Enhancing Performance with Pipelining

Section 4.5 Of

Book of David A. Patterson

Pipelining

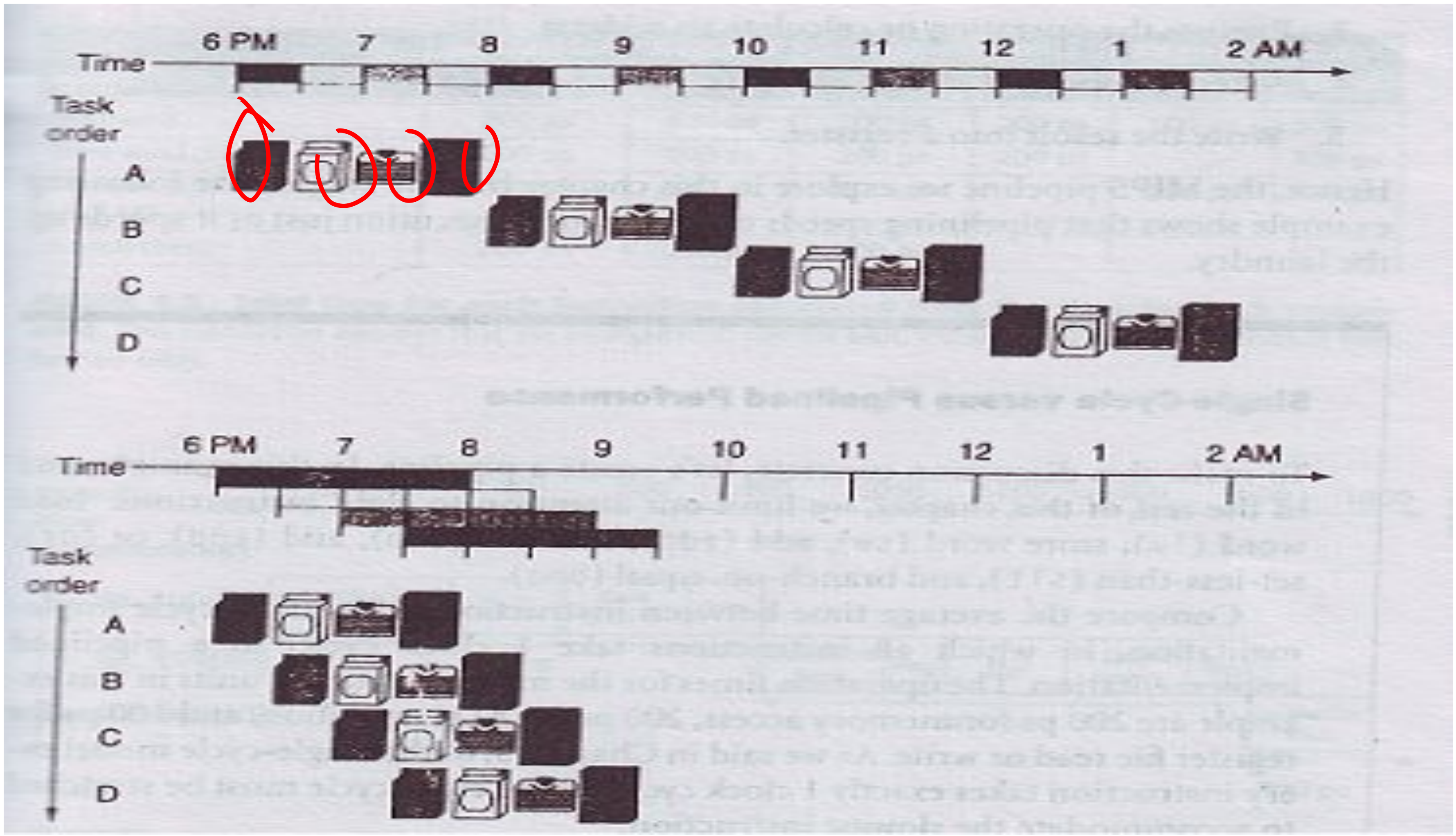
- ✔ An implementation technique in which multiple instructions are overlapped in execution.
- ✔ It is used to make the processor fast.
- ✔ Pipelining does not reduce the time to complete a single task but increases the throughput and the improvement in the throughput decreases the total time to complete the task.
- ✔ The speed up due to pipelining is equal to the number of stages in the pipeline if

  1. All the stages take about the same amount of time.

  2. The number of tasks is large compared to the number of stages.

100 $\angle$ 0 5

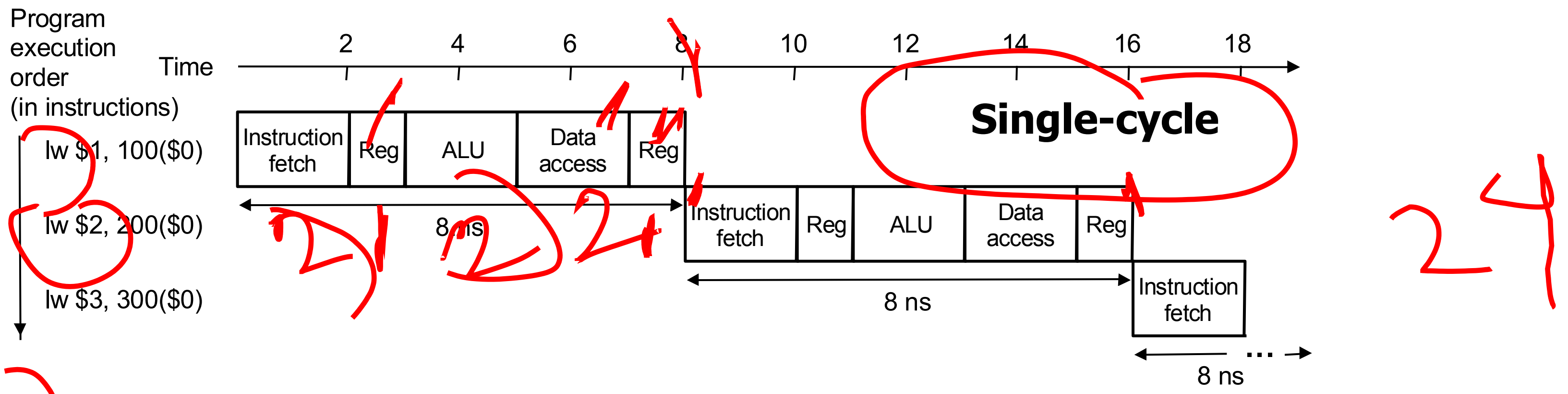
Laundry Analogy for Pipelining



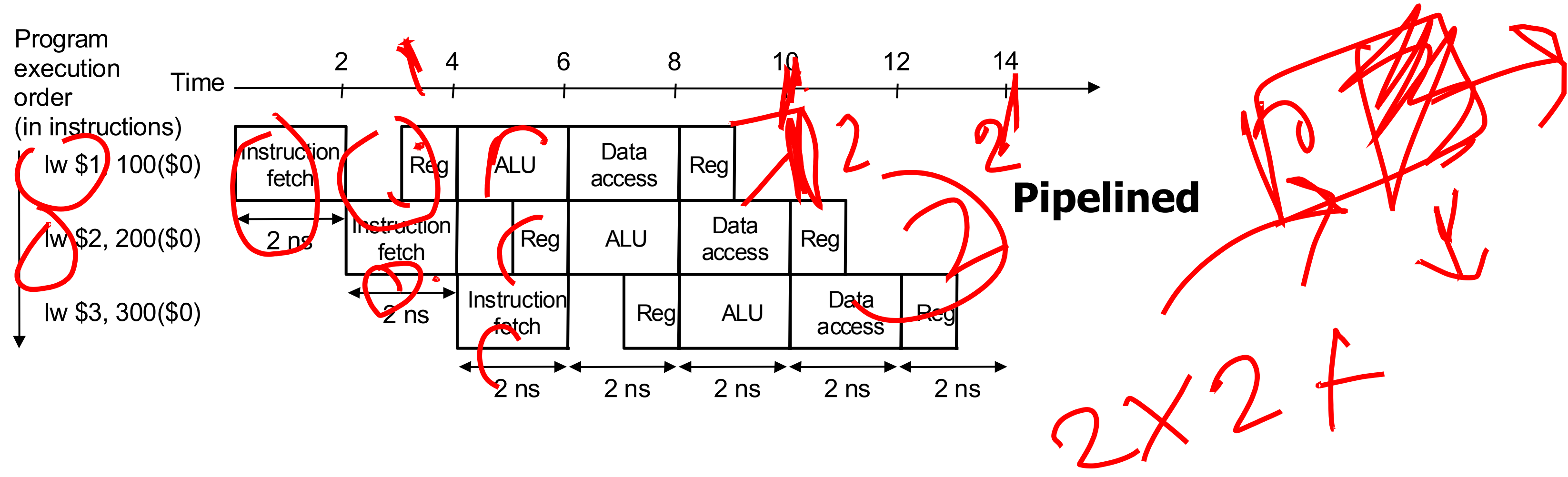
MIPS Pipeline

- ✔ Fetch instruction from the memory.
- ✔ Read register while decoding the instruction.
- ✔ Execute the operation or calculate an address.
- ✔ Access an operand in data memory.
- ✔ Write the result into a register.

Pipelined vs. Single-Cycle Instruction Execution



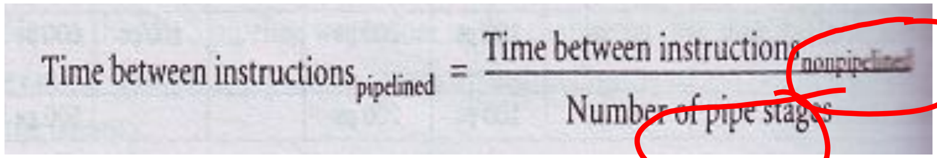
Assume 2 ns for memory access, ALU operation; 1 ns for register access: therefore, single cycle clock 8 ns; pipelined clock cycle 2 ns.



Speed Up of Pipelining

880

3.84



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

- ✔Time between the instruction must be 1.6 ns. We get 2 ns due to imperfectly balanced stage.
- ✔With respect to total execution time, we get 24 ns / 14 ns = 1.7 which is not equal to 5. This is due to small number of instruction.

3

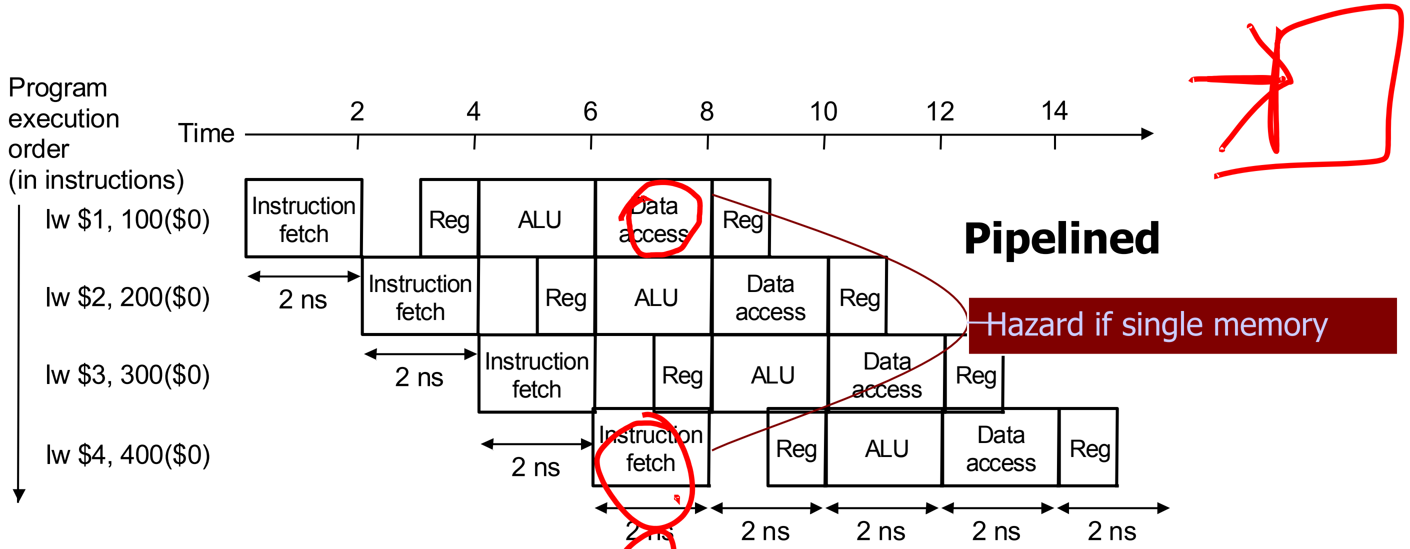
$$\frac{100 \times 8}{10 + 99 \times 2}$$

Pipelining MIPS

- ✔ all **instructions are same length**
  - -so fetch and decode stages are similar for all instructions
- ✔ just a **few instruction formats** R
  - -simplifies instruction decode and makes it possible in one stage
- ✔ **memory operands appear only in load/stores**
  - -so memory access can be deferred to exactly one later stage
- ✔ **operands are aligned in memory**
  - -one data transfer instruction requires one memory access stage

Structural Hazards

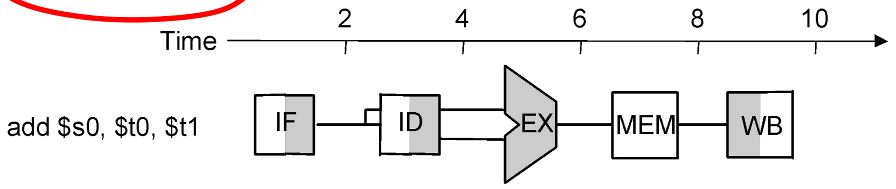
- ✔ An occurrence in which a planned instruction cannot execute in the proper clock cycle because the hardwire cannot support the combination of instruction that are set to execute in the given clock cycle.
- ✔ E.g., suppose *single – not separate* – instruction and data memory in pipeline below with *one read port*
  - ✔ then a structural hazard between first and fourth `lw` instructions



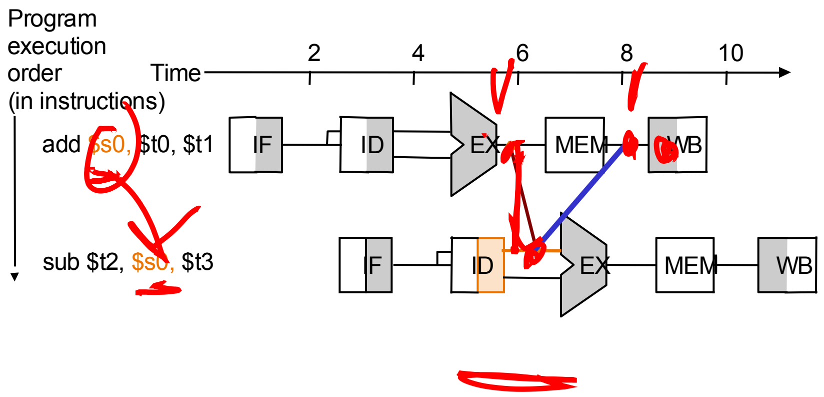
- ✔ *MIPS was designed to be pipelined*: structural hazards are easy to avoid!

Data Hazards

- ✔ An occurrence in which a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.
- ✔ Instruction needs data from the result of a previous instruction still executing in pipeline
- ✔ <u>Solution</u> *Forward* data if possible…



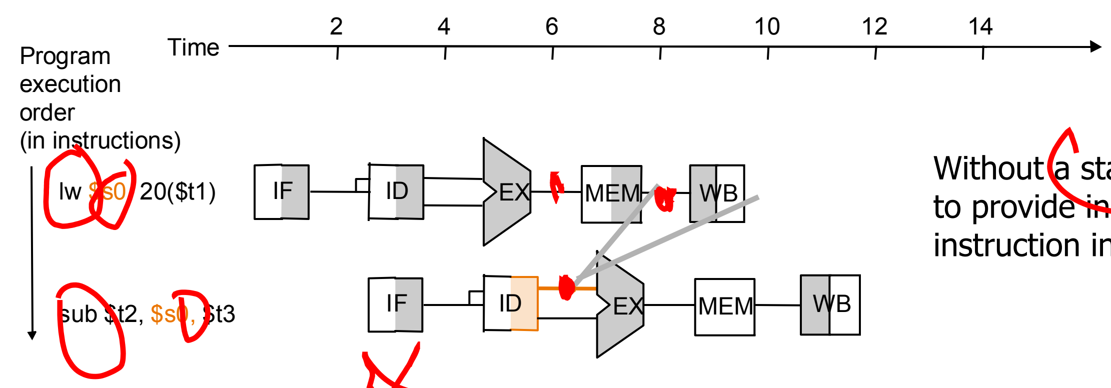
Instruction pipeline diagram: shade indicates use – left=write, right=read



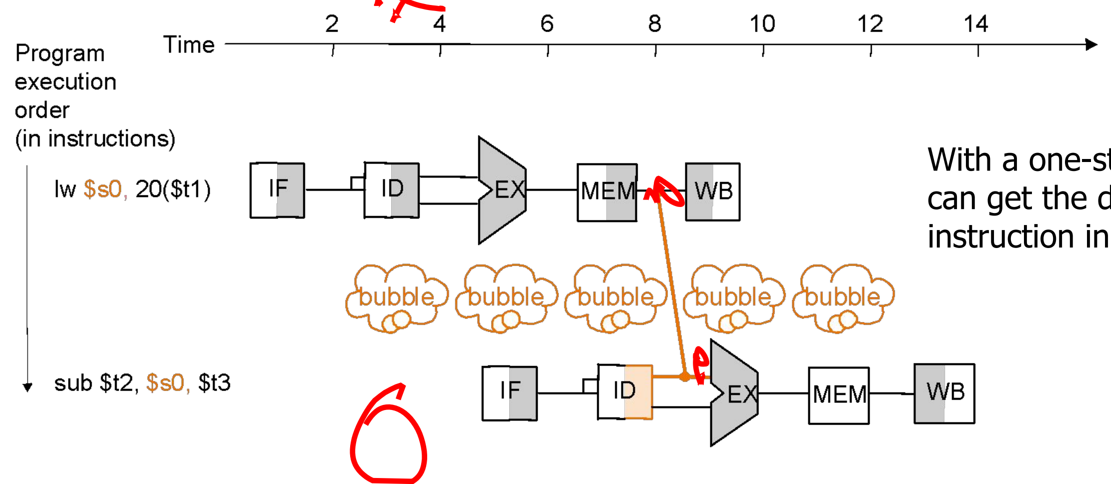
Without forwarding – blue line – data has to go back in time; with forwarding – red line – data is available in time

Data Hazards

- Forwarding may not be enough
  - e.g., if an R-type instruction following a load uses the result of the load – called *load-use data hazard*

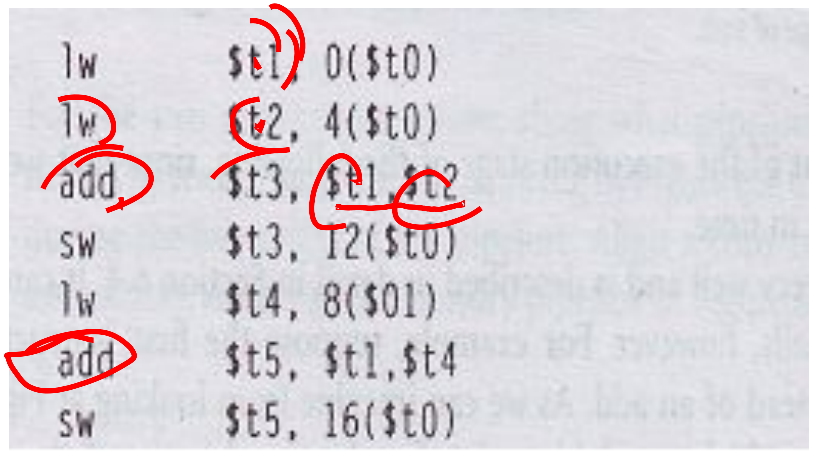


Without a stall it is impossible to provide input to the `sub` instruction in time

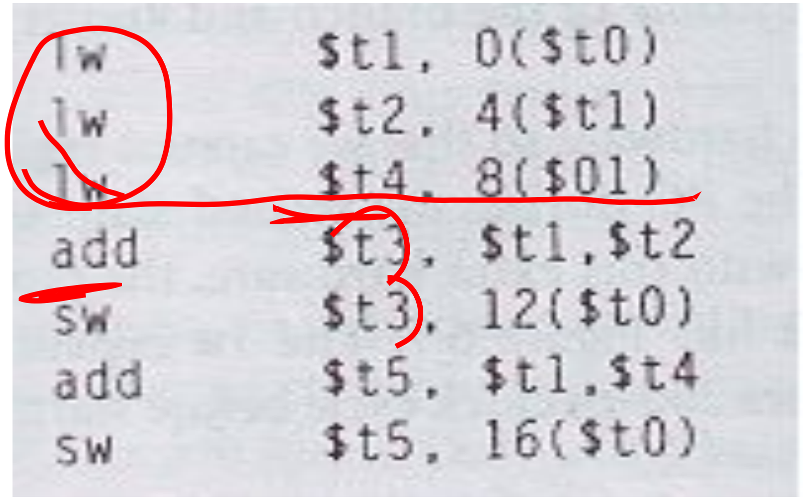


With a one-stage stall, forwarding can get the data to the `sub` instruction in time

Reordering Code to Avoid Pipeline Stall (Software Solution)



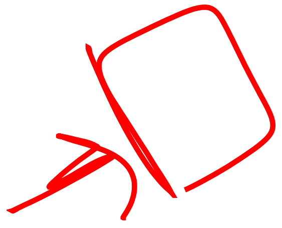
```
lw      $t1, 0($t0)
lw      $t2, 4($t0)
add     $t3, $t1,$t2
sw      $t3, 12($t0)
lw      $t4, 8($01)
add     $t5, $t1,$t4
sw      $t5, 16($t0)
```



```
lw      $t1, 0($t0)
lw      $t2, 4($t1)
lw      $t4, 8($01)
add     $t3, $t1,$t2
sw      $t3, 12($t0)
add     $t5, $t1,$t4
sw      $t5, 16($t0)
```

Control Hazards/ Branch Hazards

- ✔ An occurrence in which the proper instruction cannot execute in the proper clock cycle because the instruction that was fetched is not the one that is needed.
- ✔ Arises from the need to make a decision based on the result of one instruction while others are executing.

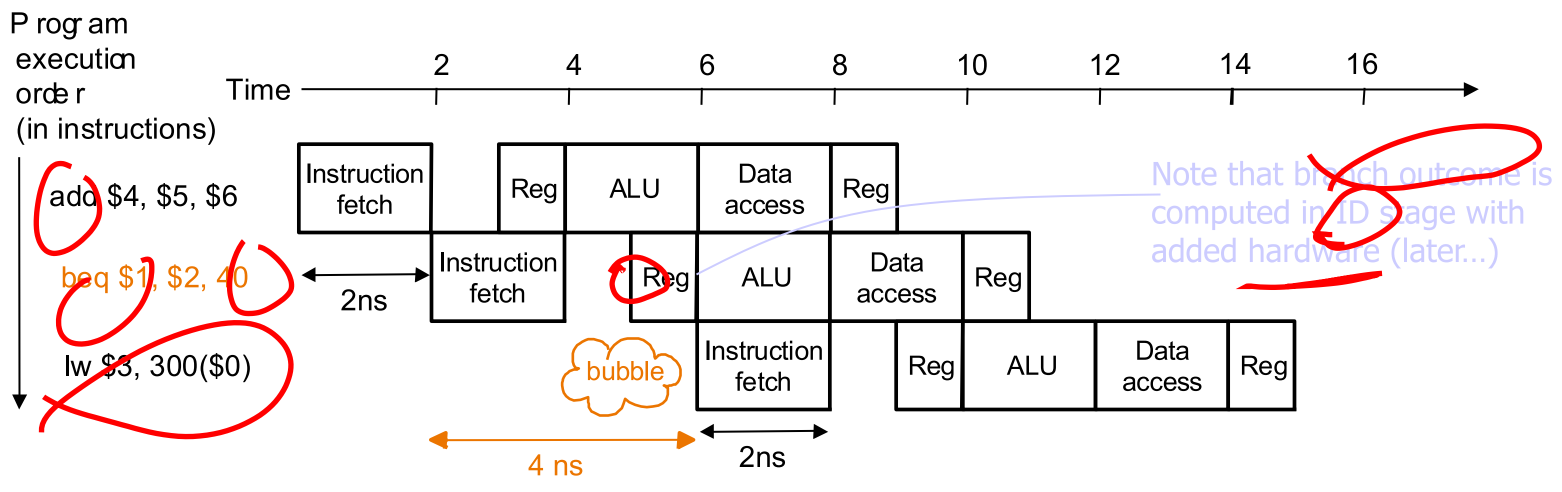


1. Structural

2. Data

Control Hazards

- *Control hazard*: need to make a decision based on the result of a previous instruction still executing in pipeline
- Solution 1: *Stall* the pipeline

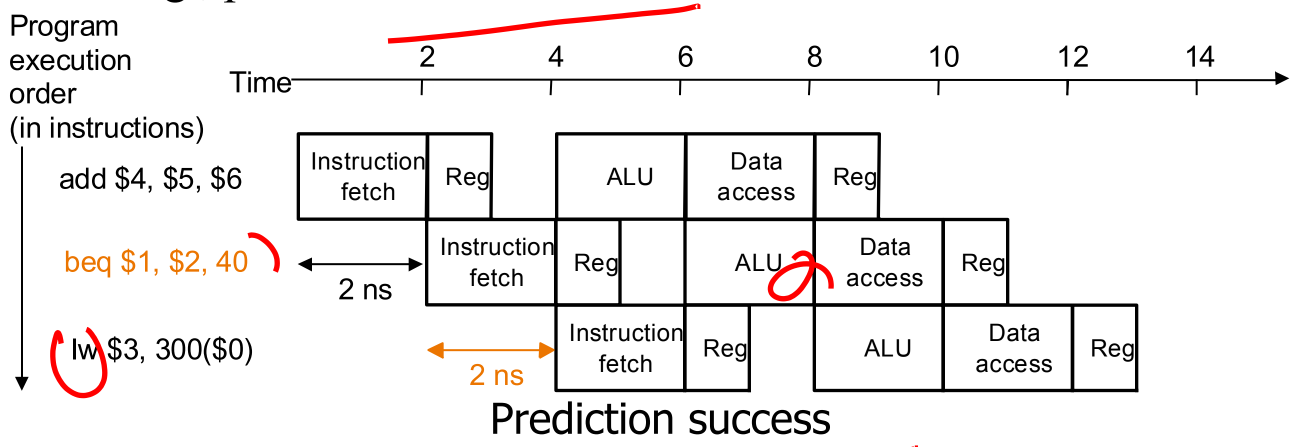


Program execution order (in instructions)
Time
2
4
6
8
10
12
14
16
add $4, $5, $6
beq $1, $2, 40
lw $3, 300($0)
Instruction fetch
Reg
ALU
Data access
2ns
bubble
4 ns
Note that branch outcome is computed in ID stage with added hardware (later…)

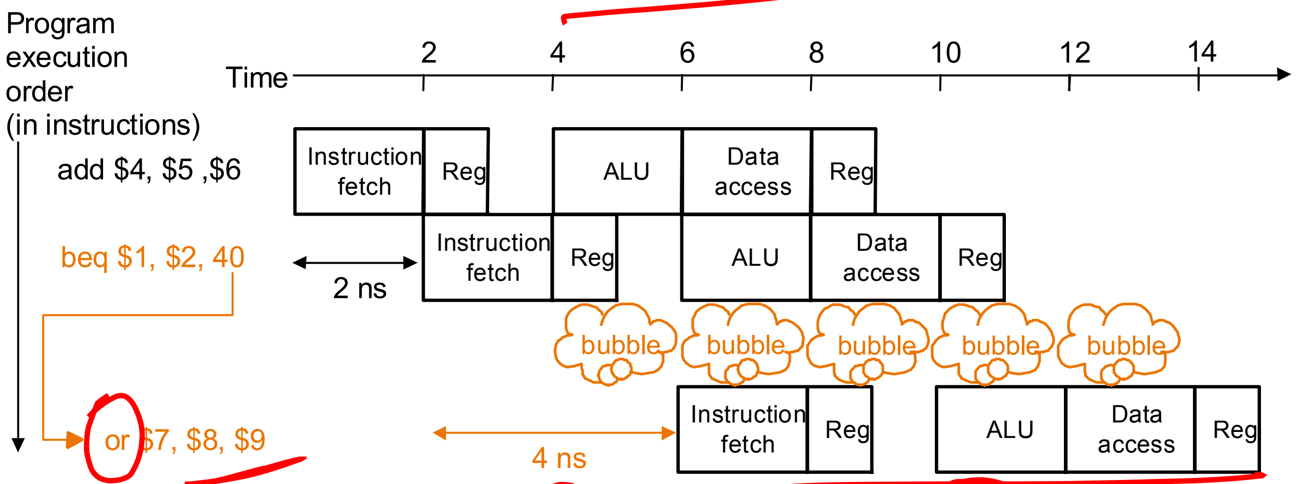
Pipeline stall

Control Hazards

- Solution 2 *Predict* branch outcome
  - e.g., predict *branch-not-taken* :



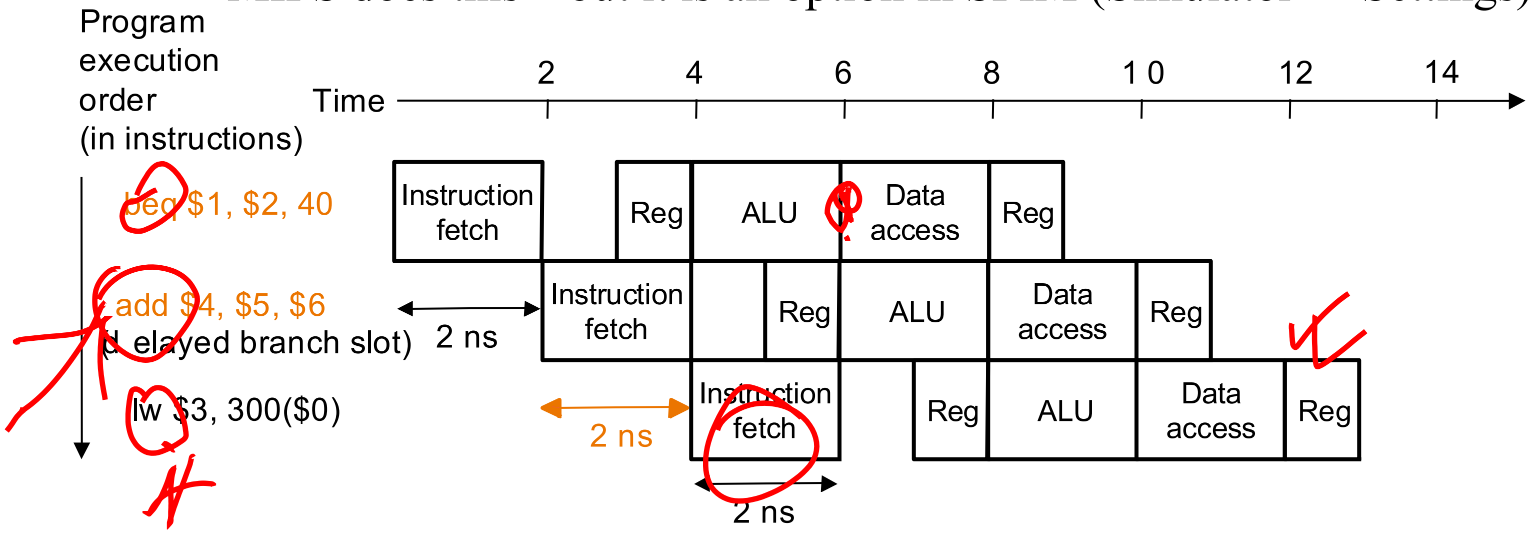
Prediction success



Prediction failure: undo (=flush) `lw`

Control Hazards

- Solution 3 *Delayed branch:* always execute the sequentially next statement with the branch executing after one instruction delay – compiler's job to find a statement that can be put in the slot that is independent of branch outcome
  - MIPS does this – but it is an option in SPIM (Simulator -> Settings)



Delayed branch `beq` is followed by `add` that is independent of branch outcome