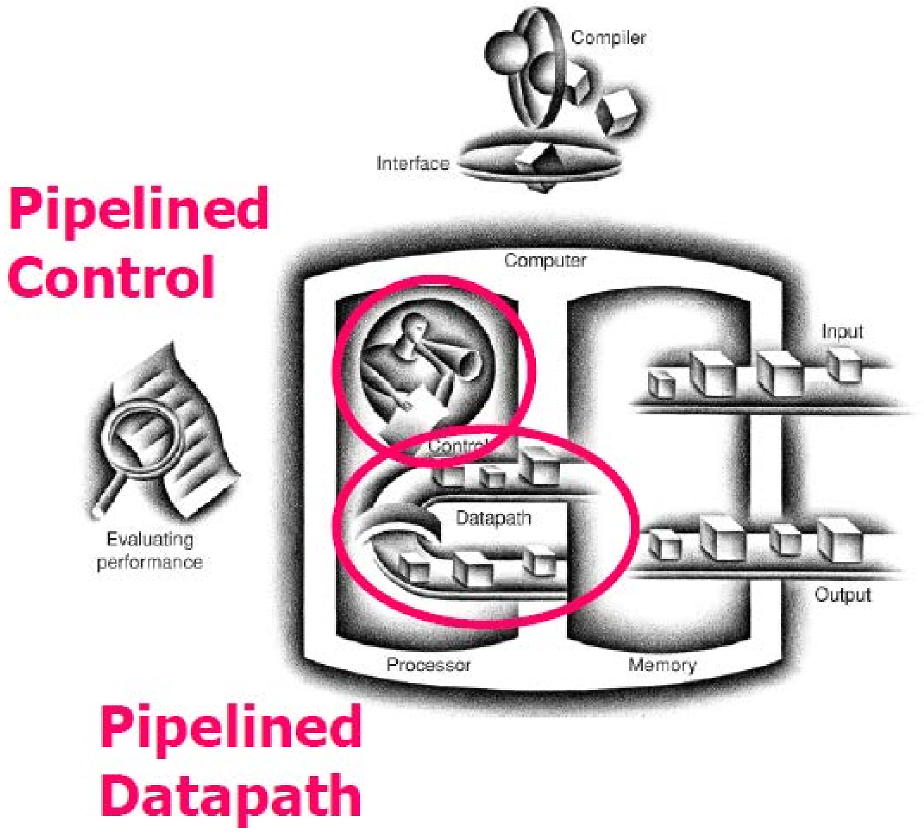
EE2011 Computer Organization

Lecture 9: An Overview of Pipelining (Ch. 4.5)

Wen-Yen Lin, Ph.D.
Department of Electrical Engineering
Chang Gung University
Email: wylin@mail.cgu.edu.tw
May 2022

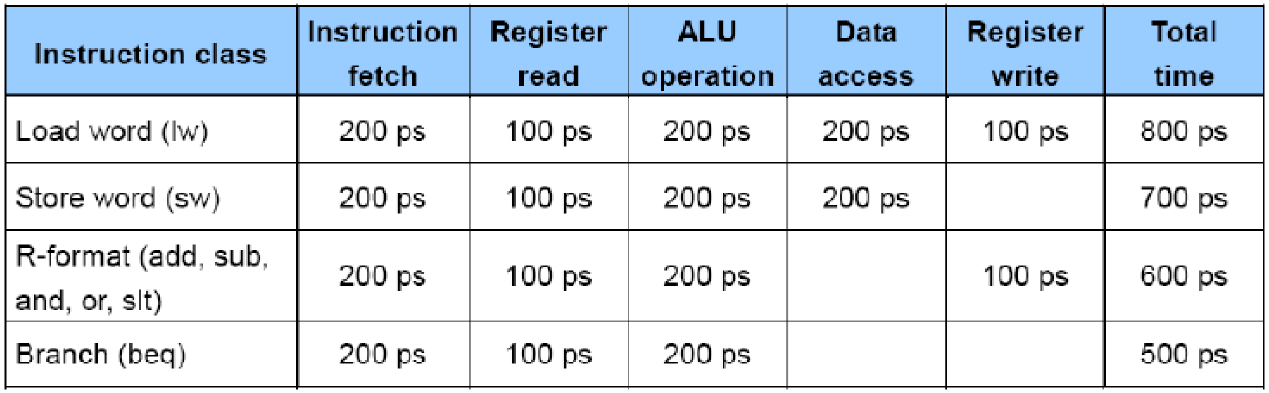
PPSoC
Parallel Processing and SoC Lab
Chang Gung University

Enhancing Performance with Pipelining ~ An Overview (Chapter 4.6)



MIPS Instruction Executions (P. 286)

- Five steps of MIPS Instruction execution
  - Instruction fetch
  - Instruction decode & Register read
  - Execute the operation or calculate an address
  - Access an operand in data memory
  - Write the result back to a register



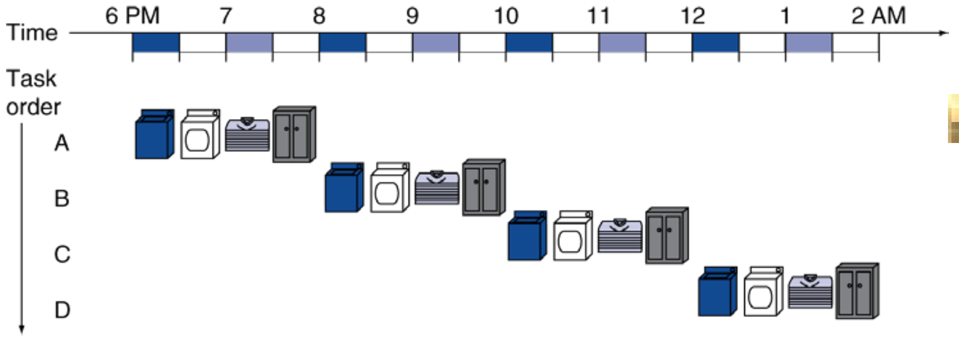
Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

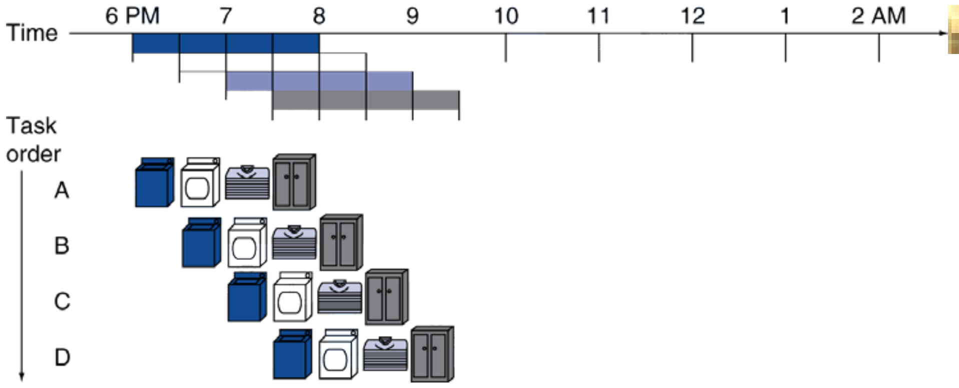
Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining

Pipelining Analogy (Fig. 4.25)

- Pipelined laundry: overlapping execution
  - ⇨ Parallelism improves performance





- Four loads:
  - ⇨ Speedup = 8/3.5 = 2.3
- Non-stop:
  - ⇨ Speedup = 2n/(0.5n + 1.5) ≈ 4 = number of stages

MIPS Pipeline

- Separate five execution steps into 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

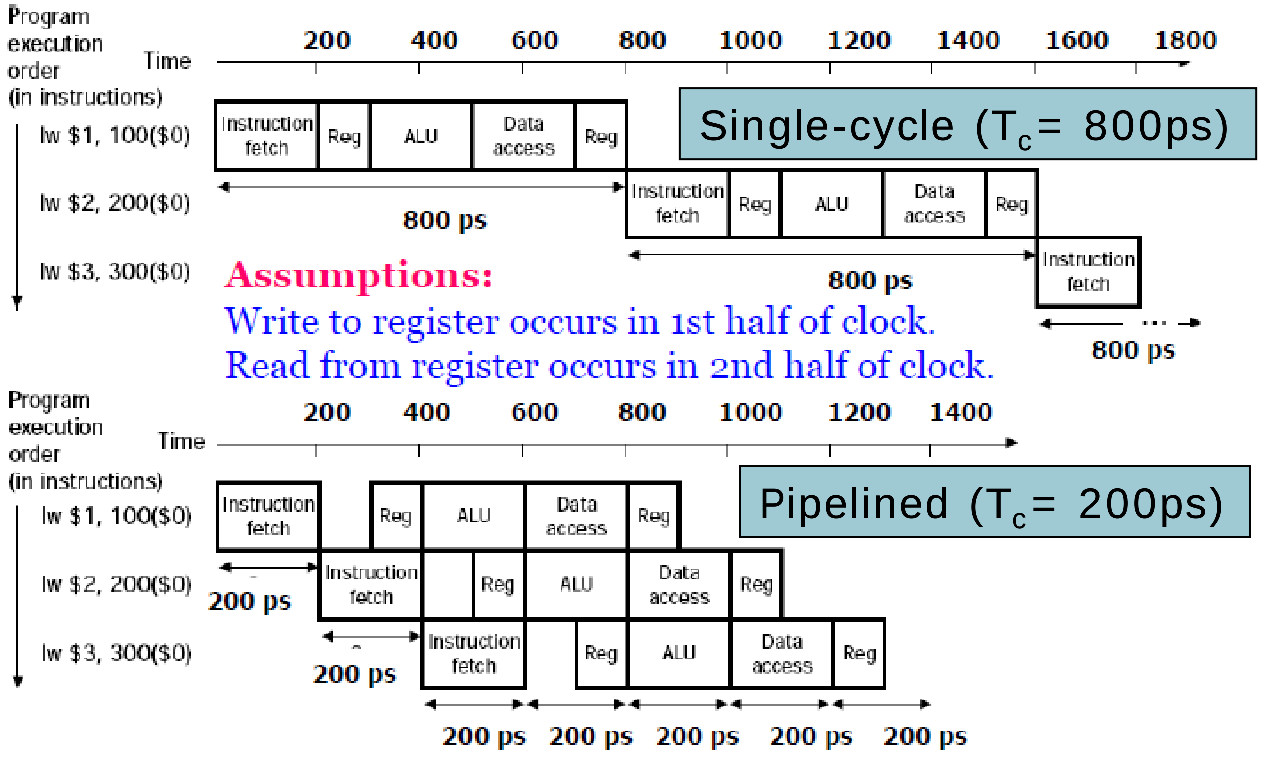
Pipeline Performance (p. 287)

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with 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

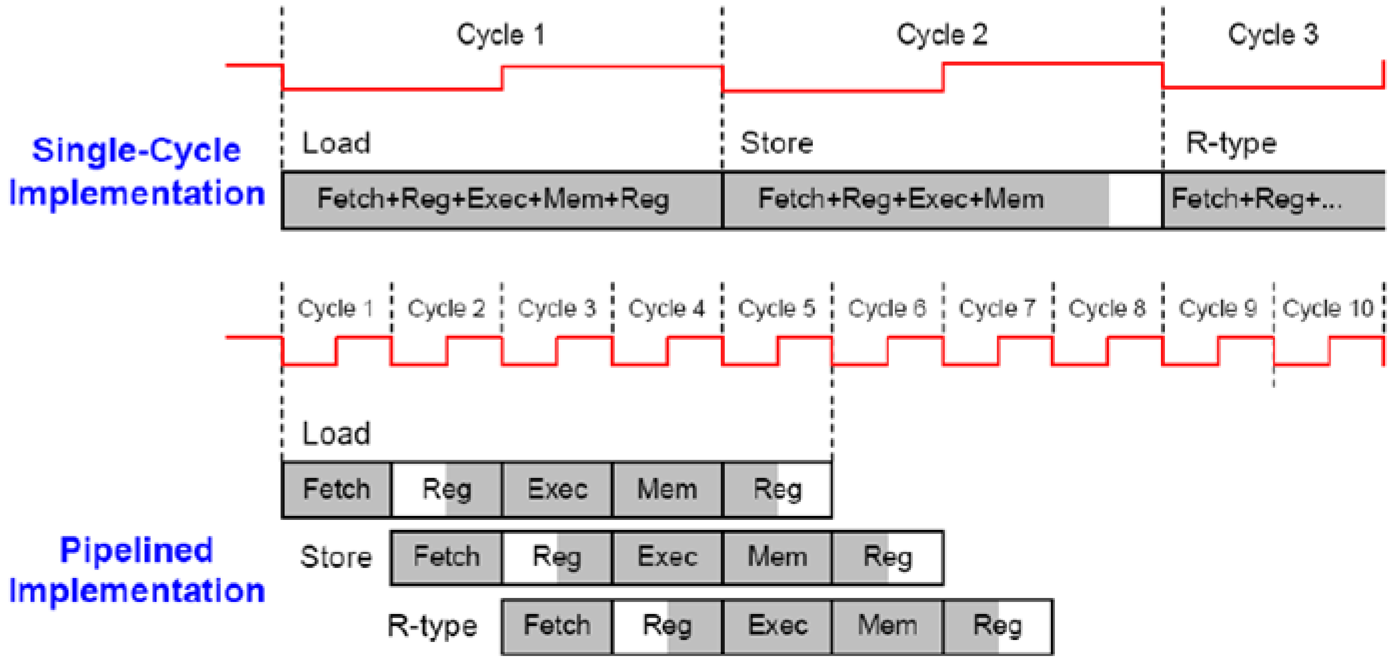
Pipeline Performance (Fig. 4.27)

- Improve performance by increasing instruction throughput



Ideal speedup is number of stages in the pipeline. Do we achieve this?

Single-Cycle and Pipelining Performances



Ideal speedup is number of stages in the pipeline. Do we achieve this?

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
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Pipelining Observations

- Pipelining is an implementation technique in which multiple instructions are overlapped, and is the key to make processors faster.
- Pipelining doesn't reduce number of stages from multicycle implementation
  - Doesn't help latency of single task
  - Helps throughput of entire process
- As long as we have separate resources, we can pipeline the tasks
- Multiple tasks operate simultaneously using different resources
- Speedup due to pipelining depends on the number stages in the pipeline.
- Pipeline rate limited by the slowest pipeline stage
  - Similar with the cycle time of multi-cycle implementation
- Time to fill up pipeline reduces speedup
- Sometimes, we have to wait in the pipeline
  - Hazards

Pipelining Observations

- Theoretically
  - Speedup should be equal to number of stages
- Practically
  - Stages are imperfectly balanced
  - Pipelining needs overhead
  - Speedup less than number of stages.
- If we only have 3 consecutive load instructions
  - Non-pipelined needs 800 x 3 = 2,400 ps
  - Pipelined needs 1,400 ps
  - Speedup = 2,400/1,400 = 1.7
- If we have 1,000 consecutive load instrucitons
  - Non-pipelined needs 1000 x 800 = 800,000 ps
  - Pipelined needs 1 x 1000 + 999 x 200 = 200,800 ps
  - Speedup = 3.98

Much better, almost = 800/200, but still not equal to the number stages, why??

Pipelining and ISA Design (p. 265)

- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and 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

Pipelining Overview Recap

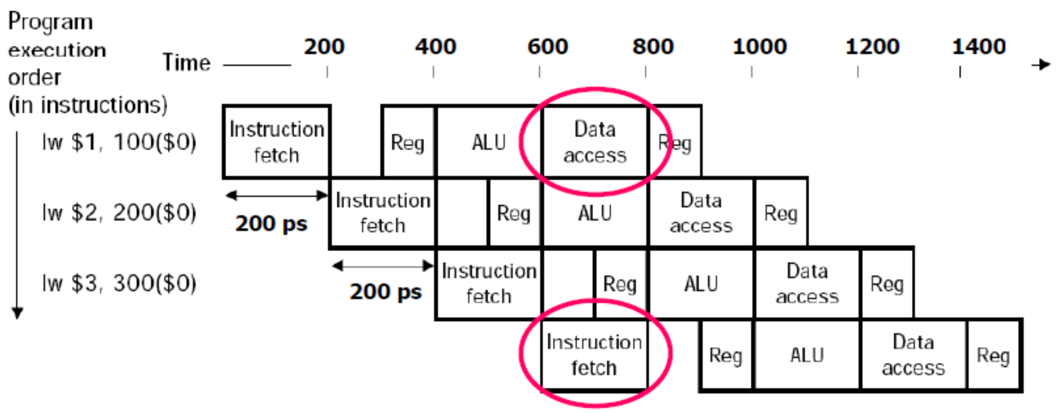
- What makes it easy
  - all instructions are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores
- What makes it hard?
  - structural hazards: suppose we had only one memory
  - control hazards: need to worry about branch instructions
  - data hazards: an instruction depends on a previous instruction
- We'll build a simple pipeline and look at these issues
- We'll talk about modern processors and what really makes it hard:
  - exception handling
  - trying to improve performance with out-of-order execution, etc.

Pipeline Hazards (p. 290)

- Situations that prevent starting the next instruction in the next cycle
- 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

Structural Hazards (P. 290)

- The hardware can not support the combination of instructions that we want to execute in the same clock cycle.
  - Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access to memory
  - Instruction fetch would have to stall for that cycle
    - Would cause a pipeline "bubble"



Structural hazard occurs if single memory!

- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

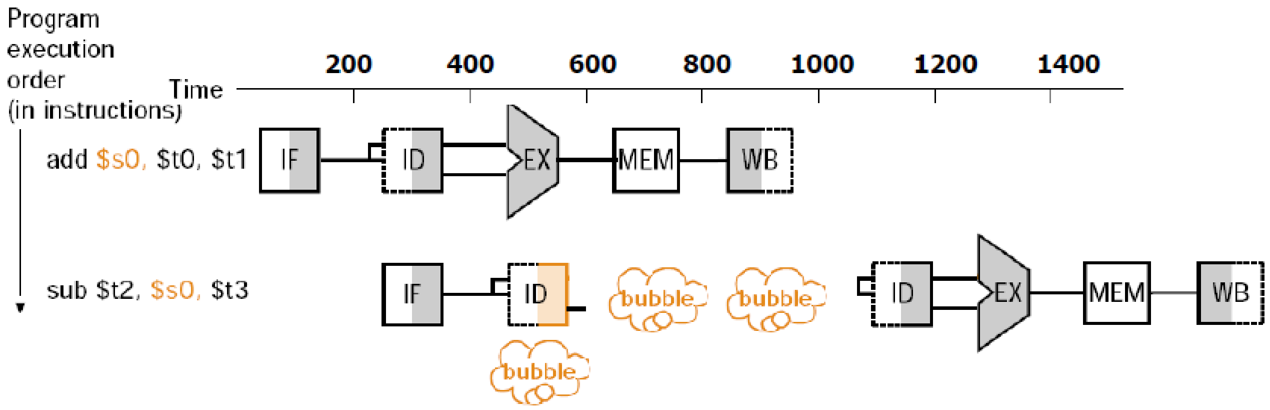
Data Hazards (P. 290)

- An instruction depends on the result of a previous instruction which is still in the pipeline
- Example

add $s0, $t0, $t1

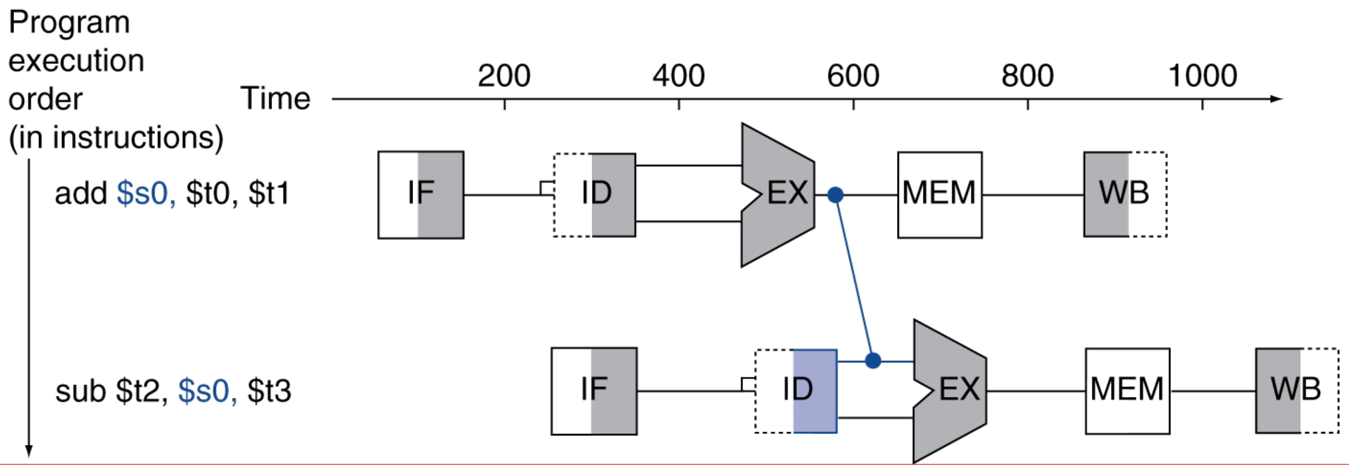
Sub $t2, $s0, $t3

  - the add instruction does not write its result until the 5th stage
  - We have to add three bubbles (stall for three cycles) to the pipeline



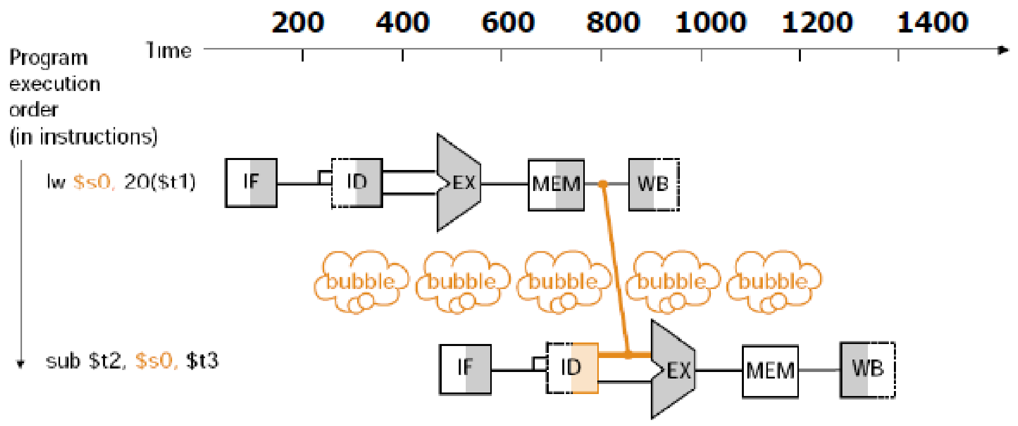
Solutions to Data Hazards ~ Forwarding (aka Bypassing) (p.291)

- We don’t have to wait for the instruction to complete (register write back) before trying to resolve the data hazard.
- As soon as the ALU creates the sum for add, we can supply it as input for the subtract
  - Requires extra connections in the datapath
- Getting the missing item early from the internal resources is called forwarding or bypassing.



Load-use Data Hazard (Fig. 4.30)

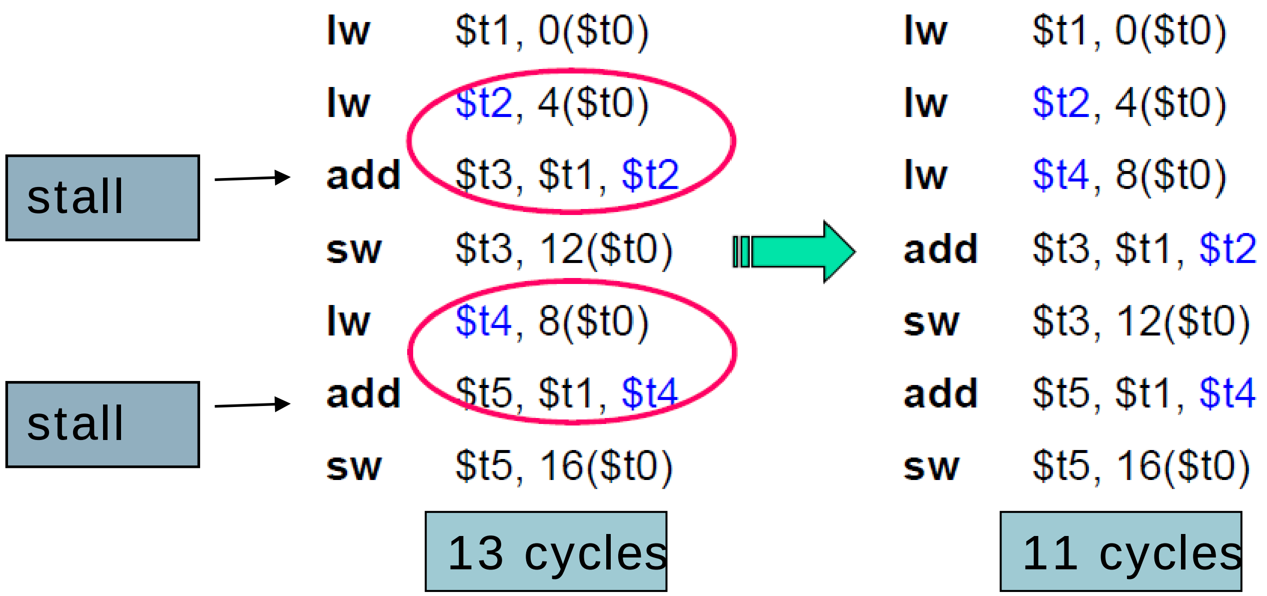
- Load-use data hazard
  - The data requested by a load instruction has not yet become available when it is requested
- Pipeline stall (Bubble)
  - A stall initiated in order to resolve a hazard
- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



Note: This figure is a simplification since we cannot know until after the subtract instruction is fetched and decoded

Reordering Code to Avoid Pipeline Stalls (P. 293)

- Sometimes we can reordering the code sequence to avoid pipeline stalls without affecting program behaviors.
- C code for A = B + E; C = B + F;
  - Assuming all variables are in memory and are addressable as offsets from $t0



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

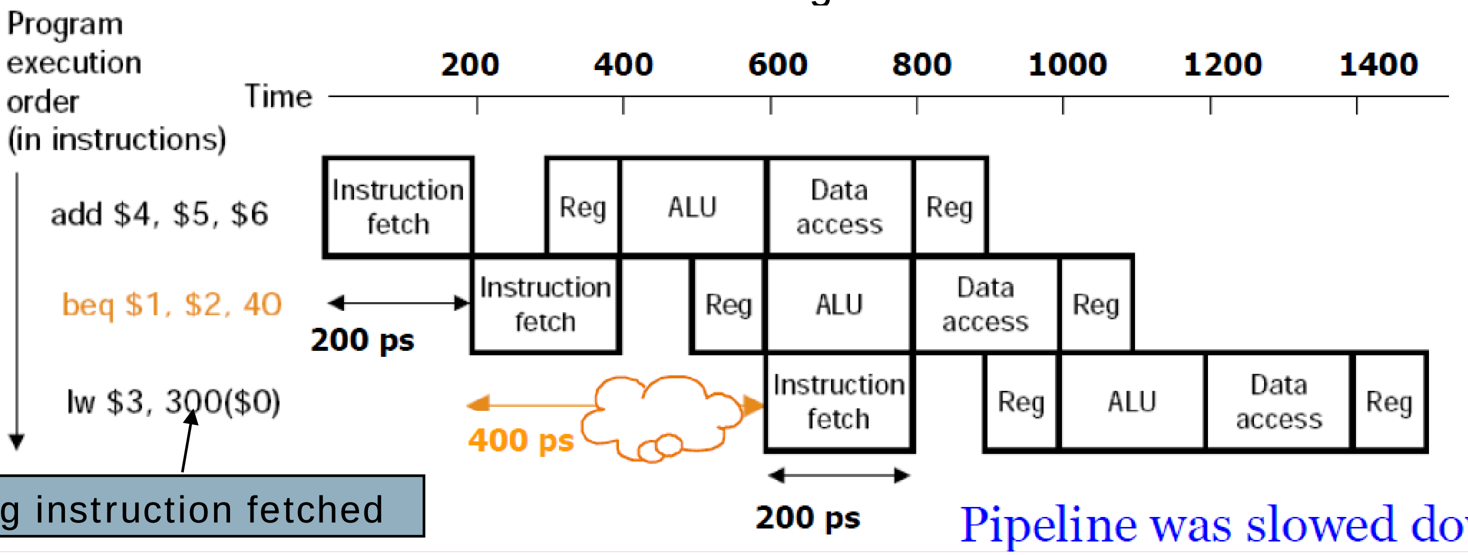
13 cycles

```
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

Control Hazards (Branch Hazards) (P. 294)

- 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

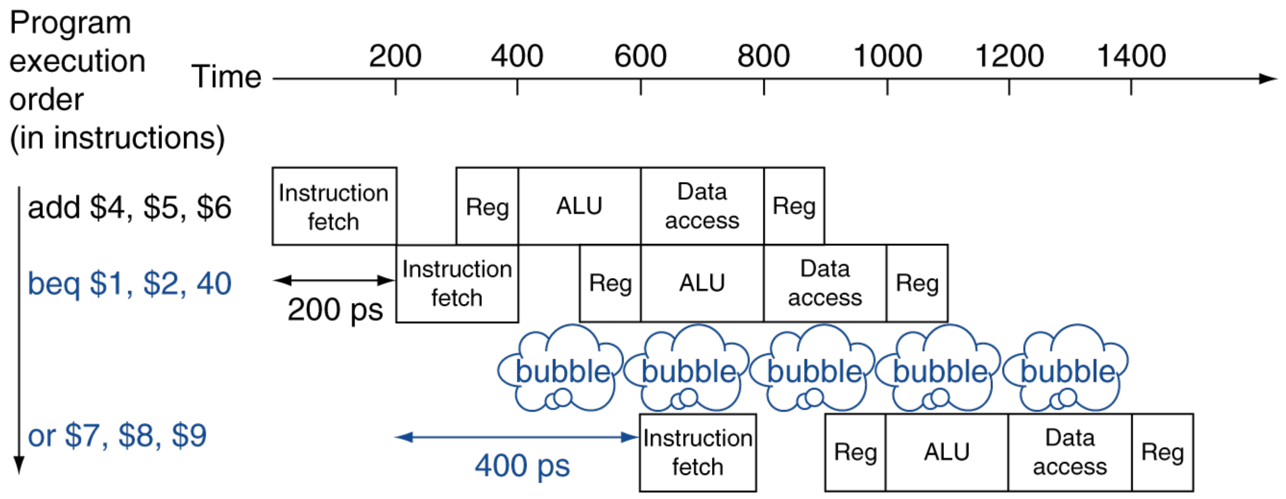


Wrong instruction fetched

Pipeline was slowed down!

Solutions to Control Hazards 1. Stall on Branch (p. 294)

- One solution – Stall
  - Wait until branch outcome determined before fetching next instruction
  - even if we add extra hardware in the 2nd stage so that the test registers (condition), but we still need stall for one cycle.

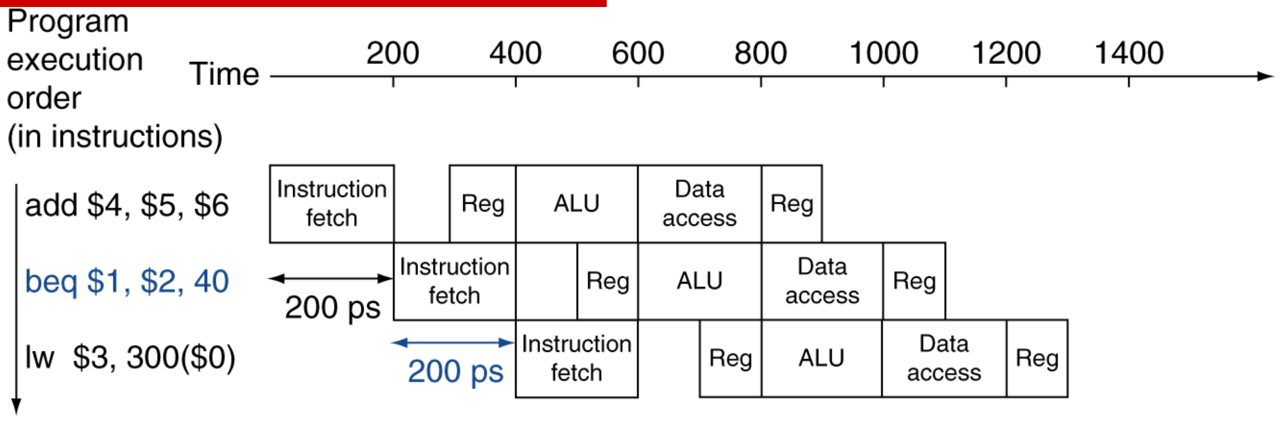


Solutions to Control Hazards 2. Branch Prediction (P. 295)

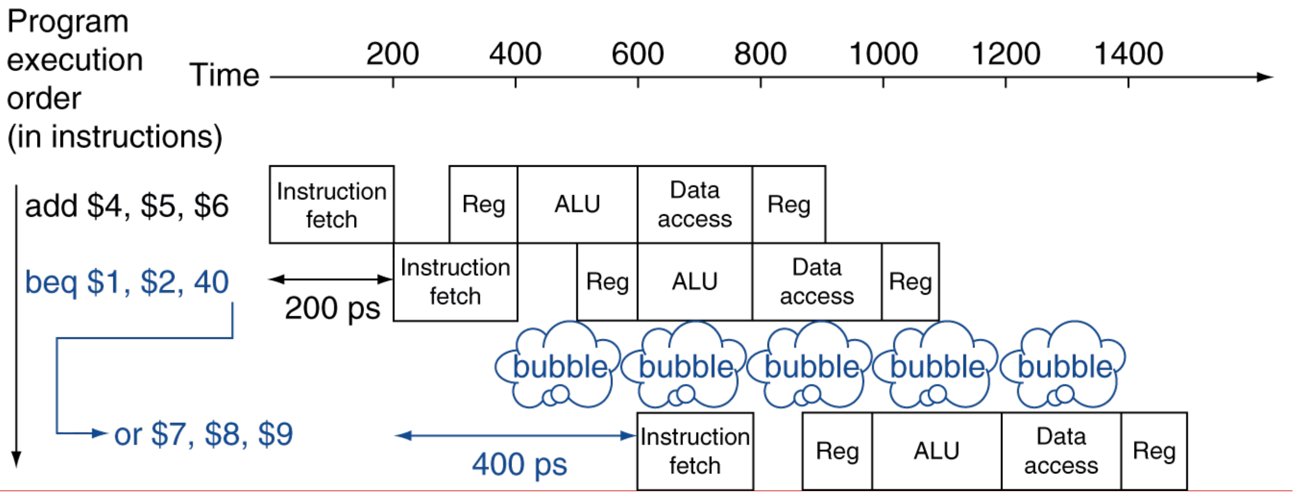
- Longer pipelines can’t readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken (Fig. 4.32)

Prediction correct



Prediction incorrect



More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

Pipeline Summary

The BIG Picture

- 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