Advanced Computer Architectures - Notes

260236

April 2024

Preface

Every theory section in these notes has been taken from two sources:

- Computer Architecture: A Quantitative Approach. [1]
- Course slides. [2]

About:

GitHub repository

Contents

1 Basic Concepts

This section is designed to review old concepts that are fundamental to this course.

1.1 Pipelining

1.1.1 MIPS Architecture

MIPS (**Microprocessor without Interlocked Pipelined Stages**) is a family of Reduced Instruction Set Computer (RISC). It is based on the concept of **executing only simple instruction in a reduced basic cycle to optimize the performance of CISC**[1] CPUs.

MIPS is a **load-store architecture** (or a register–register architecture), which means it is an Instruction Set Architecture (ISA[2]) that divides **instructions into two categories**:

- **Memory access** (load and store between memory and registers; load data from memory to registers; store data from registers to memory):

$$\text{Memory} \xrightarrow{\textbf{load}} \text{Registers}$$
$$\text{Memory} \xleftarrow{\textbf{store}} \text{Registers}$$

- ALU operations (which only occur between registers).

Finally, **MIPS** is also a **Pipeline Architecture**. It means that **it can execute a performance optimization technique based on overlapping the execution of multiple instructions derived from a sequential execution flow**.

[1]CISC processors use simple and complex instructions to complete any given task. Instead, the RISC processor uses the approach of increasing internal parallelism by executing a simple set of instructions in a single clock cycle (see more here).

[2]Instruction Set Architecture (ISA) is a part of the abstract model of a computer, which generally defines how software controls the CPU.

Reduced Instruction Set of MIPS Processor

The instruction set of the MIPS processor is the following:

- ALU instructions:
  - **Sum** between **two registers**:

    ```
    add $s1, $s2, $s3          # $s1 <- $s2 + $s3
    ```

    Take the values from `s2` and `s3`, make the sum and save the result on `s1`.
  - **Sum** between **register and constant**:

    ```
    addi $s1, $s1, 4           # $s1 <- $s1 + 4
    ```

    Take the value from `s1`, make the sum between `s1` and `4`, and save the result on `s1`.
- Load/Store instructions:
  - **Load**

    ```
    lw $s1, offset ($s2)       # $s1 <- Memory[$s2 + offset]
    ```

    From the `s2` register, calculate the index on the memory with the `offset`, take the value and store it in the `s1` register.
  - **Store**

    ```
    sw $s1, offset ($s2)       # Memory[$s2 + offset] <- $s1
    ```

    Take the value from the `s1` register, take the value from the `s2` register, calculate the index on the memory with the `offset`, and store the value taken from s1 in the memory.
- Branch instructions to control the instruction flow:
  - **Conditional branches**
    Only if the condition is true (branch on equal):

    ```
    beq $s1, $s2, L1     # if $s1 == $s1 then goto L1
    ```

    Only if the condition is false (branch on not equal):

    ```
    bne $s1, $s2, L1     # if $s1 != $s2 then goto L1
    ```
  - **Unconditional jumps**. The branch is always taken.
    Jump:

    ```
    j L1     # jump to L1
    ```

    Jump register:

    ```
    jr $s1   # jump to address contained in $s1
    ```

Formats of MIPS 32-bit Instructions

The previous instructions are divided into **three types**:

- Type **R (Register)**: ALU instructions.
- Type **I (Immediate)**: Load/Store instructions and Conditional branches.
- Type **J (Jump)**: Unconditional jumps instructions.

Every instruction **starts with a 6-bit opcode**. In addition to the opcode:

- R-type instructions specify:
  - **Three registers**: `rs`, `rt`, `rd`
  - A **shift amount field**: `shamt`
  - A **function field**: `funct`
- I-type instructions specify:
  - **Two registers**: `rs`, `rt`
  - **16-bit immediate value**: `offset/immediate`
- J-type instructions specify:
  - **26-bit jump target**: `address`

	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				

Figure 1: MIPS 32-bit architecture.

Scan (or click) the QR code below to view the table in high quality:

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)**
   (a) **Send** the **content** of Program Counter (PC) register to the Instruction Memory (IM);
   (b) **Fetch** the current **instruction** from Instruction Memory;
   (c) **Update** the Program Counter to the **next sequential address** by adding the value 4 to the Program Counter (4 because each instruction is 4 bytes!).
2. **Instruction Decode and Register Read (ID)**
   (a) Make the **fixed-filed recording** (**decode the current instruction**);
   (b) **Read** from the Register File (RF) of one or two registers corresponding to the registers specified in the instruction fields;
   (c) Sign-extension of the offset field of the instruction in case it is needed.
3. **Execution (EX)**. The ALU operates on the operands prepared in the previous cycle depending on the instruction type (see more details after this list):
   - **Register-Register** ALU instructions: ALU **executes the specified operation** on the operands read from the Register File.
   - **Register-Immediate** (Register-Constant) ALU instructions: ALU executes the specified operation on the first operand read from Register File and the sign-extended immediate operand.
   - **Memory Reference**: ALU adds the base register and the offset to calculate the **effective address**.
   - **Conditional Branches**: ALU compares the two registers read from Register File and computes the possible **branch target address** by adding the sign-extended offset to the incremented Program Counter.
4. **Memory Access (ME)**. It depends on the operation performed:
   - **Load**. Instructions require a **read access to the Data Memory using the effective address**.
   - **Store**. Instruction require a **write access to the Data Memory using the effective address** to write the data **from the source register read from the Register File**.
   - Conditional branches can **update the content of the Program Counter** with the branch target address, if the conditional test yielded true.

5. **Write-Back (WB)**. It depends on the operation performed:
   (a) **Load** instructions **write the data read from memory in the destination register of the Register File**.
   (b) **ALU** instructions **write the ALU results into the destination register of the Register File**.

Execution (EX) details

- **Register-Register ALU instructions**. Given the following pattern (where `op` can be the operators `add/addi` (+) or `sub/subi` (-), but not `mult` (×) or `div` (÷) because they required some special registers and therefore more phases):

```
op $x, $y, $z    # e.g. op=add  =>  $x <- $y + $z
```

  Cost: 4 clock cycles

  1. Instruction Fetch (IF) and update the Program Counter (next sequential address);
  2. Fixed-Field Decoding and read from Register File the registers: `y` and `z`;
  3. Execution (EX), ALU performs the operation `op` (`$ y op $ z`);
  4. Write-Back (WB), ALU writes the result into the destination register `x`.

- **Memory Reference**
  - Load. Given the following pattern:

```
lw $x, offset ($y)   # $x <- M[$y + offset]
```

    Cost: 5 clock cycles

    1. Instruction Fetch (IF) and update the Program Counter (next sequential address);
    2. Fixed-Field Decoding and read of Base and register `y` from Register File (RF);
    3. Execution (EX), ALU adds the base register and the offset to calculate the effective address: `y + offset`;
    4. Memory Access (ME), read access to the Data Memory (DM) using the effective (`y + offset`) address;
    5. Write-Back (WB), write the data read from memory in the destination register of the Register File (RF) `x`.

– <u>Store</u>. Given the following pattern:

```
sw $x, offset ($y)  # M[$y + offset] <- $x
```

Cost: 4 clock cycles

1. Instruction Fetch (IF) and update the Program Counter (next sequential address);
2. Fixed-Field Decoding and read of Base register `y` and source register `x` from Register File (RF);
3. Execution (EX), ALU adds the base register and the offset to calculate the effective address: `y + offset`;
4. Memory Access (WB), write the data read from memory in the destination register of the Register File (RF) `M(y + offset)`.

- **Conditional Branch**. Given the following pattern:

```
beq $x, $y, offset
```

Cost: 4 clock cycles

1. Instruction Fetch (IF) and update the Program Counter (next sequential address);
2. Fixed-Field Decoding and read of source registers `x` and `y` from Register File (RF);
3. Execution (EX), ALU compares two registers `x` and `y` and compute the possible branch target address by adding the sign-extended offset to the incremented Program Counter: `PC + 4 + offset`;
4. Memory Access (ME), update the content of the Program Counter with the branch target address (we assume that the conditional test is true).

1.1.2 Implementation of MIPS processor - Data Path

Implementing a MIPS processor isn't difficult. On the following page we show three different diagrams: the first is a very high level data path to allow the reader to understand how it works; the second is more detailed, but without the CU (Control Unit); the third is the complete data path and it also includes the CU (in red).

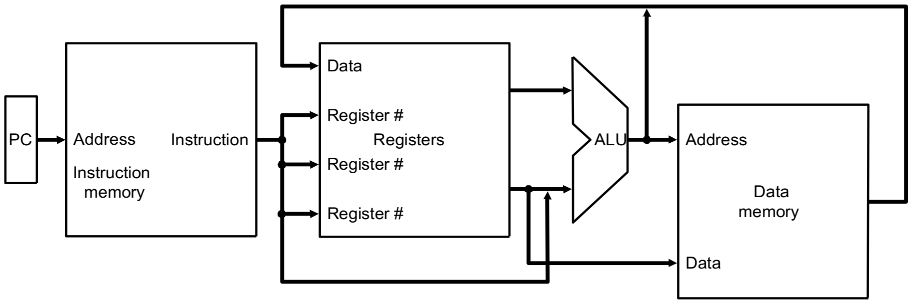


Figure 2: A basic implementation of MIPS data path. [2]

Scan (or click) the QR code below to view the figure 2 in high quality:

Two notes:

- The **Instruction Memory** (read-only memory) is separated from **Data Memory**.
- The 32 general-purpose register are organized in a **Register File** (RF) with 2 read ports and 1 write port.

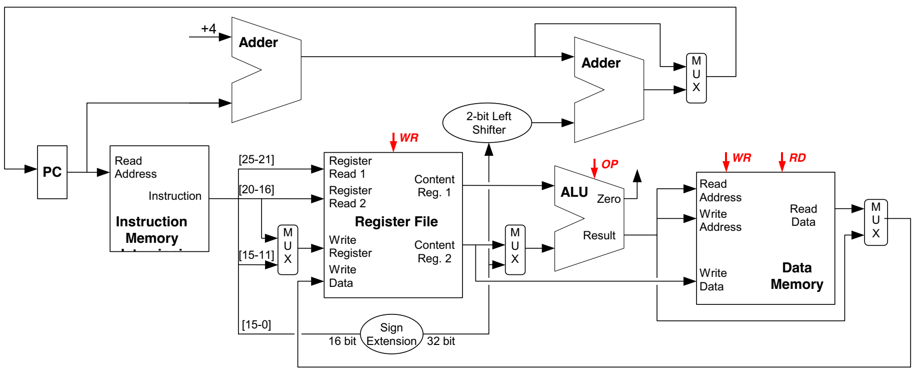


Figure 3: An implementation of MIPS data path (no Control Unit). [2]

Scan (or click) the QR code below to view the figure 3 in high quality:

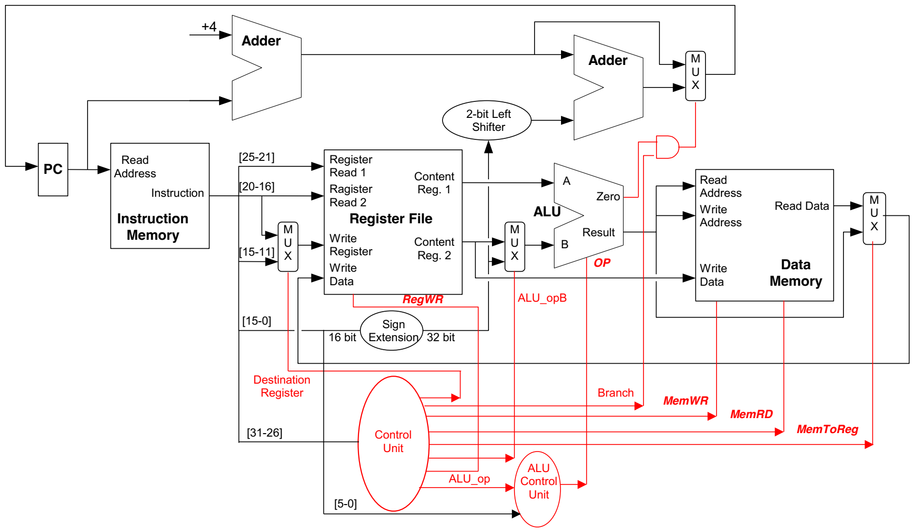


Figure 4: A complete implementation of MIPS data path. [2]

Scan (or click) the QR code below to view the figure 4 in high quality:

1.1.3 MIPS Pipelining

In simple words, the Instruction Pipelining (or Pipelining) is a technique for implementing instruction-level parallelism within a single processor. Pipelining attempts to keep every part of the processor busy with some instruction by dividing incoming instructions into a series of sequential steps (the eponymous "pipeline") performed by different processor units with different parts of instructions processed in parallel.

Definition 1: Pipelining

Pipelining is a 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** (called **pipelines stages**), requiring a fraction of the time necessary to complete the instruction. These stages are connected one to the next to form the pipeline:

1. Instructions enter the pipeline at one end;
2. Progress through the stages;
3. And exit from the other end.

As in the assembly line.

✔ **Advantage**

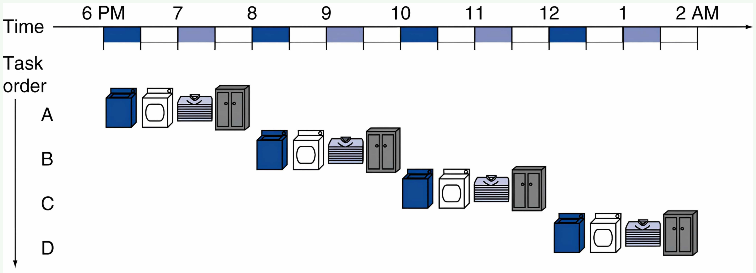
The **Pipelining is transparent for the programmer**. To understand what it means, let's make an example.

Example 1

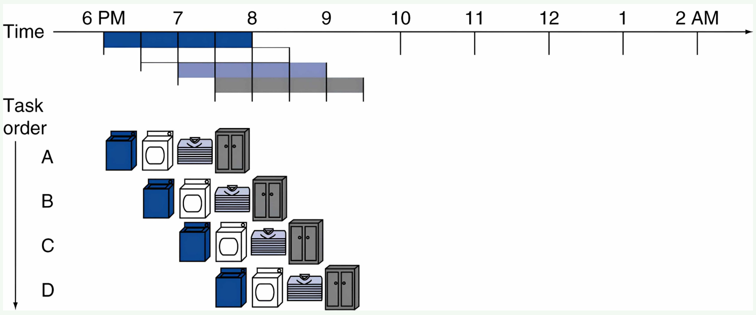
Image a car assembly line (e.g. Ferrari). A new car exits from the Ferrari assembly line in the time necessary to complete one of the phases. The pipelining technique doesn't reduce the time required to complete a car (the **latency**), BUT increases the number of vehicles produced per time unit (the **throughput**) and the frequency to complete cars.

Example 2

Image a sequential laundry jobs over time: [2]



The pipelined laundry. Overlapping execution of stages to improve throughput (number of jobs executed per hour): [2]



As introduced in the previous example, sequential execution is slower than pipelining. The following figure shows the difference (in terms of clock cycles) between sequential and pipelining.

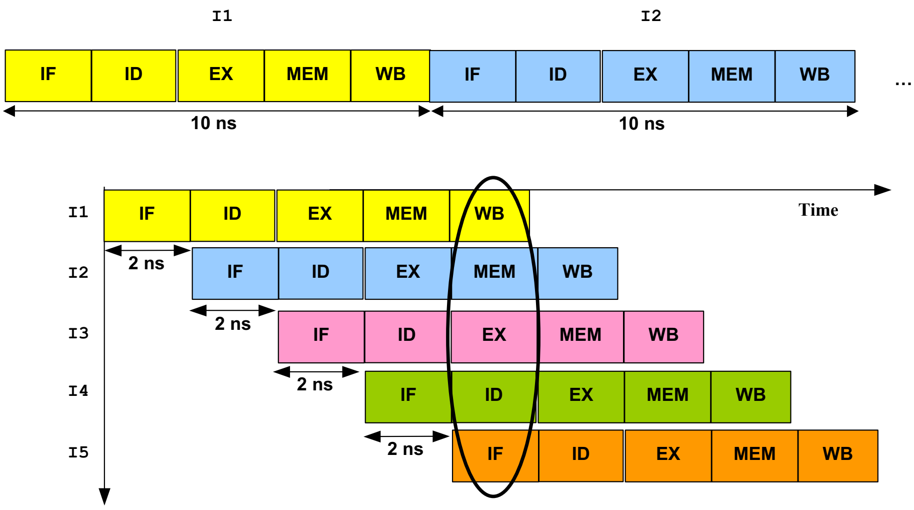


Figure 5: Sequential vs Pipelining. [2]

The time to advance the instruction of one stage in the pipeline corresponds to a clock cycle. So the total cost is: 9 clock cycles.

It's obvious that the **pipeline stages must be synchronized**: the duration of a clock cycle is defined by the time required by the slower stage in the pipeline (i.e. 2 ns). The **main 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.

Definition 2: ideal speedup

The **ideal speedup** must be the **same value of the pipeline stages**.

Look again at Figure 5. The sequential and pipelining cases consist of 5 instructions, each of which is divided into 5 low-level instructions of 2 ns each.

- The **latency** (total execution time) of each instruction is not varied, it's always 10 ns.

  Definition 3: latency

  The **latency** is the execution time of a single instruction.

- The **throughput** (number of low-level instructions completed in the time unit) is improved:
  - Sequential: 5 instructions in 50 ns (1 instruction per 10 ns, $50 \div 5 = 10$)
  - Pipelining: 5 instruction in 18 ns (1 instruction per 3.6 ns, $18 \div 5 = 3.6$)

  Definition 4: throughput

  The **throughput** is the number of instructions completed per unit of time.

Pipeline Execution of MIPS Instructions

On page 8 we discussed some MIPS instructions to understand how the MIPS architecture works. The aim of the following pages is to understand **how MIPS works in a pipelined execution**.

We want to perform the following assembly lines:

```
op $x, $y, $z        # assume $x <- $y + $z
lw $x, offset ($y)   # $x <- M[$y + offset]
sw $x, offset ($y)   # M[$y + offset] <- $x
beq $x, $y, offset
```

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

ALU Instructions: `op $x,$y,$z  # $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)  # $x ← M[$y + offset]`

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)  # M[$y + offset]← $x`

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	

Figure 6: Pipeline Execution of MIPS Instructions. [2]

Resources used during the pipeline execution

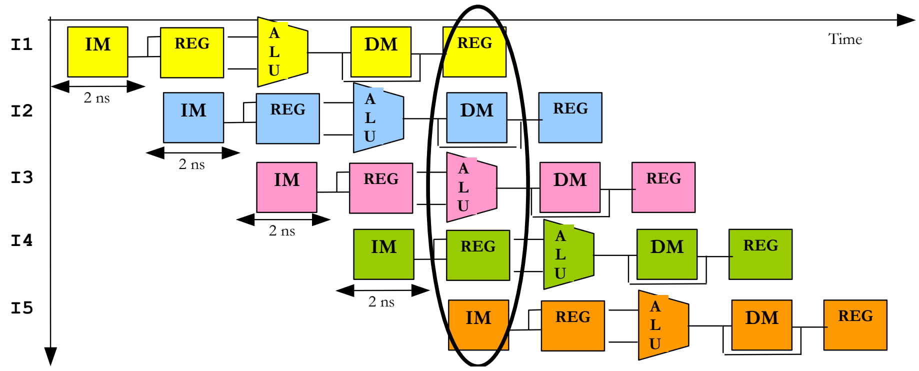


Figure 7: Resources used during the pipeline execution (IM is Instruction Memory, REG is Register File and DM is Data Memory). [2]

Implementation of MIPS pipeline

The division of the execution of each instruction in n stages implies that in each clock cycle, there are n instructions for execution. That means the CPU must have n modules corresponding to n execution stages. Therefore, to do pipelining, we need **pipeline registers to separate the different stages**.

In the following figure, we can see how the pipeline registers are implemented. Between each phase of execution of MIPS instructions (details on page 7), there is a pipeline register holding the result of the instruction.

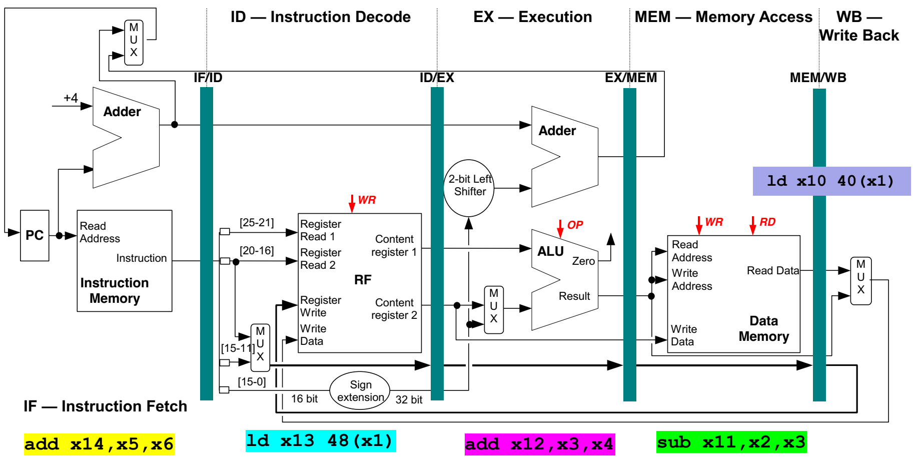


Figure 8: MIPS pipeline implementation. [2]

Note: **the data stored in the interstage registers correspond (obviously) to different instructions**.

Finally, in the following figure we can see the timeline implementation of the pipeline registers. But there are two basic assumptions to make:

1. There are no data dependencies between instructions. If there were, an instruction could read a register with an unknown value (Pipeline Hazard, page 19).

2. There are no branch/jump instructions.

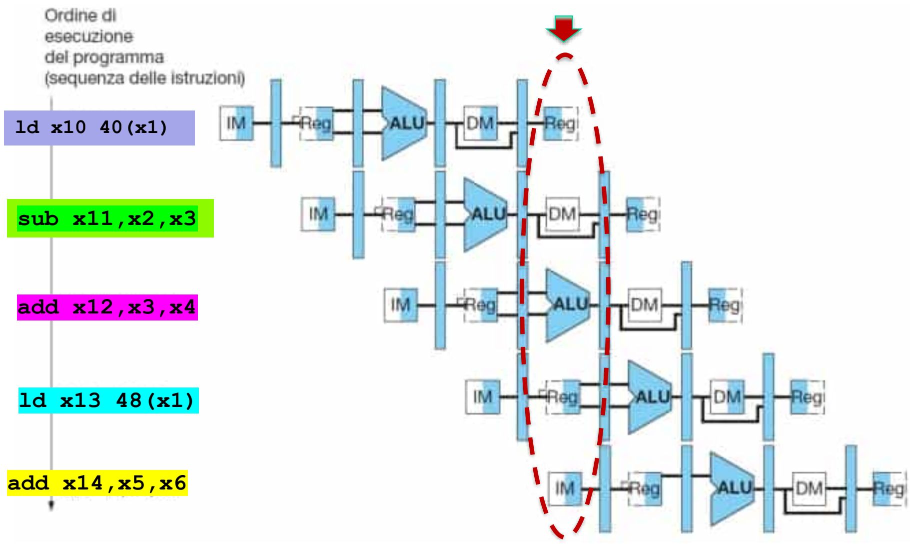


Figure 9: Timeline of MIPS pipeline implementation. [2]

1.1.4 The problem of Pipeline Hazards

Definition 5: Hazard

A **hazard (conflict)** is created whenever there a **dependence** between two 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.

Problem Consequences

The Hazards:

- **Force** the next **instruction** in the pipeline **to be executed later** than its intended clock cycle.
- **Reduced the performance** from the ideal speedup achieved by pipelining (direct previous consequence).

There are **three classes** of Hazards:

- **Structural Hazards**. Attempt to use the **same resource from different instructions simultaneously**.

  Example: single memory for instruction and data.

- **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.

  There are also **two specific forms** of data hazard, called **Load-Use Data Hazard** and **Load-Store Data Hazard**. Both occur when the **data loaded by a load instruction is not yet available when it is needed by another instruction**. In the case of Load-Use, the "another instruction" is an operator such as `add`; in the case of Load-Store, the "another instruction" is the store (`sw`) instruction.

  The following **example** shows the conflict (Load-Use Data Hazard) between two instructions. In particular, the value `lw` writes to `s2` is not available until `lw` has completed the `MEM` phase, but `and` needs this value when it enters the `EX` phase, i.e. when `lw` enters the `MEM` phase.

  ```
  lw  $s2, 20($s1)
  and $s4, $s2, $s5
  ```

- **Control Hazards**. Attempt to **make a decision on the next instruction to execute before the condition is evaluated** (more detailed analysis on page 21).

  Example: conditional branch execution.

Structural Hazards? No problem for MIPS Architecture!

There aren't any structural hazards in MIPS architecture because the Instruction Memory (IM) is separated from the Data Memory (DM). Also, the Register File (RF) is used in the same clock cycle (read access by an instruction and write access by another instruction).

❓ How to detect Data Hazards? Dependency Analysis

To **detect Data Hazards**, it is suggested to analyze the dependencies. If the instructions executed in the pipeline depend on each other, data hazards can arise **when instructions are too close**. For **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) depends on sub
add   $14, $2, $2     # 1 ($2) and 2 ($2) op.s depend on sub
sw    $15, 100($2)    # base reg. ($2) depends on sub
```

Data Hazards can occur in a variety of situations, but a **true dependency situation** is created by a **RAW (Read After Write) Hazard**.

Definition 6: Read After Write Hazard

A **RAW (Read After Write) Hazard** occurs when an instruction $n+1$ tries to read a source operand before the previous instruction n has written its value in the Register File (RF).

For **example**:

```
sub $2, $1, $3  # reg. $2 is written by sub
and $12, $2, $5 # 1 op. ($2) depends on sub
```

How to detect Control Hazards? Check conditional branches

First of all, some **examples** of conditional branches for MIPS processor are: `beq` (branch on equal) and `bne` (branch on not equal):

```
beq $s1, $s2, L1     # if $s1 == $s1 then goto L1
bne $s1, $s2, L1     # if $s1 != $s2 then goto L1
```

The **address to which you want to branch** is called the **Branch Target Address**. If the branch condition:

- Is satisfied ⇒ the **branch is taken** and the Branch Target Address is stored in the Program Counter (PC).
- Is not satisfied ⇒ the **branch is not taken** (untaken) and the instruction stream is executed sequentially with the next instruction address (PC +4).

In detail, the stages are the following:

1. `[IF]` Instruction fetch and PC increment.
2. `[ID]` Instruction Decode and Registers Read (e.g. `x` and `y`)
3. `[EX]` Compare registers (e.g. `x` and `y`) in the ALU to derive the Branch Outcome: taken or not taken. Also, computation of the Branch Target Address, so $\texttt{PC} + 4 + \texttt{offset}$
4. `[ME]` The Branch Outcome is used to decide the next PC:
   - Is satisfied ⇒ `PC` take $\texttt{PC} + 4 + \texttt{offset}$
   - Is not satisfied ⇒ `PC` take $\texttt{PC} + 4$

Let us now move on to a more interesting analysis. To understand when the Control Hazards occur, think about the Branch Outcome and the Branch Target Address. Both are ready at the end of the EX (execution) phase (so between pass number 3 and 4). Finally, branches are resolved when the Program Counter is updated at the end of the Memory Access stage (after pass number 4).

To feed the condition branch into the pipeline, we need to **create a way where the condition branch is decided before the EX stage of the next instruction**. It's obvious, because if the Branch Outcome is positive, we need to skip the next instruction and do the conditional jump instead.

This is a more detailed explanation of a control hazard. Control Hazards arise from the pipelining of conditional branches and other `jump` **instructions that change the PC**. They also **reduce the performance from the ideal speedup gained by pipelining**, because it is necessary to hold the pipeline until the branch is resolved.

MIPS Optimized Pipeline

Consider the following situation:

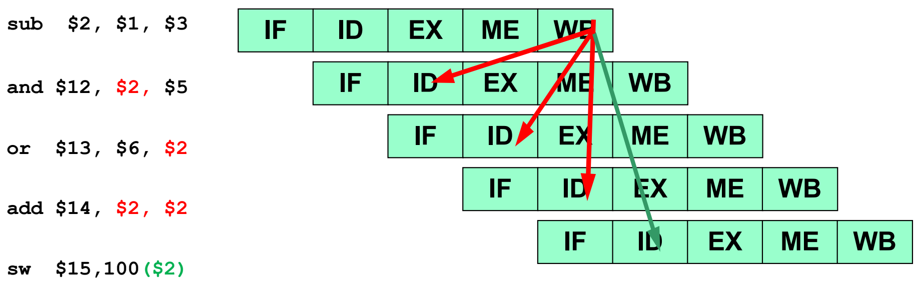


Figure 10: Why MIPS Optimized Pipeline was born. [2]

The Register File is used in 2 stages: read access during ID (`and` operation) and write access during Write Back (WB) (`sub` operation). *What happens if read and write refer to the same register in the same clock cycle?* Or we insert a stall, or we use an **optimized pipeline** (smart choice).

Definition 7: Optimized Pipeline

By selecting **Optimized Pipeline**, we assume the Register File (RF) read occurs in the second half of clock cycle and the Register File write in the first half of clock cycle.

This way **we don't need the stall**. The following Figure 11 shows an optimized pipeline.

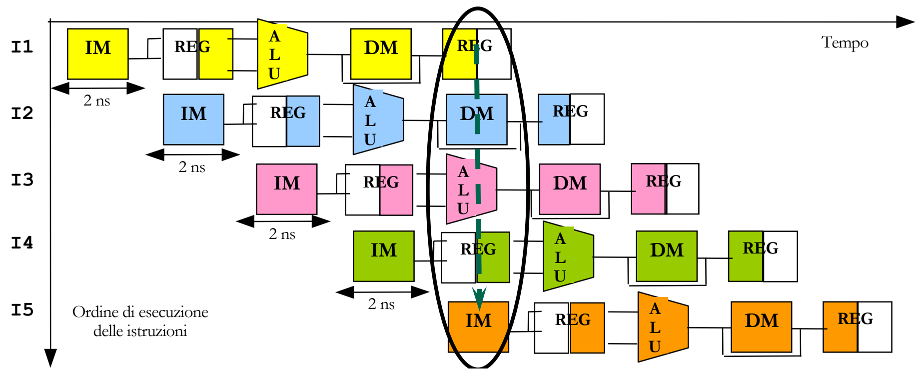


Figure 11: Optimized Pipeline (IM is Instruction Memory, REG is Register File, and DM is Data Memory). [2]

And the problem mentioned at the beginning of this paragraph is partially solved, as we can see in the following figure.

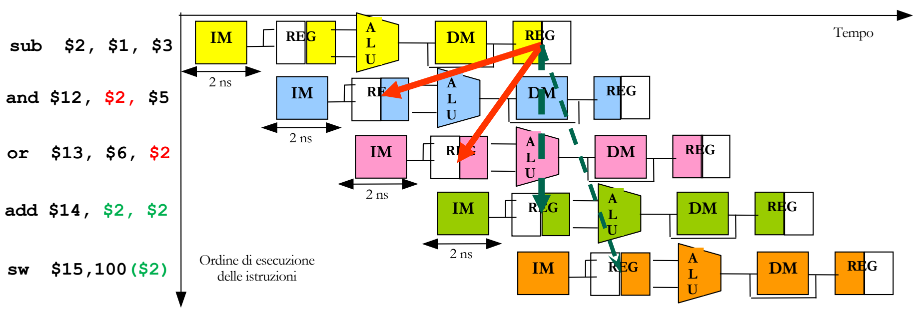


Figure 12: Optimized Pipeline to solve the example stall. [2]

1.1.5 The solution of Data Hazards

The following techniques don't solve the problem completely, but they do solve it partially. So they find a perfect balance between the ideal speedup and a situation where the hazard is total.

The solution can be applied on runtime (hardware techniques) or on compilation (static-time techniques):

- **Compilation Techniques** (static-time techniques):
  - The **insertion of `nop`** is a simple (logical) solution where we **insert a `nop` operator between dependent statements** to ensure correct operation.

    See the **example** on page 27.
  - The **instructions scheduling** is a technique used by the compiler to prevent correlating instructions from being too close together. It tries to **reorder instructions** by inserting independent instructions between correlating instructions. **If the compiler can't do this, it inserts `nop` operations**.

    See the **example** on page 28.
- **Hardware Techniques** (runtime techniques):
  - The **insertion of stalls** (called also *bubbling the pipeline*, *pipeline break*, or *pipeline stall*) is a sort of a delay before the processor can resume execution of the instruction. As we can see in the **example** on page 28, the stalls delay the stages of the correlating instructions.
  - The **data forwarding uses temporary results stored in the pipeline registers** instead of waiting for the results to be written back to the Register File (RF). To do this, it's **necessary to add new paths and multiplexers at the inputs of the ALU** to fetch inputs from the pipeline to avoid inserting stalls in the pipeline.

    See the **example** on page 28.

We have the mandatory to give more words to the data forwarding technique. First of all, its implementation needs new paths and new multiplexers. So, to adapt the MIPS architecture, the new implementation will be show in the figure 13 on page 25.

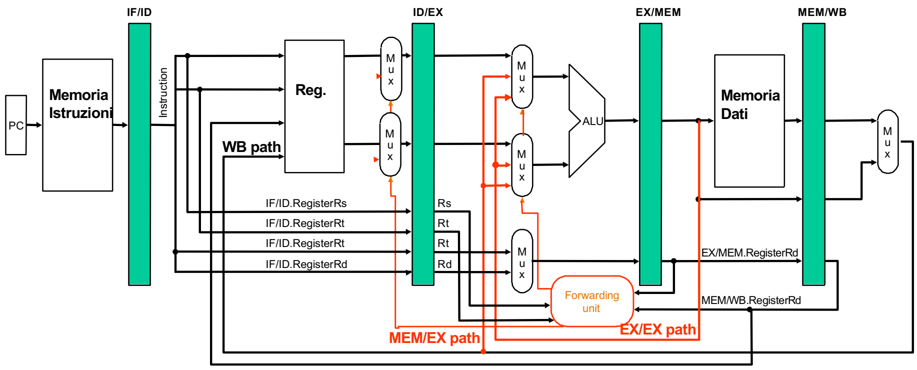


Figure 13: Implementation of MIPS with **Forwarding Unit**. [2]

Scan (or click) the QR code below to view the figure 13 in high quality:

The forwarding paths created inside the MIPS architecture are three: `EX` **to** `EX` path, `MEM` **to** `EX` path, and `MEM` **to** `MEM` path.

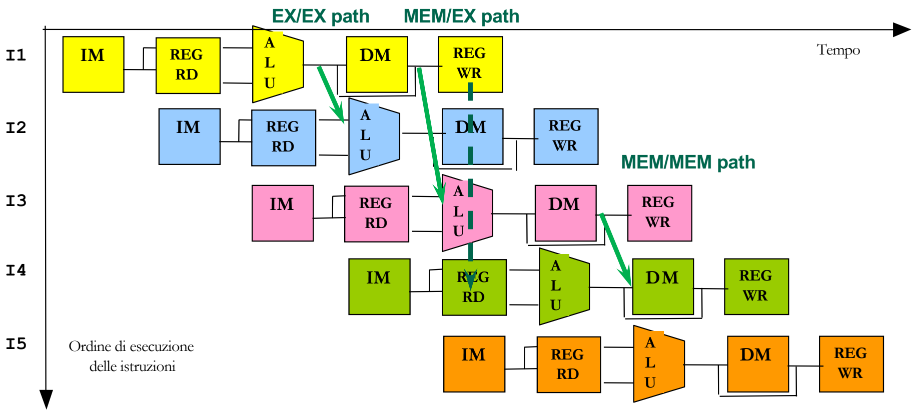


Figure 14: Forwarding paths on MIPS architecture. [2]

Furthermore, the **forwarding technique can solve** the **Load-Use** and **Load-Store** Data Hazard. It's a very interesting feature because the `MEM` to `EX` and `MEM` to `MEM` paths can solve two different situations:

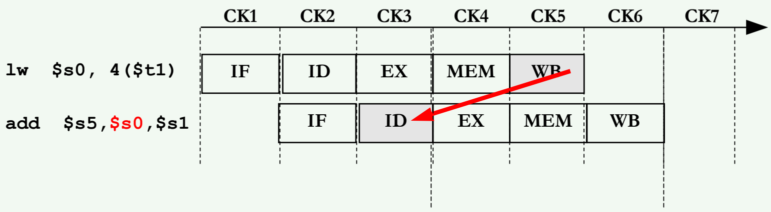
- **Load-Use** Hazard. It's **solved by MEM to EX path** because the value loaded in the MEM stage, is forwarded directly to the EX stage of the next conflict instruction (but unfortunately we need one stall to delay the run).

Example 3

Given the following code:

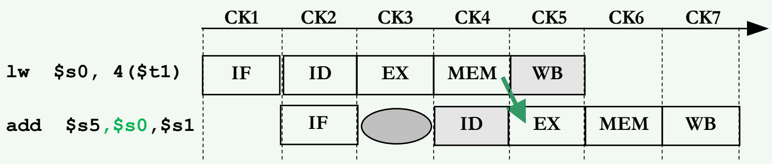
```
lw  $s0, 4($t1)      # $s0 <- M[4 + $t1]
add $s5, $s0, $s1    # 1 operand $s0 depends from lw
```

The `s0` operand depends on the load (`lw`) operator. Here, the problem of ***load-use hazard*** occurs.



The load-use hazard problem. [2]

In the figure, we can see the existing dependence. An ideal solution to the load-use hazard should be taking the value after the Memory Access operation (because the load instruction reads the effective address on the memory) and using it in the sum (operation).
The **forwarding technique solves it using the MEM-EX path but using <u>one stall</u>**.



Forwarding technique with MEM-EX path. [2]

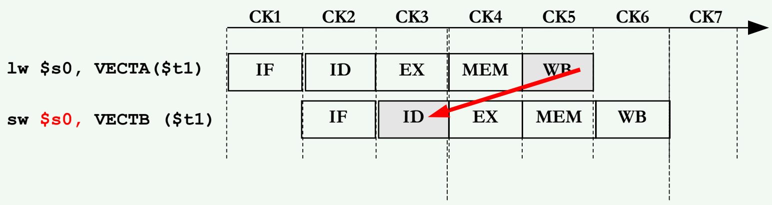
- **Load-Store** Hazard. It's **solved by MEM to MEM path** because the value loaded in the MEM stage, is forwarded directly to the MEM stage of the next conflict instruction.

Example 4

Given the following code:

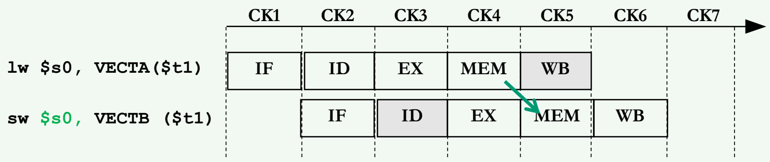
```
lw $s0, VECTA($t1)  # $s0 <- M[VECTA + $t1]
sw $s0, VECTB($t1)  # M[VECTA + $t1] <- $s0
```

The `s0` operand depends on the load (`lw`) operator. Here, the problem of ***load-store hazard*** occurs.



The load-store hazard problem. [2]

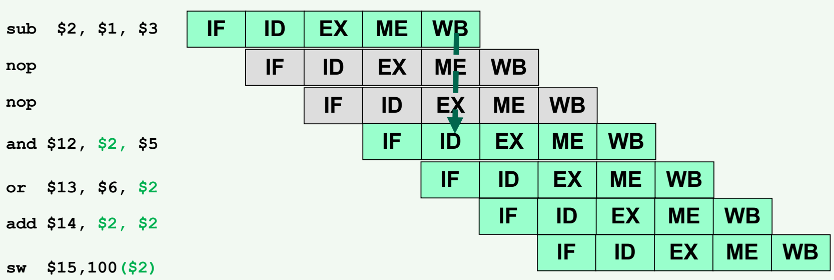
In the figure, we can see the existing dependence. An ideal solution to the load-store hazard should be taking the value after the Write Back operation (because the load instruction writes the data read from memory in the destination register of the Register File) and using it in the Instruction Decode (because the ID includes also Register Read, then it reads from the Register File (RF)).
The **forwarding technique solves it using the MEM-MEM path without any stall**.



Forwarding technique with MEM-MEM path. [2]

Example 5

In the following figure, we can see how a **compilation technique**, the **insertion of** `nop`, can be solve the data hazard problem.



Insertion of `nop`. [2]

Example 6

In the following figure, we can see how a **compilation technique**, the **instructions scheduling**, can be solve the data hazard problem.

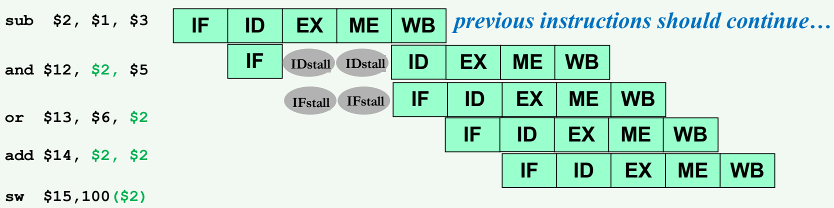
```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15,100($2)
add $4, $10, $11
and $7, $8, $9
```

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

Instructions scheduling. [2]

Example 7

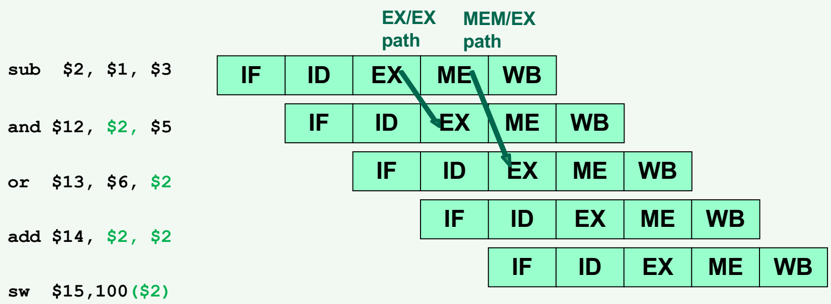
In the following figure, we can see how a **hardware technique**, the **insertion of stalls**, can be solve the data hazard problem.



Insertion of stalls. [2]

Example 8

In the following figure, we can see how a **hardware technique**, the **data forwarding**, can be solve the data hazard problem.



Data forwarding. [2]

1.1.6 The solution of Control Hazards

There are multiple techniques to resolve a Control Hazard.

✔ Conservative Solution - The Branch Stalls

The following solution is the most conservative. Solve the problem? Yes, but it's called conservative because adopt a banal technique: **stalling until resolution at the end of the Memory Access** (ME) stage of the branch.

The main problem is the **loss of performance**. Each branch costs a **penalty of 3 stalls** to decide and fetch the correct instruction flow in the pipeline:

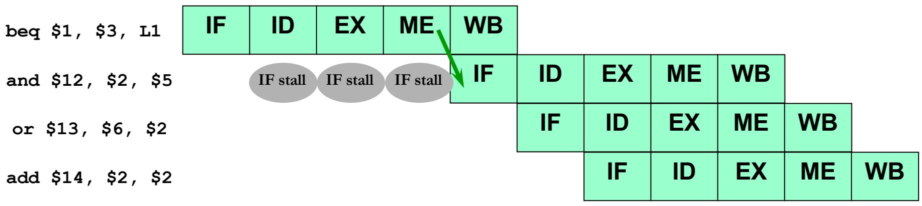


Figure 15: **Example** of a conservative solution to solve a Control Hazard.

✔ Start to think to the branch prediction - Flush solution

The branch stalls are not good because there is a reduction in throughput. So we can make *a kind of prediction* on the branch and **assume that the branch will not be taken**. So we start fetching and executing the next 3 instructions in the pipeline. Ok, but wait, **what if the branch is taken?** No problem, we ***flush*** the **next 3 instructions** before they write their result and then fetch the instruction at the branch target address.

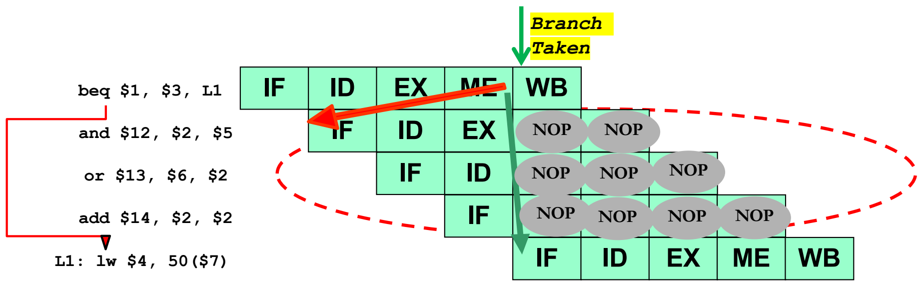


Figure 16: **Example** of a flush solution to solve a Control Hazard.

✔ Early Evaluation of the Program Counter (PC) in ID stage

It's clear that to improve performance in the event of branch hazards, we need to add more hardware features, such as:

- Compare registers to derive the Branch Outcome (BO).
- Compute the Branch Target Address (BTA).
- Update the PC register.

Fortunately, the MIPS-optimized pipeline already has these features and does so during the ID stage. As a result, the **Branch Outcome** (BO) and the **Branch Target Address** (BTA) are **known at the end of the ID stage**.

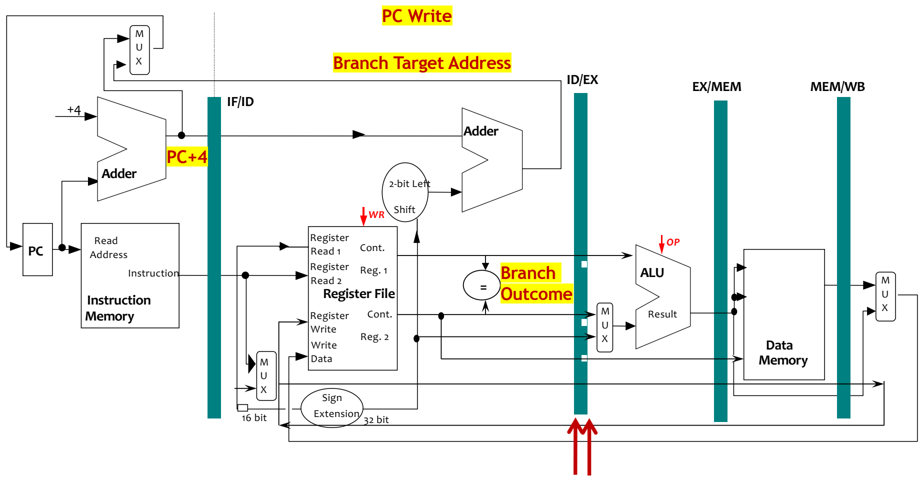


Figure 17: Early Evaluation of the Program Counter (PC) in ID stage.

Now, using the conservative solution or the flush solution, we get two different results:

- **Combo** with **Conservative Solution**: stalling until resolution at the end of the ID stage (when the Branch Outcome and the Branch Target Address are known) to decide which instruction to fetch.

  Performance consideration: each branch costs **one stall of penalty** to decide and fetch the correct instruction flow along the pipeline.

  One-cycle-delay for every branch still yields a performance loss of 10% to 30% depending on the branch frequency (Stall Cycles per Instruction due to Branches equal to Branch Frequency times to Branch Penalty):

$$\texttt{Stall Cycles} = \texttt{Branch Frequency} \times \texttt{Branch Penalty}$$

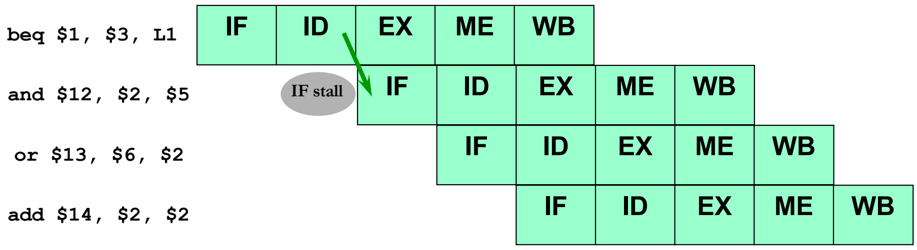


Figure 18: **Example** of conservative solution in MIPS architecture.

- **Combo** with **Fetch Solution**: we assume the **branch is not taken**.

  Performance consideration: if the Branch Outcome (BO) will be taken, it will be necessary **to flush only one instructions** before writing its results and fetch the right instruction at the Branch Target Address.

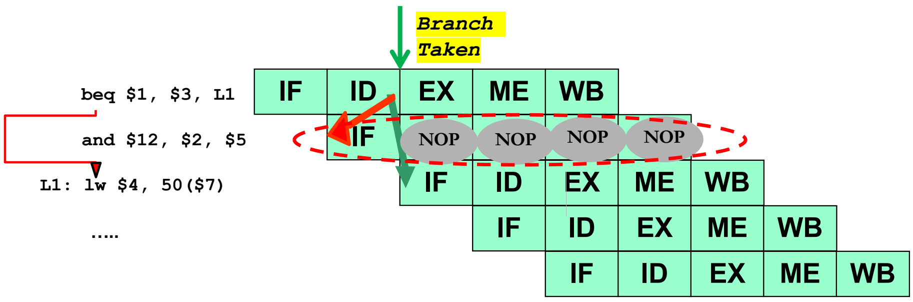


Figure 19: **Example** of fetch solution in MIPS architecture.

The unique solution is to use **branch prediction techniques** to deal with this loss of performance.

1.1.7 Performance evaluation in pipelining

As we have seen in the previous sections, the **pipelining increases the CPU instruction throughput** (number of instructions completed per unit of time) but doesn't reduce the latency (the execution time of a single instruction).

The **increase in latency is a direct consequence of two problems**:

- The **imbalance among the pipeline stages**
- The **overhead in the pipeline control**

This imbalance between the pipeline stages and the overhead are bad aspects:

- The **imbalance** reduces performance because the **clock can run no faster than the time needed for the slowest pipeline stage**;
- The **overhead** arises from the **delay introduced by interstage registers and clock skew**.

Finally, **all instructions should be the same number of pipeline stages**. Each assumption and optimization shown previously works well in this case.

Definition 8: number of Clock Cycles, Clocks Per Instructions and MIPS formula

Given:

- The **Instruction Count per iteration** as $\texttt{IC}_{\texttt{per\_iter}}$
- The **number of Stall Cycles per iteration** as `# Stall Cycles`
- The **length of the pipeline** is x

We can **calculate the number of Clock Cycles** as the sum between the Instruction Count (how many stages there are in one instruction), the number of Stall Cycles inserted by the hardware technique (called insertion of stalls), plus the length of the pipeline x:

$$\texttt{\# Clock Cycles}_{\texttt{per\_iter}} = \texttt{IC}_{\texttt{per\_iter}} + \texttt{\# Stall Cycles}_{\texttt{per\_iter}} + x \quad (1)$$

The **Clocks Per Instruction** per iteration, $\texttt{CPI}_{\texttt{per\_iter}}$, is calculated with the rapport between the number of Clock Cycles per iteration (previous equation) divided by the Instruction Count per iteration:

$$\begin{aligned} \texttt{CPI}_{\texttt{per\_iter}} &= \frac{\texttt{\# Clock Cycles}_{\texttt{per\_iter}}}{\texttt{IC}_{\texttt{per\_iter}}} \\ &= \frac{\left(\texttt{IC}_{\texttt{per\_iter}} + \texttt{\# Stall Cycles}_{\texttt{per\_iter}} + x\right)}{\texttt{IC}_{\texttt{per\_iter}}} \end{aligned} \quad (2)$$

Finally, the **MIPS formula** per iteration is calculated with the rapport between the frequency of the clock ($\texttt{f}_{\texttt{clock}}$) divided by the multiply between the Instructions Per Clock (as the ratio $1 \div \texttt{CPI}$) and 10^6 (1 million instructions):

$$\texttt{MIPS}_{\texttt{per\_iter}} = \frac{\texttt{f}_{\texttt{clock}}}{\left(\texttt{CPI}_{\texttt{per\_iter}} \times 10^6\right)} \quad (3)$$

We can asymptotically ($\texttt{AS}$) rewrite equations 1, 2 and 3 as follows:

$$\texttt{\# Clock Cycles}_{\texttt{AS}} = \texttt{IC}_{\texttt{AS}} + \texttt{\# Stall Cycles}_{\texttt{AS}} + x \tag{4}$$

$$\begin{aligned} \texttt{CPI}_{\texttt{AS}} &= \lim_{n\to\infty} \frac{\texttt{\# Clock Cycles}_{\texttt{AS}}}{\texttt{IC}_{\texttt{AS}}} \\ &= \lim_{n\to\infty} \frac{(\texttt{IC}_{\texttt{AS}} + \texttt{\# Stall Cycles}_{\texttt{AS}} + x)}{\texttt{IC}_{\texttt{AS}}} \end{aligned} \tag{5}$$

$$\texttt{MIPS}_{\texttt{AS}} = \frac{\texttt{f}_{\texttt{clock}}}{(\texttt{CPI}_{\texttt{AS}} \times 10^6)} \tag{6}$$

Note: the **ideal speedup, then Clock Per Instruction, should be equal to 1**. But stalls cause the pipeline performance to degrade from the ideal performance, so we have the **Average Clock Per Instruction (CPI)**:

$$\begin{aligned} \texttt{AVG(CPI)} &= \texttt{Ideal CPI} + \texttt{\# Stall Cycles}_{\texttt{per\_instruction}} \\ &= 1 + \texttt{\# Stall Cycles}_{\texttt{per\_instruction}} \end{aligned} \tag{7}$$

And obviously, the **Pipeline Stall Cycles per Instruction** is:

$$\texttt{PSCI} = \texttt{Structural Haz.} + \texttt{Data Haz.} + \texttt{Control Haz.} + \texttt{Memory Stalls} \tag{8}$$

1.2 Cache

1.2.1 Introduction

The cache is introduced to increase the performance of a computer through the memory system in order to:

- Provide the user the illusion to use a memory that is simultaneously large and fast.
- Provide the data to the processor at high frequency.

It takes advantage from the **Locality of Reference**.

Definition 9: Locality of Reference

Locality of reference refers to a phenomenon in which a **computer program tends to access same set of memory locations for a particular time period**.
In other words, **Locality of Reference** refers to the tendency of the computer program to access instructions whose addresses are near one another. The property of locality of reference is mainly shown by loops and subroutine calls in a program.

There are two types of Locality of Reference:

- **Temporal Locality**

  Definition 10: Temporal Locality

  Temporal Locality means that a **instruction which is recently executed have high chances of execution again**. So the instruction is kept in cache memory such that it can be fetched easily and takes no time in searching for the same instruction.

- **Spatial Locality**

  Definition 11: Spatial Locality

  Spatial Locality means that **all those instructions which are stored nearby to the recently executed instruction have high chances of execution**. It refers to the use of data elements(instructions) which are relatively close in storage locations.

Spatial Locality	Temporal Locality
In Spatial Locality, nearby instructions to recently executed instruction are likely to be executed soon.	In Temporal Locality, a recently executed instruction is likely to be executed again very soon.
It refers to the tendency of execution which involve a number of memory locations.	It refers to the tendency of execution where memory location that have been used recently have a access.
It is also known as locality in space.	It is also known as locality in time.
It only refers to data item which are closed together in memory.	It repeatedly refers to same data in short time span.
Each time new data comes into execution.	Each time same useful data comes into execution.
Example: Data elements accessed in array (where each time different, or just next, element is being accessing).	**Example**: Data elements accessed in loops (where same data elements are accessed multiple times).

Table 1: Difference between Spatial Locality and Temporal Locality.

❓ Where can we find the cache?

In general, the memory hierarchy is composed of several level. Let us consider 2 levels: cache and main memory. The **cache** (upper level) is **smaller**, **faster** and **more expensive** than the main memory (lower level).

The **minimum chunk of data that can be copied in the cache** is the ***block*** or ***cache line***. To exploit the spatial locality, the block size must be a multiple of the word size in memory. So, for example a 128-bit block size is equal to 4 words of 32-bit.

The **number of blocks in cache** is given by:

$$\texttt{Number of cache blocks} = \frac{\texttt{Cache Size}}{\texttt{Block Size}}$$

For example, if the cache size is 64K-Byte and the block size is 128-bit (16-Byte), then the number of cache blocks is 4K blocks.

1.2.2 Cache Hit and Cache Miss

Unfortunately, the Cache can't be maintain every data inside the computer. In order to be faster, it contains only some data with.

When the processor makes a request of a certain type:

- *Ideal case*. If the **requested data is** **found** **in one of the cache blocks** (upper level), then there is a hit in the cache address and it's called **Cache Hit**.
- *Problematic case*. If the **requested data is** **not found** **in one of the cache blocks** (upper level), then there is a miss in the cache address and it's called **Cache Miss**.

  But beware, in this case we need to access the lower level of the memory hierarchy to find the requested block. This causes:

  - **To stall the CPU**;
  - To require to block from the main memory;
  - To copy (write) the block in cache;
  - To repeat the cache access (hit).

Definition 12: Cache Hit

A **cache hit** is when a requested data is found in one of the cache block of the upper level of the memory.

Furthermore we define the **Hit Rate** as the **number of memory accesses that find the data in the upper level with respect to the total number of memory accesses**:

$$\texttt{Hit Rate} = \frac{\texttt{\# hits}}{\texttt{\# memory accesses}}$$

Finally we define the **Hit Time** as the **time to access the data in the upper level of the hierarchy**, **including the time needed to decide** if the attempt of access will result in a hit or miss.

Definition 13: Cache Miss

A **cache miss** is when a requested data is not found in one of the cache blocks and must be taken from the lower level of the memory.

Furthermore we define the **Miss Rate** as the **number of memory accesses not finding the data in the upper level with respect to the total number of memory accesses**:

$$\texttt{Miss Rate} = \frac{\texttt{\# misses}}{\texttt{\# memory accesses}} \tag{9}$$

Finally we define the **Miss Time** as

$$\texttt{Miss Time} = \texttt{Hit Time} + \texttt{Miss Penalty} \tag{10}$$

Where the **Miss Penalty** is **the time needed to access the lower level and to replace the block in the upper level**.

Two observations:

1. Should be obviously the definition:

$$\texttt{Hit Rate} + \texttt{Miss Rate} = 1$$

2. Typically, we have the following relation:

$$\texttt{Hit Time} \ll \texttt{Miss Penalty}$$

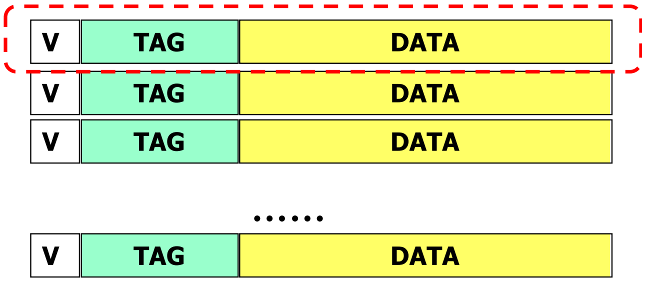
Finally, the **Average Memory Access Time (AMAT)** can be calculated as:

$$\texttt{AMAT} = \texttt{Hit Time} + \texttt{Miss Rate} * \texttt{Miss Penalty} \tag{11}$$

1.2.3 Cache Structure

Each entry (cache line) in the cache includes:

- (V) **Valid bit** to indicate if this position contains valid data or not. At the bootstrap, all the entries in the cache are marked as `INVALID`.
- (TAG) **Cache Tag(s)** contains the value that univocally identifies the memory address corresponding to the stored data.
- (DATA) **Cache Data** contains a copy of data (block or cache line).



After a general presentation of the cache structure, we answer four questions about the memory hierarchy to understand different topics:

- **Block placement** (page 39). *Where can a block be placed in the upper level?*
  - Direct Mapped Cache (page 39)
  - Fully Associative Cache (page 41)
  - n-way Set-Associative Cache (page 43)
- **Block identification**. *How is a block found if it is in the upper level?*
- **Block replacement**. *Which block should be replaced on a miss?*
- **Write strategy**. *What happens on a write?*

Block placement

The main question is: *where can a block be placed in the upper level?* In other words, the problem is: given the address of the block in the main memory, **where can the block be placed in the cache**?

So, we need to find the **correspondence between the memory address and the cache address of the block**. This correspondence **depends on the cache structure** and can be of three types:

- **Direct Mapped Cache**
- **Fully Associative Cache**
- ***n*-way Set-Associative Cache**

Direct Mapped Cache

With the **Direct Mapped Cache** structure, **each memory location corresponds to one cache location and only one cache location**. The following formula gives the **cache address of the block**:

$$(\texttt{Block Address})_{\texttt{cache}} = (\texttt{Block Address})_{\texttt{mem}} \, \texttt{mod} \, (\texttt{\# Cache Blocks}) \tag{12}$$

The *block address* of the *cache* corresponds to the modulo operation between the *block address* of the *memory* and the *number (#) of cache blocks*. The modulo operation returns a division's remainder or signed remainder after dividing one number by another.

<table>
<tr><td colspan="2">Block Address</td><td rowspan="2">Block Offset</td></tr>
<tr><td>Tag</td><td>Index</td></tr>
</table>

Figure 20: This figure shows the memory address composed of the block address (tag and index used to identify the block) and the block offset.

From Figure 21, we can see the complete structure of the cache if we choose the direct mapped cache technique.

The rectangle on the top is the memory address (Figure 20). First, we check the *Tag* value; if it's equal to the value in the cache, we check the *Valid bit* (V) to see if the position contains valid data: if the value is 1, we have a cache hit; otherwise, the data is invalid. The *Tag* contains the value that univocally identifies the memory address corresponding to the stored data. To take the *data word*, we use the *block offset* as the *selector* in the multiplexer to choose which data block to take. The index field indicates the cache row to check.

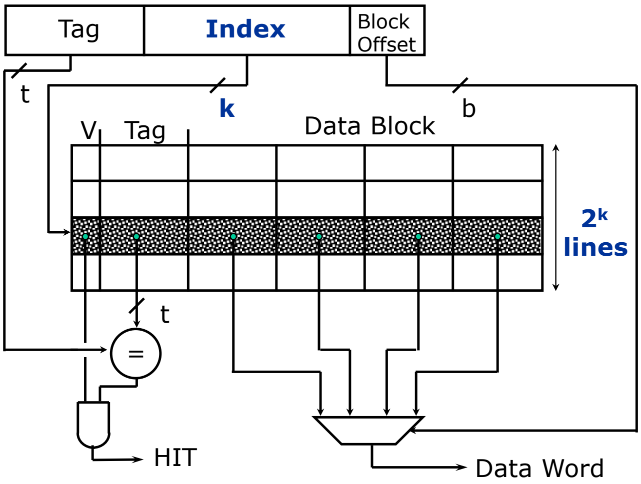


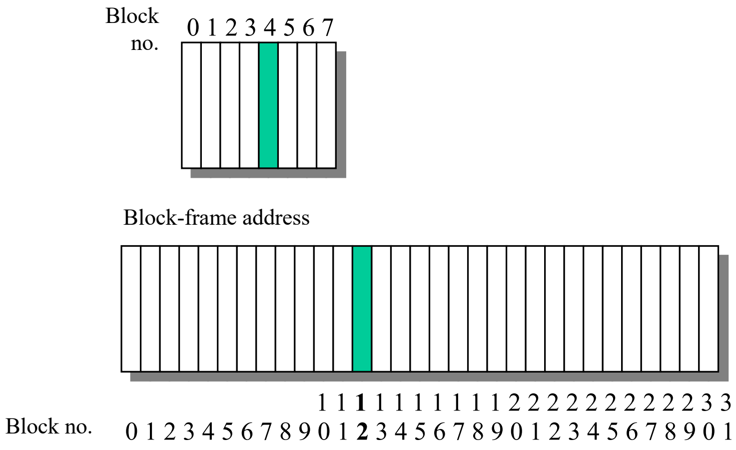
Figure 21: The cache structure of the *Direct Mapped Cache* technique.

For **example**, we assume a block-frame address composed of 32 bits. Our cache structure is direct mapped, and the number of cache blocks is 8. A possible exercise could be **determining where block 12 can be placed in the 8-block cache**.

To solve this problem, we can use the formula no 12 on page 39:

$$(\texttt{Block Address})_{\texttt{cache}} = 12 \mod 8 = 5$$

The result is 5, so the answer is: with the direct mapped technique, **the block number is 4** (because the first index of the cache blocks is zero and not 1).



Fully Associative Cache

In a **Fully Associative Cache**, the **memory block can be placed in any position of the cache**. So, all the **cache blocks must be checked during the search of the block**.

Note the **index does not exist** in the memory address; there are the Tag bits only:

$$\texttt{Number of blocks} = \frac{\texttt{Cache Size}}{\texttt{Block Size}} \tag{13}$$

The Memory Address comprises the Block Address (Tag) and the Block Offset.

Block Address	**Block Offset**
Tag	

Figure 22: The Memory Address comprises the Block Address (Tag) and the Block Offset.

The structure of the cache using this technique is as follows:

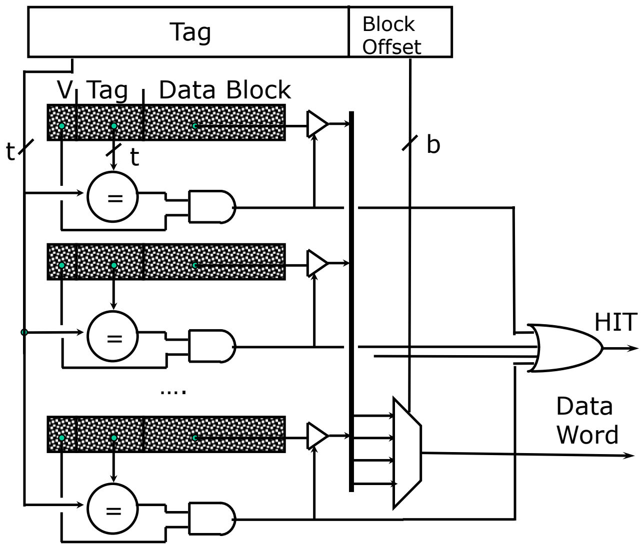
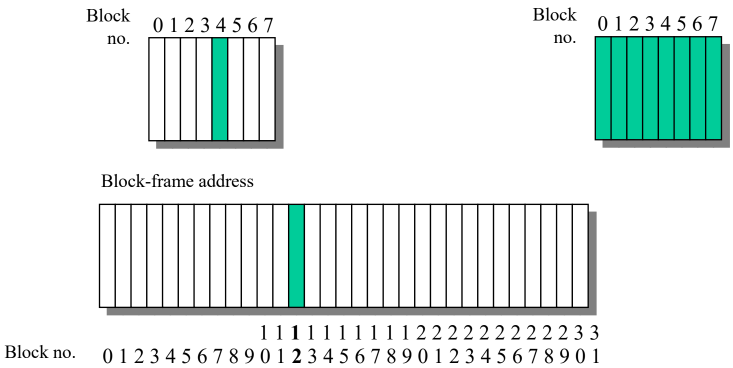


Figure 23: The cache structure of the *Fully Associative Cache* technique.

As shown in Figure 23, the cache structure is more accessible because there are no *Index* fields. We check only the *Tag* field from the memory address. Finally, the *Block Offset* chooses the *Data Block* from the cache. We have a cache hit if the *Tag* is equal to the *Tag* of the cache and the value in and with the valid bit is true.

For **example**, we assume a block-frame address composed of 32 bits. Our cache structure is fully associative, and the number of cache blocks is 8. A possible exercise could be **determining where block 12 can be placed in the 8-block cache**.

Unlike before, the position can be anywhere.



Direct Mapped on the left and Fully Associative on the right.

n-way Set-Associative Cache

In a **n-way Set Associative Cache**, the **cache is composed of sets**. Each set is composed of n blocks:

$$\begin{aligned} \texttt{Number of blocks} &= \frac{\texttt{Cache Size}}{\texttt{Block Size}} \\ \texttt{Number of sets} &= \frac{\texttt{Cache Size}}{(\texttt{Block Size} \times n)} \end{aligned} \tag{14}$$

The memory block can be placed in any block of the set, so the *search must be done on all the blocks.*

Each memory block corresponds to a single set of the cache, and the **block can be placed in whatever block of the n blocks of the set**:

$$(\texttt{Set})_{\texttt{cache}} = (\texttt{Block address})_{\texttt{mem}} \bmod (\texttt{\# sets in cache}) \tag{15}$$

Block Address		Block Offset
Tag	Index	

Figure 24: The memory address comprises the block address (Tag and index used to identify the set) and the block offset.

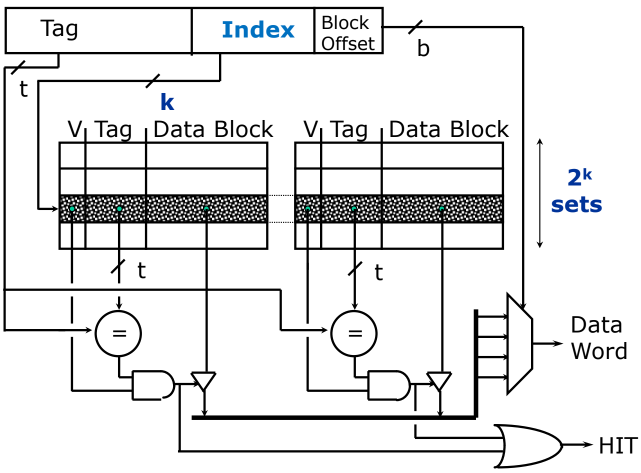
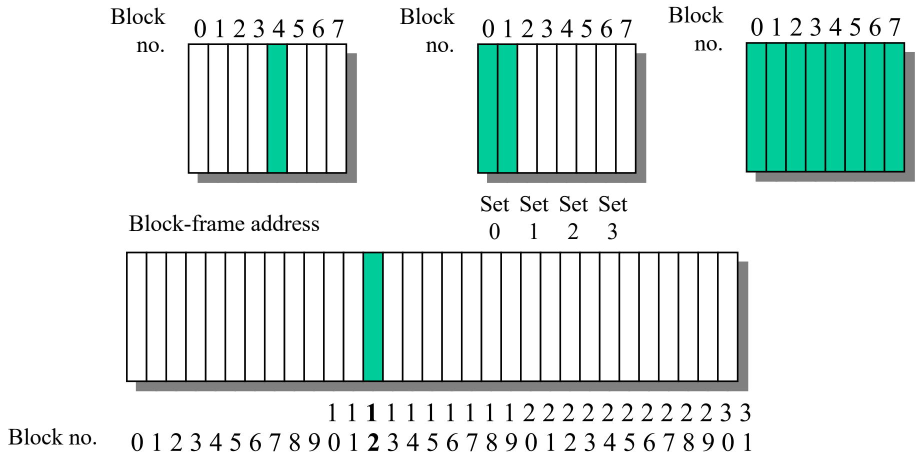


Figure 25: This structure is a **2-way Set Associative Cache**.

Taking the **examples** of previous pages, with the 2-way Set Associative, the answer is anywhere in set 0. The reason for this is that using the formula 15:

$$(\texttt{Set})_{\texttt{cache}} = 12 \mod 4 = 0$$



Direct Mapped on the left, 2-way Set Associative on the center and Fully Associative on the right.

References

[1] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. ISSN. Elsevier Science, 2017.

[2] Cristina Silvano. Lesson 1, pipelining. Slides from the HPC-E master's degree course on Politecnico di Milano, 2024.

Index