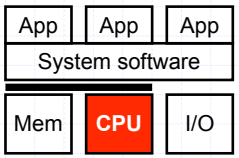
CIS 371
Computer Organization and Design

Unit 6: Pipelining

This Unit: (Scalar In-Order) Pipelining

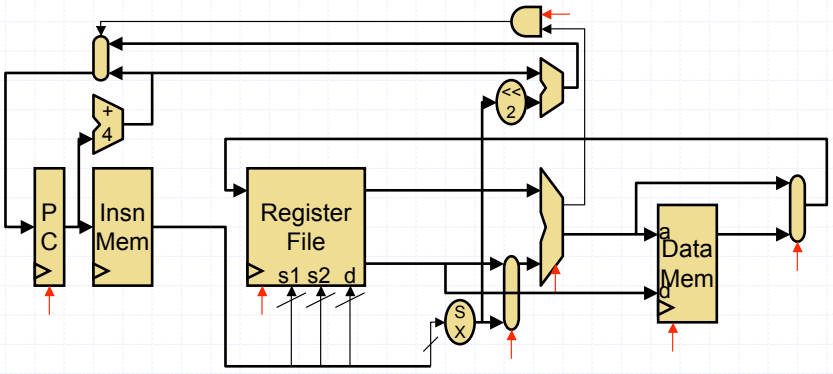


- Basic Pipelining
  - Pipeline control
- Data Hazards
  - Software interlocks and scheduling
  - Hardware interlocks and stalling
  - Bypassing
- Control Hazards
  - Branch prediction
- Multi-cycle operations
- Exceptions

Readings

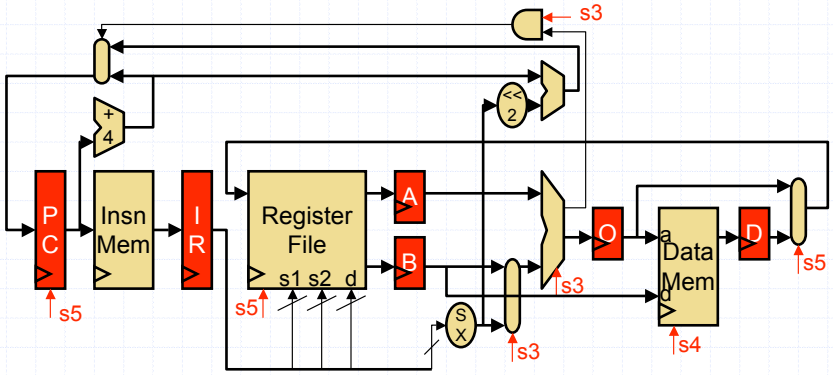
- P+H
  - Chapter 6 (6.1 - 6.8)

Single-Cycle Datapath Performance



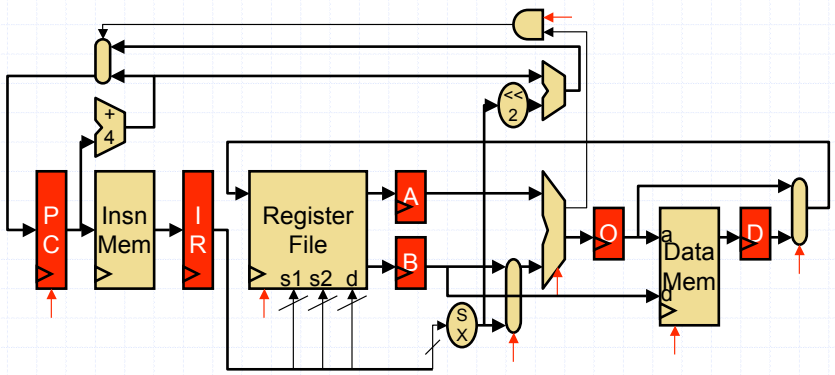
- Goes against make common case fast (MCCF) principle
  - \+ Low CPI: 1
  - – Long clock period: to accommodate slowest instruction

Alternative: Multi-Cycle Datapath



- **Multi-cycle datapath**: attacks high clock period
  - Cut datapath into multiple stages (5 here), isolate using FFs
  - FSM control "walks" insns thru stages (by staging control signals)
  - \+ Insns can bypass stages and exit early (memory ops vs alu ops)

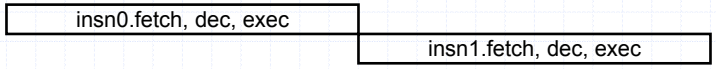
Multi-Cycle Datapath Performance



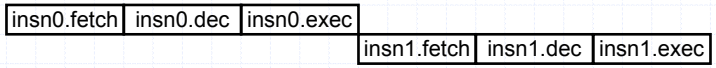
- Opposite performance split of single-cycle datapath
  - \+ Short clock period
  - – High CPI

Clock Period and CPI

- Single-cycle datapath
  - \+ Low CPI: 1
  - – Long clock period: to accommodate slowest insn



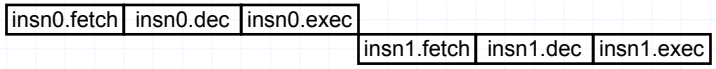
- Multi-cycle datapath
  - \+ Short clock period
  - – High CPI



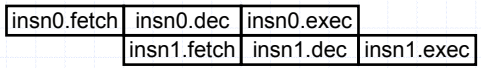
- Can we have both low CPI and short clock period?
  - – No good way to make a single insn go faster
  - \+ Insn latency doesn't matter anyway … insn throughput matters
  - Key: **exploit inter-insn parallelism**

Pipelining

- **Pipelining**: important performance technique
  - **Improves insn throughput rather than insn latency**
  - **Exploits parallelism at insn-stage level to do so**
  - Begin with multi-cycle design

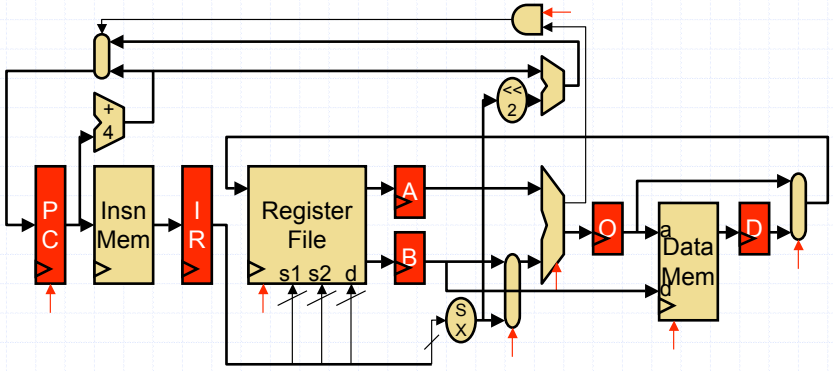


  - When insn advances from stage 1 to 2, next insn enters stage 1

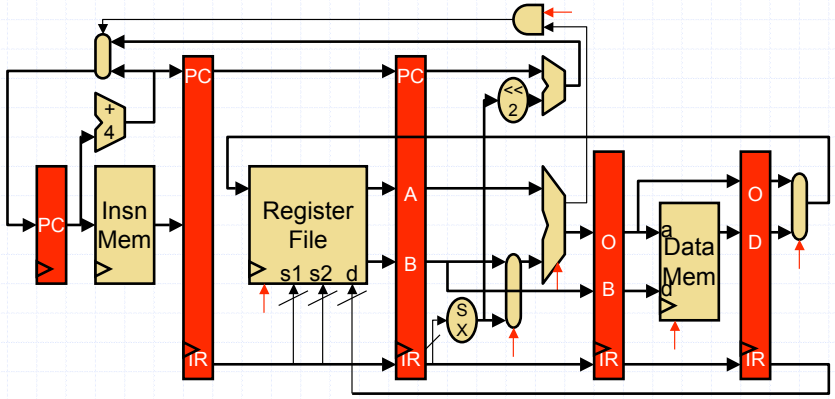


  - Individual insns take same number of stages
  - \+ **But insns enter and leave at a much faster rate**
  - Breaks "fetch/execute" Von Neumann (VN) loop ... but maintains illusion
- Automotive assembly line analogy

5 Stage Multi-Cycle Datapath

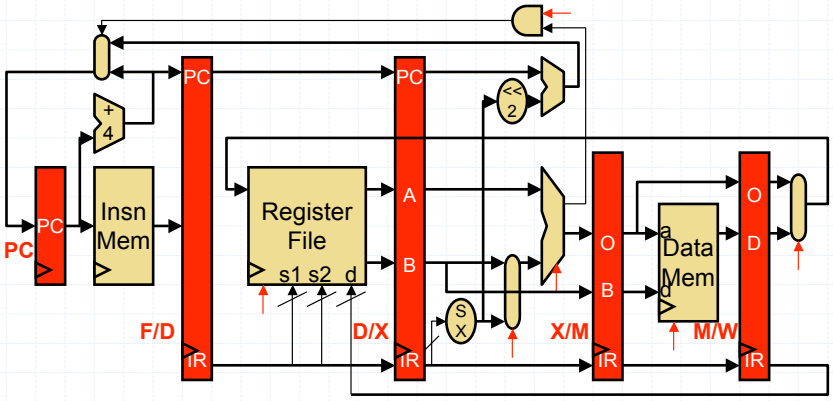


5 Stage Pipelined Datapath



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

Pipeline Terminology

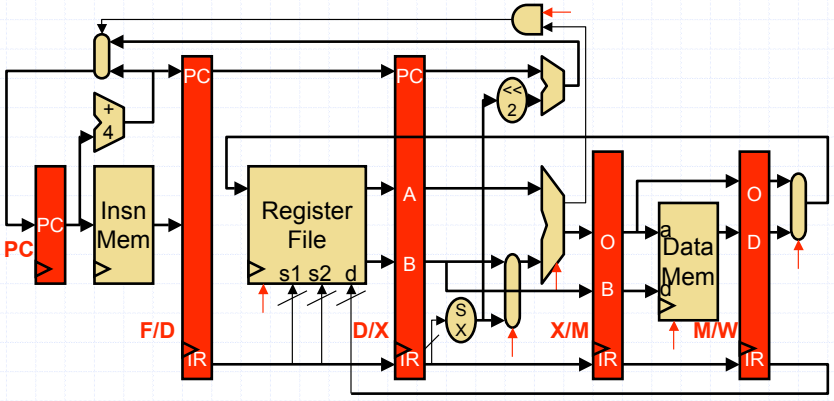


- Stages: **F**etch, **D**ecode, e**X**ecute, **M**emory, **W**riteback
- Latches (pipeline registers): **PC**, **F/D**, **D/X**, **X/M**, **M/W**

Some More Terminology

- **Scalar pipeline**: one insn per stage per cycle
  - Alternative: "superscalar" (later)
- **In-order pipeline**: insns enter execute stage in VN order
  - Alternative: "out-of-order" (maybe later)
- **Pipeline depth**: number of pipeline stages
  - Nothing magical about five
  - Trend has been to deeper pipelines

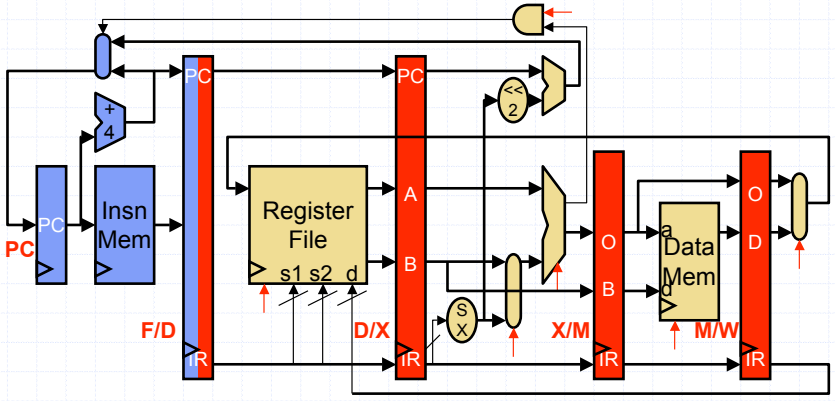
Pipeline Example: Cycle 1




```
add $3,$2,$1
```

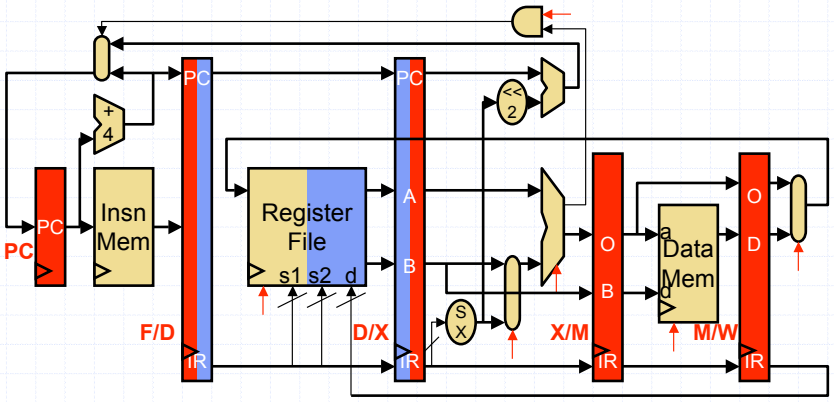
- 3 instructions

Pipeline Example: Cycle 2



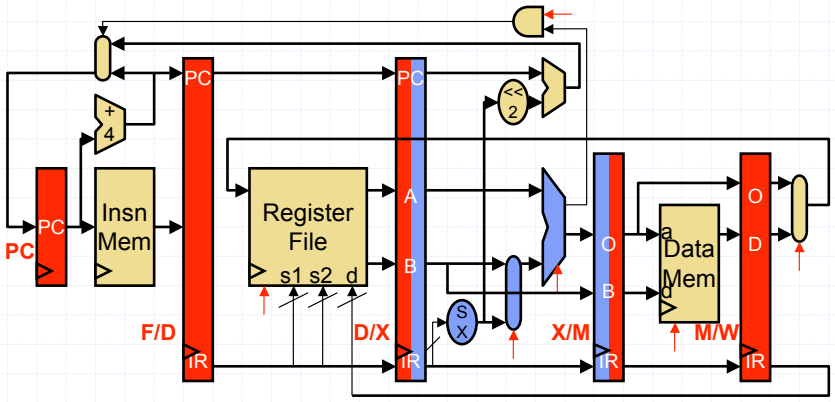

```
lw $4,0($5)     add $3,$2,$1
```


Pipeline Example: Cycle 3




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

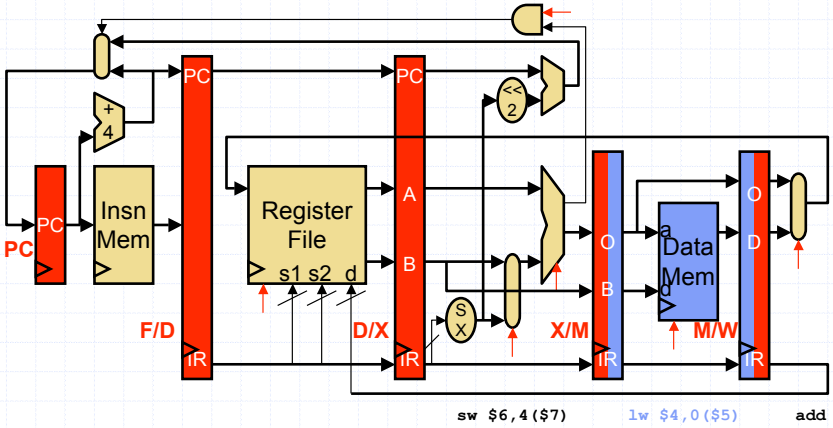

Pipeline Example: Cycle 4



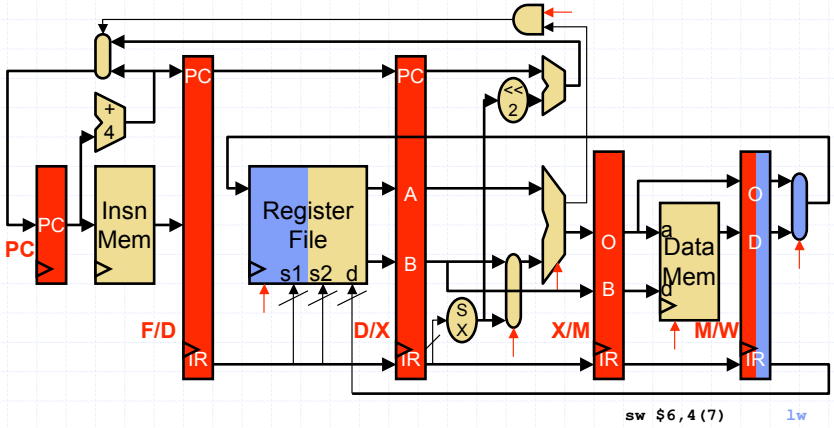

```
sw $6,4($7)     lw $4,0($5)     add $3,$2,$1
```

- 3 instructions

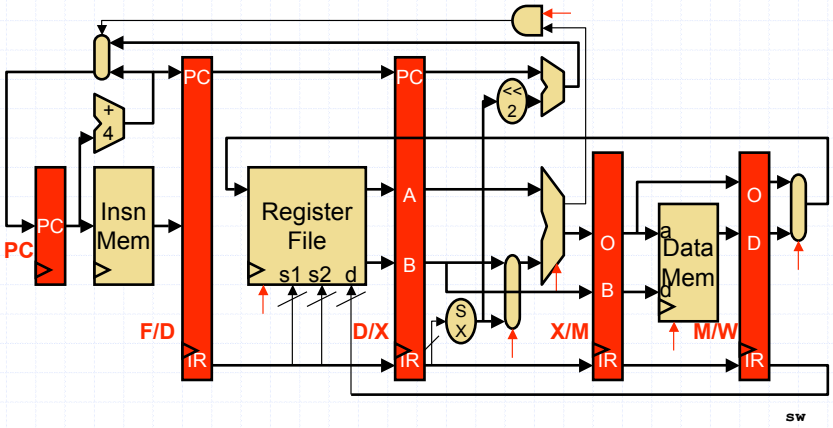
Pipeline Example: Cycle 5



Pipeline Example: Cycle 6



Pipeline Example: Cycle 7



Pipeline Diagram

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

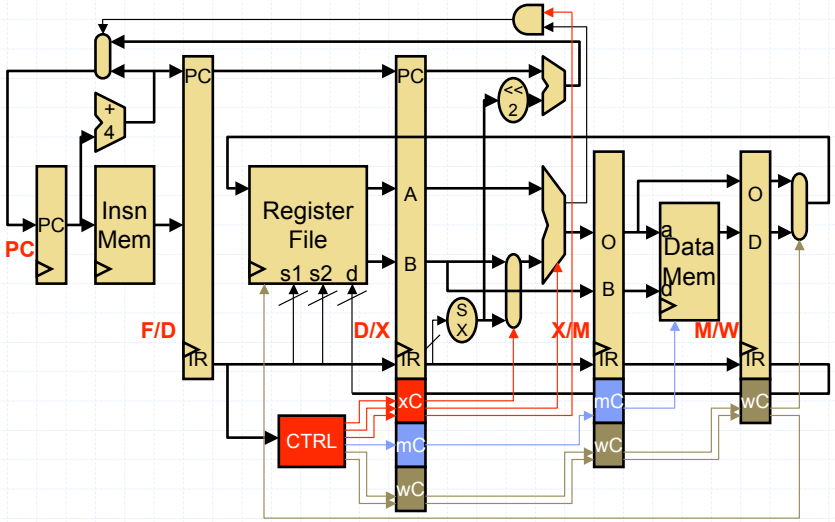
	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,0($5)`		F	D	**X**	M	W			
`sw $6,4($7)`			F	D	X	M	W		

What About Pipelined Control?

- Should it be like single-cycle control?
  - But individual insn signals must be staged
- Should it be like multi-cycle control?
  - But all stages are simultaneously active
- How many different controllers are we going to need?
  - One for each insn in pipeline?

- Solution: use simple single-cycle control, but pipeline it
  - Single controller

Pipelined Control



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 = (0.2*3+0.2*5+0.6*4) = 4
    - Why is clock period 11ns and not 10ns?
  - Performance = 44ns/insn

- Pipelined
  - Clock period = **12ns**
  - CPI = **1.5** (on average insn completes every 1.5 cycles)
  - Performance = **18ns/insn**

Q1: Why Is Pipeline Clock Period …

- … > delay thru datapath / number of pipeline stages?
  - Latches (FFs) add delay
  - Pipeline stages have different delays, clock period is max delay

- Both factors have implications for ideal number pipeline stages

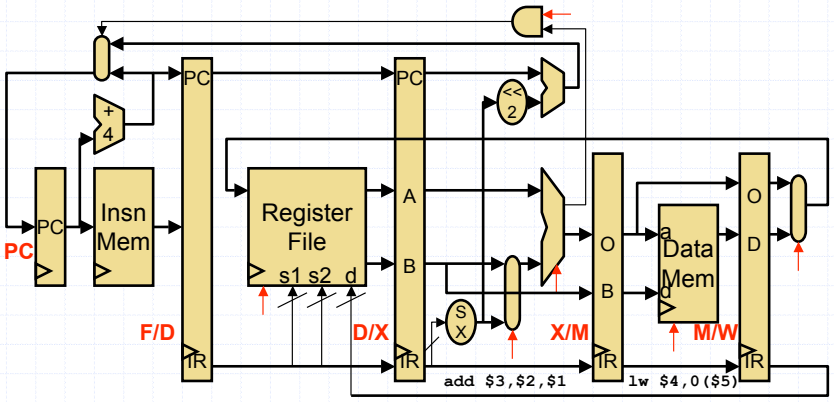
Q2: Why Is Pipeline CPI…

- … > 1?
  - CPI for scalar in-order pipeline is 1 **+ stall penalties**
  - Stalls used to resolve hazards
    - **Hazard**: condition that jeopardizes VN illusion
    - **Stall**: artificial pipeline delay introduced to restore VN illusion
- Calculating pipeline CPI
  - **Frequency of stall** * **stall cycles**
  - Penalties add (stalls generally don't overlap in in-order pipelines)
  - 1 + stall-freq$_1$*stall-cyc$_1$ + stall-freq$_2$*stall-cyc$_2$ + …
- Correctness/performance/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

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?

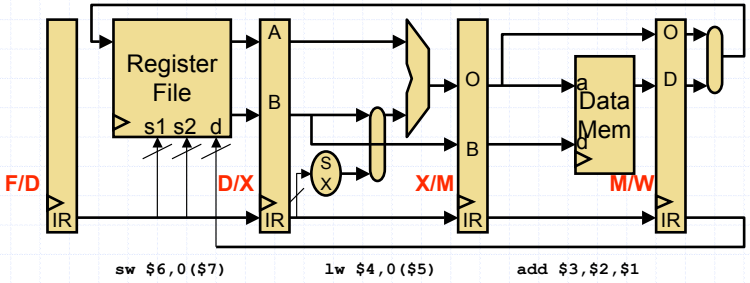


- Could/should we allow `add` to skip M and go to W? No
  - It wouldn't help: peak fetch still only 1 insn per cycle
  - **Structural hazards**: imagine `add` follows `lw`

Structural Hazards

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

Data Hazards



- Let's forget about branches and the control for a while
- The three insn sequence we saw earlier executed fine…
  - But it wasn't a real program
  - Real programs have **data dependences**
    - They pass values via registers and memory

Dependent Operations

- Independent operations

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

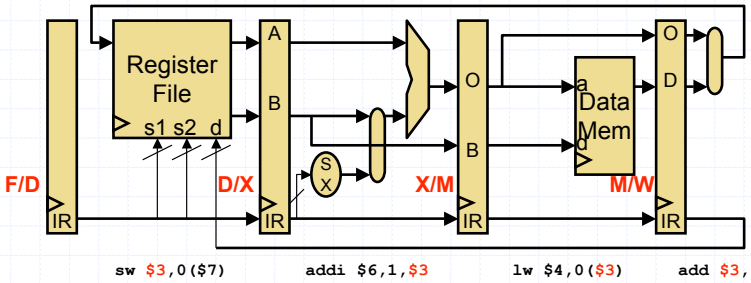
- Would this program execute correctly on a pipeline?

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

- What about this program?

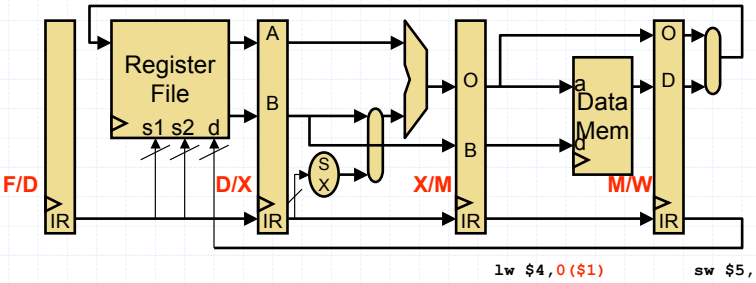
```
add $3,$2,$1
lw $4,0($3)
addi $6,1,$3
sw $3,0($7)
```


Data Hazards



- Would this "program" execute correctly on this pipeline?
  - Which insns would execute with correct inputs?
  - **add** is writing its result into $3 in current cycle
  - **lw** read $3 2 cycles ago → got wrong value
  - **addi** read $3 1 cycle ago → got wrong value
  - **sw** is reading $3 this cycle → OK (regfile timing: write first half)

Memory Data Hazards



- What about data hazards through memory? No
  - **lw** following **sw** to same address in next cycle, gets right value
  - Why? DMem read/write take place in same stage
- Data hazards through registers? Yes (previous slide)
  - Occur because register write is 3 stages after register read
  - Can only read a register value 3 cycles after writing it

Fixing Register Data Hazards

- Can only read register value 3 cycles after writing it

- **Option #1: make sure programs don't do it**
  - Compiler puts two independent insns between write/read insn pair
    - If they aren't there already
  - Independent means: "do not interfere with register in question"
    - Do not write it: otherwise meaning of program changes
    - Do not read it: otherwise create new data hazard
  - **Code scheduling**: compiler moves around existing insns to do this
  - If none can be found, must use **nops** (no-operation)

  - This is called **software interlocks**
    - **MIPS**: **M**icroprocessor w/out **I**nterlocking **P**ipeline **S**tages

Software Interlock Example

```
add $3,$2,$1
nop
nop
lw $4,0($3)
sw $7,0($3)
add $6,$2,$8
addi $3,$5,4
```

- Can any of last three insns be scheduled between first two
  - `sw $7,0($3)`? No, creates hazard with `add $3,$2,$1`
  - `add $6,$2,$8`? OK
  - `addi $3,$5,4`? No, `lw` would read `$3` from it
  - Still need one more insn, use `nop`

```
add $3,$2,$1
add $6,$2,$8
nop
lw $4,0($3)
sw $7,0($3)
addi $3,$5,4
```


Software Interlock Performance

- Same deal
  - Branch: 20%, load: 20%, store: 10%, other: 50%

- Software interlocks
  - 20% of insns require insertion of 1 `nop`
  - 5% of insns require insertion of 2 `nops`

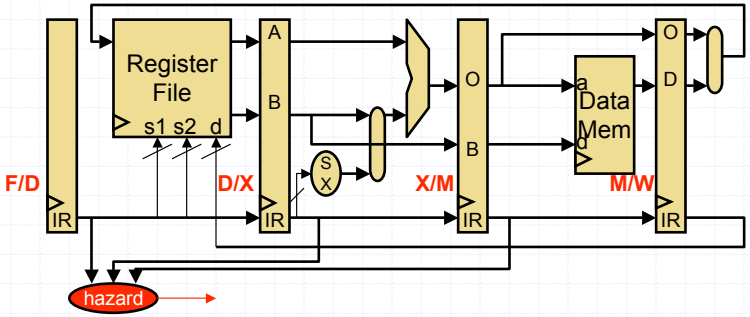
  - CPI is still 1 technically
  - But now there are more insns
  - #insns = 1 + 0.20*1 + 0.05*2 = **1.3**
  - **30% more insns (30% slowdown) due to data hazards**

Hardware Interlocks

- Problem with software interlocks? Not compatible
  - Where does **3** in "read register 3 cycles after writing" come from?
    - From structure (depth) of pipeline
  - What if next MIPS version uses a 7 stage pipeline?
    - Programs compiled assuming 5 stage pipeline will break

- A better (more compatible) way: **hardware interlocks**
  - Processor detects data hazards and fixes them
  - Two aspects to this
    - Detecting hazards
    - Fixing hazards

Detecting Data Hazards



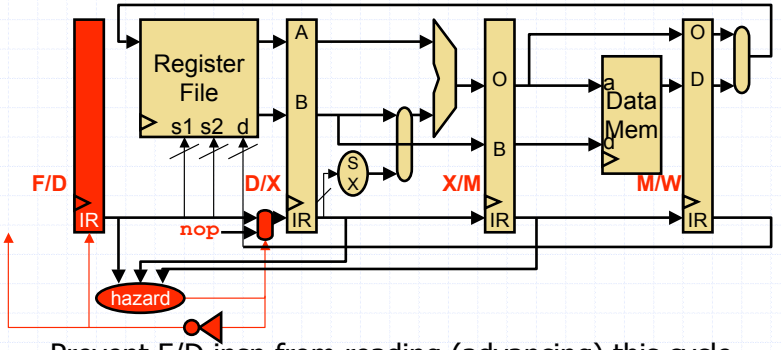
- Compare F/D insn input register names with output register names of older insns in pipeline

  Hazard =

  (F/D.IR.RS1 == D/X.IR.RD) || (F/D.IR.RS2 == D/X.IR.RD) ||

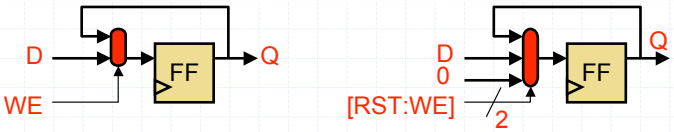
  (F/D.IR.RS1 == X/M.IR.RD) || (F/D.IR.RS2 == X/M.IR.RD)

Fixing Data Hazards



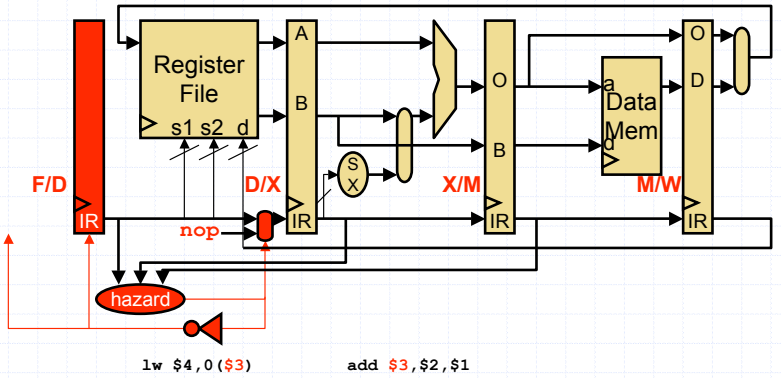
- Prevent F/D insn from reading (advancing) this cycle
  - Write `nop` into D/X.IR (effectively, insert `nop` in hardware)
  - Also reset (clear) the datapath control signals
  - Disable F/D latch and PC write enables (why?)
- Re-evaluate situation next cycle

Aside: Insert NOP/Reset Register



- Earlier: registers support separate clock, write enable
  - Useful for writes into register file
  - Also useful for implementing stalls
- Registers can also support **synchronous reset** (clear)
  - Useful for implementing stalls
  - Implement as additional hardwired 0 input to FF data mux
  - Resetting pipeline registers equivalent to inserting a NOP
    - If NOP is all zeros
    - If zero means "don't write" for all write-enable control signals
    - Design ISA/control signals to make sure this is the case

Hardware Interlock Example: cycle 1

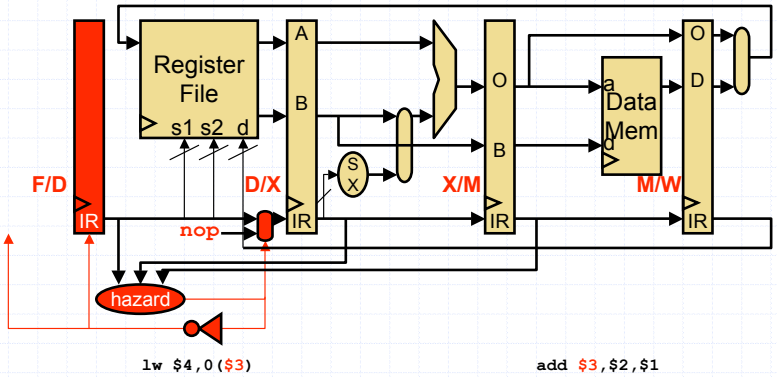


(F/D.IR.RS1 == D/X.IR.RD) || (**F/D.IR.RS2 == D/X.IR.RD**) ||

(F/D.IR.RS1 == X/M.IR.RD) || (F/D.IR.RS2 == X/M.IR.RD)

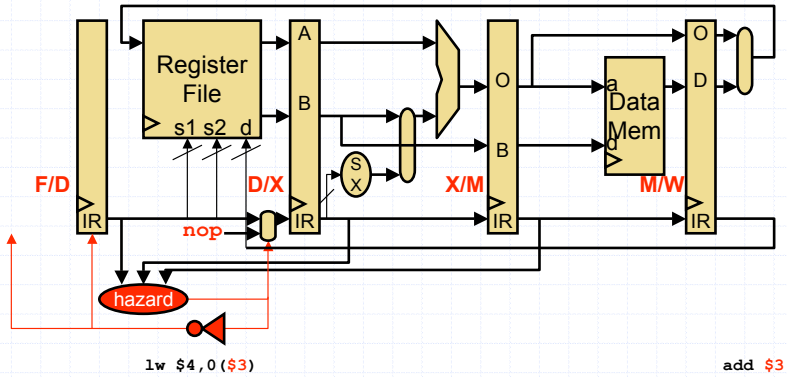
= **1**

Hardware Interlock Example: cycle 2



(F/D.IR.RS1 == D/X.IR.RD) || (F/D.IR.RS2 == D/X.IR.RD) ||
(F/D.IR.RS1 == X/M.IR.RD) || (**F/D.IR.RS2 == X/M.IR.RD**)
= **1**

Hardware Interlock Example: cycle 3



(F/D.IR.RS1 == D/X.IR.RD) || (F/D.IR.RS2 == D/X.IR.RD) ||
(F/D.IR.RS1 == X/M.IR.RD) || (F/D.IR.RS2 == X/M.IR.RD)
= **0**

Pipeline Control Terminology

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

Pipeline Diagram with Data Hazards

- Data hazard stall indicated with **d***
  - Stall propagates to younger insns

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

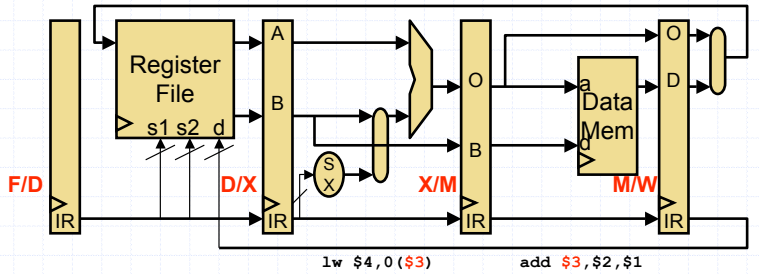
- This is not good (why?)

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

Hardware Interlock Performance

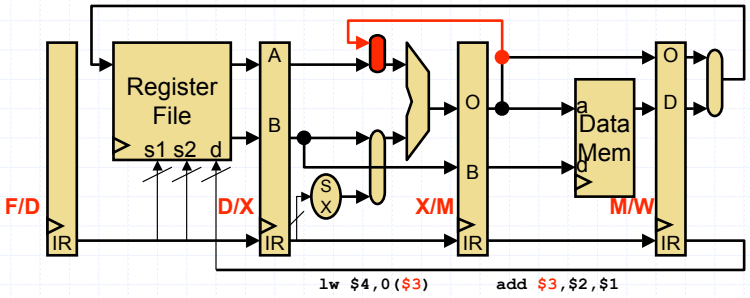
- Same deal
  - Branch: 20%, load: 20%, store: 10%, other: 50%
- Hardware interlocks: same as software interlocks
  - 20% of insns require 1 cycle stall (I.e., insertion of 1 `nop`)
  - 5% of insns require 2 cycle stall (I.e., insertion of 2 `nops`)
  - CPI = 1 * 0.20*1 + 0.05*2 = **1.3**
  - So, either CPI stays at 1 and #insns increases 30% (software)
  - Or, #insns stays at 1 (relative) and CPI increases 30% (hardware)
  - Same difference
- Anyway, we can do better

Observe



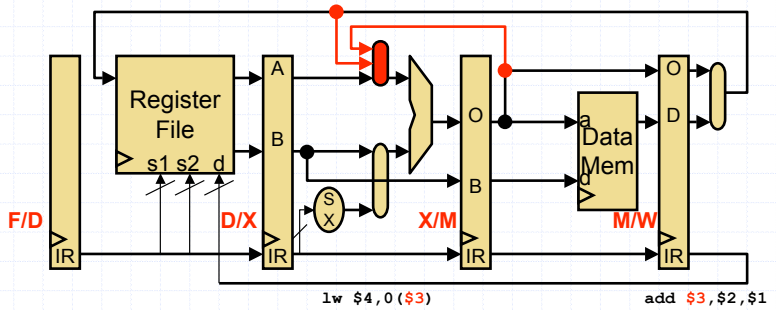
- Technically, this situation is broken
  - `lw $4,0($3)` has already read `$3` from regfile
  - `add $3,$2,$1` hasn't yet written `$3` to regfile
- But fundamentally, everything is OK
  - `lw $4,0($3)` hasn't actually used `$3` yet
  - `add $3,$2,$1` has already computed `$3`

Bypassing



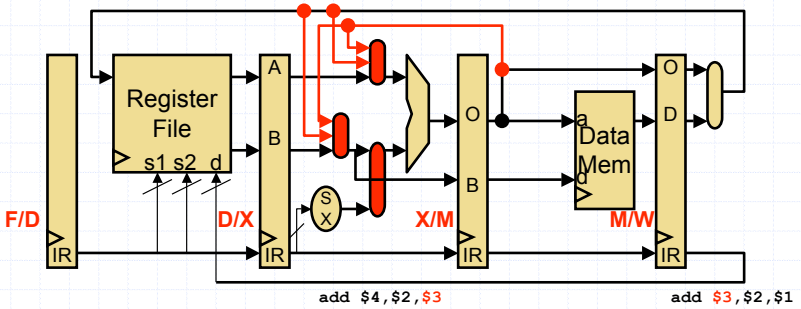
- **Bypassing**
  - Reading a value from an intermediate (μarchitectural) source
  - Not waiting until it is available from primary source
  - Here, we are bypassing the register file
  - Also called **forwarding**

WX Bypassing



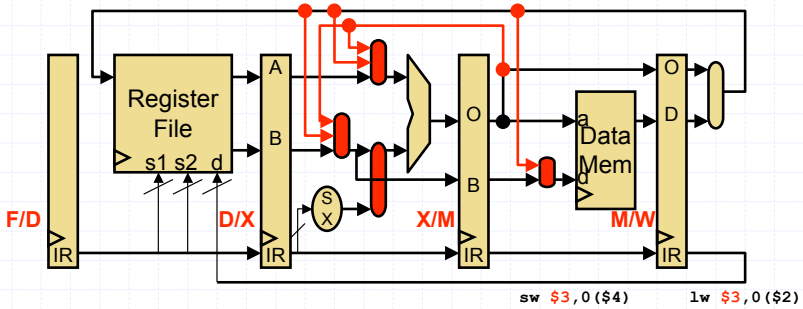
- What about this combination?
  - Add another bypass path and MUX input
  - First one was an **MX** bypass
  - This one is a **WX** bypass

ALUinB Bypassing



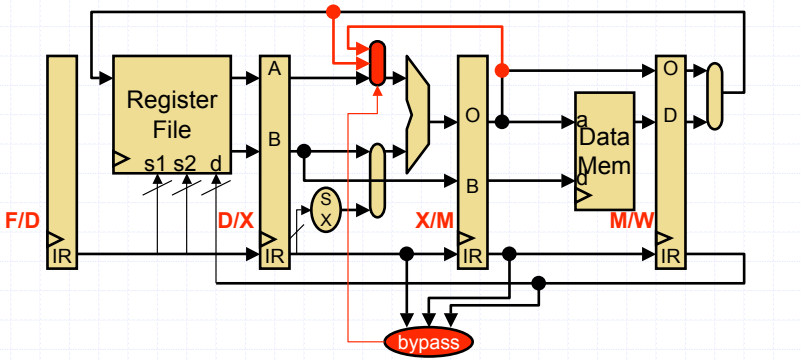
- Can also bypass to ALU input B

WM Bypassing?



- Does WM bypassing make sense?
  - Not to the address input (why not?)
  - But to the store data input, yes

Bypass Logic

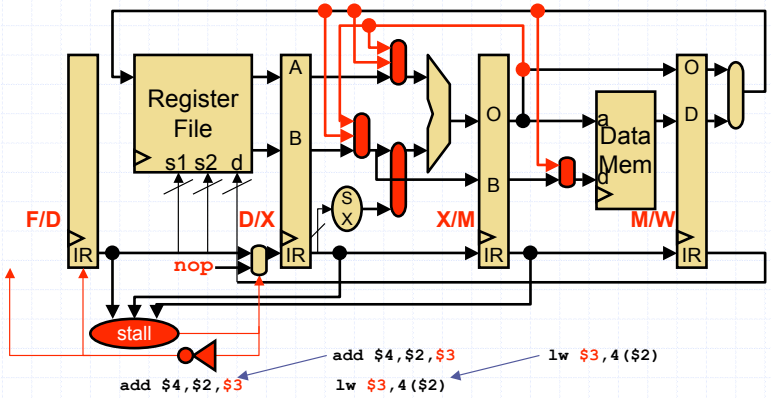


- Each MUX has its own, here it is for MUX ALUinA
  (D/X.IR.RS1 == X/M.IR.RD) => 0
  (D/X.IR.RS1 == M/W.IR.RD) => 1
  Else => 2

Bypass and Stall Logic

- Two separate things
  - Stall logic controls pipeline registers
  - Bypass logic controls MUXs
- But complementary
  - For a given data hazard: if can't bypass, must stall
- Slide #43 shows **full bypassing**: all bypasses possible
  - Is stall logic still necessary?

Yes, Load Output to ALU Input



Stall = (D/X.IR.OP == LOAD) &&
((F/D.IR.RS1 == D/X.IR.RD) ||
((F/D.IR.RS2 == D/X.IR.RD) && (F/D.IR.OP != STORE))

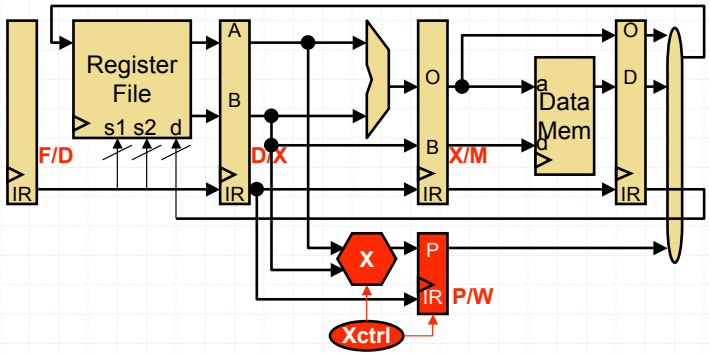
Pipeline Diagram With Bypassing

	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	

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

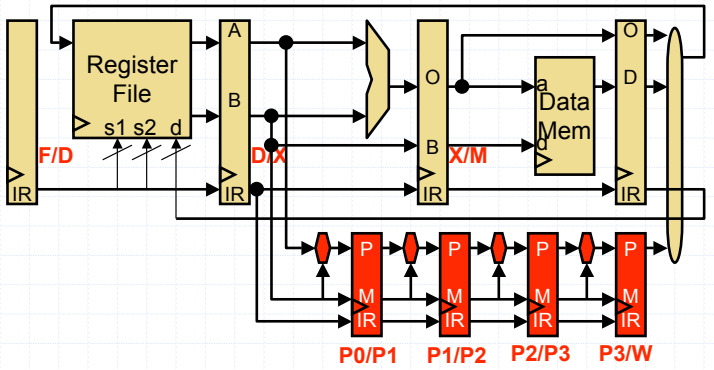
	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,4($3)`		F	D	X	M	W			
`sub $8,$3,$1`			F	D	X	M	W		
`addi $6,$4,1`				F	D	X	M	W	

Pipelining and Multi-Cycle Operations



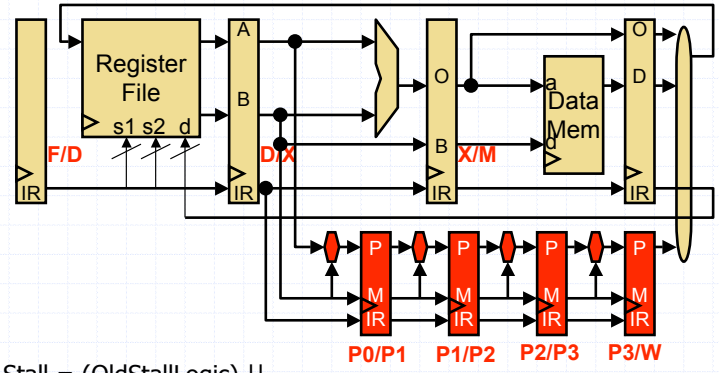
- What if you wanted to add a multi-cycle operation?
  - E.g., 4-cycle multiply
  - **P/W**: separate output latch connects to W stage
  - Controlled by pipeline control and multiplier FSM

A Pipelined Multiplier



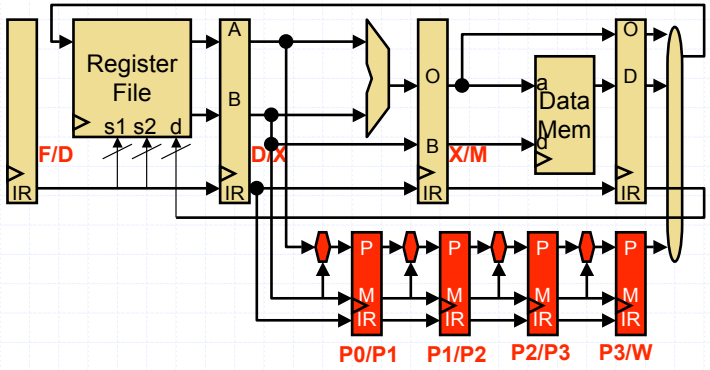
- Multiplier itself is often pipelined, what does this mean?
  - Product/multiplicand register/ALUs/latches replicated
  - Can start different multiply operations in consecutive cycles

What about Stall Logic?



Stall = (OldStallLogic) ||
(F/D.IR.RS1 == P0/P1.IR.RD) || (F/D.IR.RS2 == P0/P1.IR.RD) ||
(F/D.IR.RS1 == P1/P2.IR.RD) || (F/D.IR.RS2 == P1/P2.IR.RD) ||
(F/D.IR.RS1 == P2/P3.IR.RD) || (F/D.IR.RS2 == P2/P3.IR.RD)

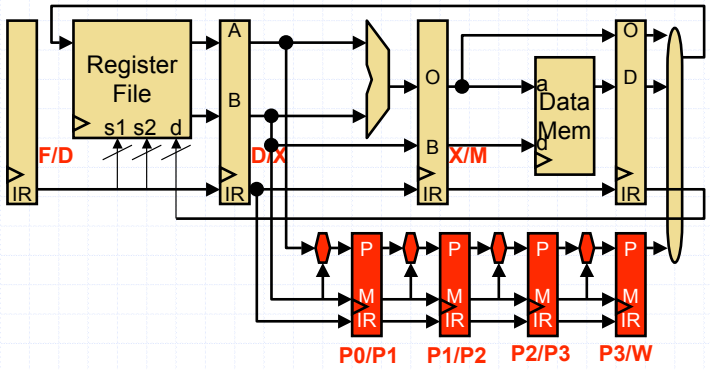
Actually, It's Somewhat Nastier



- What does this do?

Stall = (OldStallLogic) ||
(**F/D.IR.RD != -1 &&**
F/D.IR.OP != MULT && P0/P1.IR.RD != -1)

Actually, We're Not Done



- And what about this?

Stall = (OldStallLogic) ||
(**F/D.IR.RD == D/X.IR.RD && D/X.IR.OP == MULT**)

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

- This is the situation that slide #58 logic tries to avoid
  - Two instructions trying to write regfile in same cycle

	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

- This is the situation on slide #59 tries to avoid
  - 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

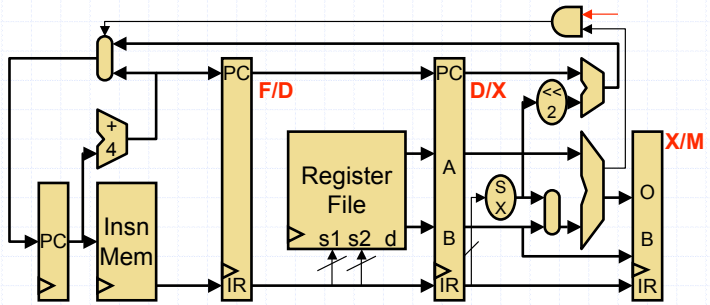
Corrected Pipeline Diagram

- 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

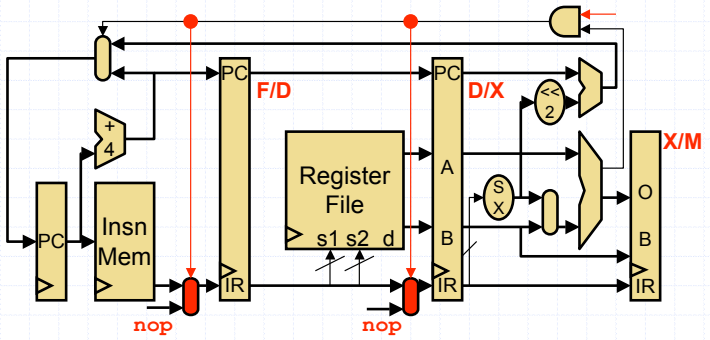
- **Multi-cycle operations complicate pipeline logic**

What About Branches?



- **Control hazards**
  - **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)

Branch Recovery Pipeline Diagram

		1	2	3	4	5	6	7	8	9
	`addi $3,$0,1`	F	D	X	M	W				
	`bnez $3,targ`		F	D	**X**	M	W			
	~~`sw $6,4($7)`~~			F	D					
`targ:`	~~`addi $8,$7,1`~~				F					
`targ:`	`addi $8,$7,1`					**F**	D	X	M	W

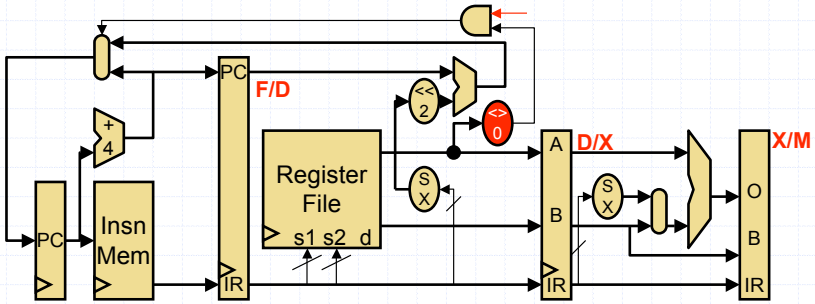
- Convention: don't fill in flushed insns
- Taken branch penalty is 2 cycles

		1	2	3	4	5	6	7	8	9
	`addi $3,$0,1`	F	D	X	M	W				
	`bnez $3,targ`		F	D	**X**	M	W			
`targ:`	`addi $8,$7,1`					**F**	D	X	M	W

Branch Performance

- Back of the envelope calculation
  - **Branch: 20%**, load: 20%, store: 10%, other: 50%
  - **75% of branches are taken**
    - Why not 50%/50%? Loop back edges
- CPI = 1 + 20% * 75% * 2 =
  1 + **0.20*0.75*2** = 1.3
  - **Branches cause 30% slowdown**
    - Even worse with deeper pipelines
  - How do we reduce this penalty?

One MIPS-Specific Way: Fast Branch



- **Fast branch**: can decide at **D**, not X
  - Test must be comparison to *zero or equality*, **no time for ALU**
  - \+ New taken branch penalty is 1
  - – Additional insns (`slt`) for more complex tests
  - – Must bypass to D-stage too

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 P37X & 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.
  - What about condition codes?

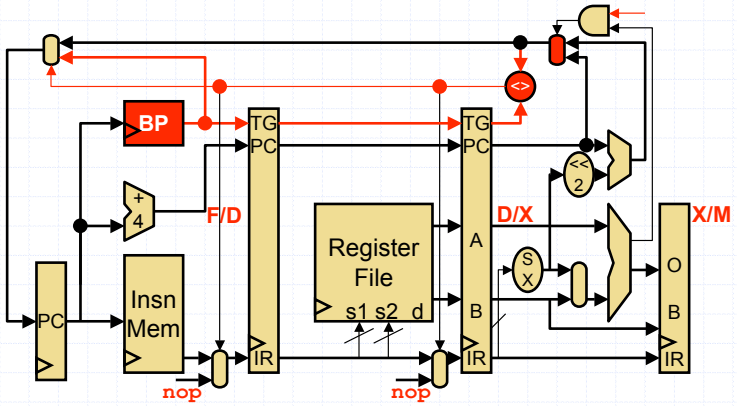
More Generally: Speculative Execution

- 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 Mechanics

- Guess branch target, start fetching at guessed position
  - Doing nothing is implicitly guessing target is PC+4
  - Can actively guess other targets: **dynamic branch prediction**
- Execute branch to verify (check) guess
  - Correct speculation? keep going
  - Mis-speculation? Flush mis-speculated insns
    - Hopefully haven't modified permanent state (Regfile, DMem)
    + Happens naturally in in-order 5-stage pipeline
- "Game" for in-order 5 stage pipeline
  - $\%_{correct}$ = ?
  - Gain = 2 cycles
  + Penalty = 0 cycles → **mis-speculation no worse than stalling**

Dynamic Branch Prediction



- **Dynamic branch prediction**: guess outcome
  - Start fetching from guessed address
  - Flush on **mis-prediction** (notice new recovery circuit)

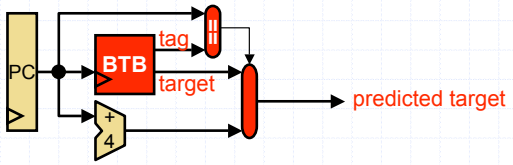
Simple Branch Target Buffer (BTB)

- Big idea: learn from past, predict the future
  - Record the past in a hardware structure
- **Branch target buffer (BTB)**: simplest branch predictor
  - "guess" the future PC based on base behavior
  - "Last time the instruction at address X was followed by address Y"
    - "So, in the future, if address X is fetched, fetch address Y next"
- Operation
  - A small RAM (like a regfile): 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

Branch Target Buffer (continued)

- At Fetch, how does insn know that it's a branch & should read BTB?
  - Answer: it doesn't have to
    - All insns read BTB
    - If insn isn't a branch entry is PC+4
- BTB can't hold all PCs…
  - …what if 2 PCs **alias** (map to same slot)?
  - Answer: doesn't matter
    - Why? BTB contents only used as a guess, can be wrong
- Which PC bits should be used to index BTB?

More Efficient BTB

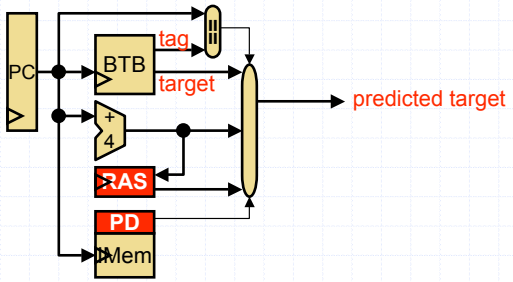


- Naïve BTB is space inefficient
  - Many entries useless entries for non-branches
- More efficient BTB
  - Only explicitly represent taken branches
  - Implement by tagging each entry with PC
  - Update at X if (target != predicted-target)
    - BTB[PC].tag = PC, BTB[PC].target = target
  - All insns access at Fetch in parallel with IMem
    - Predicted-target = (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 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? RAS[TOS++] = PC+4
  - Return? 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
  - Option #1: just wait a cycle
  - Option #2: **pre-decode bits** in Imem, written when first executed

Branch (Direction) Prediction

- BTB uses implicit **branch direction prediction**
  - BTB[PC].tag == PC & BTB[PC].target != PC+4 → "taken" (T)
  - BTB[PC].tag == PC & BTB[PC].target == PC+4 → "not-taken" (N)
  - Implied policy: predict last taken/non-taken
  - \+ Surprisingly effective: captures loop idiom (~75%)
  - – Pathological in several ways: can do much better (~95%)
- **Branch history table (BHT)**: explicit direction predictor
  - RAM, address = PC, data = N/T (0/1), typically untagged
  - Many more entries than BTB
  - Individual conditional branches often unbiased or weakly biased
    - 90%+ one way or the other considered **"biased"**
  - Advanced algorithms use inter-branch correlation, tournaments, etc
  - Still actively researched

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 DIRP 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	**N***	T	T	T	**T**	T	T
	BHR=TTN	N	N	N	N	**N***	T	T	T	**T**	T	T	T
	BHR=TTT	N	N	N	**N**	N	N	N	**N**	N	N	N	N
Outcome	N N N	T	T	T	N	T	T	T	N	T	T	T	N

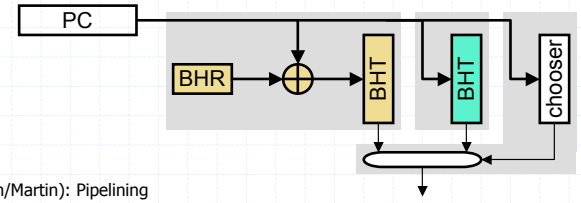
+ No mis-predictions after predictor learns all the relevant patterns

Correlated Predictor

- Design choice I: one **global** BHR or one per PC (**local**)?
  - Each one captures different kinds of patterns
  - Global is better, captures local patterns for tight loop branches
- Design choice II: how many history bits (BHR size)?
  - Tricky one
  + Given unlimited resources, longer BHRs are better, but…
  – BHT utilization decreases
    – Many history patterns are never seen
    – Many branches are history independent (don't care)
    - PC xor BHR allows multiple PCs to dynamically share BHT
    - BHR length < $\log_2$(BHT size)
  – Predictor takes longer to train
  - Typical length: 8–12

Hybrid Predictor

- **Hybrid (tournament) predictor** [McFarling]
  - Attacks correlated predictor BHT utilization problem
  - Idea: combine two predictors
    - **Simple BHT** predicts history independent branches
    - **Correlated predictor** predicts only branches that need history
    - **Chooser** assigns branches to one predictor or the other
    - 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 Fetch
  - Access BHT and BTB in parallel with instruction memory
  - Use BHT result to set next PC to either "PC+4" or "BTB[PC]"
  + No penalty when correctly predicted
  – Need to determine which PCs are conditional branches
    - BTB and/or pre-decode bits mark cond. branches
- 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
- Today's processors usually do some hybrid
  - Quick prediction at fetch, better prediction during decode

Branch Prediction Performance

- Dynamic branch prediction
  - Simple BTB at fetch; branches predicted with 75% accuracy
    - CPI = 1 + (20% * **25%** * 2)= **1.1**
  - More advanced BTB/BHT predictor at fetch: 95% accuracy
    - CPI = 1 + (20% * **5%** * 2) = **1.02**
  - BTB during Fetch, BHT during decode
    - 75% accuracy at fetch, 95% accuracy at decode
    - CPI = 1 + (20% * **25% * 1**) + (20% * **5% * 1**) = **1.06**
- Branch mis-predictions still a big problem though
  - Pipelines are long: typical mis-prediction penalty is 10+ cycles
  - Pipelines have full bypassing: compiler schedules the rest
  - Pipelines are superscalar (later)

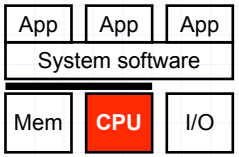
Pipelining And Exceptions

- Pipelining makes exceptions nasty
  - 5 insns in pipeline at once
  - Exception happens, how do you know which insn caused it?
    - Exceptions propagate along pipeline in latches
  - Two exceptions happen, how do you know which one to take first?
    - One belonging to oldest insn
  - When handling exception, have to flush younger insns
    - Piggy-back on branch mis-prediction machinery to do this
  - What about multi-cycle operations?
- Just FYI

Pipeline Depth

- No magic about 5 stages, trend had been to deeper pipelines
  - 486: 5 stages (50+ gate delays / clock)
  - Pentium: 7 stages
  - Pentium II/III: 12 stages
  - Pentium 4: 22 stages (~10 gate delays / clock) **"super-pipelining"**
  - Core1/2: 14 stages
- Increasing **pipeline depth**
  - \+ Increases clock frequency (reduces period)
  - – But decreases IPC (increases CPI)
  - Branch mis-prediction penalty becomes longer
  - Non-bypassed data hazard stalls become longer
  - At some point, CPI losses offset clock gains, question is when?
    - 1GHz Pentium 4 was slower than 800 MHz PentiumIII
  - What was the point? Customers buy frequency, not IPC*frequency

Summary



- Basics of pipelining
  - Pipeline diagrams
- Data hazards
  - Software interlocks/code scheduling
  - Hardware interlocks/stalling
  - Bypassing
  - Multi-cycle operations
- Control hazards
  - Branch prediction