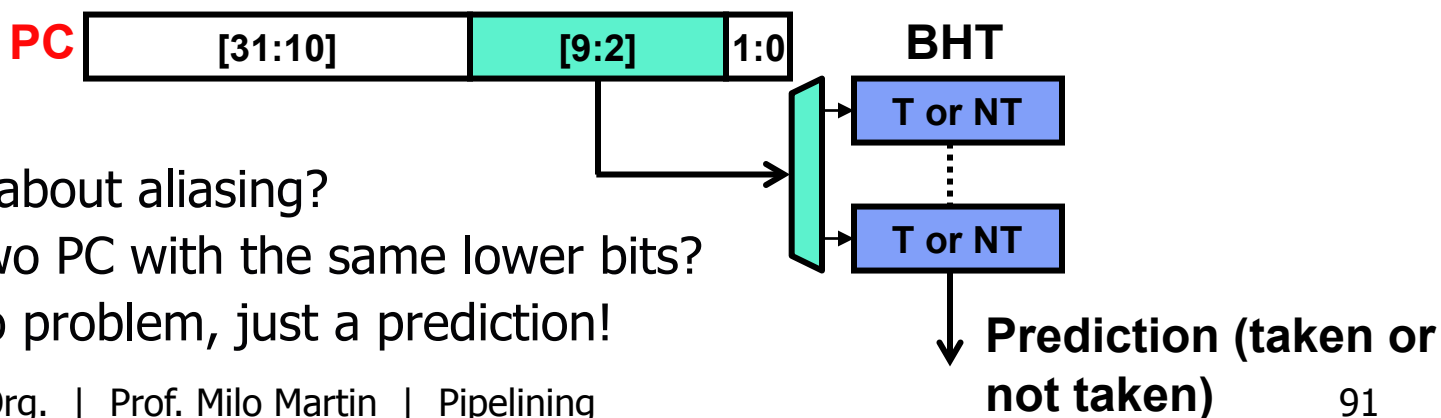
Dynamic Branch Prediction Components



- Step #1: is it a branch?
 - Easy after decode...
- Step #2: is the branch taken or not taken?
 - **Direction predictor** (applies to conditional branches only)
 - Predicts taken/not-taken
- Step #3: if the branch is taken, where does it go?
 - Easy after decode...

Branch Direction Prediction

- **Learn from past, predict the future**
 - Record the past in a hardware structure
- **Direction predictor (DIRP)**
 - Map conditional-branch PC to taken/not-taken (T/N) decision
 - Individual conditional branches often biased or weakly biased
 - 90%+ one way or the other considered **"biased"**
 - Why? Loop back edges, checking for uncommon conditions
- **Branch history table (BHT):** simplest predictor
 - PC indexes table of bits (0 = N, 1 = T), no tags
 - Essentially: branch will go same way it went last time



- What about aliasing?
 - Two PC with the same lower bits?
 - No problem, just a prediction!

Branch History Table (BHT)

- **Branch history table (BHT):**

simplest direction predictor

- PC indexes table of bits (0 = N, 1 = T), no tags
- Essentially: branch will go same way it went last time

- Problem: **inner loop branch** below

```
for (i=0;i<100;i++)  
    for (j=0;j<3;j++)  
        // whatever
```

- Two “built-in” mis-predictions per inner loop iteration
- Branch predictor “changes its mind too quickly”

Time	State	Prediction	Outcome	Result?
1	N	N	T	Wrong
2	T	T	T	Correct
3	T	T	T	Correct
4	T	T	N	Wrong
5	N	N	T	Wrong
6	T	T	T	Correct
7	T	T	T	Correct
8	T	T	N	Wrong
9	N	N	T	Wrong
10	T	T	T	Correct
11	T	T	T	Correct
12	T	T	N	Wrong

Two-Bit Saturating Counters (2bc)

- **Two-bit saturating counters (2bc)**

[Smith 1981]

- Replace each single-bit prediction
 - $(0,1,2,3) = (N,n,t,T)$
- Adds "hysteresis"
 - Force predictor to mis-predict twice before "changing its mind"
- One mispredict each loop execution (rather than two)
 - + Fixes this pathology (which is not contrived, by the way)
 - Can we do even better?

Time	State	Prediction	Outcome	Result?
1	N	N	T	Wrong
2	n	N	T	Wrong
3	t	T	T	Correct
4	T	T	N	Wrong
5	t	T	T	Correct
6	T	T	T	Correct
7	T	T	T	Correct
8	T	T	N	Wrong
9	t	T	T	Correct
10	T	T	T	Correct
11	T	T	T	Correct
12	T	T	N	Wrong

Correlated Predictor

- **Correlated (two-level) predictor** [Patt 1991]
 - Exploits observation that branch outcomes are correlated
 - Maintains separate prediction per (PC, BHR) pairs
 - **Branch history register (BHR)**: recent branch outcomes
 - Simple working example: assume program has one branch
 - BHT: one 1-bit DIRP entry
 - BHT+**2BHR**: $2^2 = 4$ 1-bit DIRP entries
 - Why didn't we do better?
 - BHT not long enough to capture pattern

Time	"pattern"	State				Prediction	Outcome	
		NN	NT	TN	TT		Result?	
1	NN	N	N	N	N	N	T	Wrong
2	NT	T	N	N	N	N	T	Wrong
3	TT	T	T	N	N	N	T	Wrong
4	TT	T	T	N	T	N	Wrong	
5	TN	T	T	N	N	N	T	Wrong
6	NT	T	T	T	N	T	T	Correct
7	TT	T	T	T	N	N	T	Wrong
8	TT	T	T	T	T	N	Wrong	
9	TN	T	T	T	N	T	T	Correct
10	NT	T	T	T	N	T	T	Correct
11	TT	T	T	T	N	N	T	Wrong
12	TT	T	T	T	T	N	Wrong	

Correlated Predictor – 3 Bit Pattern

- **Try 3 bits of history**
- 2^3 DIRP entries per pattern

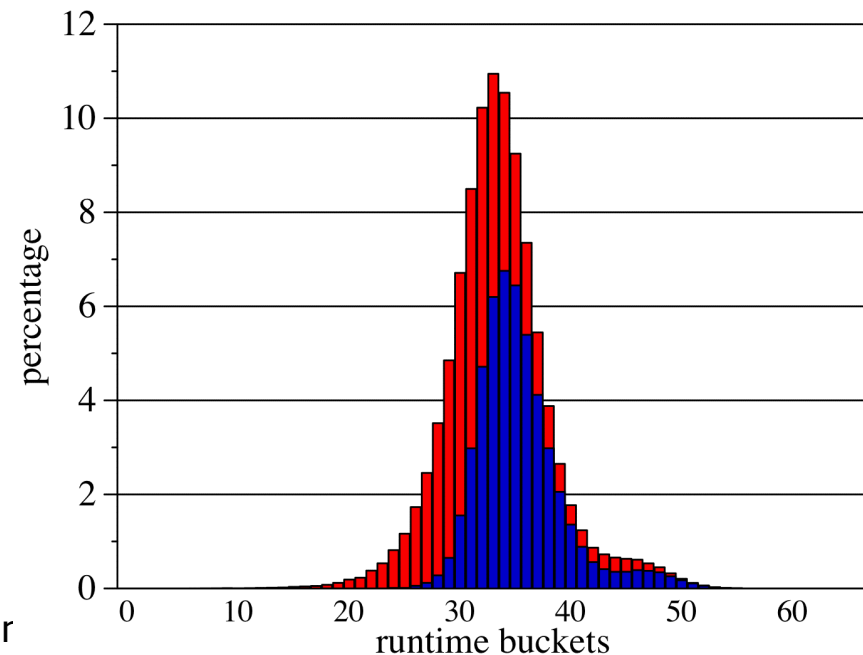
Time	"Pattern"	State								Prediction	Outcome	Result?
		NNN	NNT	NTN	NTT	TNN	TNT	TTN	TTT			
1	NNN	N	N	N	N	N	N	N	N	N	T	Wrong
2	NNT	T	N	N	N	N	N	N	N	N	T	Wrong
3	NTT	T	T	N	N	N	N	N	N	N	T	Wrong
4	TTT	T	T	N	T	N	N	N	N	N	N	Correct
5	TTN	T	T	N	T	N	N	N	N	N	T	Wrong
6	TNT	T	T	N	T	N	N	T	N	N	T	Wrong
7	NTT	T	T	N	T	N	T	T	N	T	T	Correct
8	TTT	T	T	N	T	N	T	T	N	N	N	Correct
9	TTN	T	T	N	T	N	T	T	N	T	T	Correct
10	TNT	T	T	N	T	N	T	T	N	T	T	Correct
11	NTT	T	T	N	T	N	T	T	N	T	T	Correct
12	TTT	T	T	N	T	N	T	T	N	N	N	Correct

+ No mis-predictions after predictor learns all the relevant patterns!

Recall: Fastest and Slowest Leaf Nodes

- Expectation:
 - Let's just consider the leaves
 - Same depth, similar instruction count -> similar runtime
- Some of the fastest leaves (all ~24): L = Left, R = Right
 - LLLLLLLLLLLLLLLLLL
 - LLLLLLLLLLLLLLLLLR (or any with one "R")
 - LLRLLRLLRLLRLL
 - LLRRLRLRLRLRLRLR
 - LLRRRLRLLRRRLRLLR
- RRRRRRRRRRRRRRRR
 - was worst than average (~41)
- Some of the slowest leaves:
 - RRRRLRRRLRLLRLLL (~62)
 - RRRRLRRRLRLLRLLR (~56)
 - RRRRLRRRLRLLRLLL (~56)

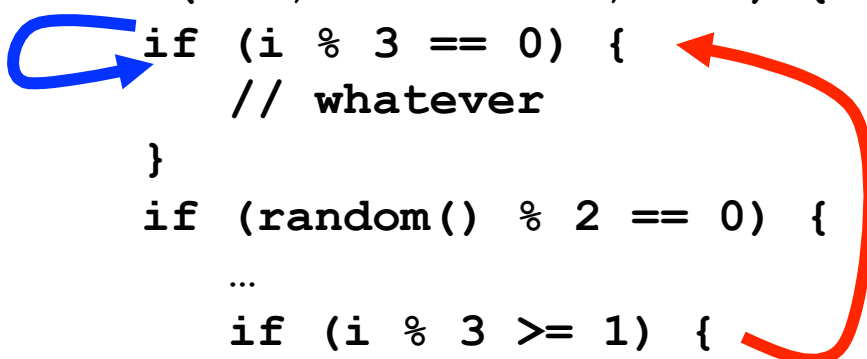
Runtime distribution - Intel Core2



Correlated Predictor Design I

- Design choice I: one **global** BHR or one per PC (**local**)?
 - Each one captures different kinds of patterns
 - Global history captures relationship among different branches
 - Local history captures "self" correlation
 - Local history requires another table to store the per-PC history
- Consider:

```
for (i=0; i<1000000; i++) { // Highly biased
    if (i % 3 == 0) { // "Local" correlated
        // whatever
    }
    if (random() % 2 == 0) { // Unpredictable
        ...
        if (i % 3 >= 1) { // "Global" correlated
            // whatever
        }
    }
}
```

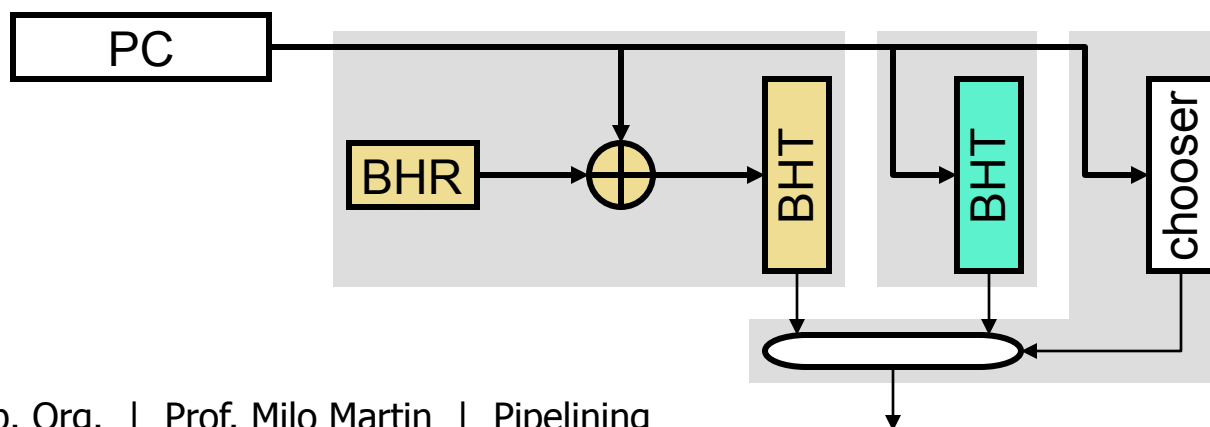


Correlated Predictor Design II

- Design choice II: how many history bits (BHR size)?
 - Tricky one
 - + Given unlimited resources, longer BHRs are better, but...
 - BHT utilization decreases
 - Many history patterns are never seen
 - Many branches are history independent (don't care)
 - PC xor BHR allows multiple PCs to dynamically share BHT
 - BHR length $< \log_2(\text{BHT size})$
 - Predictor takes longer to train
 - Typical length: 8–12

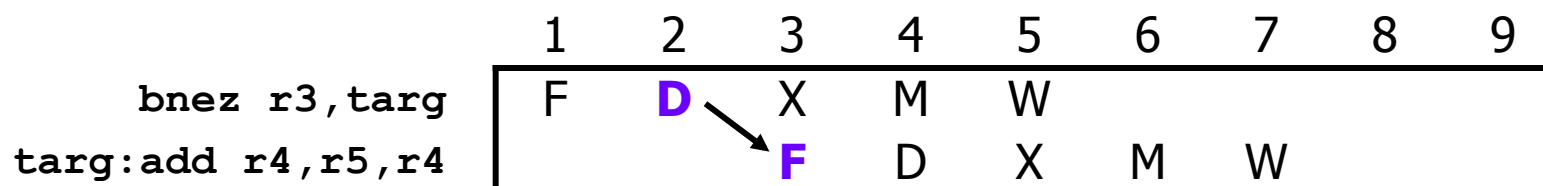
Hybrid Predictor

- **Hybrid (tournament) predictor** [McFarling 1993]
 - Attacks correlated predictor BHT capacity problem
 - Idea: combine two predictors
 - **Simple BHT** predicts history independent branches
 - **Correlated predictor** predicts only branches that need history
 - **Chooser** assigns branches to one predictor or the other
 - Branches start in simple BHT, move mis-prediction threshold
- + Correlated predictor can be made **smaller**, handles fewer branches
- + 90–95% accuracy



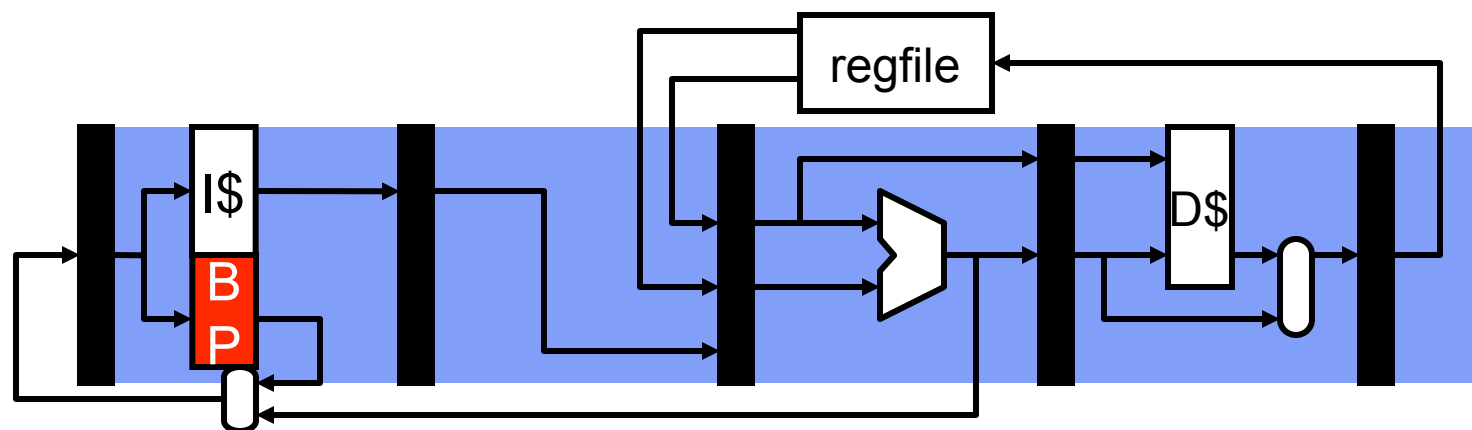
When to Perform Branch Prediction?

- Option #1: During Decode
 - Look at instruction opcode to determine branch instructions
 - Can calculate next PC from instruction (for PC-relative branches)
 - One cycle “mis-fetch” penalty **even if branch predictor is correct**



- Option #2: During Fetch?
 - How do we do that?

Revisiting Branch Prediction Components



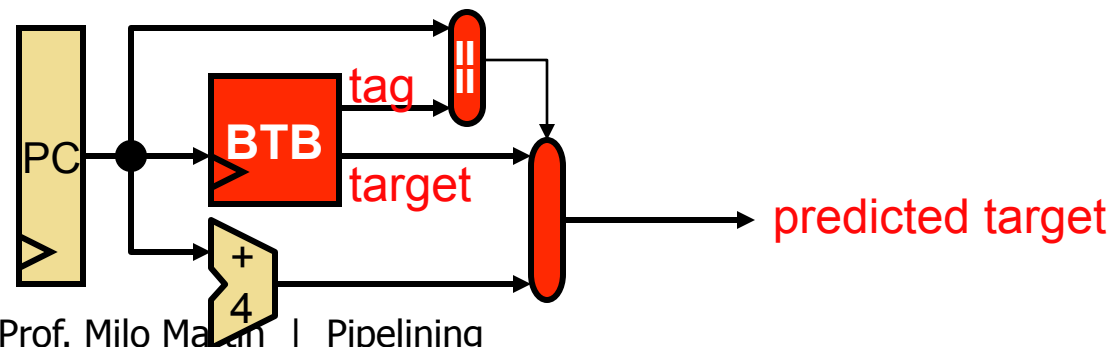
- Step #1: is it a branch?
 - Easy after decode... during fetch: **predictor**
- Step #2: is the branch taken or not taken?
 - **Direction predictor** (as before)
- Step #3: if the branch is taken, where does it go?
 - **Branch target predictor (BTB)**
 - Supplies target PC if branch is taken

Branch Target Buffer (BTB)

- As before: learn from past, predict the future
 - Record the past branch targets in a hardware structure
- **Branch target buffer (BTB):**
 - “guess” the future PC based on past behavior
 - “Last time the branch X was taken, it went to address Y”
 - “So, in the future, if address X is fetched, fetch address Y next”
- Operation
 - A small RAM: address = PC, data = target-PC
 - Access at Fetch *in parallel* with instruction memory
 - predicted-target = BTB[hash(PC)]
 - Updated at X whenever target \neq predicted-target
 - BTB[hash(PC)] = target
 - Hash function is just typically just extracting lower bits (as before)
 - Aliasing? No problem, this is only a prediction

Branch Target Buffer (continued)

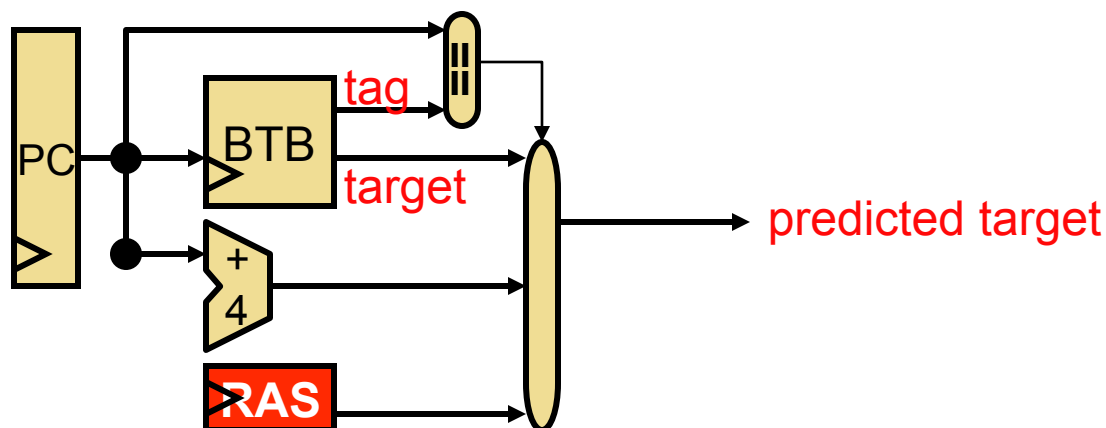
- At Fetch, how does insn know it's a branch & should read BTB? It doesn't have to...
 - **...all insns access BTB in parallel with Imem Fetch**
- Key idea: **use BTB to predict which insn are branches**
 - Implement by "tagging" each entry with its corresponding PC
 - Update BTB on every taken branch insn, record target PC:
 - $BTB[PC].tag = PC$, $BTB[PC].target = \text{target of branch}$
 - All insns access at Fetch *in parallel* with Imem
 - Check for tag match, signifies insn at that PC is a branch
 - Predicted PC = $(BTB[PC].tag == PC) ? BTB[PC].target : PC+4$



Why Does a BTB Work?

- Because most control insns use **direct targets**
 - Target encoded in insn itself → same “taken” target every time
- What about **indirect targets**?
 - Target held in a register → can be different each time
 - Two indirect call idioms
 - + Dynamically linked functions (DLLs): target always the same
 - Dynamically dispatched (virtual) functions: hard but uncommon
 - Also two indirect unconditional jump idioms
 - Switches: hard but uncommon
 - Function returns: hard and common but...

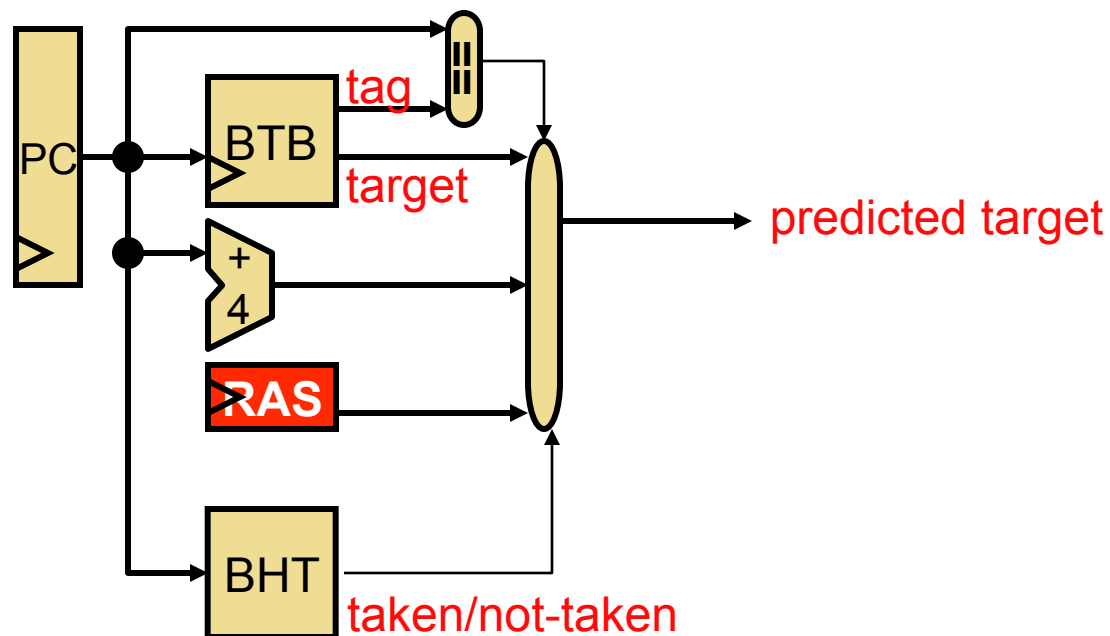
Return Address Stack (RAS)



- **Return address stack (RAS)**
 - Call instruction? $RAS[TopOfStack++] = PC+4$
 - Return instruction? Predicted-target = $RAS[--TopOfStack]$
 - Q: how can you tell if an insn is a call/return before decoding it?
 - Accessing RAS on every insn BTB-style doesn't work
 - Answer: another predictor (or put them in BTB marked as "return")
 - Or, **pre-decode bits** in insn mem, written when first executed

Putting It All Together

- BTB & branch direction predictor during fetch



- If branch prediction correct, no taken branch penalty

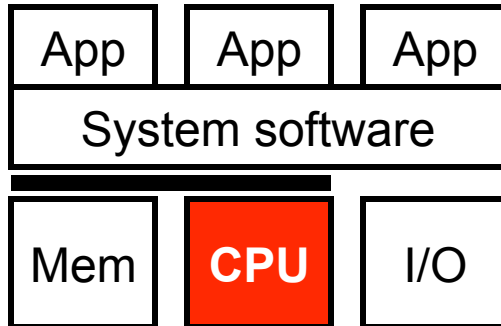
Branch Prediction Performance

- Dynamic branch prediction
 - 20% of instruction branches
 - Simple predictor: branches predicted with 75% accuracy
 - $\text{CPI} = 1 + (20\% * 25\% * 2) = 1.1$
 - More advanced predictor: 95% accuracy
 - $\text{CPI} = 1 + (20\% * 5\% * 2) = 1.02$
- Branch mis-predictions still a big problem though
 - Pipelines are long: typical mis-prediction penalty is 10+ cycles
 - For cores that do more per cycle, predictions more costly (later)

Pipeline Depth

- 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 double the stages reduce the clock period by less than 2x
 - Decreases IPC (increases CPI)
 - Branch mis-prediction penalty becomes longer
 - Non-bypassed data hazard stalls become longer
 - At some point, actually causes performance to decrease, but when?
 - 1GHz Pentium 4 was slower than 800 MHz PentiumIII
 - “Optimal” pipeline depth is program and technology specific

Summary



- Processor performance
 - Latency vs throughput
- Single-cycle & multi-cycle datapaths
- Basic pipelining
- Data hazards
 - Software interlocks and scheduling
 - Hardware interlocks and stalling
 - Bypassing
 - Load-use stalling
- Pipelined multi-cycle operations
- Control hazards
 - Branch prediction