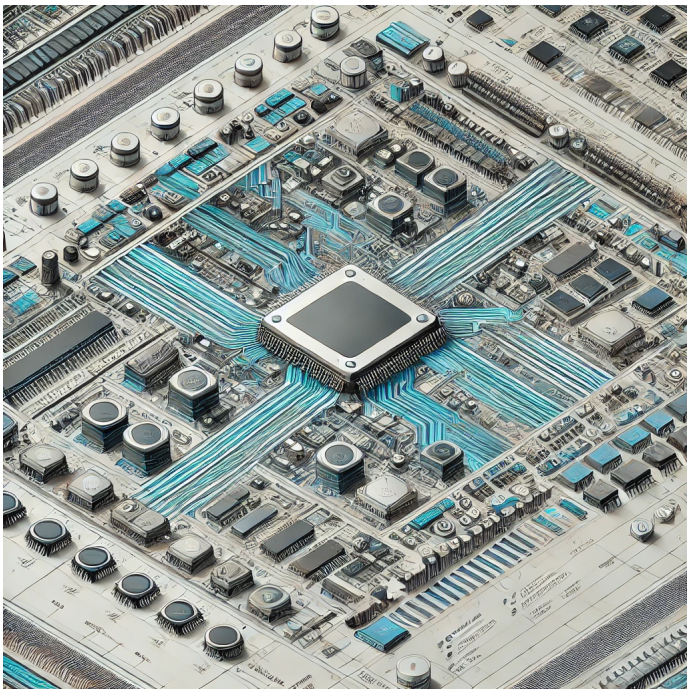
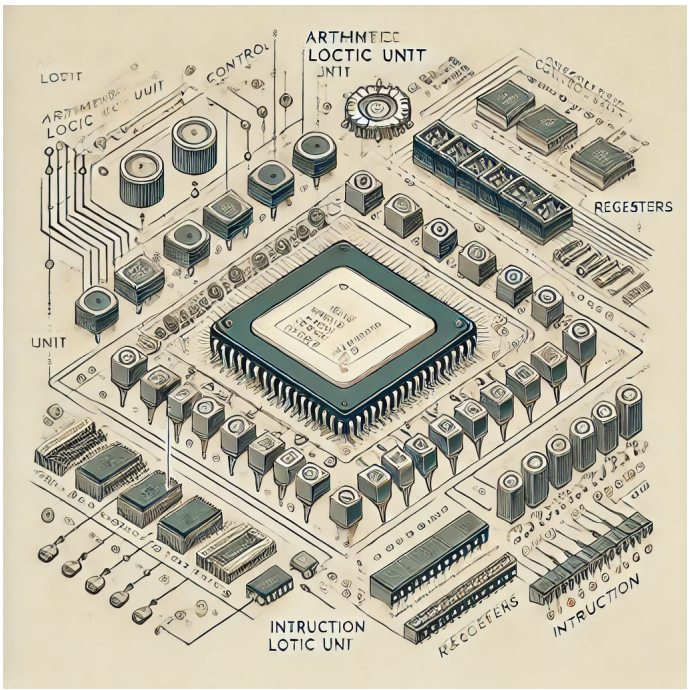
VLSI Hybrid Project Report



16-Bit RISC Processor



Vamshikrishna Redishetty

Date of Submission : 31 December 2024

Project-3

16-BIT RISC PROCESSOR

A **16-bit RISC (Reduced Instruction Set Computer) processor** is a compact and efficient computing architecture designed to execute instructions with simplicity and speed. It processes data in 16-bit chunks, meaning its registers, arithmetic logic unit (ALU), and data buses are 16 bits wide. This allows for a balanced trade-off between performance, cost, and power efficiency, making it ideal for embedded systems, controllers, and portable devices.

The processor typically consists of key components such as the **Arithmetic Logic Unit (ALU)**, a **Register File**, a **Control Unit**, a **Program Counter (PC)**, and memory interfaces. The ALU performs essential operations like addition, subtraction, logical AND/OR, and shifts, using inputs from the registers or immediate data. The register file contains a small number of general-purpose registers (e.g., 8–16), used for fast temporary data storage, minimizing memory access overhead. The Program Counter keeps track of the instruction address in memory and advances sequentially unless modified by jump or branch instructions.

The hallmark of RISC design is a streamlined instruction set. Each instruction performs a simple operation, such as load, store, arithmetic, or branch. Instructions are uniform in size, usually occupying a single 16-bit word, which simplifies instruction decoding and pipeline design. Common instruction formats include:

- **Arithmetic instructions** (e.g., ADD, SUB)
- **Logical operations** (e.g., AND, OR, XOR)
- **Load/Store instructions** for memory interaction
- **Branch instructions** for control flow

A 16-bit RISC processor often employs a **pipelined architecture** to execute multiple instructions simultaneously, increasing throughput. For example, while one instruction is fetched, another is decoded, and a third is executed. This parallelism is a key advantage of RISC design, ensuring higher performance per clock cycle.

The processor connects to memory using a 16-bit data bus, enabling it to read or write 16 bits of data per cycle. Address buses are often wider than 16 bits to support larger memory spaces, depending on the design's addressing mode. The memory access is typically optimized for simplicity, with load/store operations

directly transferring data between registers and memory.

The **Control Unit** orchestrates the operation of the processor, generating control signals to coordinate data flow between components. In a RISC processor, control logic is often hardwired rather than microprogrammed, which enhances speed and reduces complexity.

A 16-bit RISC processor is commonly used in applications requiring compact size, low power consumption, and real-time performance. These include **embedded systems**, **digital signal processing (DSP)**, and **automotive controllers**. Its simplicity and deterministic performance make it ideal for systems where reliability and predictability are critical.

In summary, a 16-bit RISC processor is a highly efficient and streamlined computing architecture designed to execute operations with minimal complexity and maximum speed. Its focus on simplicity, regularity, and parallelism enables it to deliver robust performance in a wide range of applications.

Decoding of the 16-Bit Instruction :

This table represents the decoding of a 16-bit instruction into different parts. It is Decoded into different parts in different cases. In Case 1, it is decoded as: The most significant bits, from bit 15 to bit 11, encode the Opcode, which specifies the operation to be performed. Bits 10 to 8 are designated for Ra, identifying the destination register. Bits 7 to 5 specify Rb, the first source register, and bits 4 to 2 represent Rd, the second source register. Finally, bits 1 to 0 are unused or reserved, indicated by X. In case 2, it is decoded as : The most significant bits 15 to 11 bit, encode the Opcode then 3 bits are Rd and remaining bits into Immediate.

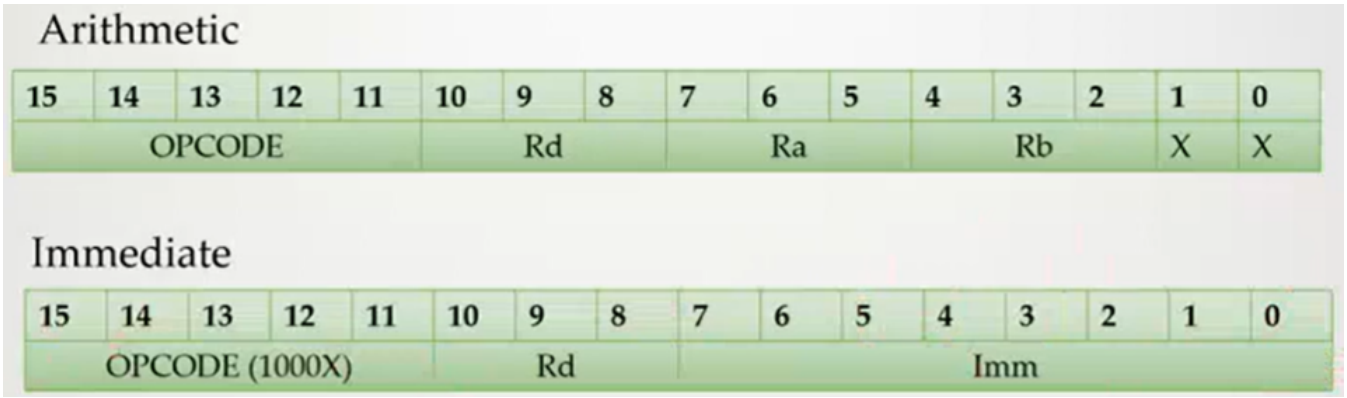


Figure 1: Table of Instruction Decoding

Operations performed by RISC Processor with OPCODE:

OPCODE					ALU Operation
0	0	0	0	0	Addition (Unsigned)
0	0	0	0	1	Addition (Signed)
0	0	0	1	X	Bitwise OR
0	0	1	0	X	Bitwise AND
0	0	1	1	X	Bitwise XOR
0	1	0	0	X	Bitwise NOT
0	1	0	1	X	Read Memory
0	1	1	0	X	Write Memory
1	0	0	0	0	Load Register (Low)
1	0	0	0	1	Load Register (High)
1	0	0	1	0	Compare (Unsigned)
1	0	0	1	1	Compare (Signed)
1	0	1	0	X	Shift Left
1	0	1	1	X	Shift Right
1	1	0	0	0	Jump (Imm.)
1	1	0	0	1	Jump (Register)

Figure 2: OPERATIONS PERFORMED OF PROCESSOR

Description of the 16 Bit RISC Processor Design :

Instruction Decoder Module:

The Instruction_Decoder module is a critical component of the RISC processor, responsible for interpreting a 16-bit instruction and generating control signals to drive the processor's operation. This module takes three primary inputs: a clock signal (I_clk), an enable signal (I_enable), and a 16-bit instruction (I_instruction). On the negative edge of the clock, if the I_enable signal is asserted, the module decodes the instruction and extracts various control fields. The most significant 5 bits of the instruction (I_instruction[15:11]) are assigned to O_ALU, which specifies the operation to be performed by the Arithmetic Logic Unit (ALU). The next three 3-bit fields (I_instruction[10:8], I_instruction[7:5], and I_instruction[4:2]) are decoded as the addresses of the destination register (O_Ra), register A (O_Rb), and register B (O_Rd), respectively. Additionally,

the least significant 8 bits of the instruction (I_instruction[7:0]) are extracted as the immediate value (O_immediate) for immediate-based operations. The module also generates a O_WriteEnable signal based on the decoded ALU operation. Specific operations (e.g., operations with opcode values 4'b0111, 4'b1100, or 4'b1101) disable writing to the registers by setting O_WriteEnable to 0, while other operations enable writes. This design ensures precise control of data flow and register updates, making the instruction decoder an essential part of the processor's execution pipeline.

Code of the Instruction Decoder Module:

```
`timescale 1ns / 1ps
module Instruction_Decoder(
//input ports
input I_clk,
input I_enable,
input [15:0] I_instruction,
//output ports
output reg [4:0] O_ALU, // Operation for ALU
output reg [2:0] O_Ra , // Address of Destination Register
output reg [2:0] O_Rb , // Address of Register A
output reg [2:0] O_Rd , // Address of Register B
output reg [15:0] O_immediate, // Address of Immediate
output reg O_WriteEnable // Write Enable
);

initial begin
    O_ALU <= 0;
    O_Rd <= 0;
    O_Ra <= 0;
    O_Rb <= 0;
    O_immediate <= 0;
    O_WriteEnable <= 0;
end

always@(negedge I_clk) begin
    if(I_enable) begin
        O_ALU <= I_instruction[15:11];
        O_Ra <= I_instruction[10:8];
        O_Rb <= I_instruction[7:5];
        O_Rd <= I_instruction[4:2];
        O_immediate <= I_instruction[7:0];
        case(I_instruction[15:11])
            4'b0111 : O_WriteEnable <= 0;
            4'b1100 : O_WriteEnable <= 0;
            4'b1101 : O_WriteEnable <= 0;
            default : O_WriteEnable <= 1;
        endcase
    end
end
endmodule
```

Control Unit Module:

The `ControlUnit` module is a sequential logic component of the RISC processor that orchestrates the operation of various stages in the processor pipeline. It determines the order in which operations are executed and ensures proper coordination between different units, such as fetching instructions, decoding them, reading from and writing to registers, executing ALU operations, and accessing memory.

The module operates based on a 6-bit state machine. On the negative edge of the clock (`I_clk`), the state transitions sequentially through predefined states unless reset (`I_reset`) is asserted. When reset is activated, the state reverts to the initial state (`6'b000001`). The state transitions occur in a cyclic manner, advancing from one state to the next in a fixed sequence: `6'b000001 -> 6'b000010 -> 6'b000100 -> 6'b001000 -> 6'b010000 -> 6'b100000 -> 6'b000001`. This cycle ensures that each operation in the processor pipeline is performed in the correct order.

Each state corresponds to a specific stage of the processor's operation:

1. **Fetch** (`O_Fetch`): Enabled when the least significant bit of the state is high (`state[0]`), signaling that the instruction fetch phase is active.

2. **Decode** (`O_Decoder`): Enabled when `state[1]` is high, allowing the instruction decoder to process the fetched instruction.

3. **Register Read** (`O_RegisterRead`): Enabled when either `state[2]` or `state[4]` is high, allowing register values to be read during these stages.

4. **ALU Operation** (`O_EnableALU`): Enabled when `state[3]` is high, signaling that an arithmetic or logical operation is being performed.

5. **Register Write** (`O_RegisterWrite`): Enabled when `state[4]` is high, indicating that the result of an operation is being written back to a register.

6. **Memory Access** (`O_EnableMemory`): Enabled when the most significant bit of the state (`state[5]`) is high, activating memory operations such as loading or storing data.

This sequential control ensures that all operations in the pipeline are executed in the correct order and without overlap, maintaining the deterministic nature of the processor's execution. The module is designed to be simple yet effective, providing a clear flow for operations in the RISC architecture.

Code of the Control Unit Module:

```
`timescale 1ns / 1ps
// This Module directs in which order things are going to happen and makes operations in
    a sequential Way
module ControlUnit(
// Input Ports
input I_clk,
input I_reset,
// Output Ports
output O_Fetch, // Enable Fetch
output O_Decoder, // Enable Decoder
output O_RegisterRead, // Enable Read
output O_RegisterWrite, // Enable Write
output O_EnableALU, // Enable ALU
output O_EnableMemory // Enable Memory
);
reg [5:0] state;
initial state <= 6'b000001;
always @(negedge I_clk) begin
    if(I_reset) state <= 6'b0000001;
    else begin
        case(state)
            6'b000001: state <= 6'b000010;
            6'b000010: state <= 6'b000100;
            6'b000100: state <= 6'b001000;
            6'b001000: state <= 6'b010000;
            6'b010000: state <= 6'b100000;
            default : state <= 6'b000001;
        endcase
    end
end
assign O_Fetch = state[0];
assign O_Decoder = state[1];
assign O_RegisterRead = state[2] | state[4];
assign O_EnableALU = state[3];
assign O_RegisterWrite = state[4];
assign O_EnableMemory = state[5];
endmodule
```

Program Counter Module:

The `ProgramCounter` module is a fundamental component of the RISC processor, responsible for managing the flow of instruction execution by keeping track of the address of the next instruction to be executed. This module uses a 16-bit register (`O_pc`) to store the current program counter value, which can be updated based on specific control signals represented by the `I_opcode` input.

Functionality :

- **Clock Input** (`I_clk`): The `ProgramCounter` updates its value on the negative edge of the clock signal, ensuring synchronization with other processor components.

- **Opcode Input (`I_opcode`)**: The 2-bit `I_opcode` determines how the program counter (`O_pc`) is updated:
  - `2'b00`: Retains the current value of the program counter, effectively pausing any advancement.
  - `2'b01`: Increments the program counter by 1, pointing to the next sequential instruction.
  - `2'b10`: Loads a new value from the `I_pc` input, allowing for branching or jumping to a specific instruction address.
  - `2'b11`: Resets the program counter to zero, typically used to reinitialize the processor or handle specific control flow events.
  - **Default**: As a safeguard, sets the program counter to zero for undefined opcodes.

Design Details :

- **Initialization**: The program counter is initialized to zero at the start of the simulation or operation, ensuring a known starting state for the processor.
- **Update Mechanism**: The value of `O_pc` is updated based on the `I_opcode` input during each negative clock edge, providing flexibility to support various control flow operations like sequential execution, jumps, and resets.

This module plays a crucial role in the processor's control flow, enabling sequential instruction execution and supporting dynamic branching or reset functionality. By implementing the program counter in a compact and efficient manner, the design ensures smooth and predictable operation within the processor pipeline.

Code of the Control Unit Module:

```
1   `timescale 1ns / 1ps
2   module ProgramCounter(
3   // Input Ports
4   input I_clk ,
5   input [1:0] I_opcode ,
6   input [15:0] I_pc ,
7   // Output Ports
8   output reg [15:0] O_pc
9   );
10  initial begin
11      O_pc <= 0;
12  end
13  always@(negedge I_clk) begin
14      case(I_opcode)
15          2'b00 : O_pc <= O_pc ;
16          2'b01 : O_pc <= O_pc + 1 ;
```

```
            2'b10 : O_pc <= I_pc ;
            2'b11 : O_pc <= 0 ;
            default : O_pc <= 0;
        endcase
    end
endmodule
```

ALU Module:

The `ALU` (Arithmetic Logic Unit) module is a key component of the RISC processor, responsible for performing arithmetic, logical, and control operations. It processes two 16-bit input data values (`I_dataA` and `I_dataB`), along with an 8-bit immediate value (`I_immediate`), based on a 5-bit opcode (`I_opcode`). The ALU generates a 16-bit output (`O_data`) and a branching signal (`O_shouldbranch`), enabling versatile computational and control functionalities.

Operation Selection: The `I_opcode` determines the operation to be performed, with a combination of a 4-bit `opcode` (`I_opcode[4:1]`) and a `check` bit (`I_opcode[0]`). The operations include arithmetic (e.g., addition, subtraction), logical (e.g., AND, OR, XOR), comparison, and control flow instructions (e.g., jump).

Arithmetic Operations:

- **Addition (`Add`):** Adds `I_dataA` and `I_dataB`. The `check` bit selects between signed and unsigned addition.
- **Subtraction (`Sub`):** Subtracts `I_dataB` from `I_dataA`, with `check` selecting signed or unsigned subtraction.

Logical Operations:

- **OR, AND, XOR, NOT:** Performs bitwise logical operations on `I_dataA` and `I_dataB`.
- **NOT:** Computes the bitwise complement of `I_dataA`.

Immediate Load and Shift:

- **Load Register (`LRG`):** Loads an 8-bit immediate value into the result, with `check` determining the position (upper or lower byte).
- **Left Shift (`LSR`):** Shifts `I_dataA` left by the number of positions specified by the lower 4 bits of `I_dataB`.
- **Right Shift (`RSR`):** Shifts `I_dataA` right by the number of positions specified by the lower 4 bits of `I_dataB`.

Comparison (COM): Compares I_dataA and I_dataB for equality, greater than, less than, and zero conditions. Signed or unsigned comparison is determined by the check bit. The results are stored in the least significant bits of the result register.

Branch and Jump Operations:

- **Jump Absolute (JA):** Outputs an immediate or data value and asserts the O_shouldbranch signal to indicate a jump.
- **Jump Relative (JR):** Outputs I_dataA and conditionally asserts O_shouldbranch based on a value in I_dataB indexed by the check bit and the lower bits of the immediate value.

Outputs:

- **O_data:** The lower 16 bits of the result register, representing the operation result.
- **O_shouldbranch:** Indicates whether a branch operation is required, crucial for control flow instructions.

The ALU module serves as the computational core of the RISC processor, supporting a wide range of operations with efficient control flow integration. Its modular design, use of local parameters for operation codes, and ability to handle signed and unsigned arithmetic make it versatile and robust, ensuring seamless functionality across diverse processing tasks.

Code of the ALU Module:

```
1   `timescale 1ns / 1ps
2   module ALU(
3   // Input Ports
4   input I_clk ,
5   input I_enable ,
6   input [4:0] I_opcode ,
7   input [15:0] I_dataA ,
8   input [15:0] I_dataB ,
9   input [7:0] I_immediate ,
10  // Output Ports
11  output [15:0] O_data,
12  output reg O_shouldbranch
13  );
14  reg [17:0] result ;
15  wire [3:0] opcode ;
16  wire check ;
17
18  localparam Add = 0,
19             Sub = 1,
20             OR = 2,
21             AND = 3,
22             XOR = 4,
```

```
            NOT = 5,
            LRG = 8,
            COM = 9,
            LSR = 10,
            RSR = 11,
            JA = 12,
            JR = 13;

    // assign statements
    assign check = I_opcode[0];
    assign opcode = I_opcode[4:1];
    // Assigning the output
    assign O_data = result[15:0];
    // Initial Block
    initial begin
        result = 0;
    end
    always@(negedge I_clk) begin
        if(I_enable) begin
            case(opcode)
                Add : begin
                      result <= (check?($signed(I_dataA)+$signed(I_dataB)):(I_dataA+I_dataB));
                      O_shouldbranch <= 0;
                      end
                Sub : begin
                      result <= (check?($signed(I_dataA)-$signed(I_dataB)):(I_dataA-I_dataB));
                      O_shouldbranch <= 0;
                      end
                OR : begin
                      result <= I_dataA | I_dataB;
                      O_shouldbranch <= 0;
                      end
                AND : begin
                      result <= I_dataA & I_dataB;
                      O_shouldbranch <= 0;
                      end
                XOR : begin
                      result <= I_dataA ^ I_dataB;
                      O_shouldbranch <= 0;
                      end
                NOT : begin
                      result <= ~I_dataA;
                      O_shouldbranch <= 0;
                      end
                LRG : begin
                      result <= (check ? ({I_immediate,8'h00}):({8'h00,I_immediate}));
                      O_shouldbranch <= 0;
                      end
                COM : begin
                        if(check)begin
                        result[0] <= ($signed(I_dataA) == $signed(I_dataB))? 1 : 0;
                        result[1] <= ($signed(I_dataA) == 0)? 1 : 0;
                        result[2] <= ($signed(I_dataB) == 0)? 1 : 0;
                        result[3] <= ($signed(I_dataA) > $signed(I_dataB))? 1 : 0;
                        result[4] <= ($signed(I_dataA) < $signed(I_dataB))? 1 : 0;
                        end
                        else begin
                        result[0] <= (I_dataA) == (I_dataB)? 1 : 0;
                        result[1] <= (I_dataA) == 0? 1 : 0;
                        result[2] <= (I_dataB) == 0? 1 : 0;
                        result[3] <= (I_dataA) > (I_dataB)? 1 : 0;
```

```
                    result[4] <= (I_dataA) <(I_dataB)? 1 : 0;
                    end
                    O_shouldbranch <= 0;
                end
            LSR : begin
                result <= (I_dataA)<<(I_dataB[3:0]);
                O_shouldbranch <= 0;
                end
            RSR : begin
                result <= (I_dataA)>>(I_dataB[3:0]);
                O_shouldbranch <= 0;
                end
            JA : begin
                result <= check ? I_dataA : I_immediate ;
                O_shouldbranch <= 1;
                end
            JR : begin
                result <= I_dataA;
                O_shouldbranch <= I_dataB[{check,I_immediate[1:0]}];
                end
        endcase
    end
end
endmodule
```

Register Handler Module:

The `RegisterHandler` module is a core component in the RISC processor architecture, designed to manage data read and write operations for a set of 8 general-purpose 16-bit registers. This module provides efficient handling of register data, enabling simultaneous reading from two registers and conditional writing to a specific register, all governed by clock and control signals.

Inputs: The module has the following inputs:

- `I_clk`**:** Clock signal that synchronizes register operations.
- `I_enable`**:** Enable signal that activates the module for read or write operations.
- `I_WriteEnable`**:** Write enable signal that determines whether data is written to the registers.
- `I_Ra`**,** `I_Rb`**, and** `I_Rd`**:** 3-bit register addresses specifying the source registers for reading (`I_Ra` and `I_Rb`) and the destination register for writing (`I_Rd`).
- `I_data`**:** 16-bit data input for writing to the destination register.

Outputs: The module provides two outputs:

- `O_dataA` **and** `O_dataB`**:** 16-bit outputs holding the values read from the registers addressed by `I_Ra` and `I_Rb`, respectively.

Functionality: At initialization, all registers are cleared to 0, and the outputs (`O_dataA` and `O_dataB`) are set to zero. On the negative edge of the clock (`negedge I_clk`), the module performs the following operations if `I_enable` is asserted:

1. If `I_WriteEnable` is high, the value of `I_data` is written to the register specified by `I_Rd`.

2. The values of the registers addressed by `I_Ra` and `I_Rb` are assigned to `O_dataA` and `O_dataB`, respectively.

This dual-read and single-write design facilitates efficient data transfer between the registers and other components of the processor. By ensuring synchronized operations and modular design, the `RegisterHandler` module contributes significantly to the overall functionality and performance of the RISC processor.

Code of the Register Handler Module:

```
`timescale 1ns / 1ps
module RegisterHandler(
// Input Ports
input I_clk,
input I_enable,
input I_WriteEnable,
input [2:0] I_Ra,
input [2:0] I_Rb,
input [2:0] I_Rd,
input [15:0] I_data,
// Output Ports
output reg [15:0] O_dataA,
output reg [15:0] O_dataB
);

reg [15:0] register [7:0];
integer count;
initial begin
    O_dataA <= 16'b0000000000000000;
    O_dataB <= 16'b0000000000000000;
    for(count = 0 ; count < 8 ; count=count+1) begin
        register[count] = 16'b0000000000000000;
    end
end
always@(negedge I_clk) begin
    if(I_enable) begin
        if(I_WriteEnable) register[I_Rd] <= I_data ;

        O_dataA <= register[I_Ra];
        O_dataB <= register[I_Rb];
    end
end
endmodule
```

RAM Module:

The `RAM` module is a simple memory unit designed for a RISC processor to store and retrieve 16-bit data. It supports reading and writing operations synchronized to a clock signal, with a configurable memory size and initial content. This module is integral to providing the processor with both program instructions and data storage capabilities.

Inputs: The module has the following inputs:

- `I_clk`**:** Clock signal that synchronizes read and write operations.
- `I_WriteEnable`**:** Control signal that determines whether a write operation occurs during the current clock cycle.
- `I_address`**:** A 16-bit address specifying the memory location for reading or writing.
- `I_data`**:** A 16-bit input representing the data to be written to the specified memory location.

Outputs: The module provides the following output:

- `O_data`**:** A 16-bit output that holds the value read from the memory location specified by `I_address`.

Functionality: The `RAM` module initializes with predefined values in specific memory locations. The initial content can represent program instructions or data, depending on the processor's requirements. The memory consists of nine 16-bit locations (`memory[0]` through `memory[8]`), and the initial values are hardcoded into the module for testing and demonstration purposes.

On the negative edge of the clock (`negedge I_clk`), the following operations occur:

1. If `I_WriteEnable` is high, the value of `I_data` is written to the memory location specified by `I_address`.
2. The content of the memory location addressed by `I_address` is assigned to `O_data` for read operations.

This design allows simultaneous read and conditional write operations. The `RAM` module serves as both an instruction memory and a data storage unit, enabling seamless integration with the processor's pipeline.

Code of the RAM Module:

```
`timescale 1ns / 1ps
module RAM(
// Input Ports
input I_clk,
input I_WriteEnable,
input [15:0] I_address,
input [15:0] I_data,
// Output Ports
output reg [15:0] O_data
);
reg [15:0] memory [8:0];
initial begin
    memory[0] = 16'b1000000011111110;
    memory[1] = 16'b1000100111101101;
    memory[2] = 16'b0010001000100000;
    memory[3] = 16'b1000001100000001;
    memory[4] = 16'b1000010000000001;
    memory[5] = 16'b0000001101110000;
    memory[6] = 16'b1100000000000101;
    memory[7] = 0;
    memory[8] = 0;
    O_data = 16'b0000000000000000;
end
always@(negedge I_clk) begin
    if(I_WriteEnable) begin
        memory[I_address[15:0]] <= I_data;
    end
    O_data <= memory[I_address[15:0]];
end
endmodule
```

Description of the 16 Bit RISC Processor Design Test Benches :

Test Bench of Instruction Decoder Module:

The module `TBDecoder` is a testbench designed to verify the functionality of the `Instruction_Decoder` module. It simulates the behavior of the decoder by providing specific input signals and observing the outputs. The key features of this testbench are as follows:

- **Inputs:**
  - `I_instruction`: A 16-bit register representing the instruction to be decoded.
  - `I_clk`: A clock signal toggling every 5 time units.
  - `I_enable`: A control signal enabling the decoder.
- **Outputs:**
  - `O_ALU`: A 5-bit wire indicating the ALU operation.

- `O_Rd`: A 3-bit wire specifying the destination register.
- `O_Ra` and `O_Rb`: 3-bit wires specifying the source registers.
- `O_immediate`: A 16-bit wire representing the immediate value.
- `O_WriteEnable`: A wire indicating whether a write operation is enabled.

- **Initialization:** At the start of the simulation, all inputs are initialized:
  - `I_clk` is set to 0.
  - `I_enable` is set to 0.
  - `I_instruction` is set to `16'b0000000000000000`.

- **Test Sequence:**
  - After 10 time units, `I_instruction` is assigned the value `16'b0001010110000100`.
  - After another 10 time units, `I_enable` is set to 1.

- **Clock Generation:** The clock signal `I_clk` toggles every 5 time units, creating a periodic waveform.

This testbench systematically stimulates the `Instruction_Decoder` module to validate its functionality by simulating different scenarios.

Code of Instruction Decoder Test Bench Module:

```
1   `timescale 1ns / 1ps
2   module TBDecoder();
3   reg [15:0] I_instruction;
4   reg I_clk;
5   reg I_enable;
6   wire [4:0] O_ALU;
7   wire [2:0] O_Rd ;
8   wire [2:0] O_Ra ;
9   wire [2:0] O_Rb ;
10  wire [15:0] O_immediate;
11  wire O_WriteEnable;
12  Instruction_Decoder DecoderBlock(
13  .I_instruction (I_instruction),
14  .I_clk (I_clk ),
15  .I_enable (I_enable ),
16  .O_ALU (O_ALU ),
17  .O_Rd (O_Rd ),
18  .O_Ra (O_Ra ),
19  .O_Rb (O_Rb ),
20  .O_immediate (O_immediate ),
21  .O_WriteEnable (O_WriteEnable)
22  );
23  initial begin
24      I_clk = 0;
```

```
        I_enable = 0;
        I_instruction = 16'b0000000000000000;

        #10 I_instruction = 16'b0001010110000100;
        #10 I_enable = 1;
    end
    always begin
        #5 I_clk = ~I_clk;
    end
    endmodule
```

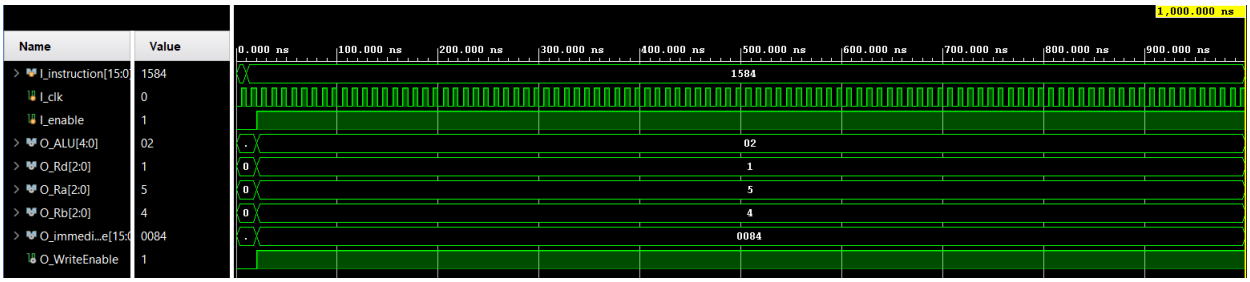


Figure 3: Output Wave Form of Instruction Decoder

Test Bench of Register Handler Module:

The module `TBRegisterHandler` serves as a testbench for verifying the functionality of the `RegisterHandler` module. It systematically stimulates the module with various inputs and observes its outputs. The key features of the testbench are as follows:

- **Inputs:**
  - `I_clk`: A clock signal that toggles every 5 time units.
  - `I_enable`: A control signal that enables the `RegisterHandler`.
  - `I_WriteEnable`: A signal that controls whether data is written to the register file.
  - `I_Ra` and `I_Rb`: 3-bit register addresses for reading data.
  - `I_Rd`: A 3-bit register address for writing data.
  - `I_data`: A 16-bit input used as the data to be written to the register.
- **Outputs:**
  - `O_dataA`: A 16-bit output corresponding to the data read from the register addressed by `I_Ra`.
  - `O_dataB`: A 16-bit output corresponding to the data read from the register addressed by `I_Rb`.

- **Initialization:** All input signals are initialized at the start of the simulation:
  - `I_clk` is set to 0.
  - `I_enable`, `I_WriteEnable`, and register addresses (`I_Ra`, `I_Rb`, `I_Rd`) are set to 0.
  - `I_data` is initialized to 0.
- **Test Sequence:**
  - At time 7, the enable signal is asserted, and specific register addresses and data are provided.
  - The write enable signal is toggled to simulate various write operations.
  - At time 50, specific registers are read by assigning values to `I_Ra` and `I_Rb`.
  - The simulation ends at time 117 with a call to `$finish`.
- **Clock Generation:** The clock signal `I_clk` toggles every 5 time units, providing a periodic waveform required for synchronous operation.

This testbench exercises the `RegisterHandler` module under different scenarios, validating its ability to correctly handle register read and write operations.

Code of Register Handler Test Bench Module:

```
1  `timescale 1ns / 1ps
2  module TBRegisterHandler();
3  reg I_clk;
4  reg I_enable;
5  reg I_WriteEnable;
6  reg [2:0] I_Ra;
7  reg [2:0] I_Rb;
8  reg [2:0] I_Rd;
9  reg [15:0] I_data;
10 wire [15:0] O_dataA;
11 wire [15:0] O_dataB;
12
13 RegisterHandler HandlerBlock(
14 .I_clk (I_clk ),
15 .I_enable (I_enable ),
16 .I_WriteEnable (I_WriteEnable),
17 .I_Ra (I_Ra ),
18 .I_Rb (I_Rb ),
19 .I_Rd (I_Rd ),
20 .I_data (I_data ),
21 .O_dataA (O_dataA ),
22 .O_dataB (O_dataB )
23 );
24
25 initial begin
```

```
        I_clk = 0 ;
        I_enable =0 ;
        I_WriteEnable =0 ;
        I_Ra = 0;
        I_Rb = 0;
        I_Rd = 0;
        I_data = 0;

        // Start TestCase
        #7
        I_enable = 1;
        I_Rb = 3'b001;
        I_Ra = 3'b000;
        I_Rd = 3'b000;
        I_data = 16'hFFFF;
        I_WriteEnable = 1'b1;

        #10
        I_WriteEnable = 1'b0;
        I_Rd = 3'b010;
        I_data = 16'h2222;

        #10
        I_WriteEnable = 1'b1;

        #10
        I_data = 16'h3333;

        #10
        I_WriteEnable = 1'b0;
        I_Rd = 3'b000;
        I_data = 16'hFEED;

        #10
        I_Rd = 3'b100;
        I_data = 16'h4444;

        #10
        I_WriteEnable = 1'b1;

        #50
        I_Ra = 3'b100;
        I_Rb = 3'b100;

        #20
        $finish;
    end

    always begin
        #5 I_clk = ~I_clk;
    end
    endmodule
```

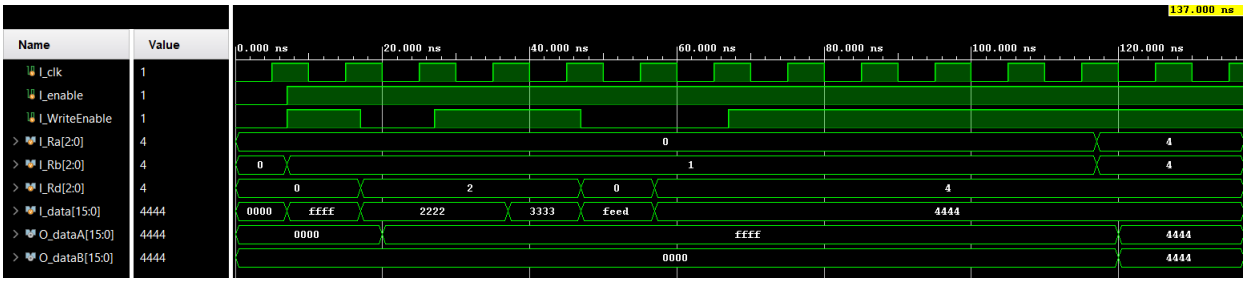


Figure 4: Output Wave Form of Register Handler

Test Bench for RISC Processor Module:

The module `RISCProcessor` models a basic Reduced Instruction Set Computer (RISC) processor. It consists of several interconnected modules that handle different stages of the processor pipeline: instruction fetching, decoding, execution, and memory access. Below is a breakdown of its key components:

- **Inputs:**
  - `clk`: Clock signal controlling the synchronous operations.
  - `reset`: A reset signal to initialize the processor state.
  - `RAM_WriteEnable`: Signal enabling write operations to RAM.
  - `dataI`: A 16-bit input data register.
- **Outputs:**
  - `Ra`, `Rb`, `Rd`: 3-bit wire addresses for source and destination registers.
  - `dataA`, `dataB`, `dataD`, `dataO`: 16-bit data wires for ALU operations, register read, and memory read/write.
  - `opcode`, `opcodePC`: Various control signals and opcode values for different processor operations.
  - `pcO`: 16-bit wire for the updated program counter value.
- **Instantiations and Interconnections:**
  - `Instruction_Decoder`: Decodes the instruction fetched from memory.
  - `ControlUnit`: Manages the control signals for different stages like Fetch, Decode, RegisterRead/Write, ALU operations, and Memory access.
  - `ALU`: Performs arithmetic and logic operations based on the opcode and inputs.
  - `ProgramCounter`: Updates the program counter value based on control signals and ALU results.

- `RegisterHandler`: Manages read and write operations to the register file.
- `RAM`: Interfaces with memory for data read/write operations.

- **Functional Flow:**
  - At initialization, `clk` is set to 0, and `reset` is activated to clear the processor state.
  - After 20 time units, the reset signal is de-asserted, allowing the processor to operate normally.
  - The processor operates synchronously with the clock, processing instructions and performing memory operations as per control signals.
- **Clock and Reset Handling:** The processor uses a clock signal to synchronize its operations, and a reset signal initializes its state to a known starting point.

This RISC processor supports basic functionalities of fetching instructions, decoding them, executing computations, handling register reads/writes, and interacting with memory.

Code of RISC Processor Bench Module:

```
1   `timescale 1ns / 1ps
2   module RISCProcessor();
3   reg clk;
4   reg reset;
5   reg RAM_WriteEnable = 0;
6   reg [15:0] dataI = 0;
7
8   wire [2:0] Ra;
9   wire [2:0] Rb;
10  wire [2:0] Rd;
11  wire [15:0] dataA;
12  wire [15:0] dataB;
13  wire [15:0] dataD;
14  wire [15:0] dataO;
15  wire [4:0] opcode;
16  wire [1:0] opcodePC;
17  wire [7:0] immediate;
18  wire [15:0] pcO;
19
20  wire shouldbranch;
21  wire Fetch ;
22  wire Decoder ;
23  wire RegisterRead ;
24  wire RegisterWrite;
25  wire EnableALU ;
26  wire EnableMemory ;
27  wire RegisterWE;
28  wire update;
```

```
29
30      assign RegisterWrite = RegisterWE & update ;
31      assign opcodePC = (reset) ? 2'b11 : ((shouldbranch) ? 2'b10 :((EnableMemory) ? 2'b01 :2'
            b00));
32
33      // Instantiations
34      Instruction_Decoder DecoderBlock(
35      .I_instruction (dataO ),
36      .I_clk (clk ),
37      .I_enable (Decoder ),
38      .O_ALU (opcode ),
39      .O_Rd (Rd ),
40      .O_Ra (Ra ),
41      .O_Rb (Rb ),
42      .O_immediate (immediate ),
43      .O_WriteEnable (RegisterWE)
44      );
45
46      ControlUnit Main_ControlBLock(
47      .I_clk (clk ),
48      .I_reset (reset ),
49      .O_Fetch (Fetch ),
50      .O_Decoder (Decoder ),
51      .O_RegisterRead (RegisterRead ),
52      .O_RegisterWrite(update ),
53      .O_EnableALU (EnableALU ),
54      .O_EnableMemory (EnableMemory )
55      );
56
57      ALU Main_ALU(
58      .I_clk (clk ) ,
59      .I_enable (EnableALU ) ,
60      .I_opcode (opcode ) ,
61      .I_dataA (dataA ) ,
62      .I_dataB (dataB ) ,
63      .I_immediate (immediate ) ,
64      .O_data (dataD ) ,
65      .O_shouldbranch(shouldbranch)
66      );
67
68      ProgramCounter Main_ProgramCounter(
69      .I_clk (clk ),
70      .I_opcode(opcodePC),
71      .I_pc (dataD ),
72      .O_pc (pcO )
73      );
74
75      RegisterHandler Main_HandlerBlock(
76      .I_clk (clk ),
77      .I_enable (RegisterRead),
78      .I_WriteEnable(RegisterWrite),
79      .I_Ra (Ra ),
80      .I_Rb (Rb ),
81      .I_Rd (Rd ),
82      .I_data (dataI ),
83      .O_dataA (dataA ),
84      .O_dataB (dataB )
85      );
86
87      RAM Main_RAM(
88      .I_clk (clk ),
```

```
    .I_WriteEnable(RAM_WriteEnable),
    .I_address (pcO ),
    .I_data (dataI ),
    .O_data (dataO )
    );

    initial begin
        clk = 0;
        reset = 1;
        #20
        reset = 0;
    end

    always begin
        #5 clk = ~clk;
    end

    endmodule
```

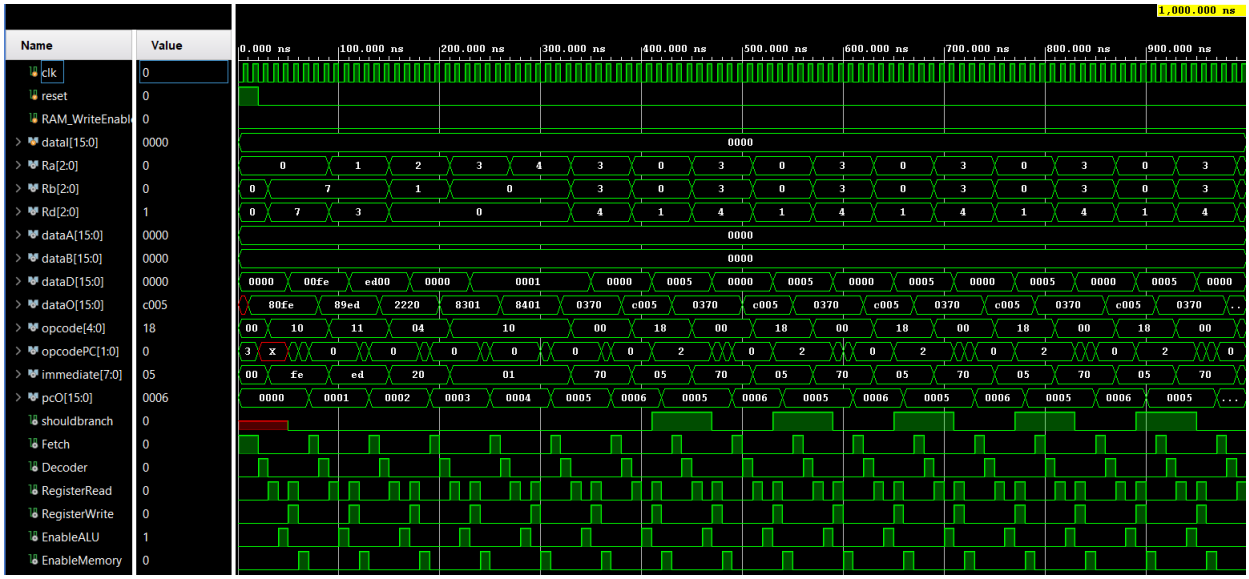


Figure 5: Output Wave Form of RISC Processor

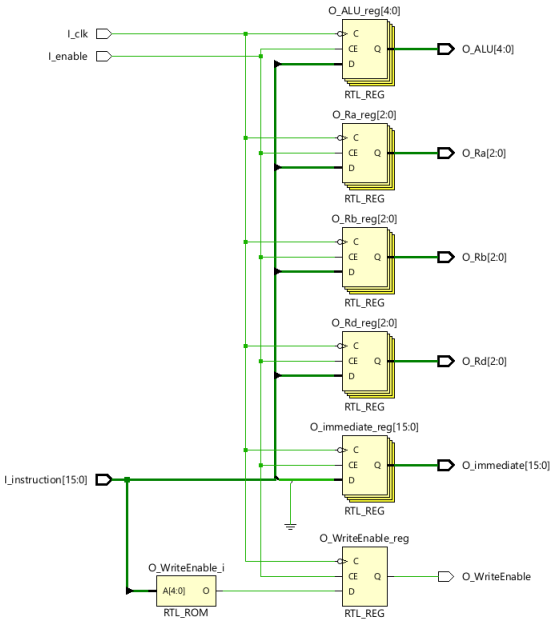


Figure 6: Elaborate Design of Instruction Decoder

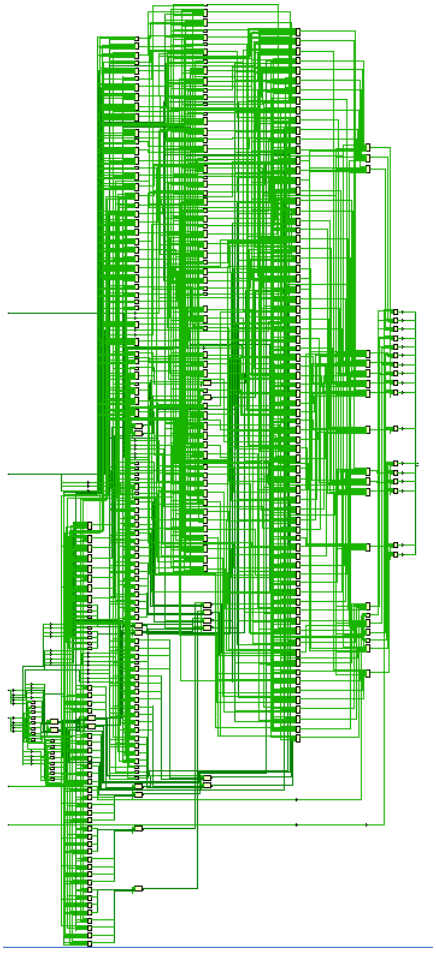


Figure 7: Elaborate Design of RISC Processor

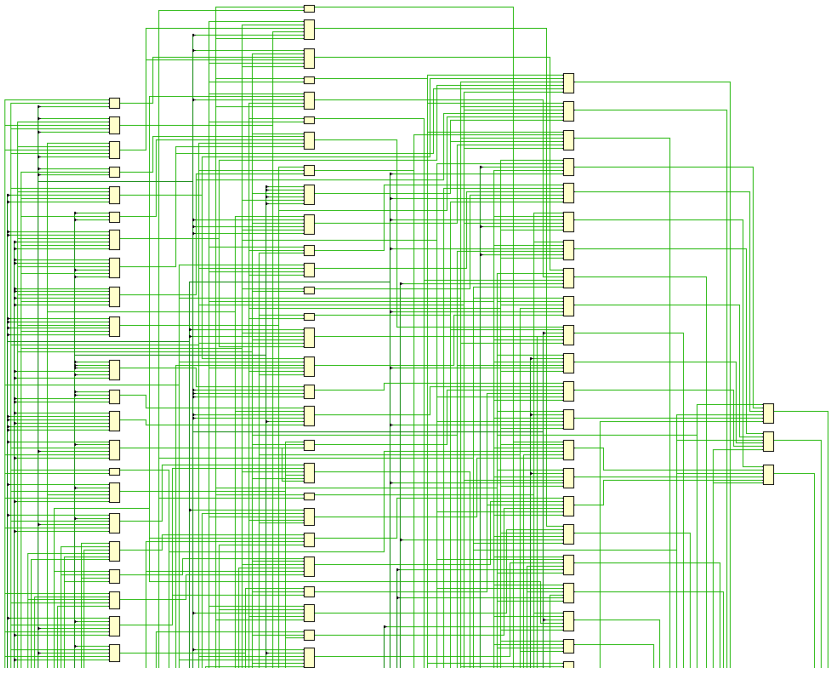


Figure 8: Zoomed part 1

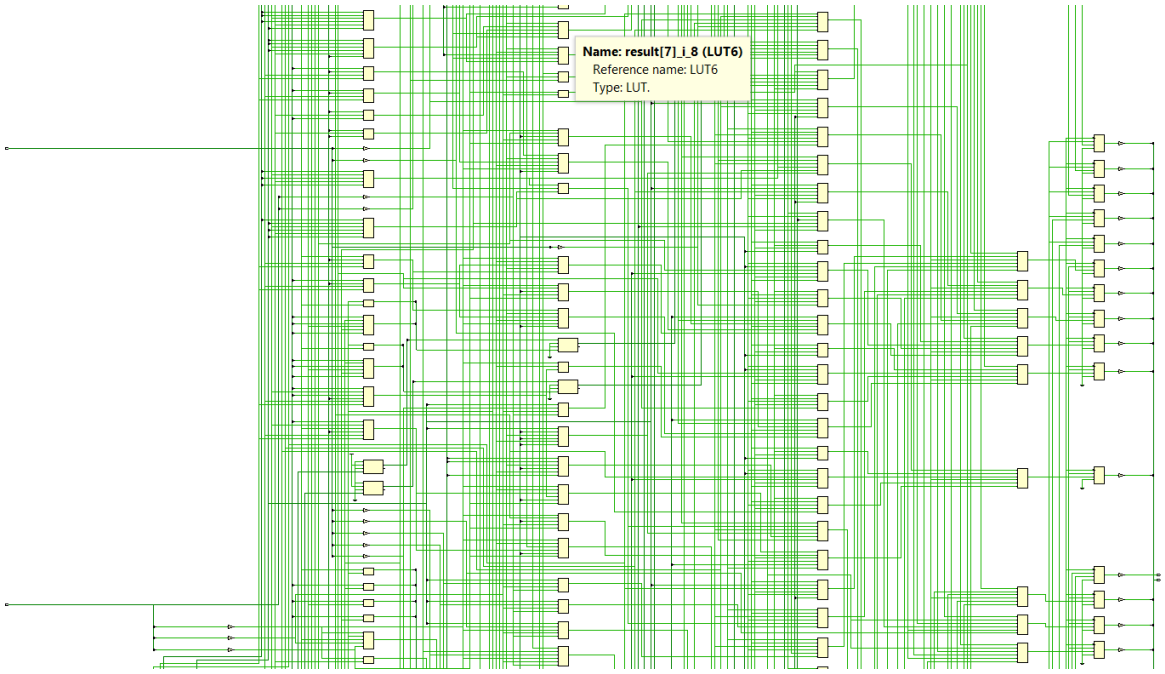


Figure 9: Zoomed Part 2

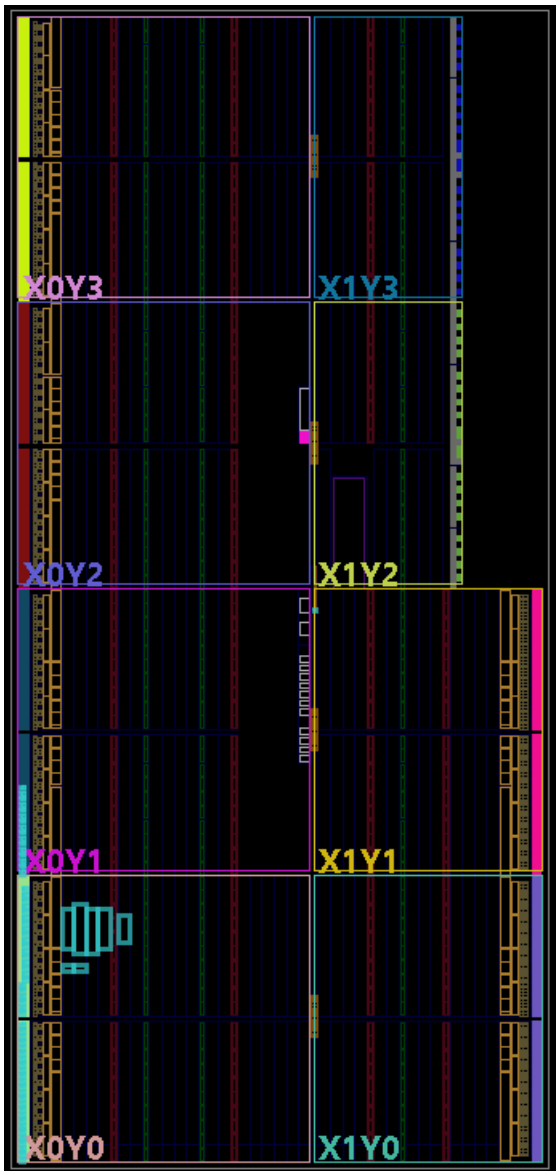


Figure 10: Schematic View of RISC Processor on FPGA