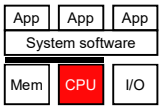
CSE 560
Computer Systems Architecture

Pipelining

Washington University in St. Louis

1

This Unit: (Scalar In-Order) Pipelining



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

Washington University in St. Louis 4

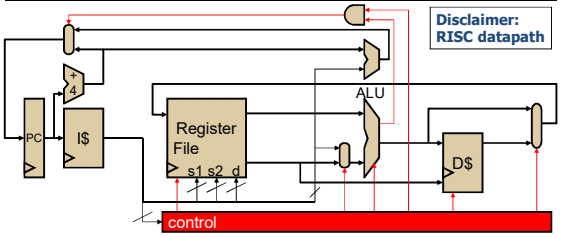
4

Datapath Background

Washington University in St. Louis 5

5

Datapath and Control

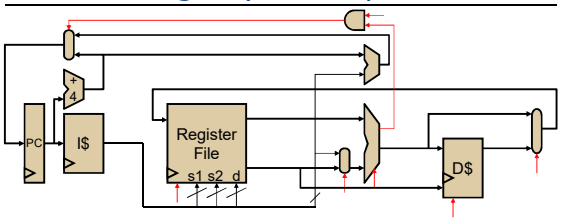


- **Datapath**: implements execute portion of fetch/exec. loop
  - Functional units (ALUs), registers, memory interface
- **Control**: implements decode portion of fetch/execute loop
  - Mux selectors, write enable signals regulate flow of data in datapath
  - Part of decode involves translating insn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

Washington University in St. Louis 6

6

Single-Cycle Datapath



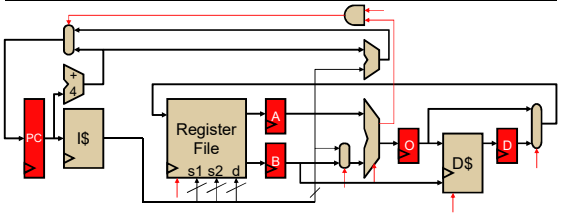
Single-cycle datapath: true "atomic" fetch/execute loop
- Fetch, decode, execute one complete instruction every cycle
- **"Hardwired control"**: opcode to control signals ROM

\+ Low CPI: 1 by definition
– Long clock period: to accommodate slowest instruction

Washington University in St. Louis 7

7

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)

Washington University in St. Louis 8

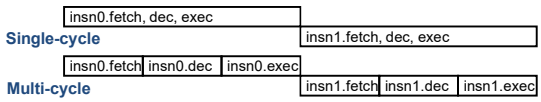
8

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%x3)+(20%x5)+(60%x4) = 4
    - Why is clock period 11ns and not 10ns?
  - Performance = **44ns/insn**
- **Aside:** CISC makes perfect sense in multi-cycle datapath

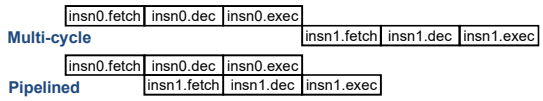
Pipelining Basics

Latency versus Throughput



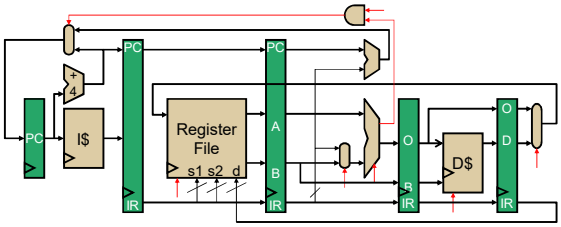
- Can we have both low CPI and short clock period?
  - Not if datapath executes only one insn at a time
- Latency vs. Throughput
  - – Latency: no good way to make a single insn go faster
  - \+ **Throughput**: luckily, single insn latency not so important
    - Goal is to make programs, not individual insns, go faster
    - Programs contain billions of insns
  - Key: **exploit inter-insn parallelism**

Pipelining



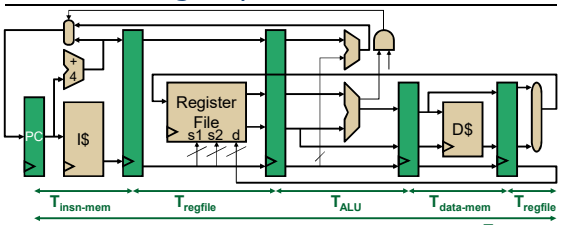
- Important performance technique
  - **Improves insn throughput rather instruction latency**
- Begin with multi-cycle design
  - One insn advances from stage 1 to 2, next insn enters stage 1
  - Form of parallelism: "insn-stage parallelism"
  - Maintains illusion of sequential fetch/execute loop
  - 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

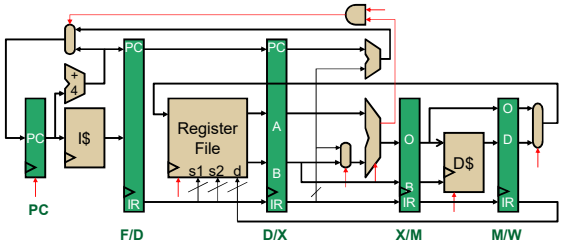
Five Stage Pipeline Performance



Pipelining: cut datapath into N stages (here five)

- 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
- Individual insn latency increases (pipeline overhead), ok

Pipeline Terminology



- Five stage: **F**etch, **D**ecode, e**X**ecute, **M**emory, **W**riteback
- Latches (pipeline registers) named by stages they separate
  - **PC**, **F/D**, **D/X**, **X/M**, **M/W**

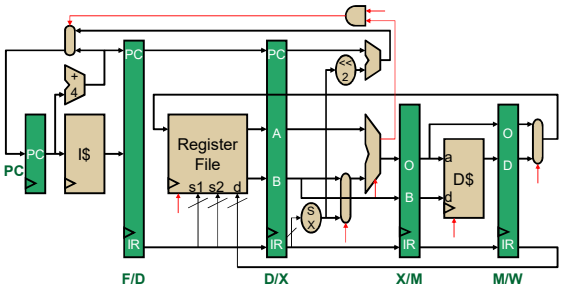
15

More Terminology & Foreshadowing

- **Scalar pipeline**: one insn per stage per cycle
  - Alternative: "superscalar", *e.g.*, 4-wide (later)
- **In-order pipeline**: insns enter execute stage in order
  - Alternative: "out-of-order" (OoO) (later)
- **Pipeline depth**: number of pipeline stages
  - Nothing magical about five (Pentium 4 had 22 stages!)
  - Trend: deeper until Pentium 4, then pulled back a bit

16

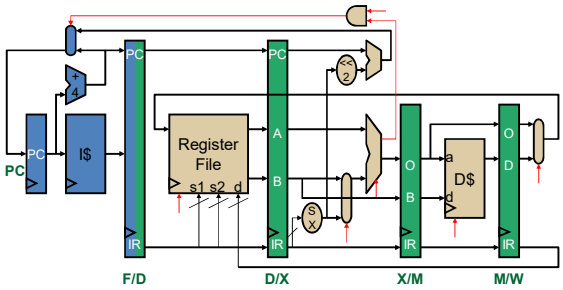
Pipeline Example: Cycle 1




```
add $3<-$2,$1
```


17

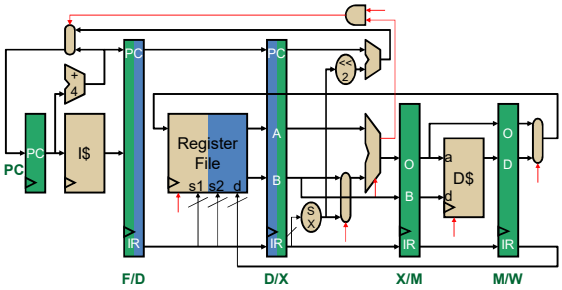
Pipeline Example: Cycle 2




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


18

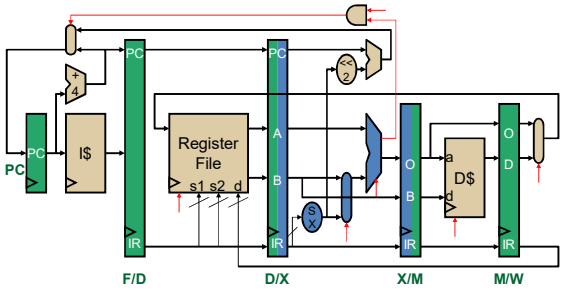
Pipeline Example: Cycle 3




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


19

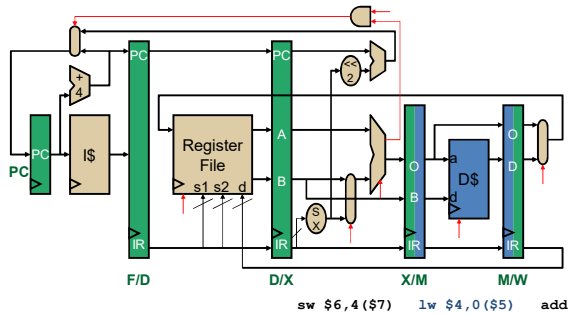
Pipeline Example: Cycle 4




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

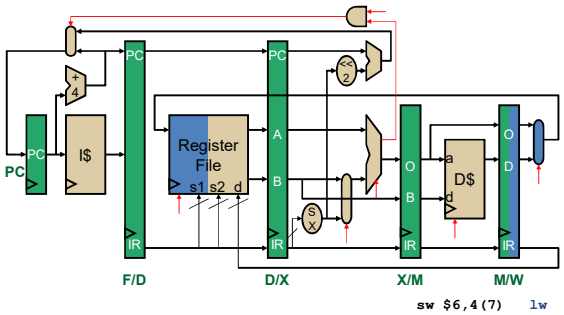

20

Pipeline Example: Cycle 5



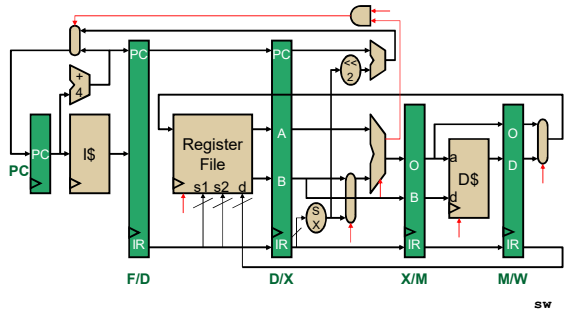
21

Pipeline Example: Cycle 6



22

Pipeline Example: Cycle 7



23

Pipeline Diagram

Pipeline diagram: shorthand for what we just saw

- Convention: **X** means `lw $4,0($5)` finishes execute stage and writes into X/M latch at end of cycle 4

Cycles →

Instructions ↓

	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		

24

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%x3)+(20%x5)+(60%x4) = 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

25

Clock Period of a Pipelined Processor

$Delay_{dp}$ = time it takes to travel through original datapath

N_{ps} = number of pipeline stages

Pipeline Clock Period > $Delay_{dp}$ / N_{ps}

- 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

26

CPI Calculation: Accounting for Stalls

Why is Pipelined CPI > 1 ?

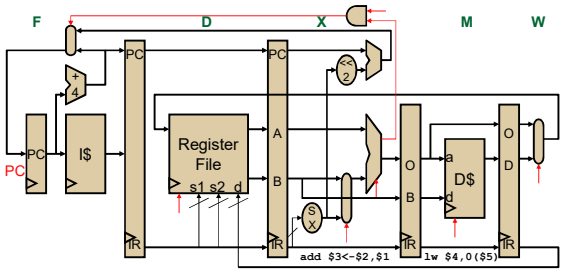
- CPI for scalar in-order pipeline is 1 **+ stall penalties**
- Stalls used to resolve hazards
  - **Hazard**: condition that jeopardizes sequential illusion
  - **Stall**: pipeline delay introduced to restore sequential illusion
- Calculating pipeline CPI
  - **Frequency of stall** x **stall cycles**
  - Penalties add (stalls generally don't overlap in in-order pipelines)
  - 1 + stall-freq$_1$ x stall-cyc$_1$ + stall-freq$_2$ x stall-cyc$_2$ + …
- Correctness/performance/make common case fast (MCCF)
  - Long penalties OK if rare, e.g., 1 + 0.01 x 10 = 1.1
  - Stalls 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**: 1 insn affects whether another executes at all
  - *Not a bad thing*, programs would be boring otherwise
  - 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?
  - It wouldn't help: peak fetch still only 1 insn per cycle
  - **Structural hazards**: who gets the register file write port?

Structural Hazards

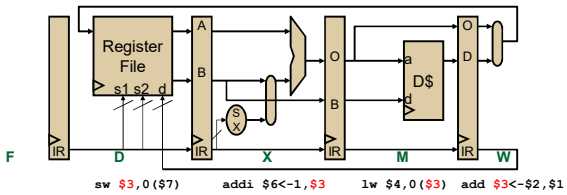
- **Structural hazards**
  - Two insns trying to use same circuit at same time
    - E.g., structural hazard on register file write port
- **To fix structural hazards**: proper ISA/pipeline design
  - Each insn uses every structure exactly once
  - For at most one cycle
  - Always at same stage relative 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	

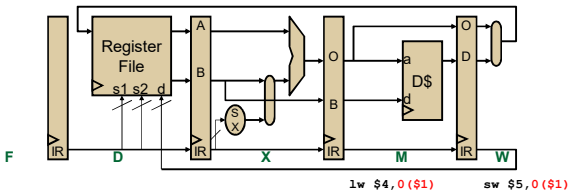
- **Structural hazard**: resource needed twice in one cycle
  - Example: unified instruction & data memories (caches)
  - Solutions:
    - Separate instruction/data memories (caches)
    - Have cache allow 2 accesses per cycle (slow, expensive)
    - Stall pipeline

Data Hazards



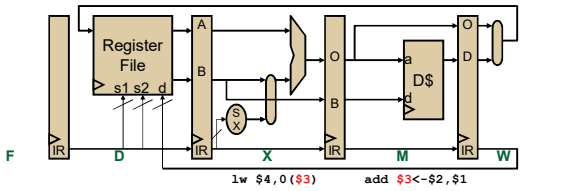
- Would these instructions execute correctly on this pipeline?
- Which instructions execute with correct inputs?
  - `add` writes result into `$3` in current cycle
  - – `lw` read `$3` two cycles ago → got wrong value
  - – `addi` read `$3` one cycle ago → got wrong value
  - `sw` reads `$3` this cycle → maybe (depends on register file)

Memory Data Hazards



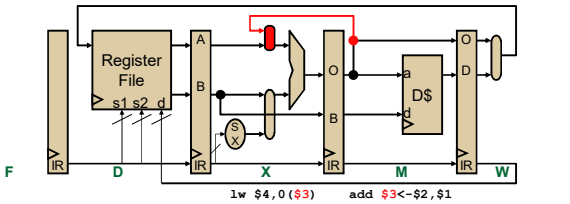
- Are memory data hazards a problem for this pipeline? No
  - `lw` following `sw` to same address in next cycle, gets right value
  - Why? D$ read/write always take place in same stage
- Data hazards through registers? Yes (previous slide)
  - Occur because register write is three stages after register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

Observation!



- *Technically*, we have a problem:
  - `lw $4,0($3)` has already read `$3` from regfile
  - `add $3<-$2,$1` hasn't yet written `$3` to regfile
- Fundamentally, this *should work*
  - `lw $4,0($3)` hasn't actually used `$3` yet
  - `add $3<-$2,$1` has already computed `$3`

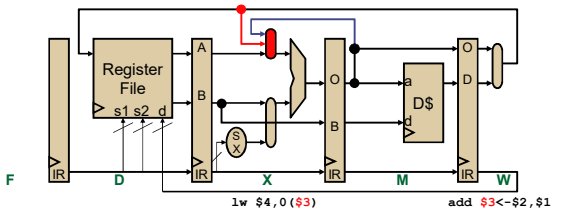
Reducing Data Hazards: Bypassing



Bypassing

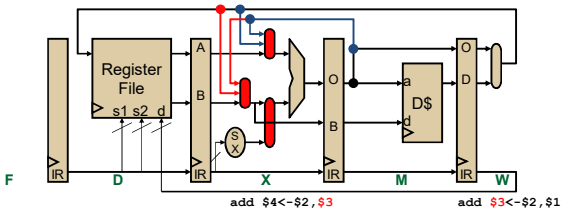
- Reading a value from an intermediate (μarchitectural) source
- Not waiting until it is available from primary source
- Here, we bypass the register file
- Also called **forwarding**

WX Bypassing



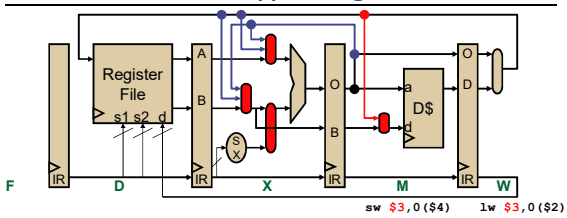
- What about this combination?
  - Add another bypass path and MUX (multiplexor) input
  - First one was an **MX** bypass
  - This one is a **WX** bypass

ALUinB Bypassing



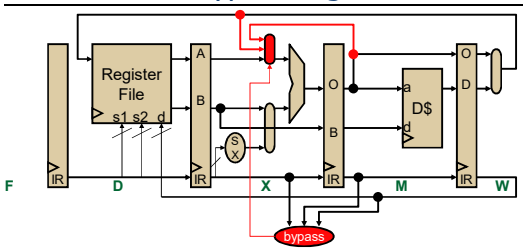
- Can also bypass to ALU input B

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 logic; 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
```

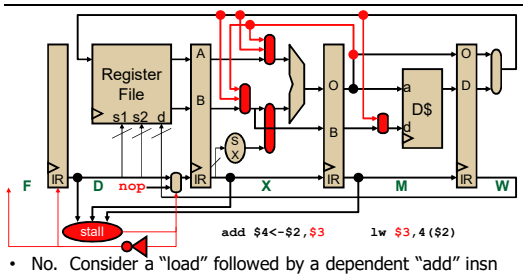
  - Example: full bypassing, use WX bypass

```
                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

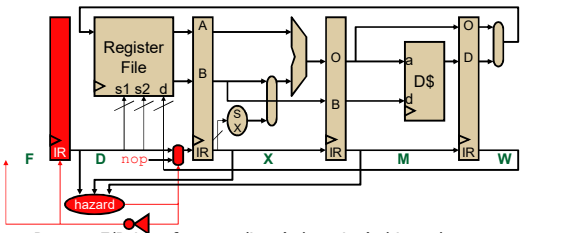
```
                1  2  3  4  5  6  7  8  9  10
add r2,r3→r1    F  D  X  M  W
?                  F  D  X  M  W
```


Have We Prevented All Data Hazards?



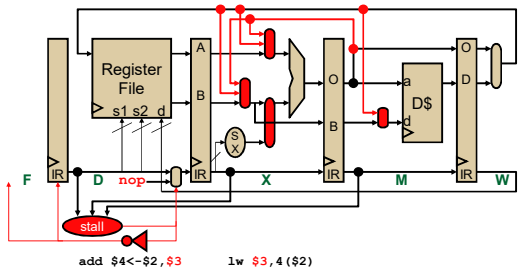
- No. Consider a "load" followed by a dependent "add" insn
- Bypassing alone isn't sufficient!
- Hardware solution: detect this situation and inject a stall cycle
- Software solution: ensure compiler doesn't generate such code

Stalling to Avoid Data Hazards



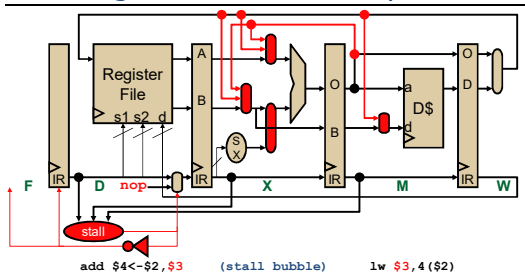
- Prevent F/D insn from reading (advancing) this cycle
  - Write **nop** into D/X.IR (effectively, insert **nop** in hardware)
  - Also reset (clear) the datapath control signals
  - Disable F/D latch and PC write enables (why?)
- Re-evaluate situation next cycle

Stalling on Load-To-Use Dependences



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))

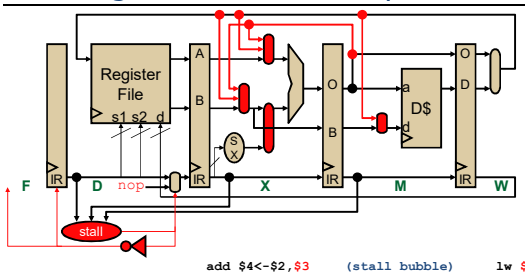
Stalling on Load-To-Use Dependences




```
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 x 20% x 50%) = **1.1**

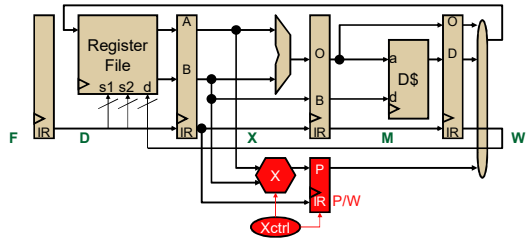
Reducing Load-Use Stall Frequency

	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

- **d*** = data hazard
- Use compiler scheduling to reduce load-use stall frequency

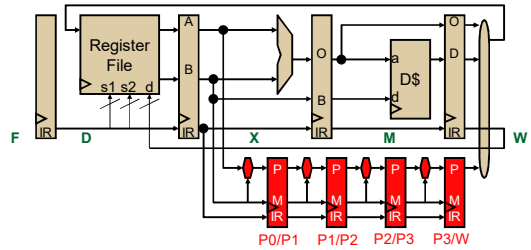
	1	2	3	4	5	6	7	8	9
`add $3<-$2,$1`	F	D	X	M	W				
`lw $4,4($3)`		F	D	X	M	W			
`sub $8<-$3,$1`			F	D	X	M	W		
`addi $6<-$4,1`				F	D	X	M	W	

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 a new multiply operation every cycle

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**		

Washington University in St.Louis

51

51

More Multiplier Nasties

- What about…
  - Mis-ordered register writes
  - SW 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
  - Frequency irrelevant: must be correct no matter how rare

Washington University in St.Louis

52

52

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

Washington University in St.Louis

53

53

Pipelined Functional Units

- Almost all multi-cycle functional units are pipelined
  - Each operation takes N cycles
  - Can initiate a new (independent) operation every cycle
  - Requires internal latching and some hardware replication
  - \+ Cheaper than multiple (non-pipelined) units

	1	2	3	4	5	6	7	8	9	10	11
`mulf f0 f1,f2`	F	D	E1	E2	E3	E4	W				
`mulf f3 f4,f5`		F	D	E1	E2	E3	E4	W			

- Exception: int/FP divide: difficult to pipeline; 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	**s***	**s***	**s***	D	E/	E/	E/	E/	W

s* = structural hazard, two insns need same structure
- ISAs and pipelines designed minimize these
- Canonical example: all insns go through M stage

Washington University in St.Louis

54

54