MIPS – Pipelining

Computer Organization

Sunday, 09 October 2022

Many slides adapted from:
Computer Organization and Design,
Patterson & Hennessy
5th Edition, © 2014, MK
and from Prof. Mary Jane Irwin, PSU

TÉCNICO LISBOA

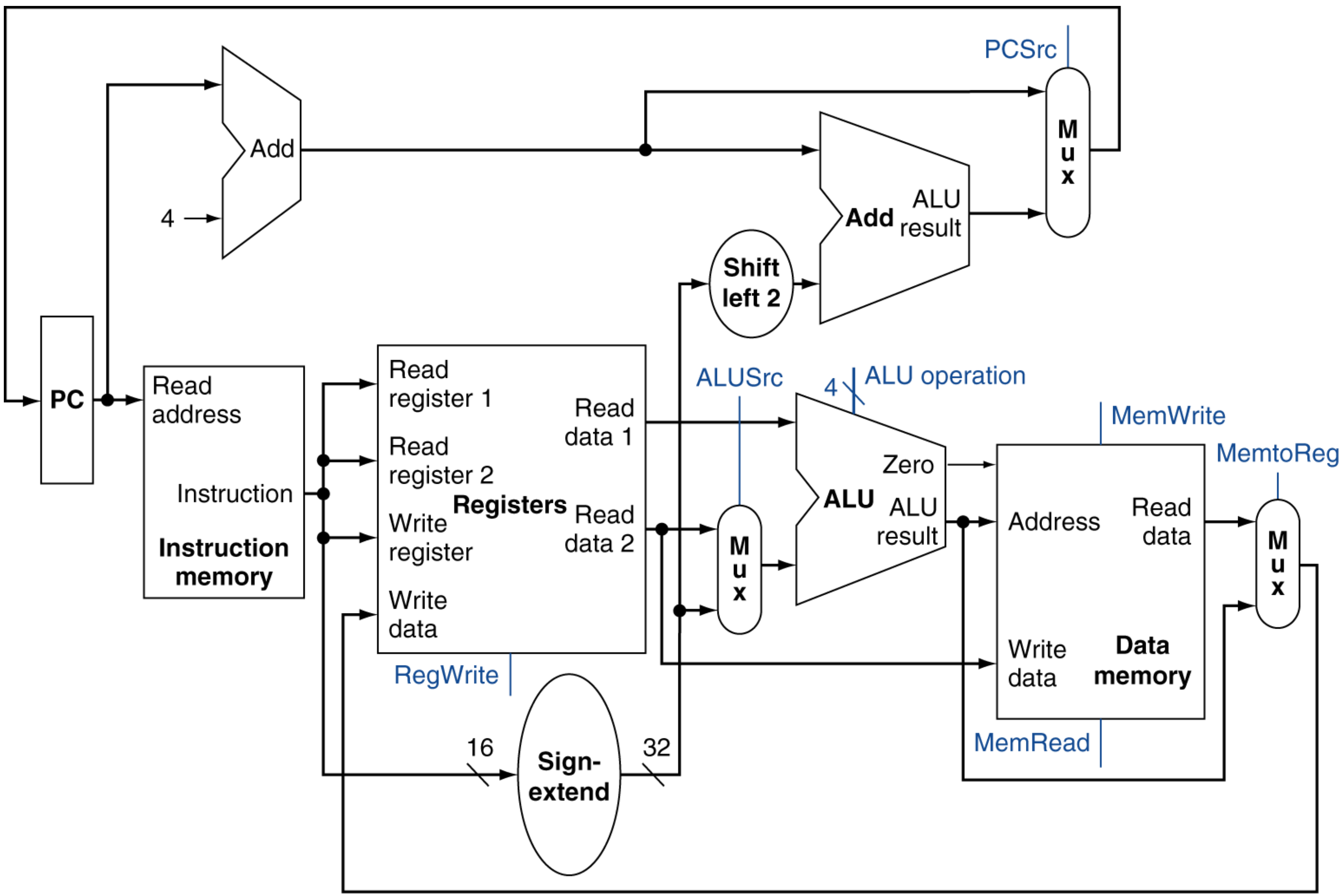
Chap. 4

Summary

- Previous Class
  - The MIPS architecture

- Today:
  - Pipeline
  - Pipeline hazards

MIPS Datapath

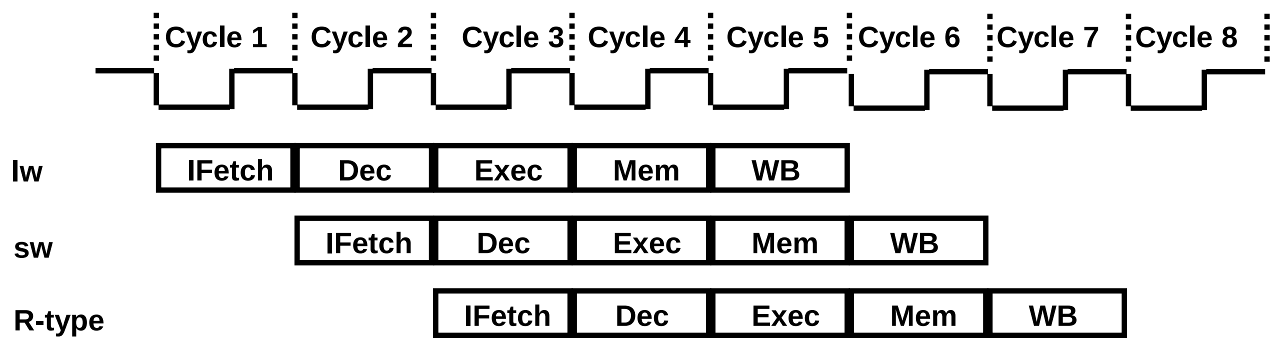


Making It Faster

- Start fetching and executing the next instruction before the current one has completed
  - Pipelining
    - modern processors are pipelined for performance

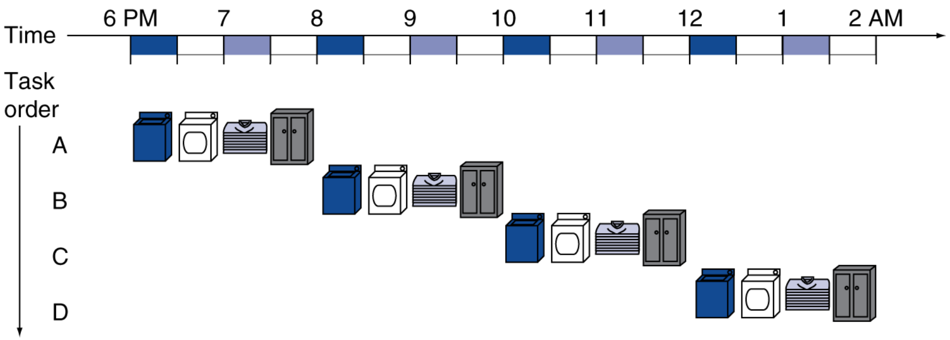
      CPU time = CPI * CC * IC

- Start the next instruction before the current one has completed
  - improves throughput
    - total amount of work done in a given time
  - instruction latency
    - (execution time, delay time, response time - time from the start of an instruction to its completion) is *not* reduced



Pipelining Analogy

- Pipelined laundry: overlapping execution
  - – Parallelism improves performance



Speedup

- Four loads:

  Speedup = 8 / 3.5 = 2.3

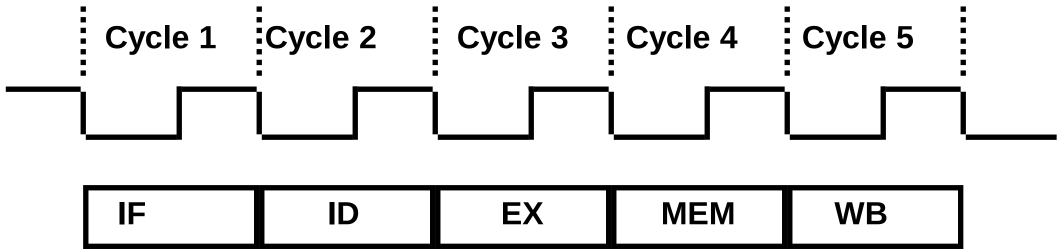
- Non-stop:

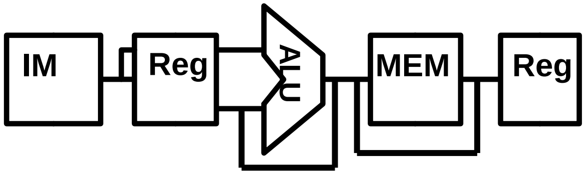
  Speedup = 2n / (0.5n + 1.5)
  ≈ 4
  = number of stages

MIPS Pipeline

Five stages, one step per stage

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



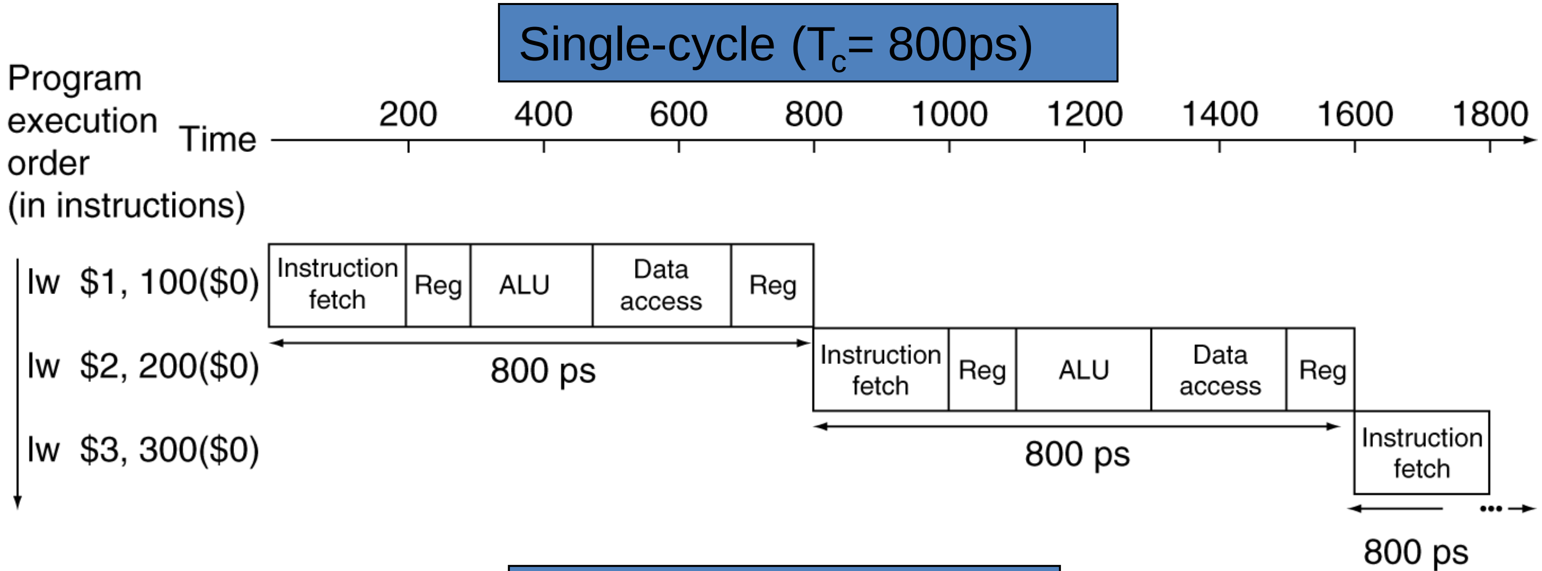


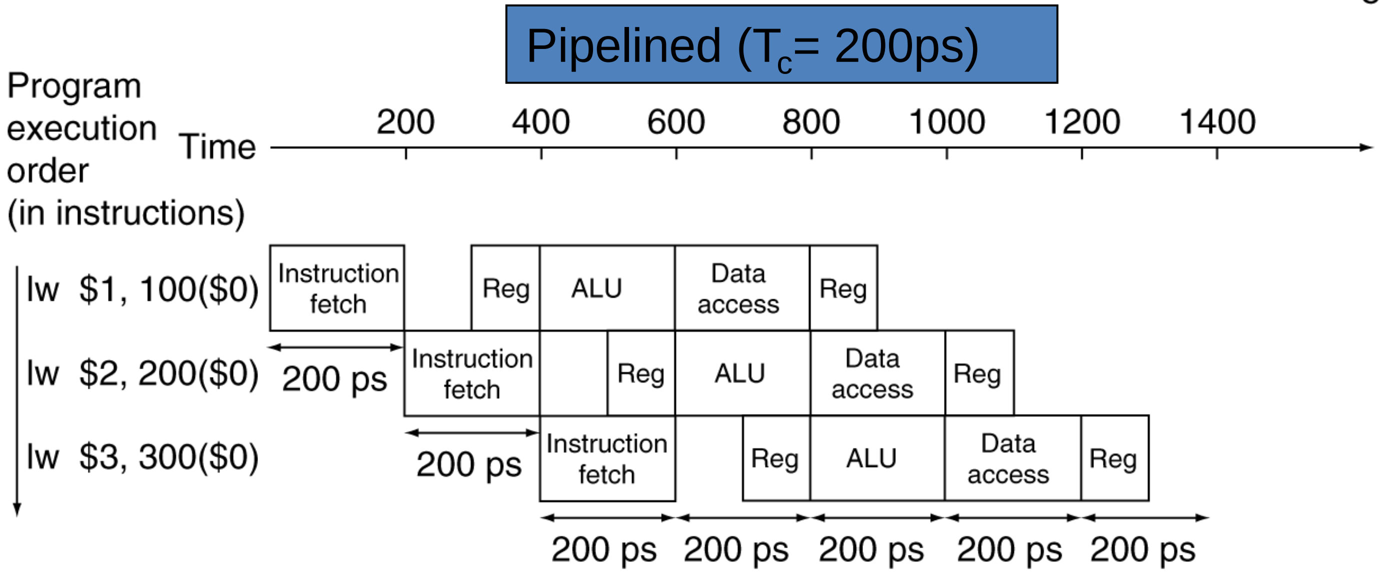
Pipeline Performance

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

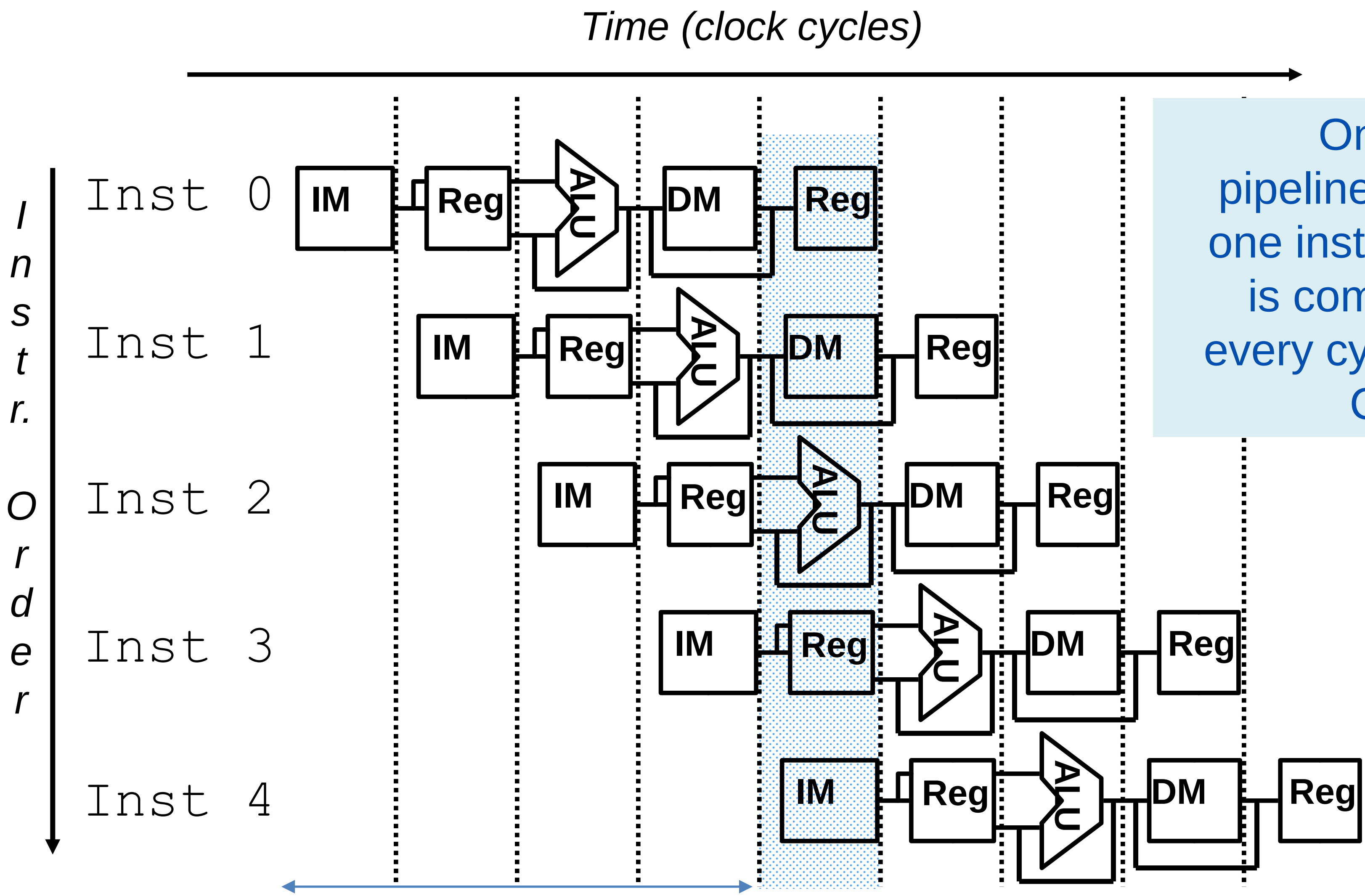
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
`lw`	200ps	100 ps	200ps	200ps	100 ps	800ps
`sw`	200ps	100 ps	200ps	200ps		700ps
`R-format`	200ps	100 ps	200ps		100 ps	600ps
`beq`	200ps	100 ps	200ps			500ps

Pipeline Performance





Pipeline for Performance



Once the pipeline is full, one instruction is completed every cycle, so CPI = 1

Time to fill the pipeline

Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$

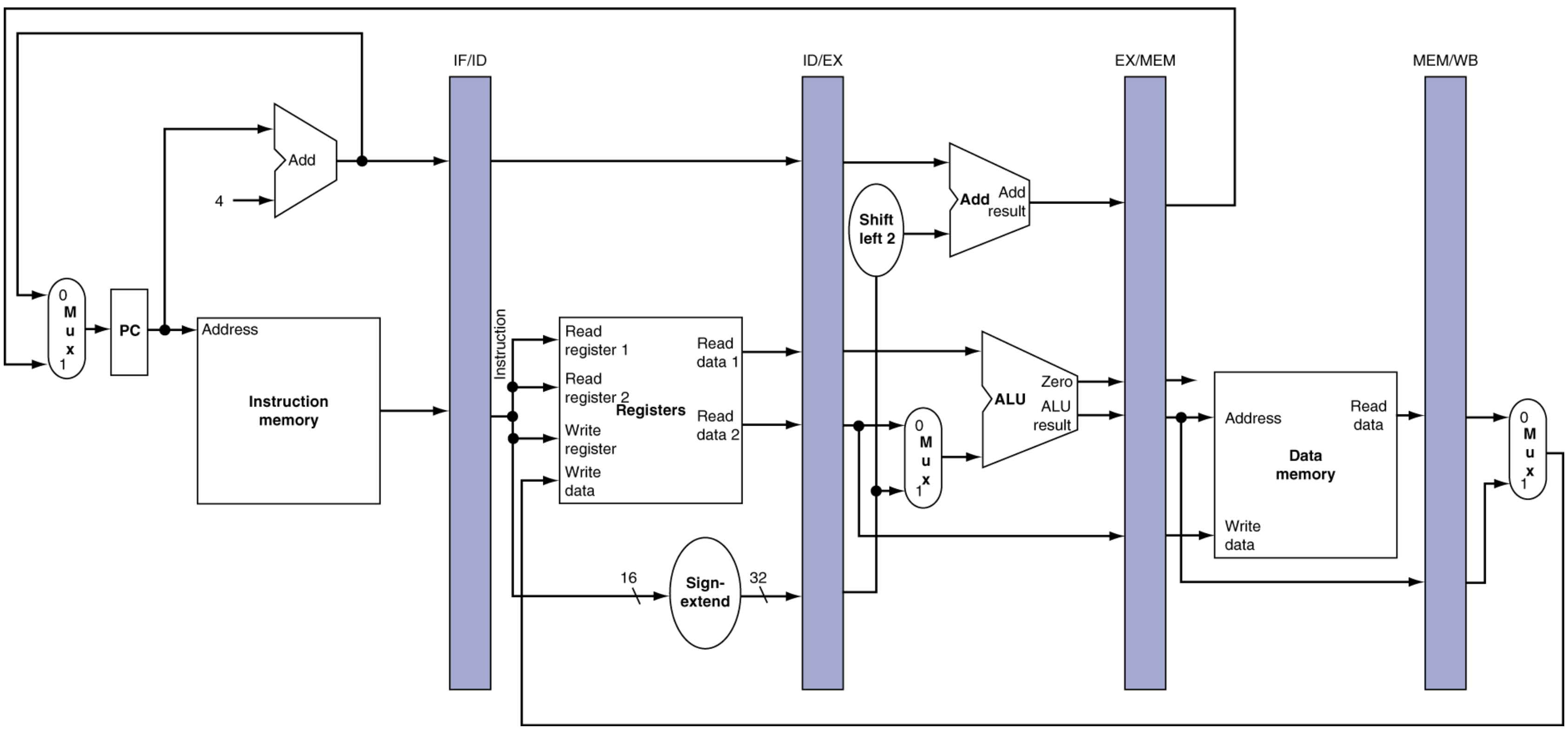
    $$= \frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Pipelining and ISA Design

- 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
  – Memory operations occur only in loads and stores
    - can use the execute stage to calculate memory addresses
  – Each instruction writes at most one result
    - and does it in the last few pipeline stages (MEM or WB)
  – Operands must be aligned in memory
    - a single data transfer takes only one data memory access

Pipeline Registers

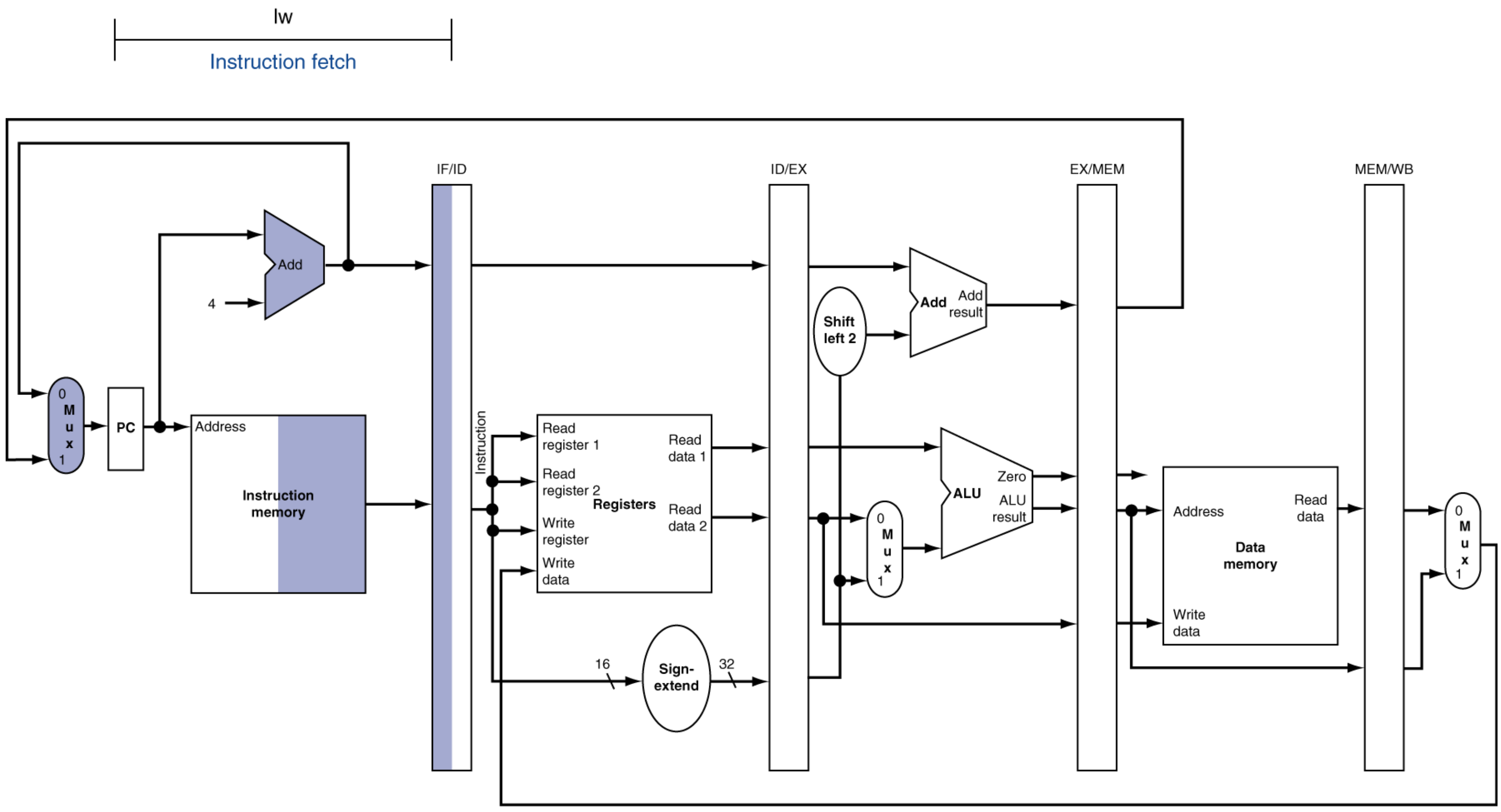
- Need registers between stages
  - To hold information produced in previous cycle



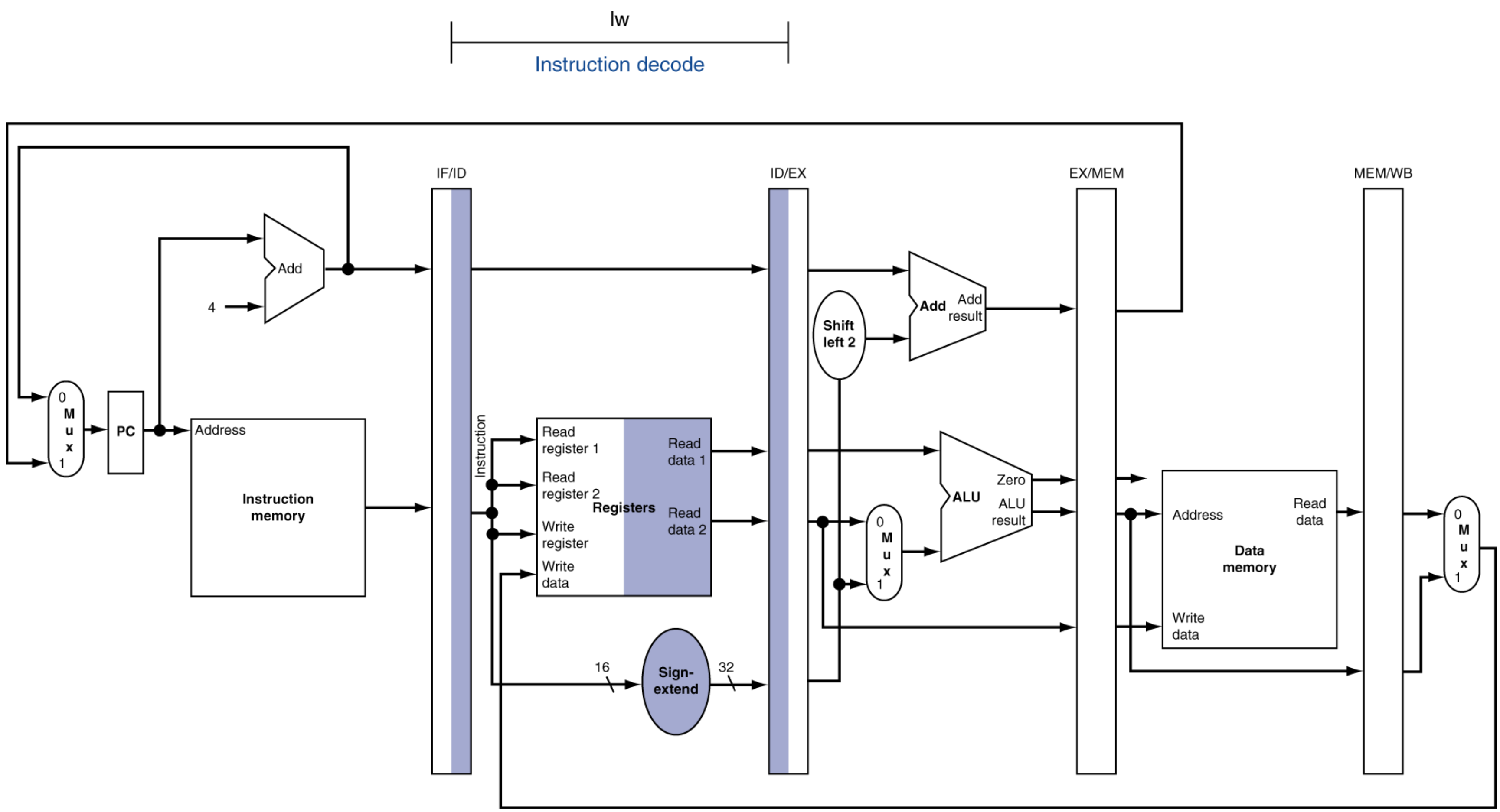
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - "Single-clock-cycle" pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. "multi-clock-cycle" diagram
    - Graph of operation over time
- We'll look at "single-clock-cycle" diagrams for load & store

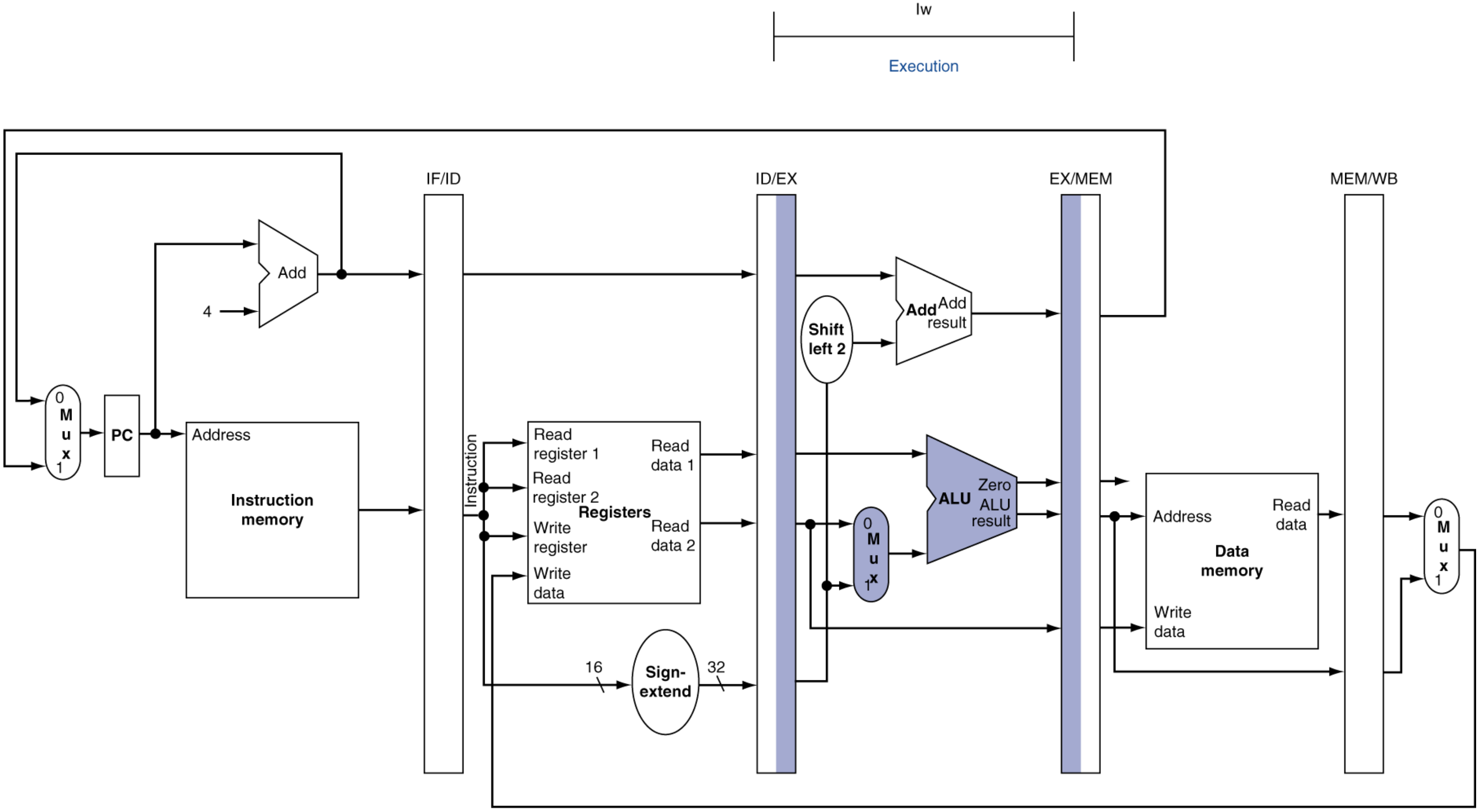
IF for Load, Store, …



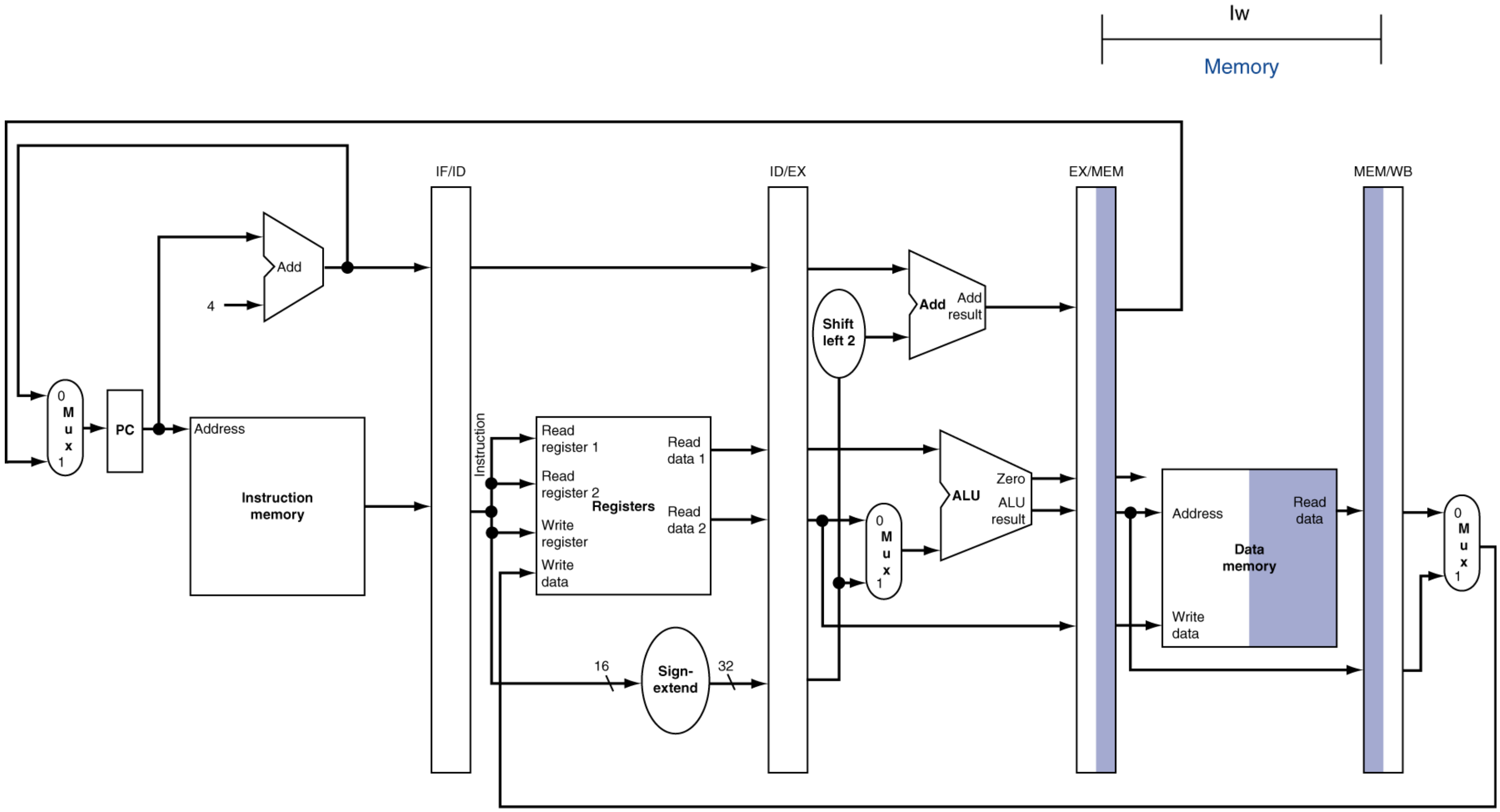
ID for Load, Store, …



EX for Load

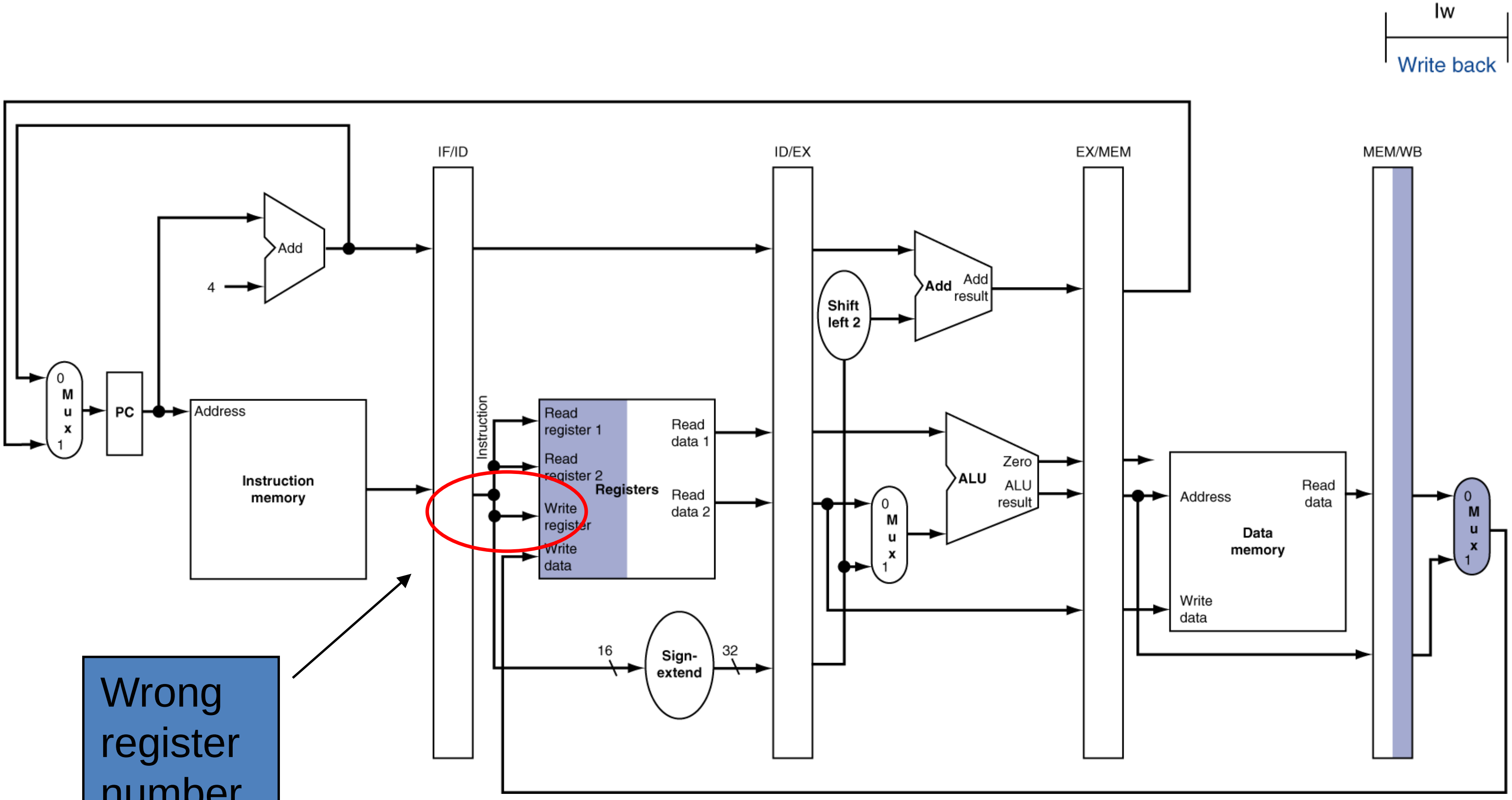

lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Write data
Read data

MEM for Load



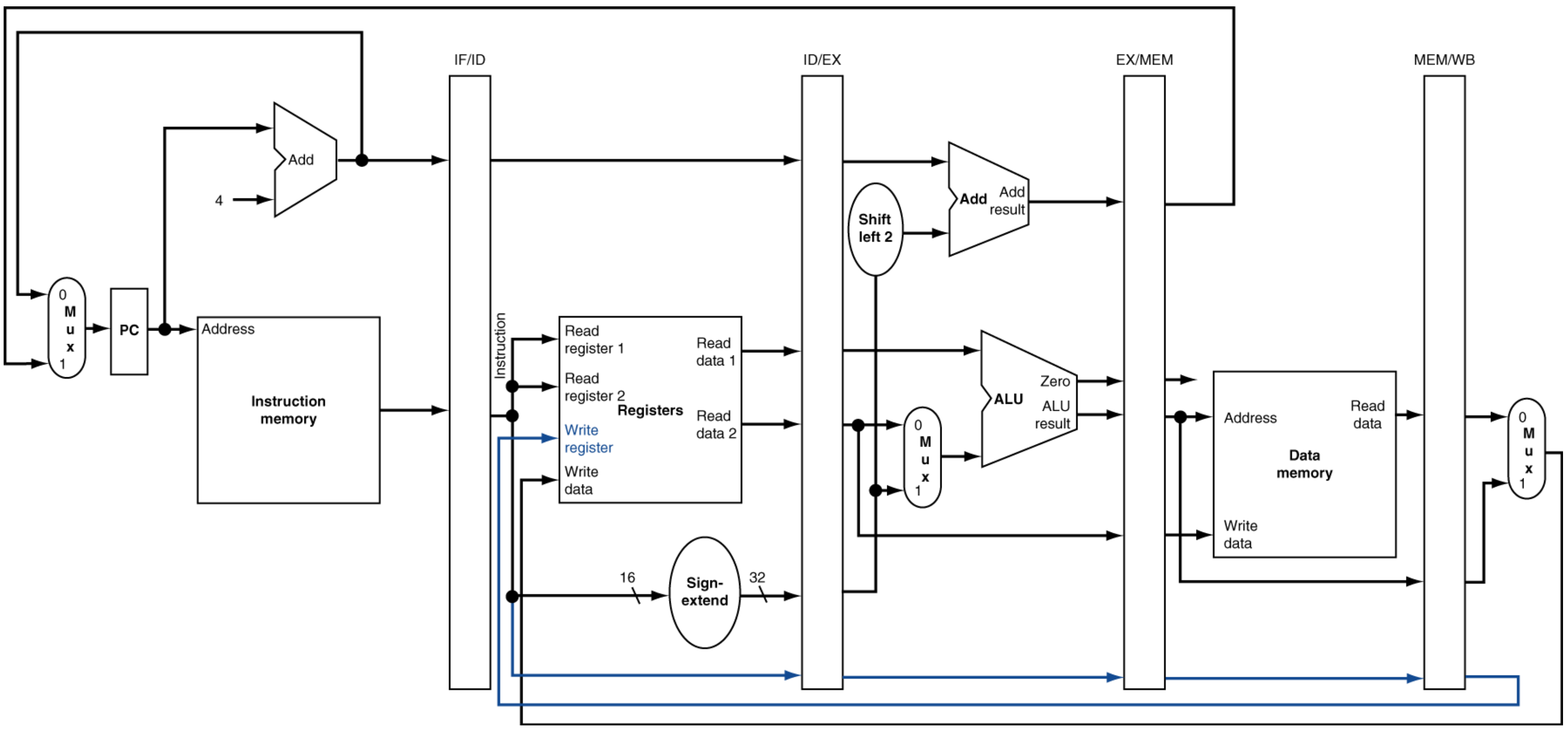
lw
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
0
M
u
x
1
Address
Data memory
Write data
Read data
0
M
u
x
1

WB for Load

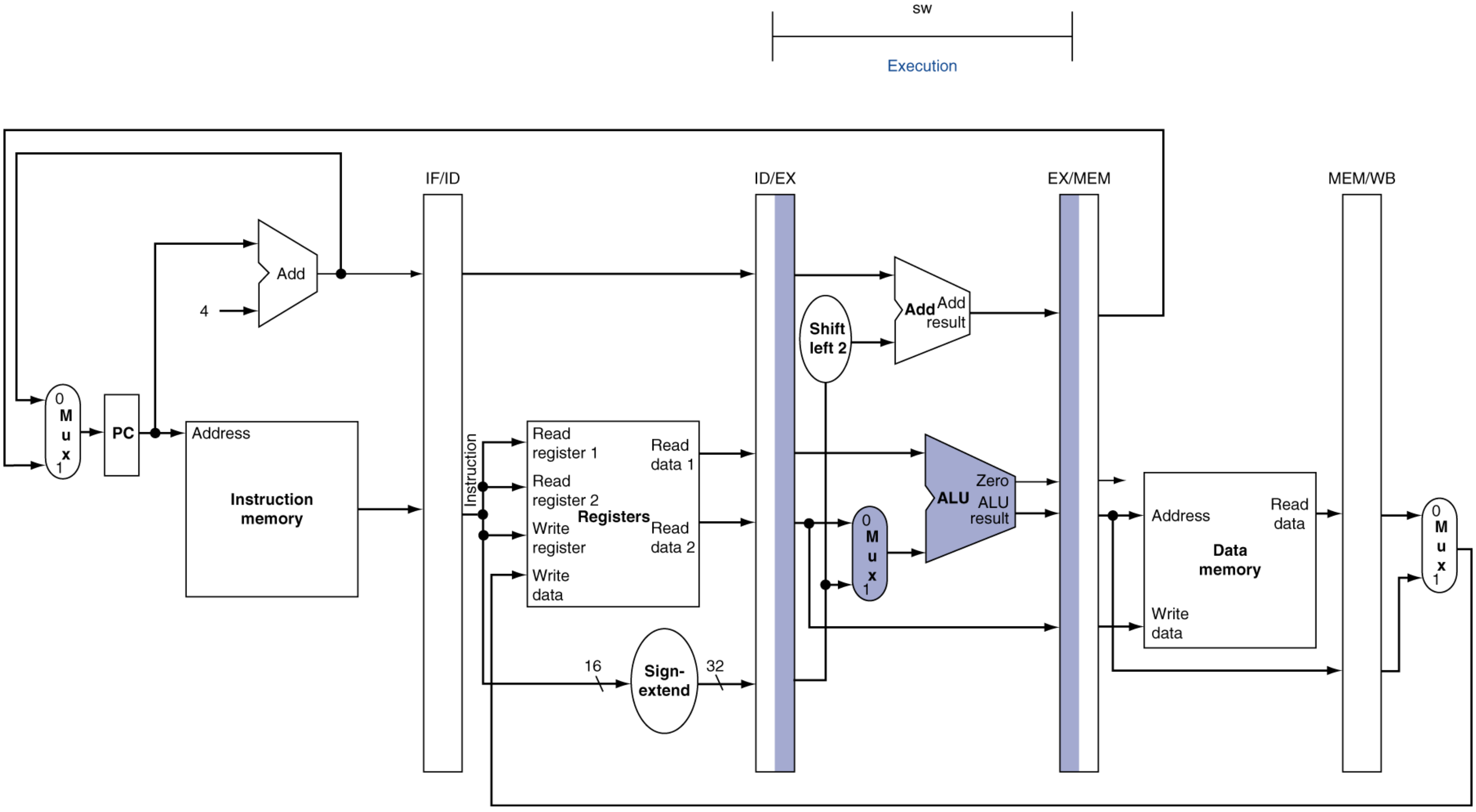


Wrong register number

Corrected Datapath for Load

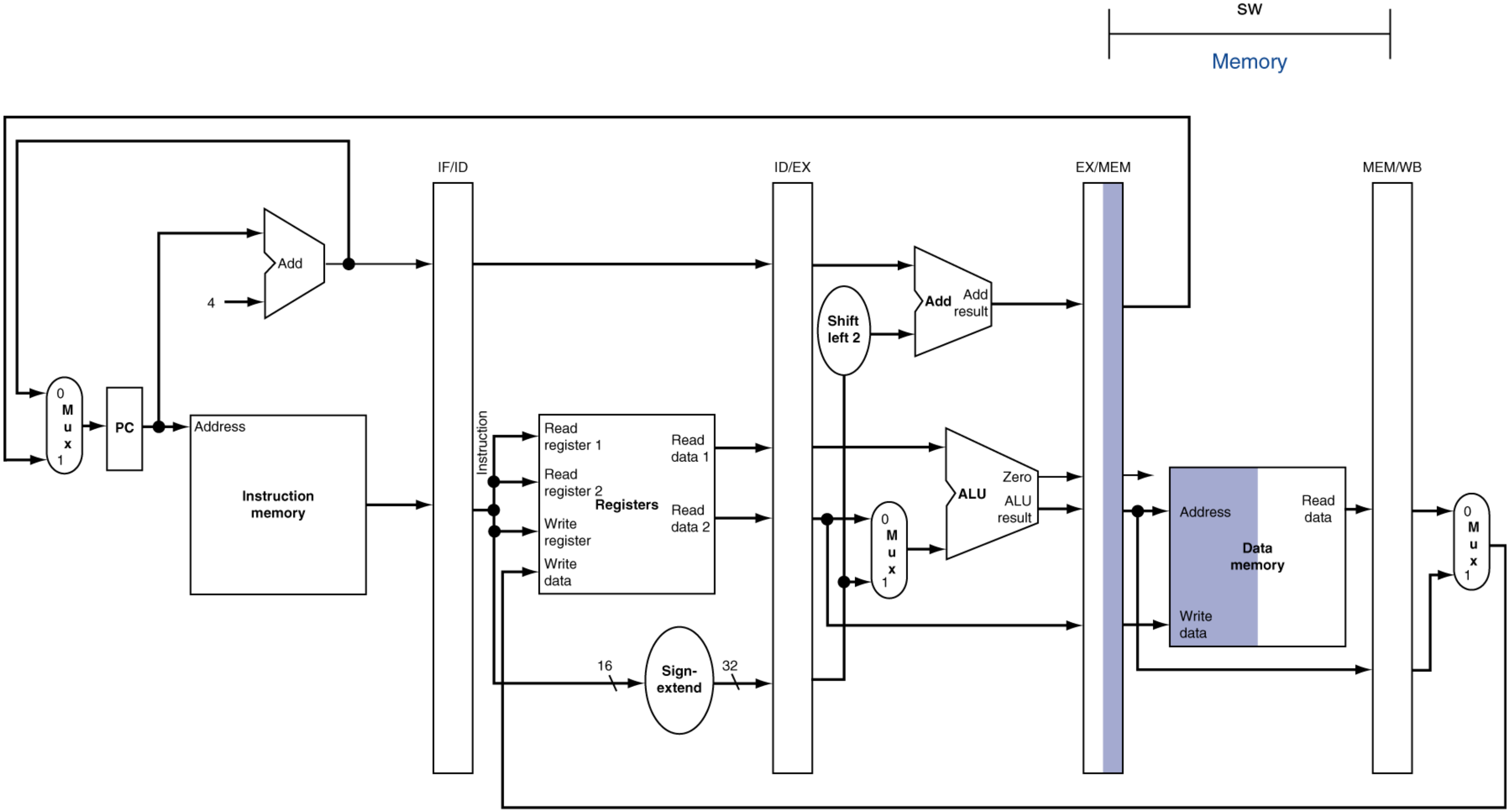


EX for Store

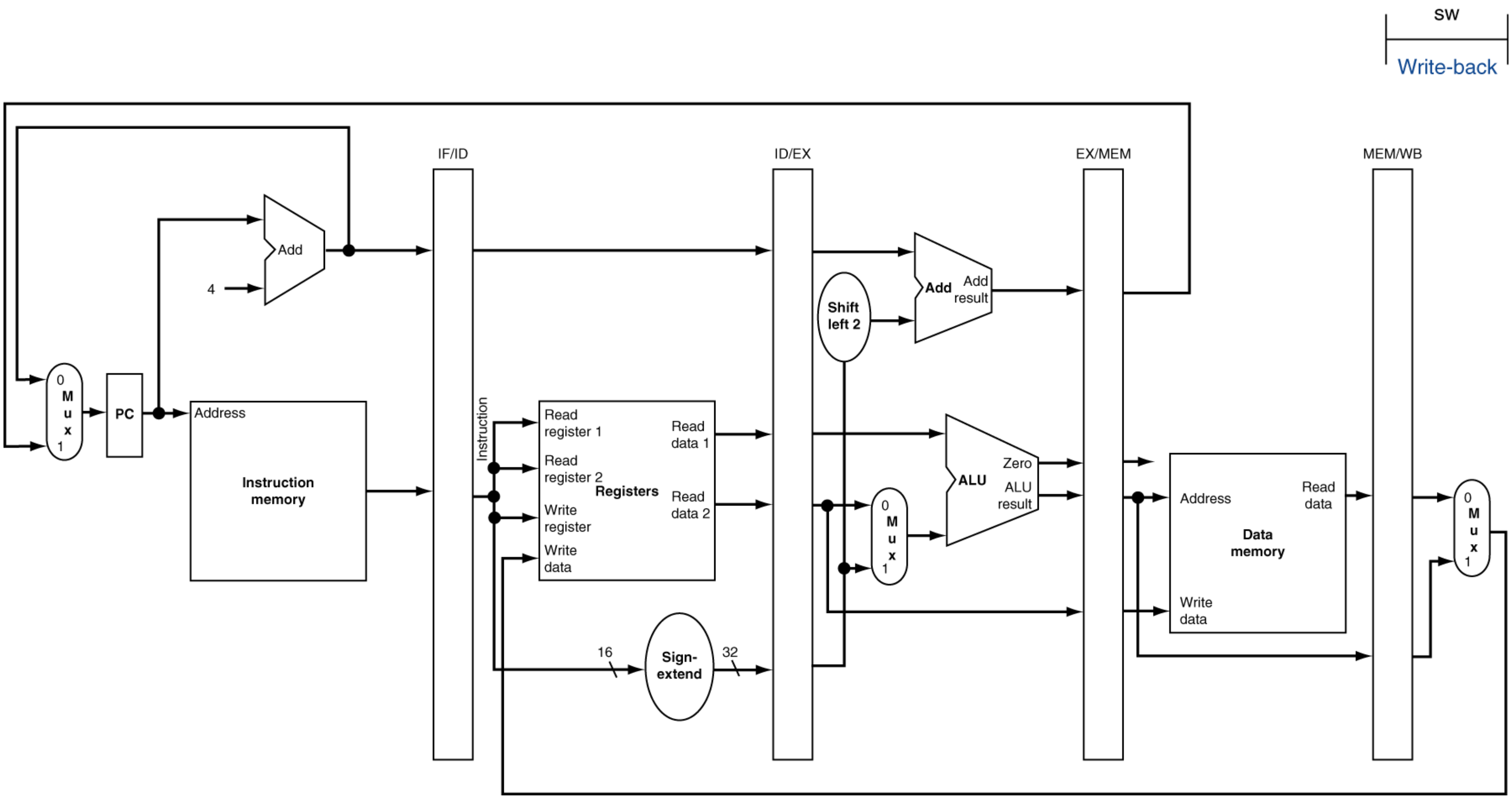


sw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
Zero
ALU
ALU result
Address
Read data
Data memory
Write data

MEM for Store



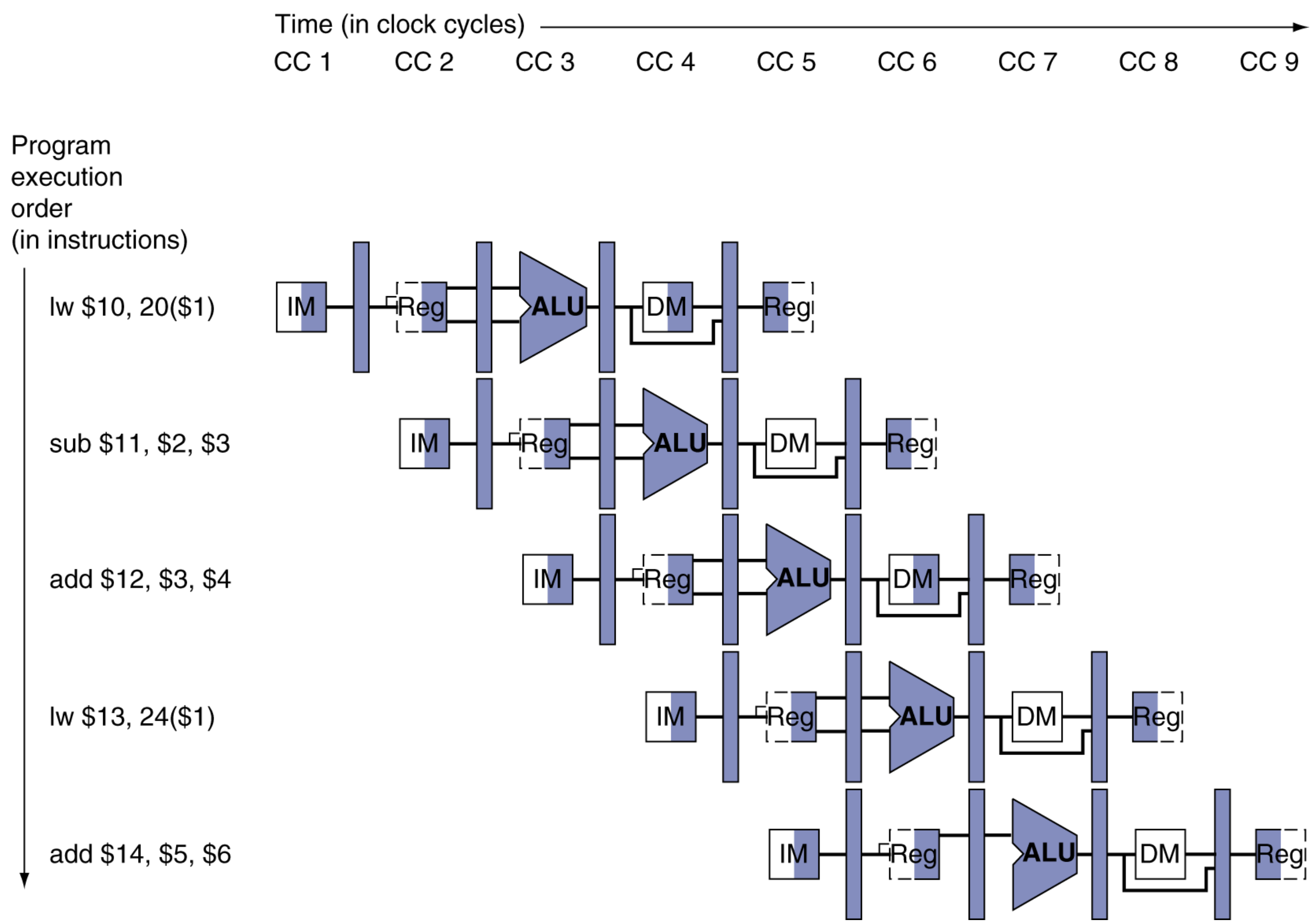
WB for Store



SW
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data

Multi-Cycle Pipeline Diagram

Form showing resource usage



Multi-Cycle Pipeline Diagram

Traditional form

Time (in clock cycles)

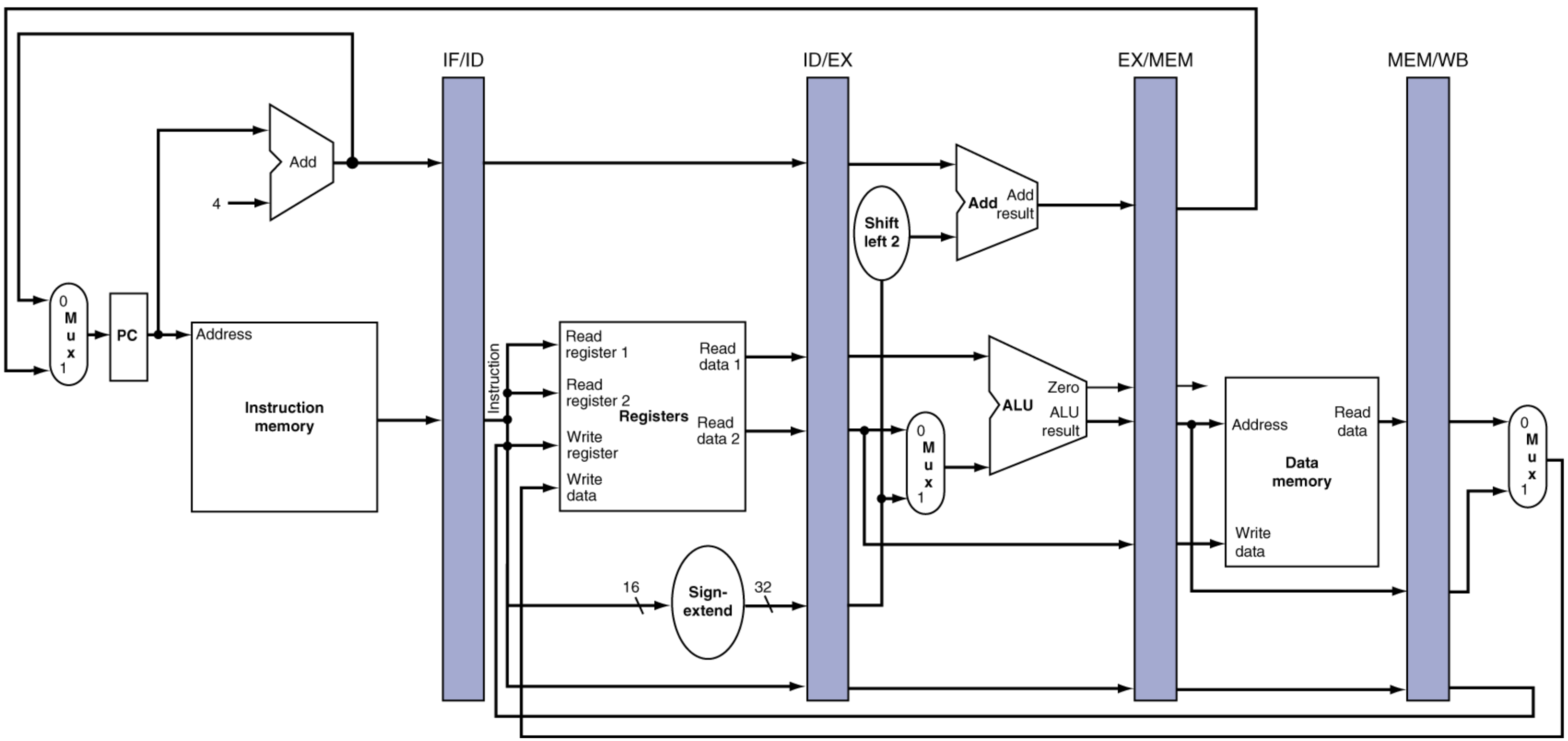
Program execution order (in instructions)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
lw $10, 20($1)	Instruction fetch	Instruction decode	Execution	Data access	Write back				
sub $11, $2, $3		Instruction fetch	Instruction decode	Execution	Data access	Write back			
add $12, $3, $4			Instruction fetch	Instruction decode	Execution	Data access	Write back		
lw $13, 24($1)				Instruction fetch	Instruction decode	Execution	Data access	Write back	
add $14, $5, $6					Instruction fetch	Instruction decode	Execution	Data access	Write back

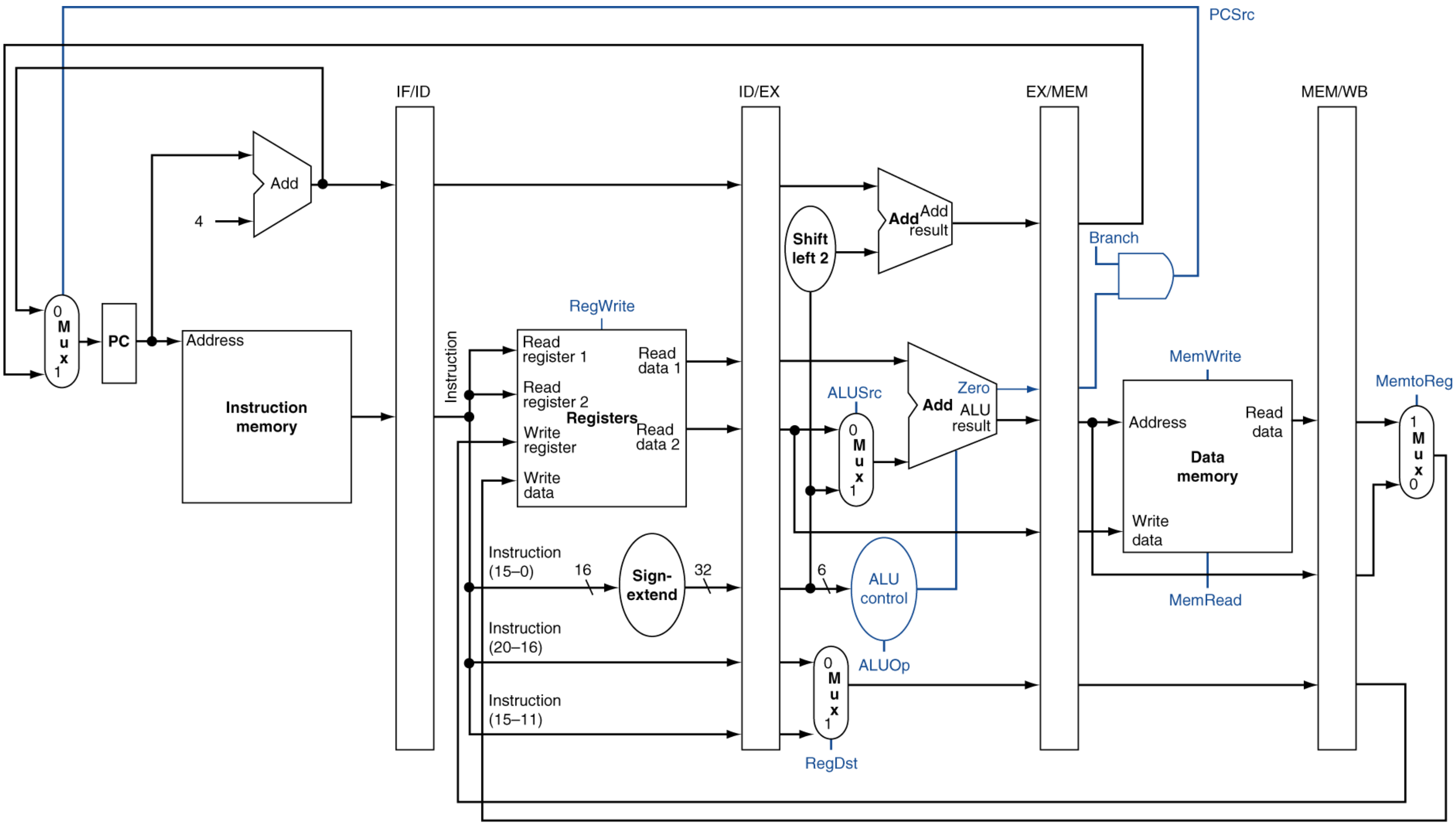
Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle

add $14, $5, $6	lw $13, 24 ($1)	add $12, $3, $4	sub $11, $2, $3	lw $10, 20($1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back

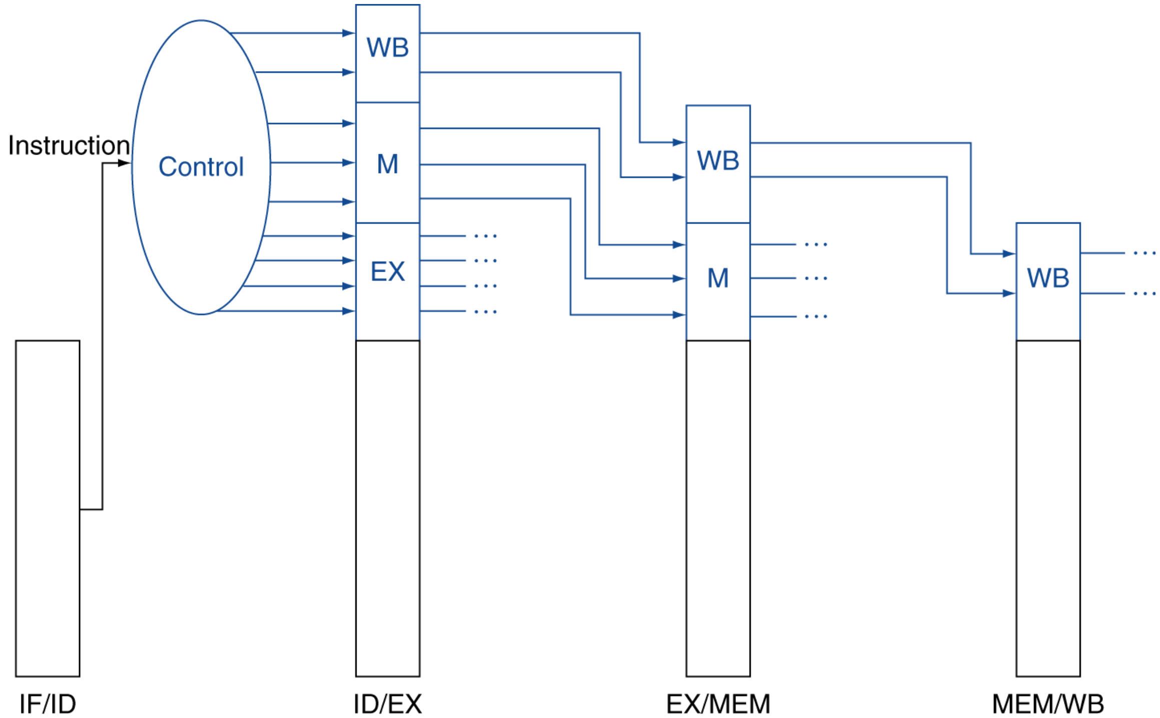


Pipelined Control (Simplified)

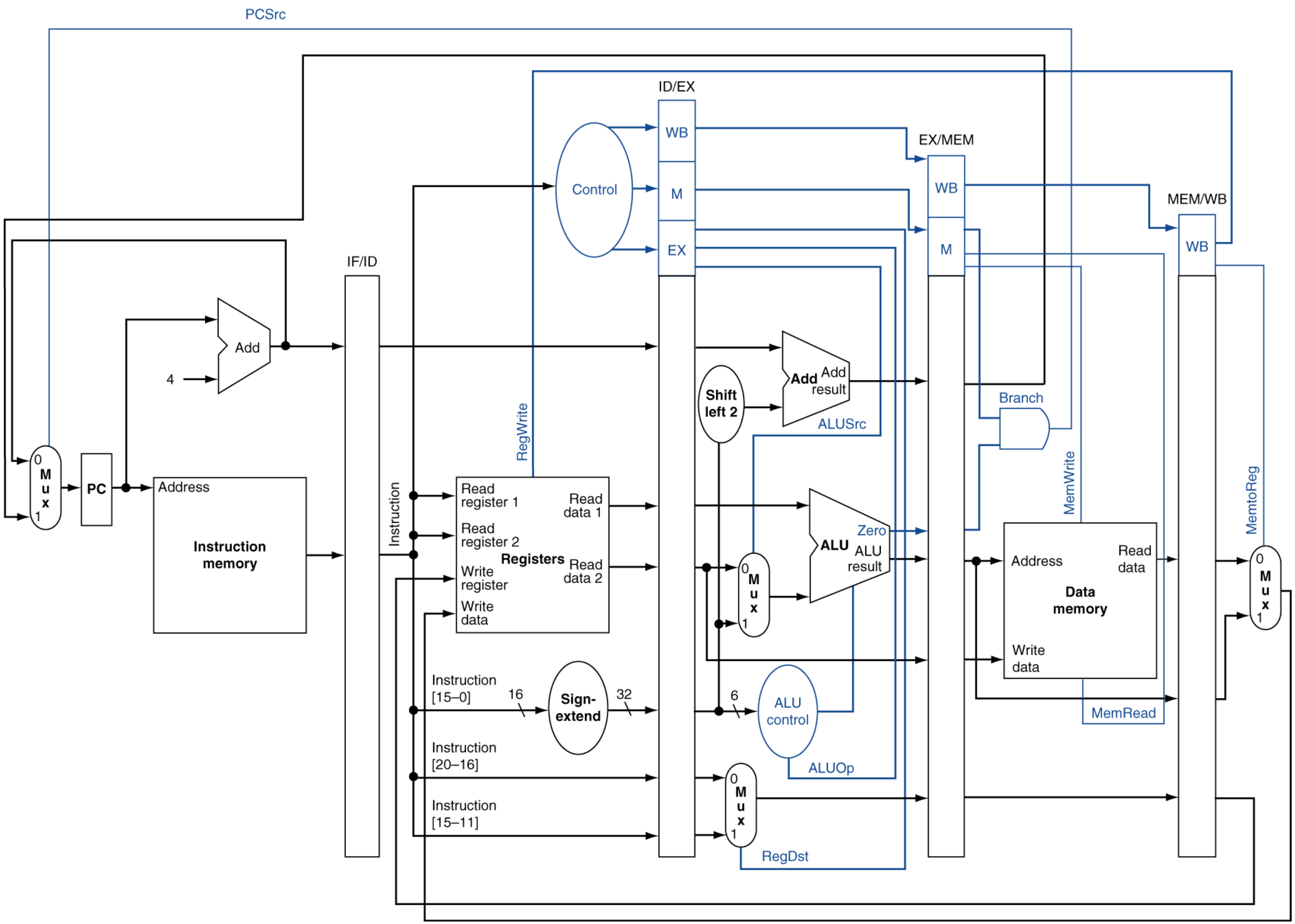


Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation



Pipelined Control



Can Pipelining Get Us Into Trouble?

- Pipeline Hazards - Situations that prevent starting the next instruction in the next cycle
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: deciding on control action depends on previous instruction
    - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
  - control hazards: attempt to make a decision about program control flow before the condition has been evaluated by a previous instruction
    - branch and jump instructions, exceptions
- Can usually resolve hazards by waiting (stall)
  - pipeline control must detect the hazard
  - and take action to resolve hazards

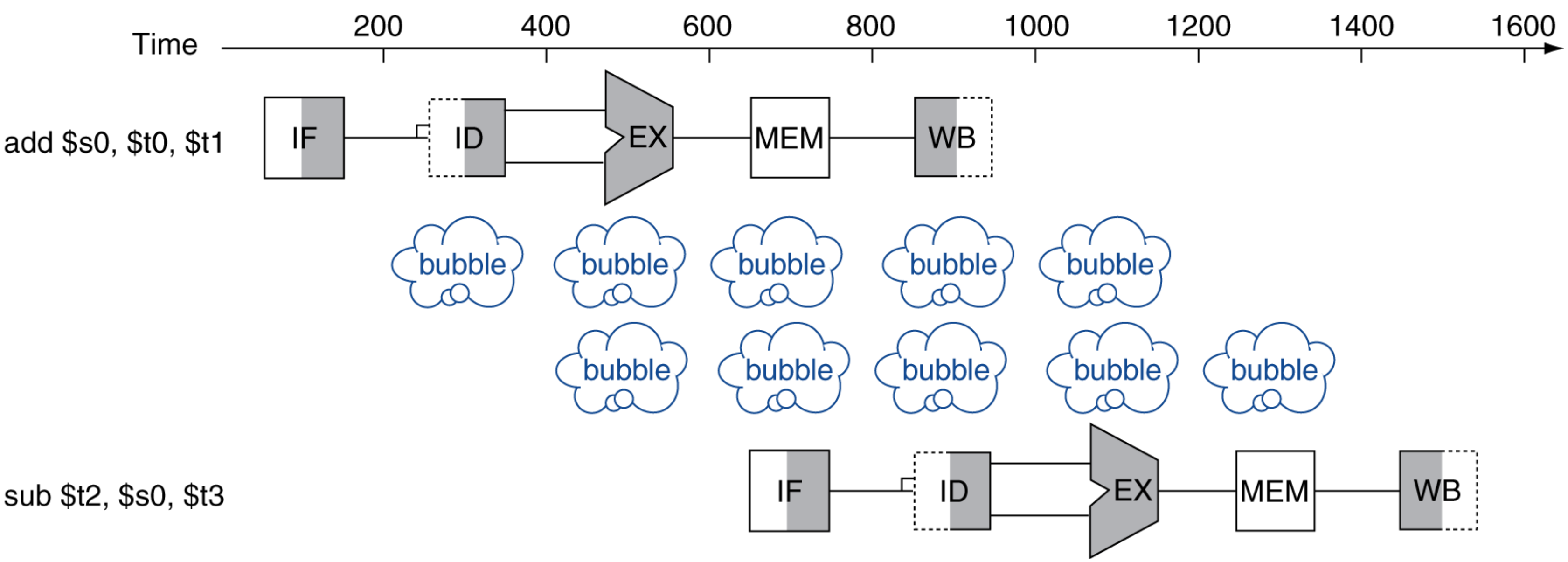
Structural Hazards

- 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

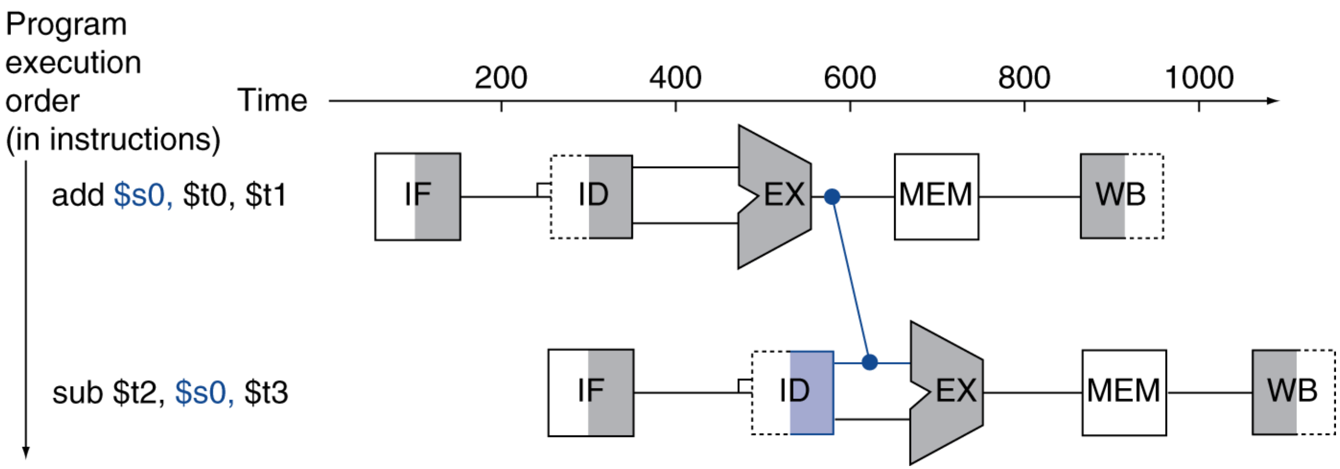
An instruction depends on completion of data access by a previous instruction

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



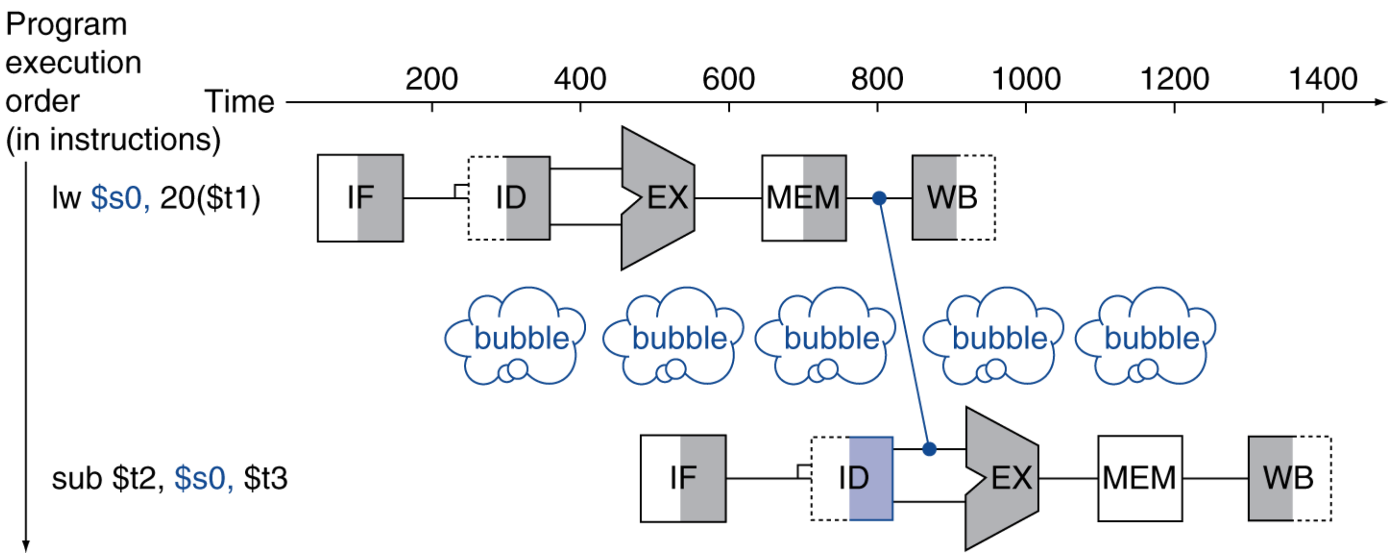
Forwarding (aka Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



Load-Use Data Hazard

- Can’t always avoid stalls by forwarding
  - If value not computed when needed
  - Can’t forward backward in time!

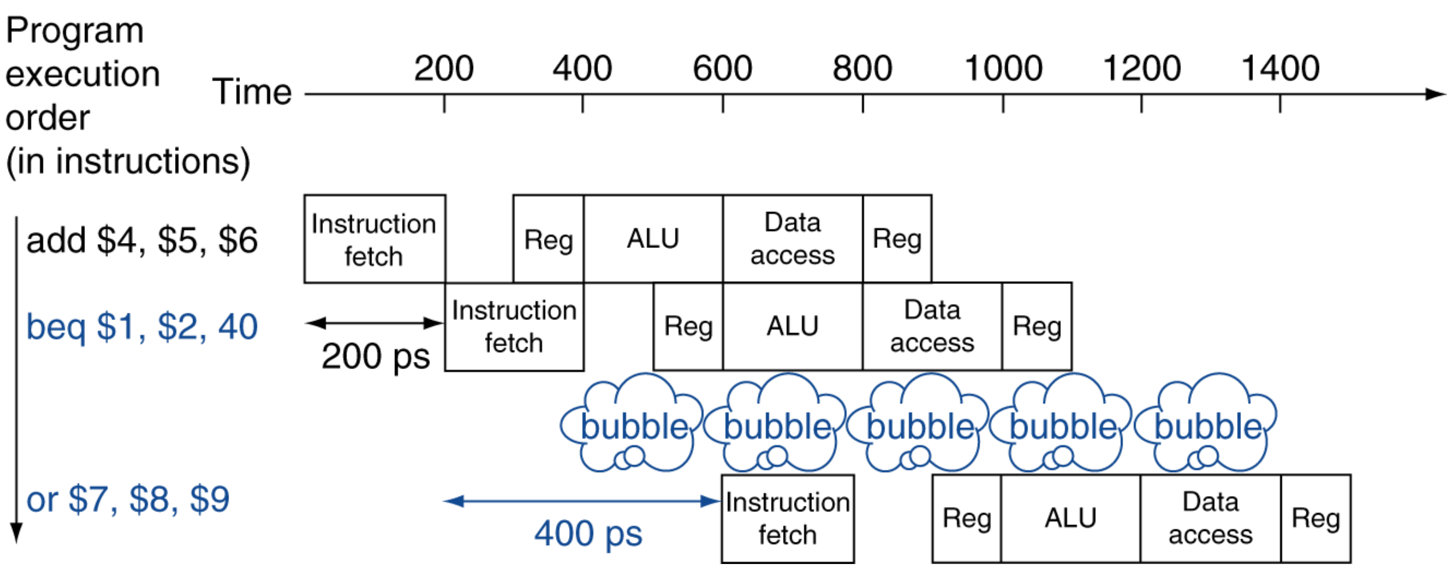


Control Hazards

- 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

Stall on Branch

- Wait until branch outcome determined before fetching next instruction



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

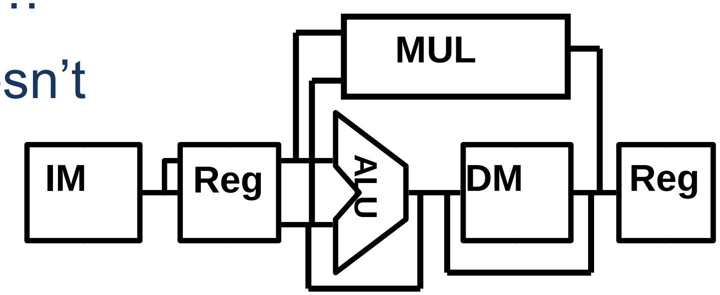
Pipeline in a nutshell

The BIG Picture

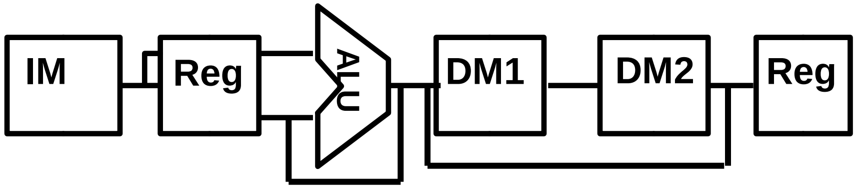
- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

Other Pipeline Structures Are Possible

- What about the (slow) multiply operation?
  - make the clock twice as slow or …
  - let it take two cycles (since it doesn't use the DM stage)

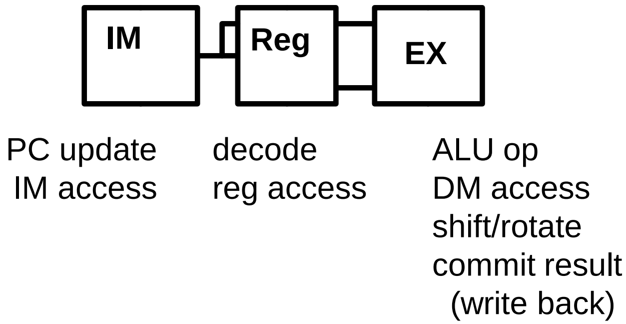


- What if the data memory access is twice as slow as the instruction memory?
  - make the clock twice as slow or …
  - let data memory access take two cycles (and keep the same clock rate)

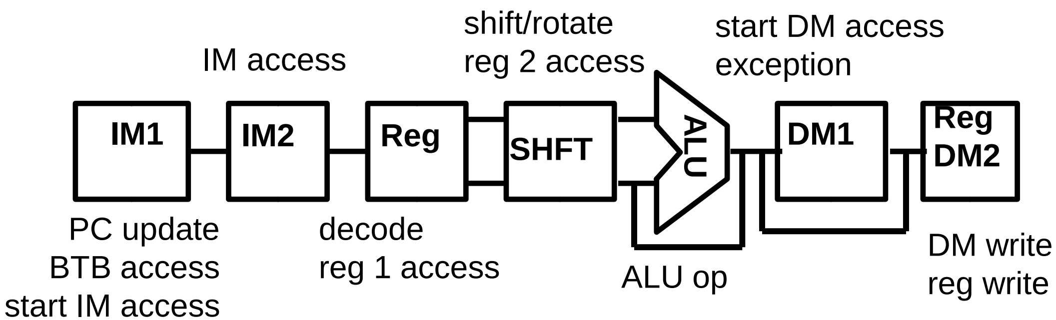


Other Sample Pipeline Alternatives

- ARM7



- XScale



Conclusion

- All modern day processors use pipelining
- Pipelining doesn’t help latency of single instruction, it helps throughput of entire workload
- Potential speedup: a CPI of 1 and faster CC
- Pipeline rate limited by slowest pipeline stage
  - Unbalanced pipe stages makes for inefficiencies
  - The time to “fill” pipeline and time to “drain” it can impact speedup for deep pipelines and short code runs
- Must detect and resolve hazards
  - Stalling negatively affects CPI (makes CPI higher than the ideal of 1)

Next Classes

- Reducing pipeline data and branch hazards
  – Forwarding
  – Delayed Branch & Branch Prediction

- Exceptions and Interrupts

MIPS – Pipelining

Computer Organization

Sunday, 09 October 2022

Many slides adapted from:
Computer Organization and Design,
Patterson & Hennessy
5th Edition, © 2014, MK
and from Prof. Mary Jane Irwin, PSU

TÉCNICO LISBOA