RTL Design and Verification Best Practices in the Semiconductor Industry

Niranjana Gurushankar

Hardware Verification Engineer at Cisco Systems

Abstract

The increasing complexity of integrated circuits (ICs) has made robust RTL (Register Transfer Level) design and verification methodologies crucial for ensuring functional correctness and minimizing time-to-market. This paper explores best practices employed in the semiconductor industry for RTL design and verification, encompassing both established techniques and emerging trends. We delve into coding styles that enhance readability and synthesis, including synchronous design principles, clock domain crossing strategies, and finite state machine implementations. Furthermore, we examine advanced verification techniques such as constrained-random verification, formal property verification, and assertion-based verification, emphasizing their role in achieving comprehensive design validation. The paper also analyzes the impact of emerging design paradigms like low-power design and design-for-test on RTL development. Finally, we discuss the role of automation and machine learning in streamlining the RTL design and verification flow, leading to improved productivity and higher quality designs. This comprehensive analysis provides valuable insights for both novice and experienced engineers seeking to optimize their RTL design and verification processes in the face of evolving industry challenges.

Keywords: Register Transfer Level(RTL), Design-for-Test(DFT), Digital Design, Design Verification, Finite State Machine (FSM), Integrated Circuits (ICs), Low-power Design, Constrained-random Verification

Introduction

The semiconductor industry faces an unrelenting demand for increasingly complex and feature-rich integrated circuits (ICs). This drive towards greater functionality, coupled with shrinking time-to-market windows, presents significant challenges for hardware design and verification. Traditional RTL (Register Transfer Level) design and verification methodologies, while fundamental, are often strained under the pressure of these escalating complexities. Errors introduced at the RTL stage can propagate through the design flow, leading to costly respins, delayed product launches, and potentially even functional failures in deployed systems. This paper addresses the critical need for robust and efficient RTL design and verification practices in this demanding landscape. We explore a comprehensive range of best practices employed within the industry, encompassing both time-tested techniques and emerging trends that are shaping the future of hardware development.This paper addresses the growing complexity of IC design and verification, focusing on efficient design, comprehensive verification, new design paradigms and Increased productivity. In the following sections, we delve into each of these areas, providing valuable insights and practical guidance for engineers striving to optimize their RTL design and verification processes in the face of evolving industry challenges. This comprehensive analysis serves as a valuable resource for both novice and experienced hardware developers seeking to enhance their skills and contribute to the creation of high-quality, reliable ICs.

RTL Design Best Practices

Coding Styles for Clarity and Synthesis

This section emphasizes writing RTL code that is not only easy for humans to understand but also optimized for the tools that will translate it into physical hardware.

Synchronous Design

Digital circuits rely on a clock signal to synchronize operations. In synchronous design, all actions (like calculations, data storage, and signal transitions) happen in step with this clock[1]. Imagine a conductor leading an orchestra – the clock is like the conductor's baton, ensuring every instrument plays at the right time.Synchronous design is crucial for preventing timing issues and a phenomenon called metastability[2]. Metastability occurs when a signal arrives at a flip-flop (a fundamental memory element) too close to the clock edge, causing unpredictable behavior. By synchronizing everything to the clock, we ensure data is captured reliably and the circuit behaves as expected. For e.g. In Verilog or VHDL, you would use constructs like always @(posedge clk) to trigger actions on the rising edge of the clock signal.

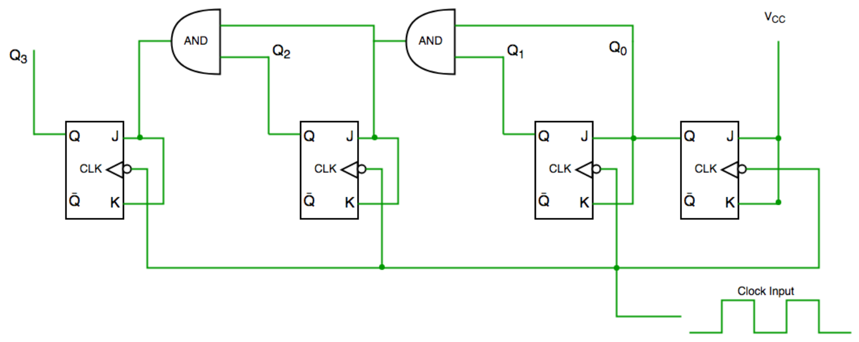

Figure 1: Example of a synchronous digital circuit where all state changes are synchronized to the rising edge of the clock signal [19]

Clock Domain Crossing (CDC)

Modern designs often have multiple clock signals operating at different frequencies or phases. When a signal needs to cross from one clock domain to another, special care must be taken[3]. Directly transferring a signal between asynchronous clock domains can lead to metastability, as the receiving flip-flop might not capture the data reliably. This can cause unpredictable errors in the system. The solution to the issue is using synchronizers first, these are circuits that use two or more flip-flops in series to reduce the probability of metastability when transferring a single signal. Next could be FIFO’s (First-In, First-Out), these are memory structures used to safely pass multiple signals or streams of data between clock domains. They act as buffers, allowing data to be written at one clock rate and read at another.Robust CDC techniques are essential to prevent data corruption and ensure reliable operation in multi-clock systems[4].

Finite State Machines (FSMs)

FSMs are a common way to model sequential logic[5], systems that transition through a series of states based on inputs and current conditions. Think of a traffic light (red -> green -> yellow) or a vending machine. These FSM coding styles are important to be considered, one of them is One-hot encoding, where each state is represented by a single bit being 'high' (1) while all others are 'low' (0). This simplifies logic and can improve timing. Next is Gray coding, only one bit changes between adjacent states. This reduces the risk of glitches during state transitions.Well-defined FSM coding styles make the RTL more readable, easier to debug, and can lead to more efficient hardware implementations[6].

By following these coding practices, RTL designers can create circuits that are robust, reliable, and easier to understand and maintain.These are critical aspects of good RTL design, especially as projects become more complex and teams grow larger. Let's delve into the details:

Design for Maintainability and Reusability

This section is all about writing RTL code that's not just functional, but also easy to understand, modify, and reuse in the future, both by yourself and by others.

Modularity

Instead of writing one massive chunk of code, you break down your design into smaller, self-contained modules[7]. Each module performs a specific function with well-defined inputs and outputs. Think of it like building with Lego bricks – each brick has a specific purpose, and you combine them to create larger structures. Advantage of this method is it is easier to understand, smaller modules are easier to grasp conceptually. You can focus on one piece of the puzzle at a time.If there's an error, you can quickly isolate it to a specific module, making troubleshooting much faster. Well-designed modules can be reused in other projects or parts of the same project, saving time and effort. Modules allow different team members to work on separate parts of the design concurrently.

Parameterization

Instead of hardcoding values directly into your RTL, you use parameters (like variables) to define things like data widths, memory sizes, or delays. You can easily change the design by modifying the parameters without altering the core logic[8]. This is great for exploring different configurations or adapting the design to different requirements. A parameterized module can be reused in different contexts with different parameter values. For e.g instead of writing reg [7:0] data; you could have parameter DATA_WIDTH = 8; reg [DATA_WIDTH-1:0] data;.You can now easily change the data width by modifying the DATA_WIDTH parameter.

Documentation

Clear and comprehensive documentation is essential for understanding and maintaining RTL code[9]. There are different types of documentation, one where you have comments within the code where it explains what each section of code does, any tricky logic, and design choices. Next is SPEC documentation separate documents that describe the overall architecture, interfaces, and functionality of the design. The overall benefit is ofcourse faster debugging, good documentation helps you (or someone else) understand the code quickly, making it easier to find and fix errors. When you need to update or modify the design in the future, documentation will be invaluable in understanding how it works. Documentation helps teams work together effectively by providing a shared understanding of the design.

By focusing on modularity, parameterization, and documentation, RTL designers can create code that is not only functional but also maintainable, reusable, and easier to work with in the long run.

Advanced Verification Techniques

Constrained-Random Verification

The traditional way of testing hardware was to write specific tests for specific scenarios. But as designs get more complex, it becomes impossible to test everything manually. CRV offers a smarter approach. Instead of hand-writing every test, you define a set of rules (constraints) and let the computer generate random test cases within those rules. This allows you to explore a much wider range of scenarios and catch corner case bugs[10] you might never have thought of.

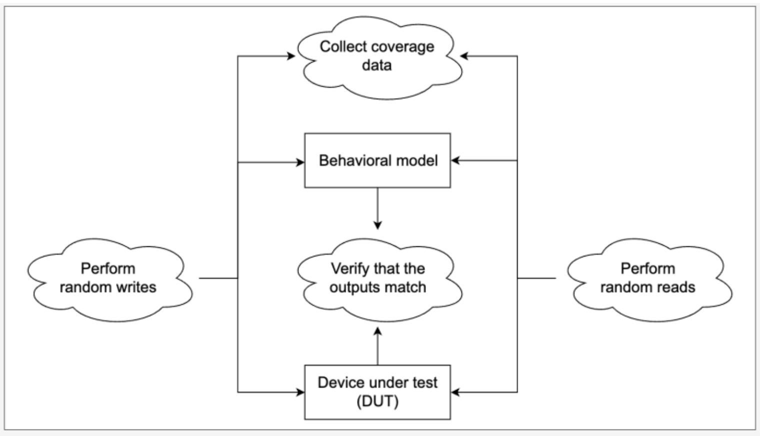

Figure 2: The constrained-random verification process, involving constraint definition, random test generation, simulation, and coverage analysis [20]

Defining Constraints

Constraints are essentially rules that define what kind of input values are allowed and how the design should be exercised. For example, different ways of defining constraints could be defining the input range, where the Input values must be integers between 0 and 255. Input A must always be greater than input B or First write to register X, then read from register Y.

The main purpose of defining constraints is to ensure that the generated tests are relevant to the design's intended functionality and avoid wasting time on meaningless scenarios.

Functional Coverage Groups

With random tests, how do you know if you've tested all the important parts of your design? The solution is Coverage groups! These are like checklists that track which features or functionalities have been exercised by the tests. If you're testing a processor, you might have coverage groups for: Different instruction types (arithmetic, logic, memory access), Different addressing modes and Exception handling. Coverage groups help you measure the effectiveness of your verification and identify any areas that need more attention.

Testbench Automation

CRV involves running a large number of tests, which can be time-consuming and tedious to do manually. Automation! This involves using tools and scripts to generate the random test cases based on the constraints, apply the tests to the design (simulation), collect the results, check for errors and analyze coverage data. Automation makes CRV much more efficient and allows you to run more tests in less time.

CRV helps you reach a wider range of scenarios than direct testing. This helps with corner case bugs since randomization is great at uncovering unexpected bugs that you might miss with manual tests. Improves efficiency since automation saves time and efforts, this gives a lot of confidence to the designers and verification engineers since comprehensive coverage and rigorous testing give you greater confidence in the design's correctness.

By integrating constrained random test generation, functional coverage analysis, and automated test execution within a SystemVerilog/UVM framework, CRV empowers engineers to comprehensively verify complex hardware and achieve high reliability.

Formal Property Verification

FPV uses mathematical analysis to prove that your design meets specific requirements[11], going beyond just testing with simulations.Considering the natural-language design requirements (e.g., "the FIFO should never overflow") and express them in a precise, mathematical language that a computer can understand.A common language for this is SVA i.e. System verilog assertions[12], where you write assertions that formally define the intended behavior.

Let’s take an e.g assert property (@(posedge clk) disable iff (!rst_n) $full(fifo) |-> !$full(fifo)); This SVA assertion checks that if the FIFO is full, it will not be full on the next clock cycle (unless reset is active).

The tool takes your design and the properties, and then systematically explores all possible states that the design can reach. It's like trying every possible combination of a lock to see if any opens it.The set of all possible combinations of values that the signals in your design can have. The tool uses clever algorithms and data structures (like Binary Decision Diagrams - BDDs) to efficiently represent and analyze this state space, even for very complex designs.

Benefits of using FPV is that unlike simulation, which can only test a subset of scenarios, FPV can explore all possible states. FPV can find bugs early in the design cycle, before they become more expensive to fix. Mathematical proof provides a very high level of confidence in the design's correctness. With these benefits, there are a quite few challenges faced using this methodology, for very complex designs, the number of possible states can be enormous, making FPV computationally challenging. Writing good formal properties requires expertise and can be time-consuming. FPV tools often require abstracting away some details of the design to manage complexity. FPV tools like Cadence JasperGold and Synopsys VC Formal, along with some open-source options, provide powerful capabilities for verifying critical design aspects and achieving high confidence in correctness. This technique often complements simulation-based approaches for a more comprehensive verification strategy.

Assertion-Based Verification

Assertions can be thought as "watchdogs" that you place within your RTL code[13]. They constantly monitor the design's behavior during simulation and alert you if something goes wrong. This helps catch bugs early on, when they're easier and cheaper to fix. Assertions are statements that specify the intended behavior of your design. They act like built-in checkers that ensure things are working as expected.

Let’s consider an e.g. assert (a && b) |-> c; This assertion says "If both 'a' and 'b' are true, then 'c' must also be true.” You embed these assertions directly within your RTL code, close to the logic they're monitoring. There are different types of assertions which can be used in your code which is discussed below,

Immediate Assertions

These assertions check for specific conditions at a particular point in time. They're like taking a snapshot of the design and making sure everything looks right at that moment. In SystemVerilog, they often use the assert keyword.

Example: usage of this assertion type assert (data_valid == 1'b1); This checks if the data_valid signal is high at that specific point in the simulation.

Concurrent Assertions

These assertions check for behavior over a period of time, looking for patterns or sequences of events. They're like watching a movie of your design's execution and making sure the plot makes sense.

Example: assert property (@(posedge clk) disable iff (!reset) req |-> ##[1:3] ack); This checks that after a request (req) is asserted, an acknowledgment (ack) must arrive within 1 to 3 clock cycles (unless reset is active).

ABV offers significant advantages, including the ability to catch bugs early in the design process, improve debugging efficiency by pinpointing errors, and serve as executable documentation. Assertions can even be used for formal verification. However, writing effective assertions requires experience, and there might be a minor impact on simulation performance

By embedding assertions directly into your hardware design using languages like SystemVerilog or Property Specification Language (PSL), ABV provides runtime monitoring that catches bugs early. This leads to higher quality and more reliable hardware, and it's supported by all major simulation tools

Emerging Design Paradigms

Low-Power Design

The goal here is to optimize your RTL code to minimize power consumption without sacrificing performance[14]. This involves clever strategies to reduce unnecessary activity in the circuit. There are three main areas to consider while focusing on low-power, Clock gating, Power gating and Voltage scaling which is discussed in detailed below.

Clock Gating

Dynamically turning off the clock signal to parts of the circuit that are not actively being used. Imagine turning off the lights in a room when you leave – you're not wasting energy illuminating an empty space. You insert clock gating cells that control the clock signal to specific modules or registers. These cells are activated by enable signals that indicate when the logic is needed.v Significant reduction in dynamic power consumption, as clock signals are a major source of power drain due to the switching activity they generate.

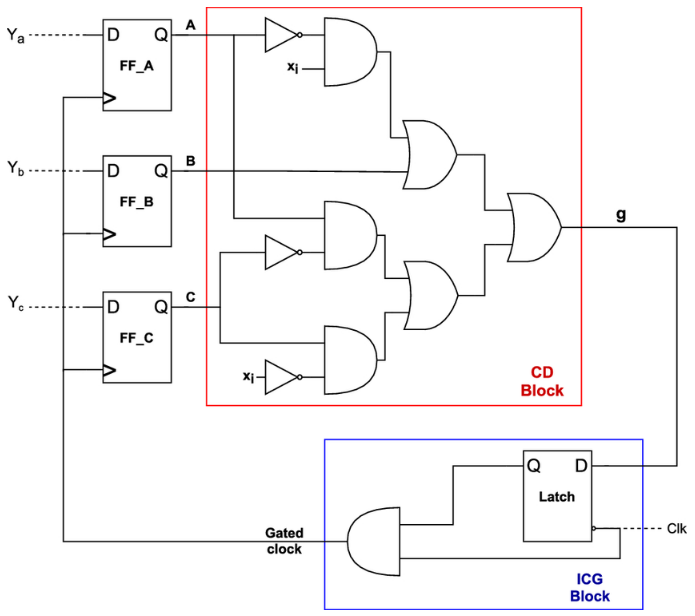

Figure 3: Example of clock gating, where the clock signal to a block of logic is disabled when the block is inactive [21]

Power Gating

Completely shutting off the power supply to inactive blocks of the circuit. This is like turning off the power strip to your entertainment system when you're not using it. Power gating involves using special power switches that can isolate entire blocks of logic. These switches are controlled by signals that indicate when the block is needed. Even more power savings than clock gating, as it eliminates both switching and leakage power in the inactive block. Power gating introduces more complexity in terms of managing power-up and power-down sequences, as well as potential glitches when turning blocks on and off.

Voltage Scaling

Dynamically adjusting the voltage supplied to the circuit based on performance needs. When less performance is required, you lower the voltage, which reduces power consumption. Voltage regulators are used to control the supply voltage. The operating system or power management unit can adjust the voltage based on the workload. Significant power savings, especially in applications with varying workloads. Voltage and frequency are closely linked. Lowering the voltage often means you need to reduce the clock frequency as well, which impacts performance. This trade-off needs to be carefully managed.

Design-for-Test (DFT)

DFT is all about adding features to your design that make it easier to test[15]. This is crucial because as chips become more complex, it gets harder to ensure they're free of manufacturing defects. Let's imagine a complex city with millions of interconnected roads (wires) and buildings (logic gates). How do you make sure every road is connected correctly and every building is functioning? That's essentially the challenge of testing a chip. This section discusses in detail about the various DFT techniques which could be used.

Scan Chains

Flip-flops (the basic memory elements in a chip) are connected together like a chain. This chain has an input and an output, allowing you to directly control the values stored in each flip-flop and observe their outputs. The flip-flops function normally within the circuit. The flip-flops are reconfigured into a long shift register (the scan chain). Test patterns are shifted into the scan chain through the input. The circuit is run for a short period, allowing the test patterns to propagate through the logic. The resulting values are captured in the flip-flops and then shifted out of the scan chain for analysis.

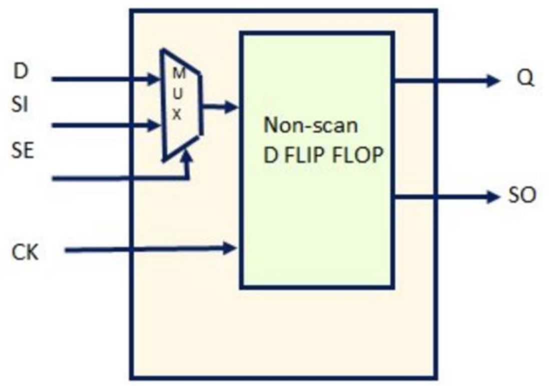

Figure 4: scan D Flip Flop [22]

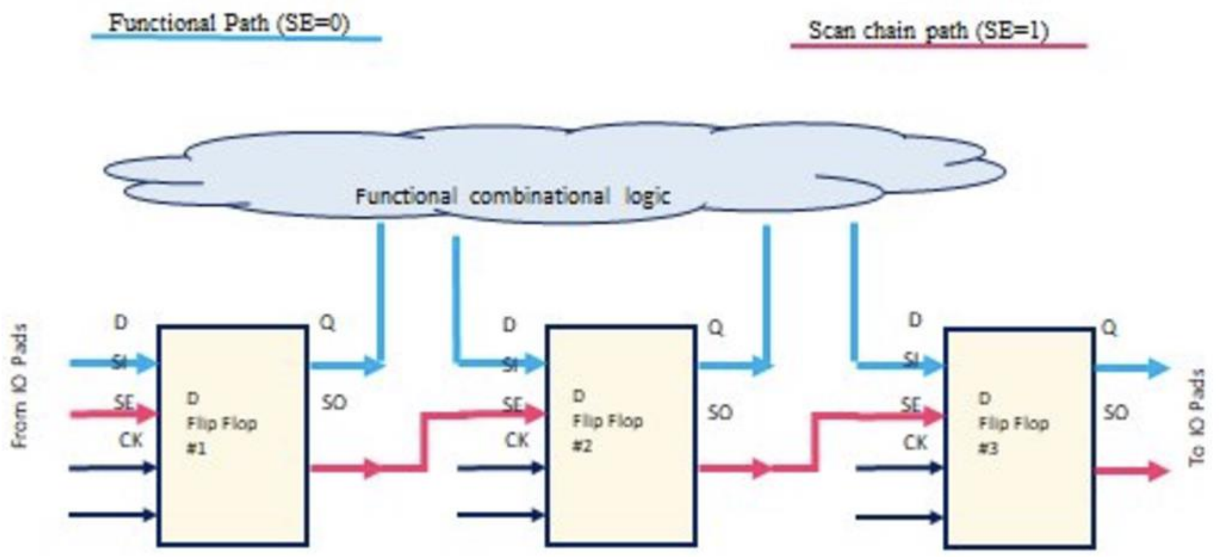

Figure 5: Scan chain used in Design-for-Test (DFT) to improve the controllability and observability of internal circuit nodes [22]

The advantage of using the above method is you can directly set the values of internal signals. Observability of the values of the internal signals is possible, scan chains help achieve very high fault coverage, meaning they can detect a large percentage of potential manufacturing defects.

Built-In Self-Test (BIST)

Instead of relying on external testing equipment, BIST integrates test structures directly onto the chip itself. It's like having a self-diagnostic system built into the city. A pseudo-random pattern generator (PRPG) on the chip creates test patterns. The patterns are applied to the circuit. The circuit's response is analyzed by an on-chip output response analyzer (ORA). The ORA compresses the response into a "signature" that's compared to a known good signature.
By integrating self-testing capabilities, BIST streamlines the testing process, resulting in faster test times, reduced reliance on expensive external equipment, and the ability to test the chip under normal operating conditions.

Automation and the Future of RTL Design

As digital systems become increasingly complex, automation is playing a crucial role in managing that complexity and accelerating the design process. Here are some key areas where automation is making a big impact:

High-Level Synthesis (HLS)

Automating RTL generation from higher-level descriptions[16], improving productivity and design exploration. Instead of writing RTL code manually, you describe the desired functionality in a higher-level language (like C/C++ or SystemC). An HLS tool then automatically generates the RTL code for you. Designers can work at a higher level of abstraction, focusing on algorithms and functionality rather than low-level implementation details. HLS allows you to quickly explore different architectures and optimizations by modifying the high-level code and regenerating the RTL.

Machine Learning (ML) in Verification

ML algorithms are used to analyze vast amounts of verification data[17] (e.g., simulation results, coverage reports) to identify patterns and insights that can help improve the verification process. ML can identify areas of the design that haven't been adequately tested and suggest new test cases to improve coverage. It can learn from past bug patterns to predict potential areas of concern in the current design. ML can help automate and optimize the verification process, leading to faster and more comprehensive testing.

Automated Formal Verification

Tools are being developed that can automatically generate formal properties[18] (assertions) from design specifications or high-level descriptions. Automating property generation saves time and effort compared to manual creation. Automated tools can potentially generate a more comprehensive set of properties.

Conclusion

These automation trends are shaping the future of RTL design, making it faster, more efficient, and more reliable. As tools and techniques continue to evolve, we can expect even greater levels of automation in the future, allowing designers to focus on higher-level innovation and creativity. This paper has explored essential RTL design and verification best practices in the semiconductor industry. By adopting these techniques, engineers can navigate the challenges of increasing complexity, ensure design correctness, and deliver high-quality ICs efficiently. As technology advances, continuous learning and adaptation of new methodologies will be crucial for success in the ever-evolving world of hardware design.

References

[1] Wakerly, J. F. (2000). *Digital Design: Principles and Practices*. Prentice Hall.
[2] Kleeman, L. (1989). The jitter model for metastability and its application to redundant synchronizers. *IEEE Transactions on Computers*, *38*(9), 1296-1302.
[3] Cummings, C. E. (2008). Clock domain crossing (CDC) design & verification techniques using SystemVerilog. *Springer Science & Business Media*.
[4] Spear, C. (2001). *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer Science & Business Media.
[5] Vahid, F. (2007). *Digital Design with RTL Design, VHDL, and Verilog*. John Wiley & Sons.
[6] Grout, I. (2008). *Digital Systems Design with FPGAs and CPLDs*. Elsevier.
[7] Smith, D. J. (2000). *HDL Chip Design: A Practical Guide for Designing, Synthesizing and Simulating ASICs and FPGAs using VHDL or Verilog*. Doone Publications.
[8] Bergeron, J. (2002). *Writing Testbenches: Functional Verification of HDL Models*. Springer Science & Business Media.
[9] IEEE Standard for Verilog Hardware Description Language (IEEE Std 1364-2005). (2005). IEEE.
[10] Yuan, J., Shulka, K., & Pixley, C. (2006). Constraint-based verification. *Springer Science & Business Media*.
[11] Bening, L., & Foster, H. (2010). *Principles of verifiable RTL design*. Springer Science & Business Media.
[12] IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language (IEEE Std 1800-2017). (2017). IEEE.
[13] Foster, H., Krolnik, A., & Lacey, D. (2004). Assertion-based design. *Springer Science & Business Media*.
[14] Chandrakasan, A. P., & Brodersen, R. W. (2005). *Low power digital CMOS design*. Springer Science & Business Media.
[15] Abramovici, M., Breuer, M. A., & Friedman, A. D. (1990). *Digital systems testing and testable design*. Computer Science Press.
[16] Coussy, P., & Morawiec, A. (2008). *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Science & Business Media.
[17] Sharma, M., Gupta, S., & Gupta, R. (2018). A Survey on Applications of Machine Learning in Verification. *Journal of King Saud University - Computer and Information Sciences*, *30*(4), 462-471.
[18] Dijk, T., & Facon, P. (2009). Formal Specification and Verification Methods. *In Model-Based Design of Heterogeneous Embedded Systems* (pp. 127-162). Springer, Dordrecht.
[19] Wakerly, J. F. (2000). *Digital Design: Principles and Practices*. Prentice Hall.
[20] Yuan, J., Shulka, K., & Pixley, C. (2006). *Constraint-based verification*. Springer Science & Business Media.
[21] Chandrakasan, A. P., & Brodersen, R. W. (2005). *Low power digital CMOS design*. Springer Science & Business Media.
[22] Abramovici, M., Breuer, M. A., & Friedman, A. D. (1990). *Digital systems testing and testable design*. Computer Science Press.