National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 8: Pipelining

CS10014 Computer Organization

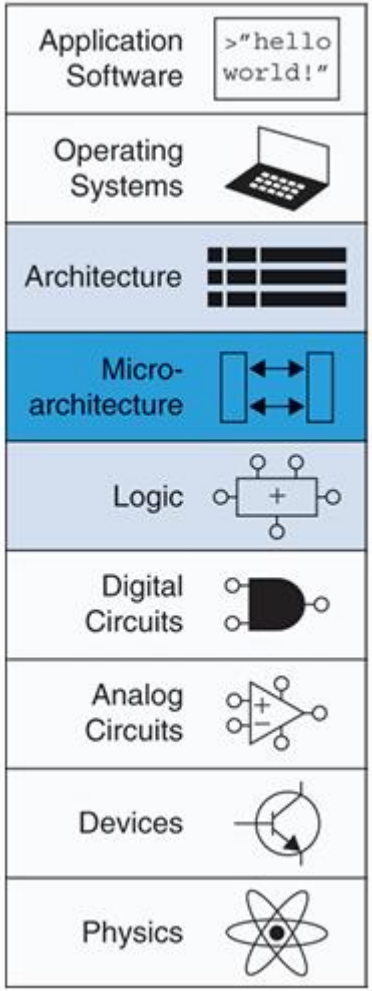
Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao University

Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - CS 252 at UC Berkeley
    - https://people.eecs.berkeley.edu/~culler/courses/cs252-s05/
  - CSCE 513 at University of South Carolina
    - https://passlab.github.io/CSCE513/

Outline

- Pipelining Execution
- Pipelining Hazard
  - Structural Hazard
  - Data Hazard
  - Control Hazard



Single-Cycle Processor

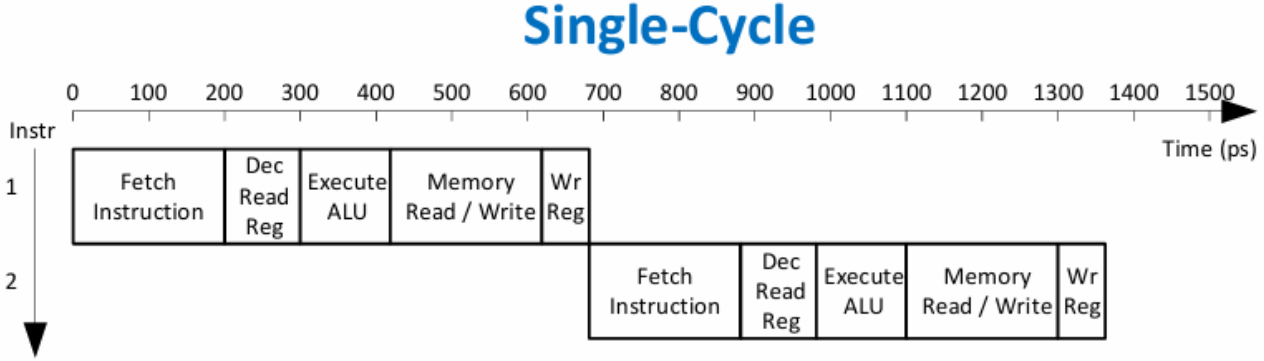
- Estimate the clock rate (frequency) of our single-cycle processor
  - 1 cycle per instruction
  - lw is the most demanding instruction
  - The max clock frequency = 1/800 ps = 1.25 GHz

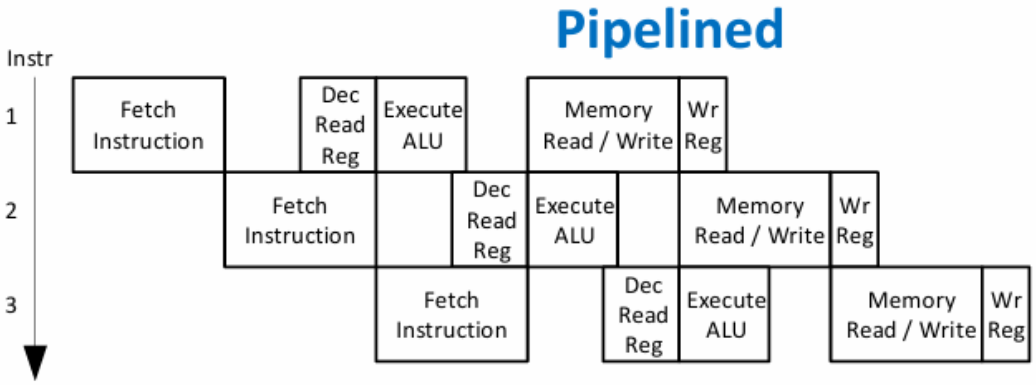
Instr	IF = 200ps	ID = 100ps	ALU = 200ps	MEM=200ps	WB = 100ps	Total
add	X	X	X		X	600ps
beq	X	X	X			500ps
jal	X	X	X		X	600ps
lw	X	X	X	X	X	800ps
sw	X	X	X	X		700ps

Pipelined RISC-V Processor

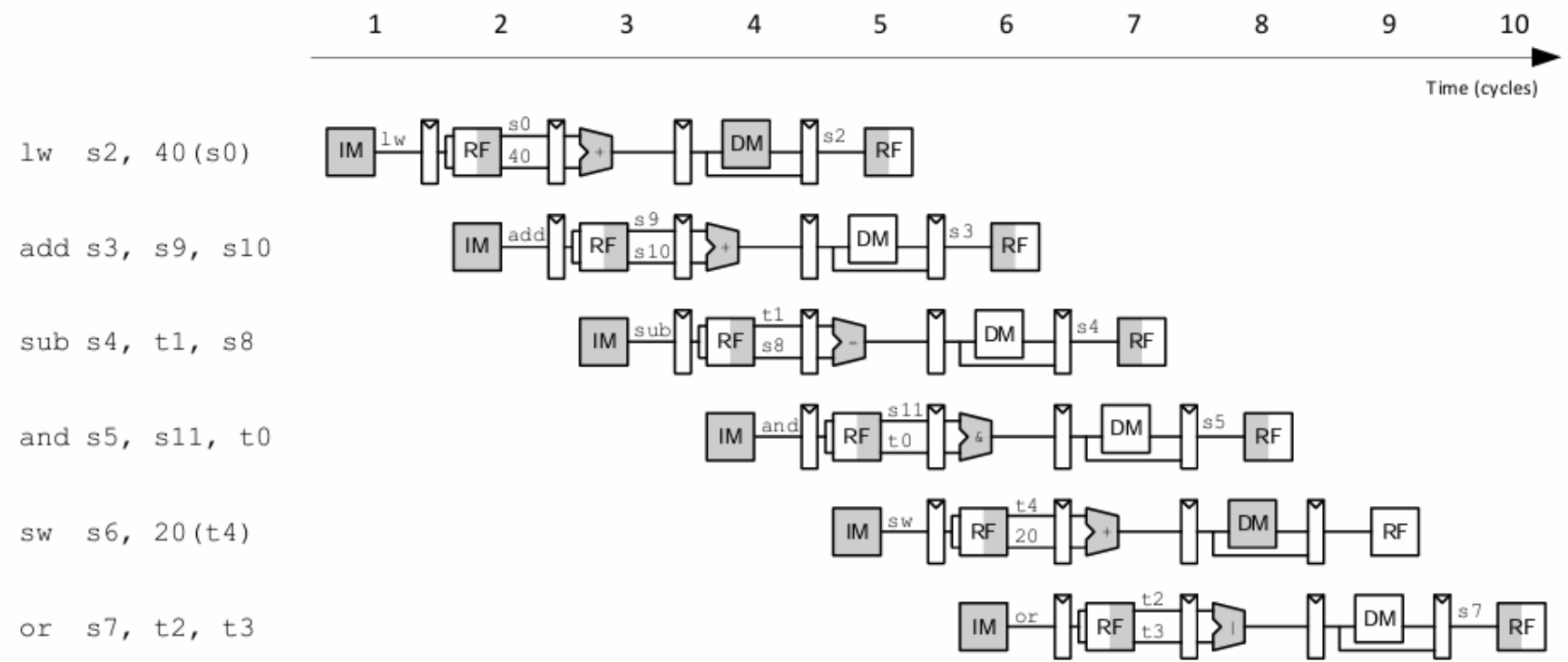
- **Temporal parallelism**
- Divide single-cycle processor into **5 stages**:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add **pipeline registers** between stages

Single-Cycle vs. Pipelined RISC-V Processor



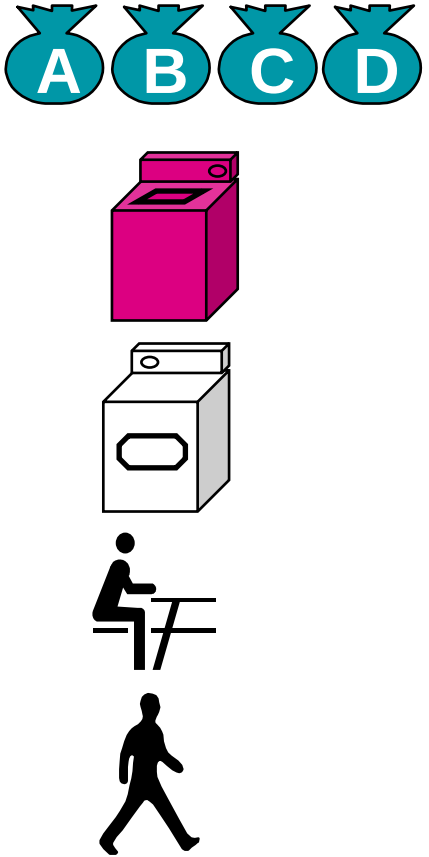


Pipelined RISC-V Abstraction



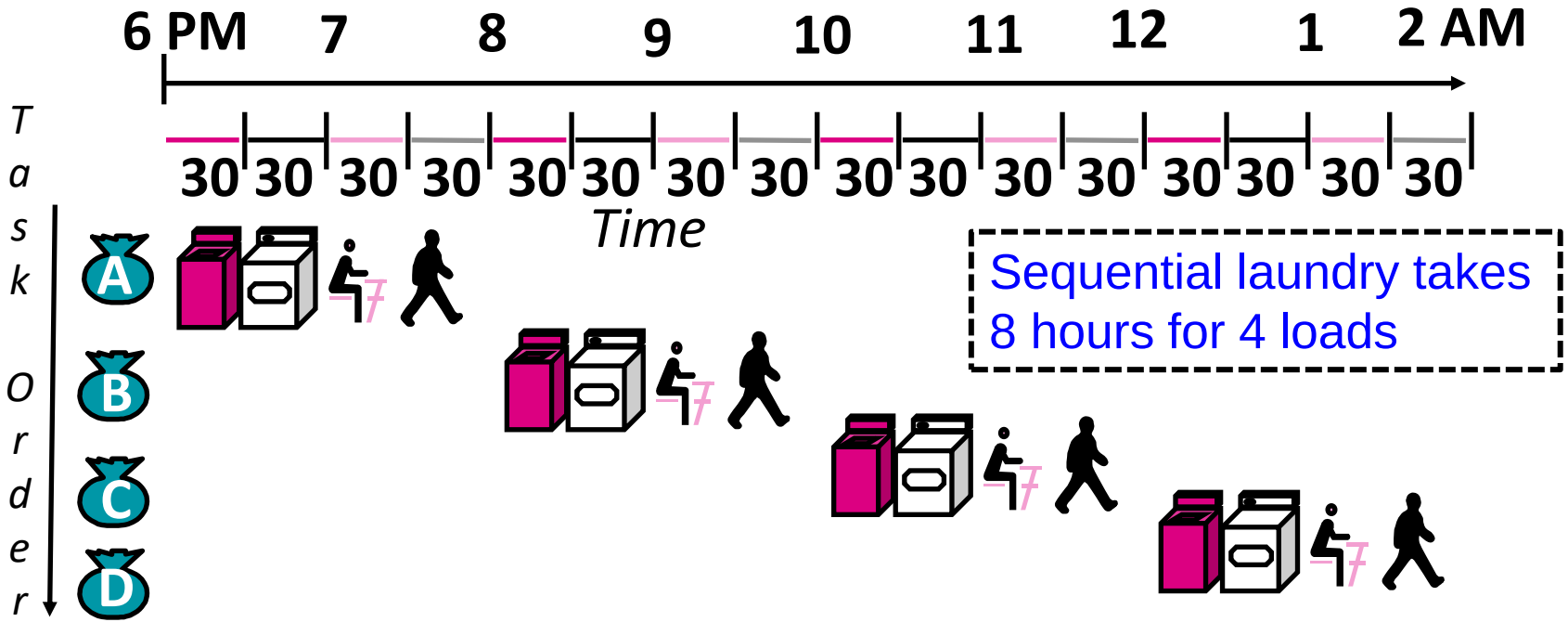
Pipelining

- Ann, Brian, Cathy, Dave
  each has one load of clothes to wash, dry, fold, and put away
  - Washer takes 30 minutes
  - Dryer takes 30 minutes
  - “Folder” takes 30 minutes
  - “Stasher” takes 30 minutes to put clothes into drawers



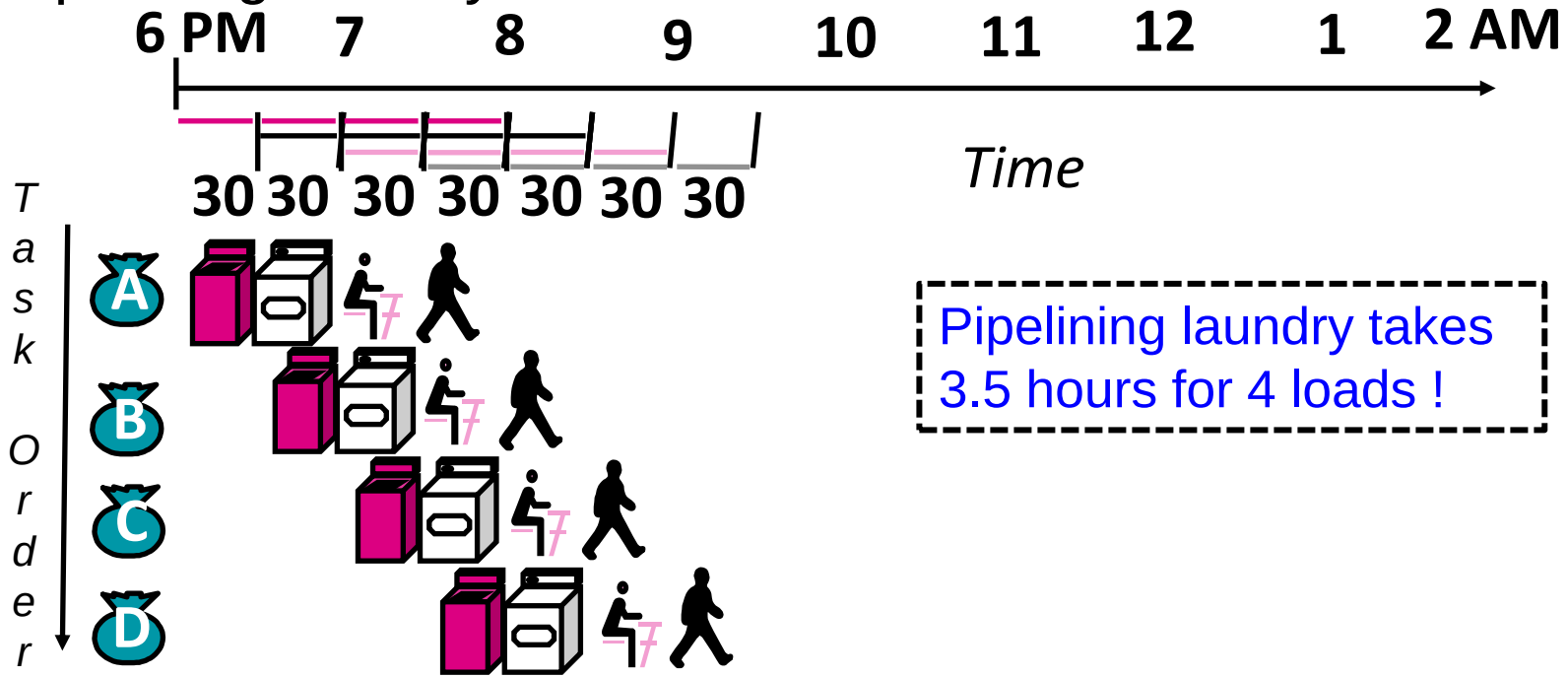
Pipelining

- Sequential Laundry



Pipelining

- Pipelining laundry



Pipelining laundry takes
3.5 hours for 4 loads !

Pipelining

- Pipelining doesn't help latency of single task, it helps the throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = number of pipelining stages
  - Pipelining rate limited by slowest pipeline stage
  - Unbalanced lengths of pipe stages reduce speedup

Pipelining Execution

- Steps in Executing RISC-V
  - IFtch: Instruction fetch, increment PC
  - Dcd: Instruction decode, read registers
  - Execute (Exec)
    - Mem-ref: Calculate Address
    - Arith-log: Perform Operation
  - Mem
    - Load: Read data from memory
    - Store: Write data to memory
  - WB: Write data back to register

Pipelining Execution

- Throughput = # instructions / time

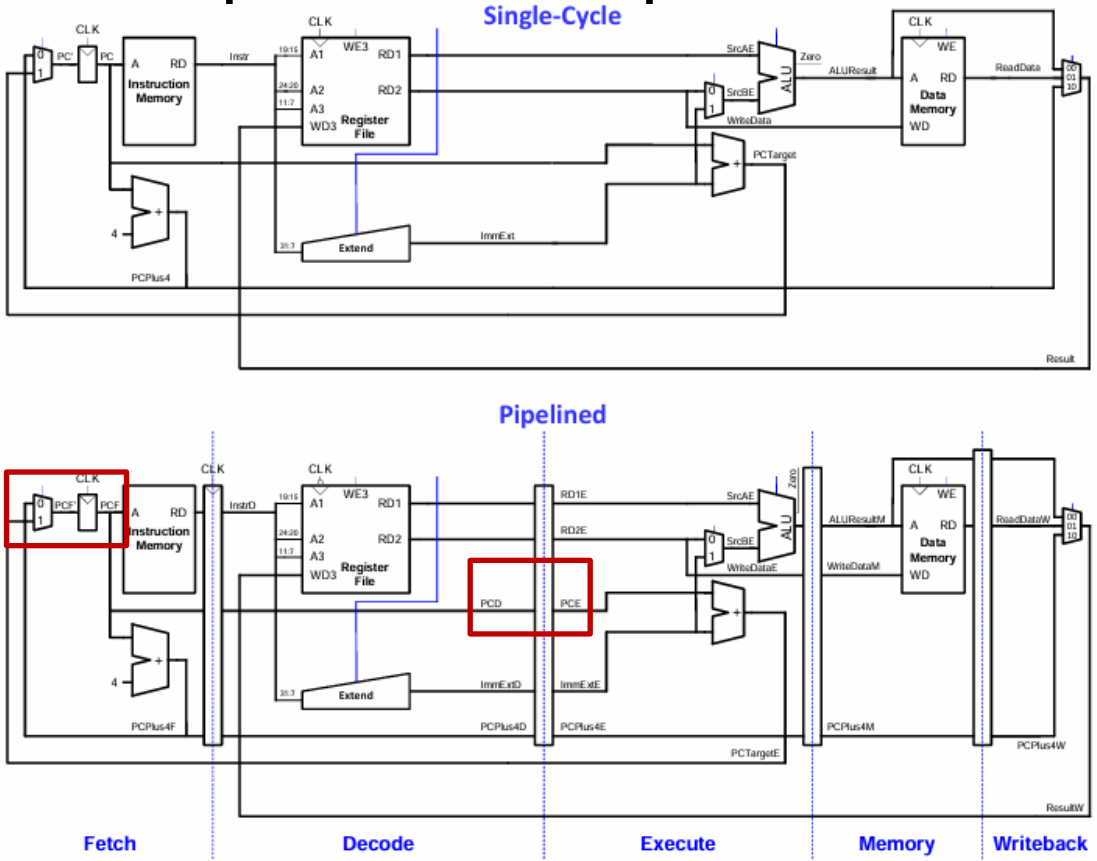
	Single Cycle	Pipelined
Timing of each stage	t_{stage} = 200, 100, 200, 200, 100 ps (Reg access stages **ID**, **WB** only 100 ps)	$t_{stage} = 200$ ps All stages same length
Instruction time (Latency)	$t_{instruction}$ = 800 ps	$t_{instruction} = 5 \cdot t_{cycle}$ = *1000 ps*
Clock cycle time, t_{cycle} Clock rate, $f_s = 1/t_{cycle}$	$t_{cycle} = t_{instruction}$ 800 = ps f_s 800/1 = ps = 1.25 GHz	$t_{cycle} = t_{stage}$ 200 = ps f_s 200/1 = ps = 5 GHz
CPI (Cycles Per Instruction)	~1 (ideal)	~1 (ideal) <1 (actual)
Relative throughput gain	1 x	**4 x**

Pipelining Execution

- The delay time of each pipeline stage
  - Memory access: 2 ns
  - ALU operation: 2 ns
  - Register file read/write: 1 ns
- **Single-cycle processor**
  - lw: IF + Read Reg + ALU + Memory + Write Reg
    = 2 + 1 + 2 + 2 + 1 = 8 ns
  - add: IF + Read Reg + ALU + Write Reg = 6 ns
- **Pipelined Execution**
  - Max (IF, Read Reg, ALU, Memory, Write Reg) = 2ns

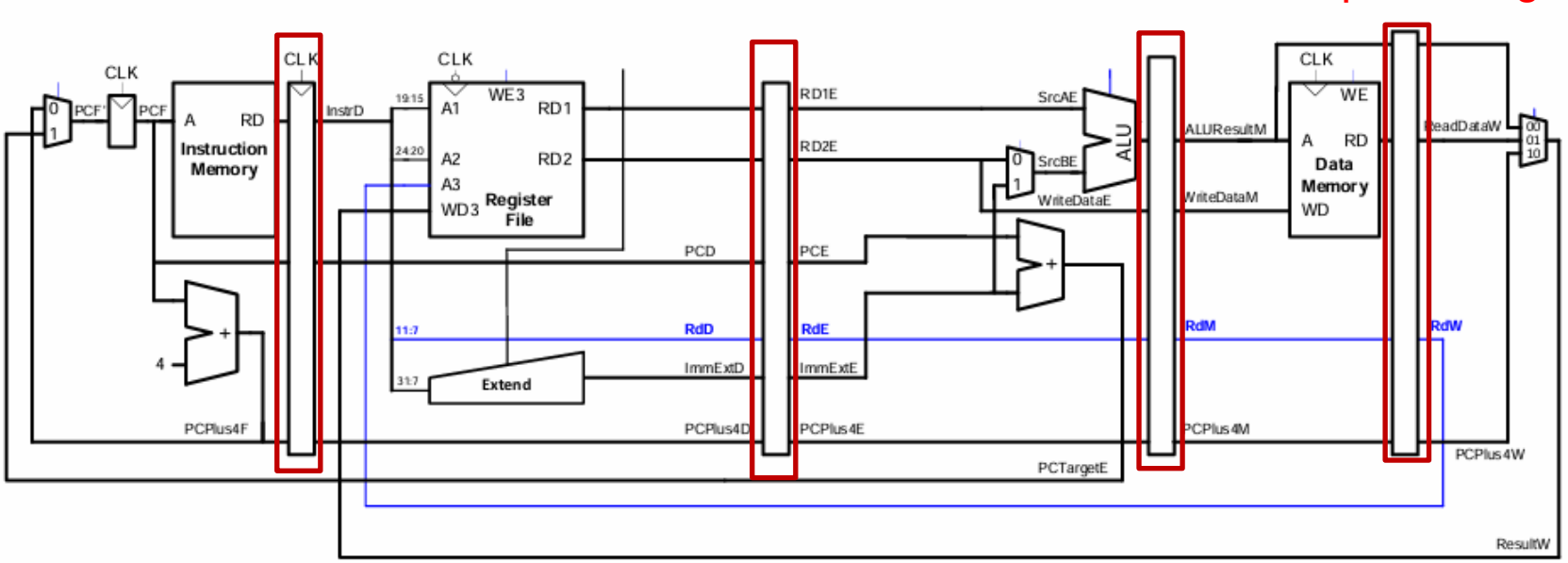
Single-Cycle & Pipelined Datapaths

Signals in Pipelined Processor are appended with first letter of stage (i.e., PCF, PCD, PCE).



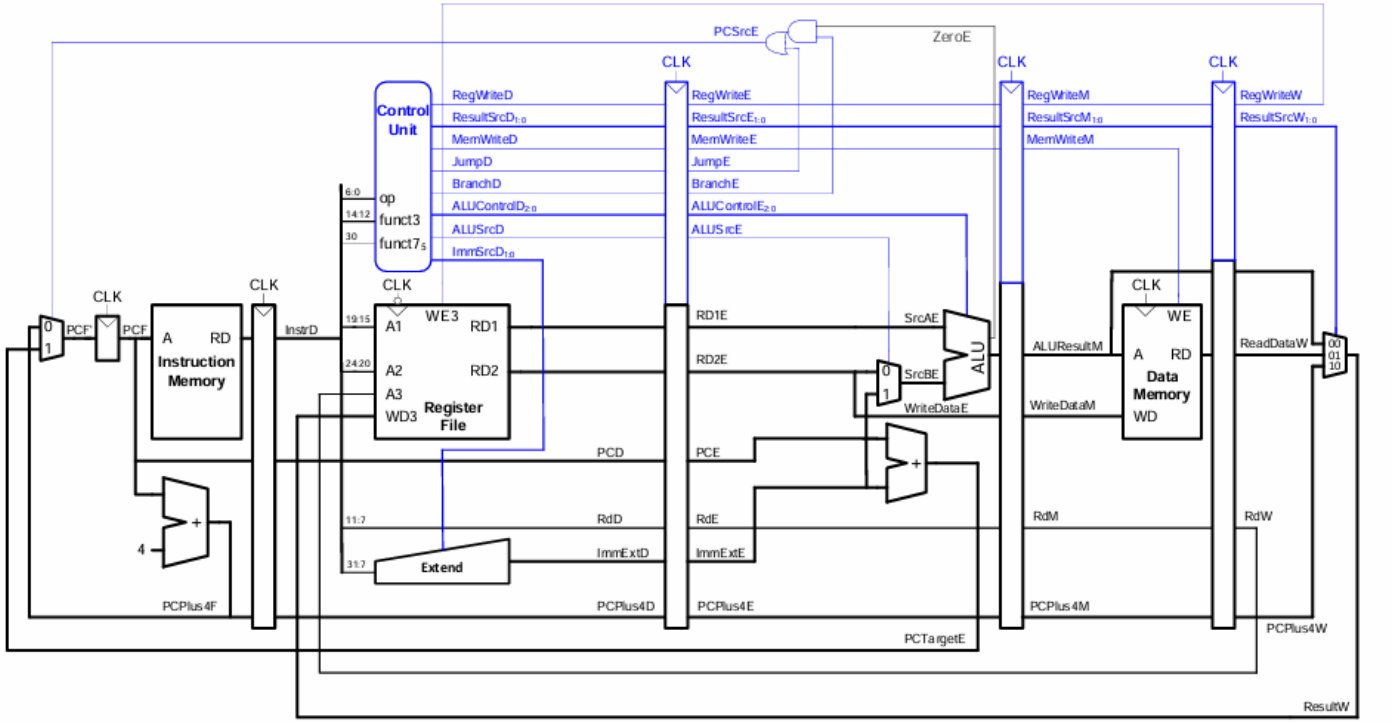
Corrected Pipelined Datapath

Pipeline register



- ***Rd*** must arrive at same time as ***Result***
- Register file written on **falling edge** of *CLK*

Pipelined Processor with Control



- **Same control unit** as single-cycle processor
- **Control signals travel with** the instruction (drop off when used)

Takeaway Questions

- Which statement is true after pipelining the single-cycle processor?
  - (a) Instructions/program (instruction counts) decreases
  - (b) Cycles/instruction (CPI) decreases
  - (c) Time/cycle (clock rate) decreases

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

Takeaway Questions

- Which of the following statement(s) is/are True or False?
  - (a) Thanks to pipelining, I have reduced the time it took me to wash my shirt.
    (b) Longer pipelines are always a win (since less work per stage & a faster clock)

Takeaway Questions

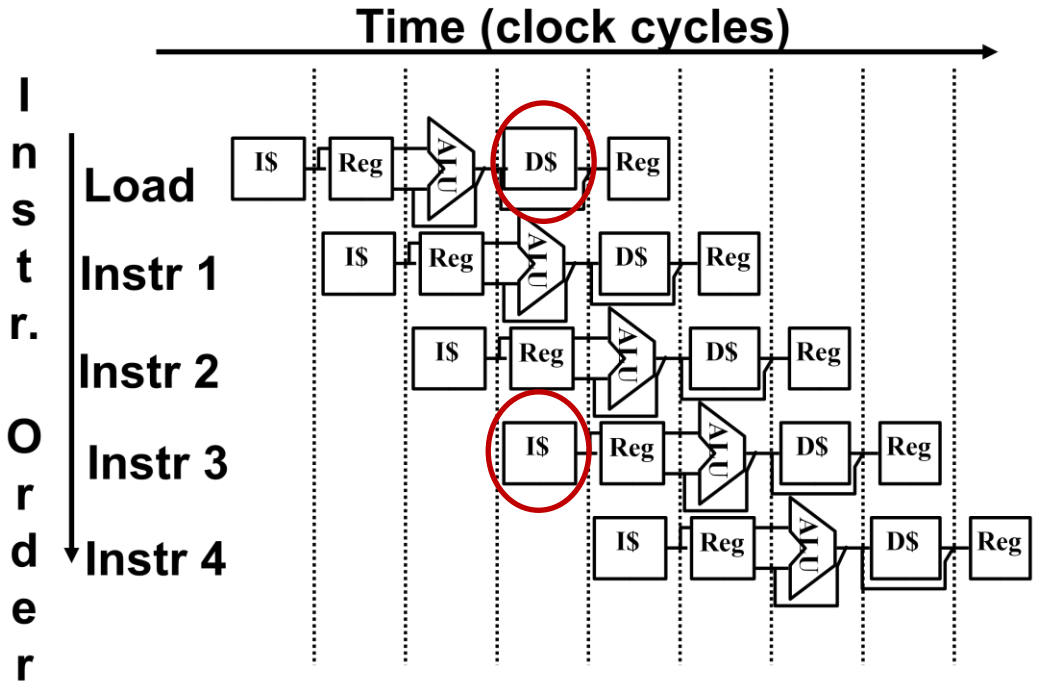
- Which of the following statement(s) is/are True or False?
  - (a) Thanks to pipelining, I have reduced the time it took me to wash my shirt. (False)
    - **Throughput better, not execution time**
  - (b) Longer pipelines are always a win (since less work per stage & a faster clock) (False)
    - **longer pipelines do usually mean faster clock, but branches cause problems!**

Pipelining Hazard

- Limits to pipelining
  - Hazards result in pipeline “**stalls**” or **“bubbles”**
  - **Structural hazards:**
    - Multiple instructions in the pipeline compete for access to a single physical resource
  - **Control hazards:**
    - Pipelining of branches causes later instruction fetches to wait for the result of the branch
  - **Data hazards:**
    - Instructions have data dependency
    - Need to wait for previous instruction complete its data read/write

Structural Hazard

- **Structural Hazard #1: Single Memory**



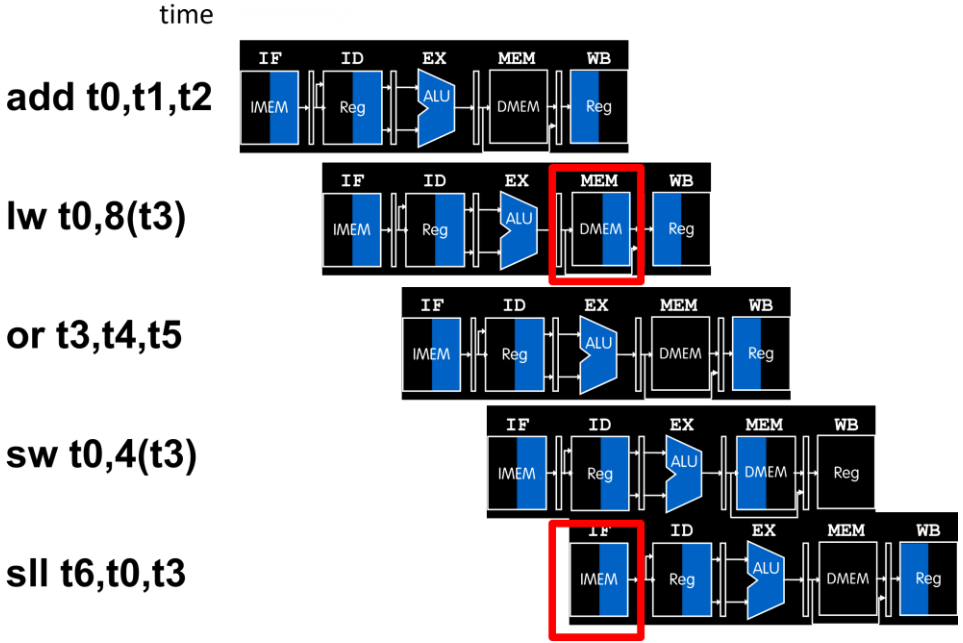
Read the same memory twice in the same clock cycle

Structural Hazard

- **Structural Hazard #1: Single Memory**
  - Infeasible and inefficient to create a second memory
  - **Solution**
    - Have both an L1 instruction cache and an L1 data cache
    - Need more complex hardware to control when both caches miss

Structural Hazard

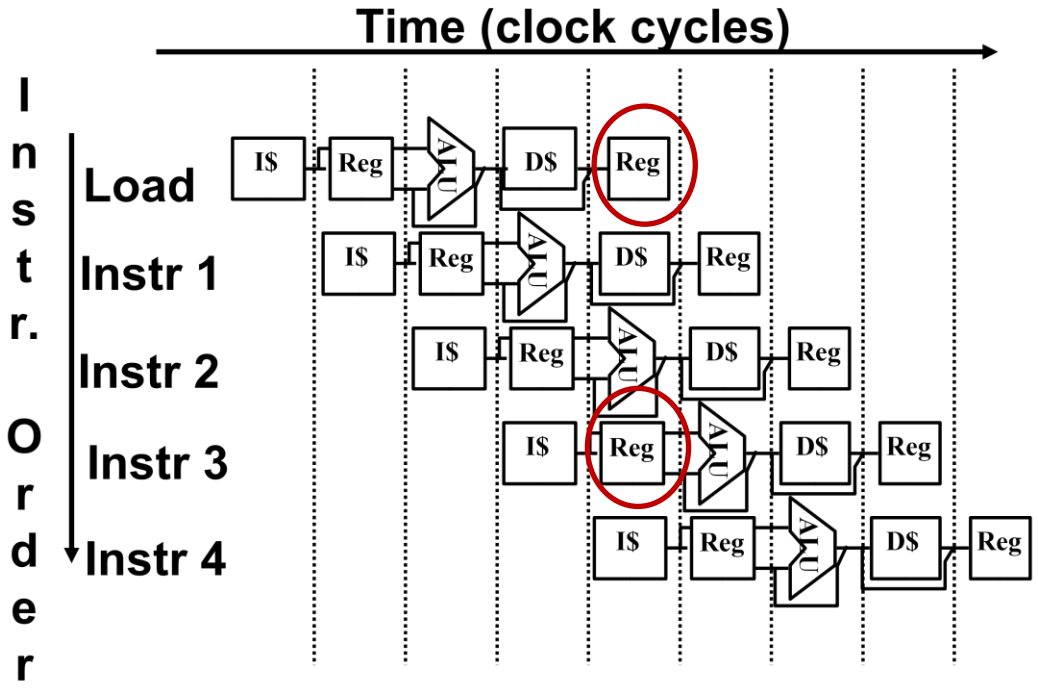
- **Structural Hazard #1: Single Memory**
  - Structural hazard if IMEM, DMEM were same hardware



RV32I separates IMEM and DMEM to avoid structural hazard

Structural Hazard

- **Structural Hazard #2: Registers**



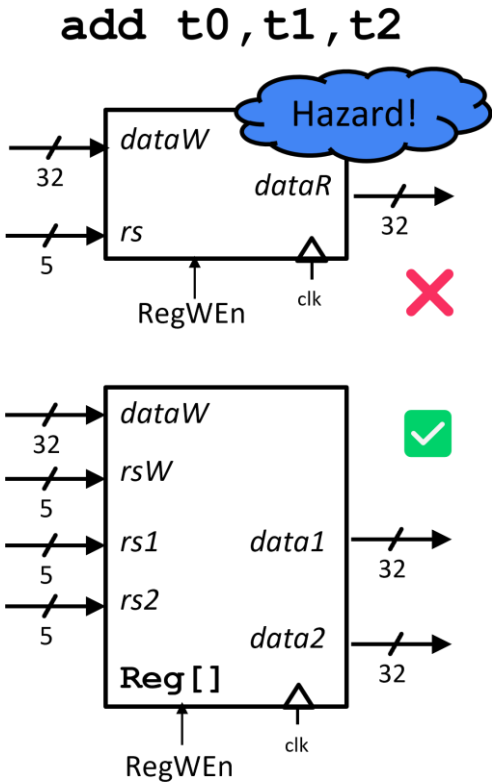
Read and write to registers simultaneously

Structural Hazard

- **Structural Hazard #2: Registers**
  - **Two different solutions** have been used
    - RegFile access is very fast: takes less than half the time of the ALU stage
      - Write to registers during the first half of each clock cycle
      - Read from registers during the second half of each clock cycle
    - Build RegFile with independent read and write ports
  - Result: can perform read and write during the same clock cycle

Structural Hazard

- **Structural Hazard #2: Registers**
  - Each RV32I instruction
    - Reads up to 2 operands in decode stage
    - Writes up to 1 operand in writeback stage
    - Structural hazard occurs if RegFile HW does **not** support simultaneous read/write !
  - RV32I RegFile-> no structural hazard
    - 2 independent read ports, 1 write port
    - Three accesses (2R/1W) can happen at the same cycle



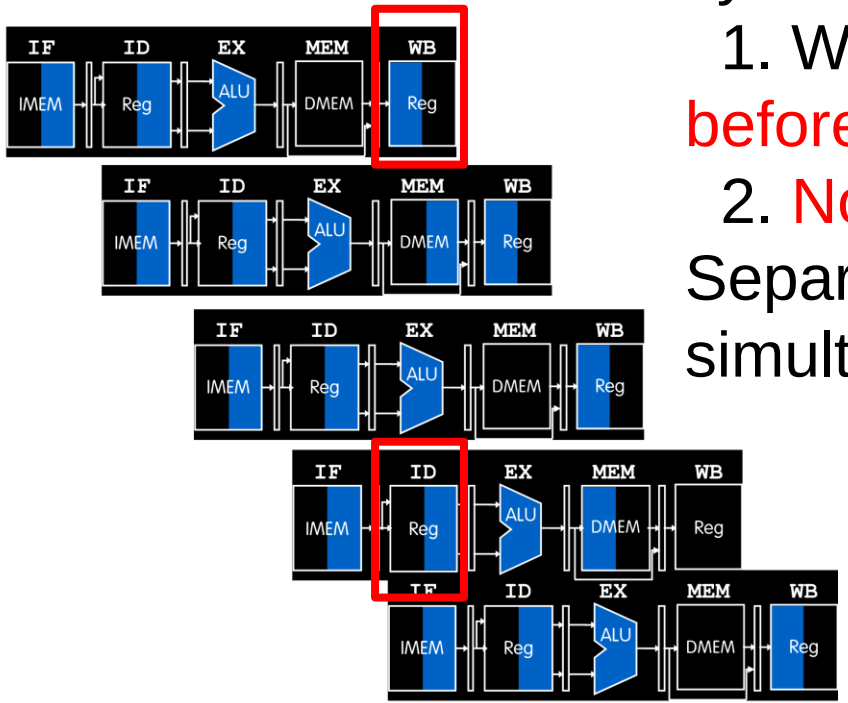
Data Hazard

- **Data hazard**
  - Instructions have data dependency
  - Need to wait for previous instruction to complete its data read/write
  - Occurs when an instruction reads a register before a previous instruction has finished writing to that register
- Three cases to consider
  - Register access
  - ALU result
  - Load data hazard

Data Hazard: REG

- **Register Access**

```
add t0,t1,t2
lw t0,8(t3)
or t3,t4,t5
sw t0,4(t3)
sll t6,t0,t3
```



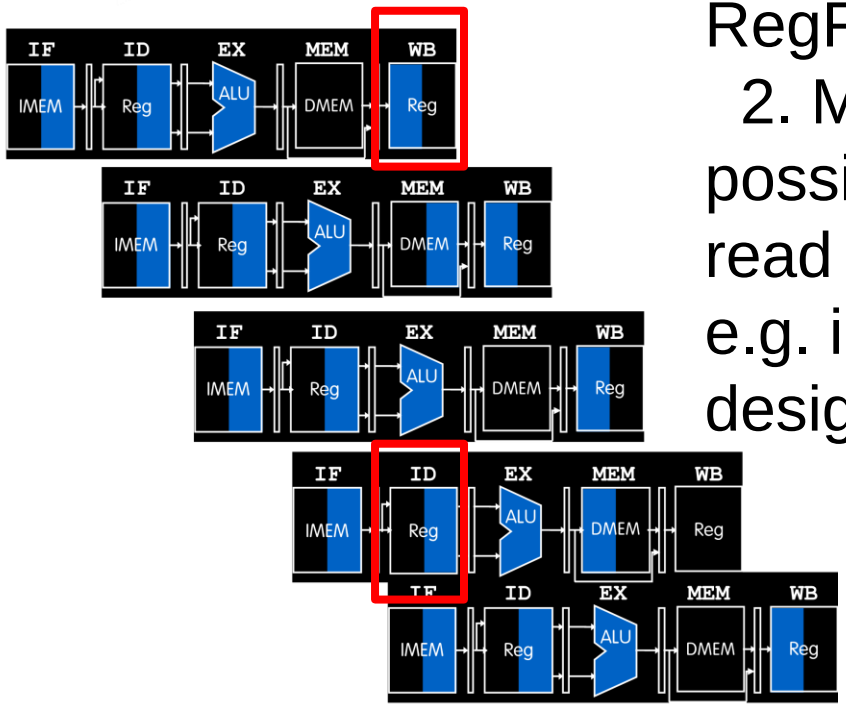
The same register is written and read in one cycle:

1. WB must write value before ID reads new value
2. No structural hazard – Separate ports allows simultaneous R/W

Data Hazard: REG

- **Register Access**

```
add t0,t1,t2
lw t0,8(t3)
or t3,t4,t5
sw t0,4(t3)
sll t6,t0,t3
```

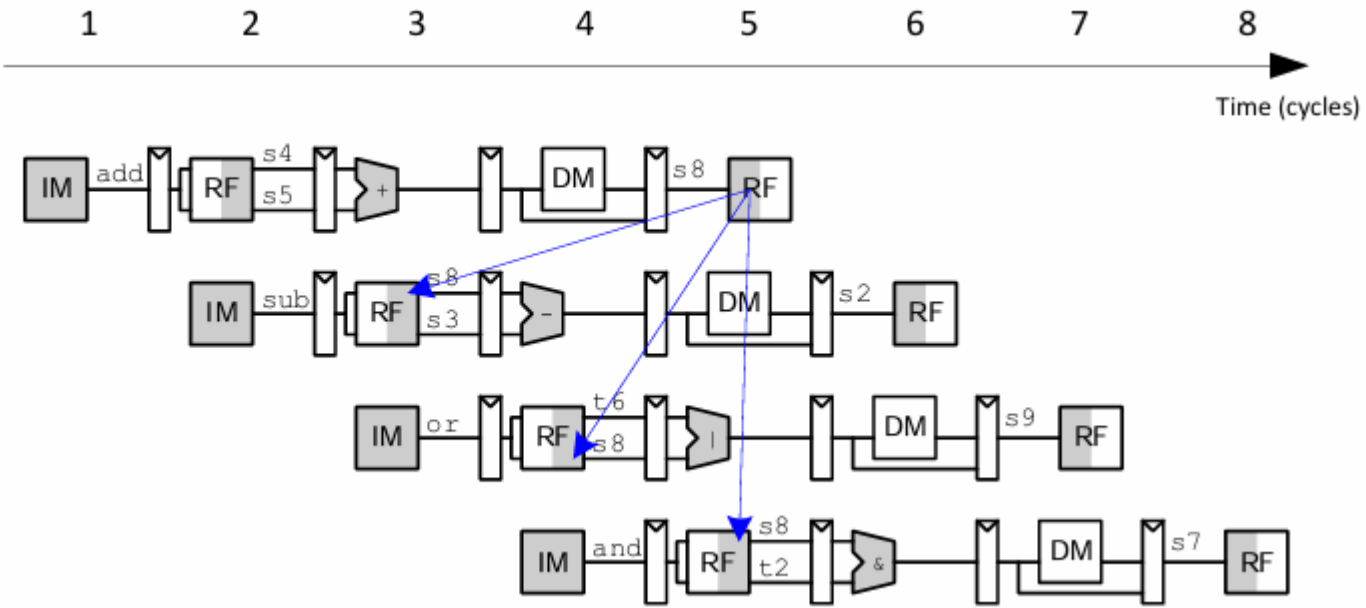


Solution: RegFile HW should write-then-read in the same cycle

1. Exploit high speed of RegFile (100 ps + 100 ps)
2. Might not always be possible to write-then-read in the same cycle., e.g. in high-frequency designs

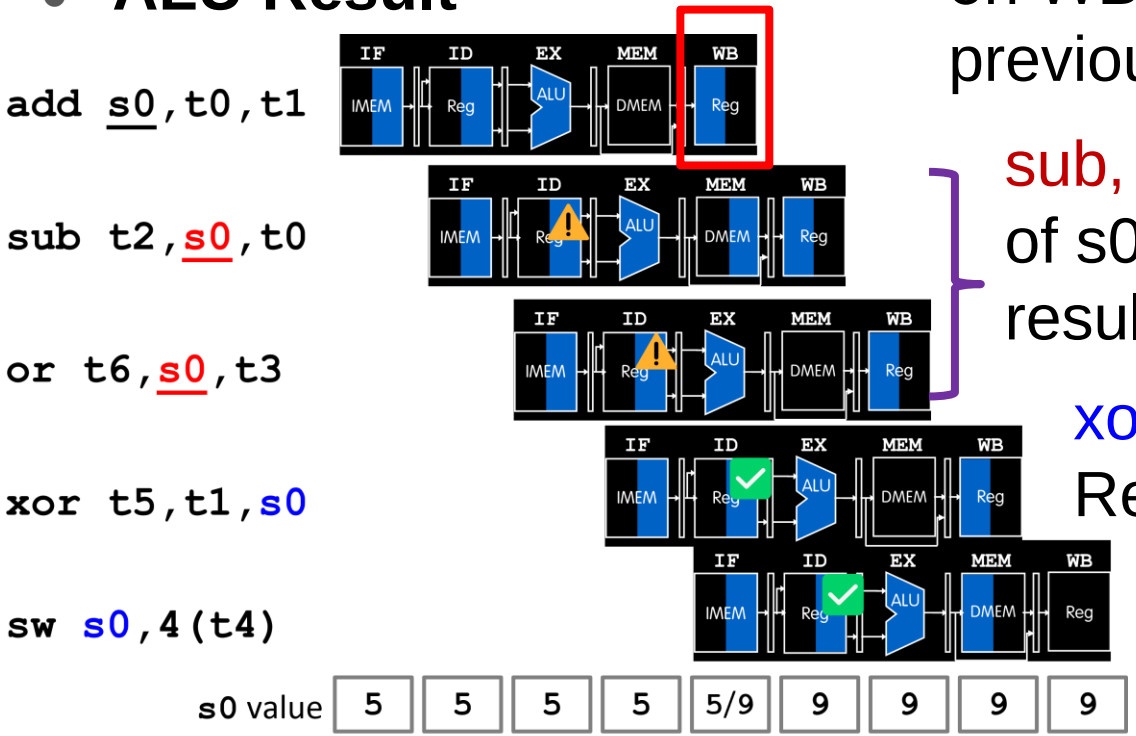
Data Hazards:REG

```
add s8, s4, s5
sub s2, s8, s3
or  s9, t6, s8
and s7, s8, t2
```



Data Hazard: ALU

- **ALU Result**



Problem: Instruction depends on WB's RegFile write from previous instruction

sub, or's ID reads old value of s0 and calculates wrong result

xor gets the right value; RegFile is write-then-read

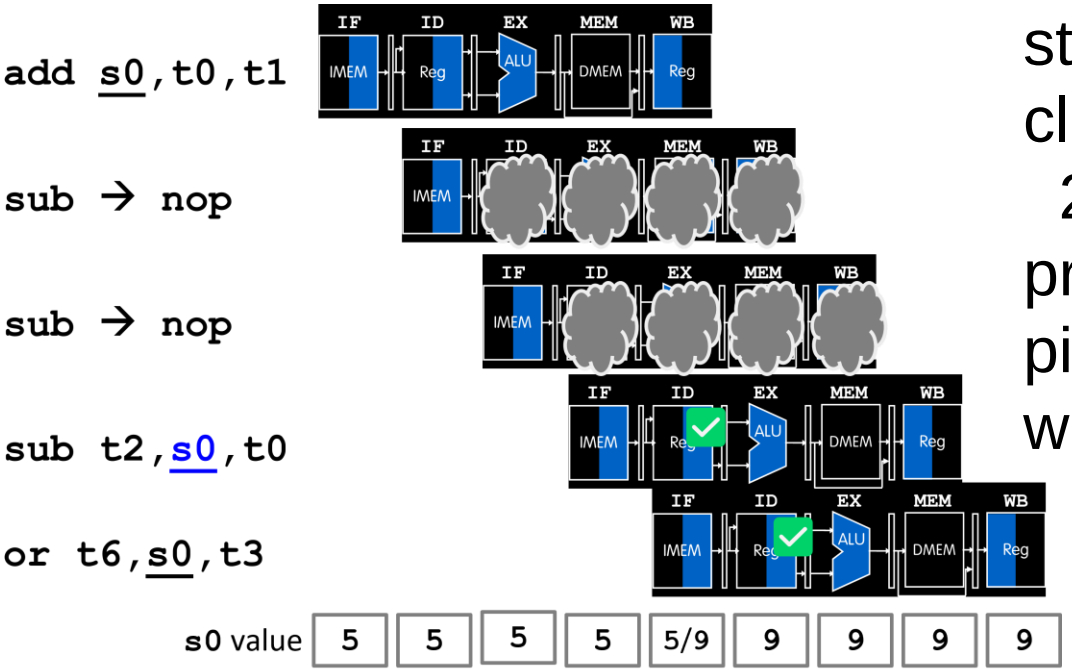
Handling Data Hazards

How do we ensure that our programs run correctly?

- Insert nops in code at compile time
- Rearrange code at compile time
- Forward data at run time
- Stall the processor at run time

Data Hazard: ALU

- **ALU solution 1: Stalling**

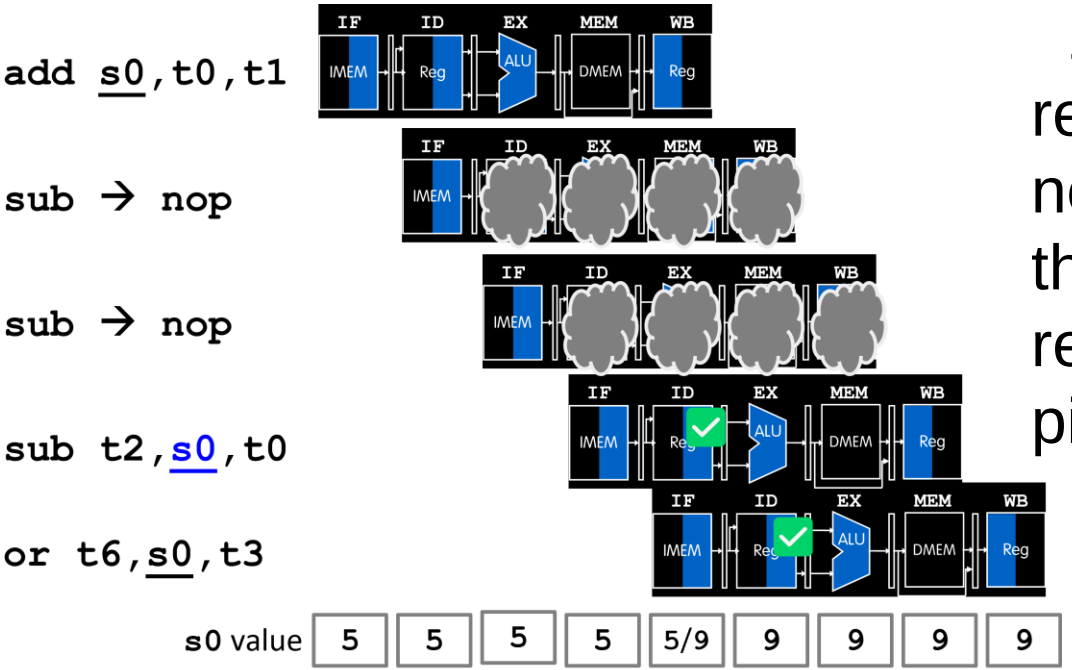


"Bubble" to effectively nop:

1. Affected pipeline stages do nothing during clock cycles
2. Stall all stages by preventing PC, IF/ID pipeline register from writing

Data Hazard: ALU

- **ALU solution 1: Stalling**

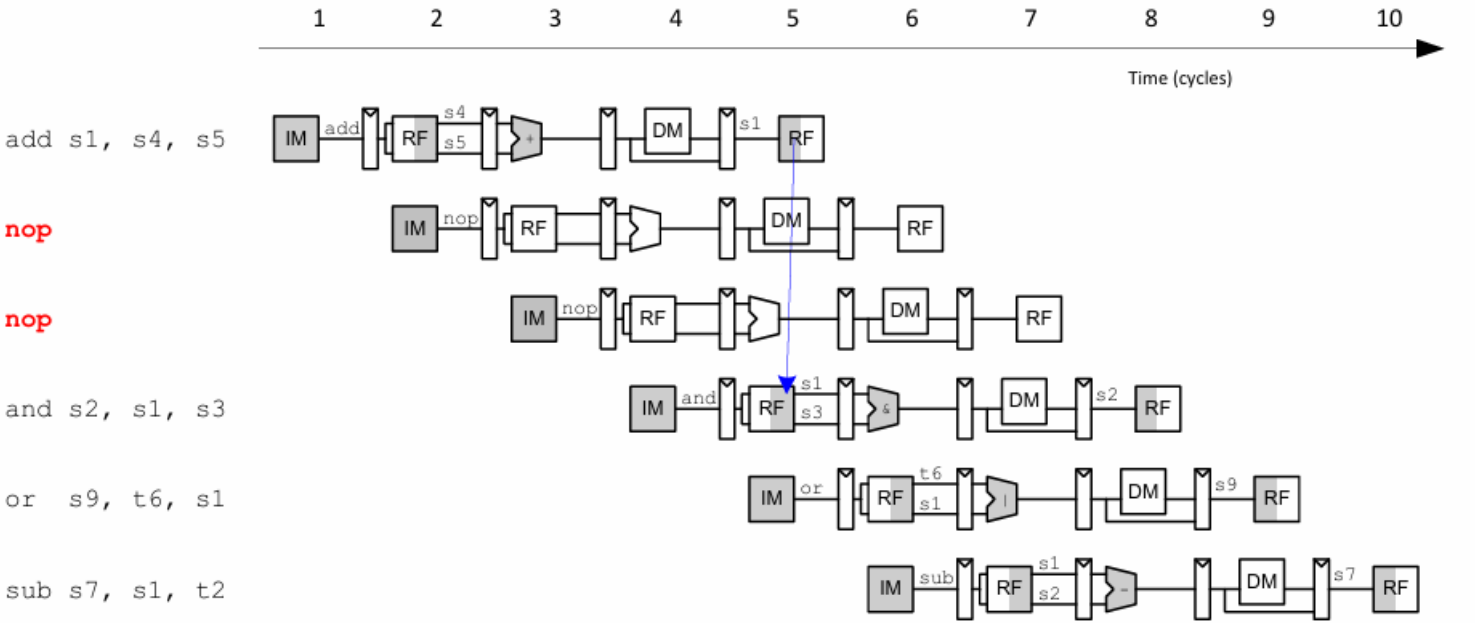


Stalls reduce performance

1. Compiler could rearrange code/insert nops to avoid hazard (and therefore stalls), but this requires knowledge of the pipeline structure

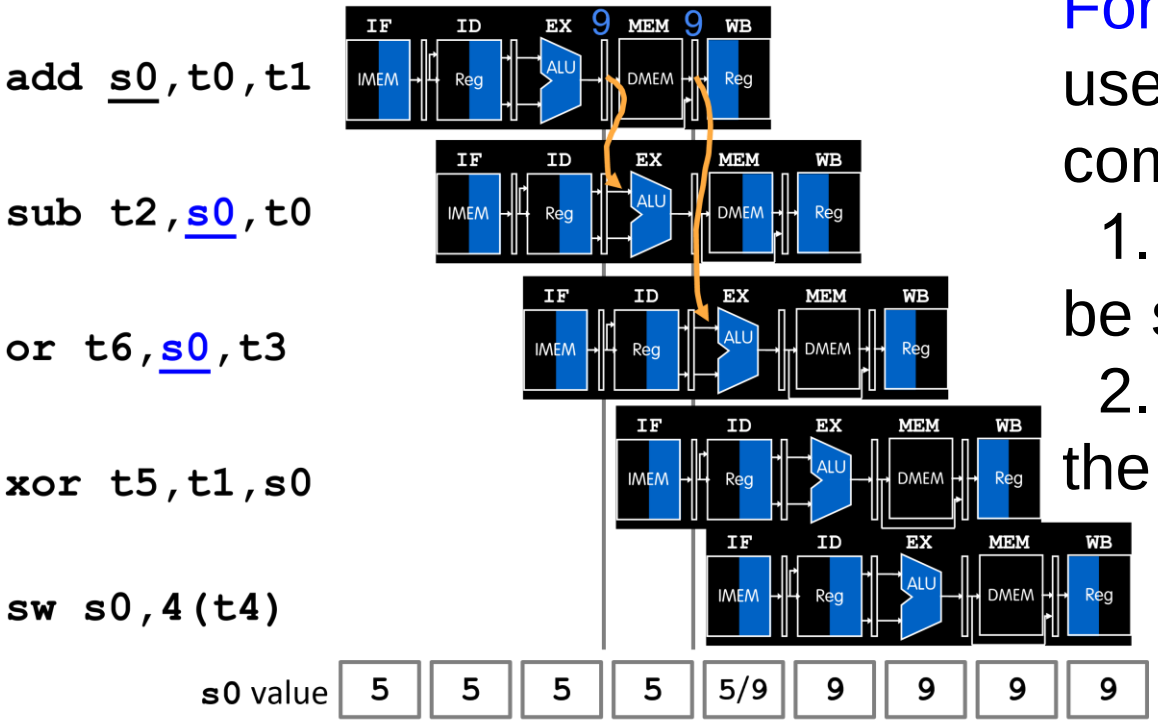
Compile-Time Hazard Elimination

- Insert enough nops for result to be ready
- Or move independent useful instructions forward



Data Hazard: ALU

- **ALU solution 2: Forwarding**

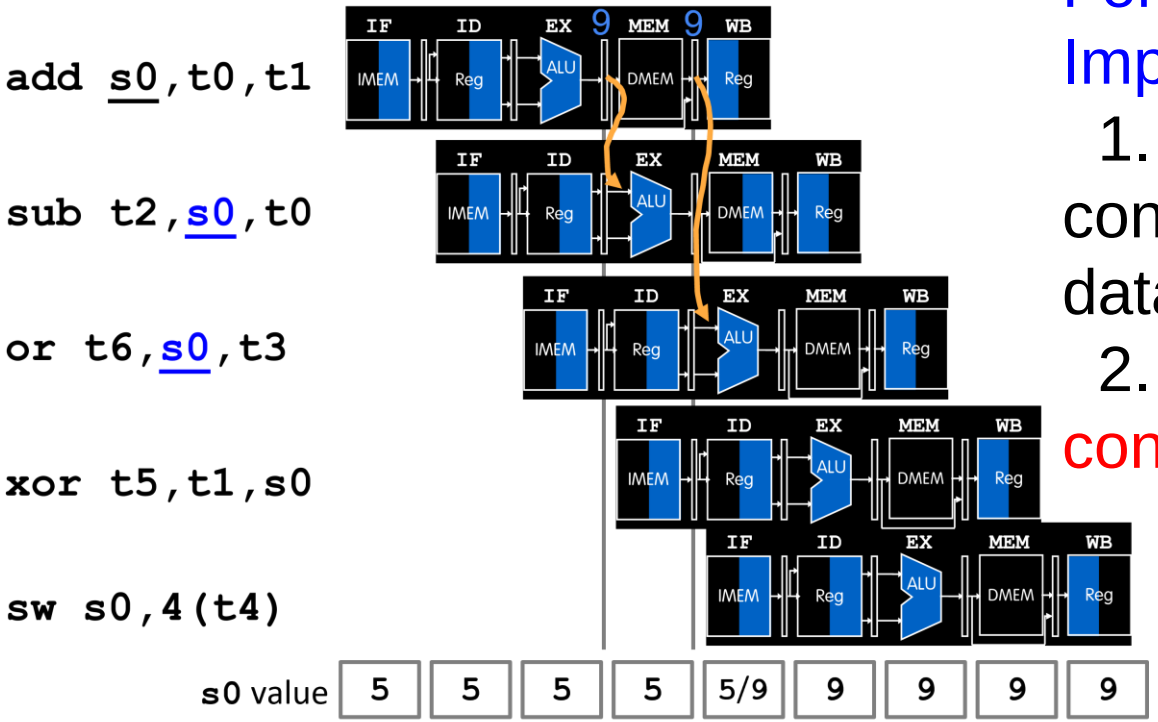


Forwarding (bypassing) uses the result when it is computed

1. Don't wait for value to be stored in RegFile
2. Grab operand from the pipeline stage

Data Hazard: ALU

- **ALU solution 2: Forwarding**

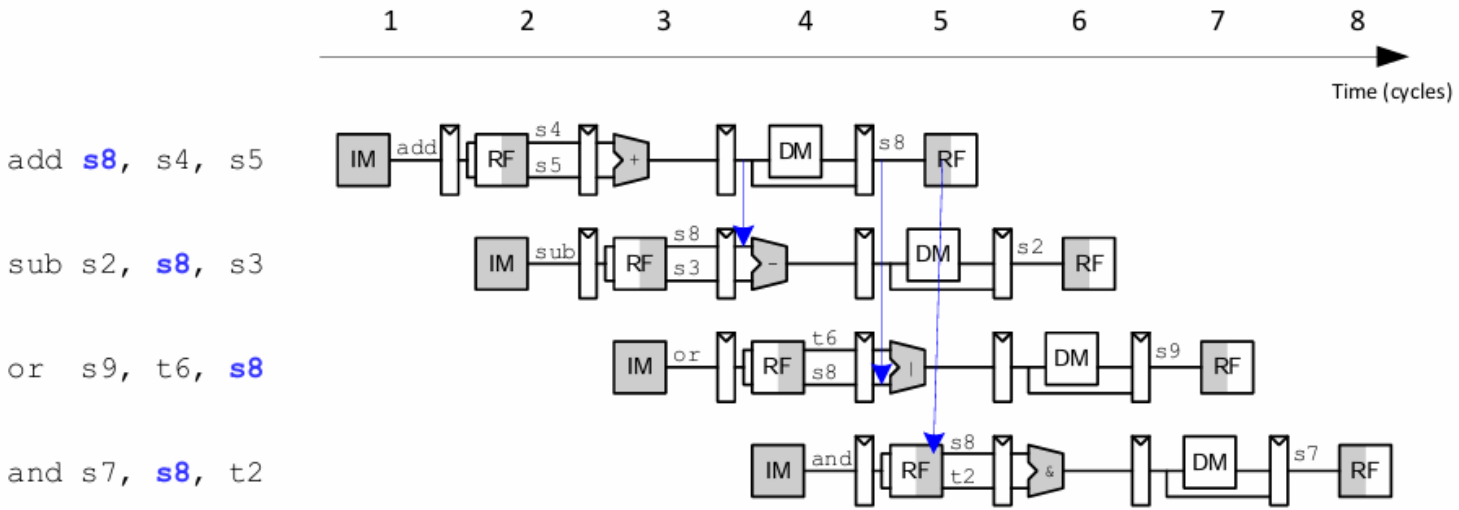


Forwarding (bypassing) Implementation:

1. Make extra connections in the datapath
2. Also add forwarding control logic

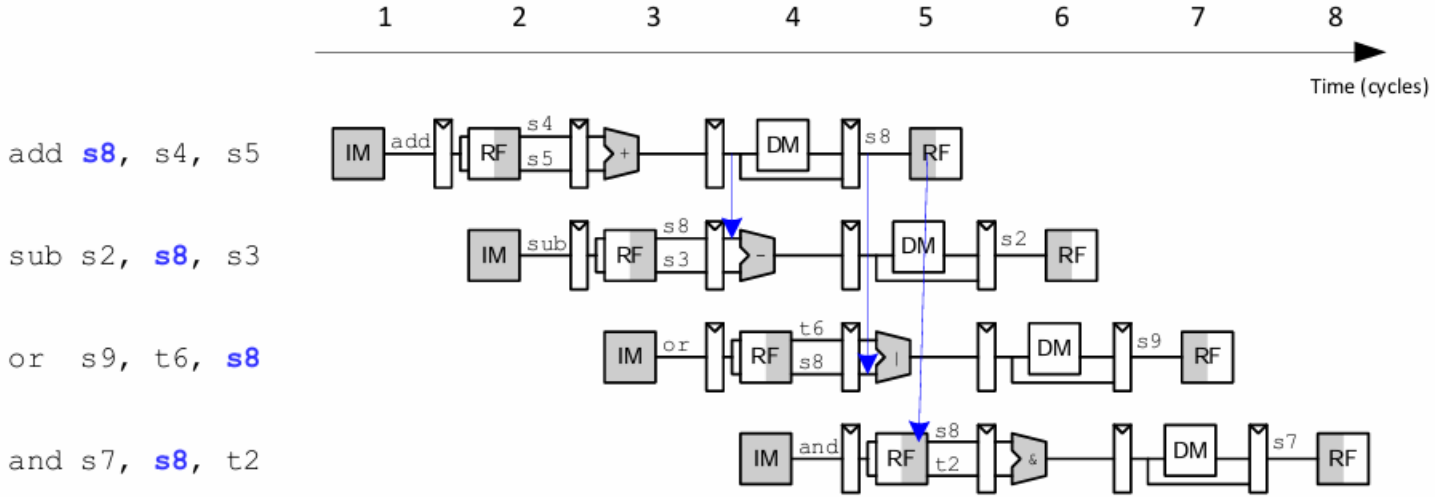
Data Forwarding

- Data is **available on internal busses** before it is written back to the register file (RF).
- **Forward data** from internal busses **to Execute stage**.



Data Forwarding

- Check if source register **in Execute stage matches** destination register of instruction **in Memory or Writeback stage**.
- If so, forward result.



Data Forwarding

- **Case 1:** **Execute** stage *Rs1* or *Rs2* matches **Memory** stage *Rd*?
  Forward from Memory stage
- **Case 2:** **Execute** stage *Rs1* or *Rs2* matches **Writeback** stage *Rd*?
  Forward from Writeback stage
- **Case 3:** Otherwise use value read from register file (as usual)

Equations for *Rs1*:

```
if      ((Rs1E == RdM) AND RegWriteM)        // Case 1
            ForwardAE = 10
else if ((Rs1E == RdW) AND RegWriteW)        // Case 2
            ForwardAE = 01
else        ForwardAE = 00                   // Case 3
```

***ForwardBE** equations are similar (replace **Rs1E** with **Rs2E**)*

Data Forwarding

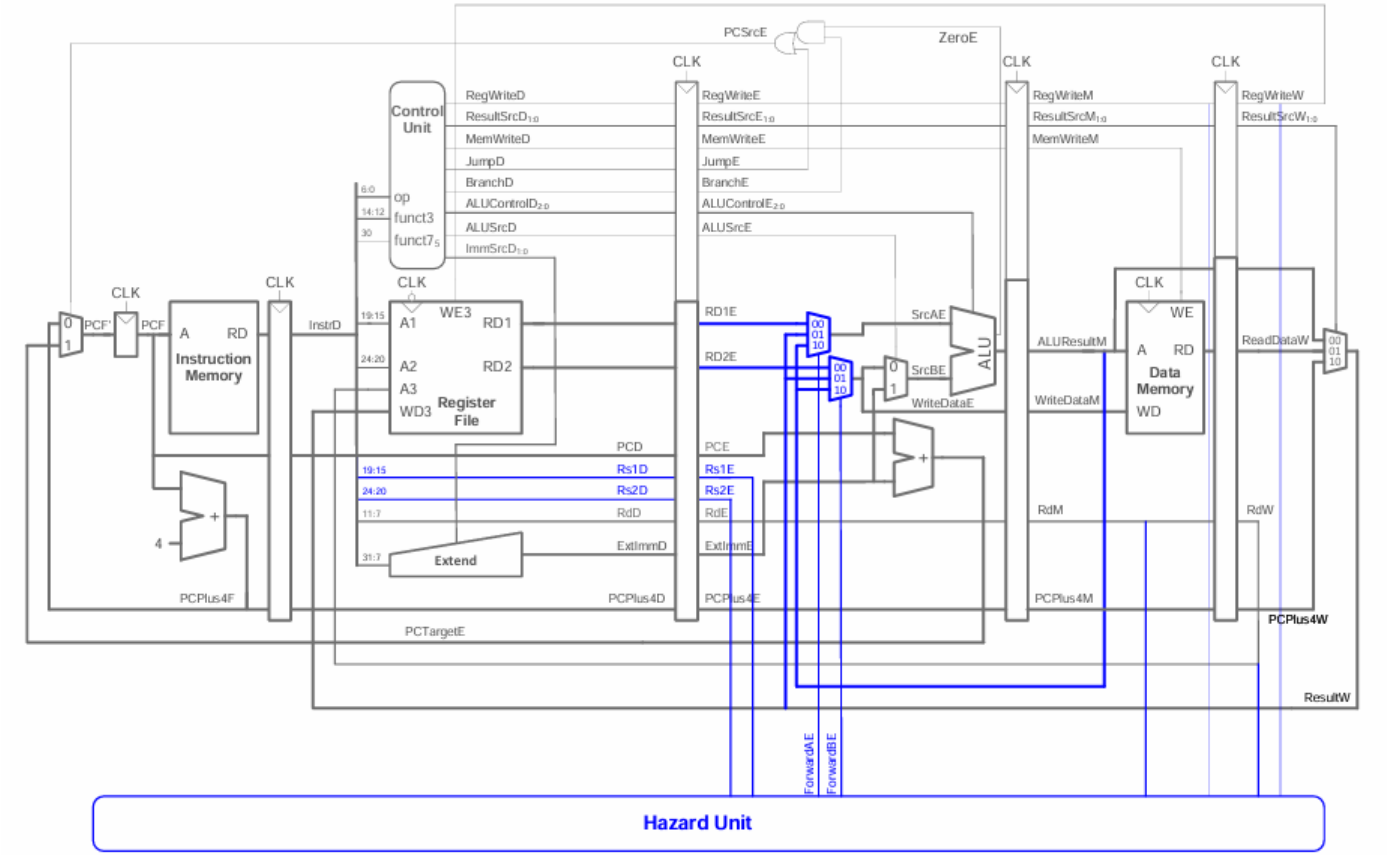
- **Case 1:** **Execute** stage *Rs1* or *Rs2* matches **Memory** stage *Rd*?
  Forward from Memory stage
- **Case 2:** **Execute** stage *Rs1* or *Rs2* matches **Writeback** stage *Rd*?
  Forward from Writeback stage
- **Case 3:** Otherwise use value read from register file (as usual)

Equations for *Rs1*:

if ((*Rs1E* == *RdM*) AND *RegWriteM*) AND (Rs1E != 0) // Case 1
    ForwardAE = 10
else if ((*Rs1E* == *RdW*) AND *RegWriteW*) AND (Rs1E != 0) // Case 2
    ForwardAE = 01
else *ForwardAE* = 00 // Case 3

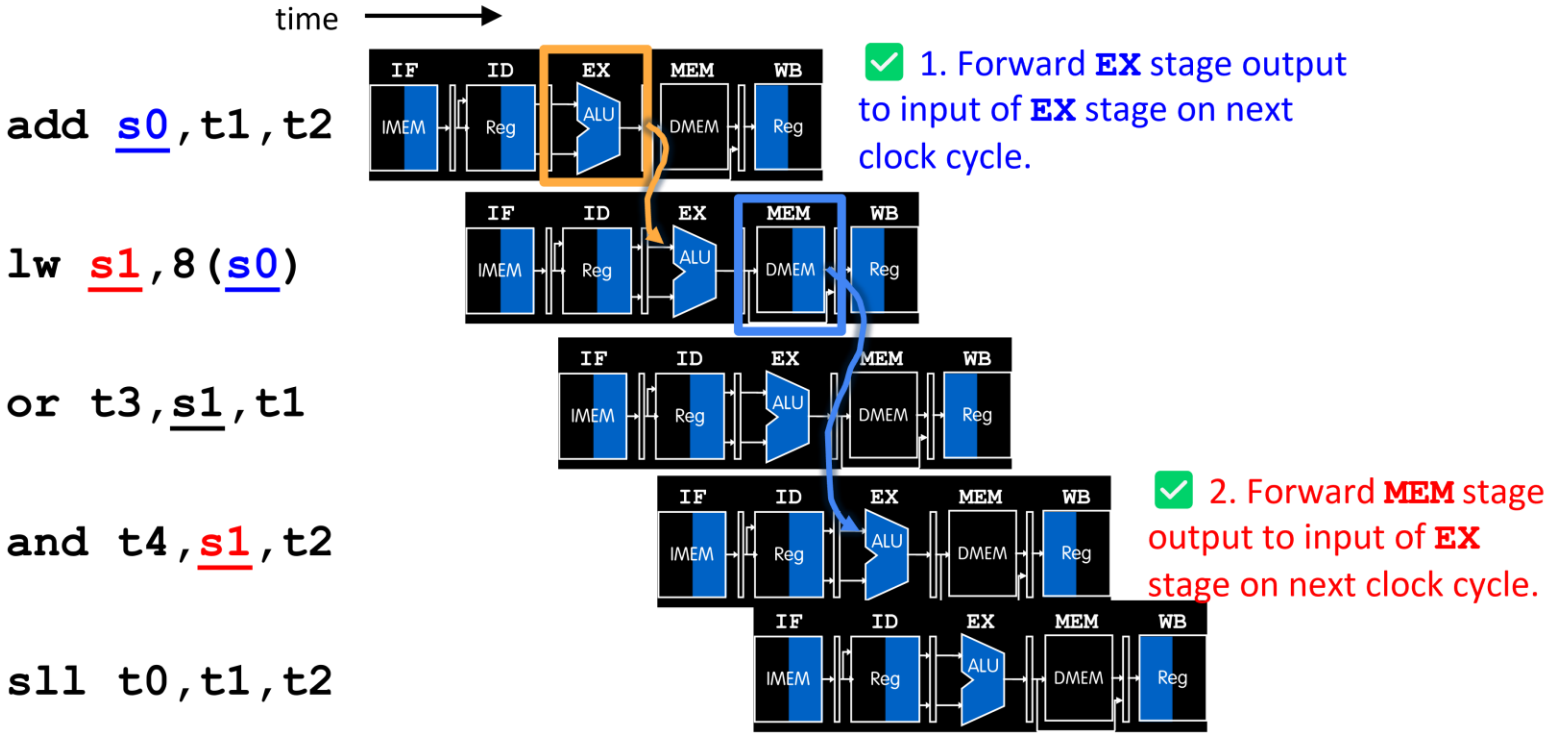
ForwardBE *equations are similar (replace* ***Rs1E*** *with* ***Rs2E****)*

Data Forwarding: Hazard Unit



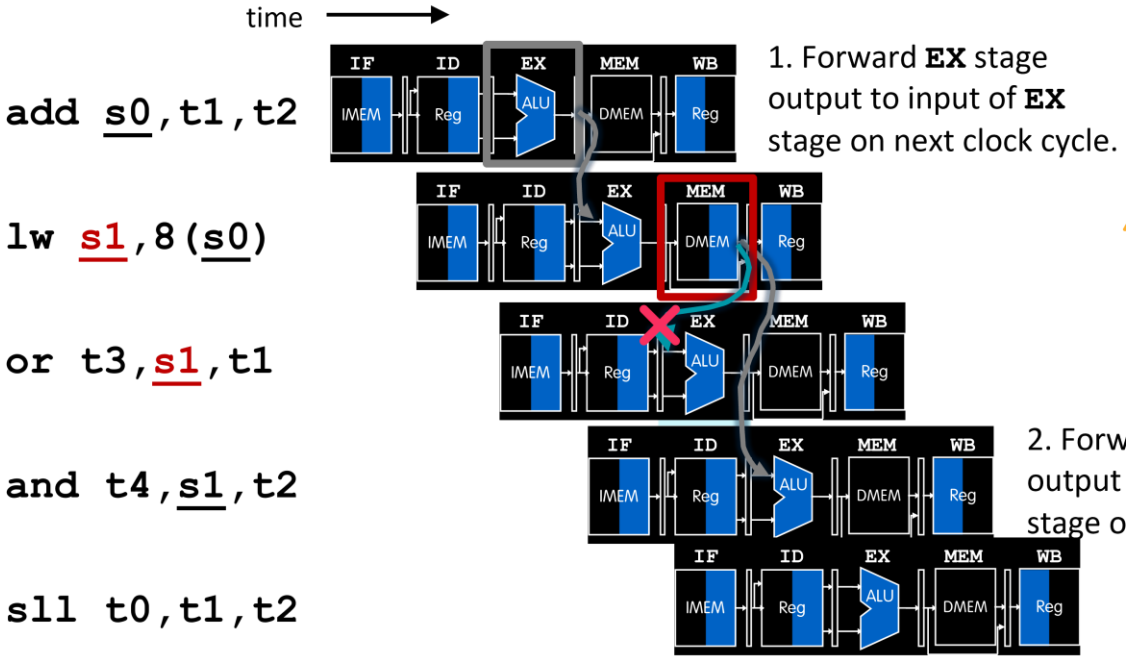
Data Hazard: Load (1/8)

- **Forwarding cannot fix all data hazards**



Data Hazard: Load (2/8)

- **Forwarding cannot fix all data hazards**



1. Forward **EX** stage output to input of **EX** stage on next clock cycle.

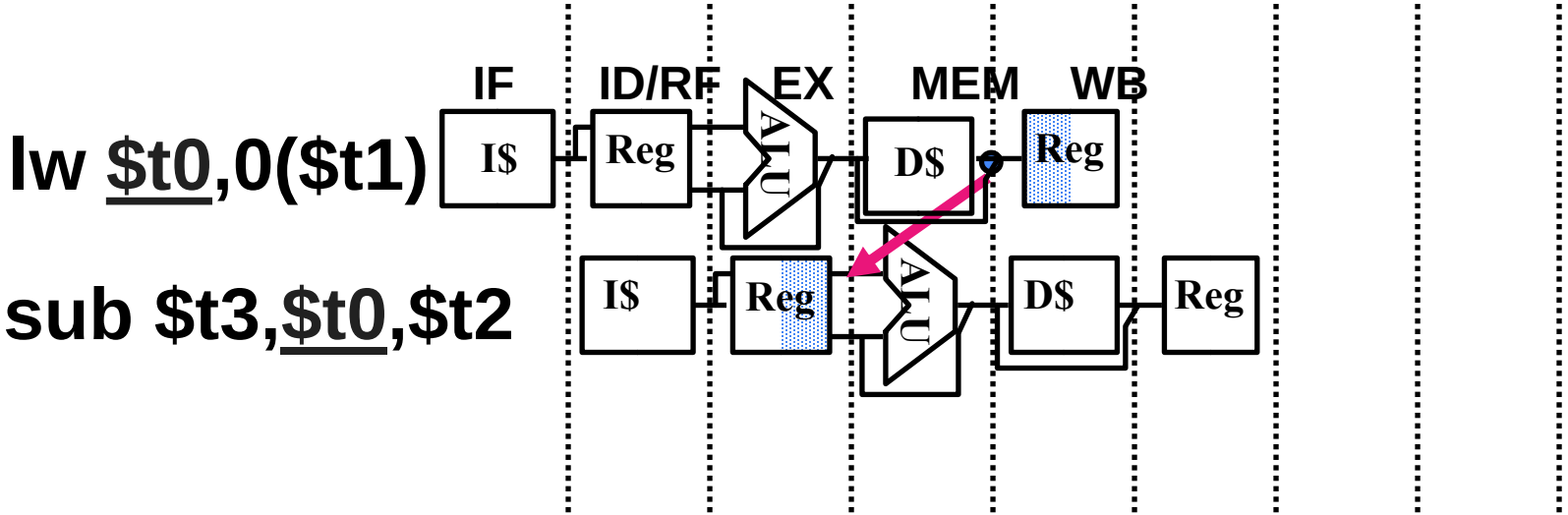
⚠ 3. **MEM** stage (**lw**)'s output needed as **EX** stage (**or**)'s input *in the same clock cycle*.

2. Forward **MEM** stage output to input of **EX** stage on next clock cycle.

Loads can result in an unavoidable pipeline stall.

Data Hazard: Load (3/8)

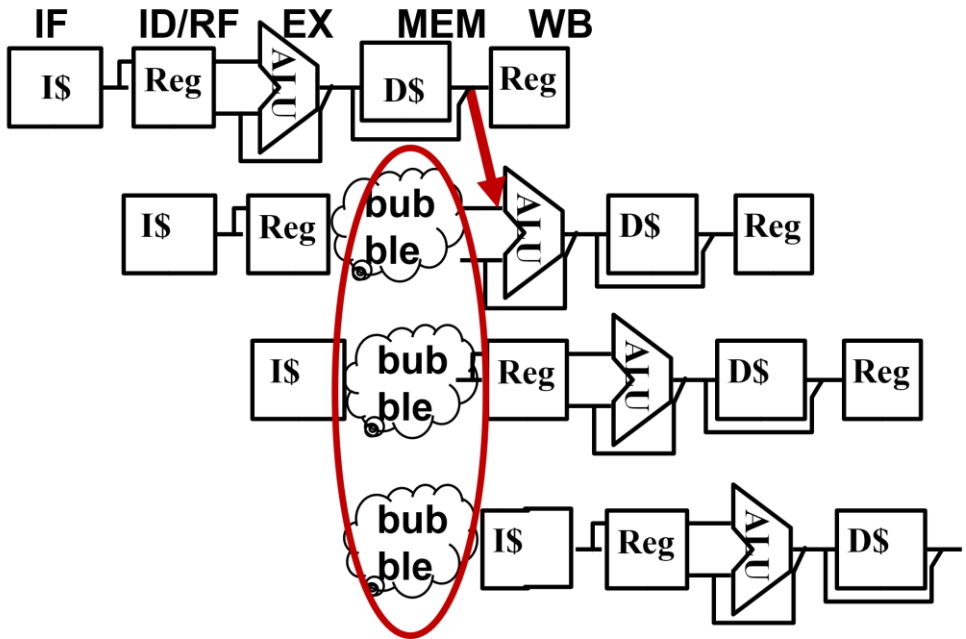
- **Forwarding cannot fix all data hazards**
  - Must stall instruction dependent on load, then forward (more hardware)



Data Hazard: Load (4/8)

- **Hardware stalls pipeline**
  - Called “interlock”

```
lw $t0, 0($t1)
sub $t3,$t0,$t2
and $t5,$t0,$t4
or  $t7,$t0,$t6
```



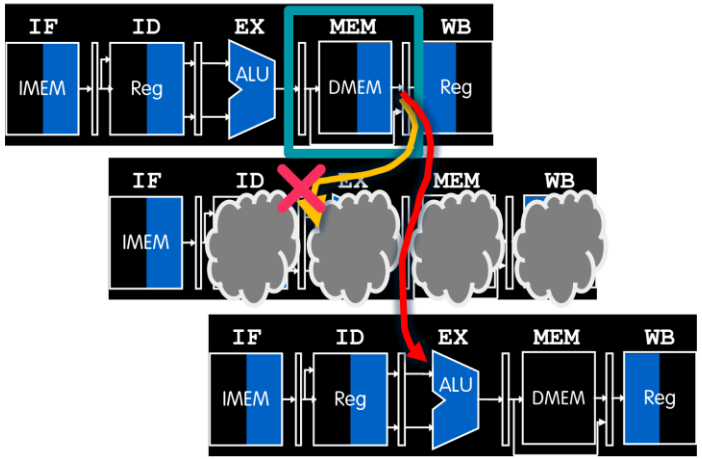
Data Hazard: Load (5/8)

- The instruction slot after a load is called load delay slot
- If this instruction uses the result of load
  - The hardware must stall for one cycle (plus forwarding)
  - This results in performance loss!

```
lw s1,8(s0)

Load delay slot:
or → nop

or t3,s1,t1
```

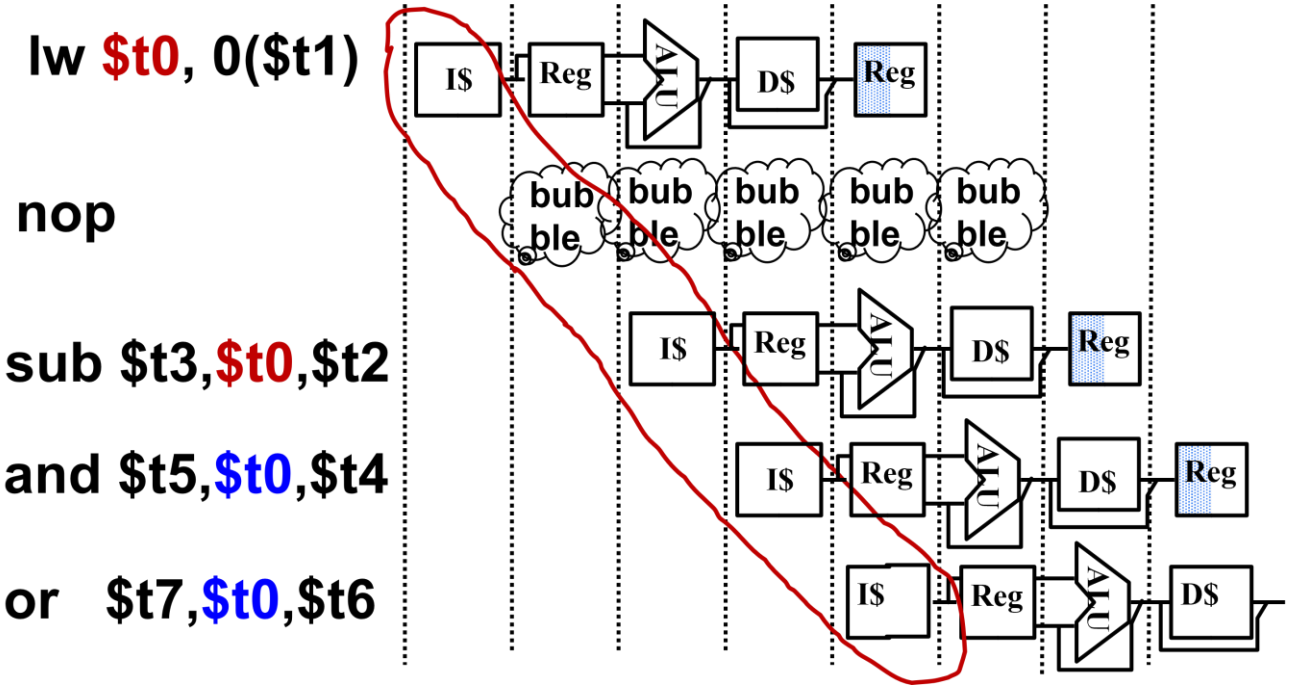


⚠ **MEM** stage (**lw**)'s output needed as **EX** stage (**or**)'s input *in the same clock cycle*.

Forwarding sends data to *the next clock cycle*.
Cannot go backwards in time!

Data Hazard: Load (6/8)

- Stall is equivalent to “nop”



Data Hazard: Load (7/8)

- Code scheduling: Fix data hazard using the compiler
  - In the delay slot, put an instruction unrelated to the load result
  - -> No performance loss!

C Code

```
A[3] = A[0] + A[1];
A[4] = A[0] + A[2];
```

Code scheduling: With knowledge of the underlying CPU pipeline, the compiler reorders code to improve performance.

⚠ Simple compilation (9 cycles for 7 instructions)

```
lw   t1, 0(t0)
lw   t2, 4(t0)
add  t3, t1, t2
sw   t3, 12(t0)
lw   t4, 8(t0)
add  t5, t1, t4
sw   t5, 16(t0)
```

Stall & forward! (+1 cycle)

(+1 cycle)

✅ Alternative (7 cycles):

```
lw   t1, 0(t0)
lw   t2, 4(t0)
lw   t4, 8(t0)
add  t3, t1, t2
sw   t3, 12(t0)
add  t5, t1, t4
sw   t5, 16(t0)
```

Forward! (+0 cycle)

Data Hazard: Load (8/8)

- Instruction slot after a load is called “load delay slot”
- If the instruction uses the result of the “**LOAD**”
  - The hardware interlock will stall it for one cycle
- If the compiler puts an unrelated instruction in that slot
  - No stall
  - Letting the hardware stall the instruction in the delay slot is equivalent to putting a NOP in the slot

Stalling Logic

- Is either **source register in the Decode stage** the same as the **destination register in the Execute stage**?

  AND

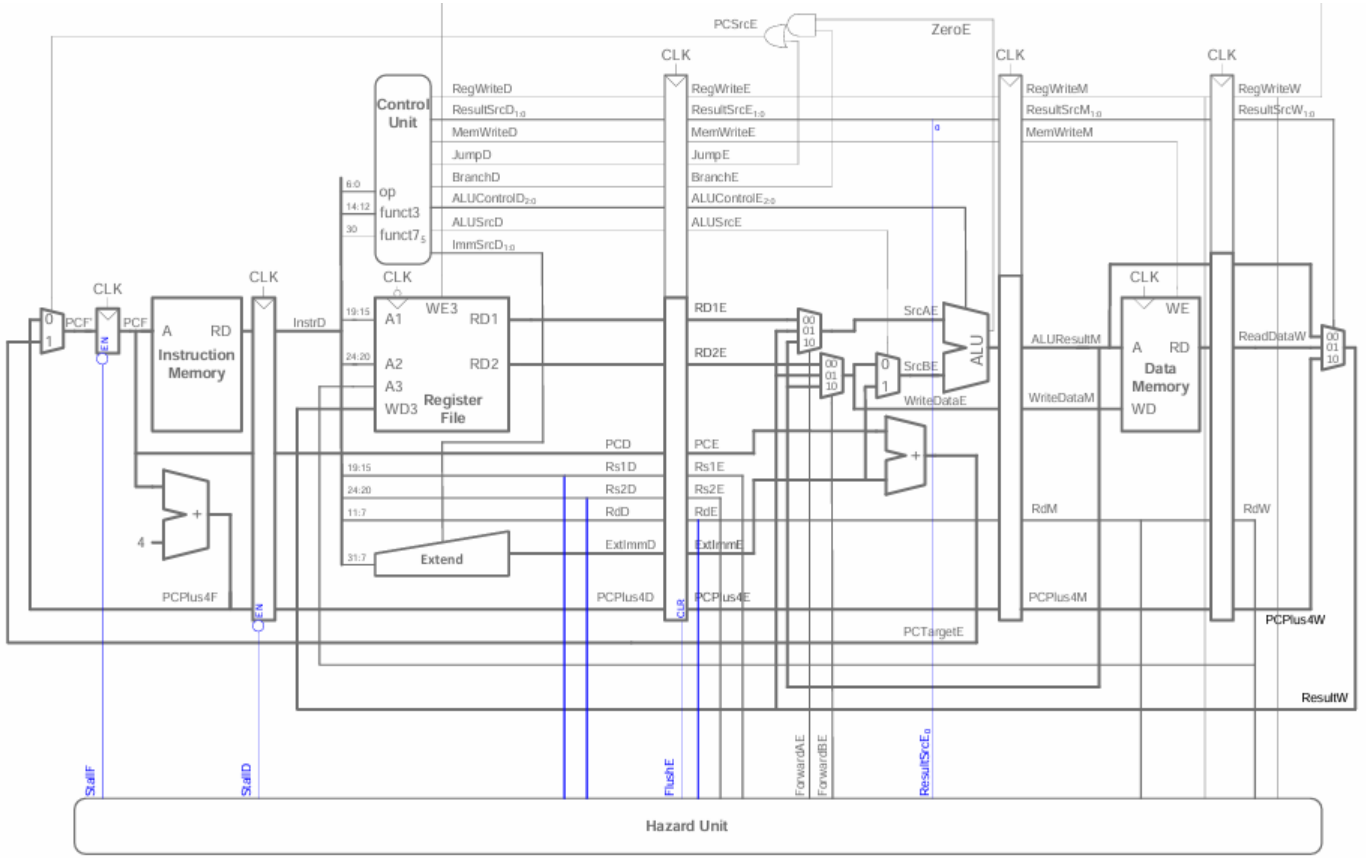
- Is the instruction in the **Execute stage a `lw`**?

$$lwStall = ((Rs1D == RdE) \text{ OR } (Rs2D == RdE)) \text{ AND } ResultSrcE_0$$

$$StallF = StallD = FlushE = lwStall$$

(Stall the Fetch and Decode stages, and flush the Execute stage.)

Stalling Hardware



Takeaway Questions

- Assume a program executed in a processor
  - Branch: 20%, load: 20%, store: 10%, others: 50%
  - 50% of loads are followed by dependent instruction
    - Require 1 cycle stall (i.e. instruction of 1 nop)
- What is the CPI of such a program in this processor?

Takeaway Questions

- As before
  - Branch: 20%, load: 20%, store: 10%, others: 50%
- Hardware interlocks: same as software interlock
  - 20% of instructions require 1 cycle stall (i.e. insertion of 1 nop)
  - 5% of instructions require 2 cycle stall (i.e. insertion of 2 nops)
- What is the CPI?

Takeaway Questions

- As before
  - Branch: 20%, load: 20%, store: 10%, others: 50%
- Hardware interlocks: same as software interlock
  - 20% of instructions require 1 cycle stall (i.e. insertion of 1 nop)
  - 5% of instructions require 2 cycle stall (i.e. insertion of 2 nops)
- What is the CPI?
  - CPI = 1 + 0.2 * 1 + 0.05 * 2 = 1.3
  - In software, # instructions would increase 30%
  - In hardware, # instructions stays at 1, but CPI would increase 30%

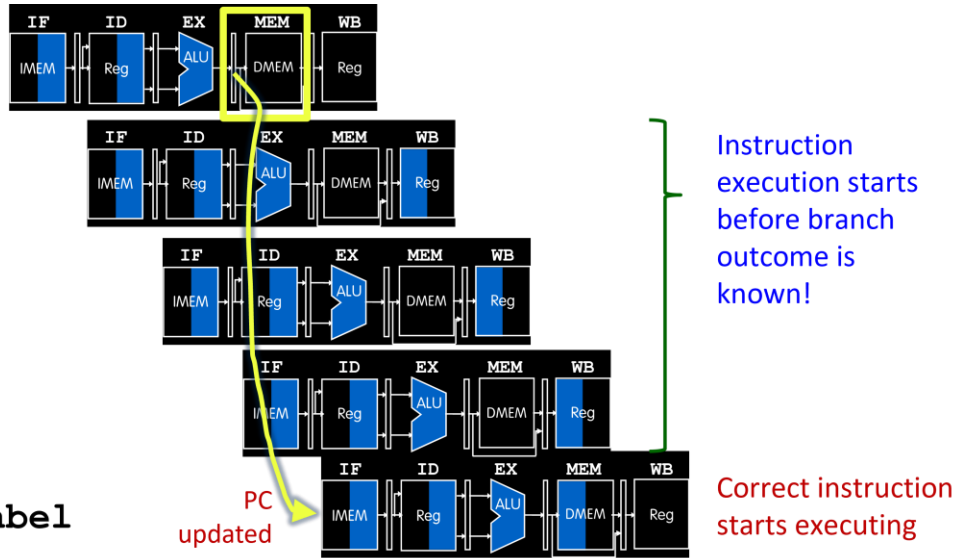
Control Hazards

- **beq:**
  - Branch **not determined until the Execute stage** of pipeline
  - **Instructions** after branch **fetched** before branch occurs
  - These **2 instructions must be flushed** if branch happens

Control Hazard

- Control hazard (conditional branch) occurs when the instruction fetched may not be the one needed
  - For example, if the “beq” branch is **taken**

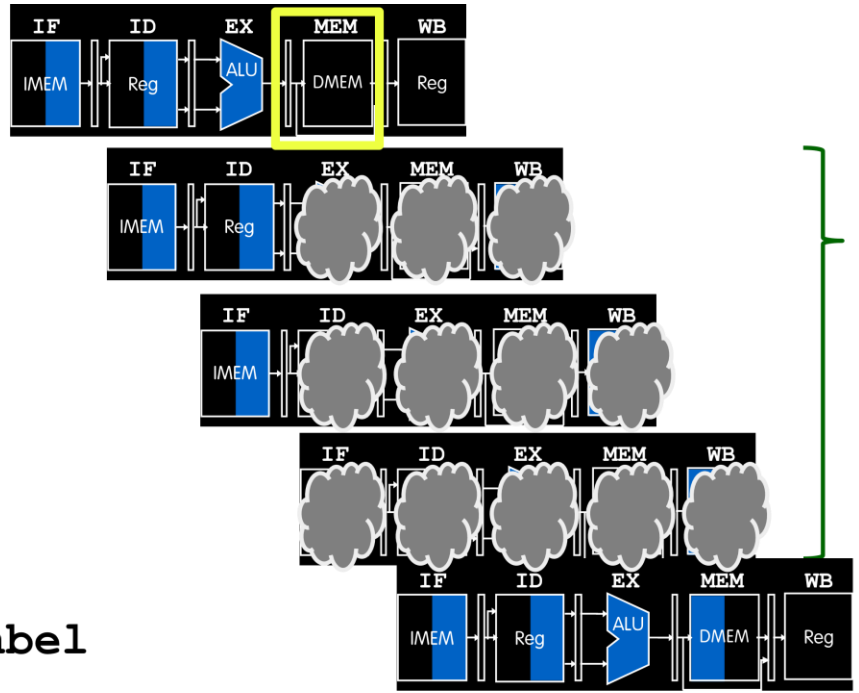
```
0x40 beq t0,t1,Label
0x44 sub t2,s0,t0
0x48 or t6,s0,t3
0x4c xor t5,t1,s0
0x70 sw s0,8(t3) # Label
```



Control Hazard

- Kill instructions after branch (if taken)

```
0x40 beq t0,t1,Label
0x44 sub t2,s0,t0
0x48 or t6,s0,t3
0x4c xor t5,t1,s0
0x70 sw s0,8(t3) # Label
```

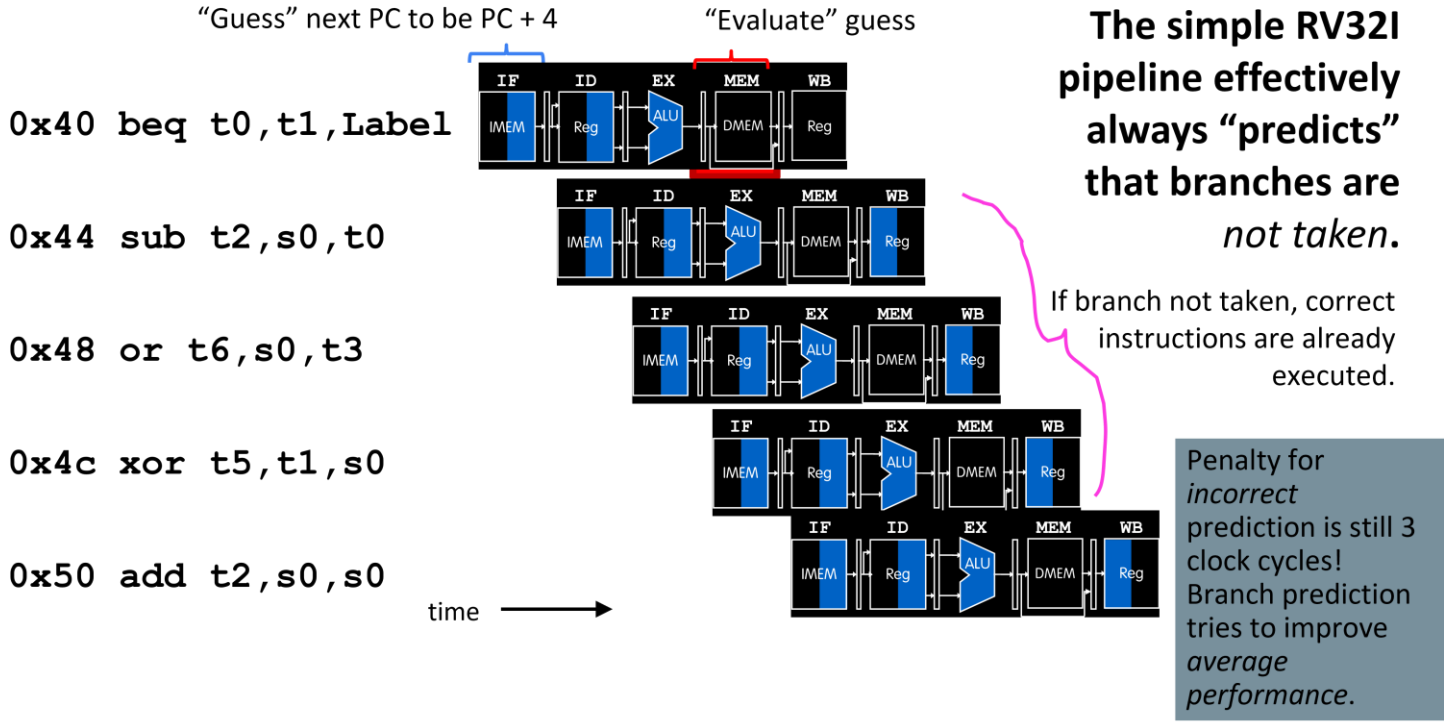


Control Hazard

- **Branch prediction reduces penalties**
  - Every taken branch in the RV32I pipeline costs 3 clock cycles
  - Note if branch is not taken, then pipeline is not stalled
  - The correct instructions are correctly fetched sequentially after the branch instruction
- **We can improve the CPU performance on average through branch prediction**
  - Early in the pipeline, guess which way branches will go
  - Flush pipeline if branch prediction was incorrect

Control Hazard

- Naïve predictor: Don't take branch



Control Hazard

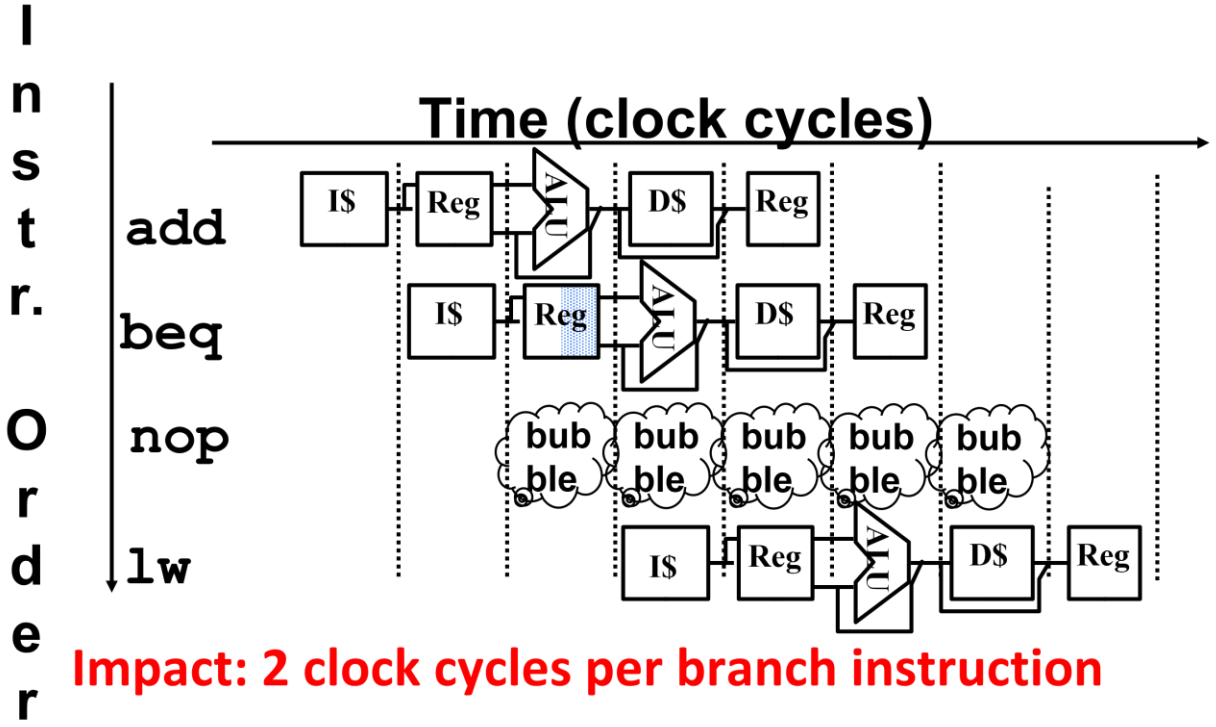
- We put branch decision-making hardware **in ALU stage**
  - Therefore, two more instructions after the branch will always be fetched, whether or not the branch is taken
- **Desired functionality of a branch**
  - If we do not take the branch, don’t waste any time and continue executing normally
  - If we take the branch, don’t execute any instructions after the branch, just go to the desired label

Control Hazard

- **Initial Solution: Stall until decision is made**
  - Insert “no-op” instructions (those that accomplish nothing, just take time) or hold up the fetch of the next instruction (for 2 cycles)
  - **Drawback**
    - Seems wasteful, particularly when the branch is not taken
    - Branches take 3 clock cycles each (assuming comparator is put in ALU stage)

Control Hazard

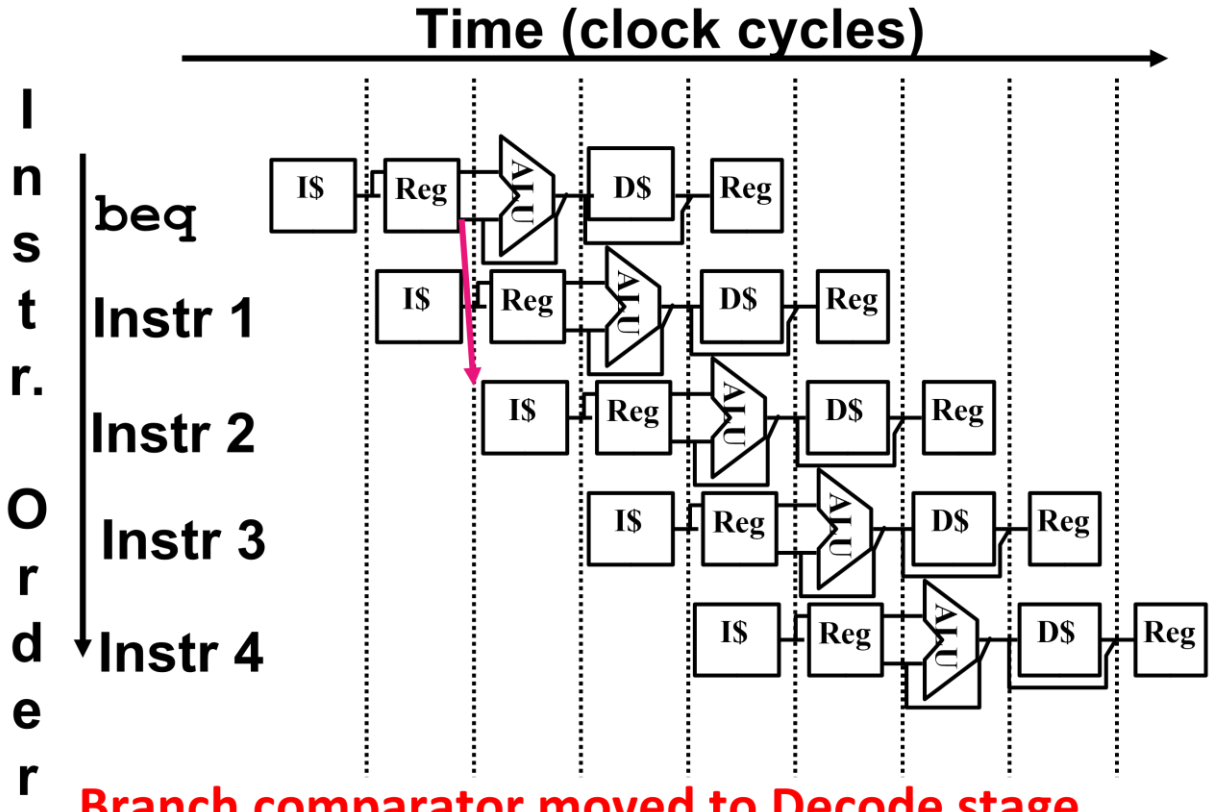
- User inserting no-op instruction



Control Hazard

- **Optimization #1**
  - Insert special branch comparator in Stage 2
  - As soon as instruction is decoded (Opcode identifies it as a branch), immediately make a decision and set the new value of the PC
  - Benefit
    - Since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is need

Control Hazard (10/10)



Branch comparator moved to Decode stage.

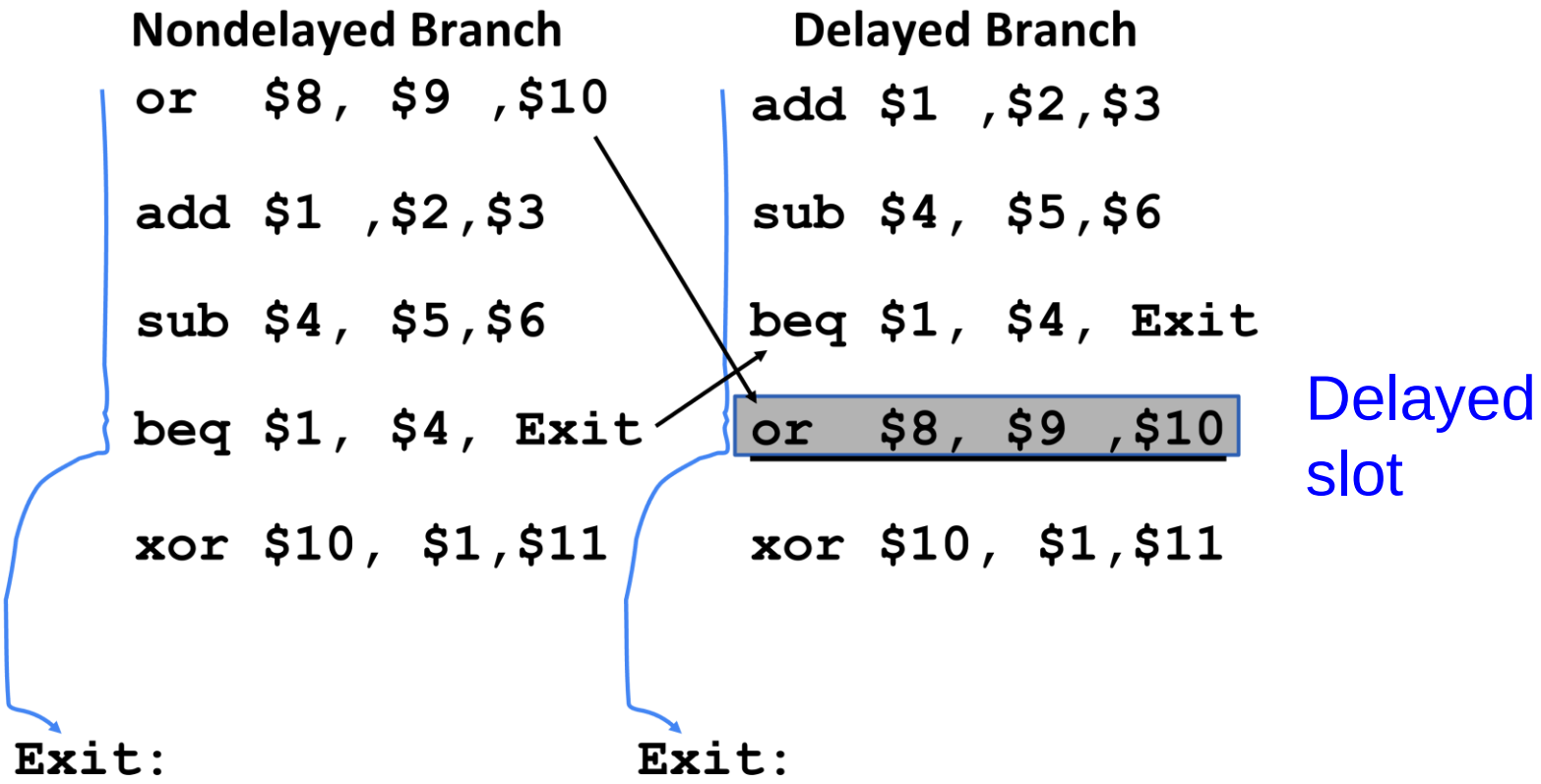
Delayed Branch Slot

- **Optimization #2: Delayed Branch Slot**
  - **Old definition:**
    - if we take the branch, none of the instructions after the branch get execute by accident
  - **New definition:**
    - Whether or not we take the branch, the single instruction immediately following the branch gets executed (called the branch-delay slot)

Delayed Branch Slot

- **Optimization #2: Delayed Branch Slot**
  - We always execute instruction after branch
  - Worst-case:
    - Can always put a no-op in the branch-delay slot
  - Better case:
    - Can find an instruction before the branch which can be placed in the branch-delay slot without affecting flow of the program
    - The compiler must be smart to find instructions to do this

Delayed Branch Slot



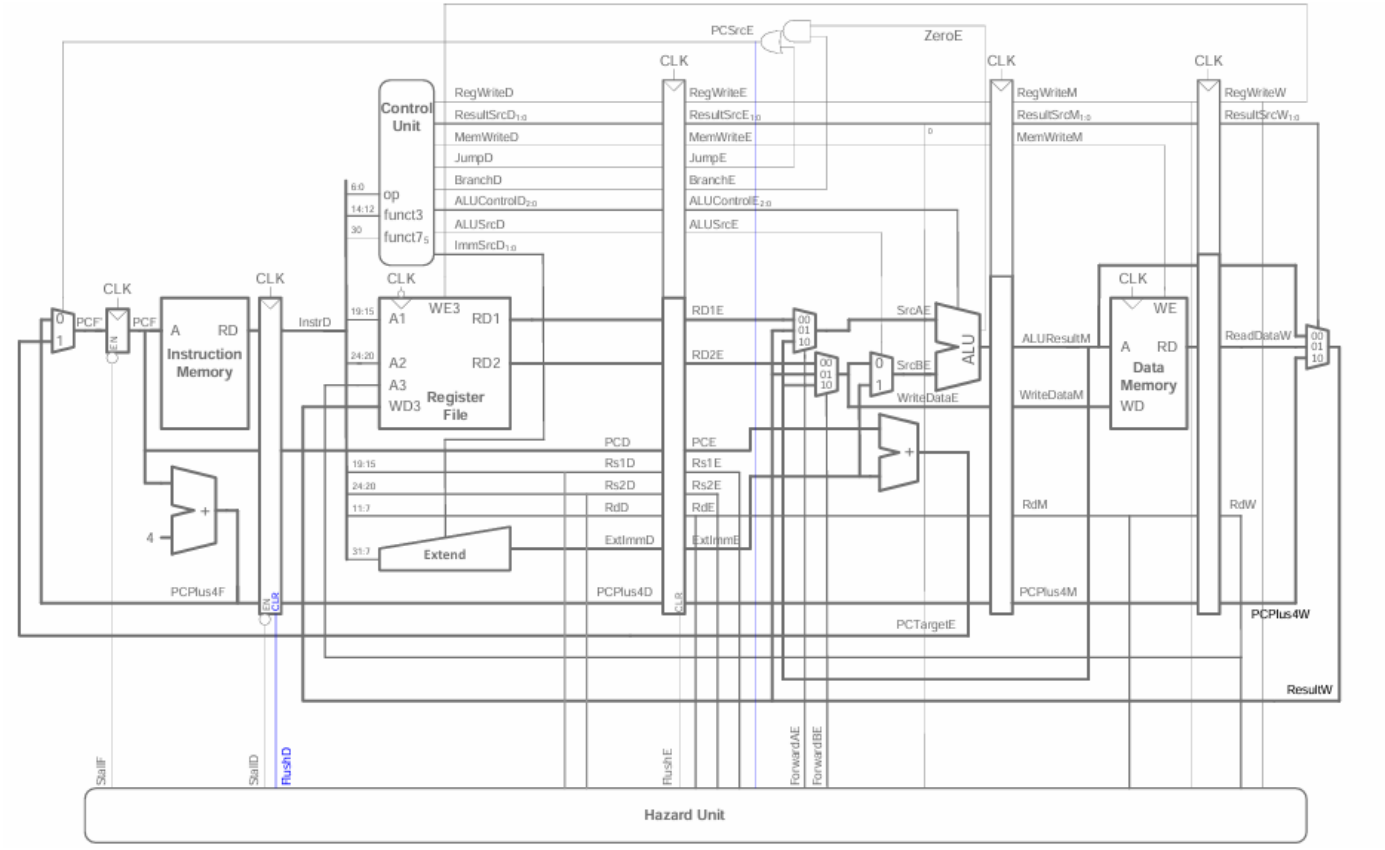
Control Hazards: Flushing Logic

- If branch is taken in execute stage, need to flush the instructions in the Fetch and Decode stages
  - Do this by clearing Decode and Execute Pipeline registers using *FlushD* and *FlushE*
- **Equations:**

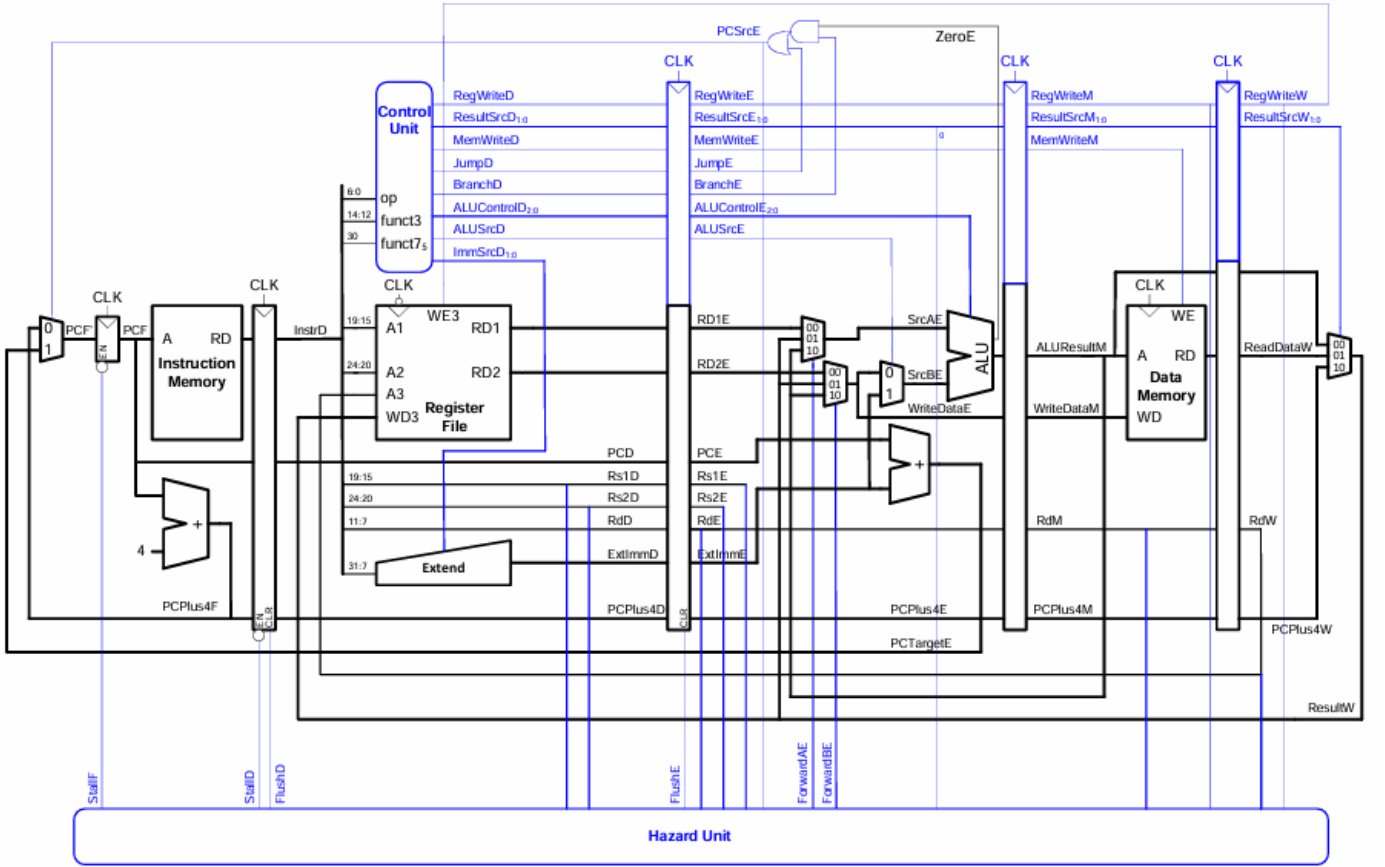
  FlushD = *PCSrcE*

  FlushE = *lwStall* OR *PCSrcE*

Control Hazards: Flushing Hardware



RISC-V Pipelined Processor with Hazard Unit



Summary of Hazard Logic

Data hazard logic (shown for SrcA of ALU):

if ((*Rs1E* == *RdM*) AND *RegWriteM*) AND (Rs1E != 0) // Case 1
ForwardAE = 10
else if ((*Rs1E* == *RdW*) AND *RegWriteW*) AND (Rs1E != 0) // Case 2
ForwardAE = 01
else ***ForwardAE*** = 00 // Case 3

Load word stall logic:

lwStall = ((*Rs1D* == *RdE*) OR (*Rs2D* == *RdE*)) AND $ResultSrcE_0$
StallF = ***StallD*** = *lwStall*

Control hazard flush:

FlushD = *PCSrcE*
FlushE = *lwStall* OR *PCSrcE*

Pipelined Processor Performance Example

- **SPECINT2000 benchmark:**
  - 25% loads
  - 10% stores
  - 13% branches
  - 52% R-type
- **Suppose:**
  - 40% of loads used by next instruction
  - 50% of branches mispredicted
- **What is the average CPI?** (Ideally it's 1, but…)

Pipelined Processor Performance Example

- **SPECINT2000 benchmark:**
  - 25% loads
  - 10% stores
  - 13% branches
  - 52% R-type
- **Suppose:**
  - 40% of loads used by next instruction
  - 50% of branches mispredicted
- **What is the average CPI?**
  - Load CPI = 1 when not stalling, 2 when stalling
    So, $\mathbf{CPI_{lw}} = 1(0.6) + 2(0.4) = 1.4$
  - Branch CPI = 1 when not stalling, 3 when stalling
    So, $\mathbf{CPI_{beq}} = 1(0.5) + 3(0.5) = 2$

Average CPI = $(0.25)(1.4) + (0.1)(1) + (0.13)(2) + (0.52)(1)$ = **1.23**

Pipelined Processor Performance Example

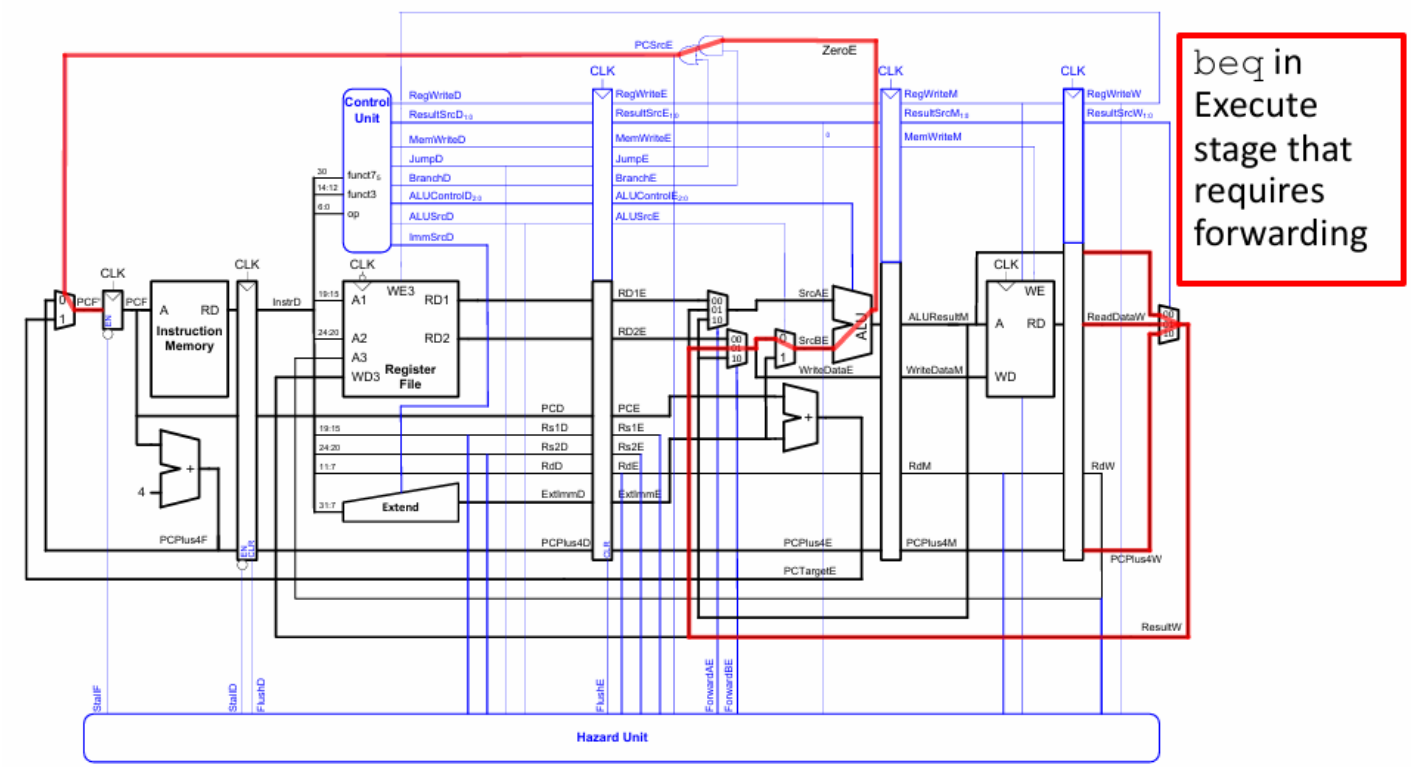
Pipelined processor critical path:

$T_{c\_pipelined}$ = max of

$t_{pcq} + t_{mem} + t_{setup}$	**Fetch**
$2(t_{RFread} + t_{setup})$	**Decode**
$t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND\text{-}OR} + t_{setup}$	**Execute**
$t_{pcq} + t_{mem} + t_{setup}$	**Memory**
$2(t_{pcq} + t_{mux} + t_{RFwrite})$	**Writeback**

- Decode and Writeback stages **both use the register file** in each cycle
- So each stage gets half of the cycle time (T_c**/2**) to do their work
- Or, stated a different way, **2x of their work** must fit in a cycle (T_c)

Pipelined Processor Critical Path



beq in Execute stage that requires forwarding

$$t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND\text{-}OR} + t_{setup}$$

Execute

Pipelined Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{AND\text{-}OR}$	20
ALU	t_{ALU}	120
Decoder (control unit)	t_{dec}	35
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

Cycle time: $T_{c3} = t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND\text{-}OR} + t_{setup}$

$= (40 + 4(30) + 120 + 20 + 50)$ ps $=$ **350 ps**

Pipelined Performance Example

Program with 100 billion instructions

Execution Time $= (\#\text{ instructions}) \times \text{CPI} \times T_c$

$= (100 \times 10^9)(1.23)(350 \times 10^{-12})$

= 43 seconds

Processor Performance Comparison

Processor	Execution Time (seconds)	Speedup (single-cycle as baseline)
Single-cycle	75	1
Multicycle	144	0.5
Pipelined	43	1.7

Conclusion

- Pipelining Execution
- Pipelining Hazard
  - Structural Hazard
  - Data Hazard
  - Control Hazard