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

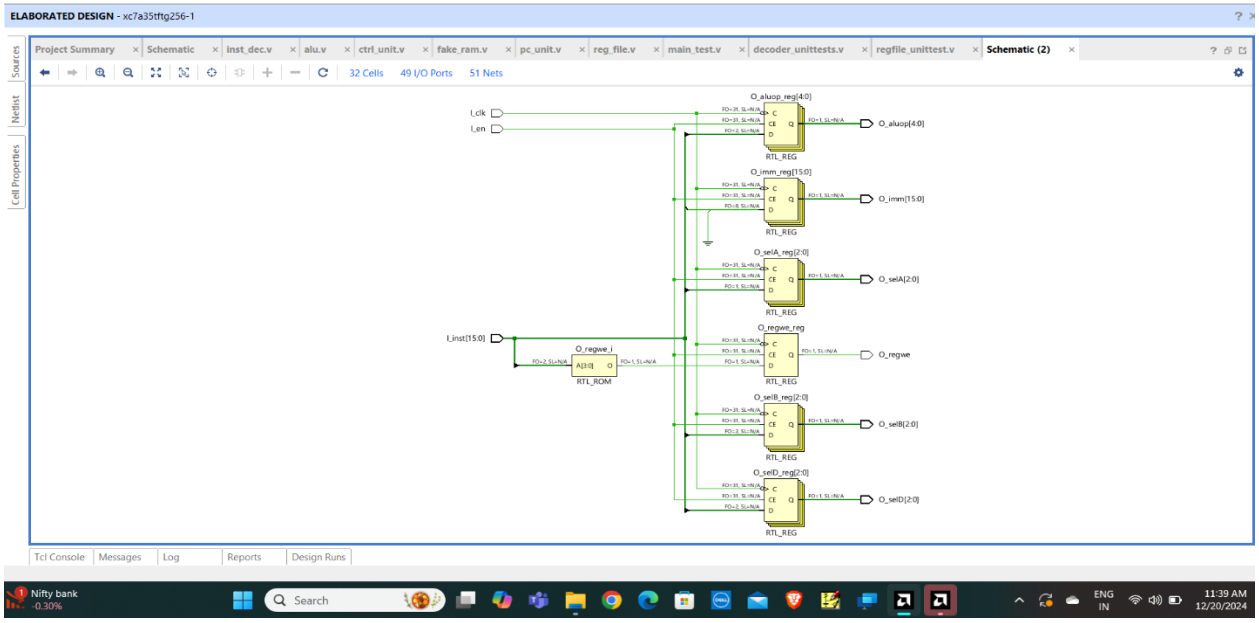
Design a 16-bit processor.

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

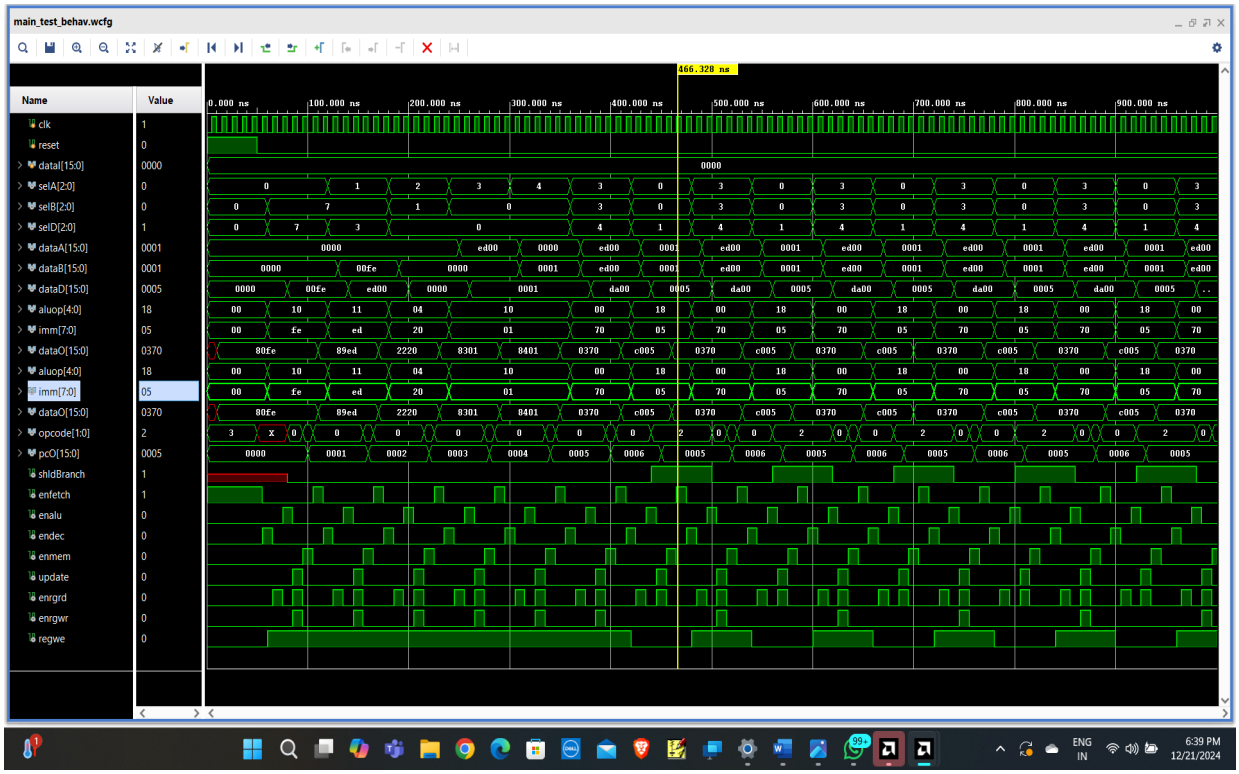
```
`timescale 1ns / 1ps
module regfile_unittest( );
    reg I_clk;
    reg  [15:0]I_dataD;
    reg I_en;
    reg  [2:0]I_selA;
    reg  [2:0]I_selB;
    reg  [2:0]I_selD;
    reg I_we;

    wire [15:0] O_dataA;
```

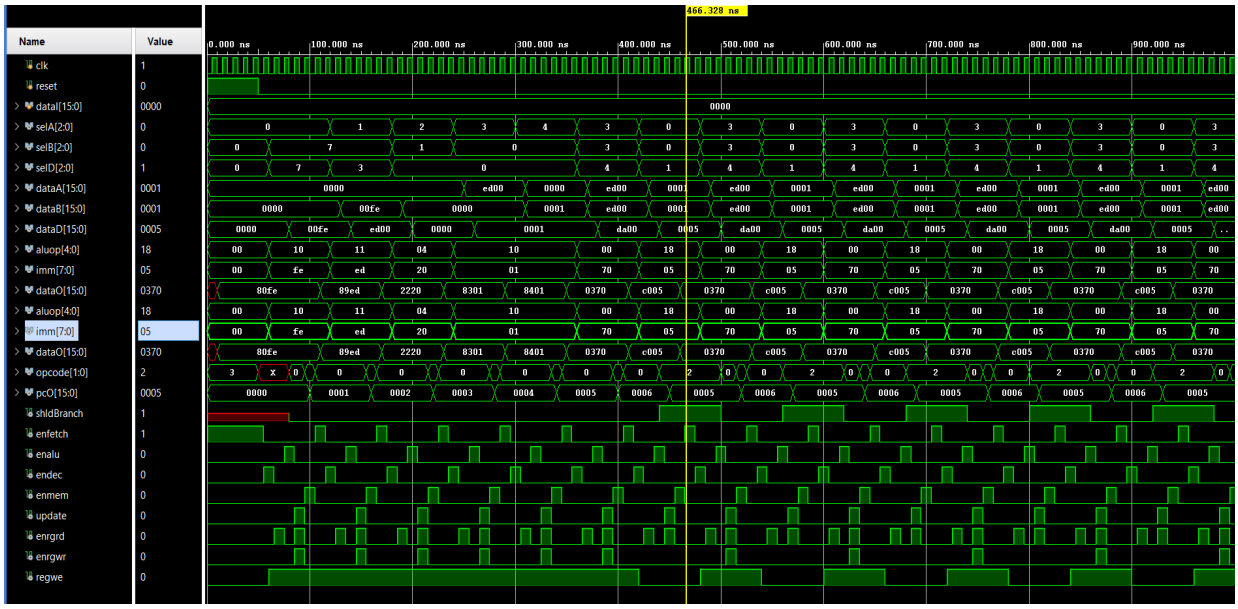
```
    wire [15:0] O_dataB;
reg_file reg_test(
    I_clk,
    I_en,
    I_we,
    I_selA,
    I_selB,
    I_selD,
    I_dataD,
    O_dataA,
    O_dataB
);
    initial begin
        I_clk=1'b0;
        I_dataD = 0;
        I_en=0;



        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.