National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 6: Pipelining Processor

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

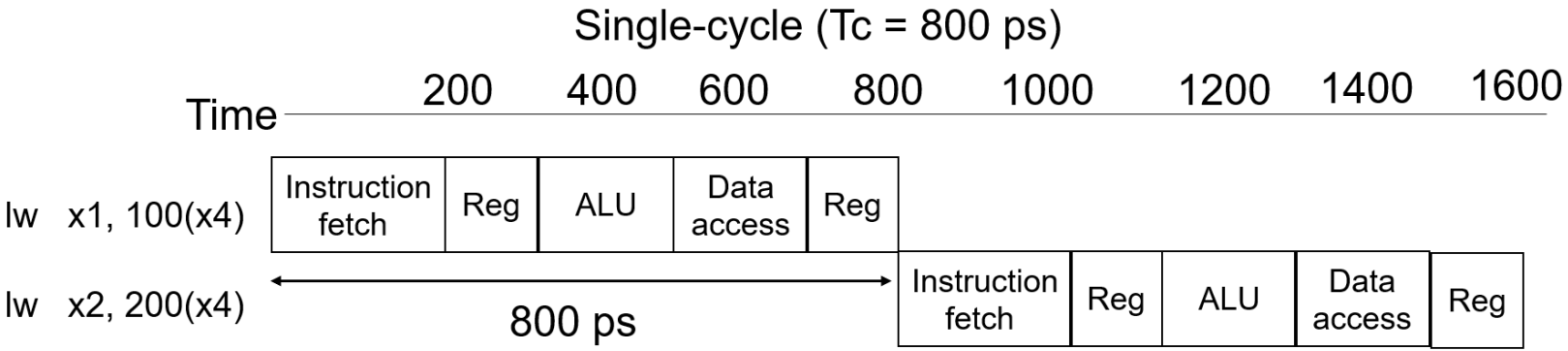
Performance on Single-Cycle Processor

- Assume time for stages is
  - 100 ps for register read or write
  - 200 ps for other stages
  - Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
sw	200 ps	100 ps	200 ps	200 ps		700 ps
R-format	200 ps	100 ps	200 ps		100 ps	600 ps
beq	200 ps	100 ps	200 ps			500 ps

Performance on Single-Cycle Processor

- Why a single-cycle implementation is not used today?
  - The longest possible path in processor determines the clock cycle
  - Although CPI is 1, the overall performance is poor, since the clock cycle is too long

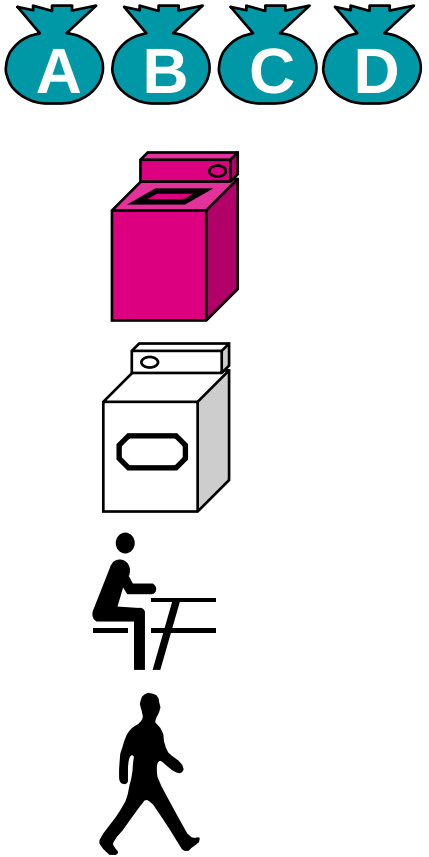


Performance on Single-Cycle Processor

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory -> register file -> ALU -> data memory -> register file
- No feasible to vary period for different instructions
- We will improve performance by pipelining

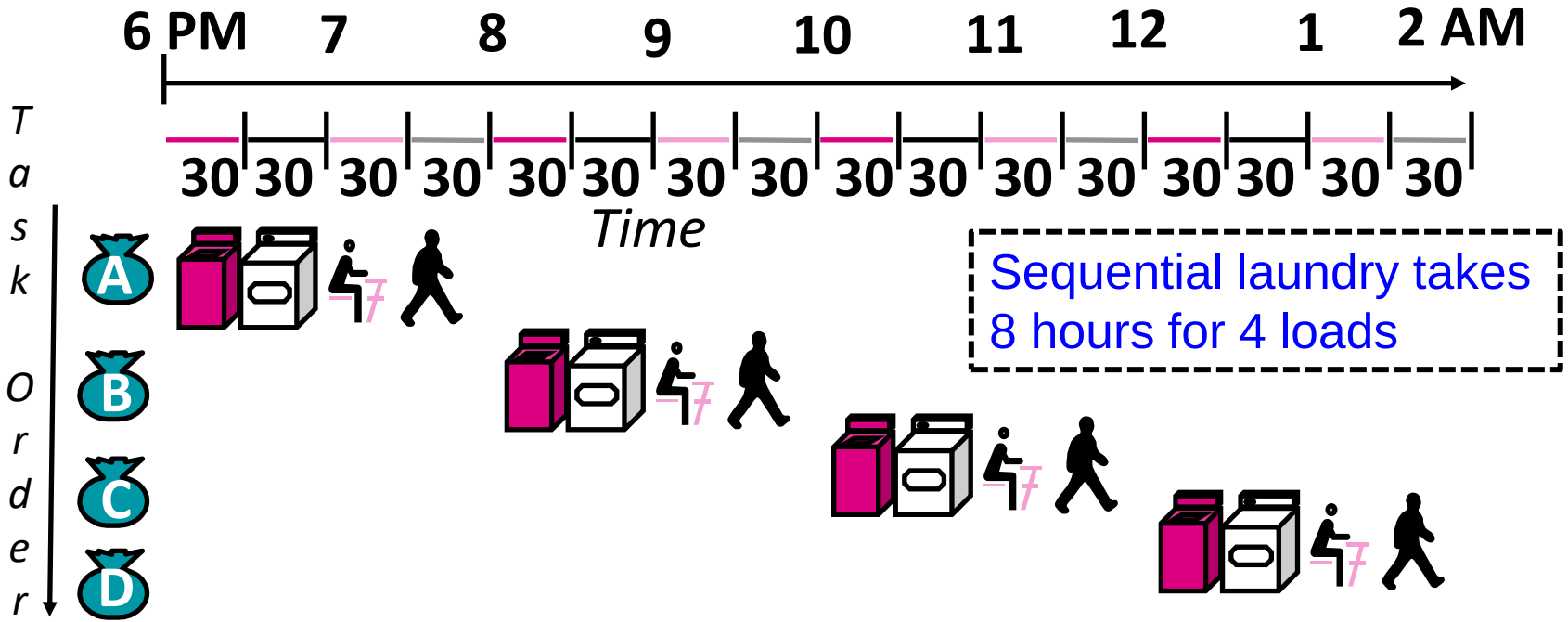
Pipelining

- 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



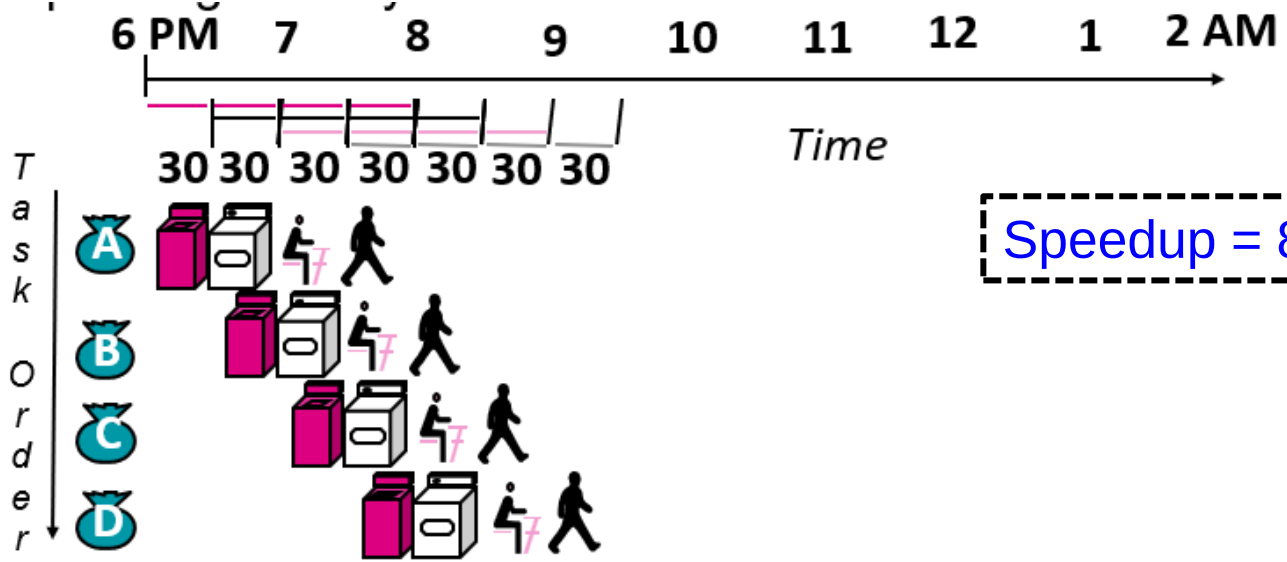
Pipelining

- Sequential Laundry



Pipelining

- Pipelining laundry
  - Overlapping execution
  - Parallelism improves performance



Pipelining laundry takes 3.5 hours for 4 loads !

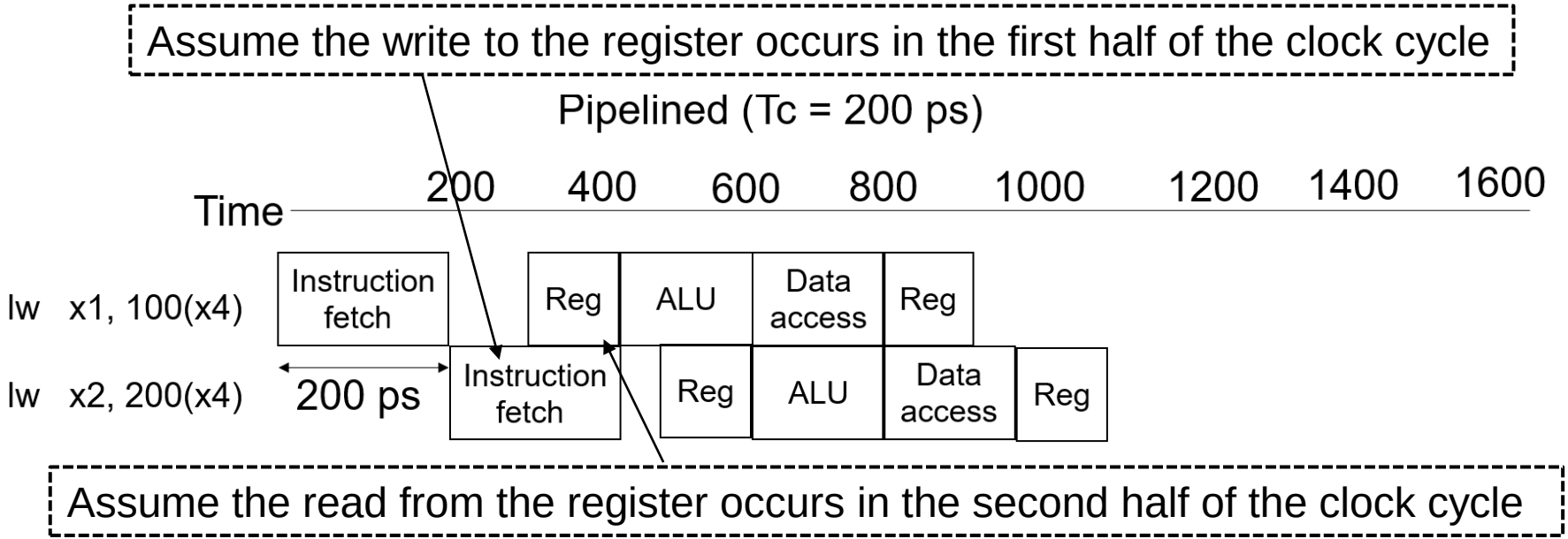
Speedup = 8/3.5 = 2.3

Pipeline on RISC-V

- Five stages, one step per stage
  - 1. **IF**: Instruction fetch from memory
  - 2. **ID**: Instruction decode & register read
  - 3. **EX**: Execute operation or calculate address
  - 4. **MEM**: Access memory operand
  - 5. **WB**: Write result back to register
  - Add pipeline registers between stages

Pipeline Performance

- A fourfold speed-up on average time between instructions, from 800 ps down to 200 ps



Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$ = $\frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$
  - If not balanced, speedup is less
  - Speedup due to increased instruction throughput
    - Latency (time for each instruction) does not decrease
    - Instruction throughput is the important metric because real programs execute billions of instructions

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

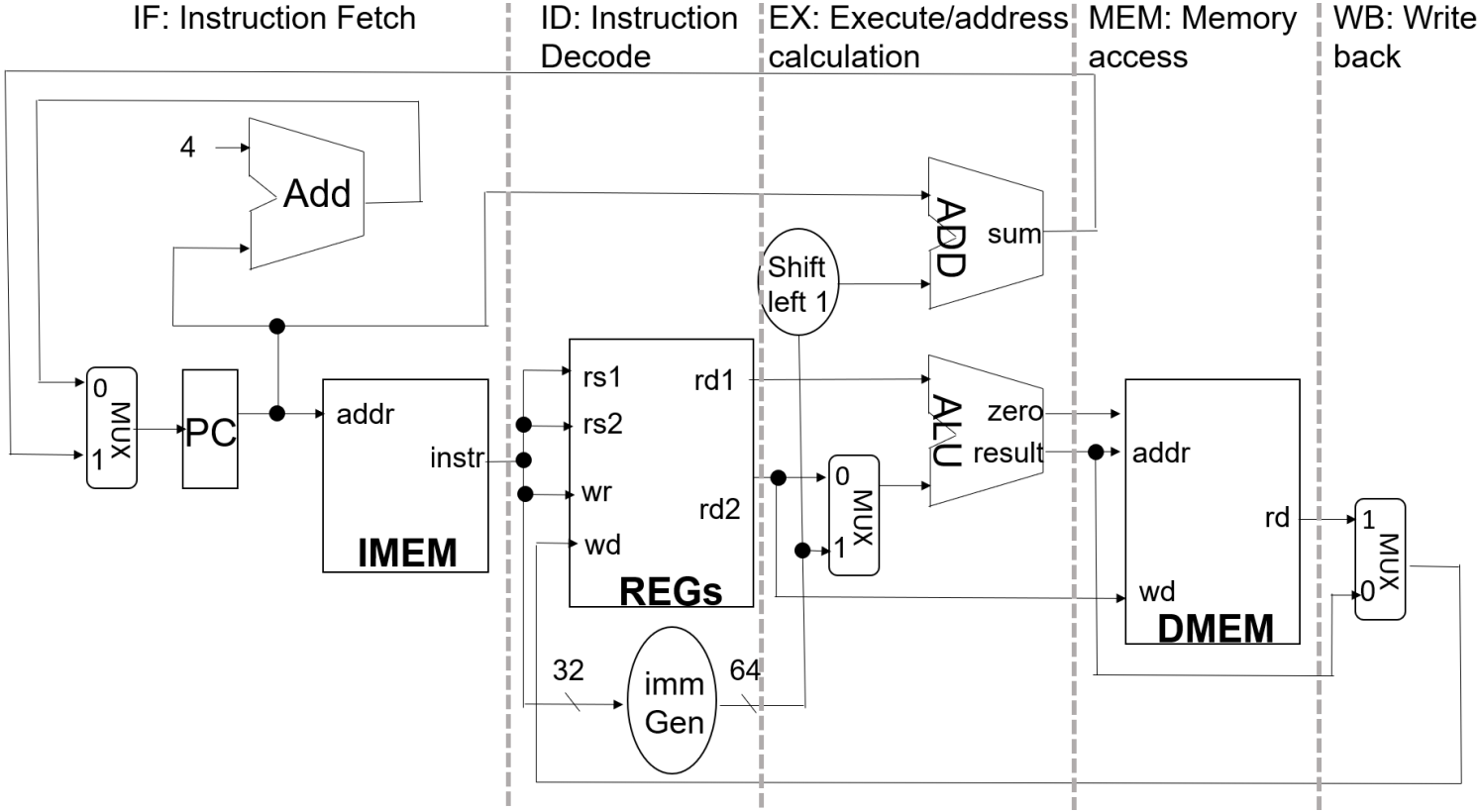
- Which of the following statement(s) is/are True or False?
  - (a) Thanks to pipelining, I have reduced the time it took me to wash my shirt.

    (b) Longer pipelines are always a win (since less work per stage & a faster clock)

Takeaway Questions

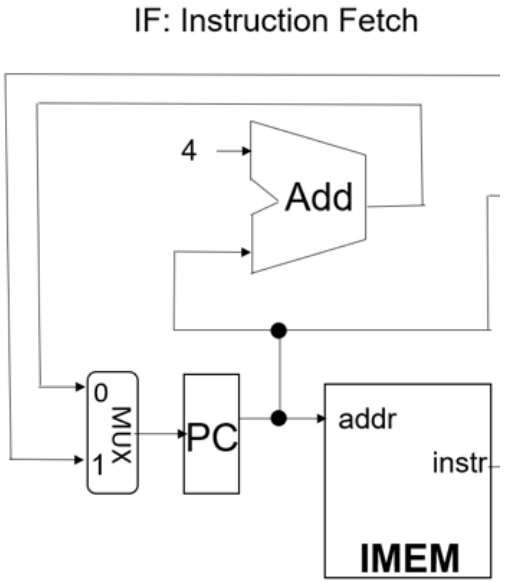
- Which of the following statement(s) is/are True or False?
  - (a) Thanks to pipelining, I have reduced the time it took me to wash my shirt. (False)
    - **Throughput better, not execution time**
  - (b) Longer pipelines are always a win (since less work per stage & a faster clock) (False)
    - **longer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

RISC-V Pipelined Datapath



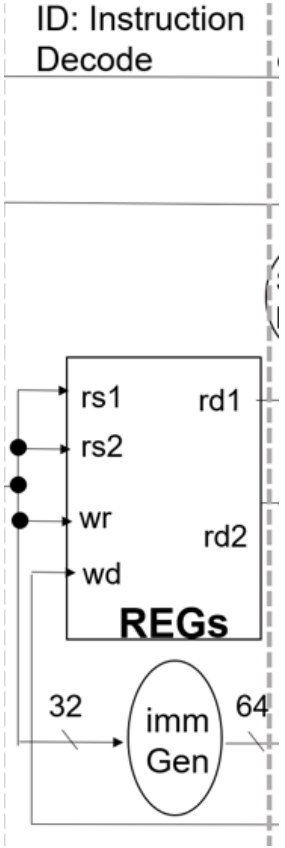
RISC-V Pipelined Datapath

- Instruction fetch (IF)
  - The control signals to read instruction memory
  - To write the PC are always asserted



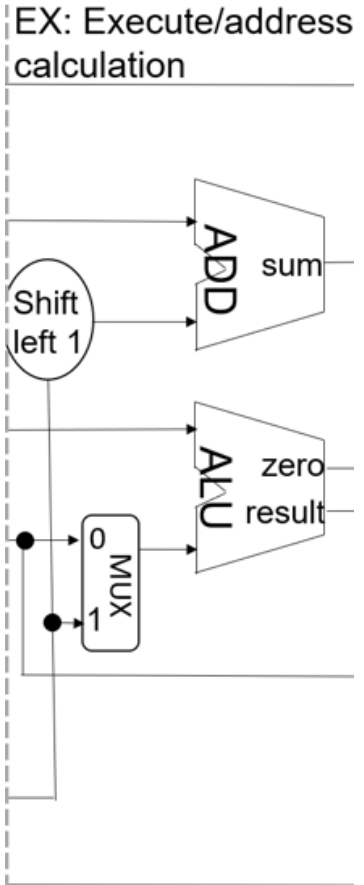
RISC-V Pipelined Datapath

- Instruction decode/register file read (ID)
  - The two source registers are always in the same location in the RISC-V instruction formats



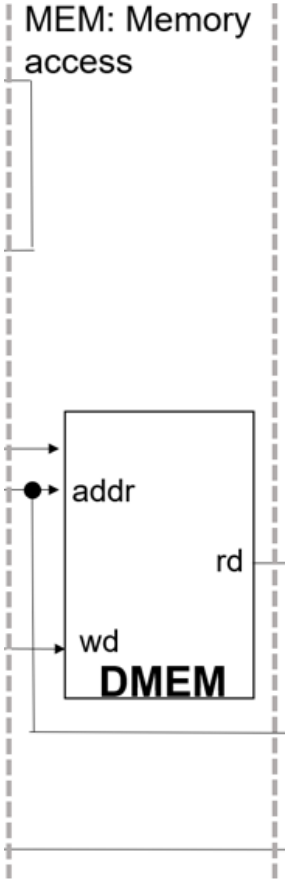
RISC-V Pipelined Datapath

- Execution/address calculation (EX)
  - The signals to be set are ALUOp and ALUSrc
  - The signal select the ALU operation
  - Either read data 2 or a signed extended immediate as inputs to the ALU



RISC-V Pipelined Datapath

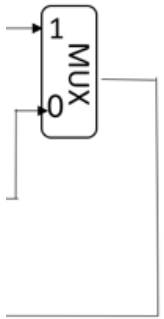
- Memory access(MEM)
  - The control lines set in this stage are
    - Branch, MemRead, MemWrite
    - The branch if equal, load, and store instructions set these signals



RISC-V Pipelined Datapath

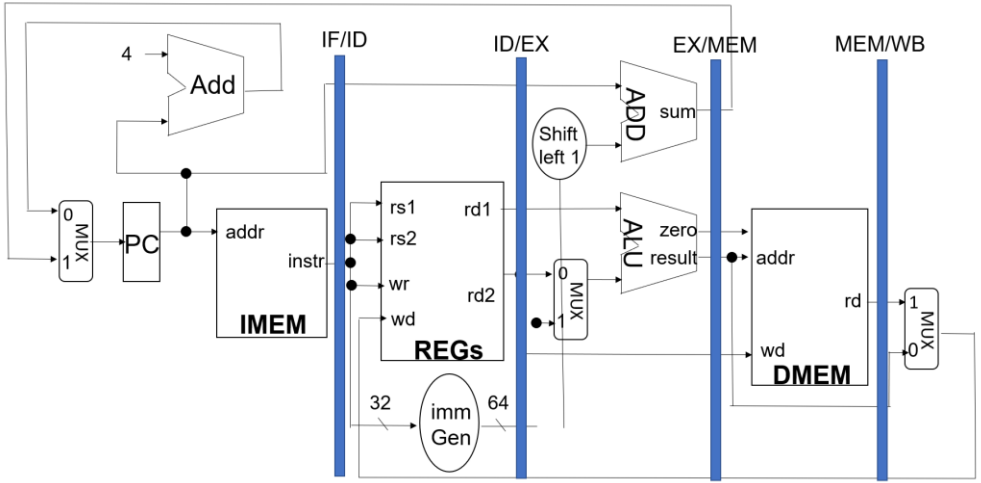
WB: Write back

- Write-Back (WB)
  - The two control lines are
    - MemtoReg : Decide between sending the ALU result or the memory value to the register file
    - RegWrite: writes the chosen value



Pipeline Registers

- Need registers between stages
  - To hold information produced in previous cycle
  - Register read must arrive at the same time as result
  - Register written on rising edge of CLK

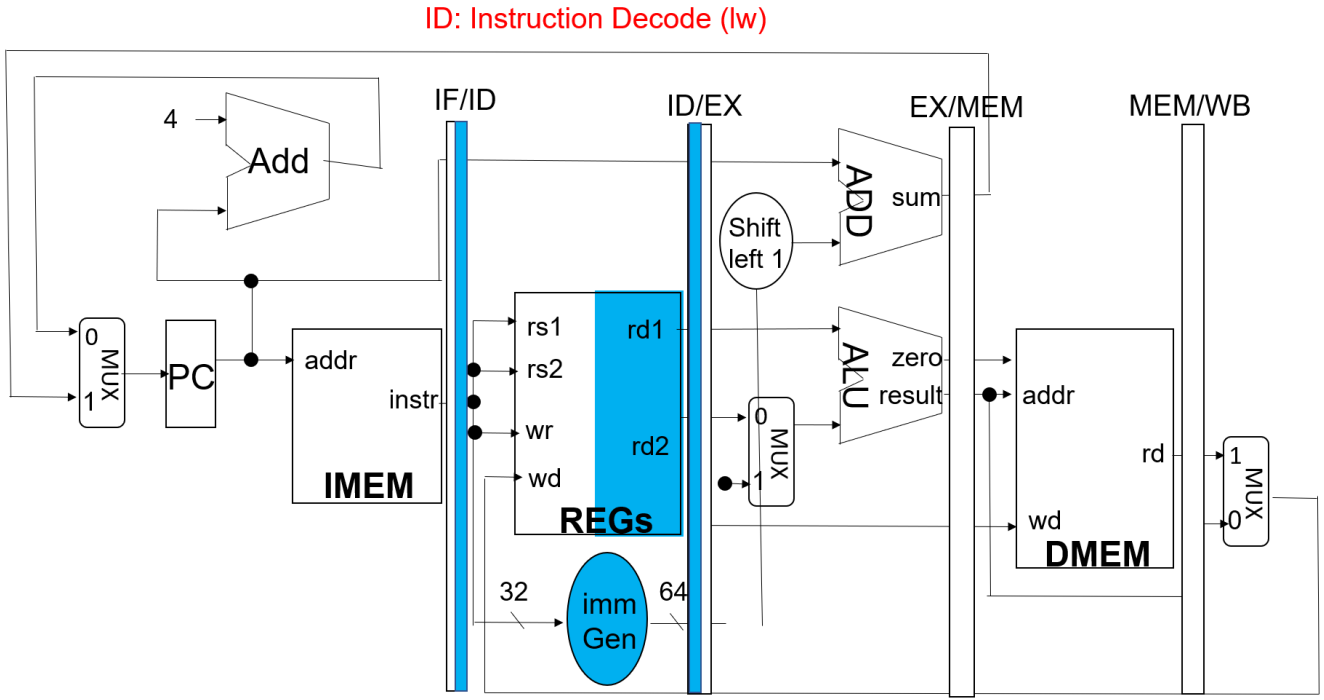


Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - “Single-clock-cycle” pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used

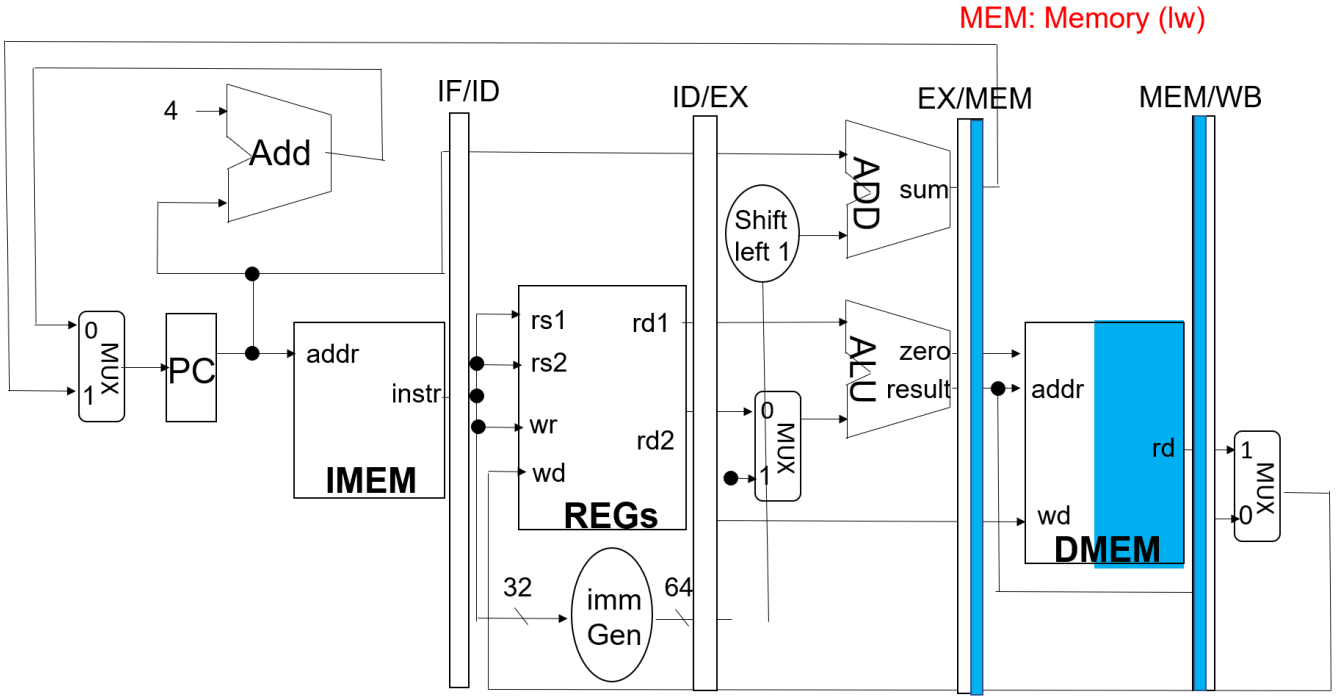
Pipeline Operation (lw)

- ID stage for load



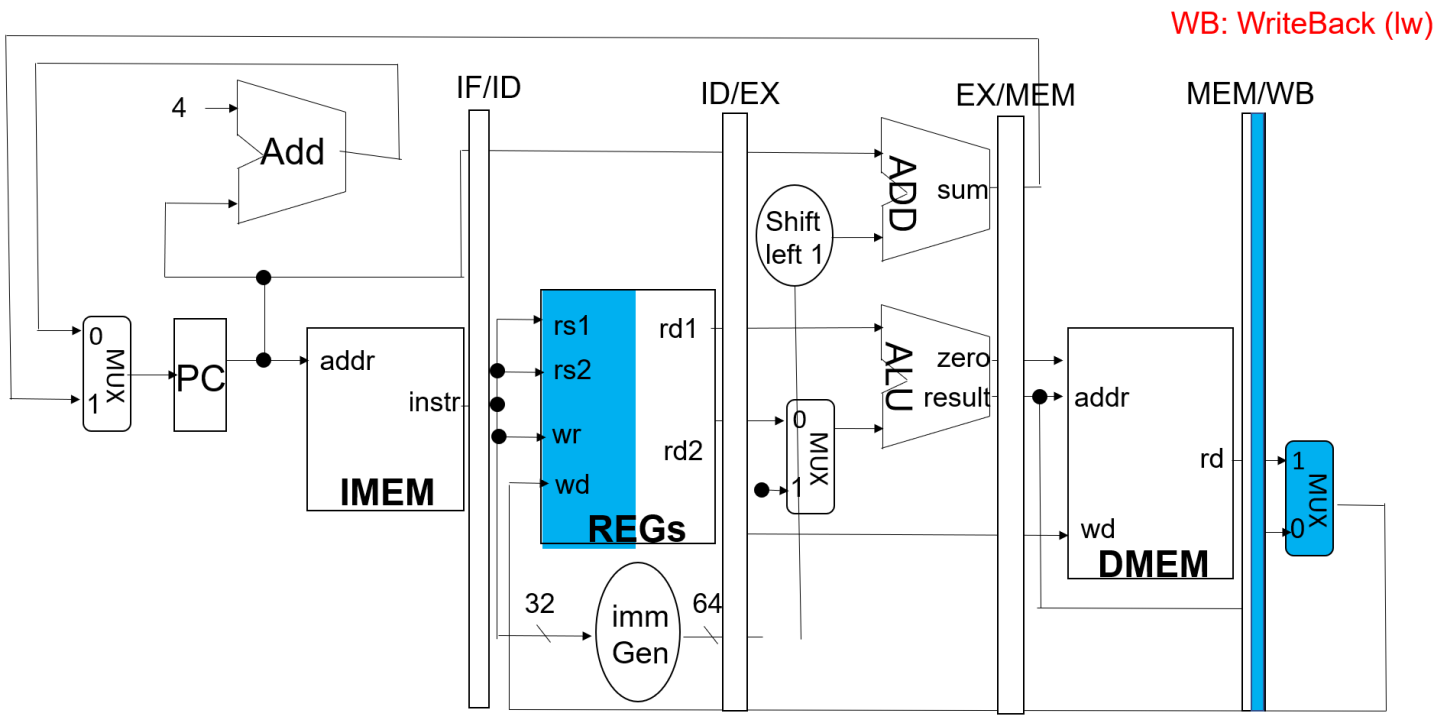
Pipeline Operation (lw)

- MEM stage for load



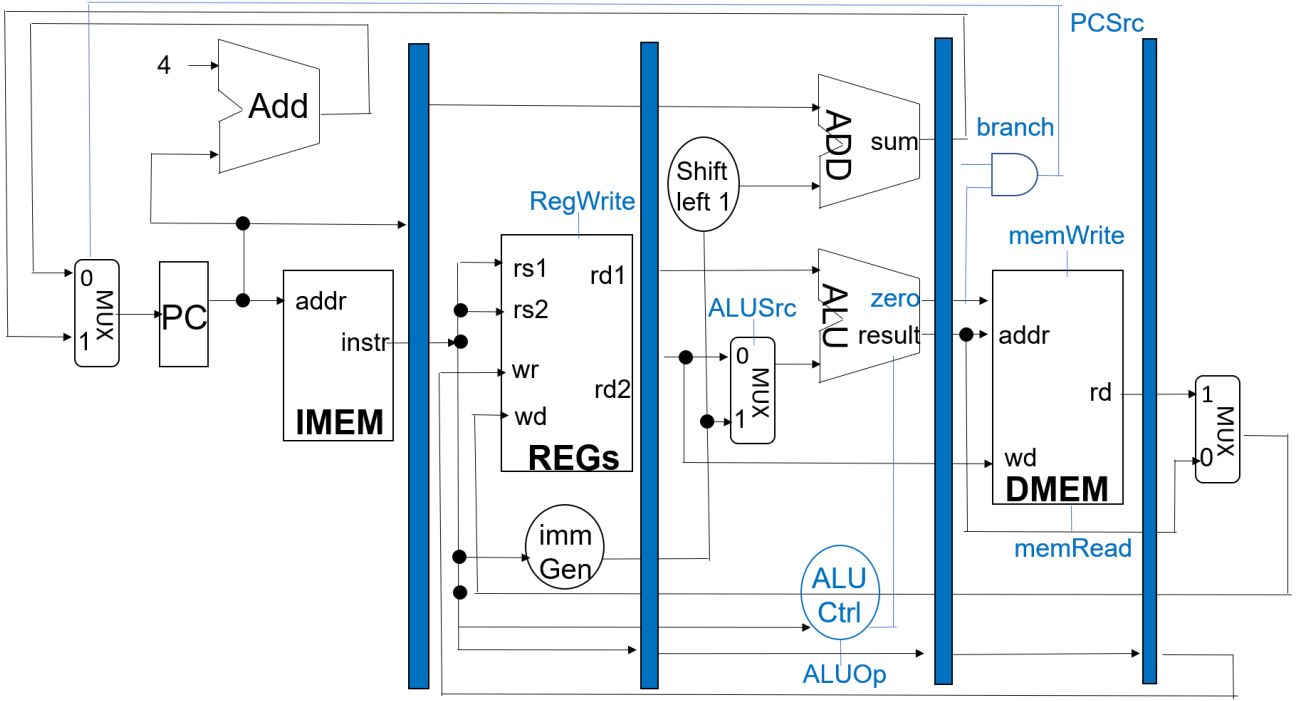
Pipeline Operation (lw)

- WB stage for load



Pipelined Control (Simplified)

- Control signals derived from instruction



Pipelining Hazard

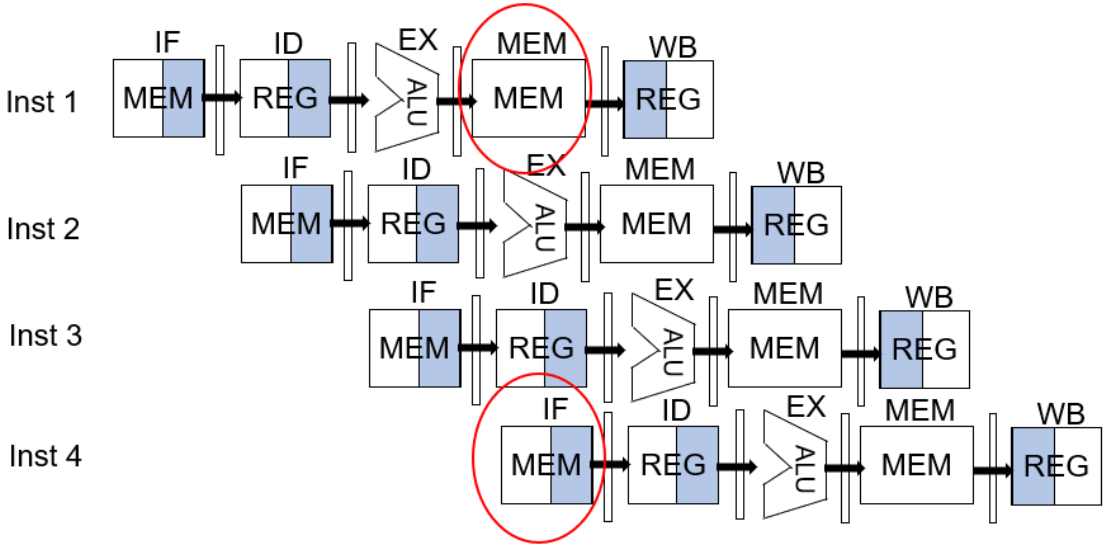
- Limits to pipelining
  - Hazards result in pipeline “**stalls**” or **“bubbles”**
  - **Structural hazards:**
    - Multiple instructions in the pipeline compete for access to a single physical resource
  - **Control hazards:**
    - Pipelining of branches causes later instruction fetches to wait for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

Structural Hazard

- Conflict for use of a resource
- In RISC-V pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to stall for that cycle
  - Would cause a pipeline “bubble”
- Pipeline datapaths require
  - Separate instruction/data memories
  - Or separate instruction/data cache

Structural Hazard

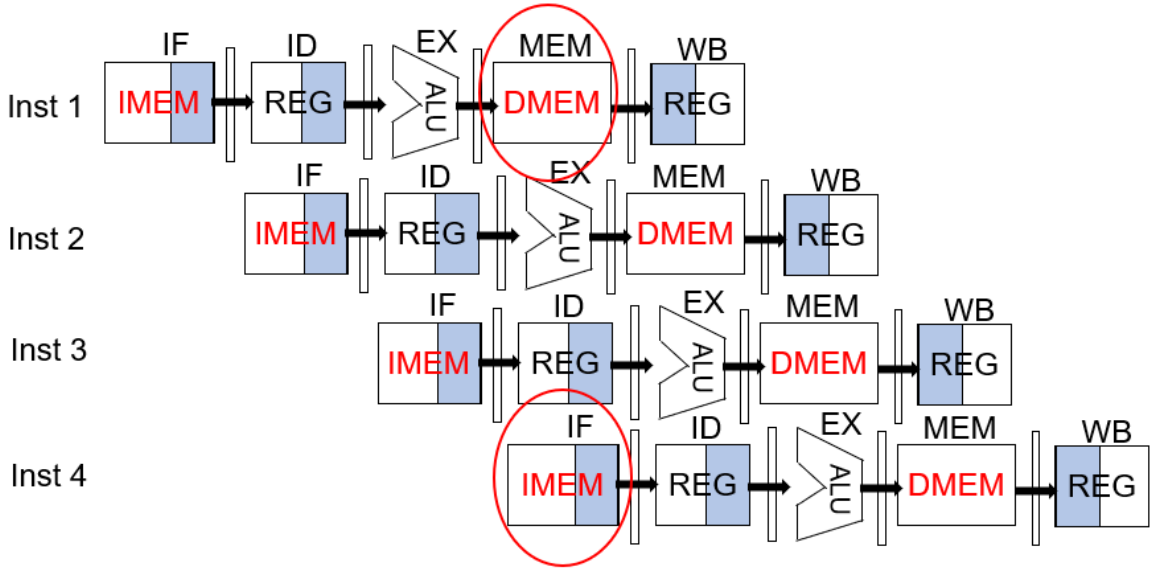
- **Structural Hazard #1: Single Memory**
  - The first instruction is accessing data from memory while the fourth instruction is fetching instruction from that same memory



Read the same memory twice in the same clock cycle

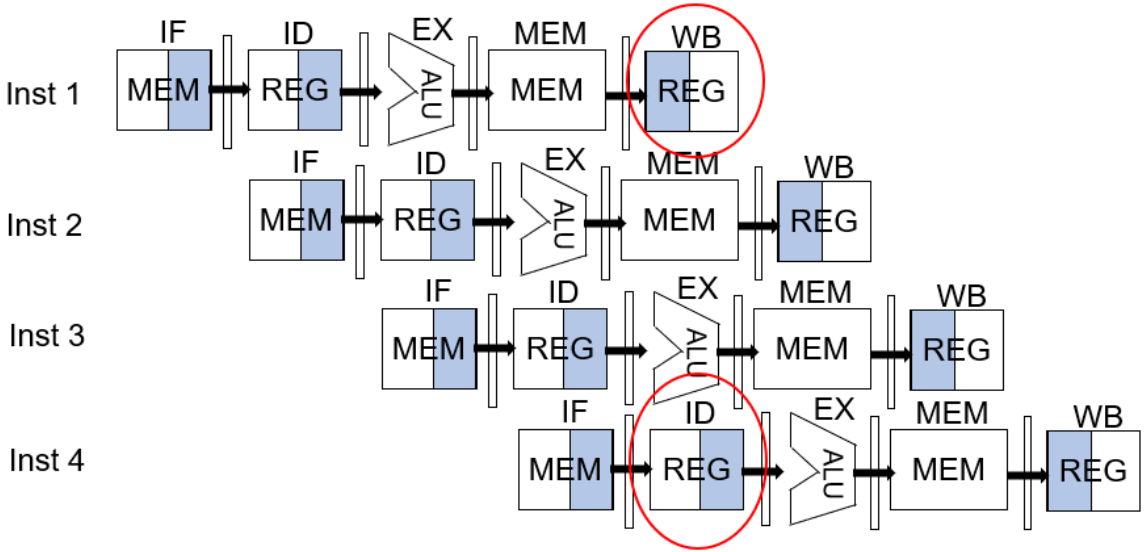
Structural Hazard

- **Structural Hazard #1: Single Memory**
  - **Solution**
    - Separate data (DMEM) and instruction memory (IMEM)



Structural Hazard

- **Structural Hazard #2: Registers**
  - Instruction 1 and instruction 4 read and write registers simultaneously

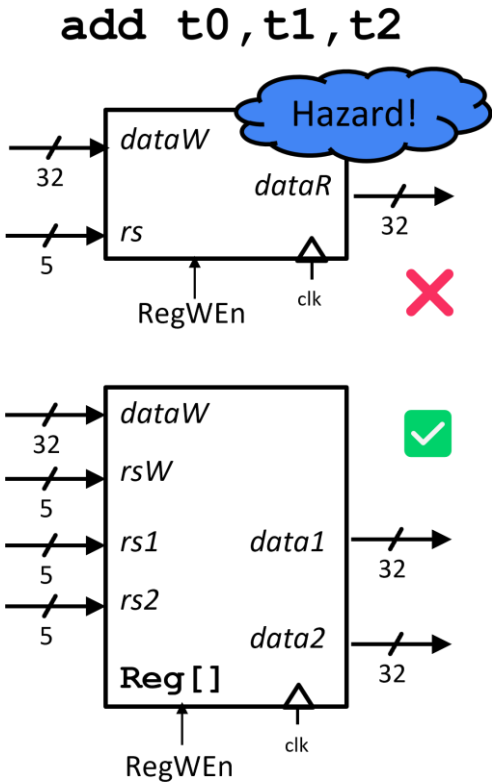


Structural Hazard

- **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

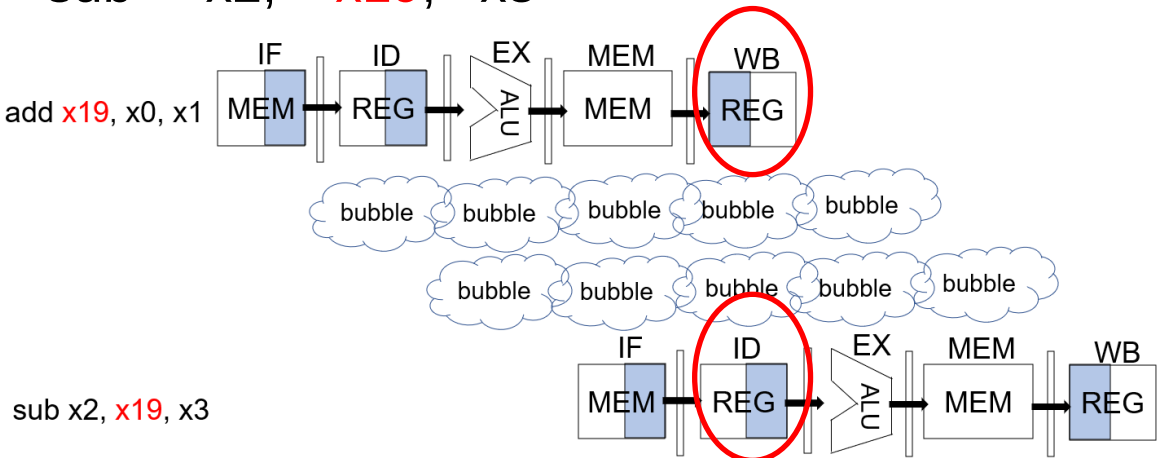
Structural Hazard

- **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**
  - An instruction depends on completion of data access by a previous instruction
    - add x19, x0, x1
      sub x2, x19, x3

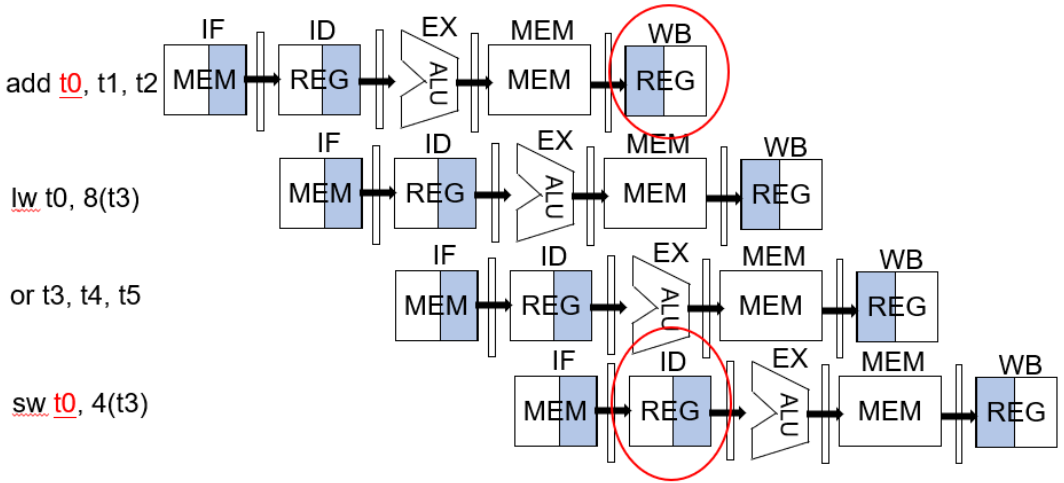


Data Hazard

- **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

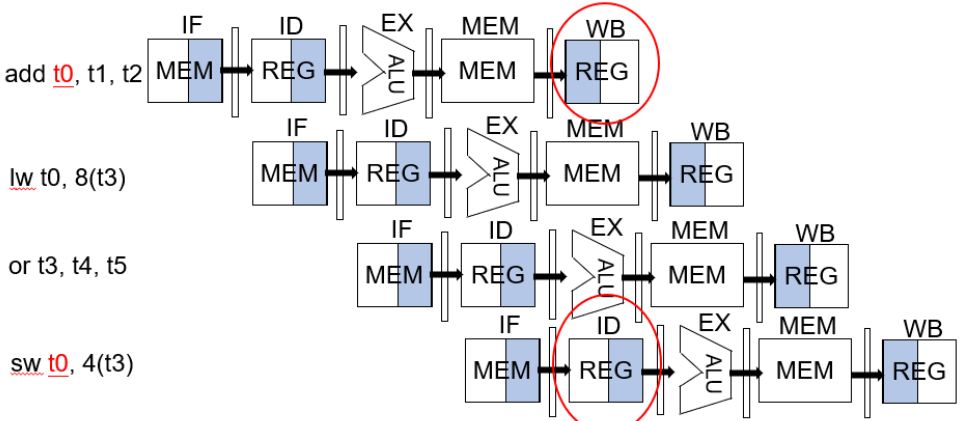
Data Hazard: REG

- **Register Access**
  - The same register is written and read in one cycle:
  - WB must write value before ID reads new value
  - No structural hazard – Separate ports allows simultaneous R/W



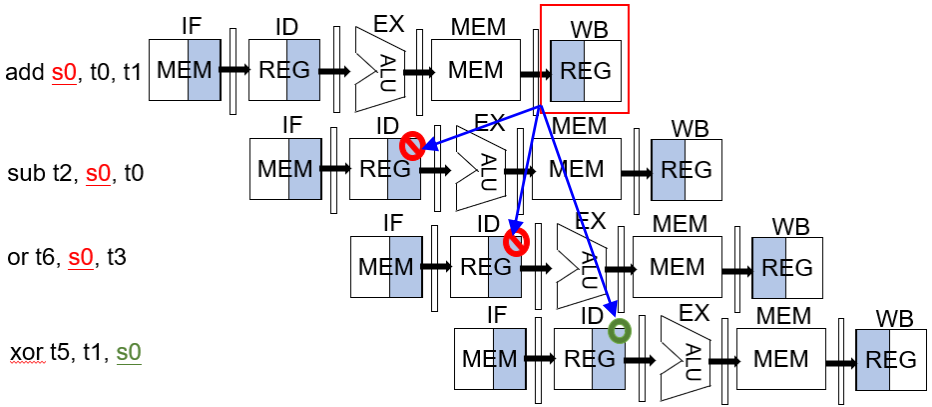
Data Hazard: REG

- **Register Access**
  - Solution: RegFile HW should write-then-read in the same cycle
  - Exploit high speed of RegFile (100 ps + 100 ps)
  - Might not always be possible to write-then-read in the same cycle., e.g. in high-frequency designs



Data Hazard: ALU

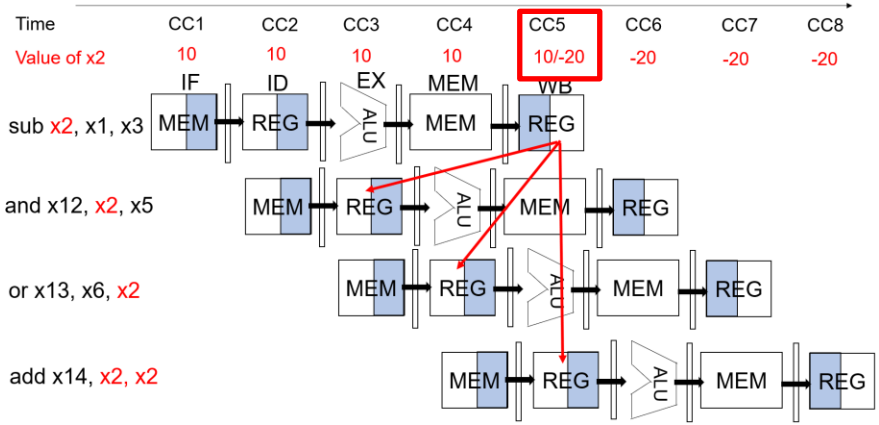
- **ALU Result**
  - Problem: Instruction depends on WB's RegFile write from previous instruction
  - sub, or's ID reads old value of s0 and calculates wrong result
  - xor gets the right value; RegFile is write-then-read



Data Hazard: Forward versus Stalling

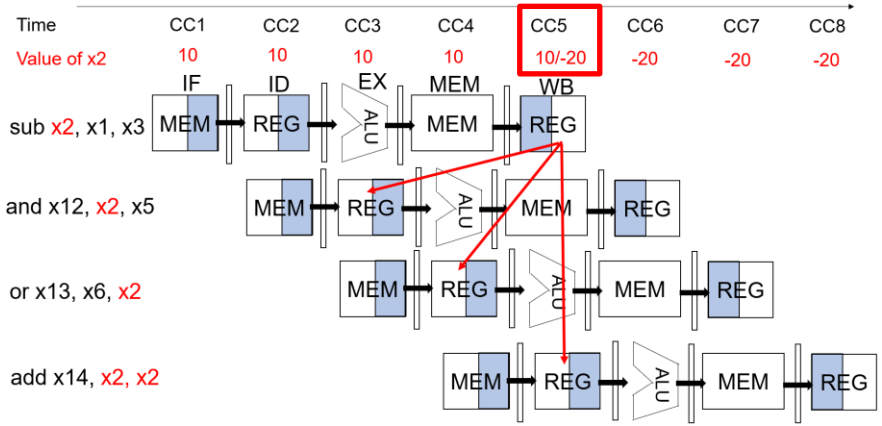
- Let's look at a sequence with many dependences

```
sub   x2, x1, x3     // Register x2 written by sub
and   x12, x2, x5    // x2 depends on sub
or    x13, x6, x2    // x2 depends on sub
add   x14, x2, x2    // 1st (x2) & 2nd (x2) depends on sub
sw    x15, 100(x2)   // Base (x2) depends on sub
```



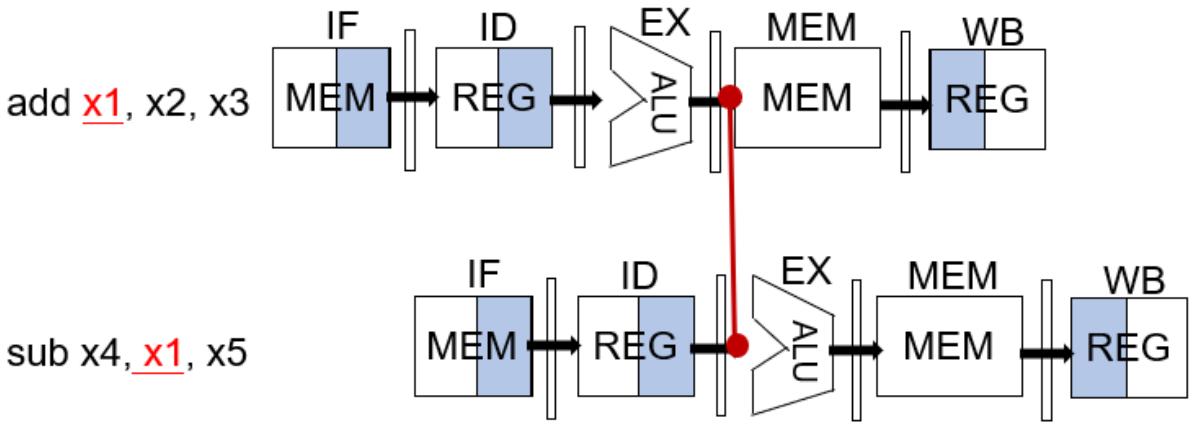
Data Hazard: Forward versus Stalling

- The type of hazards
  - The `sub-and` is EX/MEM.RegisterRd = ID/EX.RegisterRs1 = x2
  - The `sub-or` is EX/MEM.RegisterRd = ID/EX.RegisterRs2 = x2
  - The `sub-add` are not hazards
    - REG supplies the proper data during ID stage of `add`



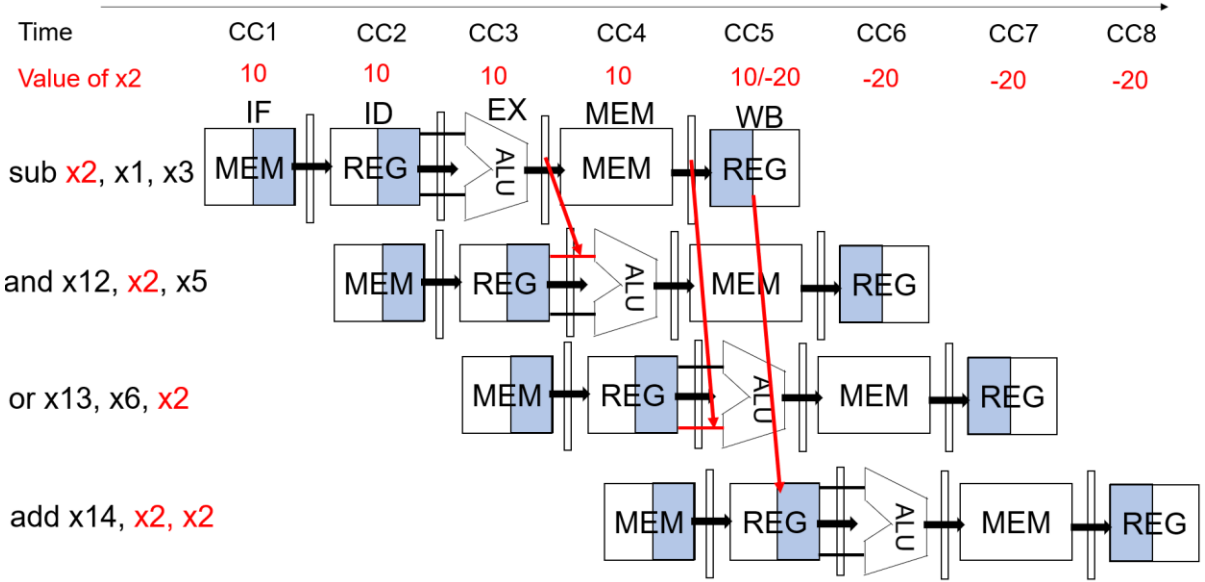
Forwarding (Bypassing)

- Use result when it is computed
  - Don’t wait for it to be stored in a register (the x1 value is ready after the completion of the EX stage)
  - Require extra connections in the datapath



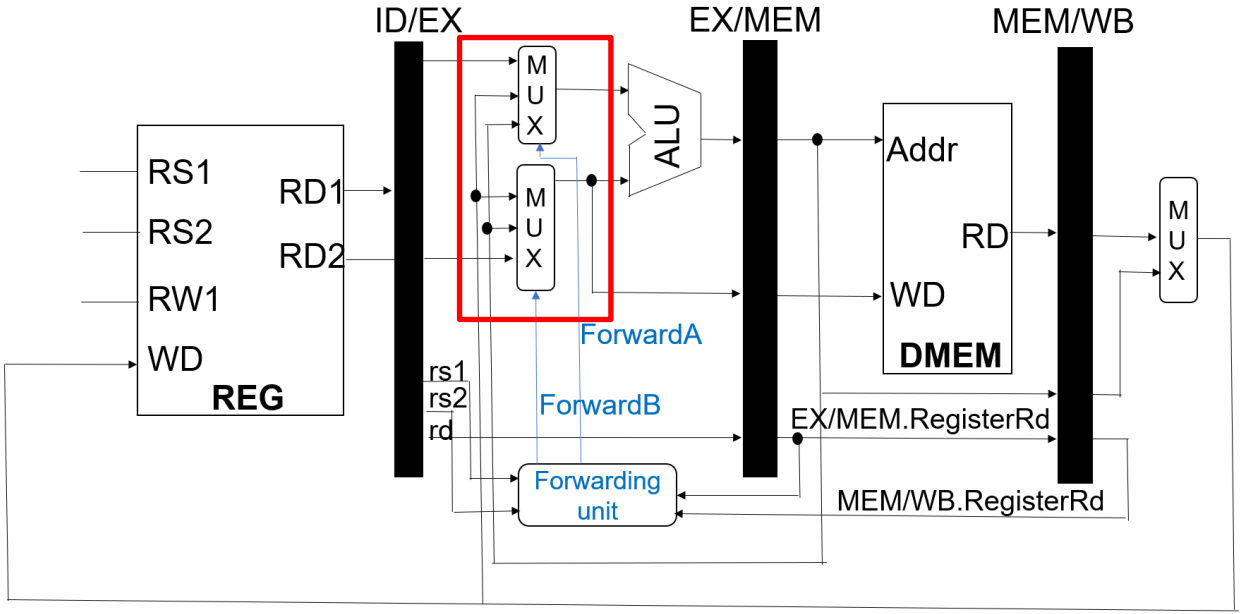
Forwarding (Bypassing)

- Forward proper data from pipeline registers
  - The pipeline registers hold the data to be forwarded



Forwarding Unit

- Add MUXs to the input of ALU with the proper control
  - The forwarding control will be in the EX stage



Detecting the Need to Forward

- EX hazard
  - If (EX/MEM.RegWrite)
    and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs1) ForwardA = 10
  - If (EX/MEM.RegWrite)
    and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs2) ForwardB = 10

We want to avoid forwarding possible nonzero result in x0.
E.g. addi x0, x1, 2

Detecting the Need to Forward

- MEM hazard
  - If (MEM/WB.RegWrite)
    and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs1) ForwardA = 01
  - If (MEM/WB.RegWrite)
    and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs2) ForwardB = 01

Forwarding Conditions

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Double Data Hazard

- Consider the sequence

  add x1, x1, x2
  add x1, x1, x3
  add x1, x1, x4

- Potential data hazards
  - Between the result of the instruction in the WB and MEM stage
  - And the source operand of the instruction in the ALU stage
- The result should be forwarded from the MEM stage
  - The result in the MEM stage is more recent result

Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite AND (MEM/WB.RegisterRd ≠ 0)
    AND not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    AND (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs1))
    AND (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
  - if (MEM/WB.RegWrite AND (MEM/WB.RegisterRd ≠ 0)
    AND not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    AND (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs2))
    AND (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01

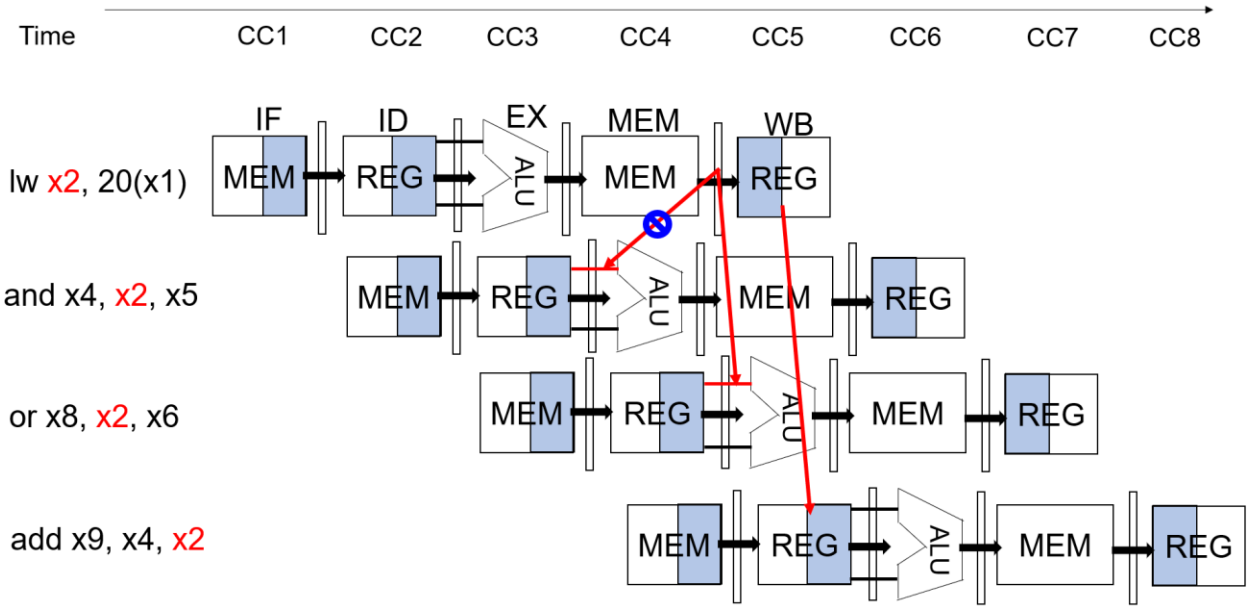
Data Hazards and Stalls

- Data forwarding does not work when
  - An instruction tries to read a register following a load instruction that writes the same register
- Load-use hazard when
  - If (ID/EX.MemRead and
    ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
    (ID/EX.RegisterRd = IF/ID.RegisterRs1)))
    Stall the pipeline
- If detected, stall and insert bubble

```
lw      x1, 0(x2)
sub     x4, x1, x5
```

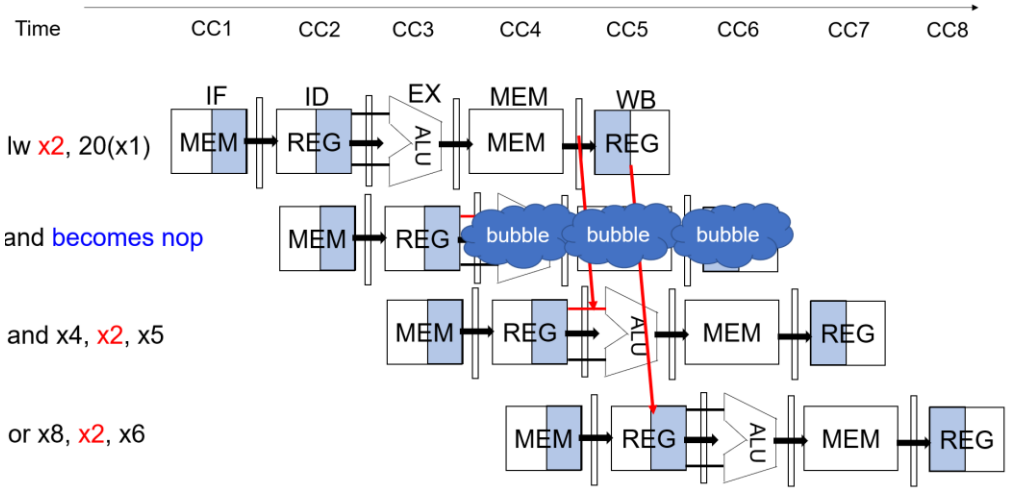
Load-Use Data Hazard

- Data forwarding cannot go backward in time
  - The `lw-and` data dependence cannot be solved by forwarding



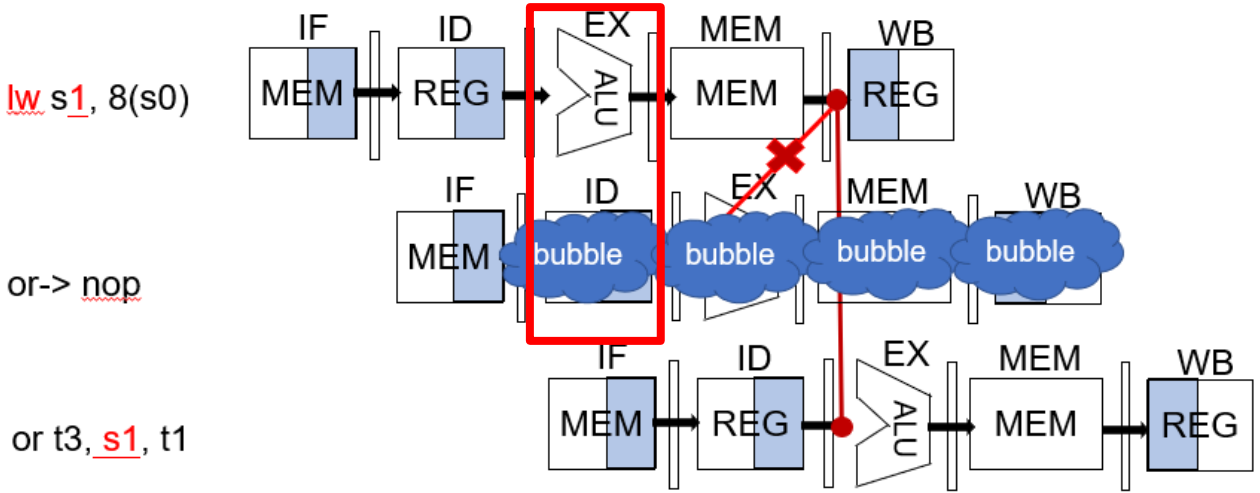
Load-Use Data Hazard

- Identify the hazard in the ID stage
  - Insert nop instruction in the pipeline, delay one cycle
  - The hazard forces the `and` and `or` instructions to repeat in clock cycle 4 what they did in clock cycle 3



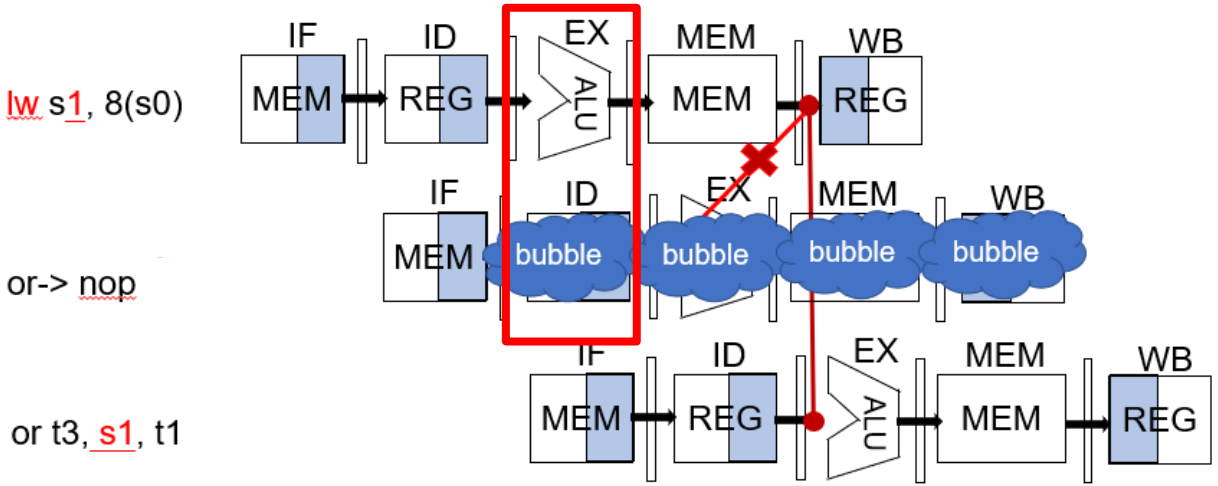
How to stall the Pipeline

- Stall logic
  - Is either source register in the ID stage the same as the destination register in the EX stage &&
  - Is the instruction in the EX stage is a lw?



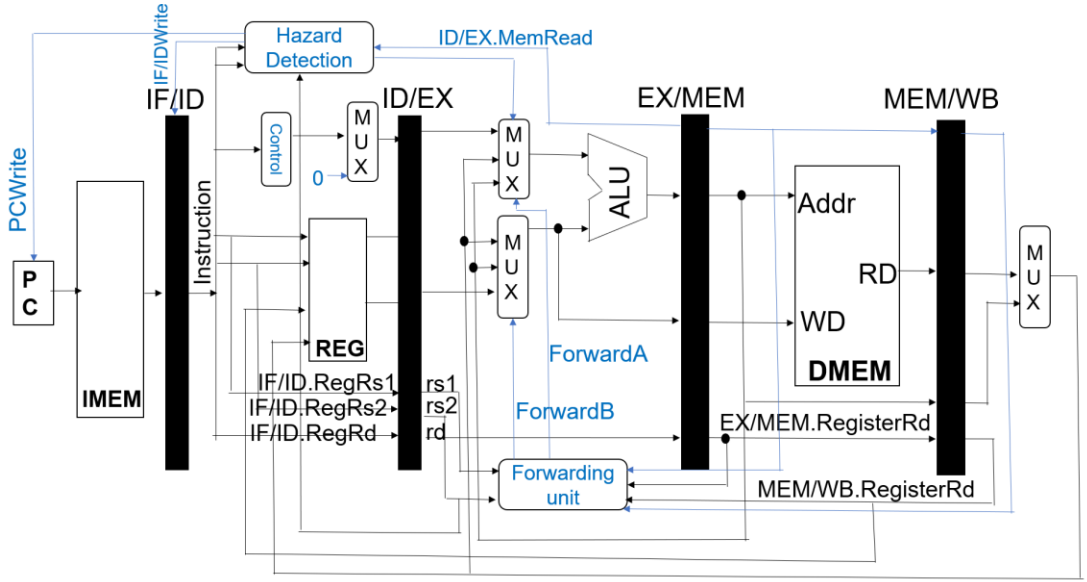
How to stall the Pipeline

- Stall logic
  - Stall the IF and ID stages, flush the EX stage
    - lwstall = ((rs1D == rdE) && (rs2D == rdE)) && $ResultSrcE_0$
    - StallF = StallD = FlushE = lwStall



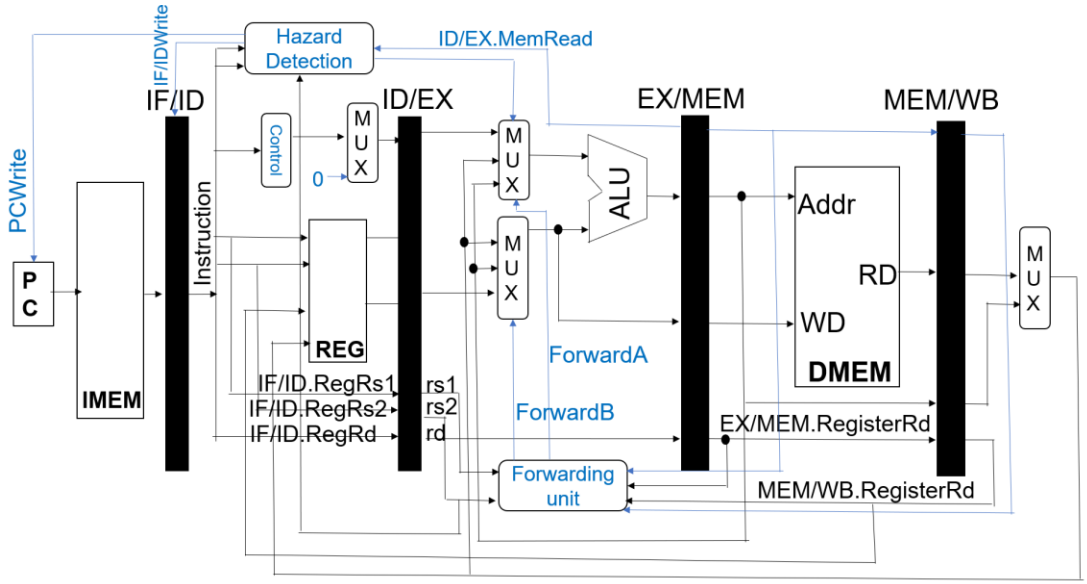
Datapath with Hazard Detection

- Hazard detection unit
  - Controls the writing of the PC and IF/ID registers + MUX that chooses between the real control values and all 0s



Datapath with Hazard Detection

- Hazard detection unit
  - Stalls and de-asserts the control fields if the load-use hazard test is true

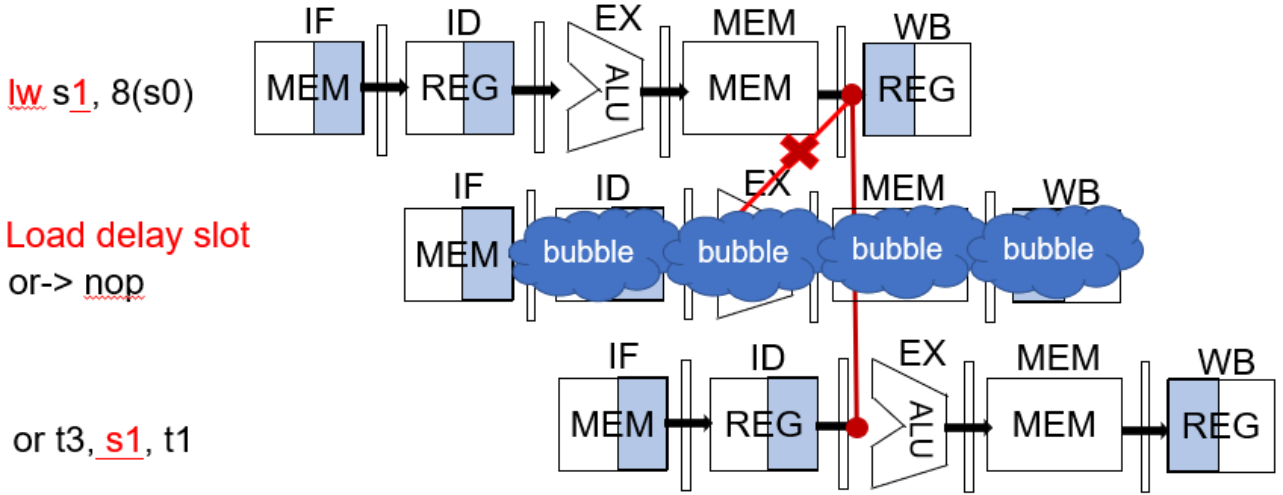


Stalls and Performance

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

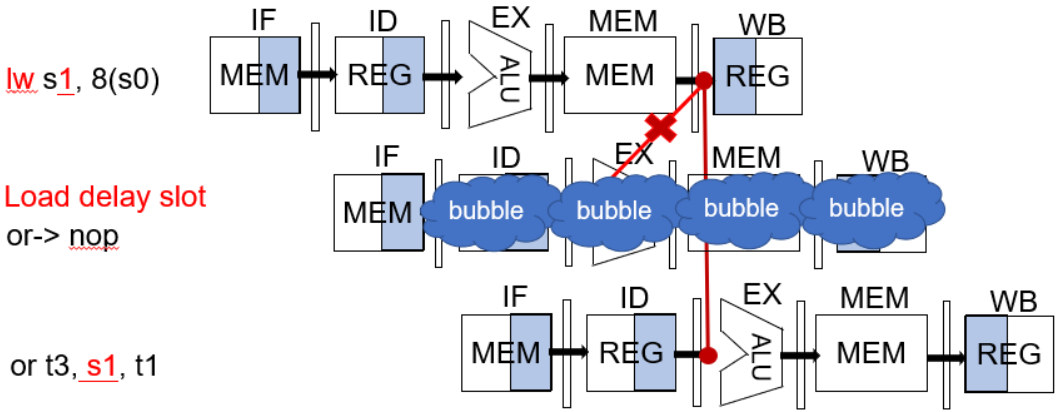
Load Delay Slot

- 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!



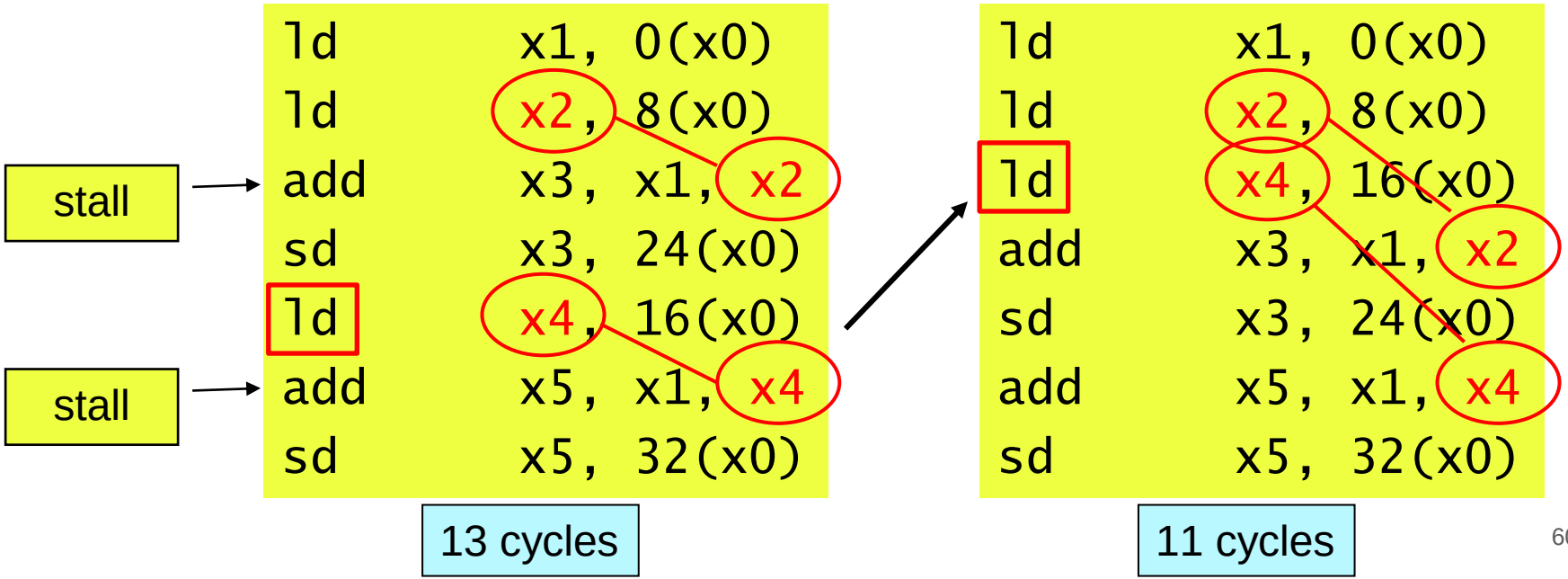
Load Delay Slot

- Code scheduling: Fix data hazard using the compiler
  - In the delay slot, put an instruction unrelated to the load result
    - -> No performance loss!
  - With the knowledge of the underlying CPU pipeline, the compiler reorders codes to improve performance of the CPU



Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next inst.
- C code for a = b + e; c = b + f;

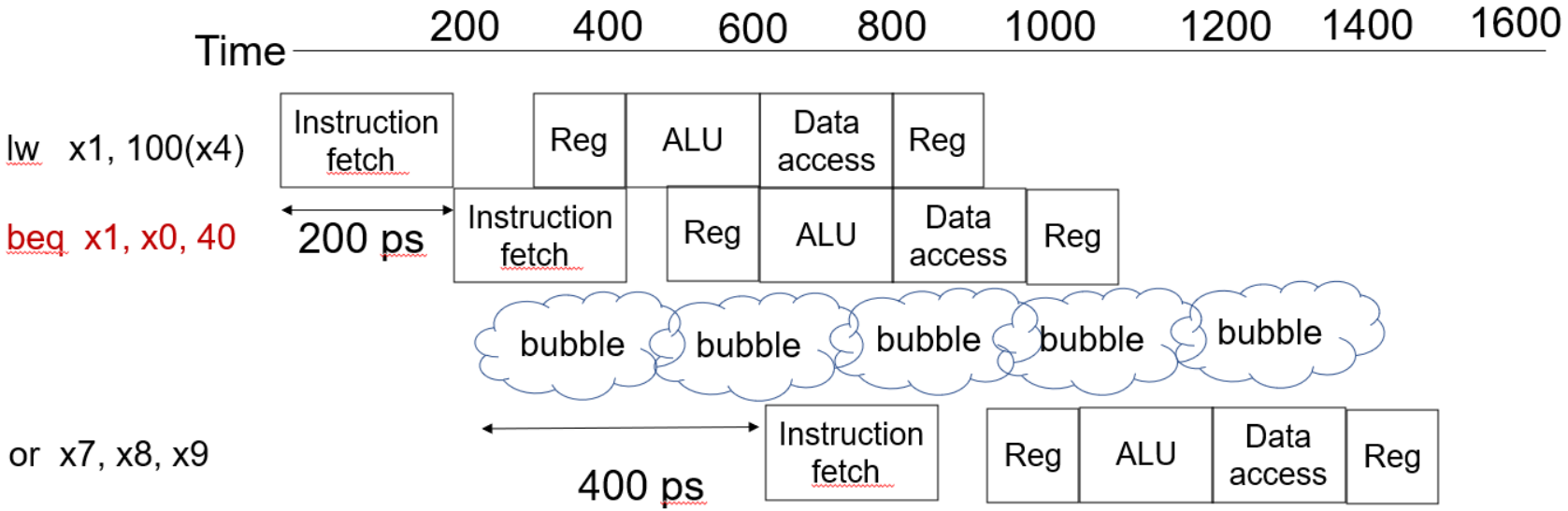


Control Hazards

- Branch determines flow of control
  - The processor does not know what instruction to fetch next because the branch decision has not been made
  - The decision is made in the EX stage
  - The pipeline would have to be stalled for two cycles at every banch
  - Severely degrade the system performance if branches occurs often

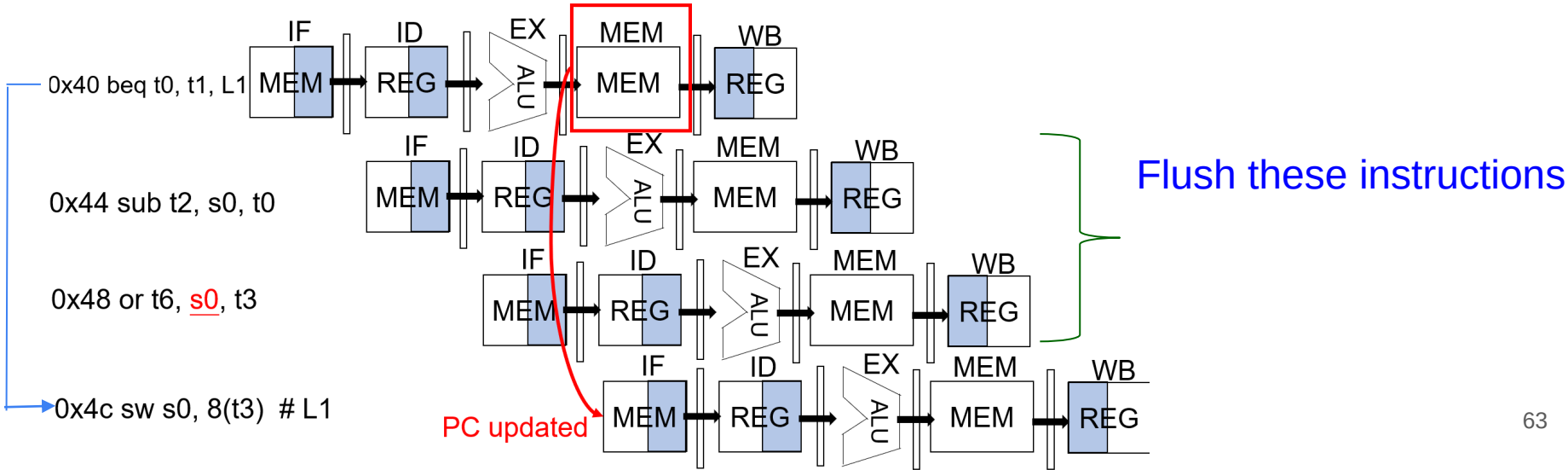
Stall on Branch

- Wait until branch outcome determined before fetching next instruction



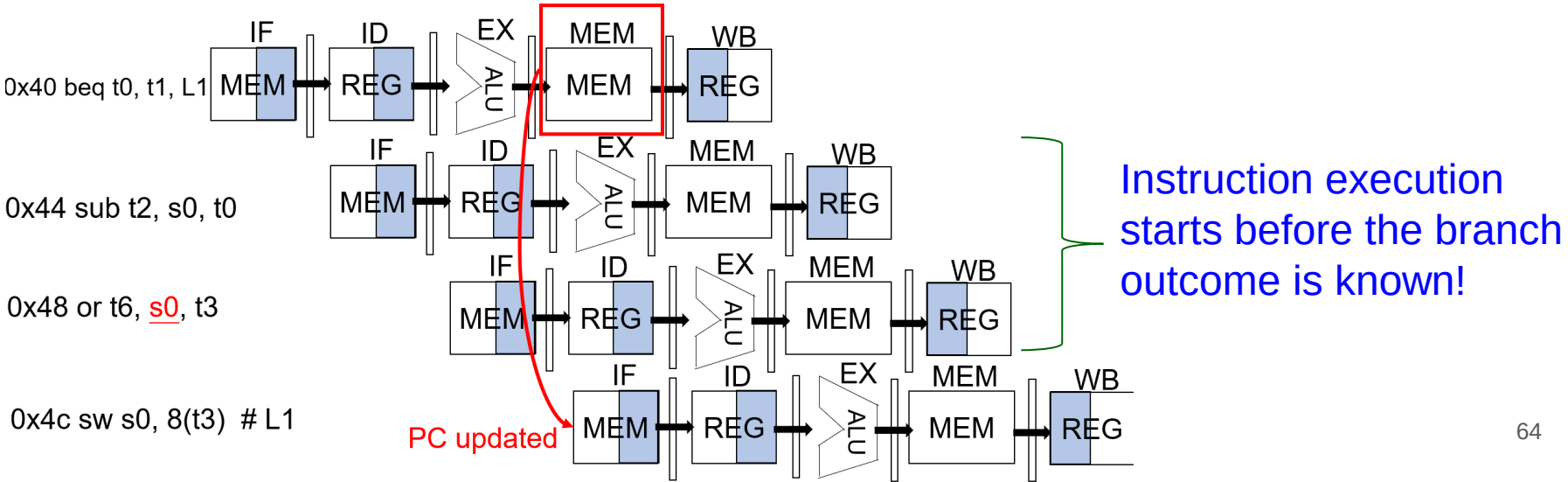
Control Hazard

- An alternative to stalling the pipeline
  - Predict whether the branch will be taken
  - Assume not taken and flush instructions when branch is taken



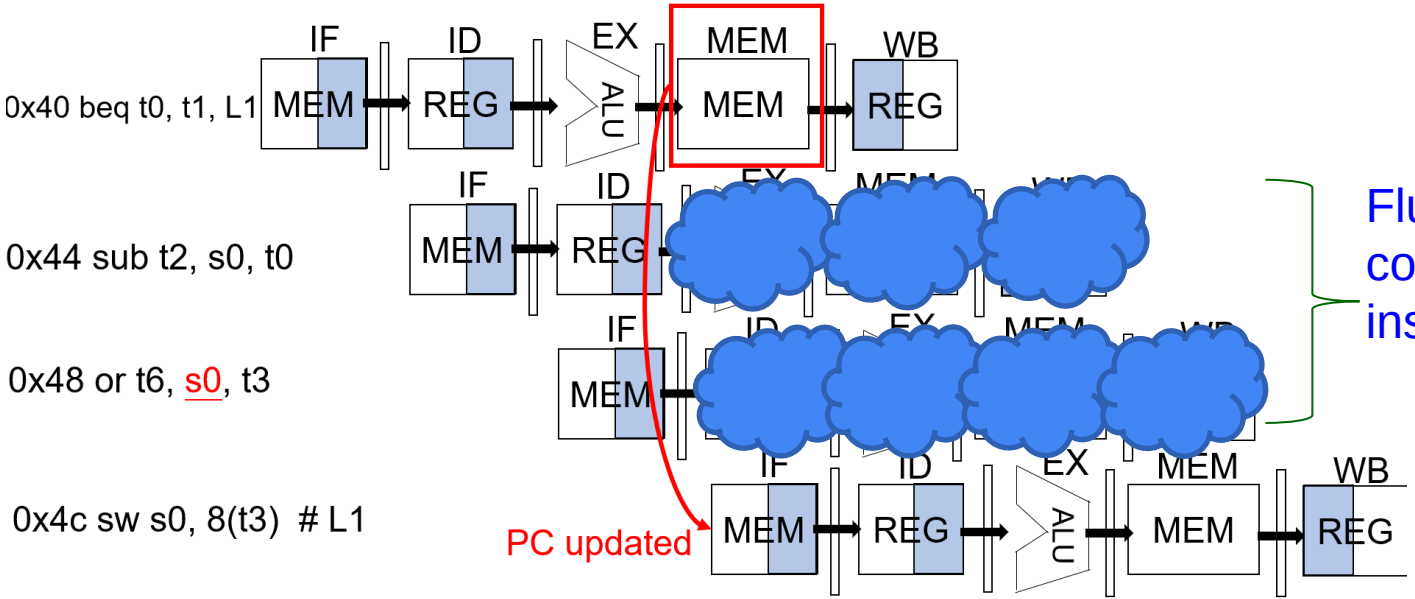
Control Hazard

- Control hazard (conditional branch) occurs when the instruction fetched may not be the one needed
  - If the “beq” branch is **taken** (jump to target label address)



Control Hazard

- Kill instructions after branch (if taken)

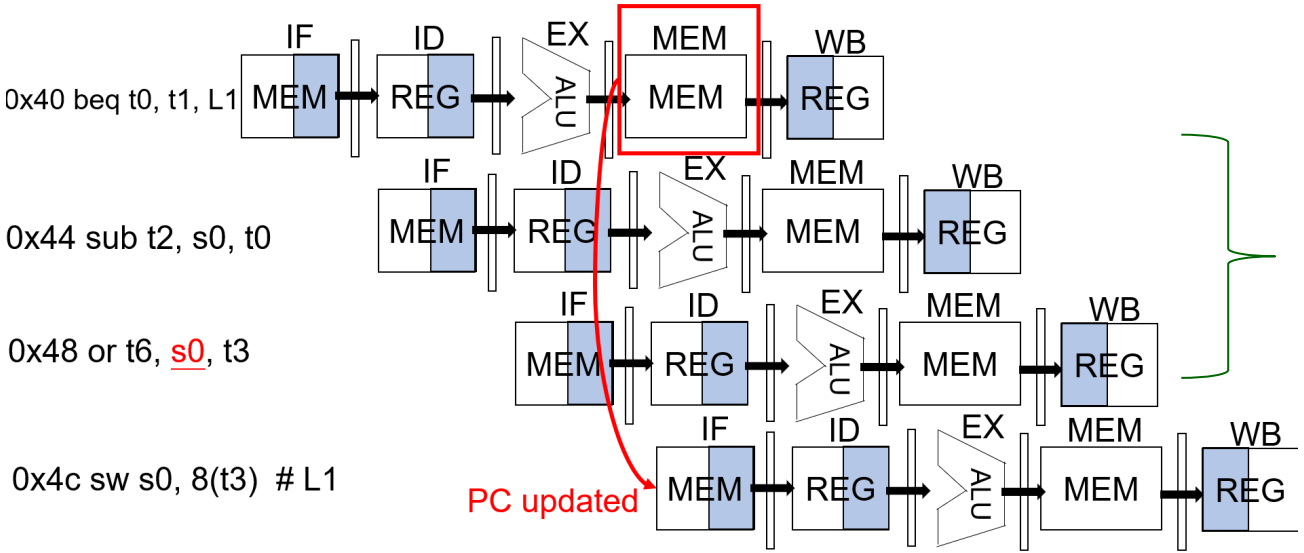


Control Hazard

- **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
  - Guess next PC to be PC + 4
  - Penalty for incorrect prediction is still 3 clock cycles



Control Hazards: Flushing Logic

- If branch is taken in EX stage, need to flush instructions in ID and IF stages
  - Do this by clearing ID and IF pipeline registers using “FlushD” and “FlushE” wire signals
- Equations
  - FlushD = PCSrcE
  - FlushE = lwStall OR PCSrcE

Control Hazard

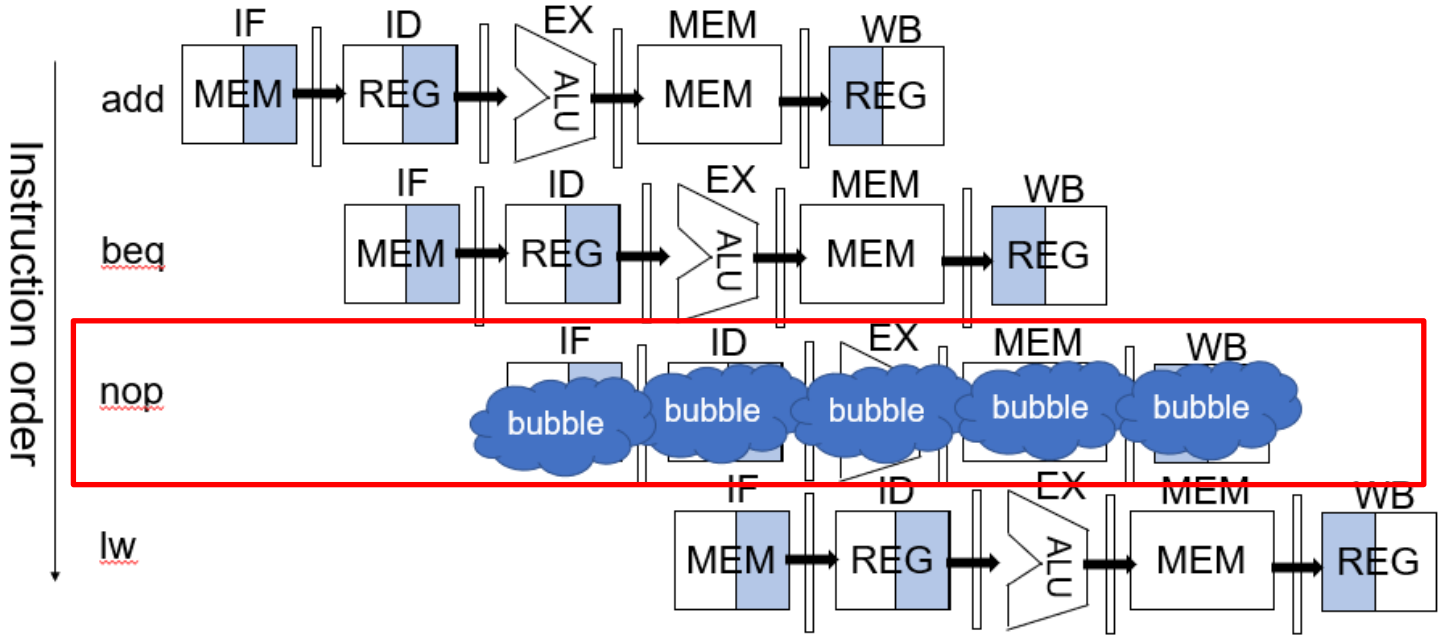
- We put branch decision-making hardware **in ALU stage**
  - Therefore, two more instructions after the branch will always be fetched, whether or not the branch is taken
- **Desired functionality of a branch**
  - If we do not take the branch, don't waste any time and continue executing normally
  - 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
  - Impact: 2 clock cycles per branch instruction

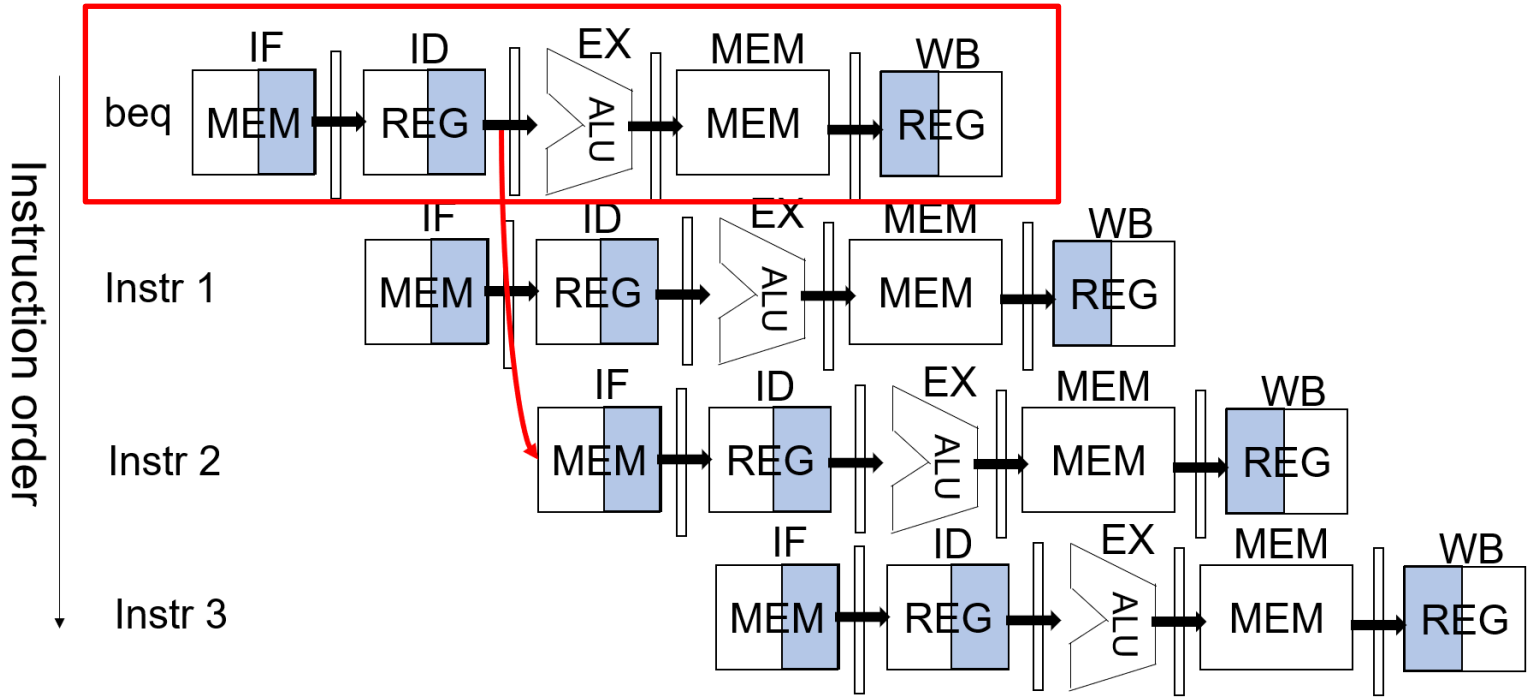


Control Hazard

- **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

- Branch comparator moved to Decode stage



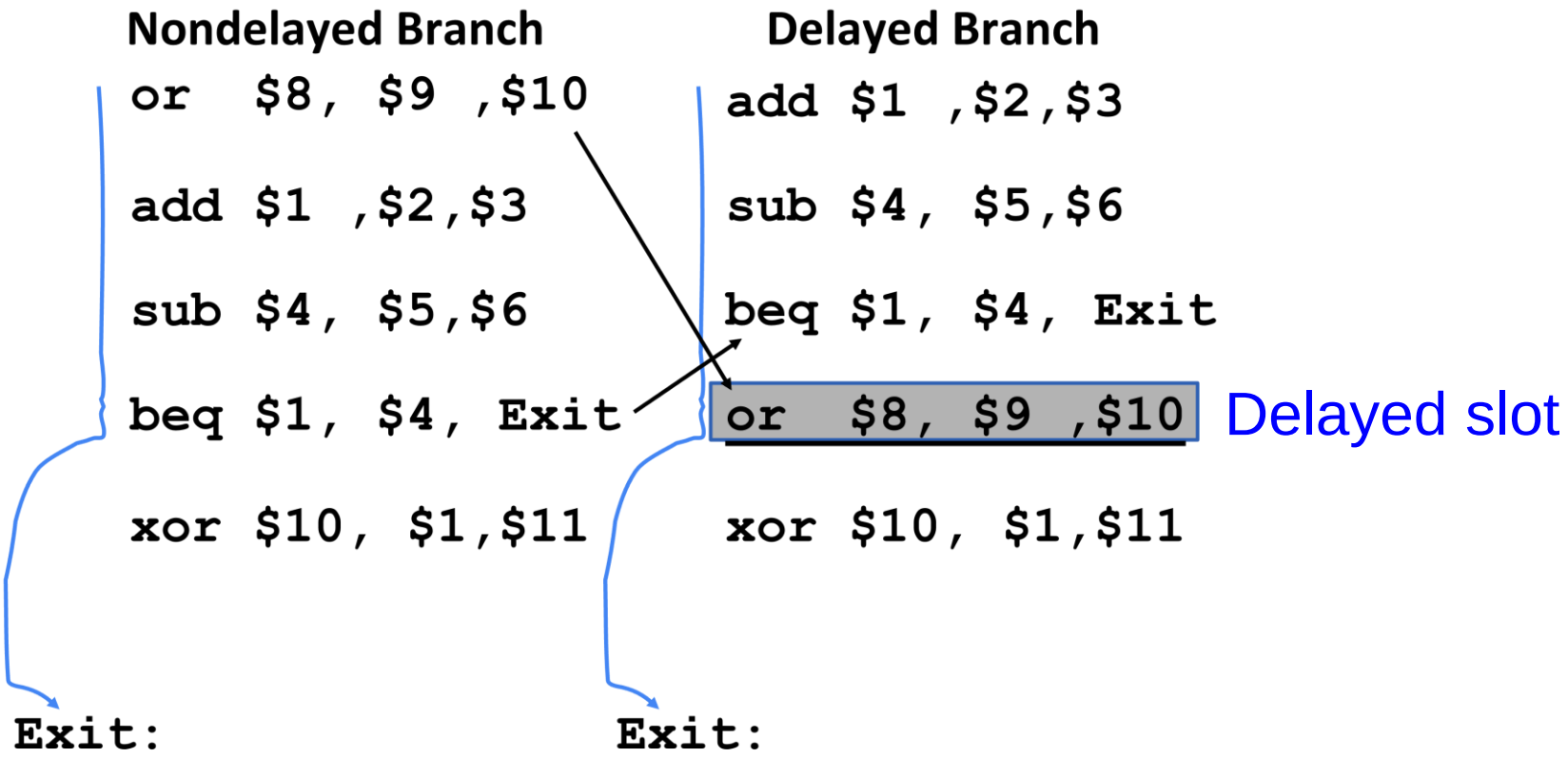
Delayed Branch Slot

- **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



Pipelined Processor Performance

- In SPECINT2000 Benchmark
  - 25% loads
  - 10% stores
  - 13% branches
  - 52% R-Type
- Suppose
  - 40% of branch used by next instruction
  - 50% of branches mis-predicted
- What is the average CPI?

Pipelined Processor Performance

40% of branch used by next instruction
50% of branches mis-predicted

- What is the average CPI?
  - The CPI of load is 1 when not stalling, 2 when stalling
    - $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$
  - The CPI of branch is 1 when not stalling, 3 when stalling
    - CPIbeq = 1(0,5) + 3(0.5) = 2

25% loads
10% stores
13% branches
52% R-Type

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

Pipelined Processor Performance

- Pipelined processor critical path

$$T_{c\_pipelined} = \max \begin{bmatrix} t_{pcq} + t_{mem} + t_{setup} & \text{Fetch} \\ 2(t_{RFread} + t_{setup}) & \text{Decode} \\ t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND\text{-}OR} + t_{setup} & \text{Execute} \\ t_{pcq} + t_{mem} + t_{setup} & \text{Memory} \\ 2(t_{pcq} + t_{mux} + t_{RFsetup}) & \text{Writeback} \end{bmatrix}$$

  - The register is written in the first half of the Writeback cycle and read in the second half of the Decode cycle
  - The critical path occurs in EX stage when
    - A branch is in the EX stage that requires forwarding (from the Writeback pipeline register to the PC register)

Pipelined Processor Performance

- Pipelined processor critical path

$$T_{c\_pipelined} = \max [\, 40 + 200 + 50,\ 2(100 + 50),\ 40 + 4(30) + 120 + 20 + 50,\ 40 + 200 + 50,\ 2(40 + 30 + 60)\,]$$
$$= 350 \text{ ps}$$

Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	t_{AND-OR}	20
ALU	t_{ALU}	120
Decoder (control unit)	t_{dec}	25
Extend unit	t_{ext}	35
Memory read	t_{mem}	200
Register File read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Pipelined Processor Performance

- In the Pipelined processor
  - What is the execution time for a program with 100 billion instructions?

$T_{pipelined}$ = (100 x 10^9 instructions) x (1.23 cycle/instruction) x (350 x 10^{-12} s/cycle)
= 43 seconds

75 seconds for the single-cycle processor

Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	t_{AND-OR}	20
ALU	t_{ALU}	120
Decoder (control unit)	t_{dec}	25
Extend unit	t_{ext}	35
Memory read	t_{mem}	200
Register File read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

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%

Pipeline Summary

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control

Conclusion

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