FEBRUARY

# 2024

JSDC competition

# PIPELINE CPU


Prepared By:

**Omar AL-khasawneh**

**Omar AL-Salah**

**Moayyad abu mallouh**



0791617606

https://github.com/OmarAl-Saleh/MIPS

# GITHUB REPOSITORY TABLE OF CONTENT

## 1 MODULES

CONTAIN VERILOG FILES THAT ARE USED FOR SIMULATION

## 2 FBGA MODULE

CONTAIN ALL FILES RELATED TO FBGA SETUP

## 3 TEST BENCHES

CONTAIN EVERY VERILOG FILE THAT IS USED AS TEST BENCH

## 4 SIMULATION & REPORTS

CONTAIN SIMULATION RESULTS AND EVERY PHASE REPORT

## 5 PROJECTS

CONTAIN EVERY QUARTUS PROJECT THAT WE BUILD-IN DESIGN

## 6 REFRENCES

CONTAIN ALL REFERENCES THAT WE DEPEND ON

# Acknowledgments

It"It is with immense gratitude that I acknowledge the support and guidance of Professor Mohammad A. Alshboul during the intricate process of designing the pipeline CPU. His profound expertise and deep understanding of the subject were pivotal in navigating the complexities inherent in such an endeavor.

Professor Alshboul's meticulous attention to detail, patience, and hands-on approach were instrumental in every phase of the design. He consistently provided insightful feedback, ensuring that our design met the highest standards of precision and efficiency. His unwavering belief in the project's potential and my capabilities were a constant source of motivation, pushing me to delve deeper and refine our work further.

In parallel, the assistance and insights from Professor Osama have been equally vital to the success of this project. His expertise in complementary areas provided a broader perspective, enhancing our design's robustness and applicability. Professor Osama's enthusiasm for innovation and his strategic advice helped overcome several challenges, making our journey not just educational but also exceptionally rewarding.

Beyond the technical aspects, Professor Alshboul's dedication to nurturing creativity and fostering a spirit of inquiry has been truly inspirational. Similarly, Professor Osama's commitment to academic excellence and his encouragement to pursue novel solutions have significantly enriched my learning experience. Both have not only been

mentors but also beacons of encouragement, instilling in me a passion for design and a commitment to excellence.

To Professor Alshboul and Professor Osama, I extend my heartfelt appreciation for your invaluable support, guidance, and mentorship. Your influence on this project, and on my academic journey as a whole, cannot be overstated. Thank you both for being the cornerstones of this achievement."

I am immensely grateful for the guidance, expertise, and unwavering support provided by Husam A. Suleiman. His contribution has been a cornerstone of the project's success.

For his invaluable contributions, steadfast support, and inspiring mentorship, I extend my deepest thanks to Husam A. Suleiman.

GitHub link → https://github.com/OmarAl-Saleh/MIPS

# Abstract

This project explores the design, implementation, and evaluation of a pipelined MIPS CPU, utilizing Verilog for the hardware description language. The objective was to create a processor architecture that enhances computational throughput and efficiency by introducing a pipelined approach to instruction execution. Unlike traditional single-cycle CPUs where each instruction is executed in one complete cycle, our pipelined MIPS CPU breaks down instruction execution into distinct stages, allowing multiple instructions to be processed simultaneously at different stages of execution.

The architecture is structured around five key stages: instruction fetch, instruction decode, execution, memory access, and write-back. This division enables the CPU to begin processing a new instruction at the fetch stage while previously fetched instructions progress through the subsequent stages, significantly improving instruction throughput compared to a single-cycle architecture.

A meticulous emphasis was placed on the modular design and scalability of the processor to facilitate future expansions and optimizations. Preliminary testing demonstrates that the pipelined MIPS CPU operates successfully, executing instructions with increased efficiency and speed. This project serves not only as a valuable educational resource for understanding pipelined CPU

GitHub link → https://github.com/OmarAl-Saleh/MIPS

architectures but also lays a solid foundation for further research into optimizing pipeline performance and implementing advanced features such as branch prediction and hazard detection.

Future work will focus on enhancing the pipeline's efficiency through the integration of these advanced mechanisms, aiming to minimize stalls and improve the overall performance of the MIPS processor. Through this project, we contribute to the ongoing evolution of CPU design, highlighting the potential for significant advancements in processing speed and efficiency offered by the pipelined approach.

GitHub link → https://github.com/OmarAl-Saleh/MIPS

# Table of Contents



GitHub link → https://github.com/OmarAl-Saleh/MIPS

GitHub link → https://github.com/OmarAl-Saleh/MIPS

GitHub link → https://github.com/OmarAl-Saleh/MIPS

# Introduction

## 1.1 Introduction

In the ever-evolving landscape of computer science and engineering, the design and implementation of central processing units (CPUs) continue to be a focal point of innovation and exploration. Among the various CPU architectures, the development of a pipeline CPU, akin to the revered MIPS, stands as a compelling and vital undertaking. This project represents our endeavor to grasp and engineer a fundamental yet pivotal component of modern computing systems.

## 1.2 The Importance of the Design

The importance of designing and understanding single-cycle CPUs is multifaceted and integral to the field of computer science and engineering. In a world where processing power and efficiency are at the forefront of technological advancement, the design of CPUs becomes an arena where every microarchitectural decision counts. The single-cycle CPU, characterized by its simplicity and transparency, holds particular significance. It offers a clear and unambiguous model for comprehending the inner workings of a CPU. By its nature, it encapsulates the essence of instruction execution in a single clock cycle, thereby facilitating a straightforward understanding of processor operations. In this project, we dive into this simplicity to harness its educational and practical merits.

## 1.3 Motivation

The motivation behind this project stems from a shared passion for learning and a profound curiosity about the architecture of modern computers. Understanding the CPU, often referred to as the "brain" of a computer, is a fundamental step in comprehending how computers execute instructions, process data, and perform complex operations. As students in the field of computer science and engineering, our motivation is twofold. Firstly, we aim to deepen our knowledge of CPU design, enabling us to demystify the underlying mechanisms of computing systems. Secondly, we aspire to create a resource for students and enthusiasts alike to embark on a journey of discovery, employing our project as an educational tool.

## 1.4 Why This Topic is Important for Students

For students, the significance of exploring the design and implementation of a single-cycle CPU lies in the educational value it holds. It serves as a comprehensive learning experience, bridging theory and

practice in the field of computer architecture. As an educational endeavor, our project offers students the opportunity to delve into the intricacies of CPU design, to comprehend the essence of instruction execution, and to witness the tangible results of their efforts. This not only enhances their academic knowledge but also fosters a deeper appreciation for the technologies that underpin our modern world. By providing a practical and accessible entry point into CPU architecture, our project empowers students to become proficient problem solvers, critical thinkers, and innovative engineers.

## 1.5 Objectives of the Project:

1. **Pipelined CPU Architecture:** The primary goal of this initiative is to develop and deploy a pipelined CPU, deeply rooted in MIPS architectural concepts. This CPU is designed to efficiently process a series of essential instructions across multiple stages within each clock cycle.

2. **Educational Resource:** We aim to create an educational resource that not only serves our learning goals but also benefits other students and enthusiasts in the field of computer science and engineering. The project aims to offer a clear, step-by-step insight into CPU design and facilitate a deeper understanding of the architectural choices involved.

3. **Comprehensive Understanding:** To achieve a comprehensive understanding of computer architecture, we will delve into the intricacies of various CPU components, including the control unit, ALU, registers, and memory hierarchy. This understanding will enable us to articulate the rationale behind design decisions.

4. **Performance Analysis:** The project seeks to analyze the performance of the single-cycle CPU in terms of execution speed, resource utilization, and comparison with other CPU architectures. This analysis will help in assessing the practical implications and limitations of the design.

## 1.6 Description of Design Achieved:

In pursuit of these objectives, we have successfully designed a single-cycle CPU model with the following characteristics:

- **MIPS-Inspired Architecture:** Our CPU design is heavily influenced by the MIPS architecture, renowned for its simplicity and elegance. This architecture served as a valuable reference point in shaping our CPU's instruction set, control unit, and data path.

- **RISC (Reduced Instruction Set Computer) Principles:** Our CPU follows the RISC principles by focusing on a limited set of simple and frequently used instructions. This design choice enhances the CPU's efficiency and ease of understanding.

- **Pipelined Processing:** Our CPU architecture is engineered to handle instructions across multiple stages, each occurring in successive clock cycles. This pipelined approach streamlines instruction fetch, decode, execution, and the writing of results to registers, enhancing throughput and efficiency in contrast to single-cycle execution models.

- **Control Unit:** We have implemented a control unit that generates control signals to direct the CPU's operations based on the current instruction. This control unit is responsible for managing the flow of data within the CPU.

- **ALU and Registers:** The Arithmetic Logic Unit (ALU) performs arithmetic and logical operations, while the registers store data. Our CPU features a specific number of registers, and the ALU is capable of executing a variety of operations.

- **Memory Hierarchy:** We have incorporated a memory hierarchy with separate instruction and data memory. This design choice aligns with modern CPU architectures and offers an accurate representation of how instructions and data are stored and accessed.

## 1.7 Design Requirements:

Our CPU design adheres to the following key requirements:

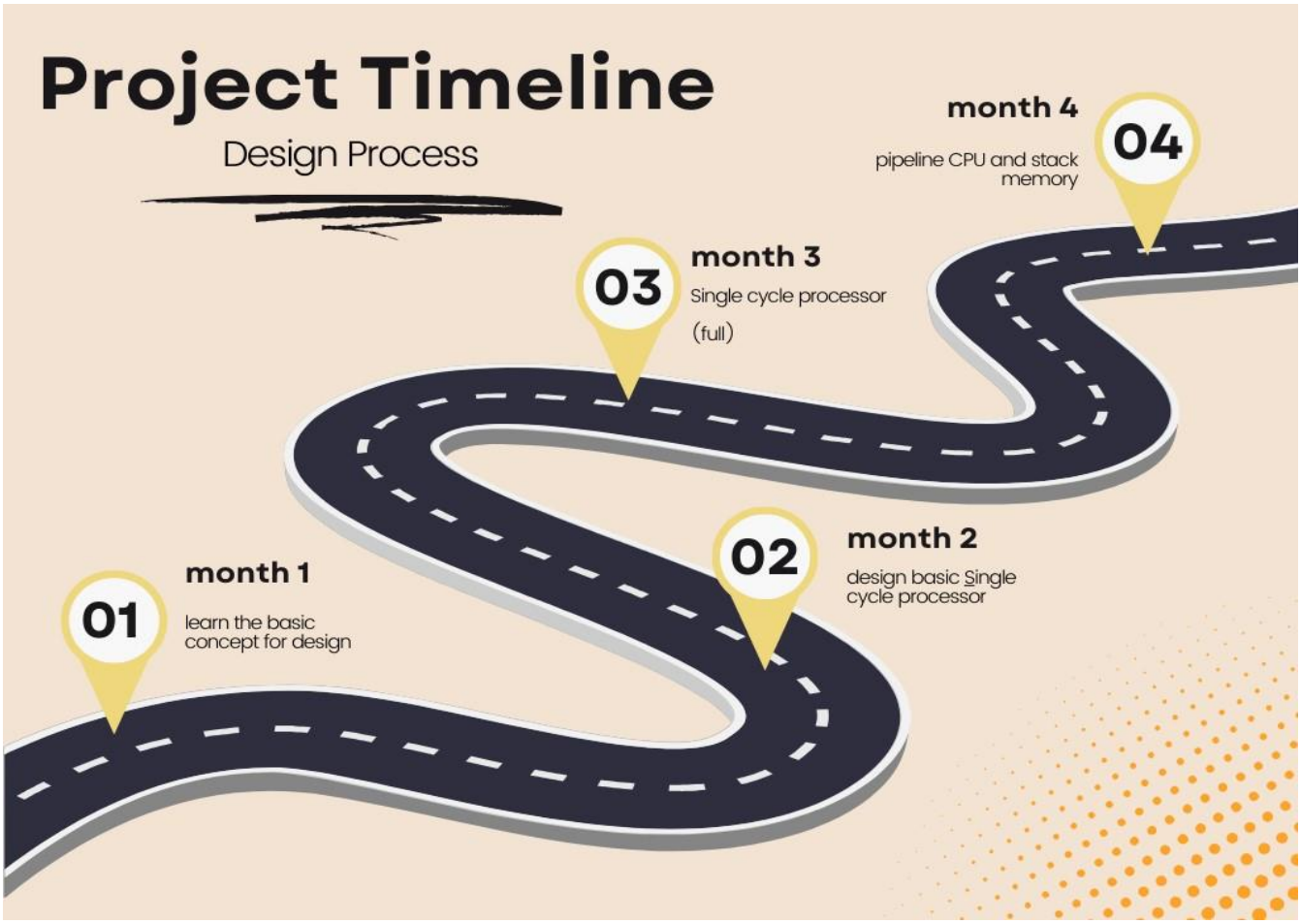
- **Simplicity:** The design should prioritize simplicity and clarity to serve as an educational tool. It should be comprehensible to students and enthusiasts with a basic understanding of digital logic and computer architecture.

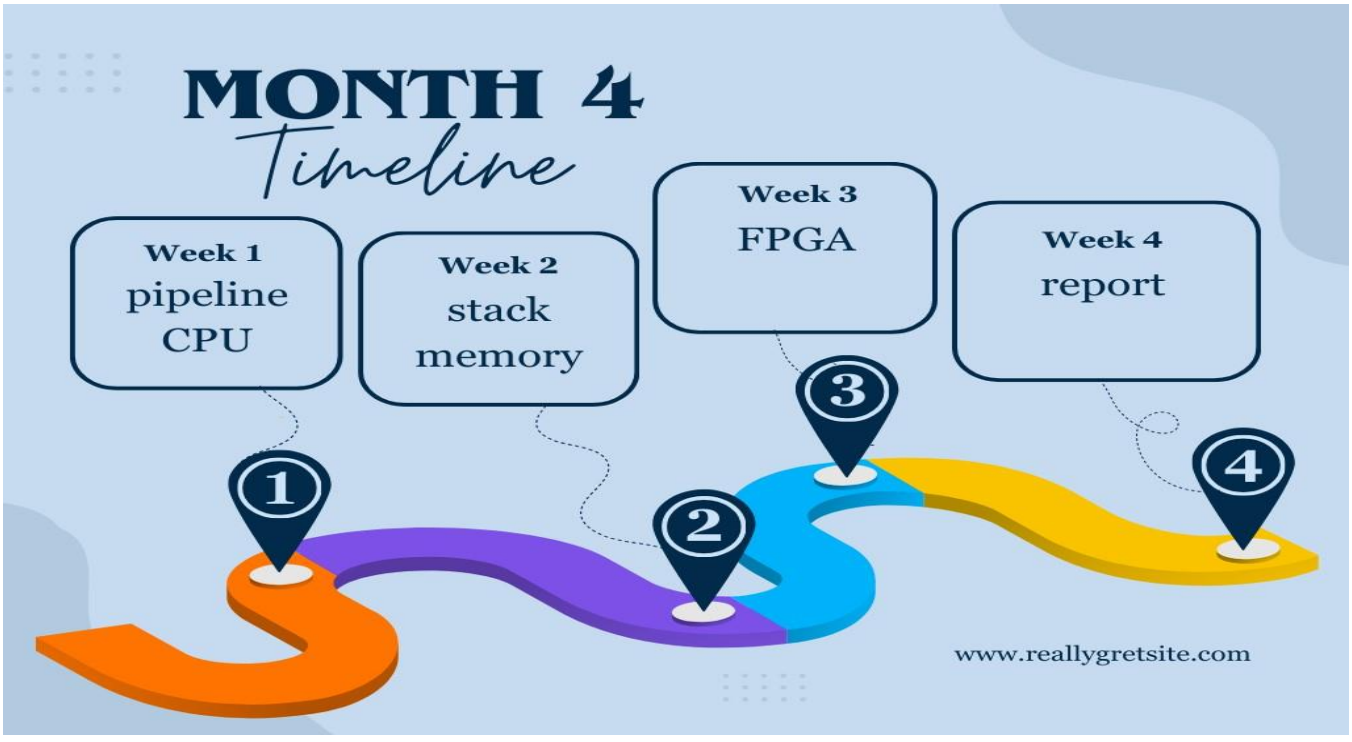
- **Pipelined Execution:** Instructions progress through multiple stages within the CPU, with each stage completing in a single clock cycle, illustrating the essential mechanics of a pipelined CPU architecture.

- **Instruction Set:** The CPU should support a defined instruction set, inspired by the MIPS architecture. This instruction set should include fundamental operations such as arithmetic, logic, and data movement instructions.

- **Efficiency:** The design should strive for efficiency by optimizing the use of resources and minimizing redundancy.

## 1.8 Timeline project




GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.9 The team's member responsibility

### 1.9.1 Omar AL-khasawneh (Leader)

- ALU design
- ALU control design
- PC design
- Forwarding
- Writing report

### 1.9.2 Omar AL-Salah

- Register file
- Control unit
- Hazard unit
- Stack Memory Implementation
- Jump Forwarding Unit
- Debugger

### 1.9.3 Moayyad Abu Mallouh

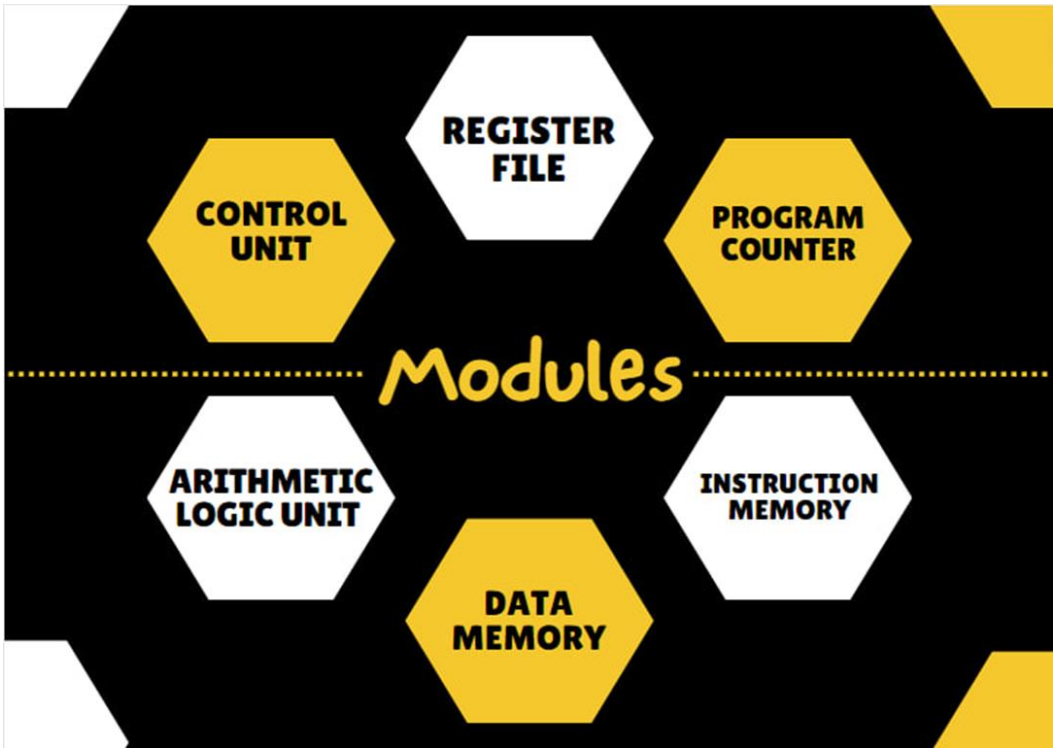
- Instruction memory
- Data memory
- Branch
- FPGA

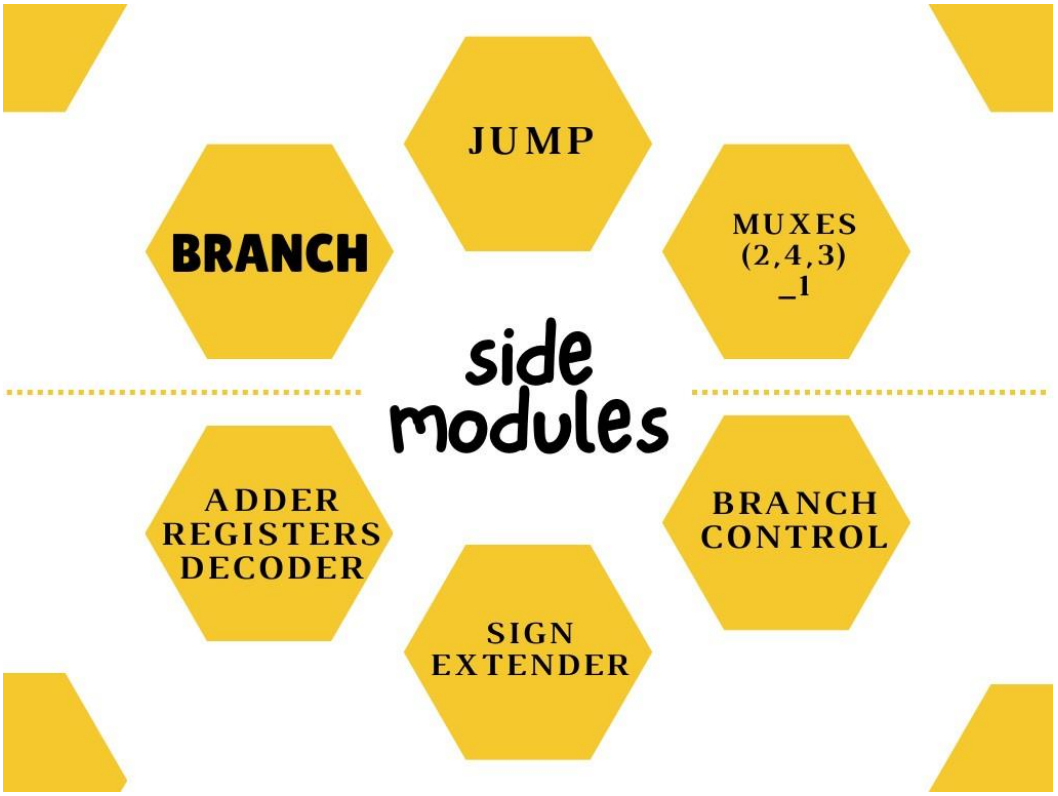
Each member assists others with various tasks

GitHub link → https://github.com/OmarAl-Saleh/MIPS

# Design

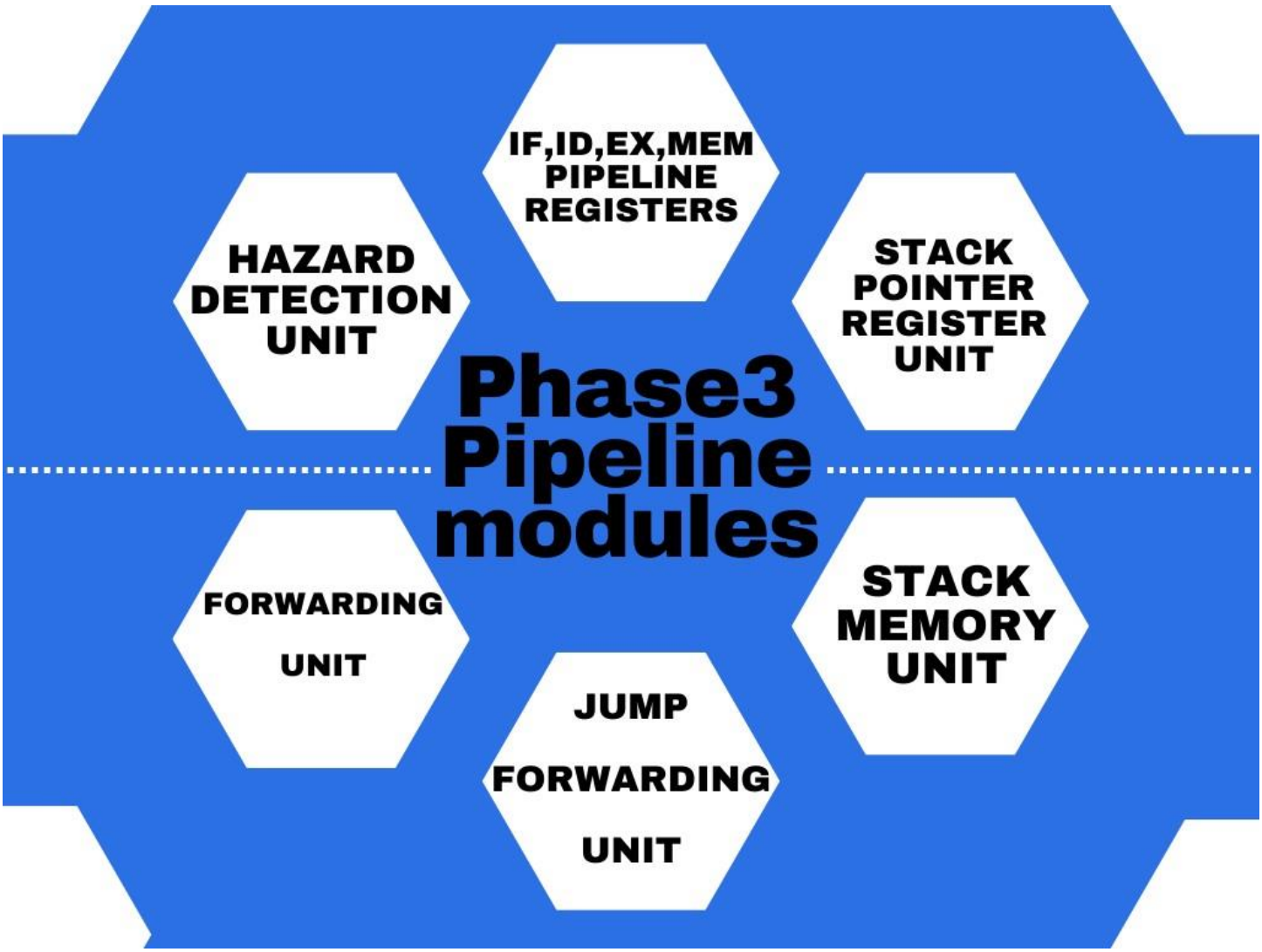
## 1.10 Hardware Design and Implementation




GitHub link → https://github.com/OmarAl-Saleh/MIPS

Extra module in phase 3




GitHub link → https://github.com/OmarAl-Saleh/MIPS

# 1.11 Component

**Data Memory :** in the context of computer architecture and digital systems, is a component that stores and retrieves data during the execution of programs. It serves as a storage medium for variables, arrays, and other data structures used by a computer's central processing unit (CPU) to perform calculations and operations.

**Instruction memory**: also known as the Instruction Cache or Code Memory, is a component in a computer's architecture that stores the machine code instructions that the CPU (Central Processing Unit) fetches, decodes, and executes to perform various tasks and operations. These instructions are part of the computer program or software being run on the CPU. Instruction memory is essential for the operation of a computer because it holds the program's instructions in a format that the CPU can understand and execute sequentially.

**Note**: We Develop both Data Memory and Instruction Memory from scratch without using the built-in Memory, and initialize them with FSM Techniques

Arithmetic logic unit: (ALU)

is a crucial component of a computer's central processing unit (CPU). It performs arithmetic and logic operations on data, such as addition, subtraction, multiplication, division, and comparisons.

The ALU is designed to perform a variety of arithmetic and logical operations, dictated by a 4-bit control signal (ALUControl). It processes 32-bit inputs (A and B), handling basic arithmetic operations like addition, subtraction, multiplication, and division, as well as logical operations including AND, OR, XOR, and NOR. Additionally, it supports shift operations (logical left and right shifts) which are controlled by a 5-bit shift amount.

One of the notable features of this ALU is its ability to respond to different branch types, determined by a 3-bit signal (branch_type). This enables the ALU to participate in decision-making processes, fundamental in computational logic, by setting a Zero flag based on specific conditions (like equal, not equal, greater than, or less than comparisons).

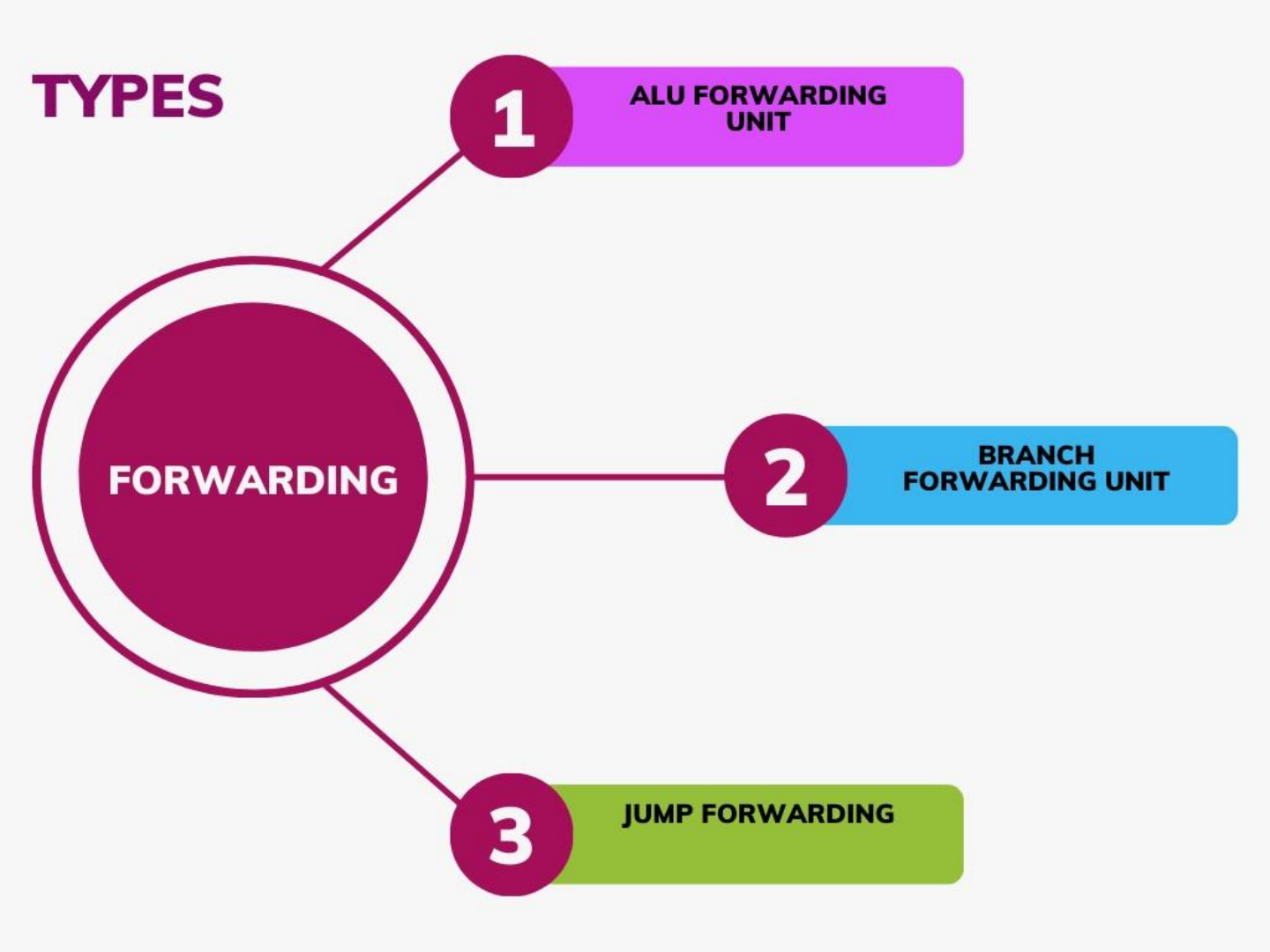
Furthermore, the ALU includes an internal mechanism to handle overflow and carry-out scenarios, which are critical in signed arithmetic operations. These scenarios are managed through dedicated blocks that set the Overflow and CarryOut flags under specific

conditions, like during addition or subtraction when the sign bit (most significant bit) might be incorrectly set or unset due to the arithmetic operation.

Overall, this ALU module illustrates the complexity and versatility of digital logic designs in performing essential computational tasks. It is a quintessential example of how various arithmetic and logical operations are implemented at a hardware level in computing systems.

GitHub link → https://github.com/OmarAl-Saleh/MIPS

1.11.1 Forwarding :




The Forwarding Unit in MIPS (Microprocessor without Interlocked Pipeline Stages) architecture plays a crucial role in optimizing the performance of pipelined processors by addressing data hazards that naturally occur due to the overlap of instruction execution.

Forwarding mechanisms play a crucial role in data dependency detection and instruction handling within the processor. The first module, the ALU Forwarding Module, operates within the EX stage and is primarily utilized for identifying data dependencies in R-type instructions. It produces output signals for ALU MUX A and

MUX B selection lines. These signals indicate various forwarding scenarios: '00' signifies no forwarding, '01' denotes forwarding from the WB stage, and '10' represents forwarding from the MEM stage.

The second component, the Branch Forwarding Unit, is deployed during the ID stage to pinpoint data dependencies in branch instructions. It generates output signals for Branch Unit MUX A and B selection lines, following a similar pattern to the ALU Forwarding Module: '00' for no forwarding, '01' for forwarding from the WB stage, and '10' for forwarding from the MEM stage.

Lastly, in the ID stage, another forwarding mechanism is employed specifically for detecting data dependencies between JAL and JS instructions. This mechanism, labeled as 'JUMP Target Address MUX,' offers output signals indicating forwarding scenarios. A value of '0' signifies no forwarding, while '1' indicates forwarding from the MEM stage. These forwarding strategies collectively enhance the efficiency and performance of the processor's instruction handling processes.

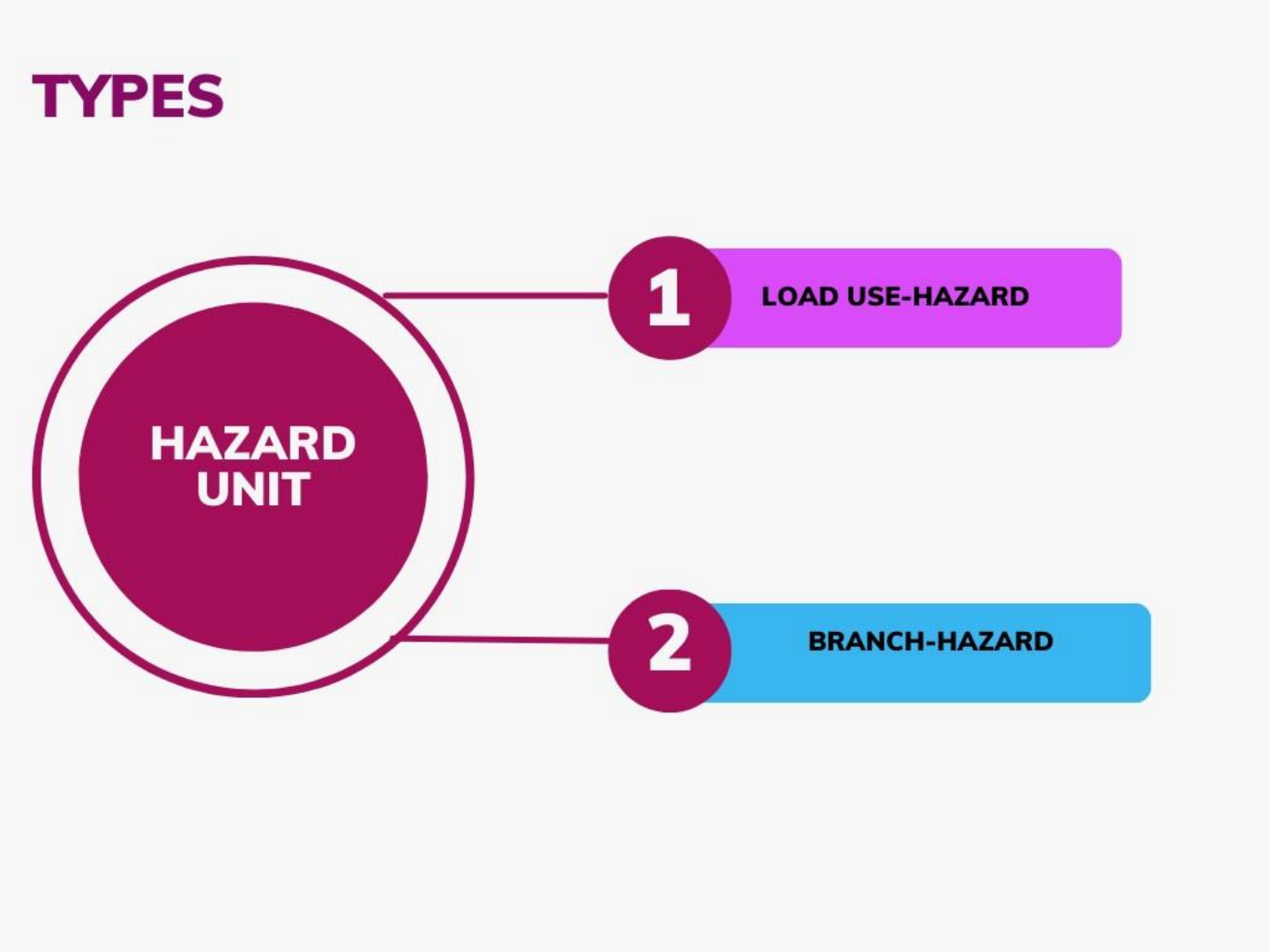
### 1.11.2 Program counter :

also known as the instruction pointer (IP) in some architectures, is a special-purpose register in a computer's central processing unit (CPU). It holds the memory address of the next instruction to be fetched and executed in a program. The PC is automatically incremented after each instruction is fetched, allowing the CPU to sequence through the program's instructions in order.

**we use state machine to make the design better and easy to upgrade .**

GitHub link → https://github.com/OmarAl-Saleh/MIPS

1.11.3 Hazard unit




The Hazard Detection Unit serves the crucial role of identifying instances where forwarding mechanisms fail to operate effectively. Two primary hazard cases are monitored:

1. Load Use-hazard:

This occurs when an instruction in the ID stage exhibits data dependency with a load instruction in the EX stage.

2. Branch Hazard:

This situation arises when a branch instruction in the ID stage demonstrates data dependency either with an instruction in the EX stage or with a load instruction in the MEM stage.

Output signals from the Hazard Detection Unit provide crucial insights into the encountered hazards:

ID_EX_FLUSH: This signal transitions from 0 to 1, indicating the need to flush the ID_EX pipeline register by forcing its signals to become NOP signals.

IF_ID_write: It switches from 1 to 0, disabling the IF_ID pipeline register to repeat the same instruction that caused the hazard.

PC_write: This signal changes from 1 to 0, disabling the PC register by repeating the same value from the last cycle.

When a hazard is detected, the signal values are adjusted accordingly:

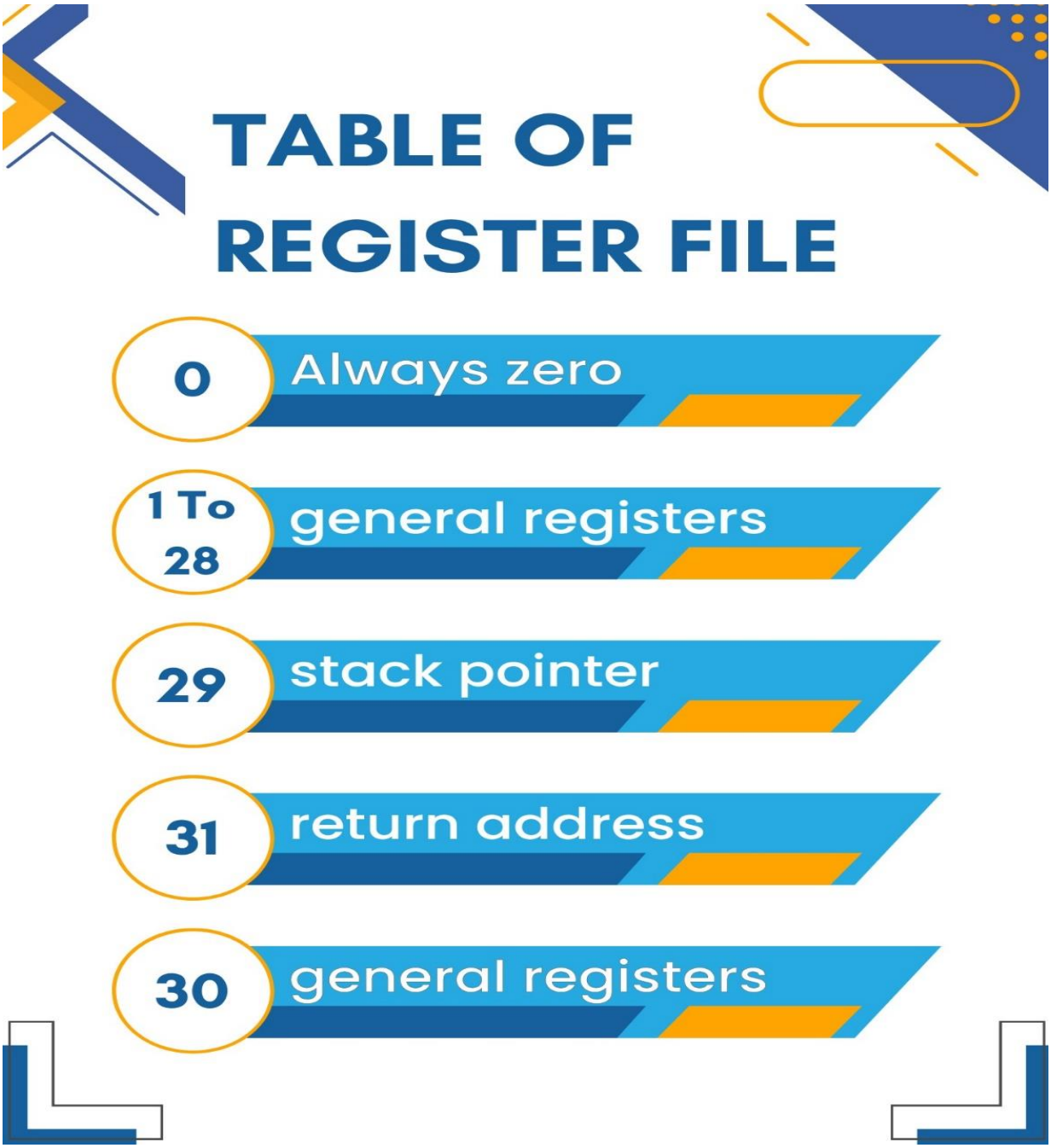
ID_EX_FLUSH is set to 1.

IF_ID_write is set to 0.

PC_write is set to 0.

These adjustments ensure proper handling of hazards and help maintain the integrity and functionality of the processor's pipeline system.

1.11.4 Register file:




is a collection of registers that are used for temporary data storage and manipulation within a central processing unit (CPU). Registers are small, high-speed storage locations

that are an integral part of the CPU. They store operands, intermediate results, and control information needed for executing instructions.

The Register File module stands as a cornerstone in the MIPS architecture, and in our CPU design, we have meticulously crafted a RegisterFile module to fulfill the essential role of managing registers efficiently. The module declaration encompasses standard signals, aligning with the MIPS convention, **including Clock, ReadReg1, ReadReg2, WriteReg, WriteData, Reg_write_Control, ReadData1, and ReadData2.**

- **We employ specific registers for distinct functionalities within our system:**

**Register 0** (Reg 0): This register is dedicated to always maintaining a value of zero.

**Register 29** (Reg 29): It serves as the stack pointer, which is routinely updated with each JAL or JS instruction. Its value is adjusted either positively or negatively, indicating stack growth or shrinkage. Initially, it holds the value 88 Using FSM techniques , representing the first address of the stack memory in RAM.

**Register 31** (Reg 31): This register functions as the return address register. It works in tandem with the stack memory during every JAL or JS instruction. Reg 31 facilitates the collaboration by retrieving the new stack pointer value from RAM, using the current Reg 29 value as an address. This mechanism ensures seamless management of return addresses within the system's execution flow.

**Control unit:** is a crucial component of a computer's central processing unit (CPU) or any computational device. It coordinates the activities of all the other hardware components in the system. Essentially, the control unit fetches instructions from memory and decodes and executes them, sending signals to the other components of the computer to control data

flow and processing operations. It acts as a central manager, directing the operation of the processor and its interaction with other hardware components.

The control unit serves as the central intelligence of our CPU design, acting as the pivotal module that orchestrates the various operations within the processor. At the core of this module lies the ControlUnit entity, taking essential signals such as Clock, Reset, opcode, RegDst, ALUSrc, MemtoReg, MemWrite, MemRead, ALUOp, RegWrite, Branch, Jump, funct, pc_load, and PC_Store.

In aligning with the standard MIPS architecture, we have implemented familiar control signals such as Clock, RegDst, ALUSrc, MemtoReg, MemWrite, MemRead, ALUOp, RegWrite, Branch, Jump, funct , . These signals are crucial for directing the flow of data and control within the CPU.

Additionally, we have introduced new signals to tailor our design for specific functionalities:

## Reset Signal:

The Reset signal serves as a mechanism to reset the control unit, initiating a no-operation (nop) instruction. This feature ensures a controlled start or restart of the CPU, enhancing the robustness and reliability of the system.

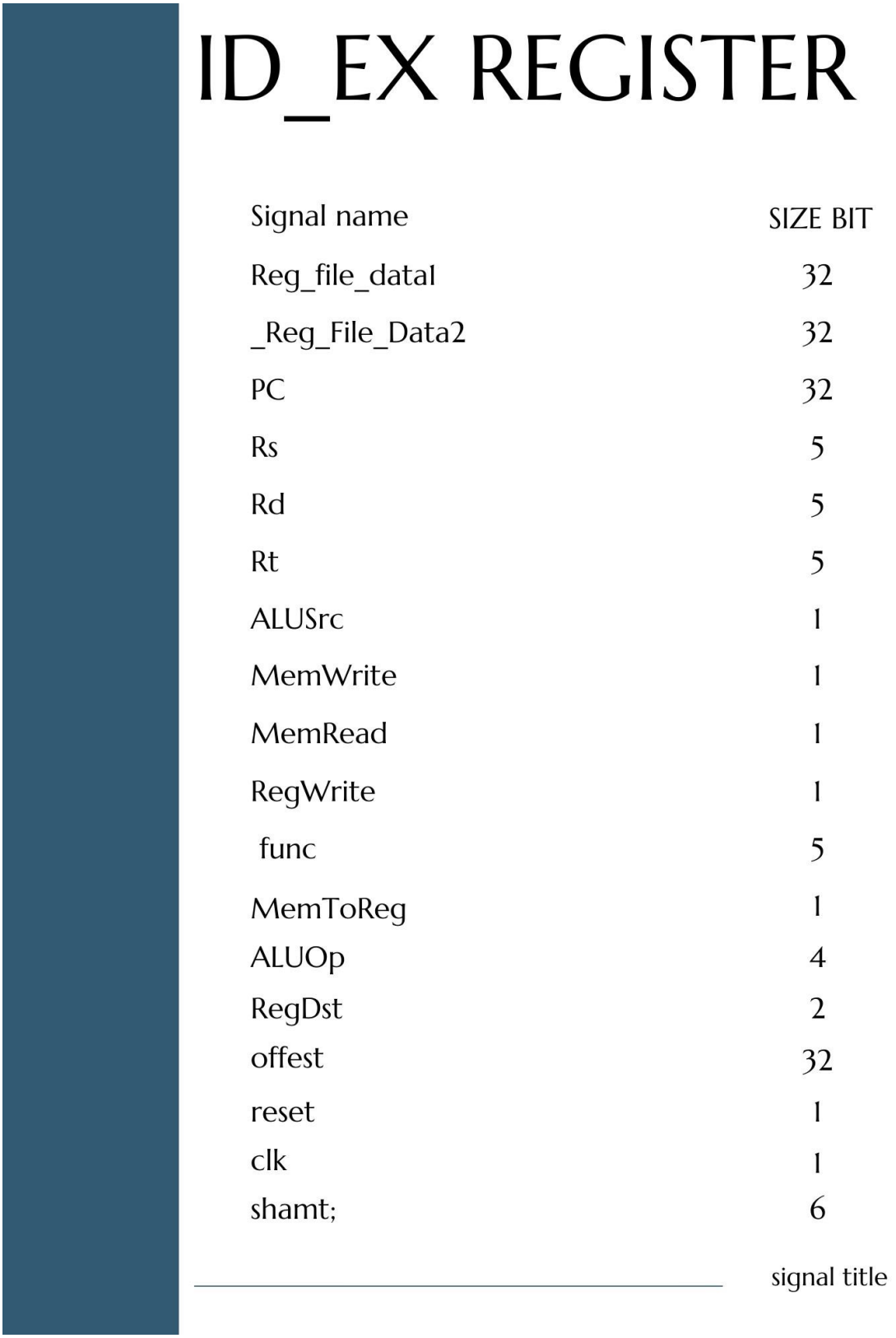
## Halt Signal:

We have developed a new control signal termed "Halt" to address the Halt instruction, which invariably serves as the final instruction in a program, signaling the cessation of execution. Upon the control unit's receipt of the Halt instruction opcode "101101," the Halt control signal is raised to a high state, denoted as 1. This action triggers the cessation of the program counter (PC) and initiates the generation of an infinite loop of Halt instructions within the IF_ID pipeline register. Consequently, this halts further cycles of execution, effectively terminating program execution.

### 1.11.5 IF_ID register

The IF/ID pipeline register is responsible for storing the output from the Instruction Memory.

1.11.6ID_EX register




GitHub link → https://github.com/OmarAl-Saleh/MIPS

1.11.7 EX_MEM register

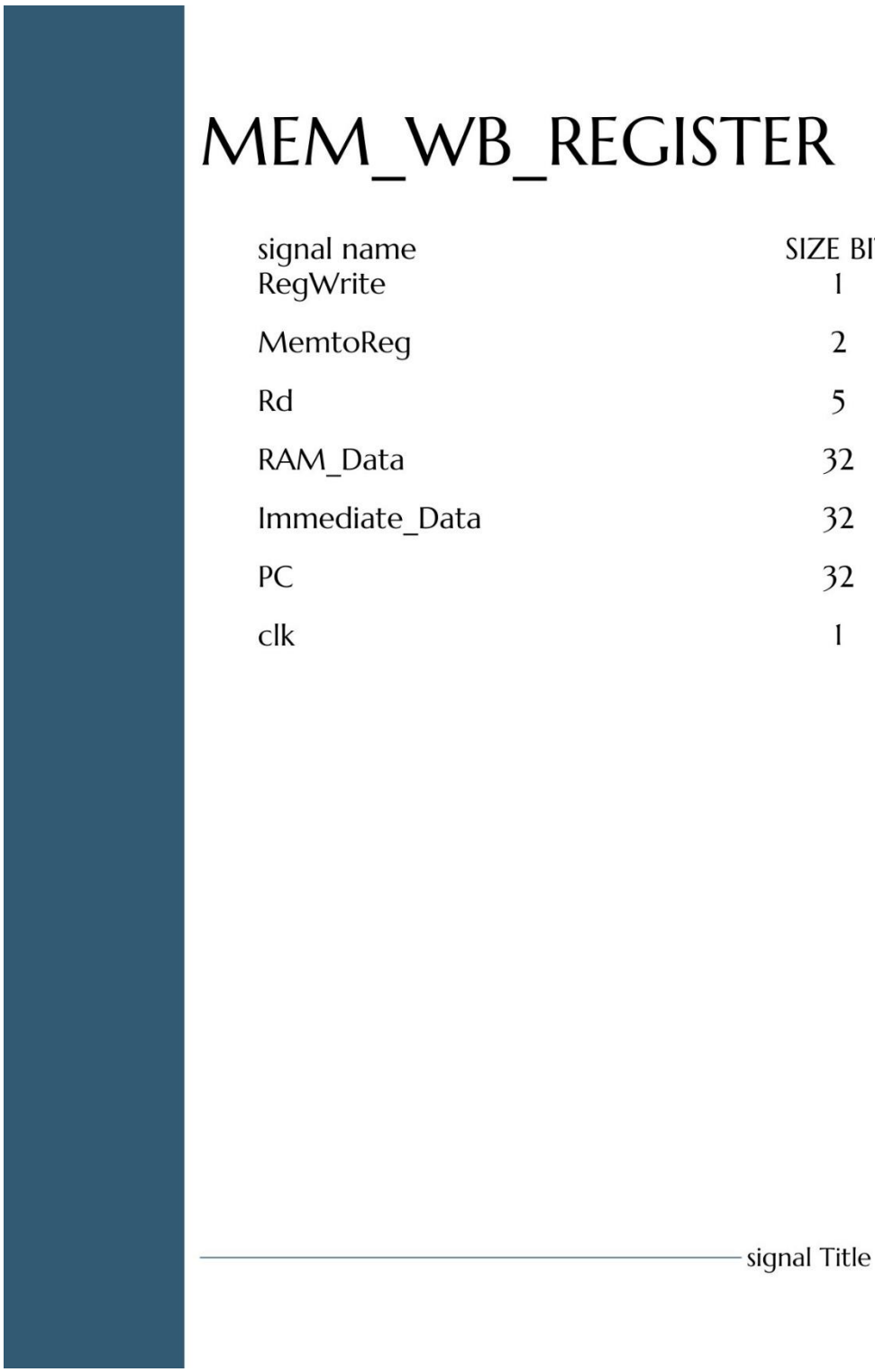
# EX_MEM_REGISTER

| signal name | SIZE BIT |
|---|---|
| Write_Data | 3 |
| PC | 8 |
| Address | 32 |
| Rd | 5 |
| MemWrite | 1 |
| MemRead | 1 |
| RegWrite | 1 |
| MemtoReg | 2 |
| clk | 1 |

signal Title

GitHub link → https://github.com/OmarAl-Saleh/MIPS

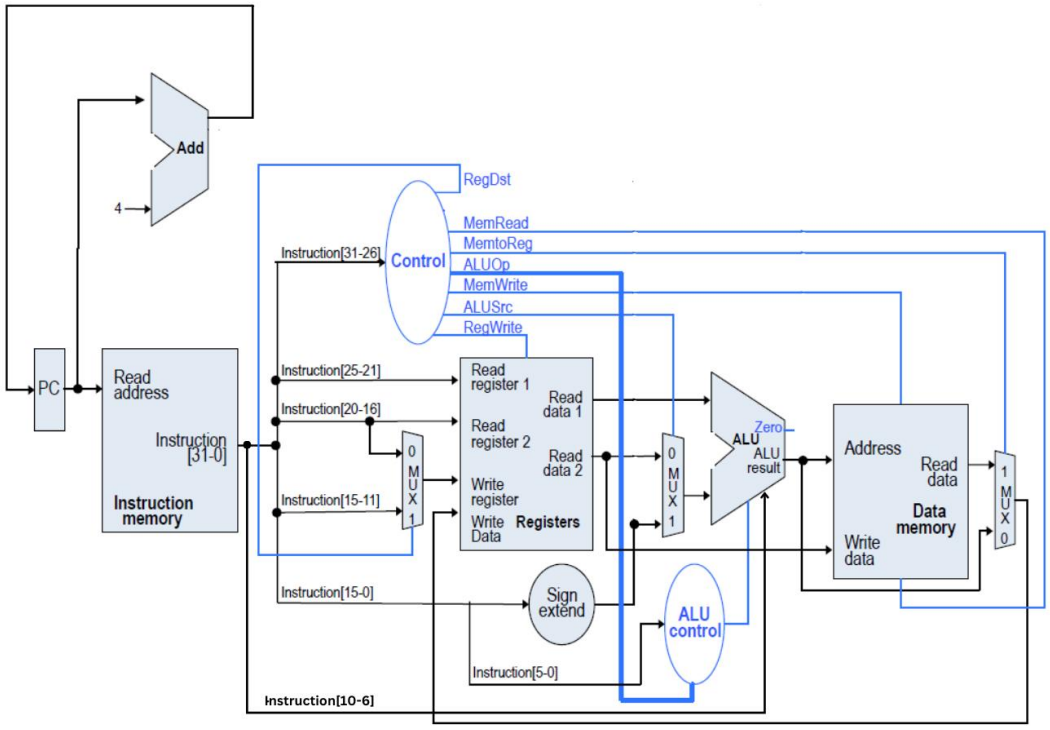
### 1.11.8 MEM_WB register




GitHub link → https://github.com/OmarAl-Saleh/MIPS

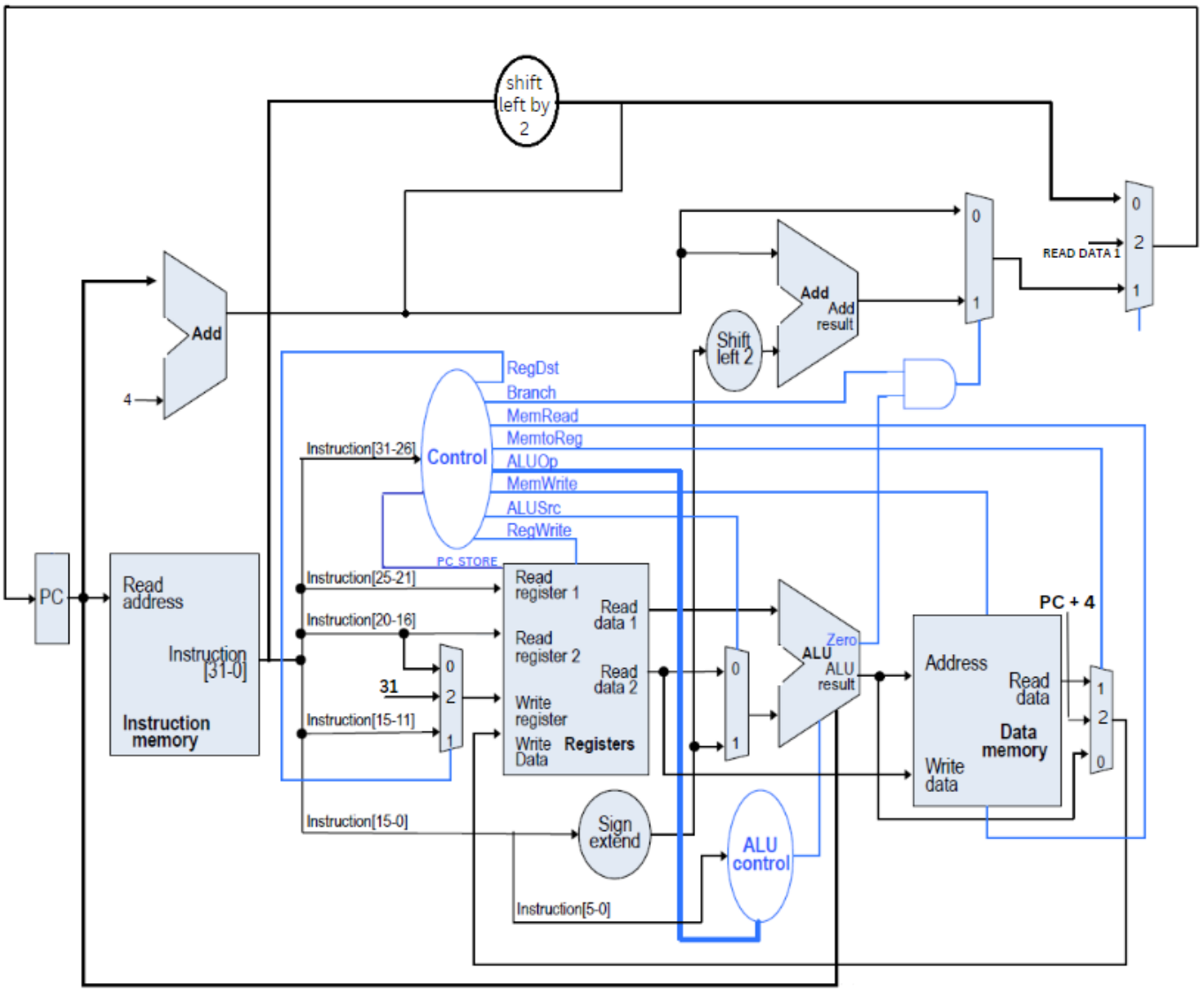
## 1.12 MIPS DATAPATH

### 1.12.1 Phase1



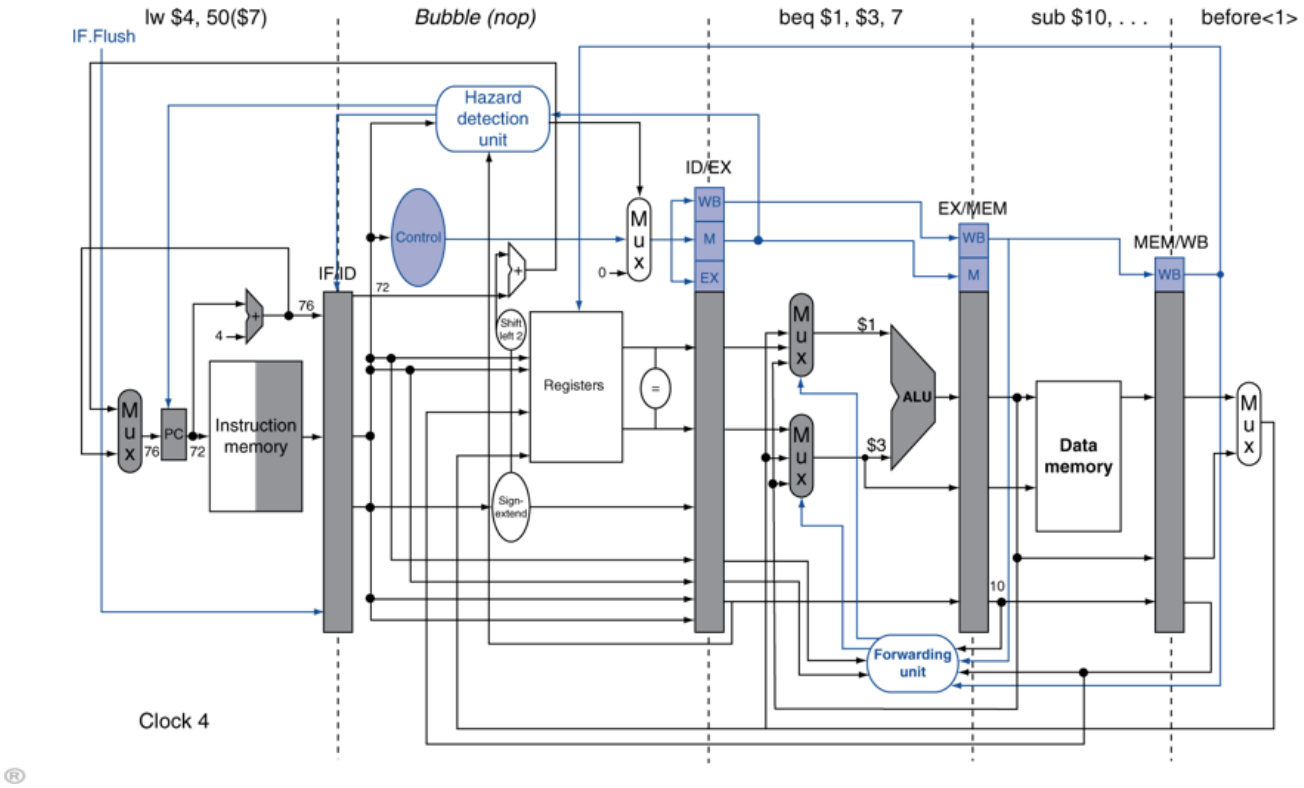

### 1.12.2 Phase 2

GitHub link → https://github.com/OmarAl-Saleh/MIPS




GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.12.3 Phase 3




GitHub link → https://github.com/OmarAl-Saleh/MIPS

# 1.13 MIPS instruction execution phase

## 1.14 Coding and Software Development

### 1.14.1 Tools

In the development and verification of our single cycle CPU, we employed a combination of industry-standard tools to ensure accuracy, efficiency, and compatibility. We used **Verilog** as our hardware description language, a popular choice for its expressive syntax and powerful capabilities in modeling complex digital systems. Verilog facilitated a clear representation of our design, allowing for systematic testing and debugging of individual modules and their integration. Complementing this, **Quartus** served as our primary platform for synthesis, placement, routing, and simulation. Its comprehensive suite of tools enabled us to transform our Verilog code into gate-level representations, ensuring that our design meets the desired performance and resource criteria. Furthermore, Quartus provided a conducive environment for iterative design optimization, ensuring that our CPU design was not only functional but also efficient in terms of resource usage and speed. The synergy between Verilog and Quartus was instrumental in realizing a robust and reliable single cycle CPU.

### 1.14.2 Challenging

1. . We face challenges due to limited online learning resources and the complexities of optimization. To address this, we seek guidance from our university professors.
2. We improved the performance of our design by 1. increasing the clock speed and 2. minimizing the area.
3. Detecting overflow in the ALU and providing the correct overflow output, **Detection of Overflow in the ALU**:
    - **For Addition**:
        - Overflow occurs if:
            - Both operands are positive, but the result is negative.

        - Both operands are negative, but the result is positive.
    - In terms of bit operations, consider the sign bits (most significant bit, MSB) of operands A and B and the result R. Overflow is detected if:
        - A's MSB is 0, B's MSB is 0, but R's MSB is 1.
        - A's MSB is 1, B's MSB is 1, but R's MSB is 0.
- **For Subtraction**: Since subtraction is the same as adding a negative number in two's complement arithmetic, the rules for addition apply here too.

**Handling Overflow in the ALU**:

- **Overflow Flag**: The ALU can set an overflow flag (often denoted as the 'V' flag in many architectures) whenever overflow occurs. This flag can be checked by subsequent instructions, and based on its value, certain actions can be taken, such as branching to error-handling routines.

consumption. There are a number of things that can be done to prevent timing violations, including careful design, the use of buffers, clock gating, and testing.

4. **FPGA**

1. Our university currently lacks access to the latest FPGA units, resulting in extended waiting periods until new FPGA resources are made available through competition allocations.

2. We had no idea how to implement our design on an FPGA, so if consume a lot of time to only know how implement our design in FPGA

GitHub link → https://github.com/OmarAl-Saleh/MIPS

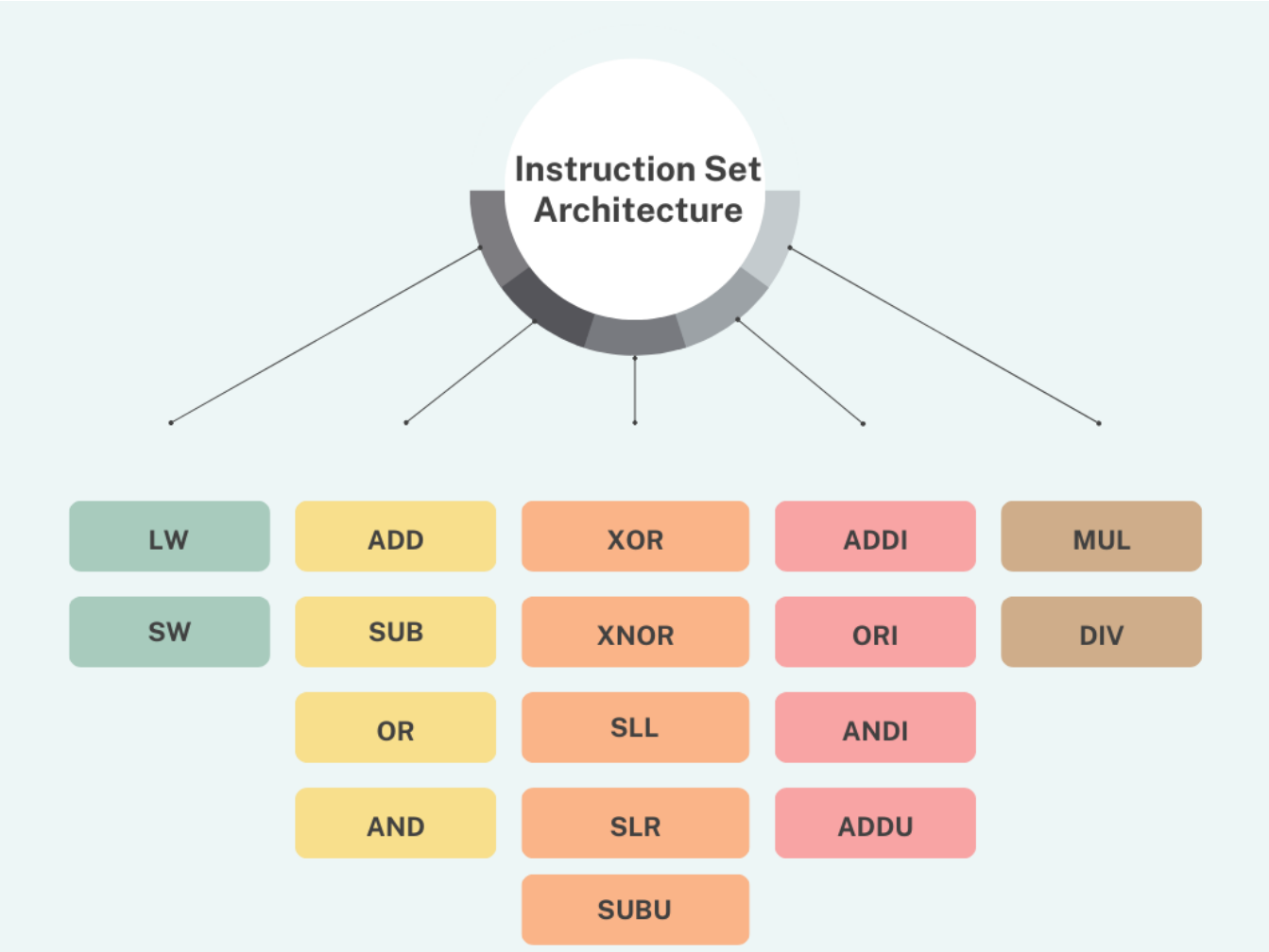
## 5. Stack Memory

One of the primary challenges we confront is the limited availability of resources discussing this topic and providing implementation guidance, primarily due to its close association with the operating system (OS). Additionally, implementing this challenge requires linking it to work seamlessly with the Register File and RAM without introducing any additional Control Unit Signals or Data bus modifications.

GitHub link → https://github.com/OmarAl-Saleh/MIPS

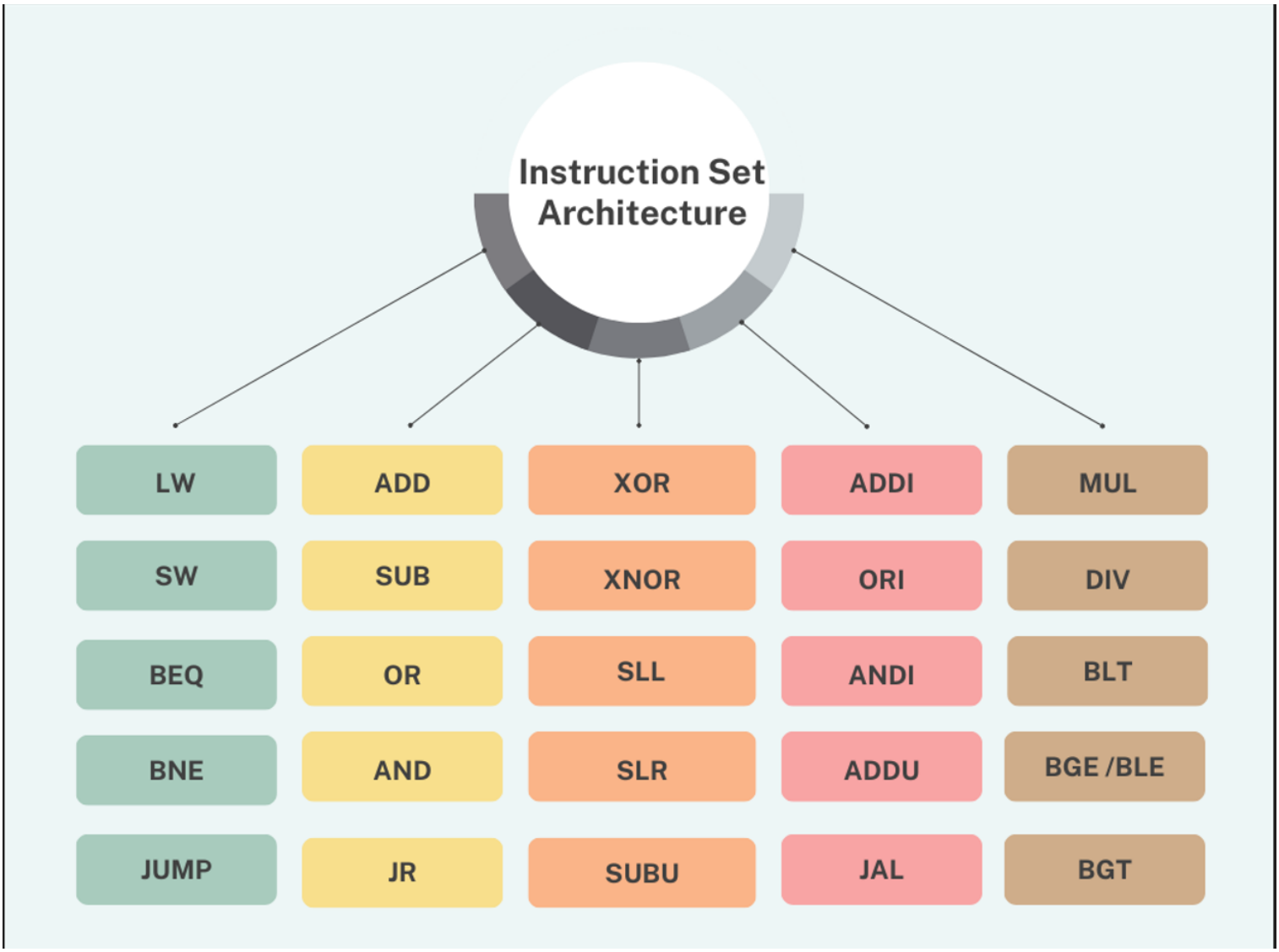
## 1.14.3 Instruction set architecture

### *1.14.3.1 Before (phase one)*



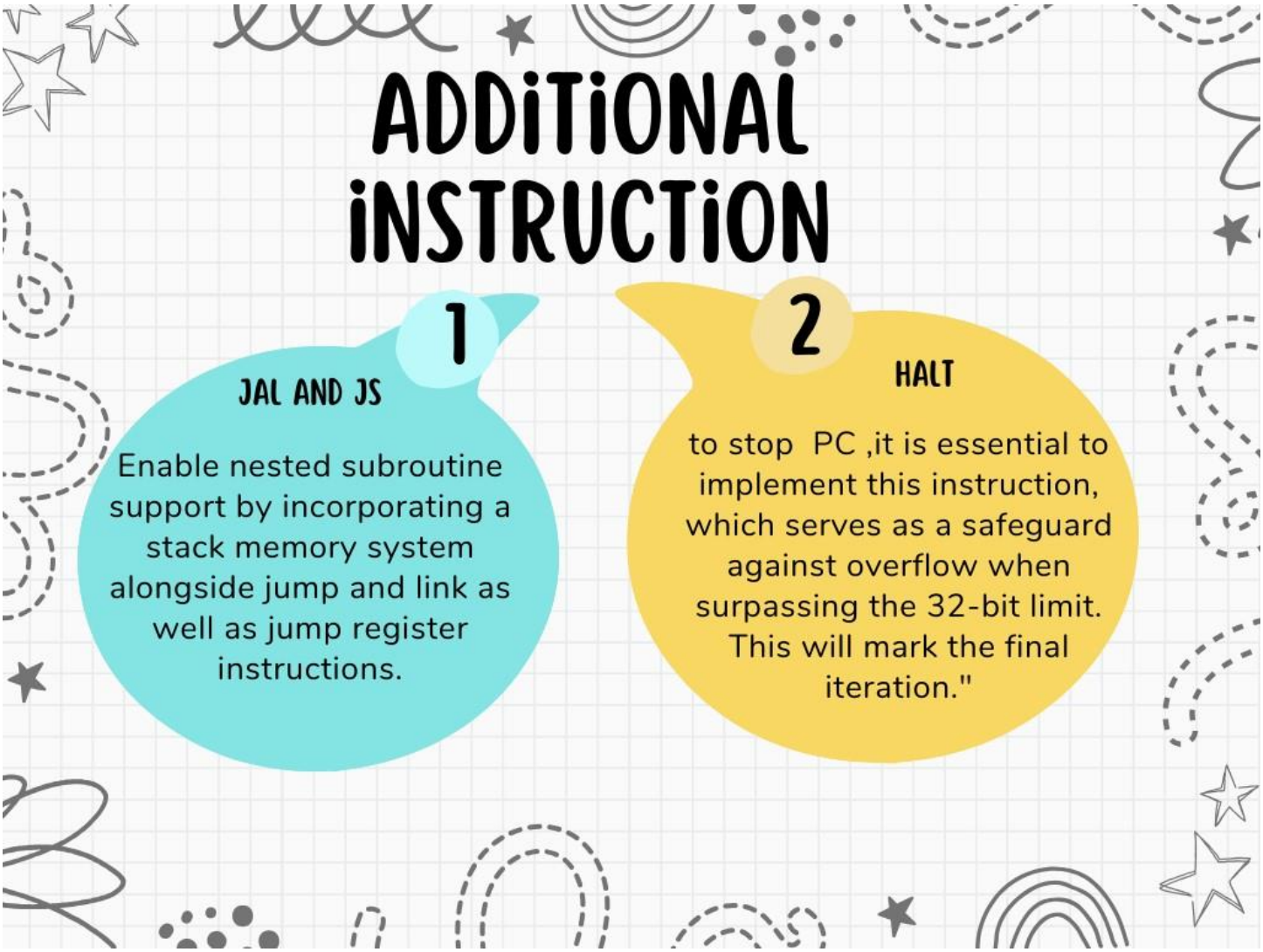

GitHub link → https://github.com/OmarAl-Saleh/MIPS

### *1.14.3.2 phase two*




GitHub link → https://github.com/OmarAl-Saleh/MIPS

*1.14.3.3 phase 3*




GitHub link → https://github.com/OmarAl-Saleh/MIPS

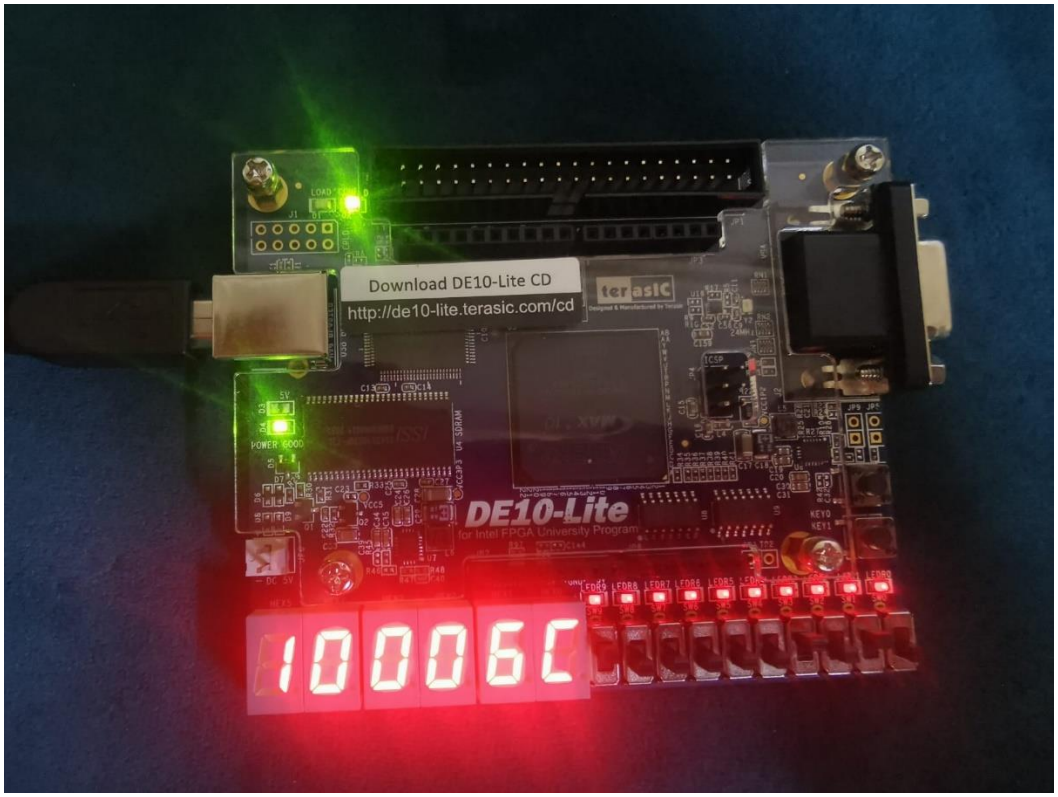
# Results

## 1.15 FPGA

**Note**: In every FPGA benchmark image, the first two bits represent the register number, while the remaining bits represent the register value.

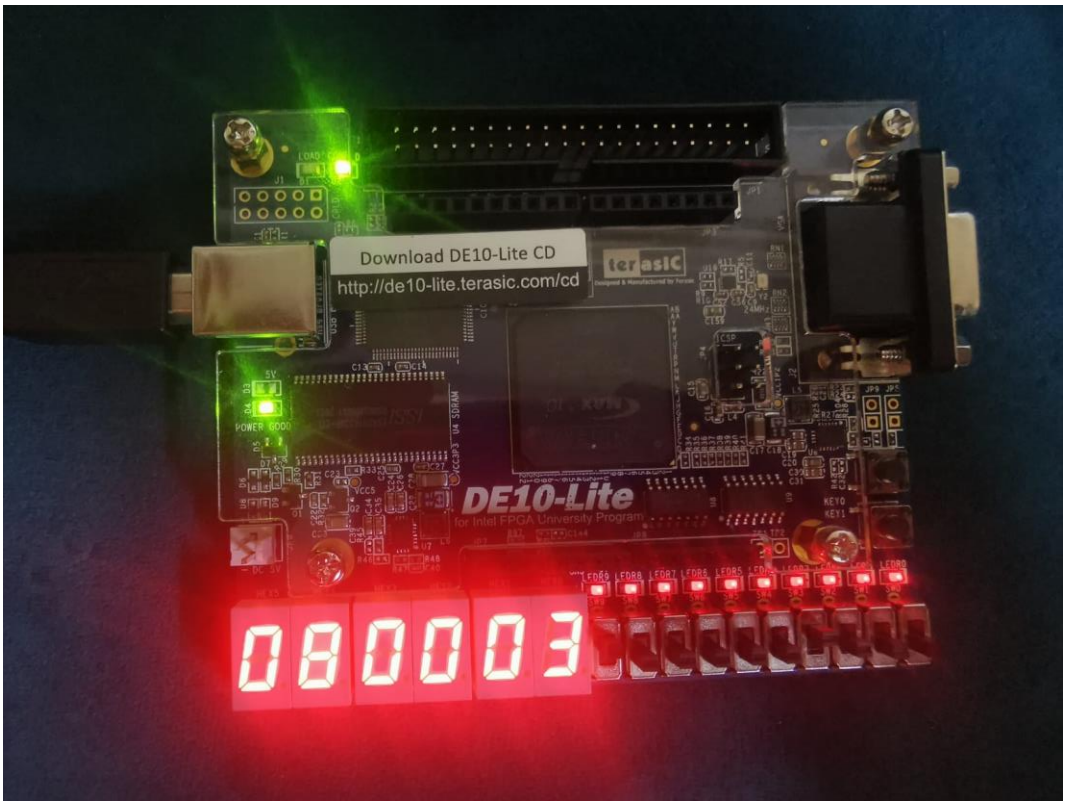
### 1.15.1 Benchmark one

#### *1.15.1.1 Register 10 ( the first two bit is register number ,other is register value)*

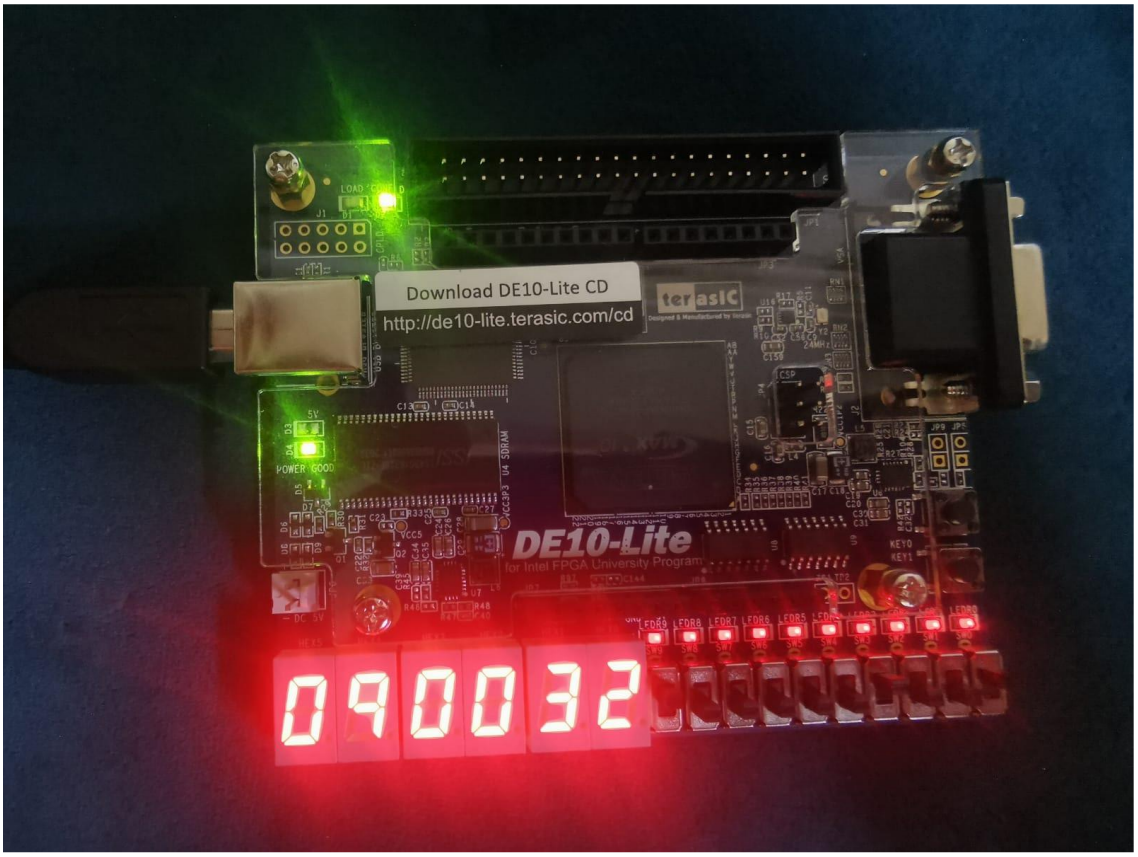



GitHub link → https://github.com/OmarAl-Saleh/MIPS

### *1.15.1.2 Register 8*

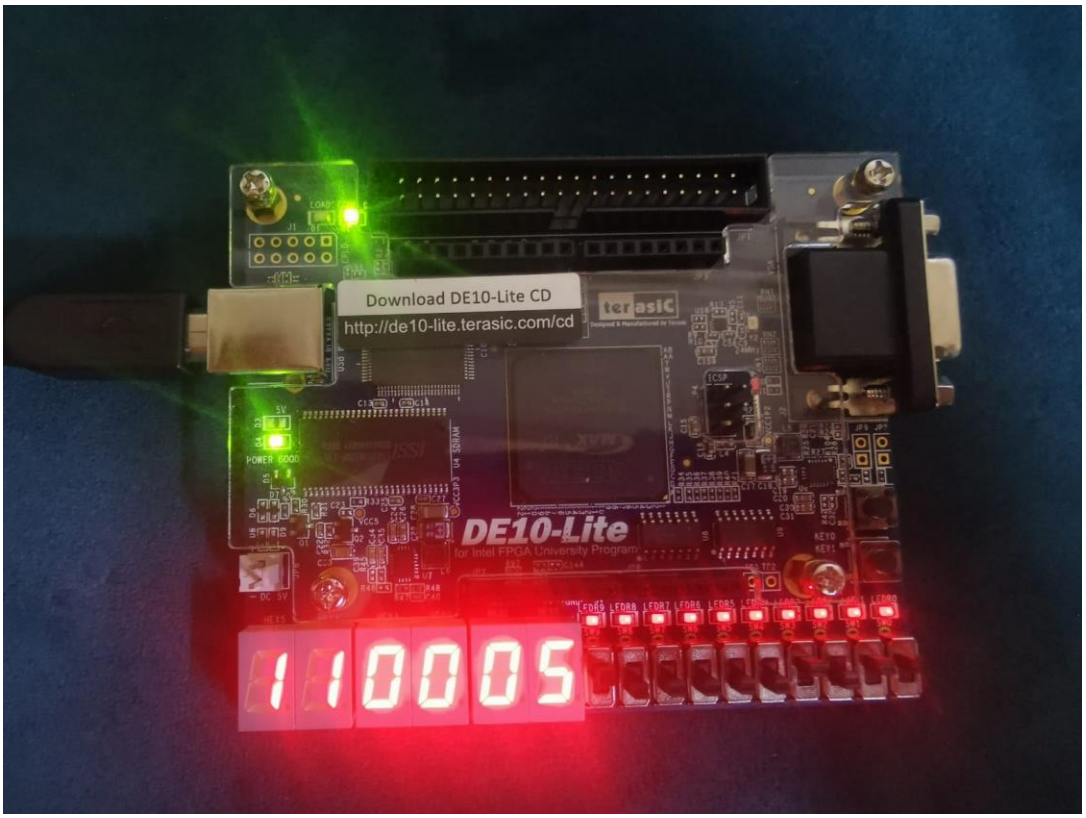



### *1.15.1.3 Register 9*




GitHub link → https://github.com/OmarAl-Saleh/MIPS

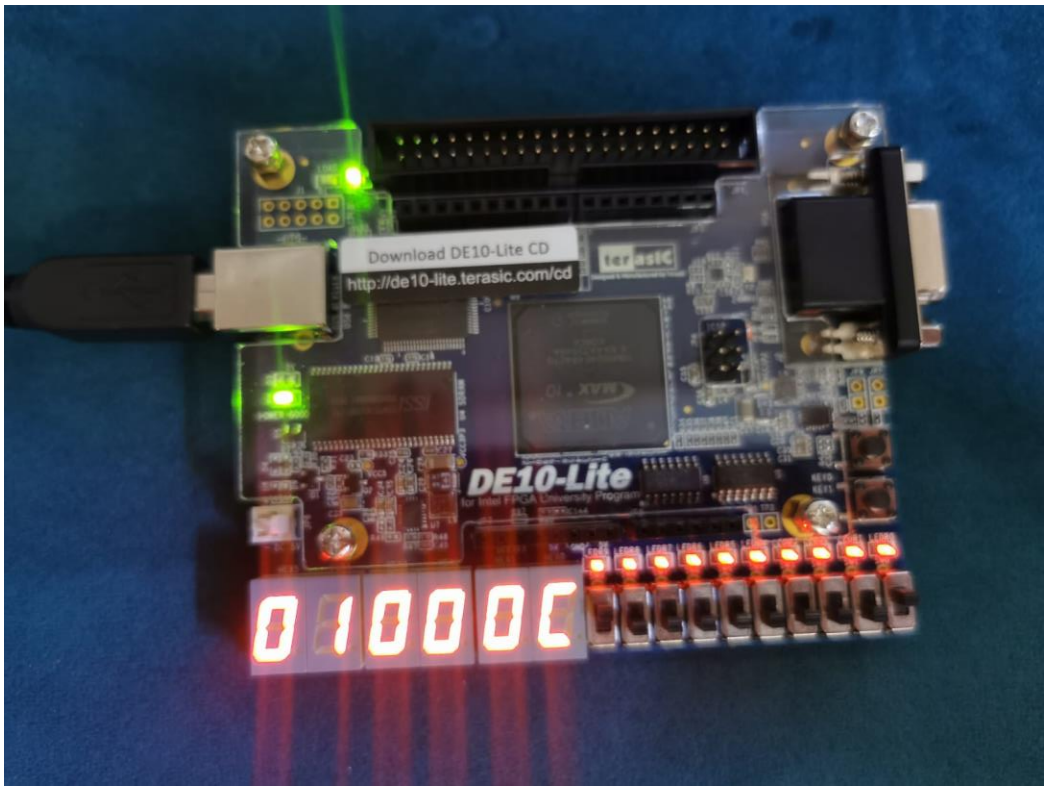
#### *1.15.1.4 Register 11*



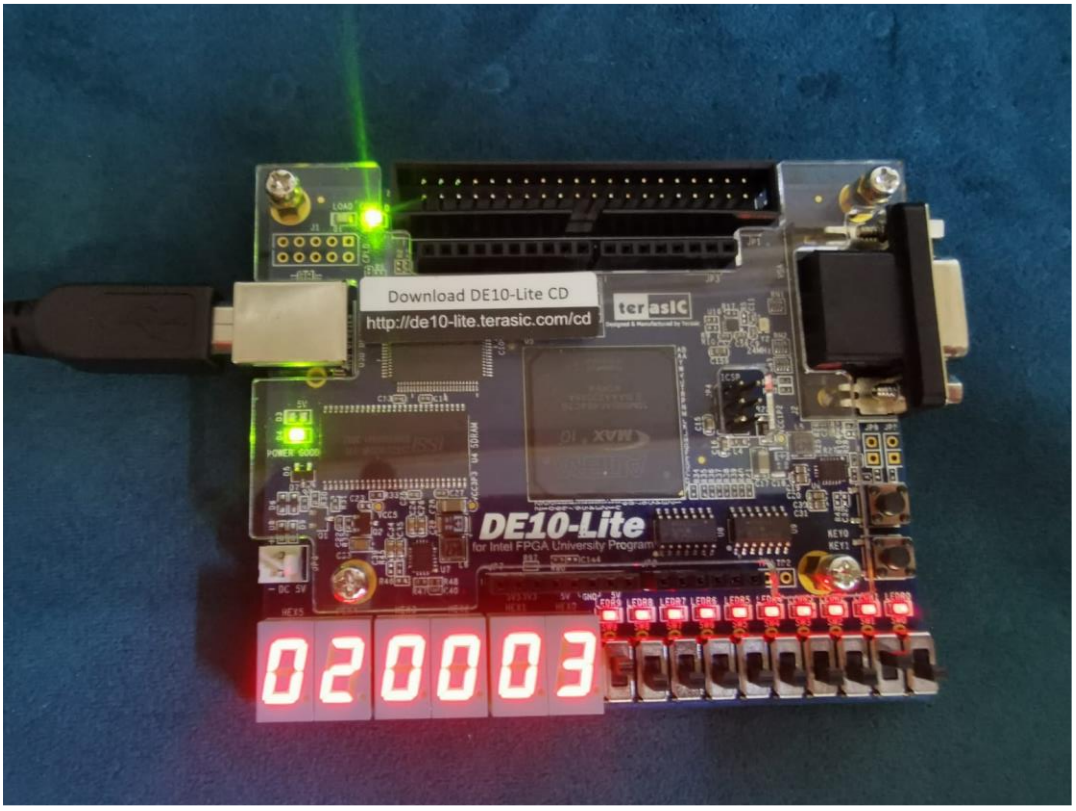

GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.15.2 Testcase 2

### *1.15.2.1 Register 1*

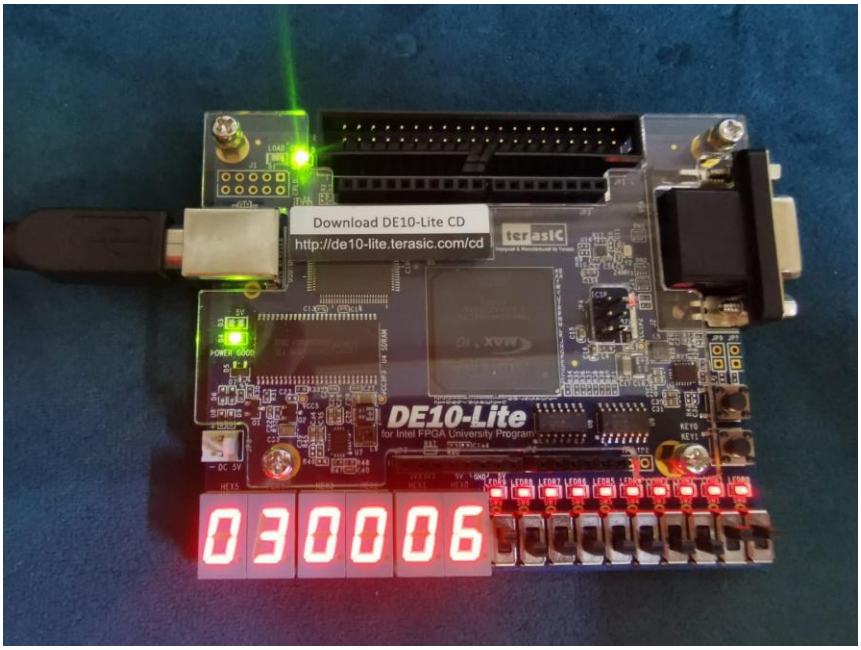



### *1.15.2.2 Register 2*

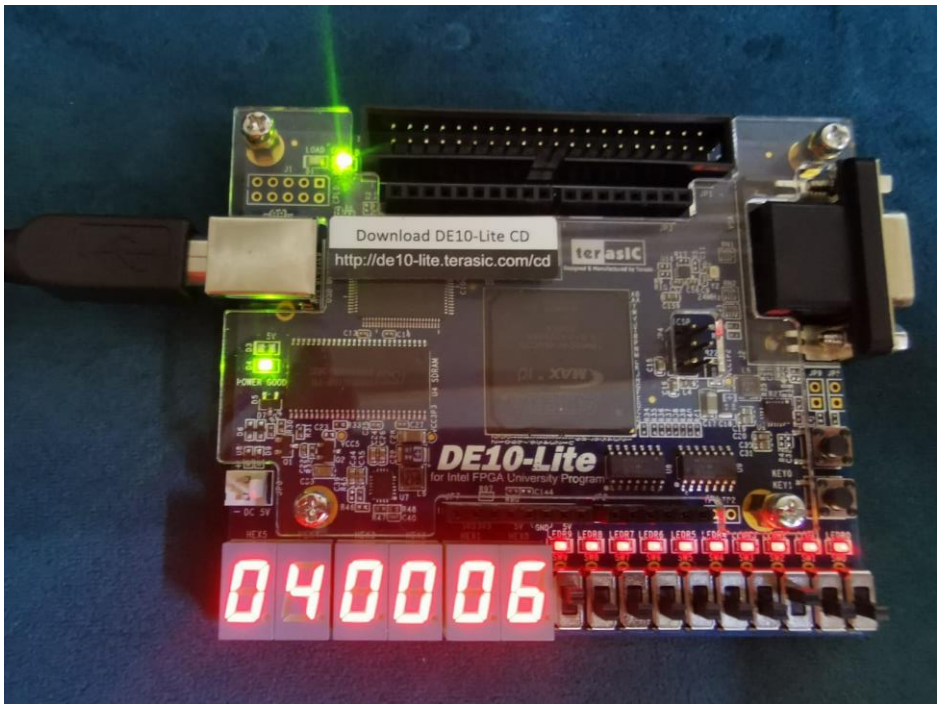



GitHub link → https://github.com/OmarAl-Saleh/MIPS

### *1.15.2.3 Register 3*



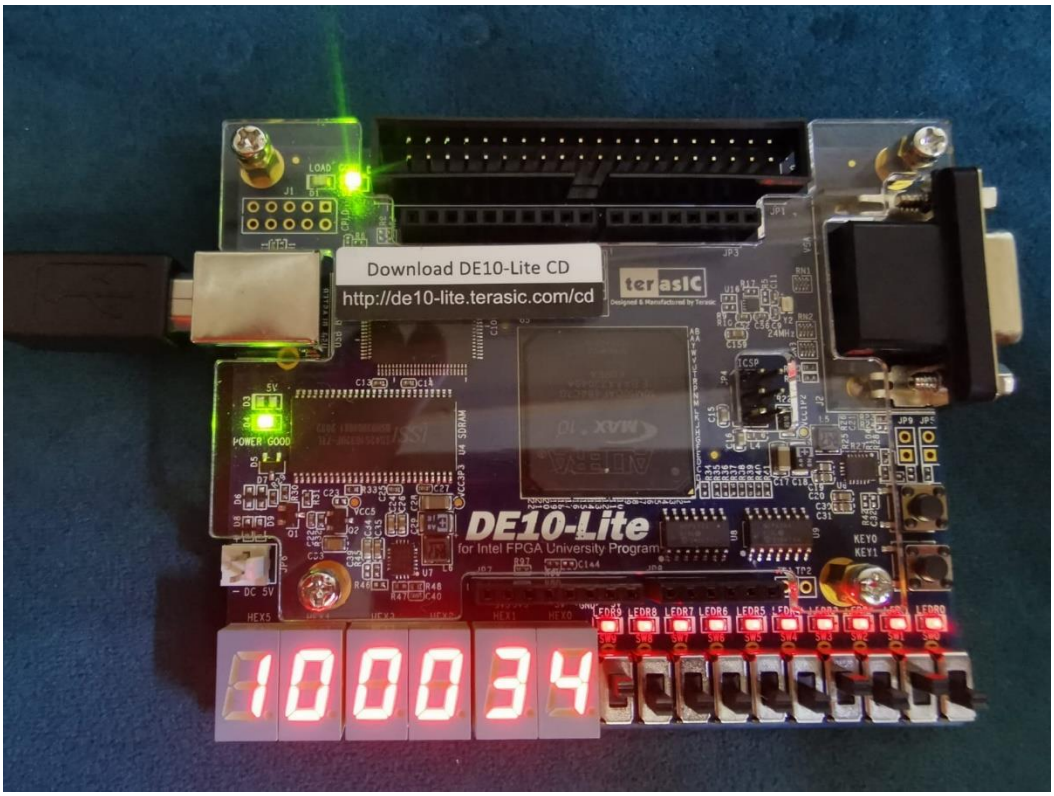

### *1.15.2.4 Register 4*



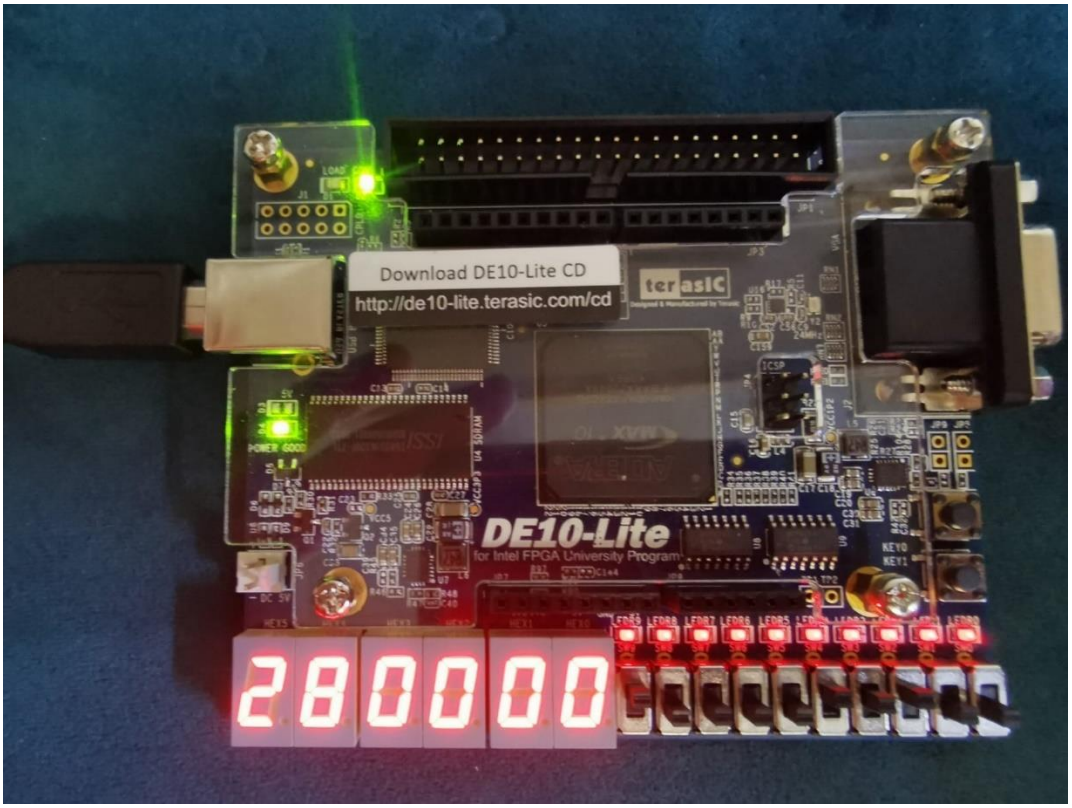

GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.15.3 Testcase 3

### *1.15.3.1 Register 10*

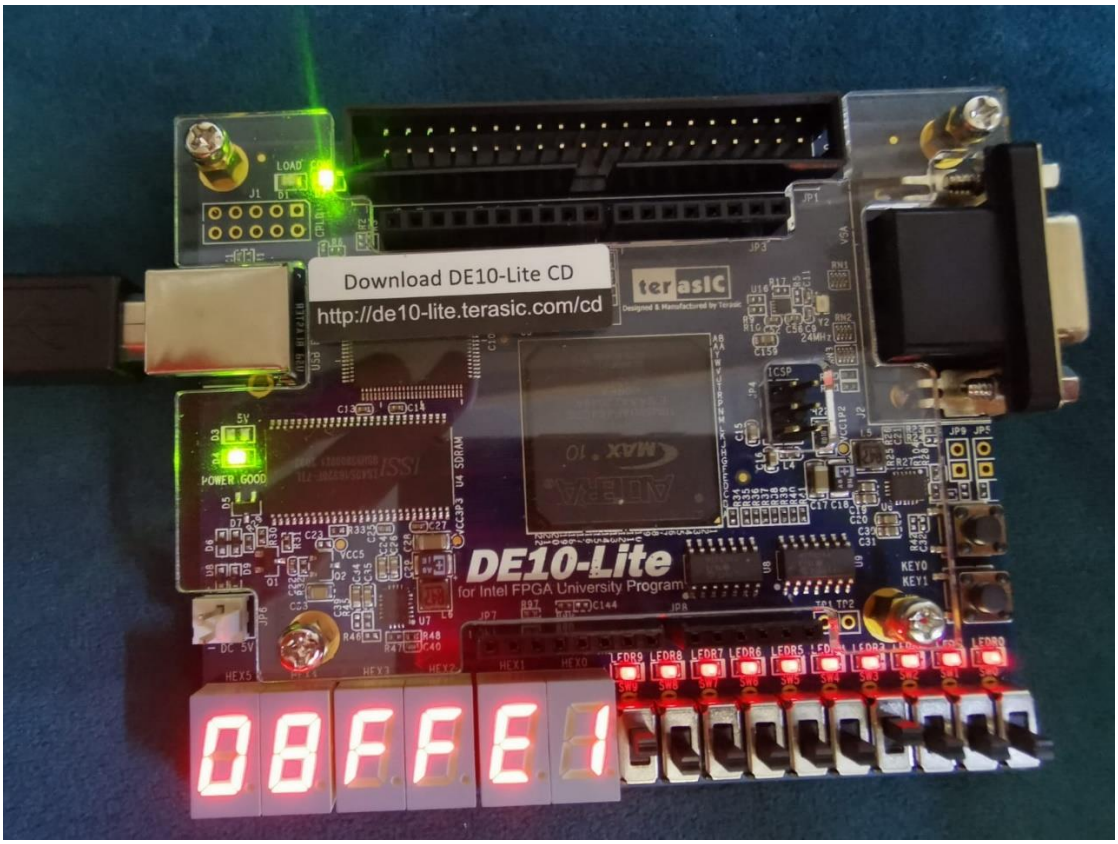



### *1.15.3.2 Register 28*

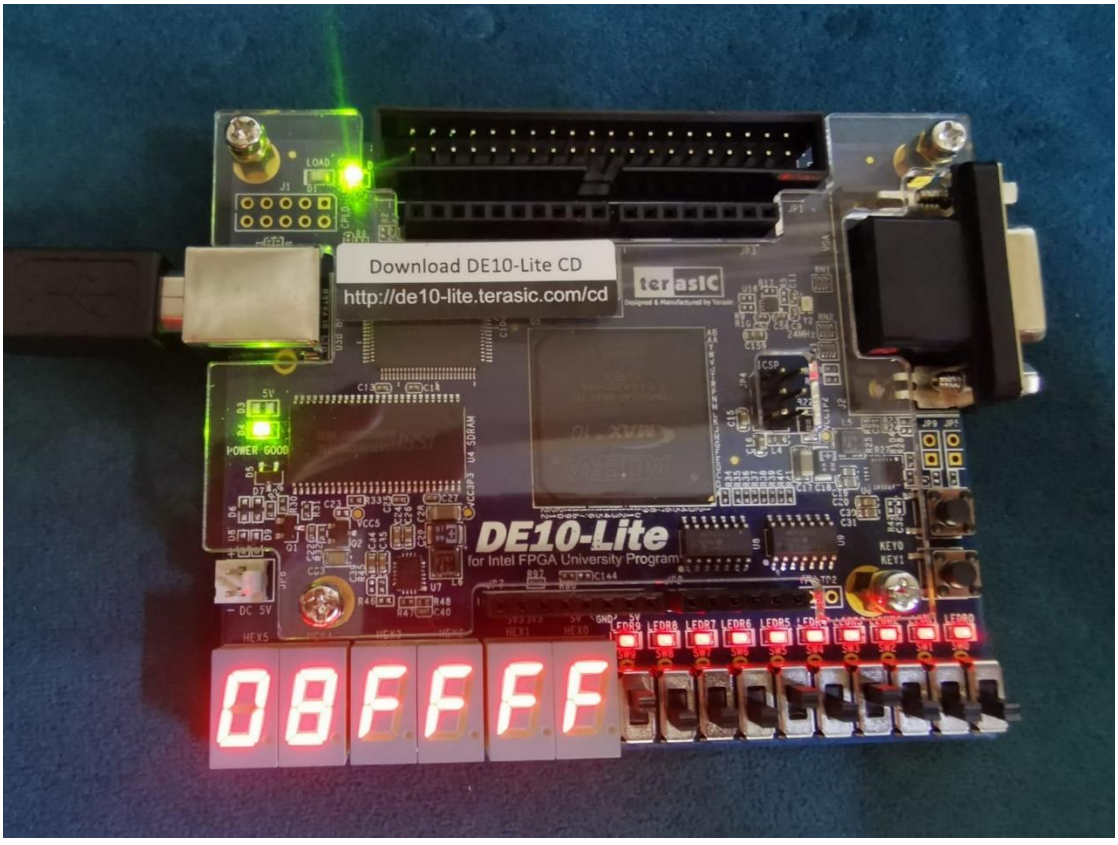



GitHub link → https://github.com/OmarAl-Saleh/MIPS

### *1.15.3.3 Register 8*

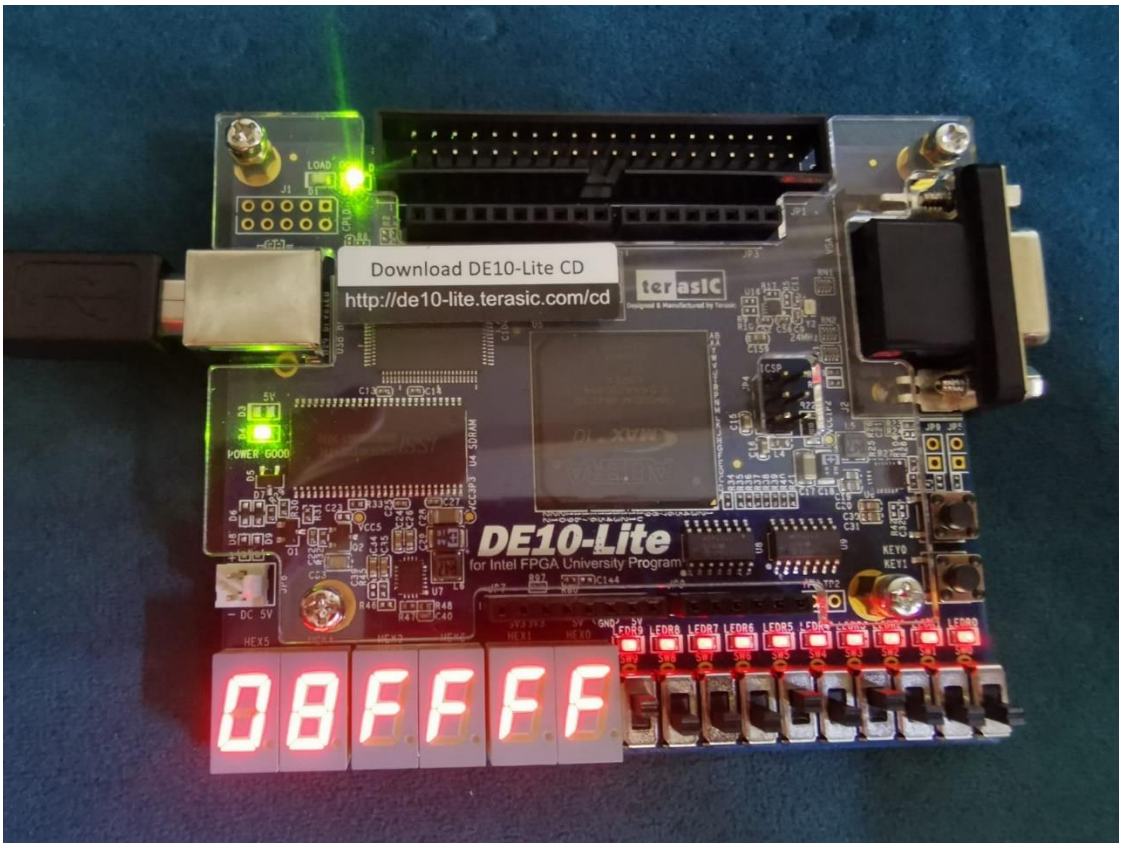



### *1.15.3.4 Register 8 upper*




GitHub link → https://github.com/OmarAl-Saleh/MIPS

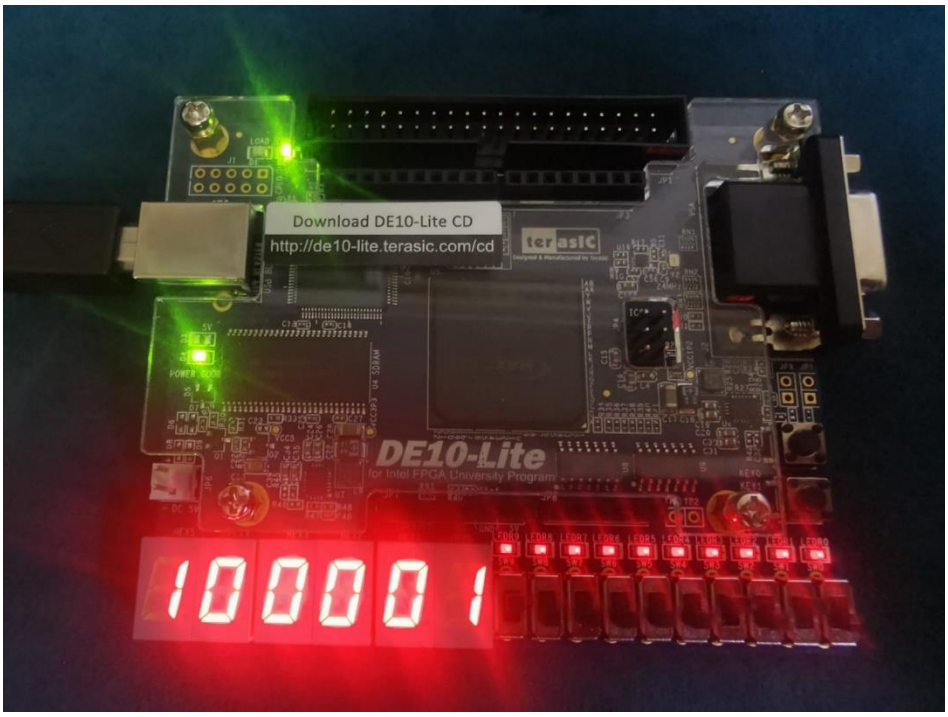
### *1.15.3.5 Register 9*



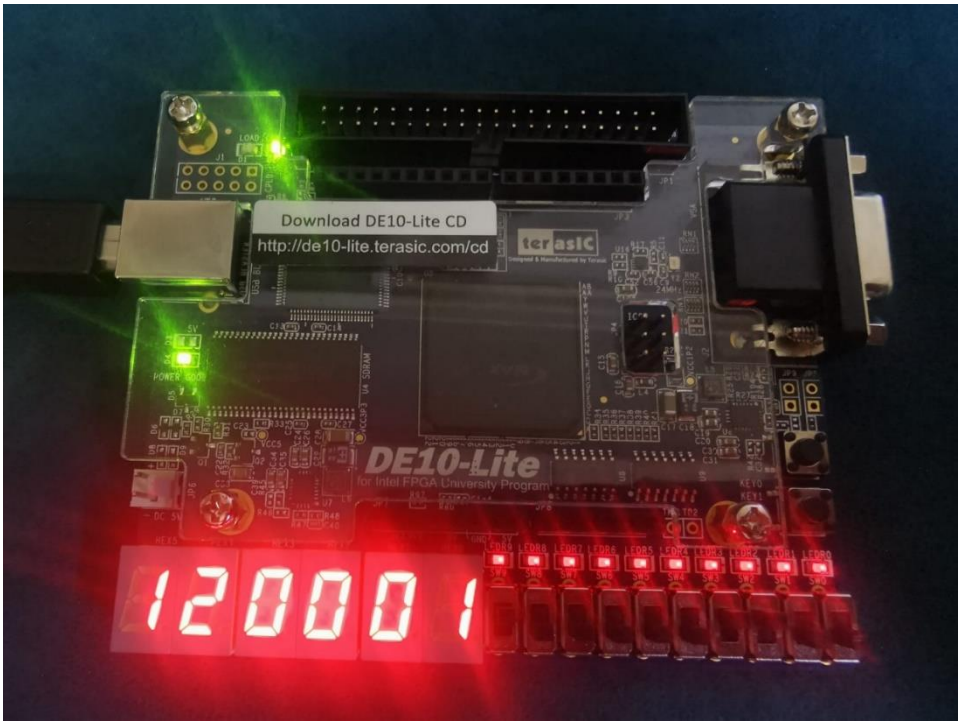

GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.15.4 Testcase 4

### *1.15.4.1 Register 10*

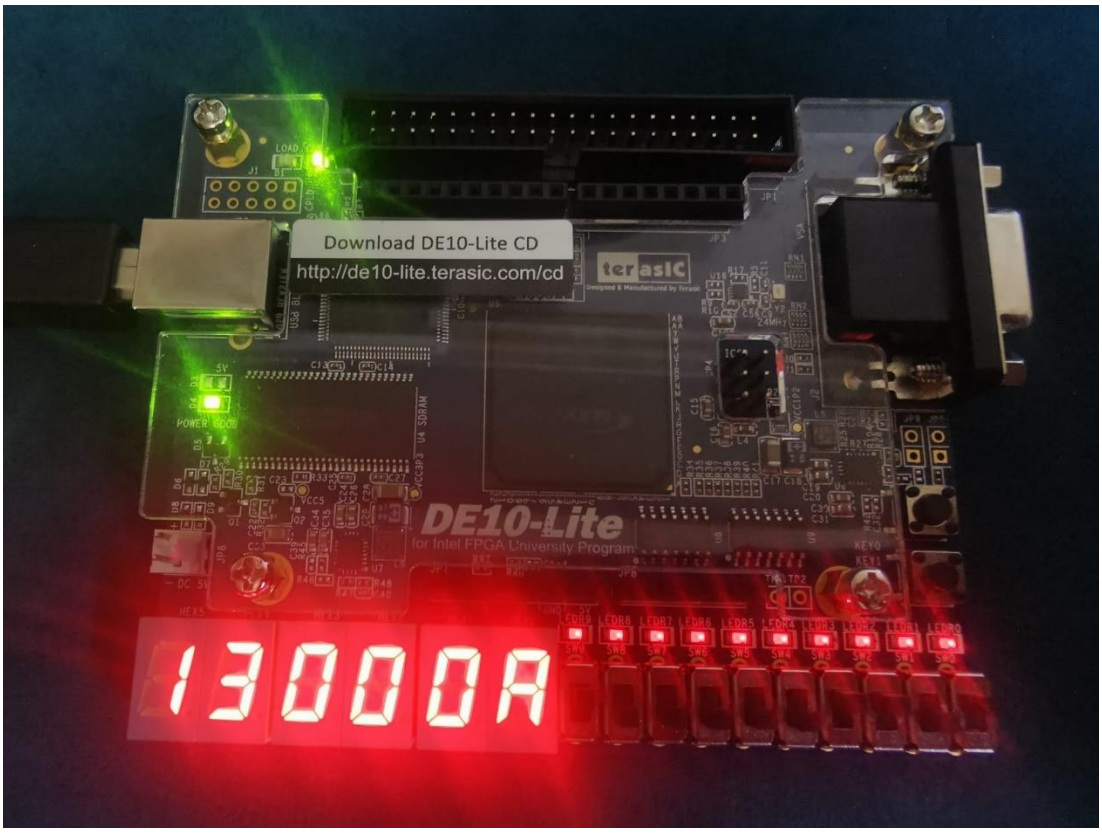



### *1.15.4.2 Register 12*

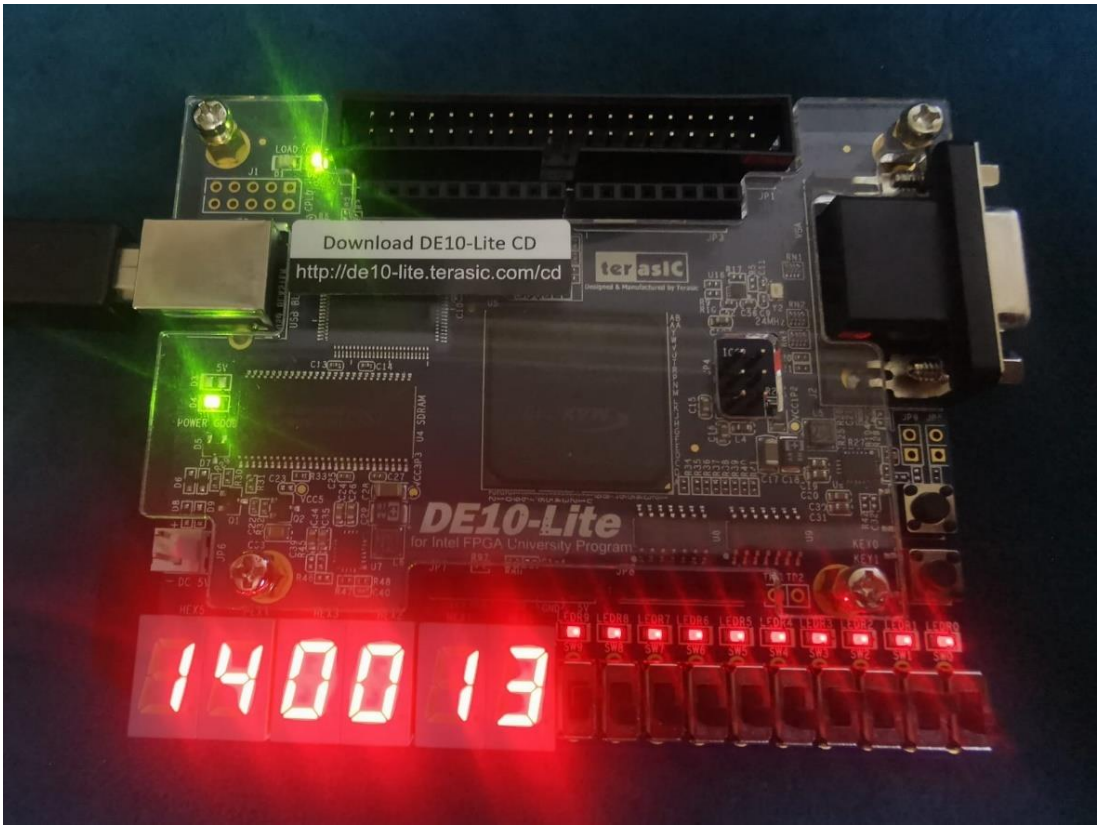



GitHub link → https://github.com/OmarAl-Saleh/MIPS

### *1.15.4.3 Register 13*

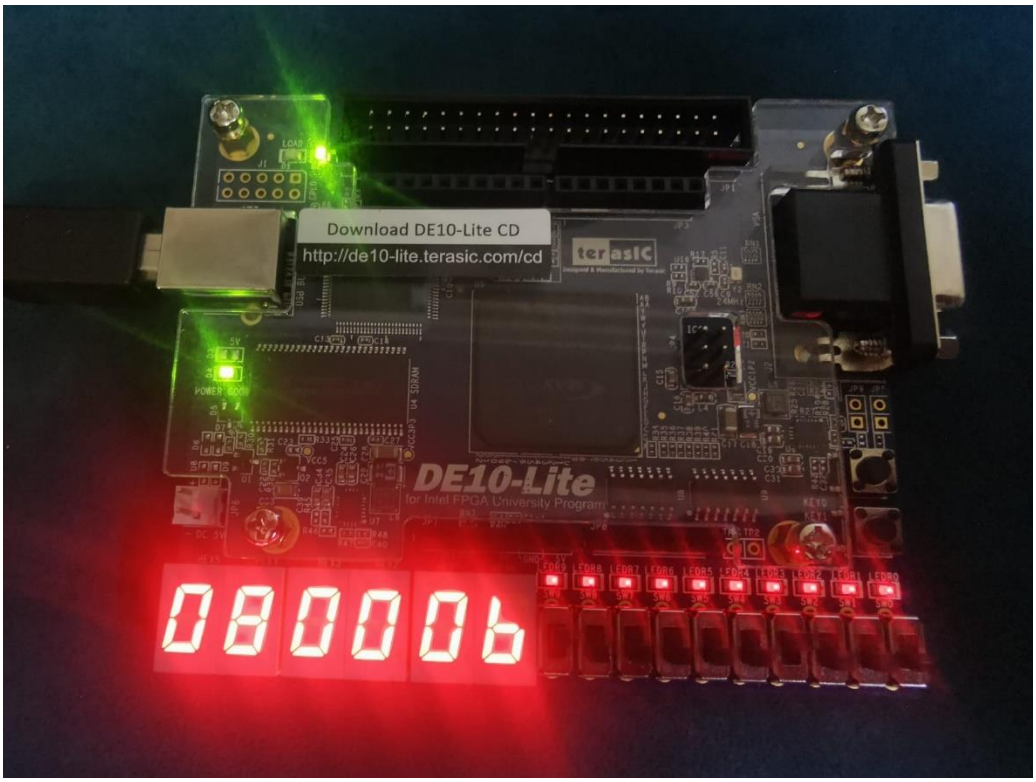



### *1.15.4.4 Register 14*

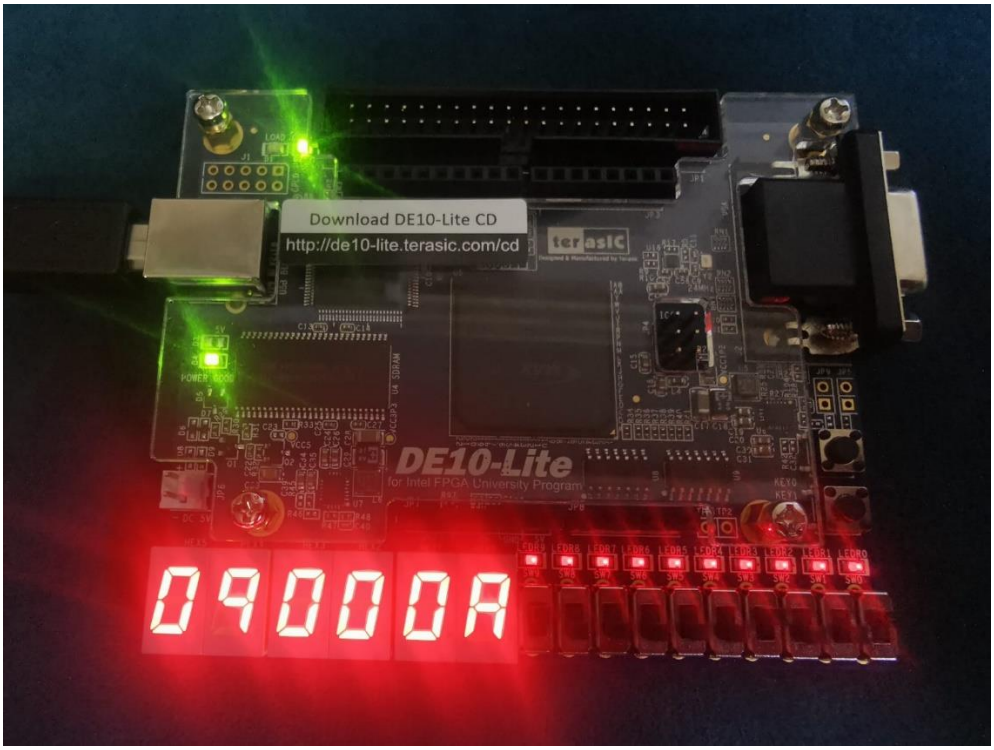



GitHub link → https://github.com/OmarAl-Saleh/MIPS

### *1.15.4.5 Register 8*




### *1.15.4.6 Register 9*

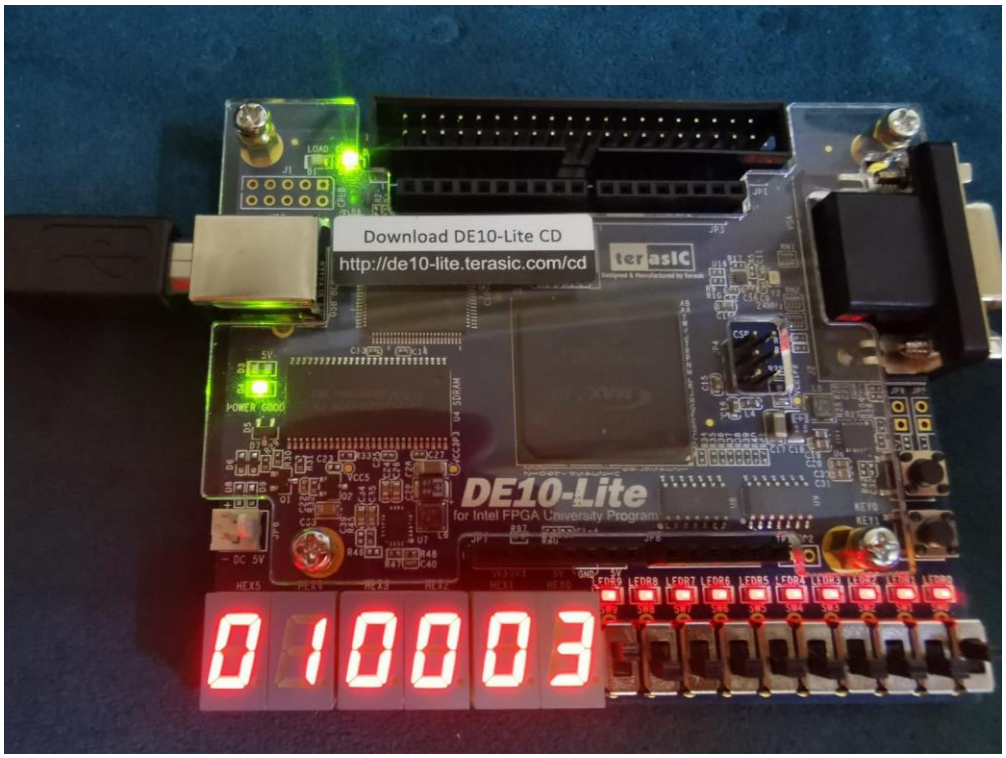



GitHub link → https://github.com/OmarAl-Saleh/MIPS

1.15.5Testcase 5

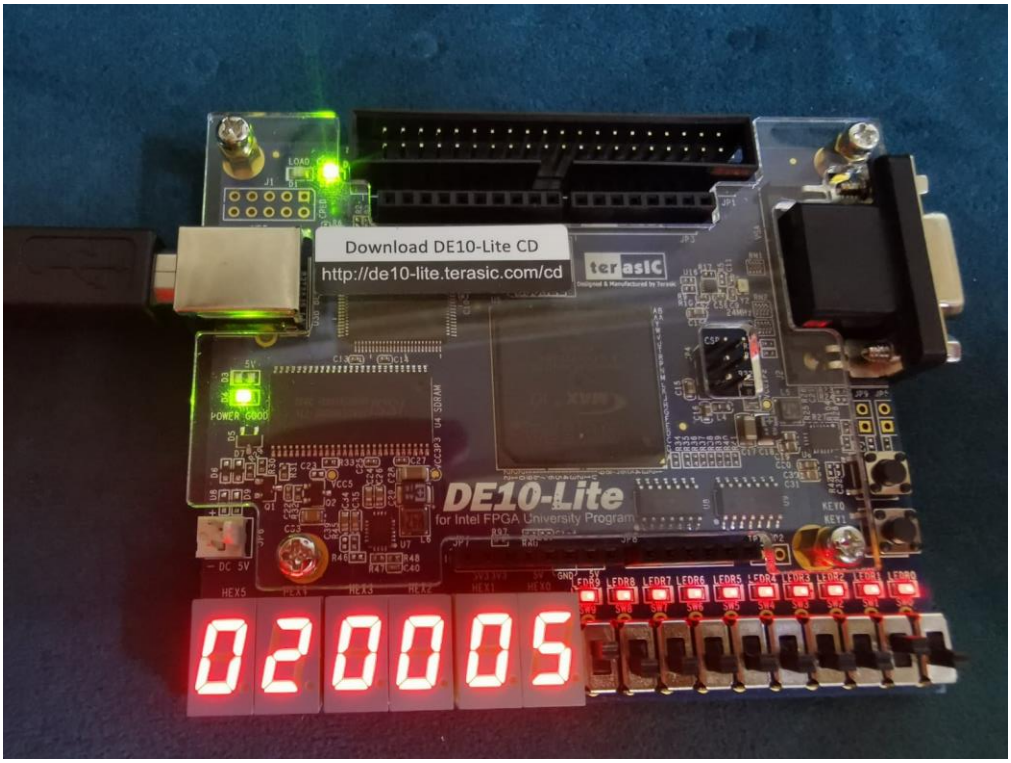
# When a=2

### *1.15.5.1 Register 1*

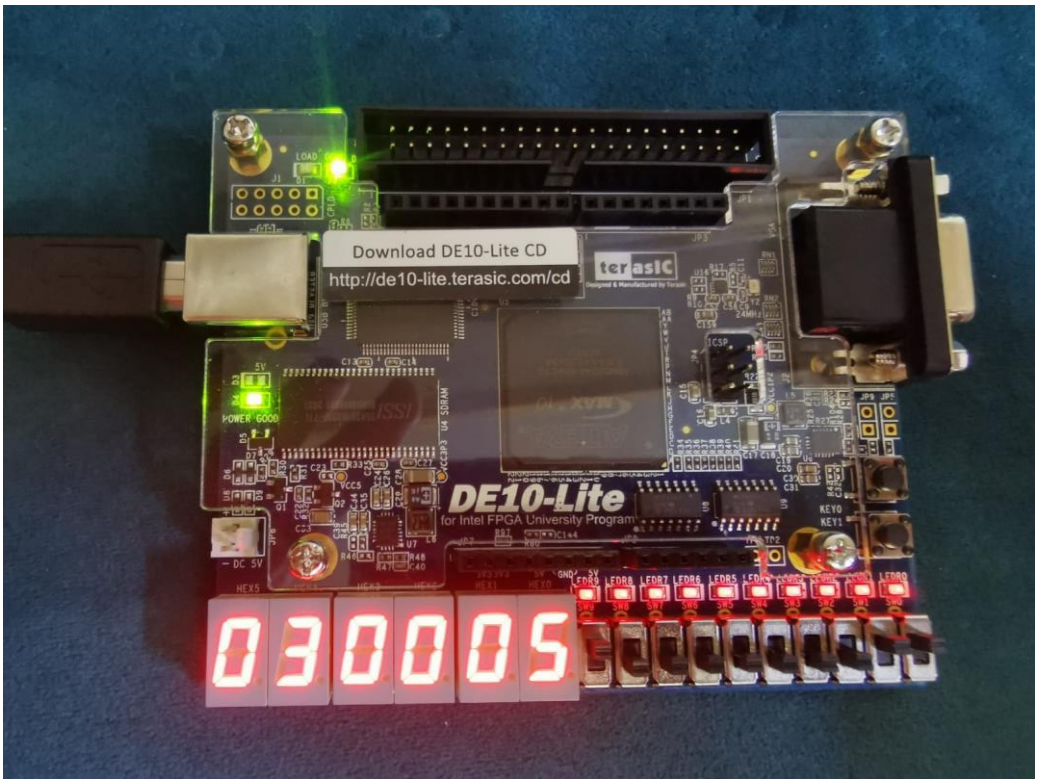



GitHub link → https://github.com/OmarAl-Saleh/MIPS

### *1.15.5.2 Register 2*



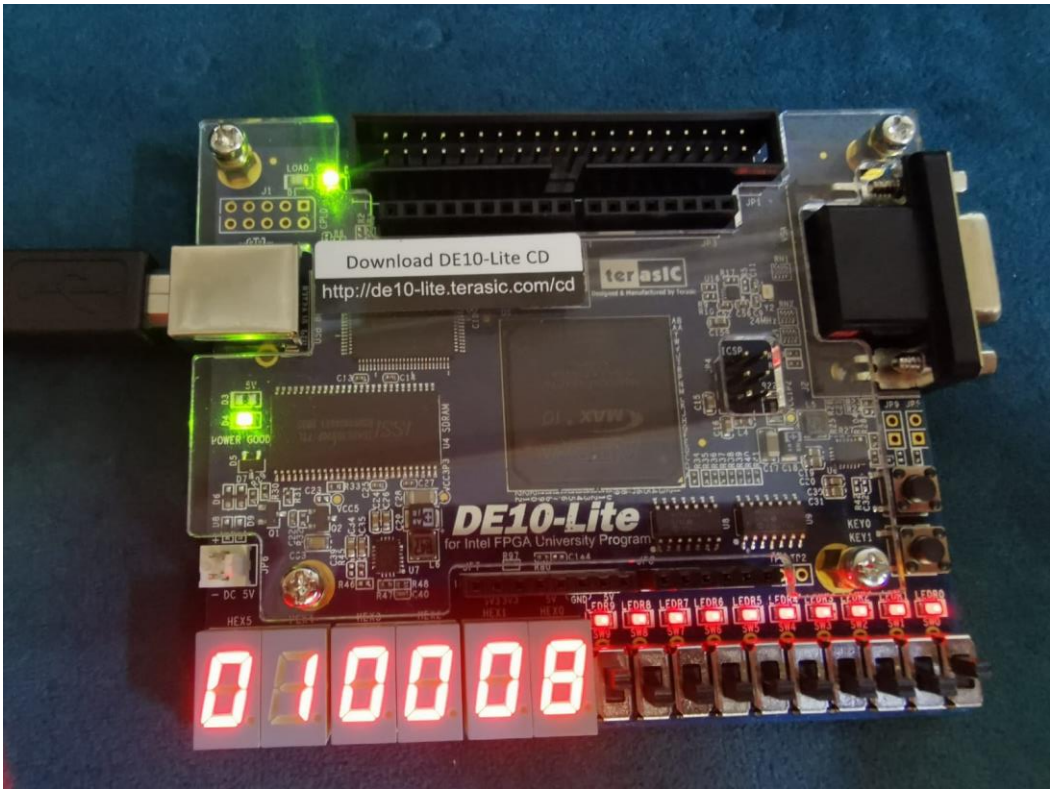

### *1.15.5.3 Register 3*



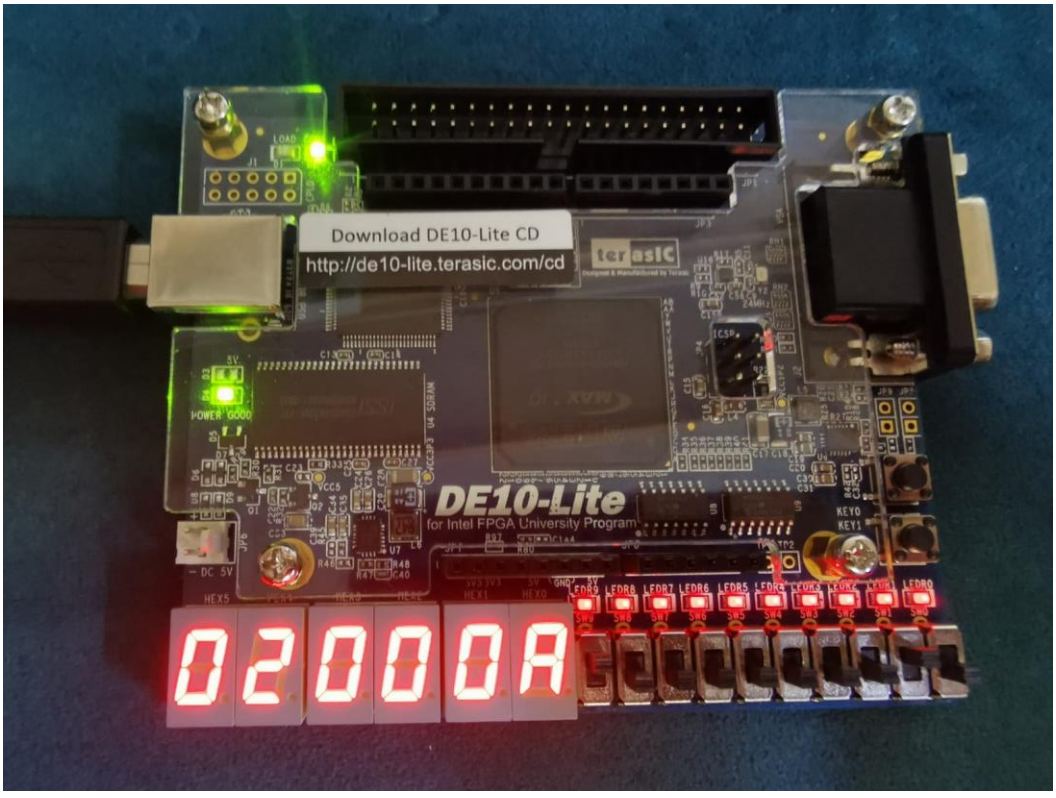

GitHub link → https://github.com/OmarAl-Saleh/MIPS

# When a =6

### *1.15.5.4 Register 1*

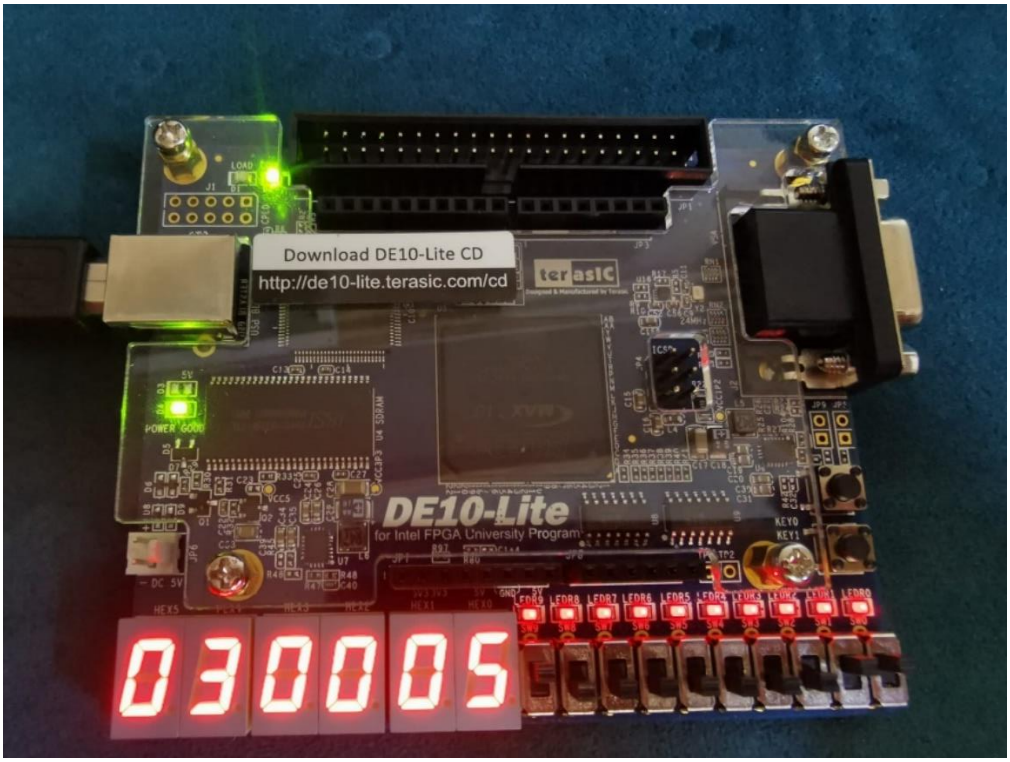



### *1.15.5.5 Register 2*




GitHub link → https://github.com/OmarAl-Saleh/MIPS

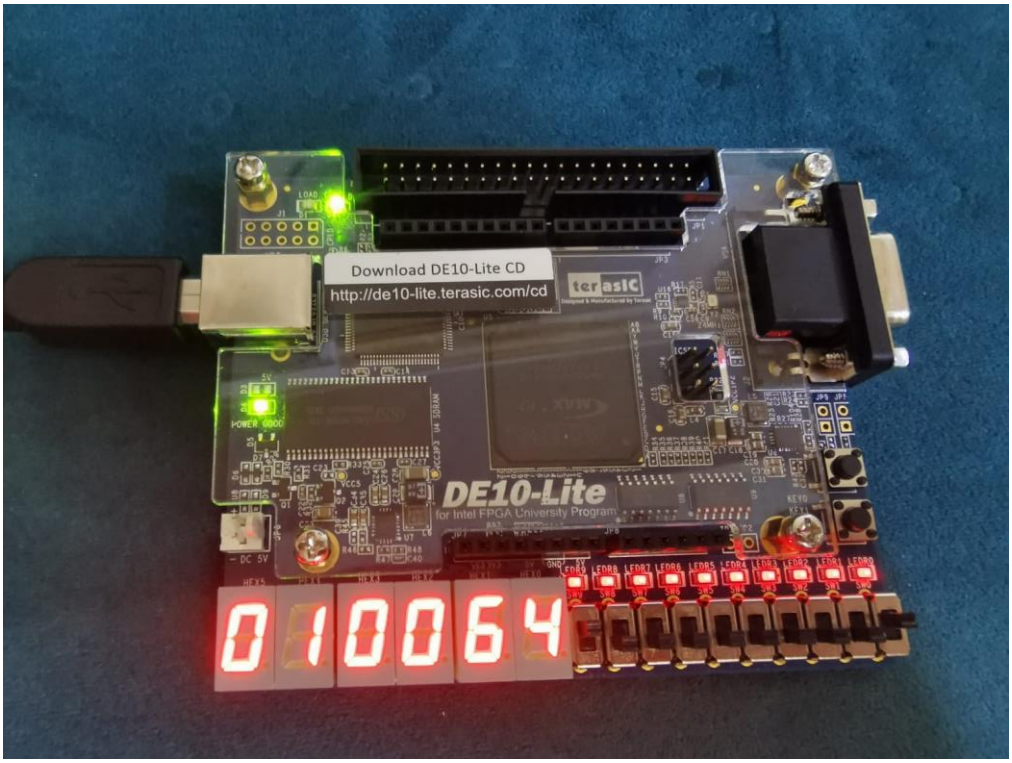
### *1.15.5.6 Register 3*



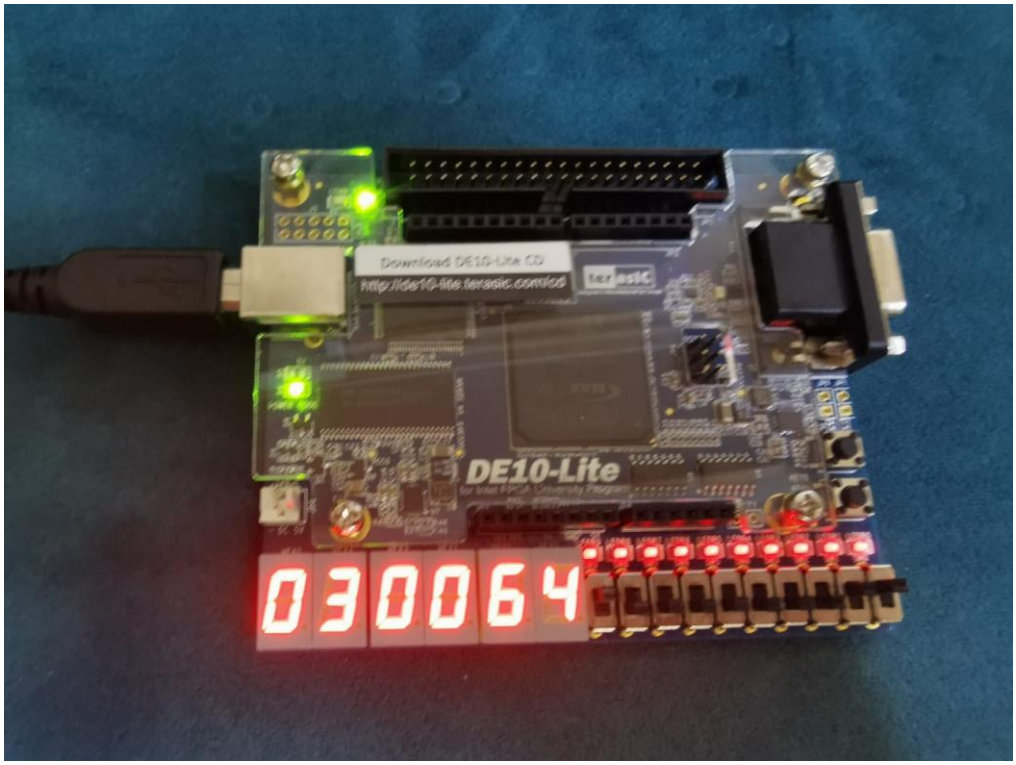

GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.15.6 Testcase 6

### *1.15.6.1 Register 1*

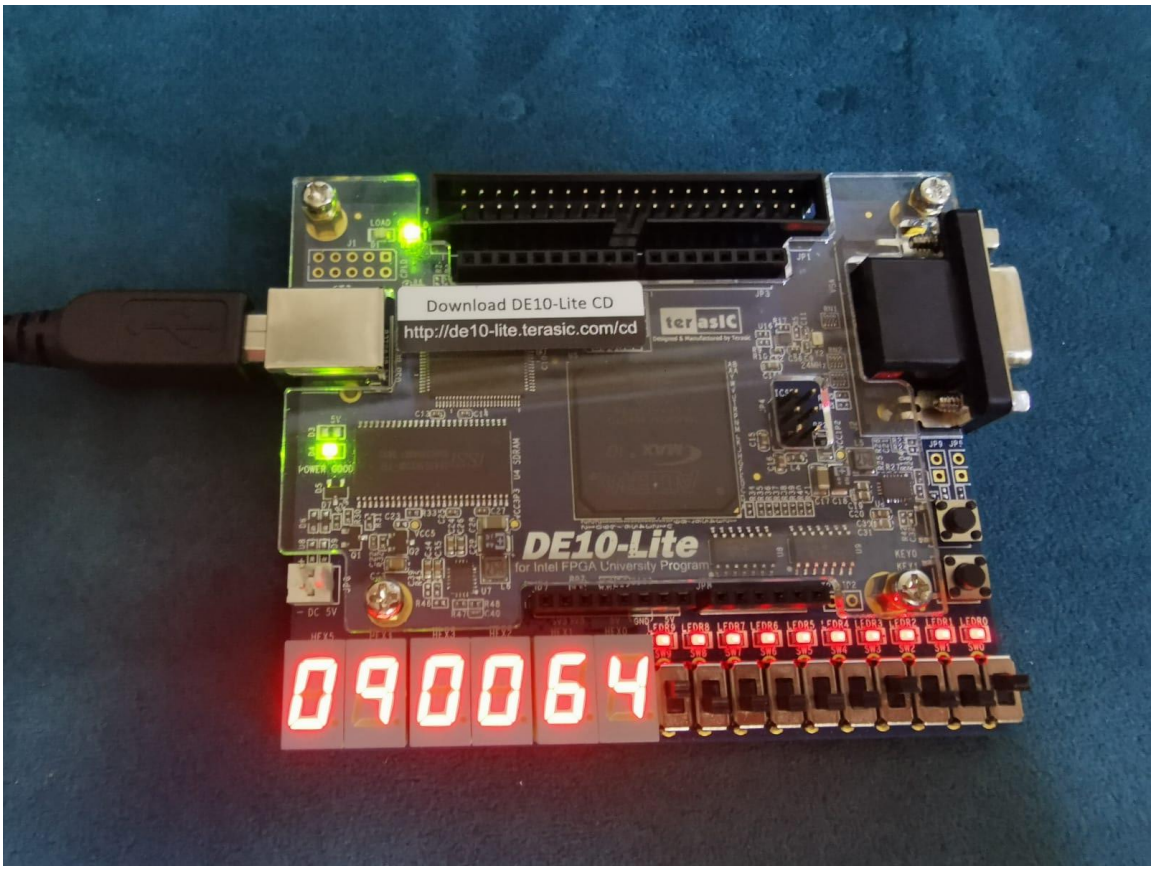



### *1.15.6.2 Register 3*




GitHub link → https://github.com/OmarAl-Saleh/MIPS

### *1.15.6.3 Register 9*




GitHub link → https://github.com/OmarAl-Saleh/MIPS

1.16 Benchmark

# 1.16.1 Test case one

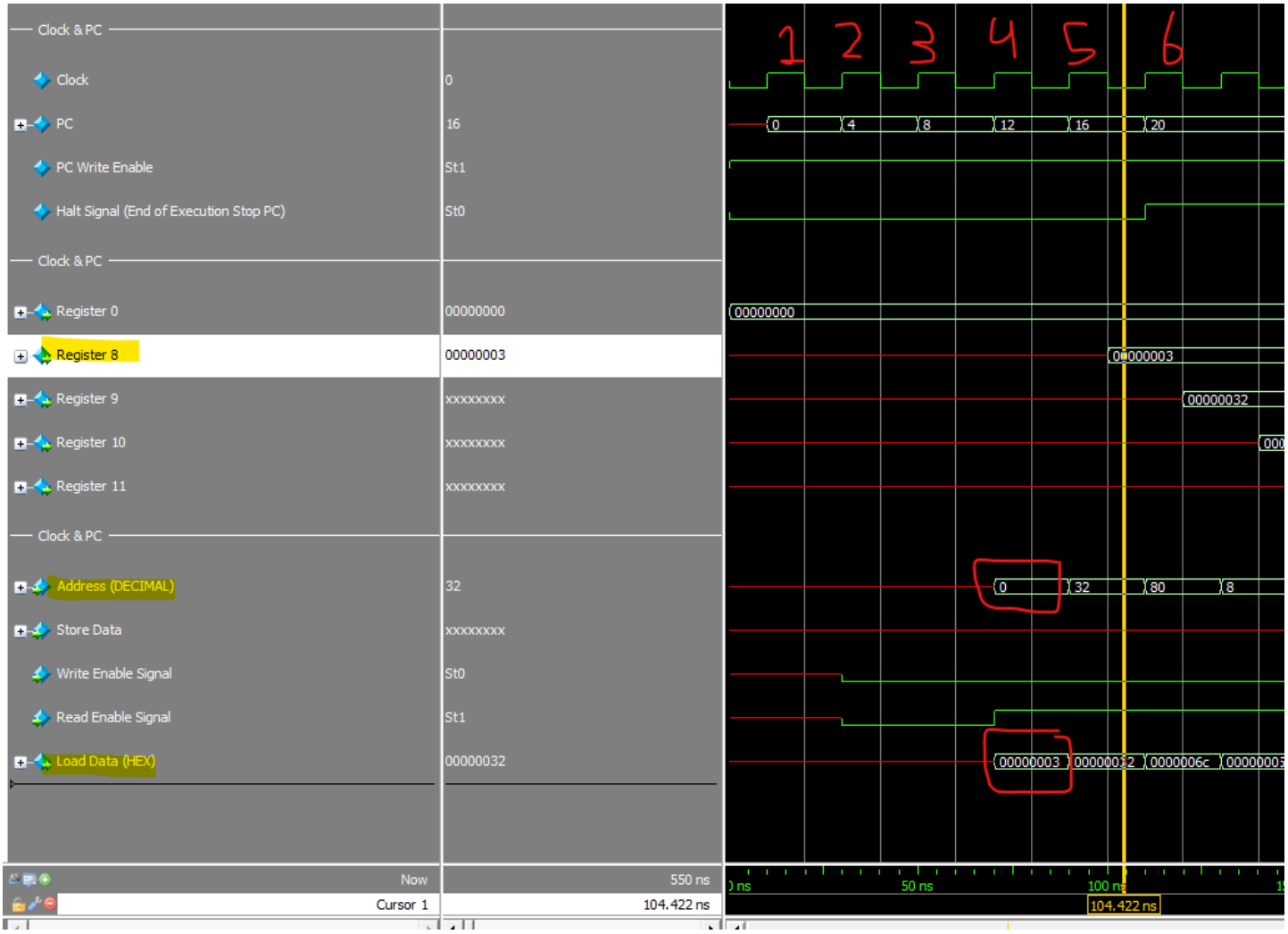
*1.16.1.1 Instruction memory and data memory*

```
//testcase 1

inst_mem[0] = 32'b10001100000010000000000000000000; //LW R8, 0(R0)
inst_mem[1] = 32'b10001100000010010000000000100000; //LW R9, 0x20(R0)
inst_mem[2] = 32'b10001100000010100000000000010000; //LW R10, 0x50(R0)
inst_mem[3] = 32'b10001100000010110000000000001000; //LW R11, 0x8(R0)


//-------------------------------------------------------------


/* LAST ADDRESS */ inst_mem[4] = 32'b10110100001000100001100000100000; //Halt
// every program should end with halt signal
```

```
//testcase 1
mem[0] = 32'h00000003;
mem[1] = 32'h00000008;
mem[2] = 32'h00000005;
mem[3] = 32'h00000002;
mem[4] = 32'h00000032;
mem[5] = 32'h00000032;
mem[6] = 32'h00000032;
mem[7] = 32'h00000032;
mem[8] = 32'h00000032;
mem[9] = 32'h00000032;
mem[10] = 32'h00000032;
mem[11] = 32'h00000032;
mem[12] = 32'h00000032;
mem[13] = 32'h00000032;
mem[14] = 32'h00000032;
mem[15] = 32'h00000032;
mem[16] = 32'h00000032;
mem[17] = 32'h00000068;
mem[18] = 32'h00000065;
mem[19] = 32'h0000006C;
mem[20] = 32'h0000006C;
mem[21] = 32'h0000006F;
mem[22] = 32'h00000000;
```

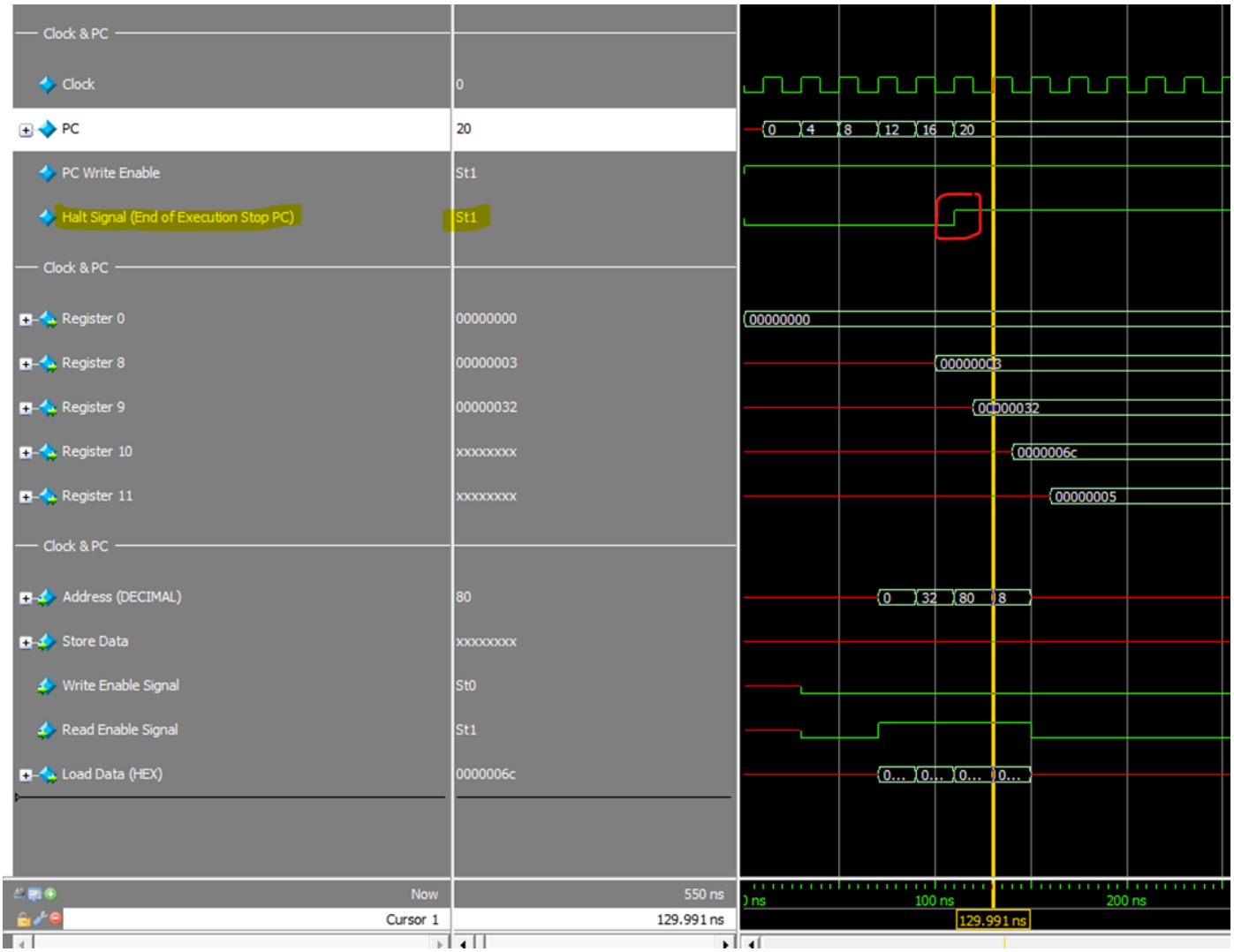
GitHub link → https://github.com/OmarAl-Saleh/MIPS

### 1.16.1.2 *First interaction* ***LW R8, 0(R0) (CPU before execution***



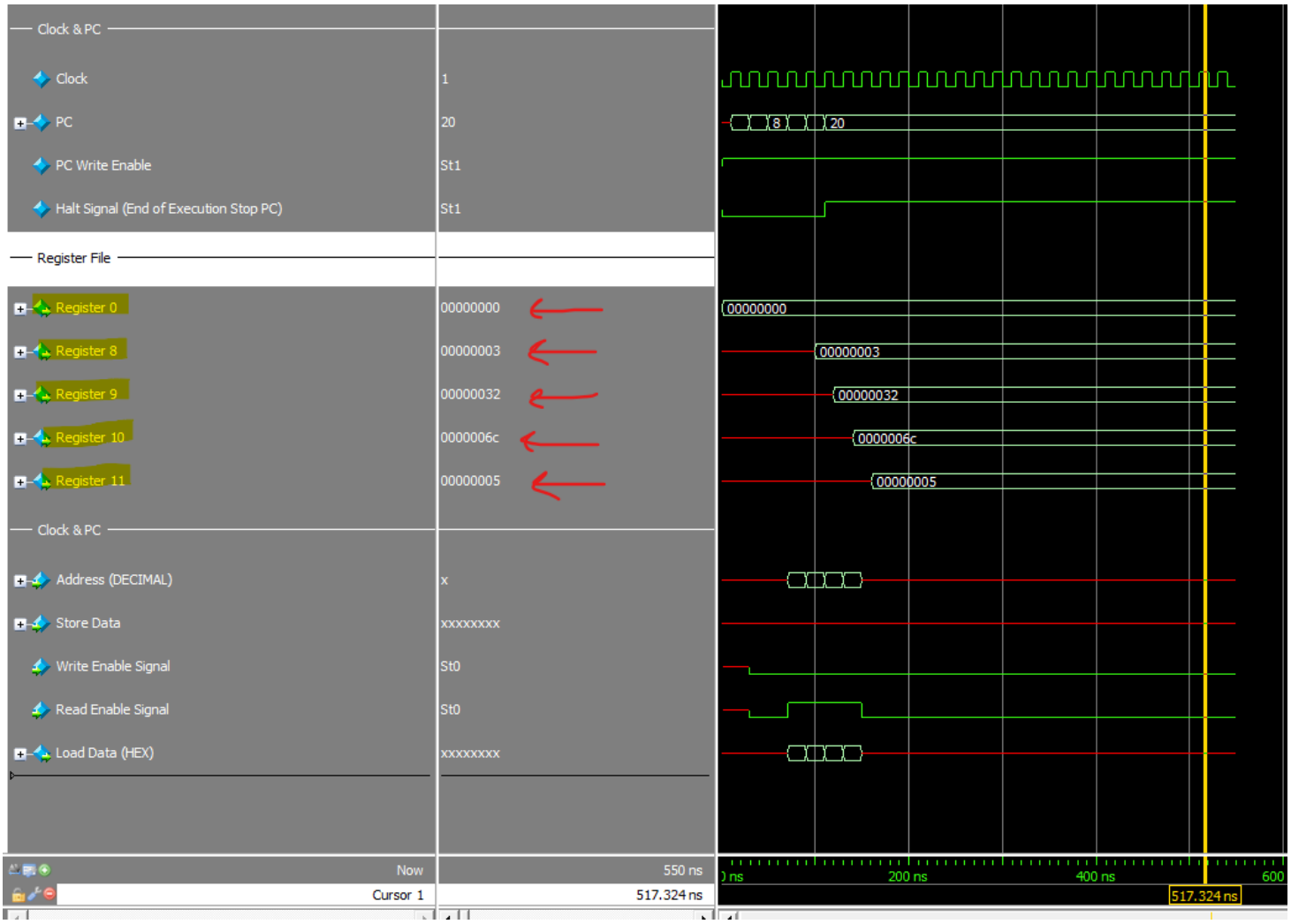

GitHub link → https://github.com/OmarAl-Saleh/MIPS

### 1.16.1.3 Halt




GitHub link → https://github.com/OmarAl-Saleh/MIPS

## *after execution the final values*




GitHub link → https://github.com/OmarAl-Saleh/MIPS

# 1.17 Test case 2

## 1.17.1 Data memory and instruction memory

```
//testcase 2

inst_mem[0] = 32'b10001100000000010000000000001000; //LW R1, 8(R0)
inst_mem[1] = 32'b00000000001000000000100010000000; //SLL R1, R1, 2
inst_mem[2] = 32'b10101100000000010000000000000100; //SW R1, 4(R0)
inst_mem[3] = 32'b10001100000000010000000000010000; //LW R2, 16(R0)
inst_mem[4] = 32'b10001100000000011000000000010000; //LW R3, 16(R0)
inst_mem[5] = 32'b00000000010000000011000001000000; //SLL R3, R2, 1
inst_mem[6] = 32'b10101100000000011000000000001100; //SW R3, 12(R0)
inst_mem[7] = 32'b10001100000001000000000000001100; //LW R4, 12(R0)
```

Hazard & Forwrading

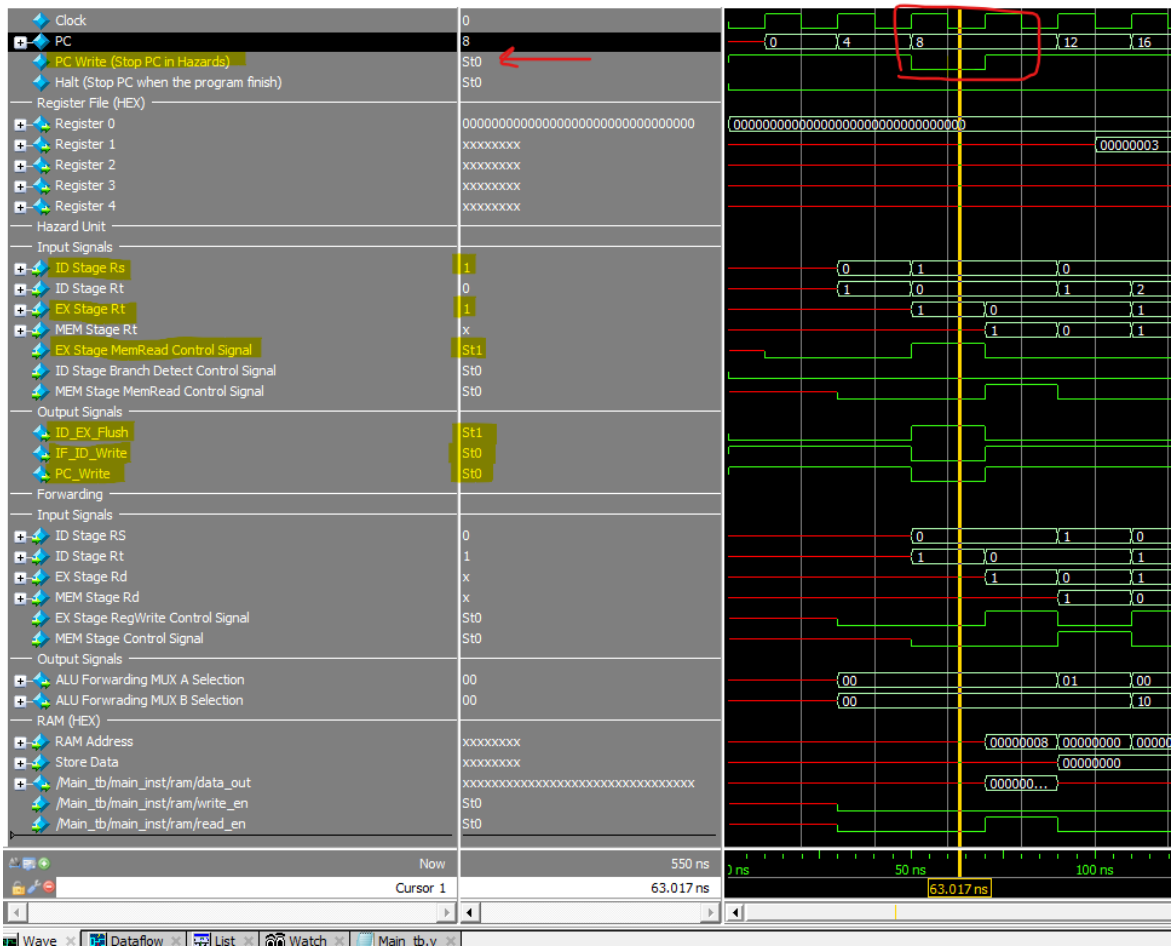
Forwarding

Forwarding

```
 //testcaase 2
mem[0] = 32'b00000000000000000000000000000011; // 3
mem[1] = 32'b00000000000000000000000000000011; // 3
mem[2] = 32'b00000000000000000000000000000011; // 3
mem[3] = 32'b00000000000000000000000000000011; // 3
mem[4] = 32'b00000000000000000000000000000011; // 3

/*
```

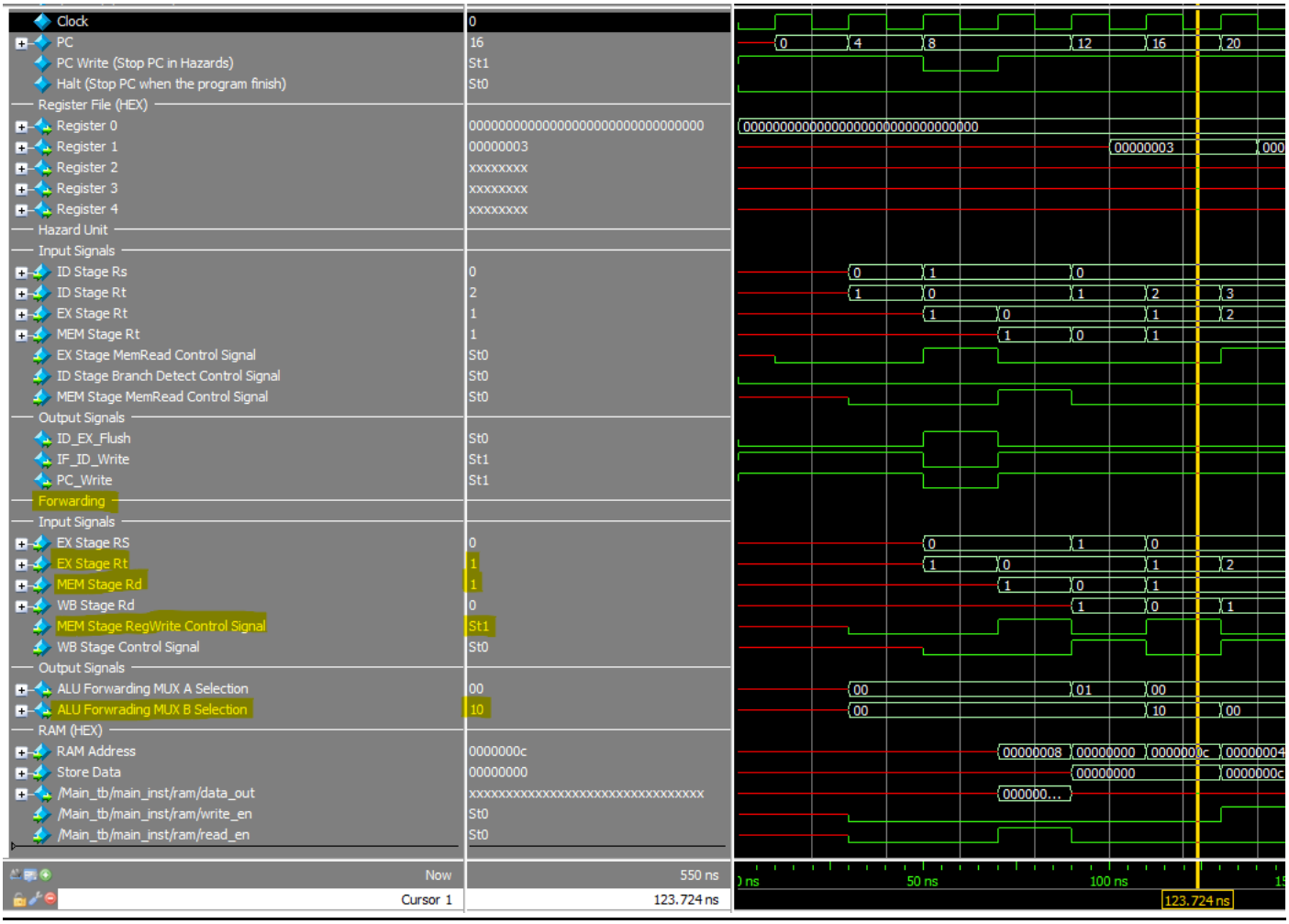
## 1.17.2 Second Instruction SLL R1, R1, 2 Hazard Catch with stall cycle

GitHub link → https://github.com/OmarAl-Saleh/MIPS



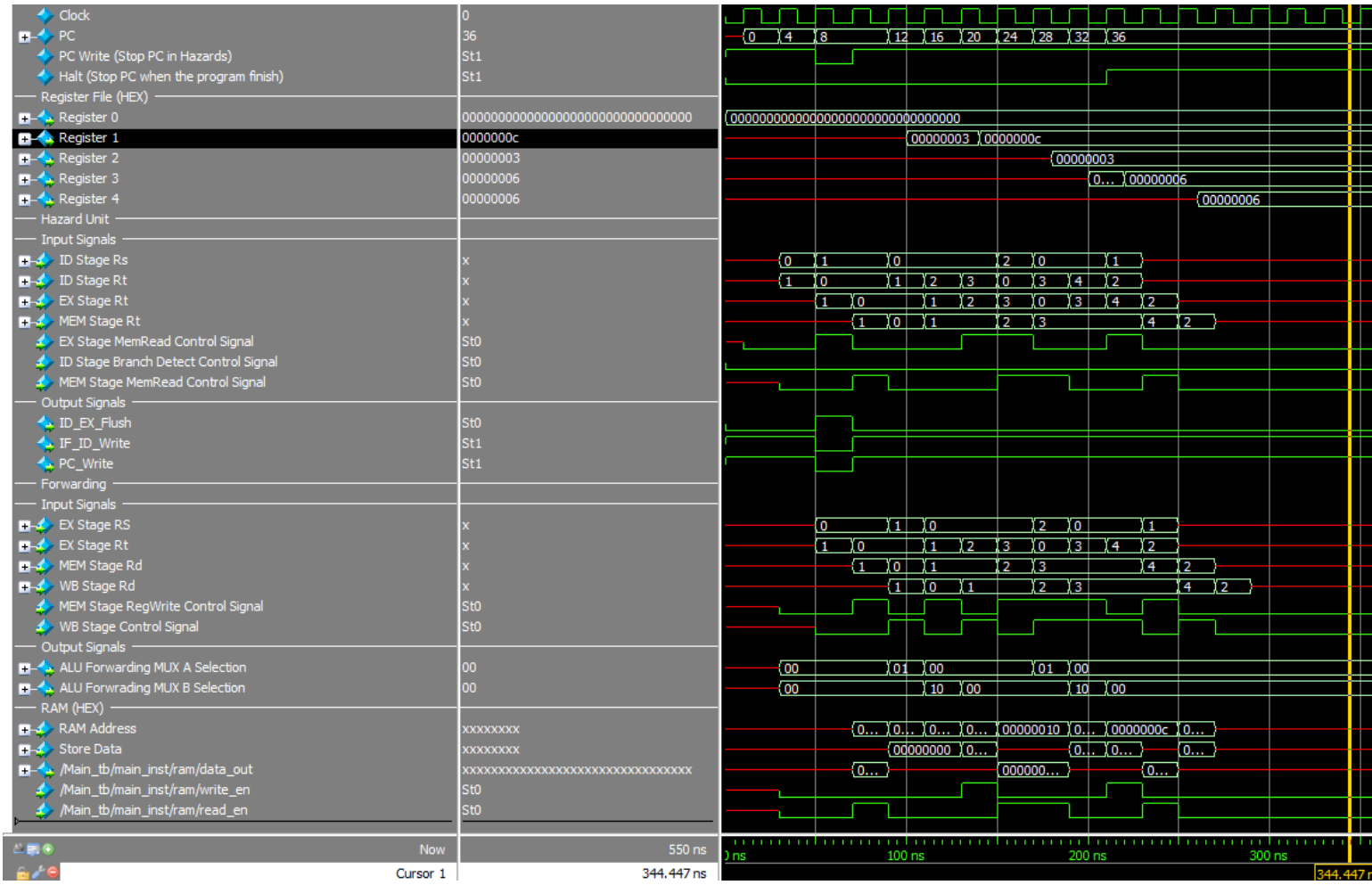

GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.17.3 Second Instruction SLL R1, R1, 2 MEM Stage Forwarding Catch




## 1.17.4 CPU Stage After Execution

GitHub link → https://github.com/OmarAl-Saleh/MIPS




GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.18 Test case 3

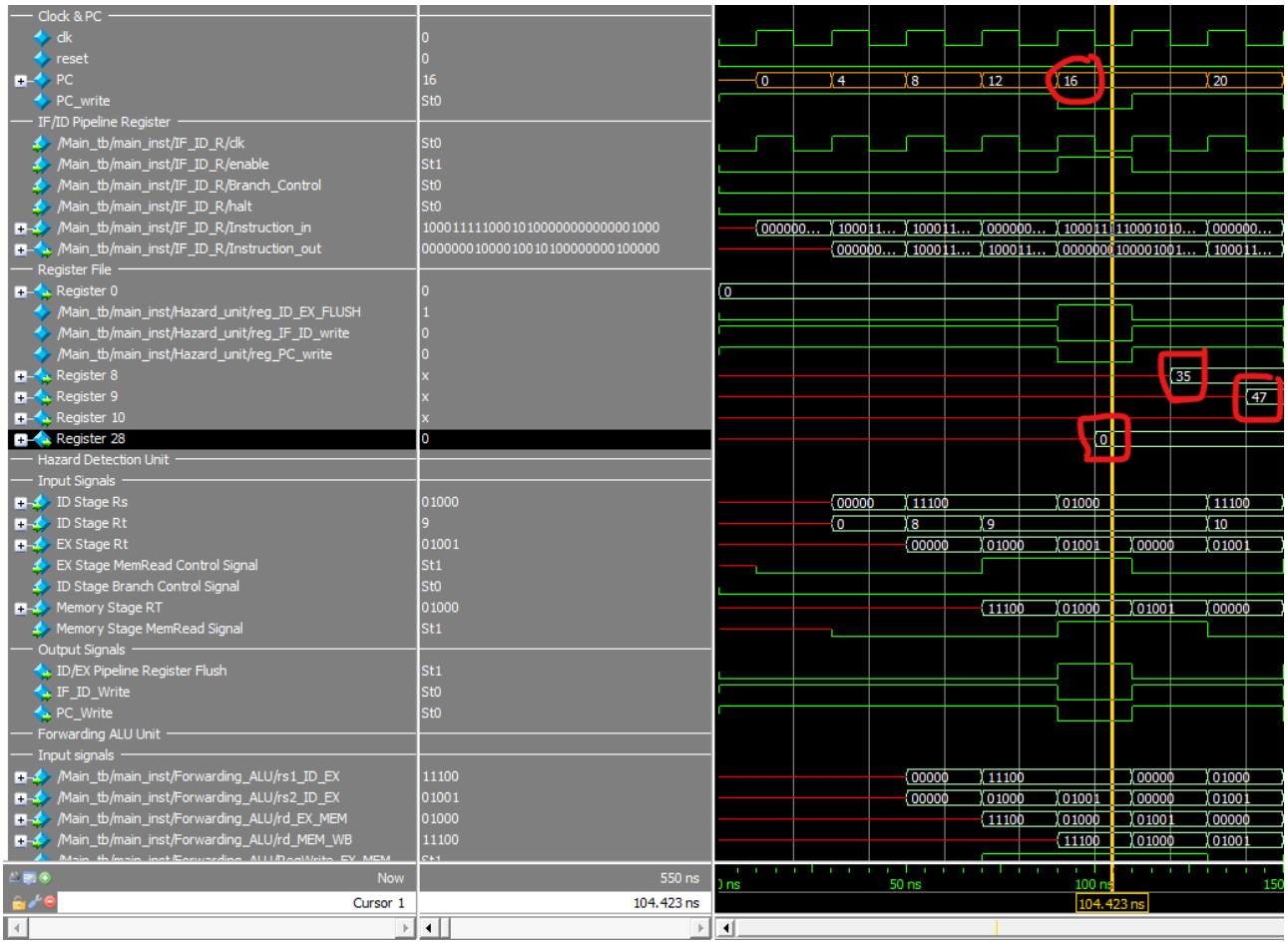
### 1.18.1 Instruction memory and data memory

```
//testcase 3

inst_mem[0] = 32'b00000000000000001110000000000000; //ADD R28,R0,R0 (R28=0)
inst_mem[1] = 32'b10001111100010000000000000000000; //LW R8, 0(R28)
inst_mem[2] = 32'b10001111100010010000000000000100; //LW R9, 4(R28)
inst_mem[3] = 32'b00000001000010010100000000100000; //ADD R8, R8, R9 Hazard & Forwarding
inst_mem[4] = 32'b10001111100010100000000000001000; //LW R10, 8(R28)
inst_mem[5] = 32'b00000001010010100101000000100000; //ADD R10, R10, R10 Hazard & Forwarding
inst_mem[6] = 32'b00000001000010100100000000100010; //SUB R8, R8, R10
inst_mem[7] = 32'b00100001000010000000000000000001; //ADDI R8, R8, 1 Forwarding
inst_mem[8] = 32'b00000000000001000010000000100010; //SUB R8, R0, R8 Forwarding


//-------------------------------------------------------------


/* LAST ADDRESS */ inst_mem[9] = 32'b10110100001000100001100000100000; //Halt
// every program should end with halt signal
```

```
//testcase 3

mem[0] = 32'h00000023; // 0x23 (35)
mem[1] = 32'h0000002F; // 0x2F (47)
mem[2] = 32'h0000001A; // 0x1A (26)


/*
```

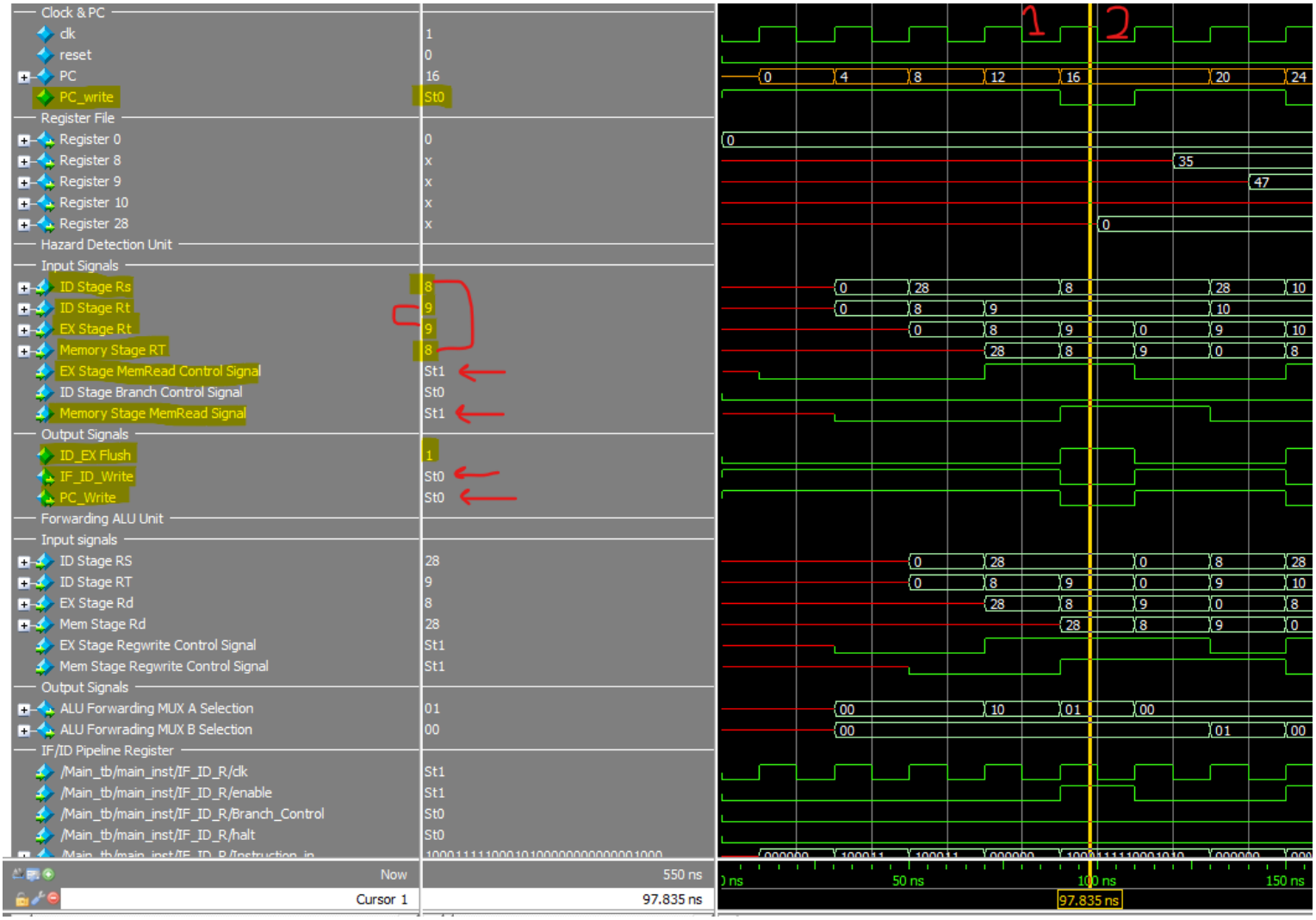
GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.18.2 First Three Instruction Execution.



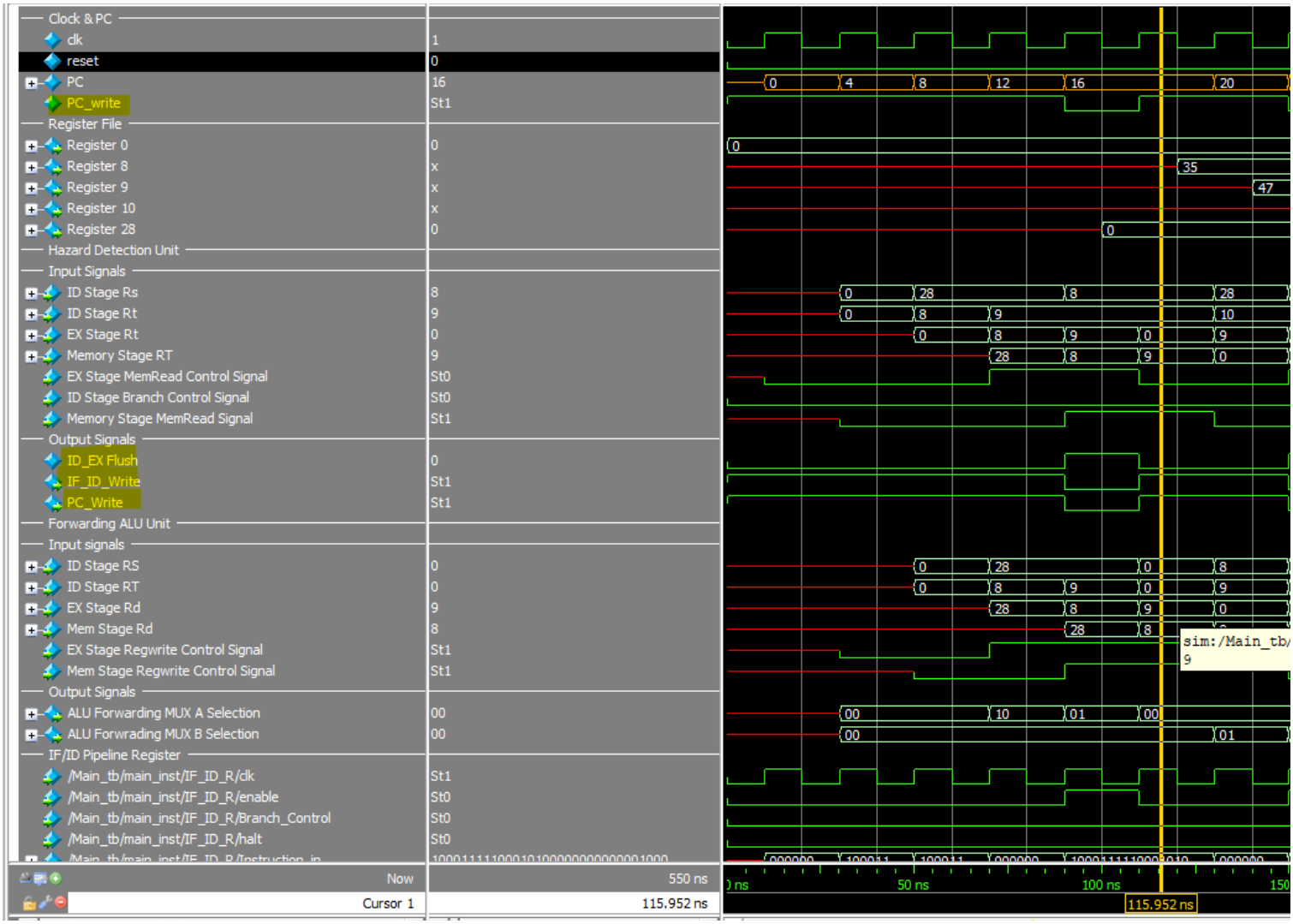

GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.18.3 Fourth Instruction ADD R8, R8, R9 Load_Use_Hazard Catch



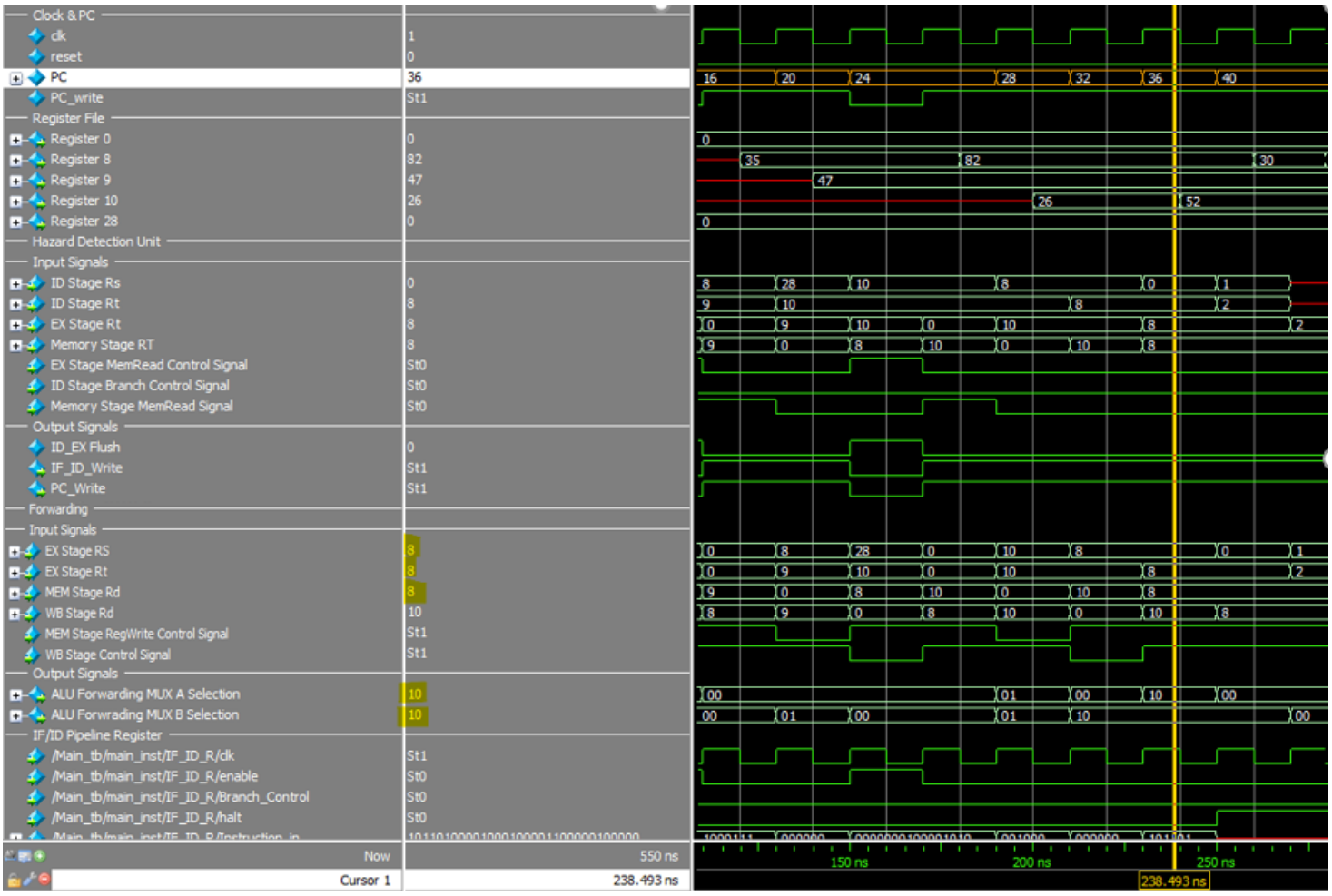

GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.18.4 Fourth Instruction Hazard End



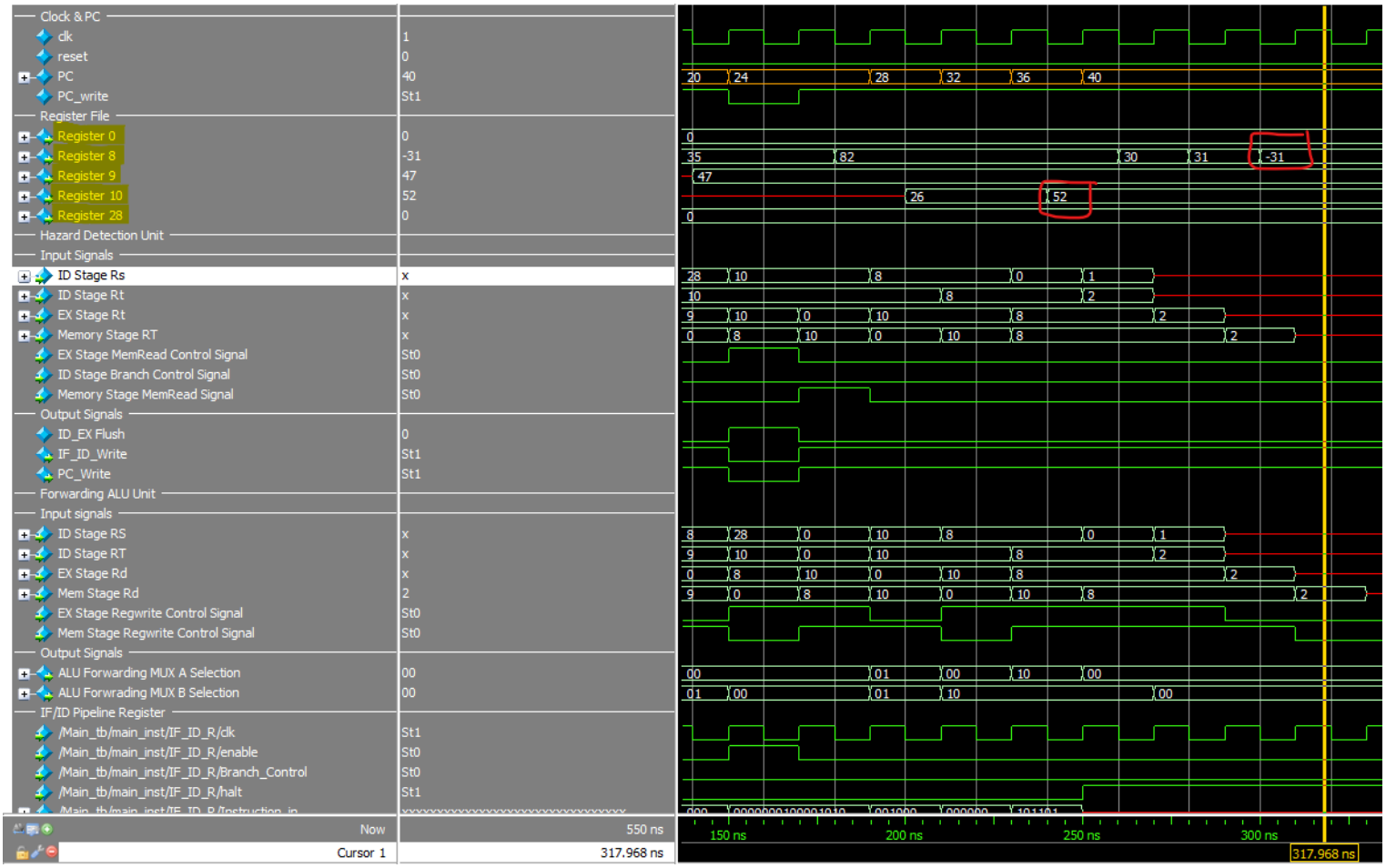

GitHub link → https://github.com/OmarAl-Saleh/MIPS

1.18.5 eighth instruction ADDI R8, R8, 1 Forwarding EX Stage ALU Forwarding.




GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.18.6 the CPU after execution




GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.19 Test case 4

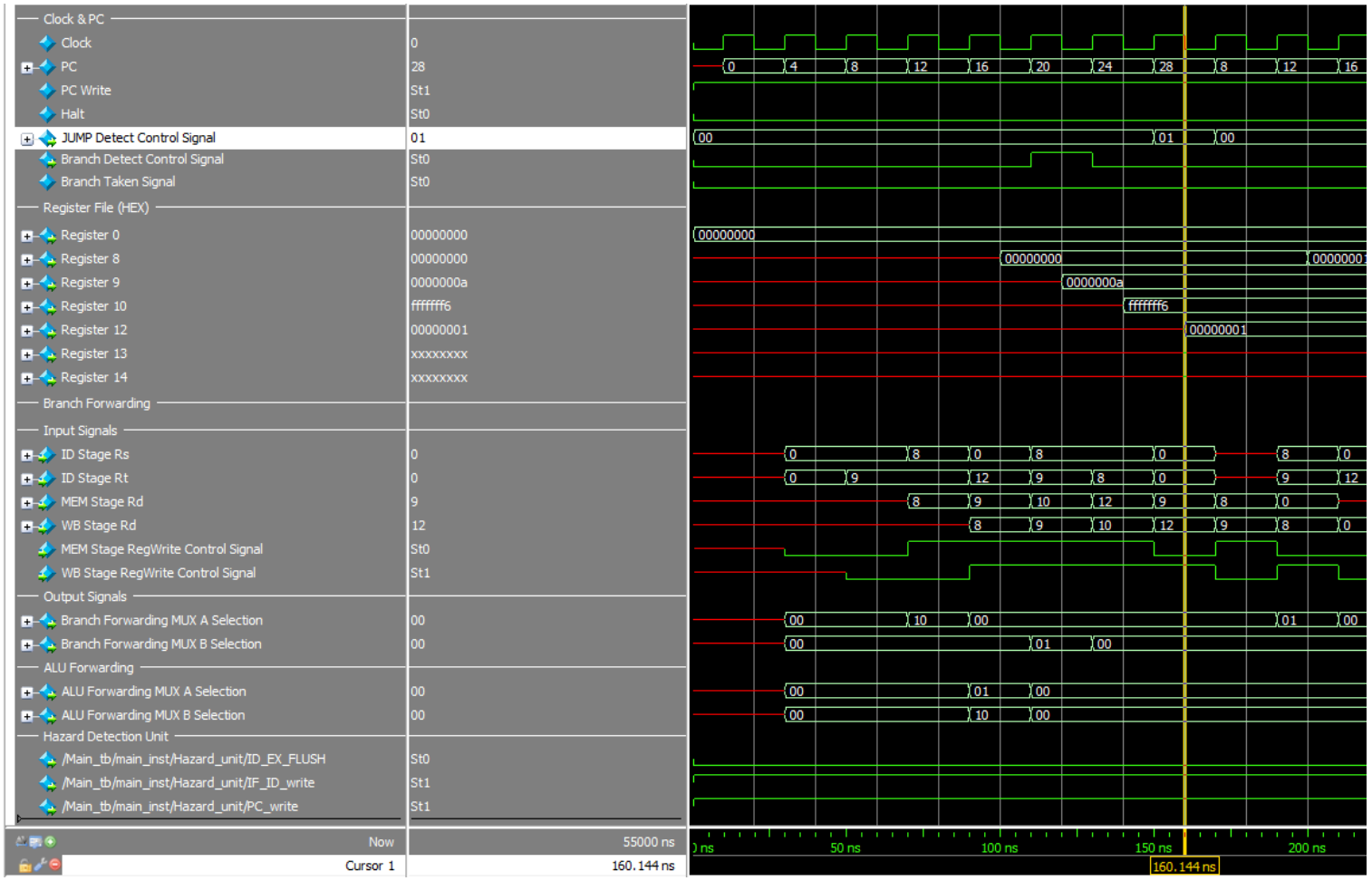
### 1.19.1 Instruction memory and data memory

```
//testcase 4


/*Address 0 */  inst_mem[0] = 32'b00000000000000000100000000100000; //ADD R8, R0, R0
/*Address 4 */  inst_mem[1] = 32'b00100000000010010000000000001010; //ADDI R9, R9, 10
/*Address 8 */  inst_mem[2] = 32'b00000001000010010101000000100010; //SUB R10, R8, R9 //Loop (
/*Address 12 */ inst_mem[3] = 32'b00100000000011000000000000000001; //ADDI R12, R0, 1
/*Address 16 */ inst_mem[4] = 32'b00011001000010010000000000000010; //BGT R8, R9, DONE
/*Address 20 */ inst_mem[5] = 32'b00100001000010000000000000000001; //ADDI R8, R8, 1
/*Address 24 */ inst_mem[6] = 32'b00001000000000000000000000000010; //JUMP LOOP
/*Address 28 */ inst_mem[7] = 32'b00000001001000000110100000100000; //ADD R13, R9, R0 //DONE
/*Address 32 */ inst_mem[8] = 32'b00100000000011100000000000011011; //ADDI R14, R0, 1B(27)
/*Address 36 */ inst_mem[9] = 32'b00110001110011100000000000010111; //ANDI R14, R14, 17(23) Fo

/*-------------------------------------------------------------
```

```
//testcase 4 & 5 & 6
//no need for RAM
```

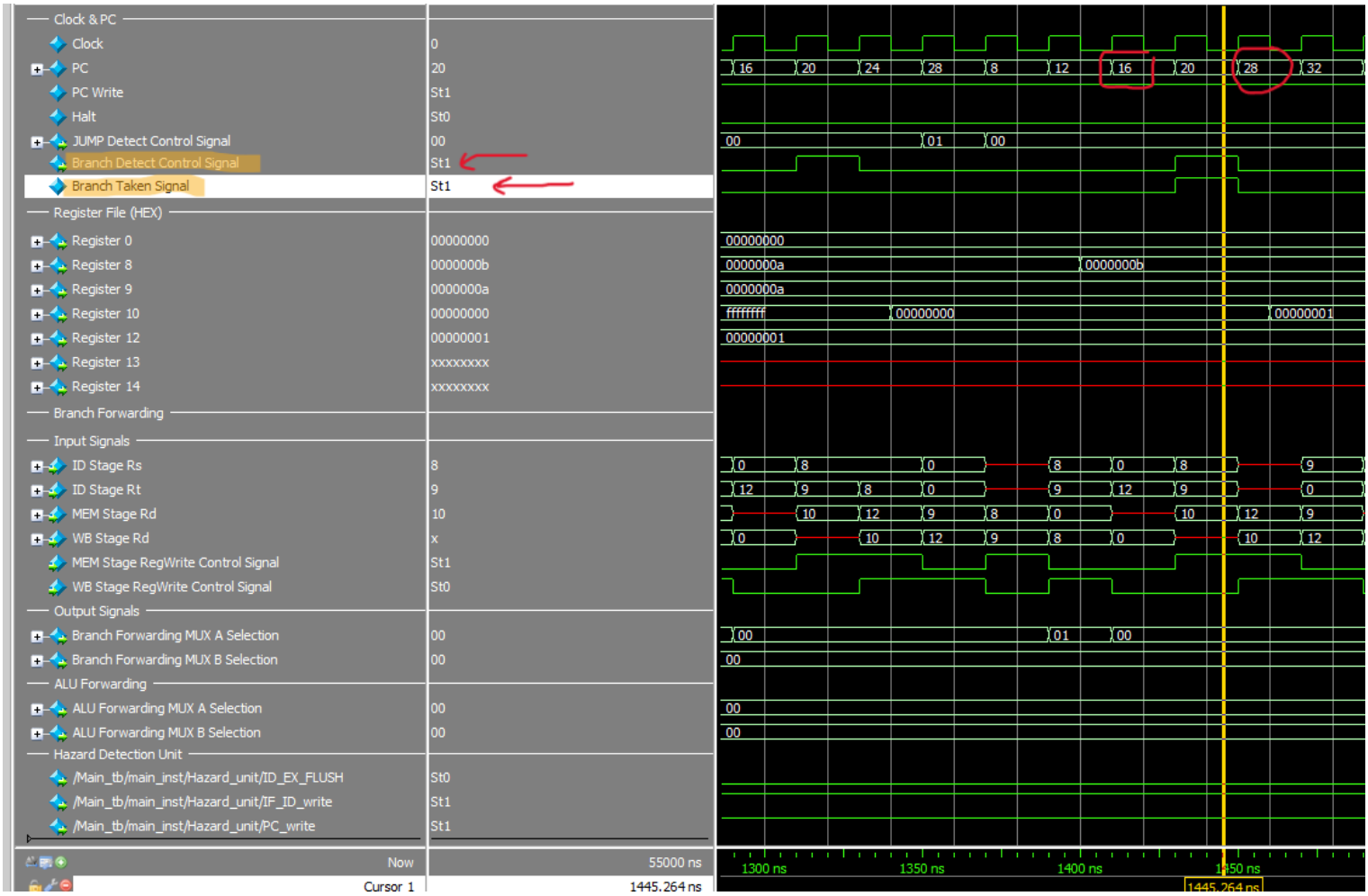
GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.19.2 First Jump Instruction Branch not Taken



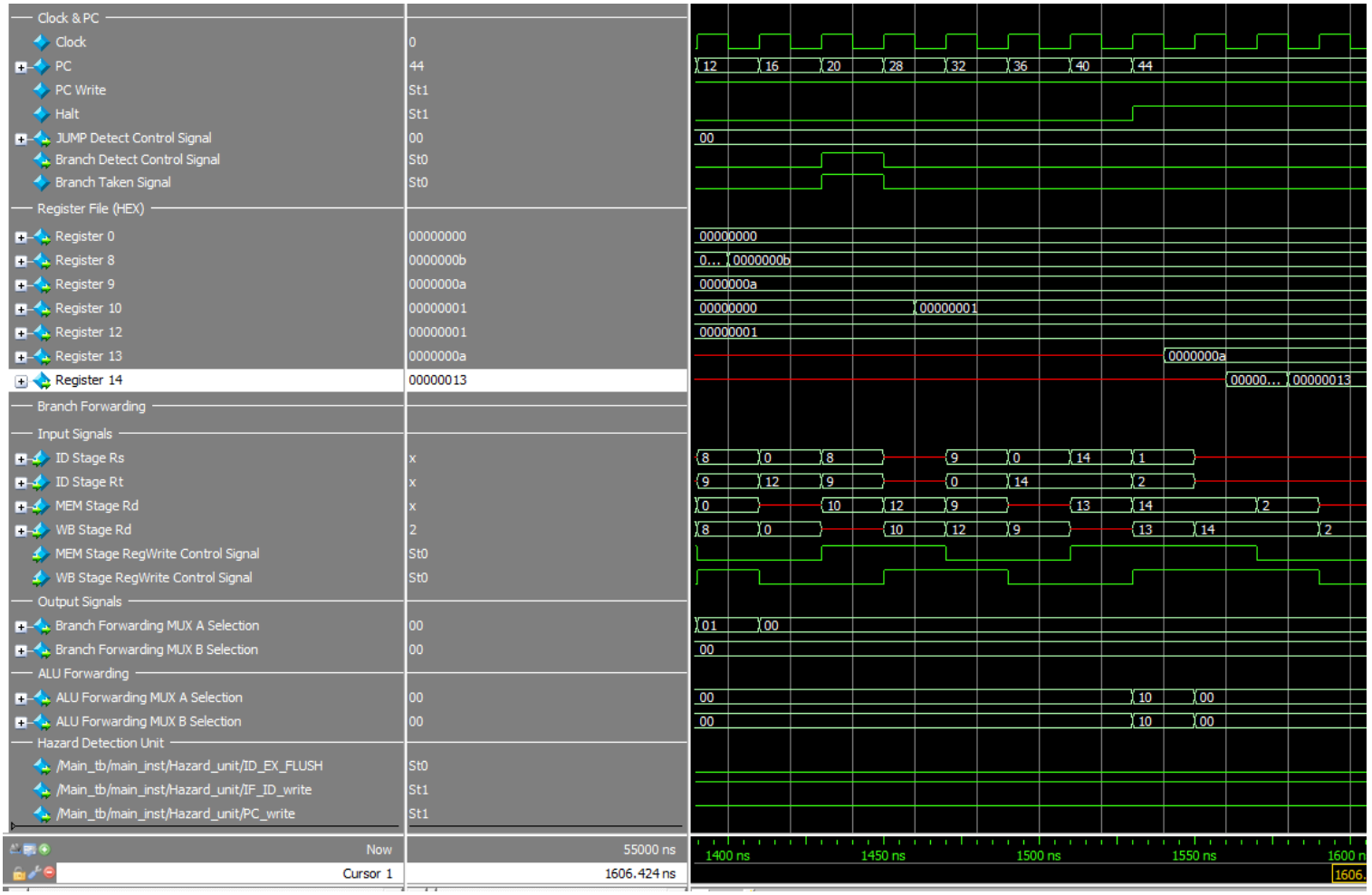

GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.19.3 Branch Taken last Branch Instruction




GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.19.4 CPU Status after execution




GitHub link → https://github.com/OmarAl-Saleh/MIPS

# 1.20 Test case 5

## 1.20.1 Instruction memory and data memory

**No need for RAM**

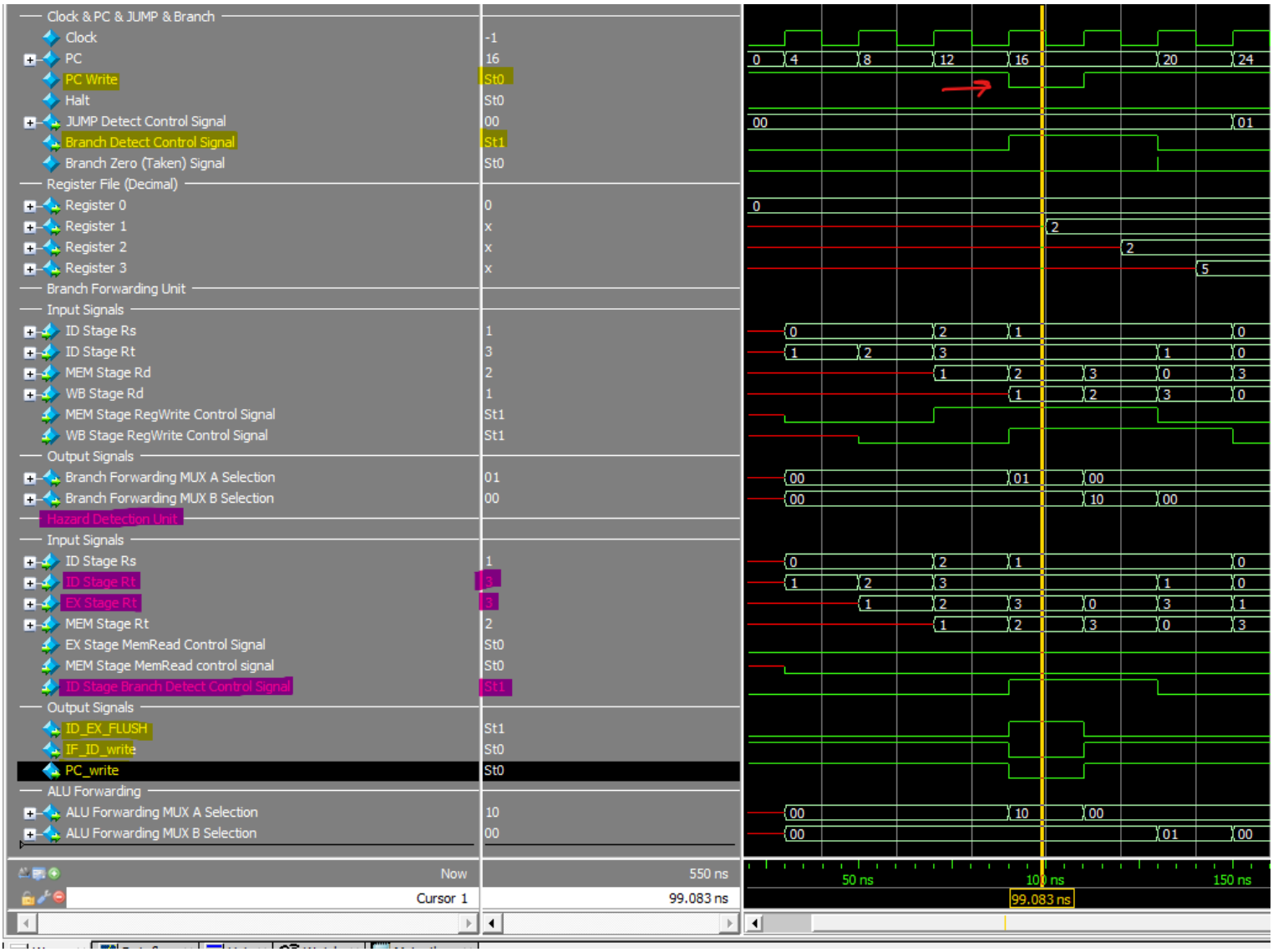
```
//testcase 5

//a=2
/*Address 0 */  inst_mem[0] = 32'b00100000000000010000000000000010; //ADDI R1, R0, 2 (a)
/*Address 4 */  inst_mem[1] = 32'b00100000000000100000000000000010; //ADDI R2, R0, 2 (b)
/*Address 8 */  inst_mem[2] = 32'b00100000010000110000000000000011; //ADDI R3, R2, 3 (b+3)
/*Address 12 */ inst_mem[3] = 32'b00100100001000110000000000000010; //BGE  R1, R3, THEN
/*Address 16 */ inst_mem[4] = 32'b00100000001000010000000000000001; //ADDI R1, R1, 1
/*Address 20 */ inst_mem[5] = 32'b00001000000000000000000000000111; //JUMP END
/*Address 24 */ inst_mem[6] = 32'b00100000001000010000000000000010; //ADDI R1, R1, 2 //THEN
/*Address 28 */ inst_mem[7] = 32'b00000000010000010001000000100000; //ADD R2, R2, R1 //END

//a=6

/*Address 0 */  inst_mem[0] = 32'b00100000000000010000000000000110; //ADDI R1, R0, 6 (a)
/*Address 4 */  inst_mem[1] = 32'b00100000000000100000000000000010; //ADDI R2, R0, 2 (b)
/*Address 8 */  inst_mem[2] = 32'b00100000010000110000000000000011; //ADDI R3, R2, 3 (b+3)
/*Address 12 */ inst_mem[3] = 32'b00100100001000110000000000000010; //BGE  R1, R3, THEN
/*Address 16 */ inst_mem[4] = 32'b00100000001000010000000000000001; //ADDI R1, R1, 1
/*Address 20 */ inst_mem[5] = 32'b00001000000000000000000000000111; //JUMP END
/*Address 24 */ inst_mem[6] = 32'b00100000001000010000000000000010; //ADDI R1, R1, 2 //THEN
/*Address 28 */ inst_mem[7] = 32'b00000000010000010001000000100000; //ADD R2, R2, R1 //END


//-------------------------------------------------------------
```

## 1.20.2 When a =2

```
//testcase 5

//a=2
/*Address 0 */  inst_mem[0] = 32'b00100000000000010000000000000010; //ADDI R1, R0, 2 (a)
/*Address 4 */  inst_mem[1] = 32'b00100000000000100000000000000010; //ADDI R2, R0, 2 (b)
/*Address 8 */  inst_mem[2] = 32'b00100000010000110000000000000011; //ADDI R3, R2, 3 (b+3) ALU Forwarding
/*Address 12 */ inst_mem[3] = 32'b00100100001000110000000000000010; //BGE  R1, R3, THEN  (Hazard Unit + Branch Forwarding)
/*Address 16 */ inst_mem[4] = 32'b00100000001000010000000000000001; //ADDI R1, R1, 1
/*Address 20 */ inst_mem[5] = 32'b00001000000000000000000000000111; //JUMP END
/*Address 24 */ inst_mem[6] = 32'b00100000001000010000000000000010; //ADDI R1, R1, 2 //THEN
/*Address 28 */ inst_mem[7] = 32'b00000000010000010001000000100000; //ADD R2, R2, R1 //END


//-------------------------------------------------------------

/* LAST ADDRESS */ inst_mem[8] = 32'b10110100001000100001100000100000; //Halt
// every program should end with halt signal
```

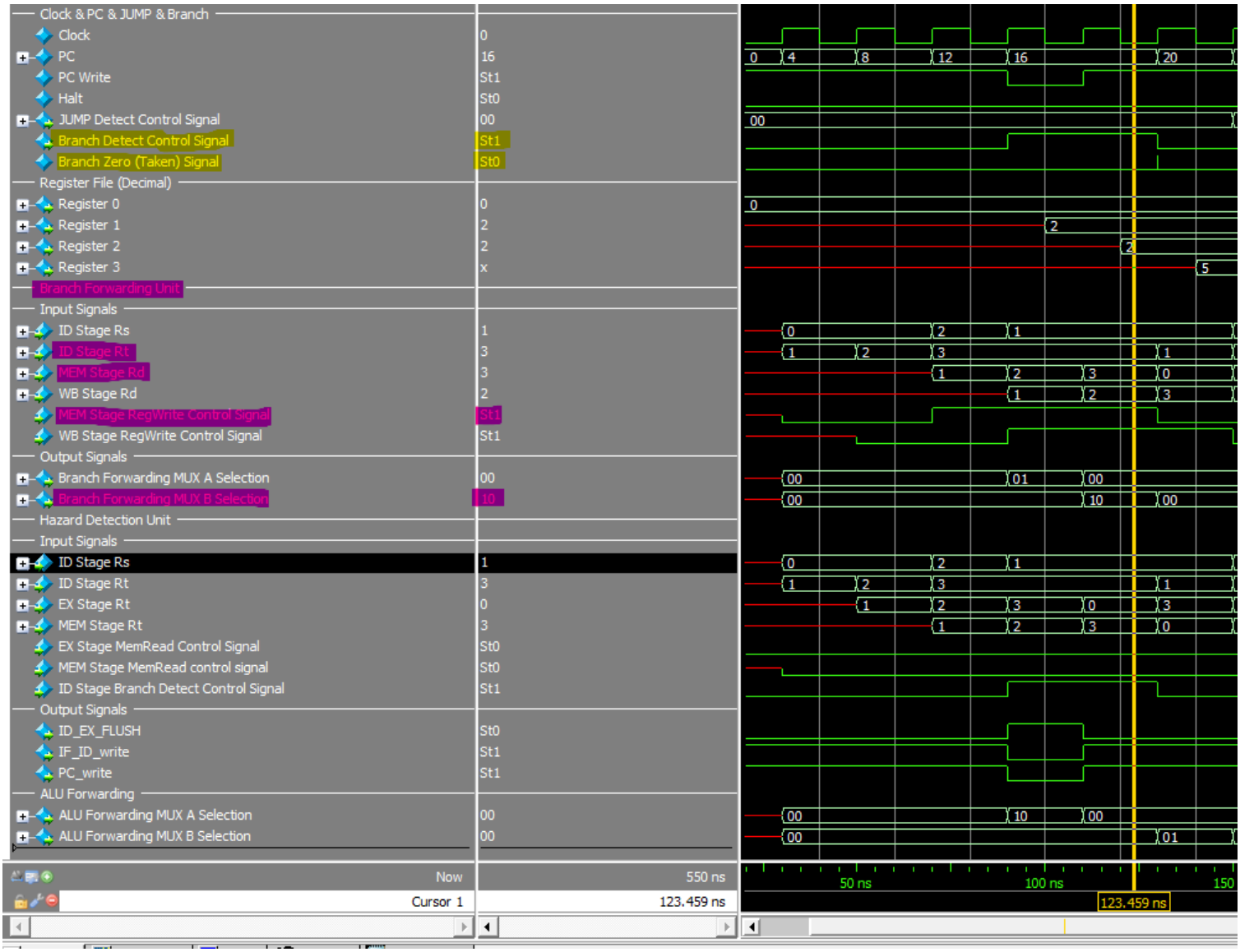
GitHub link → https://github.com/OmarAl-Saleh/MIPS

### *1.20.2.1 Fourth Instruction BGE R1, R3, THEN Hazard Detect*



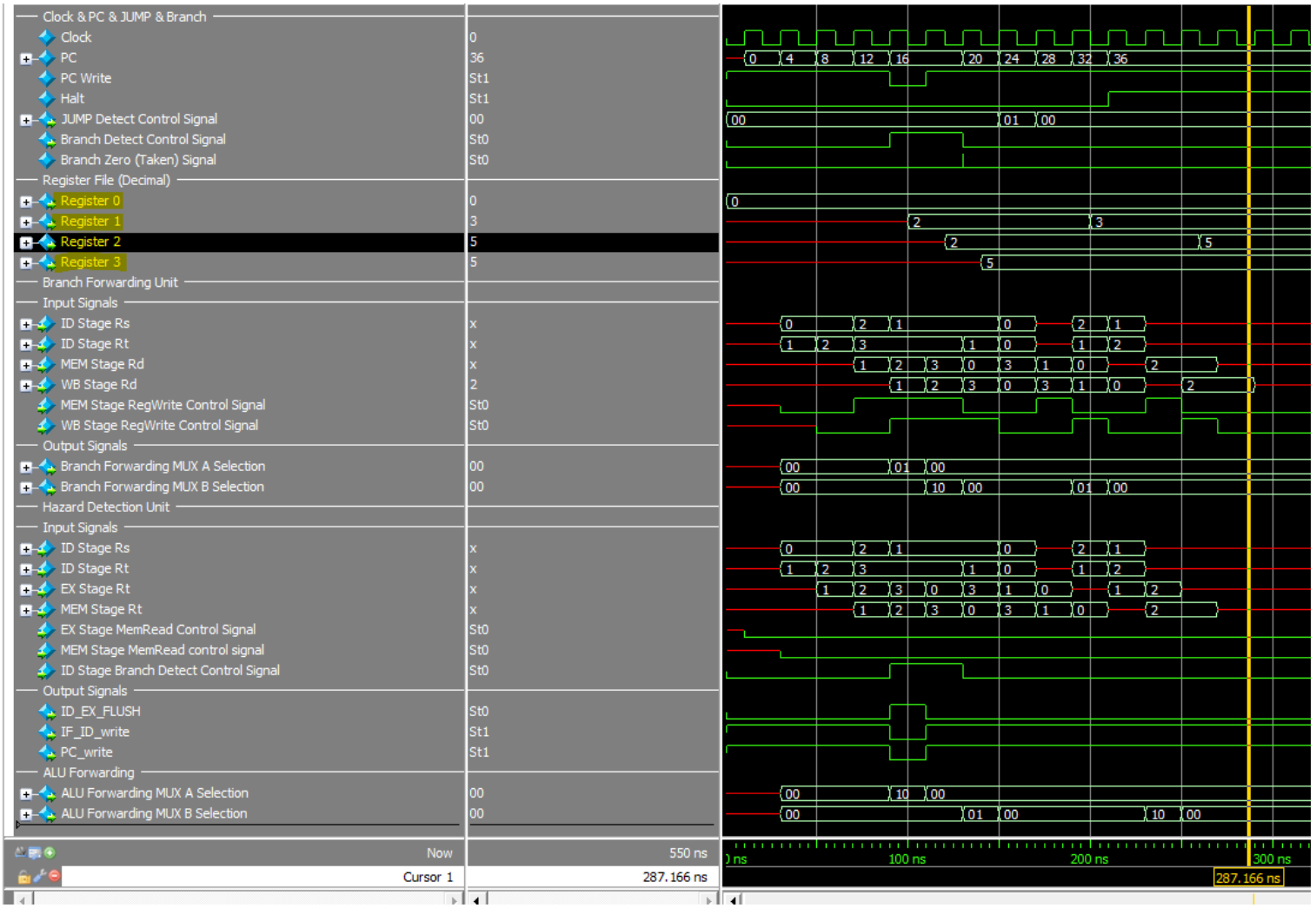

GitHub link → https://github.com/OmarAl-Saleh/MIPS

### *1.20.2.2 fourth Instruction BGE R1, R3, THEN Branch Not Taken with Branch Forwarding*




GitHub link → https://github.com/OmarAl-Saleh/MIPS

### *1.20.2.3 CPU Status After execution*



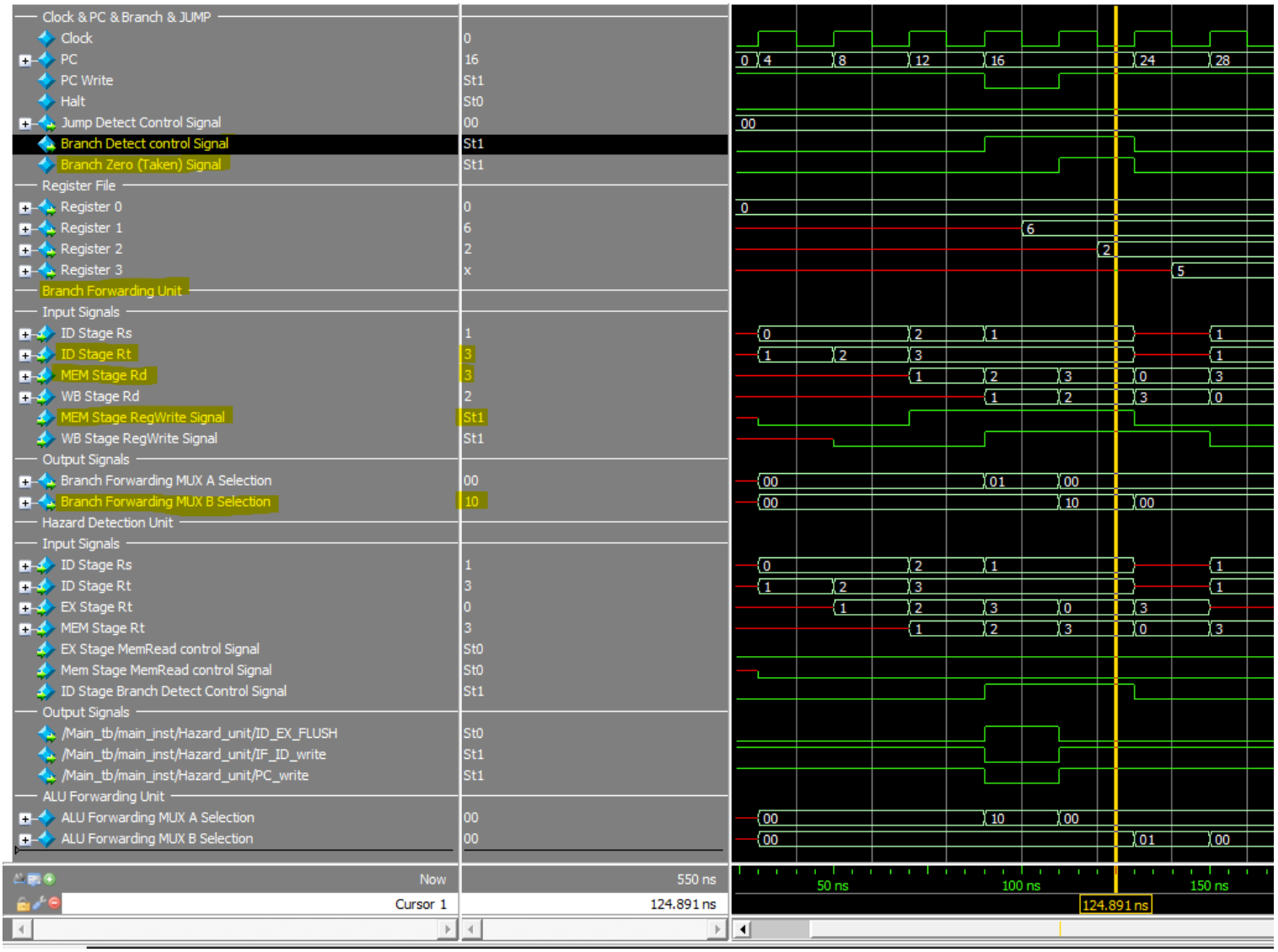

GitHub link → https://github.com/OmarAl-Saleh/MIPS

1.20.3When a =6

```
   //testcase 5

   //a=6

   /*Address 0 */  inst_mem[0] = 32'b00100000000000010000000000000110; //ADDI R1, R0, 6 (a)
   /*Address 4 */  inst_mem[1] = 32'b00100000000000100000000000000010; //ADDI R2, R0, 2 (b)
   /*Address 8 */  inst_mem[2] = 32'b00100000010000110000000000000011; //ADDI R3, R2, 3 (b+3) Fo
   /*Address 12 */ inst_mem[3] = 32'b00100100001000110000000000000010; //BGE  R1, R3, THEN (Haza
   /*Address 16 */ inst_mem[4] = 32'b00100000001000010000000000000001; //ADDI R1, R1, 1
   /*Address 20 */ inst_mem[5] = 32'b00001000000000000000000000000111; //JUMP END
   /*Address 24 */ inst_mem[6] = 32'b00100000001000010000000000000010; //ADDI R1, R1, 2 //THEN
   /*Address 28 */ inst_mem[7] = 32'b00000000010000010001000000100000; //ADD R2, R2, R1 //END

   |

//-----------------------------------------------------------------


/* LAST ADDRESS */ inst_mem[8] = 32'b10110100001000100001100000100000; //Halt
// every program should end with halt signal
```

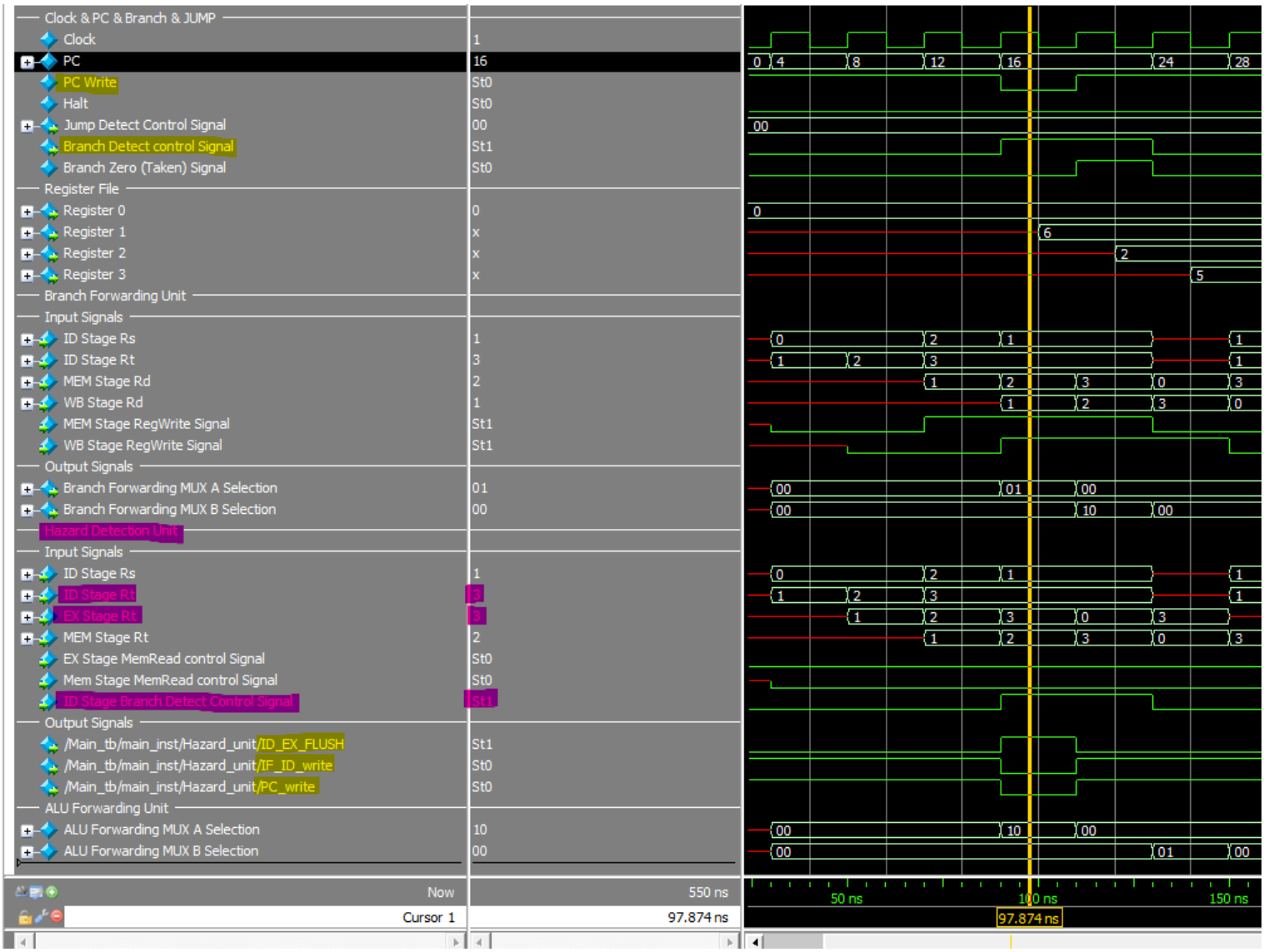
GitHub link → https://github.com/OmarAl-Saleh/MIPS

### *1.20.3.1 Fourth Instruction BGE R1,R3,Then Branch Taken Branch Forwarding Catch*



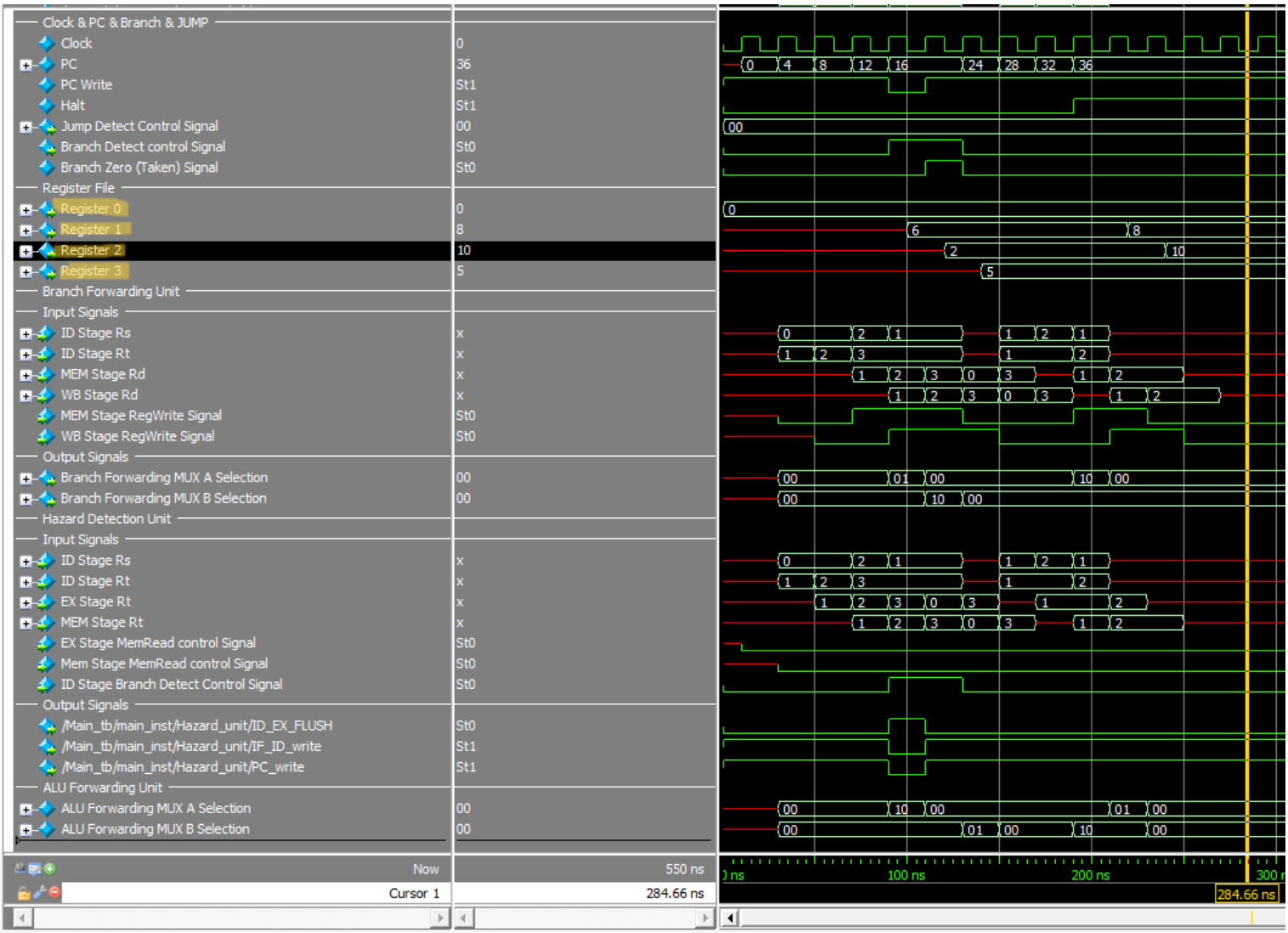

GitHub link → https://github.com/OmarAl-Saleh/MIPS

### *1.20.3.2 Fourth Instruction BGE R1, R3,THEN Hazard Catch*




GitHub link → https://github.com/OmarAl-Saleh/MIPS

### *1.20.3.3 CPU Status After execution*




GitHub link → https://github.com/OmarAl-Saleh/MIPS

# 1.21 Test case 6

## 1.21.1 Instruction memory

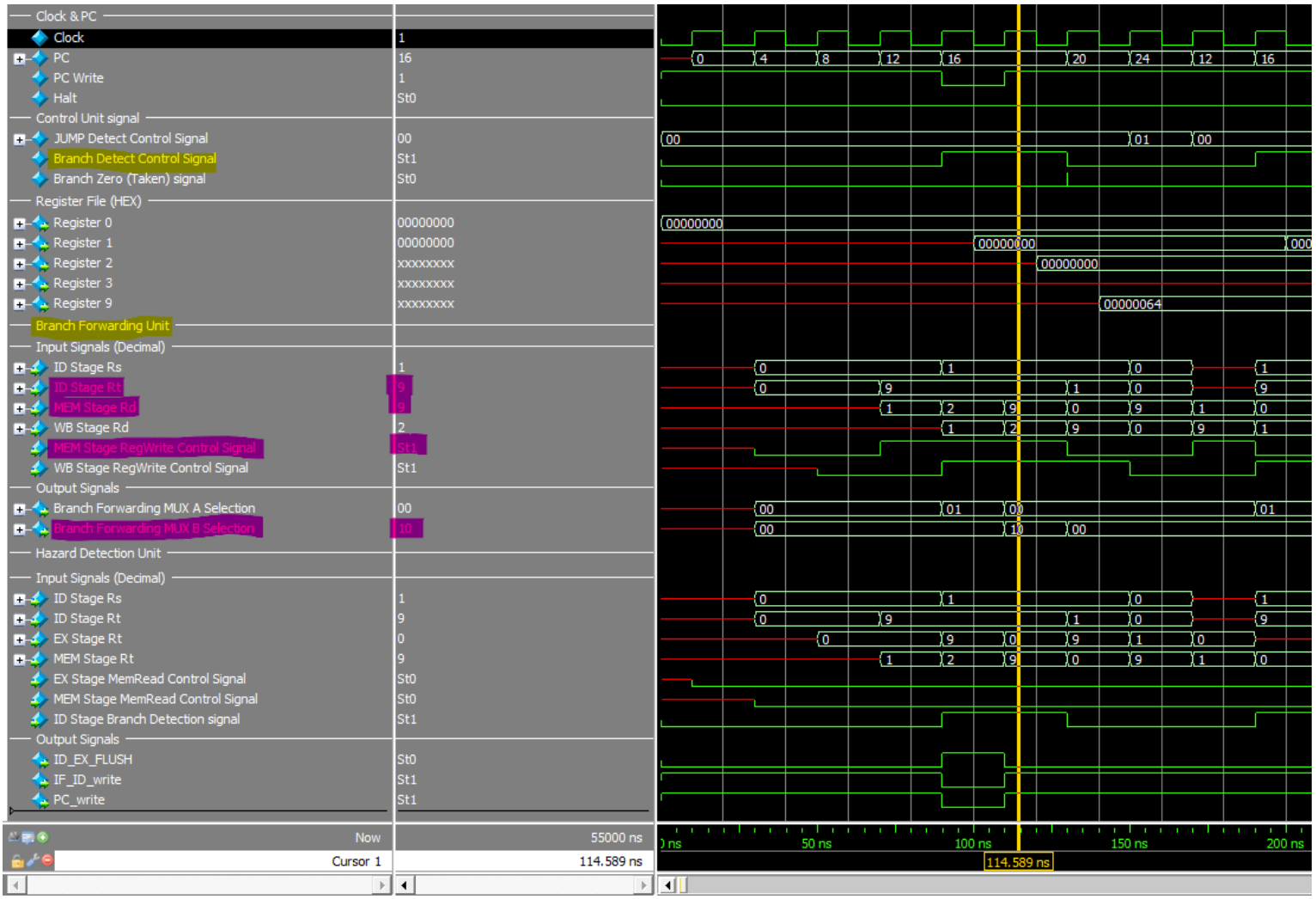
**No need for RAM**

```
   //testcase 6

   /*Address 0 */  inst_mem[0] = 32'b00000000000000000000100000100000; //ADD R1, R0, R0
   /*Address 4 */  inst_mem[1] = 32'b00000000000000000001000000100000; //ADD R2, R0, R0
   /*Address 8 */  inst_mem[2] = 32'b00100000000010010000000001100100; //ADDI R9, R0, 100
   /*Address 12 */ inst_mem[3] = 32'b00010000001010010000000000000010; //BEQ R1, R9, EXIT //START(Branch Hazard )(Branch Forwarding)
   /*Address 16 */ inst_mem[4] = 32'b00100000001000010000000000000001; //ADDI R1, R1, 1
   /*Address 20 */ inst_mem[5] = 32'b00001000000000000000000000000011; //JUMP START
   /*Address 24 */ inst_mem[6] = 32'b00000000001000100001100000100000; //ADD R3, R1, R2 //EXIT

//-------------------------------------------------------------


/* LAST ADDRESS */ inst_mem[7] = 32'b10110100001000100001100000100000; //Halt
// every program should end with halt signal
```

---

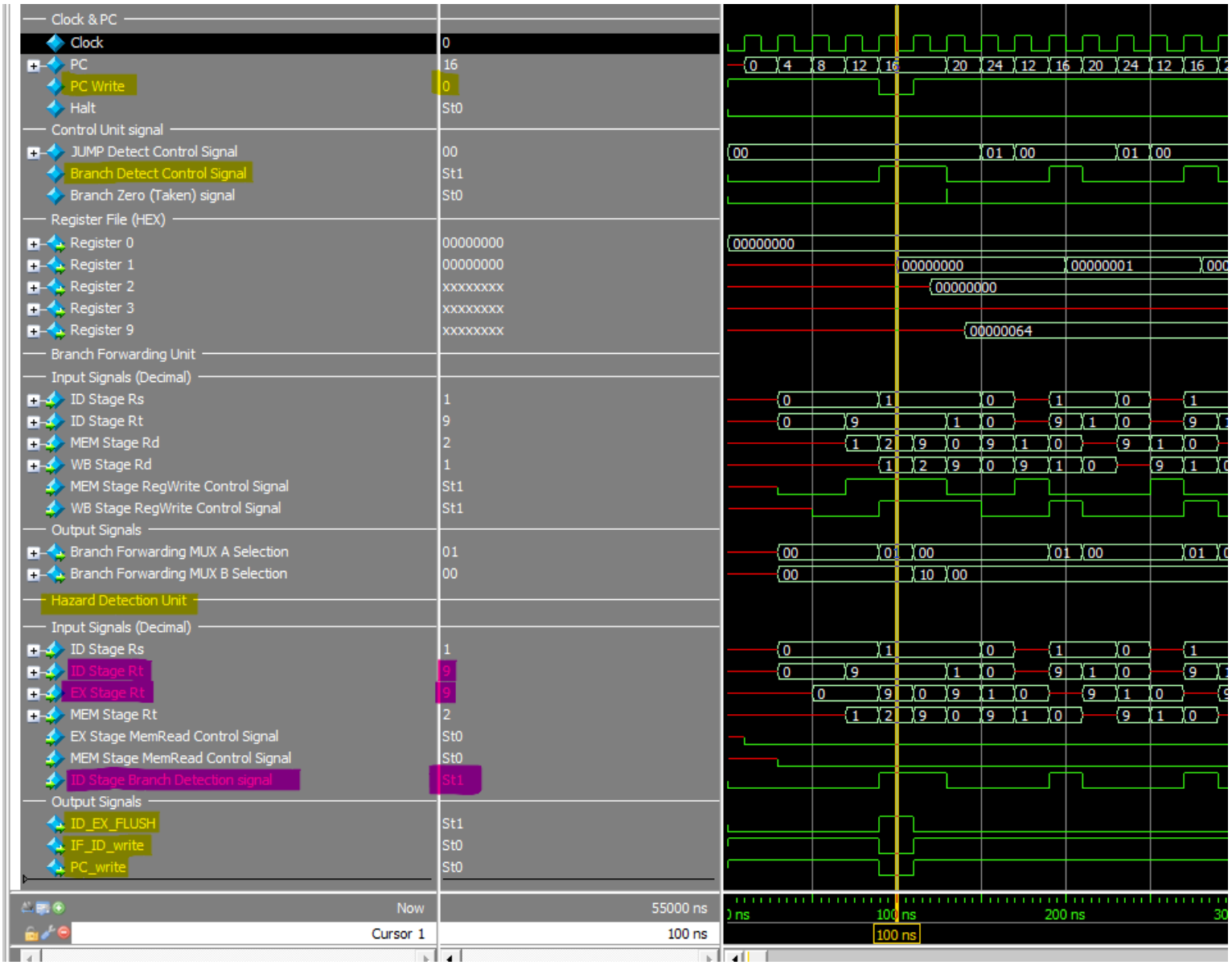
## 1.21.2 First Branch BEQ R1, R9, EXIT Instruction Catch Branch Forwarding

GitHub link → https://github.com/OmarAl-Saleh/MIPS



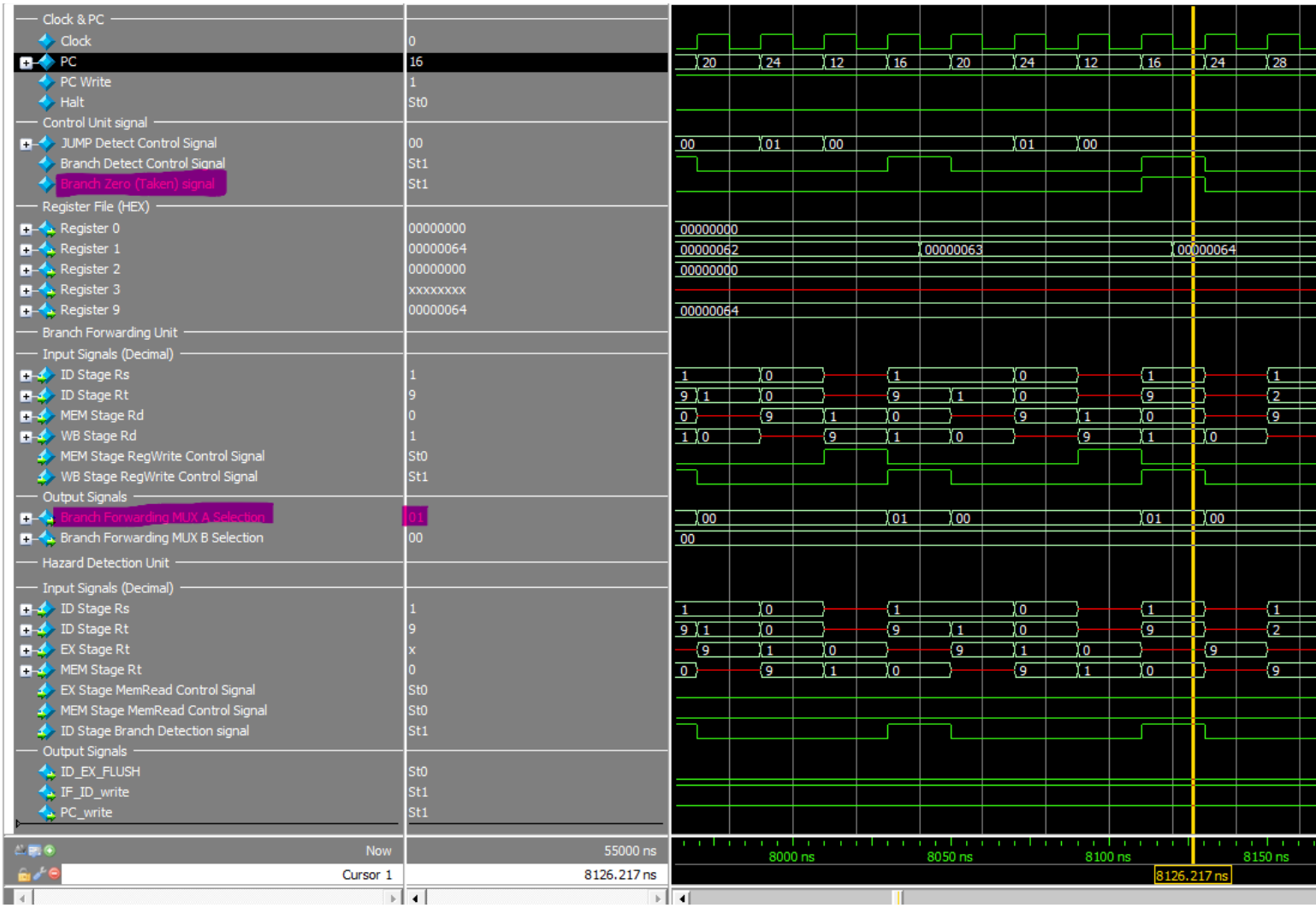

## 1.21.3 First Branch BEQ R1, R9, EXIT Instruction Catch Branch Hazard

GitHub link → https://github.com/OmarAl-Saleh/MIPS



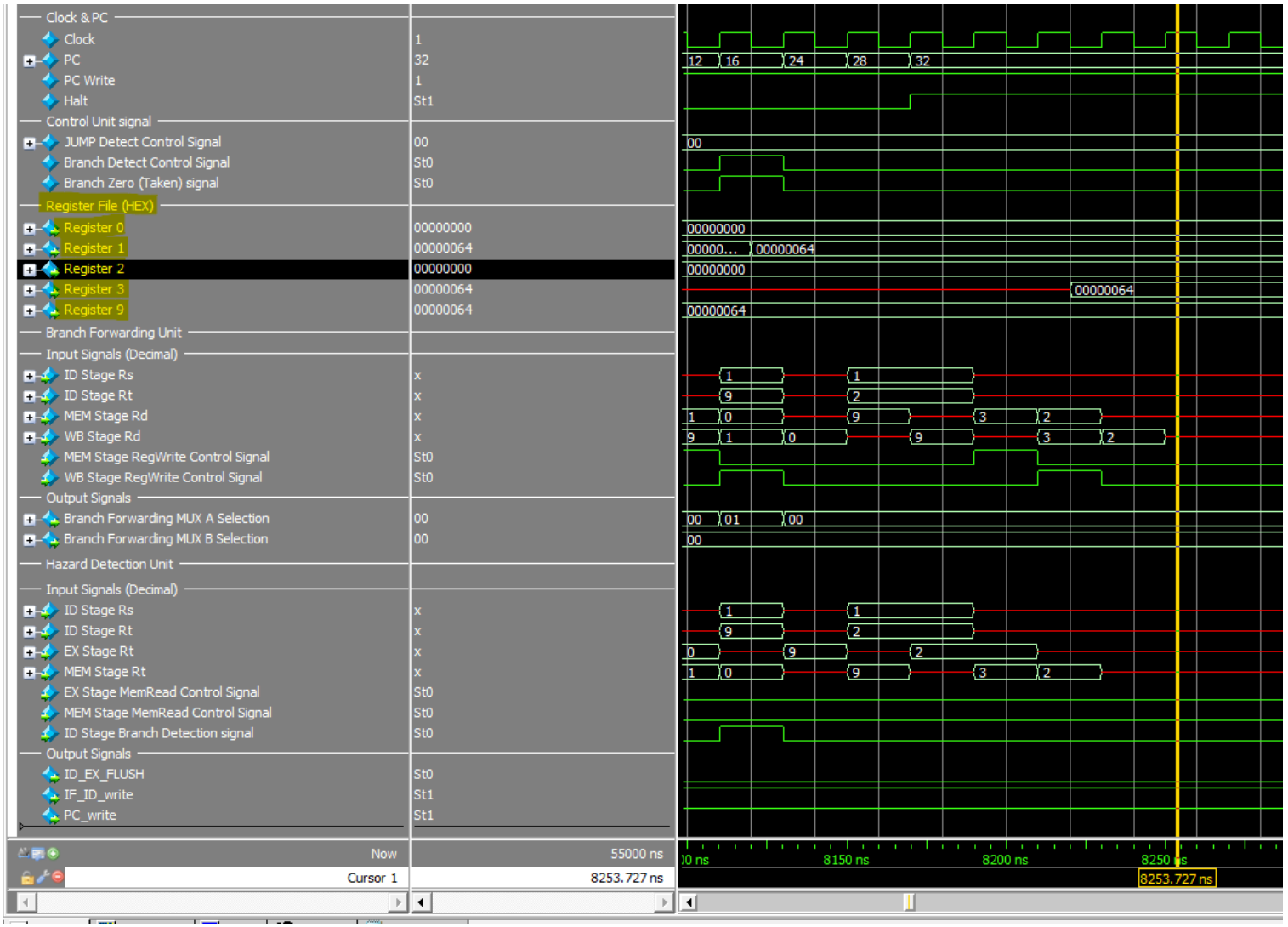

GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.21.4the last Iteration Branch Taken.




GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.21.5 The CPU Status After execution




GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.22 special case JAL & JS

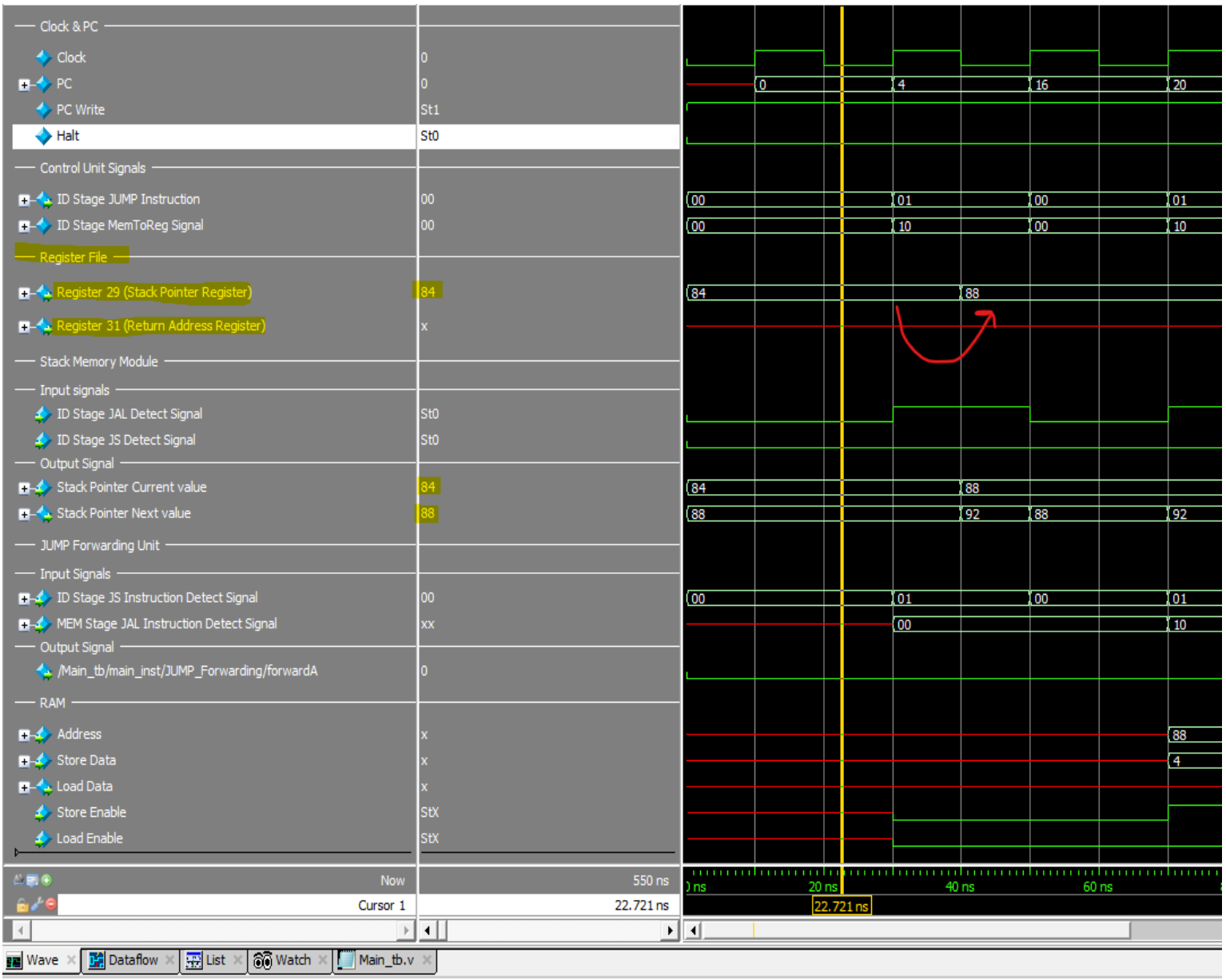
### 1.22.1 Instruction Memory Values.

```
// THIS TEST FOR STACK TESTING USING JAL & JS INSTRUCTIONS (Nested Subroutine) 3 push 3 pull
// the goal is test the stack functionality

/*Address 0 */inst_mem[0] = 32'b00001100000000000000000000000100;//JAL Jump to address 16 and save R31 = 4
/*Address 4 */inst_mem[1] = 32'b10001100000000010000000000000100;//lw reg1=3 (the jump will skip it and return later)
/*Address 8 */inst_mem[2] = 32'b00000000000000000111000000000000; //ADD R28,R0,R0 (R28=0)

/* Address 12 */ inst_mem[3] = 32'b10110100001000100001100000100000; //Halt (stop PC & End the Program)

// the PC final value will be 16

/*Address 16 */inst_mem[4] = 32'b00001100000000000000000000000111;//JAL Jump to address 28 and save R31 = 20


/*Address 20 */inst_mem[5] = 32'b00000011111100101001000000001000;// JS jump to address store in REG 31 so jump to address 4

 /*Address 28 */inst_mem[7] = 32'b00001100000000000000000000001111;//JAL Jump to address 60 and save R31 = 32

/*Address 32 */inst_mem[8] = 32'b00000011111100101001000000001000;// JS jump to address store in REG 31 so jump to address 20


 /*Address 60 */inst_mem[15] = 32'b00000011111100101001000000001000;// JS jump to address store in REG 31 so jump to address 32

   // In this instruction we need JUMP Forwarding Unit between JAL MEM Stage instruction and JS ID Stage Instruction
//-------------------------------------------------------------
```

**Benchmark Description**: In this self-generated benchmark, we will assess the stack functionality and the Jump Forwarding unit. The benchmark involves executing three JAL instructions to reach address 60, followed by three JS instructions to return to address 4, ultimately reaching the Halt instruction located at address 12 to conclude the program.

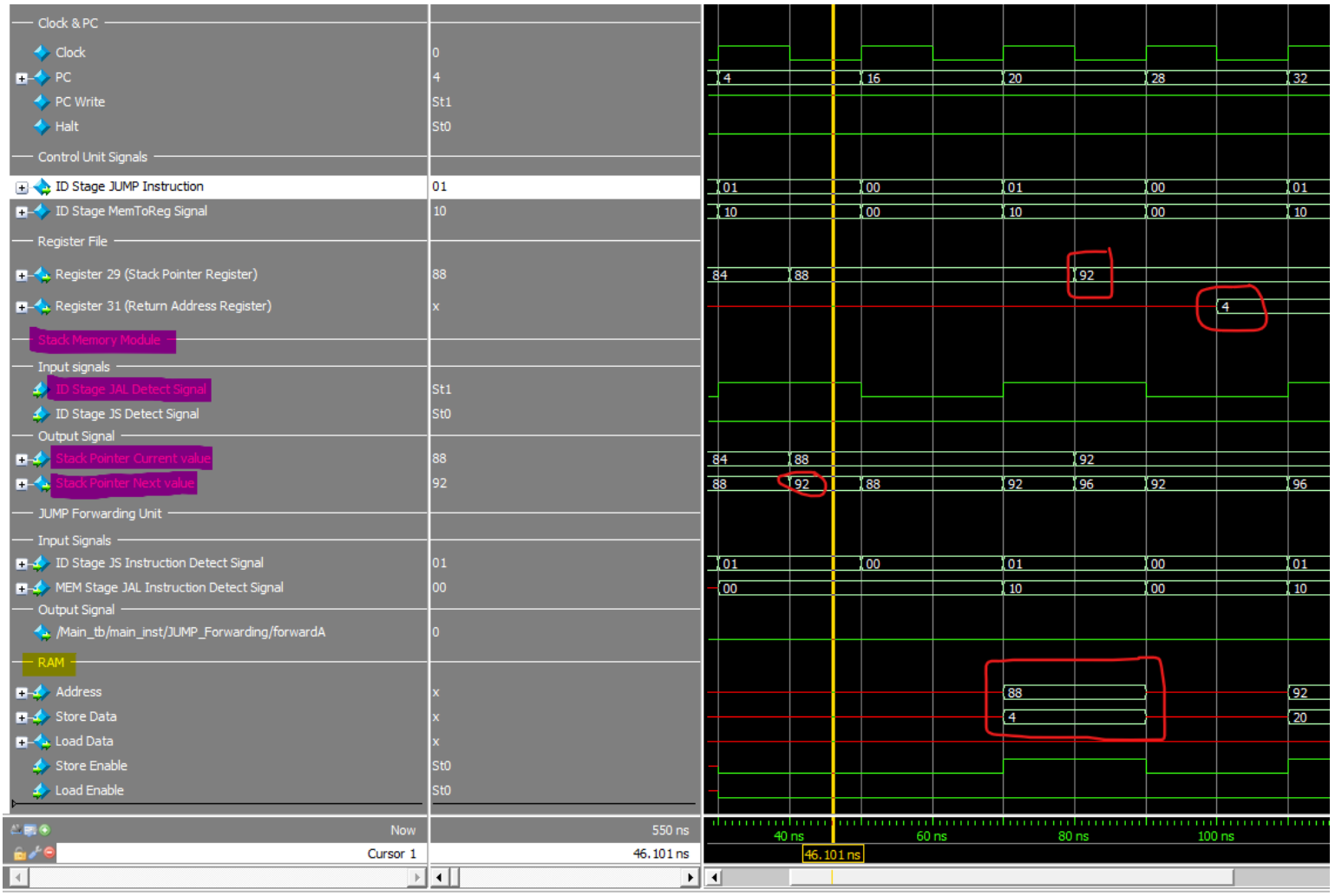
### 1.22.2 Initial Value First Cycle

GitHub link → https://github.com/OmarAl-Saleh/MIPS



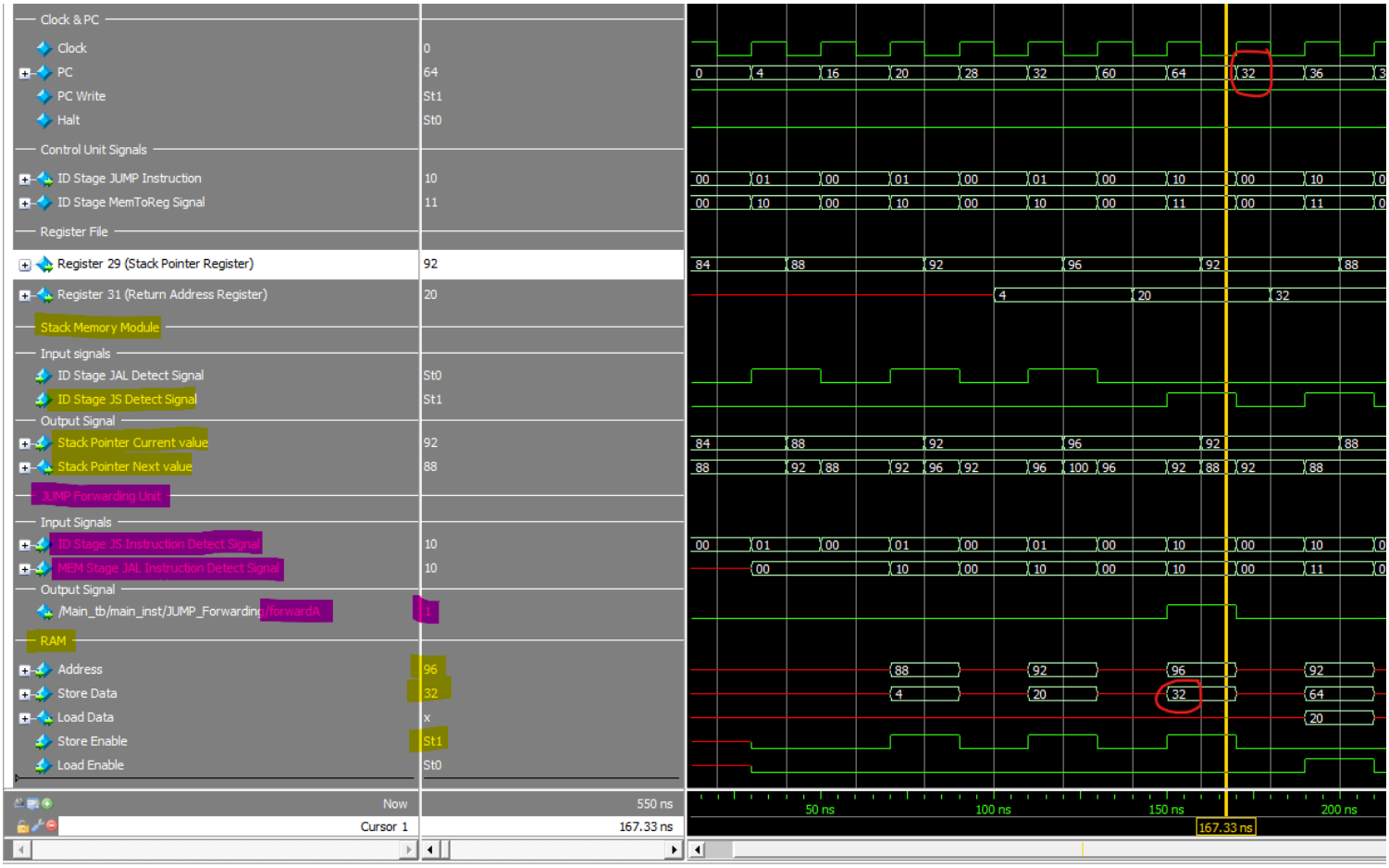

## 1.22.3First JAL Instruction

GitHub link → https://github.com/OmarAl-Saleh/MIPS



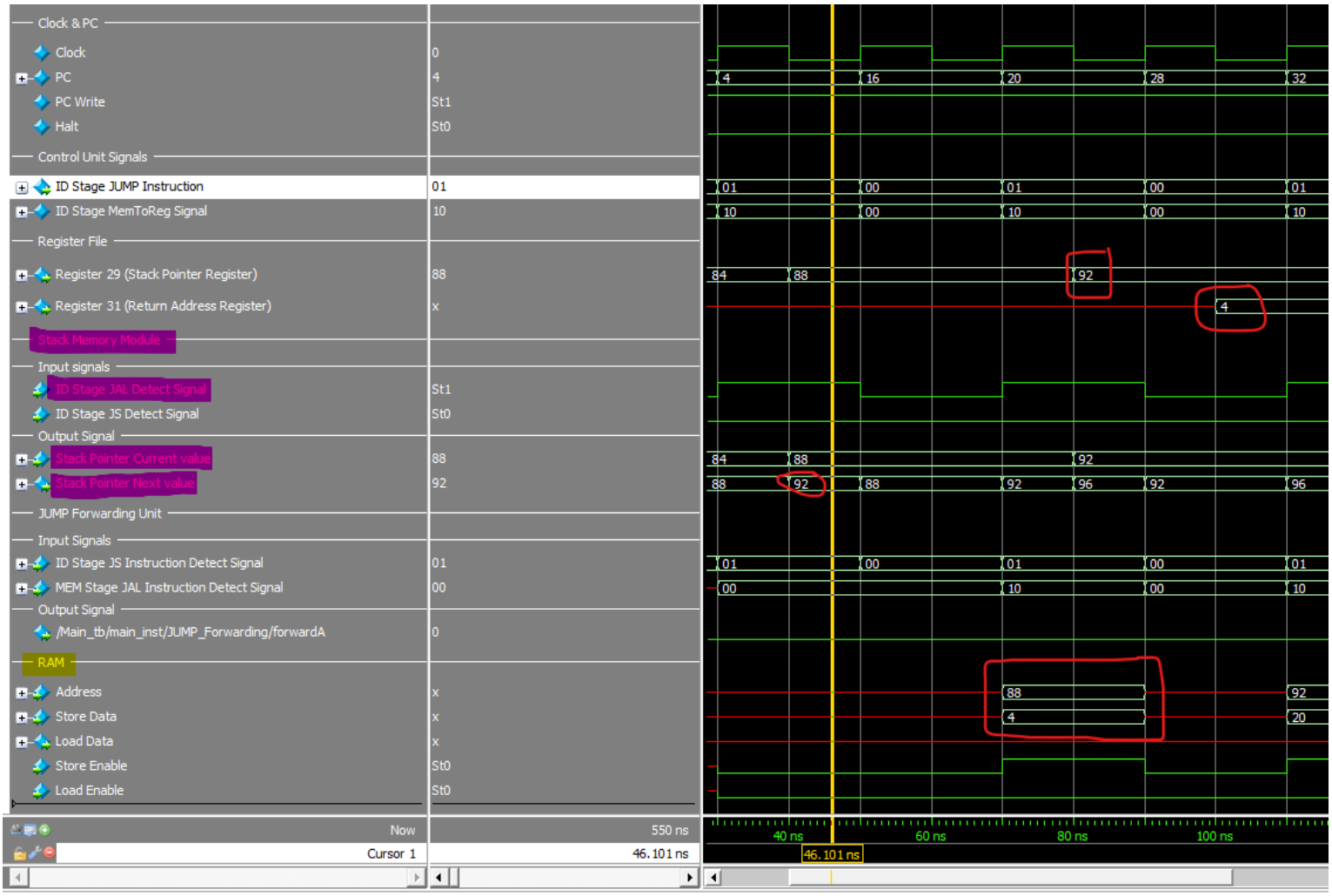

GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.22.4 First JS Instruction JUMP Forwarding



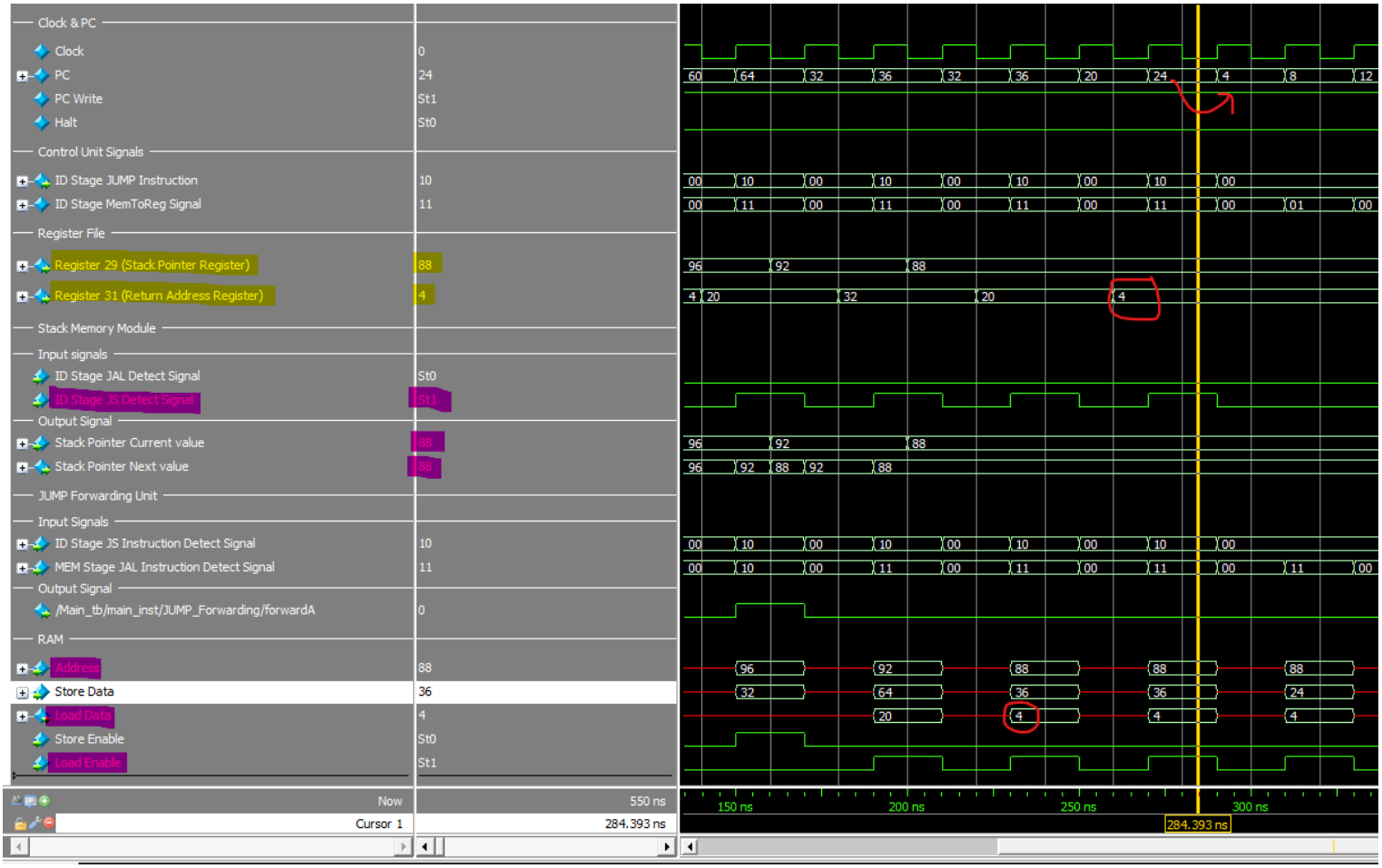

GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.22.5 First JAL Instruction



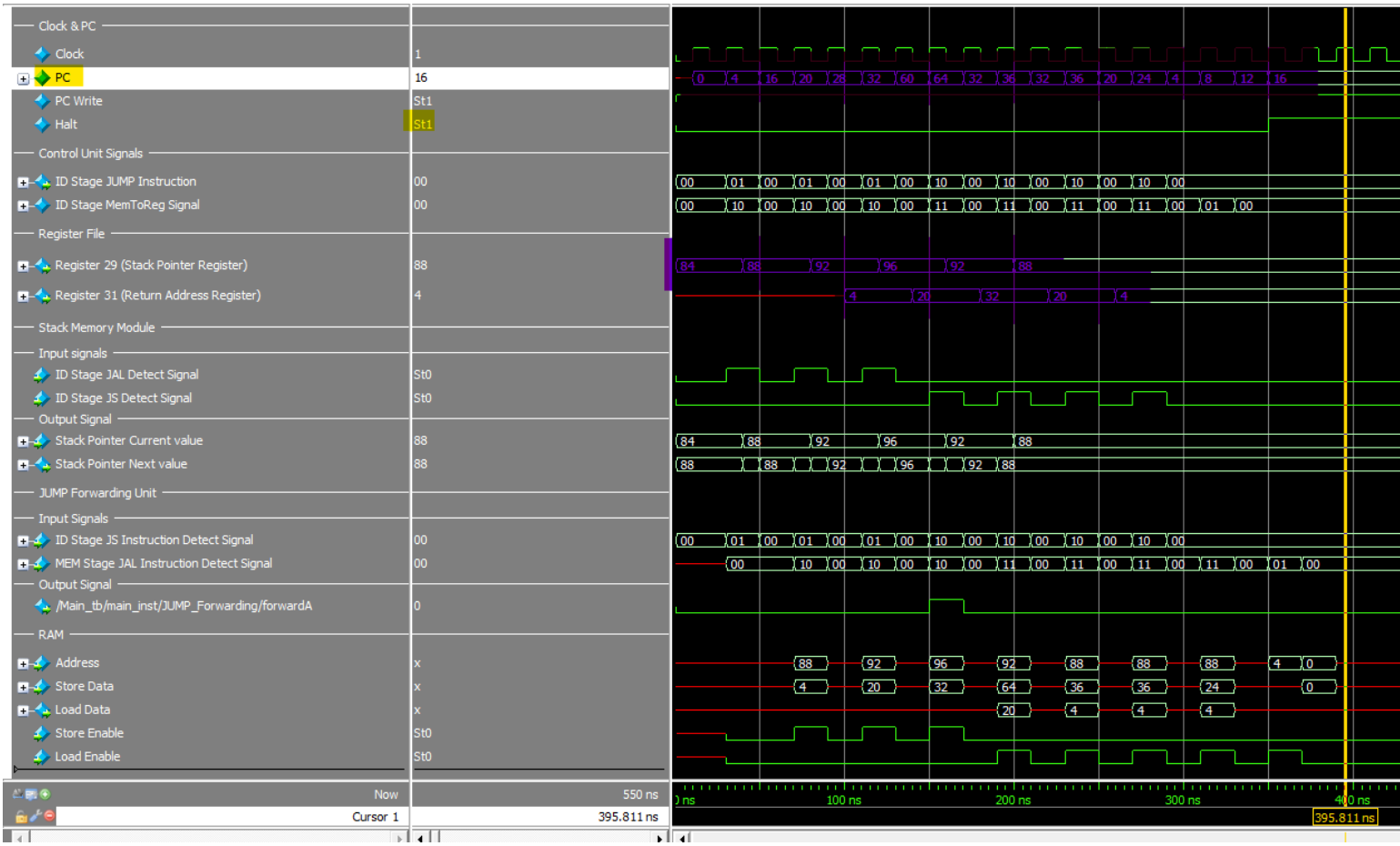

GitHub link → https://github.com/OmarAl-Saleh/MIPS

## 1.22.6 Last JS Instruction




GitHub link → https://github.com/OmarAl-Saleh/MIPS

1.22.7 CPU Status After Execution.



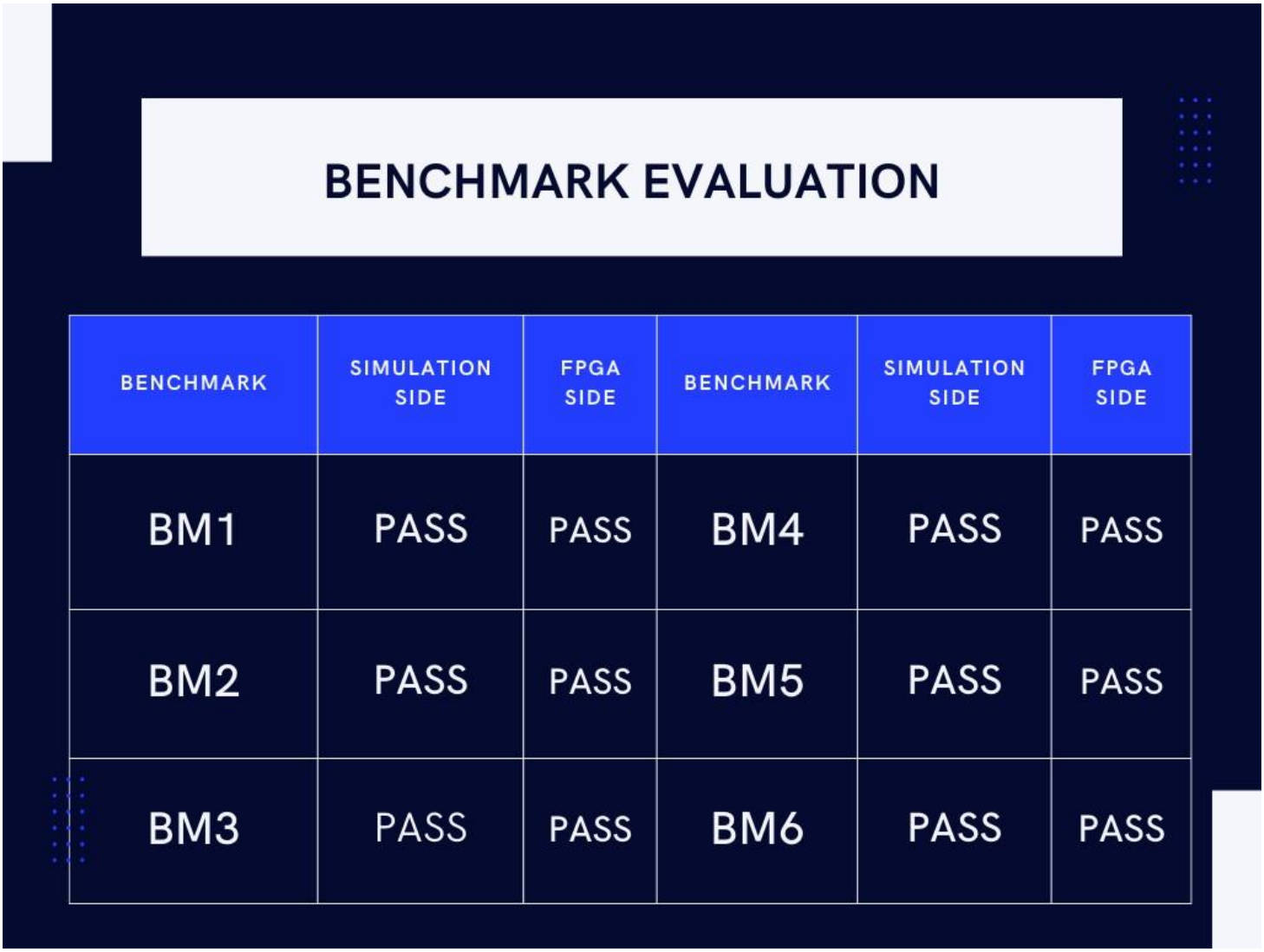

We have successfully finished the initial phase of constructing the single-cycle CPU. However, the branch component is still pending. We've complemented this phase with a robust test bench to ensure accuracy and functionality.

GitHub link → https://github.com/OmarAl-Saleh/MIPS

GitHub link → https://github.com/OmarAl-Saleh/MIPS

# Evaluation Sheet Statics

This table ensures that every benchmark, as discussed, successfully passes and yields the correct results in both simulation and FPGA implementation.




| BENCHMARK | SIMULATION SIDE | FPGA SIDE | BENCHMARK | SIMULATION SIDE | FPGA SIDE |
|---|---|---|---|---|---|
| BM1 | PASS | PASS | BM4 | PASS | PASS |
| BM2 | PASS | PASS | BM5 | PASS | PASS |
| BM3 | PASS | PASS | BM6 | PASS | PASS |

GitHub link → https://github.com/OmarAl-Saleh/MIPS

# Performance Cycle Statics Table

| Benchmark | BM1 | BM2 | BM3 | BM4 | BM5 | BM6 | AVERAGE |
|---|---|---|---|---|---|---|---|
| # INSTRUCTION EXECUTED | 4 | 9 | 10 | 74 | 9 | 406 | NA |
| #CYCLES EXECUTED | 8 | 13 | 15 | 79 | 13 | 411 | 89.9 |
| PIPELINE / SINGLE CYCLE RATION | 8/20 = 0.4 | 13/45 = 0.28 | 15/50 =0.3 | 79/370 =0.2 | 13/45 = 0.28 | 411/2030 = 0.2 | 0.27 |
| # CYCLES 100 TIMES (Loop) | 403/8 = 50.4 | 904/13 = 69.5 | 1104/15 = 73.6 | 7404/79 = 93.7 | 904/13 = 69.5 | 40700/411 = 99.5 | 76.0 |
| STATUS | Less | Less | Less | Less | Less | Less | NA |

GitHub link → https://github.com/OmarAl-Saleh/MIPS

# Performance FBGA Statics Table

| Benchmark | BM1 | BM2 | BM3 | BM4 | BM5 | BM6 | AVERAGE |
|---|---|---|---|---|---|---|---|
| **FPGA Clock Frequency (MHZ)** | **288** | **164** | **137** | **85** | **95** | **88** | **NA** |
| **# TIME UNITS (ns)** | 16 | 26 | 30 | 158 | 26 | 822 | 180 |
| **#CYCLES * PERIOD** | 8*2 | 13*2 | 15*2 | 79*2 | 13*2 | 411*2 | 90*2 |
| **Equal** | Yes | yes | yes | yes | yes | yes | yes |

GitHub link → https://github.com/OmarAl-Saleh/MIPS

# FBGA Compile Statics Table

| | LUTS | REGs | Memory Elements | Pins |
|---|---|---|---|---|
| **FPGA Design size** | **110** | **74** | **0** | **51** |
| **FPGA Compile Time** | Analysis & Synthesis : 00:00:24<br>Fitter: 00:00:16<br>Assembler:00:00:03<br>Timing Analyzer: 00:00:02<br>**Total : 00:00:45** | | | |
| **Number of Gates** | **124 / 49,760 ( < 1 % )** | | | |

GitHub link → https://github.com/OmarAl-Saleh/MIPS

# Conclusion

The journey to design a pipeline CPU, inspired by the venerable MIPS architecture, has led us to a comprehensive exploration of computer architecture, instruction execution, and the intricacies of CPU design. This endeavor has not only enriched our understanding of these fundamental concepts but has also yielded insights and a valuable educational resource for students and enthusiasts alike.

## 1.22.8 Key Findings and Implications

Through this project, we have made several key findings and drawn implications:

1. Educational Resource: We have successfully created an educational resource that unravels the intricacies of CPU design. The project offers a detailed, step-by-step account of how a single-cycle CPU operates, emphasizing simplicity and clarity, making it accessible to a broad audience.
2. Simplicity and Efficiency: The MIPS-inspired single-cycle design demonstrates the power of simplicity and efficiency in CPU architecture. We have observed that by adhering to a reduced instruction set and single-cycle execution, it is possible to achieve transparency in the execution of instructions.
3. Performance Analysis: Our performance analysis revealed that the single-cycle CPU's execution time is generally quicker compared to other designs. However, this speed comes at the expense of resource utilization, and it may not be the most efficient choice for all applications.

## 1.22.9 Project's objectives achieved.

While our project has achieved its primary objectives(**build pipeline cpu with good cycle time),** there are avenues for future work and exploration:

1. Advanced Instructions: Expanding the instruction set to include more complex operations, such as floating-point arithmetic or SIMD instructions, would further enrich the educational value of the CPU model.
2. Out of order processor: Out-of-order processors are often found in high-performance computing environments and modern microprocessors where speed and efficiency are paramount.

GitHub link → https://github.com/OmarAl-Saleh/MIPS

# Appendix A: CODE

## 1.23 ALU

```
module ALU(
            input    clk,
   input [31:0] A, // 32-bit input A
   input [31:0] B, // 32-bit input B
   input [3:0] ALUControl, // 4-bit ALU control
   input [4:0] ShiftAmount, // 5-bit input for shift amount
   input [2:0] branch_type,
            output reg [31:0] ALUOut, // 32-bit output
   output reg Zero // Zero flag


);
reg Overflow;
reg CarryOut;
//reg [31:0] ALUOut;
/*
addu
subu


*/
always @(*) begin
   case(ALUControl)
      4'b0000: ALUOut <= A + B; // Addition (signed)
      4'b0001: ALUOut <= A - B; // Subtraction (signed)
      4'b0010: ALUOut <= A * B; // Multiplication (signed)
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
        4'b0011: ALUOut <= A / B; // Division (signed/unsigned)
        4'b0100: ALUOut <= A << ShiftAmount; // Logical shift left
        4'b0101: ALUOut <= A >> ShiftAmount; // Logical shift right
        4'b0110: ALUOut <= A + B; // addu
                        4'b0111: ALUOut <= A - B; // subu
        4'b1000: ALUOut <= A & B; // Logical AND
        4'b1001: ALUOut <= A | B; // Logical OR
        4'b1010: ALUOut <= A ^ B; // Logical XOR
        4'b1011: ALUOut <= ~(A | B); // Logical NOR
        4'b1100: ALUOut <= (branch_type == 3'b101) ? (A >= B) ? 32'b1 : 32'b0 :
              (branch_type == 3'b110) ? (A <= B) ? 32'b1 : 32'b0 : 32'b0;
        4'b1101: ALUOut <= (A < B) ? 32'b1 : 32'b0; // less than
                        4'b1110: ALUOut <= (A > B) ? 32'b1 : 32'b0; // greater than




        default: ALUOut <= 32'b0; // Default operation
    endcase
end


always @(*) begin
    if (ALUControl == 4'b0000) begin // Addition (signed)
        Overflow <= (A[31] == B[31] && A[31] != ALUOut[31]);
        CarryOut <= (A[31] & B[31]) | (A[31] & ALUOut[31]) | (B[31] & ALUOut[31]);
    end else if (ALUControl == 4'b0001) begin // Subtraction (signed)
        Overflow <= (A[31] != B[31] && A[31] != ALUOut[31]);
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
        CarryOut <= (A[31] & B[31]) | (A[31] & ALUOut[31]) | (B[31] & ALUOut[31]);
    end else if (ALUControl == 4'b0010) begin // Multiplication (signed)
        Overflow <= (A[31] == B[31] && A[31] != ALUOut[31]);
        CarryOut <= 1'b0; // No carry-out for multiplication
    end else if (ALUControl == 4'b0011) begin // Division (signed/unsigned)
        Overflow <= (B == 32'b0); // Detect division by zero
        CarryOut <= 1'b0;
            end else if ((ALUOut < A) && (ALUOut < B) && (ALUControl == 4'b0110)) begin
            Overflow <= 1'b0;
            CarryOut <= 1'b1;
    end else begin
        Overflow <= 1'b0;
        CarryOut <= 1'b0;
    end
end
```

```
always @(*) begin
case(branch_type)
                                    3'b001: Zero <= (ALUOut == 32'b0) ? 1'b1 : 1'b0; // beq
        3'b010: Zero <= (ALUOut != 32'b0) ? 1'b1 : 1'b0; // bne
        3'b011: Zero <= (ALUOut == 32'b1) ? 1'b1 : 1'b0; // bgt
        3'b100: Zero <= (ALUOut == 32'b1) ? 1'b1 : 1'b0; // blt
        3'b101: Zero <= (ALUOut == 32'b1) ? 1'b1 : 1'b0; // bge
        3'b110: Zero <= (ALUOut == 32'b1) ? 1'b1 : 1'b0; // ble
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
default: Zero <= 1'b0;
endcase
end


endmodule
```

## 1.24 branch

```
module Branch(
input Branch_Flag,
input [3:0] ALUOp,
input [31:0] Data1,
input [31:0] Data2,
input [15:0] Target,
input [31:0] next_pc,
output reg [31:0] Branch_address,
output reg zero
);
always @(*) begin

        zero = 0 ;
        Branch_address [15:0] = Target [15:0] ;
        Branch_address [31:16] = {16{Target[15]}};
        Branch_address = Branch_address * 4 ;
        Branch_address = Branch_address + next_pc;

        if(Branch_Flag) begin
```

```
if(ALUOp == 4'b0100)begin // beq

        if (Data1 == Data2)begin
            zero = 1 ;
        end

    end

    else if(ALUOp == 4'b0101)begin // bne

        if (Data1 != Data2)begin
            zero = 1 ;
        end

    end

    else if(ALUOp == 4'b0110)begin // bgt

        if (Data1 > Data2)begin
            zero = 1 ;
        end

    end

    else if(ALUOp == 4'b0111)begin // blt
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
                if (Data1 < Data2)begin
                        zero = 1 ;
                end
        end

        else if(ALUOp == 4'b1000)begin // bge

                if (Data1 >= Data2)begin
                        zero = 1 ;
                end
                end

        else if(ALUOp == 4'b1001)begin // ble

                if (Data1 <= Data2)begin
                        zero = 1 ;
                end
        end

    end

end

endmodule
```

## 1.25 control unit

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
module
ControlUnit(Clock,Reset,opcode,RegDst,ALUSrc,MemtoReg,MemWrite,MemRead,ALUOp,RegWrite,Branch,Jump,funct,h
alt);

input wire [5:0] opcode;

input Clock , Reset ;  // Reset : 1 --> on | 0--> off

input wire [5:0] funct;

// wires

output wire ALUSrc,MemWrite,MemRead,RegWrite,Branch;

output wire [3:0] ALUOp;

output wire [1:0] Jump;

output wire [1:0] MemtoReg;

output wire [1:0] RegDst;

output wire halt;

// reg type

 reg reg_ALUSrc,reg_MemWrite,reg_MemRead,reg_RegWrite,reg_Branch;

 reg [3:0] reg_ALUOp;

 reg [1:0] reg_Jump;

 reg [1:0] reg_MemtoReg;

 reg [1:0] reg_RegDst;

 reg [5:0] reset_opcode ;

 reg reg_halt;




//************************** Singnal classification ********************************


//            opcode,  // 6-bit opcode from the instruction

//    RegDst,     // Control signal for selecting the destination register
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
//    ALUSrc,     // Control signal for ALU source selection (0 for register, 1 for immediate)
//    MemWrite,   // Control signal for memory write
//    MemRead,    // Control signal for memory read
//    Branch,     // Control signal for branching
//    ALUOp,      // Control signal for ALU operation
//    RegWrite    // Control signal for register write
//    reset_reg   // will handle the default value if the reset is on
//                   pc_load (    0->stall),(1->load)
//    pc_Store    // to enable REG 31 to store the return address when using JAL Instruction
//    JUMP[1:0]   // (MSB) --> is to indicate JS Instruction with Pull the stack (LSB) --> indicate jump instruction
//    MemToReg[1:0] // MSB) --> is to indicate JAL Instruction with Push the stack
//                  (LSB) --> Control signal for selecting the source for register write data

//*****************************************************************************************************
********************

always @(*)
begin
        if(Reset==1'b1)
        reset_opcode <=6'b111000; // this bits will call the default value and make the control unit reset
        else
        reset_opcode <=opcode;

end

//101101 halt opcode
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
always @(*)


        if(Reset==1'b1)begin
         // defualt value we can use it if we will implement reset or unsupported instructions
                reg_ALUSrc   = 1'b0;
                reg_RegWrite = 1'b0;
    reg_MemtoReg = 2'b00;
    reg_MemWrite = 1'b0;
    reg_MemRead  = 1'b0;
    reg_ALUOp    = 4'b0000;
    reg_RegDst   = 2'b00;
                reg_Branch   = 1'b0;
                reg_Jump     = 2'b00;
                reg_halt     = 1'b0;


end



        else begin
        case(opcode)


        6'b000000:begin
                if(funct == 6'b001000)begin
                //Jump Register (JS) instruction
                 reg_ALUSrc   = 1'b0;
                 reg_RegWrite = 1'b1;// for Stack Implementation
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
    reg_MemtoReg = 2'b11; // for Stack implementation (customize only for JS function to be MemToReg[1]=1 same as JAL Instruction)
    reg_MemWrite = 1'b0;
    reg_MemRead  = 1'b1;// for Stack Implementation
    reg_ALUOp    = 4'b0000;
    reg_RegDst   = 2'b00;
                reg_Branch   = 1'b0;
                reg_Jump     = 2'b10;
                reg_halt     = 1'b0;



                end
                else begin
                // R-type instruction
    reg_ALUSrc   = 1'b0;
                reg_RegWrite = 1'b1;
    reg_MemtoReg = 2'b00;
    reg_MemWrite = 1'b0;
    reg_MemRead  = 1'b0;
    reg_ALUOp    = 4'b0010;
    reg_RegDst   = 2'b01;
                reg_Branch   = 1'b0;
                reg_Jump     = 2'b00;
                reg_halt     = 1'b0;


        end
        end
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
                6'b100011:begin
                // load instruction
    reg_ALUSrc   = 1'b1;
                reg_RegWrite = 1'b1;
    reg_MemtoReg = 2'b01;
    reg_MemWrite = 1'b0;
    reg_MemRead  = 1'b1;
    reg_ALUOp    = 4'b0000;
    reg_RegDst   = 2'b00;
                reg_Branch   = 1'b0;
                reg_Jump     = 2'b00;
                reg_halt     = 1'b0;



        end


          6'b101011:begin
                // store instruction
    reg_ALUSrc   = 1'b1;
                reg_RegWrite = 1'b0;
    reg_MemtoReg = 2'b00;
    reg_MemWrite = 1'b1;
    reg_MemRead  = 1'b0;
    reg_ALUOp    = 4'b0000;
    reg_RegDst   = 2'b00;
                reg_Branch   = 1'b0;
```

```
                reg_Jump     = 2'b00;
                reg_halt     = 1'b0;



        end


                6'b001000:begin
                //immediate instruction (addi)
                reg_ALUSrc   = 1'b1;
                reg_RegWrite = 1'b1;
    reg_MemtoReg = 2'b00;
    reg_MemWrite = 1'b0;
    reg_MemRead  = 1'b0;
    reg_ALUOp    = 4'b0000;
    reg_RegDst   = 2'b00;
                reg_Branch   = 1'b0;
                reg_Jump     = 2'b00;
                reg_halt     = 1'b0;



        end


                6'b001100:begin
                //immediate instruction (andi)
                reg_ALUSrc   = 1'b1;
                reg_RegWrite = 1'b1;
    reg_MemtoReg = 2'b00;
```

```
reg_MemWrite = 1'b0;
reg_MemRead  = 1'b0;
reg_ALUOp    = 4'b0001;
reg_RegDst   = 2'b00;
                reg_Branch   = 1'b0;
                reg_Jump     = 2'b00;
                reg_halt     = 1'b0;



    end
                        6'b001101:begin
                //immediate instruction (ori)
                reg_ALUSrc   = 1'b1;
                reg_RegWrite = 1'b1;
reg_MemtoReg = 2'b00;
reg_MemWrite = 1'b0;
reg_MemRead  = 1'b0;
reg_ALUOp    = 4'b0011;
reg_RegDst   = 2'b00;
                reg_Branch   = 1'b0;
                reg_Jump     = 2'b00;
                reg_halt     = 1'b0;



    end

                        6'b000010:begin
```

```
                //Jump instruction (j)
                 reg_ALUSrc   = 1'b0;
                 reg_RegWrite = 1'b0;
    reg_MemtoReg = 2'b00;
    reg_MemWrite = 1'b0;
    reg_MemRead  = 1'b0;
    reg_ALUOp    = 4'b0000;
    reg_RegDst   = 2'b00;
                 reg_Branch   = 1'b0;
                 reg_Jump     = 2'b01;
                 reg_halt     = 1'b0;



        end


                6'b000011:begin
                //Jump and link instruction (jal)
                 reg_ALUSrc      = 1'b0;
                 reg_RegWrite   = 1'b1;
    reg_MemtoReg       = 2'b10;// for JAL instruction
    reg_MemWrite       = 1'b1; // for Stack implemntation
    reg_MemRead        = 1'b0;
    reg_ALUOp          = 4'b0000;
    reg_RegDst         = 2'b10;
                 reg_Branch              = 1'b0;
                 reg_Jump                = 2'b01;// for JS Instruction
                 reg_halt     = 1'b0;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
        end


                6'b000100:begin
                //branch equal
                 reg_ALUSrc             = 1'b0;
                 reg_RegWrite   = 1'b0;
    reg_MemtoReg       = 2'b00;
    reg_MemWrite       = 1'b0;
    reg_MemRead        = 1'b0;
    reg_ALUOp          = 4'b0100;
    reg_RegDst         = 2'b00;
                 reg_Branch             = 1'b1;
                 reg_Jump               = 2'b00;
                 reg_halt    = 1'b0;



        end



                6'b000101:begin
                //branch not equal
                 reg_ALUSrc             = 1'b0;
                 reg_RegWrite   = 1'b0;
    reg_MemtoReg       = 2'b00;
    reg_MemWrite       = 1'b0;
    reg_MemRead        = 1'b0;
```

```
reg_ALUOp           = 4'b0101;
reg_RegDst          = 2'b00;
                    reg_Branch              = 1'b1;
                    reg_Jump                = 2'b00;
                    reg_halt    = 1'b0;



        end


                    6'b000110:begin
                    //branch greater than
                    reg_ALUSrc              = 1'b0;
                    reg_RegWrite   = 1'b0;
reg_MemtoReg        = 2'b00;
reg_MemWrite        = 1'b0;
reg_MemRead         = 1'b0;
reg_ALUOp           = 4'b0110;
reg_RegDst          = 2'b00;
                    reg_Branch              = 1'b1;
                    reg_Jump                = 2'b00;
                    reg_halt    = 1'b0;



        end


                    6'b000111:begin
                    //branch less than
```

```
                reg_ALUSrc              = 1'b0;
                reg_RegWrite   = 1'b0;
    reg_MemtoReg      = 2'b00;
    reg_MemWrite      = 1'b0;
    reg_MemRead       = 1'b0;
    reg_ALUOp         = 4'b0111;
    reg_RegDst        = 2'b00;
                reg_Branch              = 1'b1;
                reg_Jump                = 2'b00;
                reg_halt    = 1'b0;



        end


                6'b001001:begin
                //branch greater or equal
                reg_ALUSrc              = 1'b0;
                reg_RegWrite   = 1'b0;
    reg_MemtoReg      = 2'b00;
    reg_MemWrite      = 1'b0;
    reg_MemRead       = 1'b0;
    reg_ALUOp         = 4'b1000;
    reg_RegDst        = 2'b00;
                reg_Branch              = 1'b1;
                reg_Jump                = 2'b00;
                reg_halt    = 1'b0;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
        end


                6'b001010:begin
                //branch less or equal
                 reg_ALUSrc             = 1'b0;
                 reg_RegWrite   = 1'b0;
    reg_MemtoReg       = 2'b00;
    reg_MemWrite       = 1'b0;
    reg_MemRead        = 1'b0;
    reg_ALUOp          = 4'b1001;
    reg_RegDst         = 2'b00;
                 reg_Branch             = 1'b1;
                 reg_Jump               = 2'b00;
                 reg_halt    = 1'b0;


        end


        // halt instruction


        6'b101101:begin
                 reg_ALUSrc             = 1'b0;
                 reg_RegWrite   = 1'b0;
    reg_MemtoReg       = 2'b00;
    reg_MemWrite       = 1'b0;
    reg_MemRead        = 1'b0;
    reg_ALUOp          = 1'b0;
```

```
    reg_RegDst          = 2'b00;
                reg_Branch          = 1'b0;
                reg_Jump            = 2'b00;
                reg_halt    = 1'b1;

        end

//      6'b111000:begin
//              //branch less or equal
//              reg_ALUSrc          = 1'b0;
//              reg_RegWrite   = 1'b0;
//    reg_MemtoReg    = 2'b00;
//    reg_MemWrite    = 1'b0;
//    reg_MemRead     = 1'b0;
//    reg_ALUOp               = 4'b0000;
//    reg_RegDst              = 2'b00;
//              reg_Branch          = 1'b0;
//              reg_Jump            = 2'b00;
//              reg_halt    = 1'b1;
//
//      end
//



        default : begin
         // defualt value we can use it if we will implement reset or unsupported instructions
```

```
                reg_ALUSrc   = 1'b0;
                reg_RegWrite = 1'b0;
    reg_MemtoReg = 2'b00;
    reg_MemWrite = 1'b0;
    reg_MemRead  = 1'b0;
    reg_ALUOp    = 4'b0000;
    reg_RegDst   = 2'b00;
                reg_Branch   = 1'b0;
                reg_Jump     = 2'b00;
                reg_halt     = 1'b0;

        end


endcase
end

// assign the output wires
assign RegDst = reg_RegDst;
assign ALUSrc = reg_ALUSrc;
assign MemtoReg = reg_MemtoReg;
assign MemWrite = reg_MemWrite;
assign MemRead = reg_MemRead;
assign RegWrite = reg_RegWrite;
assign ALUOp = reg_ALUOp;
assign Branch= reg_Branch;
assign Jump = reg_Jump;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
assign halt = reg_halt;



endmodule
```

## 1.26 EX_MEM_Register

```
module EX_MEM_Register (
// input signals

        clk,In_Address,In_Write_Data, In_Rd,

        // output signals
        Out_Address,Out_Write_Data, Out_Rd,
        // control unit input signals

        // MEM signals
        In_MemWrite,In_MemRead,

        // WB signals
        In_RegWrite, In_MemtoReg,

        // control unit output signals

        // MEM signals
        Out_MemWrite,Out_MemRead,

        // WB signals
        Out_RegWrite, Out_MemtoReg,
```

```
        In_PC,Out_PC
);



input clk ;

input [31:0] In_Address,In_Write_Data,In_PC ; // this refer to ALU reselt as an address and Read data 2 from register file as an write data to store in memory

input [4:0] In_Rd; // the values come from the instruction in ID stage

input In_MemWrite,In_MemRead,In_RegWrite;

input [1:0] In_MemtoReg;



output reg [31:0]  Out_Address,Out_Write_Data,Out_PC ;

output reg [4:0]   Out_Rd;

output reg Out_MemWrite,Out_MemRead,Out_RegWrite;

output reg [1:0] Out_MemtoReg;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
always @(posedge clk)
        begin

                Out_Address <= In_Address;
                Out_Write_Data <= In_Write_Data;

                Out_Rd <=In_Rd;

                Out_MemWrite <= In_MemWrite;
                Out_MemRead <= In_MemRead;
                Out_RegWrite <= In_RegWrite;

                Out_MemtoReg<=In_MemtoReg;

                Out_PC<=In_PC;

        end

endmodule
```

## 1.27 ForwardingUnit

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
module ForwardingUnit (
    input [4:0] rs1_ID_EX,
    input [4:0] rs2_ID_EX,
    input [4:0] rd_EX_MEM,
    input [4:0] rd_MEM_WB,
    input RegWrite_EX_MEM,
    input RegWrite_MEM_WB,
    output reg [1:0] forwardA,
    output reg [1:0] forwardB
);

// Forwarding logic for ALU inputs
always@(*)
begin
    // Default: No forwarding
    forwardA = 2'b00;
    forwardB = 2'b00;

    // Forwarding for rs1
    if (RegWrite_EX_MEM && (rd_EX_MEM != 0) && (rd_EX_MEM == rs1_ID_EX)) begin
        forwardA = 2'b10;  // Forward from EX/MEM
    end else if (RegWrite_MEM_WB && (rd_MEM_WB != 0) && (rd_MEM_WB == rs1_ID_EX)) begin
        forwardA = 2'b01;  // Forward from MEM/WB
    end

    // Forwarding for rs2
    if (RegWrite_EX_MEM && (rd_EX_MEM != 0) && (rd_EX_MEM == rs2_ID_EX)) begin
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
        forwardB = 2'b10;  // Forward from EX/MEM
    end else if (RegWrite_MEM_WB && (rd_MEM_WB != 0) && (rd_MEM_WB == rs2_ID_EX)) begin
        forwardB = 2'b01;  // Forward from MEM/WB
    end
end

endmodule
```

## 1.28 Hazard unit

```
module
Hazard_Unit(D_rs,D_rt,EX_rt,EX_MemRead,ID_EX_FLUSH,IF_ID_write,PC_write,ID_Branch,MEM_rt,MEM_MemRead);

input [4:0] D_rs, D_rt ; // this signals is rs and rt for the instruction in ID stage

input [4:0] EX_rt ; input EX_MemRead;// this signals is rt and Memory read control signal for instruction EX stage (first preceding)

input ID_Branch  ; // this a control signal from the id Unit to determin if we have a branch instruction in id stage or not

input [4:0] MEM_rt ; // this signal is rt for the instruction in Mem stage (second preceding instruction)

input MEM_MemRead ; // this signal is memory read control signal for instruction in Memory stage to determin if is load instruction

output wire ID_EX_FLUSH ,IF_ID_write , PC_write ;// this signals is to implement the stall

reg reg_ID_EX_FLUSH ,reg_IF_ID_write , reg_PC_write ;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
always@(*)
begin

// first type of hazard is load-use hazard that occure when we find a depandancy between load instruction and its followed instruction

reg_ID_EX_FLUSH <= 0; // to solve test case 5 bug
reg_IF_ID_write <= 1;
reg_PC_write <= 1;

        if((EX_MemRead == 1'b1) && (ID_Branch == 1'b0))
                begin
                        if((D_rs == EX_rt) || (D_rt == EX_rt))
                                begin

                                reg_ID_EX_FLUSH <= 1; //(Reset for ID_EX_REG) to select zeroes for the control mux that control the flow of ID/EX register
                                reg_IF_ID_write <= 0; // to disable the the IF/ID register so he can't load new values and keep his old value
                                reg_PC_write <= 0; // to disable the the PC so he can't load new values and keep his old value

                                end
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
		end

	//second type hazard is Data hazards for branches that occur when we find a depandancy between branch and first preceding or second preceding if is it a load instruction

	else
		begin

		if (ID_Branch == 1'b1)
			begin

			if  ( ((D_rs == EX_rt) || (D_rt == EX_rt)) || ( (MEM_MemRead == 1'b1) && ( (D_rs == MEM_rt) || (D_rt == MEM_rt) ) ))
				begin

				reg_ID_EX_FLUSH <= 1; // to select zeroes for the control mux that control the flow of ID/EX register
				reg_IF_ID_write <= 0; // to disable the the IF/ID register so he can't load new values and keep his old value
				reg_PC_write <= 0; // to disable the the PC so he can't load new values and keep his old value

				end

			end
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
        else
                begin
                        reg_ID_EX_FLUSH <= 0;
                        reg_IF_ID_write <= 1;
                        reg_PC_write <= 1;


                end
        end
end

// assign output wires
 assign ID_EX_FLUSH = reg_ID_EX_FLUSH;
 assign IF_ID_write = reg_IF_ID_write;
        assign PC_write = reg_PC_write;



endmodule
```

## 1.29 ID_EX_Register

```
module ID_EX_Register (
// input signals
        clk,reset
```

```
        ,In_Reg_File_Data1,In_Reg_File_Data2,
        In_offset,In_Rs,In_Rt,In_Rd,
        // output signals
  Out_Reg_File_Data1,Out_Reg_File_Data2,
        Out_offset,Out_Rs,Out_Rt,Out_Rd,
        // control unit input signals
        // EX signals
        In_ALUSrc, In_ALUOp ,In_RegDst,In_func, In_shamt,

        // MEM signals
        In_MemWrite,In_MemRead,

        // WB signals
        In_RegWrite, In_MemtoReg,

        // control unit output signals
        // EX signals
        Out_ALUSrc, Out_ALUOp ,Out_RegDst,Out_func, Out_shamt,

        // MEM signals
        Out_MemWrite,Out_MemRead,

        // WB signals
        Out_RegWrite, Out_MemtoReg,
        // JAL PC WB Value
        In_PC,Out_PC
);
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

input clk , reset ; // reset is active when we catch a hazard in decode we use it instead of a mux to make all the values equal zeroes

input [31:0] In_Reg_File_Data1,In_Reg_File_Data2,In_offset,In_PC ; // the data come from the register file and the offset from the sign extend

input [4:0] In_Rs,In_Rt,In_Rd; // the values come from the instruction in ID stage

input In_ALUSrc,In_MemWrite,In_MemRead,In_RegWrite;

input [3:0] In_ALUOp;

input [1:0] In_MemtoReg;

input [1:0] In_RegDst;

input [5:0] In_func;

input [4:0] In_shamt;

output reg [31:0]  Out_Reg_File_Data1, Out_Reg_File_Data2, Out_offset,Out_PC ;

output reg [4:0]   Out_Rs , Out_Rt  , Out_Rd;

output reg Out_ALUSrc,Out_MemWrite,Out_MemRead,Out_RegWrite;

output reg [3:0] Out_ALUOp;

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
output reg [1:0] Out_MemtoReg;

output reg [1:0] Out_RegDst;


output reg [5:0] Out_func;


output reg [5:0] Out_shamt;



always @(posedge clk)
        begin

                if(reset == 1'b1) // if it one so we catch a hazard
                        begin


                        Out_Reg_File_Data1<=32'b0;

                        Out_Reg_File_Data2<=32'b0;

                        Out_offset<=32'b0;


                        Out_Rs <=5'b0;

                        Out_Rt <=5'b0;

                        Out_Rd <=5'b0;


                        Out_ALUSrc <= 1'b0;

                        Out_MemWrite <= 1'b0;

                        Out_MemRead <= 1'b0;

                        Out_RegWrite <= 1'b0;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
Out_ALUOp<=4'b0;
Out_MemtoReg<=2'b0;
Out_RegDst<=2'b0;

Out_func<=6'b000000;
Out_shamt<=5'b00000;
Out_PC<=32'b0;




end

else
begin

Out_Reg_File_Data1<= In_Reg_File_Data1;
Out_Reg_File_Data2<= In_Reg_File_Data2;
Out_offset<= In_offset;

Out_Rs <=In_Rs;
Out_Rt <=In_Rt;
Out_Rd <=In_Rd;

Out_ALUSrc <= In_ALUSrc;
Out_MemWrite <= In_MemWrite;
Out_MemRead <= In_MemRead;
Out_RegWrite <= In_RegWrite;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
                Out_ALUOp<=In_ALUOp;
                Out_MemtoReg<=In_MemtoReg;
                Out_RegDst<=In_RegDst;

                Out_func<=In_func;
                Out_shamt<=In_shamt;

                Out_PC<=In_PC;

                end

        end

endmodule
```

## 1.30 ID_EX_Register

```
module IF_ID_Register (clk,reset,enable,Instruction_in,PC_in,Branch_Control,Instruction_out,PC_out,
opcode,rs,rt,rd,shamt,funct,addr,jump,halt);

input clk ;
input reset; // 0-->1 to delete the previous instruction and store a nop instruction (flush) with opcode 111000 according to our control unit default value
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

//we use it in branch miss prediction this signal come from zero in branch alu

input enable; // 0-->1 to enable the register to received new instruction we use

//it in hazard detection to make flush and the output opcode is 111000 it comes only from  the hazard detection unit

input halt ; // 0 ---> stop pc and regenerate halt instruction to control unit

input [31:0] Instruction_in; // the instruction come from fetch stage

input [31:0] PC_in; // the value of pc+4 come from fetch stage

input Branch_Control; // to know if the function cause reset (zero signal) is from branch instruction in ID stage

output reg [31:0] Instruction_out; // the instruction output to decode stage

output reg [5:0] opcode;

output reg [4:0] rs;

output reg [4:0] rt;

output reg [4:0] rd;

output reg [4:0] shamt;

output reg [5:0] funct;

output reg [15:0] addr;

output reg [25:0] jump;

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
output reg [31:0] PC_out ; // the pc output to decode stage




always@(posedge clk)
        begin

                if((enable == 1'b0) )
                        begin

                                if((reset == 1'b1) || (Branch_Control == 1'b1)  )
                                // case 3 : when to flush the register in reset status or when we catch a branch miss predict
                                        begin

                                                Instruction_out <=32'b11100000000000000000000000000000; // nop
                                                opcode <= 6'b111000;
        rs <= 5'bx;
                                                rt <= 5'bx;
                                                rd <= 5'bx;
                                                shamt <= 5'bx;
                                                funct <= 6'bx;
                                                addr <= 16'bx;
                                                jump <= 26'bx;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
                PC_out <= PC_in;



            end

        else
            begin
            // case 2 : when we do not have any problem
                Instruction_out <=Instruction_in;
                opcode <= Instruction_in [31:26];
                rs <= Instruction_in [25:21];
                rt <= Instruction_in [20:16];
                rd <= Instruction_in [15:11];
                shamt <= Instruction_in [10:6];
                funct <= Instruction_in [5:0];
                addr <= Instruction_in [15:0];
                jump <= Instruction_in [25:0];
                PC_out <= PC_in;


            end



        if((halt == 1'b1))
                begin
         // to stop the pc
```

```
                                        Instruction_out <=32'b10110100000000000000000000000000; // nop
                                        opcode <= 6'b101101;
        rs <= 5'bx;
                                        rt <= 5'bx;
                                        rd <= 5'bx;
                                        shamt <= 5'bx;
                                        funct <= 6'bx;
                                        addr <= 16'bx;
                                        jump <= 26'bx;
                                        PC_out <= PC_in;

                                        end

                    end

        end

endmodule
```

## 1.31 instruction memory

```
module INST_MEM #(
  parameter size = 32,
  parameter data_width = 32
)(
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
 // input clk,
  input reset,
  input [31:0] address,
  output reg [31:0] inst_out

);
   reg [31:0] inst_mem [0:size - 1];



 integer i;

//  endmodule

reg state = 1'b0;



//
  always @(*) begin

    if (reset) begin
      for (i = 0; i < size; i = i + 1) begin
        inst_mem[i] <= 32'b0;
      end

    end else begin
       case (state)
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
1'b0: begin
  state <= 1'b1;
                                    // Enter here the Instructions of the program

  // THIS TEST FOR STACK TESTING USING JAL & JS INSTRUCTIONS (Nested Subroutine) 3 push 3 pull
  // the goal is test the stack functionality

  /*Address 0 */inst_mem[0] = 32'b00001100000000000000000000000100;//JAL Jump to address 16 and save R31 = 4
  /*Address 4 */inst_mem[1] = 32'b10001100000000010000000000000100;//lw reg1=3 (the jump will skip it and return later)
  /*Address 8 */inst_mem[2] = 32'b00000000000000001110000000000000; //ADD R28,R0,R0 (R28=0)

 /* Address 12 */ inst_mem[3] = 32'b10110100001000100001100000100000; //Halt (stop PC & End the Program)

  // the PC final value will be 16

  /*Address 16 */inst_mem[4] = 32'b00001100000000000000000000000111;//JAL Jump to address 28 and save R31 = 20

 /*Address 20 */inst_mem[5] = 32'b00000011111001010010000000001000;// JS jump to address store in REG 31 so jump to address 4

  /*Address 28 */inst_mem[7] = 32'b00001100000000000000000000001111;//JAL Jump to address 60 and save R31 = 32
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

/*Address 32 */inst_mem[8] = 32'b00000011111001010010000000001000;// JS jump to address store in REG 31 so jump to address 20

/*Address 60 */inst_mem[15] = 32'b00000011111001010010000000001000;// JS jump to address store in REG 31 so jump to address 32

// In this instruction we need JUMP Forwarding Unit between JAL MEM Stage instruction and JS ID Stage Instruction

//------------------------------------------------------------

//

///* LAST ADDRESS */ inst_mem[3] = 32'b10110100001000100001100000100000; //Halt

// every program should end with halt signal

end

1'b1: begin

inst_out <= inst_mem[address >> 2];

end

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
      endcase
   end
end

endmodule
```

## 1.32 JUMP

```
module JUMP(
input JUMP_FLAG,
input [25:0] jump_offset,
input [31:0] next_pc,
output reg [31:0] JUMP_address
);

always @(*)
begin

        if(JUMP_FLAG)
                begin
                        JUMP_address = {next_pc[31:28], jump_offset, 2'b00};
                end

        else
                begin
                        JUMP_address =32'bx;
                end

end
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

endmodule

## 1.33 MEM_WB

```
module MEM_WB_Register (
// input signals

        clk,In_RAM_Data,In_Immediate_Data, In_Rd,

        // output signals
        Out_RAM_Data,Out_Immediate_Data, Out_Rd,
        // control unit input signals



        // WB signals
        In_RegWrite, In_MemtoReg,

        // control unit output signals

        // WB signals
        Out_RegWrite, Out_MemtoReg,
        In_PC,Out_PC

);



input clk ;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
input [31:0] In_RAM_Data,In_Immediate_Data,In_PC ; // this refer to RAM output data and Immediate data

input [4:0] In_Rd; // the values come from the instruction in ID stage

input In_RegWrite;

input [1:0] In_MemtoReg;



output reg [31:0]  Out_RAM_Data,Out_Immediate_Data,Out_PC ;

output reg [4:0]   Out_Rd;

output reg Out_RegWrite;



output reg [1:0] Out_MemtoReg;



always @(posedge clk)
        begin

                Out_RAM_Data <= In_RAM_Data;
                Out_Immediate_Data <= In_Immediate_Data;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
                Out_Rd <=In_Rd;


                Out_RegWrite <= In_RegWrite;
                Out_MemtoReg<=In_MemtoReg;


                Out_PC<=In_PC;



        end
```

## 1.34 Main

```
module Main (
input clk,
input reset
);
```

```
//*************************************************************************************************
**********************

// ************************************* Fetch Stage
**********************************************************************

//
*************************************************************************************************
****************

 wire Branch_Zero_Signal ; // we will use it in pc load
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
//PC
 wire [31:0] pc_final;//input
 wire [31:0] pc_out;//output
 wire [31:0] next_pc;//pc+4
 wire [31:0] pc_inc;//4
 assign pc_inc = 32'b00000000000000000000000000000100;// constant value(4)
 wire PC_write;//control
 // for halt instruction implementation
 wire halt; // to end the program last instruction in every program

 PC #(.first_address(0),  .pc_inc(4) )
 pc_inst (
   .clk(clk),
   .reset(reset),
          .target(pc_final),
           .pc_load(PC_write && ~halt), //second edition come from hazard detection unit
   .pc(pc_out)
  );

//end of PC
//-------------------------------------

adder add(
          .a(pc_inc),
          .b(pc_out),
          .c(next_pc)
          );
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
//-----------------------------------
//inst_mem
wire[31:0] inst_out;

INST_MEM #(.size(32),.data_width(32) )
inst_mem (
   .reset(reset),
   .address(pc_out),
   .inst_out(inst_out)
 );

 // End of Instruction Memory ---------------------------------------

// IF_ID_Register


  // Outputs
  wire [31:0] IF_ID_Instruction_out;
  wire [31:0] IF_ID_PC_out;
// to split it from Instruction memory signal

wire [5:0] IF_ID_opcode;
wire [4:0] IF_ID_rs;
wire [4:0] IF_ID_rt;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
wire [4:0] IF_ID_rd;

wire [4:0] IF_ID_shamt;

wire [5:0] IF_ID_funct;

wire [15:0] IF_ID_addrs;

wire [25:0] IF_ID_jump_offset;

 wire [1:0] Jump_signal;//Jump_signal[0] really jump signal

wire IF_ID_write ,Branch ; // the enable signal that come from hazard unit and branch signal from control Unit


IF_ID_Register IF_ID_R (

    .clk(clk),

    .reset(reset),

    .enable(~(IF_ID_write)),//  it designed to be negative because the case of first instruction when no instruction is in ID stage

    .Instruction_in(inst_out), // the output instruction from Instruction Memory

    .PC_in(next_pc),

    .Branch_Control((Branch & Branch_Zero_Signal)|(Jump_signal[0] | Jump_signal[1]) ), // if we catch branch depandancy

      //or we find jump or jump and link instructions or JS --> Jump Register

    .Instruction_out(IF_ID_Instruction_out),

    .PC_out(IF_ID_PC_out),// Maybe I must implement the pc in every register but know I will not do it

            .opcode(IF_ID_opcode),

    .rs(IF_ID_rs),

    .rt(IF_ID_rt),

    .rd(IF_ID_rd),

    .shamt(IF_ID_shamt),

    .funct(IF_ID_funct),

    .addr(IF_ID_addrs),// use for calculate the branch target address
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
        .jump(IF_ID_jump_offset),
        .halt(halt)
  );

// End of
//*************************************************************************************************
************************************

//************************************************ Decode Stage
**********************************************************************

//*************************************************************************************************
************************************


//sign extend
wire [31:0] immediate_value;
wire [31:0] ID_EX_immediate_value;

sign_extend extender (
   .extend(IF_ID_addrs),
   .extended(immediate_value)
);

//end of sign extend




//----------------------------------
//ControlUnit
  wire ALUSrc, MemWrite, MemRead, RegWrite;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
  wire [3:0] ALUOp;
  wire [1:0] RegDst;
  wire [1:0] MemtoReg;
  //wire halt;

  // we don't need pc_load and pc_store signal anymore (useless) (clear phase)

ControlUnit control_inst (
    .Clock(clk),
    .Reset(reset),
    .opcode(IF_ID_opcode),
    .RegDst(RegDst),
    .ALUSrc(ALUSrc),
    .MemtoReg(MemtoReg),
    .MemWrite(MemWrite),
    .MemRead(MemRead),
    .RegWrite(RegWrite),
    .ALUOp(ALUOp),
            .Branch(Branch),
            .Jump(Jump_signal),
            .funct(IF_ID_funct),
            .halt(halt)

  );
//end of ControlUnit
//-----------------------------------
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
// Reg_File


   wire [4:0] write_reg_input;


   wire [31:0] WB_Writedata; // we use it to hold the data from wb to register file to write it in addition we use it in forwarding Unit

        wire [4:0] MEM_WB_rd;

   wire MEM_WB_RegWrite;


        wire [31:0] ReadData1;

   wire [31:0] ReadData2;


        wire [1:0] MEM_WB_MemtoReg;// we use it in implementation of JAL Instruction as enable to register 31




          RegisterFile reg_file_inst (

      .Clock(clk),// we make the the register file write on the negative edge so we can write on a register and read his data in same clock

      .Reset(reset),

      .ReadReg1(IF_ID_rs),

      .ReadReg2(IF_ID_rt),

      .WriteReg(MEM_WB_rd),

      .Reg_write_Control(MEM_WB_RegWrite),// we must take it from WB Stage

      .WriteData(WB_Writedata),

      .ReadData1(ReadData1),
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
    .ReadData2(ReadData2),
                .PC_Store(MEM_WB_MemtoReg[1]),// to indicate JAL OR JS Instruction in WB Stage to store the
Return address that come from RAM
                        // must update in pull operation
                .PUSH_Stack(MemtoReg[1] && ~MemtoReg[0]),// to indicate JAL Instruction with Stack
    .PULL_Stack(Jump_signal[1]) // to indicate JS Instruction with Stack
  );



        wire [31:0] Branch_address;



// End of Register File ----------------------------------------------------


// implement the branch forwarding Unit

        wire [4:0] EX_MEM_rd,ID_EX_rd; // we need to use the rd in branch forwarding;
        wire ID_EX_RegWrite , EX_MEM_RegWrite;
        wire [1:0] Branch_Select_Forward_A ,Branch_Select_Forward_B;
        wire [31:0] EX_MEM_ALU_Result ;
        wire [31:0] MEM_WB_RAM_Data;

        wire [31:0] Final_Branch_ReadData1;
  wire [31:0] Final_Branch_ReadData2;// the output of forwarding MUXES

         ForwardingUnit Forwarding_Branch (
    .rs1_ID_EX(IF_ID_rs),
```

```
    .rs2_ID_EX(IF_ID_rt),
    .rd_EX_MEM(EX_MEM_rd),
    .rd_MEM_WB(MEM_WB_rd),
    .RegWrite_EX_MEM(EX_MEM_RegWrite),
    .RegWrite_MEM_WB(MEM_WB_RegWrite),
    .forwardA(Branch_Select_Forward_A),
    .forwardB(Branch_Select_Forward_B)
  );

MUX4_1 Branch_Forwarding_A_MUX(
.a(ReadData1),
.b(WB_Writedata),
.c(EX_MEM_ALU_Result),// EX_MEM
.select(Branch_Select_Forward_A),
.out(Final_Branch_ReadData1)
);

MUX4_1 Branch_Forwarding_B_MUX(
.a(ReadData2),
.b(WB_Writedata),
.c(EX_MEM_ALU_Result),
.select(Branch_Select_Forward_B),
.out(Final_Branch_ReadData2)
);

        // implement branch and jump unit with PC MUXES
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
        Branch Branch_Unit (
   .Branch_Flag(Branch),
   .ALUOp(ALUOp),
   .Data1(Final_Branch_ReadData1),// There is a  forwarding on this signal
   .Data2(Final_Branch_ReadData2),// There is a  forwarding on this signal
   .Target(IF_ID_addrs),
   .next_pc(IF_ID_PC_out),
   .Branch_address(Branch_address),
   .zero(Branch_Zero_Signal)
 );



// JUMP (JS) Forwarding Unit



// implement the branch forwarding Unit

        wire JUMP_Select_Forward_A;
        wire [1:0] EX_MEM_MemtoReg;
        wire [31:0] Final_JUMP_ReadData;
        wire [31:0] EX_MEM_PC_out;

 ForwardingUnit_JUMP JUMP_Forwarding (
   .JS_JUMP(Jump_signal),
   .EX_MEM_MemtoReg(EX_MEM_MemtoReg),
   . forwardA(JUMP_Select_Forward_A)
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
);

MUX2_1 JUMP_Forwarding_MUX(
.a(ReadData1),
.b(EX_MEM_PC_out),
.select(JUMP_Select_Forward_A),
.out(Final_JUMP_ReadData)
);

  // Calculate PC Target Address
  wire [31:0] pc_branch , JUMP_Target_address;

MUX2_1 pc_target(
.a(next_pc),
.b(Branch_address),
.select((Branch & Branch_Zero_Signal)),
.out(pc_branch)
);
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
// implement jump here


JUMP Jump_Unit (
   .JUMP_FLAG(Jump_signal[0]),
   .jump_offset(IF_ID_jump_offset),
            .next_pc(next_pc),
   .JUMP_address(JUMP_Target_address)
  );


MUX4_1 pc_final_main(
.a(pc_branch),
.b(JUMP_Target_address),// I don't know where is he
.c(Final_JUMP_ReadData),// must be the output from JS Instruction Forwarding MUX
.select(Jump_signal),
.out(pc_final)
);
// end branch and Jump implementation


// Hazard dediction Unit
wire ID_EX_FLUSH , ID_EX_MemRead, EX_MEM_MemRead;


wire [4:0]  ID_EX_rt;
Hazard_Unit Hazard_unit (
   .D_rs(IF_ID_rs),
```
GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
    .D_rt(IF_ID_rt),
    .EX_rt(ID_EX_rt),
    .EX_MemRead(ID_EX_MemRead),
            .ID_EX_FLUSH(ID_EX_FLUSH),
            .IF_ID_write(IF_ID_write),// we use it as enable to IF_ID REG
    .PC_write(PC_write),
    .ID_Branch(Branch),
    .MEM_rt(EX_MEM_rd),// we must implement it in EX/MEM REG
    .MEM_MemRead(EX_MEM_MemRead)

  );

// End of hazard detection Unit

// ID_EX_Pipeline_Register ---------------------------------------------

  wire [31:0] ID_EX_Reg_File_Data1, ID_EX_Reg_File_Data2,ID_EX_PC_out;
  wire [4:0] ID_EX_rs;
  wire ID_EX_ALUSrc, ID_EX_MemWrite;
  wire [3:0] ID_EX_ALUOp;
  wire [1:0]ID_EX_MemtoReg, ID_EX_RegDst;
  wire [5:0] ID_EX_func;
  wire [4:0] ID_EX_shamt;

  // Instantiate the module
  ID_EX_Register ID_EX_R (
    .clk(clk),
```

```
        .reset(ID_EX_FLUSH),
    .In_Reg_File_Data1(ReadData1),
    .In_Reg_File_Data2(ReadData2),
    .In_offset(immediate_value),
        .In_PC(IF_ID_PC_out),
    .In_Rs(IF_ID_rs),
    .In_Rt(IF_ID_rt),
    .In_Rd(IF_ID_rd),
    .In_ALUSrc(ALUSrc),
    .In_MemWrite(MemWrite),
    .In_MemRead(MemRead),
    .In_RegWrite(RegWrite),
    .In_ALUOp(ALUOp),
    .In_MemtoReg(MemtoReg),
    .In_RegDst(RegDst),
        .In_func(IF_ID_funct),
        .In_shamt(IF_ID_shamt),
    .Out_Reg_File_Data1(ID_EX_Reg_File_Data1),// may be the address of top of stack to store or load from Memory in JS or JAL Instructions
    .Out_Reg_File_Data2(ID_EX_Reg_File_Data2),
    .Out_offset(ID_EX_immediate_value),
        .Out_PC(ID_EX_PC_out),
    .Out_Rs(ID_EX_rs),
    .Out_Rt(ID_EX_rt),
    .Out_Rd(ID_EX_rd),
    .Out_ALUSrc(ID_EX_ALUSrc),
    .Out_MemWrite(ID_EX_MemWrite),
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
   .Out_MemRead(ID_EX_MemRead),
   .Out_RegWrite(ID_EX_RegWrite),
   .Out_ALUOp(ID_EX_ALUOp),
   .Out_MemtoReg(ID_EX_MemtoReg),
   .Out_RegDst(ID_EX_RegDst),
          .Out_func(ID_EX_func),
          .Out_shamt(ID_EX_shamt)
  );

// End of ID_EX_Register




//*************************************************************************************************************
***********************************

//********************************************** Execution Stage
**********************************************************************

//*************************************************************************************************************
***********************************


wire [3:0] Operation;
wire [2:0] branch_type;


alu_control alu_ctrl (
          .clk(clk),
   .FuncField(ID_EX_func),
   .ALUOp(ID_EX_ALUOp),
   .Operation(Operation),
```

```
        .branch_type(branch_type)
);

//end of alu_cntrl

// Determine RD write register

MUX5bit mux_inst (
    .a(ID_EX_rt),
    .b(ID_EX_rd),
    .select(ID_EX_RegDst), // we must take it from EX Stage
    .out(write_reg_input)
  );

// END of Determine RD Register

//ALU Forwarding Unit

        wire [1:0] ALU_Select_Forward_A ,ALU_Select_Forward_B;

        wire [31:0] Final_ALU_ReadData1;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
   wire [31:0] Final_ALU_ReadData2;// the output of forwarding MUXES


            ForwardingUnit Forwarding_ALU (
      .rs1_ID_EX(ID_EX_rs),
      .rs2_ID_EX(ID_EX_rt),
      .rd_EX_MEM(EX_MEM_rd),
      .rd_MEM_WB(MEM_WB_rd),
      .RegWrite_EX_MEM(EX_MEM_RegWrite),
      .RegWrite_MEM_WB(MEM_WB_RegWrite),
      .forwardA(ALU_Select_Forward_A),
      .forwardB(ALU_Select_Forward_B)
   );
           ///// we are here
MUX4_1 ALU_Forwarding_A_MUX(
.a(ID_EX_Reg_File_Data1),
.b(WB_Writedata),
.c(EX_MEM_ALU_Result),// EX_MEM
.select(ALU_Select_Forward_A),
.out(Final_ALU_ReadData1)
);


MUX4_1 ALU_Forwarding_B_MUX(
.a(ID_EX_Reg_File_Data2),
.b(WB_Writedata),
.c(EX_MEM_ALU_Result),
.select(ALU_Select_Forward_B),
.out(Final_ALU_ReadData2)
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
);

//-----------------------------------
// MUX2_1 alu_sec_input

wire [31:0] alu_second_input;
 MUX2_1 alu_sec_input (
       .a(Final_ALU_ReadData2),
       .b(ID_EX_immediate_value),
       .select(ID_EX_ALUSrc),
       .out(alu_second_input)
   );

//end of MUX2_1 alu_sec_input



//ALU ------------------------------------------------

wire [31:0] alu_output , Address;
wire zero ;

ALU alu (
            .clk(clk),
   .A(Final_ALU_ReadData1),
   .B(alu_second_input),
   .ALUControl(Operation),
```

```
    .ShiftAmount(ID_EX_shamt),
            .branch_type(branch_type),
    .ALUOut(alu_output),
    .Zero(zero)
);

// Determin the Final address From ALU or From Register file (Stack)
MUX2_1 Address_MUX (
        .a(alu_output),
        .b(ID_EX_Reg_File_Data2),
        .select(ID_EX_MemtoReg[1]), // we want the second choice(ReadData1) in stack operations JAL --> Store or JS---->Load
        .out(Address)
    );



//end of ALU
//-----------------------------------
// EX_MEM_Register

// Signals
  wire [31:0] EX_MEM_Write_Data;
  wire EX_MEM_MemWrite;
//ID_EX_PC_out
  // Instantiate the module
  EX_MEM_Register EX_MEM_R (
    .clk(clk),
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
        .In_Address(Address),
   .In_Write_Data(Final_ALU_ReadData2),
        .In_PC(ID_EX_PC_out),
   .In_Rd(write_reg_input),


   .In_MemWrite(ID_EX_MemWrite),
   .In_MemRead(ID_EX_MemRead),
   .In_RegWrite(ID_EX_RegWrite),
   .In_MemtoReg(ID_EX_MemtoReg),
   .Out_Address(EX_MEM_ALU_Result),
   .Out_Write_Data(EX_MEM_Write_Data),
        .Out_PC(EX_MEM_PC_out),
   .Out_Rd(EX_MEM_rd),
   .Out_MemWrite(EX_MEM_MemWrite),
   .Out_MemRead(EX_MEM_MemRead),
   .Out_RegWrite(EX_MEM_RegWrite),
   .Out_MemtoReg(EX_MEM_MemtoReg)
  );




// End of EX_MEM_Register
//----------------------------------
//*************************************************************************************************
************************************
//********************************************** Memory Stage
**********************************************************************
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
//*************************************************************************************************************
************************************
```

```
wire [31:0] Final_Data;

// Determin the Final Data From Normal operations or From PC (Stack)

MUX2_1 RAM_Data_MUX (

      .a(EX_MEM_Write_Data),

      .b(EX_MEM_PC_out),

      .select(EX_MEM_MemtoReg[1]), // we want the second choice(ReadData1) in stack operations JAL --> Store or JS----
>Load

      .out(Final_Data)

   );



// DATA MAM

wire [31:0] Read_data;


RAM #(

   .size(32),

   .data_width(32)


) ram (

   .clk(clk),// I think we must edit it maybe we do it like register file

   .reset(reset),

   .address(EX_MEM_ALU_Result),

   .data_write(Final_Data),
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
    .write_en(EX_MEM_MemWrite),
    .read_en(EX_MEM_MemRead),
    .data_out(Read_data)



);
//end of DATA_MEM
//-----------------------------------


//MEM_WB_Register


// Signals


  wire [31:0] MEM_WB_ALU_Data,MEM_WB_PC_out;



  // Instantiate the module
  MEM_WB_Register MEM_WB_R (
    .clk(clk),
    .In_RAM_Data(Read_data),
    .In_Immediate_Data(EX_MEM_ALU_Result),
            .In_PC(EX_MEM_PC_out),
    .In_Rd(EX_MEM_rd),
    .In_RegWrite(EX_MEM_RegWrite),
    .In_MemtoReg(EX_MEM_MemtoReg),
    .Out_RAM_Data(MEM_WB_RAM_Data),
    .Out_Immediate_Data(MEM_WB_ALU_Data),
```

```
            .Out_PC(MEM_WB_PC_out),
    .Out_Rd(MEM_WB_rd),
    .Out_RegWrite(MEM_WB_RegWrite),
    .Out_MemtoReg(MEM_WB_MemtoReg)
  );



// End of MEM_WB_Register


//*************************************************************************************************
*****************************************

//***************************************************** Write Back Stage
*********************************************************************

//*************************************************************************************************
*****************************************


//Write back
WB_MUX4_1 Write_back (
      .a(MEM_WB_ALU_Data),
      .b(MEM_WB_RAM_Data),
                  .c(MEM_WB_PC_out),// we will implement it in ID Stage so we need update the mem to reg control
unit and invent a new signal for it
                  .d(MEM_WB_RAM_Data),// for JS instruction to write the previous subroutine on REG 31
      .select(MEM_WB_MemtoReg),
      .out(WB_Writedata)
    );
```

## 1.35 PC

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
module PC #(
   parameter first_address = 0,
   parameter pc_inc = 4
)(
   input wire clk,
   input wire reset,
   input wire [31:0] target,
            input wire pc_load,
   output reg [31:0] pc
);

reg state = 1'b0;

always @(posedge clk or posedge reset) begin

   if (reset) begin
      pc <= 32'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
                        state=1'b0;

   end else begin
      case (state)
         1'b0: begin
            state <= 1'b1;
            pc <= first_address;
         end
         1'b1: begin
            if (pc_load) begin
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
                pc <= target;
            end
            end
            default: begin
               state <= 1'b0;
               pc <= 32'h00000000;
            end
        endcase
    end
end


endmodule
```

## 1.36 RAM

```
module RAM #(
  parameter size = 32,
  parameter data_width = 32
)(
  input clk,
  input reset,                  // Reset signal
  input [31:0] address,
  input [data_width-1:0] data_write,
  input write_en,
  input read_en,
  output wire [data_width-1:0] data_out


 // output reg error
);
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
 reg [31:0] mem [0:31];
 reg error;



 // Declare a memory array with parameterized size and data width.
// reg [data_width-1:0] mem [0:size - 1];



assign data_out=(read_en)? ((reset)?8'h00000000:mem[address >> 2]):32'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;


reg state = 1'b0;
integer i;


//
 always @(*) begin


   if (reset) begin
    for (i = 0; i < size; i = i + 1) begin
      mem[i] <= 32'b0;
    end


   end
           else if (write_en || state==1'b0) begin
      case (state)
         1'b0: begin
           state <= 1'b1;
                                    // Enter RAM Data here
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
                                        //testcaase 2
  mem[0] = 32'b00000000000000000000000000000011; // 3
  mem[1] = 32'b00000000000000000000000000000011; // 3
  mem[2] = 32'b00000000000000000000000000000011; // 3
  mem[3] = 32'b00000000000000000000000000000011; // 3
  mem[4] = 32'b00000000000000000000000000000011; // 3



          end
          1'b1: begin

           mem[address >> 2] <= data_write;

          end

       endcase
    end
end

endmodule
```

## 1.37 Register file

```
module
RegisterFile(Clock,Reset,ReadReg1,ReadReg2,WriteReg,WriteData,Reg_write_Control,ReadData1,ReadData2,PC_Store,PUSH_Stack,PULL_Stack);

input Clock , Reset;

input [4:0] ReadReg1 , ReadReg2 , WriteReg;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
input Reg_write_Control;
input PUSH_Stack,PULL_Stack,PC_Store;
input [31:0] WriteData;
output [31:0] ReadData1;
output [31:0] ReadData2;

// define bus (wires)
wire [31:0] Reg_Enable , StackData;
wire [31:0] Registers_Read [31:0];



// to get the register enable we want to write on
RegFile_decoder dex(WriteReg,Reg_write_Control,Reg_Enable);

// we first write then we read from register file

//write on Register file
RegFile_regn Reg_0(WriteData, 1'b1, Reg_Enable[0], Clock,Registers_Read[0]);
RegFile_regn Reg_1(WriteData, Reset, Reg_Enable[1], Clock,Registers_Read[1]);
RegFile_regn Reg_2(WriteData, Reset, Reg_Enable[2], Clock,Registers_Read[2]);
RegFile_regn Reg_3(WriteData, Reset, Reg_Enable[3], Clock,Registers_Read[3]);
RegFile_regn Reg_4(WriteData, Reset, Reg_Enable[4], Clock,Registers_Read[4]);
RegFile_regn Reg_5(WriteData, Reset, Reg_Enable[5], Clock,Registers_Read[5]);
RegFile_regn Reg_6(WriteData, Reset, Reg_Enable[6], Clock,Registers_Read[6]);
RegFile_regn Reg_7(WriteData, Reset, Reg_Enable[7], Clock,Registers_Read[7]);
RegFile_regn Reg_8(WriteData, Reset, Reg_Enable[8], Clock,Registers_Read[8]);
RegFile_regn Reg_9(WriteData, Reset, Reg_Enable[9], Clock,Registers_Read[9]);
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
RegFile_regn Reg_10(WriteData, Reset, Reg_Enable[10], Clock,Registers_Read[10]);
RegFile_regn Reg_11(WriteData, Reset, Reg_Enable[11], Clock,Registers_Read[11]);
RegFile_regn Reg_12(WriteData, Reset, Reg_Enable[12], Clock,Registers_Read[12]);
RegFile_regn Reg_13(WriteData, Reset, Reg_Enable[13], Clock,Registers_Read[13]);
RegFile_regn Reg_14(WriteData, Reset, Reg_Enable[14], Clock,Registers_Read[14]);
RegFile_regn Reg_15(WriteData, Reset, Reg_Enable[15], Clock,Registers_Read[15]);
RegFile_regn Reg_16(WriteData, Reset, Reg_Enable[16], Clock,Registers_Read[16]);
RegFile_regn Reg_17(WriteData, Reset, Reg_Enable[17], Clock,Registers_Read[17]);
RegFile_regn Reg_18(WriteData, Reset, Reg_Enable[18], Clock,Registers_Read[18]);
RegFile_regn Reg_19(WriteData, Reset, Reg_Enable[19], Clock,Registers_Read[19]);

RegFile_regn Reg_20(WriteData, Reset, Reg_Enable[20], Clock,Registers_Read[20]);
RegFile_regn Reg_21(WriteData, Reset, Reg_Enable[21], Clock,Registers_Read[21]);
RegFile_regn Reg_22(WriteData, Reset, Reg_Enable[22], Clock,Registers_Read[22]);
RegFile_regn Reg_23(WriteData, Reset, Reg_Enable[23], Clock,Registers_Read[23]);
RegFile_regn Reg_24(WriteData, Reset, Reg_Enable[24], Clock,Registers_Read[24]);
RegFile_regn Reg_25(WriteData, Reset, Reg_Enable[25], Clock,Registers_Read[25]);
RegFile_regn Reg_26(WriteData, Reset, Reg_Enable[26], Clock,Registers_Read[26]);
RegFile_regn Reg_27(WriteData, Reset, Reg_Enable[27], Clock,Registers_Read[27]);
RegFile_regn Reg_28(WriteData, Reset, Reg_Enable[28], Clock,Registers_Read[28]);

// customize register for hold stack pointer
Stack_regn Reg_29(StackData, Reset, PUSH_Stack || PULL_Stack , Clock,Registers_Read[29]);
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
RegFile_regn Reg_30(WriteData, Reset, Reg_Enable[30], Clock,Registers_Read[30]);



// customize register for hold the value of top of stack (Return address Register)

RegFile_regn Reg_31(WriteData, Reset, PC_Store, Clock,Registers_Read[31]);// Return address Register



Stack_Memory MIPS_Stack (PUSH_Stack,PULL_Stack,Registers_Read[29],StackData);

// Read from Register file

// MUX1: Read first operand

//always @(posedge Clock) begin

assign ReadData1= Registers_Read[ReadReg1];



//end



// MUX2: Read second operand

// in Stack Instructions we use it to carry the address of SP

assign ReadData2= (PUSH_Stack == 1'b1 || PULL_Stack == 1'b1 )? Registers_Read[29] : Registers_Read[ReadReg2];// the address that will go to RAM in JS or JAL
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

endmodule

## 1.38 Stack memory

```
module Stack_Memory(JAL_signal,JS_signal,Top_Stack_old,Top_Stack_new);


input JAL_signal,JS_signal; // to indicate if the instruction is JAL or JS


input [31:0] Top_Stack_old ;


output reg [31:0] Top_Stack_new ;



//*****size of stack********
// First entry in address Entry(22) : 88 32'h00000058
// last entry in address Entry(31) : 124 32'h0000007C
//**********************************************************



always@(*)
 begin


//


// check if the stack pointer is in allowed range otherwise I assign the fist index to him (its to avoid initial begin )
  if( Top_Stack_old < 32'h00000058  || Top_Stack_old > 32'h0000007C )
                    begin  Top_Stack_new <= 32'h00000058; end
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
  else

        begin

// JAL PUSH Operation

            if(JAL_signal == 1'b1 && Top_Stack_old != 32'h0000007C ) // check if the stack full

                begin

                    Top_Stack_new <= Top_Stack_old+4;

                end

// JS POP Operation
            else if(JS_signal == 1'b1 && Top_Stack_old != 32'h00000058 ) // check if the stack empty

                begin

                    Top_Stack_new <= Top_Stack_old-4;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
                end

            else

            begin

                Top_Stack_new <= Top_Stack_old;


            end


        end

end


endmodule
```

## 1.39 ALU control

```
module alu_control(

input clk,

input [5:0] FuncField,

input [3:0] ALUOp,
```

```
output reg [3:0] Operation,
output reg [2:0] branch_type
);



   always @(ALUOp , FuncField) begin
            if (ALUOp == 4'b0000) begin
                    Operation = 4'b0000;
                    end
                    else if(ALUOp == 4'b0010) begin
      case (FuncField)
        6'b100000: Operation = 4'b0000;  // add
        6'b100010: Operation = 4'b0001;  // sub
                                    6'b011000: Operation = 4'b0010;  //mul
                                    6'b011010: Operation = 4'b0011;  //div
                                    6'b000000: Operation = 4'b0100;  //sll
                                    6'b000010: Operation = 4'b0101;  //srl
                                    6'b100100: Operation = 4'b1000;  // and
        6'b100101: Operation = 4'b1001;  // or
                                    6'b100110: Operation = 4'b1010;  //xor
                                    6'b100111: Operation = 4'b1011;  //nor
        6'b101010: Operation = 4'b1110;  // SLT
                                    6'b100001: Operation = 4'b0110;          //addu
                                    6'b100011: Operation = 4'b0111;  //subu

        default: Operation <= 4'b0000;   // Default add operation
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
    endcase

end

        else if(ALUOp == 4'b0001)begin
                Operation = 4'b1000;  // andi
        end

         else if(ALUOp == 4'b0011)begin
                Operation = 4'b1001;  // ori
        end

        else if(ALUOp == 4'b0100)begin
                Operation = 4'b0001;  // beq
                branch_type = 3'b001;
        end

        else if(ALUOp == 4'b0101)begin
                Operation = 4'b0001;  // bne
                branch_type = 3'b010;
        end

        else if(ALUOp == 4'b0110)begin
                Operation = 4'b1110;  // bgt
                branch_type = 3'b011;
        end
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
        else if(ALUOp == 4'b0111)begin
                Operation = 4'b1101;  // blt
                branch_type = 3'b100;
        end

        else if(ALUOp == 4'b1000)begin
                Operation = 4'b1100;  // bge
                branch_type = 3'b101;
        end

        else if(ALUOp == 4'b1001)begin
                Operation = 4'b1100;  // ble
                branch_type = 3'b110;
        end









        else begin Operation = 4'b1111;
        end
        end
endmodule
```

## 1.40 Branch control

```
module branch_control(
   input wire [31:0] extended_address,
   input wire [31:0] next_pc,
```

```
   output wire [31:0] branch
);
   assign branch = (extended_address << 2) + next_pc;
endmodule
```

### 1.41 Sign_extend

```
module sign_extend (
   input wire  [15:0] extend,
   output wire [31:0] extended
);

assign    extended[15:0] = extend[15:0];
assign    extended[31:16] = {16{extend[15]}};

endmodule
```

# appendix B FPGA code

### 1.42 Main

```
module Main (
input clk,
input reset,
output   [31:0] reg0,
output   [31:0] reg1,
output   [31:0] reg2,
output   [31:0] reg3,
output   [31:0] reg4,
output   [31:0] reg5,
output   [31:0] reg6,
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
output   [31:0] reg7,
output   [31:0] reg8,
output   [31:0] reg9,
output   [31:0] reg10,
output   [31:0] reg11,
output   [31:0] reg12,
output   [31:0] reg13,
output   [31:0] reg14,
output   [31:0] reg15,
output   [31:0] reg16,
output   [31:0] reg17,
output   [31:0] reg18,
output   [31:0] reg19,
output   [31:0] reg20,
output   [31:0] reg21,
output   [31:0] reg22,
output   [31:0] reg23,
output   [31:0] reg24,
output   [31:0] reg25,
output   [31:0] reg26,
output   [31:0] reg27,
output   [31:0] reg28,
output   [31:0] reg29,
output   [31:0] reg30,
output   [31:0] reg31
);
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
// ************************************* Fetch Stage
**************************************************************************


 wire Branch_Zero_Signal ; // we will use it in pc load



//PC

 wire [31:0] pc_final;//input

 wire [31:0] pc_out;//output

 wire [31:0] next_pc;//pc+4

 wire [31:0] pc_inc;//4

 assign pc_inc = 32'b00000000000000000000000000000100;// constant value(4)

 wire PC_write;//control


 PC #(.first_address(0),  .pc_inc(4) )

 pc_inst (

   .clk(clk),

   .reset(reset),

           .target(pc_final),

            .pc_load(PC_write), //second edition come from hazard detection unit

   .pc(pc_out)

  );


//end of PC

//-----------------------------------


adder add(
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
        .a(pc_inc),
        .b(pc_out),
        .c(next_pc)
        );



//-----------------------------------
//////**********************we can update the Instruction memory and delete all output signals except the inst_out;
//inst_mem
wire[31:0] inst_out;

INST_MEM #(.size(32),.data_width(32) )
inst_mem (
   .reset(reset),
   .address(pc_out),
   .inst_out(inst_out)
 );

 // End of Instruction Memory ---------------------------------------

// IF_ID_Register


  // Outputs
  wire [31:0] IF_ID_Instruction_out;
  wire [31:0] IF_ID_PC_out;
```
GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
// to split it from Instruction memory signal


wire [5:0] IF_ID_opcode;

wire [4:0] IF_ID_rs;

wire [4:0] IF_ID_rt;

wire [4:0] IF_ID_rd;

wire [4:0] IF_ID_shamt;

wire [5:0] IF_ID_funct;

wire [15:0] IF_ID_addrs;

wire [25:0] IF_ID_jump_offset;

 wire [1:0] Jump_signal;//Jump_signal[0] really jump signal

wire IF_ID_write ,Branch ; // the enable signal that come from hazard unit and branch signal from control Unit


IF_ID_Register IF_ID_R (

   .clk(clk),

   .reset(reset),

   .enable(~(IF_ID_write)),//  it designed to be negative because the case of first instruction when no instruction is in ID stage

   .Instruction_in(inst_out), // the output instruction from Instruction Memory

   .PC_in(next_pc),

   .Branch_Control((Branch & Branch_Zero_Signal)|(Jump_signal[0] | Jump_signal[1]) ), // if we catch branch depandancy

     //or we find jump or jump and link instructions or JS --> Jump Register

   .Instruction_out(IF_ID_Instruction_out),

   .PC_out(IF_ID_PC_out),// Maybe I must implement the pc in every register but know I will not do it

           .opcode(IF_ID_opcode),

   .rs(IF_ID_rs),
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
   .rt(IF_ID_rt),
   .rd(IF_ID_rd),
   .shamt(IF_ID_shamt),
   .funct(IF_ID_funct),
   .addr(IF_ID_addrs),// use for calculate the branch target address
            .jump(IF_ID_jump_offset)
  );

// End of

//************************************************ Decode Stage
**********************************************************************



//sign extend
wire [31:0] immediate_value;
wire [31:0] ID_EX_immediate_value;

sign_extend extender (
   .extend(IF_ID_addrs),
   .extended(immediate_value)
);

//end of sign extend
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
//-----------------------------------
//ControlUnit
  wire ALUSrc, MemWrite, MemRead, RegWrite;
  wire [3:0] ALUOp;
  wire [1:0] RegDst;
  wire [1:0] MemtoReg;

  // we don't need pc_load and pc_store signal anymore (useless) (clear phase)

ControlUnit control_inst (
    .Clock(clk),
    .Reset(reset),
    .opcode(IF_ID_opcode),
    .RegDst(RegDst),
    .ALUSrc(ALUSrc),
    .MemtoReg(MemtoReg),
    .MemWrite(MemWrite),
    .MemRead(MemRead),
    .RegWrite(RegWrite),
    .ALUOp(ALUOp),
            .Branch(Branch),
            .Jump(Jump_signal),
            .funct(IF_ID_funct)

  );
//end of ControlUnit
//-----------------------------------
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
// Reg_File


   wire [4:0] write_reg_input;


   wire [31:0] WB_Writedata; // we use it to hold the data from wb to register file to write it in addition we use it in forwarding Unit

        wire [4:0] MEM_WB_rd;

   wire MEM_WB_RegWrite;


        wire [31:0] ReadData1;

   wire [31:0] ReadData2;


        wire [1:0] MEM_WB_MemtoReg;// we use it in implementation of JAL Instruction as enable to register 31




          RegisterFile reg_file_inst (

      .Clock(clk),// we make the the register file write on the negative edge so we can write on a register and read his data in same clock

      .Reset(reset),

      .ReadReg1(IF_ID_rs),

      .ReadReg2(IF_ID_rt),

      .WriteReg(MEM_WB_rd),

      .Reg_write_Control(MEM_WB_RegWrite),// we must take it from WB Stage

      .WriteData(WB_Writedata),
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

.ReadData1(ReadData1),

.ReadData2(ReadData2),

.PC_Store(MEM_WB_MemtoReg[1]),// to indicate JAL OR JS Instruction in WB Stage to store the Return address that come from RAM

// must update in pull operation

.PUSH_Stack(MemtoReg[1] && ~MemtoReg[0]),// to indicate JAL Instruction with Stack

.PULL_Stack(Jump_signal[1]), // to indicate JS Instruction with Stack

//this is added by moayyad

.reg0(reg0),

.reg1(reg1),

.reg2(reg2),

.reg3(reg3),

.reg4(reg4),

.reg5(reg5),

.reg6(reg6),

.reg7(reg7),

.reg8(reg8),

.reg9(reg9),

.reg10(reg10),

.reg11(reg11),

.reg12(reg12),

.reg13(reg13),

.reg14(reg14),

.reg15(reg15),

.reg16(reg16),

.reg17(reg17),

```
                .reg18(reg18),
                .reg19(reg19),
                .reg20(reg20),
                .reg21(reg21),
                .reg22(reg22),
                .reg23(reg23),
                .reg24(reg24),
                .reg25(reg25),
                .reg26(reg26),
                .reg27(reg27),
                .reg28(reg28),
                .reg29(reg29),
                .reg30(reg30),
                .reg31(reg31)
    );


        wire [31:0] Branch_address;


// End of Register File ----------------------------------------------------


// implement the branch forwarding Unit


        wire [4:0] EX_MEM_rd,ID_EX_rd; // we need to use the rd in branch forwarding;
        wire ID_EX_RegWrite , EX_MEM_RegWrite;
        wire [1:0] Branch_Select_Forward_A ,Branch_Select_Forward_B;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
        wire [31:0] EX_MEM_ALU_Result ;
        wire [31:0] MEM_WB_RAM_Data;

        wire [31:0] Final_Branch_ReadData1;
  wire [31:0] Final_Branch_ReadData2;// the output of forwarding MUXES

         ForwardingUnit Forwarding_Branch (
    .rs1_ID_EX(IF_ID_rs),
    .rs2_ID_EX(IF_ID_rt),
    .rd_EX_MEM(EX_MEM_rd),
    .rd_MEM_WB(MEM_WB_rd),
    .RegWrite_EX_MEM(EX_MEM_RegWrite),
    .RegWrite_MEM_WB(MEM_WB_RegWrite),
    .forwardA(Branch_Select_Forward_A),
    .forwardB(Branch_Select_Forward_B)
  );

MUX4_1 Branch_Forwarding_A_MUX(
.a(ReadData1),
.b(WB_Writedata),
.c(EX_MEM_ALU_Result),// EX_MEM
.select(Branch_Select_Forward_A),
.out(Final_Branch_ReadData1)
);

MUX4_1 Branch_Forwarding_B_MUX(
.a(ReadData2),
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
.b(WB_Writedata),
.c(EX_MEM_ALU_Result),
.select(Branch_Select_Forward_B),
.out(Final_Branch_ReadData2)
);



        // implement branch and jump unit with PC MUXES


        Branch Branch_Unit (
   .Branch_Flag(Branch),
   .ALUOp(ALUOp),
   .Data1(Final_Branch_ReadData1),// There is a  forwarding on this signal
   .Data2(Final_Branch_ReadData2),// There is a  forwarding on this signal
   .Target(IF_ID_addrs),
   .next_pc(IF_ID_PC_out),
   .Branch_address(Branch_address),
   .zero(Branch_Zero_Signal)
 );



// JUMP (JS) Forwarding Unit



// implement the branch forwarding Unit


        wire JUMP_Select_Forward_A;
```

```
        wire [1:0] EX_MEM_MemtoReg;
        wire [31:0] Final_JUMP_ReadData;
        wire [31:0] EX_MEM_PC_out;

 ForwardingUnit_JUMP JUMP_Forwarding (
   .JS_JUMP(Jump_signal),
   .EX_MEM_MemtoReg(EX_MEM_MemtoReg),
   . forwardA(JUMP_Select_Forward_A)

);

MUX2_1 JUMP_Forwarding_MUX(
.a(ReadData1),
.b(EX_MEM_PC_out),
.select(JUMP_Select_Forward_A),
.out(Final_JUMP_ReadData)
);

  // Calculate PC Target Address
  wire [31:0] pc_branch , JUMP_Target_address;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
MUX2_1 pc_target(
.a(next_pc),
.b(Branch_address),
.select((Branch & Branch_Zero_Signal)),
.out(pc_branch)
);



// implement jump here

JUMP Jump_Unit (
   .JUMP_FLAG(Jump_signal[0]),
   .jump_offset(IF_ID_jump_offset),
            .next_pc(next_pc),
   .JUMP_address(JUMP_Target_address)
  );

MUX4_1 pc_final_main(
.a(pc_branch),
.b(JUMP_Target_address),// I don't know where is he
.c(Final_JUMP_ReadData),// must be the output from JS Instruction Forwarding MUX
.select(Jump_signal),
.out(pc_final)
);
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
// end branch and Jump implementation


// Hazard dediction Unit
wire ID_EX_FLUSH , ID_EX_MemRead, EX_MEM_MemRead;


wire [4:0]  ID_EX_rt;
Hazard_Unit Hazard_unit (
   .D_rs(IF_ID_rs),
   .D_rt(IF_ID_rt),
   .EX_rt(ID_EX_rt),
   .EX_MemRead(ID_EX_MemRead),
          .ID_EX_FLUSH(ID_EX_FLUSH),
          .IF_ID_write(IF_ID_write),// we use it as enable to IF_ID REG
   .PC_write(PC_write),
   .ID_Branch(Branch),
   .MEM_rt(EX_MEM_rd),// we must implement it in EX/MEM REG
   .MEM_MemRead(EX_MEM_MemRead)

 );

// End of hazard detection Unit

// ID_EX_Pipeline_Register ----------------------------------------------

  wire [31:0] ID_EX_Reg_File_Data1, ID_EX_Reg_File_Data2,ID_EX_PC_out;
  wire [4:0] ID_EX_rs;
  wire ID_EX_ALUSrc, ID_EX_MemWrite;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
 wire [3:0] ID_EX_ALUOp;
 wire [1:0]ID_EX_MemtoReg, ID_EX_RegDst;
 wire [5:0] ID_EX_func;
 wire [4:0] ID_EX_shamt;

 // Instantiate the module
 ID_EX_Register ID_EX_R (
  .clk(clk),
        .reset(ID_EX_FLUSH),
  .In_Reg_File_Data1(ReadData1),
  .In_Reg_File_Data2(ReadData2),
  .In_offset(immediate_value),
        .In_PC(IF_ID_PC_out),
  .In_Rs(IF_ID_rs),
  .In_Rt(IF_ID_rt),
  .In_Rd(IF_ID_rd),
  .In_ALUSrc(ALUSrc),
  .In_MemWrite(MemWrite),
  .In_MemRead(MemRead),
  .In_RegWrite(RegWrite),
  .In_ALUOp(ALUOp),
  .In_MemtoReg(MemtoReg),
  .In_RegDst(RegDst),
        .In_func(IF_ID_funct),
        .In_shamt(IF_ID_shamt),
   .Out_Reg_File_Data1(ID_EX_Reg_File_Data1),// may be the address of top of stack to store or load from Memory in JS or JAL Instructions
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
   .Out_Reg_File_Data2(ID_EX_Reg_File_Data2),
   .Out_offset(ID_EX_immediate_value),
            .Out_PC(ID_EX_PC_out),
   .Out_Rs(ID_EX_rs),
   .Out_Rt(ID_EX_rt),
   .Out_Rd(ID_EX_rd),
   .Out_ALUSrc(ID_EX_ALUSrc),
   .Out_MemWrite(ID_EX_MemWrite),
   .Out_MemRead(ID_EX_MemRead),
   .Out_RegWrite(ID_EX_RegWrite),
   .Out_ALUOp(ID_EX_ALUOp),
   .Out_MemtoReg(ID_EX_MemtoReg),
   .Out_RegDst(ID_EX_RegDst),
            .Out_func(ID_EX_func),
            .Out_shamt(ID_EX_shamt)
  );

// End of ID_EX_Register




//********************************************** Execution Stage
**********************************************************************



wire [3:0] Operation;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
wire [2:0] branch_type;


alu_control alu_ctrl (
            .clk(clk),
   .FuncField(ID_EX_func),
   .ALUOp(ID_EX_ALUOp),
   .Operation(Operation),
            .branch_type(branch_type)
);


//end of alu_cntrl


// Determine RD write register


MUX5bit mux_inst (
      .a(ID_EX_rt),
      .b(ID_EX_rd),
      .select(ID_EX_RegDst), // we must take it from EX Stage
      .out(write_reg_input)
   );


// END of Determine RD Register
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
//ALU Forwarding Unit



        wire [1:0] ALU_Select_Forward_A ,ALU_Select_Forward_B;



        wire [31:0] Final_ALU_ReadData1;
   wire [31:0] Final_ALU_ReadData2;// the output of forwarding MUXES


         ForwardingUnit Forwarding_ALU (
      .rs1_ID_EX(ID_EX_rs),
      .rs2_ID_EX(ID_EX_rt),
      .rd_EX_MEM(EX_MEM_rd),
      .rd_MEM_WB(MEM_WB_rd),
      .RegWrite_EX_MEM(EX_MEM_RegWrite),
      .RegWrite_MEM_WB(MEM_WB_RegWrite),
      .forwardA(ALU_Select_Forward_A),
      .forwardB(ALU_Select_Forward_B)
   );
        ///// we are here
MUX4_1 ALU_Forwarding_A_MUX(
.a(ID_EX_Reg_File_Data1),
.b(WB_Writedata),
.c(EX_MEM_ALU_Result),// EX_MEM
.select(ALU_Select_Forward_A),
.out(Final_ALU_ReadData1)
);
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
MUX4_1 ALU_Forwarding_B_MUX(
.a(ID_EX_Reg_File_Data2),
.b(WB_Writedata),
.c(EX_MEM_ALU_Result),
.select(ALU_Select_Forward_B),
.out(Final_ALU_ReadData2)
);

//-----------------------------------
// MUX2_1 alu_sec_input

wire [31:0] alu_second_input;
 MUX2_1 alu_sec_input (
       .a(Final_ALU_ReadData2),
       .b(ID_EX_immediate_value),
       .select(ID_EX_ALUSrc),
       .out(alu_second_input)
   );

//end of MUX2_1 alu_sec_input



//ALU ------------------------------------------------

wire [31:0] alu_output , Address;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
wire zero ;


ALU alu (
            .clk(clk),
   .A(Final_ALU_ReadData1),
   .B(alu_second_input),
   .ALUControl(Operation),
   .ShiftAmount(ID_EX_shamt),
            .branch_type(branch_type),
   .ALUOut(alu_output),
   .Zero(zero)
);


// Determin the Final address From ALU or From Register file (Stack)
MUX2_1 Address_MUX (
      .a(alu_output),
      .b(ID_EX_Reg_File_Data2),
      .select(ID_EX_MemtoReg[1]), // we want the second choice(ReadData1) in stack operations JAL --> Store or JS----
>Load
      .out(Address)
   );



//end of ALU
//-----------------------------------
// EX_MEM_Register
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
// Signals
  wire [31:0] EX_MEM_Write_Data;
  wire EX_MEM_MemWrite;
//ID_EX_PC_out
  // Instantiate the module
  EX_MEM_Register EX_MEM_R (
    .clk(clk),
            .In_Address(Address),
    .In_Write_Data(Final_ALU_ReadData2),
            .In_PC(ID_EX_PC_out),
    .In_Rd(write_reg_input),

    .In_MemWrite(ID_EX_MemWrite),
    .In_MemRead(ID_EX_MemRead),
    .In_RegWrite(ID_EX_RegWrite),
    .In_MemtoReg(ID_EX_MemtoReg),
    .Out_Address(EX_MEM_ALU_Result),
    .Out_Write_Data(EX_MEM_Write_Data),
            .Out_PC(EX_MEM_PC_out),
    .Out_Rd(EX_MEM_rd),
    .Out_MemWrite(EX_MEM_MemWrite),
    .Out_MemRead(EX_MEM_MemRead),
    .Out_RegWrite(EX_MEM_RegWrite),
    .Out_MemtoReg(EX_MEM_MemtoReg)
  );
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
// End of EX_MEM_Register
//-----------------------------------


//********************************************** Memory Stage
**************************************************************************


wire [31:0] Final_Data;
// Determin the Final Data From Normal operations or From PC (Stack)
MUX2_1 RAM_Data_MUX (
      .a(EX_MEM_Write_Data),
      .b(EX_MEM_PC_out),
      .select(EX_MEM_MemtoReg[1]), // we want the second choice(ReadData1) in stack operations JAL --> Store or JS----
>Load
      .out(Final_Data)
   );



// DATA MAM
wire [31:0] Read_data;


RAM #(
   .size(32),
   .data_width(32)

) ram (
   .clk(clk),// I think we must edit it maybe we do it like register file
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
    .reset(reset),
    .address(EX_MEM_ALU_Result),
    .data_write(Final_Data),
    .write_en(EX_MEM_MemWrite),
    .read_en(EX_MEM_MemRead),
    .data_out(Read_data)



);
//end of DATA_MEM
//-------------------------------------


//MEM_WB_Register


// Signals


  wire [31:0] MEM_WB_ALU_Data,MEM_WB_PC_out;



  // Instantiate the module
  MEM_WB_Register MEM_WB_R (
    .clk(clk),
    .In_RAM_Data(Read_data),
    .In_Immediate_Data(EX_MEM_ALU_Result),
            .In_PC(EX_MEM_PC_out),
    .In_Rd(EX_MEM_rd),
    .In_RegWrite(EX_MEM_RegWrite),
```

```
   .In_MemtoReg(EX_MEM_MemtoReg),
   .Out_RAM_Data(MEM_WB_RAM_Data),
   .Out_Immediate_Data(MEM_WB_ALU_Data),
           .Out_PC(MEM_WB_PC_out),
   .Out_Rd(MEM_WB_rd),
   .Out_RegWrite(MEM_WB_RegWrite),
   .Out_MemtoReg(MEM_WB_MemtoReg)
  );



// End of MEM_WB_Register


//************************************************** Write Back Stage
**************************************************************************



//Write back
WB_MUX4_1 Write_back (
      .a(MEM_WB_ALU_Data),
      .b(MEM_WB_RAM_Data),
                  .c(MEM_WB_PC_out),// we will implement it in ID Stage so we need update the mem to reg control
unit and invent a new signal for it
                  .d(MEM_WB_RAM_Data),// for JS instruction to write the previous subroutine on REG 31
      .select(MEM_WB_MemtoReg),
      .out(WB_Writedata)
   );
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

//----------

//********** END MAIN **********

Endmodule

## 1.43 FPGA

```
module FBGA(
  input clk,
  input en,
  input [4:0] mux32,
  input selector,
  input reset,
  output reg [6:0] digit_1,
  output reg [6:0] digit_2,
  output reg [6:0] digit_3,
  output reg [6:0] digit_4,
  output reg [6:0] digit_5,
  output reg [6:0] digit_6

);
reg [31:0] show_reg;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
wire [31:0] All_Registers [0:31];



 Main m_dut(
                        .clk(clk),
                        .reset(1'b0),
                        .reg0(All_Registers[0]),
                        .reg1(All_Registers[1]),
                        .reg2(All_Registers[2]),
                        .reg3(All_Registers[3]),
                        .reg4(All_Registers[4]),
                        .reg5(All_Registers[5]),
                        .reg6(All_Registers[6]),
                        .reg7(All_Registers[7]),
                        .reg8(All_Registers[8]),
                        .reg9(All_Registers[9]),
                        .reg10(All_Registers[10]),
                        .reg11(All_Registers[11]),
                        .reg12(All_Registers[12]),
                        .reg13(All_Registers[13]),
                        .reg14(All_Registers[14]),
                        .reg15(All_Registers[15]),
                        .reg16(All_Registers[16]),
                        .reg17(All_Registers[17]),
                        .reg18(All_Registers[18]),
                        .reg19(All_Registers[19]),
                        .reg20(All_Registers[20]),
```

```
                .reg21(All_Registers[21]),
                .reg22(All_Registers[22]),
                .reg23(All_Registers[23]),
                .reg24(All_Registers[24]),
                .reg25(All_Registers[25]),
                .reg26(All_Registers[26]),
                .reg27(All_Registers[27]),
                .reg28(All_Registers[28]),
                .reg29(All_Registers[29]),
                .reg30(All_Registers[30]),
                .reg31(All_Registers[31])
        );



//*************************************************************************
        reg [39:0] counter; // Counter to control the delay
//      reg [39:0] counter2;
initial counter = 0;
//initial counter2= 0;
always @(posedge clk) begin
   if (~en) begin
            counter <= counter + 1;
     if (counter < 100000000) begin
       // First sentence
       digit_6 <= 7'b1100001;
       digit_5 <= 7'b1000001;
       digit_4 <= 7'b0010010;
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
        digit_3 <= 7'b0000111;
      end else if (counter >= 100000000 && counter < 200000000) begin
   // Second sentence after 1 second delay
   digit_6 <= 7'b1000110;
   digit_5 <= 7'b0001100;
   digit_4 <= 7'b1000001;
   digit_3 <= 7'b1111111;
end else if (counter >= 200000000 && counter < 300000000) begin
   // Third sentence after another 1 second delay
   digit_6 <= 7'b0000011;
   digit_5 <= 7'b0101111;
   digit_4 <= 7'b0100011;
   digit_3 <= 7'b1111111;
end


                    else begin
                    counter <= 40'h0;
                    end
   end
   else begin


            case(mux32)
          5'b00000:begin show_reg <= All_Registers[0]; digit_5 <= 7'b1000000;digit_6 <= 7'b1000000; end
          5'b00001:begin show_reg <= All_Registers[1]; digit_5 <= 7'b1111001;digit_6 <= 7'b1000000; end
          5'b00010:begin show_reg <= All_Registers[2]; digit_5 <= 7'b0100100;digit_6 <= 7'b1000000; end
          5'b00011:begin show_reg <= All_Registers[3]; digit_5 <= 7'b0110000;digit_6 <= 7'b1000000; end
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
5'b00100:begin show_reg <= All_Registers[4]; digit_5 <= 7'b0011001;digit_6 <= 7'b1000000; end
5'b00101:begin show_reg <= All_Registers[5] ;digit_5 <= 7'b0010010;digit_6 <= 7'b1000000; end
5'b00110:begin show_reg <= All_Registers[6] ;digit_5 <= 7'b0000010;digit_6 <= 7'b1000000; end
5'b00111:begin show_reg <= All_Registers[7] ;digit_5 <= 7'b1111000;digit_6 <= 7'b1000000; end
5'b01000:begin show_reg <= All_Registers[8] ;digit_5 <= 7'b0000000;digit_6 <= 7'b1000000; end
5'b01001:begin show_reg <= All_Registers[9] ;digit_5 <= 7'b0011000;digit_6 <= 7'b1000000; end
5'b01010:begin show_reg <= All_Registers[10] ;digit_5 <= 7'b1000000;digit_6 <= 7'b1111001; end
5'b01011:begin show_reg <= All_Registers[11] ;digit_5 <= 7'b1111001;digit_6 <= 7'b1111001; end
5'b01100:begin show_reg <= All_Registers[12] ;digit_5 <= 7'b0100100;digit_6 <= 7'b1111001; end
5'b01101:begin show_reg <= All_Registers[13] ;digit_5 <= 7'b0110000;digit_6 <= 7'b1111001; end
5'b01110:begin show_reg <= All_Registers[14] ;digit_5 <= 7'b0011001;digit_6 <= 7'b1111001; end
5'b01111:begin show_reg <= All_Registers[15] ;digit_5 <= 7'b0010010;digit_6 <= 7'b1111001; end
5'b10000:begin show_reg <= All_Registers[16] ;digit_5 <= 7'b0000010;digit_6 <= 7'b1111001; end
5'b10001:begin show_reg <= All_Registers[17] ;digit_5 <= 7'b1111000;digit_6 <= 7'b1111001; end
5'b10010:begin show_reg <= All_Registers[18] ;digit_5 <= 7'b0000000;digit_6 <= 7'b1111001; end
5'b10011:begin show_reg <= All_Registers[19] ;digit_5 <= 7'b0011000;digit_6 <= 7'b1111001; end
5'b10100:begin show_reg <= All_Registers[20] ;digit_5 <= 7'b1000000;digit_6 <= 7'b0100100; end
5'b10101:begin show_reg <= All_Registers[21] ;digit_5 <= 7'b1111001;digit_6 <= 7'b0100100; end
5'b10110:begin show_reg <= All_Registers[22] ;digit_5 <= 7'b0100100;digit_6 <= 7'b0100100; end
5'b10111:begin show_reg <= All_Registers[23] ;digit_5 <= 7'b0110000;digit_6 <= 7'b0100100; end
5'b11000:begin show_reg <= All_Registers[24] ;digit_5 <= 7'b0011001;digit_6 <= 7'b0100100; end
5'b11001:begin show_reg <= All_Registers[25] ;digit_5 <= 7'b0010010;digit_6 <= 7'b0100100; end
5'b11010:begin show_reg <= All_Registers[26] ;digit_5 <= 7'b0000010;digit_6 <= 7'b0100100; end
5'b11011:begin show_reg <= All_Registers[27] ;digit_5 <= 7'b1111000;digit_6 <= 7'b0100100; end
5'b11100:begin show_reg <= All_Registers[28] ;digit_5 <= 7'b0000000;digit_6 <= 7'b0100100; end
5'b11101:begin show_reg <= All_Registers[29] ;digit_5 <= 7'b0011000;digit_6 <= 7'b0100100; end
5'b11110:begin show_reg <= All_Registers[30] ;digit_5 <= 7'b1000000;digit_6 <= 7'b0110000; end
```

```
        5'b11111:begin show_reg <= All_Registers[31] ;digit_5 <= 7'b1111001;digit_6 <= 7'b0110000; end

        endcase

//          show_reg<=32'h0000000B;
                    if(~selector) begin
   case (show_reg[3:0])
     4'b0000: digit_1 <= 7'b1000000; // 0
     4'b0001: digit_1 <= 7'b1111001; // 1
     4'b0010: digit_1 <= 7'b0100100; // 2
     4'b0011: digit_1 <= 7'b0110000; // 3
                    4'b0100: digit_1 <= 7'b0011001; // 4
     4'b0101: digit_1 <= 7'b0010010; // 5
                    4'b0110: digit_1 <= 7'b0000010; // 6
     4'b0111: digit_1 <= 7'b1111000; // 7
                    4'b1000: digit_1 <= 7'b0000000; // 8
     4'b1001: digit_1 <= 7'b0011000; // 9
                    4'b1010: digit_1 <= 7'b0001000; // A
     4'b1011: digit_1 <= 7'b0000011; // B
                    4'b1100: digit_1 <= 7'b1000110; // C
     4'b1101: digit_1 <= 7'b0100001; // D
                    4'b1110: digit_1 <= 7'b0000110; // E
     4'b1111: digit_1 <= 7'b0001110; // F
                    default: digit_1 <= 7'b0000100; // error: e
   endcase
                   //***************
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
                case (show_reg[7:4])
    4'b0000: digit_2 <= 7'b1000000; // 0
    4'b0001: digit_2 <= 7'b1111001; // 1
    4'b0010: digit_2 <= 7'b0100100; // 2
    4'b0011: digit_2 <= 7'b0110000; // 3
                4'b0100: digit_2 <= 7'b0011001; // 4
    4'b0101: digit_2 <= 7'b0010010; // 5
                4'b0110: digit_2 <= 7'b0000010; // 6
    4'b0111: digit_2 <= 7'b1111000; // 7
                4'b1000: digit_2 <= 7'b0000000; // 8
    4'b1001: digit_2 <= 7'b0011000; // 9
                4'b1010: digit_2 <= 7'b0001000; // A
    4'b1011: digit_2 <= 7'b0000011; // B
                4'b1100: digit_2 <= 7'b1000110; // C
    4'b1101: digit_2 <= 7'b0100001; // D
                4'b1110: digit_2 <= 7'b0000110; // E
    4'b1111: digit_2 <= 7'b0001110; // F
                default: digit_2 <= 7'b0000100; // Default: Display e
  endcase


                case (show_reg[11:8])
    4'b0000: digit_3 <= 7'b1000000; // 0
    4'b0001: digit_3 <= 7'b1111001; // 1
    4'b0010: digit_3 <= 7'b0100100; // 2
    4'b0011: digit_3 <= 7'b0110000; // 3
                4'b0100: digit_3 <= 7'b0011001; // 4
    4'b0101: digit_3 <= 7'b0010010; // 5
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
                4'b0110: digit_3 <= 7'b0000010; // 6
    4'b0111: digit_3 <= 7'b1111000; // 7
                4'b1000: digit_3 <= 7'b0000000; // 8
    4'b1001: digit_3 <= 7'b0011000; // 9
                4'b1010: digit_3 <= 7'b0001000; // A
    4'b1011: digit_3 <= 7'b0000011; // B
                4'b1100: digit_3 <= 7'b1000110; // C
    4'b1101: digit_3 <= 7'b0100001; // D
                4'b1110: digit_3 <= 7'b0000110; // E
    4'b1111: digit_3 <= 7'b0001110; // F
                default: digit_3 <= 7'b0000100; // Default: Display e
  endcase


      case (show_reg[15:12])
    4'b0000: digit_4 <= 7'b1000000; // 0
    4'b0001: digit_4 <= 7'b1111001; // 1
    4'b0010: digit_4 <= 7'b0100100; // 2
    4'b0011: digit_4 <= 7'b0110000; // 3
                4'b0100: digit_4 <= 7'b0011001; // 4
    4'b0101: digit_4 <= 7'b0010010; // 5
                4'b0110: digit_4 <= 7'b0000010; // 6
    4'b0111: digit_4 <= 7'b1111000; // 7
                4'b1000: digit_4 <= 7'b0000000; // 8
    4'b1001: digit_4 <= 7'b0011000; // 9
                4'b1010: digit_4 <= 7'b0001000; // A
    4'b1011: digit_4 <= 7'b0000011; // B
                4'b1100: digit_4 <= 7'b1000110; // C
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
    4'b1101: digit_4 <= 7'b0100001; // D
                4'b1110: digit_4 <= 7'b0000110; // E
    4'b1111: digit_4 <= 7'b0001110; // F
                default: digit_4 <= 7'b0000100; // Default: Display e
  endcase
        end

        else begin
          case (show_reg[19:16])
                4'bxxxx: digit_1 <= 7'b0000100;
    4'b0000: digit_1 <= 7'b1000000;  // 0
    4'b0001: digit_1 <= 7'b1111001;  // 1
    4'b0010: digit_1 <= 7'b0100100;  // 2
    4'b0011: digit_1 <= 7'b0110000; // 3
                4'b0100: digit_1 <= 7'b0011001; // 4
    4'b0101: digit_1 <= 7'b0010010; // 5
                4'b0110: digit_1 <= 7'b0000010; // 6
    4'b0111: digit_1 <= 7'b1111000; // 7
                4'b1000: digit_1 <= 7'b0000000; // 8
    4'b1001: digit_1 <= 7'b0011000; // 9
                4'b1010: digit_1 <= 7'b0001000; // A
    4'b1011: digit_1 <= 7'b0000011; // B
                4'b1100: digit_1 <= 7'b1000110; // C
    4'b1101: digit_1 <= 7'b0100001; // D
                4'b1110: digit_1 <= 7'b0000110; // E
    4'b1111: digit_1 <= 7'b0001110; // F
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
                default: digit_1 <= 7'b0000100; // Default: Display e
    endcase



                //****************
                case (show_reg[23:20])
                 4'bxxxx: digit_2 <= 7'b0000100;
    4'b0000: digit_2 <= 7'b1000000; // 0
    4'b0001: digit_2 <= 7'b1111001; // 1
    4'b0010: digit_2 <= 7'b0100100; // 2
    4'b0011: digit_2 <= 7'b0110000; // 3
                 4'b0100: digit_2 <= 7'b0011001; // 4
    4'b0101: digit_2 <= 7'b0010010; // 5
                 4'b0110: digit_2 <= 7'b0000010; // 6
    4'b0111: digit_2 <= 7'b1111000; // 7
                 4'b1000: digit_2 <= 7'b0000000; // 8
    4'b1001: digit_2 <= 7'b0011000; // 9
                 4'b1010: digit_2 <= 7'b0001000; // A
    4'b1011: digit_2 <= 7'b0000011; // B
                 4'b1100: digit_2 <= 7'b1000110; // C
    4'b1101: digit_2 <= 7'b0100001; // D
                 4'b1110: digit_2 <= 7'b0000110; // E
    4'b1111: digit_2 <= 7'b0001110; // F
                 4'bxxxx: digit_2 <= 7'b0000100;
                 default: digit_2 <= 7'b0000100; // Default: Display e
    endcase
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
                    case (show_reg[27:24])
            4'bxxxx: digit_3 <= 7'b0000100;
        4'b0000: digit_3 <= 7'b1000000; // 0
        4'b0001: digit_3 <= 7'b1111001; // 1
        4'b0010: digit_3 <= 7'b0100100; // 2
        4'b0011: digit_3 <= 7'b0110000; // 3
                    4'b0100: digit_3 <= 7'b0011001; // 4
        4'b0101: digit_3 <= 7'b0010010; // 5
                    4'b0110: digit_3 <= 7'b0000010; // 6
        4'b0111: digit_3 <= 7'b1111000; // 7
                    4'b1000: digit_3 <= 7'b0000000; // 8
        4'b1001: digit_3 <= 7'b0011000; // 9
                    4'b1010: digit_3 <= 7'b0001000; // A
        4'b1011: digit_3 <= 7'b0000011; // B
                    4'b1100: digit_3 <= 7'b1000110; // C
        4'b1101: digit_3 <= 7'b0100001; // D
                    4'b1110: digit_3 <= 7'b0000110; // E
        4'b1111: digit_3 <= 7'b0001110; // F
                    4'bxxxx: digit_3 <= 7'b0000100;
                    default: digit_3 <= 7'b0000100; // Default: Display e
    endcase


            case (show_reg[31:28])
        4'bxxxx: digit_4 <= 7'b0000100;
                    4'b0000: digit_4 <= 7'b1000000; // 0000
        4'b0001: digit_4 <= 7'b1111001; // 1
        4'b0010: digit_4 <= 7'b0100100; // 2
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
    4'b0011: digit_4 <= 7'b0110000; // 3
                4'b0100: digit_4 <= 7'b0011001; // 4
    4'b0101: digit_4 <= 7'b0010010; // 5
                4'b0110: digit_4 <= 7'b0000010; // 6
    4'b0111: digit_4 <= 7'b1111000; // 7
                4'b1000: digit_4 <= 7'b0000000; // 8
    4'b1001: digit_4 <= 7'b0011000; // 9
                4'b1010: digit_4 <= 7'b0001000; // A
    4'b1011: digit_4 <= 7'b0000011; // B
                4'b1100: digit_4 <= 7'b1000110; // C
    4'b1101: digit_4 <= 7'b0100001; // D
                4'b1110: digit_4 <= 7'b0000110; // E
    4'b1111: digit_4 <= 7'b0001110; // F
                4'bxxxx: digit_4 <= 7'b0000100;
                default: digit_4 <= 7'b0000100; // Default: Display e
  endcase
        end









        end



end
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS

```
//**************************************************************



endmodule

/*

                    7'b1000000; // 0

    7'b1111001; // 1

    7'b0100100; // 2

    7'b0110000; // 3

    7'b0011001; // 4

    7'b0010010; // 5

    7'b0000010; // 6

    7'b1111000; // 7

    7'b0000000; // 8

                    7;b0011000; // 9

                    7'b0001000; // A

                    7;b0000011; // b

                    7'b1000110; // C

                    7'b0100001; // d

                    7'b0000110; // E

                    7'b0001110; // F
```

GitHub link → https://github.com/OmarAl-Saleh/MIPS