Design Documentation: RISC-V Pipelined Processor Implementation

June 25, 2025

1. Introduction

This document describes the RTL design and implementation of a multi-layer feedforward neural network for MNIST digit classification using Verilog HDL. The system consists of four fully connected layers implemented as separate modules, along with ReLU activation and max-finding logic. The design is suitable for FPGA-based deployment and supports fixed-point inference for resource-efficient computation.

The RISC-V instruction set architecture (ISA) provides a clean and extensible platform for modern processor design. This project implements a 5-stage pipelined RISC-V processor in Verilog, supporting basic integer instructions including branches and jumps. The processor is designed for simulation and FPGA prototyping.

The pipelined architecture enhances throughput by allowing multiple instructions to be executed simultaneously across different stages of the pipeline. This design follows the classic five-stage MIPS-inspired pipeline structure, dividing instruction execution into clearly separated fetch, decode, execute, memory, and write-back stages. Each stage is implemented as a modular Verilog component, allowing for ease of debugging and future extension.

This processor supports a basic subset of RISC-V instructions, including arithmetic operations, immediate variants, load/store memory access, and control flow instructions such as unconditional jumps (JAL) and conditional branches (BEQ, BNE). Forwarding and hazard detection logic ensures the correct order of execution even in the presence of data dependencies, while maintaining the performance advantage of pipelining.

This document provides a breakdown of each module, describes simulation strategies, and includes recommendations for enhancement. The design is suitable for implementation on FPGA platforms such as Xilinx and Intel development boards, and can serve as a foundation for a full RISC-V CPU.

2. Pipeline Architecture Overview

The pipeline consists of five stages:

- **IF (Instruction Fetch)**: The instruction fetch (IF) stage is responsible for supplying the processor with the next instruction to execute. It uses the program counter (PC) to address the instruction memory and retrieve the corresponding instruction word. Typically, the PC is incremented by four to point to the next sequential instruction, as RISC-V instructions are 32 bits wide. However, if a branch or jump instruction alters the control flow, the PC is updated with the appropriate target address. The IF stage also forwards the current PC value alongside the instruction to the next pipeline stage for use in later computations, such as branch resolution or return address calculation. This stage is critical in maintaining a smooth flow of instructions into the pipeline, and any stalls or flushes due to hazards must be managed efficiently to avoid performance degradation.

- **ID (Instruction Decode)**: The instruction decode (ID) stage interprets the raw instruction bits received from the fetch stage and prepares the necessary operands and control signals for execution. It extracts fields such as the opcode, function codes, and register indices from the instruction format. Using these fields, it accesses the register file to read the contents of the source registers specified in the instruction. If the instruction includes an immediate operand, this value is sign-extended or zero-extended as appropriate by the extender module. The ID stage also initiates the generation of control signals through dedicated decoder modules, which determine the behavior of downstream units such as the ALU, memory, and write-back logic. Additionally, this stage incorporates early hazard detection logic to signal stalls or flushes when necessary, ensuring data dependencies and control hazards are handled correctly in the subsequent cycles.

- **EX (Execute)**: The execute (EX) stage is responsible for performing arithmetic, logical, and comparison operations using the Arithmetic Logic Unit (ALU). Operands for the ALU are selected from the register file outputs or immediate values, typically routed through multiplexers that support operand forwarding. This stage also handles computation of branch target addresses by adding the program counter to a sign-extended immediate value. For conditional branch instructions, the EX stage evaluates the specified condition (e.g., equality for BEQ) and generates a branch decision signal. In cases where data hazards are detected, forwarding paths ensure that the most recent operand values are used in computations, preventing incorrect results and minimizing pipeline stalls. Overall, this stage plays a crucial role in determining control flow and executing the core computational logic of the processor.

- **MEM (Memory Access)**: The memory access (MEM) stage interfaces with the data memory to support load and store instructions. For load operations, the stage reads a 32-bit word from the computed memory address and forwards it to the write-back stage. In the case of store instructions, it writes data from the source register into the designated memory address. The address used for both load and store operations is typically calculated during the execute stage and passed along via pipeline registers. This stage must account for memory latency and alignment, although the current design assumes single-cycle memory access for simplicity. For instructions that do not require memory access, the MEM stage simply passes through the ALU result to the next pipeline stage unaltered. Its correct functioning is essential to ensure data consistency between the processor and memory system.

- **WB (Write Back)**: The write-back (WB) stage finalizes instruction execution by writing the result back into the register file. Depending on the instruction type, this result is either the output of the ALU (for arithmetic and logic operations) or the data retrieved from memory (for load instructions). The selection between these two sources is controlled by the 'MemToReg' signal, generated during instruction decoding and propagated through the pipeline. The destination register address is also preserved across stages to ensure accurate write-back. Proper operation of the WB stage is essential to maintain data correctness and support subsequent instructions that may depend on the updated register values. Although it is the final stage, its timing and integration with hazard resolution mechanisms play a critical role in achieving overall pipeline stability and correctness.

3. Design Modules

The processor is implemented using modular RTL design, with a clear separation between datapath and control logic. Each core component of the pipeline is captured in a dedicated

Verilog file, allowing for scalability and ease of testing. Below is a focused summary of the most critical modules used in the implementation:

3.1 Fetch Unit

The instruction fetch logic is implemented across ‘pc_module.v‘, ‘pc_adder.v‘, and ‘instruction_memory.v‘. The ‘pc_module‘ handles sequential PC updates as well as branch/jump redirection. The ‘pc_adder‘ computes the incremented PC value (PC + 4), and the instruction memory provides the instruction corresponding to the current PC. These components are integrated in ‘fetch_cycle.v‘, which bundles the fetch logic into a single stage interface. This unit outputs both the instruction and the PC value to the next pipeline stage.

3.2 Decode Unit

Instruction decoding and register file access are managed by ‘controller.v‘, ‘maindec.v‘, ‘aludec.v‘, and ‘regfile.v‘. The ‘controller‘ module determines the control signals based on instruction opcode fields, while ‘maindec‘ and ‘aludec‘ specialize in high-level and ALU-specific decoding respectively. The ‘regfile‘ allows simultaneous reading of two source registers and supports synchronous write-back. The ‘extend.v‘ module is responsible for extracting and sign-extending immediate values to a 32-bit format.

3.3 Execution Unit

The arithmetic and logical operations are performed in the ‘alu.v‘ module, guided by ALU control signals derived from ‘aludec.v‘. Operand selection is achieved using multiplexers (‘mux.v‘, ‘mux2.v‘, and ‘mux3.v‘), allowing flexibility in choosing between register data, immediate values, or forwarded results from later pipeline stages. This stage also evaluates branch conditions and calculates memory or branch target addresses.

3.4 Memory Unit

The data memory interface is implemented in ‘dmem.v‘, while ‘imem.v‘ provides the instruction memory logic for testing. These modules simulate single-cycle read and write behavior. The memory address, computed in the execution stage, is passed into this stage via pipeline registers. The write enable signal and data inputs are driven by control logic from the decode stage.

3.5 Write-Back Logic

In the final stage of the pipeline, results are selected using ‘mux4.v‘, which chooses between ALU and memory outputs based on the ‘MemToReg‘ signal. This result is then written to the destination register in ‘regfile.v‘. The correctness of the write-back path is crucial to ensure the availability of updated data to future instructions in the pipeline.

3.6 Hazard and Pipeline Control

The ‘hazardunit.v‘ module is responsible for identifying and managing data hazards. It introduces stalls where necessary and enables operand forwarding paths when dependencies can be resolved dynamically. Pipeline registers such as ‘if_id.v‘, ‘id.v‘, ‘iex_imem.v‘, and ‘imem_iw.v‘ preserve control and data signals across cycles. Synchronization is maintained using ‘flopr.v‘ and ‘flopenr.v‘ to register outputs conditionally or unconditionally based on control flow.

3.7 Top-Level Integration

The ‘top_module.v‘ serves as the central integration point, instantiating the datapath and controller units and wiring them with clock, reset, and memory interfaces. It exposes critical test signals such as data memory address and write data, making it suitable for simulation and waveform inspection. This top module encapsulates the full functionality of the processor and provides the interface for testbenches and FPGA-level deployment.

4. Forwarding and Hazard Control

Forwarding and hazard resolution are key components in preserving the correctness and efficiency of pipelined execution. The design employs a dedicated hazardunit.v module that detects both data and control hazards and responds accordingly to maintain correct program behavior.

Data hazards, such as those caused by dependencies between source and destination registers of overlapping instructions, are resolved primarily through forwarding. Forwarding logic identifies when a result from a later stage (e.g., EX/MEM or MEM/WB) needs to be redirected to an earlier stage (e.g., ID/EX) and selects the correct operand source using multiplexers. This eliminates many unnecessary stalls and allows dependent instructions to execute without delay when safe.

In cases where forwarding is not sufficient—for example, when a load instruction is followed immediately by an instruction that uses its result—a stall is introduced. The hazard unit raises a stall signal, and the pipeline register between the IF and ID stages is frozen for one cycle, delaying subsequent instructions to allow the correct data to become available.

Control hazards occur due to branching instructions, where the actual control flow is not known until the branch is resolved in the EX stage. To handle this, a one-cycle delay is introduced using flushing logic. If a branch is taken, the instruction in the ID stage is invalidated, and the PC is updated with the branch target. This flushing mechanism ensures that incorrect instructions do not reach later stages.

Overall, the hazard unit is a central component that dynamically manages inter-instruction dependencies and control flow redirection, ensuring the pipeline operates without violating data or control consistency.

5. Testbench and Verification

A comprehensive suite of module-level and top-level testbenches was developed to verify the functional correctness of the RISC-V pipelined processor design. Each major module—including control logic, datapath components, forwarding logic, memory, and inter-stage pipeline registers—was tested individually to ensure timing behavior and signal interactions were correct.

Testbenches were implemented using SystemVerilog, with ‘initial‘ blocks providing clock-/reset generation and test stimuli. Assertions and ‘display‘ statements were used to monitor key signals such as register outputs, program counter updates, memory read/write actions, and stall or flush behavior across cycles.

The following files were used for simulation:

- `tb_top_module.v` – End-to-end verification of the integrated processor pipeline
- `tb_datapath.v`, `tb_controller.v` – Test control/data logic interactions
- `tb_hazardunit.v` – Verify stall, flush, and forwarding logic under data/control hazards
- `tb_alu.v`, `tb_extend.v`, `tb_regfile.v` – Validate arithmetic, decoding, and register behavior

- `tb_fetch_cycle.sv`, `tb_instruction_memory.sv`, `tb_pc_module.sv` – Verify instruction fetch mechanisms

The ‘memfile.hex‘ file was used to preload instruction memory for system-level testing. The simulation process confirmed that the processor handled ALU operations, branching, and memory transactions as expected, with correct hazard handling and data forwarding behavior observed during test execution.

6. Conclusion

This project successfully demonstrates the design and verification of a modular, five-stage pipelined RISC-V processor using Verilog. The implementation covers key architectural concepts such as instruction fetch, decode, execution, memory access, and write-back, along with hazard detection and forwarding mechanisms. Each pipeline stage was modeled as an independent RTL module to ensure clarity and extensibility.

The design includes robust handling of data hazards through forwarding and stalling, and introduces a simple yet effective control hazard solution using flush logic. Module-level and system-level simulation testbenches confirmed the functional correctness of the processor under a variety of instruction sequences. With a clear structure and working implementation, this processor serves as a solid foundation for further enhancements such as support for CSRs, exceptions, pipelined memory, or cache integration.

The modular structure, test-driven development approach, and simulation-based verification make this implementation both educational and practically useful for FPGA-based deployment and academic research.