National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 8: Pipelining I

CS10014 Computer Organization

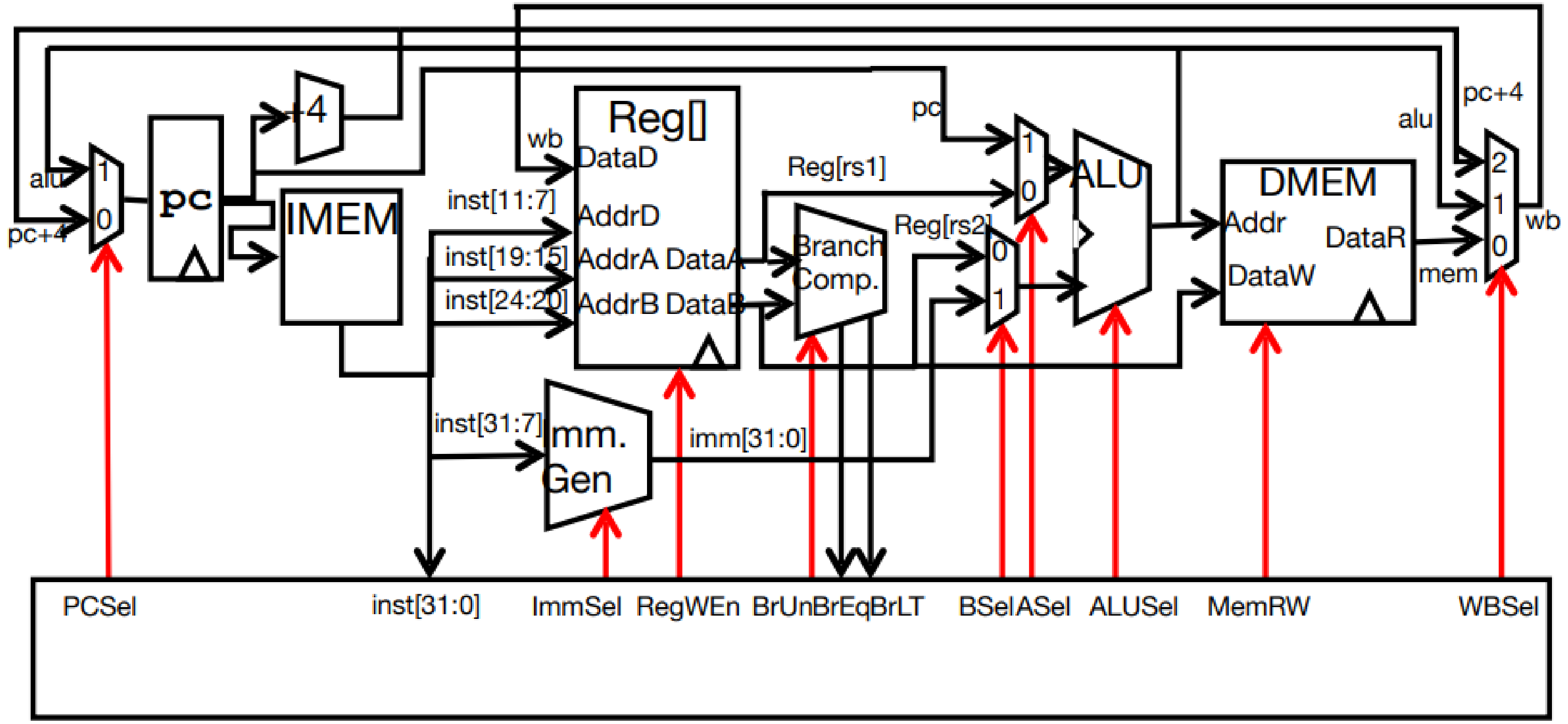
Department of Computer Science
Tsung Tai Yeh
Thursday: 1:20 pm– 3:10 pm
Classroom: EC-022

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

Single-Cycle RISC-V RV32I Datapath (1/3)



Single-Cycle RISC-V RV32I Datapath (2/3)

- 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

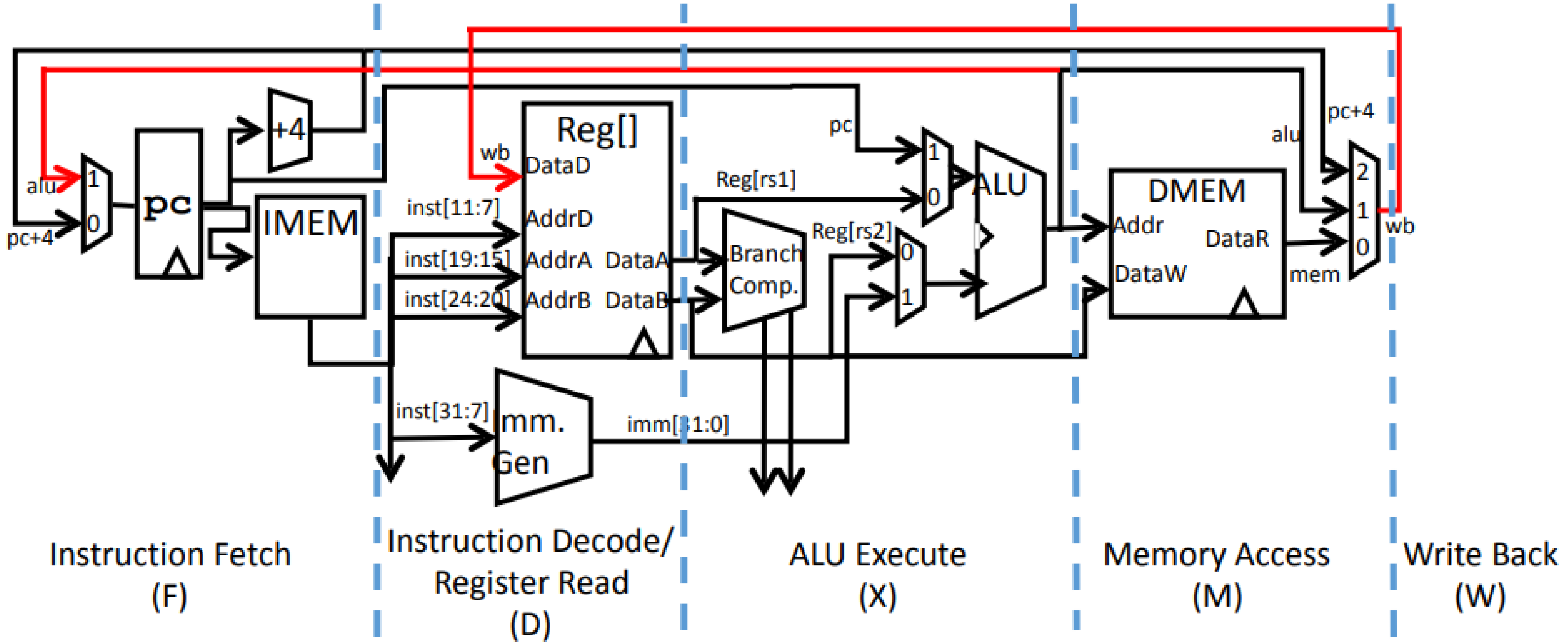
Instr	IF = 200ps	ID = 100ps	ALU = 200ps	MEM=200ps	WB = 100ps	Total
add	X	X	X		X	600ps
beq	X	X	X			500ps
jal	X	X	X		X	600ps
lw	X	X	X	X	X	800ps
sw	X	X	X	X		700ps

Single-Cycle RISC-V RV32I Datapath (3/3)

- How to improve the clock rate?
- Will clock rate improvement help the performance as well?
  - We want to increase the clock rate to result in programs executing quicker

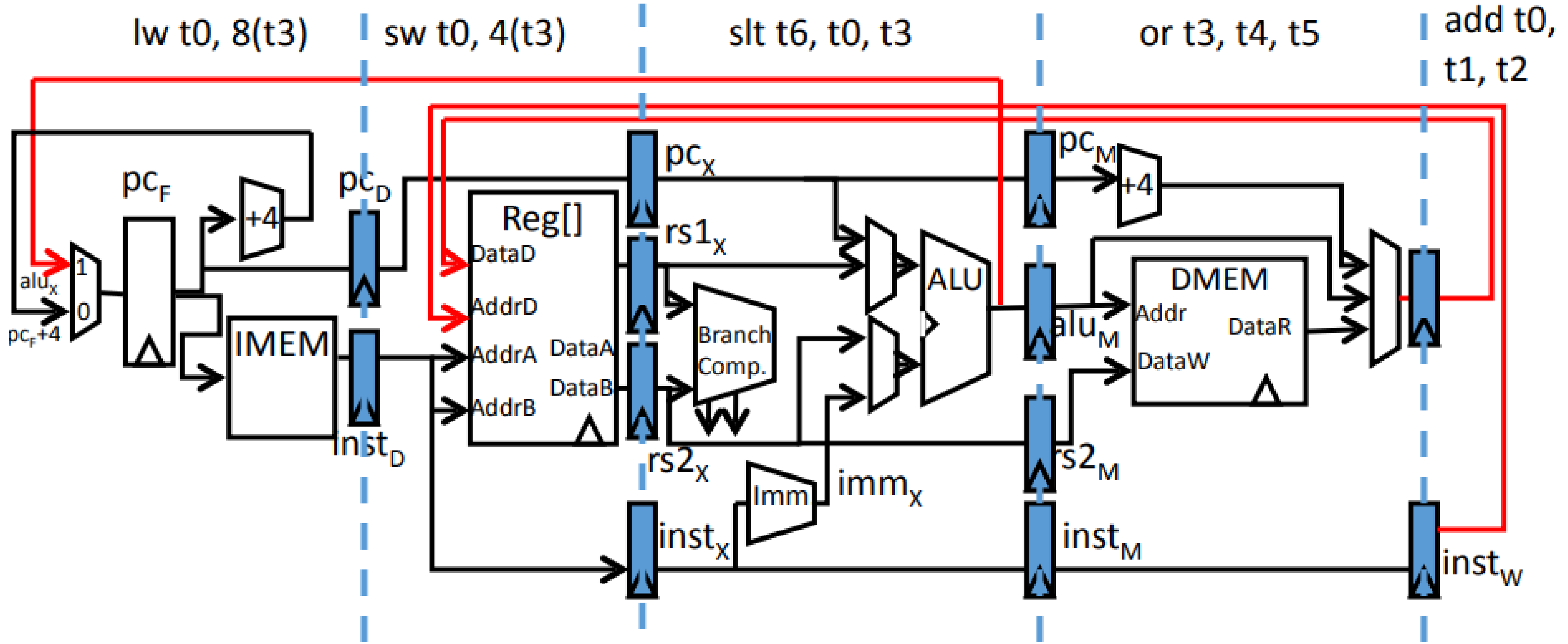
Instr	IF = 200ps	ID = 100ps	ALU = 200ps	MEM=200ps	WB = 100ps	Total
add	X	X	X		X	600ps
beq	X	X	X			500ps
jal	X	X	X		X	600ps
lw	X	X	X	X	X	800ps
sw	X	X	X	X		700ps

Pipelining RISC-V RV32I Datapath (1/2)



Pipelining RISC-V RV32I Datapath (2/2)

- Each stage operates on different 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

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

Iron Law of Processor Performance (1/4)

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

CPI - Cycles Per Instructions

Iron Law of Processor Performance (2/4)

- **Instructions per program determined by**
  - Algorithm, e.g. $O(N^2)$ vs $O(N)$
  - Programming language
  - Compiler
  - Instruction Set Architecture (ISAs)

Iron Law of Processor Performance (3/4)

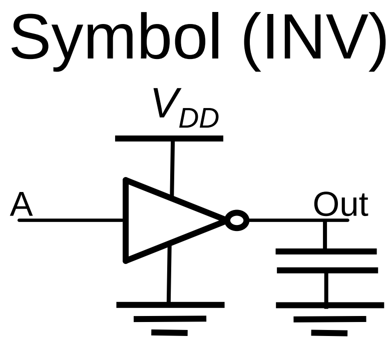
- **CPI determined by**
  - ISA
  - Processor implementation (or microarchitecture)
  - E.g. the single-cycle RISC-V design, CPI = 1
  - Complex instructions (e.g. strcpy), CPI >> 1
  - Superscalar processors, CPI < 1 (next lectures)

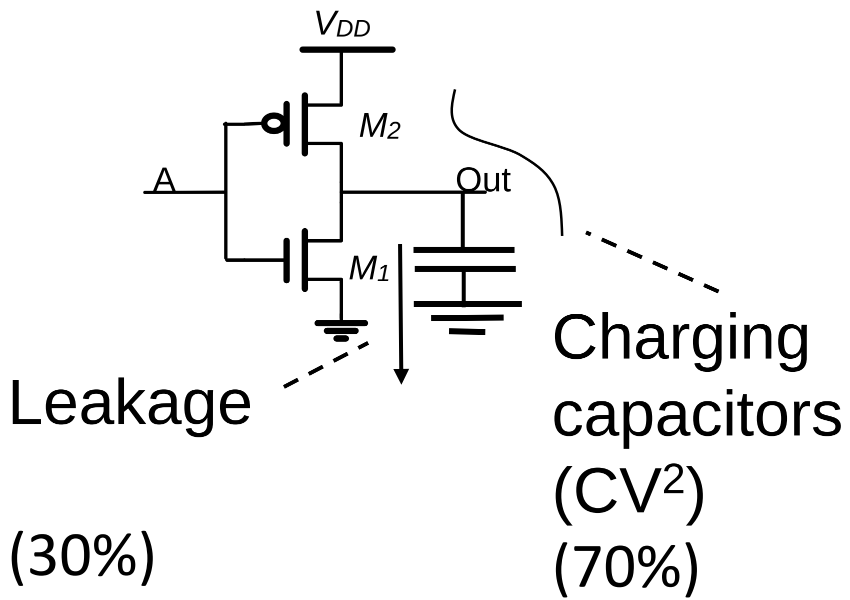
Iron Law of Processor Performance (4/4)

- **Time per cycle determined by**
  - ISA
  - Processor microarchitecture (determines the critical path through logic gates)
  - Technology (e.g. 5 nm vs. 28 nm)
  - Power budget (lower voltages reduce transistor speed)

Energy Efficiency (1/5)

- **Where does energy go in CMOS?**





Energy Efficiency (2/5)

- **Energy per task**
  - Want to reduce capacitance and voltage to reduce energy/task

$$\boxed{\frac{\text{Energy}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Energy}}{\text{Instruction}}}$$

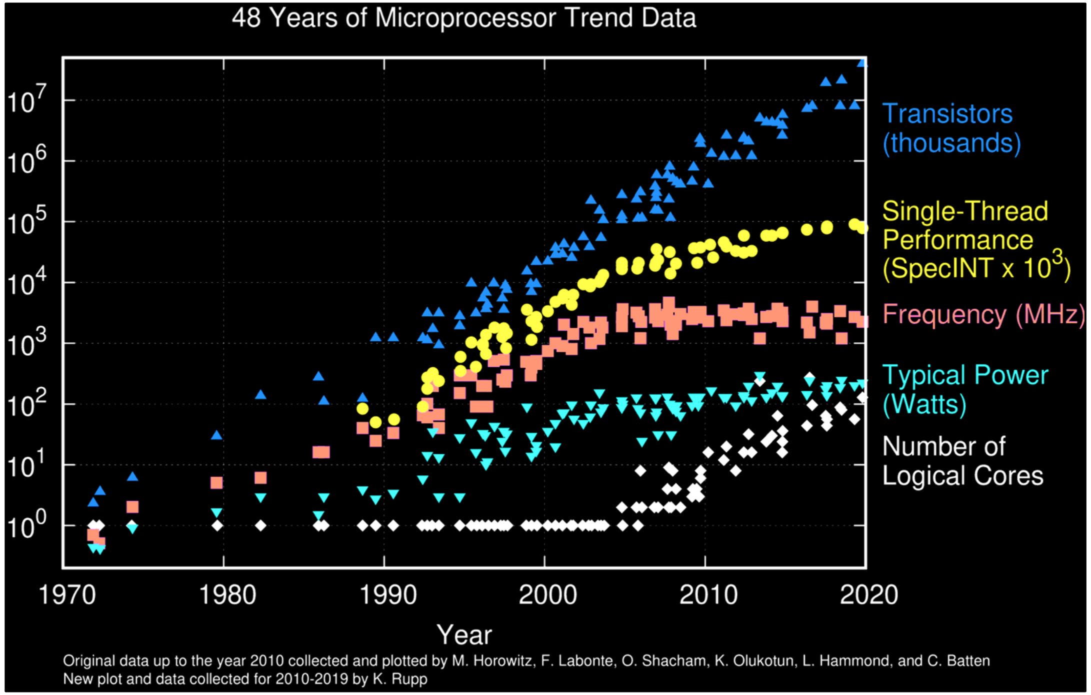
$$\boxed{\frac{\text{Energy}}{\text{Program}} \ \alpha \ \frac{\text{Instructions}}{\text{Program}} * C \ V^2}$$

“Capacitance” depends on technology,
processor features
e.g. # of cores

Supply voltage,
e.g. 1V

Energy Efficiency (3/5)

- **Performance/power trends**



Energy Efficiency (4/5)

- **End of Dennard Scaling**
  - Dennard Scaling: Power density remained constant for a given area of silicon while the dimension of the transistor shrank
  - In next-generation processors, significantly improved energy efficiency thanks to
    - Moore's Law
      - The size of transistors is not shrinking as much as before
      - Need to go to 3D
    - Reduce supply voltage
      - Increasing "leakage power" where transistor switches don't fully turn off
    - Power becomes a growing concern – the "power wall"

Energy Efficiency (5/5)

- **Energy “Iron Law”**

Performance =	**Power**	***** **Energy Efficiency**
(Tasks/Second)	*(Joules/Second)*	*(Tasks/Joule)*

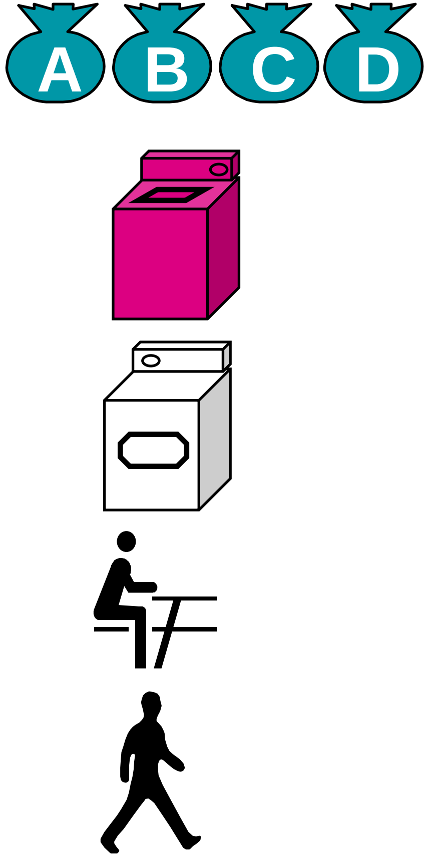
  - Energy efficiency is key metric in all computing devices
  - For power-constrained systems (e.g. 20 MW datacenter), need better energy efficiency to get more performance at the same power
  - For energy-constrained systems (e.g. 1W phone), need better energy efficiency to prolong battery life

Outline

- Pipelining
- Pipelining Execution
- Pipelining Datapath
- Pipelining Hazard
- Structural Hazard

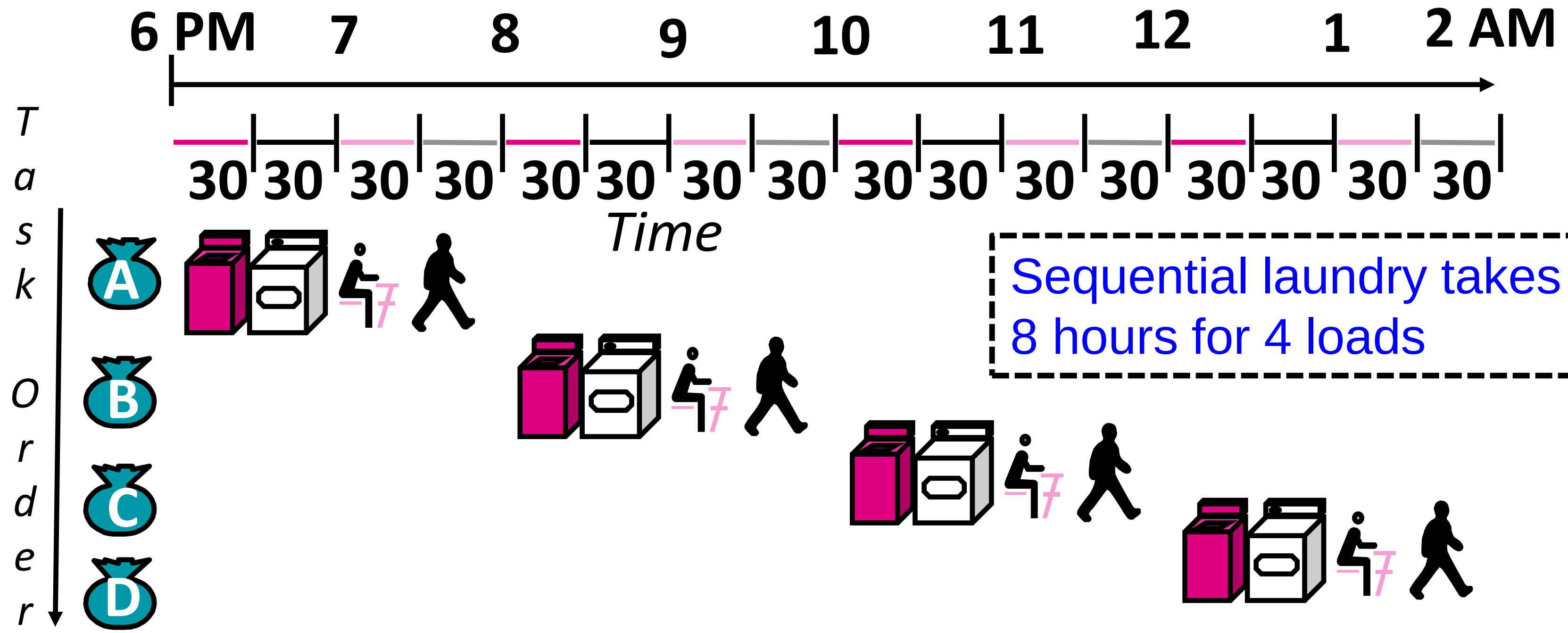
Pipelining (1/4)

- 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 (2/4)

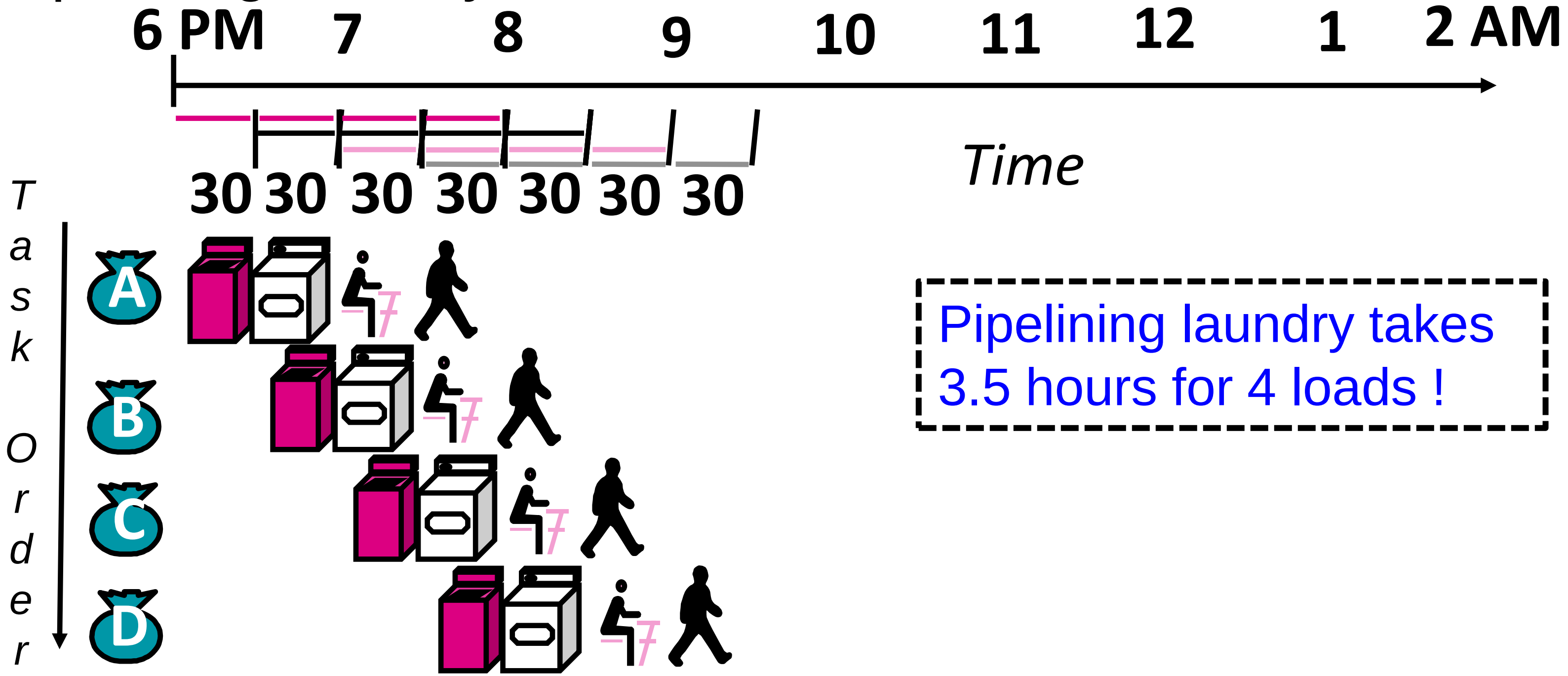
- Sequential Laundry



Sequential laundry takes 8 hours for 4 loads

Pipelining (3/4)

- Pipelining laundry



Pipelining laundry takes 3.5 hours for 4 loads !

Pipelining (4/4)

- 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

Outline

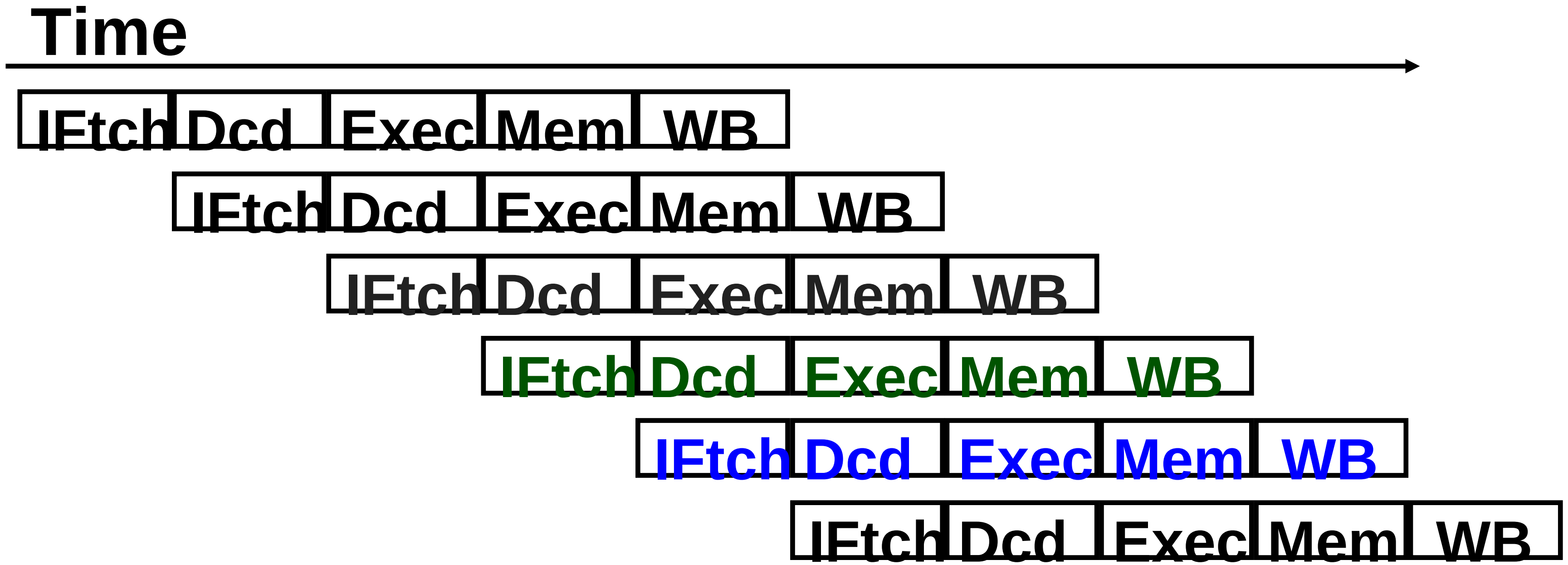
- Pipelining
- Pipelining Execution
- Pipelining Datapath
- Pipelining Hazard
- Structural Hazard

Pipelining Execution (1/10)

- 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

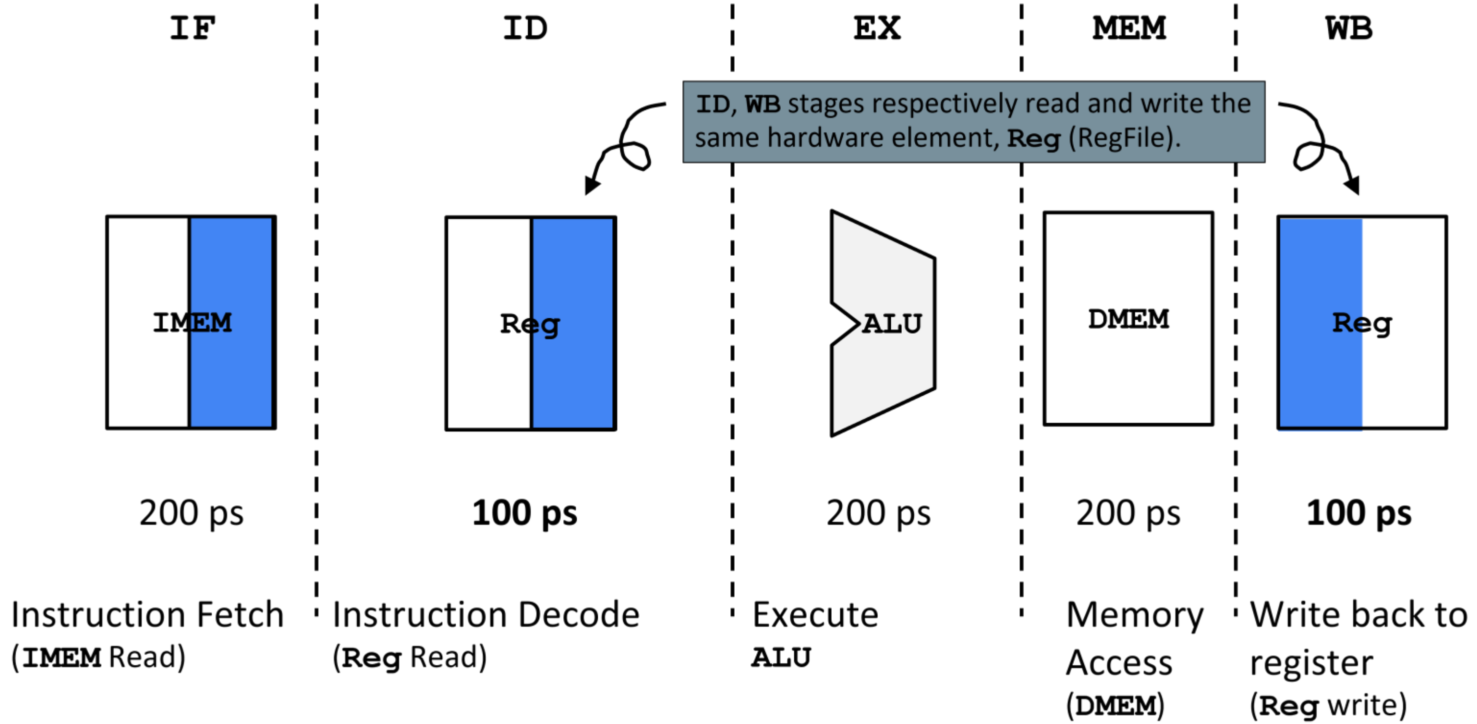
Pipelining Execution (2/10)

- Every instruction must take the same number of steps, also called the pipeline “stage”, so some will go idle sometimes



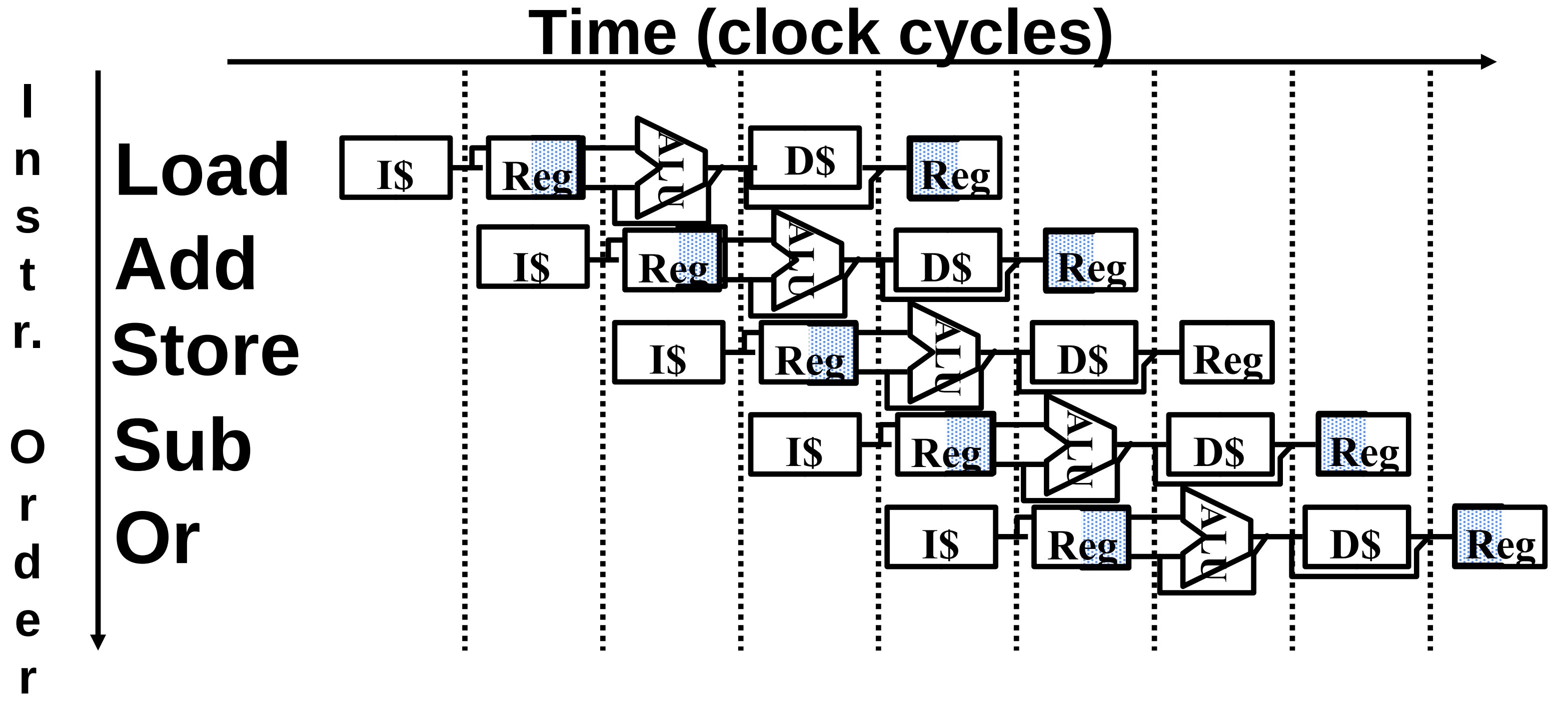
Pipelining Execution (3/10)

- Symbolic Representation of 5 Stages


IF
ID
EX
MEM
WB
ID, **WB** stages respectively read and write the same hardware element, **Reg** (RegFile).
IMEM
Reg
ALU
DMEM
Reg
200 ps
100 ps
200 ps
200 ps
100 ps
Instruction Fetch (**IMEM** Read)
Instruction Decode (**Reg** Read)
Execute **ALU**
Memory Access (**DMEM**)
Write back to register (**Reg** write)

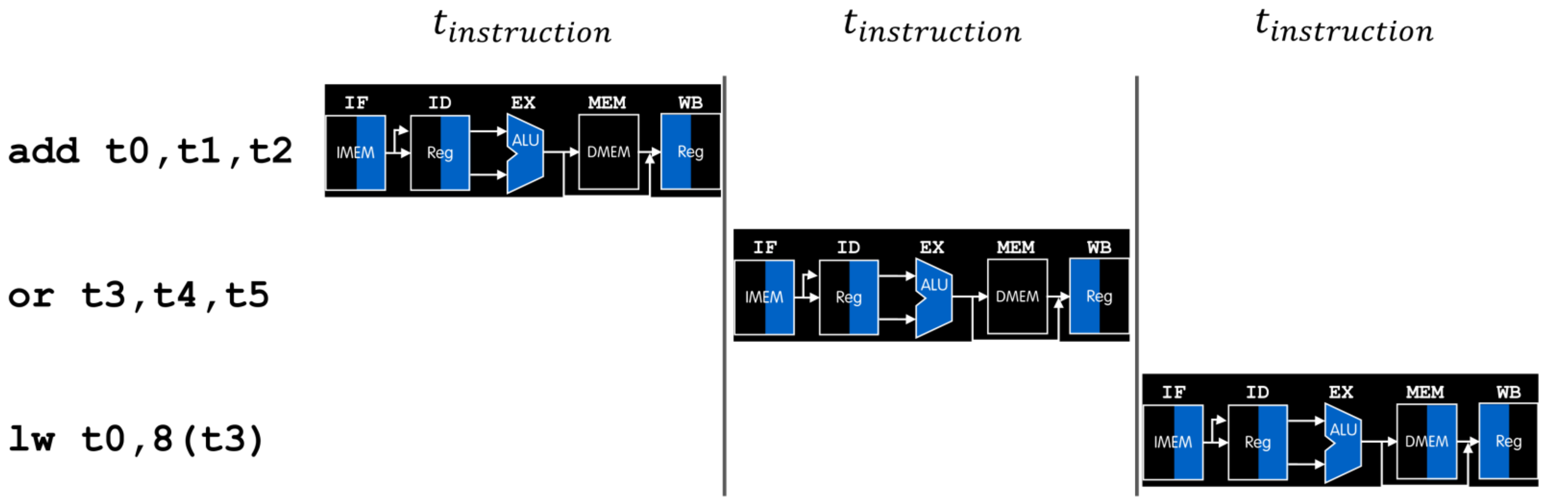
Pipelining Execution (4/10)

- In register, right half highlight read, left half write



Pipelining Execution (5/10)

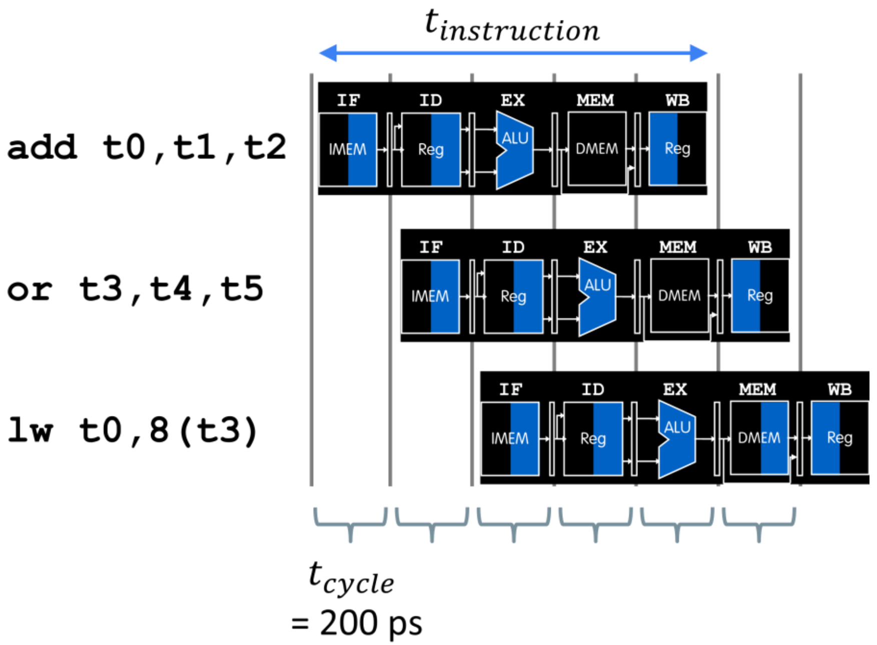
- In a single-cycle CPU, only one instruction can access any resources in one clock cycle



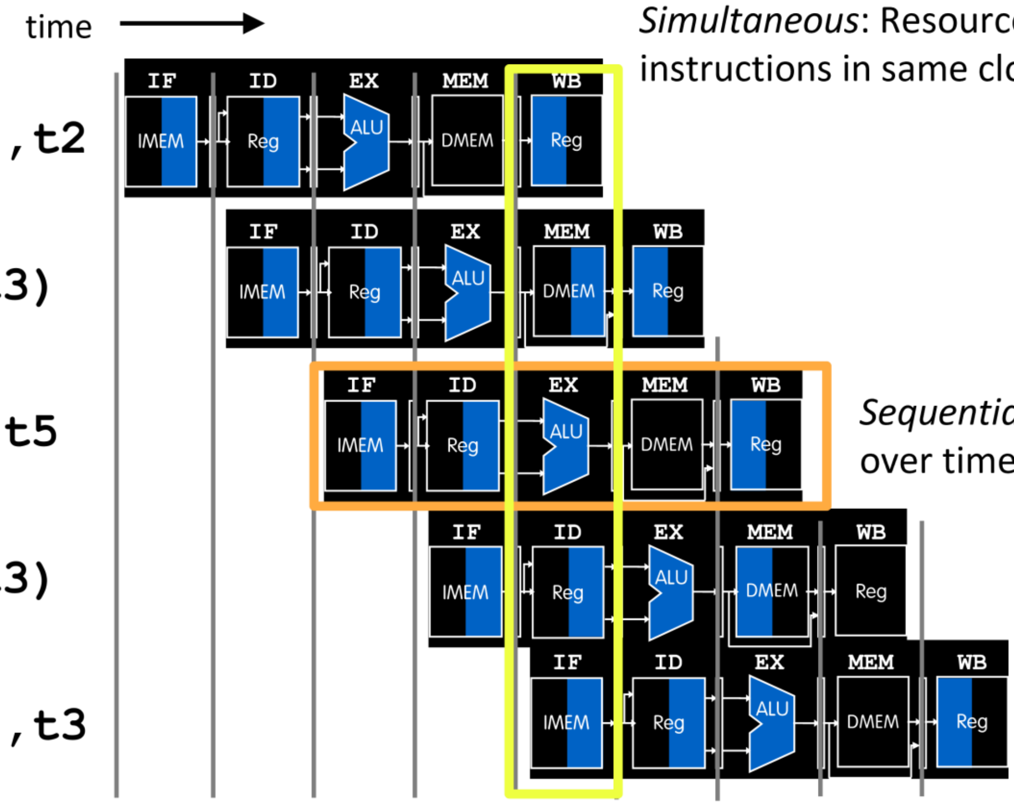
t_{cycle} = 800 ps

Pipelining Execution (6/10)

- In a pipelined CPU, multiple instructions access resources in one clock cycle



Pipelining Execution (7/10)



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

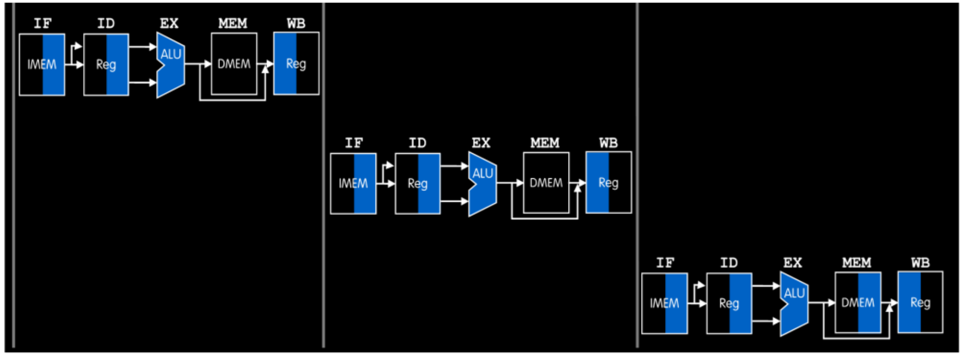
Simultaneous: Resource use by multiple instructions in same clock cycle.

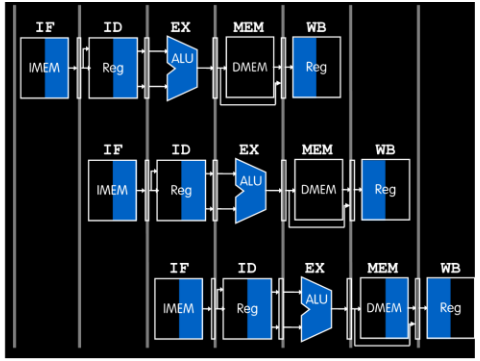
Sequential: Resource use by same instruction over time (multiple clock cycles)

Pipelining Execution (8/10)

- The pipelined CPU uses one clock for all stages; clock cycle time is limited by the slower stages

```
add t0,t1,t2

or t3,t4,t5

lw t0,8(t3)
```





	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*

Pipelining Execution (9/10)

- 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 (10/10)

- 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

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

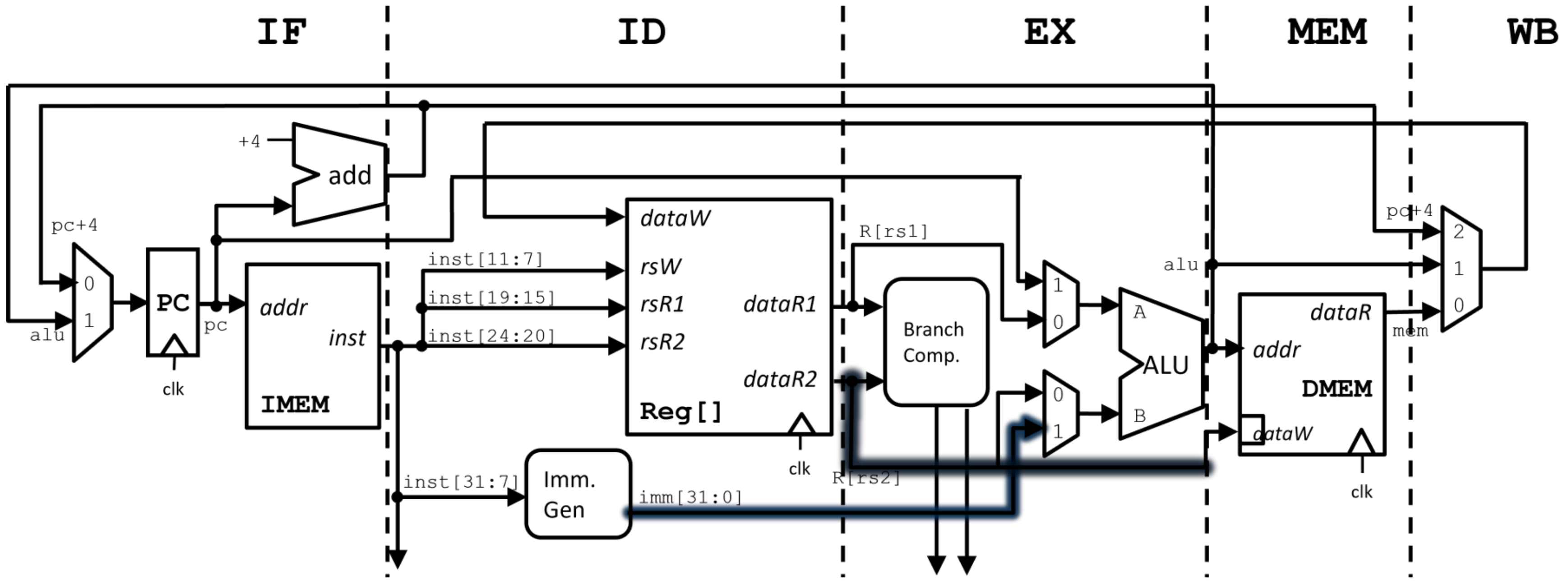
- Which of the following statement(s) is/are True or False?
  - (a) Thanks to pipelining, I have reduced the time it took me to wash my shirt. (False)
    - **Throughput better, not execution time**
  - (b) Longer pipelines are always a win (since less work per stage & a faster clock) (False)
    - **longer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

Outline

- Pipelining
- Pipelining Execution
- Pipelining Datapath
- Pipelining Hazard
- Structural Hazard

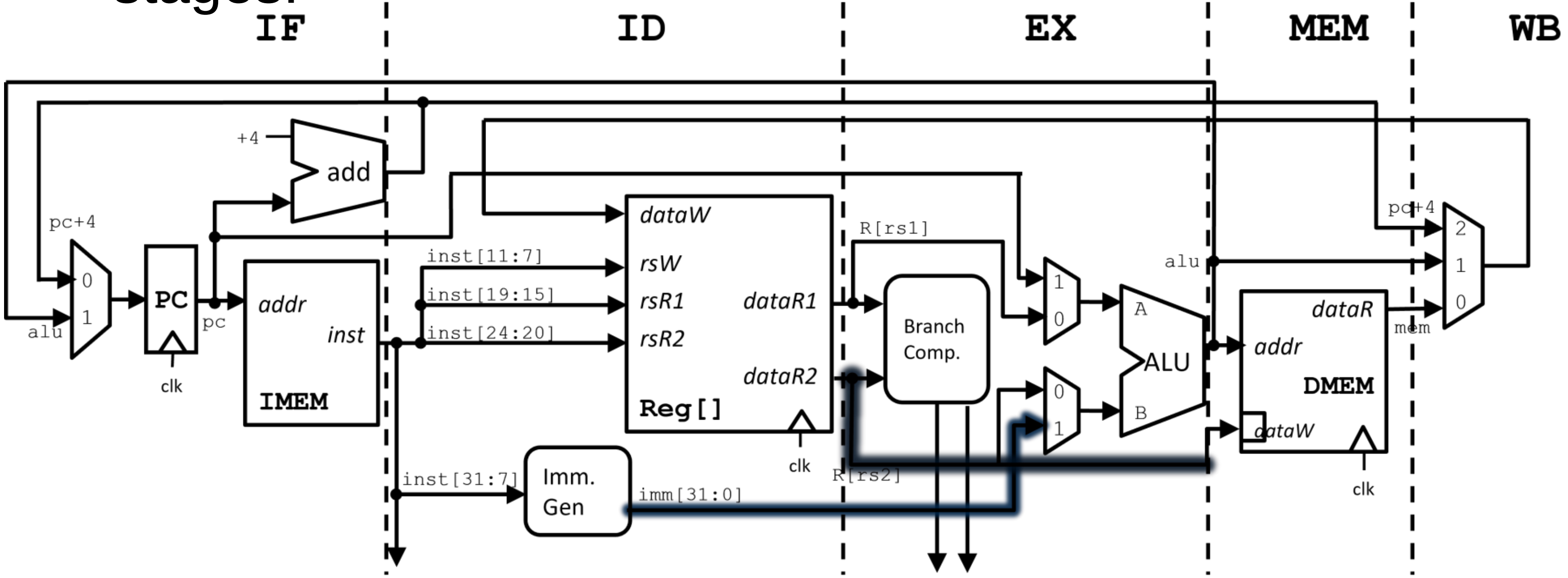
Pipelining Datapath (1/10)

- Each stage needs to process data from a different inst.



Pipelining Datapath (2/10)

- Use pipeline registers to carry instruction data between stages!



Pipelining Datapath (3/10)

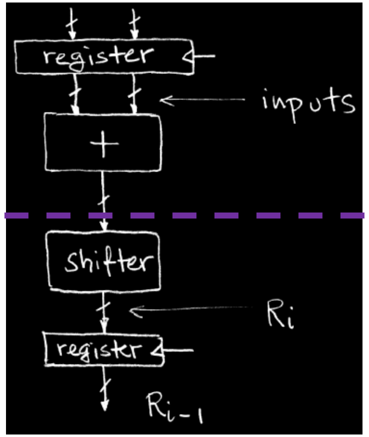
Single-cycle datapath means 1 clock cycle, from input to output.

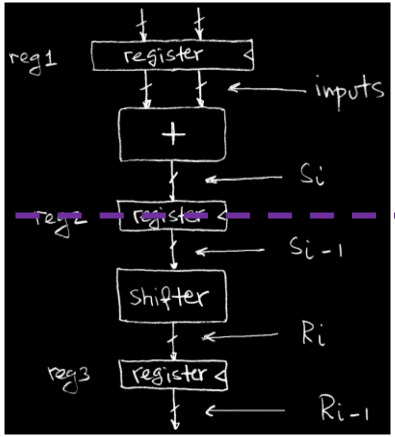
- Clock period limited by propagation delays of adder and shifter.

Insertion of pipeline register allows higher clock frequency.

Clock period now limited by *max* {adder/shifter prop. delays}.

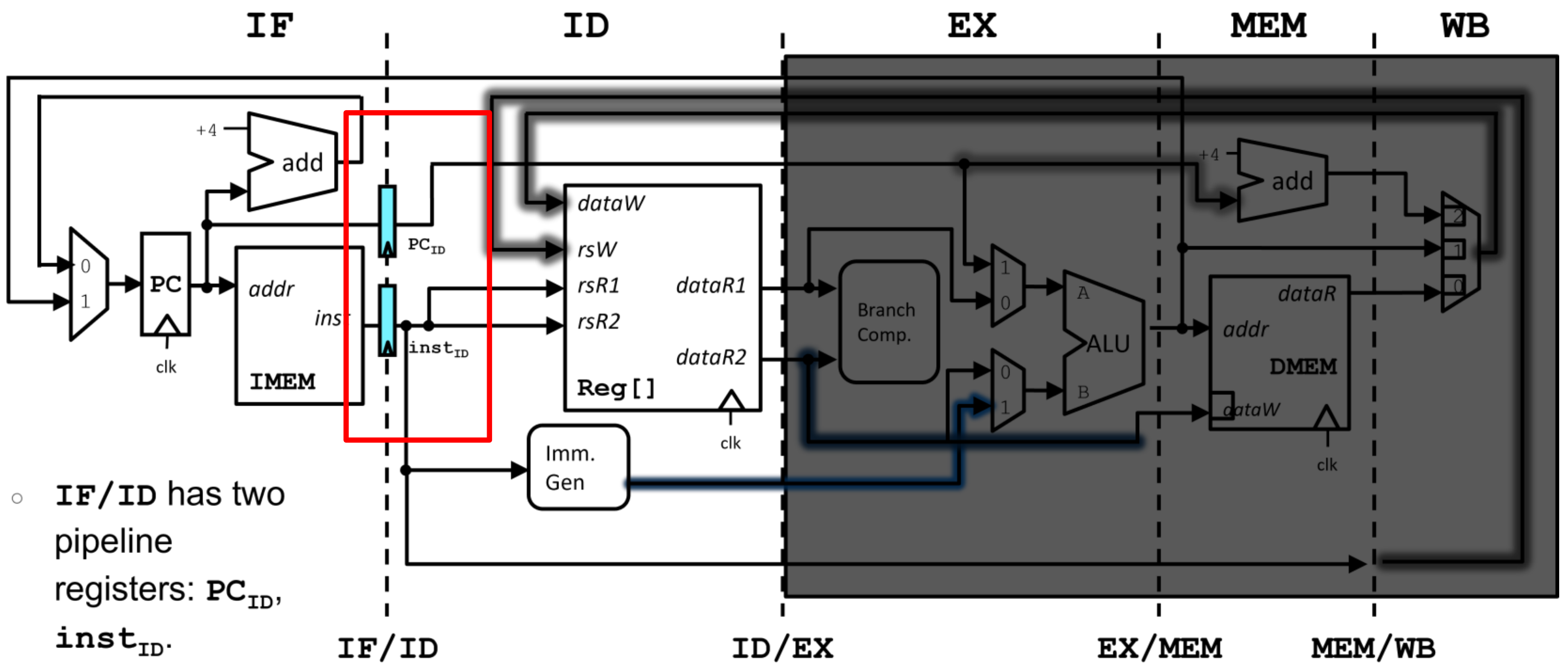
Higher throughput (outputs/s).





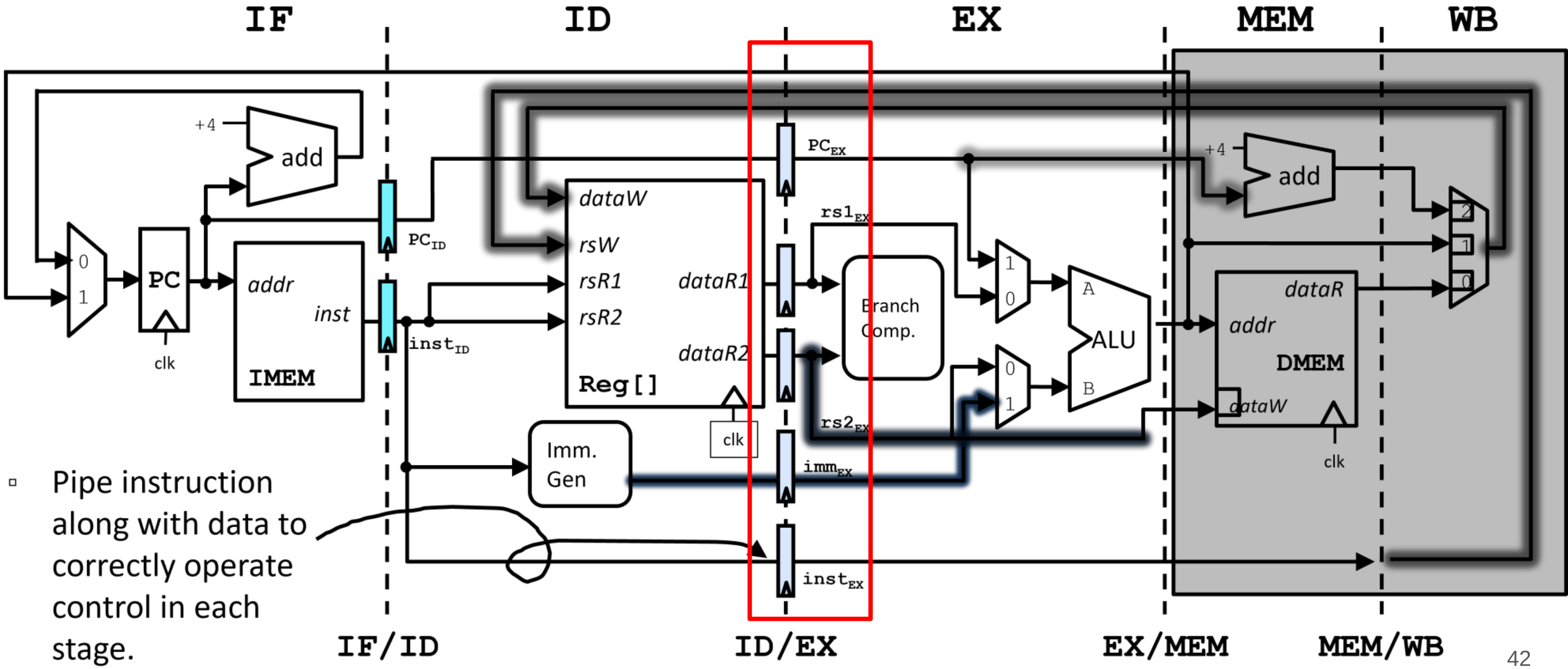
Pipeline Register

Pipelining Datapath (4/10)



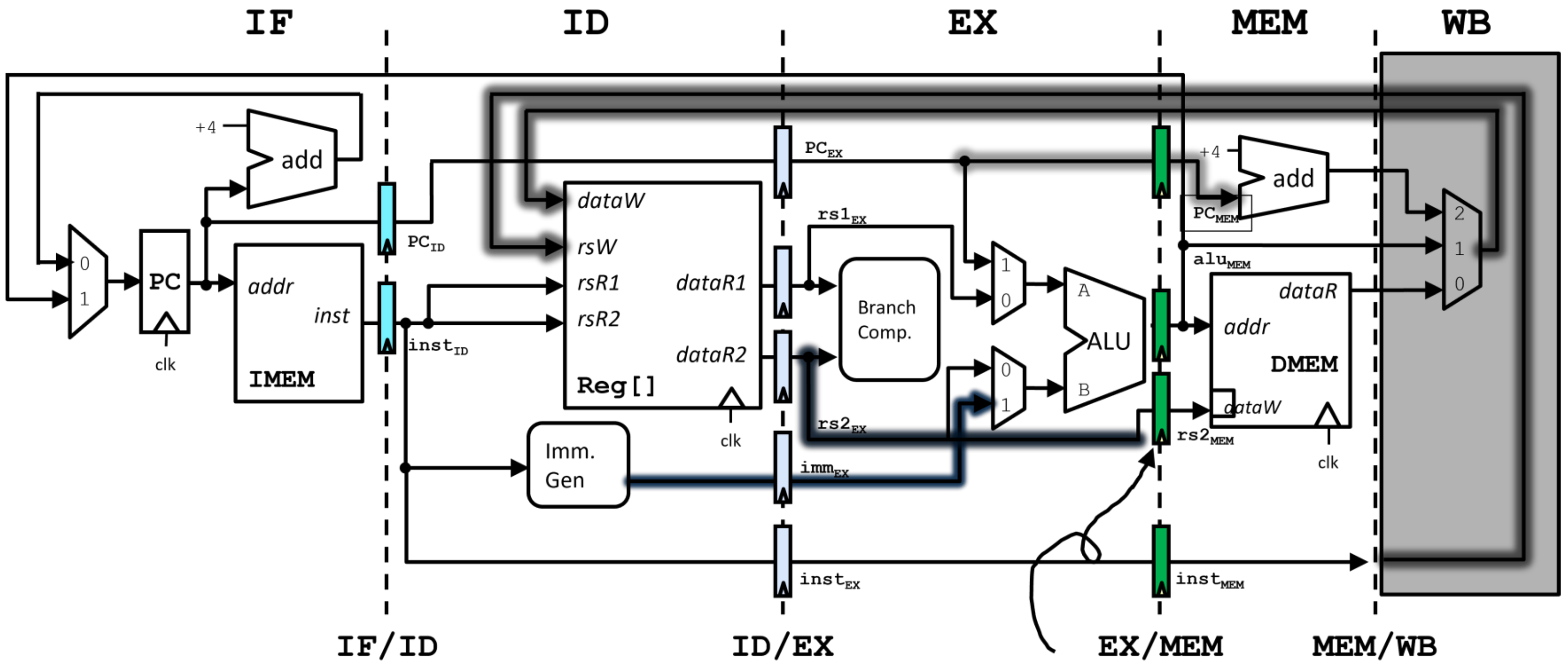
- **IF/ID** has two pipeline registers: $\mathbf{PC_{ID}}$, $\mathbf{inst_{ID}}$.
- Increment PC to PC + 4 for next cycle's **IF** stage.

Pipelining Datapath (5/10)



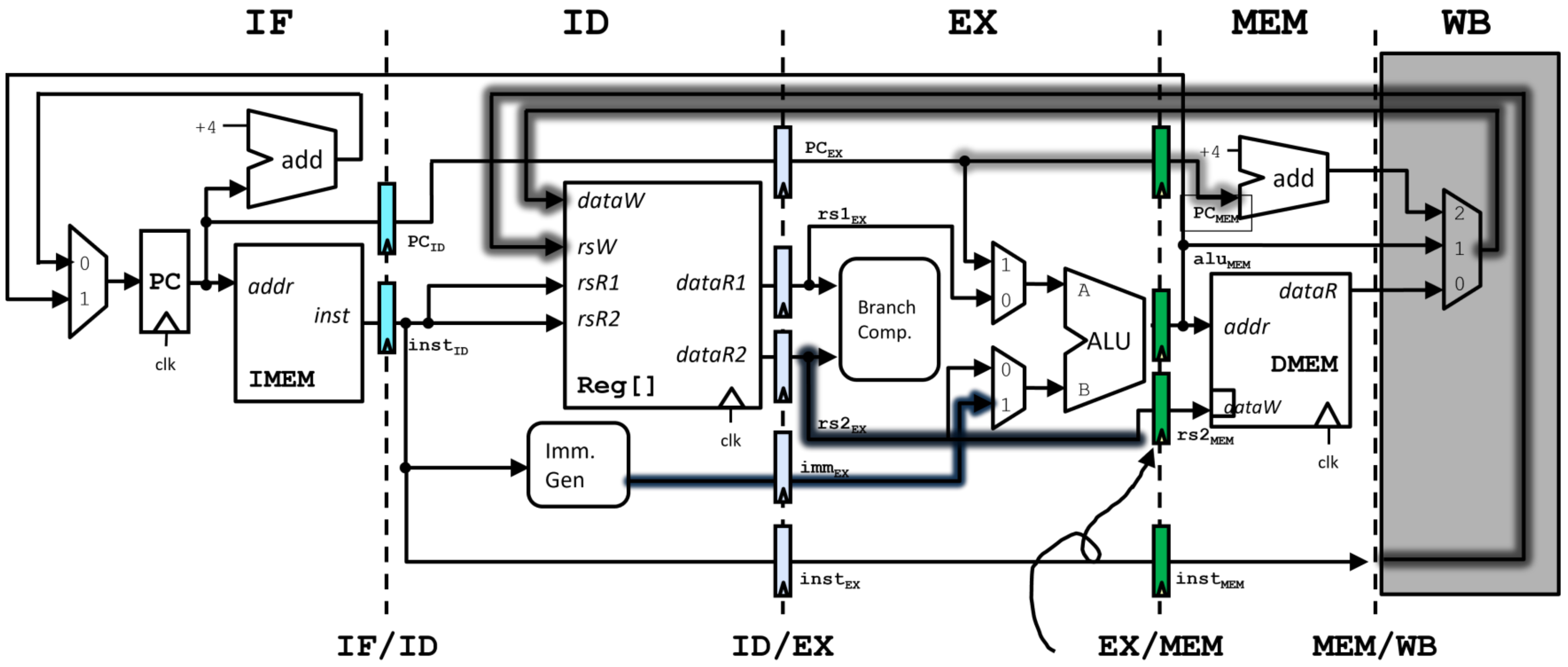
- Pipe instruction along with data to correctly operate control in each stage.

Pipelining Datapath (6/10)



- **rs2** (data to store) needs to be piped through to **MEM**.

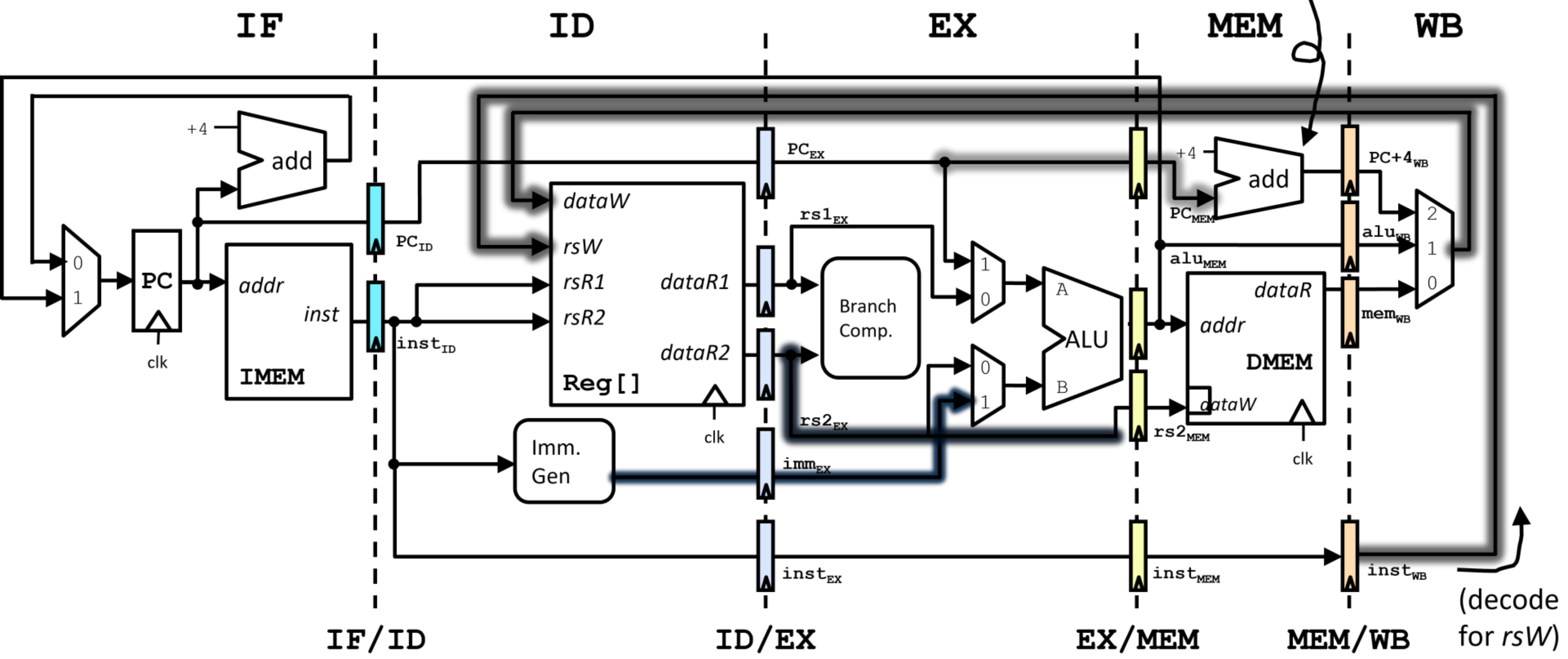
Pipelining Datapath (7/10)



- **rs2** (data to store) needs to be piped through to **MEM**.

Pipelining Datapath (8/10)

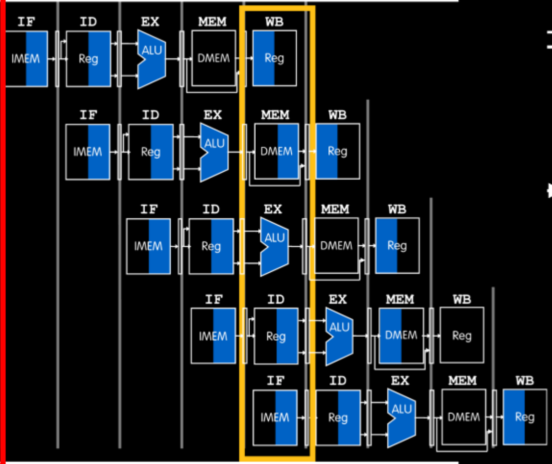
Recalculate **PC+4** to avoid sending both **PC**, **PC+4** down pipeline.

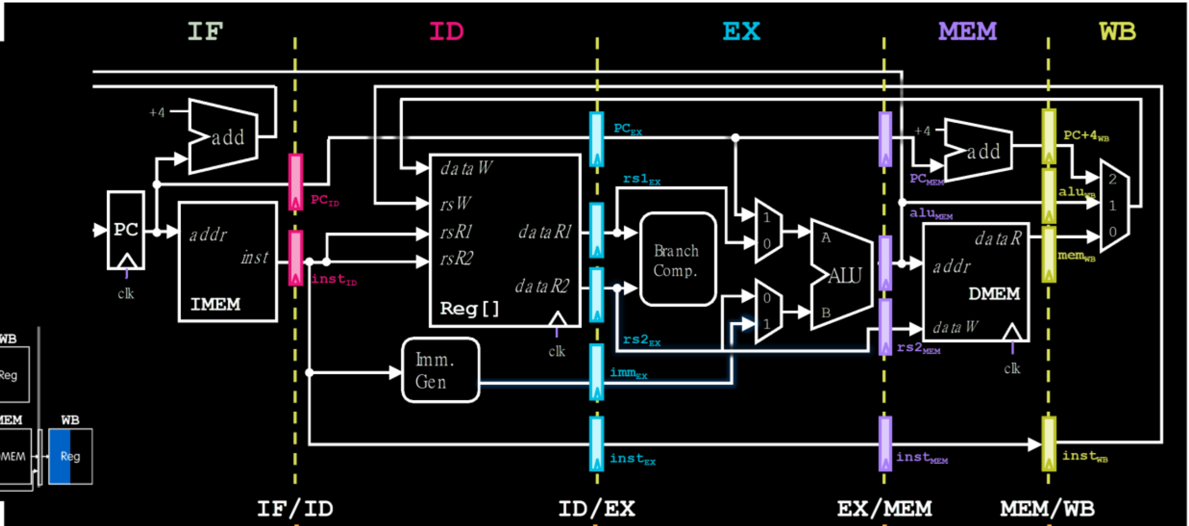


(decode for *rsW*)

Pipelining Datapath (9/10)

```
add t0,t1,t2

lw t0,8(t3)

or t3,t4,t5

sw t0,4(t3)

sll t6,t0,t3
```





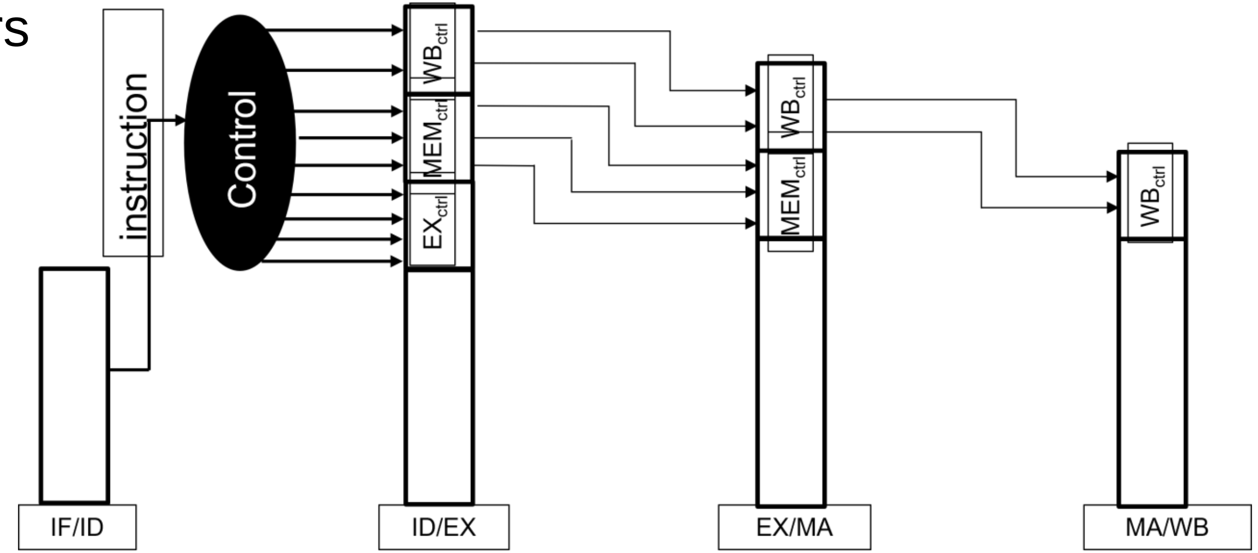
Suppose the CPU is currently executing this clock cycle.

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

The leftmost stage (**IF**) contains the most recent instruction. On the next clock cycle, pipeline registers carry the instruction/data to the next stage (**ID**).

Pipelining Datapath(10/10)

- Like the single-cycle CPU, control is usually computed during instruction decode (ID)
  - Control information for later stages is stored in pipeline registers



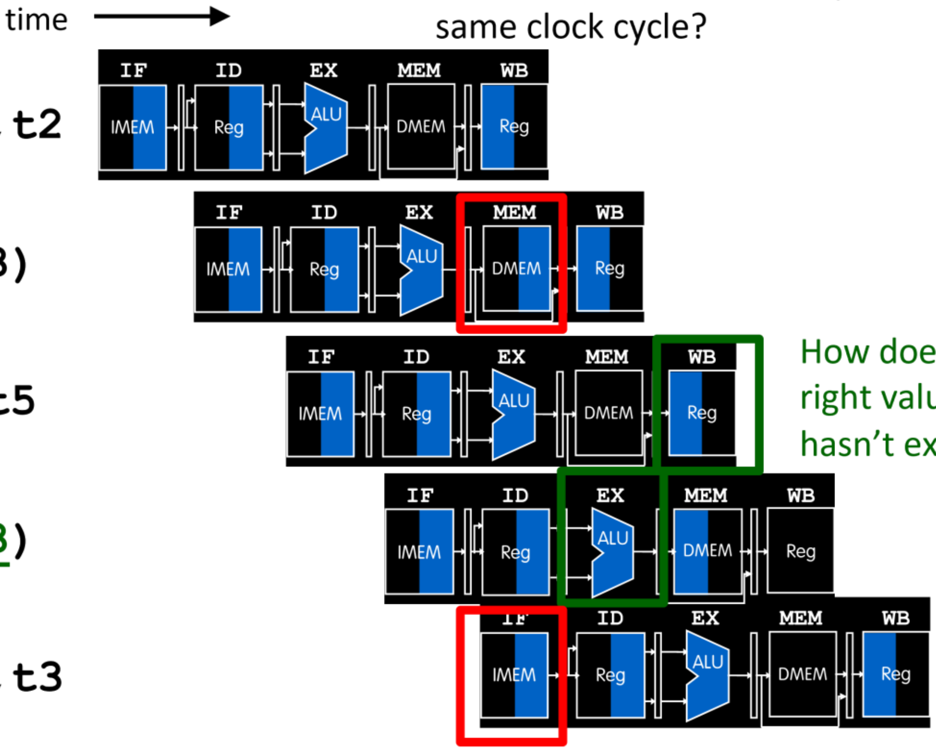
Outline

- Pipelining
- Pipelining Execution
- Pipelining Datapath
- Pipelining Hazard
- Structural Hazard

Pipelining Hazard (1/2)

Can we read from memory twice in the same clock cycle?

```
add t0,t1,t2

lw t0,8(t3)

or t3,t4,t5

sw t0,4(t3)

sll t6,t0,t3
```



How does **sw**'s **EX** stage get the right value of **t3** if **or**'s **WB** stage hasn't executed yet??

How do branches work???

Pipelining Hazard (2/2)

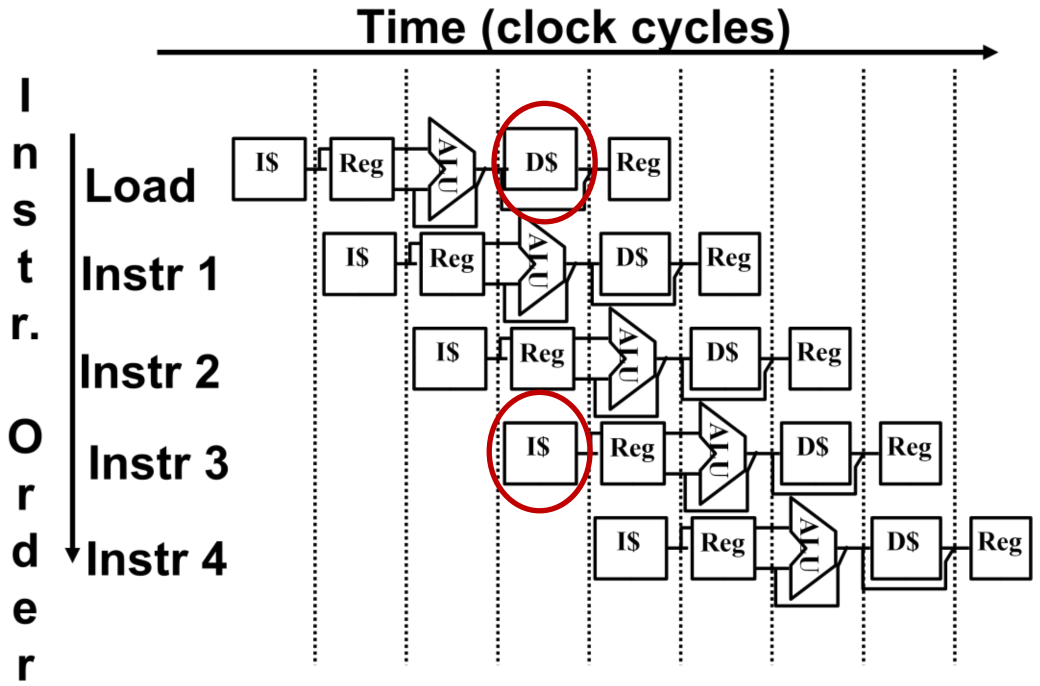
- Limits to pipelining
  - Hazards result in pipeline “**stalls**” or **“bubbles**”
  - **Structural hazards:**
    - Multiple instructions in the pipeline compete for access to a single physical resource
  - **Control hazards:**
    - Pipelining of branches causes later instruction fetches to wait for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

Outline

- Pipelining
- Pipelining Execution
- Pipelining Datapath
- Pipelining Hazard
- Structural Hazard

Structural Hazard (1/6)

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



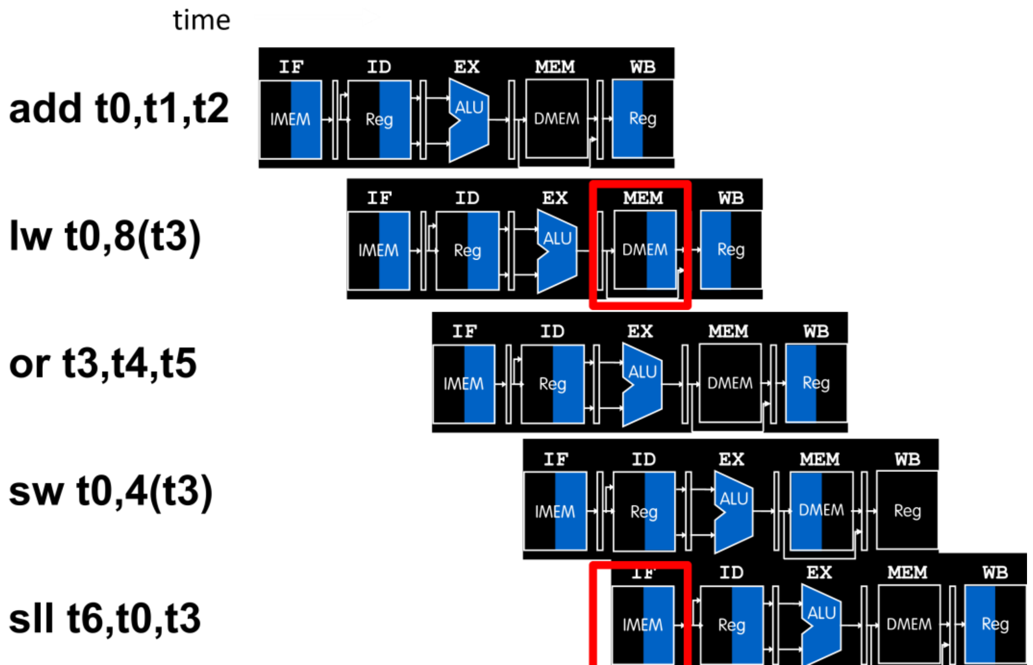
Read the same memory twice in the same clock cycle

Structural Hazard (2/6)

- **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 (3/6)

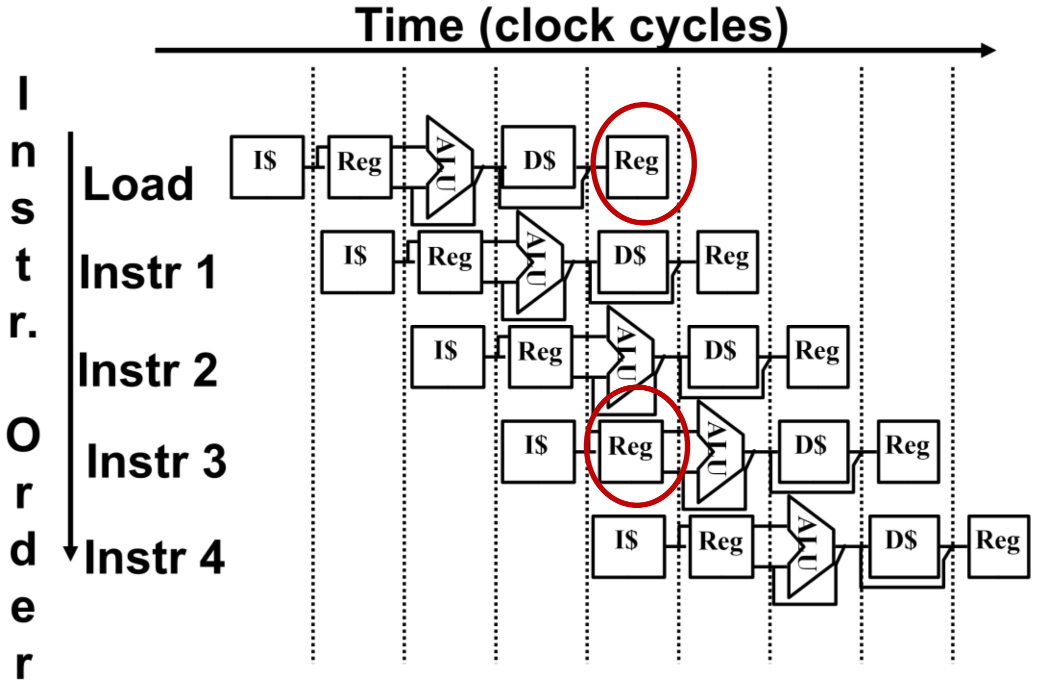
- **Structural Hazard #1: Single Memory**
  - Structural hazard if IMEM, DMEM were same hardware



RV32I separates IMEM and DMEM to avoid structural hazard

Structural Hazard (4/6)

- **Structural Hazard #2: Registers**



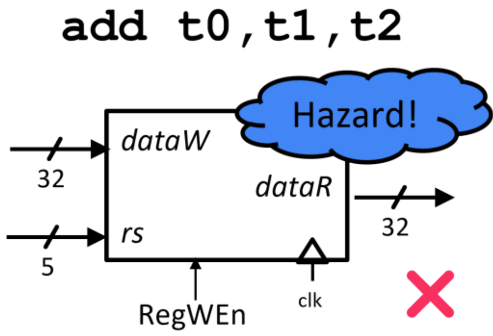
Read and write to registers simultaneously

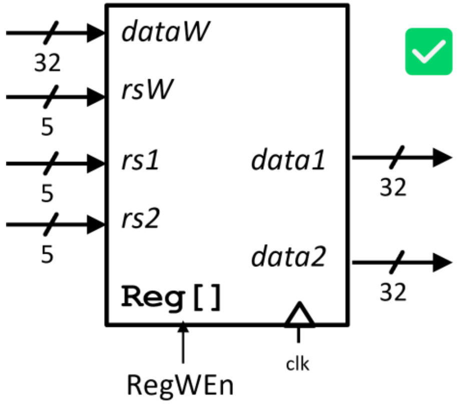
Structural Hazard (5/6)

- **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 (6/6)

- **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





Conclusion

- Optimal Pipeline
  - Each stage is executing part of an instruction each clock cycle
  - One instruction finishes during each clock cycle
  - On average, execute far more quickly
- What makes this work?
  - Similarities between instructions allow us to use same stages for all instructions