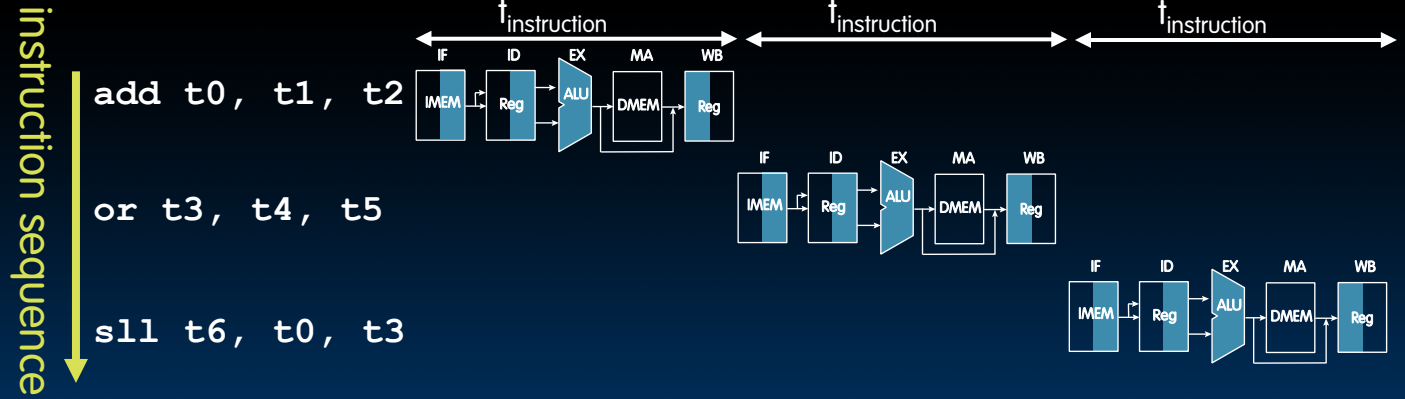
Pipelining RISC-V

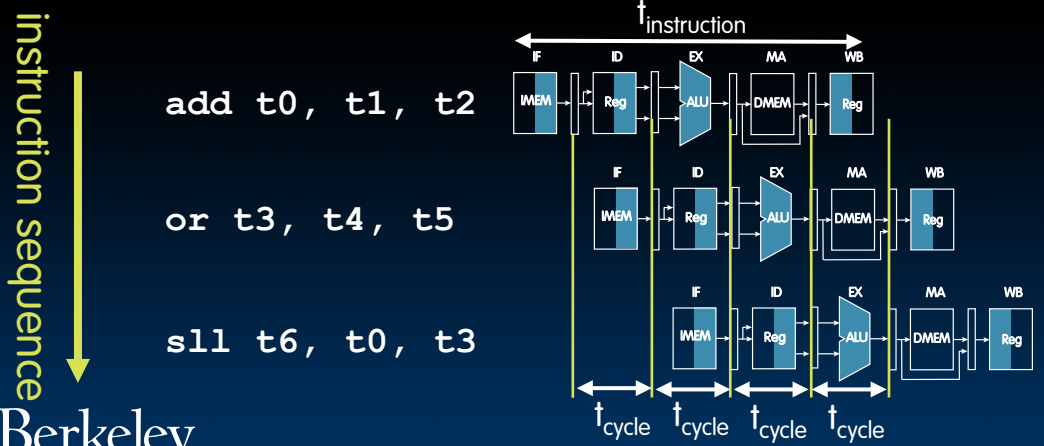
'Sequential' RISC-V Datapath

Phase	Pictogram	t_{step} Serial
Instruction Fetch	IMEM	200 ps
Reg Read	Reg	100 ps
ALU	ALU	200 ps
Memory	DMEM	200 ps
Register Write	Reg	100 ps
$t_{instruction}$	IMEM Reg ALU DMEM Reg	800 ps

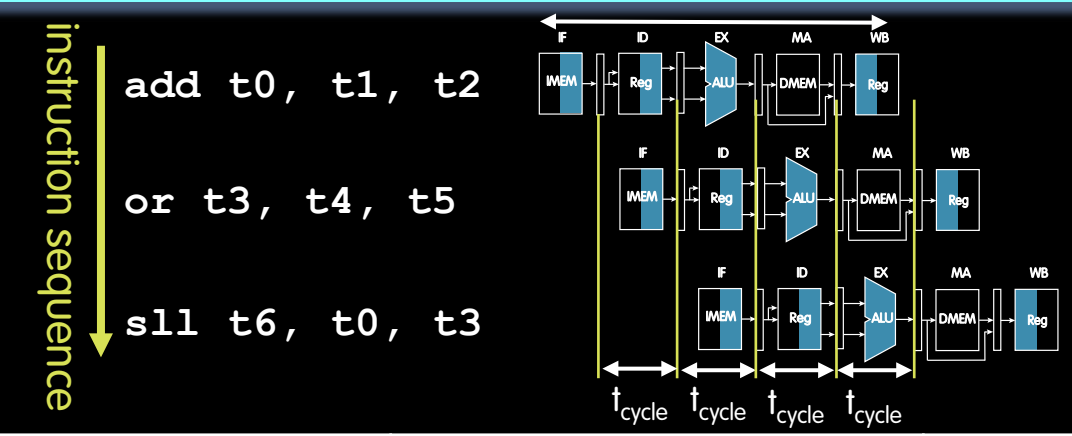


Pipelined RISC-V Datapath

Phase	Pictogram	t_{step} Serial	t_{cycle} Pipelined
Instruction Fetch		200 ps	200 ps
Reg Read		100 ps	200 ps
ALU		200 ps	200 ps
Memory		200 ps	200 ps
Register Write		100 ps	200 ps
$t_{instruction}$		800 ps	1000 ps



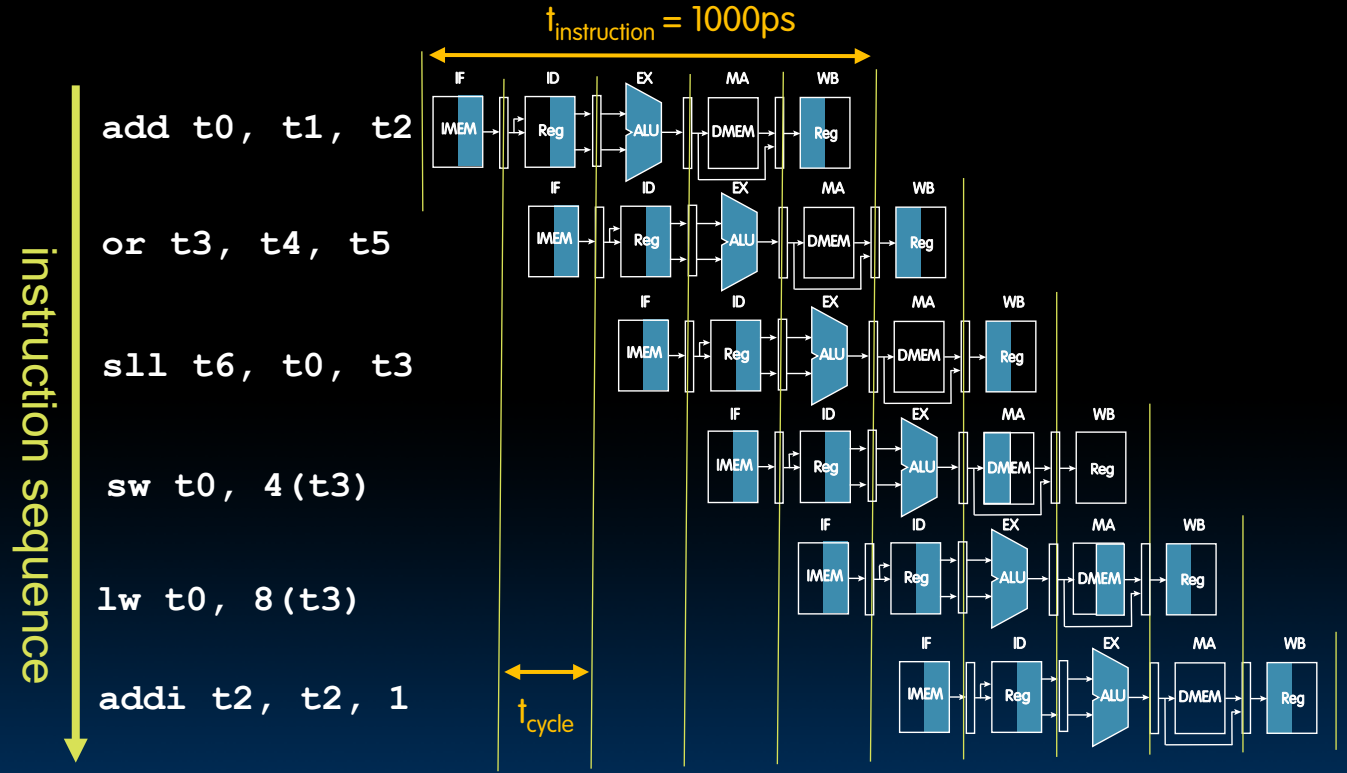
Pipelined RISC-V Datapath



	Single Cycle	Pipelined
Timing	t_{step} = 100 … 200 ps	t_{cycle} = 200 ps
	Register access only 100 ps	All cycles same length
Instruction time, $t_{instruction}$	= t_{cycle} = 800 ps	1000 ps
CPI (Cycles Per Instruction)	~1 (ideal)	~1 (ideal), <1 (actual)
Clock rate, f_s	1/800 ps = 1.25 GHz	1/200 ps = 5 GHz
Relative speed	1 x	4 x

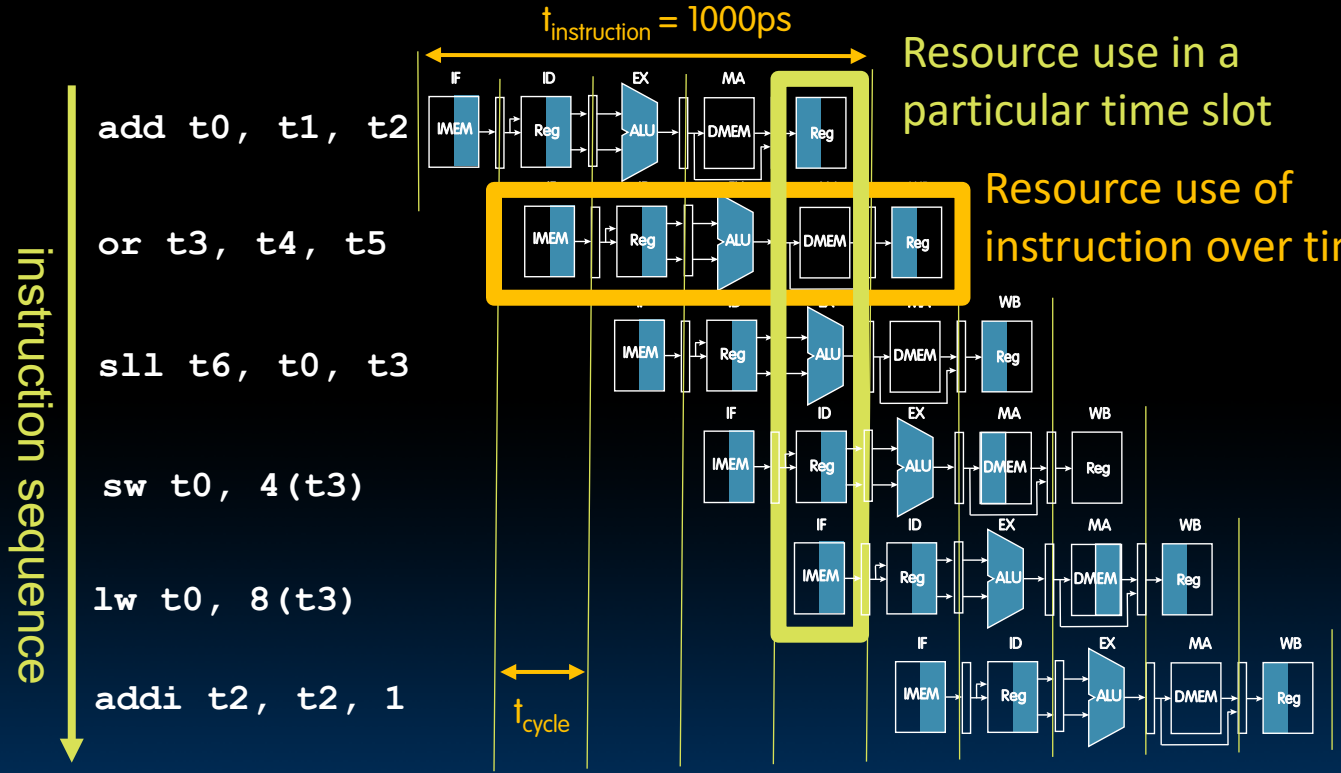
Sequential vs. Simultaneous

- What happens sequentially and what simultaneously?



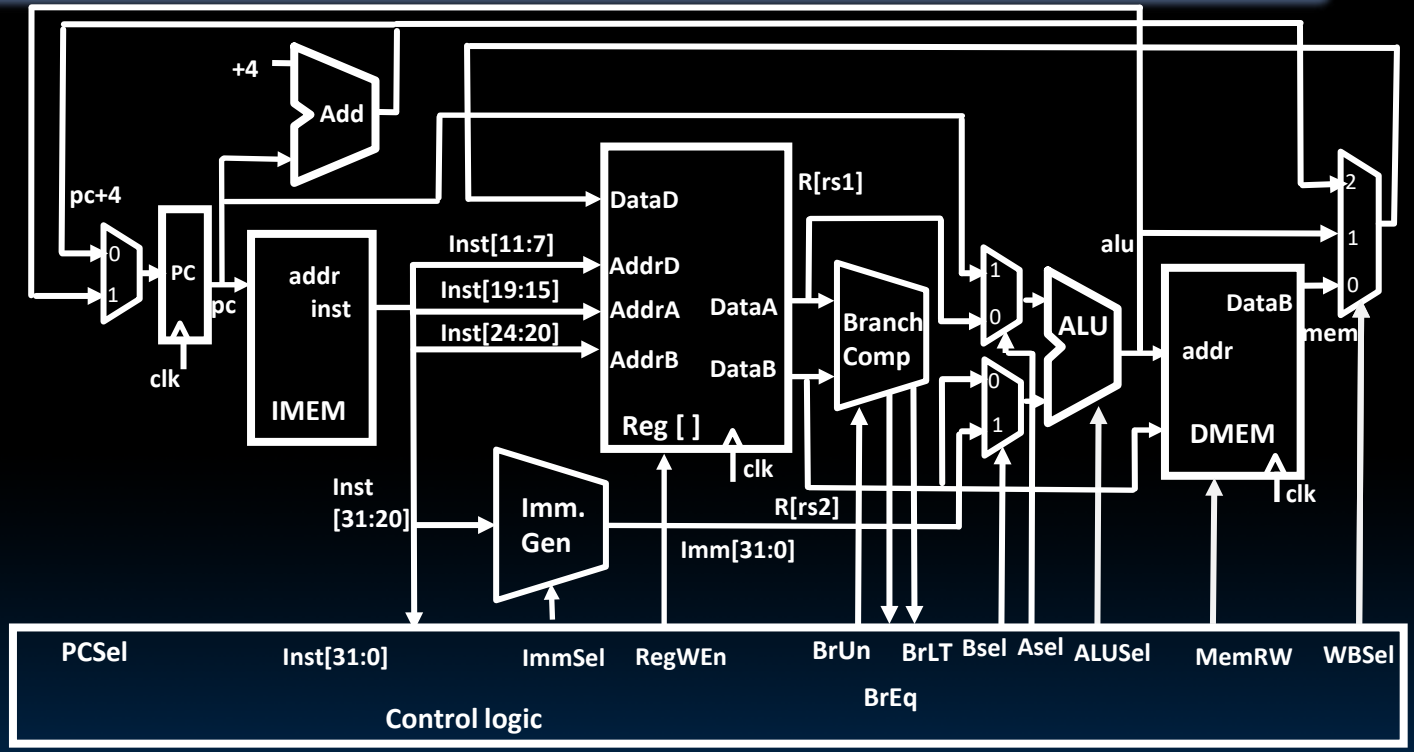
Sequential vs. Simultaneous

- What happens sequentially and what simultaneously?

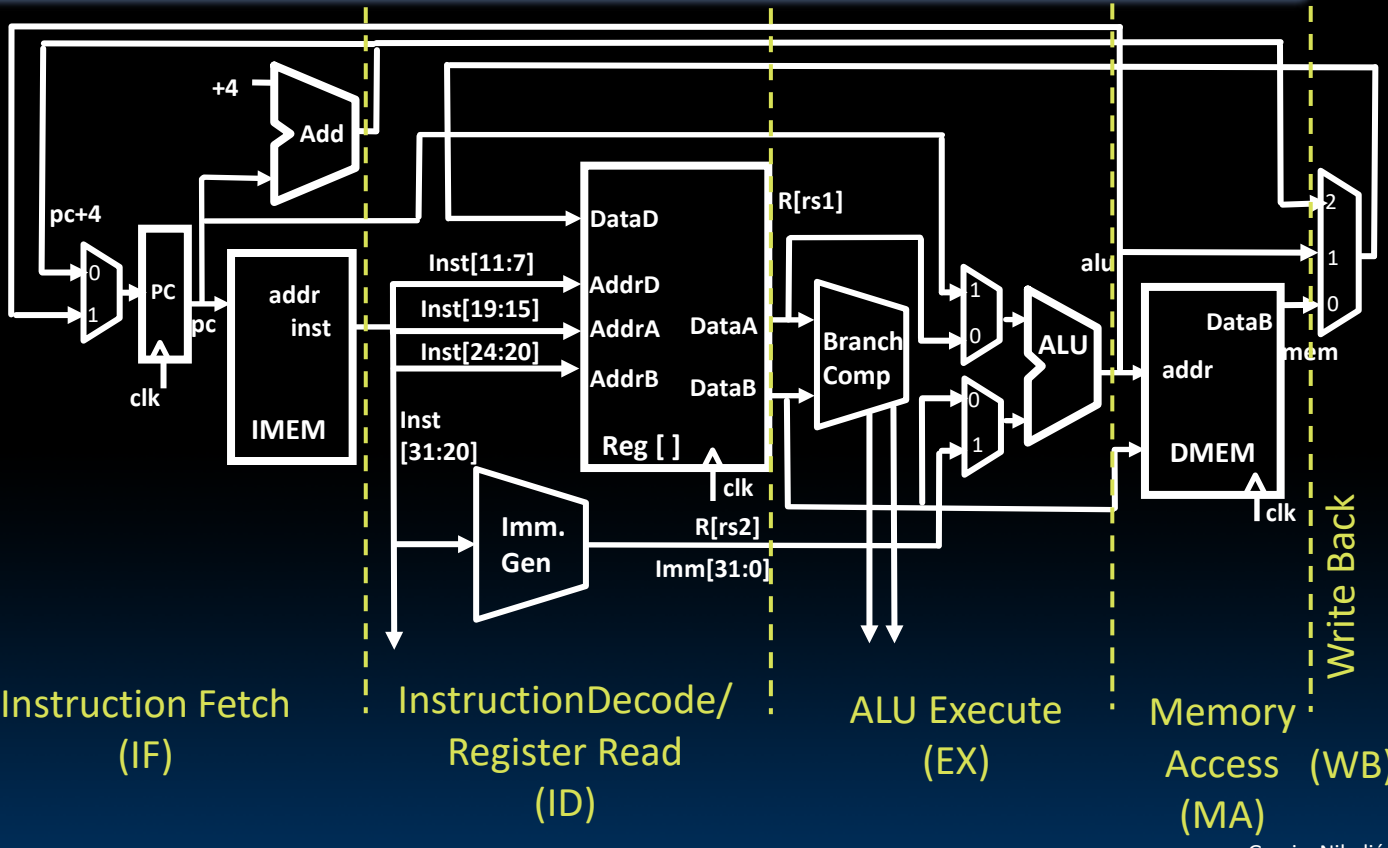


Pipelining Datapath

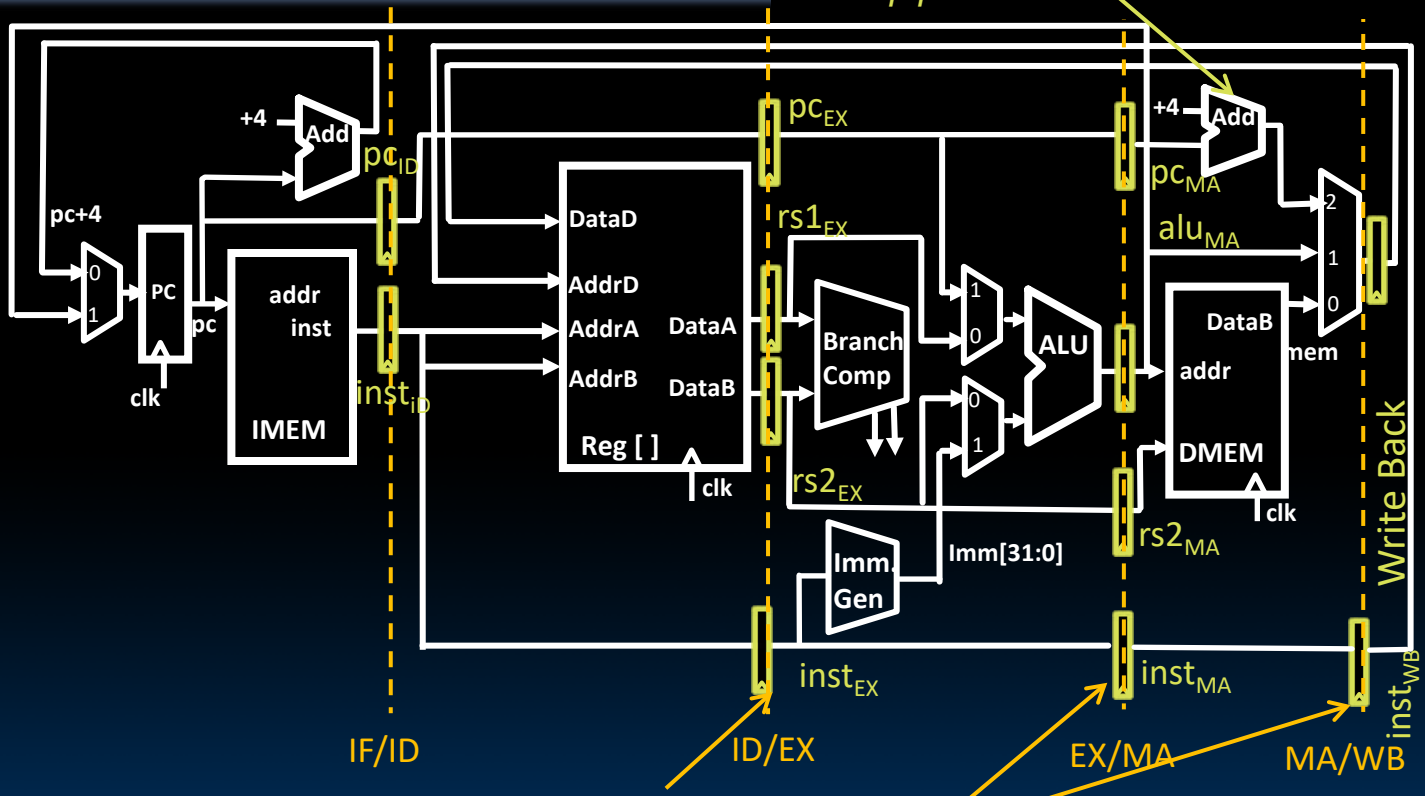
Single-Cycle RV32I Datapath



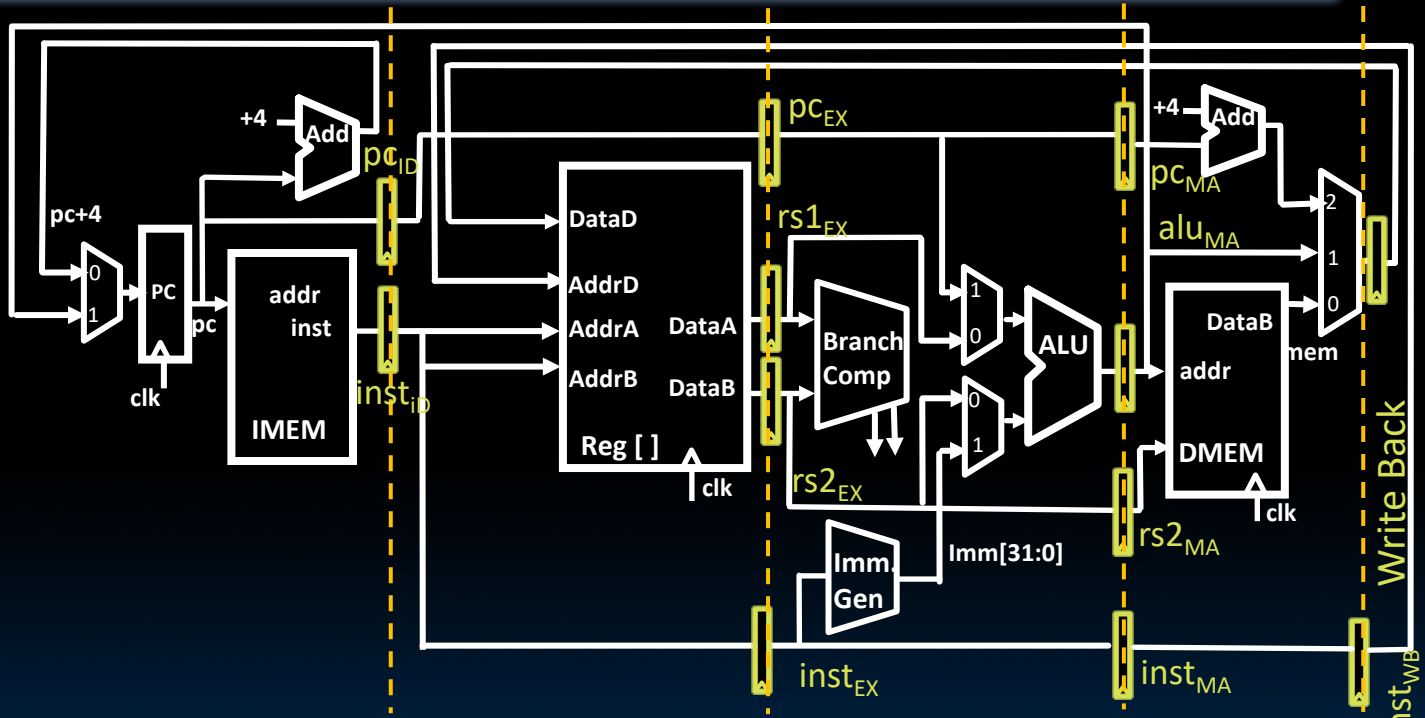
Single-Cycle RV32I Datapath



Pipelined RV32I Datapath



Pipelined RV32I Datapath

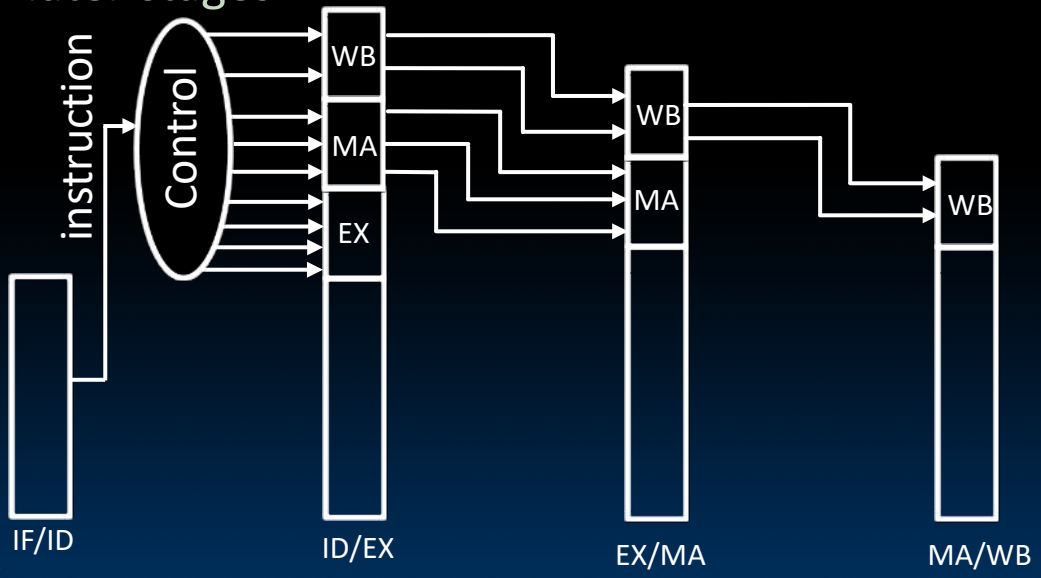



```
lw t0,8(t3)      sw t0,4(t3)   slt t6,t0,t3   or t3,t4,t5
```

Pipeline registers separate stages, hold data for each instruction in flight

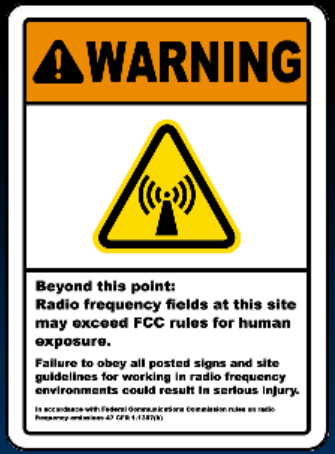
Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation
  - Information is stored in pipeline registers for use by later stages



Pipeline Hazards

Hazards Ahead!



Pipelining Hazards

A *hazard* is a situation that prevents starting the next instruction in the next clock cycle

1) *Structural hazard*
   - A required resource is busy (e.g. needed in multiple stages)
2) *Data hazard*
   - Data dependency between instructions
   - Need to wait for previous instruction to complete its data read/write
3) *Control hazard*
   - Flow of execution depends on previous instruction

Structural Hazard

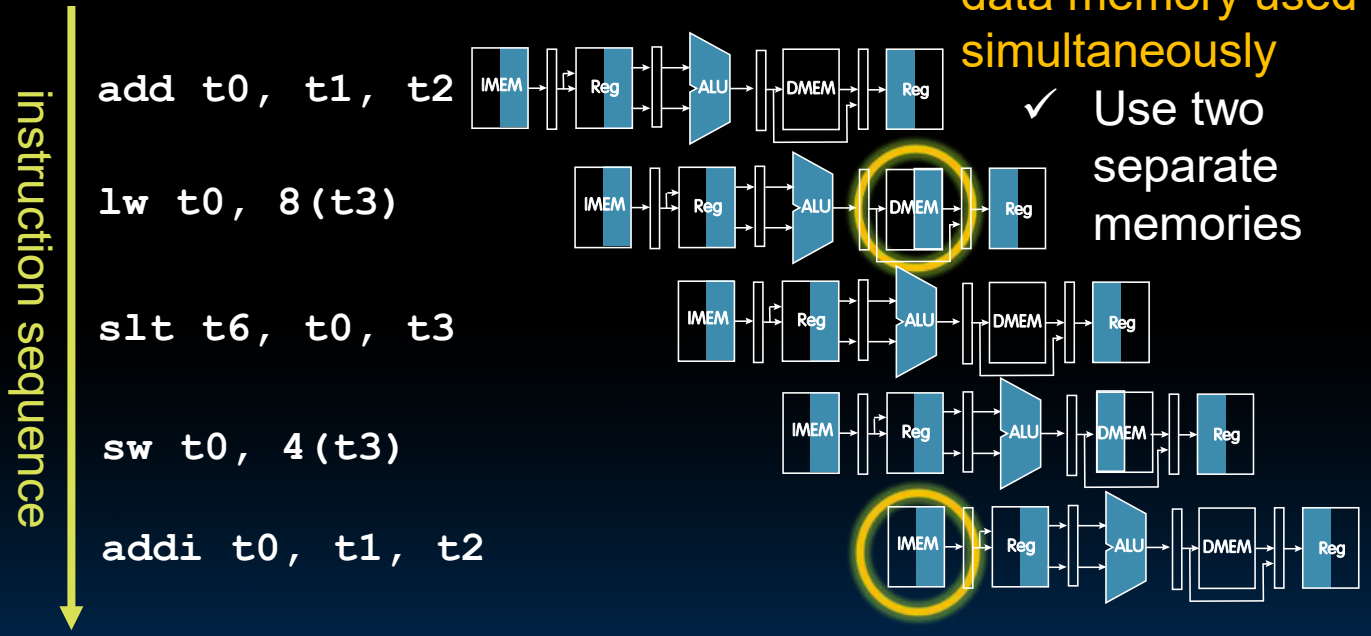
- **Problem:** Two or more instructions in the pipeline compete for access to a single physical resource

- **Solution 1:** Instructions take it in turns to use resource, some instructions have to stall
- **Solution 2:** Add more hardware to machine
- Can always solve a structural hazard by adding more hardware

Regfile Structural Hazards

- Each instruction:
  - Can read up to two operands in decode stage
  - Can write one value in writeback stage
- Avoid structural hazard by having separate "ports"
  - Two independent read ports and one independent write port
- Three accesses per cycle can happen simultaneously

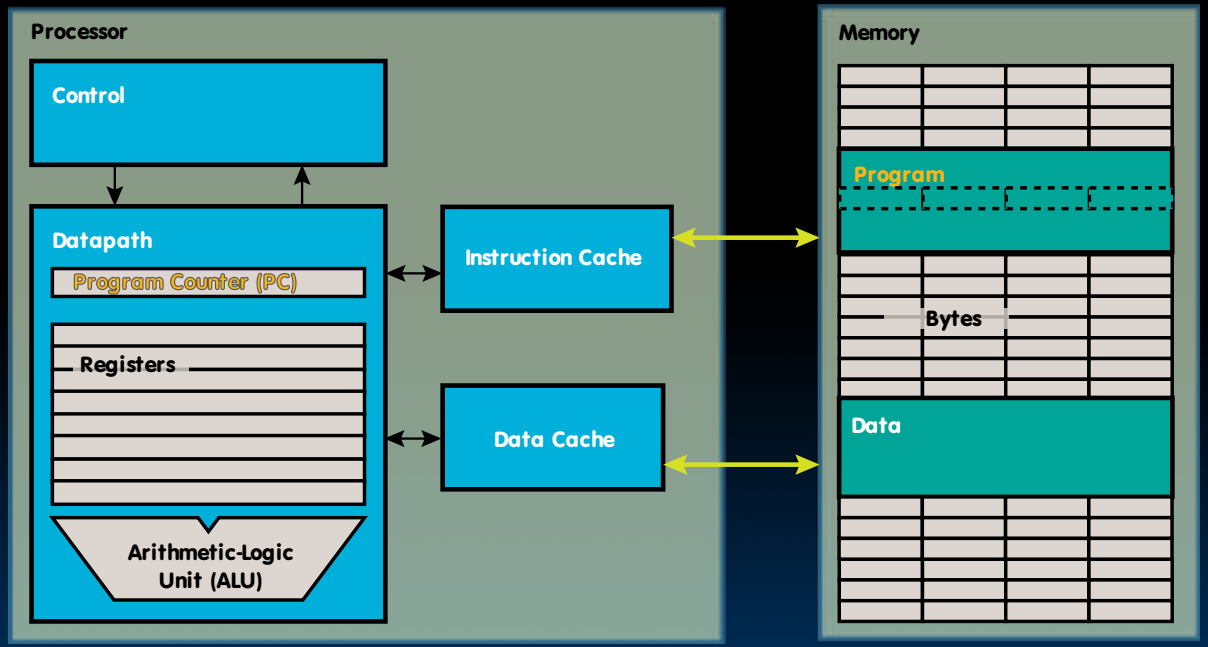
Structural Hazard: Memory Access



- Instruction and data memory used simultaneously
  - ✓ Use two separate memories

Instruction and Data Caches

- Fast, on-chip memory, separate for instructions and data



Structural Hazards – Summary

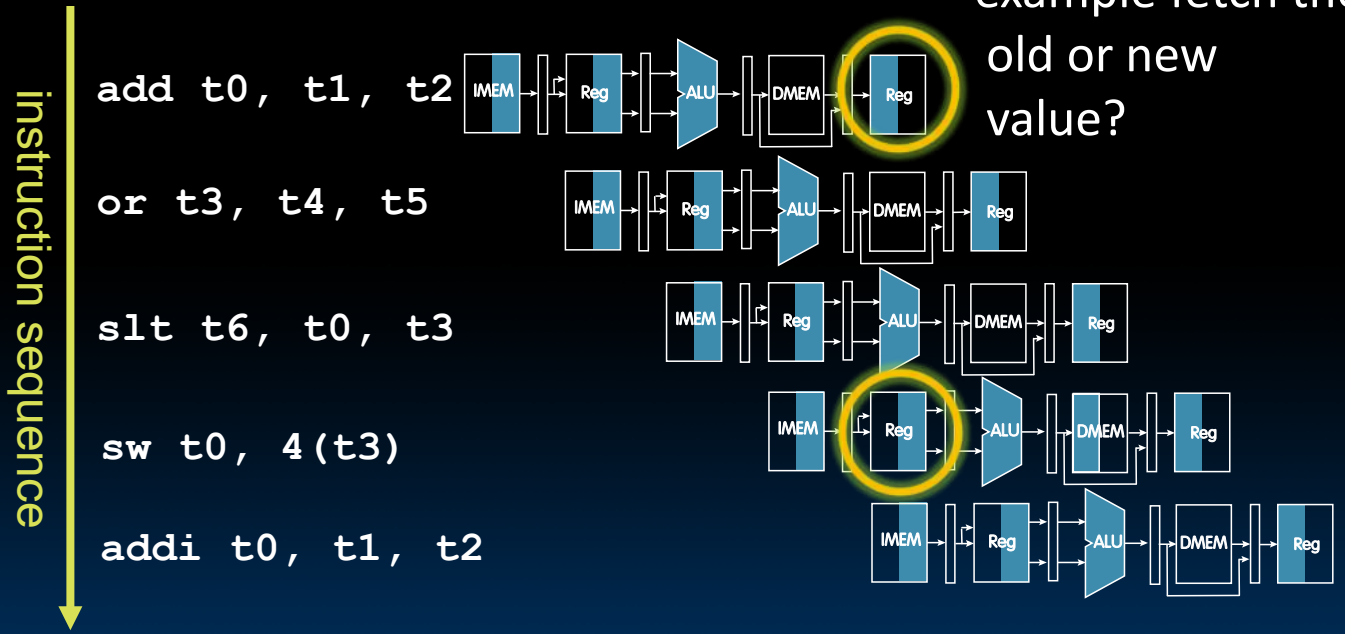
- Conflict for use of a resource
- In RISC-V pipeline with a single memory
  - Load/store requires data access
  - Without separate memories, instruction fetch would have to stall for that cycle
    - All other operations in pipeline would have to wait
- Pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches
- RISC ISAs (including RISC-V) designed to avoid structural hazards
  - e.g. at most one memory access/instruction

Data Hazards

Data Hazard: Register Access

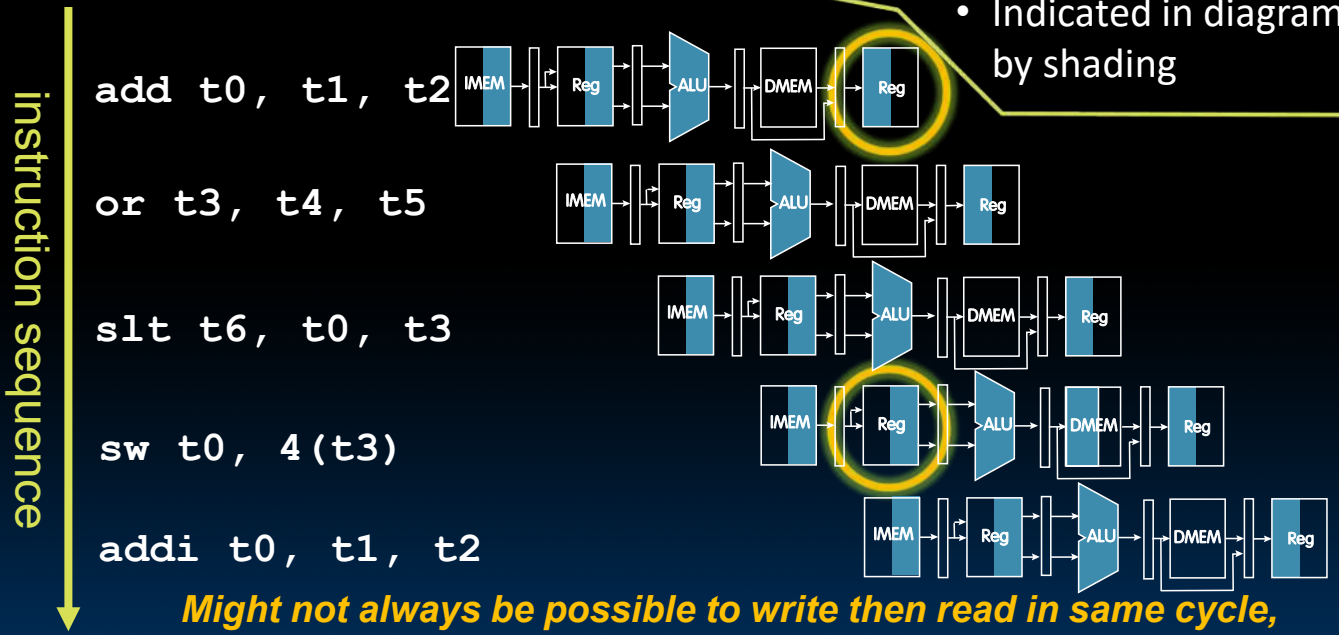
- Separate ports, but what if write to same register as read?

Does **`sw`** in the example fetch the old or new value?



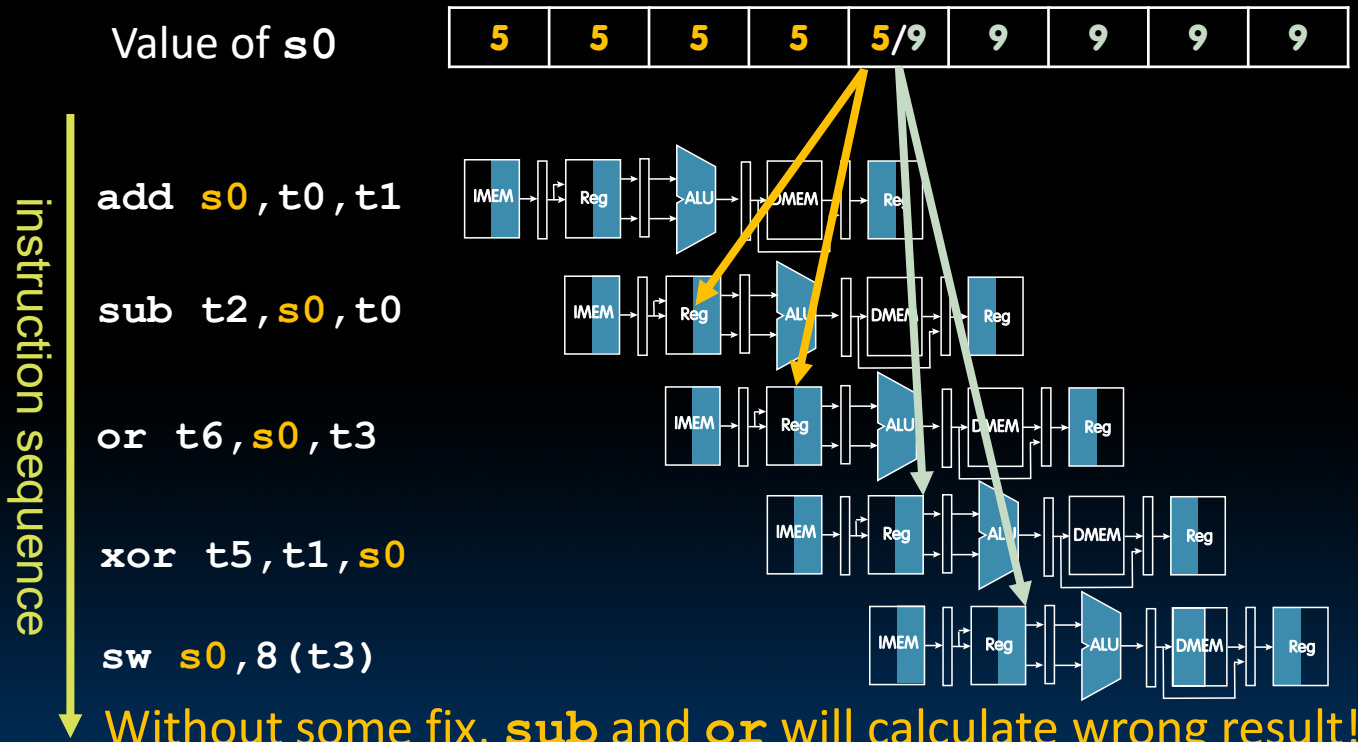
Data Hazard: Register Access

- Exploit high speed of register file (100 ps)
  1) WB updates value
  2) ID reads new value
- Indicated in diagram by shading



Might not always be possible to write then read in same cycle, especially in high-frequency designs. Check assumptions in any question.

Data Hazard: ALU Result



Without some fix, **sub** and **or** will calculate wrong result!

Solution 1: Stalling

- Problem: Instruction depends on result from previous instruction

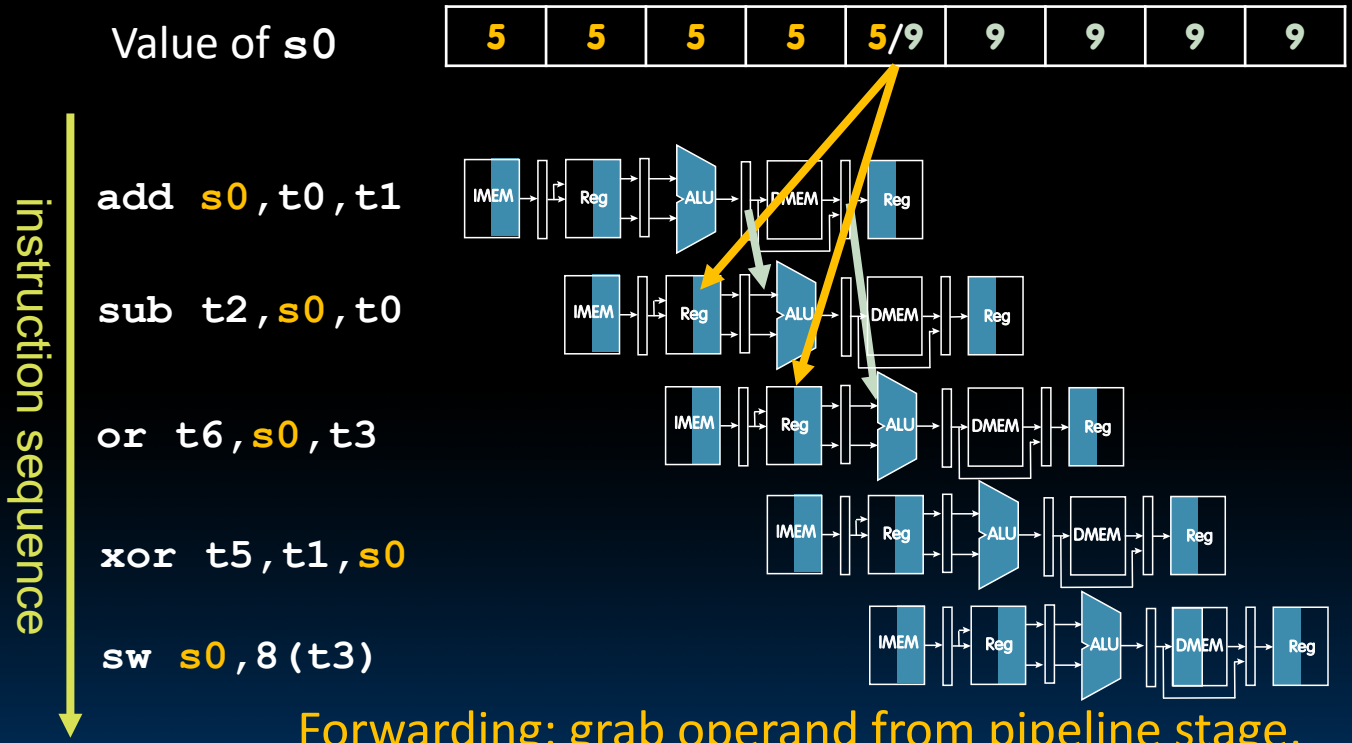
```
add  s0, t0, t1
sub  t2, s0, t3
```

- Bubble:
  - Effectively **nop**: Affected pipeline stages do "nothing"

Stalls and Performance

- Stalls reduce performance
  - But stalls are required to get correct results
- Compiler can arrange code or insert **`nop`**s (**`addi x0, x0, 0`**) to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Solution 2: Forwarding

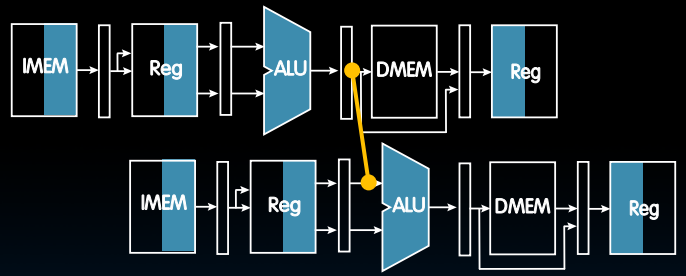


Forwarding: grab operand from pipeline stage, rather than register file

Forwarding (aka Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath

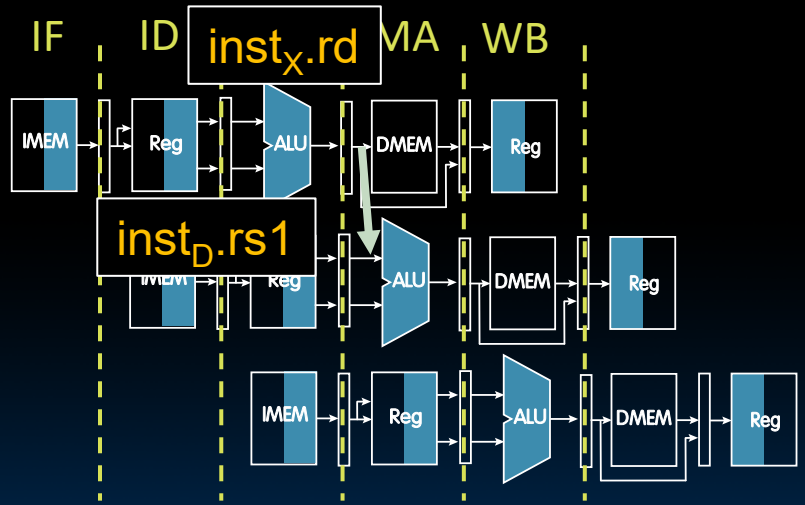
```
add s0, t0, t1
sub t2, s0, t3
```



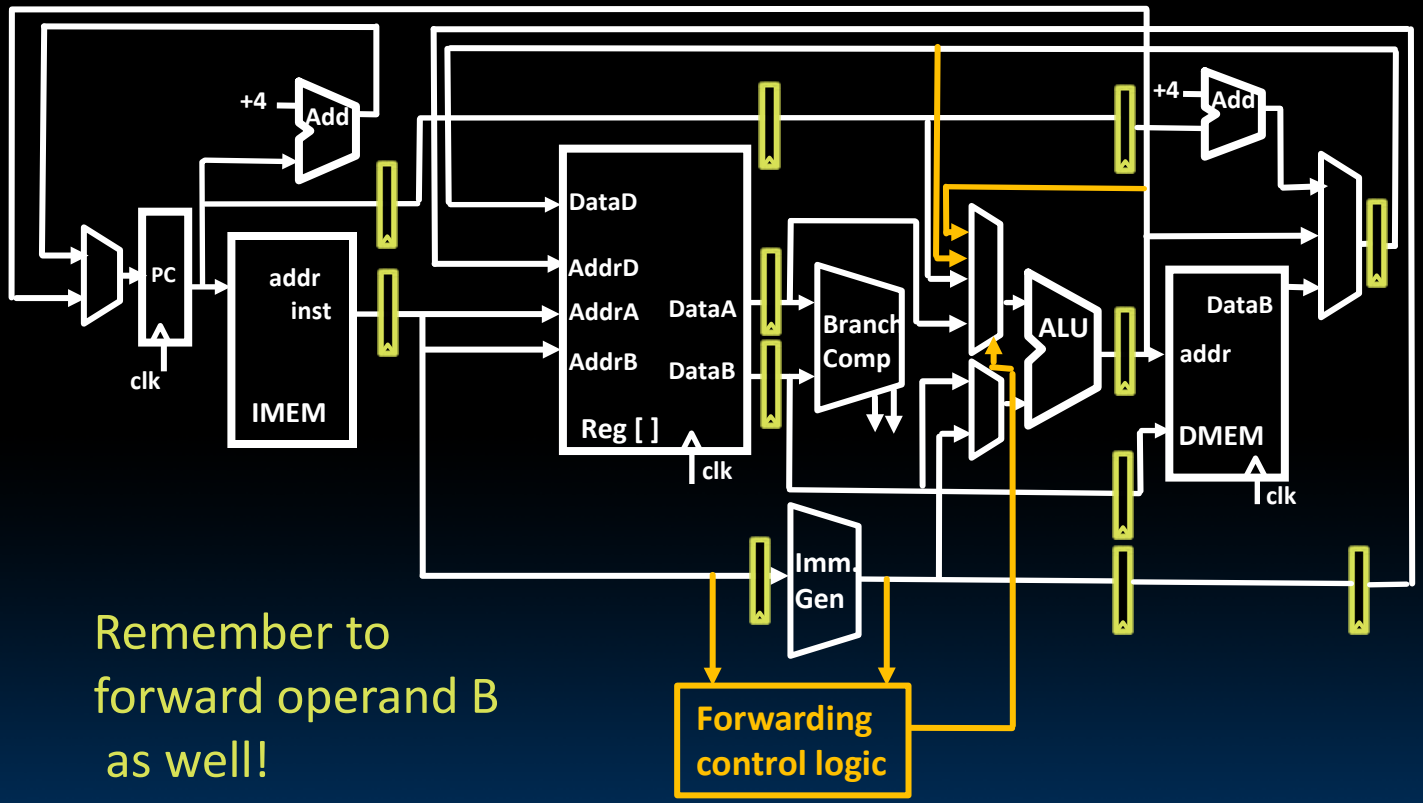
Data Needed for Forwarding (Example)

- Compare destination of older instructions in pipeline with sources of new instruction in decode stage.
- Must ignore writes to x0!

```
add t0, t0, t1
sub t3, t0, t5
sub t6, t0, t3
```



Pipelined RV32I Datapath



Remember to forward operand B as well!