

Computer Architecture 1

Computer Organization and Design
THE HARDWARE/SOFTWARE INTERFACE

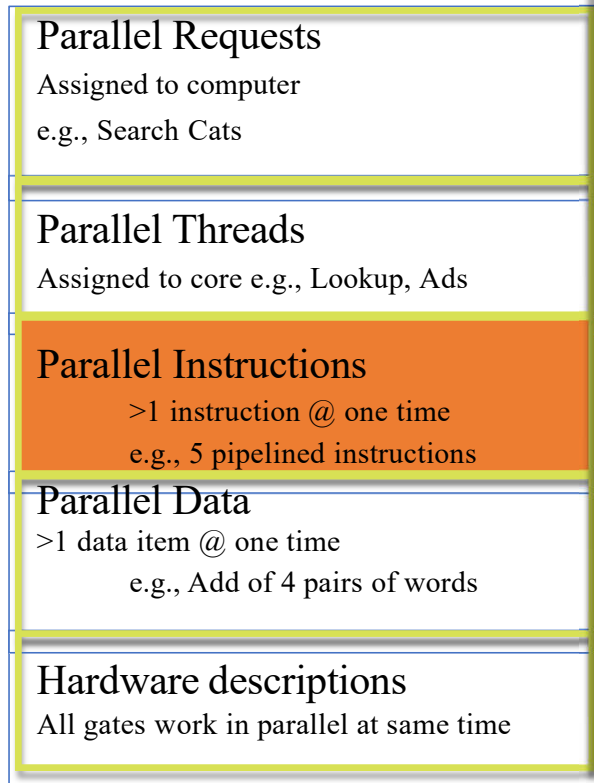
[Adapted from Computer Organization and Design, RISC-V Edition, Patterson & Hennessy, © 2018, MK]

[Adapted from Great ideas in Computer Architecture (CS 61C) lecture slides, Garcia and Nikolić, © 2020, UC Berkeley]

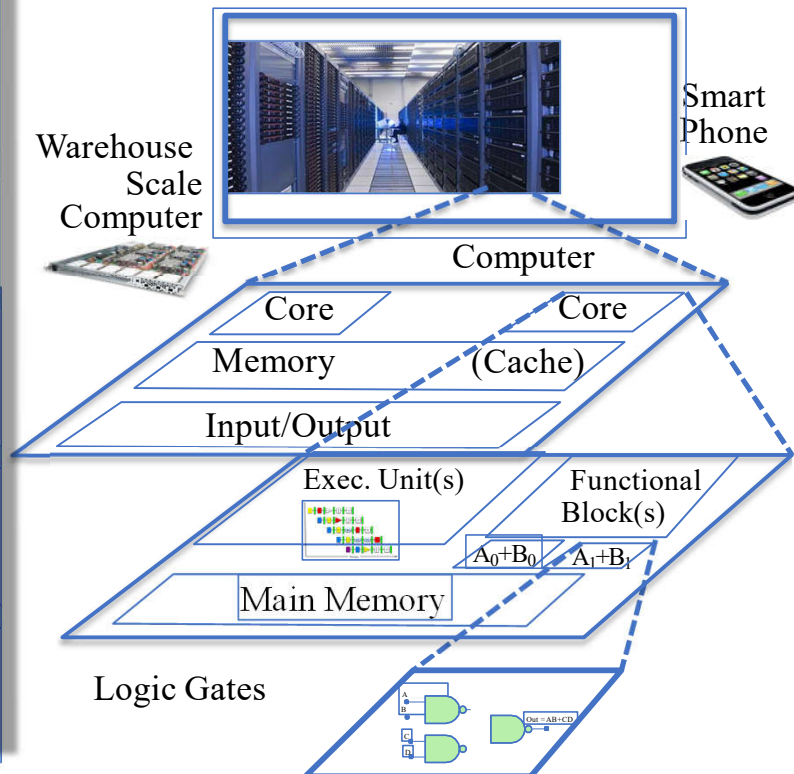
New-School Machine Structures

Harness
Parallelism &
achieve high
performance

Software



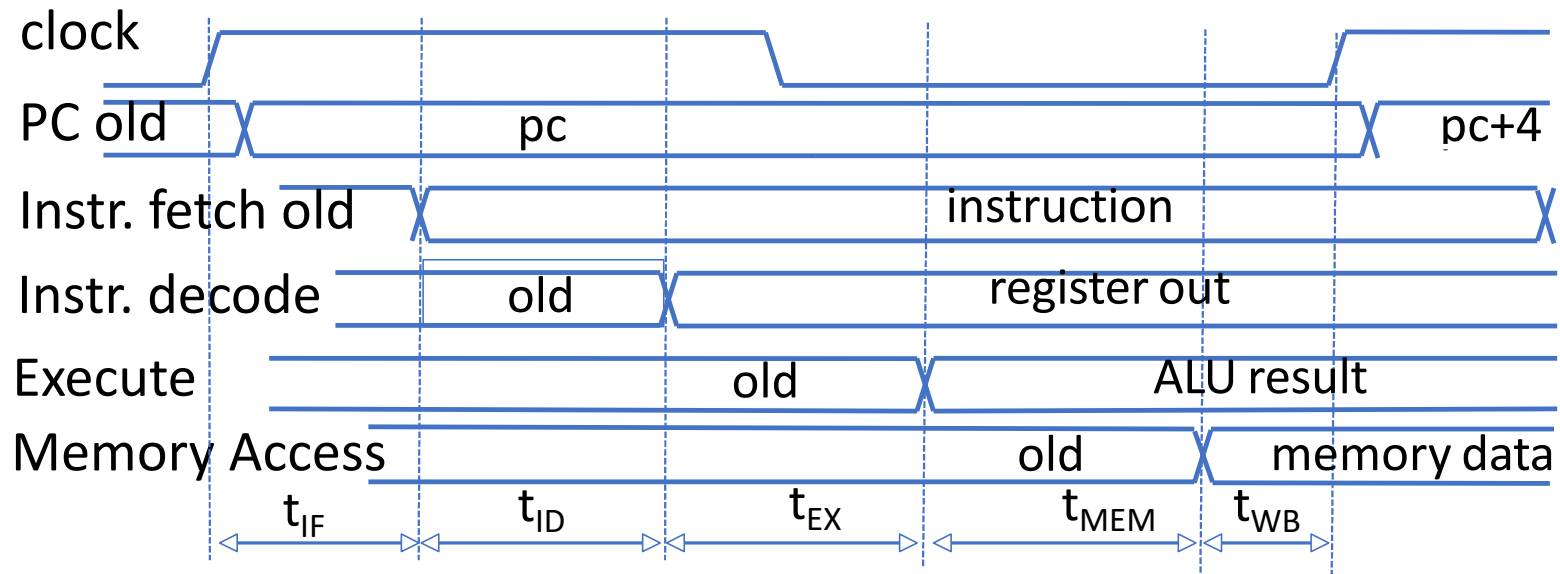
Hardware



6 Great Ideas in Computer Architecture

1. Abstraction (Layers of Representation/Interpretation)
2. Moore's Law
3. Principle of Locality/Memory Hierarchy
4. **Parallelism**
5. Performance Measurement & Improvement
6. Dependability via Redundancy

Instruction Timing



IF	ID	EX	MEM	WB	Total
I-MEM	Reg Read	ALU	D-MEM	Reg W	
200 ps	100 ps	200 ps	200 ps	100 ps	800 ps

Instruction Timing

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

- Maximum clock frequency

- $f_{\max} = 1/800\text{ps} = 1.25 \text{ GHz}$

Performance Measures

- “Our” Single-cycle RISC-V CPU executes instructions at 1.25 GHz
 - 1 instruction every 800 ps
- Can we improve its performance?
 - What do we mean with this statement?
 - Not so obvious:
 - Quicker response time, so one job finishes faster?
 - More jobs per unit time (e.g. web server returning pages, spoken words recognized)?
 - Longer battery life?

Transportation Analogy



	Sports Car	Bus
Passenger Capacity	2	50
Travel Speed	200 mph	50 mph
Gas Mileage	5 mpg	2 mpg

50 Mile trip (assume they return instantaneously)

	Sports Car	Bus
Travel Time	15 min	60 min
Time for 100 passengers	750 min (50 2-person trips)	120 min (two 50-person trips)
Gallons per passenger	5 gallons	0.5 gallons

Computer Analogy

Transportation	Computer
Trip Time	Program execution time: e.g. time to update display
Time for 100 passengers	Throughput: e.g. number of server requests handled per hour
Gallons per passenger	Energy per task*: e.g. how many movies you can watch per battery charge or energy bill for datacenter

* Note:

Power is not a good measure, since low-power CPU might run for a long time to complete one task consuming more energy than faster computer running at higher power for a shorter time

Processor Performance Iron Law

“Iron Law” of Processor Performance

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



CPI = Cycles Per Instruction

Defining (Speed) Performance

- Normally interested in reducing
 - **Response time** (aka execution time) – the time between the start and the completion of a task
 - Important to individual users
 - Thus, to maximize performance, need to **minimize** execution time

$$\text{performance}_X = 1 / \text{execution_time}_X$$

If X is n times faster than Y, then

$$\frac{\text{performance}_X}{\text{performance}_Y} = \frac{\text{execution_time}_Y}{\text{execution_time}_X} = n$$

- Throughput – the total amount of work done in a given time
 - Important to data center managers
- Decreasing response time almost always improves throughput

Performance Factors

- Want to distinguish elapsed time and the time spent on our task
- CPU execution time (CPU time) – time the CPU spends working on a task
 - Does not include time waiting for I/O or running other programs

$$\text{CPU execution time for a program} = \frac{\# \text{ CPU clock cycles for a program}}{\text{clock cycle time}}$$

or

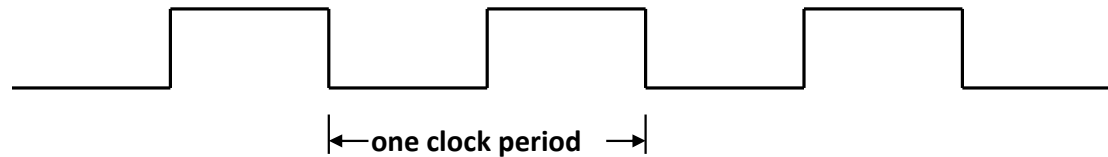
$$\text{CPU execution time for a program} = \frac{\# \text{ CPU clock cycles for a program}}{\text{clock rate}}$$

- Can improve performance by reducing either the **length of the clock cycle** or the **number of clock cycles required for a program**

Review: Machine Clock Rate

- Clock rate (MHz, GHz) is inverse of clock cycle time (clock period)

$$CC = 1 / CR$$



10 nsec clock cycle => 100 MHz clock rate

5 nsec clock cycle => 200 MHz clock rate

2 nsec clock cycle => 500 MHz clock rate

1 nsec clock cycle => 1 GHz clock rate

500 psec clock cycle => 2 GHz clock rate

250 psec clock cycle => 4 GHz clock rate

200 psec clock cycle => 5 GHz clock rate

Clock Cycles per Instruction

- Not all instructions take the same amount of time to execute
 - One way to think about execution time is that it equals the number of instructions executed multiplied by the average time per instruction

$$\begin{array}{l} \# \text{ CPU clock cycles} \\ \text{for a program} \end{array} = \begin{array}{l} \# \text{ Instructions} \\ \text{for a program} \end{array} \times \begin{array}{l} \text{Average clock cycles} \\ \text{per instruction} \end{array}$$

- Clock cycles per instruction (CPI) – the average number of clock cycles each instruction takes to execute
 - A way to compare two different implementations of the same ISA

	CPI for this instruction class		
	A	B	C
CPI	1	2	3

Effective CPI

- Computing the overall effective CPI is done by looking at the different types of instructions and their individual cycle counts and averaging

$$\text{Overall effective CPI} = \sum_{i=1}^n (\text{CPI}_i \times \text{IC}_i)$$

- Where IC_i is the count (percentage) of the number of instructions of class i executed
- CPI_i is the (average) number of clock cycles per instruction for that instruction class
- n is the number of instruction classes
- The overall effective CPI varies by instruction mix – a measure of the dynamic frequency of instructions across one or many programs

THE Performance Equation

- Our basic performance equation is then

$$\text{CPU time} = \text{Instruction_count} \times \text{CPI} \times \text{clock_cycle}$$

or

$$\text{CPU time} = \frac{\text{Instruction_count} \times \text{CPI}}{\text{clock_rate}}$$

- These equations separate the **three key** factors that affect performance
 - Can measure the CPU execution time by running the program
 - The clock rate is usually given
 - Can measure overall instruction count by using profilers/ simulators without knowing all of the implementation details
 - CPI varies by instruction type and ISA implementation for which we must know the implementation details

A Simple Example

Op	Freq	CPI _i	Freq x CPI _i			
ALU	50%	1	.5	.5	.5	.25
Load	20%	5	1.0	.4	1.0	1.0
Store	10%	3	.3	.3	.3	.3
Branch	20%	2	.4	.4	.2	.4
			$\Sigma =$ 2.2	1.6	2.0	1.95

- How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?
CPU time new = 1.6 x IC x CC so 2.2/1.6 means 37.5% faster
- How does this compare with using branch prediction to shave a cycle off the branch time?
CPU time new = 2.0 x IC x CC so 2.2/2.0 means 10% faster
- What if two ALU instructions could be executed at once?
CPU time new = 1.95 x IC x CC so 2.2/1.95 means 12.8% faster

Instructions per Program

Determined by

- Task
- Algorithm, e.g. $O(N^2)$ vs $O(N)$
- Programming language
- Compiler
- Instruction Set Architecture (ISA)

(Average) Clock Cycles per Instruction (CPI)

Determined by

- ISA
- Processor implementation (or *microarchitecture*)
- E.g. for “our” single-cycle RISC-V design, $\text{CPI} = 1$
- Complex instructions (e.g. **strcpy**), $\text{CPI} \gg 1$
- Superscalar processors, $\text{CPI} < 1$ (next lectures)

Time per Cycle ($1/\text{Frequency}$)

Determined by

- Processor microarchitecture (determines critical path through logic gates)
- Technology (e.g. 5nm versus 28nm)
- Power budget (lower voltages reduce transistor speed)

Speed Tradeoff Example

- For some task (e.g. image compression) ...

	Processor A	Processor B
# Instructions	1 Million	1.5 Million
Average CPI	2.5	1
Clock rate f	2.5 GHz	2 GHz
Execution time	1 ms	0.75 ms

Processor B is faster for this task, despite executing more instructions and having a slower clock rate!

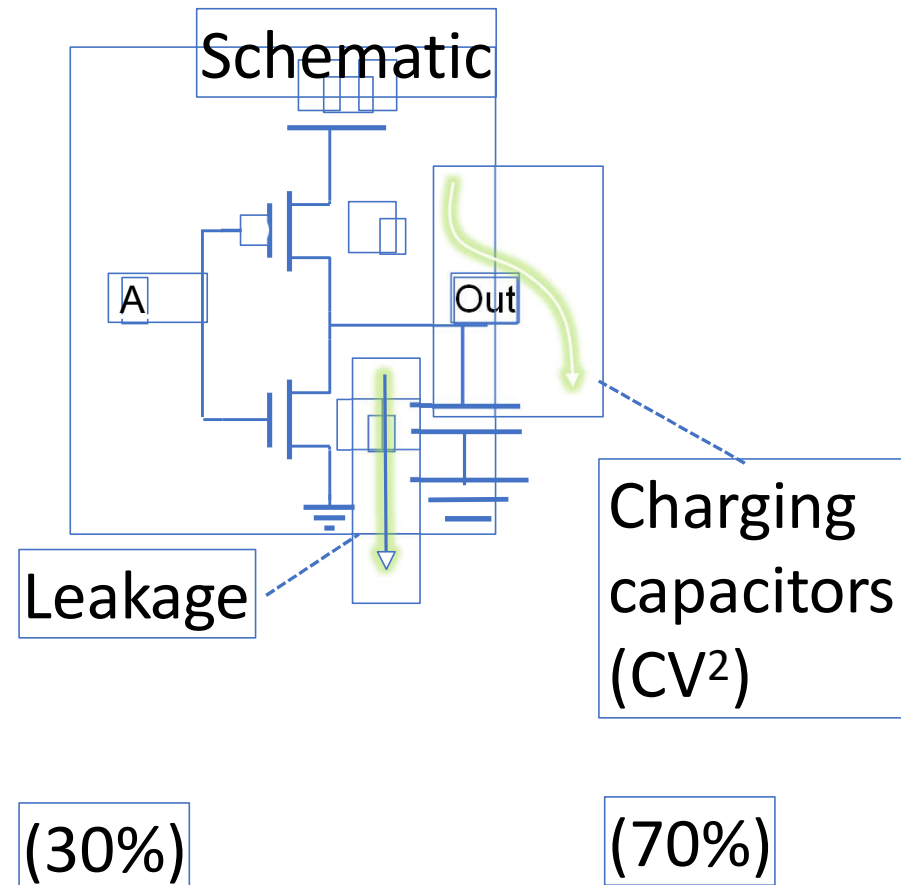
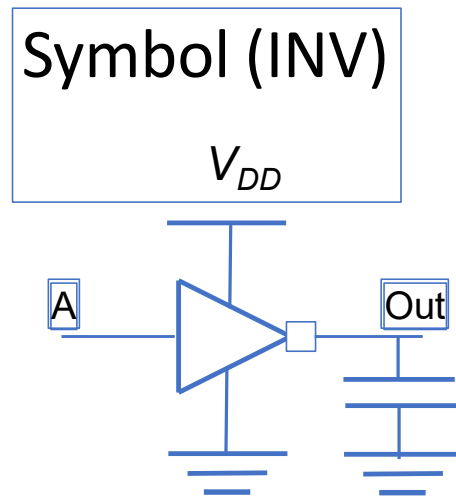
Determinates of CPU Performance

$$\text{CPU time} = \text{Instruction_count} \times \text{CPI} \times \text{clock_cycle}$$

	Instruction_ count	CPI	clock_cycle
Algorithm	x	x	
Programming language	x	x	
Compiler	x	x	
ISA	x	x	x
Processor organization		x	x
Technology			x

Energy Efficiency

Where Does Energy Go in CMOS?



Energy per Task

$$\text{Energy}_{\text{Program}} = \text{Instructions}_{\text{Program}} * \text{Energy}_{\text{Instruction}}$$

$$\text{Energy}_{\text{Program}} \propto \text{Instructions}_{\text{Program}} * C V^2$$

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

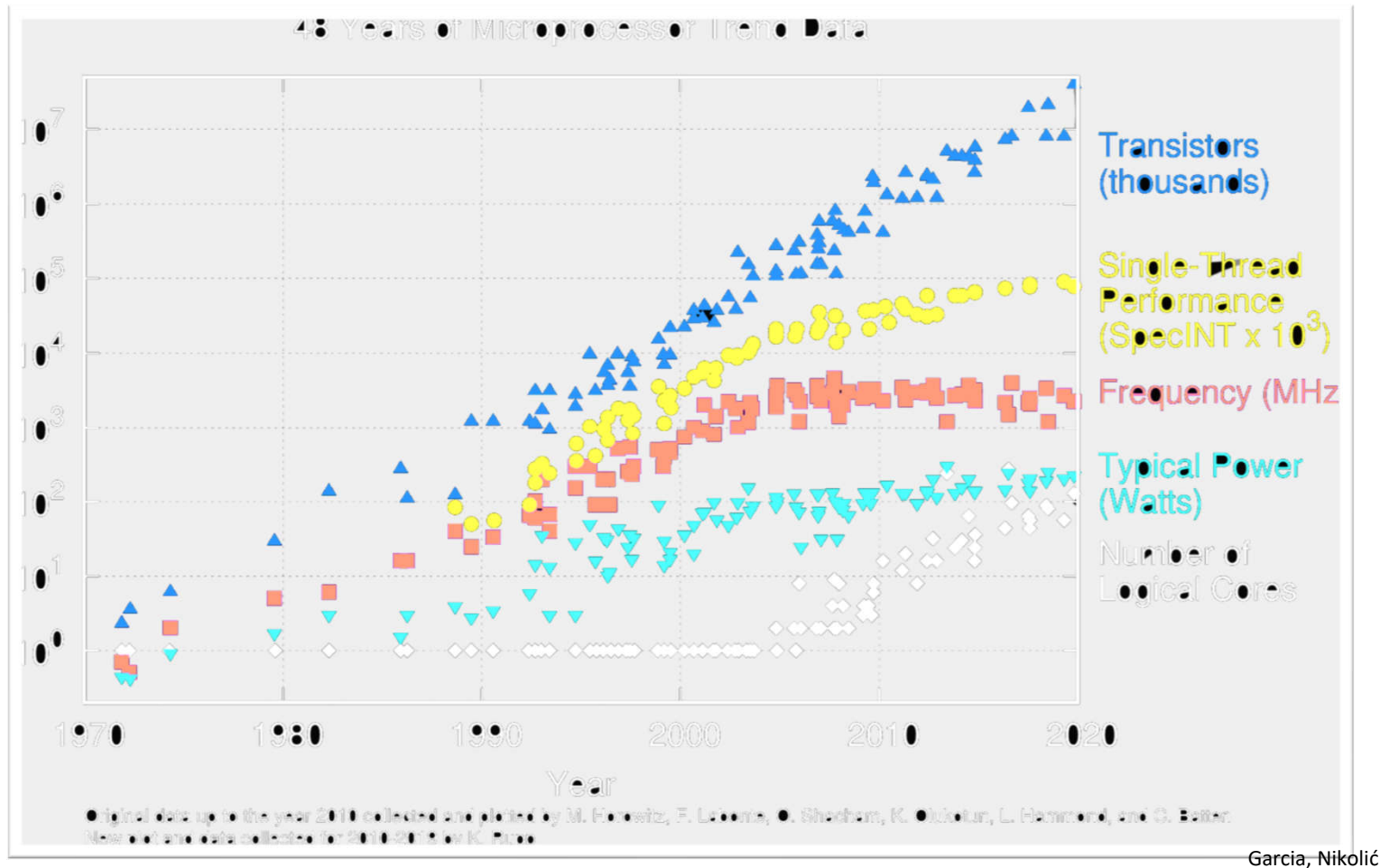
Supply voltage,
e.g. 1V

Want to reduce capacitance and voltage to reduce energy/task

Energy Tradeoff Example

- “Next-generation” processor
 - C (Moore’s Law): -15 %
 - Supply voltage, V_{sup} : -15 %
 - Energy consumption: $0 - (1 - 0.85^3) = -39 \%$
- Significantly improved energy efficiency thanks to
 - Moore’s Law AND
 - Reduced supply voltage

Performance/Power Trends



RISC-V

End of Scaling

- In recent years, industry has not been able to reduce supply voltage much, as reducing it further would mean increasing “leakage power” where transistor switches don’t fully turn off (more like dimmer switch than on-off switch)
- Also, size of transistors and hence capacitance, not shrinking as much as before between transistor generations
 - Need to go to 3D
- Power becomes a growing concern – the “power wall”

Energy “Iron Law”

$$\text{Performance} = \frac{\text{Power}}{\text{Energy Efficiency}}$$

(Tasks/Second) (Joules/Second) (Tasks/Joule)

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

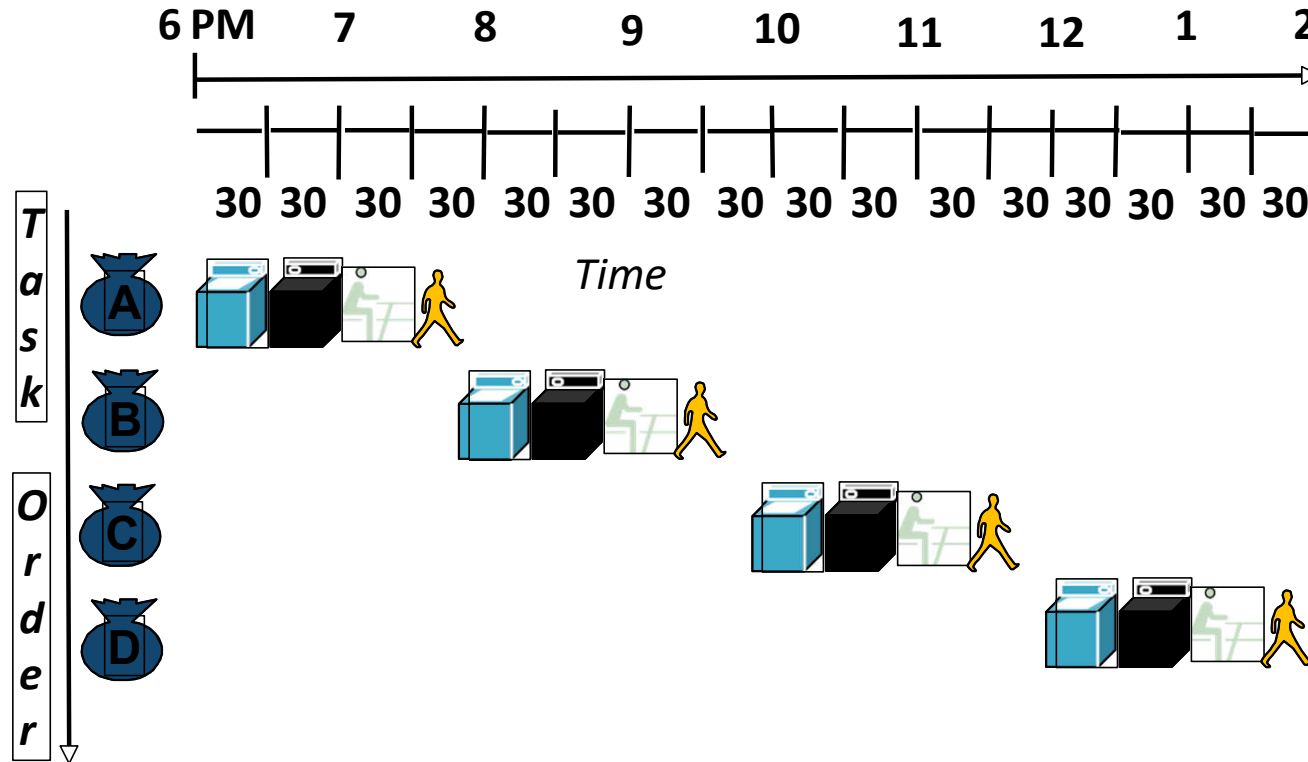
Introduction to Pipelining

Gotta Do Laundry

- Avi, Bora, Caroline, Dan each have 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

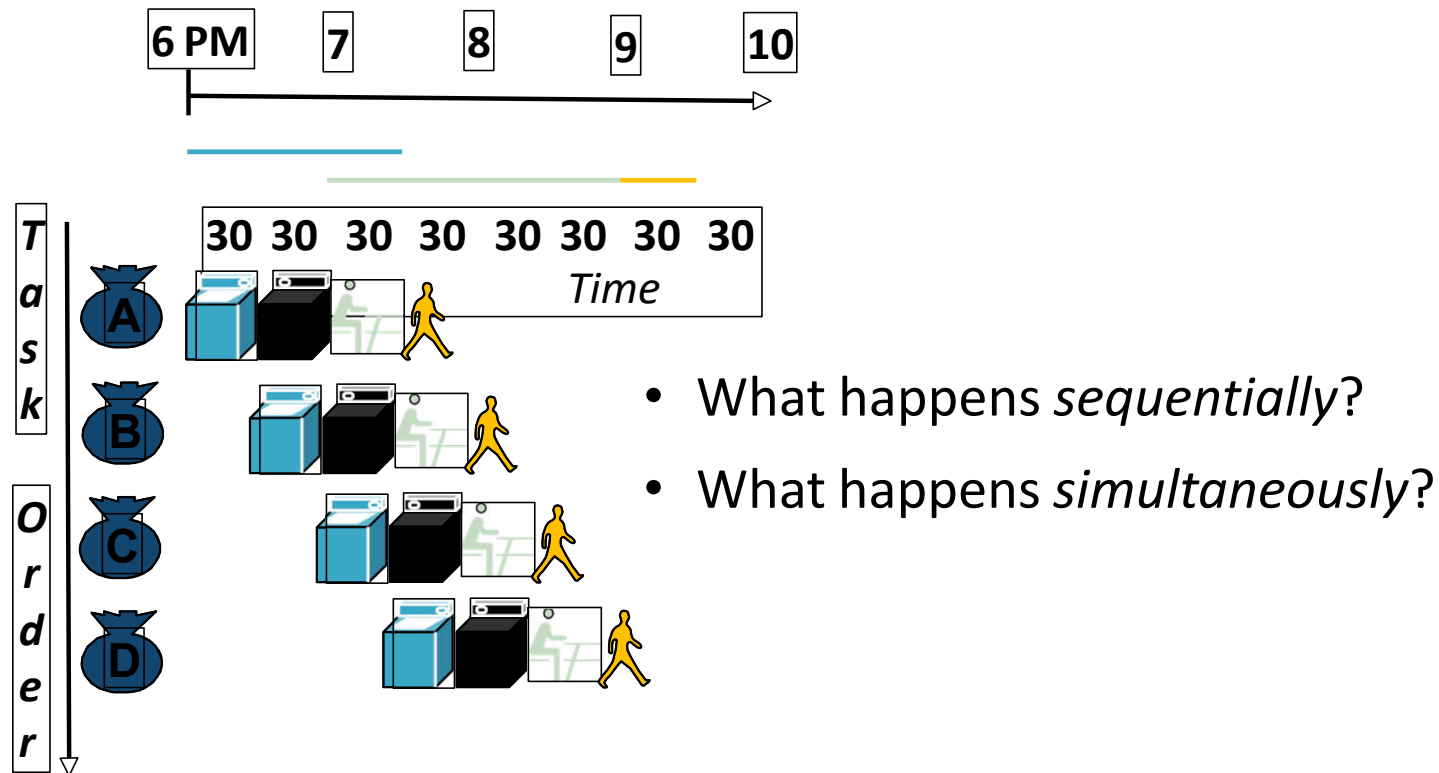


Sequential Laundry



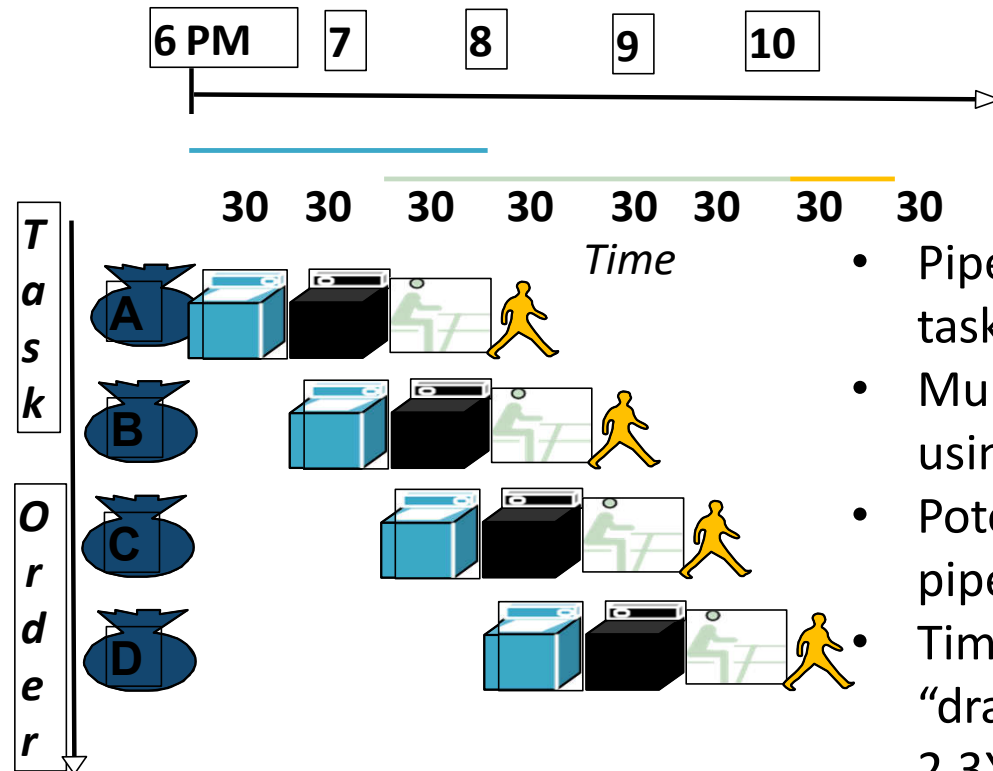
Sequential laundry takes 8 hours for 4 loads!

Pipelined Laundry



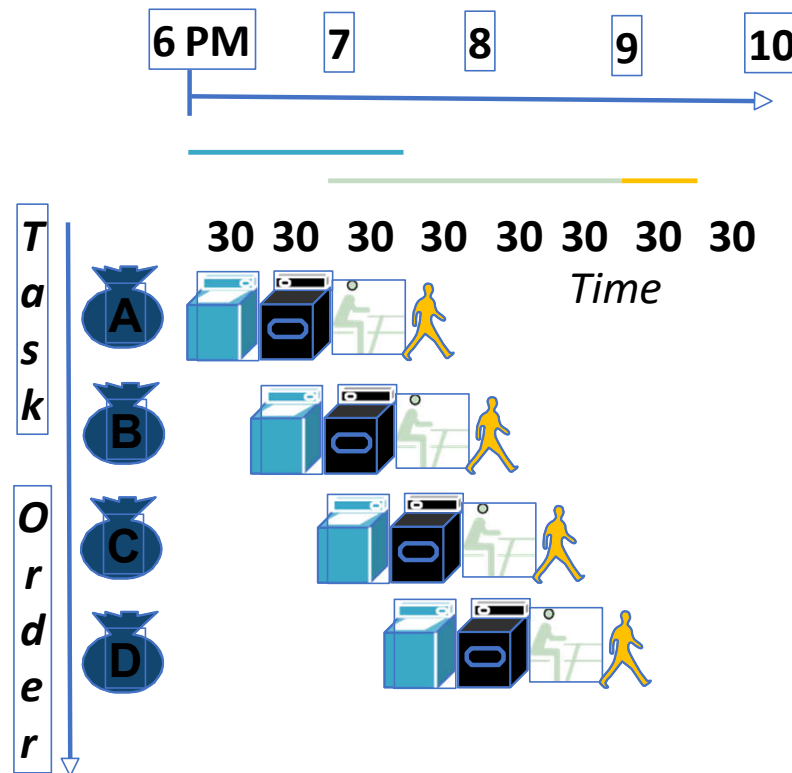
Pipelined laundry takes 3.5 hours for 4 loads!

Sequential Laundry



- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number of pipe stages
- Time to "fill" pipeline and time to "drain" it reduces speedup: 2.3X v. 4X in this example

Sequential Laundry



Suppose:

- new Washer takes 20 minutes
- new Stasher takes 20 minutes.

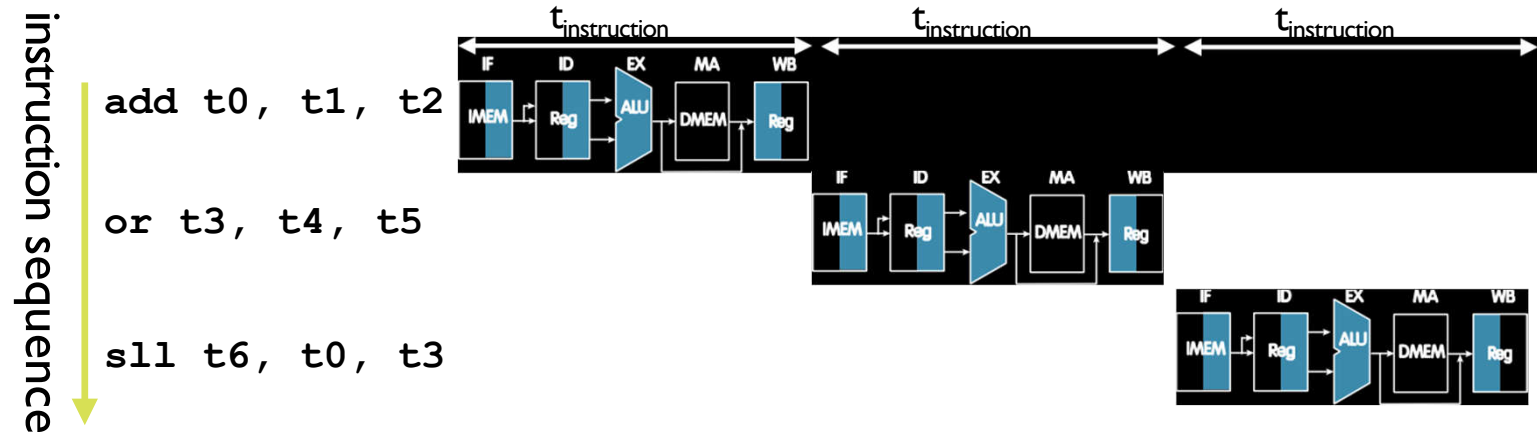
How much faster is pipeline?

Pipeline rate limited by slowest pipeline stage







Unbalanced lengths of pipe stages reduce speedup

'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
$t_{\text{instruction}}$		800 ps



Pipelined RISC-V Datapath

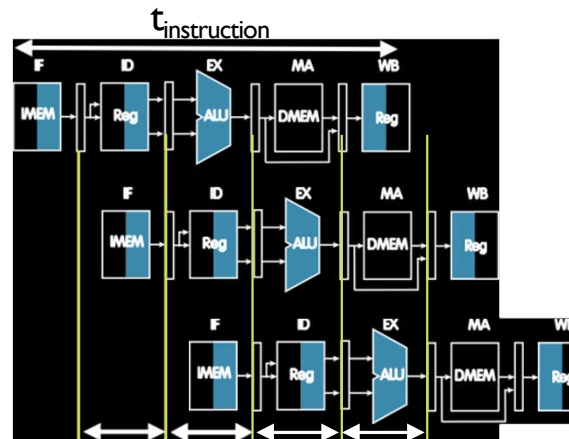
Phase	Pictogram	t_{step} Serial	t_{cycle} Pipelined
Instruction Fetch		200 ps	200 ps
Reg Read		100 ps	200 ps
ALU		200 ps	200 ps
Memory		200 ps	200 ps
Register Write		100 ps	200 ps
$t_{\text{instruction}}$		800 ps	1000 ps

instruction sequence ↓

add t0, t1, t2

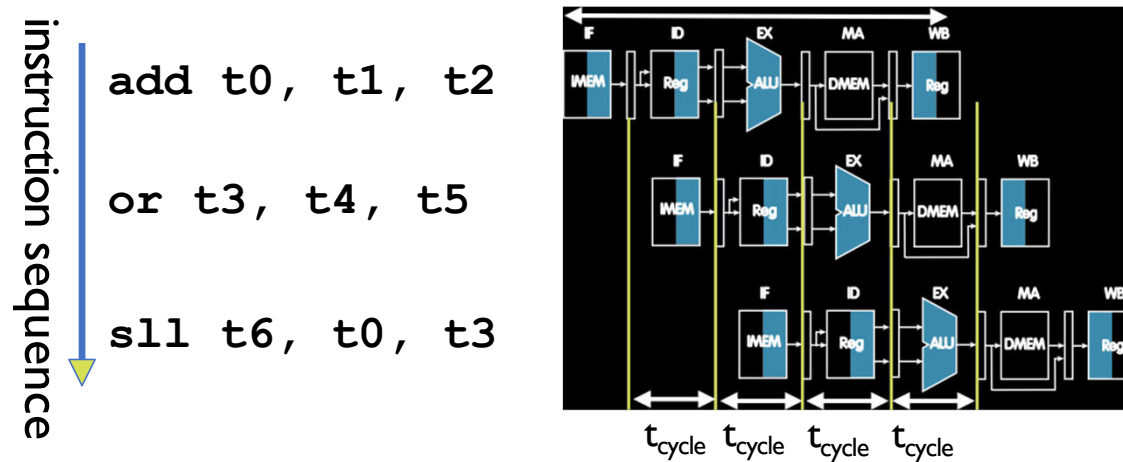
or t3, t4, t5

sll t6, t0, t3



RISC-V

Pipelined RISC-V Datapath

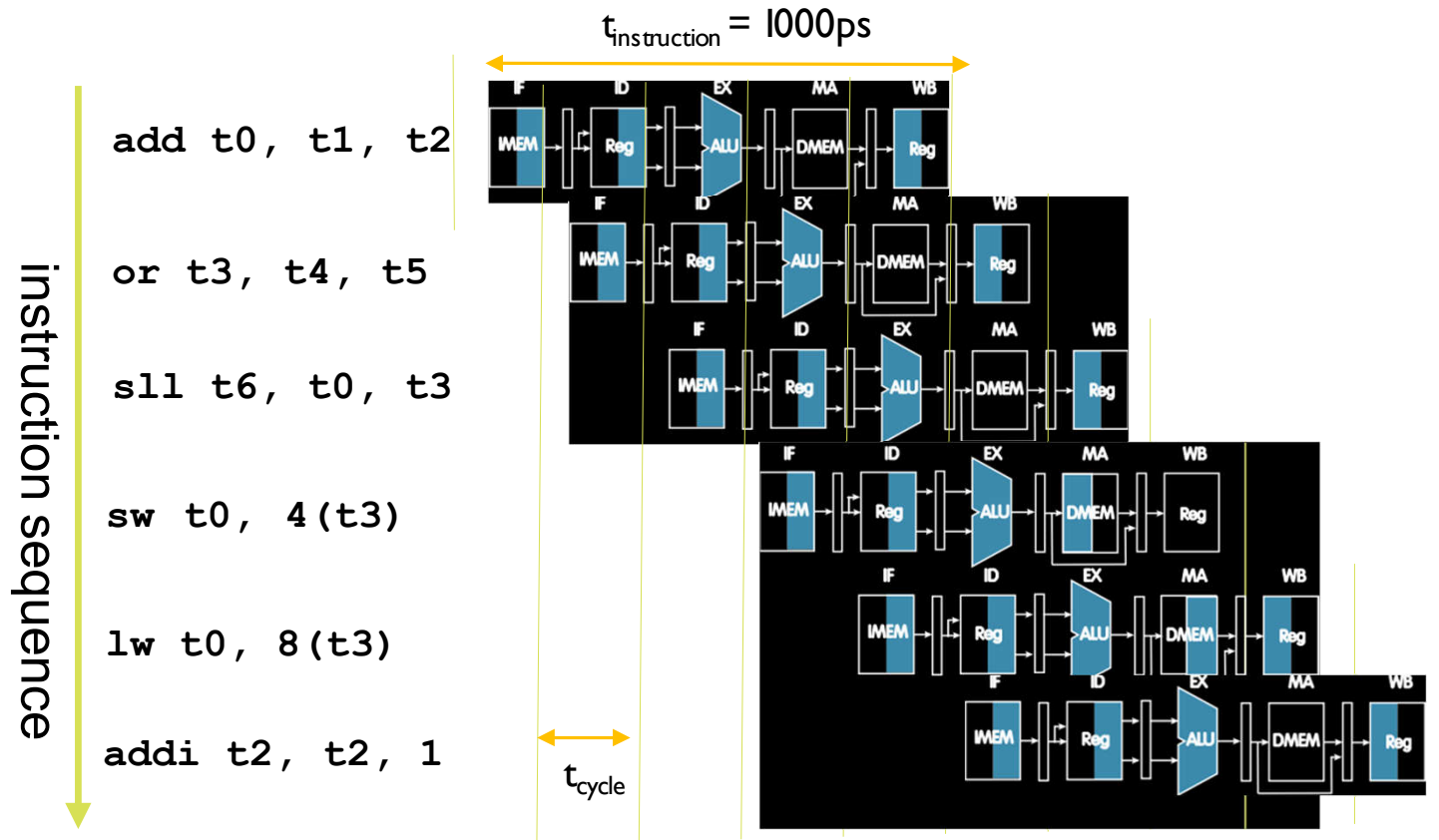


	Single Cycle	Pipelined
Timing	$t_{\text{step}} = 100 \dots 200 \text{ ps}$	$t_{\text{cycle}} = 200 \text{ ps}$
	Register access only 100 ps	All cycles same length
Instruction time, $t_{\text{instruction}}$	$= t_{\text{cycle}} = 800 \text{ ps}$	1000 ps
CPI (Cycles Per Instruction)	~ 1 (ideal)	~ 1 (ideal), < 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

RISC-V (31)

Sequential vs. Simultaneous

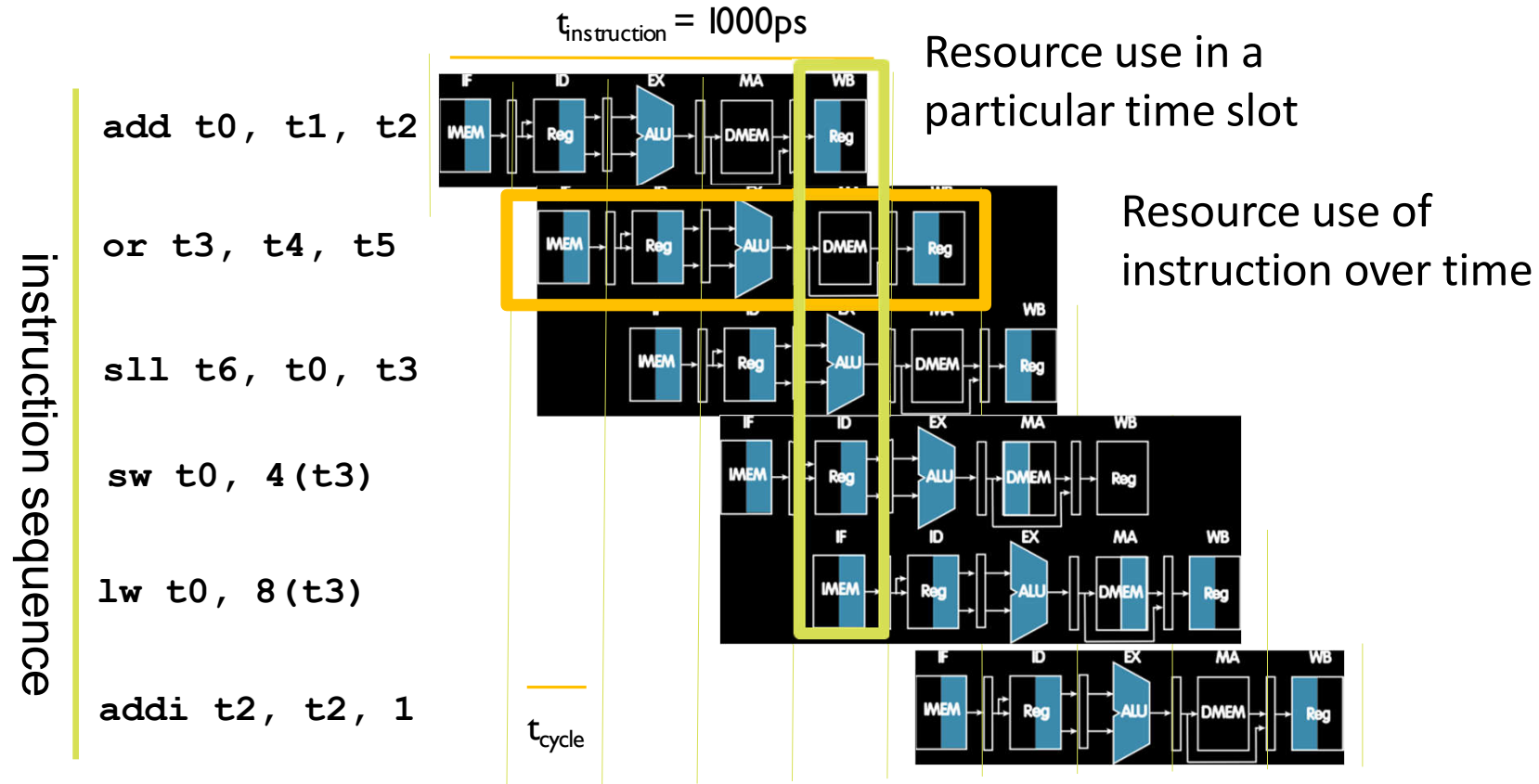
- What happens sequentially and what simultaneously?



RISC-V

Sequential vs. Simultaneous

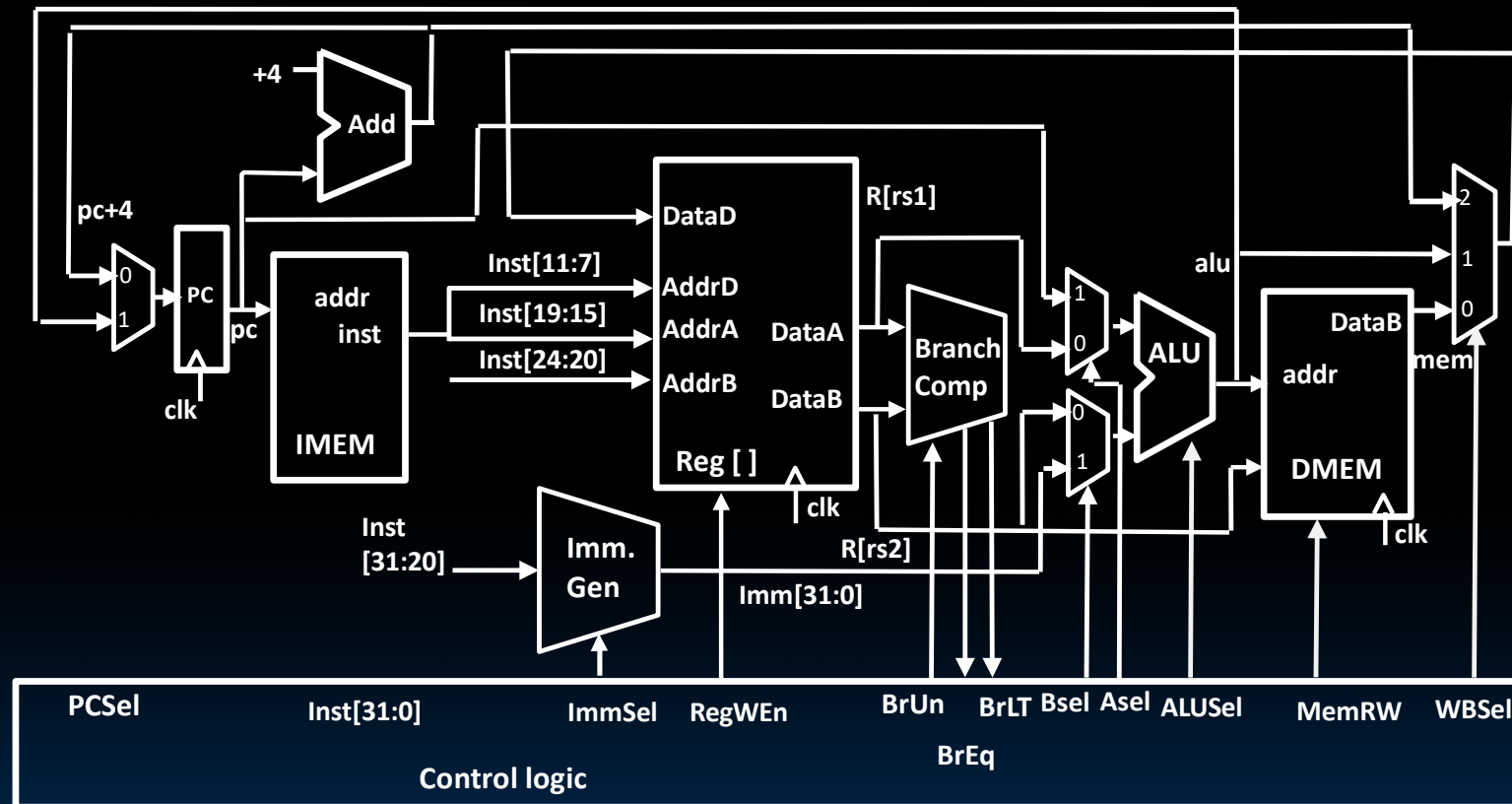
- What happens sequentially and what simultaneously?



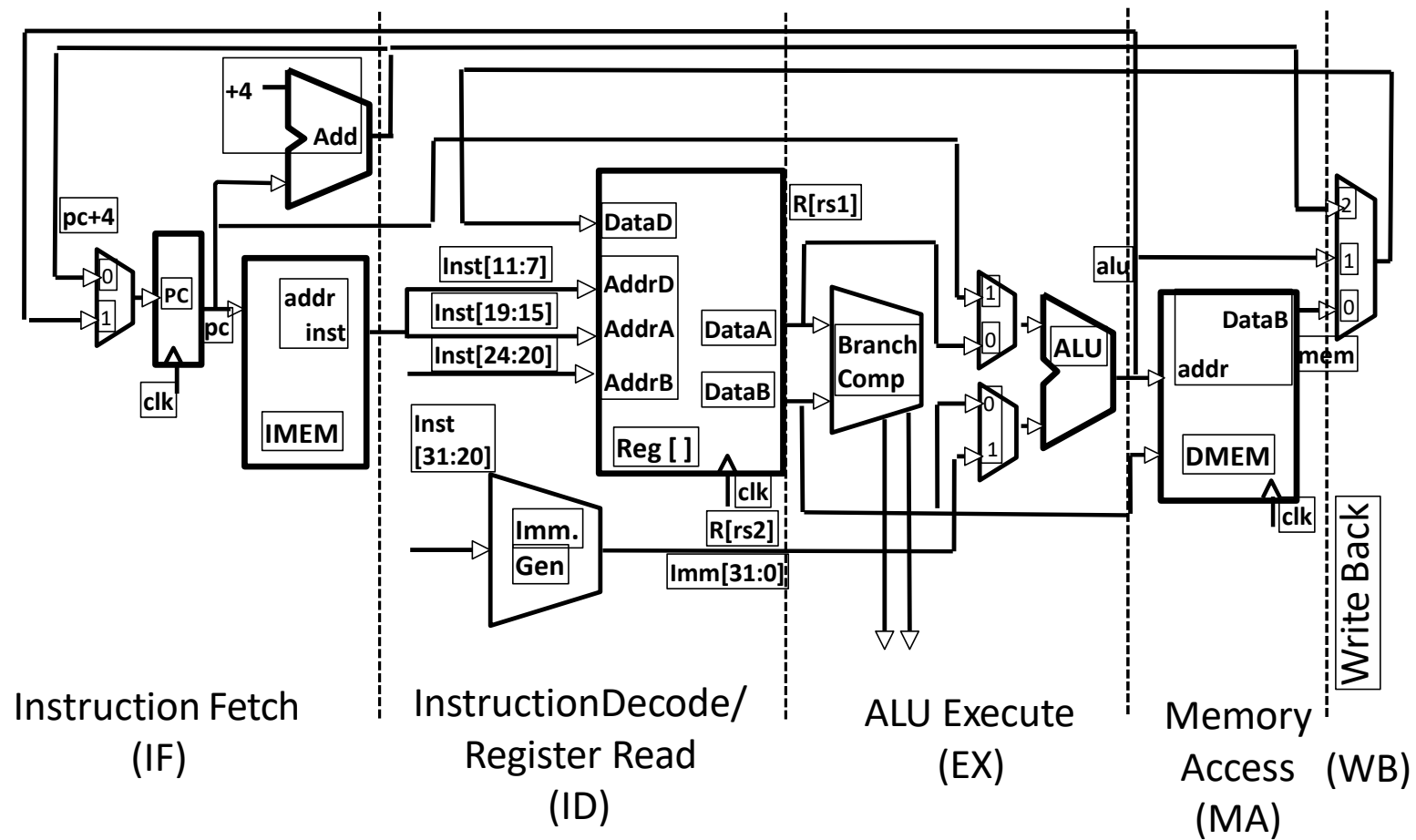
RISC-V

Pipelining Datapath

Single-Cycle

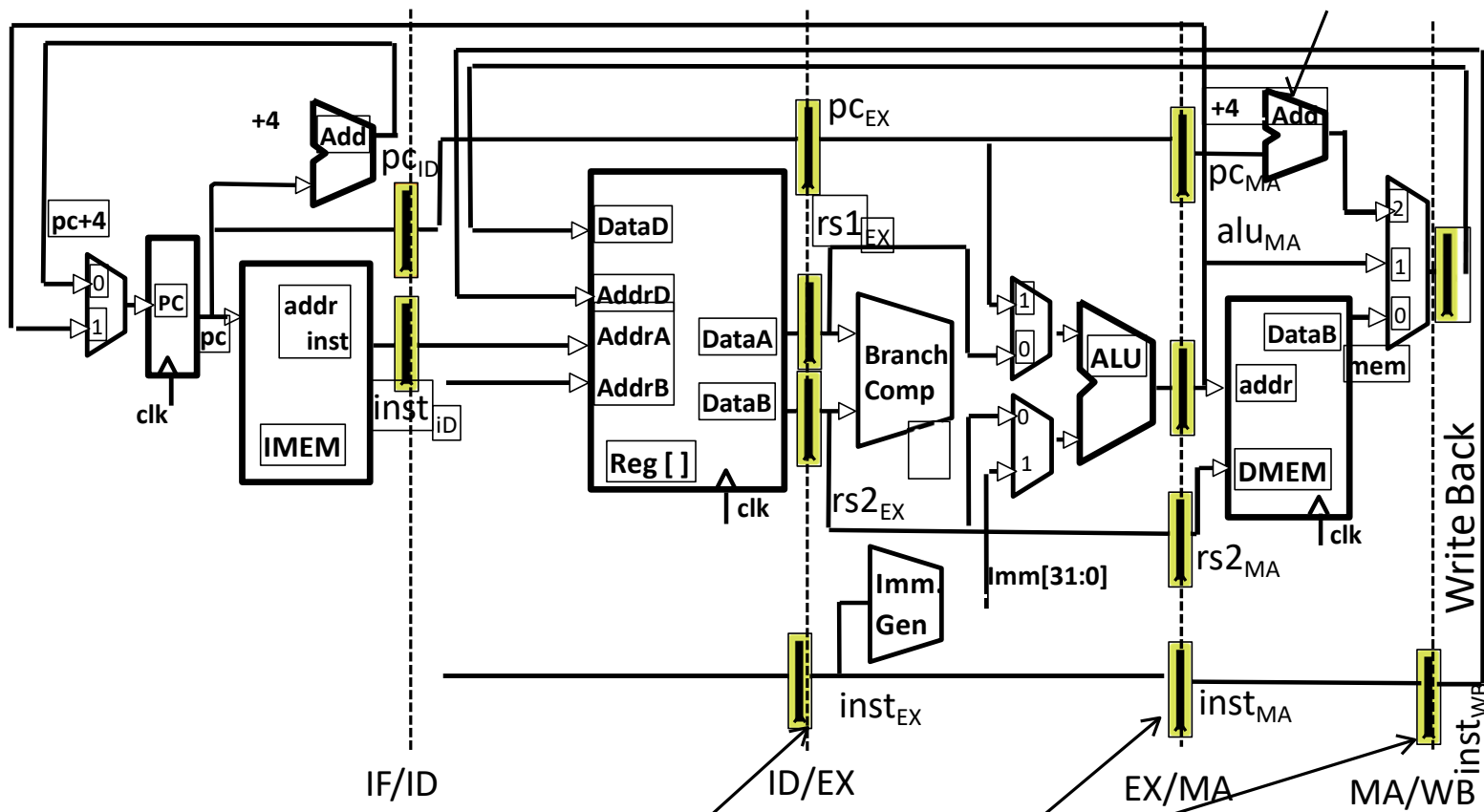


Single-Cycle RV32I Datapath



RISC-V (36)

Pipelined RV32I Datapath

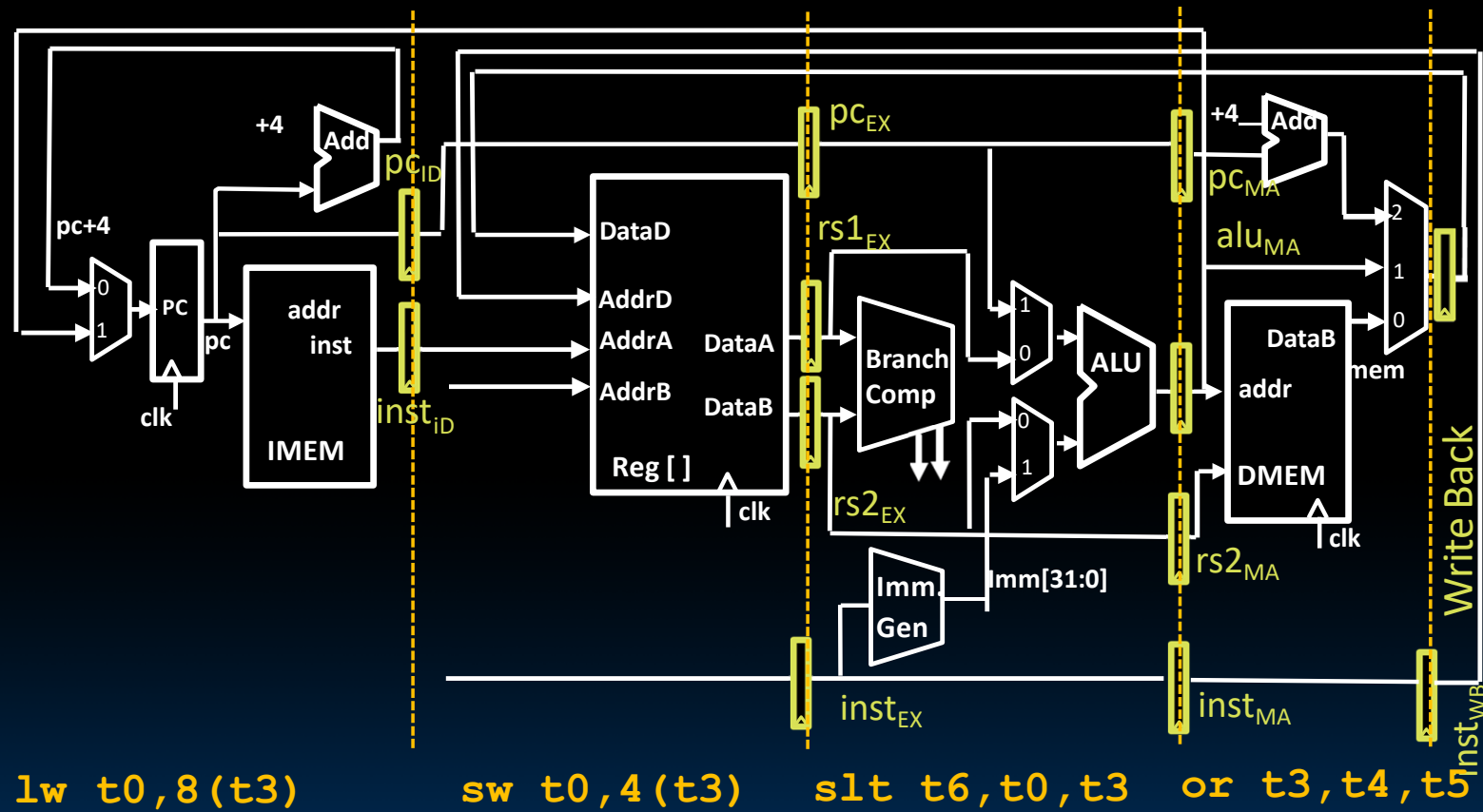


Recalculate PC+4 in M stage to avoid sending both PC and PC+4 down pipeline

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

RISC-V (37)

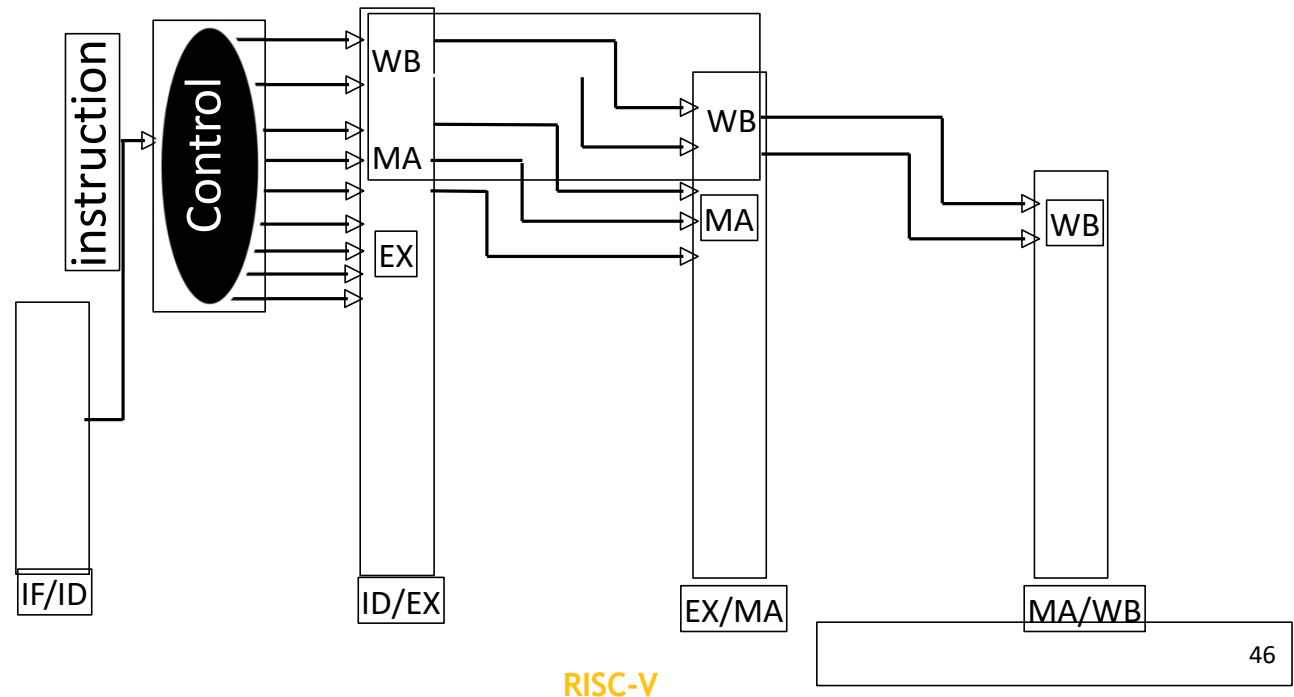
Pipelined RV32I Datapath



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

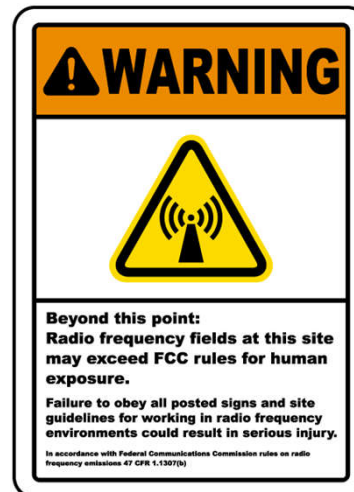
Pipelined Control

- 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!



RISC-V

RISC-V

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

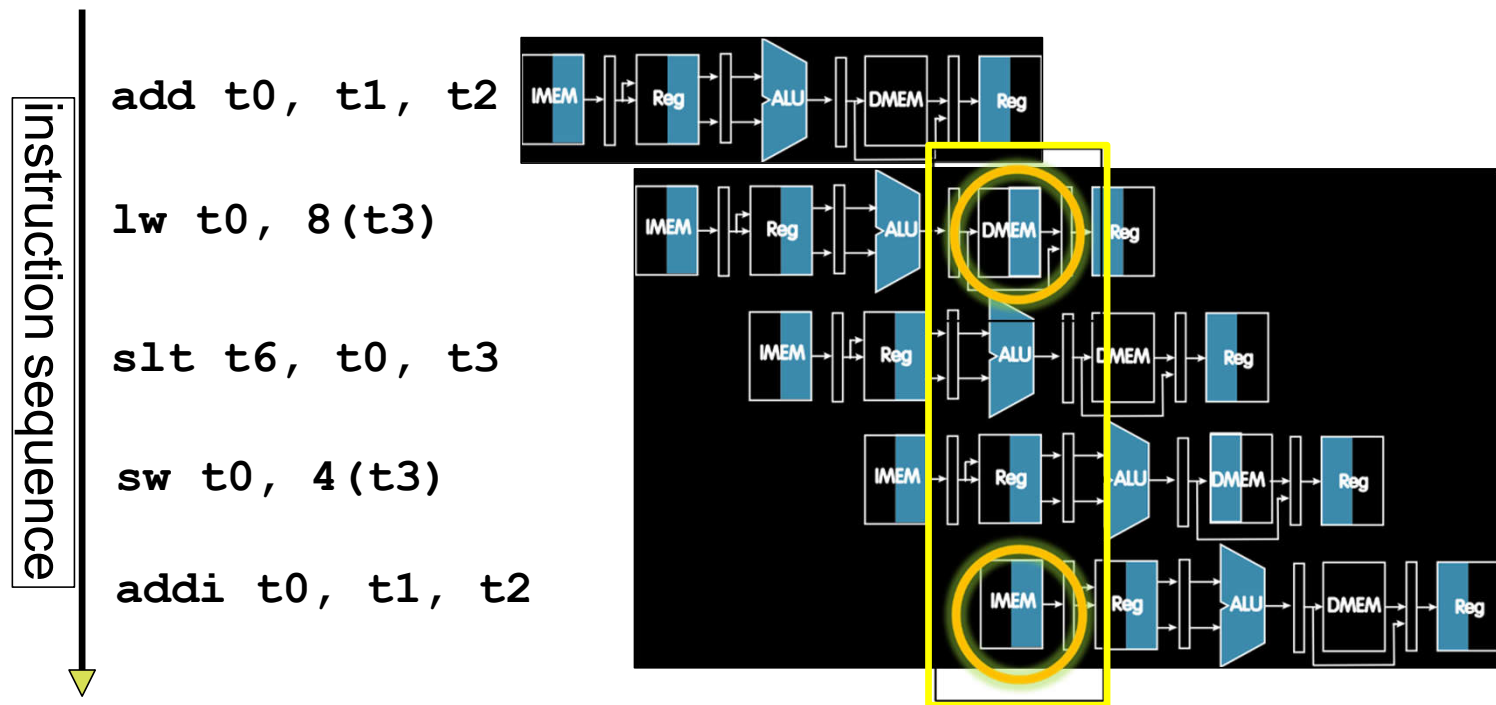
- **Problem:** Two or more instructions in the pipeline compete for access to a single physical resource
- **Solution 1:** Instructions take it in turns to use resource, some instructions have to stall
- **Solution 2:** Add more hardware to machine
- Can always solve a structural hazard by adding more 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

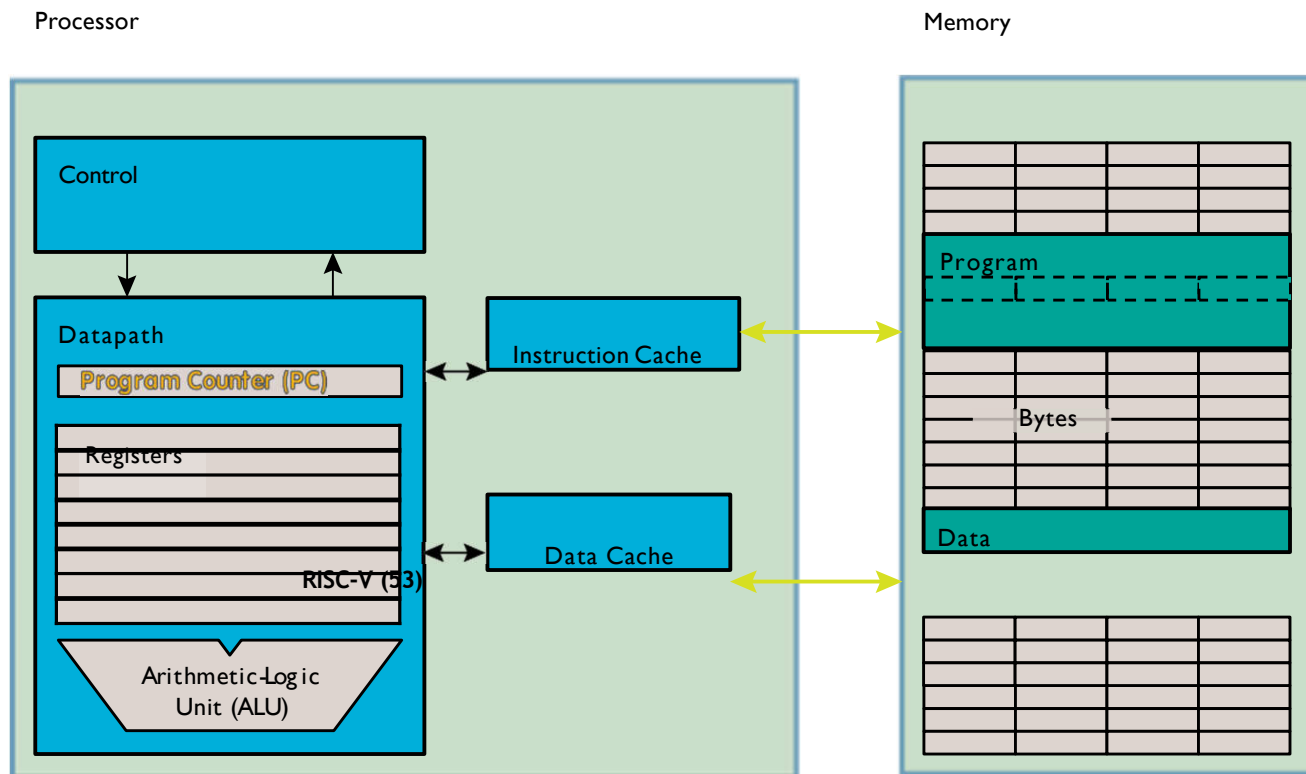
Structural Hazard: Memory Access

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



Instruction and Data Caches

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



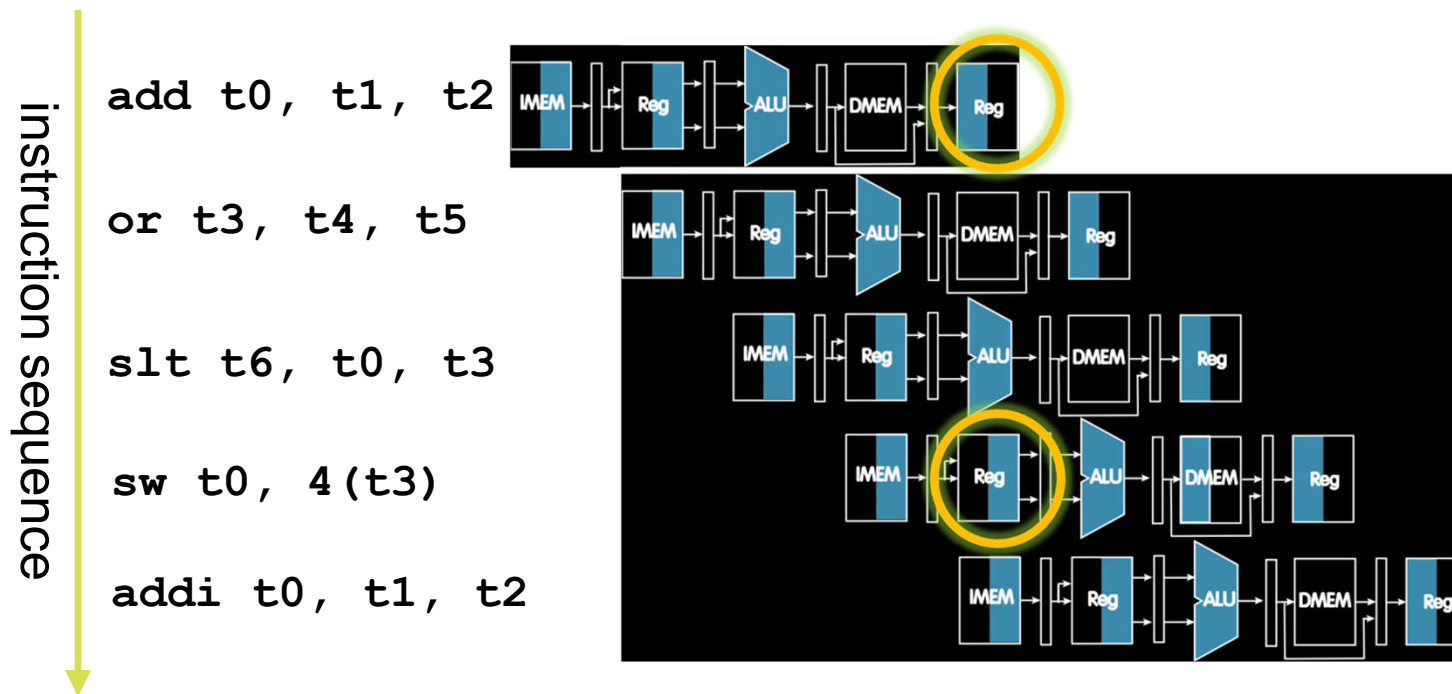
Structural Hazards – Summary

- 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

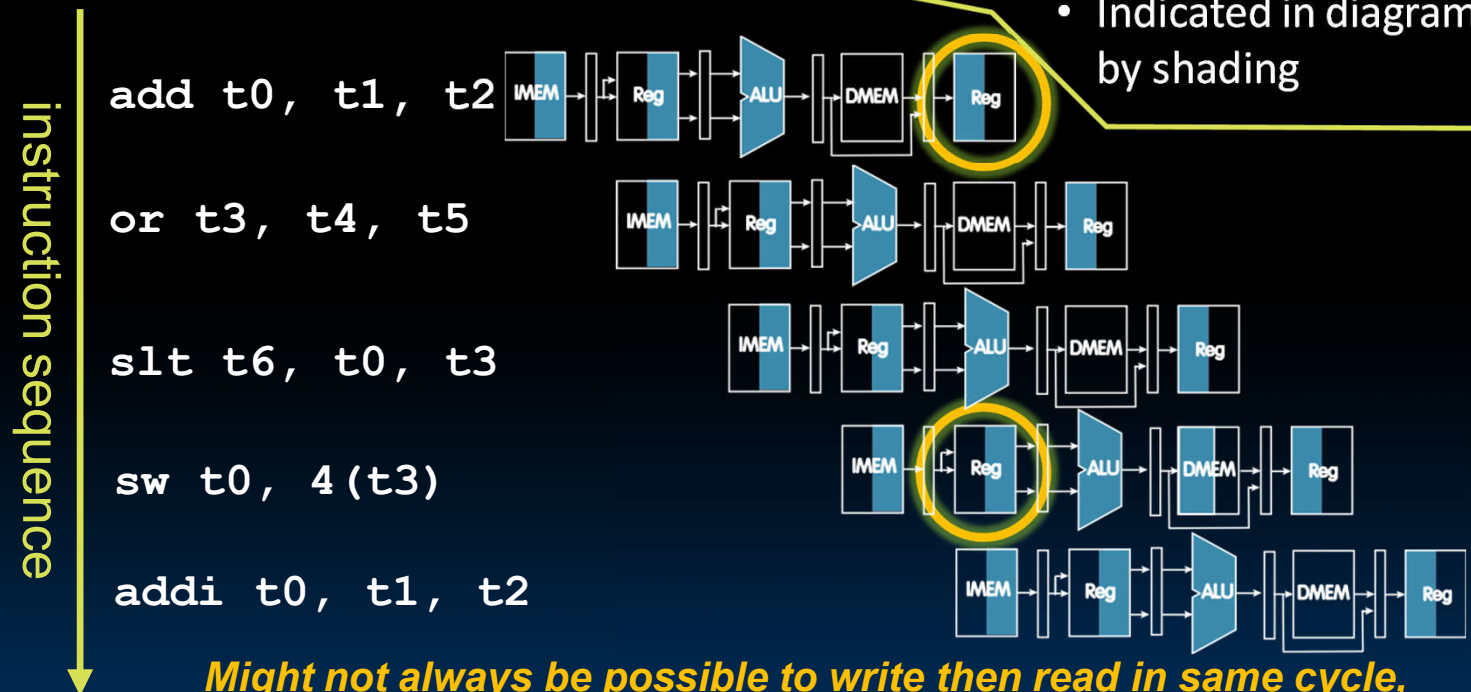
- Separate ports, but what if write to same register as read?
- Does **sw** in the example fetch the old or new value?



- Exploit high speed of register file (100 ps)

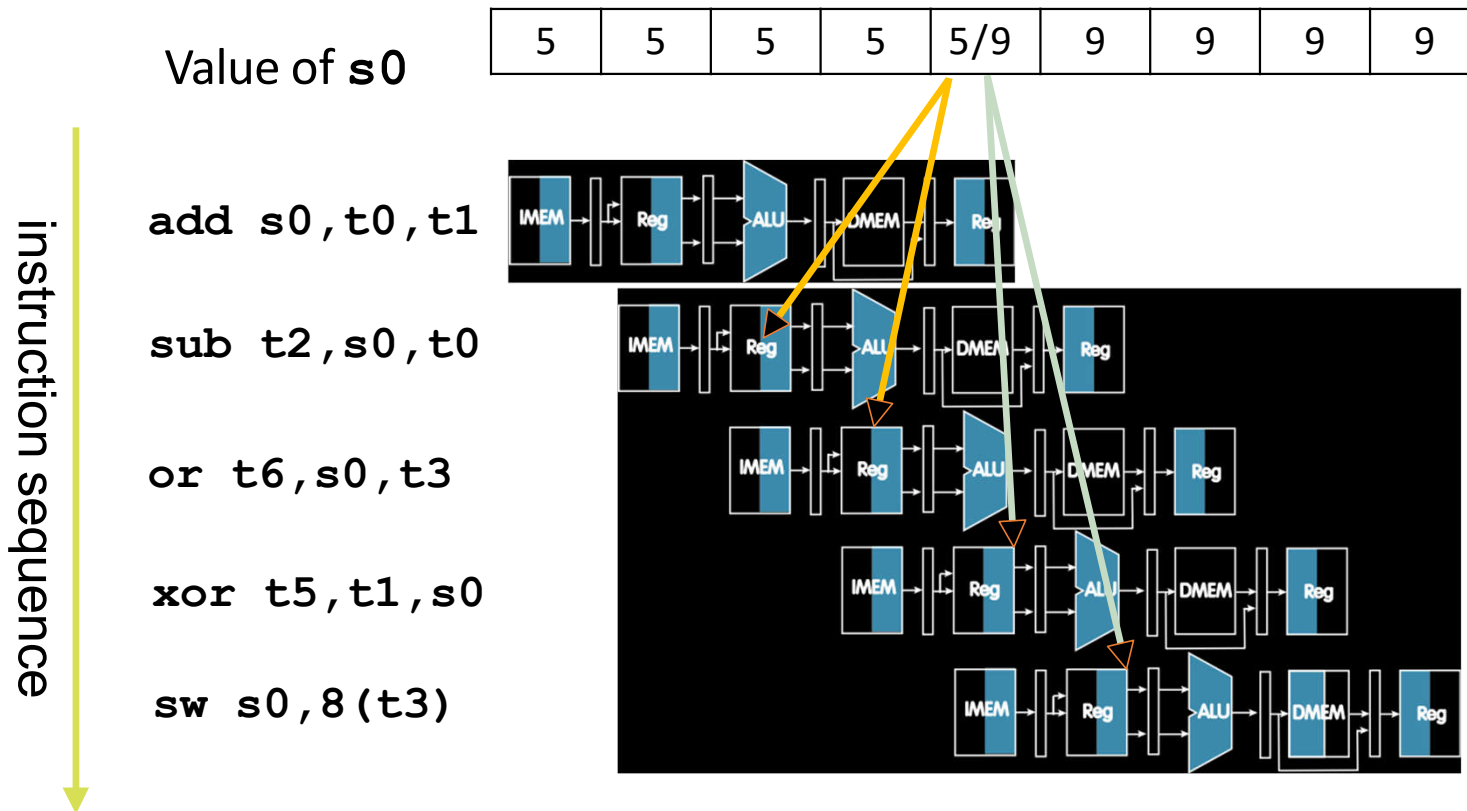
- 1) WB updates value
- 2) ID reads new value

- 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!

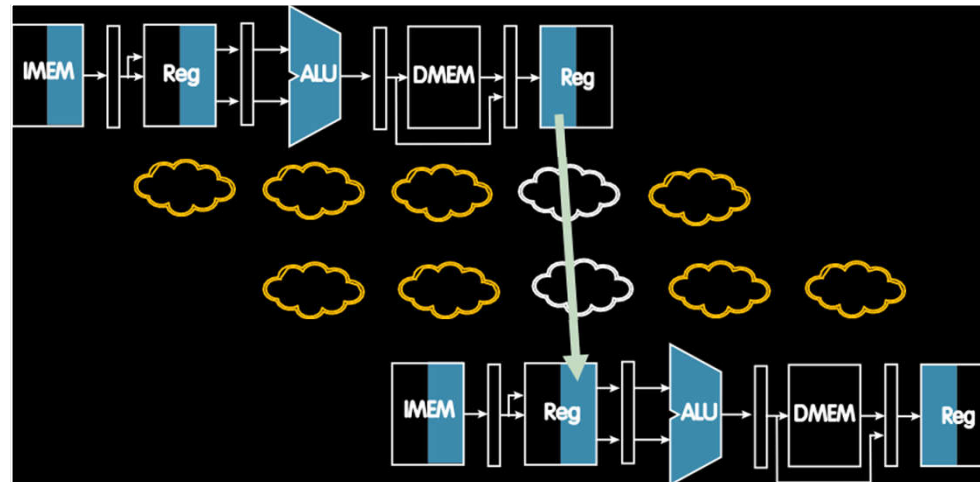
Solution 1: Stalling

- Problem: Instruction depends on result from previous instruction

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

```
add s0, t0, t1  
    "bubbles"
```

```
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

- Value of **s0**

• `add s0, t0, t1` `sub t2, s0, t0`

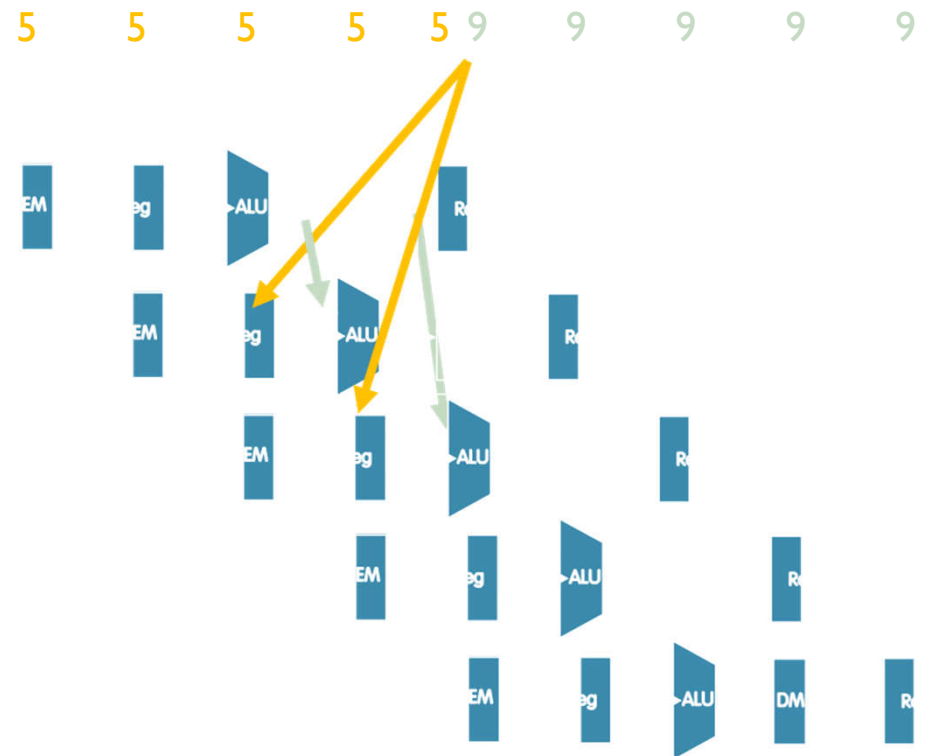
• `or t6, s0, t3`

• `xor t5, t1, s0` `sw s0, 8(t3)`

instruction sequence

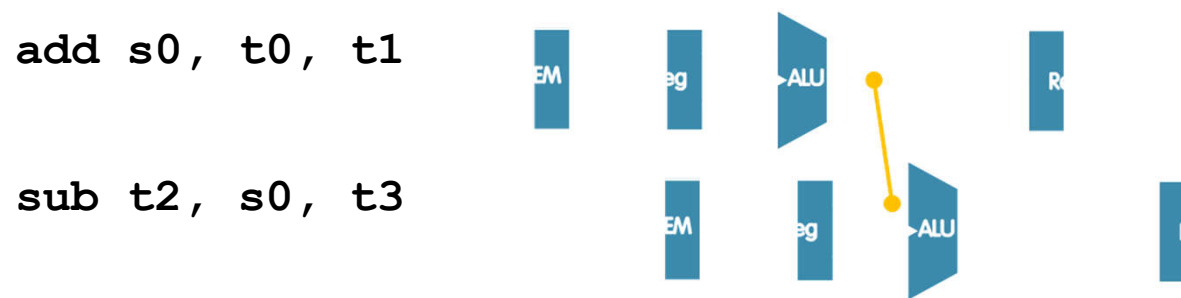
- Forwarding: grab operand from pipeline stage, rather than register file

RISC-V (61)



Forwarding (aka Bypassing)

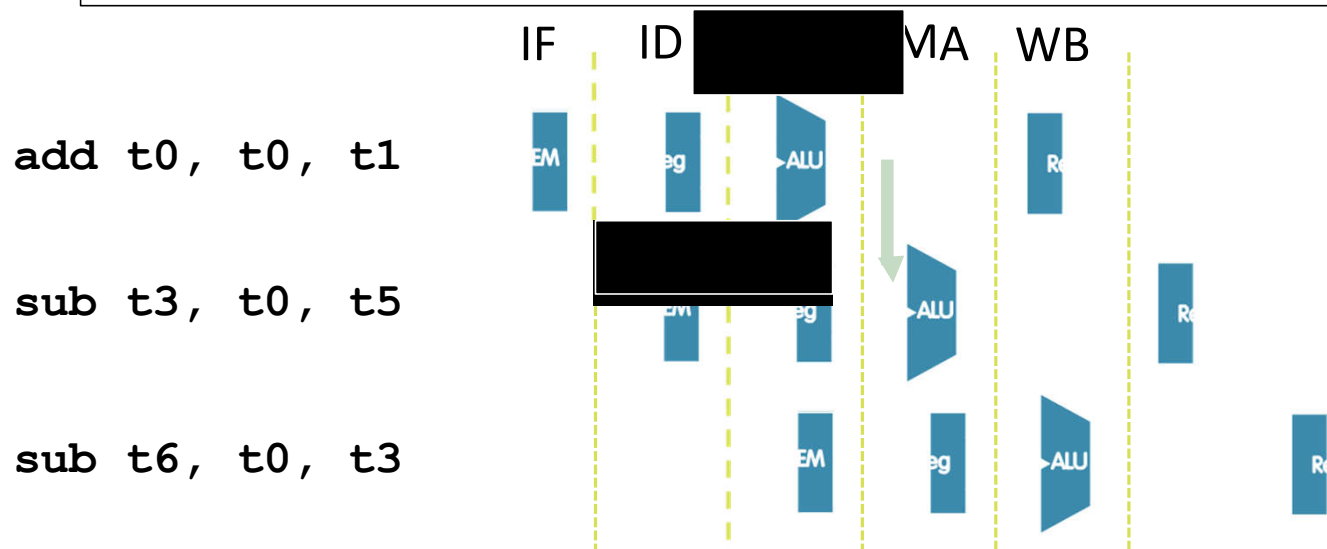
- Use result when it is computed
 - Don't wait for it to be stored in a register
 - Requires extra connections in the datapath



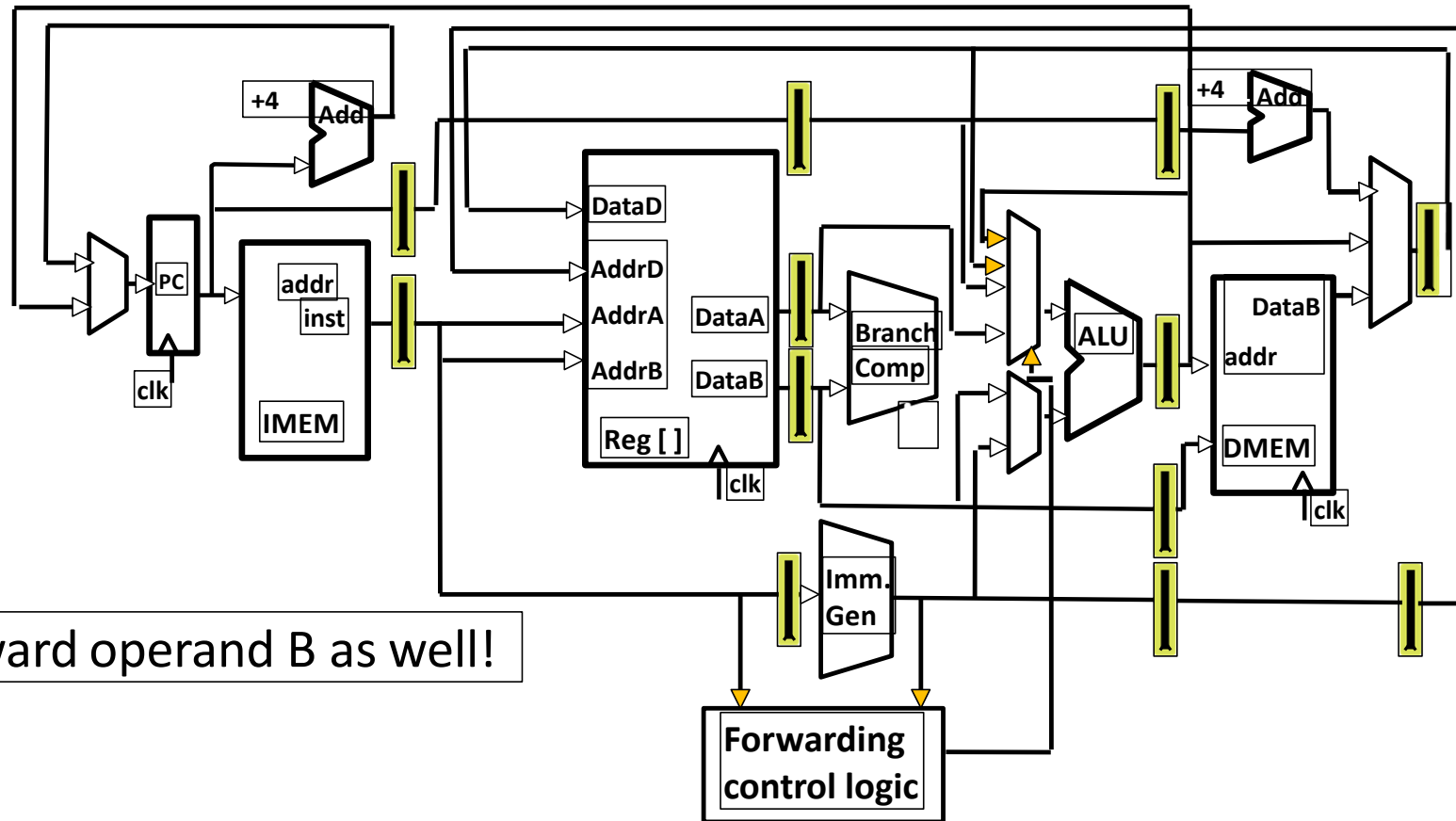
RISC-V (62)

Data Needed for Forwarding (Example)

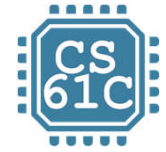
- Compare destination of older instructions in pipeline with sources of new instruction in decode stage.
- Must ignore writes to x0!



Pipelined RV32I Datapath



Remember to forward operand B as well!



Data Hazard and Forwarding

- Value of **s0**

5	5	5	5	5/9	9	9	9	9
---	---	---	---	-----	---	---	---	---

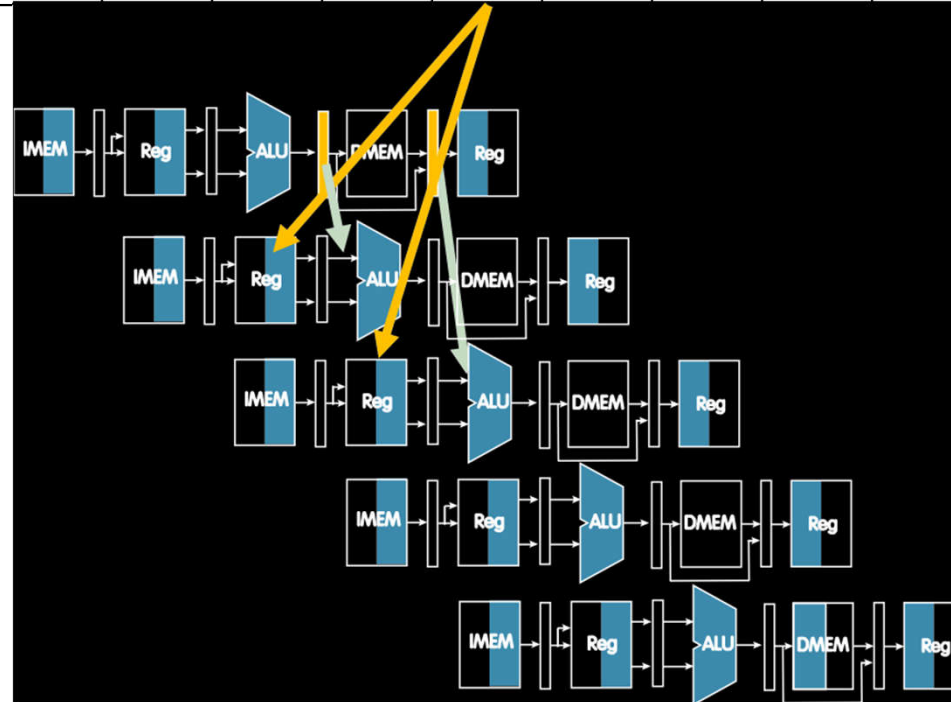
- add s0, t0, t1

- ```
•sub t2, s0, t0
```

- or  $t_6, s_0, t_3$

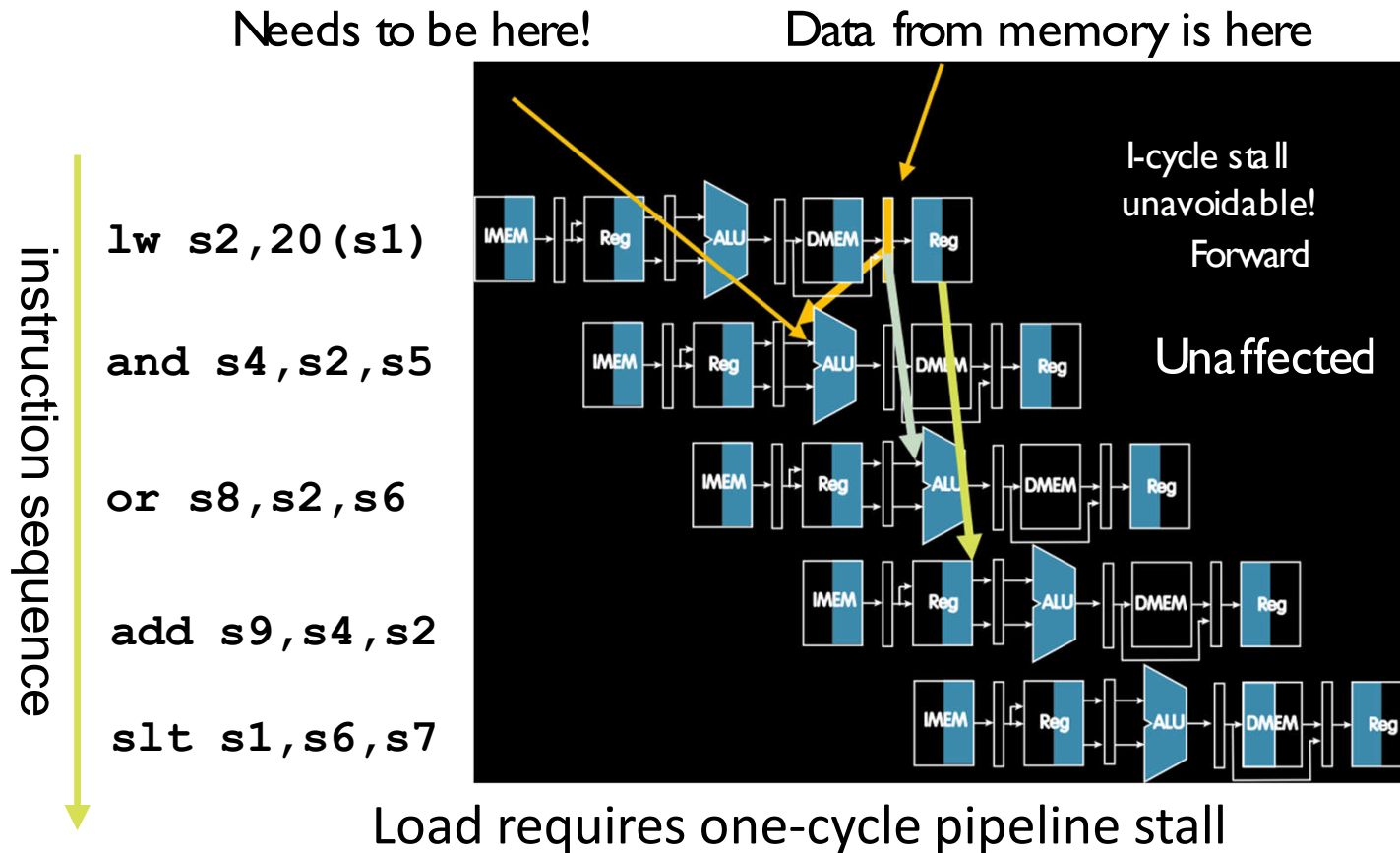
- `xor t5,t1,s0`

- **sw** **s0**, 8 (t3)

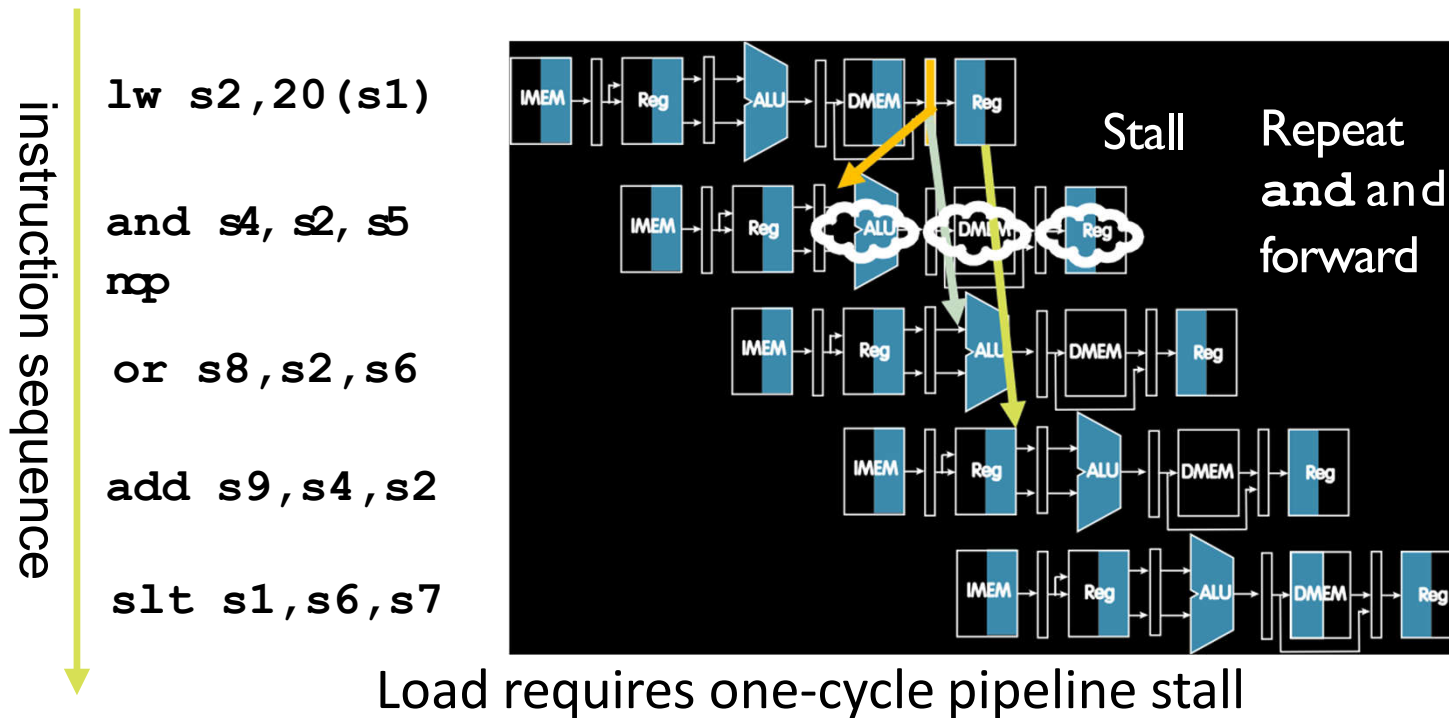


- Forwarding: grab operand from pipeline stage, rather than register file

# Load Data Hazard



# Stall Pipeline

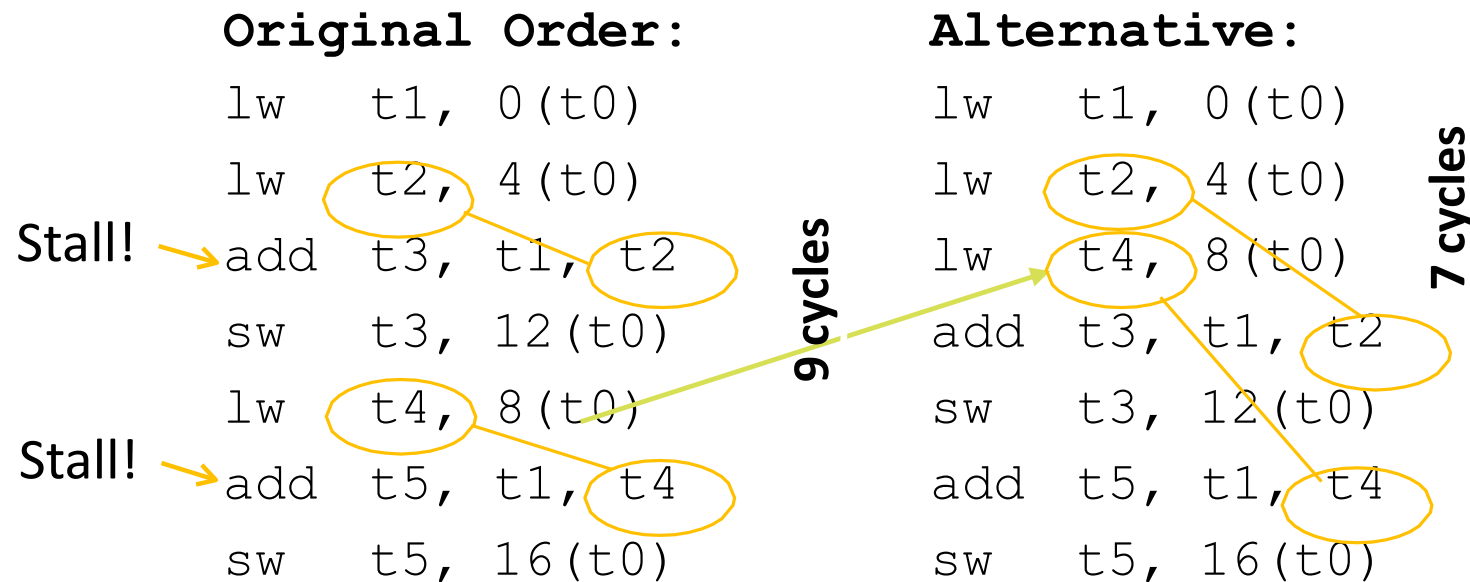


# lw Data Hazard

- 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

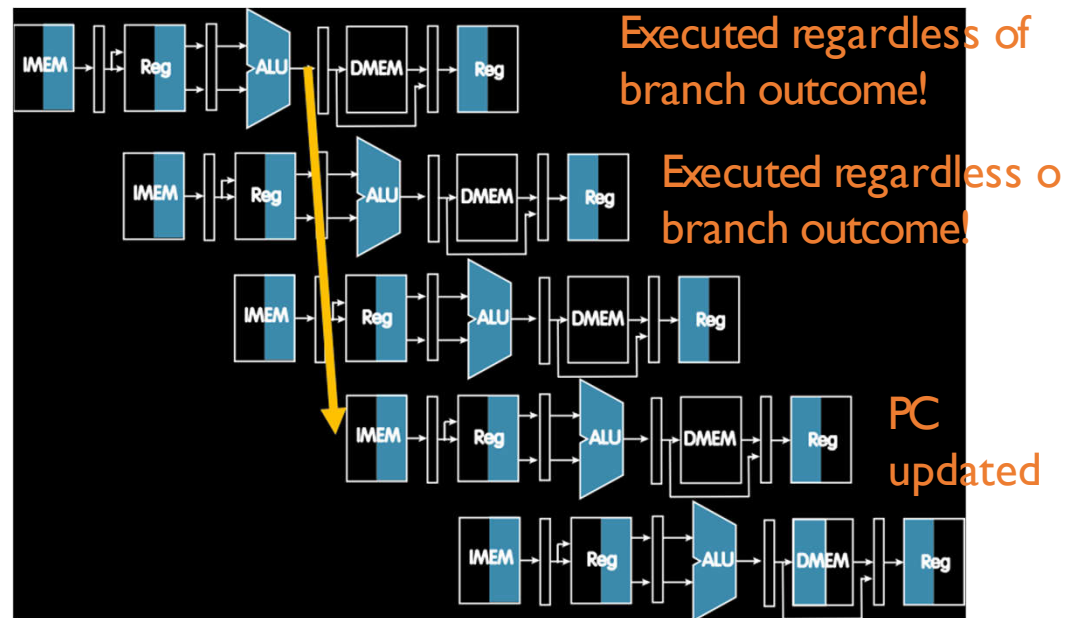
`beq t0,t1,Label`

`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



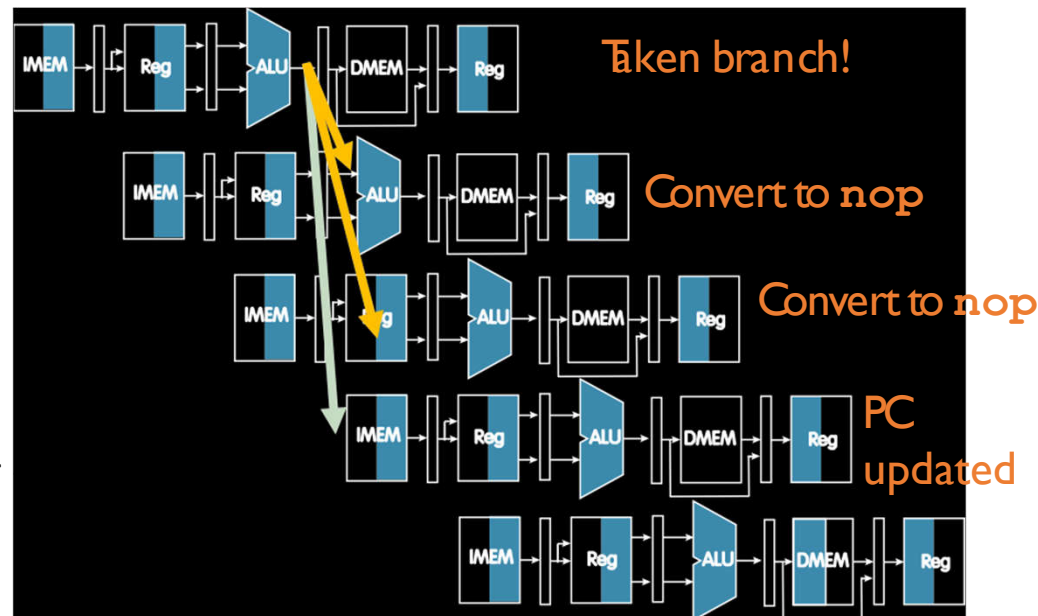
# Kill Instructions after Branch if Taken

`beq t0,t1,Label`

`sub t2,s0,t0`

`or t6,s0,t3`

`Label: xor t5,t1`



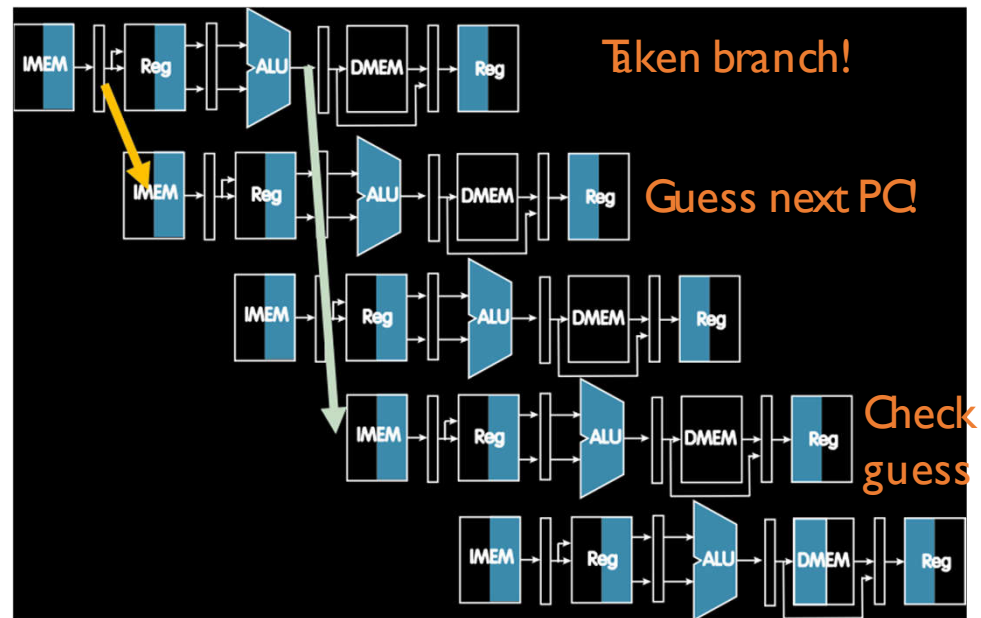
# Reducing Branch Penalties

- 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

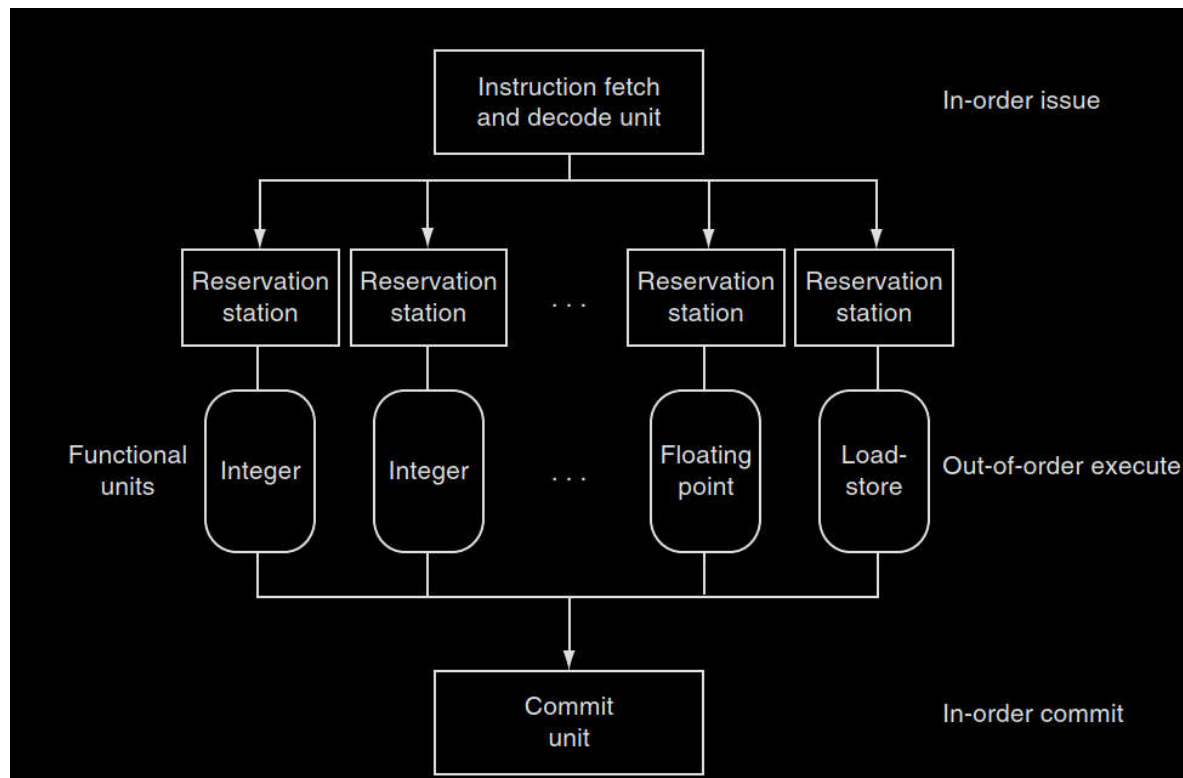
- “Overlap” instruction execution
- Deeper pipeline: 5 => 10 => 15 stages
  - Less work per stage → shorter clock cycle
  - But more potential for hazards ( $CPI > 1$ )

## 3. Multi-issue “superscalar” processor

# Superscalar Processor

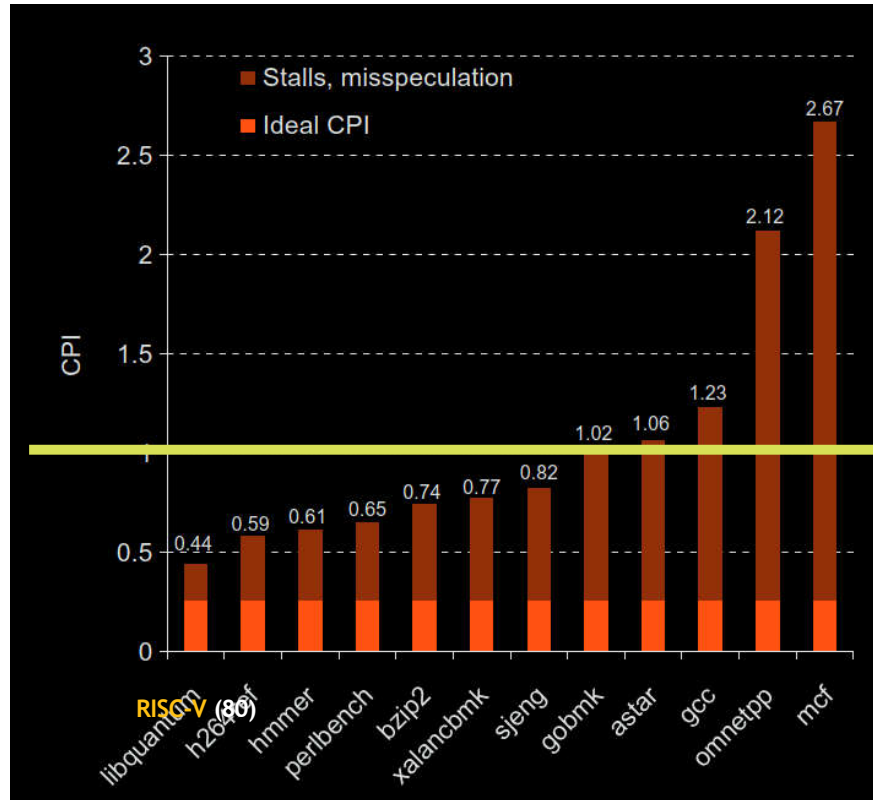
- Multiple issue “superscalar”
  - Replicate pipeline stages  $\Rightarrow$  multiple pipelines
  - Start multiple instructions per clock cycle
  - $CPI < 1$ , so use Instructions Per Cycle (IPC)
  - E.g., 4GHz 4-way multiple-issue
    - 16 BIPS, peak  $CPI = 0.25$ , peak  $IPC = 4$
  - Dependencies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

# Superscalar Processor



RISC-V

# Benchmark: CPI of i7



CPI = 1





# “Iron Law” of Processor Performance

CPI = Cycles Per Instruction

Can time

Can count

Can look up

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

RISC-V (81)

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

Berkeley  
UNIVERSITY OF CALIFORNIA



# 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 3<sup>rd</sup> stage, access memory in 4<sup>th</sup> 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