16-Bit RISC Processor

Objective

To design, implement and verify the 16-bit RISC(Reduced Instruction Set Computer) using Verilog HDL on FPGA board and

1. Design Description

Design part of the 16-bit RISC processor includes instruction decoder, control unit, ALU(Arithmetic Logic Unit), registers handler, program counter and RAM(Random Access Memory).

Instruction Format

Arithmetic Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OPCODE					Rd			Ra			Rb			X	X

Immediate Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OPCODE (1000X)				Rd			Immediate (Imm)								

- OPCODE – Operational code
- Rd – Destination register to store the result.
- Ra – First source register (input A).
- Rb – Second source register (input B).
- Imm (Immediate) – A constant value embedded directly in the instruction.
- X – Don't-care condition.
- Arithmetic Instruction Format – Uses two registers as input operands.
- Immediate Instruction Format – Uses one register and one constant as operands.

1.1 Instruction Decoder

The instruction decoder decodes a 16-bit instruction and generate control signals for the ALU and registers handler. It extracts the ALU operation code, register addresses, and immediate values from specific instruction bits.

1.2 Control Unit

The Control Unit controls the processor by FSM(Finite State Machine) in sequence of fetching instructions, decoding them, reading registers, running the ALU, accessing memory, and writing back results to registers. It sends signals to make sure all parts work together at the right time.

1.3 ALU(Arithmetic Logic Unit)

The ALU performs arithmetic and logical operations as specified by the operational code, including addition, subtraction, bitwise operations(OR, AND, XOR, NOT), shifting of bits(left/right), loading of values into registers, jumps and comparisons. It produces the operation result and a branch control signal.

1.4 Registers Handler

The registers handler stores eight 16-bit registers. It supports reading two source registers simultaneously and writing to one destination register.

1.5 Program Counter

The program counter maintains the address of the next instruction. It increments sequentially, jumps to a new address, or resets based on operational code inputs, updating the counter.

1.6 RAM(Random Access Memory)

The ram acts as instruction and data memory with synchronous read and write operations. It stores 16-bit words indexing a 16-bit address and updates memory contents on write enable at the negative edge of the clock.

2. Verilog Codes

2.1 16-Bit RISC Processor Design Source Files

2.1.1 Instruction Decoder

```
// Timescale
‘timescale 1ns / 1ps

// Module Definition
module inst_dec (
    // Inputs
    input i_clk,
    input i_en,
    input [15:0] i_inst,
    // Outputs
    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 block to set default values
initial begin
    o_aluop <= 0;
    o_selA <= 0;
    o_selB <= 0;
    o_selD <= 0;
    o_imm <= 0;
    o_regwe <= 0;
end

// Instruction Decoding Logic
always @(negedge i_clk) begin
    if(i_en) begin
        o_aluop     <= i_inst[15:11];   // ALU operation code
        o_selD      <= i_inst[10:8];    // Select destination register
        o_selB      <= i_inst[7:5];     // Select B input
        o_selA      <= i_inst[4:2];     // Select A input
        o_imm       <= i_inst[7:0];     // Immediate value

        // Register Write Enable Logic
        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
```

Listing 1: inst_dec.v

2.1.2 Control Unit

```
// Timescale
`timescale 1ns / 1ps

// Module Definition
```

```verilog
module ctrl_unit (
    // Inputs
    input i_clk,
    input i_reset,
    // Outputs
    output o_enfetch,
    output o_endec,
    output o_enrgrd,
    output o_enalu,
    output o_enrgwr,
    output o_enmem
);

// Register Declarations
reg [5:0] state;

// Initial Block
initial begin
    state <= 6'b000001; // Initial state
end

// State Selection Logic
always @(posedge i_clk) begin
    if (i_reset)
        state <= 6'b000001; // Reset to initial state
    else begin
        case (state)
            6'b000001: state <= 6'b000010; // Fetch instruction
            6'b000010: state <= 6'b000100; // Decode instruction
            6'b000100: state <= 6'b001000; // Read registers
            6'b001000: state <= 6'b010000; // Execute ALU operation
            6'b010000: state <= 6'b100000; // Write back to register
            default  : state <= 6'b000001; // Default case to reset
        endcase
    end
end

// Assign Enable Signals
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
```

Listing 2: ctrl_unit.v

2.1.3 ALU(Arithmetic Logic Unit)

```
// Timescale
‘timescale 1ns / 1ps

// Module Definition
module alu (
    // Inputs
    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,
    // Outputs
    output [15:0] o_dataresult,
    output reg o_shldBranch
);

// Register/Wire Declaration
reg [17:0] int_result;
wire op_lsb;
wire [3:0] opcode;

// Parameter Declaration for ALU operations
localparam Add      = 0 ;
localparam Sub      = 1 ;
localparam OR       = 2 ;
localparam AND      = 3 ;
localparam XOR      = 4 ;
localparam NOT      = 5 ;
localparam Load     = 8 ;
localparam Cmp      = 9 ;
localparam SHL      = 10;
localparam SHR      = 11;
localparam JMPA     = 12;
localparam JMPR     = 13;

// Initial Block
initial begin
    int_result <= 0; // Initialize result to zero
end

// Assigning Values
assign op_lsb = i_aluop[0];
assign opcode = i_aluop[4:1];
assign o_dataresult = int_result[15:0];

// ALU Operations
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]; // Shift left operation
                o_shldBranch <= 0 ;
            end
            SHR      : begin
                int_result <= i_dataA >> i_dataB[3:0]; // Shift right
    operation
                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
```

Listing 3: alu.v

2.1.4 Registers Handler

```
// Timesacle
`timescale 1ns / 1ps

// Module Definition
module reg_file (
    // Inputs
    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,
    // Outputs
    output reg [15:0] o_dataA,
    output reg [15:0] o_dataB
);

// Internal Register Declaration
reg [15:0] regs [7:0];

// Loop Variable Declaration
integer count ;
```

```
// Initial Block to Initialize Registers
initial begin
    o_dataA = 0;
    o_dataB = 0;

    for (count = 0; count < 8; count = count + 1) begin
        regs[count] = 0;
    end
end

// Assigning Values to Outputs
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
```

Listing 4: reg_file.v

2.1.5 Program Counter

```
// Timescale
`timescale 1ns / 1ps

// Module Definition
module pc_unit (
    // Inputs
    input i_clk,
    input [1:0] i_opcode,
    input [15:0] i_pc,
    // Outputs
    output reg [15:0] o_pc
);

// Initial Block
initial begin
    o_pc <= 0; // Initialize program counter to zero
end

// Program Counter State
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
```
Listing 5: pc_unit.v

2.1.6 RAM(Random Access Memory)

```
// Timescale
`timescale 1ns / 1ps

// Module Definition
module fake_ram (
    // Inputs
    input i_clk,
    input i_we,
    input [15:0] i_addr,
    input [15:0] i_data,
    // Outputs
    output reg [15:0] o_data
);

// Memory Declaration
reg [15:0] mem [8:0];

// Initialize Memory
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

// RAM Operations
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
```
Listing 6: fake_ram.v

2.2 Testbench Files

2.2.1 Instruction Decoder Testbench

```
// Timescale
`timescale 1ns / 1ps

// Module Declaration
module decoder_unitests;

  // Variable Declarations
  // Registers
  reg I_Clk;
  reg I_En;
  reg [15:0] I_Inst;

  // Wires
  wire O_Regwe;
  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;

  // Instantiation
  inst_dec inst_unit(
    I_Clk,
    I_En,
    I_Inst,
    O_Aluop,
    O_SelA,
    O_SelB,
    O_SelD,
    O_Imm,
    O_Regwe
  );

  // Initial Block
  initial begin
    // Initialize
    I_Clk = 0;
    I_En = 0;
    I_Inst = 0;

    // After 10ns, apply instruction
    #10;
    I_Inst = 16'b0001011100000100;

    // After another 10ns, enable decoding
    #10;
    I_En = 1;
    #20 $stop;
  end

  // Clock Generation
  always #5 I_Clk = ~I_Clk;

endmodule
```

Listing 7: decoder_unitests.v

2.2.2 Registers Handler Testbench

```
// Timescale
‘timescale 1ns / 1ps

// Module Declaration
module regfile_unitests;

    // Registers (Inputs)
    reg I_Clk;
    reg [15:0] I_DataD;
    reg I_En;
    reg I_We;
    reg [2:0] I_SelA;
    reg [2:0] I_SelB;
    reg [2:0] I_SelD;

    // Wires (Outputs)
    wire [15:0] O_DataA;
    wire [15:0] O_DataB;

    // Instantiation
    reg_file reg_unit(
        I_Clk,
        I_En,
        I_We,
        I_SelA,
        I_SelB,
        I_SelD,
        I_DataD,
        O_DataA,
        O_DataB
    );

    // Initial Block
    initial begin
        // Reset All Inputs
        I_Clk = 1’b0;
        I_DataD = 0;
        I_En = 0;
        I_SelA = 0;
```

```verilog
        I_SelB = 0;
        I_SelD = 0;
        I_We = 0;

        // Start Test
        // Time = 7
        #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;

        // Time = 17
        #10;
        I_We = 1'b0;
        I_SelD = 3'b010;
        I_DataD = 16'h2222;

        // Time = 27
        #10;
        I_We = 1;

        // Time = 37
        #10;
        I_DataD = 16'h3333;

        // Time = 47
        #10;
        I_We = 0;
        I_SelD = 3'b000;
        I_DataD = 16'hFEED;

        // Time = 57
        #10;
        I_SelD = 3'b100;
        I_DataD = 16'h4444;

        // Time = 67
        #10;
        I_We = 1;

        // Time = 117
        #50;
        I_SelA = 3'b100;
        I_SelB = 3'b100;

        // Time = 137
        #20;
        $finish;
```

```
    end

    // Clock Generation
    always begin
        #5;
        I_Clk = ~I_Clk;
    end

endmodule
```

Listing 8: regfile_unitests.v

2.2.2 Main Testbench

```
// Timescale
`timescale 1ns / 1ps

// Module Declaration
module main_test;

// Variable Declarations
// Registers
reg clk;
reg reset;
reg ram_we = 0;
reg [15:0] dataI = 0;
// Wires
wire [2:0] selA, selB, selD;
wire [15:0] dataA, dataB, dataD;
wire [4:0] aluop;
wire [7:0] imm;
wire [15:0] dataO;
wire [1:0] opcode;
wire shldBranch;
wire enfetch, endec, enmem, enalu, enrgrd, enrgwr;
wire [15:0] pcO;
wire regwe;
wire update;

// Assignments
assign enrgwr = regwe & update;
assign opcode = (reset) ? 2'b11 : ((shldBranch) ? 2'b10 : ((enmem) ? 2'b01 :
    2'b00));

// Instantiate Register File
reg_file main_reg (
    clk,
    enrgrd,
    enrgwr,
    selA,
    selB,
```

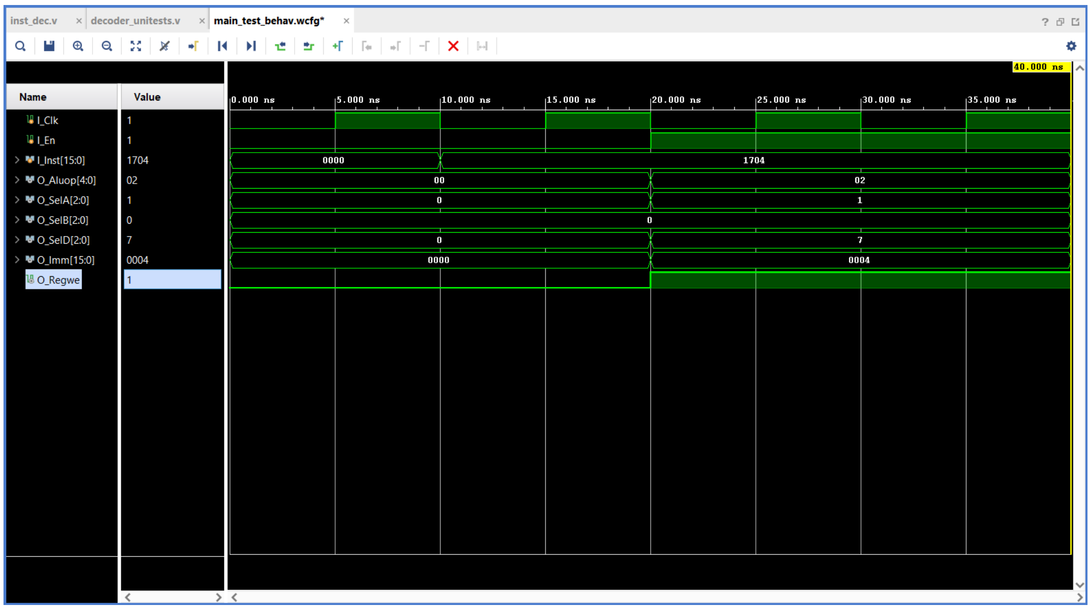
```verilog
    selD,
    dataD,
    dataA,
    dataB
);

// Instantiate Instruction Decoder
inst_dec main_inst (
    clk,
    endec,
    dataO,
    aluop,
    selA,
    selB,
    selD,
    imm,
    regwe
);

// Instantiate ALU
alu main_alu (
    clk,
    enalu,
    aluop,
    dataA,
    dataB,
    imm,
    dataD,
    shldBranch
);

// Instantiate Control Unit
ctrl_unit main_ctrl (
    clk,
    reset,
    enfetch,
    endec,
    enrgrd,
    enalu,
    update,
    enmem
);

// Instantiate PC Unit
pc_unit main_pc (
    clk,
    opcode,
    dataD,
    pcO
);
```

```
// Instantiate RAM (Fake RAM)
fake_ram main_ram (
    clk,
    ram_we,
    pcO,
    dataI,
    dataO
);

// Initial Block
initial begin
    clk = 0;
    reset = 1;
    #20;
    reset = 0;
end

// Clock Generation
always begin
    #5 clk = ~clk;
end

endmodule
```

Listing 9: main_test.v

3. Outputs

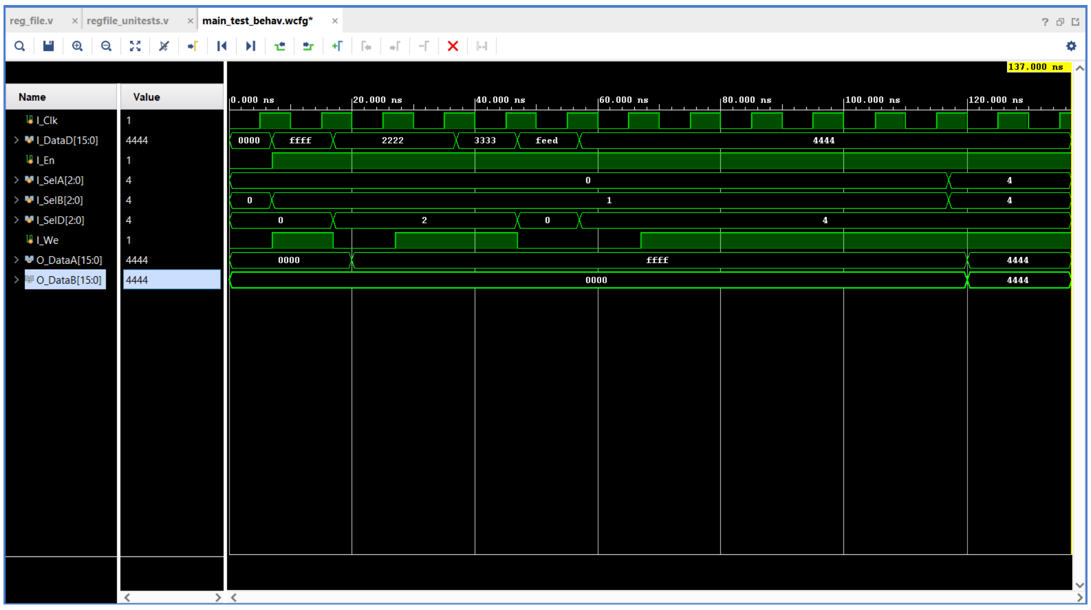
3.1 Instruction Decoder Output



Name	Value
I_Clk	1
I_En	1
I_Inst[15:0]	1704
O_Aluop[4:0]	02
O_SelA[2:0]	1
O_SelB[2:0]	0
O_SelD[2:0]	7
O_Imm[15:0]	0004
O_Regwe	1

Figure 1: Waveform showing output of the instruction decoder for the given testbench.

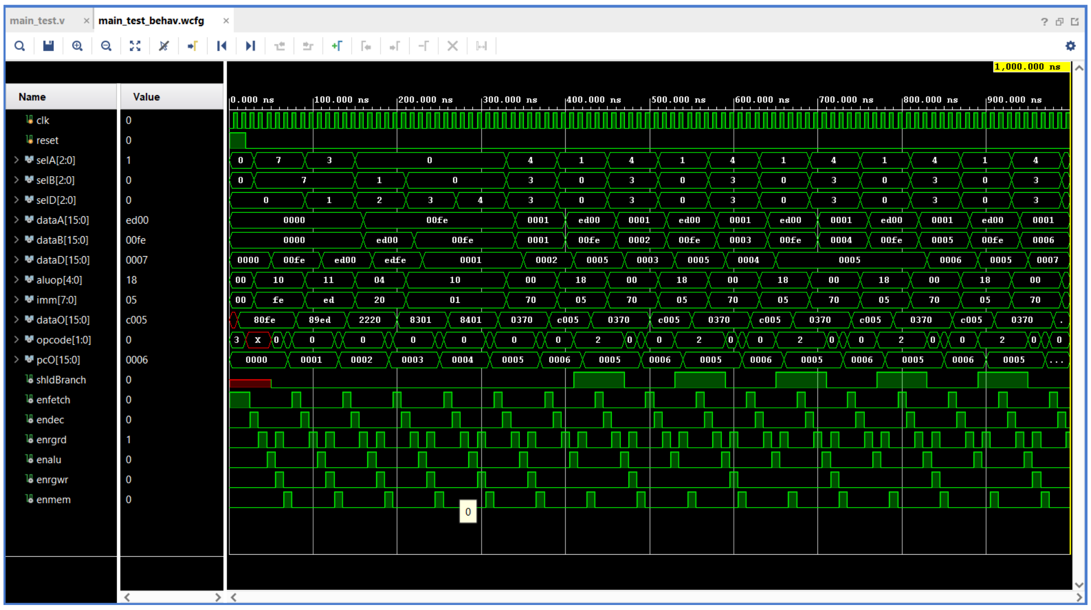
3.2 Registers Handler Output



Name	Value
I_Clk	1
I_DataD[15:0]	4444
I_En	1
I_SelA[2:0]	4
I_SelB[2:0]	4
I_SelD[2:0]	4
I_We	1
O_DataA[15:0]	4444
O_DataB[15:0]	4444

Figure 2: Waveform showing output of the registers handler for the given testbench.

3.3 Main Output



Name	Value
clk	0
reset	0
selA[2:0]	1
selB[2:0]	0
selD[2:0]	0
dataA[15:0]	ed00
dataB[15:0]	00fe
dataD[15:0]	0007
aluop[4:0]	18
imm[7:0]	05
dataO[15:0]	c005
opcode[1:0]	0
pcO[15:0]	0006
shldBranch	0
enfetch	0
endec	0
enrgrd	1
enalu	0
enrgwr	0
enmem	0

Figure 3: Waveform showing output of the whole process for the given main testbench.