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

MEET OUR TEAM

OMAR AL-SALEH

TEAM MEMBER

OMAR AL-KHASAWNEH

TEAM LEADER

MOYYAD ABU MALLOUH

TEAM MEMBER

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 ..... *iv*

Introduction *7*

1.1 Introduction 7

1.2 The Importance of the Design 7

1.3 Motivation 7

1.4 Why This Topic is Important for Students 7

1.5 Objectives of the Project: 8

1.6 Description of Design Achieved: 8

1.7 Design Requirements: 9

1.8 The teams member responsibility 9

1.8.1 Omar AL-khasawneh (Leader) 10

1.8.2 Omar AL-salah 10

1.8.3 Moayyad Abu Mallouh 10

1.9 Organization of the rest of the documentation. 11

Design *12*

1.10 Hardware Design and Implementation 12

1.10.1 Component 12

1.11 mips instruction execution phase 14

1.12 Coding and Software Development 14

1.12.1 Tools 14

1.12.2 Challenging 15

1.12.3 Biggest challenge 15

1.12.4 Instruction set architecture 18

Results *19*

1.13 Program flow 19

1.13.1 Initial Data 19
1.13.2 Programm instruction.....19
1.13.3 First cycle.....20
1.13.4 Second cycle 20
1.13.5 Third cycle 21
1.13.6 Cycle four.....21
1.13.7 Cycle five 22
1.13.8 Cycle six 22
1.13.9 Cycle seven 23
1.13.10 Cycle eight 23
1.13.11 Cycle nine.....24
1.13.12 Cycle tin 24
1.13.13 Cycle eleven 24
1.13.14 Cycle twelve.....25
1.13.15 Cycle thirteen.....25
1.13.16 Cycle Fourteen 26
1.13.17 Cycle Fiveteen 26
1.13.18 Cycle sixteen 27
1.14 Clock testing 29
Conclusion 30
1.14.1 Key Findings and Implications.....30
1.14.2 Project's objectives achieved.....30
Appendix A: CODE 31

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 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.

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. **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.

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.

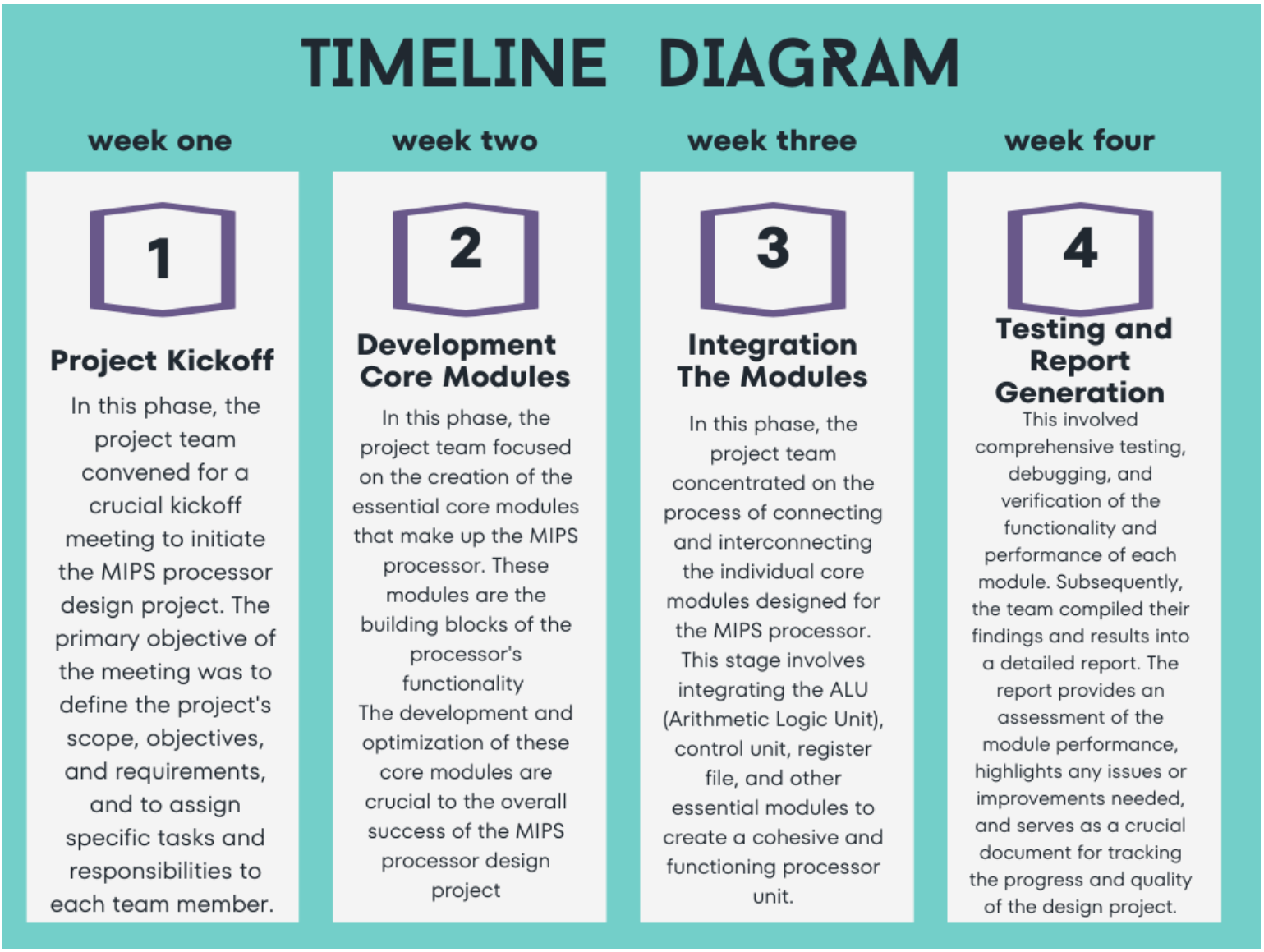
- **Single-Cycle Execution:** Our CPU is designed to execute instructions in a single clock cycle. This efficient one-cycle process involves fetching instructions, decoding them, executing operations, and writing results back to registers.
- **Control Unit:** We have implemented a control unit that generates 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.

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.
- **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.

1.8 The team's member responsibility



1.8.1 Omar AL-khasawneh (Leader)

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

1.8.2 Omar AL-salah

- Register file
- Control unit
- Make graph and image

1.8.3 Moayyad Abu Mallouh

- Instruction memory
- Data memory
- Mux between component

1.9 Organization of the rest of the documentation.

TABLE OF CONTENTS

01 VERILOG CODE
- PC.v
- ALU.v
- INS_MEM.v
- RAM.v
- Control_Unit.v
- Main.v
- AluControl.v
- SignExtend.v
- 5bit_Decoder.v
- 5bit_Mux.v

REFERENCES 02
- Instruction set Architecture.txt
- JoSDC_CPU_reference_Design_Guidelines.pdf

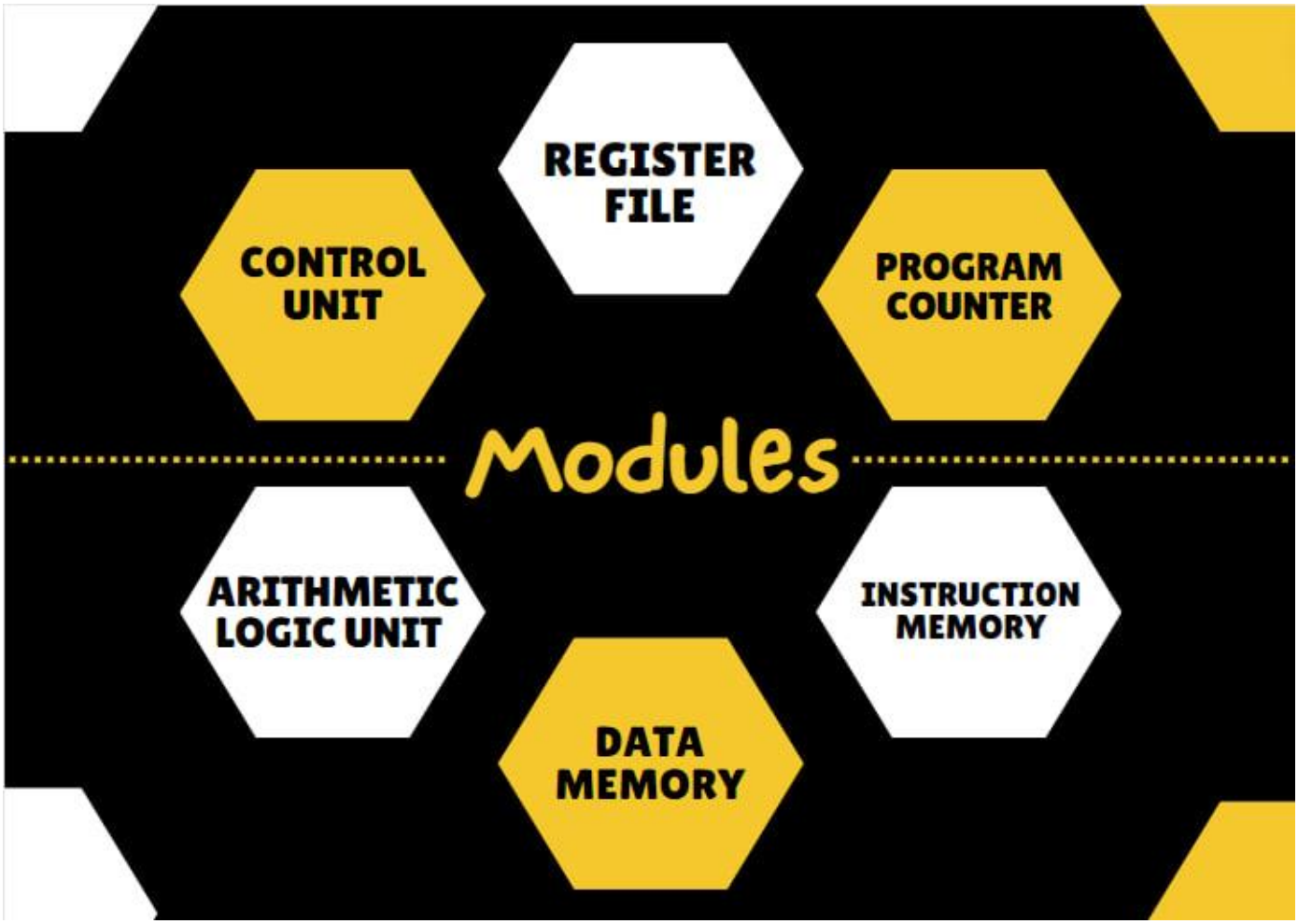
03 TEST BENCHES
- PC_tb.v
- ALU_tb.v
- INS_MEM_tb.v
- RAM_tb.v
- Control_Unit_tb.v
- Main_tb.v
- AluControl_tb.v
- SignExtend_tb.v
- 5bit_Decoder_tb.v
- 5bit_Mux_tb.v

SIMULATION 04
- Program Flow
- load Instruction Flow
- Graphs
- statistics

Design

1.10 Hardware Design and Implementation

We are used this component in our project:



- Describe the hardware design and components used.

1.10.1 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 takes input from registers or memory, processes it based on instructions, and produces output. It plays a key role in executing program instructions and performing mathematical calculations in a computer. The ALU's operations are fundamental to all computing tasks.

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.

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.

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.

1.11 mips instruction execution phase

1.12 Coding and Software Development

1.12.1 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.

1.12.2 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.

1.12.3 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.

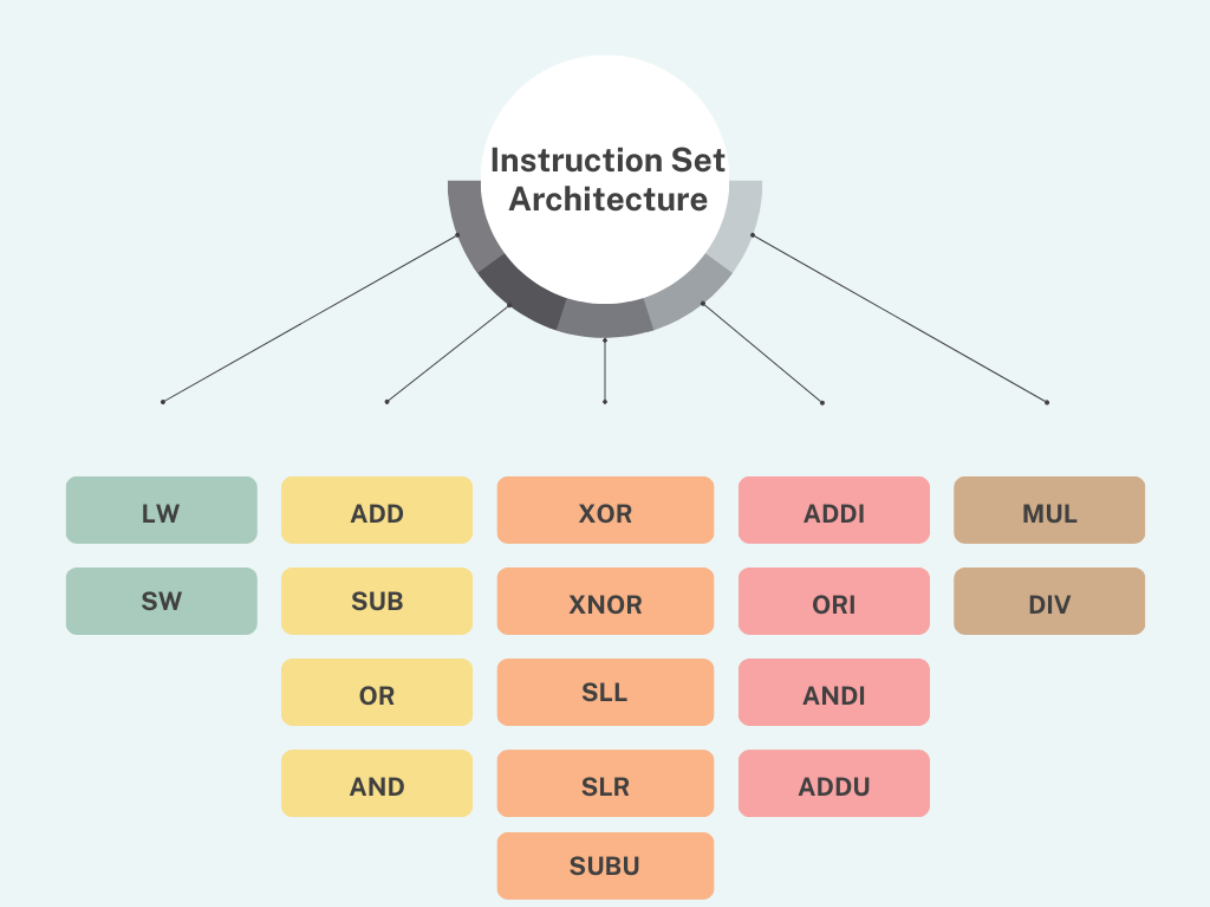
Testing: The CPU should be thoroughly tested to ensure that it meets all timing requirements.

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.

1.12.4 Instruction set architecture

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



Results

1.13 Program flow

1.13.1 Initial Data

```
//              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
```

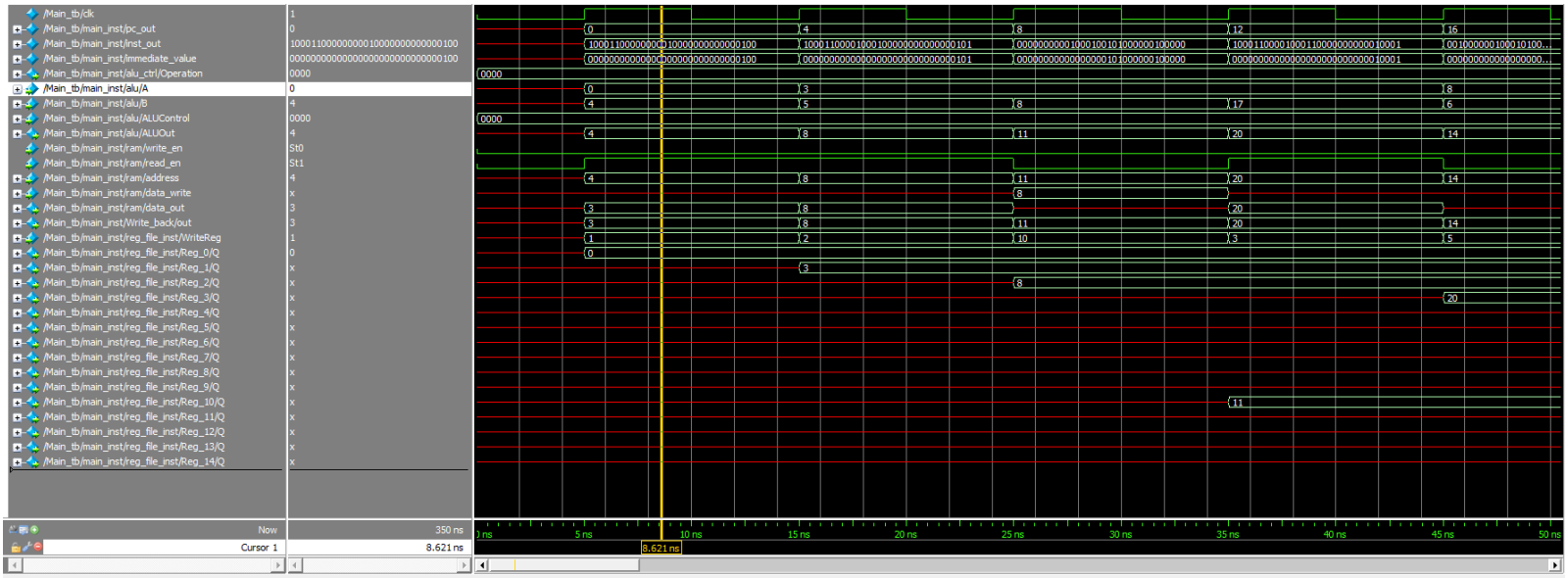
1.13.2 Programm instruction

```
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 reg 2 to 6 (8+6 =14) then store it in reg5 =14
inst_mem[5] = 32'b10001100010001000000000000000100;           //LW $4 , 4($2)  -> 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 reg2 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 reg1 (ans:3) then store it in reg8=3
inst_mem[9] = 32'b00000000001010100100000000100101;           //or     $9,$1,$10 -> or   the content of reg10 with reg1 (ans:11) then store it in reg9=11
inst_mem[10] = 32'b00000000110001000101100000100110;   //xor  $11,$6,$4->xor   the content of reg6 with reg4 (ans:7) then store it in reg11=7
inst_mem[11] = 32'b00000000010001100110000000100111;   //nor  $12,$2,$6->nor   the content of reg2 with reg6 (ans:-12) 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

end

/*
```

1.13.3 First cycle

Instructions:

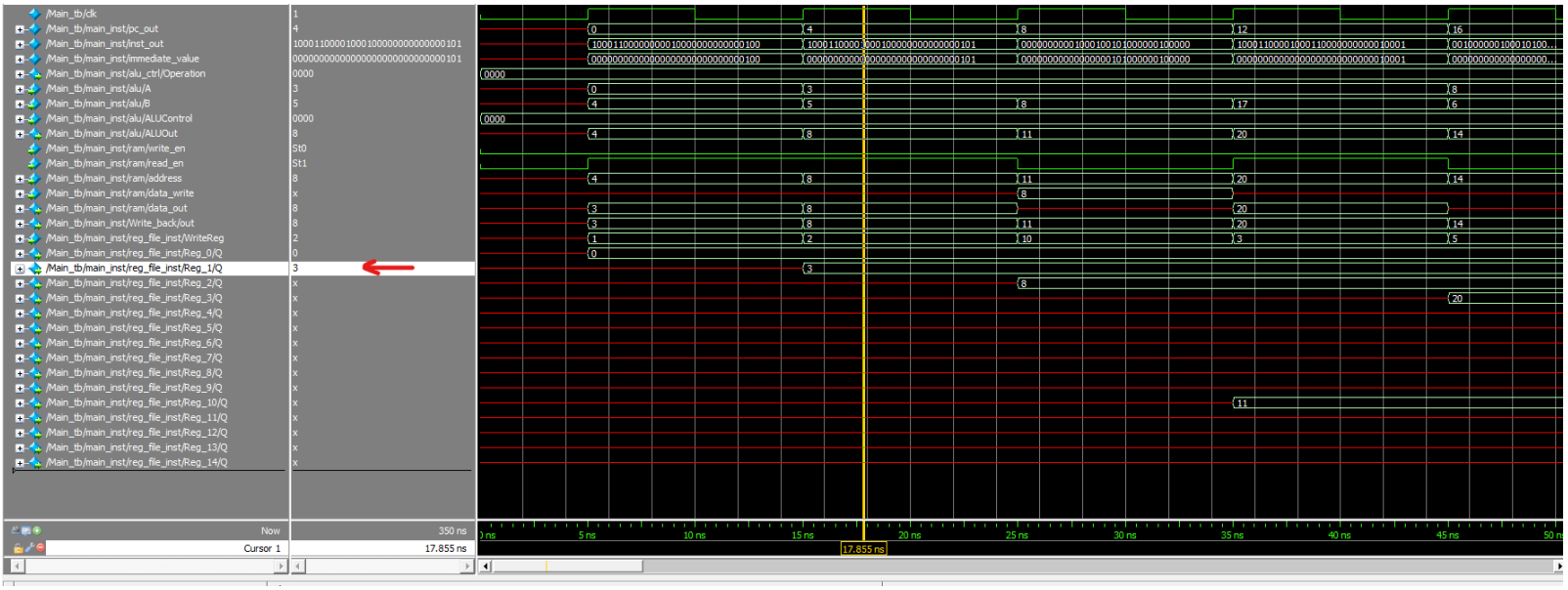
- **LW $1 , $4(0)**



1.13.4 Second cycle

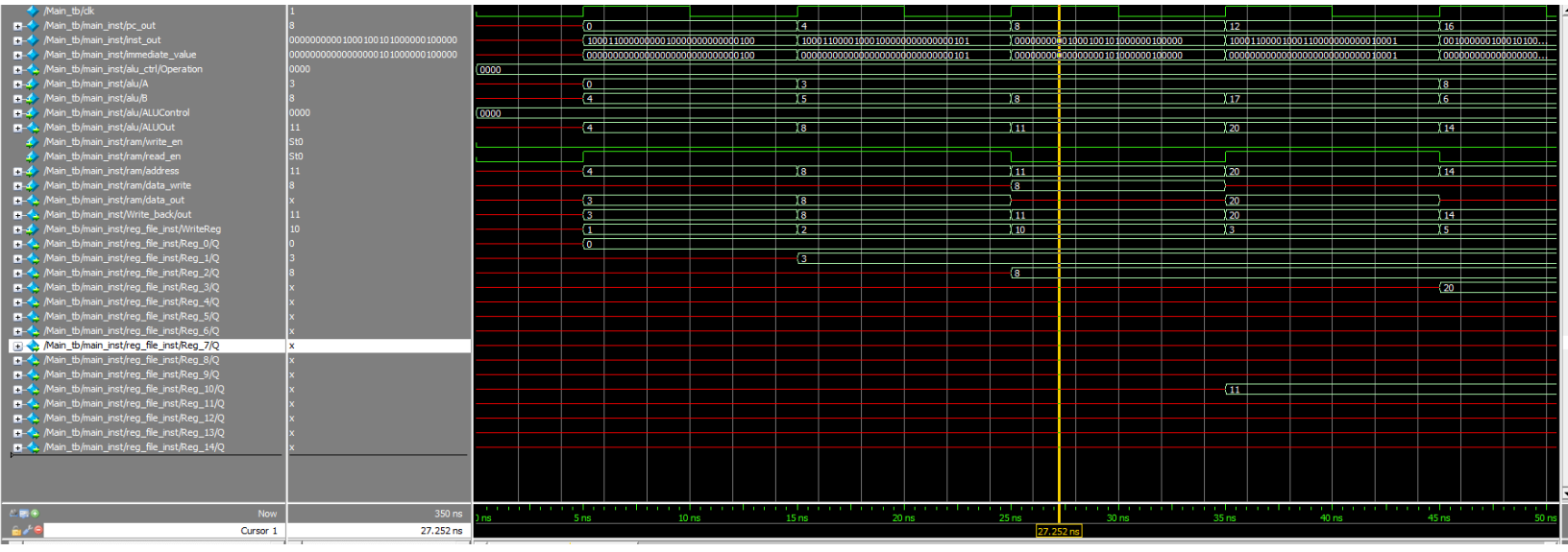
We have the instructions:

- **LW $1 , $4(0)**
- **LW $2 , $5(1)**



1.13.5 Third cycle

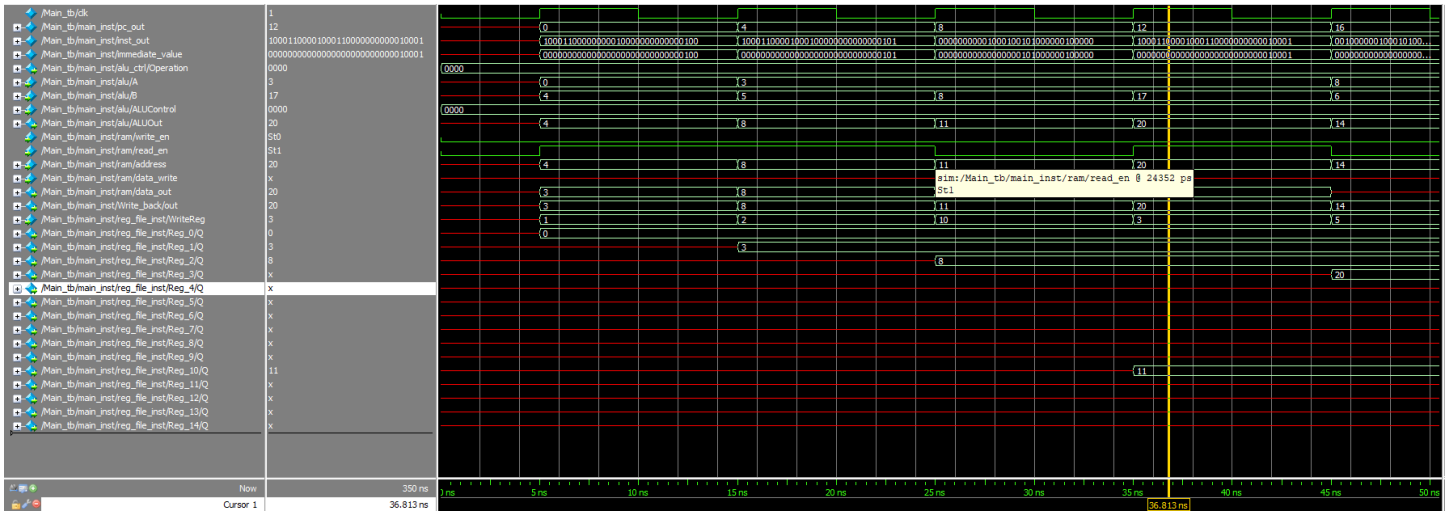
- **LW $1 , $4(0)**
- **LW $2 , $5(1)**
- **add $10,$1,$2**



1.13.6 Cycle four

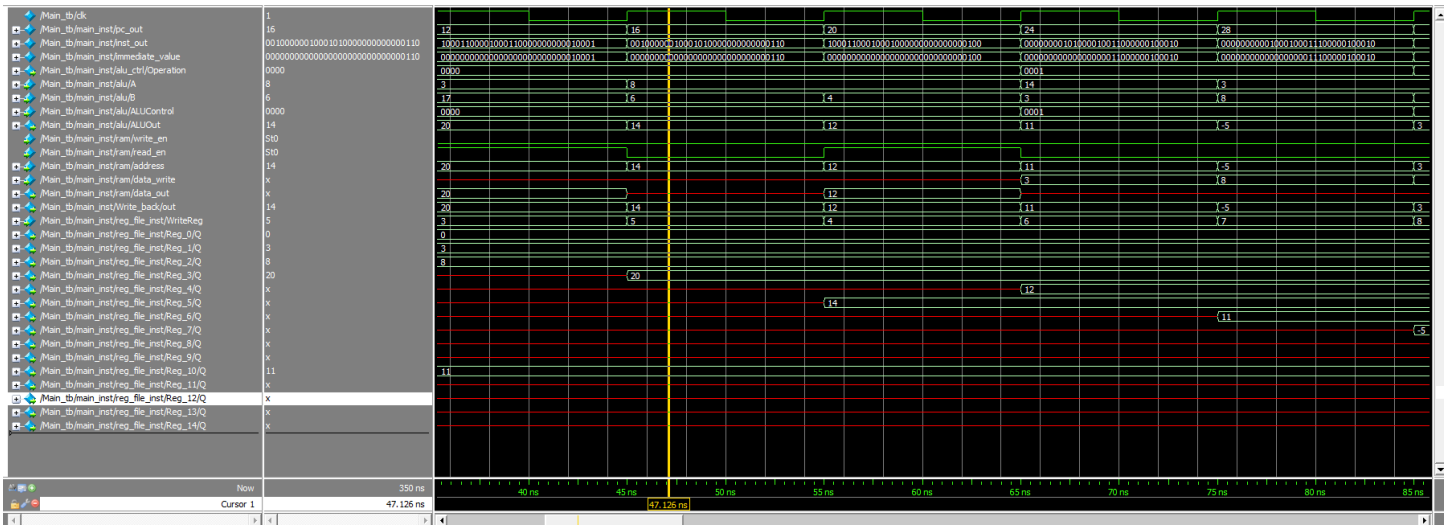
- **LW $1 , $4(0)**
- **LW $2 , $5(1)**
- **add $10,$1,$2**

- **LW $3 , $17(1)**



1.13.7 Cycle five

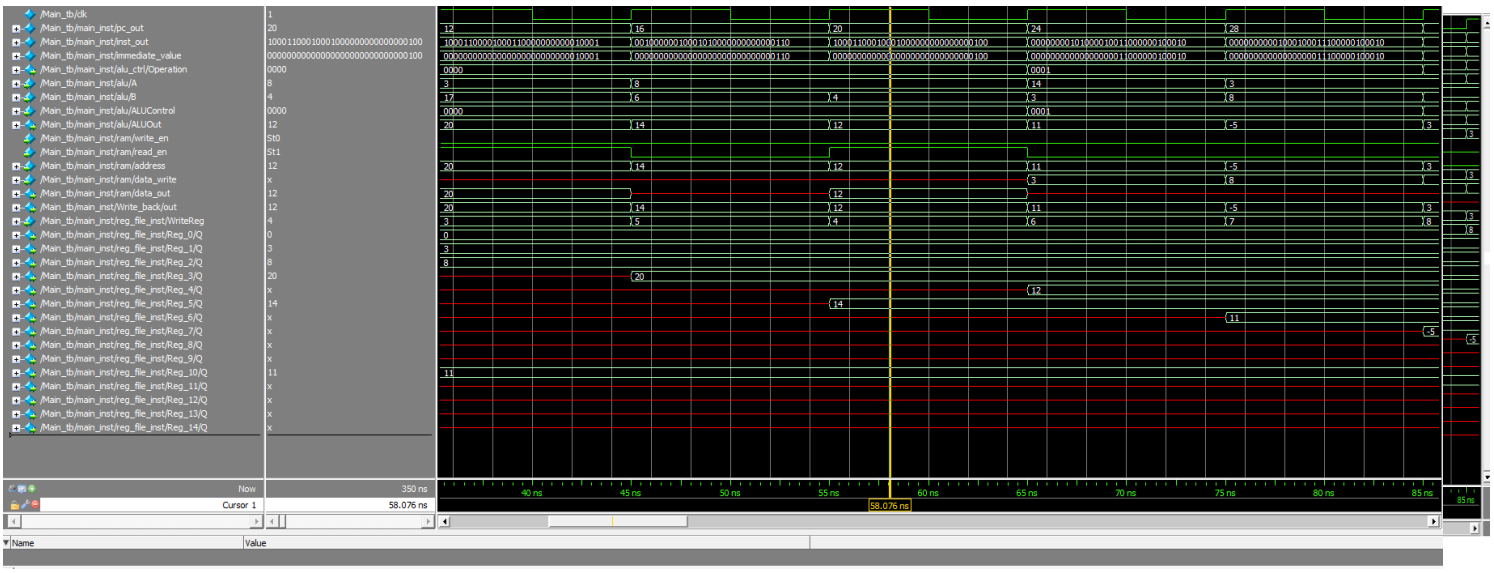
- **LW $1 , $4(0)**
- **LW $2 , $5(1)**
- **add $10,$1,$2**
- **LW $3 , $17(1)**
- **addi $5,$2,6**



1.13.8 Cycle six

- **LW $1 , $4(0)**
- **LW $2 , $5(1)**
- **add $10,$1,$2**

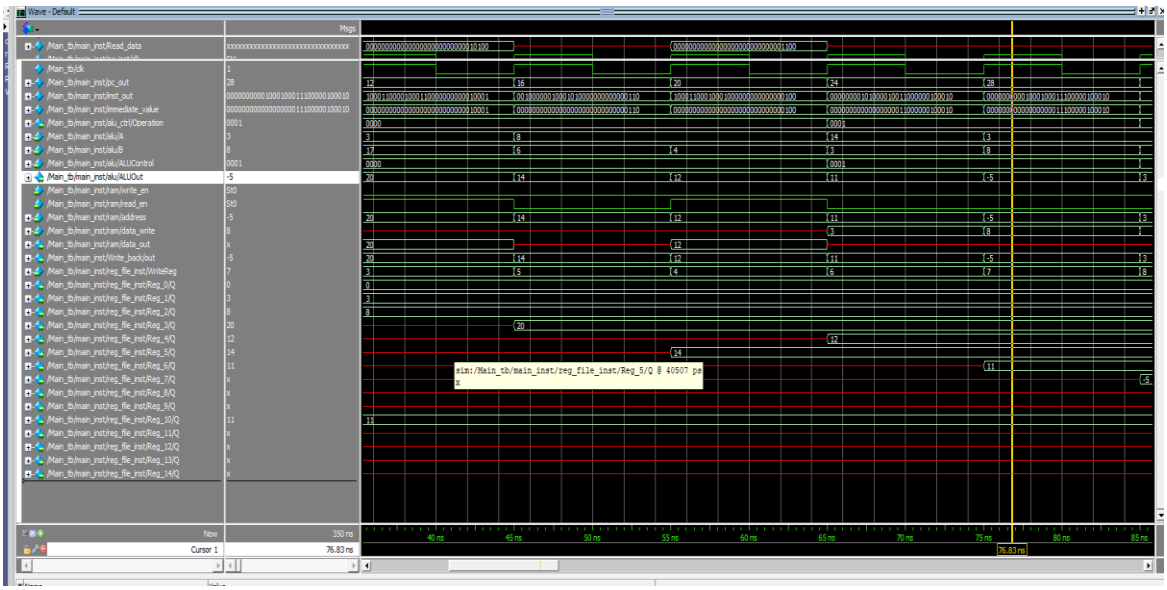
- **LW $3 , $17(1)**
- **addi $5,$2,6**
- **LW $4 , $4(3)**



1.13.9 Cycle seven

- **LW $1 , $4(0)**
- **LW $2 , $5(1)**
- **add $10,$1,$2**
- **LW $3 , $17(1)**
- **addi $5,$2,6**
- **LW $4 , $4(3)**
- **sub $6,$5,$1**

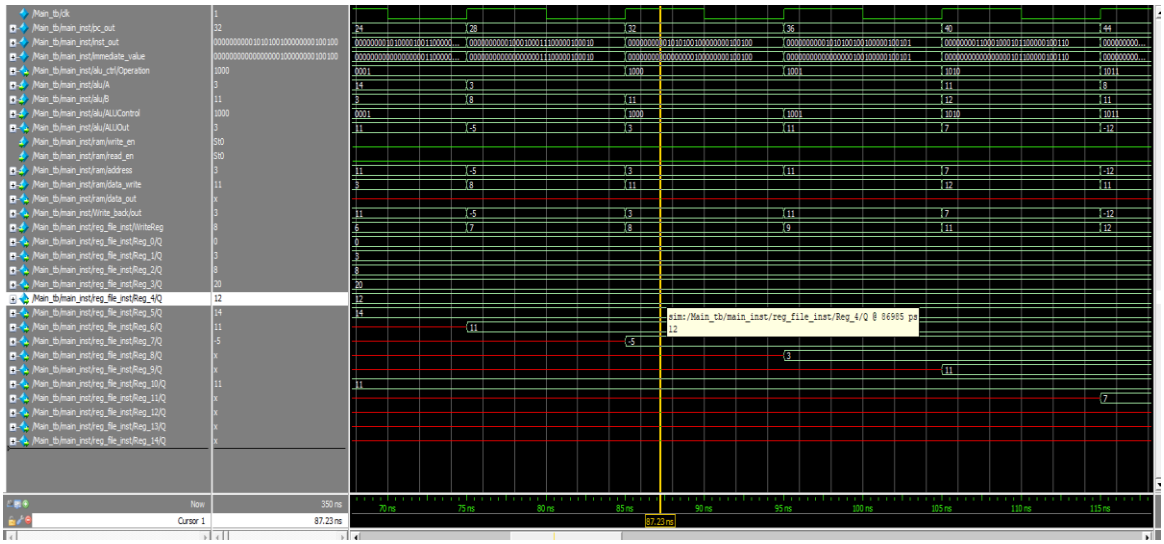
1.13.10 Cycle eight



- **LW $1 , $4(0)**
- **LW $2 , $5(1)**
- **add $10,$1,$2**
- **LW $3 , $17(1)**
- **addi $5,$2,6**
- **LW $4 , $4(3)**
- **sub $6,$5,$1**
- **sub $7,$1,$2**

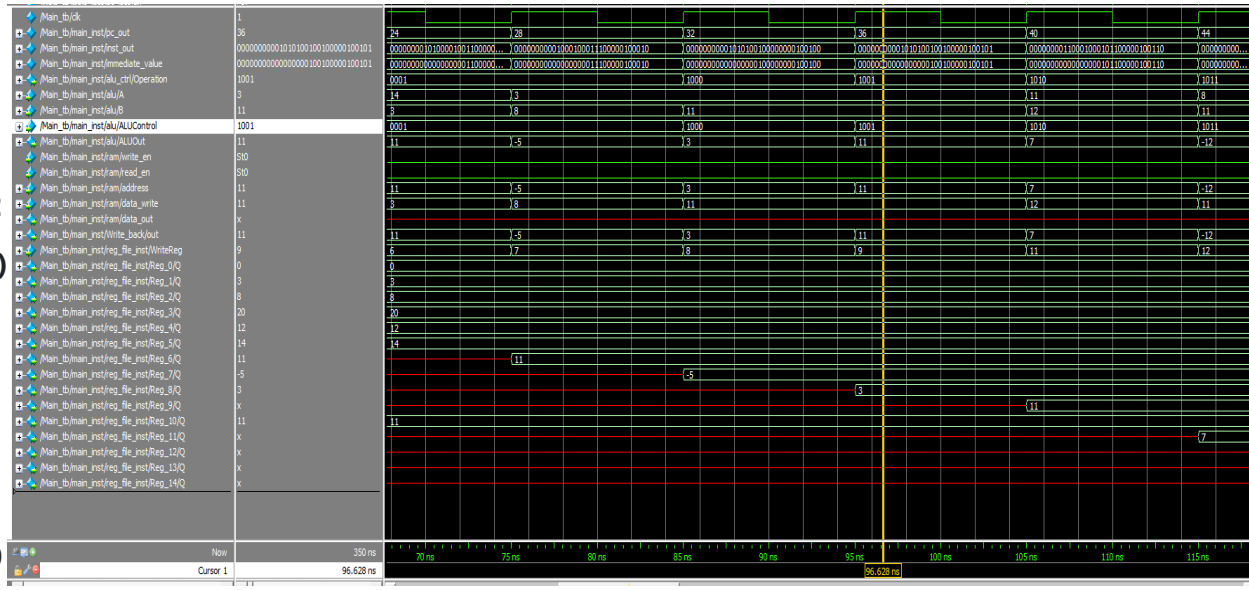
1.13.11 Cycle nine

- **LW $1 , $4(0)**
- **LW $2 , $5(1)**
- **add $10,$1,$2**
- **LW $3 , $17(1)**
- **addi $5,$2,6**
- **LW $4 , $4(3)**
- **sub $6,$5,$1**
- **sub $7,$1,$2**
- **and $8,$1,$10**



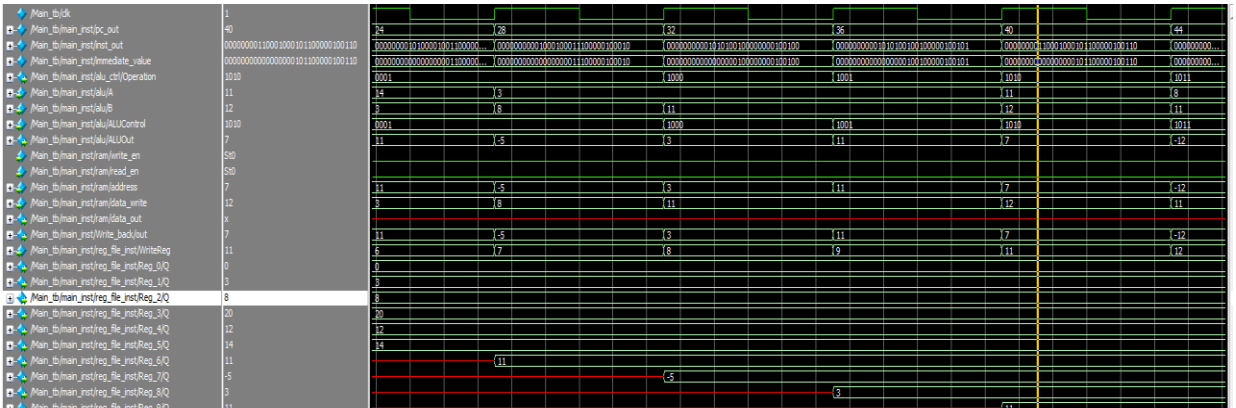
1.13.12 Cycle tin

- **LW $1 , $4(0)**
- **LW $2 , $5(1)**
- **add $10,$1,$2**
- **LW $3 , $17(1)**
- **addi $5,$2,6**
- **LW $4 , $4(3)**
- **sub $6,$5,$1**
- **sub $7,$1,$2**
- **and $8,$1,$10**
- **or $9,$1,$10**



1.13.13 Cycle eleven

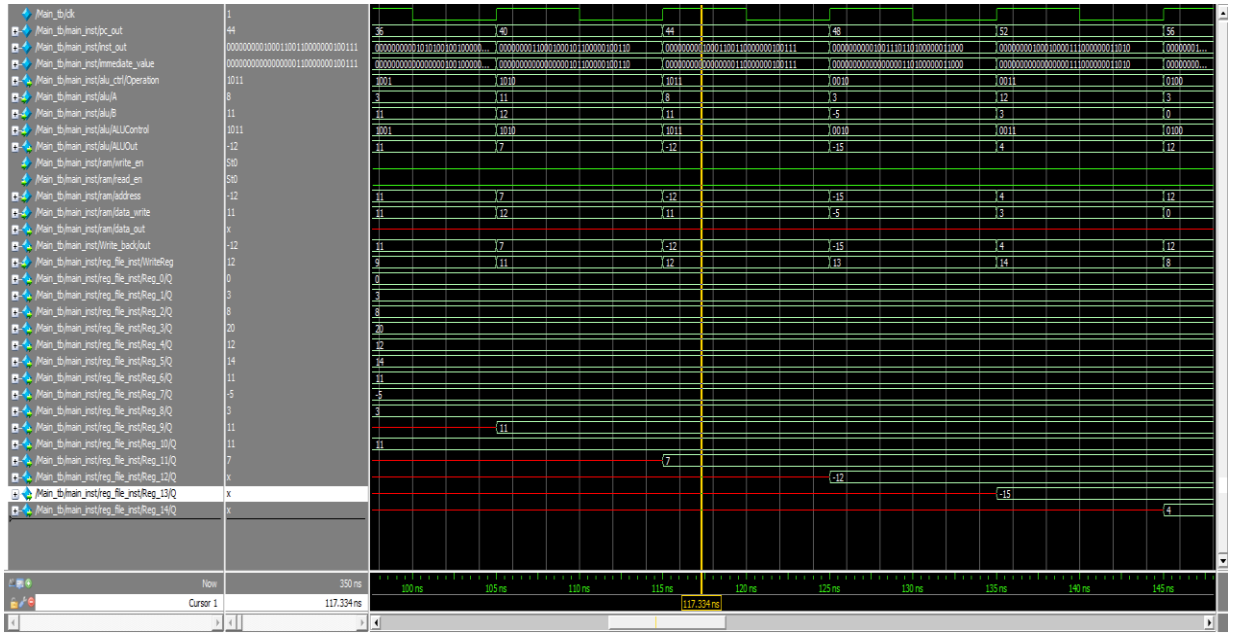
- **LW $1 , $4(0)**
- **LW $2 , $5(1)**
- **add $10,$1,$2**
- **LW $3 , $17(1)**
- **addi $5,$2,6**



- **LW $4 , $4(3)**
- **sub $6,$5,$1**
- **sub $7,$1,$2**
- **and $8,$1,$10**
- **or $9,$1,$10**
- **xor $11,$6,$4**

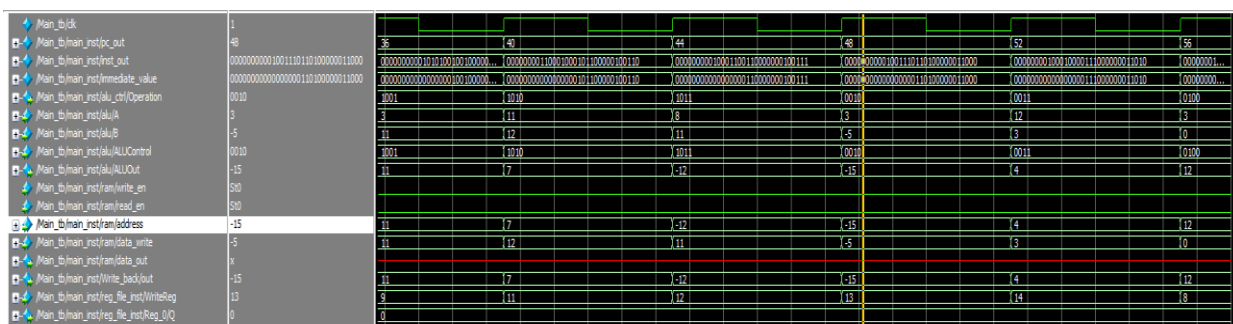
1.13.14 Cycle twelve

- **LW $1 , $4(0)**
- **LW $2 , $5(1)**
- **add $10,$1,$2**
- **LW $3 , $17(1)**
- **addi $5,$2,6**
- **LW $4 , $4(3)**
- **sub $6,$5,$1**
- **sub $7,$1,$2**
- **and $8,$1,$10**
- **or $9,$1,$10**
- **xor $11,$6,$4**
- **nor $12,$2,$6**



1.13.15 Cycle thirteen

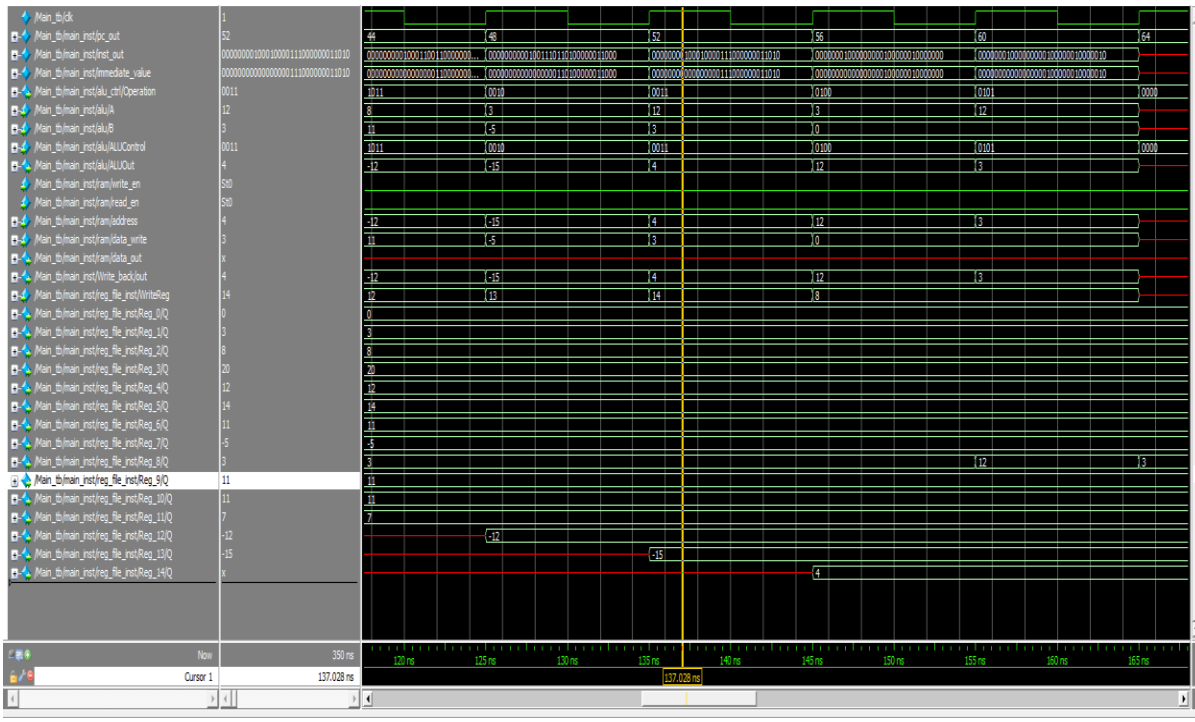
- **LW $1 , $4(0)**
- **LW $2 , $5(1)**
- **add $10,$1,$2**



- **LW $3 , $17(1)**
- **addi $5,$2,6**
- **LW $4 , $4(3)**
- **sub $6,$5,$1**
- **sub $7,$1,$2**
- **and $8,$1,$10**
- **or $9,$1,$10**
- **xor $11,$6,$4**
- **nor $12,$2,$6**
- **mul $13,$1,$7**

1.13.16 Cycle Fourteen

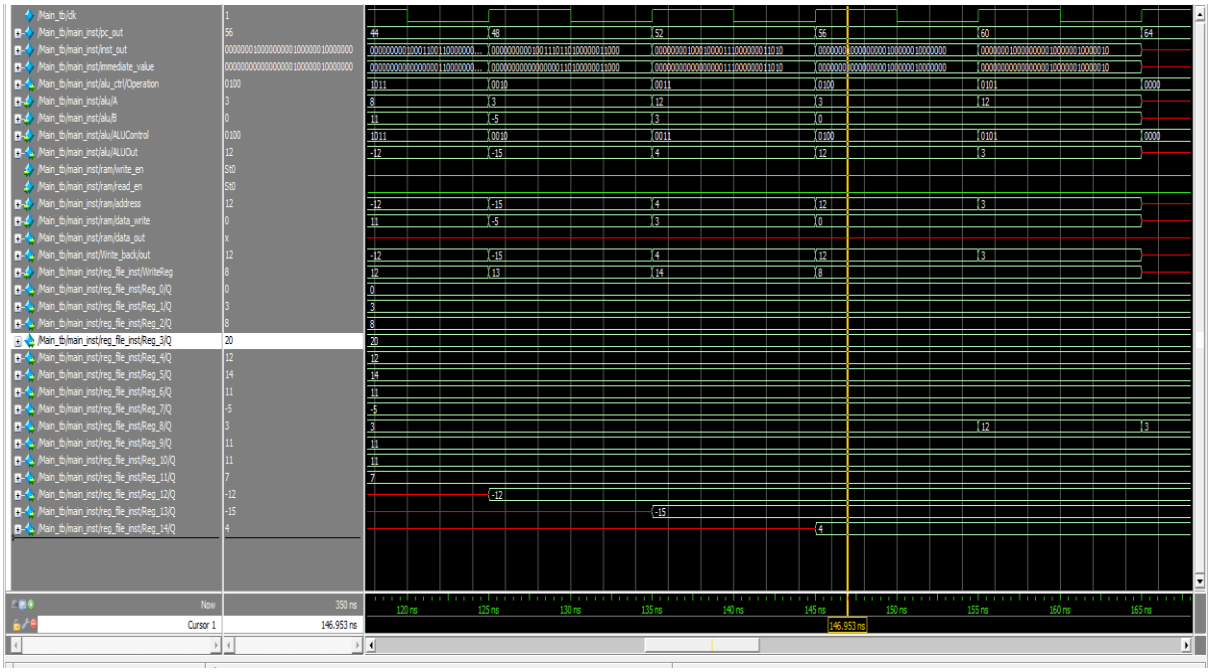
- **LW $1 , $4(0)**
- **LW $2 , $5(1)**
- **add $10,$1,$2**
- **LW $3 , $17(1)**
- **addi $5,$2,6**
- **LW $4 , $4(3)**
- **sub $6,$5,$1**
- **sub $7,$1,$2**
- **and $8,$1,$10**
- **or $9,$1,$10**
- **xor $11,$6,$4**
- **nor $12,$2,$6**
- **mul $13,$1,$7**
- **div $14,$4,$8**



1.13.17 Cycle Fiveteen

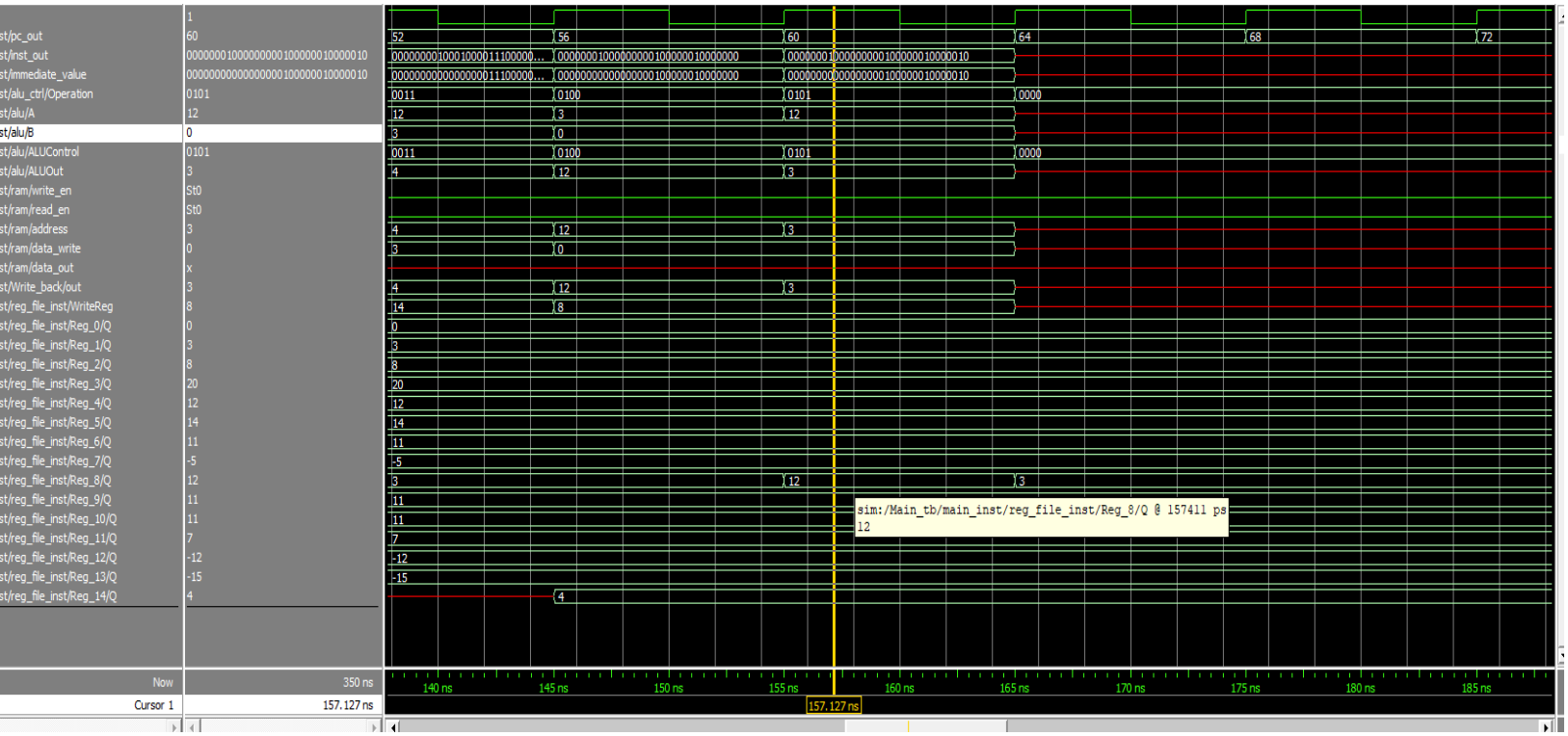
- **LW $1 , $4(0)**
- **LW $2 , $5(1)**

- **add $10,$1,$2**
- **LW $3 , $17(1)**
- **addi $5,$2,6**
- **LW $4 , $4(3)**
- **sub $6,$5,$1**
- **sub $7,$1,$2**
- **and $8,$1,$10**
- **or $9,$1,$10**
- **xor $11,$6,$4**
- **nor $12,$2,$6**
- **mul $13,$1,$7**
- **div $14,$4,$8**
- **sll $8 ,$0,$8**



1.13.18 Cycle sixteen

- **LW $1 , $4(0)**
- **LW $2 , $5(1)**
- **add $10,$1,$2**
- **LW $3 , $17(1)**
- **addi $5,$2,6**
- **LW $4 , $4(3)**
- **sub $6,$5,$1**
- **sub $7,$1,$2**
- **and $8,$1,$10**
- **or $9,$1,$10**
- **xor $11,$6,$4**
- **nor $12,$2,$6**
- **mul $13,$1,$7**
- **div $14,$4,$8**
- **sll $8 ,$0,$8**
- **slr $8 ,$0,$8**


st/pc_out
st/inst_out
st/immediate_value
st/alu_ctrl/Operation
st/alu/A
st/alu/B
st/alu/ALUControl
st/alu/ALUOut
st/ram/write_en
st/ram/read_en
st/ram/address
st/ram/data_write
st/ram/data_out
st/Write_back/out
st/reg_file_inst/WriteReg
st/reg_file_inst/Reg_0/Q
st/reg_file_inst/Reg_1/Q
st/reg_file_inst/Reg_2/Q
st/reg_file_inst/Reg_3/Q
st/reg_file_inst/Reg_4/Q
st/reg_file_inst/Reg_5/Q
st/reg_file_inst/Reg_6/Q
st/reg_file_inst/Reg_7/Q
st/reg_file_inst/Reg_8/Q
st/reg_file_inst/Reg_9/Q
st/reg_file_inst/Reg_10/Q
st/reg_file_inst/Reg_11/Q
st/reg_file_inst/Reg_12/Q
st/reg_file_inst/Reg_13/Q
st/reg_file_inst/Reg_14/Q
sim:/Main_tb/main_inst/reg_file_inst/Reg_8/Q @ 157411 ps
Now
Cursor 1
350 ns
157.127 ns
140 ns
145 ns
150 ns
155 ns
160 ns
165 ns
170 ns
175 ns
180 ns
185 ns
157.127 ns

1.14 Clock testing

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

MIPS Processor Clock Testing and Optimal Cycle Time Evaluation

CLOCK	PERIOD	FREQUENCY	RAISE	FALL	SLACK	TNS	ACHIEVED
clk 1	2 ns	500 MHZ	0.00	1.00	- 2.00	-2.00	✗
clk 2	3.5 ns	285.71 MHZ	0.00	1.75	-0.50	-0.50	✗
clk 3	4 ns	250 MHZ	0.00	2.00	0.00	0.00	✓
clk4	5 ns	200 MHZ	0.00	2.50	1.00	0.00	✓

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.

1.14.1 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.

1.14.2 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