

SINGLE CYCLE PROCESSOR USING VERILOG

**HEMANTH S, NAVEEN KUMAR B,
SABARISH MOHAN JS**



RISC-V

RISC-V BASED SINGLE CYCLE PROCESSOR

Contributors:

Hemanth S

Naveen Kumar B

Sabarish Mohan JS

Introduction:

This document details the design and functional verification of a **single-cycle RISC-V processor**. The project is implemented using a hardware description language and is verified entirely within a **simulation environment**. By focusing on a logical and software-based approach, this work provides a comprehensive understanding of core computer architecture principles without the complexities of physical hardware. The primary goal is to demonstrate the correct behavior of a RISC-V compliant processor core, validating its functionality through a rigorous **testbench-driven verification methodology**. The project showcases the fundamental stages of instruction execution, including fetching, decoding, and executing instructions within a single clock cycle, ensuring the design's logical correctness and integrity.

Key Steps:

Here are the key steps for your single-cycle RISC-V processor project, based on the structure of the UART document:

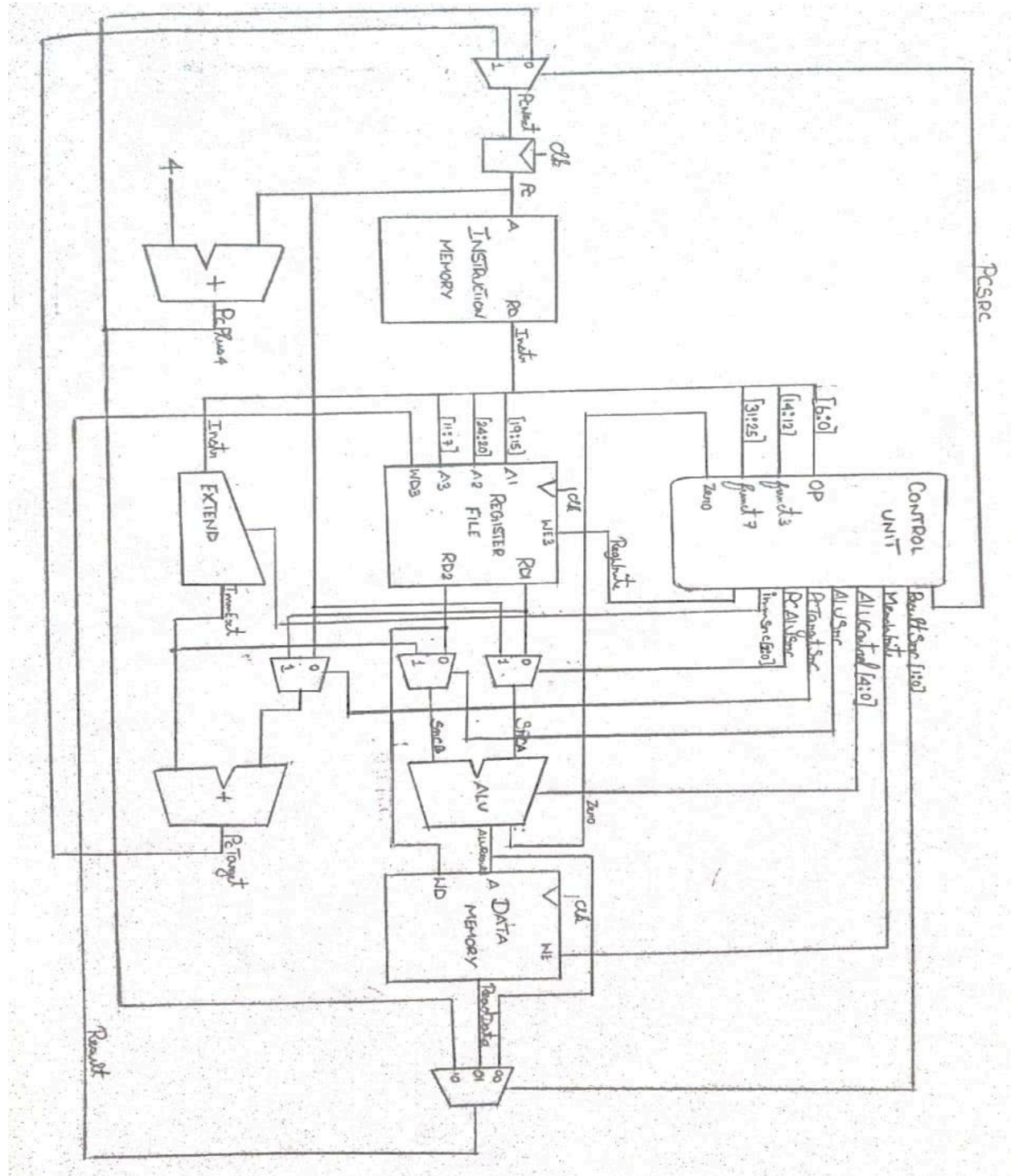
- **Gain a deep understanding of the RISC-V ISA** and single-cycle processor architecture.
 - **Implement robust Verilog RTL** for core components such as the ALU, Register File, and Control Unit.
 - **Integrate all modules** into a single-cycle datapath.
 - **Conduct rigorous simulation verification** to ensure all instructions execute correctly.
 - **Leverage Xilinx Vivado tools** for simulation.
 - **Maintain detailed project logs** and source control on GitHub.
-

Workflow:

1. Define supported RV32I instructions (R, I, S, B, U, J types).
2. Implement Program Counter (PC) register and next-PC logic.
3. Create Instruction Memory (ROM with '\$readmemh').
4. Decode instruction fields ('opcode', 'rd', 'rs1', 'rs2', 'funct3', 'funct7', 'imm').
5. Design Control Unit to generate control signals.
6. Build Register File with 32×32 registers (2 read, 1 write).
7. Implement Immediate Generator (Extend Unit).
8. Create ALU supporting ADD, SUB, AND, OR, XOR, SLT, shifts.
9. Add ALU Control and Control Unit to decode 'funct3/funct7' into ALU operations and to generate control signals.
10. Design Data Memory for load/store instructions.
11. Add Multiplexers: ALUSrc, ResultSrc, PCSrc.
12. Implement Branch/Jump Target Calculation (PC + ImmExt).
13. Handle Write Back (ALU result / Mem data / PC+4 → Register File).

14. Connect all blocks in a Top Module('riscv32_singlecycle.v').
15. Write a Testbench, load the program with '\$readmemh', simulate, and verify.

RISC-V Architecture:



Module explanation:

pc:

- On a negative edge of the **reset signal (rst)**, the PC is initialized to 32'b0³. This is the starting address for the program.
- On a positive edge of the **clock signal (clk)**, the PC updates its value to the address provided by the PCNext input.

inst_mem:

The inst_mem module is the **Instruction Memory** of the processor. Its main function is to hold and provide instructions to the processor¹. It works as a read-only memory for the main CPU module

- **Initialization:** The module starts by loading a program from an external file named "instruction_memory.mem"². This means it's pre-populated with a set of instructions.
- **Instruction Access:** It takes a 32-bit address (A) as input and outputs a 32-bit instruction (RD)³.
- **Address Translation:** The module uses bits [6:2] of the input address A to access its internal memory array⁴. This effectively divides the byte-address from the Program Counter by 4, converting it to a word-address, which is a standard practice for fetching 32-bit instructions.

pc_next:

- pc_next module calculates the next instruction address in the RISC-V processor.
- It takes the current **Program Counter (PC)** as input.
- Adds **4** (since each instruction is 32 bits = 4 bytes).
- Outputs PCPlus4, the address of the next sequential instruction.

extend:

The extend module generates the **immediate value** from the instruction based on its type.

Input: 32-bit instruction (instr) and immediate source selector (ImmSrc).

Uses case to decode and **sign-extend** immediates for I, S, B, J, and U type instructions.

Output: 32-bit extended immediate (ImmExt) used in ALU and branch/jump calculations.

Register file

The register_file module is a set of 32 registers that store 32-bit data. It performs both read and write operations.

- **Read Operation:** This is an asynchronous process. Data from two registers, specified by addresses A1 and A2, are immediately available at the RD1 and RD2 outputs, respectively.
- **Write Operation:** This is a synchronous process, controlled by the clock signal. On the positive edge of the clk, if the WE3 (write enable) signal is high, the data on the WD3 input is written to the register specified by the A3 address.

pc_target:

The pc_target module computes the **branch/jump target address**.

Inputs: current PC value (PCSrcA) and extended immediate (ImmExt).

It adds both to calculate the target instruction address.

Output: PCTarget, used when a branch or jump is taken

alu:

The alu module is a combinational circuit that performs arithmetic, logical, and shift operations on two 32-bit inputs, SrcA and SrcB. The specific operation is determined by the ALUControl input.

- **Operation Selection:** A case statement evaluates the ALUControl input to select the desired operation, such as addition, subtraction, AND, OR, XOR, or various shift and comparison operations.
- **Result Output:** The result of the selected operation is stored in the ALUResult register.
- **Zero Flag:** The Zero output is set to 1 if the final ALUResult is 32'b0; otherwise, it is 0. This flag is crucial for conditional branching.

data_mem:

The data_mem module implements the **data memory** for load and store instructions.

Input: address (A), write data (WD), write enable (WE), clock (clk), and reset (rst).

On read, it outputs the stored word (RD) from the given address.

On write (when WE=1), it stores the input data (WD) into memory at address A.

Memory is initialized from an external file (data_mem.mem) using \$readmemh.

mux_regtoalu:

The mux_regtoalu module selects the **second operand (SrcB)** for the ALU.

Inputs: register value (RD2), extended immediate (ImmExt), and control signal (ALUSrc).

If ALUSrc=1, it forwards **ImmExt**; otherwise, it forwards **RD2**.

Used in immediate-type instructions (like **ADDI**, **LW**, **SW**) where ALU needs an immediate operand.

reg_to_pctarget:

The reg_to_pctarget module selects the source for the branch/jump base address.

Inputs: register value (RD1), program counter (PC), and control signal (PCTargetSrc).

If PCTargetSrc=1, it chooses RD1 (for instructions like **JALR**); otherwise, it uses PC.

Output: PCSrcA, the base address for target calculation.

mux_dmtoreg:

The mux_dmtoreg module is a **multiplexer** that selects the final value to be written back to the register file..

- The three possible inputs are the **ALUResult**, **ReadData** from the data memory, and the **PCPlus4** value.
- Based on ResultSrc (from the control unit), one of these three values is routed to the **Result** output, which is then written to the register file.
- This module is positioned after the ALU and data memory stages to choose the correct final value for different instruction types.

mux_pctoalu:

The mux_pctoalu module selects the **first operand (SrcA)** for the ALU.

Inputs: program counter (PC), register value (RD1), and control signal (PCALUSrc).

If PCALUSrc=1, it forwards the **PC**; otherwise, it forwards **RD1**.

Used in instructions like **AUIPC**, where PC is added with an immediate

Pcmux:

The PCMux module is a **multiplexer** that selects the next address for the program counter. It's a simple logical gate that determines the flow of the program.

- **Inputs:** It takes a PCSrc signal from the control unit and two potential addresses: PCTarget (for branches and jumps) and PCPlus4 (for sequential execution).
- **Output:** The PCNext signal is the chosen address, which is then sent to the Program Counter.

aludecoder:

1. The aludecoder is the **main control unit** of the RISC-V single-cycle processor.
2. It interprets the **opcode**, **funct3**, and **funct7** fields of the instruction.
3. Based on these fields, it decides what type of instruction is being executed (R-type, I-type, Load, Store, Branch, Jump, U-type).

4. It generates **control signals** that guide the datapath: register writes, memory access, immediate generation, ALU input selection, and result selection.
5. It produces the **ALUControl** signal that tells the ALU which operation to perform (add, subtract, shift, logic, compare, etc.).
6. It manages **branch and jump decisions**, using the Zero flag and jump control to select the next PC.
7. It ensures the **correct immediate format** (I, S, B, J, U) is extracted and extended for the instruction.
8. It controls whether the ALU operates on register values or immediates, depending on the instruction type.
9. It determines if data should be read from memory, written to memory, or bypassed directly from the ALU.
10. In essence, it acts as the **brain of the processor datapath**, coordinating all modules so that each instruction is executed correctly in a single cycle

topmodule:

The top_module is the main file that connects all the other modules to create a complete single-cycle RISC-V processor.

1. It acts as a **central hub** that integrates all the sub-modules like the PC, instruction memory, and ALU.
 2. It defines the **wires** that connect the inputs and outputs of each module.
 3. It passes the **clock (clk) and reset (rst)** signals to all the components that need them for synchronization.
 4. It connects the **PC's output** to the instruction memory to fetch instructions.
 5. It links the instruction's register fields to the **register file** for reading and writing data.
 6. It directs the data from the register file and immediate extender to the **ALU** for computation.
 7. It connects the outputs of the ALU and data memory to a multiplexer that selects the final data for **write-back** to the register file.
 8. It connects the control signals from the **control unit** to all the relevant components (e.g., RegWrite, MemWrite, ALUControl).
 9. It sets up the logic for **program flow** by connecting the PC and branch/jump multiplexers.
 10. The top_module essentially builds the complete **datapath** of the processor, orchestrating the flow of instructions and data through the entire system.
-

Simulation:

Example : sum of the first 10 natural numbers

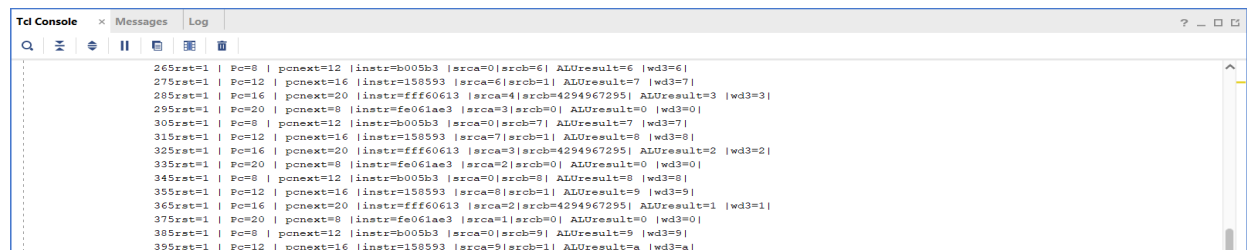
RISC-V Assembly Language:

```
addi a1,zero,0
addi a2,zero,10
loop:
    add a1,zero,a1
    addi a1,a1,1
    addi a2,a2,-1
    bne a2,zero,loop
```

Machine language:

```
00000593
00a00613
00b005b3
00158593
fff60613
Fe061ae3
```

(Store the machine language instructions in the instruction_memory.mem file)



```
Tcd Console x Messages Log
265rst=1 | Pc=8 | pnext=12 | instr=b005b3 | srca=0 | srcb=6 | ALUresult=6 | wd3=6 |
275rst=1 | Pc=12 | pnext=16 | instr=158593 | srca=6 | srcb=1 | ALUresult=7 | wd3=7 |
285rst=1 | Pc=16 | pnext=20 | instr=fff60613 | srca=4 | srcb=4294967295 | ALUresult=3 | wd3=3 |
295rst=1 | Pc=20 | pnext=8 | instr=fe061ae3 | srca=3 | srcb=0 | ALUresult=0 | wd3=0 |
305rst=1 | Pc=8 | pnext=12 | instr=b005b3 | srca=0 | srcb=7 | ALUresult=7 | wd3=7 |
315rst=1 | Pc=12 | pnext=16 | instr=158593 | srca=7 | srcb=1 | ALUresult=8 | wd3=8 |
325rst=1 | Pc=16 | pnext=20 | instr=fff60613 | srca=3 | srcb=4294967295 | ALUresult=2 | wd3=2 |
335rst=1 | Pc=20 | pnext=8 | instr=fe061ae3 | srca=2 | srcb=0 | ALUresult=0 | wd3=0 |
345rst=1 | Pc=8 | pnext=12 | instr=b005b3 | srca=0 | srcb=8 | ALUresult=8 | wd3=8 |
355rst=1 | Pc=12 | pnext=16 | instr=158593 | srca=8 | srcb=1 | ALUresult=9 | wd3=9 |
365rst=1 | Pc=16 | pnext=20 | instr=fff60613 | srca=2 | srcb=4294967295 | ALUresult=1 | wd3=1 |
375rst=1 | Pc=20 | pnext=8 | instr=fe061ae3 | srca=1 | srcb=0 | ALUresult=0 | wd3=0 |
385rst=1 | Pc=8 | pnext=12 | instr=b005b3 | srca=0 | srcb=9 | ALUresult=9 | wd3=9 |
395rst=1 | Pc=12 | pnext=16 | instr=158593 | srca=9 | srcb=1 | ALUresult=a | wd3=a |
```

Github repository link:

[Github repo link for RISC-V single cycle processor.](#)

Future improvements:

Pipelining: Divide the processor into stages (like Instruction Fetch, Decode, Execute, Memory, and Write Back) to allow multiple instructions to be processed at the same time. This will greatly increase the processor's speed.

References:

Sarah Harris Digital Design & Computer Architecture RISC-V Edition