Design of Embedded Hardware and Firmware

Pipeline

Andrea Guerrieri

HES-SO//Genève

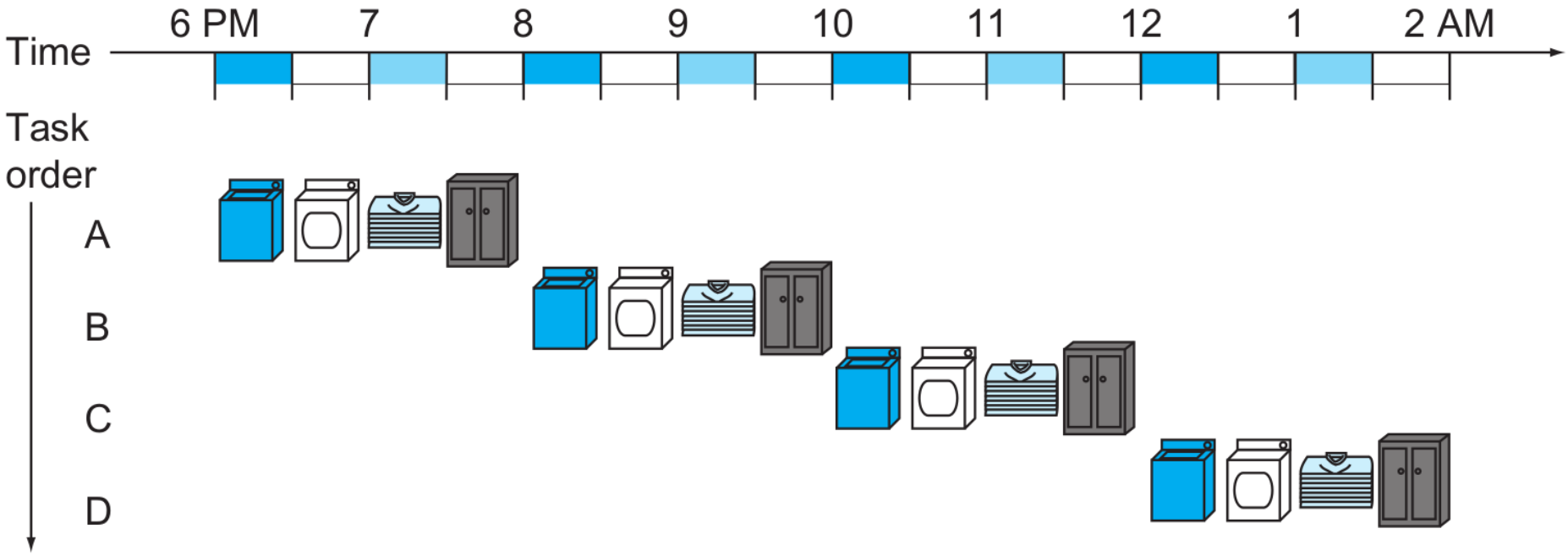
andrea.guerrieri@hesge.ch

Motivation

- We want to maximize performance:
  - $\frac{Time}{program} = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Time}{Cycle}$
- $\frac{Instructions}{Program}$: Minimizing number of instructions
- $\frac{Cycles}{Instruction}$: CPI
- $\frac{Time}{Cycle}$: t_{CLK}: Frequency of operation

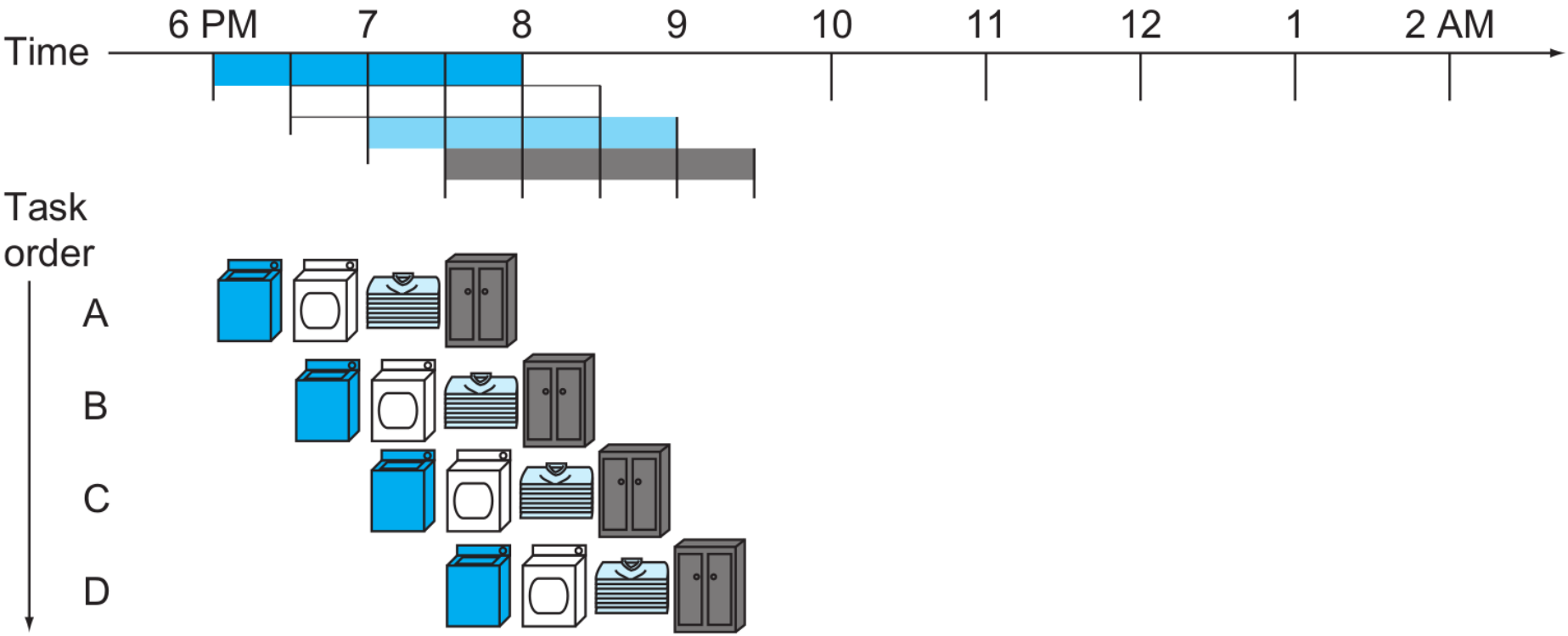
Motivation

- Single cycle



Motivation

- Pipelined



Pipelining

- Exploit parallelism
- Latency is the same, throughput is higher
- Better with high loads
- Potentially N times faster (N=stages in the pipeline)
- Frequency of system = frequency of slowest stage
- Unbalanced stages reduce the speed
- Filling/emptying times reduce the speed
- We need to wait for dependencies

Pipelining in MIPS. Example

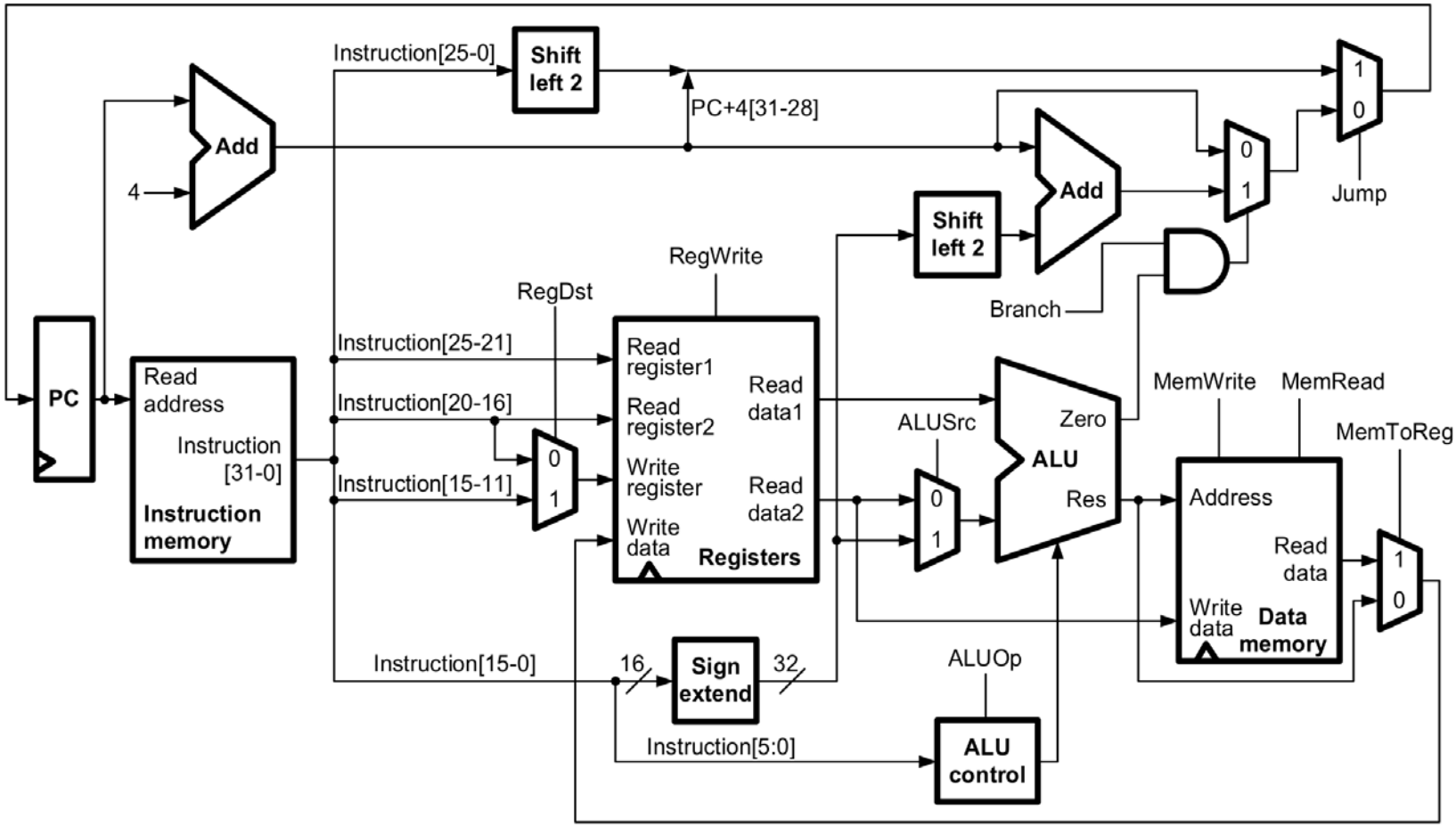
- Consider the MIPS architecture
  - Latency characteristics:
    - Memory access: 200 ps
    - ALU operation: 200 ps
    - Register file read/write: 100 ps

- Total time for each instruction:

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

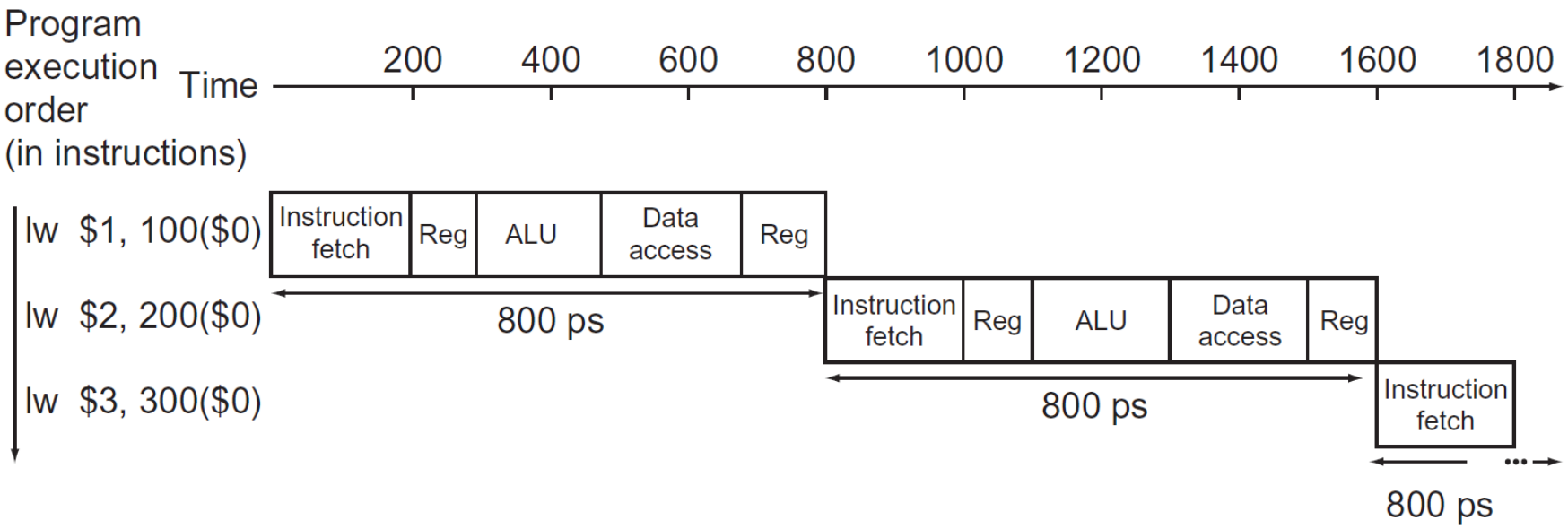
Pipelining in MIPS. Example

- Single cycle



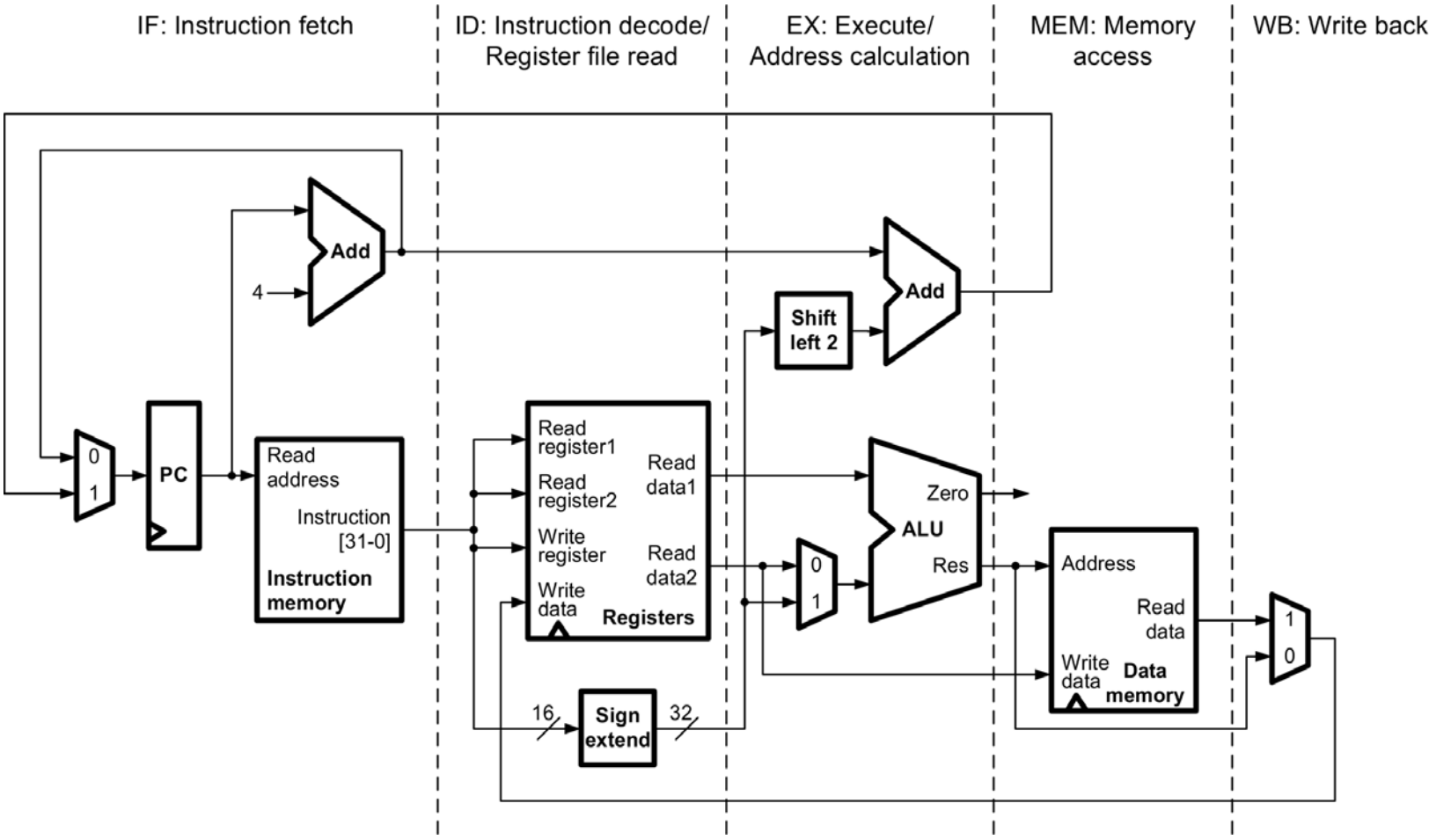
Pipelining in MIPS. Example

- Single cycle



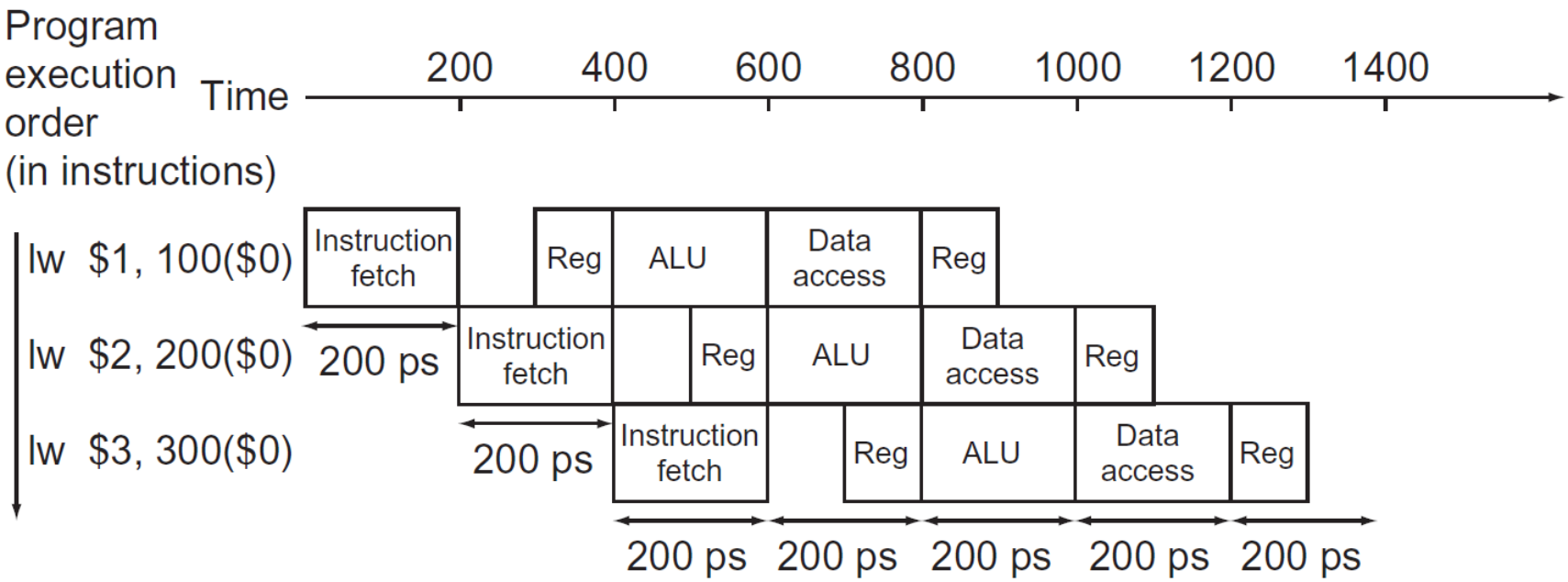
Pipelining in MIPS. Example

- Pipelined



Pipelining in MIPS. Example

- Pipelined



Pipelining in MIPS. Example

- Single cycle accommodates to slowest operation
  - Load: 800 ps
- Ideal pipelined version is about N times faster
  - With 5 stages, it would be: Load: ~160 ps
  - However, due to unbalanced stages: Load: 200 ps
- In the example:
  - Single cycle: 3 Loads: 2400 ps → 800 ps/instr.
  - Pipelined: 3 Loads: 1400 ps → ~467 ps/instr.
- However, for 10000 load operations:
  - Single cycle: 10000 Loads: 8 μs → 800 ps/instr.
  - Pipelined: 10000 Loads: 2.0008 μs → ~200 ps/instr.

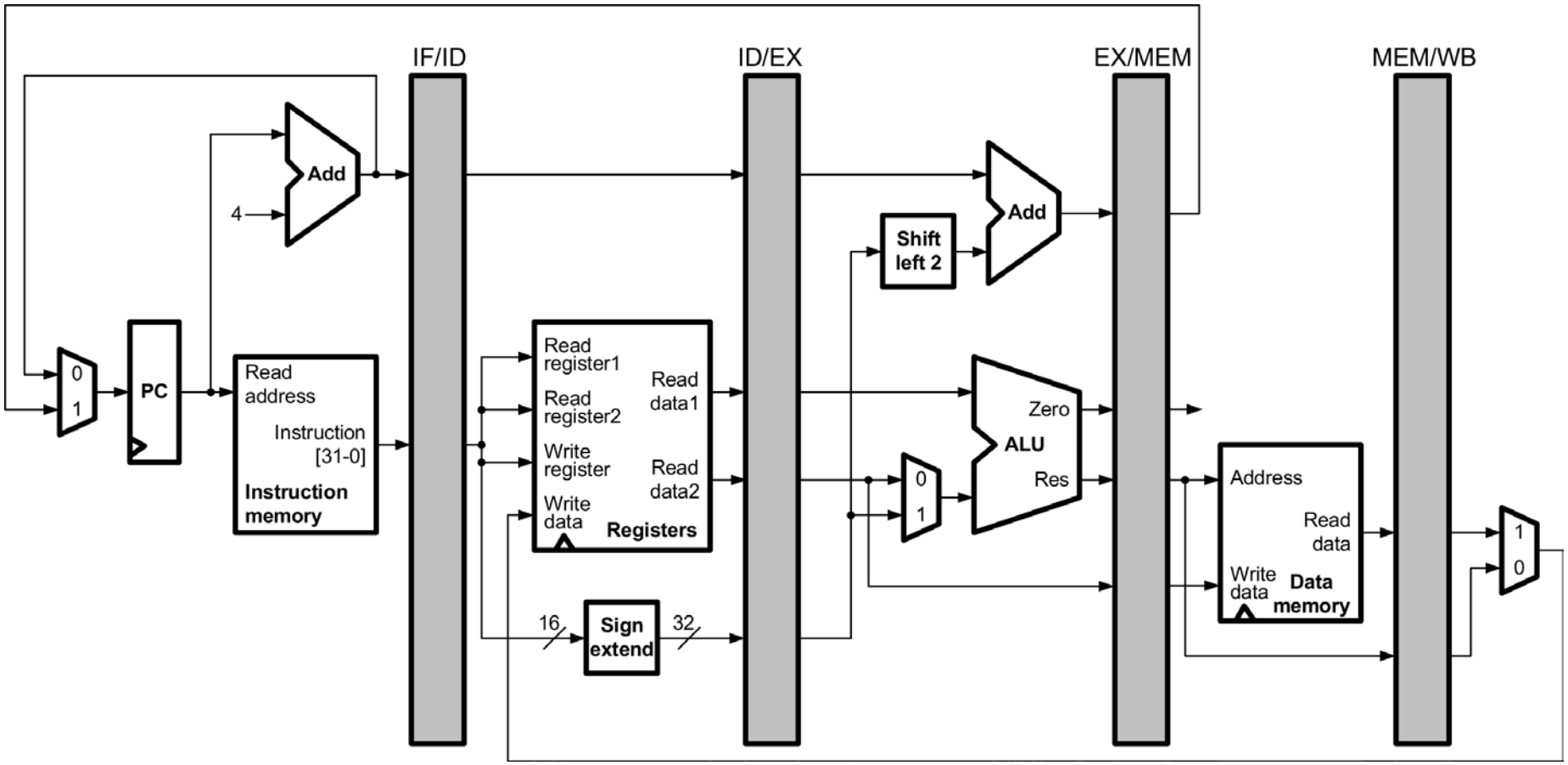
Pipelining in MIPS. Example

- We want to maximize performance:
  - $$\frac{Time}{program} = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Time}{Cycle}$$

- $\frac{Instructions}{Program}$: Minimizing number of instructions
- $\frac{Cycles}{Instruction}$: CPI
  - Single cycle: 1 CPI
  - Pipelined: ~1 CPI (+ 4 filling cycles)
- $\frac{Time}{Cycle}$: t_{CLK}: Frequency of operation
  - Single cycle: Slowest path: 800 ps
  - Pipelined: Slowest element in pipeline: 200 ps

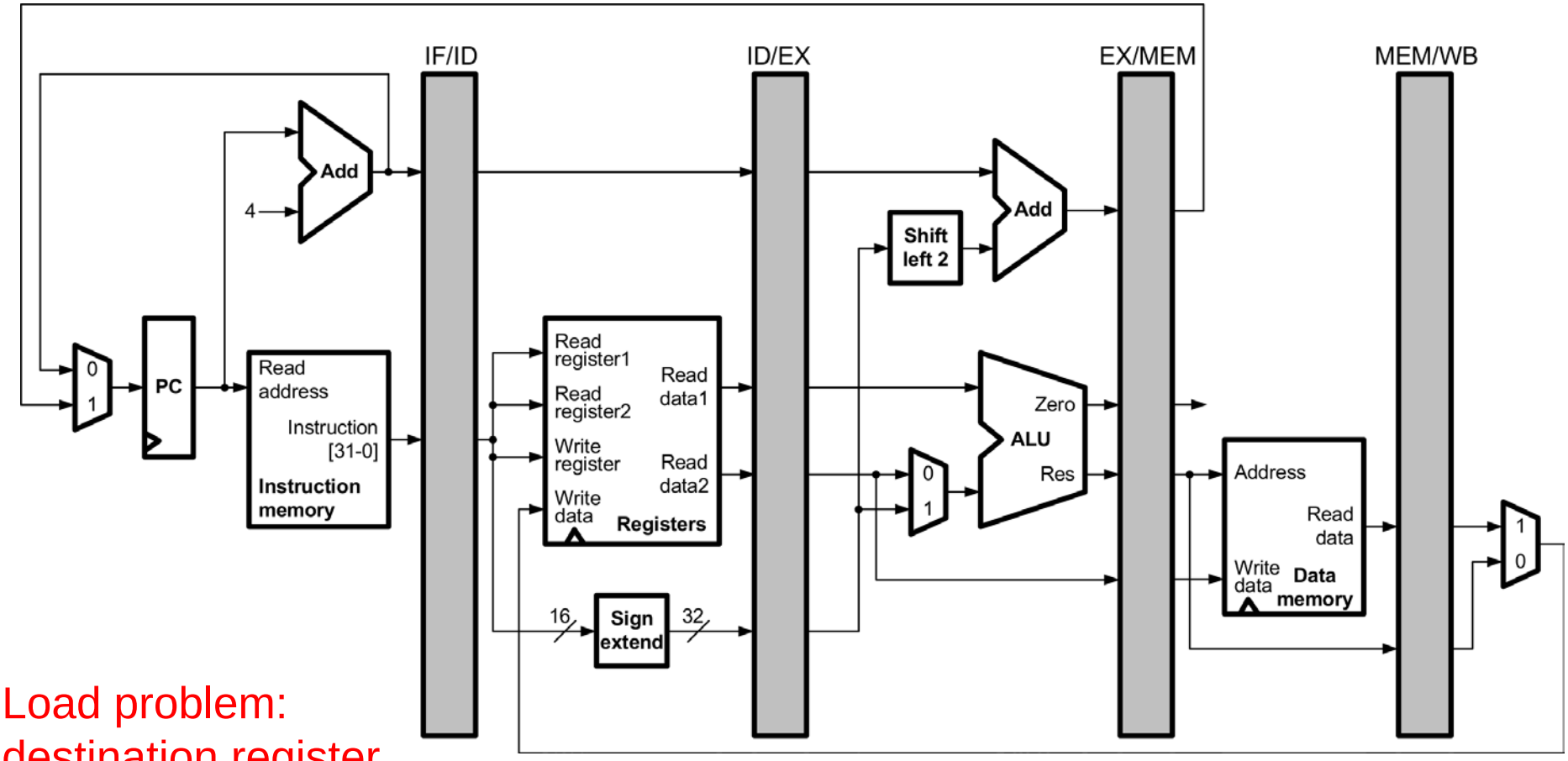
Pipelining in MIPS

- MIPS instruction set has been designed for pipelining



Pipelining in MIPS

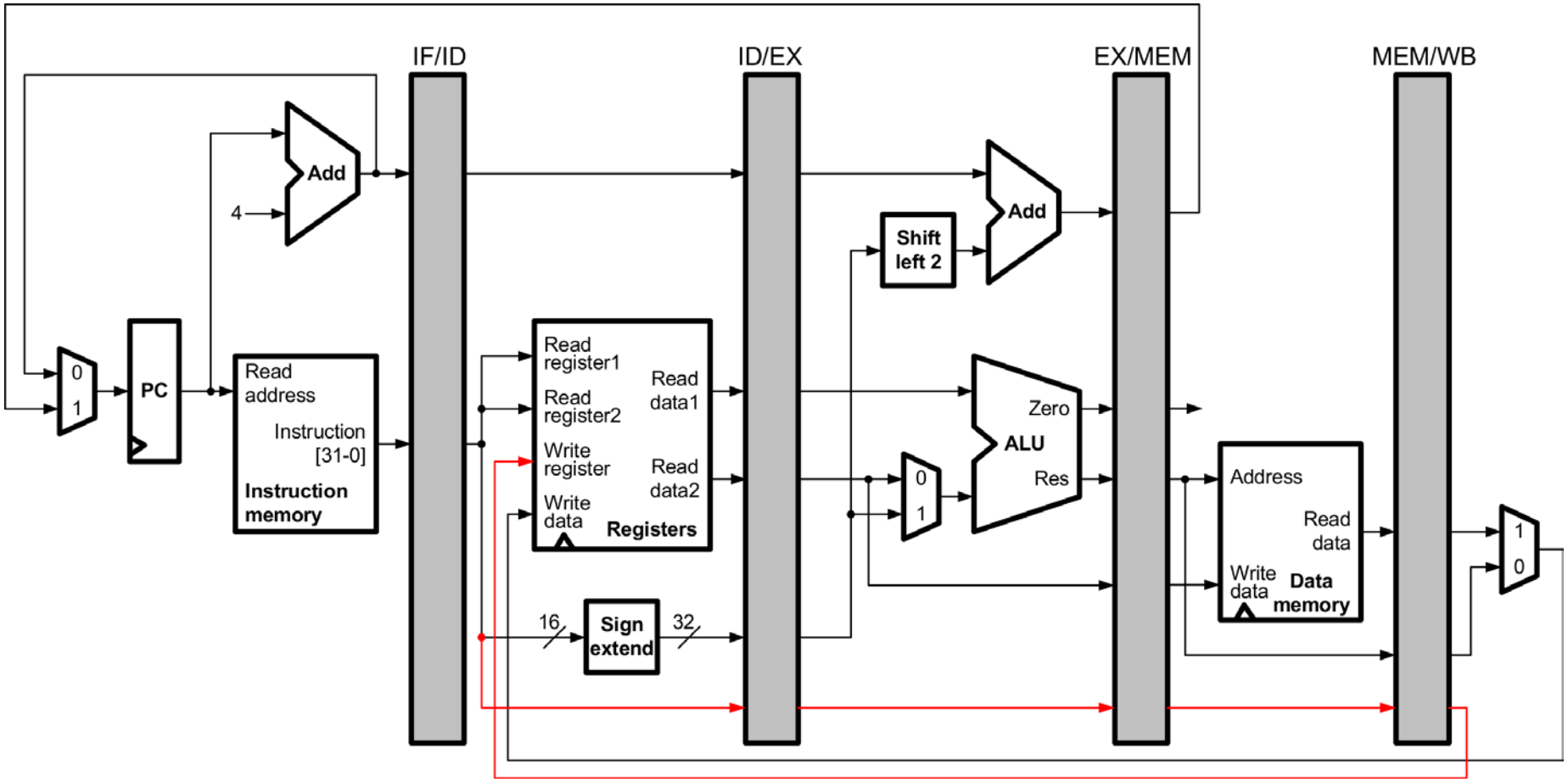
- MIPS instruction set has been designed for pipelining



Load problem: destination register

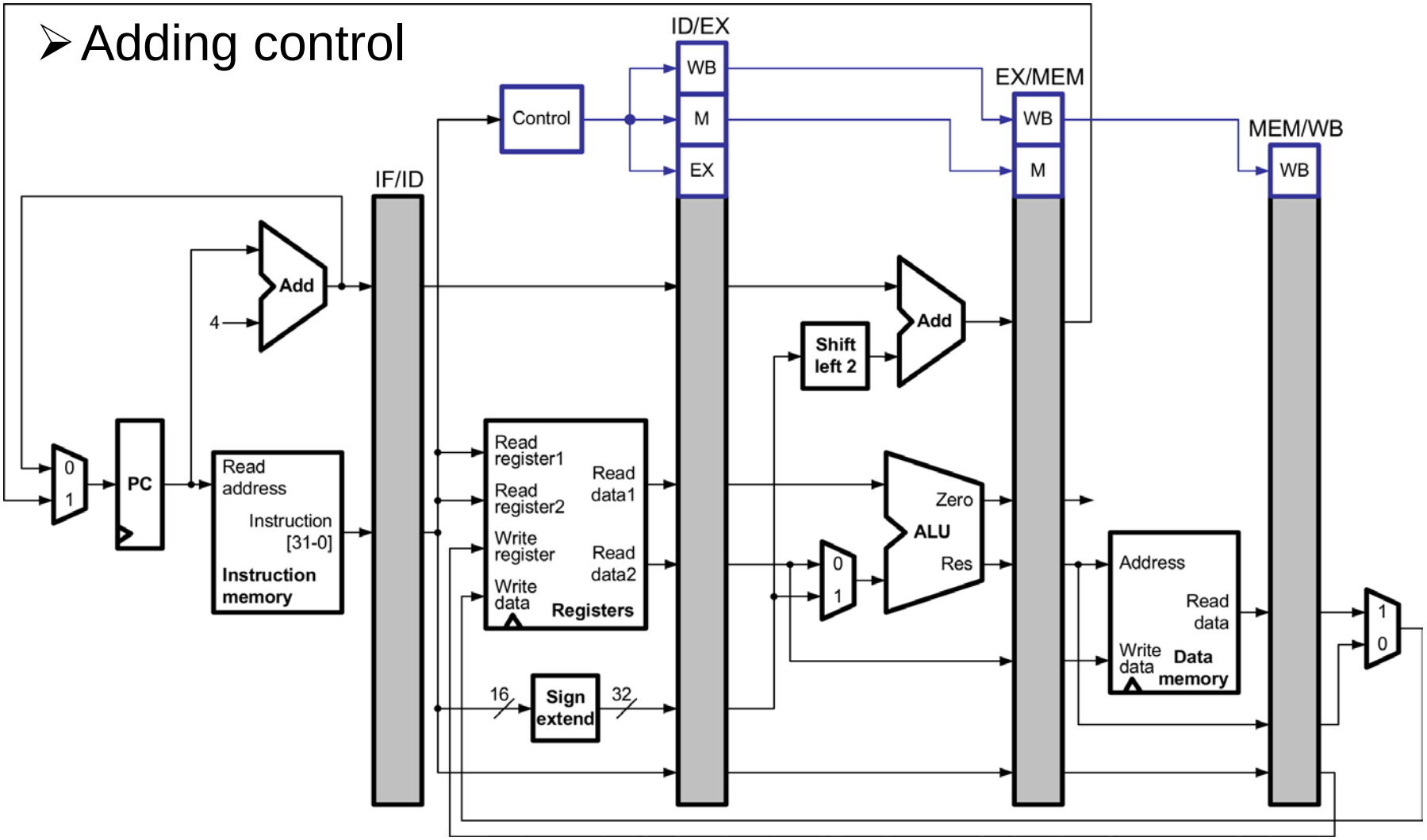
Pipelining in MIPS

- Passing destination register



Pipelining in MIPS

- Adding control



Pipeline hazards

Three categories:

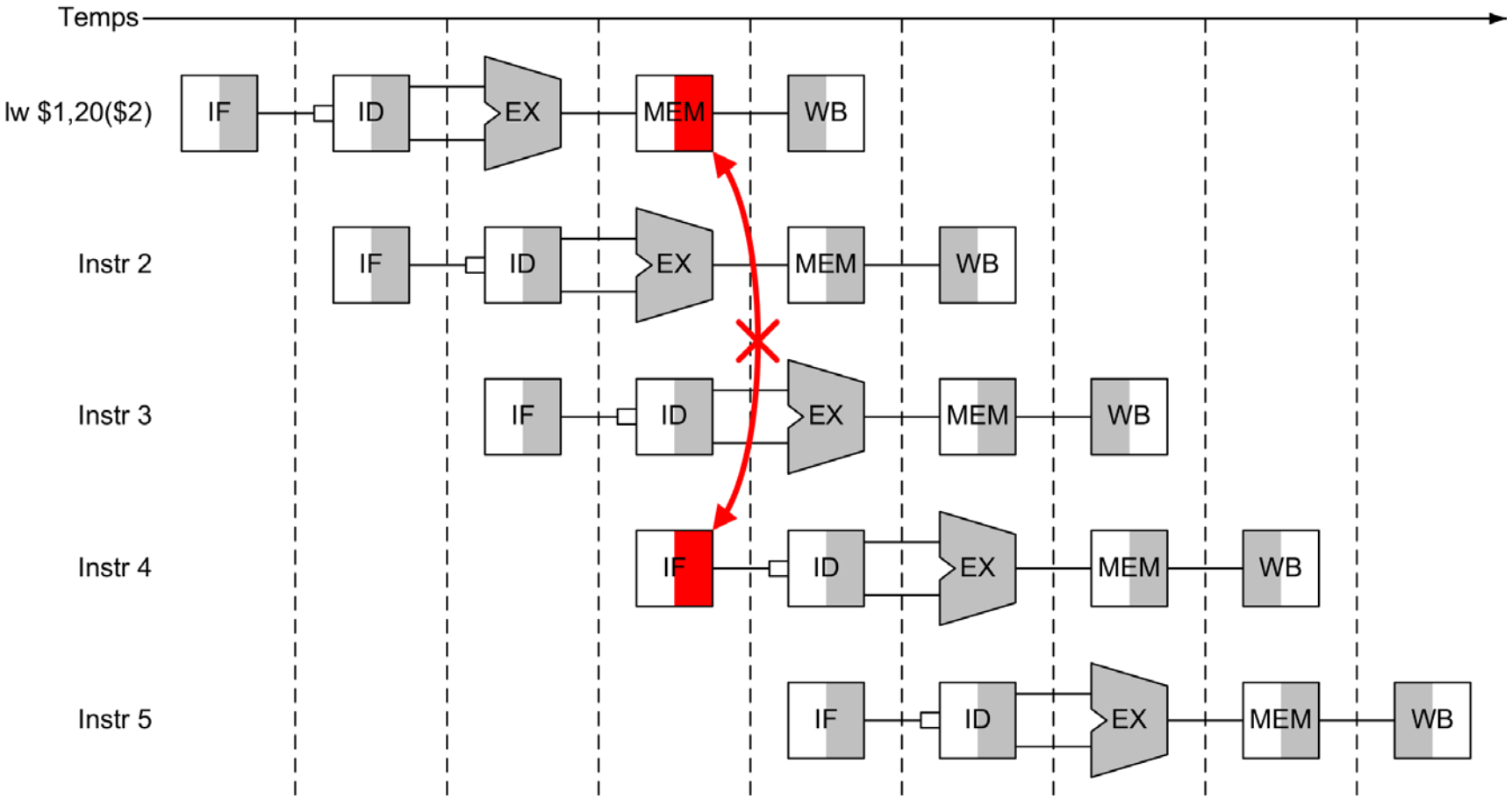
- Structural hazard
- Data hazard
- Control hazard

Pipeline hazards

- Structural hazard
  - The hardware cannot execute the combination of instructions in the pipeline at the same time
  - Example: With only one memory for instructions/data

Pipeline hazards

- Structural hazard



Pipeline hazards

- Structural hazard
  - The hardware cannot execute the combination of instructions in the pipeline at the same time
  - Example: With only one memory for instructions/data

- Data hazard
  - When one step in the pipeline must wait for another to complete
  - Example:

```
add   $1, $2, $3
sub   $4, $1, $5
```

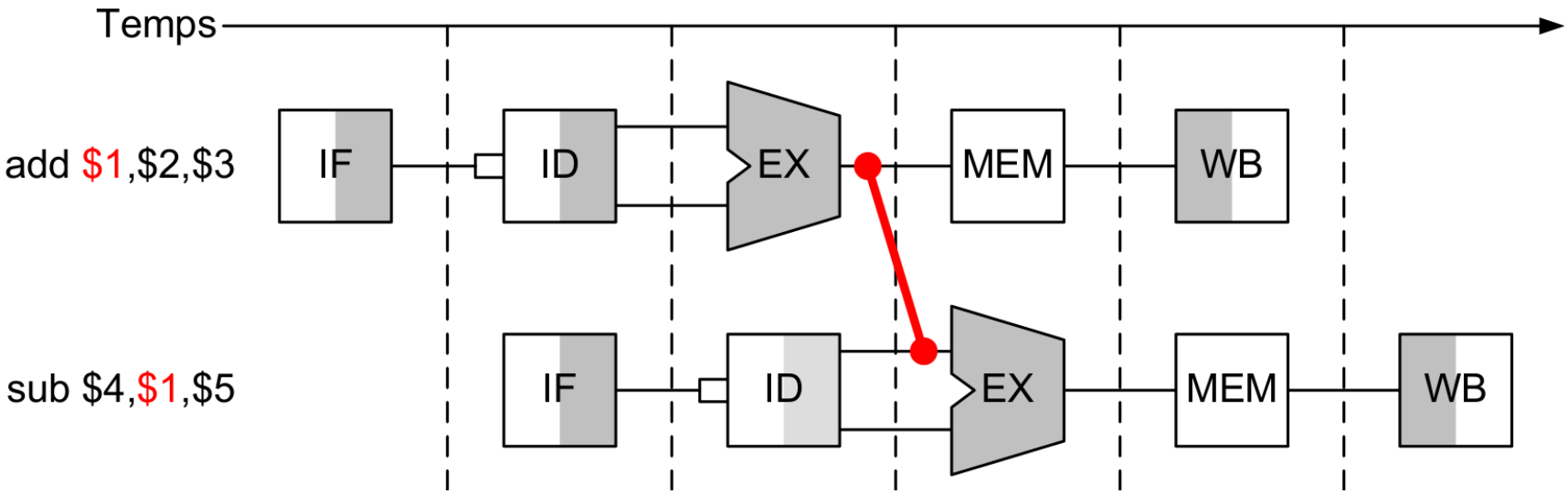
Pipeline hazards

- Data hazard types

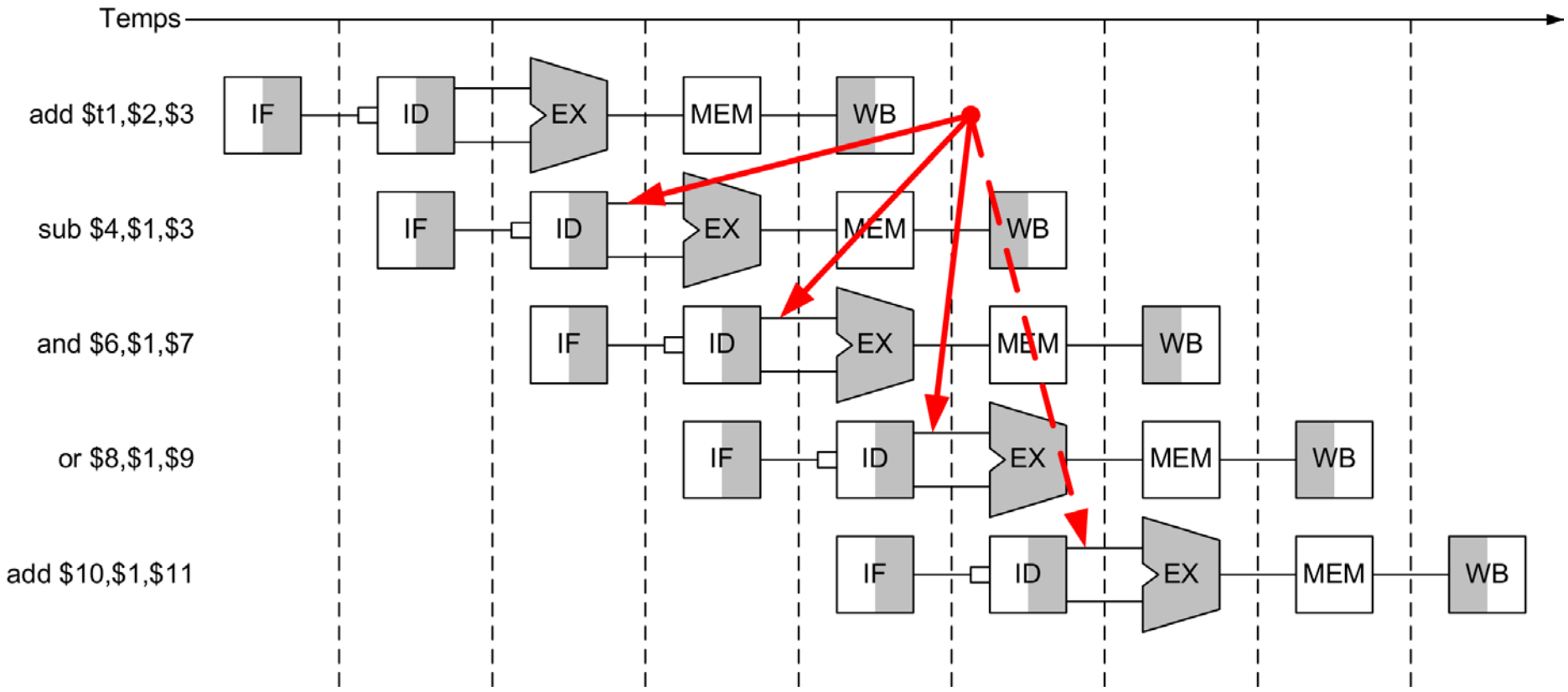
- For two consecutive instructions i_1, i_2 :
  - RAW (Read After Write)
    - i_2 tries to read before i_1 has written
  - WAR (Write After Read)
    - i_2 tries to write before i_1 has read
    - Not possible in MIPS
  - WAW (Write After Write)
    - i_2 tries to write before i_1 has written
    - Writing order modified
    - Not possible in MIPS

Data hazard

- RAW (Read After Write)



Data hazard. Problem

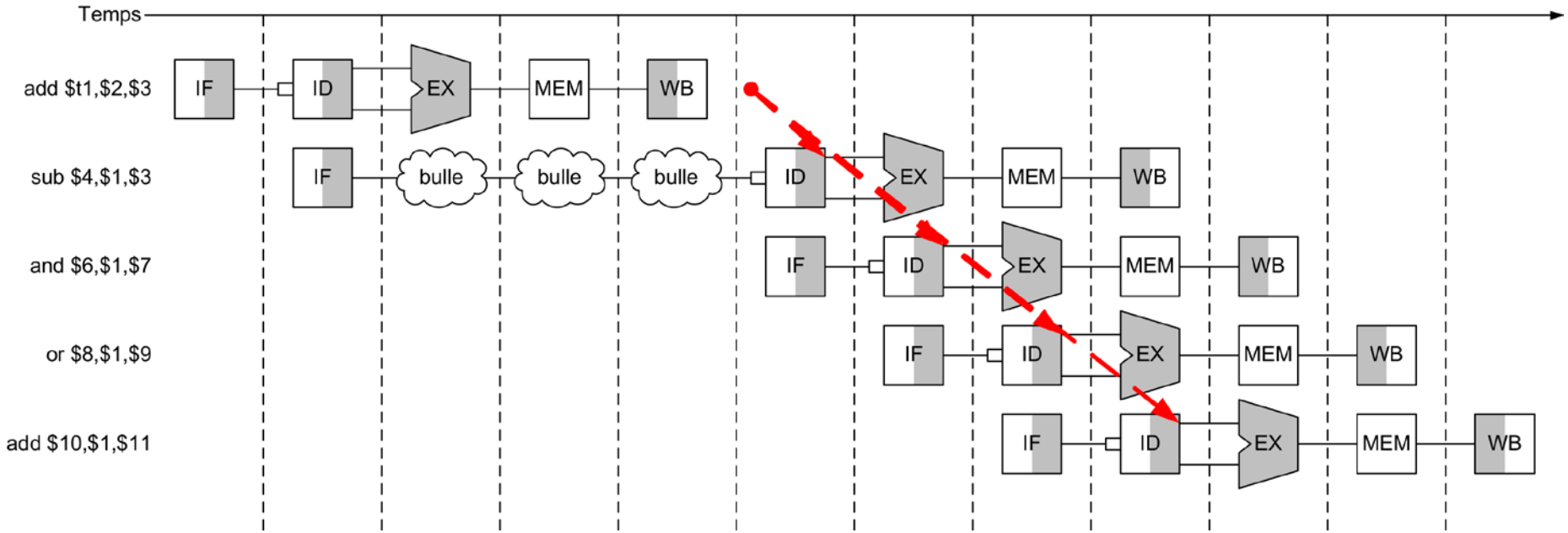


Data hazard. Solutions

- Stall
  - Freeze pipeline waiting for the result

Data hazard. Solutions

- Stall

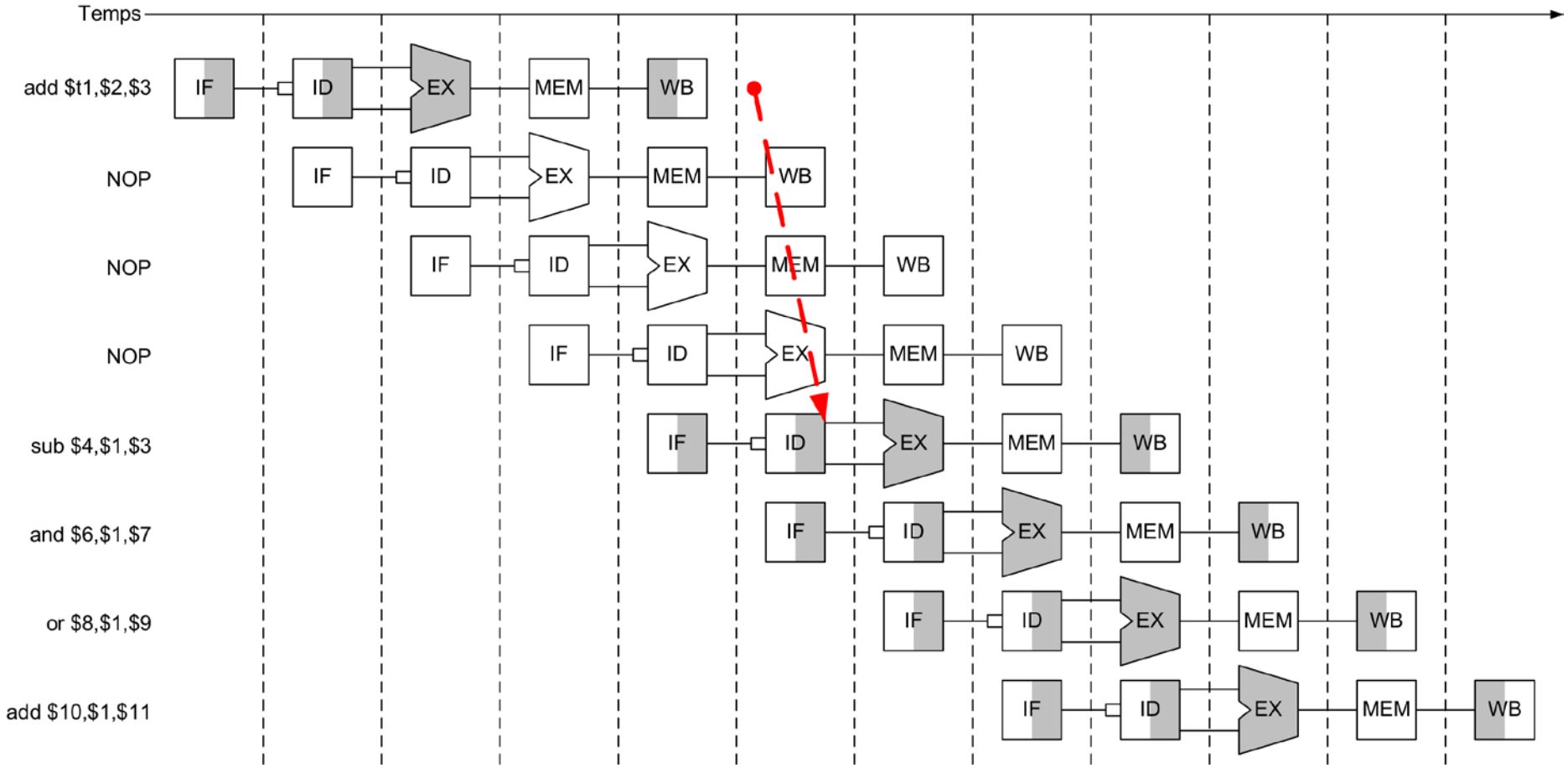


Data hazard. Solutions

- Stall
  - Freeze pipeline waiting for the result
  - Using NOP operation

Data hazard. Solutions

- Stall. Insertion of NOPs



Data hazard. Solutions

- Stall
  - Freeze pipeline waiting for the result
  - Using NOP operation
  - Problem: Increase of CPI

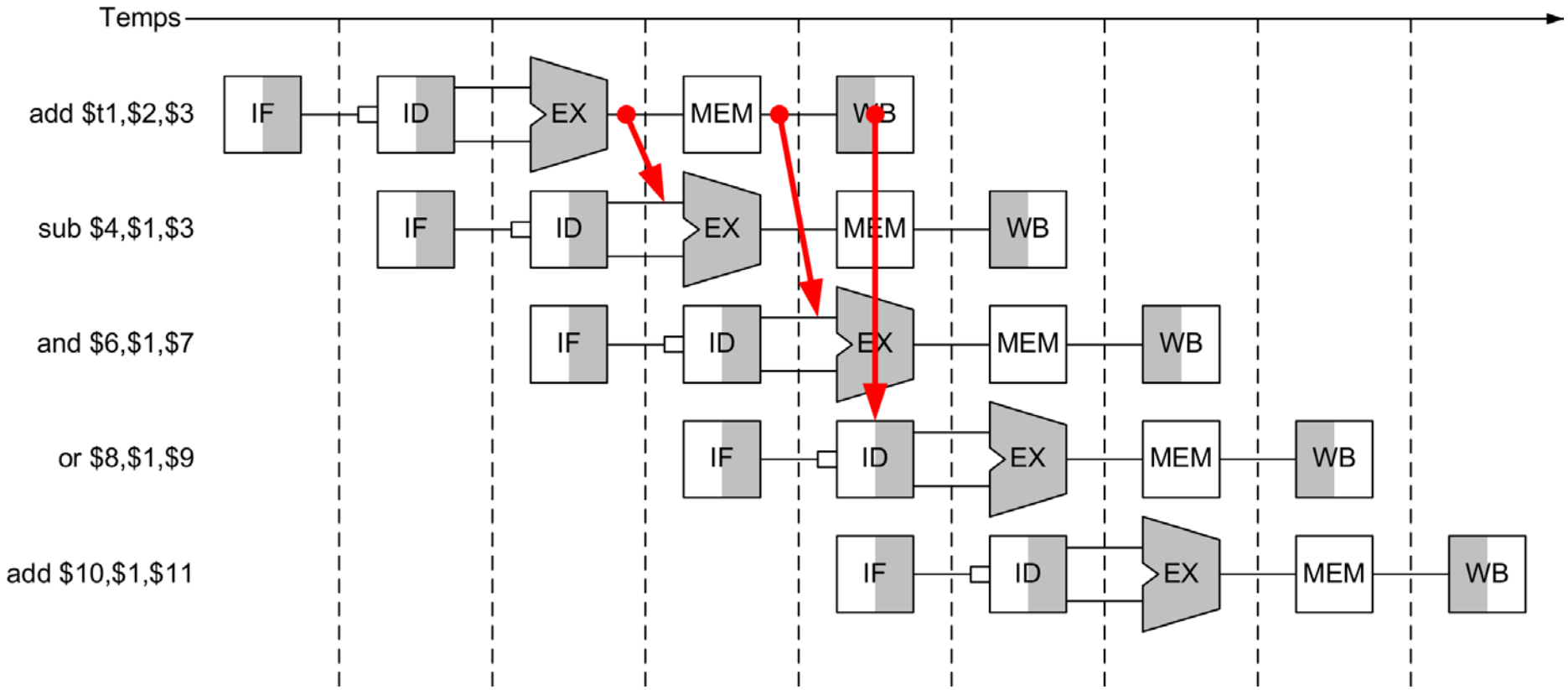
Data hazard. Solutions

- Stall
  - Freeze pipeline waiting for the result
  - Using NOP operation
  - Problem: Increase of CPI

- Forwarding
  - Bypassing pipeline stages

Data hazard. Solutions

- Forwarding

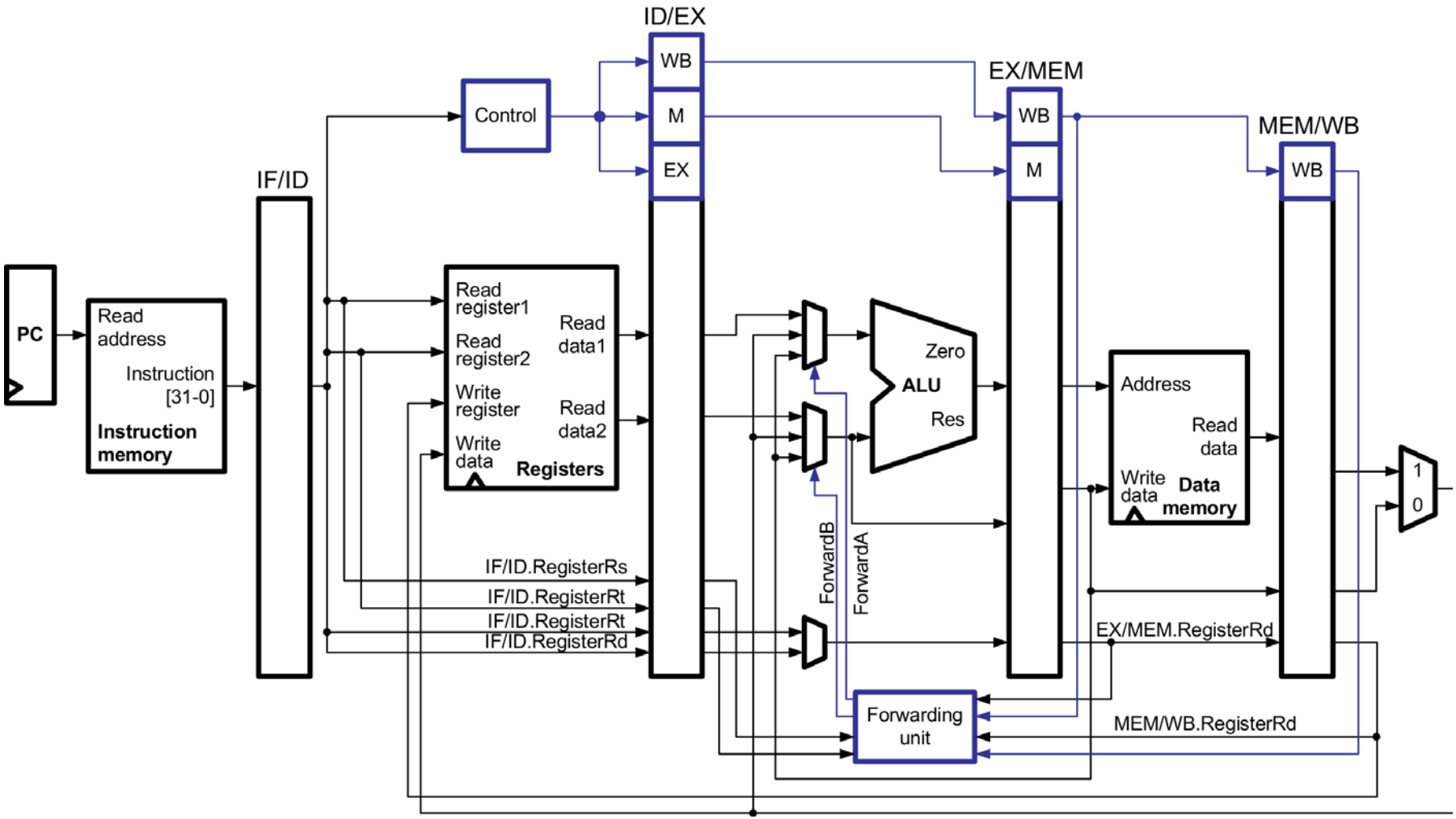


Data hazard. Solutions

- Stall
  - Freeze pipeline waiting for the result
  - Using NOP operation
  - Problem: Increase of CPI

- Forwarding
  - Bypassing pipeline stages
  - Increases number of inputs and controls to each stage multiplexers (area increase)

Pipelining in MIPS (with forwarding)



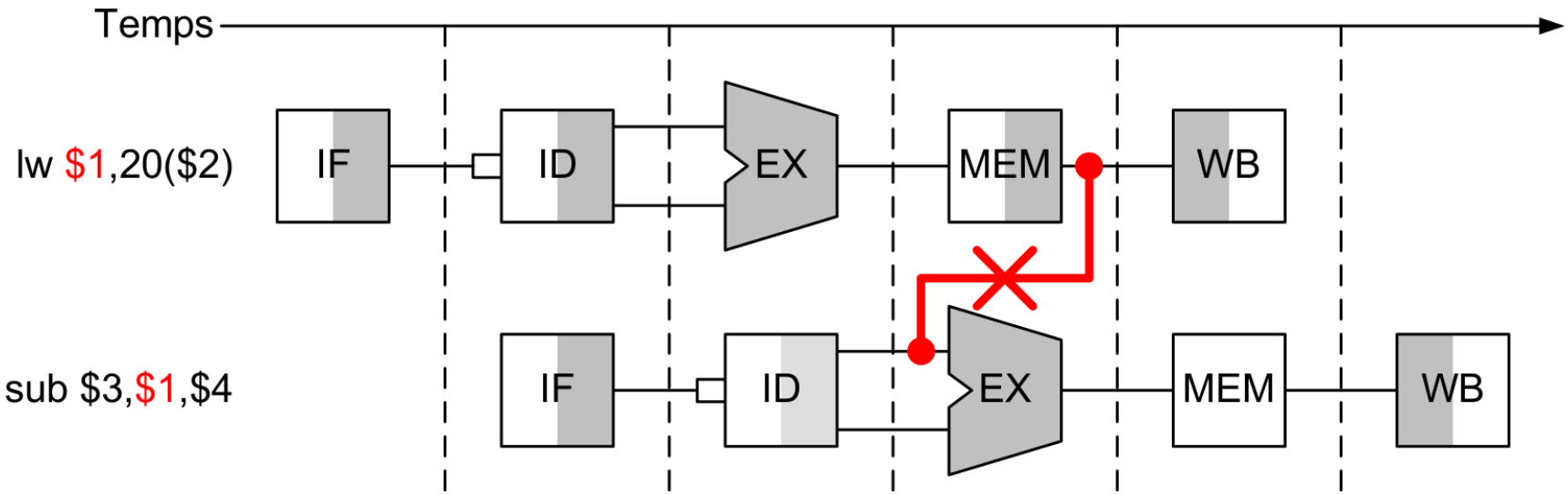
Data hazard. Solutions

- Stall
  - Freeze pipeline waiting for the result
  - Using NOP operation
  - Problem: Increase of CPI

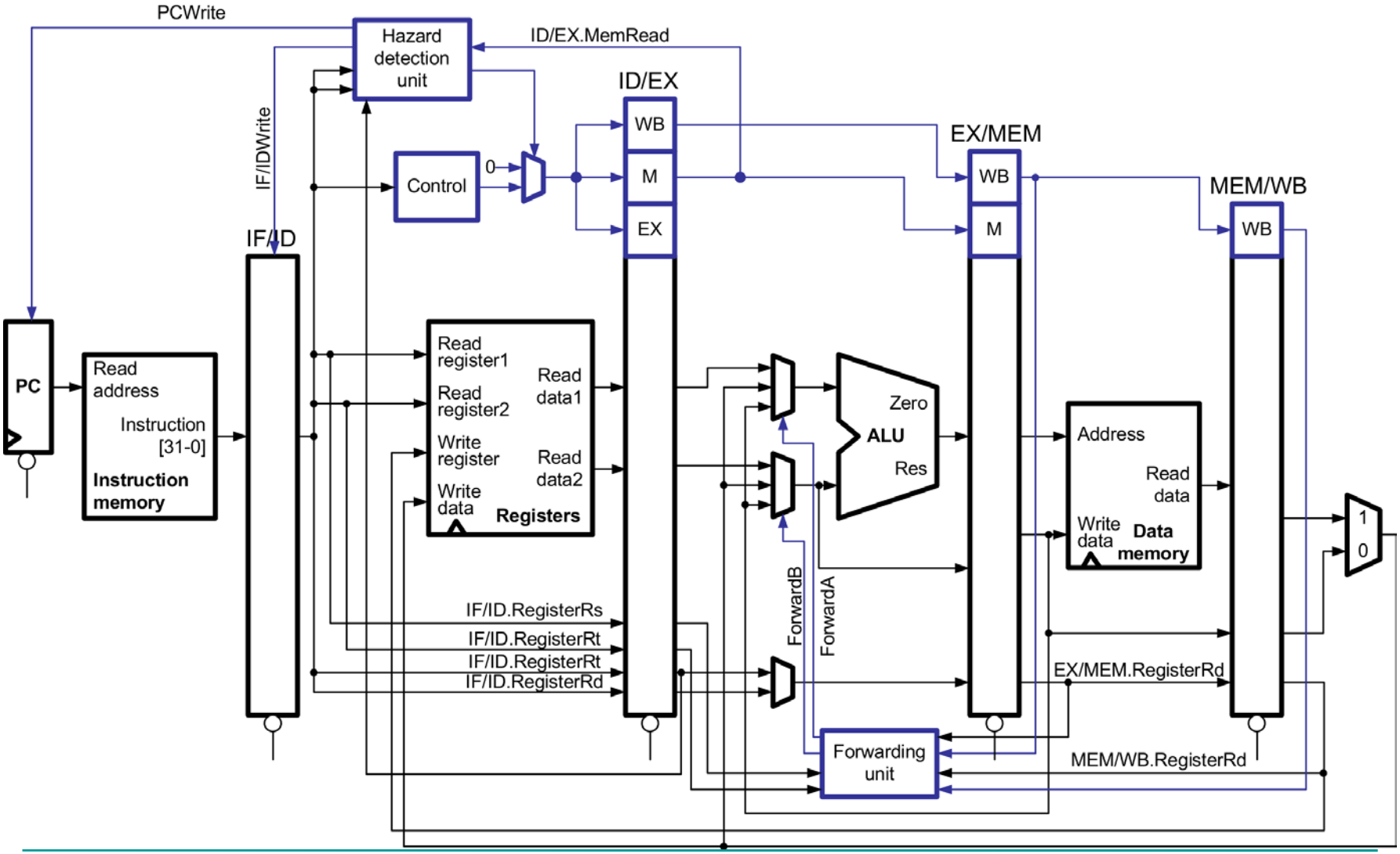
- Forwarding
  - Bypassing pipeline stages
  - Increases number of inputs and controls to each stage multiplexers (area increase)
  - Not always applicable: Load-use hazard

Data hazard

- Load-use hazard
  - Solution: Detection and injection of NOP



Pipelining in MIPS (forwarding + hazard detection)



Data hazard. Solutions

- Stall
  - Freeze pipeline waiting for the result
  - Using NOP operation
  - Problem: Increase of CPI

- Forwarding
  - Bypassing pipeline stages
  - Increases number of inputs and controls to each stage multiplexers (area increase)
  - Not always applicable: Load-use hazard

Pipeline hazards

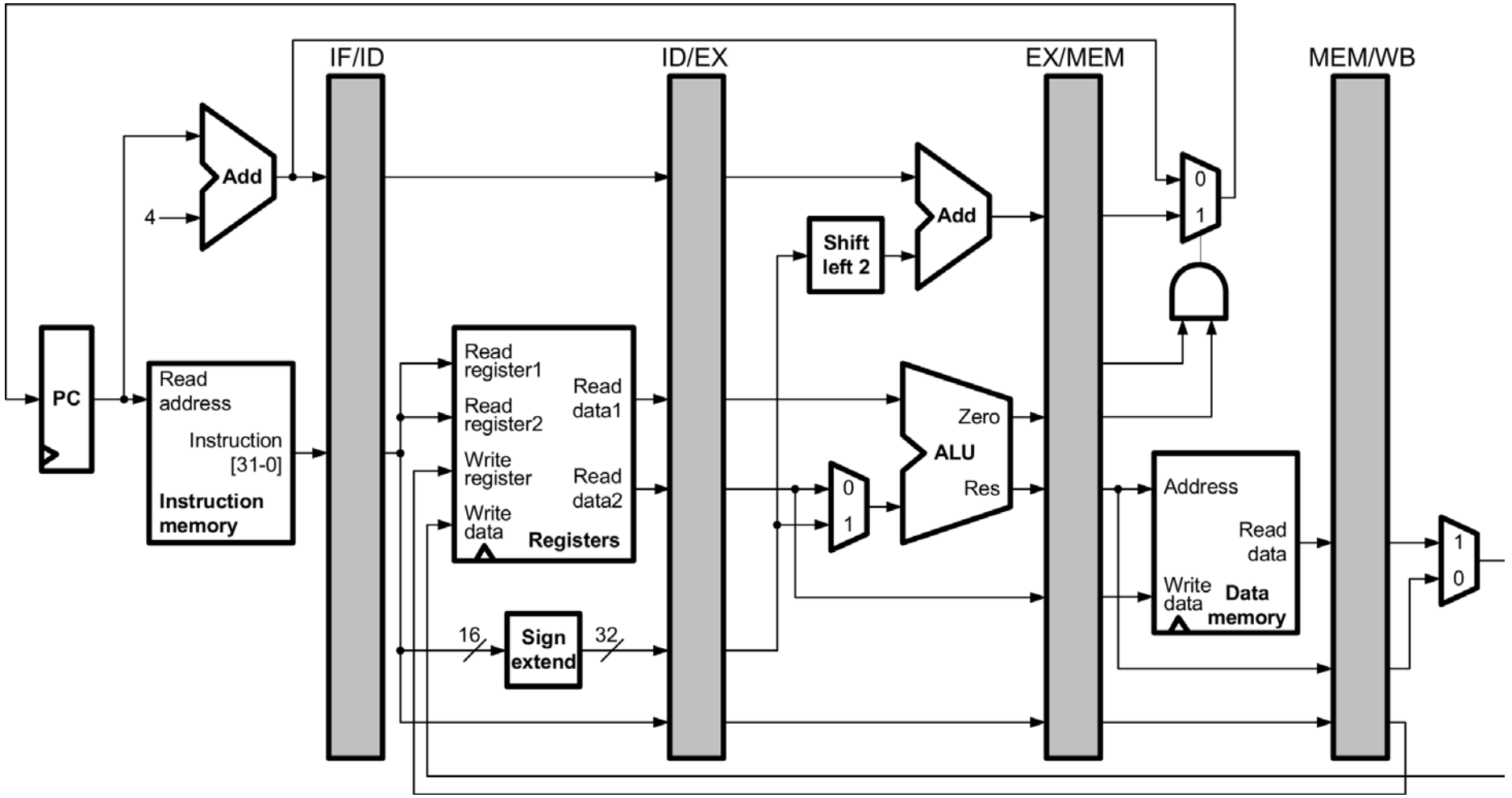
- Structural hazard
  - The hardware cannot execute the combination of instructions in the pipeline at the same time
  - Example: With only one memory for instructions/data

- Data hazard
  - When one step in the pipeline must wait for another to complete
  - Example:

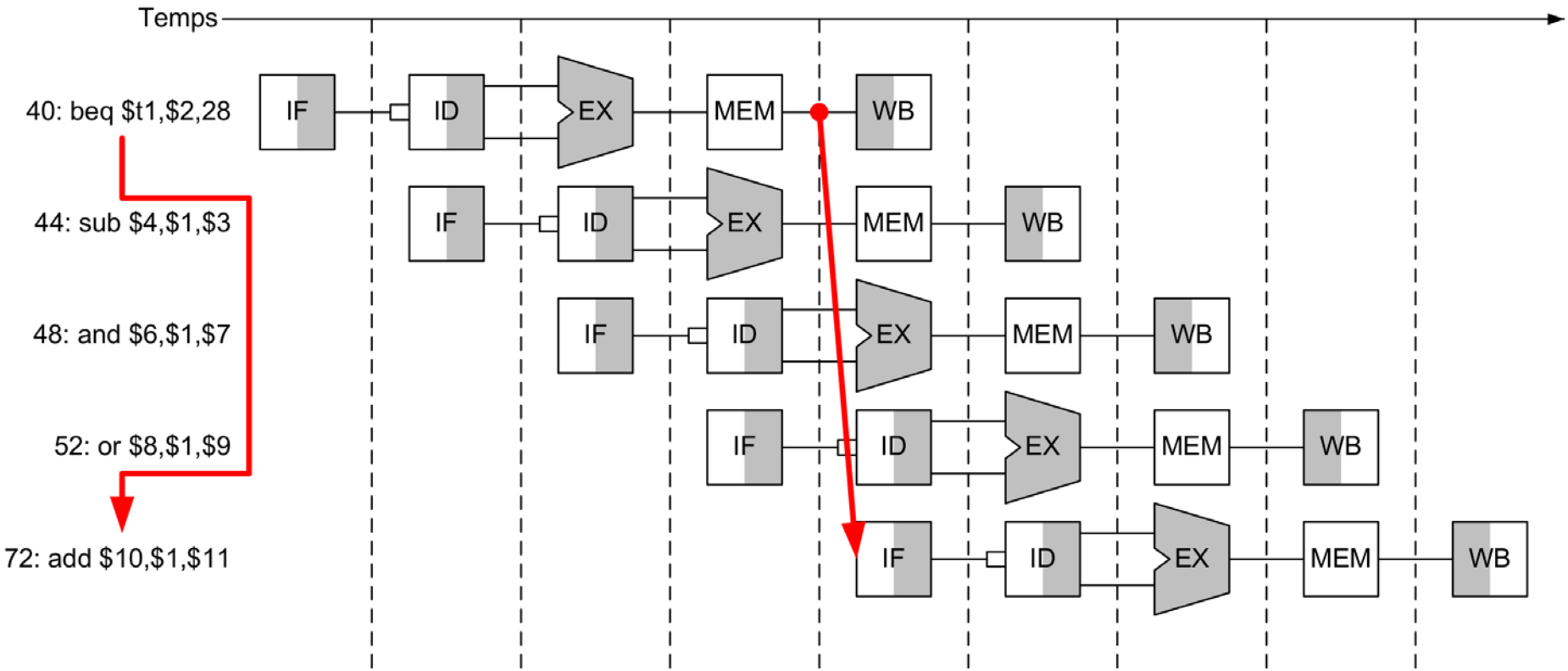
```
add   $1, $2, $3
sub   $4, $1, $5
```

- Control hazard or branch hazard
  - When next instruction to execute depends on the result of the previous instruction (branching, jumps)

Pipelining in MIPS (branching)



Control hazard. Problem



Control hazard. Solutions

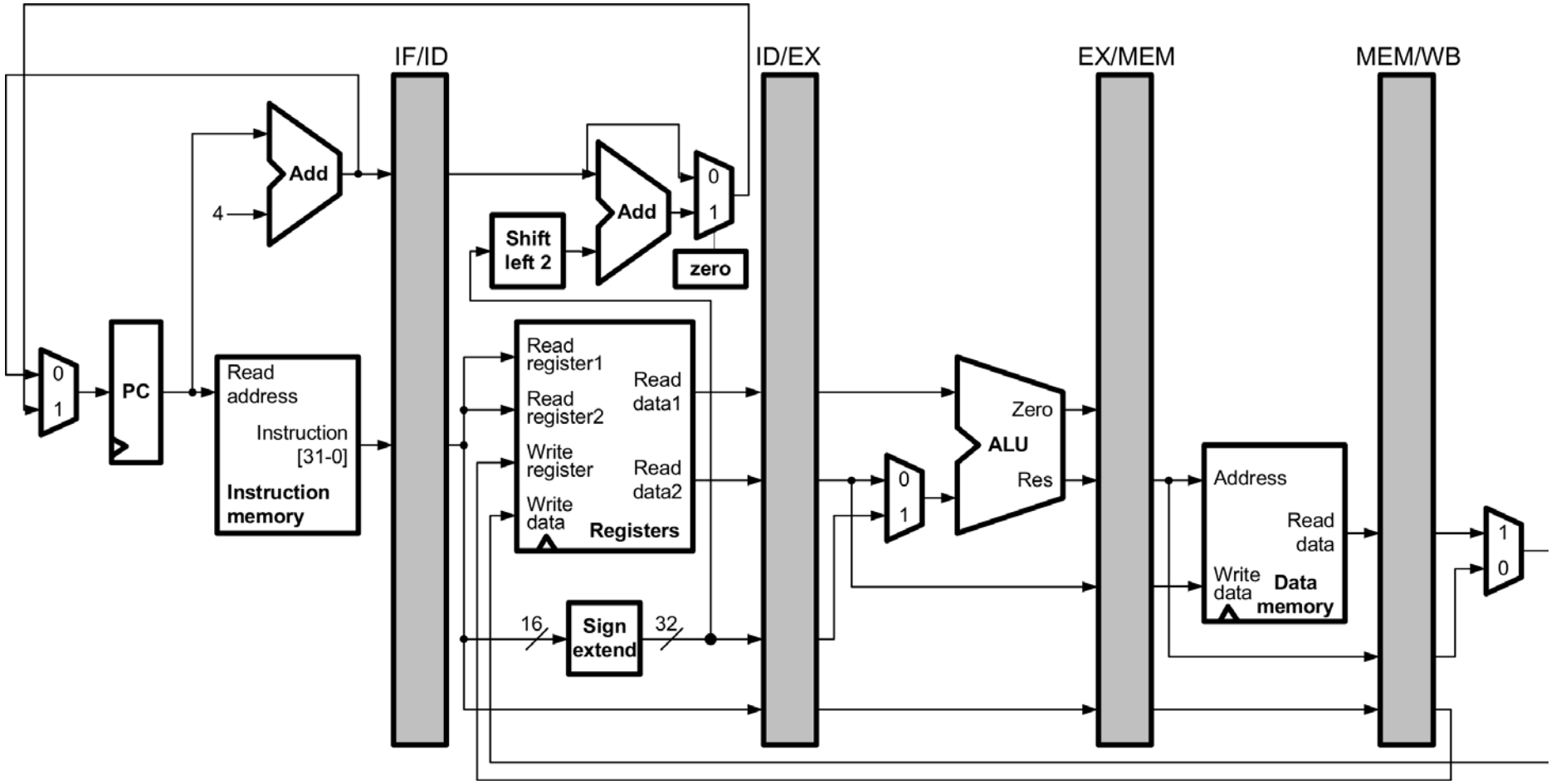
Control hazard. Solutions

- Stall
  - Freeze pipeline waiting for the result
  - Increase of CPI. Quite inefficient (worse for deeper pipelines)

Control hazard. Solutions

- Stall
  - Freeze pipeline waiting for the result
  - Increase of CPI. Quite inefficient (worse for deeper pipelines)
- Anticipated branching
  - Only one NOP needed

Pipelining in MIPS (anticipated branching)

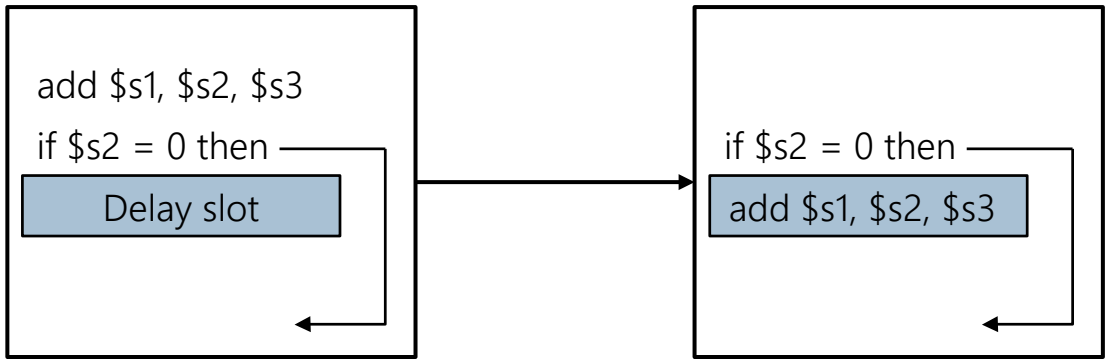


Control hazard. Solutions

- Stall
  - Freeze pipeline waiting for the result
  - Increase of CPI. Quite inefficient (worse for deeper pipelines)
- Anticipated branching
  - Only one NOP needed

Control hazard. Solutions

- Stall
  - Freeze pipeline waiting for the result
  - Increase of CPI. Quite inefficient (worse for deeper pipelines)
- Anticipated branching
  - Only one NOP needed
- Branch delay slot
  - Following instruction (unrelated to branching) is always executed



Control hazard. Solutions

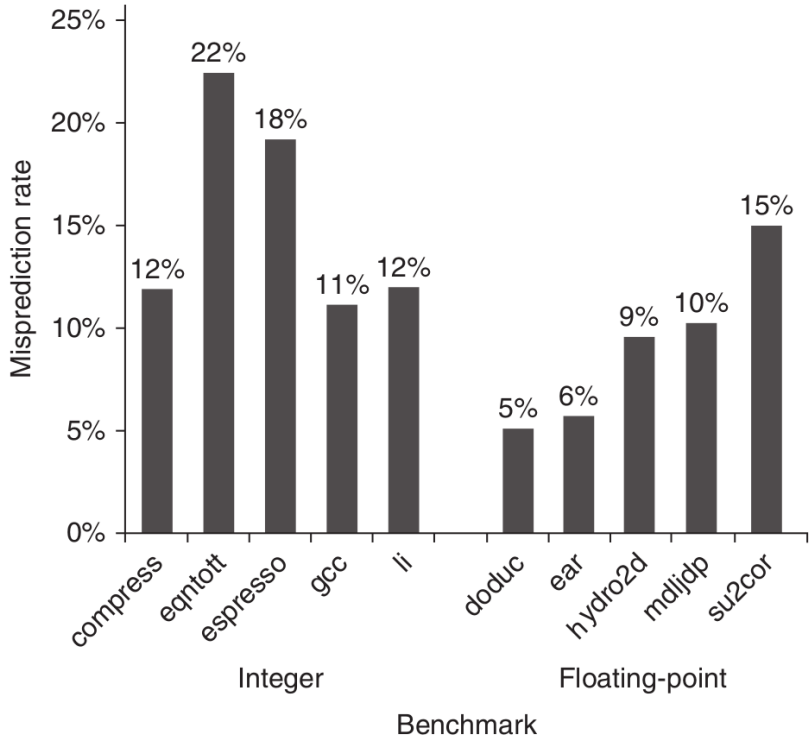
- Stall
  - Freeze pipeline waiting for the result
  - Increase of CPI. Quite inefficient (worse for deeper pipelines)
- Anticipated branching
  - Only one NOP needed
- Branch delay slot
  - Following instruction (unrelated to branching) is always executed
- Speculation
  - Static branch prediction

Static branch prediction

- Assume branch not taken
  - Simple but inefficient. If branch taken, flush pipeline
- Fixed decision implemented in some architectures

Static branch prediction

- Assume branch not taken
  - Simple but inefficient. If branch taken, flush pipeline
- Fixed decision implemented in some architectures
- With MIPS, the compiler can decide
  - Example of execution of SPEC92:



Control hazard. Solutions

- Stall
  - Freeze pipeline waiting for the result
  - Increase of CPI. Quite inefficient (worse for deeper pipelines)
- Anticipated branching
  - Only one NOP needed
- Branch delay slot
  - Following instruction (unrelated to branching) is always executed
- Speculation
  - Static branch prediction
  - Dynamic branch prediction

Dynamic branch prediction

- Decision taken during execution
- Wrong decision also causes flushing the pipeline

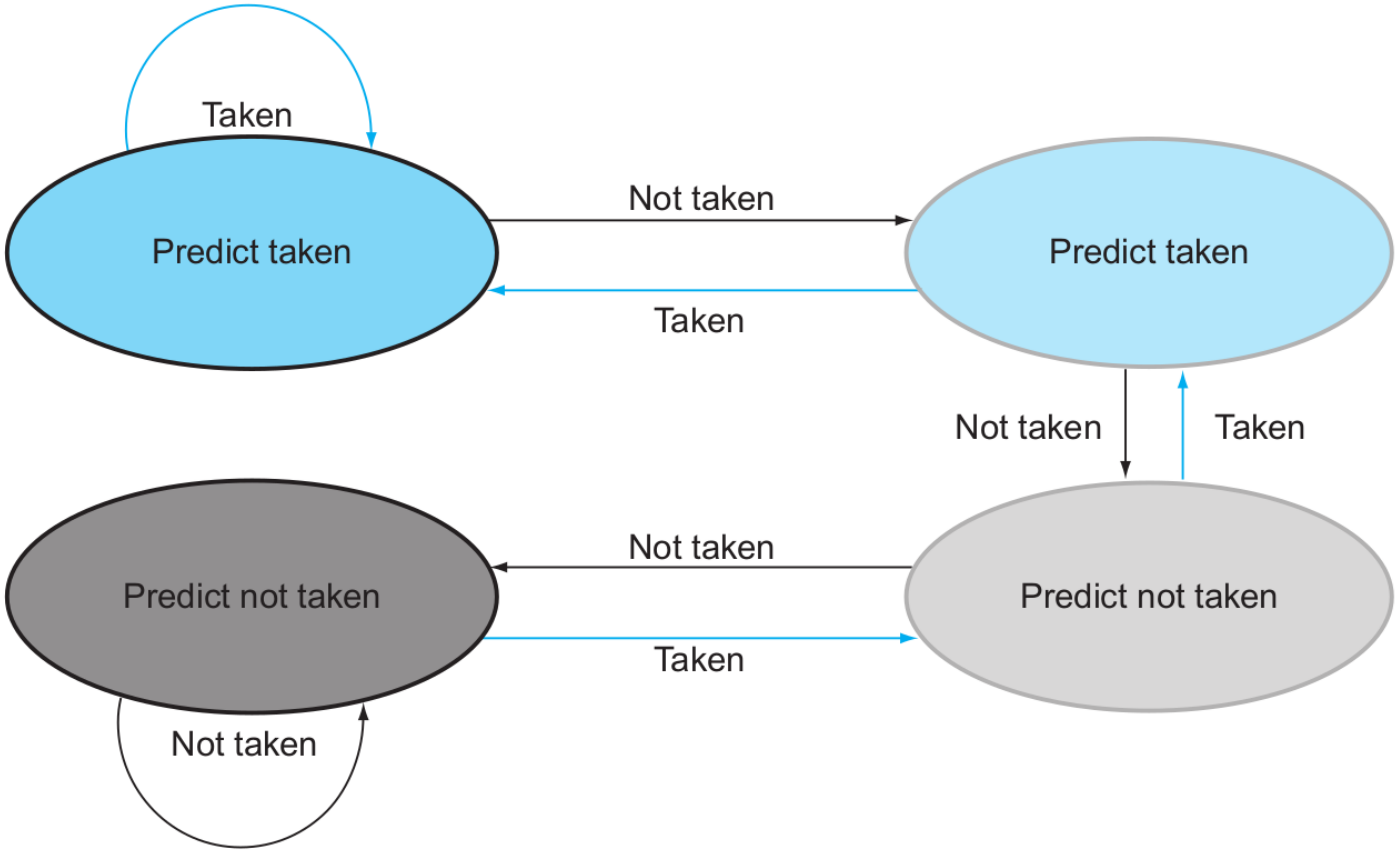
Dynamic branch prediction

- Decision taken during execution
- Wrong decision also causes flushing the pipeline
- A simple implementation is to use a Branch History Table (BHT) or branch prediction buffer
  - Small memory containing: lower portion of the address of the branch instruction and a bit that says if branch was recently taken
    - Simplest version. Inefficient with regular loops

Dynamic branch prediction

- Decision taken during execution
- Wrong decision also causes flushing the pipeline
- A simple implementation is to use a Branch History Table (BHT) or branch prediction buffer
  - Small memory containing: lower portion of the address of the branch instruction and a bit that says if branch was recently taken
    - Simplest version. Inefficient with regular loops
  - Saving 2 bits in each position:
    - Counter increments when taken, otherwise decrements
    - Branching is taken when the MSB is ‘1’

2-bit Branch History Table



Dynamic branch prediction

- Decision taken during execution
- Wrong decision also causes flushing the pipeline
- A simple implementation is to use a Branch History Table (BHT) or branch prediction buffer
  - Small memory containing: lower portion of the address of the branch instruction and a bit that says if branch was recently taken
    - Simplest version. Inefficient with regular loops
  - Saving 2 bits in each position:
    - Counter increments when taken, otherwise decrements
    - Branching is taken when the MSB is ‘1’
- Many other methods: target buffers, correlating predictors…

How to Speed Up More?

- SuperPipelining
  - Pipe structure at multiple levels

- More parallelism
  - Superscalar machines
  - SIMD (single-instruction multiple data)
  - MIMD (multiple-instruction multiple data)

- Dynamic scheduling
  - OoO (out-of-order) CPUs

Pipelining summary

- Exploits parallelism, increases performance
  - Increasing frequency of operation and maintaining a low CPI
- Structure, data and control hazards to be considered
- Compilers can optimize code to avoid hazards
- More stages in the pipeline lowers t_{CLK}, increases data hazards
- Interrupts/exceptions

Nios-II/f pipeline

- 6 stages:
  - Fetch, Decode, Execute, Memory, Align, Writeback
- Branch prediction selectable: Static/dynamic
  - Dynamic branch prediction uses 2-bit BHT
- Pipeline stalls:
  - Multi-cycle instructions
  - Avalon-MM instruction master port read accesses
  - Avalon-MM data master port read/write accesses
  - Data dependencies in long latency instructions (for example: load, multiply, shift)