Computer Organizaton CS202-2024s - CPU Project Report

Contributors: Jaouhara ZERHOUNI KHAL (12211456), Layheng HOK (12210736), Harrold TOK Kwan Hang (12212025)
Instructor: Jin ZHANG
Lab Session: Friday 7-8

TABLE OF CONTENTS

I. Contribution

Members	Tasks	Ratio
Jaouhara ZERHOUNI KHAL	- CPU Design - Assembly code + Basic Test Scenario 1 + Basic Test Scenario 2 (3'b000 & 3'b1000) - Documentation	1/3
Layheng HOK	- CPU Design - Assembly code + Basic Test Scenario 2 (3'b110 & 3'b111) - Documentation	1/3
Harrold TOK Kwan Hang	- CPU Design - Assembly code - Basic Test Scenario 2 (3'b001, 3'b010, 3'b011 & 3'b101) + Documentation	1/3

II. CPU Architecture Design Description

CPU Features

Bits	31-25	24-20	19-15	14-12	11-7	6-0
R-type	funct7	rs2	rs1	funct3	rd	opcode
I-type	imm[11:0]	-	rs1	funct3	rd	opcode
S-type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode

Bits	31-25	24-20	19-15	14-12	11-7	6-0
B-type	imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]imm[11]
U-type	imm[31:12]	-	-	-	rd	opcode
J-type	imm[20	10:1	11	19:12]	rd	opcode

R-type Instructions

Instruction	Encoding	Usage Method
ADD	7'b0110011 + funct3:000 + funct7:0000000	`add rd, rs1, rs2`
SUB	7'b0110011 + funct3:000 + funct7:0100000	`sub rd, rs1, rs2`
AND	7'b0110011 + funct3:111 + funct7:0000000	`and rd, rs1, rs2`
OR	7'b0110011 + funct3:110 + funct7:0000000	`or rd, rs1, rs2`
SLL	7'b0110011 + funct3:001 + funct7:0000000	`sll rd, rs1, rs2`
SRL	7'b0110011 + funct3:101 + funct7:0000000	`srl rd, rs1, rs2`
SRA	7'b0110011 + funct3:101 + funct7:0100000	`sra rd, rs1, rs2`

I-type Instructions

Instruction	Encoding	Usage Method
ADDI	7'b0010011 + funct3:000	`addi rd, rs1, imm`
ANDI	7'b0010011 + funct3:111	`andi rd, rs1, imm`
XORI	7'b0010011 + funct3:100	`xori rd, rs1, imm`
SRLI	7'b0010011 + funct3:101 + funct7:0000000	`srli rd, rs1, imm`
SLLI	7'b0010011 + funct3:001 + funct7:0000000	`slli rd, rs1, imm`
SRAI	7'b0010011 + funct3:101 + funct7:0100000	`srai rd, rs1, imm`
LW	7'b0000011 + funct3:010	`lw rd, offset(rs1)`
LB	7'b0000011 + funct3:000	`lb rd, offset(rs1)`
LBU	7'b0000011 + funct3:100	`lbu rd, offset(rs1)`
JALR	7'b1100111 + funct3:000	`jalr rd,imm`

S-type Instructions

Instruction	Encoding	Usage Method
SW	7'b0100011 + funct3:010	`sw rs2, offset(rs1)`

B-type Instructions

Instruction	Encoding	Usage Method
BEQ	7'b1100011 + funct3:000	`beq rs1, rs2, label`
BNE	7'b1100011 + funct3:001	`bne rs1, rs2, label`
BLT	7'b1100011 + funct3:100	`blt rs1, rs2, label`
BGE	7'b1100011 + funct3:101	`bge rs1, rs2, label`
BLTU	7'b1100011 + funct3:110	`bltu rs1, rs2, label`
BGEU	7'b1100011 + funct3:111	`bgeu rs1, rs2, label`

J-type Instructions

Instruction	Encoding	Usage Method
JAL	7'b1101111	`jal rd, label`

U-type Instructions

Instruction	Encoding	Usage Method
LUI	7'b0110111	`lui rd, imm`

Reference ISA

The instructions listed above are based on the RISC-V Instruction Set Architecture (ISA).

Updates or Optimizations

- Enhanced the control logic to support additional branch conditions (e.g., `blt`, `bge`, `bltu`, `bgeu`).
- Improved memory and I/O handling by distinguishing between memory access and I/O access using the high bits of the ALU result.
- Added support for I/O read and write instructions with specific encoding to handle unsigned and signed data correctly.
- Incorporated additional ALU operations such as `sll`, `sra`, `slli`and `srli` to support shift operations.
- **Registers**:
    - Bit width: 32 bits
    - Number: 32 Registers

Clock Frequency and CPI

- **CPU Clock Frequency**: The clock frequency of the CPU is 23 MHz. This means the CPU's clock ticks 23 million times per second.

- **CPI (Cycles Per Instruction)**: The CPI is 1, since we use 1 cycle per instruction. This is because our CPU is a single-cycle CPU, meaning each instruction is completed in one clock cycle.

CPU Type

- **Single-cycle or Multi-cycle CPU**: This CPU is a single-cycle CPU. Each instruction completes in a single clock cycle, which simplifies the design but can limit performance at higher clock frequencies.
- **Pipeline Support**: This CPU does not support pipelining.

Addressing Space Design

- **Structure**:
    - Harvard structure: The CPU uses the Harvard architecture. This structure is characterized by having separate memory spaces for instructions and data, which allows simultaneous access to both memories. This can improve performance by allowing the CPU to fetch instructions and read/write data at the same time.
- **Addressing Unit**:
    - Data Read and Write Bits: 32 bits (4 bytes)
- **Size of Instruction Space and Data Space**: 64 KB (2^14 * 4 bytes)
- **Base Address of Stack Space**: 0x7fffeffc

Support for I/O Devices

- Use `lw`/`lb`/`lbu` with negative address to get input.
- Use `sw` with negative address to display output.
- Address to IO mapping:
    - `0xffffffc00` 16 switches
    - `0xffffffc10` left 8 switches
    - `0xffffffc20` button V1
    - `0xffffffc22` button R11
    - `0xffffffc24` button R17
    - `0xffffffc26` button U4
    - `0xffffffc40` 16 LED
    - `0xffffffc60` right 8 LED
    - `0xffffffc69` tube 32-bit
    - `0xffffffc70` tube 16-bit

CPU Interface

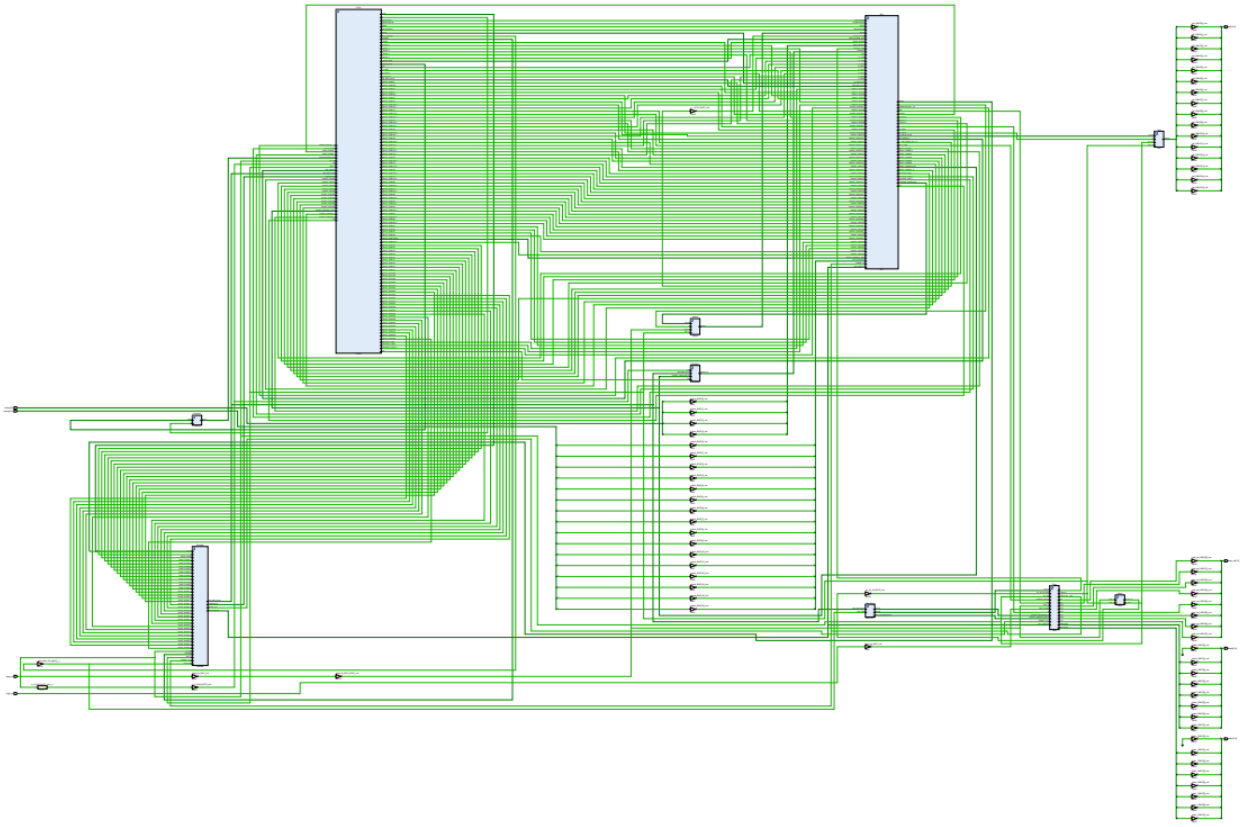
- **clk**: The clock signal (clk) is the primary timing signal for the CPU. It synchronizes the operation of all components by providing a consistent time reference. Each tick of the clock signal corresponds to a clock cycle, which is used to drive the execution of instructions within the CPU. In this project, the clock frequency is 23 MHz, meaning the CPU operates at 23 million cycles per second.
- **reset**: The reset signal (reset) is used to initialize the CPU to a known state. When the reset signal is asserted, it sets the CPU to its initial state, clearing all registers, resetting the program counter, and

preparing the CPU to start executing instructions from the beginning. This signal is crucial for ensuring that the CPU starts correctly and can recover from errors or unwanted states.

- **Description about other normal I/O interface**:
  - **Data Bus**: The data bus is used to transfer data between the CPU and memory or I/O devices. It is bidirectional, allowing the CPU to read from and write to memory or peripherals.
  - **Address Bus**: The address bus carries the addresses of memory locations or I/O devices. It is used by the CPU to specify the location from which it wants to read or to which it wants to write data.
  - **Control Signals**: These include signals such as read and write which indicate the type of operation being performed (e.g., memory read, memory write, I/O read, I/O write).

CPU Internal Structure

Interface Connection Diagram of Internal Submodules of CPU



Design Description of CPU Internal Submodules

- **CPU**
  - input `fpga_rst`: FPGA reset signal
  - input `fpga_clk`: FPGA clock signal
  - input `switch`: 16-bit input from switches
  - input `button`: 4-bit input from buttons
  - output `led`: 16-bit output to LEDs
  - output `tube_sel`: 8-bit output to select which tube to display
  - output `tube0`: 8-bit output for tube0 display

    - output `tube1`: 8-bit output for tube1 display

- **Clock**
    - input `clk_in1`: FPGA clock input
    - input `rst`: FPGA reset input
    - output `clk_out1`: Single cycle CPU clock output
    - output `clk_out2`: Data output clock

- **IFetch**
    - input `clk`: Clock signal
    - input `rst`: Reset signal
    - input `imm32`: 32-bit immediate value
    - input `jump`: Jump signal
    - input `jalr`: Jump and link register signal
    - input `beq`: Branch if equal signal
    - input `bne`: Branch if not equal signal
    - input `blt`: Branch if less than signal
    - input `bltu`: Branch if less than unsigned signal
    - input `bge`: Branch if greater than or equal signal
    - input `bgeu`: Branch if greater than or equal unsigned signal
    - input `zero`: Zero flag signal
    - input `ALUResult`: 32-bit ALU result
    - output `ra`: 32-bit register address (output, reg)
    - output `inst`: 32-bit instruction (output, wire)

- **Decoder**
    - input `clk`: Clock signal
    - input `rst`: Reset signal
    - input `ALU_result`: 32-bit ALU result
    - input `MemOrIOtoReg`: Memory or IO to register signal
    - input `regWrite`: Register write signal
    - input `inst`: 32-bit instruction
    - input `ra`: 32-bit register address
    - input `jump`: Jump signal
    - input `IOData`: 32-bit IO data
    - output `rs1Data`: 32-bit data for source register 1
    - output `rs2Data`: 32-bit data for source register 2
    - output `imm32`: 32-bit immediate value (output, reg)

- **Controller**
    - input `inst`: 32-bit instruction
    - input `ALU_result_high`: 22-bit high part of ALU result
    - output `Jump`: Jump signal
    - output `jalr`: Jump and link register signal
    - output `beq`: Branch if equal signal

    - output `blt`: Branch if less than signal
    - output `bge`: Branch if greater than or equal signal
    - output `bltu`: Branch if less than unsigned signal
    - output `bgeu`: Branch if greater than or equal unsigned signal
    - output `MemRead`: Memory read signal
    - output `ALUOp`: 2-bit ALU operation (output, reg)
    - output `MemWrite`: Memory write signal
    - output `ALUSrc`: ALU source signal (output, reg)
    - output `RegWrite`: Register write signal
    - output `MemOrIOtoReg`: Memory or IO to register signal
    - output `IOReadUnsigned`: IO read unsigned signal
    - output `IOReadSigned`: IO read signed signal
    - output `IOWrite`: IO write signal
    - output `bne`: Branch if not equal signal

- **DataMem**

    - input `clk`: Clock signal
    - input `mem_read`: Memory read signal
    - input `mem_write`: Memory write signal
    - input `addr`: 32-bit address
    - input `din`: 32-bit data input
    - output `dout`: 32-bit data output

- **ALU**

    - input `ReadData1`: 32-bit data from source register 1
    - input `ReadData2`: 32-bit data from source register 2
    - input `imm32`: 32-bit immediate value
    - input `ALUSrc`: ALU source signal
    - input `ALUOp`: 2-bit ALU operation
    - input `funct3`: 3-bit function field
    - input `funct7`: 7-bit function field
    - output `ALUResult`: 32-bit ALU result (output, reg)
    - output `zero`: Zero flag signal (output, wire)

- **MemOrIO**

    - input `mRead`: Memory read signal
    - input `mWrite`: Memory write signal
    - input `IOReadUnsigned`: IO read unsigned signal
    - input `IOReadSigned`: IO read signed signal
    - input `IOWrite`: IO write signal
    - input `addr_in`: 32-bit address input
    - output `addr_out`: 32-bit address output
    - input `m_rdata`: 32-bit data read from memory
    - input `io_rdata`: 16-bit data read from IO
    - output `r_wdata`: 32-bit data to register file (output, reg)

    - input `r_rdata`: 32-bit data read from register file
    - output `write_data`: 32-bit data to memory or IO (output, reg)
    - output `write_data_32`: 32-bit data to memory or IO (original 32-bit value) (output, reg)
    - output `LEDCtrl`: LED chip select
    - output `SwitchCtrl`: Switch chip select
    - output `TubeCtrl`: Tube chip select

- **Switch**

    - input `clk`: Clock signal
    - input `rst`: Reset signal
    - input `IOReadUnsigned`: IO read unsigned signal
    - input `IOReadSigned`: IO read signed signal
    - input `SwitchCtrl`: Switch control signal
    - input `button`: 4-bit button input
    - input `switchaddr`: 8-bit switch address
    - input `switch`: 16-bit switch input
    - output `switchrdata`: 16-bit switch read data (output, reg)

- **Led**

    - input `clk`: Clock signal
    - input `rst`: Reset signal
    - input `IOWrite`: IO write signal
    - input `LEDCtrl`: LED control signal
    - input `ledaddr`: 8-bit LED address
    - input `ledwdata`: 16-bit LED write data
    - output `led`: 16-bit LED output (output, reg)

- **Tube**

    - input `clk`: Clock signal
    - input `fpga_clk`: FPGA clock signal
    - input `rst`: Reset signal
    - input `clk_div_2s`: 2-second clock divider
    - input `button`: Test case button input
    - input `IOWrite`: IO write signal
    - input `TubeCtrl`: Tube control signal
    - input `tubeaddr`: 8-bit tube address
    - input `tubewdata`: 16-bit tube write data
    - input `tubewdata32`: 32-bit tube write data
    - input `testcase`: 16-bit test case number
    - output `sel`: 8-bit bit selective signal (output, reg)
    - output `tube0`: 8-bit segment-selected signal (output, reg)
    - output `tube1`: 8-bit segment-selected signal (output, reg)

III. Usage Instructions about the System

Hardware Interface Description

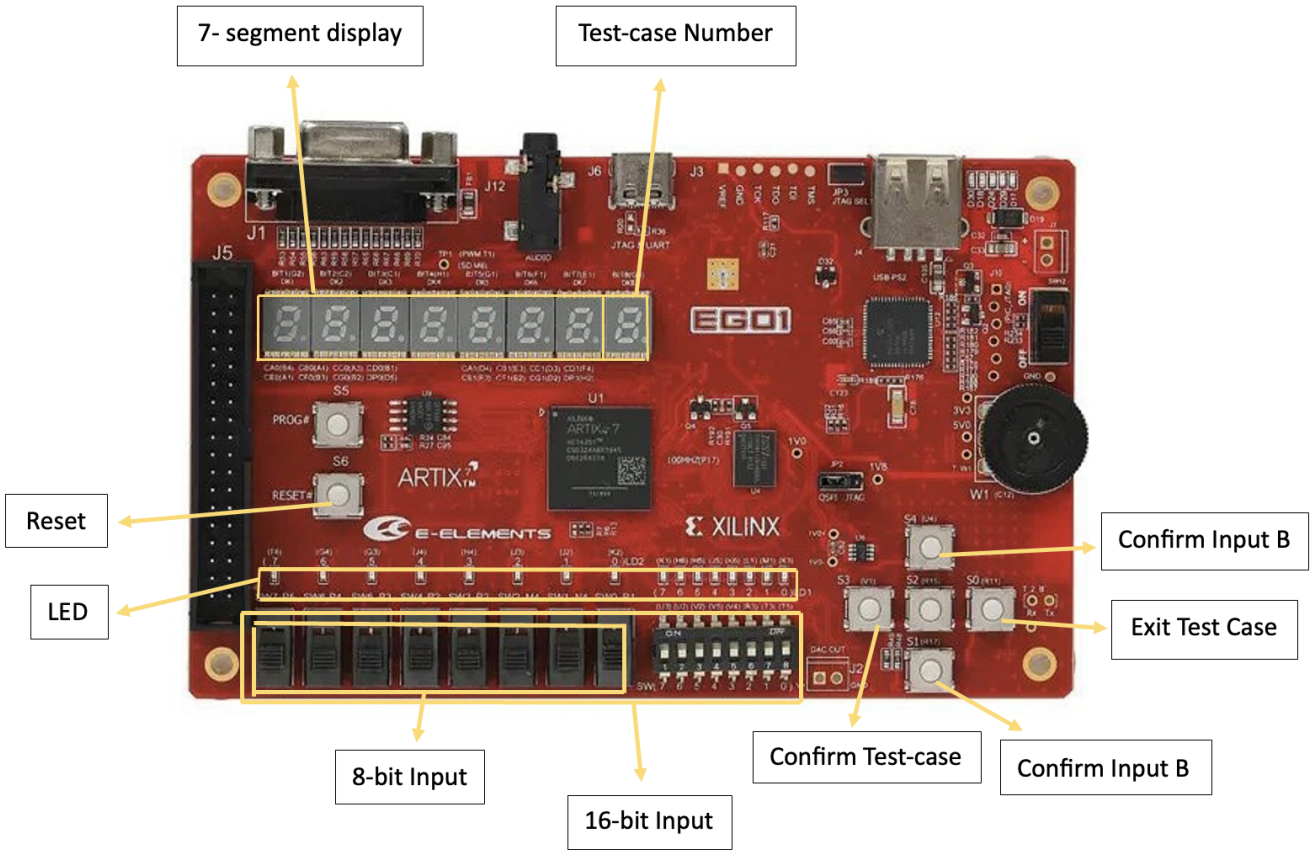
Buttons and Switches:

- **Reset Button**
  - **Description**: Connected to pin `P15` on the FPGA, used to reset the CPU.
- **Confirm Test-case Button**
  - **Address**: `0xfffffc20`
  - **Description**: Connected to pin `V1` on the FPGA, used to confirm the test-case number.
- **Confirm Input A Button**
  - **Address**: `0xfffffc24`
  - **Description**: Connected to pin `R17` on the FPGA, used to confirm input A.
- **Confirm Input B Button**
  - **Address**: `0xfffffc26`
  - **Description**: Connected to pin `U4` on the FPGA, used to confirm input B.
- **Exit test-case Button**
  - **Address**: `0xfffffc22`
  - **Description**: Connected to pin `R11` on the FPGA, used to confirm the end of the test-case (turn off both tube and LED).
- **16-bit Switches**
  - **Address**: `0xfffffc00`
  - **Description**: Connected to pin from `T5` to pin `P5` on the FPGA, used to for test cases that require 16-bit input and 12-bit input.
- **8-bit Switches**
  - **Address**: `0xfffffc10`
  - **Description**: Connected to pin from `R1` to pin `P5` on the FPGA, used to for test cases that require 8-bit input.

LED and Tube:

- **LED 16-bit**
  - **Address**: `0xfffffc40`
  - **Description**: Connected to pin from `K3` to pin `F6` on the FPGA, initially turned on indicating the start of the program. Turned off when button `Exit Test_case` is pressed.
- **LED 8-bit**
  - **Address**: `0xfffffc60`
  - **Description**: Connected to pin from `K3` to pin `K1` on the FPGA, used to display the result.
- **Tube 16-bit (7-segment)**
  - **Address**: `0xfffffc70`
  - **Description**: Used to display the 16-bit output of the corresponding test cases. The right box indicating the test_case number (0-7).
- **Tube 32-bit (7-segment)**
  - **Address**: `0xfffffc69`
  - **Description**: Used to display the 32-bit output of the corresponding test cases. The right box indicating the test_case number (0-7).

FPGA Board



Sample:

```
module Switch(input clk,                        // 23MHz CPU clk
              input rst,                        // Reset
              input IOReadUnsigned,             // IO read unsigned
              input IOReadSigned,               // IO read signed
              input SwitchCtrl,                 // Switch ctrl
              input [3:0] button,               // Button
              input [7:0] switchaddr,           // Switch address
              input [15:0] switch,              // Switch
              output reg [15:0] switchrdata); // Switch read data

    always@(negedge clk or negedge rst) begin
        if (~rst) begin
            switchrdata <= 16'h0000;
        end
        else begin
             if (SwitchCtrl && (IOReadUnsigned || IOReadSigned)) begin
                if (switchaddr == 8'h00) begin //0xffffc00
                    switchrdata[15:0] <= switch[15:0]; //read from the 16 switches
                end else if (switchaddr == 8'h10) begin //0xffffc10
                     switchrdata[15:0] <= {8'h00, switch[15:8]}; //read from the
left 8 switches
                end else if (switchaddr == 8'h20) begin //0xffffc20
                     switchrdata[15:0] <= {15'b000_0000_0000_0000, button[3]};
//read from V1 (test_case button)
                end else if (switchaddr == 8'h22) begin //0xffffc22
```

```
                    switchrdata[15:0] <= {15'b000_0000_0000_0000, button[2]};
//read from R11 (end test button)
                end else if (switchaddr == 8'h24) begin //0xffffc24
                    switchrdata[15:0] <= {15'b000_0000_0000_0000, button[0]};
//read from R17 (input A button)
                end else if (switchaddr == 8'h26) begin //0xffffc26
                    switchrdata[15:0] <= {15'b000_0000_0000_0000, button[1]};
//read from U4 (input B button)
                end
            end
        end
    end

endmodule
```

IV. Self Testing Instructions

Testing Method	Testing Type	Testcase Description	Test Result (Pass/Fail)
Simulation	Unit Testing	Test each submodule individually, including but not limited to `DataMem` and `MemOrIO`.	Pass
Simulation	Integration Testing	Simulate the whole flow process of CPU.	Pass
On Board	Integration Testing	Test the functionality of the whole CPU with test cases provided in Scenario 1 with FPGA board.	Pass
On Board	Integration Testing	Test the functionality of the whole CPU with test cases provided in Scenario 2 with FPGA board.	Pass

Final Test Conclusion

All testcases were executed successfully, and the system passed all the self-testing phases, both in simulation and on-board testing. The CPU design functions as expected, meeting all specified requirements and ensuring correct behavior for each testcase.

V. Problem and Summary

Problems Encountered During Development

In this project, we built a single-cycle CPU based on the RISC-V ISA. Our goal was to create a working CPU that could run a set of instructions correctly. However, we faced several problems during development:

- **Understanding CPU Design**: Learning the details of CPU design was quite challenging and needed a deep understanding of both the hardware and the RISC-V instruction set.
- **Running RISC-V Codes on the FPGA**: It wasn't evident how to run the RISC-V codes on the FPGA. This included figuring out the correct procedure to load and execute the RISC-V programs on the hardware.

- **Verilog Integration**: Connecting the RISC-V code with the FPGA using Verilog was complex. We needed to make sure that all parts worked together correctly.
- **Debugging**: Debugging complex issues in both the Verilog code and the RISC-V assembly programs required meticulous attention to detail and often involved extended problem-solving efforts.
- **Meeting Deadlines**: Strict deadlines added pressure, necessitating efficient time management and prioritization of tasks to ensure timely completion of project milestones.

Reflections and Summary

Despite these challenges, the project provided valuable learning experiences and opportunities for growth:

- **Enhanced Technical Skills**: We gained a deeper understanding of CPU architecture and the RISC-V ISA, improving our proficiency in both Verilog and hardware design.
- **Problem-Solving Abilities**: Overcoming the various obstacles enhanced our problem-solving abilities, teaching us to approach issues methodically and persistently.
- **Collaboration and Teamwork**: The project underscored the importance of effective collaboration and communication within a team, reinforcing the need for clear roles and consistent updates.
- **Project Management**: Managing the project's scope, timeline, and resources taught us valuable lessons in project management, particularly in balancing technical work with administrative responsibilities.

In summary, while the development process was filled with challenges, it was ultimately rewarding. We successfully implemented a single-cycle CPU capable of executing a set of RISC-V instructions. The experience has equipped us with critical skills and knowledge that will be invaluable in future projects.