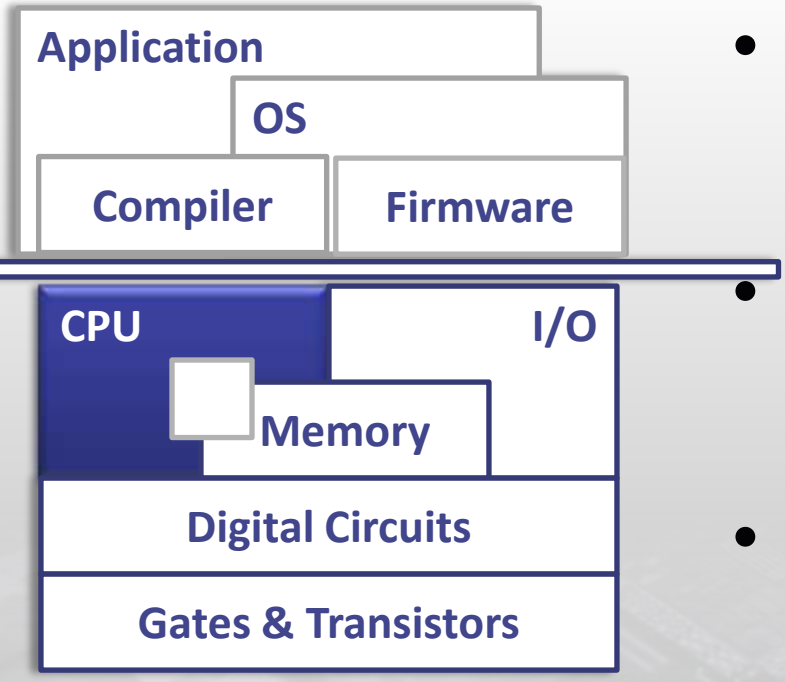
Instruction Level Parallelism I: Pipelining

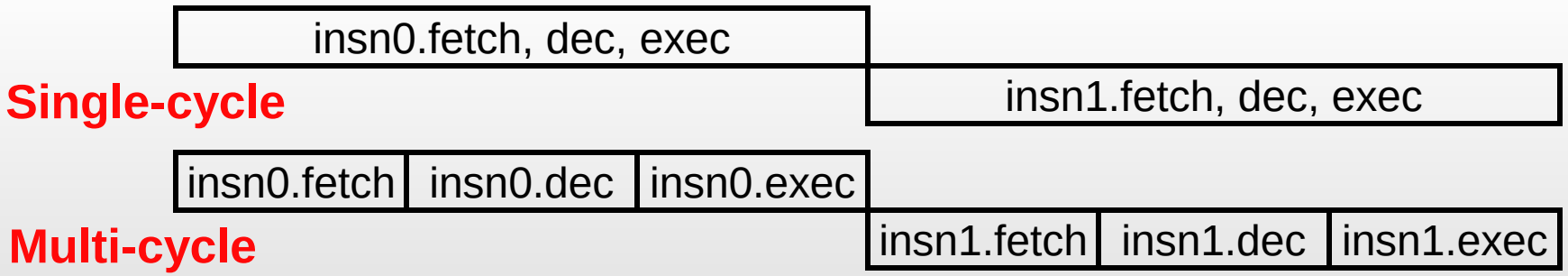
Readings: H&P Appendix A

This Unit: Pipelining



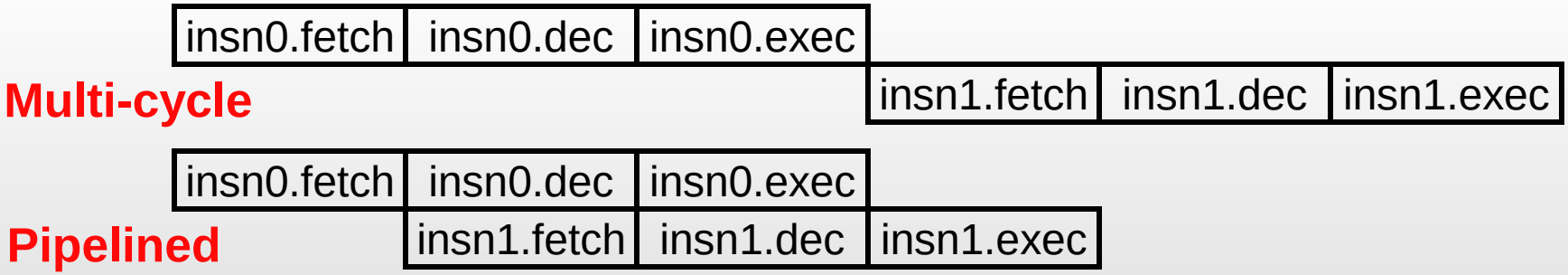
- Basic Pipelining
  - Single, in-order issue
  - Clock rate vs. IPC
- Data Hazards
  - Hardware: stalling and bypassing
  - Software: pipeline scheduling
- Control Hazards
  - Branch prediction
- Precise state

Quick Review



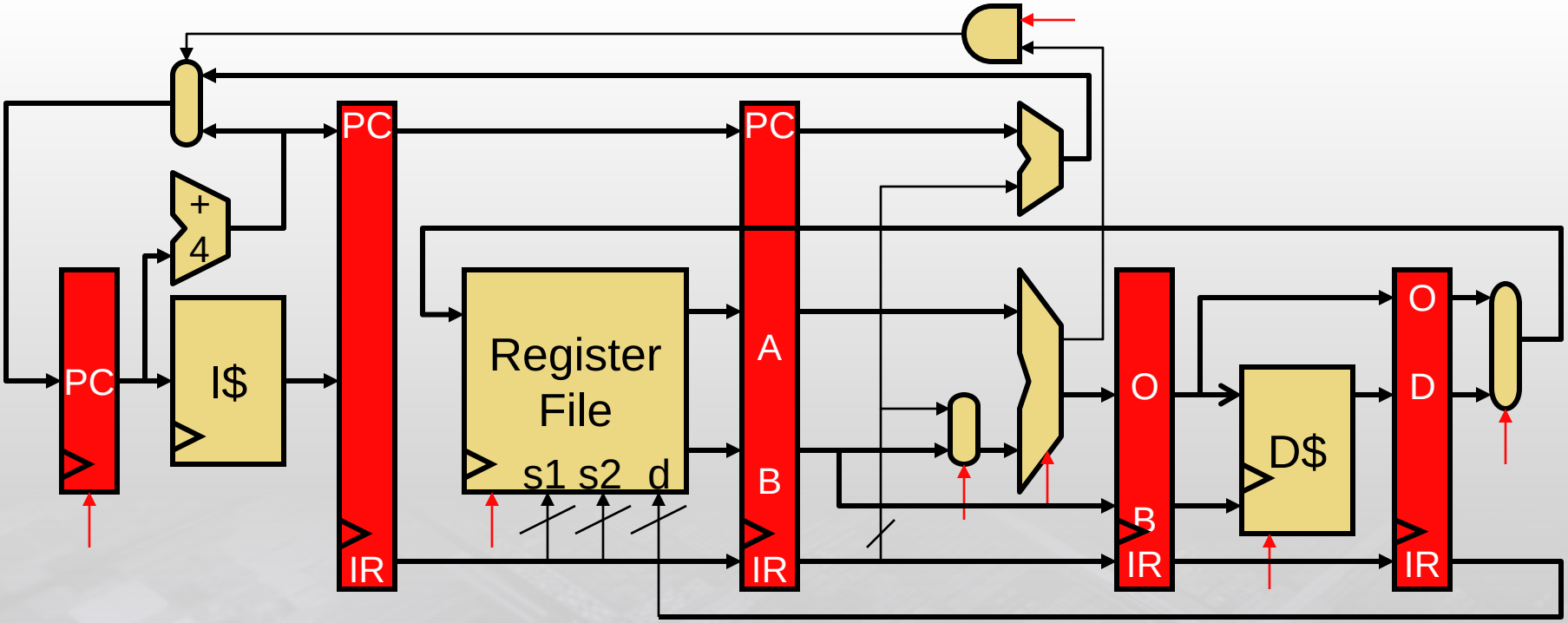
- Basic **datapath**: fetch, decode, execute
- **Single-cycle control**: hardwired
  - \+ Low CPI (1)
  - – Long clock period (to accommodate slowest instruction)
- **Multi-cycle control**: micro-programmed
  - \+ Short clock period
  - – High CPI
- Can we have both low CPI and short clock period?
  - Not if datapath executes only one instruction at a time
  - No good way to make a single instruction go faster

Pipelining



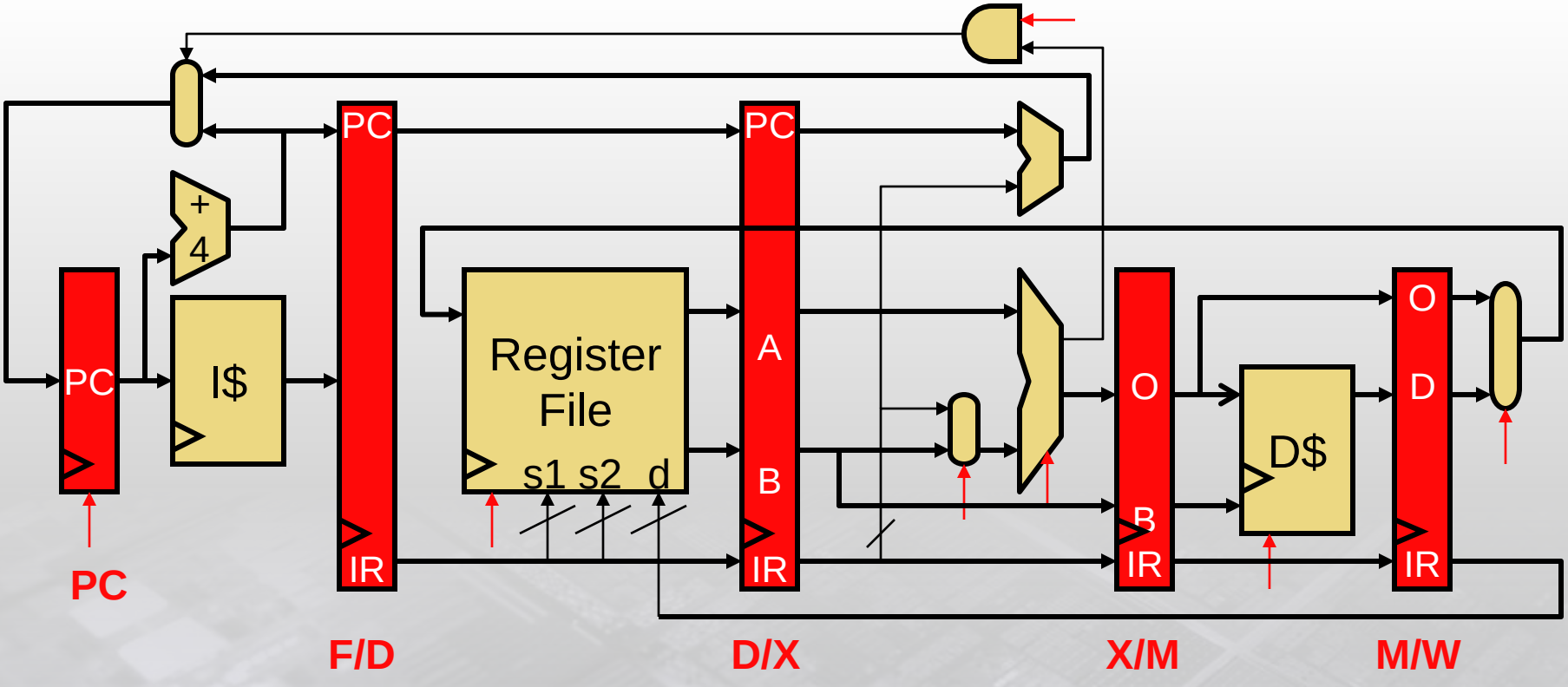
- Important performance technique
  - **Improves instruction throughput rather instruction latency**
- Begin with multi-cycle design
  - When instruction advances from stage 1 to 2
  - Allow next instruction to enter stage 1
  - Form of parallelism: “insn-stage parallelism”
  - Individual instruction takes the same number of stages
  - \+ **But instructions enter and leave at a much faster rate**
- Automotive assembly line analogy

5 Stage Pipelined Datapath



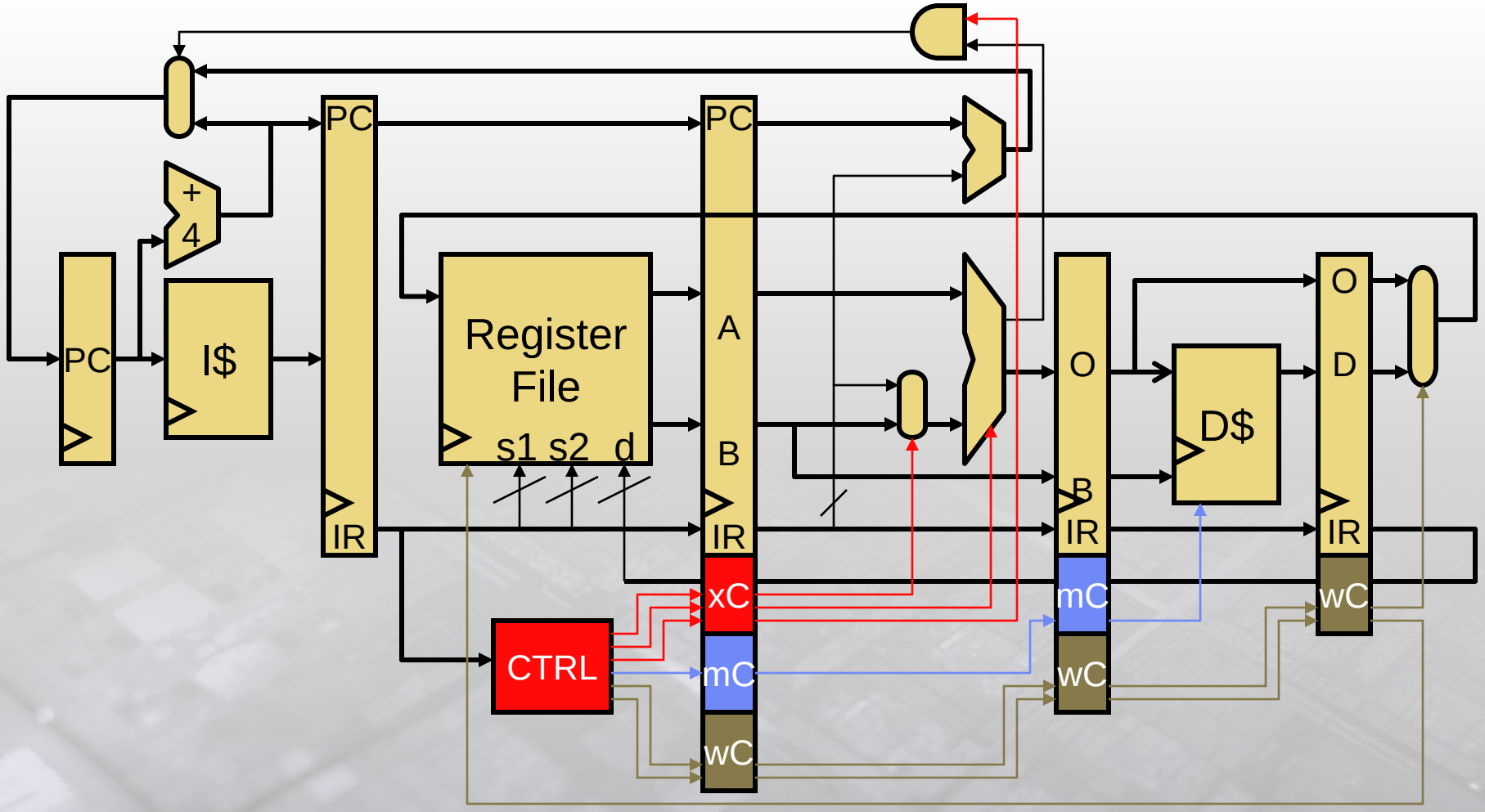
- Temporary values (PC,IR,A,B,O,D) re-latched every stage
  - Why? 5 insns may be in pipeline at once, they share a single PC?
  - Notice, PC not latched after ALU stage (why not?)

Pipeline Terminology



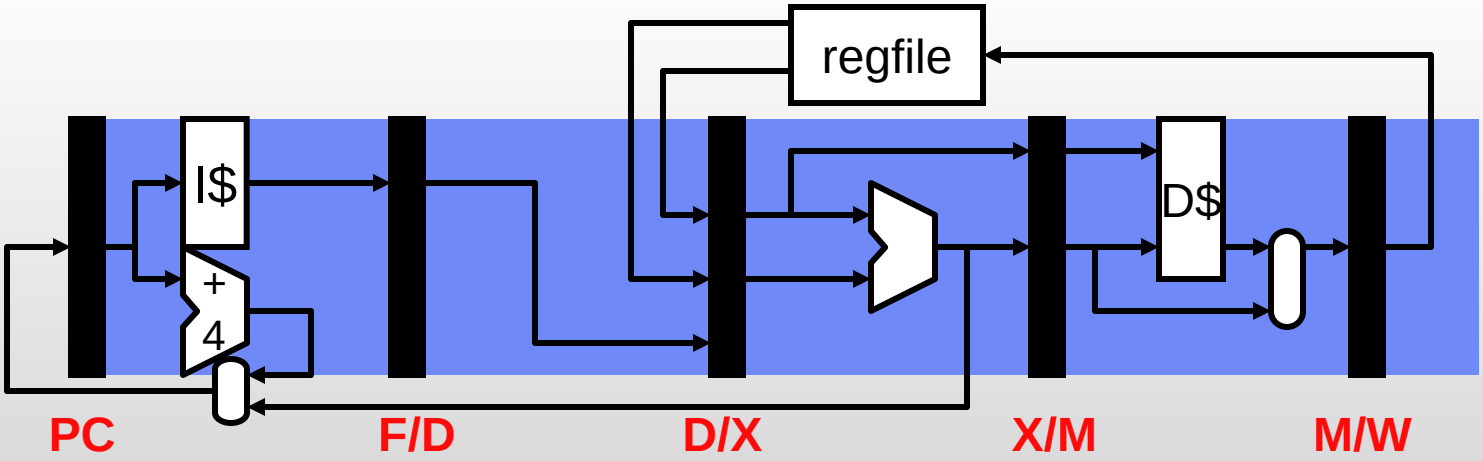
- Five stage: **F**etch, **D**ecode, e**X**ecute, **M**emory, **W**riteback
  - Nothing magical about the number 5 (Pentium 4 has 22 stages)
- Latches (pipeline registers) named by stages they separate
  - **PC**, **F/D**, **D/X**, **X/M**, **M/W**

Pipeline Control



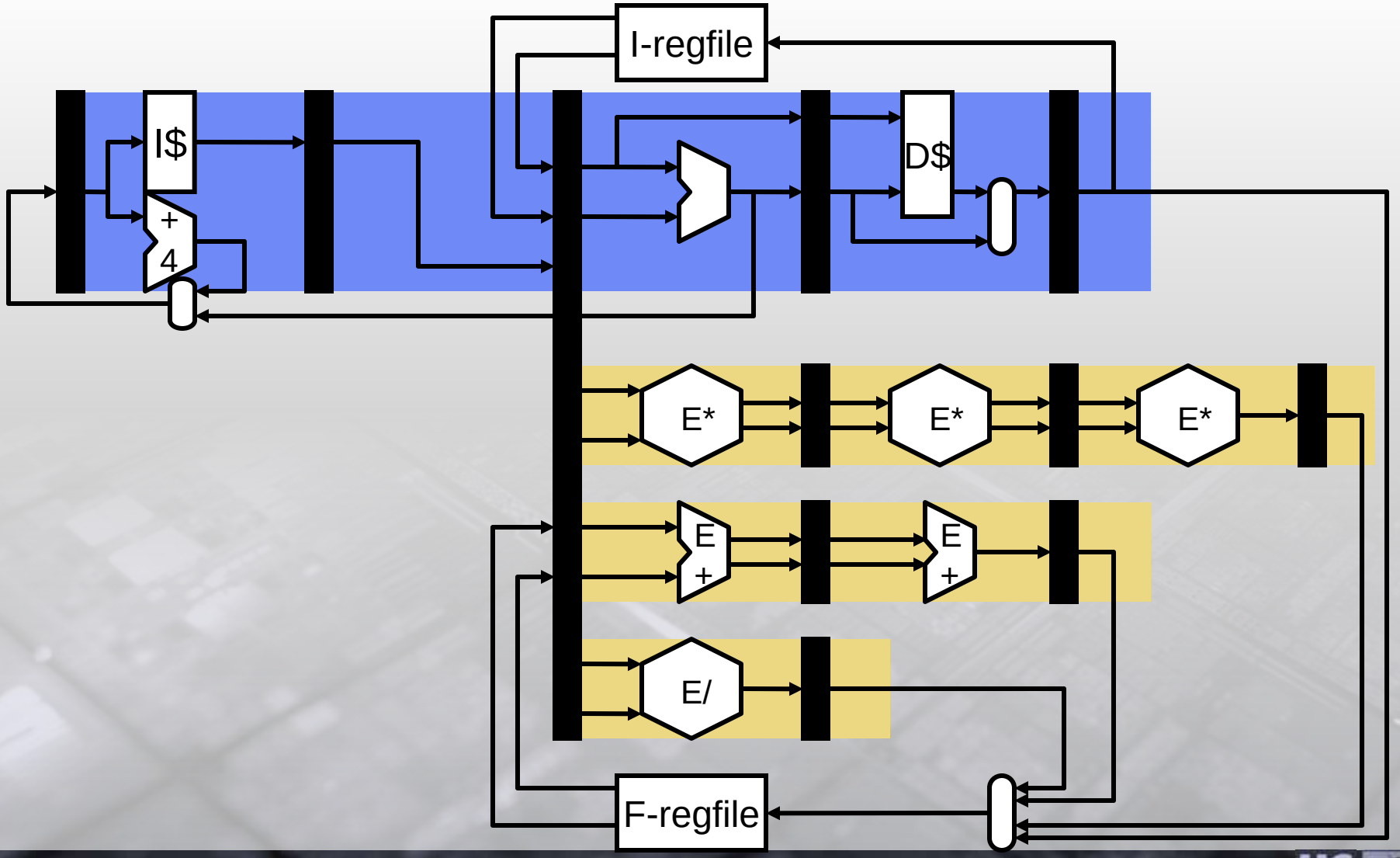
- One single-cycle controller, but pipeline the control signals

Abstract Pipeline



- This is an **integer pipeline**
  - Execution stages are X,M,W
- Usually also one or more **floating-point (FP) pipelines**
  - Separate FP register file
  - One "pipeline" per functional unit: E+, E*, E/
    - "Pipeline": functional unit need not be pipelined (e.g, E/)
  - Execution stages are E+,E+,W (no M)

Floating Point Pipelines



Pipeline Diagram

	1	2	3	4	5	6	7	8	9
`add r3,r2,r1`	F	D	X	M	W				
`ld r4,0(r5)`		F	D	**X**	M	W			
`st r6,4(r7)`			F	D	X	M	W		

- **Pipeline diagram**
  - Cycles across, insns down
  - Convention: **X** means **`ld r4,0(r5)`** finishes execute stage and writes into X/M latch at end of cycle 4
- Reverse stream analogy
  - "Downstream": earlier stages, younger insns
  - "Upstream": later stages, older insns
  - Reverse? instruction stream fixed, pipeline flows over it
    - Architects see instruction stream as fixed by program/compiler

Pipeline Performance Calculation

- Back of the envelope calculation
  - Branch: 20%, load: 20%, store: 10%, other: 50%
- Single-cycle
  - Clock period = 50ns, CPI = 1
  - Performance = 50ns/insn
- Pipelined
  - Clock period = **12ns**
  - CPI = **1** (each insn takes 5 cycles, but 1 completes each cycle)
  - Performance = **12ns/insn**

Principles of Pipelining

- Let: insn execution require **N** cycles, each takes t_n seconds
  - $\mathbf{L_1}$ (1-insn latency) = $\sum t_n$
  - **T** (throughput) = $1/L_1$
  - $\mathbf{L_M}$ (M-insn latency, where M>>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?

No, Part I: Pipeline Overhead

- Let: **O** be extra delay per pipeline stage
  - Latch overhead: pipeline latches take time
  - Clock/data skew

- Now: N-stage pipeline with overhead
  - Assume $\max(t_n) = L_1/N$
  - $L_{1+P+O} = L_1 + N*O$
  - $T_{+P+O} = 1/(L_1/N + O) = 1/(1/T_{+P} + O) \leq T_{+P}$, **$\leq 1/O$**
  - $L_{M+P+O} = M*L_1/N + M*O = L_{M+P} + M*O$
  - $S_{+P+O} = [M*L_1 / (M*L_1/N + M*O)] = \leq N = S_{+P}$, **$\leq L_1/O$**

- **O** limits throughput and speedup $\rightarrow$ useful N

No, Part II: Hazards

- **Dependence**: relationship that serializes two insns
  - **Structural**: two insns want to use same structure
  - **Data**: two insns use same storage location
  - **Control**: one instruction affects whether another executes at all
- **Hazard**: dependence and both insns in pipeline together
  - Possibility for getting order wrong
  - Often fixed with **stalls**: insn stays in same stage for multiple cycles
- Let: **H** be average number of hazard stall cycles per instruction
  - $L_{1+P+H} = L_{1+P}$ (no hazards for one instruction)
  - $T_{+P+H} = [N/(N+H)]*N/L_1 = [N/(N+H)] * T_{+P}$
  - $L_{M+P+H} = M* L_1/N * [(N+H)/N] = [(N+H)/N] * L_{M+P}$
  - $S_{+P+H} = M*L_1 / M*L_1/N*[(N+H)/N] = [N/(N+H)]*S_{+P}$
- **H** also limit throughput, speedup $\rightarrow$ useful N
  - N$\uparrow \rightarrow$ H$\uparrow$ (more insns "in flight" $\rightarrow$ dependences become hazards)
  - Exact H depends on program, requires detailed simulation

Clock Rate vs. IPC

- Deeper pipeline (bigger N)
  - \+ frequency↑
  - – IPC↓
  - Ultimate metric is *IPC * frequency*
    - But people buy *frequency*, not *IPC * frequency*
- Trend has been for deeper pipelines
  - Intel example:
    - 486: 5 stages (50+ gate delays / clock)
    - Pentium: 7 stages
    - Pentium II/III: 12 stages
    - Pentium 4: 22 stages (10 gate delays / clock)
    - 800 MHz Pentium III was faster than 1 GHz Pentium4
  - No more Ghz, fewer stages: Core 1/2 = 14 stages, Core i7 16 stages

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 instruction per stage
    - Simplistic: **real** **X function** much, much more **complex**
- Compute optimal N (pipeline stages) given G,O,X
  - IPC = $1/(CPI_{ideal} + CPI_{stall}) = 1/(1 + X * N)$
  - $f = 1/t_n = 1 / (G / N + O)$
  - 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

Managing a Pipeline

- Proper flow requires two pipeline operations
  - Mess with latch write-enable and clear signals to achieve
- Operation I: **stall**
  - Effect: stops some insns in their current stages
  - Use: make younger insns wait for older ones to complete
  - Implementation: de-assert write-enable
- Operation II: **flush**
  - Effect: removes insns from current stages
  - Use: see later
  - Implementation: assert clear signals
- Both stall and flush must be **propagated** to younger insns

Structural Hazards

	1	2	3	4	5	6	7	8	9
`ld r2,0(r1)`	F	D	X	**M**	W				
`add r1,r3,r4`		F	D	X	M	W			
`st r6,0(r1)`			F	D	X	M	W		
`sub r1,r3,r5`				**F**	D	X	M	W	

- **Structural hazard**: resource needed twice in one cycle
  - Example: shared I/D$

Fixing Structural Hazards

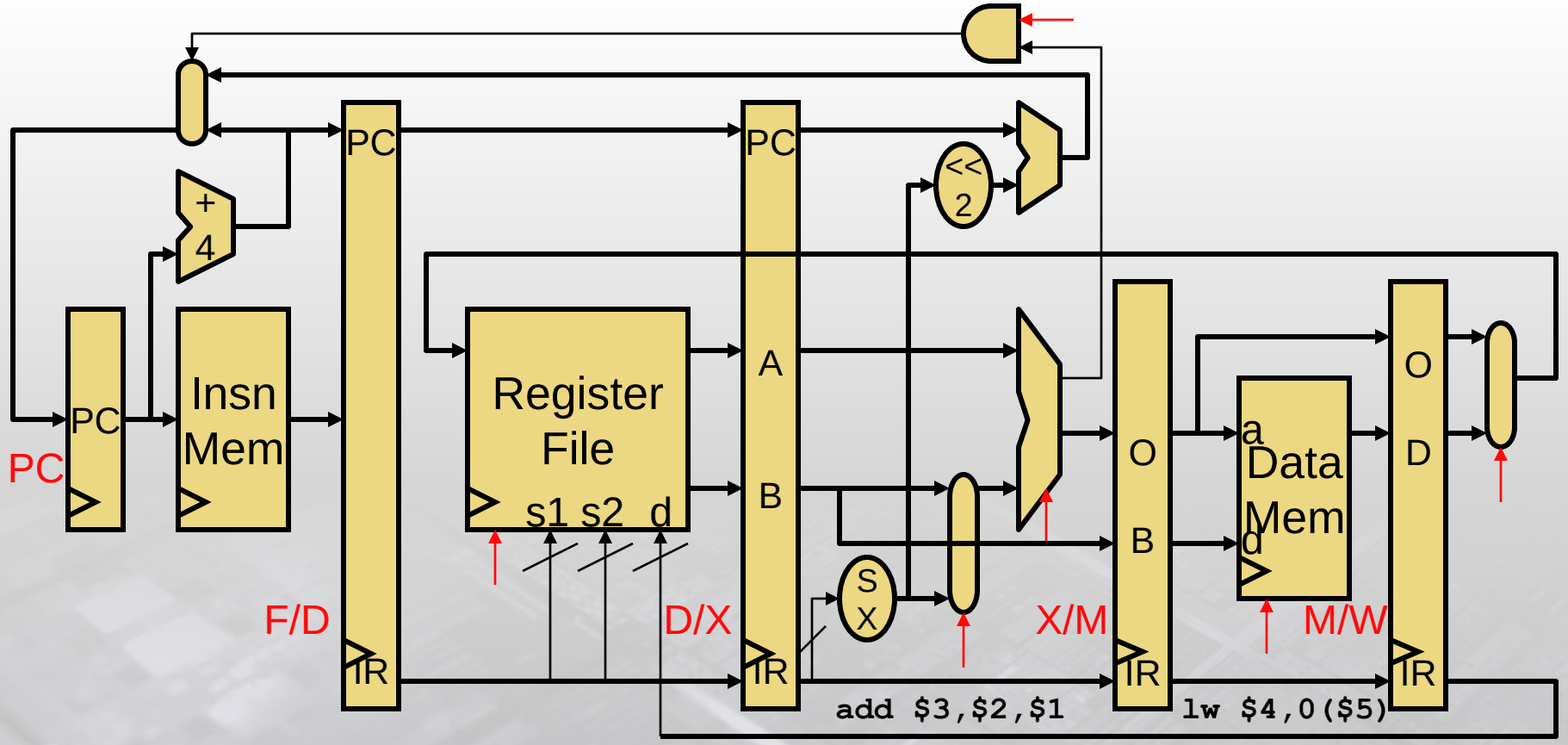
	1	2	3	4	5	6	7	8	9
`ld r2,0(r1)`	F	D	X	M	W				
`add r1,r3,r4`		F	D	X	M	W			
`st r6,0(r1)`			F	D	X	M	W		
`sub r1,r3,r5`				**s***	F	D	X	M	W

- Can fix structural hazards by stalling
  - **s*** = structural stall
  - Q: which one to stall: **`ld`** or **`sub`** ?
    - Always safe to stall younger instruction (here **`sub`**)
      - Fetch stall logic: (D/X.op == **`ld`** || D/X.op == **`st`**)
    - But not always the best thing to do performance wise (?)
  - \+ Low cost, simple
  - – Decreases IPC
  - Upshot: better to avoid by design, then to fix

Avoiding Structural Hazards

- Replicate the contended resource
  - \+ No IPC degradation
  - – Increased area, power, latency (interconnect delay?)
  - For cheap, divisible, or highly contended resources (e.g, I$/D$)
- Pipeline the contended resource
  - \+ No IPC degradation, low area, power overheads
  - – Sometimes tricky to implement (e.g., for RAMs)
  - For multi-cycle resources (e.g., multiplier)
- Design ISA/pipeline to reduce structural hazards (RISC)
  - Each insn uses a resource always for one cycle
  - And at most once
  - Always in same pipe stage
    - Reason why integer operations forced to go through M stage

Why Integer Operations Take 5 Cycles?



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

Data Hazards

- Real insn sequences pass values via registers/memory
  - Three kinds of **data dependences** (where's the fourth?)

`add r2,r3➔r1`
`sub r1,r4➔r2`
`or r6,r3➔r1`
	`add r2,r3➔r1`
	`sub r5,r4➔r2`
	`or r6,r3➔r1`
		`add r2,r3➔r1`
		`sub r1,r4➔r2`
		`or r6,r3➔r1`

Read-after-write (RAW) True-dependence	Write-after-read (WAR) Anti-dependence	Write-after-write (WAW) Output-dependence

  - Only one dependence between any two insns (RAW has priority)

- **Data hazards**: function of data dependences and pipeline
  - Potential for executing dependent insns in wrong order
  - Require both insns to be in pipeline ("in flight") simultaneously

Dependences and Loops

- Data dependences in loops
  - **Intra-loop**: within same iteration
  - **Inter-loop**: across iterations
  - Example: DAXPY (**D**ouble precision **A** **X** **P**lus **Y**)

```
for (i=0;i<100;i++)
   Z[i]=A*X[i]+Y[i];

0: ldf    X(r1)→f2
1: mulf   f0,f2→f4
2: ldf    Y(r1)→f6
3: addf   f6,f4→f8
4: stf    f8→Z(r1)
5: addi   8,r1→r1
6: slti   r1,800→r2
7: beq    r2,Loop
```

- RAW intra: 0→1(**f2**), 1→3(**f4**), 2→3(**f6**), 3→4(**f8**), 5→6(**r1**), 6→7(**r2**)
- RAW inter: 5→0(**r1**), 5→2(**r1**), 5→4(**r1**), 5→5(**r1**)
- WAR intra: 0→5(**r1**), 2→5(**r1**), 4→5(**r1**)
- WAR inter: 1→0(**f2**), 3→1(**f4**), 3→2(**f6**), 4→3(**f8**), 6→5(**r1**), 7→6(**r2**)
- WAW intra: none
- WAW inter: 0→0(**f2**), 1→1(**f4**), 2→2(**f6**), 3→3(**f8**), 6→6(**r2**)

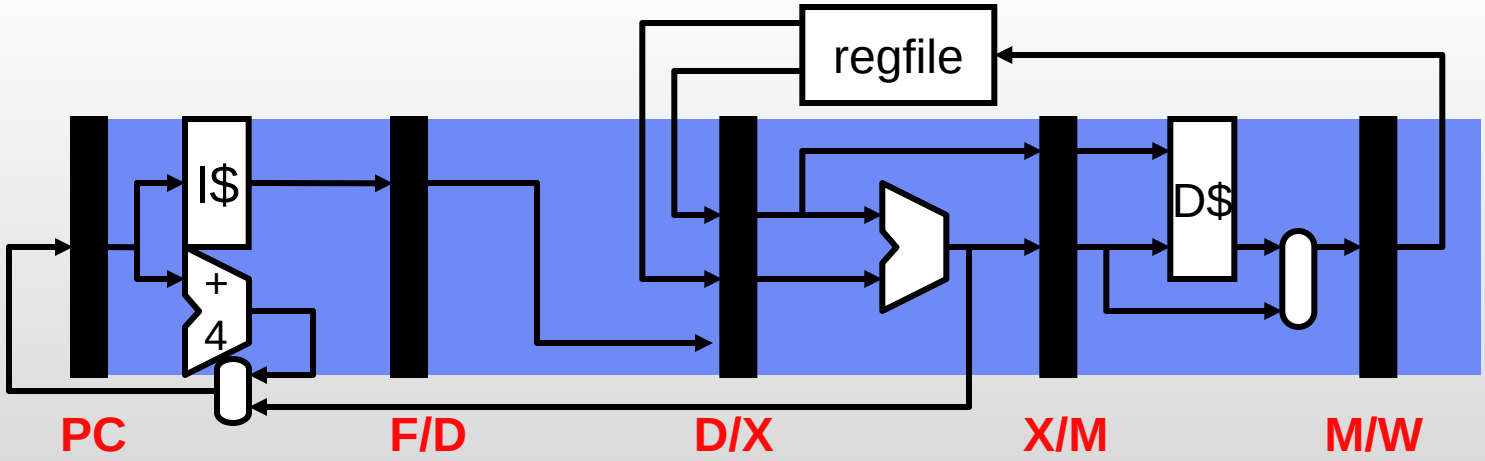
RAW

- **Read-after-write (RAW)**

```
add r2,r3➔r1
sub r1,r4➔r2
or r6,r3➔r1
```

  - Problem: swap would mean **sub** uses wrong value for **r1**
  - **True**: value flows through this dependence
    - Using different output register for **add** doesn't help

RAW: Detect and Stall



- **Stall logic**: detect and stall reader in D

  (F/D.IR.rs1 & (F/D.IR.rs1==D/X. IR.rd | F/D. IR.rs1==X/M. IR.rd | F/D. IR.rs1==M/W. IR.rd)) |

  (F/D. IR.rs2 & (F/D.IR.rs2==D/X. IR.rd | F/D. IR.rs2==X/M. IR.rd | F/D. IR.rs2==M/W. IR.rd))

  - Re-evaluated every cycle until no longer true
  + Low cost, simple
  – IPC degradation, dependences are the common case

Two Stall Timings (without bypassing)

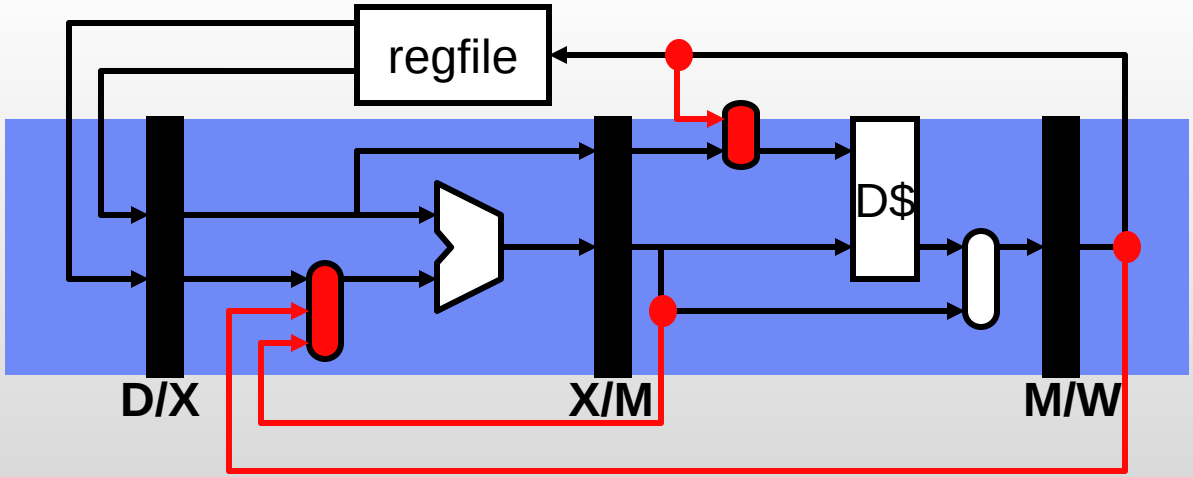
- Depend on how D and W stages share regfile
  - Each gets regfile for half a cycle
  - 1st half D reads, 2nd half W writes 3 cycle stall
  - **d*** = data stall, **p*** = propagated stall

	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***	**d***	**d***	D	X	M	W	
`add r5,r6➔r7`			**p***	**p***	**p***	F	D	X	M	W

  - \+ 1st half W writes, 2nd half D reads 2 cycle stall
  - How does the stall logic change here?

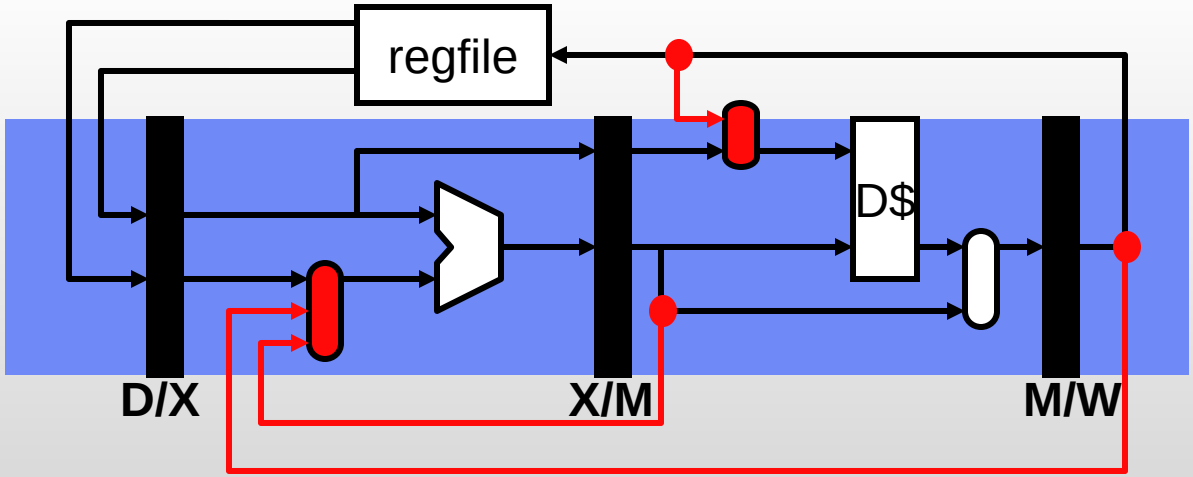
	1	2	3	4	5	6	7	8	9	10
`add r2,r3➔r1`	F	D	X	M	W					
`sub r1,r4➔r2`		F	**d***	**d***	D	X	M	W		
`add r5,r6➔r7`			**p***	**p***	F	D	X	M	W	

Reducing RAW Stalls with Bypassing



- Why wait until W stage? Data available after X or M stage
  - **Bypass** (aka **forward**) data directly to input of X or M
    - **MX**: from beginning of M (X output) to input of X
    - **WX**: from beginning of W (M output) to input of X
    - **WM**: from beginning of W (M output) to data input of M
    - Two each of MX, WX (figure shows 1) + WM = **full bypassing**
  - \+ Reduces stalls in a big way
  - – Additional wires and muxes may increase clock cycle

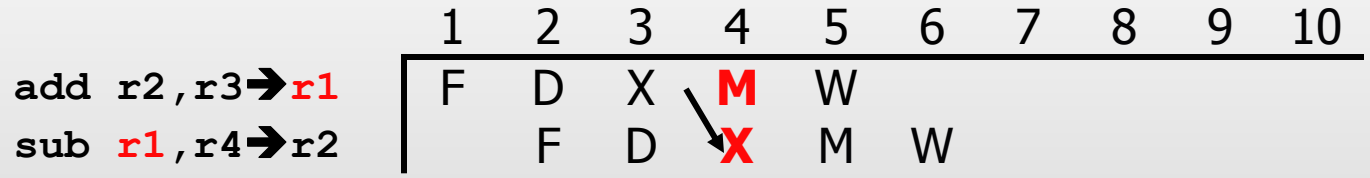
Bypass Logic



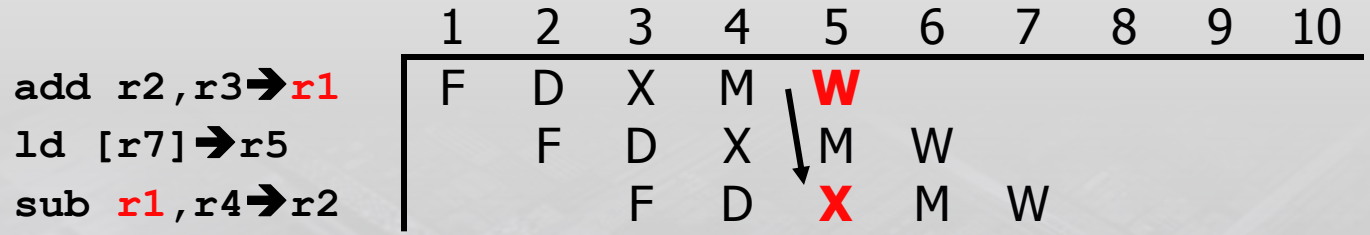
- Bypass logic: similar to but separate from stall logic
  - Stall logic controls latches, bypass logic controls mux inputs
  - Complement one another: can't bypass → must stall
  - ALU input mux bypass logic
    - (D/X.IR.rs2 & X/M.rd==D/X. IR.rs2) → 2 // check first
    - (D/X. IR.rs2 & M/W.rd==D/X. IR.rs2) → 1 // check second
    - (D/X. IR.rs2) → 0 // check last

Pipeline Diagrams with Bypassing

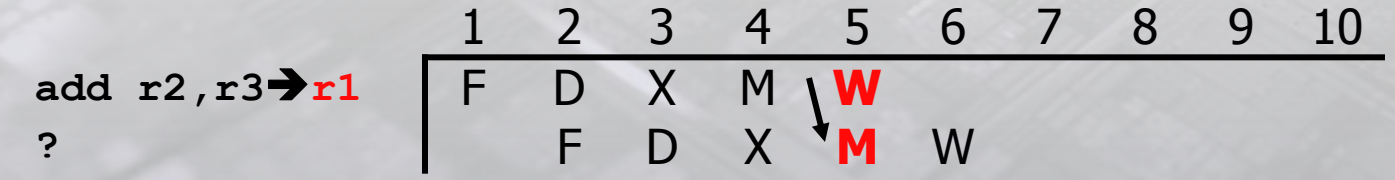
- If bypass exists, "from"/"to" stages execute in same cycle
  - Example: full bypassing, use MX bypass



  - Example: full bypassing, use WX bypass

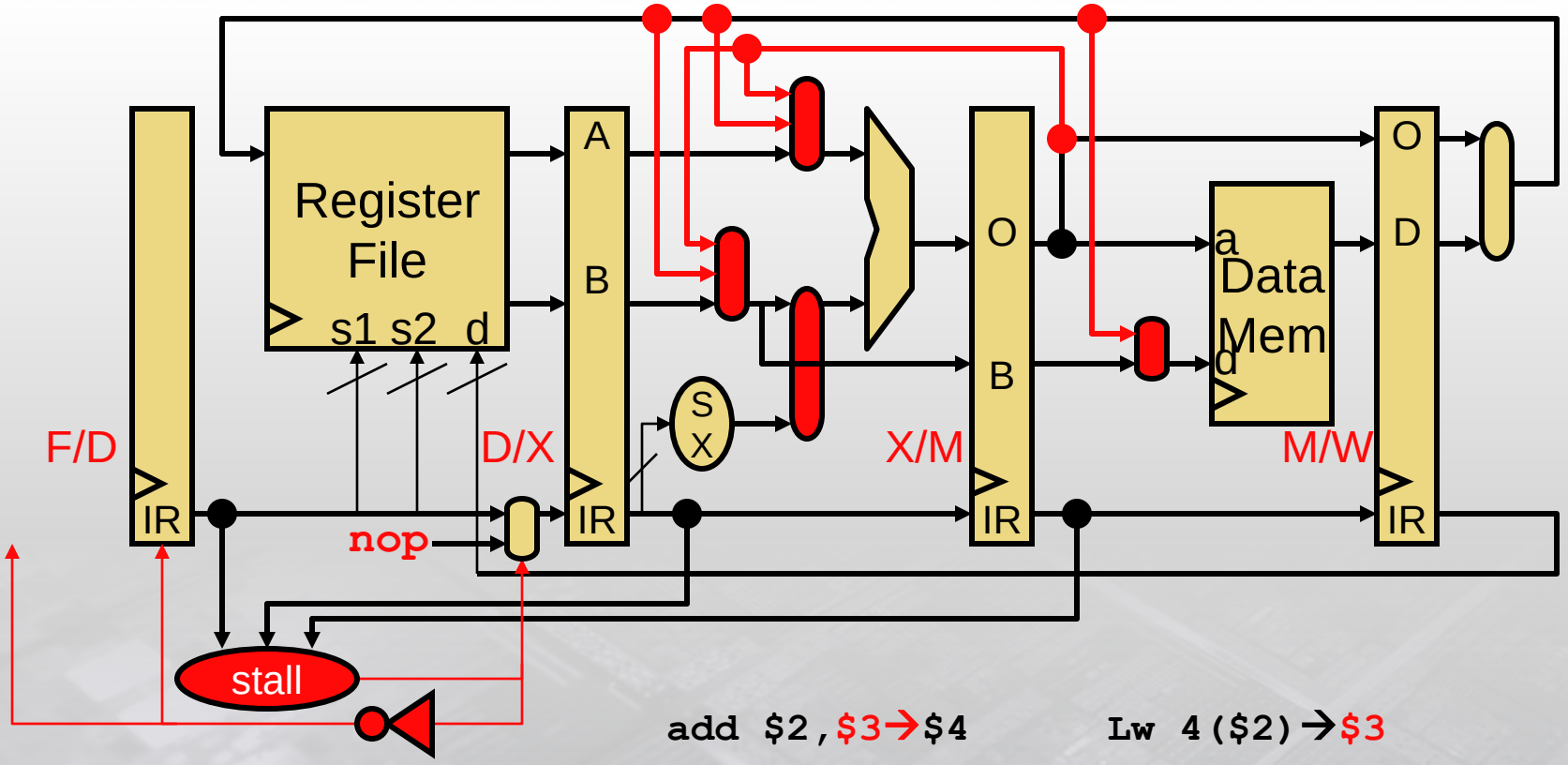


  - Example: WM bypass



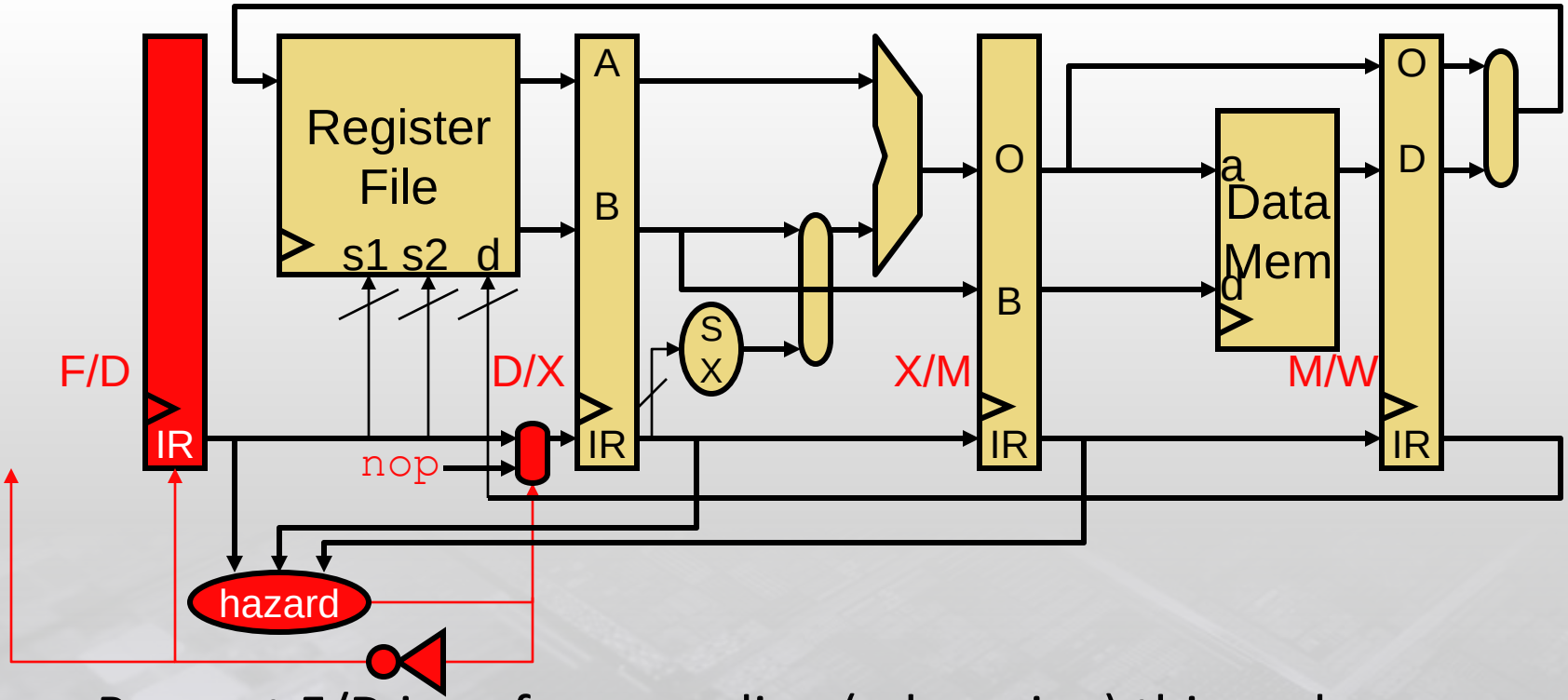
    - Can you think of a code example that uses the WM bypass?

Have We Prevented All Data Hazards?



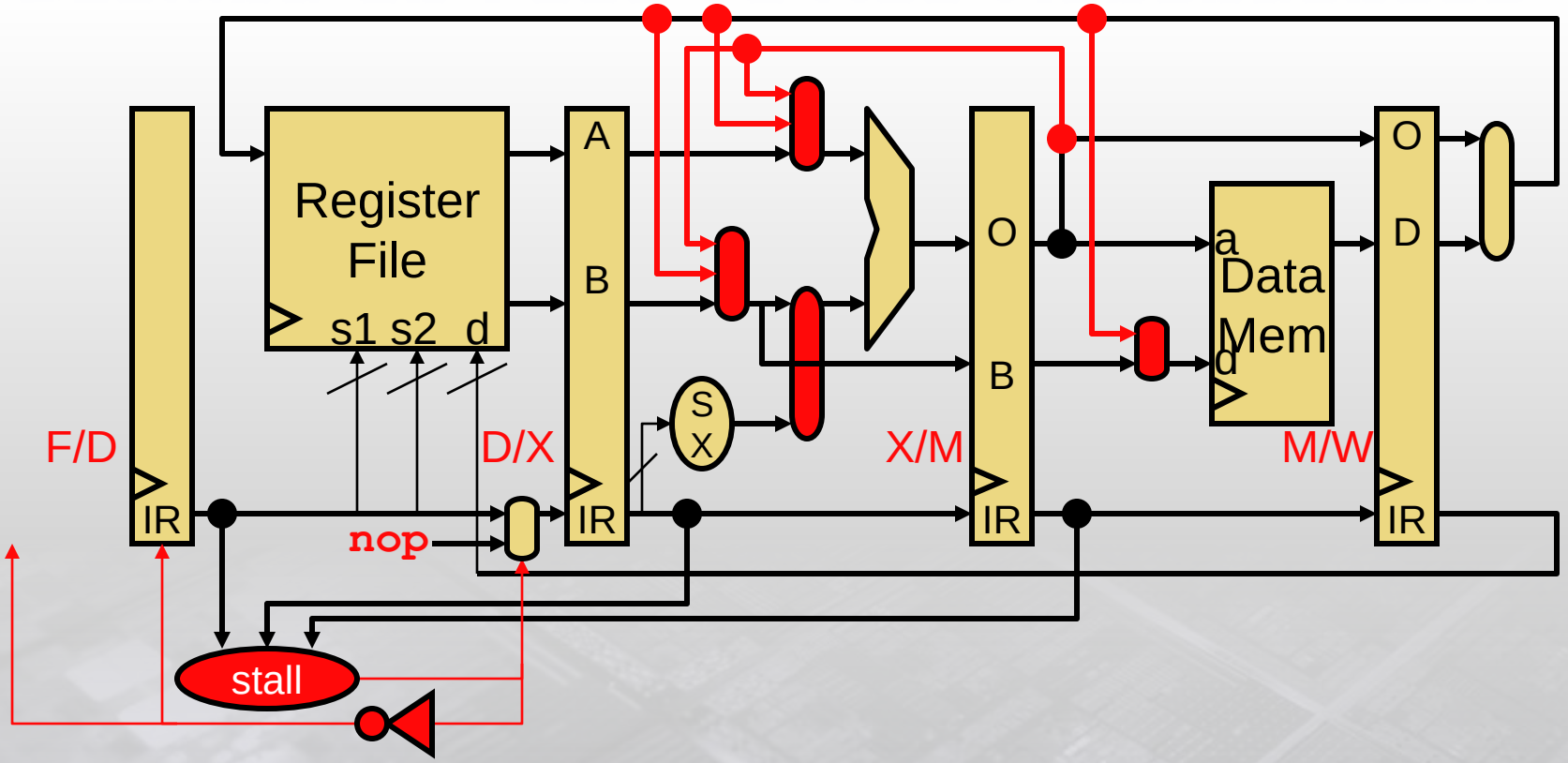
- No. Consider a "load" followed by a dependent "add" insn
- Bypassing alone isn't sufficient
- Solution? Detect this, and then stall the "add" by one cycle

Stalling to Avoid 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

Stalling on Load-To-Use Dependences

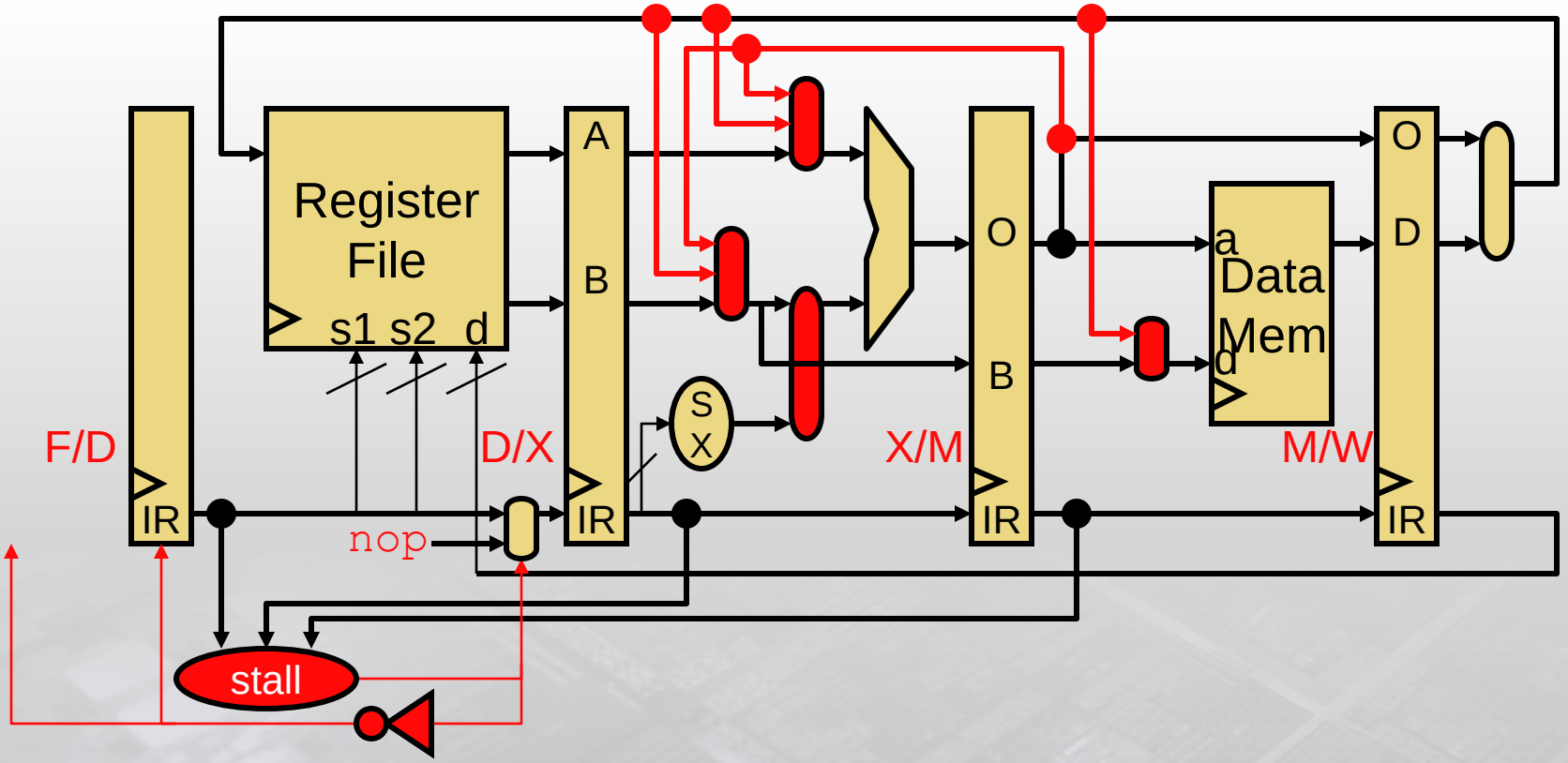


Stall = (D/X.IR.op == ld) &&

((F/D.IR.rs1 == D/X.IR.rd) ||

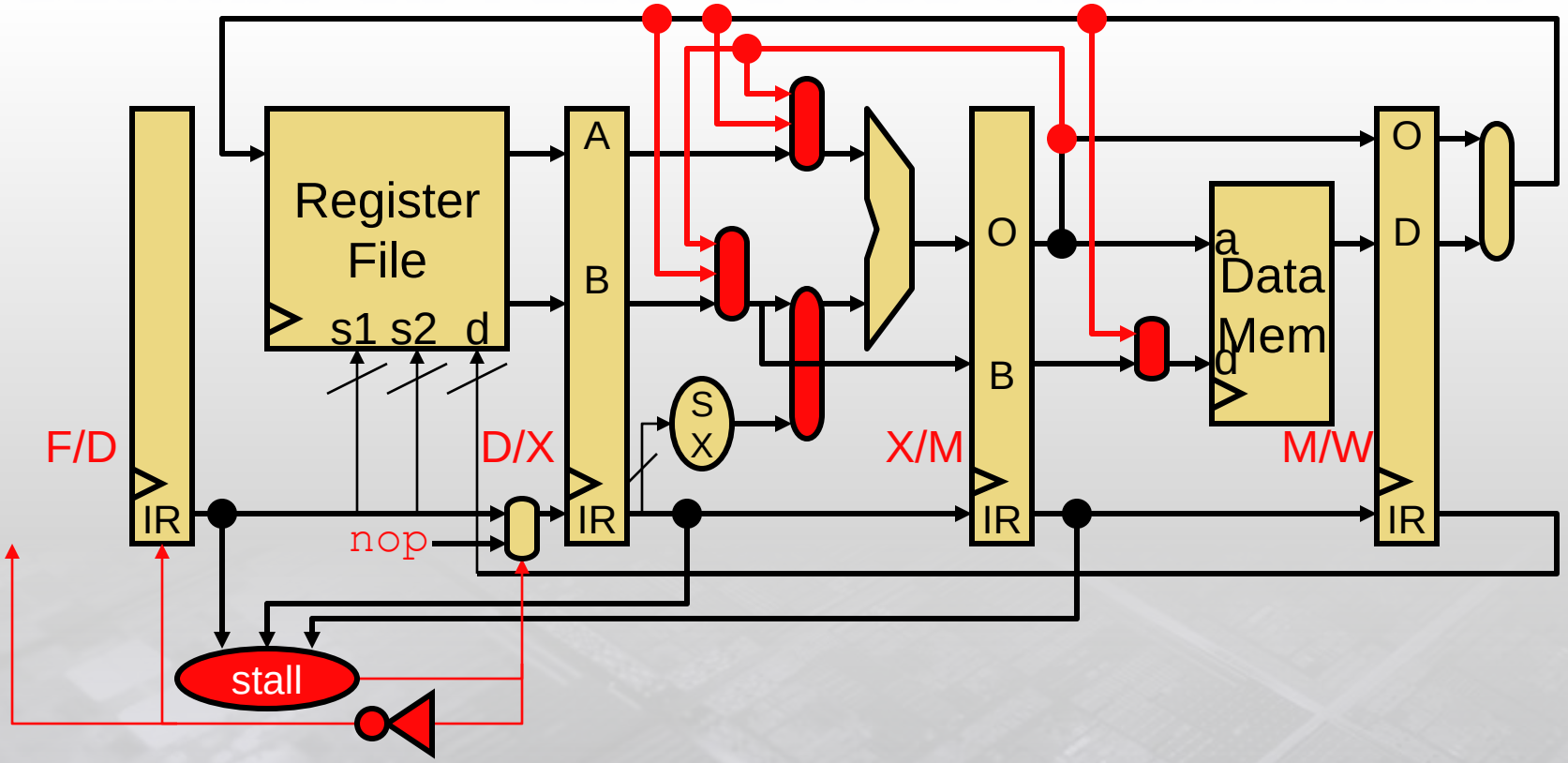
((F/D.IR.rs2 == D/X.IR.rd) && (F/D.IR.OP != st))

Stalling on Load-To-Use Dependences



Stall = (D/X.IR.op == ld) &&
((F/D.IR.rs1 == D/X.IR.rd) ||
((F/D.IR.rs2 == D/X.IR.rd) && (F/D.IR.OP != st))

Stalling on Load-To-Use Dependences

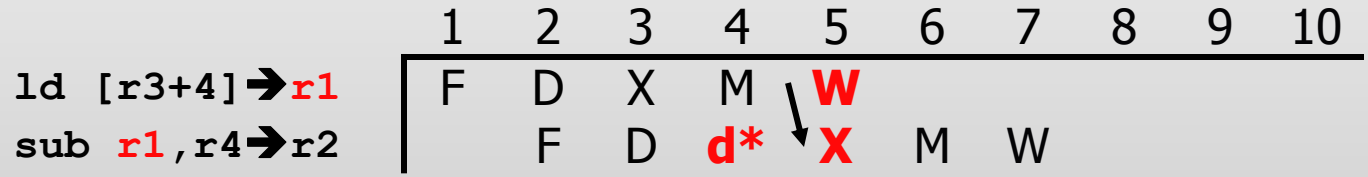



```
add $2,$3→$4      (stall bubble)      lw $3,…
```

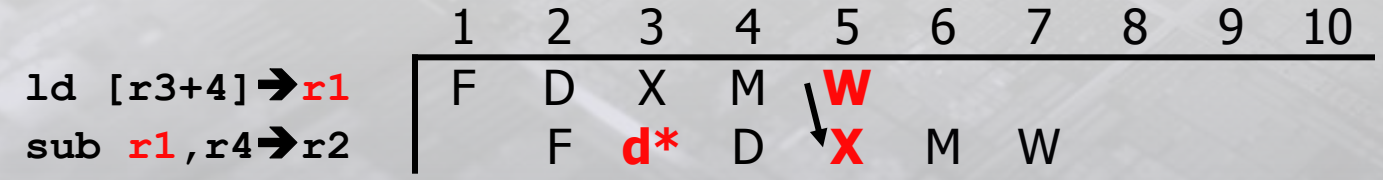
Stall = (D/X.IR.op == ld) &&
((F/D.IR.rs1 == D/X.IR.rd) ||
((F/D.IR.rs2 == D/X.IR.rd) && (F/D.IR.OP != st))

Load-Use Stalls

- Even with full bypassing, stall logic is unavoidable
  - **Load-use stall**
    - Load value not ready at beginning of M → can't use MX bypass
    - Use WX bypass



- Aside: with WX bypassing, stall logic can be in D or X



- Aside II: how does stall/bypass logic handle cache misses?

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
  - CPI = 1 + (1 * 20% * 50%) = **1.1**

Compiler Scheduling

- Compiler can schedule (move) insns to reduce stalls
  - **Basic pipeline scheduling**: eliminate back-to-back load-use pairs
  - Example code sequence: `a = b + c; d = f - e;`
  - MIPS Notation: (**NO MORE "→"**)
    - "ld r2,4(sp)" is "ld [sp+4]→r2" "st r1, 0(sp)" is "st r1→[sp+0]"

Before

```
ld r2,4(sp)
ld r3,8(sp)
add r1,r3,r2   //stall
st r1,0(sp)
ld r5,16(sp)
ld r6,20(sp)
sub r4,r5,r6 //stall
st r4,12(sp)
```

After

```
ld r2,4(sp)
ld r3,8(sp)
ld r5,16(sp)
add r1,r3,r2   //no stall
ld r6,20(sp)
st r1,0(sp)
sub r5,r5,r6   //no stall
st r4,12(sp)
```

Compiler Scheduling Requires

- **Large scheduling scope**
  - Independent instruction to put between load-use pairs
  + Original example: large scope, two independent computations
  – This example: small scope, one computation

Before

```
ld r2,4(sp)
ld r3,8(sp)
add r1,r3,r2 //stall
st r1,0(sp)
```

After

```
ld r2,4(sp)
ld r3,8(sp)
add r1,r3,r2 //stall
st r1,0(sp)
```

Compiler Scheduling Requires

- **Enough registers**
  - To hold additional "live" values
  - Example code contains 7 different values (including **sp**)
  - Before: max 3 values live at any time → 3 registers enough
  - After: max 4 values live → 3 registers not enough → WAR violations

Original

```
ld r2,4(sp)
ld r1,8(sp)
add r1,r1,r2 //stall
st r1,0(sp)
ld r2,16(sp)
ld r1,20(sp)
sub r1,r2,r1 //stall
st r1,12(sp)
```

Wrong!

```
ld r2,4(sp)
ld r1,8(sp)
ld r2,16(sp)
add r1,r1,r2 //WAR
ld r1,20(sp)
st r1,0(sp)   //WAR
sub r1,r2,r1
st r1,12(sp)
```

Compiler Scheduling Requires

- **Alias analysis**
  - Ability to tell whether load/store reference same memory locations
    - Effectively, whether load/store can be rearranged
  - Example code: easy, all loads/stores use same base register (**sp**)
  - New example: can compiler tell that **r8** = **sp**?

Before

```
ld r2,4(sp)
ld r3,8(sp)
add r1,r3,r2 //stall
st r1,0(sp)
ld r5,0(r8)
ld r6,4(r8)
sub r4,r5,r6  //stall
st r4,8(r8)
```

Wrong(?)

```
ld r2,4(sp)
ld r3,8(sp)
ld r5,0(r8)
add r1,r3,r2
ld r6,4(r8)
st r1,0(sp)
sub r4,r5,r6
st r4,8(r8)
```

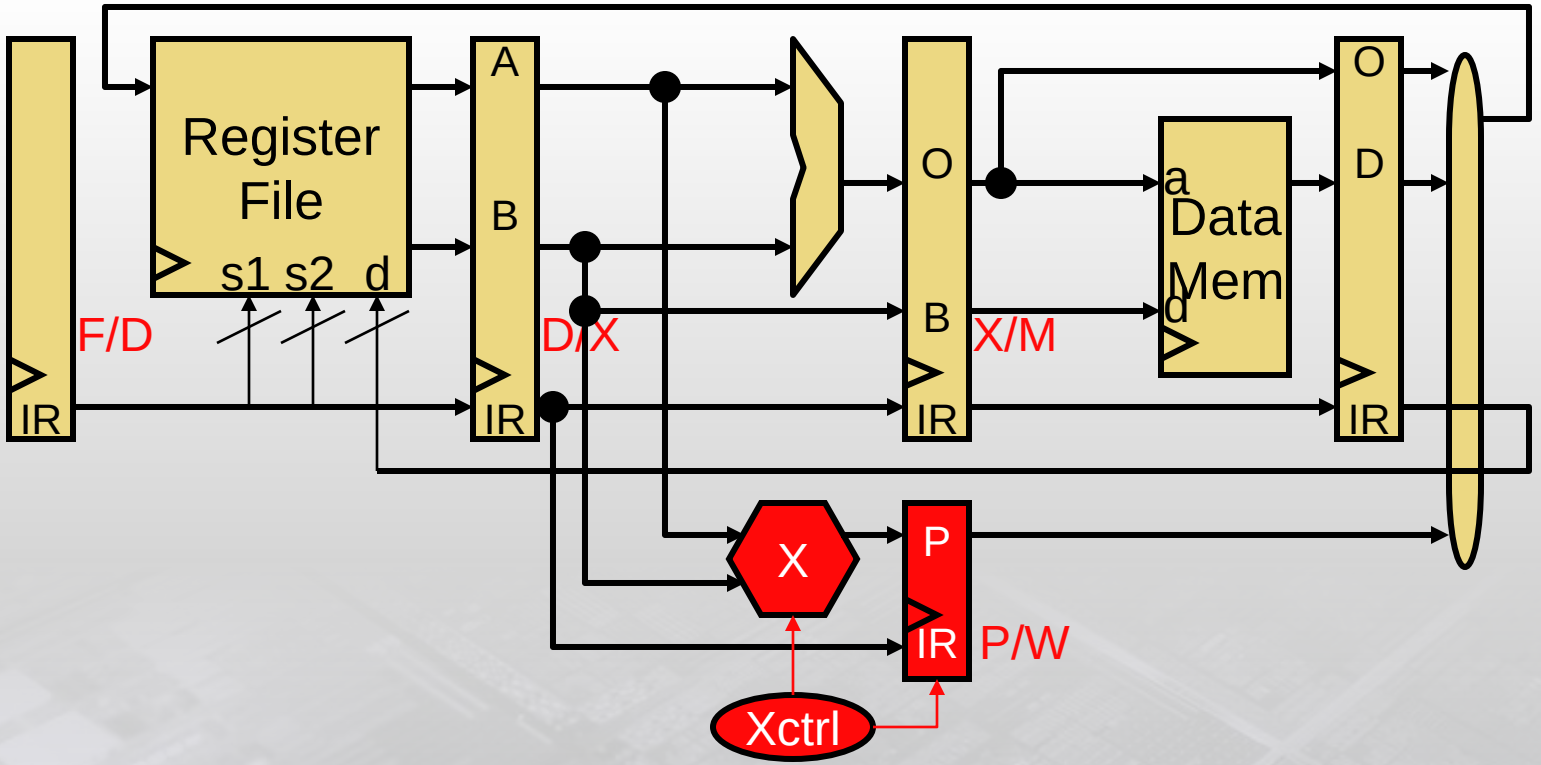
WAW Hazards

- **Write-after-write (WAW)**

```
add r1,r2,r3
sub r2,r1,r4
or  r1,r6,r3
```

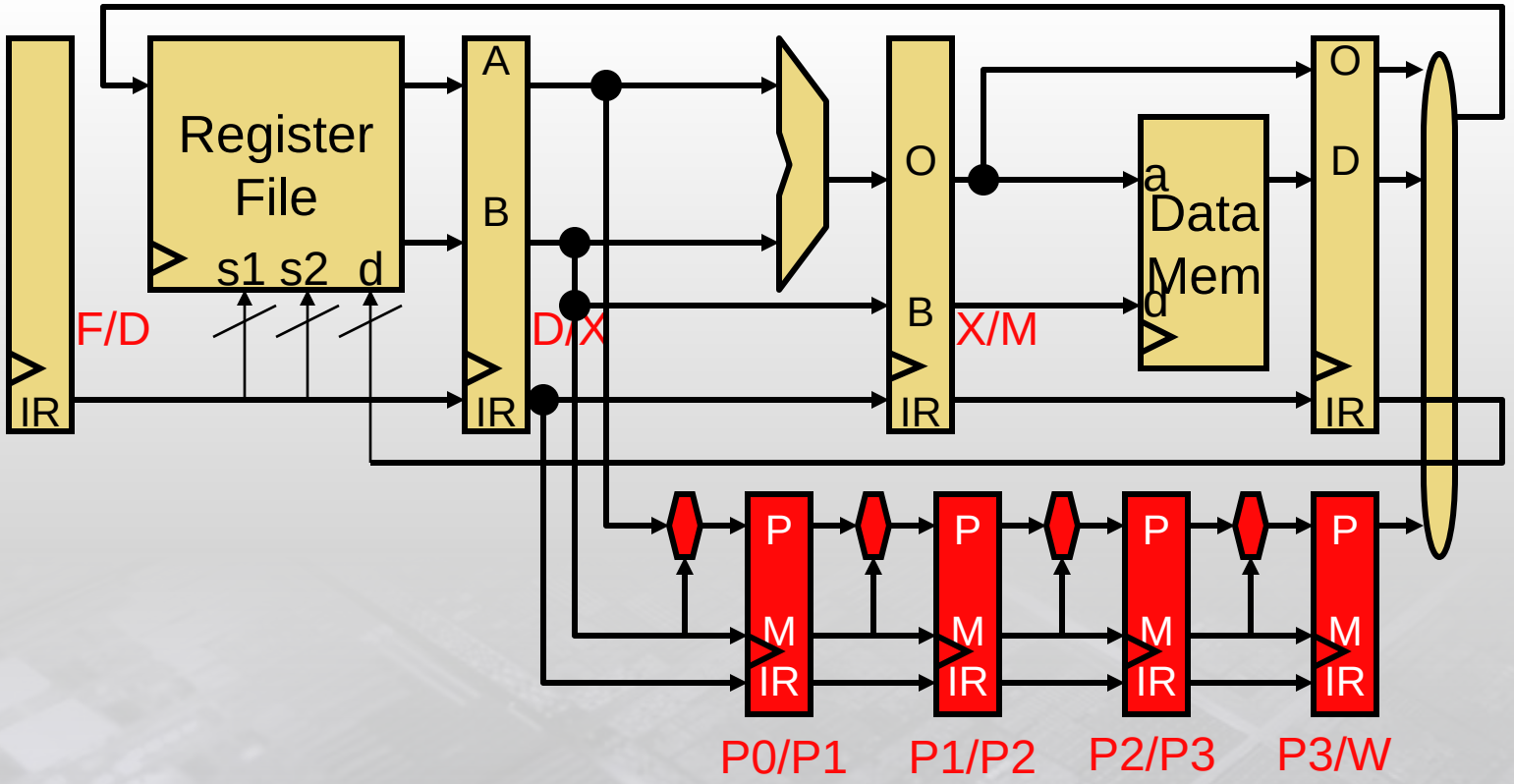
- Compiler effects
  - Scheduling problem: reordering would leave wrong value in **r1**
    - Later instruction reading **r1** would get wrong value
  - **Artificial**: no value flows through dependence
    - Eliminate using different output register name for **or**
- Pipeline effects
  - Doesn't affect in-order pipeline with **single-cycle operations**
    - Another reason for making ALU operations go through M stage
  - Can happen with multi-cycle operations (e.g., FP, some int. insn or cache misses)

Pipelining and Multi-Cycle Operations



- What if you wanted to add a multi-cycle operation?
  - E.g., 4-cycle multiply
  - **P/W**: separate output latch connects to W stage
  - Controlled by pipeline control and multiplier 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

Aside: 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 improve throughput than multiple non-pipelined units

	1	2	3	4	5	6	7	8	9	10	11
`mulf f0,f1,f2`	F	D	E*	E*	E*	E*	W				
`mulf f3,f4,f5`		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
`divf f0,f1,f2`	F	D	E/	E/	E/	E/	W				
`divf f3,f4,f5`		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 Diagram with Int. Multiplier

	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…
  - Two instructions could try to write regfile 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**		

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
`mul $4,$3,$5`	F	D	P0	P1	P2	P3	W		
`addi $4,$1,1`		F	D	X	M	**W**			
…									
…									
`add $10,$4,$6`					F	D	X	M	W

- **Multi-cycle operations introduces WAW hazard on simple pipelines**
  - Not necessarily supplementary FP pipeline

Handling WAW Hazards

	1	2	3	4	5	6	7	8	9	10
`div f0,f1➔f2`	F	D	E/	E/	E/	E/	E/	**W**		
`stf f2➔[r1]`		F	D	d*	d*	d*	X	M	W	
`addf f0,f1➔f2`			F	D	E+	E+	**W**			

- What to do?
  - Option I: stall younger instruction (**`addf`**) at writeback
    - \+ Intuitive, simple
    - – Lower performance, cascading W structural hazards
  - Option II: cancel older instruction (**`divf`**) writeback
    - \+ No performance loss
    - – What if **`divf`** or **`stf`** cause an exception (e.g., /0, page fault)?

Handling Interrupts/Exceptions

- How are interrupts/exceptions handled in a pipeline?
  - **Interrupt**: external, e.g., timer, I/O device requests
  - **Exception**: internal, e.g., /0, page fault, illegal instruction
  - We care about **restartable** interrupts (e.g. **`stf`** page fault)

	1	2	3	4	5	6	7	8	9	10
`divf f0,f1➔f2`	F	D	E/	E/	E/	E/	E/	W		
`stf f2➔[r1]`		F	D	d*	d*	d*	X	M	W	
`addf f0,f1➔f2`			F	D	E+	E+	W			

- VonNeumann says
  - "Insn execution should appear sequential and atomic"
    - Insn X should complete before instruction X+1 should begin
    + Doesn't physically have to be this way (e.g., pipeline)
    - But be ready to restore to this state at a moments notice
  - Called **precise state** or **precise interrupts**

Handling Interrupts

	1	2	3	4	5	6	7	8	9	10
`divf f0,f1➔f2`	F	D	E/	E/	E/	E/	E/	W		
`stf f2➔[r1]`		F	D	d*	d*	d*	X	**M**	W	
`addf f0,f1➔f2`			F	D	E+	E+	W			

- In this situation
  - Make it appear as if **`divf`** finished and **`stf`**, **`addf`** haven't started
  - Allow **`divf`** to writeback
  - **Flush** **`stf`** and **`addf`** (so that's what a flush is for)
    - But **`addf`** has already written back
    - Keep an "undo" register file? Complicated
    - Force in-order writebacks? Slow
  - Invoke exception handler
  - Restart **`stf`**

More Interrupt Nastiness

	1	2	3	4	5	6	7	8	9	10
`divf f0,f1➔f2`	F	D	E/	E/	E/	E/	E/	W		
`stf f2➔[r1]`		F	D	d*	d*	d*	X	**M**	W	
`divf f0,f4➔f2`			F	D	**E/**	E/	E/	E/	E/	W

- What about two simultaneous in-flight interrupts
  - Example: **`stf`** page fault, **`divf`** /0
  - Interrupts must be handled in program order (**`stf`** first)
    - Handler for **`stf`** must see program as if **`divf`** hasn't started
  - Must defer interrupts until writeback **and** force in-order writeback
  - Kind of a bogus example, /0 is non-restartable
- In general: interrupts are really nasty
  - Some processors (Alpha) only implement precise integer interrupts
  - Easier because fewer WAW scenarios
  - Most floating-point interrupts are non-restartable anyway

WAR Hazards

- **Write-after-read (WAR)**

```
add r1,r2,r3
sub r2,r5,r4
or r6,r3,r1
```

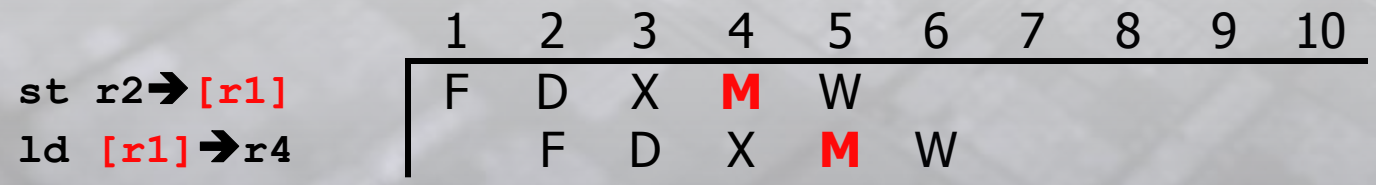
- Compiler effects
  - Scheduling problem: reordering would mean **add** uses wrong value for **r2**
  - **Artificial**: solve using different output register name for **sub**
- Pipeline effects
  - Can't happen in simple in-order pipeline
  - Can happen with out-of-order execution

Memory Data Hazards

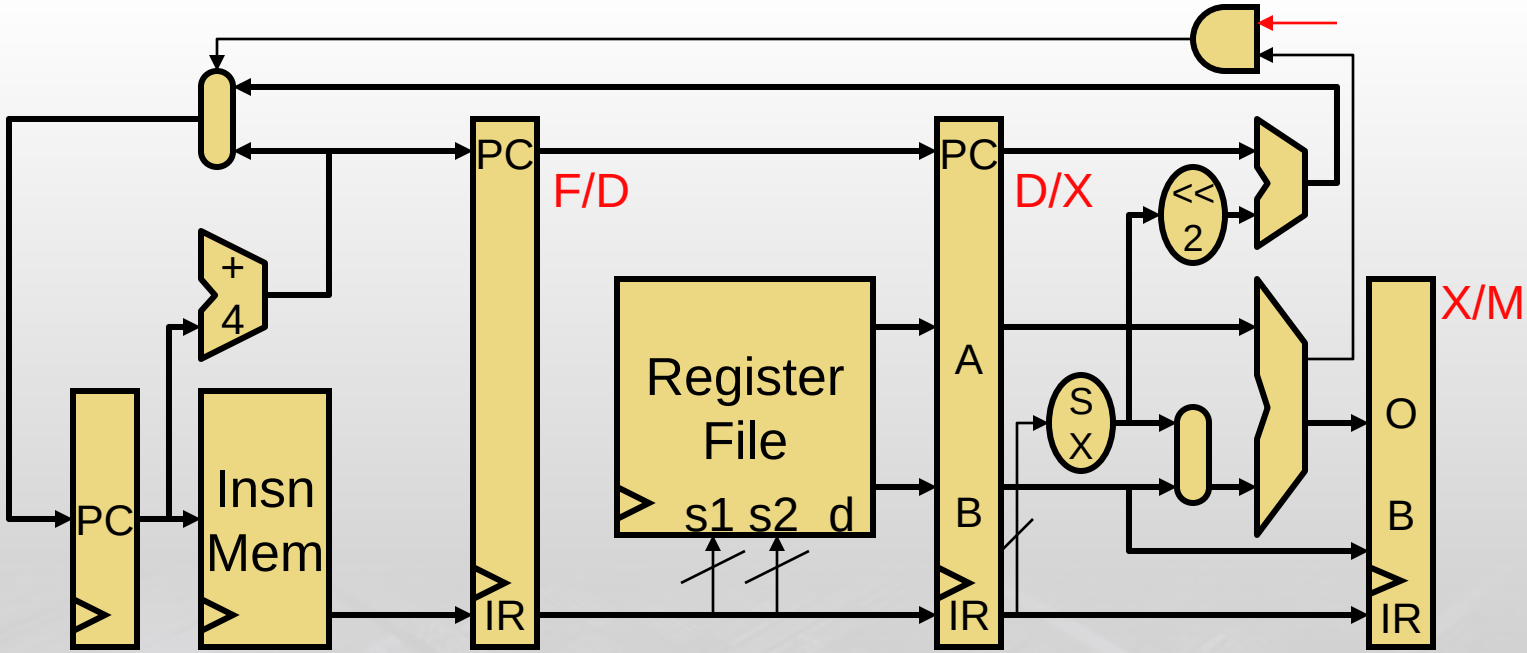
- So far, have seen/dealt with register dependences
  - Dependences also exist through memory

`st r2➔[r1]`
`ld [r1]➔r4`
`st r5➔[r1]`
Read-after-write (RAW)
	`st r2➔[r1]`
	`ld [r1]➔r4`
	`st r5➔[r1]`
	Write-after-read (WAR)
		`st r2➔[r1]`
		`ld [r1]➔r4`
		`st r5➔[r1]`
		Write-after-write (WAW)

- But in an in-order pipeline like ours, they do not become hazards
- Memory read and write happen at the same stage
  - Register read happens three stages earlier than register write
- In general: memory dependences more difficult than register

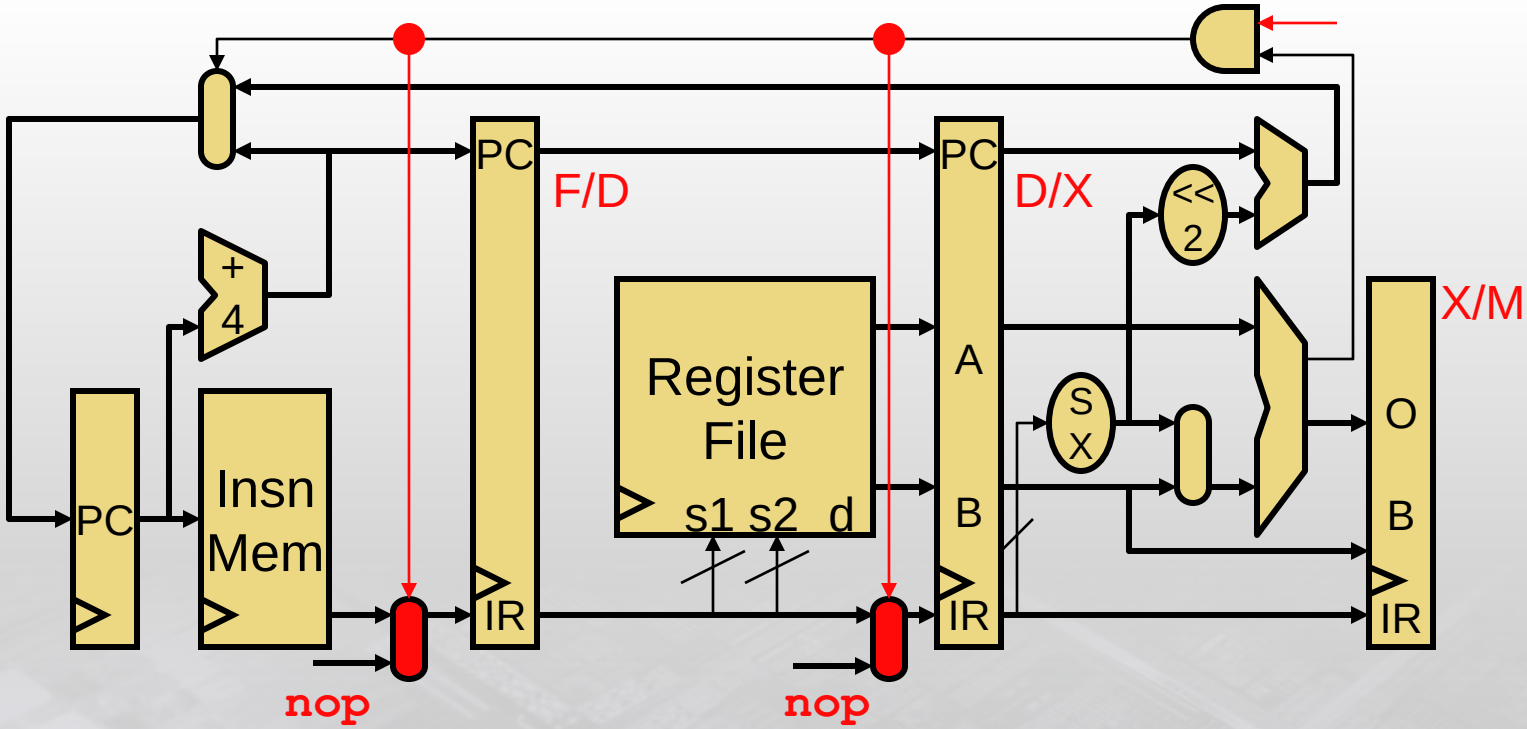


Control hazards



- What About **Branches**?
  - 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 F/D and D/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 Performance

- Back of the envelope calculation
  - **Branch: 20%**, load: 20%, store: 10%, other: 50%
  - Say, **75% of branches are taken**

- CPI = 1 + 20% * 75% * 2 =
  1 + **0.20 * 0.75 * 2** = 1.3
  - **Branches cause 30% slowdown**
    - Even worse with deeper pipelines
  - How do we reduce this penalty?

Big Idea: Speculation

- **Speculation**
  - "Engagement in risky transactions on the 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 instructions
  - – Must undo incorrect changes (recover pre-speculation state)

The "game": **[$\%_{correct}$ * gain] – [(1–$\%_{correct}$) * penalty]**

Control Hazards: Control Speculation

- Deal with control hazards with **control speculation**
  - Unknown parameter: are these the correct insns to execute next?
- Mechanics
  - Guess branch target, start fetching at guessed position
  - Execute branch to verify (check) guess
    - Correct speculation? keep going
    - Mis-speculation? Flush mis-speculated insns
  - Don't write registers or memory until prediction verified
- Speculation game for in-order 5 stage pipeline
  - Gain = 2 cycles
  - Penalty = 0 cycles
    - No penalty $\rightarrow$ mis-speculation no worse than stalling
  - $\%_{correct}$ = **branch prediction**
    - Static (compiler) OK, **dynamic** (hardware) much better

Control Speculation and Recovery

Correct:

	1	2	3	4	5	6	7	8	9
`addi r1,1➔r3`	F	D	X	M	W				
`bnez r3,targ`		F	D	X	M	W			
`st r6➔[r7+4]`			**F**	**D**	X	M	W		
`targ:add r4,r5➔r4`				**F**	D	X	M	W	

speculative

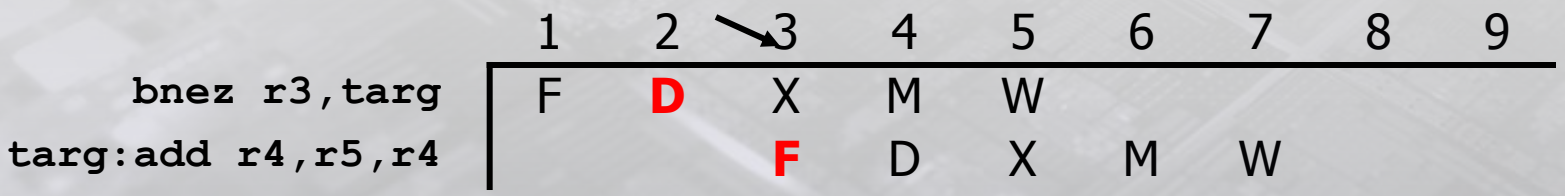
- **Mis-speculation recovery**: what to do on wrong guess
  - Not too painful in an in-order pipeline
  - Branch resolves in X
  - \+ Younger insns (in F, D) haven't changed permanent state
  - **Flush** insns currently in F/D and D/X (i.e., replace with **nops**)

Recovery:

	1	2	3	4	5	6	7	8	9
`addi r1,1➔r3`	F	D	X	M	W				
`bnez r3,targ`		F	D	**X**	M	W			
~~`st r6➔[r7+4]`~~			**F**	**D**	--	--	--		
~~`targ:add r4,r5➔r4`~~				**F**	--	--	--	--	
`targ:add r4,r5➔r4`					**F**	D	X	M	W

Reducing Penalty: Fast Branches

- **Fast branch**: targets control-hazard penalty
  - Basically, branch insns that can resolve at D, not X
    - Test must be comparison to zero or equality, **no time for ALU**
  - \+ New taken branch penalty is 1
  - – Additional comparison insns (e.g., **`slt`**) for complex tests
  - – Bypass logic should bypass into decode stage now, too (more mux, longer wires)
  - – Additional nastiness with exceptions



Fast Branch Performance

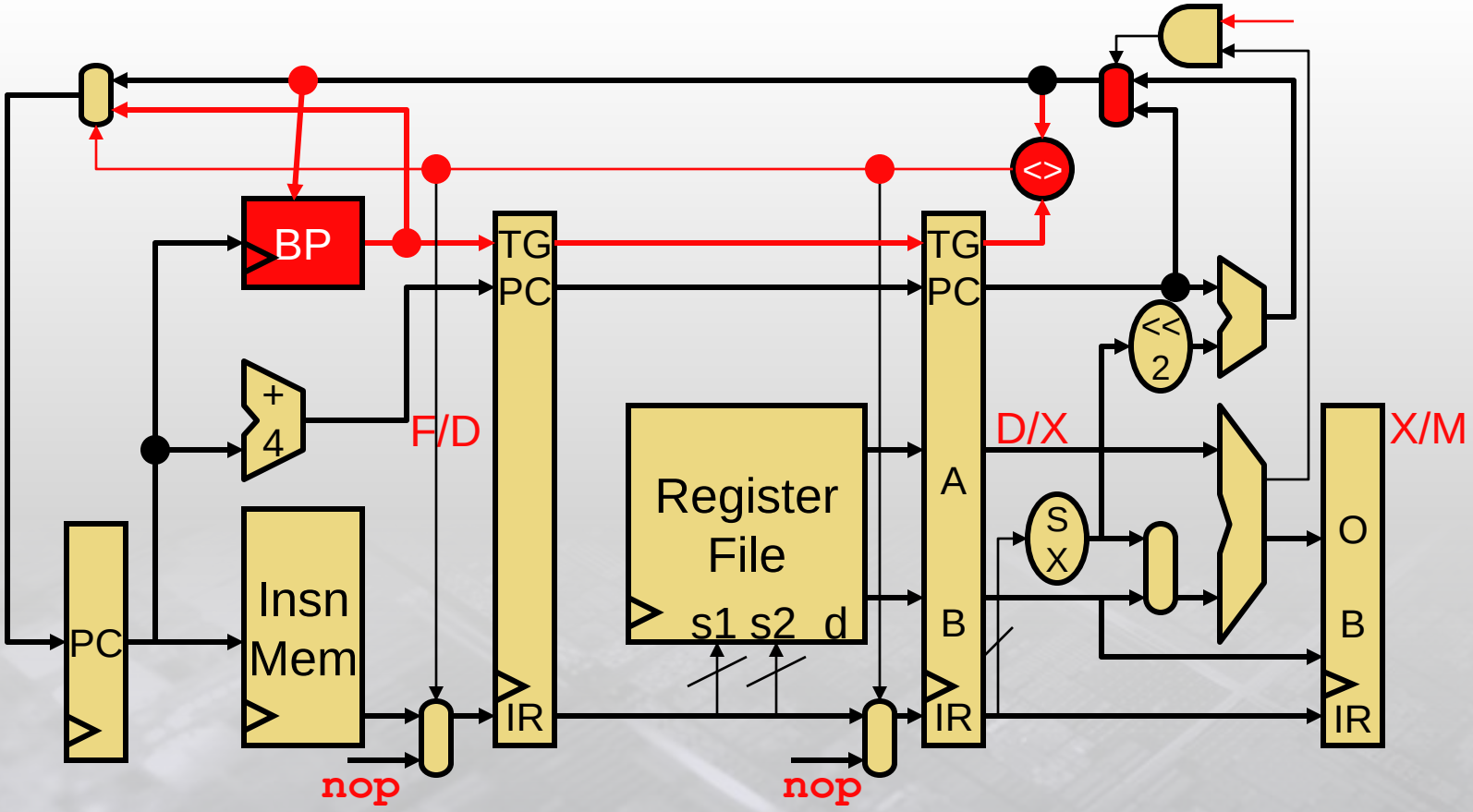
- Assume: Branch: 20%, 75% of branches are taken
  - CPI = 1 + 20% * 75% * **1** = 1 + **0.20*0.75*1** = **1.15**
    - **15% slowdown** (better than the 30% from before)

- But wait, fast branches assume only simple comparisons
  - Fine for MIPS
  - But not fine for ISAs with "branch if $1 > $2" operations

- In such cases, say 25% of branches require an extra insn
  - CPI = 1 + (20% * 75% * 1) + 20%*25%*1(extra insn) = **1.2**

- Example of ISA and micro-architecture interaction
  - Type of branch instructions
  - Another option: "Delayed branch" or "branch delay slot"
  - What about condition codes?

Fewer Mispredictions: Branch Prediction

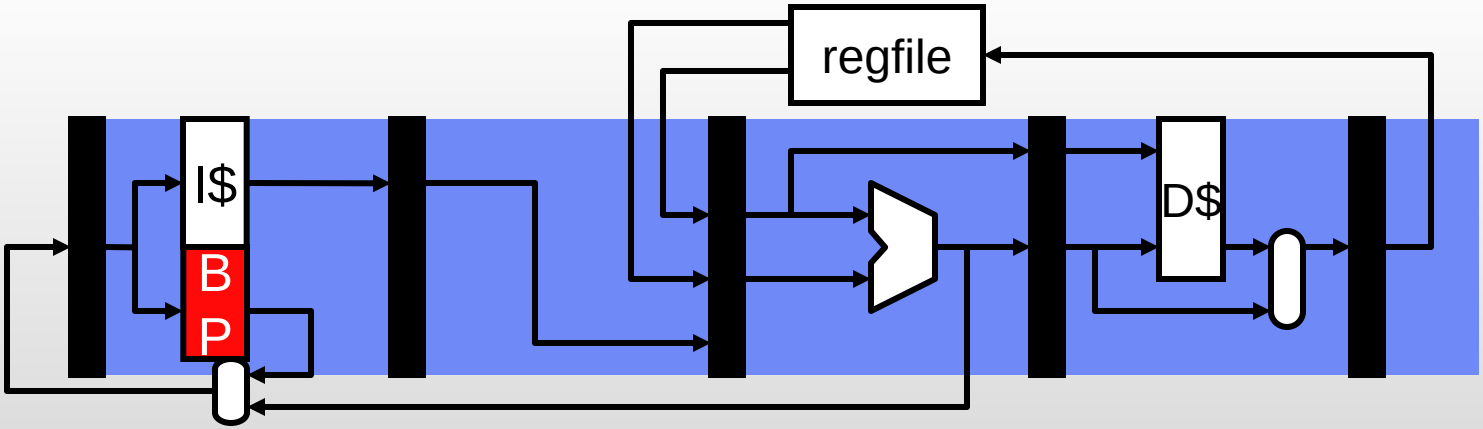


- **Dynamic branch prediction**:
  - Hardware guesses outcome
  - Start fetching from guessed address

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
  - CPI = 1 + 20% * 5% * 2 = **1.02**

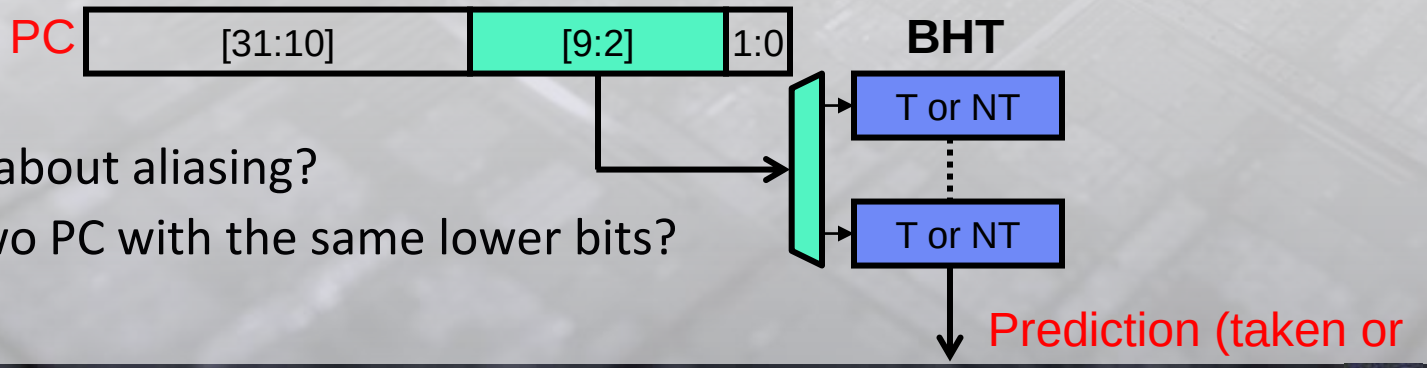
Dynamic Branch Prediction



- 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 unbiased 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?



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: consider **inner loop branch** below (* = mis-prediction)

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

State/prediction	N*	T	T	T*	N*	T	T	T*	N*	T	T	T*
Outcome	T	T	T	N	T	T	T	N	T	T	T	N

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

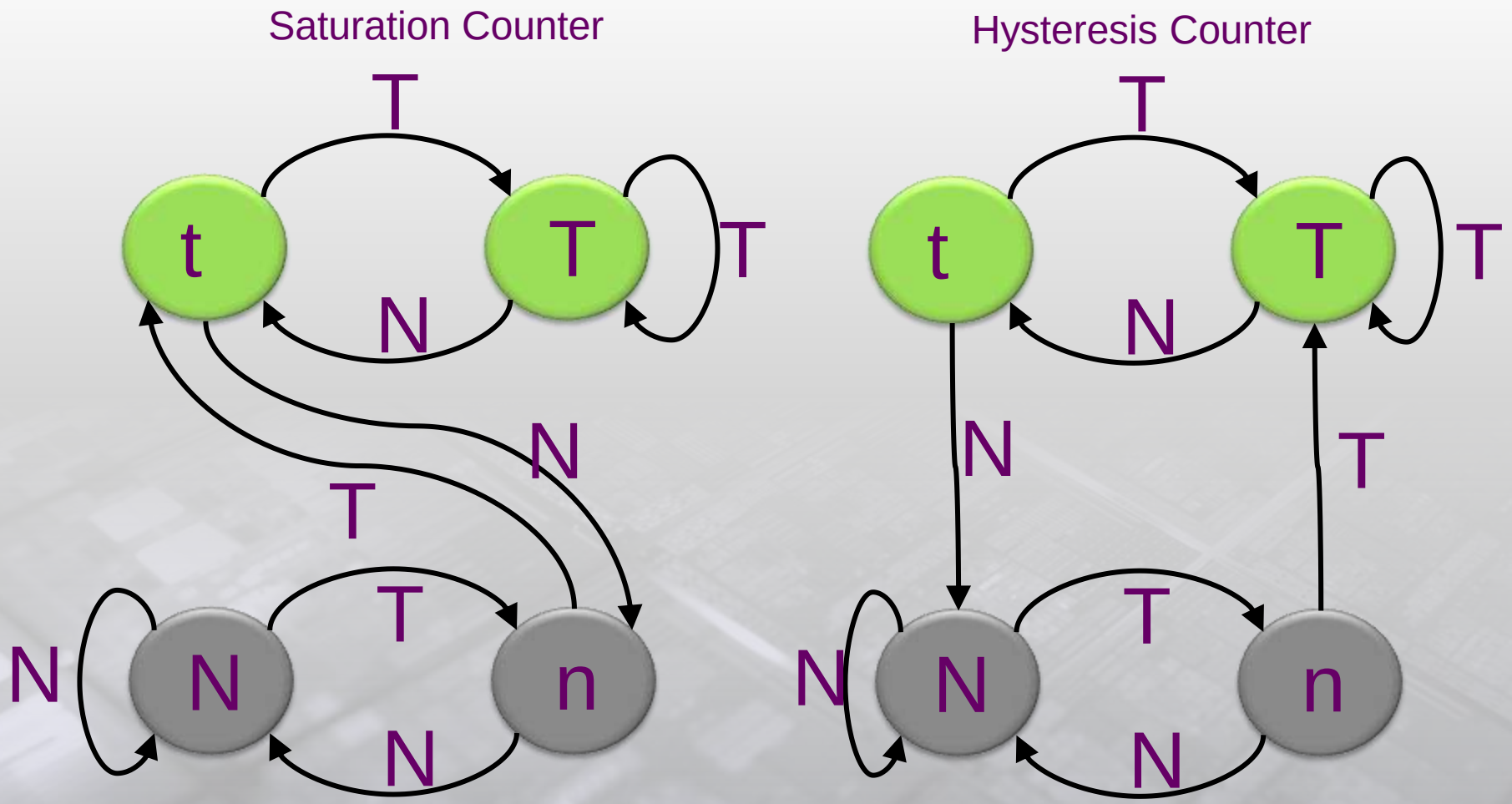
Two-Bit Saturating Counters (2bc)

- **Two-bit saturating counters (2bc)** [Smith]
  - 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"

State/prediction	**N***	**n***	**t**	**T***	**t**	**T**	**T**	**T***	**t**	**T**	**T**	**T***
Outcome	T	T	T	N	T	T	T	N	T	T	T	N

  - One mispredict each loop execution (rather than two)
    - \+ Fixes this pathology
    - Can we do even better?

Aside: Two different alternatives



Correlated Predictor

- **Correlated (two-level) predictor** [Patt, MICRO 1994]
  - Exploits observation that branch outcomes are correlated
  - Maintains separate prediction per (PC, BHR)
    - **Branch history register (BHR)**: recent branch outcomes
  - Simple working example: assume program has one branch (only one PC)
    - BHT: one 1-bit DIRP entry
    - BHT+**2BHR**: 2^2 = **4** 1-bit DIRP entries

State/prediction	BHR=NN	**N***	T	T	T	T	T	T	T	T	T	T	T
"active pattern"	BHR=NT	N	**N***	T	T	T	**T**	T	T	T	**T**	T	T
	BHR=TN	N	N	N	N	**N***	T	T	T	**T**	T	T	T
	BHR=TT	N	N	**N***	**T***	N	N	**N***	**T***	N	N	**N***	**T***
Outcome	N N	T	T	T	N	T	T	T	N	T	T	T	N

  - We didn't make anything better, what's the problem?

Correlated Predictor

- What happened?
  - BHR wasn't long enough to capture the pattern
  - Try again: BHT+**3BHR**: 2^3 = **8** 1-bit DIRP entries

State/prediction	BHR=NNN	**N***	T	T	T	T	T	T	T	T	T	T	T
	BHR=NNT	N	**N***	T	T	T	T	T	T	T	T	T	T
	BHR=NTN	N	N	N	N	N	N	N	N	N	N	N	N
"active pattern"	BHR=NTT	N	N	**N***	T	T	T	**T**	T	T	T	**T**	T
	BHR=TNN	N	N	N	N	N	N	N	N	N	N	N	N
	BHR=TNT	N	N	N	N	N	**N***	T	T	T	**T**	T	T
	BHR=TTN	N	N	N	N	**N***	T	T	T	**T**	T	T	T
	BHR=TTT	N	N	N	**N**	N	N	N	**N**	N	N	N	N
Outcome	N N N	T	T	T	N	T	T	T	N	T	T	T	N

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

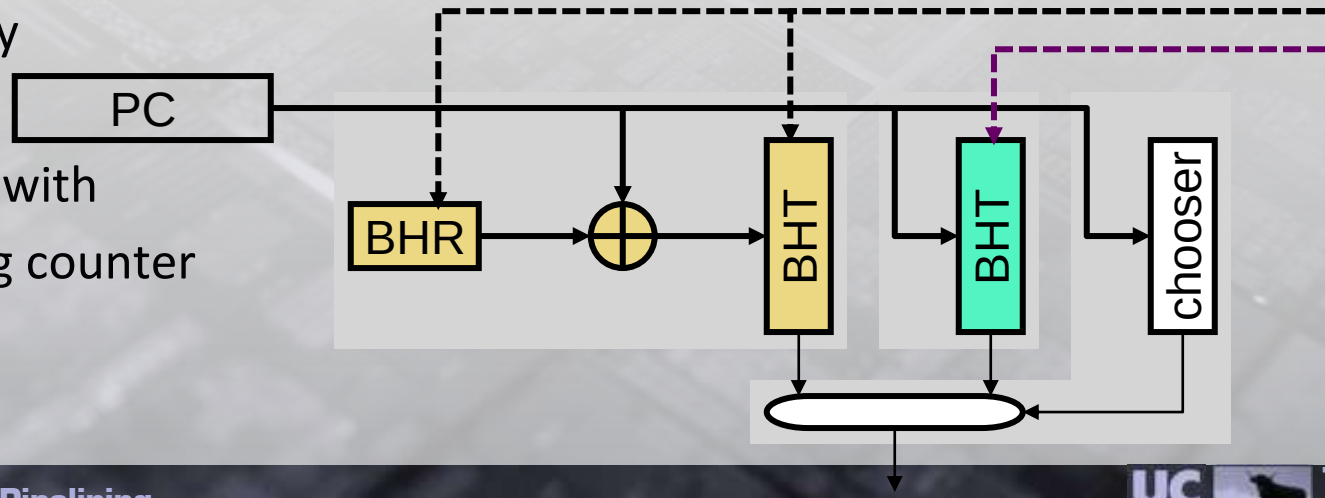
Correlated Predictor

- Design choice I: one **global** BHR or one per PC (**local**)?
  - Each one captures different kinds of patterns
  - Global is better, captures local patterns for tight loop branches

- 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 < $\log_2$(BHT size)
  – Predictor takes longer to train
  - Typical length: 8–12

Hybrid Predictor

- **Hybrid (tournament) predictor** [McFarling]
  - Attacks correlated predictor BHT utilization 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 (and update)
    - Branches start in simple BHT, move mis-prediction threshold
  - \+ Correlated predictor can be made smaller, handles fewer branches
  - \+ 90–95% accuracy
  - Alpha 21264
    Hybrid Gshare with
    2-bit saturating counter



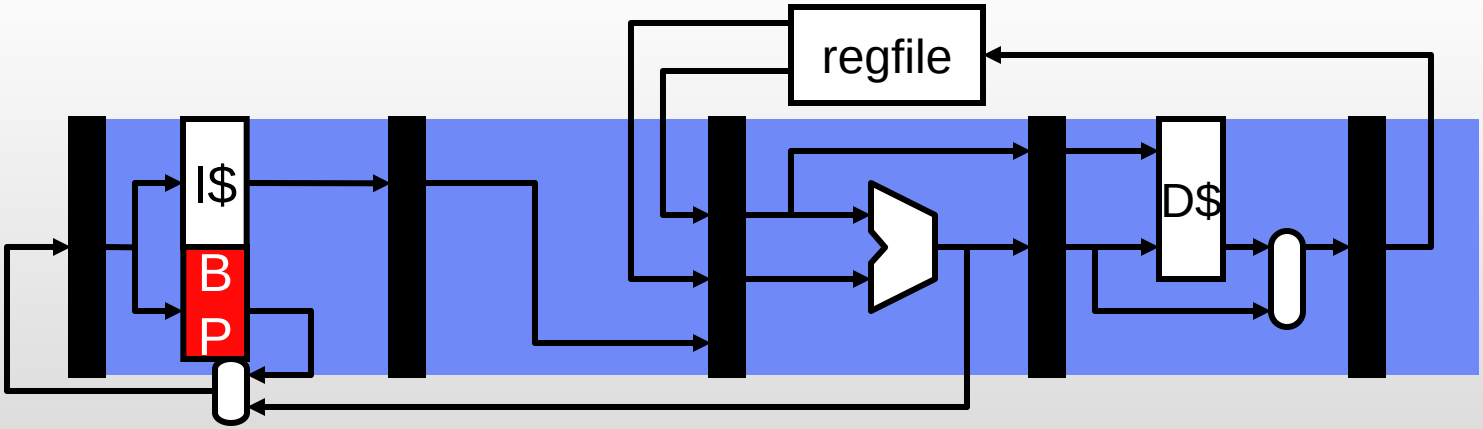
When to Perform Branch Prediction?

- 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

	1	2	3	4	5	6	7	8	9
`bnez r3,targ`	F	**D**	X	M	W				
`targ:add r4,r5,r4`			**F**	D	X	M	W		

- 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)

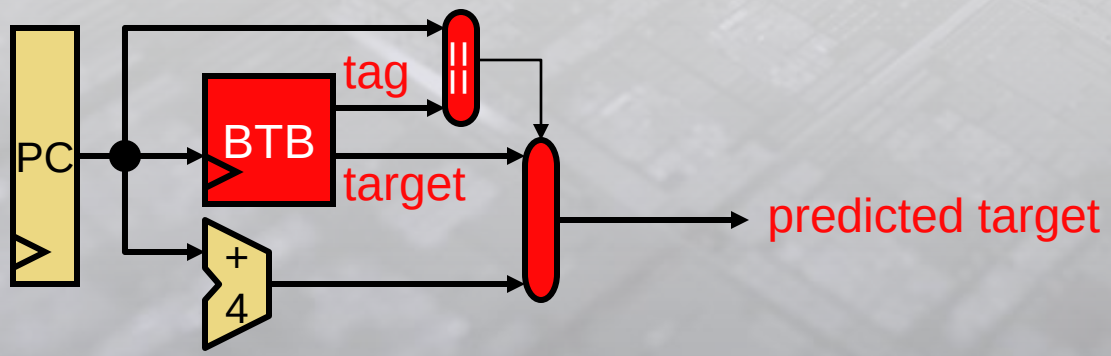
- As before: learn from past, predict the future
  - Record the past branch targets in a hardware structure

- **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
  - Like a cache: address = PC, data = target-PC
  - Access at Fetch *in parallel* with instruction memory
    - predicted-target = BTB[PC]
  - Updated at X whenever target != predicted-target
    - BTB[PC] = target
  - Aliasing? No problem, this is only a prediction
    - (Really?) What happens with miss-speculation recovery?

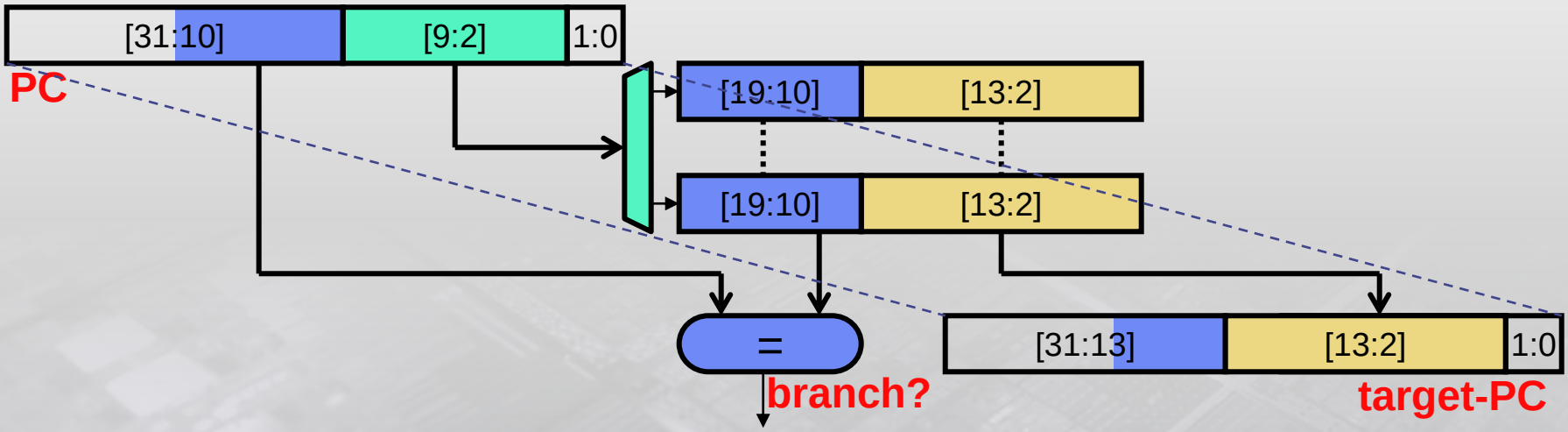
Branch Target Buffer (continued)

- At Fetch, how does insn know that it's a branch & should read BTB?
  - Answer: it doesn't have to, all insns read BTB
- Key idea: use BTB to predict which insn are branches
  - Tag each entry (with bits of the PC)
    - Just like a cache
  - Tag hit signifies instruction at the PC is a branch
  - Update only on taken branches (thus only taken branches in table)
- Access BTB at Fetch in parallel with instruction memory



Both: partial tags and PC-relative target-PC

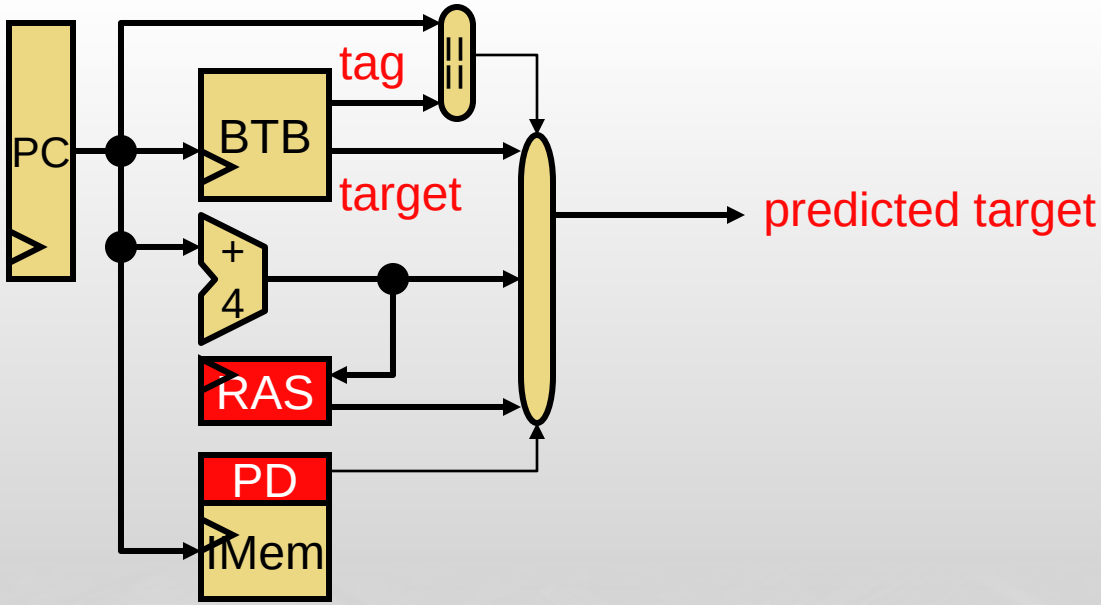
- Saving time in prediction is mandatory
- Broader alive branches for less BTB bits



Why Does a BTB Work?

- Because most control insns use **direct targets**
  - Target encoded in insn itself → same target every time
- What about **indirect targets**?
  - Target held in a register → can be different each time
  - Indirect conditional jumps are not widely supported
  - 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 …

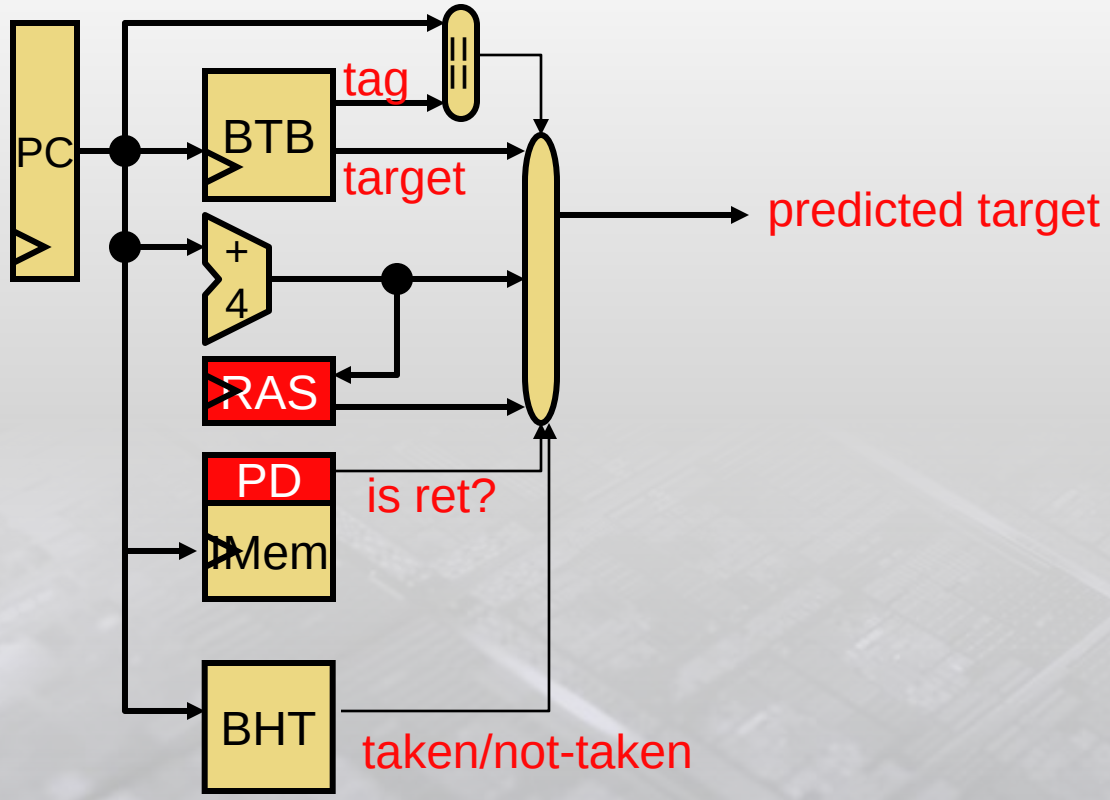
Return Address Stack (RAS)



- **Return address stack (RAS)**
  - Call instruction? RAS[TOS++] = PC+4
  - Return instruction? Predicted-target = RAS[--TOS]
  - 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: **pre-decode bits** in Imem, written when first executed
    - Can also be used to signify branches

Putting It All Together

- BTB & branch direction predictor during fetch



- If branch prediction correct, no taken branch penalty

Branch Prediction Performance

- Dynamic branch prediction
  - Simple predictor at fetch; branches predicted with 75% accuracy
    - CPI = 1 + (20% * **25%** * 2)= **1.1**
  - More advanced predictor at fetch: 95% accuracy
    - 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
  - Pipelines are superscalar (later): typical mis-prediction penalty is huge

- Advanced branch predictors could jeopardize fech stage delay
  - Solution I: Pipelining Branch prediction
  - Solution II: Speculate using a fast predictor (Fech next insn) and in parallel predict using a slower BP. If both disagree, assume mis-prediction Prophet/Critic branch predictors

Avoiding Branches via ISA: Predication

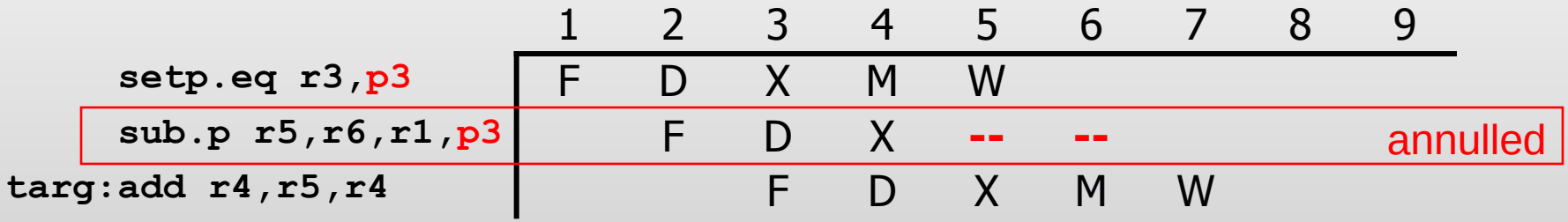
- Conventional control
  - Conditionally executed insns also conditionally fetched

	1	2	3	4	5	6	7	8	9
`beq r3,targ`	F	D	X	M	W				
`sub r5,r6,r1`		F	D	--	--	--	flushed: wrong path		
`targ:add r4,r5,r4`			F	--	--	--	--	flushed: why?	
`targ:add r4,r5,r4`				F	D	X	M	W	

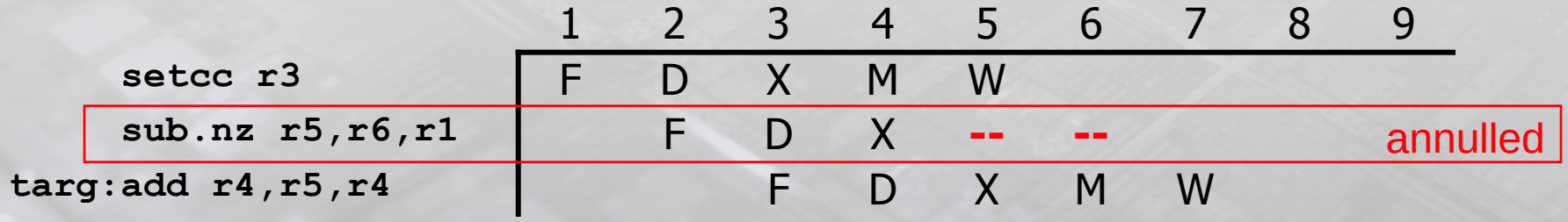
    - If **`beq`** mis-predicts, both **`sub`** and **`add`** must be flushed
    - Waste: **`add`** is independent of mis-prediction
- **Predication**: not prediction, predication
  - ISA support for conditionally-executed unconditionally-fetched insns
  - If **`beq`** mis-predicts, annul **`sub`** in place, preserve **`add`**
    - Example is if-then, but if-then-else can be predicated too
  - How is this done? How does **`add`** get correct value for **`r5`**?

Full Predication

- **Full predication**
  - Every insn can be annulled, annulment controlled by…
  - Predicate registers: additional register in each insn (e.g., IA64)



  - Predicate codes: condition bits in each insn (e.g., ARM)



  - Only ALU insn shown (**sub**), but this applies to all insns, even stores
  - Branches replaced with "set-predicate" insns

Conditional Register Moves (CMOVs)

- **Conditional (register) moves**
  - Construct appearance of full predication from one primitive

```
cmoveq r1,r2,r3          // if (r1==0) r3=r2;
```

  - May require some code duplication to achieve desired effect
  - Painful, potentially impossible for some insn sequences
  - Requires more registers
  - Only good way of retro-fitting predication onto ISA (e.g., IA32, Alpha)

```
      sub r9,r6,1
      cmovne r3,r9,r5
targ:add r4,r5,r4
```

	1	2	3	4	5	6	7	8	9
sub r9,r6,1	F	D	X	M	W				
cmovne r3,r9,r5		F	D	X	M	W			
targ:add r4,r5,r4			F	D	X	M	W		

Predication Performance

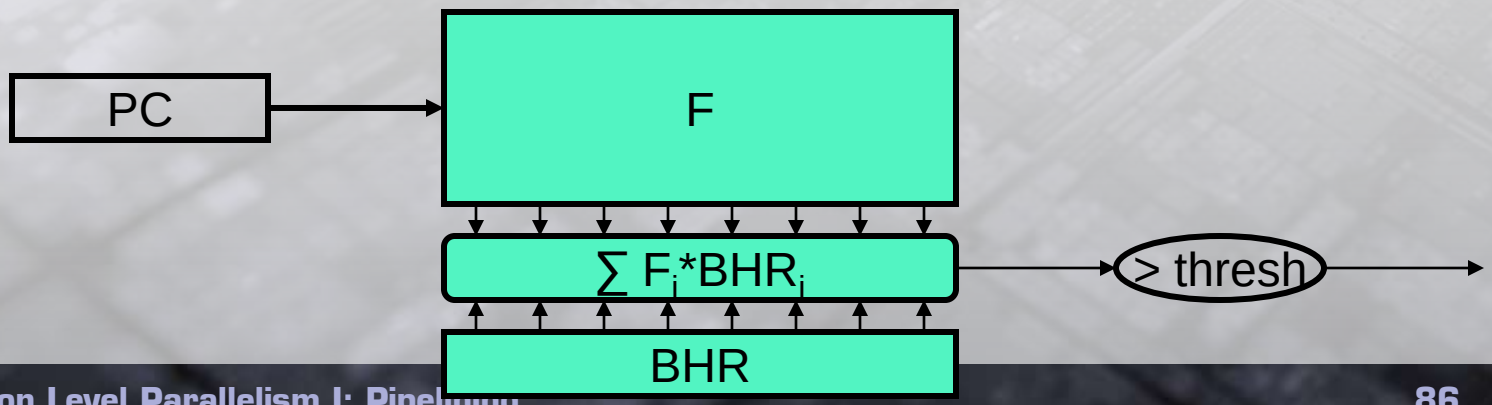
- Predication overhead is additional insns
  - Sometimes overhead is zero
    - Not-taken if-then branch: predicated insns executed
  - Most of the times it isn't
    - Taken if-then branch: all predicated insns annulled
    - Any if-then-else branch: half of predicated insns annulled
    - Almost all cases if using conditional moves
- Calculation for a given branch, predicate (vs speculate) if…
  - Average number of additional insns > overall mis-prediction penalty
  - For an individual branch
    - Mis-prediction penalty in a 5-stage pipeline = 2
    - Mis-prediction rate is <50%, and often <20%
    - Overall mis-prediction penalty <1 and often <0.4
  - So when is predication worth it?

Predication Performance

- What does predication actually accomplish?
  - In a scalar 5-stage pipeline (penalty = 2): nothing
  - In a 4-way superscalar 15-stage pipeline (penalty = 60): something
    - Use when mis-predictions >10% and insn overhead <6
  - In a 4-way out-of-order superscalar (penalty ~ 150)
    - Should be used in more situations
  - **Still**: only useful for branches that mis-predict frequently
- Strange: ARM typically uses scalar 5-9 stage pipelines
  - Why is the ARM ISA predicated then?
  - **Low-power:** eliminates the need for a complex branch predictor
  - **Real-time:** predicated code performs consistently
  - **Loop scheduling:** effective software pipelining requires predication

Research: Perceptron Predictor

- **Perceptron predictor** [Jimenez]
  - Attacks BHR size problem using machine learning approach
  - BHT replaced by table of function coefficients F_i (signed)
  - Predict taken if $\sum(BHR_i * F_i) >$ threshold
  + Table size #PC*|BHR|*|F| (can use long BHR: ~60 bits)
    - Equivalent correlated predictor would be $\#PC * 2^{|BHR|}$
  - How does it learn? Update F_i when branch is taken
    - $BHR_i == 1 \; ? \; F_i{+}{+} : F_i{-}{-}$;
    - "don't care" F_i bits stay near 0, important F_i bits saturate
  + Hybrid BHT/perceptron accuracy: 95–98%



More Research: GEHL Predictor

- Problem with both correlated predictor and perceptron
  - Same BHT real-estate dedicated to 1st history bit (1 column) …
  - … as to 2nd, 3rd, 10th, 60th…
  - Not a good use of space: 1st bit much more important than 60th

- **GEometric History-Length predictor** [Seznec, ISCA'05]
  - Multiple BHTs, indexed by geometrically longer BHRs (0, 4, 16, 32)
    - BHTs are (partially) tagged, not separate "chooser"
    - Predict: use matching entry from BHT with longest BHR
    - Mis-predict: create entry in BHT with longer BHR
  - \+ Only 25% of BHT used for bits 16-32 (not 50%)
    - Helps amortize cost of tagging
  - \+ Trains quickly
  - 95-97% accurate

Acknowledgments

- Slides developed by Amir Roth of University of Pennsylvania with sources that included University of Wisconsin slides by Mark Hill, Guri Sohi, Jim Smith, and David Wood.
- Slides enhanced by Milo Martin and Mark Hill with sources that included Profs. Asanovic, Falsafi, Hoe, Lipasti, Shen, Smith, Sohi, Vijaykumar, and Wood
- Slides re-enhanced by V. Puente of University of Cantabria