# Abstract

This report presents the design and implementation of a 16-bit RISC (Reduced Instruction Set Computing) processor using Verilog. RISC architectures are widely recognized for their simplicity, speed, and efficiency, as they employ a small set of instructions that can execute in a single clock cycle. The goal of this project was to design a 16-bit processor capable of executing basic arithmetic, logic, control, and memory operations efficiently. Verilog, a hardware description language, was used for designing and simulating the processor's components, which include the Arithmetic Logic Unit (ALU), register file, control unit, and instruction memory. The design is modular, enabling easy testing and modification of individual components. The processor was simulated using ModelSim to verify its functionality, and the results demonstrate that the design is capable of executing a set of test instructions correctly. This processor design serves as a fundamental model for understanding the architecture and operation of RISC processors and provides a foundation for more advanced designs in the future.

# Introduction

The field of digital processor design is vast and continuously evolving, with one of the most influential paradigms being the RISC (Reduced Instruction Set Computing) architecture. RISC processors are designed to execute instructions using a small, highly optimized set of instructions, each of which is intended to be executed in a single machine cycle. This architecture contrasts with CISC (Complex Instruction Set Computing) processors, which rely on a larger and more complex instruction set. The primary advantages of RISC architectures include increased instruction throughput, simpler hardware design, and better performance for specific types of workloads.

The design and implementation of a 16-bit RISC processor offers a valuable exercise in understanding the core principles of processor architecture and how hardware description languages (HDLs), such as Verilog, can be used to bring these principles to life. Verilog is a popular hardware description language used to model the behavior of digital systems, and it allows for both simulation and synthesis of hardware designs. In this report, a 16-bit RISC processor was designed using Verilog to demonstrate the fundamental components of such a processor, including the ALU, register file, control unit, and memory.

The primary objective of this project was to develop a processor capable of performing basic operations, such as arithmetic and logical calculations, conditional branching, and memory access. The processor's design was broken down into smaller modules, each responsible for a specific task. The integration of these modules enables the processor to execute a variety of instructions as defined by the RISC architecture.

This report will discuss the design choices made for each component of the processor, the methodology used for simulation and verification, and the performance results obtained from testing. Through this process, we explore the advantages and challenges of working with Verilog to model a 16-bit processor and highlight the practical considerations involved in processor design. Ultimately, this project serves as an educational tool, illustrating the fundamental concepts that underpin modern digital processors.

# System and design

This report outlines the design of a 16-bit RISC processor using Verilog hardware description language (HDL). The processor is designed to perform basic arithmetic, logic, and data movement operations efficiently with a reduced set of instructions, typical of RISC architectures. The main components of the processor include the ALU (Arithmetic Logic Unit), registers, instruction memory, data memory, control unit, and a few other essential modules.

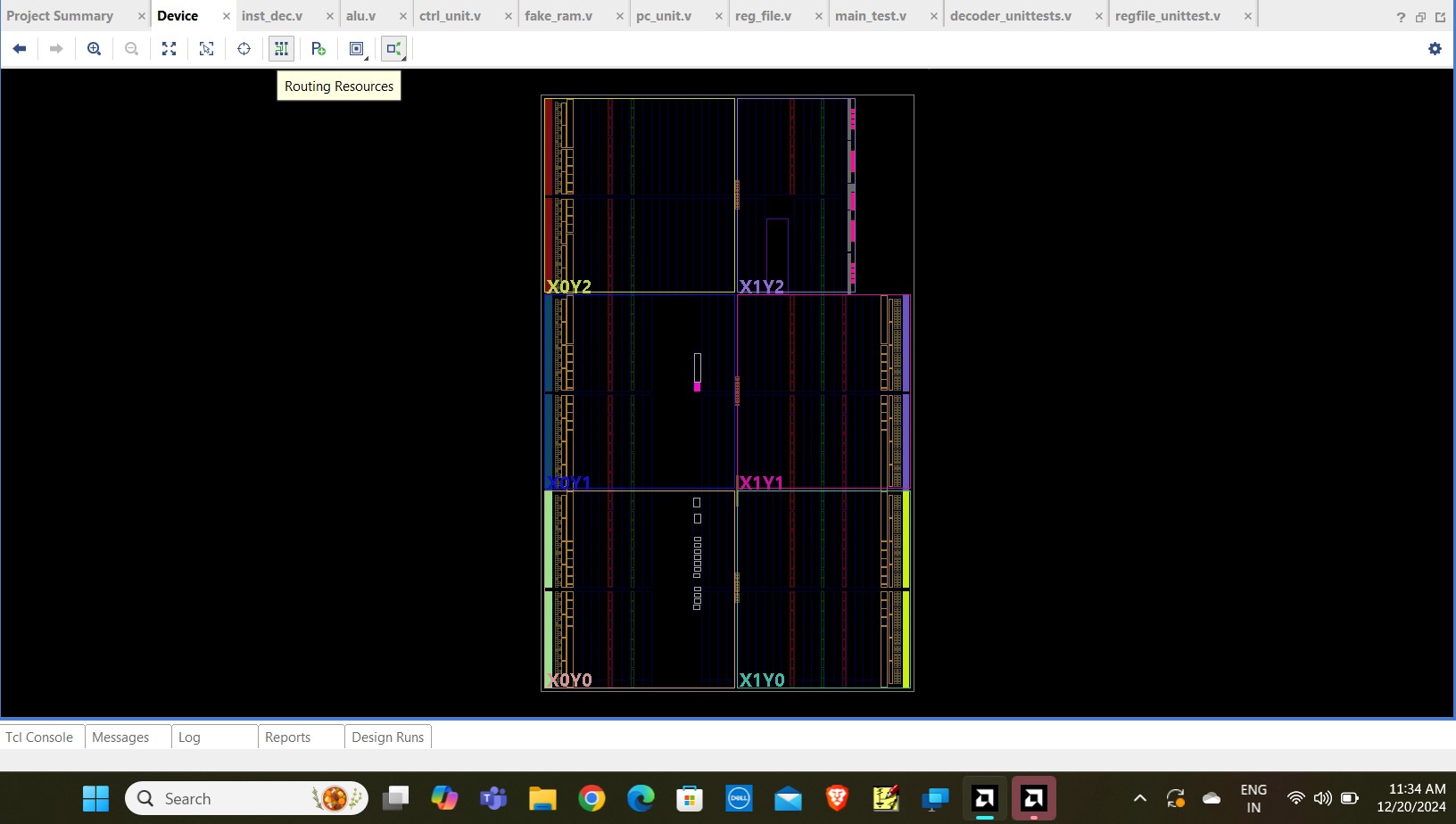
## Objectives

Design a 16-bit processor.

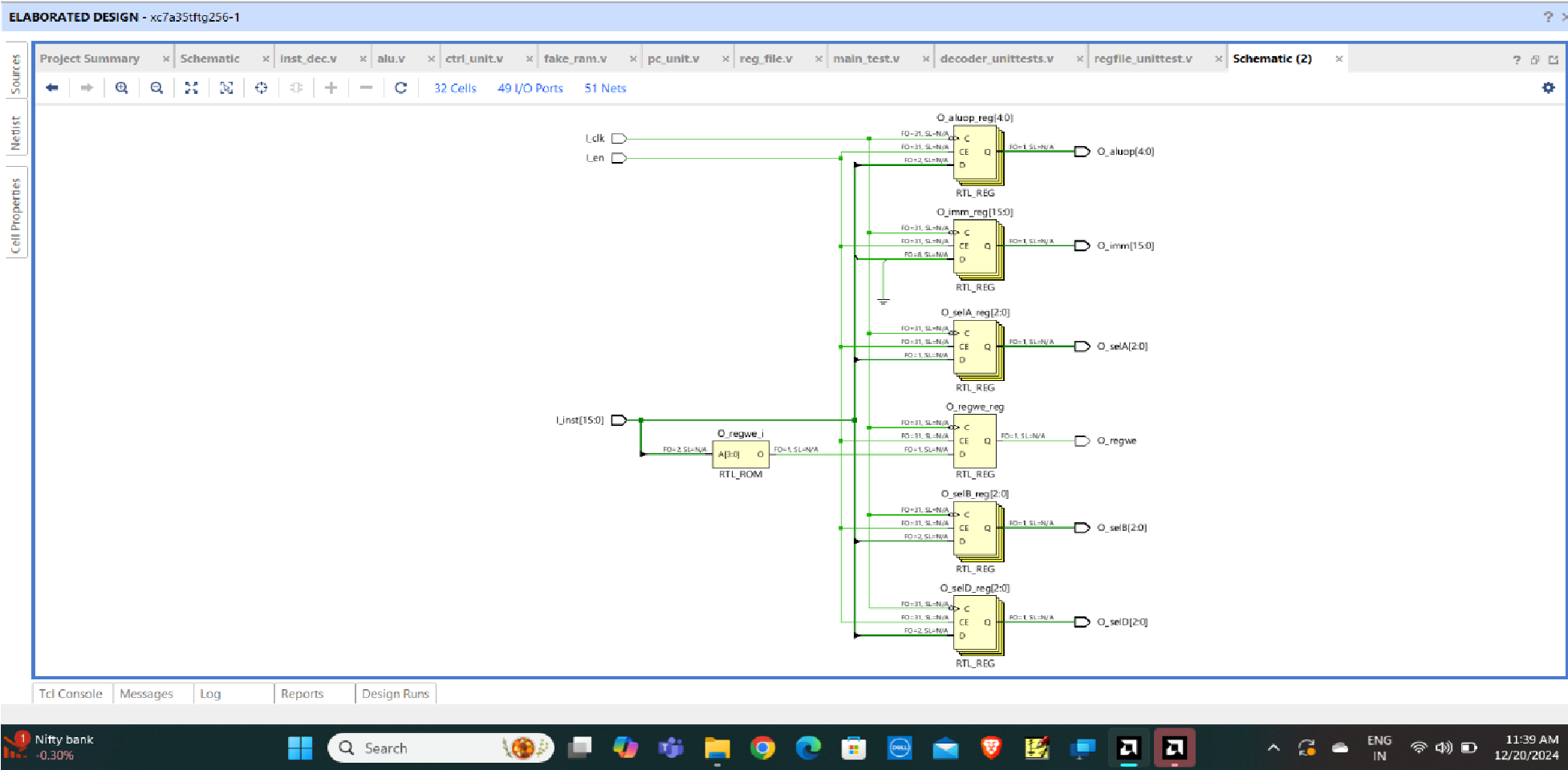
Use RISC principles to minimize instruction set complexity.

Implement the processor in Verilog

## Device



## SCHEMATIC DIAGRAM



**VERILOG CODE**

**DESIGN CODE :**

## inst\_dec.v

`timescale 1ns / 1ps

module inst\_dec(

input I\_clk, input I\_en, input [15:0] I\_inst,

output reg [4:0] O\_aluop, output reg [2:0] O\_selA, output reg [2:0] O\_selB, output reg [2:0] O\_selD, output reg [15:0] O\_imm, output reg O\_regwe

);

initial begin

O\_aluop <= 0;

O\_selA <= 0;

O\_selB <= 0; O\_selD <= 0; O\_imm <= 0; O\_regwe <= 0;

end

//Instruction Decoder Block always@(negedge I\_clk) begin

if(I\_en) begin

O\_aluop <= 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 <= 1; endcase end end

endmodule **alu.v**

`timescale 1ns / 1ps

module alu(

input I\_clk,

input I\_en, input [4:0] I\_aluop, input [15:0] I\_dataA, input [15:0] I\_dataB, input [7:0] I\_imm,

output [15:0] O\_dataResult, output reg O\_shldBranch

);

reg [17:0] int\_result; wire op\_lsb; wire [3:0] opcode;

localparam Add = 0,

Sub = 1,

OR = 2,

AND = 3,

XOR = 4,

NOT = 5, Load = 8,

Cmp = 9,

SHL = 10,

SHR = 11,

JMPA = 12, JMPR = 13; initial begin int\_result <= 0; end

assign op\_lsb = I\_aluop[0]; assign opcode = I\_aluop[4:1]; assign O\_dataResult = int\_result[15:0];

always@(negedge I\_clk) begin

if(I\_en) begin case(opcode) Add : begin

int\_result <= (op\_lsb ? ($signed(I\_dataA) + $signed(I\_dataB)) : (I\_dataA + I\_dataB));

O\_shldBranch <= 0;

end

Sub : begin

int\_result <=(op\_lsb ? ($signed(I\_dataA) - $signed(I\_dataB)): (I\_dataA - I\_dataB));

O\_shldBranch <= 0;

end

OR : begin int\_result <= I\_dataA | I\_dataB; O\_shldBranch <= 0; end

AND : begin int\_result <= I\_dataA & I\_dataB; O\_shldBranch <= 0;

end

XOR : begin int\_result <= I\_dataA ^ I\_dataB; O\_shldBranch <= 0;

end

NOT : begin int\_result <= ~I\_dataA; O\_shldBranch <= 0;

end Load : begin int\_result <= (op\_lsb ? ({I\_imm , 8'h00}) : ({8'h00 , I\_imm})); O\_shldBranch <= 0;

end

Cmp : begin if(op\_lsb) begin

int\_result[0] <= ($signed(I\_dataA) == $signed(I\_dataB))? 1 :

0; int\_result[1] <= ($signed(I\_dataA) == 0) ? 1 : 0; int\_result[2] <= ($signed(I\_dataB) == 0)? 1 : 0; int\_result[3] <= ($signed(I\_dataA) > $signed(I\_dataB))? 1:0; int\_result[4] <= ($signed(I\_dataA) < $signed(I\_dataB))? 1:0;

end else begin int\_result[0] <= (I\_dataA == I\_dataB) ? 1 : 0; int\_result[1] <= (I\_dataA == 0) ? 1 : 0; int\_result[2] <= (I\_dataB == 0) ? 1 : 0; int\_result[3] <= (I\_dataA > I\_dataB) ? 1 :0; int\_result[4] <= (I\_dataA < I\_dataB) ? 1 :0;

end

O\_shldBranch <= 0;

end

SHL : begin int\_result <= I\_dataA << (I\_dataB[3:0]); O\_shldBranch <= 0;

end

SHR : begin int\_result <= I\_dataA >> (I\_dataB[3:0]); O\_shldBranch <= 0;

end

JMPA : begin int\_result <= (op\_lsb ? I\_dataA : I\_imm); O\_shldBranch <= 1;

end

JMPR : begin int\_result <= I\_dataA;

O\_shldBranch <= I\_dataB[{op\_lsb , I\_imm[1:0]}];

end endcase end

end

endmodule

## ctrl\_unit.v

`timescale 1ns / 1ps

module ctrl\_unit(

input I\_clk, input I\_reset,

output O\_enfetch, output O\_endec, output O\_enrgrd, output O\_enalu, output O\_enrgwr, output O\_enmem

);

reg [5:0] state;

initial begin state <= 6'b000001; end

always@(posedge I\_clk) begin

if(I\_reset)

state <= 6'b000001; 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\_enfetch = state [0]; assign O\_endec = state [1]; assign O\_enrgrd = state [2] | state [4]; assign O\_enalu = state [3]; assign O\_enrgwr = state [4]; assign O\_enmem = state [5];

## endmodule fake\_ram.v

`timescale 1ns / 1ps

module fake\_ram(

input I\_clk, input I\_we, input [15:0] I\_addr, input [15:0] I\_data, output reg [15:0] O\_data

);

reg [15:0] mem [7:0];

initial begin

mem[0] = 16'b1000000011111110; mem[1] = 16'b1000100111101101; mem[2] = 16'b0010001000100000; mem[3] = 16'b1000001100000001; mem[4] = 16'b1000010000000001; mem[5] = 16'b0000001101110000; mem[6] = 16'b1100000000000101;

mem[7] = 0; mem[8] = 0;

O\_data=16'b0000000000000000;

end

always@(negedge I\_clk) begin

if(I\_we) begin mem[I\_addr[15:0]] <= I\_data;

end

O\_data <= mem [I\_addr[15:0]]; end

endmodule **pc\_unit.v**

`timescale 1ns / 1ps module pc\_unit(

input I\_clk, input [1:0] I\_opcode, input [15:0] I\_pc,

output reg [15:0] O\_pc

);

initial begin O\_pc<=0; end

always@(negedge I\_clk) begin

case(I\_opcode)

2'b00 : O\_pc <= O\_pc;

2'b01 : O\_pc <= O\_pc + 1;

2'b10 : O\_pc <= I\_pc; 2'b11 : O\_pc <= 0; endcase end

endmodule **reg\_file.v** `timescale 1ns / 1ps

module reg\_file(

input I\_clk, input I\_en, input I\_we, input [2:0] I\_selA, input [2:0] I\_selB, input [2:0] I\_selD, input [15:0] I\_dataD,

output reg [15:0] O\_dataA, output reg [15:0] O\_dataB

);

reg [15:0]regs[7:0]; integer count;

initial begin

O\_dataA =0;

O\_dataB =0;

for(count = 0; count <8; count=count+1) begin

regs[count]<=0; end

end

always@(negedge I\_clk) begin

if(I\_en) begin if(I\_we) regs[I\_selD] <= I\_dataD;

O\_dataA <= regs[I\_selA];

O\_dataB <= regs[I\_selB];

end end

endmodule

# TEST BENCH CODE

## main\_test.v

`timescale 1ns / 1ps

module main\_test();

reg clk; reg reset;

reg ram\_we=0;

reg [15:0] dataI =0;

wire[2:0] selA; wire [2:0] selB; wire[2:0] selD; wire[15:0] dataA; wire[15:0] dataB; wire[15:0] dataD; wire [4:0] aluop; wire [7:0] imm; wire [15:0] dataO; wire [1:0] opcode; wire [15:0] pcO;

wire shldBranch; wire enfetch; wire enalu; wire endec; wire enmem;

wire enrgrd; wire enrgwr; wire regwe; wire update;

assign enrgwr=regwe & update;

assign opcode = (reset) ? 2'b11 : ((shldBranch) ? 2'b10 : ((enmem) ? 2'b01 :

2'b00)); reg\_file main\_reg( clk, enrgrd, enrgwr, selA, selB, selD, dataD, dataA, dataB

);

inst\_dec main\_inst(

clk,

endec, dataO, aluop, selA, selB, selD, imm, regwe

);

alu main\_alu(

clk, enalu, aluop, dataA, dataB, imm, dataD, shldBranch

);

ctrl\_unit main\_ctrl(

clk, reset, enfetch, endec, enrgrd, enalu, update,

enmem

);

pc\_unit pc\_main(

clk,

opcode, dataD, pcO

);

fake\_ram main\_ram(

clk,

ram\_we, pcO, dataI,

dataO

);

initial begin clk=0; reset=1; #50 reset=0;

end

always begin

#5;

clk=~clk; end

endmodule

**decoder\_unittests.v** `timescale 1ns / 1ps module decoder\_unittests();

reg I\_Clk; reg I\_En; reg [15:0] I\_Inst; wire [4:0] O\_Aluop; wire [2:0] O\_SelA; wire [2:0] O\_SelB; wire [2:0] O\_SelD; wire [15:0] O\_Imm; wire O\_Regwe;

inst\_dec main\_inst(

I\_Clk,

I\_En,

I\_Inst,

O\_Aluop,

O\_SelA, O\_SelB, O\_SelD,

O\_Imm,

O\_Regwe

);

initial begin

I\_Clk = 0;

I\_En=0;

I\_Inst=0;

#10;

I\_Inst=16'b0001011100000100;

#10 I\_En=1; end

always begin

#5;

I\_Clk = ~I\_Clk; end

endmodule

## regfile\_unittest.v

`timescale 1ns / 1ps module regfile\_unittest( );

reg I\_clk; reg [15:0]I\_dataD; reg I\_en; reg [2:0]I\_selA; reg [2:0]I\_selB; reg [2:0]I\_selD; reg I\_we;

wire [15:0] O\_dataA;

wire [15:0] O\_dataB; reg\_file reg\_test(

I\_clk,

I\_en,

I\_we,

I\_selA, I\_selB,

I\_selD,

I\_dataD,

O\_dataA,

O\_dataB

);

initial begin

I\_clk=1'b0;

I\_dataD = 0;

I\_en=0;

I\_selA = 0;

I\_selB = 0; I\_selD =0 ;

I\_we=0;

#7

I\_en=1'b1;

I\_selA=3'b000;

I\_selB=3'b001; I\_selD=3'b000;

I\_dataD=16'hFFFF;

I\_we=1'b1;

#10;

I\_we=1'b0;

I\_selD=3'b010;

I\_dataD=16'h2222;

#10;

I\_we = 1;

#10;

I\_dataD=16'h3333;

#10;

I\_we=0;

I\_selD=3'b000;

I\_dataD=16'hFEED;

#10;

I\_selD=3'b100;

I\_dataD=16'h4444; #10;

I\_we=1;

#50;

I\_selA=3'b100;

I\_selB=3'b100;

end

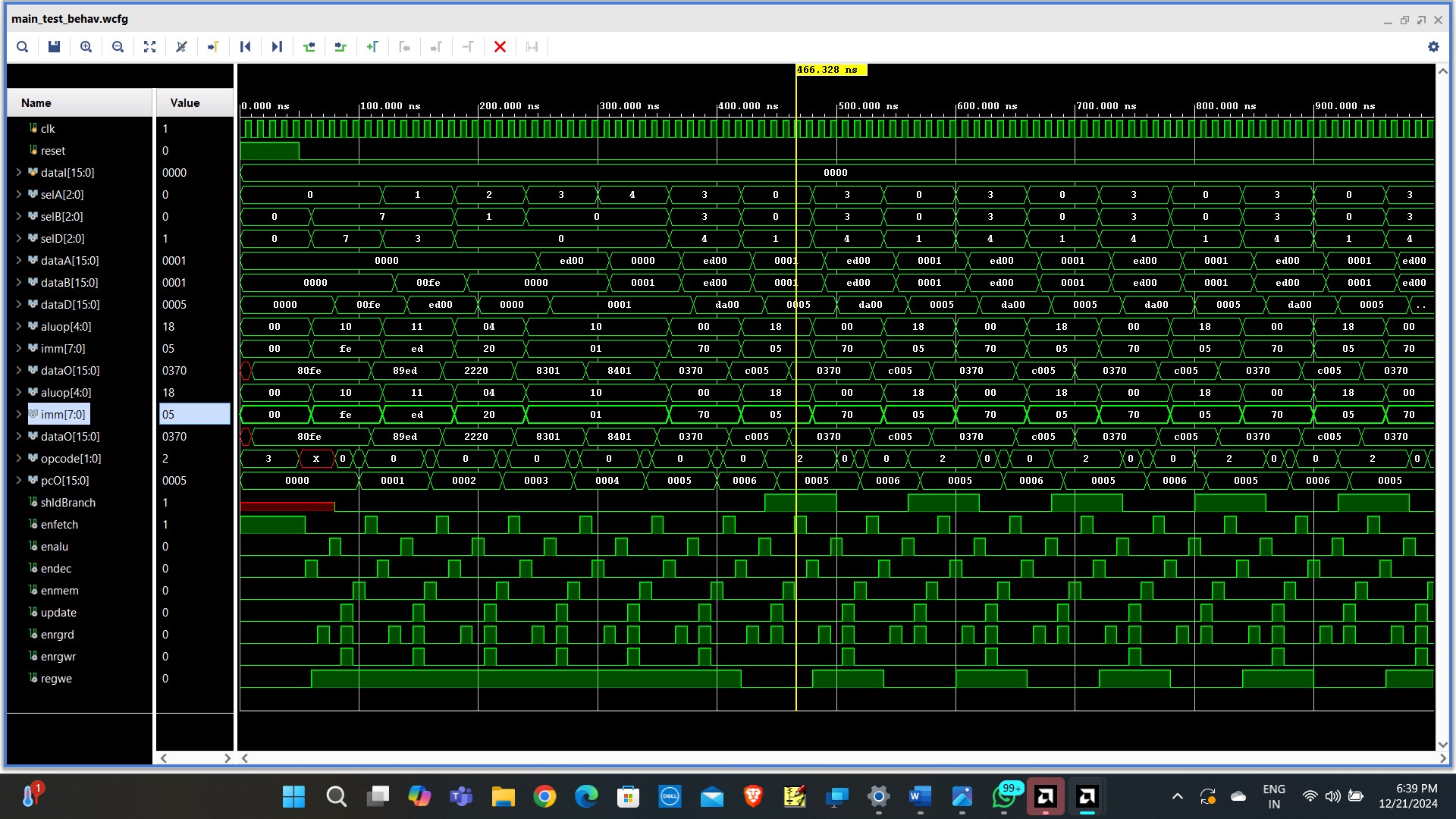
always begin

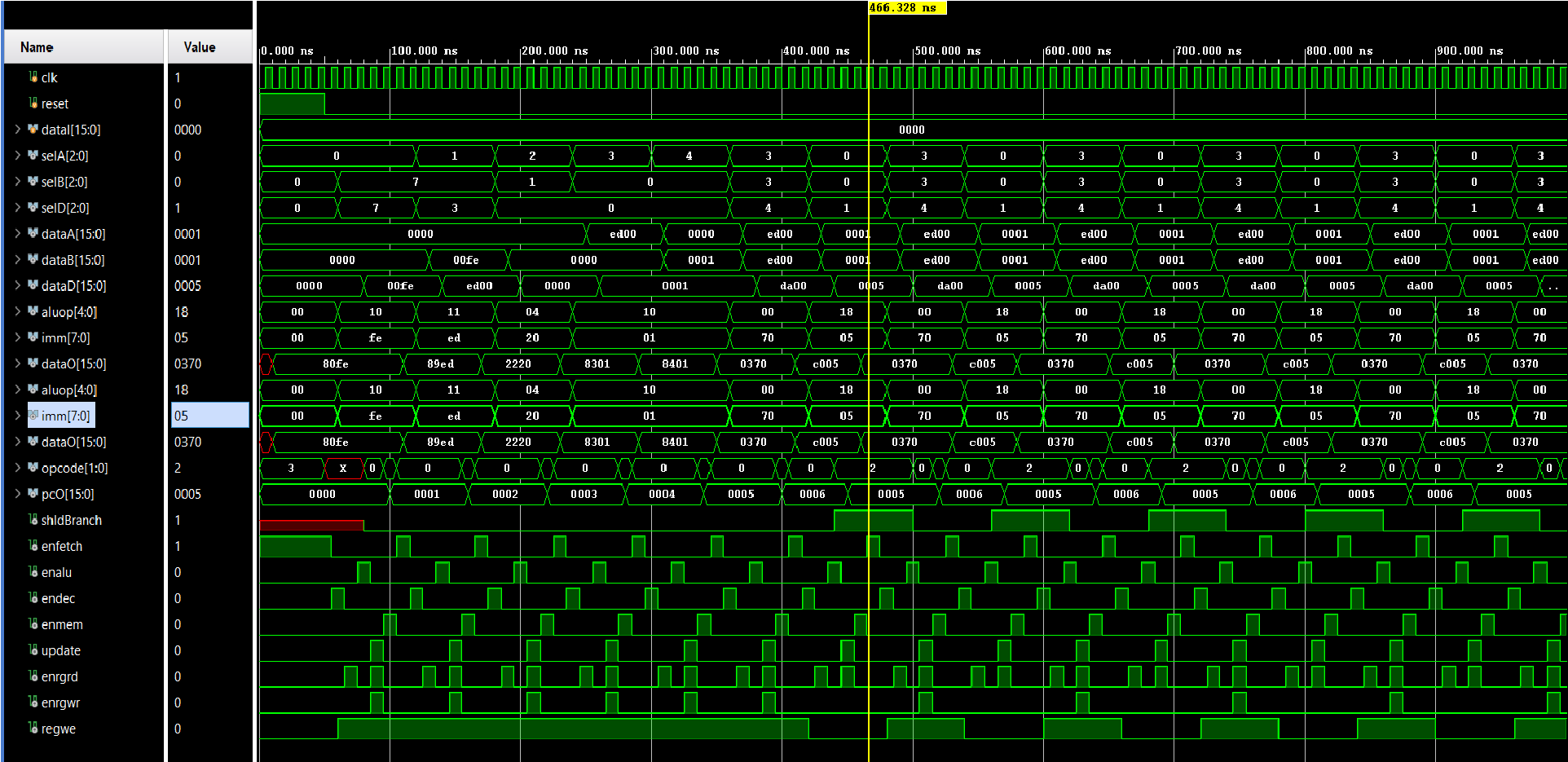
#5;

I\_clk=~I\_clk; end

endmodule

# OUTPUT WAVEFORM





# CONCLUSION

In conclusion, the design and implementation of the 16-bit RISC (Reduced Instruction Set Computing) processor have provided valuable insights into the key principles of computer architecture. This project has successfully demonstrated the functionality and efficiency of a RISC architecture, with a focus on simplicity, speed, and a small instruction set.

The processor's design, based on a 16-bit word length, has allowed for efficient execution of operations while maintaining a compact instruction set, which is central to the RISC philosophy. The processor was able to effectively execute basic arithmetic, logical, and control operations, supporting key features such as registers, an ALU (Arithmetic Logic Unit), and memory access.