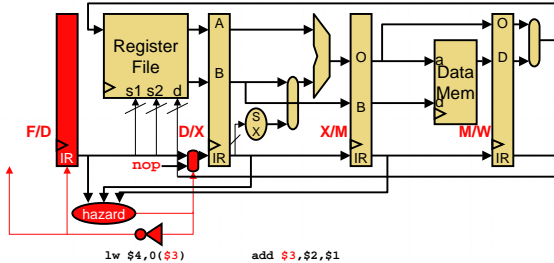


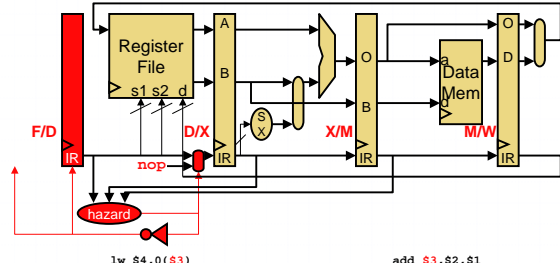
Hardware Interlock Example: cycle 1



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

$$(F/D.IR.RS1 == D/X.IR.RD) \parallel (F/D.IR.RS2 == D/X.IR.RD) \parallel (F/D.IR.RS1 == X/M.IR.RD) \parallel (F/D.IR.RS2 == X/M.IR.RD) = 1$$

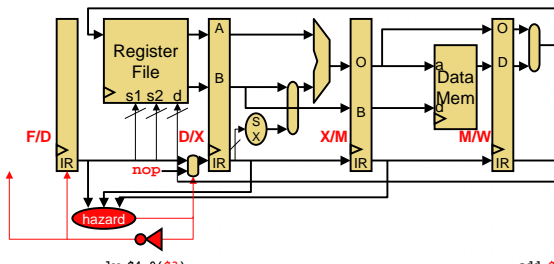
Hardware Interlock Example: cycle 2



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

$$(F/D.IR.RS1 == D/X.IR.RD) \parallel (F/D.IR.RS2 == D/X.IR.RD) \parallel (F/D.IR.RS1 == X/M.IR.RD) \parallel (F/D.IR.RS2 == X/M.IR.RD) = 1$$

Hardware Interlock Example: cycle 3



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

$$(F/D.IR.RS1 == D/X.IR.RD) \parallel (F/D.IR.RS2 == D/X.IR.RD) \parallel (F/D.IR.RS1 == X/M.IR.RD) \parallel (F/D.IR.RS2 == X/M.IR.RD) = 0$$

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
- Distinguished 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 OK (why?)

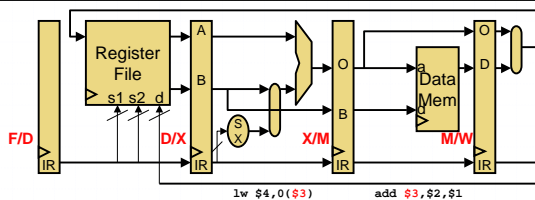
	1	2	3	4	5	6	7	8	9
add \$3,\$2,\$1	F	D	X	M	W				
lw \$4,0(\$3)		F	d*	d*	D	X	M	W	
sw \$6,4(\$7)			F	D	X	M	W		



Hardware Interlock Performance

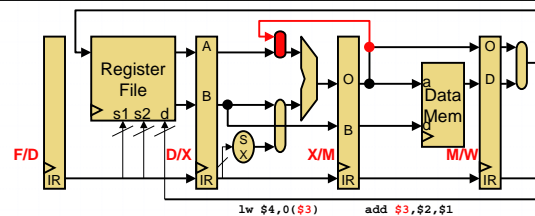
- Same deal
 - Branch: 20%, load: 20%, store: 10%, other: 50%
- Hardware interlocks: same as software interlocks
 - 20% of insns require 1 cycle stall (i.e., insertion of 1 **nop**)
 - 5% of insns require 2 cycle stall (i.e., insertion of 2 **nops**)
- CPI = $1 + 0.20 \cdot 1 + 0.05 \cdot 2 = 1.3$
 - 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



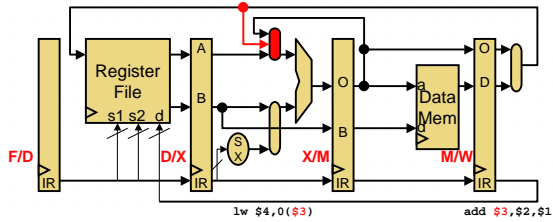
- This situation seems broken
 - lw \$4,0(\$3) has already read \$3 from regfile
 - add \$3,\$2,\$1 hasn't yet written \$3 to regfile
- But fundamentally, everything is still OK
 - lw \$4,0(\$3) hasn't actually used \$3 yet
 - add \$3,\$2,\$1 has already computed \$3

Bypassing



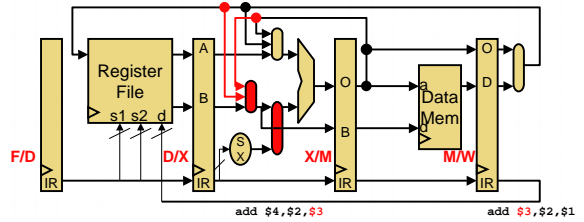
- Bypassing**
 - Reading a value from an intermediate (μ architectural) source
 - Not waiting until it is available from primary source (RegFile)
 - Here, we are bypassing the register file
 - Also called **forwarding**

WX Bypassing



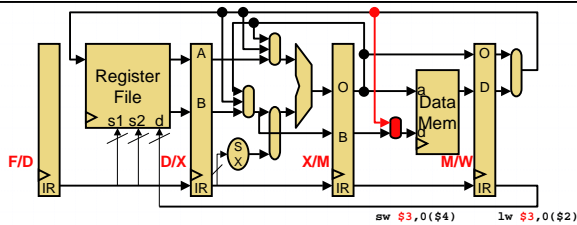
- What about this combination?
 - Add another bypass path and MUX input
 - First one was an **MX** bypass
 - This one is a **WX** bypass

ALUinB Bypassing



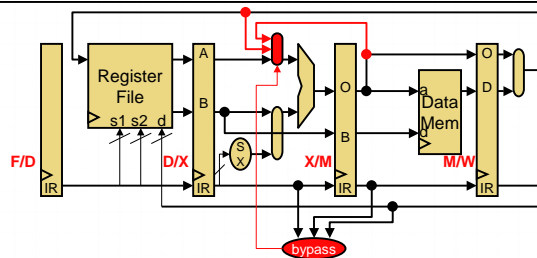
- Can also bypass to ALU input B

WM Bypassing?



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

Bypass Logic

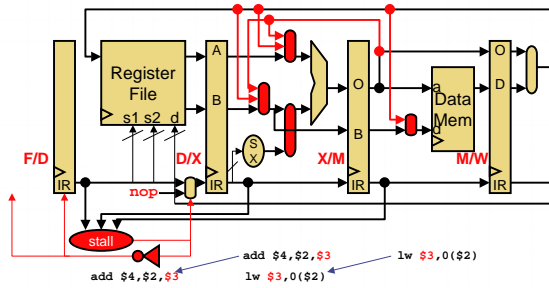


- Each MUX has its own, here it is for MUX ALUinA
 - (D/X.IR.RS1 == X/M.IR.RD) → mux select = 0
 - (D/X.IR.RS1 == M/W.IR.RD) → mux select = 1
 - Else → mux select = 2

Bypass and Stall Logic

- Two separate things
 - Stall logic controls pipeline registers
 - Bypass logic controls muxes
- But complementary
 - For a given data hazard: if can't bypass, must stall
- Slide #42 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))

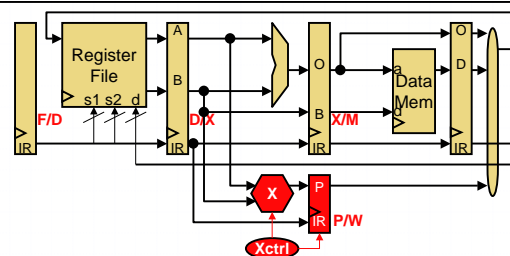
Pipeline Diagram With Bypassing

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

- Sometimes you will see it like this
 - Denotes that stall logic implemented at X stage, rather than D
 - Equivalent, doesn't matter when you stall as long as you do

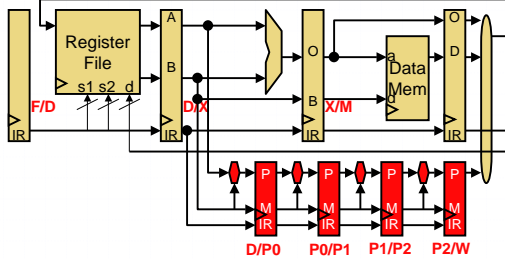
	1	2	3	4	5	6	7	8	9
add \$3,\$2,\$1	F	D	X	M	W				
lw \$4,0(\$3)		F	D	X	M	W			
addi \$6,\$4,1			F	D	d*	X	M	W	

Pipelining and Multi-Cycle Operations



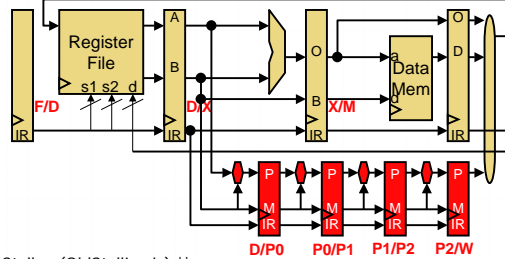
- What if you wanted to add a multi-cycle operation?
 - E.g., 4-cycle multiply
 - P/W: separate output latch connects to W stage
 - Controlled by pipeline control and multiplier FSM

A Pipelined Multiplier



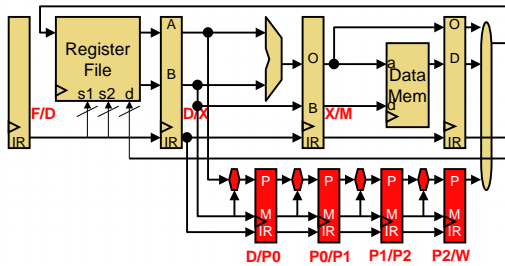
- Multiplier itself is often pipelined: what does this mean?
 - Product/multiplicand register/ALUs/latches replicated
 - Can start different multiply operations in consecutive cycles

What about Stall Logic?



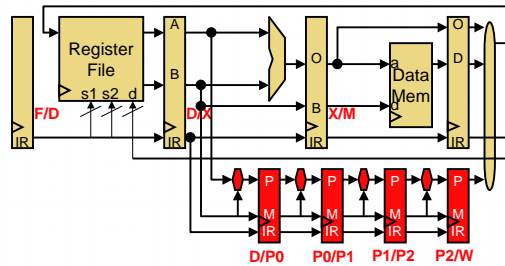
Stall = (OldStallLogic) ||
 (F/D.IR.RS1 == D/P0.IR.RD) || (F/D.IR.RS2 == D/P0.IR.RD) ||
 (F/D.IR.RS1 == P0/P1.IR.RD) || (F/D.IR.RS2 == P0/P1.IR.RD) ||
 (F/D.IR.RS1 == P1/P2.IR.RD) || (F/D.IR.RS2 == P1/P2.IR.RD)

Actually, It's Somewhat Nastier



- What does this do? Hint: think about structural hazards
 Stall = (OldStallLogic) ||
 (F/D.IR.RD != null && P0/P1.IR.RD != null)

Honestly, It's Even Nastier Than That



- And what about this?
 Stall = (OldStallLogic) ||
 (F/D.IR.RD == D/P0.IR.RD) || (F/D.IR.RD ==
 P0/P1.IR.RD)

Pipeline Diagram with Multiplier

	1	2	3	4	5	6	7	8	9
mul \$4,\$3,\$5	F	D	PO	P1	P2	P3	W		
addi \$6,\$4,1		F	d*	d*	d*	D	X	M	W

- This is the situation that slide #50 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	PO	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 that slide #51 logic tries to avoid
 - Mis-ordered writes to the same register
 - Compiler 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	PO	P1	P2	P3	W		
addi \$4,\$1,1		F	D	X	M	W			
...									
...									
add \$10,\$4,\$6					F	D	X	M	W

- Multi-cycle operations complicate pipeline logic**
 - They're not impossible, but they require more complexity