CS465: Computer Systems Architecture

Lecture 7: Processor II – Pipelined Processor Design

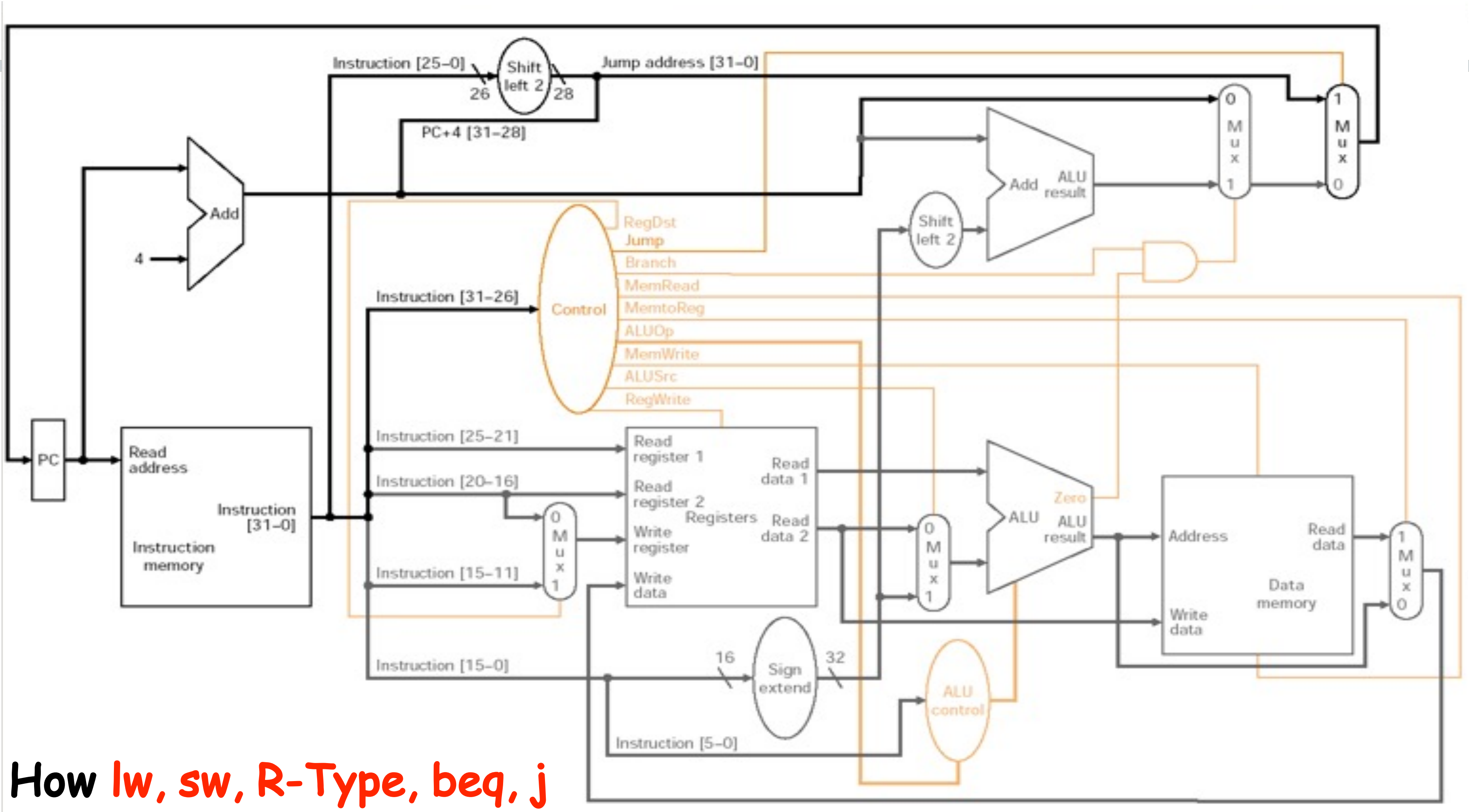
*Slides adapted from Computer Organization and Design by Patterson and Henessey

Review: Single Cycle Processor

- Subset of the core MIPS ISA
  - Arithmetic/Logic instructions: AND, OR, ADD, SUB, SLT
  - Data flow instructions: LW, SW
  - Branch instructions: BEQ, J

- Five steps in processor design
  - Analyze the instruction
  - Determine the datapath components
  - Assemble the components
  - Determine the control
  - Design the control unit

Complete Single Cycle Processor

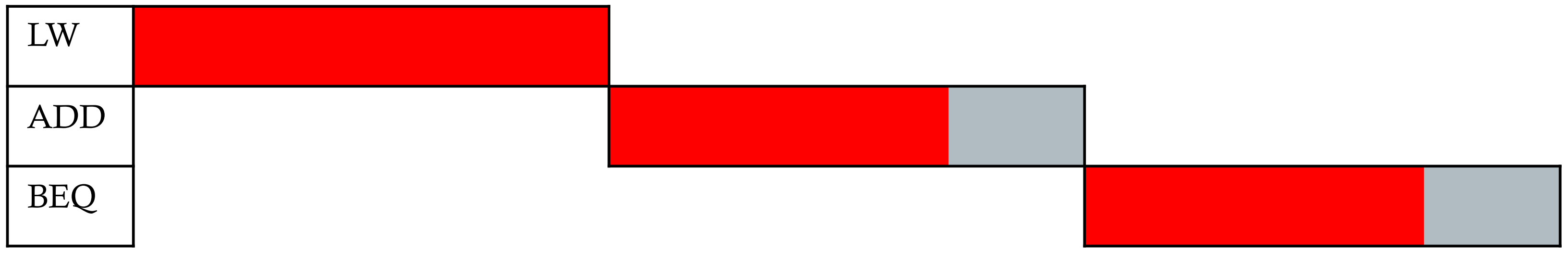


How lw, sw, R-Type, beq, j instructions work?

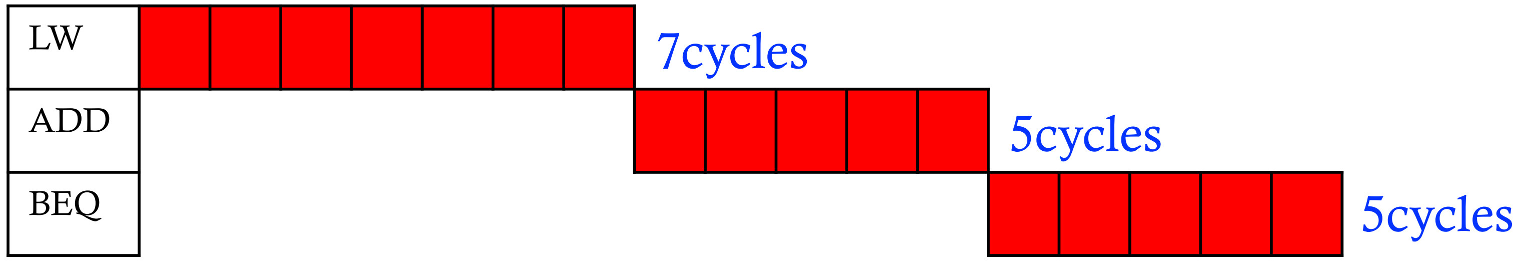
Remarks on Single Cycle Datapath

- Single cycle datapath ensures the execution of any instruction within one clock cycle
  - Functional units must be duplicated if used multiple times by one instruction, e.g. ALU/Adder
  - Functional units can be shared if used by different instructions
- Single cycle datapath is not efficient in time
  - Clock cycle time is determined by the instruction taking the longest time, eg. lw in MIPS
  - Variable clock cycle time is too complicated
- Alternative design/implementation approaches
  - Multiple clock cycles per instruction
  - Pipelining

Different Processor Designs

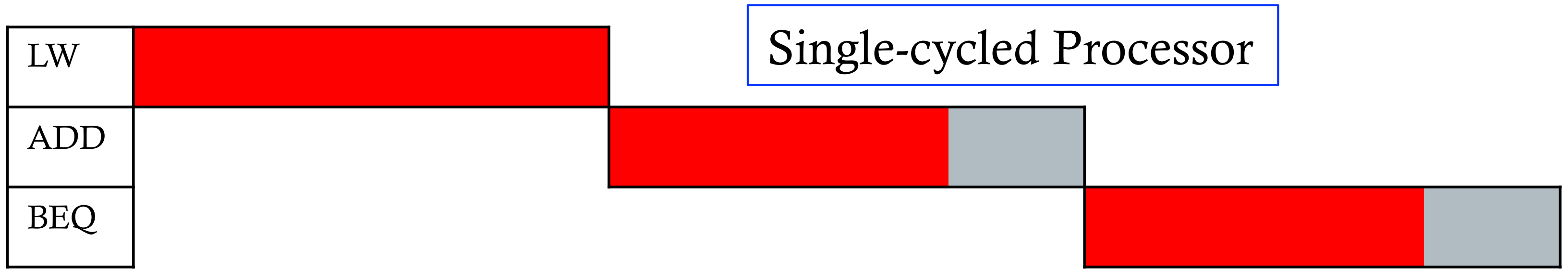


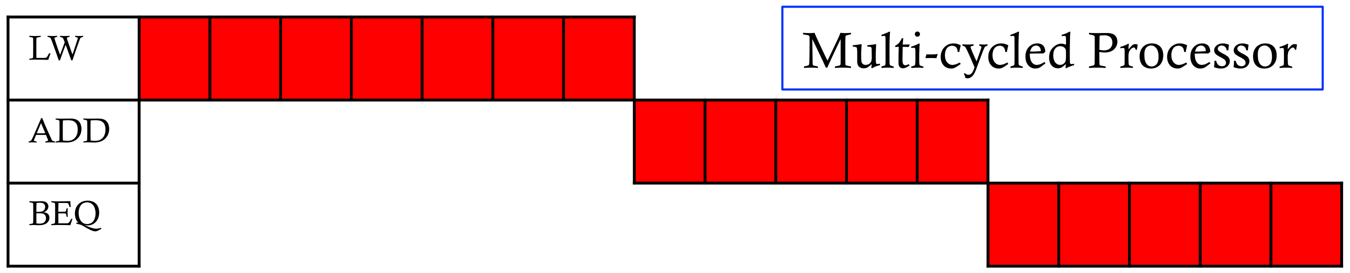
Single-cycled Processor (cycle time = 7ns)

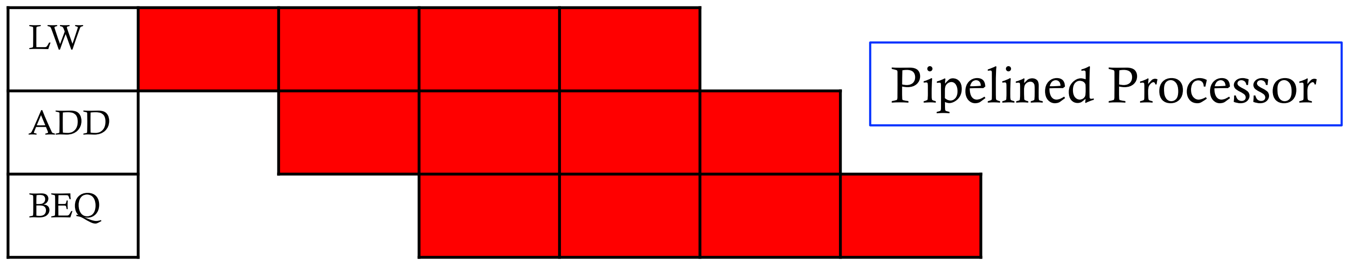


Multi-cycled Processor (cycle time = 1ns)

Different Processor Designs





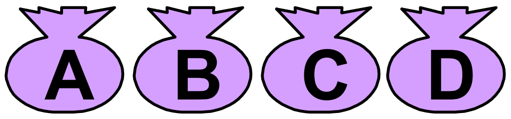


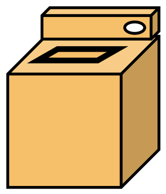
Pipeline

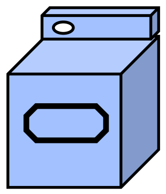
- Pipelining is an implementation technique in which multiple instructions are overlapped in execution
- Outline
  - High-level pipeline introduction
    - Stages, hazards
  - Pipelined datapath and control design
  - Pipelined execution for subset of MIPS instructions
    - lw, sw, R-type(and, or, add, sub, slt), beq

Pipelining is Natural!

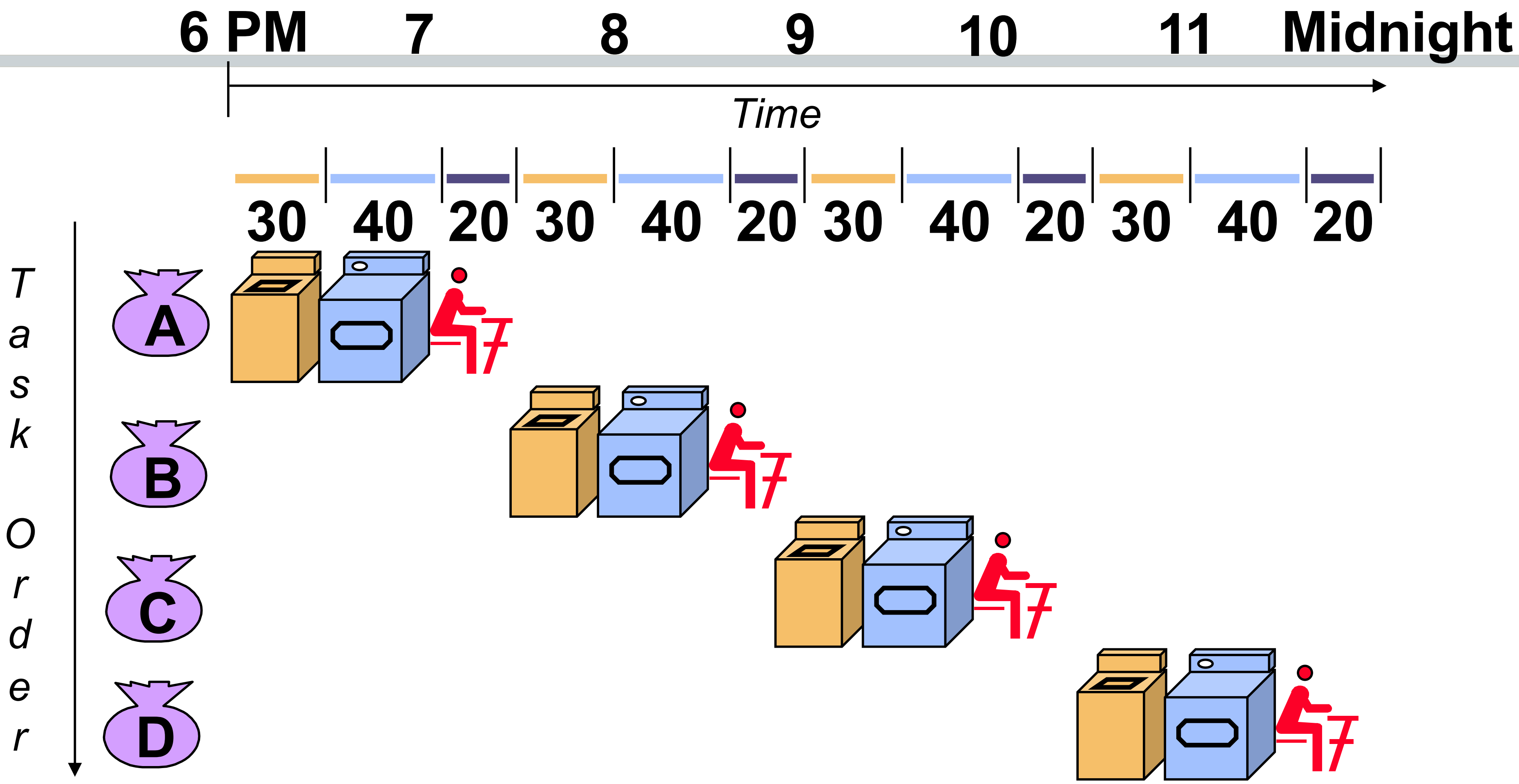
- Laundry example
  - Ann, Brian, Cathy, Dave each has one load of clothes to wash, dry, and fold
  - Washer takes 30 minutes
  - Dryer takes 40 minutes
  - "Folder" takes 20 minutes





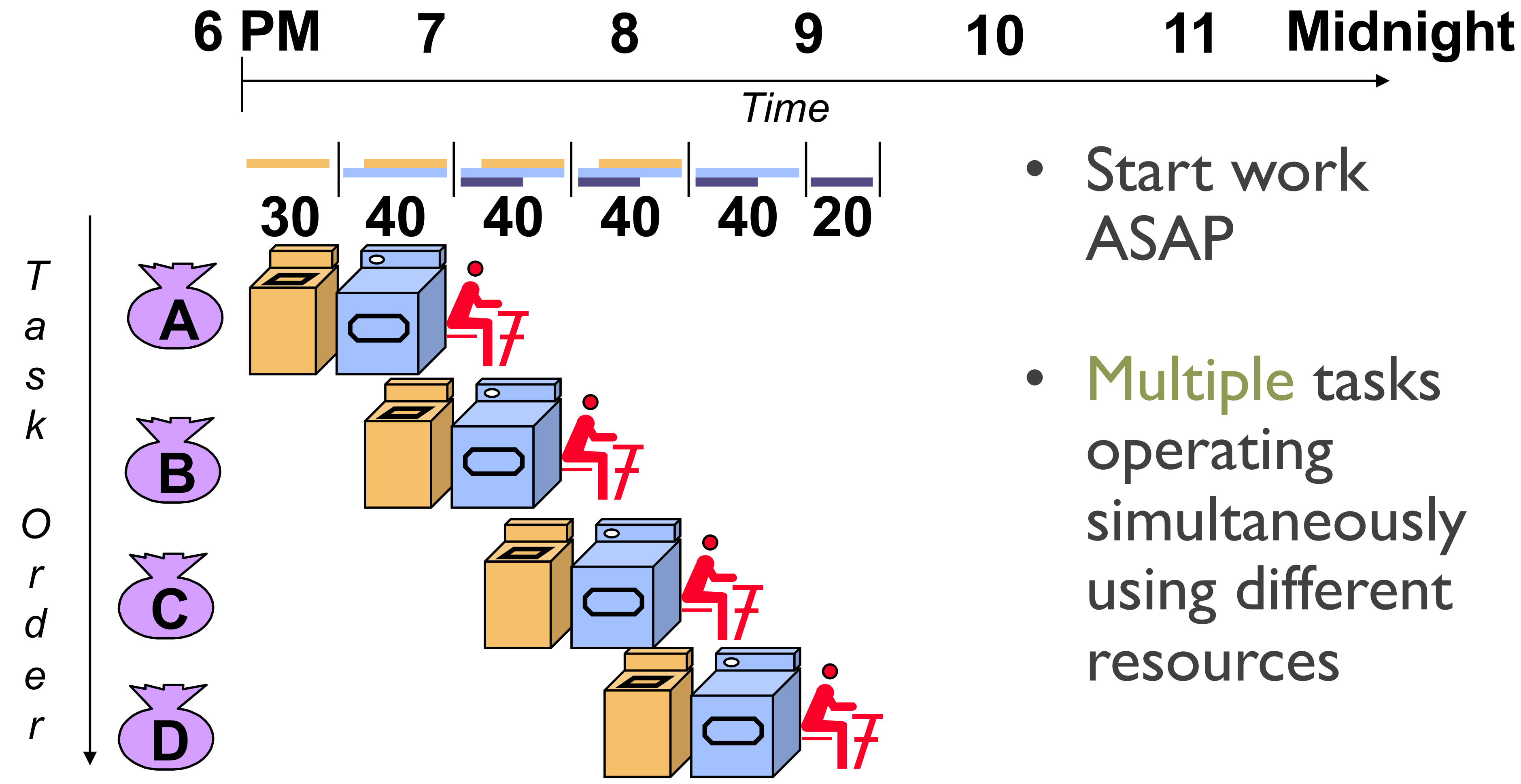


Sequential Laundry



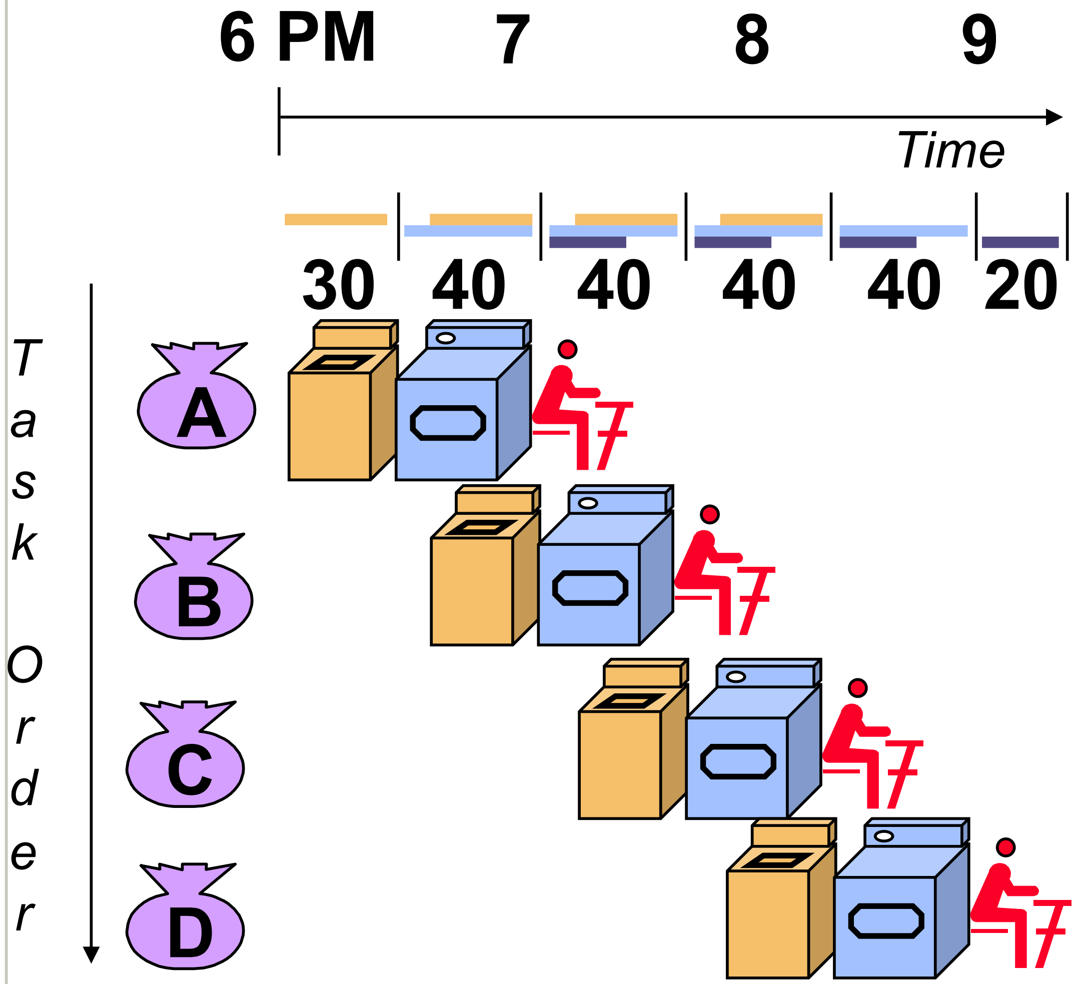
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would it take?

Pipelined Laundry



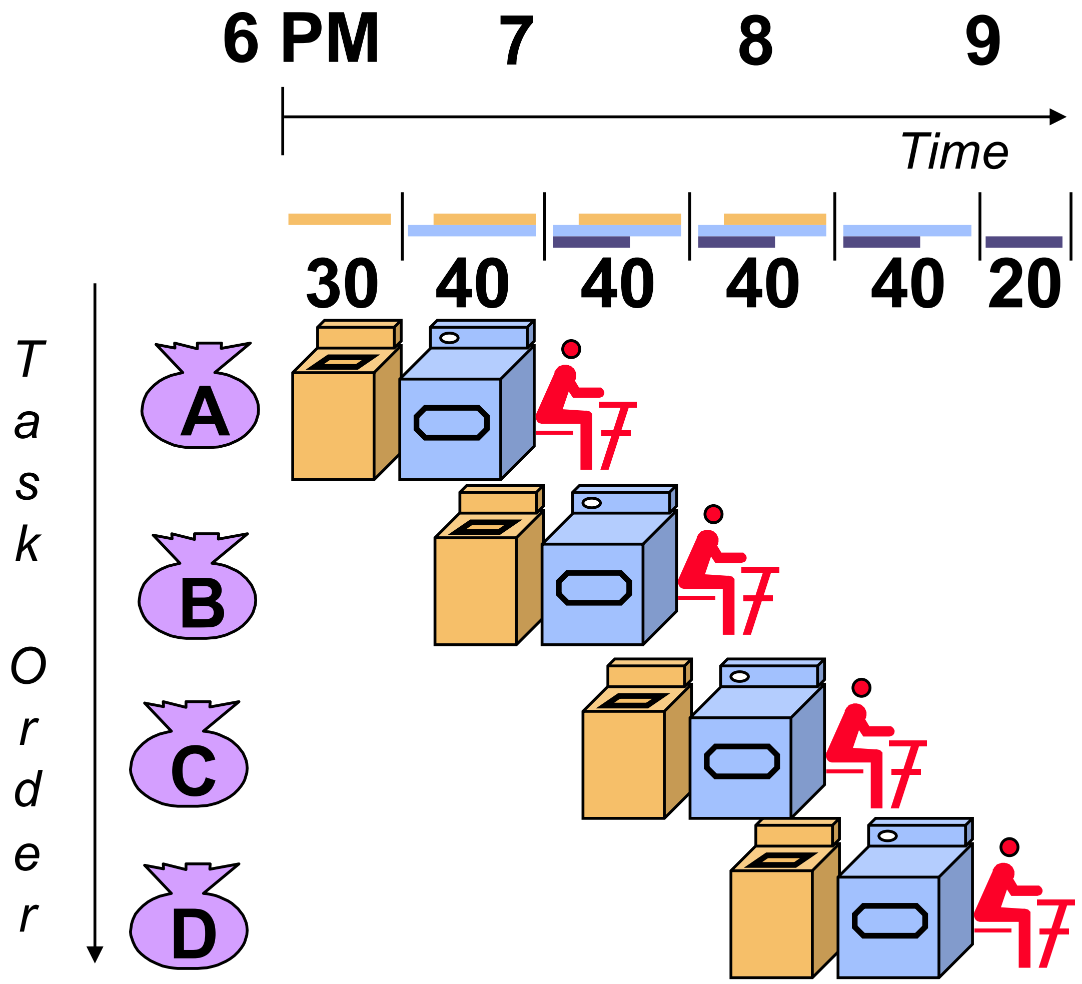
- Start work ASAP
- Multiple tasks operating simultaneously using different resources

Pipelining Lessons (I)



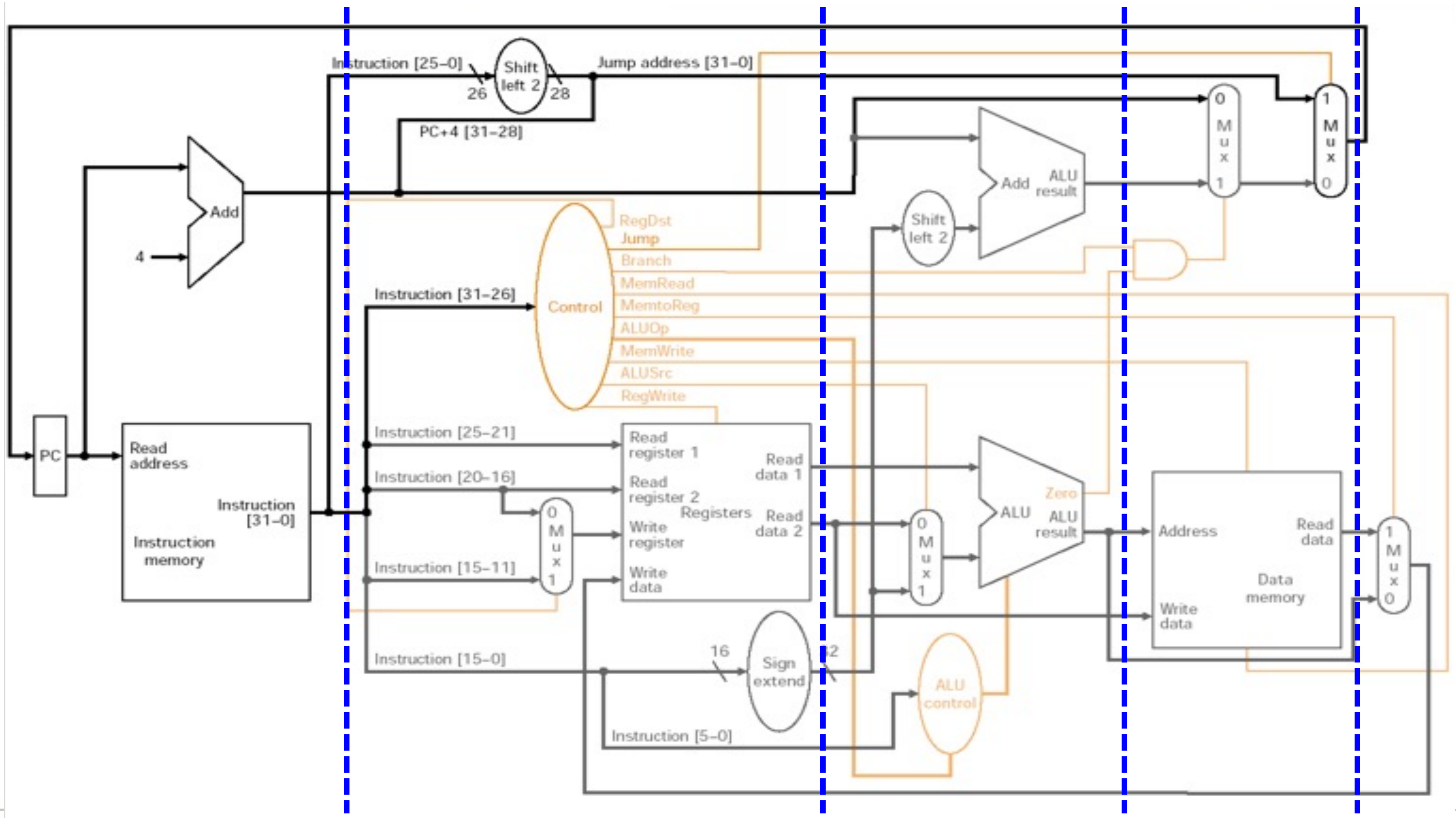
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
  - Pipelined laundry takes 3.5 hours for 4 loads
- Potential speedup = Number pipeline stages (hard to achieve)

Pipelining Lessons (II)



- Pipeline rate is limited by slowest pipeline stage
  - Unbalanced lengths of pipeline stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup- startup and wind down
- Stall for other restrictions (hazards)

Single Cycle Datapath



MIPS Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

Instr	IF	ID	EX	MEM	WB
lw	X	X	X	X	X
sw	X	X	X	X	
R-format	X	X	X		X
beq	X	X	X		

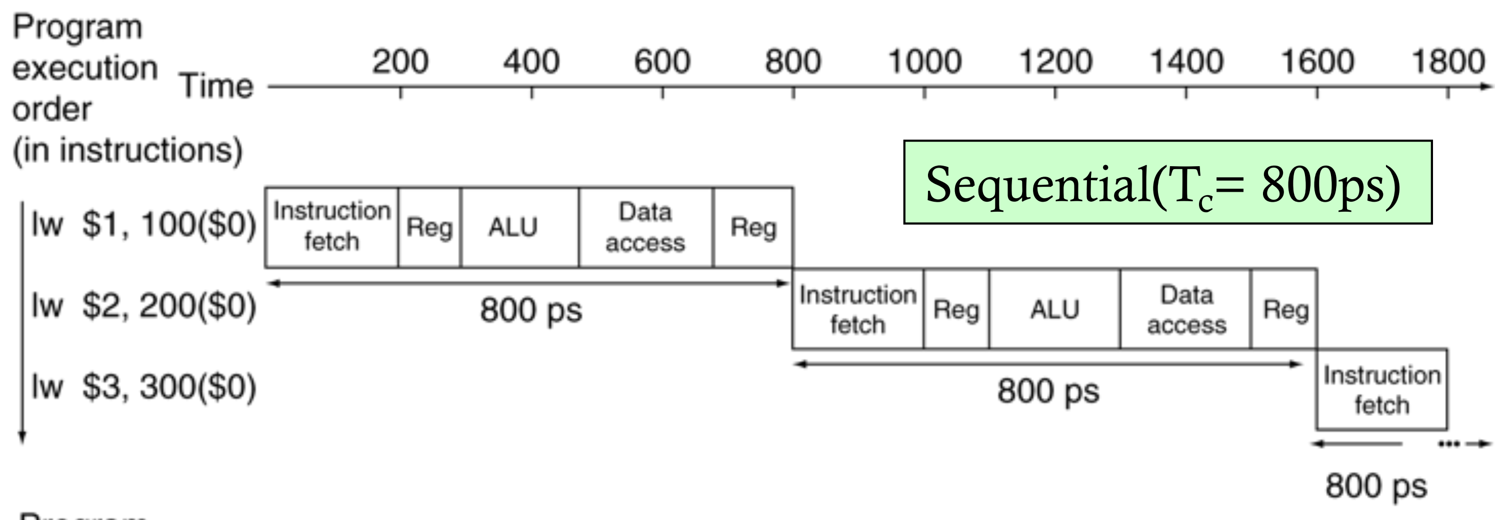
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages

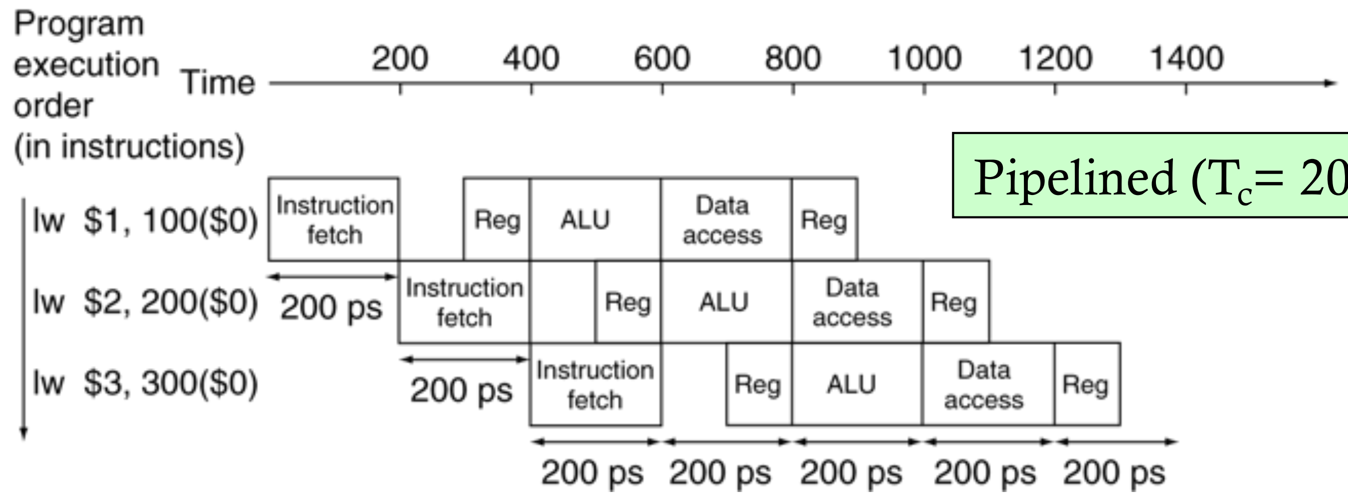
- Latency

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance



Sequential(T_c= 800ps)



Pipelined (T_c= 200ps)

Pipeline Speedup

- If all stages are balanced, i.e., all take the same time
  - $$\frac{\text{Time between instructions}_{\text{non-pipelined}}}{\text{Time between instructions}_{\text{pipelined}}} = \text{Number of stages}$$
  - Ideal speedup = Number of stages
  - If stages not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

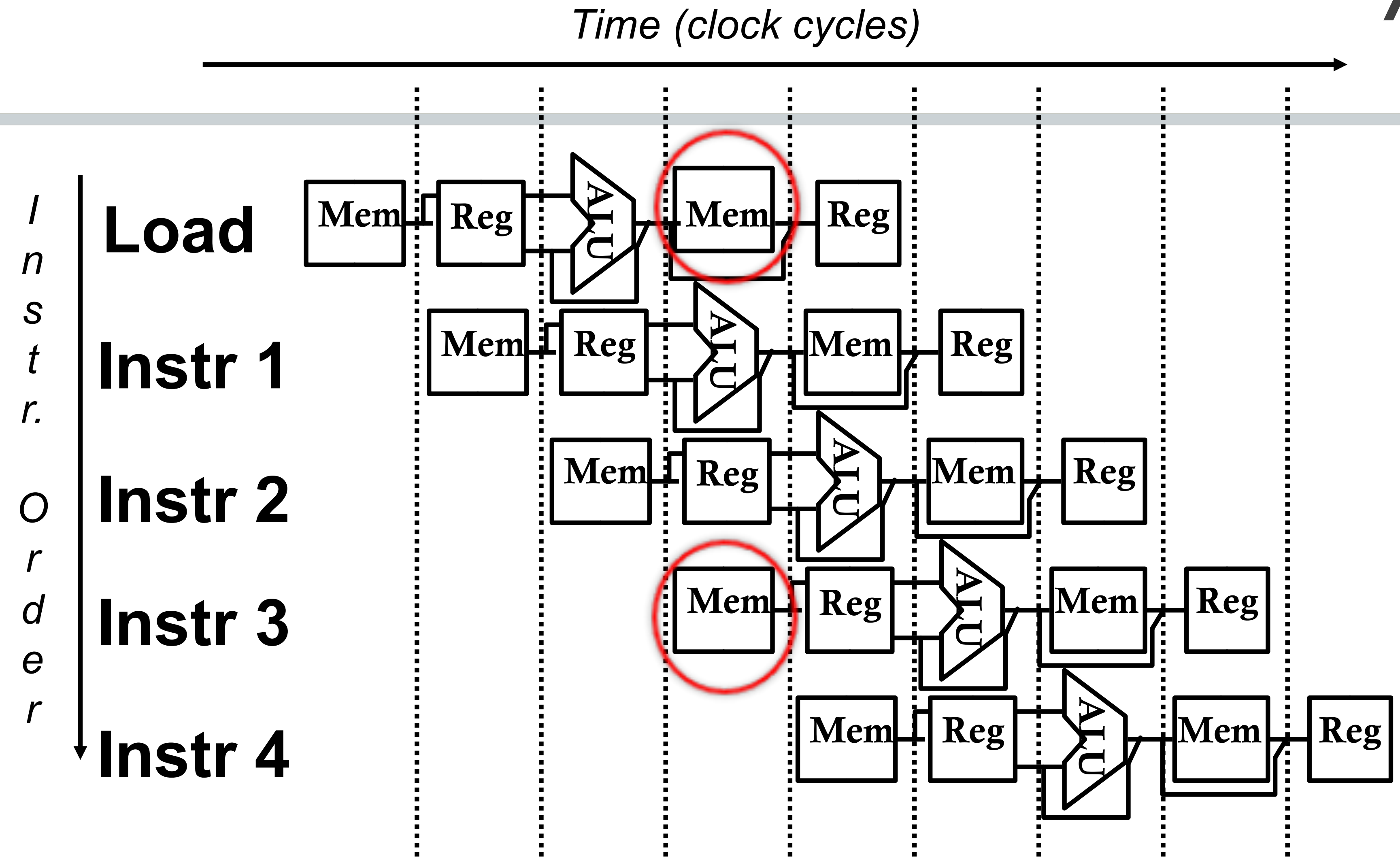
Pipelining and ISA Design

- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch or decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

Hazards

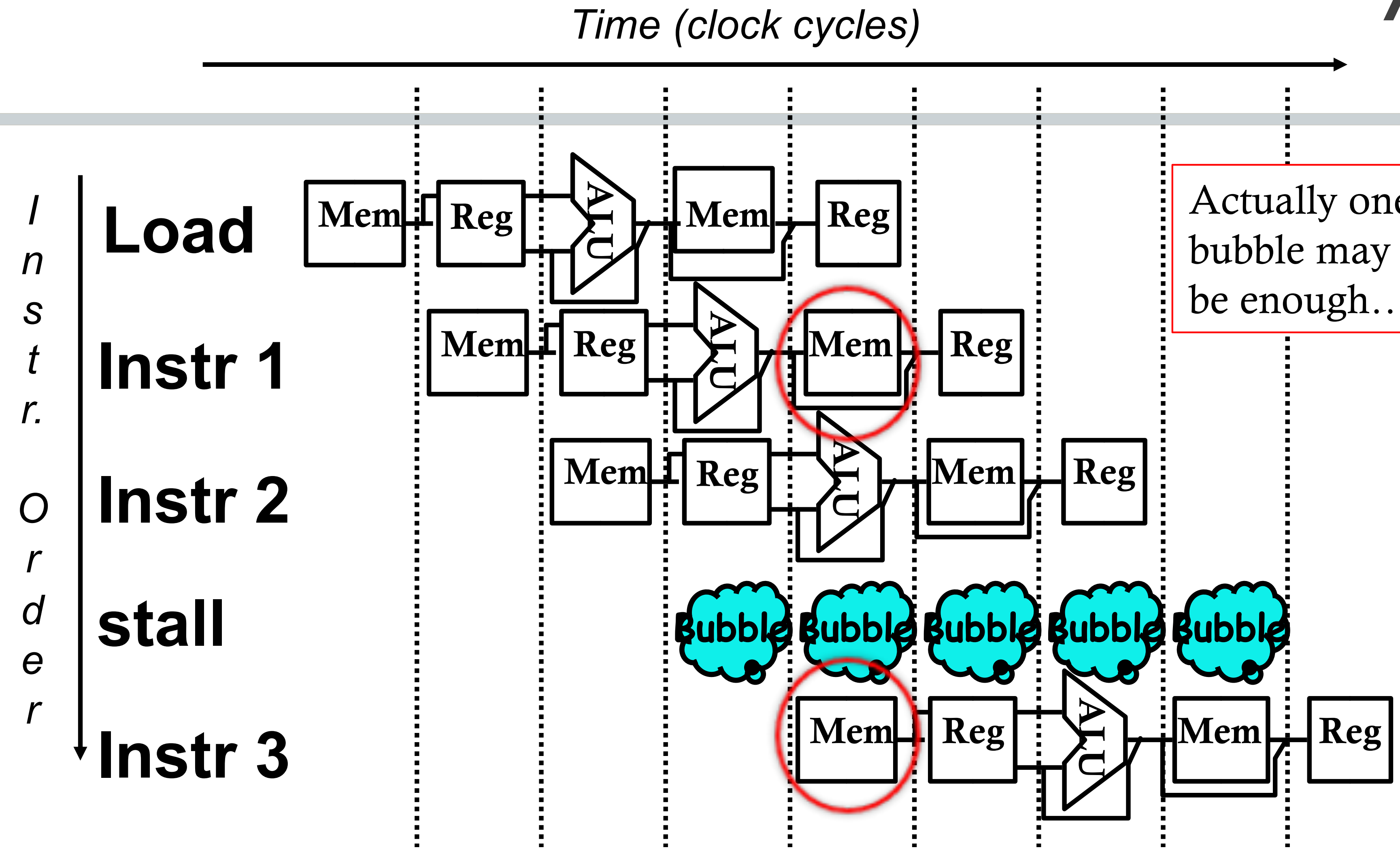
- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazards
  - Need to wait for previous instruction to complete its data read/write
- Control hazards
  - Deciding on control action depends on previous instruction

Structural Hazard: One Memory



- Resource conflicts can always be resolved by waiting
- Instruction fetch would have to stall for that cycle

Structural Hazard: One Memory



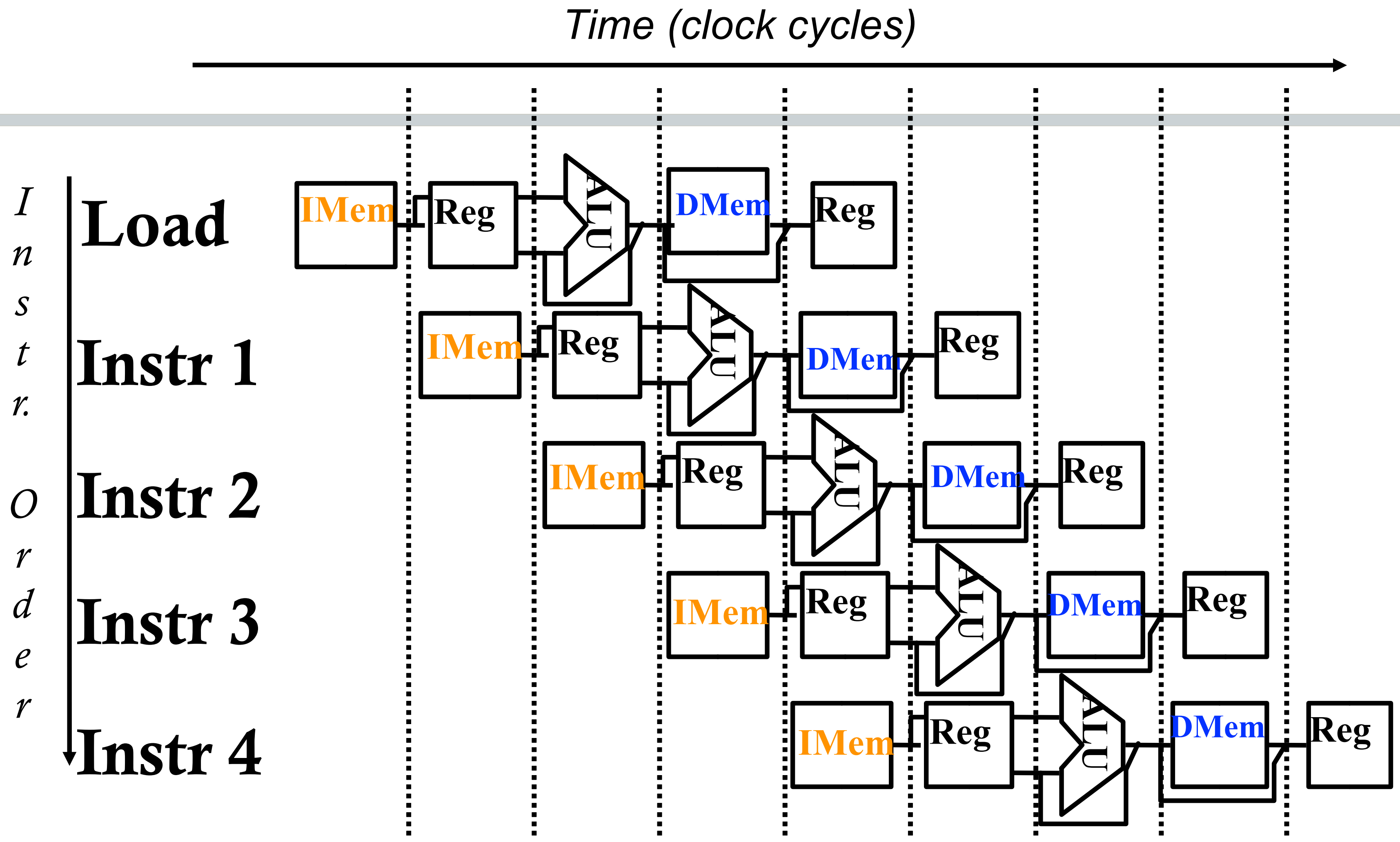
Actually one bubble may not be enough…

•Instruction fetch would have to stall for that cycle: would cause a pipeline "bubble"

Structure Hazards

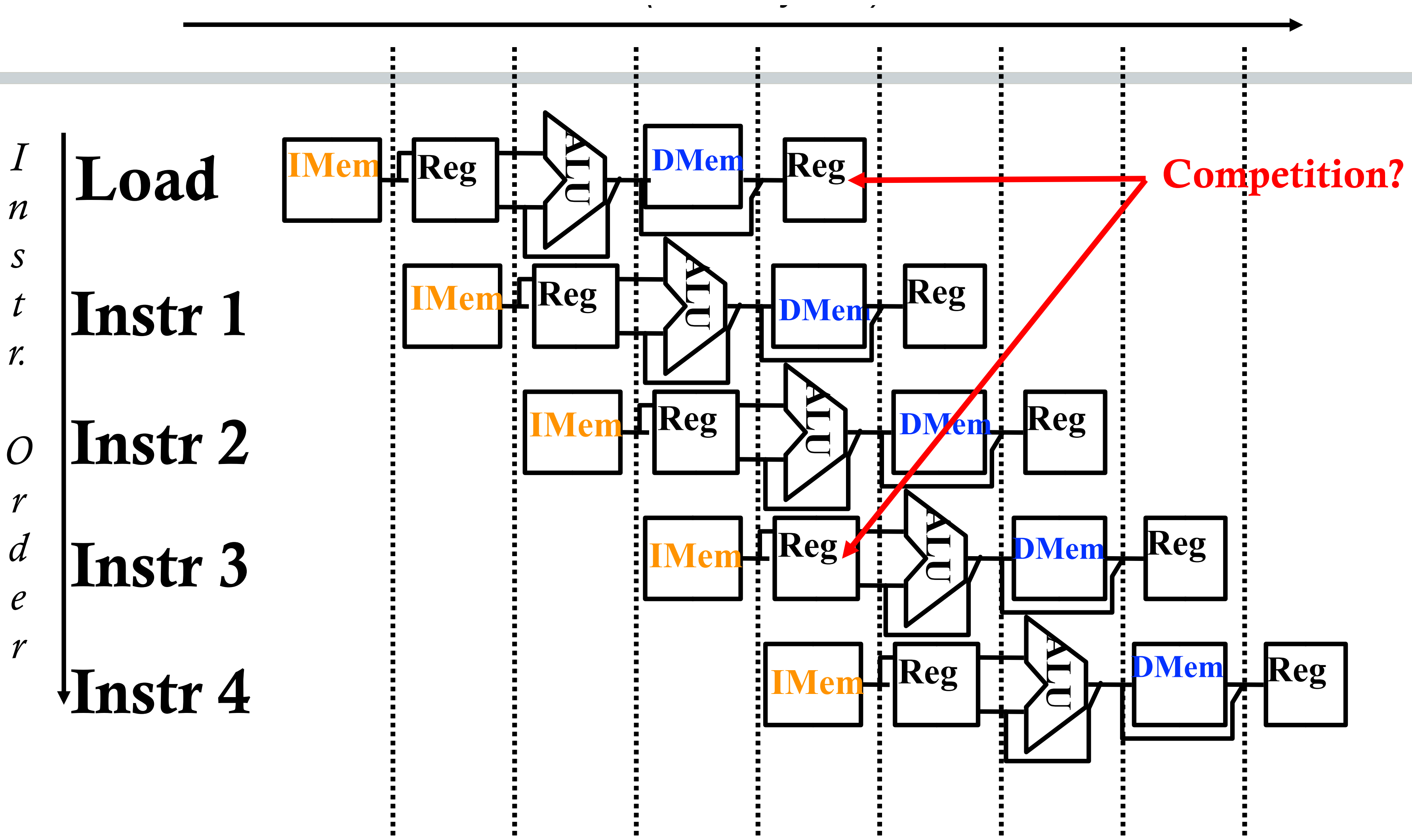
- Conflict for use of a resource
  - E.g. MIPS pipeline with a single memory
    - Load/store requires data access
    - Instruction fetch would have to *stall* for that cycle
      - Would cause a pipeline "bubble"
- Or, we can add more resources to deal with structure hazards
  - E.g. pipelined datapaths include separate instruction/data memories
    - Or separate instruction/data caches

Structural Hazard: Two Memories



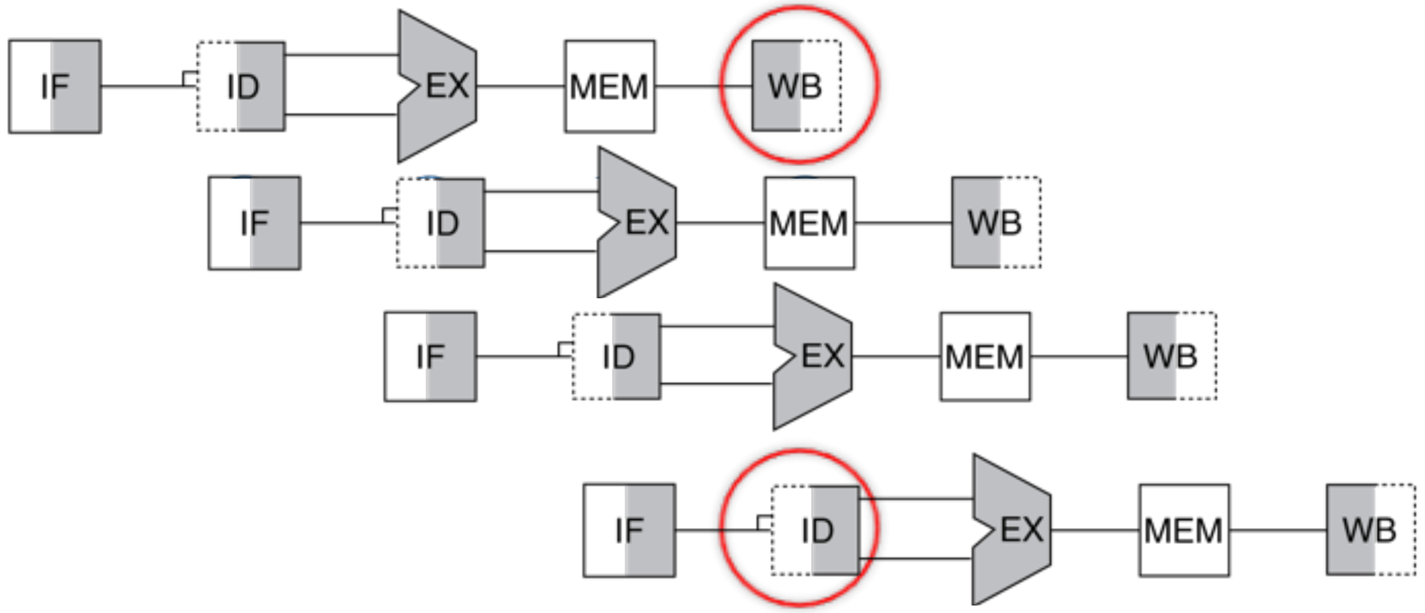
- Solution 2: add more HW
- Separate instruction/data memories or caches

How about Register File?



Register Access

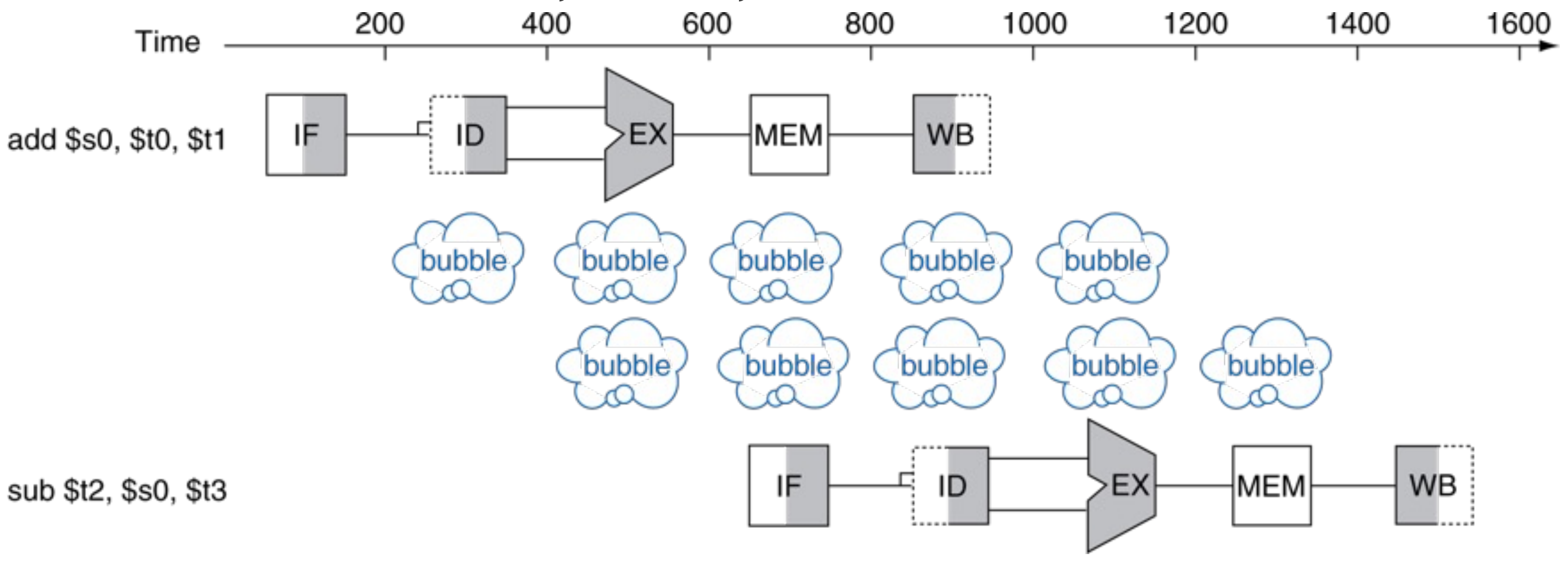
- Divide one cycle into two halves:
  - Register updating in the first half; register reading in the second half: conflict avoided!



Data Hazards

- An instruction depends on data generated by a previous instruction

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



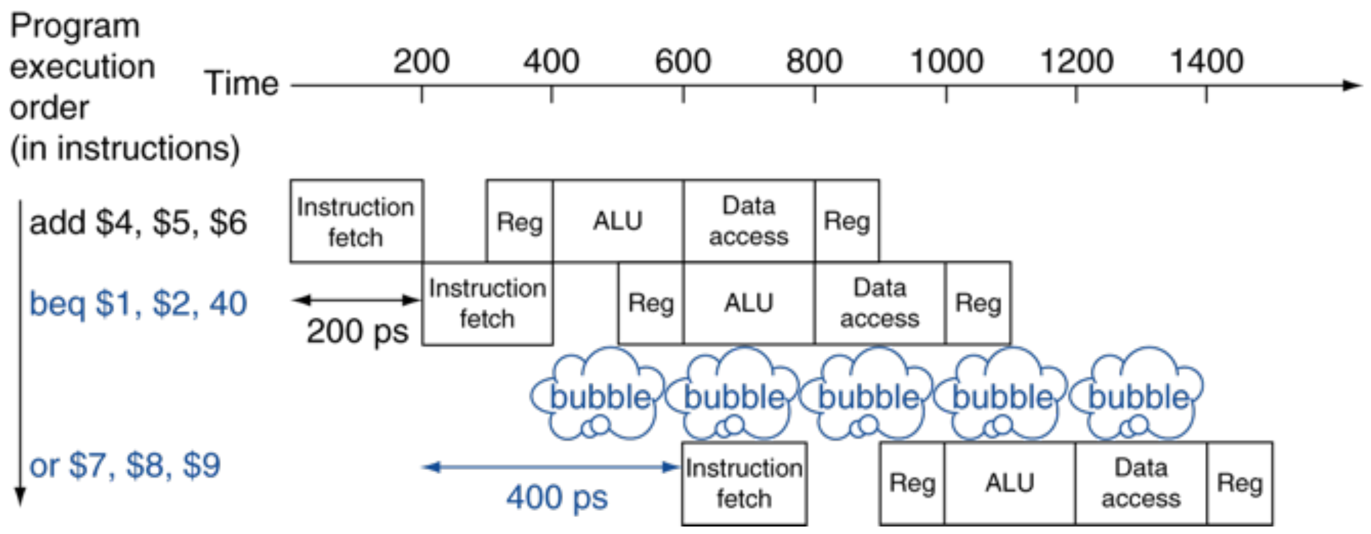
- Better approach: forwarding; reordering

Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - If branch decision is made in EX stage, fetch from the correct addr in MEM stage of the branch instead of ID stage: 2-cycle delay
- How to reduce the delay?
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage
    - Still 1-cycle delay

Stall on Branch

- Wait until branch outcome determined before fetching next instruction



- Better approach: guess/predict branch outcome; branch delay slot