Study on the architecture of ISA 64 processors

Student:

Pop Dan Alexandru

Universitatea Tehnica din Cluj-Napoca

Facultatea de Automatica si Calculatoare

Sectia Calculatoare si Tehnologia Informatiei

Anul III, grupa 30433

Chapters:

1. Introduction.....page 3
2. Objectives……...page 3
3. Theoretical objective foundation……page 4
4. Implementation and design……page 10
5. 5.1. Testing (experimental results)…….page 20
   5.2. Conclusion……...…….……..page 38
6. References and User manual…….…….page 39

1. Introduction (final goal and history)

The aim of the project is to perform a study on the ISA 64 processors and to help the reader understand the working principles of modern processors through an example provided into a desktop application. The paper will highlight the main characteristics of processors.

The Instruction Set Architecture (ISA) is the interface between the hardware (the object) and software (the user).[11] ISA is the set of basic instructions that a processor can understand and execute, and can be viewed as the design of the computer from the perspective of the programmer.[4]

ISA defines the types of instructions supported by a processor, being either Arithmetical and Logical instructions, Data transfer instructions or Branch and Jump instructions.[3] ISA also establishes the maximum number of bits contained by all the instructions, and the endianness used in encoding of instructions (either little endian – with least significant bit at lower address, or big endian – with most significant bit at lower address).[5][11]

Represents one of the most important abstractions in computing, because it enabled binary compatibility between different generations of computers, meaning that the software written for an ISA can run on different implementations of the same ISA. For example, the AMD Athlon and the Core 2 Duo processors have different implementations, but they support the same set of basic instructions as defined in the x86 Instruction Set.[3][4]

MIPS (Microprocessor without Interlocked Pipeline Stages) is one of the most used ISA for educational purposes. Another examples of Instruction Set Architectures are: x86, ARM.[3][4][5][6][11]

Taking into consideration the abstraction levels of a computing system, ISA sits above the Microarchitecture. Because of this, two different processors can have the same ISA, but different microarchitecture implementations, resulting that the two processors can have different efficiency and performance. Therefore, when a processor is designed, the microarchitecture is implemented only after the ISA was established.[4]

Intel developed the x86 ISA, but each of its processors (i3, i5, i7) has a different microarchitecture. ARM developed the ARM ISA, used in the processors of today smartphones, but each manufacturer (such as Apple with A-series processors, and Qualcomm with Snapdragon processors) implements its own microarchitecture.[4][5][6]

2. Objectives

In the following chapter, called Theoretical Foundation, will be given general information about ISAs and will be presented their main characteristics. Also, the chapter will present the types of ISA instructions, how the registers are used and how instructions are defined. Moreover, will be presented the MIPS ISA and its types of organization (Single-cycle, Multi-cycle and Pipelined).

Will be used the MIPS to illustrate the functioning principles of an ISA, for a couple of its instructions: add, ori, lw, sw, beq, jump. This will be done in a Java application with a graphical interface built with Swing. The application will simulate how a specific instruction uses the datapath of the MIPS, by highlighting the used components and the path.

In the chapter 4, called Implementation and Design, will be explained the Java code. Chapter 5, called Testing and Experimental results, will contain screenshots of the application interface, while chapter 6 will contain References and User manual.

3. Theoretical objective foundation

Instruction Set Architectures for CPUs (Central Processing Units) can be classified in two types: CISC and RISC.[3][11]

Complex Instruction Set Computers (CISC) have numerous specialized instructions, some being never used by usual user programs. Processors with this type of processors are hard to pipeline because they have a small number of registers. Memory is accessed through most of the available instructions. Moreover, the instructions have a variable number of bits.[3]

Reduced Instruction Set Computers (RISC) have a small number of registers and few instructions, being easy to pipeline. Memory is accessed only through the Load and Store instructions. Moreover, instructions have a fixed number of bits. They have the advantage of being very efficient, because they can execute one machine instruction in a single Clock cycle.[3][7]

Processors with RISC architecture use fewer transistors than those with CISC architecture (such as the x86 processors found in most personal computers), therefore they are cheaper, power efficient, and dissipate heat better. These characteristics are desirable for light, portable, battery-powered devices—including smartphones, laptops and tablet computers. This is the reason why mobile devices use ARM processors with RISC architecture, while workstations use CISC architecture.[10]

Modern ISAs are General Purpose Registers (GPRs), because the operands of the Arithmetic and Logic Unit (ALU) are either registers or memory locations. Exist four classes of ISAs:[11]

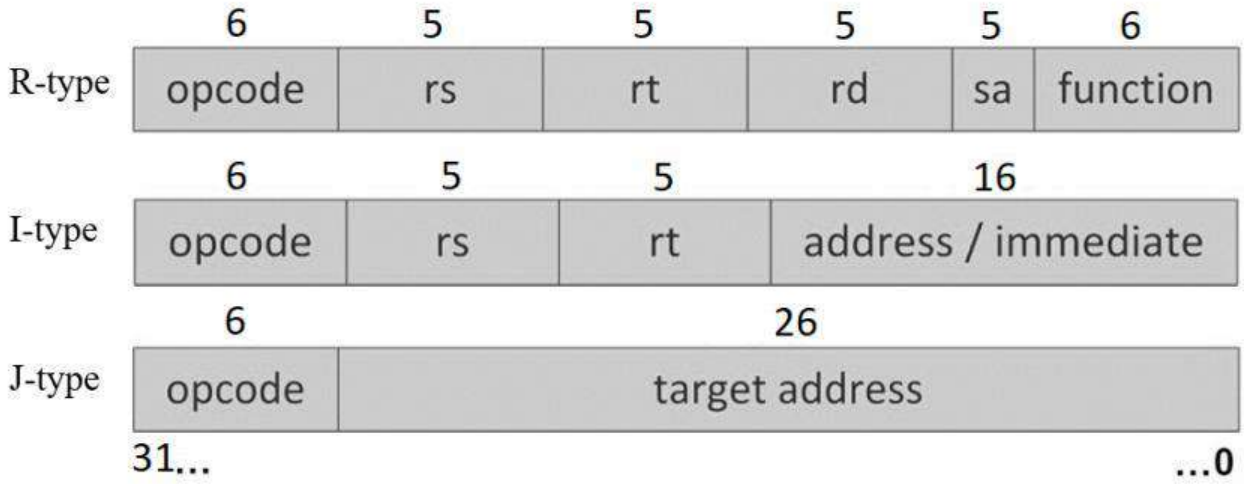
- First one is the Register-to-Memory ISA, in which ALU operations are done with data from both: registers and memory. The result is one of the source registers.
- Second class is the Register-to-Register ISA (also called Load-Store ISA), in which ALU operations are done with data provided only by registers, not by memory. Therefore, both operands must be registers. The memory is accessed by using Load and Store instructions.
- Third one is the Stack, in which the operands are implicitly the Top-of-Stack (TOS) and Second-on-Stack (SOS). For performing memory transfers are used Push and Pop instructions. The result of an operation goes to the TOS.
- Forth class is the Accumulator, in which one operand is the accumulator register and the other is explicitly given.

The MIPS (Microprocessor without Interlocked Pipeline Stages) is a RISC processor, therefore is using a Load-Store ISA, because only Load and Store instructions access the Memory.[11] It was developed by MIPS Technologies company from USA. Also, MIPS implies that instructions are aligned in the memory. Because of this, data is accessed faster. MIPS uses little-endian notation, meaning that the Least Significant Bit is the right-most bit.[5]

MIPS uses 32 GPRs, R[0] has the fixed value of "0" and R[31] contains the returning address of a call.[11]

The type of instructions performed and supported by modern ISAs are:[11][4][5][8]

- R-type instructions that are used for arithmetical and logical operations → use 6 registers → examples of instructions: ADD
- I-type instructions that are used for memory data transfers or conditional jumps/branches or for arithmetical and logical operations with an immediate → use 4 registers → examples of instructions: ORI, SW, BEQ, LW
- J-type instructions that are used for unconditional jumps → use 2 registers for encoding → examples of instructions: J



Opcode is on 6 bits. For I-type and J-type instructions, Opcode encodes uniquely the instructions, while for R-type instructions Opcode is "000000" and the operation executed in ALU is encoded uniquely by Function field. [11][5][9 → minute 18:40]

- "rs" is the source register
- "rt" is the target register
- "rd" is the destination register
- "sa" is the shift amount
- "function" is used to make distinction between R-type instructions
- "immediate" is the binary representation of the Immediate value
- "target address" is the address where must be jumped

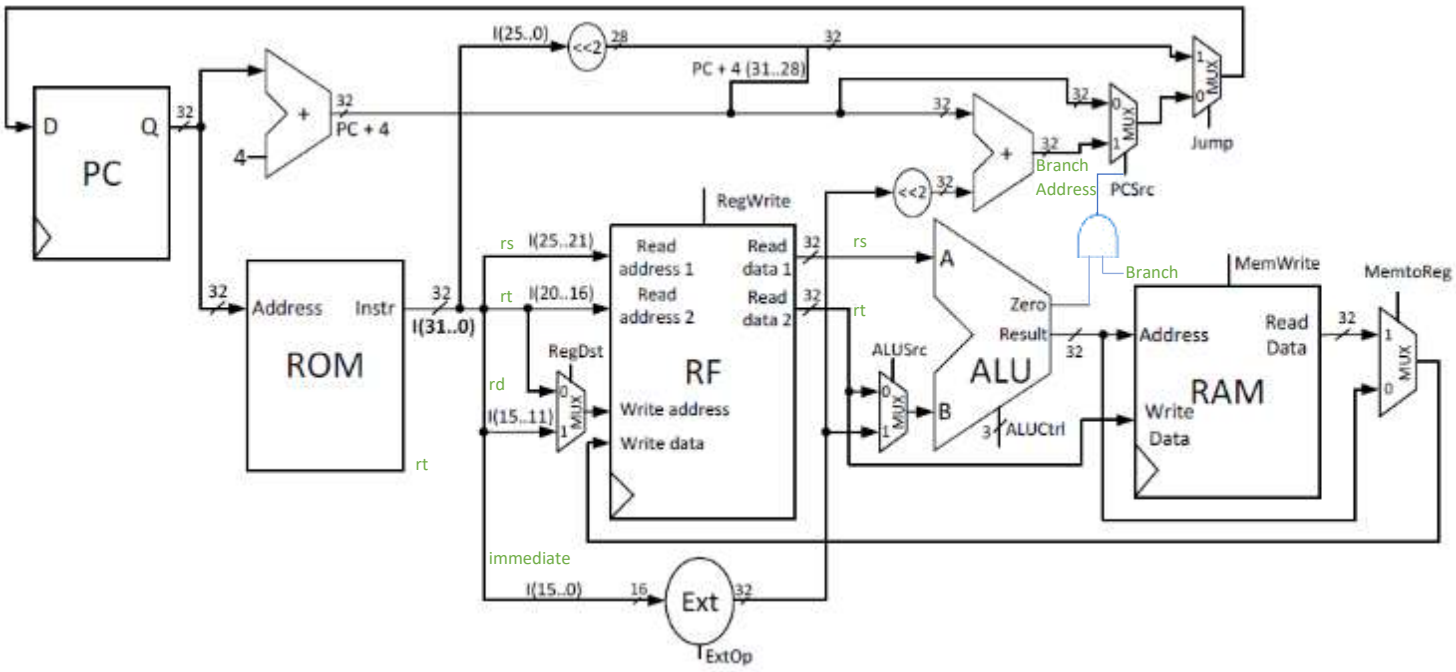
If the Immediate value has Sign then is used the Sign Extension, else is used Zero Extension.[11]

For arithmetical operations is used Sign extension, while for logical operations is used Zero extension.[11]

Because MIPS is on 32 bits, then registers are on 32 bits and ALU operands are also on 32 bits. The Program Counter (PC) contains the address of the next instruction that must be fetched (extracted).[11]

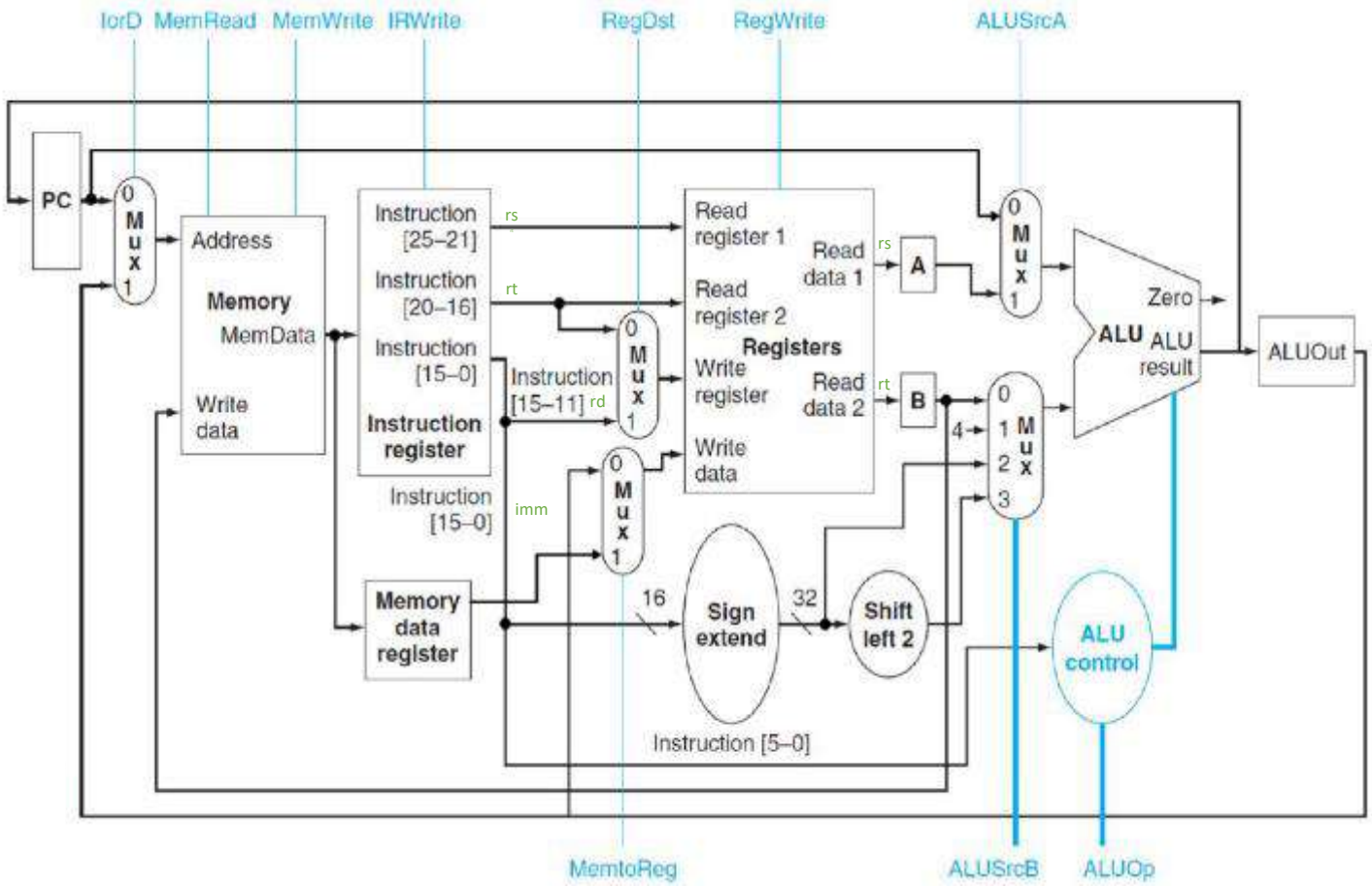
MIPS ISA can have three types of organizations: Single-cycle, Multi-cycle or Pipelined.[11]

1) Single-cycle MIPS[11]



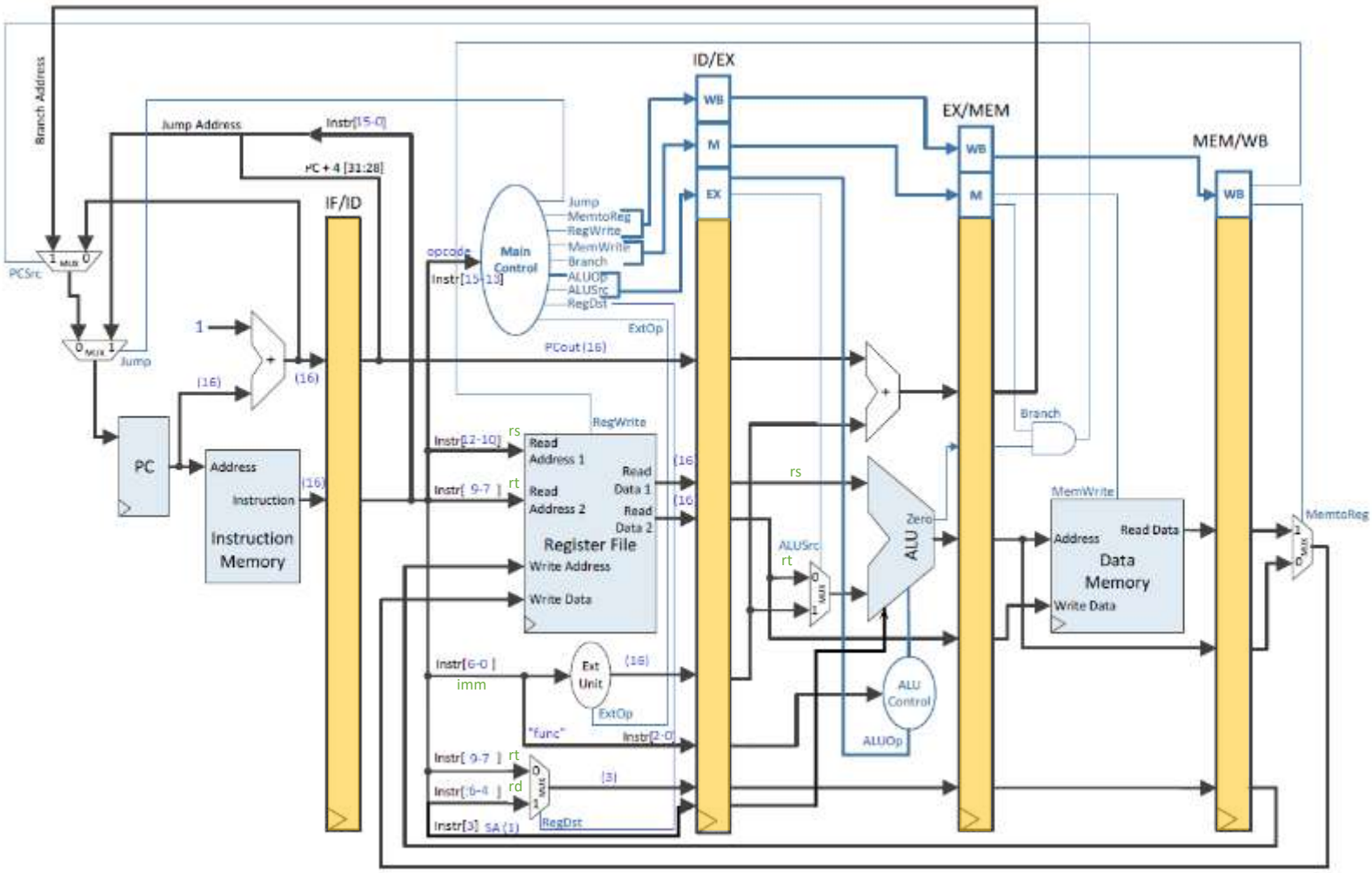
➔ It is the most inefficient, because a result is obtained only after execution of current instruction is finished.

2) Multi-cycle MIPS[11]



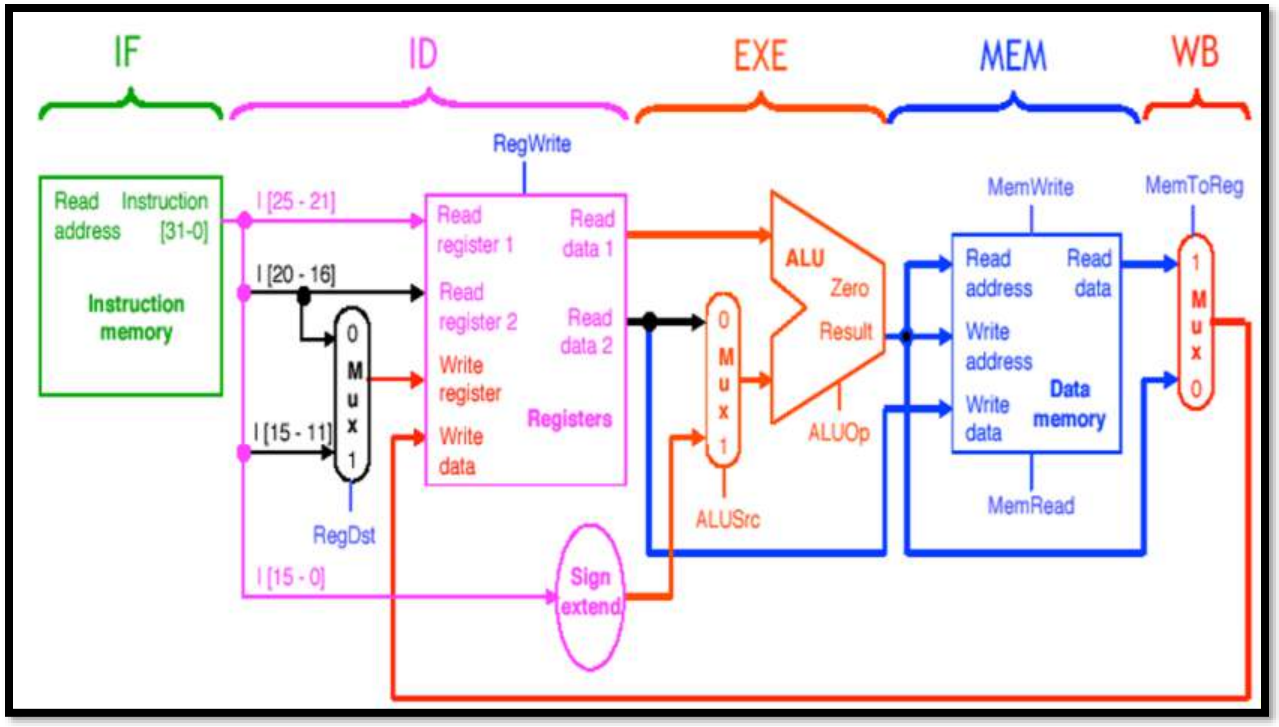
- Is used one memory for both data and instructions
- ALU increments the PC and computes the Branch Address
- Are used 5 registers to reduce the CLK cycles: Instruction Register, Memory data register, register A, register B, ALUout Register
- Execution of instructions take between 3 to 5 CLK cycles

3) Pipelined MIPS[11]



The Pipeline is the set of stages connected in series through registers. MIPS has 5 stages:

- Instruction Fetch
- Instruction Decode
- Execute
- Memory
- Write back (output returns to the Register File through port WD, Write Data)



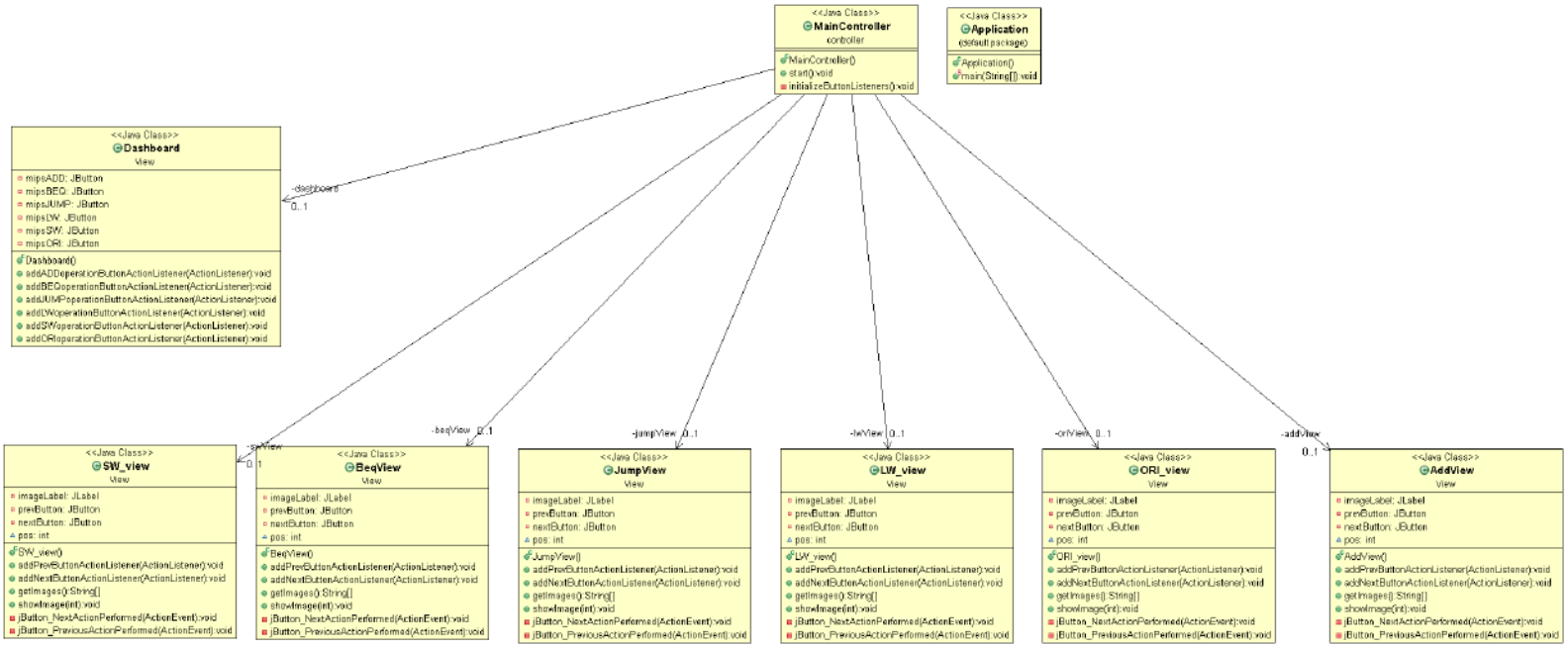
Executing a MIPS instruction can take up to five steps.[5][6][7][11]

Step	Name	Description
Instruction Fetch	IF	Read an instruction from memory.
Instruction Decode	ID	Read source registers and generate control signals.
Execute	EX	Compute an R-type result or a branch outcome.
Memory	MEM	Read or write the data memory.
Writeback	WB	Store a result in the destination register.

However, not all instructions use all five steps in Pipelined MIPS in order to be executed.

Instruction	Steps required				
beq	IF	ID	EX		
R-type	IF	ID	EX		WB
sw	IF	ID	EX	MEM	
lw	IF	ID	EX	MEM	WB

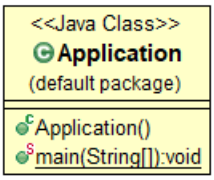
4. Implementation and design



UML diagram of the application

The application is organized on three packages: *default package*, *controller* and *view*. Also, the application contains a "View" class for each type operation.

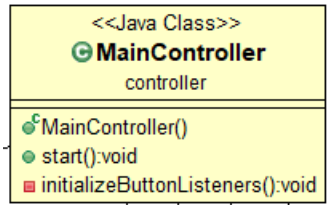
1. Class "Application"



The "Application" class contains the main method of the entire application, which creates the controller and "starts" the application.

```java
import controller.MainController;

public class Application {

    public static void main(final String[] args) {

        new MainController().start();   // creates the main controller which starts the application

    }
}
```

2. Class "MainController"



The class creates the windows and makes them visible in the method "start" and in the method "initializeButtonListeners" are given the set of actions for each action listener in the inside functions.

The class uses the libraries:

- UIManager from SWING UI library → manages the current look and feel, and the methods for setting various looks.
- ActionEvent → indicates that a component-defined action occurred. This high-level event is generated by a component (such as a Button) when the component-specific action occurs (such as being pressed). The event is passed to every ActionListener object that registered to receive such events using the component's addActionListener method.
- ActionListener → it is the listener interface for receiving action events. The class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the component's addActionListener method. When the action event occurs, that object's actionPerformed method is invoked.

```java
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

The class uses the imports of classes from the "View" package, which contains the description of Views/Windows, to be able to instantiate them.

```java
import View.AddView;
import View.BeqView;
import View.Dashboard;
import View.JumpView;
import View.LW_view;
import View.ORI_view;
import View.SW_view;
```

The class uses instances of Views as fields, in order to be able to construct the Views when the application is started, and to make each View visible when necessary.

```
17 public class MainController {
18
19     private Dashboard dashboard;
20     private AddView addView;
21     private BeqView beqView;
22     private JumpView jumpView;
23     private LW_view lwView;
24     private SW_view swView;
25     private ORI_view oriView;
```

The method “start()” creates the Views and makes them visible.

```
27     public void start() {
28
29         try {
30             UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
31         } catch (Exception e) {
32             e.printStackTrace();
33         }
34
35         dashboard = new Dashboard();
36         dashboard.setTitle("Dashboard Window");
37
38         addView = new AddView();
39         addView.setTitle("ADD operation");
40
41         beqView = new BeqView();
42         beqView.setTitle("BEQ operation");
43
44         jumpView = new JumpView();
45         jumpView.setTitle("JUMP operation");
46
47         lwView = new LW_view();
48         lwView.setTitle("LW operation");
49
50         swView = new SW_view();
51         swView.setTitle("SW operation");
52
53         oriView = new ORI_view();
54         oriView.setTitle("ORI operation");
55
56         initializeButtonListeners();
57
58         // DISPLAY THE FRAME FOR THE USER:
59         dashboard.setVisible(true);         // make visible the dashboard
60     }
```

In the try-catch statement is specified that the interface of the application will have the default theme, which is the Windows Stock Theme. Other custom themes could be applied. For example, if the application is executed on a Mac computer, the interface would be different, because the Operating System uses other icons and shapes for the UI elements.

Are created the windows, because the fields receive through assignation new instances of the corresponding Views, for example: “addView = new AddView()”. Each View, which is a window, has set the title.

In this method are also initialized the Listeners of the buttons, by calling the other method of the class “`initializeButtonListeners()`”.

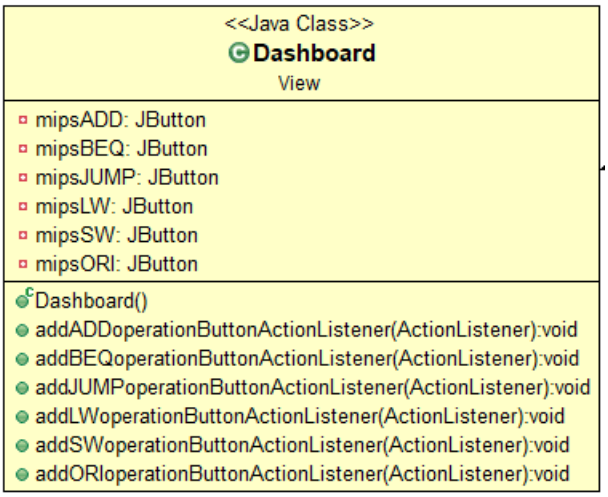
When the application starts, the Dashboard View Window is displayed. The Dashboard represents the main menu of the application, through which the user can choose the operation that must be illustrated, by pressing the corresponding button.

The method “`initializeButtonListeners()`” prepares the behavior of the application.

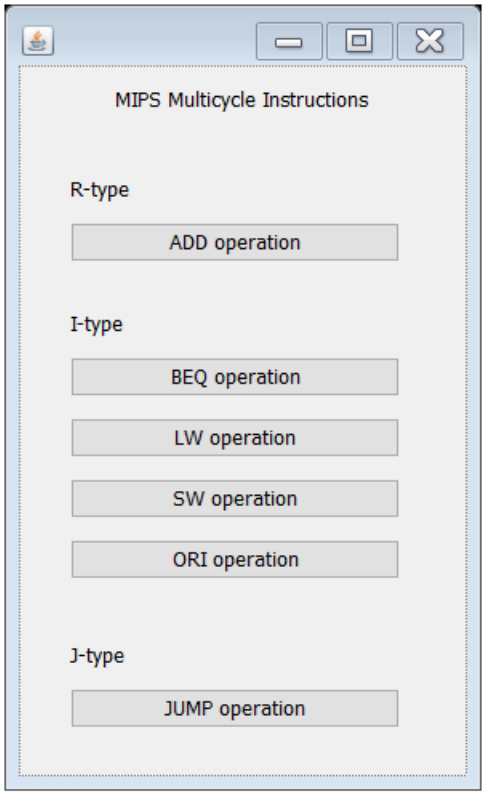
```
63      // prepare the logic (the behavior)
64      private void initializeButtonListeners() {
65
66          dashboard.addADDoperationButtonActionListener(new ActionListener() {
67              public void actionPerformed(ActionEvent e) {
68                  addView.setVisible(true);
69              }
70          });
71
72          dashboard.addBEQoperationButtonActionListener(new ActionListener() {
73              public void actionPerformed(ActionEvent e) {
74                  beqView.setVisible(true);
75              }
76          });
77
78          dashboard.addJUMPoperationButtonActionListener(new ActionListener() {
79              public void actionPerformed(ActionEvent e) {
80                  jumpView.setVisible(true);
81              }
82          });
83
84          dashboard.addSWoperationButtonActionListener(new ActionListener() {
85              public void actionPerformed(ActionEvent e) {
86                  swView.setVisible(true);
87              }
88          });
89
90          dashboard.addLWoperationButtonActionListener(new ActionListener() {
91              public void actionPerformed(ActionEvent e) {
92                  lwView.setVisible(true);
93              }
94          });
95
96          dashboard.addORIoperationButtonActionListener(new ActionListener() {
97              public void actionPerformed(ActionEvent e) {
98                  oriView.setVisible(true);
99              }
100         });
101
102     }
103
104 }
```

The method contains the six action listeners that specify the behavior when a certain button is pressed. If a button for an operation is pressed in Dashboard Window, then the corresponding View (Window) will be displayed: `…operation…View.setVisible(true);`

3. Class "Dashboard"



Is the main View of the application and represents the main menu of the application, from which the user can choose what operation to be illustrated (what Window to be displayed).



The class uses the UI elements from the Java SWING library, such as JButton, JLable, JFrame. The class extends the JFrame class in order to make it displayable and to add UI elements to it.

```
9  public class Dashboard extends JFrame{
10
11      private JButton mipsADD;
12      private JButton mipsBEQ;
13      private JButton mipsJUMP;
14      private JButton mipsLW;
15      private JButton mipsSW;
16      private JButton mipsORI;
```

The fields of the class are instances of UI elements mentioned above.

```
18      public Dashboard() {
19
20          this.setBounds(150, 190, 298, 491);     // initial location to be rendered: x,y, width, heights
21          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22          getContentPane().setLayout(null);
23
24          mipsADD = new JButton("ADD operation");
25          mipsADD.setBounds(32, 98, 206, 25);
26          getContentPane().add(mipsADD);
27
28          mipsBEQ = new JButton("BEQ operation");
29          mipsBEQ.setBounds(32, 182, 206, 25);
30          getContentPane().add(mipsBEQ);
31
32          mipsJUMP = new JButton("JUMP operation");
33          mipsJUMP.setBounds(32, 389, 206, 25);
34          getContentPane().add(mipsJUMP);
35
36          mipsLW = new JButton("LW operation");
37          mipsLW.setBounds(32, 220, 206, 25);
38          getContentPane().add(mipsLW);
39
40          mipsSW = new JButton("SW operation");
41          mipsSW.setBounds(32, 258, 206, 25);
42          getContentPane().add(mipsSW);
43
44          mipsORI = new JButton("ORI operation");
45          mipsORI.setBounds(32, 296, 206, 25);
46          getContentPane().add(mipsORI);
47
48          JLabel lblRtypeInstruction = new JLabel("R-type");
49          lblRtypeInstruction.setBounds(32, 69, 44, 16);
50          getContentPane().add(lblRtypeInstruction);
51
52          JLabel lblItypeInstructions = new JLabel("I-type");
53          lblItypeInstructions.setBounds(32, 153, 44, 16);
54          getContentPane().add(lblItypeInstructions);
55
56          JLabel lblMipsMulticycleInstructions = new JLabel("MIPS Multicycle Instructions");
57          lblMipsMulticycleInstructions.setBounds(60, 13, 163, 16);
58          getContentPane().add(lblMipsMulticycleInstructions);
59
60          JLabel lblJtype = new JLabel("J-type");
61          lblJtype.setBounds(32, 360, 44, 16);
62          getContentPane().add(lblJtype);
63      }
```

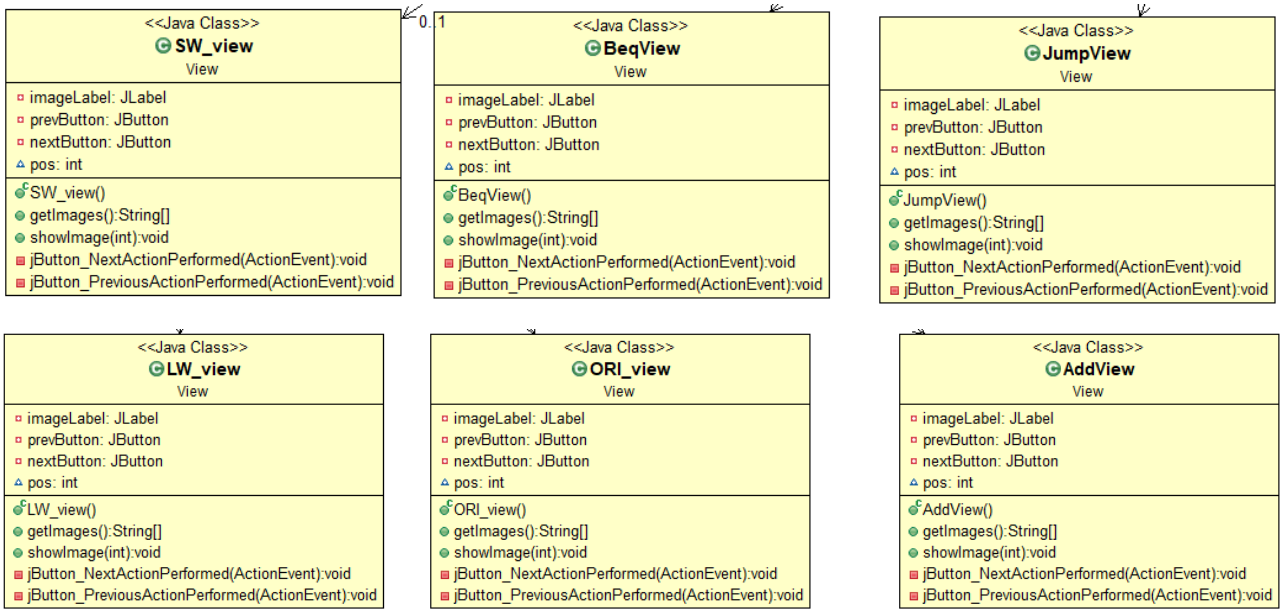
The constructor of the class, “Dashboard()” specifies:

- The coordinates X and Y where the window should be drawn on the screen and the Width and Height of the window
  `this.setBounds(150, 190, 298, 491);`
- specifies that the process of the application will be finished when the user closes this window, not only hiding it
  `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`
- All windows have the layout set to null. By setting the layout to Null, the graphical elements are drawn only by specifying the coordinates and the dimension with the “SetBounds” method.
  `getContentPane().setLayout(null);`
- The coordinates of the UI elements (buttons, labels) and their dimension, and adding them to the Content Pane: `getContentPane().add(component);`

The class contains finally the action listener methods for the buttons, called from the Controller.

```
65      // action listeners for the buttons
66      public void addADDoperationButtonActionListener(final ActionListener e) {
67          mipsADD.addActionListener(e);
68      }
69
70      public void addBEQoperationButtonActionListener(final ActionListener e) {
71          mipsBEQ.addActionListener(e);
72      }
73
74      public void addJUMPoperationButtonActionListener(final ActionListener e) {
75          mipsJUMP.addActionListener(e);
76      }
77
78      public void addLWoperationButtonActionListener(final ActionListener e) {
79          mipsLW.addActionListener(e);
80      }
81
82      public void addSWoperationButtonActionListener(final ActionListener e) {
83          mipsSW.addActionListener(e);
84      }
85
86      public void addORIoperationButtonActionListener(final ActionListener e) {
87          mipsORI.addActionListener(e);
88      }
89  }
```

4. Classes of type "OperationView": AddView, BeqView, JumpView, LW_view, SW_view, ORI_view



All these classes have identical implementation, the difference between them being the path to the location of images that must be shown into the window.

The "View" classes also use the UI elements from the Java SWING library, such as JButton, JLabel, JFrame. Also, the classes use libraries `java.awt.Image`, `java.io.File`, `javax.swing.ImageIcon` in order to access and display images.S

The classes also extend the JFrame class in order to make them displayable and to add UI elements to it.

```
1  package View;
2
3  import java.awt.Image;
4  import java.awt.event.ActionListener;
5  import java.io.File;
6
7  import javax.swing.ImageIcon;
8  import javax.swing.JButton;
9  import javax.swing.JFrame;
10 import javax.swing.JLabel;

12 public class AddView extends JFrame {
13
14     private JLabel imageLabel;
15     private JButton prevButton;
16     private JButton nextButton;
17
18     int pos = 0;     // index of the first image
```

The fields of the class are instances of UI elements. Also is used a variable “pos” which represents the index of the first shown image.

```
20    public AddView() {
21
22        this.setBounds(450, 190, 1441, 731); // the initial location where it should be rendered, last 2 parameters
23
24        getContentPane().setLayout(null);
25
26        imageLabel = new JLabel("");
27        imageLabel.setBounds(430, 0, 993, 619);
28        getContentPane().add(imageLabel);
29
30
31        prevButton = new JButton("<  Prev");
32        prevButton.setBounds(802, 646, 97, 25);
33        getContentPane().add(prevButton);
34        prevButton.addActionListener(new java.awt.event.ActionListener() {
35            public void actionPerformed(java.awt.event.ActionEvent evt) {
36                jButton_PreviousActionPerformed(evt);
37            }
38        });
39
40
41        nextButton = new JButton("Next  >");
42        nextButton.setBounds(966, 646, 97, 25);
43        getContentPane().add(nextButton);
44        nextButton.addActionListener(new java.awt.event.ActionListener() {
45            public void actionPerformed(java.awt.event.ActionEvent evt) {
46                jButton_NextActionPerformed(evt);
47            }
48        });
49
50
51        // adding fixed image
52        JLabel imageLabel = new JLabel();
53        Image img = new ImageIcon(this.getClass().getResource("/images/fixed_images/add_img0.jpg")).getImage();
54        imageLabel.setIcon(new ImageIcon(img));
55        imageLabel.setBounds(0, 0, 418, 619);
56        getContentPane().add(imageLabel);
57
58
59        showImage(pos);     // display the first image from the folder when the window is opened
60    }
61    // until now, was specified just how the frame will look like when will be initialized
```

The constructor for each of the “View” classes have described the size of the window and the position where it should be rendered, and also the position and dimension of the UI elements. Also, the constructor contains the calls of the Action Listeners assigned to the PREV and NEXT buttons. In the constructor is also added a fixed image into a JLabel, by saving into a variable the image given with the path:

```
ImageIcon(this.getClass().getResource("/images/fixed_images/add_img0.jpg")).getImage();
```

On the JLabel is applied method `setIcon(new ImageIcon(img));` in order to add the saved image from the variable “img” into the JLabel. Finally, the JLabel is added to the Content Pane (window). This fixed image represents the steps executed for each instruction in an RTL Concrete table.

In the constructor is called the method `showImage(pos);` in order to display automatically the first image from the folder when the window is rendered (when the controller makes it visible).

→ The method `getImages` goes to the folder given as parameter and retrieves the name for all images and places them into the list of String called "imagesList". The method returns the list of names of the images.

```
65      public String[] getImages() {
66          File file = new File(getClass().getResource("/images/add_operation/").getFile());
67          String[] imagesList = file.list();
68          return imagesList;
69      }
```

→ The method `showImage` uses the previous method `getImages` . The method has as parameter the index of the image, from the list of names of images returned by `getImages` method. The method displays the image with the given index. The method "getScaleInstance" forces the image to fit into the dimension of the JLabel of the window. The method extracts and displays all the images from the given folder.

```
75      public void showImage(int index) {
76          String[] imagesList = getImages();
77          String imageName = imagesList[index];
78          ImageIcon icon = new ImageIcon(getClass().getResource("/images/add_operation/" + imageName));
79          Image image = icon.getImage().getScaledInstance(imageLabel.getWidth(), imageLabel.getHeight(), Image.SCALE_SMOOTH);
80          imageLabel.setIcon(new ImageIcon(image));
81      }
```

→ The method `jButton_NextActionPerformed` contains the implementation of the behavior when the NEXT button is pressed. The index of the image is incremented and is displayed into the JLabel. The if statement verifies if the index is at the last image, then it stops. Therefore nothing happens when NEXT button is pressed and the last image is shown.

```
83      private void jButton_NextActionPerformed(java.awt.event.ActionEvent evt) {
84          pos = pos + 1;
85          if (pos >= getImages().length) {
86              pos = getImages().length - 1;
87          }
88          showImage(pos);
89      }
```

→ The method `jButton_PreviousActionPerformed` contains the implementation of the behavior when the PREV button is pressed. The index of the image is decremented and is displayed into the JLabel. The if statement verifies if the index is at the first image, and if it is, then it stops, because is not needed to further decrement the index to a negative value. Therefore nothing happens when the PREV button is pressed and the first image is shown.

```
92    private void jButton_PreviousActionPerformed(java.awt.event.ActionEvent evt) {
93        pos = pos - 1;
94        if (pos < 0) {
95            pos = 0;
96        }
97        showImage(pos);
98    }
```

Observation:

The Dashboard is used to select what operation to be illustrated. The action listeners for its buttons have the behavior implemented into the MainController.

The View classes have the action listeners implemented internally, because each class uses images from a different folder.

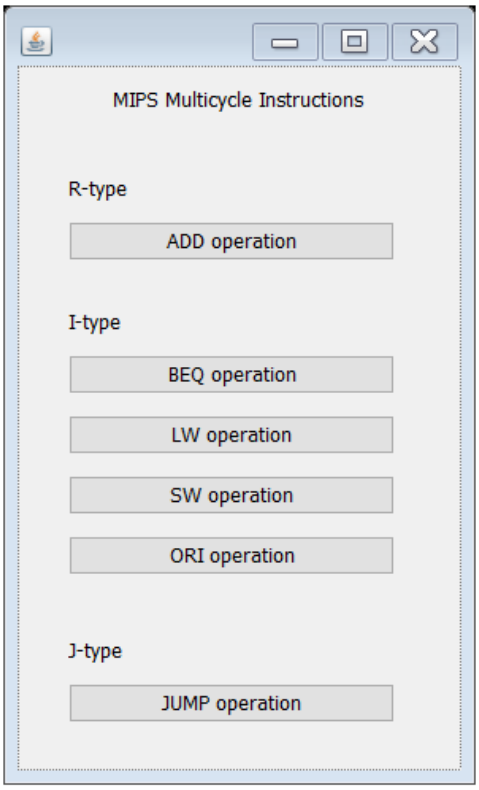
5. Testing (experimental results) and Conclusions

5.1. Testing

In order to test if the application behaves correctly, I have run the Application.java Class, which represents the main class of the application. The Dashboard Window is displayed, being the main menu of the application, from where the user can choose what operation to be viewed by pressing the corresponding button.

When the user closes the Dashboard Window, the entire application stops running, because is the main window of the application. Therefore, all the opened "Operation" windows will be also closed. If the user closes an "Operation" window, then the application will continue to run, since these windows are secondary windows. The user can open and close an "Operation" window as many times the user wants. Also, can be opened and viewed simultaneously all the six "Operation" windows.

The interface of the Dashboard window behaves correctly, because the correct "Operation" window is displayed when the user presses a button associated to an operation.



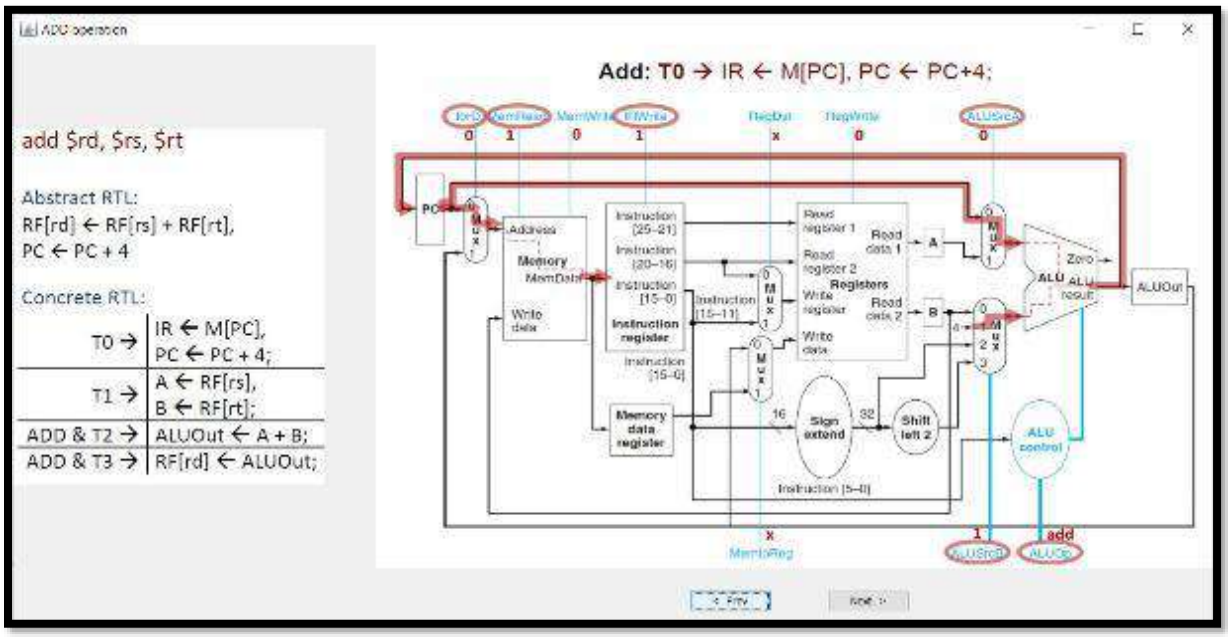
➔ The **operation windows** behave correctly, because in the left part is displayed the correct image into a JLabel. The fixed image presents the RTL (Register Transfer Level) Concrete[11] of the operation, and the RTL Abstract. Also, on the right part of the window are displayed correctly the first image and the navigation buttons.

➔ The buttons work correctly, because if the user tries to navigate to the previous image when the first image is displayed, then nothing changes (first image will be still displayed). When the user tries to navigate to the next image, the following image is displayed into the right JLabel of the window. However, when is displayed the last image, pressing the NEXT button will produce the correct behavior, since the last image will be still displayed. Therefore, user can navigate only from first image to last image.

➔ The images displayed in the right JLabel highlight the datapath and the control signals that are active in order to allow the data to pass, representing the RTL Concrete.

***R-type instruction*: ADD operation Window** (add into the Destination register, the content of the Source and Target registers)

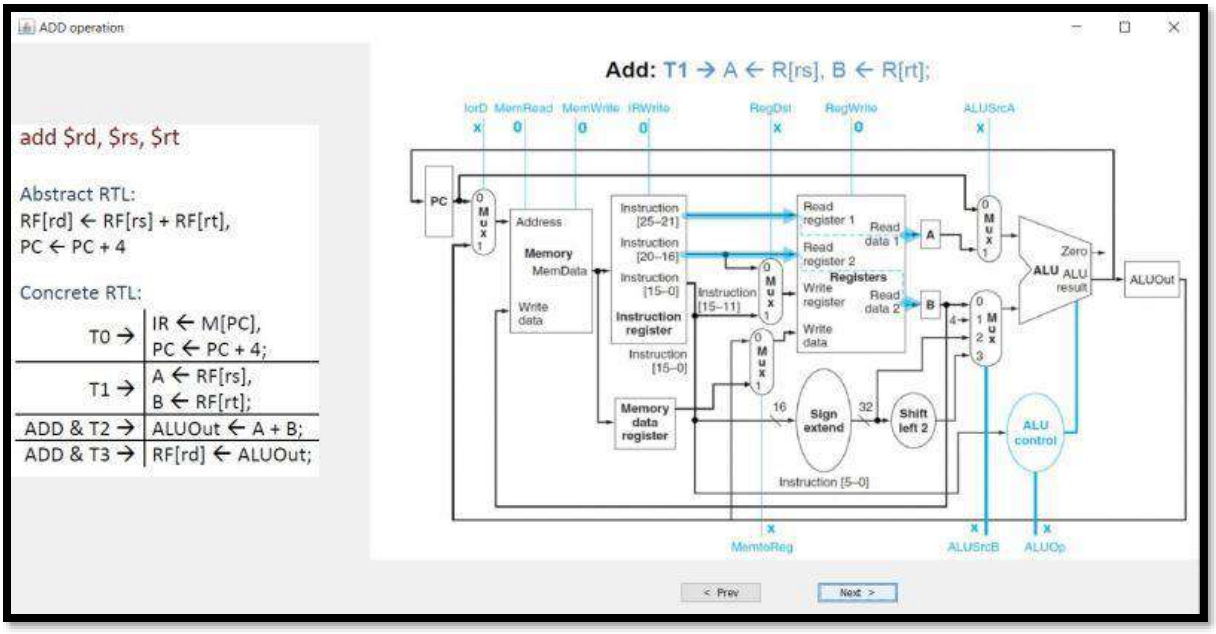
- **First image**



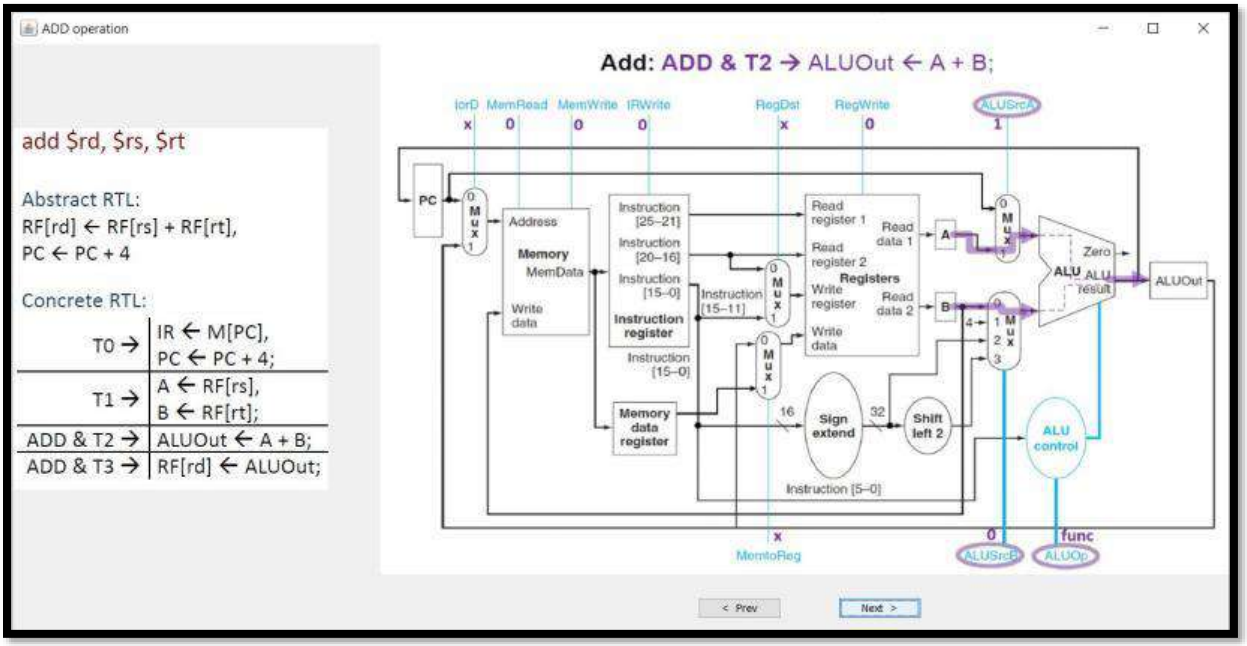
➔ At time T0, the Program Counter (PC) is passed to the Memory Unit as an address, by making the select signal IorD for the first Multiplexer to be '0' in order to allow PC to pass and the control signal MemRead is activated with value '1' in order to allow storage at address PC. The content of the Memory Unit at address pointed by the value of PC will be assigned to the Instruction Register (IR) by activating the IRWrite control signal with '1'. The Register File Unit is inactive, since control signal RegWrite is '0'. Because the control signals ALUsrcA='0' and ALUsrcB ='1' and ALUop='add', the PC is incremented with 4 in the Arithmetic and Logic Unit (ALU). Operation "PC ← PC + 4" means that after the current instruction is finished to be executed, the next instruction will begin execution.

- **Second image**

➔ At time T1, in the next Clock cycle, the RS (source register = Instruction[25..21]) and RT (target register = Instruction[20..16]) from Instruction Register Unit are read into the Register File Unit and are passed to the registers A and B, respectively.

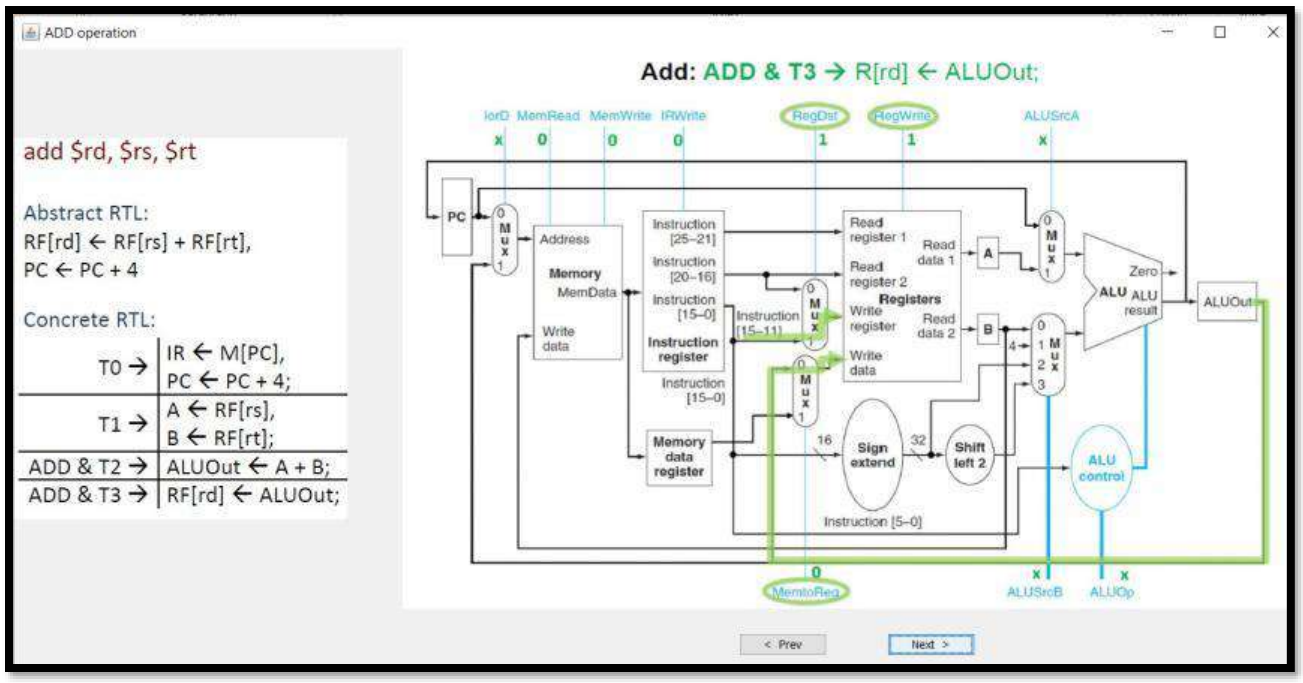


- **Third image**



➔ At time T3, in the next CLK cycle, ALUout register receives as input the result of addition of the content of A and B registers, provided by the output of the ALU. Control signals have values: ALUsrcA=’1’ and ALUsrcB=’0’ in order to pass the A and B registers to inputs of the ALU.

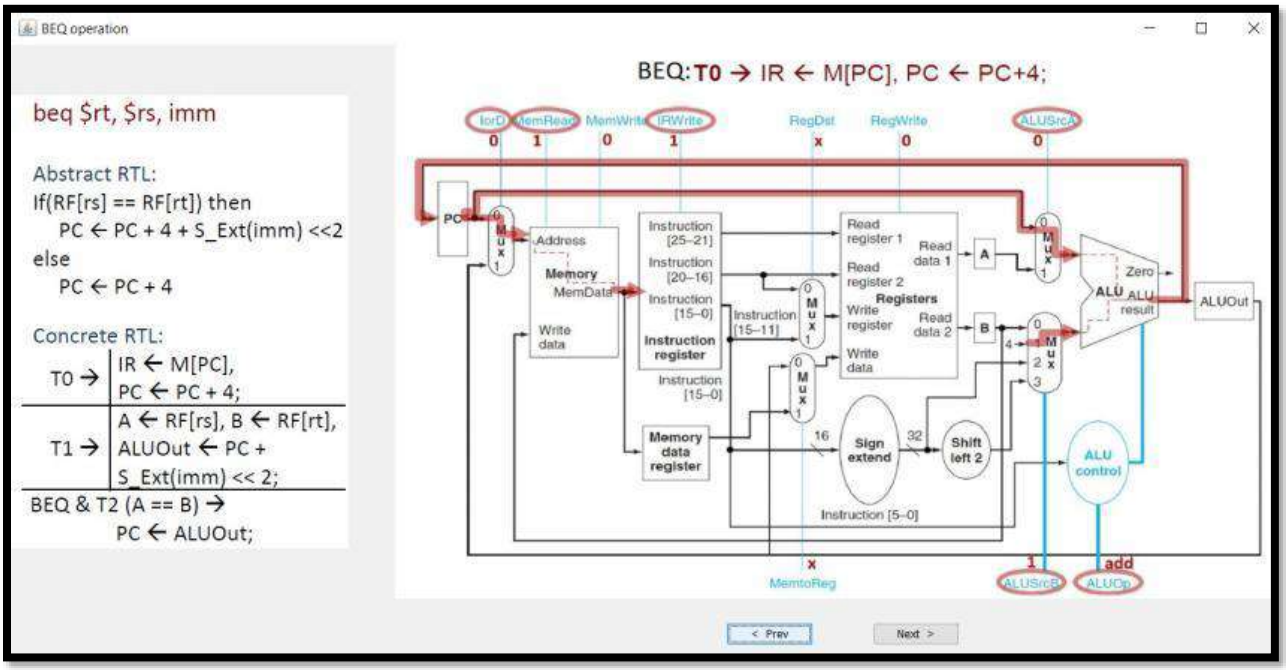
- **Forth image**



➔ At time T3, the final CLK cycle, content of the ALUout register is passed to the RD (destination register) address from the Register File Unit. The “Write register” input port of the RF is the RD, to which is assigned the content of ALUout passed through the input port “Write data” of RF (Write register ← Write data). In order to select RD, RegDst is ‘1’. In order to pass content of ALUout to “Write data” port, the MemToReg is ‘0’. Control signal “RegWrite” is activated with ‘1’ in order to assign to “Write register” (RD) the “Write data” (ALUout).

***I-type instruction*: BEQ operation Window** (if the content of the Source and Target registers are identical, then the PC receives a value, else go to next instruction (execute next instruction))

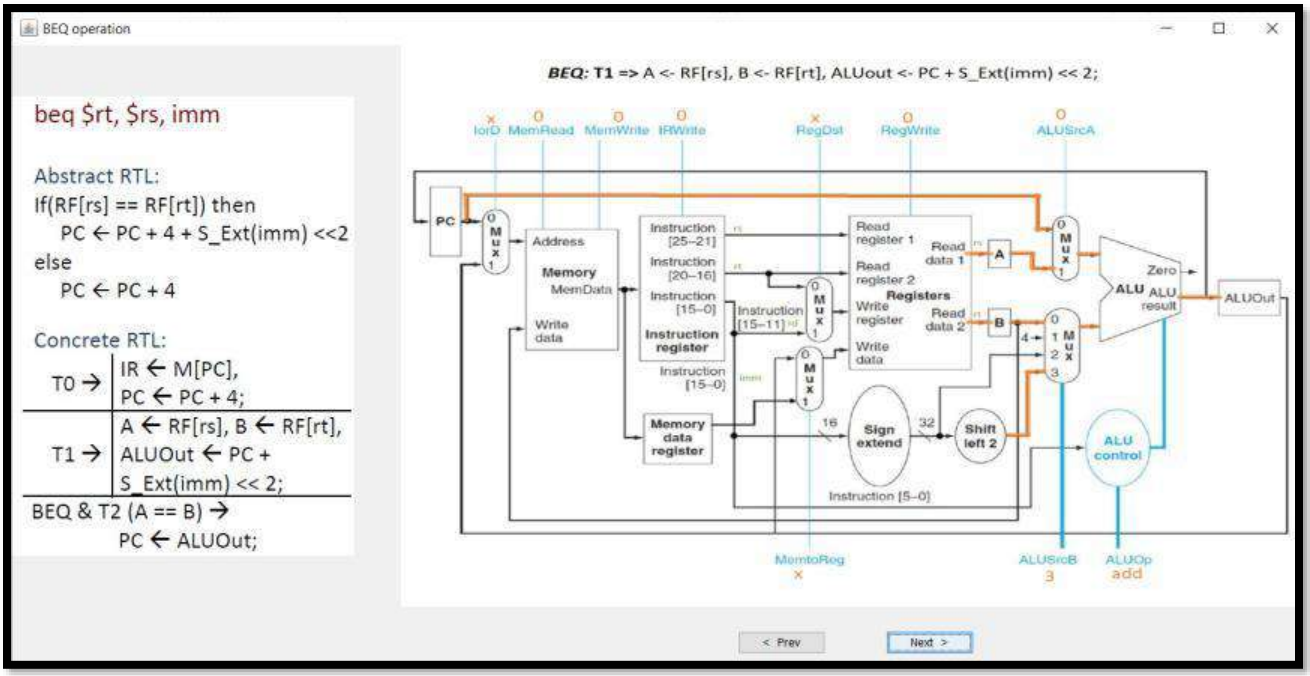
- **First image**



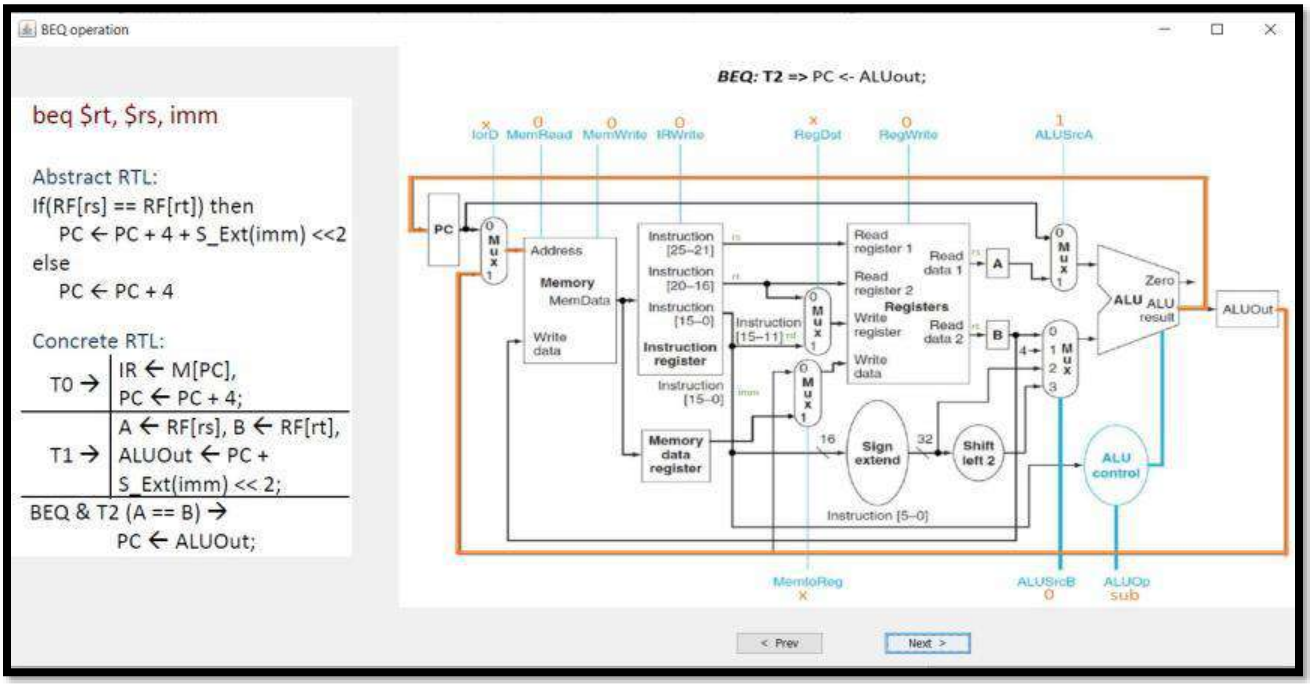
➔ At time T0, the Program Counter (PC) is passed to the Memory Unit as an address, by making the select signal IorD for the first Multiplexer to be '0' in order to allow PC to pass and the control signal MemRead is activated with value '1' in order to allow storage at address PC. The content of the Memory Unit at address pointed by the value of PC, will be assigned to the Instruction Register (IR) by activating the IRWrite control signal with '1'. The Register File Unit is inactive, since control signal RegWrite is '0'. Because the control signals ALUsrcA='0' and ALUsrcB ='1' and ALUop='add', the PC is incremented with 4 in the Arithmetic and Logic Unit (ALU). Operation "PC ← PC + 4" means that after the current instruction is finished to be executed, the next instruction will begin execution.

- **Second image**

➔ At time T1, in the next Clock cycle, the RS (source register = Instruction[25..21]) and RT (target register = Instruction[20..16]) from Instruction Register Unit are read into the Register File Unit and are passed to the registers A and B, respectively. Consequently, ALUout register receives as input the result of the addition between the PC and the Immediate Value shifted to left with two positions. This is done because ALUsrcA='0' and ALUsrcB='3' in order to allow data to pass to the inputs of the ALU, which computes Addition Operation as specified by the ALUop control signal.



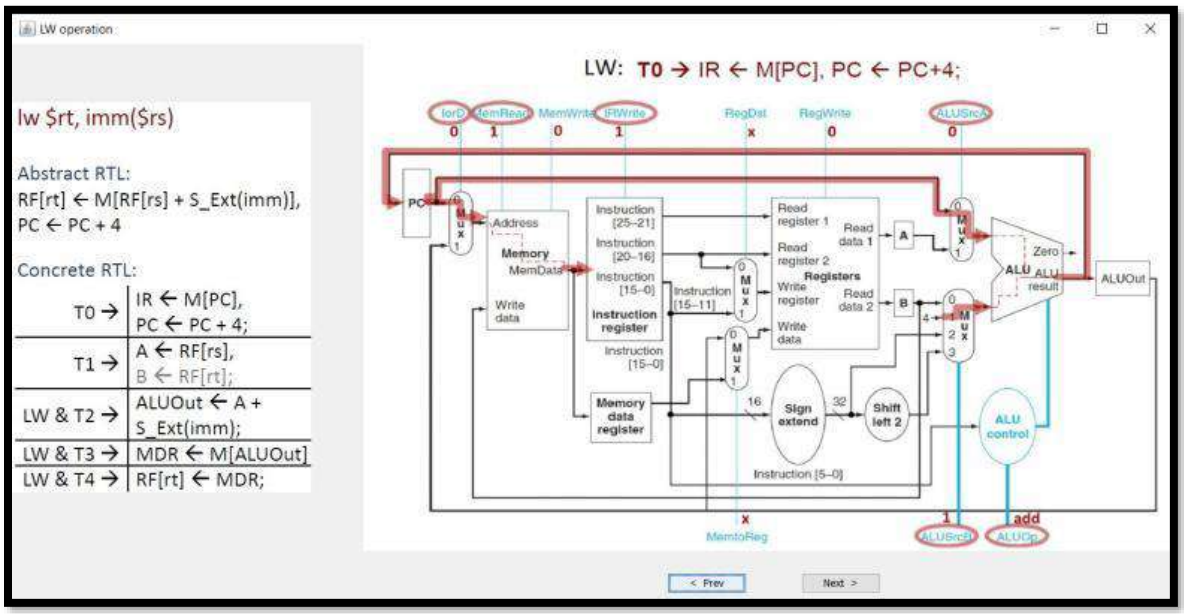
- **Third image**



➔ In the final CLK cycle, time T2, the content of the register ALUout is stored in the Program Counter (PC) in order to complete the execution of the instruction. This is done by setting the control signals ALUsrcA=’1’ and ALUsrcB=’0’ and ALUop=’sub’.

***I-type instruction*: LW operation Window** (load the content of the Memory from the specified address into the target register, then go to next instruction)

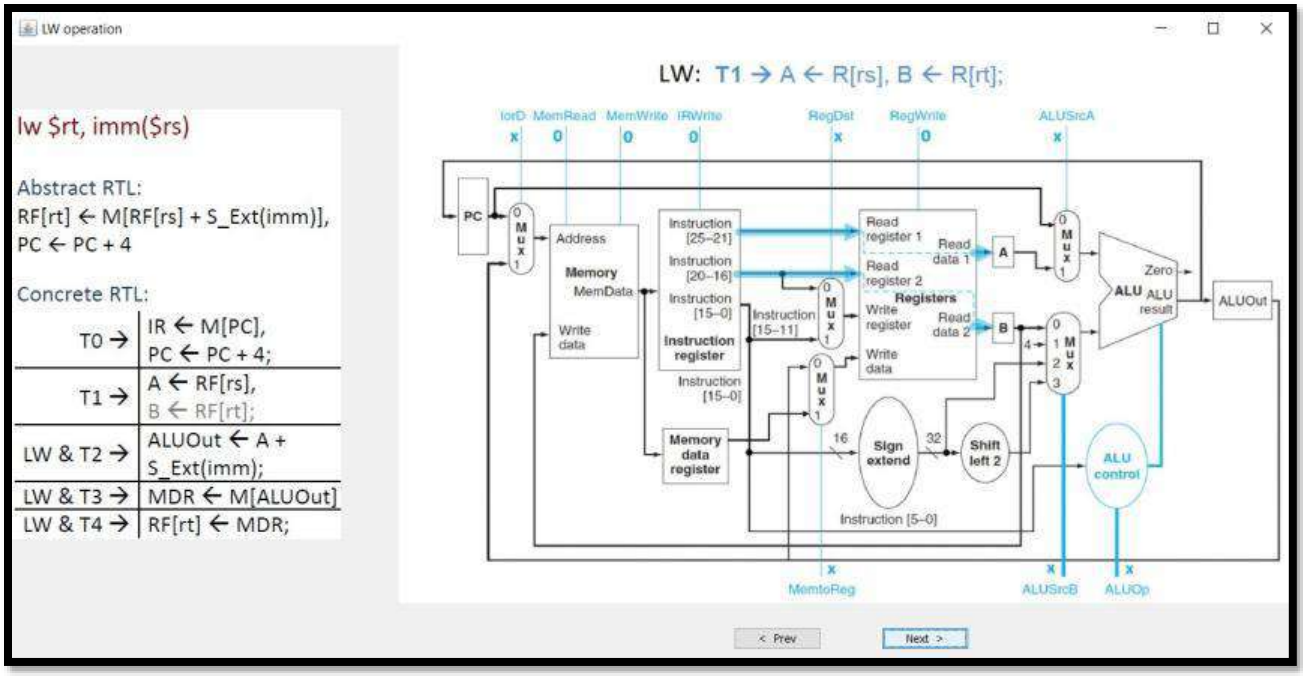
- **First image**



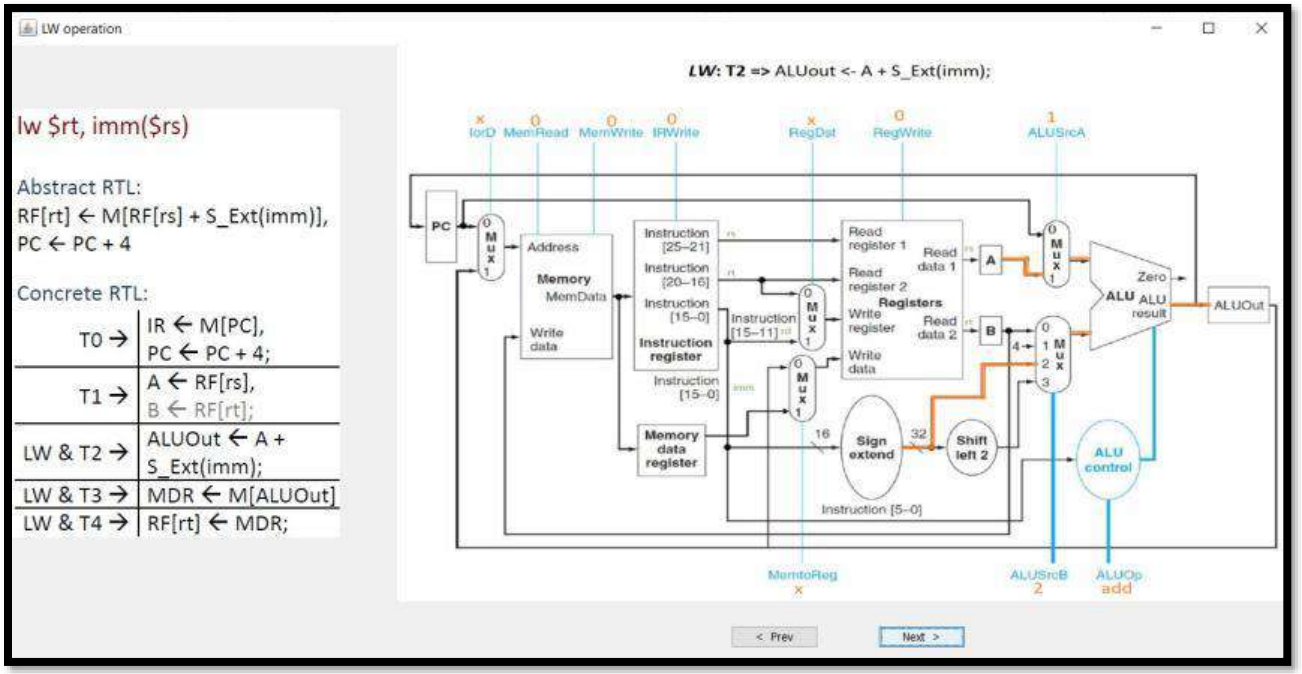
➔ At time T0, the Program Counter (PC) is passed to the Memory Unit as an address, by making the select signal IorD for the first Multiplexer to be '0' in order to allow PC to pass and the control signal MemRead is activated with value '1' in order to allow storage at address PC. The content of the Memory Unit at address pointed by the value of PC, will be assigned to the Instruction Register (IR) by activating the IRWrite control signal with '1'. The Register File Unit is inactive, since control signal RegWrite is '0'. Because the control signals ALUsrcA='0' and ALUsrcB ='1' and ALUop='add', the PC is incremented with 4 in the Arithmetic and Logic Unit (ALU). Operation "PC ← PC + 4" means that after the current instruction is finished to be executed, the next instruction will begin execution.

- **Second image**

➔ At time T1, in the next Clock cycle, the RS (source register = Instruction[25..21]) and RT (target register = Instruction[20..16]) from Instruction Register Unit are read into the Register File Unit and are passed to the registers A and B, respectively.



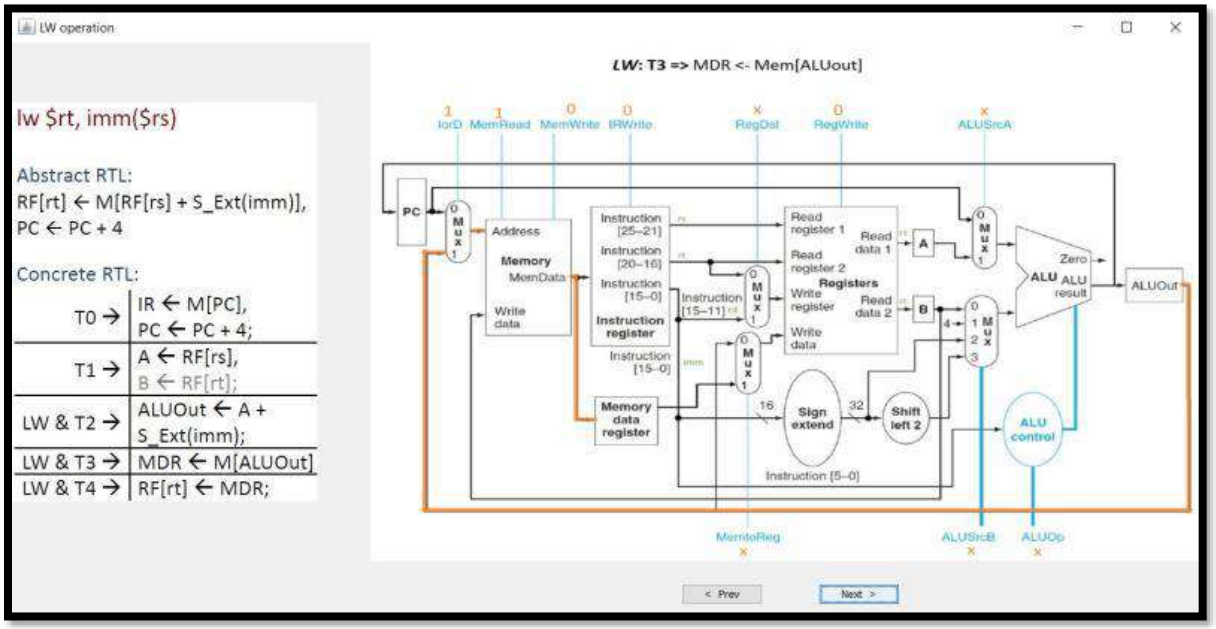
- **Third image**



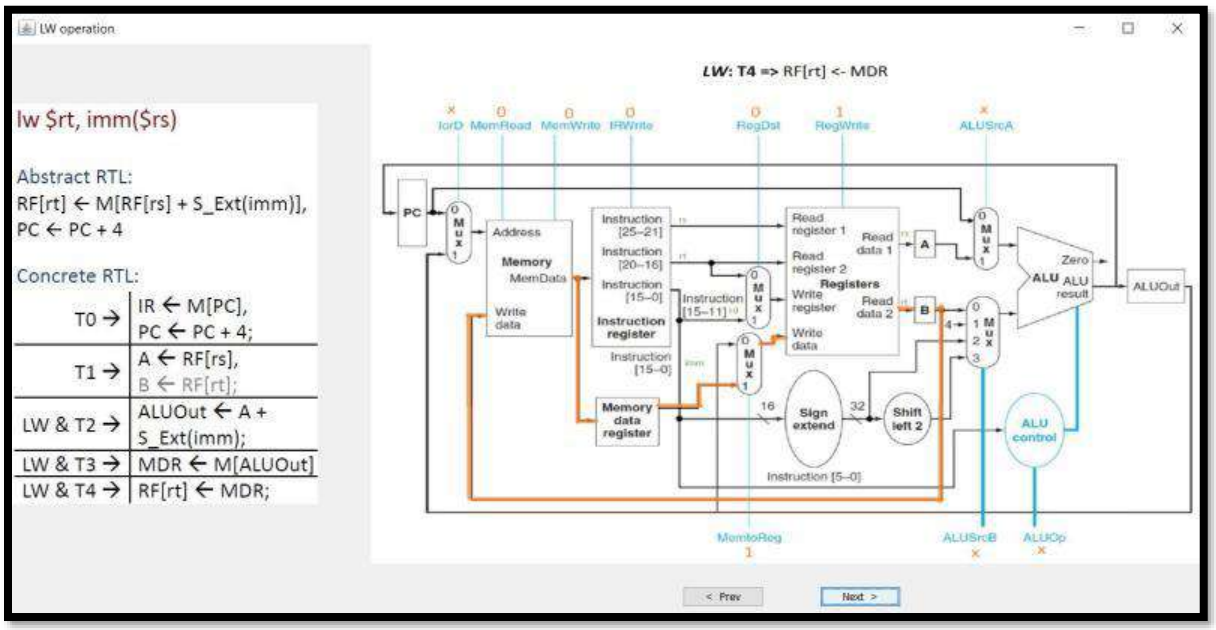
➔ At time T2, the next CLK cycle, the ALU performs the addition between the content of the A register and the Signed-extended version of the Instruction, storing the result into the ALUout register. This is done by setting the control signals: AlusrcA=’1’ and ALUsrcB=’2’ and ALUop=’add’.

- **Forth image**

➔ At time T3, the content of the ALUout is stored into the Memory Unit and passed to the input of the Memory Data Register (MDR). This is done by activating the control signals: IorD=’1’ and MemRead=’1’.



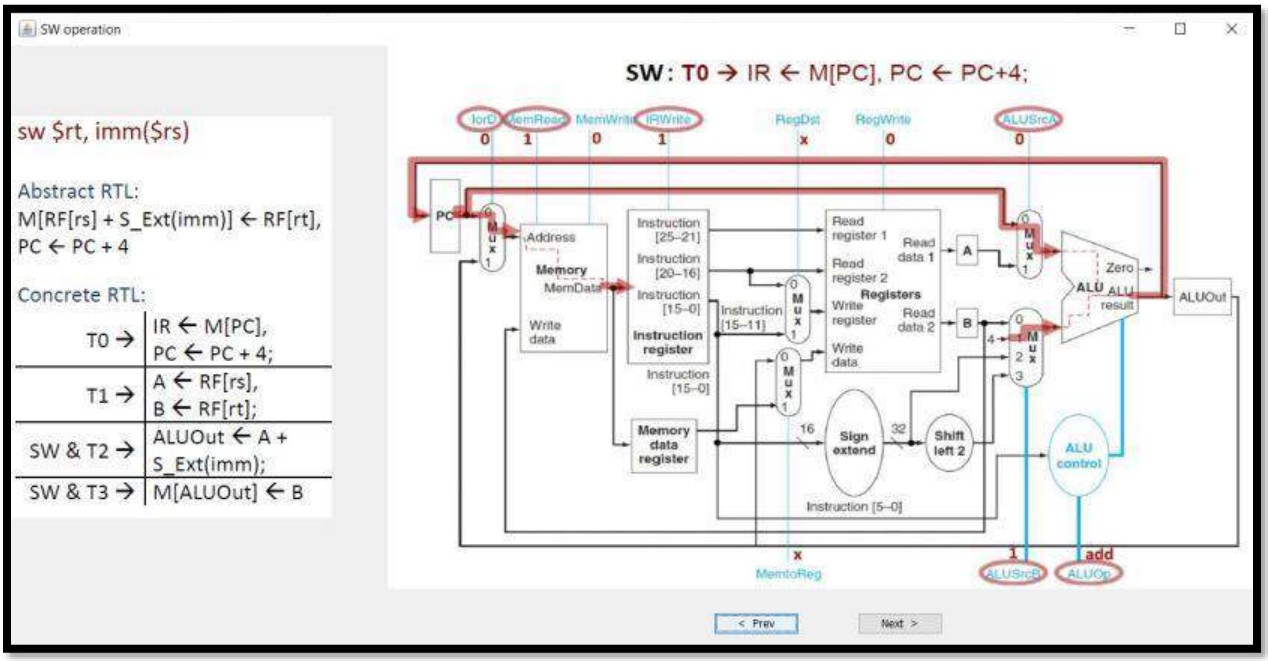
- **Fifth image**



➔ In the last CLK cycle, at time T4, the content of the MDR is passed to register RT of the Register File Unit (to input “Write data” of the RF). This is done by setting the control signals: RegDst=’0’ and MemToReg=’1’ and RegWrite=’1’.

***I-type instruction*: SW operation Window** (store the content of the register into the Memory at the specified address, then go to next instruction)

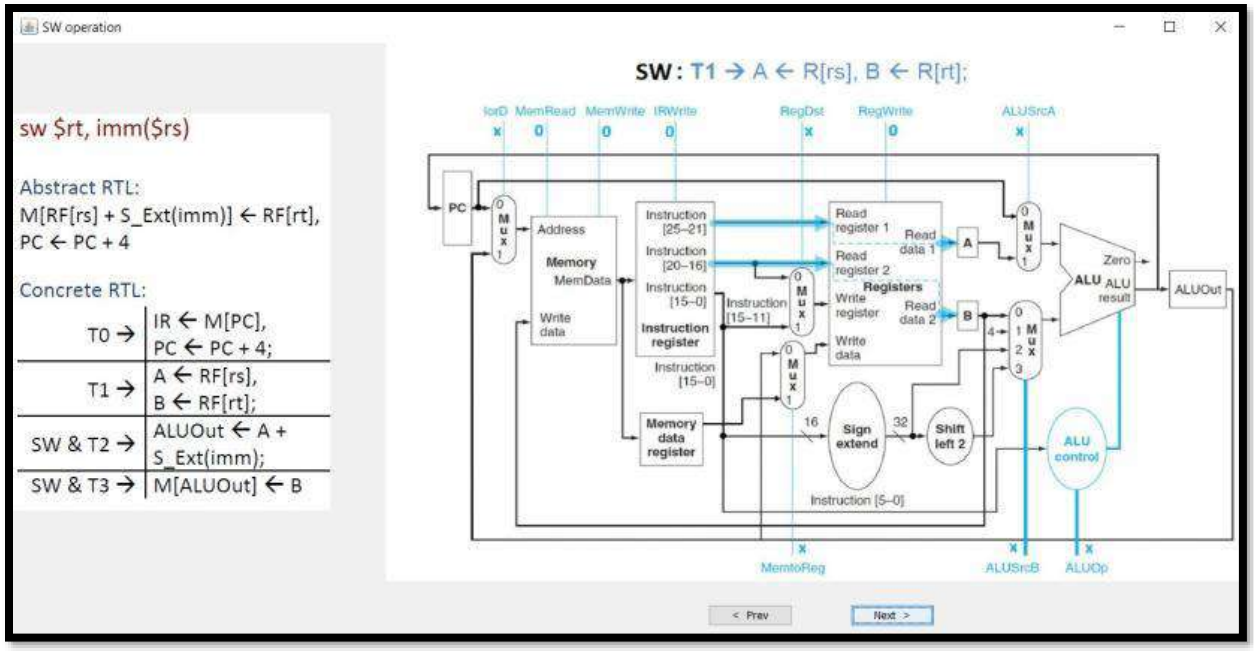
- **First image**



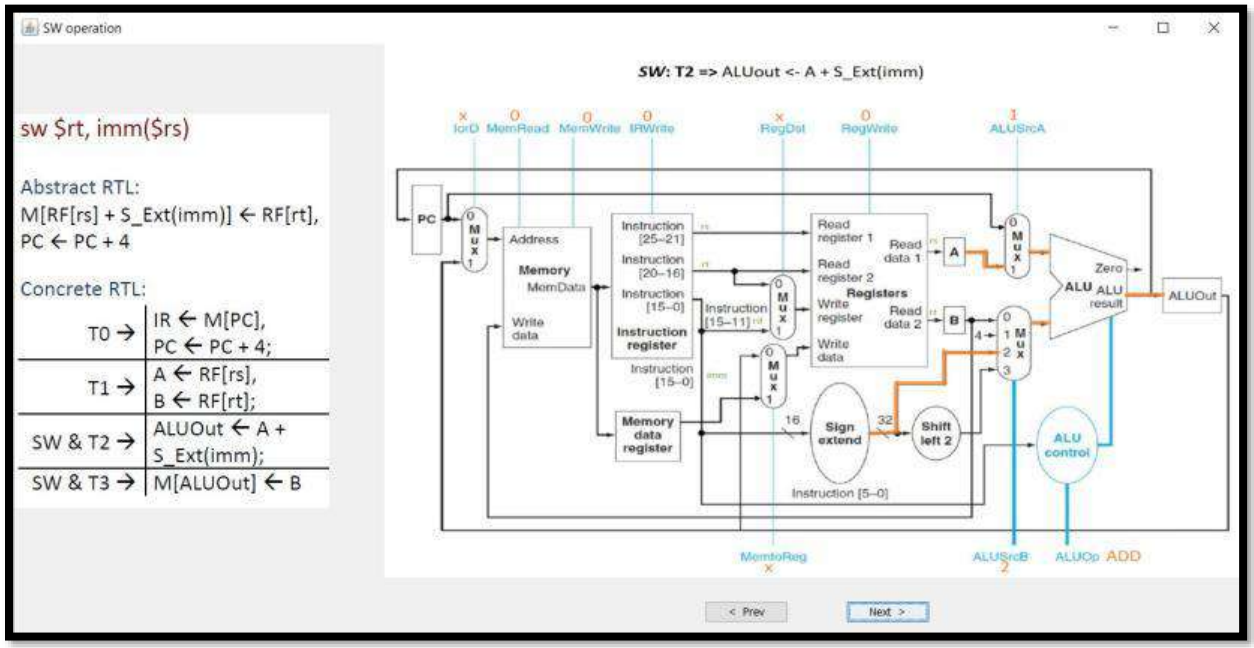
➔ At time T0, the Program Counter (PC) is passed to the Memory Unit as an address, by making the select signal IorD for the first Multiplexer to be '0' in order to allow PC to pass and the control signal MemRead is activated with value '1' in order to allow storage at address PC. The content of the Memory Unit at address pointed by the value of PC, will be assigned to the Instruction Register (IR) by activating the IRWrite control signal with '1'. The Register File Unit is inactive, since control signal RegWrite is '0'. Because the control signals ALUsrcA='0' and ALUsrcB ='1' and ALUop='add', the PC is incremented with 4 in the Arithmetic and Logic Unit (ALU). Operation "PC ← PC + 4" means that after the current instruction is finished to be executed, the next instruction will begin execution.

- **Second image**

➔ At time T1, in the next Clock cycle, the RS (source register = Instruction[25..21]) and RT (target register = Instruction[20..16]) from Instruction Register Unit are read into the Register File Unit and are passed to the registers A and B, respectively.

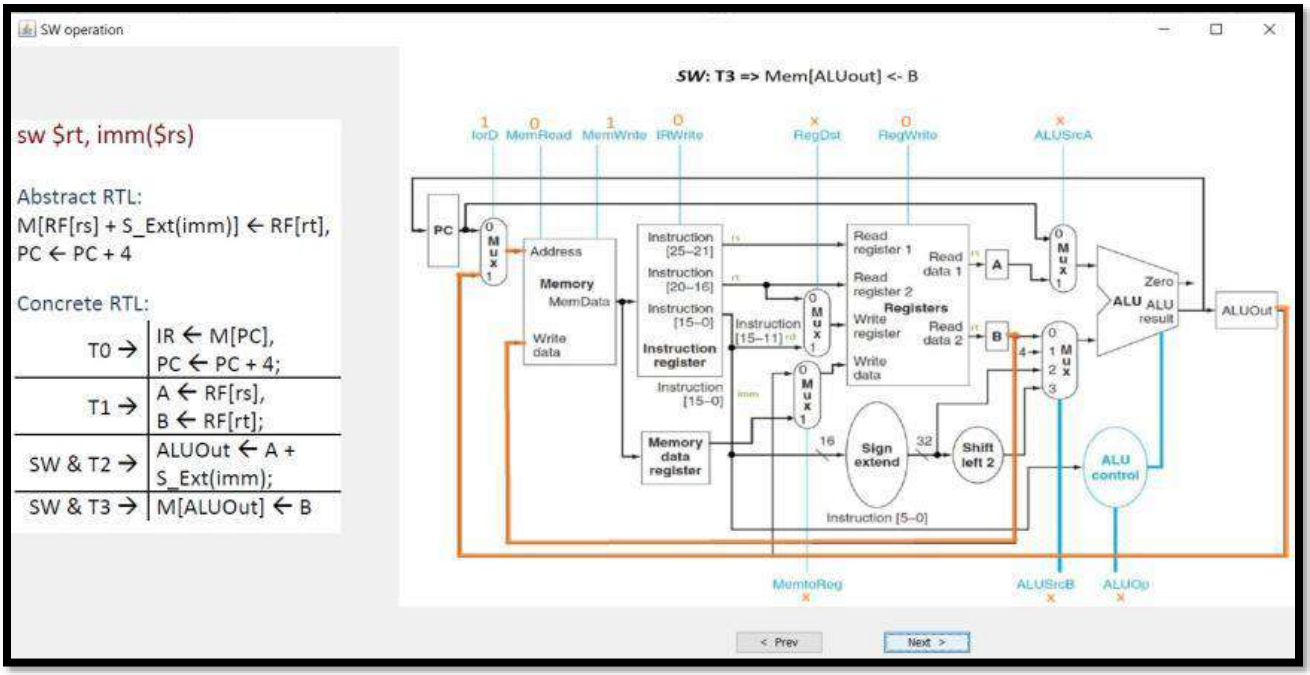


- **Third image**



➔ At time T2, the next CLK cycle, the ALU performs the addition between the content of the A register and the Signed-extended version of the Instruction (output of the IR Unit), storing the result into the ALUout register. This is done by setting the control signals: AlusrcA=’1’ and ALUsrcB=’2’ and ALUop=’add’.

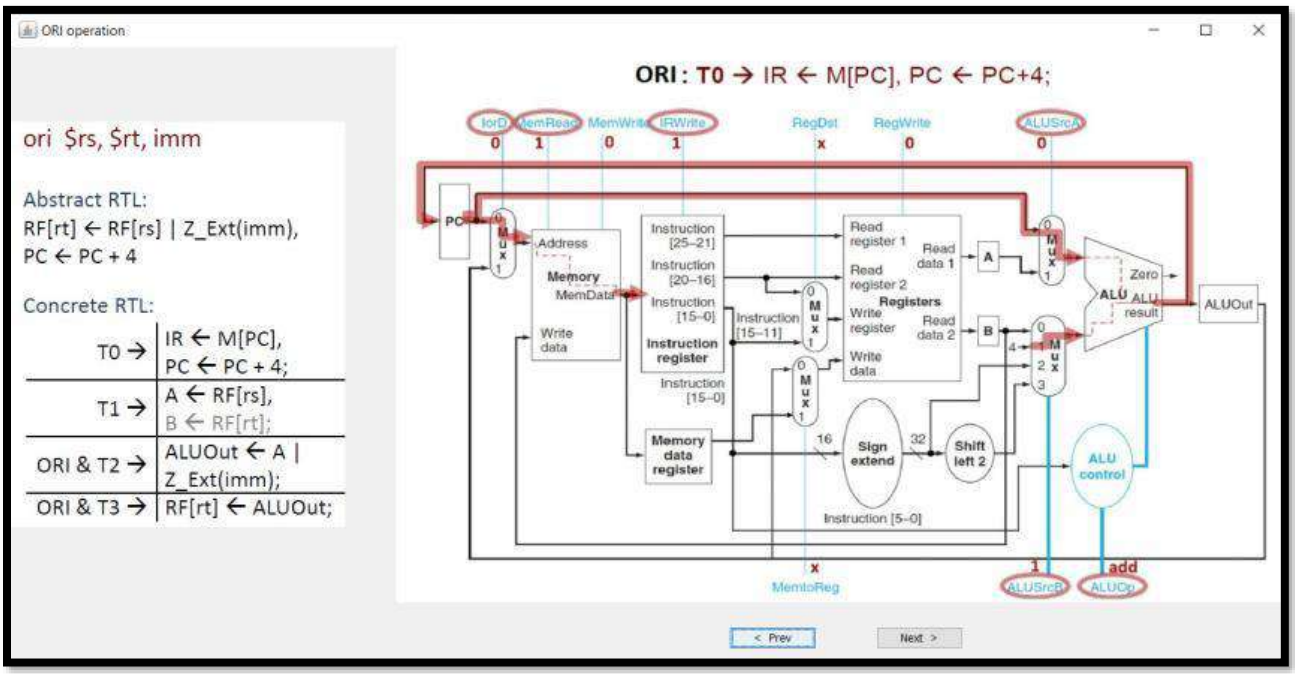
- **Forth image**



➔ In the final CLK cycle, time T3, the content of the B register is assigned to the address of the Memory unit pointed by the value of the ALUout register. This is done by setting the control signals: IorD=’1’ and MemWrite=’1’.

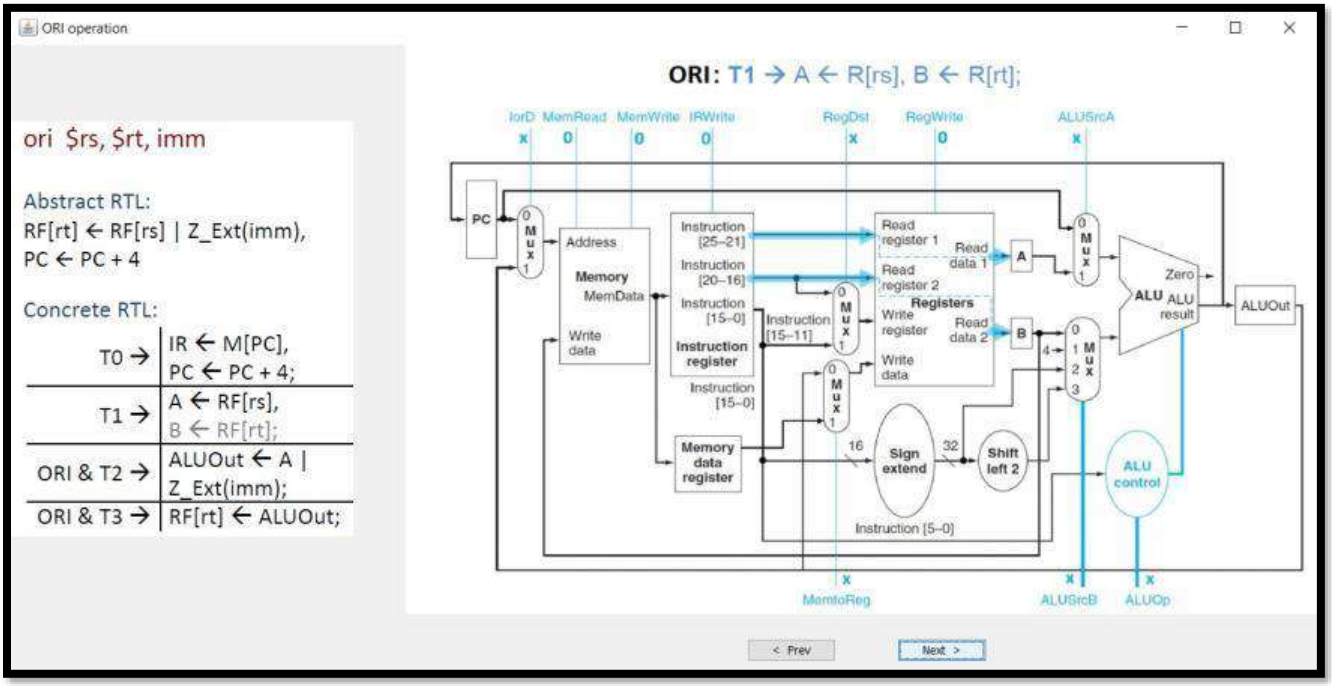
***I-type instruction*: ORI operation Window** (performs the OR operation between the value of the Source register RS and the given Immediate Value, and storing the result into the Register RT)

- **First image**



➔ At time T0, the Program Counter (PC) is passed to the Memory Unit as an address, by making the select signal IorD for the first Multiplexer to be '0' in order to allow PC to pass and the control signal MemRead is activated with value '1' in order to allow storage at address PC. The content of the Memory Unit at address pointed by the value of PC, will be assigned to the Instruction Register (IR) by activating the IRWrite control signal with '1'. The Register File Unit is inactive, since control signal RegWrite is '0'. Because the control signals ALUsrcA='0' and ALUsrcB ='1' and ALUop='add', the PC is incremented with 4 in the Arithmetic and Logic Unit (ALU). Operation "PC ← PC + 4" means that after the current instruction is finished to be executed, the next instruction will begin execution.

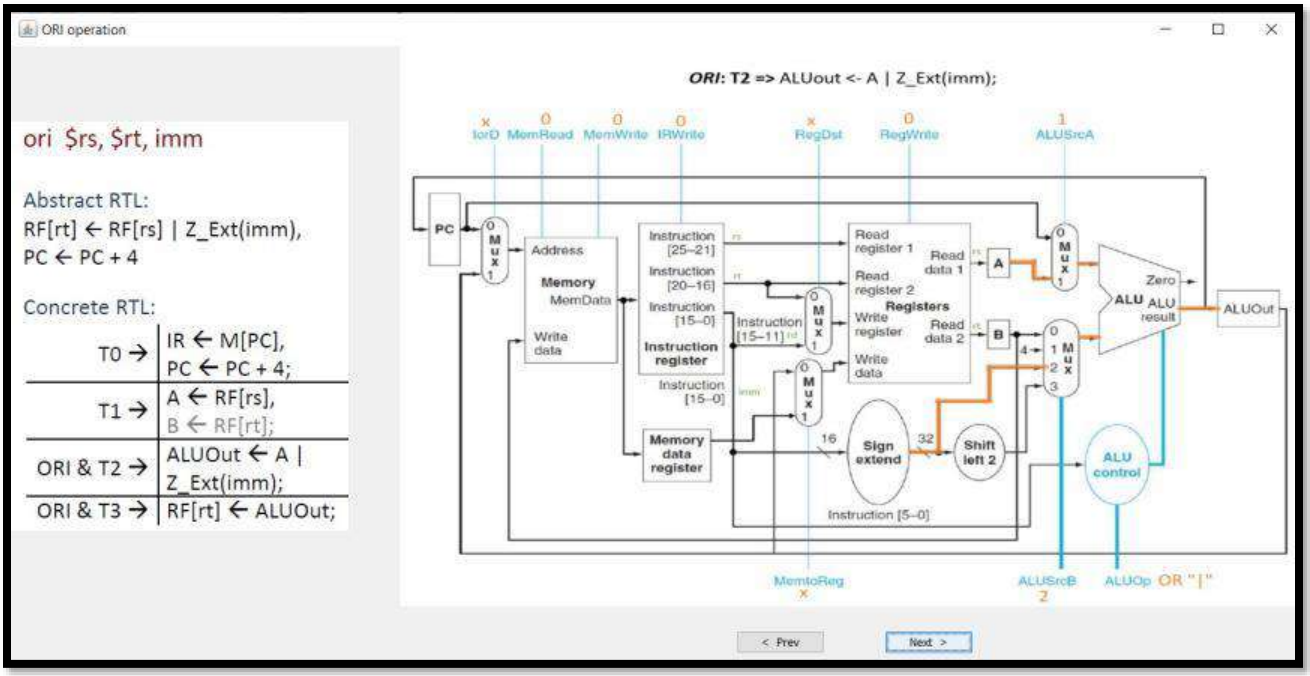
- **Second image**



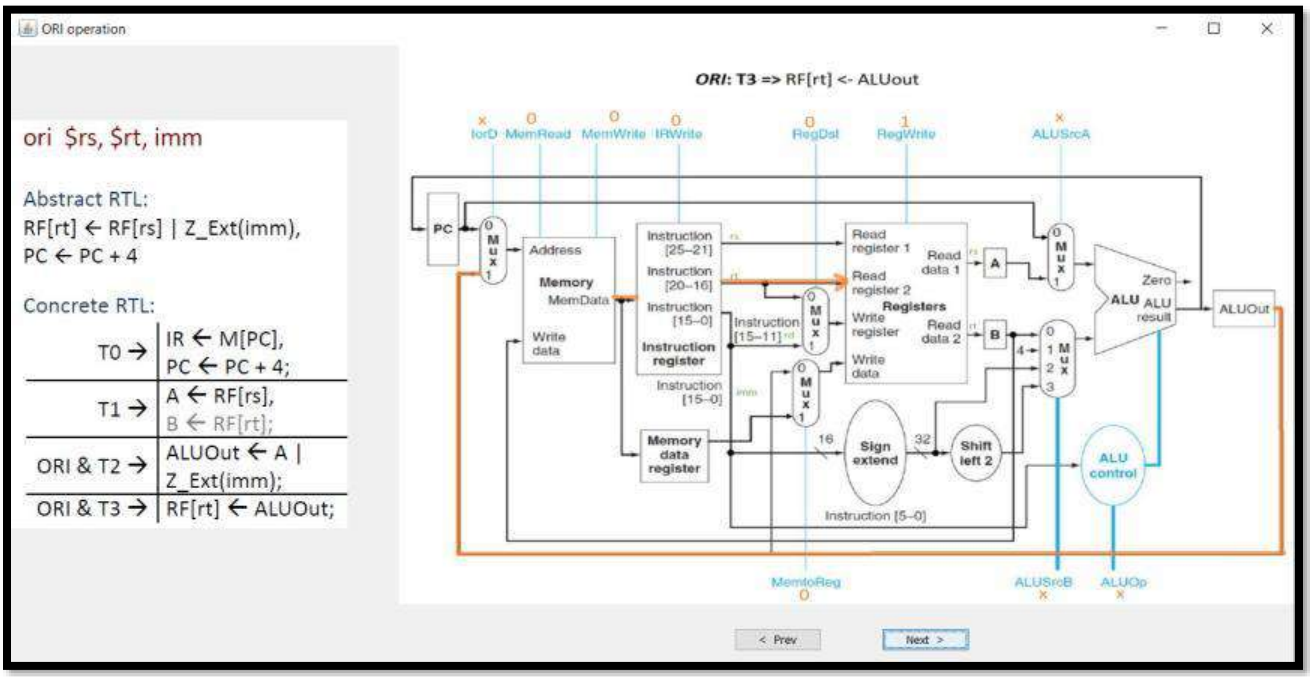
➔ At time T1, in the next Clock cycle, the RS (source register = Instruction[25..21]) and RT (target register = Instruction[20..16]) from Instruction Register Unit are read into the Register File Unit and are passed to the registers A and B, respectively.

- **Third image**

➔ At time T2, the ALUout register receives the result of the OR operation between A register and the Signed extended Version of the Instruction from the IR. The OR operation is performed by the ALU. In order to execute this CLK cycle, the control signals must be set: ALUsrcA='1' and ALUsrcB='2' and ALUop='or'.



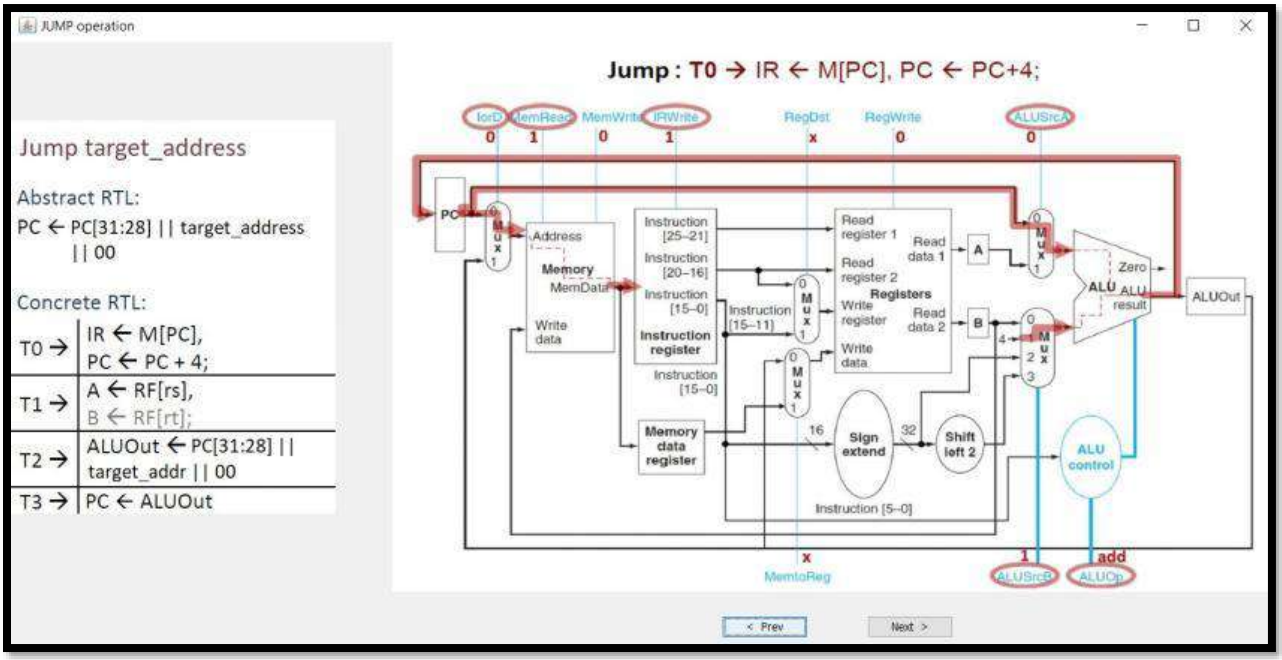
- **Forth image**



➔ In the final CLK cycle, at time T3, the result of the OR operation computed by ALU, stored in ALUout register, is assigned to the Target Register of the Register File Unit. This is done by activating the RegWrite signal, making it '1'. Therefore, the instruction was executed, as described by the Abstract RTL.

***J-type instruction*: JUMP operation Window**

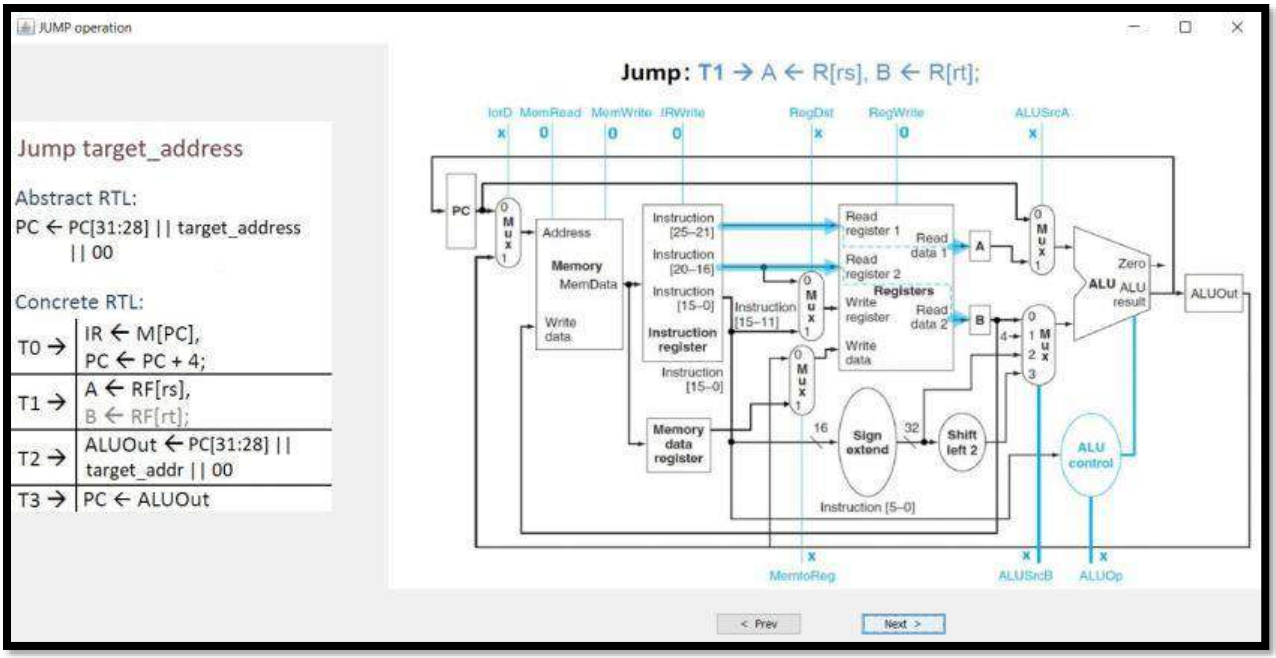
- **First image**



➔ At time T0, the Program Counter (PC) is passed to the Memory Unit as an address, by making the select signal IorD for the first Multiplexer to be '0' in order to allow PC to pass and the control signal MemRead is activated with value '1' in order to allow storage at address PC. The content of the Memory Unit at address pointed by the value of PC, will be assigned to the Instruction Register (IR) by activating the IRWrite control signal with '1'. The Register File Unit is inactive, since control signal RegWrite is '0'. Because the control signals ALUsrcA='0' and ALUsrcB ='1' and ALUop='add', the PC is incremented with 4 in the Arithmetic and Logic Unit (ALU). Operation "PC ← PC + 4" means that after the current instruction is finished to be executed, the next instruction will begin execution.

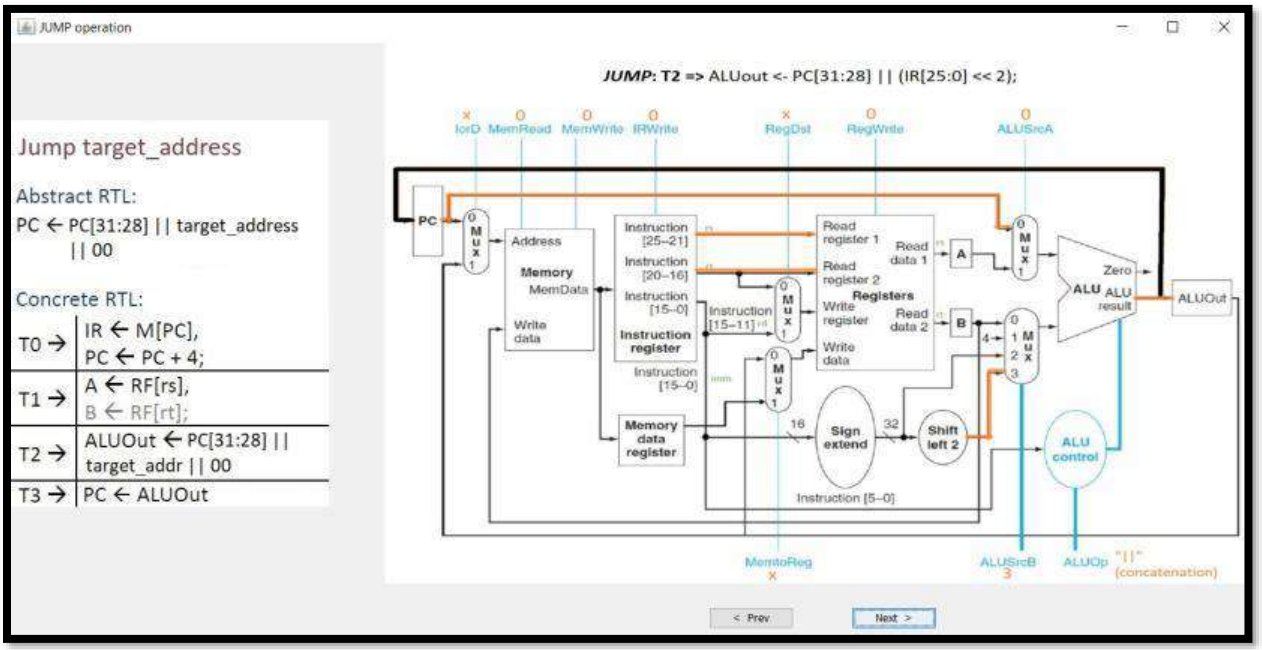
- **Second image**

➔ At time T1, in the next Clock cycle, the RS (source register = Instruction[25..21]) and RT (target register = Instruction[20..16]) from Instruction Register Unit are read into the Register File Unit and are passed to the registers A and B, respectively.

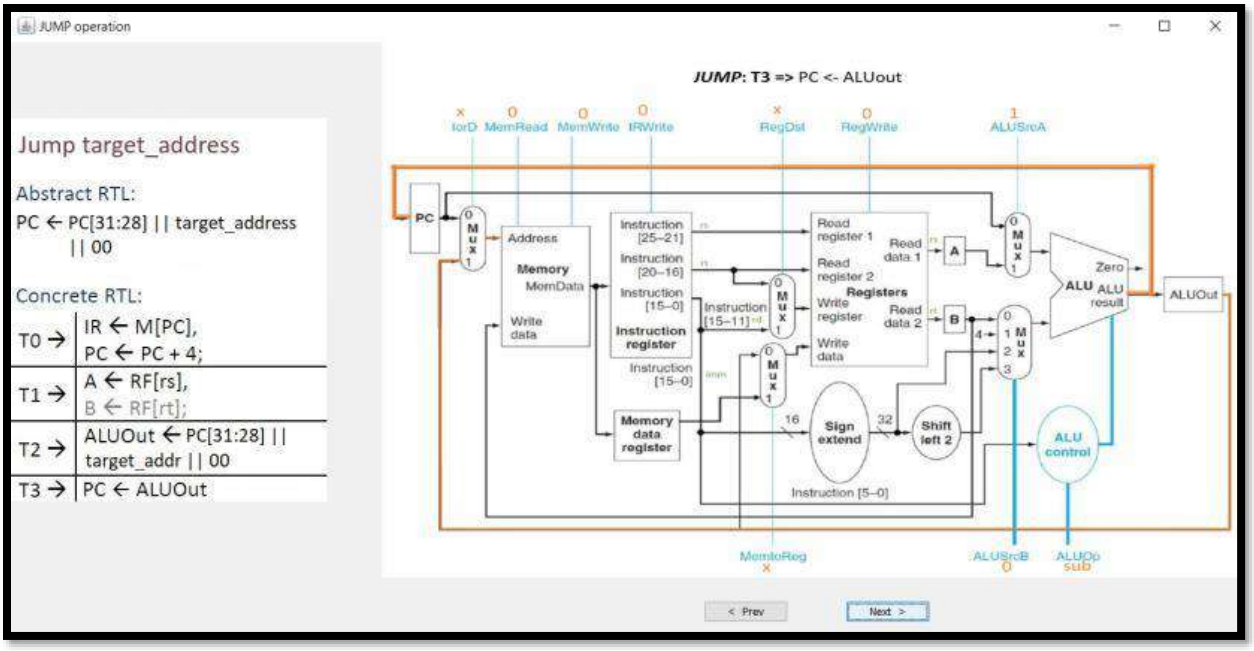


- **Third image**

➔ At time T2, the next CLK cycle, the first 4 Most Significant Bits of the PC are concatenated with the Target Address and with “00”. The Target Address represents the address where the jump must be performed. The control signals must have values: ALUsrcA=’0’ and ALUsrcB=’3’ and ALUop=’concatenation’.



- **Forth image**



➔ In the last CLK cycle, the content of the ALUout register is passed to the PC, in order to complete the execution of the JUMP instruction. To perform this assignment, the control signals must be set as follows: ALUsrcA=’1’, ALUsrcB=’0’, ALUop=’sub’.

5.2. Conclusion

In this documentation was presented the general information about the Instruction Set Architecture, its history and its types. The functioning mode of an ISA was illustrated based on the Multi-cycle MIPS on 32 bits, by using a Java desktop application. The application presents the execution and the highlighted data path for each executed step, for six instructions: ADD, BEQ, LW, SW, ORI, JUMP.

6. References and user manual

6.1. References

1. https://en.wikipedia.org/wiki/IA-64
2. https://en.wikipedia.org/wiki/64-bit_computing
3. https://en.wikipedia.org/wiki/Instruction_set_architecture
4. https://www.geeksforgeeks.org/microarchitecture-and-instruction-set-architecture/
5. https://en.wikipedia.org/wiki/MIPS_architecture
6. https://www.quora.com/What-kind-of-instruction-set-architecture-do-modern-processors-use
7. http://www.cs.kent.edu/~durand/CS0/Notes/Chapter05/isa.html
8. https://en.wikibooks.org/wiki/Microprocessor_Design/Instruction_Set_Architectures
9. https://www.coursera.org/lecture/comparch/isa-characteristics-WXq25
10. https://en.wikipedia.org/wiki/ARM_architecture
11. Computer Architecture Lectures and Laboratories, teacher Negru Mihai (UTCN, second semester from the second year of Computer Science study line)

6.2. User manual

After the application starts, the Dashboard will be displayed. The user can choose one of the available operations by pressing the corresponding button.

A new window will be displayed, containing the RTL Concrete of the operation into a fixed image to the left of the window (as a reference), and another image to the right of the window, representing the highlighted data path from the current step. The user can navigate through the images with the data path by pressing the NEXT and PREV buttons.