Design and Implementation of an RV64IMF Pipelined Processor with Integrated FPU

Author Name Department / Institution

September 29, 2025

Contents

1 Objectives

The primary objectives of this project are:

- Implement an RV64IMF processor core in Verilog that can be synthesized for an FPGA.
- Support the RISBUJ instruction formats and the course-provided RISC-V instruction subset (excluding system call and breakpoint instructions).
- Provide basic hazard management: data forwarding, pipeline stalls for load-use and long-latency operations, and simple branch prediction with recovery.
- Verify functionality through simulation and demonstrate operation on an FPGA board.
- Develop the processor in three phases: (1) functional non-pipelined core with FPU, (2) add a 5-stage pipeline with integer hazard handling, (3) integrate a non-blocking/pipelined FPU into the main pipeline.

2 Project Specifications

2.1 ISA and Instruction Support

The processor implements the RV64 base integer ISA plus the following extensions:

- **I**: RV64 integer base instructions (load/store, arithmetic, logical, control flow).
- **M**: Integer multiply and divide.
- **F**: Single-precision floating point (IEEE-754) operations — implemented via an FPU unit integrated into the pipeline.

All R-type, I-type, S-type, B-type, U-type, and J-type instruction encodings described by the RISBU formats (as provided in the class RISC-V card) will be implemented, except `ecall` and `ebreak` which are intentionally omitted.

3 Architecture Overview

Include an architectural block diagram here describing the major components:

- Instruction fetch unit (PC generation, instruction memory interface, branch prediction unit).
- Register file (integer and floating-point register files; note: F extension uses separate FP registers but can be architected as a single register file with type handling — decide in implementation).
- Integer ALU with multiply/divide unit (M extension); optionally a separate multiplier pipeline.
- Floating-Point Unit (FPU): single-precision adder/subtractor, multiplier, divide/sqrt (optional) and conversion units.
- Data memory interface (AXI-lite or simple synchronous SRAM interface depending on FPGA board).
- Pipeline control and hazard detection unit (forwarding paths, stall logic).

4 Pipeline Design

This project follows a staged pipeline introduction. The baseline pipeline will be the classical 5-stage RISC pipeline:

1. Fetch (IF)
2. Decode/Register Read (ID)
3. Execute/ALU (EX)
4. Memory Access (MEM)
5. Write Back (WB)

For floating-point operations the FPU may have a multi-cycle or pipelined implementation. During Phase 2 FP instructions will not be fully pipelined into the main pipeline; instead they will be handled via a blocking (stalling) FPU interface. Phase 3 will integrate the FPU such that FP instructions can flow through the pipeline with appropriate forwarding and hazard handling.

4.1 Hazard Management

Data Hazards: Implement standard forwarding paths from ALU/MEM/WB outputs to ALU inputs in ID/EX as required. When forwarding cannot resolve hazards (e.g., load-use hazards), the control logic will insert a bubble (stall).

Control Hazards: A simple static branch predictor (e.g., predict-not-taken or a 1-bit bimodal predictor) will be implemented in the IF/ID stage to reduce branch penalties. On misprediction the pipeline will flush incorrectly fetched instructions and update the PC.

5 Floating-Point Unit (FPU)

5.1 Design Goals

- Support single-precision (IEEE-754) arithmetic: add/sub, mul, fma (if feasible), comparisons, loads/stores, and conversions between integer and float.
- Provide interfaces for exception flags and rounding modes (to the extent required by the course specification).
- Offer both a simple single-cycle (for early phases) and a more efficient pipelined or multi-cycle implementation for final integration.

5.2 Integration Strategy

During Phase 1, the FPU will be implemented as a functional unit callable from the EX stage but may take multiple cycles to complete, blocking the pipeline (stalling). During Phase 3 the FPU will present a pipelined interface with result-bypassing so that later instructions can make use of results via forwarding without unnecessary stalls.

6 Implementation Plan (Phases)

6.1 Phase 1: Functional Processor without Pipelining (with FPU)

- Implement a single-cycle or multi-cycle non-pipelined processor (control simpler to verify).
- Implement FPU as a multi-cycle combinational/sequential unit accessed from EX stage (pipeline stalls entire processor while FPU operates).
- Develop Verilog modules: instruction memory (ROM), datapath, control unit, register files, simple bus to external memory.
- Deliverables: RTL code, simulation testbenches for integer and FP instruction sets, synthesis report on FPGA.

6.2 Phase 2: Pipeline Implementation without Pipelined FP

- Convert datapath to a 5-stage pipelined design with registers between stages.
- Implement data forwarding and hazard detection for integer instructions.
- Integrate branch prediction and misprediction recovery logic.
- FPU remains non-pipelined; FP instructions will cause the pipeline to stall until FPU finishes.
- Deliverables: pipelined RTL, updated testbench, performance measurements (CPI estimates), FPGA bitstream.

6.3 Phase 3: FP Integration into Pipeline

- Rework FPU into a pipelined or non-blocking multi-cycle unit with result tagging so that multiple FP operations can be in-flight.
- Add forwarding paths for FP results into integer and FP instructions as required.
- Handle FP-specific exceptions and rounding modes in control/status registers.

- Deliverables: fully integrated RTL, extended testbench including concurrent integer+FP sequences, final FPGA bitstream, performance evaluation.

7 Verification and Testing

7.1 Simulation

- Unit-level testbenches for ALU, FPU, register files, and memory interfaces.
- Instruction-level tests using assembly programs that exercise all implemented instructions (edge cases, saturations, corner cases for FP like NaN/Inf/denormals if supported).
- Automated regression and self-checking testbenches where simulation results are compared against a reference model (e.g., spike or a golden C model).