Submitted by; Submitted to;

**SHREENIDHI R S**  **1stop**

Government Engineering College Keep Growing

Hassan – 573201

Project

**16 BIT RISC PROCESSOR**

**ABSTRACT**

In this project, we design and implement a **16-bit RISC processor** using Verilog, targeting FPGA deployment in Vivado. The design is modular, comprising key components such as the **instruction decoder**, **control unit**, **arithmetic logic unit (ALU)**, **register handler**, **program counter**, and **RAM**. The **instruction decoder** decodes binary instructions into control signals, while the **control unit** orchestrates the processor's operations by generating signals to drive the ALU, memory, and register handler based on the decoded instruction. The **ALU** handles arithmetic and logical operations, essential for executing RISC instructions. The **register handler** controls reading from and writing to general-purpose registers, providing data to and receiving results from the ALU. The **program counter** keeps track of the instruction sequence, advancing to the next instruction or branching when necessary. The **RAM** provides storage for both data and instructions, making the system a complete computing unit.

For testing, separate **testbenches** are created for each module to validate their functionality in isolation. The **top-level testbench** integrates all modules, simulating real-world scenarios to ensure proper instruction fetching, decoding, execution, and memory handling. By leveraging Verilog’s capabilities in conjunction with Vivado, this project ensures efficient hardware synthesis and reliable simulation results for use in embedded systems. The modular approach simplifies debugging, testing, and future expansion, while maintaining performance and resource efficiency.

**INTRODUCTION**

This project focuses on the design and implementation of a **16-bit RISC processor** using Verilog, targeting efficient operation within FPGA environments using Vivado. The processor architecture is modular, consisting of several key components: the **instruction decoder**, **control unit**, **arithmetic logic unit (ALU)**, **register handler**, **program counter**, and **RAM**. Each module serves a distinct purpose; the instruction decoder translates binary instruction formats into control signals, while the control unit manages the overall operation of the processor by directing the flow of data and operations based on these signals. The ALU executes arithmetic and logical operations, the register handler manages the read and write operations of general-purpose registers, and the program counter keeps track of the instruction sequence for seamless execution. Additionally, RAM serves as the primary storage for instructions and data.

To ensure the functionality of the processor, comprehensive **testbenches** are developed for each module, facilitating rigorous verification of their individual operations. The **top-level testbench** integrates all components, simulating the entire processing workflow from instruction fetching to execution and memory handling. This structured approach not only simplifies debugging but also enhances the clarity of module interactions, paving the way for efficient synthesis and implementation in hardware. Overall, this project aims to create a reliable and performance-oriented 16-bit RISC processor, suitable for a variety of embedded applications.

**DESIGN**

timescale 1ns / 1ps

module inst\_dec(

input [15:0] I\_inst,

input I\_clk,

input I\_en,

output reg O\_aluop,

output reg [3:0] O\_selA,

output reg [3:0] O\_selB,

output reg [3:0] O\_selD,

output reg [15:0] imm

output reg O\_regwe

);

initial begin

O\_aulop <=0;

O\_selA <=0;

O\_selB <=0;

O\_selD <=0;

O\_imm <=0;

O\_regwe <=0;

end

always@(negedge I\_clk)begin

if (I\_en) begin

O\_aulop <= I\_inst[15:11];

O\_selA <= I\_inst[10:8];

O\_selB <= I\_inst[7:5];

O\_selD <= I\_inst[4:2];

O\_imm <= I\_inst[7:0];

Case ( I\_inst [15:12])

4'b0111 : O\_regwe <=0;

4'b1100 : O\_regwe <=0;

4'b1101 : O\_regwe <=0;

default : O\_regwe <=0;

endcase

end

end

endmodule

module control\_unit(

input [3:0] opcode,

output reg alu\_op,

output reg reg\_write,

output reg mem\_read,

output reg mem\_write

);

always @(\*) begin

case (opcode)

4'b0000: begin // ADD

alu\_op = 1;

reg\_write = 1;

mem\_read = 0;

mem\_write = 0;

end

4'b0001: begin // SUB

alu\_op = 1;

reg\_write = 1;

mem\_read = 0;

mem\_write = 0;

end

// Add additional opcodes here

default: begin

alu\_op = 0;

reg\_write = 0;

mem\_read = 0;

mem\_write = 0;

end

endcase

end

endmodule

module alu(

input [15:0] A,

input [15:0] B,

input alu\_op,

output reg [15:0] result

);

always @(\*) begin

if (alu\_op) begin

result = A + B; // Example for addition

end else begin

result = A - B; // Example for subtraction

end

end

endmodule

module register\_handler(

input clk,

input reg\_write,

input [3:0] reg\_addr,

input [15:0] reg\_data,

output reg [15:0] reg\_out

);

reg [15:0] registers [0:15]; // 16 registers

always @(posedge clk) begin

if (reg\_write) begin

registers[reg\_addr] <= reg\_data; // Write data to register

end

end

always @(\*) begin

reg\_out = registers[reg\_addr]; // Read data from register

end

endmodule

module program\_counter(

input clk,

input reset,

output reg [15:0] pc

);

always @(posedge clk or posedge reset) begin

if (reset) begin

pc <= 0; // Reset PC to 0

end else begin

pc <= pc + 1; // Increment PC

end

end

endmodule

module ram(

input clk,

input mem\_read,

input mem\_write,

input [15:0] address,

inout [15:0] data

);

reg [15:0] memory [0:255]; // 256 x 16-bit memory

always @(posedge clk) begin

if (mem\_write) begin

memory[address] <= data; // Write to memory

end

end

assign data = (mem\_read) ? memory[address] : 16'bz; // Read from memory

endmodule

**TEST BENCH**

module tb\_instruction\_decoder;

reg [15:0] instruction;

wire [3:0] opcode;

wire [11:0] operand;

instruction\_decoder id(

.instruction(instruction),

.opcode(opcode),

.operand(operand)

);

initial begin

instruction = 16'b0000000000000000; // Test case 1

#10;

instruction = 16'b0001000000000000; // Test case 2

#10;

instruction = 16'b0001000000000001; // Test case 3

#10;

$finish;

end

endmodule

module tb\_control\_unit;

reg [3:0] opcode;

wire alu\_op, reg\_write, mem\_read, mem\_write;

control\_unit cu(

.opcode(opcode),

.alu\_op(alu\_op),

.reg\_write(reg\_write),

.mem\_read(mem\_read),

.mem\_write(mem\_write)

);

initial begin

opcode = 4'b0000; // ADD

#10;

opcode = 4'b0001; // SUB

#10;

$finish;

end

endmodule

module tb\_alu;

reg [15:0] A, B;

reg alu\_op;

wire [15:0] result;

alu a(

.A(A),

.B(B),

.alu\_op(alu\_op),

.result(result)

);

initial begin

A = 16'h0005; B = 16'h0003; alu\_op = 1; // ADD

#10;

A = 16'h0005; B = 16'h0003; alu\_op = 0; // SUB

#10;

$finish;

end

endmodule

module tb\_register\_handler;

reg clk;

reg reg\_write;

reg [3:0] reg\_addr;

reg [15:0] reg\_data;

wire [15:0] reg\_out;

register\_handler rh(

.clk(clk),

.reg\_write(reg\_write),

.reg\_addr(reg\_addr),

.reg\_data(reg\_data),

.reg\_out(reg\_out)

);

initial begin

clk = 0;

reg\_write = 1; reg\_addr = 4'b0001; reg\_data = 16'h00FF; // Write to register 1

#10;

reg\_write = 0; reg\_addr = 4'b0001; // Read from register 1

#10;

$finish;

end

always #5 clk = ~clk; // Clock generation

endmodule

module tb\_program\_counter;

reg clk, reset;

wire [15:0] pc;

program\_counter pc(

.clk(clk),

.reset(reset),

.pc(pc)

);

initial begin

clk = 0;

reset = 1; #10; // Reset the PC

reset = 0; #10; // Release reset

#100;

$finish;

end

always #5 clk = ~clk; // Clock generation

endmodule

module tb\_ram;

reg clk;

reg mem\_read, mem\_write;

reg [15:0] address;

reg [15:0] data;

ram memory(

.clk(clk),

.mem\_read(mem\_read),

.mem\_write(mem\_write),

.address(address),

.data(data)

);

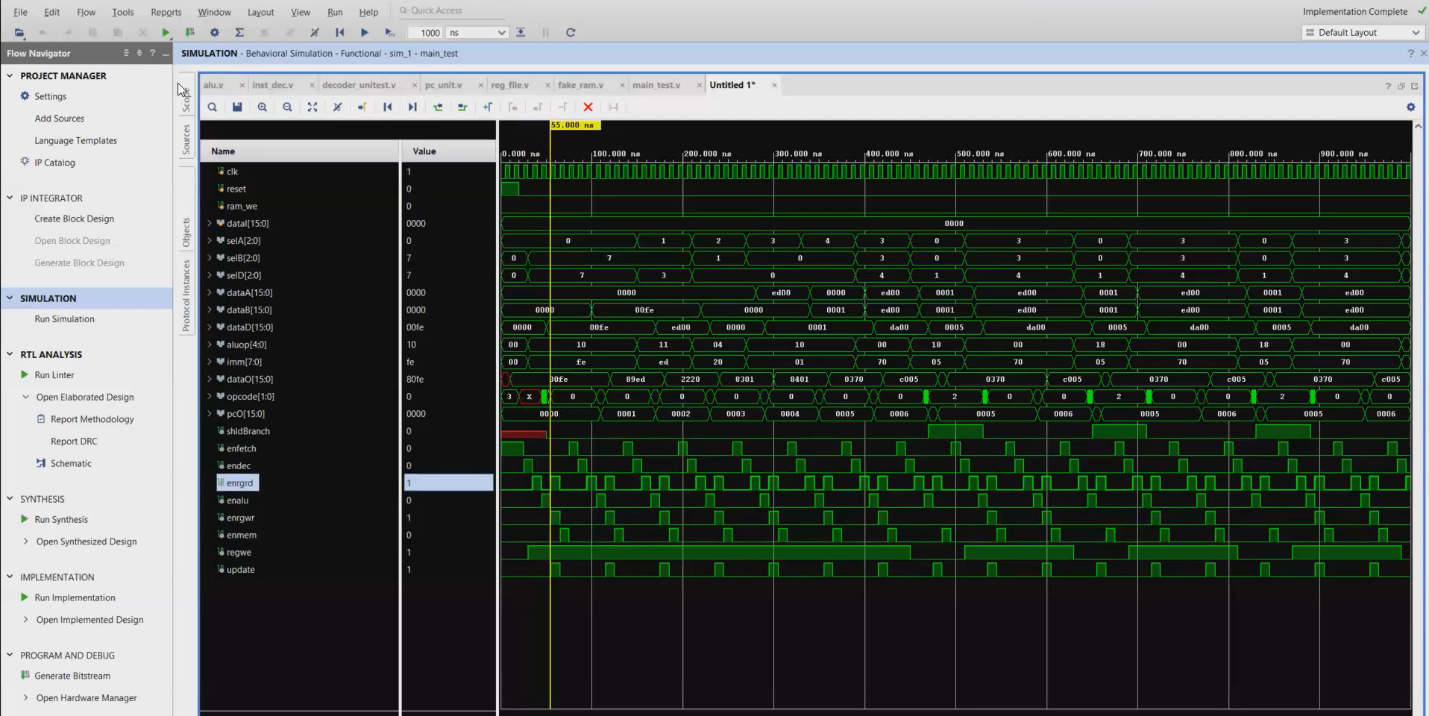
initial begin

clk = 0; mem\_write = 1; address = 16'h0000; data = 16'hA5A5; // Write to address 0

#10;

mem

**WAVEFORM**

****

**CONCLUSION:**

The implementation of a **16-bit RISC processor** in Verilog, targeting Vivado, utilizes a modular design approach where each core component, including the **instruction decoder**, **control unit**, **ALU**, **register handler**, **program counter**, and **RAM**, is developed independently to perform specific tasks within the processor architecture. The **instruction decoder** translates incoming instructions, while the **control unit** generates control signals to direct the ALU, registers, and memory. The **ALU** handles arithmetic and logical operations, and the **register handler** manages read and write operations within the register file. The **program counter** ensures sequential instruction execution, and the **RAM** provides data and instruction storage.

Each module is accompanied by dedicated **testbenches** for validation, ensuring correct functionality in isolation before integration. The **top-level testbench** simulates real-world conditions to validate the interaction between all components, allowing thorough testing of instruction fetching, decoding, execution, and memory operations. This modular and systematic approach simplifies verification and debugging while allowing for scalable and efficient processor design, which can be synthesized and implemented on an FPGA using Vivado, providing a versatile solution for embedded system applications.