Pipelining

CS6133 - Computer Architecture I

Vikram Padman

Polytechnic Institute of New York University

vikram@poly.edu

Agenda

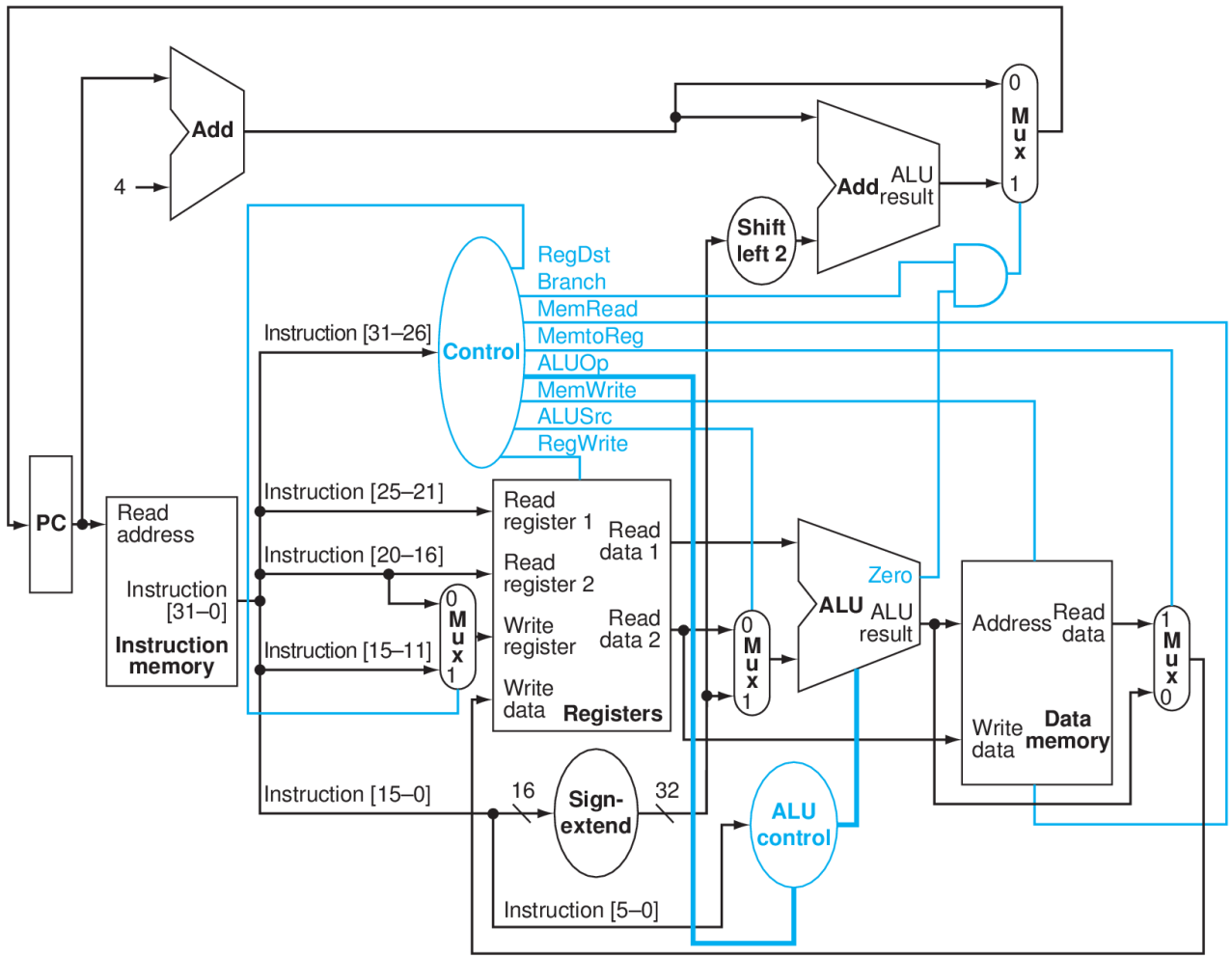
1. **What is CPU Pipelining?**
2. **Simple CPU**
   1. Instruction fetch **IF**
   2. Instruction decode/register fetch **ID**
   3. Execution/effective address cycle **EX**
   4. Memory Access **MEM**
   5. Write-back cycle **WB**
3. **Pipelined CPU**
4. **Hazards**
   1. Structural
   2. Data
   3. Control
5. **Activity**

What is Pipelining?

Pipelined CPU

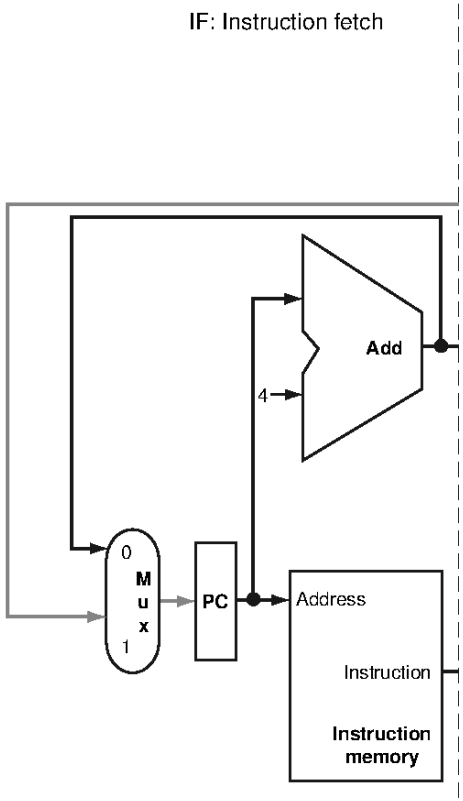
- "Pipelining is an implementation technique whereby multiple instructions are overlapped in execution."
- Case for pipelining a CPU:
  1. An instruction is executed by many stages within a CPU, sequentially.
  2. In an unpiplined CPU only one stage is active at any given clock cycle.
  3. Pipelining increases CPU's efficiency dramatically by executing subsequent instruction at every clock cycle.
  4. In a pipelined CPU every stage could be active every clock cycle.

Simple CPU
Un-Pipelined



From "Computer Organization and Design" page 322

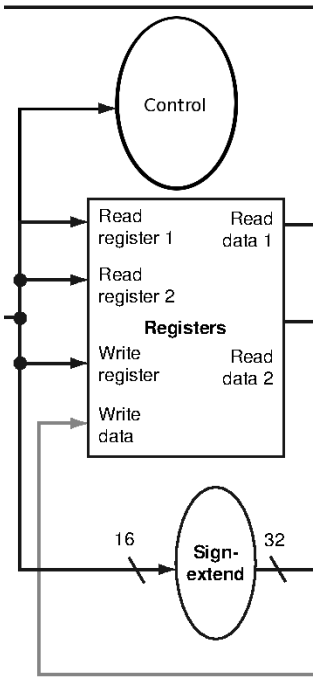
Instruction Fetch Stage **IF**



- Send PC to Instruction Memory and fetch a new instruction
- Increment PC by 4 or load PC

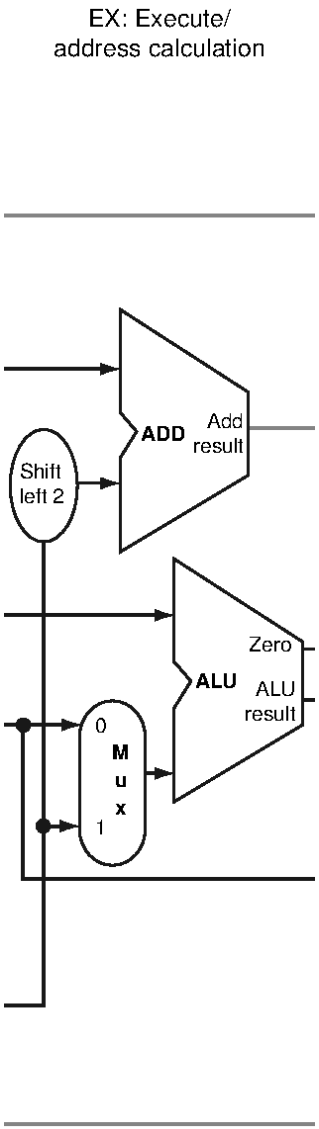
Instruction Decode/Register Fetch Stage **ID**

ID: Instruction decode/ register file read



- The Control Unit decodes the instruction
- The Register file reads source operands as specified in the instruction
- Sign-extend the offset field

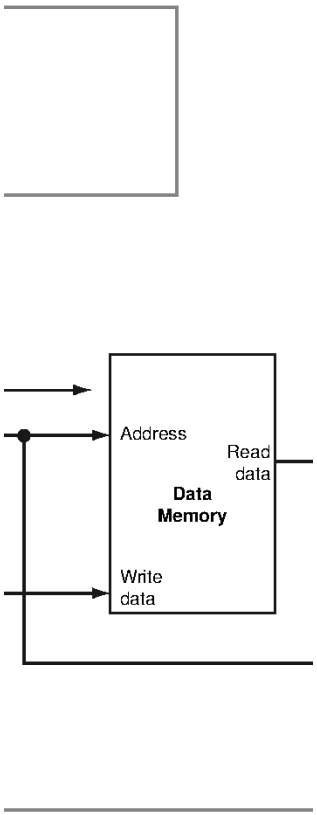
Execution Stage **EX**



- Calculate new PC by adding sign-extended offset to current PC, in case of branch instruction
- The ALU performs one of the following operation:
  1. Memory reference - ALU adds the base register and sign-extended offset to form the effective address
  2. Register-Register - ALU operates on operands as specified by ALU-Opcode
  3. Register-Immediate - ALU uses the first operand and sign-extended offset to perform operation specified by ALU-Opcode

Memory **MEM**

MEM: Memory access

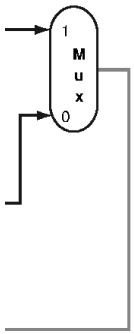


- The memory either write or output data from the effective address calculated in the previous stage
- For a store instruction causes the memoru to store data present in “write data”

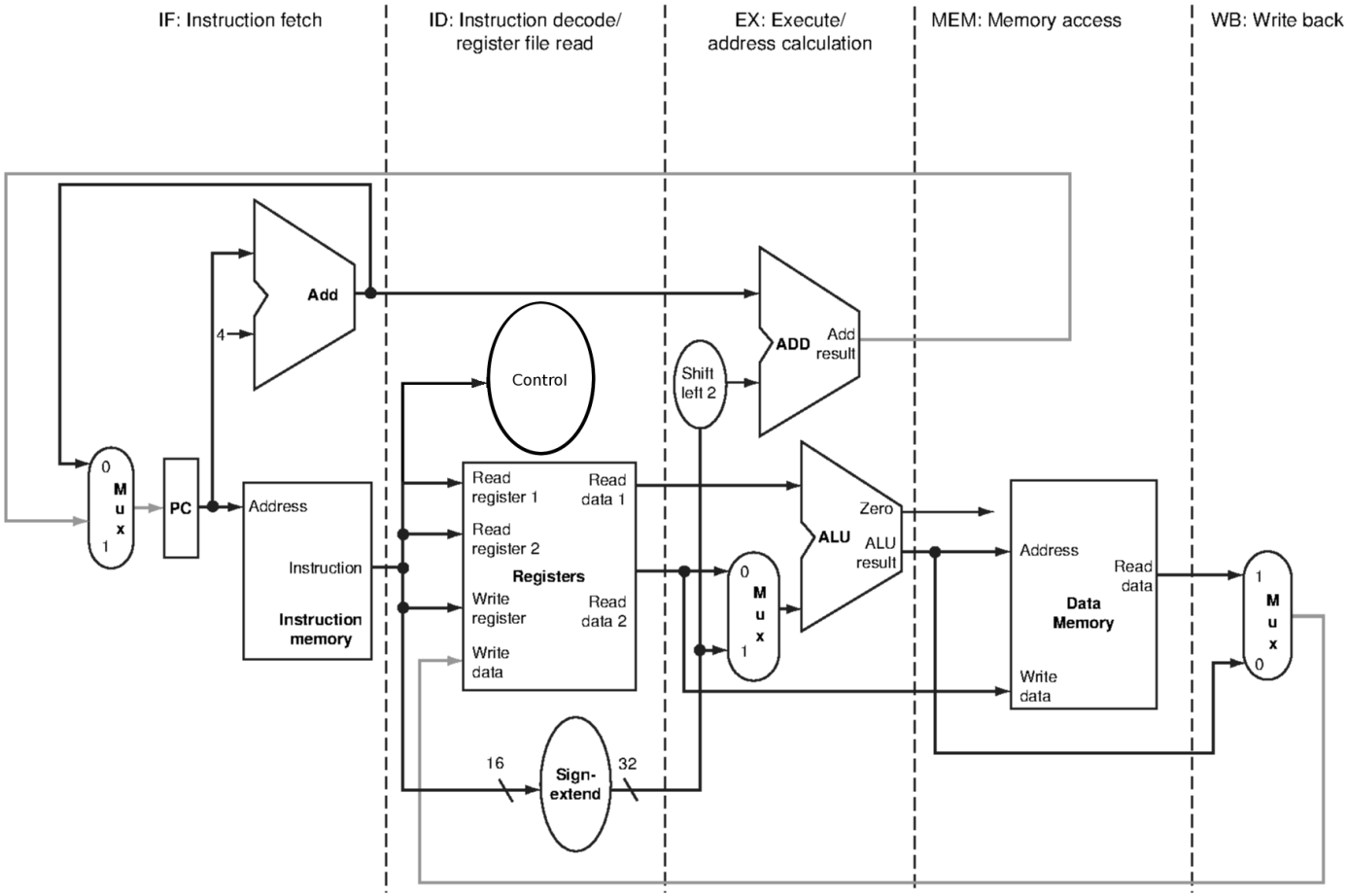
Write Back **WB**

WB: Write back

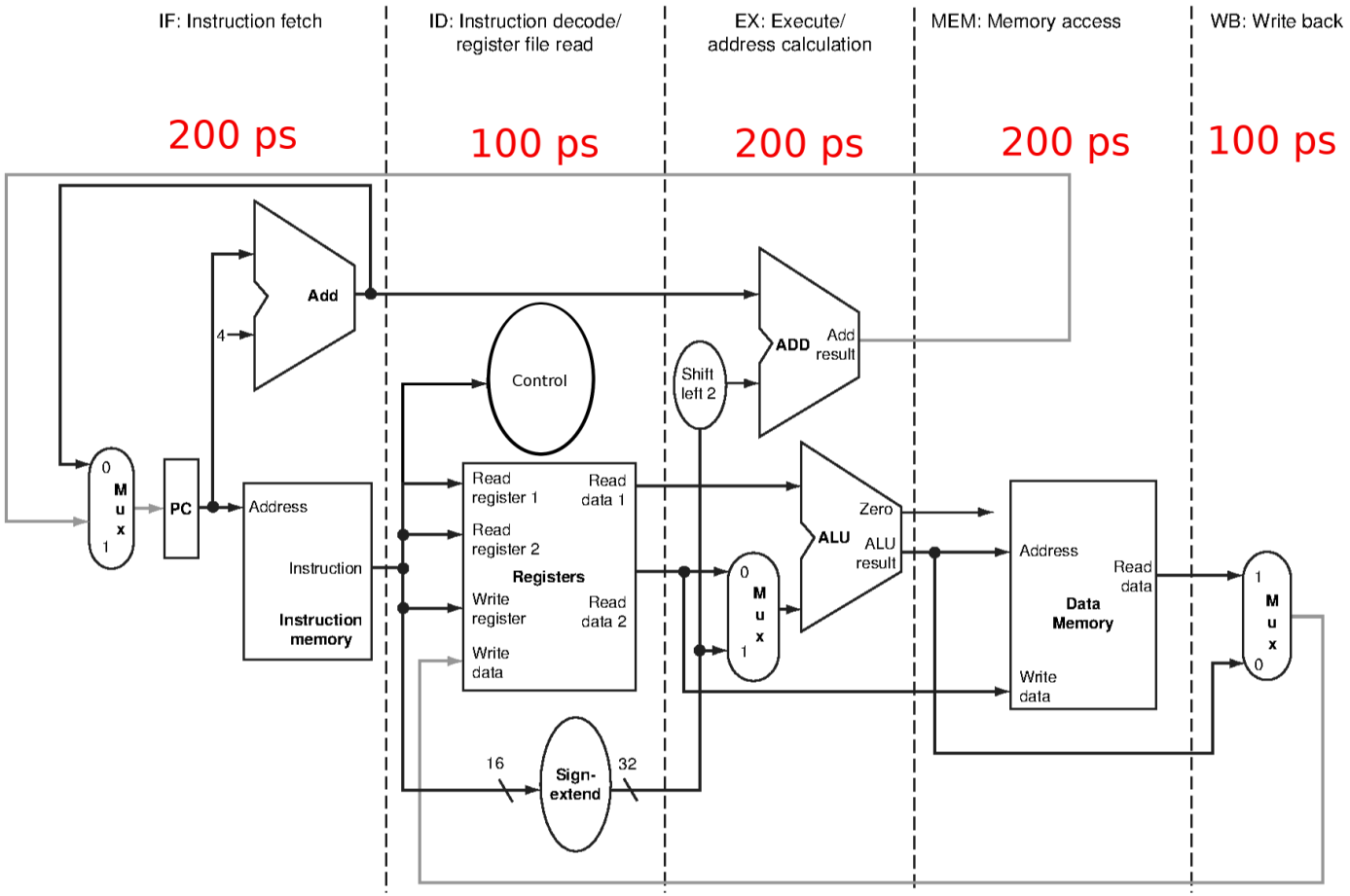
- Data results from R-type (ALU) or load instruction is written back to register file.



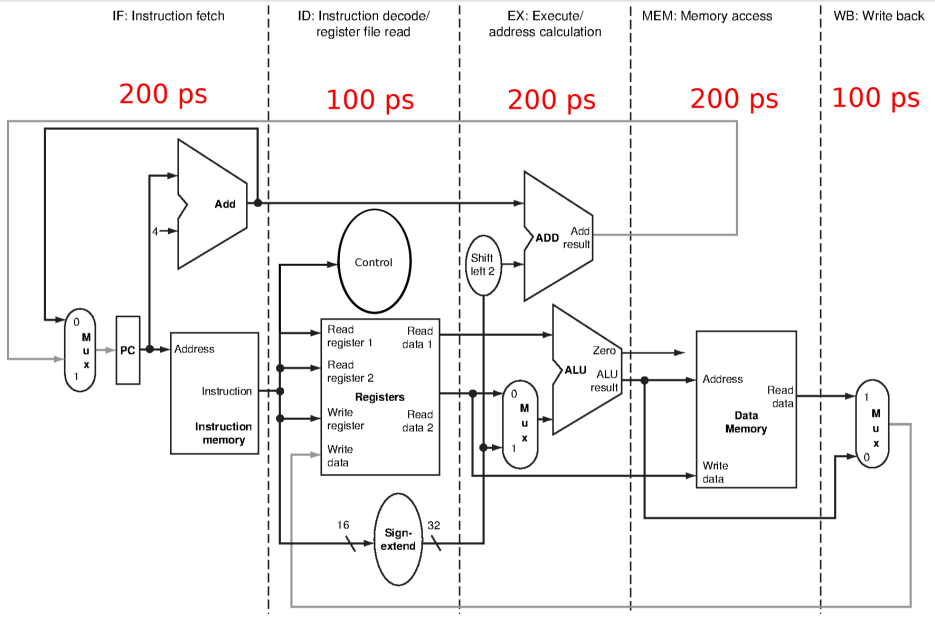
Simple CPU's Pipeline Boundaries

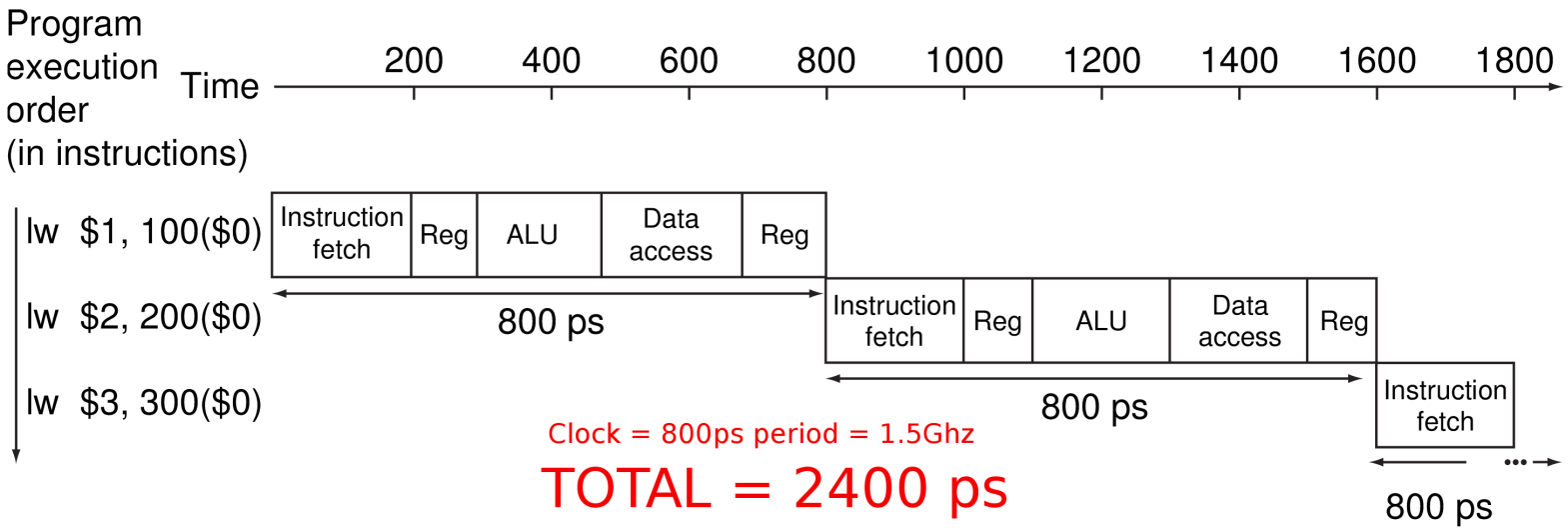


Performance of Simple CPU



Performance of Simple CPU

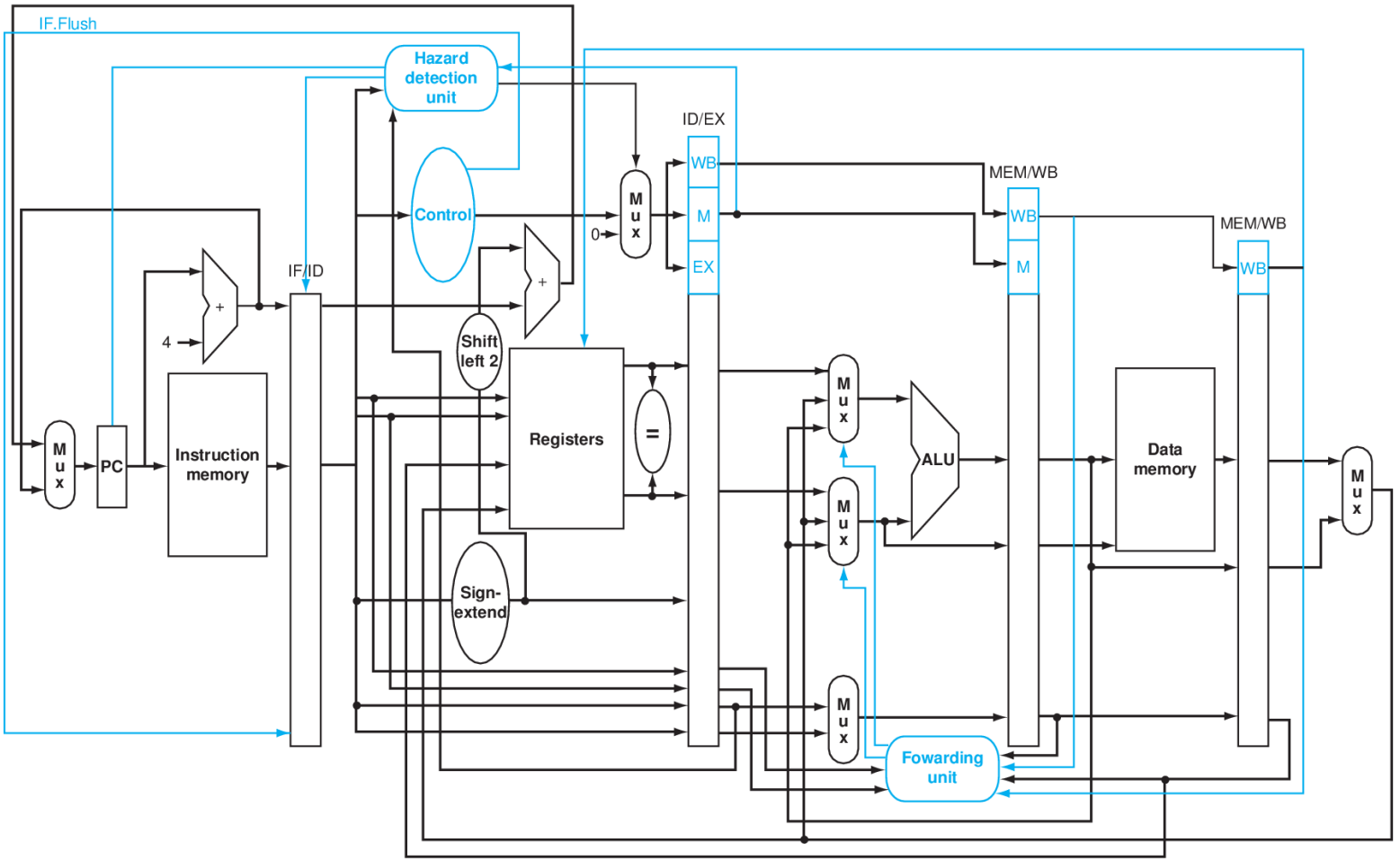




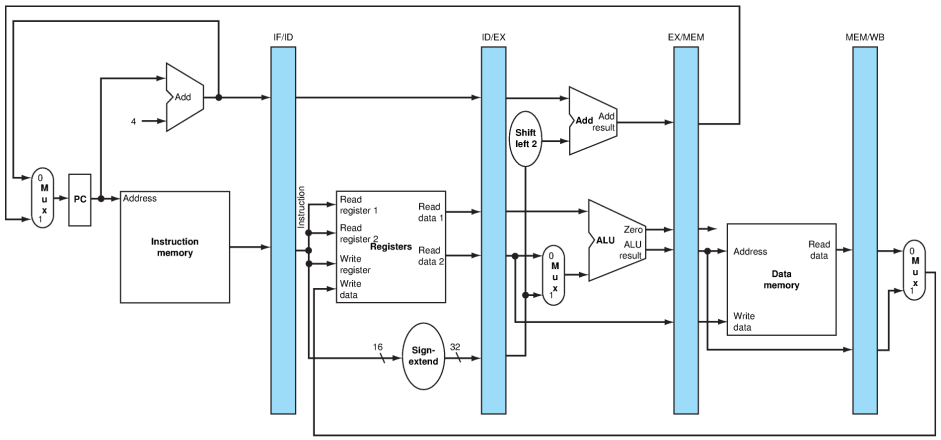
Clock = 800ps period = 1.5Ghz

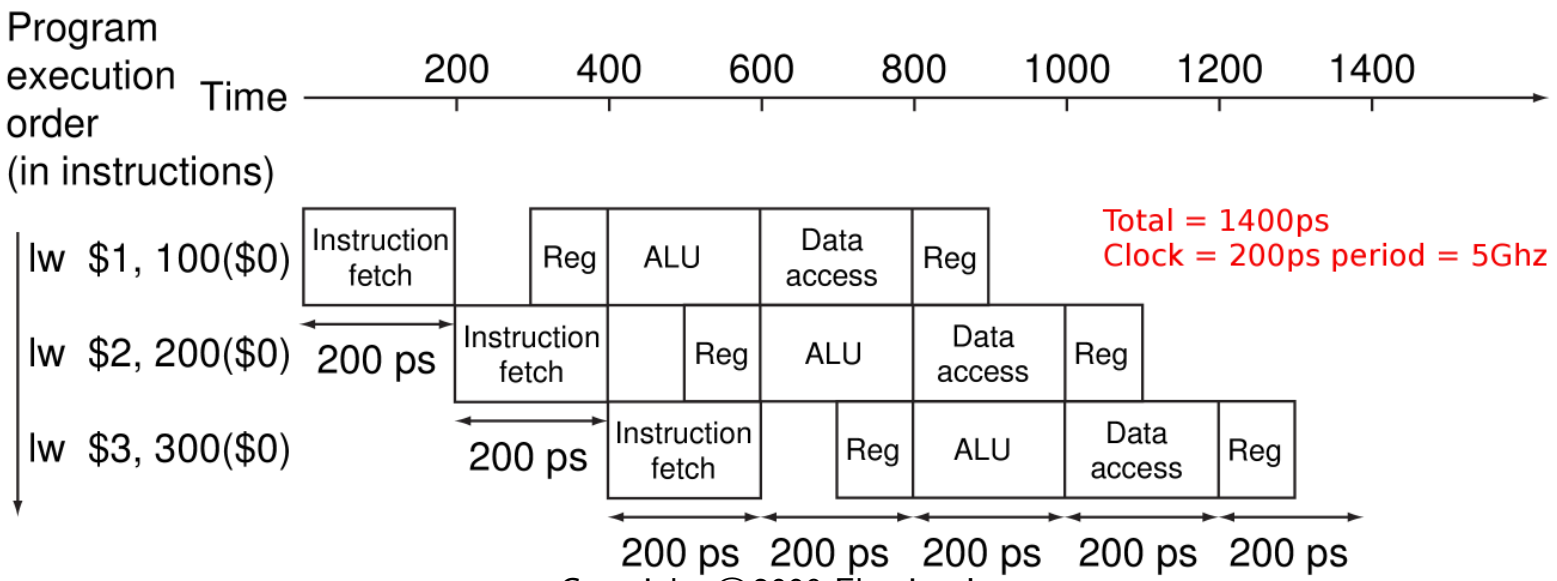
TOTAL = 2400 ps

Pipelined CPU



Performance of Pipelined CPU





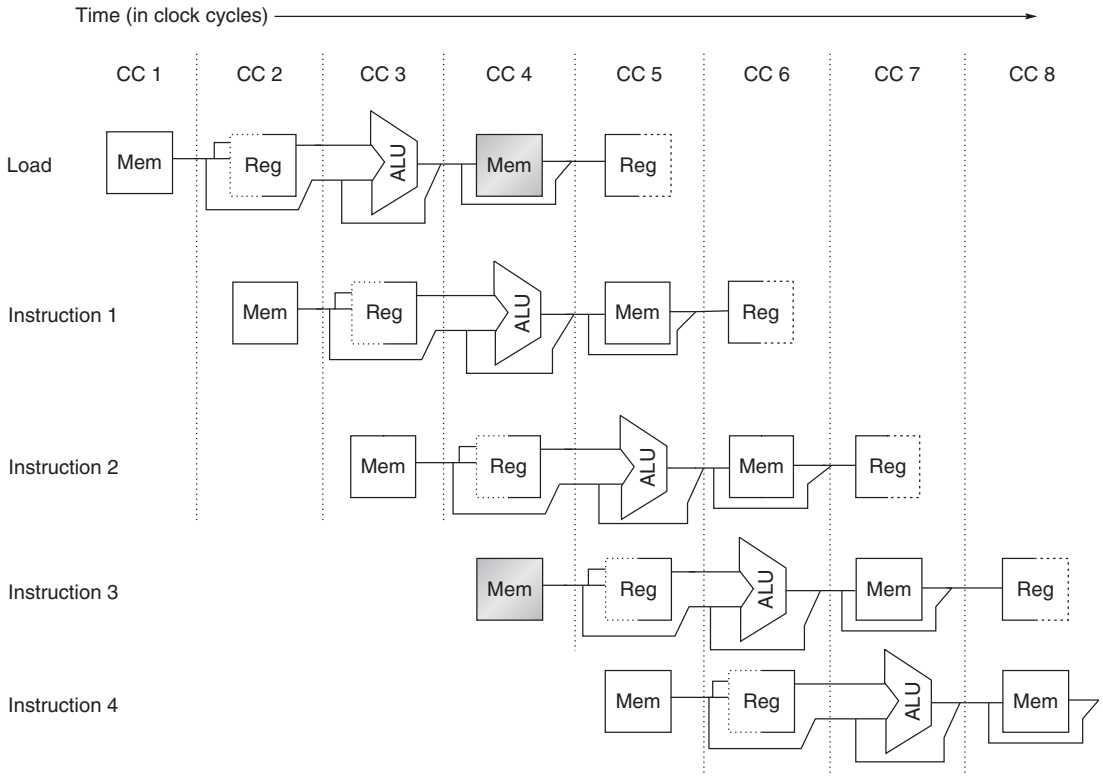
Hazards

- *Hazards* are conditions that prevents the execution of an instruction in its designated clock cycle. Hazards could reduces the performance gained from pipeling and could be categorized into three major types:
  1. **Structural hazards** arise due to resource conflicts
  2. **Data hazards** due to overlapping instructions
  3. **Control hazards** due to flow altering instructions (Branch/Jump)
- Stalls or "NOPs" are injected into the pipeline to mitigate hazards
- Compilers re-organize the instruction stream to mitigate hazards
- Finally, the pipeline by itself could be re-organized or redesigned with additional hardware to prevent hazards and stalls

Hazards

Structural Hazards

- Typically caused due to resource constraints in a CPU that is not fully pipelined.
- Assume there was only one memory, instead of two:

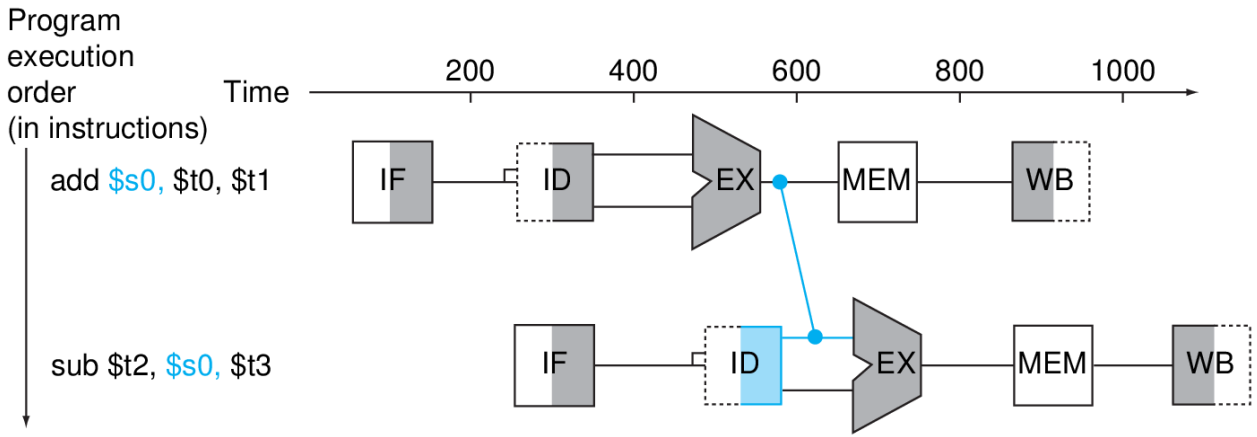


Hazards

Structural Hazards

- *Structural Hazards* are generally fixed by adding or modifying components in a CPU
- In situations were power, size and/or application requirements prevent additional hardware. Software assistance is necessary to inject "NOPs" or re-order instructions to prevent structural hazards.

- *Data Hazards* occurs when there are data dependencies within instructions that are in the pipeline. For example:
  1. add $s0, $t0, $t1
  2. sub $t2, $s0, $t3

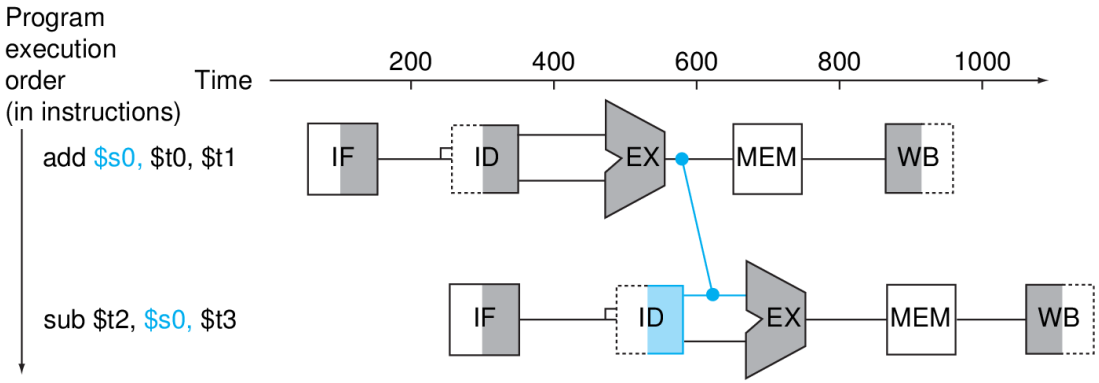


Hazards
Data Hazards

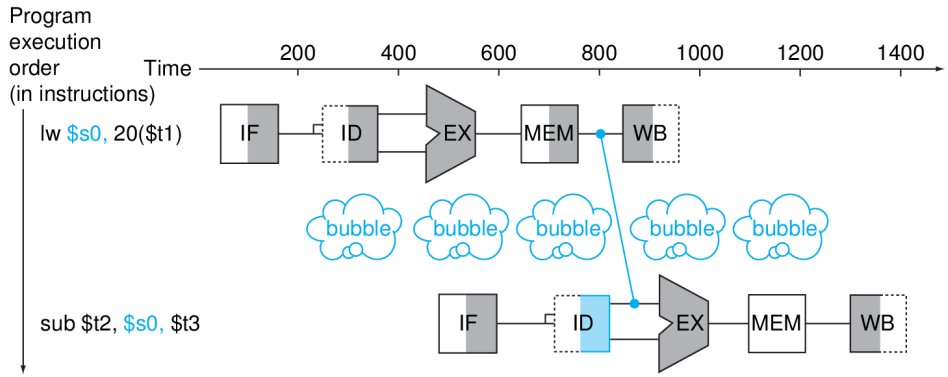
Data Hazards could be solved by:

1. **Forwarding** A hardware module that allows data to bypass some modules.
2. **Stalls/NOPs** Injecting NOPs into the pipeline
3. **Instruction Reordering** Either a compiler or CPU itself re-order instruction to mitigate data hazards

- **Forwarding :**



- **Stall/NOPs :**

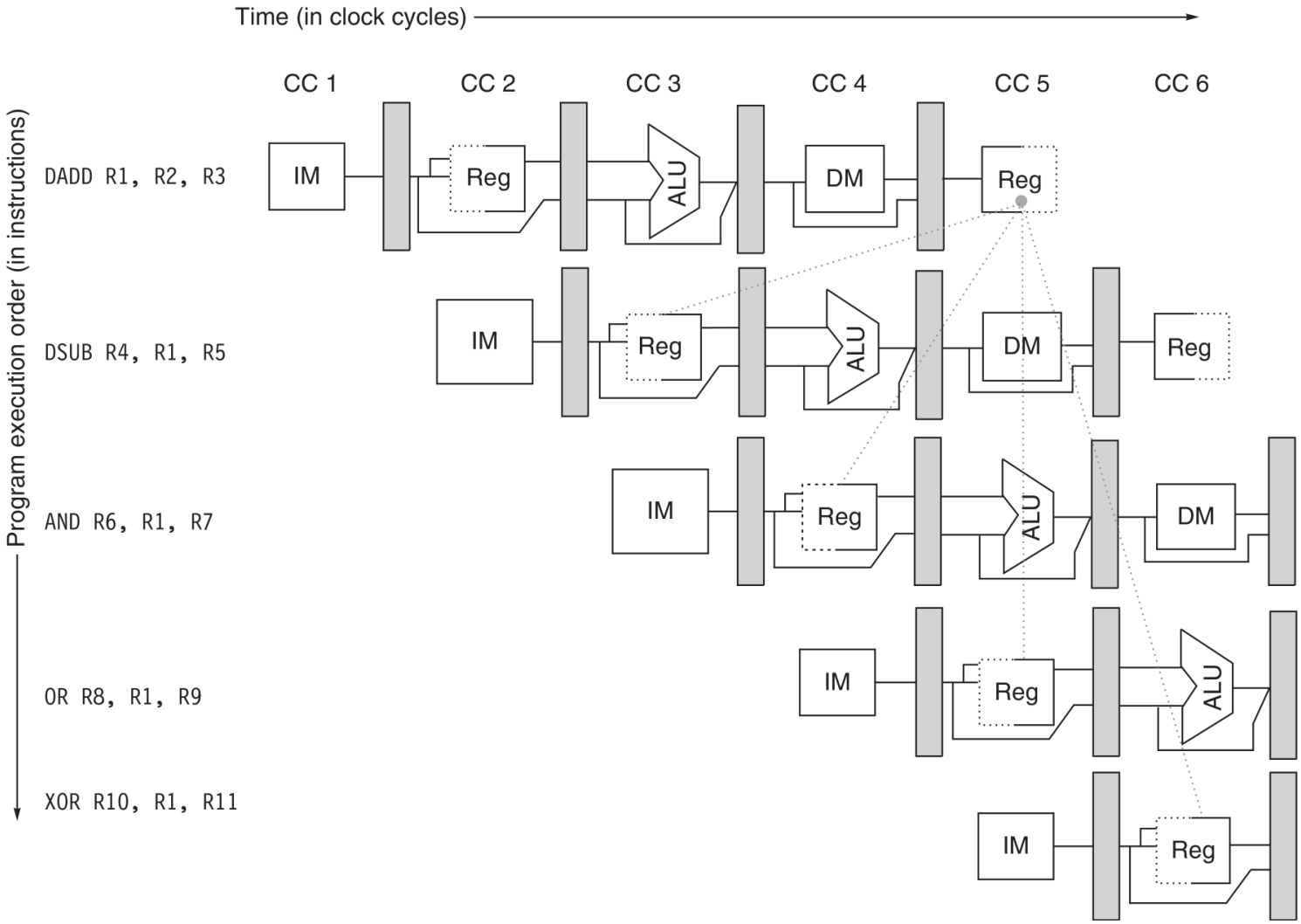


- **Instruction Reordering :**
  1. lw $s0, 20($t1)
  2. sub $t2, $s0, $t3
  3. add $s5, $s8, $t6

  1. lw $s0, 20($t1)
  2. add $s5, $s8, $t6
  3. sub $t2, $s0, $t3

Hazards

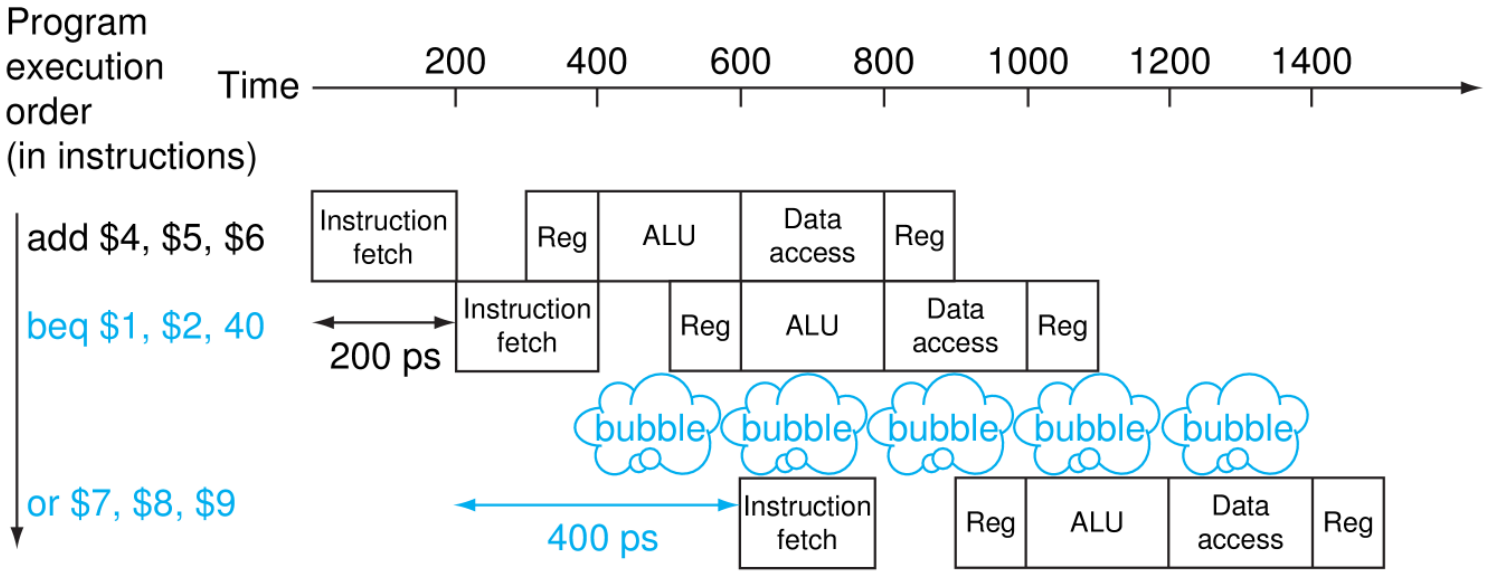
Data Hazards



Hazards

Control Hazards

- *Control Hazards* are more complex and expensive than the previous once.
- Expensive in terms of performance penalties it causes and implementation complicity to mitigate control hazards.
- This hazard occurs from the fact that a branch or Jump depends on the result of another instruction



Week 10 Activity 1

1. Read section 2 in Appendix C(Ed. 5) or A (Ed. 4) in "Computer Architecture - A Quantitative Approach" and answer the following:
    1. How could the effects of control hazards be minimized by reordering instructions?
    2. Describe static and dynamic branch predication techniques.
    3. How could a compiler influence the branch predication hardware in a CPU?