# RISC-V Processor Design

***Abstract***- *This report delves into the design and implementation of a RISC-V processor using the Verilog hardware description language. We explore the fundamental principles of the RISC-V architecture, examine the design process from instruction set definition to circuit realization, and discuss the verification and validation methodologies employed to ensure functional correctness and optimal performance. The document also presents an analysis of the processor's performance and explores techniques for optimization.*

## Introduction

The RISC-V architecture is a modern, open-source instruction set architecture (ISA) gaining significant traction in academia and industry. Its modularity and openness encourage innovation and customization, making it suitable for a wide range of applications, from embedded systems to high-performance computing. The RISC-V ISA features a simple and elegant design, prioritizing efficiency and performance. It employs a load-store architecture, meaning that data access is limited to explicit load and store instructions, promoting a streamlined execution model. One of the key advantages of RISC-V is its extensibility. The architecture allows for custom instructions to be added, tailoring the ISA to specific application requirements. This flexibility fosters innovation and enables developers to optimize performance for their particular use cases. The RISC-V instruction set is divided into various categories based on their functionality, encompassing basic arithmetic operations, data transfer instructions, control flow mechanisms, and more.

## Design and Implementation of the RISC-V Processor

The design and implementation of a RISC-V processor using Verilog involves multiple stages, from specification to synthesis and testing. We start by defining the desired features and functionality of the processor, including the instruction set, register file size, and memory organization. The Verilog code is then developed based on this specification, representing the processor's hardware components in a structured and modular manner. The design utilizes a combination of combinational and sequential logic to implement the processor's various functionalities.

The design process involves careful consideration of performance trade-offs, such as pipeline stages, memory access schemes, and data path optimization. Verilog provides a powerful and flexible tool for describing the hardware design, enabling precise control over the processor's behavior. The Verilog code is then synthesized into a hardware description language (HDL) that can be understood by hardware synthesis tools, ultimately generating a circuit description suitable for fabrication.

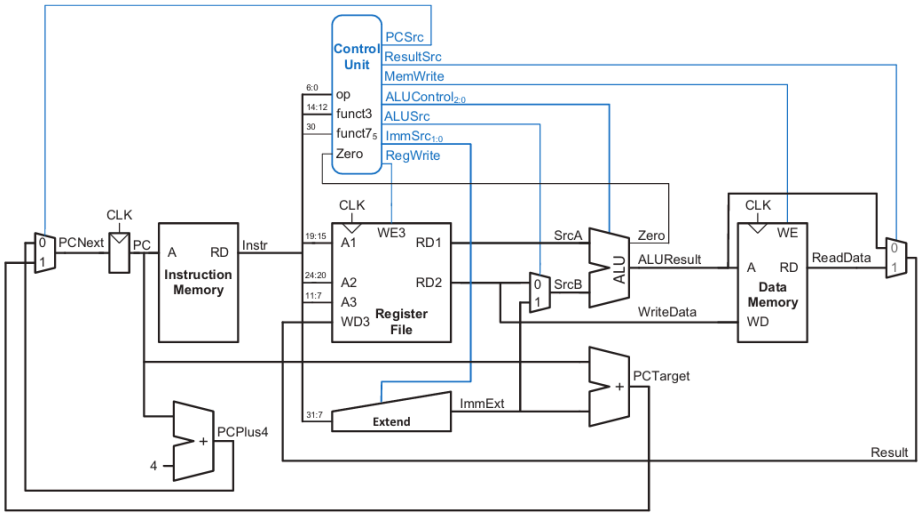


Figure 7.12 Complete single-cycle processor

In a single-cycle RISC-V processor, each instruction is executed in one clock cycle. Here's the process:

1. **Instruction Fetch**: The instruction is fetched from memory, and the PC is incremented.
2. **Instruction Decode**: The instruction is decoded to determine its type and generate control signals.
3. **Register Read**: Source registers are read from the register file.
4. **Execution**: The ALU performs the operation (e.g., addition, comparison) or calculates the memory address for load/store.
5. **Memory Access**: Data is read from or written to memory (for load/store instructions).
6. **Write Back**: The result is written back to the destination register.

| Modules Used | I/O ports |
| --- | --- |
| **module top_riscv_cpu** | input clk, reset,<br>input Ext_MemWrite,<br>input [31:0] Ext_WriteData, Ext_DataAdr,<br>output MemWrite,<br>output [31:0] WriteData, DataAdr, ReadData,<br>output [31:0] PC, Result |
| **module adder** | input [WIDTH-1:0] a, b,<br>output [WIDTH-1:0] sum |
| **module alu** | input [WIDTH-1:0] a, b,<br>input [3:0] alu_ctrl,<br>output reg [WIDTH-1:0] alu_out,<br>output zero |
| **module alu_decoder** | input opb5,<br>input [2:0] funct3,<br>input funct7b5,<br>input [1:0] ALUOp,<br>output reg [3:0] ALUControl |
| **module controller** | input [6:0] op,<br>input [2:0] funct3,<br>input funct7b5,<br>input Zero,ALUR31,<br>output [1:0] ResultSrc,<br>output MemWrite,<br>output PCSrc, ALUSrc,<br>output RegWrite, Jump, Jalr,<br>output [1:0] ImmSrc,<br>output [3:0] ALUControl |
| **module data_mem** | input clk, wr_en,<br>input [2:0] funct3,<br>input [ADDR_WIDTH-1:0] wr_addr, wr_data,<br>output reg [DATA_WIDTH-1:0] rd_data_mem |
| | |

| | |
|---|---|
| **module datapath** | input clk, reset,<br>input [1:0] ResultSrc,<br>input PCSrc, ALUSrc,<br>input RegWrite,<br>input [1:0] ImmSrc,<br>input [3:0] ALUControl,<br>input Jalr,<br>output Zero,ALUR31,<br>output [31:0] PC,<br>input [31:0] Instr,<br>output [31:0] Mem_WrAddr, Mem_WrData,<br>input [31:0] ReadData,<br>output [31:0] Result |
| **module imm_extend** | input [31:7] instr,<br>input [ 1:0] immsrc,<br>output reg [31:0] immext |
| **module instr_mem** | input [ADDR_WIDTH-1:0] instr_addr,<br>output [DATA_WIDTH-1:0] instr |
| **module main_decoder** | input [6:0] op,<br>input [2:0] funct3,<br>input Zero,ALUR31,<br>output [1:0] ResultSrc,<br>output MemWrite, Branch, ALUSrc,<br>output RegWrite, Jump,Jalr,<br>output [1:0] ImmSrc,<br>output [1:0] ALUOp |
| **module reg_file** | input clk,<br>input wr_en,<br>input [4:0] rd_addr1, rd_addr2, wr_addr,<br>input [DATA_WIDTH-1:0] wr_data,<br>output [DATA_WIDTH-1:0] rd_data1, rd_data2 |
| **module riscv_cpu** | input clk, reset,<br>output [31:0] PC,<br>input [31:0] Instr,<br>output MemWrite, |

| | |
|---|---|
| | output [31:0] Mem_WrAddr, Mem_WrData,<br>input [31:0] ReadData, |