Comparison of HLS vs HDL in FPGA Design Flow

Pisula Mayan Guruge
Department of Electronic Engineering
Winter Semester 2025
Hamm-Lippstadt University of Applied Sciences
deundara-palliya-pisula-mayan.guruge@stud.hshl.de

***Abstract*—Field-programmable gate arrays (FPGAs) are widely used for implementing high-performance and energy-efficient digital systems due to their flexibility and reconfigurability. Traditionally, FPGA designs are developed using hardware description languages (HDLs) such as Verilog or VHDL at the register-transfer level (RTL), which provides precise control over timing, parallelism, and resource utilization. However, RTL-based design flows often require significant development effort and deep hardware expertise, especially as system complexity increases. To address these challenges, High-Level Synthesis (HLS) has emerged as an alternative design methodology that enables designers to describe hardware functionality using high-level programming languages such as C or C++, which are then automatically translated into synthesizable RTL. This paper presents a comparative analysis of HDL-based and HLS-based FPGA design flows. The comparison focuses on key aspects including design productivity, performance, resource utilization, power efficiency, and verification complexity. By examining the underlying design processes and reported case studies, the paper highlights the strengths and limitations of each approach and discusses their suitability for different application domains. The analysis shows that while HDL-based design remains advantageous for performance-critical and tightly constrained systems, HLS can significantly improve development productivity and design space exploration when applied appropriately. The paper concludes that HLS and HDL are best viewed as complementary methodologies, and that hybrid design flows combining both approaches are increasingly relevant in modern FPGA-based system design. [3], [7], [10]**

I. Introduction

Field-programmable gate arrays (FPGAs) have become a key platform for implementing high-performance, energy-efficient, and application-specific digital systems. Unlike ASICs, whose design incurs significant upfront costs, FPGAs allow complex hardware architectures to be implemented and refined through reconfiguration avoiding the large one-time costs that usually accompany ASIC development. Therefore, FPGAs are widely used in domains such as digital signal processing, image and video processing, networking, automotive systems, and machine learning acceleration. Traditionally, FPGA designs have been developed using hardware description languages such as Verilog or VHDL at the register-transfer level (RTL). This approach gives designers control over clocking, data paths, parallelism, and timing, making it well suited for highly optimized and performance-critical implementations. For this reason, RTL-based design continues to be the dominant methodology in such systems. However, achieving this level of control comes at a cost. RTL designs demand substantial hardware expertise and is often accompanied by long development cycles, driven by manual coding, extensive verification, and repeated iterations to meet timing requirements. As FPGA devices continue to grow in complexity, productivity has become a key challenge in RTL-based design flows. Modern designs can easily exceed hundreds of thousands of lines of HDL code, making development, debugging, and long-term maintenance increasingly demanding. At the same time, many FPGA-based applications are inherently algorithm-driven and are often first developed using high-level programming languages. Converting these algorithmic descriptions into cycle-accurate RTL introduces a substantial semantic gap between software-level design concepts and their eventual hardware realization. To address growing productivity concerns, High-Level Synthesis (HLS) changes how FPGA designs are described and implemented. HLS tools allow designers to describe functionality using high-level languages such as C or C++, which are then automatically translated into synthesizable RTL. By automating tasks such as scheduling, pipelining, and resource allocation, HLS aims to reduce development time and makes it easier to evaluate multiple design alternatives. As a result, HLS has gained increasing attention in both academic research and industrial design flows. Despite its advantages, HLS is not a replacement for traditional HDL-based design. While HLS can significantly improve productivity, its impact on performance, area, and power consumption may vary depending on the application and tool configuration. Optimal hardware implementations often still require hardware-aware coding styles and careful use of synthesis directives. As a result, modern FPGA projects often rely on a combination of HLS- and HDL-based approaches instead of choosing one over the other. The coexistence of HDL- and HLS-based methodologies motivates a closer examination of how these design flows compare in practice. This paper examines both approaches in terms of development effort, performance, resource utilization, power efficiency, and verification complexity. The paper focuses on clarifying the key trade-offs between HLS and HDL and on identifying where each approach fits best in practice. The resulting comparison offers practical guidance for choosing suitable design methodologies in modern FPGA-based systems. [3], [7], [9], [12]

II. Overview of FPGA Design Flow

The FPGA design flow defines the set of steps required to convert the functional specification to the hardware setup required to program an FPGA device. This flow provides

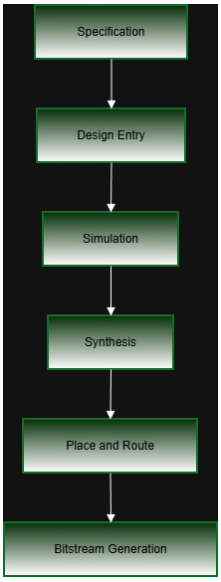

Fig. 1. Overview of the conventional FPGA design flow

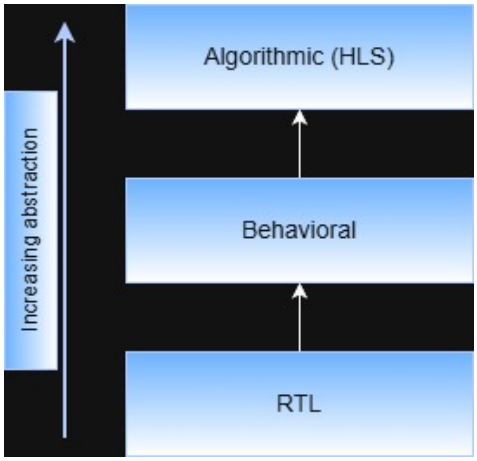

Fig. 2. Abstraction levels in FPGA design from RTL to high-level synthesis

the structural foundation for both traditional HDL-based design methodologies and High-Level Synthesis (HLS)-based approaches. Although the design entry style could be different in each paradigm, the back-end flow is identical.

The design phase begins with system specification, where functional details, performance requirements, interface constraints, and power consumption goals are defined. In this phase, important design choices such as architecture details regarding data widths, clock frequencies, memory requirements, and communication interfaces are made by designers. These early decisions heavily impact later stages of the design flow and determine the feasibility of meeting timing and resource constraints on the target FPGA platform.

As per the specifications, the design entry phase is performed. In the conventional workflow for FPGAs, the design entry phase involves the use of hardware description languages including Verilog or VHDL, carried out on the register-transfer level (RTL). The RTL description systematically defines the operation of clocked processes, combinational logic, and state machines, allowing precise control over hardware behaviour. In contrast, HLS-based workflows enable design entry using high-level programming languages such as C, C++, or SystemC. These descriptions focus on algorithmic behavior rather than explicit clock-cycle timing, leaving scheduling and hardware mapping decisions to the synthesis tool. Regardless of the abstraction level used during design entry, functional verification is a critical next step. Simulation-based verification ensures that the design behaves as intended before hardware implementation. In RTL-based workflows, testbenches are developed manually for verification that stimulate the design and observe signal-level responses. This process can become increasingly complex as system size grows. In HLS-based flows, verification often begins at the algorithmic level using software-style test environments, followed by RTL simulation of the generated hardware to ensure functional equivalence. While HLS may simplify early verification, comprehensive RTL-level verification remains necessary for reliable FPGA deployment.

After the verification for functionality is completed, the design moves on to the synthesis phase. During synthesis, the RTL is further transformed into a netlist that is technology-mapped according to the capabilities of the FPGA chip, including lookup tables (LUTs), flip-flops, block RAMs, and DSP slices. Synthesis software is used for optimizations within the boundaries set by the user for timing, area, and power. In the HLS design methodology flow, synthesis is followed by another stage involving the transformation of the high-level specification for scheduling and subsequent translation to RTL. This stage offers the ability to explore architectural design variations for pipelining stages or degrees of parallelism.

The resulting netlist is passed to the implementation phase, which includes the placement and routing processes. In this process, the logical functionalities are placed in physical locations on the FPGA architecture, and routing is used to provide connections among different logical functionalities for them to function. This stage determines the actual signal delays and has a significant impact on achievable clock frequency. Static timing analysis is performed to verify that all timing constraints are satisfied. Achieving timing closure at this stage is often one of the most challenging aspects of FPGA design, particularly for high-performance or highly parallel systems. After successful placement and routing, the configuration bitstream is created. This bitstream is used to configure the FPGA and establish the run-time behavior of the hardware. The design can now be implemented onto the target hardware for physical validation and performance analysis. The post-implementation verification phase can include chip debugging, logic analyzers, and real-world workload execution to confirm that the design meets its functional and performance objectives. In this overall design flow, the main difference between the HDL-based and HLS-based approaches lies in the design entry and abstraction level. This common back-end

design flow also lets the designer compare both approaches on various other common parameters such as timing performance, resource usage, and power consumption. The following sections discuss the details of both the HDL- and HLS-based design flows that highlight their individual characteristics to enable a structured comparison within the unified FPGA design framework.

III. HDL-Based FPGA Design Flow

The HDL-based FPGA design flow is focused mainly on register-transfer level (RTL) modelling using hardware description languages such as Verilog or VHDL. This approach allows the designer to control the hardware structure, timing behavior, and resource allocation, making it the dominant approach for implementing performance-critical and tightly constrained FPGA systems. In an RTL-based flow, hardware functionality is described in terms of clocked registers, combinational logic, and finite state machines, closely reflecting the underlying hardware architecture.

The design entry process for an HDL flow begins with manual specification of RTL modules. Designers explicitly define datapaths, control logic, and interfaces, often decomposing complex systems into hierarchical modules to improve readability and reuse. The low-level description provides a designer a great deal of control over parallelism and data movement so that a designer can tailor particular hardware structures according to concrete needs in terms of specific performance and latency requirements. However, this level of control also demands significant expertise in digital design principles, including clocking strategies, synchronization, and timing constraints.

Verification plays an important role in HDL-based design flows and typically represents a substantial portion of the overall development effort. The functional verification process is implemented through simulator-based techniques, where testbenches are written to drive the RTL design to check its signal levels. As system complexity increases, verification environments often become sophisticated, incorporating constrained-random testing, assertions, and coverage analysis. Debugging RTL designs requires careful inspection of waveforms and internal signals, which can be time-consuming and error-prone, particularly for deeply pipelined or highly parallel architectures.

After the process of functional verification, the RTL description moves to the synthesis stage. Logic synthesis translates the RTL code into a netlist consisting of the available FPGA resources, such as lookup tables (LUTs), flip-flops, block RAMs, and DSP blocks. During this stage, synthesis tools perform optimizations to improve area efficiency and timing performance while attempting to satisfy user-defined constraints. The quality of results obtained at this stage is strongly influenced by the coding style being used, as small differences in RTL descriptions can lead to significantly different hardware implementations. Later, the process moves into its implementation phase, which includes placement and routing. Placement involves mapping logical elements onto physical locations on the FPGA, and routing involves connecting various elements using the routing resources available. Static timing analysis is also performed at this stage to ensure that all constraints are satisfied. Reaching timing closure is often one of the toughest steps involved in an HDL-based design flow, particularly for high-frequency designs or systems with multiple clock domains. Designers may need to iteratively refine the RTL code, adjust constraints, or modify the architecture to resolve timing violations.

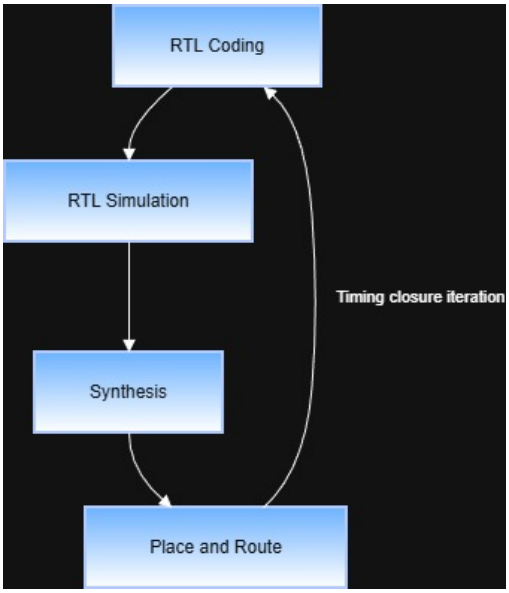

Fig. 3. RTL-based FPGA design flow highlighting iterative timing closure

One major advantage offered by the HDL-based design flow is its high degree of predictability and controllability. Through the explicit definition of hardware behavior at the cycle level, the resulting implementations can be carefully optimized for performance, area, and power consumption. This is one of the reasons why the design flow is particularly useful for applications with strict real-time requirements or limited hardware resources. In addition, well-established design methodologies have made RTL-based workflows highly reliable in industrial settings.

Despite these advantages, HDL-based design flows also exhibit notable limitations. Development time can be substantial, especially for large systems or designs that undergo frequent functional changes. Design space exploration is often manual and limited, as evaluating alternative architectures requires significant code modifications and repeated synthesis and implementation runs. Furthermore, maintaining and modifying large RTL codebases can be challenging, particularly when design intent is not easily captured at the algorithmic level.

In conclusion, the use of the HDL-based FPGA design flow gives designers intricate control throughout the hardware implementation process and remains the preferred choice for performance-critical designs. However, its reliance on low-level abstractions and manual optimization introduces productivity challenges that have motivated the exploration of

higher-level design methodologies. These limitations provide the context for the introduction of High-Level Synthesis, which is examined in the following section.

IV. High-Level Synthesis (HLS)-Driven FPGA Design

High-Level Synthesis (HLS) represents a paradigm shift in the design of FPGAs in that the level of abstraction used to define its functionality is increased. Instead of manually specifying cycle-accurate behavior using hardware description languages, HLS enables designers to express functionality using high-level programming languages such as C, C++, or SystemC. These descriptions are then converted to synthesizable RTL descriptions by performing a set of compiler-driven analyses and optimisation steps. The primary goal of HLS is to achieve increased design productivity without sacrificing hardware performance.

In an HLS-based design flow, design entry starts from an algorithmic description which looks similar to software code. Designers tend to stress the functionality and algorithmic organization rather than the clocking and hardware control. Loops, conditionals, and data structures express computation, while timing and concurrency are deduced by the synthesis tool. This allows designers to reuse existing software models and facilitates early validation using software-style simulation environments.

Scheduling is one of the key steps in High-Level Synthesis (HLS). Based on the specified target clock period and user-defined constraints, the HLS compiler decides which operations can be run in parallel, as well as those that need to be run over multiple cycles. This process can affect latency, throughput, and resource usage. The process can be controlled by directives used by designers. These directives allow partial control over the generated hardware while preserving the benefits of high-level abstraction.

Next is resource allocation and binding. During this stage, the HLS tool assigns operations in the program to hardware elements like adders, multipliers, registers, and memories. The choice between resource sharing and duplication greatly influences area and power consumption. For example, enabling resource sharing can reduce hardware utilization at the expense of increased latency, whereas aggressive unrolling and parallelization can improve throughput while significantly increasing resource usage. HLS tools provide mechanisms to explore these trade-offs efficiently, enabling rapid design space exploration that is difficult to achieve in traditional RTL-based flows.

Once the process of scheduling and resource allocation is completed, the HLS tool produces an RTL description of the design. This can now be passed through the conventional FPGA back-end tools such as those for synthesis, placement, and routing, and bitstream generation. It is now that the HLS-based flow merges with the conventional HDL-based flow as described above. The RTL description is functionally equivalent to the HDL code but can have a vastly different

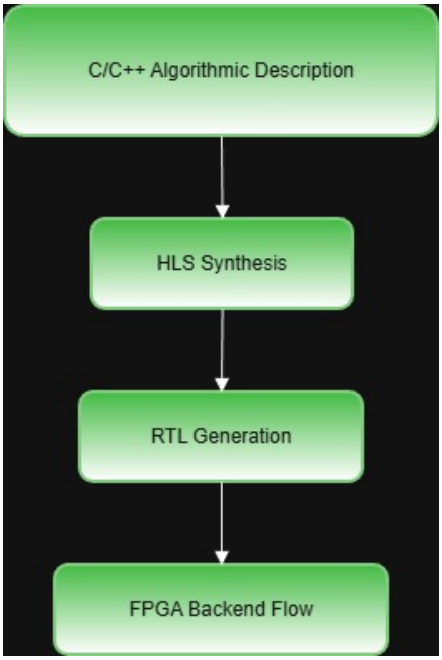

Fig. 4. High-level synthesis based FPGA design flow

implementation, depending upon the HLS tool setup and programming paradigm adopted.

Verification in an HLS-based flow normally happens on multiple abstraction levels. Functional verification is done at an algorithmic level through software testbenches. Early functional verification helps in reducing initial debugging work. Cycle-accurate simulations are required after RTL generation in order to verify that the hardware behavior matches the intended functionality. While HLS can simplify early verification, debugging issues in the generated RTL may be challenging due to the indirect relationship between the high-level source code and the resulting hardware implementation.

One of the main benefits of HLS is the substantial improvement in development productivity. High-level descriptions are generally more concise and easier to modify than RTL code, allowing designers to iterate rapidly and adapt designs to changing requirements. HLS makes it easy to try out different architectures by changing the directives used in synthesis without necessarily changing a large portion of the code. These capabilities make HLS particularly attractive for algorithm-centric applications such as signal processing and machine learning acceleration.

However, there are some limitations associated with HLS. The quality of results obtained from the tools of HLS can depend on many factors, including coding style, compiler heuristics, and tool maturity. Automatically generated RTL may be less efficient than carefully optimized hand-written HDL, especially for designs with strict performance or resource constraints. Moreover, to get proper results for an HLS approach, one requires strong knowledge of underlying hardware concepts because inefficient code written in high-level languages may not always translate into optimal results

Aspect	HLS-Based Design
Abstraction level	Algorithmic
Timing control	Directive-based
Productivity	High
QoR predictability	Medium
Design space exploration	Automated

Table 2. Key characteristics of HLS-based FPGA design.

Fig. 5. Key characteristics of HLS-based FPGA design

when synthesized.

In conclusion, the HLS-based FPGA design flow provides a high level of abstraction and the ability to significantly improve productivity and design exploration. Although the HLS method does not render hardware knowledge unnecessary, HLS provides a complementary approach to traditional HDL-based design and serves particularly well for complex and algorithmically driven systems. The next section builds upon this discussion by directly comparing HLS and HDL design flows across key evaluation metrics.

V. Comparative Analysis: HLS vs. HDL

High-Level Synthesis (HLS) design methodology and traditional HDL-based design are two entirely different methodologies used for FPGA design. The primary differences between the two are their level of abstraction, design effort, and degree of hardware control. Even though both design methodologies end up producing the same synthesizable RTL, their impact on productivity, performance, and design quality can vary significantly. This section provides a comparative analysis of HLS and HDL-based FPGA design flows across several critical parameters relevant to modern hardware design.

A. Design Productivity and Development Effort

One of the widely quoted strengths of HLS is its ability to improve design productivity. Traditional HDL design involves manually implementing the hardware description of the design at the register-transfer level. Therefore, RTL codebases can become large and difficult to maintain, particularly for algorithm-intensive applications. Even minor functional changes may require substantial modifications to RTL code and corresponding updates to verification environments.

In contrast, HLS allows programming with high-level constructs to express functional implementations, thus making code sizes and development time much smaller. Algorithmic changes can be implemented by modifying a short portion of high-level code and then carrying out re-synthesis with new updated directives. This enables rapid iteration and makes HLS particularly suitable for early prototyping and frequent design space exploration.

B. Performance & Timing Predictability

Performance is a critical metric in FPGA design, particularly for real-time and high-throughput applications. HDL-based design offers precise control over timing behavior, as designers define pipeline stages, register placement, and parallelism.

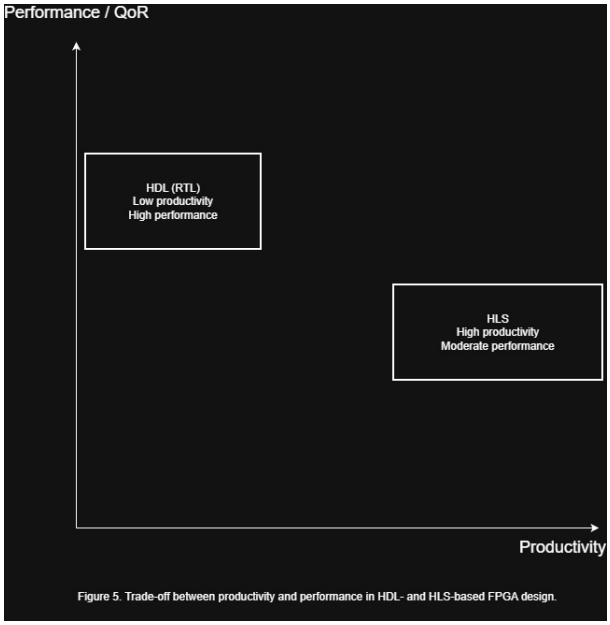

Fig. 6. Trade-off between productivity and performance in HDL- and HLS-based FPGA design

This level of control allows experienced designers to achieve high clock frequencies and predictable timing closure, making HDL the preferred choice for performance-critical systems. HLS-based designs rely on compiler-driven scheduling and optimization techniques to satisfy timing requirements. While modern HLS tools provide mechanisms to guide scheduling through directives, the resulting hardware structure may be less predictable than hand-written RTL. In certain cases, designs synthesized from HLS can have comparable performance to those written in HDL, especially when the kernels are well-structured and compute-intensive. Achieving optimal performance requires careful optimization of directives and understanding how high-level constructs map to hardware.

C. Resource Utilization and Area Efficiency

Resource utilization is another important factor that needs to be considered while comparing the HLS- and HDL-based design flows. In the HDL-based design flow, the designer has direct control over resource usage. Resource sharing, pipeline depth, and parallelism are manually managed, allowing efficient use of FPGA resources such as LUTs, registers, block RAMs, and DSP slices. HLS tools automatically allocate resources based on the high-level description and synthesis directives. While this automation can simplify design, it could result in suboptimal resource usage if the high-level code is not designed with resource implications in mind. Aggressive loop unrolling or pipelining could increase parallelism but could result in increased area usage to some extent. Similarly, excessive resource sharing can reduce area at the cost of performance.

D. Power Consumption

Power efficiency is closely related to both performance and resource utilization. HDL-based design allows power efficiency to be optimized by managing switching activity, clock gating, and resource utilization, and this has been found to be particularly important within embedded designs where strict power budgets must be met. In the case of HLS-based flows, power consumption is indirectly impacted by synthesis and the resulting hardware architecture. While HLS tools can generate power-efficient designs, achieving optimal power characteristics often requires iterative refinement and detailed analysis of the generated RTL. As a result, power optimization may be more challenging in HLS-based designs compared to traditional RTL workflows, especially when low-level power-saving techniques are required.

E. Verification and Debugging Complexity

Verification accounts for a large part of FPGA development time. HDL-based verification typically involves cycle-accurate simulation using detailed testbenches, which can become complex and time-consuming for large designs. Nevertheless, the close correspondence between RTL code and hardware behavior makes debugging relatively straightforward for experienced designers, as signal-level behavior can be directly inspected. The use of HLS-based design methods introduces the concept of a multi-level verification process. Algorithmic verification performed in the initial stage using software-style testbenches can lower the debugging effort; however, discrepancies between high-level code and generated RTL may be difficult to trace. Debugging HLS-generated hardware often requires correlating high-level constructs with low-level RTL signals, which can be challenging without tool support.

F. Design Portability and Maintainability

Design portability and long-term maintainability are increasingly important factors in modern FPGA projects. HDL-based designs are quite portable; however, substantial work may be required in adapting these designs depending on new architectures or performance requirements. Large RTL codebases can also be difficult to maintain and extend over time. HLS-based designs, on the other hand, can provide improved portability at the algorithmic level. High-level descriptions can be easily retargeted for different FPGA platforms by appropriately adjusting the synthesis directives and constraints, without necessarily having to recreate the entire design. This aspect of HLS can be highly attractive for projects that require frequent updates or support multiple target devices.

G. Summary of Comparison

In summary, HLS and HDL-based design flows have complementary characteristics in terms of their strengths and weaknesses. HDL-based design excels in performance predictability, resource efficiency, and precise control, making it well suited for performance-critical and tightly constrained applications. HLS-based design offers significant productivity gains and provides rapid design space exploration, particularly for algorithm-centric workloads. This comparison highlights the trade-offs across key evaluation metrics and serves as a reference point for methodology selection.

VI. Applications & Case Studies

The choice between HDL-based and HLS-based FPGA design flows depends largely on the target application and its performance, flexibility, and development-time requirements. Although the HDL and HLS design flows have the capability of providing the same hardware functionality, the advantages of these design flows lie elsewhere in different application domains. This section discusses representative application areas and reported case-study characteristics to illustrate how HLS and HDL are used in practice.

A. Digital Signal Processing Applications

Digital Signal Processing (DSP) is considered one of the most common applications of FPGA-based acceleration. Typical DSP workloads, such as finite impulse response (FIR) filtering, fast Fourier transforms (FFT), and matrix operations, are highly parallel and computationally intensive. These characteristics make DSP kernels well suited for both HDL-based and HLS-based implementations. It has been observed that in HDL-based designs, DSP applications are often implemented using carefully optimized pipelines and explicit control of data paths. Designers manually manage parallelism, latency, and resource sharing to achieve high throughput and predictable timing behavior. This approach is particularly effective when strict real-time constraints must be met or when hardware resources are limited. HLS-based approaches have found prominence within the DSP arena because of their ability to map algorithmic descriptions quickly and robustly into hardware. By taking advantage of loop pipelining and unrolling, different throughput–area trade-offs are explored with minimal code changes. As a result, HLS is frequently used for rapid prototyping and architectural exploration in DSP systems, while final performance-critical kernels may still be refined using HDL.

B. Image and Video Processing

Applications dealing with image and video processing are generally quite demanding in terms of data throughput and memory bandwidth. Such applications involve operations like convolution, edge detection, and color space conversion, and can be accelerated by the use of FPGAs. These applications often involve streaming data paths and repetitive computations, making them suitable candidates for HLS-based design. HLS enables designers to express image processing algorithms through the use of nested loops and array data structures, closely matching their software representations. Using pipelining and data flow directives, HLS tools generate streaming architectures that process one pixel per clock cycle. This is easier to implement and can easily adapt to changing parameters of the algorithm and image resolution. On the other hand, HDL-based designs for image processing pipelining allow for more control over memory interfaces, buffering

strategies, and timing behavior. This becomes more beneficial while optimizing bandwidth or integrating custom interfaces. Thus, hybrid design approaches are common, where HLS-generated processing kernels are combined with HDL-based control logic and memory subsystems.

C. Machine Learning and Data Analytics Accelerators

Machine learning (ML) acceleration has been identified as a major application area for FPGAs. Neural network inference, in particular, benefits from the parallelism and configurability offered by FPGA architectures. ML workloads are typically algorithm-driven and undergo frequent modifications, making development productivity a critical concern. HLS is widely employed in this field due to its ability to efficiently compile high-level algorithmic descriptions into hardware accelerators. Designers can implement compute-intensive kernels such as matrix multiplication and convolution using high-level constructs, while adjusting performance characteristics through synthesis directives. This enables rapid exploration of different architectural configurations, including varying degrees of parallelism and precision. However, HDL-based designs are still preferred in scenarios where maximum performance or energy efficiency is required. Custom RTL implementations can exploit application-specific optimizations that are difficult to express at a high level. As a result, many ML accelerators employ a combination of HLS for algorithmic blocks and HDL for performance-critical components.

D. Rapid Prototyping and Research

Another area in which HLS is more beneficial is in rapid prototyping. In academic research and early-stage development, design requirements often evolve quickly, and the ability to iterate rapidly is more important than achieving optimal hardware efficiency. The HLS tool allows designers to focus on algorithm development and functional validation, significantly reducing time-to-prototype. HDL-based design, while more time-consuming, remains valuable in later stages of development when design requirements stabilize. This is the stage where performance and resource efficiency are of prime importance; thus, the need to apply an RTL-based refinement. This progression from HLS-based prototyping to HDL-based optimization reflects a common workflow in both academic and industrial settings.

E. Summary of Application Trends

Across application domains, a consistent pattern emerges: HLS is particularly effective for algorithm-centric, compute-intensive workloads and for scenarios requiring rapid development and flexibility. HDL-based design remains the preferred approach for applications with stringent performance, timing, or power constraints. In practice, hybrid design flows that combine HLS and HDL are increasingly common, allowing designers to leverage the strengths of both methodologies.

VII. Challenges & Future Trends

Despite the increasing implementation of HLS in the design methodology of FPGAs, many challenges have been identified that are yet to be addressed to fully utilize HLS to replace the traditional HDL design methodology. Many of these challenges arise from the inherent trade-off between abstraction and hardware control. Understanding these limitations is essential for realistic evaluation of HLS and for identifying directions for future development.

A. Challenges in HLS-Based Design

One of the main issues related to HLS is the predictability of quality of result (QoR). Unlike HDL-based design, where hardware structure and timing behavior are explicitly defined by the designer, HLS involves compiler-driven decisions for scheduling, resource allocation, and binding. As a result, small changes in high-level code or synthesis directives can lead to significant differences in performance, resource utilization, and power consumption. This variability makes it difficult to guarantee optimal results without iterative experimentation and tool-specific expertise. Another challenge is the dependency on coding style and synthesis directives. While HLS aims to simplify hardware design, achieving efficient implementations still requires a strong understanding of how high-level constructs map to hardware. Naïve use of loops, data structures, or memory access patterns can result in inefficient hardware architectures. Consequently, effective HLS usage demands hardware-aware programming practices, reducing the perceived abstraction gap between HLS and RTL-based design. The process of debugging and verification is also a challenge for HLS-based designs. While algorithm-level verification can be done early by using a software-style testbench, debugging RTL code generated from HLS can be challenging, mainly because of indirect relationships between the HLS source code and the synthesised hardware. Since tracing back performance-related issues or functional discrepancies may involve both RTL- and HLS-level analysis, debugging is a complex process. In contrast, HDL-based design faces its own challenges, primarily related to development time and scalability. As system complexity grows, manual RTL development and verification become increasingly time-consuming. Design space exploration is often limited by the effort required to modify and re-verify RTL code, making it difficult to rapidly evaluate alternative architectures.

B. Hybrid Design Flows

To address these challenges, the hybrid HLS–HDL design flow has been realized as an effective solution. Within the hybrid HLS–HDL design flow, HLS can be employed for the development of compute-intensive or algorithmic components, and HDL can be utilized for the development of control logic, interfaces, and performance-critical modules. This approach allows designers to benefit from the productivity advantages of HLS while retaining precise control where necessary. Moreover, hybrid flows are supported more in commercial FPGA

tool flows, making seamless integration of the generated HLS-generated RTL with handwritten HDL modules. This trend proves that HLS and HDL are complementary rather than competing methodologies.

C. Future Trends in FPGA Design Methodologies

Looking forward, there are several trends that are likely to shape the future of FPGA design flow. One of these is the increasing use of domain-specific abstractions, where HLS tools are designed alongside specific domains of applicability, such as digital signal processing and machine learning. Another trend is the integration of artificial intelligence and machine learning approaches into design automation tools. AI-assisted synthesis and optimization have the potential to improve scheduling decisions, predict QoR outcomes, and reduce the manual effort required to tune HLS directives. Such techniques could significantly enhance the usability and reliability of HLS-based design flows. At the same time, the growing shift toward heterogeneous computing is reshaping how systems are designed. As FPGAs are increasingly deployed alongside CPUs and GPUs within complex platforms, design approaches that support system-level integration are becoming more relevant. In this setting, HLS offers a natural link between software-oriented development and hardware implementation.

D. Summary

Although HLS- and HDL-based design flows each come with their own challenges, continued advances in tools and design methodologies are steadily broadening where and how they can be applied. In practice, hybrid workflows and increasingly capable automation techniques point toward a design approach that combines higher-level productivity with the efficiency and control traditionally associated with hardware-centric development.

VIII. Conclusion

This paper has presented a structured comparison of High-Level Synthesis (HLS) and traditional HDL-based FPGA design flows, with emphasis on their design processes, levels of control, and practical trade-offs. By examining both methodologies within a common FPGA design framework, the analysis has shown how differences in design entry and automation influence productivity, performance, and overall implementation quality. HDL-based design continues to be the most dependable choice for applications with strict performance, timing, and resource constraints. The explicit control provided by register-transfer level modeling enables predictable timing behavior, fine-grained resource optimization, and well-established verification practices. These properties make HDL indispensable for performance-critical systems and designs that demand maximum efficiency and determinism. HLS, in contrast, addresses key productivity limitations inherent in RTL-based workflows by allowing functionality to be described using high-level programming languages. This can significantly shorten development cycles and support faster iteration, which is particularly beneficial for algorithm-driven applications and early-stage prototyping. At the same time, achieving high-quality results with HLS often depends on careful coding practices and informed use of synthesis directives, highlighting the continued importance of hardware awareness. The comparison clearly indicates that HLS and HDL should not be viewed as competing approaches. Instead, they play complementary roles within modern FPGA design flows. Hybrid strategies that combine HLS-generated compute kernels with hand-written HDL control logic are increasingly common, offering a practical balance between development efficiency and hardware performance. Ultimately, the choice between HLS and HDL is strongly influenced by application requirements, design constraints, and project priorities. While HDL remains essential for achieving optimal hardware efficiency, HLS provides valuable advantages in development speed and design flexibility. As FPGA architectures and design automation tools continue to evolve, hybrid workflows that integrate high-level design techniques with low-level hardware control are likely to become an increasingly central aspect of FPGA-based system design.

References

[1] D. C. Zissulescu *et al.*, "Raising the Level of Abstraction in FPGA Design," *IEEE Design & Test*, vol. 35, no. 2, pp. 8–17, Apr. 2018.

[2] J. Cong and Z. Zhang, "An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 1, pp. 1–12, Jan. 2006.

[3] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*, IEEE Press, 2008.

[4] R. Tessier and W. Burleson, "Reconfigurable Computing for Digital Signal Processing," *IEEE Signal Processing Magazine*, vol. 18, no. 3, pp. 24–38, May 2001.

[5] J. Bhasker, *A SystemC Primer*, IEEE Press, 2004.

[6] A. K. Verma and P. Ienne, "Improved Use of DSP Blocks in FPGA-Based Soft Processors," *IEEE Transactions on VLSI Systems*, vol. 18, no. 3, pp. 450–463, Mar. 2010.

[7] J. Cong *et al.*, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[8] S. Gupta *et al.*, "SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations," *IEEE Transactions on Computer-Aided Design*, vol. 22, no. 3, pp. 281–295, Mar. 2003.

[9] J. Cong and Y. Zou, "FPGA-Based Hardware Acceleration of Machine Learning Algorithms," *IEEE Micro*, vol. 39, no. 4, pp. 19–28, Jul. 2019.

[10] P. Coussy and A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*, Springer / IEEE, 2008.

[11] S. L. Harris and D. M. Harris, *Digital Design and Computer Architecture: ARM Edition*, Morgan Kaufmann, 2016.

[12] X. Yang, *Integrated Circuit Design: IC Design Flow and Project-Based Learning*, Springer, 2020.

IX. AFFIDAVIT - DEUNDARA PALLIYA PISULA MAYAN GURUGE

I hereby confirm that I have written this paper independently and have not used any sources or aids other than those indicated. All statements taken from other sources in wording or sense are clearly marked. Furthermore, I assure that this paper has not been part of a course or examination in the same or a similar version.

Deundara Palliya Pisula Mayan Guruge
Lippstadt, 14.12.2025