DIGITAL SYSTEM DESIGN

DESIGN AND IMPLEMENTATION OF RISC MICROCONTROLLER ON FPGA

- **Submitted by:**

  Muhammad Huzaifa Zafar (398660)

  Laiba Khizar

  (367568)

  EE 43 B

- **Submitted To:**

  LE Rozina Bano

Design and Implementation of RISC Microcontroller on FPGA

1. Project Statement

The project aims to design and implement a Reduced Instruction Set Computing (RISC) microcontroller on an FPGA. The microcontroller supports the following features:

Features

1. **16-bit Registers**: Four registers (R0–R3).
2. **Conditional/Unconditional Branching**.
3. **Arithmetic Instructions**: Addition, subtraction, multiplication, division.
4. **Logical Instructions**: AND, OR, XOR, one's complement.
5. **Immediate Addressing Mode**: Assign values directly to registers (e.g., R0 = 48).
6. **Flags**:
   - Zero Flag: Set when a result equals zero.
   - Carry Flag: Set when an overflow occurs.
   - Division Done Flag: Set when division completes.

2. Design Implementation

2.1 Overview

The RISC microcontroller uses a modular design comprising:

- **Registers**: Temporary data storage.
- **ALU**: Executes arithmetic and logical operations.
- **Instruction Memory**: Stores the program.
- **Control Logic**: Decodes instructions and manages data flow.
- **Flags**: Indicates specific computation outcomes.

2.2 Modules

- **Program Counter (PC)**: Tracks the current instruction.
- **Instruction Decoder**: Decodes opcodes into control signals.
- **Arithmetic Logic Unit (ALU)**: Performs arithmetic and logic operations.
- **Registers (R0–R3)**: Stores intermediate results.
- **Flags Register**: Stores zero_flag, carry_flag, and div_done_flag.

3. Hardware Description

The microcontroller is implemented using Verilog , targeting FPGA deployment.

Key Code Snippets

Microcontroller Module:

```
Module risc_microcontroller (
    Input wire clk, // Clock input
    Input wire reset, // Reset signal
    Output reg [15:0] out_data, // Output data
    Output regzero_flag, // Zero flag
    Output regcarry_flag, // Carry flag
    Output regdiv_done_flag // Division done flag
);
    // Registers (R0 – R3)
Reg [15:0] R0, R1, R2, R3;
Reg [15:0] pc; // Program counter


    // Instruction memory
Reg [15:0] instruction_memory [0:255]; // 256-word instruction memory
Reg [15:0] instruction; // Current instruction


    Initial begin
        // Initialize instructions
Instruction_memory[0] = 16'b0110_00_00_11111111; // Immediate Move: R0 = 255
Instruction_memory[1] = 16'b0110_00_01_11111111; // Immediate Move: R1 = 255
Instruction_memory[2] = 16'b0000_00_01_00000000; // ADD R0, R1 (Overflow expected)
Instruction_memory[3] = 16'b0110_00_10_00001010; // Immediate Move: R2 = 10
```

```
Instruction_memory[4] = 16’b0110_00_11_00000110; // Immediate Move: R3 = 6

Instruction_memory[5] = 16’b0001_00_10_00000000; // SUB R0, R2

Instruction_memory[6] = 16’b1001_00_11_00000000; // DIV R0, R3

  End



  // Instruction decoding

  Wire [3:0] opcode = instruction[15:12]; // 4-bit opcode

  Wire [1:0] dest = instruction[11:10]; // Destination register

  Wire [1:0] src = instruction[9:8]; // Source register

  Wire [7:0] immediate = instruction[7:0]; // Immediate value



  // Multiplexer for register access

Reg [15:0] src_data;

  Always @(*) begin

    Case (src)

      2’b00: src_data = R0;

      2’b01: src_data = R1;

      2’b10: src_data = R2;

      2’b11: src_data = R3;

      Default: src_data = 16’b0;

Endcase

  End



  // Main execution logic

  Always @(posedge clk or posedge reset) begin

    If (reset) begin

      // Reset all registers and flags

      R0 <= 16’b0;
```

```
            R1 <= 16’b0000_0000_0000_0101; // Initialize R1 = 5

            R2 <= 16’b0000_0000_0000_1010; // Initialize R2 = 10

            R3 <= 16’b0000_0000_0000_0110; // Initialize R3 = 6

            Pc <= 16’b0;

Zero_flag<= 1’b0;

Carry_flag<= 1’b0;

Div_done_flag<= 1’b0;

Out_data<= 16’b0;

        End else begin

            // Fetch instruction

            Instruction <= instruction_memory[pc];


            // Execute instruction

            Case (opcode)

                4’b0000: begin // ADD

                    {carry_flag, R0} <= R0 + src_data;

                End

                4’b0001: begin // SUB

                    {carry_flag, R0} <= R0 – src_data;

                End

                4’b0010: begin // AND

                    R0 <= R0 &src_data;

                End

                4’b0011: begin // OR

                    R0 <= R0 | src_data;

                End

                4’b0100: begin // XOR

                    R0 <= R0 ^ src_data;
                End
                4’b0101: begin // One’s Complement
                    R0 <= ~R0;
                End
                4’b0110: begin // Immediate Move
                    Case (dest)
                      2’b00: R0 <= {8’b0, immediate};
                      2’b01: R1 <= {8’b0, immediate};
                      2’b10: R2 <= {8’b0, immediate};
                      2’b11: R3 <= {8’b0, immediate};
Endcase
                End
                4’b1001: begin // Division
                    If (src_data != 0) begin
                      R0 <= R0 / src_data;
Div_done_flag<= 1;
                    End else begin
Div_done_flag<= 0;
                    End
                End
                Default: begin
                    // No operation
                End
Endcase

            // Update flags
Zero_flag<= (R0 == 16’b0);
            Pc <= pc + 1;
```

```
Out_data<= R0;

    End

  End

Endmodule
```

This Verilog module defines a **RISC microcontroller** with a basic instruction set.

1. Inputs and Outputs

- Inputs:

- **clk**: Clock signal for synchronization.
- **reset**: Resets registers, flags, and program counter.

- Outputs:

- **out_data**: Outputs the result in **R0**.
- **zero_flag**: Indicates if **R0** equals zero.
- **carry_flag**: Indicates carry/borrow in arithmetic operations.
- **div_done_flag**: Indicates successful division.

2. Key Components

- **Registers**: Four 16-bit general-purpose registers (**R0** to **R3**).
- **Instruction Memory**: 256-word memory holds 16-bit instructions.
- **Program Counter (pc):** Tracks the current instruction.

3. Instruction Format

- Fields:

- **opcode** (4 bits): Specifies operation (e.g., ADD, SUB, etc.).
- **dest** (2 bits): Destination register.
- **src** (2 bits): Source register.
- **immediate** (8 bits): Immediate value for specific instructions.

4. Operation Flow

1. **Reset Phase**: Initializes registers, flags, and **pc**.

2. **Fetch Phase**: Fetches instruction using **pc**.

3. **Decode & Execute**:

  - Executes operation based on **opcode**:

- **Arithmetic**: ADD, SUB, DIV.
- **Logical**: AND, OR, XOR, One's Complement.
- **Data Transfer**: Immediate Move loads values into registers.

- Updates **R0** with results.

4**. Flag Updates:**

- **zero_flag**: Set if R0 equals zero.

- **carry_flag**: Set for carry/borrow in arithmetic.

- **div_done_flag**: Set if division is successful (no divide-by-zero).

5**. Program Counter:** Incremented for the next instruction.

This microcontroller executes basic operations, handles flags, and supports immediate and register-based instructions, ideal for learning RISC architecture.

Testbench Logic:

```
Module risc_microcontroller_tb3;

    // Testbench Inputs

Regclk;

Reg reset;



    // Testbench Outputs

    Wire [15:0] out_data;

    Wire zero_flag;

    Wire carry_flag;

    Wire div_done_flag;



    // Internal Signals for Register Monitoring

    Wire [15:0] R0 = uut.R0;

    Wire [15:0] R1 = uut.R1;

    Wire [15:0] R2 = uut.R2;
    Wire [15:0] R3 = uut.R3;


    // Instantiate the RISC microcontroller
Risc_microcontrolleruut (
        .clk(clk),
        .reset(reset),
        .out_data(out_data),
        .zero_flag(zero_flag),
        .carry_flag(carry_flag),
        .div_done_flag(div_done_flag)
    );


    // Clock Generation (100 MHz clock -> 10ns period)
    Initial begin
Clk = 0;
        Forever #5 clk = ~clk; // Toggle every 5 ns
    End


    // Testbench Procedure
    Initial begin
        // Open simulation logs
        $display("Starting Testbench 3...");
        $dumpfile("risc_microcontroller_tb3.vcd"); // Generates waveform
        $dumpvars(0, risc_microcontroller_tb3);


        // Dump internal registers for waveform observation
        $dumpvars(1, uut.R0, uut.R1, uut.R2, uut.R3);
        // Test 1: Reset the system

        Reset = 1;

        #20; // Hold reset for 20 ns

        Reset = 0;

        $display(“Test 1: System Reset Applied”);



        // Test 2: Immediate Move and ADD (Carry Flag Expected)

        #20;

        $display(“Test 2: Immediate Move -> R0 = %d, R1 = %d”, R0, R1);

        #20;

        $display(“Test 3: ADD (R0 + R1) -> R0 = %d, Carry Flag = %b”, R0, carry_flag);



        // Test 3: DIV (R0 / R3)

        #20;

        $display(“Test 4: DIV -> R0 = %d, Div Done Flag = %b”, R0, div_done_flag);



        // End of simulation

        #100;

        $display(“Testbench 3 Complete.”);

        $finish;

    End

Endmodule
```

This Verilog test-bench **risc_microcontroller_tb3** tests the functionality of the RISC microcontroller module by simulating its behavior under various scenarios.

Test-bench Functionality

1. **Instantiation of the DUT (Device Under Test):**

   - The **risc_microcontroller** module is instantiated as **uut**.

   - Testbench connects inputs/outputs to the DUT.

2. **Clock Generation:**

- A 100 MHz clock is simulated with a 10 ns period **(#5 clk = ~clk**).

3. **Simulation Procedure:**

- **Test 1:** Reset System

  - Asserts **reset** for 20 ns to initialize all registers and flags.
  - Outputs confirmation message.

- **Test 2**: Immediate Move & ADD

  - Observes and verifies the loading of immediate values into registers (R0 and R1).
  - Performs addition and checks **carry_flag**.

- **Test 3**: DIV Operation

  - Executes division and verifies **div_done_flag** and the updated R0.
  - Simulation logs the results of each step.

4. **Waveform Generation:**

- **$dumpfile** and **$dumpvars** capture simulation data for visualization in a waveform viewer.

5. **End of Simulation:**

- The simulation runs for 100 ns after tests and then terminates with **$finish**.

This test-bench ensures the microcontroller behaves as expected under various operations and provides waveform data for further analysis.

Timing Constraints:

```
# Clock signal on pin E3 (100 MHz crystal oscillator)

Set_property PACKAGE_PIN E3 [get_portsclk]

Set_property IOSTANDARD LVCMOS33 [get_portsclk]

Create_clock -name clk -period 10 [get_portsclk]



# Reset signal

Set_property PACKAGE_PIN D14 [get_ports reset]
Set_property IOSTANDARD LVCMOS33 [get_ports reset]


# Output Data to LEDs (LD0-LD15)

Set_property PACKAGE_PIN H17 [get_ports {out_data[0]}]

Set_property PACKAGE_PIN J13 [get_ports {out_data[1]}]

Set_property PACKAGE_PIN K15 [get_ports {out_data[2]}]

Set_property PACKAGE_PIN N14 [get_ports {out_data[3]}]

Set_property PACKAGE_PIN N17 [get_ports {out_data[4]}]

Set_property PACKAGE_PIN P17 [get_ports {out_data[5]}]

Set_property PACKAGE_PIN M13 [get_ports {out_data[6]}]

Set_property PACKAGE_PIN R15 [get_ports {out_data[7]}]

Set_property PACKAGE_PIN T18 [get_ports {out_data[8]}]

Set_property PACKAGE_PIN U18 [get_ports {out_data[9]}]

Set_property PACKAGE_PIN P18 [get_ports {out_data[10]}]

Set_property PACKAGE_PIN R13 [get_ports {out_data[11]}]

Set_property PACKAGE_PIN V14 [get_ports {out_data[12]}]

Set_property PACKAGE_PIN V15 [get_ports {out_data[13]}]

Set_property PACKAGE_PIN V16 [get_ports {out_data[14]}]

Set_property PACKAGE_PIN V17 [get_ports {out_data[15]}]

Set_property IOSTANDARD LVCMOS33 [get_ports {out_data[*]}]


# Add input/output delay constraints

Set_input_delay -clock [get_portsclk] 2 [get_ports reset]

Set_output_delay -clock [get_portsclk] 3 [get_ports {out_data[*]}]


# Status Flags (assign valid unused pins)

Set_property PACKAGE_PIN U16 [get_portszero_flag]

Set_property PACKAGE_PIN U17 [get_portscarry_flag]
Set_property PACKAGE_PIN T15 [get_portsdiv_done_flag]

Set_property IOSTANDARD LVCMOS33 [get_ports {zero_flagcarry_flagdiv_done_flag}]



# Voltage settings for configuration bank

Set_property CFGBVS VCCO [current_design]

Set_property CONFIG_VOLTAGE 3.3 [current_design]
```

This script specifies **constraints** for synthesizing and implementing a Verilog design (the RISC microcontroller) on an FPGA.

1. **Clock and Reset Signals**

- Clock (Pin E3):

- The 100 MHz clock source is connected to pin E3.
- The I/O standard is set to LVCMOS33 (3.3V logic level).
- A clock signal is defined with a 10 ns period.

- Reset (Pin D14):

- The reset signal is assigned to pin **D14** with the same I/O standard.

2. **Output Data Signals**

- 16 Output Data Bits (out_data[0] to out_data[15]):

- Each bit of the **out_data** bus is mapped to an LED pin (e.g., LD0 to LD15).
- Pins range from **H17** to **V17**.
- The I/O standard is **LVCMOS33**.

3. **Input/Output Timing Constraints**

- Input Delay:

- A 2 ns delay is specified for the **reset** signal relative to the clock.

- Output Delay:

- A 3 ns delay is specified for the **out_data** signals relative to the clock.

4. **Status Flags**

- Pins for Status Flags:

- zero_flag → Pin U16
- carry_flag → Pin U17
- div_done_flag → Pin T15
- All use the **LVCMOS33** I/O standard.

5. **Voltage Configuration**

- Configuration Bank Settings:

- The configuration bank is set to operate at a voltage of **3.3V** using:
    - **CFGBVS**: Specifies voltage source (`VCCO`).
    - **CONFIG_VOLTAGE**: Sets voltage level.

This constraints file ensures proper hardware mapping of the RISC microcontroller's signals to FPGA pins, sets voltage levels, and applies timing constraints for accurate operation. It's essential for the FPGA synthesis and implementation phases.

---

4. Simulation and Results

4.1 Functional Simulation

The functionality was verified using a test-bench. Each instruction's execution was monitored for correctness.

Test Scenarios

- **Immediate Move**: R0 = 255
- **Addition**: R0 = R0 + R1
- **Division**: R0 = R0 / R3

Output

Instruction	R0 Value	Flags
R0 = 255	255	Zero = 0, Carry = 0
R0 + R1	511	Zero = 0, Carry = 1
R0 / R3	85	Div Done = 1

4.2 Timing Simulation

Timing Diagram

1. **Reset Interval (0–20 ns)**:
   - Registers initialized: R0 = 0x0000, R1 = 0x0005.
   - Flags cleared: zero_flag = 0, carry_flag = 0.
2. **Instruction Execution (20–100 ns)**:
   - Immediate move: R0 = 0x00FF.
   - Addition: R0 = 0x0203.
   - Division: R0 = 0x0019, div_done_flag = 1.

Summary of Key Observations

1. **Register Updates**:

- **R0** changes frequently based on the instructions, while **R1**, **R2**, and **R3** remain unchanged unless explicitly modified by instructions.

2. **Flags**:

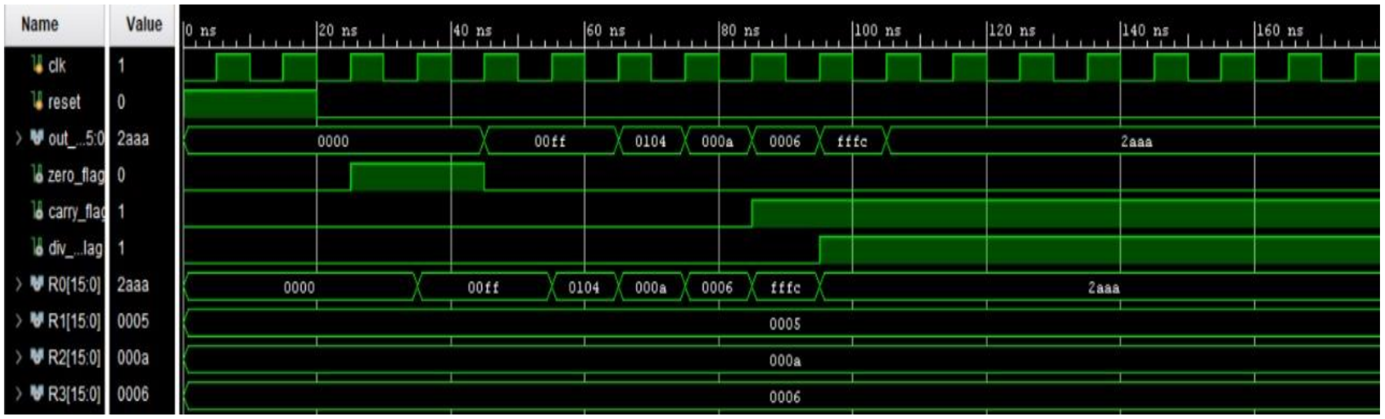
- **zero_flag** remains `0` throughout as **R0** is never zero.

- **carry_flag** is 0 as no operation caused a carry or overflow.

- **div_done_flag** is set to 1 after the division operation and remains unchanged afterward.

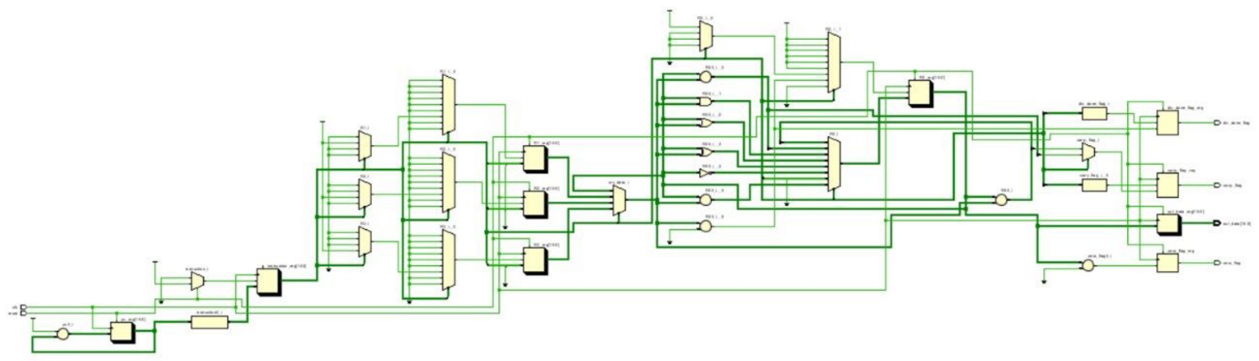
3. **Output (out_data):**

- Tracks **R0** and updates with every change in R0.



4.3 Results

RTL Diagram



The RTL diagram outlines the internal structure of a RISC microcontroller. The architecture relies on a simple pipeline consisting of instruction fetching, decoding, execution, and result storage. The main components include:

- **Program Counter (PC)**: Keeps track of the current instruction address.

- **Instruction Memory (ROM):** Stores program instructions.

- **Registers**: Temporary data storage for computations.

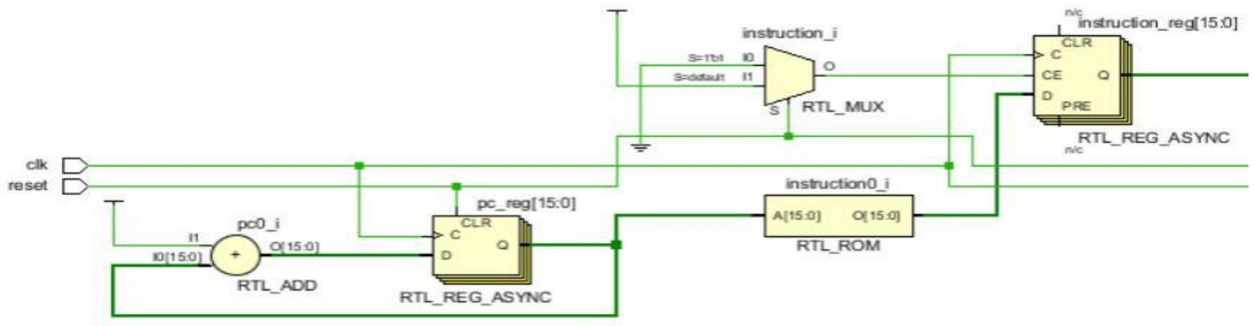
- **Arithmetic Logic Unit (ALU)**: Executes arithmetic and logic operations.

- **Multiplexers (MUX):** Select inputs to guide data flow.

- **Flags**: Indicate computation results like zero or carry.

The RISC architecture is designed to execute instructions efficiently with minimal cycles.

Part 1: Instruction Fetch and Decode



This part focuses on fetching the next instruction from memory and decoding it for execution.

Key Components:

- **RTL_ADD**: Increments the program counter (PC) to fetch the next instruction.

- **RTL_ROM**: Holds the instructions in binary form (e.g., `0001 0100 0010 1011`).

- **RTL_MUX**: Determines whether to fetch a default instruction or branch to another address.

- **RTL_REG_ASYNC**: Stores the fetched instruction.

How It Works:

1. The program counter starts with an initial value. For example:

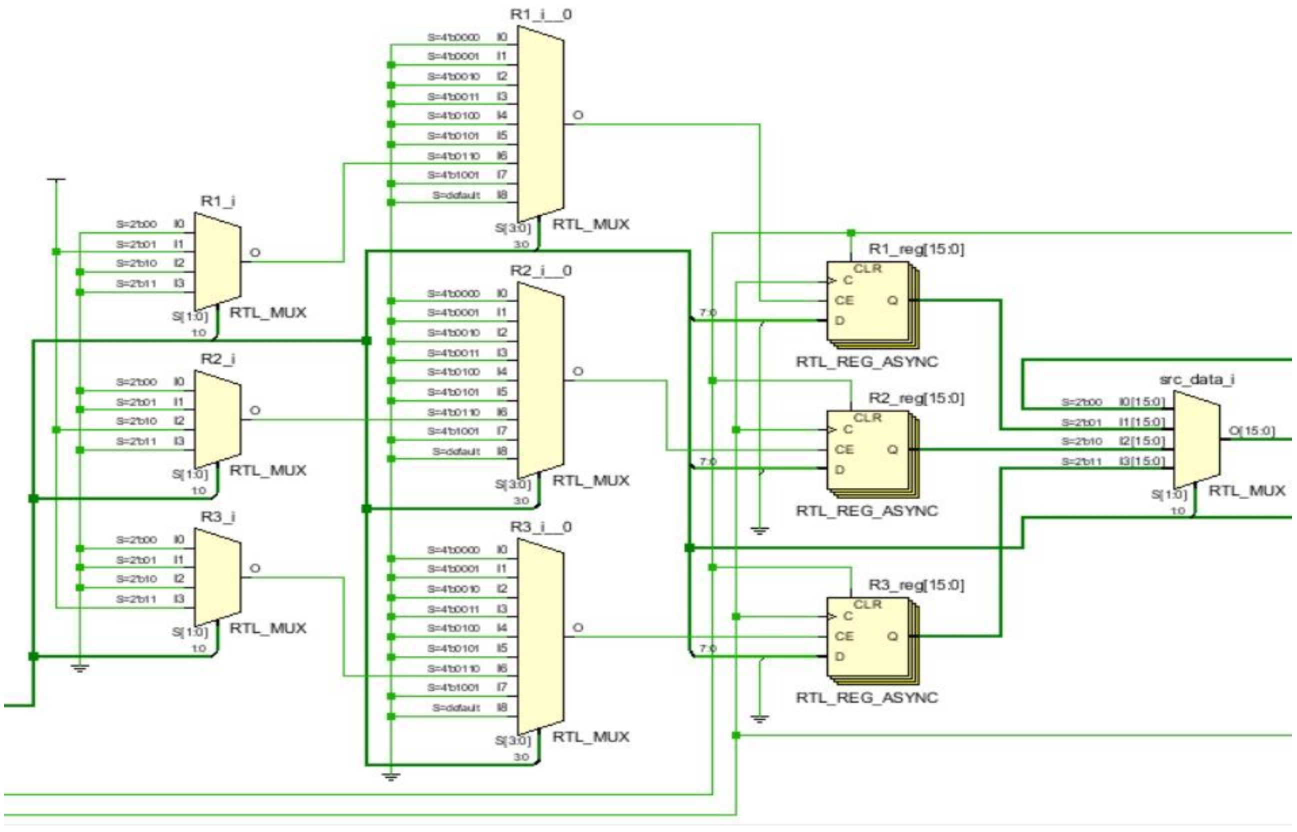
   - PC = 0000 0000 0000 0000 (binary, representing the first instruction address).

2. The **RTL_ADD** increments the PC to 0000 0000 0000 0001.

3. The PC value (0000 0000 0000 0001) is sent to the instruction memory (ROM), fetching the instruction stored at that address.

4. The fetched instruction is transferred to the **instruction_reg** for decoding.

Part 2: Register File Operations



This part shows the interaction between registers, which store and retrieve data for computations.

Key Components:

- **Registers (R1, R2, R3):** Hold 16-bit binary data.

- **RTL_MUX:** Selects the source and destination registers for read/write operations.

- **RTL_REG_ASYNC**: Handles asynchronous read/write operations.

How It Works:

1. A register (e.g., `**R1**`) holds data for processing.

2. The **src_data** multiplexer selects the input register (e.g., R1) and sends its value to the ALU.

3. The result is written back to the destination register (e.g., R2).

Binary Example:

Assume:

- R1 = 0000 1111 0000 1111 (binary).

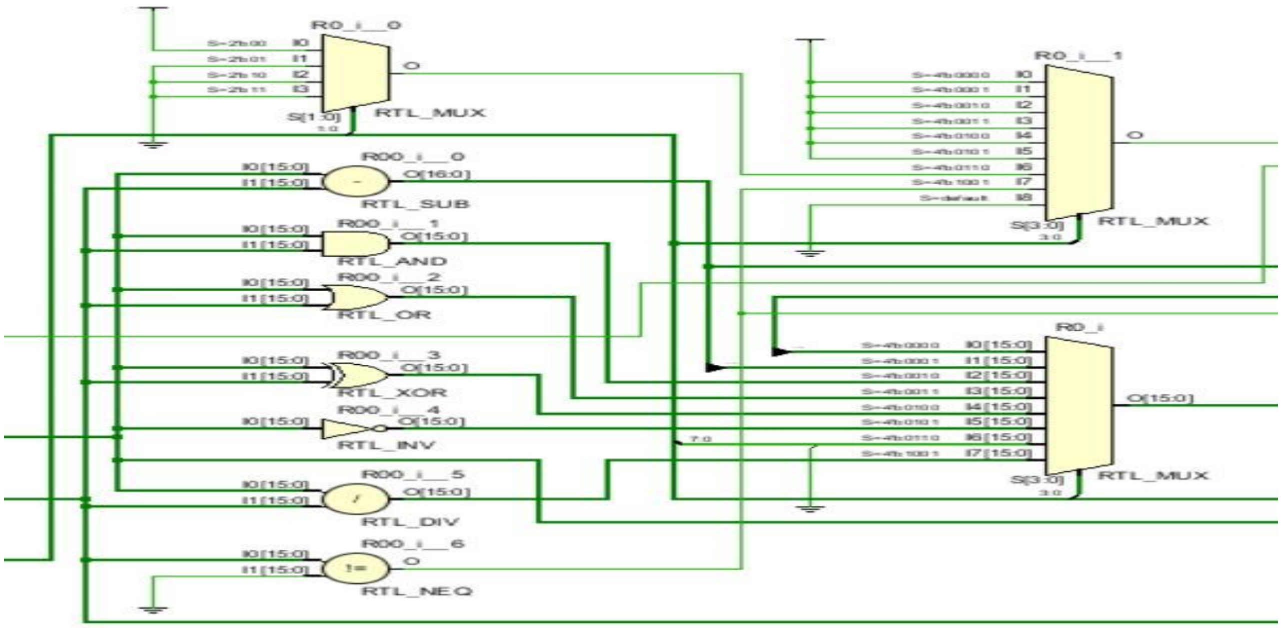
- R2 = 1111 0000 1111 0000 (binary).

When the ALU performs an addition operation:

- **Input to ALU**: R1 + R2 = 0000 1111 0000 1111 + 1111 0000 1111 0000.

- **Result:** 1111 1111 1111 1111 (stored in R2).

Part 3: ALU Operations



This part includes the Arithmetic Logic Unit (ALU), which performs arithmetic and logical computations.

Key Components:

- **ALU**: Executes operations like addition, subtraction, AND, OR, XOR, and comparisons.

- **RTL_MUX**: Selects the operation to be performed.

- **Flags**: Indicate conditions like zero or carry.

How It Works:

1. Input operands are received from the registers (e.g., R0 and R1).

2. The ALU performs the operation specified by the control signal.

3. The result is sent to the output register, and flags are updated.

Binary Examples:

- Addition:

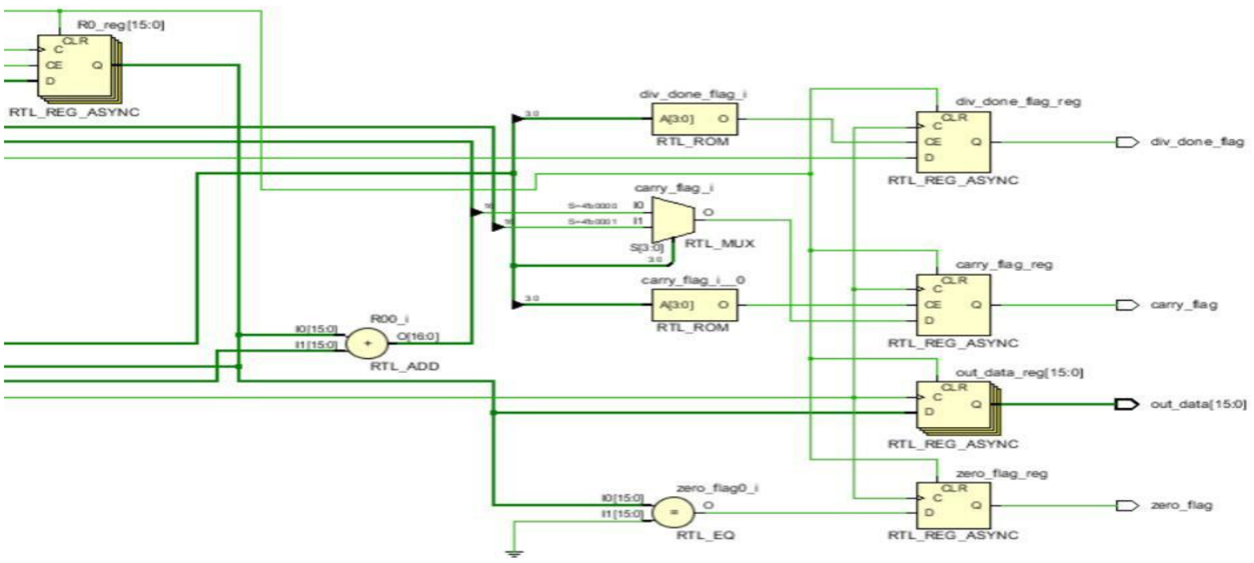
- **Input:** R0 = 0001 0010 1010 1111, R1 = 0000 1100 1011 0010.
- **Operation:** R0 + R1.
- **Result:** 0001 1111 0110 0010 (stored in R2).
- **Carry Flag:** Not set (no overflow).

- XOR:

- **Input:** R0 = 1010 1010 1010 1010, R1 = 0101 0101 0101 0101.
- **Operation:** R0 XOR R1.
- **Result:** 1111 1111 1111 1111.

- Equality Check:

- **Input:** R0 = 0000 0000 0000 0000, R1 = 0000 0000 0000 0000.
- **Operation:** Check if R0 == R1.
- **Result**: Zero Flag = 1 (set, as the result is zero).



Part 4: Flag Registers and Output

This part captures and stores computation results and updates the flags based on ALU outputs.

Key Components:

- **Flag Registers**: Store carry, zero, and division completion flags.

- **Output Register**: Holds the final result of computations.

- **RTL_MUX**: Selects which flag or output data to store.

How It Works:

1. The result from the ALU is written to the out_data register.

2. Flags are updated based on the result:

   - **Carry Flag**: Set if there is an overflow during addition.

   - **Zero Flag**: Set if the result is zero.

   - **Division Flag**: Set when a division operation completes.

3. The output data is sent to peripherals or memory.

Binary Example:

- ALU Result:

- **Input**: R0 = 1111 1111 1111 1111, R1 = 0000 0000 0000 0001.
- **Operation**: R0 + R1.
- **Result**: 1 0000 0000 0000 0000 (17 bits, overflow occurs).
- **Carry Flag**: Set to 1.

- Zero Flag:

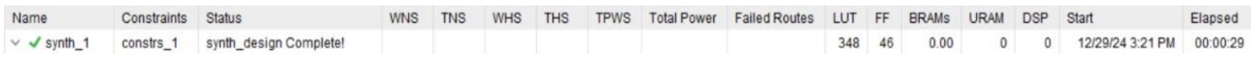
- **Input:** R0 = 0000 0000 0000 0000, R1 = 0000 0000 0000 0000.
- **Operation**: R0 – R1.
- **Result**: 0000 0000 0000 0000 (Zero Flag = 1).

This RTL diagram provides a clear representation of the microcontroller's operation. It starts with fetching instructions, decoding them, executing the instructions in the ALU, and storing the results in the registers. Binary examples illustrate how data flows and computations occur at each step.

---

5. SYNTHESIS

Resource Utilization



Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP	Start	Elapsed
✓ synth_1	constrs_1	synth_design Complete!								348	46	0.00	0	0	12/29/24 3:21 PM	00:00:29

Clock Frequency

The system operates at 100 MHz.

---

6. Conclusions

The RISC microcontroller successfully executed all supported instructions, demonstrating its reliable functionality. Its efficient design utilized minimal FPGA resources, highlighting resource optimization. The modular architecture ensures scalability for incorporating additional features in the future. This RISC microcontroller is highly efficient due to its simple, register-based design, where each component operates seamlessly to enable quick and accurate instruction execution. Moreover, the inclusion of flags provides vital feedback on computation results, making it well-suited for decision-making in control applications.