Microarchitectural Timing Optimization of the CV32E40P RISC-V Core on FPGA

Nikas Ioannis Iason Tsiantos Dimitrios Tsogkas Panagiotis Nikolaos
Department of Electrical and Computer Engineering
University of Thessaly
Volos, Greece

***Abstract*—Open-source RISC-V cores provide a practical platform for studying how microarchitectural choices affect FPGA implementation quality. This work evaluates forwarding modifications to CV32E40P, a compact 32-bit in-order RISC-V core, by comparing the baseline design with no-multiplier-forwarding, no-ALU-forwarding, and combined no-ALU/no-multiplier-forwarding variants. The modifications remove selected forwarding paths and preserve architectural correctness through existing writeback availability and dependency stalls. All variants are evaluated using the same minimal FPGA wrapper, RTL regression flow, Vivado implementation methodology, and benchmark suite. Post implementation results show that the no-multiplier-forwarding variant meets a tighter 14 ns clock constraint and improves the estimated frequency from 66.8 MHz to 73.2 MHz, while keeping DSP and BRAM usage unchanged and adding only modest LUT and register overhead. Benchmark execution demonstrates that this timing gain translates into approximately 8.7% lower execution time for workloads that do not immediately consume multiplier results, including added control-flow, cryptographic, and network benchmarks. However, multiplier-heavy workloads can lose this benefit because the removed forwarding path introduces extra stall cycles. The results show that selective forwarding removal can be a useful FPGA timing-closure optimization, but its value depends on workload dependency patterns and must be evaluated using both implementation reports and benchmark-level cycle measurements.**

***Index Terms*—RISC-V, CV32E40P, FPGA, timing closure, forwarding, pipeline hazards, microarchitecture**

I. Introduction

Open-source instruction-set architectures and processor implementations have become increasingly important in computer architecture research, hardware education, and embedded-system prototyping. RISC-V is particularly suitable for such work because it defines an open and modular instruction-set architecture (ISA), allowing implementations to select a base integer ISA and combine it with optional standard extensions according to the needs of the target system [1]. This openness enables designers to move beyond black-box processor usage and study the relationship between ISA support, pipeline organization, register-transfer-level (RTL) structure, and physical implementation results.

The CV32E40P core is a representative example of this ecosystem. It is a compact 32-bit in-order RISC-V processor maintained by the OpenHW Group and originating from the RI5CY core developed within the PULP project [2]–[4]. The core targets embedded and Internet-of-Things-class systems, where small area, predictable execution, and efficient implementation are important design objectives. Its four-stage pipeline makes it sufficiently simple to analyze in an academic setting, while still containing realistic processor mechanisms such as instruction fetch, decode, execution, writeback, data-hazard handling, arithmetic units, and forwarding paths. These characteristics make CV32E40P a suitable case study for evaluating the implementation impact of a focused microarchitectural modification.

A central challenge in FPGA processor implementation is timing closure. A processor may be functionally correct at the RTL level but still fail to satisfy a requested clock period after synthesis, placement, and routing. In such cases, the achievable clock frequency is determined by the worst timing paths of the implemented design rather than by the nominal simplicity of the pipeline. Forwarding networks are a common example of logic that improves pipeline throughput but can also introduce implementation cost. By routing producer results directly to dependent consumer instructions, forwarding reduces the number of stall cycles caused by data hazards. However, it also adds multiplexers, control logic, and long inter-stage connections, which may become part of timing-critical paths after FPGA implementation.

This project studies this trade-off in the context of execute-stage result forwarding in CV32E40P. Multiplication support is part of the RISC-V integer multiplication and division extension, and the associated datapath can be more timing-sensitive than simple integer operations [1]. The baseline CV32E40P design includes forwarding mechanisms that allow dependent instructions to consume ALU or multiplier results before they are written back to the register file. The modified designs considered in this work remove multiplier forwarding, ALU forwarding, or both, and rely on pipeline hazard handling plus writeback-stage availability to preserve correct architectural behavior. The purpose of the modifications is to evaluate whether simplifying specific forwarding paths can improve FPGA timing closure.

The main contributions of this work are as follows. First, it frames the CV32E40P modifications as a controlled microarchitectural experiment comparing a baseline core and three forwarding variants under the same implementation methodology. Second, it evaluates the designs using post-implementation FPGA reports rather than only RTL-level inspection. Third, it quantifies the timing, resource, and power trade-offs using metrics such as worst-case slack, critical-path delay, LUT

usage, register usage, DSP usage, block RAM usage, and estimated on-chip power. Fourth, it performs benchmark-based evaluation of the different designs using a shared bare-metal RISC-V benchmark suite, allowing cycle-count differences to be compared across the baseline and modified cores. Finally, it discusses the distinction between improved timing closure and application-level performance: a design that meets a tighter clock constraint may still require benchmark-level cycle-count analysis to determine whether the removal of forwarding improves or degrades end-to-end execution time.

II. Background

A. RISC-V and Open Processor Design

RISC-V is an open ISA designed around a small base instruction set and a collection of optional extensions. For 32-bit implementations, the RV32I base integer ISA defines the fundamental integer instructions, registers, memory operations, branches, and jumps. Standard extensions add features such as integer multiplication and division, compressed instructions, atomics, and floating-point operations [1]. This modularity is important for hardware design because it allows an implementation to include only the architectural features required by its intended application domain, thereby controlling area, power, verification effort, and timing complexity.

Open-source RISC-V cores extend this flexibility from the ISA level to the implementation level. Designers can inspect and modify the RTL, synthesize the design for a target technology, and evaluate the effect of architectural or microarchitectural decisions. This property is especially valuable in education and research, where the goal is often to understand why a processor behaves in a certain way after implementation. Projects such as PULP and CORE-V have contributed several open processor cores and system components that support this style of experimentation [3], [5].

B. The CV32E40P Core

CV32E40P is a 32-bit in-order RISC-V processor core maintained by the OpenHW Group. It implements a four-stage pipeline consisting of instruction fetch, instruction decode, execute, and writeback stages [2]. The execute stage contains the main integer datapath elements, including the arithmetic logic unit, multiplier, divider, branch decision logic, and load/store address generation. The core supports the RISC-V integer ISA and selected optional extensions, and it also inherits features from the PULP/RI5CY family aimed at efficient embedded processing [4], [6].

The processor family was designed for energy-efficient embedded and low-power computing. Prior work on RI5CY and the PULP platform emphasizes the value of open, efficient RISC-V cores for near-threshold and Internet-of-Things applications, including support for digital-signal-processing-oriented extensions and tightly integrated embedded systems [4], [7]. Although this work targets an FPGA implementation rather than a near-threshold ASIC, the same design philosophy is relevant: small implementation choices in a compact in-order core can have measurable effects on timing, area, and resource utilization.

C. Pipeline Hazards and Forwarding

Pipelining improves processor throughput by overlapping the execution of multiple instructions. However, this overlap introduces hazards. A data hazard occurs when an instruction requires a value produced by an earlier instruction that has not yet been written back to the register file. A simple way to handle such hazards is to stall the pipeline until the value becomes available. Stalling is functionally correct but reduces instruction throughput.

Forwarding, also called bypassing, reduces this penalty by routing a produced value directly from a later pipeline stage or functional unit to a dependent instruction that needs it. In an in-order core such as CV32E40P, forwarding paths can reduce the number of bubbles inserted for common instruction sequences. The benefit is architectural performance potential, because fewer cycles may be required for dependent operations. The cost is additional datapath and control complexity. The design must compare source and destination registers, select between register-file data and forwarded data, and route values across pipeline stages. These paths can be especially relevant for FPGA timing because the physical delay after placement and routing includes both logic delay and routing delay.

ALU and multiplier forwarding are specific instances of this general trade-off. If an ALU or multiplication result can be consumed immediately by a following dependent instruction, forwarding can avoid or reduce a stall. However, the related datapath and selection logic can increase the complexity of the execute/decode interaction. Removing a forwarding path can simplify part of the timing-sensitive network, but it can also increase the number of cycles for instruction streams with dependent operations. Therefore, each optimization must be evaluated as a trade-off between implementation timing and potential cycle-count impact.

D. FPGA Timing Closure Metrics

FPGA implementation consists of synthesis, placement, and routing. Synthesis maps RTL to technology primitives, placement assigns those primitives to physical FPGA resources, and routing connects them using the device interconnect. The final timing of a design depends on both logic delay and routing delay. Consequently, timing behavior cannot be fully determined by RTL structure alone; it must be measured using implementation reports generated by the FPGA tool.

The most important timing metric used in this work is slack. For a given clock constraint, slack is the difference between the required arrival time and the actual arrival time of a signal along a timing path. Worst negative slack (WNS) reports the slack of the most critical path. A non-negative WNS indicates that all analyzed paths meet the specified timing constraint, whereas a negative WNS indicates a timing violation. Additional metrics, such as total negative slack, critical-path delay, route delay, and the distribution of path

slacks, help explain whether timing closure is limited by isolated critical paths or by a broader set of near-critical paths. Resource metrics such as LUTs, flip-flops, DSP blocks, and block RAMs complement the timing analysis by quantifying the area cost of the microarchitectural modification. Estimated power reports are also useful, but they should be interpreted as tool estimates under the selected activity assumptions rather than as measured board power [8], [9].

III. Proposed Microarchitectural Modifications

A. Baseline Forwarding Behavior

The baseline CV32E40P pipeline contains forwarding logic that allows results produced in later pipeline stages to be selected as operands for dependent instructions before those results are committed to the register file. This mechanism is important for maintaining pipeline throughput in short dependent instruction sequences. In the implementation considered in this work, the execute-stage forwarding interface is shared by several result sources. In particular, the execute stage drives an ALU forwarding write-data signal that may carry an ALU result, a multiplier result, or a CSR read value depending on the instruction currently in the execute stage. The decode-stage forwarding control then selects between register-file operands, execute-stage forwarded data, and writeback-stage forwarded data.

From a functional point of view, this organization is attractive because it reduces the number of stalls required after arithmetic instructions. However, from an implementation point of view, a shared forwarding network can increase selection complexity and routing pressure. The forwarded data path is 32 bits wide, is controlled by dependency-detection logic, and interacts with operand multiplexers and register-file update logic. On an FPGA, this type of network can be sensitive to placement and routing because its delay is not determined only by the number of logic levels, but also by the length and fanout of the routed nets [8], [9].

The baseline timing evidence used in this project should therefore be interpreted carefully. The worst reported path in the baseline design is not a pure multiplier-output path. Instead, it is located in the ID/register-file/operand-generation region of the core and is dominated by routing delay. This observation does not prove that the multiplier output alone is the critical net. Rather, it indicates that the region containing operand selection, forwarding, and register-file-related logic is timing-sensitive. Since the RTL shows that multiplier results are part of the shared execute-stage forwarding network, removing multiplier results from that network is a reasonable controlled experiment for testing whether timing pressure can be reduced.

B. Forwarding-Removal Variants

The implemented variants remove selected result sources from the direct execute-to-decode forwarding path. Fig. 2 illustrates the no-multiplier-forwarding design, which keeps ALU and CSR results eligible for execute-stage forwarding,

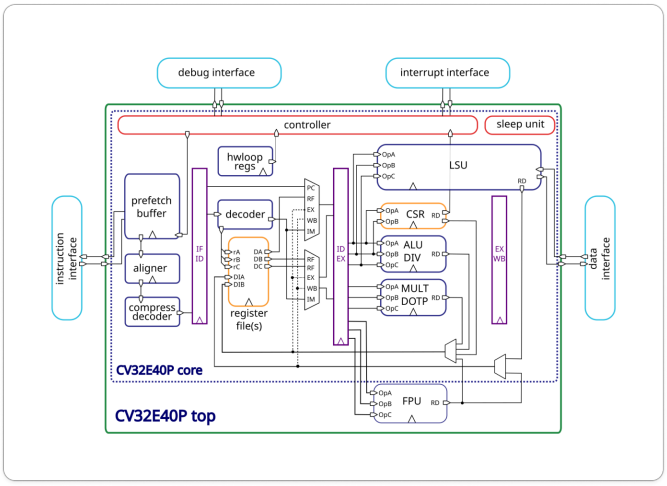


Fig. 1. Block diagram of the baseline design.

but multiplier results are no longer driven onto the same forwarding data path. Instead, a multiplier result is captured into the EX/WB pipeline register and is made available through the existing writeback path in the following cycle. Conceptually, this changes the multiplier-result dataflow from

$$\text{MUL result} \rightarrow \text{EX forwarding path} \rightarrow \text{ID operands} \quad (1)$$

into

$$\text{MUL result} \rightarrow \text{EX/WB register} \rightarrow \text{WB path.} \quad (2)$$

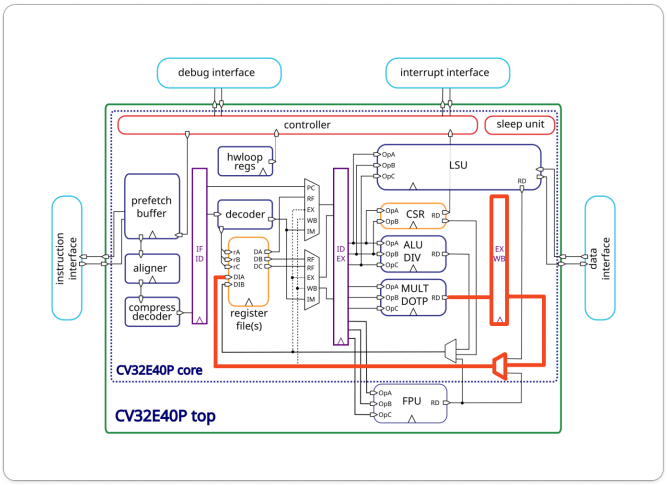


Fig. 2. The new execute to write-back multiplication path (marked in bold). The execute to register multiplication file path is removed.

As shown in Fig. 3, the no-ALU-forwarding variant applies the same principle to ALU results: immediate execute-stage ALU forwarding is removed, and dependent instructions are held until the result can be obtained through the later pipeline path. The combined no-ALU/no-multiplier-forwarding variant described in Fig. 4 removes both ALU and multiplier results from the immediate execute-stage forwarding network. In

all cases, the change is microarchitectural rather than architectural. The same instructions remain supported, while the control logic detects newly exposed dependencies and stalls the dependent instruction until the value is available from the safe path.

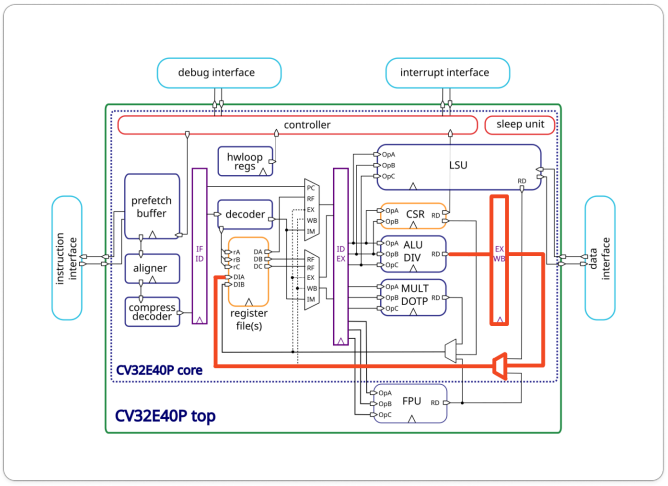


Fig. 3. The new execute to write-back ALU path. The execute to register file ALU path is removed.

The key implementation idea is to reduce the number of result sources participating in the execute-stage forwarding muxing and routing network. In the baseline design, the execute-stage forwarded write data can be selected from the ALU result, multiplier result, or CSR data. The variants remove one or more of these immediate forwarding cases and preserve correctness through registered writeback-stage availability and dependency stalls. This can reduce timing pressure, but it can also increase cycle count for workloads that contain tight producer-consumer dependencies on the removed forwarding source.

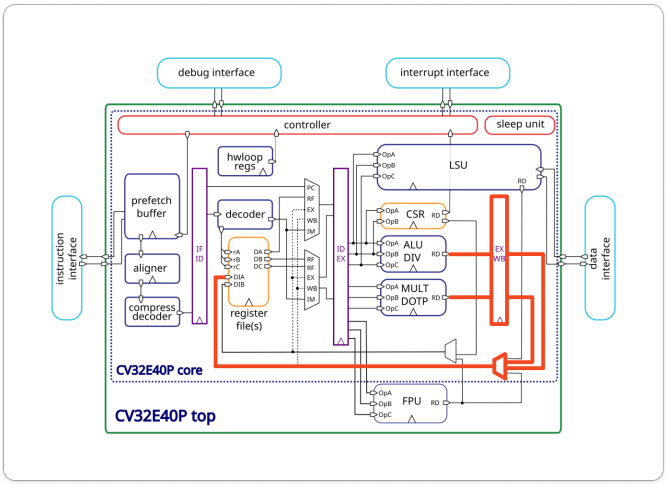


Fig. 4. The new execute to write-back ALU and multiplication paths. The execute to register file ALU and multiplication paths are removed.

C. Expected Timing and Performance trade-off

The expected benefit of this modification is improved FPGA timing closure. Removing ALU or multiplier results from the direct execute forwarding path can reduce mux complexity, fanout, and routing pressure in a region that already contains operand-selection and register-file-related logic. Because FPGA timing is strongly affected by routing after placement, even a localized RTL change can alter the physical implementation of nearby logic and reduce the delay of the final critical path. The expected cost is a possible cycle-count penalty for dependent instruction streams, because a consumer of a removed forwarding source may need to wait until the result reaches the writeback stage.

This distinction is important for interpreting the results. A higher achievable clock frequency or improved worst-case slack does not automatically imply higher application-level performance for every workload. Programs with few removed-forwarding dependencies may benefit from the improved timing potential with little or no cycle-count penalty. Conversely, code with tight ALU or multiplier producer-consumer dependencies may lose cycles due to the added stalls. Consequently, this work evaluates the modification primarily as a timing-closure optimization. A complete performance evaluation would additionally require benchmark execution, cycle-count measurement, and workload-dependent analysis.

D. Experimental Control

To isolate the effect of the microarchitectural changes, all designs are evaluated using the same top-level wrapper, target FPGA device, project-generation flow, and report-extraction methodology. The experiment compares four versions of the same core: the unmodified baseline, no-multiplier-forwarding, no-ALU-forwarding, and no-ALU/no-multiplier-forwarding. The main reported metrics are worst-case slack, critical-path delay, clock period, estimated frequency potential, FPGA utilization, and estimated on-chip power. This controlled setup allows the comparison to focus on the implementation impact of the forwarding modifications while keeping the surrounding FPGA system and tool flow constant.

IV. Experimental Setup

All variants are integrated into the same minimal ZedBoard-oriented wrapper and implemented with the same Vivado report flow. The generated reports used in this paper are post-implementation reports. The baseline, no-ALU-forwarding, and combined no-ALU/no-multiplier-forwarding variants are constrained at 15 ns, corresponding to 66.667 MHz. The no-multiplier-forwarding variant is constrained at 14 ns, corresponding to 71.429 MHz, because previous timing exploration showed that this variant could meet a tighter target. Therefore, comparisons involving WNS should be interpreted together with the clock constraint: the no-multiplier-forwarding result is not merely a slack comparison at the same period, but evidence that the design can satisfy a more aggressive timing target.

The RTL variants are kept as separate CV32E40P submodules: `cv32e40p_baseline`,

cv32e40p_no_mul_forwarding, cv32e40p_no_alu_forwarding, and cv32e40p_no_alu_mul_forwarding. The verification and implementation flow applies the same regression, synthesis, implementation, timing-report, utilization-report, and power-report procedures to each variant. This separation keeps the baseline and experimental RTL versions reproducible while allowing both per-variant and aggregate comparisons.

A. Minimal ZedBoard SoC Wrapper

The FPGA top level is the minimal SoC-style wrapper shown in Fig. 5. It is used primarily as a stable implementation context for timing, utilization, and power experiments, not as a complete board software stack. The wrapper instantiates one cv32e40p_top core, an instruction memory, and a data memory. Both memories are inferred as block RAMs using the ram_style = "block" attribute. In the default implementation configuration each memory contains 4096 32-bit words, i.e., 16 KiB for IMEM and 16 KiB for DMEM. The wrapper exposes only clock/reset inputs and LED status outputs; there is no AXI interconnect, UART, external memory controller, operating system support, or other peripheral subsystem in the evaluated top level.

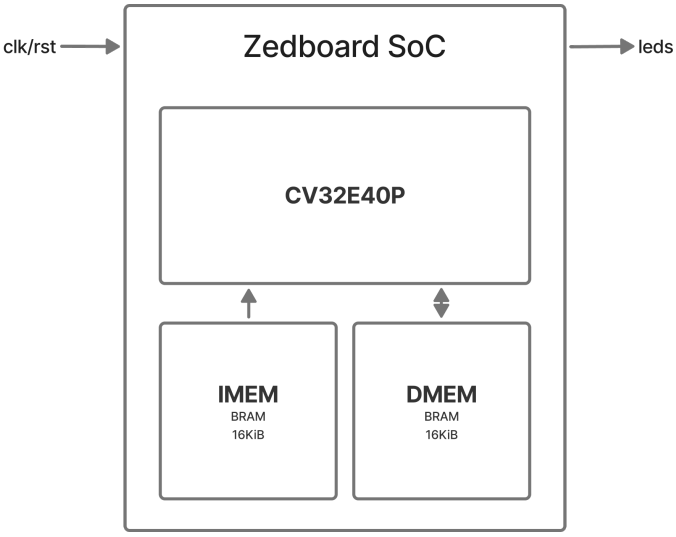


Fig. 5. Minimal ZedBoard SoC wrapper used for the implementation experiments. The core is connected to local IMEM and DMEM BRAMs and only simple board-level status outputs are exposed.

The CV32E40P instruction and data interfaces are connected to the local memories through a simple synchronous handshake. A memory request is granted immediately by tying instr_gnt to instr_req and data_gnt to data_req. Because the BRAMs are synchronous, the corresponding rvalid signal is generated by delaying the request by one clock cycle. The core produces byte addresses, while the BRAM arrays are word indexed; therefore the wrapper drops the two byte-offset bits and uses address bits [AW+1:2] to select a 32-bit word. For DMEM, the same low-bit mapping is used for data regions placed at addresses such as 0x0001_0000, so 0x0001_0000 maps to DMEM word 0 and 0x0001_0004 maps to word 1. The data memory is organized as four byte-wide arrays and uses the CV32E40P byte-enable signals for stores.

The core configuration is intentionally kept simple and constant across variants. The wrapper disables the PULP and cluster options by setting COREV_PULP=0 and COREV_CLUSTER=0; it also disables floating-point support through FPU=0 and ZFINX=0. Interrupts and debug requests are tied inactive, fetch_enable_i is tied high, and the boot address is fixed at 0x0000_0000. Clock gating is not used in the experiment: the local cv32e40p_clock_gate cell forwards clk_i directly to clk_o, pulp_clock_en_i is tied high, and scan clock-gate enable is tied low. These choices reduce board-level and SoC-level variability so that the reported differences are dominated by the CV32E40P forwarding changes.

B. RTL Functional Regression Flow

Before relying on implementation reports, each RTL variant is checked with a Python-driven Icarus Verilog simulation flow, summarized in Fig. 6. The regression driver assembles each test using riscv-none-elf-gcc, emits an instruction-memory image with riscv-none-elf-objcopy, and generates architectural reference dumps with a Python golden model. The selected CV32E40P SystemVerilog sources and the shared wrapper files are converted to Verilog with sv2v, compiled with iverilog, and executed with vvp [10], [11].

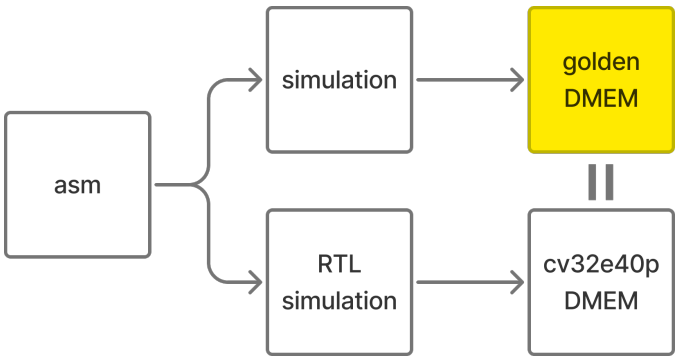


Fig. 6. RTL regression flow. The same assembly program is evaluated by the Python golden model and by the RTL simulation; the final data-memory contents are compared to detect functional regressions.

The RTL testbench receives the expected DMEM and register-file dump paths through plusargs, runs the design until the program writes the DONE word or the timeout is reached, and then dumps the final RTL-visible memories. Correctness is determined by comparing the final data memory against expected_dmem.mem and the final register file against expected_regfile.mem. This makes the simulation a regression test for the RTL edits: if a forwarding change preserves the architectural result, the final DMEM and register-

file state should match the Python reference for the same program.

C. Vivado Implementation and Report Flow

Implementation metrics are generated using a batch Vivado flow [8], [9]. The project-restoration Tcl script recreates the ZedBoard project for the selected experiment, while the report flow launches or opens the requested implementation stage and exports timing, utilization, power, and path-distribution reports. The comparison flow can run all variants and produce both per-experiment raw reports and aggregate summaries. The flow records whether a metric is post-synthesis or post-implementation; the measurements used in this report are interpreted as post-implementation FPGA results.

D. Benchmark Execution Flow

The benchmark flow is separate from the assembly correctness regression and targets bare-metal C workloads. Each benchmark is compiled as a freestanding RV32IM program with `riscv-none-elf-gcc` using `-march=rv32im`, `-mabi=ilp32`, `-O2`, `-ffreestanding`, `-nostdlib`, and the project linker script and startup/runtime files [12]. `riscv-none-elf-objcopy` extracts the `.text` and `.data` sections into separate IMEM and DMEM memory images, which are loaded by the benchmark testbench before running the RTL simulation.

During benchmark execution, the runtime writes memory-mapped status values that mark the beginning and end of the region of interest and report completion. The benchmark testbench prints a metric line containing `total_cycles`, `roi_cycles`, the return code, and a signature. The Python runner parses this line, assigns the clock period used for the corresponding RTL variant, and computes execution time as

$$T_{\mathrm{ROI}} = N_{\mathrm{ROI}} \cdot T_{\mathrm{clk}}, \tag{3}$$

where N_{ROI} is the reported region-of-interest cycle count and T_{clk} is the variant clock period. Thus the benchmark results combine RTL-level cycle counts with the clock constraint used for that variant; they should be interpreted together with the functional simulation status and the Vivado timing results.

V. Results

A. Timing Results

Table I summarizes the post-implementation timing metrics. All variants meet their requested constraints with positive WNS. The baseline reaches an estimated Fmax of 66.8 MHz at a 15 ns constraint. The no-multiplier-forwarding variant meets the tighter 14 ns constraint with 0.339 ns WNS and an estimated Fmax of 73.201 MHz. At the common 15 ns constraint, the no-ALU-forwarding variant improves WNS modestly to 0.114 ns, while the combined no-ALU/no-multiplier-forwarding variant gives the largest reported WNS, 0.630 ns, and the highest estimated Fmax among the 15 ns designs, 69.589 MHz.

TABLE I
POST-IMPLEMENTATION TIMING SUMMARY.

Variant	Period (ns)	WNS (ns)	Est. Fmax (MHz)	Crit. delay (ns)
Baseline	15.000	0.030	66.800	14.871
No MUL fwd.	14.000	0.339	73.201	13.440
No ALU fwd.	15.000	0.114	67.177	14.769
No ALU/MUL fwd.	15.000	0.630	69.589	14.154

The critical path locations also change between variants. The baseline critical path starts in the ID-stage ALU operand register and ends in the register file, with 12.773 ns of route delay out of a 14.871 ns datapath delay. The no-multiplier-forwarding critical path remains in the ID/register-file region but has lower route delay, 11.136 ns. The no-ALU-forwarding variant shifts the critical path toward multiplier-control/operand signals, and the combined variant reports a path from an ALU operator register to the EX/WB write-data register.

B. Utilization and Power Results

Table II reports the main FPGA resource counts. The no-multiplier-forwarding variant has the largest LUT and register overhead, increasing LUT usage from 4171 to 4314 and register usage from 2057 to 2100. The no-ALU-forwarding and combined variants have smaller LUT increases. DSP and block RAM usage remain unchanged across all variants, which is expected because the modifications change forwarding and control paths rather than the number of multiplier blocks or memories.

TABLE II
POST-IMPLEMENTATION UTILIZATION SUMMARY.

Variant	LUTs	FFs	DSPs	BRAM
Baseline	4171	2057	5	4.5
No MUL fwd.	4314	2100	5	4.5
No ALU fwd.	4200	2096	5	4.5
No ALU/MUL fwd.	4186	2092	5	4.5

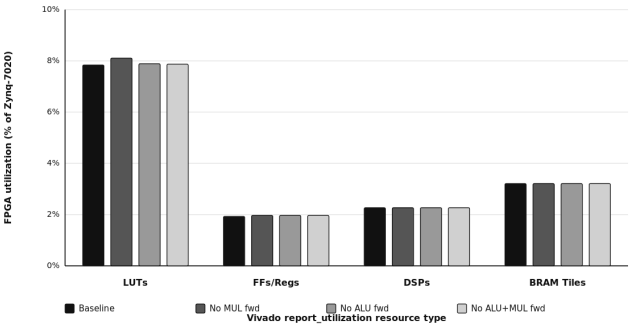


Fig. 7. Post-implementation utilization summary chart.

Estimated total on-chip power is nearly unchanged across the experiment: 0.120 W for the baseline, 0.121 W for no-multiplier-forwarding, and 0.119 W for both no-ALU-forwarding variants. Dynamic power is between 0.012 W and

0.014 W, while static power is 0.106 W for all variants. Therefore, the observed trade-off is primarily timing versus small logic/register overhead, not a large power change. Because these are Vivado estimates, they should be used for relative comparison under the same flow rather than as measured board-level power.

C. Critical Path Analysis

To identify the bottleneck in the baseline design, we extracted the 1000 slowest paths from the Vivado post implementation timing reports. The resulting delay histogram serves as a powerful diagnostic tool, illustrating not only the absolute worst case delays but also the overall timing pressure and localized routing congestion across the design.

In the baseline design, the delay distribution exhibits high concentration of paths smeared all the way out to the 14.9 ns mark. This wide, highly distributed shape is a signature of severe routing congestion, which was reportedly an issue from the vivado reports. It indicates that the physical routing resources in the affected region of the FPGA were saturated.

In contrast, the modified forwarding variants successfully eliminate these severe outliers by altering the congestion profile. The ”No MUL fwd” and combined ”No ALU+MUL fwd” architectures shift the distribution significantly to the left, containing their maximum delays well below 14.2 ns. Notably, the ”No ALU+MUL fwd” variant replaces the baseline’s smeared tail with sharp spike of over 800 paths around the 12 ns mark. When paths cluster tightly together at a specific timing mark rather than spreading out, it usually indicates that the routing is unobstructed

It is important to note, however, the presence of a smaller tail extending to the right of the primary 12 ns spike in the ”No ALU+MUL fwd” distribution. While the baseline design suffered from global routing saturation, these remaining trailing paths in the modified architecture indicate localized routing problems, likely associated with the newly introduced execution to writeback register control logic.

This data demonstrates that the lowering of routing congestion played a major role in the elimination of the critical path. By removing the forwarding logic, the modification did not only shorten a single isolated wire, but it also eliminated a localized congestion, in the form of dense operand multiplexers and control structures. Freeing up this physical area allowed the routing tool to optimize the remaining connections, thus achieving a greater clock frequency.

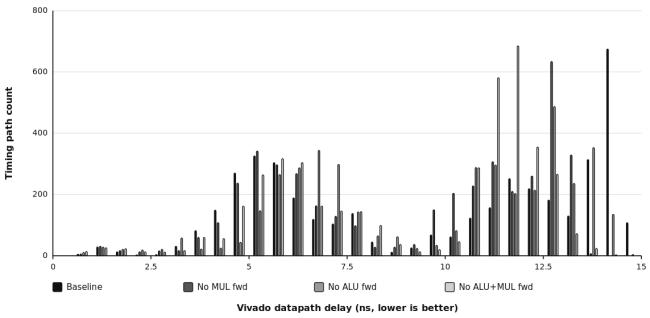


Fig. 8. Datapath delay histogram.

D. Theoretical Performance and Execution Time Analysis

To evaluate the theoretical impact of the microarchitectural modifications on overall performance, we can model the trade-off between the achieved clock frequency improvement and the potential cycles-per-instruction (CPI) penalty introduced by the removed forwarding paths. For this analysis we assume that the baseline CPI is equal to 1.

Moving from the baseline frequency of 66.6 MHz to the improved 73.2 MHz yields a direct performance benefit for instruction streams unaffected by newly exposed data hazards. Specifically, in an ideal workload with a 0.0% multiplier dependency rate the core leverages the higher clock speed without any stall penalties, resulting in a maximum theoretical execution time decrease of 9.02%.

However, because removing the direct forwarding path introduces a one-cycle stall for every dependent instruction, this CPI penalty gradually erodes the frequency-driven performance gains as the dependency rate increases. The performance model establishes a strict break-even cutoff at a dependency rate of 9.9%.

Ultimately, this theoretical model demonstrates that the forwarding optimization is highly beneficial for the majority of general purpose embedded workloads, provided the specific application domain does not rely heavily on unbroken chains of dependent multiplication instructions.

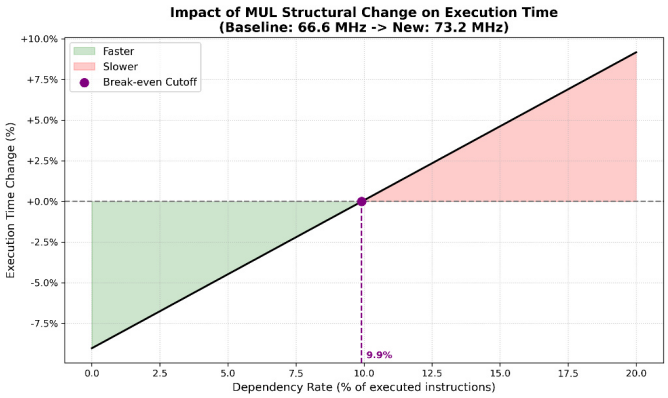


Fig. 9. Execution time based on dependency rate.

To systematically evaluate whether the removal of a forwarding path is justified, we must model the mathematical relationship between achievable clock frequency and the pipeline stalls introduced by unmitigated data hazards. The primary condition for the modified architecture to yield an overall performance improvement is that its total execution time must be strictly less than that of the baseline design ($T_{new} < T_{base}$). Execution time (T) is defined by the product of the Instruction Count (IC), the Cycles Per Instruction (CPI), and the clock period, or inversely, the clock frequency (f):

$$IC \times CPI_{new} \times \frac{1}{f_{new}} < IC \times CPI_{base} \times \frac{1}{f_{base}} \quad (4)$$

Because the microarchitectural modification does not alter the ISA or the compiler output, the Instruction Count (IC) remains constant and cancels out of the inequality. Rearranging the remaining terms to isolate the required new frequency yields:

$$f_{new} > f_{base} \times \frac{CPI_{new}}{CPI_{base}} \quad (5)$$

The new CPI is simply the baseline CPI plus the additional stall penalty incurred by the removed forwarding path. Assuming a one-cycle penalty for every dependent instruction, CPI_{new} expands to $CPI_{base} + (1 \times \text{Dependency Rate})$. Substituting this into the inequality defines the absolute break-even frequency limit. The exact clock speed the FPGA router must achieve to neutralize the CPI penalty for a given dependency rate:

$$f_{break_even} = f_{base} \times \left(\frac{CPI_{base} + \text{Dependency Rate}}{CPI_{base}} \right) \quad (6)$$

This mathematical relationship is visually mapped in the provided plot, which depicts the required target frequency across a dependency rate spanning from 0% to 50%. The central blue line represents the theoretical break-even boundary where $T_{new} = T_{base}$.

By plotting our specific implementation results onto this model, we can visualize the precise boundaries of our microarchitectural trade-off. Our baseline of the CV32E40P implementation operates at 66.6 MHz with an assumed base CPI of 1.0. The modified design, lacking multiplier forwarding, successfully routed at 73.2 MHz. Tracing the 73.2 MHz horizontal line to the break-even boundary intersects exactly at the 9.9% dependency mark. This visually confirms that as long as the workload's multiplier dependency rate remains below 9.9%, the 73.2 MHz implementation will reside securely within the Performance Gain Zone.

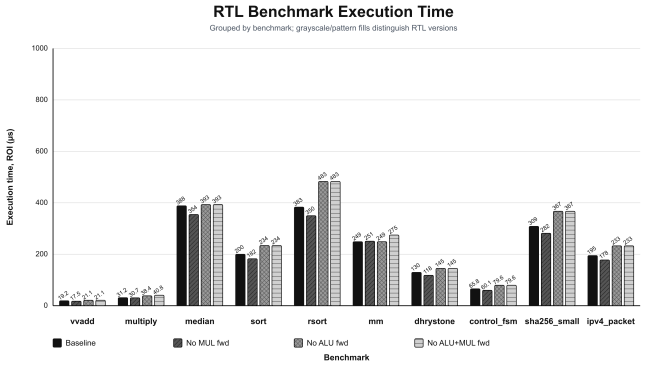


Fig. 11. Benchmarks of realistic bare-metal benchmarks.

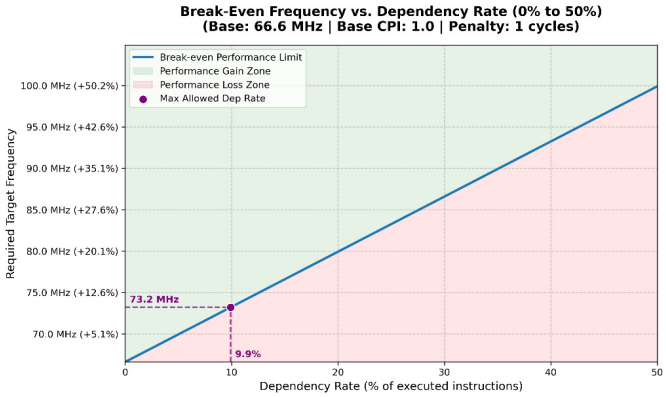


Fig. 10. Break-even Frequency plot.

E. Application-Level Performance Impact

To evaluate the practical implications of the microarchitectural modifications, the theoretical performance model is compared against empirical execution times from a suite of realistic bare-metal benchmarks.

For general-purpose integer workloads (`vvadd`, `median`, `sort`, `rsort`, `dhrystone`, `sha256_small`, `control_fsm` and `ipv4_packet`), the instruction streams do not trigger the newly exposed structural multiplier hazards. Consequently, their multiplier dependency rate is 0.0%. Because the removal of the forwarding path introduces zero stall cycles for these applications, they fully leverage the clock frequency increase from 66.6 MHz to 73.2 MHz. This translates to the maximum theoretical execution time reduction of approximately 9.0%, placing these benchmarks firmly within the optimal region of the performance gain zone.

The `multiply` benchmark represents workloads with moderate multiplier utilization, exhibiting a measured dependency rate of 7.7%. While the absent forwarding path introduces a measurable number of penalty cycles, this rate remains below the break-even threshold. Consequently, the 73.2 MHz clock speed sufficiently offsets the CPI penalty, resulting in a net overall speedup of 2.0%.

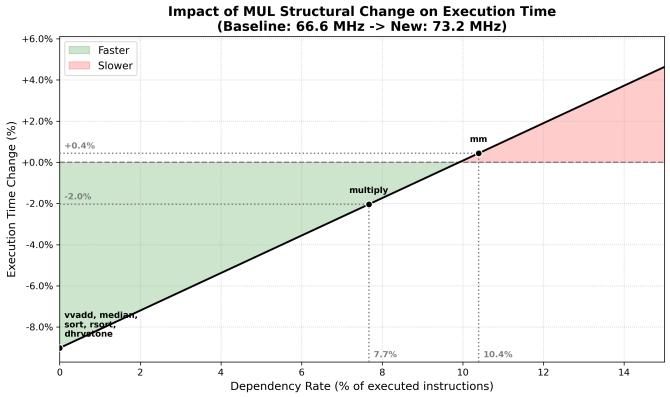


Fig. 12. Execution time changes for Benchmarks.

The matrix multiplication (`mm`) benchmark demonstrates the penalty of exceeding the theoretical break-even limit. By heavily utilizing the multiplier, it reaches a dependency rate of 10.4%, crossing into the performance loss zone. In this scenario, the accumulation of pipeline stalls outpaces the benefits of the frequency improvement, resulting in a net execution time penalty of +0.4%.

1) Domain-Specific Benchmark Coverage: In addition to the standard integer kernels, three domain specific workloads were added to evaluate whether the proposed no-multiplier-forwarding design benefits realistic embedded application classes rather than only synthetic tests. The `control_fsm` benchmark models control dominated IoT firmware using branch heavy state transitions and event processing. The `sha256_small` benchmark represents cryptographic hashing, dominated by shifts, XORs and additions. The `ipv4_packet` benchmark models network packet processing through IPv4 header validation, checksum computation, protocol classification and a small rewrite-style data path. These domains are relevant to embedded processors because they stress control flow, integer ALU operations, and memory accesses while avoiding chains of immediately dependent multiplication instructions.

The measured RTL results confirm this expectation. For all three domain-specific benchmarks, the no-multiplier-forwarding core produces the same region of interest cycle count and the same correctness signature as the baseline. Therefore, the proposed design does not pay a CPI penalty on these workloads. Table III summarizes the measured execution times.

TABLE III
DOMAIN-SPECIFIC BENCHMARK EXECUTION TIME FOR THE BASELINE AND NO-MULTIPLIER-FORWARDING DESIGNS.

Benchmark	Baseline (μs)	No MUL fwd (μs)	Reduction
`control_fsm`	65.81	60.05	8.7%
`sha256_small`	308.62	281.64	8.7%
`ipv4_packet`	194.86	177.83	8.7%

These results support the workload dependent interpretation of the optimization. The proposed design performs better in domains where the instruction stream is branch, ALU, or memory dominated but does not repeatedly consume multiplier results in the immediately following instruction. In such cases, the forwarding removal improves implementation timing without increasing the benchmark cycle count, so application-level execution time decreases.

The `multiply` benchmark represents workloads with moderate multiplier utilization, exhibiting a measured dependency rate of 7.7%. While the absent forwarding path introduces a measurable number of penalty cycles, this rate remains below the break-even threshold. Consequently, the 73.2 MHz clock speed sufficiently offsets the CPI penalty, resulting in a net overall speedup.

The matrix multiplication (`mm`) benchmark demonstrates the penalty of exceeding the theoretical break-even limit. By heavily utilizing the multiplier, it reaches a dependency rate of 10.4%, crossing into the performance loss zone. In this scenario, the accumulation of pipeline stalls outpaces the benefits of the frequency improvement, resulting in a small net execution-time penalty.

VI. Discussion

The experimental results show that removing a forwarding path can be an effective FPGA timing closure technique, but only when the affected dependency class is not dominant in the target workload. The no-multiplier-forwarding variant is the clearest example of this trade-off. It meets a tighter 14 ns constraint, improves the estimated operating frequency from 66.8 MHz to 73.2 MHz, and reduces the implemented critical path delay from 14.871 ns to 13.440 ns. These timing gains are obtained without changing the ISA or the correctness visible behavior of the benchmarks, as the modified core still produces matching signatures for all tested programs.

The key observation is that the timing improvement does not come from reducing the number of executed instructions or cycles in the common case. Instead, it comes from simplifying part of the execute stage forwarding network and allowing the FPGA implementation tools to route the operand generation and register file region more effectively. For workloads such as `vvadd`, `median`, `sort`, `rsort`, `dhrystone`, `control_fsm`, `sha256_small`, and `ipv4_packet`, the no-multiplier-forwarding design preserves the same region-of-interest cycle count as the baseline. Their execution time improvement is therefore a direct consequence of the shorter clock period.

The benchmark results also illustrate the limitation of the optimization. When a program contains immediate consumers of multiplication results, the removed forwarding path must be replaced by stall cycles. The `multiply` benchmark remains slightly faster because its additional cycles are still below the break-even point implied by the frequency improvement. The `mm` benchmark, however, crosses this threshold and becomes slightly slower. This confirms that the proposed design is not a universally faster core; it is a workload-dependent implementation trade-off that is most attractive for embedded

workloads dominated by control flow, integer ALU operations, memory access, and infrequent dependent multiplication.

The ALU-forwarding variants further reinforce this point. Although removing ALU forwarding can improve slack at the implementation level, ALU dependencies are much more common in ordinary integer code than multiplier dependencies. As a result, the no-ALU-forwarding and combined variants can introduce cycle-count penalties across a wider range of benchmarks. This makes the no-multiplier-forwarding variant the most balanced design point in this study: it provides a measurable timing benefit while preserving cycle counts for most tested workloads and requiring only modest resource overhead. The measured LUT increase from 4171 to 4314 and register increase from 2057 to 2100 are small relative to the size of the implemented system, while DSP and BRAM usage remain unchanged.

There are several limitations to the evaluation. First, the FPGA power results are Vivado estimates rather than physical board measurements, so they are useful mainly for relative comparison within the same flow. Second, the benchmark suite is intentionally small and deterministic to fit the bare-metal RTL simulation environment; it does not represent every possible embedded application. Third, the reported performance combines RTL cycle counts with implementation-derived clock periods, so it evaluates the expected effect of running each design at its achieved clock target rather than measuring execution on a complete deployed SoC. Future work could extend the benchmark suite, evaluate additional FPGA devices or placement seeds, measure board level power, and explore selective forwarding policies that preserve high value ALU or multiplier cases while reducing the timing pressure of the full forwarding network.

VII. Conclusion

This work evaluated timing-oriented forwarding modifications to the CV32E40P RISC-V core on an FPGA implementation flow. The study compared the baseline core against no-multiplier-forwarding, no-ALU-forwarding, and combined no-ALU/no-multiplier-forwarding variants using post-implementation timing, utilization, power, and RTL benchmark results. The objective was to determine whether removing selected execute-stage forwarding paths could reduce timing pressure without violating architectural correctness.

The results show that multiplier forwarding is a profitable target for this design. The no-multiplier-forwarding variant meets a 14 ns timing constraint and improves the estimated frequency from 66.8 MHz to 73.2 MHz, while keeping DSP and BRAM usage unchanged and introducing only modest LUT and register overhead. Functional and benchmark simulations confirm that the modified core preserves correctness for the evaluated programs.

At the application level, the benefit depends on the dependency structure of the workload. Benchmarks without immediate multiplier-result consumers maintain the same region-of-interest cycle count as the baseline and therefore achieve an execution-time reduction of approximately 8.7% from the shorter clock period. This includes the added domain-specific benchmarks for control-oriented firmware, cryptographic hashing, and network packet processing. Multiplier heavy workloads show the opposite side of the trade-off: moderate dependency rates can still benefit slightly, while the matrix multiplication benchmark incurs enough extra cycles to become slightly slower.

Overall, the project demonstrates that forwarding logic should not be treated as an unqualified performance feature in FPGA processor implementations. Forwarding improves CPI for dependent instruction streams, but it can also worsen physical timing through muxing, fanout, and routing pressure. For the evaluated CV32E40P configuration, selectively removing multiplier forwarding provides the best balance between timing closure and application-level performance, especially for embedded workloads with limited dependent multiplication. The main lesson is that microarchitectural optimizations must be judged using both implementation reports and benchmark-level cycle measurements, since either source of evidence alone can give an incomplete view of the final design trade-off.

REFERENCES

[1] RISC-V International, *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*, RISC-V International, 2024. [Online]. Available: https://riscv.org/technical/specifications/

[2] OpenHW Group, *CORE-V CV32E40P User Manual*, OpenHW Group, 2024, accessed: 2026-06-17. [Online]. Available: https://docs.openhwgroup.org/projects/cv32e40p-user-manual/

[3] ——, "CORE-V CV32E40P RISC-V IP," GitHub repository, 2024, accessed: 2026-06-17. [Online]. Available: https://github.com/openhwgroup/cv32e40p

[4] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, Oct. 2017.

[5] D. Rossi, I. Loi, A. Pullini, F. Conti, and L. Benini, "PULP: A parallel ultra low power platform for next generation iot applications," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, 2015, pp. 1–39.

[6] P. D. Schiavone, D. Rossi, A. Pullini, E. Flamand, F. K. Gürkaynak, and L. Benini, "Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017, pp. 1–8.

[7] D. Rossi, F. Conti, M. Eggiman, S. Mach, A. D. Mauro, M. Guermandi, G. Tagliavini, A. Pullini, I. Loi, J. Chen, E. Flamand, and L. Benini, "Energy-efficient near-threshold parallel computing: The PULPv2 cluster," *IEEE Micro*, vol. 37, no. 5, pp. 20–31, Sep. 2017.

[8] Xilinx, *Vivado Design Suite User Guide: Implementation*, Xilinx, 2022, accessed: 2026-06-17. [Online]. Available: https://docs.xilinx.com/r/en-US/ug904-vivado-implementation

[9] ——, *Vivado Design Suite User Guide: Design Analysis and Closure Techniques*, Xilinx, 2022, accessed: 2026-06-17. [Online]. Available: https://docs.xilinx.com/r/en-US/ug906-vivado-design-analysis

[10] Z. Yedidia, "sv2v: SystemVerilog to Verilog Conversion," GitHub repository, accessed: 2026-06-22. [Online]. Available: https://github.com/zachjs/sv2v

[11] Icarus Verilog Project, "Icarus Verilog," Project website, accessed: 2026-06-22. [Online]. Available: https://steveicarus.github.io/iverilog/

[12] RISC-V Collaboration, "RISC-V GNU Compiler Toolchain," GitHub repository, accessed: 2026-06-22. [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain