Chair of Computer Architecture and Operating Systems
School of Computation, Information and Technology
Technical University of Munich

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

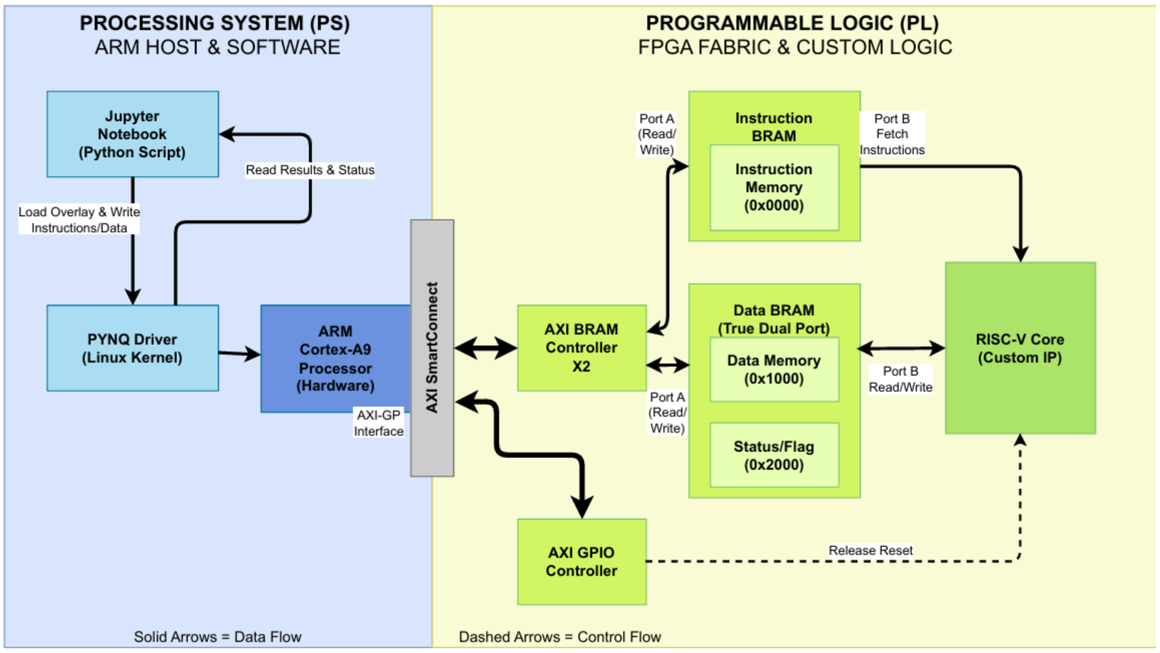


Figure 1 Separate Instruction and Data BRAM Layout

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 Module Description and Implementation

2.1 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.

2.2 Block Design

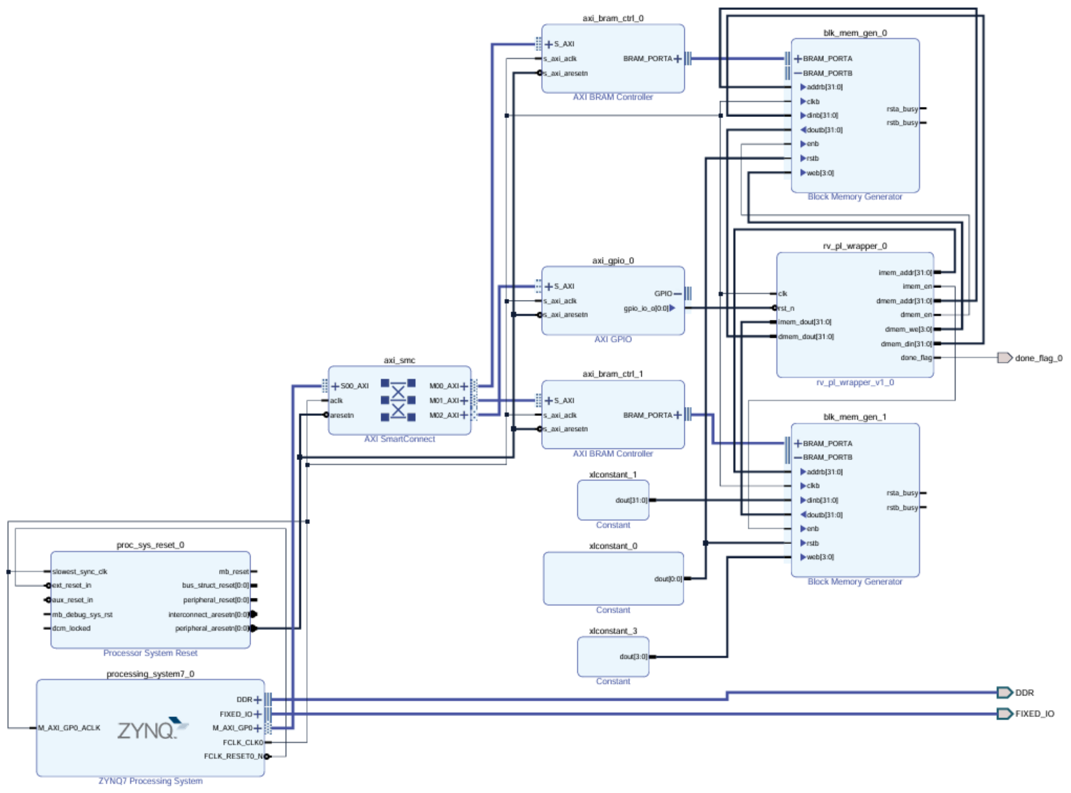


Figure 2 Block diagram of the FPGA-based RISC-V system

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.

2.3 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:

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

3 FPGA Implementation Analysis

3.1 Hierarchical Utilisation Overview

The design was implemented on a Xilinx Zynq-7000 SoC (xc7z020clg400-1) using Vivado 2021.2. The top-level `design_1_wrapper` consumes a total of 5783 LUTs (4982 logic LUTs, 470 LUTRAMs, and 331 SRLs), 5791 flip-flops (FFs), and 8 RAMB36 blocks. The resource breakdown across the major submodules of the design is summarised in Table 1.

Table 1 Hierarchical FPGA Resource Utilisation Summary

Module	Total LUTs	Logic LUTs	LUTRAMs	FFs	RAMB36
`axi_bram_ctrl_0`	170	170	0	184	0
`axi_interconnect_0`	4611	3864	416	4820	0
`blk_mem_gen_0`	10	2	0	0	4
`blk_mem_gen_1`	10	2	0	0	4
`proc_sys_reset_0`	17	16	0	33	0
`processing_system7_0`	0	0	0	0	0
`rv_pl_wrapper_0`	772	728	44	554	0

3.2 RISC-V Pipeline Processor Utilisation

The RISC-V pipelined processor is instantiated as `rv_pl_wrapper_0` and is decomposed into its five pipeline stage registers and supporting units. Table 2 presents the resource utilisation at the submodule level within the processor.

Table 2 RISC-V Pipeline Processor Submodule Utilisation

Instance	Stage / Function	Total LUTs	Logic LUTs	LUTRAMs	FFs
PLR1 (IF/ID register)	Fetch → Decode	118	118	0	93
PLR2 (Decode/Execute register)	Decode → Execute	520	520	0	186
PLR3 (Execute/Memory register)	Execute → Memory	54	54	0	105
PLR4 (Memory/Writeback register)	Memory → Writeback	36	36	0	72
RF (Register File)	Decode stage (read/write)	44	0	44	0
`hazard_unit`	Pipeline control	1	1	0	1
`main_alu`	Execute stage (ALU)	0	0	0	0
`pc_reg`	Program Counter	0	0	0	32
`pc_r_add` / `pc_target_add`	PC adders	1	1	0	0
(rv_pl top-level)	Pipeline control logic	0	0	0	64

3.3 Analysis: Which Stage Consumes the Most Logic?

The Decode-to-Execute pipeline register (PLR2) is by far the most resource-intensive stage, consuming **520 logic LUTs** and **186 FFs** accounting for approximately 67% of all logic LUTs used within the processor core and 34% of its flip-flops. Several architectural factors explain this dominance:

Wide Datapath Forwarding and Control Muxes

The decode/execute boundary is where all control signals are generated and latched. In a RV32I implementation this includes the ALU operation selector, immediate value selection (I-, S-, B-, U-, J-type immediates), source/destination register addresses, memory access type, writeback mux select, and branch/jump decision logic. Each of these multiplexers and their associated decode logic maps into LUTs, and because many of these signals are 32 bits wide the combinational fan-in at this boundary is high.

Immediate Extension Logic

RISC-V defines five distinct immediate formats. The sign-extension and bit-rearrangement logic for these formats must be resolved in the decode stage before the values can be registered and passed to the execute stage. This produces a significant combinational logic cone that maps into the LUTs attributed to PLR2.

Branch Condition and ALU Control Decoding

The ALU control signals determining whether the execution unit performs addition, subtraction, logical operations, shifts, or comparisons are decoded here. The relatively large number of RISC-V base integer instructions means the decoder is a wide priority logic block, contributing further to the LUT count.

Contrast With Other Stages

`PLR1` (IF/ID, 118 LUTs / 93 FFs) is smaller because it only latches the instruction word and the current PC; no decoding is performed there. `PLR3` (Execute/Memory, 54 LUTs / 105 FFs) has fewer LUTs but more FFs because it is primarily passing a wide 32-bit result and memory address through to the memory stage with minimal combinational logic. `PLR4` (Memory/Writeback, 36 LUTs / 72 FFs) is the smallest register stage, as it simply selects between the ALU result and the memory read data before writeback. Notably, the `main_alu` shows zero LUTs in this hierarchical report; this is a consequence of Vivado merging the ALU combinational logic across hierarchy boundaries into the LUT cells attributed to `PLR2`, consistent with the note at the bottom of the report that *"the sum of lower-level cells may be larger than their parent cells total, due to cross-hierarchy LUT combining"*.

Register File Implementation

The register file (`RF`) consumes 44 LUTRAMs and zero logic LUTs or FFs, indicating that Vivado inferred the 32-entry $\times$ 32-bit register file using distributed RAM (LUT-based RAM primitives) rather than flip-flops or block RAM. This is the expected and efficient mapping for a small, synchronously-read register file on 7-series devices.

Block RAM Usage

The processor itself uses no RAMB36 blocks. The eight RAMB36 instances in the design are split between `blk_mem_gen_0` and `blk_mem_gen_1` (four each), which provide the instruction and data memory accessible via the AXI BRAM controller, consistent with a Harvard-style memory interface exposed over AXI to the Zynq PS.

In summary, the Decode/Execute stage dominates FPGA logic utilisation because it concentrates the widest and most complex combinational decode, immediate-extension, and control-signal generation logic of the entire pipeline.

3.4 Critical Path Timing Analysis

3.4.1 Path Identification

Following implementation and routing, the critical timing path of the RISC-V pipeline processor was identified using Vivado's *Report Timing Summary* tool. The worst-case setup path is shown in Figure 3. The path is consistent with the execution of a `LW` (load word) instruction and represents one of the most well-known critical paths in pipelined RISC processor design. It originates at a flip-flop in `PLR2` (the Decode/Execute pipeline register), propagates through forwarding mux logic and a 28-bit ripple-carry adder inside the ALU, is registered at `PLR3` (the Execute/Memory pipeline register), and terminates at the setup pin of the `RAMB36E1` data memory primitive within `blk_mem_gen_0`.

3.4.2 The `LW` Instruction Datapath

The `LW` instruction requires the execute stage to compute an effective memory address by adding a base register value to a sign-extended 12-bit immediate offset:

$$\mathrm{addr} = \mathrm{rs1} + \mathrm{imm}_{12} \tag{1}$$

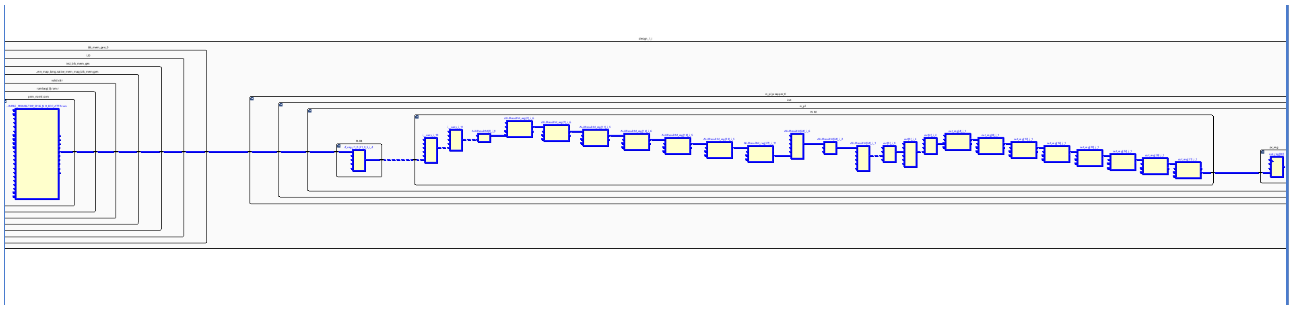


Figure 3 Vivado schematic view of the critical timing path, tracing the `LW` address computation from `PLR2` through the ALU ripple-carry adder to the `RAMB36E1` data memory address input.

This address must be computed, registered into `PLR3`, and simultaneously presented to the BRAM address port before the next rising clock edge. The critical path therefore spans the entire execute stage in a single clock cycle, making it the tightest timing constraint in the design.

The logic cells traversed on this path, reading from the schematic in Figure 3 from top to bottom, are as follows:

1. **Startpoint** A flip-flop in `PLR2` releases the base register value (`rs1`) or a forwarded equivalent on the rising clock edge.

2. **Forwarding mux and operand selection** Three LUT stages (ALUResultM[24]_i_1 LUT6, ALUResultM[24]_i_2 LUT3, ALUResultM[24]_i_6 LUT6) implement the forwarding unit mux, selecting between the register file output and values forwarded from `PLR3` or `PLR4` to resolve data hazards. The selected operand and the sign-extended immediate are then fed into the adder.

3. **Ripple-carry address adder** Seven cascaded `CARRY4` primitives (`ALUResultM_reg[2]_i_6` through `ALUResultM_reg[27]_i_11`) implement the 28-bit ripple-carry addition. Each `CARRY4` propagates carry across four bits, contributing approximately 0.5-0.6 ns per stage on the xc7z020 speed grade -1 device.

4. **Result selection logic** Additional LUT stages (`i__carry_i_11`, `i__carry_i_19`, `out[0]_i_4`, `out[0]_i_9`, `out[4]_i_2`) and further `CARRY4` stages finalise the computed address.

5. **Endpoint** The computed address is registered in `PLR3` and presented to the `ADDRBWRADDR` port of the `RAMB36E1` data memory, which imposes a setup-time requirement before the next clock edge can trigger the memory read.

3.4.3 Bottleneck Identification

The critical path delay is dominated by the **ripple-carry address adder**, not the forwarding mux logic or the BRAM setup time, for the following reasons.

The seven chained `CARRY4` primitives propagate carry serially from the least-significant to the most-significant bit group across 28 bits. At approximately 0.50.6 ns per stage, the carry chain alone contributes an estimated 3.54.2 ns of logic delay - by far the largest single contributor on the path.

The forwarding mux LUTs preceding the adder add two to three levels of LUT delay (approximately 1.01.5 ns). Although non-trivial, this is significantly smaller than the carry chain contribution. Critically, the mux delay is *serialised* with the carry propagation: the forwarding decision must be resolved before the correct operand can be presented to the adder input, meaning both delays accumulate directly on the critical path. This is a direct consequence of the load-use forwarding scenario, where a value produced in the execute stage of one instruction is needed as the base address of a `LW` in the immediately following cycle.

The `RAMB36E1` setup time at the endpoint (approximately 0.51.0 ns) is the smallest individual contributor. It represents a hard deadline imposed by the synchronous BRAM architecture - the address must be stable before the clock edge for the memory read to begin - but it is not the root cause of the timing bottleneck.

3.4.4 Relationship to the Load-Use Hazard

This critical path is intrinsically linked to the **load-use data hazard**, one of the fundamental hazards in a pipelined RISC processor. When a `LW` instruction is immediately followed by an instruction that consumes the loaded value, the pipeline cannot forward the result in time because the memory read does not complete until the end of the memory stage. The hazard unit in this design (`hazard_unit`) detects this condition and inserts a one-cycle stall (bubble) into the pipeline. While this stall correctly resolves the data hazard, it also means that the critical path identified here - the address computation for the `LW` instruction - directly determines the maximum achievable clock frequency of the entire processor.

3.4.5 Implications and Potential Optimisations

The maximum operating frequency f_{max} is bounded by the total critical path delay:

$$f_{\text{max}} = \frac{1}{t_{\text{clk-q}} + t_{\text{fwd\_mux}} + t_{\text{carry}} + t_{\text{routing}} + t_{\text{setup}}} \tag{2}$$

where t_{carry} is the dominant term. Should a higher clock frequency be required, the following optimisations could be considered.

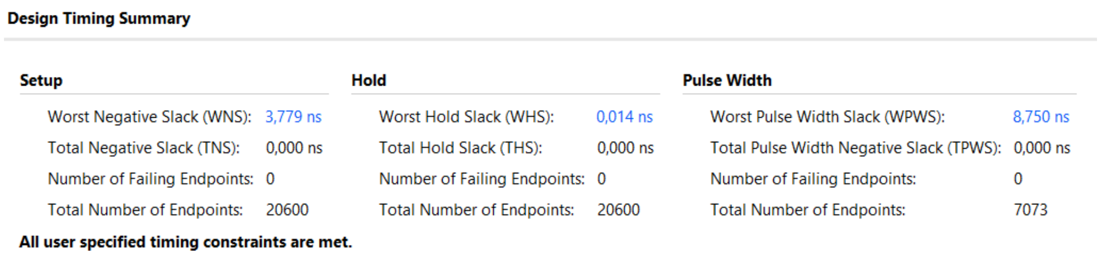
Carry-lookahead or carry-select adder: Restructuring the adder to use a carry-lookahead or carry-select topology would reduce carry propagation depth from $O(n)$ to $O(\log n)$ logic levels. Vivado can infer these structures automatically if the RTL is written without constraining the carry topology, or they can be instantiated explicitly using Xilinx primitive macros.

Forwarding mux retiming: Moving the forwarding operand selection into the preceding clock cycle, so that the correct operand is already registered before the execute stage begins, would decouple the mux delay from the adder delay and remove it from the critical path entirely.

Additional pipeline stage: Splitting the execute stage into two sub-stages - one for operand selection and one for the ALU computation - would halve the combinational depth at the cost of one additional cycle of execution latency, requiring corresponding updates to the hazard detection and forwarding logic.

3.5 Timing Performance

The Design Timing Summary produced by Vivado following place-and-route is reproduced in Figure 4. All user-specified timing constraints are met, with zero failing endpoints across all 20,600 setup paths, 20,600 hold paths, and 7,073 pulse-width checks.



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3,779 ns	Worst Hold Slack (WHS):	0,014 ns	Worst Pulse Width Slack (WPWS):	8,750 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	20600	Total Number of Endpoints:	20600	Total Number of Endpoints:	7073

All user specified timing constraints are met.

Figure 4 Vivado Design Timing Summary showing setup, hold, and pulse-width results after routing on the xc7z020clg400-1 device.

3.5.1 Maximum Operating Frequency

The Worst Negative Slack (WNS) reported by Vivado is +3.779 ns, indicating that the critical path completes with 3.779 ns to spare relative to the constrained clock period. The design was constrained to the default Zynq PL fabric clock period of $T_{\text{clk}} = 10\,\text{ns}$ (100 MHz). The minimum achievable clock period T_{min} is given by subtracting the available slack from the constrained period:

$$T_{\min} = T_{\mathrm{clk}} - \mathrm{WNS} = 10.000 - 3.779 = 6.221 \text{ ns} \tag{3}$$

The corresponding maximum operating frequency is therefore:

$$f_{\max} = \frac{1}{T_{\min}} = \frac{1}{6.221 \times 10^{-9}} \approx 160.7 \text{ MHz} \tag{4}$$

This result indicates that the implemented design has a timing margin of approximately 60.7% above the operating frequency of 100 MHz, and could in principle be re-constrained to operate at up to 160.7 MHz before any setup timing violations would occur. The positive WNS across all endpoints confirms that the ripple-carry critical path identified in Section **??**, while the tightest path in the design, is fully closed at the target frequency.

3.5.2 Hold and Pulse-Width Margins

The Worst Hold Slack (WHS) is +0.014 ns, indicating that all hold-time constraints are met, albeit with a much tighter margin than setup. Hold violations are independent of clock frequency and are instead sensitive to routing delay imbalances between launch and capture paths; the near-zero WHS suggests the router worked close to its hold-fixing limits on at least one path, though no violations were produced. The Worst Pulse Width Slack (WPWS) of +8.750 ns confirms that all clock signals meet their minimum high and low time requirements with substantial margin.

4 Debugging and Testing

4.1 Testing Methodology

The RISC-V pipeline processor was validated through a two-stage strategy: RTL behavioural simulation followed by physical hardware-in-the-loop verification on the PYNQ-Z1/Z2 FPGA board. The primary benchmark used throughout both stages was a 32-element signed integer bubble sort, chosen because it exercises all critical pipeline features simultaneously ; load/store instructions with data hazards, ALU comparisons, conditional branches with flushes, and register forwarding across multiple pipeline stages.

4.2 Simulation-Based Testing

4.2.1 Testbench Structure

The processor was first verified using a dedicated behavioural testbench (`rv_pl_behavior_simulation_tb.v`) and later a more comprehensive bubble sort testbench (`tb_bubble_sort.v`). Both testbenches instantiate the RISC-V core with separate instruction and data memory arrays, mimic BRAM's one-cycle read latency, and monitor pipeline stage signals via `$display` statements. Six functional test categories were implemented in the initial testbench, covering RAW forwarding, load-use hazard detection, branch-taken loops, branch slip (flush) detection, `lw`→`sw` forwarding, and branch-not-taken PC progression.

4.2.2 Initial Simulation Failures and Root Causes

Several of these tests failed during early simulation runs, revealing fundamental issues in the pipeline implementation. These were diagnosed through waveform inspection and debug print statements embedded in the testbench.

Program Counter Misalignment. When monitoring PC values across the Decode, Execute, and Memory stages, the PC appeared significantly behind the expected instruction flow ; downstream stages were operating on stale PC values rather than those corresponding to the instruction currently being evaluated. This was traced to the BRAM's registered output mode, which introduces a read latency that the original design had not accounted for. The instruction fetch stage was effectively delayed relative to the rest of the pipeline, causing all subsequent stage PCs to be misaligned.

Memory Write Enable Not Asserted. During store operations, the memory write enable signal remained low even when a store instruction reached the Memory stage. The control signals were generated correctly in the Decode stage but were not properly latched through the intervening pipeline registers, so by the time the instruction reached the Memory stage the write-enable had been lost. This caused the load-use hazard and `lw`→`sw` forwarding tests to fail initially.

Delayed Branch Resolution. Branch decisions were observed to resolve several cycles later than expected, with PC redirection occurring only after the branch instruction had progressed further down the pipeline than intended. Because branch target computation relied on values not yet fully stabilised due to BRAM latency, incorrect instructions were allowed to propagate into the Decode and Execute stages before being flushed, producing the branch slip failures observed in simulation.

Pipeline Stage Offset. Across multiple tests, instructions were consistently found to be approximately two stages behind their expected position in the pipeline. The original design assumed single-cycle memory access; the BRAM IP core's two-cycle read delay invalidated this assumption and required additional synchronisation logic throughout the pipeline.

4.2.3 Design Corrections

To resolve these issues, pipeline registers were realigned to account for the BRAM latency, and hazard detection logic was extended to insert the correct number of stall cycles for `lw` dependencies. Branch handling logic was updated to flush invalid instructions at the right pipeline stage, and all control signals ; particularly the memory write enable ; were explicitly pipelined through the stage registers to ensure correct propagation into the Memory stage. Three specific architectural changes were made as part of this process.

An **instruction buffer register** was inserted between the BRAM output and the Fetch stage to synchronise instruction fetch with the BRAM's registered output:

```
always @(posedge clk) begin
    if (!rst_n)
        F_instr_buffered <= 32'h00000013;  // NOP on reset
    else if (!stallF)
        F_instr_buffered <= F_instr_from_bram;
end
assign F_instr = F_instr_buffered;
```

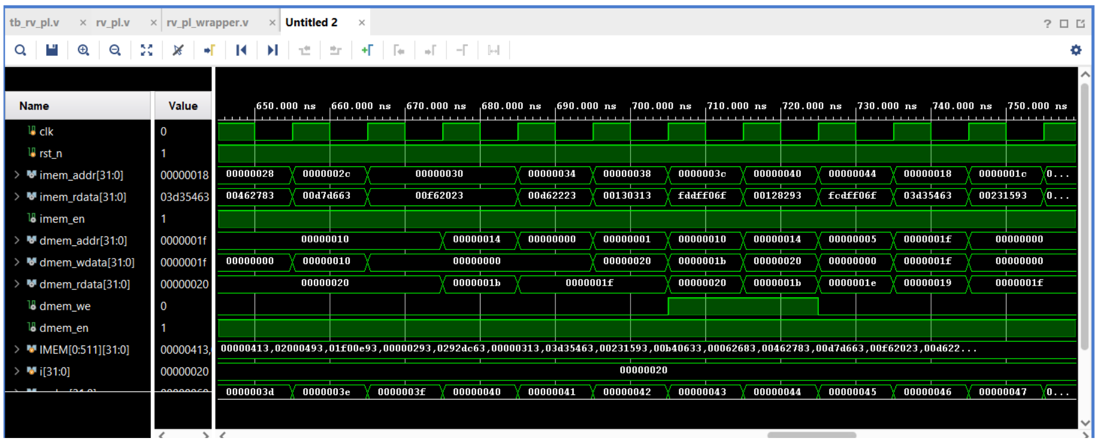
An **additional stall condition** (`lwstall_M`) was added to the hazard unit to handle the case where a store instruction's write data depends on a load result still in the Writeback stage ; a dependency not covered by the original stall logic:

```
assign lwstall_M = M_MemWrite && W_RegWrite
                && (M_Rs2 == W_Rd) && (W_Rd != 0)
                && W_ResultSrc[0];
assign D_stall = lwstall_D || lwstall_E || lwstall_M || bram_read_stall;
```

The **forwarding network** was updated so that the Writeback-stage result correctly includes the buffered load data, preventing stale values from being forwarded during load-use sequences:

```
assign W_Result = W_ResultSrc[0] ? W_ReadData_buffered : W_ALUResult;
```

After these corrections, all six simulation test categories passed, including the previously failing load-use hazard and branch tests. The bubble sort testbench subsequently completed in **4,523 cycles** with all 32 elements correctly sorted, as confirmed by the testbench waveform shown in Figure 5.



```
Name               Value      650.000 ns  660.000 ns  670.000 ns  680.000 ns  690.000 ns  700.000 ns  710.000 ns  720.000 ns  730.000 ns  740.000 ns  750.000 ns
clk                0
rst_n              1
imem_addr[31:0]    00000018   00000028 | 0000002c | 00000030 | 00000034 | 00000038 | 0000003c | 00000040 | 00000044 | 00000018 | 0000001c | 0...
imem_rdata[31:0]   03d35463   00462783 | 00d7d663 | 00f62023 | 00d62223 | 00130313 | fddff06f | 00128293 | fcdff06f | 03d35463 | 00231593 | 0...
imem_en            1
dmem_addr[31:0]    0000001f   00000010 | 00000014 | 00000000 | 00000001 | 00000010 | 00000014 | 00000005 | 0000001f | 00000000
dmem_wdata[31:0]   0000001f   00000000 | 00000010 | 00000000 | 00000020 | 0000001b | 00000020 | 00000000 | 0000001f | 00000000
dmem_rdata[31:0]   00000020   00000020 | 0000001b | 0000001f | 00000020 | 0000001b | 0000001e | 00000019 | 0000001f
dmem_we            0
dmem_en            1
IMEM[0:511][31:0]  00000413,  00000413,02000493,01f00e93,00000293,0292dc63,00000313,03d35463,00231593,00b40633,00062683,00462783,00d7d663,00f62023,00d622...
i[31:0]            00000020   00000020
                              0000003d | 0000003e | 0000003f | 00000040 | 00000041 | 00000042 | 00000043 | 00000044 | 00000045 | 00000046 | 00000047 | 0...
```

Figure 5 Vivado simulation waveform showing the bubble sort benchmark completing successfully after pipeline corrections.

4.3 Hardware-in-the-Loop Testing

4.3.1 PYNQ Deployment Workflow

After simulation, the design was deployed to the PYNQ board. The bitstream was loaded via the PYNQ overlay API, the processor held in reset while instructions and test data were written to the BRAM via the AXI BRAM controller, and reset then released. A Python script polled the completion flag address (`0x100`) for the sentinel value `0xDEADBEAF`, and upon detection read back the 32 output values for comparison against Python's native `sorted()` function.

4.3.2 Hardware vs. Simulation Discrepancy: The Zero-Write Issue

Despite all simulation tests passing, a critical discrepancy was discovered on the physical board. When the original bubble sort program ; which used `jal` instructions for loop control ; was executed on the FPGA, spurious zero values were written to data memory in place of the correct positive integer values. Negative integers sorted correctly in all cases. Table 3 summarises the observed behaviour.

Table 3 Zero-Write Issue: Simulation vs. Hardware Behaviour

Environment	Test Case	Result	Notes
Vivado Simulator	All data patterns	Pass	Idealised memory timing
FPGA (with `jal`)	Positive integers	Fail	Zeros written to memory
FPGA (with `jal`)	Negative integers	Pass	MSB=1 distinguishable from PC values
FPGA (unrolled)	All data patterns	Pass	`jal` instructions eliminated

4.4 Known Issue: `JAL`/`JALR` Hardware Behaviour

4.4.1 Root Cause Hypothesis

The suspected mechanism is a forwarding conflict between the return address written by `jal` (PC+4) and the data values being forwarded through the pipeline. In the original looping program, the `jal` instruction executed repeatedly on every loop iteration, continuously inserting small PC-derived values (e.g., `0x00000004`, `0x00000008`) into the forwarding network. The hypothesis is that the synthesised forwarding multiplexer incorrectly prioritised these PC-derived values over loaded data values when both were simultaneously present in the pipeline. Positive integers have MSB = 0 ; the same as typical program counter values in this design ;

making them vulnerable to being displaced. Negative integers have MSB = 1 and are therefore distinguishable from address values, which explains why they were consistently unaffected.

This defect was entirely invisible in Vivado's behavioural simulation, which is consistent with a synthesis-level difference between the RTL model and the routed netlist. The simulator's timing model does not fully replicate post-synthesis forwarding path characteristics or BRAM latency interactions, and may mask this class of bug.

4.4.2 Workaround

The `jal` instructions were eliminated by fully unrolling the bubble sort algorithm. Each of the 496 compare-and-swap operations was written as a flat, sequential block of six instructions (two loads, one comparison, one conditional forward branch, and two conditional stores), producing a 2,980-instruction program with no backward jumps. This workaround fully resolved the zero-write issue across all data patterns tested. The trade-off is a significantly larger instruction memory footprint and the inability to generalise the program to variable-length inputs without reintroducing `jal`.

4.5 Test Coverage

The unrolled bubble sort benchmark exercises the RV32I instructions shown in Table 4. The `jal` and `jalr` instructions are intentionally excluded from the current test suite due to the hardware issue described above.

Table 4 Instruction Coverage in the Bubble Sort Benchmark

Instruction	Count	Coverage Details
`lw`	992	Load operations (2 per compare-swap block)
`sw`	368	Store operations (executed swaps only)
`beq`	500	Conditional branches (swap decision)
`slt`	496	Signed comparison (every compare-swap block)
`slli`	496	Address offset calculation
`add`	496	Base + offset address formation
`addi`	8	Constant loading
`sub`	31	Loop limit calculation
`lui`	1	Completion flag (`0xDEADBEAF`)
Total	**2,980**	

RAW hazards were covered for all three forwarding paths (Execute-to-Execute, Memory-to-Execute, and Writeback-to-Execute). Load-use hazards were covered for both the standard case and the load-store dependency case. Control hazards were covered for both branch-taken and branch-not-taken paths. Write-After-Write and Write-After-Read hazards are not applicable to a single-issue in-order pipeline.

4.6 Reset Design: GPIO vs. Physical Button

4.6.1 Why Reset is Critical on Hardware

Reset is essential for reliable processor operation on the FPGA board. Without a defined reset state, pipeline registers, the program counter, and the hazard unit all power up with undefined values. On the PYNQ board, the processor begins fetching instructions immediately when the bitstream is loaded ; before any program or data has been written to the BRAMs. Without asserting reset first, the core will attempt to execute whatever stale or uninitialised values exist in instruction memory, potentially issuing spurious memory writes that corrupt the data array before the test even begins. Reset therefore serves two purposes: it holds the processor in a known idle state while the PS writes the program and test data over AXI, and it provides a clean restart mechanism between test runs.

4.6.2 GPIO-Based Reset

The primary reset method uses an AXI GPIO peripheral connected to the processor's active-low reset pin. The PS writes to the GPIO data register over AXI-Lite to assert or de-assert reset programmatically:

```
def freeze_processor():
    reset_controller.write(0x0, 0x0)   # assert reset

def release_processor():
    reset_controller.write(0x0, 0x1)   # de-assert reset, core runs from PC=0
```

The execution flow is fully deterministic: the processor is frozen before program upload, the AXI BRAM writes complete, and reset is released in a single Python call. There is no ambiguity about processor state at any point in the sequence, and the transition from reset to running is instantaneous from the software's perspective.

4.6.3 Physical Button Reset

An alternative bitstream was implemented in which BTN0 on the PYNQ board drives the processor reset directly through a button controller module in the PL fabric. This approach introduced several practical complications that are discussed below.

Jitter and Debouncing. Mechanical push-buttons produce a noisy signal when pressed or released, as the contacts bounce between open and closed for several milliseconds before settling. On an FPGA running at 100 MHz, this bounce appears as a rapid sequence of spurious rising and falling edges, which would cause the reset signal to toggle many times in the space of a single button press. A hardware debouncer was implemented in the button controller module to filter these transitions. The standard approach is a synchroniser followed by a counter-based filter: the button input is first registered through two flip-flops to prevent metastability, and the output is only allowed to change state after the input has remained stable for a fixed number of clock cycles (typically 1020 ms worth of counts at the operating frequency). This ensures that each physical press produces exactly one clean rising and one clean falling edge on the reset signal.

The State Detection Problem. A practical difficulty with button-driven reset is that the Python verification script has no direct visibility into whether the processor is currently in reset or running. If the user runs a test cell without first holding BTN0, the script will load program and data into BRAM while the processor is actively executing, corrupting the test. To address this, a sentinel-based state detection method was implemented:

```
def is_processor_in_reset() -> bool:
    TEST_SENTINEL = 0xCAFEBABE
    data_memory.write(DRAM_COMPLETION_FLAG_ADDR, TEST_SENTINEL)
    time.sleep(0.2)
    current = read_completion_flag()
    return current == TEST_SENTINEL  # unchanged => processor not running
```

The script writes a known sentinel value (`0xCAFEBABE`) to the completion flag address, waits 200 ms, and reads it back. If the value is unchanged, the processor has not written to that address and is therefore either in reset or halted; if the value has changed, the processor is actively executing. The script loops with a live countdown display, prompting the user to hold BTN0 until the in-reset condition is confirmed before proceeding with program upload.

Execution Delay. Each call to `is_processor_in_reset()` introduces a 200 ms polling delay. Because the function is called repeatedly until the button is confirmed held, the total overhead before a test can begin is at least 0.9 s per run, in addition to any time the user takes to physically press the button. This delay is entirely absent from the GPIO method, where `freeze_processor()` is instantaneous.

Furthermore, users must be explicitly instructed to *hold* the button rather than simply press it, because a brief tap may not be registered across the 200 ms polling window. The button notebook therefore prints a persistent prompt:

```
print('Press and HOLD BTN0 now!')
print('(Watching for processor to enter reset...)')
```

4.6.4 Comparison and Design Choice

Table 5 summarises the key differences between the two reset methods.

Table 5 GPIO Reset vs. Physical Button Reset

Property	**GPIO (AXI)**	**Physical Button**
Reset timing	Deterministic, instantaneous	Manual, variable latency
Jitter handling	Not required	Hardware debouncer required
State visibility	Explicit (software-controlled)	Requires sentinel polling
Per-run overhead	$\approx$ 0 ms	$\geq$ 900 ms
User intervention	None	Must hold button per run
Automation	Fully automatable	Cannot be fully automated
Suitable for	Scripted test suites	Interactive/manual testing

The GPIO method is clearly superior for automated verification. It gives the Python script complete, unambiguous control over the processor's reset state, eliminates per-run overhead, and requires no physical interaction from the user. The entire test sequence - freeze, load program, load data, release, poll for completion, verify results - executes as a single deterministic Python function call with no external dependencies.

The physical button approach is nonetheless useful for interactive debugging sessions, where a user may want manual control over when the processor starts, the ability to single-step through execution, or visual confirmation of state via the board LEDs. The button controller bitstream exposed additional status signals on LED1LED3 (halted, single-step, force-done) which provided useful visibility during early hardware debugging. For production test runs, however, GPIO reset was the preferred and recommended method.

4.7 Key Lessons Learned

Two significant engineering insights emerged from this process. First, **simulation success does not guarantee hardware correctness**. The zero-write defect caused by `jal` was entirely invisible in behavioural simulation yet consistent and reproducible on the physical device, demonstrating that hardware validation on the target board must be treated as a mandatory step in the verification flow. Second, **BRAM latency is a first-class architectural constraint**. The majority of the early simulation failures ; PC misalignment, stale control signals, and delayed branch resolution ; all traced back to the assumption of single-cycle memory access. Incorporating the BRAM timing model into the pipeline design from the outset, rather than retrofitting it, would have avoided the cascade of downstream synchronisation issues encountered during development.

5 Conclusion

This project demonstrates a functional FPGA-based RISC-V RV32I pipelined processor implemented on the Xilinx Zynq-7000 SoC (xc7z020clg400-1) and integrated into a Vivado block design alongside AXI interconnect and Block RAM peripherals. The five-stage pipeline - Fetch, Decode, Execute, Memory, and Writeback - correctly handles the core hazards of a pipelined RISC architecture: RAW data hazards are resolved through a three-path forwarding network, load-use hazards are mitigated by a one-cycle stall, and control hazards are handled by flushing the Execute stage on a taken branch.

The primary technical challenges encountered during development centred on two themes. The first was BRAM latency management: the Xilinx Block RAM's registered output mode introduced a one-cycle read delay that invalidated the original assumption of single-cycle memory access, cascading into PC misalignment, lost control signals, and delayed branch resolution across the pipeline. Resolving this required the addition of an instruction buffer register, realigned stall logic, and explicit pipelining of all control signals through the stage registers. The second challenge was the simulation-to-hardware gap exposed by the `jal/jalr` defect, in which spurious zero values were written to data memory for positive integers during hardware execution despite all simulation tests passing. This was attributed to a forwarding path conflict between PC-derived return addresses and data values in the synthesised netlist - a discrepancy invisible to Vivado's behavioural simulator. The workaround of fully unrolling the bubble sort algorithm eliminated the defect and produced a stable, hardware-validated implementation.

From a timing perspective, the routed design meets all 20,600 timing endpoints at 100 MHz with a worst negative slack of +3.779 ns, corresponding to a maximum achievable operating frequency of approximately 160.7 MHz. Resource utilisation is modest: the processor core (`rv_pl_wrapper_0`) consumes 772 LUTs and 554 flip-flops, with the Decode/Execute pipeline register (`PLR2`) accounting for the majority of logic due to the wide immediate-extension and ALU control decode logic concentrated at that stage boundary. The critical path runs through the `LW` address adder - a ripple-carry chain driven by the forwarding mux - and terminates at the data memory BRAM setup pin, a well-known bottleneck in pipelined RISC designs.

Future work should prioritise root-cause resolution of the `jal/jalr` hardware defect using Integrated Logic Analyser (ILA) probes to capture forwarding multiplexer behaviour during jump execution. Additional directions include implementing the RV32M multiply/divide extension, adding a basic cache hierarchy to reduce memory access penalties, and extending the hazard unit to support interrupt handling and CSR instructions.

The project source code and verification scripts are available at:

https://github.com/chits-nema/FPGA_RISC_V_PP

6 Contributions

All three team members contributed equally across the overall project. The specific responsibilities were distributed as follows.

Chitsidzo Varaidzo Nemazuwa led the RTL implementation, taking primary responsibility for the pipeline register design, hazard detection logic, and forwarding network. Chitsidzo also co-authored this report.

Kanaya Nisa Ozora led the testing and verification effort, designing and running the simulation testbenches, diagnosing pipeline failures through waveform analysis, and coordinating hardware-in-the-loop testing on the PYNQ board. Kanaya also co-authored this report.

Nawal Salama took primary responsibility for the Vivado block design, AXI interconnect integration, and memory-mapped peripheral configuration, and led the preparation of the project presentation. Remaining tasks - including debugging sessions, design reviews, and verification runs - were shared equally across all three members.

7 AI-Generated Content (AIGC) Declaration

In accordance with the academic integrity policy of the Chair of Computer Architecture and Operating Systems, TU Munich, the authors declare the following use of AI-assisted tools during the preparation of this project and report.

- **Claude (Anthropic, 2025)** was used to assist with writing and editing sections of this report, and to support the understanding of technical concepts related to FPGA implementation and pipeline design.

- **ChatGPT (OpenAI, 2025)** was used to support the understanding of RISC-V architectural concepts and Vivado toolflow.

- **GitHub Copilot (GitHub/Microsoft, 2025)** was used as a code completion assistant during RTL development in Verilog.

All AI-generated or AI-assisted content was reviewed, verified, and edited by the authors. The final intellectual responsibility for the design, implementation, analysis, and conclusions presented in this report lies solely with the authors.

References

[1] A. Waterman and K. Asanović, Eds., *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*, RISC-V Foundation, Document Version 20191213, 2019. [Online]. Available: https://riscv.org/technical/specifications/

[2] D. M. Harris and S. L. Harris, *Digital Design and Computer Architecture: RISC-V Edition*. Morgan Kaufmann, 2021.

[3] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: RISC-V Edition*, 2nd ed. Morgan Kaufmann, 2020.

[4] Xilinx Inc., *Zynq-7000 SoC Technical Reference Manual*, Document UG585, v1.13, 2023. [Online]. Available: https://docs.xilinx.com/r/en-US/ug585-zynq-7000-TRM

[5] Xilinx Inc., *Block Memory Generator v8.4 Product Guide*, Document PG058, 2021. [Online]. Available: https://docs.xilinx.com/r/en-US/pg058-blk-mem-gen

[6] Xilinx Inc., *AXI BRAM Controller v4.1 Product Guide*, Document PG078, 2021. [Online]. Available: https://docs.xilinx.com/r/en-US/pg078-axi-bram-ctrl

[7] Xilinx Inc., *Vivado Design Suite User Guide: Design Analysis and Closure Techniques*, Document UG906, v2021.2, 2021. [Online]. Available: https://docs.xilinx.com/r/en-US/ug906-vivado-design-analysis

[8] Xilinx Inc., *PYNQ-Z1 Reference Manual*, Digilent Inc., 2021. [Online]. Available: http://www.pynq.io/

[9] Anthropic, "Claude (Version claude-sonnet-4-6)," Large language model, 2025. [Online]. Available: https://www.anthropic.com Accessed: Feb. 2026.

[10] OpenAI, "ChatGPT (GPT-4)," Large language model, 2025. [Online]. Available: https://openai.com/chatgpt Accessed: Feb. 2026.

[11] GitHub Inc., "GitHub Copilot," AI-powered code completion tool, 2025. [Online]. Available: https://github.com/features/copilot Accessed: Feb. 2026.