inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

Lecture 32 – Pipeline II

2018-11-07

Instructors

Dan Garcia and Bora Nikolic

www.independent.co.uk/life-style/gadgets-and-tech/news/human-brain-supercomputer-neurons-computer-simulation-manchester-university-spinnaker-artificial-a8612966.html

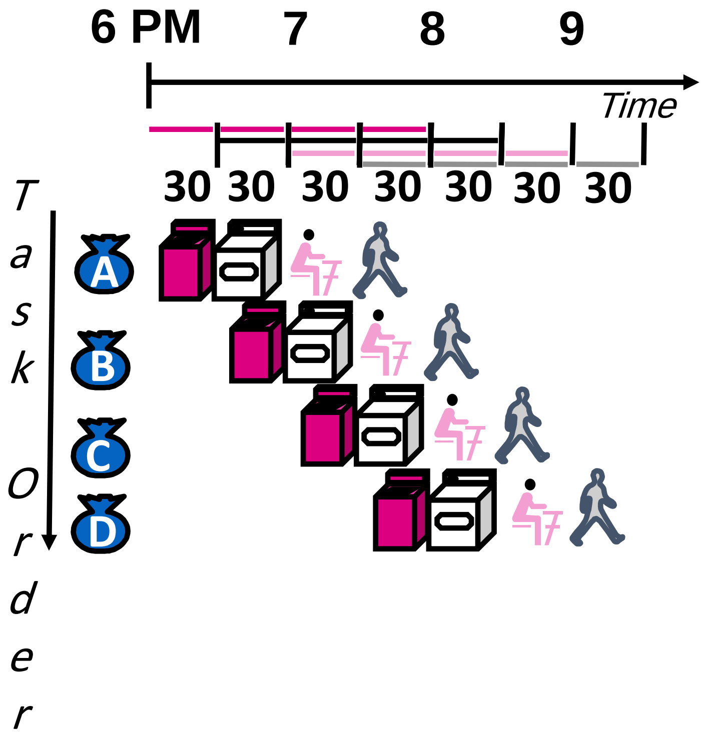
'Human Brain' Supercomputer Switched On to Unlock Secrets of Mind

Researchers at the University of Manchester in the U.K. are about to launch a supercomputer designed to mimic the human brain, comprised of 1 million processors capable of 200 trillion actions per second. The Spiking Neural Network Architecture (SpiNNaker) mimics the parallel communication architecture of the brain by sending small amounts of information to different destinations concurrently. Although SpiNNaker is the first step toward creating a model of 1 billion biological neurons in real time, for now it will provide new insight into how the brain works. Researchers will use SpiNNaker to run large-scale, real-time simulations of various regions of the brain; the system also can be adapted to power an artificial intelligence robot, which can navigate and interpret objects in the real world. Manchester's Steve Furber says the robotic system "works as real-time neural simulator that allows roboticists to design large-scale neural networks into mobile robots so they can walk, talk, and move with flexibility and low power."

Review

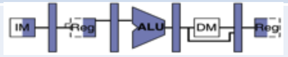
- Controller
  - Tells universal datapath how to execute each instruction
- Instruction timing
  - Set by instruction complexity, architecture, technology
  - Pipelining increases clock frequency, “instructions per second”
    - But does not reduce time to complete instruction
- Performance measures
  - Different measures depending on objective
    - Response time
    - Jobs / second
    - Energy per task

Pipelining Overview (Review)



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

Pipelining with RISC-V

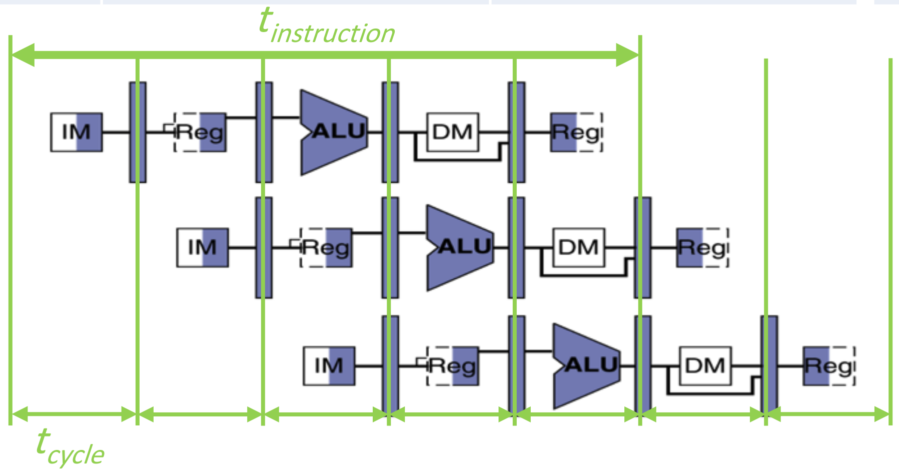
Phase	Pictogram	t_{step} Serial	t_{cycle} Pipelined
Instruction Fetch		200 ps	200 ps
Reg Read		100 ps	200 ps
ALU		200 ps	200 ps
Memory		200 ps	200 ps
Register Write		100 ps	200 ps
$t_{instruction}$		**800 ps**	**1000 ps**

instruction sequence

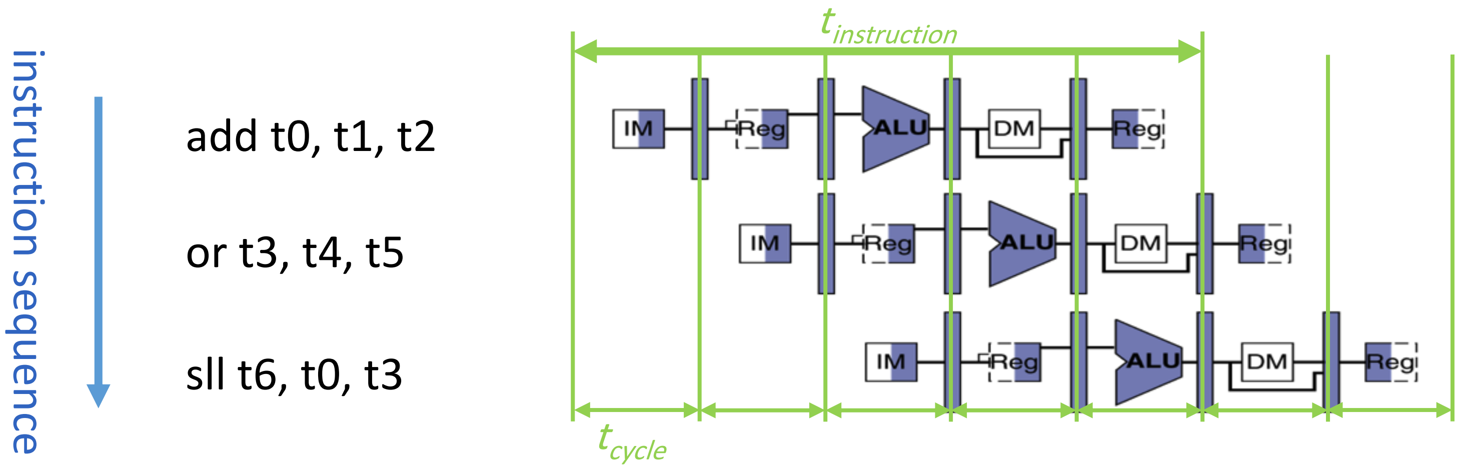
add t0, t1, t2

or t3, t4, t5

sll t6, t0, t3



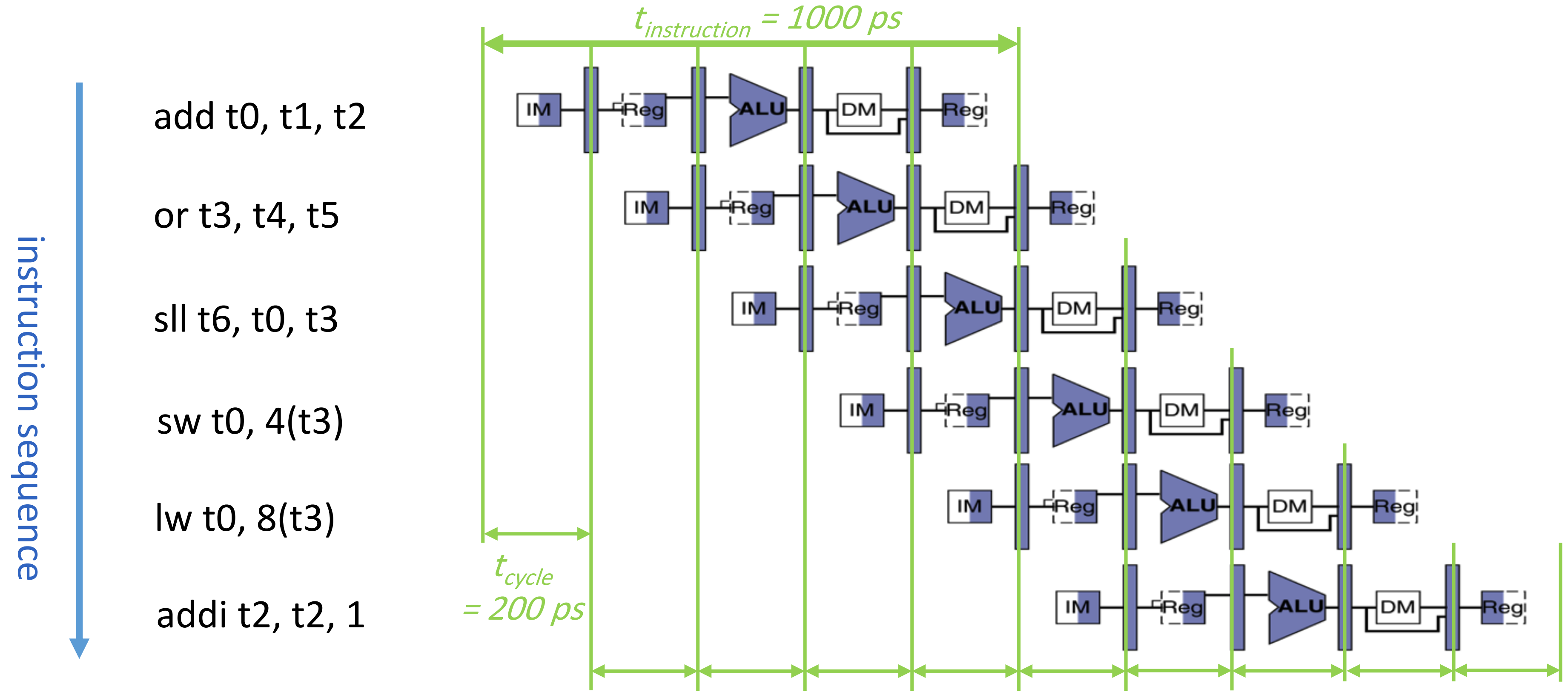
Pipelining with RISC-V



	Single Cycle	Pipelining
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, what happens simultaneously?



RISC-V Pipeline

instruction sequence

add t0, t1, t2

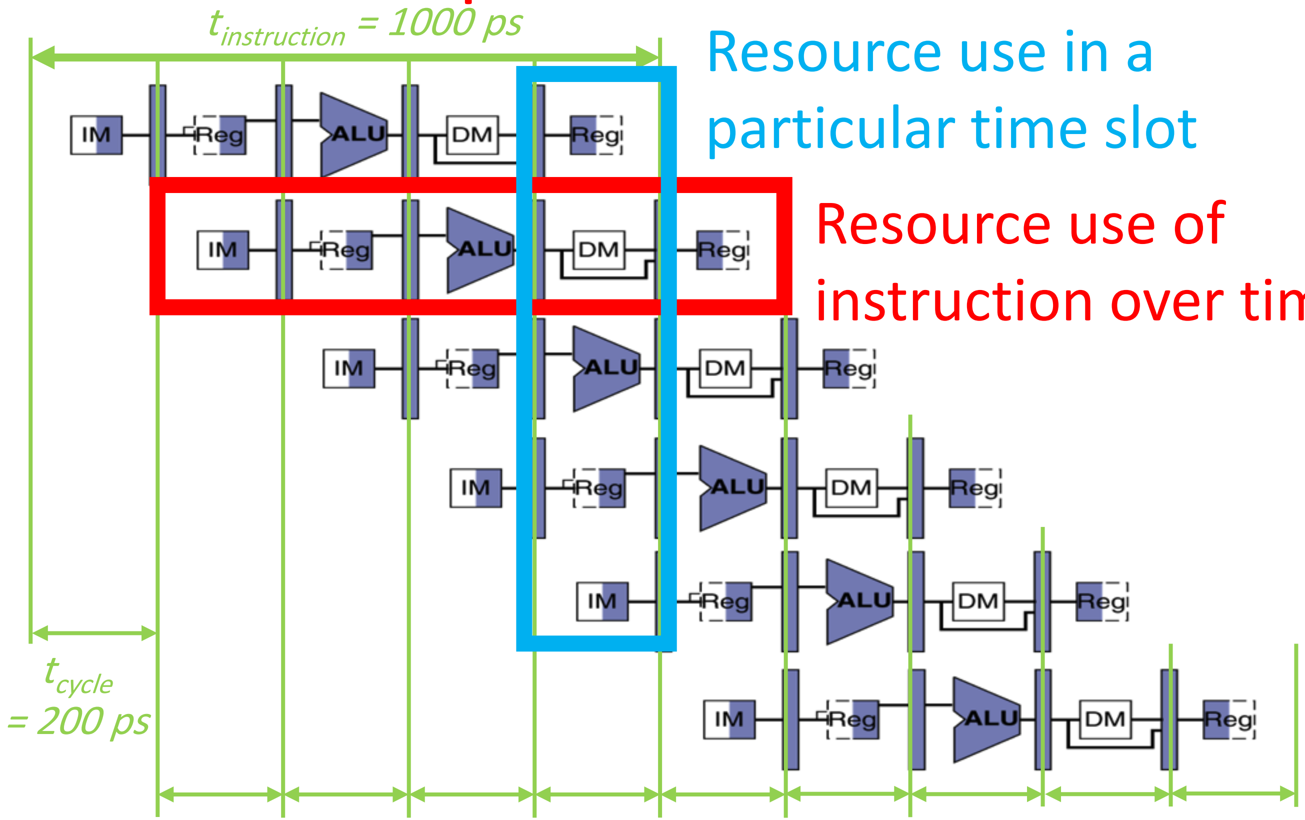
or t3, t4, t5

slt t6, t0, t3

sw t0, 4(t3)

lw t0, 8(t3)

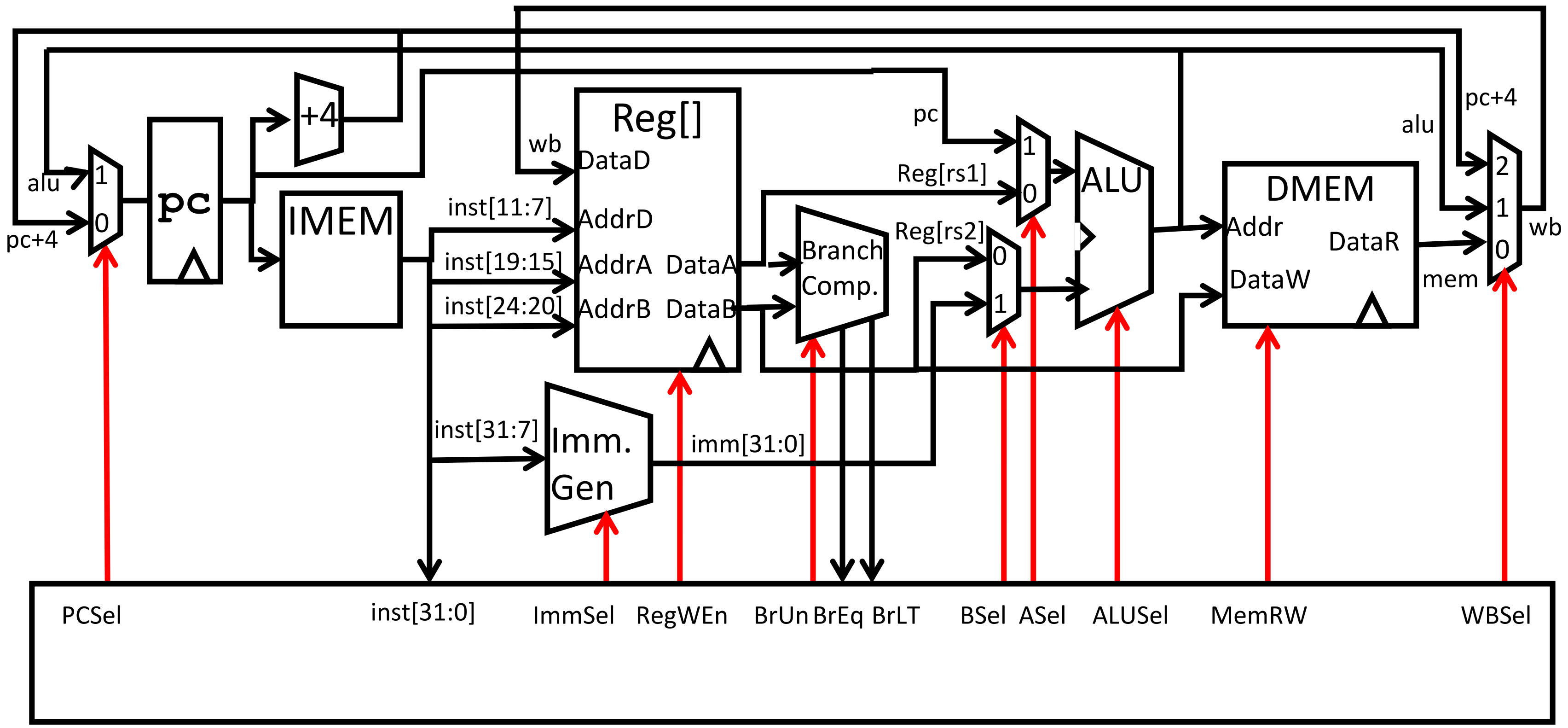
addi t2, t2, 1



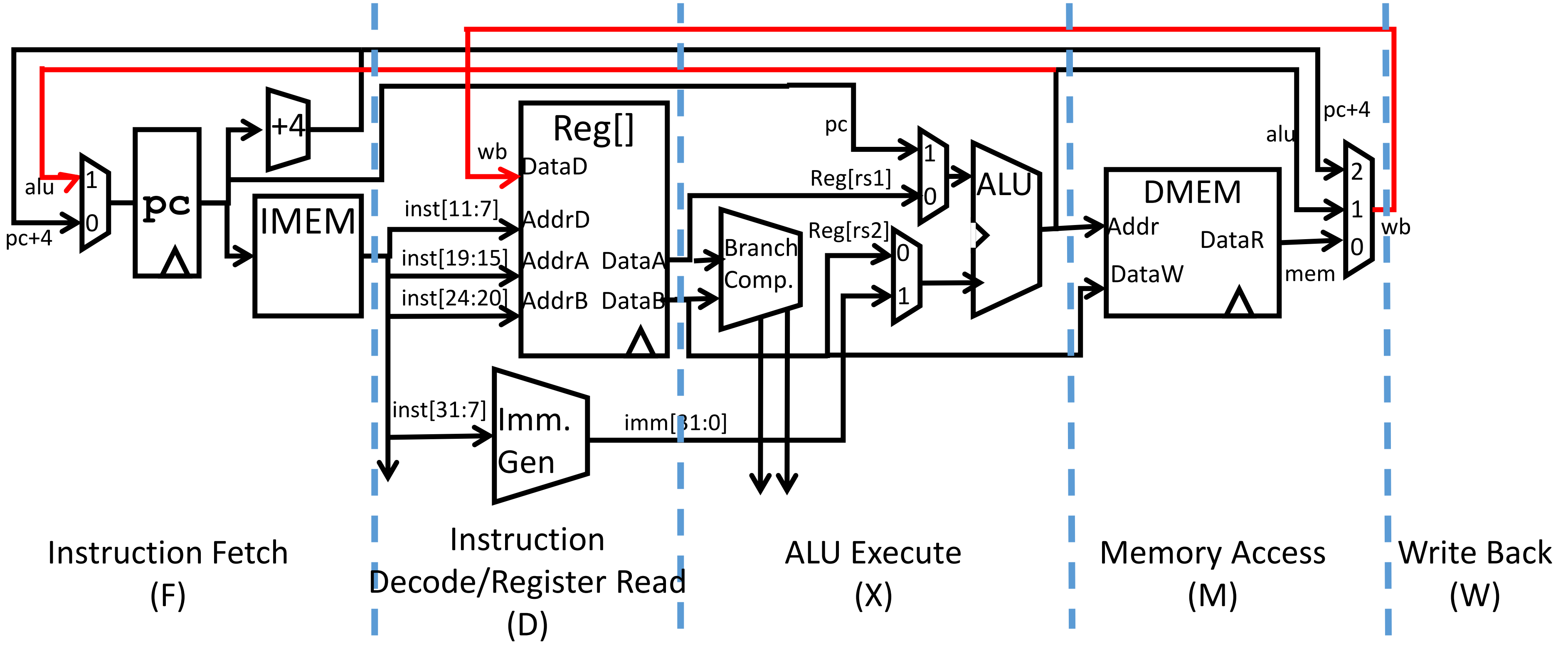
Resource use in a particular time slot

Resource use of instruction over time

Single-Cycle RISC-V RV32I Datapath



Pipelining RISC-V RV32I Datapath



Instruction Fetch (F)

Instruction Decode/Register Read (D)

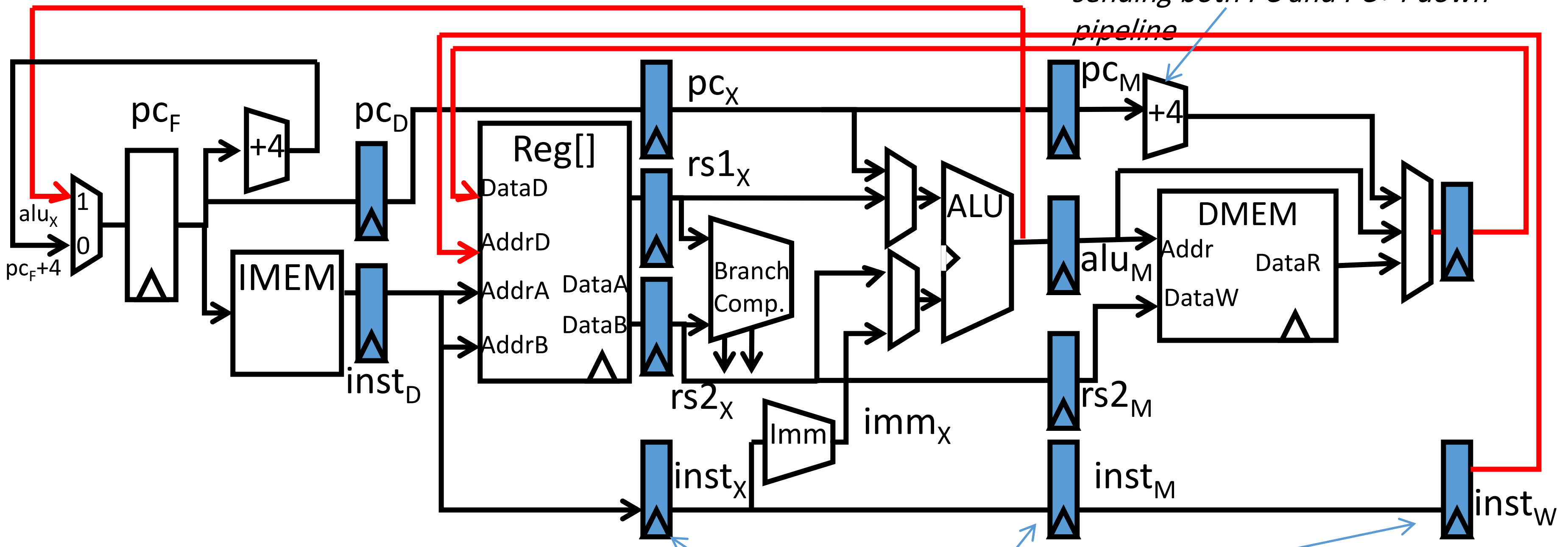
ALU Execute (X)

Memory Access (M)

Write Back (W)

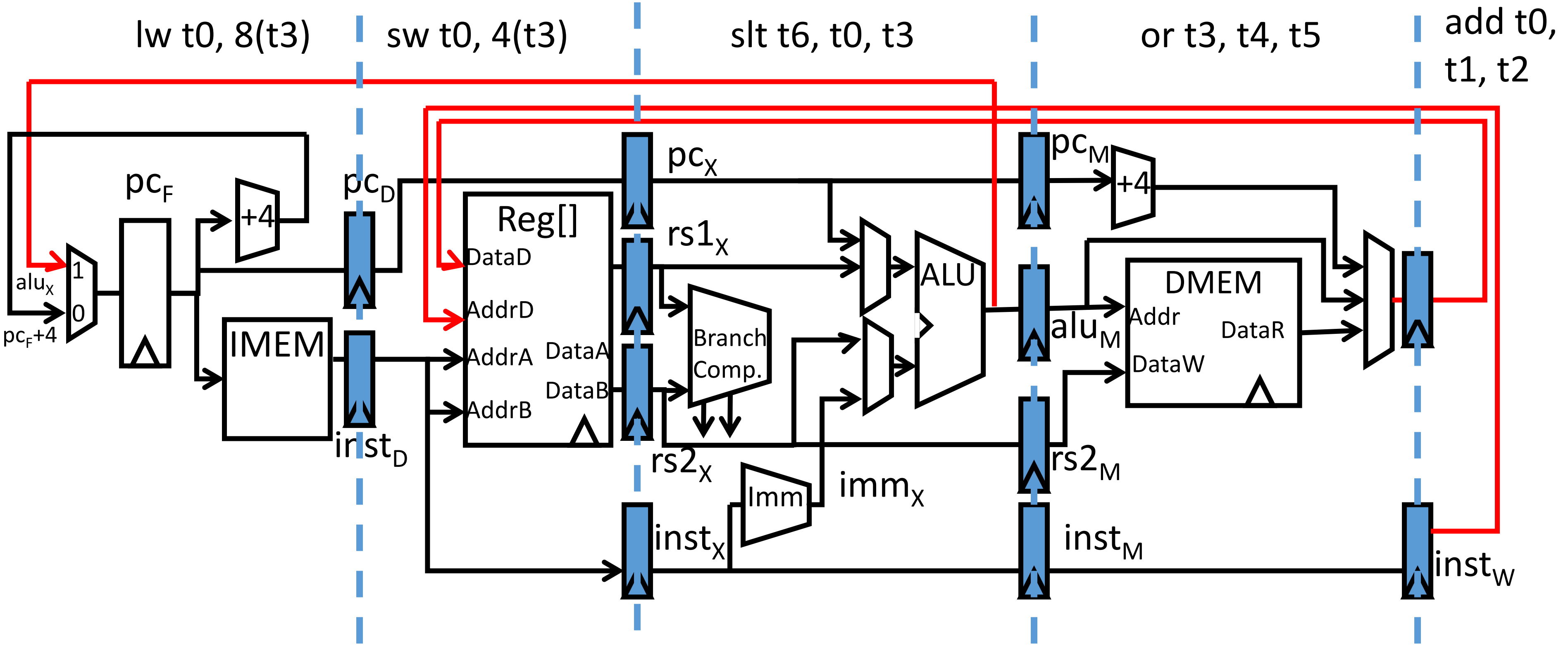
Pipelined RISC-V RV32I Datapath

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

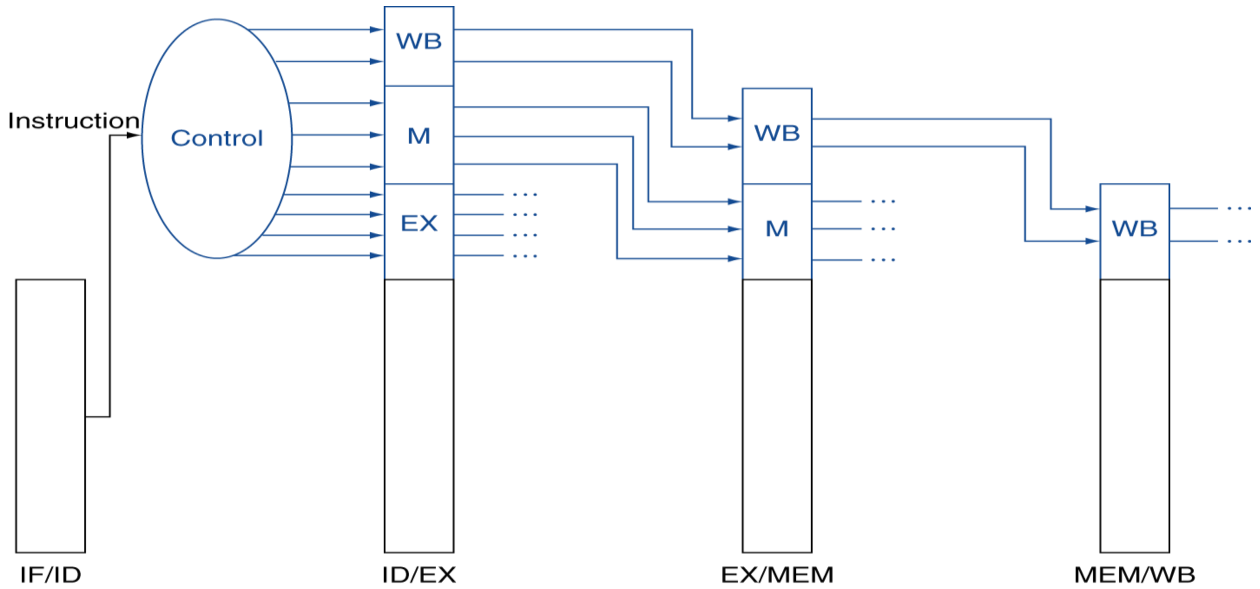
Each stage operates on different instruction



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



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

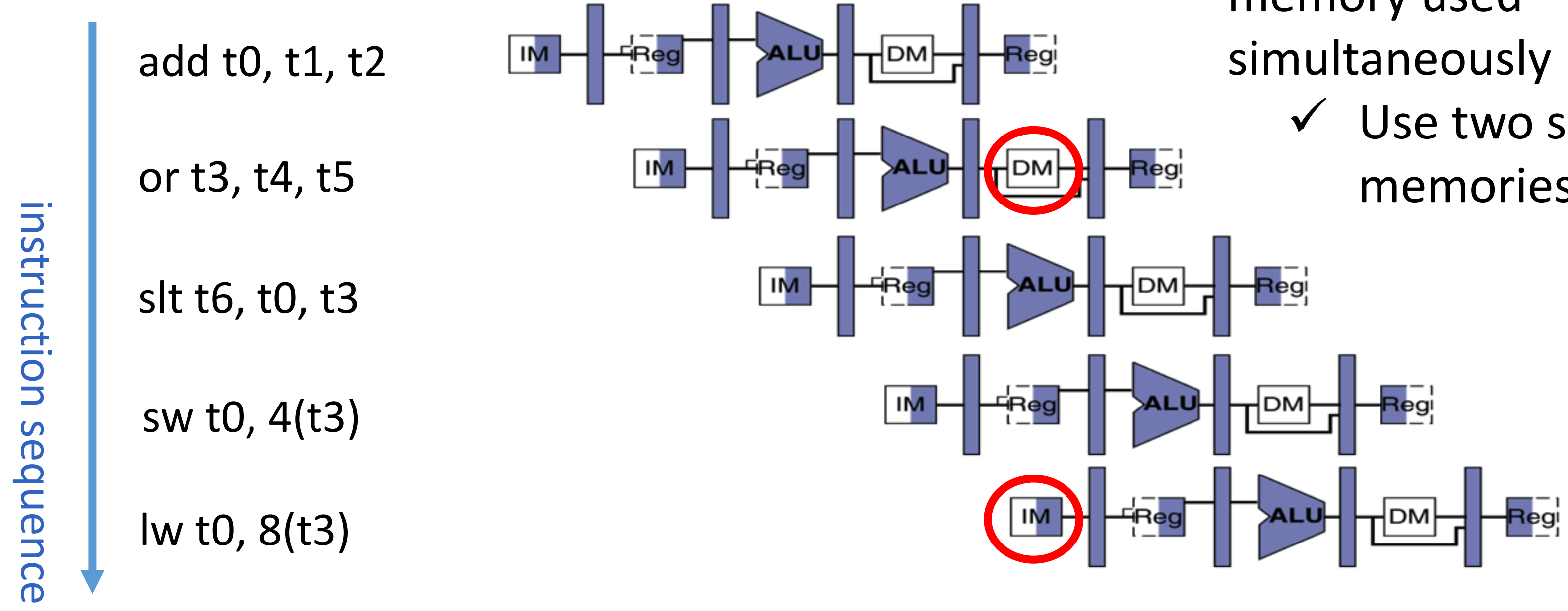
Structural Hazard

- **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

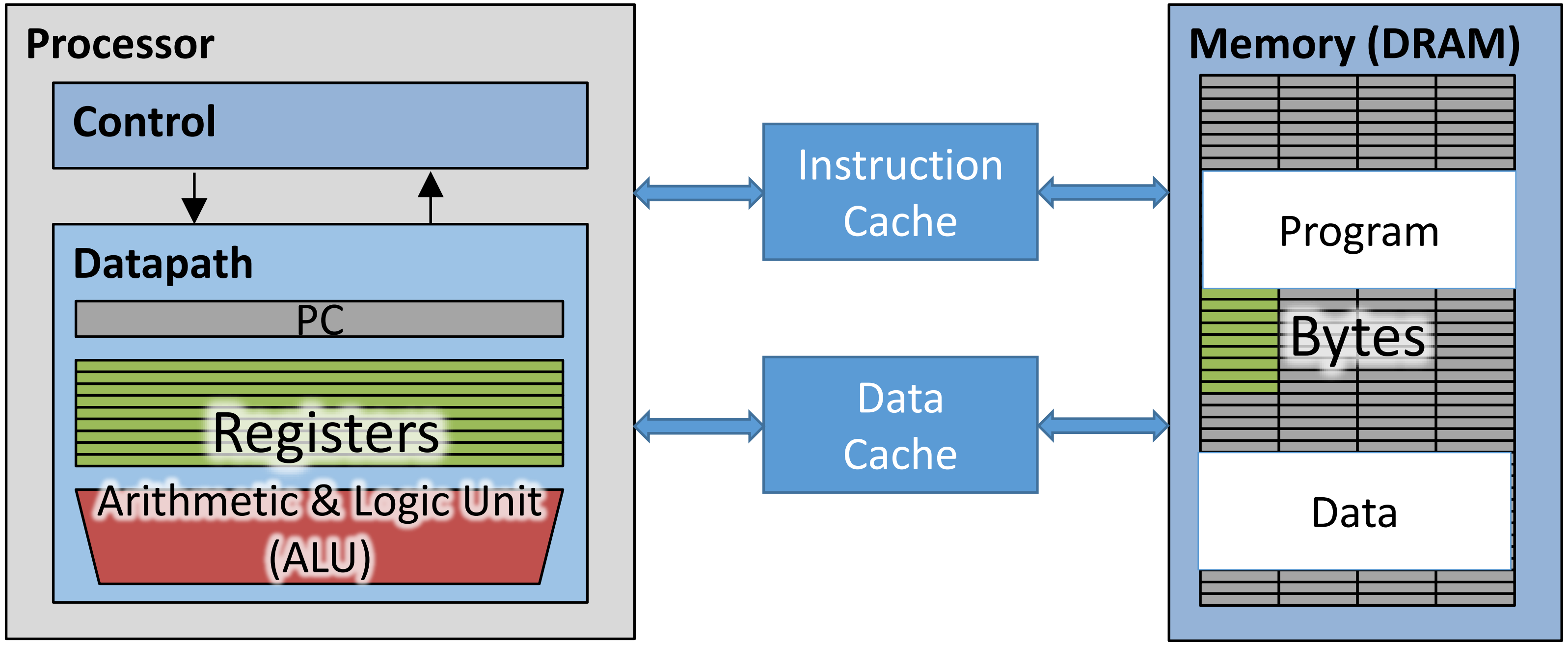
- Each instruction:
  - can read up to two operands in decode stage
  - can write one value in writeback stage
- Avoid structural hazard by having separate “ports”
  - two independent read ports and one independent write port
- Three accesses per cycle can happen simultaneously

Structural Hazard: Memory Access



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

Instruction and Data Caches



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 Hazard: Register Access

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

instruction sequence

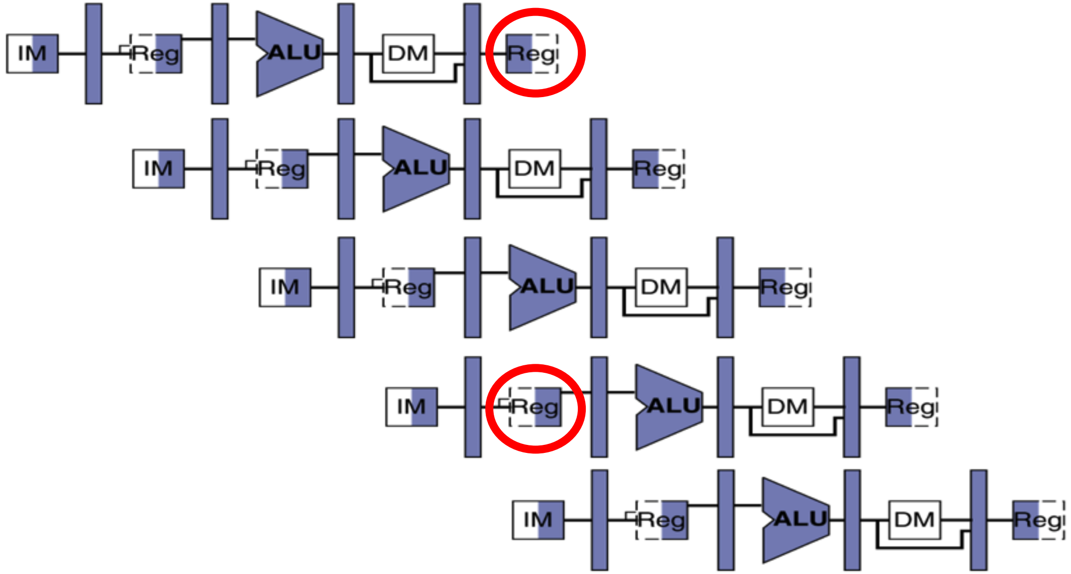
add t0, t1, t2

or t3, t4, t5

slt t6, t0, t3

sw t0, 4(t3)

lw t0, 8(t3)



Register Access Policy

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

instruction sequence

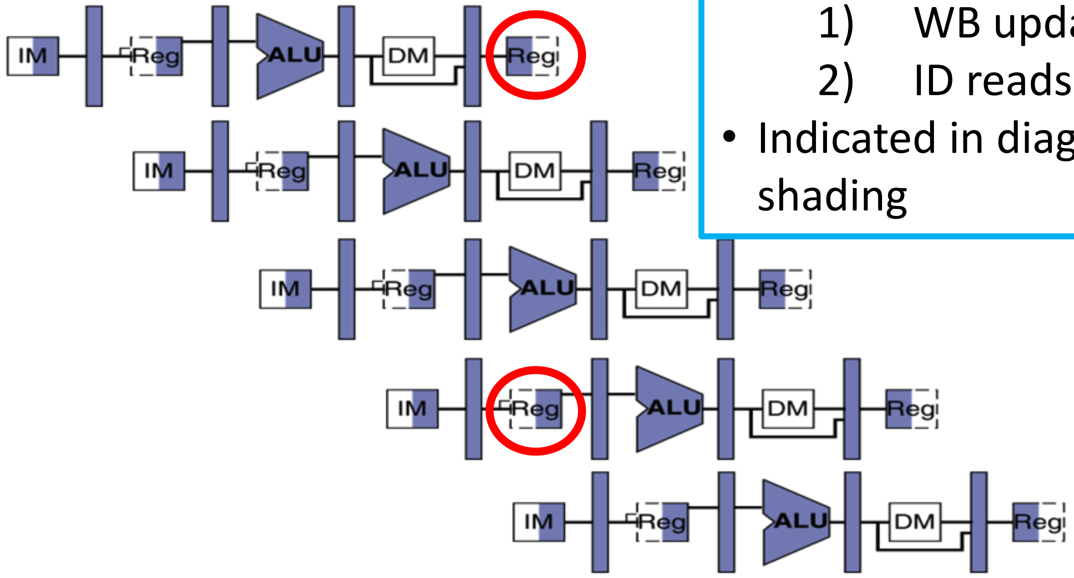
add t0, t1, t2

or t3, t4, t5

slt t6, t0, t3

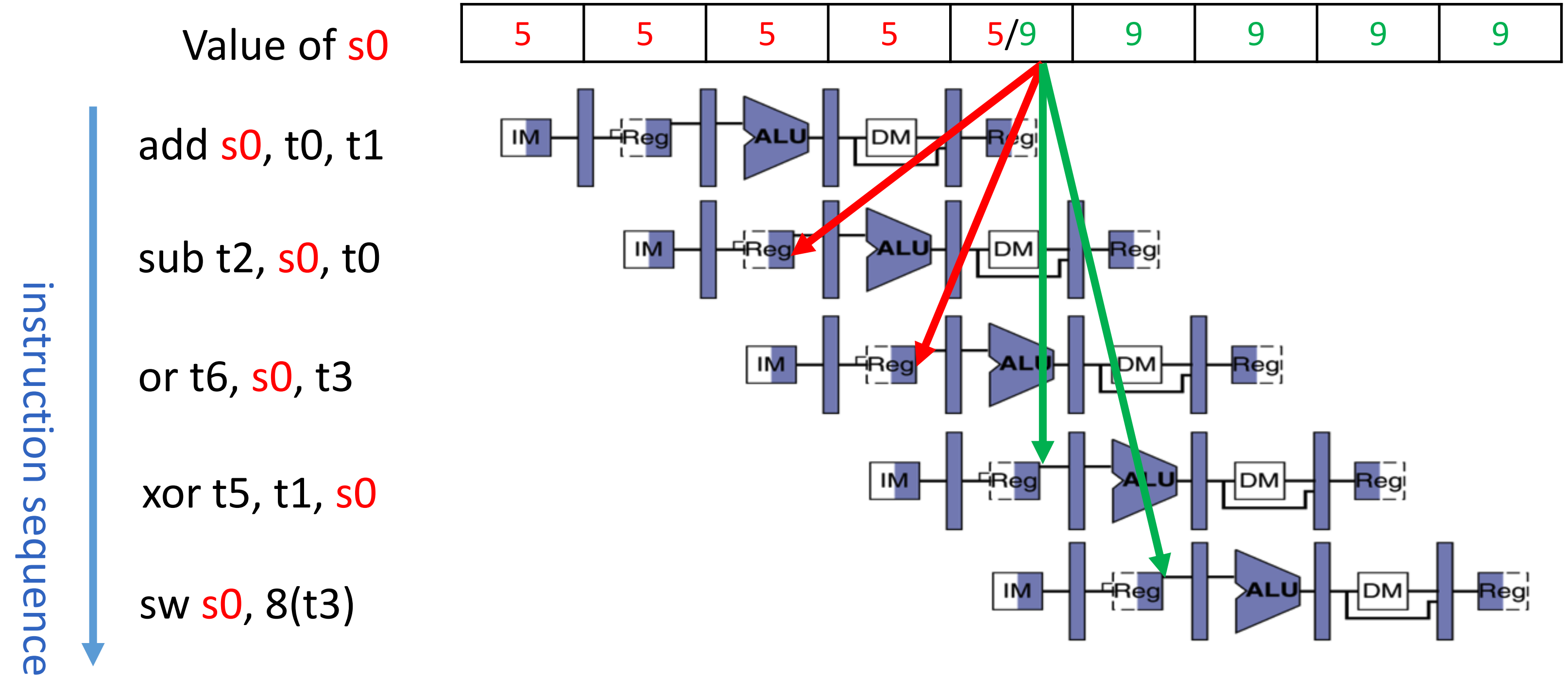
sw t0, 4(t3)

lw t0, 8(t3)



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

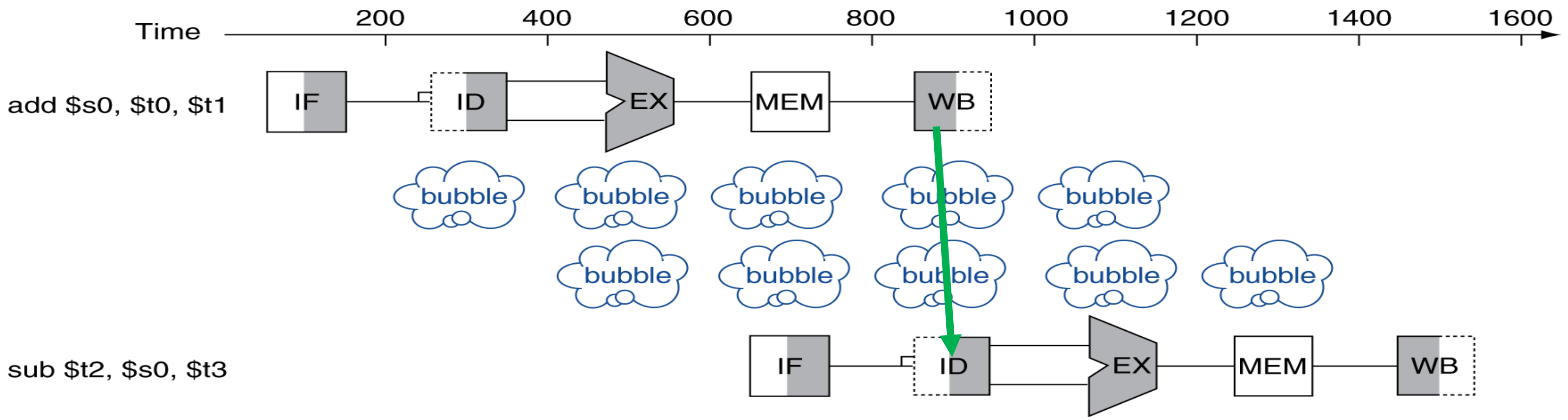
Data Hazard: ALU Result



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



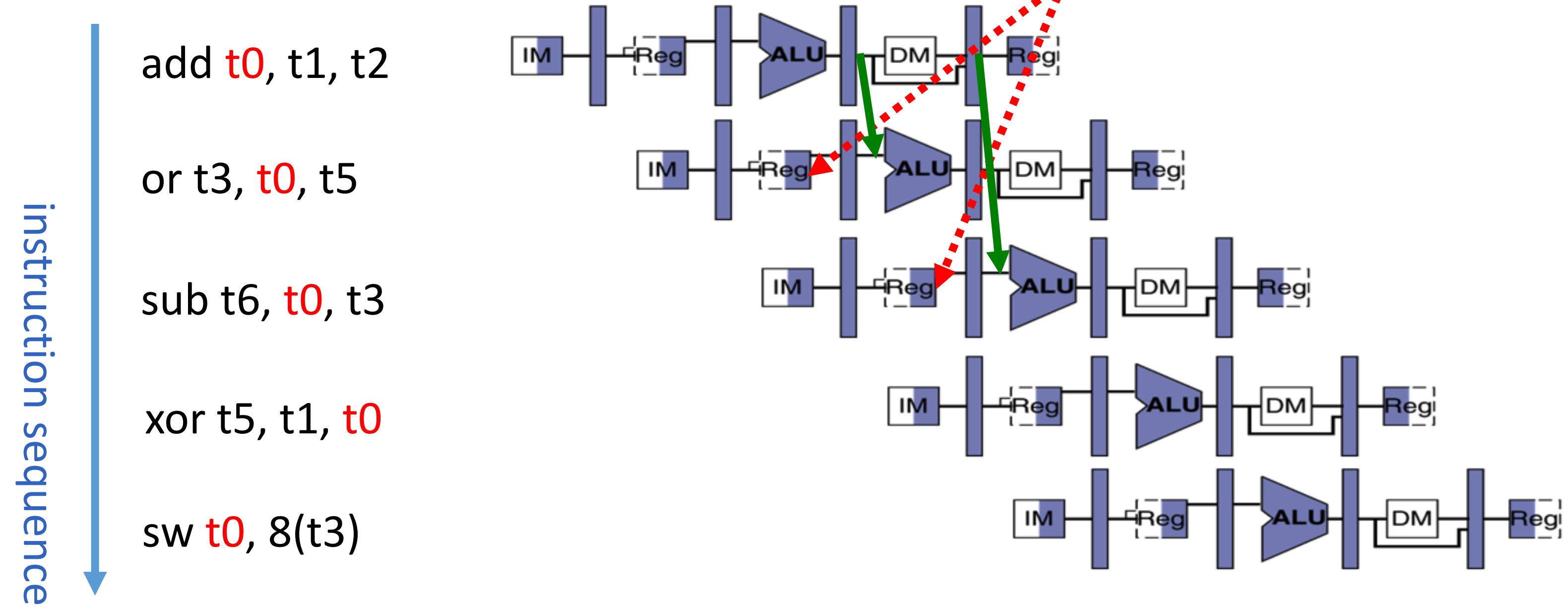
- Bubble:
  - effectively NOP: affected pipeline stages do “nothing”

Stalls and Performance

- Stalls reduce performance
  - But stalls are required to get correct results
- Compiler can arrange code or insert NOPs (writes to register x0) to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Solution 2: Forwarding

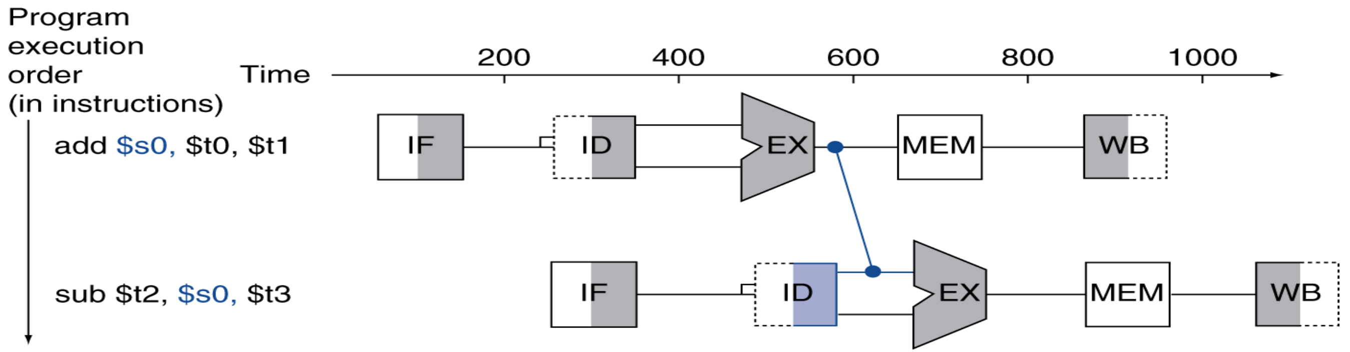
Value of t0	5	5	5	5	5/9	9	9	9	9



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



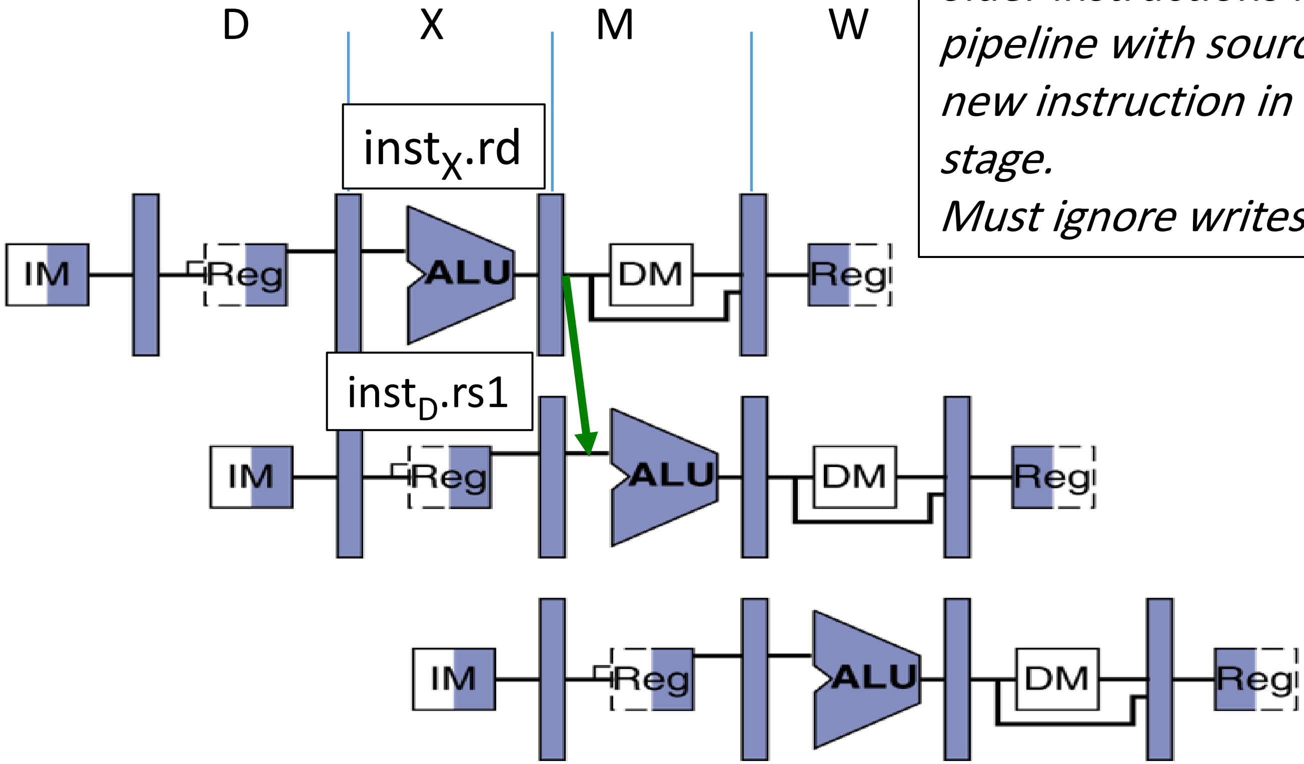
Detect Need for Forwarding (example)

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

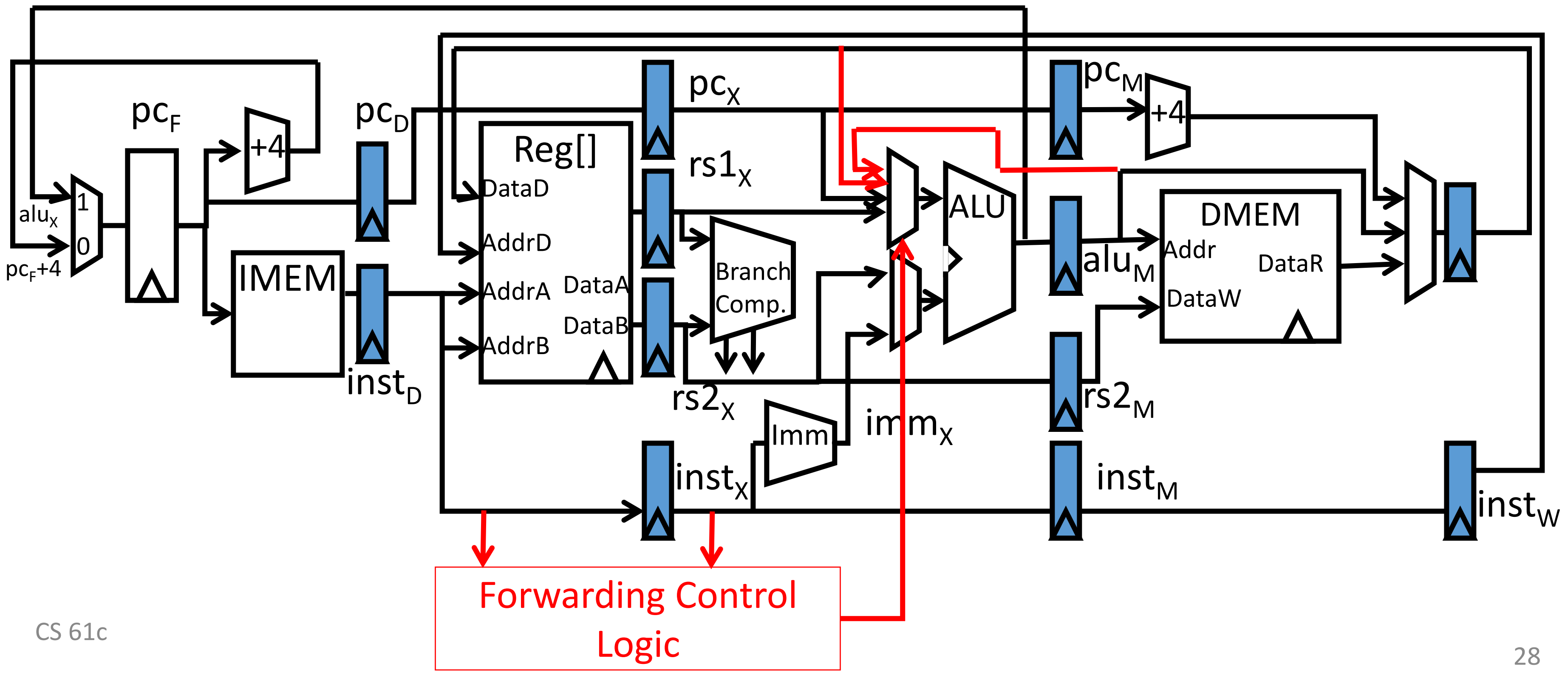
add t0, t1, t2

or t3, t0, t5

sub t6, t0, t3



Forwarding Path



PEER INSTRUCTION

Question: For each code sequences below, choose one of the statements below:

1:

```
addi $t1,$t0,1
addi $t2,$t0,2
addi $t3,$t0,2
addi $t3,$t0,4
addi $t5,$t1,5
```

2:

```
add $t1,$t0,$t0
addi $t2,$t0,5
addi $t4,$t1,5
```

3:

```
lw   $t0,0($t0)
add $t1,$t0,$t0
```

A) No stalls as is

B) No stalls with forwarding

C) Must stall

In Conclusion

- Pipelining increases throughput by overlapping execution of multiple instructions
- All pipeline stages have same duration
  - Choose partition that accommodates this constraint
- Hazards potentially limit performance
  - Maximizing performance requires programmer/compiler assistance
  - We've seen so far **Structure** & **Data** hazards so far…