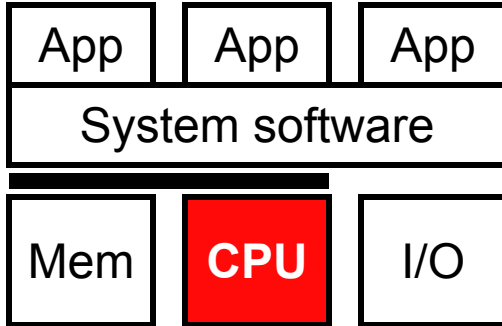


Computer Architecture

Lec 5: Pipelining

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



Recall: 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*

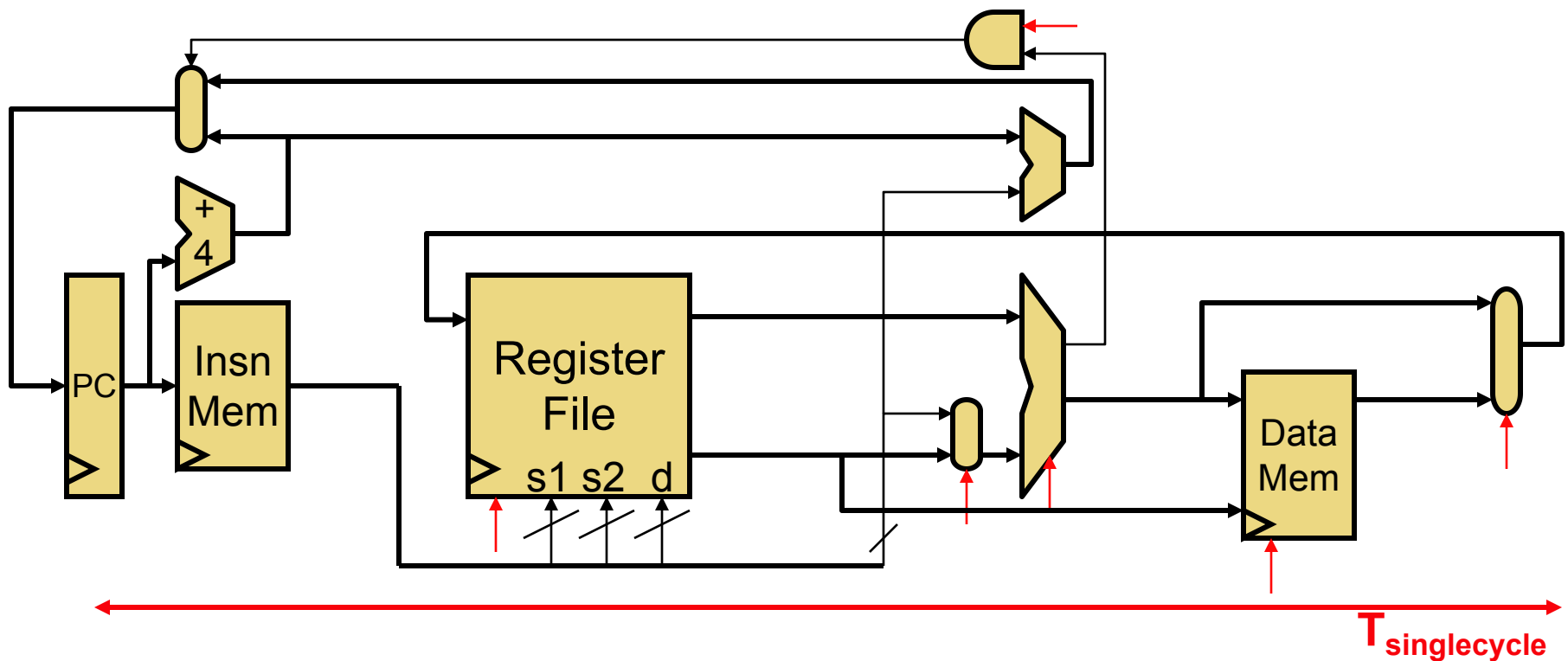
Recall: Cycles per Instruction (CPI)

- **CPI**: Cycle/instruction **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

Recall: Improving Clock Frequency

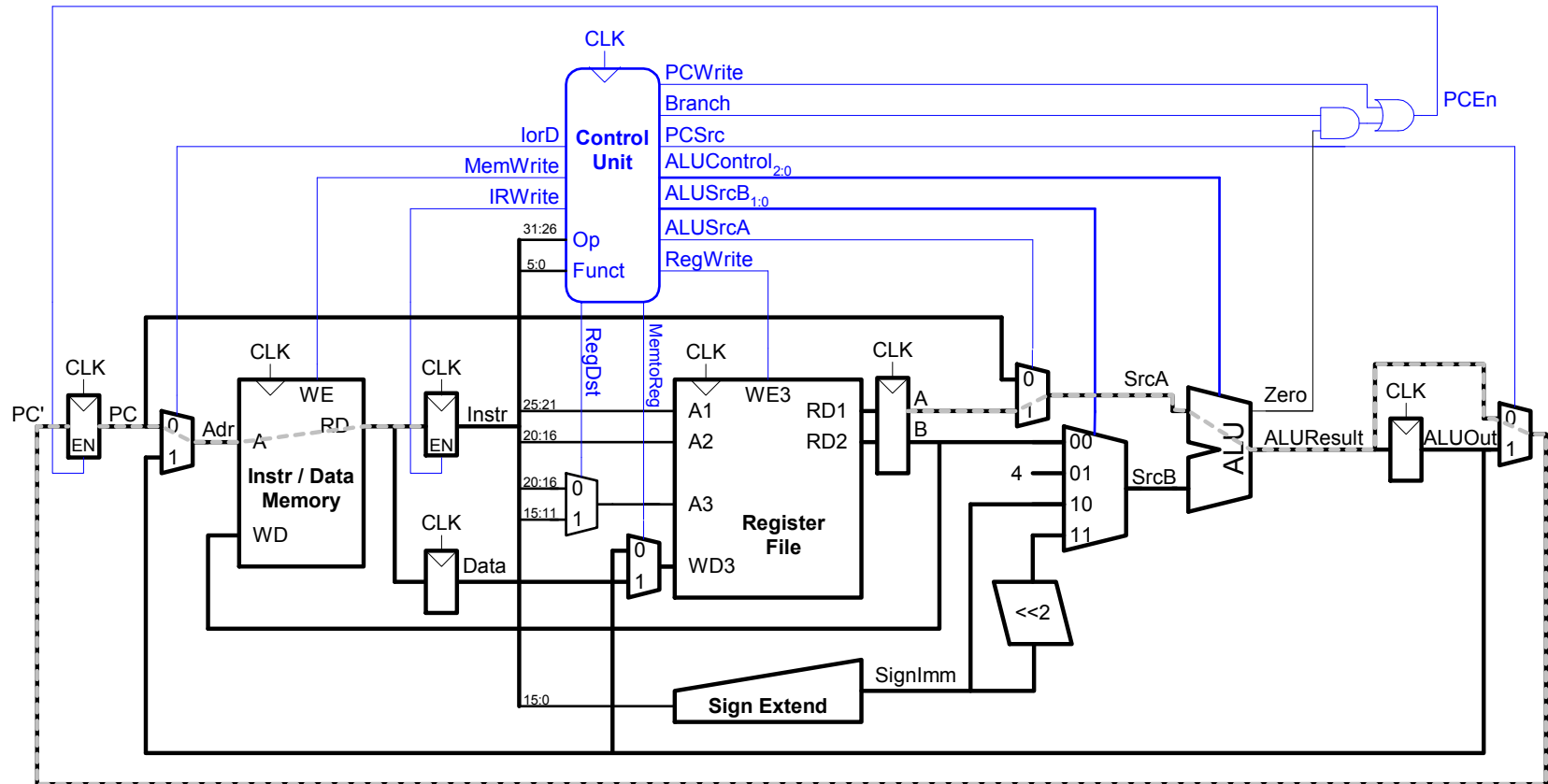
- **Faster transistors**
- Micro-architectural techniques
 - **Multi-cycle processors**
 - Break each instruction into small steps
 - Less logic delay -> improved clock frequency
 - Different instructions take different number of cycles
 - $CPI > 1$
 - **Pipelined processors**
 - As above, but overlap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

Recall: 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)

Reall: Multi-Cycle Datapath



- **Multi-cycle datapath**

- Divide all instructions into multiple steps
- + Less logic delay -> improved clock frequency
- + Different instructions take different number of cycles
- $CPI > 1$

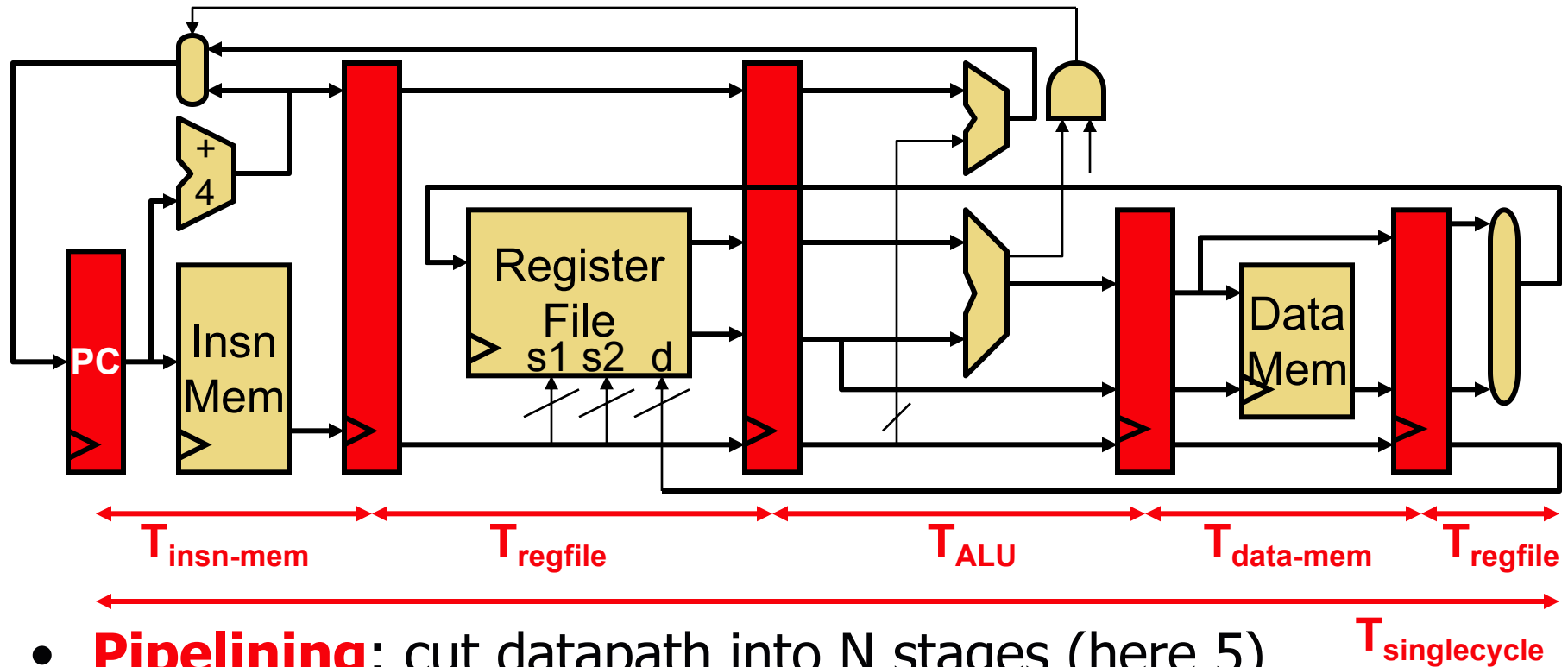
Pipelined Datapath

Pipelining



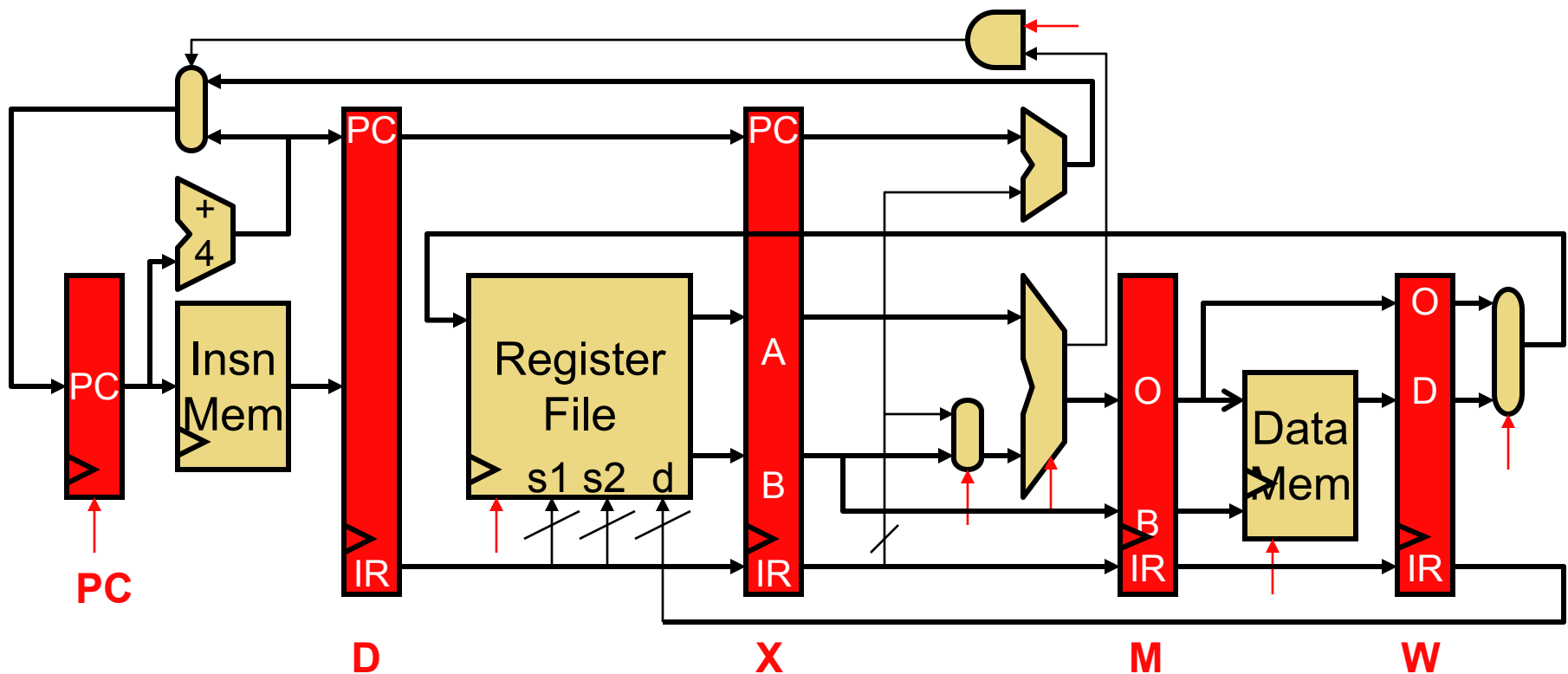
- Important performance technique
 - **Improves instruction throughput, not instruction latency**
- Break instruction execution into stages
 - When insn advances from stage 1 to 2, next insn enters at stage 1
 - Form of parallelism: “insn-stage parallelism”
 - Maintains illusion of sequential fetch/execute loop
 - Individual instruction takes the same number of stages
 - + **But instructions enter and leave at a much faster rate**
- Just like the laundromat

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 = $\text{MAX}(T_{\text{insn-mem}}, T_{\text{regfile}}, T_{\text{ALU}}, T_{\text{data-mem}}, T_{\text{writeback}})$
 - + Base CPI = 1: insn enters and leaves every cycle
 - Actual CPI > 1: pipeline must often “stall”
 - Individual insn latency **increases** (pipeline overhead)

5 Stage Pipelined Datapath

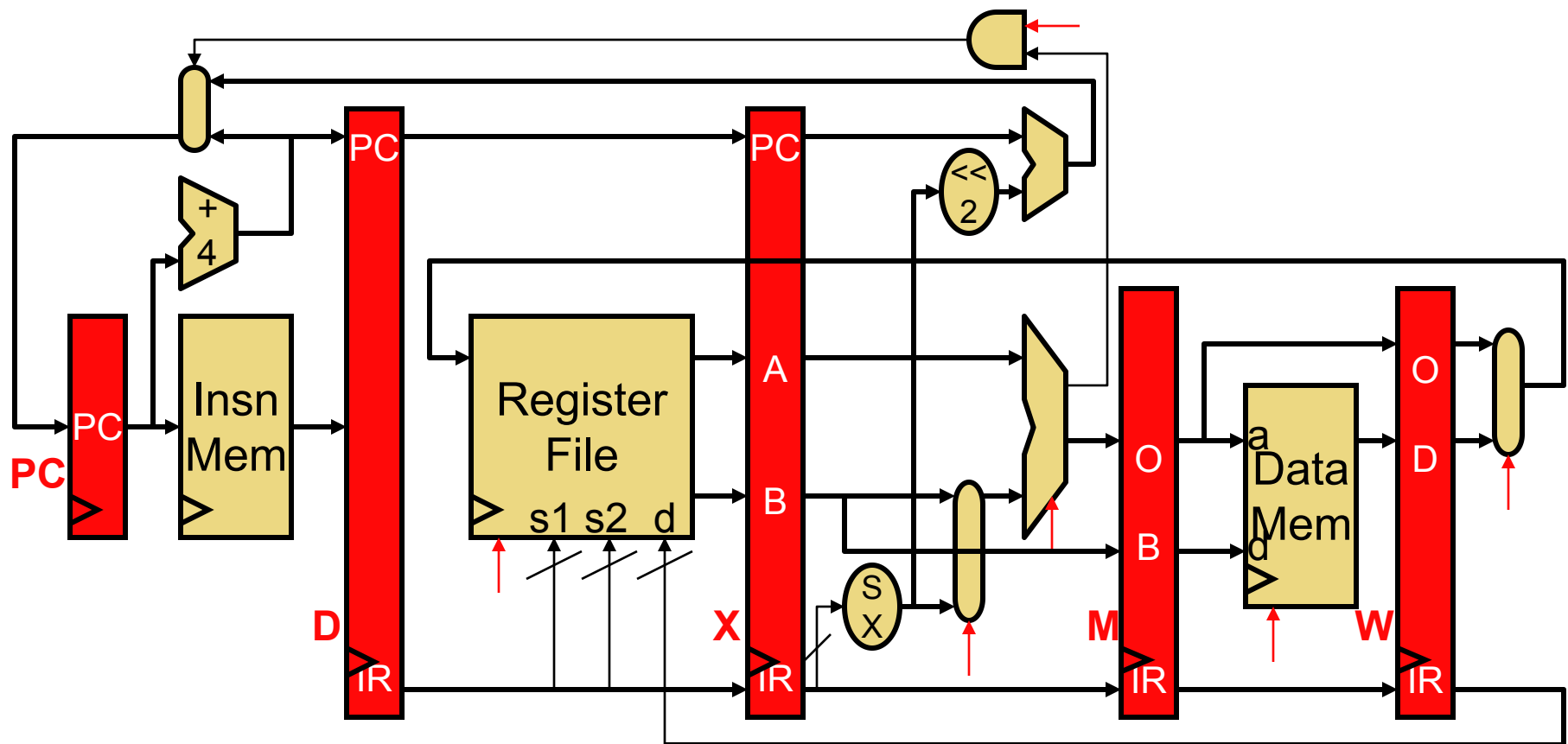


- Five stage: **F**etch, **D**ecode, e**X**ecute, **M**emory, **W**riteback
 - Nothing magical about 5 stages (Pentium 4 had 22 stages!)
- Pipeline registers named by stages they begin
 - **PC, D, X, M, W**

Pipeline Terminology

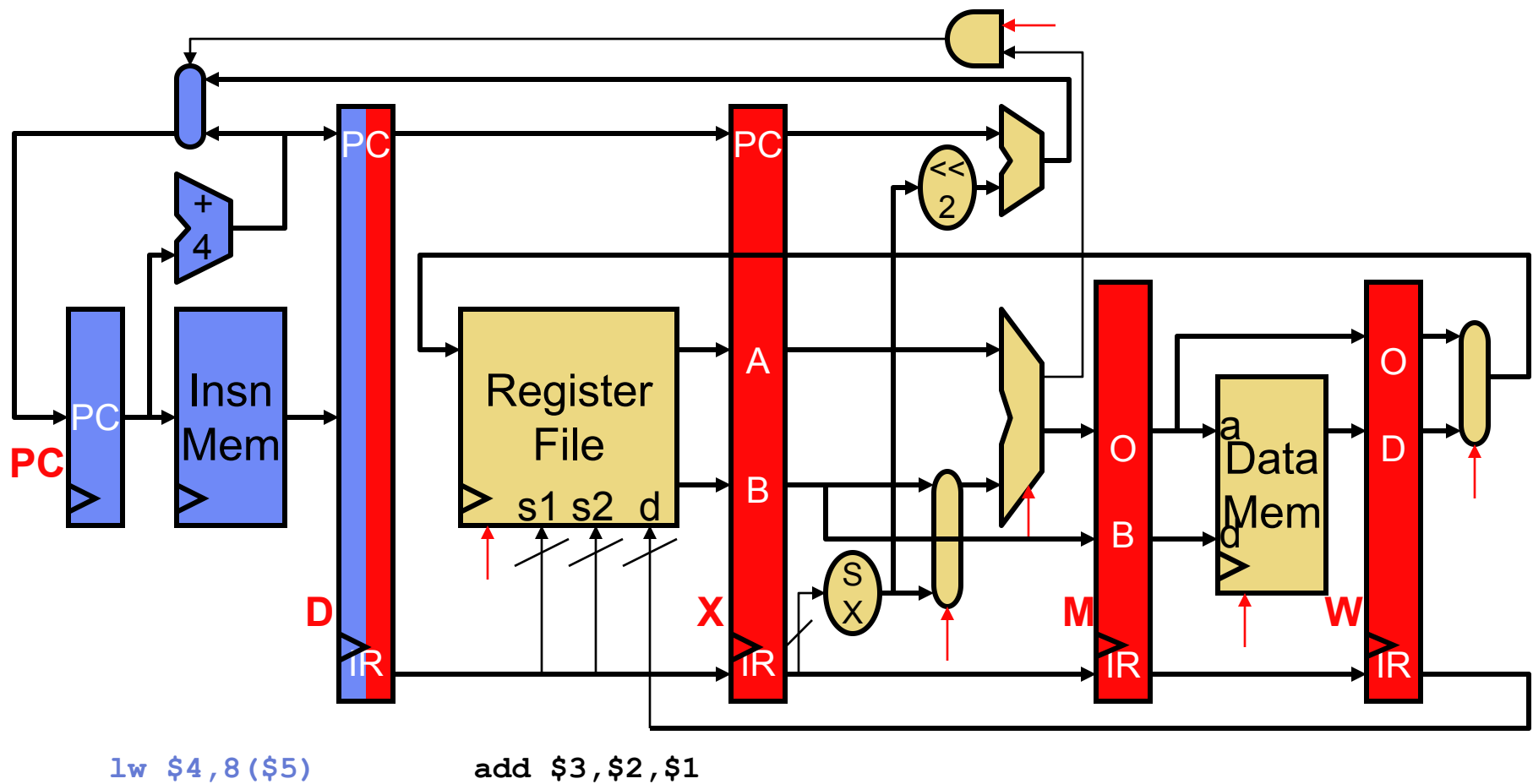
- **Scalar pipeline**: one insn per stage per cycle
 - Alternative: “superscalar” (later)
- **In-order pipeline**: insns enter execute stage in order
 - Alternative: “out-of-order” (later)
- **Pipeline depth**: number of pipeline stages
 - Nothing magical about five
 - Contemporary high-performance cores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

Pipeline Example: Cycle 1

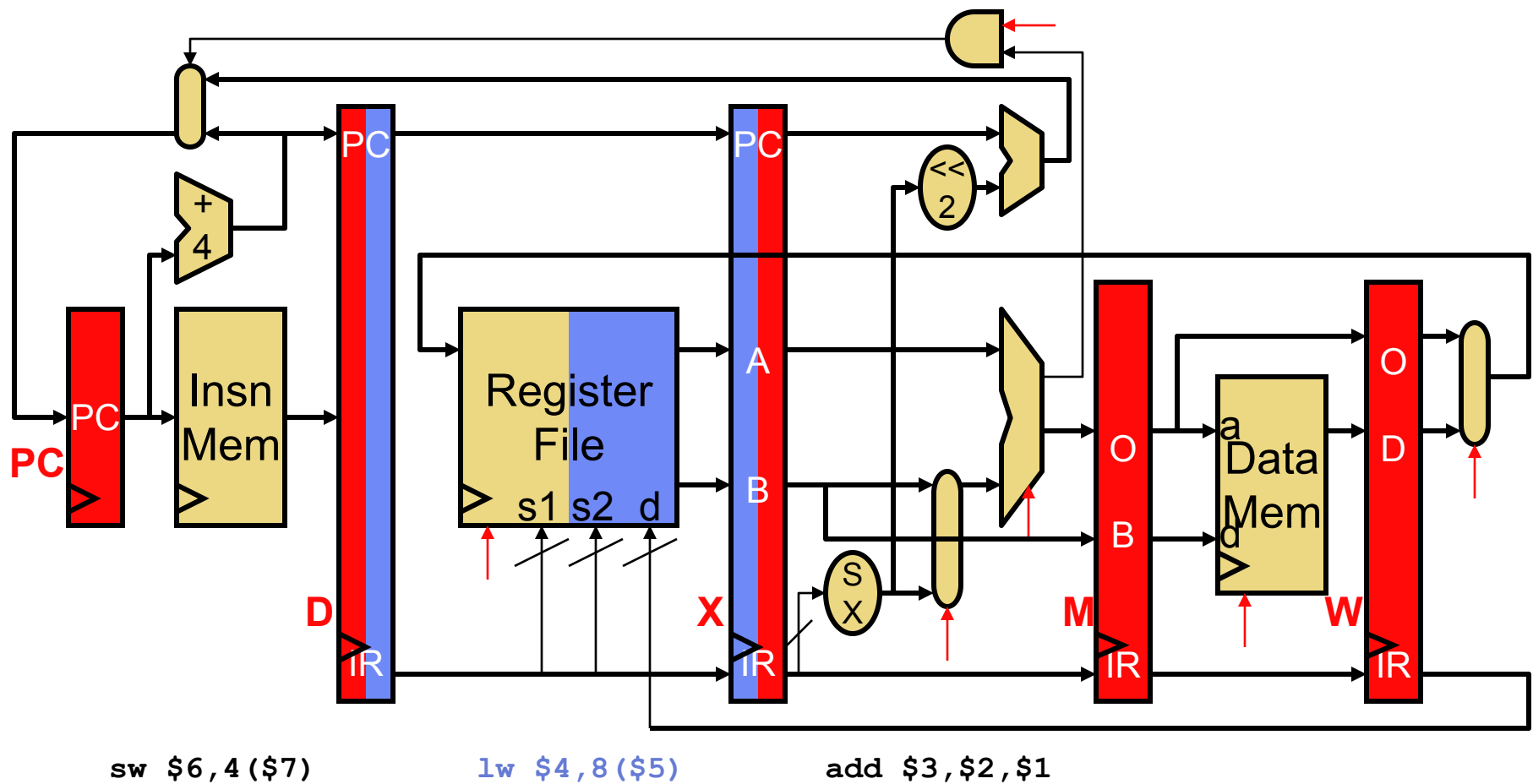


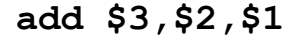
`add $3, $2, $1`

Pipeline Example: Cycle 2

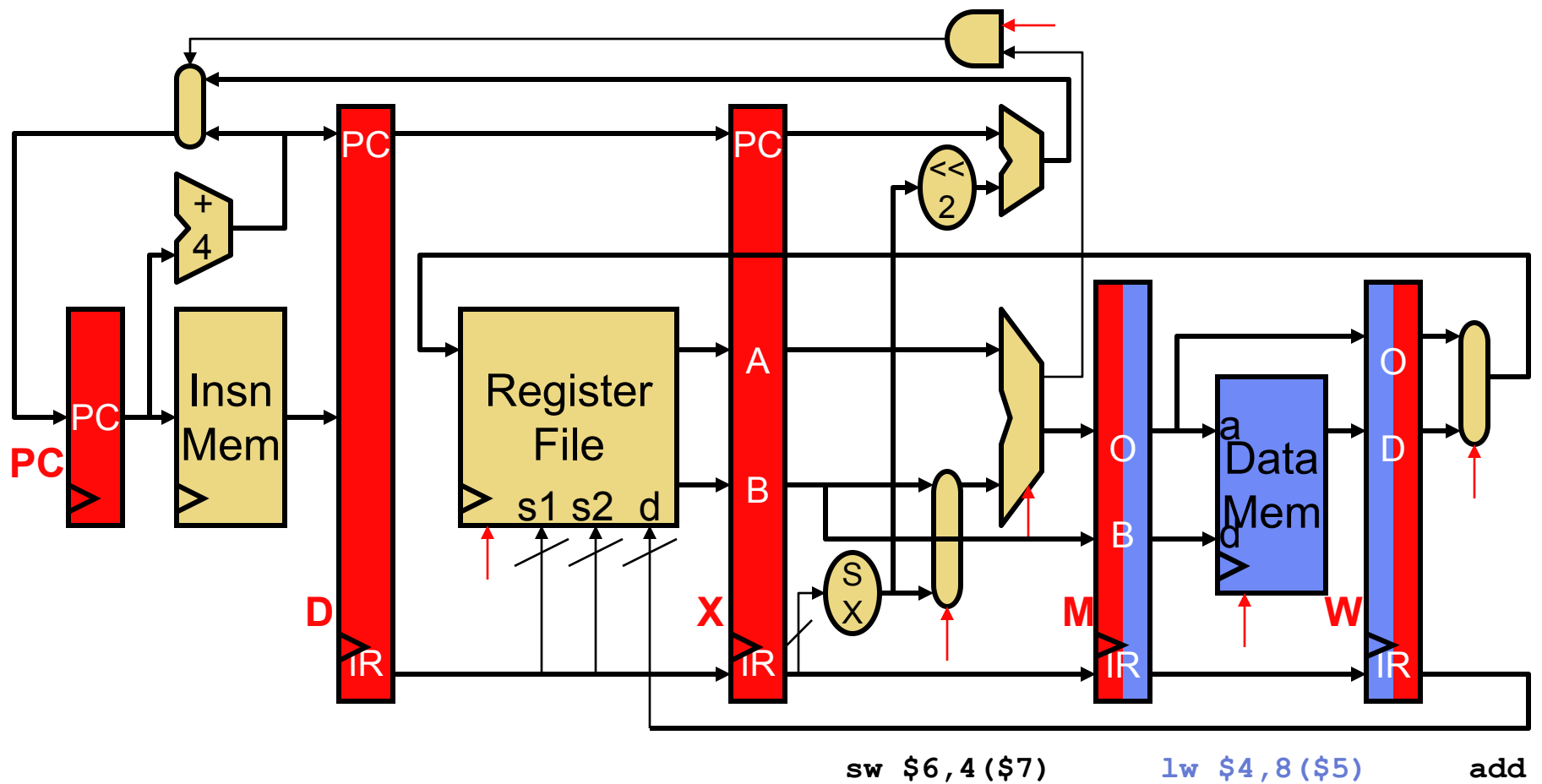


Pipeline Example: Cycle 3

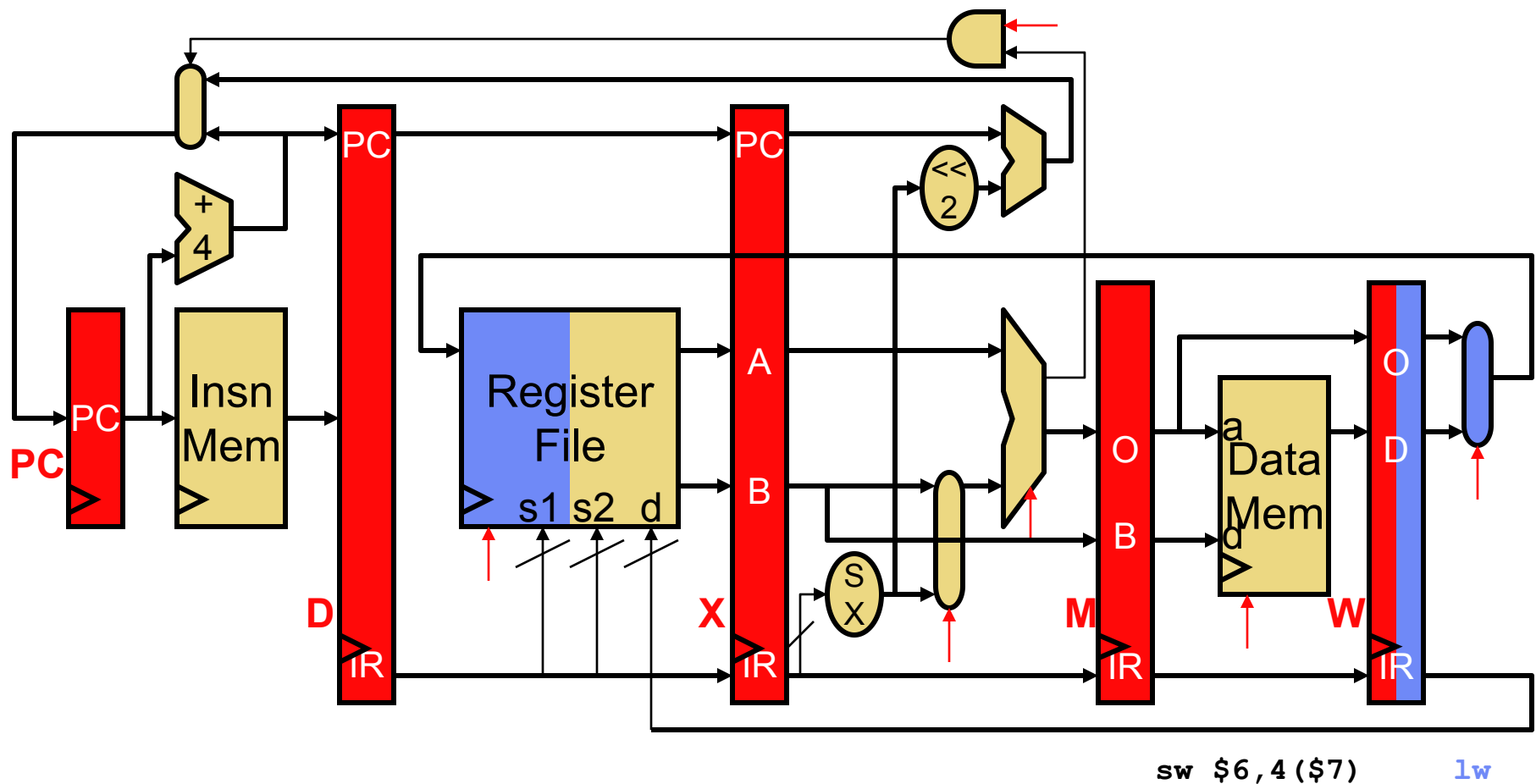




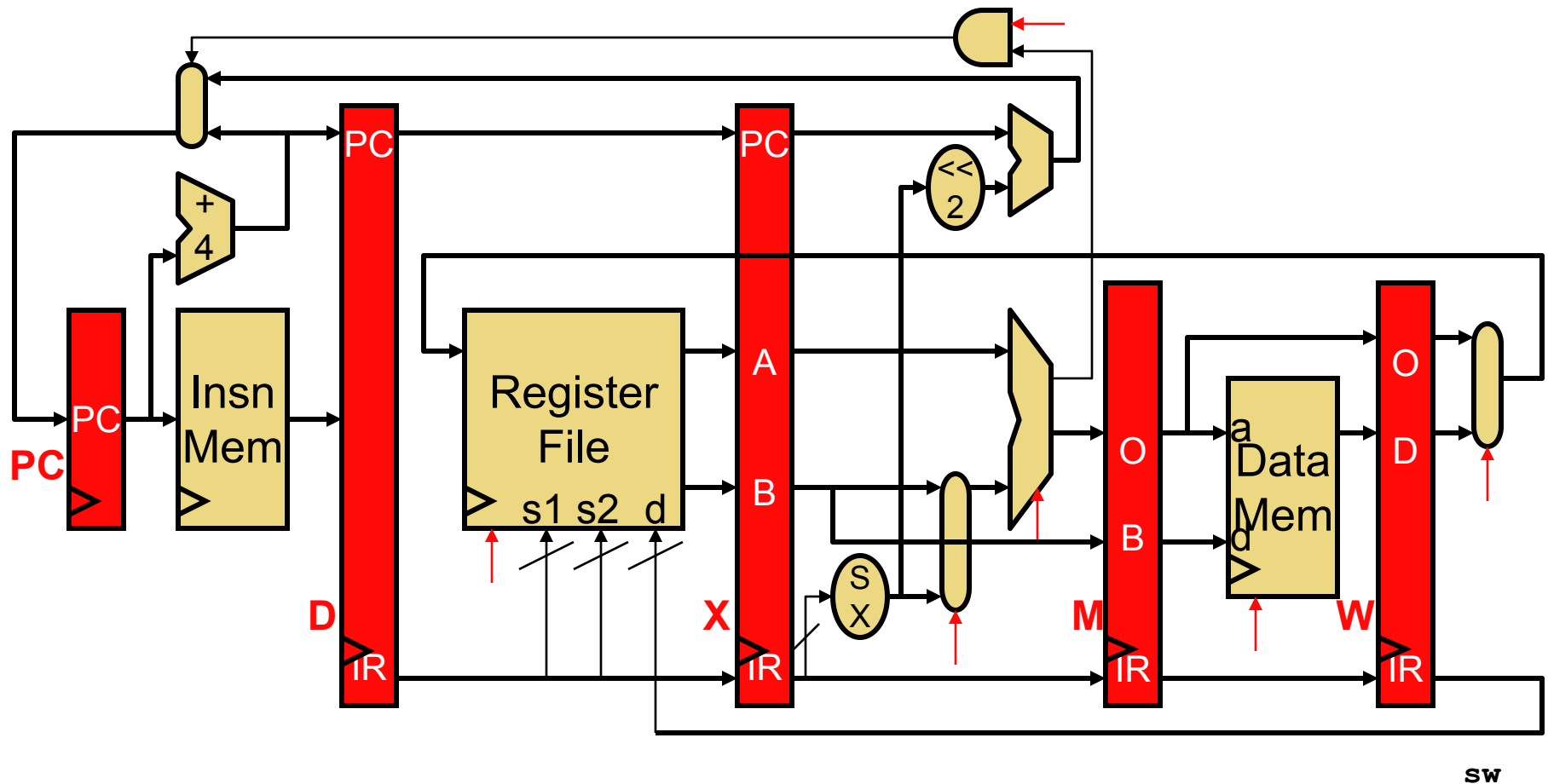
Pipeline Example: Cycle 5



Pipeline Example: Cycle 6

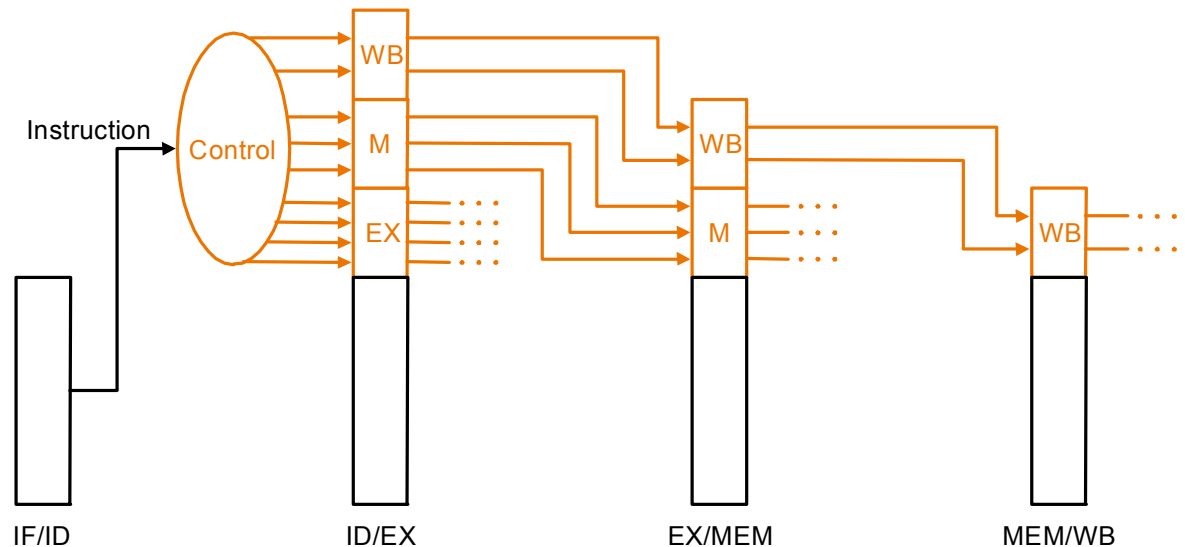


Pipeline Example: Cycle 7



Control Signals in a Pipeline

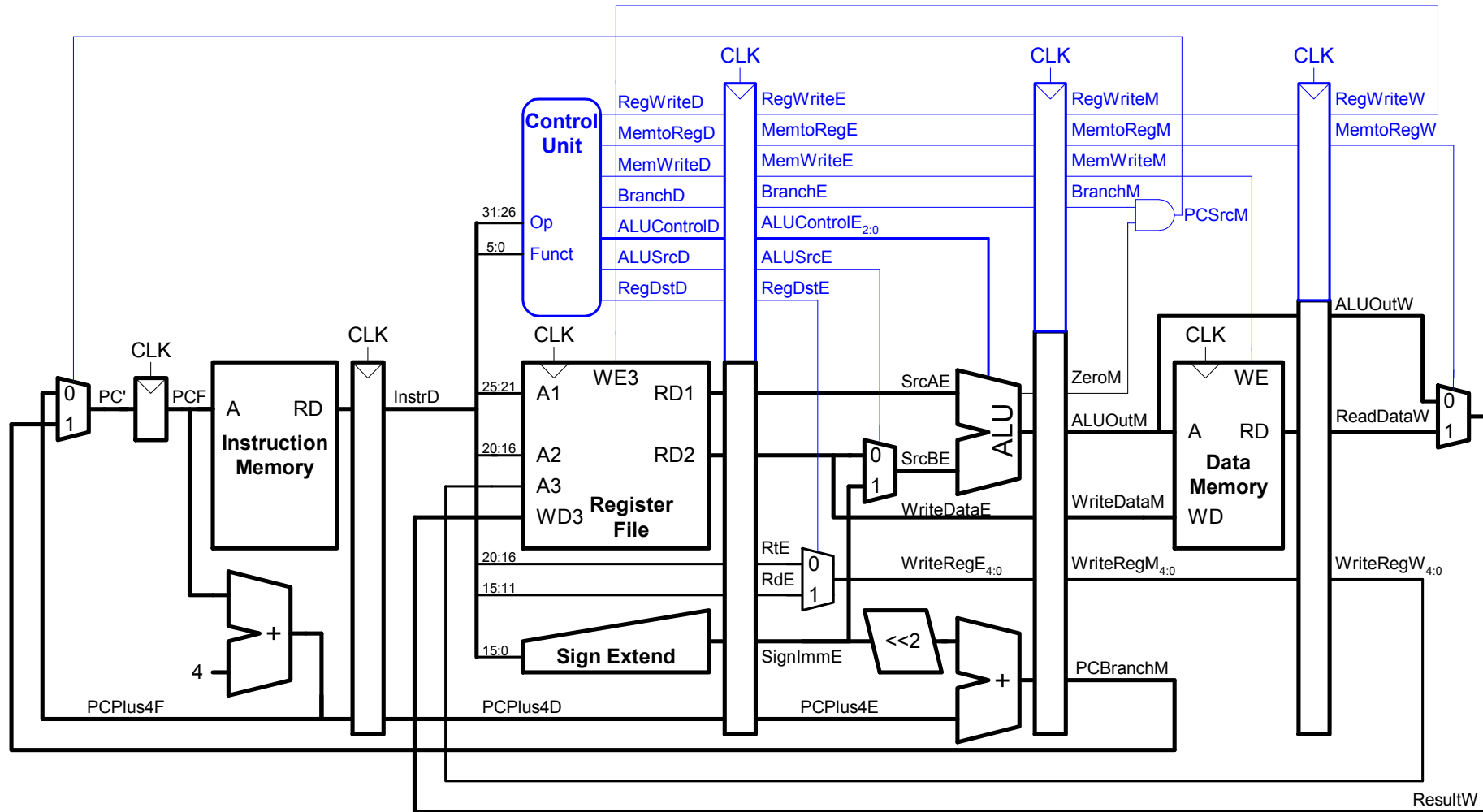
- For a given instruction
 - same control signals as single-cycle, but
 - control signals required at different cycles, depending on stage
- ⇒ Option 1: decode once using the same logic as single-cycle and buffer signals until consumed



- ⇒ Option 2: carry relevant “instruction word/field” down the pipeline and decode locally within each or in a previous stage

Which one is better?

An Example: Pipelined Control



- Same control unit as single-cycle processor
Control delayed to proper pipeline stage

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 register at **end** of cycle 4
 - assuming no *stalls* (discussed later)

	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
- 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

Q1: Why Is Pipeline Clock Period ...

- ... > (delay thru datapath) / (number of pipeline stages)?
 - Three reasons:
 - Registers 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 typically 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 must not be externally visible
 - **Stall**: for order by keeping younger insn in same stage
 - Hazards are a bad thing: stalls reduce performance

Data Dependences

- Types of data dependences
 - Flow dependence (true data dependence – read after write, RAW)
 - Output dependence (write after write, WAW)
 - Anti dependence (write after read, WAR)
- Which ones cause stalls in a pipelined machine?
 - For all of them, we need to ensure semantics of the program is correct
 - Flow dependences always need to be obeyed because they constitute true dependence on a value
 - Anti and output dependences exist due to limited number of architectural registers
 - They are dependence on a name, not a value
 - We will later see what we can do about them

Dependence types

- **RAW** (Read After Write) = “true dependence” (true)

mul r0 * r1 → **r2**

...

add **r2** + r3 → r4



- **WAW** (Write After Write) = “output dependence” (false)

mul r0 * r1 → **r2**

...

add r1 + r3 → **r2**



- **WAR** (Write After Read) = “anti-dependence” (false)

mul r0 * **r1** → r2

...

add r3 + r4 → **r1**



- WAW & WAR are “false”, Can be **totally eliminated** by “renaming”

Register Renaming

- WAR and WAW dependencies are not true dependencies
 - WHY? The same register refers to values that have nothing to do with each other
 - **They exist because not enough register ID's (i.e. names) in the ISA**
- How register renaming eliminates WAR and WAW dependencies?
 - Require: enough registers

Register Renaming Example

- “Few” vs “Many” registers
 - Few: `r1, r2, r3`
 - Many: `p1, p2, p3, p4, p5, p6, p7`

Original insns

Time

add	r2, r3	→ r1
sub	r2, r1	→ r3
mul	r2, r3	→ r3
div	r1, 4	→ r1

Register Renaming Example

- “Few” vs “Many” registers
 - Few: `r1, r2, r3`
 - Many: `p1, p2, p3, p4, p5, p6, p7`

Original insns

Time

add	r2, r3	→ r1
sub	r2, r1	→ r3
mul	r2, r3	→ r3
div	r1, 4	→ r1

Time	MapTable			FreeList
	r1	r2	r3	
	p1	p2	p3	p4 , p5 , p6 , p7

Original insns

add r2, r3 → r1

Renamed insns

add p2, p3 → p4

Register Renaming Example

- “Few” vs “Many” registers
 - Few: `r1, r2, r3`
 - Many: `p1, p2, p3, p4, p5, p6, p7`

Original insns

Time

add	r2, r3	→	r1
sub	r2, r1	→	r3
mul	r2, r3	→	r3
div	r1, 4	→	r1

Time

	r1	r2	r3
	p1	p2	p3
	p4	p2	p3

p4, p5, p6, p7
p5, p6, p7

Original insns

add	r2, r3	→	r1
sub	r2, r1	→	r3

Renamed insns

add	p2, p3	→	p4
sub	p2, p4	→	p5

Register Renaming Example

- “Few” vs “Many” registers
 - Few: `r1, r2, r3`
 - Many: `p1, p2, p3, p4, p5, p6, p7`

Original insns

Time

add	r2, r3	→	r1
sub	r2, r1	→	r3
mul	r2, r3	→	r3
div	r1, 4	→	r1

Time	MapTable			FreeList
	r1	r2	r3	
	p1	p2	p3	p4, p5, p6, p7
	p4	p2	p3	p5, p6, p7
	p4	p2	p5	p6, p7

Original insns

add r2, r3 → r1
sub r2, r1 → r3
mul r2, r3 → r3

Renamed insns

add p2, p3 → p4
sub p2, p4 → p5
mul p2, p5 → p6

Register Renaming Example

- “Few” vs “Many” registers
 - Few: `r1, r2, r3`
 - Many: `p1, p2, p3, p4, p5, p6, p7`

Original insns

Time

	add	r2, r3	→	r1
	sub	r2, r1	→	r3
	mul	r2, r3	→	r3
	div	r1, 4	→	r1

- + Removes **false** dependences
- + Leaves **true** dependences intact!

Time	MapTable			FreeList
	r1	r2	r3	
	p1	p2	p3	p4 , p5 , p6 , p7
	p4	p2	p3	p5 , p6 , p7
	p4	p2	p5	p6 , p7
	p4	p2	p6	p7

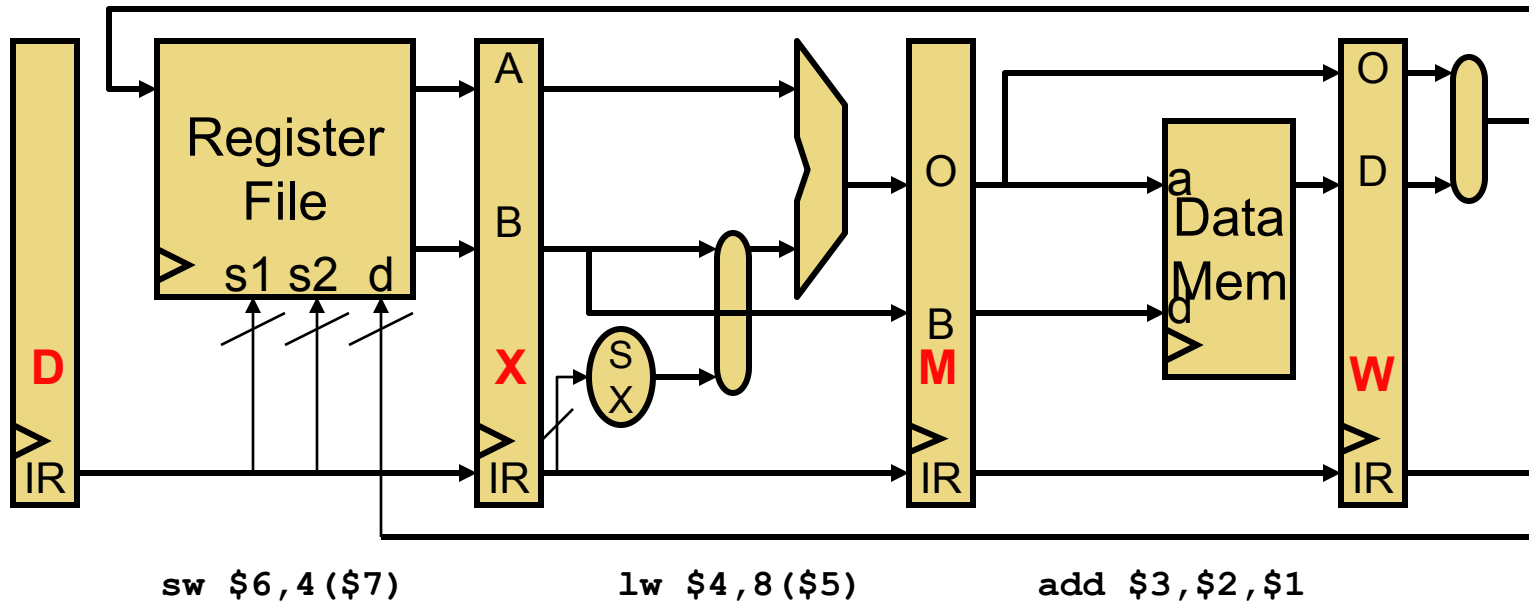
Original insns

add	r2, r3	→	r1
sub	r2, r1	→	r3
mul	r2, r3	→	r3
div	r1, 4	→	r1

Renamed insns

add	p2, p3	→	p4
sub	p2, p4	→	p5
mul	p2, p5	→	p6
div	p4, 4	→	p7

Data Hazards



- Let's forget about branches for now
- 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
```

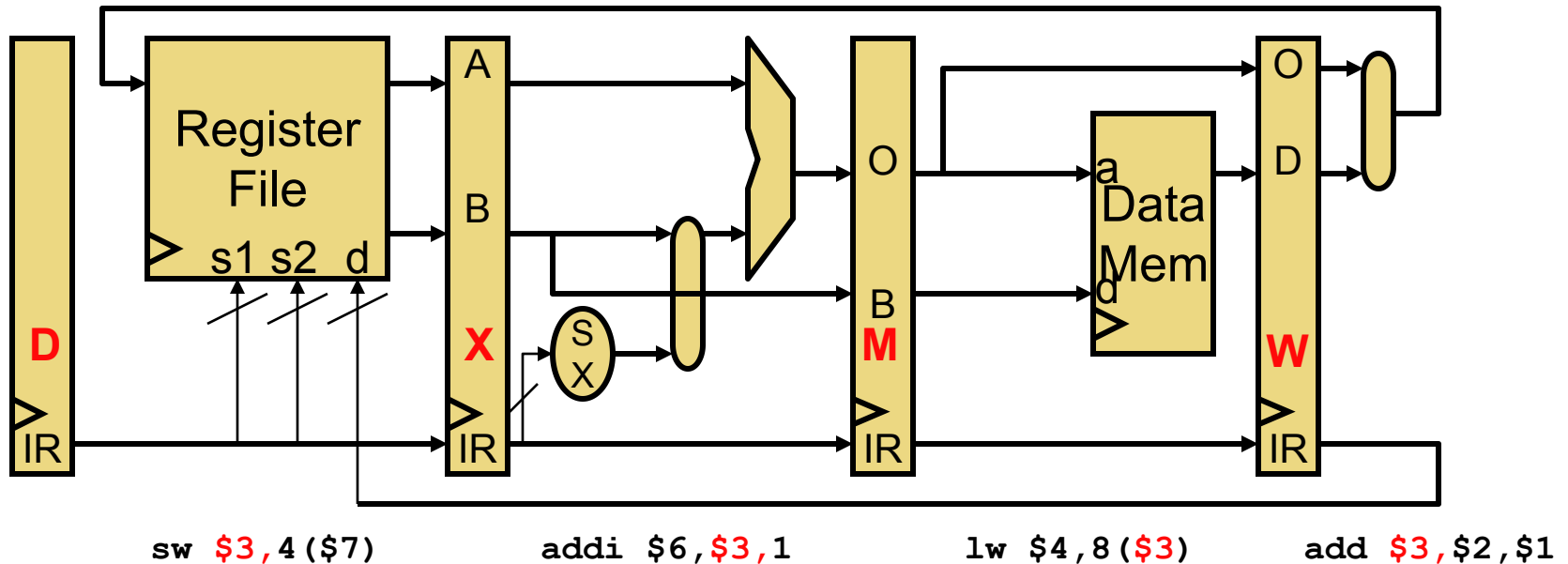
- Would this program execute correctly on a pipeline?

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

- What about this program?

```
add $3,$2,$1  
lw $4,8($3)  
addi $6,$3,1  
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 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)
sub  $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`
 - `sub $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
sub  $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 won't work!
- **Option #2: hardware interlocks**
 - Processor detects data hazards and fixes them
 - Resolves the above compatibility concern
 - Two aspects to this
 - Detecting hazards
 - Fixing hazards

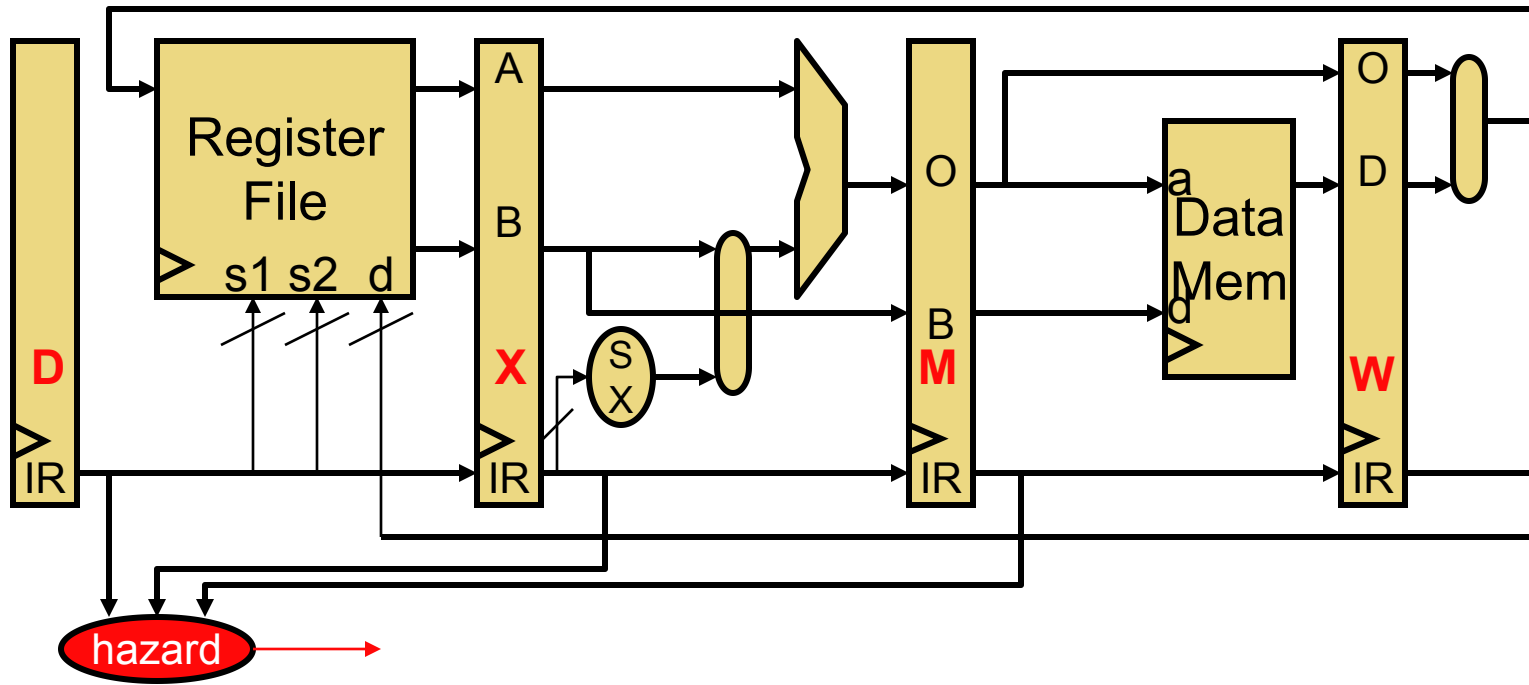
Approaches to Dependence Detection (I)

- Scoreboarding
 - Each register in register file has a Valid bit associated with it
 - An instruction that is writing to the register resets the Valid bit
 - An instruction in Decode stage checks if all its source and destination registers are Valid
 - Yes: No need to stall... No dependence
 - No: Stall the instruction
- Advantage:
 - Simple. 1 bit per register
- Disadvantage:
 - Need to stall for all types of dependences, not only flow dep.

Approaches to Dependence Detection (II)

- **Combinational dependence check logic**
 - Special logic that checks if any instruction in later stages is supposed to write to any source register of the instruction that is being decoded
 - Yes: stall the instruction/pipeline
 - No: no need to stall... no flow dependence
- **Advantage:**
 - No need to stall on anti and output dependences
- **Disadvantage:**
 - Logic is more complex than a scoreboard
 - Logic becomes more complex as we make the pipeline deeper and wider (flash-forward: think superscalar execution)

Detecting Data Hazards

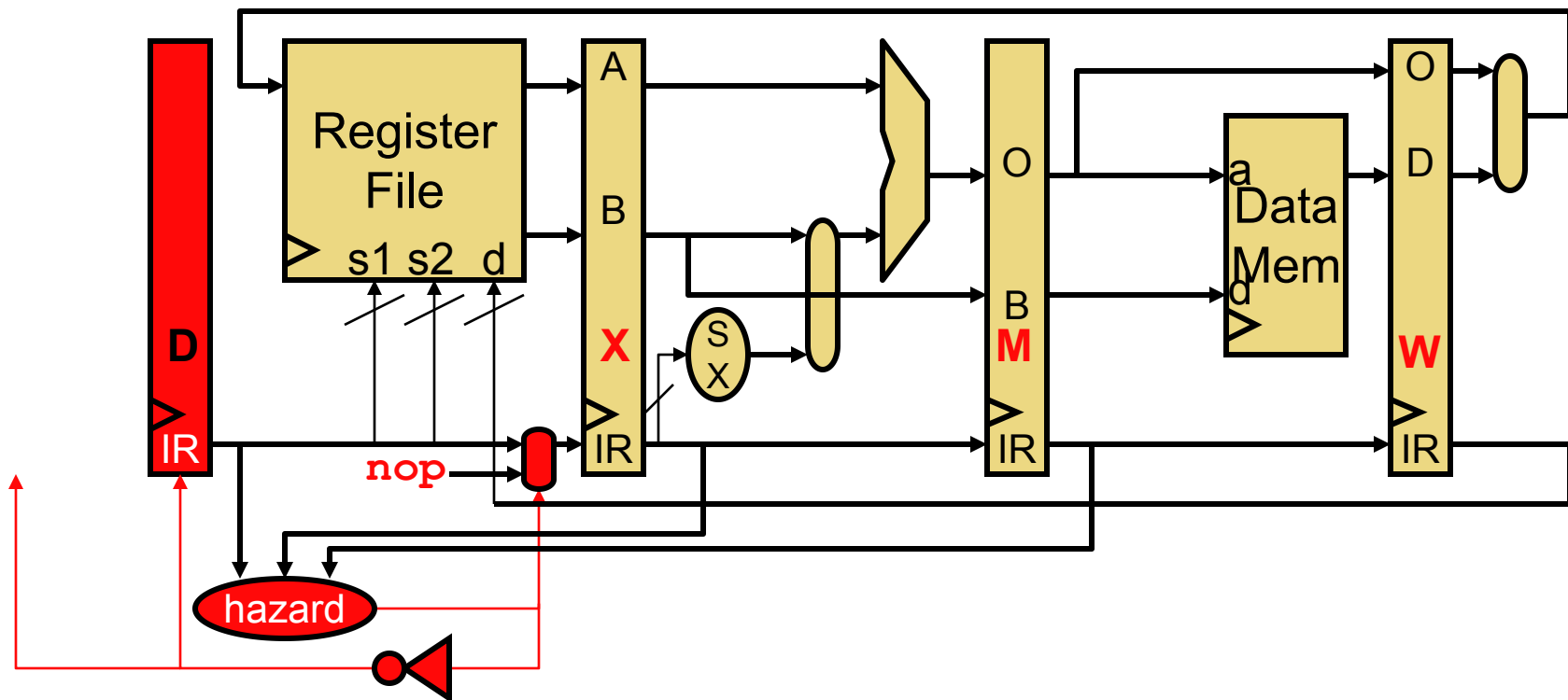


- Compare input register names of insn in D stage with output register names of older insns in pipeline

Stall =

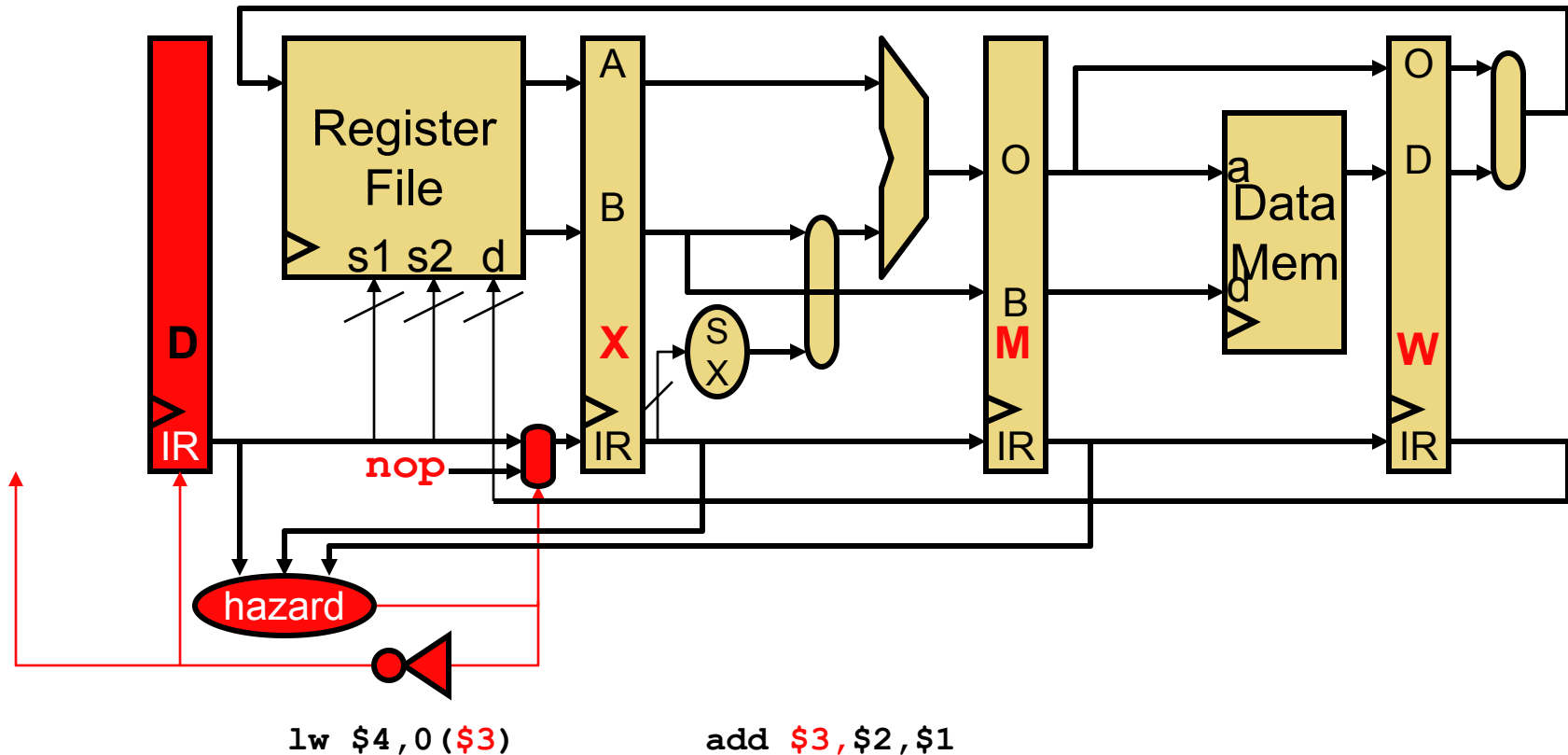
```
(D.IR.RegSrc1 == X.IR.RegDest) ||  
(D.IR.RegSrc2 == X.IR.RegDest) ||  
(D.IR.RegSrc1 == M.IR.RegDest) ||  
(D.IR.RegSrc2 == M.IR.RegDest)
```

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 register and PC write enables (why?)
- Re-evaluate situation next cycle

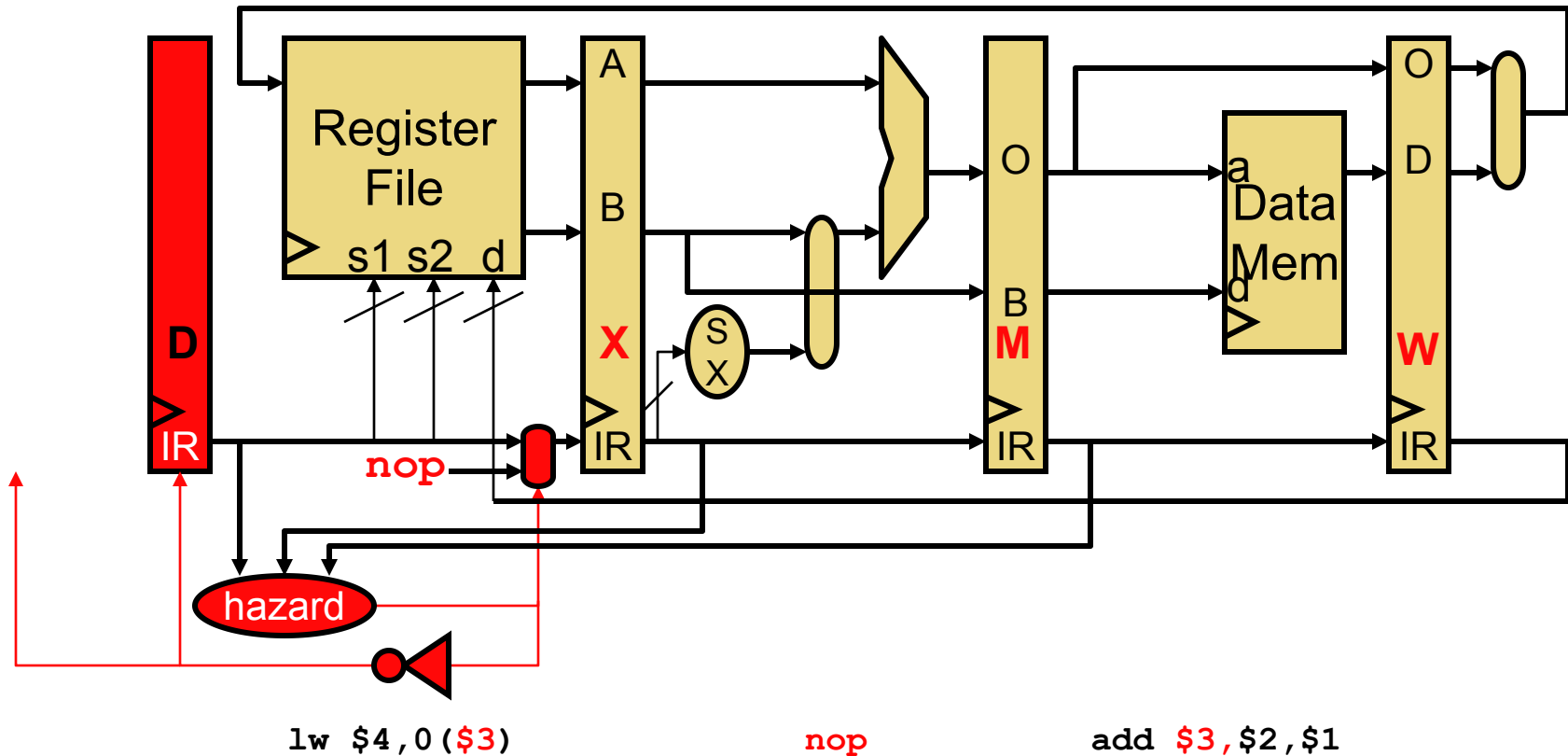
Hardware Interlock Example: cycle 1



Stall =

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

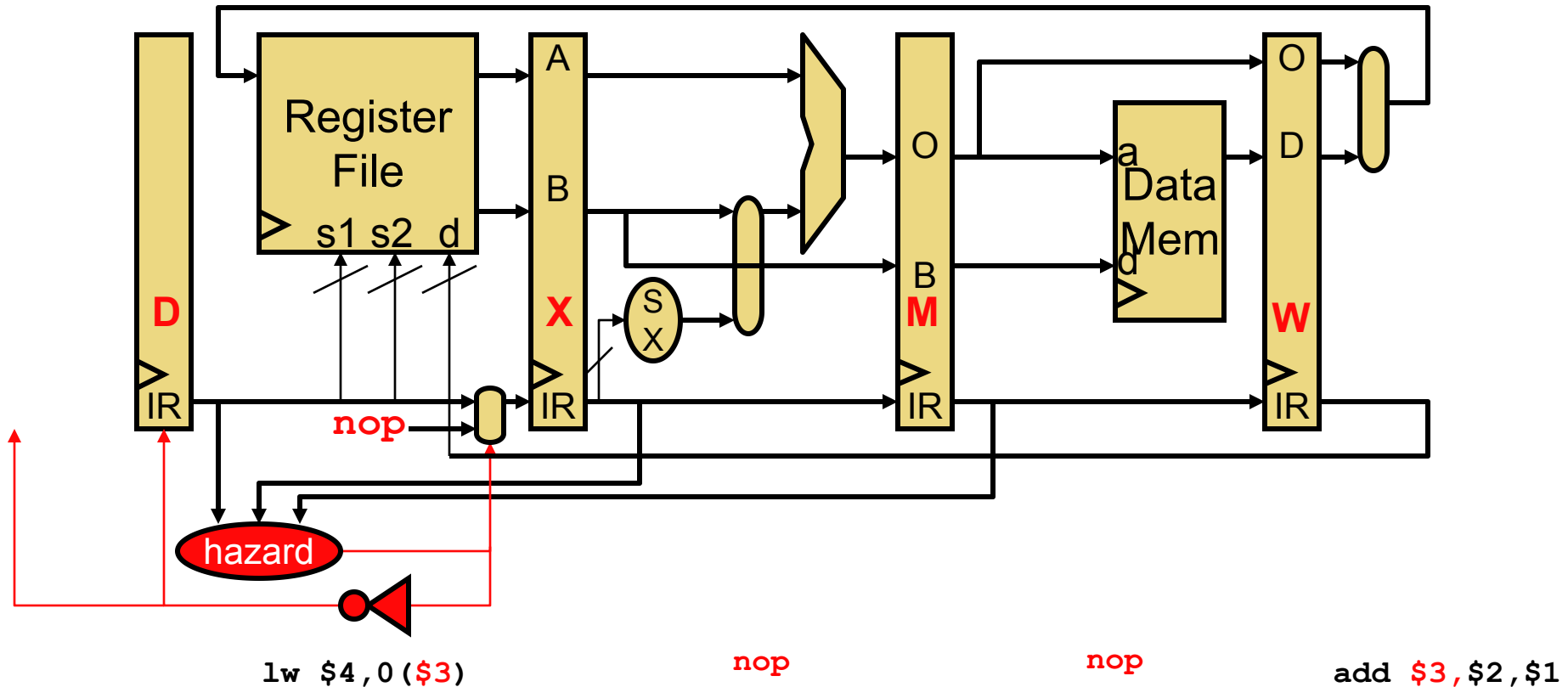
Hardware Interlock Example: cycle 2



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 3



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) = 0
```

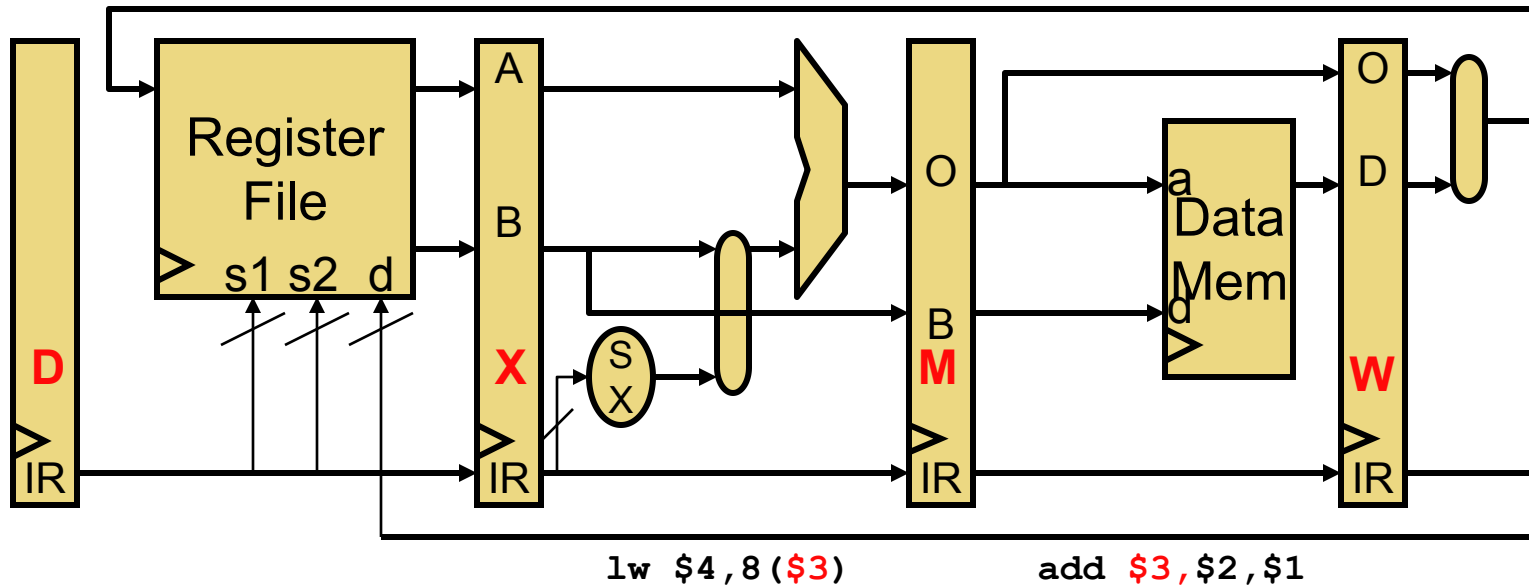
Pipeline Control Terminology

- Hardware interlock maneuver is called **stall** or **bubble**
- Mechanism is called **stall logic**
- Part of more general **pipeline control** mechanism
 - Controls the advance of insns through pipeline
- Distinguish from **pipelined datapath control**
 - Controls datapath **within** each stage
 - **Pipeline control** controls advancement of **datapath control**

Hardware Interlock Performance

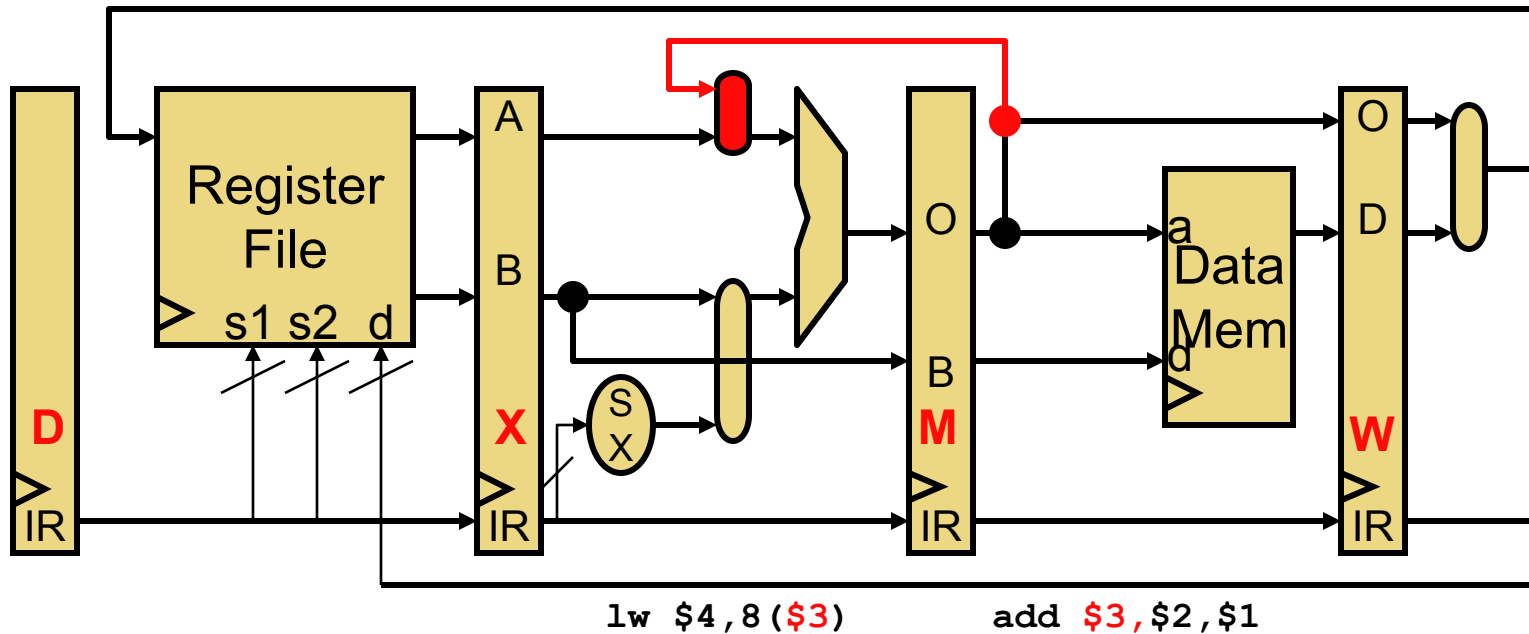
- As before:
 - Branch: 20%, load: 20%, store: 10%, other: 50%
- Hardware interlocks: same as software interlocks
 - 20% of insns require 1 cycle stall (I.e., insertion of 1 `nop`)
 - 5% of insns require 2 cycle stall (I.e., insertion of 2 `nops`)
 - $\text{CPI} = 1 + 0.20 \times 1 + 0.05 \times 2 = \mathbf{1.3}$
 - So, either CPI stays at 1 and #insns increases 30% (software)
 - Or, #insns stays at 1 (relative) and CPI increases 30% (hardware)
 - Same performance ☹ but better software compatibility ☺
- 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`

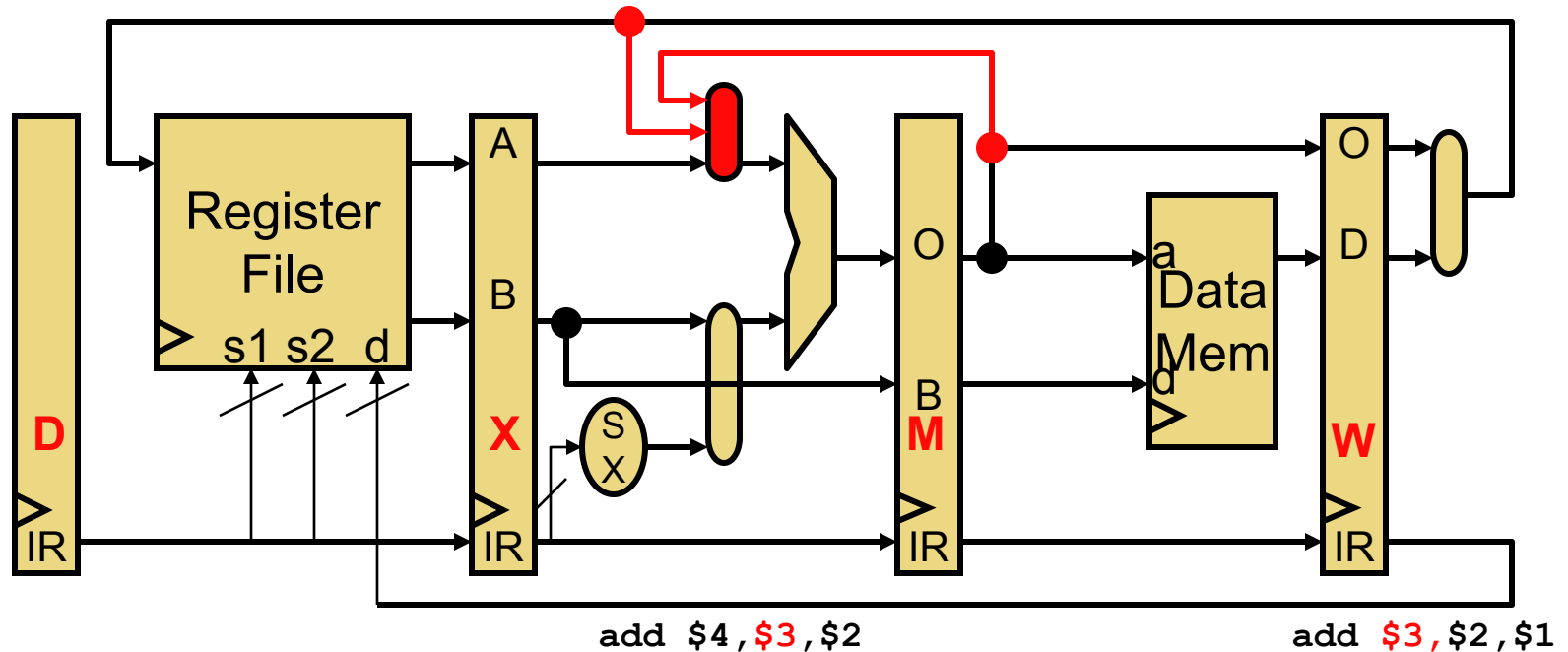
Bypassing



- **Bypassing**

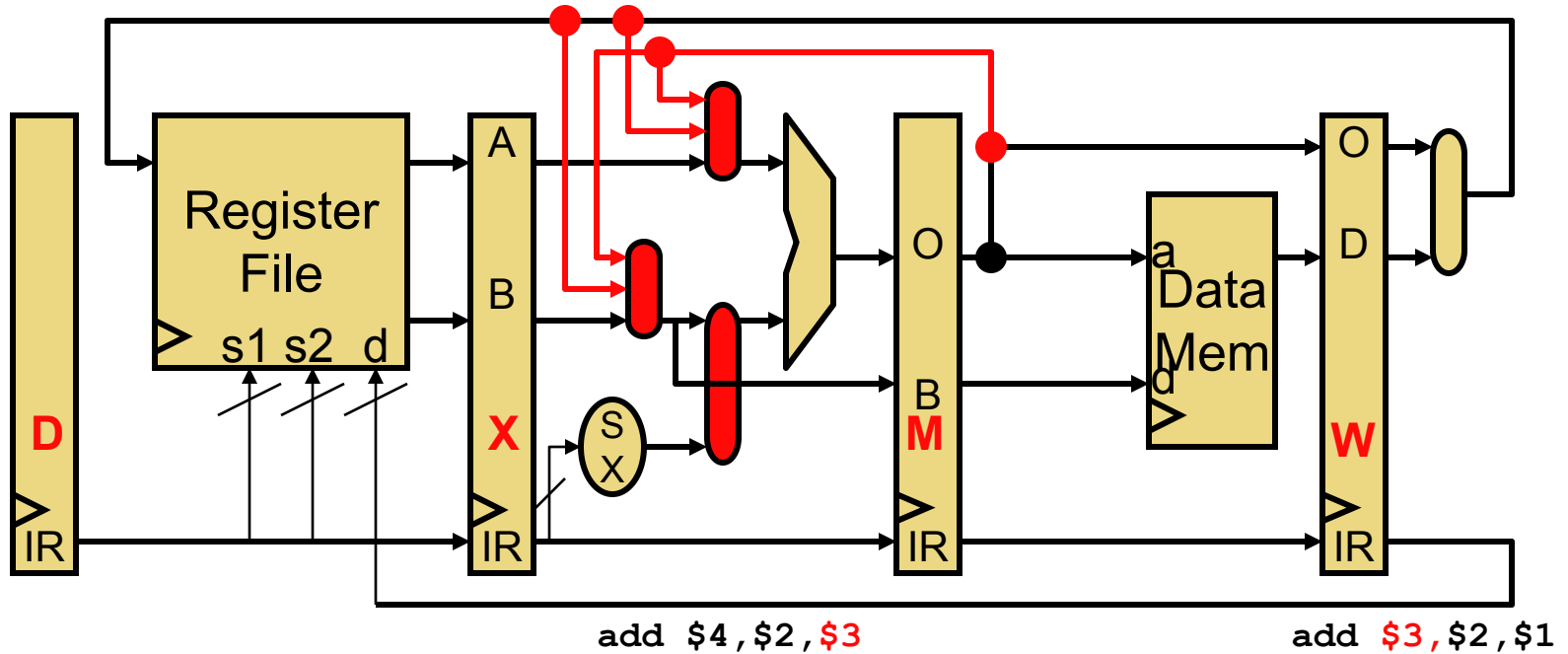
- Reading a value from an intermediate source
- Not waiting until it is available from primary source
- Here, we are bypassing the register file
- Also called **forwarding**

WX Bypassing



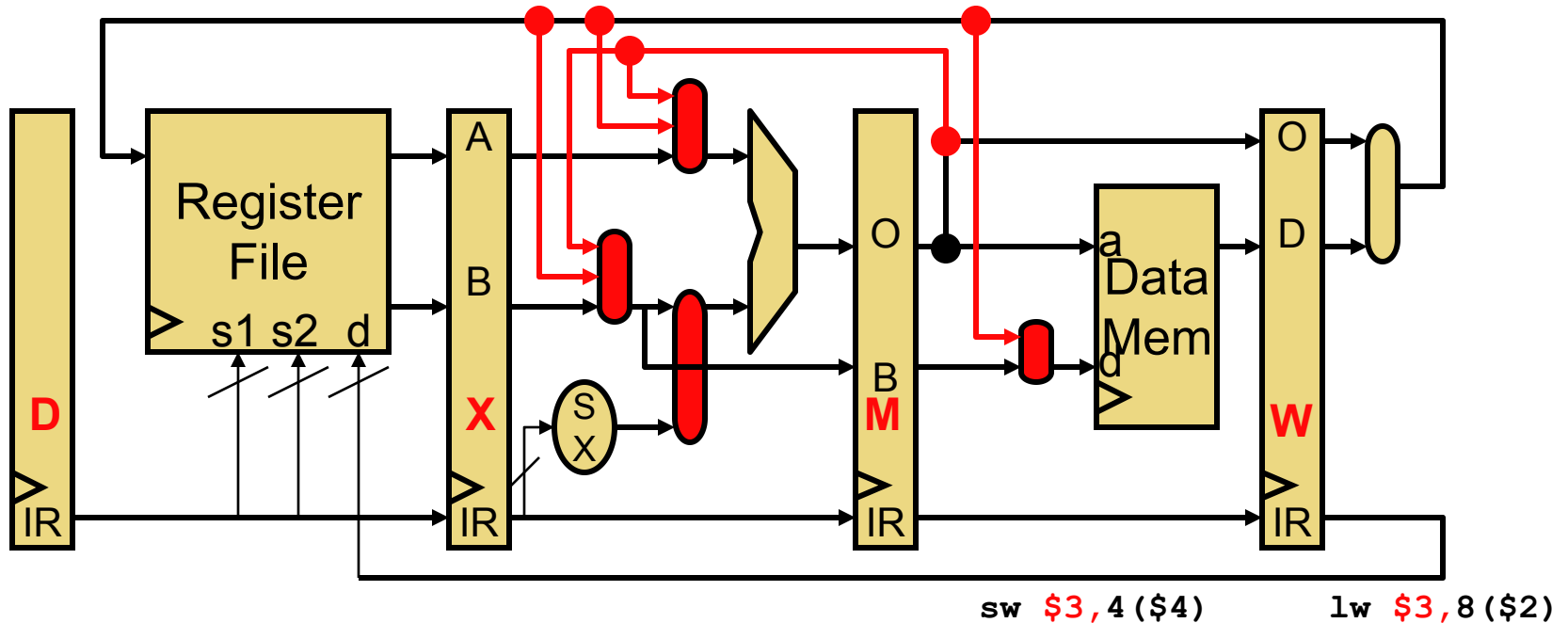
- What about this combination?
 - Add another bypass path and MUX (multiplexor) input
 - First one was an **MX** bypass
 - This one is a **WX** bypass

ALUinB Bypassing



- Can also bypass to ALU input B

WM Bypassing?



- Does WM bypassing make sense?
 - Not to the address input (why not?)

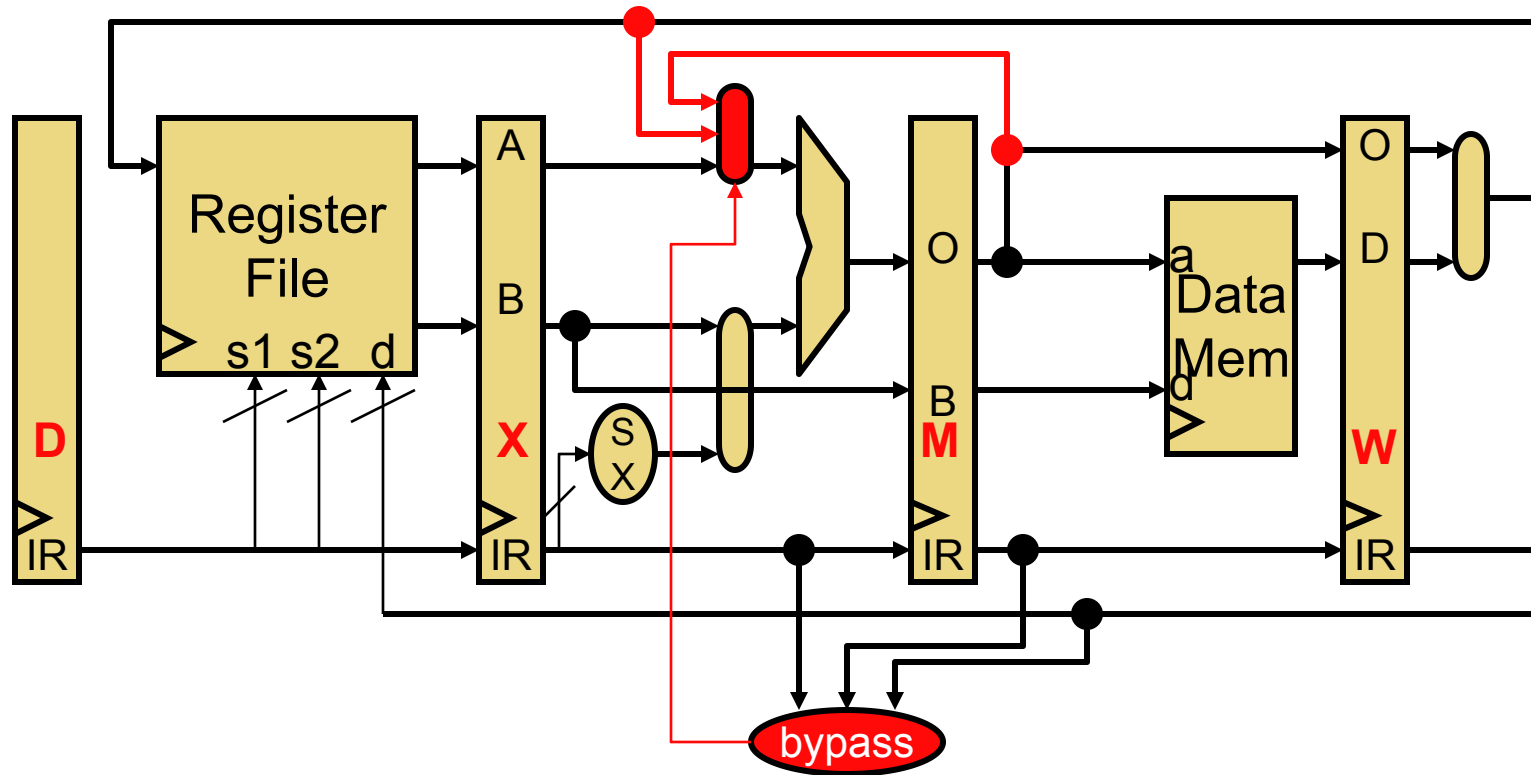
`sw $4, 4 ($3)` `lw $3, 8 ($2)`

X

- But to the store data input, yes

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

Bypass Logic

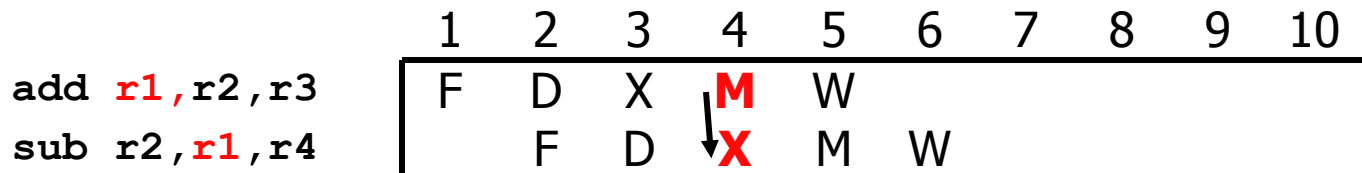


- Each multiplexor has its own, here it is for "ALUinA"
 $(X.IR.RegSrc1 == M.IR.RegDest) \Rightarrow 0$
 $(X.IR.RegSrc1 == W.IR.RegDest) \Rightarrow 1$
Else $\Rightarrow 2$

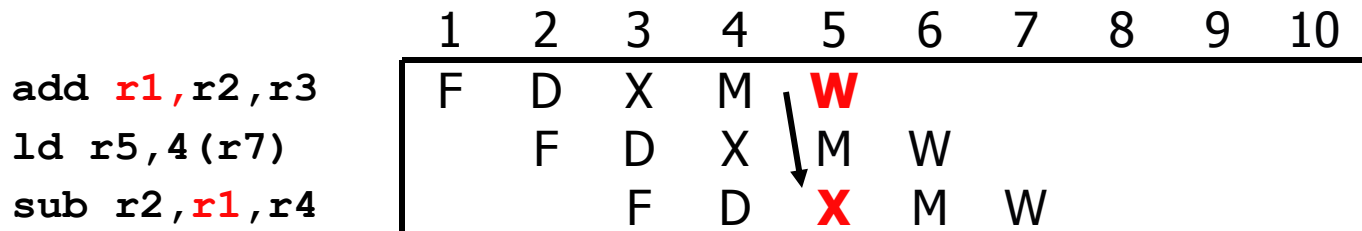
Pipeline Diagrams with Bypassing

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

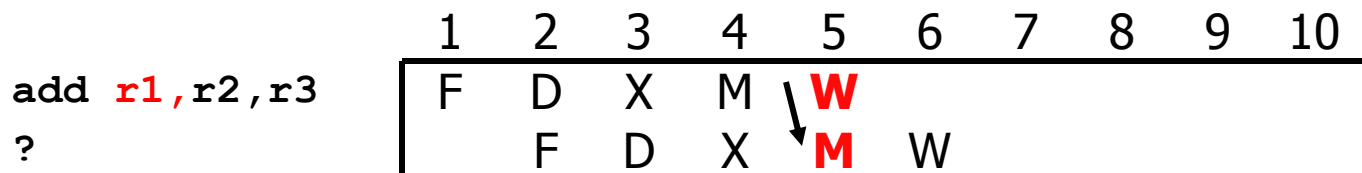
- Example: MX bypass



- Example: WX bypass



- Example: WM bypass

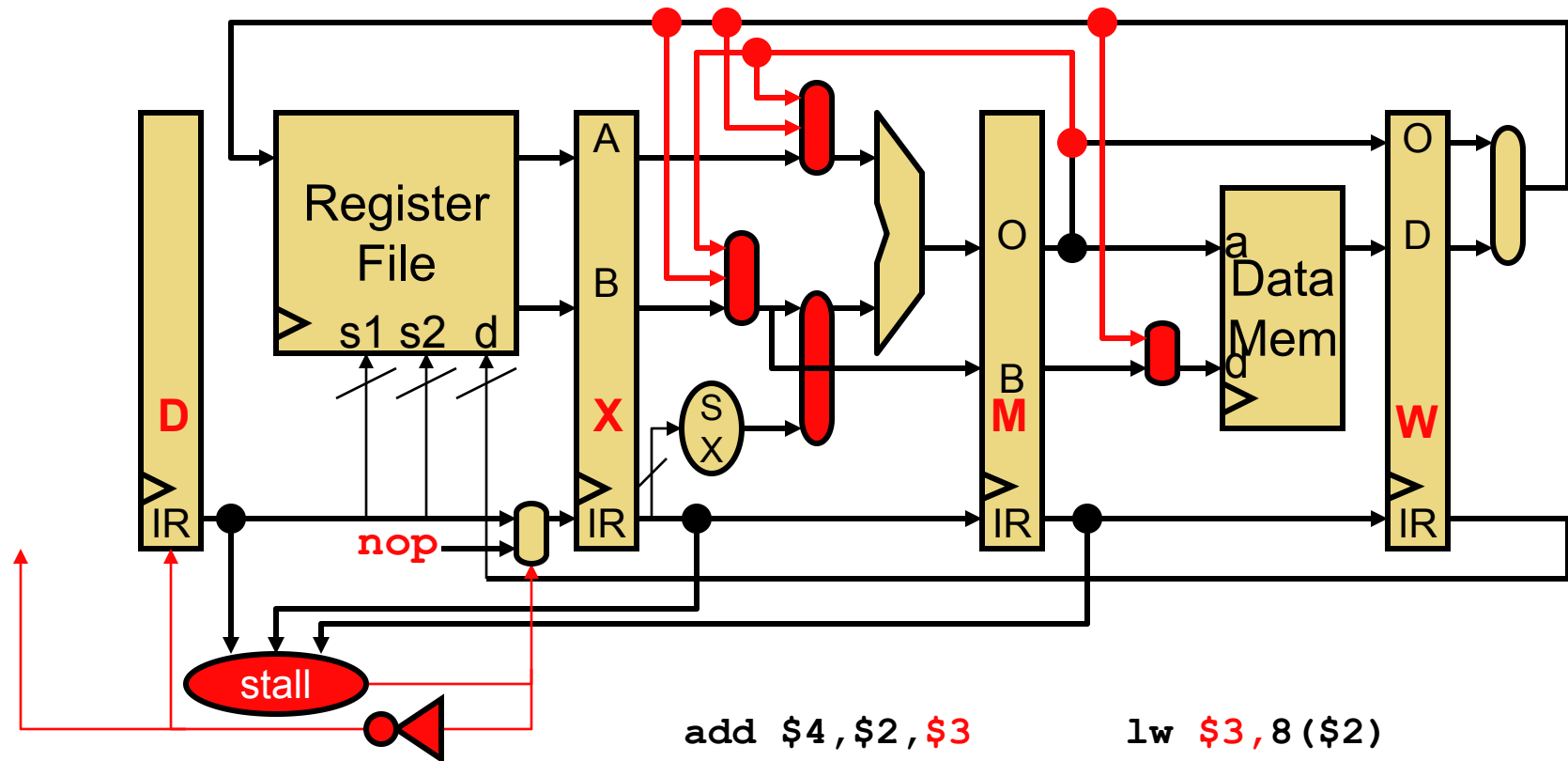


- Can you think of a code example that uses the WM bypass?

Bypass and Stall Logic

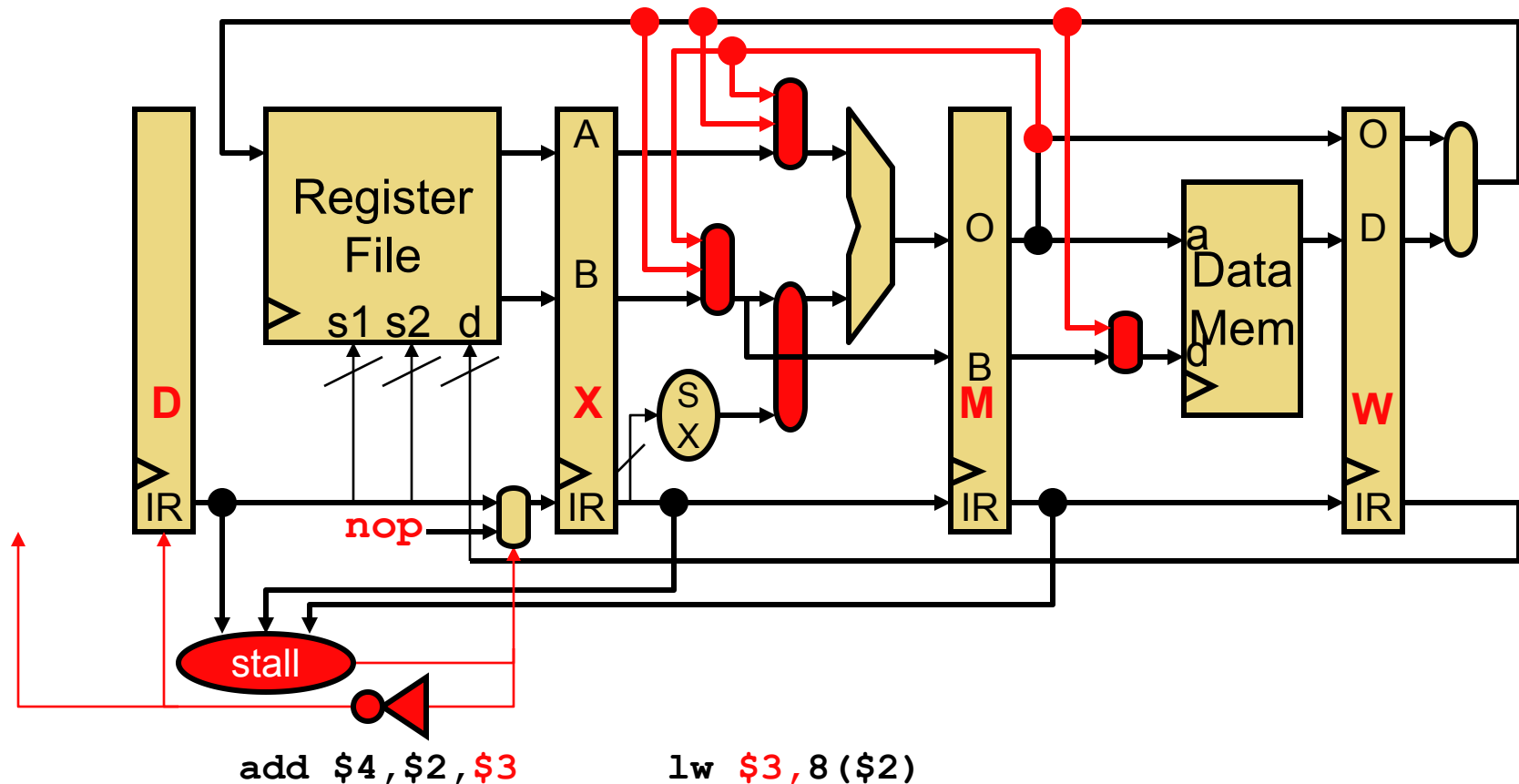
- Two separate things
 - Stall logic controls pipeline registers
 - Bypass logic controls multiplexors
- But complementary
 - For a given data hazard: if you can't bypass, must stall
- Previous slide shows **full bypassing**: all bypasses possible
 - Have we prevented all data hazards? (Thus obviating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

Have We Prevented All Data Hazards?



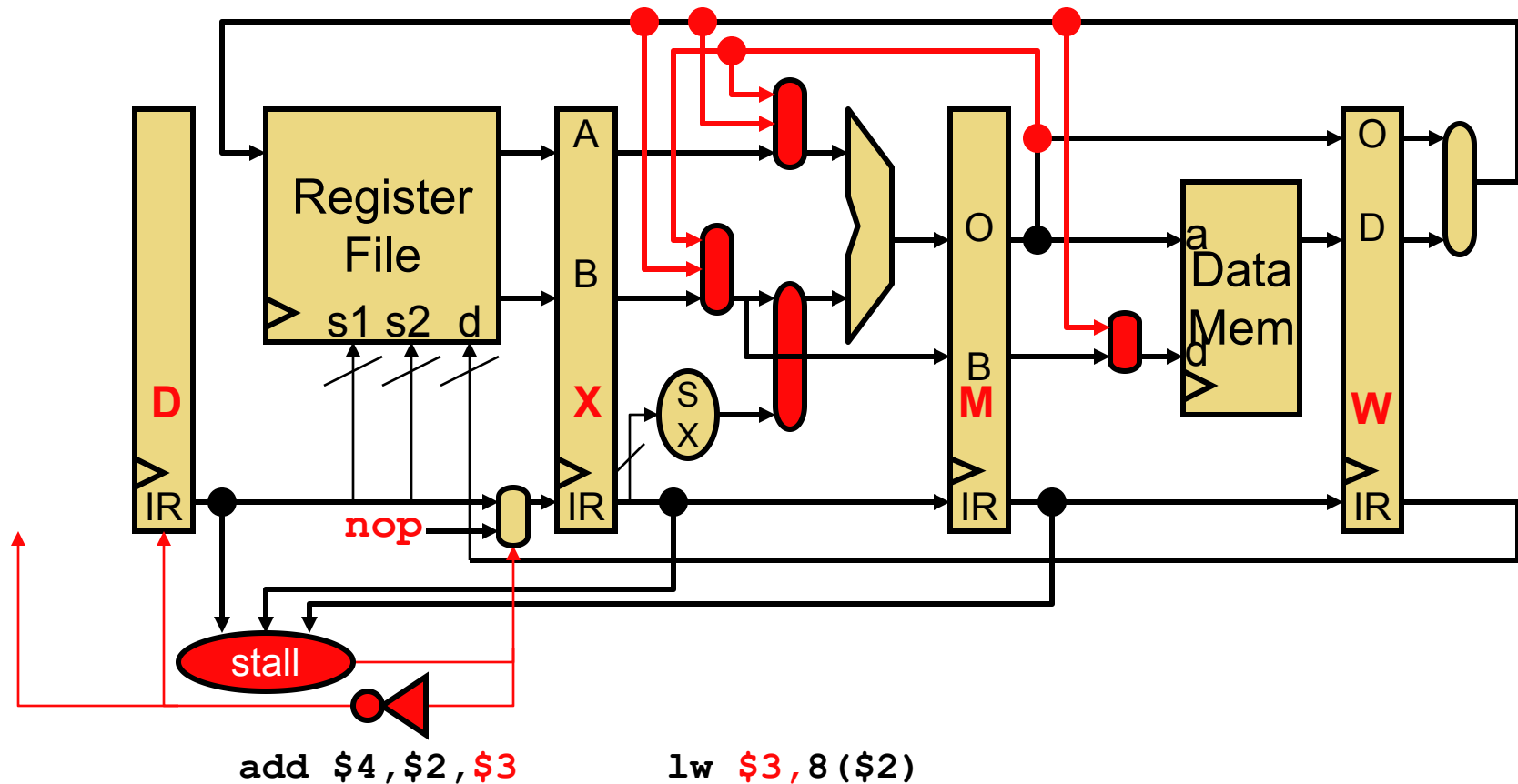
- No. Consider a “load” followed by a dependent “add” insn
- Bypassing alone isn’t sufficient!
- **Hardware solution**: detect this situation and inject a stall cycle
- **Software solution**: ensure compiler doesn’t generate such code

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

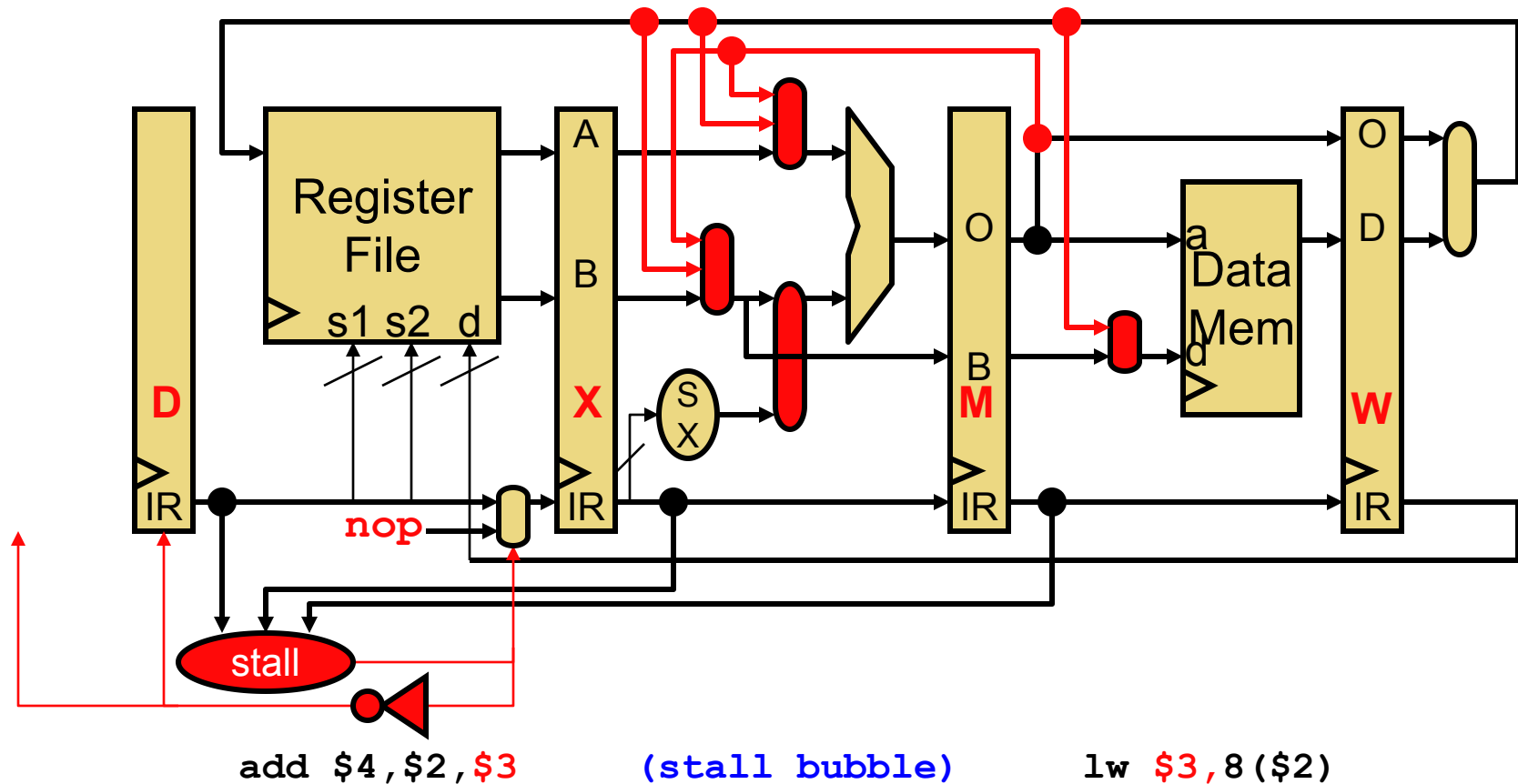
Stalling on Load-To-Use Dependences



```

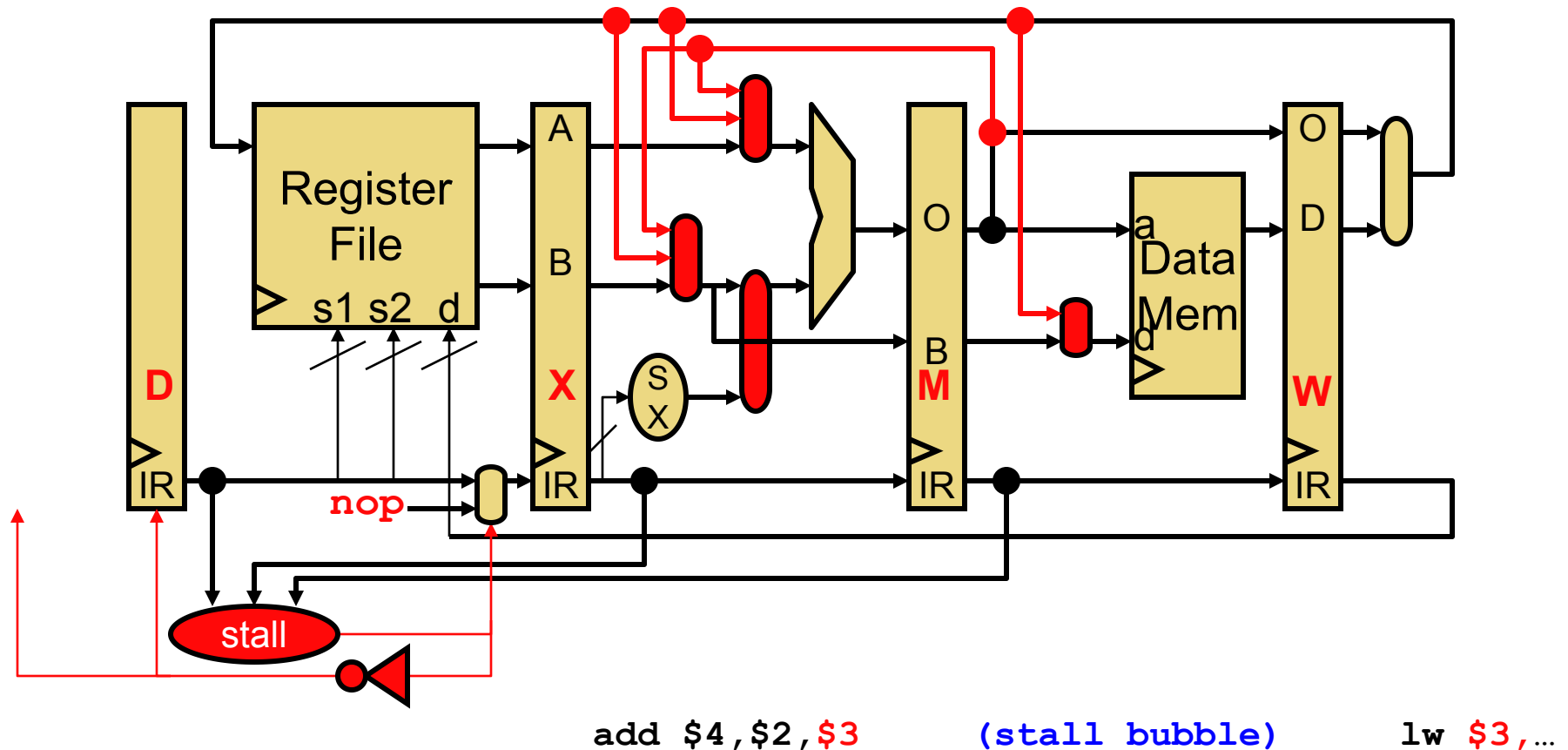
Stall = (X.IR.Operation == LOAD) &&
        ( (D.IR.RegSrc1 == X.IR.RegDest) ||
          ((D.IR.RegSrc2 == X.IR.RegDest) && (D.IR.Op != STORE))
        )
    
```

Stalling on Load-To-Use Dependences



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



```

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
 - $\text{CPI} = 1 + (1 * 20\% * 50\%) = \mathbf{1.1}$

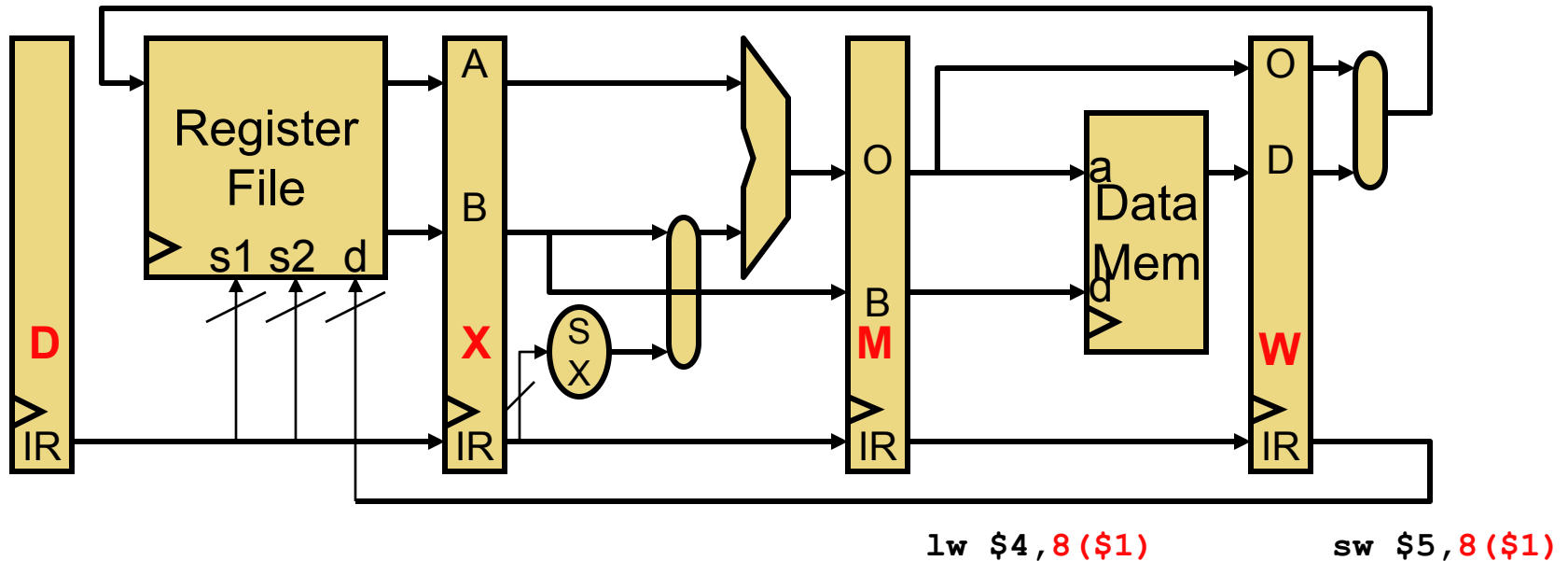
Reducing Load-Use Stall Frequency

	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*	D	X	M	W

- **d*** shows stall, addi writes into X register only in cycle 5
- 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	

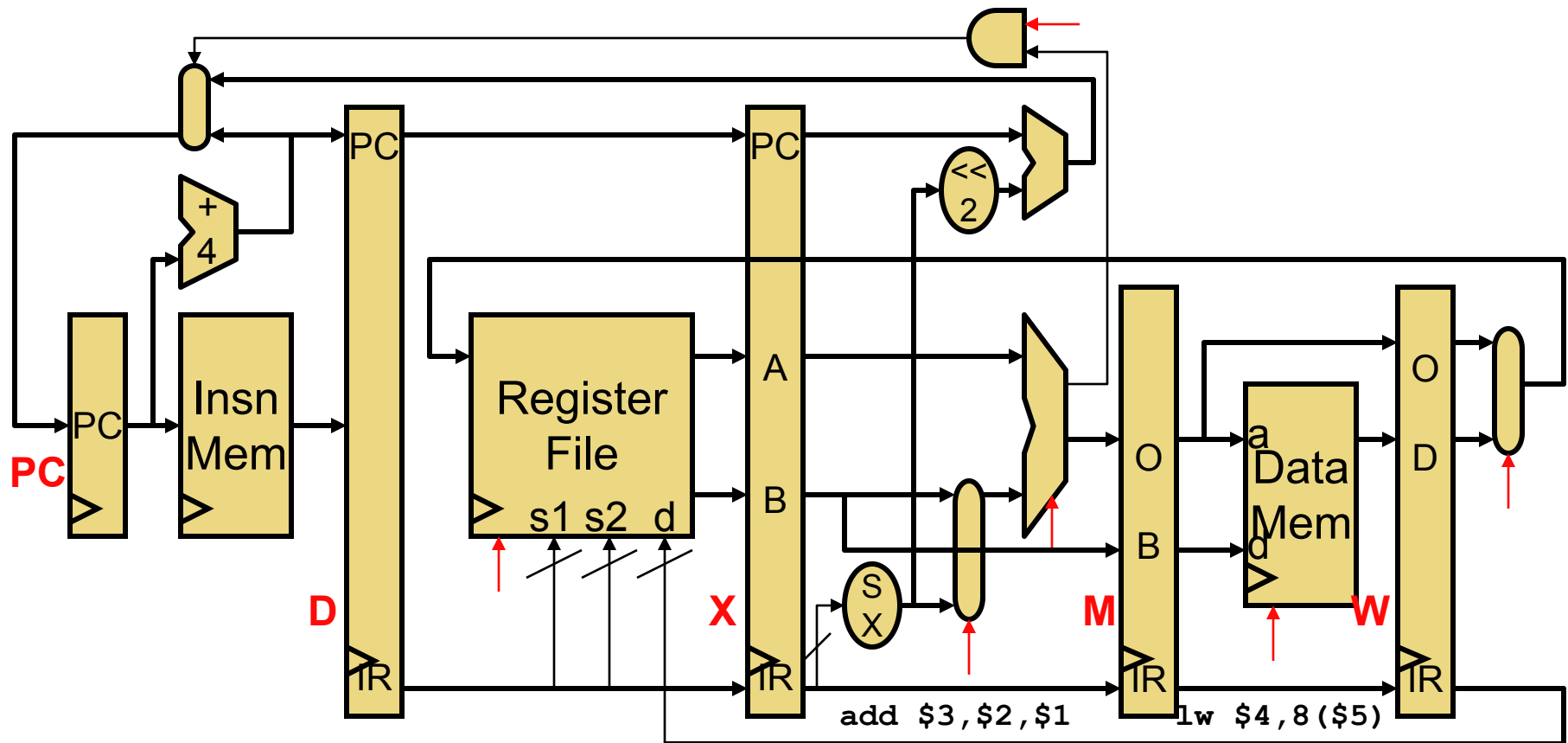
Dependencies Through Memory



- Are “load to store” memory dependencies a problem?
 - Nope
 - `lw` following `sw` to same address in next cycle, gets right value
 - Why? Data mem read/write always take place in same stage

Multi-Cycle Operations

Why Does Every Insn Take 5 Cycles?

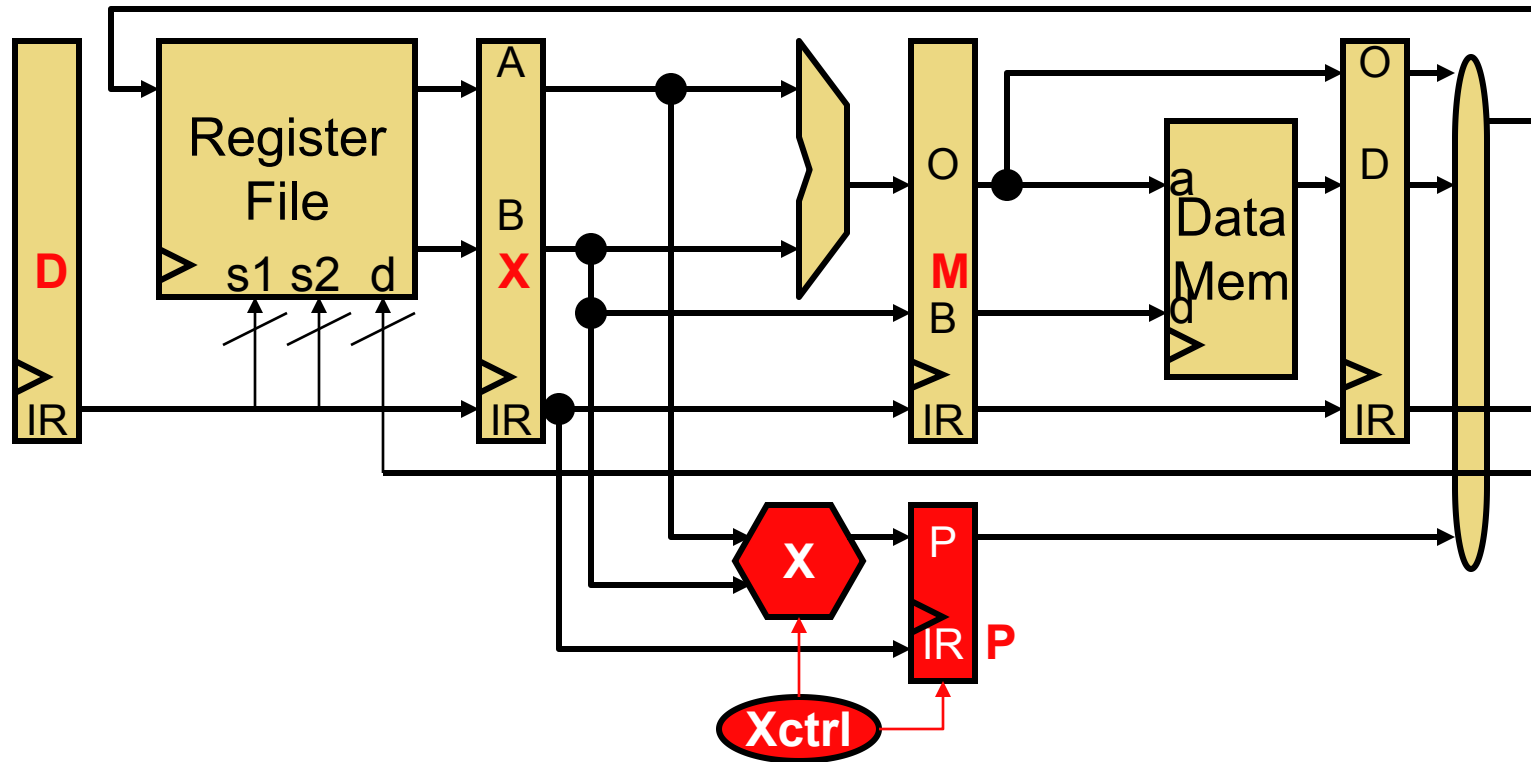


- Could/should we allow **add** to skip **M** and go to **W**? No
 - It wouldn't help: peak fetch still only 1 insn per cycle
 - **Structural hazards**: imagine **add** after **lw** (only 1 reg. write port)

Structural Hazards

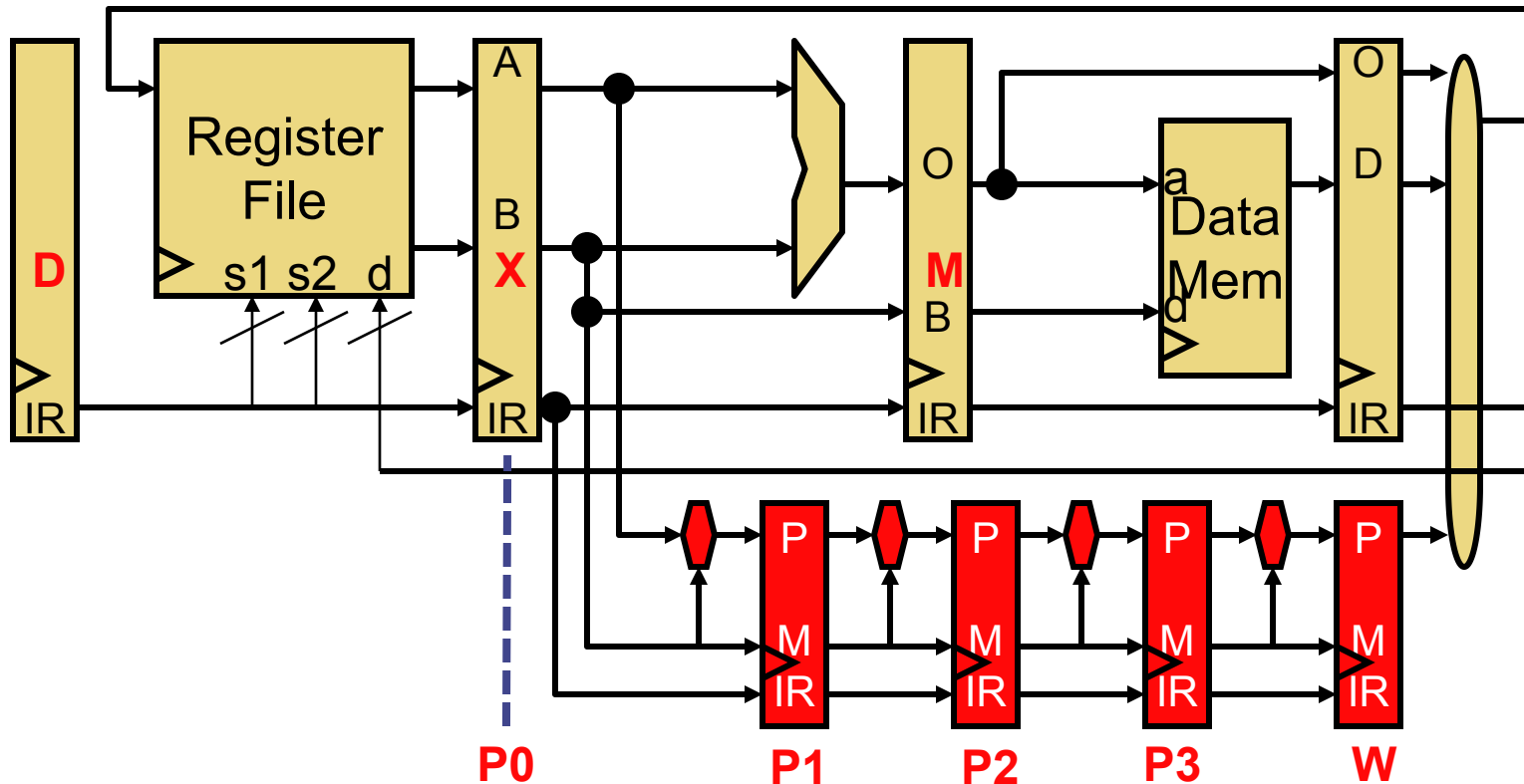
- **Structural hazards**
 - Two insns trying to use same circuit at same time
 - E.g., structural hazard on register file write port
- **To avoid structural hazards**
 - Avoided if:
 - Each insn uses every structure exactly once
 - For at most one cycle
 - All instructions travel through all stages
 - 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

Pipelining and Multi-Cycle Operations



- What if you wanted to add a multi-cycle operation?
 - E.g., 4-cycle multiply
 - **P**: separate output register 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 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			

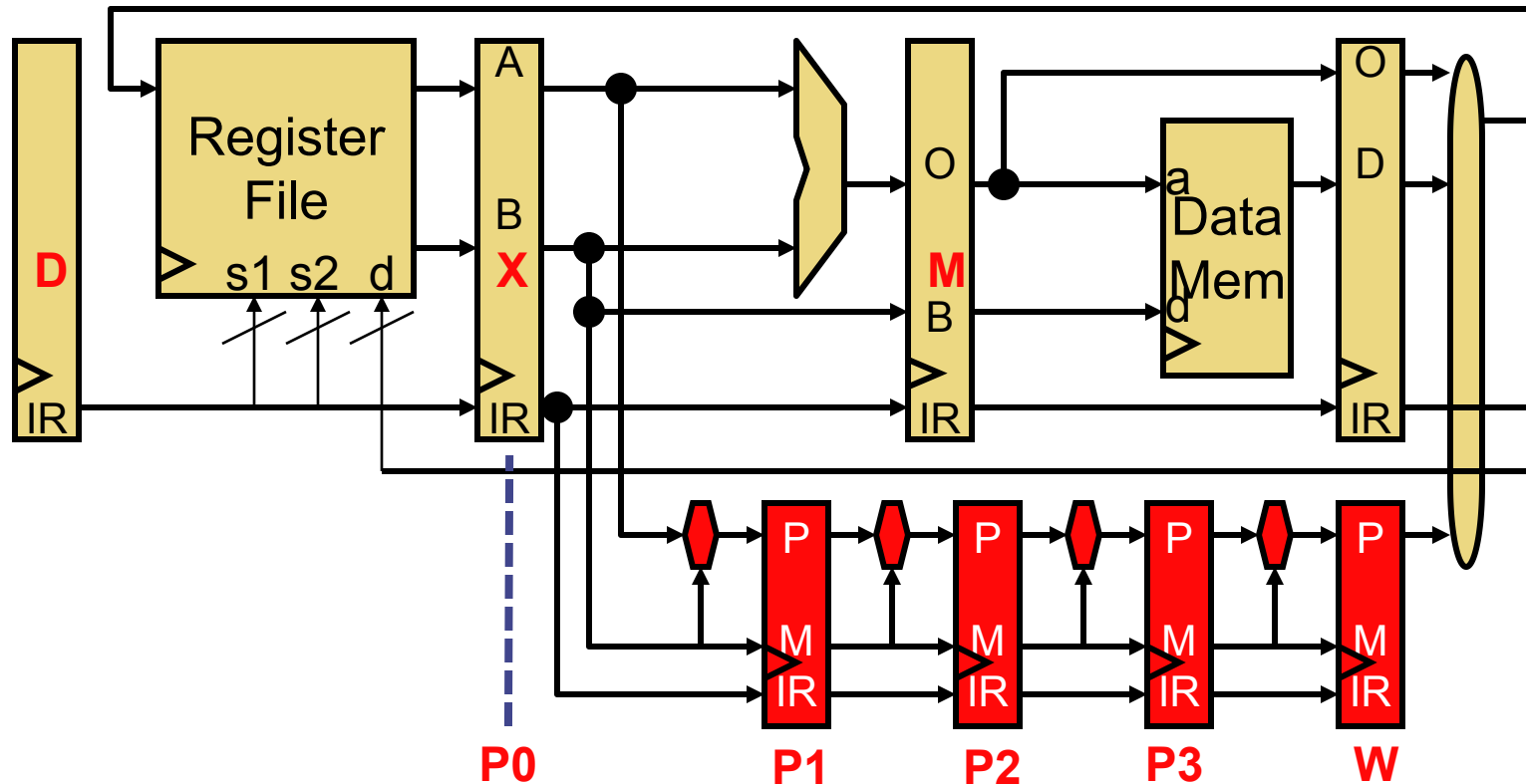
- Even allow **independent multiply** instructions

	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?



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

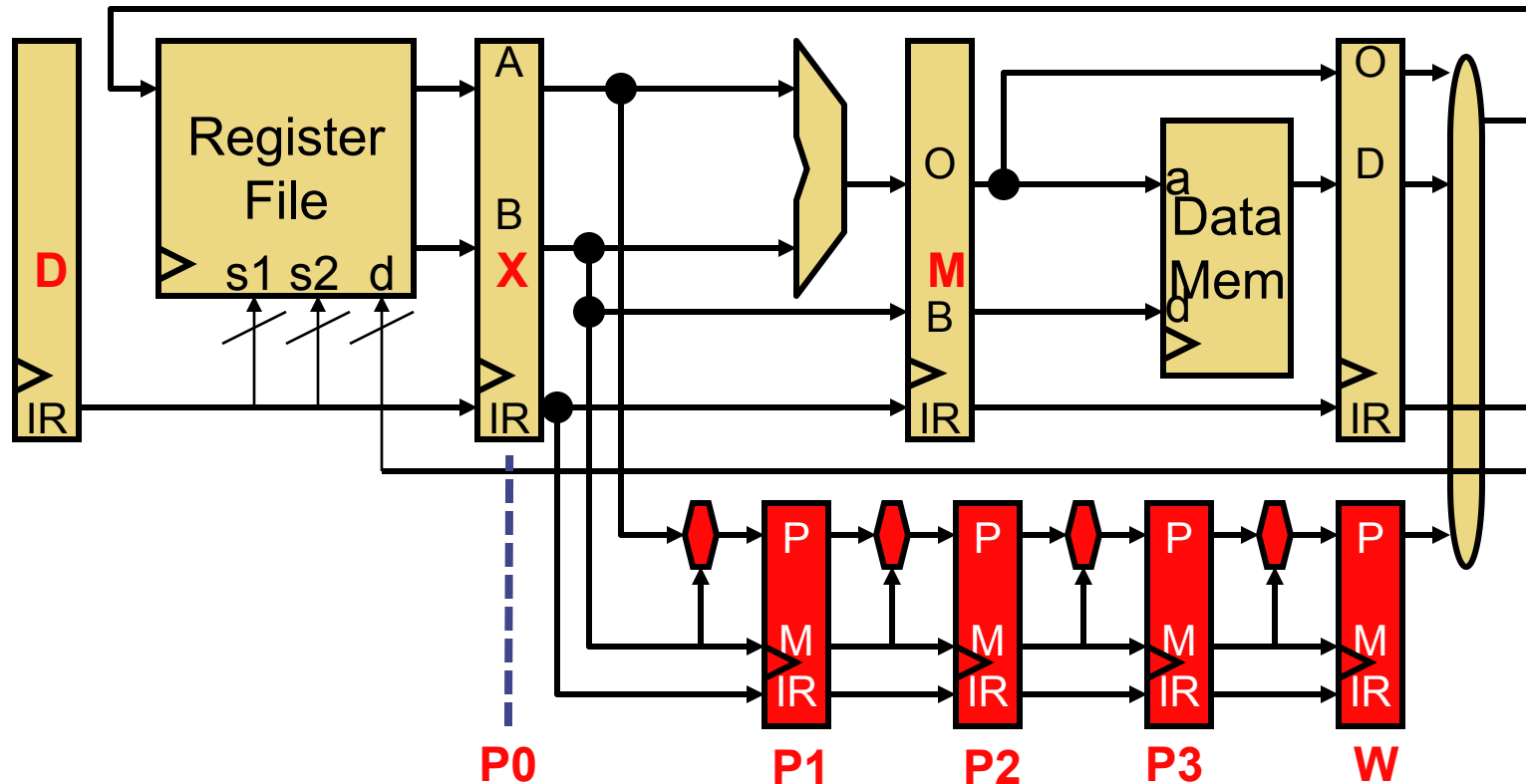
- What about...
 - Two instructions trying to write register file in same cycle?
 - Structural hazard!
- Must prevent:

	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")

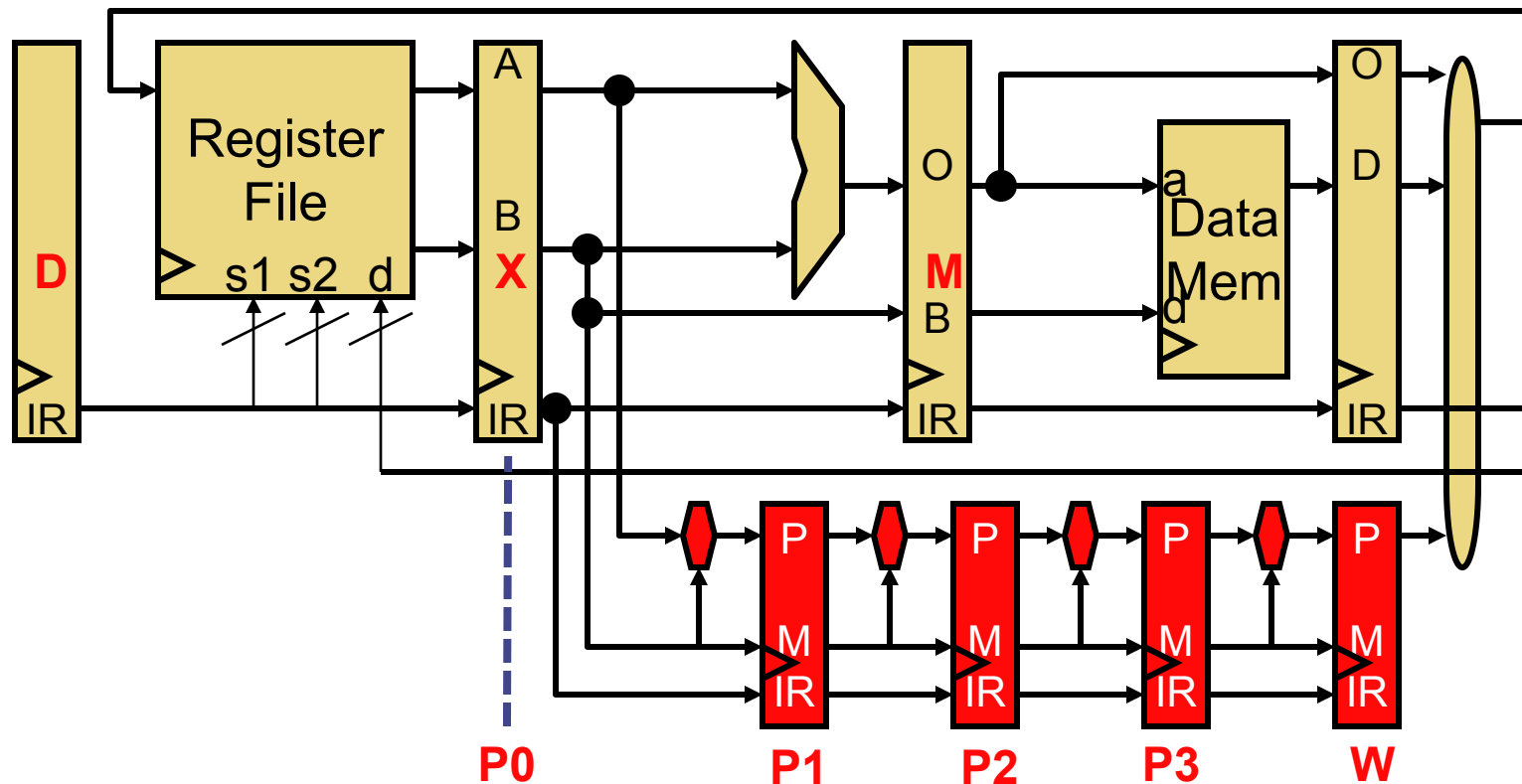
More Multiplier Nasties

- 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 corrected

Preventing Mis-Ordered Reg. Write



- Fix to problem on previous slide:

Stall = (OldStallLogic) ||

**((D.IR.RegDest == P0.IR.RegDest) ||
(D.IR.RegDest == P1.IR.RegDest))**

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 registers 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

Pipeline Depth

Principles of Pipelining

- Let: insn execution require **N** stages, each takes t_n time
 - L_1 (1-insn latency) = $\sum t_n$
 - **T** (throughput) = $1/L_1$
 - L_M (M-insn latency, where $M \gg 1$) = $M * L_1$
- Now: N-stage pipeline
 - $L_{1+P} = L_1$
 - $T_{+P} = 1/\max(t_n) \leq \mathbf{N}/L_1$
 - If t_n are equal (i.e., $\max(t_n) = L_1/N$), throughput = N/L_1
 - $L_{M+P} = M * \max(t_n) \geq M * L_1 / \mathbf{N}$
 - S_{+P} (speedup) = $[M * L_1 / (\geq M * L_1 / N)] = \leq \mathbf{N}$
- Q: for arbitrarily high speedup, use arbitrarily high N?

Pipeline Overhead

- Let: **O** be extra delay per pipeline stage
 - Register overhead: pipeline registers take time
 - Clock/data skew
- Now: N-stage pipeline with overhead
 - Assume $\max(t_n) = L_1/N$
 - $L_{1+P+O} = L_1 + N*O$
 - $T_{+P+O} = 1/(L_1/N + O) = 1/(1/T + O) \leq T, \leq \mathbf{1/O}$
 - $L_{M+P+O} = M*L_1/N + M*O = L_{M+P} + M*O$
 - $S_{+P+O} = [M*L_1 / (M*L_1/N + M*O)] = \leq N = S_{+P}, \leq \mathbf{L_1/O}$
- **O** limits throughput and speedup \rightarrow useful N

Optimizing Pipeline Depth

- Parameterize clock cycle in terms of gate delays
 - G gate delays to process (fetch, decode, execute) a single insn
 - O gate delays overhead per stage
 - X average stall per insn per stage
 - Simplistic: real X function much, much more complex
- Compute optimal N (pipeline stages) given G,O,X
 - $IPC = 1 / (1 + X * N)$
 - $freq = 1 / (G / N + O)$
 - Example: G = 80, O = 1, X = 0.16,

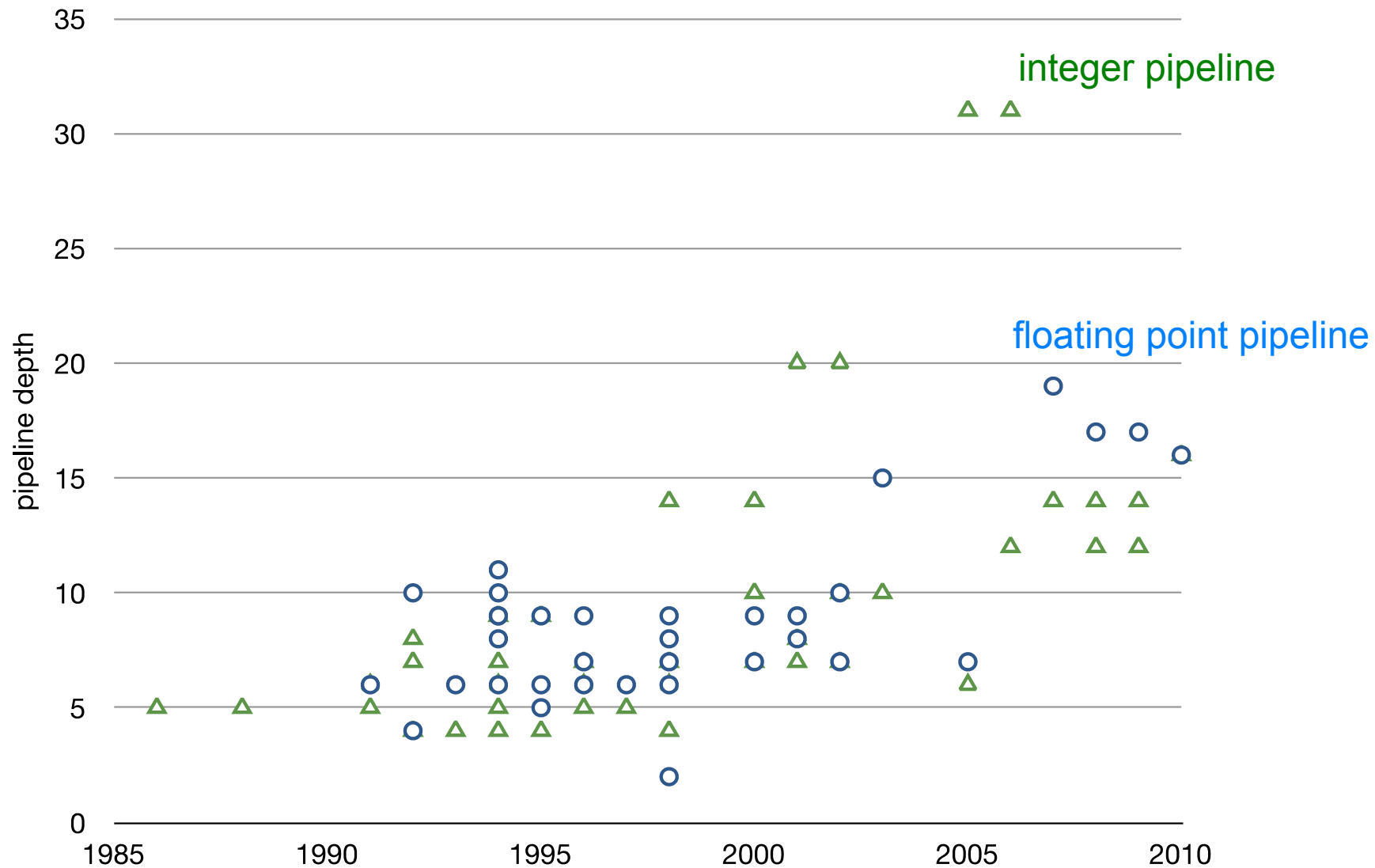
N	$IPC = 1/(1+0.16*N)$	$freq=1/(80/N+1)$	$IPC*freq$
5	0.56	0.059	0.033
10	0.38	0.110	0.042
20	0.33	0.166	0.040

Pipelining: Clock Frequency vs. IPC

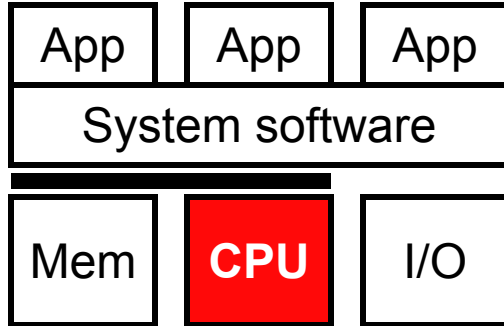
- Increase number of pipeline stages (“pipeline depth”)
 - Keep cutting datapath into finer pieces
 - + Increases clock frequency (decreases clock period)
 - **Register overhead & unbalanced stages** cause sub-linear scaling
 - Double the number of stages won’t quite double the frequency
 - Increases CPI (decreases IPC)
 - More pipeline “hazards”, higher branch penalty
 - Memory latency relatively higher (same absolute lat., more cycles)
 - Result: after some point, deeper pipelining can decrease performance
 - “Optimal” pipeline depth is program and technology specific

Pipeline Depth

data from <http://cpudb.stanford.edu/>



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