Course on: “Advanced Computer Architectures”

Pipelining: Basic Concepts

Prof. Cristina Silvano
Politecnico di Milano
email: cristina.silvano@polimi.it

Outline

- Reduced Instruction Set of MIPS™ Processor
- Implementation of MIPS Processor
- Performance Optimization: Pipelining
- Implementation of MIPS Processor Pipeline
- The Problem of Pipeline Hazards
- Performance Issues in Pipelining

Main Characteristics of MIPS™ Architecture

- **RISC (Reduced Instruction Set Computer) Architecture**
  Based on the concept of executing only simple instructions in a reduced basic cycle to optimize the performance of CISC CPUs.
- **LOAD/STORE Architecture**

  ALU operands come from the CPU general purpose registers and they cannot directly come from the memory.
  Dedicated instructions are necessary to:
  - *load* data from memory to registers
  - *store* data from registers to memory
- **Pipeline Architecture:**

  Performance optimization technique based on the overlapping of the execution of multiple instructions derived from a sequential execution flow.

Reduced Instruction Set of MIPS™ Processor

- ALU instructions:

```
add $s1, $s2, $s3          # $s1 ← $s2 + $s3
addi $s1, $s1, 4           # $s1 ← $s1 + 4
```

- Load/store instructions:

```
lw $s1, offset ($s2)       # $s1 ← M[$s2+offset]
sw $s1, offset ($s2)       # M[$s2+offset] ← $s1
```

- Branch instructions to control the control flow of the program:
  - **Conditional branches:** the branch is taken only if the condition is satisfied. Examples: **beq** (*branch on equal*) and **bne** (*branch on not equal*)

```
beq $s1, $s2, L1    # go to L1 if ($s1 == $s2)
bne $s1, $s2, L1    # go to L1 if ($s1 != $s2)
```

  - **Unconditional jumps:** the branch is always taken. Examples: `j` *(jump)* and `jr` *(jump register)*

```
j   L1              # go to L1
jr  $s1             # go to add. contained in $s1
```

R-Format for Register-Register ALU Instructions

op	rs	rt	rd	shamt	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

- **`op:` (opcode)** identifies the ALU instruction type;
- **`rs:`** first source operand
- **`rt:`** second source operand
- **`rd:`** destination register
- **`shamt:`** shift amount
- **`funct:`** identifies the different type of ALU instructions

I-Format for Immediate ALU Instructions

op	rs	rt	immediate
6 bit	5 bit	5 bit	16 bit

- **`op (opcode):`** identifies immediate instruction type;
- **`rs:`** source register;
- **`rt:`** destination register;
- **`immediate:`** contains the value of the immediate operand (in the range -2^{15} $+2^{15}-1$).

I-Format for Load/Store Instructions

op	rs	rt	offset
6 bit	5 bit	5 bit	16 bit

- **`op (opcode):`** identifies the load/store instruction type;
- **`rs:`** base register;
- **`rt:`** destination/source register for the data loaded/stored from/to memory;
- **`offset:`** The sum – called **effective address** – of the contents of the base register and the sign-extended offset is used as memory address.

I-Format for Conditional Branches

op	rs	rt	address
6 bit	5 bit	5 bit	16 bit

- **op (opcode):** identifies the conditional branch instruction type;
- **`rs:`** first source register to compare;
- **`rt:`** second source register to compare;
- **`address`** (16-bit) indicates the word offset relative to the PC **(PC-relative word address)**
- The offset corresponding to the L1 label (Branch Target Address) is relative to the Program Counter (PC):
  (L1- PC) /4

```
beq $s1, $s2, L1
```

J-Format for Unconditional Jumps

op	address

6 bit 26 bit

- **op (opcode):** identifies the jump instruction type;
- **`address`:** contains 26-bit of 32-bit absolute word address of jump destination:

4 bit	26 bit	2 bit
PC+4 [31-28]	address [25-0]	00

Formats of MIPS 32-bit Instructions

- **Type R (Register)**
  - ALU Instructions
- **Type I (Immediate)**
  - Immediate Instructions
  - Load/store instructions
  - Conditional branch instructions
- **Tipo J (jump)**
  - Unconditional jumps instructions

	6-bit	5-bit	5-bit	5-bit	5-bit	6-bit
	31 26	25 21	20 16	15 11	10 6	5 0
R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	offset/immediate		
J	op	address				

Phases of execution of MIPS Instructions

Every instruction in the MIPS subset can be implemented in **at most 5 clock cycles (phases)** as follows:

1) Instruction Fetch (IF):

- Send the content of **Program Counter** register to **Instruction Memory** and fetch the current instruction from Instruction Memory.
  Update the PC to the next sequential address by adding 4 to the PC (since each instruction is 4 bytes).

2) Instruction Decode and Register Read (ID):

- Decode the current instruction **(fixed-field decoding)** and read from the Register File of one or two registers corresponding to the registers specified in the instruction fields.
- Sign-extension of the offset field of the instruction in case it is needed.

Phases of execution of MIPS Instructions

3) Execution (EX):

The ALU operates on the operands prepared in the previous cycle depending on the instruction type:

- Register-Register ALU Instructions:
  - ALU executes the specified operation on the operands read from the RF
- Register-Immediate ALU Instructions:
  - ALU executes the specified operation on the first operand read from the RF and the sign-extended immediate operand
- Memory Reference:
  - ALU adds the base register and the offset to calculate the **effective address.**
- Conditional branches:
  - Compare the two registers read from RF and compute the possible **branch target address** by adding the sign-extended offset to the incremented PC.

Phases of execution of MIPS Instructions

- **Memory Access (ME)**
  - *Load* instructions require a read access to the Data Memory using the effective address
  - *Store* instructions require a write access to the Data Memory using the effective address to write the data from the source register read from the RF
  - Conditional branches can update the content of the PC with the branch target address, if the conditional test yielded true.
- **Write-Back Cycle (WB)**
  - Load instructions write the data read form memory in the destination register of the RF
  - ALU instructions write the ALU results into the destination register of the RF.

Phases of execution of MIPS Instructions

ALU Instructions: `op $x,$y,$z`

Instr. Fetch &. PC Increm.	Read of Source Regs. `$y` and `$z`	ALU OP (`$y op $z`)	Write Back of Destinat. Reg. `$x`

Load Instructions: `lw $x,offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y`	ALU Op. (`$y+offset`)	Read Mem. `M($y+offset)`	Write Back of Destinat. Reg. `$x`

Store Instructions: `sw $x,offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y` & Source `$x`	ALU Op. (`$y+offset`)	Write Mem. `M($y+offset)`

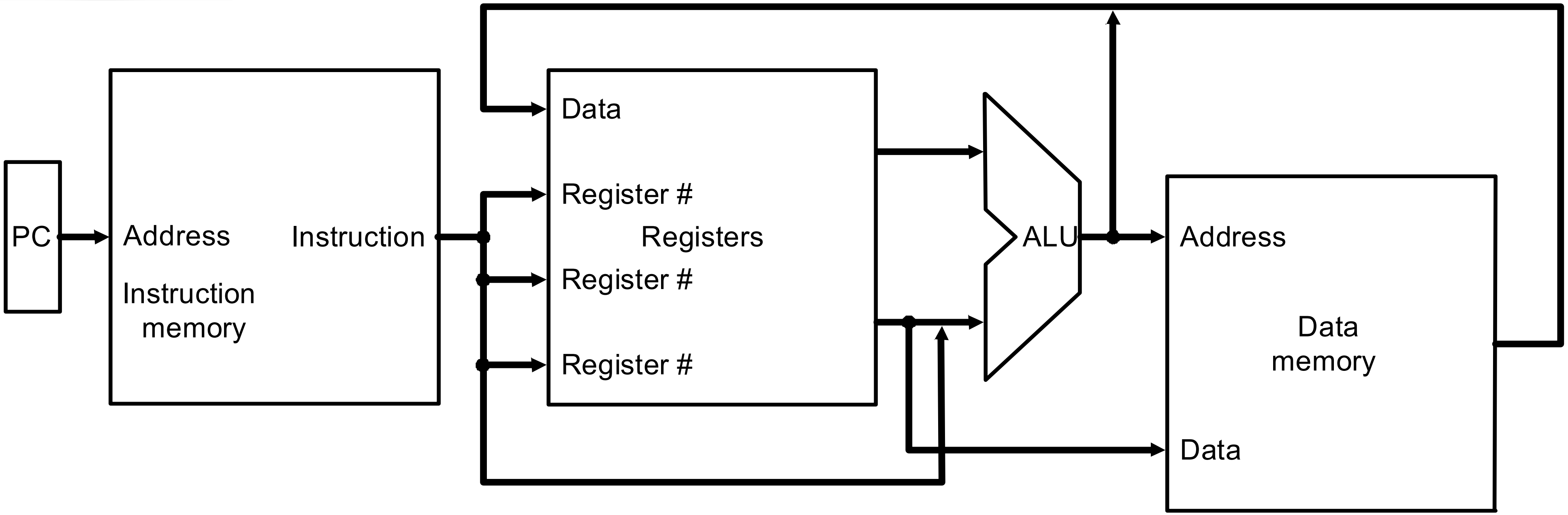
Conditional Branch: `beq $x,$y,offset`

Instr. Fetch & PC Increm.	Read of Source Regs. `$x` and `$y`	ALU Op. (`$x-$y`) & (`PC+4+offset`)	Write PC

Instruction Latency

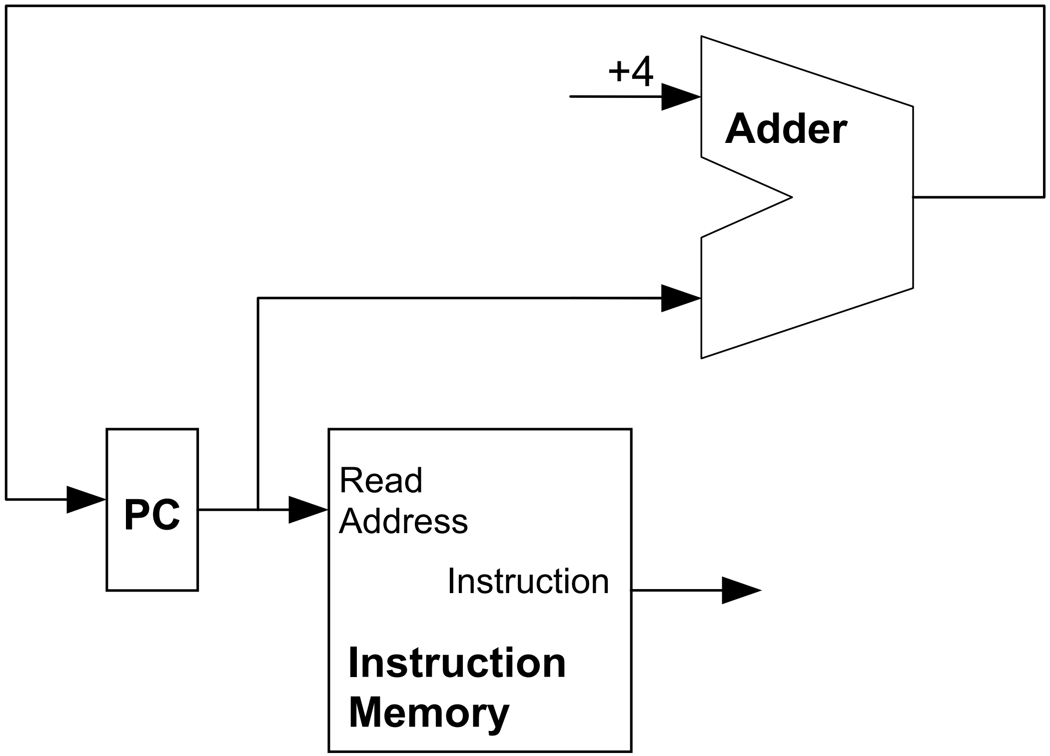
Instruction Type	Instruct. Mem.	Register Read	ALU Op.	Data Memory	Write Back	Total Latency
ALU Instr.	2	1	2	0	1	6 ns
Load	2	1	2	2	1	8 ns
Store	2	1	2	2	0	7 ns
Cond. Branch	2	1	2	0	0	5 ns
Jump	2	0	0	0	0	2 ns

Basic Implementation of MIPS data path

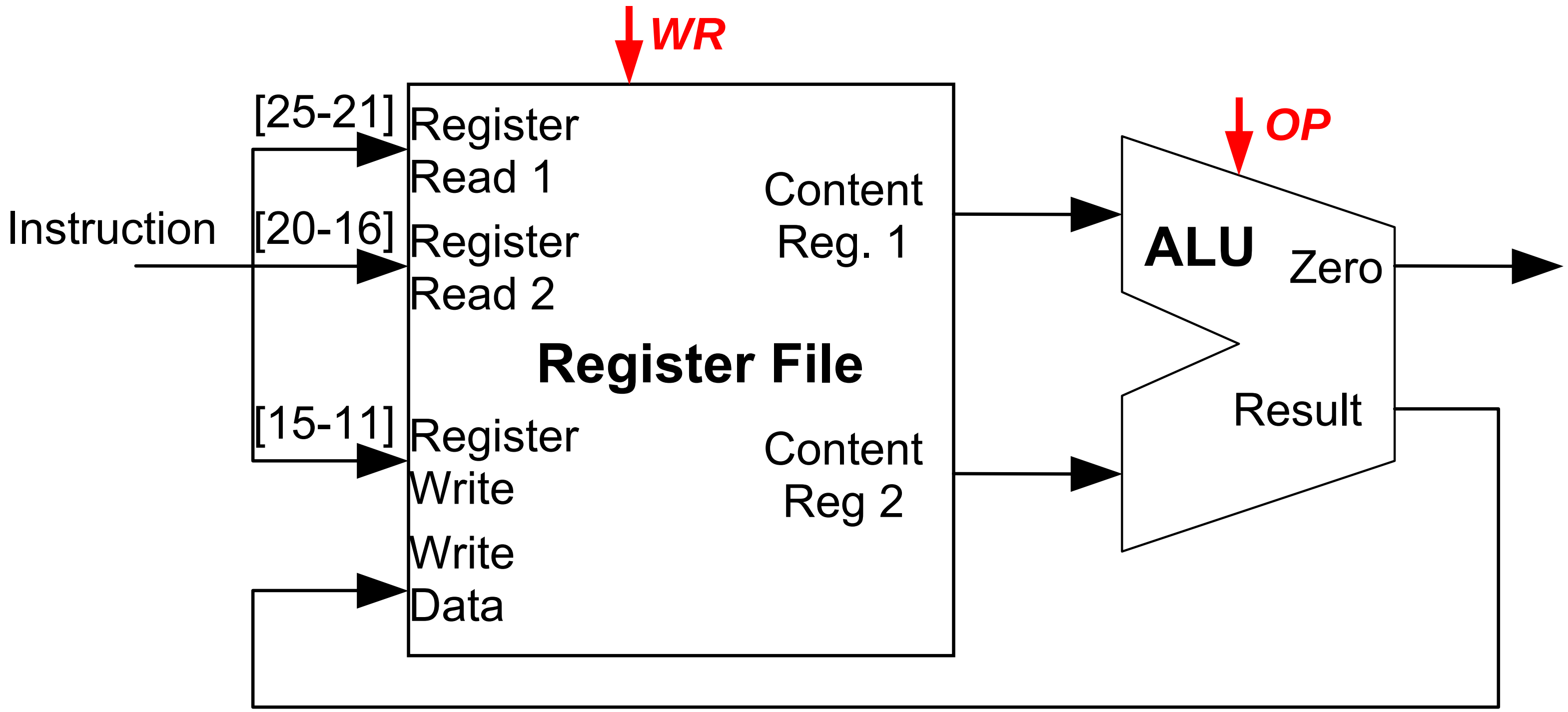


- **Instruction Memory** (read-only memory) separated from **Data Memory**
- 32 General-Purpose Registers organized in a **Register File** (RF) with 2 read ports and 1 write port.

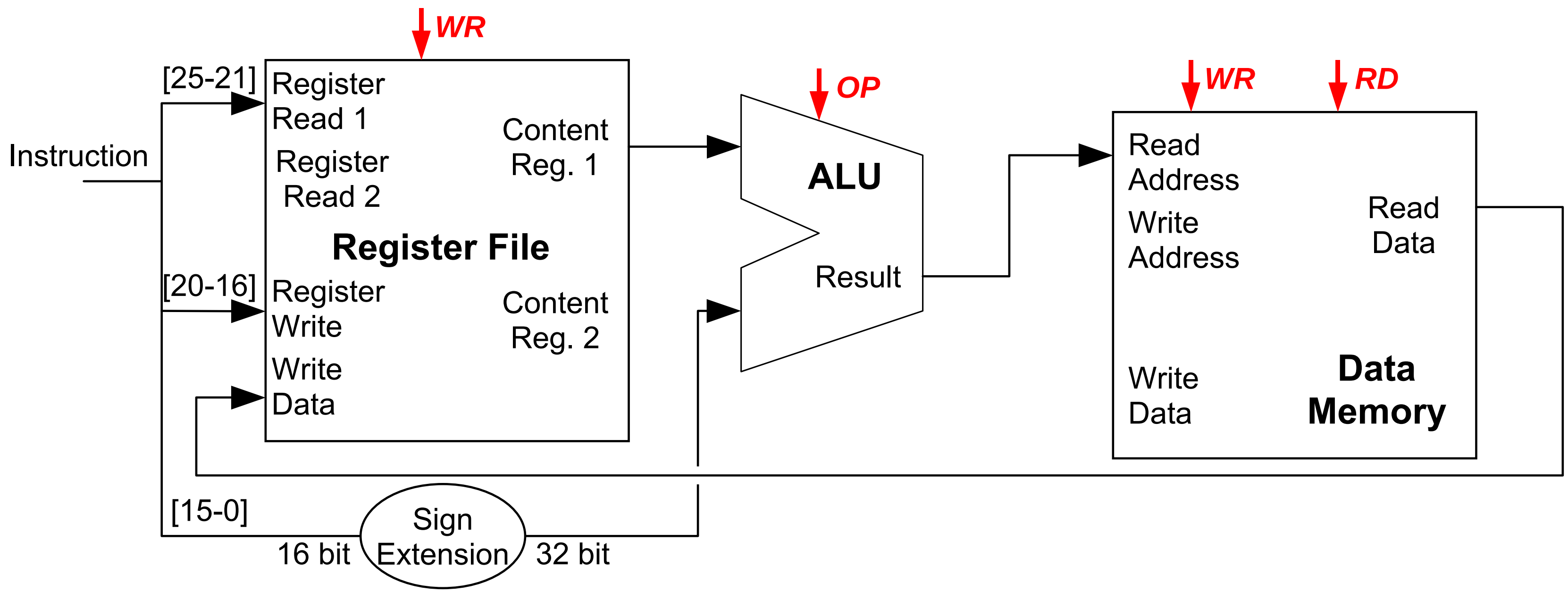
Implementation of **Instruction Fetch** phase



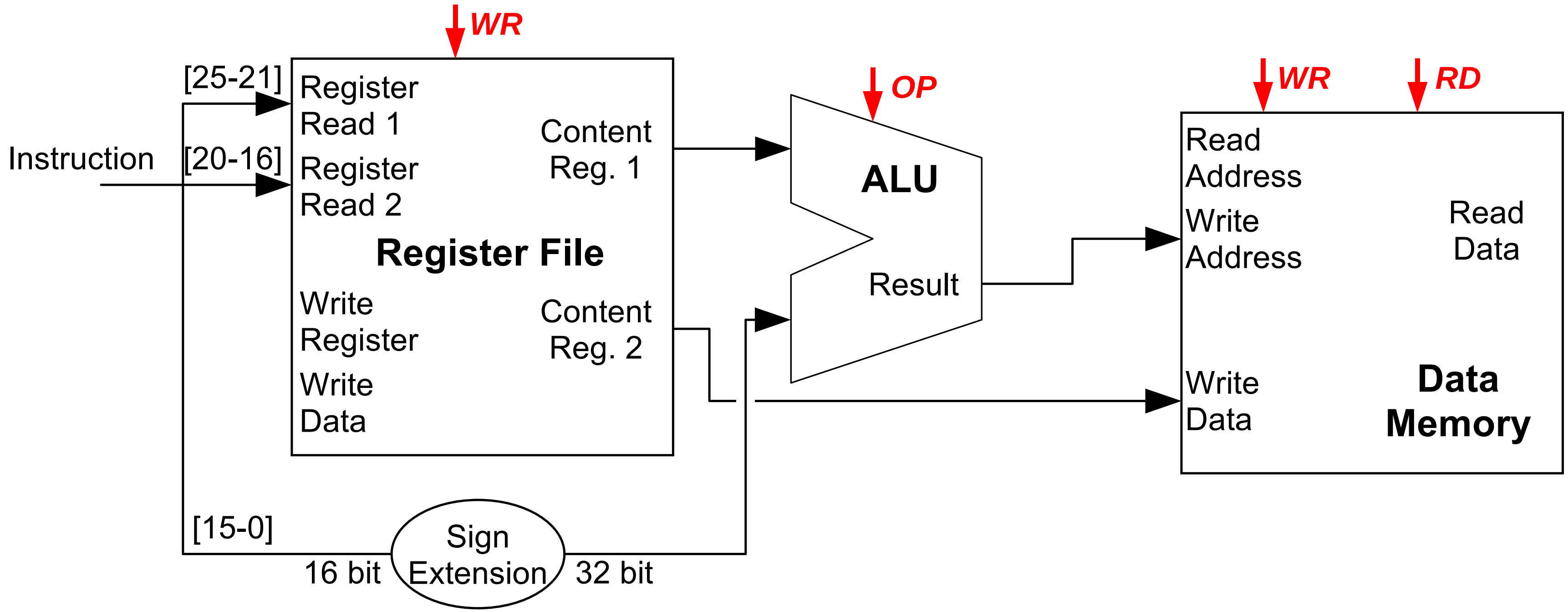
Implementation of ALU Instructions



Implementation of Load Instructions



Implementation of Store Instructions



Single-cycle Implementation of MIPS

- The length of the clock cycle is defined by the critical path given by the load instruction: *T = 8 ns (f = 125 MHz).*
- We assume each instruction is executed in a **single clock cycle**
  - Each module must be used once in a clock
  - The modules used more than once in a cycle must be duplicated.
- We need an Instruction Memory separated from the Data Memory.
- Some modules must be duplicated, while other modules must be shared from different instruction flows
- To share a module between two different instructions, we need a **multiplexer** to enable multiple inputs to a module and select one of different inputs based on the configuration of control lines.

Implementation of ALU and Load/Store Instructions

Main differences between ALU and load/store instructions:

- Different number of Write Register in the RF ([15-11] bit for ALU instructions or [20-16] bit for load/store instructions)

  ⇒ **MUX at the input of Write Register of the RF**

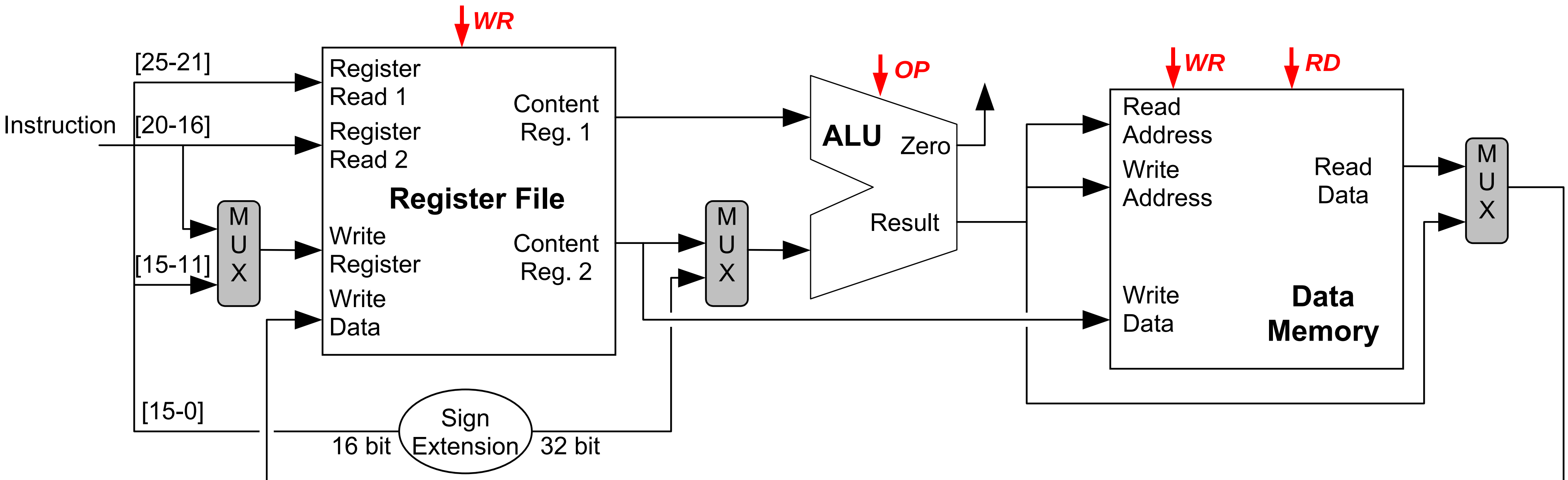
- The second input of ALU is a register for ALU instructions or the least significant part of the instruction for load/store instructions

  ⇒ **MUX at the second input of the ALU;**

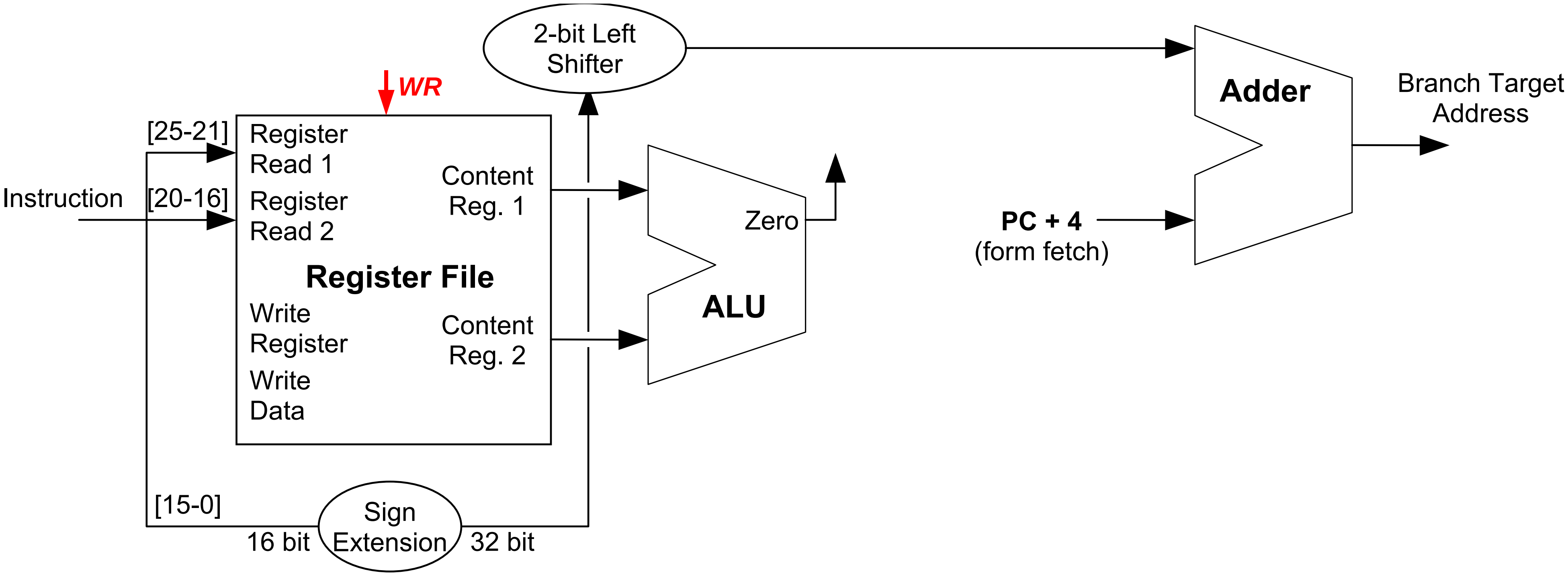
- Write data in the destination register can come from the ALU result for ALU instructions or from the Data Memory for load instructions

  ⇒ **MUX at the write data input of the RF.**

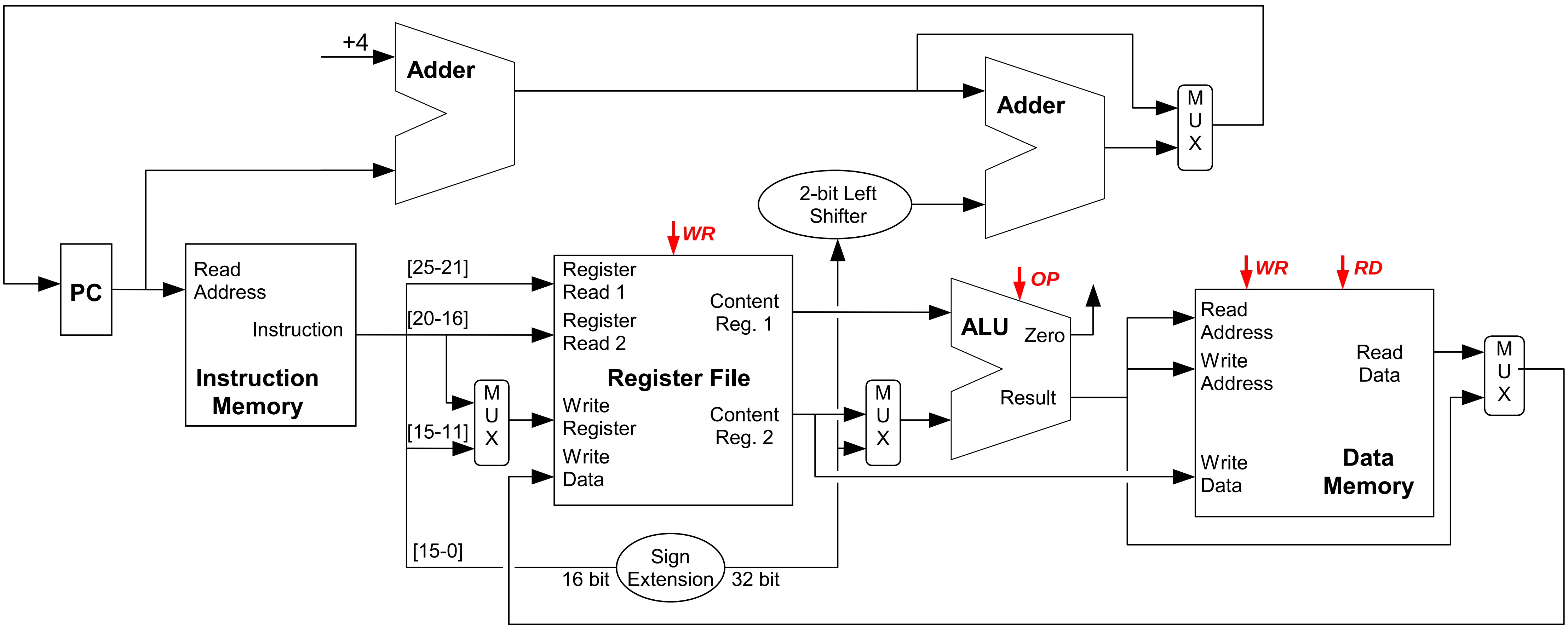
Implementation of ALU and Load/Store Instructions



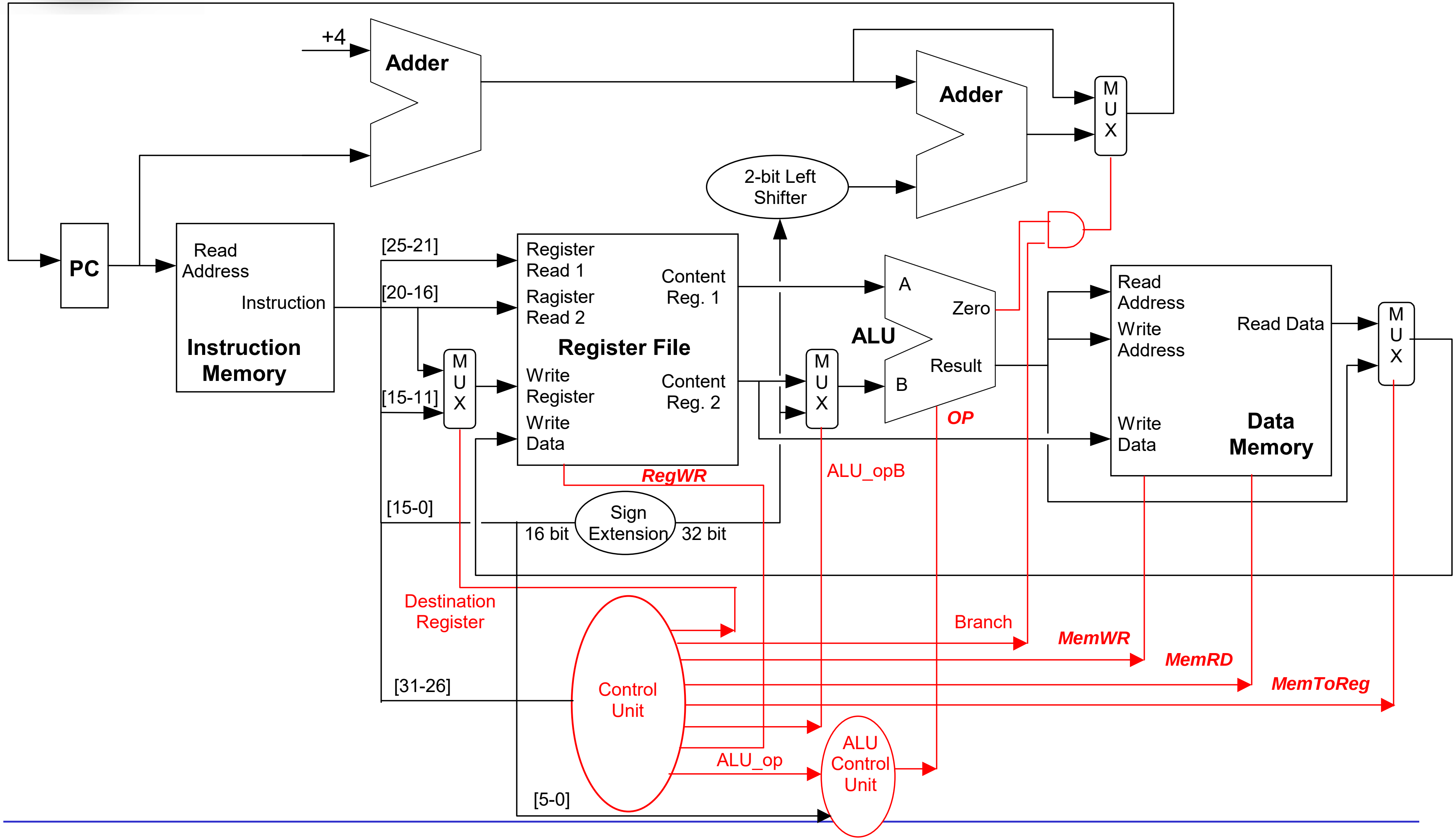
Implementation of Conditional Branch Instructions



Implementation of MIPS data path



Implementation of MIPS data path with Control Unit



Multi-cycle Implementation

- The instruction execution is distributed on multiple cycles (5 cycles for MIPS)
- The basic cycle is smaller
(2 ns $\Rightarrow$ instruction latency = 10 ns)
- Implementation of multi-cycle CPU:
  - Each phase of the instruction execution requires a clock cycle
  - Each module can be used more than once per instruction in different clock cycles: possible sharing of modules
  - We need internal registers to store the values to be used in the next clock cycles.

PIPELINING

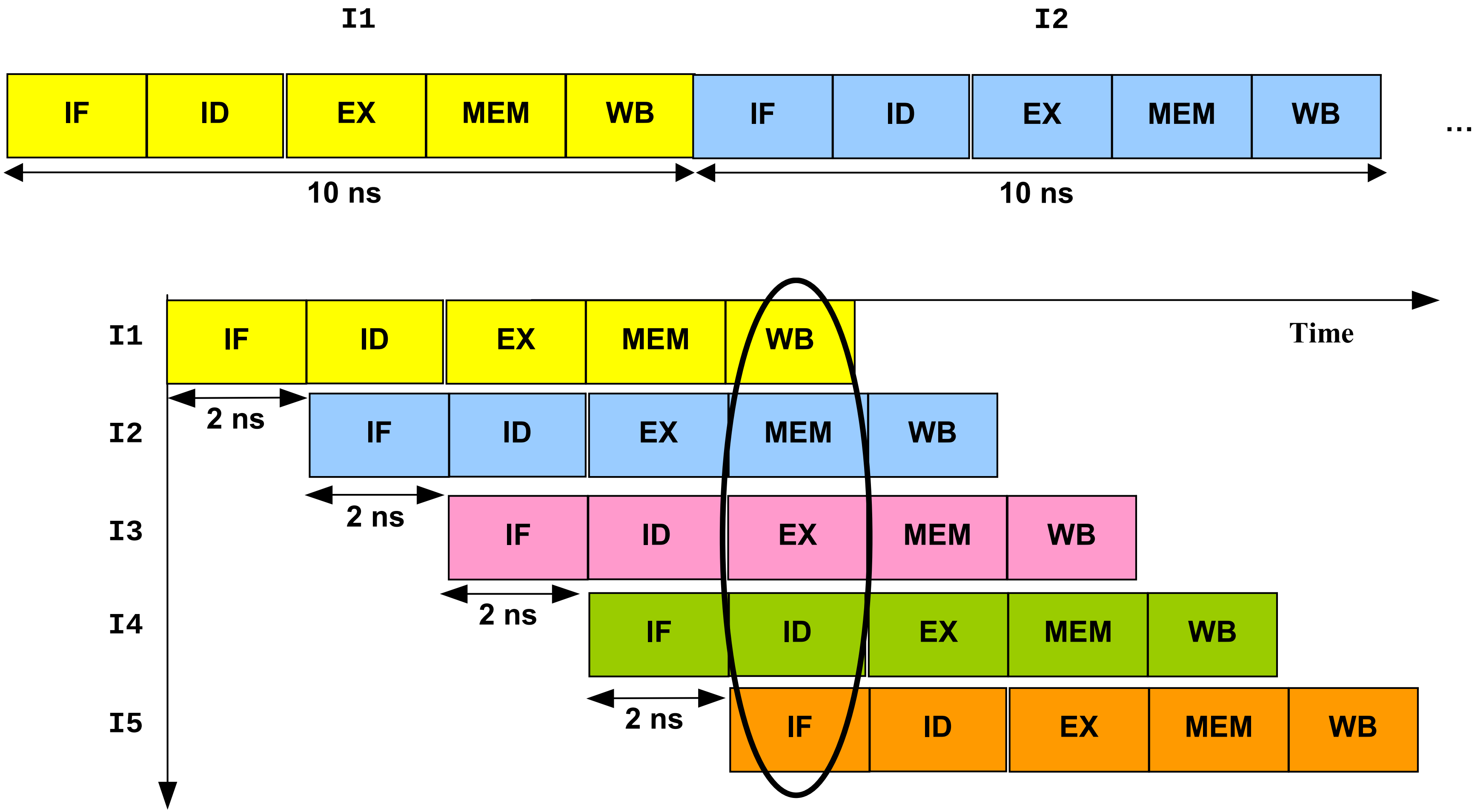
Pipelining

- Performance optimization technique based on the overlap of the execution of multiple instructions deriving from a sequential execution flow.
- Pipelining exploits the parallelism among instructions in a sequential instruction stream.
- **Basic idea:**
  The execution of an instruction is divided into different phases **(pipelines stages),** requiring a fraction of the time necessary to complete the instruction.
- The stages are connected one to the next to form the pipeline: instructions enter in the pipeline at one end, progress through the stages, and exit from the other end, as in an assembly line.

Pipelining

- Advantage: technique transparent for the programmer.
- Technique similar to a assembly line: a new car exits from the assembly line in the time necessary to complete one of the phases.
- An assembly line does not reduce the time necessary to complete a car, but increases the number of cars produced simultaneously and the frequency to complete cars.

Sequential vs. Pipelining Execution



Pipelining

- The time to advance the instruction of one stage in the pipeline corresponds to a clock cycle.
- The pipeline stages must be **synchronized:** the duration of a clock cycle is defined by the time requested by the slower stage of the pipeline *(i.e. 2 ns).*
- The goal is to **balance** the length of each pipeline stage
- If the stages are perfectly balanced, the **ideal speedup** due to pipelining is equal to the number of pipeline stages.

Performance Improvement

- Ideal case (asymptotically):If we consider the single-cycle unpipelined *CPU1* with clock cycle of *8 ns* and the pipelined *CPU2* with 5 stages of 2 ns :
  - The **latency** (total execution time) of each instruction is worsened: from *8 ns* to *10 ns*
  - The **throughput** (number of instructions completed in the time unit) is improved of **4 times:**
    (1 instruction completed each *8 ns)* vs.
    (1 instruction completed each *2 ns)*

Performance Improvement

- Ideal case (asymptotically): If we consider the multi-cycle unpipelined *CPU3* composed of 5 cycles of *2 ns* and the pipelined *CPU2* with 5 stages of 2 ns :
  - The **latency** (total execution time) of each instruction is not varied *(10 ns)*
  - The **throughput** (number of instructions completed in the time unit) is improved of *5* **times:**
    (1 instruction completed every *10 ns) vs.*
    (1 instruction completed every *2 ns)*

Pipeline Execution of MIPS Instructions

IF Instruction Fetch	ID Instruction Decode	EX Execution	ME Memory Access	WB Write Back

Pipeline Execution of MIPS Instructions

IF Instruction Fetch	ID Instruction Decode	EX Execution	ME Memory Access	WB Write Back

ALU Instructions: `op $x,$y,$z`

Instr. Fetch & PC Increm.	Read of Source Regs. `$y` and `$z`	ALU Op. (`$y op $z`)		Write Back Destinat. Reg. `$x`

Load Instructions: `lw $x,offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y`	ALU Op. (`$y+offset`)	Read Mem. `M($y+offset)`	Write Back Destinat. Reg. `$x`

Store Instructions: `sw $x,offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y` & Source `$x`	ALU Op. (`$y+offset`)	Write Mem. `M($y+offset)`	

Conditional Branches: `beq $x,$y,offset`

Instr. Fetch & PC Increm.	Read of Source Regs. `$x` and `$y`	ALU Op. (`$x-$y`) & (`PC+4+offset`)	Write PC	

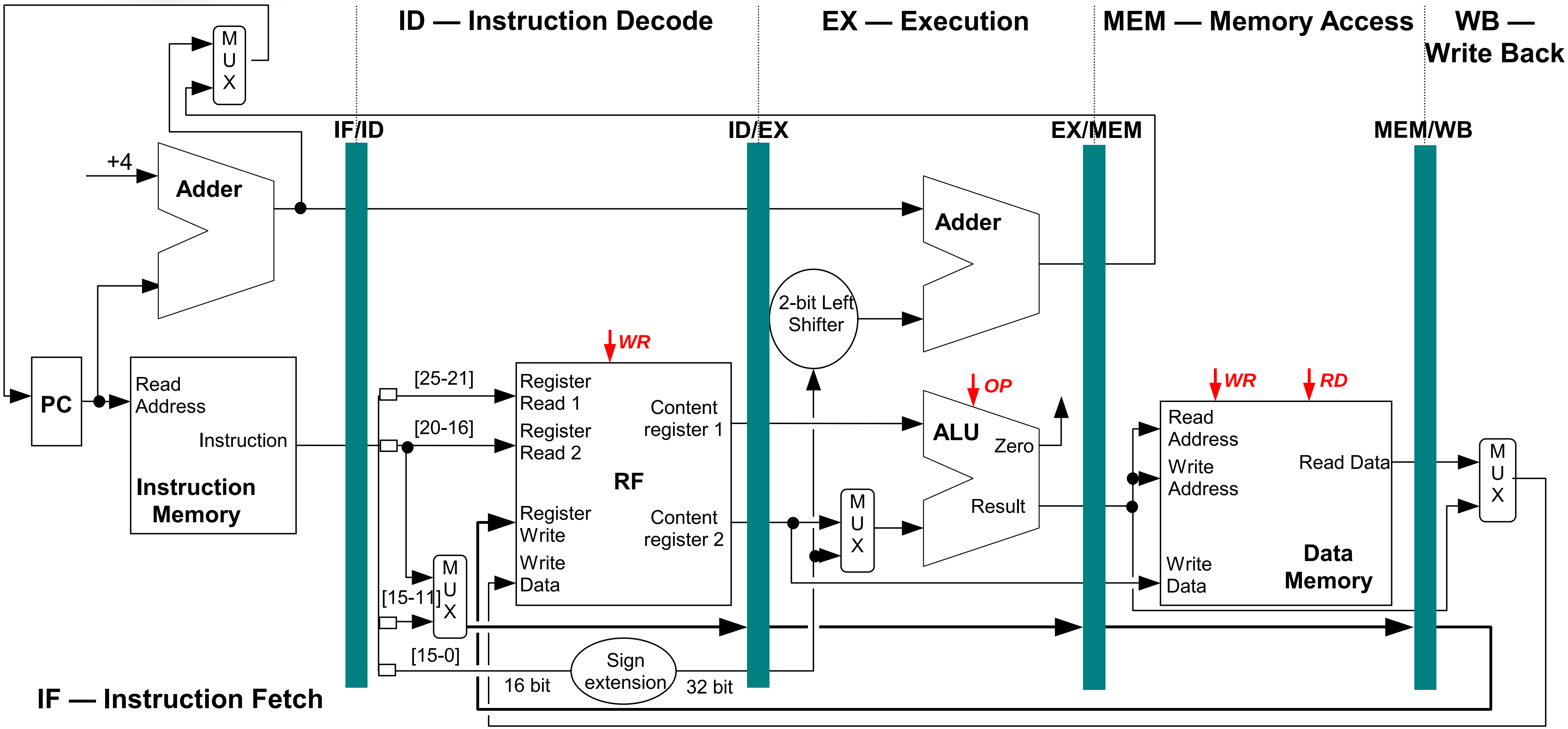
Implementation of MIPS pipeline

- The division of the execution of each instruction in 5 stages implies that in each clock cycle 5 instructions are in execution

  $\Rightarrow$ the implementation of pipelined *CPU* with 5 stages must be composed of 5 modules corresponding to 5 execution stages

  $\Rightarrow$ we need **pipeline registers** to separate the different stages

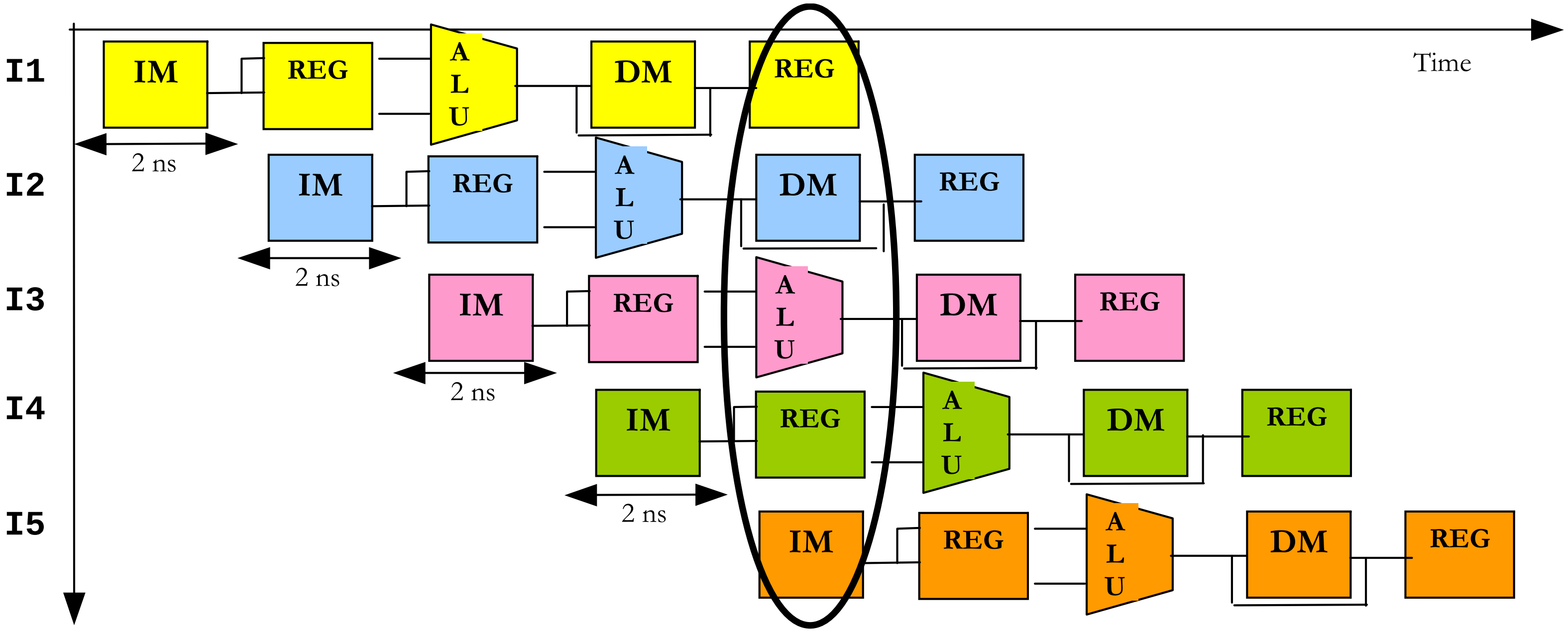
Implementation of MIPS pipeline



Implementation of MIPS pipeline

- We need to pass throughout the pipeline registers (*ID/EX*, *EX/MEM* and *MEM/WB*), the address of the register to write during the WB stage.
- The address of the write register pass throughout the pipeline registers and then come from the MEM/WB pipeline register with the write data

Resources used during the pipeline execution



IM = Instruction Memory

REG = Register File

DM = Data Memory

The Problem of Pipeline Hazards

- A **hazard (conflict)** is created whenever there is a dependence between instructions, and instructions are close enough that the overlap caused by pipelining would change the order of access to the operands involved in the dependence.
- Hazards prevent the next instruction in the pipeline from executing during its designated clock cycle.
- Hazards reduce the performance from the ideal speedup gained by pipelining.

Three Classes of Hazards

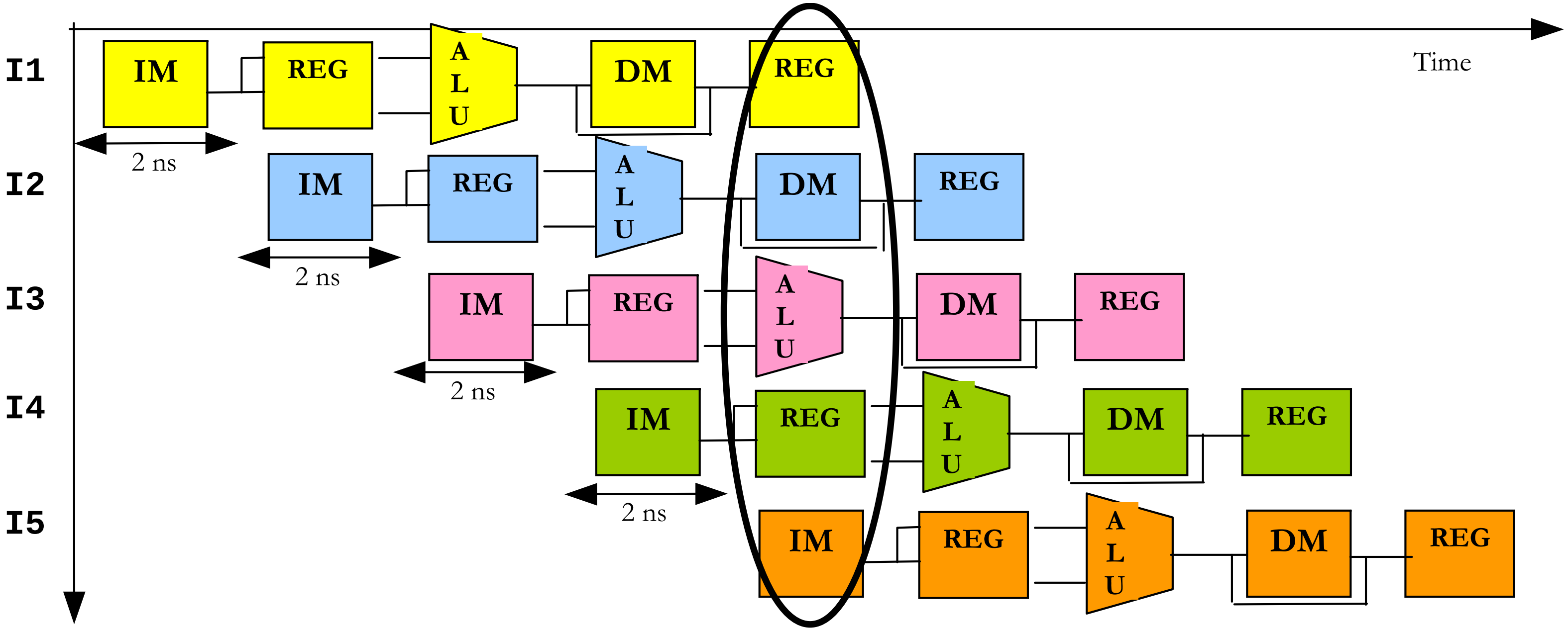
1) **Structural Hazards:** Attempt to use the same resource from different instructions simultaneously
   - Example: Single memory for instructions and data

2) **Data Hazards:** Attempt to use a result before it is ready
   - Example: Instruction depending on a result of a previous instruction still in the pipeline

3) **Control Hazards:** Attempt to make a decision on the next instruction to execute before the condition is evaluated
   - Example: Conditional branch execution

Structural Hazards

- No structural hazards in MIPS architecture:
  - Instruction Memory separated from Data Memory
  - Register File used in the same clock cycle: Read access by an instruction and write access by another instruction



Data Hazards

- If the instruction executed in the pipeline are **dependent,** data hazards can arise when instructions are too close
- Example:

```
sub  $2, $1, $3   # Reg. $2 written by sub
and $12, $2, $5   # 1° operand ($2) depends on sub
or $13, $6, $2    # 2° operand ($2) depend on sub
add $14, $2, $2   # 1° ($2) & 2° ($2) depend on sub
sw $15,100($2)    # Base reg. ($2) depends on sub
```

Data Hazards: Example

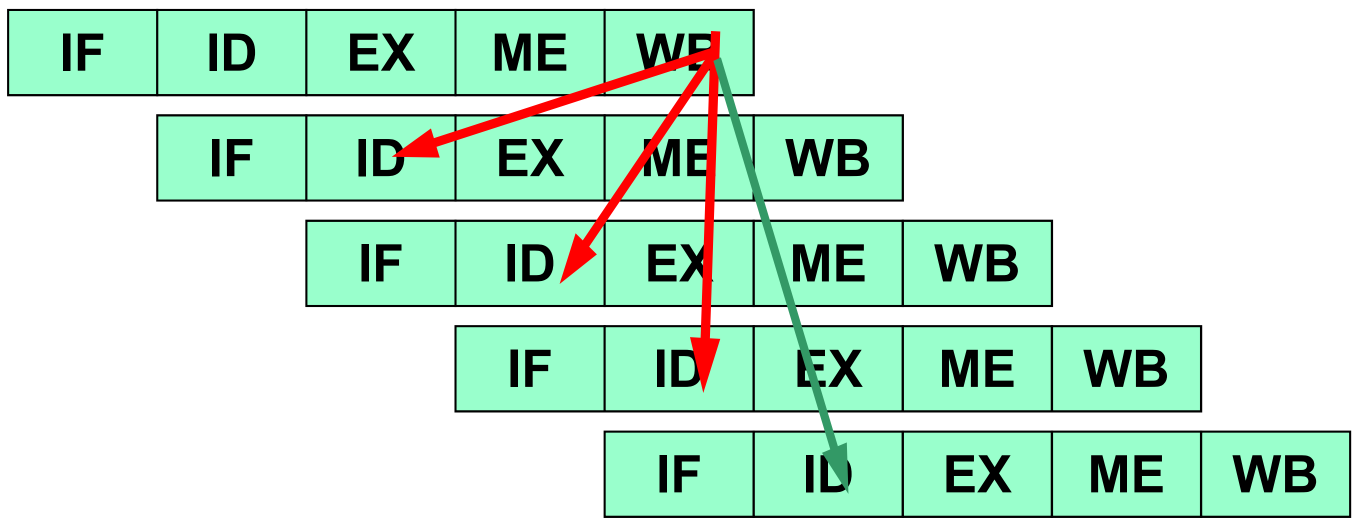
```
sub  $2, $1, $3

and $12, $2, $5

or  $13, $6, $2

add $14, $2, $2

sw  $15,100($2)
```



Data Hazards: Possible Solutions

- **Compilation Techniques:**
  a) Insertion of **nop** (*no operation)* instructions

  b) Instructions scheduling to avoid that correlating instructions are too close
    - The compiler tries to insert independent instructions among correlating instructions
    - When the compiler does not find independent instructions, it insert **nops.**
- **Hardware Techniques:**
  c) Insertion of *stalls or "bubbles"* in the pipeline

  d) Data forwarding or bypassing

a) Insertion of **nops**: Example

`sub $2, $1, $3`	IF	ID	EX	ME	WB							
`nop`		IF	ID	EX	ME	WB						
`nop`			IF	ID	EX	ME	WB					
`nop`				IF	ID	EX	ME	WB				
`and $12, $2, $5`					IF	ID	EX	ME	WB			
`or $13, $6, $2`						IF	ID	EX	ME	WB		
`add $14, $2, $2`							IF	ID	EX	ME	WB	
`sw $15,100($2)`								IF	ID	EX	ME	WB

b) Scheduling: Example

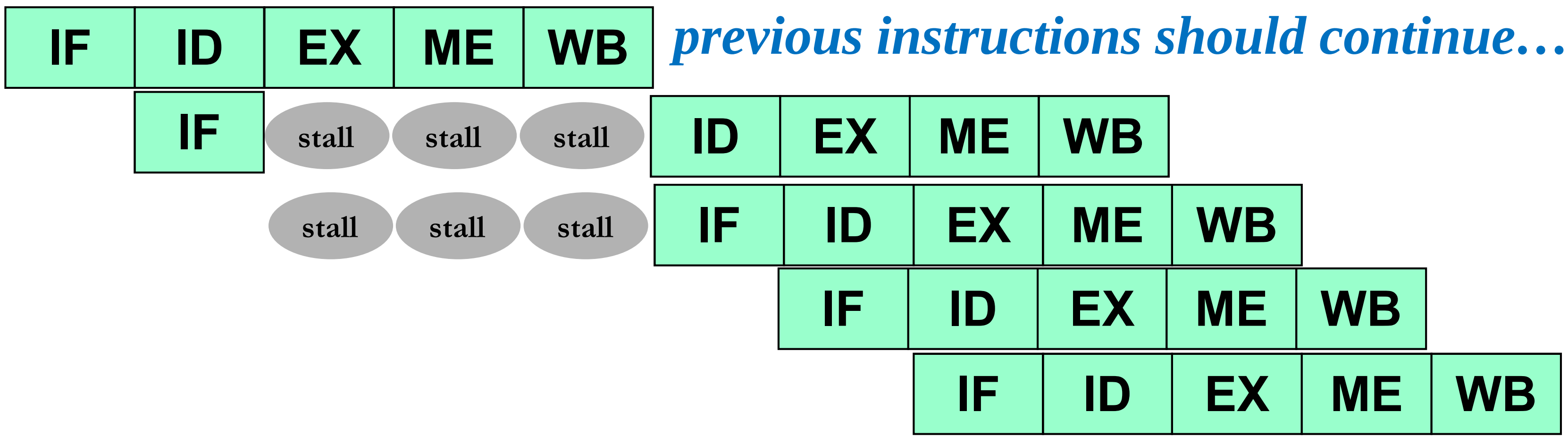
- Example:

```
sub  $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15,100($2)
add $4, $10, $11
and $7, $8, $9
lw $16, 100($18)
```

```
sub     $2, $1, $3
add $4, $10, $11
and $7, $8, $9
lw $16, 100($18)
and    $12, $2, $5
or $13, $6, $2
add    $14, $2, $2
sw $15,100($2)
```

c) Insertion of Stalls: Example

```
sub  $2, $1, $3
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15,100($2)
```

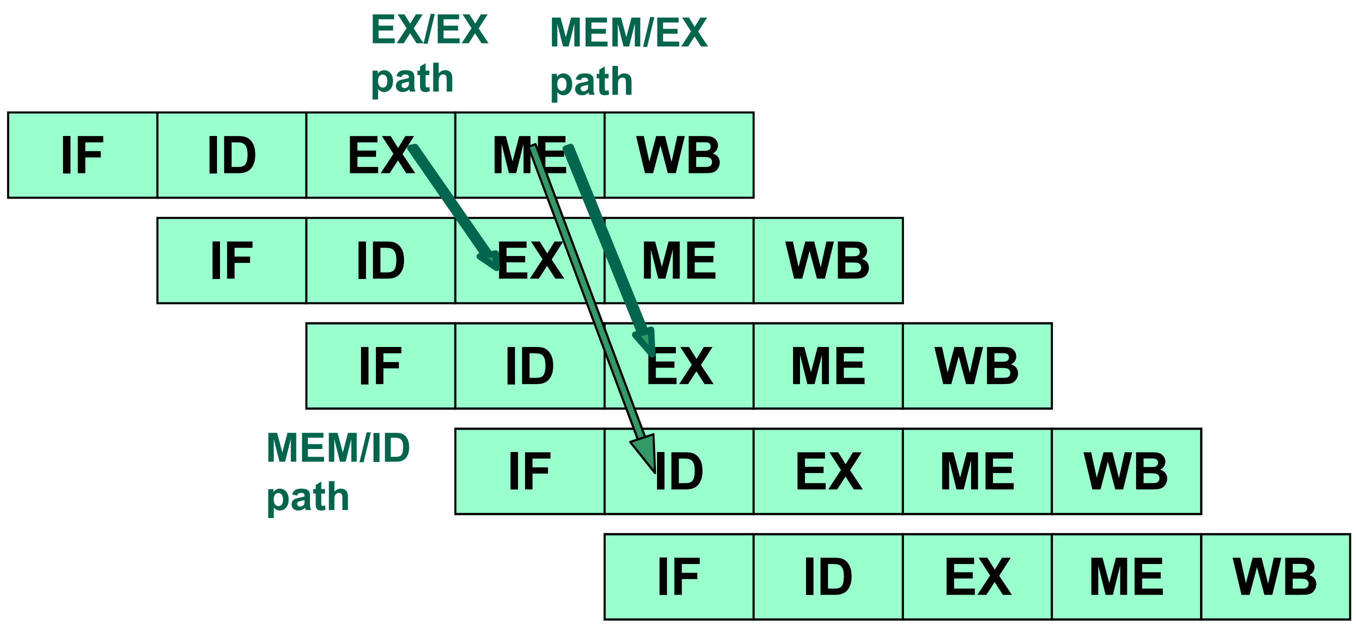


d) Forwarding

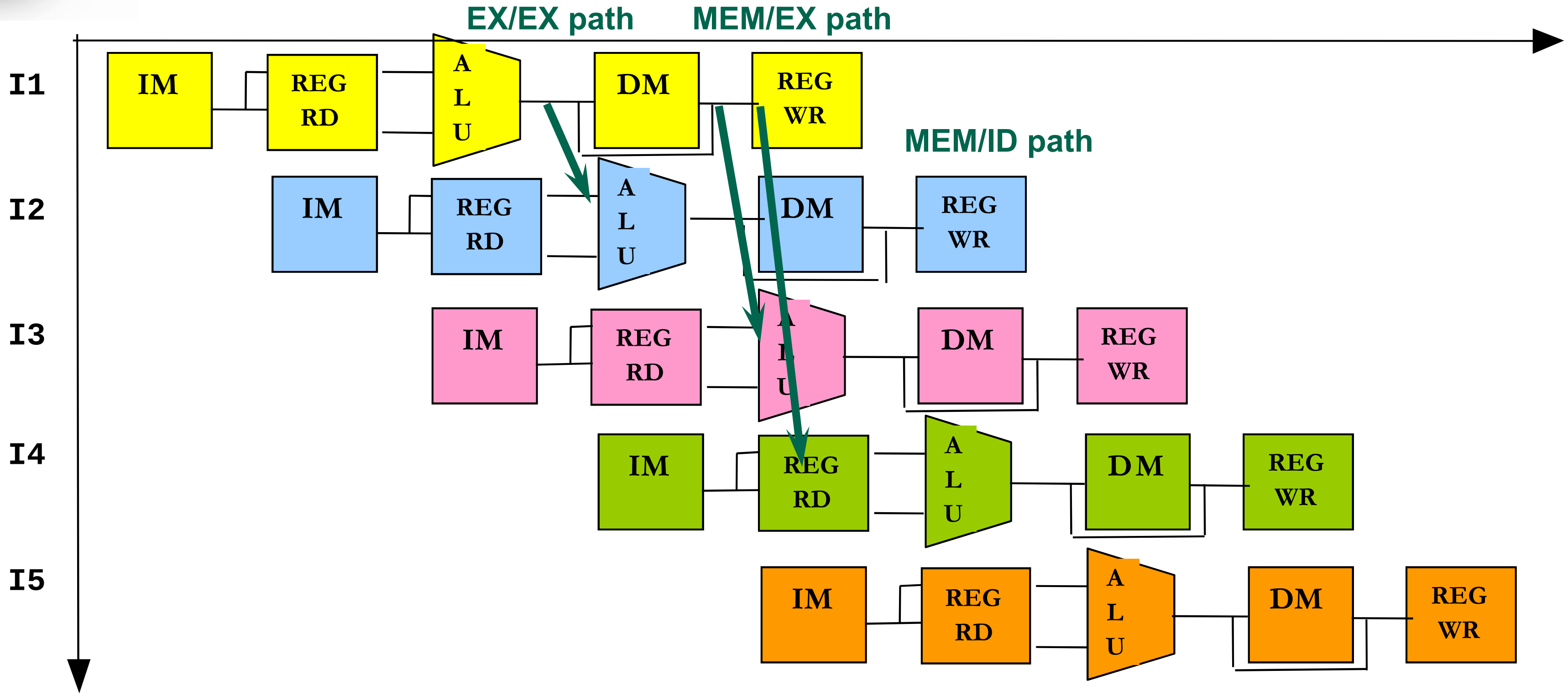
- Data forwarding uses temporary results stored in the pipeline registers instead of waiting for the write back of results in the RF.
- We need to add **multiplexers** at the inputs of ALU to fetch inputs from pipeline registers to avoid the insertion of stalls in the pipeline.

Forwarding: Example

```
sub  $2, $1, $3
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15,100($2)
```

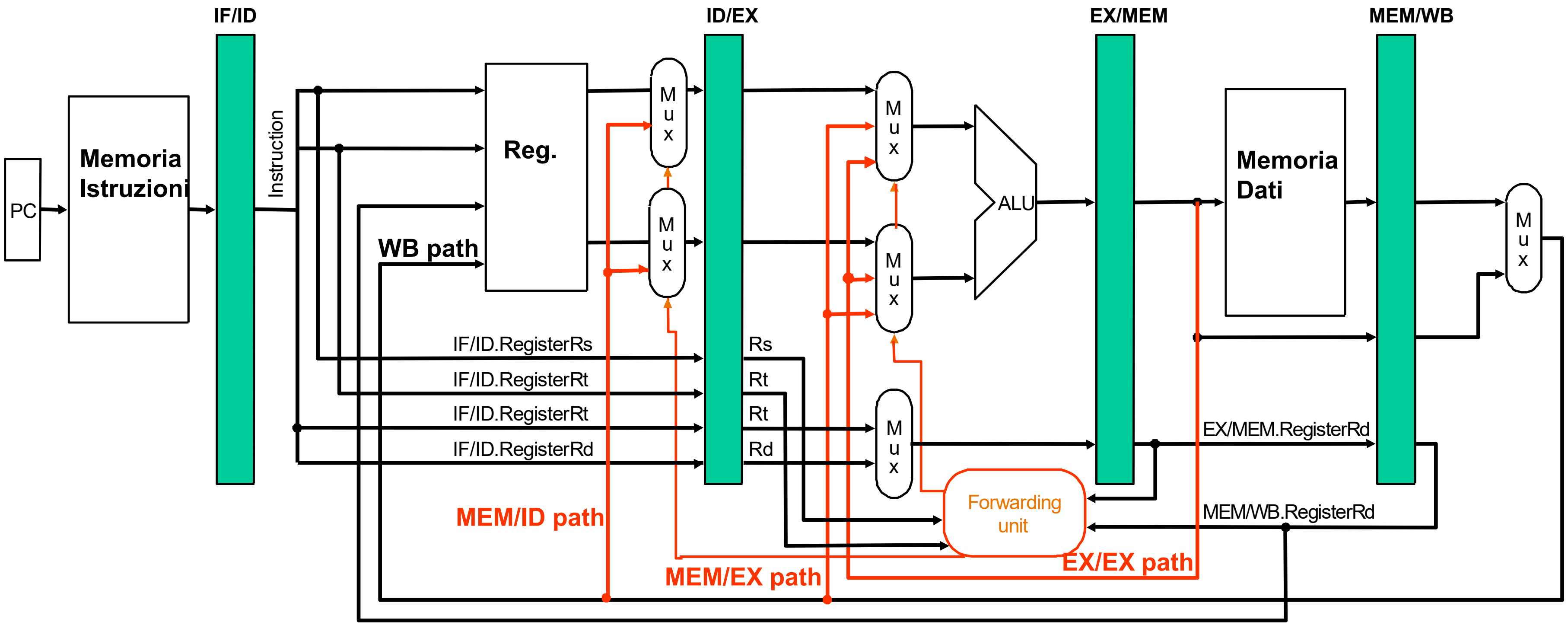


Forwarding Paths



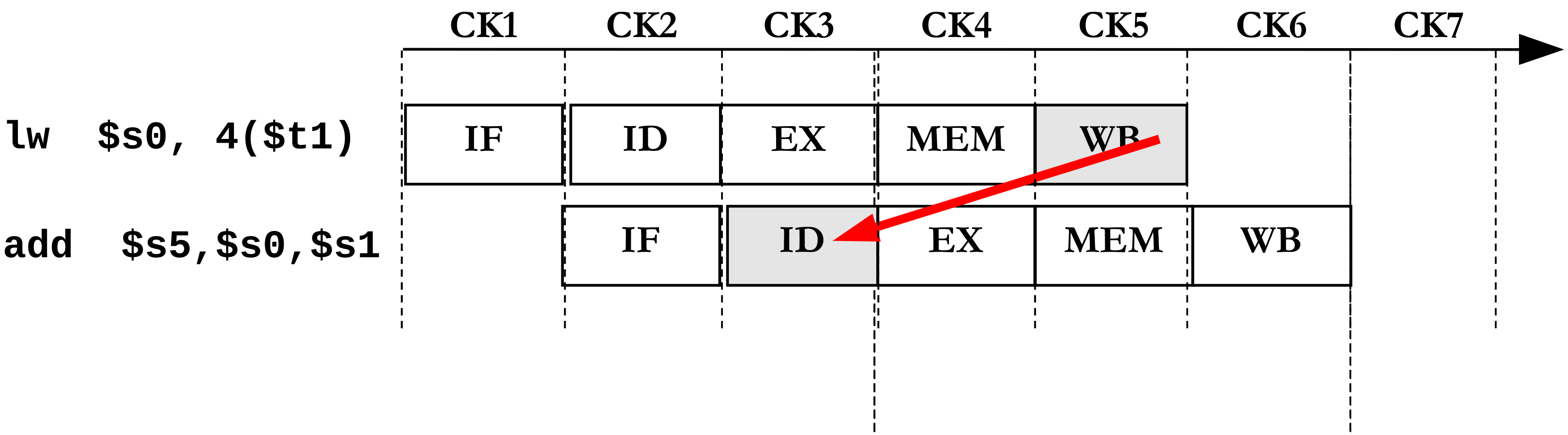
- Three data forwarding paths:
  - **EX/EX path**
  - **MEM/EX path**
  - **MEM/ID path**

Implementation of MIPS with Forwarding Unit



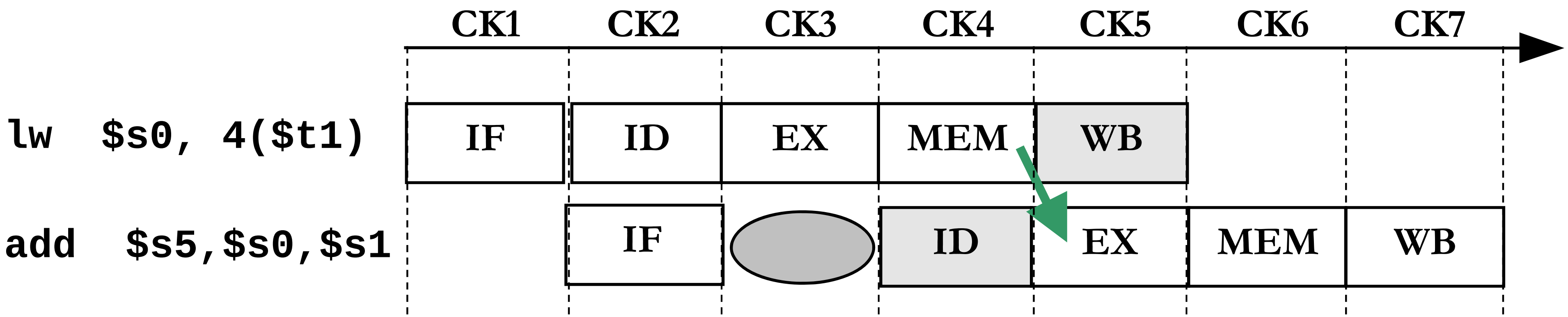
Data Hazards: Load/Use Hazard

```
L1: lw $s0, 4($t1)    # $s0 <- M [4 + $t1]
L2: add $s5, $s0, $s1 # 1° operand depends from L1
```



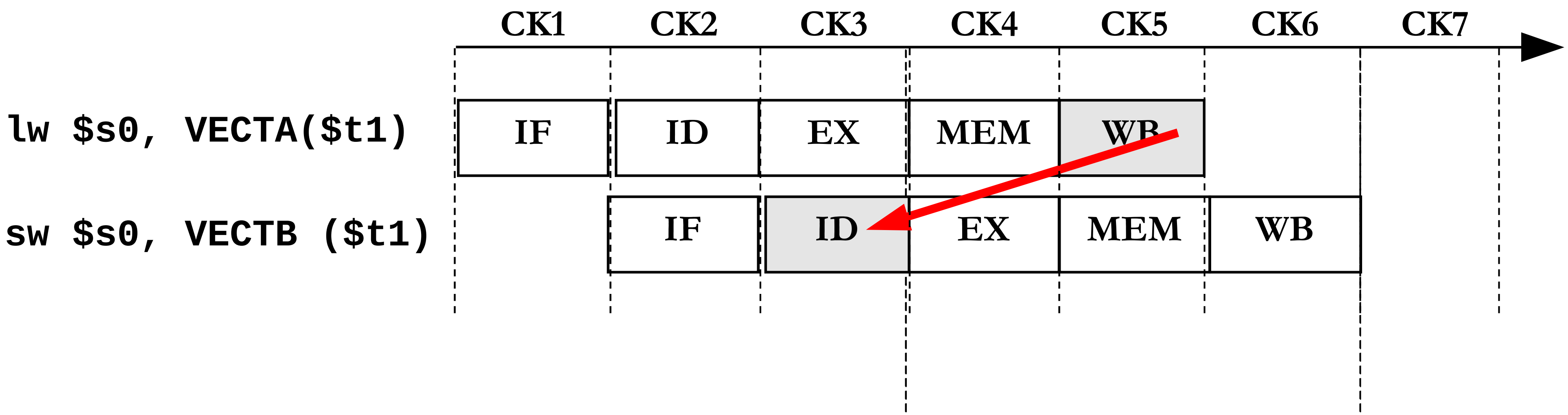
Data Hazards: Load/Use Hazard

- With forwarding using the MEM/EX path: **1 stall needed**



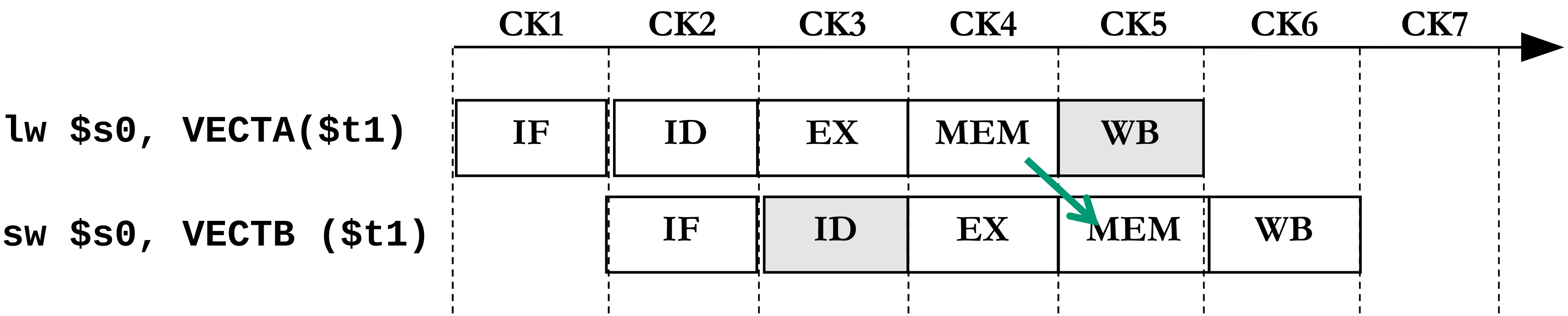
Data Hazards: Load/Store Hazard

```
L1: lw $s0, VECTA($t1)      # $s0 <- M [VECTA + $t1]
L2: sw $s0, VECTB($t1)      # M [VECTB + $t1] <- $s0
```

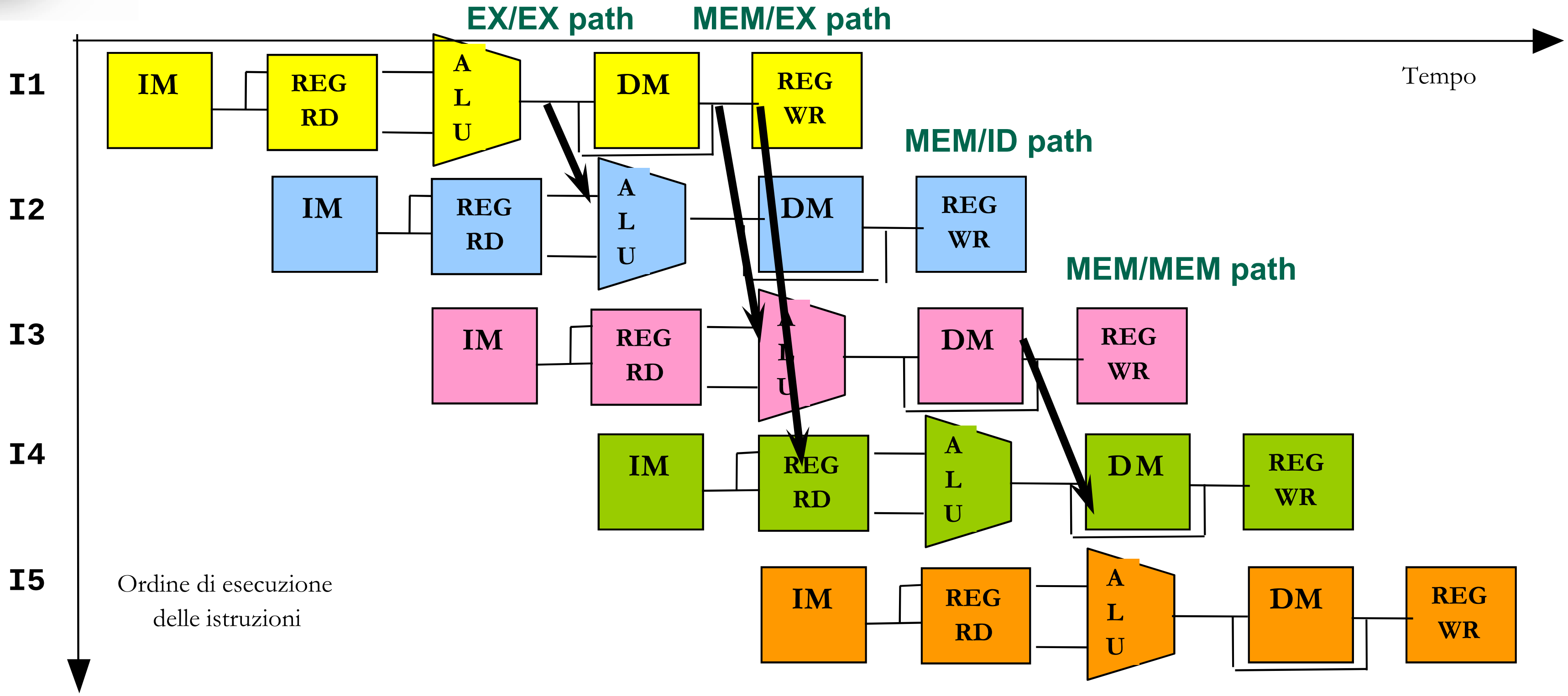


Data Hazards: Load/Store Hazard

- With forwarding by introducing the **MEM/MEM path:** **solved**



Forwarding Paths

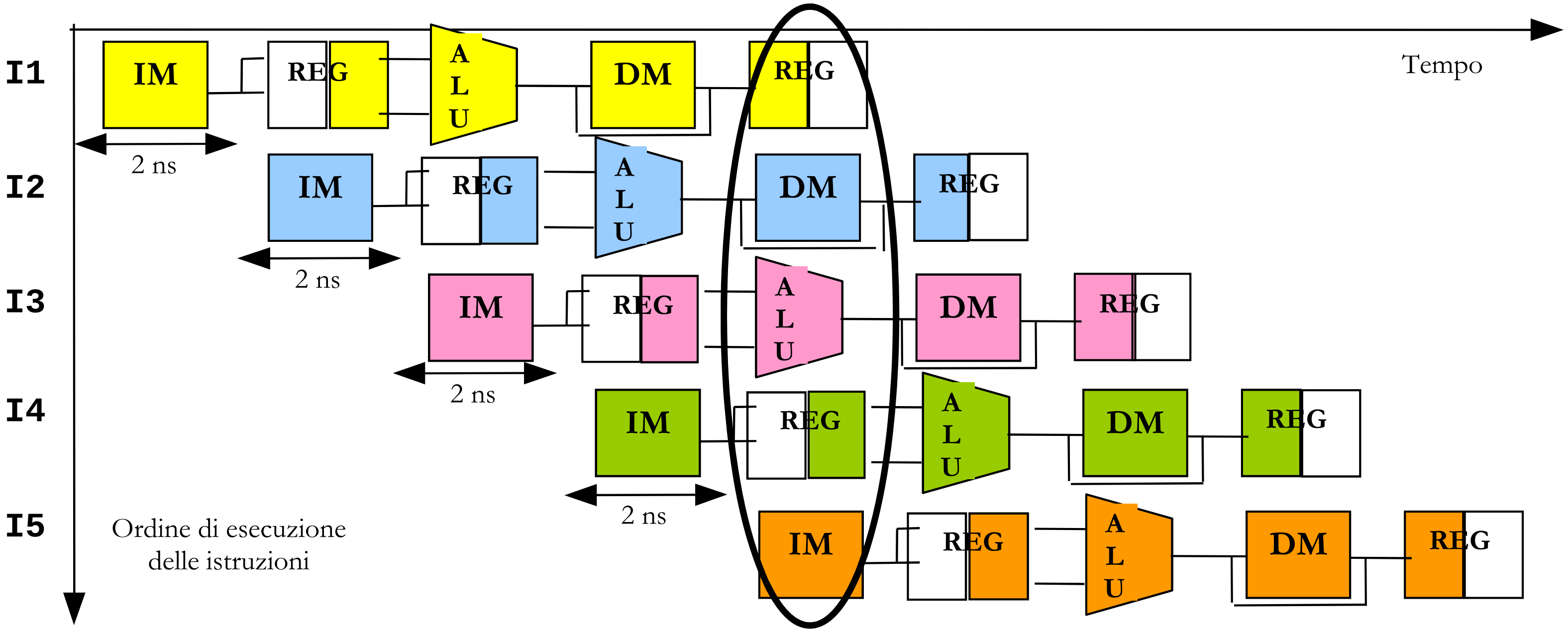


- Four data forwarding paths:
  - **EX/EX path**
  - **MEM/EX path**
  - **MEM/ID path**
  - **MEM/MEM path (for LOAD/STOREs)**

Optimized Pipeline

- Register File used in 2 stages: Read access during ID and write access during WB
- What happens if read and write refer to the same register in the same clock cycle?
  - It is necessary to insert one stall
- **Optimized Pipeline: we assume the RF read occurs in the second half of clock cycle and the RF write in the first half of clock cycle**
- What happens if read and write refer to the same register in the same clock cycle?
  - It is not necessary to insert one stall

Resources Used in the Optimized Pipeline



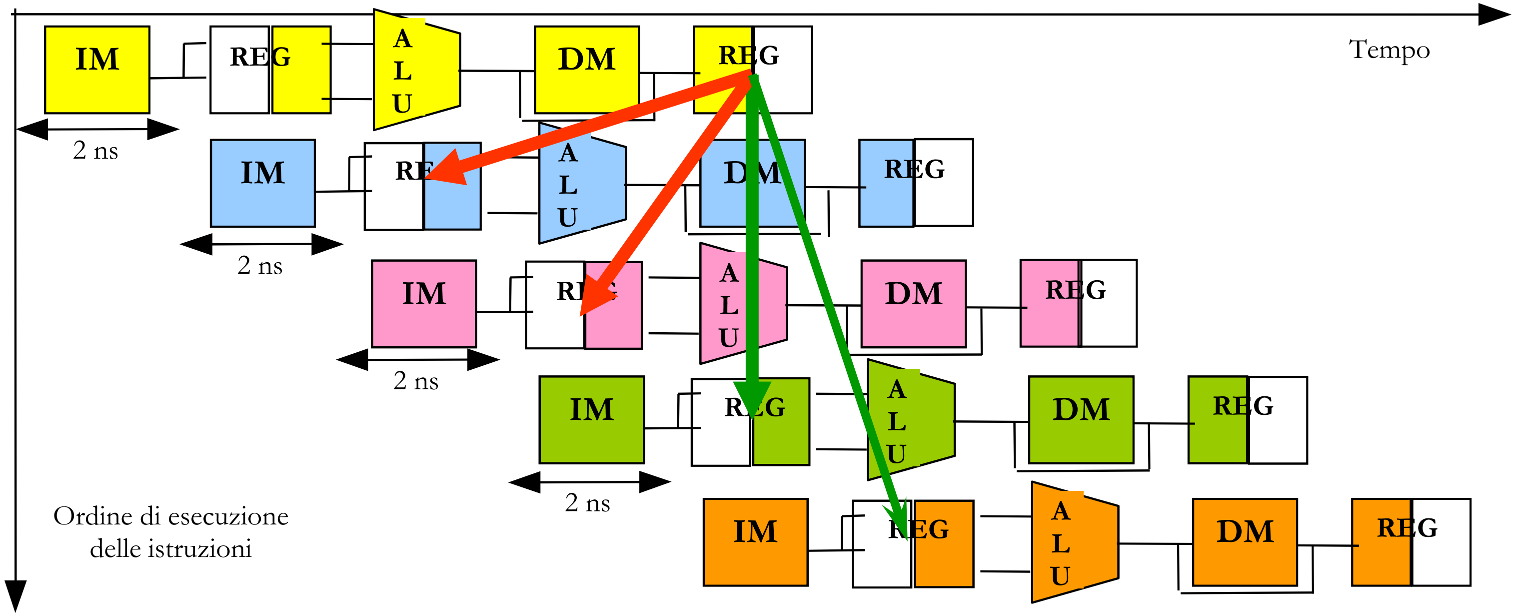
IM = Instruction Memory

REG = Register File

DM = Data Memory

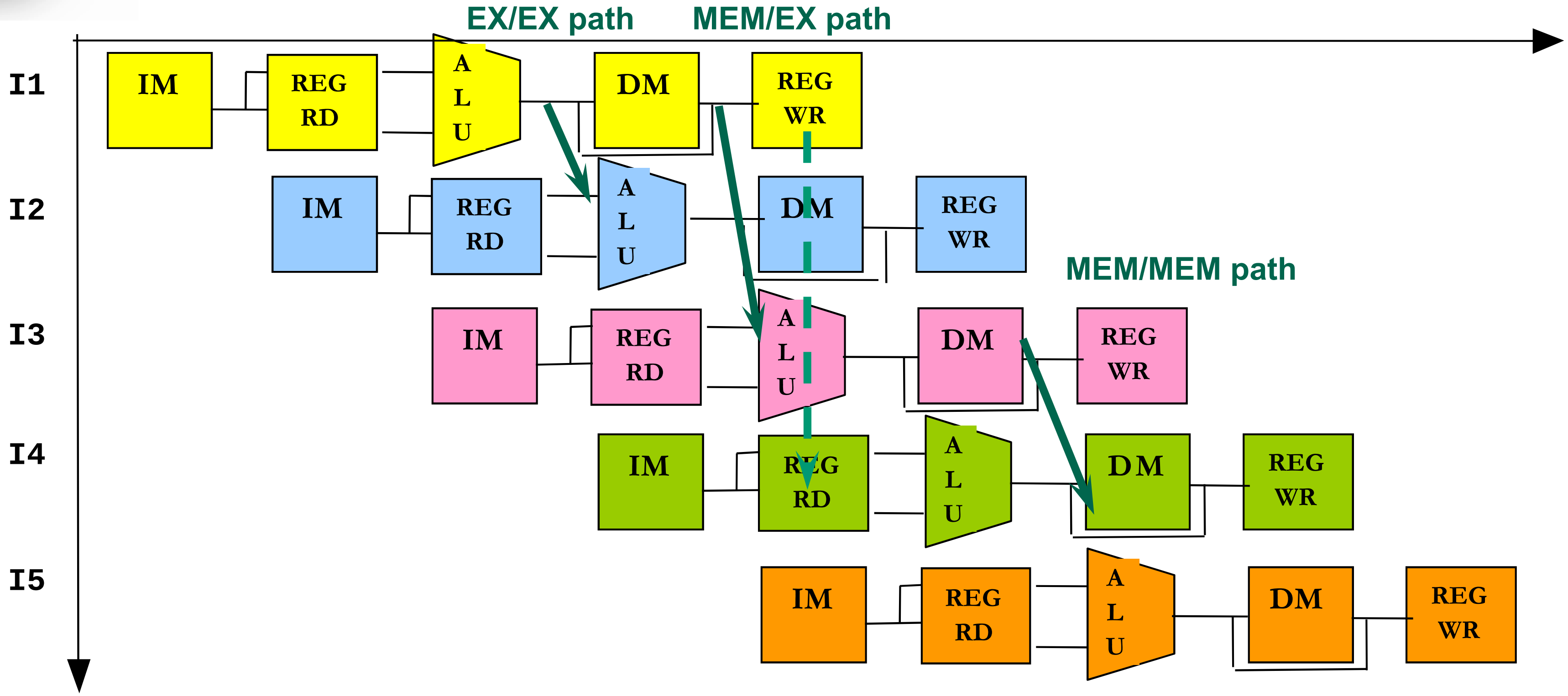
Data Hazards in the Optimized Pipeline: Example

```
sub  $2, $1, $3
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15,100($2)
```



It is necessary to insert **two** stalls

Forwarding Paths in the Optimized Pipeline



- Three forwarding paths:
  - **EX/EX path**
  - **MEM/EX path**
  - **MEM/MEM path (for LOAD/STOREs)**

Data Hazards

- Data hazards analyzed up to now are:
  1) **RAW (READ AFTER WRITE) hazard:** instruction *n+1* tries to read a source register before the previous instruction *n* has written it in the RF.
     - Example:

```
add $r1, $r2, $r3
sub $r4, $r1, $r5
```

     - By using forwarding, it is always possible to solve this conflict without introducing stalls, except for the load/use hazards where it is necessary to add one stall

Data Hazards

- Other types of data hazards in the pipeline:

2) WAW (WRITE AFTER WRITE) hazard

3) WAR (WRITE AFTER READ) hazard

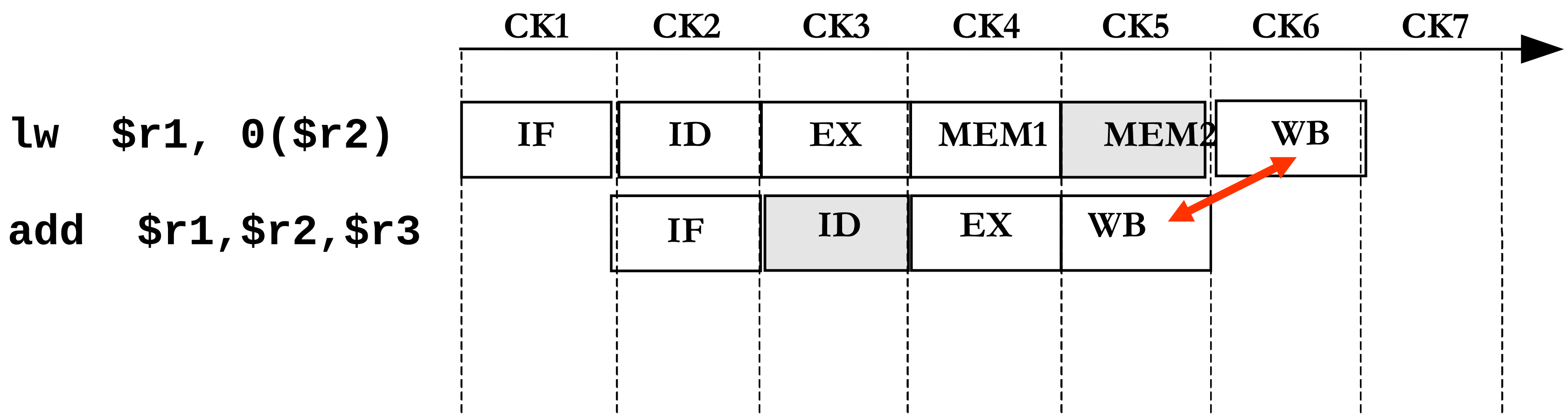
- WAW and WAR hazards occur more easily when instructions are executed **out-of-order** such as in **multi-cycle** operations to execute or to access the data memory

Data Hazards: WAW (WRITE AFTER WRITE)

- **WAW (WRITE AFTER WRITE) hazard:** Instruction *n+1* tries to write a destination operand before it has been written by the previous instruction *n* $\Rightarrow$ write operations executed in the wrong order *(out-of-order)*
  - **WAW** hazards could **not** occur in the MIPS pipeline because all the register write operations occur in the WB stage.
  - **WAW** hazards could occur in the MIPS pipeline when extending to handle **multi-cycle operations** to execute or to access the data memory because in this case instructions can complete in a different order than they were issued.

Data Hazards: WAW (WRITE AFTER WRITE)

- Example: If we assume the register write in the ALU instructions occurs in the fourth stage and that load instructions require two stages (MEM1 and MEM2) to access the data memory, we can have:



Data Hazards: WAW (WRITE AFTER WRITE)

- Example: If we assume the floating point ALU operations require a multi-cycle execution, we can have:

	CK1	CK2	CK3	CK4	CK5	CK6	CK7	CK8
`mul $f6,$f2,$f2`	IF	ID	MUL1	MUL2	MUL3	MUL4	MEM	WB
`add $f6,$f2,$f2`		IF	ID	AD1	AD2	MEM	WB	

Data Hazards: WAR (WRITE AFTER READ)

- **WAR (WRITE AFTER READ) hazard:** Instruction *n+1* tries to write a destination operand before it has been read from the previous instruction *n*
  $\Rightarrow$ instruction *n* reads the wrong value. For example:

```
sw $y, 0($x)        # sw has to read $x
addi $x, $x, 4      # addi writes Sx
```

  - WAR hazards could not occur in the MIPS pipeline because Read Operands always occur in the ID stage and write results in the WB stage.
  - As before, if we assume the register write in the ALU instructions occurs in the fourth stage and that we need two stages to access the data memory, some instructions could read operands too late in the pipeline.

Performance Evaluation in Pipelining

- Pipelining increases the CPU instruction **throughput** (number of instructions completed per unit of time), but it does not reduce the execution time (latency) of a single instruction.
- Pipelining usually slightly increases the **latency** of each instruction due to the imbalance among the pipeline stages and overhead in the control of the pipeline.
  - Imbalance among pipeline stages reduces performance since the clock can run no faster than the time needed for the slowest pipe stage.
  - Pipeline overhead arises from pipeline register delay and clock skew.
  - All instructions should be the same number of pipeline stages

Performance Metrics

IC = Instruction Count

Clock Cycles = IC + # Stall Cycles + 4

CPI = Clock Per Instruction = # Clock Cycles / IC = (IC + # Stall Cycles + 4) / IC

MIPS = f_{clock} / (CPI * 10^6)

Example

IC = Instruction Count = 5

\# Clock Cycles = IC + # Stall Cycles + 4 = 5 + 3 + 4 = 12

CPI = Clock Per Instruction = # Clock Cycles / IC = 12 / 5 = 2.4

MIPS = f_{clock} / (CPI * 10 ^{6}) = 500 MHz / 2.4 * 10 ^{6} = 208.3

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
`sub $2, $1, $3`	IF	ID	EX	ME	WB							
`and $12, $2, $5`		IF	stall	stall	stall	ID	EX	ME	WB			
`or $13, $6, $2`			stall	stall	stall	IF	ID	EX	ME	WB		
`add $14, $2, $2`							IF	ID	EX	ME	WB	
`sw $15,100($2)`								IF	ID	EX	ME	WB

Performance Metrics (2)

- Let us consider **n** iterations of a loop composed of **m** instructions per iteration requiring **k** stalls per iteration

$IC_{per\_iter} = m$

$\# \text{ Clock Cycles}_{per\ iter} = IC_{per\_iter} + \# \text{ Stall Cycles}_{per\_iter} + 4$

$CPI_{per\_iter} = (IC_{per\ iter} + \# \text{ Stall Cycles}_{per\_iter} + 4) / IC_{per\_iter}$

$= (m + k + 4) / m$

$MIPS_{per\_iter} = f_{clock} / (CPI_{per\_iter} * 10^6)$

Asymptotic Performance Metrics

- Let us consider **n** iterations of a loop composed of **m** instructions per iteration requiring **k** stalls per iteration

IC_{AS} = Instruction Count $_{AS}$ = m * n

Clock Cycles = IC $_{AS}$ + # Stall Cycles$_{AS}$ + 4

$$CPI_{AS} = \lim_{n \to \infty} (IC_{AS} + \#\,Stall\,Cycles_{AS} + 4) / IC_{AS}$$

$$= \lim_{n \to \infty} (m * n + k * n + 4) / m * n$$

$$= (m + k) / m$$

$$MIPS_{AS} = f_{clock} / (CPI_{AS} * 10^{6})$$

Performance Issues in Pipelining

- The **ideal CPI** on a pipelined processor would be 1, but stalls cause the pipeline performance to degrade form the ideal performance, so we have:

Ave. CPI Pipe = Ideal CPI + Pipe Stall Cycles per Instruction
= 1 + Pipe Stall Cycles per Instruction

- Pipe Stall Cycles per Instruction are due to Structural Hazards + Data Hazards + Control Hazards + Memory Stalls

Performance Issues in Pipelining

$$\text{Pipeline Speedup} = \frac{\text{Ave. Exec. Time Unpipelined}}{\text{Ave. Exec. Time Pipelined}} =$$

$$= \frac{\text{Ave. CPI Unp.}}{\text{Ave. CPI Pipe}} \times \frac{\text{Clock Cycle Unp.}}{\text{Clock Cycle Pipe}} =$$

Performance Issues in Pipelining

- If we ignore the cycle time overhead of pipelining and we assume the stages are perfectly balanced, the clock cycle time of two processors can be equal, so:

$$\text{Pipeline Speedup} = \frac{\text{Ave. CPI Unp.}}{1 + \text{Pipe Stall Cycles per Instruction}}$$

- Simple case: All instructions take the same number of cycles, which must also equal to the number of pipeline stages (called pipeline depth):

$$\text{Pipeline Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipe Stall Cycles per Instruction}}$$

- If there are no pipeline stalls (ideal case), this leads to the intuitive result that pipelining can improve performance by the depth of the pipeline.

Performance of Branch Schemes

- What is the performance impact of conditional branches?

$$\text{Pipeline Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipe Stall Cycles per Instruction due to Branches}}$$

$$= \frac{\text{Pipeline Depth}}{1 + \text{Branch Frequency x Branch Penalty}}$$

Reference

- **Appendix A** of the textbook:

  J. Hennessey, D. Patterson,

  "Computer Architecture: A Quantitative Approach"
  4th Edition, Morgan-Kaufmann Publishers.