|  |
| --- |
|  |

Jordan National Semiconductor Design Competition (JOSDC’2023)

MIPS single cycle 32 bit

By

Omar AL-khasawneh & Omar Salah & Moayyad Abu Mallouh

Amman, Jordan

October 2023



Acknowledgments

It is with immense gratitude that I acknowledge the support and guidance of Professor Mohammad A. Alshboul during the intricate process of designing the single cycle 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.

Beyond the technical aspects, Professor Alshboul's dedication to nurturing creativity and fostering a spirit of inquiry has been truly inspirational. He has not only been a mentor but also a beacon of encouragement, instilling in me a passion for design and a commitment to excellence.

To Professor Alshboul, 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 for being the cornerstone of this achievement.

# Abstract

This project presents the design and implementation of a MIPS-based single-cycle Central Processing Unit (CPU) without branch functionality, utilizing Verilog as the primary hardware description language. The decision to exclude branch operations aims to simplify the architecture, providing an accessible model for educational purposes and foundational exploration into CPU design. The implementation focuses on core MIPS functionalities, such as arithmetic, logic, and memory operations. Through the use of Verilog, the project offers a clear and modular representation of the CPU components, facilitating understanding and potential scalability. Preliminary tests indicate that the design successfully emulates the expected operations of a MIPS single-cycle CPU, establishing it as a viable tool for educational demonstrations and further research into computer architecture. Future work may include the incorporation of the branch mechanism and the transition to a pipelined architecture for performance enhancement.

In the course of the project, special emphasis was placed on ensuring the accuracy and fidelity of the Verilog representations to the theoretical MIPS architecture. The CPU was systematically verified against a suite of benchmark programs to ensure its reliability and functionality. Additionally, the design prioritizes modularity, enabling easy future expansions or modifications. The project not only serves as a pedagogical tool but also as a foundation for more complex architectural explorations. Insights gained from this endeavor highlight the potential for creating more intricate designs, emphasizing the versatility and robustness of Verilog in modeling and simulating hardware systems.

**Table of Contents**

[Abstract iii](#_Toc149588473)

[Introduction 7](#_Toc149588474)

[1.1 Introduction 7](#_Toc149588475)

[1.2 The Importance of the Design 7](#_Toc149588476)

[1.3 Motivation 7](#_Toc149588477)

[1.4 Why This Topic is Important for Students 7](#_Toc149588478)

[1.5 Objectives of the Project: 8](#_Toc149588479)

[1.6 Description of Design Achieved: 8](#_Toc149588480)

[1.7 Design Requirements: 9](#_Toc149588481)

[1.8 The teams member responsibility 9](#_Toc149588482)

[1.8.1 Omar AL-khasawneh (Leader) 10](#_Toc149588483)

[1.8.2 Omar AL-salah 10](#_Toc149588484)

[1.8.3 Moayyad Abu Mallouh 10](#_Toc149588485)

[1.9 Organization of the rest of the documentation. 11](#_Toc149588486)

[Design 13](#_Toc149588487)

[1.10 Hardware Design and Implementation 13](#_Toc149588488)

[1.10.1 Component 13](#_Toc149588489)

[1.11 mips instruction execution phase 15](#_Toc149588490)

[1.12 Coding and Software Development 15](#_Toc149588491)

[1.12.1 Tools 15](#_Toc149588492)

[1.12.2 Challenging 16](#_Toc149588494)

[1.12.3 Biggest challenge 16](#_Toc149588495)

[1.12.4 Instruction set architecture 19](#_Toc149588496)

[Results 20](#_Toc149588497)

[1.13 Program flow 20](#_Toc149588498)

[**1.14** **Clock testing** 30](#_Toc149588517)

[Conclusion 31](#_Toc149588518)

[1.14.1 Key Findings and Implications 31](#_Toc149588519)

[1.14.2 Project’s objectives achieved. 31](#_Toc149588520)

[Appendix A: CODE 32](#_Toc149588521)

# Introduction

## 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 single-cycle CPU, akin to the revered MIPS (Microprocessor without Interlocked Pipeline Stages), 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.

## 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.

## 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.

## 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.

## Objectives of the Project:

1. **Single-Cycle CPU Design:** The primary objective of this project is to design and implement a single-cycle CPU, heavily inspired by the MIPS architecture. This CPU will be capable of executing a set of fundamental instructions in a single 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.

## Description of Design Achieved:

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

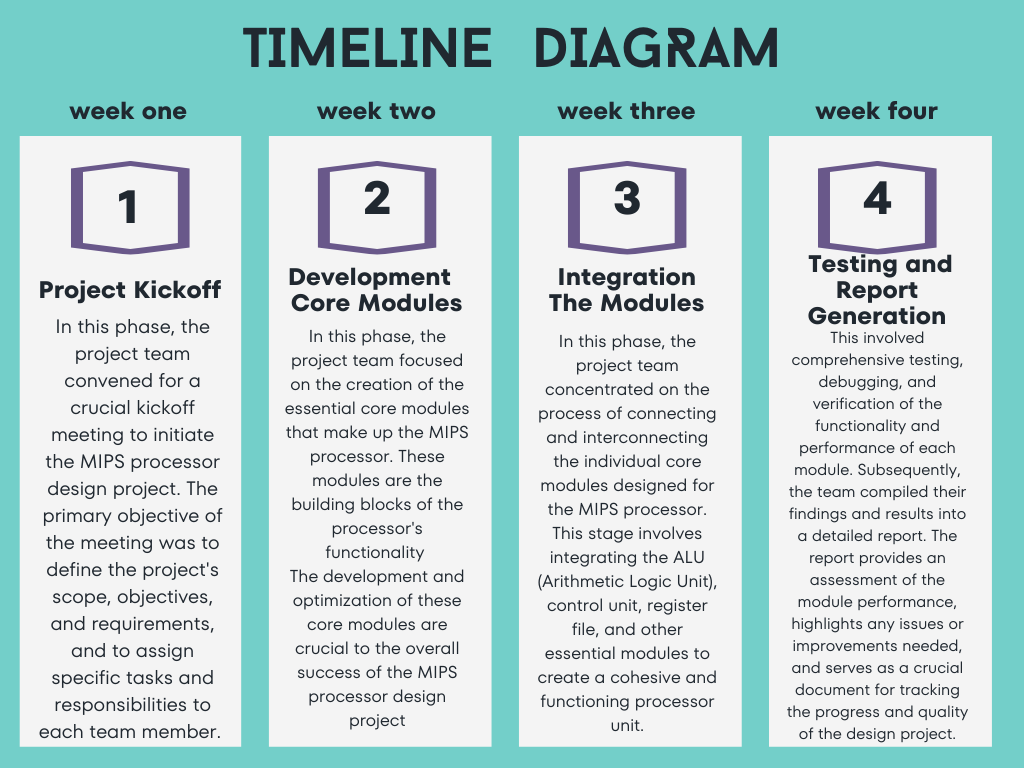
* **MIPS-Inspired Architecture:** Our CPU design is heavily influenced by the MIPS architecture, renowned for its simplicity and elegance. This architecture served as a valuable reference point in shaping our CPU's instruction set, control unit, and data path.
* **RISC (Reduced Instruction Set Computer) 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.
* **Single-Cycle Execution:** Our CPU is designed to execute instructions in a single clock cycle. This efficient one-cycle process involves fetching instructions, decoding them, executing operations, and writing results back to registers.
* **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.

## 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.
* **One-Cycle Execution:** All instructions should be executed in a single clock cycle, reflecting the core concept of a single-cycle CPU.
* **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.

## The team’s member responsibility



### Omar AL-khasawneh (Leader)

* ALU design
* ALU control design
* PC design
* Writing report

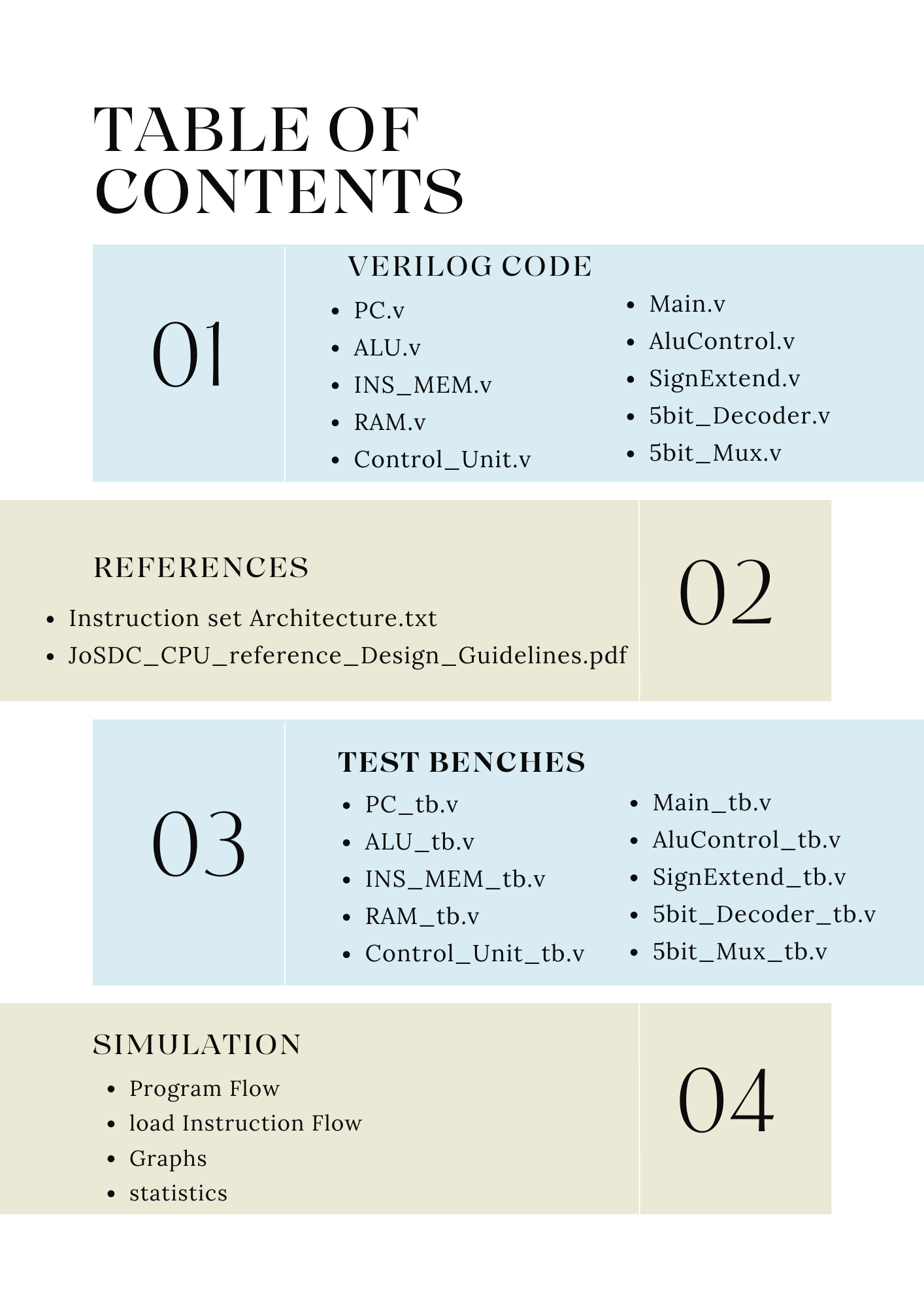
### Omar AL-salah

* Register file
* Control unit
* Make graph and image

### Moayyad Abu Mallouh

* Instruction memory
* Data memory
* Mux between component

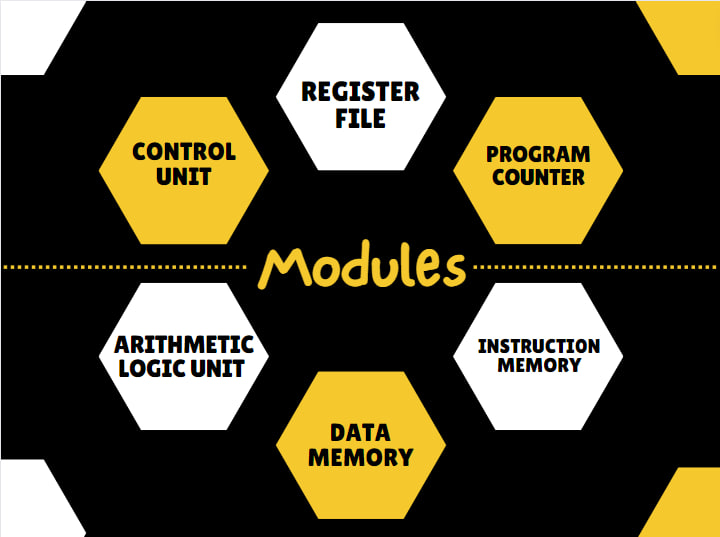
## Organization of the rest of the documentation.



# Design

## Hardware Design and Implementation

We are used this component in our project:



* Describe the hardware design and components used.

### 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.

**Arithmetic logic unit:** 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.

**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 .**

**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.**

Our design extends beyond the conventional MIPS Register File by incorporating additional signals to tailor the functionality to our specific requirements:

**Reset Signal:**

The Reset signal serves as a mechanism to reset all registers to a don’t care (rubbish value) when activated. This feature ensures a clean and controlled initialization, contributing to the reliability and predictability of the system.

**PC\_Store Signal:**

The PC\_Store signal is a custom addition to our Register File module, specifically designed to interact with register number 31. This signal, initiated by the control unit, enables register 31 to store a value from the write bus. The stored value represents the program counter value and plays a crucial role in the execution of instructions that involve branching, such as Jump And Link and Jump And Link Register.

In addition to these custom signals, we have implemented special-purpose registers to enhance the versatility and efficiency of the Register File:

**First Register (Register Number 0):**

This register is designed to always maintain a value of zero. It serves as a constant reference point and is exclusively readable. No instruction is permitted to write to this register, ensuring its consistency as a zero value register.

**Last Register (Register Number 31):**

Register 31 has been specially configured to serve a unique purpose in our design. It is enabled by the PC\_Store signal from the control unit, allowing it to store the program counter value. This functionality is essential for instructions like Jump And Link and Jump And Link Register, where the program counter value is stored in register 31, facilitating the management of subroutine calls and returns.

By incorporating these features, our Register File module not only adheres to MIPS standards but also offers enhanced functionality and customization tailored to the specific requirements of our CPU design

**Control unit:** s 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 input 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, and 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.

**Pc\_load Signal:**

The Pc\_load signal plays a pivotal role in the program counter's behavior. When activated, it disables the program counter from updating, effectively holding its current value. This functionality is useful in scenarios where a specific program counter value needs to be retained without progression.

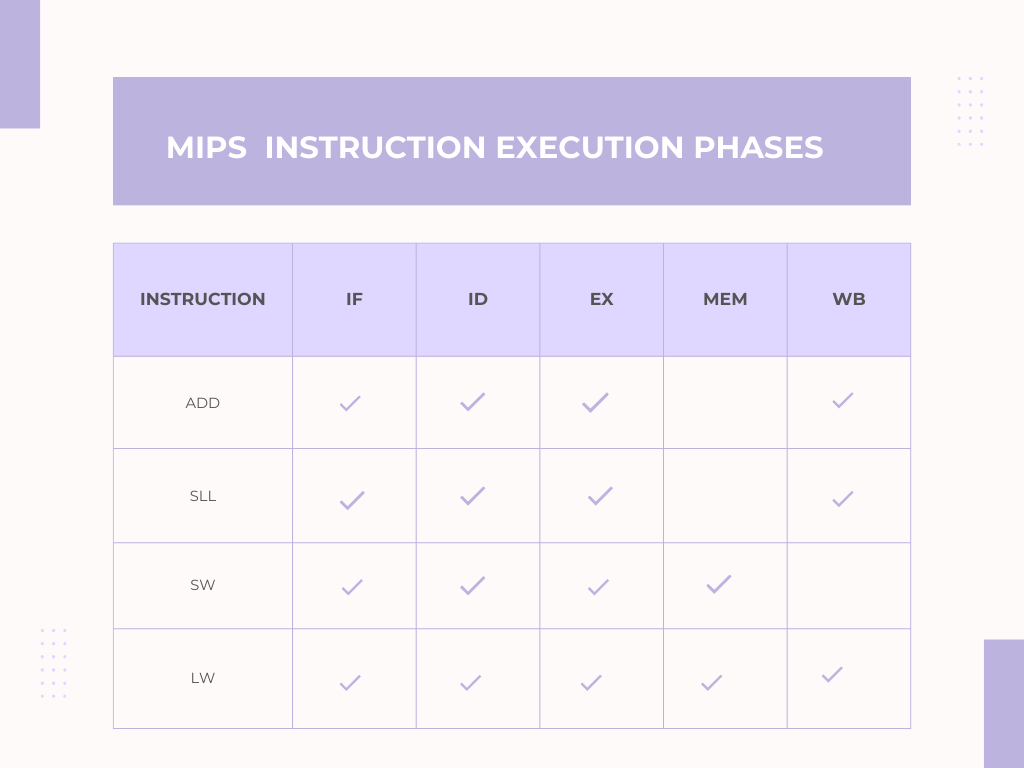
**PC\_Store Signal:**

The PC\_Store signal is a unique addition designed to interact with the register file. Its purpose is to enable the register associated with register 31 to store a specific value. This value corresponds to the current program counter value that we intend to preserve. This feature allows for strategic storage of program counter values for future reference or manipulation.

By incorporating these new signals, our control unit not only adheres to established MIPS standards but also offers a level of customization that enhances the adaptability and efficiency of our CPU design. The combination of standard and bespoke control signals ensures a versatile

and finely-tuned control mechanism, making our CPU well-suited for a variety of computing tasks.

## mips instruction execution phase



## Coding and Software Development

### 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.

### 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.

### Biggest challenge

**Timing Violation in MIPS 32-bit Architecture Single Cycle CPU**

A timing violation in a MIPS 32-bit architecture single cycle CPU occurs when a signal does not arrive at its destination within the required time window. This can happen for a number of reasons, including:

* Clock skew: Clock skew is the variation in the arrival time of the clock signal at different parts of the CPU. This can be caused by differences in the length of the clock wires or by variations in the manufacturing process.
* Hold time violation: A hold time violation occurs when a signal does not remain stable for the required amount of time before the clock edge. This can be caused by a slow signal driver or by a long signal path.
* Setup time violation: A setup time violation occurs when a signal does not arrive at its destination within the required amount of time before the clock edge. This can be caused by a fast signal driver or by a short signal path.
* Glitches: A glitch is a short-duration, spurious pulse that can occur on a signal line. Glitches can be caused by noise or by crosstalk between signal lines.

Types of Timing Violations

There are two main types of timing violations:

* 1. Combinational timing violations: Combinational timing violations occur in combinational logic circuits, such as adders and multipliers. These circuits are designed to produce an output signal within a certain amount of time after the input signals arrive. If the input signals do not arrive within the required time window, or if the circuit is not designed properly, the output signal may not be correct.
  2. Sequential timing violations: Sequential timing violations occur in sequential logic circuits, such as registers and flip-flops. These circuits are designed to store a state signal and then update it on the next clock edge. If the clock edge arrives before the state signal has settled, or if the circuit is not designed properly, the state signal may not be updated correctly.

Consequences of Timing Violations

Timing violations can lead to a number of problems, including:

Incorrect results: Timing violations can cause the CPU to produce incorrect results. This can lead to errors in programs and data corruption.

System instability: Timing violations can cause the CPU to become unstable. This can lead to system crashes and other problems.

Increased power consumption: Timing violations can cause the CPU to consume more power. This can lead to overheating and reduced battery life.

Preventing Timing Violations

There are a number of things that can be done to prevent timing violations, including:

Careful design: The CPU should be designed carefully to minimize clock skew and signal path delays.

Use of buffers: Buffers can be used to reduce signal path delays.

Use of clock gating: Clock gating can be used to disable the clock to unused circuits, which can reduce power consumption and improve timing performance.

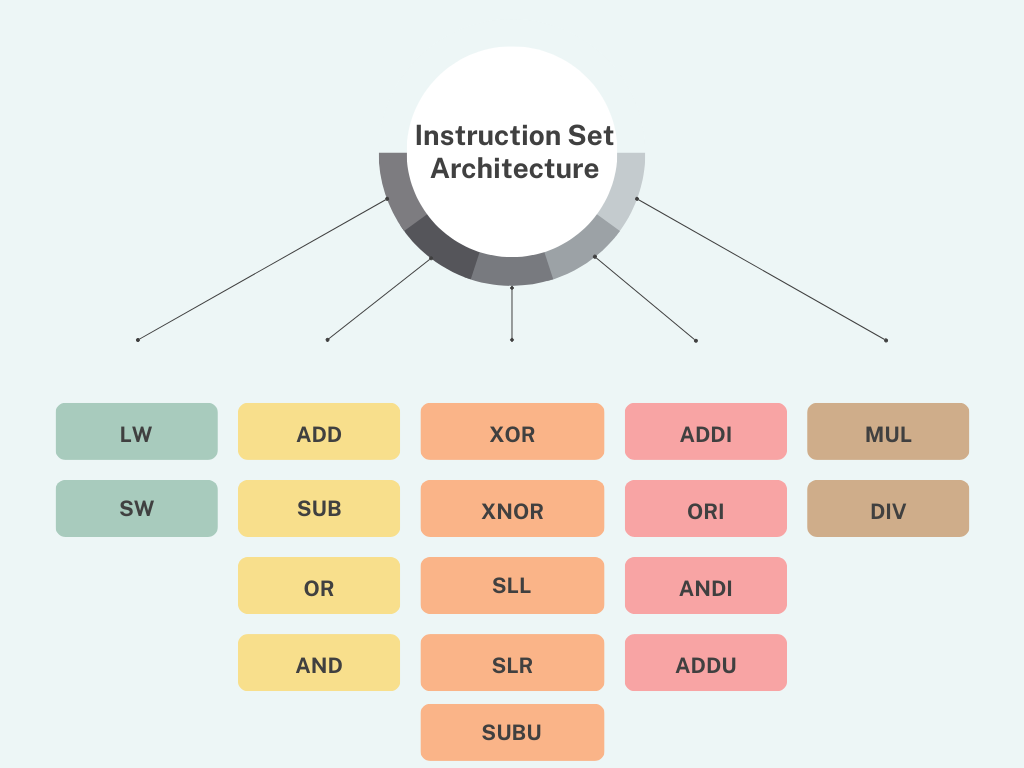
Testing: The CPU should be thoroughly tested to ensure that it meets all timing requirements.

Conclusion

Timing violations are a serious problem in MIPS 32-bit architecture single cycle CPUs. They can lead to incorrect results, system instability, and increased power 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.

### Instruction set architecture

The our instruction set architecture, it will improve in the future.

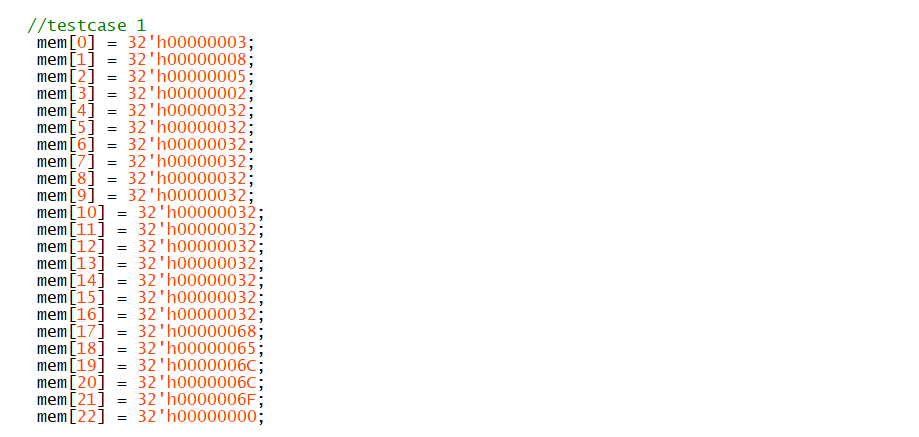
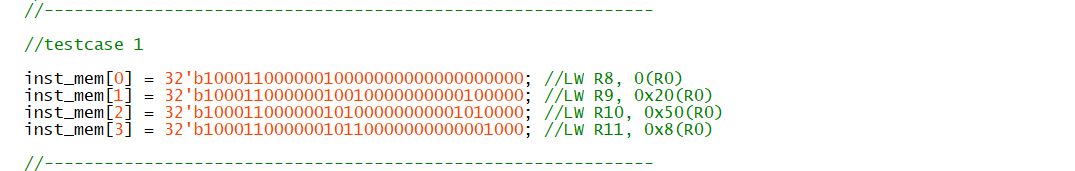


# Results

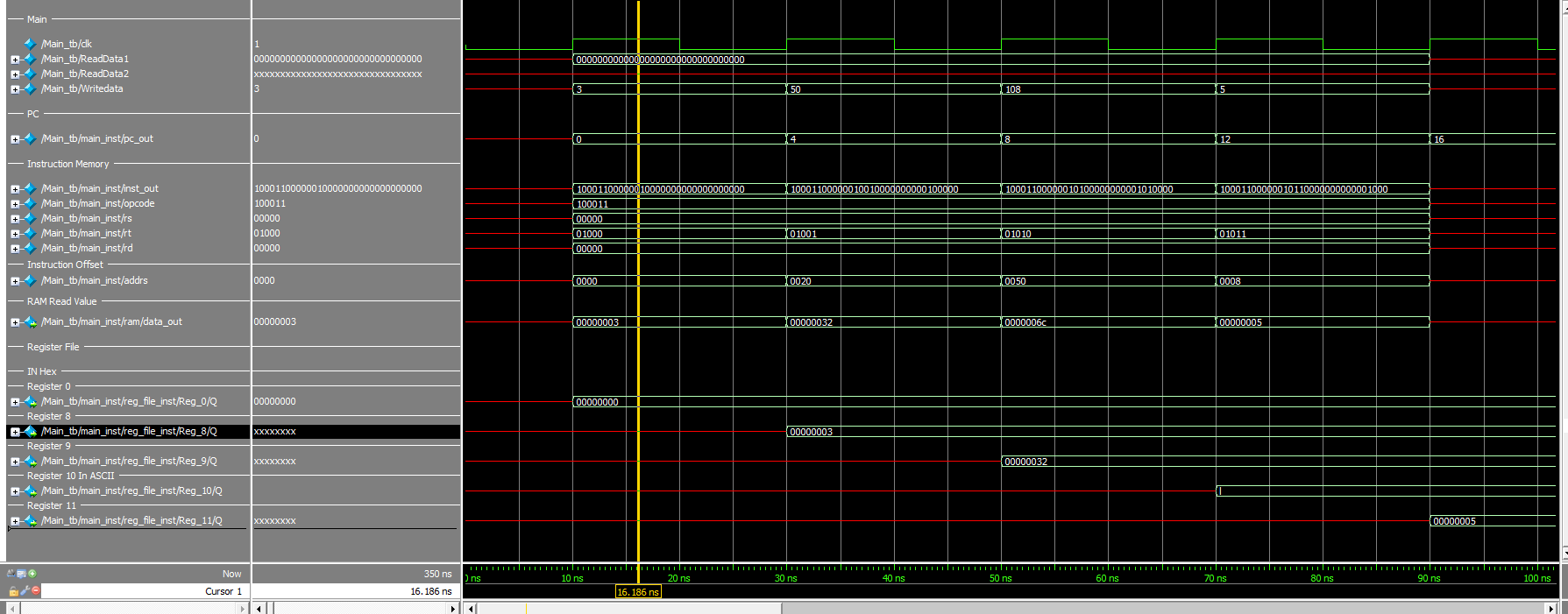
## Benchmark

### **Test case one**

#### Instruction memory and data memory



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

  
  
  
pc\_out=> current value of pc

Pc\_next=> the next pc

Data\_out=> read data from RAM

We have 32 registers , we only show the register the related to instruction .

The value save in register or memory in next positive edge .

The image contains Verilog code that appears to be initializing an instruction memory array with a set of instructions, denoted as **inst\_mem**. Each line of the code assigns a 32-bit binary value to a different index in the **inst\_mem** array, which corresponds to a machine instruction. The comments at the end of each line provide a human-readable interpretation of each instruction.

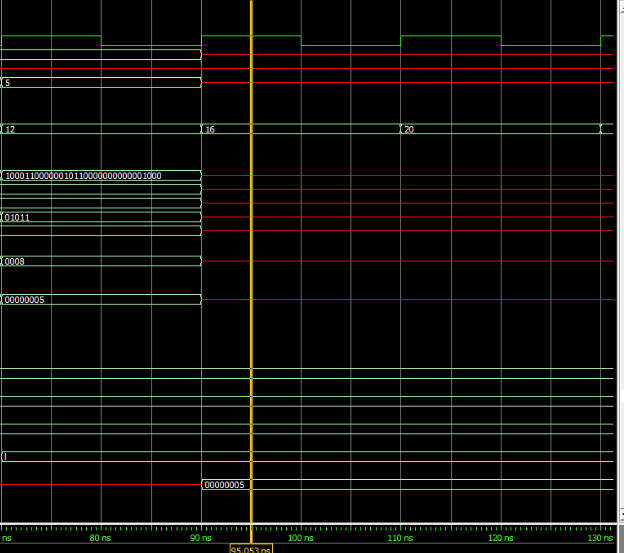
Here is a description of the instructions as per the comments:

* **inst\_mem[0]**: This instruction loads a word into register 8 from the memory address contained in register 0 (R0) with no offset. The comment **//LW R8, 0(R0)** stands for "Load Word" into R8 from the address **0 + (contents of R0)**.
* **inst\_mem[1]**: This instruction loads a word into register 9 from the memory address contained in register 0 (R0) with an offset of 0x20. The comment **//LW R9, 0x20(R0)** means "Load Word" into R9 from the address **0x20 + (contents of R0)**.
* **inst\_mem[2]**: This line loads a word into register 10 from the memory address contained in register 0 (R0) with an offset of 0x50. The comment **//LW R10, 0x50(R0)** signifies "Load Word" into R10 from the address **0x50 + (contents of R0)**.
* **inst\_mem[3]**: This instruction loads a word into register 11 from the memory address contained in register 0 (R0) with an offset of 0x80. The comment **//LW R11, 0x80(R0)** indicates "Load Word" into R11 from the address **0x80 + (contents of R0)**.

Each **inst\_mem** entry is a 32-bit binary value representing the opcode and the operand(s) for the corresponding load instruction. The opcode determines the type of operation to be performed, and the operands specify the registers and memory addresses involved in the operation. The binary values are likely encoded according to a specific instruction set architecture, which dictates the format of the binary instruction (such as opcode, source register, destination register, and immediate values or offsets).

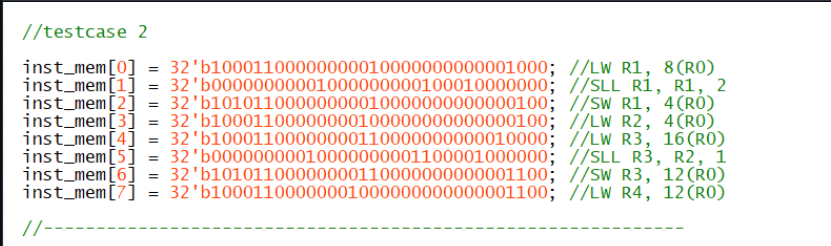
#### Third instruction LW R10, 0x50(r0) after execution the final values

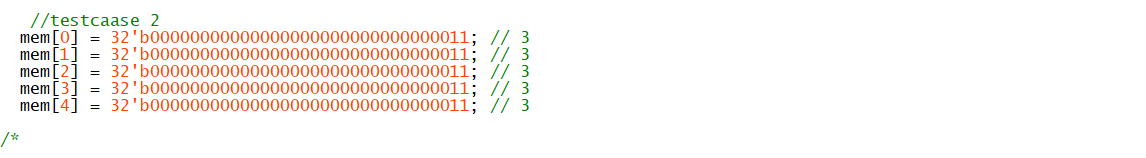
A screenshot of a computer

Description automatically generated

## **Test case 2**

### Data memory and instruction memory





### First instruction LW R1 ,8(R0)

**pc\_out=> current value of pc**

**Pc\_next=> the next pc**

**Data\_out=> read data from RAM**

**We have 32 registers , we only show the register the related to instruction .**

**The value save in register or memory in next positive edge .**

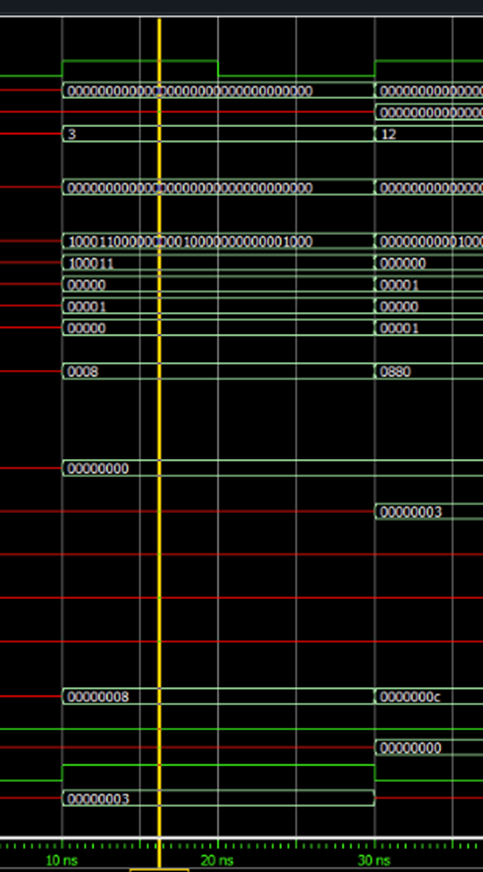
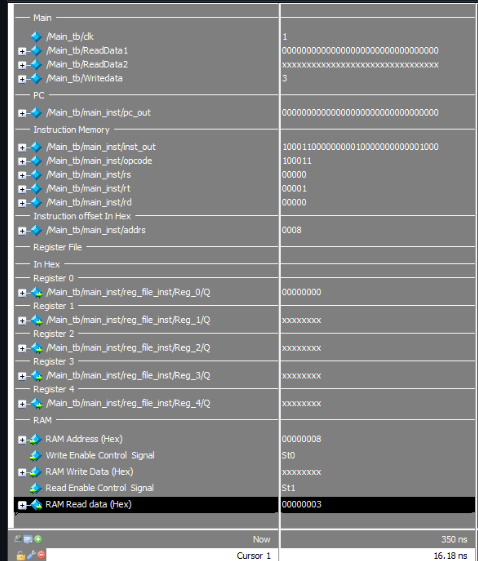
**The image contains a segment of Verilog code which is setting up a test case (testcase 2) by initializing an array named inst\_mem with a series of 32-bit binary instructions. These instructions are likely meant to be used in a simulated CPU or instruction set architecture environment. Each line of the array corresponds to a machine instruction, and the comments at the end of each line describe the assembly language equivalent of these machine instructions.**

**Here's a description of the instructions:**

* **inst\_mem[0]: Load word into register R1 from the address computed by adding 8 to the value in register R0 (LW R1, 8(R0)).**
* **inst\_mem[1]: Shift left logical by 2 positions the value in register R1 and store the result back in R1 (SLL R1, R1, 2).**
* **inst\_mem[2]: Store word from register R1 into the memory address computed by adding 4 to the value in register R0 (SW R1, 4(R0)).**
* **inst\_mem[3]: Load word into register R2 from the address computed by adding 4 to the value in register R0 (LW R2, 4(R0)).**
* **inst\_mem[4]: Load word into register R3 from the address computed by adding 16 to the value in register R0 (LW R3, 16(R0)).**
* **inst\_mem[5]: Shift left logical by 1 position the value in register R3 and store the result back in R2 (SLL R3, R2, 1).**
* **inst\_mem[6]: Store word from register R3 into the memory address computed by adding 12 to the value in register R0 (SW R3, 12(R0)).**
* **inst\_mem[7]: Load word into register R4 from the address computed by adding 12 to the value in register R0 (LW R4, 12(R0)).**

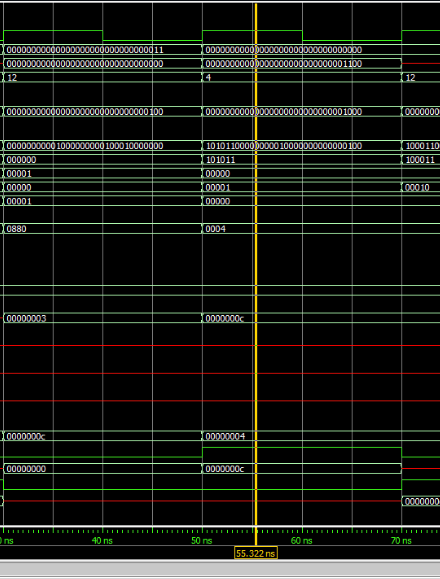
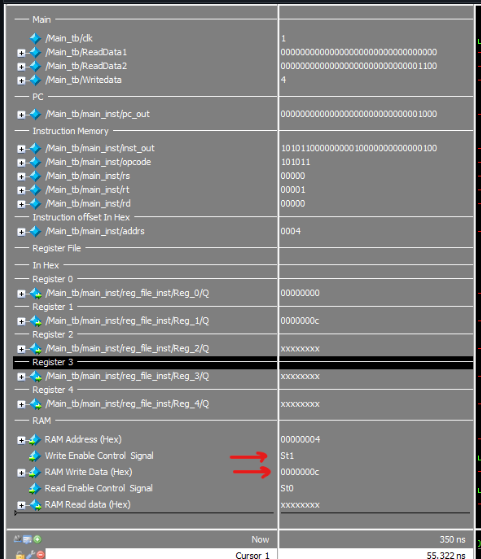
**Each instruction is encoded in a binary format where the opcode and the operands are represented by specific bits within the 32-bit instruction word. The exact format of these bits would be determined by the instruction set architecture (ISA) that the simulated CPU follows.**

**This set of instructions seems to perform a sequence of loads, stores, and shifts, which could be part of a larger algorithm or function being tested. The LW and SW instructions are for loading and storing data from memory, while SLL is a shift operation that modifies data within the registers. These are typical instructions found in many ISAs.**

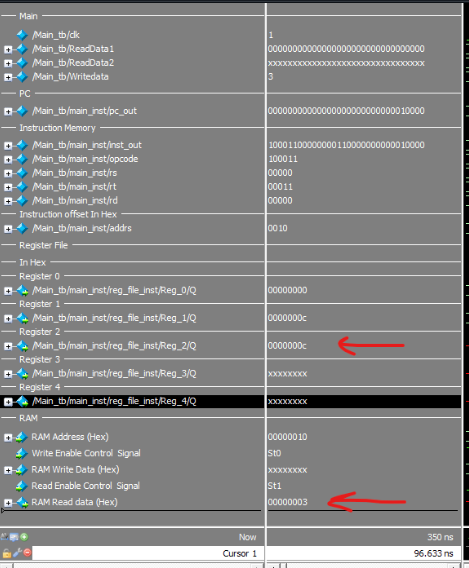


### Third instruction SW R1, 4(R0)

Please look careful for read color in the image



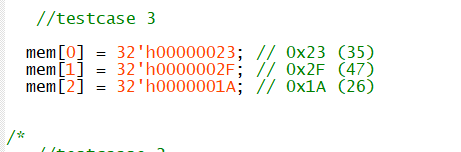
### fifth Instruction LW R3, 16(R0) with the result of previous instruction in R2 LW R2, 4(R0)

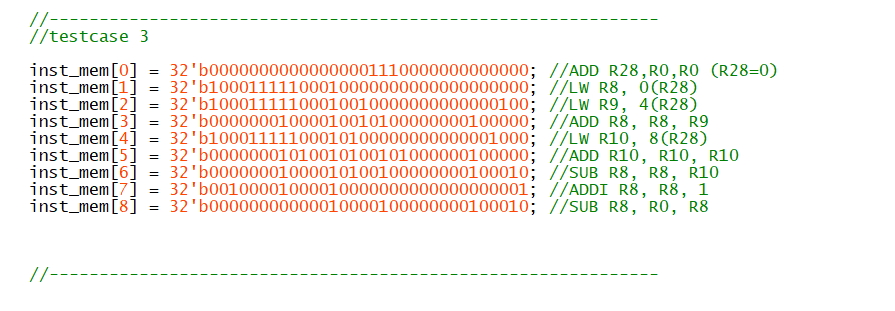


Very important the change in the next positive edge

## **Test case 3**

### Instruction memory and data memory





### First Instruction (ADD R28,R0,R0

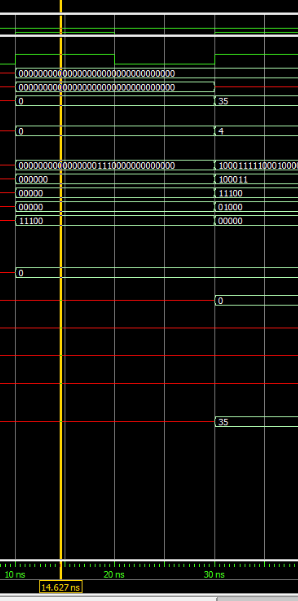
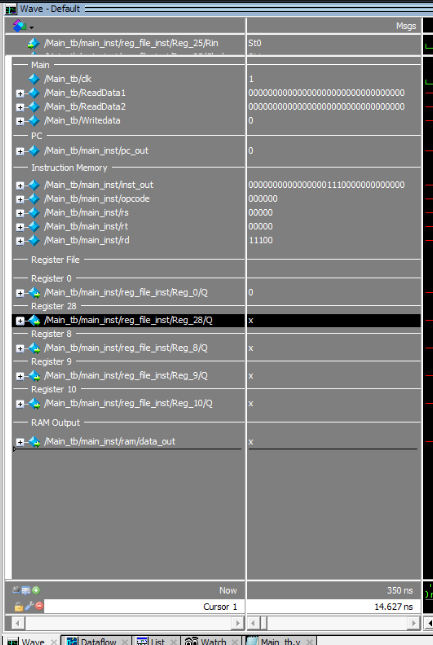
**pc\_out=> current value of pc**

**Pc\_next=> the next pc**

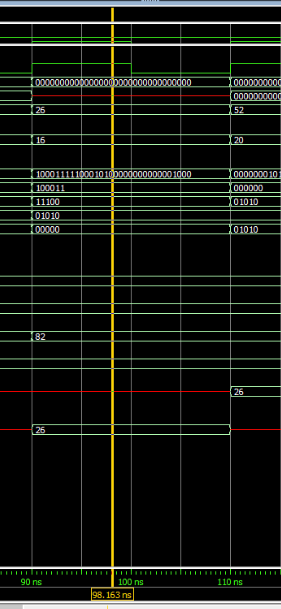
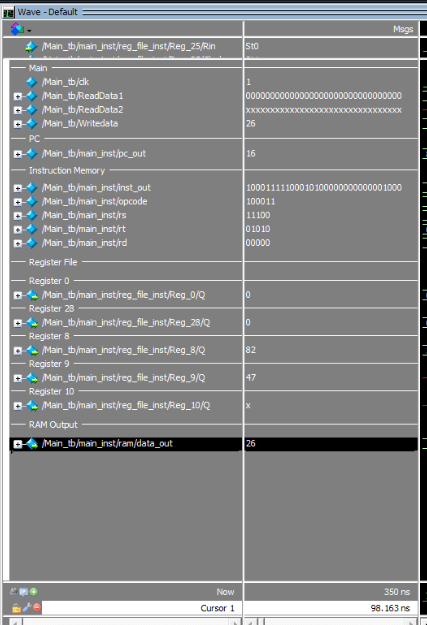
**Data\_out=> read data from RAM**

**We have 32 registers , we only show the register the related to instruction .**

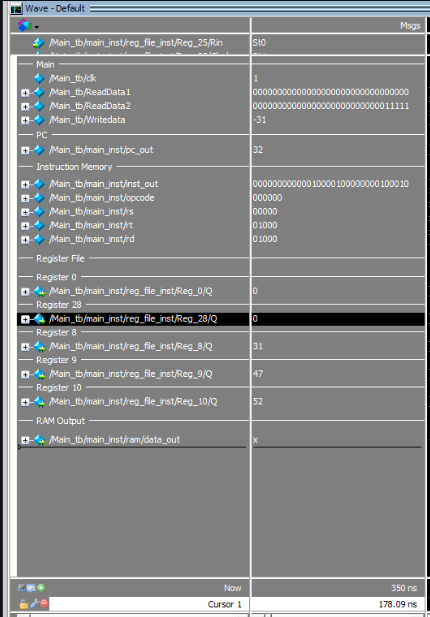
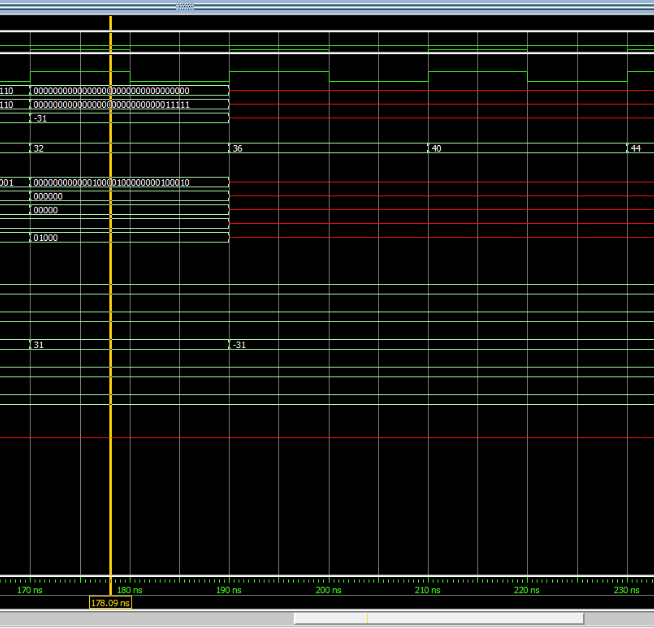
**The value save in register or memory in next positive edge .**



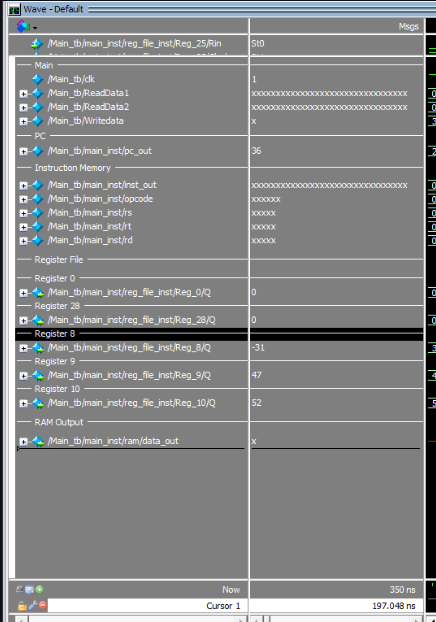
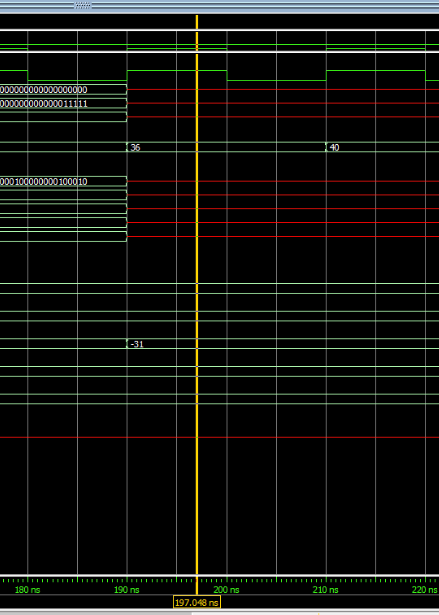
### **Fiveth Instruction (LW R10, 8(R28)**



### the last Instruction (SUB R8 , R0 , R8)

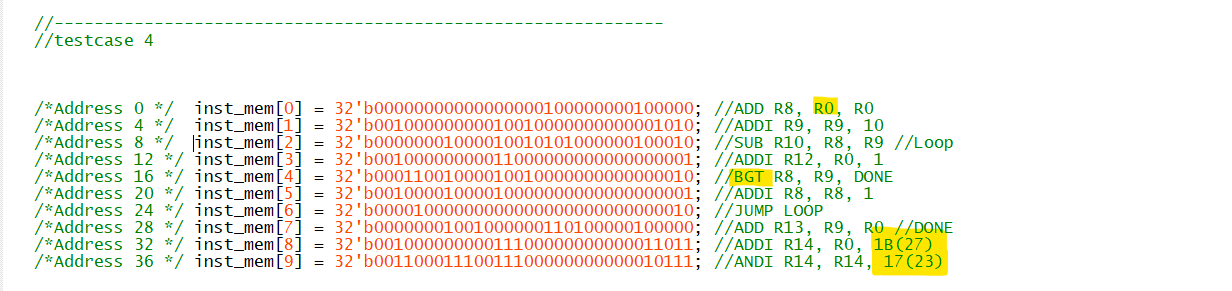


### the cpu after execution



## **Test case 4**

### Instruction memory and data memory





### First Instruction ADD R8, R0, R0 CPU Status before execution

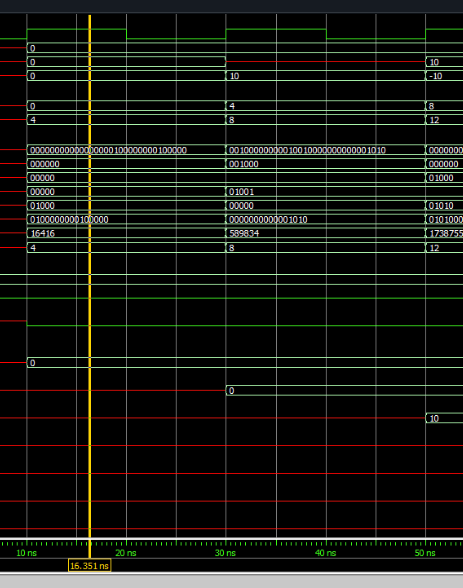
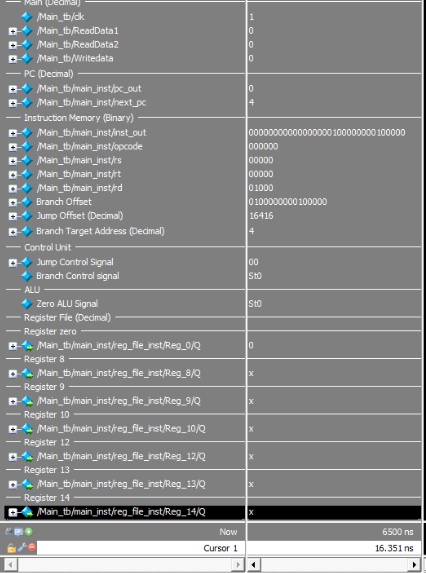
**pc\_out=> current value of pc**

**Pc\_next=> the next pc**

**Data\_out=> read data from RAM**

**We have 32 registers , we only show the register the related to instruction .**

**The value save in register or memory in next positive edge** .

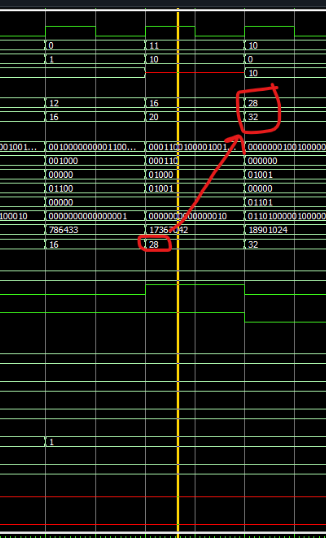


### seventh Instruction JUMP LOOP First Iteration .

A screenshot of a computer

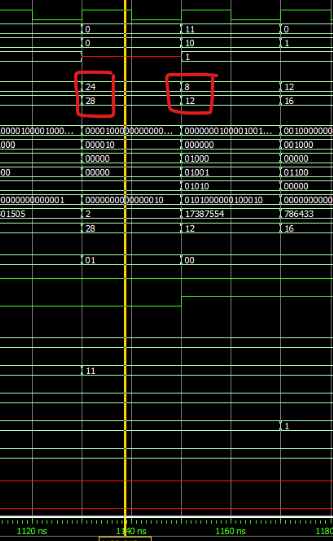
Description automatically generated

### Branch Taken BGT R8, R9,

A screenshot of a computer

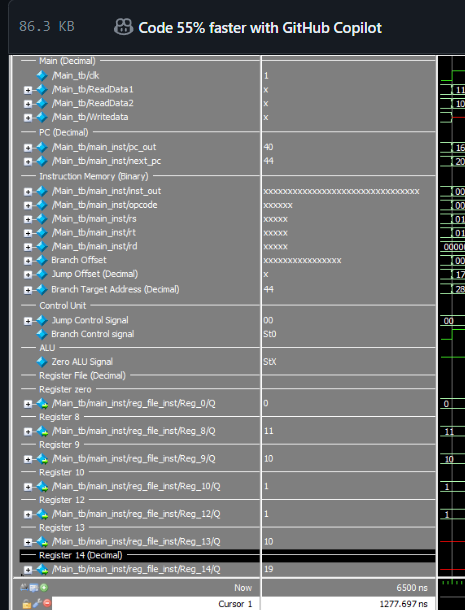
Description automatically generated

### Instruction JUMP LOOP the Last Iteration

A screenshot of a computer

Description automatically generated

### CPU Status after execution

A screen shot of a computer

Description automatically generated

## **Test case 5**

**pc\_out=> current value of pc**

**Pc\_next=> the next pc**

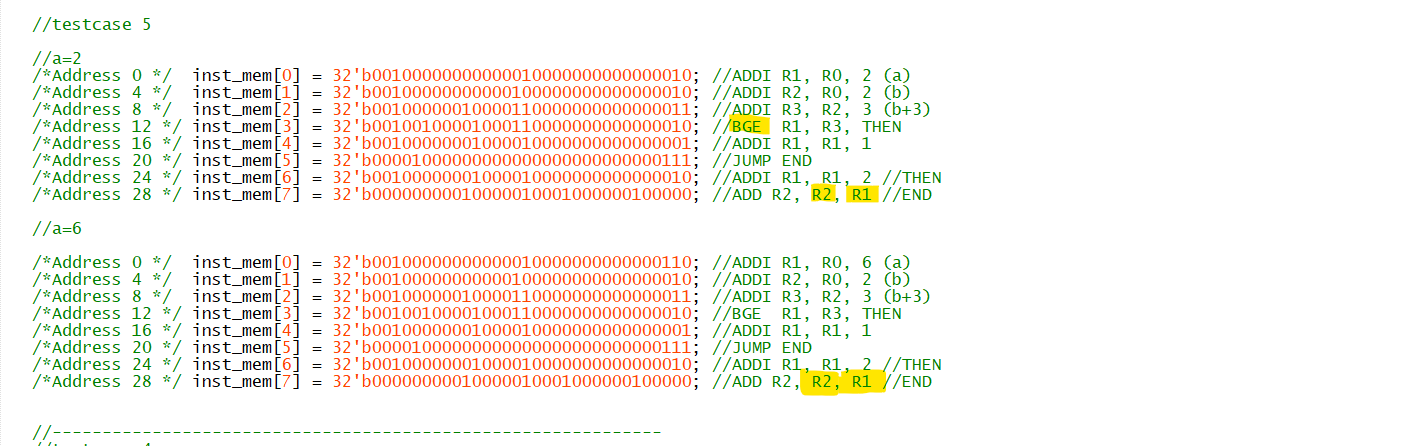
**Data\_out=> read data from RAM**

**We have 32 registers , we only show the register the related to instruction .**

**The value save in register or memory in next positive edge .**

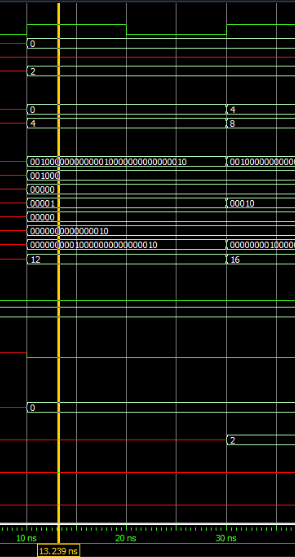
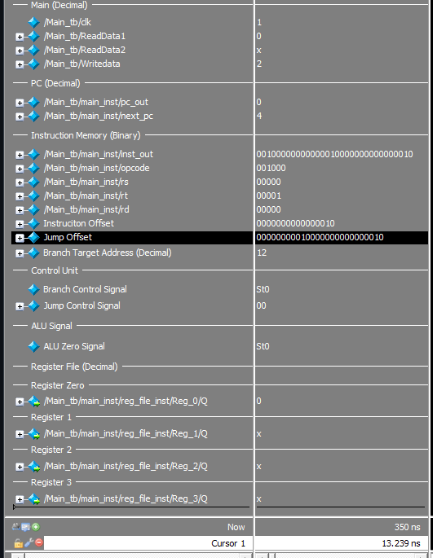
### Instruction memory and data memory

### **No need for RAM**

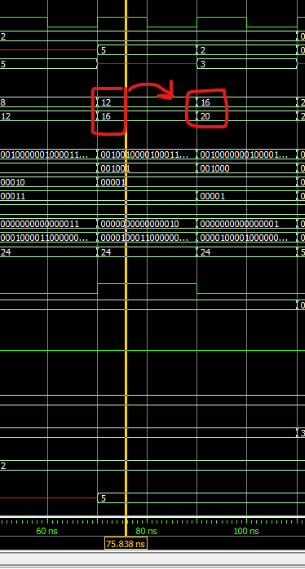
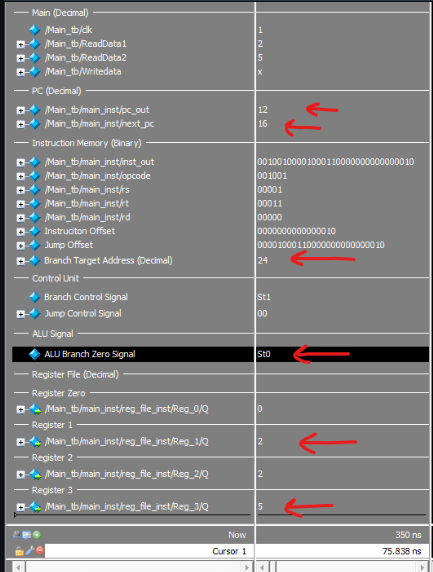


### When a =2

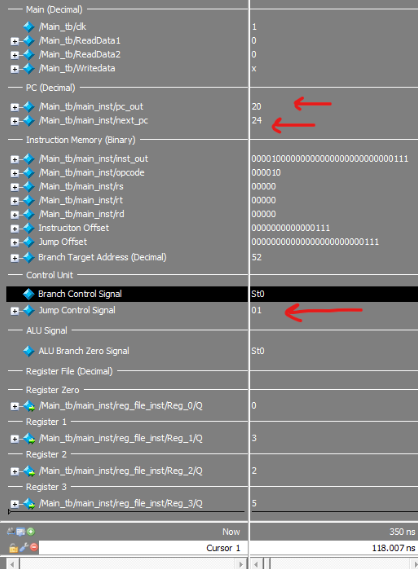
#### **First Instruction ADDI R1, R0, 2 (a) CPU Status before execution**



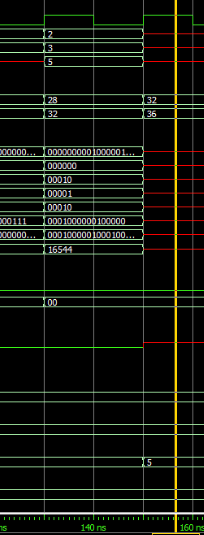
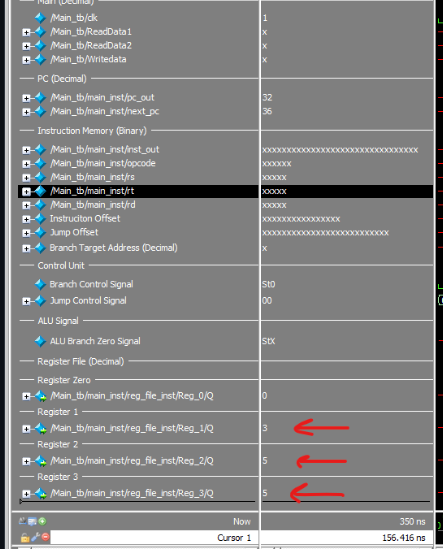
#### Fourth Instruction BGE R1, R3, THEN IF condition True the Branch Not Taken



#### **Sixth Instruction JUMP**

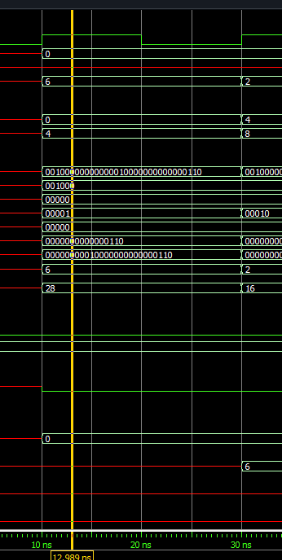
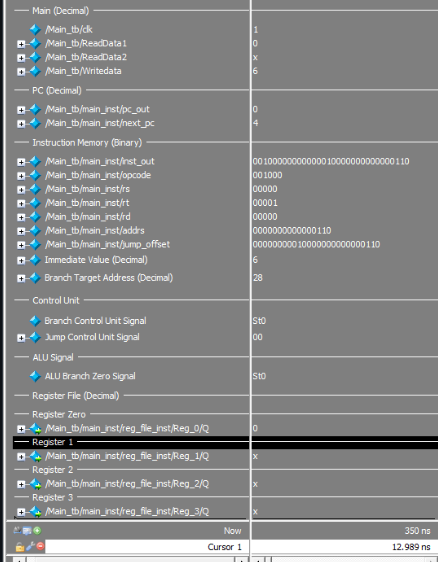


#### **CPU Status After execution**

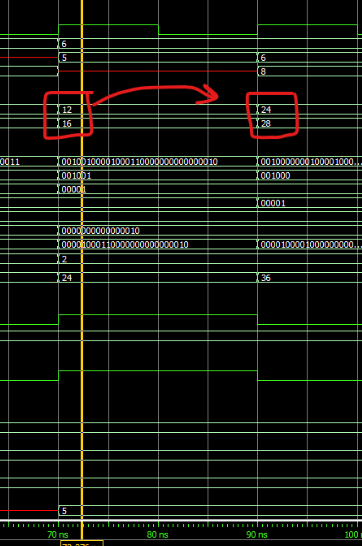
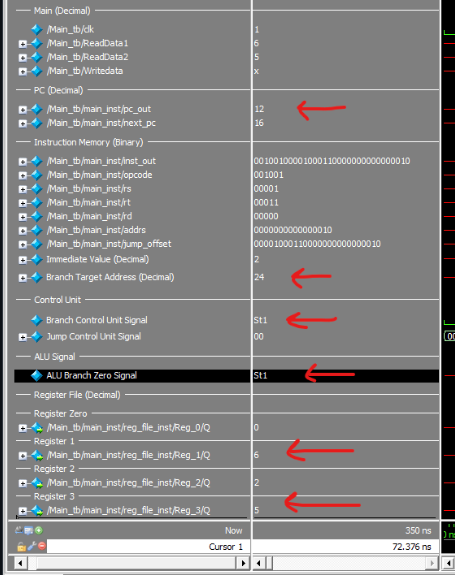


### When a =6

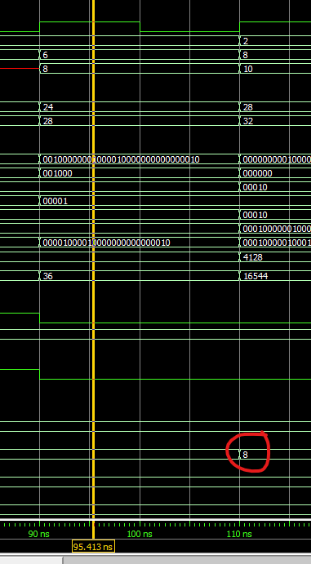
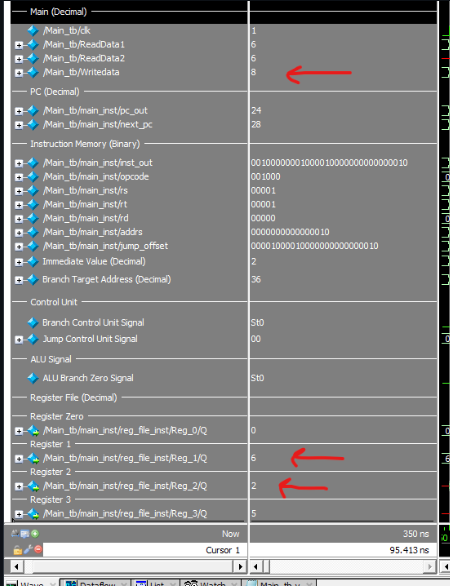
#### **First Instruction ADDI R1, R0, 6 (a) CPU Status before execution.**



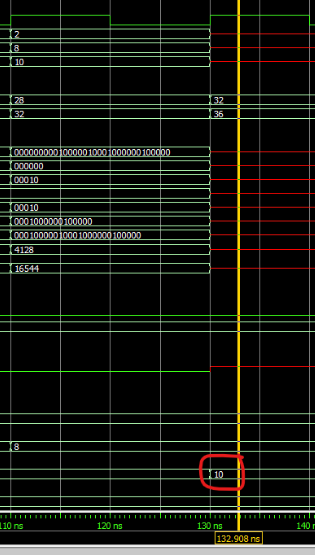
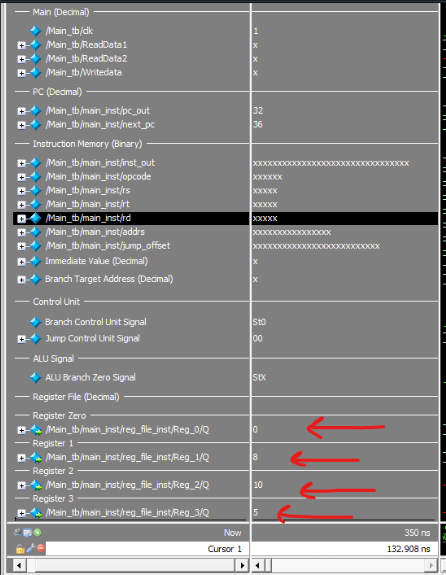
#### **Fourth Instruction BGE R1, R3, THEN Branch Taken IF statment is False**



#### **seventh Instruction ADDI R1, R1, 2**



#### **CPU Status After execution**



## **Test case 6**

### Instruction memory

**No need for RAM**

A computer code with red numbers

Description automatically generated

### First Instruction ADD R1, R0, R0 CPU Status before execution

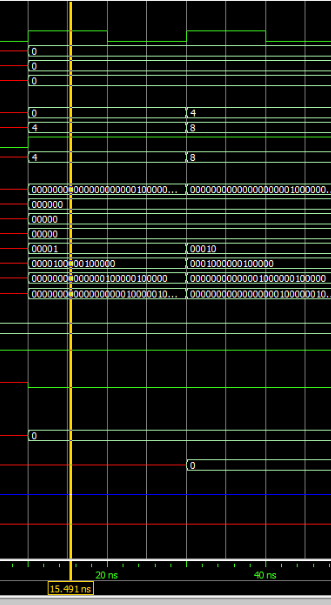
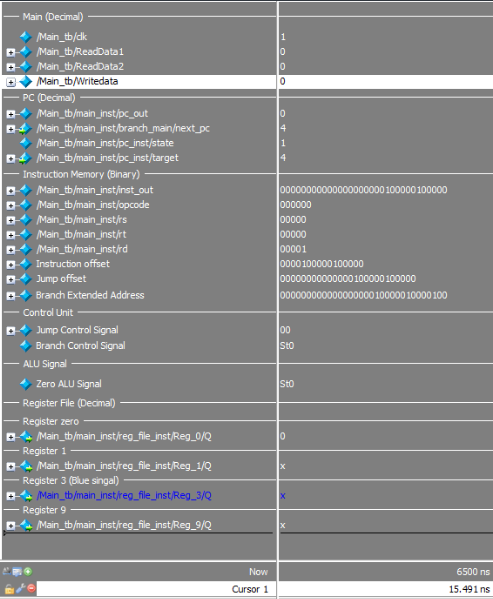
**pc\_out=> current value of pc**

**Pc\_next=> the next pc**

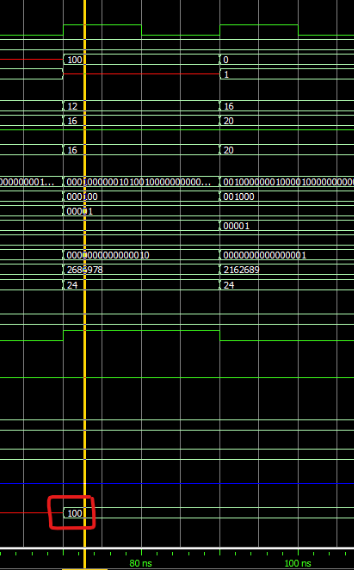
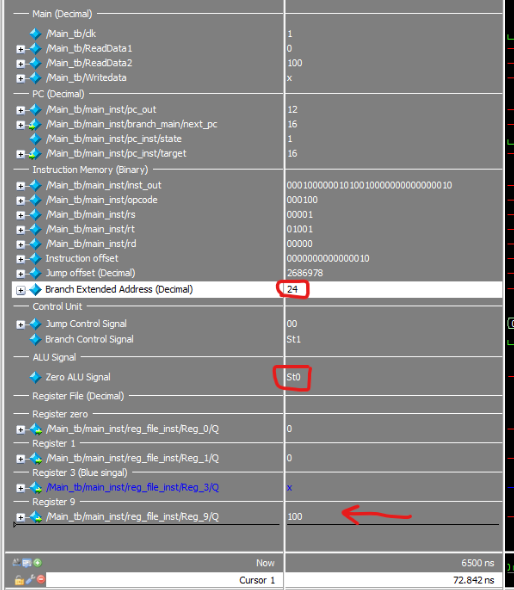
**Data\_out=> read data from RAM**

**We have 32 registers , we only show the register the related to instruction .**

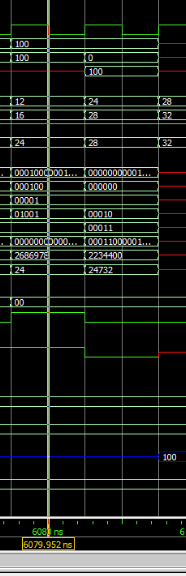
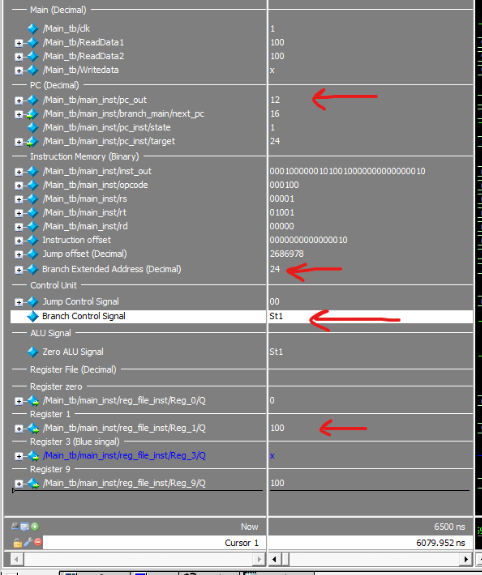
**The value save in register or memory in next positive edge**



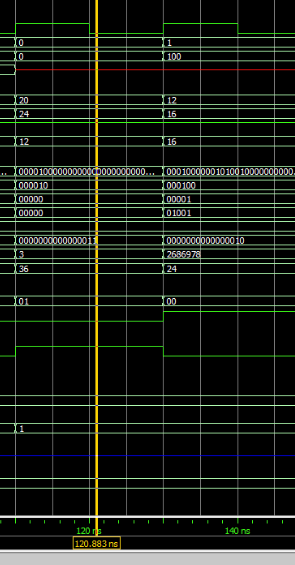
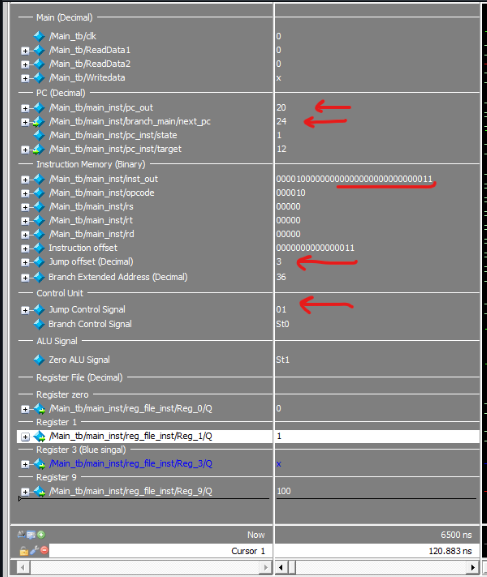
### Fourth Instruction BEQ R1, R9, EXIT



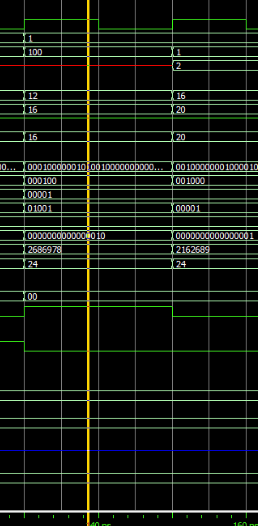
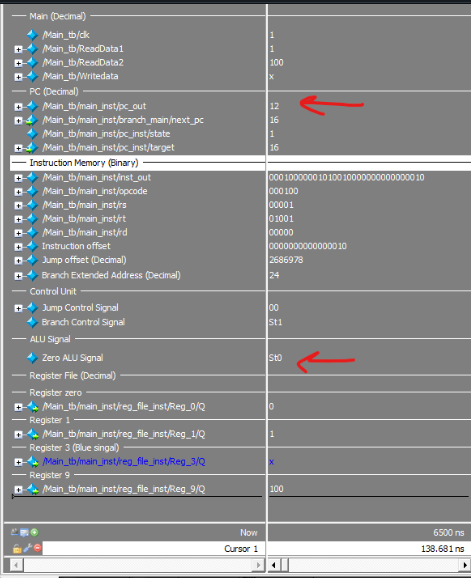
### . Branch Taken to Instruction BEQ R1, R9, EXIT



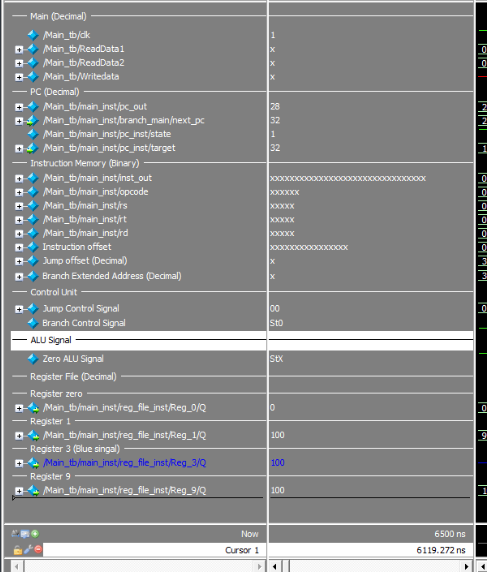
### Instruction JUMP START First turn.



### The second Iteration first Instruction BEQ R1, R9, EXIT after the firt jump execute.

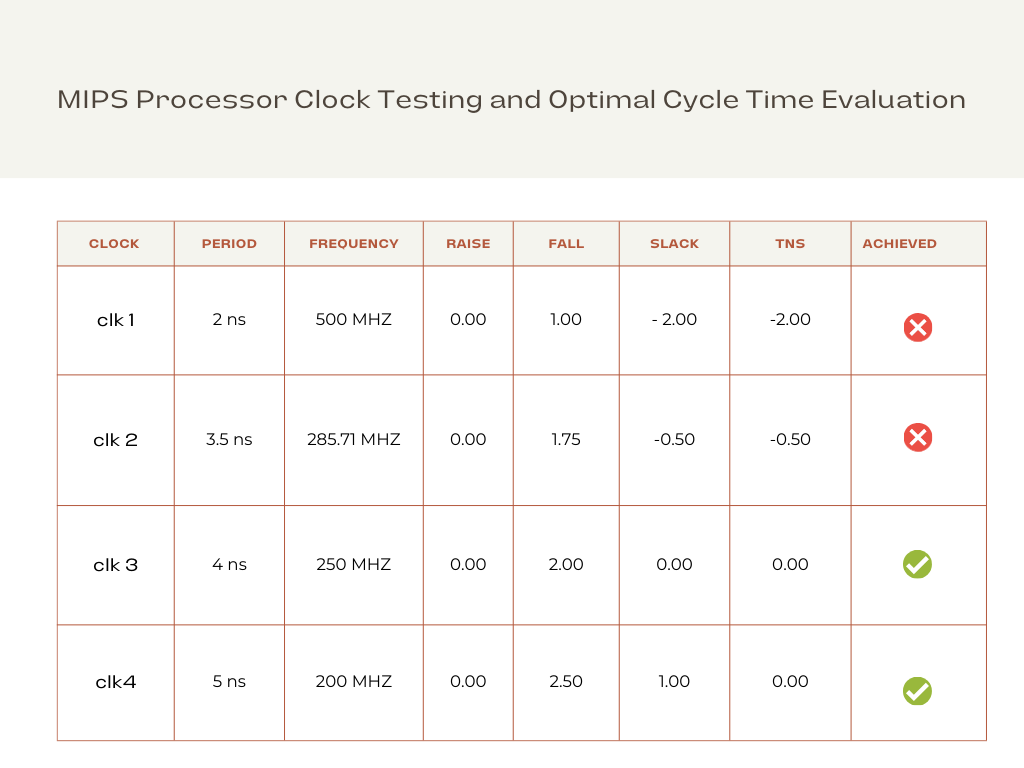


### The CPU Status After execution



## **Clock testing**

As I mentioned earlier, this has been the most challenging part, but I'm proud that we've reached this point



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 testbench to ensure accuracy and functionality.

# Conclusion

The journey to design a single-cycle 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.

### 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.

### Project’s objectives achieved.

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

1. Pipeline CPU Design: Future projects could delve into the design of pipeline CPUs, which allow for a more balanced trade-off between execution speed and resource utilization. A pipeline design can be more efficient and is widely used in modern processors..
2. 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.
3. 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.
4. Exception handler

# Appendix A: CODE

## PC (state machine)

­ 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

pc <= target;

end

end

default: begin

state <= 1'b0;

pc <= 32'h00000000;

end

endcase

end

end

endmodule

## 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

// output reg Overflow, // Overflow flag

// output reg CarryOut // Carry-out 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)

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

// 4'b1111: ALUOut <= (A == B) ? 32'b1 : 32'b0; // Equal comparison ERROR

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]);

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

if (Overflow && (ALUControl != 0110) && (ALUControl != 0111)) begin //signed overflow check

ALUOut = 32'bx;

end else if(CarryOut && ((ALUControl == 4'b0110) || (ALUControl == 4'b0111))) begin //unsigned overflow check

ALUOut = 32'bx;

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

default: Zero <= 1'b0;

endcase

end

endmodule

## Control unit

module ControlUnit(Clock,Reset,opcode,RegDst,ALUSrc,MemtoReg,MemWrite,MemRead,ALUOp,RegWrite,Branch,Jump,funct,pc\_load,PC\_Store);

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,PC\_Store;

output wire [3:0] ALUOp;

output wire [1:0] Jump;

output wire [1:0] MemtoReg;

output wire [1:0] RegDst;

output wire pc\_load;

// reg type

reg reg\_ALUSrc,reg\_MemWrite,reg\_MemRead,reg\_RegWrite,reg\_Branch,reg\_PC\_Store;

reg [3:0] reg\_ALUOp;

reg [1:0] reg\_Jump;

reg [1:0] reg\_MemtoReg;

reg [1:0] reg\_RegDst;

reg reg\_pc\_load;

reg [5:0] reset\_opcode ;

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

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

// ALUSrc, // Control signal for ALU source selection (0 for register, 1 for immediate)

// MemtoReg, // Control signal for selecting the source for register write data

// 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)

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

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\_pc\_load = 1'b0;

reg\_PC\_Store = 1'b0;

end

else begin

case(opcode)

6'b000000:begin

if(funct == 6'b001000)begin

//Jump Register instruction

reg\_ALUSrc = 1'bx;

reg\_RegWrite = 1'b0;

reg\_MemtoReg = 2'bxx;

reg\_MemWrite = 1'b0;

reg\_MemRead = 1'bx;

reg\_ALUOp = 4'bxxxx;

reg\_RegDst = 2'bxx;

reg\_Branch = 1'bx;

reg\_Jump = 2'b10;

reg\_pc\_load = 1'b1;//load

reg\_PC\_Store = 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\_pc\_load = 1'b1;

reg\_PC\_Store = 1'b0;

end

end

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\_pc\_load = 1'b1;

reg\_PC\_Store = 1'b0;

end

6'b101011:begin

// store instruction

reg\_ALUSrc = 1'b1;

reg\_RegWrite = 1'b0;

reg\_MemtoReg = 2'b00; //error (should be 0 ) i am an idiot ...\*\*\*\*\*

reg\_MemWrite = 1'b1;

reg\_MemRead = 1'b0;

reg\_ALUOp = 4'b0000;

reg\_RegDst = 2'b00; //don't care

reg\_Branch = 1'b0;

reg\_Jump = 2'b00;

reg\_pc\_load = 1'b1;

reg\_PC\_Store = 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\_pc\_load = 1'b1;

reg\_PC\_Store = 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\_pc\_load = 1'b1;

reg\_PC\_Store = 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\_pc\_load = 1'b1;

reg\_PC\_Store = 1'b0;

end

6'b000010:begin

//Jump instruction (j)

reg\_ALUSrc = 1'bx;

reg\_RegWrite = 1'b0;

reg\_MemtoReg = 2'bxx;

reg\_MemWrite = 1'b0;

reg\_MemRead = 1'bx;

reg\_ALUOp = 4'bxxxx;

reg\_RegDst = 2'bxx;

reg\_Branch = 1'b0;

reg\_Jump = 2'b01;

reg\_pc\_load = 1'b1;

reg\_PC\_Store = 1'b0;

reg\_PC\_Store = 1'b0;

end

6'b000011:begin

//Jump and link instruction (jal)

reg\_ALUSrc = 1'bx;

reg\_RegWrite = 1'b1;

reg\_MemtoReg = 2'b10;

reg\_MemWrite = 1'b0;

reg\_MemRead = 1'bx;

reg\_ALUOp = 4'bxxxx;

reg\_RegDst = 2'b10;

reg\_Branch = 1'bx;

reg\_Jump = 2'b01;

reg\_pc\_load = 1'b1;

reg\_PC\_Store = 1'b1;

end

6'b000100:begin

//branch equal

reg\_ALUSrc = 1'b0;

reg\_RegWrite = 1'b0;

reg\_MemtoReg = 2'bxx;

reg\_MemWrite = 1'b0;

reg\_MemRead = 1'bx;

reg\_ALUOp = 4'b0100;

reg\_RegDst = 2'bxx;

reg\_Branch = 1'b1;

reg\_Jump = 2'b00;

reg\_pc\_load = 1'b1;

reg\_PC\_Store = 1'b0;

end

6'b000101:begin

//branch not equal

reg\_ALUSrc = 1'b0;

reg\_RegWrite = 1'b0;

reg\_MemtoReg = 2'bxx;

reg\_MemWrite = 1'b0;

reg\_MemRead = 1'bx;

reg\_ALUOp = 4'b0101;

reg\_RegDst = 2'bxx;

reg\_Branch = 1'b1;

reg\_Jump = 2'b00;

reg\_pc\_load = 1'b1;

reg\_PC\_Store = 1'b0;

end

6'b000110:begin

//branch greater than

reg\_ALUSrc = 1'b0;

reg\_RegWrite = 1'b0;

reg\_MemtoReg = 2'bxx;

reg\_MemWrite = 1'b0;

reg\_MemRead = 1'bx;

reg\_ALUOp = 4'b0110;

reg\_RegDst = 2'bxx;

reg\_Branch = 1'b1;

reg\_Jump = 2'b00;

reg\_pc\_load = 1'b1;

reg\_PC\_Store = 1'b0;

end

6'b000111:begin

//branch less than

reg\_ALUSrc = 1'b0;

reg\_RegWrite = 1'b0;

reg\_MemtoReg = 2'bxx;

reg\_MemWrite = 1'b0;

reg\_MemRead = 1'bx;

reg\_ALUOp = 4'b0111;

reg\_RegDst = 2'bxx;

reg\_Branch = 1'b1;

reg\_Jump = 2'b00;

reg\_pc\_load = 1'b1;

reg\_PC\_Store = 1'b0;

end

6'b001001:begin

//branch greater or equal

reg\_ALUSrc = 1'b0;

reg\_RegWrite = 1'b0;

reg\_MemtoReg = 2'bxx;

reg\_MemWrite = 1'b0;

reg\_MemRead = 1'bx;

reg\_ALUOp = 4'b1000;

reg\_RegDst = 2'bxx;

reg\_Branch = 1'b1;

reg\_Jump = 2'b00;

reg\_pc\_load = 1'b1;

reg\_PC\_Store = 1'b0;

end

6'b001010:begin

//branch less or equal

reg\_ALUSrc = 1'b0;

reg\_RegWrite = 1'b0;

reg\_MemtoReg = 2'bxx;

reg\_MemWrite = 1'b0;

reg\_MemRead = 1'bx;

reg\_ALUOp = 4'b1001;

reg\_RegDst = 2'bxx;

reg\_Branch = 1'b1;

reg\_Jump = 2'b00;

reg\_pc\_load = 1'b1;

reg\_PC\_Store = 1'b0;

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 = 2'b00;

reg\_RegDst = 2'b00;

reg\_Branch = 1'b0;

reg\_Jump = 2'b00;

reg\_pc\_load = 1'b1;

reg\_PC\_Store = 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;

assign pc\_load = reg\_pc\_load;

assign PC\_Store = reg\_PC\_Store;

endmodule

## Register file

module RegisterFile(Clock,Reset,ReadReg1,ReadReg2,WriteReg,WriteData,Reg\_write\_Control,ReadData1,ReadData2,PC\_Store);

input Clock , Reset;

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

input Reg\_write\_Control;

input PC\_Store;

input [31:0] WriteData;

output [31:0] ReadData1;

output [31:0] ReadData2;

// define bus (wires)

wire [31:0] Reg\_Enable;

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]);

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]);

RegFile\_regn Reg\_29(WriteData, Reset, Reg\_Enable[29], Clock,Registers\_Read[29]);

RegFile\_regn Reg\_30(WriteData, Reset, Reg\_Enable[30], Clock,Registers\_Read[30]);

RegFile\_regn Reg\_31(WriteData, Reset, PC\_Store, Clock,Registers\_Read[31]);

// Read from Register file

// MUX1: Read first operand

//always @(posedge Clock) begin

assign ReadData1= Registers\_Read[ReadReg1];

//end

// MUX2: Read second operand

assign ReadData2= Registers\_Read[ReadReg2];

endmodule

///////////////\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Register File Modules \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*////////////

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1 - decoder 5 --> 32 bit\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*///////////

module RegFile\_decoder(inputs,enable,outputs);

input [4:0] inputs;

input enable;

output [31:0] outputs;

reg [31:0] decoder\_output;

always @ (\*) begin

if (enable == 1'b1) begin

case (inputs)

5'b00000: decoder\_output = 32'b00000000000000000000000000000001;

5'b00001: decoder\_output = 32'b00000000000000000000000000000010;

5'b00010: decoder\_output = 32'b00000000000000000000000000000100;

5'b00011: decoder\_output = 32'b00000000000000000000000000001000;

5'b00100: decoder\_output = 32'b00000000000000000000000000010000;

5'b00101: decoder\_output = 32'b00000000000000000000000000100000;

5'b00110: decoder\_output = 32'b00000000000000000000000001000000;

5'b00111: decoder\_output = 32'b00000000000000000000000010000000;

5'b01000: decoder\_output = 32'b00000000000000000000000100000000;

5'b01001: decoder\_output = 32'b00000000000000000000001000000000;

5'b01010: decoder\_output = 32'b00000000000000000000010000000000;

5'b01011: decoder\_output = 32'b00000000000000000000100000000000;

5'b01100: decoder\_output = 32'b00000000000000000001000000000000;

5'b01101: decoder\_output = 32'b00000000000000000010000000000000;

5'b01110: decoder\_output = 32'b00000000000000000100000000000000;

5'b01111: decoder\_output = 32'b00000000000000001000000000000000;

5'b10000: decoder\_output = 32'b00000000000000010000000000000000;

5'b10001: decoder\_output = 32'b00000000000000100000000000000000;

5'b10010: decoder\_output = 32'b00000000000001000000000000000000;

5'b10011: decoder\_output = 32'b00000000000010000000000000000000;

5'b10100: decoder\_output = 32'b00000000000100000000000000000000;

5'b10101: decoder\_output = 32'b00000000001000000000000000000000;

5'b10110: decoder\_output = 32'b00000000010000000000000000000000;

5'b10111: decoder\_output = 32'b00000000100000000000000000000000;

5'b11000: decoder\_output = 32'b00000001000000000000000000000000;

5'b11001: decoder\_output = 32'b00000010000000000000000000000000;

5'b11010: decoder\_output = 32'b00000100000000000000000000000000;

5'b11011: decoder\_output = 32'b00001000000000000000000000000000;

5'b11100: decoder\_output = 32'b00010000000000000000000000000000;

5'b11101: decoder\_output = 32'b00100000000000000000000000000000;

5'b11110: decoder\_output = 32'b01000000000000000000000000000000;

5'b11111: decoder\_output = 32'b10000000000000000000000000000000;

default: decoder\_output = 32'b00000000000000000000000000000000;

endcase

end

else begin

decoder\_output = 32'b00000000000000000000000000000000;

end

end

assign outputs = decoder\_output;

endmodule

/////\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 3 - register 32-bit \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//////

// don't forget to change the parameter when you do not 32 bit register

module RegFile\_regn(R, Resetn, Rin, Clock, Q);

parameter n = 32;

input [n-1:0] R;

input Resetn, Rin, Clock;

output [n-1:0] Q;

reg [n-1:0] Q;

always @(posedge Clock)

if (Resetn)

Q <= 0;

else if (Rin)

Q <= R;

Endmodule

## 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

## Main

module Main (

input clk,

input reset,

output wire [31:0] ReadData1,

output wire [31:0] ReadData2,

output wire [31:0] Writedata

);

//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;

wire pc\_load;//control

PC #(.first\_address(0), .pc\_inc(4) )

pc\_inst (

.clk(clk),

.reset(reset),

.target(pc\_final),

.pc\_load(pc\_load),

// .branch\_condition(Branch&zero),

.pc(pc\_out)

);

//end of PC

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

//inst\_mem

wire[31:0] inst\_out;

wire [5:0] opcode;

wire [4:0] rs;

wire [4:0] rt;

wire [4:0] rd;

wire [4:0] shamt;

wire [5:0] funct;

wire [15:0] addrs;

wire [25:0] jump\_offset;

INST\_MEM #(.size(32),.data\_width(32) )

inst\_mem (

// .clk(clk),

.reset(reset),

.address(pc\_out),

.inst\_out(inst\_out),

.opcode(opcode),

.rs(rs),

.rt(rt),

.rd(rd),

.shamt(shamt),

.funct(funct),

.addr(addrs),

.jump(jump\_offset)

);

//end of inst\_mem

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

//sign extend

wire [31:0] immediate\_value;

sign\_extend extender (

.extend(addrs),

.extended(immediate\_value)

);

//end of sign extend

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

//ControlUnit

wire ALUSrc, MemWrite, MemRead, RegWrite,Branch;

wire [3:0] ALUOp;

wire [1:0] RegDst;

wire [1:0] MemtoReg;

wire [1:0] Jump\_signal;

ControlUnit control\_inst (

.Clock(clk),

.Reset(reset),

.opcode(opcode),

.RegDst(RegDst),

.ALUSrc(ALUSrc),

.MemtoReg(MemtoReg),

.MemWrite(MemWrite),

.MemRead(MemRead),

.RegWrite(RegWrite),

.ALUOp(ALUOp),

.Branch(Branch),

.Jump(Jump\_signal),

.funct(funct),

.pc\_load(pc\_load)

);

//end of ControlUnit

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

// Reg\_File

wire [4:0] write\_reg\_input;

MUX5bit mux\_inst (

.a(rt),

.b(rd),

.select(RegDst),

.out(write\_reg\_input)

);

//reg [31:0] ReadData0;

/\* output wire [31:0] ReadData1;

output wire [31:0] ReadData2;

output wire [31:0] Writedata;\*/

RegisterFile reg\_file\_inst (

.Clock(clk),

.Reset(reset),

.ReadReg1(rs),

.ReadReg2(rt),

.WriteReg(write\_reg\_input),

.Reg\_write\_Control(RegWrite),

.WriteData(Writedata), //still error // Writedata

.ReadData1(ReadData1),

.ReadData2(ReadData2)

//.Reg\_Enable(Reg\_Enable), // Connect Reg\_Enable signal

//.Registers\_Read(Registers\_Read) // Connect Registers\_Read signals

);

/\* always @(posedge clk) begin

assign ReadData0 = ReadData1 ;

end\*/

//end of Reg\_File\*/

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

//alu\_cntrl

wire [3:0] Operation;

wire [2:0] branch\_type;

alu\_control alu\_ctrl (

.clk(clk),

.FuncField(funct),

.ALUOp(ALUOp),

.Operation(Operation),

.branch\_type(branch\_type)

);

//end of alu\_cntrl

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

// MUX2\_1 alu\_sec\_input

wire [31:0] alu\_second\_input;

MUX2\_1 alu\_sec\_input (

.a(ReadData2),

.b(immediate\_value),

.select(ALUSrc),

.out(alu\_second\_input)

);

//end of MUX2\_1 alu\_sec\_input

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

//ALU

wire [31:0] alu\_output;

wire zero ;

ALU alu (

.clk(clk),

.A(ReadData1),

.B(alu\_second\_input),

.ALUControl(Operation),

.ShiftAmount(shamt),

.branch\_type(branch\_type),

.ALUOut(alu\_output),

.Zero(zero)

);

//end of ALU

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

// DATA MAM

wire [31:0] Read\_data;

RAM #(

.size(32),

.data\_width(32)

) ram (

.clk(clk),

.reset(reset),

.address(alu\_output),

.data\_write(ReadData2),

.write\_en(MemWrite),

.read\_en(MemRead),

.data\_out(Read\_data)

);

//end of DATA\_MEM

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

//Write back

MUX4\_1 Write\_back (

.a(alu\_output),

.b(Read\_data),

.c(next\_pc),

.select(MemtoReg),

.out(Writedata)

);

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

//handling jump instruction

adder add(

.a(pc\_inc),

.b(pc\_out),

.c(next\_pc)

);

wire [31:0] jump\_target;

assign jump\_target = {next\_pc[31:28], jump\_offset, 2'b00};

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

//branch

wire [31:0] branch;

branch\_control branch\_main(

.extended\_address(immediate\_value),

.next\_pc(next\_pc),

.branch(branch)

);

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

wire [31:0] pc\_branch;

MUX2\_1 pc\_target(

.a(next\_pc),

.b(branch),

.select((Branch & zero)),

.out(pc\_branch)

);

MUX4\_1 pc\_final\_main(

.a(pc\_branch),

.b(jump\_target),

.c(ReadData1),

.select(Jump\_signal),

.out(pc\_final)

);

Endmodule

## 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

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

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

## Adder

module adder (

input wire [31:0] a,

input wire [31:0] b,

output wire [31:0] c

);

assign c = a + b;

endmodule

## 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

## RAM

module RAM #(

parameter size = 32,

parameter data\_width = 32

// parameter once=1

)(

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

);

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

reg error;

// reg first=once;

// Declare a memory array with parameterized size and data width.

// reg [data\_width-1:0] mem [0:size - 1];

initial begin

//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;

//testcase 4 & 5 & 6

//no need for RAM

//testcase 3

//

// mem[0] = 32'h00000023; // 0x23 (35)

// mem[1] = 32'h0000002F; // 0x2F (47)

// mem[2] = 32'h0000001A; // 0x1A (26)

//

//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

/\*

mem[0] = 32'b00000000000000000000000000000011; // 3

mem[1] = 32'b00000000000000000000000000000101; // 5

mem[2] = 32'b00000000000000000000000000001000; // 8

mem[3] = 32'b00000000000000000000000000001100; // 12

mem[4] = 32'b00000000000000000000000000000110; // 6

mem[5] = 32'b00000000000000000000000000010100; // 20

mem[6] = 32'b00000000000000000000000000011000; // 24

mem[7] = 32'b00000000000000000000000000011100; // 28

mem[8] = 32'b00000000000000000000000000100000; // 32

mem[9] = 32'b00000000000000000000000000100100; // 36

//mem[10]= 32'b00000000000000000000000000001110; // from insrtuction sw =>14

mem[11]= 32'b01111111111111111111111111111111; // big number

mem[12]= 32'b00000000000000000000000000000111; // 7

mem[13]= 32'b00100000000000000000000000000000; // big number -for overflow check-

mem[14]= 32'b11110000000000000000000000000000; // big number -for overflow check-

\*/

//$readmemh("data\_memory\_initialization", mem );

end

// assign data\_out = mem[address >> 2];

// assign data\_out = (reset)?8'h00000000:mem[address >> 2];

assign data\_out=(read\_en)? ((reset)?8'h00000000:mem[address >> 2]):32'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;

integer i;

always @(posedge clk) begin

if (reset) begin

for (i = 0; i < size; i = i + 1) begin

mem[i] <= 32'b0;

end

error <= 0;

end

else if (write\_en) begin

// Write data

mem[address >> 2] <= data\_write;

end

end

endmodule

## instruction memory

module INST\_MEM #(

parameter size = 32,

parameter data\_width = 32

)(

// input clk,

input reset,

input [31:0] address,

output reg [31:0] inst\_out,

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

);

reg [31:0] inst\_mem [0:size - 1];

initial begin // this should be removed because it is NOT synthesizable

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

//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

// /\*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

//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

//

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

//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)

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

//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

// inst\_mem[4] = 32'b10001111100010100000000000001000; //LW R10, 8(R28)

// inst\_mem[5] = 32'b00000001010010100101000000100000; //ADD R10, R10, R10

// inst\_mem[6] = 32'b00000001000010100100000000100010; //SUB R8, R8, R10

// inst\_mem[7] = 32'b00100001000010000000000000000001; //ADDI R8, R8, 1

// inst\_mem[8] = 32'b00000000000010000100000000100010; //SUB R8, R0, R8

//

//

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

//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'b10001100000000100000000000000100; //LW R2, 4(R0)

// inst\_mem[4] = 32'b10001100000000110000000000010000; //LW R3, 16(R0)

// inst\_mem[5] = 32'b00000000010000000001100001000000; //SLL R3, R2, 1

// inst\_mem[6] = 32'b10101100000000110000000000001100; //SW R3, 12(R0)

// inst\_mem[7] = 32'b10001100000001000000000000001100; //LW R4, 12(R0)

//

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

//testcase 1

// inst\_mem[0] = 32'b10001100000010000000000000000000; //LW R8, 0(R0)

// inst\_mem[1] = 32'b10001100000010010000000000100000; //LW R9, 0x20(R0)

// inst\_mem[2] = 32'b10001100000010100000000001010000; //LW R10, 0x50(R0)

// inst\_mem[3] = 32'b10001100000010110000000000001000; //LW R11, 0x8(R0)

//

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

//THIS TEST FOR BLT

/\*

inst\_mem[0] = 32'b10001100000000010000000000000100;//lw reg1=3

inst\_mem[1] = 32'b10001100001000100000000000000101;//lw reg2=8

inst\_mem[2] = 32'b00011100001000100000000000000001;//bLt address=16 -->4\*1+12 \*\*should work\*\*

inst\_mem[3] = 32'b10001100001000110000000000010001;//lw reg3=20 \*\*should not work\*\*

inst\_mem[4] = 32'b10001100010001000000000000000100;//lw reg4=12 \*\*should work\*\*

\*/

//THIS TEST FOR BLT not working

/\*

inst\_mem[0] = 32'b10001100000000010000000000000100;//lw reg1=3

inst\_mem[1] = 32'b10001100001000100000000000000101;//lw reg2=8

inst\_mem[2] = 32'b00011100010000010000000000000001;//bLt address=16 -->4\*1+12 \*\*should not work\*\*

inst\_mem[3] = 32'b10001100001000110000000000010001;//lw reg3=20 \*\*should work\*\*

inst\_mem[4] = 32'b10001100010001000000000000000100;//lw reg4=12 \*\*should work\*\*

\*/

//THIS TEST FOR BGT

/\*

inst\_mem[0] = 32'b10001100000000010000000000000100;//lw reg1=3

inst\_mem[1] = 32'b10001100001000100000000000000101;//lw reg2=8

inst\_mem[2] = 32'b00011000010000010000000000000001;//bgt address=16 -->4\*1+12 \*\*should work\*\*

inst\_mem[3] = 32'b10001100001000110000000000010001;//lw reg3=20 \*\*should not work\*\*

inst\_mem[4] = 32'b10001100010001000000000000000100;//lw reg4=12 \*\*should work\*\*

\*/

//THIS TEST FOR BGT not working

/\*

inst\_mem[0] = 32'b10001100000000010000000000000100;//lw reg1=3

inst\_mem[1] = 32'b10001100001000100000000000000101;//lw reg2=8

inst\_mem[2] = 32'b00011000001000100000000000000001;//bgt address=16 -->4\*1+12 \*\*should not work\*\*

inst\_mem[3] = 32'b10001100001000110000000000010001;//lw reg3=20 \*\*should work\*\*

inst\_mem[4] = 32'b10001100010001000000000000000100;//lw reg4=12 \*\*should work\*\*

\*/

//THIS TEST FOR BNE

/\*

inst\_mem[0] = 32'b10001100000000010000000000000100;//lw reg1=3

inst\_mem[1] = 32'b10001100001000100000000000000101;//lw reg2=8

inst\_mem[2] = 32'b00010100001000100000000000000001;//bne address=16 -->4\*1+12 \*\*should not work\*\*

inst\_mem[3] = 32'b10001100001000110000000000010001;//lw reg3=20 \*\*should not work\*\*

inst\_mem[4] = 32'b10001100010001000000000000000100;//lw reg4=12 \*\*should work\*\*

\*/

//THIS TEST FOR BNE not working

/\*

inst\_mem[0] = 32'b10001100000000010000000000000100;//lw reg1=3

inst\_mem[1] = 32'b10001100000000100000000000000100;//lw reg2=3

inst\_mem[2] = 32'b00010100001000100000000000000001;//bne address=16 -->4\*1+12

inst\_mem[3] = 32'b10001100001000110000000000010001;//lw reg3=20 \*\*should work\*\*

inst\_mem[4] = 32'b10001100010001000000000000001001;//lw reg4=12 \*\*should work\*\*

\*/

//THIS TEST FOR BEQ

/\*

inst\_mem[0] = 32'b10001100000000010000000000000100;//lw reg1=3

inst\_mem[1] = 32'b10001100000000100000000000000100;//lw reg2=3

inst\_mem[2] = 32'b00010000001000100000000000000001;//beq address=16 -->4\*1+12

inst\_mem[3] = 32'b10001100001000110000000000010001;//lw reg3=20 \*\*should not work\*\*

inst\_mem[4] = 32'b10001100010001000000000000001001;//lw reg4=12 \*\*should work\*\*

\*/

//THIS TEST FOR BEQ not working

/\*

inst\_mem[0] = 32'b10001100000000010000000000000100;//lw reg1=3

inst\_mem[1] = 32'b10001100001000100000000000000101;//lw reg2=8

inst\_mem[2] = 32'b00010000001000100000000000000001;//beq address=16 -->4\*1+12 \*\*should not work\*\*

inst\_mem[3] = 32'b10001100001000110000000000010001;//lw reg3=20 \*\*should work\*\*

inst\_mem[4] = 32'b10001100010001000000000000000100;//lw reg4=12 \*\*should work\*\*

\*/

/\* THIS TEST FOR JUMP INSTRUCTIONS

inst\_mem[0] = 32'b10001100000000010000000000001100;//load 12 in reg1

inst\_mem[1] = 32'b00001100000000000000000000000011;//jump to inst\_mem[3] and save pc+1 in reg31 (jal)

// inst\_mem[1] = 32'b00000000001000000000000000001000;//jump to content of reg1 (pc=12)

// inst\_mem[1] = 32'b00001000000000000000000000000100;//jump to inst\_mem[4] and skip inst\_mem[3]

inst\_mem[2] = 32'b10001100000000100000000000010100;//load 20 in reg2

inst\_mem[3] = 32'b10001100000000110000000000001000;//load 8 in reg3

\*/

/\*

inst\_mem[0] = 32'b10001100000000010000000000000100; //LW $1 , $4(0) -> load the content of address (content of reg 0 + 4=4) in ram to reg1 =3

inst\_mem[1] = 32'b10001100001000100000000000000101; //LW $2 , $5(1) -> load the content of address (content of reg 3 + 5=8) in ram to reg2 =8

inst\_mem[2] = 32'b00000000001000100101000000100000; //add $10,$1,$2 -> add the content of reg 1 and 2 then store it in reg 10 = 11

inst\_mem[3] = 32'b10001100001000110000000000010001; //LW $3 , $17(1) -> load the content of address (content of reg 1(3) + 17=20) in ram to reg3 =20

inst\_mem[4] = 32'b00100000010001010000000000000110; //addi $5,$2,6 -> add the content of reg2 to 6 (8+6 =14) then store it in reg5 =14

inst\_mem[5] = 32'b10001100010001000000000000000100; //LW $4 , $4(3) -> load the content of address (content of reg2 (8) + 4=12) in ram to reg4 =12

inst\_mem[6] = 32'b00000000101000010011000000100010; //sub $6,$5,$1 -> sub the content of reg1 from reg5 (14-3=11) then store it in reg6=11

inst\_mem[7] = 32'b00000000001000100011100000100010; //sub $7,$1,$2 -> sub the content of reg10 from reg1 (3-8=-5) then store it in reg7=-5

inst\_mem[8] = 32'b00000000001010100100000000100100; //and $8,$1,$10 -> and the content of reg10 with reg2 (ans:3) then store it in reg8=3

inst\_mem[9] = 32'b00000000001010100100100000100101; //or $9,$1,$10 -> or the content of reg10 with reg2 (ans:11) then store it in reg9=11

inst\_mem[10] = 32'b00000000110001000101100000100110; //xor $11,$4,$6 -> xor the content of reg15 with reg10 (ans:3) then store it in reg11=7

inst\_mem[11] = 32'b00000000010001100110000000100111; //nor $12,$2,$6 -> nor the content of reg15 with reg10 (ans:8) then store it in reg12=-12

inst\_mem[12] = 32'b00000000001001110110100000011000; //mul $13,$1,$7 -> mul the content of reg1 with reg7 (ans:-5\*3=-15) then store it in reg13=-15

inst\_mem[13] = 32'b00000000100010000111000000011010; //div $14,$4,$8 -> div the content of reg4 by reg8 (ans:12/3=4) then store it in reg14 = 4

inst\_mem[14] = 32'b00000001000000000100000010000000; //sll $8 ,$0,$8 -> reg8=3\*4=12

inst\_mem[15] = 32'b00000001000000000100000010000010; //slr $8 ,$0,$8 -> reg8=12/4=3

inst\_mem[16] = 32'b10101100000001010000000000101000; //sw $5,$40($0) -> sw {m[40]=14}

inst\_mem[17] = 32'b10001100000011110000000000101000; //lw $15,$40($0) -> reg15 = m[40] = 14

inst\_mem[18] = 32'b00110100100100000000000000000101; //ori $16,$4,5 -> or the content of reg4 (1100) with 5(0101) and sw in reg16=13

inst\_mem[19] = 32'b00110000100100010000000000000101; //andi $17,$4,5 -> and the content of reg4 (1100) with 5(0101) and sw in reg17=4

inst\_mem[20] = 32'b10001100000100100000000000101100; //LW $18,$44($0) -> reg18=big num

inst\_mem[21] = 32'b10001100000100110000000000101100; //LW $19,$44($0) -> reg19=big num

inst\_mem[22] = 32'b00000010010100111010000000100000; //add $20,$18,$19 -> reg20= 32'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

inst\_mem[23] = 32'b10001100000101010000000000110100; //LW $21,$0(0)

inst\_mem[24] = 32'b10001100000101100000000000111000; //Lw $22,$40(0)

inst\_mem[25] = 32'b00000010101101101011100000100001; //addu $23,22,21 -> should do unsigned overflow !

inst\_mem[26] = 32'b00000000111011011100000000100001; //addu $24,$7,$13 -> reg24= should do unsigned overflow !

inst\_mem[27] = 32'b00000000111011011100100000100000; //add $25,$7,$13 -> reg25= -20

inst\_mem[28] = 32'b00000000111011011101000000100011; //subu $26,$7,$13 -> reg26= 10

inst\_mem[29] = 32'b00000000111011011101100000100010; //sub $27,$7,$13 -> reg27= 10\*/

/\*

checking LW muliable times

inst\_mem[0] = 32'b10001100000000010000000000000100; //reg1=3

inst\_mem[1] = 32'b10001100001000100000000000000101; //reg2=8

inst\_mem[2] = 32'b10001100001000110000000000010001; //reg3=20

inst\_mem[3] = 32'b10001100000011110000000000101000; //reg15=14

inst\_mem[4] = 32'b10001100000100100000000000101100; //reg18=big

inst\_mem[5] = 32'b10001100000100110000000000101100; //reg19=big

inst\_mem[6] = 32'b10001100000101010000000000101100; //reg21=big\_num

inst\_mem[7] = 32'b10101100000101010000000000000000;// m[0]=big\_num

inst\_mem[8] = 32'b10001100000101100000000000000000;//reg22=big\*/

end

integer i;

always @(\*) begin

if (reset) begin

for (i = 0; i < size; i = i + 1) begin

inst\_mem[i] <= 32'b0;

end

end

else begin

inst\_out <= inst\_mem[address >> 2];

opcode <= inst\_out[31:26];

rs <= inst\_out[25:21];

rt <= inst\_out[20:16];

rd <= inst\_out[15:11];

shamt <= inst\_out[10:6];

funct <= inst\_out[5:0];

addr <= inst\_out[15:0];

jump <= inst\_out[25:0];

end

end

/\*

after doing this code the registers values are :

reg1 3

reg2 8

reg3 20

reg4 12

reg5 14

reg6 11

reg7 -5

reg8 3

reg9 11

reg10 11

reg11 7

reg12 -12

reg13 -15

reg14 4

reg15 14

reg16 13

reg17 4

reg18 (2^31)-1

reg19 (2^31)-1

reg20 don't care

reg21 big num -overflow check-

reg22 big num -overflow check-

reg23 don't care

\*/

// ((I type -> load and store))

// Opcode | RS | RD | Offset/Immediate

// 6 5 5 16

// (( R type))

// Opcode | RS | RT | RD | Function Code

// 6 5 5 5 6

// RAM :

// add data

// 0 2

// 4 3

// 8 8

// 12 12

// 16 6

// 20 20

// 24 24

// 28 28

// 32 32

// 36 36

// 40 14

// 44 (2^31-1)

// 48 8'h2000000

// 52 8'hF000000

//$readmemh("./memfile\_text.hex",mem,0,63);

/\* 'initial begin' should be removed because it is NOT synthesizable

status register must be added

initialize memories after reset

subu overflow check !!!

\*/

endmodule