IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511205

Implementation of RISC-V Pipelined Architecture on FPGA

Prof. Sujatha S Ari[1], Manya R[2], Lekhana B S[3], Pooja S K[4], Nruthya R[5]

Assistant Professor, Electronics and Communication Engineering, East West Institute of Technology, Bengaluru, India[1]

Students, Electronics and Communication Engineering, East West Institute of Technology, Bengaluru, India[2,3,4,5]

Abstract: This paper presents a five-stage pipelined processor based on the RISC-V RV32I instruction set, developed in Verilog and implemented on the Sipeed Tang Primer 25K Dock FPGA. The datapath is divided into instruction fetch, decode, execute, memory access and write back stages. A hazard unit combines data forwarding with a one cycle load-use stall, while branches and jumps are resolved in the execute stage by flushing the earlier stages. Correct operation was confirmed through simulation and a self-checking program whose register results were reported over UART and read as PASS on a serial terminal. Post-implementation timing analysis against a 100 MHz constraint gives a maximum operating frequency of 64.548 MHz, with the critical path running from the data memory output to the program counter. The design occupies 4,738 look-up tables and 2,059 registers and draws a total power of 81.008 mW, compared with 96 mW reported for a reference five stage design on a different device. Our test program completes in 46 cycles with a single stall.

Keywords: RISC-V, RV32I, pipelined architecture, Verilog, hazard handling, UART

I. INTRODUCTION

Modern computing depends on fast processors, which are built as integrated circuits [2]. The architectures that dominate the market, such as ARM and Intel x86, are protected by intellectual property rights. Using them for research or teaching usually involves licence fees and restricted documentation, so students and small research groups find it hard to study processor design on real hardware. An open instruction set removes this barrier, and RISC-V is a widely adopted example. Its specification is freely available, and it has attracted strong interest in both academia and industry, including work on logic synthesis of RISC-V designs across technology nodes [3]. Chip design as a whole covers device level studies [4], training of engineers in industry standard design flows [5], ASIC implementation of complex systems [6] and the verification of large designs [7]. Verification of RISC-V cores in particular has been used as a teaching tool [8], which shows how well suited the architecture is to learning.

RISC-V follows the reduced instruction set principles, which favour a small set of simple and regular instructions over a large and complex one [9]. This regularity keeps the decoding logic simple and makes the architecture a good fit for a student implementation. Several open-source RISC-V cores have already been evaluated on FPGAs [10], and complete designs have been reported on this platform [11]. Pipelined designs with five stages have also been implemented there to raise throughput [1].

The main objective of this work is to design and implement a processor that executes the instructions of the RV32I base integer set. A single cycle design is easy to understand, but its clock period is fixed by the slowest instruction [1]. Pipelining overcomes this by dividing execution into fetch, decode, execute, memory access and write back stages, so that several instructions are processed at the same time. The cost of this speed is a set of hazards, namely data, load use and control hazards, which must be resolved for programs to run correctly [1]. This paper explains how these were handled through forwarding, a load use stall and branch handling.

The processor is written in Verilog and implemented on the Sipeed Tang Primer 25K Dock. Its operation was first checked in simulation with a self-checking testbench, and it was then observed on the board, with results displayed over UART on a serial terminal.

The rest of the paper is organised as follows. Section II describes background and related work. Section III presents the design of the proposed pipelined processor. Section IV discusses the verification and results. Section V gives the conclusions, and Section VI outlines the future scope of the work.

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511205

II. BACKGROUND AND RELATED WORK

The RISC-V base integer ISA, called RV32I, contains 47 instructions grouped into register-register, register-immediate, load, store, branch, jump and upper-immediate formats [15]. All instructions are 32 bits wide and the source and destination register fields occupy the same bit positions in every format, which simplifies decoding hardware. The load-store nature of the ISA also means that only two instruction classes touch data memory, a property that fits neatly into a dedicated pipeline stage.

Several open implementations exist. PicoRV32 is a small multi-cycle core optimized for area rather than speed [12]. The Rocket generator produces a parameterized in-order pipeline together with caches and an on-chip interconnect, aimed at system-on-chip designs [13]. The textbook design in [14] describes a classical five-stage organization that serves as the starting point for many university projects. The present work follows the same stage partitioning but concentrates on a self-contained and readable implementation, with hazard conditions stated explicitly and targeted at the low-cost Sipeed Tang Primer 25K Dock [16].

III. PROPOSED PIPELINED PROCESSOR

A. Supported Instructions

The core implements the RV32I user-level instructions: arithmetic and logic operations in register and immediate forms, shifts, set-less-than comparisons, word, half-word and byte loads and stores, six conditional branches, JAL, JALR, LUI and AUIPC. System calls, the fence instruction and control and status registers are outside the scope of this design. Register x0 is hard-wired to zero. Decoding is kept shallow: one control module looks only at the seven opcode bits, and the finer choice of ALU operation is left to funct3 and one bit of funct7.

B. Pipeline Organization

Fig. 1 shows the overall organization. Each stage performs a limited amount of work so that the clock period is set by the slowest single stage rather than by the whole instruction path. Pipeline registers sit between neighbouring stages and carry the data and the control bits that the later stages still need.

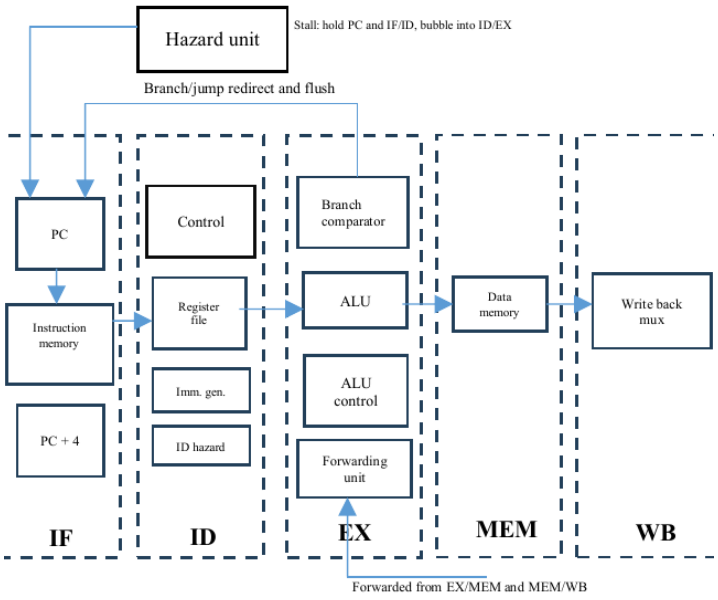

Fig. 1 Block diagram of the five-stage RV32I pipeline

1) **Instruction Fetch (IF):** The program counter addresses the instruction memory, a 256-word array that is loaded from a hexadecimal file when the design is built, and is incremented by four every cycle unless a taken branch or jump supplies a new target. If several conditions arise together, the next address is chosen in a fixed order: the halted state first, then a redirect from EX, then a stall, and only then PC+4.
2) **Instruction Decode (ID):** The instruction is split into its fields, the two source operands are read from the 32-entry register file, and the immediate is assembled according to the instruction format. The register file is written on the rising clock edge, but its read ports are transparent: a value that is being written back in a given cycle is

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511205

already visible to the instruction in decode in that same cycle. A second bypass in ID takes the newest value from the MEM or WB stage before the operands are latched into ID/EX. The second operand is forced to zero for instructions that do not read it, so that unrelated bits of the instruction word cannot create a false dependency.

3) **Execute (EX):** The ALU performs the arithmetic or logic operation or computes the effective address for memory instructions. For LUI the first operand is replaced by zero and for AUIPC by the program counter, so both instructions reuse the adder. Branch conditions are evaluated here by a dedicated comparator working on the forwarded register values, and the branch target is formed by a separate adder that sums the program counter and the immediate. A JALR target is taken from the ALU result with bit 0 cleared.
4) **Memory Access (MEM):** Loads and stores access the data memory, with byte enables generated from the funct3 field (Section III-E).
5) **Write-Back (WB):** A multiplexer chooses between the ALU result, the loaded data and the link address (PC+4 for JAL and JALR), and the selected value is written to the destination register.

Table I lists the control word that the control unit produces for each instruction class. The two-bit field alu_op separates three cases. The value 00 is used where the ALU only adds (address and link arithmetic), 01 selects subtraction for the branch comparison path, and 10 hands the decision to funct3 and bit 30 of the instruction. That last bit needs care. It tells SUB from ADD and SRA from SRL, but an ADDI with a negative immediate has the same bit set, so the ALU control module also receives a flag marking I-type instructions and ignores the bit for ADDI. The ALU offers eleven operations: add, subtract, AND, OR, XOR, three shifts, signed and unsigned set-less-than, and a pass-through code that the present control unit does not use. Shift amounts are taken from the low five bits of the second operand.

TABLE I CONTROL SIGNALS BY INSTRUCTION CLASS

Class	RegW	ALUSrc	MemR	MemW	M2R	alu_op	Flag
R-type	1	0	0	0	0	10	-
I-type	1	1	0	0	0	10	-
Load	1	1	1	0	1	00	-
Store	0	1	0	1	0	00	-
Branch	0	0	0	0	0	01	branch
JAL	1	0	0	0	0	00	jump
JALR	1	1	0	0	0	00	jalr
LUI	1	1	0	0	0	00	lui
AUIPC	1	1	0	0	0	00	auipc

C. Data Hazards

A read-after-write hazard occurs when an instruction needs a result that an earlier instruction has not yet written to the register file. Most such cases are solved without delay by a forwarding unit that routes results from the EX/MEM and MEM/WB registers back to the ALU inputs. The unit compares the destination register of the older instructions with the source registers of the instruction in EX. The conditions are summarized in Table II.

TABLE II FORWARDING CONDITIONS IN THE EXECUTE STAGE

Select	Condition	Operand source
10	RegWrite in MEM, rd(MEM) ≠ x0 and rd(MEM) = rs(EX)	EX/MEM result
01	No match above, RegWrite in WB, rd(WB) ≠ x0 and rd(WB) = rs(EX)	MEM/WB value
00	Otherwise	ID/EX register

The priority given to the EX/MEM match matters when two older instructions write the same register: the most recent value must win. Loads are the exception to forwarding, since the data only becomes available in the MEM stage. A load immediately followed by a dependent instruction therefore costs one stall cycle, as illustrated in Fig. 2(a). The hazard unit raises the stall when the instruction in EX is a load whose destination is not x0 and matches either source register of the instruction in ID. While the stall lasts, the program counter and the IF/ID register hold their contents and a bubble

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511205

enters ID/EX. In the following cycle the load is in MEM and the waiting instruction picks up its data through the bypass in ID, so one cycle is always enough.

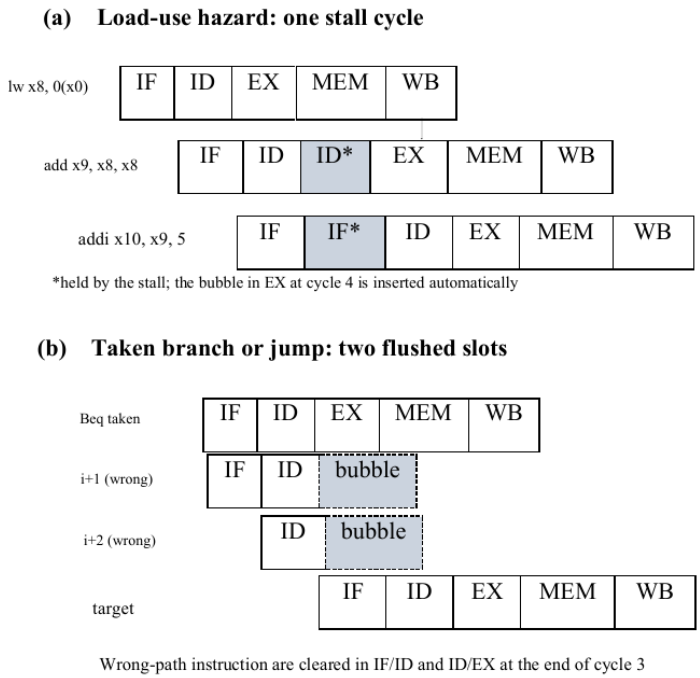

Fig. 2 Pipeline timing: (a) the lw/add pair taken from the test program, stall cycle marked by an asterisk (*), (b) a taken branch.

D. Control Hazards

Branches and jumps are resolved in the EX stage. The fetch unit simply assumes that a branch is not taken and continues to fetch sequential instructions. If the comparator later finds that the branch is taken, or if a jump is executed, the two instructions that were fetched in the meantime are discarded by clearing the IF/ID and ID/EX registers, and the program counter is loaded with the target address (Fig. 2(b)). A taken control transfer thus costs two cycles. JAL and JALR take the same path, so a jump costs two cycles even when its target is the next instruction. Resolving branches earlier would reduce this penalty, but it would lengthen the decode stage and place the comparator on the register-file read path.

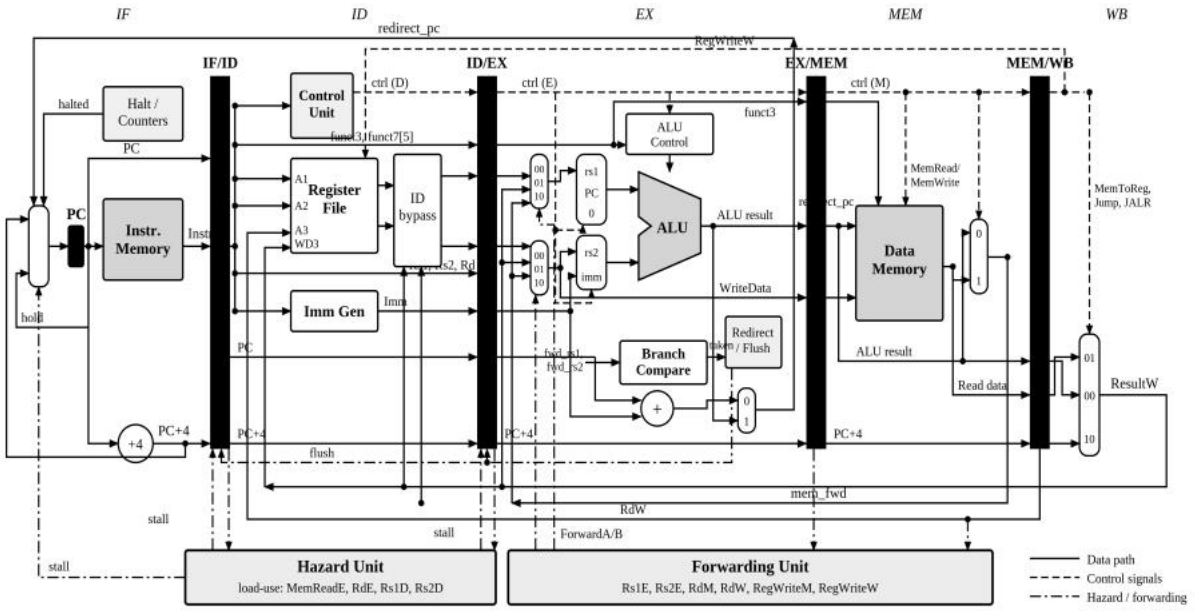

Fig. 3 Pipelined architecture of the RISC-V processor

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511205

E. Memory Subsystem

The data memory holds 256 words and is built from four byte-wide arrays, one per byte lane. Each lane has its own write enable, derived from funct3 and the two low address bits: a byte store enables one lane, a half-word store enables the lower or the upper pair, and a word store enables all four. The store data is replicated onto every lane that a byte or half-word store may target, so the enabled lane always sees the correct byte. For loads the whole word is read first and the requested byte or half-word is then selected, sign-extended for LB and LH and zero-extended for LBU and LHU. Only address bits [9:2] select the word, so the memory wraps after 1 KiB, and accesses that straddle a word boundary are not supported.

The read path is combinational. Load data are therefore valid in the same cycle in which the address sits in the EX/MEM register, which is what lets the pipeline limit the load-use penalty to a single cycle. A memory with a registered read port would deliver the word one cycle later and would need either an address taken from EX or a second stall cycle.

F. Run Control and Instrumentation

A test program ends with the self-branch beq x0,x0,0. When this instruction reaches EX and is taken, the core notes its address and keeps running for five more cycles so that all older instructions finish writing back. It then raises a halted flag that freezes the program counter and both counters. Two 32-bit counters are kept: one counts every cycle until the halt and the other counts the load-use stall cycles. Together with the number of instructions executed they give the cycles per instruction of a program directly. A debug port on the register file lets the top-level logic read any register without disturbing the pipeline.

IV. VERIFICATION AND RESULTS

A. Verification Method

Checking a pipeline against a few hand-written programs is not enough, because hazards depend on the exact spacing between instructions. There are three parts: an assembler for the instruction formats used here, an instruction set model written independently of the Verilog, and a testbench that loads a program, runs it to the halt and compares all 31 writable registers and the first 16 words of data memory with the model.

Random programs of about 120 instructions were generated from a pool of eight working registers so that neighbouring instructions often share operands. The generator covers all 37 computational, load, store, branch and jump instructions of RV32I, with byte, half-word and word accesses to a 64-byte window, forward branches that are sometimes taken and sometimes not, and JAL and JALR. Of 600 generated programs, 378 terminated in the model and were used; the others contained a JALR whose base register had been skipped by an earlier branch and ran away. Simulations were run with ModelSim.

B. Simulation Results

The results are given in Tables III and IV. The two directed programs complete with the expected register contents. The first is the 29-instruction program that is also used on the board; it exercises forwarding, one load-use pair, a counted loop and a JAL. The second adds the integers 1 to 10 in a loop and ends with 55 in x1. In both cases the measured cycle count equals the number of instructions, plus one cycle per stall, plus two cycles per taken transfer (counting the JAL), plus a fixed overhead of eight cycles for pipeline fill and the halt sequence. The same relation, with the same overhead, held for each of the 124 random programs examined for it, and the stall counts agreed with the load-use pairs found in the model.

TABLE III DIRECTED TEST PROGRAMS

Program	Instr.	Cycles	Stalls	Taken	Registers
program.hex	29	46	1	4	match
test4_sum_loop.hex	34	60	0	9	match

The random regression also exposed two faults in the first version of the data memory. Its read port was registered, so a load received the word addressed in the previous cycle instead of its own, and a half-word store to the upper half of a word wrote the wrong bytes. A load that directly followed a store to the same address happened to return correct data in simulation order, which is why the short directed program alone did not show the problem. After the read path was made combinational and the lane data corrected, all 378 programs matched the model (Table IV).

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511205

TABLE IV RANDOM REGRESSION RESULTS

Data memory	Programs	Passed	Failed
First version	378	131	247
Corrected	378	378	0

Because the directed programs are short, the fixed overhead dominates their cycles per instruction (1.59 and 1.76). Without it the figures are 1.31 and 1.53, which reflects the two-cycle cost of every taken transfer in the loop of the second program.

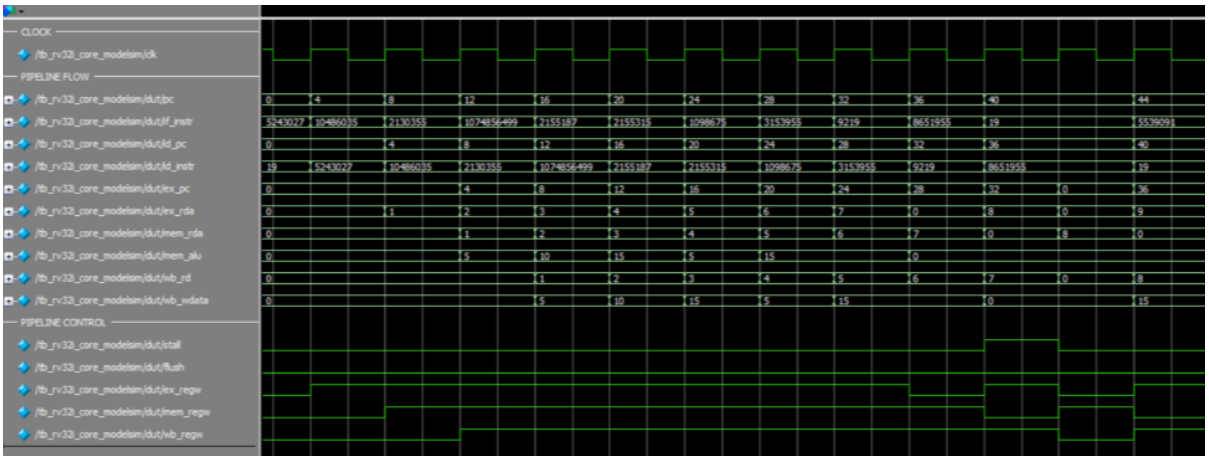

Fig. 4 Simulation waveform of the pipeline registers and control signals for program.hex.

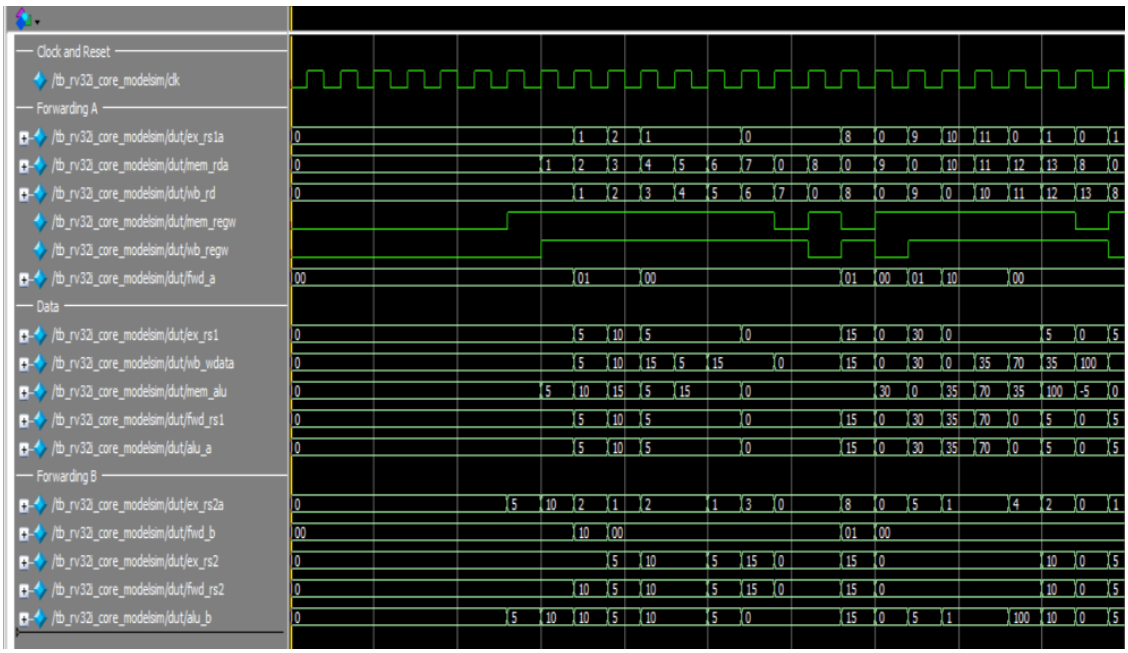

Fig. 5 Forwarding unit activity during the same run

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511205

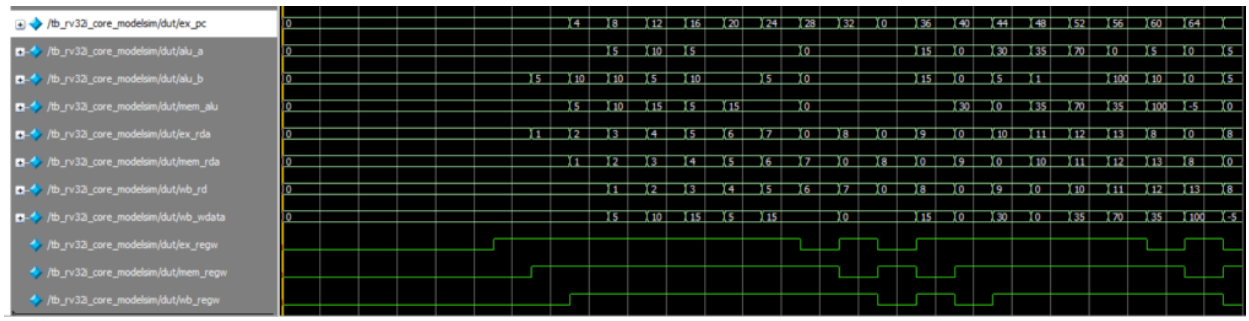

Fig. 6 Operand and result flow through the EX, MEM and WB stages over the program run.

The waveforms in Figs. 4 to 6 were captured from the testbench run of program.hex in ModelSim. Fig. 4 shows the pipeline registers and control signals. The program counter advances by four in every cycle and each instruction moves one stage per cycle. The load at address 32 reaches EX while the dependent instruction at address 36 is in ID; the stall signal is raised for one cycle, the program counter and the IF/ID register hold their values, and a bubble (ex_pc = 0) passes through EX, as described in Section III-C. Fig. 5 shows the forwarding selects for the same run. A select value of 10 takes the result held in the MEM stage, 01 takes the write-back value and 00 keeps the operand read in decode, in agreement with Table II. Fig. 6 follows the operands and results of the ALU through the MEM and WB stages. The result of each instruction is visible in mem_alu one cycle after EX and in wb_wdata one cycle after that, and the write-enable signals drop in the cycles that carry a bubble or an instruction that writes no register.

C. FPGA Implementation

The design targets the Sipeed Tang Primer 25K Dock [16], which carries a Gowin GW5A-LV25MG121 device and a 50 MHz oscillator, and is built with the Gowin IDE. The top-level wrapper uses the oscillator directly as the core clock. Button btn0 is the reset and btn1 steps through the registers x0 to x31; it is sampled about every 21 ms for debouncing. The serial report is produced by a small state machine at 115200 baud, 8 data bits, no parity and one stop bit.



Fig. 7 Implementing code to Sipeed Tang Primer 25K Dock FPGA

While the program runs, a line with the program counter and the selected register is sent about five times per second. After the halt, the machine sweeps registers x0 to x17 and, for each one, prints the value followed by OK or NG according to a table of expected values that was derived by simulating the instruction sequence. The sweep also lists the program counter at the halt, the cycle count and the stall count, ends with PASS or FAIL, and repeats after roughly one second. Register x14 is expected to stay at zero because the program never writes it.

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511205

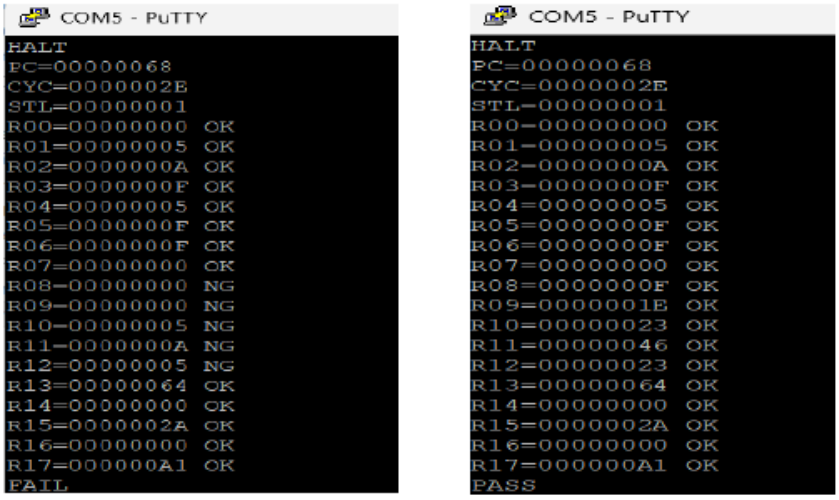

Fig. 8 UART output display which indicated FAIL/PASS

TABLE V POWER AND DESIGN UTILIZATION FOR PIPELINED ARCHITECTURE

Parameters	Usage	Utilization %
Power	81.008 mW	-
IO	10	12
BSRAM	4 SDPB, 1 pROM	9
LUT	4,738	24
FlipFlop	2,059	9

V. CONCLUSION

A five stage RV32I processor was written in Verilog and prepared for a low cost FPGA board. Forwarding from the MEM and WB stages, a decode-stage bypass and a one cycle load-use stall resolve data hazards, and branches that are resolved in EX cost two cycles when taken. The design is small enough to be read in an afternoon, and each hazard rule corresponds to a few lines of code. A randomized comparison with an independent instruction set model, together with two directed programs, showed correct behaviour for all supported instructions once two faults in the data memory had been removed. The board interface reports the final register file over UART, so the same check can be repeated on hardware without a logic analyzer.

VI. FUTURE SCOPE

The proposed RISC-V pipelined processor can be further enhanced for collaborative robot (cobot) applications by integrating real-time motor control, encoder interfaces, PWM generation, PID controllers, and CAN communication. Additional features such as watchdog timers, interrupt handling, and emergency-stop mechanisms can improve reliability and operational safety. Future work will focus on reducing control latency, optimizing FPGA resource utilization, and achieving efficient real-time performance for industrial robotic applications. These enhancements aim to establish a reliable, customizable, and cost-effective FPGA-based processor platform for next-generation collaborative robots.

REFERENCES

[1]. F. Hussain and S. Sarkar, “Design and FPGA implementation of five stage pipelined RISC-V processor”, in *Proc. 2024 IEEE 9th Int. Conf. Convergence Technol. (I2CT)*, Pune, India, Apr. 2024, pp. 1-6, doi: 10.1109/I2CT61223.2024.10544184.

[2]. K. H. Yeap, M. M. Isa, and S. H. Loh, “Introductory chapter: Integrated circuit chip”, in *Integrated Circuits/Microchips*, K. H. Yeap and J. J. Sayago Hoyos, Eds. IntechOpen, 2020.



International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026
DOI: 10.17148/IJARCCE.2026.1511205

[3]. S. H. Loh, Y. N. Koay, K. H. Yeap, and J. J. Sim, "Analysis of logic synthesis results with various technology nodes for RISC-V design", in *Proc. 2024 IEEE 14th Int. Conf. Control Syst., Comput. Eng. (ICCSCE)*, Penang, Malaysia, Aug. 2024, pp. 100-105.

[4]. K. H. Yeap, J. Y. Lee, W. L. Yeo, H. Nisar, and S. H. Loh, "Design and characterization of a 10 nm FinFET", *Malaysian J. Fundam. Appl. Sci.*, vol. 15, no. 4, pp. 609-612, Aug. 2019.

[5]. S. H. Loh, I. M. Tan, and J. J. Sim, "VLSI design course with commercial EDA tools to meet industry demand: From logic synthesis to physical design", in *Proc. 2021 11th IEEE Int. Conf. Control Syst., Comput. Eng. (ICCSCE)*, Penang, Malaysia, Aug. 2021, pp. 55-60.

[6]. S. H. Loh, P. Y. Aw Yong, J. J. Sim, and K. H. Yeap, "Implementation of individual and combined in-loop filters for HEVC and AVC video coding standards on ASIC platform using 32 nm technology library", *Appl. Model. Simul.*, vol. 9, pp. 292-302, Jul. 2025.

[7]. Z. N. Lim *et al.*, "A reconfigurable and scalable verification environment for NoC design", in *Proc. 2013 Conf. New Media Stud. (CoNMedia)*, Tangerang, Indonesia, Nov. 2013, pp. 1-4.

[8]. S. H. Loh, Y. H. Liew, and J. J. Sim, "VLSI design course with verification of RISC-V design using Universal Verification Methodology (UVM)", in *Proc. 2022 IEEE 12th Int. Conf. Control Syst., Comput. Eng. (ICCSCE)*, Penang, Malaysia, Oct. 2022, pp. 7-12.

[9]. A. D. George, "An overview of RISC vs. CISC", in *Proc. 22nd Southeastern Symp. Syst. Theory*, Jan. 1990, pp. 436-437.

[10]. R. Holler *et al.*, "Open-source RISC-V processor IP cores for FPGAs: Overview and evaluation", in *Proc. 8th Mediterranean Conf. Embedded Comput. (MECO)*, Budva, Montenegro, Jun. 2019, pp. 1-6.

[11]. L. Poli, S. Saha, X. Zhai, and K. D. McDonald-Maier, "Design and implementation of a RISC V processor on FPGA", in *Proc. 17th Int. Conf. Mobility, Sens. Netw. (MSN)*, Dec. 2021, pp. 161-166.

[12]. C. Wolf, "PicoRV32: A size-optimized RISC-V CPU", GitHub repository. [Online]. Available: https://github.com/YosysHQ/picorv32

[13]. K. Asanović *et al.*, "The Rocket Chip generator", Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016-17, Apr. 2016. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[14]. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 2nd ed. Cambridge, MA, USA: Morgan Kaufmann, 2020.

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

[16]. Sipeed, "Tang Primer 25K", Sipeed Wiki. Available: https://wiki.sipeed.com/hardware/en/tang/tang-primer-25k/primer25k.html (accessed Oct. 2026).

 This work is licensed under a Creative Commons Attribution 4.0 International License