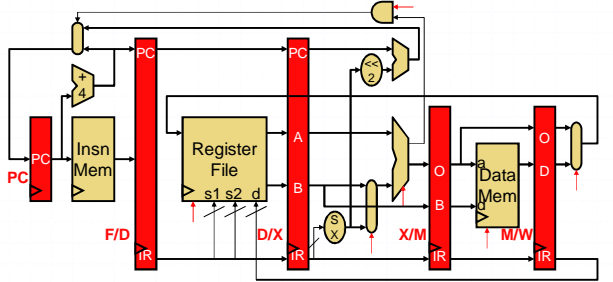


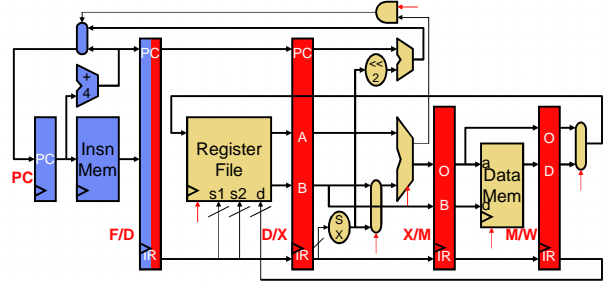
Pipeline Example: Cycle 1



add \$3,\$2,\$1

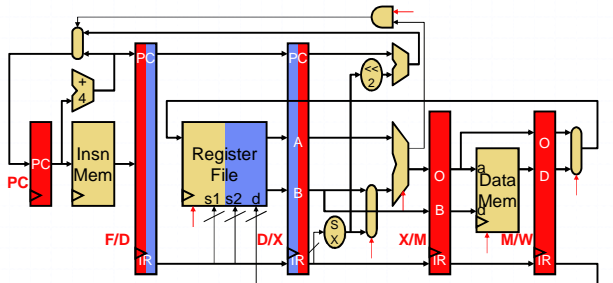
- 3 instructions

Pipeline Example: Cycle 2



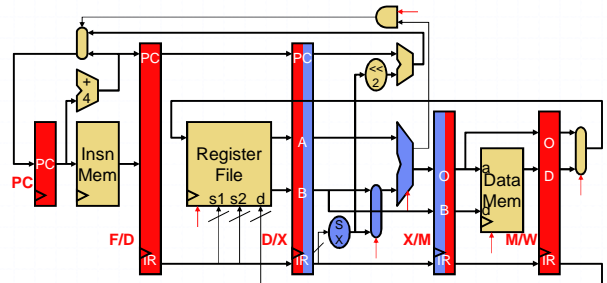
lw \$4,0(\$5) add \$3,\$2,\$1

Pipeline Example: Cycle 3



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

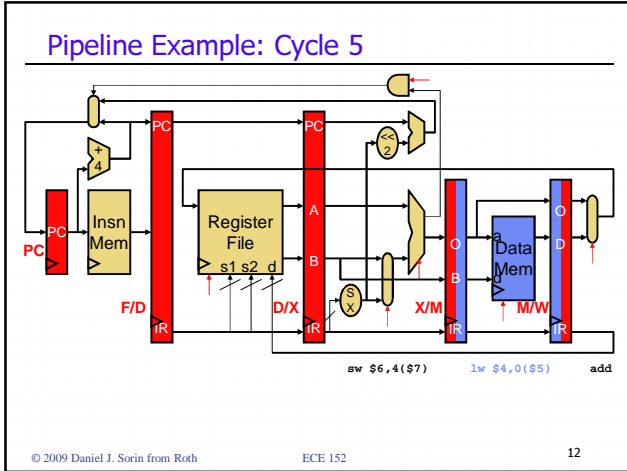
Pipeline Example: Cycle 4



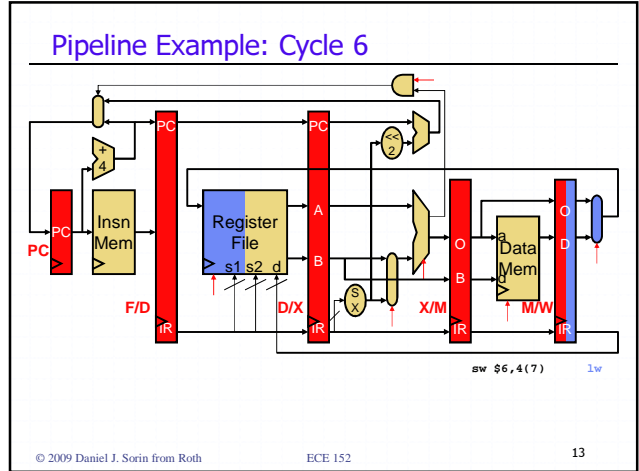
sw \$6,4(\$7) lw \$4,0(\$5) add \$3,\$2,\$1

- 3 instructions

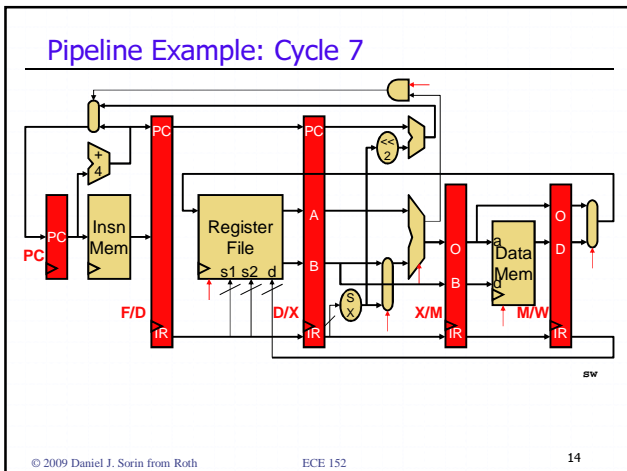
Pipeline Example: Cycle 5



Pipeline Example: Cycle 6



Pipeline Example: Cycle 7



Pipeline Diagram

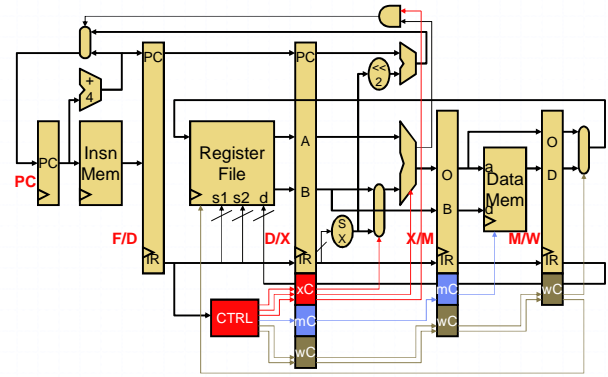
- **Pipeline diagram:** shorthand for what we just saw
 - Across: cycles
 - Down: insns
 - Convention: **X** means `lw $4,0($5)` finishes execute stage and writes into X/M latch at end of cycle 4

	1	2	3	4	5	6	7	8	9
<code>add \$3,\$2,\$1</code>	F	D	X	M	W				
<code>lw \$4,0(\$5)</code>		F	D	X	M	W			
<code>sw \$6,4(\$7)</code>			F	D	X	M	W		

What About Pipelined Control?

- Should it be like single-cycle control?
 - But individual insn signals must be staged
- How many different control units do we need?
 - One for each insn in pipeline?
- Solution: use simple single-cycle control, but pipeline it
 - Single controller
 - **Key idea: pass control signals with instruction through pipeline**

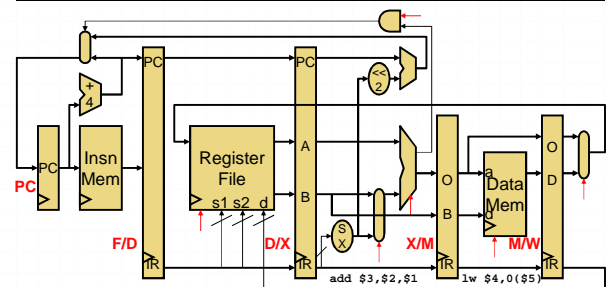
Pipelined Control



Pipeline Performance Calculation

- Single-cycle
 - Clock period = 50ns, CPI = 1
 - Performance = 50ns/insn
- Pipelined
 - Clock period = **12ns (why not 10ns?)**
 - CPI = **1** (each insn takes 5 cycles, but 1 completes each cycle)
 - Performance = **12ns/insn**

Why Does Every Insn Take 5 Cycles?

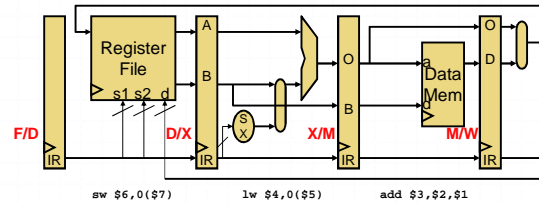


- Why not let `add` skip M and go straight to W?
 - It wouldn't help: peak fetch still only 1 insn per cycle
 - **Structural hazards**: not enough resources per stage for 2 insns

Pipeline Hazards

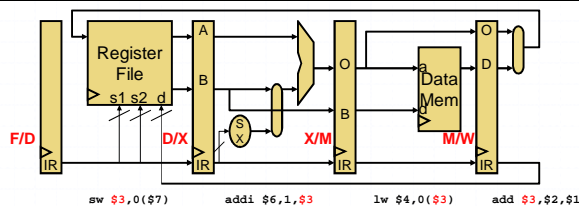
- **Hazard:** condition leads to incorrect execution if not fixed
 - "Fixing" typically increases CPI
 - Three kinds of hazards
- **Structural hazards**
 - Two insns trying to use same circuit at same time
 - E.g., structural hazard on RegFile write port
 - Fix by proper ISA/pipeline design: 3 rules to follow
 - Each insn uses every structure exactly once
 - For at most one cycle
 - Always at same stage relative to F
- **Data hazards** (next)
- **Control hazards** (a little later)

Data Hazards



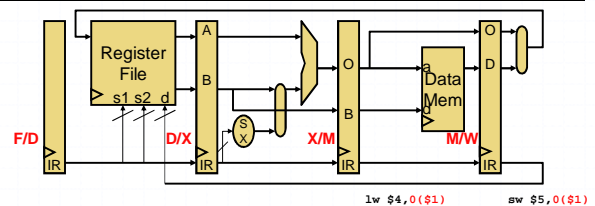
- Let's forget about branches and control for a while
- The sequence of 3 insns we saw earlier executed fine...
 - But it wasn't a real program
 - Real programs have **data dependences**
 - They pass values via registers and memory

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
- One way to enforce this: make sure programs can'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**
- This is called **software interlocks**
 - **MIPS**: Microprocessor w/out Interlocking Pipeline Stages

Software Interlock Example

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

- Can any of last 3 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
```

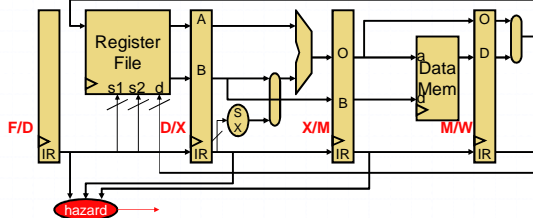
Software Interlock Performance

- Software interlocks
 - Assume 20% of insns require insertion of 1 **nop**
 - Assume 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

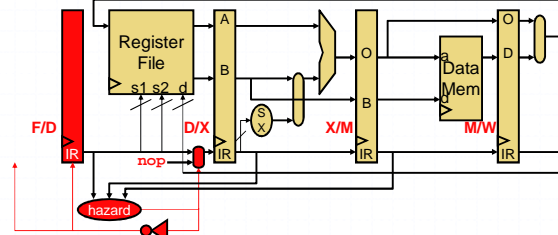


- Compare F/D insn input register names with output register names of older insns in pipeline

Hazard =

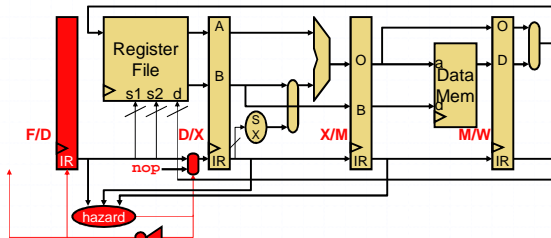
$$(F/D.IR.RS1 == D/X.IR.RD) \parallel (F/D.IR.RS2 == D/X.IR.RD) \parallel (F/D.IR.RS1 == X/M.IR.RD) \parallel (F/D.IR.RS2 == X/M.IR.RD)$$

Fixing Data Hazards



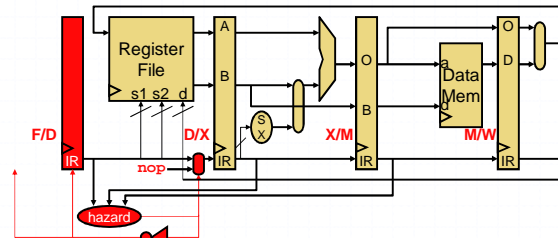
- Prevent F/D insn from reading (advancing) this cycle
 - Write `nop` into D/X.IR (effectively, insert `nop` in hardware)
 - Also clear the datapath control signals
 - Disable F/D latch and PC write enables (why?)
- Re-evaluate situation next cycle

Hardware Interlock Example: cycle 1



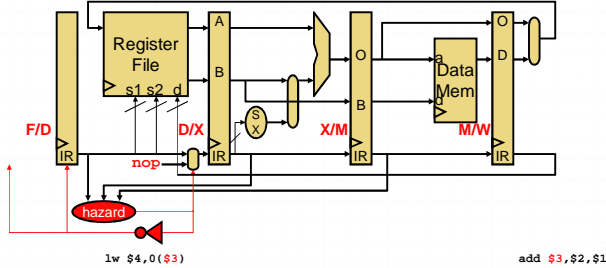
$$(F/D.IR.RS1 == D/X.IR.RD) \parallel (F/D.IR.RS2 == D/X.IR.RD) \parallel (F/D.IR.RS1 == X/M.IR.RD) \parallel (F/D.IR.RS2 == X/M.IR.RD) = 1$$

Hardware Interlock Example: cycle 2



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

Hardware Interlock Example: cycle 3



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

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?)

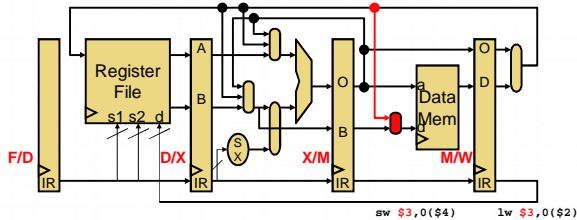
	1	2	3	4	5	6	7	8	9
add \$3,\$2,\$1	F	D	X	M	W				
lw \$4,0(\$3)		F	d*	d*	D	X	M	W	
sw \$6,4(\$7)			F	D	X	M	W		



Hardware Interlock Performance

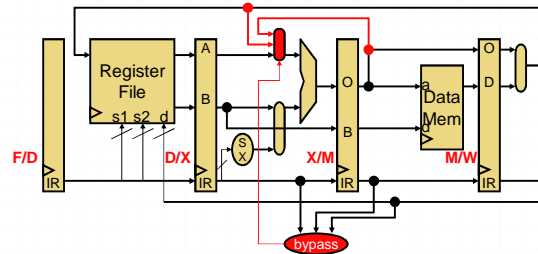
- Hardware interlocks: same as software interlocks
 - 20% of insns require 1 cycle stall (i.e., insertion of 1 `nop`)
 - 5% of insns require 2 cycle stall (i.e., insertion of 2 `nops`)
- $CPI = 1 + 0.20*1 + 0.05*2 = 1.3$
- So, either CPI stays at 1 and #insns increases 30% (software)
- Or, #insns stays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

WM Bypassing?



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

Bypass Logic

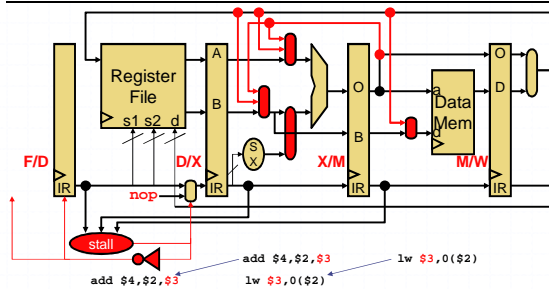


- Each MUX has its own, here it is for MUX ALUinA
 - $(D/X.IR.RS1 == X/M.IR.RD) \rightarrow \text{mux select} = 0$
 - $(D/X.IR.RS1 == M/W.IR.RD) \rightarrow \text{mux select} = 1$
 - Else $\rightarrow \text{mux select} = 2$

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 #40 shows **full bypassing**: all bypasses possible
 - Is stall logic still necessary?

Yes, Load Output to ALU Input



$\text{Stall} = (D/X.IR.OP == \text{LOAD}) \ \&\&$
 $((F/D.IR.RS1 == D/X.IR.RD) \ ||$
 $((F/D.IR.RS2 == D/X.IR.RD) \ \&\& (F/D.IR.OP != \text{STORE}))$

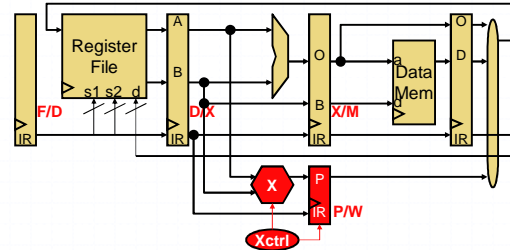
Pipeline Diagram With Bypassing

	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

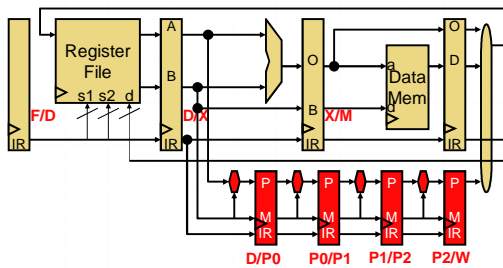
	1	2	3	4	5	6	7	8	9
add \$3,\$2,\$1	F	D	X	M	W				
lw \$4,0(\$3)		F	D	X	M	W			
addi \$6,\$4,1			F	D	d*	X	M	W	

Pipelining and Multi-Cycle Operations



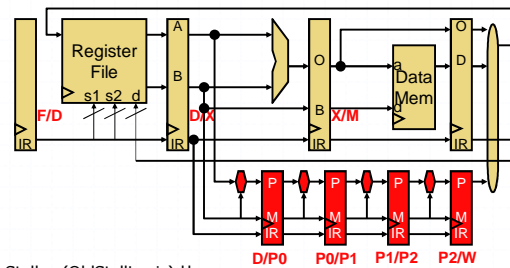
- What if you wanted to add a multi-cycle operation?
 - E.g., 4-cycle multiply
 - P/W**: separate output latch connects to W stage
 - Controlled by pipeline control and multiplier FSM

A Pipelined Multiplier



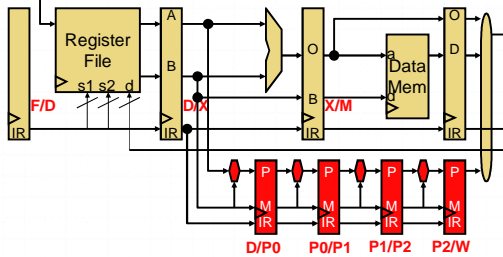
- Multiplier itself is often pipelined: what does this mean?
 - Product/multiplicand register/ALUs/latches replicated
 - Can start different multiply operations in consecutive cycles

What about Stall Logic?



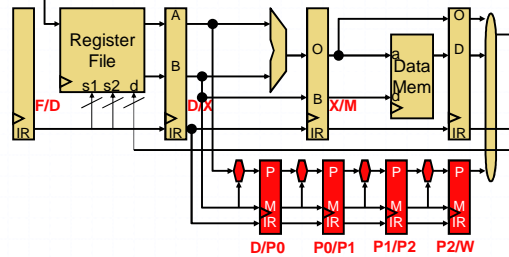
Stall = (OldStallLogic) ||
 (F/D.IR.RS1 == D/P0.IR.RD) || (F/D.IR.RS2 == D/P0.IR.RD) ||
 (F/D.IR.RS1 == P0/P1.IR.RD) || (F/D.IR.RS2 == P0/P1.IR.RD) ||
 (F/D.IR.RS1 == P1/P2.IR.RD) || (F/D.IR.RS2 == P1/P2.IR.RD)

Actually, It's Somewhat Nastier



- What does this do? Hint: think about structural hazards
 $\text{Stall} = (\text{OldStallLogic}) \parallel (\text{F/D.IR.RD} \neq \text{null} \ \&\& \ \text{P0/P1.IR.RD} \neq \text{null})$

Honestly, It's Even Nastier Than That



- And what about this? ("WAR" hazard)
 $\text{Stall} = (\text{OldStallLogic}) \parallel (\text{F/D.IR.RD} == \text{D/P0.IR.RD}) \parallel (\text{F/D.IR.RD} == \text{P0/P1.IR.RD})$

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 #48 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 that slide #49 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	P0	P1	P2	P3	W		
addi \$4,\$1,1		F	D	X	M	W			
...									
...									
add \$10,\$4,\$6					F	D	X	M	W

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