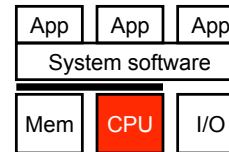# CIS 501
Computer Architecture

Unit 6: Pipelining

Slides originally developed by Amir Roth with contributions by Milo Martin at University of Pennsylvania with sources that included University of Wisconsin slides by Mark Hill, Guri Sohi, Jim Smith, and David Wood.

This Unit: (Scalar In-Order) Pipelining

App	App	App
System software		

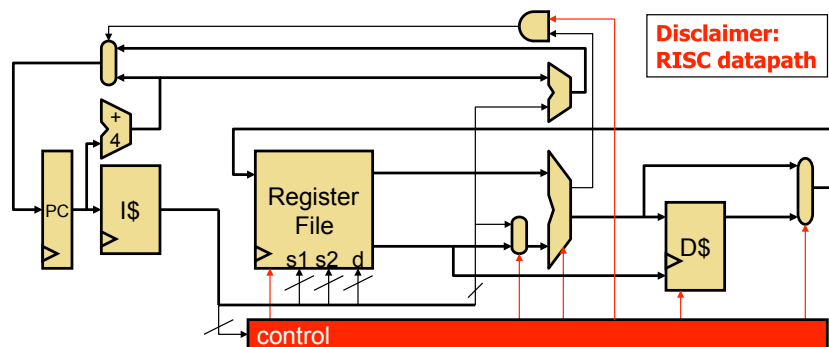| Mem	CPU	I/O

- Principles of pipelining
  - Effects of overhead and hazards
  - Pipeline diagrams
- Data hazards
  - Stalling and bypassing
- Control hazards
  - Branch prediction
  - Predication


Readings

- H&P
  - Appendix A

Datapath Background

Datapath and Control


Disclaimer: RISC datapath

- **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
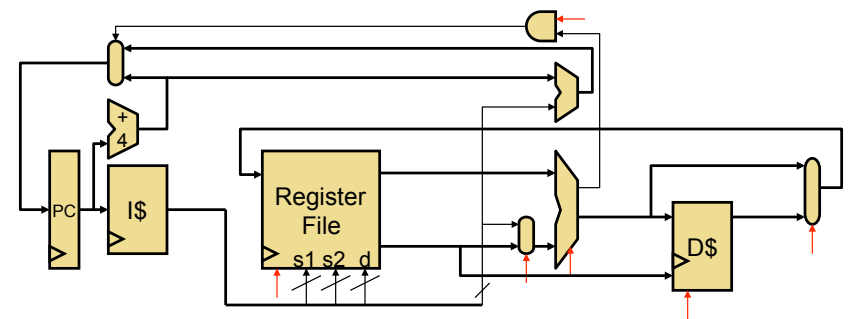
- **Single-cycle datapath**: true "atomic" fetch/execute loop
  - Fetch, decode, execute one complete instruction every cycle
  - **"Hardwired control"**: opcode to control signals ROM
  + Low CPI: 1 by definition
  – Long clock period: to accommodate slowest instruction
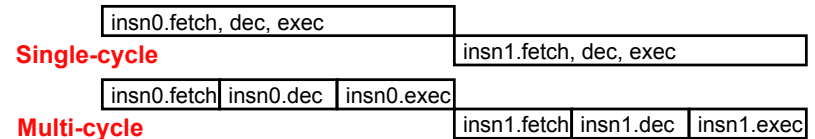
Multi-Cycle Datapath
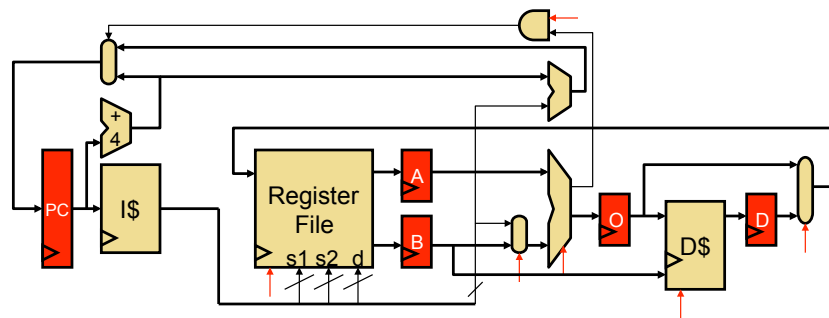
- **Multi-cycle datapath**: attacks slow clock
  - Fetch, decode, execute one complete insn over multiple cycles
  - **Micro-coded control**: "stages" control signals
  - **Allows insns to take different number of cycles** (main point)
  ± Opposite of single-cycle: short clock period, high CPI (think: CISC)

Single-cycle vs. Multi-cycle Performance

- Single-cycle
  - Clock period = 50ns, CPI = 1
  - Performance = **50ns/insn**

- Multi-cycle has opposite performance split of single-cycle
  + Shorter clock period
  – Higher CPI

- Multi-cycle
  - Branch: 20% (**3** cycles), load: 20% (**5** cycles), ALU: 60% (**4** cycles)
  - Clock period = **11ns**, CPI = (20%*3)+(20%*5)+(60%*4) = 4
    - Why is clock period 11ns and not 10ns?
  - Performance = **44ns/insn**
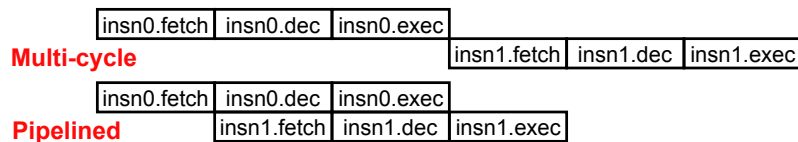
- Aside: CISC makes perfect sense in multi-cycle datapath

Pipelining Basics

Latency versus Throughput

Single-cycle	insn0.fetch, dec, exec	
		insn1.fetch, dec, exec

|			
Multi-cycle	insn0.fetch	insn0.dec	insn0.exec
		insn1.fetch	insn1.dec	insn1.exec

- Can we have both low CPI and short clock period?
  - Not if datapath executes only one insn at a time

- Latency vs. Throughput
  - Latency: no good way to make a single insn go faster
  - **Throughput**: fortunately, no one cares about single insn latency
    - Goal is to make programs, not individual insns, go faster
    - Programs contain billions of insns
  - Key: **exploit inter-insn parallelism**

Pipelining

Multi-cycle	insn0.fetch	insn0.dec	insn0.exec
		insn1.fetch	insn1.dec	insn1.exec

|			
Pipelined	insn0.fetch	insn0.dec	insn0.exec
		insn1.fetch	insn1.dec	insn1.exec

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

Five Stage Pipelined Datapath
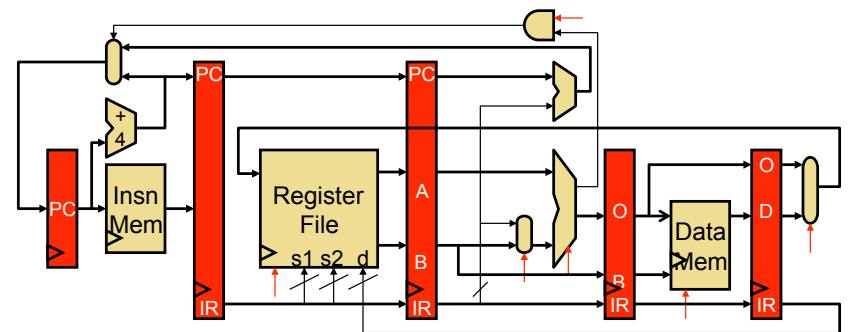
- Temporary values (PC,IR,A,B,O,D) re-latched every stage
  - Why? 5 insns may be in pipeline at once with different PCs
  - Notice, PC not latched after ALU stage (not needed later)
  - **Pipelined control**: one single-cycle controller
    - Control signals themselves pipelined

Five Stage Pipeline Performance

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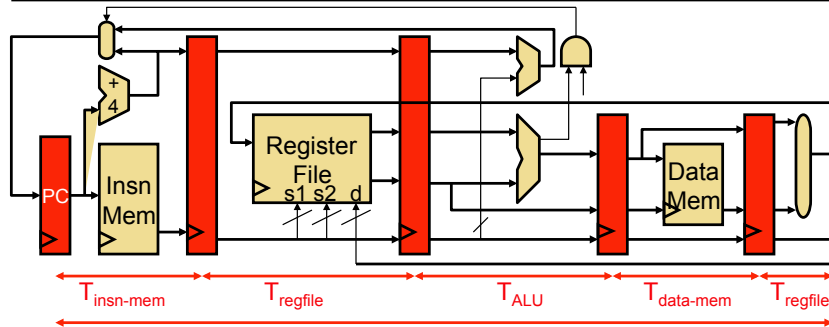
- **Pipelining**: cut datapath into N stages (here five) $T_{singlecycle}$
  - One insn in each stage in each cycle
  - + Clock period = MAX($T_{insn-mem}$, $T_{regfile}$, T_{ALU}, $T_{data-mem}$)
  - + Base CPI = 1: insn enters and leaves every cycle
  - − Actual CPI > 1: pipeline must often stall
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

Pipeline Terminology
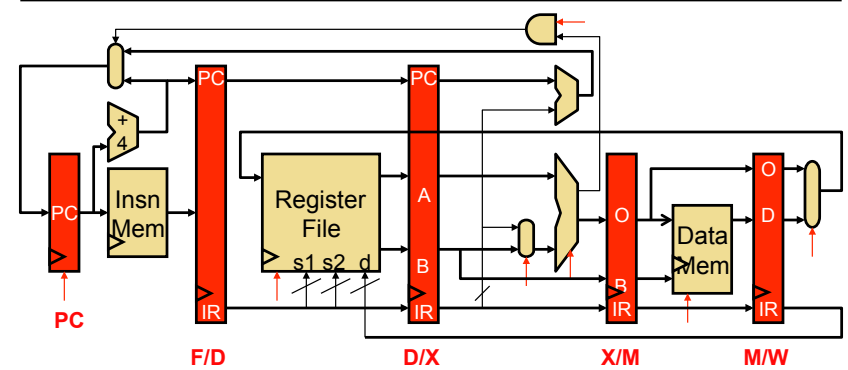
PC   **F/D**   **D/X**   **X/M**   **M/W**

- Five stage: **F**etch, **D**ecode, e**X**ecute, **M**emory, **W**riteback
  - Nothing magical about 5 stages (Pentium 4 had 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
  - Alternative: "superscalar" (later)

- **In-order pipeline**: insns enter execute stage in order
  - Alternative: "out-of-order" (later)

- **Pipeline depth**: number of pipeline stages
  - Nothing magical about five
  - Trend has been to deeper pipelines (again, more later)

Instruction Convention

- Different ISAs use inconsistent register orders

- Some ISAs (for example MIPS)
  - Instruction destination (i.e., output) **on the left**
  - add $1, $2, $3 means $1 ← $2+$3

- Other ISAs
  - Instruction destination (i.e., output) **on the right**
  - `add r1,r2,r3` means `r1+r2→r3`
  - `ld 0(r5),r4` means `mem[r5+8]→r4`
  - `st r4,0(r5)` means `r4→mem[r5+8]`

- Will try to specify to avoid confusion, next slides MIPS style
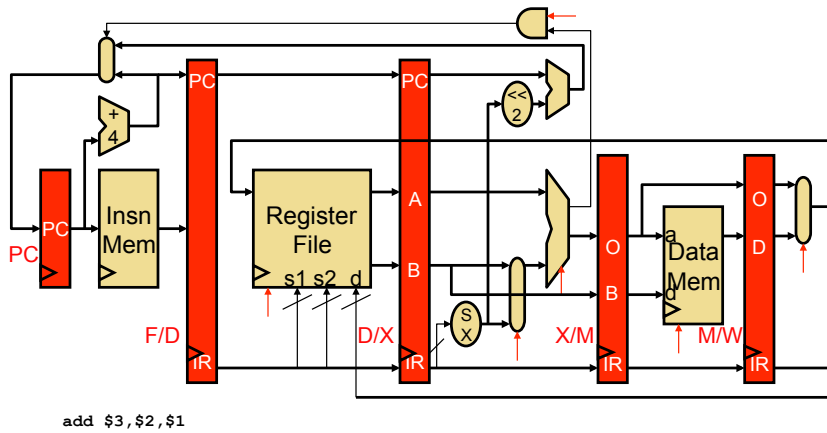
Pipeline Example: Cycle 1

add $3,$2,$1

- 3 instructions

Pipeline Example: Cycle 2

lw $4,0($5)   add $3,$2,$1

Pipeline Example: Cycle 3

sw $6,4($7)   lw $4,0($5)   add $3,$2,$1

Pipeline Example: Cycle 4

sw $6,4($7)   lw $4,0($5)   add $3,$2,$1

- 3 instructions

Pipeline Example: Cycle 5

sw $6,4($7)   lw $4,0($5)   add
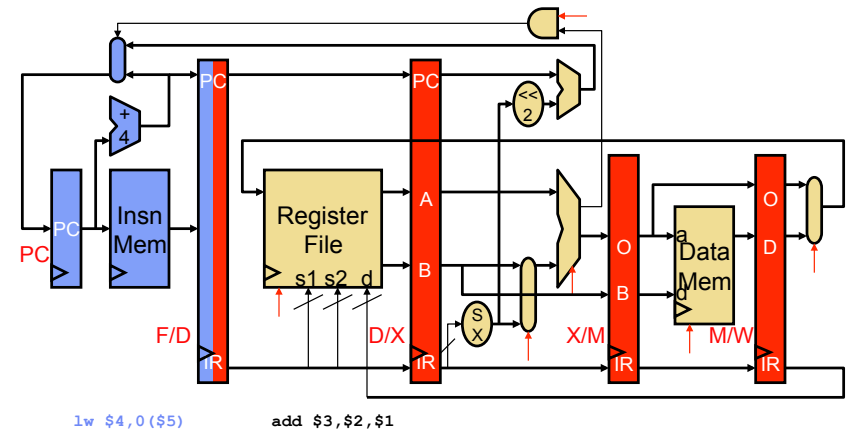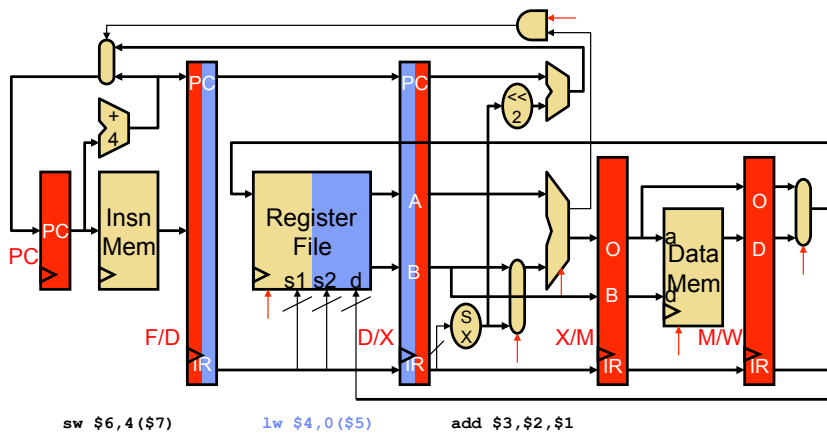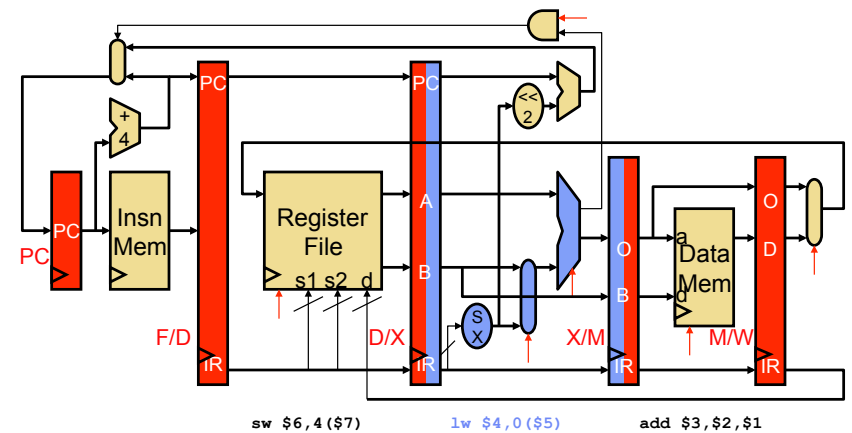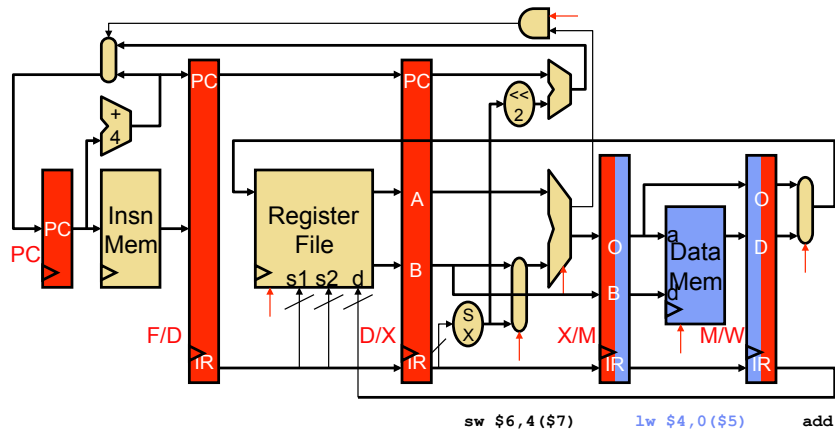
Pipeline Example: Cycle 6

sw $6,4(7)   lw
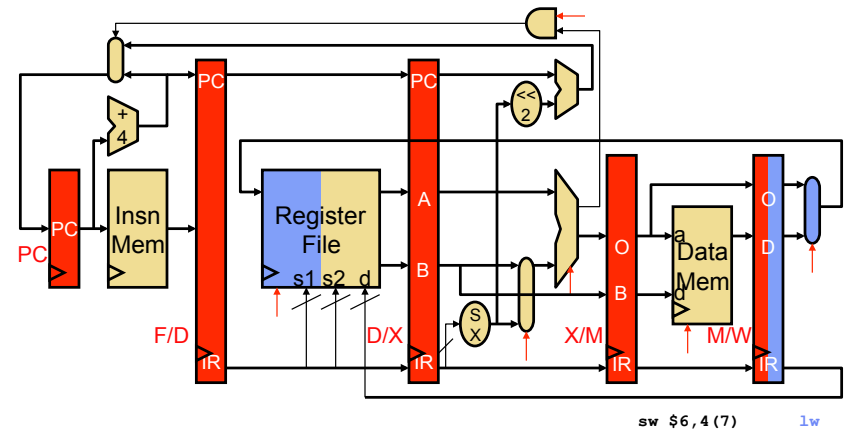
Pipeline Example: Cycle 7

sw
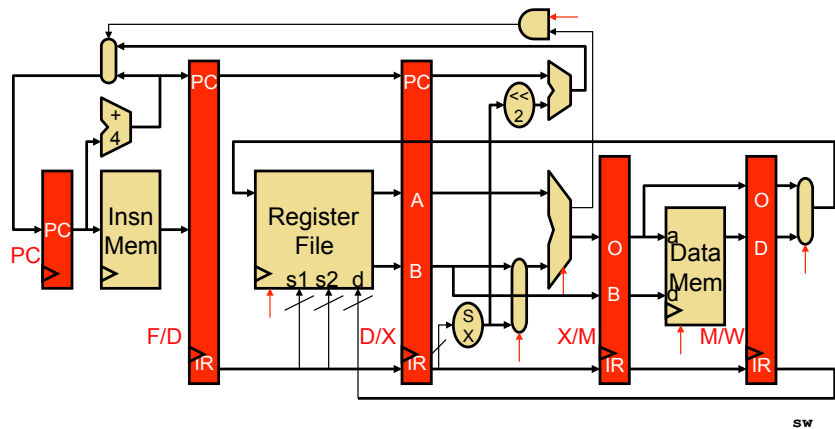
Pipeline Diagram

- **Pipeline diagram**: shorthand for what we just saw
  - Across: cycles
  - Down: insns
  - Convention: **X** means `lw $4,0($5)` finishes execute stage and writes into X/M latch at end of cycle 4

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

Example Pipeline Perf. Calculation

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

Q1: Why Is Pipeline Clock Period …

- … > (delay thru datapath) / (number of pipeline stages)?

  - A few reasons:
    - Latches add delay
    - Extra "bypassing" logic adds delay
    - Pipeline stages have different delays, clock period is max delay

  - 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)
  - $1 + \text{stall-freq}_1 * \text{stall-cyc}_1 + \text{stall-freq}_2 * \text{stall-cyc}_2 + …$

- Correctness/performance/make common case fast (MCCF)
  - Long penalties OK if they happen rarely, e.g., 1 + 0.01 * 10 = 1.1
  - Stalls also have implications for ideal number of pipeline stages

Data Dependences, Pipeline Hazards, and Bypassing

Dependences and Hazards

- **Dependence**: relationship between two insns
  - **Data**: two insns use same storage location
  - **Control**: one insn affects whether another executes at all
  - Not a bad thing, programs would be boring without them
  - Enforced by making older insn go before younger one
    - Happens naturally in single-/multi-cycle designs
    - But not in a pipeline

- **Hazard**: dependence & possibility of wrong insn order
  - Effects of wrong insn order cannot be externally visible
    - **Stall**: for order by keeping younger insn in same stage
  - Hazards are a bad thing: stalls reduce performance

Why Does Every Insn Take 5 Cycles?
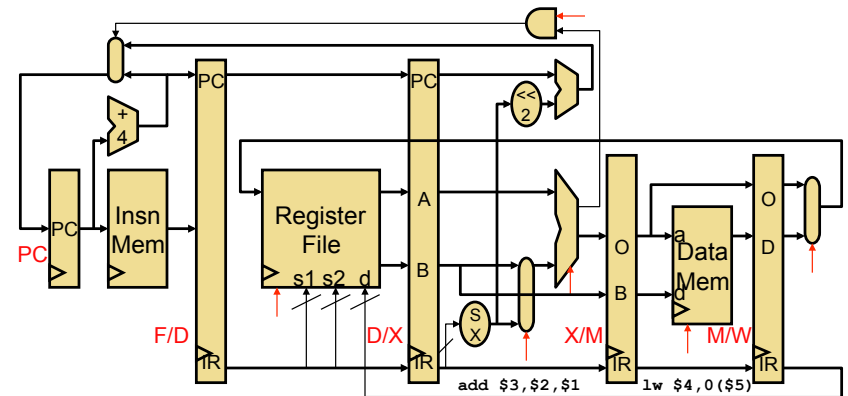
- Could/should we allow **add** to skip M and go to W? No
  - It wouldn't help: peak fetch still only 1 insn per cycle
  - **Structural hazards**: imagine **add** follows **lw**

Structural Hazards

- **Structural hazards**
  - Two insns trying to use same circuit at same time
    - E.g., structural hazard on register file write port

- **To fix structural hazards**: proper ISA/pipeline design
  - Each insn uses every structure exactly once
  - For at most one cycle
  - Always at same stage relative to F (fetch)

- **Tolerate structure hazards**
  - Add stall logic to stall pipeline when hazards occur

Example Structural Hazard

	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: unified instruction & data memories (caches)
  - Solutions:
    - Separate instruction/data memories (caches)
    - Redesign cache to allow 2 accesses per cycle (slow, expensive)
    - Stall pipeline

Data Hazards

sw $6,0($7)   lw $4,0($5)   add $3,$2,$1
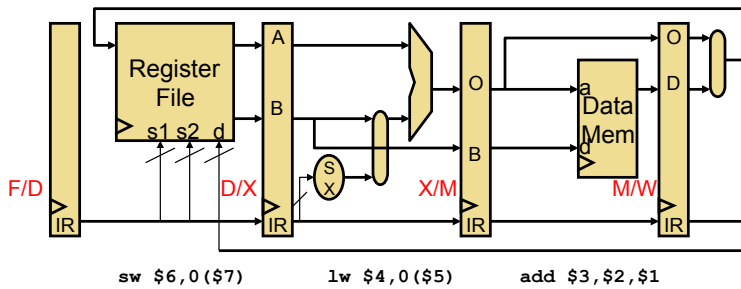
- Let's forget about branches and the control for a while
- The three insn sequence we saw earlier executed fine…
  - But it wasn't a real program
  - Real programs have **data dependences**
    - They pass values via registers and memory

Dependent Operations

- Independent operations

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

- Would this program execute correctly on a pipeline?

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

- What about this program?

  ```
  add $3,$2,$1
  lw $4,0($3)
  addi $6,1,$3
  sw $3,0($7)
  ```

Data Hazards

sw $3,0($7)   addi $6,1,$3   lw $4,0($3)   add $3,$2,$1
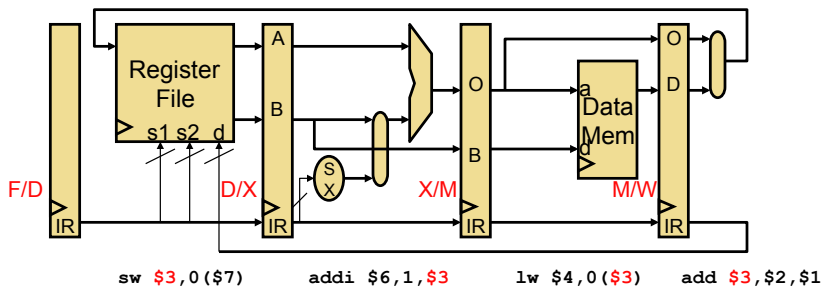
- Would this "program" execute correctly on this pipeline?
  - Which insns would execute with correct inputs?
  - **add** is writing its result into $3 in current cycle
  - **lw** read $3 two cycles ago → got wrong value
  - **addi** read $3 one cycle ago → got wrong value
  - **sw** is reading $3 this cycle → maybe (depending on regfile design)

Memory Data Hazards

lw $4,0($1)   sw $5,0($1)
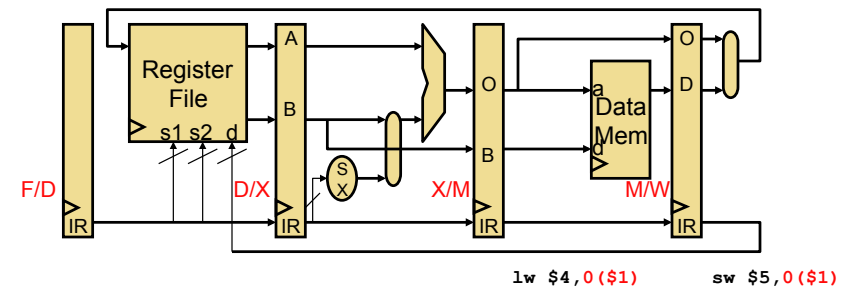
- Are memory data hazards a problem for this pipeline? No
  - **lw** following **sw** to same address in next cycle, gets right value
  - Why? Data mem read/write always take place in same stage
- Data hazards through registers? Yes (previous slide)
  - Occur because register write is three stages after register read
  - Can only read a register value three cycles after writing it

Observation!
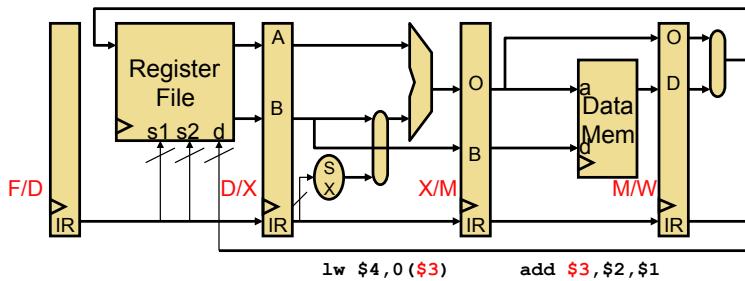
lw $4,0($3)   add $3,$2,$1

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

Reducing Data Hazards: Bypassing

lw $4,0($3)   add $3,$2,$1
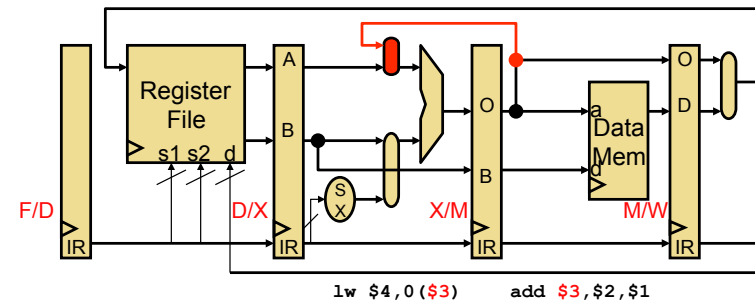
- **Bypassing**
  - Reading a value from an intermediate (μarchitectural) source
  - Not waiting until it is available from primary source
  - Here, we are bypassing the register file
  - Also called **forwarding**

WX Bypassing

lw $4,0($3)   add $3,$2,$1
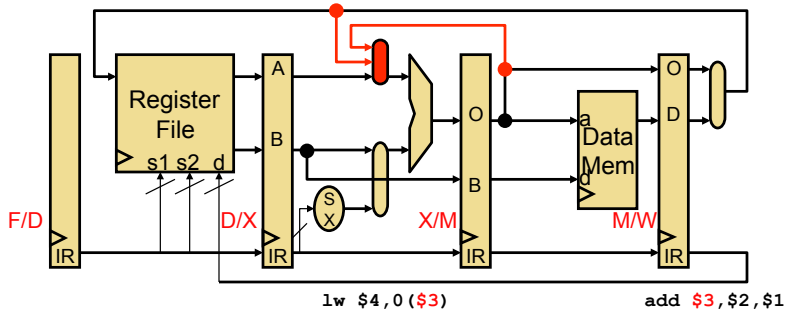
- What about this combination?
  - Add another bypass path and MUX (multiplexor) input
  - First one was an **MX** bypass
  - This one is a **WX** bypass

ALUinB Bypassing

add $4,$2,$3   add $3,$2,$1
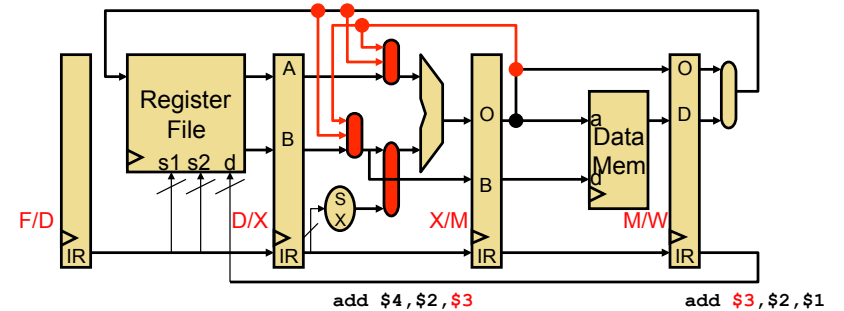
- Can also bypass to ALU input B

WM Bypassing?

sw $3,0($4)   lw $3,0($2)
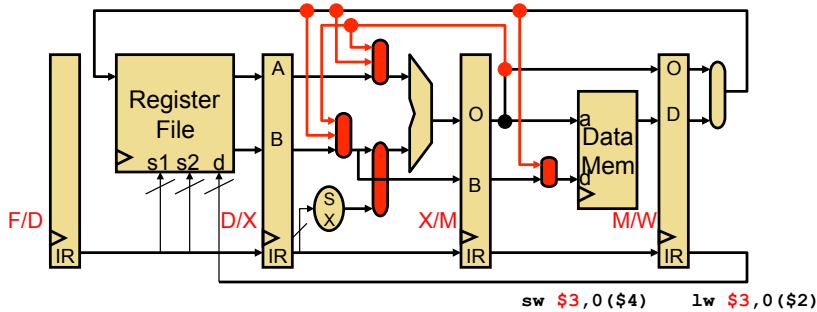
- Does WM bypassing make sense?
  - Not to the address input (why not?)
  - But to the store data input, yes

Bypass Logic
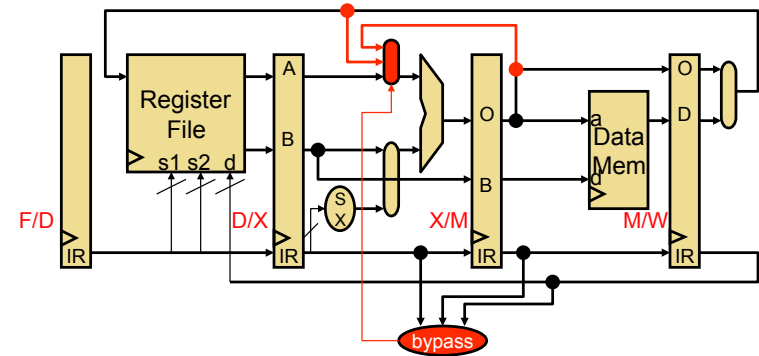
- Each MUX has its own, here it is for MUX ALUinA

  (D/X.IR.RegSource1 == X/M.IR.RegDest) => 0
  (D/X.IR.RegSource1 == M/W.IR.RegDest) => 1
  Else => 2

Pipeline Diagrams with Bypassing

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

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

  - Example: full bypassing, use WX bypass

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

  - Example: WM bypass

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

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

Have We Prevented All Data Hazards?

add $4,$2,$3   lw $3,4($2)
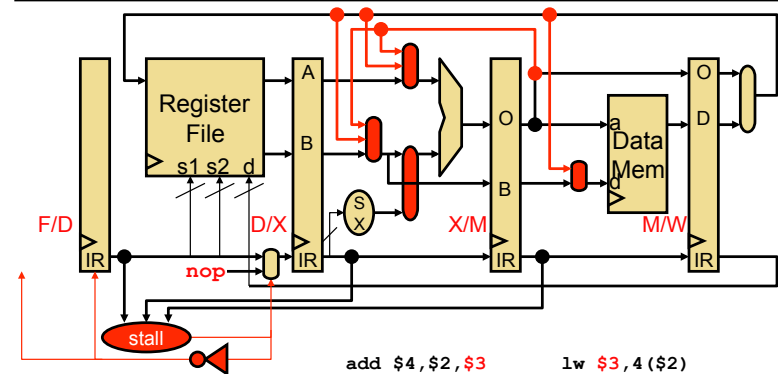
- No. Consider a "load" followed by a dependent "add" insn
- Bypassing alone isn't sufficient!
- Hardware solution: detect this situation and inject a stall cycle
- Software solution: ensure compiler doesn't generate such code

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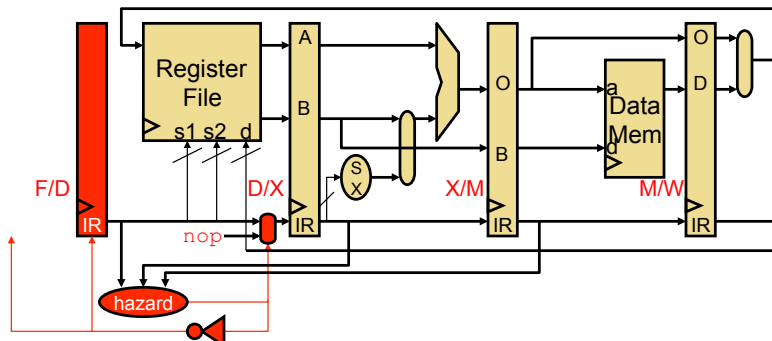
- Prevent F/D insn from reading (advancing) this cycle
  - Write **nop** into D/X.IR (effectively, insert **nop** in hardware)
  - Also reset (clear) the datapath control signals
  - Disable F/D latch and PC write enables (why?)
- Re-evaluate situation next cycle

Stalling on Load-To-Use Dependences
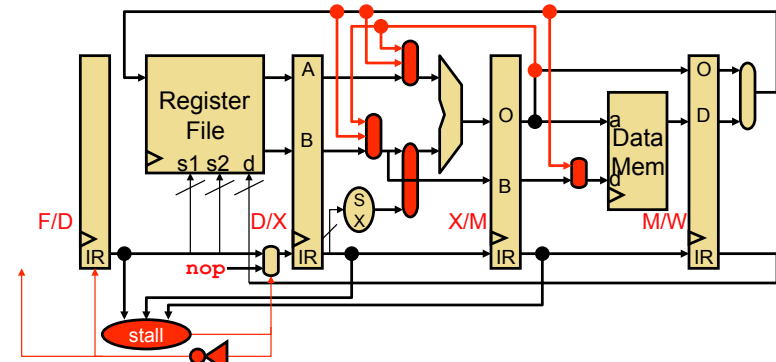
add $4,$2,$3   lw $3,4($2)

Stall = (D/X.IR.Operation == LOAD) &&
     ((F/D.IR.RegSrc1 == D/X.IR.RegDest) ||
      ((F/D.IR.RegSrc2 == D/X.IR.RegDest) && (F/D.IR.OP != STORE))

Stalling on Load-To-Use Dependences
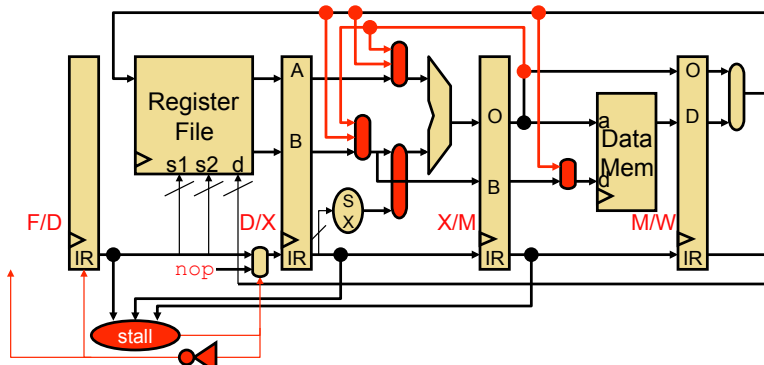
add $4,$2,$3   (stall bubble)   lw $3,4($2)

Stall = (D/X.IR.Operation == LOAD) &&
     ((F/D.IR.RegSrc1 == D/X.IR.RegDest) ||
      ((F/D.IR.RegSrc2 == D/X.IR.RegDest) && (F/D.IR.OP != STORE))

Stalling on Load-To-Use Dependences
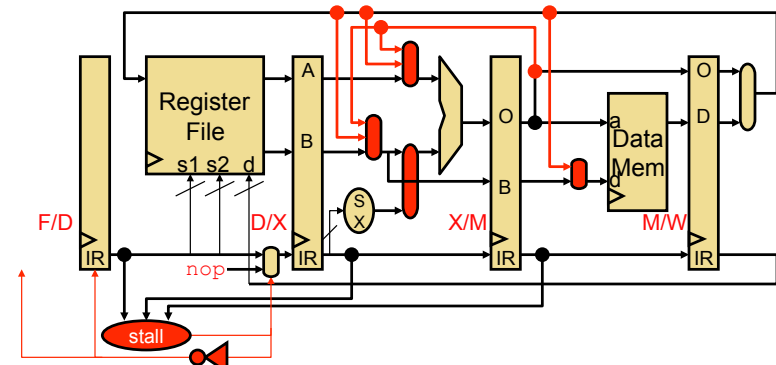
add $4,$2,$3   (stall bubble)   lw $3,…

Stall = (D/X.IR.Operation == LOAD) &&
     ((F/D.IR.RegSrc1 == D/X.IR.RegDest) ||
      ((F/D.IR.RegSrc2 == D/X.IR.RegDest) && (F/D.IR.OP != STORE))

Performance Impact of Load/Use Penalty

- Assume
  - Branch: 20%, load: 20%, store: 10%, other: 50%
  - 50% of loads are followed by dependent instruction
    - require 1 cycle stall (I.e., insertion of 1 `nop`)

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

Reducing Load-Use Stall Frequency

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

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

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

Pipelining and Multi-Cycle Operations
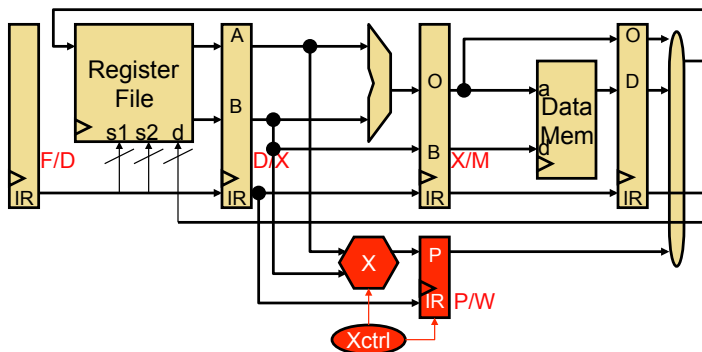
- What if you wanted to add a multi-cycle operation?
  - E.g., 4-cycle multiply
  - **P/W**: separate output latch connects to W stage
  - Controlled by pipeline control finite state machine (FSM)

A Pipelined Multiplier
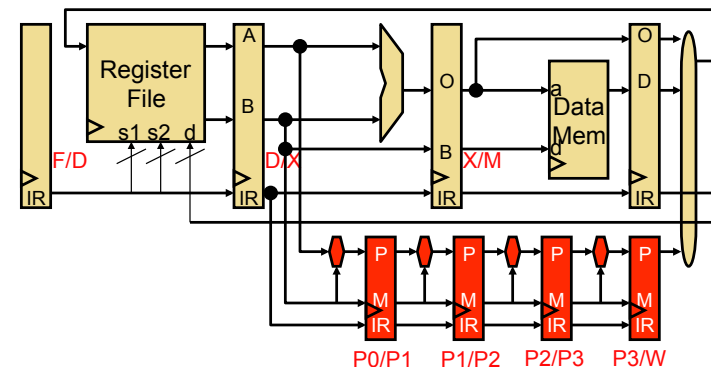
- Multiplier itself is often pipelined, what does this mean?
  - Product/multiplicand register/ALUs/latches replicated
  - Can start different multiply operations in consecutive cycles

Pipeline Diagram with 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 trying 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

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

Corrected Pipeline Diagram

- With the correct stall logic
  - Prevent mis-ordered writes to the same register
  - Why two cycles of delay?

	1	2	3	4	5	6	7	8	9
mul $4,$3,$5	F	D	P0	P1	P2	P3	W		
addi $4,$1,1		F	d*	d*	D	X	M	**W**	
…									
…									
add $10,$4,$6						F	D	X	M	W

- **Multi-cycle operations complicate pipeline logic**

Pipelined Functional Units

- Almost all multi-cycle functional units are pipelined
  - Each operation takes N cycles
  - But can start initiate a new (independent) operation every cycle
  - Requires internal latching and some hardware replication
  + A cheaper way to add bandwidth than multiple non-pipelined units

	1	2	3	4	5	6	7	8	9	10	11
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

Control Dependences and Branch Prediction

What About Branches?
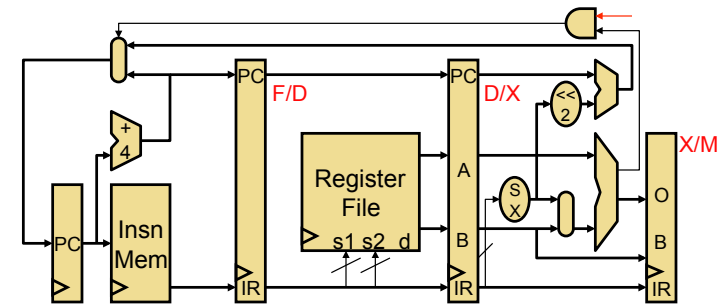
- **Control hazards options**
  - Could just stall to wait for branch outcome (two-cycle penalty)
  - **Fetch past branch insns before branch outcome is known**
    - Default: assume "**not-taken**" (at fetch, can't tell it's a branch)

Branch Recovery
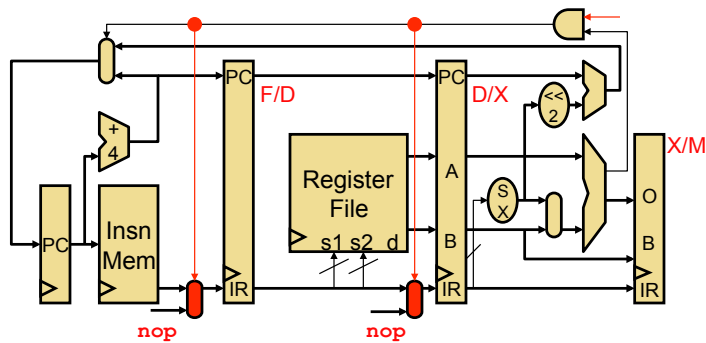
- **Branch recovery**: what to do when branch is actually taken
  - Insns that will be written into F/D and D/X are wrong
  - **Flush them**, i.e., replace them with **nops**
  + They haven't had written permanent state yet (regfile, DMem)
  − Two cycle penalty for taken branches

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 =
  $$1 + \mathbf{0.20 * 0.75 * 2} = 1.3$$
  − **Branches cause 30% slowdown**
    - Even worse with deeper pipelines
  - How do we reduce this penalty?

Big Idea: Speculative Execution

- Speculation: "risky transactions on chance of profit"

- **Speculative execution**
  - Execute before all parameters known with certainty
  - **Correct speculation**
    + Avoid stall, improve performance
  - **Incorrect speculation (mis-speculation)**
    – Must abort/flush/squash incorrect insns
    – Must undo incorrect changes (recover pre-speculation state)
  - The "game": **[%$_{correct}$ * gain] – [(1–%$_{correct}$) * penalty]**

- **Control speculation**: speculation aimed at control hazards
  - Unknown parameter: are these the correct insns to execute next?

Control Speculation and Recovery

		1	2	3	4	5	6	7	8	9
Correct:	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`)

		1	2	3	4	5	6	7	8	9
Recovery:	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., `cmplt`, `slt`) for complex tests
  – Must bypass into decode stage now, too

	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			

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
  - Flush on **mis-prediction**

Branch Prediction Performance

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

Dynamic Branch Prediction Components

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

Branch Direction Prediction

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

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

Branch History Table (BHT)

- **Branch history table (BHT)**: simplest direction predictor
  - PC indexes table of bits (0 = N, 1 = T), no tags
  - Essentially: branch will go same way it went last time
  - Problem: 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 (which is not contrived, by the way)
  - Can we do even better?

Correlated Predictor

- **Correlated (two-level) predictor** [Patt]
  - 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
    - 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*	T	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 capacity problem
  - Idea: combine two predictors
    - **Simple BHT** predicts history independent branches
    - **Correlated predictor** predicts only branches that need history
    - **Chooser** assigns branches to one predictor or the other
    - Branches start in simple BHT, move mis-prediction threshold
  - + Correlated predictor can be made **smaller**, handles fewer branches
 - + 90–95% accuracy
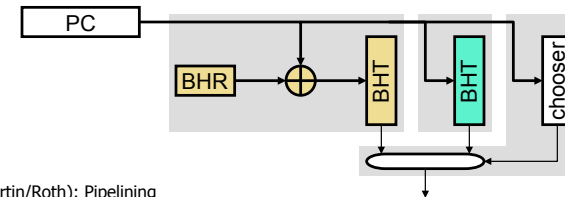
When to Perform Branch Prediction?

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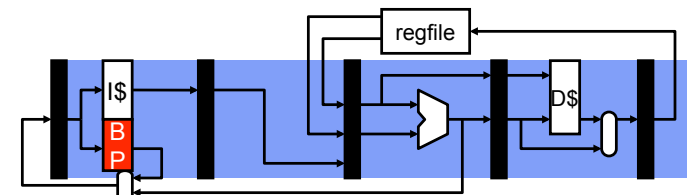
- Step #1: is it a branch?
  - Easy after decode… during fetch: **predictor**
- Step #2: is the branch taken or not taken?
  - **Direction predictor** (as before)
- Step #3: if the branch is taken, where does it go?
  - **Branch target predictor (BTB)**
  - Supplies target PC if branch is taken

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. As before, this is only a prediction

# Branch Target Buffer (continued)

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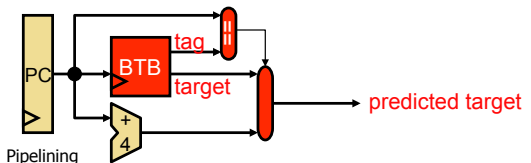
Why Does a BTB Work?

- Because most control insns use **direct targets**
  - Target encoded in insn itself → same "taken" target every time

- What about **indirect targets**?
  - Target held in a register → can be different each time
  - 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 but...

Return Address Stack (RAS)
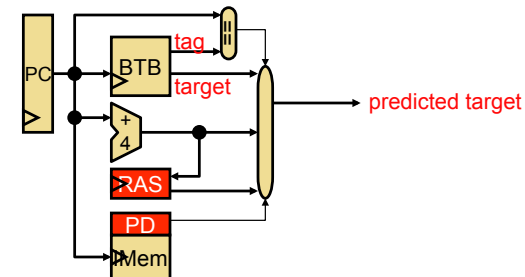
- **Return address stack (RAS)**
  - Call instruction? RAS[TOS++] = PC+4
  - Return instruction? Predicted-target = RAS[--TOS]
  - Q: how can you tell if an insn is a call/return before decoding it?
    - Accessing RAS on every insn BTB-style doesn't work
  - Answer: **pre-decode bits** in Imem, written when first executed
    - Can also be used to signify branches

Putting It All Together

- BTB & branch direction predictor during fetch
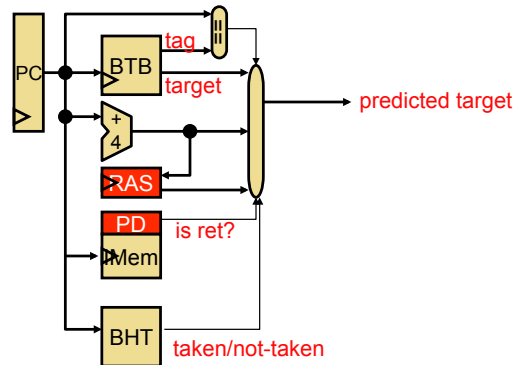
- If branch prediction correct, no taken branch penalty

Branch Prediction Performance

- Dynamic branch prediction
  - 20% of instruction branches
  - Simple predictor: branches predicted with 75% accuracy
    - CPI = 1 + (20% * **25%** * 2) = **1.1**
  - More advanced predictor: 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)

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 r6,1,r5		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, predic**a**tion
  - 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)

	1	2	3	4	5	6	7	8	9	
setp.eq r3,p3	F	D	X	M	W					
sub.p r6,1,r5,p3		F	D	X	--	--				annulled
targ:add r4,r5,r4			F	D	X	M	W			

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

	1	2	3	4	5	6	7	8	9	
setcc r3	F	D	X	M	W					
sub.nz r6,1,r5		F	D	X	--	--				annulled
targ:add r4,r5,r4			F	D	X	M	W			

  - 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)

|	1	2	3	4	5	6	7	8	9
`sub r6,1,r9`		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

- Cost/benefit analysis
  - Benefit: predication avoids branches
    - Thus avoiding mis-predictions
    - Also reduces pressure on predictor table (few branches to track)
  - Cost: extra (annulled) instructions
- As branch predictors are highly accurate…
  - Might not help:
    - 5-stage pipeline, two instruction on each path of if-then-else
    - No performance gain, likely slower if branch predictable
  - Or even hurt!
  - But can help:
    - Deeper pipelines, hard-to-predict branches, and few added insn
- Thus, prediction is useful, but not a panacea

Research

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%
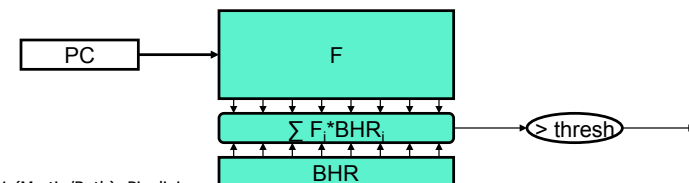
# More Research: GEHL Predictor

- 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

Championship Branch Prediction

- **CBP**
  - Workshop held in conjunction with MICRO
  - Submitted code is tested on standard branch traces
  - Highest prediction accuracy wins

- Two tracks
  - Idealistic: predictor simulator must run in under 2 hours
  - Realistic: predictor must synthesize into 32KB + 256 bits or less

- 2006 winners
  - Realistic: L-TAGE (GEHL follow-on)
  - Idealistic: GTL (another GEHL follow-on)
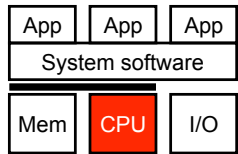
Research: Runahead Execution
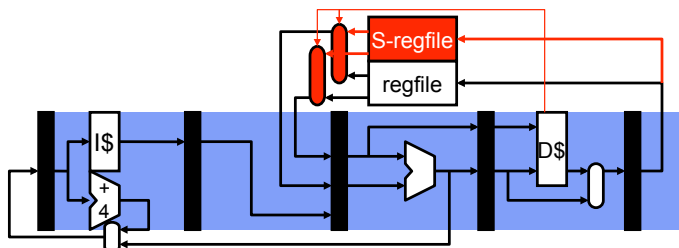
- In-order writebacks essentially imply stalls on D$ misses
  - Can save power … or use idle time for performance
- **Runahead execution** [Dundas+]
  - Shadow regfile kept in sync with main regfile (write to both)
  - D$ miss: continue executing using shadow regfile (disable stores)
  - D$ miss returns: flush pipe and restart with stalled PC
  - + Acts like a smart prefetch engine
  - + Performs better as cache t_{miss} grows (relative to clock period)

Research: Razor
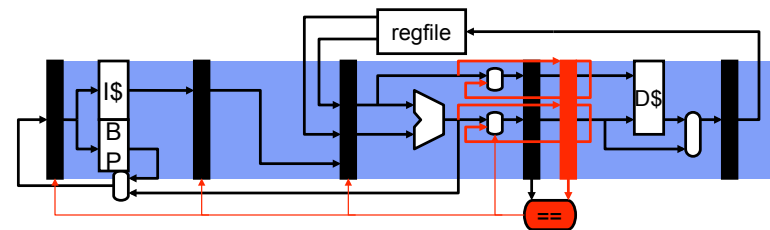
- **Razor** [Uht, Ernst+]
  - Identify pipeline stages with narrow signal margins (e.g., **X**)
  - Add **"Razor"** X/M latch: relatches X/M input signals after safe delay
  - Compare X/M latch with "safe" razor X/M latch, different?
    - Flush F,D,X & M
    - Restart M using X/M razor latch, restart 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

Summary

App	App	App
System software		

Mem	CPU	I/O

- Principles of pipelining
  - Effects of overhead and hazards
  - Pipeline diagrams
- Data hazards
  - Stalling and bypassing
- Control hazards
  - Branch prediction
  - Predication