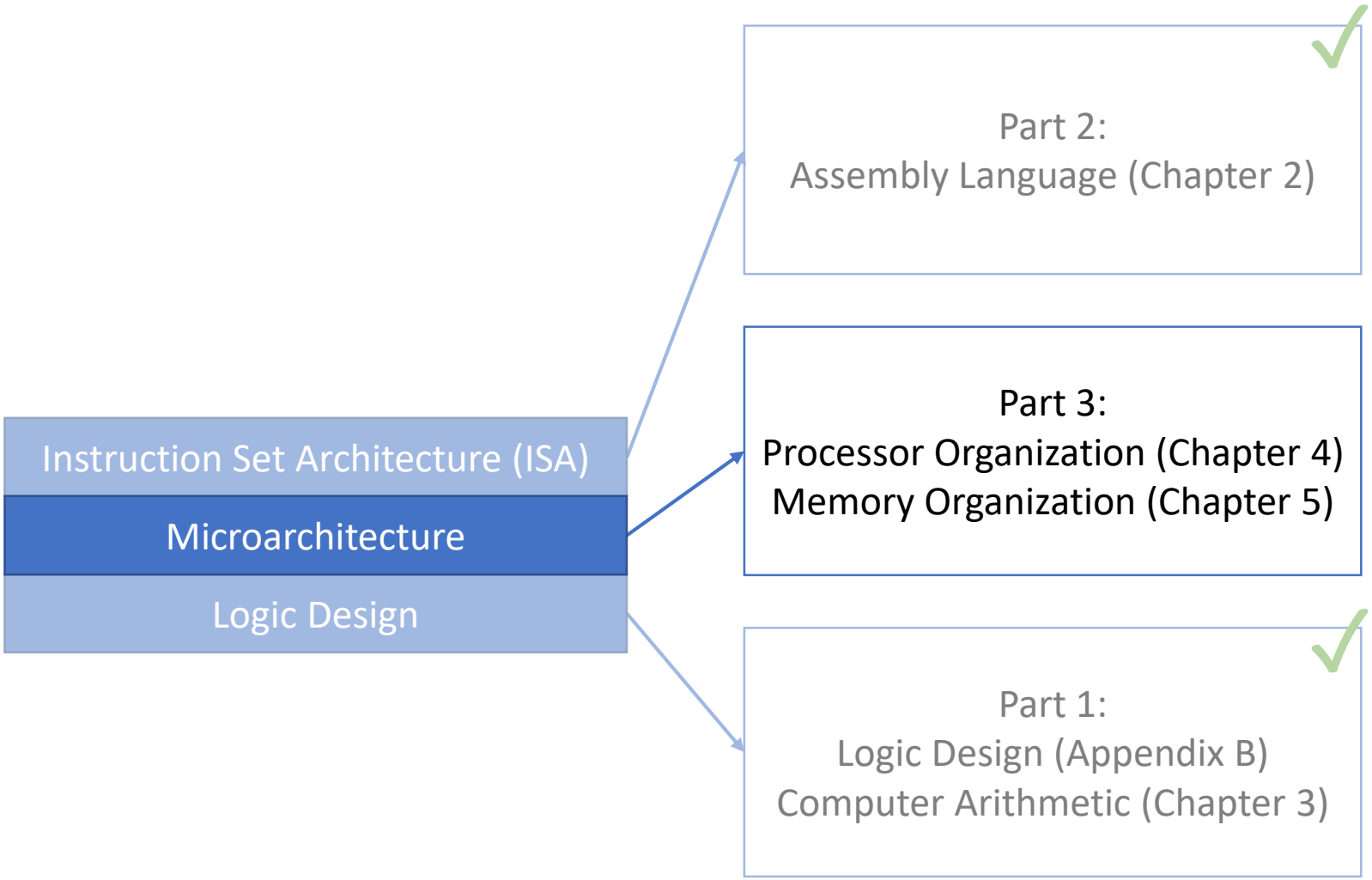
Lecture 24: Pipeline Hazards

CMPS 221 – Computer Organization and Design

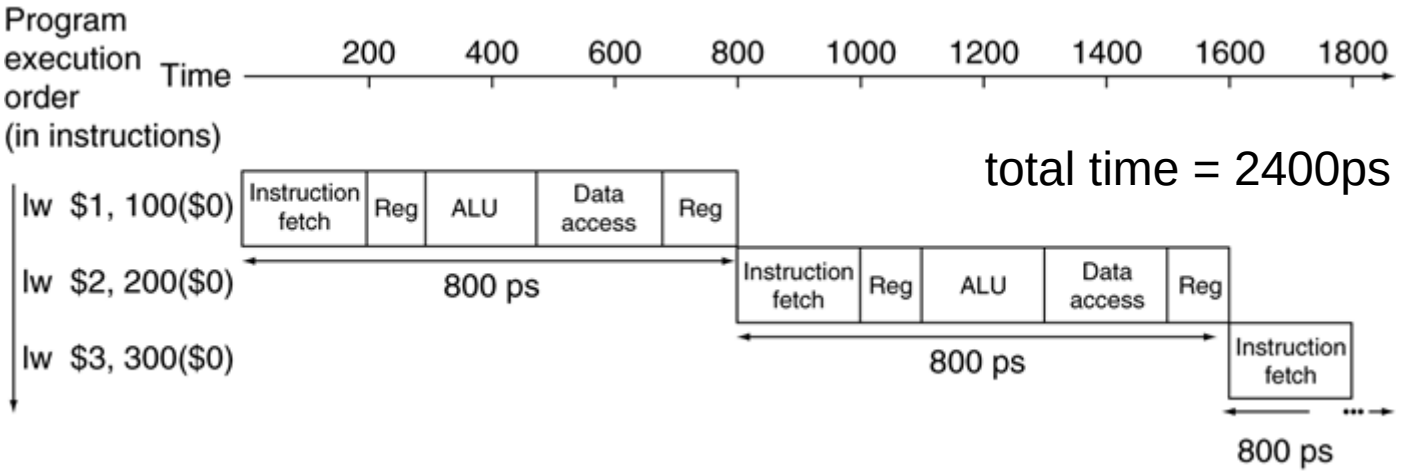
Adapted by Izzat El Hajj from textbook slides for Computer Organization and Design, by David Patterson and John Hennessy, Morgan Kaufmann.

Last Time: Pipelining the Datapath

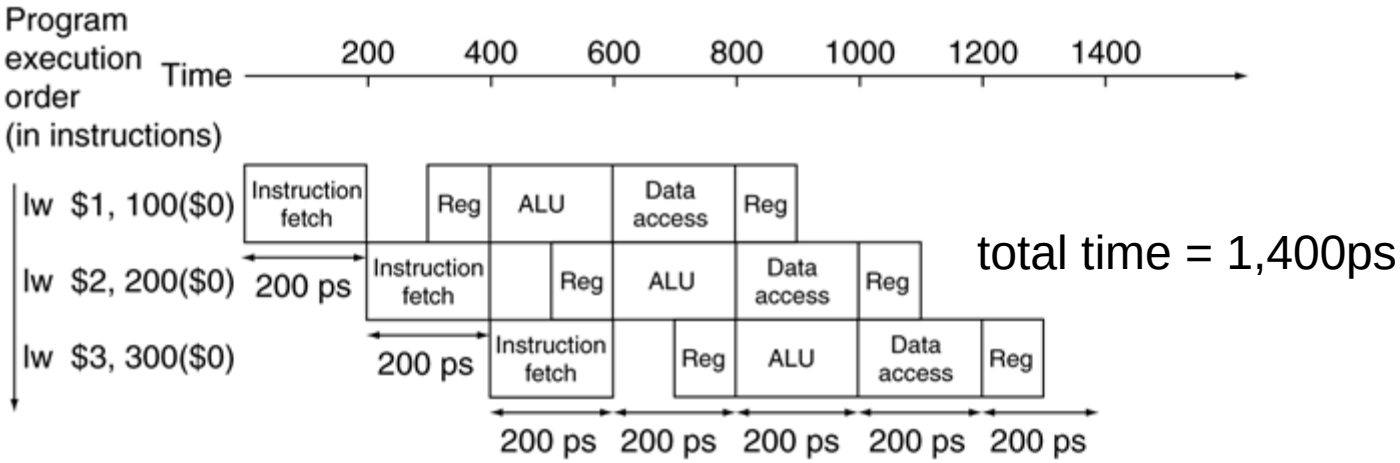


Performance with Pipelining

Without Pipelining

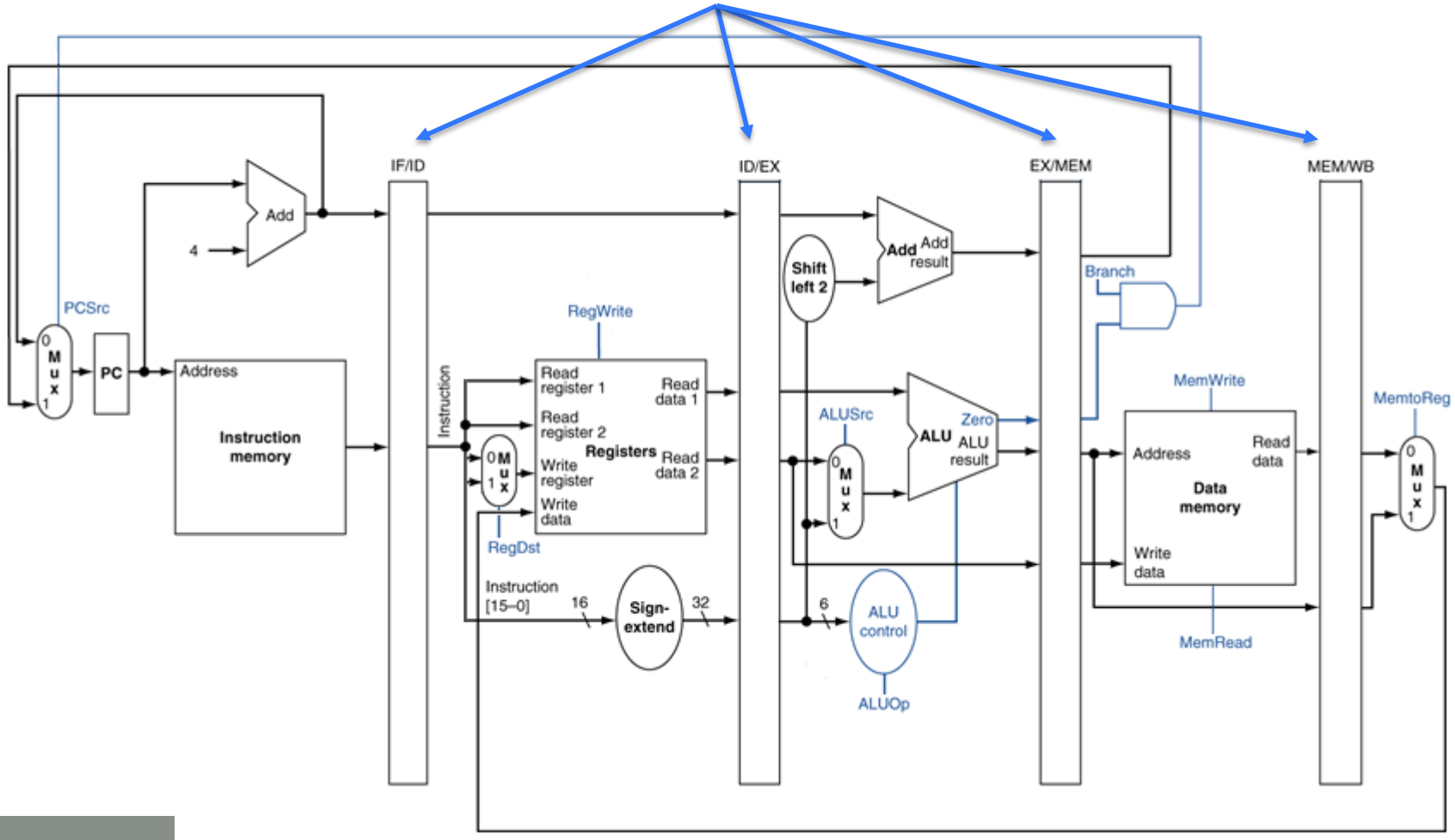


With Pipelining

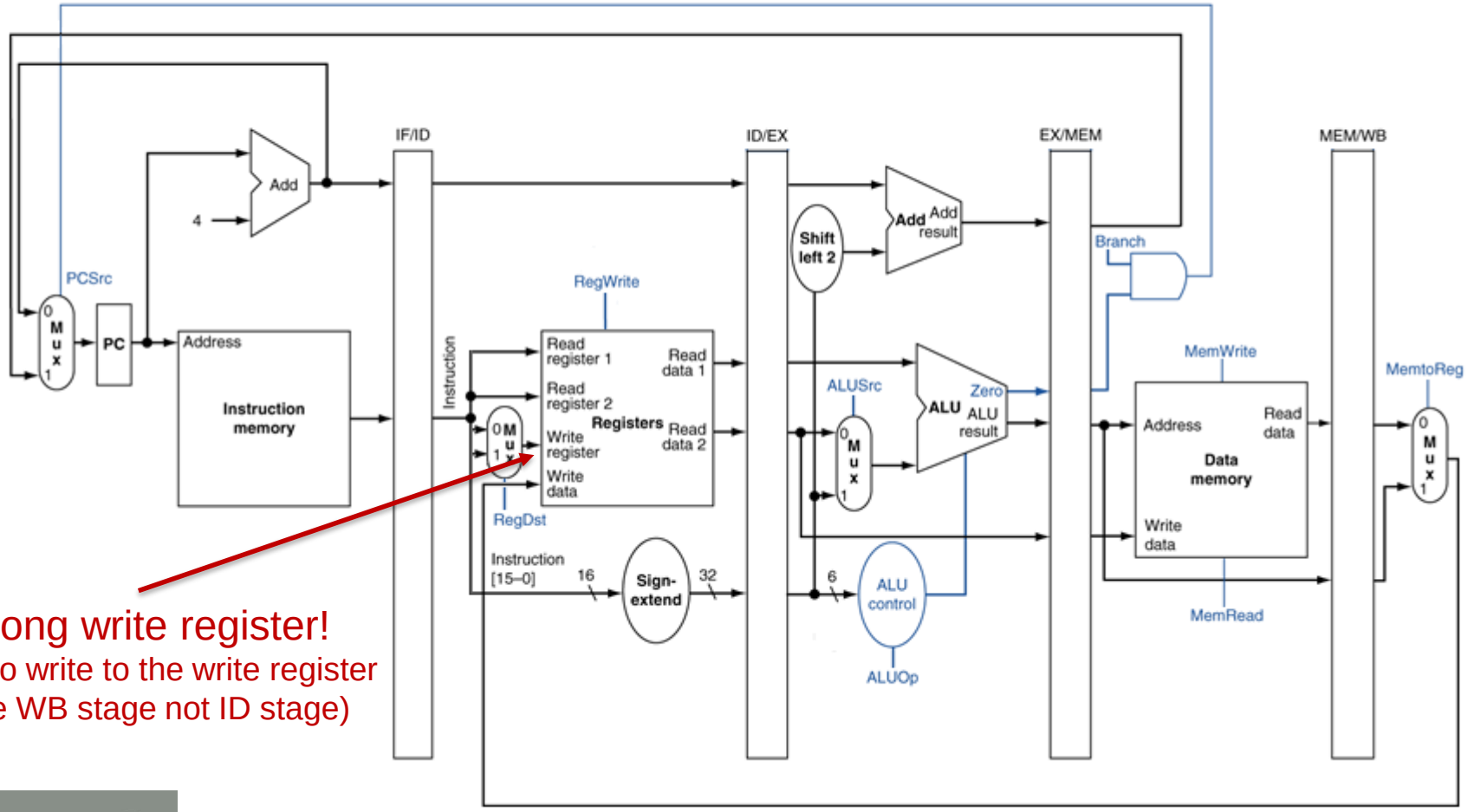


Pipelined Datapath

Pipeline registers ensure that the input to a stage is held for the entire cycle and updated on the next cycle



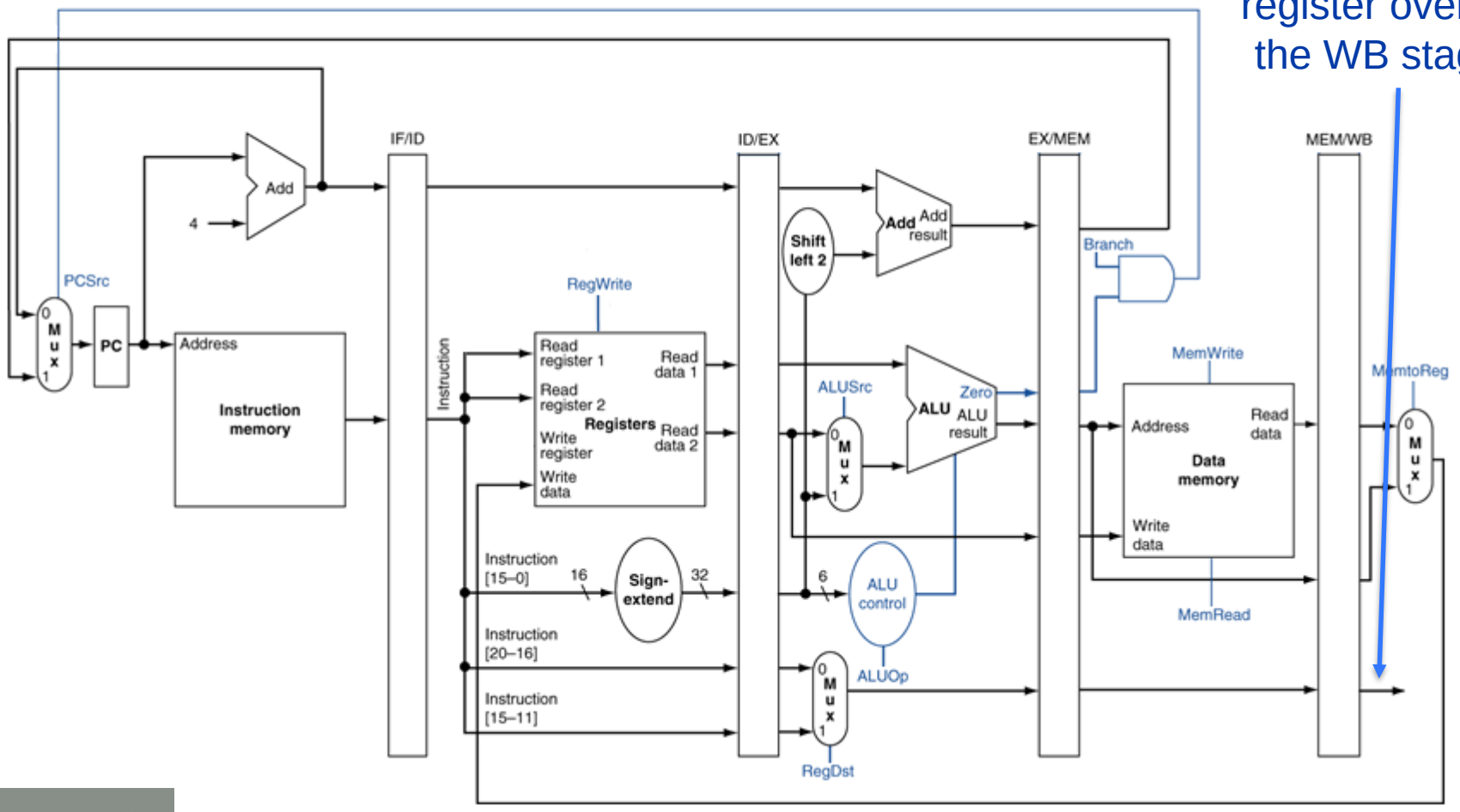
Fixing Writeback Destination



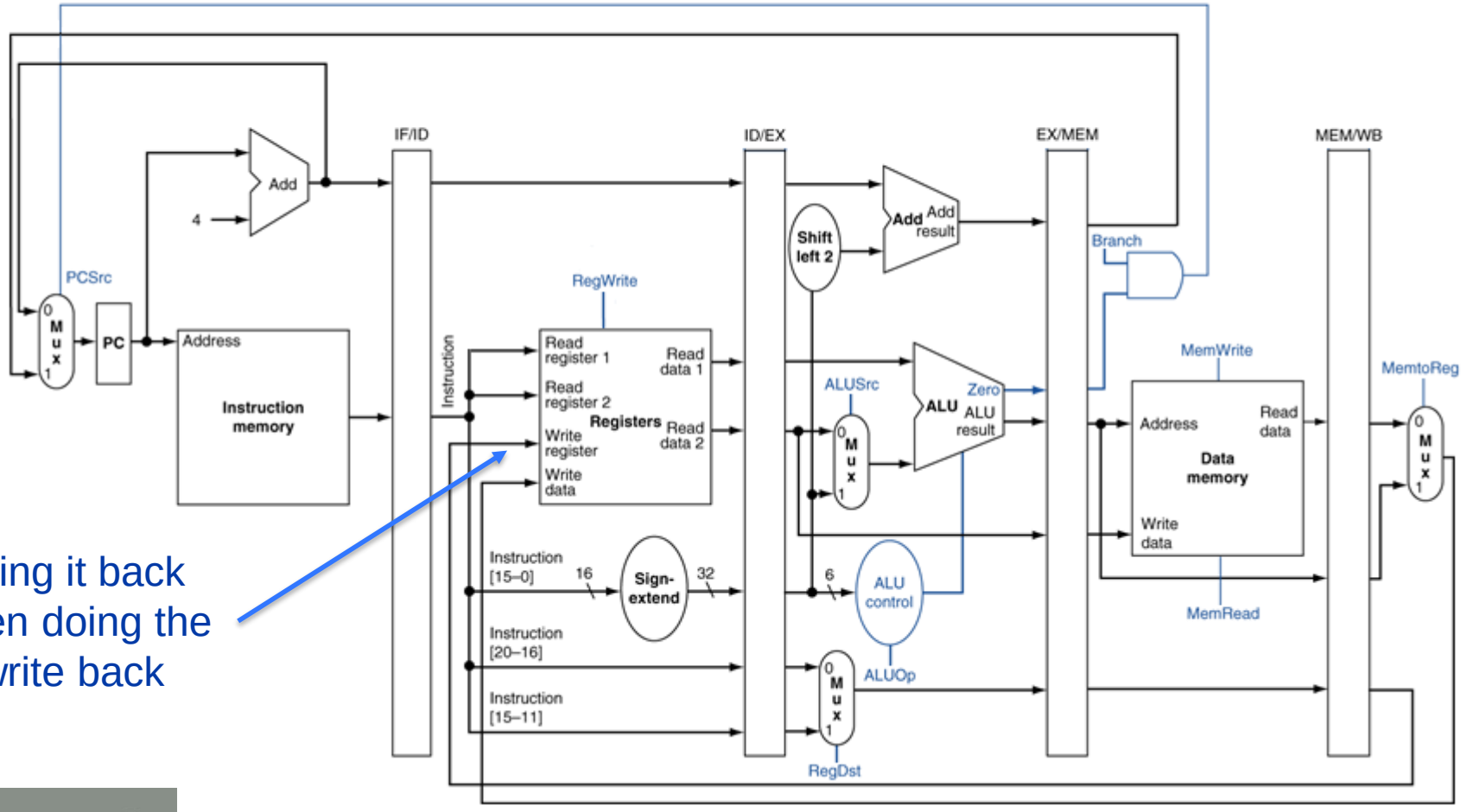
Wrong write register!
(need to write to the write register of the WB stage not ID stage)

Fixing Writeback Destination

Bring write register over to the WB stage

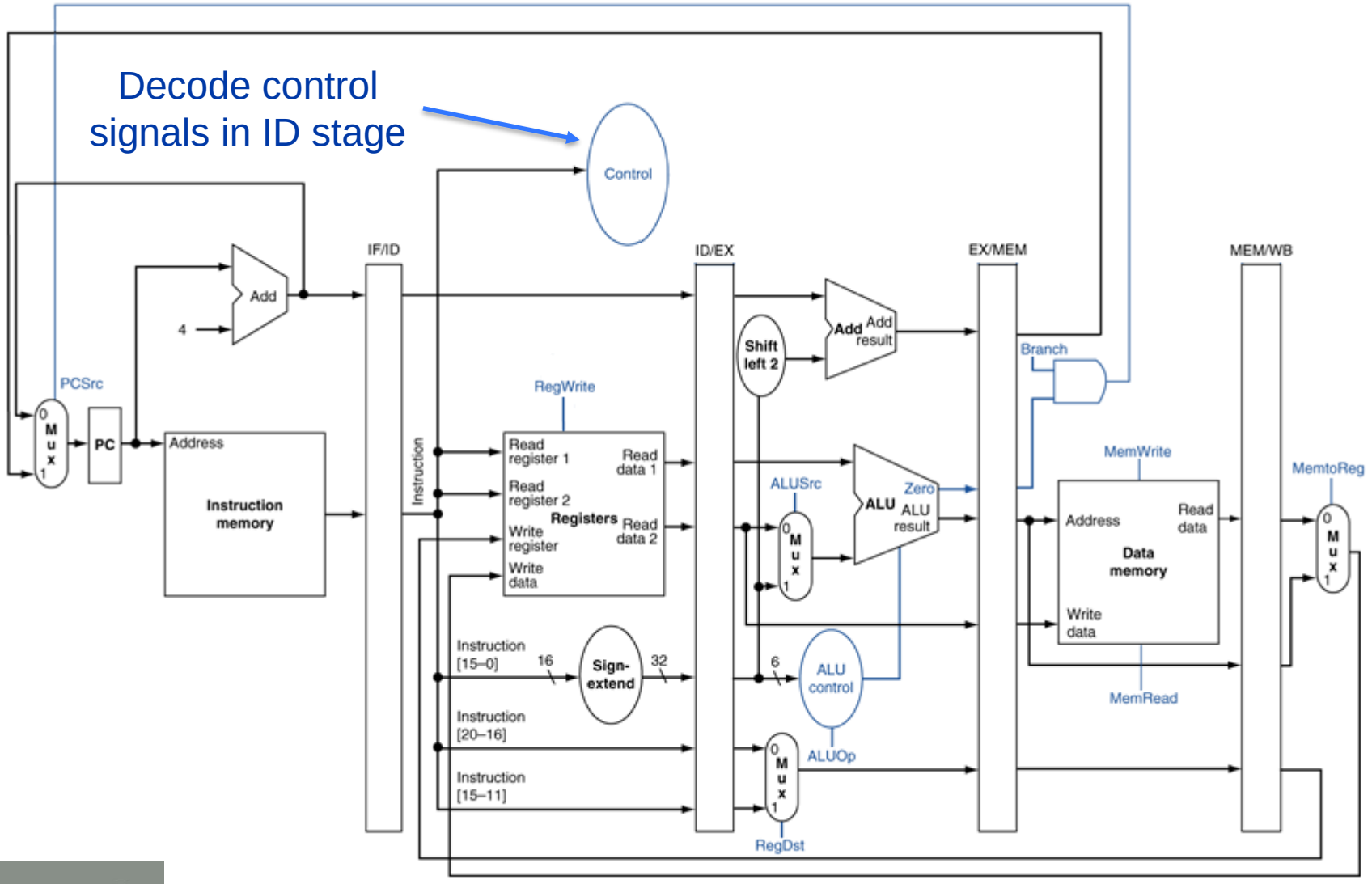


Fixing Writeback Destination

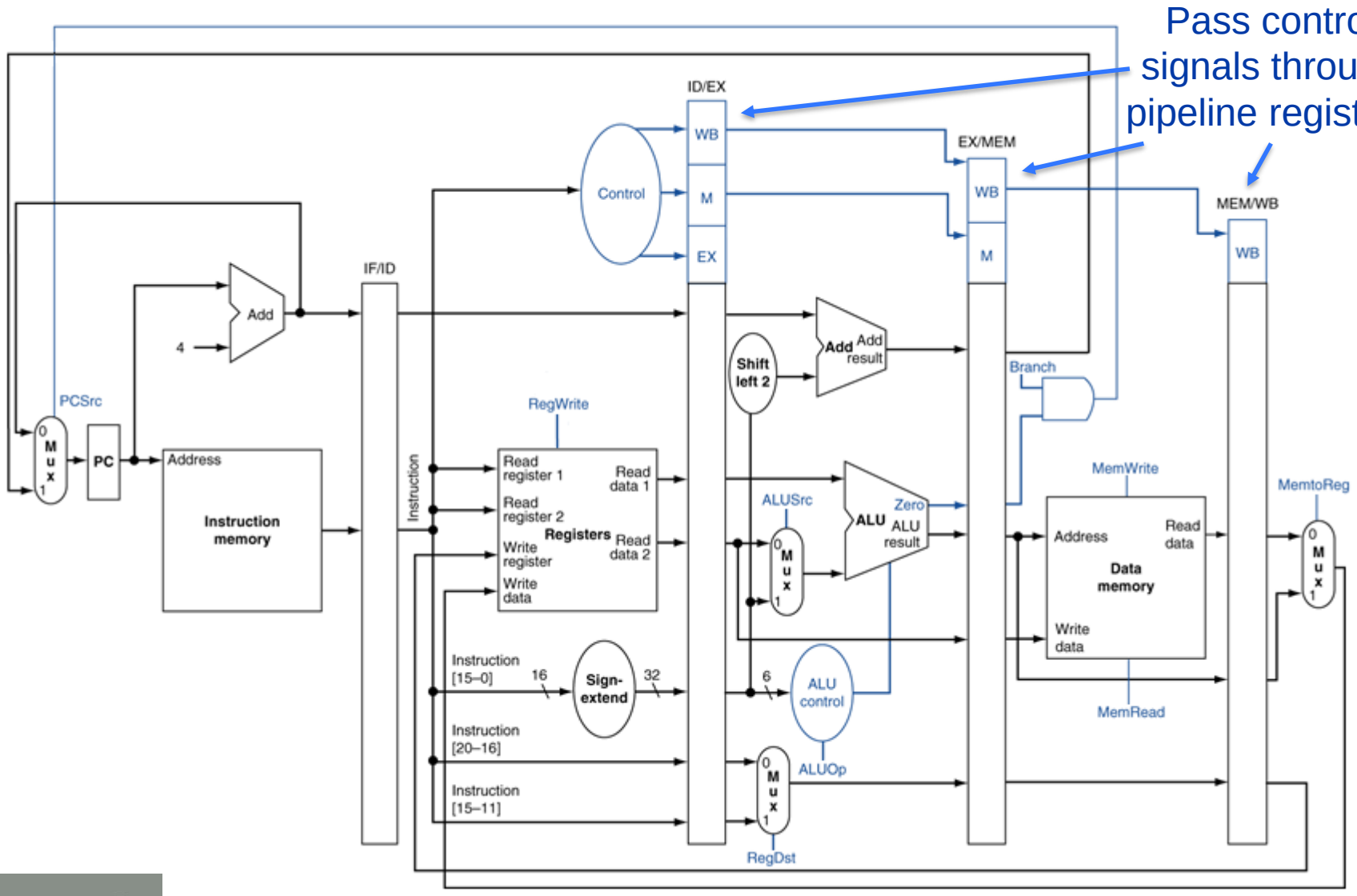


Bring it back when doing the write back

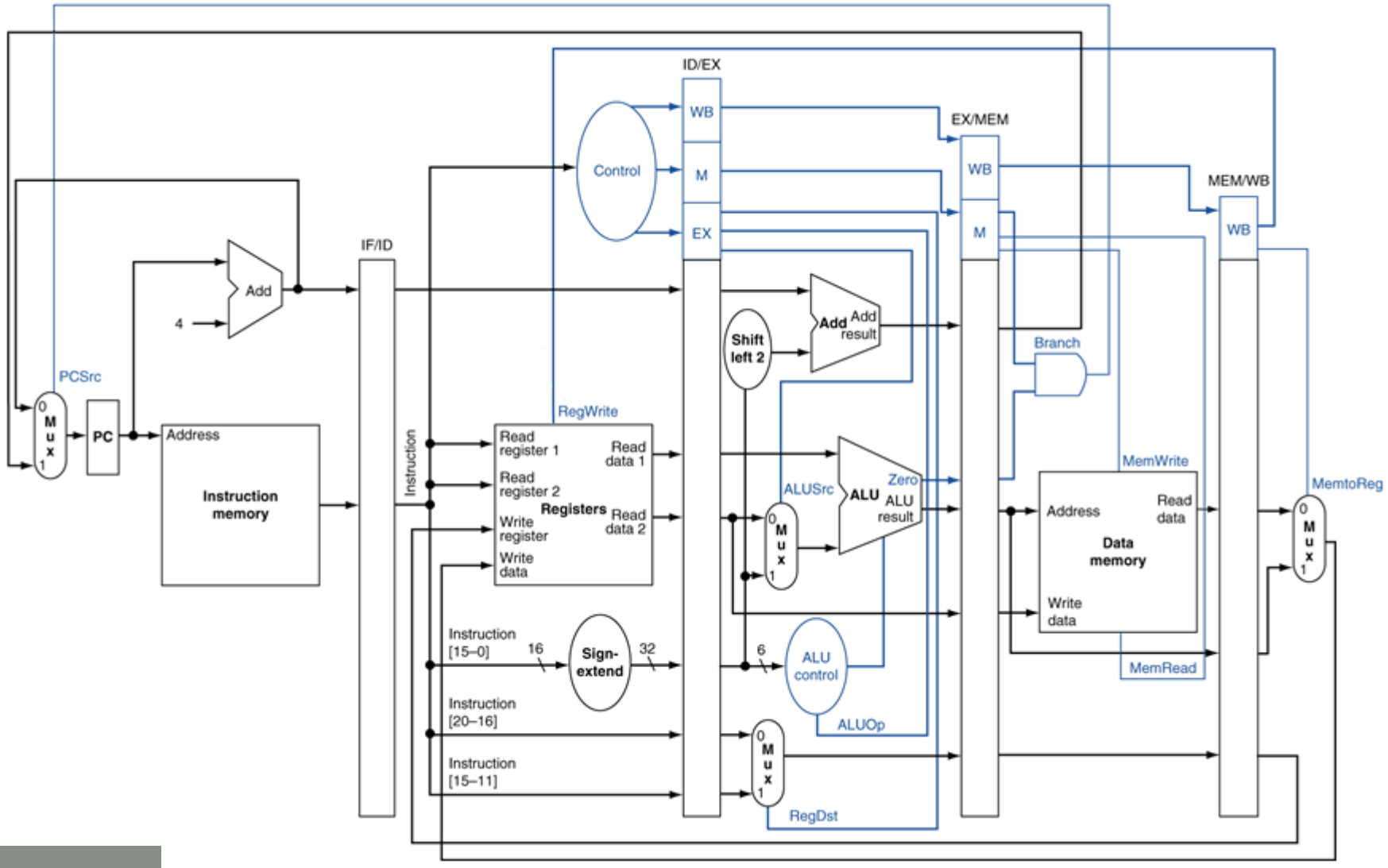
Pipeline with Control Signals



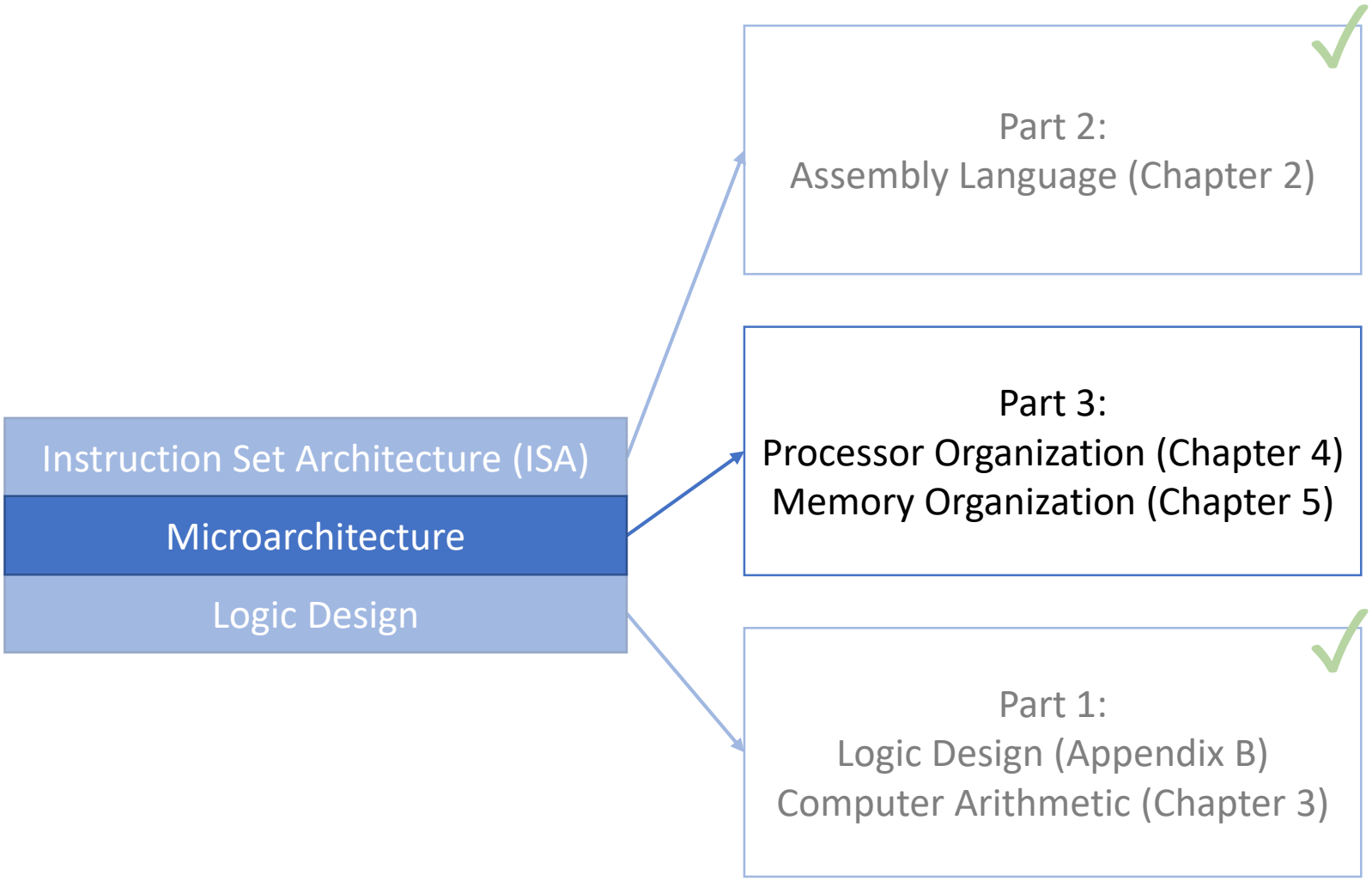
Pipeline with Control Signals



Pipeline with Control Signals



Today: Pipelining the Datapath



Hazards

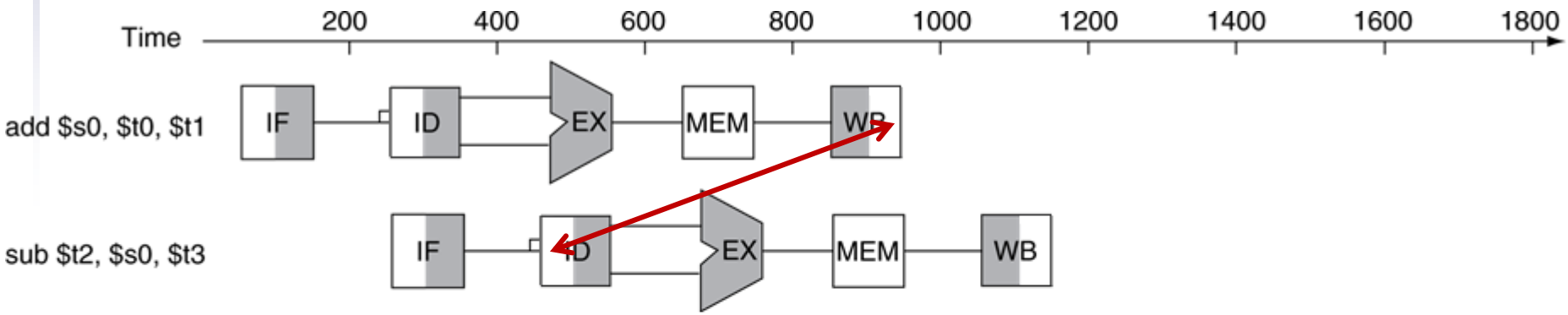
- A **hazard** is a situation that prevents an instruction from executing a pipeline stage in the next cycle. Types of hazards:
  - **Data hazards**
    - An instruction needs to wait for another executing instruction to complete its data read/write
  - **Structure hazards**
    - A hardware structure required by an instruction is being used by another executing instruction
  - **Control hazards**
    - Deciding the next instruction to fetch depends on the outcome of another executing instruction

Hazards

- A **hazard** is a situation that prevents an instruction from executing a pipeline stage in the next cycle. Types of hazards:
  - **Data hazards**
    - An instruction needs to wait for another executing instruction to complete its data read/write
  - **Structure hazards**
    - A hardware structure required by an instruction is being used by another executing instruction
  - **Control hazards**
    - Deciding the next instruction to fetch depends on the outcome of another executing instruction

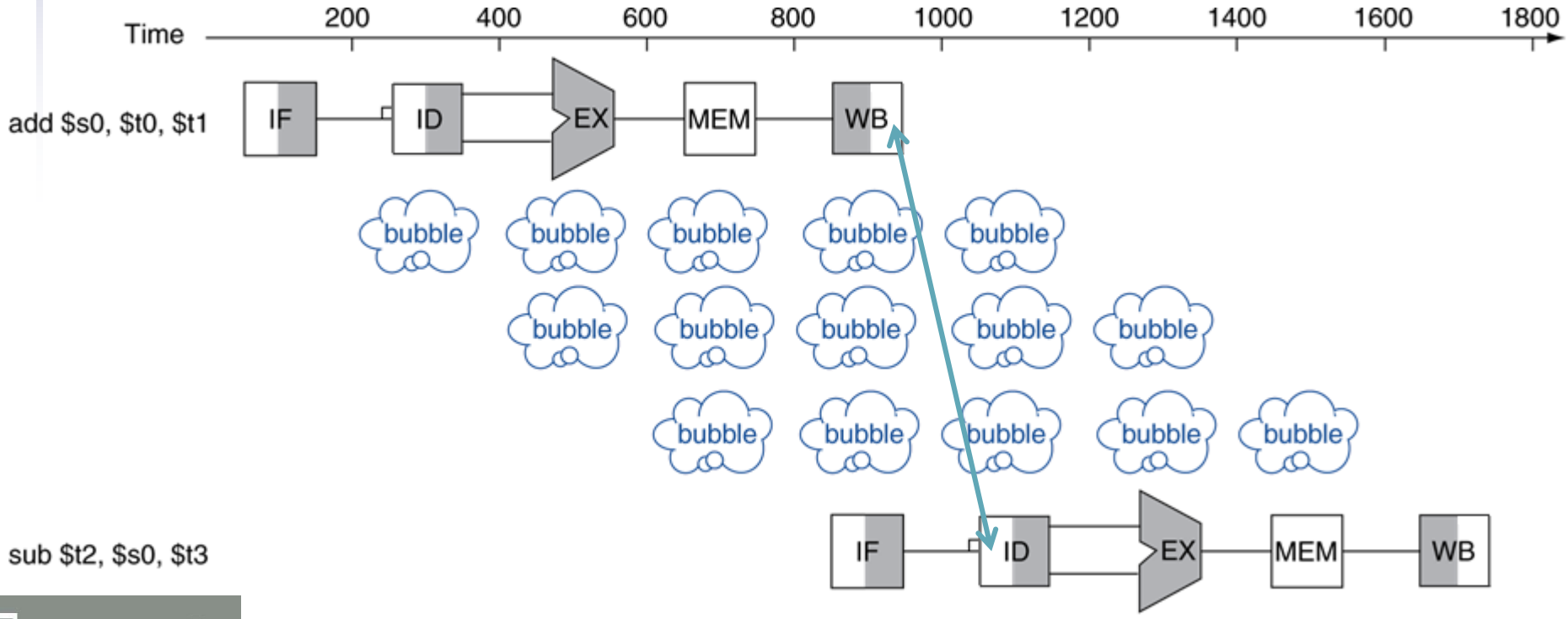
Data Hazards

- Recall: An instruction needs to wait for another executing instruction to complete its data read/write



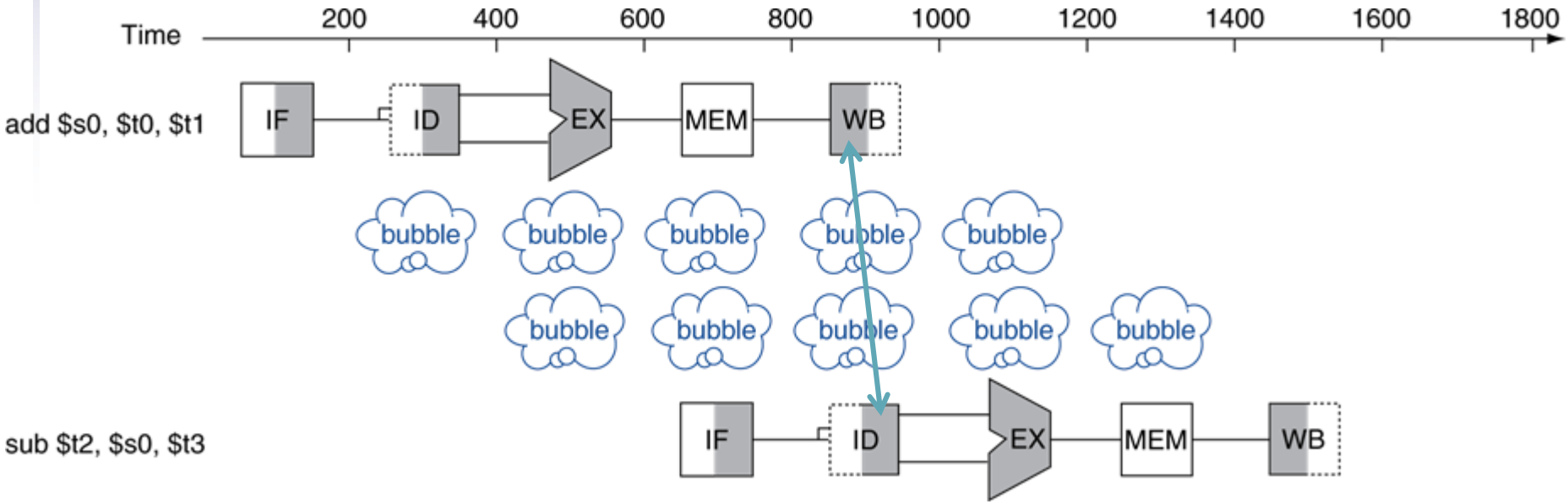
Stalling

- Solution: **stall** the instruction until the data hazard is gone
  - Insert “bubbles” in the pipeline
  - Disadvantage: many cycles wasted



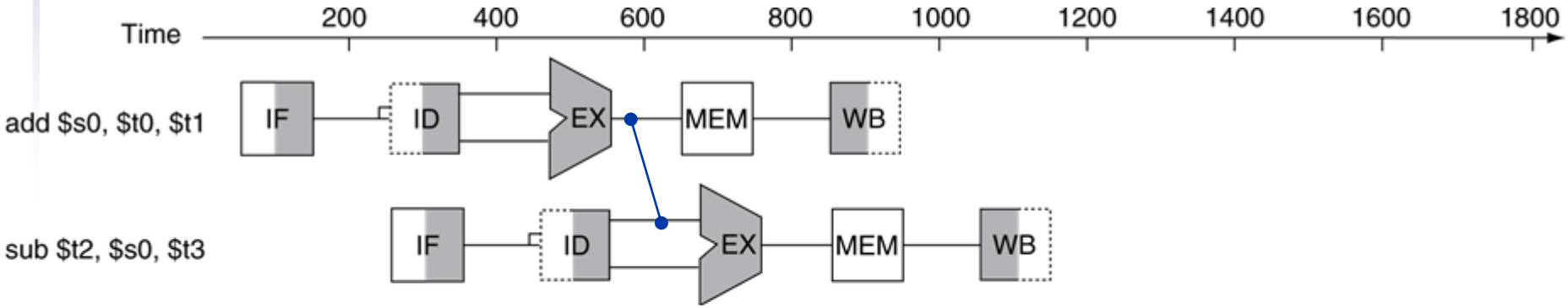
Optimizing WB/ID

- Another solution: perform WB in the first half of the cycle and ID in the second half
  - Allows values written during a cycle to be read in the same cycle



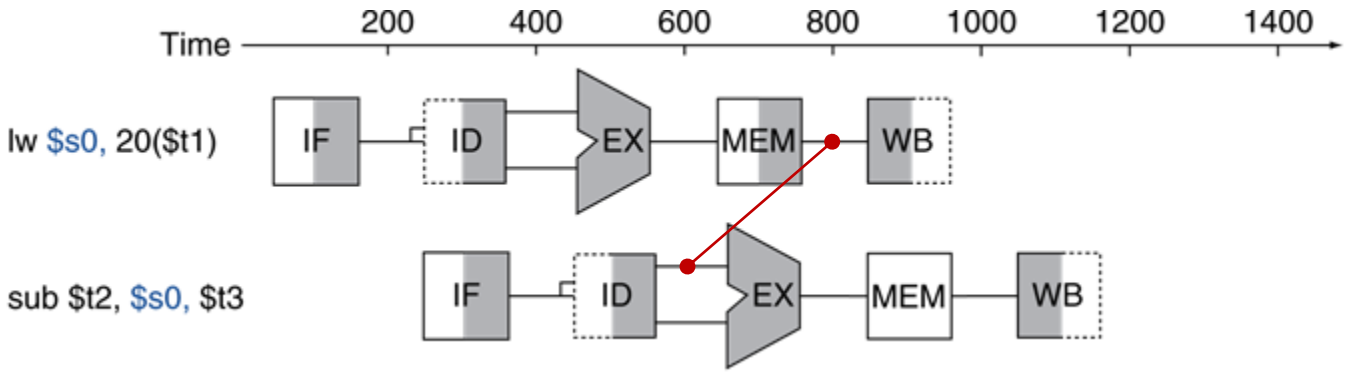
Forwarding (aka Bypassing)

- Yet another solution: **forwarding**
  - Result can be used as soon as it is computed without waiting for it to be stored in a register
  - Requires extra connections in the datapath



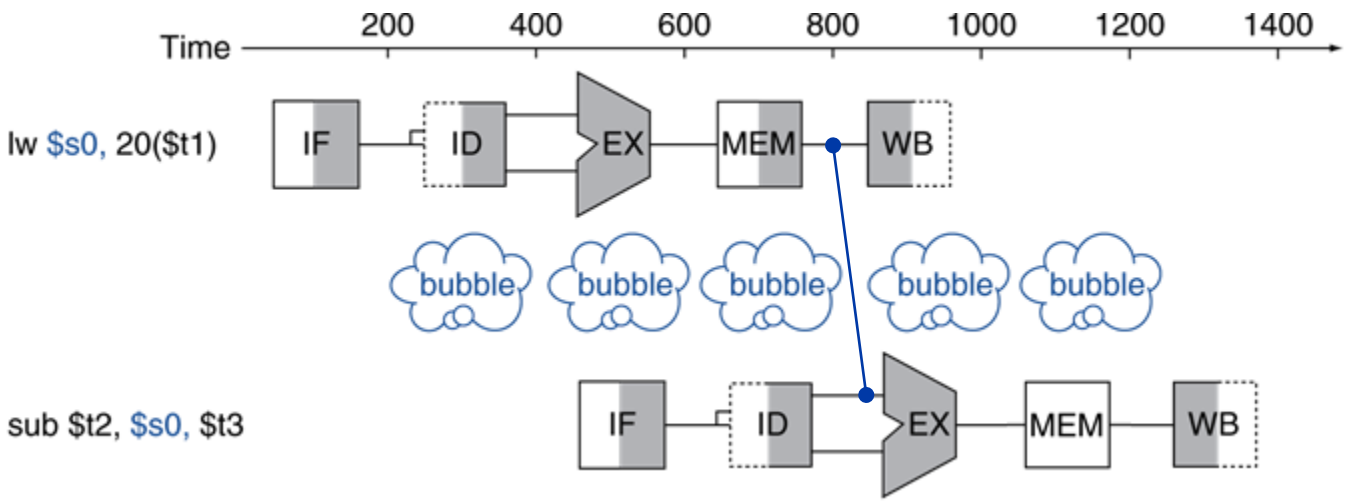
Load-Use Data Hazard

- Cannot always avoid stalls by forwarding
  - e.g., result of a load not available until after the MEM stage



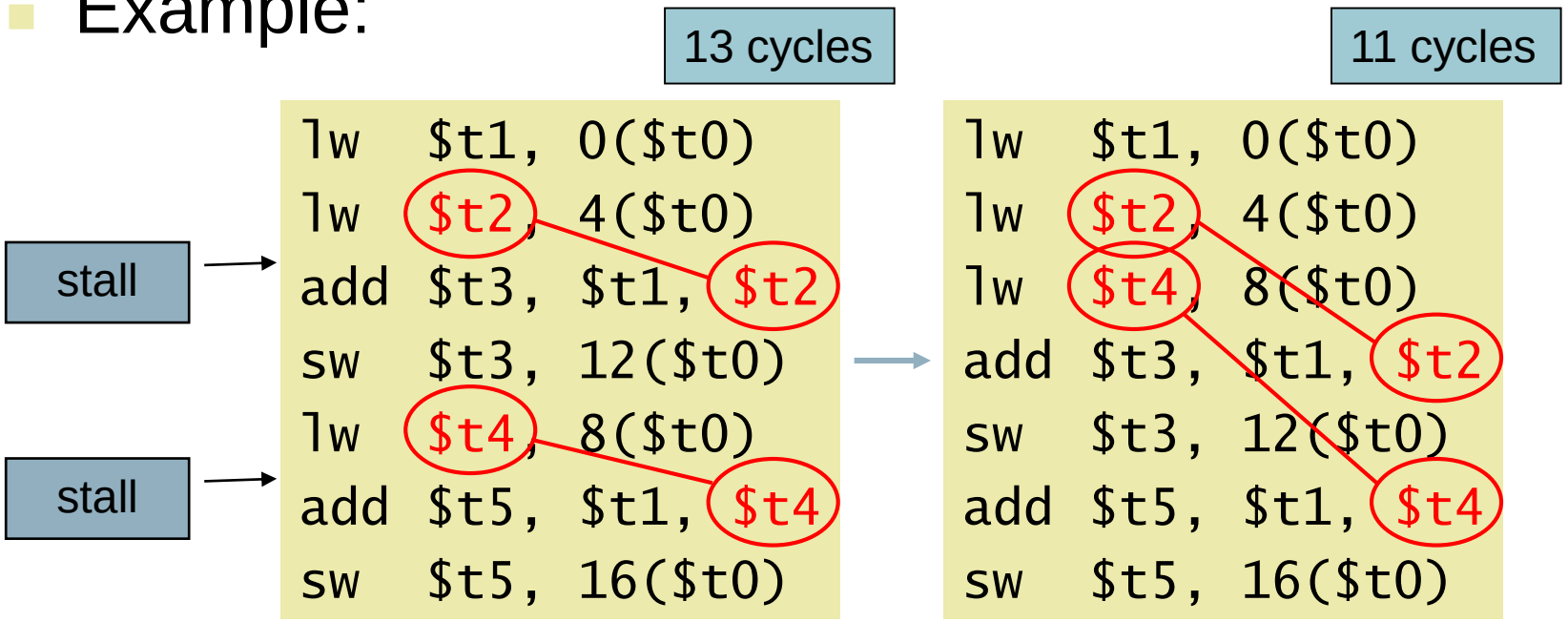
Load-Use Data Hazard

- Solution: stalling is inevitable in this case



Scheduling to Avoid Stalls

- **Instruction scheduling** by the compiler reorders code to avoid use of a load result in next instruction
- Example:



Hazards

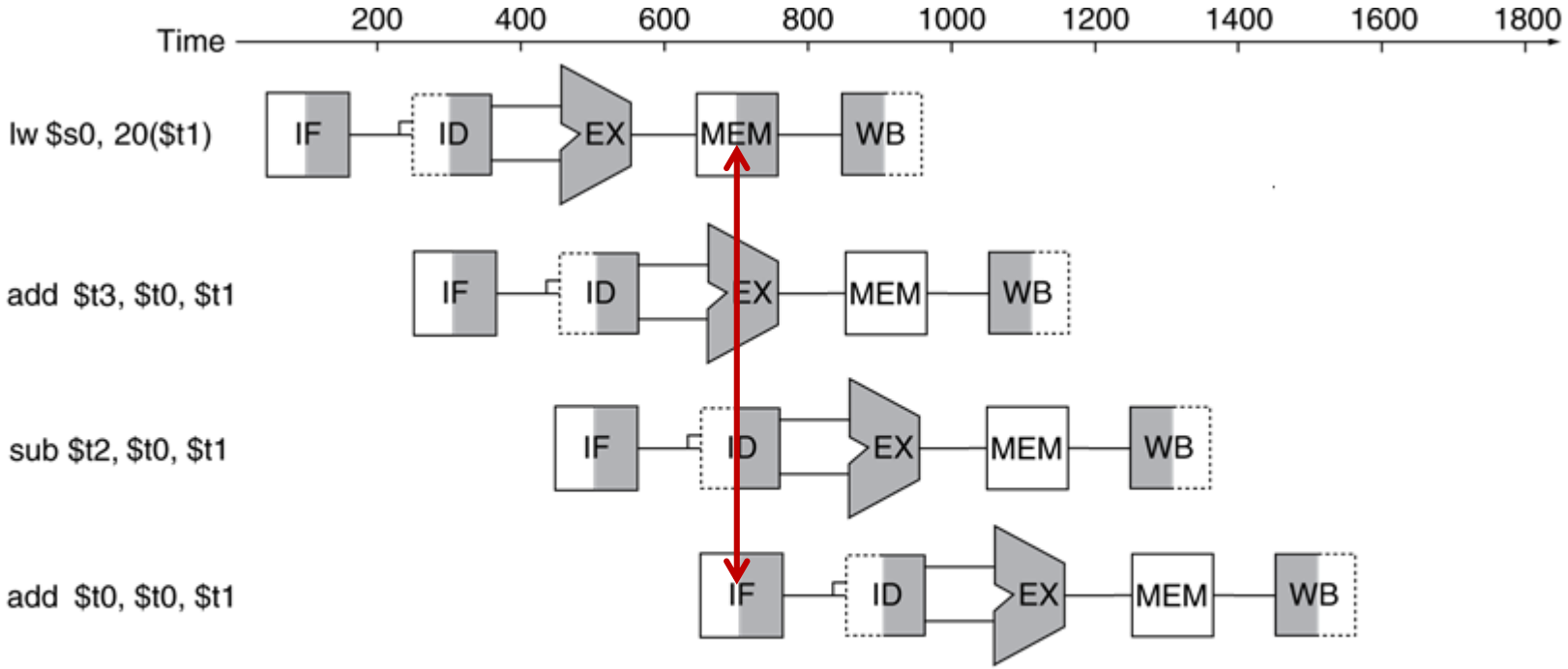
- A **hazard** is a situation that prevents an instruction from executing a pipeline stage in the next cycle. Types of hazards:
  - **Data hazards**
    - An instruction needs to wait for another executing instruction to complete its data read/write
  - **Structure hazards**
    - A hardware structure required by an instruction is being used by another executing instruction
  - **Control hazards**
    - Deciding the next instruction to fetch depends on the outcome of another executing instruction

Structure Hazards

- Recall: A hardware structure required by an instruction is being used by another executing instruction
- The MIPS pipeline does not have structure hazards because:
  - It requires separate instruction and data memories
    - More precisely separate instruction and data caches
    - Otherwise, there could be a structure hazard between the IF and MEM stages on the memory structure
  - It allows reading to and writing from the register file on the same cycle
    - Otherwise, there could be a structure hazard between the ID and WB stages on the register file structure

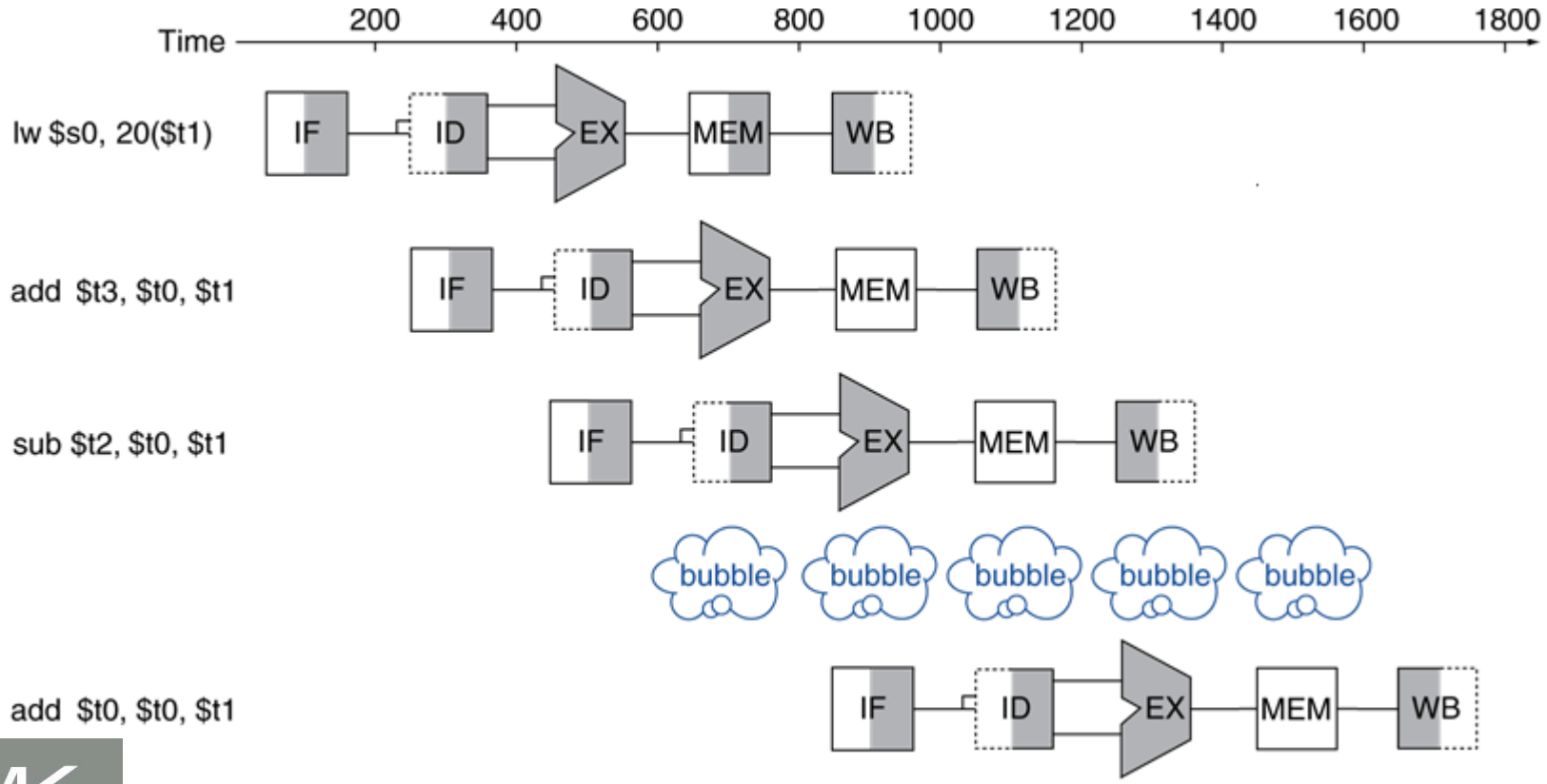
Structure Hazard Example

- Hypothetically, if the MIPS pipeline used a single memory for instructions and data, there could be a structure hazard between IF and MEM stages



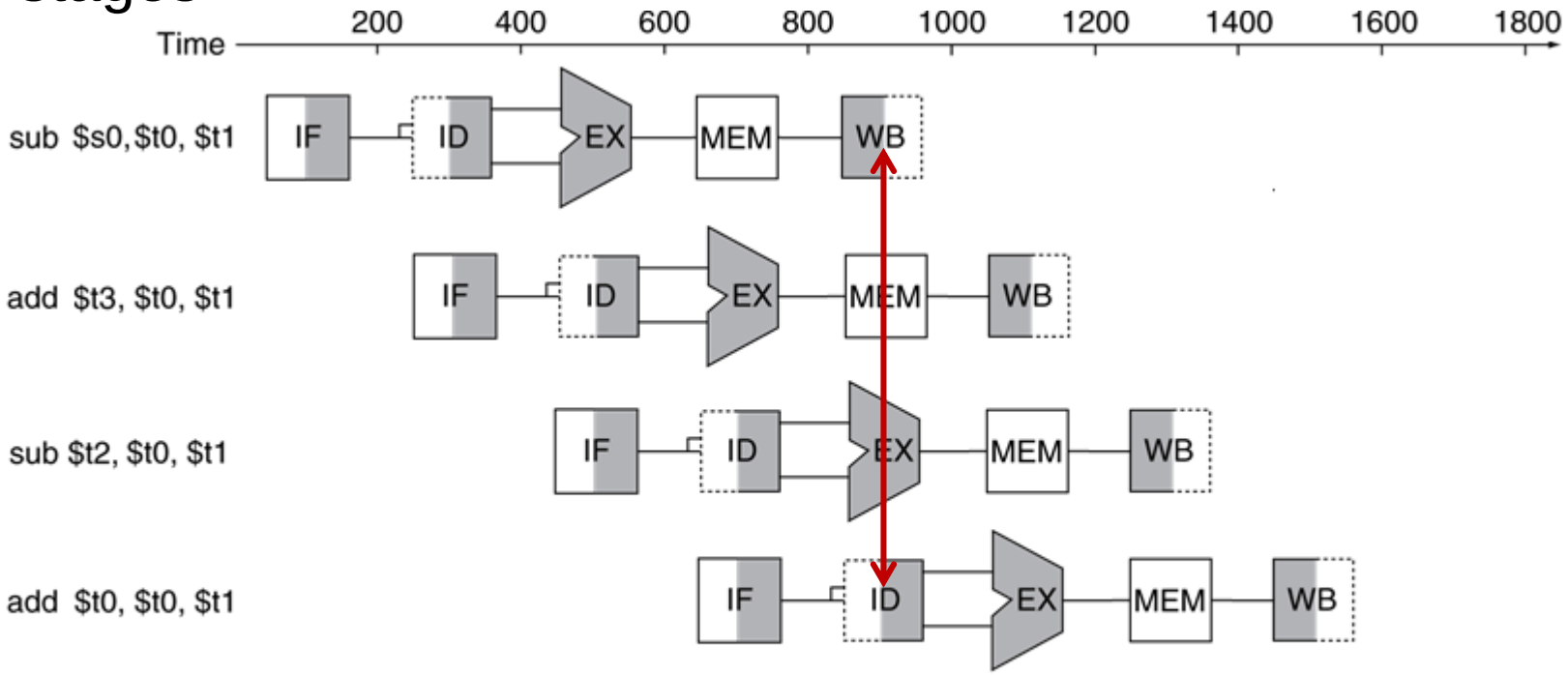
Structure Hazard Example

- Instruction fetch would have to *stall* when load/store instructions performs data access



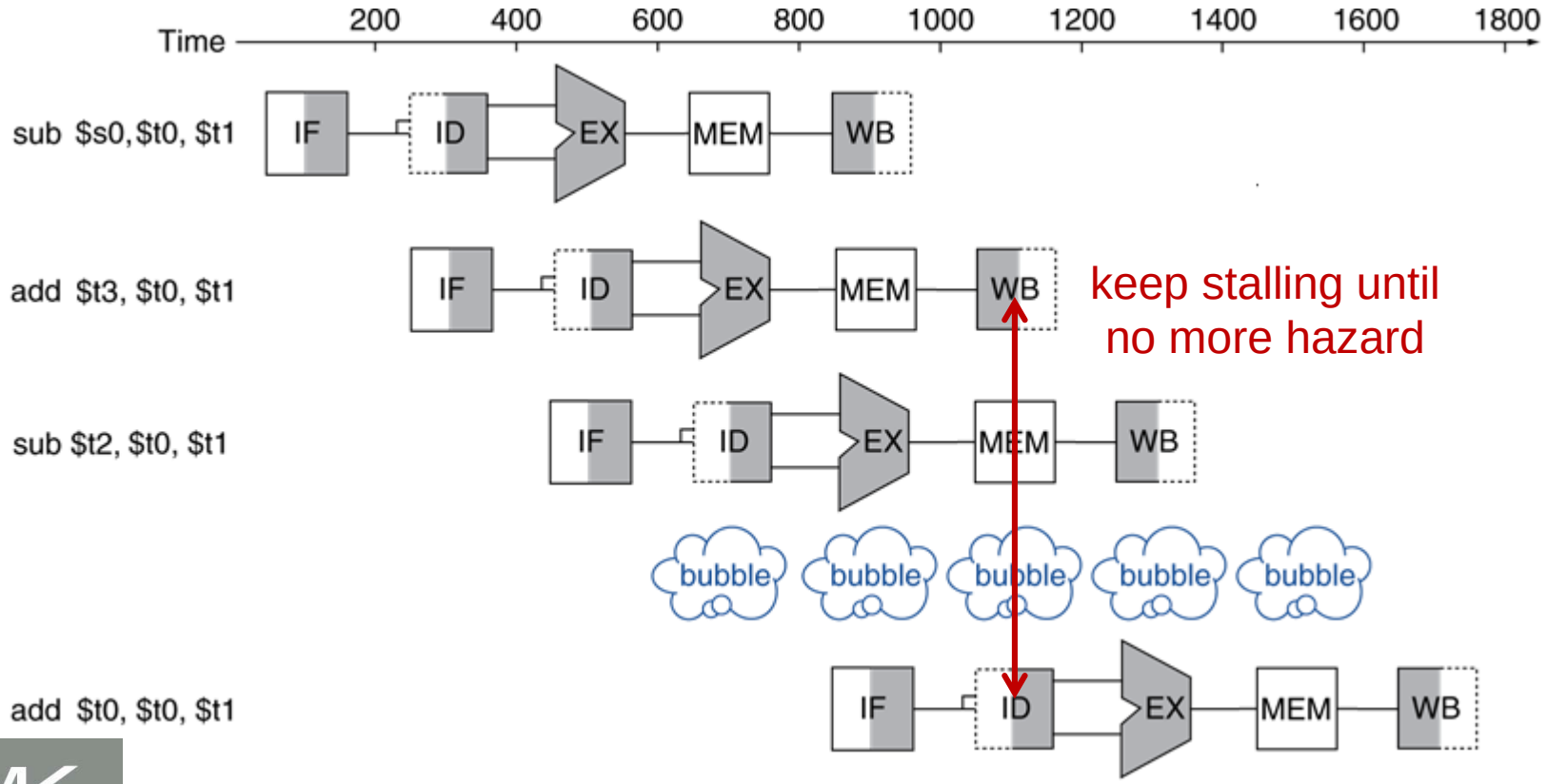
Structure Hazard Example

- Hypothetically, if the MIPS pipeline did not allow reading to and writing from the register file on the same cycle, there could be a structure hazard between ID and WB stages



Structure Hazard Example

- Instruction decode would have to *stall* when other instructions need to write back

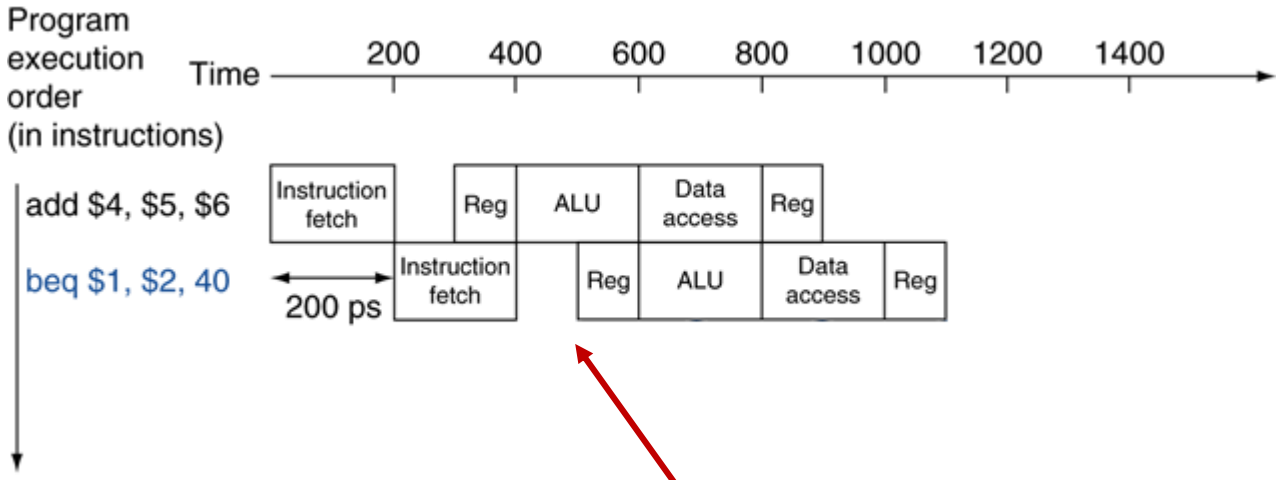


Hazards

- A **hazard** is a situation that prevents an instruction from executing a pipeline stage in the next cycle. Types of hazards:
  - **Data hazards**
    - An instruction needs to wait for another executing instruction to complete its data read/write
  - **Structure hazards**
    - A hardware structure required by an instruction is being used by another executing instruction
  - **Control hazards**
    - Deciding the next instruction to fetch depends on the outcome of another executing instruction

Control Hazards

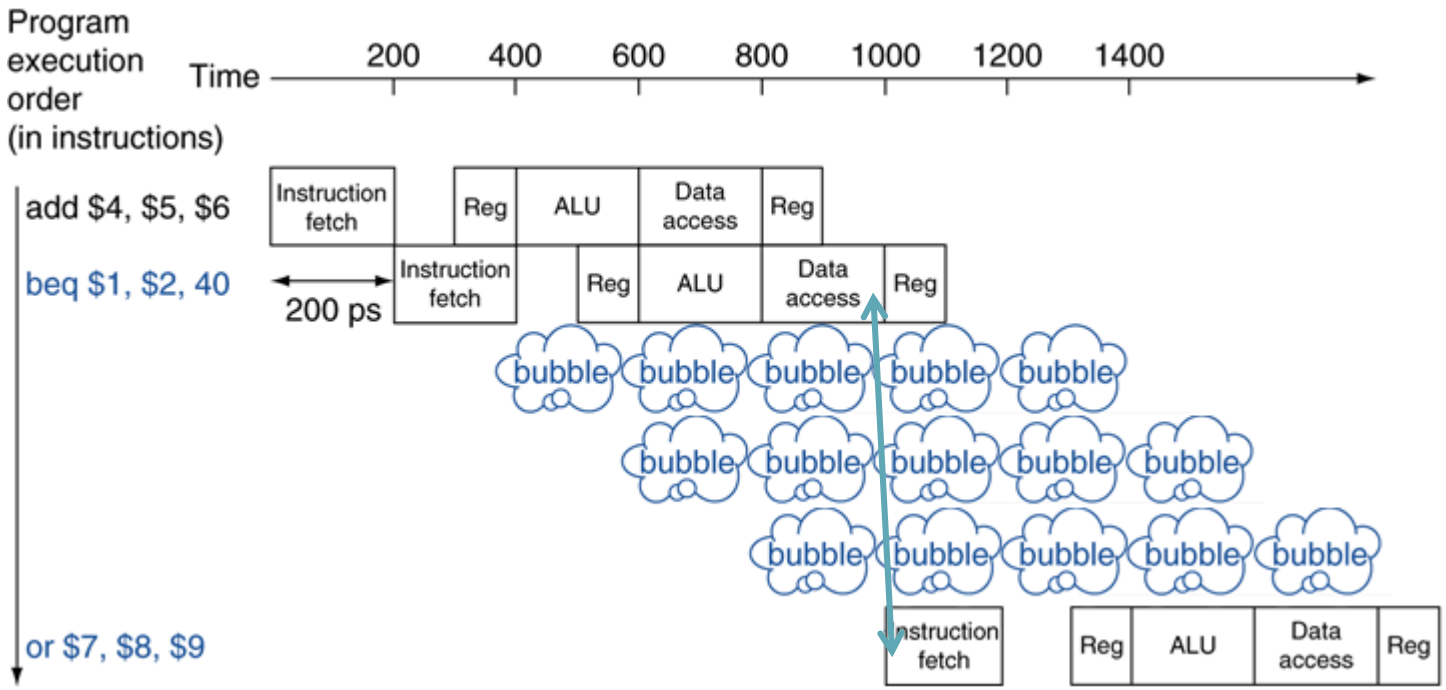
- Recall: Deciding the next instruction to fetch depends on the outcome of another executing instruction
  - e.g., branches



cannot fetch next instruction because branch outcome unknown

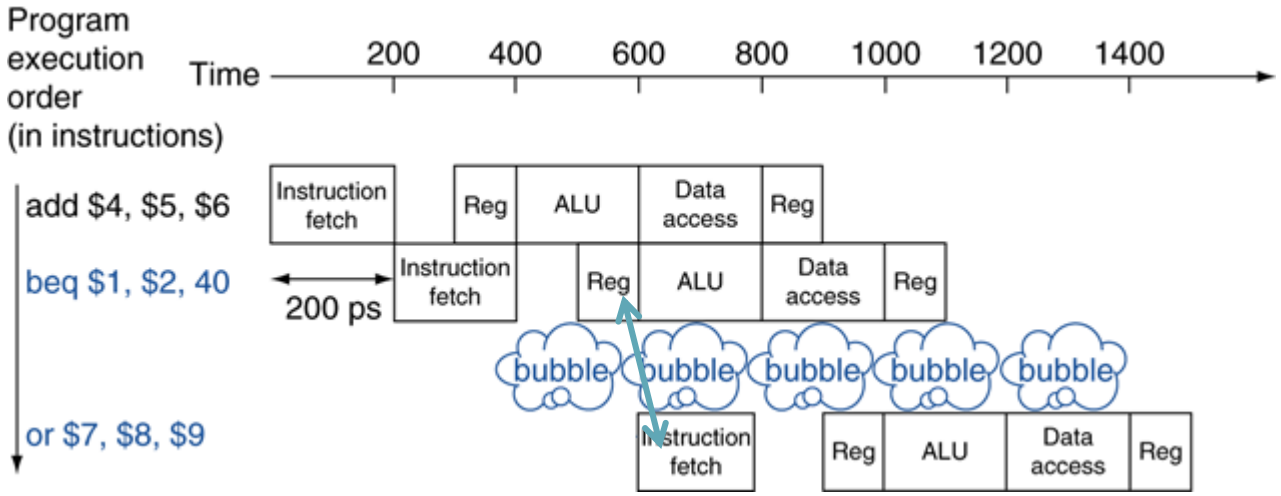
Stall on Branch

- Solution: stall until the branch outcome is determined before fetching the next instruction



Resolve Branches Earlier

- Another solution: compare registers and compute target earlier in the pipeline
  - Add hardware to do it in ID stage instead of EX stage

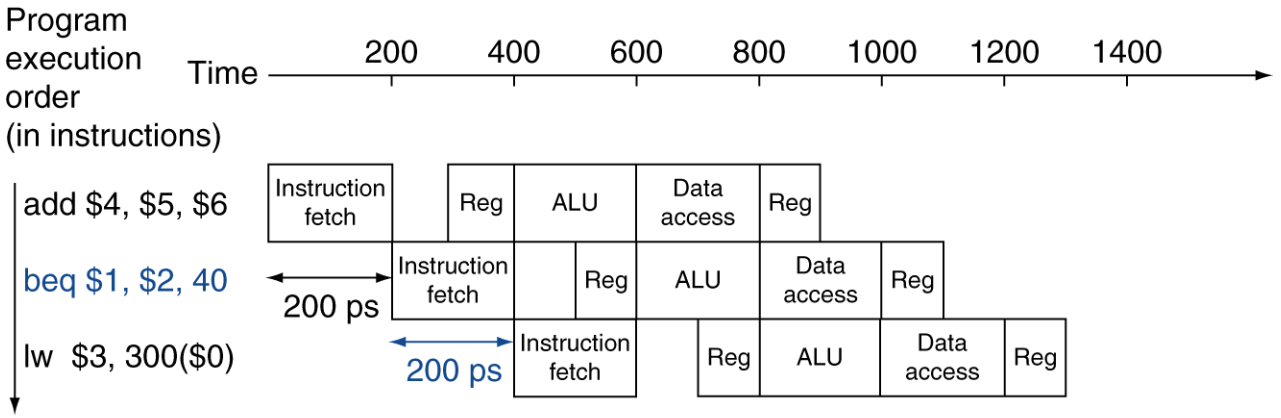


Branch Prediction

- Longer pipelines cannot readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Use **branch prediction** to predict the outcome of a branch
  - Only stall if the prediction is wrong
- In the MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

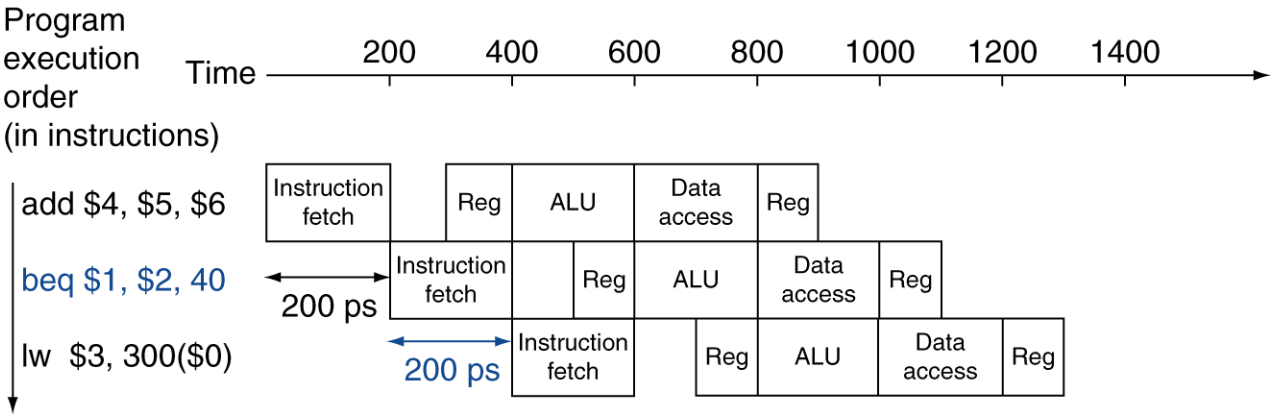
MIPS with Predict Not Taken

Prediction correct

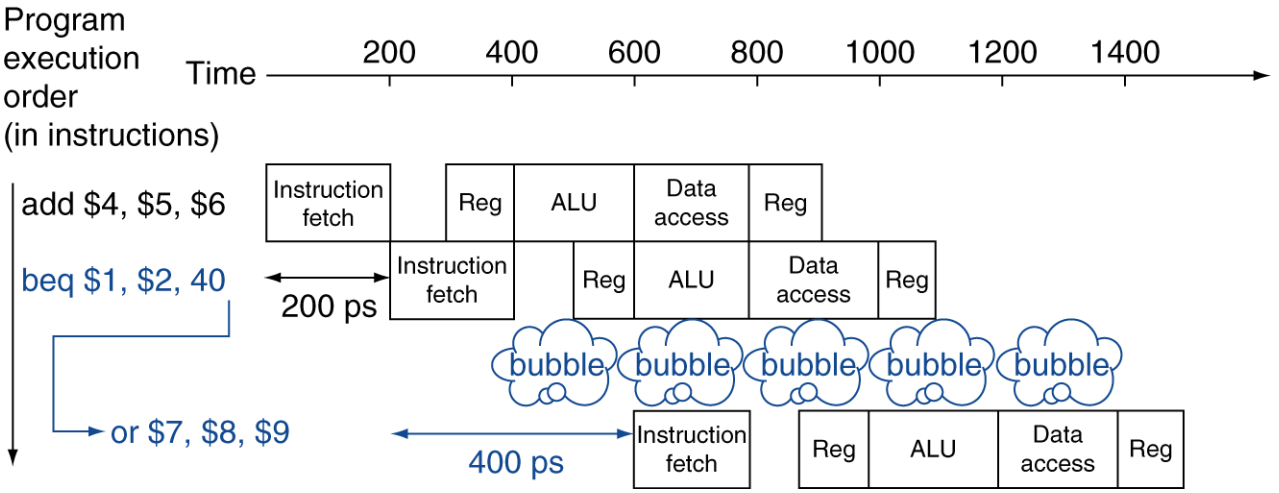


MIPS with Predict Not Taken

Prediction correct



Prediction incorrect



More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict forward branches not taken
    - Predict backward branches 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

Textbook Sections

- The content in these slides corresponds to:
  - Textbook:
    - *Computer Organization and Design, 5th Edition by David Patterson and John Hennessy, Morgan Kaufmann, 2014.*
  - Sections:
    - 4.5