

Pipelining RISC-V

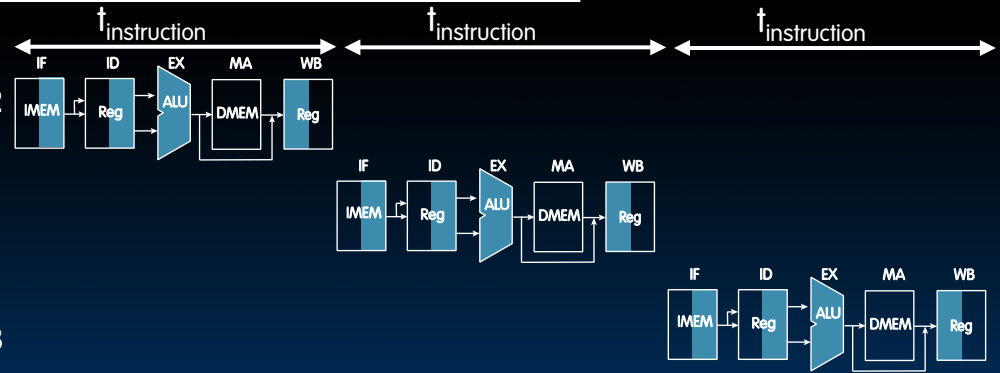
'Sequential' RISC-V Datapath

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







instruction sequence ↓

```

add t0, t1, t2
or t3, t4, t5
sll t6, t0, t3
    
```



Pipelined RISC-V Datapath

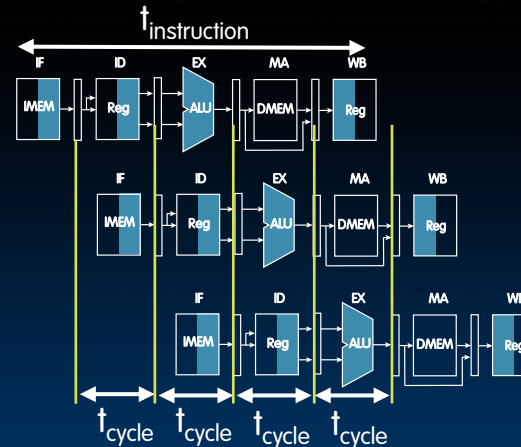
Phase	Pictogram	t_{step} Serial	t_{cycle} Pipelined
Instruction Fetch		200 ps	200 ps
Reg Read		100 ps	200 ps
ALU		200 ps	200 ps
Memory		200 ps	200 ps
Register Write		100 ps	200 ps
$t_{instruction}$		800 ps	1000 ps

instruction sequence

add t0, t1, t2

or t3, t4, t5

sll t6, t0, t3



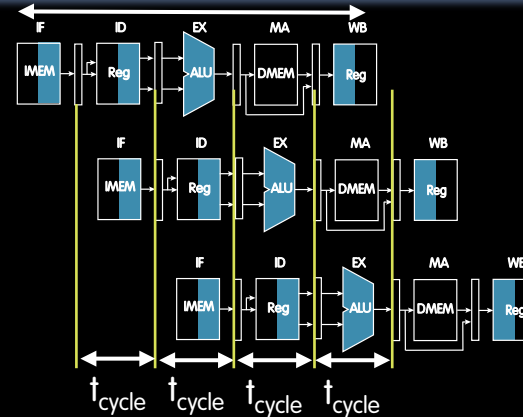
Pipelined RISC-V Datapath

instruction sequence

add t0, t1, t2

or t3, t4, t5

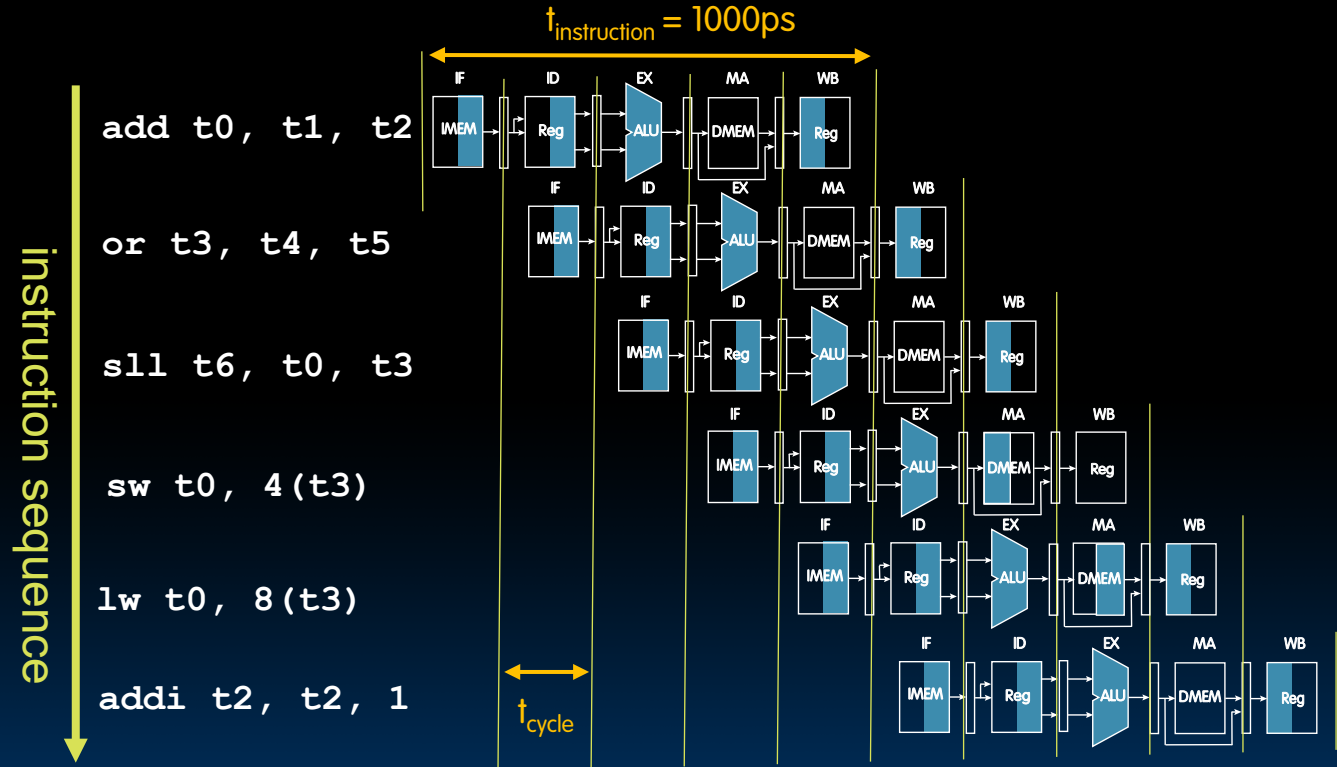
sll t6, t0, t3



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

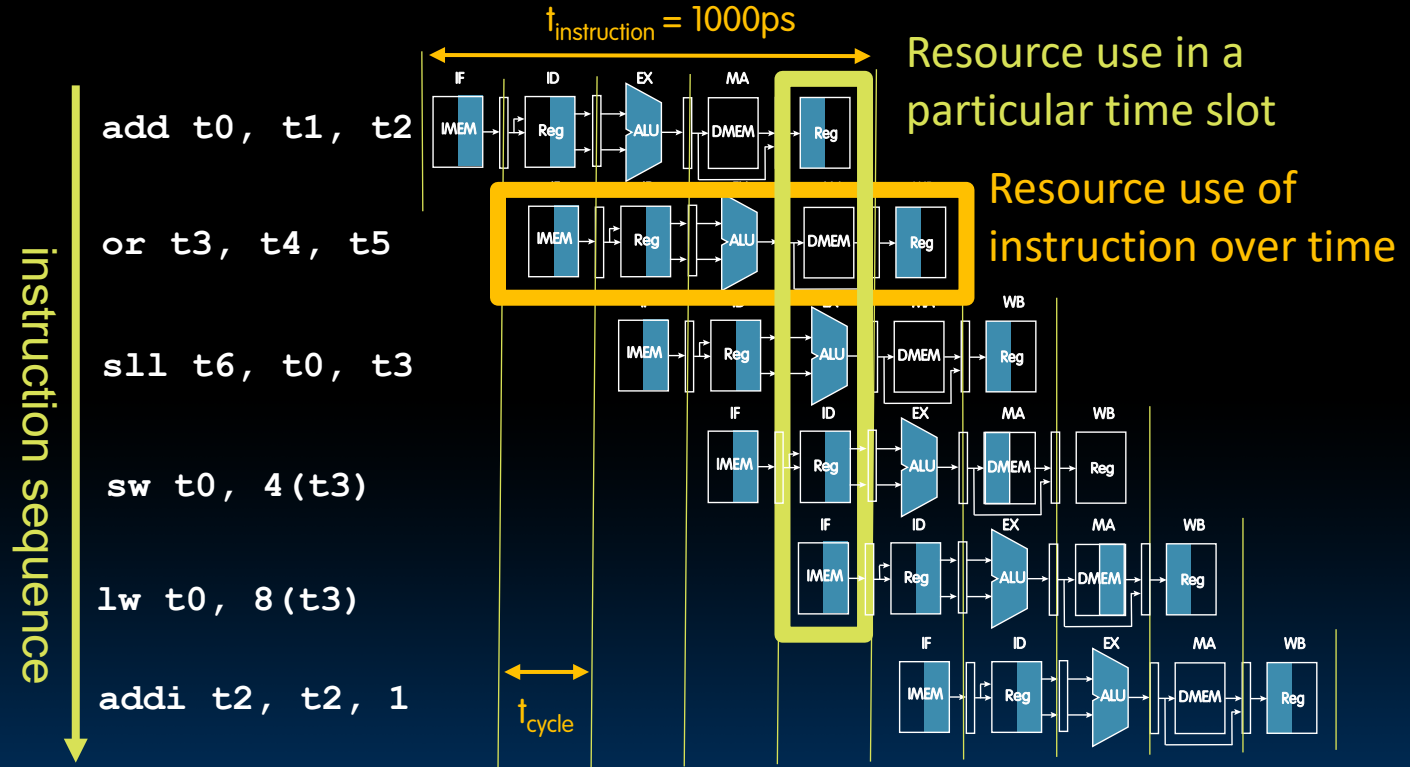
Sequential vs. Simultaneous

- What happens sequentially and what simultaneously?



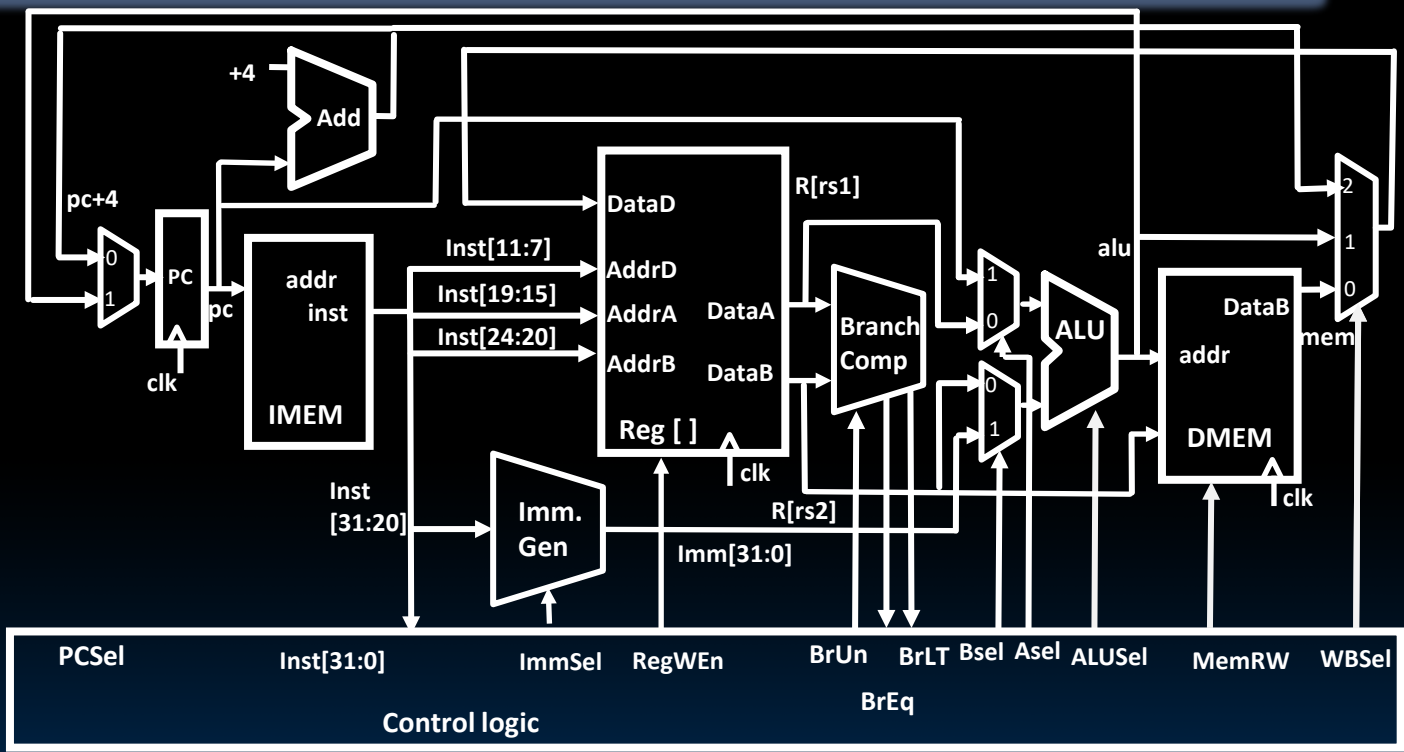
Sequential vs. Simultaneous

- What happens sequentially and what simultaneously?

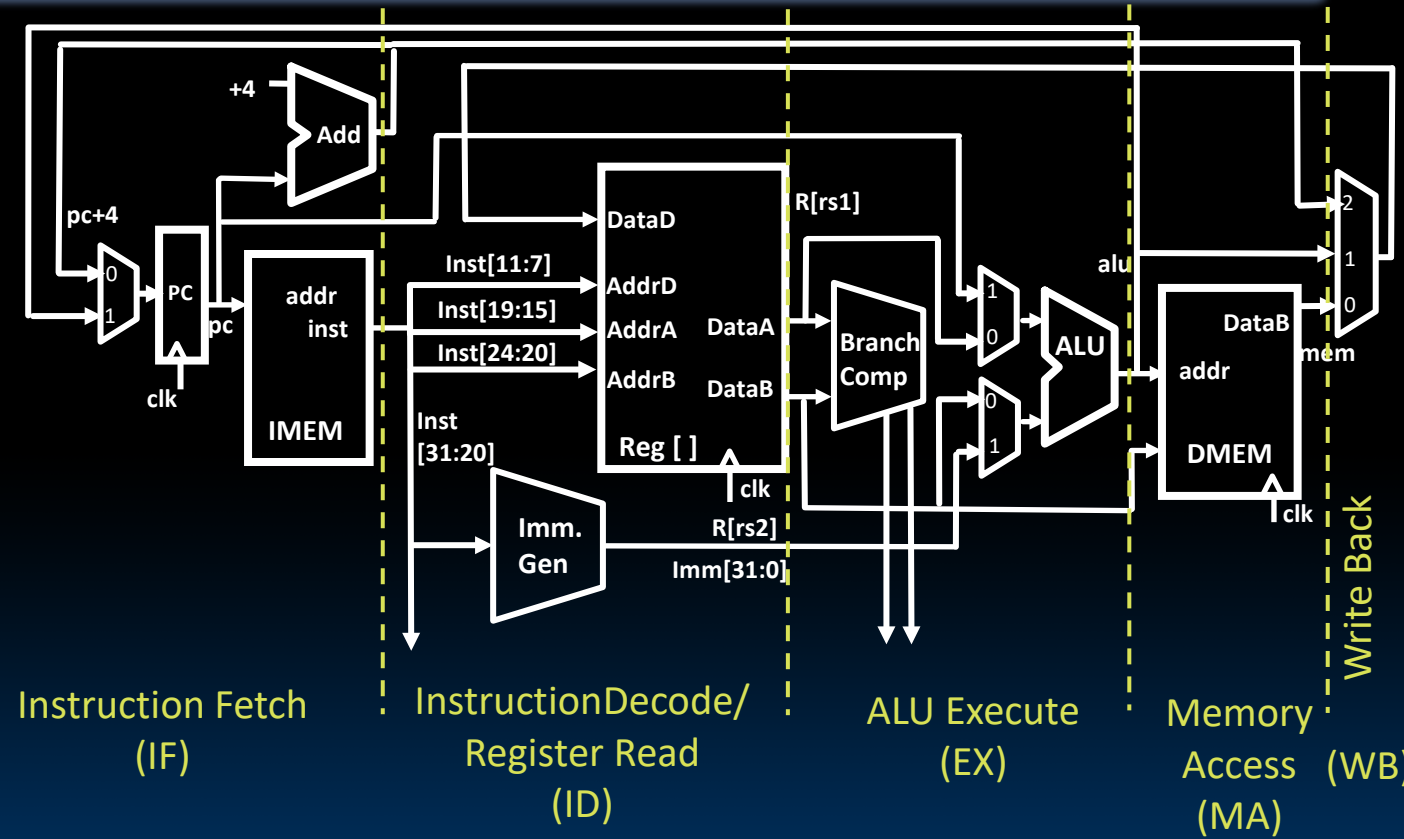


Pipelining Datapath

Single-Cycle RV32I Datapath

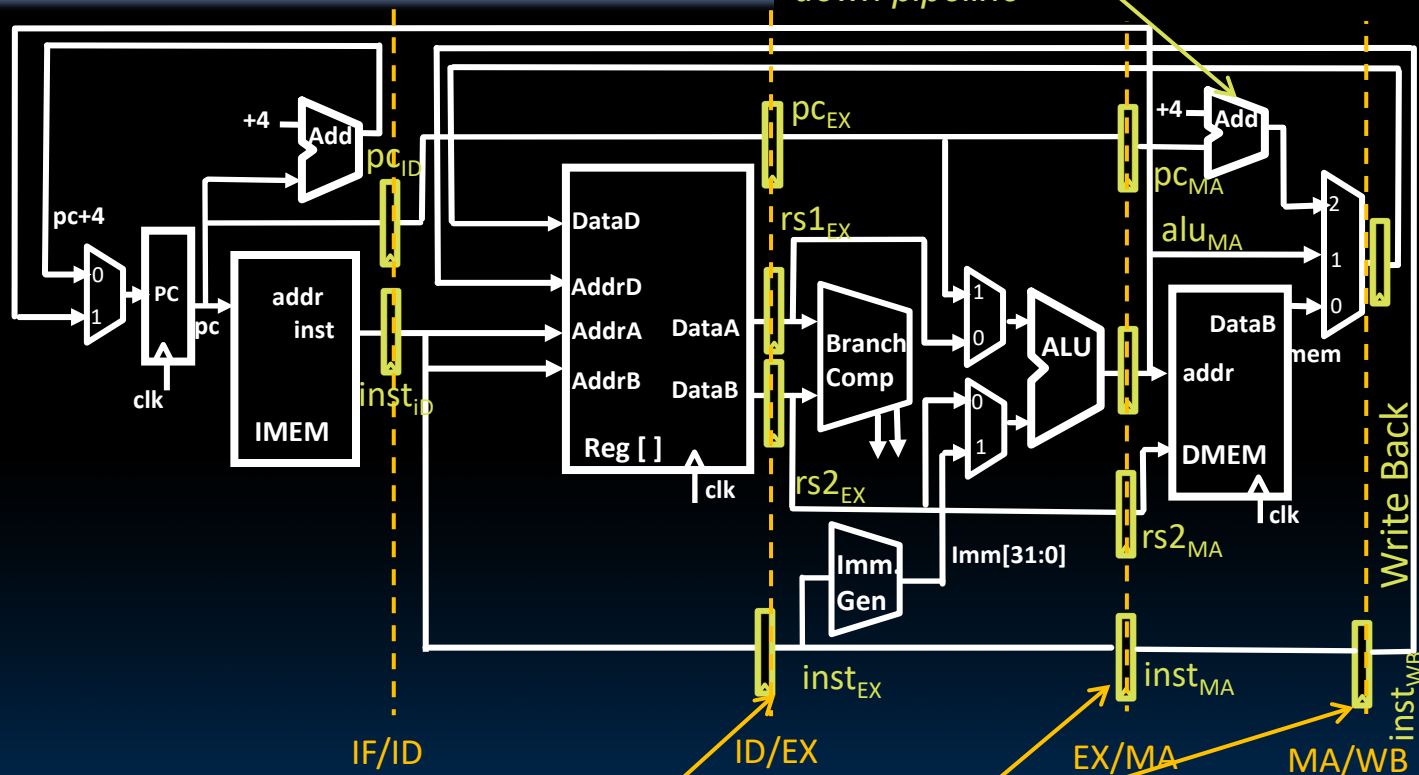


Single-Cycle RV32I Datapath



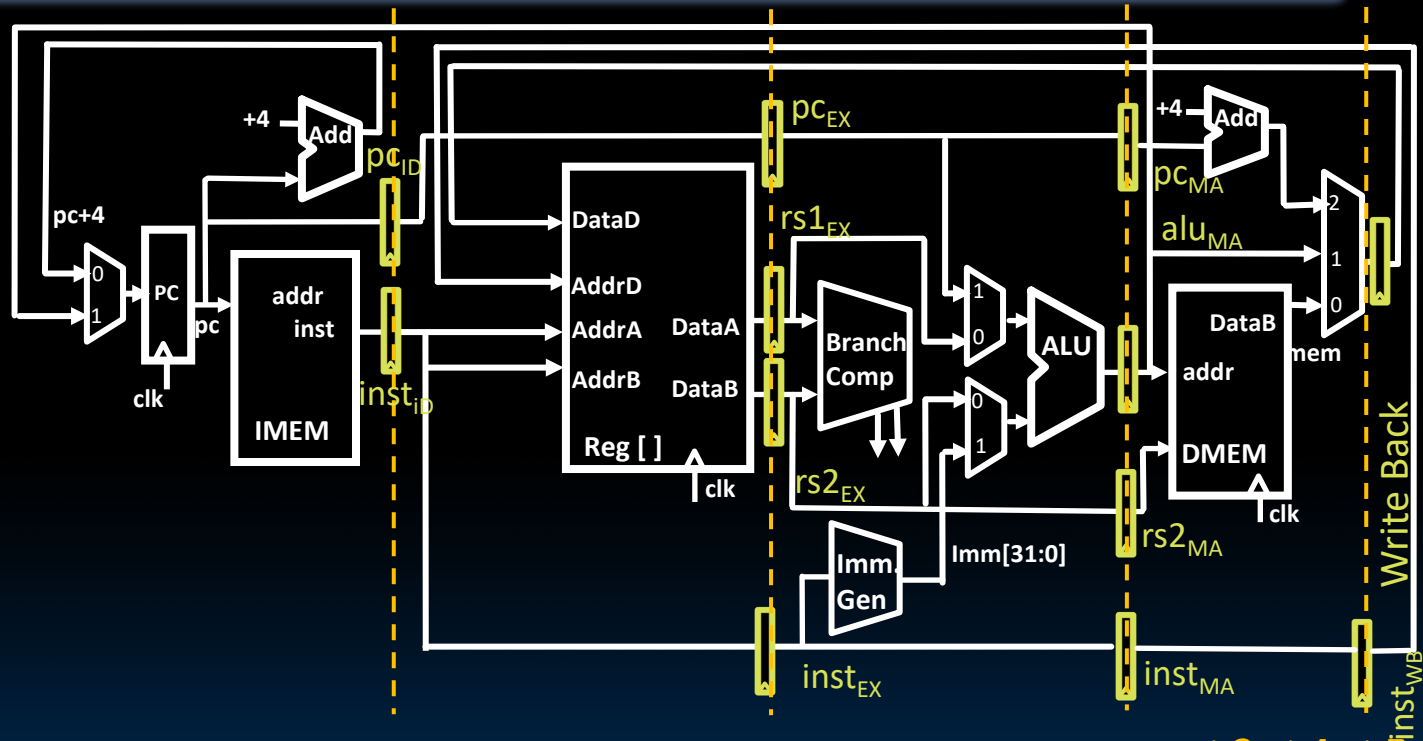
Pipelined RV32I Datapath

Recalculate PC+4 in M stage to avoid sending both PC and PC+4 down pipeline



Must pipeline instruction along with data, so control operates correctly in each stage

Pipelined RV32I Datapath

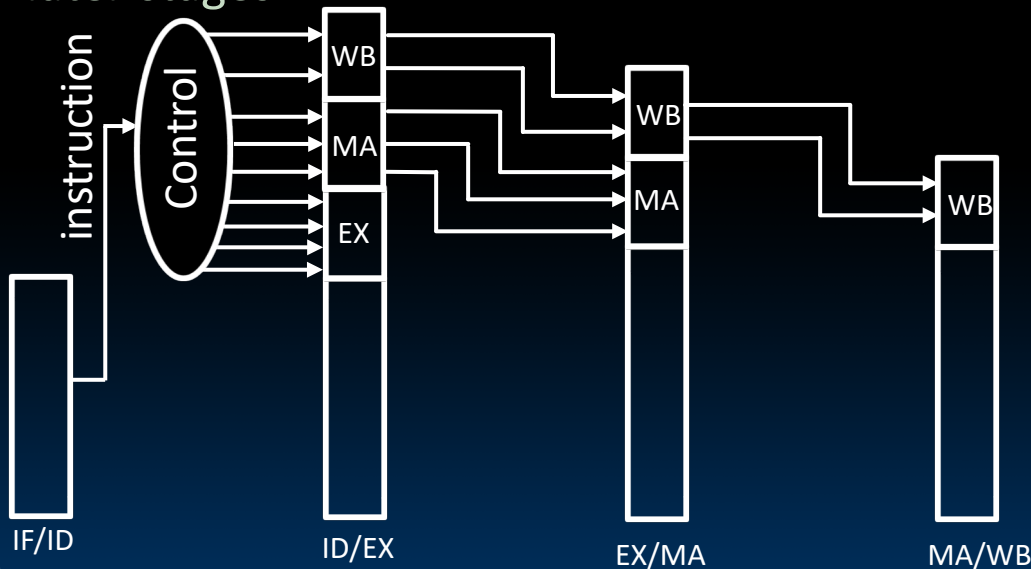


`lw t0, 8(t3)` `sw t0, 4(t3)` `slt t6, t0, t3` `or t3, t4, t5`

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

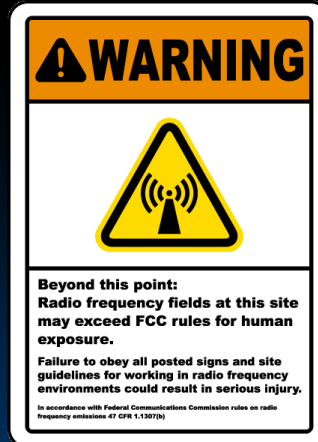
Pipelined Control

- 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

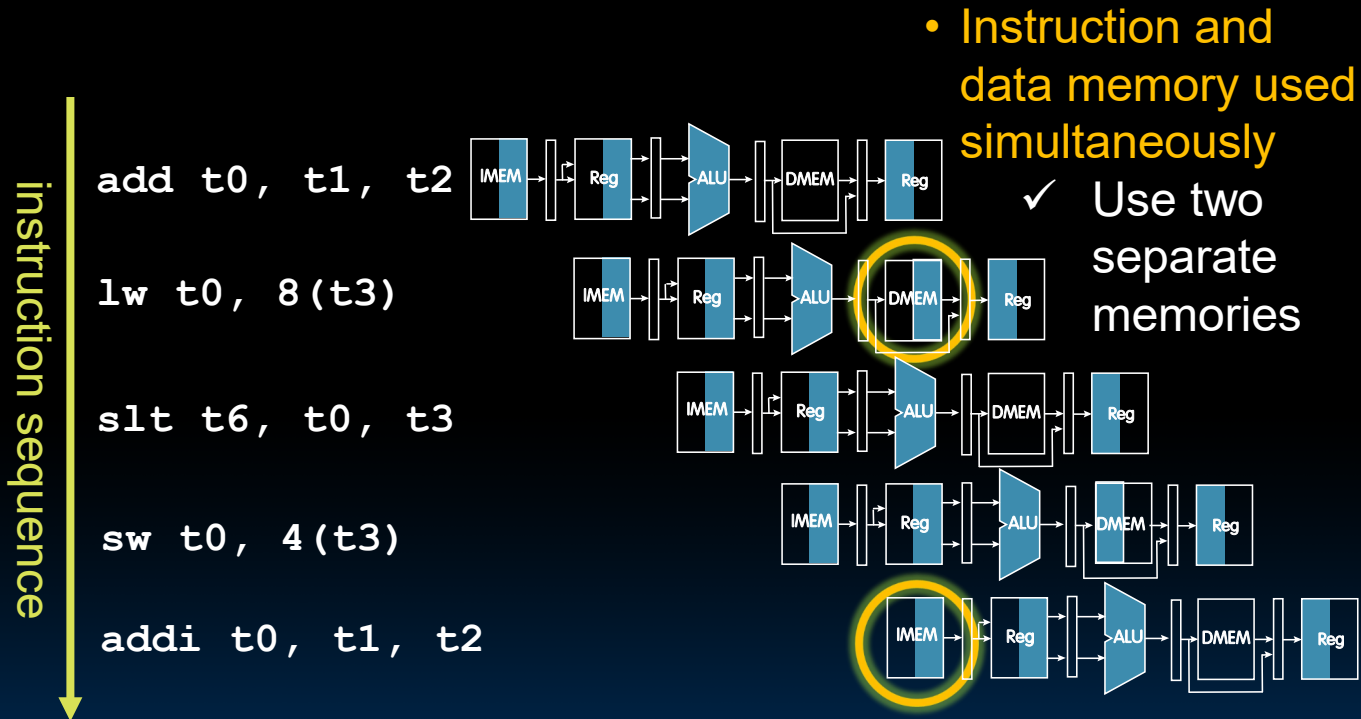
Structural Hazard

- **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

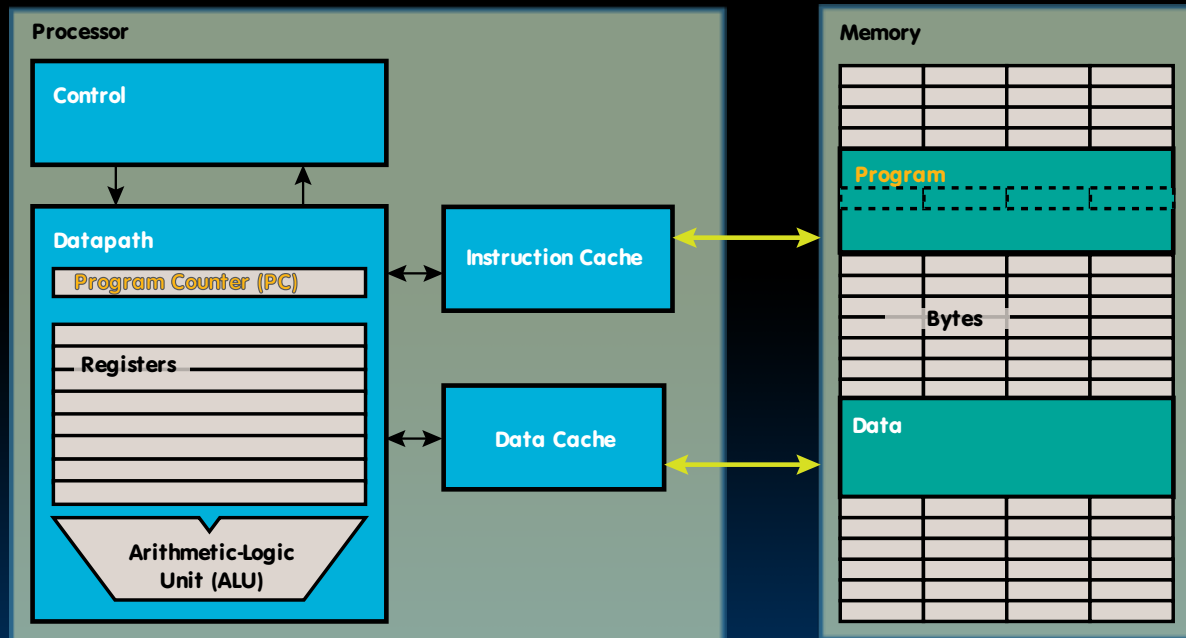
- Each instruction:
 - Can read up to two operands in decode stage
 - Can write one value in writeback stage
- Avoid structural hazard by having separate “ports”
 - Two independent read ports and one independent write port
- Three accesses per cycle can happen simultaneously

Structural Hazard: Memory Access



Instruction and Data Caches

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



Structural Hazards – Summary

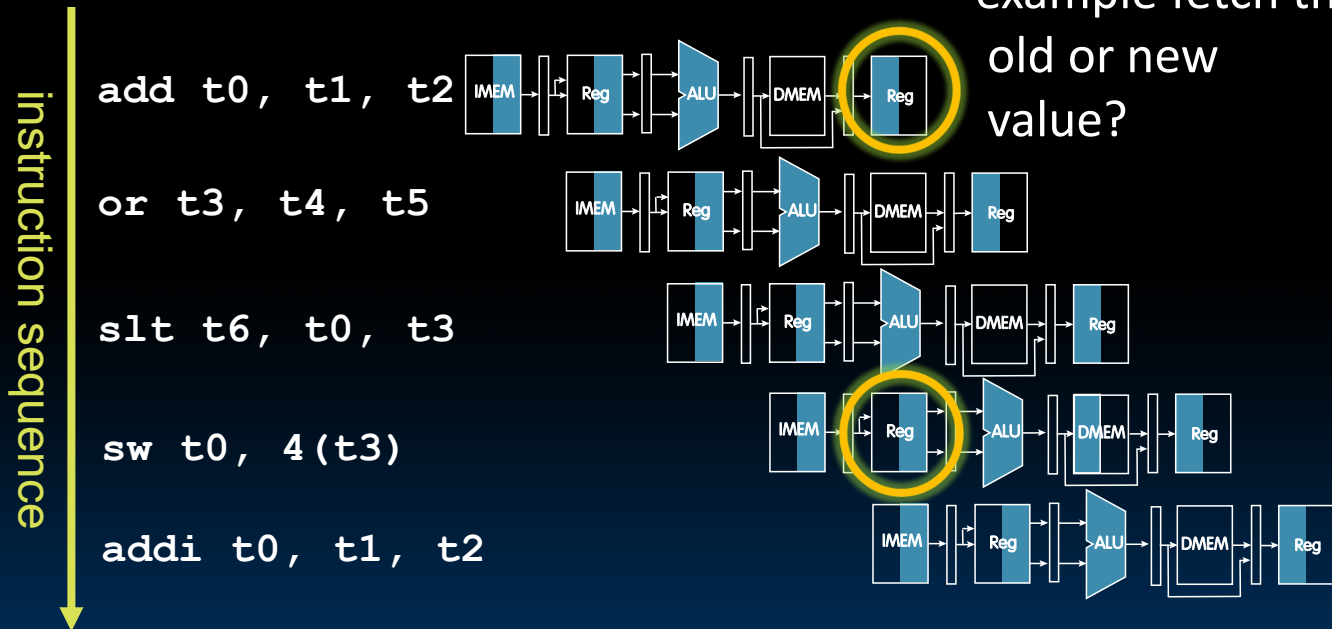
- Conflict for use of a resource
- In RISC-V pipeline with a single memory
 - Load/store requires data access
 - Without separate memories, instruction fetch would have to **stall** for that cycle
 - All other operations in pipeline would have to wait
- Pipelined datapaths require separate instruction/data 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

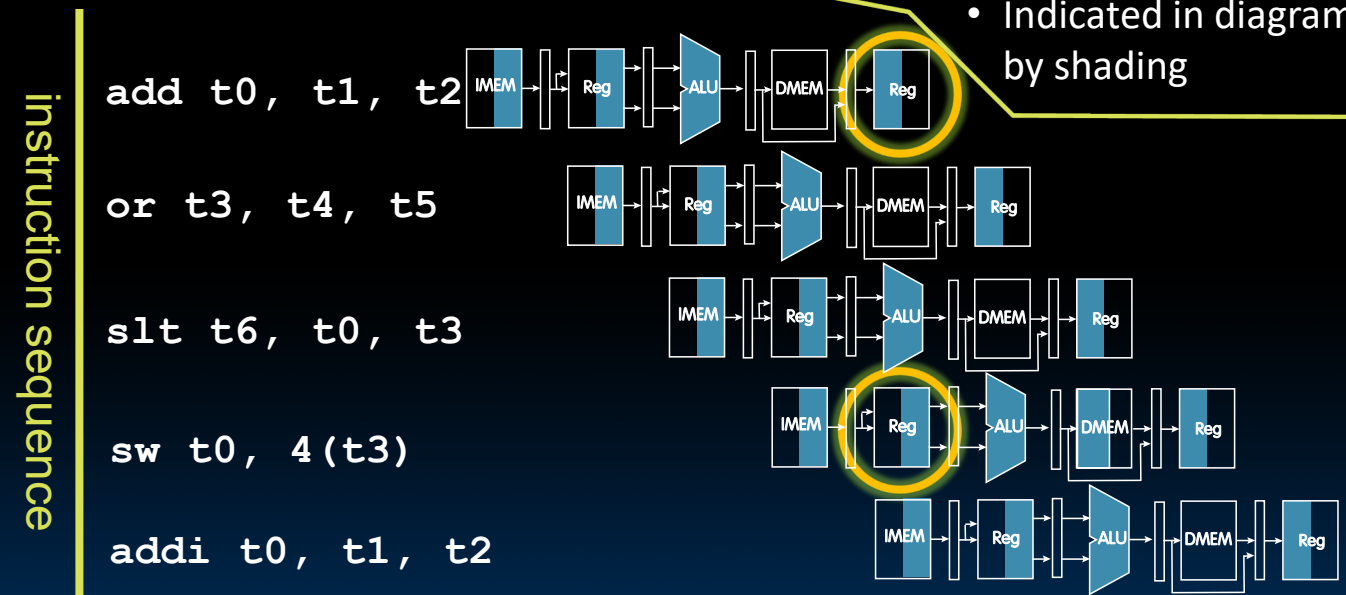
- Separate ports, but what if write to same register as read?

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



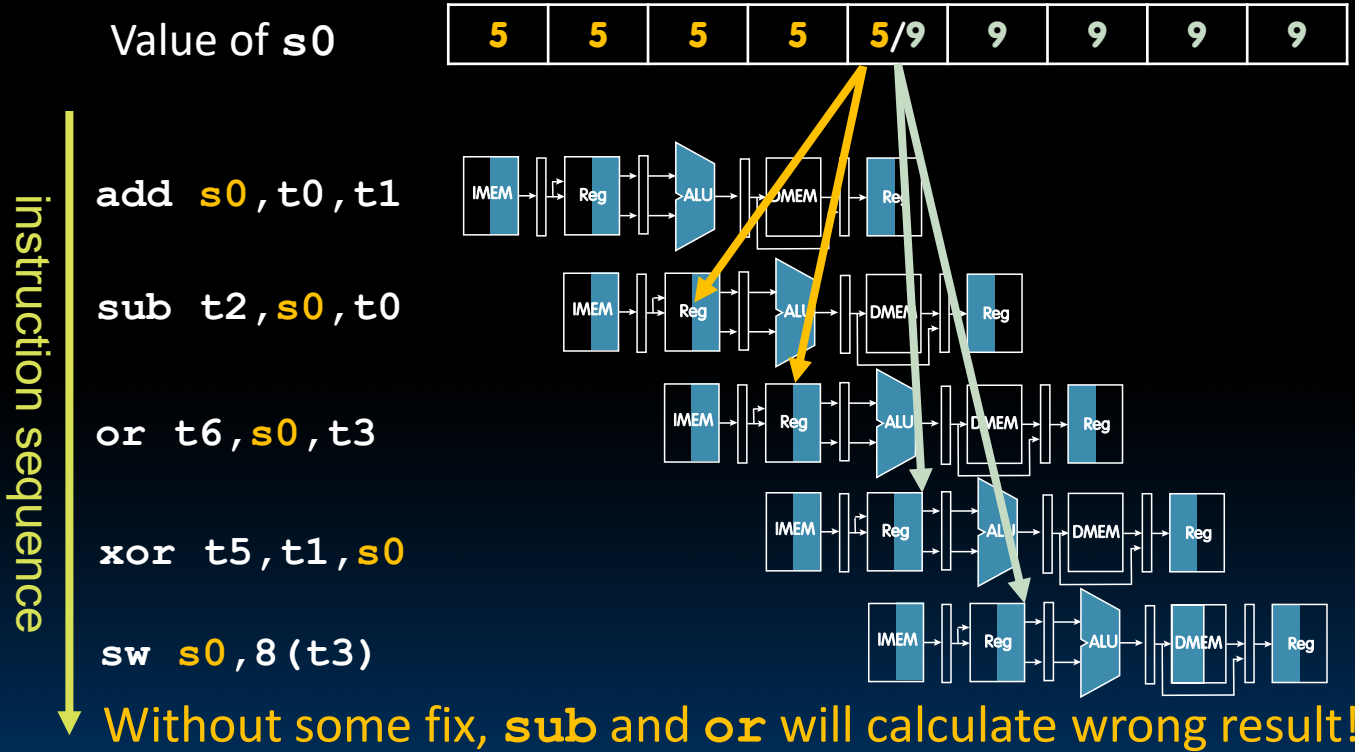
Data Hazard: Register Access

- 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.

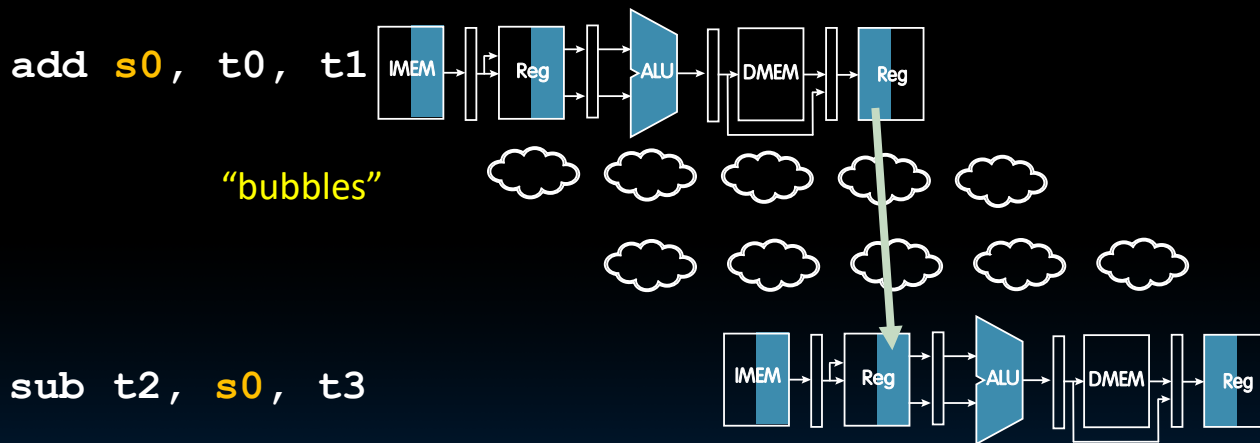
Data Hazard: ALU Result



Solution 1: Stalling

- Problem: Instruction depends on result from previous instruction

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

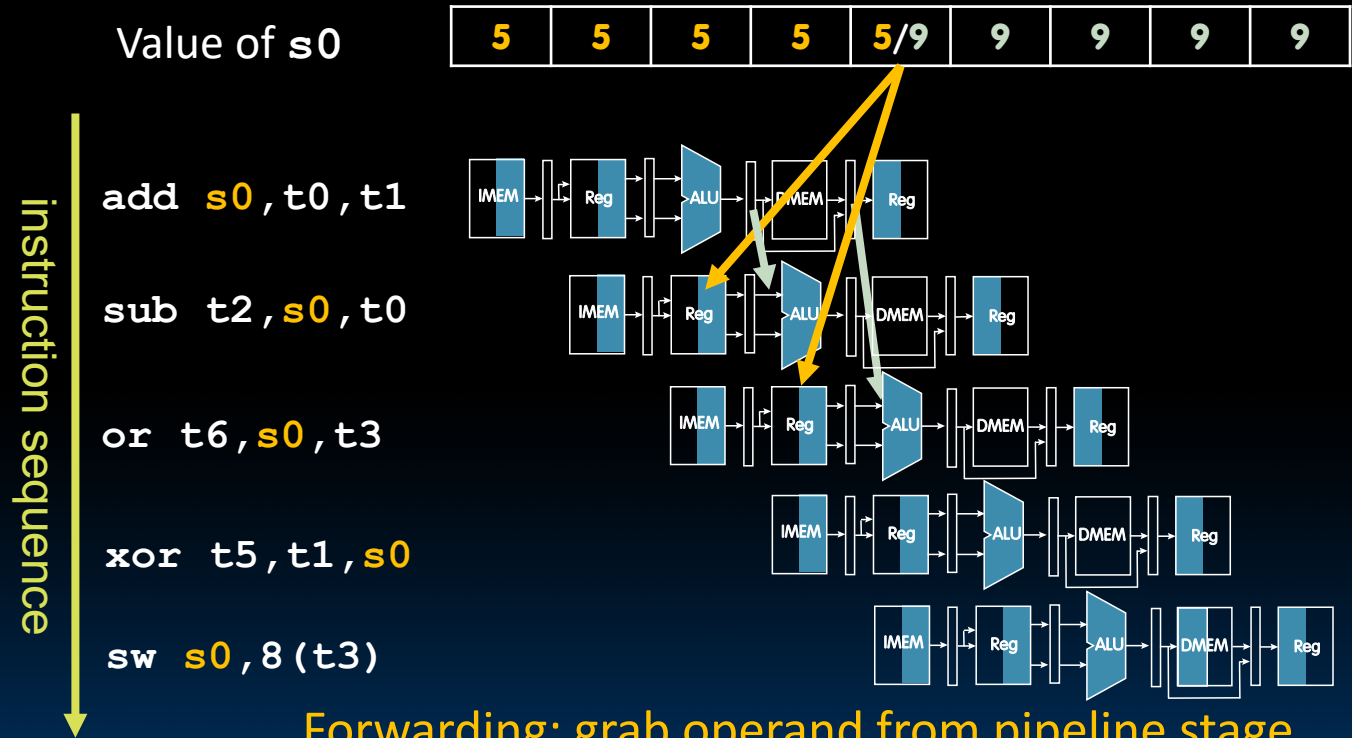


- Bubble:
 - Effectively `nop`: Affected pipeline stages do “nothing”

Stalls and Performance

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

Solution 2: Forwarding



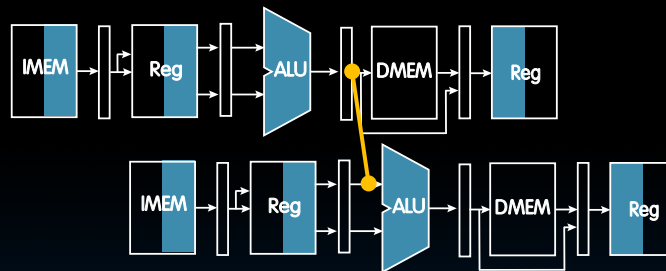
Forwarding: grab operand from pipeline stage,
rather than register file

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



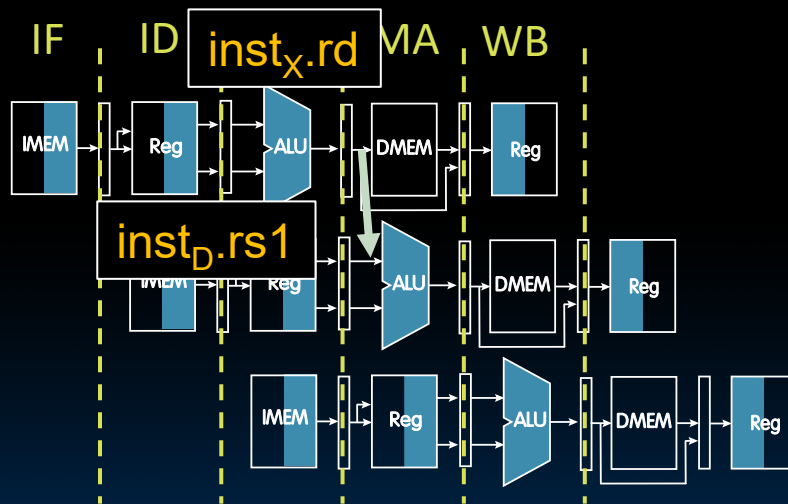
Data Needed for Forwarding (Example)

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

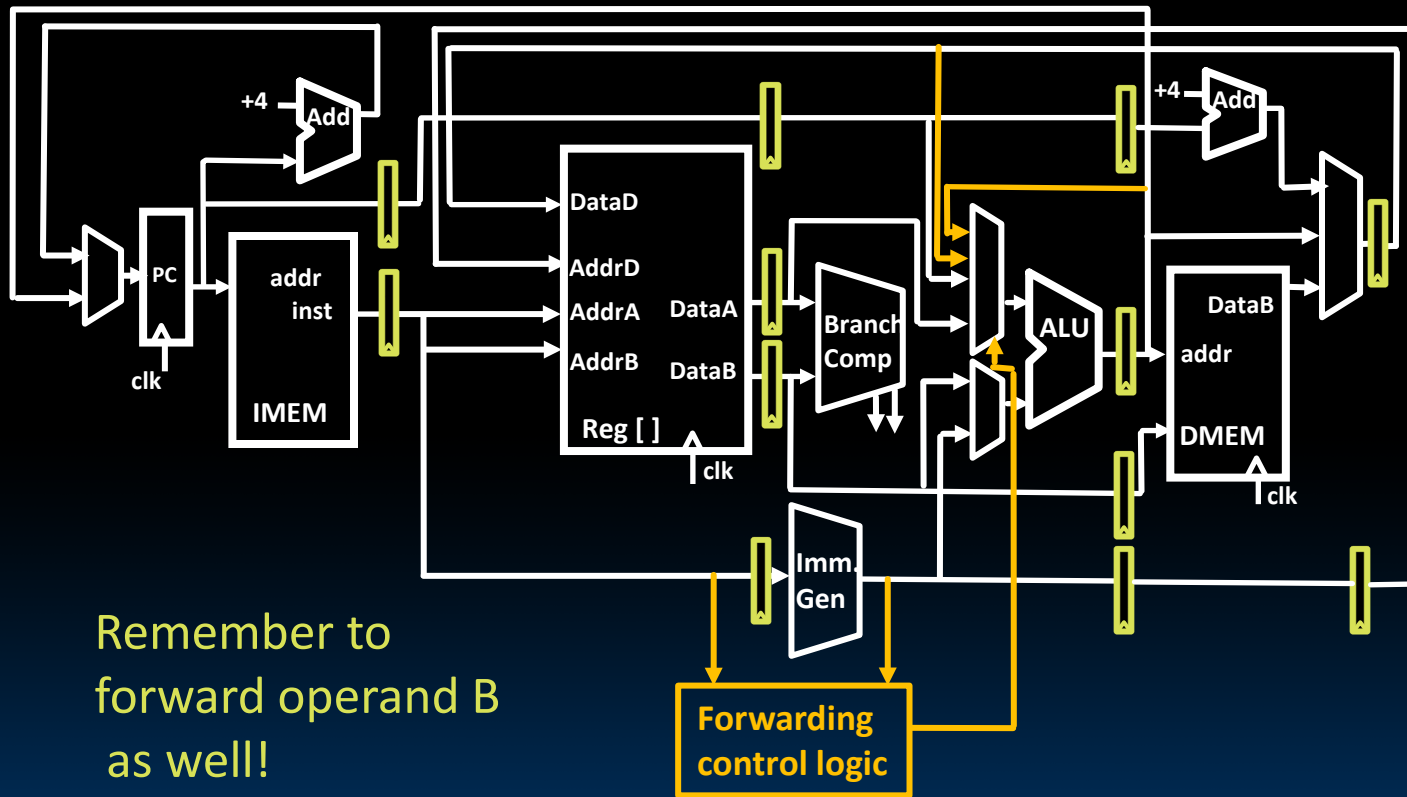
add t0, t0, t1

sub t3, t0, t5

sub t6, t0, t3

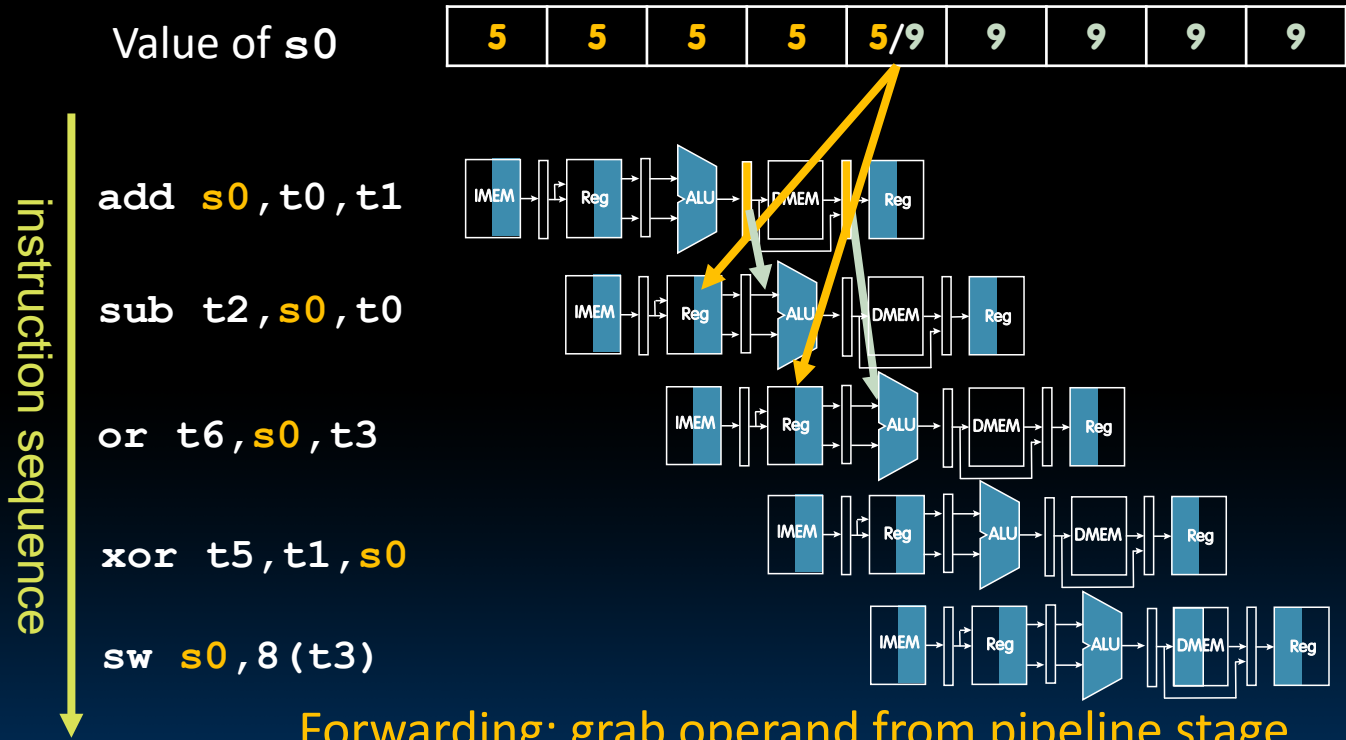


Pipelined RV32I Datapath



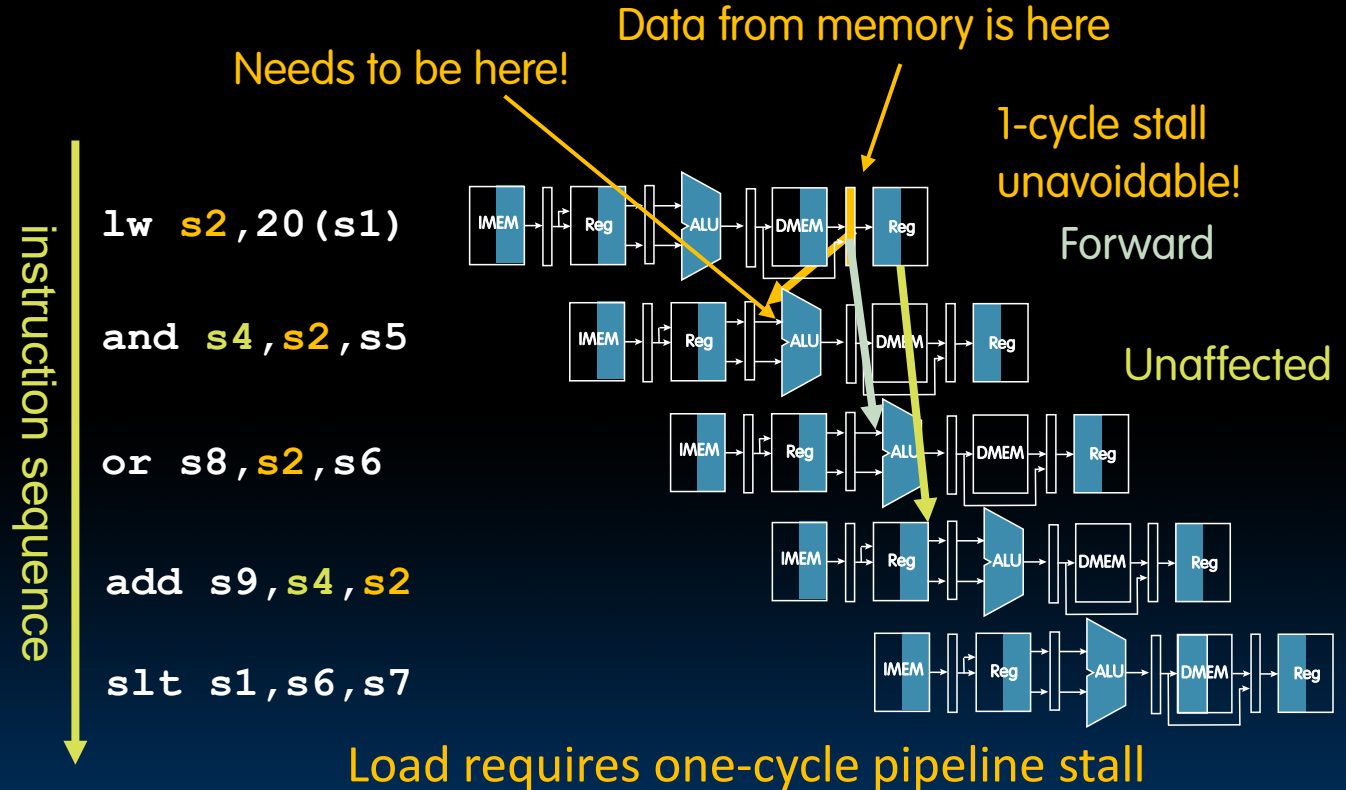
Load Data Hazard

Data Hazard and Forwarding



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

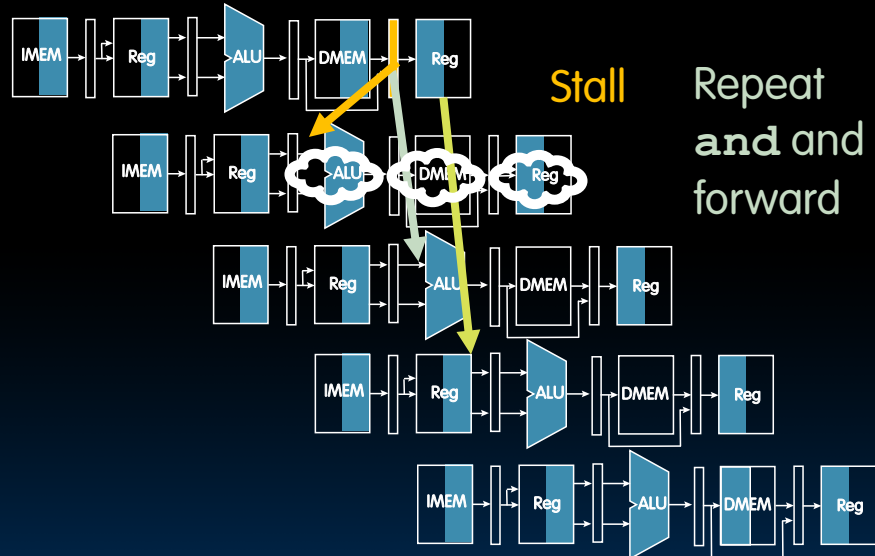
Load Data Hazard



Stall Pipeline

instruction sequence

```
lw s2, 20(s1)
and s4, s2, s5
or s8, s2, s6
add s9, s4, s2
slt s1, s6, s7
```



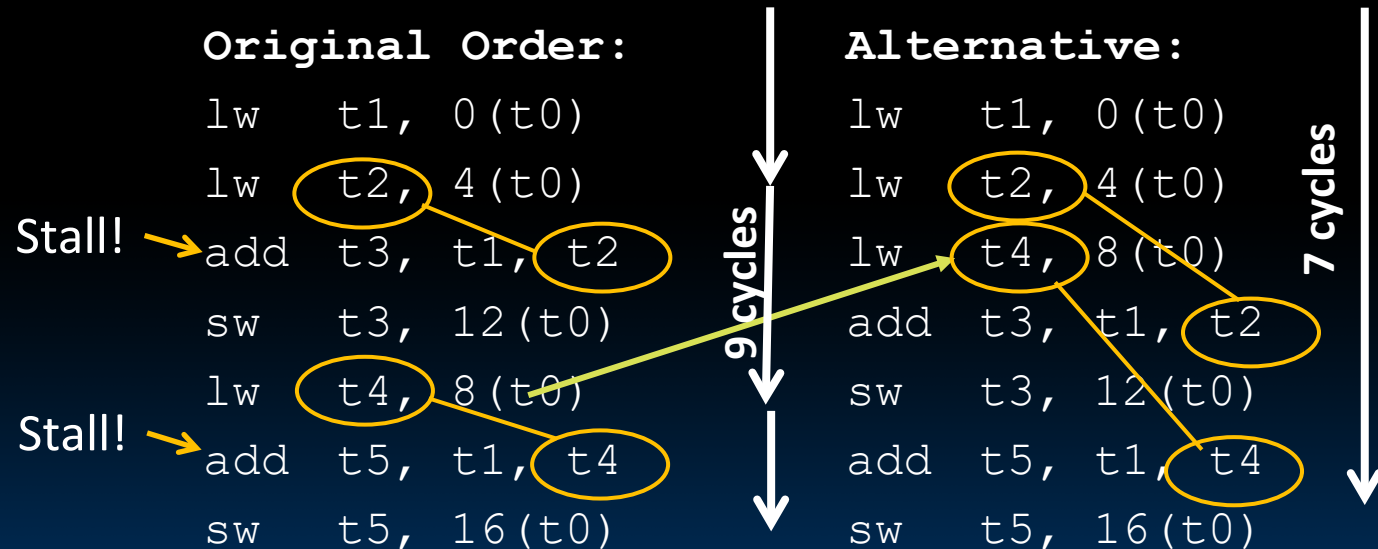
Load requires one-cycle pipeline stall

1w Data Hazard

- Slot after a load is called a *load delay slot*
 - If that instruction uses the result of the load, then the hardware will stall for one cycle
 - Equivalent to inserting an explicit `nop` in the slot
 - except the latter uses more code space
 - Performance loss
- Idea:
 - Put unrelated instruction into load delay slot
 - No performance loss!

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instr!
- RISC-V code for $A[3]=A[0]+A[1]$; $A[4]=A[0]+A[2]$



Control Hazards

Control Hazards

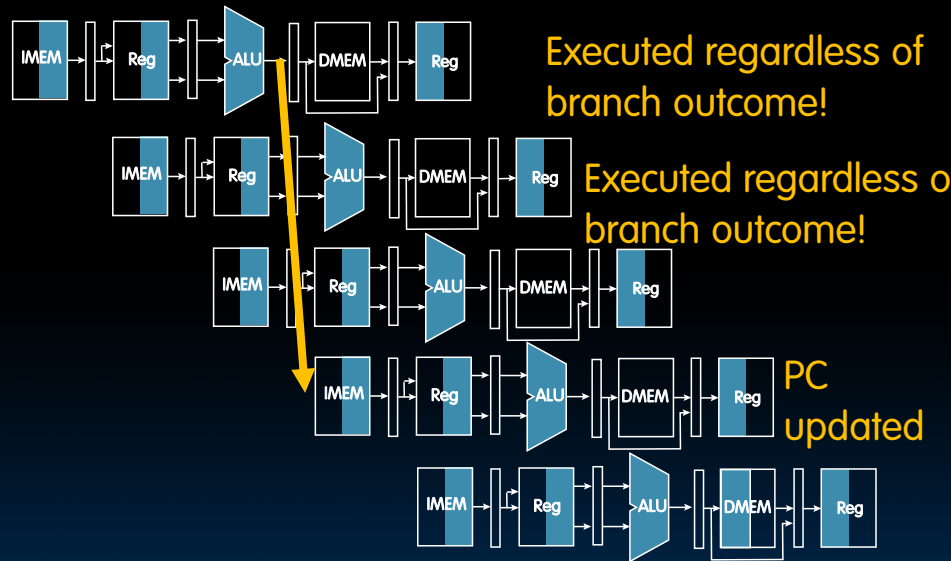
```
beq t0,t1,Label
```

```
sub t2,s0,t0
```

```
or t6,s0,t3
```

```
xor t5,t1,s0
```

```
sw s0,8(t3)
```

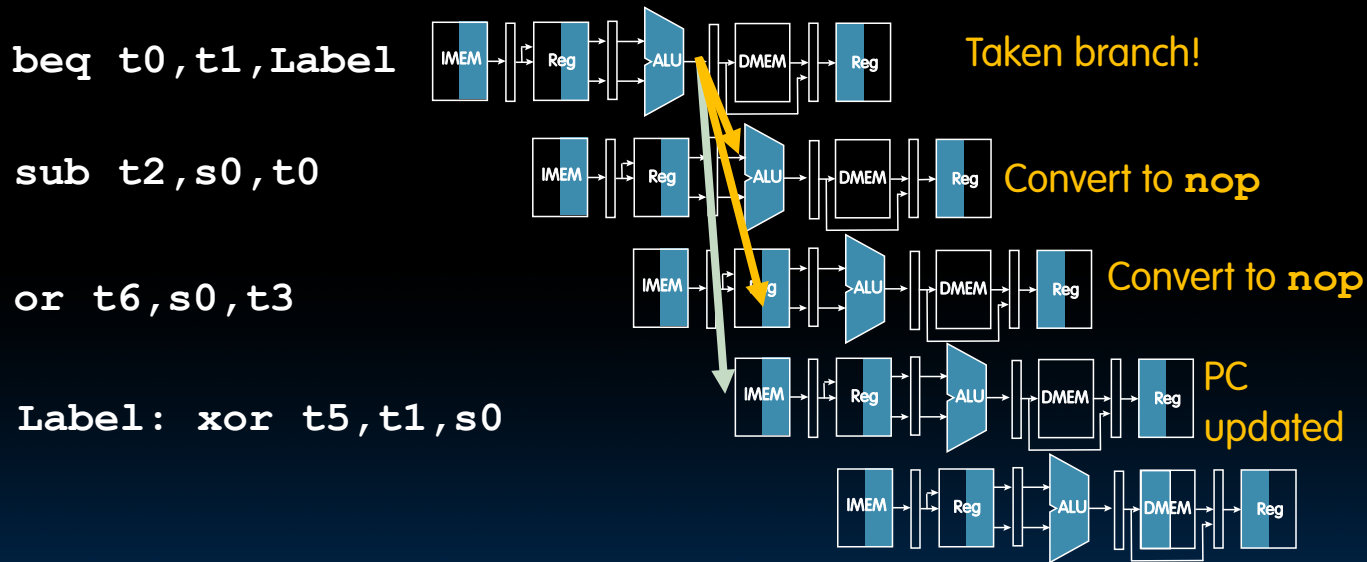


Two stall cycles after a branch!

Observation

- If branch not taken, then instructions fetched sequentially after branch are correct
- If branch or jump taken, then need to flush incorrect instructions from pipeline by converting to NOPs

Kill Instructions after Branch if Taken



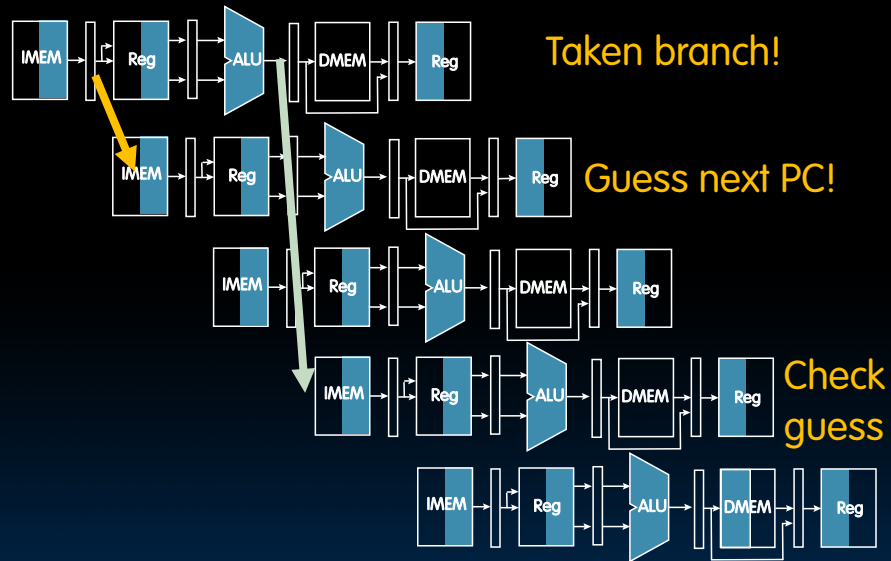
Reducing Branch Penalties

- Every taken branch in simple pipeline costs 2 dead cycles
- To improve performance, use “branch prediction” to guess which way branch will go earlier in pipeline
- Only flush pipeline if branch prediction was incorrect

Branch Prediction

```
beq t0,t1,Label
```

```
Label :...
```



Superscalar Processors

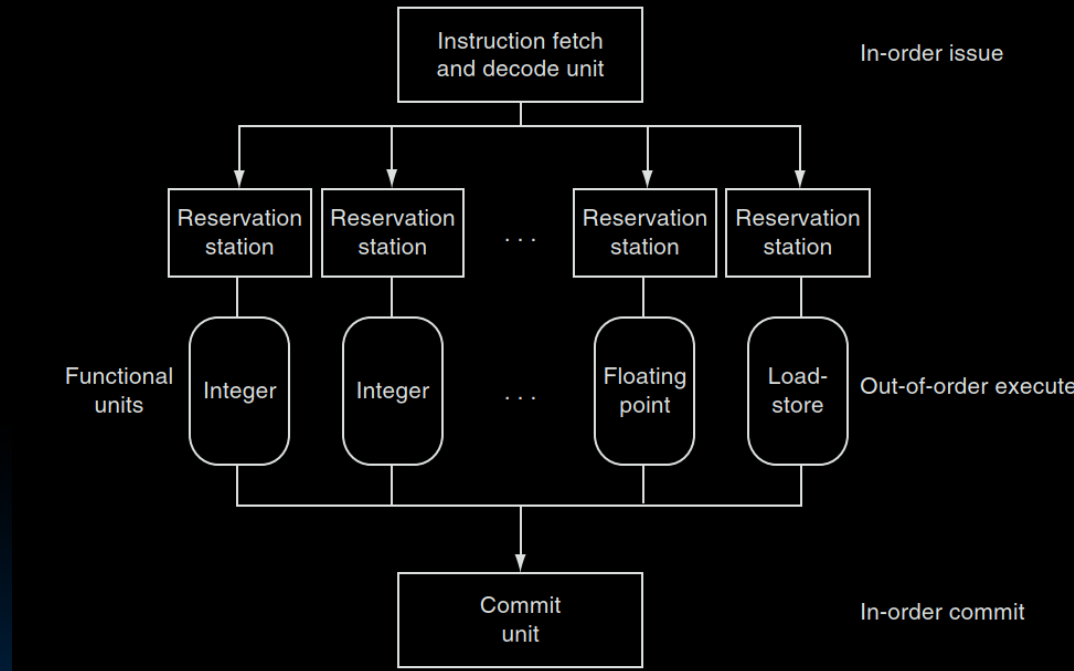
Increasing Processor Performance

1. Clock rate
 - Limited by technology and power dissipation
2. Pipelining
 - “Overlap” instruction execution
 - Deeper pipeline: 5 => 10 => 15 stages
 - Less work per stage → shorter clock cycle
 - But more potential for hazards ($CPI > 1$)
3. Multi-issue “superscalar” processor

Superscalar Processor

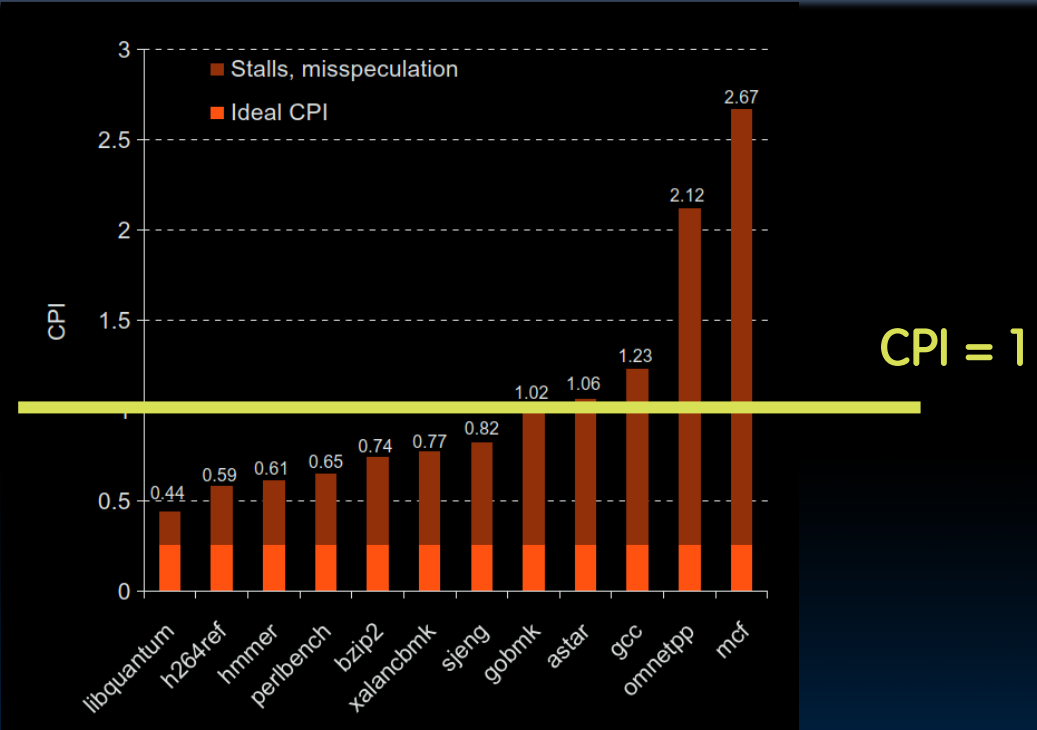
- Multiple issue “superscalar”
 - Replicate pipeline stages \Rightarrow multiple pipelines
 - Start multiple instructions per clock cycle
 - $CPI < 1$, so use Instructions Per Cycle (IPC)
 - E.g., 4GHz 4-way multiple-issue
 - 16 BIPS, peak $CPI = 0.25$, peak $IPC = 4$
 - Dependencies reduce this in practice
- “Out-of-Order” execution
 - Reorder instructions dynamically in hardware to reduce impact of hazards
- *CS152 discusses these techniques!*

Superscalar Processor



P&H, p.330

Benchmark: CPI of i7



"Iron Law" of Processor Performance

CPI = Cycles Per Instruction



Can time

Can count

Can look up

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



$$\text{CPI} = \frac{\text{Cycles}}{\text{Instruction}} = \frac{\text{Time}}{\text{Program}} \div \left(\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Time}}{\text{Cycle}} \right)$$

Pipelining and ISA Design

- RISC-V ISA designed for pipelining
 - All instructions are 32-bits
 - Easy to fetch and decode in one cycle
 - Versus x86: 1- to 15-byte instructions
 - Few and regular instruction formats
 - Decode and read registers in one step
 - Load/store addressing
 - Calculate address in 3rd stage, access memory in 4th stage
 - Alignment of memory operands
 - Memory access takes only one cycle

“And In conclusion...”

- We have built a processor!
 - Capable of executing all RISC-V instructions in one cycle each
- 5 Phases of execution
 - IF, ID, EX, MEM, WB
 - Not all instructions are active in all phases
- Controller specifies how to execute instructions
 - Implemented as ROM or logic
- Pipelining improves performance
 - But we must resolve hazards