Design and Implementation of a Single-Cycle RISC-V Processor

Name: Meesam Abbas
Course: VLSI
Date: 10 - October -2025

Table of Contents

Abstract:

This report presents the design, implementation, and verification of a Single-Cycle RISC-V Processor using SystemVerilog. The processor executes one instruction per clock cycle, covering all five major stages of a CPU Instruction Fetch, Decode, Execute, Memory Access, and Write Back within a single clock period. The design follows the RV32I base integer instruction set, which includes arithmetic, logical, branch, memory access, and jump instructions.

Each functional block such as the ALU, Control Unit, Register File, and Data Memory was designed as a separate SystemVerilog module to promote modularity, reusability, and clarity. Simulation and verification were carried out using ModelSim, and synthesis was performed in Intel Quartus Prime.

This design demonstrates a foundational RISC-V architecture suitable for understanding CPU operation and provides a base for further extensions such as pipelining, hazard detection, and peripheral interfacing.

Introduction:

Modern processors are based on two fundamental design philosophies: Complex Instruction Set Computing (CISC) and Reduced Instruction Set Computing (RISC). The RISC-V architecture, originating from the University of California, Berkeley, is a free and open-source ISA that embodies the simplicity and efficiency of RISC design.

RISC-V aims to provide:

- A **modular ISA**, where extensions (integer, floating-point, atomic, etc.) can be added as needed.
- **Open access**, allowing universities, researchers, and companies to build custom processors without licensing costs.
- **Scalability**, making it suitable for both microcontrollers and high-performance CPUs.

In this project, a **single-cycle** RISC-V CPU is implemented — meaning each instruction is executed entirely within one clock cycle. This makes the control and data paths simpler, though it sacrifices clock speed compared to pipelined or multi-cycle designs.

Objectives:

The objectives of this project are:

1. To implement a functional single-cycle RISC-V processor using SystemVerilog.
2. To design and integrate individual functional modules including control logic, ALU, and memory blocks.
3. To execute a subset of RV32I instructions accurately.
4. To simulate, verify, and synthesize the design using ModelSim and Quartus Prime.
5. To build a modular foundation for further enhancements such as pipelining and cache integration.

Background and Literature Review:

The RISC-V ISA defines a 32-bit base integer instruction set (RV32I) that includes:

- Arithmetic and logic operations (ADD, SUB, AND, OR, SLT, etc.)
- Memory operations (LW, SW)
- Control flow operations (BEQ, BNE, JAL, JALR)
- Immediate operations (ADDI, LUI)

Single-cycle processors implement each instruction completely in one clock cycle. Although simple and easy to understand, they are not optimal in terms of performance or clock frequency because the cycle time must be long enough to accommodate the slowest instruction (like memory access).

The structure of this design is inspired by textbook architectures such as:

- *Computer Organization and Design: RISC-V Edition* by Patterson and Hennessy.

Design Methodology:

The CPU is divided into datapath and control path:

- Datapath: Handles the flow of data and operations (registers, ALU, memory, etc.)
- Control Path: Generates signals to guide datapath behavior (selects, enables, writes, etc.)

Design Flow:

1. Specification : Define instruction set and required functionality.
2. Datapath Design : Develop register file, ALU, memory, and PC logic.
3. Control Design : Implement main decoder and ALU decoder.
4. Integration: Connect all modules in the top-level SystemVerilog module.
5. Simulation : Verify execution sequence and data correctness.
6. Synthesis: Compile the design for FPGA compatibility.

System Architecture:

The Single-Cycle RISC-V CPU performs the following operations per cycle:

1. **Instruction Fetch (IF)**

   Fetch the instruction from instruction memory.

2. **Instruction Decode (ID)**

   Decode opcode and generate control signals.

3. **Execution (EX)**

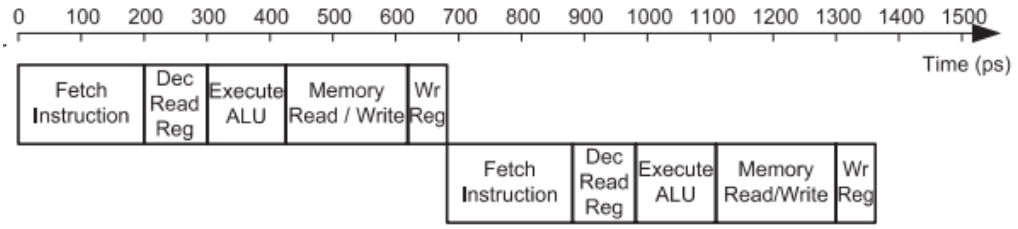
   Perform ALU operations or calculate branch target.

4. **Memory Access (MEM)**

   Read/write from/to data memory if required.

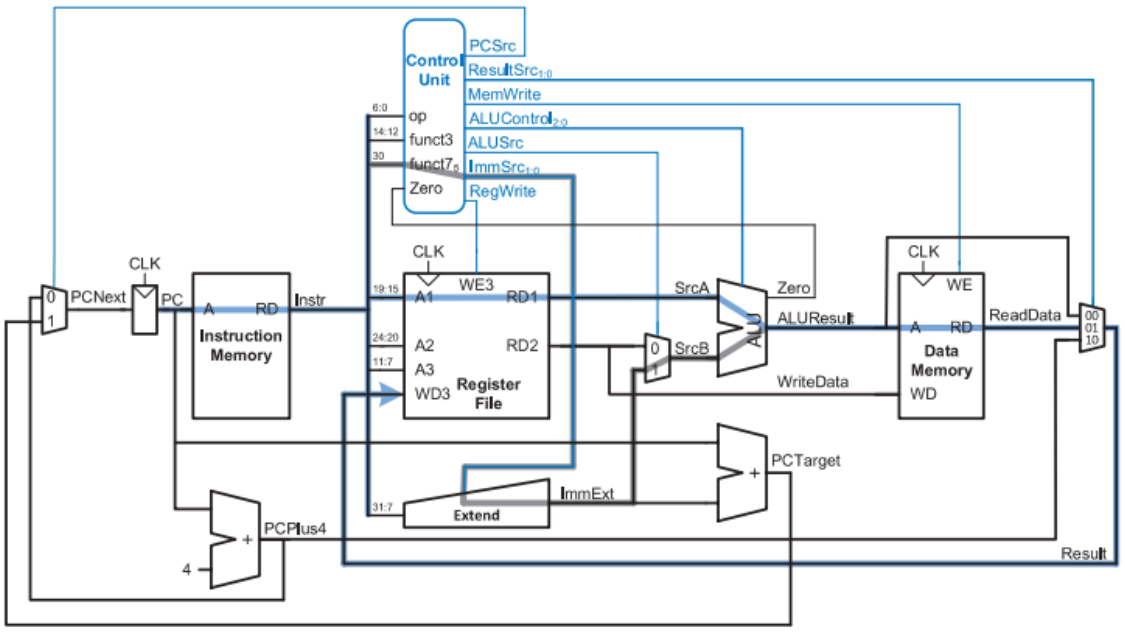
5. **Write Back (WB)**

   Write results back to the register file.



A simplified block diagram includes:

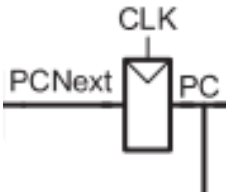
- Program Counter (PC)
- Instruction Memory
- Register File
- Immediate Extender
- ALU
- Data Memory
- Control Unit (Main + ALU Decoder)
- Branch Unit
- Multiplexers (2:1) / (3:1)
- Pipeline Registers



Module Descriptions

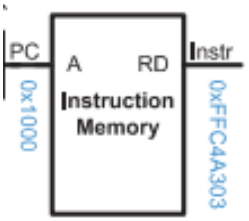
Program Counter (PC):

- A 32-bit register holding the address of the current instruction.
- On each clock cycle, updates to PC + 4, or to branch/jump target.



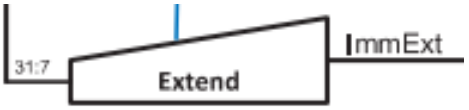
Instruction Memory:

- ROM that stores machine code instructions.
- Addressed by PC, outputs 32-bit instruction.



Immediate Extender:

- Extracts immediate fields from instruction and extends them to 32 bits based on type (I, S, B, U, J).

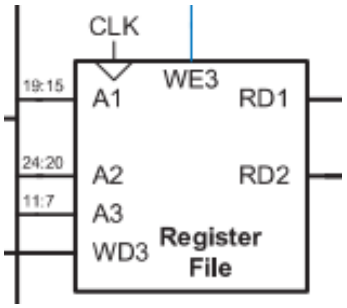


Supported Operations

ImmSrc	ImmExt	Type	Description
00	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
01	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
10	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1'b0}	B	13-bit signed immediate
11	{{12{*Instr*[31]}}, *Instr*[19:12], *Instr*[20], *Instr*[30:21], 1'b0}	J	21-bit signed immediate

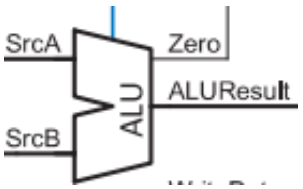
Register File:

- 32 registers × 32 bits each.
- Two read ports and one write port.
- Controlled by regwrite signal.



ALU (Arithmetic Logic Unit):

- Performs arithmetic and logical operations.
- Input sources selected through a 2:1 multiplexer (alusrc).
- Sets a zero flag if result equals zero (used for branching).



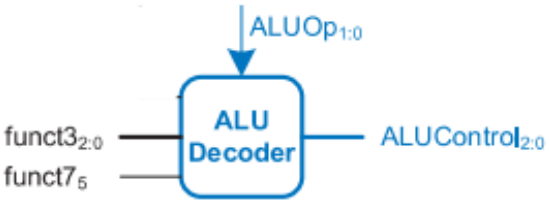
Supported Operations

ALUControl	Instruction
000 (add)	`lw, sw`
001 (subtract)	`beq`
000 (add)	`add`
001 (subtract)	`sub`
101 (set less than)	`slt`
011 (or)	`or`
010 (and)	`and`

ALU Decoder:

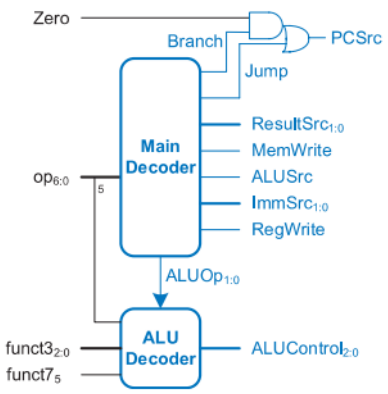
- Interprets funct3, funct7, and aluop signals to generate ALU control

ALUOp	funct3	$\{op_5, funct7_5\}$	ALUControl	Instruction
00	x	x	000 (add)	`lw, sw`
01	x	x	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add`
	000	11	001 (subtract)	`sub`
	010	x	101 (set less than)	`slt`
	110	x	011 (or)	`or`
	111	x	010 (and)	`and`



Control Unit:

- Decodes opcode to generate control signals:
    - Regwrite: enables register write
    - Memwrite: enables memory write
    - branch, jump, jalr : control PC logic
    - Alusrc: select between register or immediate
    - Resultsrc: select between ALU result or Memory data
    - Aluop: selects ALU operation type

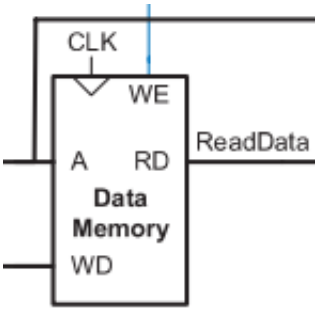


Supported Operations

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
lw	0000011	1	00	1	0	01	0	00	0
sw	0100011	0	01	1	1	xx	0	00	0
R-type	0110011	1	xx	0	0	00	0	10	0
beq	1100011	0	10	0	0	xx	1	01	0
I-type ALU	0010011	1	00	1	0	00	0	10	0
jal	1101111	1	11	x	0	10	0	xx	1

Data Memory:

- Byte-addressable memory (256 bytes).
- Supports read and write operations.
- Used by load (LW) and store (SW) instructions.

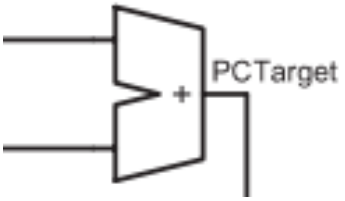


Multiplexer (mux2):

- Selects between two 32-bit inputs based on a control signal.
- Used for ALU input and write-back path selection.

Branch Unit:

- Calculates next PC address.
- Determines if branch/jump should be taken:
    - For BEQ/BNE → depends on branch & zero
    - For JAL/JALR → unconditional jump



Pipeline Register:

- Used to hold intermediate signals for pipelined extension.

Working Principle:

The single-cycle RISC-V processor executes one complete instruction per clock cycle. The sequence of operation in Quartus simulation and hardware behavior is as follows:

1. **Instruction Fetch:**
   The Program Counter (PC) fetches the instruction from instruction_memory.
2. **Instruction Decode:**
   The control_unit and imm_extender decode the opcode, funct3, and funct7 fields to generate necessary control signals and immediate values.
3. **Execution:**
   The alu performs the required arithmetic or logic operation based on ALU control signals.
4. **Memory Access:**
   For LW and SW, data is read or written through the data_memory module.
5. **Write Back:**
   The final result (either from ALU or memory) is written back into the register_file.
6. **PC Update:**
   The branch_unit and mux2 decide whether the next instruction address comes from a sequential increment (PC + 4) or a branch/jump target.

Quartus Implementation and Working:

The designed Single-Cycle RISC-V Processor was implemented in Intel Quartus Prime software to verify synthesis and hardware feasibility.

System Verilog:

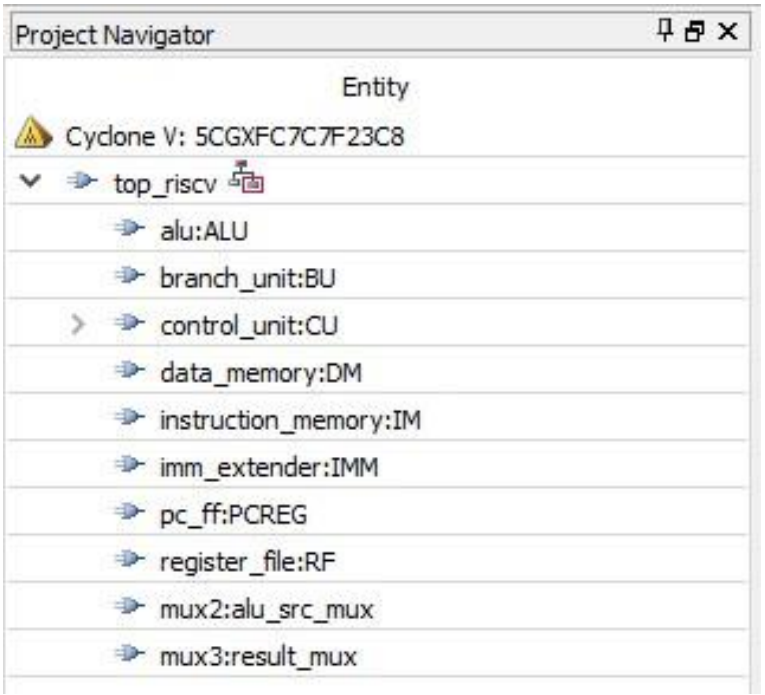
```
module top_riscv (
    input  logic clk,
    input  logic reset
);
    // ----- core signals -----
    logic [31:0] pc, pc_next, pc_plus4, instr;
    logic [31:0] imm_ext;
    logic [31:0] rd1, rd2;
    logic [31:0] alu_b, alu_result;
    logic [31:0] mem_rd, write_data;
    logic [31:0] pc_target;
    logic [6:0]  opcode, funct7;
    logic [2:0]  funct3;
    logic zero, branch, jump, jalr;
    logic alusrc, regwrite, memwrite;
    logic [1:0] resultsrc;
    logic [2:0] immsrc;
    logic [2:0] alucontrol;
    logic pcsrc, take_branch;

    // ----- instruction memory -----
    instruction_memory IM (
        .addr(pc),
        .instr(instr)
    );
```

```
    // decode fields
    assign opcode = instr[6:0];
    assign funct3 = instr[14:12];
    assign funct7 = instr[31:25];

    // PC + 4
    assign pc_plus4 = pc + 32'd4;

    // immediate
    imm_extender IMM (
        .immsrc(immsrc),
        .instr(instr),
        .Imm(imm_ext)
    );

    // register file
    register_file RF (
        .clk(clk),
        .regwrite(regwrite),
        .A1(instr[19:15]),
        .A2(instr[24:20]),
        .A3(instr[11:7]),
        .WD(write_data),
        .RD1(rd1),
        .RD2(rd2)
    );
```

```
53
54      // control unit (uses instr fields + ALU zero)
55      control_unit CU (
56          .opcode(opcode),
57          .func3(funct3),
58          .func7(funct7),
59          .zero(zero),
60          .pcsrc(pcsrc),
61          .alusrc(alusrc),
62          .branch(branch),
63          .jump(jump),
64          .jalr(jalr),
65          .regwrite(regwrite),
66          .memwrite(memwrite),
67          .resultsrc(resultsrc),
68          .immsrc(immsrc),
69          .alucontrol(alucontrol)
70      );
71
72      // ALU input mux (reg_b or immediate)
73      mux2 #(32) alu_src_mux (
74          .d0(rd2),
75          .d1(imm_ext),
76          .sel(alusrc),
77          .y(alu_b)
78      );
79
```

```
79
80      // ALU (3-bit control)
81      alu ALU (
82          .a(rd1),
83          .b(alu_b),
84          .alu_control(alucontrol),
85          .result(alu_result),
86          .zero(zero)
87      );
88
89      // Data memory (byte-addressable)
90      data_memory DM (
91          .clk(clk),
92          .memwrite(memwrite),
93          .addr(alu_result),
94          .wd(rd2),
95          .rd(mem_rd)
96      );
97
98      // Write-back mux (ALU / MEM / PC+4)
99      mux3 #(32) result_mux (
100         .d0(alu_result),
101         .d1(mem_rd),
102         .d2(pc_plus4),
103         .s(resultsrc),
104         .y(write_data)
105     );
106
```

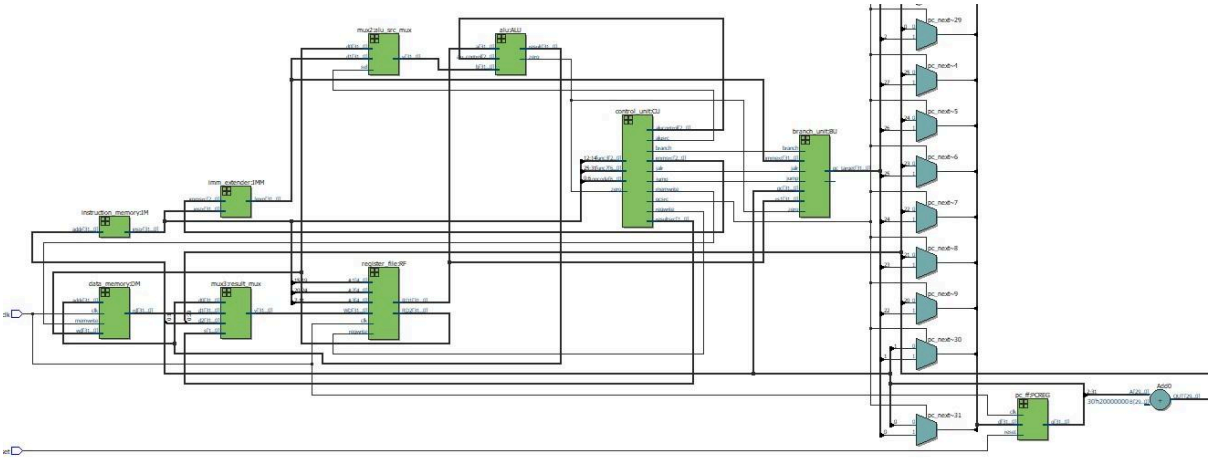
```
107         // branch/jump target generator
108         branch_unit BU (
109             .branch(branch),
110             .jump(jump),
111             .jalr(jalr),
112             .zero(zero),
113             .pc(pc),
114             .immext(imm_ext),
115             .rs1(rd1),
116             .pc_target(pc_target),
117             .take_branch(take_branch)
118         );
119
120         // PC next selection (use pcsrc computed by CU)
121         assign pc_next = pcsrc ? pc_target : pc_plus4;
122
123         // PC register
124         pc_ff PCREG (
125             .clk(clk),
126             .reset(reset),
127             .d(pc_next),
128             .q(pc)
129         );
130
131     endmodule
132
```



Design Verification and Visualization

After compilation, Quartus tools were used to analyze and confirm the correct design structure:

- **RTL Viewer:**
  Displayed the hierarchical interconnection of all modules — Control Unit, ALU, Register File, Data Memory, and PC logic. This confirmed proper top-level integration.



Limitations:

- Single-cycle design leads to long critical path (slow clock frequency).
- Does not handle data or control hazards (since it is not pipelined).
- No interrupt or exception handling.
- Limited instruction set (basic RV32I subset).

Future Enhancements:

1. **Pipeline Design:** Introduce IF/ID, ID/EX, EX/MEM, MEM/WB registers to improve throughput.
2. **Hazard Detection Unit:** Resolve data and control hazards automatically.
3. **Forwarding Unit:** Allow results to bypass pipeline stalls.
4. **Cache Integration:** Add instruction and data cache.
5. **Peripheral Interfacing:** Extend design to support UART, timers, or GPIO for embedded systems.

Conclusion:

The project successfully demonstrates the design and implementation of a Single-Cycle RISC-V processor using SystemVerilog.
All components were built from scratch, interconnected, and verified through simulation. The processor executes essential RV32I instructions and illustrates the internal working of a CPU datapath and control logic.

This work lays the foundation for more advanced designs such as pipelined or multi-cycle processor and strengthens understanding of computer architecture concepts in practical hardware design.

References:

1. David A. Patterson & John L. Hennessy, *Computer Organization and Design: RISC-V Edition*, Elsevier.
2. Harris, David & Sarah Harris, *Digital Design and Computer Architecture: RISC-V Edition*, Morgan Kaufmann.
3. riscv.org — Official RISC-V documentation and resources.