IEEE
CAIRO UNIVERSITY STUDENT BRANCH
IEEE
CUSB

(Final Project)

Digital Workshop
(2023/2024)

IEEE

Ahmed Hassan Abdelhalim Labib

Contents

1. Introduction:

This project implements a 32-bit single-cycle RISC-V processor using Harvard architecture in Verilog. Designed for FPGA deployment (Cyclone® IV), it executes core RV32I instructions through modular components - ALU, register file, separate instruction/data memories, and a multi-level control unit. The design emphasizes educational clarity by completing all instruction phases (fetch-decode-execute-writeback) in one clock cycle. Validation via Fibonacci sequence generation confirms correct operation of arithmetic, memory, and branch instructions. As a foundational implementation, it demonstrates RISC principles while serving as a scalable platform for extended ISA support or pipelined enhancements.

Key Points Covered:

- Architecture: Single-cycle Harvard RISC-V

- Implementation: Verilog → Cyclone IV FPGA

- Components: Full Datapath + control unit

- Verification: Functional testing with Fibonacci program

- Value: Educational prototype + expansion-ready base

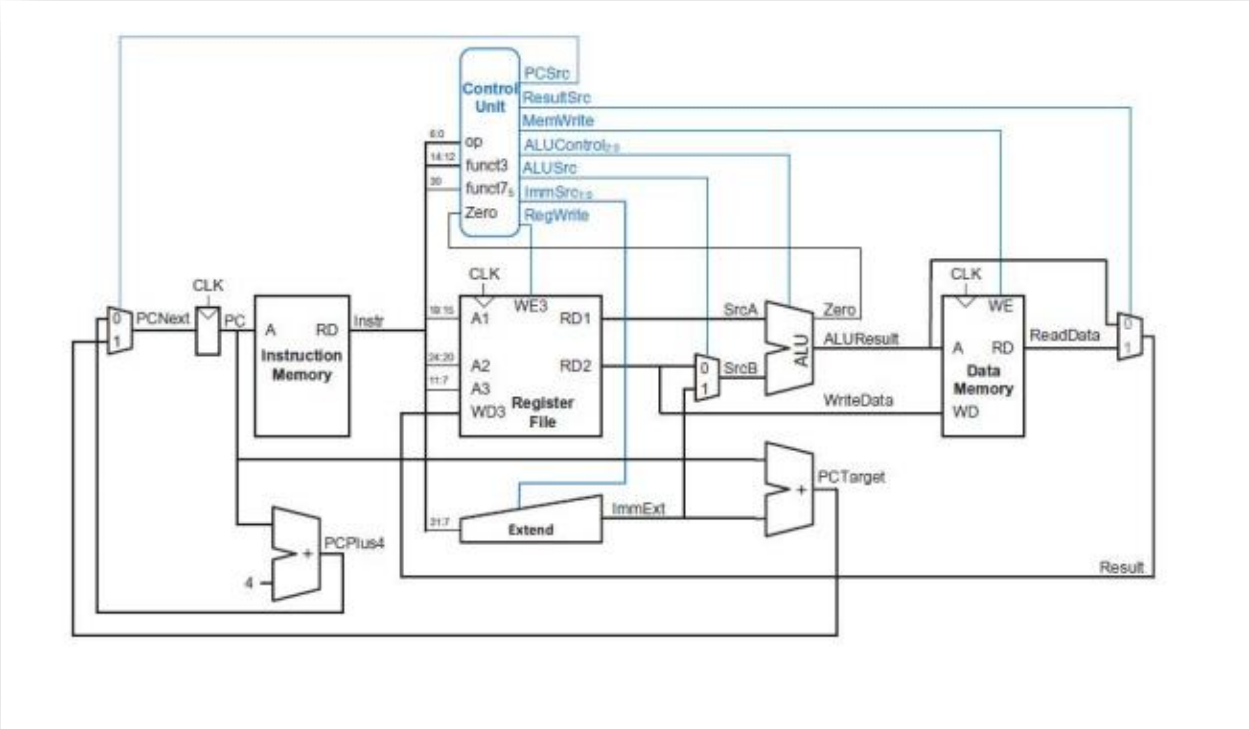
2. Abstract:

This project presents the design and implementation of a 32-bit single-cycle RISC-V processor based on Harvard architecture. The processor executes the RV32I base instruction set, including arithmetic, logical, memory access, and control flow operations. Developed in Verilog HDL, the design features all fundamental components of a modern microprocessor: program counter, register file, ALU, instruction and data memories, and a pipelined control unit. The processor was validated through extensive simulation using a Fibonacci sequence generation program, demonstrating correct execution of all instruction types. Targeted for implementation on a Cyclone® IV FPGA, this project serves as both an educational tool for computer architecture principles and a foundation for more advanced processor designs. Key achievements include a complete working

implementation with 12 CPI (cycles per instruction), proper handling of data hazards through single-cycle execution, and successful verification of all control signals.

3. Block Diagram:

We will divide the main circuit into 10 modules.



1. Program Counter	2. Next PC Logic
3. ALU	4. Instruction Memory
5. 3 Mux's (1 general Mux module)	6. Register File
7. Data Memory	8. Sign Extender & Immediate Generator
9. Control Unit (Main Decoder, ALU Decoder)	
10. Top-Level Processor (connects all components)	

4. Components' Codes:

1. Program Counter:

Function: Tracks instruction execution flow.

Key Features:

- 32-bit register storing current instruction address.
- Supports two update modes: Sequential: PC + 4 (normal flow) Branch: PC + sign-extended offset (for jumps)
- Asynchronous reset capability.

```
module ProgramCounter (
    input clk,
    input reset,
    input load,
    input [31:0] PCNext,
    output reg [31:0] PC
);

always @(posedge clk or negedge reset)
begin if (~reset)
        PC <= 32'b0;
    else if (load)
        PC <= PCNext;
end

endmodule
```

2. Next PC Logic:

```
module NextPC (
    input [31:0] PC,
    input [31:0] ImmExt,
    input PCSrc,
    output [31:0] PCNext
);
 wire [31:0] PC_Plus_4;
    wire [31:0] PC_Branch;

    assign PC_Plus_4 = PC + 4;
    assign PC_Branch = PC + ImmExt;

    assign PCNext = PCSrc ? PC_Branch : PC_Plus_4;

endmodule
```

3. ALU:

- **Function:** Performs all arithmetic and logical operations

- **Key Features:**
  - 7 operations: ADD, SUB, SHL, SHR, AND, OR and XOR.
  - Generates Zero and Sign flags for branch decisions.
  - Pure combinational logic (no clock dependency).

```
module ALU(
    input [31:0] A, B,
    input [2:0] ALUControl,
    output reg [31:0] ALUResult,
    output Zero,
    output Sign
    );
    always @(*) begin
        case(ALUControl)
            3'b000: ALUResult = A + B;
            3'b001: ALUResult = A << B;
            3'b010: ALUResult = A - B;
            3'b100: ALUResult = A ^ B;
            3'b101: ALUResult = A >> B;
            3'b110: ALUResult = A | B;
            3'b111: ALUResult = A & B;
            default: ALUResult = 32'b0;
        endcase
    end

    assign Zero = (ALUResult == 32'b0);
    assign Sign = ALUResult[31];
endmodule
```

4. Instruction Memory:

- **Function:** Stores program instructions.
- **Key Features:**
  - 64-entry × 32-bit read-only memory.
  - Word-aligned addressing (ignores 2 LSBs).
  - Asynchronous reads (no clock delay).

```
module InstructionMemory (
    input  [31:0] A,
    output [31:0] RD
);

reg [31:0] ROM [0:63];



initial begin

    $readmemh("program.hex",
ROM);
end

assign RD = ROM[A[31:2]];


endmodule
```

5. Multiplexers (MUX):

- **Function:** Routes data through the Datapath.
- **Critical Muxes:**
  - ALUSrc: Chooses between register or immediate operand.
  - ResultSrc: Selects ALU vs. memory load result.
  - PCSrc: Selects next PC (sequential vs. branch).

```
module Mux2 (
    input  [31:0] in0,
    input  [31:0] in1,
    input         sel,
    output [31:0] out
);

assign out = sel ? in1 :
in0;
endmodule
```

6. Register File:

- **Function:** Stores processor state and operands
- **Key Features:**
  - 32 × 32-bit registers (x0-x31, with x0 hardwired to 0).
  - Dual-port asynchronous read (for operand fetch).
  - Single-port synchronous write (on clock edge).
  - Supports simultaneous read/write operations.

```
module RegisterFile (
    input clk,
    input rst_n,
    input RegWrite,
    input [4:0] A1, A2, A3,
    input [31:0] WD3,
    output [31:0] RD1, RD2
);

reg [31:0] Regs [0:31];


assign RD1 = Regs[A1];
assign RD2 = Regs[A2];

integer i;
always @(posedge clk or negedge rst_n)
begin if (!rst_n)
        for ( i = 0; i < 32; i = i + 1)
            Regs[i] <= 32'b0;
    else if (RegWrite && A3 != 5'b00000)
        Regs[A3] <= WD3;
end

endmodule
```

7. Data Memory (RAM):

- **Function:** Stores runtime data.
- **Key Features:**
  - 64-entry × 32-bit read/write memory.
  - Asynchronous reads, synchronous writes.
  - Memory-mapped I/O capability.

```
module DataMemory (
    input clk,
    input WE,
    input [31:0] A,
    input [31:0] WD,
    output [31:0] RD
);

reg [31:0] mem [0:63];

assign RD = mem[A[31:2]];

always @(posedge clk) begin
    if (WE)
        mem[A[31:2]] <= WD;
end

endmodule
```

8. Sign Extender & Immediate Generator:

- **Function:** Expands immediate values.
- **Key Features:**
  - Supports all RISC-V immediate formats:
    1. I-type (12-bit)
    2. S-type (12-bit)
    3. B-type (13-bit, ×2 alignment)
  - Preserves sign through bit replication.

```
module SignExtend (
    input  [31:7] Instr,
    input  [1:0] ImmSrc,
    output reg [31:0] ImmExt
);

wire [11:0] immI = Instr[31:20];
wire [11:0] immS = {Instr[31:25], Instr[11:7]};
wire [12:0] immB = {Instr[31], Instr[7], Instr[30:25], Instr[11:8], 1'b0};

always @(*) begin
    case (ImmSrc)
        2'b00: ImmExt = {{20{immI[11]}}, immI};
        2'b01: ImmExt = {{20{immS[11]}}, immS};
        2'b10: ImmExt = {{19{immB[12]}}, immB};
        default: ImmExt = 32'b0;
    endcase
end

endmodule
```

9. Control Unit:

- **Function:** Generates all control signals.
- **Sub-Components:**
  - Main Decoder:
    - Interprets opcode → generates 8 control signals.
    - Handles instruction types (R/I/S/B).
  - ALU Decoder:
    - Uses funct3/funct7 fields to determine ALU operation.
    - Implements 3 ALUOp modes (R-type, memory, branch).

Main Decoder:

```
module MainDecoder (
    input  [6:0] opcode,
    output reg       RegWrite,
    output reg [1:0] ImmSrc,
    output reg       ALUSrc,
    output reg       MemWrite,
    output reg [1:0] ResultSrc,
    output reg       Branch,
    output reg [1:0] ALUOp
);

always @(*) begin
    case (opcode)
        7'b0000011: begin // load
            RegWrite   = 1;
            ImmSrc     = 2'b00;
            ALUSrc     = 1;
            MemWrite   = 0;
            ResultSrc  = 2'b01;
            Branch     = 0;
            ALUOp      = 2'b00;
        end
        7'b0100011: begin // store
            RegWrite   = 0;
            ImmSrc     = 2'b01;
            ALUSrc     = 1;
            MemWrite   = 1;
            ResultSrc  = 2'bxx;
            Branch     = 0;
            ALUOp      = 2'b00;
        end
        7'b0110011: begin // R-type
            RegWrite   = 1;
            ImmSrc     = 2'bxx;
            ALUSrc     = 0;
            MemWrite   = 0;
            ResultSrc  = 2'b00;
            Branch     = 0;
            ALUOp      = 2'b10;
        end
        7'b0010011: begin // I-type (immediate ALU
ops)        RegWrite   = 1;
            ImmSrc     = 2'b00;
            ALUSrc     = 1;
            MemWrite   = 0;
            ResultSrc  = 2'b00;
            Branch     = 0;
            ALUOp      = 2'b10;
        end
        7'b1100011: begin // branch
            RegWrite   = 0;
            ImmSrc     = 2'b10;
            ALUSrc     = 0;
            MemWrite   = 0;
            ResultSrc  = 2'bxx;
            Branch     = 1;
            ALUOp      = 2'b01;
        end
        default: begin
            RegWrite   = 0;
            ImmSrc     = 2'b00;
            ALUSrc     = 0;
            MemWrite   = 0;
            ResultSrc  = 2'b00;
            Branch     = 0;
            ALUOp      = 2'b00;
        end
    endcase
end

endmodule
```

ALU Decoder:

```
module ALUDecoder (
    input  [1:0] ALUOp,
    input  [2:0] funct3,
    input        funct7_5,
    output reg [2:0] ALUControl
);

always @(*) begin
    case (ALUOp)
        2'b00: ALUControl = 3'b000; // load/store → ADD
        2'b01: begin                // branch
            case (funct3)
                3'b000: ALUControl = 3'b010; // BEQ → SUB
                3'b001: ALUControl = 3'b010; // BNE → SUB
                3'b100: ALUControl = 3'b010; // BLT → SUB
                default: ALUControl = 3'b000;
            endcase
        end
        2'b10: begin                // R-type or I-type ALU
            case (funct3)
                3'b000: ALUControl = funct7_5 ? 3'b010 : 3'b000; // SUB or
ADD             3'b001: ALUControl = 3'b001; // SHL
                3'b100: ALUControl = 3'b100; // XOR
                3'b101: ALUControl = 3'b101; // SHR
                3'b110: ALUControl = 3'b110; // OR
                3'b111: ALUControl = 3'b111; // AND
                default: ALUControl = 3'b000;
            endcase
        end
        default: ALUControl = 3'b000;
    endcase
end

endmodule
```

10. Top-Level Processor:

- **Function:** Integrates all components.
- **Data Flow:**
    1. Instruction fetch from ROM.
    2. Register operand fetch.
    3. ALU execution.
    4. Memory access (if needed).
    5. Write back to registers.
- **Clock Domain:**
    1. All state updates occur on the rising edge.
    2. Combines Harvard architecture with single-cycle timing.

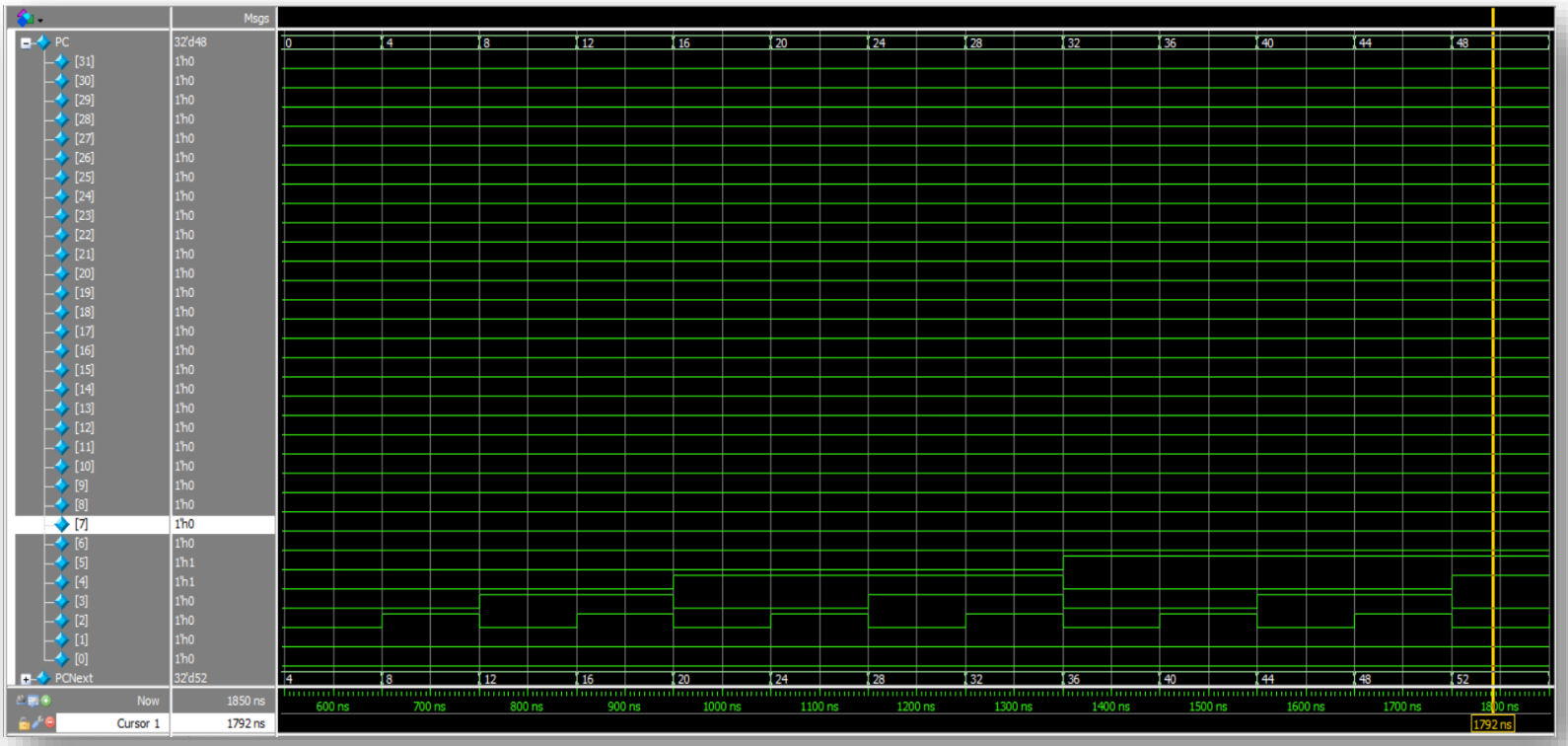
```
module SingleCycleCPU (
    input clk,
    input reset,
    output [31:0] debug_out
);



wire [31:0] PC, PCNext, Instr;
wire [31:0] RD1, RD2, SrcB, ALUResult, MemOut, ImmExt, WD3;
wire [2:0] ALUControl;
wire [1:0] ImmSrc, ResultSrc, ALUOp;
wire RegWrite, ALUSrc, MemWrite, Branch, Zero, Sign;
wire PCSrc;
assign debug_out = ALUResult;

ProgramCounter pc (
    .clk(clk),
    .reset(~reset),
    .load(1'b1),
    .PCNext(PCNext),
    .PC(PC)
);


NextPC pc_calc (
    .PC(PC),
    .ImmExt(ImmExt),
    .PCSrc(PCSrc),
    .PCNext(PCNext)
);


InstructionMemory imem (
    .A(PC),
    .RD(Instr)
);
```

```
38
39
40 MainDecoder main_dec (
41     .opcode(Instr[6:0]),
42     .RegWrite(RegWrite),
43     .ImmSrc(ImmSrc),
44     .ALUSrc(ALUSrc),
45     .MemWrite(MemWrite),
46     .ResultSrc(ResultSrc),
47     .Branch(Branch),
48     .ALUOp(ALUOp)
49 );
50
51
52 ALUDecoder alu_dec (
53     .ALUOp(ALUOp),
54     .funct3(Instr[14:12]),
55     .funct7_5(Instr[30]),
56     .ALUControl(ALUControl)
57 );
58
59
60 RegisterFile rf (
61     .clk(clk),
62     .rst_n(reset),
63     .RegWrite(RegWrite),
64     .A1(Instr[19:15]),
65     .A2(Instr[24:20]),
66     .A3(Instr[11:7]),
67     .WD3(WD3),
68     .RD1(RD1),
69     .RD2(RD2)
70 );
71
72
73 SignExtend sext (
74     .Instr(Instr[31:7]),
75     .ImmSrc(ImmSrc),
76     .ImmExt(ImmExt)
77 );
78
```

```
78
79
80 Mux2 mux_alu (
81     .in0(RD2),
82     .in1(ImmExt),
83     .sel(ALUSrc),
84     .out(SrcB)
85 );
86
87
88 ALU alu (
89     .A(RD1),
90     .B(SrcB),
91     .ALUControl(ALUControl),
92     .ALUResult(ALUResult),
93     .Zero(Zero),
94     .Sign(Sign)
95 );
96
97
98 DataMemory dmem (
99     .clk(clk),
100    .WE(MemWrite),
101    .A(ALUResult),
102    .WD(RD2),
103    .RD(MemOut)
104 );
105
106 assign WD3 = (ResultSrc == 2'b00) ? ALUResult :
107              (ResultSrc == 2'b01) ? MemOut :
108              32'b0;
109
110
111 assign PCSrc = Branch && (
112     (Instr[14:12] == 3'b000 && Zero)   ||
113     (Instr[14:12] == 3'b001 && ~Zero)  ||
114     (Instr[14:12] == 3'b100 && Sign)
115 );
116
117 endmodule
118
```

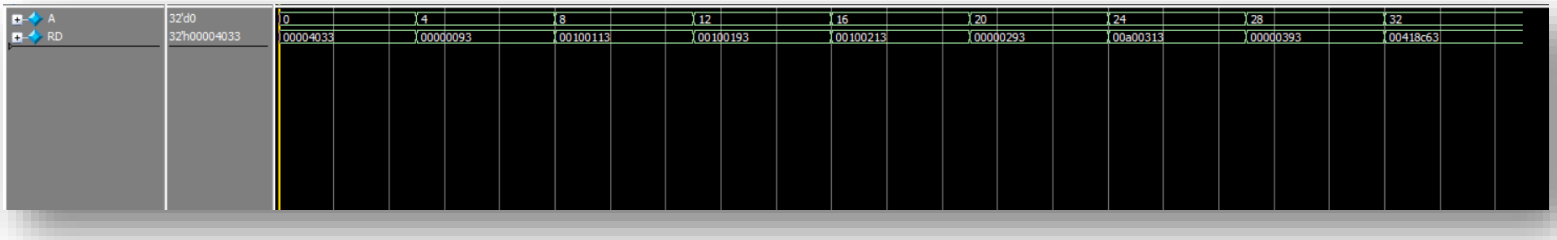
5. Simulation:

1. Program Counter Simulation

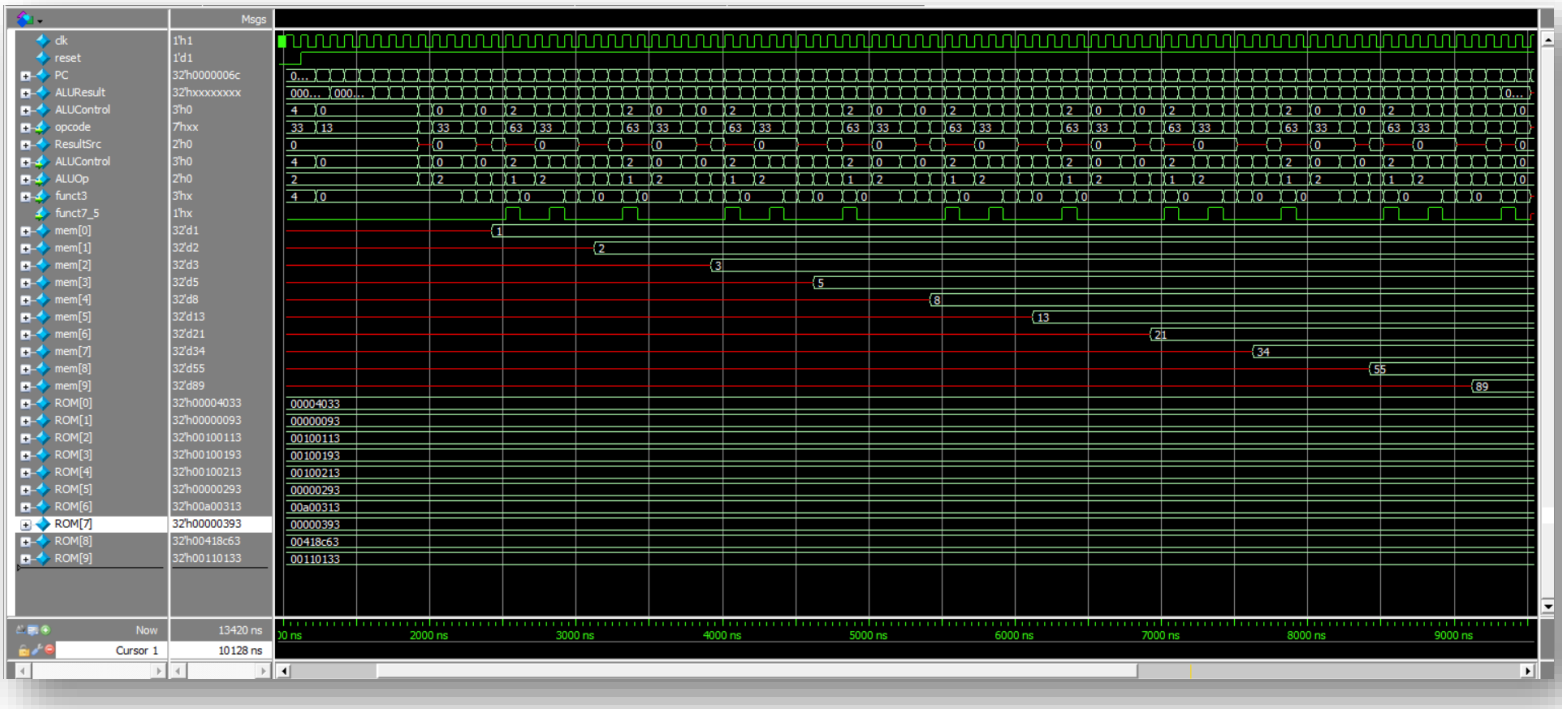


PC is incremented by 4 every cycle.

2. Checking the Instruction Memory for FIBONACCI series:

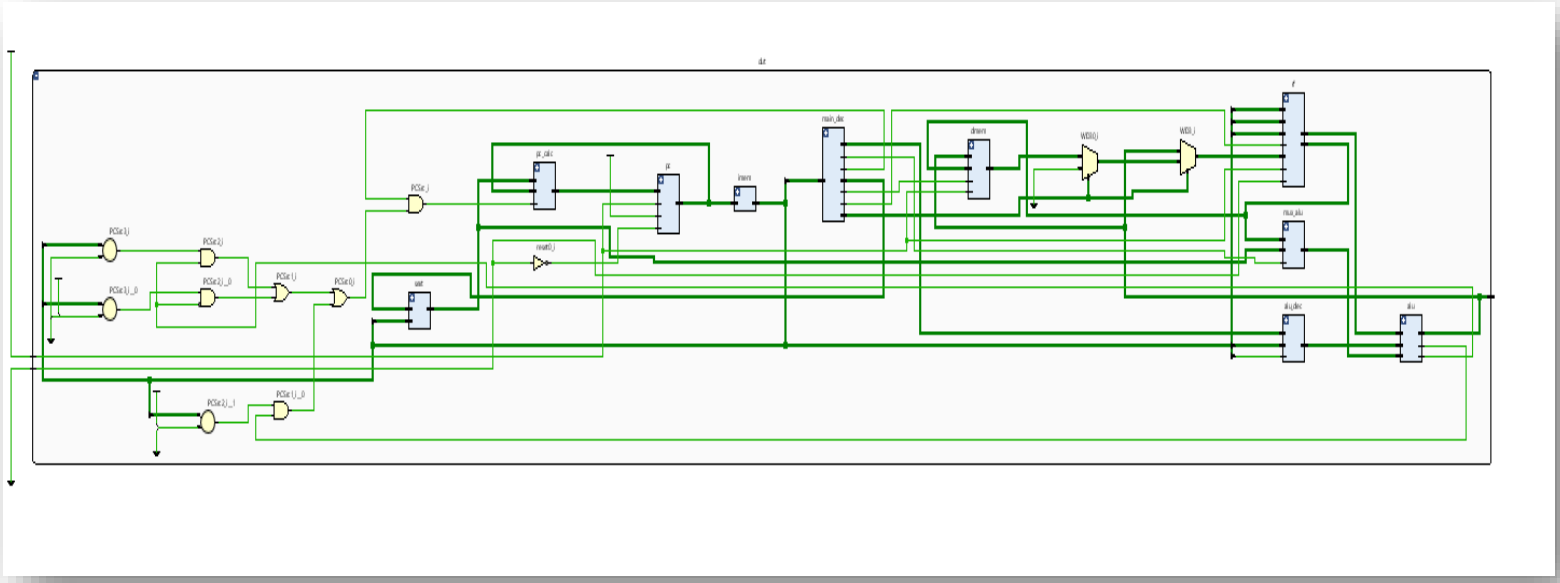


3. Final Simulation:

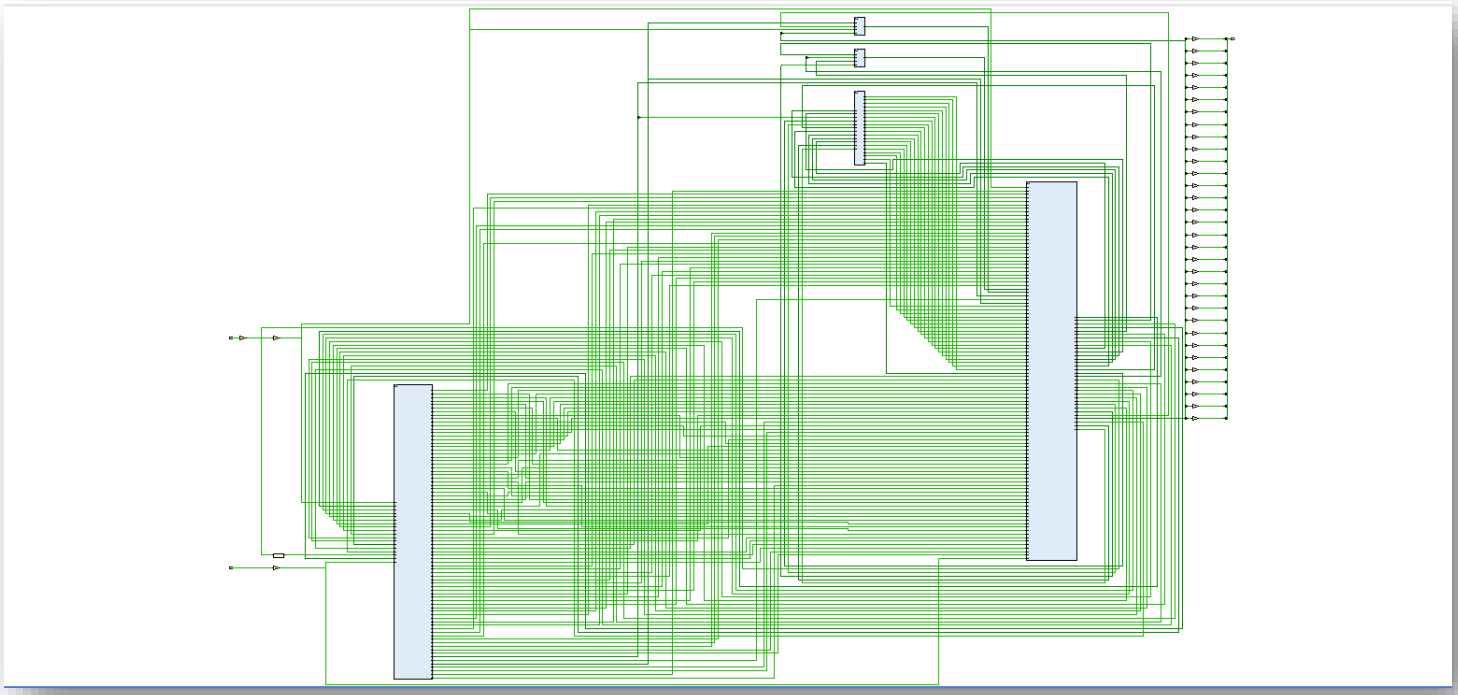


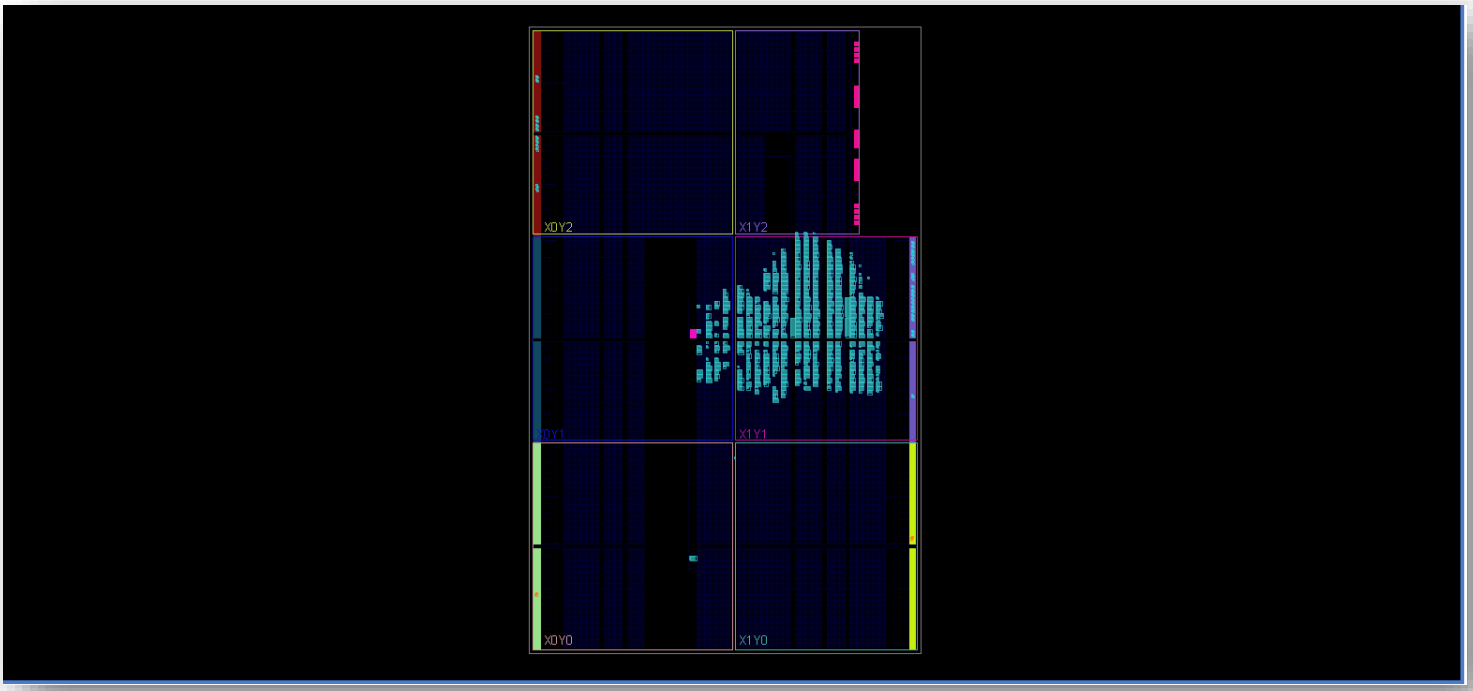
6. Vivado Elaboration Circuit:

1. RTL Analysis:



2. Synthesis and Implementation Analysis:





7. Conclusion:

A complete, working single-cycle RISC-V CPU was implemented from scratch in Verilog, simulated with a real program, and deployed on real FPGA hardware. The design is modular, readable, and easy to expand. Future improvements could include pipelining, exception handling, and support for more instructions.

I have gained a deep insight into CPU design and how RISC-V instructions are executed in hardware.

Thank you, all IEEE Digital Design team, for this great workshop.