

From Code to RTL: A Methodology for Analyzing the HLS Hardware Generation Process

Henrique Gregory Gimenez¹, Edson Toshimi Midorikawa¹,
Felipe Valencia de Almeida¹, Liria Matsumoto Sato¹

¹ Department of Computer Engineering and Digital Systems
Polytechnic School – University of São Paulo (USP)

{henriquegregory, emidorik, fvalencia, liria.sato}@usp.br

Abstract. *High-Level Synthesis (HLS) abstracts the hardware accelerator design process, but evaluating the generated RTL often becomes a “black-box” challenge for developers. To address this issue, we propose a systematic methodology to track and analyze the translation of C/C++ algorithms into hardware (RTL). Our approach integrates synthesis via Xilinx Vitis/Vivado 2024.2 with datapath mapping using Intel Quartus and behavioral waveform validation via ModelSim. By applying this methodology to different algorithms, we mapped how the compiler translates constant complexity codes into linear pipelines and iterative logic into multi-stage Finite State Machines (FSMs). To evaluate the architectural results extracted by our method, we conducted performance evaluations on a PYNQ-Z2 board. Experimental results show that while the hardware reduces clock cycles, the absolute speedup is constrained by communication overhead. We conclude that the proposed methodology provides a replicable pathway for developers to understand the HLS-to-RTL mapping.*

1. Introduction

The growing demand for high-performance computing (HPC) has driven the use of hardware accelerators [Vaithianathan et al. 2023]. Unlike CPUs, which process instructions sequentially and generically, reconfigurable architectures such as FPGAs can offer greater efficiency, significantly reducing execution time and energy consumption compared to purely software-based approaches [Vaithianathan et al. 2023]. However, the traditional register-transfer level (RTL) design flow using hardware description languages (HDLs) remains a costly, time-consuming, and error-prone process [Xiong et al. 2025].

High-Level Synthesis (HLS) stands out in this scenario, emerging as an alternative to mitigate these costs. HLS tools allow the design of digital circuits from behavioral specifications described in high-level programming languages [Advanced Micro Devices, Inc. 2026]. These tools have aided system development, increasing productivity and abstraction when designing complex circuits [Curzel 2024].

Nevertheless, this abstraction creates a lack of architectural visibility: evaluating the tools’ efficiency and ensuring adequate Power, Performance, and Area (PPA) requires understanding how C/C++ semantics are translated into RTL [Cong et al. 2022]. As subtle changes in the code can significantly alter the generated hardware [Herklotz et al. 2021], the lack of standardized procedures to systematically open this “black box” leaves developers without a clear path to correlate the high-level behavior

with its hardware implementation. As discussed in Section 2, to the best of our knowledge, the literature lacks detailed methodological approaches on how to track and analyze the translation of high-level code into internal RTL structures by commercial HLS compilers.

Motivated by this gap, this work proposes a structured methodology for mapping hardware generation via C/C++ HLS using Xilinx Vitis and Vivado 2024.2. Unlike existing approaches focused on optimization directives or automated verification, our methodology directly correlates high-level C/C++ variables with their datapath and FSM structures via reverse engineering. Our main contributions include: (i) A methodology for mapping the RTL generated by the Vitis HLS compiler, utilizing Intel Quartus to track datapath and finite-state machines (FSMs), and ModelSim to analyze behavioral waveforms; (ii) The validation of the proposed method through architectural case studies; (iii) The behavioral classification derived from our methodology, illustrating how $O(1)$ algorithms are converted into linear pipelines and iterative algorithms into multi-stage FSMs (including the coupling of multiple loops); (iv) Experimental validation on the PYNQ-Z2 board, comparing the execution time and clock cycles between software (CPU) and hardware (FPGA) processing to support our structural findings.

2. Related Works

Recent studies highlight that a Quality of Results (QoR) gap still exists between HLS-generated circuits and manual RTL designs [Lahti and Hämäläinen 2025]. Understanding the root of this gap is essential for HPC developers to maximize hardware efficiency.

To mitigate HLS limitations and optimize performance, recent literature has increasingly focused on unveiling the internal mechanisms of commercial compilers. For instance, studies have reverse-engineered optimization directives (pragmas) in AMD-Xilinx Vitis HLS to empirically evaluate the impact of techniques like loop unrolling, pipelining, and flattening on latency and resource utilization [Koronis et al. 2025]. Similarly, methodologies have been proposed to overcome commercial restrictions of HLS tools, enabling the synthesis of arbitrary-scale designs for different hardware targets [Carrion Schafer and Sathe 2025]. Furthermore, frameworks have been developed to reverse-engineer the mapping between C-level variables and RTL registers, aiding developers in correlating high-level behavior with its hardware implementation [Abderehman et al. 2021].

Alongside architectural exploration, verifying and debugging HLS-generated RTL remains a critical bottleneck. Recent advancements propose FPGA-accelerated co-simulation methods, such as the *Hercules* framework, which provides cycle-accurate visibility and high-speed verification to trace HLS bugs [Xu et al. 2025]. Additionally, the literature proposes lightweight partial HLS approaches for the automatic correction of faulty RTL designs while preserving manual optimizations [Alizadeh and Shiroei 2023]. The integration of Large Language Models (LLMs) into the HLS workflow has also gained traction, prompting the creation of comprehensive benchmarking suites to evaluate the PPA of AI-generated hardware codes [Khan et al. 2026].

Unlike prior work focused on optimization, verification, or AI models, this paper investigates commercial HLS compiler internals by mapping how algorithms translate into FSMs and datapaths.

3. An HLS-to-RTL Translation Analysis Methodology

Building upon the need to understand the internal mechanisms of commercial compilers highlighted in the previous section, we propose an approach to systematically characterize these compiler mechanisms. This section details our methodology for understanding the translation of the high-level C/C++ code into RTL datapaths and control units. By providing a correlation between software code and generated hardware, this methodology aims to help HPC engineers with the possible insights necessary to optimize hardware accelerator designs.

3.1. Methodology Overview

Our methodology consists of an iterative procedure to reverse-engineer the HLS-to-RTL translation. Rather than a linear checklist, this process operates as an interactive feedback loop. Figure 1 illustrates our three-phase methodology. Phase 1 maps algorithm logic to hardware I/O and modules to establish a baseline. Phase 2 formulates a structural mapping hypothesis via preliminary AI scanning and detailed signal isolation. Finally, Phase 3 validates this hypothesis through behavioral simulation. As highlighted, any mismatch triggers an iterative loop back to structural analysis to refine the register and variable mapping, until the hypothesis converges or the iteration limit defined is reached.

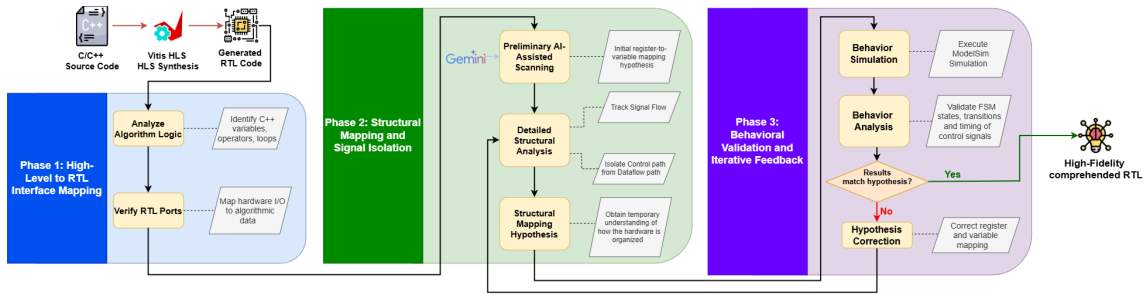


Figure 1. Proposed Iterative Methodology for HLS-Generated RTL Analysis

3.2. Tools and Evaluation Environment

We utilized Xilinx Vitis and Vivado 2024.2 for High-Level Synthesis (HLS), RTL generation, bitstream compilation, and extracting resource and power metrics. To optimize the analysis of complex RTL structures, Gemini Pro 3.1 assisted in the initial code triage and variable mapping. Subsequently, Intel Quartus Prime 20.1 (RTL Viewer) was employed for its clearer schematic representation, essential for tracking datapaths and finite-state machines (FSMs). ModelSim 2020.1 validated dynamic control signals via waveform simulation. Experimental validation and performance benchmarking were conducted on the Xilinx PYNQ-Z2 board (Zynq-7000 SoC), featuring an ARM Cortex-A9 processor (650 MHz) and FPGA logic (100 MHz). Automated benchmarking was executed via a Python-based Jupyter environment, using 10,000 random input iterations to compare CPU and FPGA processing efficiencies in terms of execution time and clock cycles.

3.3. Proposed Structural and Behavioral Mapping Methodology

The three main phases of our methodology are:

Phase 1: High-Level to RTL Interface Mapping. The first phase establishes the correlation between the high-level algorithm logic and the top-level hardware signal ports. It begins by thoroughly identifying the variables used and the operations performed (such as additions, comparisons, and loops) within the C/C++ source code. With this baseline, we verify the input and output ports and the modules in the generated RTL to understand how the HLS tool wrapped the algorithm into a hardware module.

Phase 2: Structural Mapping and Signal Isolation. Once the baseline is set, we utilize a hybrid approach combining artificial intelligence and visual schematic analysis. Initially, we use Gemini Pro 3.1 to perform a preliminary triage of the generated HDL code. Since HLS-generated RTL can be highly verbose and difficult to navigate, the AI assists in scanning the files to quickly identify naming conventions, isolate candidate state registers, and map generated signals back to their corresponding high-level C/C++ variables. This LLM-assisted inspection accelerates the reverse-engineering process by narrowing down the components that require manual verification in the schematic viewer.

Following this initial AI triage, a crucial step is identifying the Control Unit and isolating its signals from the datapath components. In the generated HDL code, the Control Unit can be identified by locating state transition logic, which often manifests as switch-case structures or dedicated state registers. In practice, this means distinguishing the mathematical and logical operations carrying the actual algorithmic data (the datapath) from the signals orchestrating when these operations occur (the control path). Using the Quartus RTL Viewer tool, the researcher tracks multiplexer selection lines, write-enable pins on registers, and the specific outputs of the identified FSM blocks. By visually separating the Control Unit's state registers from the data accumulators or multipliers, we can accurately map the exact pipeline stages or iteration cycles to the hardware. The final output of Phase 2 is a detailed structural mapping. By isolating these signals, we derive the Control Unit's state transition diagram and the corresponding datapath schematic, which serve as the theoretical models for the next phase. The resulting state transition diagrams and datapaths presented in this work are visual abstractions drawn by the authors. They were derived directly from the schematics obtained via the Quartus RTL Viewer and State Machine Viewer tools.

Phase 3: Behavioral Validation and Iterative Feedback. To validate the structural mapping (the state diagrams and datapaths) obtained in Phase 2, we simulate the generated RTL's dynamic behavior. To achieve this, we employ manually written HDL testbenches, which are the simulation environments used to apply inputs (stimuli) to the generated hardware and observe its responses. Using ModelSim, these HDL testbenches are designed to reproduce the same stimuli from the C/C++ testbench used in Vitis. The expected behavior is then evaluated by cross-referencing the dynamic waveform results from the simulation with the structural models established in Phase 2. We stimulate the RTL module with specific input data and verify if intermediate calculations, register updates, and the assertion of top-level control signals occur at the exact pipeline stages or iteration cycles anticipated.

Importantly, this methodology is iterative. Whenever discrepancies arise between the hypothesized hardware structure and the observed behavior, such as unexpected control signal timing or mismatched signal routes, the researcher returns to the structural analysis to refine the correlations between high-level variables and RTL components.

The loop converges when three conditions hold simultaneously: the state sequence in the waveform matches the Phase 2 state diagram, the intermediate register updates occur in the cycles predicted by the datapath, and the top-level outputs match the expected values of the C/C++ testbench. To prevent endless refinement, the maximum number of iterations is defined by the analyst according to the complexity of the design.

4. Methodology Application and Case Studies

To validate the proposed methodology, we selected algorithms of two primary behavioral categories:

- **Deterministic Algorithms:** These algorithms possess closed-form analytical solutions with $O(1)$ time complexity. They result in hardware with simplified Finite State Machines (FSMs) that do not require complex control logic.
- **Purely Iterative Algorithms:** These algorithms do not have a closed-form solution and require iterative calculations. Consequently, HLS tools must generate circuits with dedicated, multi-stage FSMs to control loop execution and stopping conditions.

4.1. Circuit Analysis

For the deterministic algorithms, we evaluated the sum of n terms of an arithmetic progression to represent this category. For iterative algorithms, we analyzed the factorial, integer square root, and a combined factorial-square root. Small kernels were chosen to provide a clear understanding of the HLS-to-RTL translation process. Although not evaluated in this paper, the proposed methodology can be extended to more complex or recursive algorithms. All source codes and generated RTL files are publicly available in our GitHub repository: <https://github.com/henriregreg/analysis-of-the-hardware-generation-process-via-hls>. The repository also lists the complete module hierarchy and the full names of all generated modules for each case study. Vitis prefixes generated module names with the top-level function name. For readability, we omit this prefix when referring to submodules.

4.1.1. Arithmetic Progression Sum

Given an arithmetic progression with first term a_1 and common difference r , the sum of the first n terms is given by the Equation 1.

$$S_n = a_1 + (a_1 + r) + \cdots + [a_1 + (n - 1) \cdot r] = \sum_{k=1}^n [a_1 + (k - 1) \cdot r] \quad (1)$$

Applying Phase 1 of our methodology, we initially analyzed the C++ algorithm that computes the sum described by Equation 1. We identified the algorithmic variables (a_1 , r , and n). Upon synthesizing this code with Vitis, we observed how the tool established the hardware interface: rather than implementing a loop, the tool inferred an AP sum and algebraically manipulated the expression to optimize resource usage, applying the following closed-form Equation 2.

$$S_n = a_1 + (n - 1) \cdot (a_1 + r) + \frac{(n - 1) \cdot (n - 2) \cdot r}{2} \quad (2)$$

Moving to Phase 2, our detailed structural analysis identified that Vitis HLS generated three RTL modules: the top-level `arith_prog_sum`, which implements the main logic of the AP sum, and two 32-bit multipliers, one signed and one unsigned.

The equivalent circuit for the code is represented by a 6-stage pipeline: an initial cycle, four execution cycles, and a final cycle. Thus, the `arith_prog_sum` module is responsible for calculating the AP sum, being controlled by a state machine with six states (represented by the parameters from `ap_ST_fsm_state1` to `ap_ST_fsm_state6`).

The module exposes the standard HLS block-level control interface (`ap_clk`, `ap_rst`, `ap_start`, `ap_done`, `ap_idle`, `ap_ready`), the inputs `a1`, `r` and `n`, and the output `ap_return`, which holds the sum.

During the initial idle state, the circuit precomputes $(n - 1)$ and $(n - 2)$. Upon `ap_start` activation, execution cycles 1 through 4 sequentially process the intermediate multiplications and additions using the dedicated signed and unsigned modules. Finally, the last state computes the complete closed-form sum, routing it to the `ap_return` output, and asserts the `ap_done` and `ap_ready` signals (the output defaults to 0 if $n < 0$). Figure 2 shows the control unit (an idle state, four execution states and a final state), and Figure 3 shows the datapath, where the three multipliers are separated by registers that delimit the pipeline stages. Some signals are omitted for clarity.

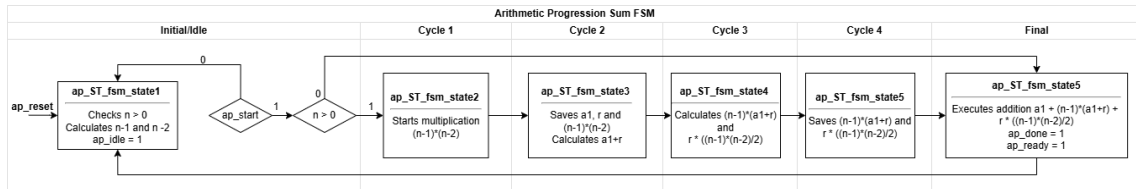


Figure 2. State transition diagram of the Arithmetic Progression Sum Control Unit (modeled from Quartus State Machine Viewer)

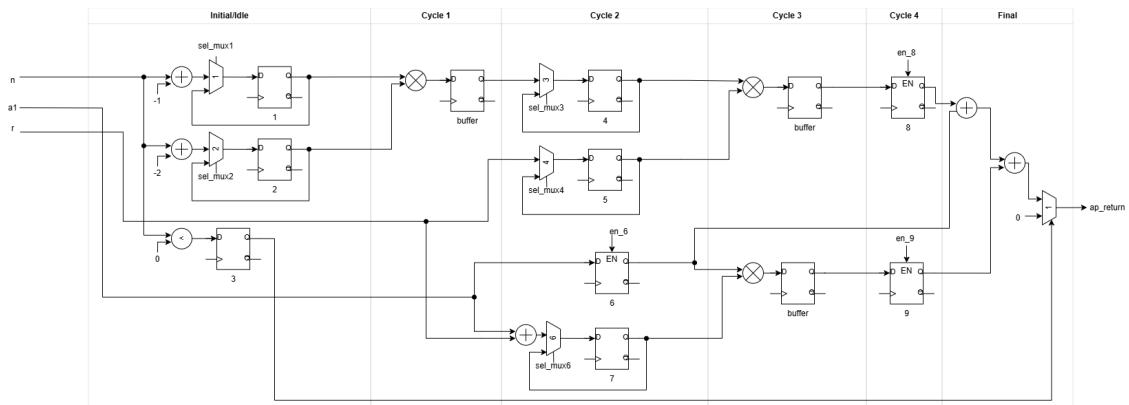


Figure 3. Arithmetic Progression Sum datapath (modeled from Quartus RTL Viewer)

Phase 3 tests whether this six-state FSM computes the sum in a fixed number of cycles. Figure 4 shows the simulation for $a_1 = 3$, $r = 2$ and $n = 5$: the `ap_CS_fsm` signal goes through the six states in order (red box), the intermediate products appear in the cycles predicted by the datapath (colored boxes), and `ap_done` and `ap_ready` are asserted together with `ap_return = 35`, which matches the expected result.

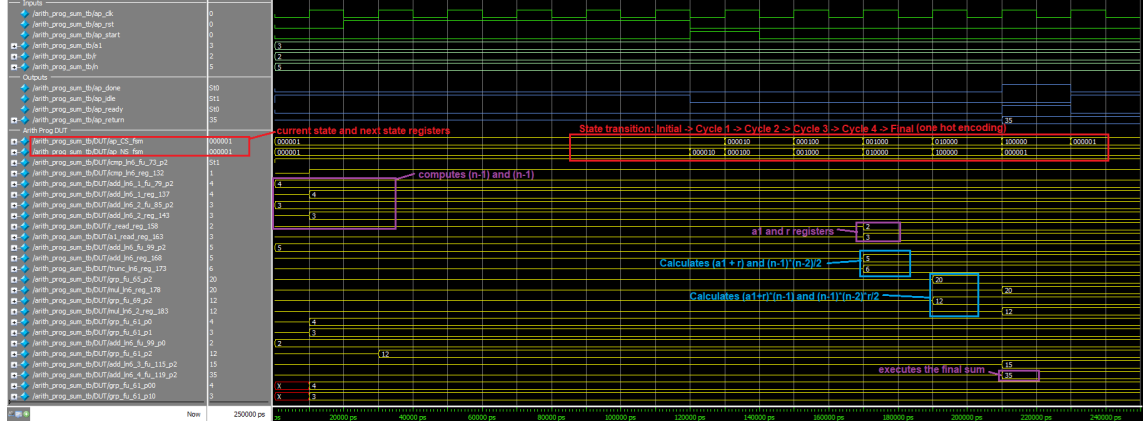


Figure 4. Waveform for the sum of arithmetic progression circuit test with $a_1 = 3$, $r = 2$, and $n = 5$ (simulated and generated via ModelSim)

4.1.2. Factorial

Applying Phase 1 of our methodology, we evaluated a non-recursive C++ implementation for calculating the factorial of a number n . Because this is a purely iterative algorithm without a direct analytical solution, Vitis cannot obtain a closed-form expression from the code analysis. Consequently, our initial interface mapping indicated that the tool would need to create a dedicated control unit to manage the iterations.

Transitioning to Phase 2, we identified the hierarchical organization of the generated hardware. The HLS tool generated four RTL modules: the top-level `factorial` module, which performs the state transition logic and encapsulates the others; a loop pipeline module (`Pipeline_FOR_LOOP_STATEMENT`), which manages the accumulation and iteration variables; a signed 16-bit multiplier; and a standard loop-control module (`flow_control_loop_pipe_sequential_init`), present in all iterative implementations in this work, which manages the loop start and end signals. During the signal isolation step of Phase 2, we observed that Vitis produces a control unit with three states: an initial state, an execution state, and a final state, as shown in Figure 5. The FSM stays in the initial state until `ap_start` is asserted and remains in the execution state until the stopping condition holds.

Upon `ap_start` activation, the initialization state stores variables a and i , and evaluates if $a = 0$. The FSM then triggers the loop pipeline module. During execution, an internal accumulator (initialized to 1) is multiplied by i , which increments per clock cycle. The loop-control module continuously monitors the stopping condition, signaling the main FSM when the loop concludes. In the final state, a multiplexer outputs the result (or 1 if $a = 0$) and asserts the done signal. Figure 6 shows a high-level diagram of how the integration between the described components is performed.

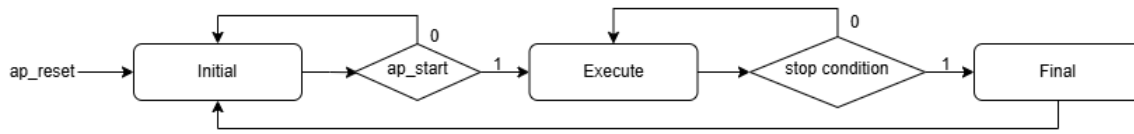


Figure 5. Control Unit for calculating the factorial of a number (modeled from Quartus State Machine Viewer)

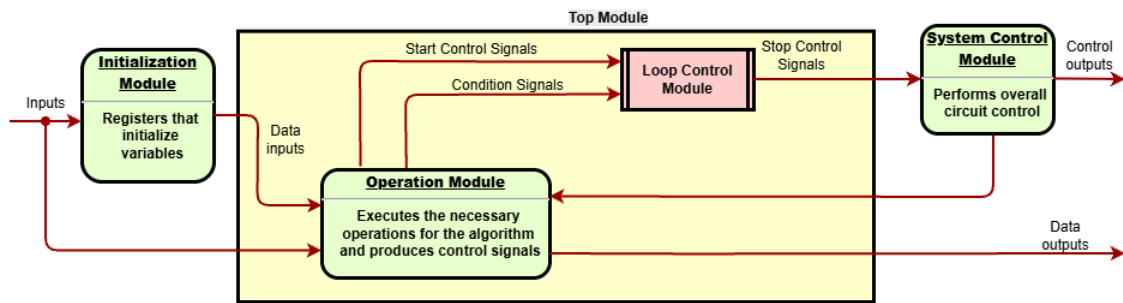


Figure 6. Integration of the modules for the factorial of a number (modeled from Quartus RTL Viewer)

Phase 3 checks this three-state organization, in which the execution state runs the loop pipeline and an accumulator is multiplied by the loop counter at each cycle. Figure 7 shows the simulation for $a = 4$: the state signal goes from initial to execution to final, the loop counter increments at every clock cycle, and `ap_done` and `ap_ready` are asserted with `out_r = 4! = 24`, as expected.

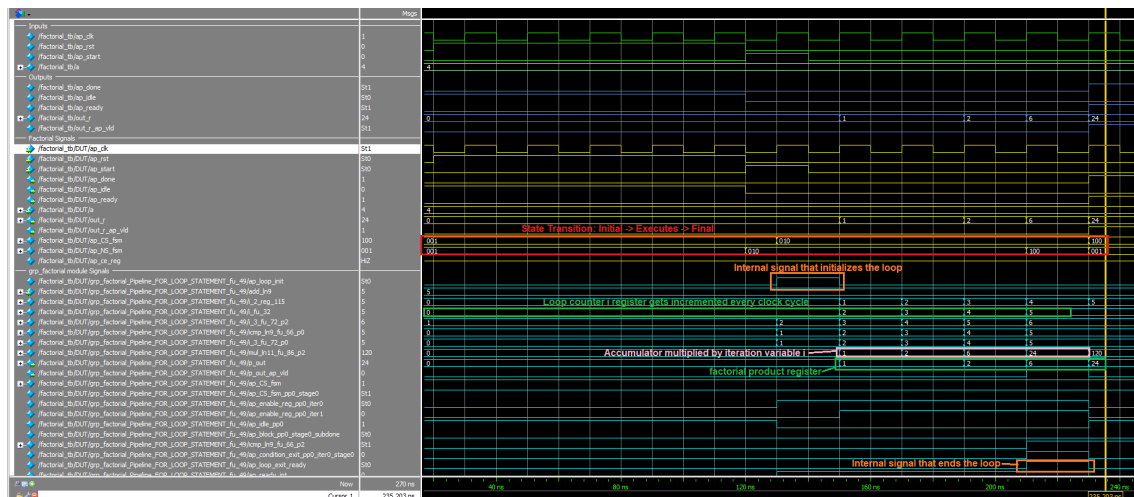


Figure 7. Waveform for the factorial circuit test with $a = 4$ (simulated and generated via ModelSim)

4.1.3. Integer Square Root (SQRT)

Executing Phase 1 on our third case study, we evaluated an algorithm designed to calculate the integer square root of a positive integer n . Similar to the factorial calculation, our initial analysis indicated that this algorithm is purely iterative by nature.

Applying Phase 2, we confirmed that the produced circuit follows a similar pattern to the one identified in Section 4.1.2. The structural analysis identified the top-level `sqrt_hls` module, which controls the states and instantiates the loop pipeline module to execute the iterations and return the `delta_out` control variable. As observed in the factorial circuit, loop synchronization relies on the standard loop-control module.

The FSM remains idle until `ap_start` is asserted, at which point `a` is registered, and variables `square` and `delta` initialize to 1 and 3, respectively. Triggered by the loop-control module, the execution state pipeline increments `delta` by 2 and computes `square + delta` at each clock cycle. Once the stopping condition is met, the internal iteration ends. The final hardware division by 2 is efficiently performed via a one-bit right shift, followed by a subtraction by 1, before the FSM transitions to the final state to assert done. Therefore, the integer square root circuit has a state machine similar to that of Figure 5 and is integrated like Figure 6.

Phase 3 checks whether the circuit indeed repeats the structure of the factorial, with `delta` increasing by 2 at each cycle and no multiplier. Figure 8 shows the simulation for $n = 25$: the states follow the same initial, execution and final sequence, `delta` is incremented by 2 per clock cycle (blue box), the loop-end signal is raised just before the final state, and $res = \sqrt{25} = 5$.

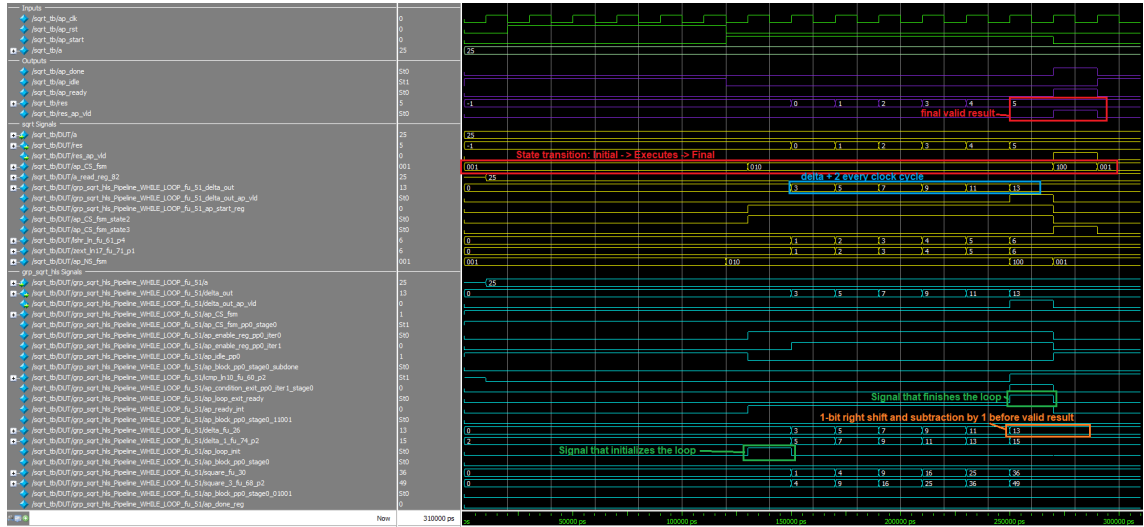


Figure 8. Waveform for the square root circuit test with $n = 25$ (simulated and generated via ModelSim)

4.1.4. Factorial-SQRT

To evaluate a more complex control coupling, we applied Phase 1 to a combined algorithm that calculates the factorial of a positive integer and then determines its square root. In this high-level interface identification step, we anticipated the need for the synthesis tool to integrate two distinct iterative loops within a single top-level hardware entity.

Proceeding to Phase 2, our structural analysis mapped the generation of a top-level `factorial_sqrt` module that integrates two independent loop pipelines, one for the

factorial and one for the square root. The design also uses a 32-bit signed multiplier, and the loop start and end signals are managed by the standard loop-control module.

The top-level control unit coordinates a five-state FSM: initial, factorial execution, transition, square root execution, and final. Upon `ap_start`, n is registered, and the FSM enters the factorial execution state, delegating the iterative processing to its dedicated sub-module. Upon completion, an intermediate transition state prepares the datapath for the integer square root loop. The system remains in the square root execution state until its stopping condition is met, then reaching the final state to provide the valid result and assert done. The circuit's state transition diagram is presented in Figure 9, and the module integration can be seen at a high level in Figure 10. Figure 9 shows that each loop has its own execution state and done flag, and Figure 10 shows the two loop pipelines in series, with the factorial output feeding the square root.

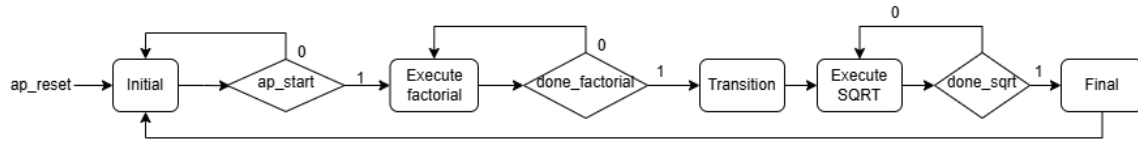


Figure 9. Factorial-SQRT Control Unit (modeled from Quartus State Machine Viewer)

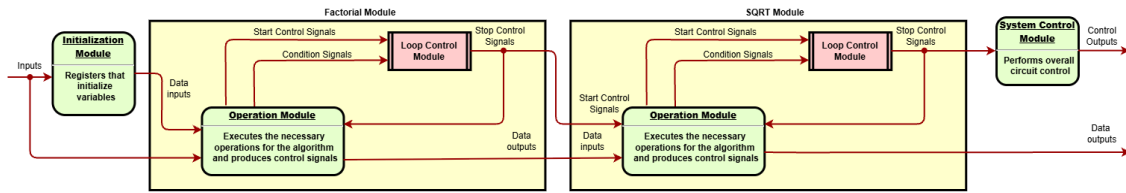


Figure 10. Integration of the Factorial-SQRT modules (modeled from Quartus RTL Viewer)

Phase 3 checks whether the two loops run in sequence under the five-state FSM. Figure 11 shows the simulation for $n = 4$: the states follow initial, factorial execution, transition, square root execution and final, the factorial-done flag is raised before the square root loop starts, and the valid result appears only in the final state.

4.2. Performance Analysis

Table 1 shows the hardware resource consumption and total dissipated power for the selected algorithms. As shown in Table 1, the AP Sum requires more DSPs (9) than the other algorithms. This higher consumption occurs because its $O(1)$ implementation requires multiple simultaneous pipeline multiplications to process its closed-form expression.

Table 2 presents the performance of the synthesized circuits in terms of clock cycles, obtained through co-simulation in Vitis HLS. Corroborating the theoretical analysis, it is noted that the deterministic AP Sum algorithm has $O(1)$ complexity and executes in a strictly constant number of clock cycles (5 cycles). On the other hand, algorithms of an iterative nature (such as Factorial, Square Root, and Factorial-SQRT) present variations according to the processed inputs, requiring a variable number of cycles to complete the control unit operations.

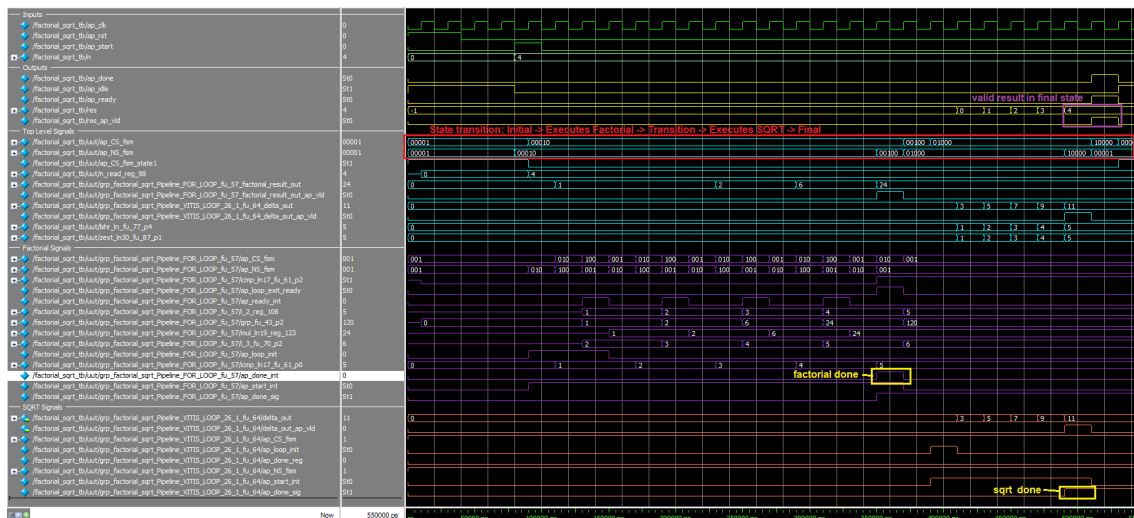


Table 3. Average performance comparison (Time and Cycles) and Speedup between ARM Cortex-A9 (650 MHz) and FPGA (100 MHz)

Benchmark	SW Time (μ s)	SW Cycles	HW Time (μ s)	HW Cycles	Time Speedup	Cycle Speedup
Factorial	25.36	16483	147.27	14726	0.17	1.12
Square Root	28.66	18628	149.11	14911	0.19	1.25
Factorial-SQRT	31.15	20245	169.60	16959	0.18	1.19
AP Sum	31.48	20464	153.07	15307	0.21	1.34

The experimental performance evaluations align with the architectural insights extracted in our three-phase methodology. Empirical metrics, such as the constant cycle count and high DSP utilization of the deterministic AP Sum, versus the variable execution times of iterative algorithms, reflect the datapath and FSM structures mapped in Phase 2 and validated in Phase 3. By relating execution patterns and communication bottlenecks to the original C/C++ semantics, these results support the viability of our reverse-engineering approach. For HPC developers, this highlights the value of the proposed methodology to better understand structural overheads and design application-specific accelerators.

5. Conclusion

This paper proposed and validated a systematic, three-phase methodology to reverse-engineer and comprehend the High-Level Synthesis (HLS) translation process from C/C++ to RTL. By integrating AI-assisted preliminary scanning, structural datapath mapping and signal isolation using Intel Quartus, and behavioral waveform validation via ModelSim, our approach offers a replicable method for understanding HLS-generated hardware.

The application of this methodology to our case studies illustrated its capacity to extract architectural insights. For deterministic algorithms, such as the AP sum, our method systematically mapped how HLS tools optimize the code by inferring a closed-form mathematical formula of $O(1)$ complexity, generating a linear pipeline without the need for complex flow controls. Conversely, by tracking data and control paths, we showed that purely iterative algorithms (such as factorial and integer square root) need the generation of a three-state control unit and a standard submodule to manage repetition loops. Furthermore, our structural mapping isolated a five-stage state machine and distinct submodules when evaluating the coupling of combined iterative algorithms.

Regarding performance, the evaluation revealed that the hardware processes the logic more efficiently, requiring fewer clock cycles than the software. However, the speedup is overshadowed by the data and control transfer time required by the PYNQ-Z2/Jupyter environment, resulting in a higher absolute execution time on the FPGA.

As future work, we propose: (i) evaluating the algorithms on distinct FPGA architectures and platforms; (ii) exploring the impact of HLS optimization directives (pragmas) on the generated RTL; (iii) analyzing complex computing kernels, such as matrix multiplication; and (iv) scaling our methodology to map and evaluate hardware generation in parallel accelerator architectures, targeting multi-instance and full-application scale.

References

- Abderehman, M., Gupta, R., and Karfa, C. (2021). Reverse engineering register to variable mapping in high-level synthesis. In *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 37–42.
- Advanced Micro Devices, Inc. (2026). *Vitis High-Level Synthesis User Guide (UG1399)*. AMD, San Jose, CA. v2025.2.
- Alizadeh, B. and Shiroei, M. (2023). Automatic correction of rtl designs using a lightweight partial high level synthesis. *Integration*, 91:173–181.
- ARM Limited (2016). *SoC Designer AXI4 Protocol Bundle User Guide*.
- Carrion Schafer, B. and Sathe, C. (2025). From high-level synthesis lite to high-level synthesis full: Unlocking hls tool limitations. *ACM Trans. Des. Autom. Electron. Syst.*, 31(2).
- Cong, J., Lau, J., Liu, G., Neuendorffer, S., Pan, P., Vissers, K., and Zhang, Z. (2022). Fpga hls today: Successes, challenges, and opportunities. *ACM Trans. Reconfigurable Technol. Syst.*, 15(4).
- Curzel, S. (2024). *Modern High-Level Synthesis: Improving Productivity with a Multi-level Approach*, pages 15–25. Springer Nature Switzerland, Cham.
- Herklotz, Y., Du, Z., Ramanathan, N., and Wickerson, J. (2021). An empirical study of the reliability of high-level synthesis tools. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 219–223.
- Khan, M. Z. S., Azar, K., and Kamali, H. (2026). Bench4hls: End-to-end evaluation of llms in high-level synthesis code generation. In *2026 Design, Automation Test in Europe Conference (DATE)*, pages 1–7.
- Koronis, J., Garnica, O., Ignacio Hidalgo, J., and Lanchares Dávila, J. (2025). Reverse-engineering optimization techniques of high-level synthesis: Practical insights into accelerating applications with amd-xilinx vitis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 44(7):2558–2570.
- Lahti, S. and Hämäläinen, T. D. (2025). High-level synthesis for fpgas—a hardware engineer’s perspective. *IEEE Access*, 13:28574–28593.
- Molina, R. S., Gil-Costa, V., Crespo, M. L., and Ramponi, G. (2022). High-level synthesis hardware design for fpga-based accelerators: Models, methodologies, and frameworks. *IEEE Access*, 10:90429–90455.
- Vaithianathan, M., Patil, M., Ng, S. F., and Udkar, S. (2023). Comparative study of fpga and gpu for high-performance computing and ai. *ESP International Journal of Advancements in Computational Technology (ESP-IJACT)*, 1(1):37–46.
- Xiong, C., Liu, C., Li, H., and Li, X. (2025). Hlspilot: Llm-based high-level synthesis. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design, ICCAD ’24*, New York, NY, USA. Association for Computing Machinery.
- Xu, S., Jiang, Z., Zhang, Y., Boland, D., Bao, Y., and Shi, K. (2025). Hercules: Efficient verification of high-level synthesis designs with fpga acceleration. In *Proceedings of*

the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays,
FPGA '25, page 56–66, New York, NY, USA. Association for Computing Machinery.