SJSU SAN JOSÉ STATE UNIVERSITY

Department of Electrical Engineering (Spring25)

EE271: Digital System Design and Synthesis

PROJECT REPORT
16 BIT RISC-V PROCESSOR

Submitted to :

Prof. Binh Le

Submitted By:

TEAM 2

Anish Sathaye (017900662)

Sai Preeth Aduwala (017506138)

Soumya Guruvenkatesh Katti (018280431)

Siri Simpana Tiptur Shashidharamurthy (018282472)

Table of Contents

I. Introduction

This project focuses on the design and implementation of a custom 16-bit RISC-V processor, adhering to a simplified subset of the RISC-V instruction set architecture (ISA). RISC-V, an open-source and modular ISA, is widely used in academic and industrial settings for its flexibility, extensibility, and simplicity. Although the official RISC-V standard typically defines 32-bit, 64-bit, and 128-bit versions, this project adapts the ISA to a 16-bit architecture for educational purposes. This modification allows for a compact and manageable design while preserving the core principles of RISC-V.

The processor operates on a 16-bit data path, meaning the instruction and register widths are 16 bits. The design is modularly constructed using Verilog HDL (Hardware Description Language), enabling individual testing and integration of each component. The main modules include the ALU (Arithmetic Logic Unit), register file, control unit, datapath, program counter, instruction memory, RAM, sign extension unit, and various multiplexers and latches. Each module performs a specific role in the instruction execution cycle and contributes to the overall functionality of the processor.

The instruction memory holds the program instructions in binary format, which are fetched based on the current value of the program counter. The control unit plays a critical role by decoding each instruction and generating the necessary control signals to guide the data flow throughout the processor. The datapath connects all the modules and facilitates instruction execution. The ALU handles essential arithmetic and logic operations such as addition, subtraction, AND, OR, and comparison, which are central to most computations.

The register file contains general-purpose registers that temporarily hold data during program execution. The sign extension unit ensures that immediate values are correctly interpreted and aligned with the 16-bit data path. The program counter is responsible for tracking the address of the next instruction to execute and updates based on control signals, supporting both sequential execution and branching operations. Multiplexers are employed to choose between different data inputs, improving flexibility in control flow and instruction handling. Latches are used to store intermediate values and maintain state across clock cycles.

The processor supports a subset of RISC-V instructions, including R-type (register-register), I-type (immediate), and basic branching instructions. While it does not support the full range of RISC-V functionality, the processor includes enough instruction types to execute basic programs and demonstrate core CPU operations. The execution model is simplified, using a single-cycle or multi-cycle approach depending on design decisions and implementation constraints. Advanced features such as pipelining, hazard detection, and interrupts are intentionally omitted to maintain simplicity and focus on foundational design concepts.

One of the major design challenges was adapting the 32-bit RISC-V ISA to a 16-bit word size. This required efficient encoding of opcodes, register addresses, and immediate values to fit within

16 bits. Careful consideration was given to control signal synchronization, clocking, and data path alignment to ensure reliable and predictable operation. Instruction testing was carried out using a custom set of manually written programs, with simulation performed through Verilog testbenches to verify output accuracy and logic behavior.

Overall, this project demonstrates a solid understanding of computer architecture principles and hardware design using Verilog. The modular approach not only facilitates easier debugging and testing but also provides a scalable framework for future enhancements. These could include support for pipelining, larger instruction sets, memory-mapped I/O, or cache integration. The 16-bit RISC-V processor thus serves as both a functional CPU design and an effective educational platform, bridging theoretical learning and hands-on digital system development.

II. Hardware and Software Tools

i. Hardware: Nexys A7 100T FPGA Development Board

The design of the ALU system was implemented on the **Nexys A7-100T** FPGA development board, which is a versatile and powerful platform designed for digital circuit development. The Nexys A7 features the **Xilinx Artix-7** FPGA, providing ample resources for the implementation of various logic and processing systems. With **16K logic cells** and **100,000 logic elements**, the Nexys A7-100T board offers the processing power required for complex digital designs. Additionally, it includes:

- **1GB of DDR3 SDRAM** for data storage and processing
- **4,000,000 logic gates** to accommodate large designs
- **A large set of I/O peripherals**, including switches, LEDs, push-buttons, and 7-segment displays, enabling real-time feedback and interaction with the system

The board's flexibility, combined with its extensive I/O capabilities, makes it ideal for prototyping and testing FPGA designs, such as the ALU-based operations and display system implemented in this project.

ii. Software: Vivado 2024.2

For synthesizing, simulating, and deploying the design, **Xilinx Vivado 2024.2** was used. Vivado is a comprehensive suite of development tools for FPGA design, offering a complete environment for digital design, verification, and implementation. The Vivado suite includes:

- **RTL Design**: Vivado provides tools for writing and synthesizing Verilog code, making it easy to design and simulate complex systems.

- **Simulation**: Vivado supports both behavioral and post-synthesis simulation, allowing verification of the functionality of the design in a controlled environment.
- **Synthesis and Implementation**: Vivado's powerful synthesis tools translate RTL designs into FPGA-optimized netlists and automatically map them to available resources on the target FPGA.
- **Programming and Debugging**: Vivado offers support for downloading the bitstream to the FPGA board, enabling real-time testing and debugging of the design on the hardware.

In this project, Vivado was used to write and synthesize the Verilog code for the ALU and associated modules, simulate the functionality of the design, and ultimately deploy it to the Nexys A7 board. The tool provided a seamless workflow for the entire design process, from initial development to final testing on hardware.

III. Design Overview

The processor design utilizes a modular architecture where each component performs specific tasks to ensure the processor operates efficiently. The main goal of the design is to execute a range of basic operations such as arithmetic and logical computations, memory access, and program control. Below is an overview of the design:

Modular Architecture

The processor's design consists of several key components:

- **Main Control Unit (MCU)**: Generates control signals to govern the operation of the entire system.
- **Arithmetic and Logic Unit (ALU)**: Executes various arithmetic and logical operations based on the control signals and operands.
- **Register File**: Stores and retrieves data for execution.
- **Data Memory**: Handles read and write operations to memory.
- **Branch Adder**: Computes target addresses for branch instructions.
- **Display Interface**: Converts results into a human-readable format for display.

These components work together to provide a functioning processor capable of executing a variety of operations, including those necessary for basic program flow, data manipulation, and memory access.

i. Control Flow and Data Paths

The processor uses a control flow mechanism, where the **Main Control Unit** decodes the opcode from each instruction and generates a set of control signals. These signals are then passed to different components, such as the **ALU**, **Register File**, and **Memory**, to dictate the operations. The control signals direct the ALU to perform operations like addition, subtraction, logical operations, and comparisons. Similarly, the control unit ensures data is written to registers or memory when required.

ii. ALU and Operations

The **ALU** is responsible for executing arithmetic and logical operations. The **ALU Control** unit determines which operation the ALU should perform based on the decoded instruction and the control signals. Supported operations include addition, subtraction, AND, OR, XOR, shifts, and comparisons. The **ALU** takes in two operands (either from registers or immediate values) and computes the result.

iii. Register File and Memory

The **Register File** stores the processor's data, supporting both read and write operations. The **Data Memory** module handles read and write to the memory, enabling load and store operations. The **Register File** and **Data Memory** work together to ensure that data is moved between registers, memory, and the ALU correctly, as dictated by the control signals.

iv. Branching and Program Counter (PC) Updates

Branching instructions involve calculating a new address based on the current PC and an offset provided by the instruction. The **Branch Adder** computes this address, and the PC is updated accordingly if a branch condition is met. This mechanism allows for conditional and unconditional branching in the program flow, supporting loops, function calls, and jumps.

v. Display Mechanism

To visualize the output of operations, the system uses a **7-Segment Display** to show the result of computations. The **BCD to 7-Segment Conversion** module converts the 16-bit result into Binary Coded Decimal (BCD) form. The **Display Interface** drives the 7-segment display to show each digit sequentially, allowing for the user to view the result of ALU operations in real time. This component ensures that the output is presented in a readable format, even for multi-digit numbers.

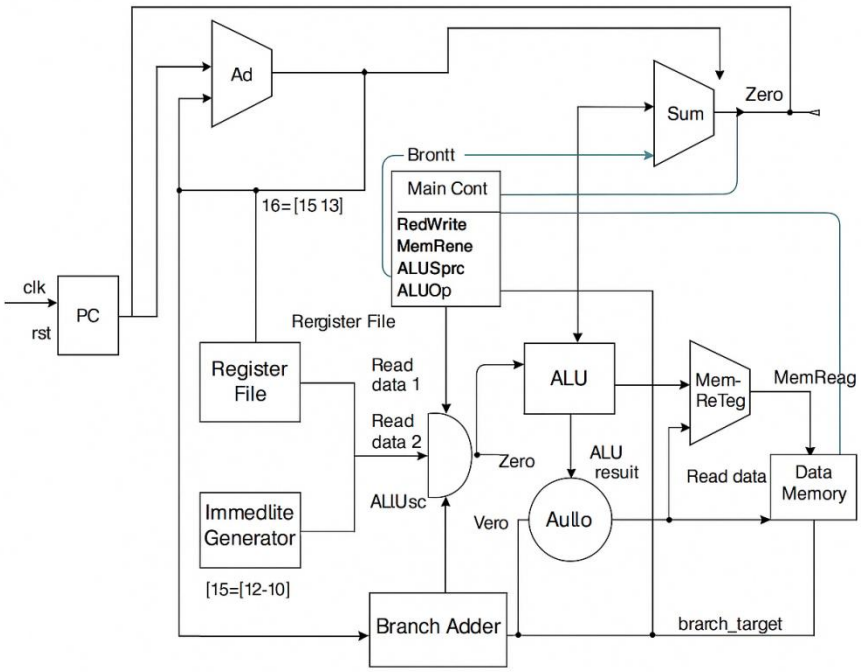
vi. Clocking and Reset Mechanisms

A single clock signal synchronizes all components in the design. Every operation—whether reading from memory, performing an ALU computation, or writing back to registers—is timed based on the clock cycle. A reset mechanism ensures that the processor starts in a known, initialized

state, with all registers and memory set to default values. This provides a clean slate for execution and ensures proper system initialization.

vii. Control Signals

The processor relies on several control signals that are generated by the **Main Control Unit** to manage various operations. These include signals for **ALU operations**, **memory read/write operations**, **register file writes**, and **branching decisions**. The control signals ensure that each part of the processor knows what operation to perform at each clock cycle, ensuring the correct execution of instructions.



16-bit-Bit RISC-V top module processr op module

IV. Working

The system is designed to perform basic Arithmetic and Logic Unit (ALU) operations, handle memory operations, and display the results using a 7-segment display. The architecture includes several key modules that work together to process instructions, perform computations, and display the results. Each module operates in tandem with others, ensuring correct data flow through the system. The modules and their interconnections are described below.

i. Main Control Unit

The **Main Control Unit** is responsible for decoding the instruction opcode and generating control signals. These control signals regulate the entire processor's behavior, including whether to perform arithmetic operations, read/write memory, or branch. Based on the opcode, the unit asserts the following control signals:

- **RegWrite**: Enables writing to the register file.
- **MemRead**: Allows reading data from memory.
- **MemWrite**: Enables writing to memory.
- **MemToReg**: Determines whether data written back to the register is from memory or the ALU.
- **ALUSrc**: Decides whether the second operand for the ALU is from a register or an immediate value.
- **Branch**: Controls the branch operation for conditional jumps.
- **ALUOp**: Specifies the ALU operation (e.g., ADD, SUB).

This unit is essential for coordinating the processor's operation, translating the opcode into control signals that direct subsequent operations.

ii. ALU Control

The **ALU Control** module is responsible for translating the control signals and instruction-specific information into precise ALU operations. It takes as input the funct3, funct7, and ALUOp signals, which define the type of operation to perform (e.g., ADD, SUB, AND). This module generates a 4-bit signal, ALUcontrol_Out, that determines the exact operation the ALU should perform. By decoding the instruction, the ALU Control ensures that the correct arithmetic or logical operation is executed.

iii. ALU (Arithmetic and Logic Unit)

The **ALU** module is the heart of the arithmetic operations and logical operations within the processor. It takes two 16-bit inputs, A and B, and performs an operation based on the ALU control signal. The ALU supports a variety of operations such as addition, subtraction, AND, OR, XOR, shifts (logical and arithmetic), and comparisons. The result of the operation is stored in Result, and if the result is zero, the Zero flag is set. The ALU ensures that the desired mathematical or logical operation is carried out based on the current instruction.

iv. Immediate Generator

The **Immediate Generator** is responsible for extracting and sign-extending immediate values from instructions. These values are needed for I-type, Load, Store, and Branch instructions. The generator examines the instruction's format and produces a 16-bit sign-extended immediate value, which is then used as an operand in the ALU or for memory access.

v. 2-to-1 Multiplexer

A **2-to-1 Multiplexer (MUX)** is used extensively throughout the processor to select between two data inputs. For example, when choosing between data from the ALU or data from memory, the MUX is controlled by the select signal. If select is 0, the MUX passes the ALU result; if select is 1, the MUX passes the memory data. This selection mechanism is critical for enabling the appropriate data path at any given time.

vi. Data Memory

The **Data Memory** module is responsible for managing data storage in the processor. It supports both read and write operations, depending on the MemRead and MemWrite control signals. The memory is initialized to zero upon reset, with specific data values stored at predefined addresses. When reading from memory, the value at the specified address is provided as the read_data. When writing to memory, the data is stored at the given address if the MemWrite signal is active.

vii. 2-to-1 Multiplexer for Data Memory

Another **2-to-1 Multiplexer** is used in conjunction with the Data Memory to select whether the data to be written back to the register comes from the ALU or from memory. The select signal controls this choice, allowing the processor to write the appropriate result to the register file.

viii. Branch Adder

The **Branch Adder** is used to calculate the target address for a branch instruction. By adding an offset to the current Program Counter (PC) value, the Branch Adder computes the target address for a branch operation. The target address is then used to control the flow of the program during conditional jumps.

ix. BCD to 7-Segment Display Conversion

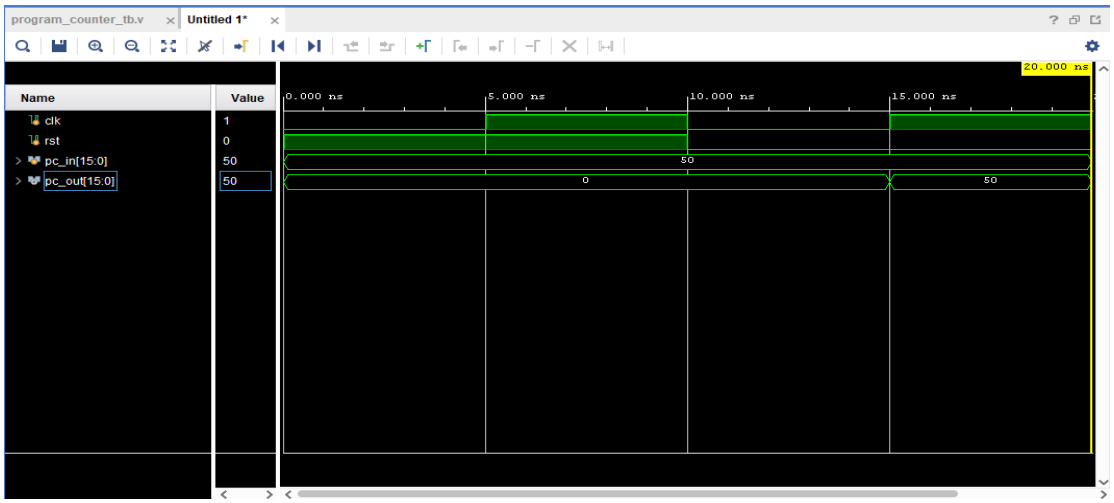
The **BCD to 7-Segment Display Conversion** module is responsible for converting the result of an ALU operation (in binary format) into a BCD (Binary Coded Decimal) representation. This BCD value is then used to drive the 7-segment display. The conversion is achieved using a shift-and-add-3 algorithm, which ensures that the binary value is converted correctly to its decimal equivalent, suitable for display.

x. 7-Segment Display

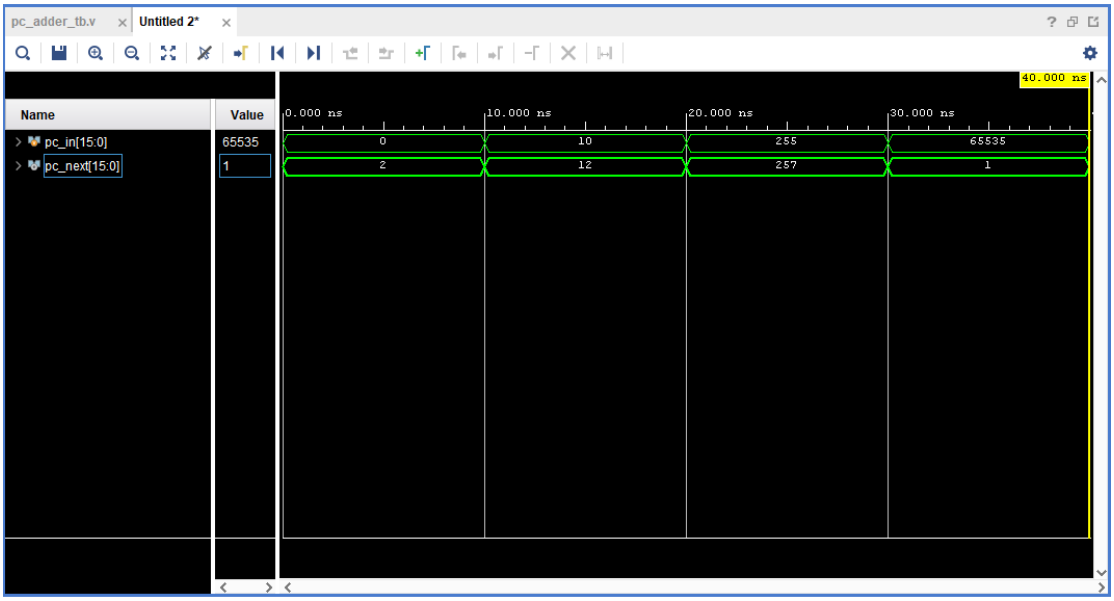
The **7-Segment Display** module is responsible for displaying the results of computations on a 5-digit 7-segment display. It receives the BCD value from the previous conversion module and uses it to drive each of the 7-segment displays. The clk signal controls the digit switching, ensuring that each of the 5 digits is displayed in a cycle. The module encodes the BCD digits to their corresponding 7-segment patterns and activates the correct digit at the appropriate time. This allows the result of the ALU operation to be visualized on the 7-segment display in real-time.

V. Simulation Results

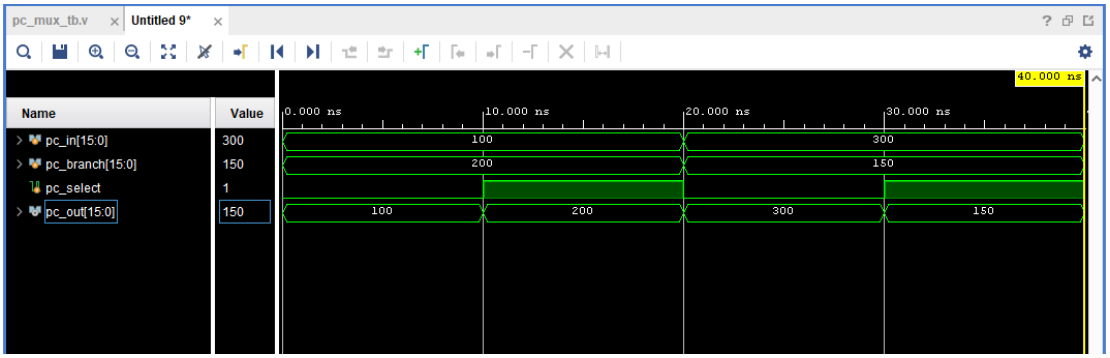
Program Counter:



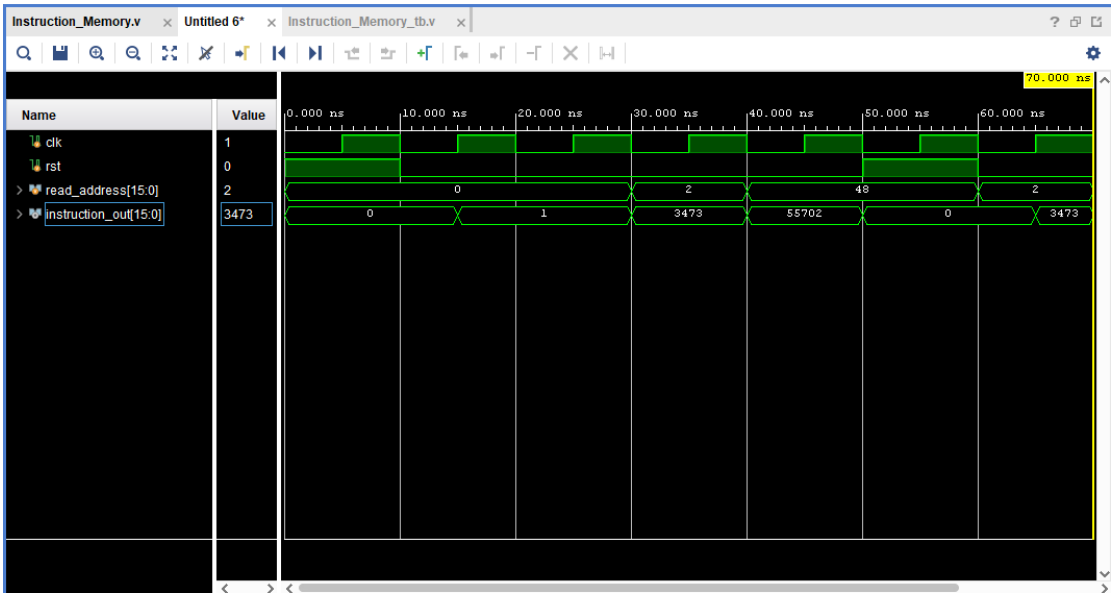
Program Counter Adder:



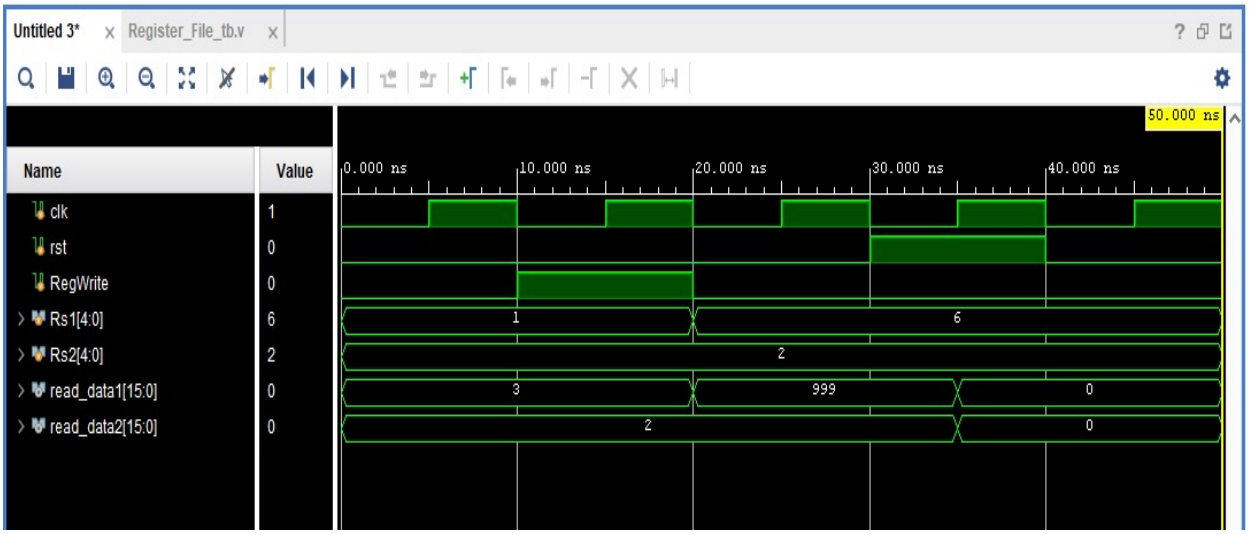
Program Counter MUX:



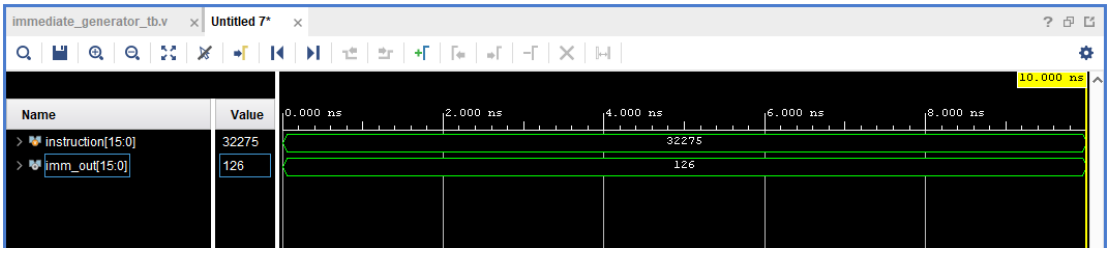
Instruction Memory:



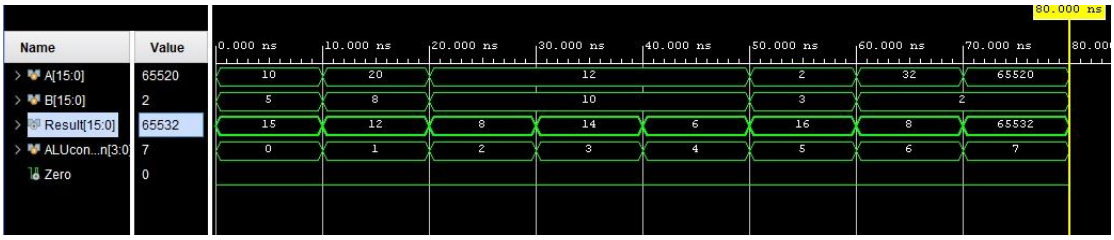
Register File:



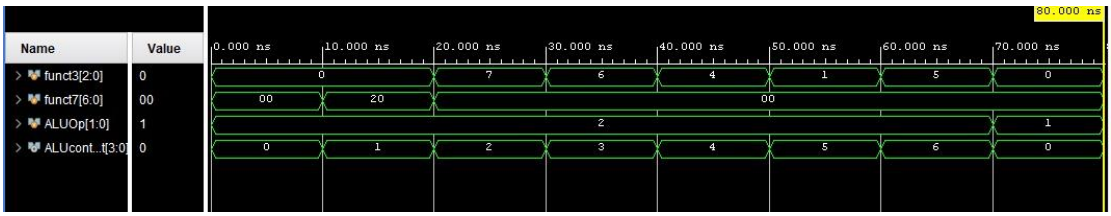
Immediate Generator:



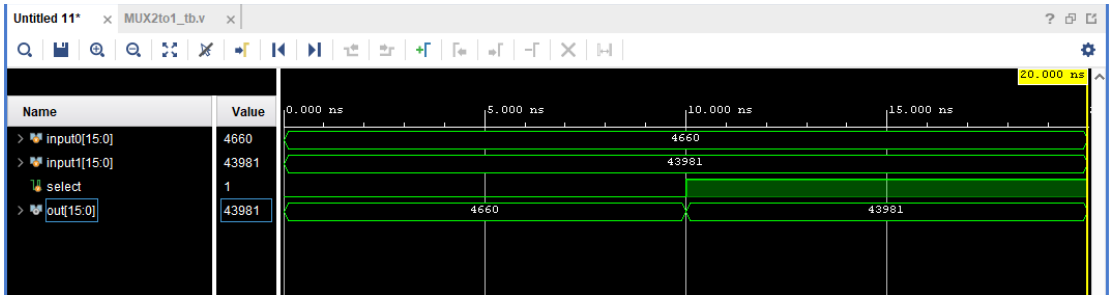
ALU:



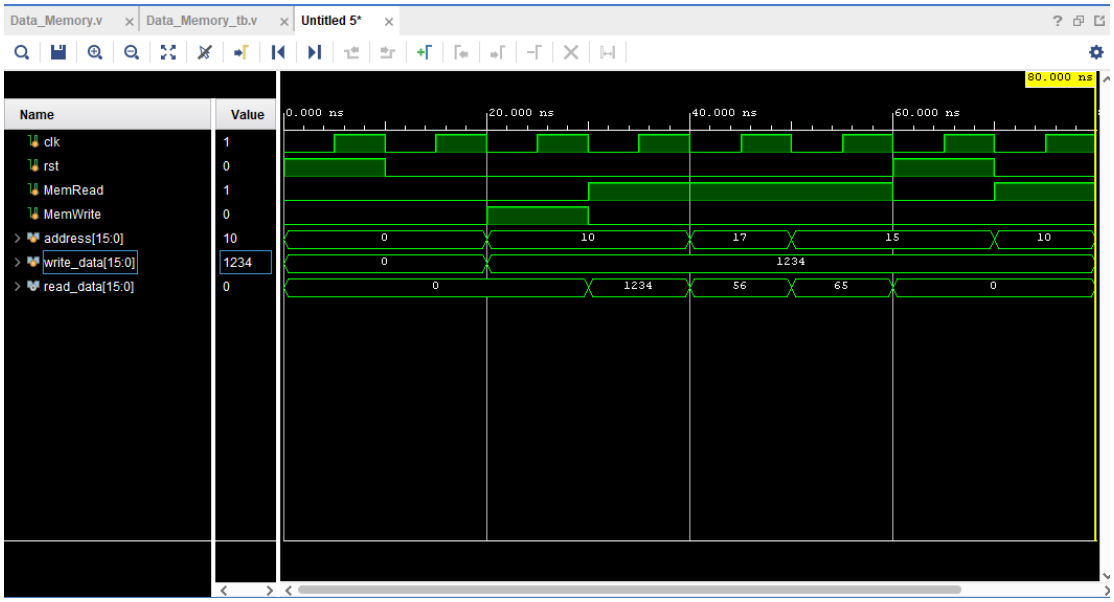
ALU Control:



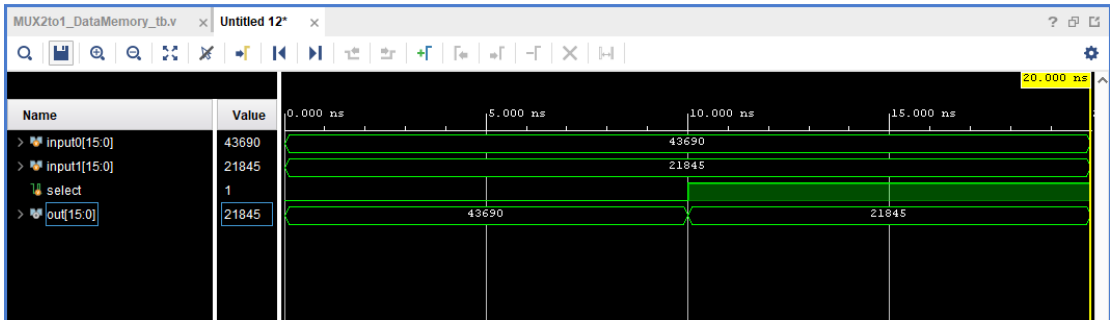
MUX 2x1:



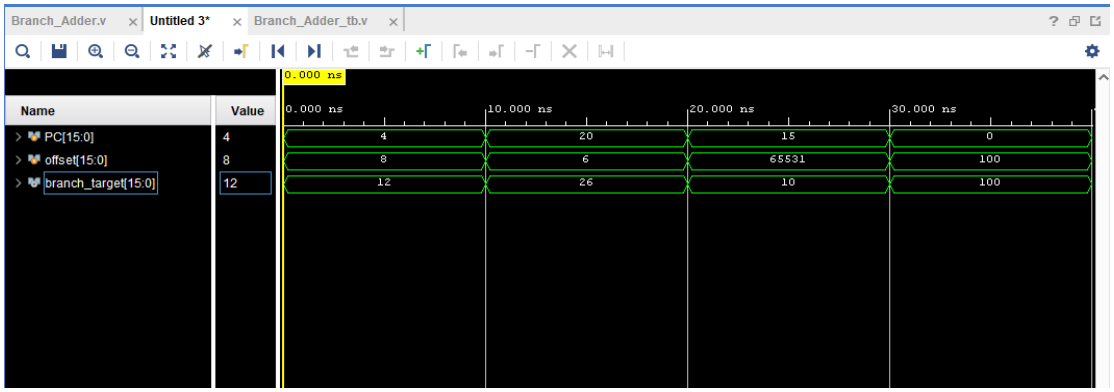
Data Memory:



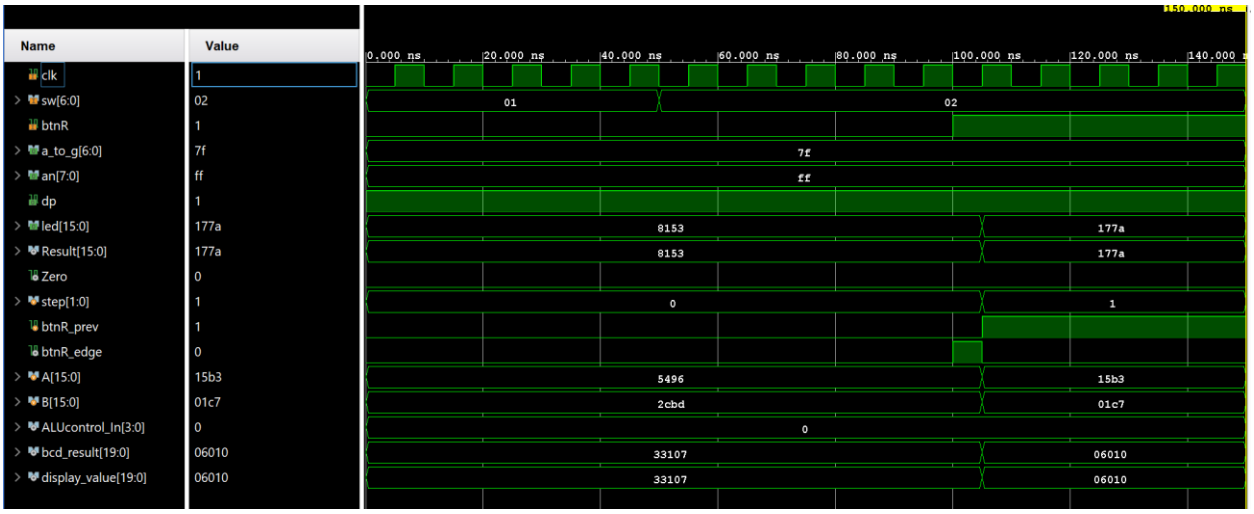
MUX 2x1 Data Memory:



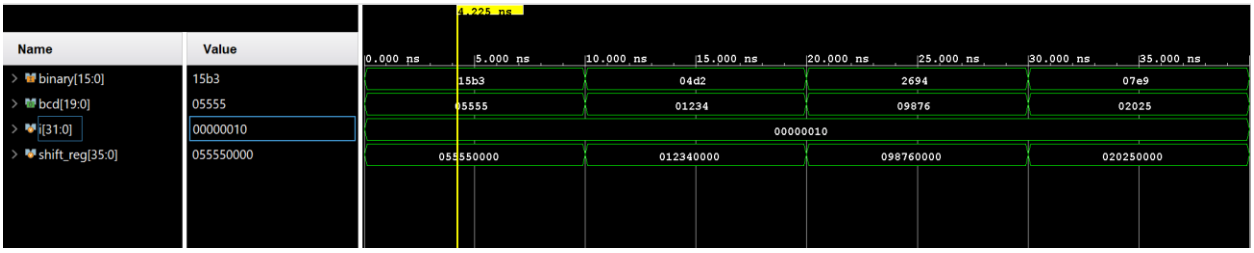
Branch Adder:



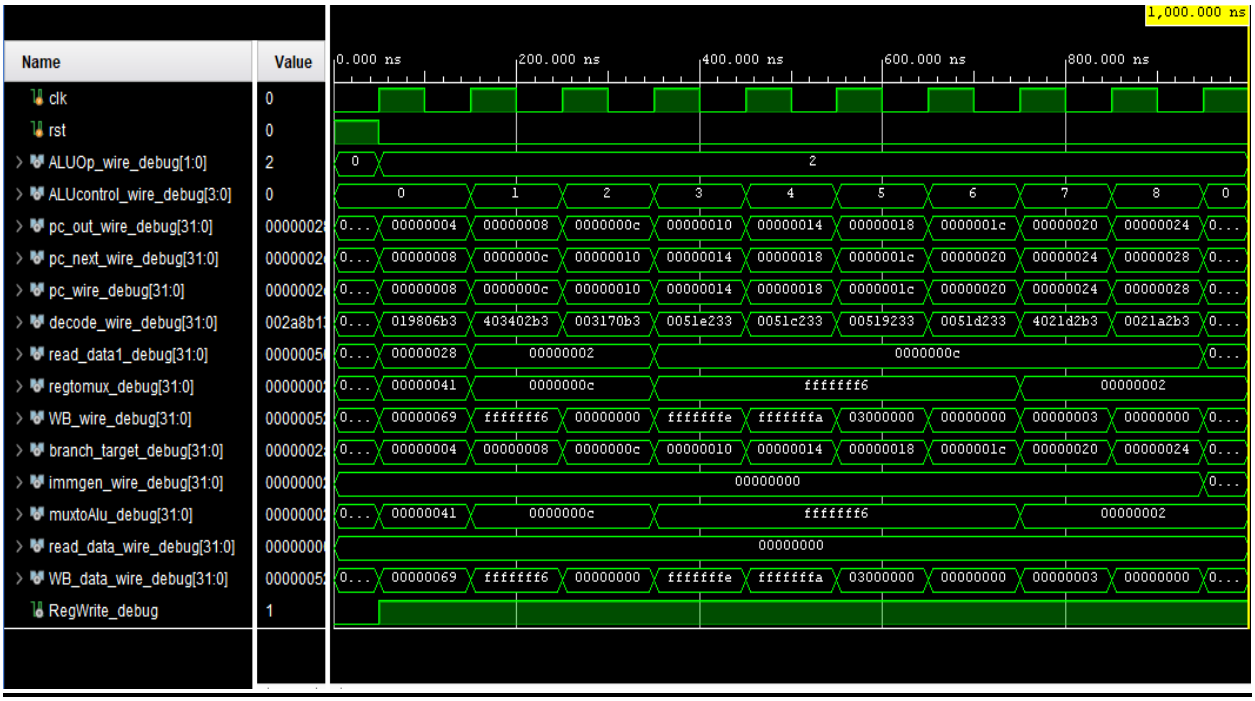
ALU Display:



Binary to Decimal:



Risc V Top Module:



VI. Conclusion

In conclusion, the development of the 16-bit RISC-V processor has provided valuable insights into the fundamentals of computer architecture and digital logic design. By constructing a simplified yet functional version of a RISC-V CPU, this project has demonstrated the feasibility of adapting a modern open-source ISA to a smaller bit-width system for educational and experimental use. The modular design, implemented in Verilog, facilitated a clear understanding of how components such as the ALU, control unit, register file, and memory interact within a processor. Despite the absence of advanced features like pipelining or interrupt handling, the processor successfully executes a variety of basic instructions and showcases core concepts such as instruction decoding, data routing, and control signal management. This project not only strengthened practical HDL skills and design thinking but also laid a strong foundation for future extensions involving performance optimizations, ISA expansion, and integration of more complex architectural features.

VII. Contribution

Siri Simpana T S: Responsible for the integration of all Verilog modules into the top_ALU_display module, implementation of the ALU logic, and the development of the seven-segment display control logic. Also led the debugging and final synthesis of the design in Vivado 2024.2.

Soumya Guruvenkatesh Katti: Designed and tested the Data_Memory, MUX2to1, and Branch_Adder modules. Handled the creation and simulation of test benches for individual modules and verified correctness through waveform analysis.

Sai Preeth Aduwala: Implemented the immediate_generator and binary16_to_bcd modules. Also supported in the development of the display conversion logic and coordinated the physical deployment onto the Nexys A7-100T FPGA board.

Anish Sathaye: Documented the project design, wrote the “Design Overview” and “Working” sections of the report, formatted the IEEE-style document, and managed version control of the code and documentation.

VIII. Reference

[1] Xilinx, "Nexys A7 FPGA Development Board," **[Online].** Available: https://www.digilent.com/reference/programmable-logic/nexys-a7-100t. [Accessed: 10-May-2025].

[2] Xilinx, "Vivado Design Suite 2024.2," **[Online].** Available: https://www.xilinx.com/products/design-tools/vivado.html. [Accessed: 10-May-2025].

[3] R. S. H. I. Z. B. Asan and M. C. J. K. Gupta, "Design and Implementation of ALU on FPGA," *Journal of FPGA Design*, vol. 23, no. 2, pp. 134-140, 2024.

[4] T. C. Johnson and R. L. C. Zhao, "Implementation of Digital Logic Circuits on FPGA Using Verilog," *IEEE Trans. on Computer-Aided Design*, vol. 33, no. 4, pp. 755-761, Apr. 2024.

[5] D. P. S. Thomas and M. A. McDonald, "Practical Design of ALU in Hardware," *Proc. of the IEEE International Conference on Digital Systems Design*, 2024, pp. 243-248.

[6] D. Z. Yang and H. Y. Guo, "Verilog-Based Implementation of ALU for Embedded Systems," *IEEE Embedded Systems Journal*, vol. 18, no. 1, pp. 45-52, Jan. 2024.