Midterm Exam

- 12:00 – 13:20, October 19 (Monday), 2009
- #330110 (odd student id)
- #330118 (even student id)

- Scope:
  - Chap. 1 – 4.4
- Closed-book exam

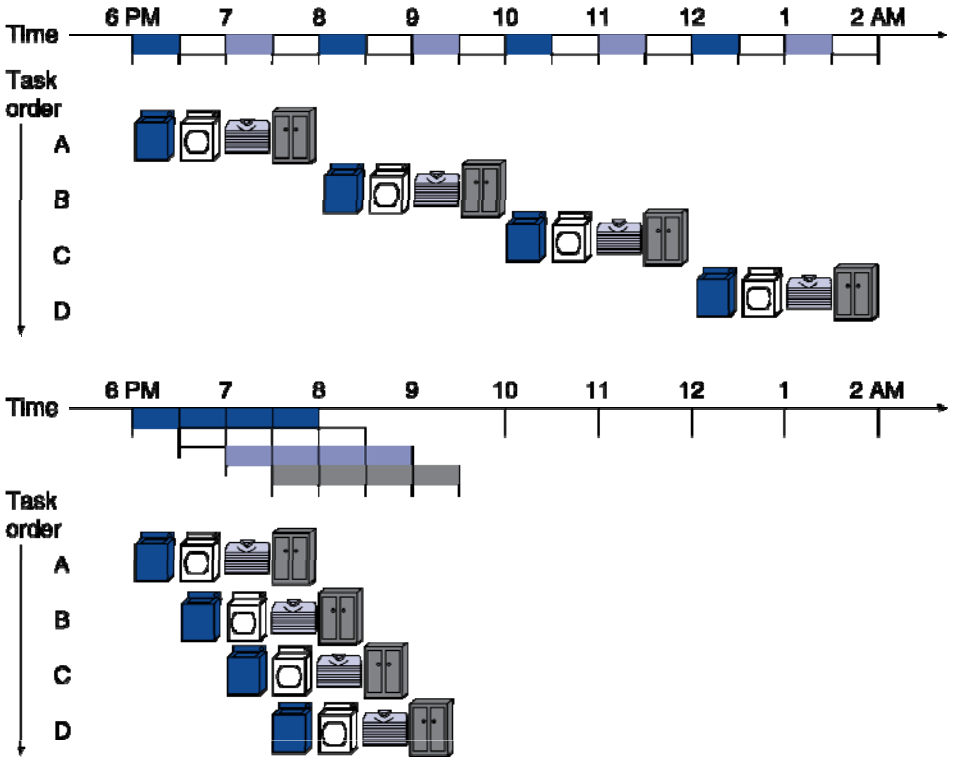
Pipelining

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

A Real Pipelining Example

Pipelining Analogy

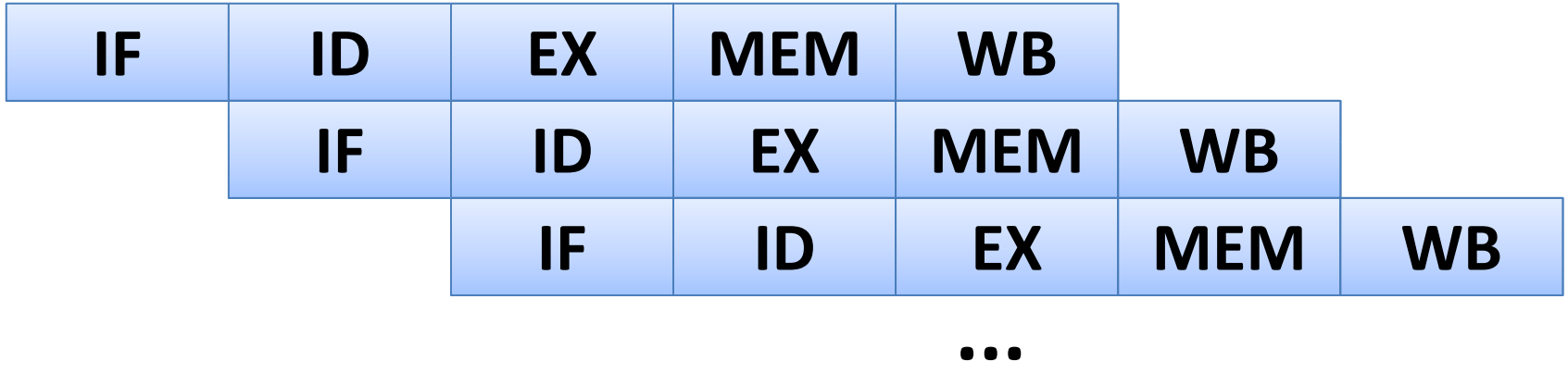
- **Pipelined laundry: overlapping execution**
  - Parallelism improves performance



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

MIPS Pipeline

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



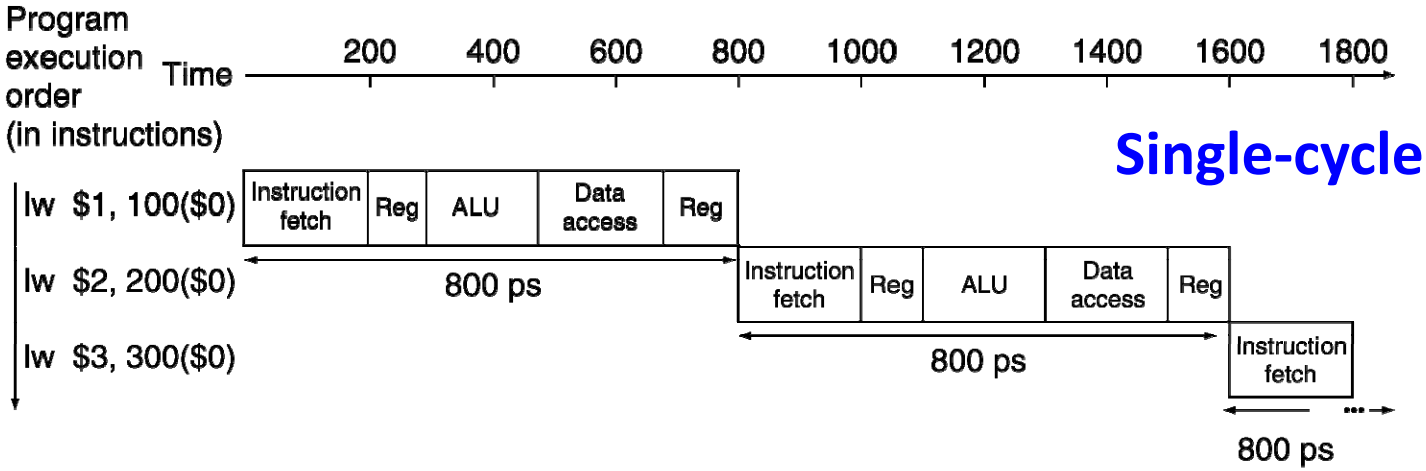
Pipeline Performance (1)

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

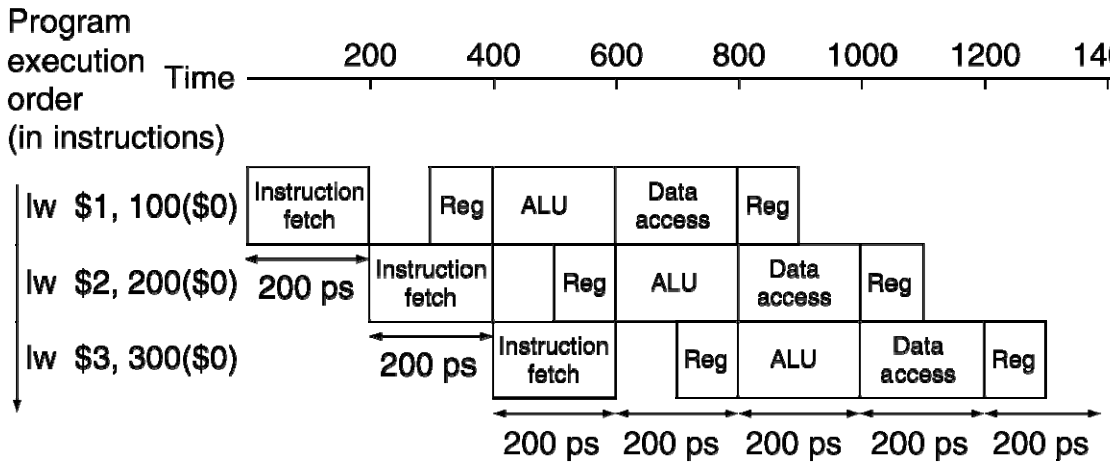
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
`lw`	200ps	100 ps	200ps	200ps	100 ps	800ps
`sw`	200ps	100 ps	200ps	200ps		700ps
`R-format`	200ps	100 ps	200ps		100 ps	600ps
`beq`	200ps	100 ps	200ps			500ps

Pipeline Performance (2)

- Single-cycle datapath vs. Pipelined datapath



Single-cycle: Tc = 800ps



Pipelined: Tc = 200ps

Pipeline Performance (3)

- **Pipeline speedup**
  - If all stages are balanced (i.e., all take the same time):

$$Time\ between\ instructions_{pipelined} = \frac{Time\ between\ instructions_{nonpipelined}}{Number\ of\ stages}$$

  - If 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 and decode in one cycle
    - (cf.) 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

- **What are 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

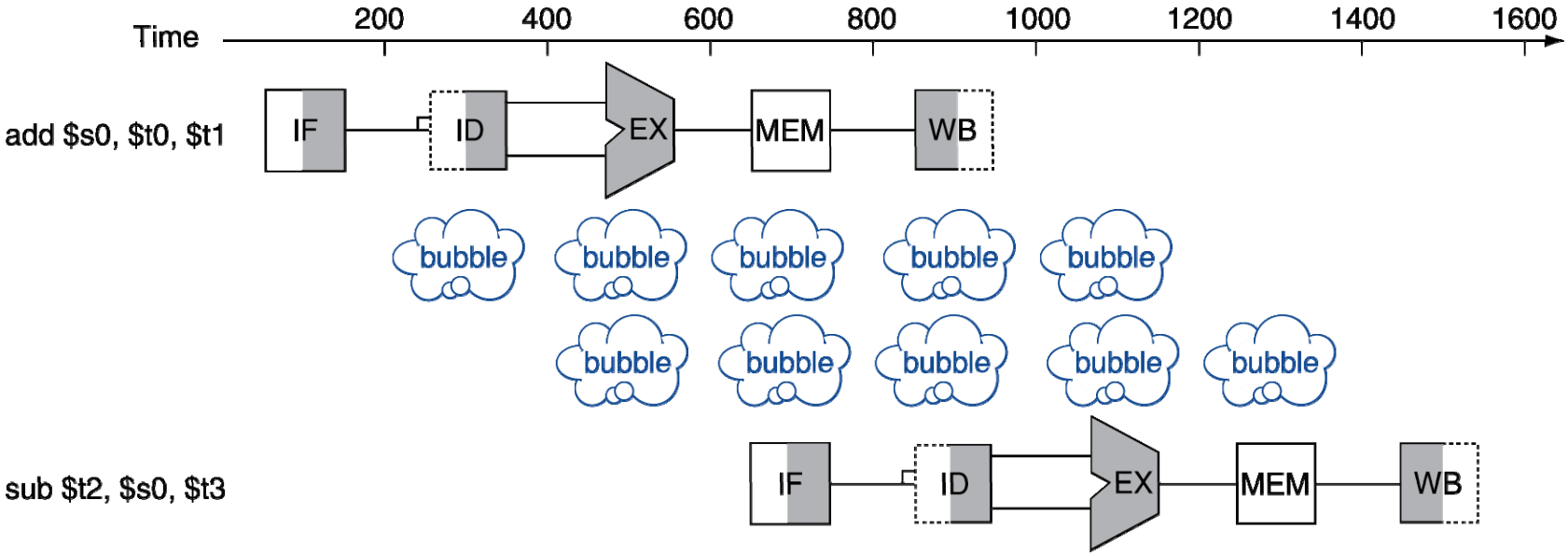
Structure Hazards

- **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

Data Hazards (1)

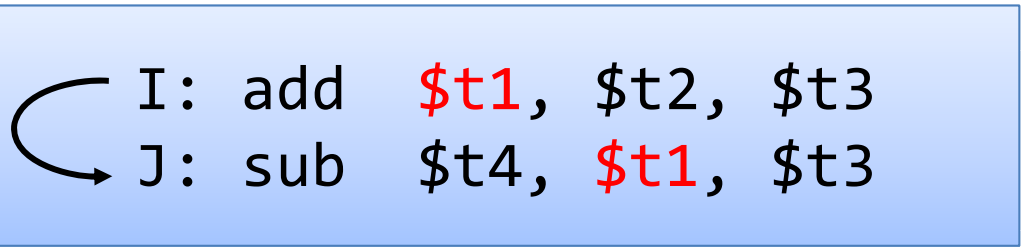
- An instruction depends on completion of data access by a previous instruction

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



Data Hazards (2)

- **Read After Write (RAW)**
  - Inst `J` tries to read operand before Inst `I` writes it:



  - Caused by a "(true) **dependence**". This hazard results from an actual need for communication.

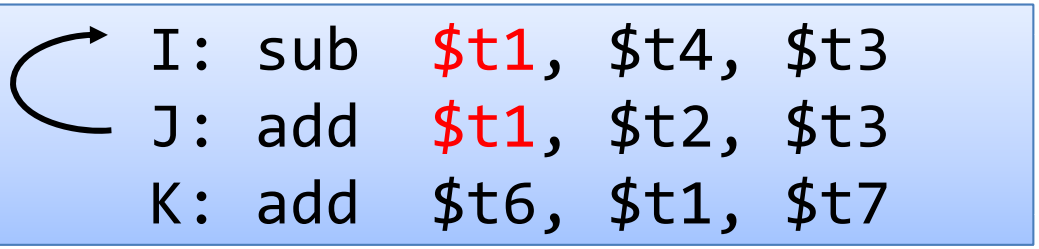
Data Hazards (3)

- **Write After Read (WAR)**
  - Inst `J` writes operand before Inst `I` reads it:

  - Called an "**anti-dependence**" by compiler writers. This results from the reuse of the name "`$t1`".

Data Hazards (4)

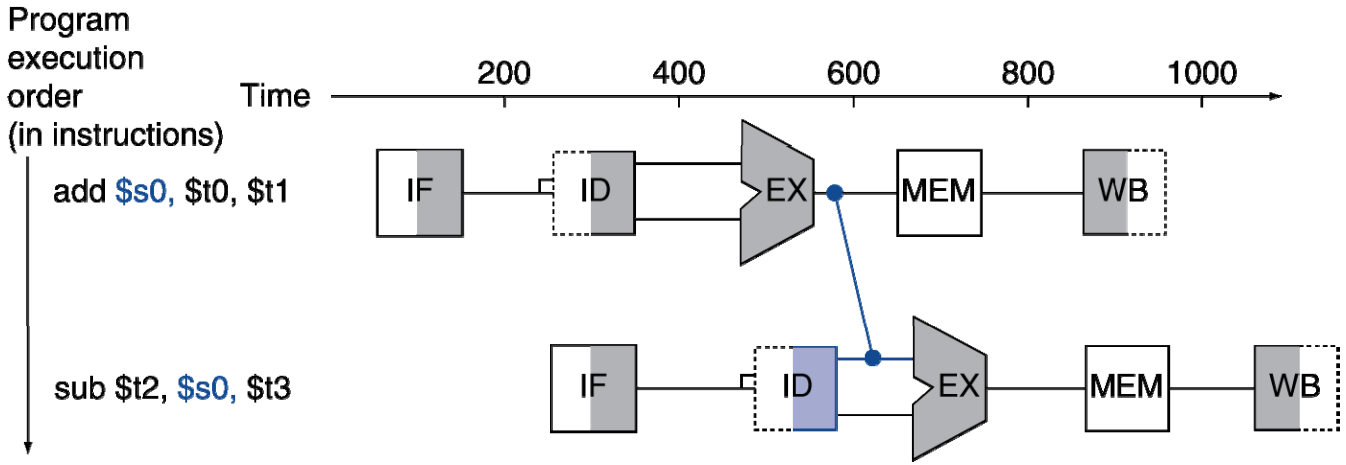
- **Write After Write (WAW)**
  - Inst `J` writes operand before Inst `I` writes it:



  - Called an "**output dependence**" by compiler writers. This also results from the reuse of the name "`$t1`".

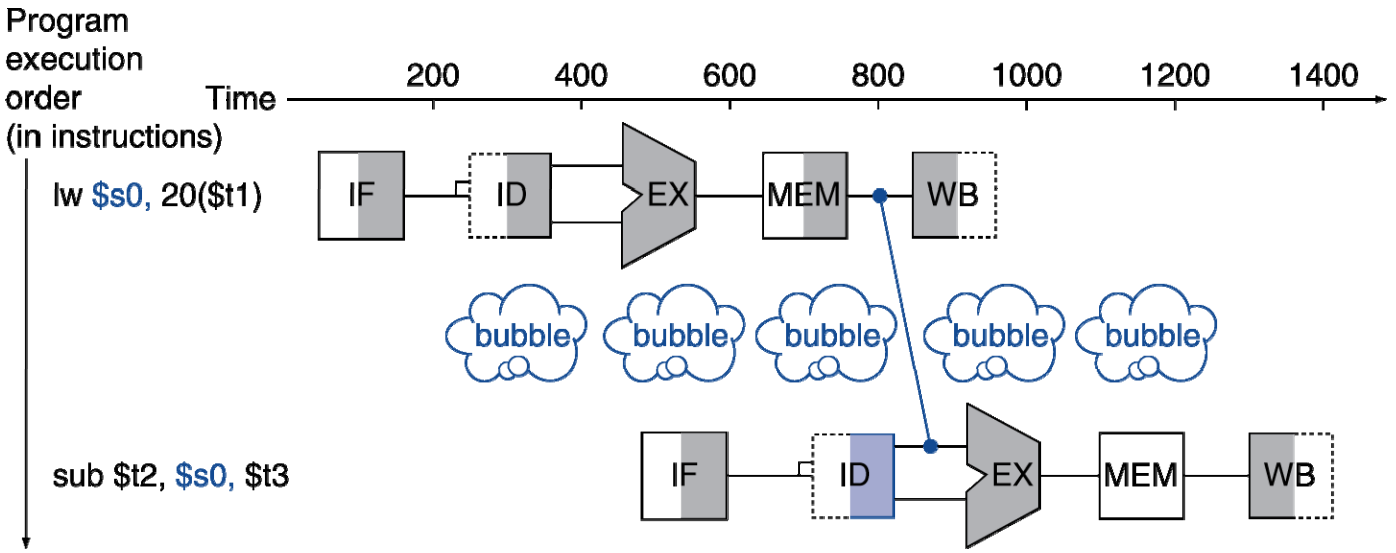
Data Hazards (5)

- **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



Data Hazards (6)

- **Load-Use data hazards**
  - Can't always avoids stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



Data Hazards (7)

- **Code scheduling to avoid stalls**
  - Reorder code to avoid use of load result in the next instruction

```
(C code)   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)
```

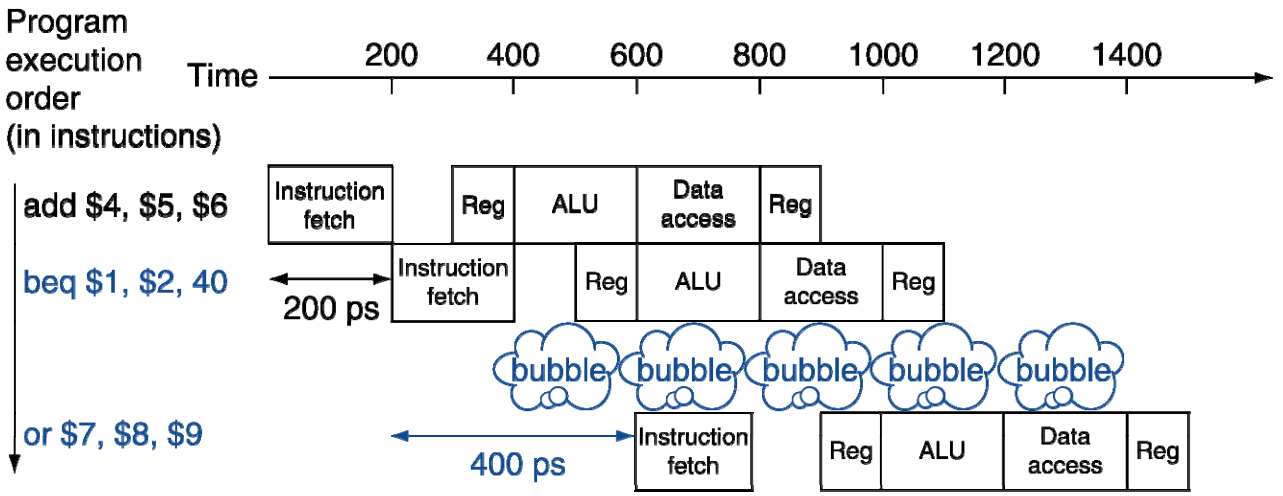
11 cycles

Control Hazards (1)

- **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

Control Hazards (2)

- **Stall on branch**
  - Wait until branch outcome determined before fetching next instruction



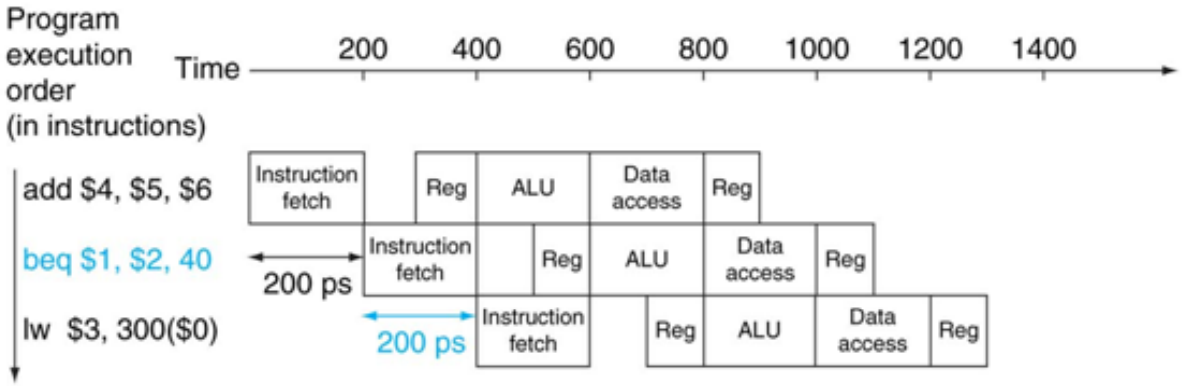
Control Hazards (3)

- **Branch prediction**
  - 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

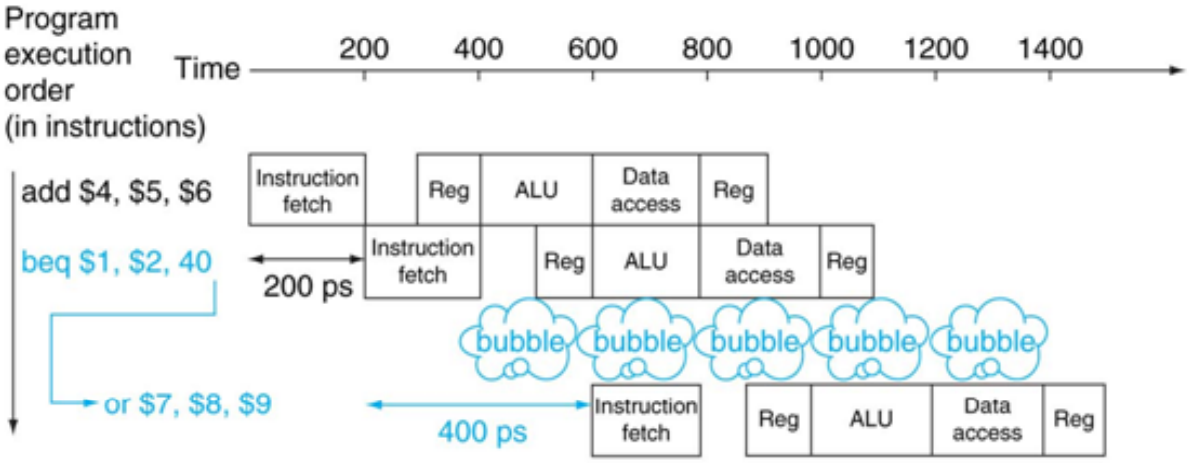
Control Hazards (4)

- MIPS with "Predict Not Taken"

Prediction correct



Prediction incorrect



Control Hazards (5)

- **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

Summary

- **Pipelining**
  - 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