CSE 2021 Computer Organization

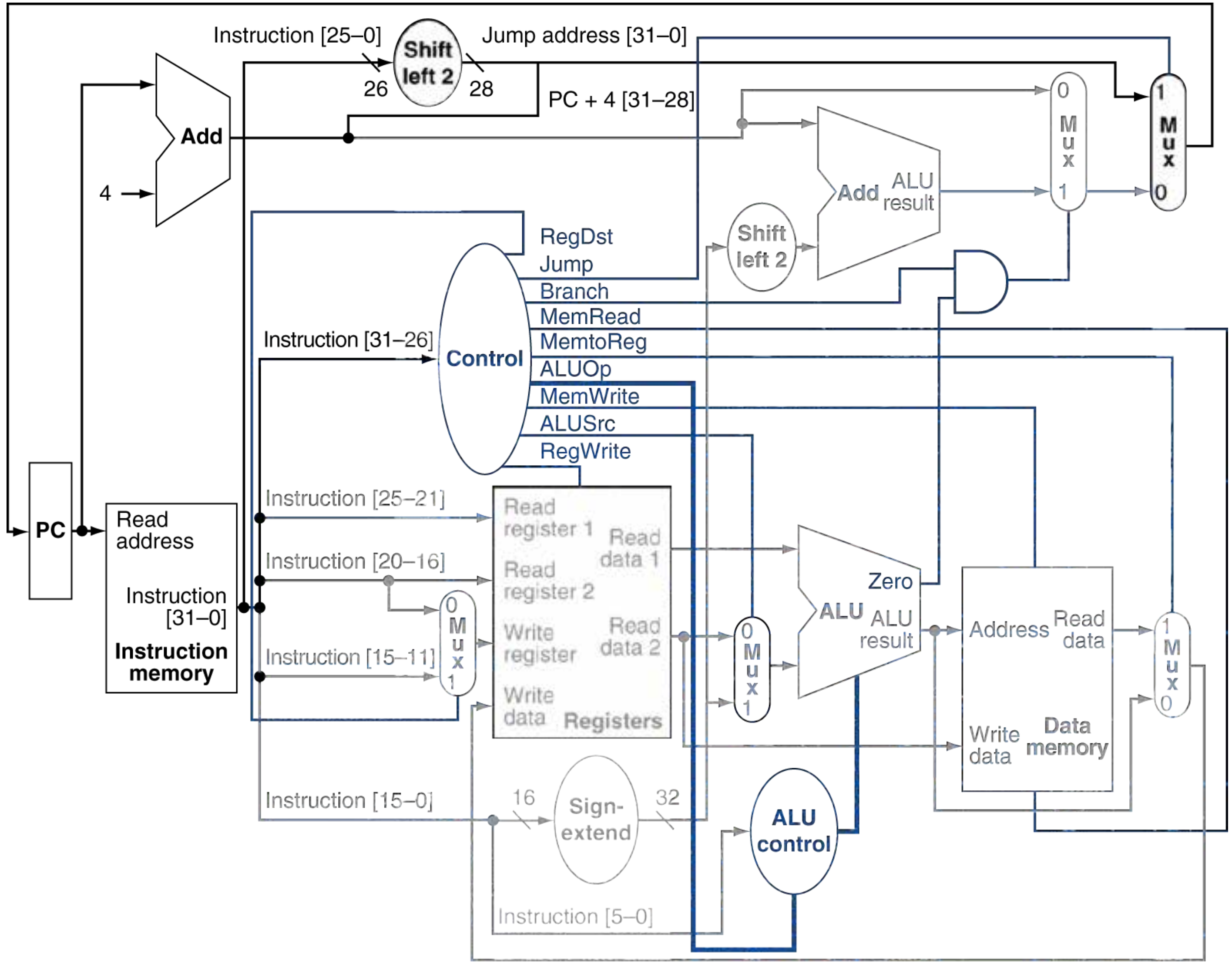
Chapter 4 Part 2

The Processor - Pipelining

Outline

- CPU overview
- Single cycle MIPS implementation
  - Simple subset
  - Memory reference: lw, sw
  - Arithmetic/logical: add, sub, and, or, slt
  - Control transfer: beq, j
- Pipelined MIPS implementation

Single Cycle Implementation



Why not single-cycle implementation?

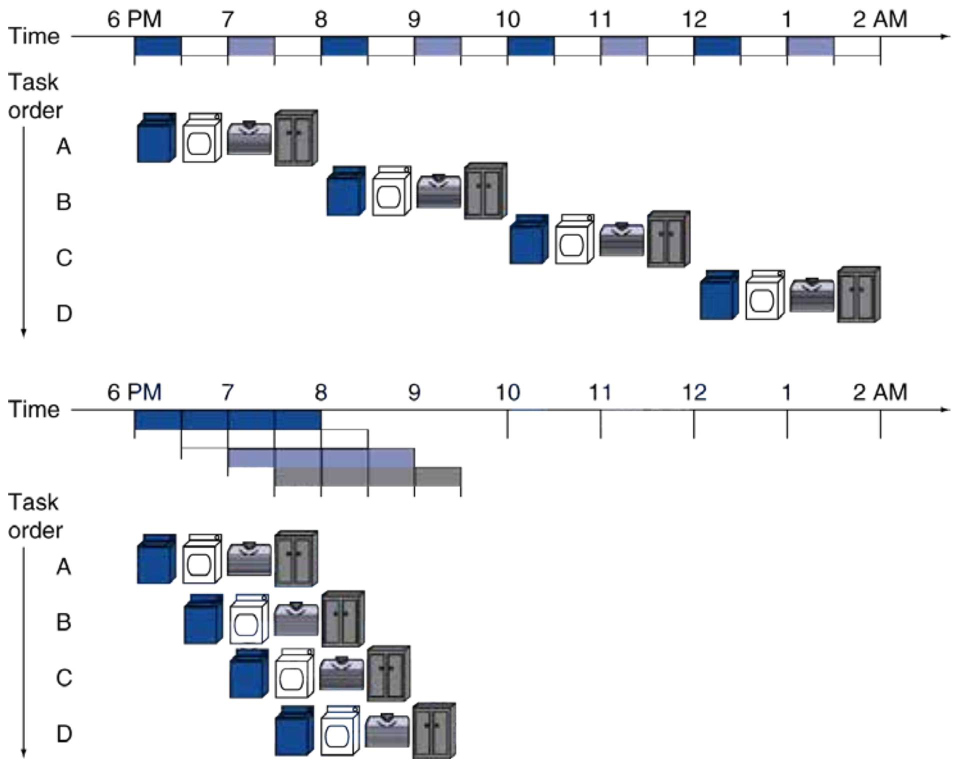
- Assuming no delay at adder, sign extension, shift left unit, PC, control unit and mux
  - lw requires 5 functional units: instruction fetch, register access, ALU, data memory access, register access
  - sw requires 4 functional units: instruction fetch, register access, ALU, data memory access
  - R-type requires 4 functional units: instruction fetch, register access, ALU, register access
  - Branch requires 3 functional units: instruction fetch, register access, ALU
  - Jump requires 1 functional unit, instruction fetch

Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction (lw)
  - Involving 5 functional units
- Using a clock cycle of equal duration for each instruction is a waster of resources
- Not feasible to vary period for different instructions
- We will improve performance by pipelining

Pipelining Analogy

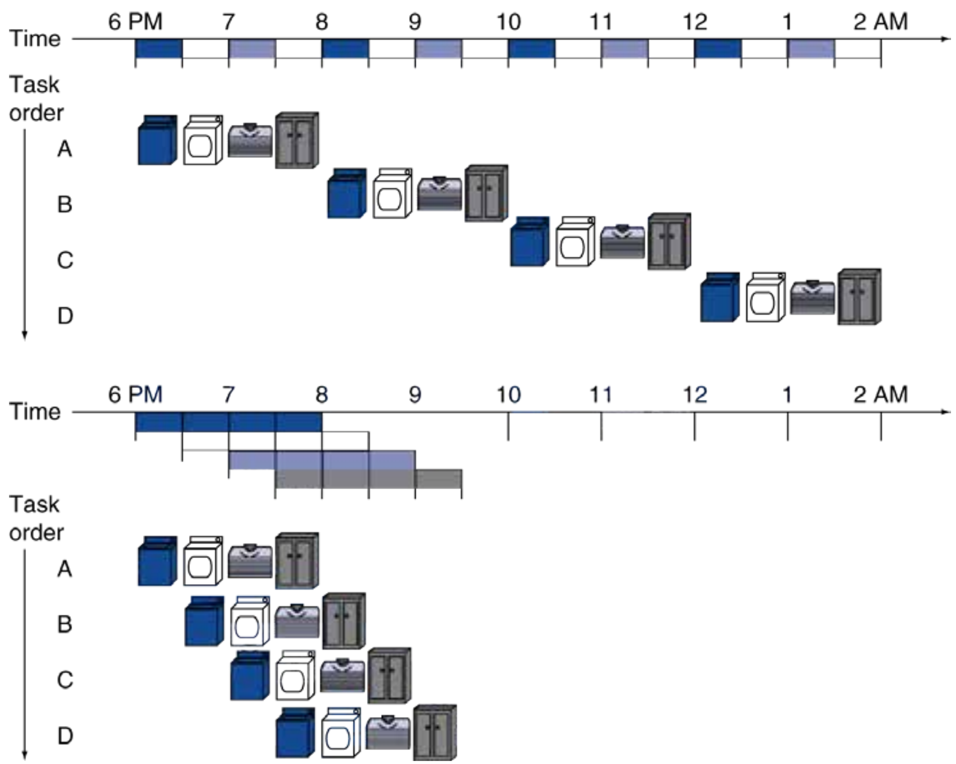
- Pipelined laundry: overlapping execution
  - Parallelism improves performance



- 4 loads:
  - Speedup = 8/3.5 = 2.3

Activity 1

- Calculate what is the speedup factor if there are 1000 washing jobs running in parallel?



MIPS Pipeline

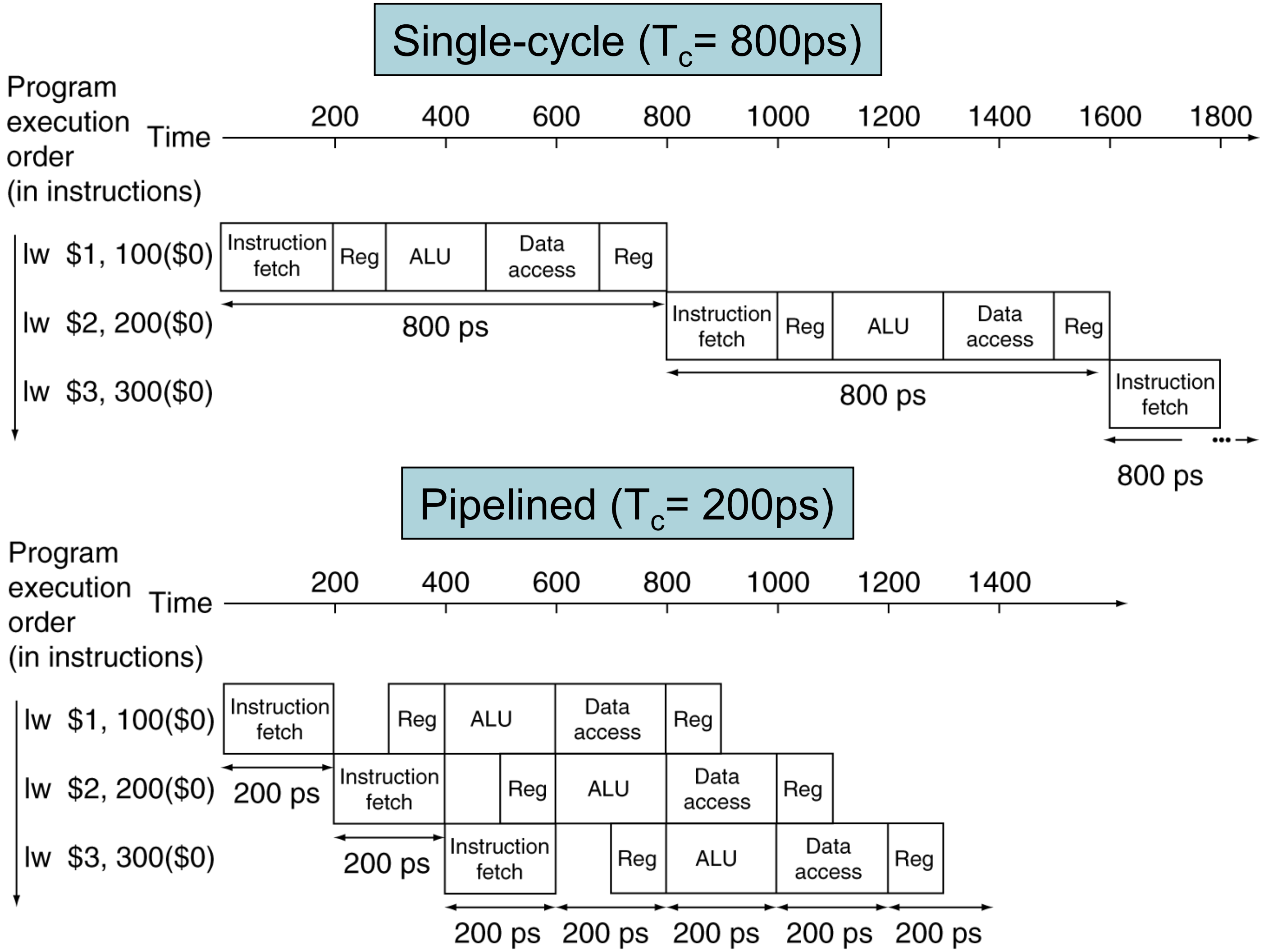
- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Time for different types of single-cycle datapath

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

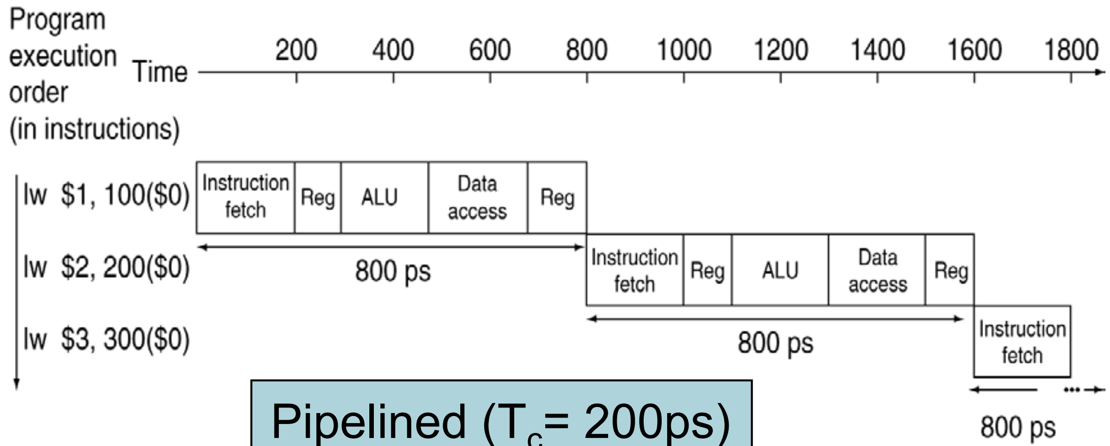
Pipeline Performance



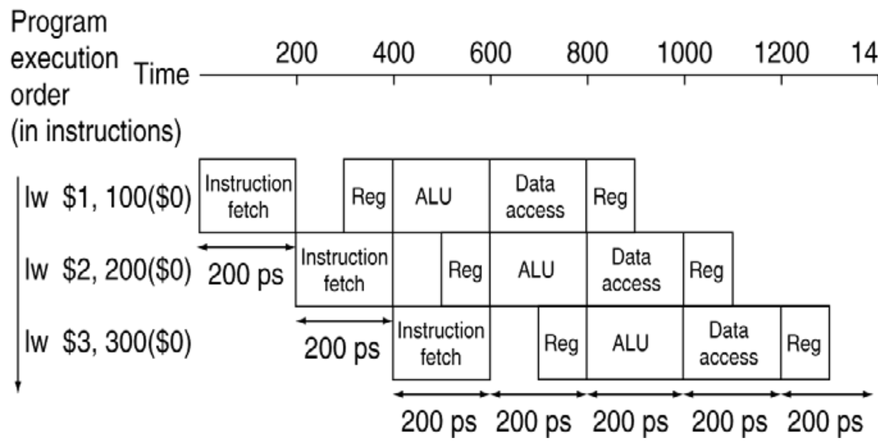
Activity 2

Calculate the speedup factor for running 2000 pipelined Load instructions.

Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{\text{pipelined}}$

$$= \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$

- If not balanced, speedup is less
- Pipelining added some overhead (additional 100ps for Register Read)
- Speedup due to increased throughput
- Latency (execution time for each instruction) remains the same

Pipelining and ISA Design

- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in 1st and 2nd stage
  - Few and regular instruction formats
    - Registers staying specified at almost the same bit positions.
  - Load/store addressing
    - MIPS does not allow operands to be directly used from the memory. Operands are first loaded into the registers.
  - Alignment of memory operands
    - Data can be transferred from memory to registers in a single data transfer command

80x86

- Instructions in 80x86 have variable length from 1 byte to 17 bytes. This makes the first two stages, IF and ID, more challenging making pipelining difficult.
- Due to variable instruction length in 80x86, the registers are specified at different bit positions.
- 80x86 allows direct operation on operands while in memory. An additional address stage is therefore needed in 80x86.

Pipelining Hazards

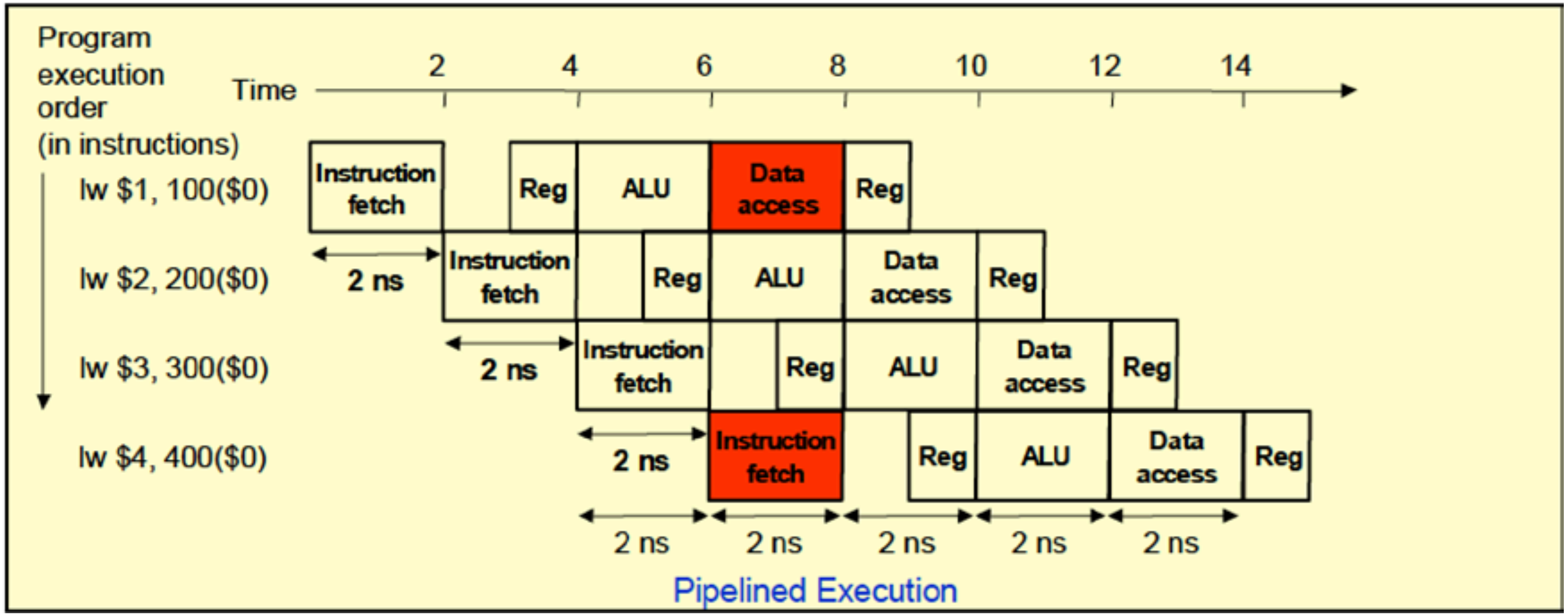
- Hazards occur when the next instruction in a pipelined program can not be executed until the prior instruction has been executed.
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

Structure Hazards

- Conflict for use of a resource
- In 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”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

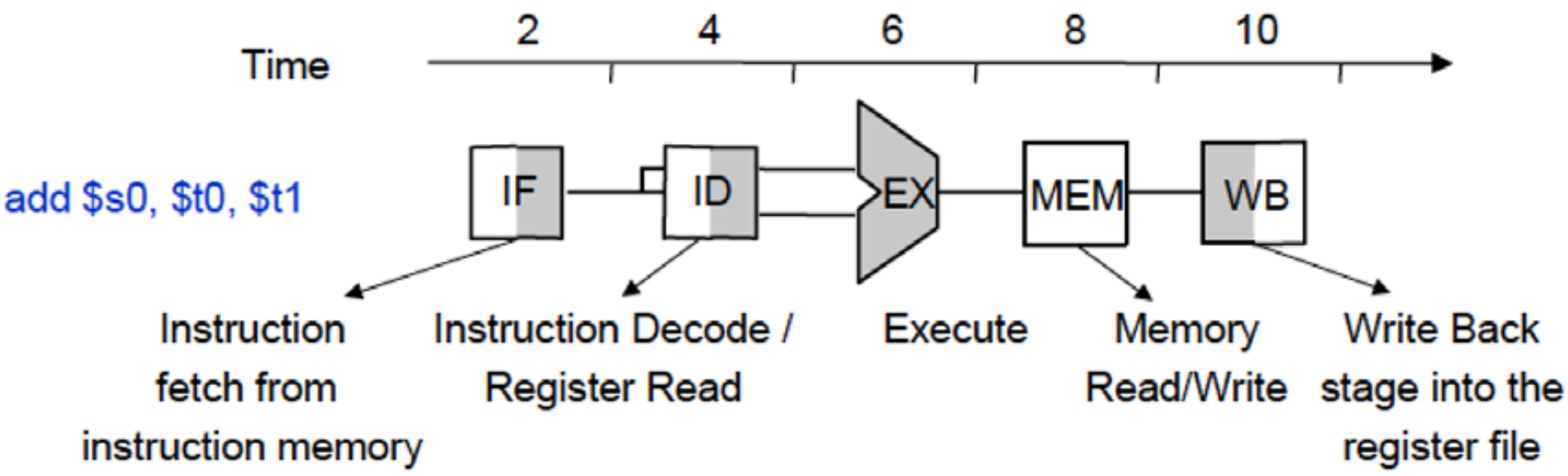
Structure Hazards

- Laundry analogy: A washer-dryer combo is used where a load of clothes is washed and then dried in the same machine.
- MIPS: A single memory unit used for data and instructions results in structural hazard



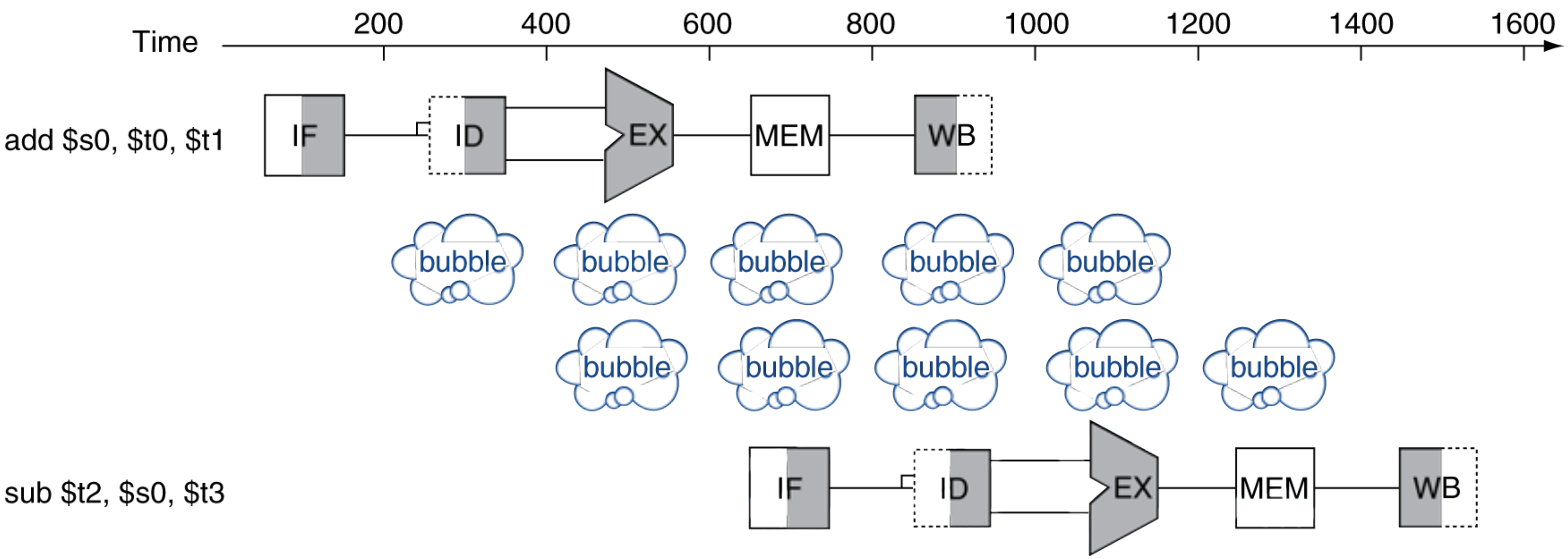
Graphical Representation

- Shading in each block indicates what the element is used for in the instruction
- Shading on left half of the block indicates the element is being written
- Shading on the right half of the block indicates that the element is being read



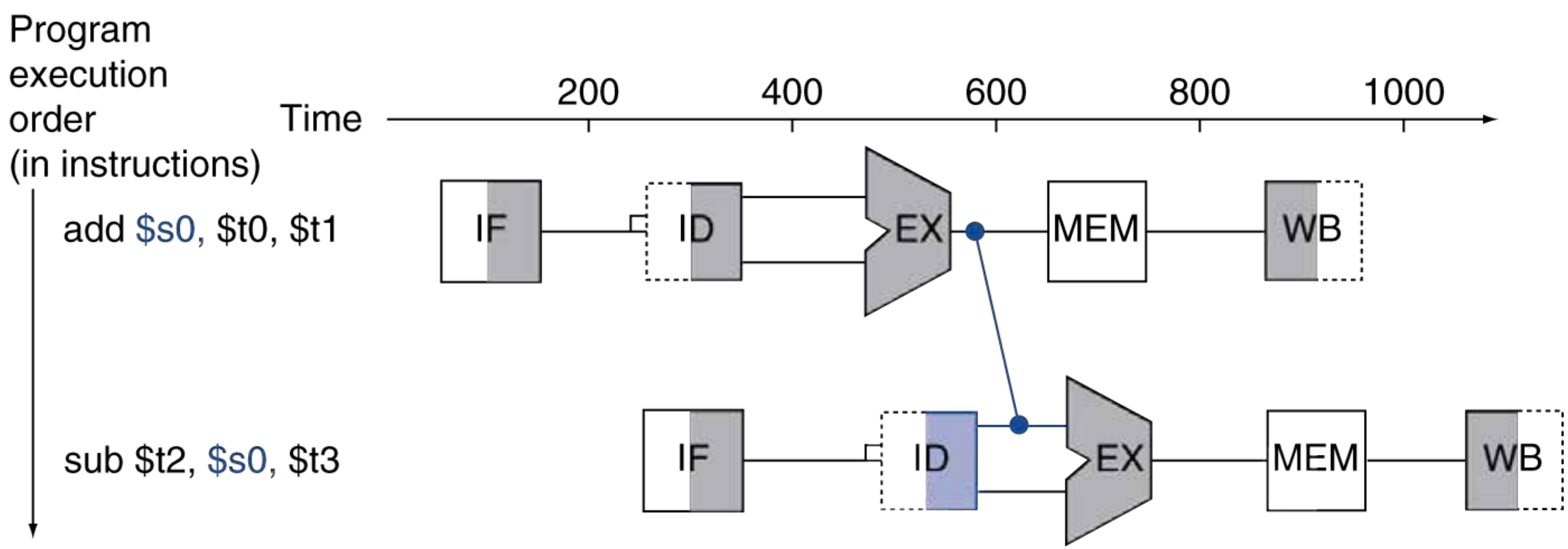
Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - ```
    add  $s0, $t0, $t1
    sub  $t2, $s0, $t3
    ```


    ```

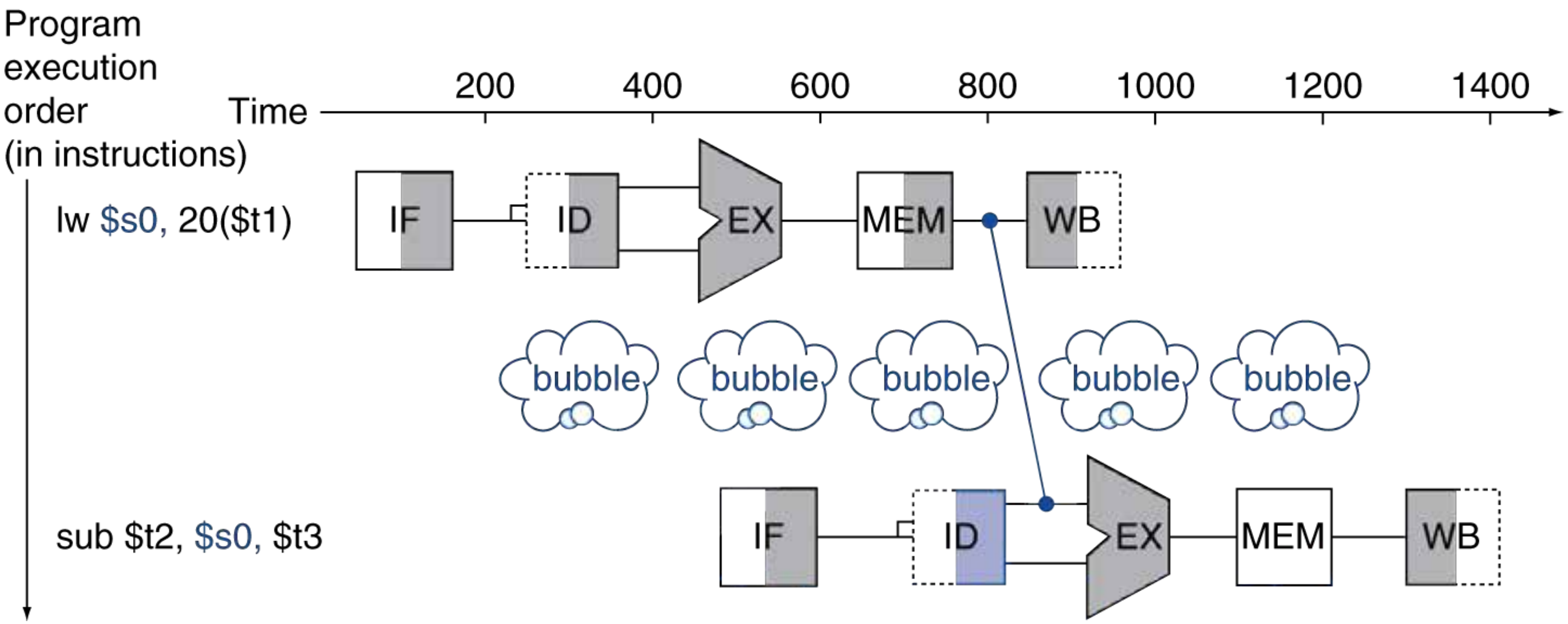
Forwarding (aka Bypassing)

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



Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for A = B + E; C = B + F;

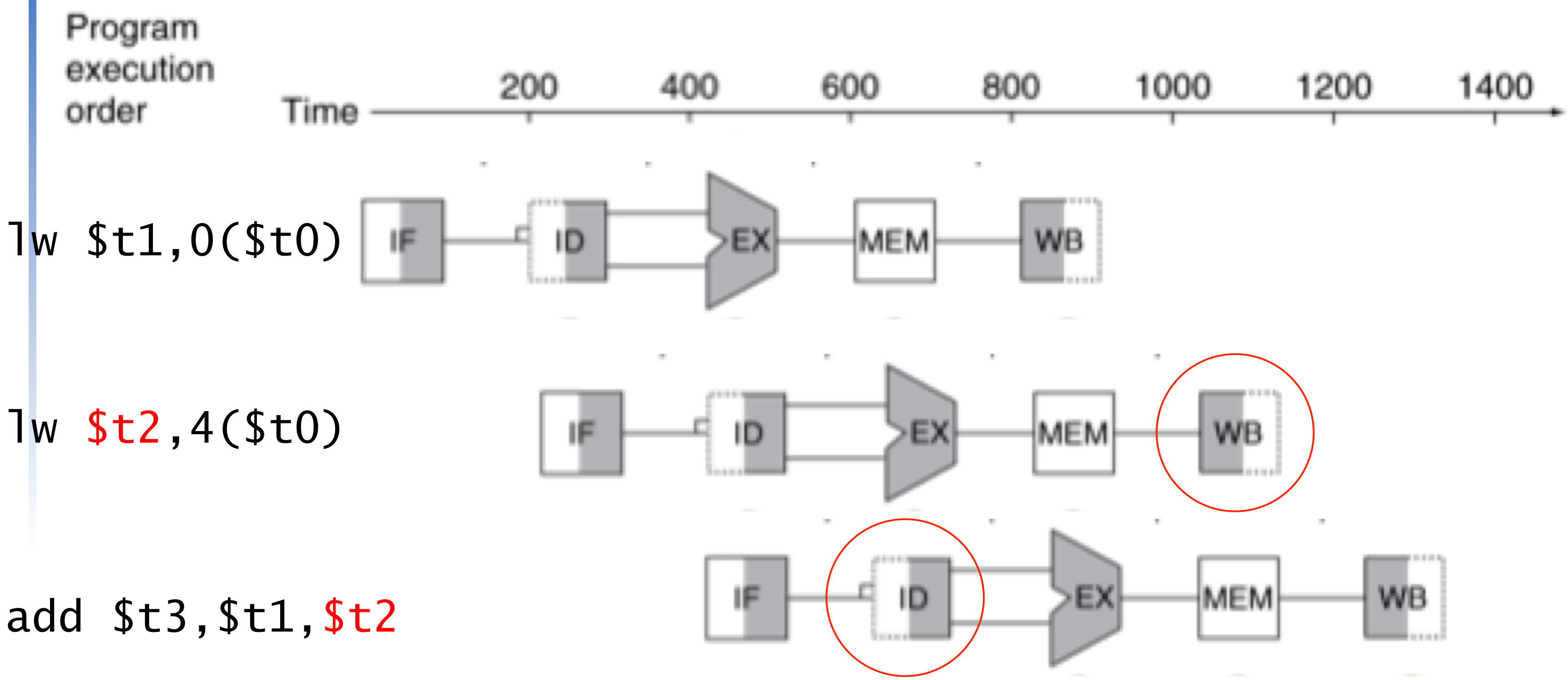
```
      lw  $t1, 0($t0)
      lw  $t2, 4($t0)
stall add $t3, $t1, $t2
      sw  $t3, 12($t0)
      lw  $t4, 8($t0)
stall add $t5, $t1, $t4
      sw  $t5, 16($t0)
```

13 cycles

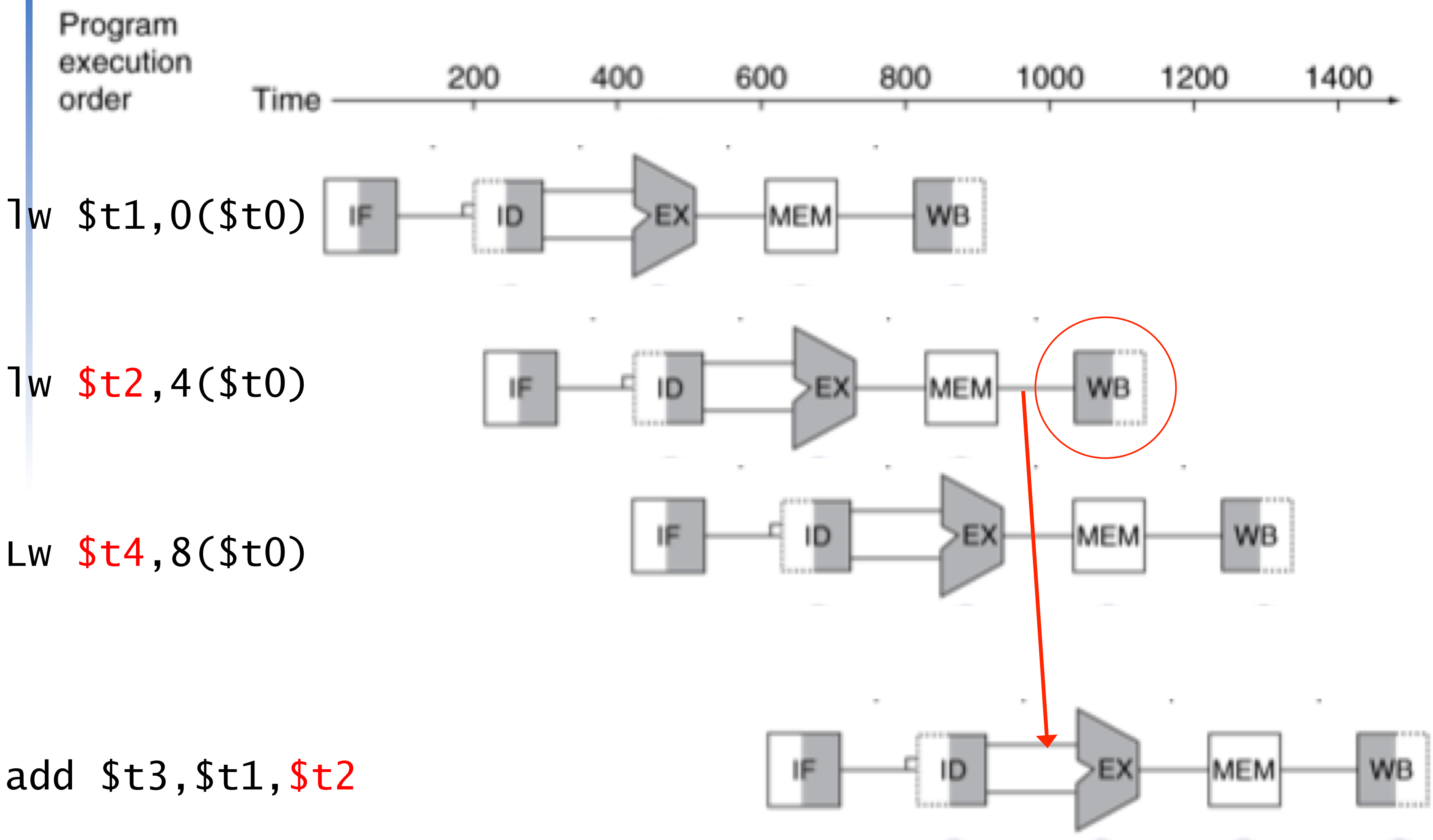
```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
lw  $t4, 8($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

11 cycles

Code Scheduling to Avoid Stalls



Code Scheduling to Avoid Stalls

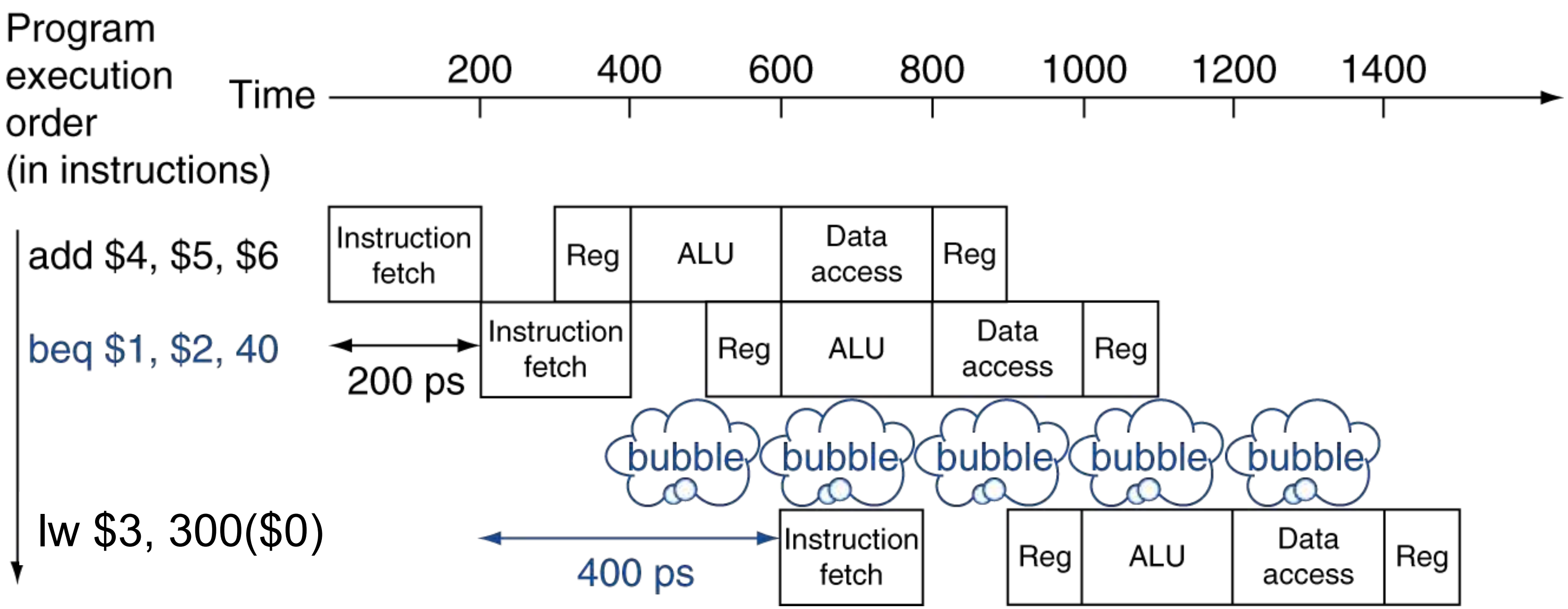


Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

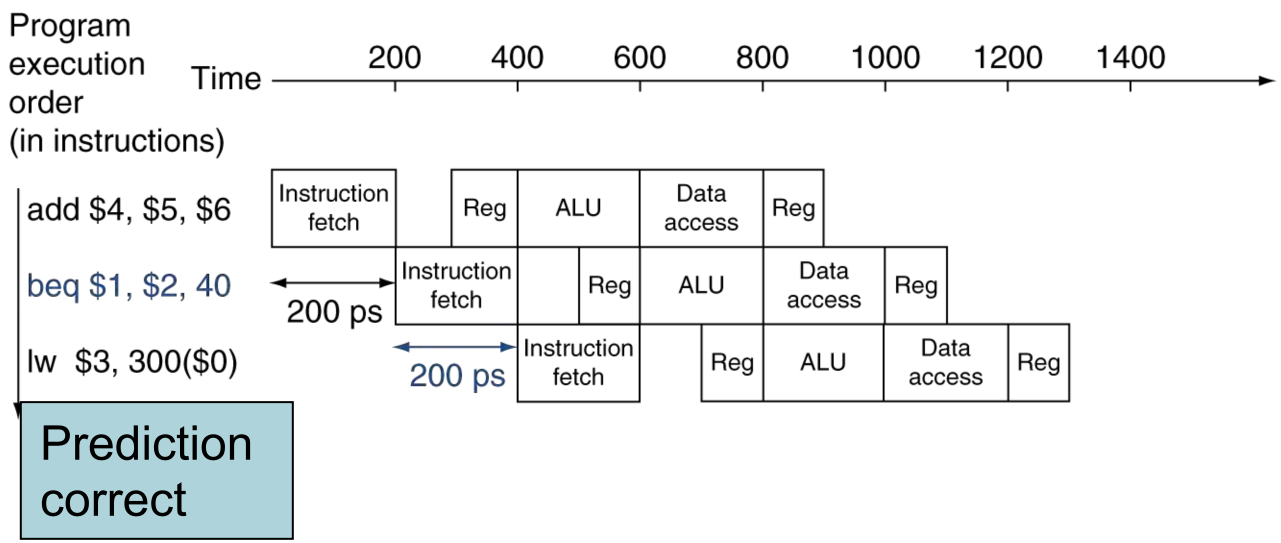
Stall on Branch

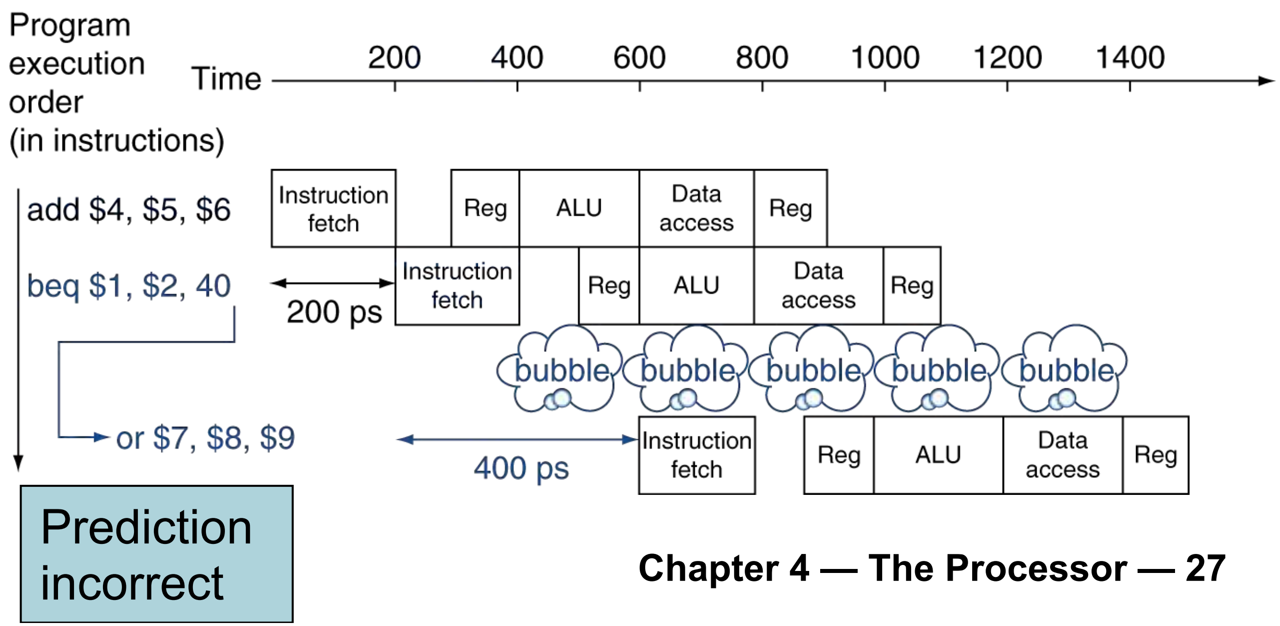
- Wait until branch outcome determined before fetching next instruction – slow!
- Adding extra hardware to determine the branch address – still stalled!



Solution to Control Hazards

- Always predict that the branch will fail and keep executing the program
- Stall if branch is taken





Activity 3

Using the graphical representation, show that the following program has a pipeline hazard. Find a solution to avoid pipeline stall.

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```

Pipeline Summary

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation