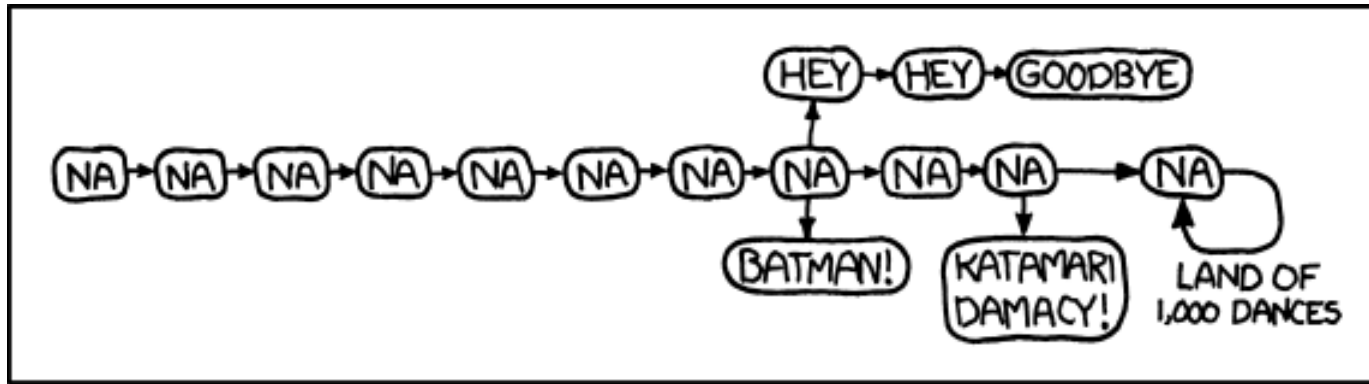


CS/ECE 752: Advanced Computer Architecture I



Prof. Matthew D. Sinclair

Source: XKCD

Pipelining

Slide History/Attribution Diagram:

UW Madison
Hill, Sohi,
Smith, Wood

UPenn
Amir Roth,
Milo Martin

Various Universities
Asanovic, Falsafi, Hoe, Lipasti,
Shen, Smith, Vijaykumar

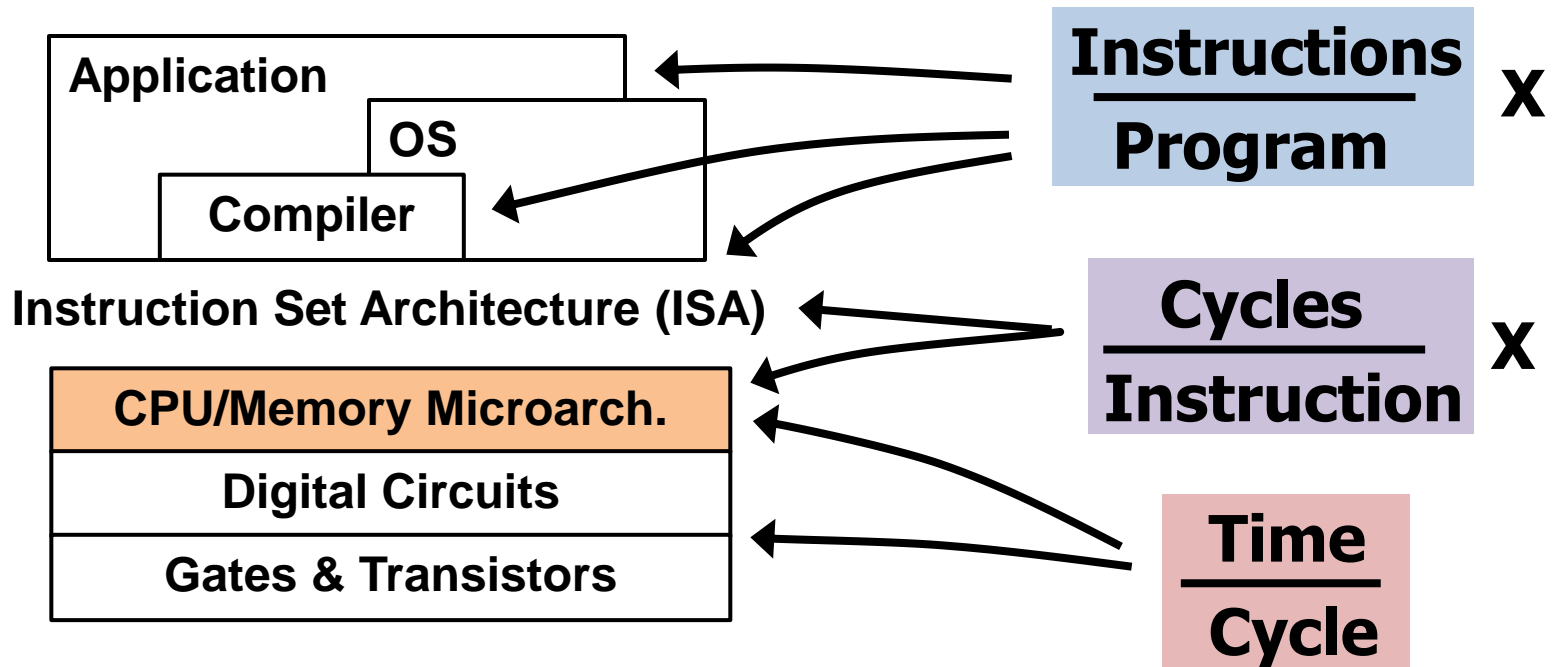
UW Madison
Hill, Sohi, Wood,
Sankaralingam, Sinclair

UCLA
Nowatzki

Announcements 9/17/24

- HW1 Grades Released
 - Let me know how I can help (if need help)
 - Solution posted on Canvas
 - Scripts are your friend – strongly recommend moving forward
 - Remember you need to do your own writeup, etc.
- HW2 Due Saturday
- Friday office hours moved to 1-2 PM
- HW3
 - Will probably be pushed back 1 week to allow more pre-Exam 1 material to be covered
 - New due date: 10/4/24
 - Because of quick turnaround with exam I will likely release solutions while also grading these

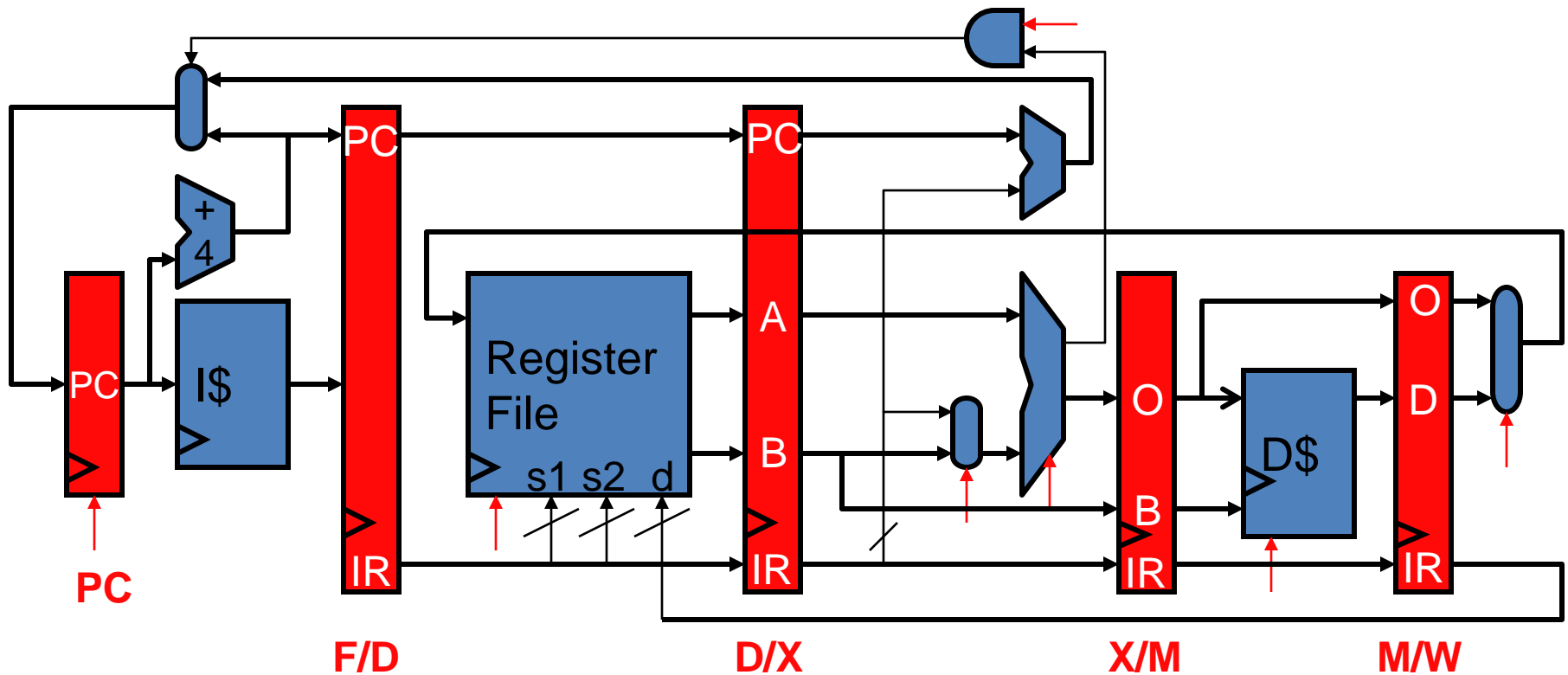
Computer System Layers



This Unit: Pipelining

- Basic Pipelining
 - Single, in-order issue
 - Clock rate vs. IPC
- Data Hazards
 - Hardware: stalling and bypassing
 - Software: pipeline scheduling
- Basic pipelining and data hazards are 552 review – available but not covered
- Control Hazards
 - Branch prediction
- Precise state

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**

More Terminology & Foreshadowing

- **Scalar pipeline**: one insn per stage per cycle (≈ 552)
 - Alternative: "superscalar" (later)
- **In-order pipeline**: insns enter execute stage in order (≈ 552)
 - Alternative: "out-of-order" (later)
- **Pipeline depth**: number of pipeline stages
 - Nothing magical about five
 - Contemporary high-performance cores have ~ 15 stage pipelines
 - (even Intel atom, an in-order core, uses 16 stages)

Control Dependences and Branch Prediction

In-Class Assignment

- With a partner, answer the following questions:
 - Is the complexity of TAGE worth it?
 - Is TAGE plausible to implement?
- In 3 minutes we'll discuss as a class

In-Class Assignment

Are a "free" now?

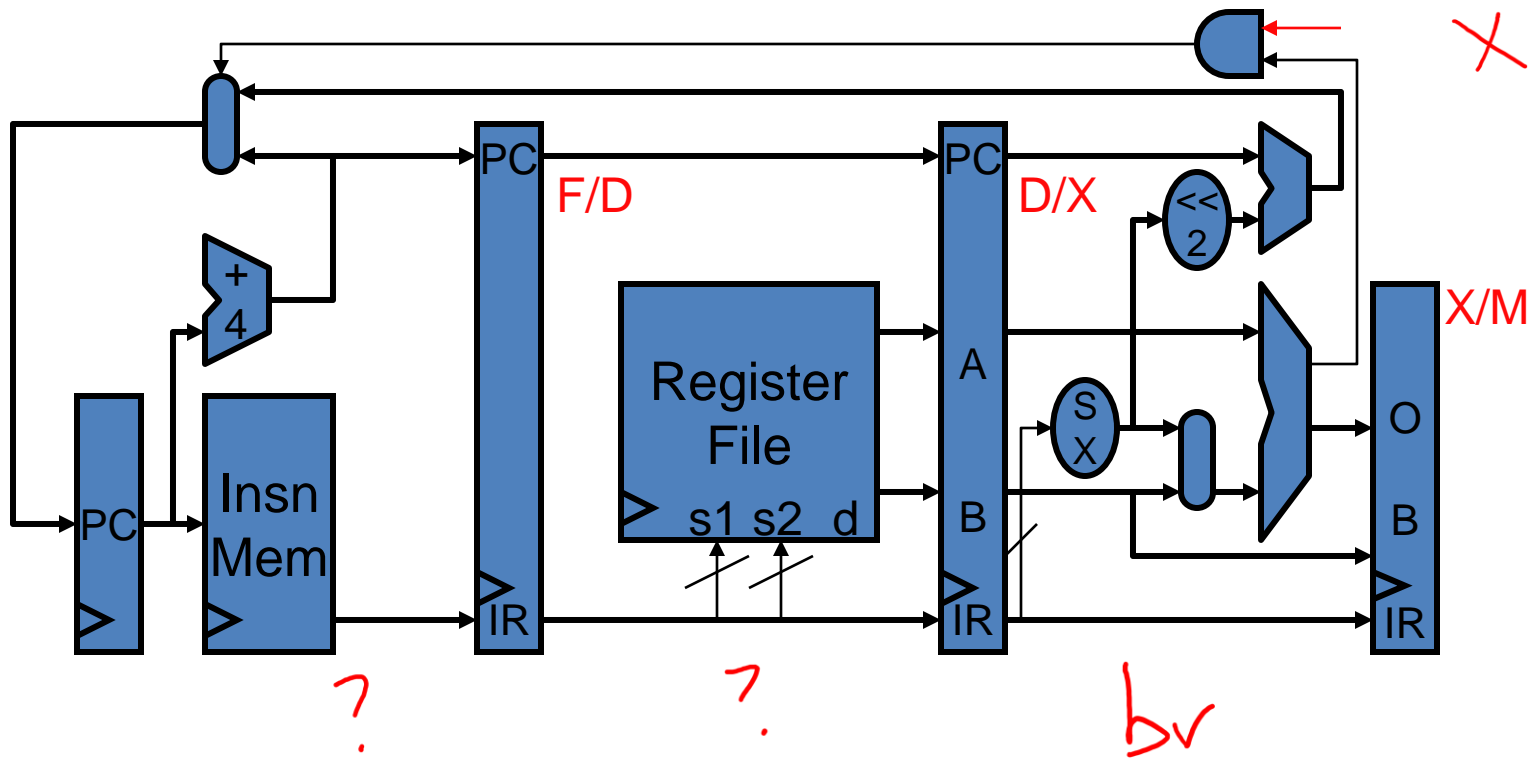
- With a partner, answer the following questions:
 - Is the complexity of TAGE worth it?

Highest accuracy vs. power vs. area vs. complexity
Mispredict = death for deep pipelines
↓
so often yes

- Is TAGE plausible to implement?

Originally unlimited area → no
subsequent work made more plausible
Now in modern procs.

What About Branches?



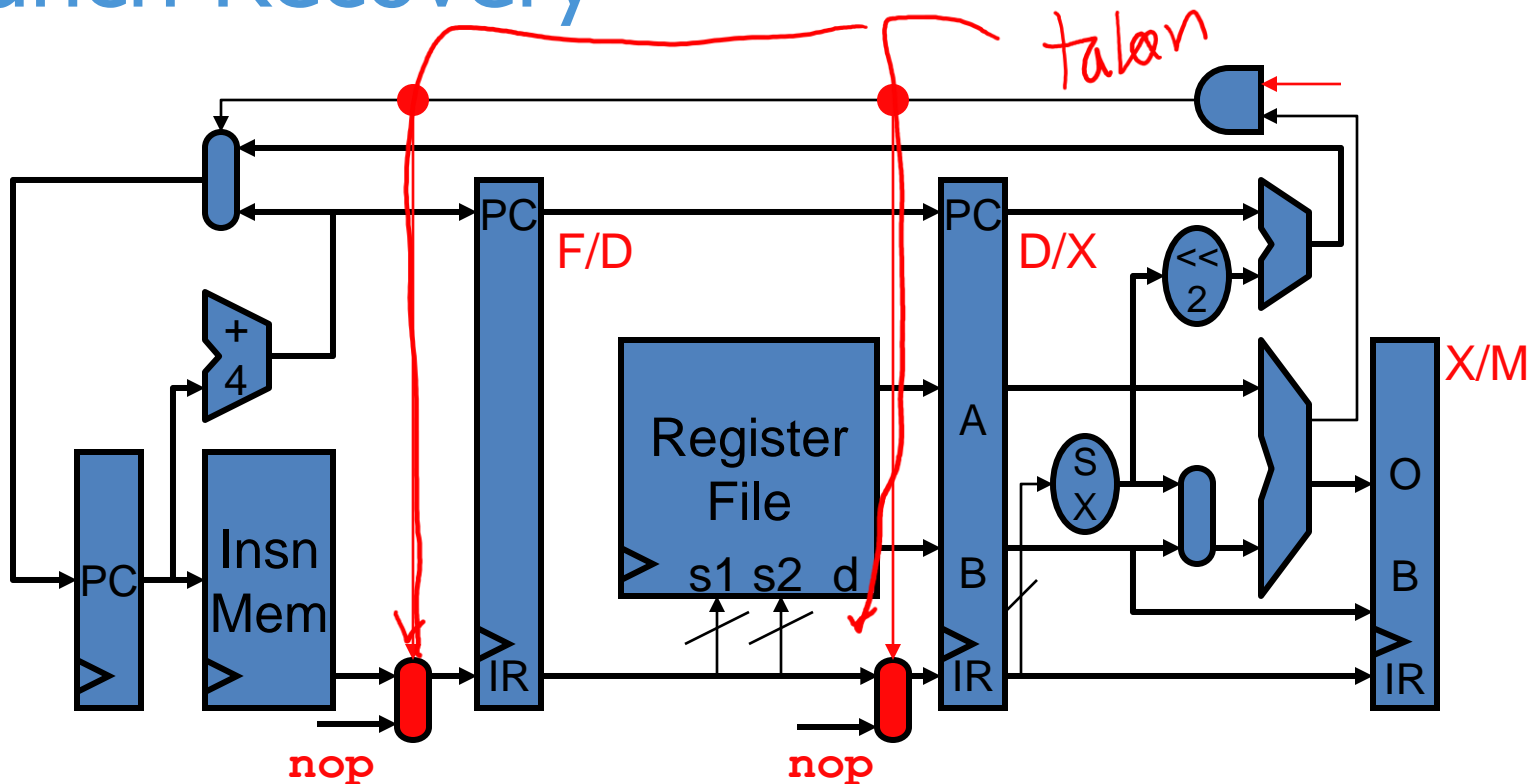
Control hazards options

1. Could just stall to wait for branch outcome (two-cycle penalty)

(6ue ss) 2. **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
- but Spectre/Meltdown*

Control Hazards

- **Control hazards**

- Control hazards indicated with **F*** (or not at all)
- Taken branch penalty is 2 cycles

	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*	F*	F	D	X	M	W

- Back of the envelope calculation

- **Branch: 20%**, other: 80%,
- Say, **75% of branches are taken**
- $CPI_{BASE} = 1$
- $CPI_{BASE+BRANCH} = 1 + 0.20 \cdot 0.75 \cdot 2 = 1.3$

wasted work
we flush

- **Branches cause 30% slowdown**

- Worse with deeper pipelines (higher misprediction penalty)

ISA Branch Techniques

- **Fast branch:** resolves at D, not X
 - Test must be comparison to zero or equality, no time for ALU
 - + New taken branch penalty is 1
 - Must bypass into decode now, too e.g., `cmplt`, `slt`
 - Complex tests still 2-cycle delay? Or just split into compare + branch?
- **Delayed branch:** branch that takes effect one insn later
 - Insert insns that are independent of branch into “branch delay slot”
 - Preferably from before branch (always helps then)
 - But from after branch OK too
 - As long as no undoable effects (e.g., a store)
 - Upshot: short-sighted feature (e.g., MIPS regrets it)
 - Not a big win in today’s pipelines
 - Complicates interrupt handling

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



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 → mis-speculation no worse than stalling
 - $\%_{\text{correct}} = \text{branch prediction}$
 - Static (compiler) ~85%, **dynamic** (hardware) >95%
 - Not much better? Static has 3X mispredicts!


Control Speculation and Recovery

Correct:

```
addi r1,1→r3
bnez r3,targ
st r6→(r7+4)
targ:add r4,r5→r4
```

	1	2	3	4	5	6	7	8	9
	F	D	X	M	W				
		F	D	X	M	W			
			F	D	X	M	W		
				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**)

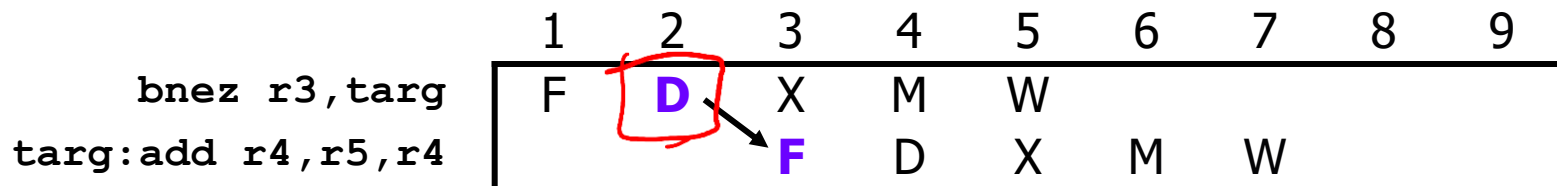
Recovery:

```
addi r1,1→r3
bnez r3,targ
st r6→(r7+4)
targ:add r4,r5→r4
targ:add r4,r5→r4
```

	1	2	3	4	5	6	7	8	9
	F	D	X	M	W				
		F	D	X	M	W			
			F	D	--	--	--		
				F	--	--	--	--	
					F	D	X	M	W

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**



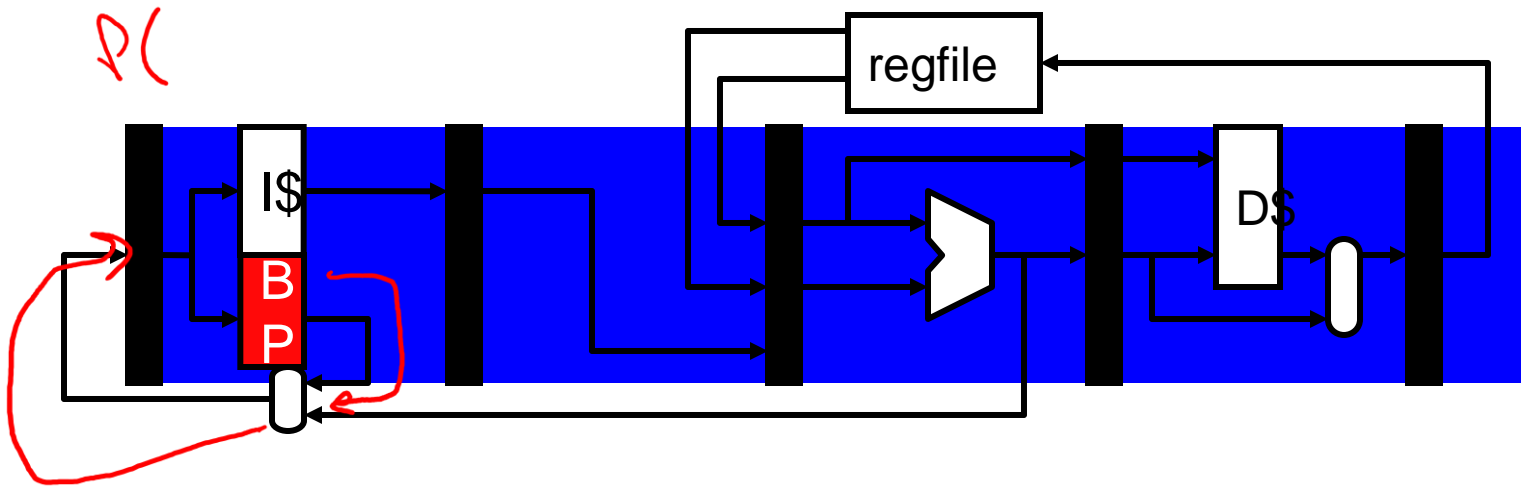
- Option #2: During Fetch?
 - How do we do that?

Announcements 9/19/24

report.pdf → report - <net ID>.pdf

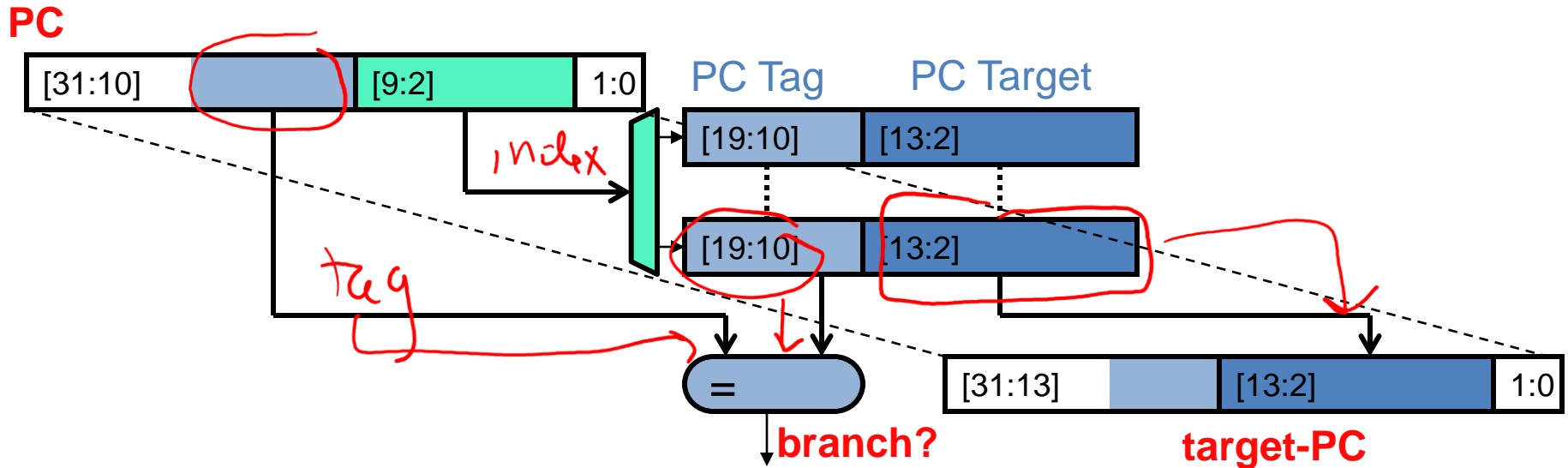
- HW2 Due Saturday
 - Lots of questions on Piazza (now aggregated into FAQ)
 - **Important:** redownload cpu.py if originally got before Wednesday
 - Added all of you to Gradescope last night
 - **If you weren't added please let me know ASAP**
 - **Please put your name in top right of first page**
- Friday (today's) office hours moved to 1-2 PM
- HW3
 - Pushed back 1 week to allow more pre-Exam 1 material to be covered
 - New due date: 10/4/24
 - Because of quick turnaround with exam I will likely release solutions during grading
- Next week content:
 - Moved today's "read" paper to next week to align with content

Dynamic Branch Prediction



- BP part I: **target predictor** (if taken)
 - Applies to all control transfers
 - Supplies target PC, tells if insn is a branch prior to decode + Easy
- BP part II: **direction predictor**
 - Applies to conditional branches only
 - Predicts taken/not-taken
 - Harder (or at least more options)

Branch Target Buffer (BTB)

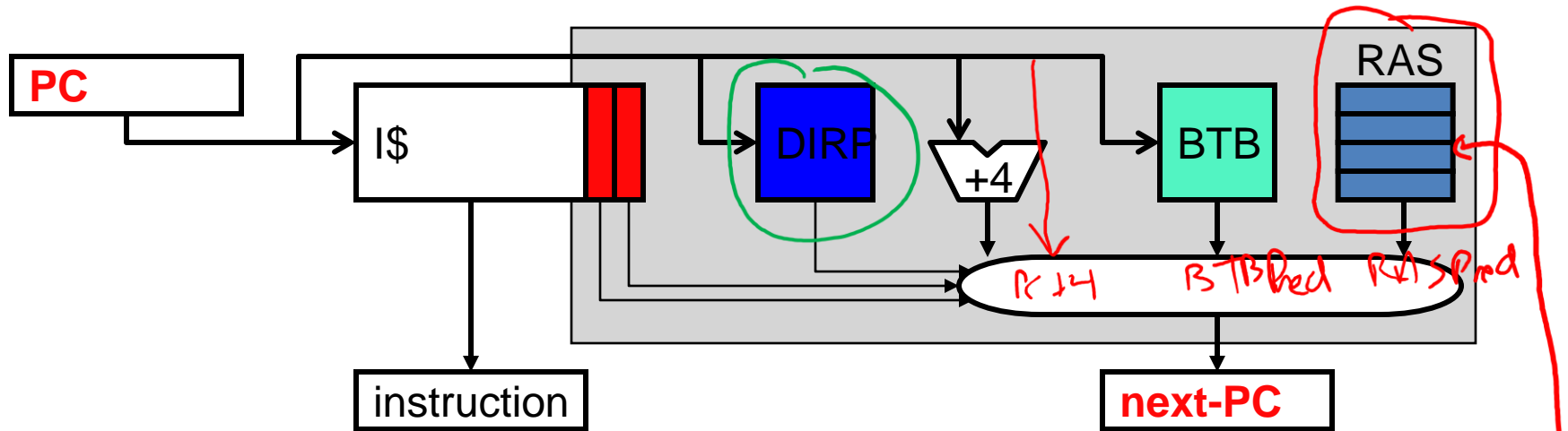


- A **small** cache: address = PC, data = target-PC
 - Hit? This is a control insn and it's going to target-PC (if "taken")
 - Miss? Not a control insn, or one I have never seen before
- Partial data/tags: full tag not necessary, target-PC is just a guess
 - **Aliasing**: tag match, but not actual match (OK for BTB)
- Insert into BTB when (taken) branch is resolved
- Pentium4 BTB: 2K entries, 4-way set-associative

Why Does a BTB Work?

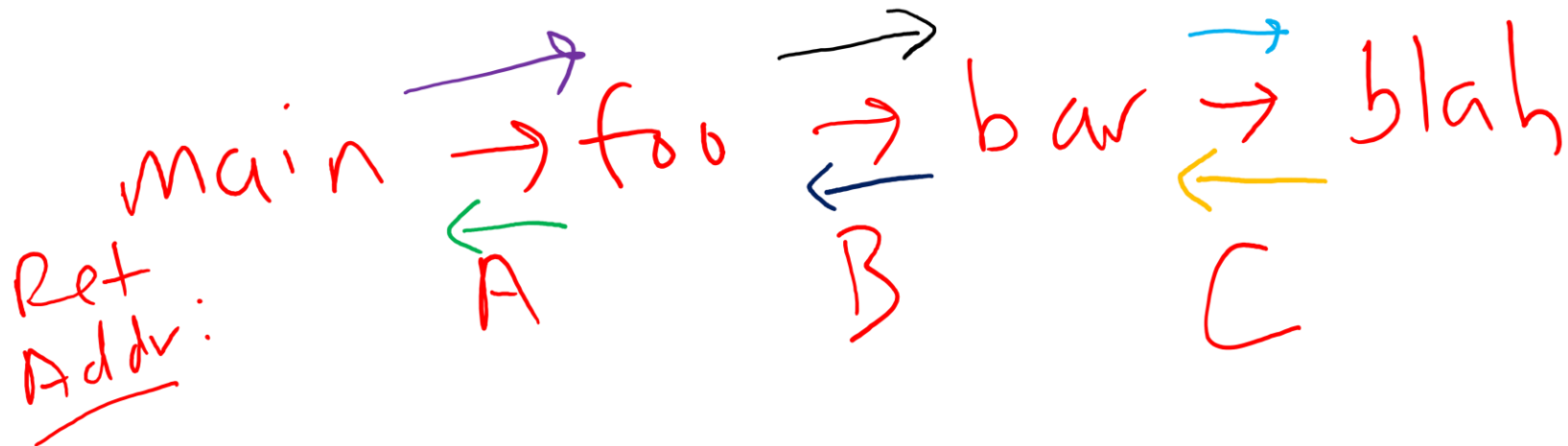
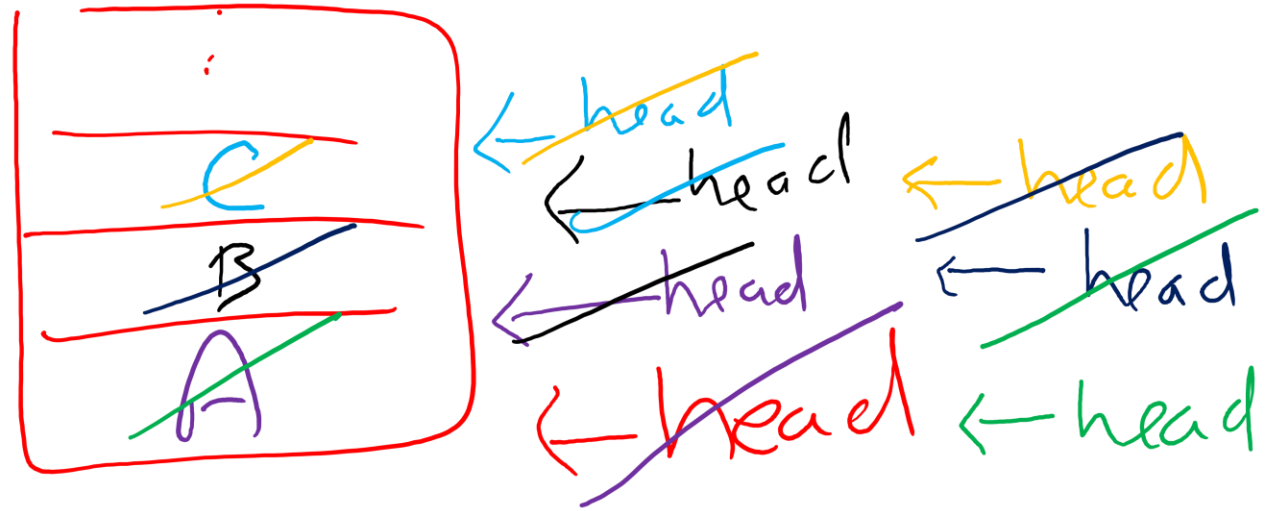
- Because control insn targets are stable
 - **Direct** means constant target, **indirect** means register target
 - + Direct conditional branches? ✓
 - + Direct calls? ✓
 - + Direct unconditional jumps? ✓
 - + Indirect conditional branches? Not that useful → not widely supported
 - Indirect calls? Two idioms:
 - + Dynamically linked functions (DLLs)? ✓
 - + Dynamically dispatched (virtual) functions? ✓—
 - Indirect unconditional jumps? Two idioms
 - Switches? ✗ but these are rare — diff't targets
 - Returns? ✓— but... we should know based on the program where we are returning!
↳ e.g. func. called from mult. places

Return Address Stack (RAS)



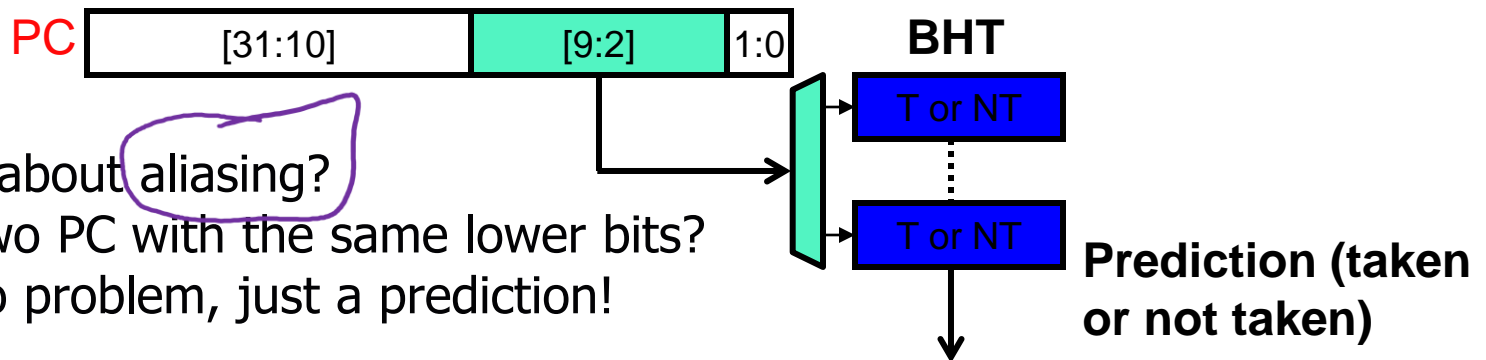
- Return addresses are easy to predict without a BTB
 - Hardware **return address stack (RAS)** tracks call sequence
 - Calls push $PC+4$ onto RAS
 - Prediction for returns is $RAS[TOS]$
 - Q: how can you tell if an insn is a return before decoding it?
 - A1: Add tags to make RAS a cache (have to check it...)
 - A2: (Better) attach **pre-decode bits** to I\$
 - Written after first time insn executes
 - Two useful bits: return?, conditional-branch?

RAS



Branch Direction Prediction

- **Direction predictor (DIRP)**
 - Map conditional-branch PC to taken/not-taken (T/N) decision
 - Can be based on additional information
- **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

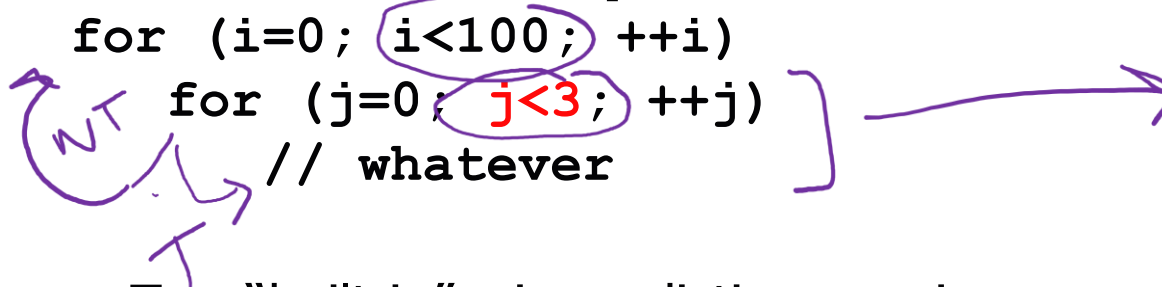


- What about aliasing?
 - Two PC with the same lower bits?
 - No problem, just a prediction!
- Why: Individual conditional branches often biased or weakly biased
 - 90%+ one way or the other considered **"biased"**
 - Why? Loop back edges, checking for uncommon conditions

Branch History Table (BHT)

- Problem: **inner loop branch** below

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

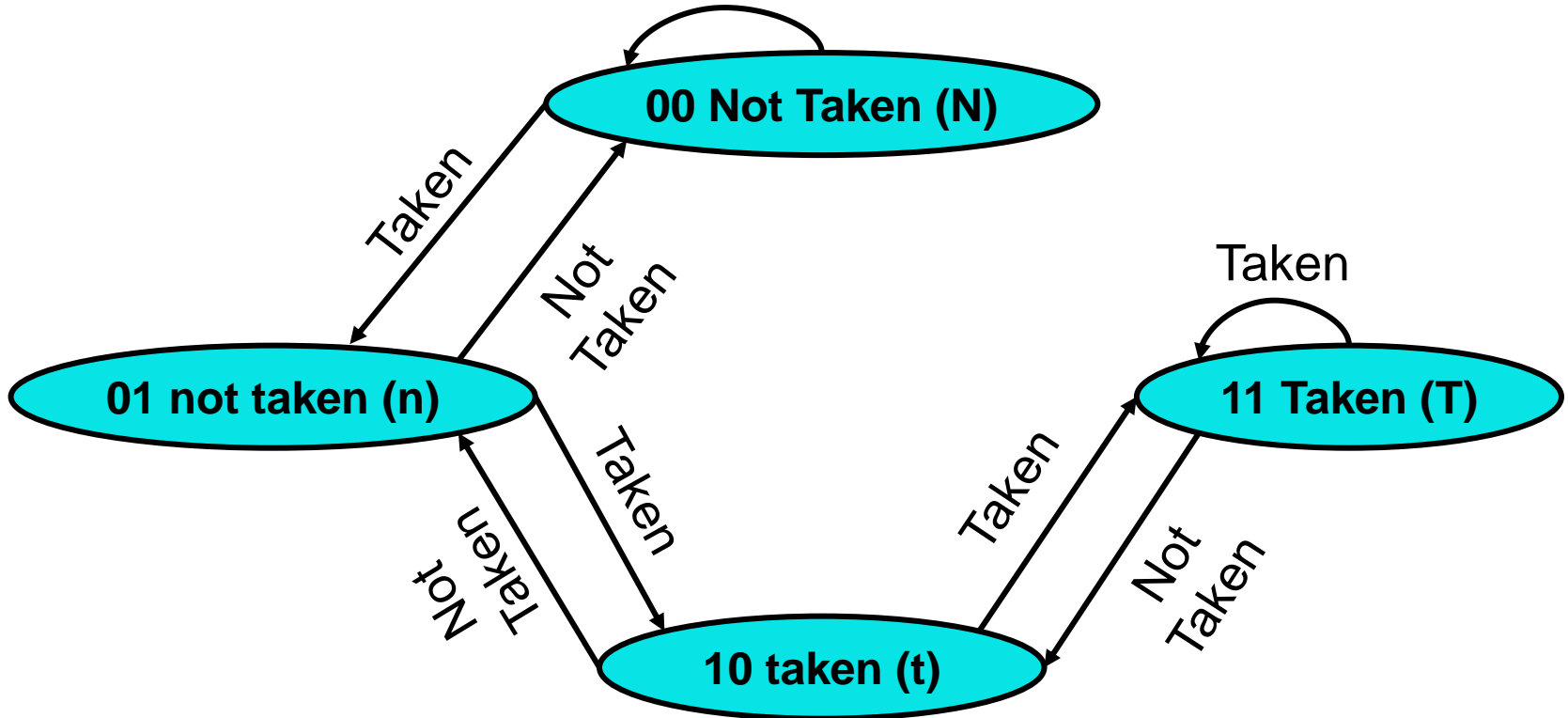


- Two “built-in” mis-predictions per inner loop iteration
- Branch predictor “changes its mind too quickly”

Time	Outcome
1	T
2	T
3	T
4	N
5	T
6	T
7	T
8	N
9	T
10	T
11	T
12	N

Two-Bit Saturating Counters (2bc)

- **Two-bit saturating counters (2bc)** [Smith 1981]
 - Replace each single-bit prediction
 - $(0,1,2,3) = (N,n,t,T)$
 - **Strong not-taken**, weak not-taken, weak taken, **strong taken**

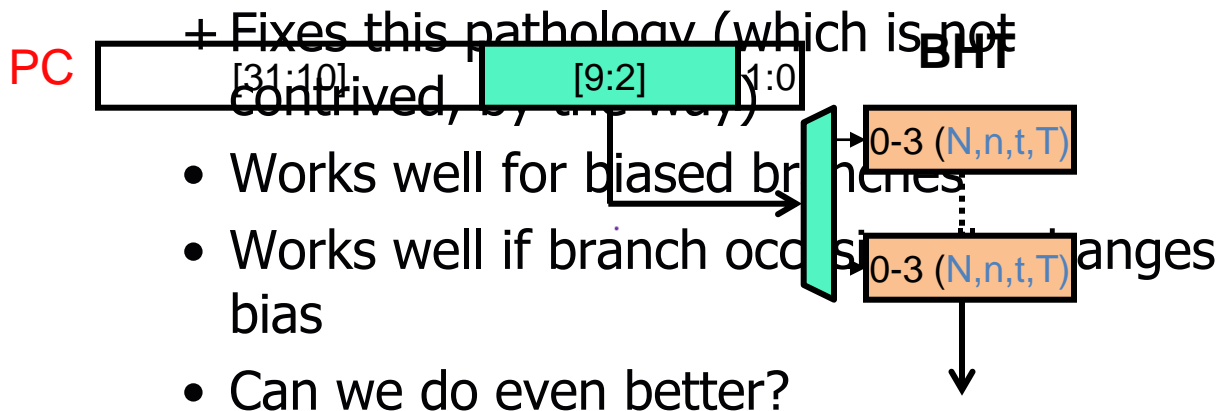


Two-Bit Saturating Counters (2bc)

- **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)

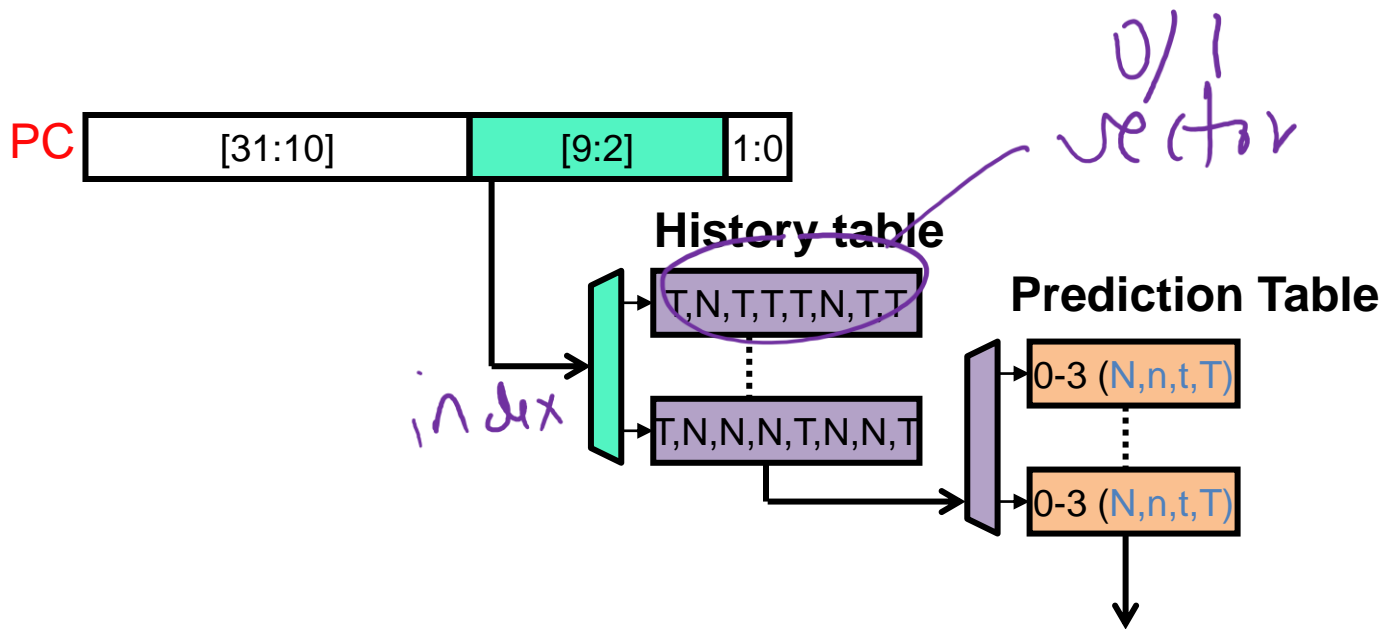


Time	State	Prediction	Outcome	Result?
1	N	N	T	Wrong
2	n	N	T	Wrong
3	t	T	T	Correct
4	T	T	N	Wrong
5	t	T	T	Correct
6	T	T	T	Correct
7	T	T	T	Correct
8	T	T	N	Wrong
9	t	T	T	Correct
10	T	T	T	Correct
11	T	T	T	Correct
12	T	T	N	Wrong

Two-level Predictor

Handwritten: $\text{if}(A) \rightarrow \text{if}(A \& B)$
 $\text{if}(B) \rightarrow \dots$

- **Correlated (two-level) predictor** [Patt 1991]
 - Exploits observation that branch outcomes are correlated
 - Branch history table stores past branches



Correlated Predictor \rightarrow Bit History

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

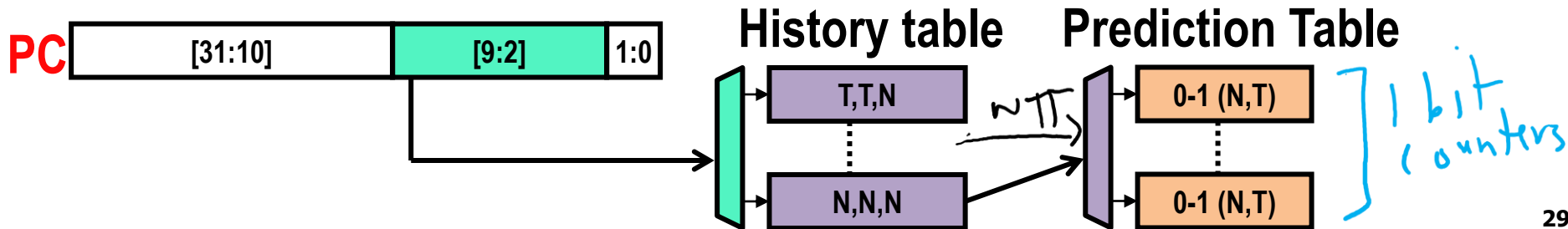
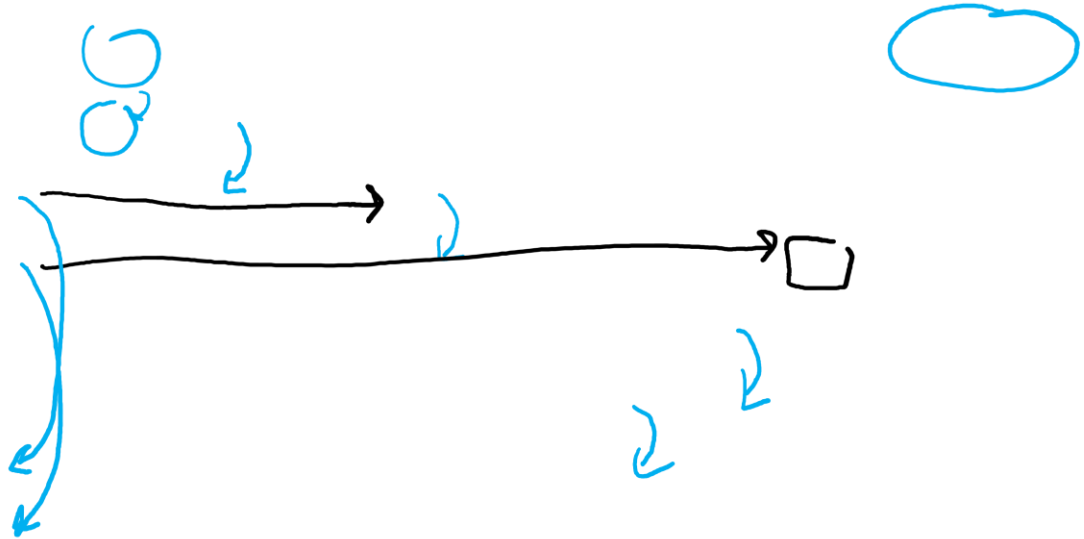
- Actual Pattern:

T,T,T,N,T,T,T,N,T

...

- Want:

- T,T,T \rightarrow
- N,T,T \rightarrow
- T,N,T \rightarrow
- T,T,N \rightarrow



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

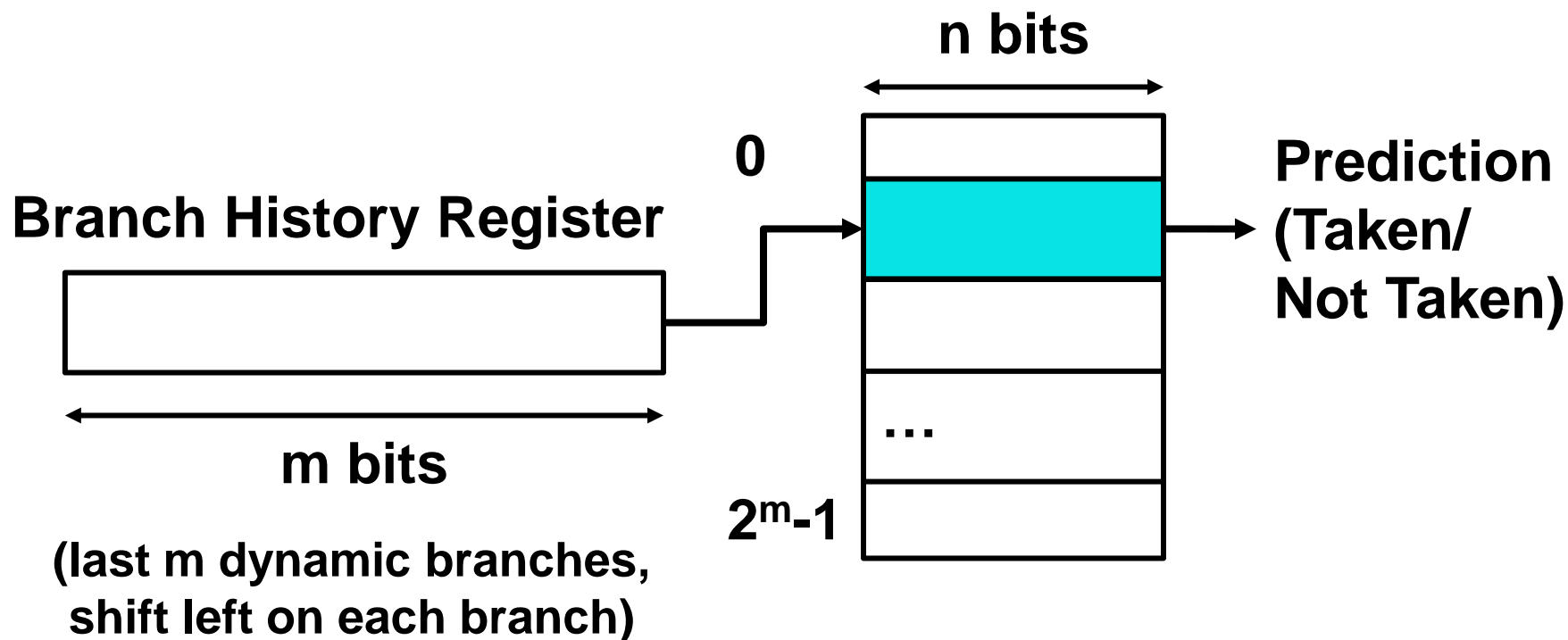
Time	"Pattern"	State				Prediction	Outcome		Result?
		NN	NT	TN	TT		Outcome	Result?	
1	NN	N	N	N	N	N	T	Wrong	
2	NT	T	N	N	N	N	T	Wrong	
3	TT	T	T	N	N	N	T	Wrong	
4	TT	T	T	N	T	T	N	Wrong	
5	TN	T	T	N	N	N	T	Wrong	
6	NT	T	T	T	N	T	T	Correct	
7	TT	T	T	T	N	N	T	Wrong	
8	TT	T	T	T	T	T	N	Wrong	
9	TN	T	T	T	N	T	T	Correct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

Correlated Predictor Design

- Design choice I: one **global** BHR or one per PC (**local**)?
 - Each one captures different kinds of patterns
 - Global 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 $< \log_2(\text{BHT size})$
 - Predictor takes longer to train
 - Typical length: 8–12

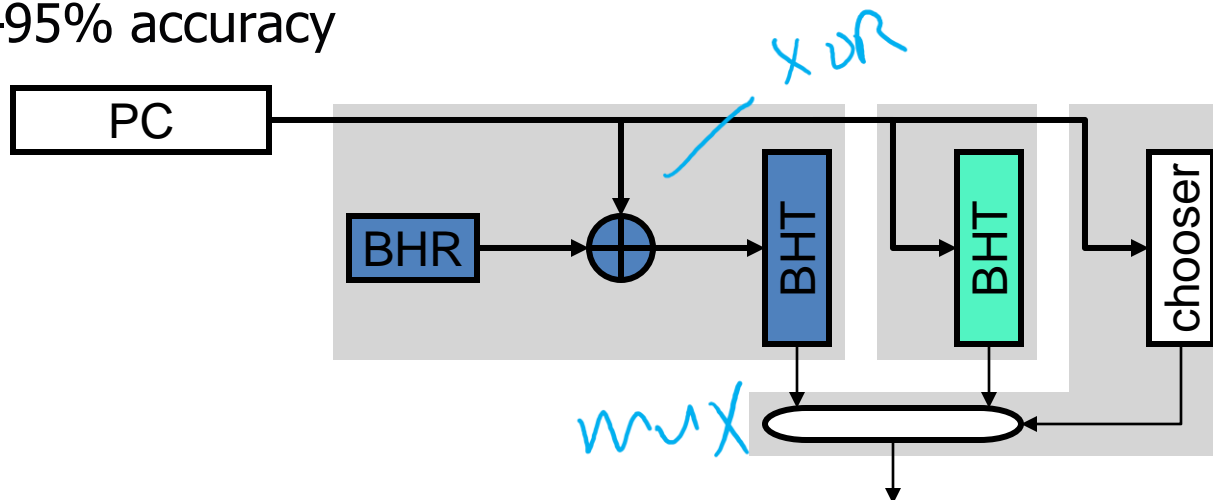
(m,n) Correlated Predictor

- Generalizing, an (m,n) predictor is:
 - $N = n$ -bit saturating counter
 - 2^n counters that can be indexed
 - $M = m$ -bit global history register
 - 2^m locations per PC (e.g., in BHT)



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



Branch Prediction Performance

- Same parameters
 - **Branch: 20%**, load: 20%, store: 10%, other: 50%
 - 75% of branches are taken
- Dynamic branch prediction
 - Branches predicted with 95% accuracy
 - $\text{CPI} = 1 + 0.20 \times 0.05 \times 2 = \mathbf{1.02}$

↑ Br ↑ 0% wrong ← 2 cycle penalty (flush)

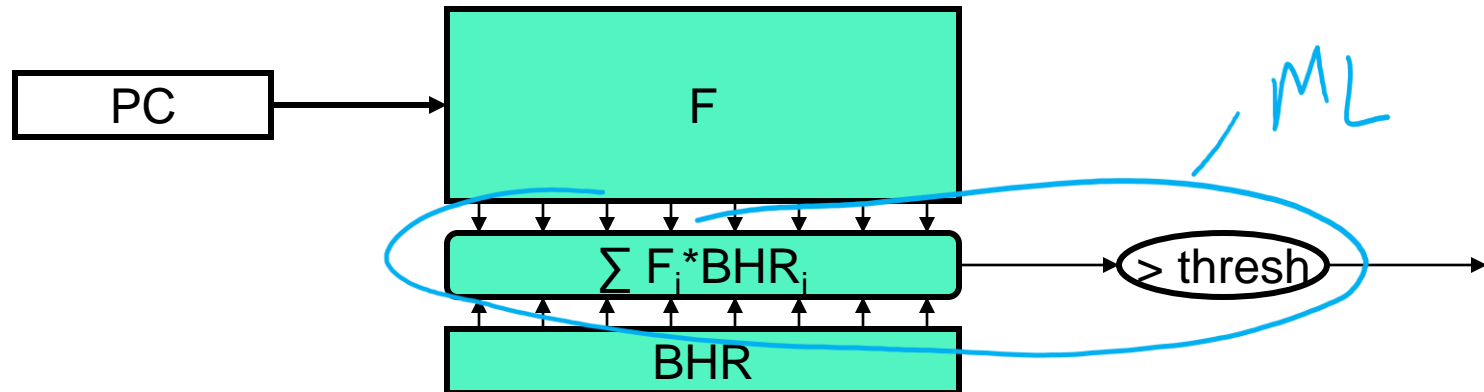
- **So are we done with branch prediction?**
 - **No, not yet ... penalties for out-of-order core are VERY HIGH even with 5% mispredictions**
 - More than 5 stages – CPI impacted more
 - This is where TAGE and Perceptron [Jimenez HPCA '01] come in
 - Aside: why is early 2000s work still state-of-art?

State-of-the-Art: TAGE

- Modern hybrid predictor
- Insights
 - Entry tagging → reduce aliasing
 - Partially tag entries – avoids aliasing problems with long(er) branch histories
 - But partial tags still help pick between options
 - Fine-grained entry selection → reduces warmup time
 - Track “goodness” of different branch scenarios for same branch
 - Effectively trades off *area/perf.*
 - Longer history
 - First two insights enable tracking more state without downsides
- Issue: massive amount of memory to store all of this
- VTAGE [Seznec HPCA '14] – more practical
- Modern CPUs largely use TAGE or Perceptron

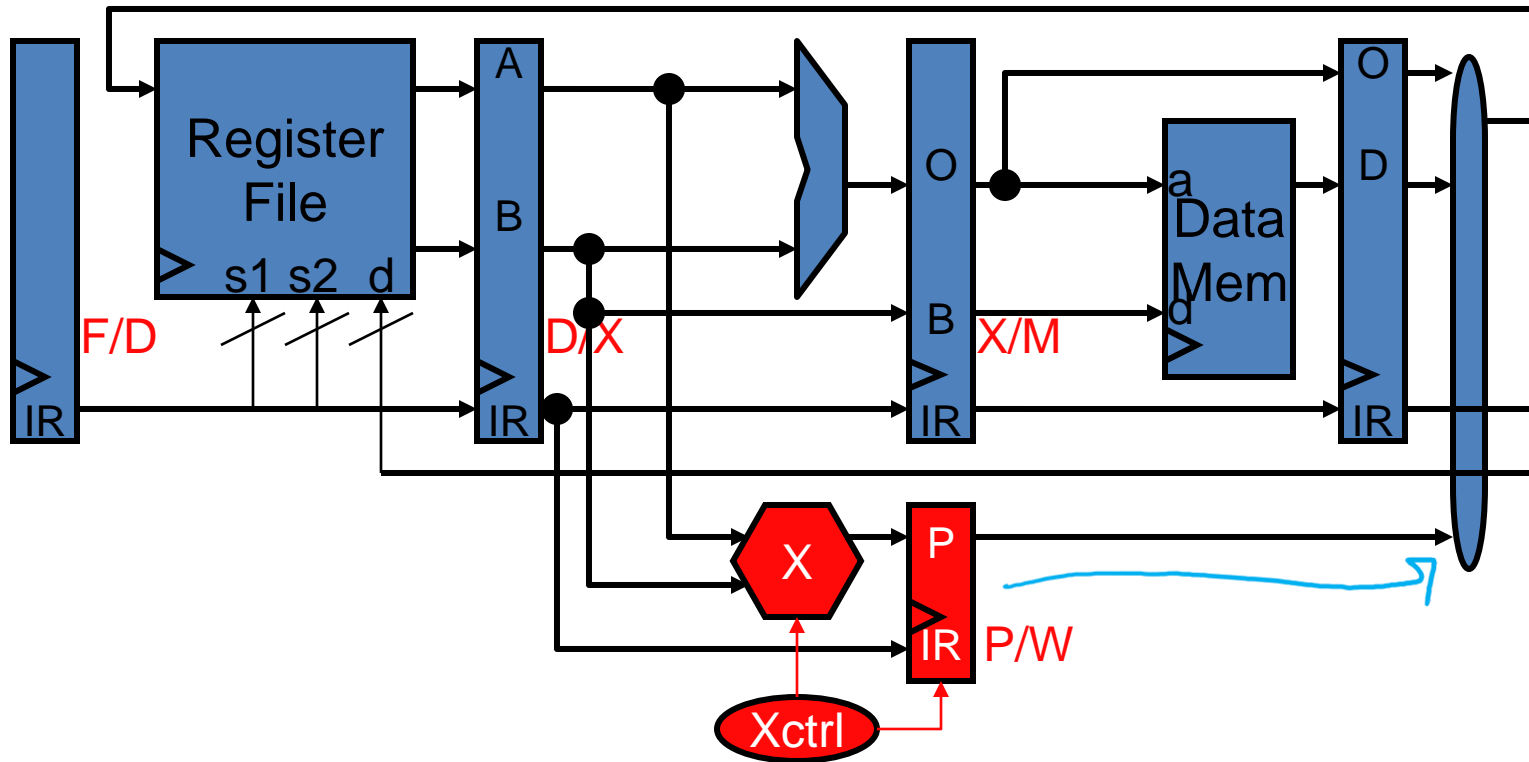
State-of-the-Art: Perceptron Predictor

- **Perceptron predictor** [Jimenez HPCA '01]
 - Attacks BHR size problem using machine learning
 - BHT replaced by table of function coefficients F_i (signed)
 - Predict taken if $\sum(BHR_i * F_i) > \text{threshold}$
- + 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% (now $\sim 99\%$)



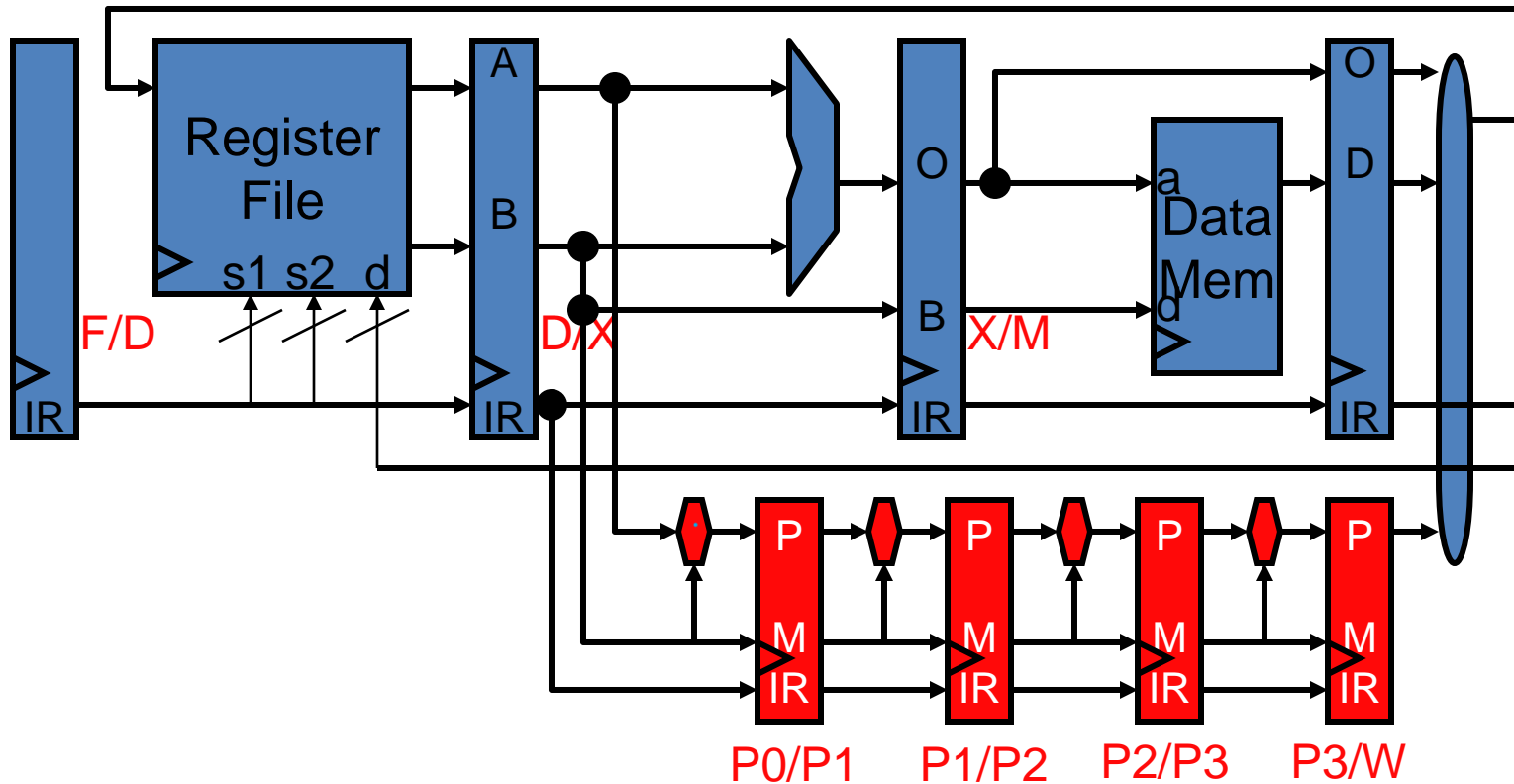
Multi Cycle/Pipelined Functional Units

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 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

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	D*	D*	D*	E/	E/	E/	E/	W

- **D*** = structural hazard (in this case)

Pipeline Diagram with Multiplier

- Allow **independent** instructions

	1	2	3	4	5	6	7	8	9
mul r4,r3→r5	F	D	P0	P1	P2	P3	W		
addi r6,1→r7		F	D	X	M	W			

- Even allow **independent multiply** instructions

	1	2	3	4	5	6	7	8	9
mul r4,r3→r5	F	D	P0	P1	P2	P3	W		
mul r6,r7→r8		F	D	P0	P1	P2	P3	W	

- But must stall subsequent **dependent** instructions

	1	2	3	4	5	6	7	8	9
mul r3,r5→r4	F	D	P0	P1	P2	P3	W		
addi r4,1→r6		F	D	D*	D*	D*	X	M	W

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
<code>mul r3,r5→r4</code>	F	D	P0	P1	P2	P3	W		
<code>addi r1,1→r6</code>		F	D	X	M	W			
<code>add r6,r10→r7</code>			F	D	X	M	W		

- Solution: stall the offending instruction:

	1	2	3	4	5	6	7	8	9
<code>mul r3,r5→r4</code>	F	D	P0	P1	P2	P3	W		
<code>addi r1,1→r6</code>		F	D	X	M	W			
<code>add r6,r10→r7</code>			F	D	D*	X	M	W	

WAW Hazards

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

add r2,r3→r1

sub r1,r4→r2

or r6,r3→r1

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

Multi-cycle WAW

- Mis-ordered writes to the same register
 - Software thinks `add` gets `r4` from `addi`, actually gets it from `mul`

	1	2	3	4	5	6	7	8	9
<code>mul r3,r5→r4</code>	F	D	P0	P1	P2	P3	W		
<code>addi r1,1→r4</code>		F	D	X	M	W			
...									
<code>Add r4,r6→r10</code>					F	D	X	M	W

- Solution: In order writes to same register

	1	2	3	4	5	6	7	8	9
<code>mul r3,r5→r4</code>	F	D	P0	P1	P2	P3	W		
<code>addi r1,1→r4</code>		F	d*	d*	D	X	M	W	
...									
<code>Add r4,r6→r10</code>					F	D	X	M	W

- Maybe we can just cancel the first write!

(and shift back first instruction)

WAW and Precise Interrupts

Optimizing WAW Hazards

	1	2	3	4	5	6	7	8	9	10
<code>divf f0, f1 → f2</code>	F	D	E/	E/	E/	E/	E/	W		
<code>stf f2 → (r1)</code>		F	D	d*	d*	d*	X	M	W	
<code>addf f0, f1 → f2</code>			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
<code>divf f0, f1 → f2</code>	F	D	E/	E/	E/	E/	E/	W		
<code>stf f2 → (r1)</code>		F	D	D*	D*	D*	X	M	W	
<code>addf f0, f1 → f2</code>			F	D	E+	E+	W			

- Von Neumann 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
<code>divf f0, f1 → f2</code>	F	D	E/	E/	E/	E/	E/	W		
<code>stf f2 → (r1)</code>		F	D	D*	D*	D*	X	M	W	
<code>addf f0, f1 → f2</code>			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
<code>divf f0, f1 → f2</code>	F	D	E/	E/	E/	E/	E/	W		
<code>stf f2 → (r1)</code>		F	D	D*	D*	D*	X	M	W	
<code>divf f0, f4 → f2</code>			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
- 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
 - `divf` /0 → rescale computation to prevent underflow
 - Typically doesn't restart computation at excepting instruction

WAR Hazards

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

`add r3, r1 → r2`

`sub r4, r2 → r5`

`or r3, r1 → r6`

- 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

<code>st r2 → (r1)</code> <code>ld (r1) → r4</code> <code>st r5 → (r1)</code> Read-after-write (RAW)	<code>st r2 → (r1)</code> <code>ld (r1) → r4</code> <code>st r5 → (r1)</code> Write-after-read (WAR)	<code>st r2 → (r1)</code> <code>ld (r1) → r4</code> <code>st r5 → (r1)</code> 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

	1	2	3	4	5	6	7	8	9	10
<code>st r2 → (r1)</code>	F	D	X	M	W					
<code>ld (r1) → r4</code>		F	D	X	M	W				

Pipeline Performance Summary

- Base CPI is 1, but hazards increase it
- Nothing magical about a 5 stage pipeline
 - Pentium4 has 22 stage pipeline
- Increasing **pipeline depth**
 - + Increases clock frequency (that's why companies used to do it)
 - But decreases IPC
 - Branch mis-prediction penalty becomes longer
 - More stages between fetch and whenever branch computes
 - Non-bypassed data hazard stalls become longer
 - More stages between register read and write
 - Ultimate metric is $IPC * frequency$
 - At some point, CPI losses offset clock gains

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
 - $IPC = 1 / (1 + X * N)$
 - $f = 1 / (G / N + O)$
 - Example: G = 80, O = 1, X = 0.16,

Optimizes performance!
What about power?



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

Dynamic Pipeline Power

- Remember control-speculation game
 - $[2 \text{ cycles} * \%_{\text{correct}}] - [\mathbf{0 \text{ cycles}} * (1 - \%_{\text{correct}})]$
 - No penalty \rightarrow mis-speculation no worse than stalling
 - This is a performance-only view
 - From a power standpoint, mis-speculation is worse than stalling
- **Power control-speculation game**
 - $[0 \text{ nJ} * \%_{\text{correct}}] - [\mathbf{X \text{ nJ}} * (1 - \%_{\text{correct}})]$
 - No benefit \rightarrow correct speculation no better than stalling
 - Not exactly, increased execution time increases static power
 - How to balance the two?

Trends...

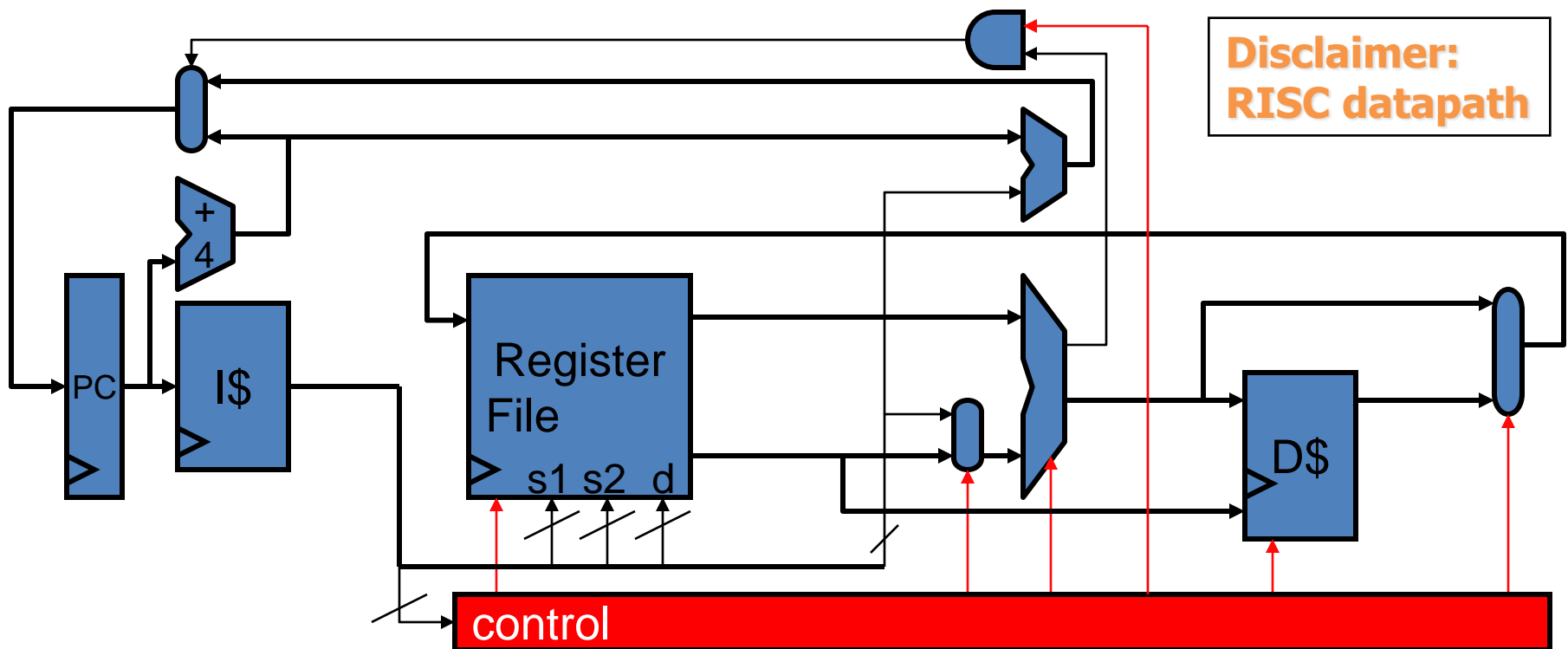
- Trend has been for deeper pipelines
 - Intel example:
 - 486: 5 stages (50+ gate delays / clock)
 - Pentium: 7 stages
 - Pentium II/III: 12 stages
 - Pentium 4: 22 stages (10 gate delays / clock)
 - 800 MHz Pentium III was faster than 1 GHz Pentium4
 - Intel Core2: 14 stages, less than Pentium 4
 - Nehalem (2008): 20-24 Stages
 - Haswell (2013): 14-19 Stages
 - Skylake (2017): 14-19 Stages
 - Cooper Lake (2019): 14-19 Stages

Summary

- Principles of pipelining
 - Effects of overhead and hazards
 - Pipeline diagrams
- Data hazards
 - Stalling and bypassing
- Control hazards
 - Branch prediction
- Power techniques
 - Dynamic power: speculation gating
 - Static and dynamic power: razor latches

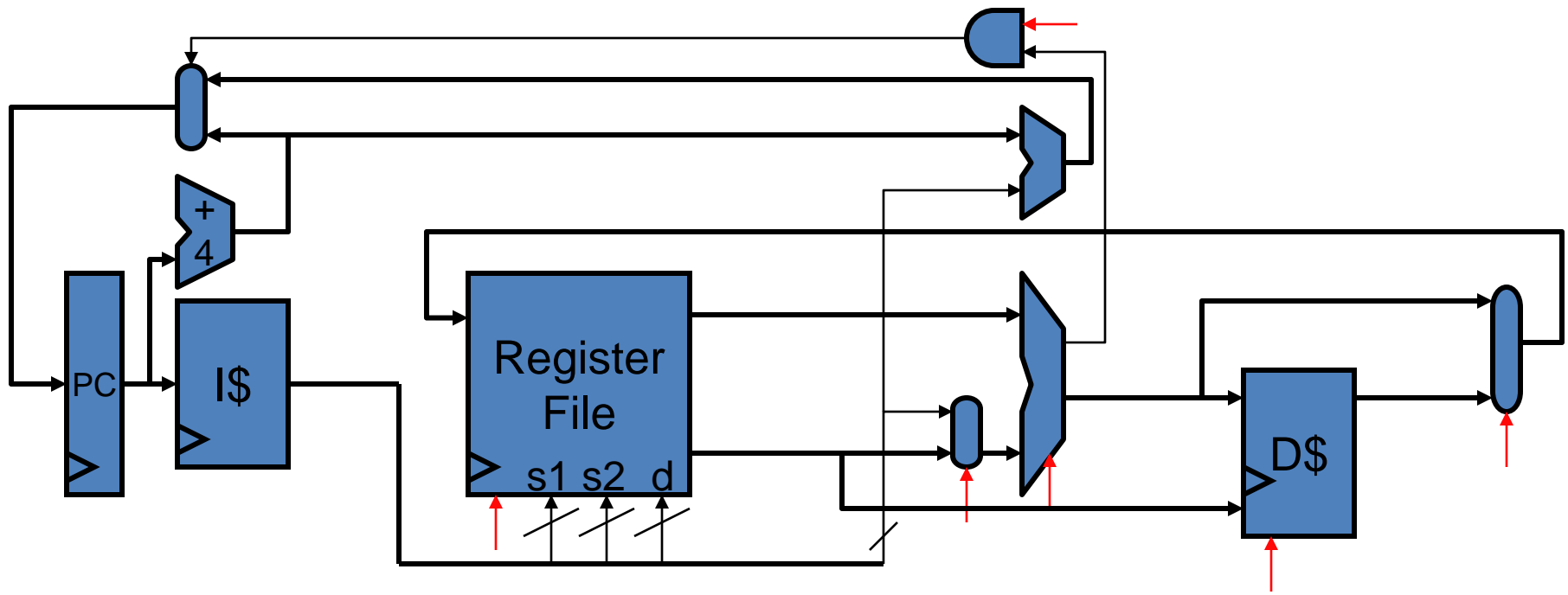
552 Review

Datapath and Control



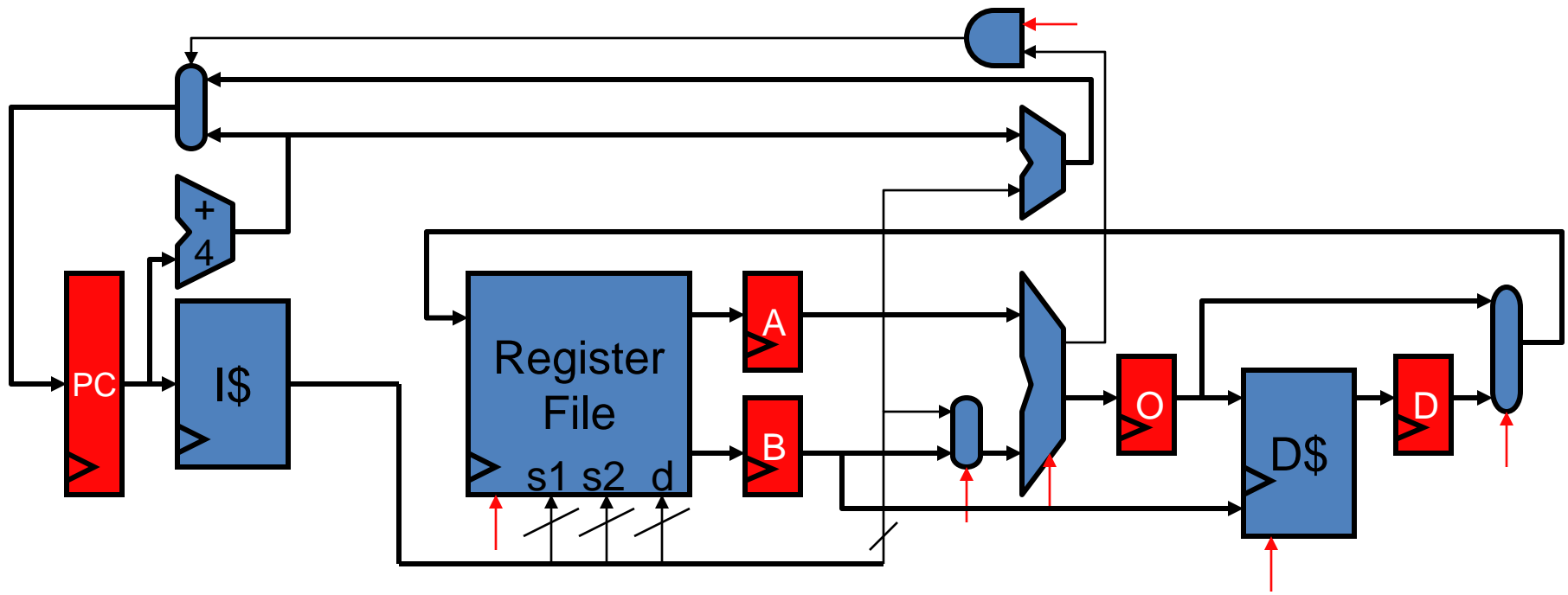
- **Datapath:** implements execute portion of fetch/exec. loop
 - Functional units (ALUs), registers, memory interface
- **Control:** implements decode portion of fetch/execute loop
 - Mux selectors, write enable signals regulate flow of data in datapath
 - Part of decode involves translating insn opcode into control signals

Single-Cycle Datapath



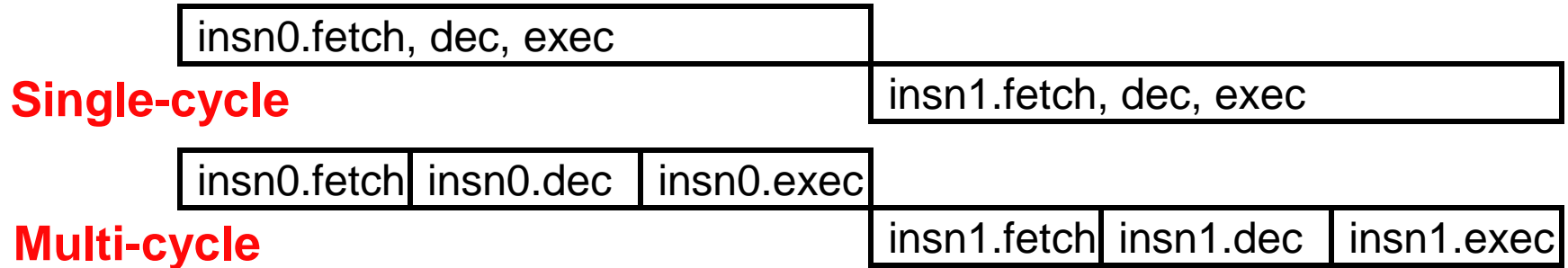
- **Single-cycle datapath:** true “atomic” fetch/execute loop
 - Fetch, decode, execute one complete instruction every cycle
 - + Takes 1 cycle to execution any instruction by definition (“CPI” is 1)
 - Long clock period: to accommodate slowest instruction (worst-case delay through circuit, must wait 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** (main point)
 - + Opposite of single-cycle: short clock period (less "work" per cycle)
 - Multiple cycles per instruction (higher "CPI")

Single-cycle vs. Multi-cycle

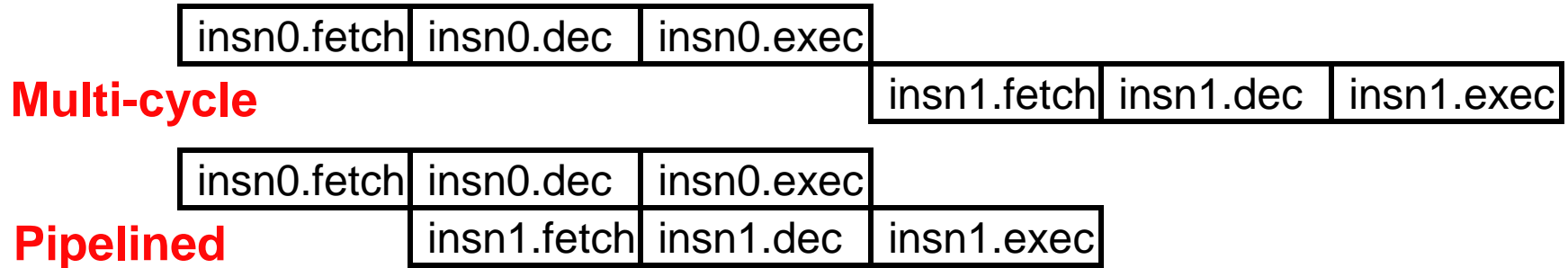


- **Single-cycle datapath:**
 - Fetch, decode, execute one complete instruction every cycle
 - + Low CPI: 1 by definition
 - Long clock period: to accommodate slowest instruction
- **Multi-cycle datapath:** attacks slow clock
 - Fetch, decode, execute one complete insn over multiple cycles
 - + Short 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

Single-cycle vs. Multi-cycle Performance

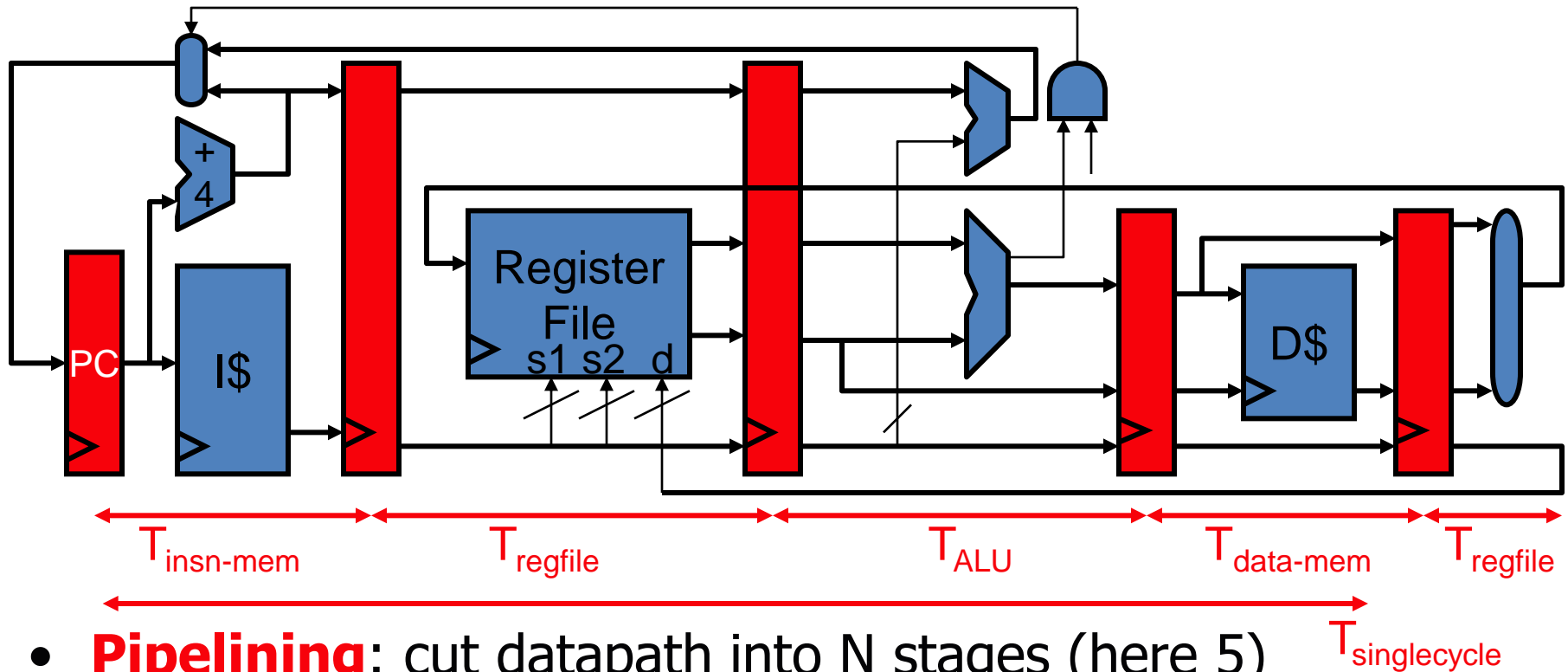
- Single-cycle
 - Clock period = 50ns, CPI = 1
 - Performance = **50ns/insn**
- Multi-cycle has opposite performance split of single-cycle
 - + Shorter clock period
 - Higher CPI
- Multi-cycle
 - Branch: 20% (**3** cycles), load: 20% (**5** cycles), ALU: 60% (**4** cycles)
 - Clock period = **11ns**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

Pipelining



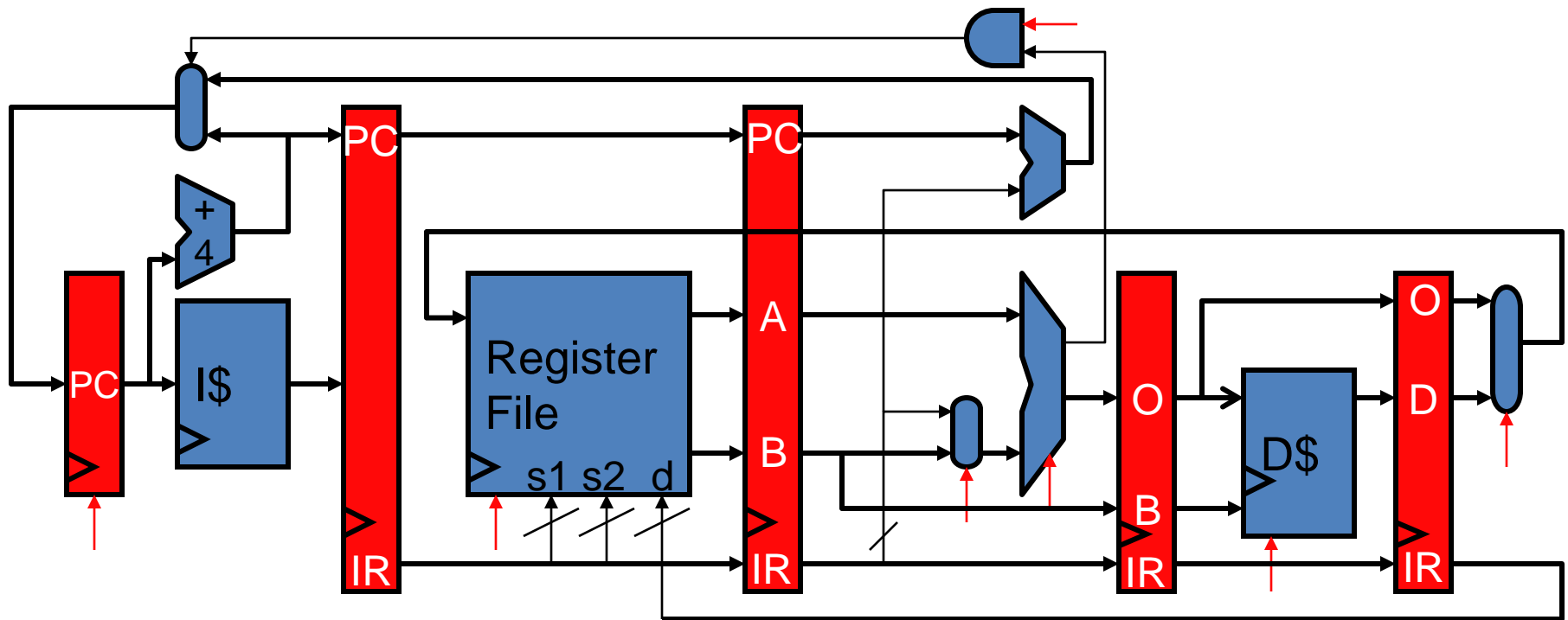
- Important performance technique
 - **Improves instruction throughput rather instruction latency**
- Begin with multi-cycle design
 - When instruction advances from stage 1 to 2
 - Allow next instruction to enter stage 1
 - Form of parallelism: “insn-stage parallelism”
 - Individual instruction takes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

Five Stage Pipeline Performance



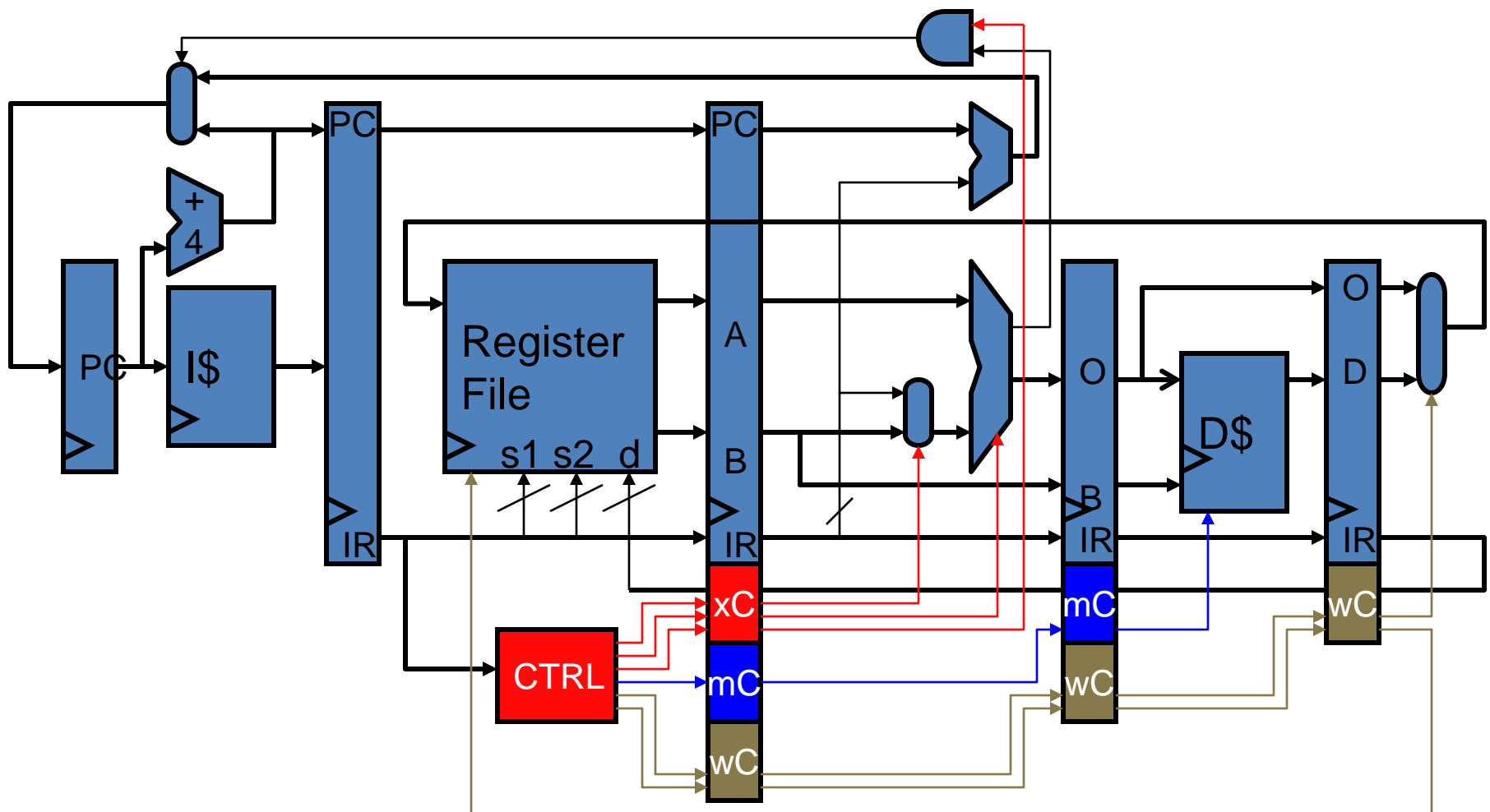
- **Pipelining:** cut datapath into N stages (here 5)
 - One insn in each stage in each cycle
 - + Clock period = $\text{MAX}(T_{\text{insn-mem}}, T_{\text{regfile}}, T_{\text{ALU}}, T_{\text{data-mem}})$
 - + Base CPI = 1: insn enters and leaves every cycle
 - Actual CPI > 1: pipeline must often stall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

5 Stage Pipelined Datapath



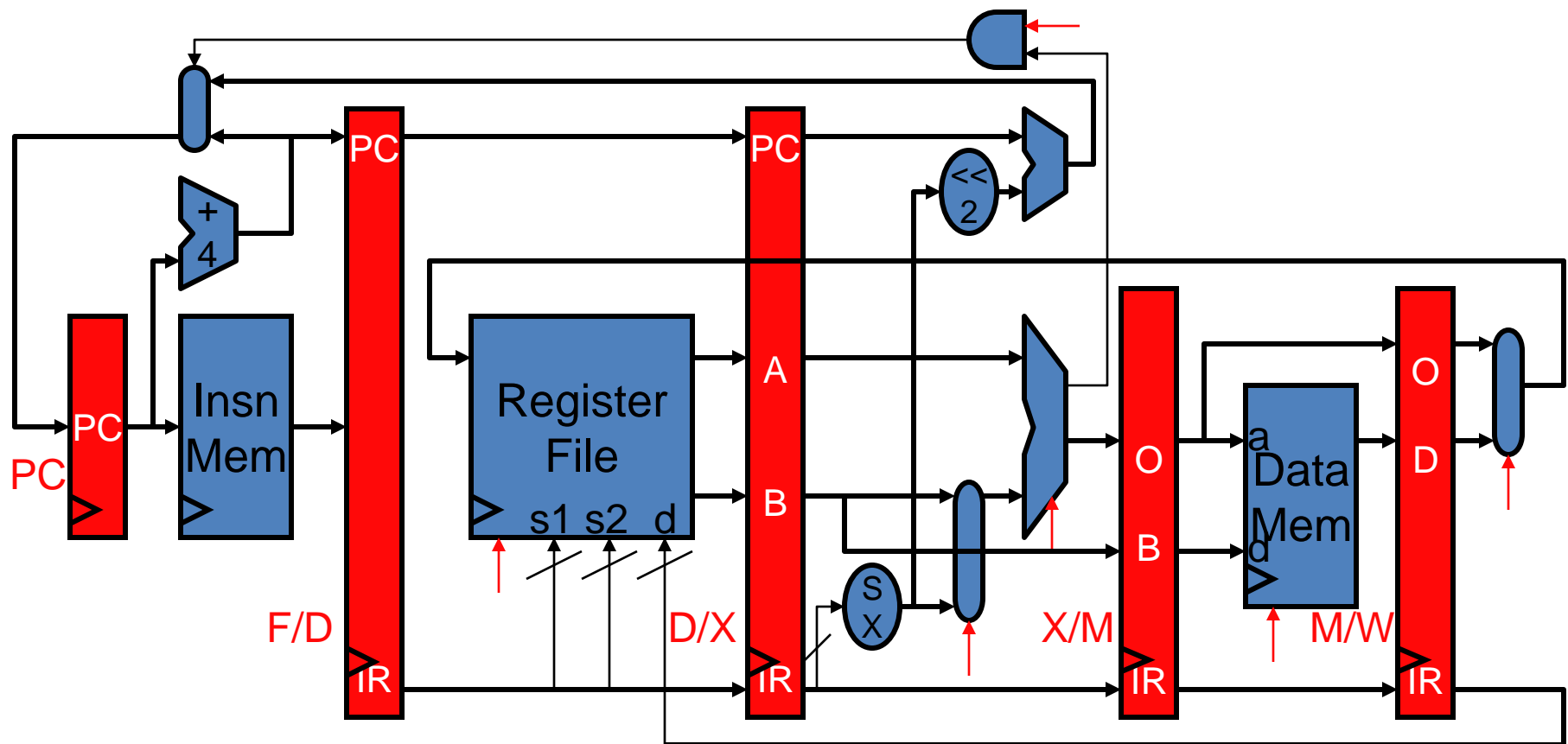
- Temporary values (PC,IR,A,B,O,D) re-latched every stage
 - Why? 5 insns may be in pipeline at once, they share a single PC?
 - Notice, PC not latched after ALU stage (why not?)

Pipeline Control



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

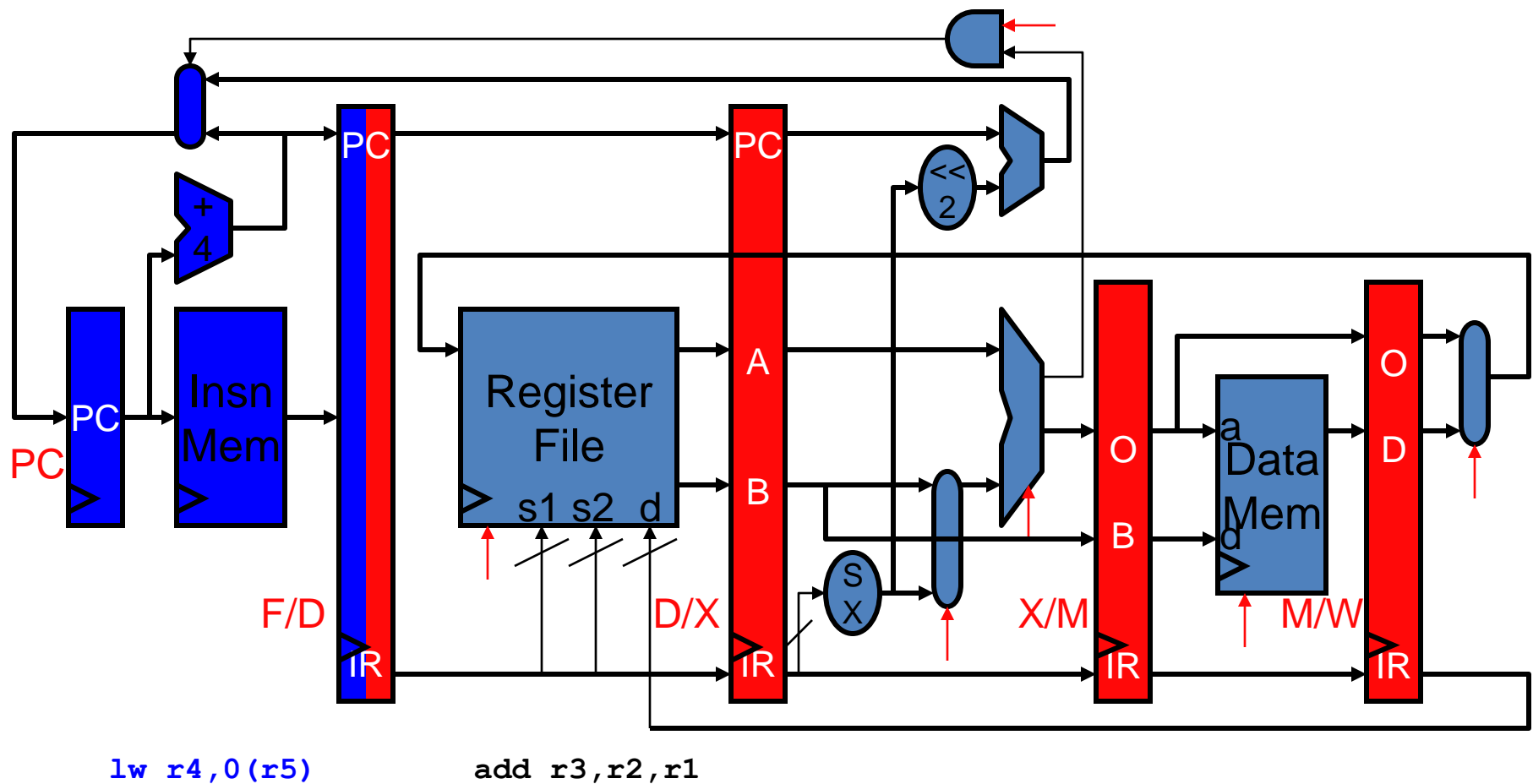
Pipeline Example: Cycle 1



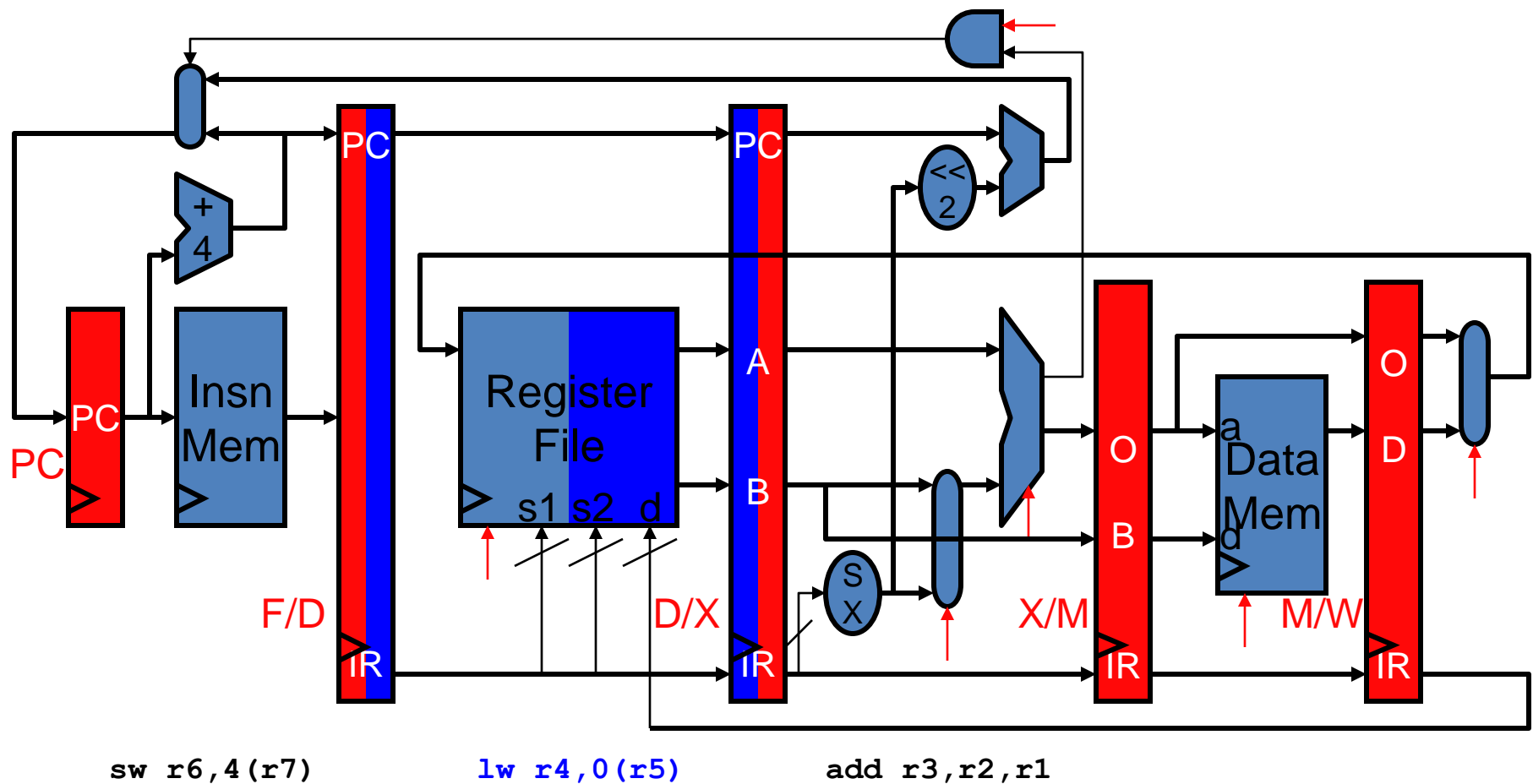
`add r3, r2, r1`

- 3 instructions

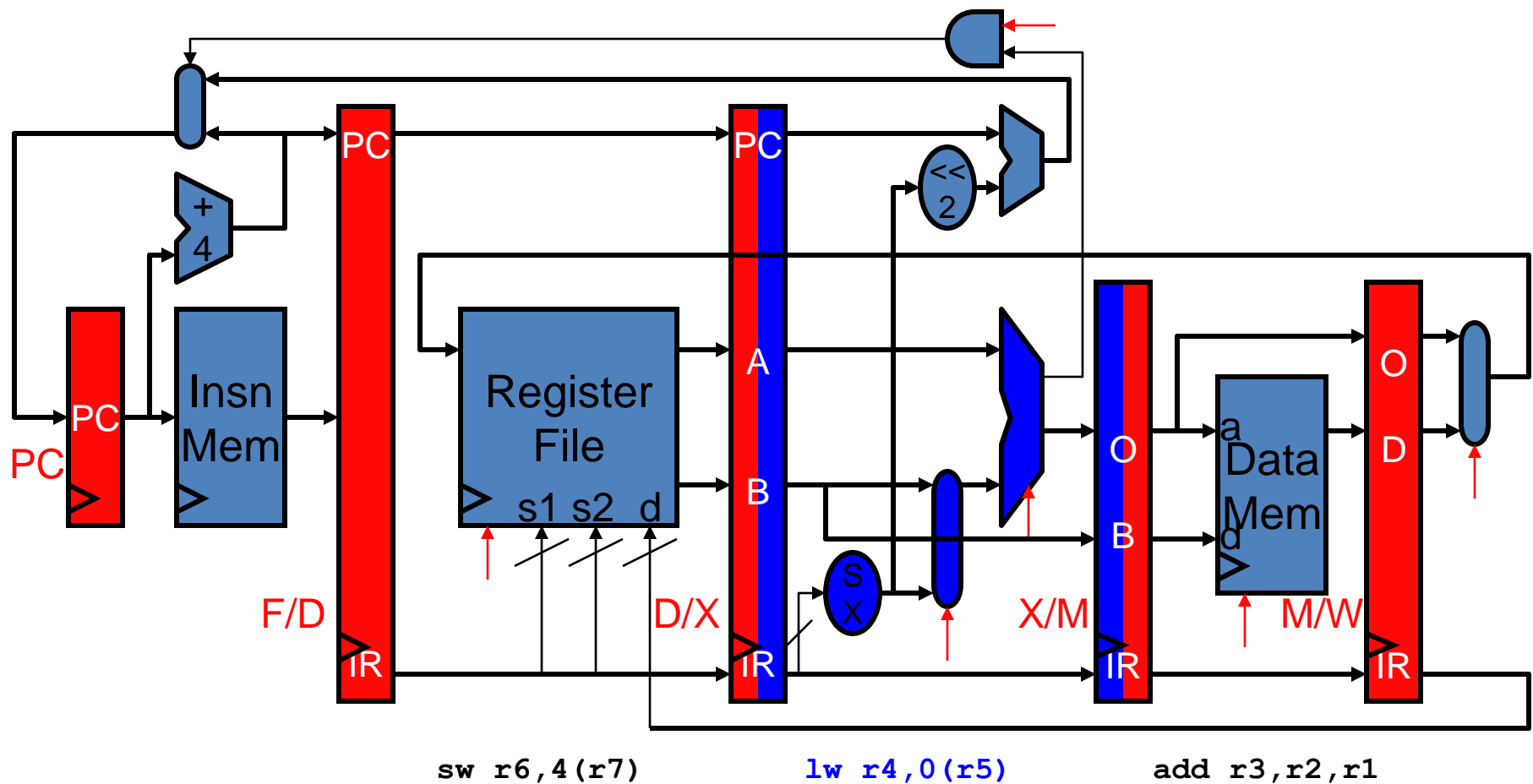
Pipeline Example: Cycle 2



Pipeline Example: Cycle 3

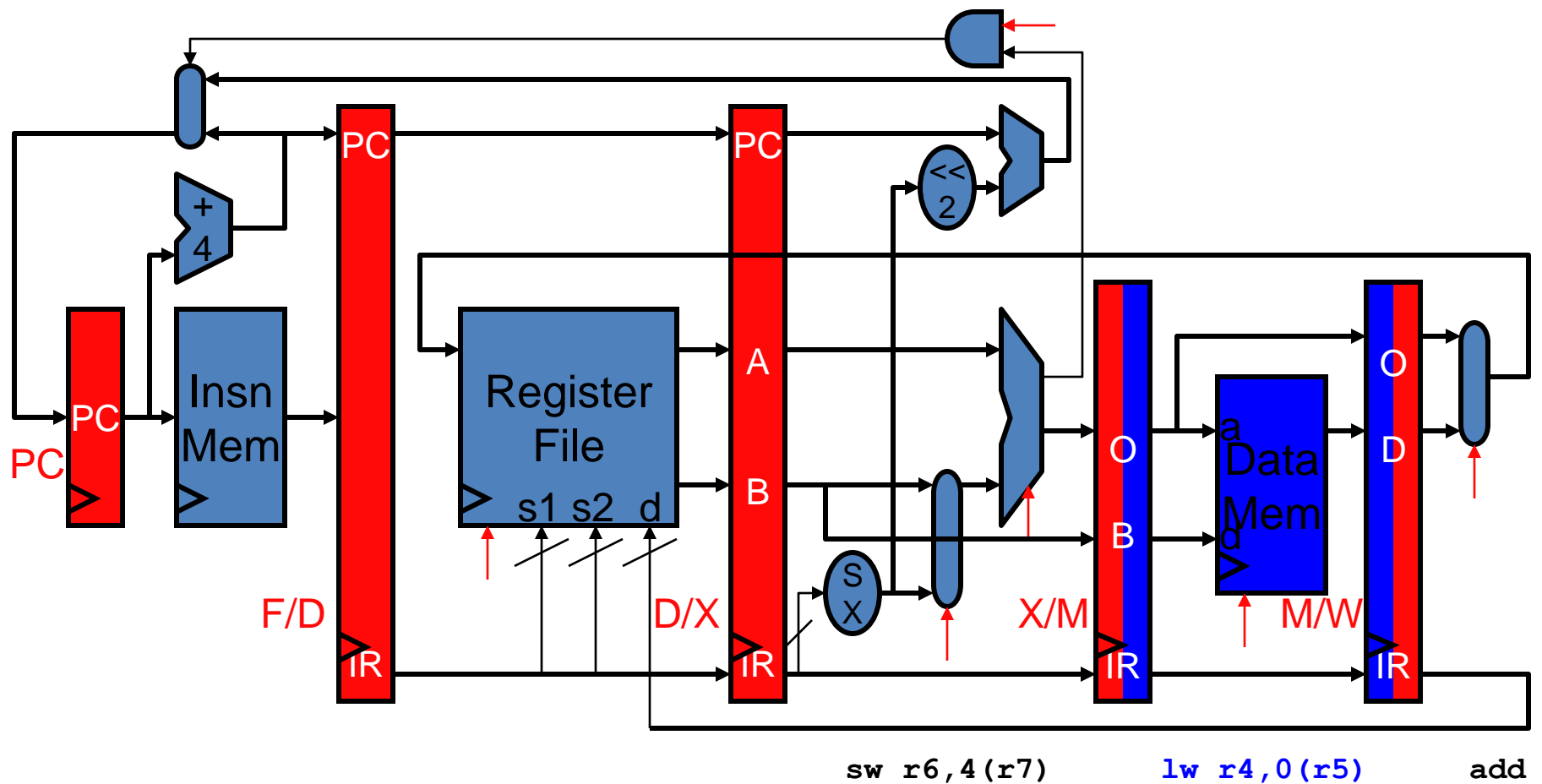


Pipeline Example: Cycle 4

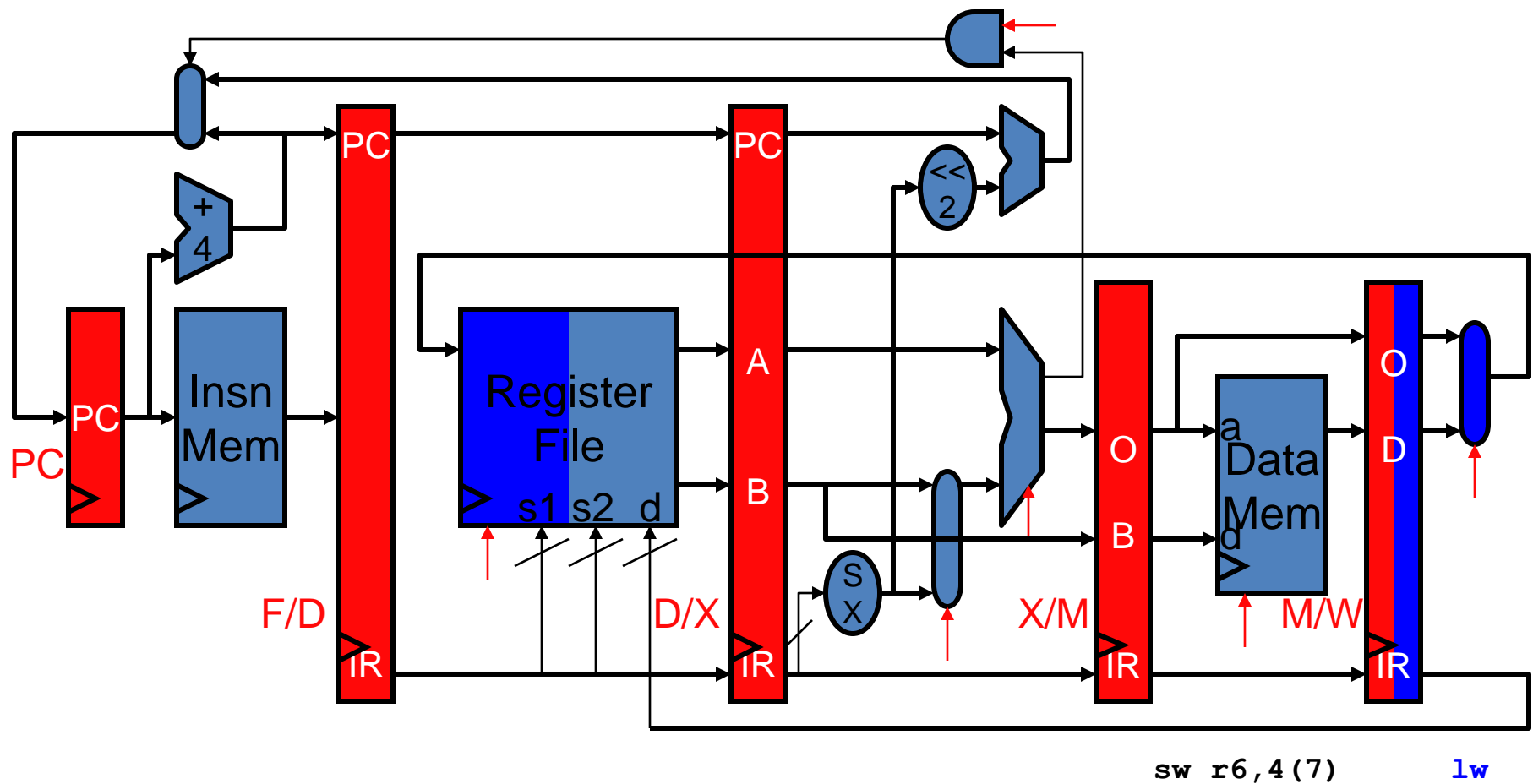


- 3 instructions

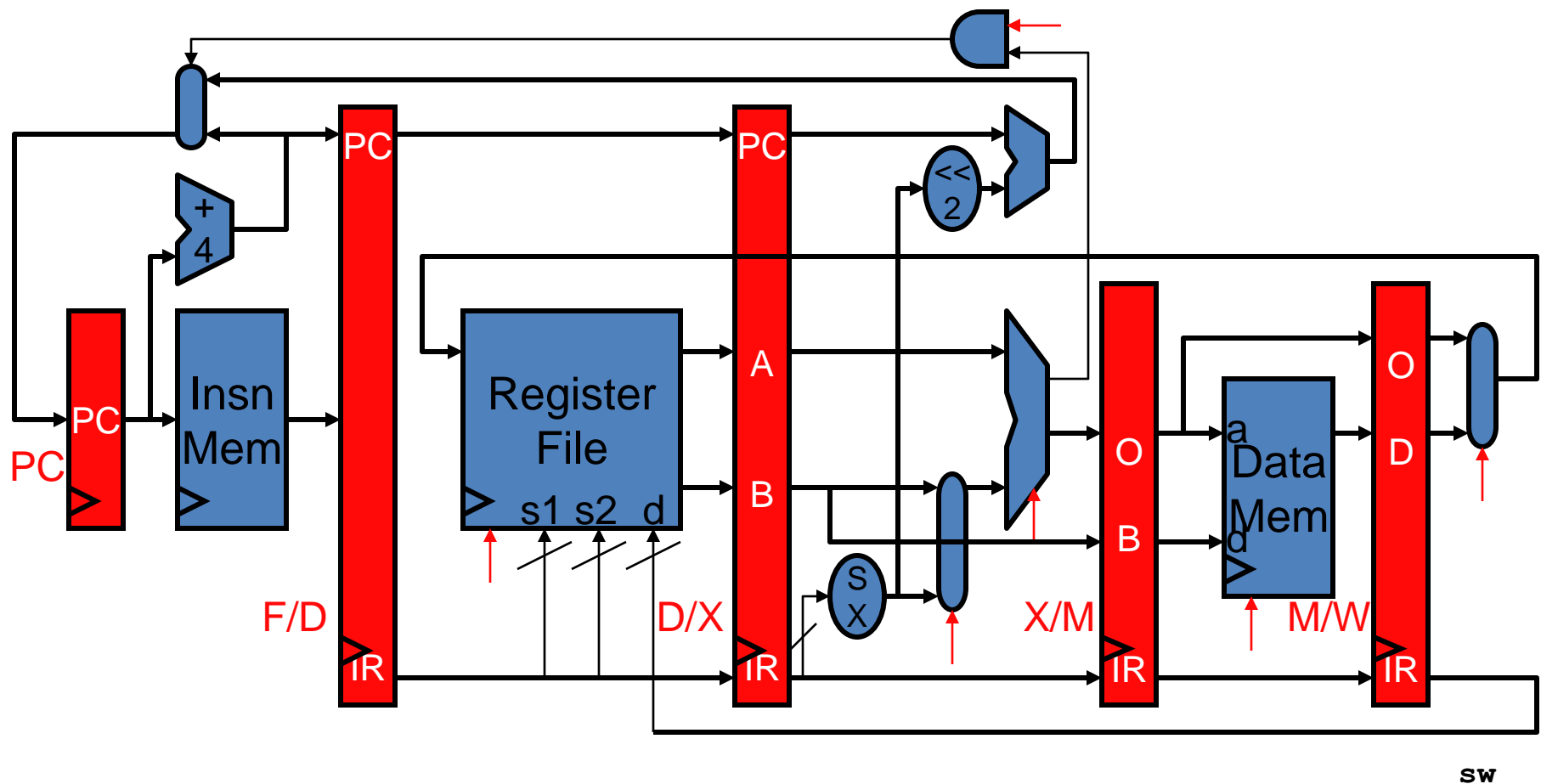
Pipeline Example: Cycle 5



Pipeline Example: Cycle 6



Pipeline Example: Cycle 7



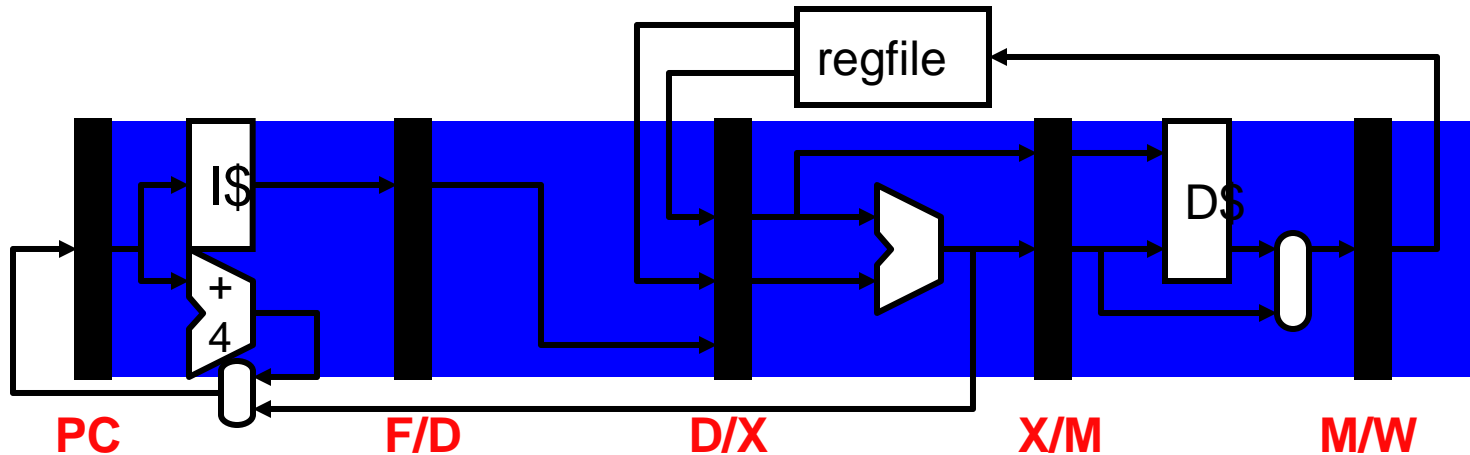
Pipeline Diagram

	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**

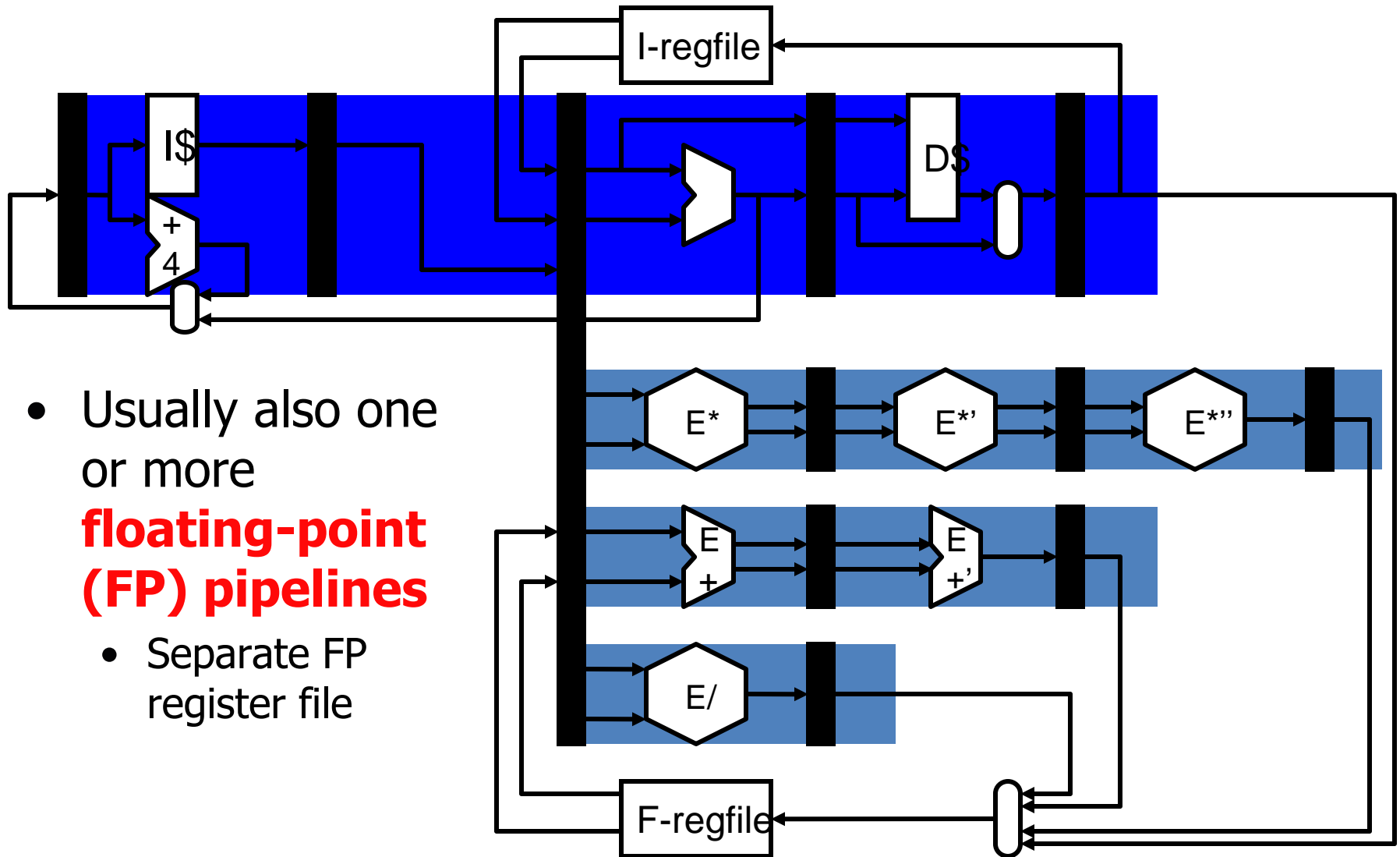
- Cycles across, insns down
- Convention: **X** means `ld r4,0(r5)` finishes execute stage and writes into X/M latch at end of cycle 4

Abstract Pipeline



- This is an **integer pipeline**
 - Execution stages are X,M,W

Floating Point Pipelines



Pipeline Performance Calculation

- Single-cycle
 - Clock period = 50ns, CPI = 1
 - Performance = 50ns/insn
- Pipelined
 - Clock period = **12ns** (50ns / 5 stages) + overheads
 - Optimistic Model:
 - CPI = **1** (each insn takes 5 cycles, but 1 completes each cycle)
 - Performance = **12ns/insn**
 - Realistic Model: (adds pipeline penalty)
 - 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 ...

- ... $> (\text{delay thru datapath}) / (\text{number of pipeline stages})$?
 - Three reasons:
 - Latches add delay
 - Pipeline stages have different delays, clock period is max delay
 - [Later:] 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)
 - $1 + \text{stall-freq}_1 * \text{stall-cyc}_1 + \text{stall-freq}_2 * \text{stall-cyc}_2 + \dots$
 - Long penalties OK if they happen rarely, e.g., $1 + 0.01 * 10 = 1.1$

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
 - Programs differ depending on data/control dependences
 - 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
 - Hazards are a bad thing: stalls reduce performance

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			
sub r1,r3,r5			F	D	X	M	W		
st r6,0(r1)				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
<code>ld r2,0(r1)</code>	F	D	X	M	W				
<code>add r1,r3,r4</code>		F	D	X	M	W			
<code>sub r1,r3,r5</code>			F	D	X	M	W		
<code>and r6,r1,r2</code>				F*	F	D	X	M	W

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

Avoiding Structural Hazards

- **Pipeline** the contended resource
 - + No IPC degradation, low area, power overheads
 - For multi-cycle resources (e.g., multiplier)
 - Doesn't help for single-cycle resources...
- **Replicate** the contended resource
 - + No IPC degradation
 - Increased area, power, latency (interconnect delay?)
 - For cheap, divisible, or highly contended resources (e.g., I\$/D\$)
- **Schedule** pipeline to reduce structural hazards (RISC)
 - Design ISA so insn uses a resource at most once
 - Eliminate same insn hazards
 - Always in same pipe stage (hazards between two of same insn)
 - Reason why integer operations forced to go through M stage
 - And always for one cycle

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 Read-after-write (RAW) True-dependence	add r2 , r3 → r1 sub r5, r4 → r2 or r6, r3 → r1 Write-after-read (WAR) Anti-dependence	add r2, r3 → r1 sub r1, r4 → r2 or r6, r3 → r1 Write-after-write (WAW) Output-dependence
--	--	--

- Only one dependence matters between any two insns (RAW has priority)
- Dependence is property of the program and ISA
- **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

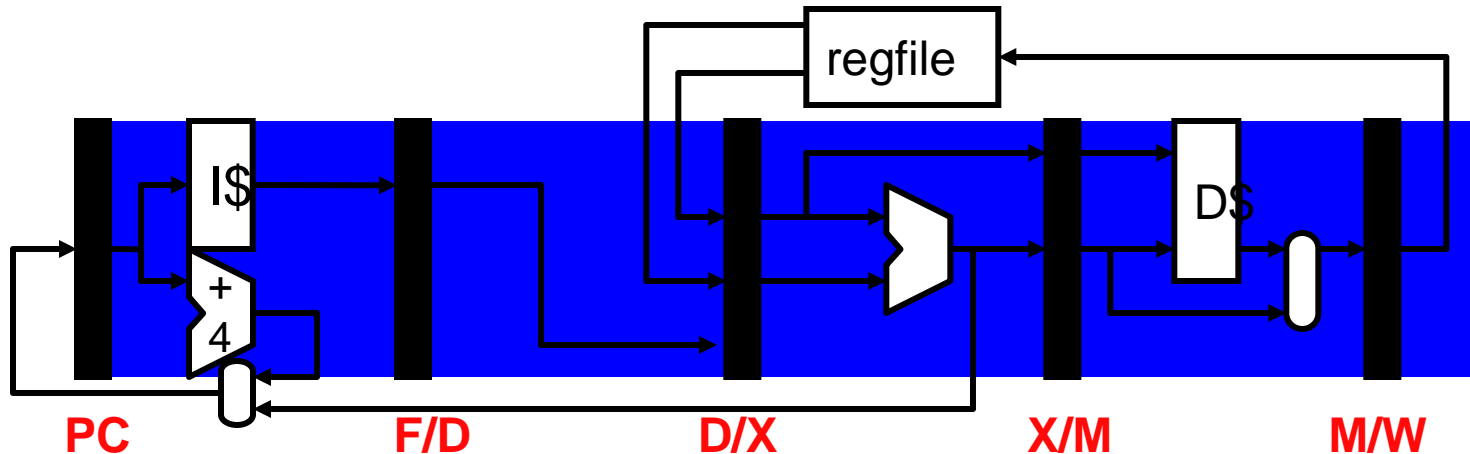
RAW

- **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
 - $(F/D.rs1 \ \& \ (F/D.rs1 == D/X.rd \mid F/D.rs1 == X/M.rd \mid F/D.rs1 == M/W.rd)) \mid$
 $(F/D.rs2 \ \& \ (F/D.rs2 == D/X.rd \mid F/D.rs2 == X/M.rd \mid F/D.rs2 == M/W.rd))$
 - Re-evaluated every cycle until no longer true
 - + Low cost, simple
 - IPC degradation, dependences are the common case

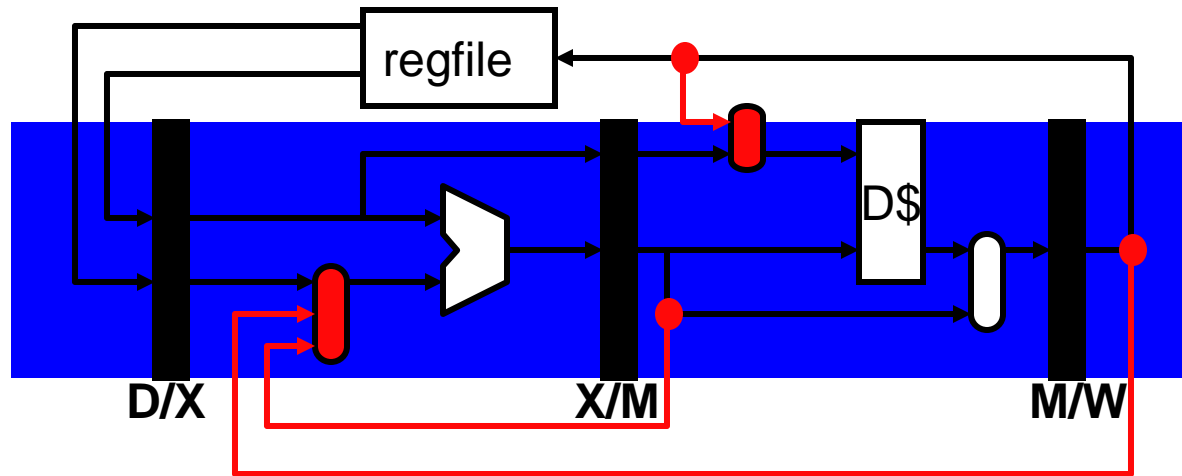
Stall Timing

- Stall Types:
 - data stall,
 - propagated stall
- D and W stages share regfile

	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			F*	F*	F	D	X	M	W	

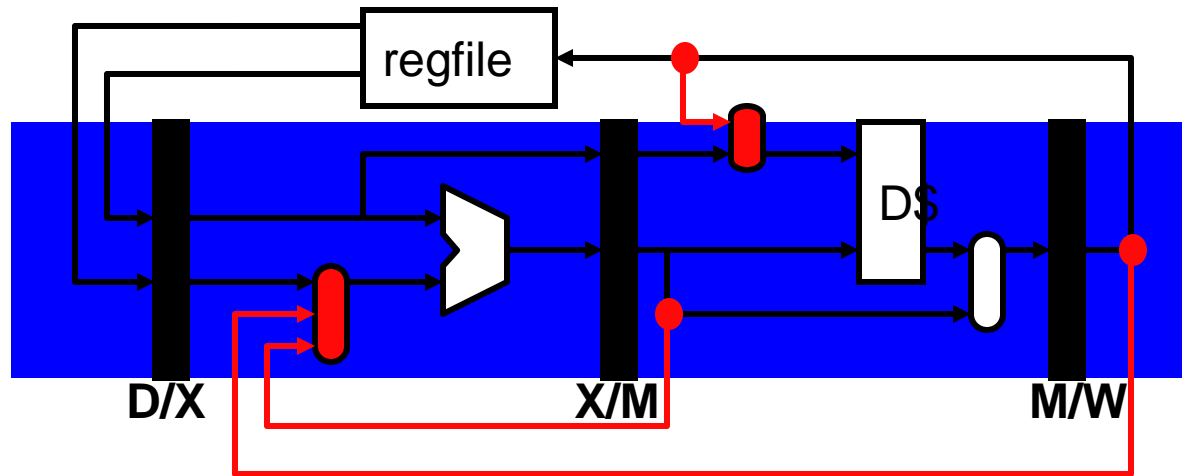
(assumes RF bypassing: 1st half W writes, 2nd half D reads 2 cycle stall. Also, see backup slides for more on this.)

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
 - **X → X**: from beginning of M (X output) to input of X
 - **M → X**: from beginning of W (M output) to input of X
 - **M → M**: from beginning of W (M output) to data input of M
 - **"full bypassing"**:
 - Two each of X → X, M → X (figure shows 1) + M → M =
 - + Reduces stalls in a big way
 - Additional wires and muxes may increase clock cycle

Bypass Logic

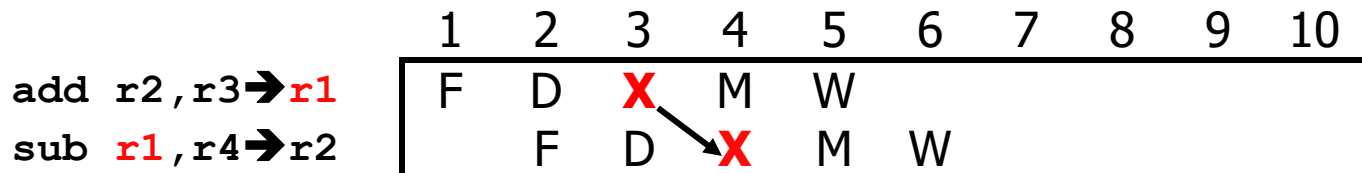


- Bypass logic: similar to but separate from stall logic
 - Stall logic controls latches, bypass logic controls mux inputs
 - Complement one another: can't bypass → must stall
 - ALU input mux bypass logic
 - $(D/X.rs2 \ \& \ X/M.rd == D/X.rs2) \rightarrow 2$ // check first
 - $(D/X.rs2 \ \& \ M/W.rd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

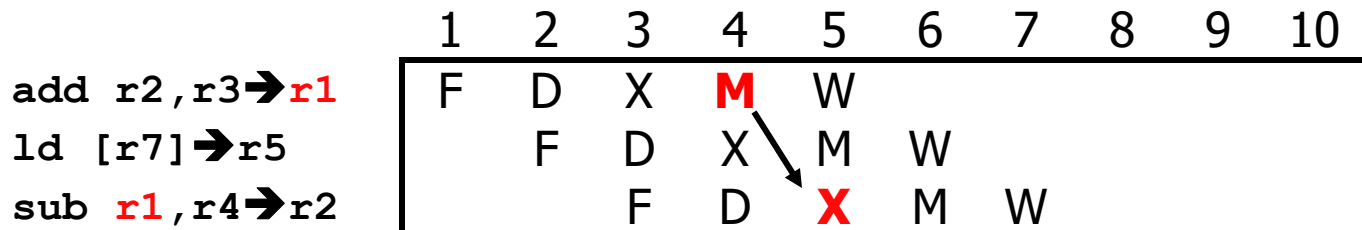
Pipeline Diagrams with Bypassing

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

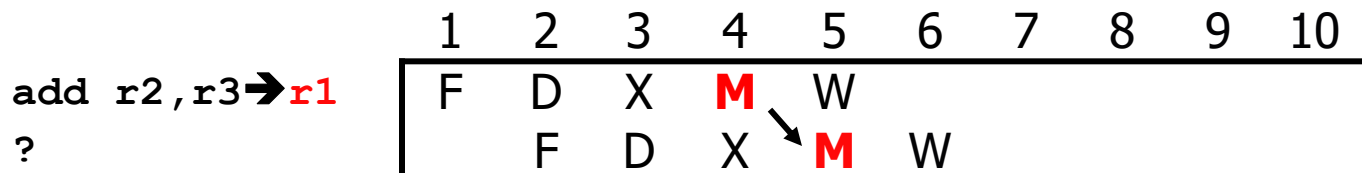
- Example: full bypassing, use X \rightarrow X bypass



- Example: full bypassing, use M \rightarrow X bypass

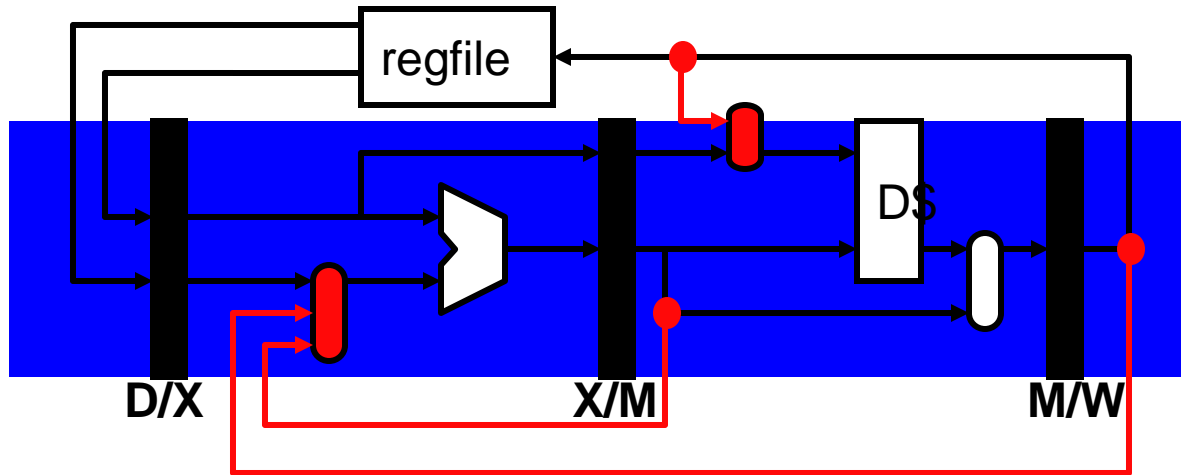


- Example: M \rightarrow M bypass



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

Bypass Logic



- Does $M \rightarrow M$ bypassing make sense?

- Not to the address input (why not?)

`st r6, 4(r1) add r2, r3 → r1`

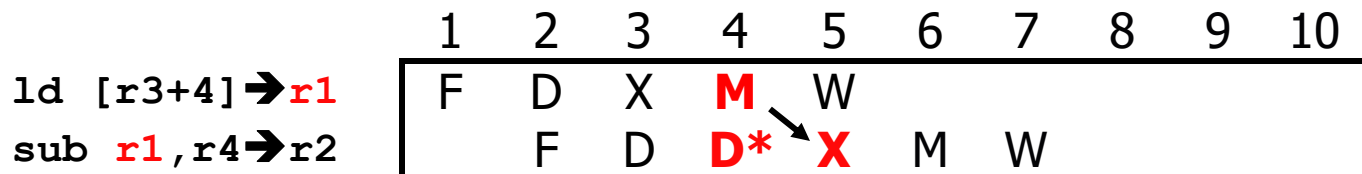
X

- But to the store data input, yes

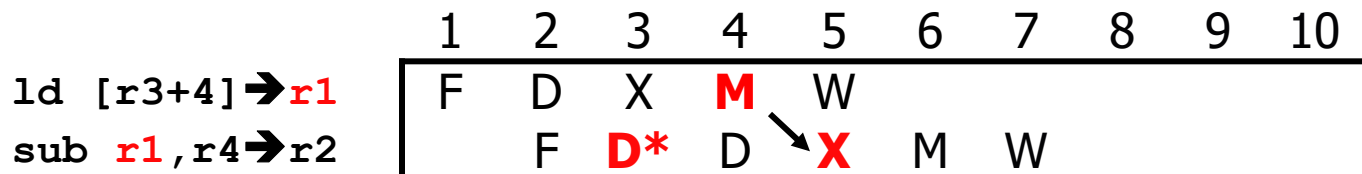
`st r1, 4(r6) add r2, r3 → r1`

Load-Use Stalls

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



- Aside: with M \rightarrow X bypassing, stall logic can be in D or X



Reducing Load-Use Stall Frequency

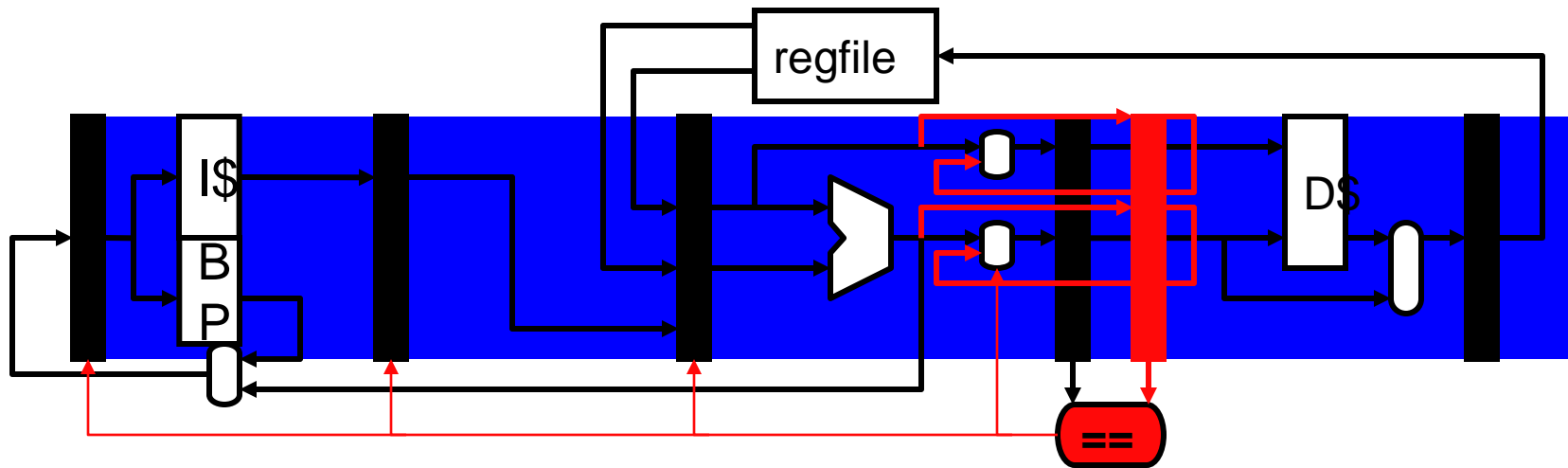
	1	2	3	4	5	6	7	8	9
add r2,r1→r3	F	D	X	M	W				
lw 4(r3)→r4		F	D	X	M	W			
addi r4,1→r6			F	D*	D	X	M	W	
sub r3,r1→r8					F	D	X	M	W

- Use compiler scheduling to reduce load-use stall frequency
 - More on compiler scheduling later

	1	2	3	4	5	6	7	8	9
add r2,r1→r3	F	D	X	M	W				
lw 4(r3)→r4		F	D	X	M	W			
sub r3,r1→r8			F	D	X	M	W		
addi r4,1→r6				F	D	X	M	W	

Hidden Bonus Slides

Research: Razor

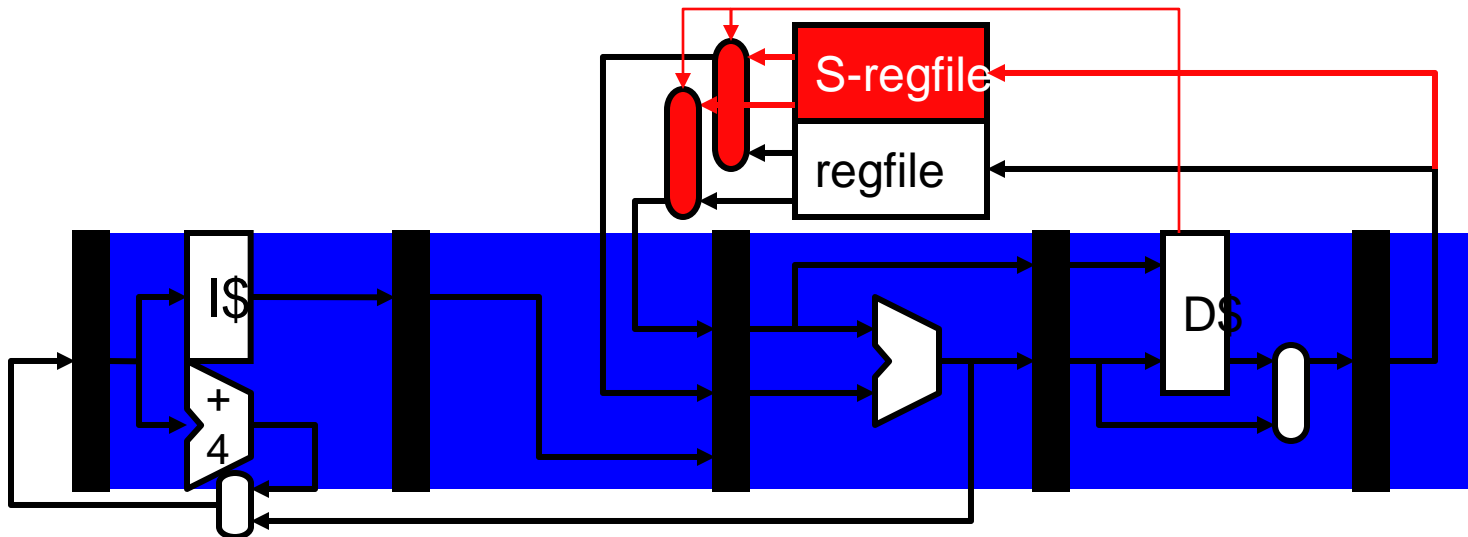


- **Razor** [Uht, Ernst+]
 - Identify pipeline stages with narrow signal margins (e.g., **X**)
 - Add "**Razor**" X/M latch: relatches X/M input signals after safe delay
 - Compare X/M latch with "safe" razor X/M latch, different?
 - Flush F,D,X & M
 - Restart M using X/M razor latch, restart F using D/X latch
 - + Pipeline will not "break" → reduce V_{DD} until flush rate too high
 - + Alternatively: "over-clock" until flush rate too high

Research: Speculation Gating

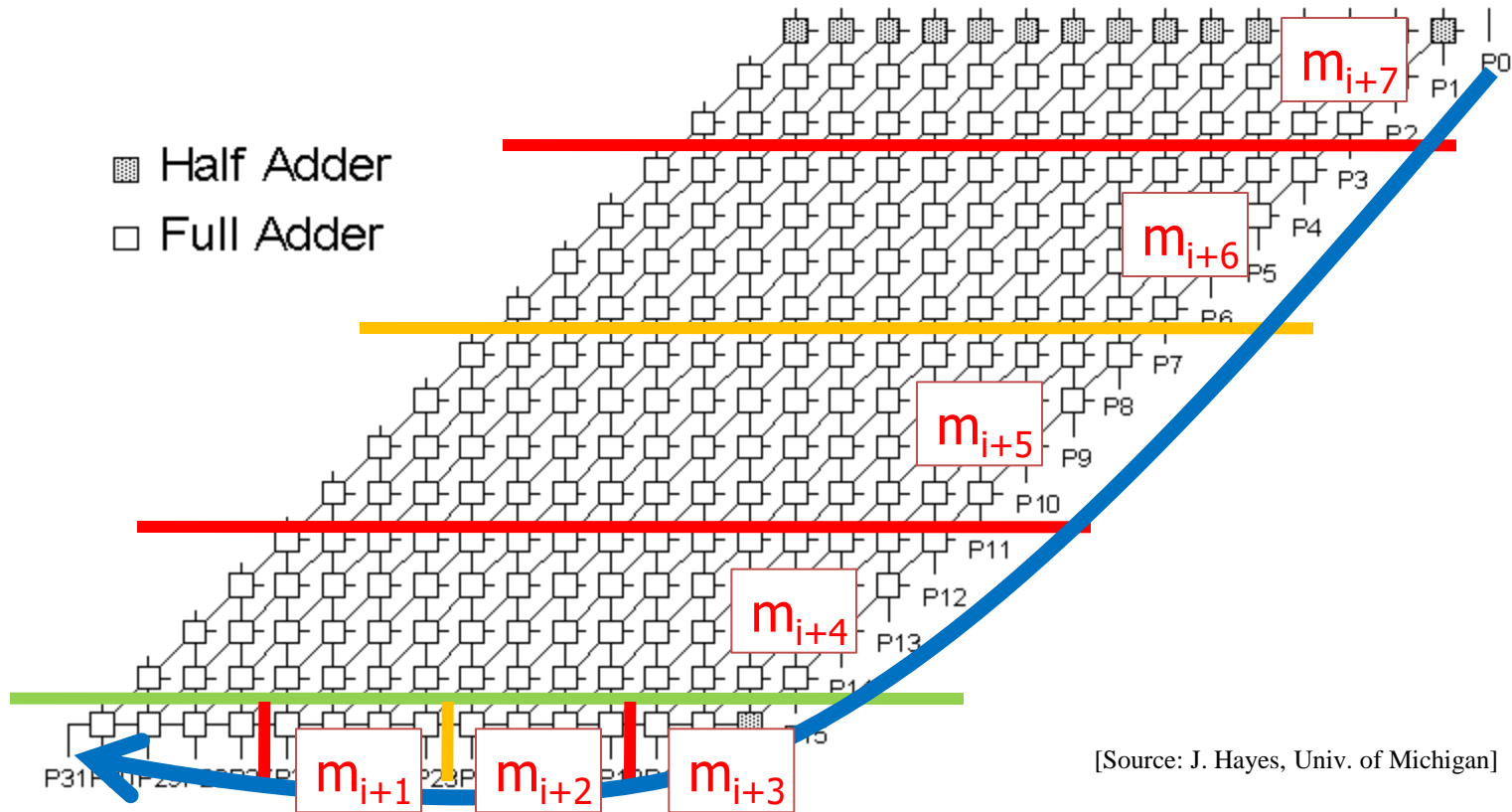
- **Speculation gating** [Manne+]
 - Extend branch predictor to give prediction + **confidence**
 - Speculate on high-confidence (mis-prediction unlikely) branches
 - Stall (save energy) on low-confidence branches
- **Confidence estimation**
 - What kind of hardware circuit estimates confidence?
 - Hard in absolute sense, but easy relative to given threshold
 - Counter-scheme similar to %_{miss} threshold for cache resizing
 - Example: assume 90% accuracy is high confidence
 - PC-indexed table of confidence-estimation counters
 - Correct prediction? $\text{table}[\text{PC}] += 1$: $\text{table}[\text{PC}] -= 9$;
 - Prediction for PC is confident if $\text{table}[\text{PC}] > 0$;

Research: Runahead Execution



- In-order writebacks essentially imply stalls on D\$ misses
 - Can save power ... or use idle time for performance
- **Runahead execution** [Dundas+ 97]
 - Shadow regfile kept in sync with main regfile (write to both)
 - D\$ miss: continue executing using shadow regfile (disable stores)
 - D\$ miss returns: flush pipe and restart with stalled PC
 - + Acts like a smart prefetch engine
 - + Performs better as cache t_{miss} grows (relative to clock period)

Example: Integer Multiplier



[Source: J. Hayes, Univ. of Michigan]

16x16 combinational multiplier

Dependences and Loops

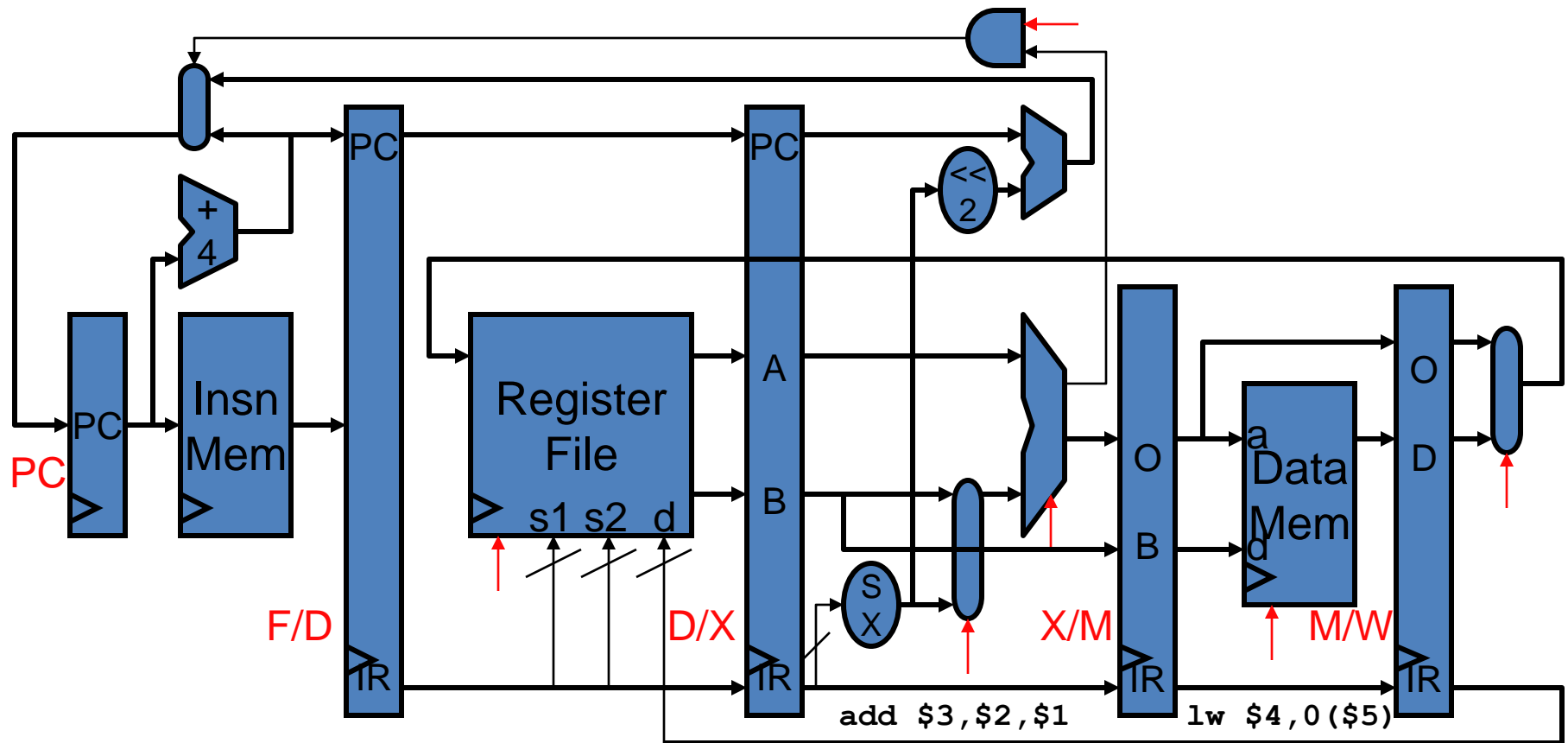
- Data dependences in loops
 - **Intra-loop**: within same iteration
 - **Inter-loop**: across iterations
 - Example: DAXPY (**D**ouble precision **A X Plus Y**)

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

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

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

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** follows **lw**

Simple Analytical Pipeline Model

- Let: insn execution require **N** stages, each takes t_n time
- Single-cycle execution
 - 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?

N-stages $\neq \infty$ due to 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 $\max(t_n) = L_1/N$
 - $L_{1+P+O} = L_1 + N*O$
 - $T_{+P+O} = 1/(L_1/N + O) = 1/(1/T + O) \leq T, \leq T/O$
 - $L_{M+P+O} = M*L_1/N + M*O = L_{M+P} + M*O$
 - $S_{+P+O} = [M*L_1 / (M*L_1/N + M*O)] = \leq N = S_{+P}, \leq L_1/O$
- O limits throughput and speedup \rightarrow useful N

N-stages != due to Hazards

- **Dependence**: relationship that serializes two insns
 - **Data**: two insns use the same value or storage location
 - **Control**: one instruction affects whether another executes at all
 - **Maybe**: two insns *may* have a dependence
- **Hazard**: dependence causes potential incorrect execution
 - Possibility of using or corrupting data or execution flow
 - **Structural**: two insns want to use same structure, one must wait
 - 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 more dependences become hazards)
 - Exact H depends on program, requires detailed simulation/model

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:
 - “`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 r3,r2,r1 //stall
st r1,0(sp)
ld r5,16(sp)
ld r6,20(sp)
sub r5,r6,r4 //stall
st r4,12(sp)
```


After

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

After

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

Wrong!

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

Compiler Scheduling Requires

- **Alias analysis**

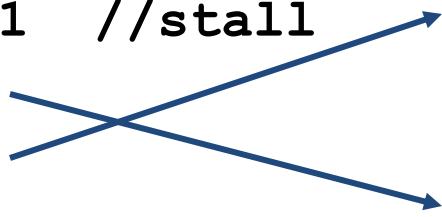
- Ability to tell whether load/store reference same memory locations
 - Effectively, whether load/store can be rearranged
- Example code: easy, all loads/stores use same base register (**sp**)
- New example: can compiler tell that **r8 = sp**?

Before

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

Wrong(?)

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



- 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

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	