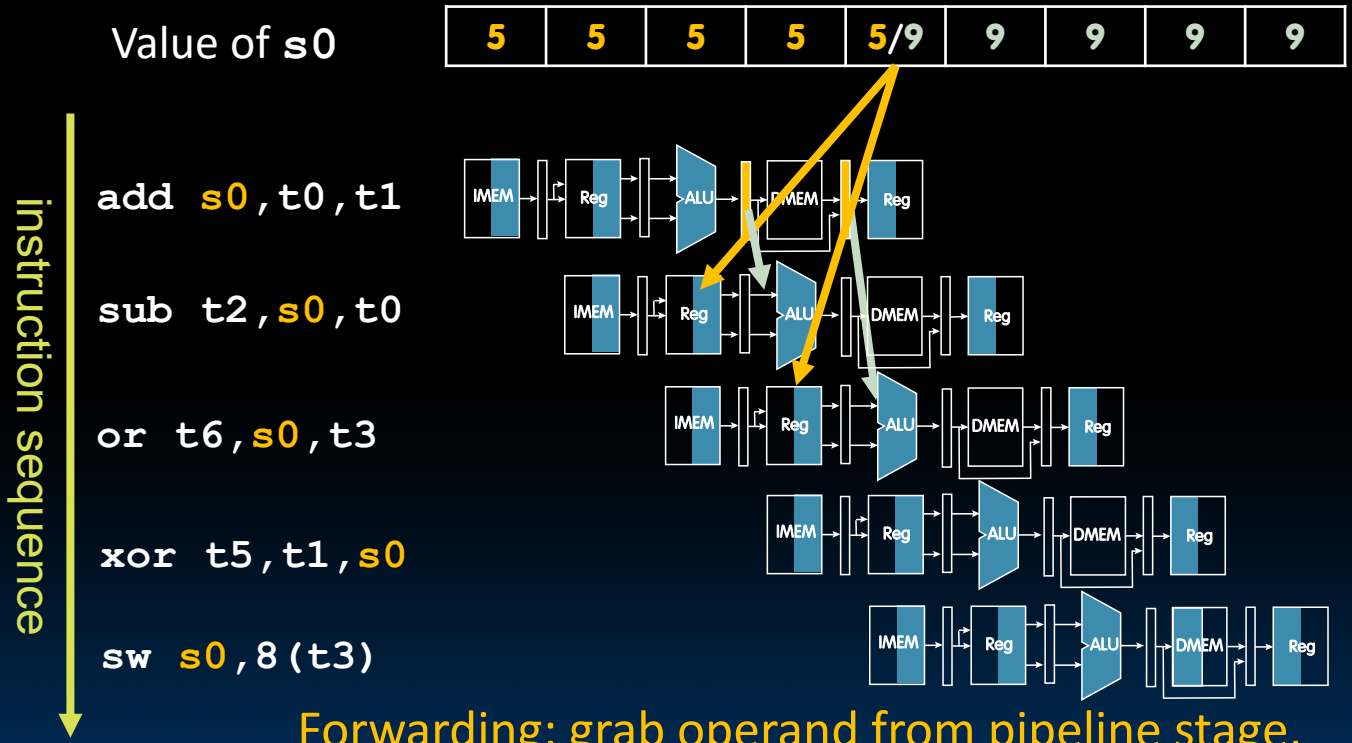
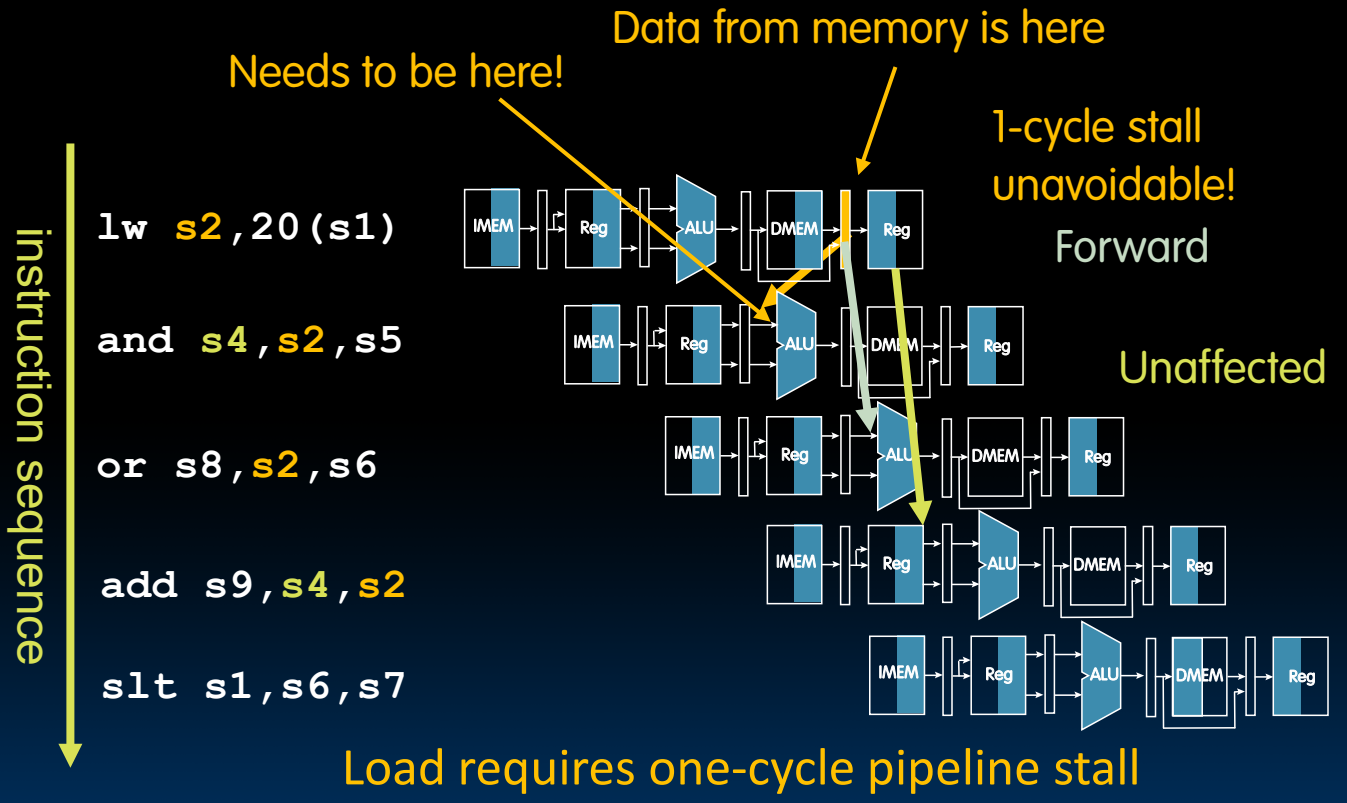
Load Data Hazard

Data Hazard and Forwarding



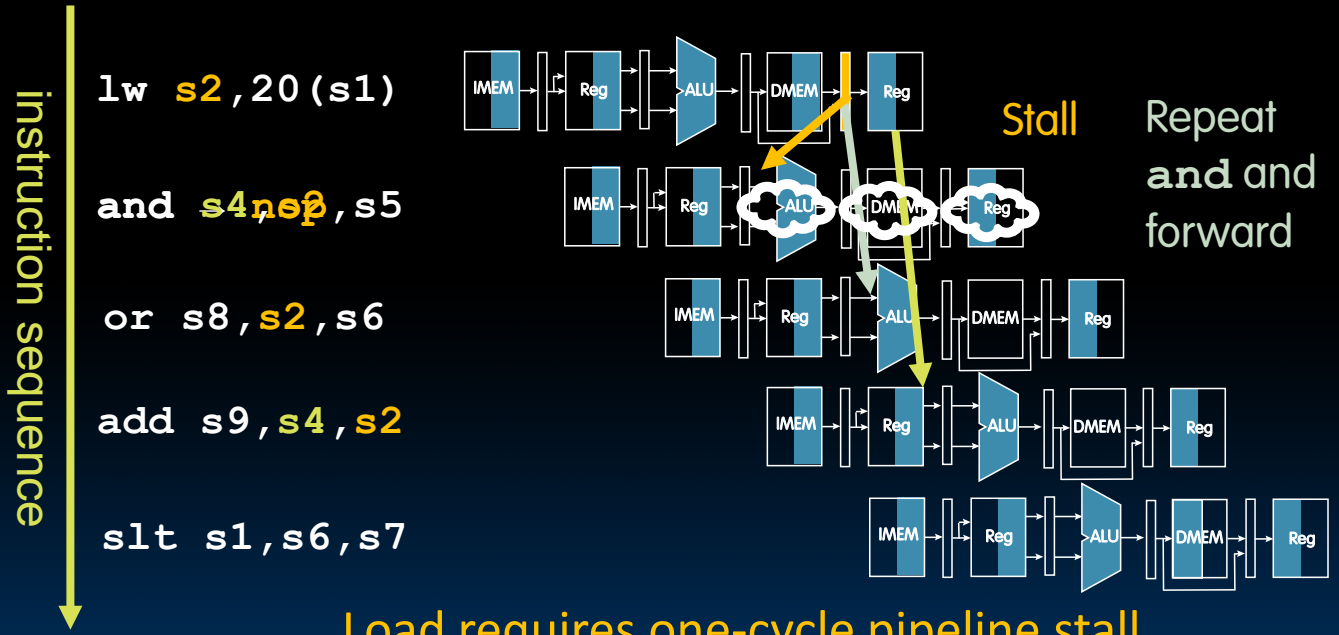
Forwarding: grab operand from pipeline stage, rather than register file

Load Data Hazard



Load requires one-cycle pipeline stall

Stall Pipeline



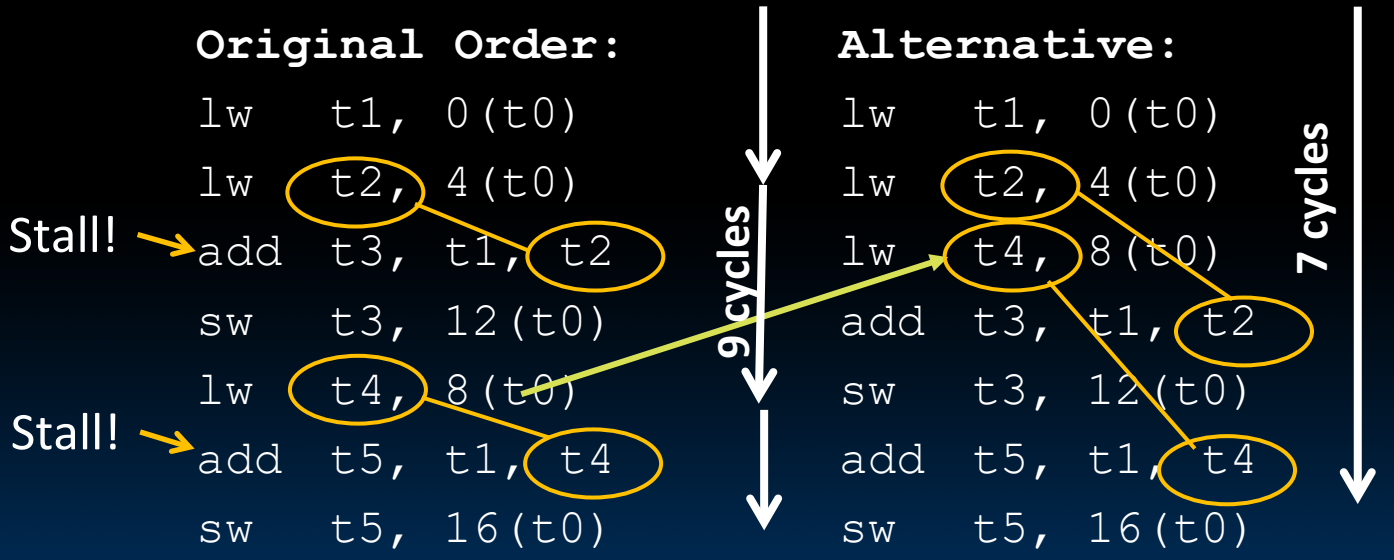
Load requires one-cycle pipeline stall

`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

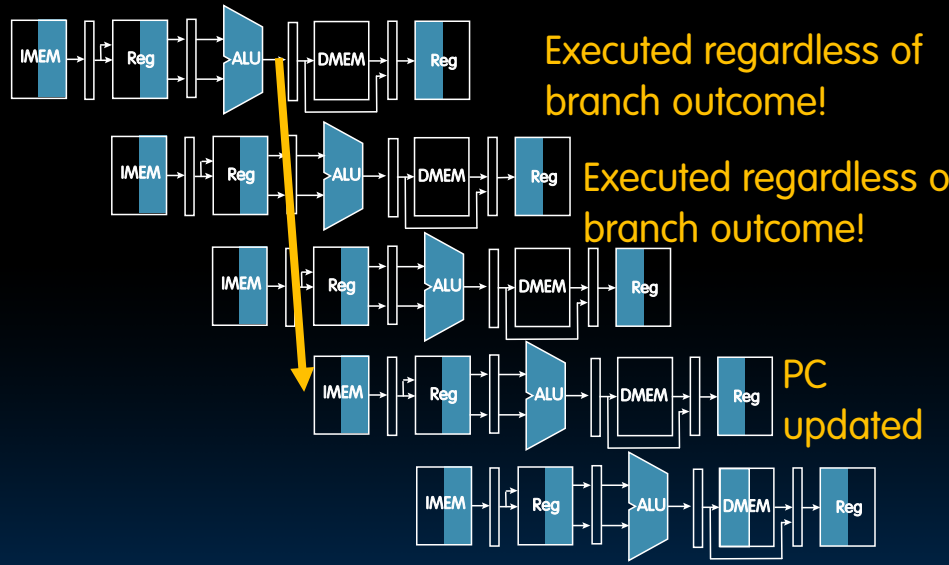
- Reorder code to avoid use of load result in the next instr!
- RISC-V code for **`A[3]=A[0]+A[1]; A[4]=A[0]+A[2]`**



Control Hazards

Control Hazards

```
beq t0,t1,Label
sub t2,s0,t0
or t6,s0,t3
xor t5,t1,s0
sw s0,8(t3)
```



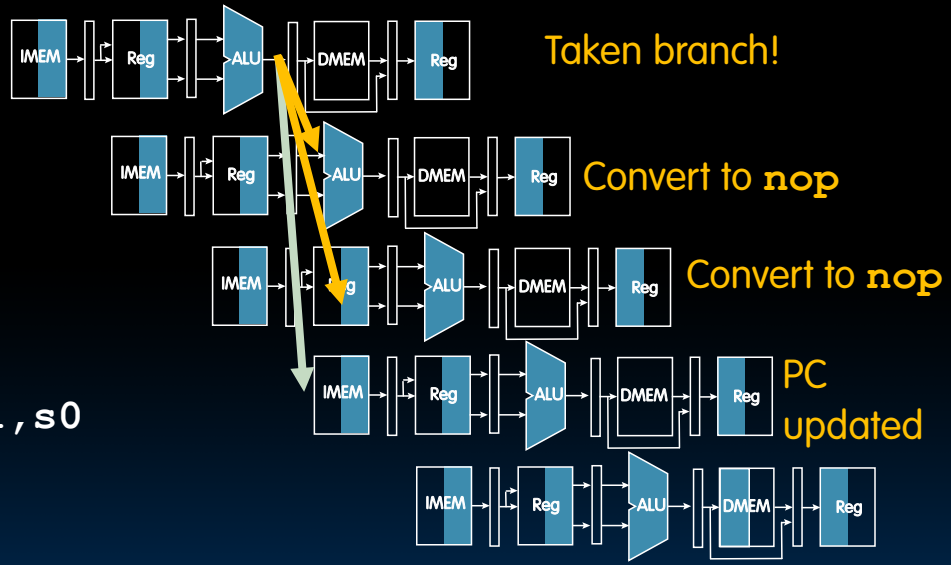
Two stall cycles after a branch!

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,s0
```



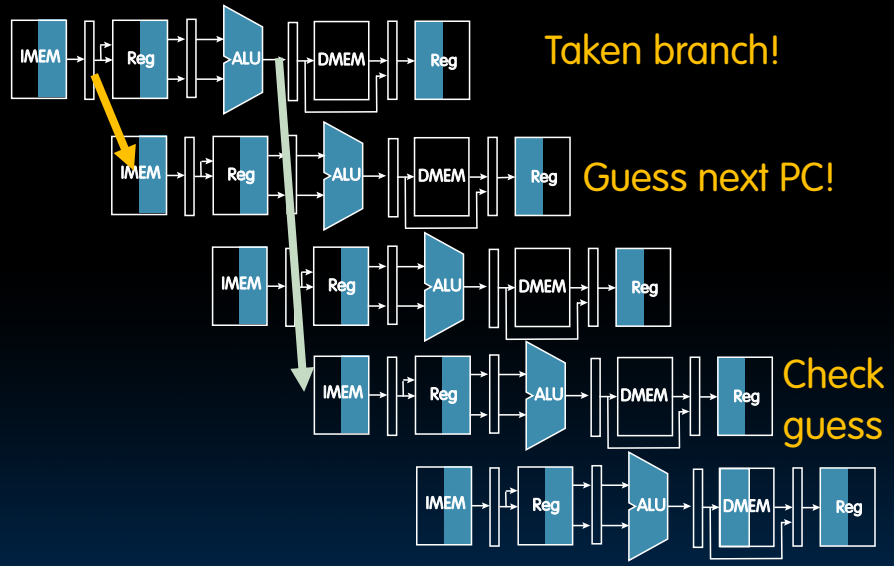
Reducing Branch Penalties

- 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

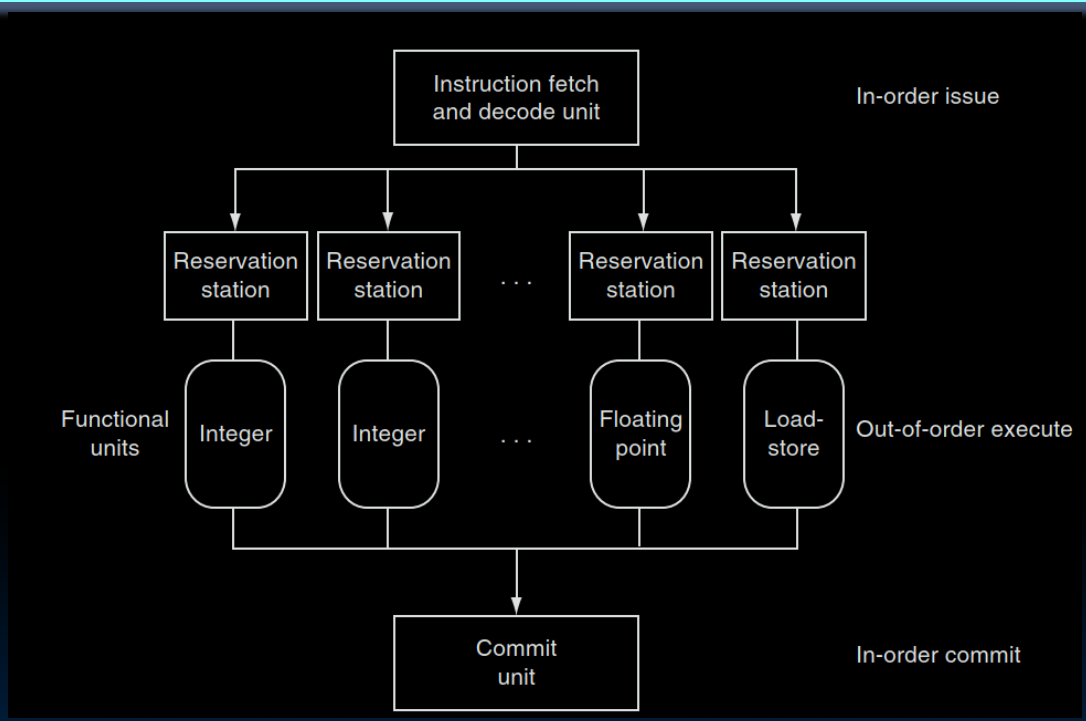
Increasing Processor Performance

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

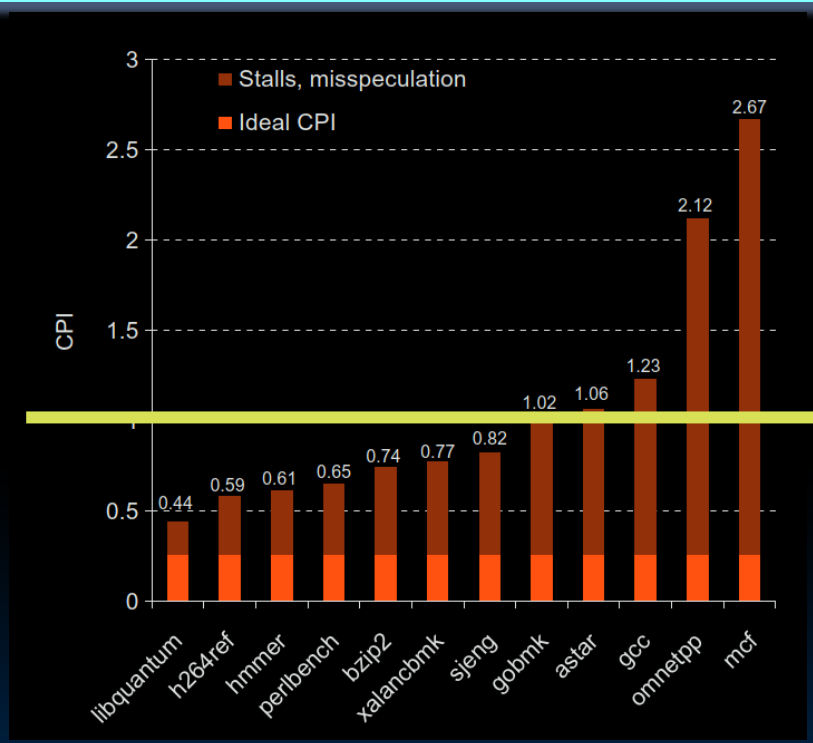
- Multiple issue "superscalar"
  - Replicate pipeline stages ⇒ multiple pipelines
  - Start multiple instructions per clock cycle
  - CPI < 1, so use Instructions Per Cycle (IPC)
  - E.g., 4GHz 4-way multiple-issue
    - 16 BIPS, peak CPI = 0.25, peak IPC = 4
  - Dependencies reduce this in practice
- "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



CPI = 1

P&H, p.330

"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