COMPUTER ARCHITECTURE

Chapter 4: Microarchitecture

Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two MIPS implementations
  - A simplified version: CPI = 1
  - A more realistic pipelined version: CPI $\approx$ 1
- Simple subset, shows most aspects
  - Memory reference: lw, sw
  - Arithmetic/logical: add, sub, and, or, slt
  - Control transfer: beq, j

Instruction execution

1. PC → **instruction memory (cache)**, fetch instruction
2. Register numbers → **registers file**, read registers
3. Depending on instruction class
    3.1. Use ALU to calculate
        - Arithmetic result → done
        - Memory address for load/store → 3.2
        - Branch target address → 3.3
    3.2. Access data memory for load/store
    3.3. PC ← target address or PC + 4

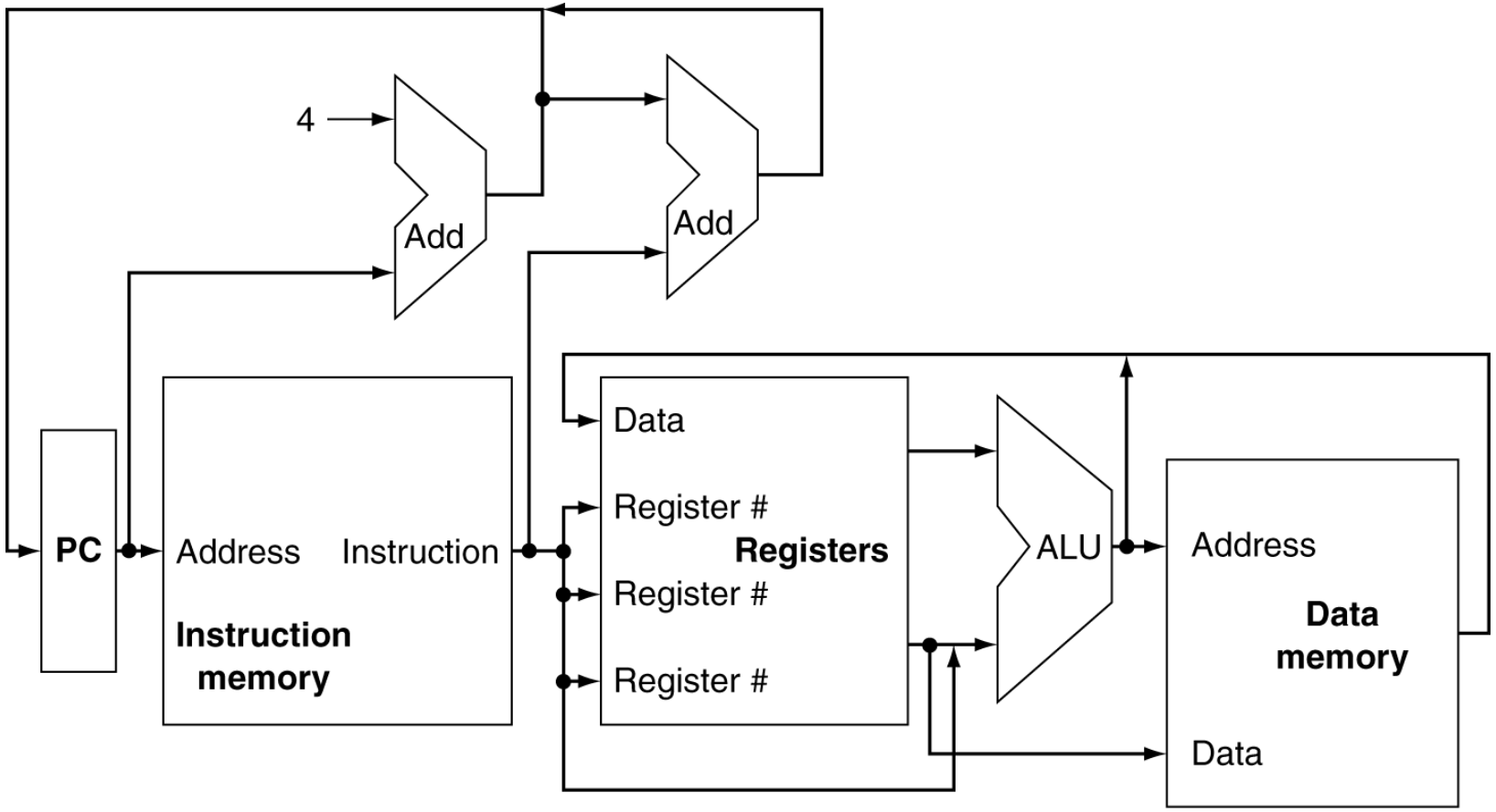
Execution stages

1. **Instruction fetch** (**IF**): PC → instruction address
2. **Instruction decode** (**ID**): register operands → register file
3. **Execute** (**EXE**):
   - Load/store: compute a memory address
   - Arithmetic/logical: compute an arithmetic/logical result
4. **Memory access** (**MEM**):
   - Load: read data memory
   - Store: write data memory
5. **Write back** (**WB**):
   - Store a result of register file

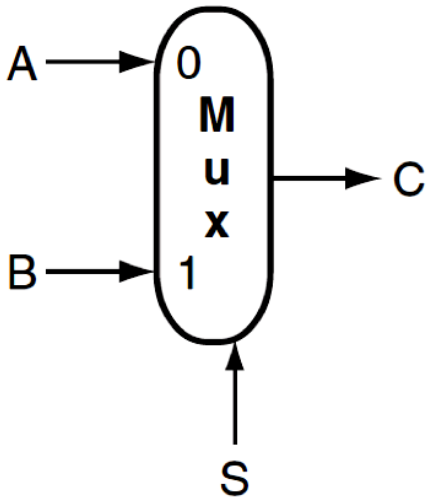
Datapath - controller

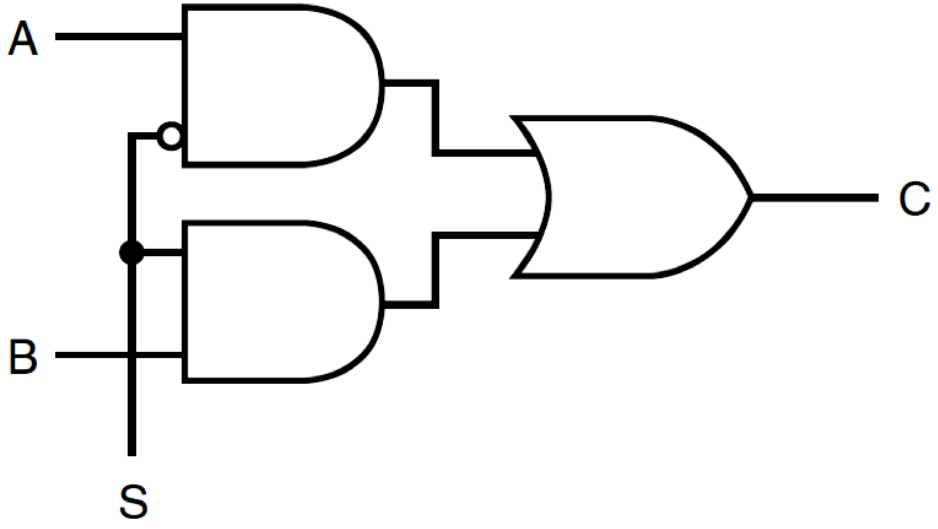
- **Datapath**: contains information that is operated on by a functional unit
  - Instruction memory: contain instructions (**IF**)
  - Registers file: 32 32-bit registers (**ID** & **WB**)
  - ALU: calculate arithmetic and logical operations (**EXE**)
  - Data memory: contain data (**MEM**)
- **Control signals**: used for multiplexer selection or for directing the operation of a functional unit
  - Control unit
  - Multiplexer

Datapath overview



Multiplexer - MUX

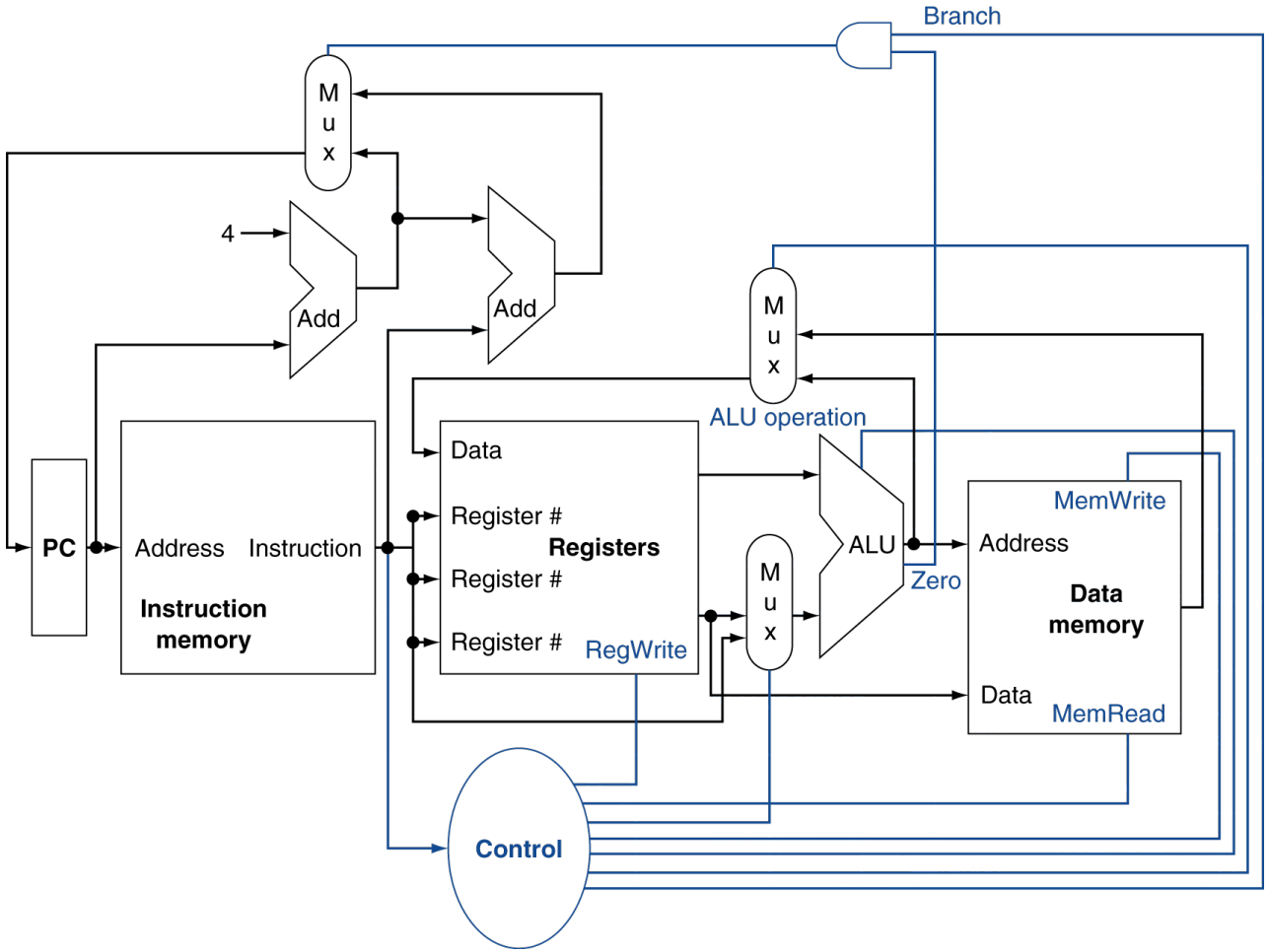




S = 0 → C = A

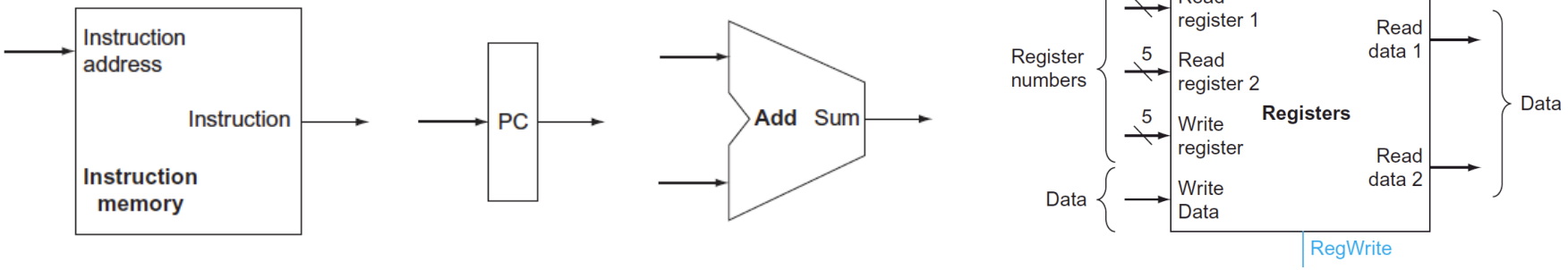
S = 1 → C = B

Controller overview



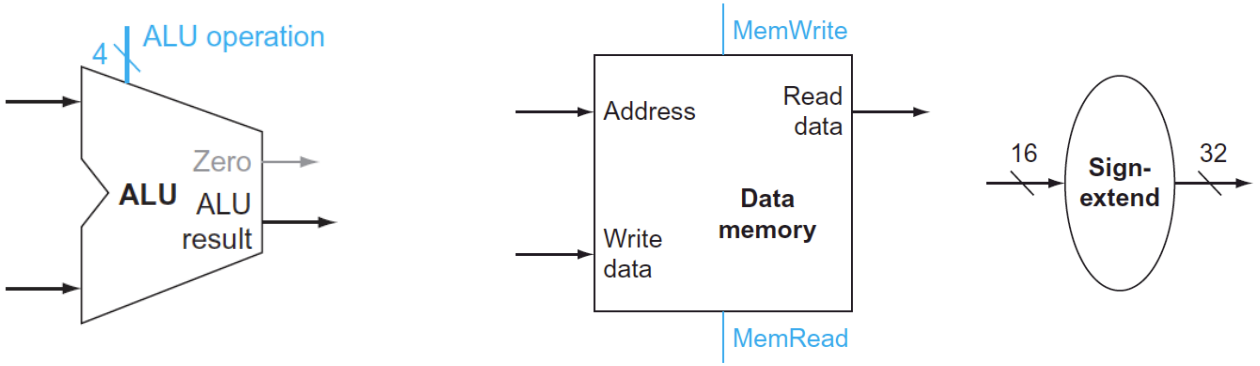
Building datapath

- Hardware components:



IF

ID & WB

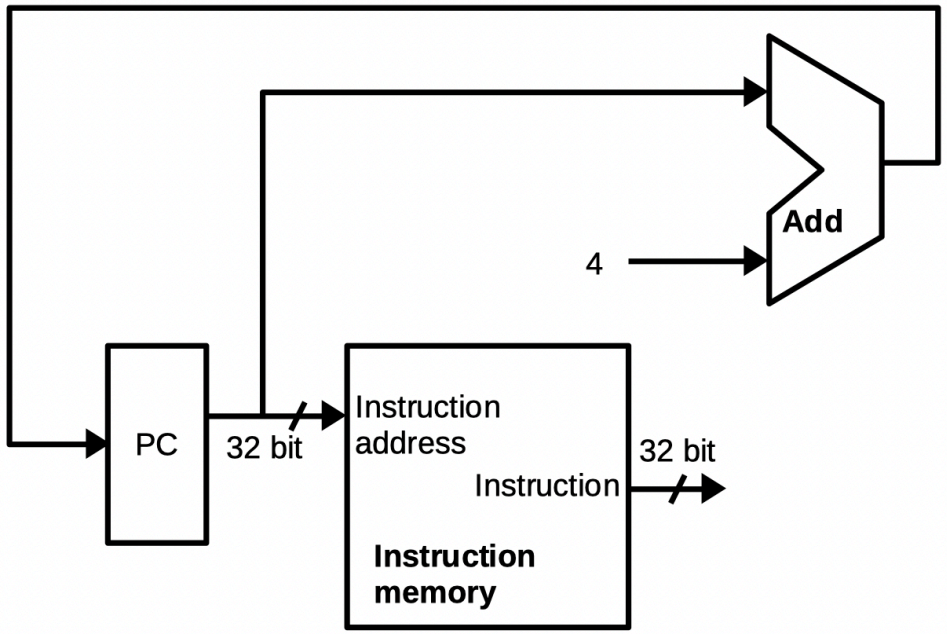


EXE

MEM

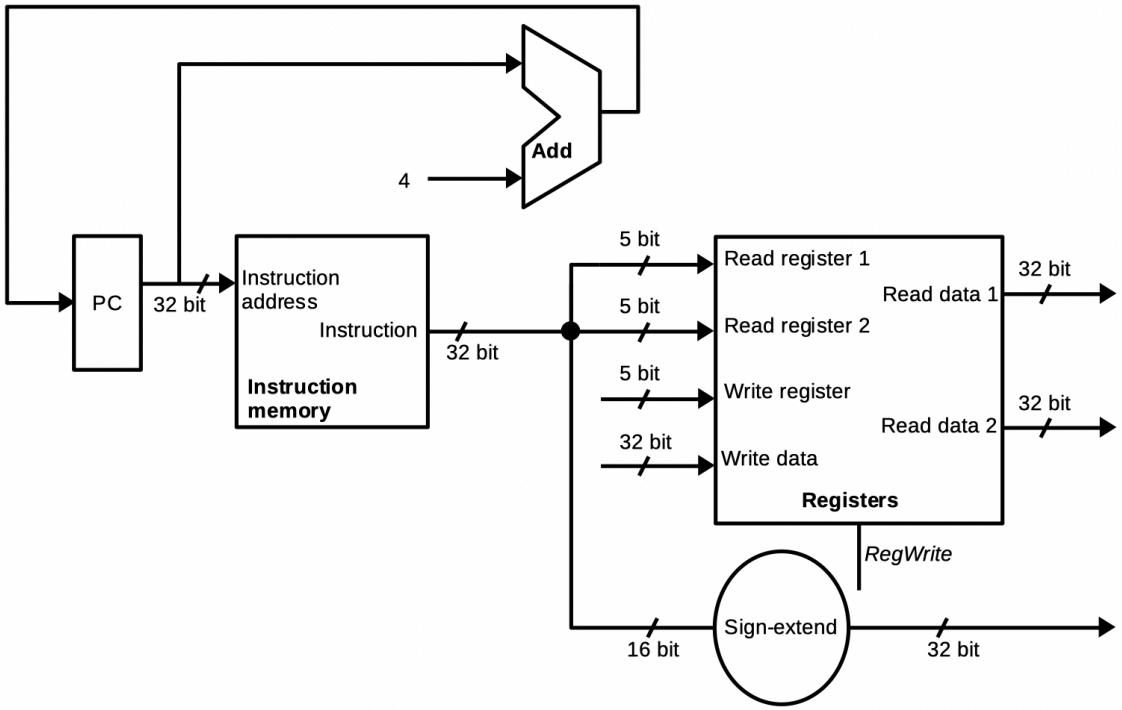
Instruction fetch

- **Main operations**: fetching the next instruction from Instruction memory
  - **PC** → instruction address
  - **Instruction memory** → instruction (32 bits)
  - **PC** ← **PC** + 4 (using the **Add** component)
- **Results**:
  - 32 bits machine instruction (Instruction)
  - Address of the next instruction in PC



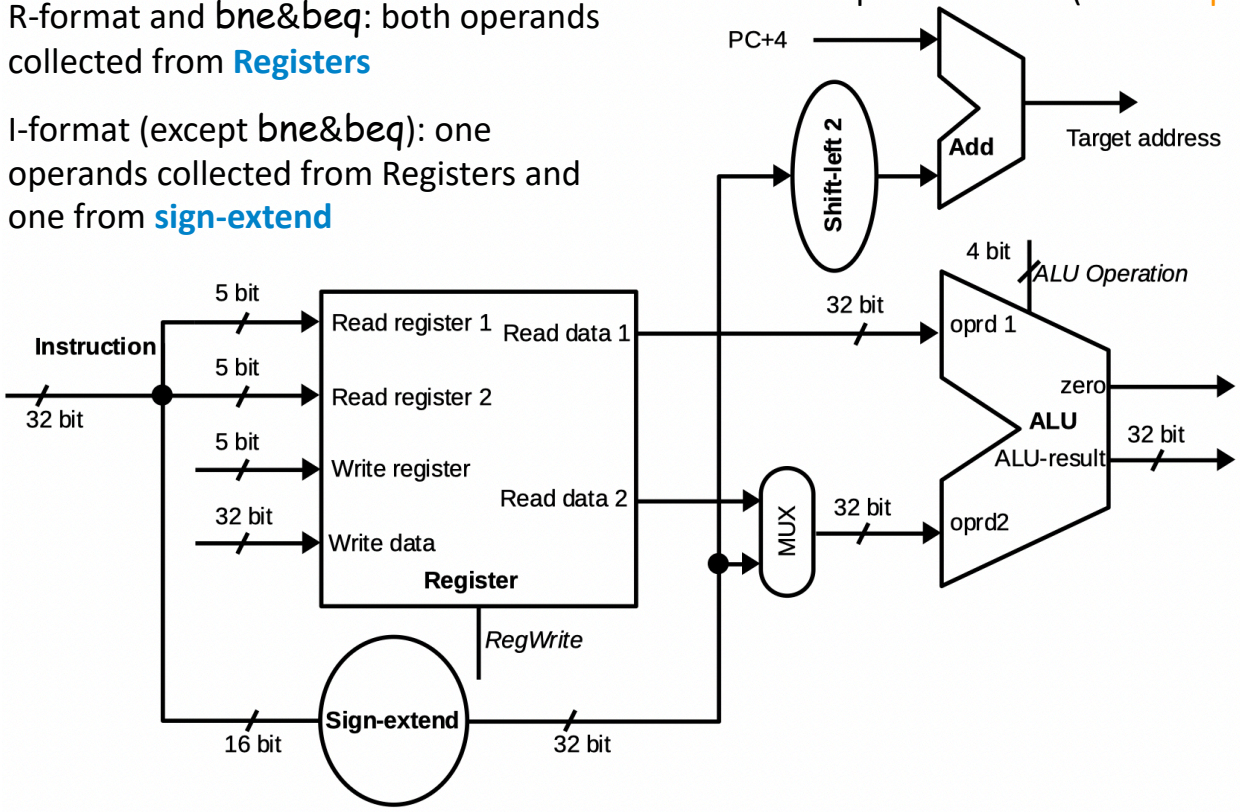
Instruction decode

- **Main operations**: Extract machine instructions
  - R-format, branch, and store: rs, rt → Registers → values
  - I-format: rs → Registers → values & immediate → sign-extend
- **Results**:
  - Values (32-bit) for the next stage



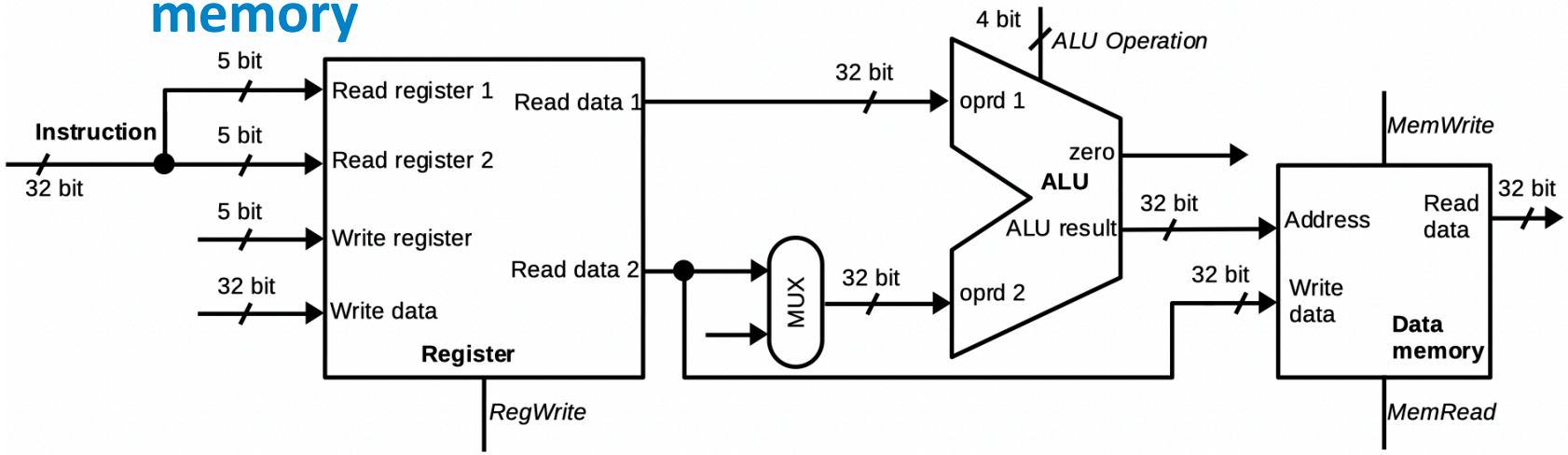
Instruction execution

- **Main operation**
  - Calculate the arithmetic operations/ address of memory/register comparison
    - R-format and bne&beq: both operands collected from **Registers**
    - I-format (except bne&beq): one operands collected from Registers and one from **sign-extend**
- **Results**:
  - Arithmetic result/memory address (ALU-result)
  - Comparison result (zero-output)



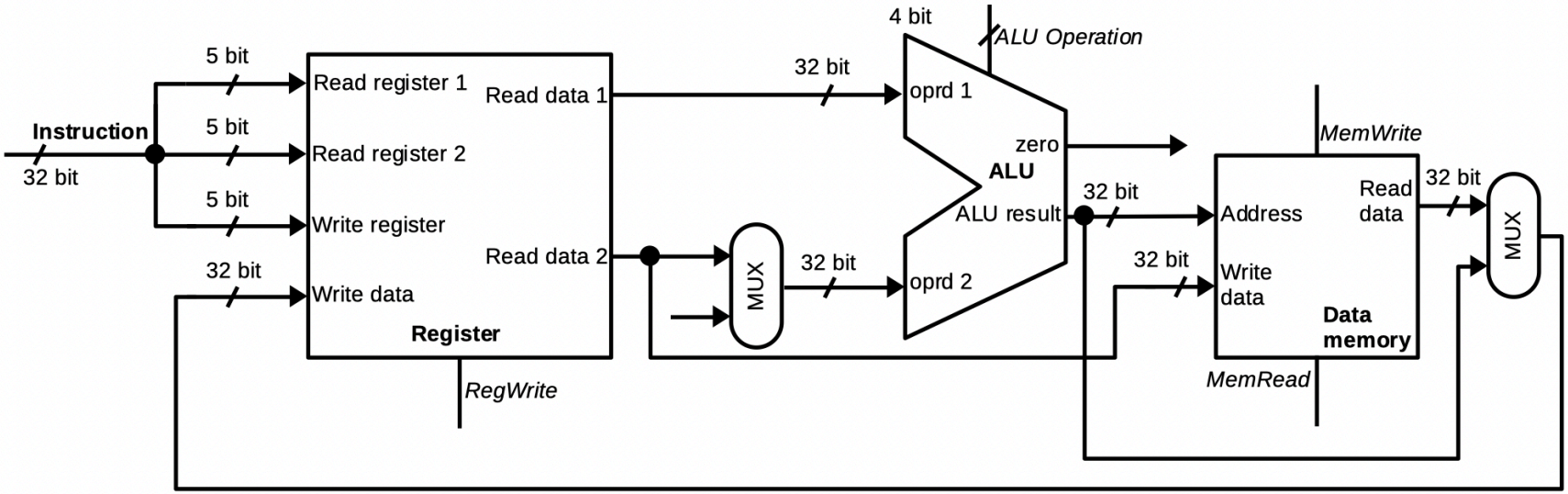
Memory access

- **Main operations**
  - Load: get data from **Data memory**
  - Store: write data from **Registers** to **Data memory**
- **Results:**
  - Load: values of **Data memory** (Read data)
  - Store: N/A

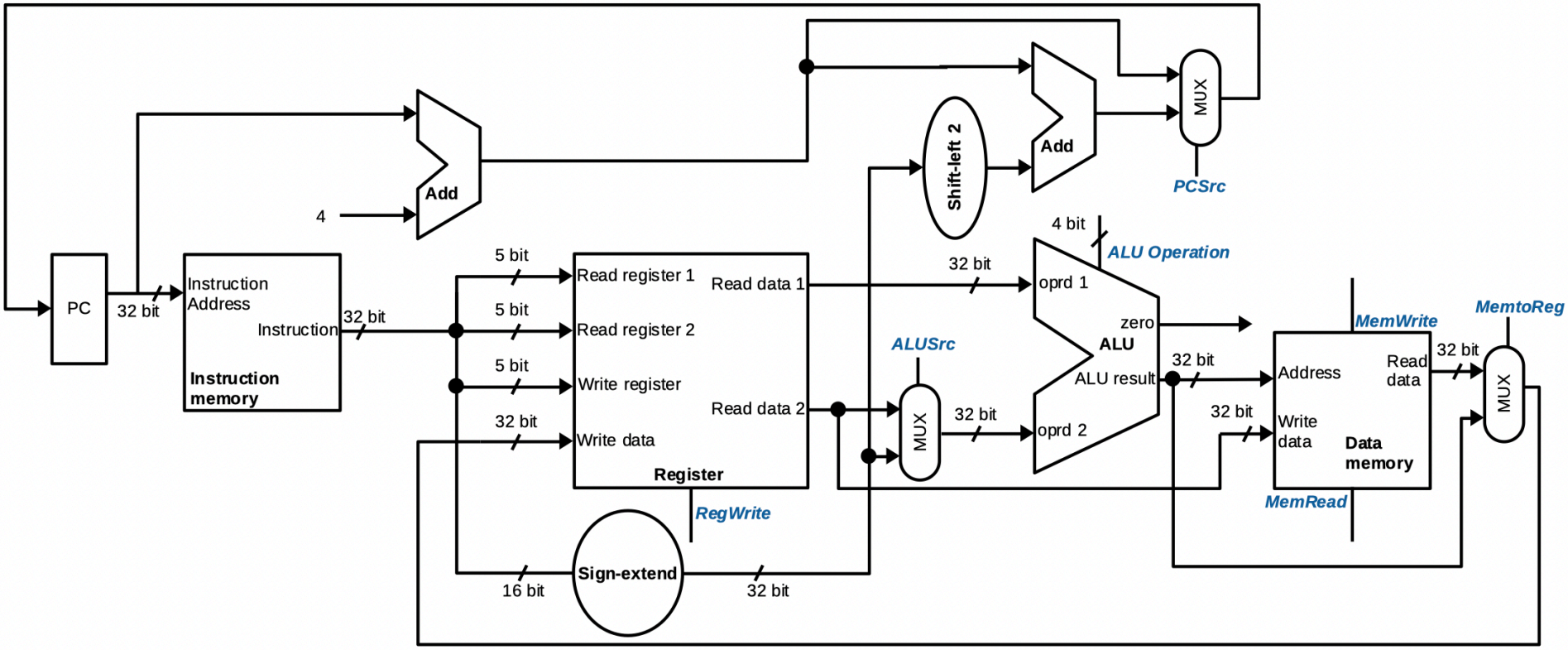


Write back

- **Main operations**:
  - Write values back to **Registers** (arithmetic/load)
- **Result**:
  - N/A



Full datapath



Example

- **Question**: Assume that the processor is executing

```
add $s0, $s1, $s2
```

  - Identify values of functional units' inputs/outputs
- **Answer**:
  - The machine code is: $000000\_10001\_10010\_10000\_00000\_100000_2$
  - Instruction memory:
    - **Instruction address** = PC (word address where we store the above instruction)
    - **Instruction** = machine code
  - Registers:
    - **Read register 1** = 10001_2 => **Read data 1** = content ($s1)
    - **Read register 2** = 10010_2 => **Read data 2** = content ($s2)
    - **Write register** = 10000_2 & **Write data** = content ($s1) + content ($s2)
  - …sign-extend: input = 10000_00000_100000

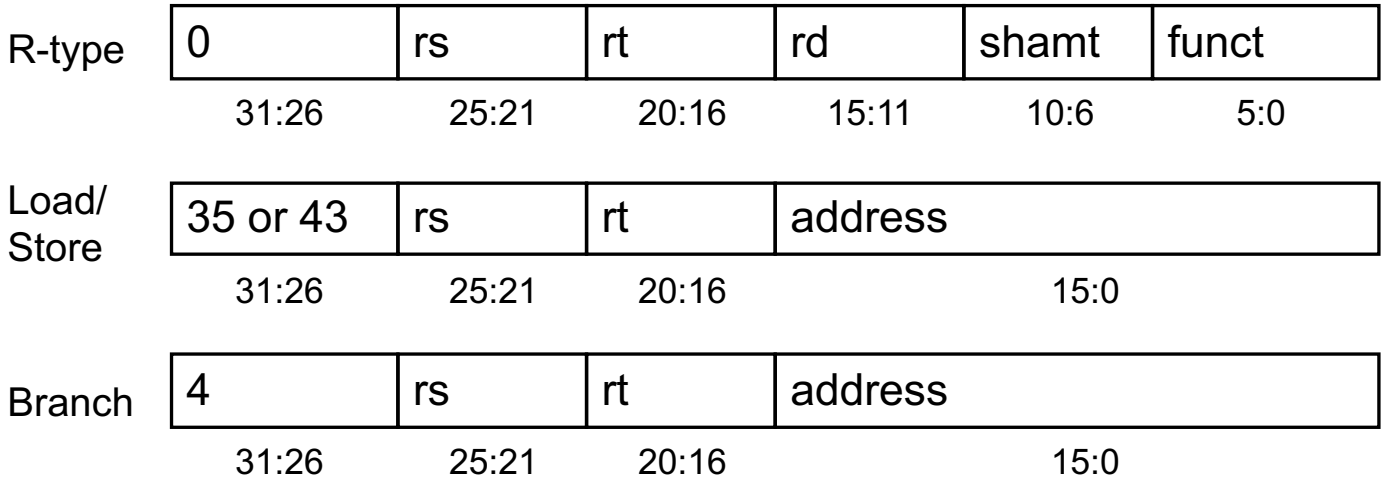
Building controller

- Extracting bits from 32-bit instructions
  - Read register 1, Read register 2, and Write registers
  - Sign-extend
  - **Control** block
- Building a **Control** block:
  - Handling multiplexers
  - Handling control signals of functional units
    - *RegWrite*
    - *MemRead*
    - *MemWrite*
    - …

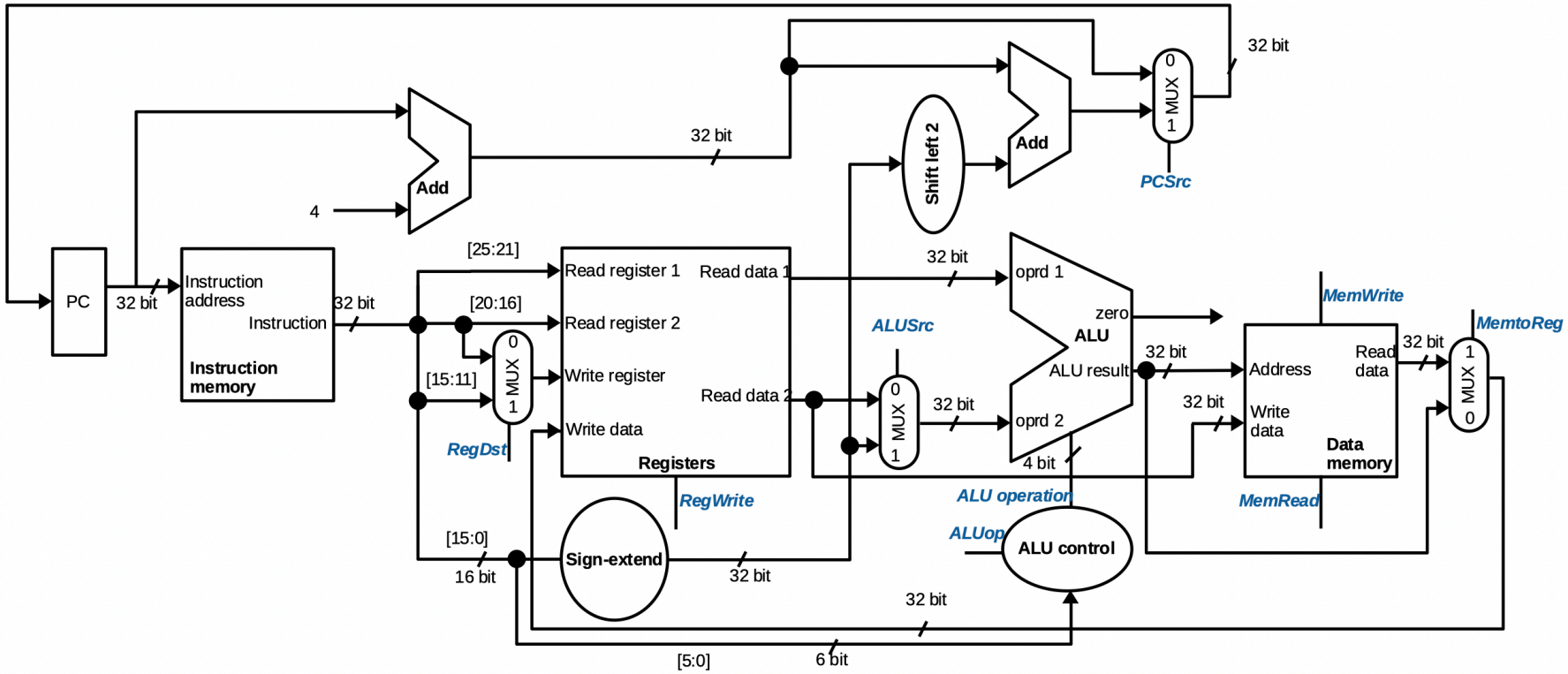
Extracting bits

NOT update any Register: sw, beq
Update some Register: addi, lw, slti….

- **Registers**:
  - Read register 1 ⇐ rs (instruction[25:21])
  - Read register 2 ⇐ rt (instruction[20:16])
  - Write register ⇐ rt/rd → need a multiplexer
- Sign-extend ⇐ address (instruction[15:0])



Datapath with bit-selection



Control block

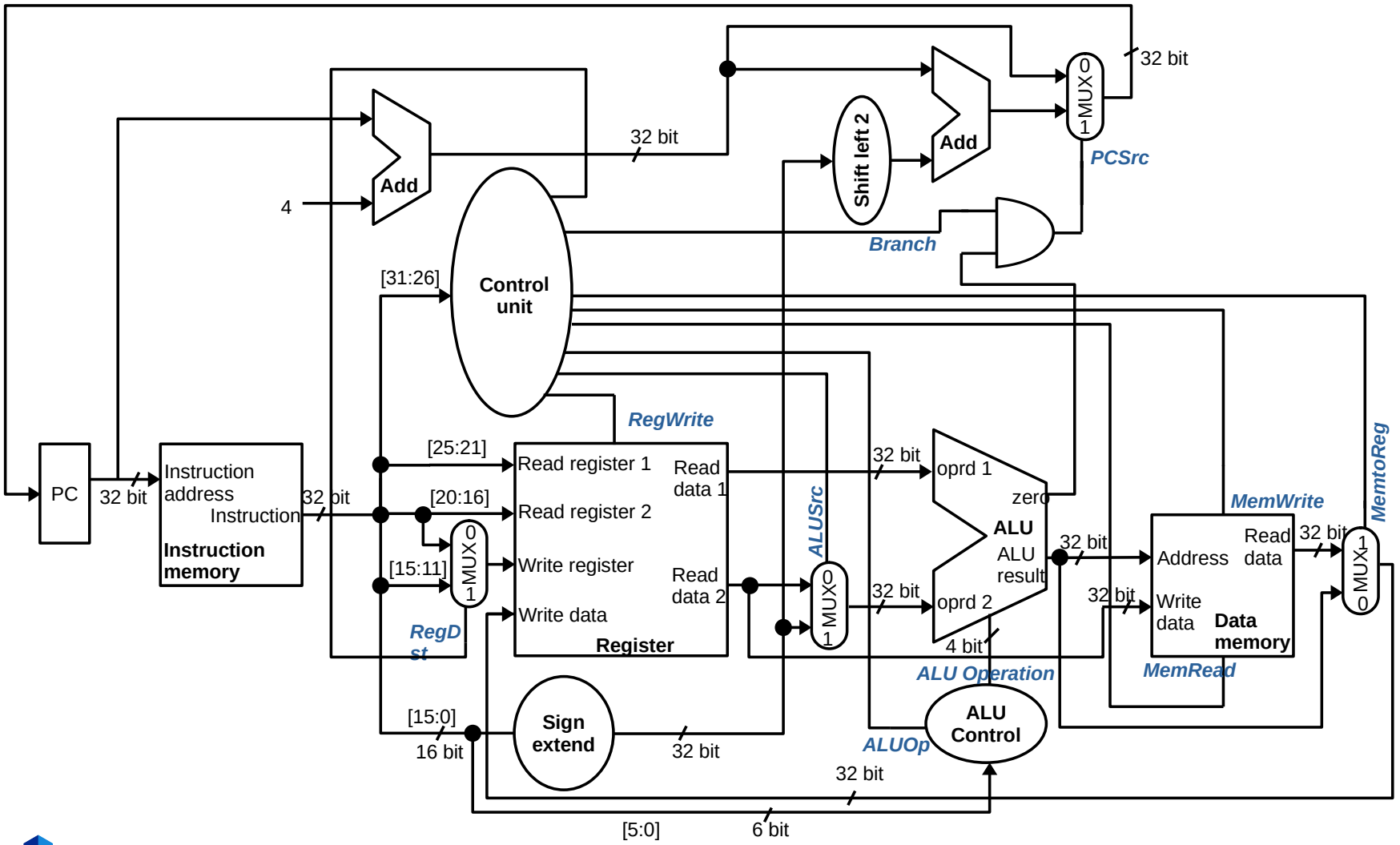
- Main function: handling multiplexers and functional units' control signals
  - ALU: two levels of decoding
    - Level 1: 6 bit opcode → 2 bit ***ALUop***
    - Level 2: 2 bit ***ALUop*** (+ 6 bit function field) → 4 bit ***ALU operation***
  - Muxes and control signals of functional units (**except ALU**): 6 bit opcode

- Predefined *ALU operation* values

Operator	ALU Opeation
and	0000
or	0001
add	0010
sub	0110
slt	0111

- *ALUop*:
  - opcode → add operator → ***ALUop*** = 00
  - opcode → sub operator → ***ALUop*** = 01
  - opcode → unknown → ***ALUop*** = 10

Full microarchitecture



Exercise

- **Question**: assume that registers store values of two times their numbers, for instance $s1 stores values of $17 \times 2 = 34$, please identify:

1. Which functional units contribute to the processing of following instructions
2. Values of inputs/outputs of functional units when processing following instructions
3. Values of control signals when processing following instructions

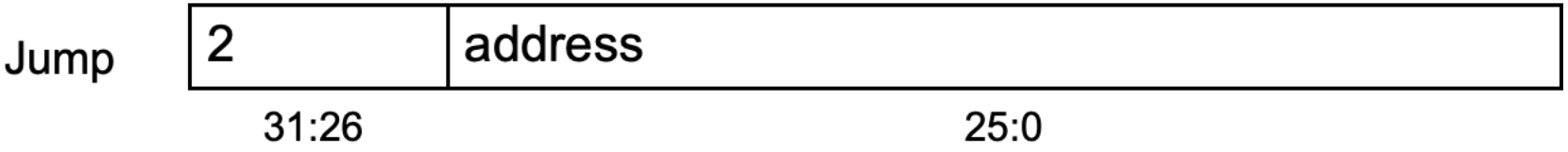
   - add $t0, $t1, $t2
   - addi $s0, $s1, 100
   - lw $s0, 100($s2) # memory word at address 136 stores values of 2025
   - sw $s0, 100($s2)
   - beq $s1, $s0, L1

Unconditional jump instructions

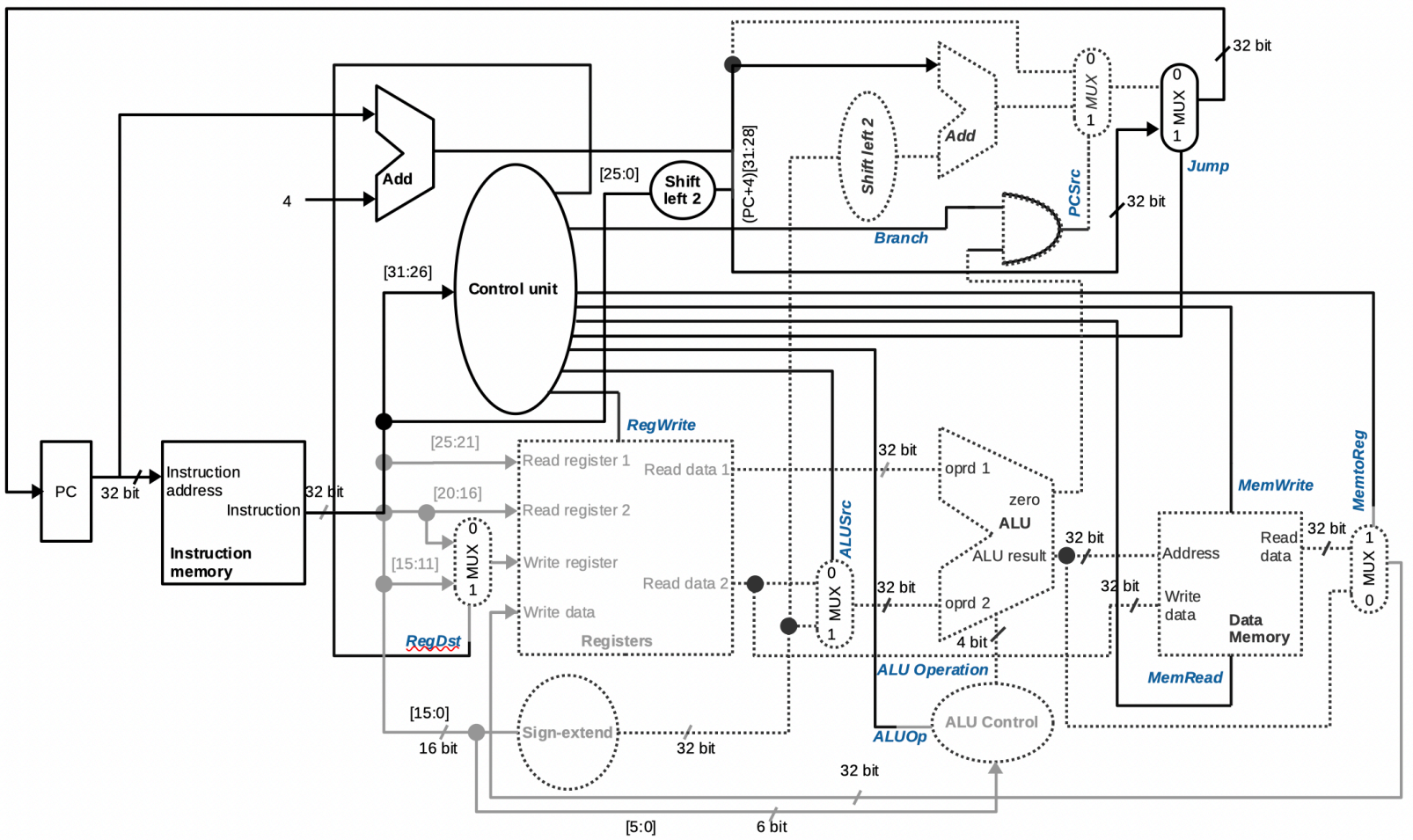
- Chapter 2: update **PC** with concatenation of
  - Top 4 bits of old **PC**
  - 26-bit jump address
  - 00

PC = {PC[31:28],address,00}

- One more way to update **PC** => Need a multiplexer & an extra control signal decoded from opcode



Datapath with Jumps added

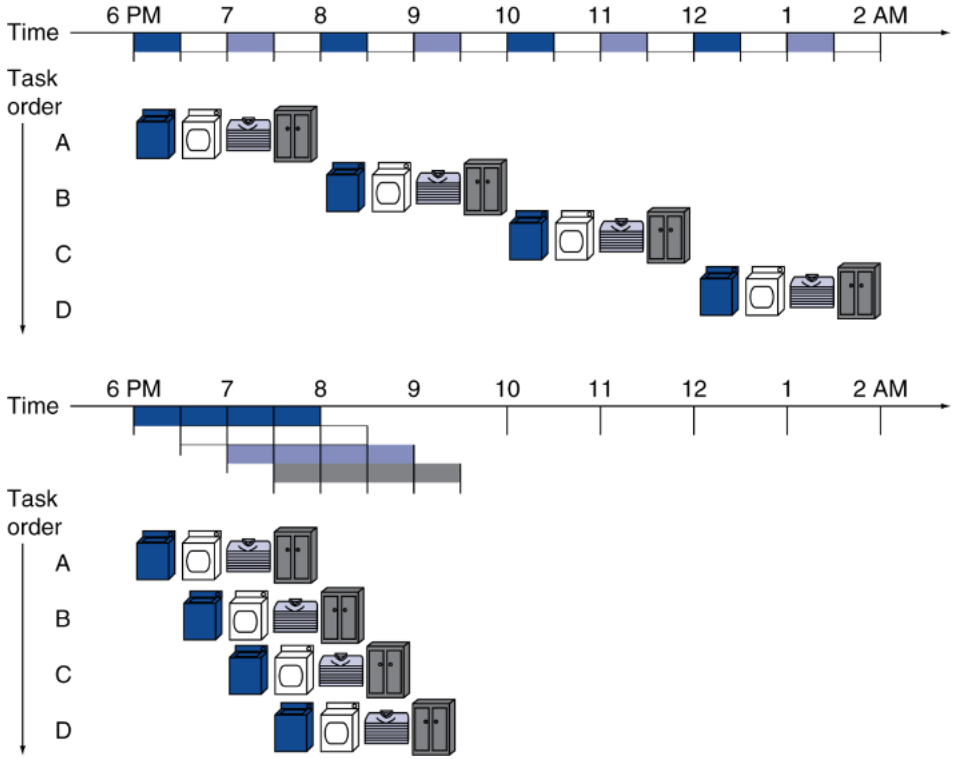


Performance issue

- **Simplified version** (CPI = 1): every instruction executed in **only one cycle**
  - Longest delay determines clock period
  - What is the longest instruction?
    - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- We will improve performance by pipelining

Pipelining analogy

- Pipeline laundry:
  - Overlapping execution
  - Improving performance (time for entire group)
- Four loads
  - Speed-up = 2.3×
  - Not impressive
- Non-stop (#loads → ∞)
  - Speed-up?
  - **Number of stages**



MIPS pipeline

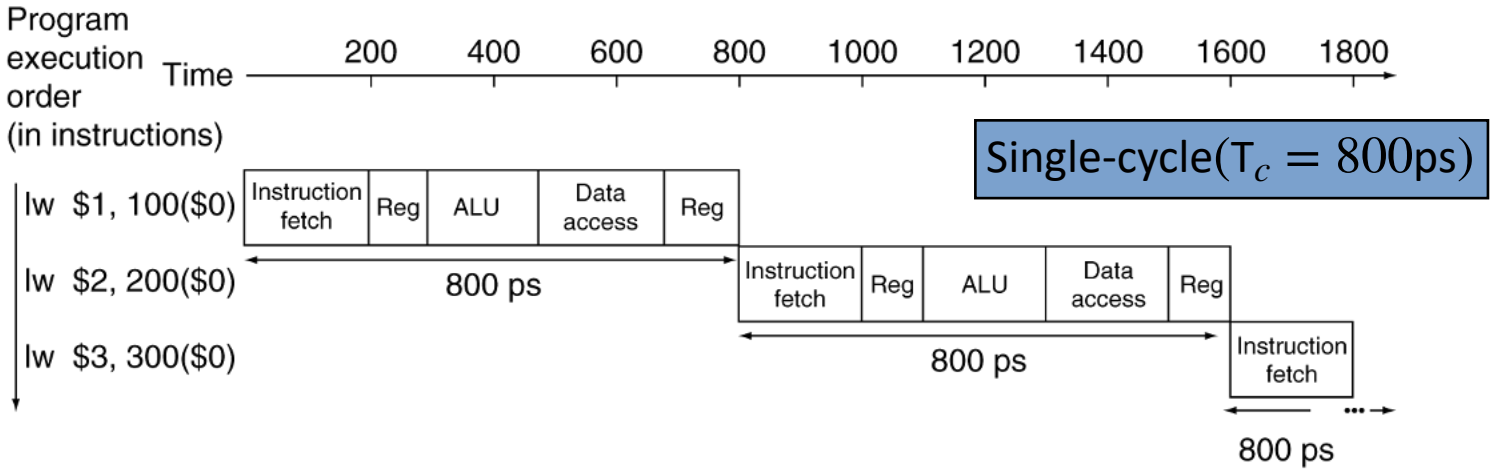
- **Five stages**, one step per stage
  1. Instruction fetch (**IF**): PC → instruction address
  2. Instruction decode (**ID**): register operands → register file
  3. Execute (**EXE**):
     - Load/store: compute a memory address
     - Arithmetic/logical: compute an arithmetic/logical result
  4. Memory access (**MEM**):
     - Load: read data memory
     - Store: write data memory
  5. Write back (**WB**):
     - Store a result of register file

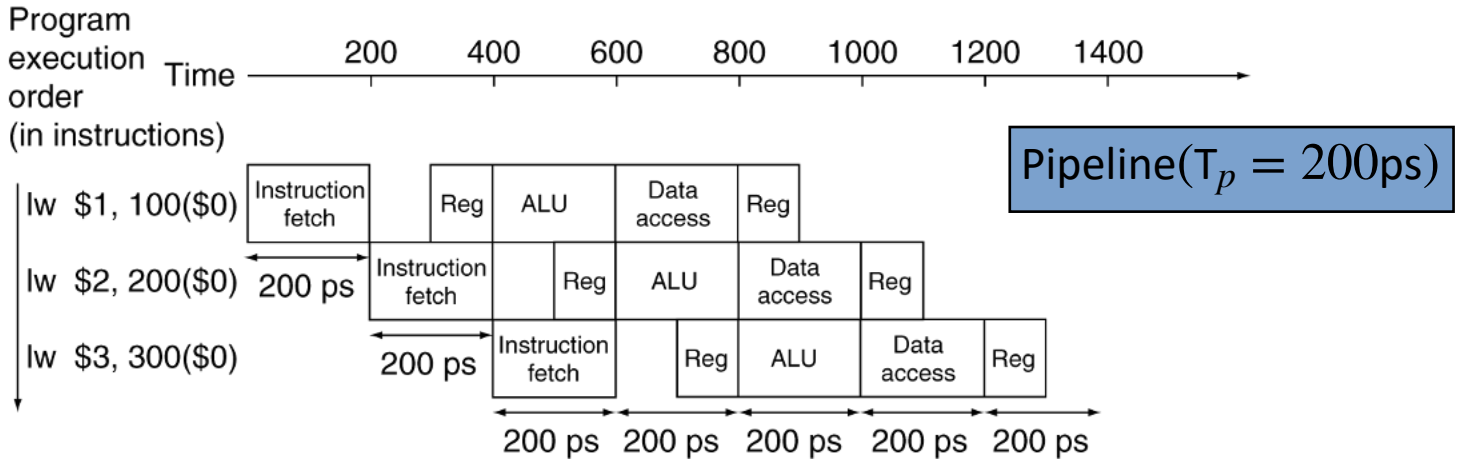
Pipeline performance

- Assume time for stages is
  - 100ps for register read or write (**ID** & **WB**)
  - 200ps for other stages (**IF**, **EXE**, & **MEM**)
- Compare **pipelined datapath** with **single-cycle datapath**

Instruction	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



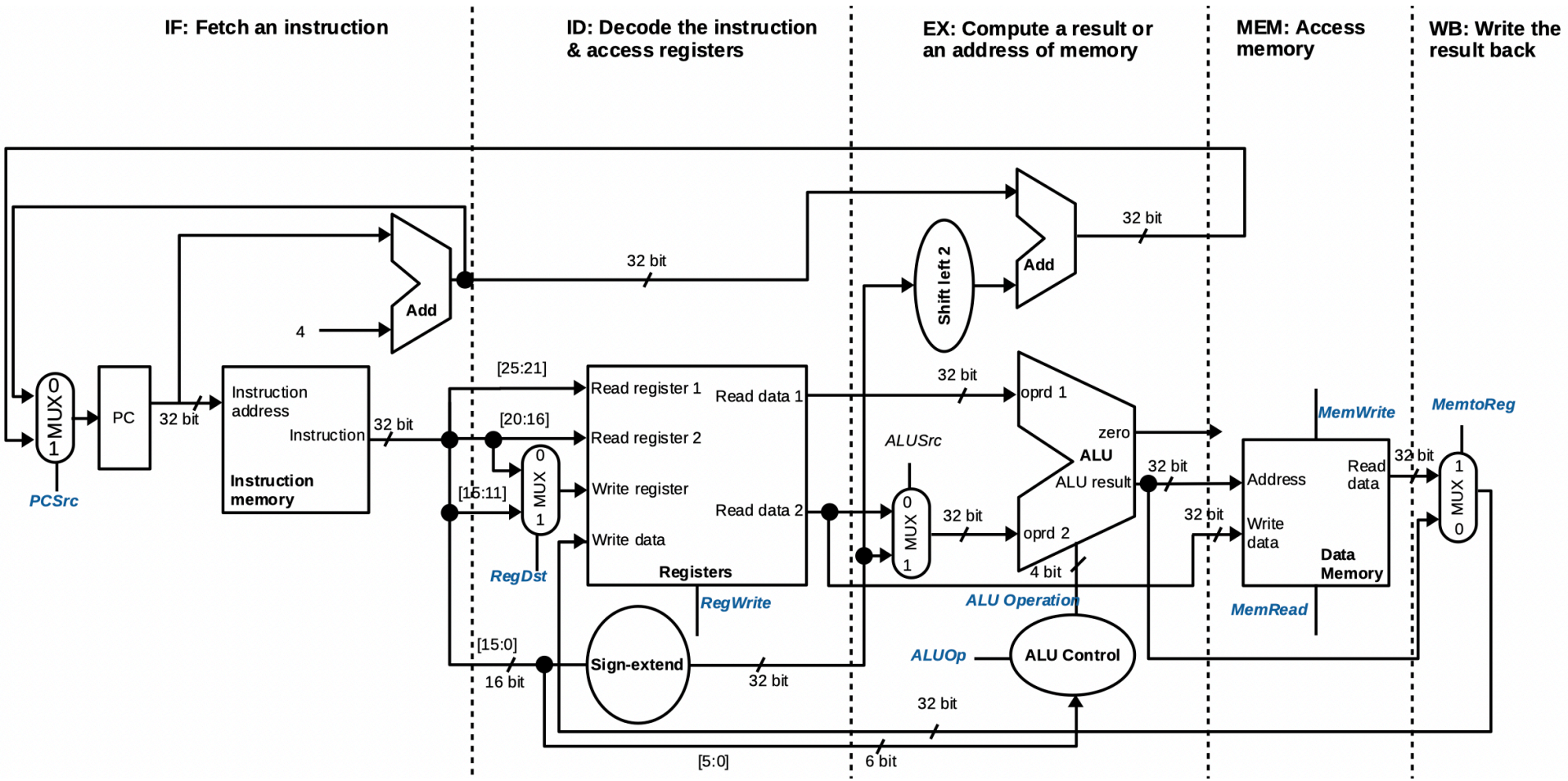


Pipeline speed-up

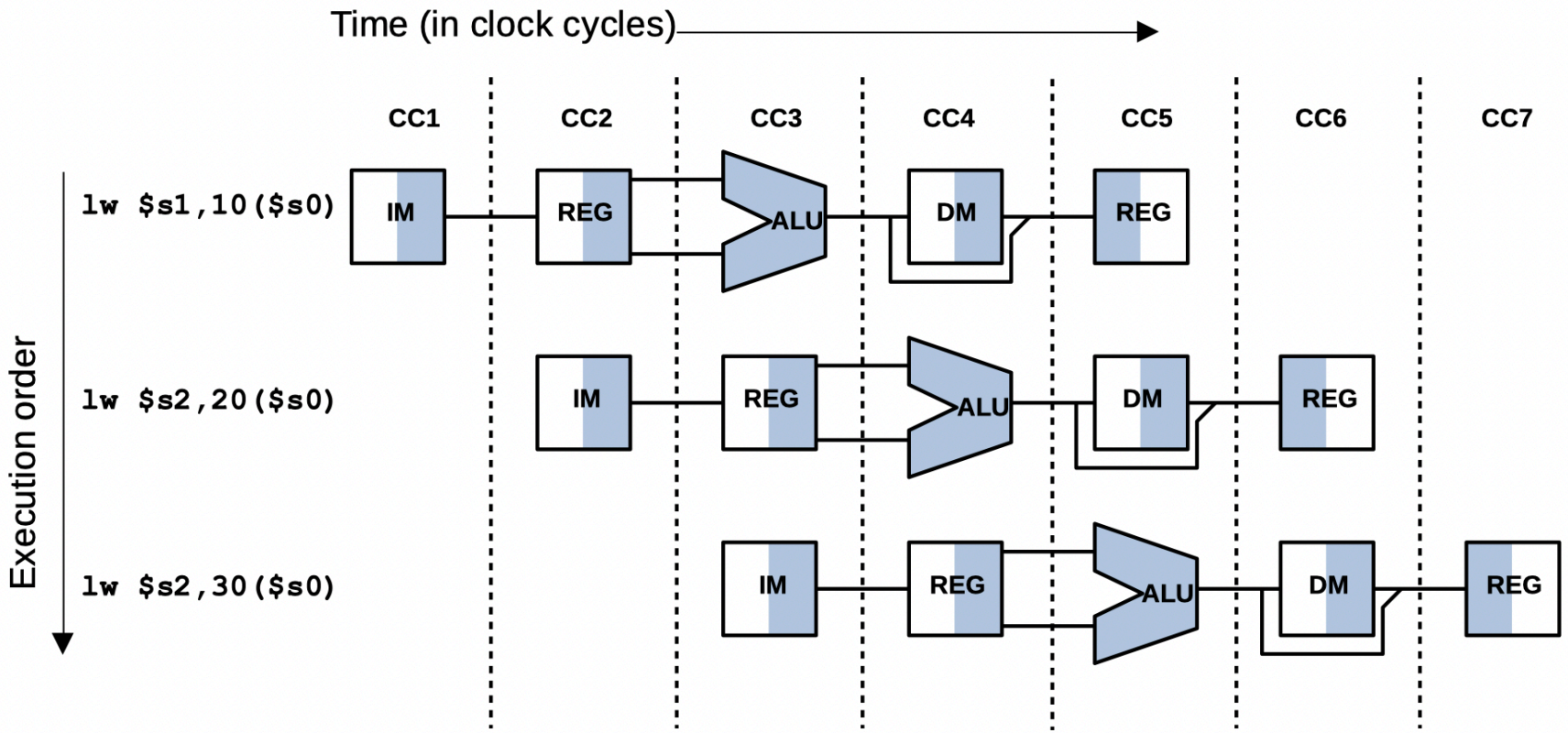
$$\text{speed-up} = \frac{\text{time of single-cycle}}{\text{time of pipelined}} = \frac{\text{time bw instruction}_{\text{single-cycle}}}{\text{time bw instruction}_{\text{pipelined}}}$$

- If all stages are **balanced**:
  - speed_up = number of pipe stages
- If **not balanced**, speedup is less
- Source of speedup
  - Throughput increased
  - Latency (time for each instruction) does not decrease
    - Sometimes **increased**

Pipeline datapath



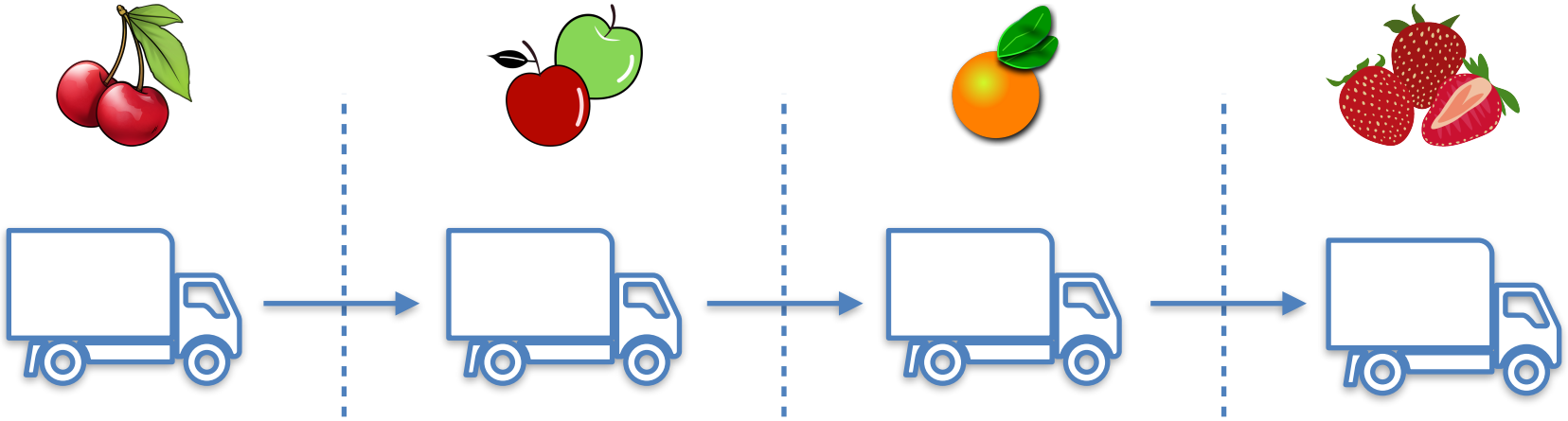
Instructions execution



Multi-Cycle Pipeline Diagram

How can we keep data when stages are not balanced?

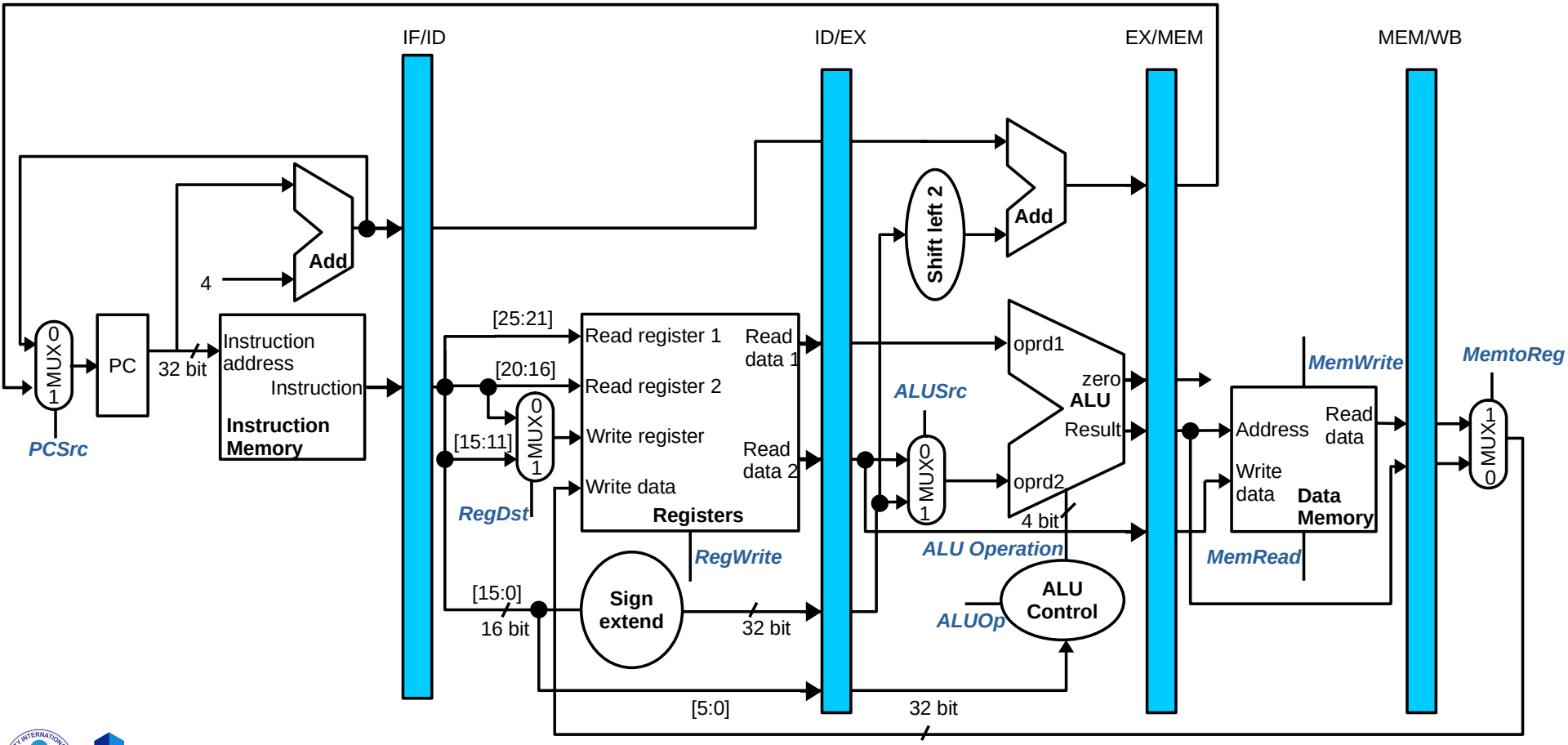
Wholesale market example



- Trucks move forward when completed
  - **Accident** at the Apple store
  - **Barriers** can help: open every hour (**cycle**)

Pipeline registers

Barriers in wholesale markets ⇔ **Registers** in digital circuits



Example

- **Question**: Given the following MIPS sequence:

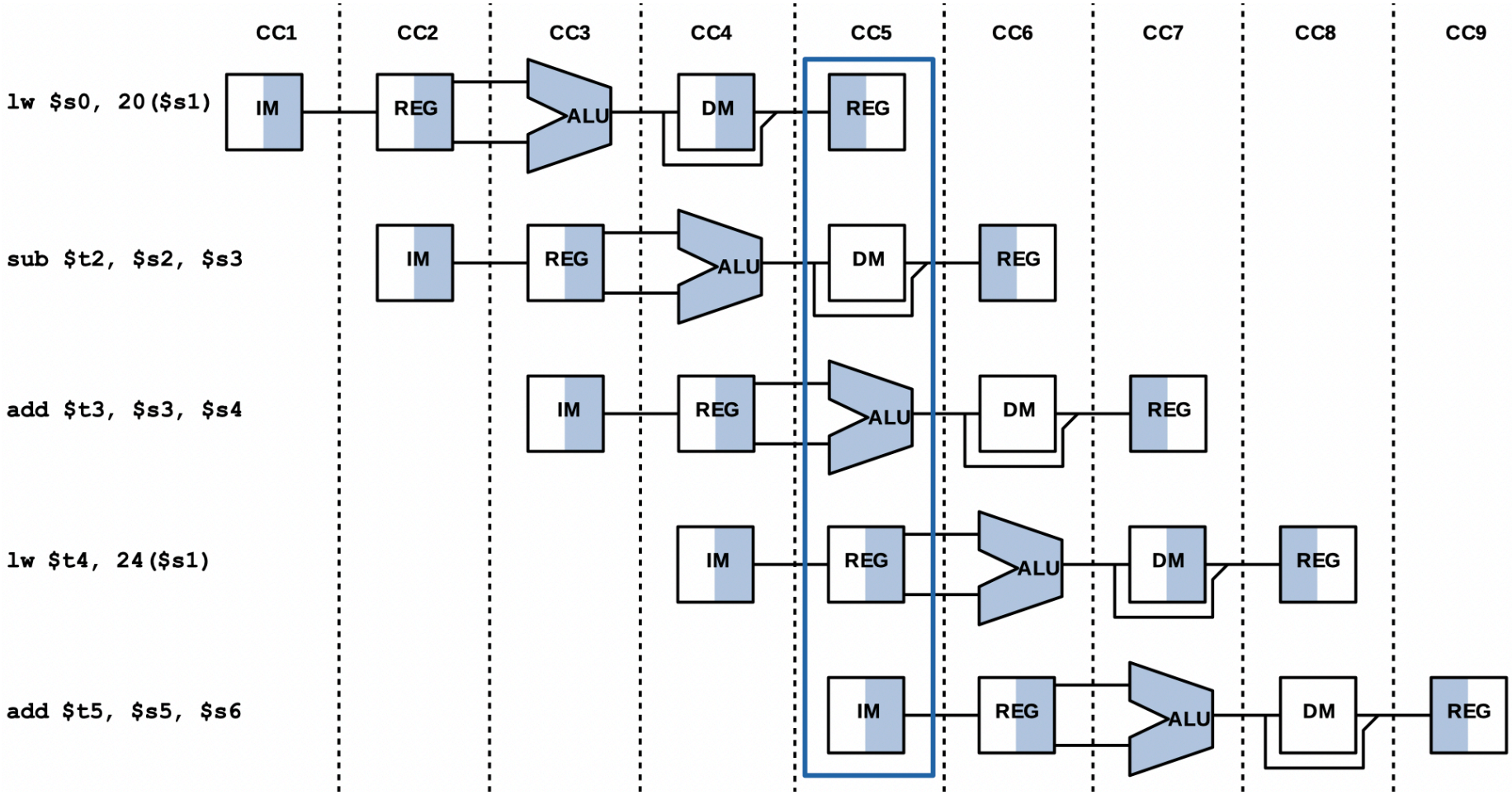
```
1. lw   $s0, 20($s1)
2. sub  $t2, $s2, $s3
3. add  $t3, $s3, $s4
4. lw   $t4, 24($s1)
5. add  $t5, $s5, $s6
```

Assume that the sequence is executed by a 5-stage pipelined MIPS processor

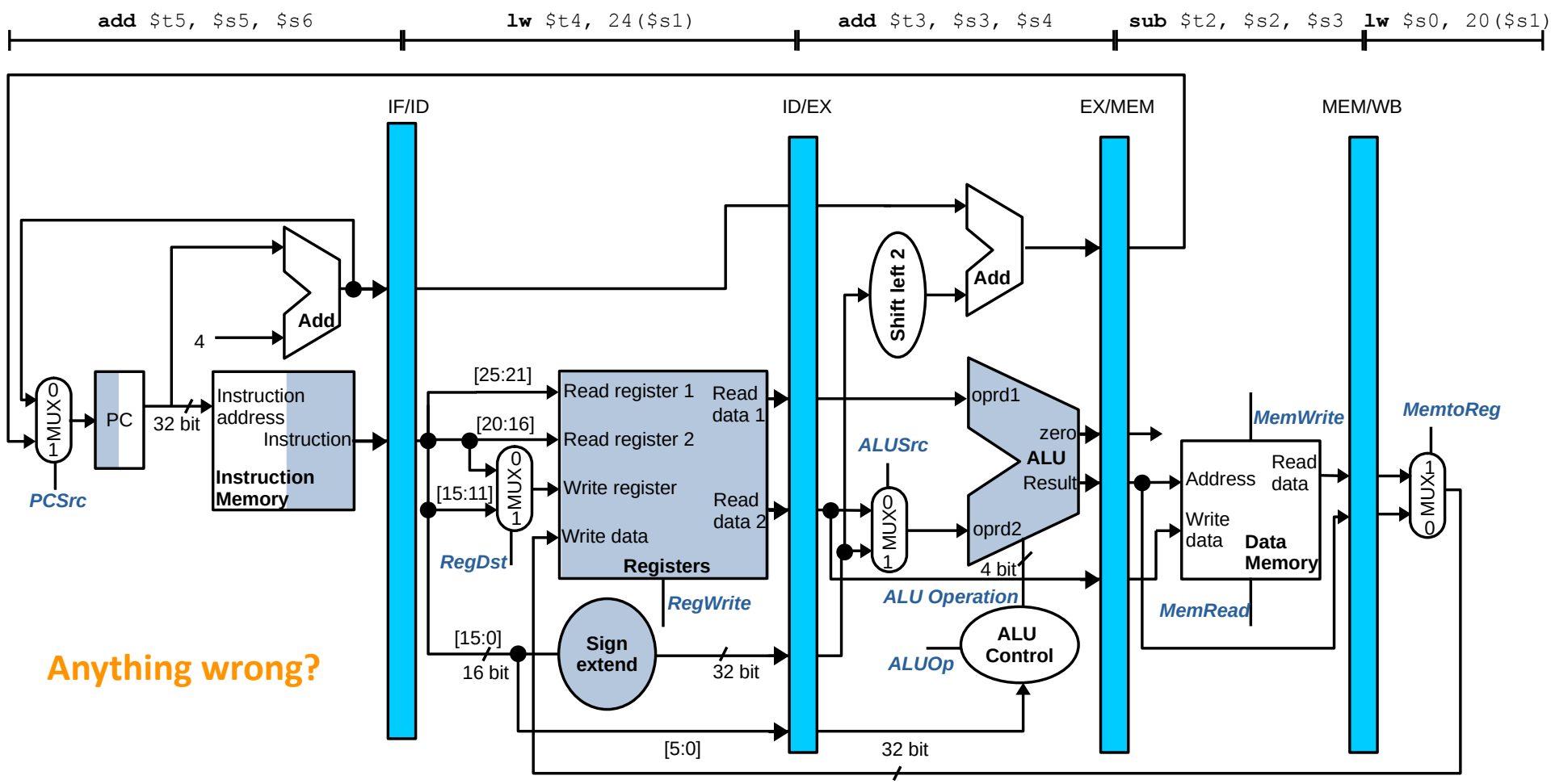
a) Draw a multi-cycle pipeline diagram for the sequence

b) Analyze the 5^{th} cycle with the datapath diagram in the previous slide

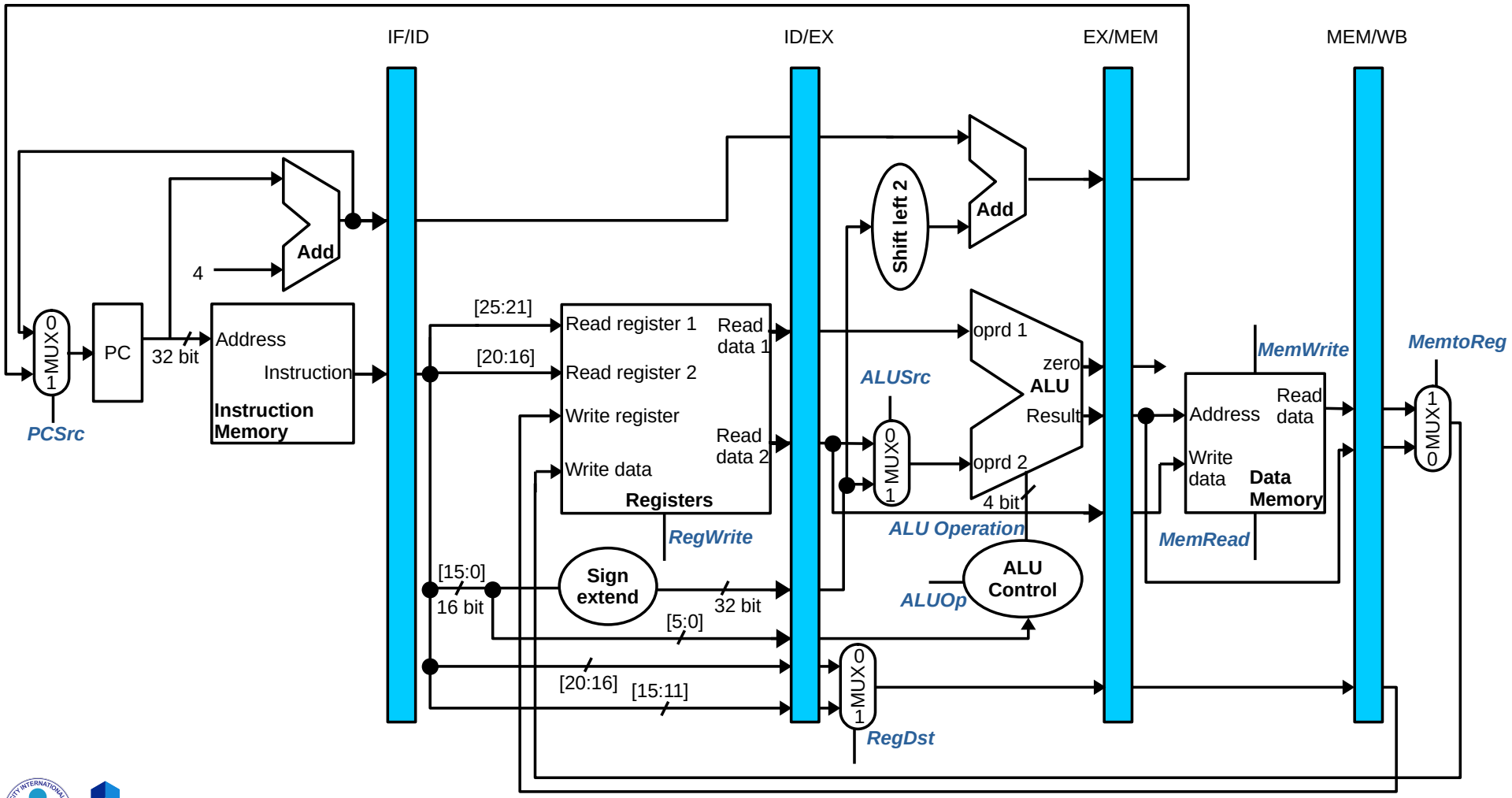
Multi-cycle pipeline diagram



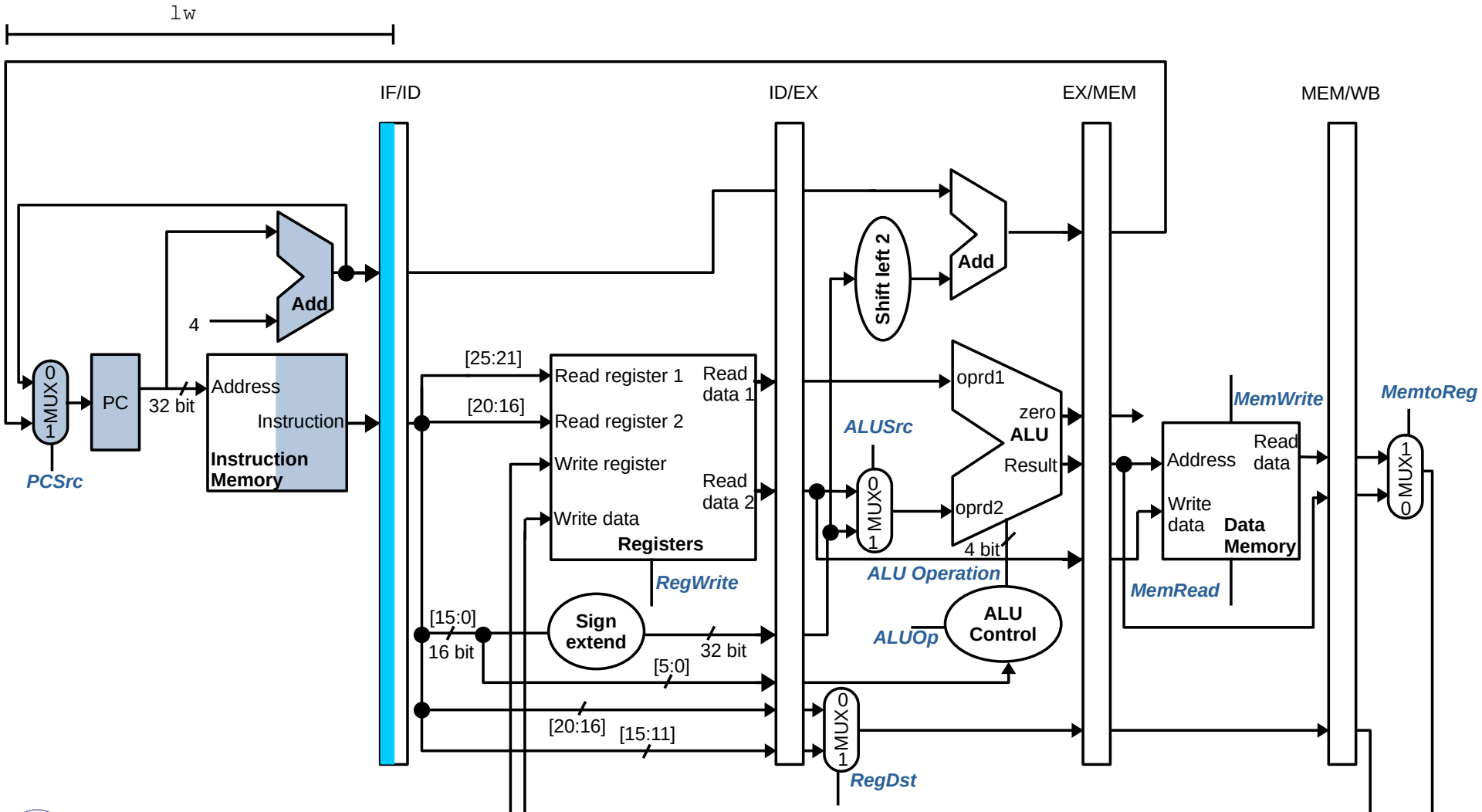
Single-cycle pipeline diagram



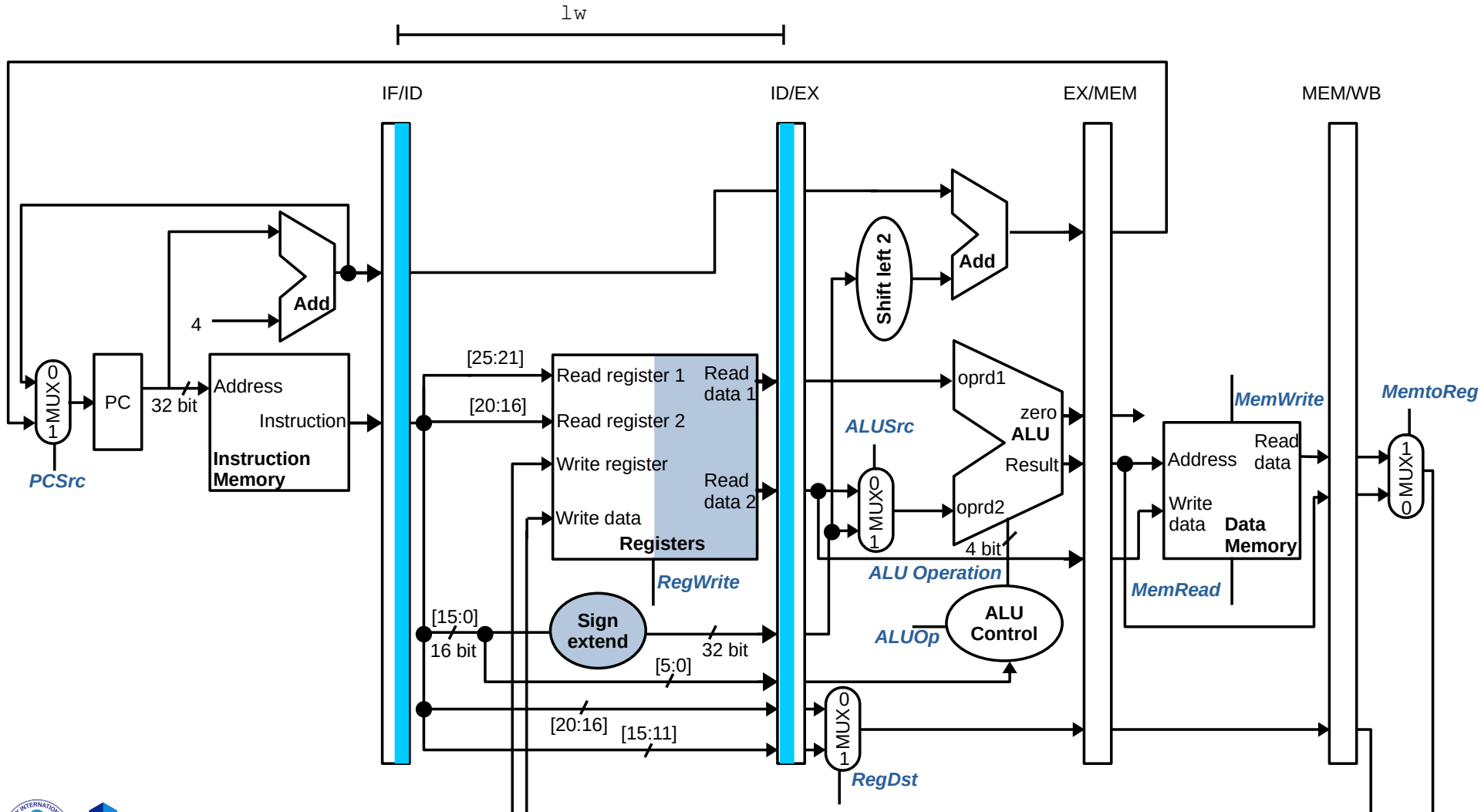
Corrected pipeline datapath



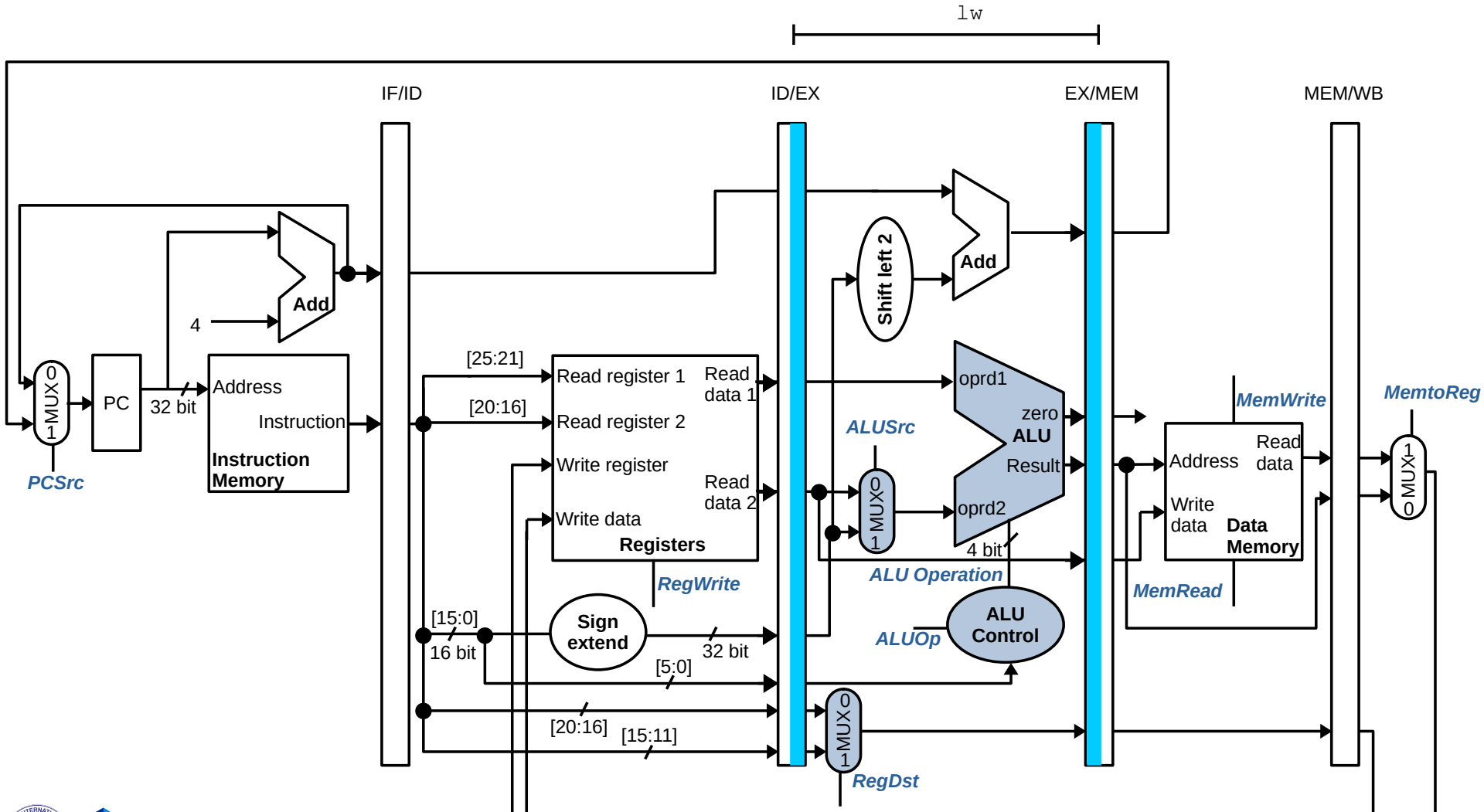
Example of lw - IF



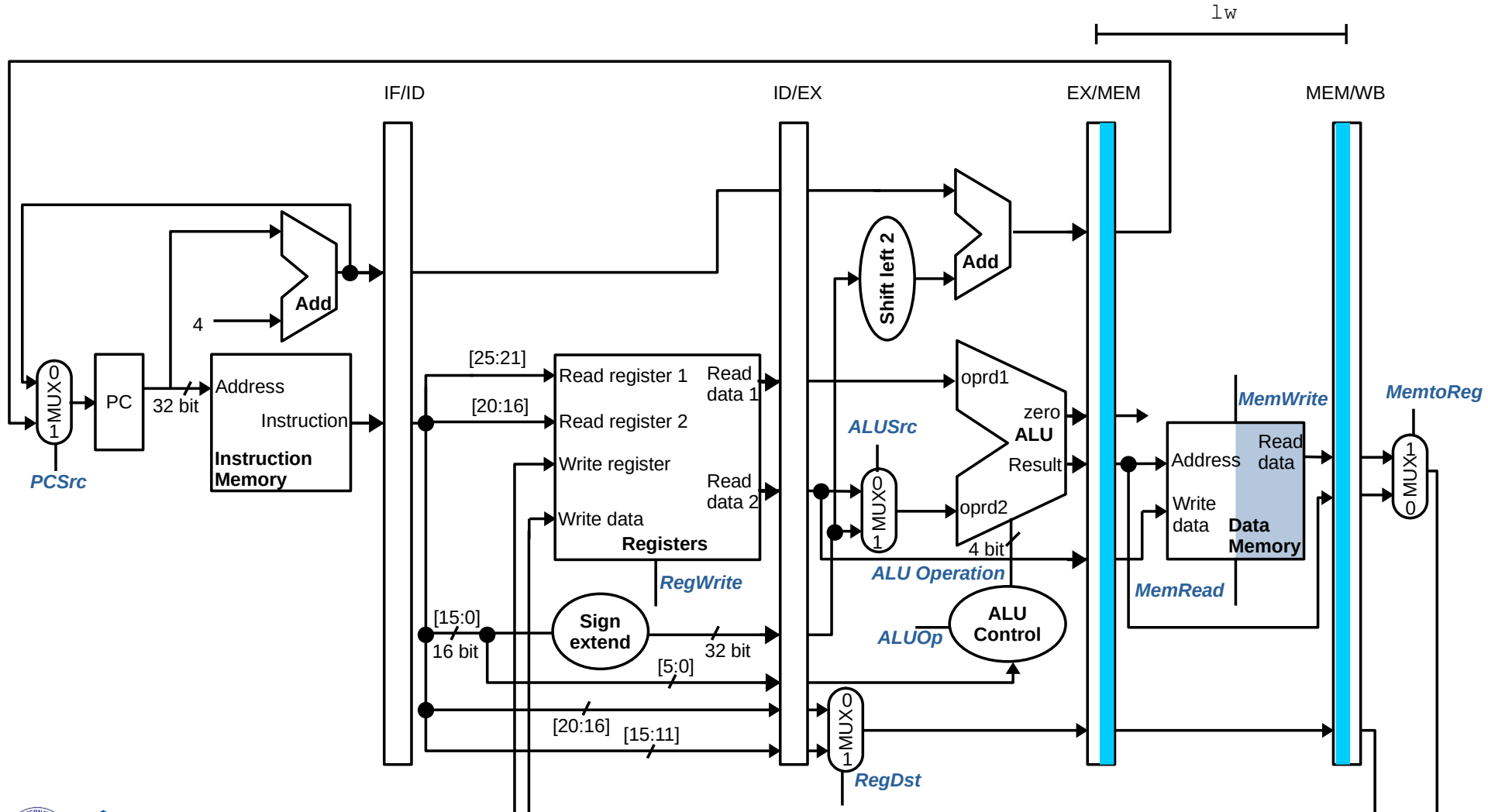
Example of lw - ID



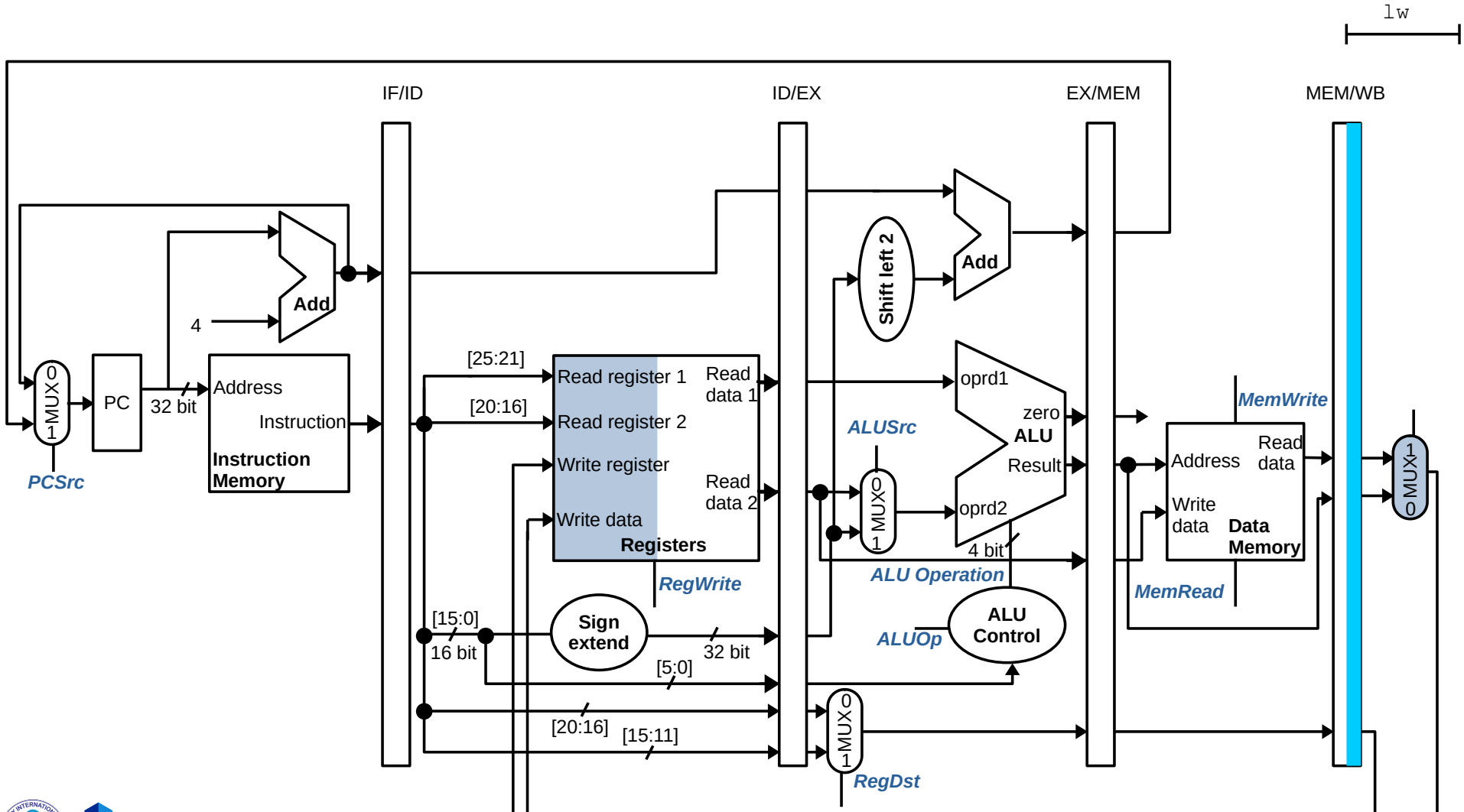
Example of lw - EXE



Example of lw - MEM

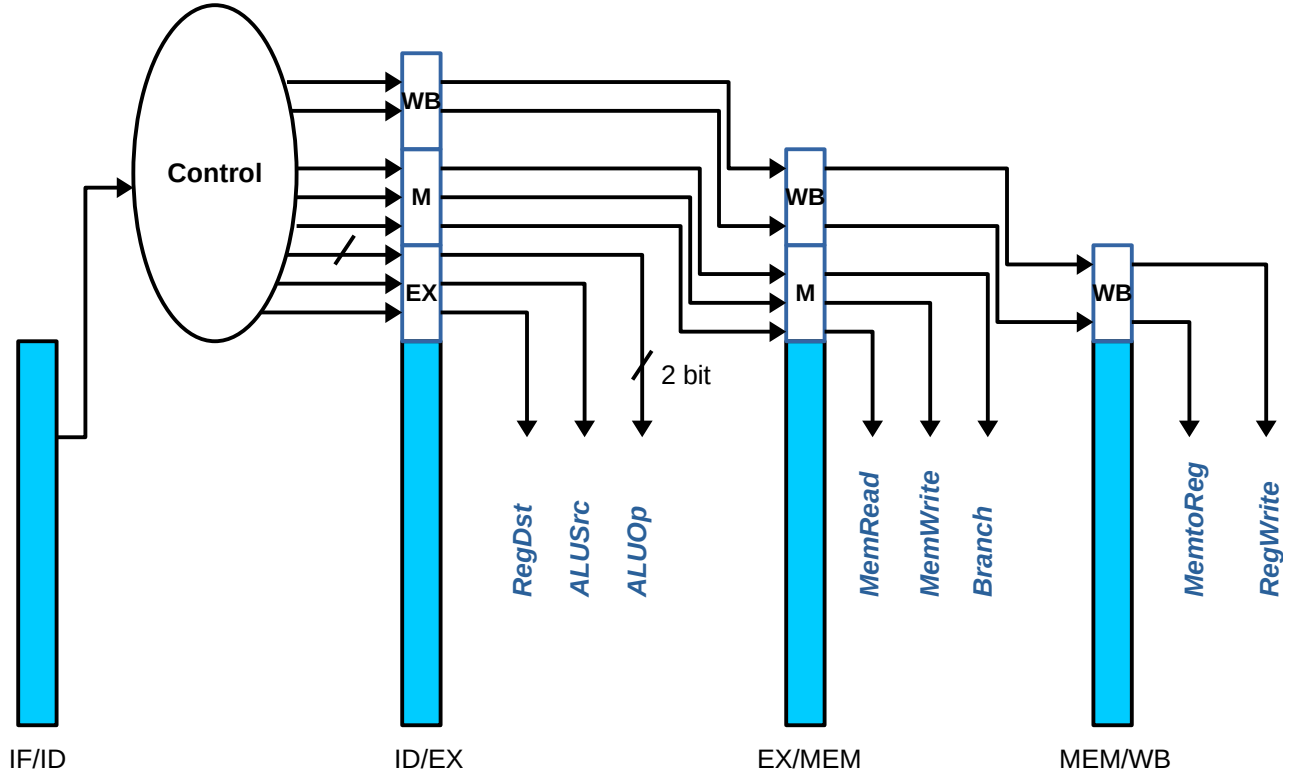


Example of lw - WB



Control signals

- Control signals travel across pipeline registers



Exercise

- **Question**: Given the following MIPS sequence:

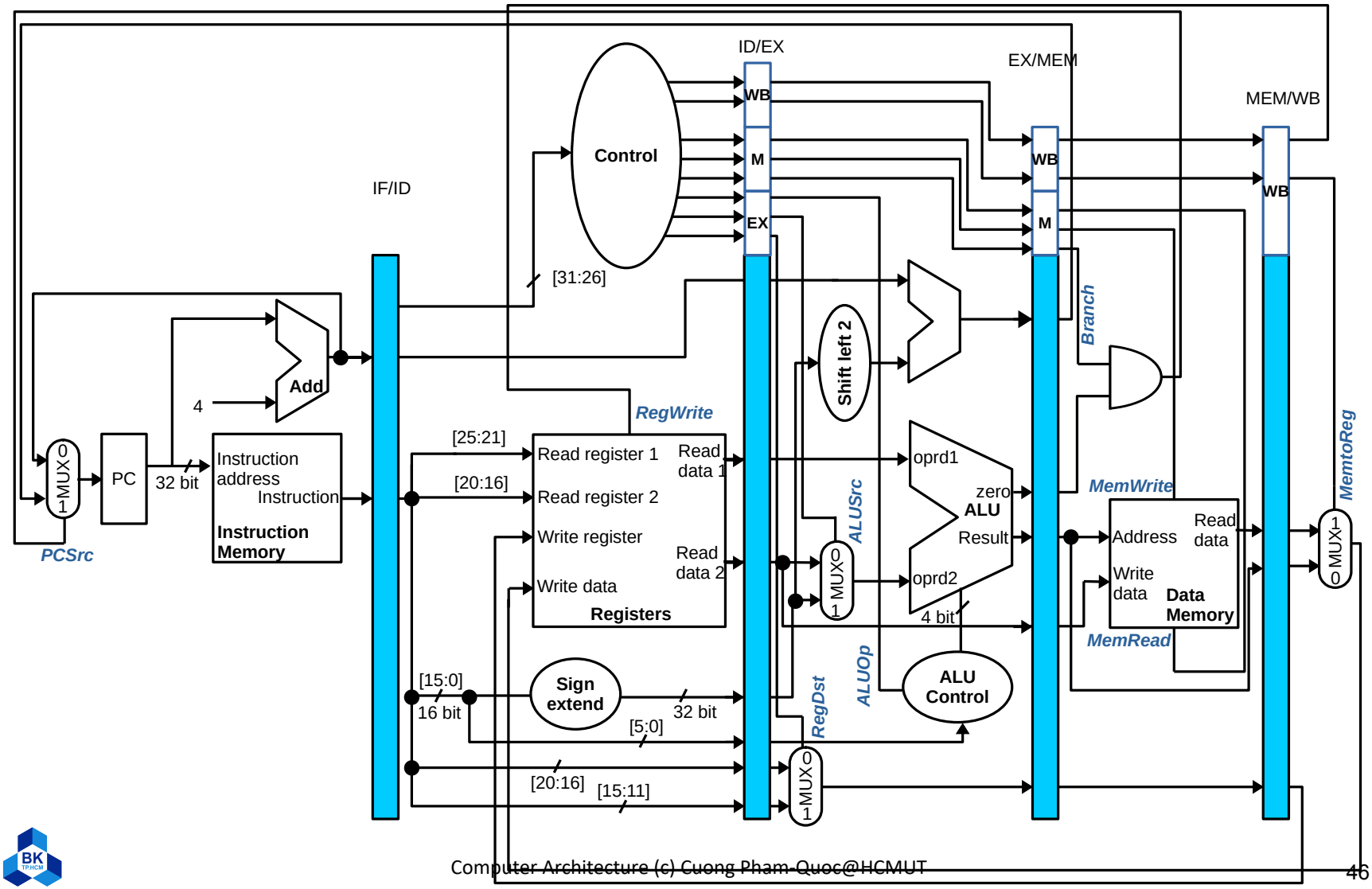
```
lw   $s0, 20($s1)
sub  $t2, $s2, $s3
add  $t3, $s3, $s4
lw   $t4, 24($s1)
add  $t5, $s5, $s6
```

Assume that the sequence is executed by a 5-stage pipelined MIPS processor

a) Identify values of control signals in cycle 5 **at the functional units**

b) Identify values of control signals in cycle 5 **at the Control block**

Full pipeline micro-architecture



Hazards

- Situations that **prevent** starting the next instruction in the next cycle
- Structure hazard
  - A required **resource** is busy
- Data hazard
  - Need to wait for previous instruction to complete its **data** read/write
- Control hazard
  - Deciding on **control action** depends on previous instruction

Structure hazards

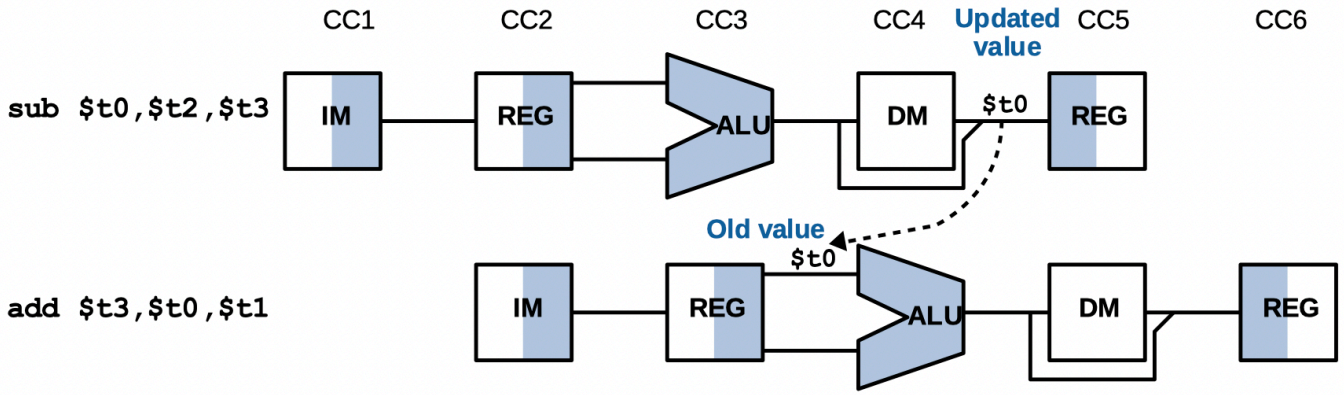
- **Conflict** for use of a resource
  - e.g.,: the laundry process, B forgot to bring clothes from the washing machine to dryer
- Should be eliminated entirely
  - **Stall** the pipe for that cycle
  - May repeat many times
- MIPS processors already **solved** all structure hazards
  - Separated instructions and data memory (caches)
  - Read and write registers use **different ports**

Data hazards

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

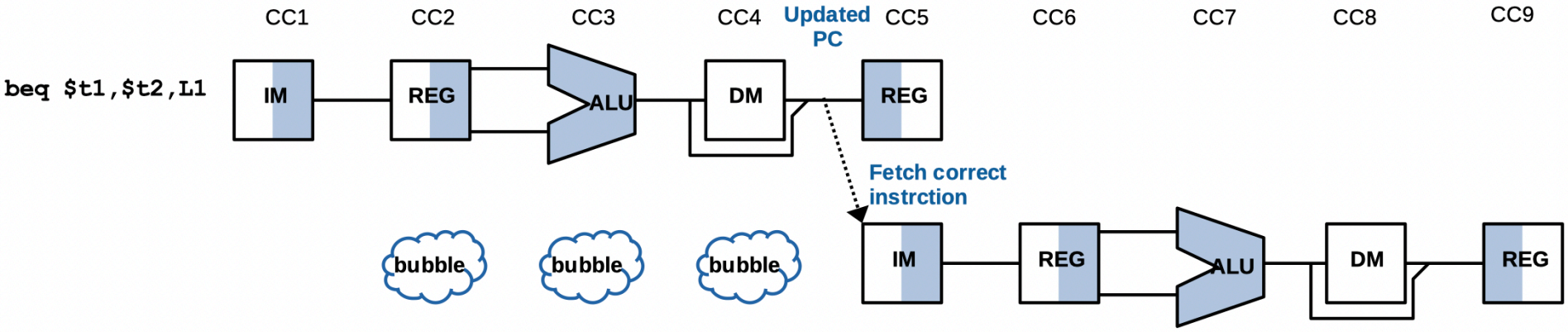
```
sub $t0, $t2, $t3
add $t3, $t0, $t1
```

- No any issues with the **simplified version**
- **Problem** with pipeline



Control hazards

- **Branch** determines flow of control
  - Fetching next instruction depends on branch outcome
- Pipeline **updates PC** in the MEM stage
  - Still working on ID stage of branch

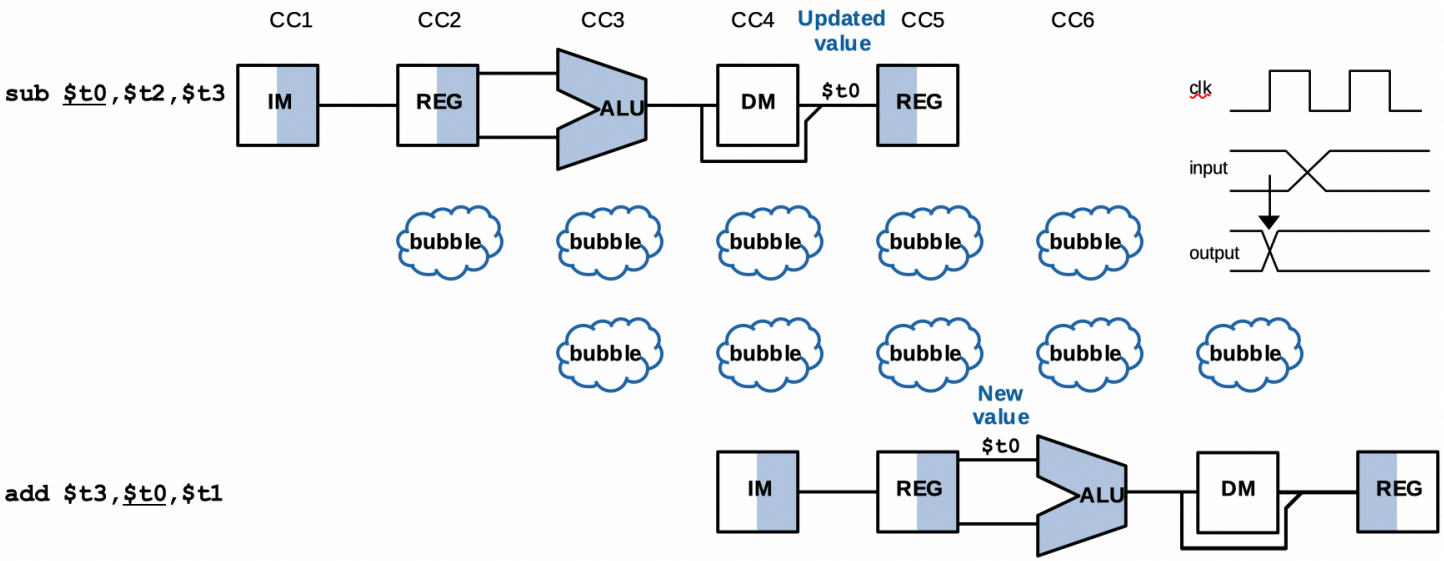


Data hazards solutions

1. **Code rescheduling**
   - Done by **compiler** (software level)
   - *Sometimes cannot* find solutions
2. **Delay or stalls insertion**
   - Done by **hardware**
   - **Always** can find solutions
   - *Increase* execution time & need *extra hardware* resources
3. **Forwarding**
   - Done by **hardware**
   - *Cannot* find solutions in a *special case*
   - Requires *extra hardware* resources

Code re-scheduling

- When **haven't** data hazards happened?
  - **Read** after or in the same cycle with **Write** (RAW)

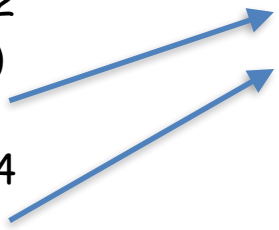


- **Find** and **swap** **data-independent** instructions

Example

- **Question:** given the following sequence of MIPS instructions

```
1: lw   $t1, 0($t0)
2: lw   $t2, 4($t0)
3: add  $t3, $t1, $t2
4: sw   $t5, 12($t0)
5: lw   $t4, 8($t0)
6: add  $t5, $t1, $t4
7: addi $t6, $t0, 4
```

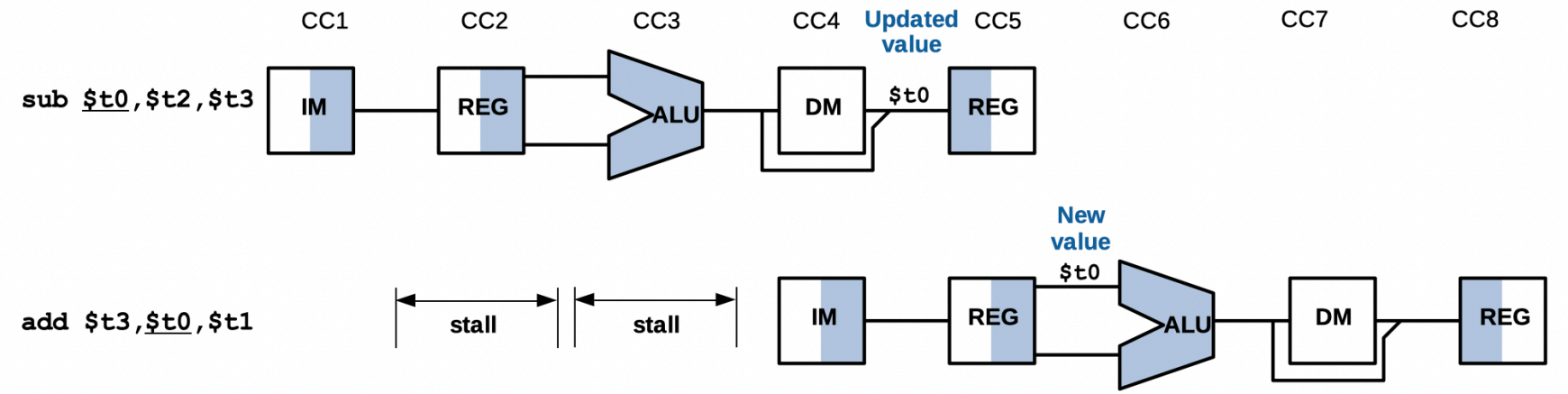


```
1: lw   $t1, 0($t0)
2: lw   $t2, 4($t0)
5: lw   $t4, 8($t0)
7: addi $t6, $t0, 4
3: add  $t3, $t1, $t2
4: sw   $t5, 12($t0)
6: add  $t5, $t1, $t4
```

  - Identify data hazards and solve by code re-scheduling
- **Answer**:
  - The third and the sixth instructions (add $t3, $t1, $t2 and add $t5, $t1, $t4) have data hazards
  - Move two data-independent instructions 5 and 7 to right after the second instruction

Stalls insertion (Delay)

- When **haven't** data hazards happened?
  - **Read** after or in the same cycle with **Write** (RAW)



- **Delay** instructions that use data:
  - **ID** stage of the instruction **using** data ≡ **WB** stage of the instruction **producing** data

Example

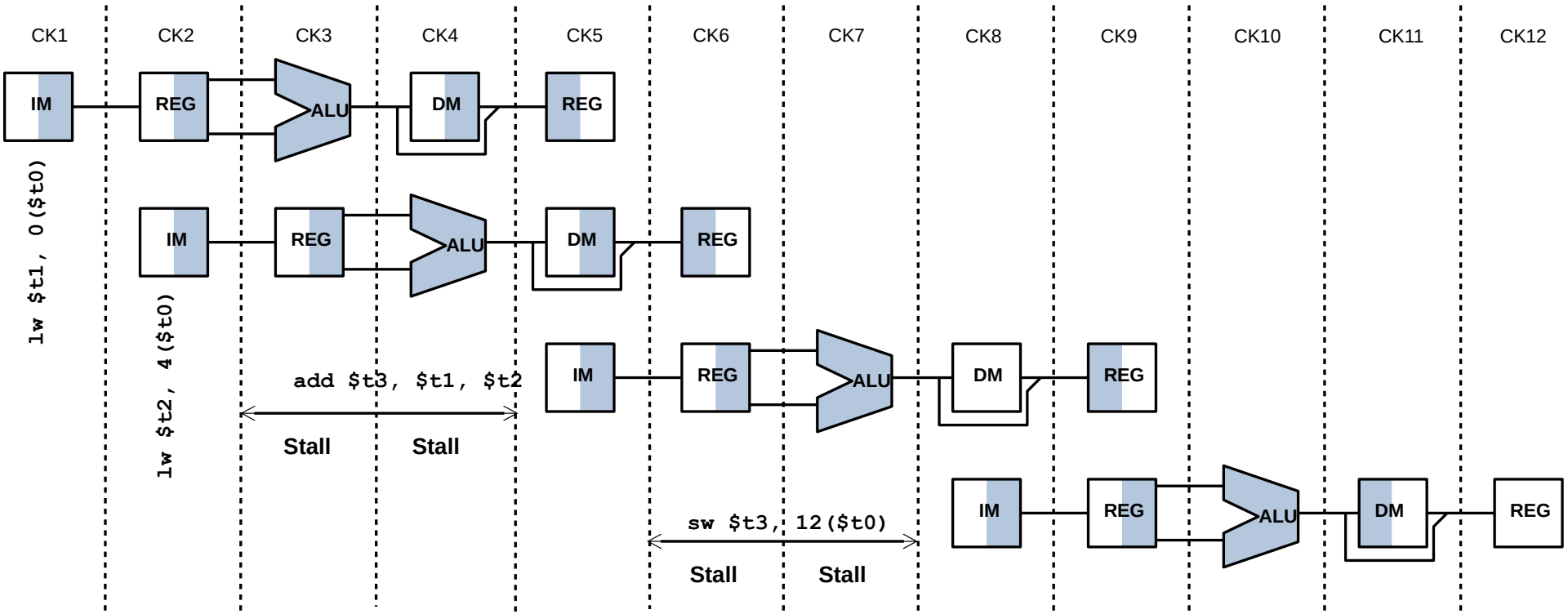
- **Questions**: given the following MIPS sequence of instructions

```
1: lw $t1, 0($t0)
2: lw $t2, 4($t0)
3: add $t3, $t1, $t2
4: sw $t3, 12($t0)
```

  - Identify data hazards and solve them by the stalls insertion method; how many cycles needed for the sequence?

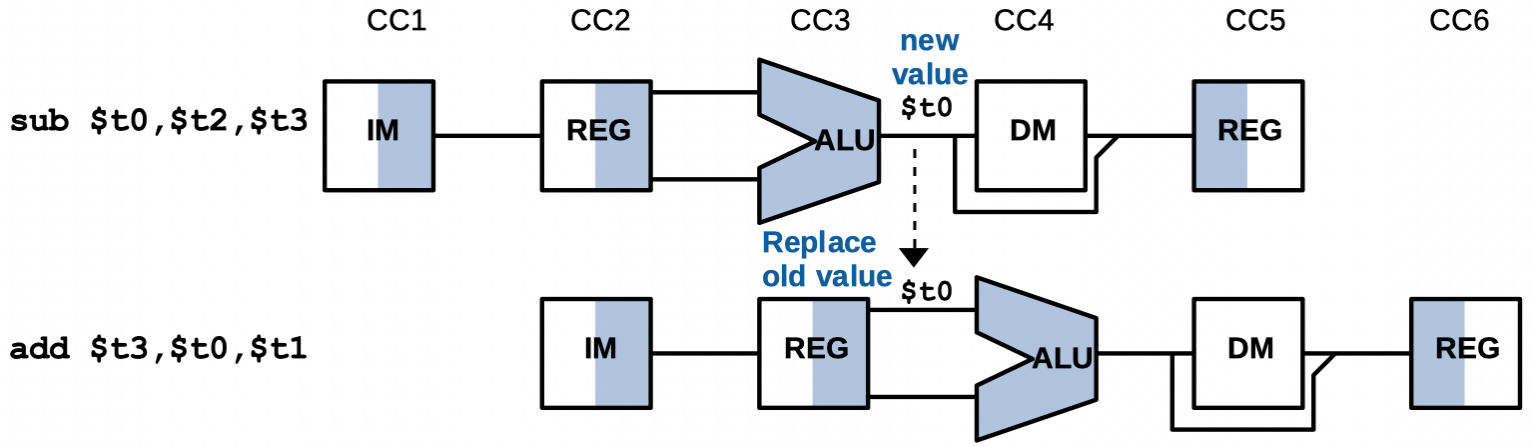
- **Answer**:
  - Data hazards (2) - (3) & (3) - (4)
  - Insert two stalls for the 3^{rd} instruction & two stalls for the 4^{th} instruction
  - 12 cycles needed

Example (cont.)



Forwarding

- When can an instruction use data at **soonest**?
  - **Right after** data is **produced** (**EXE** stage & **MEM** stage)
- When is data **processed**?
  - **Apply** operators: **EXE** state



- **Use** data right after created

Example

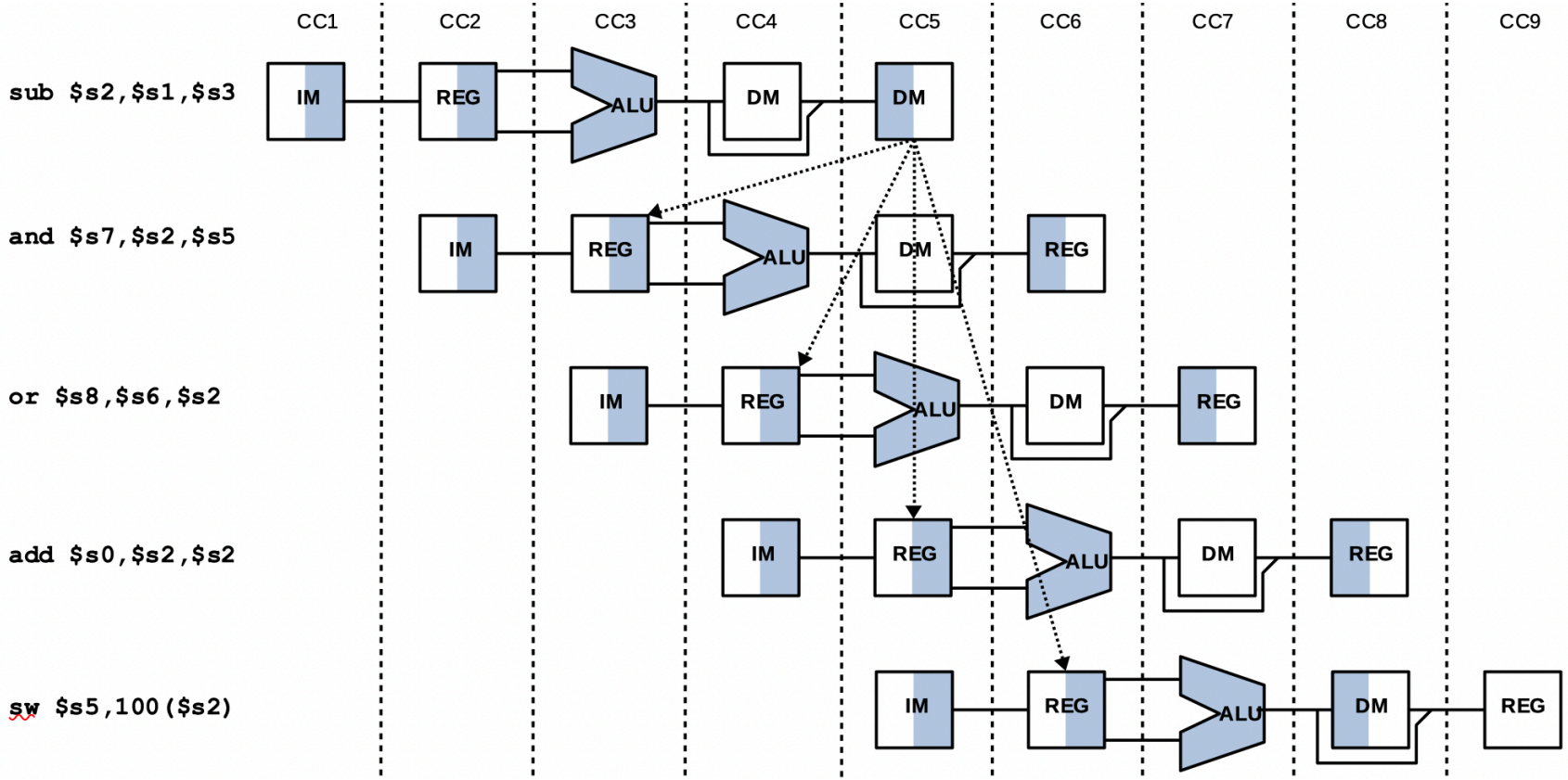
- **Question**: given the following MIPS sequence

```
1: sub $s2, $s1, $s3
2: and $s7, $s2, $s5
3: or $s8, $s6, $s2
4: add $s0, $s2, $s2
5: sw $s5, 100($s2)
```

  - Analyze data dependencies & identify data hazards
- **Answer**:
  - $s2 **produced** by the 1^{st} instruction **used** by all other instructions
  - **Data hazards**: 2^{nd} & 3^{rd} instructions

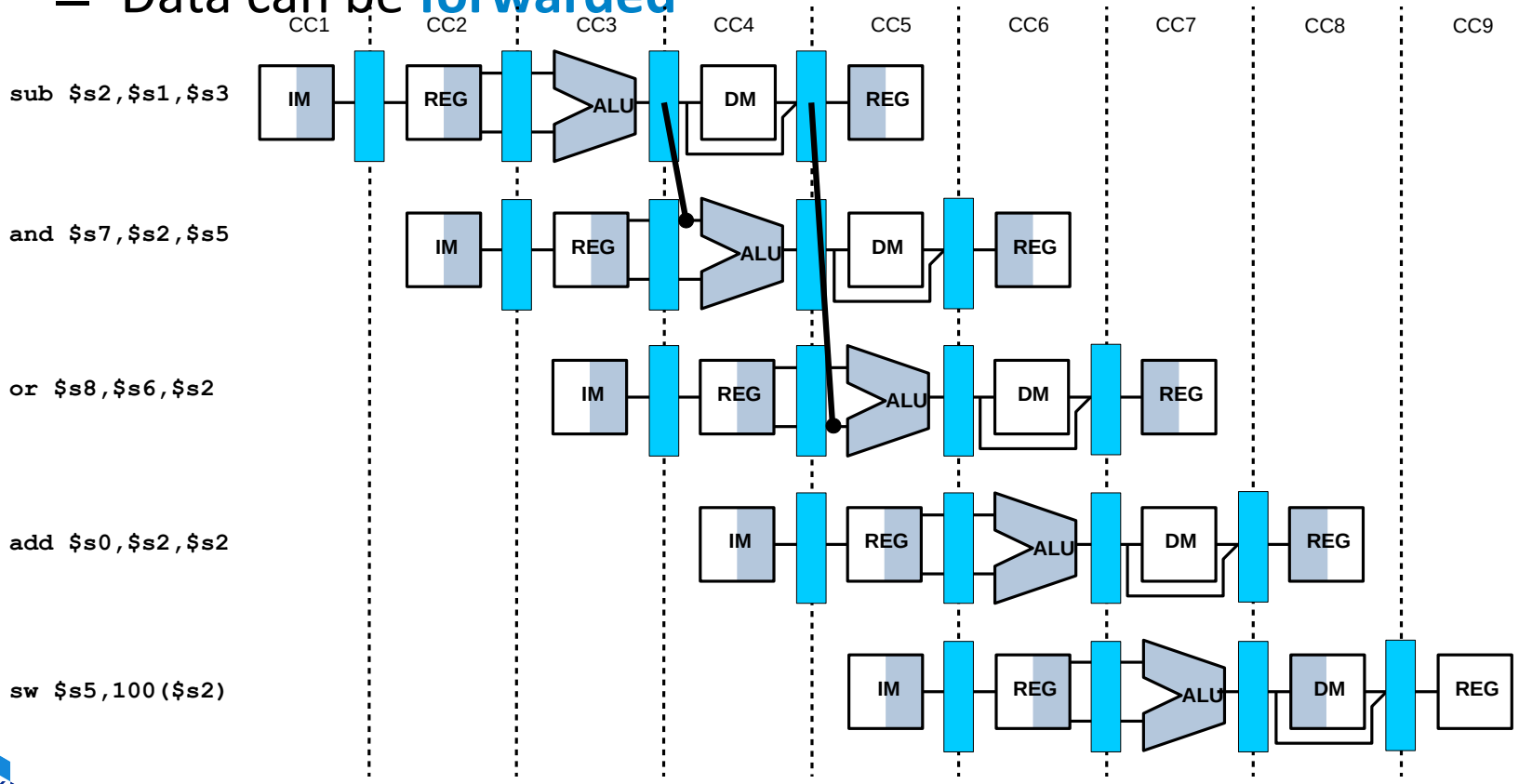
Example (cont.)

- **Answer** (cont.):



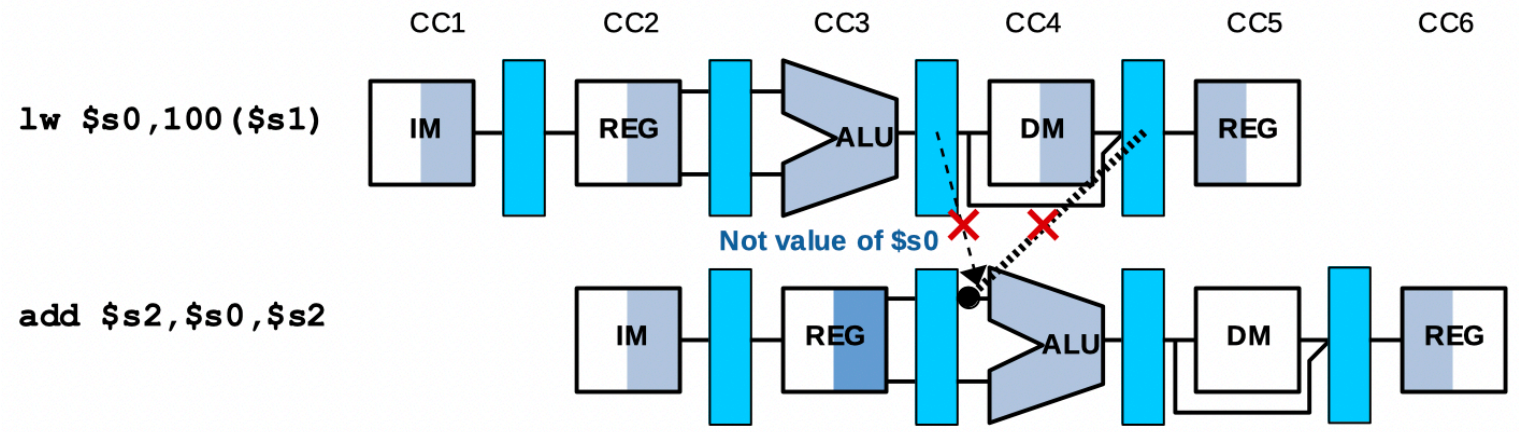
Forwarding

- Consider the previous sequence of MIPS instructions
  - Data can be **forwarded**



Load used data hazards

- When instruction producing data is a **load**, can data be forwarded?
  - **Cannot forward** to the next instruction since data is produced **later**

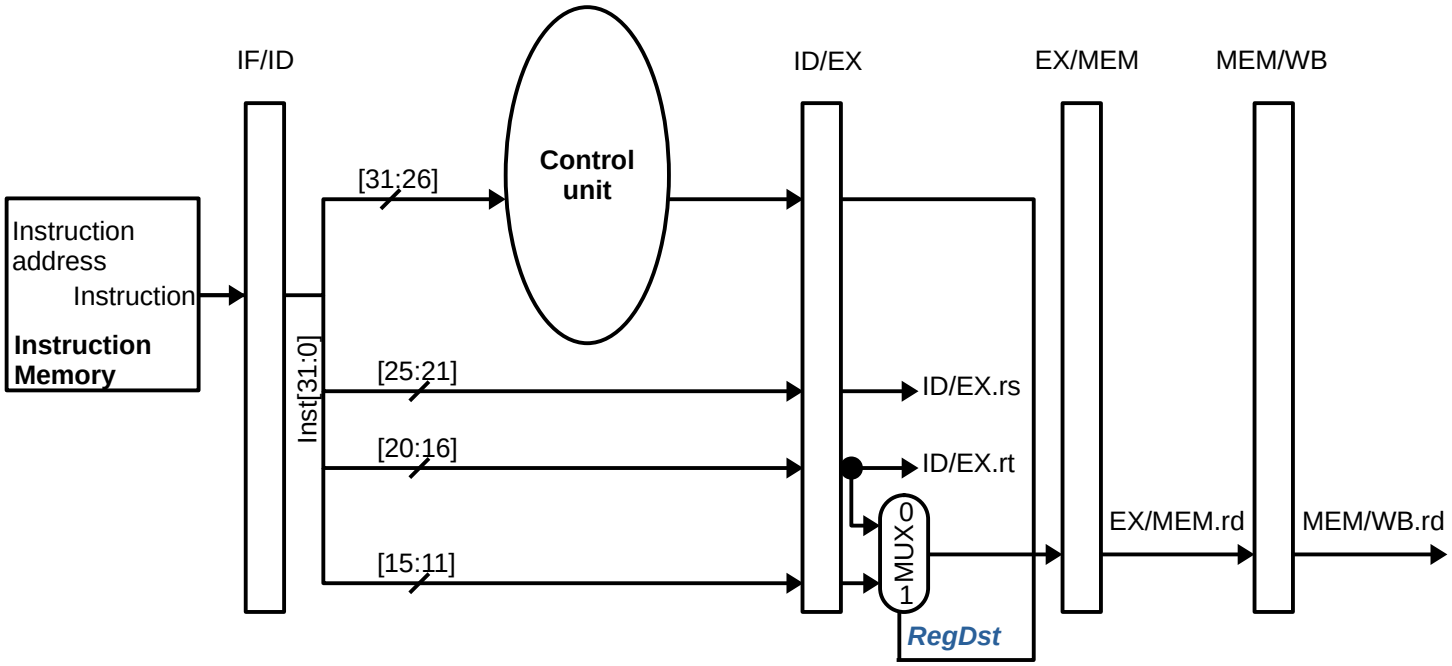


  - Delay:
    - **1 stall with** forwarding or **2 stalls without** forwarding

Detecting data hazards

- **EXE** hazard **versus** **MEM** hazard
  - **EXE** hazard: forward from the **EX/MEM** register
    - **Destination register** in the MEM stage ≡ one of the source registers of the instruction in EXE
  - **MEM** hazard: forward from the **MEM/WB** register
    - **Destination register** in the WB stage ≡ one of the source registers of the instruction in EXE
- **Passing** register numbers along the pipe
  - ID/EX.rs & ID/EX.rt: first & second sources
  - EX/MEM.rd; MEM/WB.rd: destinations

EX hazard conditions



1a. ID/EX.rs = EX/MEM.rd

1b. ID/EX.rt = EX/MEM.rd (not happened when an **I-format** instruction is in the **EXE** stage)

MEM hazard conditions

- Consider the following MIPS sequence of instructions

```
1: add $s0, $s0, $s1
2: add $s0, $s0, $s2
3: add $s0, $s0, $s3
```

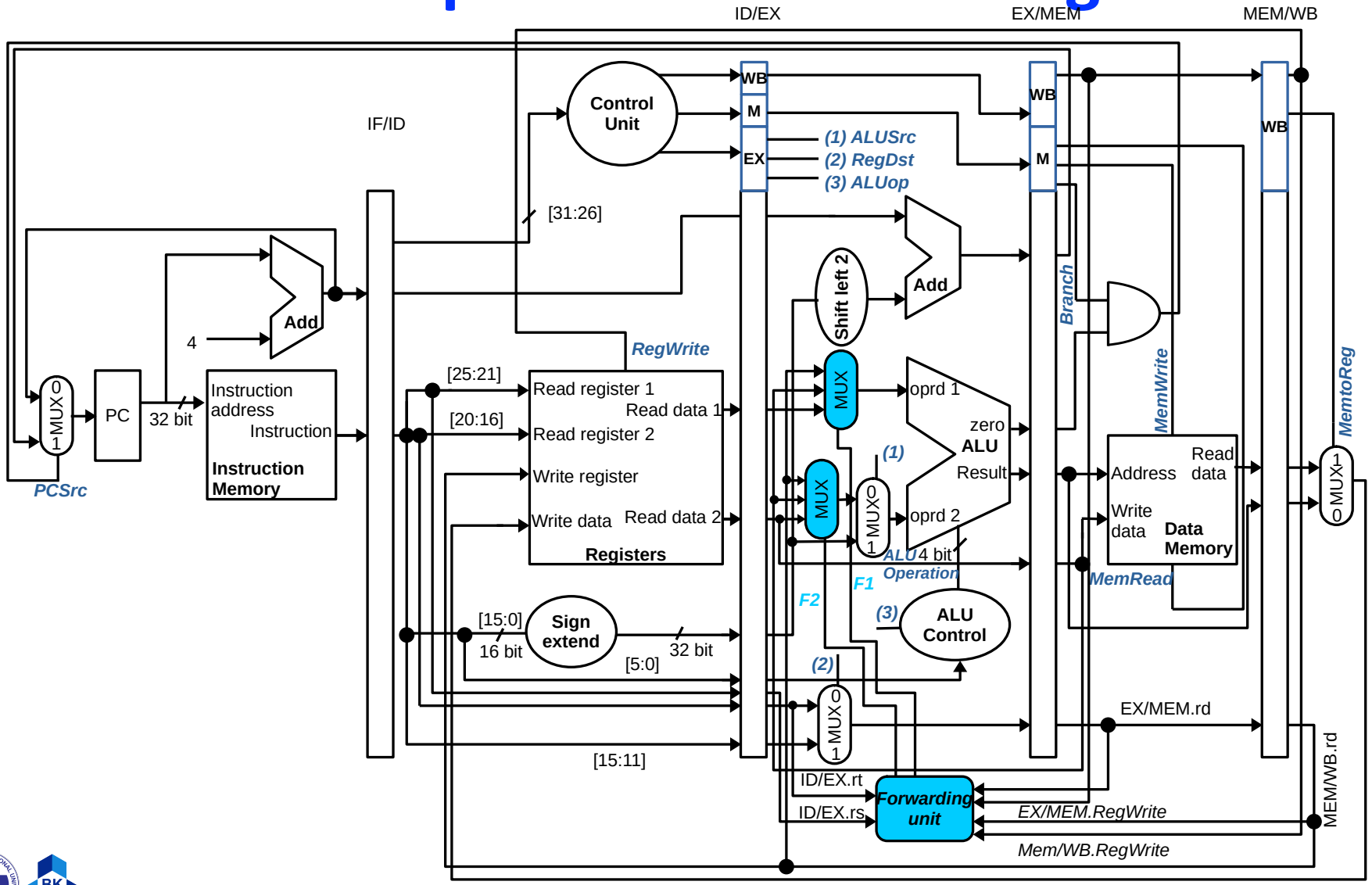
```
1. IF ID EX ME WB
2.    IF ID EX  ME  WB
3.       IF  ID   EX
```

- Both **EX** and **MEM hazards** seem occur
  - EX hazard is correct
- MEM hazard conditions

  2a. (ID/EX.rs = MEM/WB.rd) & (ID/EX.rs != EX/MEM.rd)

  2b. (ID/EX.rt = MEM/WB.rd) & (ID/EX.rt != EX/MEM.rd)
  (not happened when an **I-format** instruction is in the **EXE** stage)

Datapath with forwarding



Forwarding control values

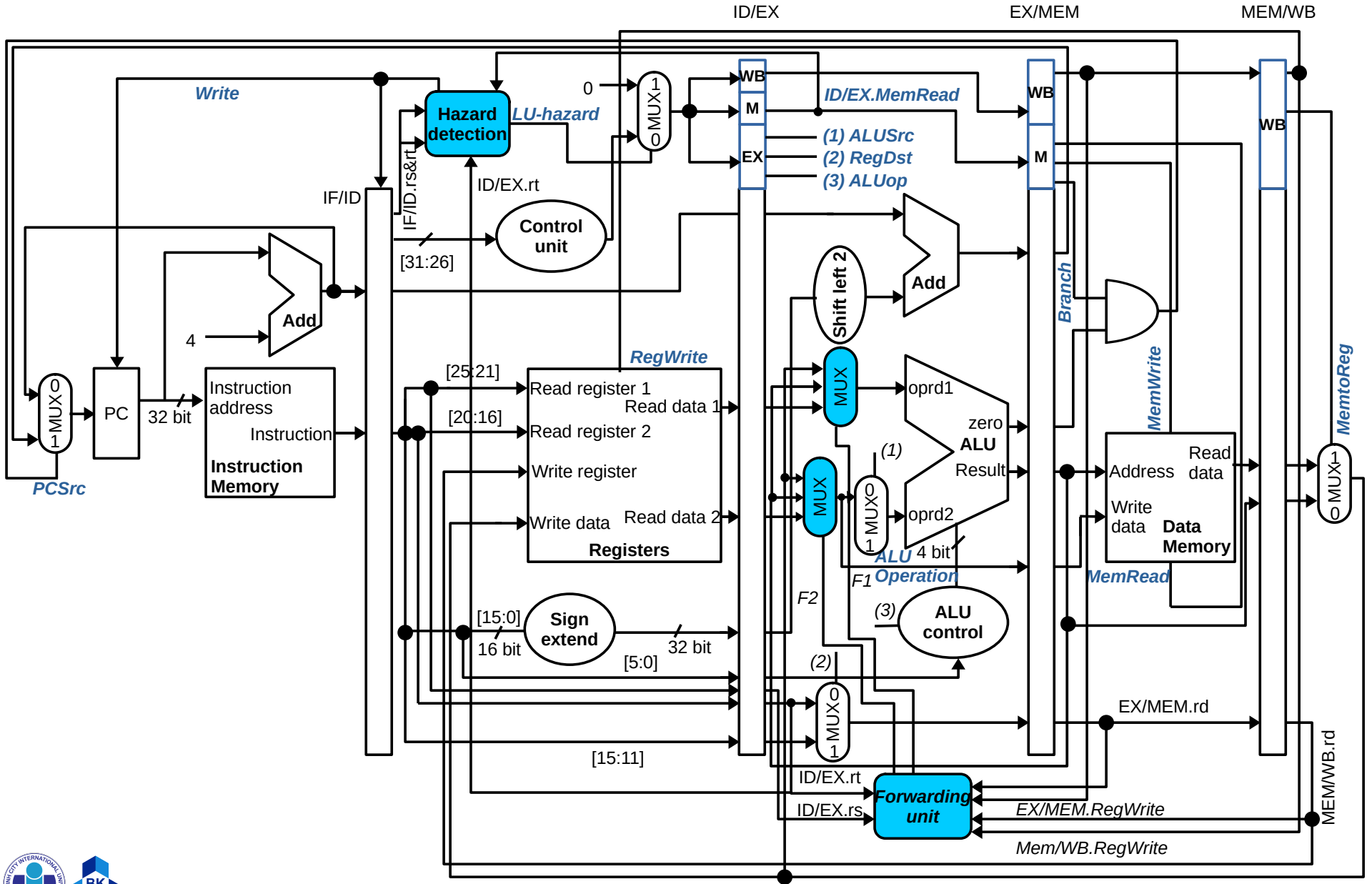
Mux values (binary)	Source	Explanation
F1 = 00	ID/EX	The first ALU operand comes from the registers file
F1 = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result
F1 = 01	MEM/WB	The first ALU operand is forwarded from data memory of an earlier ALU result
F2 = 00	ID/EX	The second ALU operand comes from the registers file
F2 = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result
F2 = 01	MEM/WB	The second ALU operand is forwarded from data memory of an earlier ALU result

Forwarding conditions

- **EX hazard**
  - **1a:** if (EX/MEM.**RegWrite** and (EX/MEM.**rd ≠ 0**) and (EX/MEM.rd = ID/EX.rs)) **F1 = 10**
  - **1b:** if (EX/MEM.**RegWrite** and (EX/MEM.**rd ≠ 0**) and (EX/MEM.rd = ID/EX.rt)) **F2 = 10**
- **MEM hazard**
  - **2a:** if (MEM/WB.RegWrite and (MEM/WB.rd ≠ 0) and **not** (EX/MEM.RegWrite and (EX/MEM.rd ≠ 0) and (EX/MEM.rd = ID/EX.rs)) and (MEM/WB.rd = ID/EX.rs)) **F1 = 01**
  - **2b:** if (MEM/WB.RegWrite and (MEM/WB.rd ≠ 0) and **not** (EX/MEM.RegWrite and (EX/MEM.rd ≠ 0) and (EX/MEM.rd = ID/EX.rt)) and (MEM/WB.rd = ID/EX.rt)) **F2 = 01**

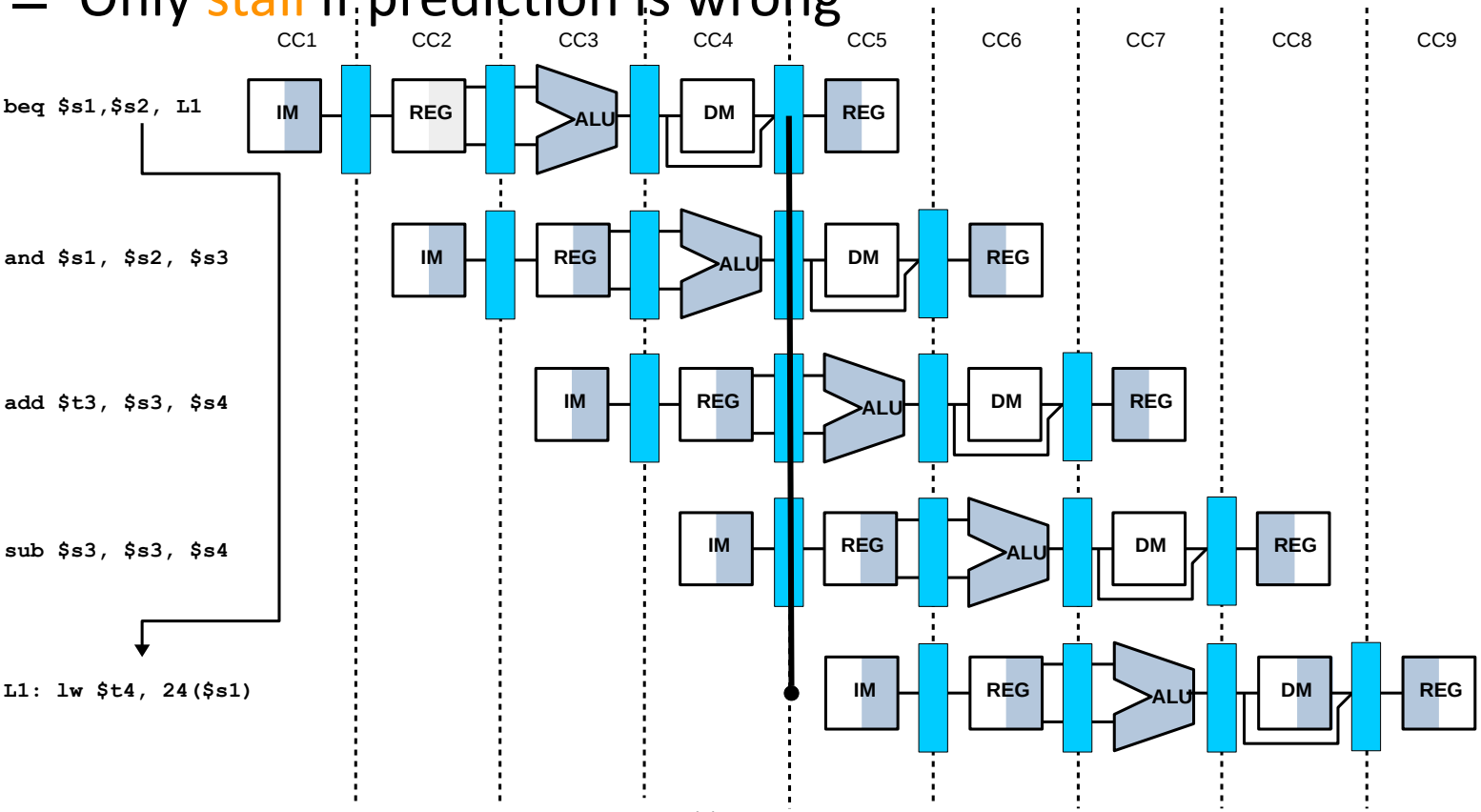
Detecting Load-use data hazards

- Check when using data instruction is decoded in **ID** stage
  - Producing data instruction (load) is in the **EXE** stage
  - The **sooner** the **better** due to a stall inserted
- Load-use hazard when
  - ID/EX.**MemRead** and ((ID/EX.**rt** = IF/ID.rs) or (ID/EX.**rt** = IF/ID.rt))
- **How to stall the pipeline?**
  - **Force** control signals in ID/EX register to 0 ⇒ EXE, MEM, and WB do nothing
  - **Prevent** updating PC and ID/EX register


ID/EX
EX/MEM
MEM/WB
Write
Hazard detection
LU-hazard
0
MUX
1
0
WB
M
EX
ID/EX.MemRead
(1) ALUSrc
(2) RegDst
(3) ALUop
IF/ID
IF/ID.rs&rt
ID/EX.rt
Control unit
[31:26]
Add
4
Shift left 2
Add
Branch
RegWrite
[25:21]
[20:16]
Read register 1
Read data 1
Read register 2
Write register
Write data
Read data 2
Registers
MUX
oprd1
zero
ALU
Result
oprd2
(1)
MUX
0
1
4 bit
ALU Operation
F1
F2
(3)
ALU control
MemWrite
MemRead
Address
Read data
Write data
Data Memory
MemtoReg
PC
32 bit
Instruction address
Instruction
Instruction Memory
PCSrc
[15:0]
16 bit
Sign extend
32 bit
[5:0]
(2)
[15:11]
EX/MEM.rd
MEM/WB.rd
ID/EX.rt
ID/EX.rs
Forwarding unit
EX/MEM.RegWrite
Mem/WB.RegWrite

Branch hazards solutions

- **Predict** outcome of branch
  - Only stall if prediction is wrong



Branch prediction

- **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

Concluding remarks

- ISA influences design of **datapath** and **control**
- Datapath and control influence design of ISA
- **Pipelining** improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- **Hazards**: structural, data, control

The end