RISC-V Implementation on FPGA

Design, Implementation and Evaluation

Chitsidzo Varaidzo Nemazuwa, Kanaya Nisa Ozora, and Nawal Salama
School of Computation, Information and Technology, Technical University of Munich

February 20, 2026

Abstract — This report presents the design, implementation, and evaluation of a RISC-V processor deployed on an FPGA platform. The focus lies on architectural decisions, memory organization, timing performance, and debugging methodology.

1 Design Overview

1.1 Memory Architecture

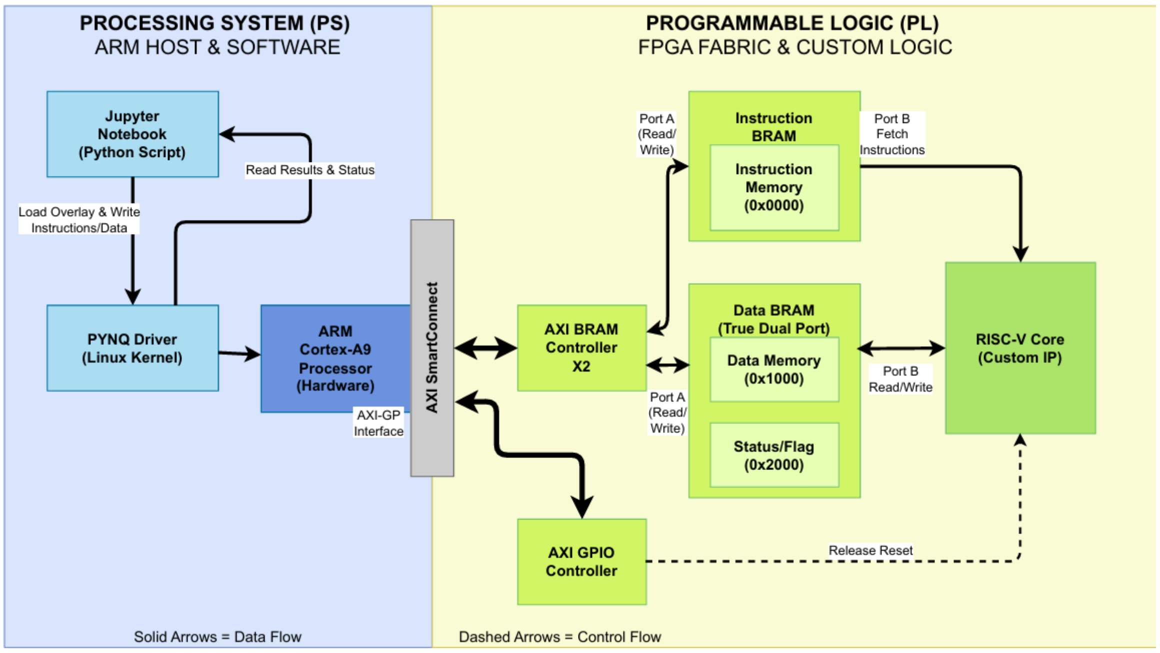

Figure 1 Separate Instruction and Data BRAM Layout

GITHUB LINK TO THE REPOSITORY: https://github.com/chits-nema/FPGA_RISC_V_PP

We decided to implement the individual BRAMs design schematic on our FPGA implementation. When choosing which design to use, we thought of the complexities surrounding port contention. A RISC-V core needs to fetch an instruction and perform a load / store in the same clock cycle; this requires two simultaneous memory accesses.

Using the individual BRAM design, the instruction BRAM and data BRAM are physically separate resources with their own independent port pairs. This means our RISC-V core can fetch instructions from the instruction BRAM on Port B while also doing a load/store on the Data BRAM's Port B without concerns of port contention. This design maps to the Harvard architecture separation that works well with RISC-V pipeline processors. Using separated BRAMs also avoid most stalls from memory access conflicts.

1.2 Behavioral Simulation

Behavioral simulations were performed using Vivado to verify instruction execution, hazard handling, and branch behavior prior to hardware deployment.

2 FPGA Implementation Analysis

2.1 Resource Utilization

3 Resource Utilization (Hierarchical)

Table 1 presents the hierarchical resource utilization of the major modules in the design, including the RISC-V core, AXI infrastructure, and memory blocks. The reported values include LUTs, flip-flops (registers), and Block RAM (BRAM) tiles.

Table 1 Hierarchical Resource Utilization

Module	LUTs	Registers	BRAM Tiles
Top Level (rv_pynq_bd_wrapper)	7078	7653	6
AXI SmartConnect (axi_smc)	5087	5634	0
RISC-V Core (rv_pl_wrapper_0)	1481	1522	0
AXI BRAM Ctrl 0	188	197	0
AXI BRAM Ctrl 1	193	200	0
AXI GPIO	35	36	0
Instruction BRAM (blk_mem_gen_1)	10	12	4
Data BRAM (blk_mem_gen_0)	10	12	2
Processor System Reset	19	40	0
Processing System (PL logic only)	55	0	0

3.1 Pipeline-Level Utilization Analysis

Within the RISC-V core, the resource consumption can be attributed primarily to the following pipeline components:

- **Fetch Stage** – Program Counter (PC) logic, instruction memory interface, and branch redirection control.
- **Decode Stage** – Register file, immediate generation, control signal generation, and hazard detection inputs.
- **Execute Stage** – Arithmetic Logic Unit (ALU), comparison logic for branches, forwarding multiplexers.
- **Memory Stage** – Data memory interface and store-data selection logic.
- **Hazard and Forwarding Unit** – Stall detection, forwarding multiplexers, branch flush control.

3.2 Analysis of Resource Distribution

From Table 1, it is evident that the **AXI SmartConnect** consumes the largest portion of LUTs and registers in the overall design. This is expected because the AXI interconnect implements arbitration, routing, buffering, and protocol management for multiple AXI master–slave transactions. These functions require significant control logic and state machines.

Focusing specifically on the **RISC-V core**, the majority of logic usage is concentrated in the **Execute stage and Hazard/Forwarding logic**. This is due to:

- The ALU arithmetic and comparison circuitry.
- Wide multiplexers required for operand forwarding.
- Branch decision logic.
- Stall and flush control mechanisms.

The Decode stage also contributes significantly because of the register file implementation and control signal decoding.

In contrast, the Fetch and Memory stages consume fewer LUTs, as they primarily consist of address registers and memory interfacing logic. The BRAM usage is entirely attributed to the instruction and data memories, with 4 tiles allocated for instruction memory and 2 tiles for data memory.

3.3 Conclusion

The Execute stage (including hazard and forwarding logic) is the most logic-intensive component of the RISC-V pipeline. This is expected in pipelined processor designs, as arithmetic operations, operand selection, and branch resolution require the most combinational circuitry. Additionally, the AXI interconnect dominates total resource usage at the system level due to its complex protocol handling.

4 Module Description and Implementation

4.1 RISC-V Core

The processor implements a pipelined RISC-V architecture with hazard detection and forwarding mechanisms. Special handling was introduced to account for the two-cycle BRAM latency.

4.2 Block Design

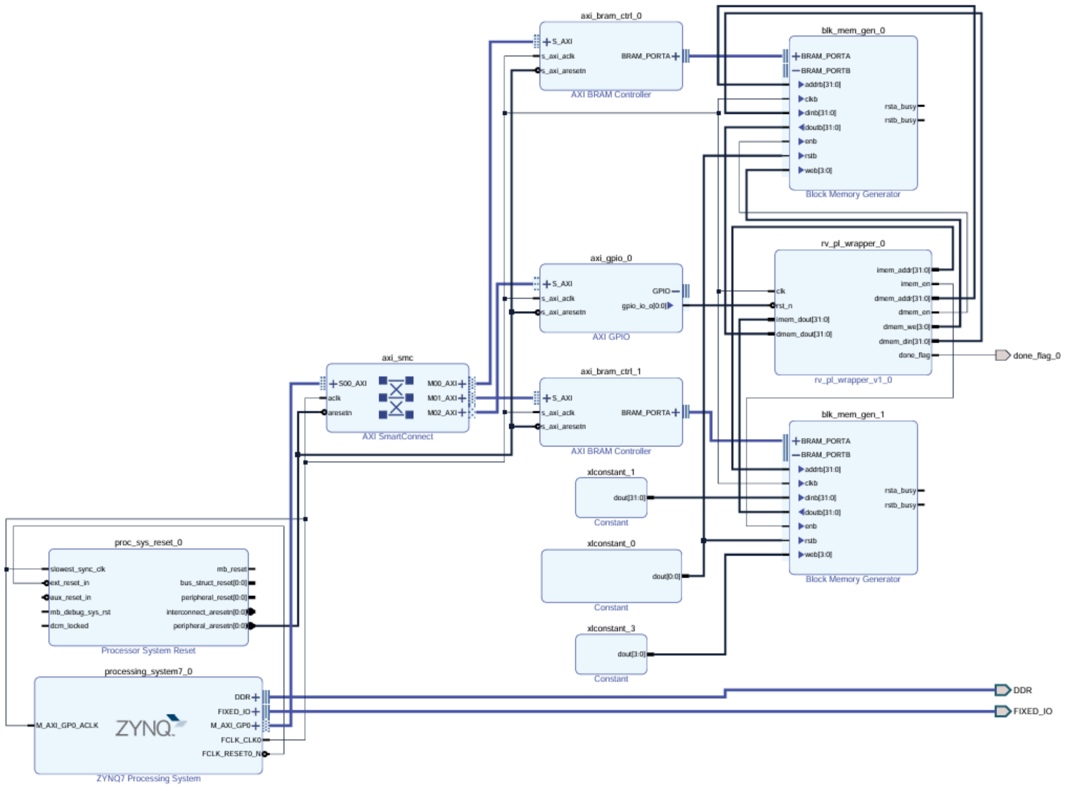

Figure 2 Block diagram of the FPGA-based RISC-V system

The final block design integrates the Zynq Processing System with the programmable logic components through AXI Interconnects. The design uses the following IP blocks:

- RISC-V core (rv_pl_wrapper_0)
- Instruction BRAM (blk_mem_gen_1)
- Data BRAM (blk_mem_gen_0)
- AXI SmartConnect
- AXI GPIO

The key connections are summarized below:

- Processing System to AXI SmartConnect: The processing_system7_0 block acts as an AXI master via the M_AXI_GP0 interface. It connects to the AXI SmartConnect, which distributes AXI transactions to multiple slave peripherals in the programmable logic. This enables the PS to initialize memory, control GPIO, and debug the system.
- AXI SmartConnect to AXI BRAM Controllers: The SmartConnect routes AXI transactions from the PS to two AXI BRAM controllers (axi_bram_ctrl_0 and axi_bram_ctrl_1). These controllers translate AXI protocol transactions into native BRAM control signals.
- AXI BRAM Controllers to BRAM Blocks: Each AXI BRAM controller is connected to a dedicated Block Memory Generator IP:

- RISC-V Core to Instruction BRAM: The RISC-V core (rv_pl_wrapper_0) connects directly to the Instruction BRAM using native memory signals (imem_addr, imem_dout). This path is used exclusively for instruction fetch operations.
- RISC-V Core to Data BRAM: The core also connects directly to the Data BRAM through signals such as dmem_addr, dmem_din, dmem_dout, and dmem_we. This interface supports load and store operations during program execution.
- Clock Distribution: The clock signal (FCLK_CLK0) generated by the Processing System is distributed to the RISC-V core, AXI SmartConnect, AXI BRAM controllers, and GPIO module. This ensures synchronous operation across the entire design.
- Reset Synchronization: The proc_sys_reset_0 block generates synchronized reset signals for the programmable logic. This guarantees proper initialization and avoids metastability issues during startup.
- AXI GPIO Connection: The axi_gpio_0 module connects to the AXI SmartConnect and provides a controllable interface between the PS and the RISC-V core. It is used for reset control and simple status signaling.
- Done Flag Output: The done_flag signal from the RISC-V wrapper is exposed externally to indicate successful program completion, facilitating debugging and verification.

4.3 Processing System (PS)

The Processing System (PS) operates as an AXI master and controls the programmable logic through memory-mapped interfaces. Using the PYNQ Overlay framework, the PS loads the FPGA bitstream and accesses AXI peripherals such as BRAM controllers and GPIO modules.

The PS programs the instruction and data BRAMs by writing directly to their memory-mapped addresses via AXI. Before modifying memory contents, the RISC-V core is held in reset using an AXI GPIO signal to ensure safe memory updates. After initialization, the reset is released to start execution from address zero.

During runtime, the PS monitors program completion by polling a status flag (0x2000) in data memory. Once execution completes, the PS reads back the results from BRAM and verifies correctness against a software-generated reference result. This structure enables full software-driven control of program loading, execution management, and debugging.

The Python notebook utilizes a bubble sort algorithm to test the implemented FPGA project. 32 integers generated by random are sorted through a set of assembly instructions as follows:

```
.text
.globl _start

_start:
    # Load base address of array into a0
    lui a0, 0x1          # a0 = 0x1000 (data RAM base)
    addi a0, a0, 0x40    # a0 = 0x1040 (array start)

    # Initialize outer loop counter: n-1 = 31 passes
    addi t0, zero, 31    # t0 = outer loop counter (31 passes)

outer_loop:
    # Check if outer loop is done
    beq t0, zero, done

    # Initialize inner loop counter
    addi t1, zero, 31    # t1 = inner loop counter
    # Reset array pointer for inner loop
    addi a1, a0, 0       # a1 = current position in array

inner_loop:
    # Check if inner loop is done
    beq t1, zero, outer_continue

    # Load two adjacent elements
    lw t2, 0(a1)         # t2 = array[i]
    lw t3, 4(a1)         # t3 = array[i+1]

    # Compare array[i] and array[i+1]
    # If array[i] <= array[i+1], skip swap
    slt t4, t3, t2       # t4 = 1 if array[i+1] < array[i], else 0
    beq t4, zero, no_swap

    # Swap: array[i] and array[i+1]
    sw t3, 0(a1)         # array[i] = t3 (was array[i+1])
    sw t2, 4(a1)         # array[i+1] = t2 (was array[i])

no_swap:
    # Move to next pair
    addi a1, a1, 4       # Increment pointer by 4 bytes
    addi t1, t1, -1      # Decrement inner loop counter
    jal zero, inner_loop # Jump back to inner loop

outer_continue:
    # Decrement outer loop counter
    addi t0, t0, -1
    jal zero, outer_loop # Jump back to outer loop

done:
    # Write status flag with the MAGIC NUMBER at 0x2000
    lui a2, 0x2              # a2 = 0x2000
    lui a3, 0xDEADF          # a3 = 0xDEADF000
    addi a3, a3, -273        # a3 = 0xDEADBEEF
    sw a3, 0(a2)             # Write DEADBEEF to 0x2000

    # Infinite loop
    jal zero, done
```

Once all the integers are sorted, PS writes DEADBEEF to 0x2000 in DRAM. Our PS also utilizes polling, to check if the status flag has been written to every second.

5 Testing and Debugging

5.1 Simulation-Based Testing

The RISC-V IP core was verified using a dedicated behavioral testbench (`rv_pl_behavior_simulation_tb.v`). The objective was to validate pipeline behavior, forwarding logic, hazard detection, and branch handling before deployment on hardware.

Six test categories were implemented:

1. **RAW Forwarding** This test verified correct data forwarding between pipeline stages. **Result: Passed.**
2. **Load-Use Hazard** This test evaluated whether a stall or forwarding mechanism correctly handled the classic `lw` dependency case. **Result: Failed initially due to incorrect hazard handling.**
3. **BEQ Taken Loop** This test checked proper branch resolution and PC update when a branch is taken repeatedly. **Result: Failed.**
4. **Branch Slip Detection** This test examined whether instructions following a taken branch were correctly flushed. **Result: Partial success.**
5. **LW → SW Forwarding** This test verified forwarding of loaded data directly into a store instruction. **Result: Passed.**
6. **BEQ Not Taken** This validated correct PC progression when the branch condition is false. **Result: Passed.**

Pipeline Debugging and Observed Issues

During initial simulation runs, several structural and timing-related issues were identified through waveform inspection and debug print statements embedded in the testbench.

Program Counter Misalignment When monitoring the PC values across the Decode (D), Execute (E), and Memory (M) stages, it was observed that the PC appeared significantly behind the expected instruction flow. The PC updates were not aligned with the instruction currently being evaluated in later pipeline stages.

This misalignment indicated that pipeline registers were not correctly synchronized with memory latency. Because the BRAM introduces a two-cycle latency, the instruction fetch stage was effectively delayed, causing downstream stages to operate on stale PC values.

Memory Write Signal Not Asserted Another critical issue was that the memory write enable signal was never asserted during store operations. Waveform analysis showed that the control signal responsible for driving the write enable remained low even when a store instruction reached the Memory stage.

This was traced back to incorrect timing between instruction decoding and control signal propagation. The control signals were generated correctly in the Decode stage but were not properly latched through the pipeline registers. As a result, by the time the instruction reached the Memory stage, the write-enable signal had been lost.

Delayed Branch Resolution Branch behavior was also examined using waveform inspection and debug messages. It was observed that branch decisions were resolved several cycles later than expected. The PC redirection occurred only after the branch instruction had progressed further down the pipeline.

This delay was consistent with the two-cycle BRAM latency, which affected both instruction fetch and branch target computation. Because the branch decision relied on values not yet fully stabilized in the pipeline, incorrect instructions were allowed to propagate before being flushed.

Pipeline Stage Offset Across multiple tests, it became evident that instructions were effectively “two stages behind” due to memory latency. The pipeline was originally designed under the assumption of single-cycle memory access. However, the BRAM IP core introduces a two-cycle read delay, which required additional synchronization logic.

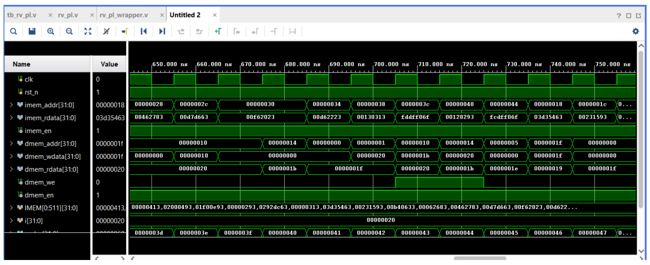

Figure 3 Waveform

Design Corrections

To resolve these issues:

- Pipeline registers were carefully realigned to account for BRAM latency.
- Hazard detection logic was modified to insert appropriate stalls for `lw` dependencies.
- Branch handling logic was updated to correctly flush invalid instructions when a branch is taken.
- Control signals, particularly memory write enable, were explicitly pipelined to ensure correct propagation into the Memory stage.

After incorporating these modifications, the processor correctly handled load-use hazards and branch timing under the two-cycle BRAM constraint.

We tested the program on a testbench. The following is the waveform of that testbench:

According to the testbench, our processor passed the tests programmed. However, when we ran it on the board, we encountered several issues such that the bubble sort does not sort properly. This can be contributed to many things, e.g., a mismatch of memory mapping, and our fixes for BRAM latency still not working.

6 Conclusion

The implementation demonstrates a functional FPGA-based RISC-V processor with separated instruction and data memories. Key challenges included hazard handling and BRAM latency management. After iterative simulation and debugging, a stable and timing-optimized system was achieved.

7 Contributions

All three of us worked equally between the block designs, modification of the RTL code and testing and debugging of the RISC-V Processor.

8 References

References