Received 26 June 2025; revised 28 September 2025; accepted 7 October 2025; Date of publication 10 October 2025; date of current version 18 November 2025.

*Digital Object Identifier 10.1109/IEEEDATA.2025.3620273*

# Descriptor: *A Corpus of Synthesizable Verilog RTL Modules Dataset for EDA Research (CORE)*

**KIAN KIT CHEAH** [1,2] **(GRADUATE STUDENT MEMBER, IEEE), FU QI CHUA** [1,2] **(GRADUATE STUDENT MEMBER, IEEE), YUNXIANG ZHANG** [1], **ZHUOFAN LIN** [1], **YUXIN JI** [1] **(MEMBER, IEEE), YUHANG ZHANG** [1], **XINFEI GUO** [3] **(SENIOR MEMBER, IEEE), HARIKRISHNAN RAMIAH** [2] **(SENIOR MEMBER, IEEE), AND YONGFU LI** [1] **(SENIOR MEMBER, IEEE)**

[1]School of Integrated Circuits, Shanghai Jiao Tong University, Shanghai 200240, China
[2]Department of Electrical Engineering, University of Malaya, Kuala Lumpur 506033, Malaysia
[3]Global College, Shanghai Jiao Tong University, Shanghai 200240, China

CORRESPONDING AUTHORS: Harikrishnan Ramiah (e-mail: hrkhari@um.edu.my) and Yongfu Li (e-mail: yongfu.li@sjtu.edu.cn).

This work was supported in part by the National Natural Science Foundation of China under Grant 62350610271 and Grant 62201340, and in part by the Explorers Program of Shanghai (Basic Research Funding) under Grant 24TS140020.

**ABSTRACT** This article introduces the Verilog Core Dataset (CORE), a systematically curated corpus of 137 synthesizable Verilog HDL modules spanning 24 distinct categories of digital logic designs. Unlike existing register-transfer level (RTL) benchmark suites that suffer from outdated designs, inconsistent quality, and limited configurability, CORE is purpose-built to address these limitations through a rigorous methodology that combines large language model-assisted code generation, human expert refinement, and comprehensive verification pipelines. Each module is accompanied by a respective testbench for functional simulation, and adheres to Verilog-1995 and Systemverilog-2012 standards, ensuring broad tool compatibility and synthesis readiness. CORE's parameterized and scalable designs enable structural variation, making it uniquely suited for ML4EDA research, design-space exploration, and design-technology co-optimization workflows. By offering a reproducible, extensible, and standards-compliant RTL corpus, CORE establishes a new foundation for benchmarking, education, and innovation in digital system design.

**IEEE SOCIETY/COUNCIL** Circuits and Systems Society (CASS)

**DATA TYPE/LOCATION** Text; Worldwide

**DATA DOI/PID** 10.21227/ddb9-0921

**INDEX TERMS** Corpus of register-transfer level designs (CORE), dataset, digital design, register-transfer level (RTL).

## BACKGROUND

The semiconductor industry's continued advancement relies on design-technology co-optimization (DTCO), a methodology that integrates circuit design with manufacturing process development [1], [2], [3]. Fig. 1 depicts design-technology co-optimization (DTCO) as a closed-loop feedback methodology for the precise optimization of process technology and circuit design [4]. The methodology involves technology computer-aided design (TCAD) simulations [5], [6], process integration [7], [8], [9], standard cell design [10], [11], [12], process design kit (PDK) development [13], [14], [15], circuit implementation, and performance-power-area (PPA) evaluation. The entire DTCO flow is initiated by register-transfer level (RTL) circuit descriptions, which serve as

TABLE I. Overview of Existing RTL Benchmark Suites: Contributions and Limitations

| Benchmark | Contribution | Limitation |
|---|---|---|
| **ISCAS Benchmarks** [16], [17], [18], [19] | Foundational combinational (ISCAS'85) and sequential (ISCAS'89) circuits, widely used in early Automatic Test Pattern Generation (ATPG) and logic synthesis research, provided a common evaluation standard for early EDA algorithms. | Outdated designs; ISCAS'85 contains 11 small combinational circuits (ranging from 6 gates in c17 to ∼3.5k gates in c7552). The ISCAS'89 extends the suite with ∼30 sequential circuits (from s27 with 27 gates up to s38584 with ∼38k gates). None are parameterized or modernized, and documentation is limited. |
| **EPFL Benchmark Suite** [18], [20] | Offers a modern collection of 23 combinational circuits (10 arithmetic, 10 random/control, and 3 very large "MtM" designs), written in Verilog-2001 and widely used in logic synthesis research. | Designs are fixed and non-parameterized; the arithmetic set totals ∼0.37M nodes, random/control ∼76k nodes, and MtM ∼60M nodes. All are purely combinational, limiting applicability to sequential design research. Scale may not represent modern System-on-Chip (SoC), and documentation can be inconsistent. |
| **OpenCores** [21] | A large, open repository of real-world IP repository with over 800 open-source projects spanning CPUs, controllers, and SoCs, offering high functional diversity across different application domains. | The corpus is not a curated benchmark: quality, documentation, and verification maturity vary widely. Only a fraction of projects are synthesizable, and fewer than 20% include testbenches. Inconsistent coding styles and ad-hoc verification pipelines hinder systematic, apples-to-apples comparisons, making reproducible evaluations difficult. |
| **BaseJump STL** [22], [23] | A comprehensive open-source SystemVerilog IP library with hundreds of reusable modules, from basic blocks to complex components like Network-on-Chip and DRAM controllers. The library has been validated in silicon, notably in TSMC Celerity SoC with 511 RISC-V cores and 385M transistors, underscoring its practical relevance. | Despite its breadth, BaseJump STL is not a curated benchmark suite but a reusable IP library verification coverage is uneven, with many modules tagged experimental documentation is incomplete in places, and reliance on module generators requires specific expertise, making systematic benchmarking and reproducible evaluation difficult. |
| **FuseSoC Cores** [24], [25], [26] | A hardware description language (HDL) package manager that provides a versioned library of over 100 reusable open-source IP cores, including CPUs, controllers, and SoCs. It simplifies design reuse and dependency management, promotes modern file standards, and enables seamless integration of cores across projects. | As a collection of pointers to external repositories, it is not a curated, self-contained benchmark suite. Consequently, the cores exhibit significant inconsistencies in coding style, documentation, quality, and adherence to Verilog standards. |

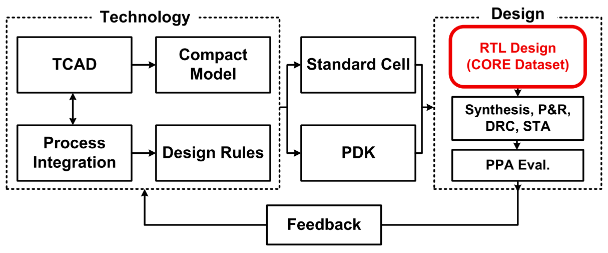


FIG. 1. Design-technology co-optimization (DTCO) flow.

the foundational input for implementation and evaluation (Fig. 1). Consequently, the accuracy of power-performance-area-cost (PPAC) evaluation is fundamentally constrained by the quality, consistency, and diversity of the RTL benchmarks that initiate the flow.

The limitations of established RTL benchmark suites directly reflect these challenges. As summarized in Table I, these suites can be broadly categorized into two groups. The first group includes foundational academic benchmarks, such as the ISCAS benchmarks [16], [17], [18] and the EPFL benchmark suite [18], [20]. While instrumental in early EDA research, many of their designs are now outdated, non-configurable, and often lack comprehensive documentation. The second group comprises large-scale IP core collections that are often repurposed for benchmarking purposes. This includes repositories like OpenCores [21], IP libraries such as BaseJump STL [22], [23], and HDL package managers like FuseSoC Cores [24], [25], [26]. Their primary limitation is that they were not curated as formal benchmark suites, resulting in highly inconsistent quality, coding styles, verification status, and documentation.

This fragmented landscape poses a fundamental barrier: without a systematic, standardized, and parameterizable RTL corpus, researchers lack a reliable basis for reproducible evaluation. The need for such a corpus serves two distinct and complementary purposes. First, for design-technology co-optimization (DTCO) and broader digital design exploration, a curated RTL dataset must prioritize structural diversity and parameterizability, allowing for consistent PPA, process-sensitivity, scalability, and microarchitecture studies across tools, libraries, and process corners. Second, for AI-driven workflows, particularly large language models (LLMs) used for code generation, verification, and

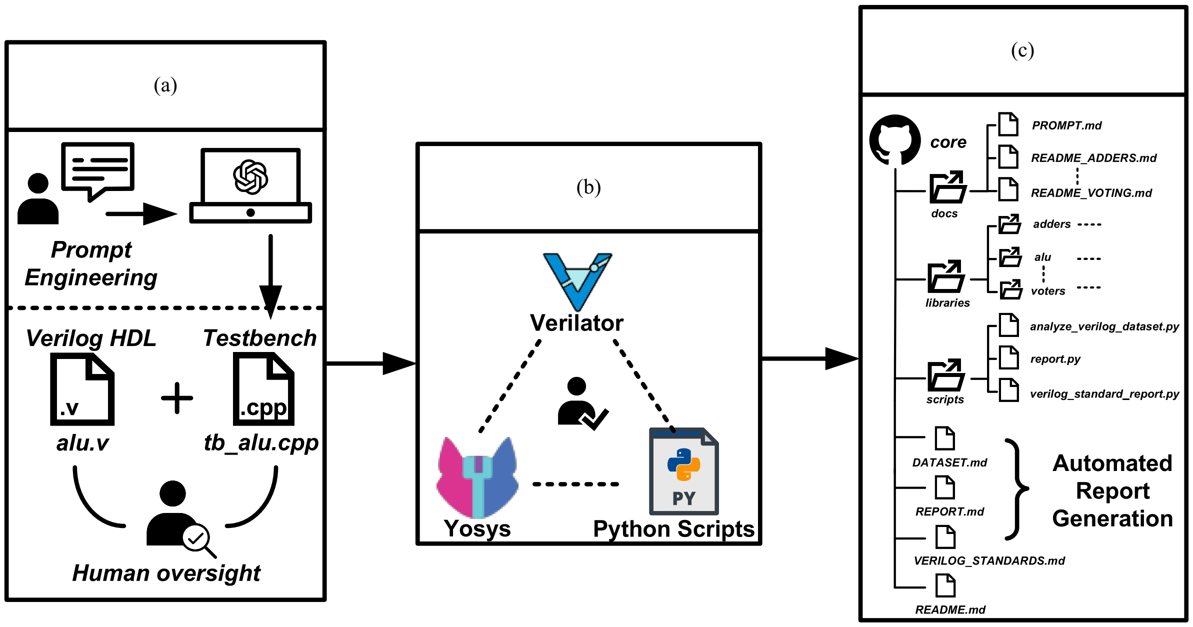


**FIG. 2.** **Overall workflow of data collection and design process. (a) LLM-assisted HDL and testbench co-generation. (b) Functional and performance verification. (c) Data compilation, structuring and documentation.**

design-space exploration [27], [28], [29], the dataset must prioritize consistent coding style, clear documentation, and labeled examples to enable robust training, fine-tuning, and benchmarking. In short, DTCO requires breadth and configurable structural variation while LLM development requires consistency, annotation, and repeatable coding conventions. The absence of a single, curated dataset, therefore, both constrains the reproducibility of DTCO and limits progress in LLM-based hardware design automation. A high-quality RTL corpus that explicitly addresses these two needs is essential for rigorous EDA evaluation and for advancing ML/LLM methods in digital design.

To address these limitations, we present the corpus of RTL designs for EDA research (CORE) dataset, curated to advance EDA for synchronous VLSI systems through

*Comprehensive and evolving library*: The dataset provides a broad and continually expanding foundation for EDA research. It currently features 137 unique Verilog modules across 24 distinct categories, including adders, comms, and memory controllers, each with detailed documentation on its functionality, parameters, and usage. The open-source nature of the collection encourages community contributions and ensures its future growth.

*Parameterized and scalable designs*: Modules are highly parameterized and configurable, enabling diverse structural variations. Synthesized instances range from a few to over 100 000 gates, supporting scalability studies and ML4EDA [30] dataset generation.

*Verified, synthesizable, and standardized modules*: The dataset's quality is ensured through rigorous testing and adherence to established Verilog standards. All modules are functionally verified through their respective testbenches. Synthesis data for all modules is also available, making them reliable, IP-like building blocks for larger systems. To ensure broad tool compatibility, modules are written using synthesizable constructs primarily from the Verilog-1995 (IEEE 1364-1995) [31] and SystemVerilog-2012 (IEEE 1800-2012) [32] standards. This adherence is programmatically verifiable using an included Python analysis script, which reports on the specific language features used in each module.

## COLLECTION METHODS AND DESIGN

The CORE dataset was developed through a systematic methodology. This approach emphasizes the co-generation of register transfer level (RTL) code and its corresponding testbenches using large language models (LLMs), followed by human expert oversight for refinement, and a rigorous verification and characterization pipeline. The resulting Verilog modules are accompanied by a comprehensive suite of documentation, which encompasses detailed functional descriptions for each module (found in category-specific `README_*.md` files, e.g., `docs/README_ADDERS.md`), functional verification summaries (`REPORT.md`), and key characterization metrics such as complexity and logic synthesis results (primarily compiled in the `DATASET.md` file). The overall integrated workflow for module generation, processing, and documentation is depicted in Fig. 2. The data collection and design process can be segmented into several key phases: 1) LLM-assisted HDL and testbench co-generation; 2) functional and performance verification methodology; and 3) data compilation, structuring, and documentation.

### LLM-Assisted HDL and Testbench Co-Generation

The foundation of the dataset lies in an LLM-assisted process for co-generating both the Verilog HDL and its associated testbenches. This phase began with the definition of clear specifications for each module, detailing its desired functionality, input/output (I/O) ports, and parameters for configurability. These detailed requirements were then meticulously translated into effective prompts designed to guide the LLM agents—a crucial step known as "Prompt Engineering." The prompts, as shown below, are documented in `PROMPT.md` files, ensuring transparency in the generation of directives.

*You are an expert in Verilog programming. You are tasked with generating random, functional, parameterized Verilog code and its testbench that is not available in this project, and performing Verilog verification with Verilator and Yosys. If it is incorrect, please identify the root cause from the Verilator and Yosys outputs and correct the RTL and its testbench accordingly.*

Following prompt engineering, LLM agents were employed to produce initial drafts of both the Verilog RTL and testbench code. As Verilog RTL for hardware implementation and robust testbenches demand high precision and adherence to strict design principles, this LLM-generated code served as a foundational starting point. It was then followed by human expert review, modification, and refinement for both the HDL and the testbenches. Human oversight was paramount to ensure the correctness of the generated code, synthesizability, and overall alignment with the intended design goals and established quality standards.

### Functional and Performance Verifications Methodology

The verification and characterization of the dataset was based on three complementary methods: dynamic simulation for functional correctness, static analysis to characterize code quality and complexity, and logic synthesis for hardware implementation metrics.

*Functional verification (Verilator)*: The primary method for verifying the functional correctness of each module was dynamic simulation. This procedure involved executing a dedicated C++ testbench for each design. The Verilator tool [33] was used to compile the Verilog modules into C++ models, which were then simulated against their testbenches to confirm that the module's behavior matched its intended functionality under various test scenarios.

*Static code analysis (Python scripts)*: To complement dynamic verification, a static analysis methodology was employed to characterize the structural properties of the Verilog code without execution. An analysis script `analyze_verilog_dataset.py` was used to parse the source files and extract key metrics. This included measurements of code size (Lines of Code), interface complexity (port count), and reusability (parameter count). The script also quantified the prevalence of common HDL design patterns, such as generate blocks and procedural loops, to profile the coding style across the dataset.

*Logic Synthesis (Yosys)*: To evaluate the hardware implementation characteristics of the modules, logic synthesis was performed. The open-source synthesis tool Yosys [34] was utilized to generate a netlist for each design. Key metrics extracted from the synthesis reports included gate/cell count, wire, and memory element counts for each module, as detailed in `DATASET.md`.

### Data Compilation, Structuring, and Documentation

The data and metrics generated throughout the previous phases were systematically compiled, structured, and documented to ensure usability and reproducibility.

*Categorization and organization:* The 137 Verilog modules are organized into 24 functional categories (e.g., `libraries/adders`, `libraries/voters`). Each category directory contains the corresponding Verilog source (`.v`) and C++ testbench (`tb_*.cpp`) files.

*Automated report generation:* Two summary reports are automatically generated by Python scripts. The `report.py` script executes all testbenches to produce `REPORT.md`, which documents the pass/fail status from dynamic functional verification for each module. The `analyze_verilog_dataset.py` script performs static analysis and synthesis to generate `DATASET.md`. This report contains characterization metrics, including code complexity (LoC, parameter count), design pattern usage, and hardware statistics (gate/cell count) from Yosys.

*Supporting documentation:* Module-level documentation, including functional descriptions, I/O specifications, and parameter usage, is provided in category-specific `README_*.md` files (e.g., `docs/README_ADDERS.md`). To ensure structural consistency, initial drafts of this documentation were generated by LLM agents and subsequently refined by human experts for technical accuracy [35]. The top-level `README.md` file contains project setup, installation, and usage instructions.

## VALIDATION AND QUALITY

The quality and accuracy of the dataset are validated through a three-dimensional assessment: 1) data coverage and diversity; 2) functional verification and synthesis analysis; and 3) code quality and standards compliance.

### Data Coverage and Diversity

The dataset contains 137 Verilog modules organized into 24 functional categories, including "adders," "alu," "arbiters," and so on, as shown in Fig. 3. This structure provides a broad sample of standard digital design components. Module complexity, measured by lines of code (LoC), is distributed across three tiers: 38 simple (0–50 LoC), 81 medium (51–200 LoC), and 18 complex (>200 LoC). This distribution, illustrated in Fig. 4, supports studies targeting various design scales. Further contributing to design variety, the dataset incorporates 115 synchronous and 22 asynchronous designs, enabling research on different clocking and timing strategies. The modules are designed for reusability. On average, each module has 2.77 configurable parameters and 11.09 ports for connectivity. Fig. 5 shows a representative example of

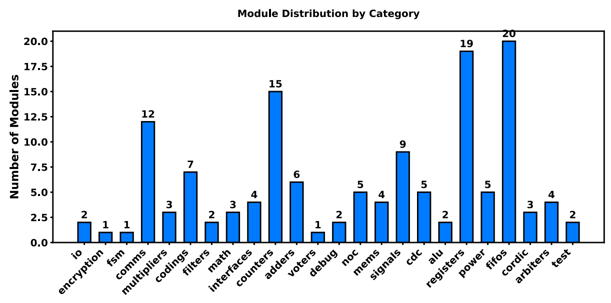


**FIG. 3.** Distribution of Verilog modules by functional category.

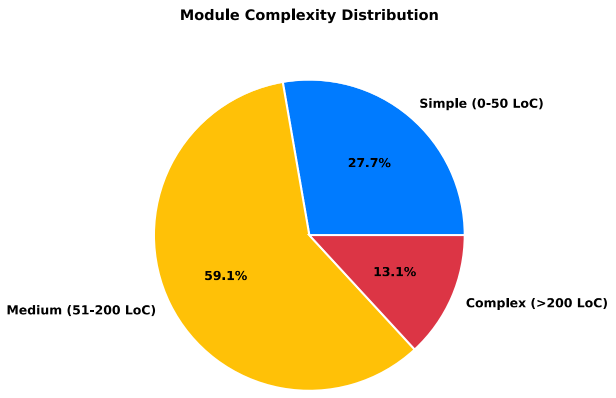
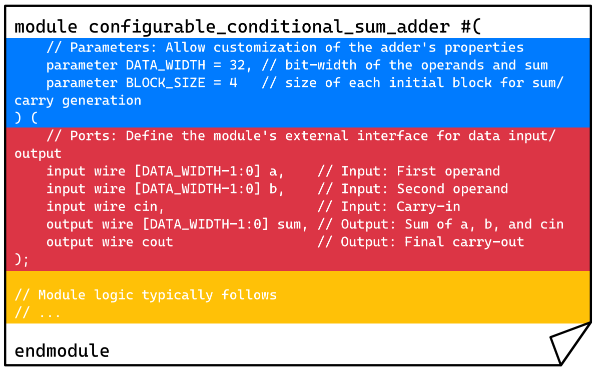


**FIG. 4.** Distribution of Verilog module complexity based on lines of code (LoC).



```
module configurable_conditional_sum_adder #(
    // Parameters: Allow customization of the adder's properties
    parameter DATA_WIDTH = 32, // bit-width of the operands and sum
    parameter BLOCK_SIZE = 4   // size of each initial block for sum/
carry generation
) (
    // Ports: Define the module's external interface for data input/
output
    input wire [DATA_WIDTH-1:0] a,    // Input: First operand
    input wire [DATA_WIDTH-1:0] b,    // Input: Second operand
    input wire cin,                   // Input: Carry-in
    output wire [DATA_WIDTH-1:0] sum, // Output: Sum of a, b, and cin
    output wire cout                  // Output: Final carry-out
);

// Module logic typically follows
// ...

endmodule
```

**FIG. 5.** Visual representation of a typical module interface from the dataset, configurable_conditional_sum_adder.v, showcasing its configurable parameters and port structure.

a module's parameter and port interface, which facilitates flexible integration into larger systems.

## Functional Verification and Synthesis Analysis

The integrity and implementability of the modules are confirmed through functional verification and synthesis analysis.

*Functional verification*: All 137 modules with testbenches were simulated using Verilator. The results, summarized in Table II, confirm a high degree of functional correctness. The module pass rate was 100%, with a 100% pass rate across 1921 individual test cases. This outcome reflects the converged state of the dataset after iterative refinement and validation, ensuring that only fully verified modules are included in the final dataset release. The verification process is efficient, with a total runtime of 74.11 s, averaging 0.54 s per module.

*Synthesis analysis*: To ensure physical implementability, modules were synthesized using Yosys [34]. Synthesis results are available for all 137 modules. For these modules, the average gate count is 1729.52, and the average wire count is 1342.76. The distributions of both gate and wire counts are shown in Figs. 6 and 7, respectively. These

**TABLE II.** Overall Verification Summary

| Metric | Value |
|---|---|
| **Total modules scanned** | 137 |
| **Modules with missing testbenches** | 0 |
| **Total modules tested** | 137 |
| **Modules Passed** | 137 |
| **Modules Failed** | 0 |
| **Module Pass Rate** | 100% |
| **Total tests executed** | 1921 |
| **Total tests passed** | 1921 |
| **Test Pass Rate** | 100% |
| **Total runtime (s)** | 74.11 |
| **Average runtime per module (s)** | 0.54 |

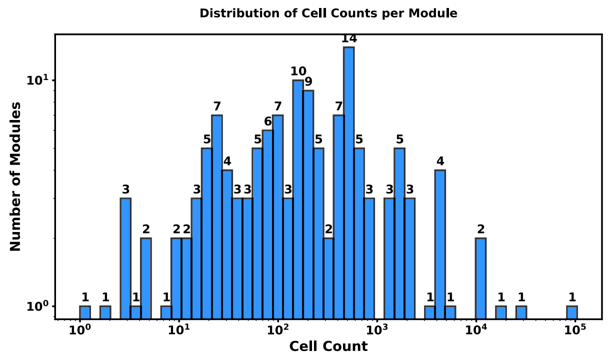


**FIG. 6.** Distribution of post-synthesis gate (cell) counts for the 137 successfully synthesized modules.

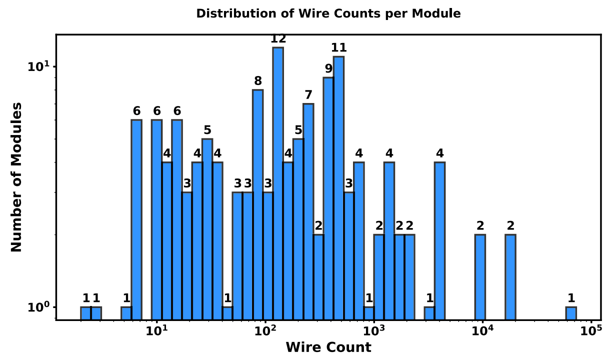


**FIG. 7.** Distribution of post-synthesis wire counts for the 137 successfully synthesized modules.

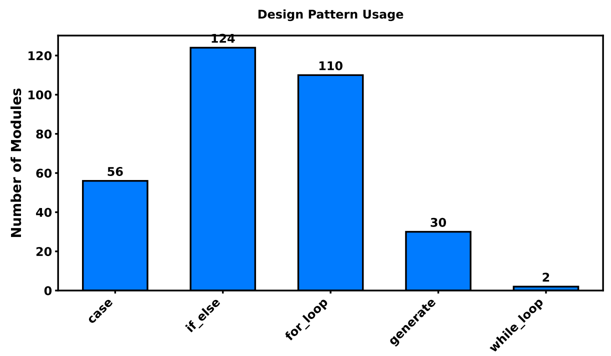


FIG. 8. **Prevalence of common Verilog design patterns observed across the 138 modules in the dataset.**

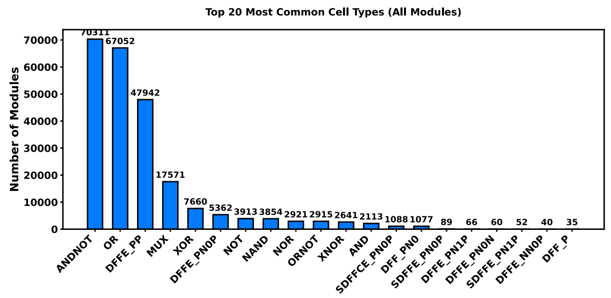


FIG. 9. **Distribution of the most common logic cell types across all synthesized modules.**

metrics serve as observational outputs from the synthesis process, providing transparency into structural characteristics across the dataset. The synthesis experiments were conducted using the default standard cell libraries provided by Yosys. These libraries are technology-agnostic and do not rely on any proprietary PDKs or foundry-specific timing models, ensuring broad accessibility and reproducibility. While the current release does not include synthesis with proprietary or open-source PDKs, incorporating open-source PDKs (e.g., ASAP7, SkyWater 130 nm, FreePDK45, and others) is a promising direction for future extensions of this dataset.

### Code Quality and Standards Compliance

Further quantitative insights into the dataset are provided by analyzing specific code attributes and the performance characteristics of the verification process.

*Code Characteristics*: The average module size is 112.29 LoC. The coding style is characterized by common Verilog constructs: “if_else” blocks are used in 90.5% of modules (124 instances), “for_loop” structures in 80.3% (110 instances), and “case” statements in 40.9% (56 instances), as shown in Fig. 8. Analysis of the synthesized netlists reveals the most common logic cell types, with a detailed breakdown shown in Fig. 9. Across all synthesized modules, the most frequent are “ANDNOT” (70 311), “OR” (67052), and “DFFE_PP” flip-flops (47 942).

*Standard Compliance*: The adherence of the codebase to major hardware description language standards was programmatically verified. The verification method involved a static analysis of each Verilog file, scanning the source code to detect the presence of keywords and syntactical constructs unique to specific standard revisions. For example, compliance with Verilog-2001 (IEEE 1364-200) [36] was identified by detecting features such as `generate` blocks or `signed` data types, while the use of keywords like `logic`, or `interface` signaled adoption of the SystemVerilog-2005 (IEEE 1800-2005) [37] standard. The results of this analysis, summarized in Fig. 10, show that all modules are compliant with the foundational Verilog-1995 standard. More significantly, 60 modules utilize features from SystemVerilog-2005, and 70 modules incorporate constructs from Verilog-2001.

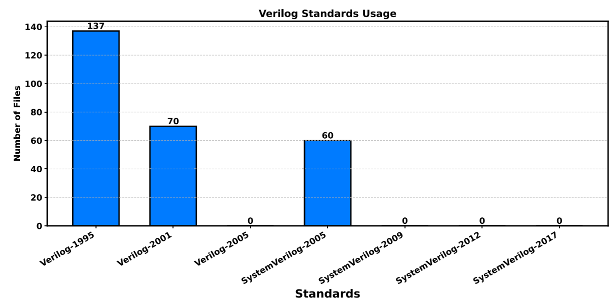


FIG. 10. **Compliance of modules with major Verilog and SystemVerilog standards.**

## RECORDS AND STORAGE

The dataset is contained in a Git repository with a defined file structure, summarized in Table III. The `libraries/` directory forms the core of the dataset. It contains 24 subdirectories, each corresponding to a specific hardware category like `adders` or `fifos`. Within these subdirectories are the Verilog source files (`.v`) and their associated C++ testbenches (`.cpp`). Documentation is located in the `docs/` directory. This directory contains 25 Markdown (`.md`) files: one for each of the 24 module categories and an additional `PROMPT.md` file. The `plots/` directory stores all graphical outputs (`.png`) generated by the analysis scripts. The `scripts/` directory contains three key Python scripts: `analyze_verilog_dataset.py` for performing dataset analysis, `report.py` for generating verification reports, and `verilog_standard_report.py` for checking compliance with Verilog and SystemVerilog standards. A `Makefile` in the root directory automates the execution of these scripts. The root directory also contains essential documentation files, including the main `README.md`, a detailed dataset analysis in `DATASET.md`, a summary of verification results in `REPORT.md`, and Verilog coding standards in `VERILOG_STANDARDS.md`. A `requirements.txt` file is also included to list all necessary Python dependencies. In addition, a continuous integration workflow is provided in `.github/workflows/ci.yml`, which automatically reruns verification and regenerates reports to ensure reproducibility across environments.

**TABLE III. Summary of Primary Directories and Files in the Dataset Repository**

| Directory/File | Description | File Format(s) |
|---|---|---|
| `libraries/` | Core Verilog modules and C++ testbenches | `.v`, `.cpp` |
| `docs/` | Detailed documentation for each module category | `.md` |
| `plots/` | Generated plots from dataset analysis | `.png` |
| `scripts/analyze_verilog_dataset.py` | Python script for dataset analysis | `.py` |
| `scripts/report.py` | Python script for generating verification reports | `.py` |
| `scripts/verilog_standard_report.py` | Python script for Verilog standards compliance | `.py` |
| `Makefile` | Orchestrates building, testing, and reporting | Makefile |
| `DATASET.md` | Detailed analysis of the dataset | Markdown |
| `README.md` | Main repository README with overview and instructions | Markdown |
| `REPORT.md` | Generated verification results report | Markdown |
| `VERILOG_STANDARDS.md` | Verilog coding standards documentation | Markdown |
| `requirements.txt` | Python dependencies | `.txt` |
| `.github/workflows/ci.yml` | Continuous integration workflow for automated verification and report generation | YAML |

## INSIGHTS AND NOTES

This section explores potential applications for the CORE dataset that extend beyond its primary function in general EDA research. This section explores its utility as a library of soft intellectual property (IP) cores for accelerating system-on-chip (SoC) development, its role as a diverse benchmark for evaluating hardware-accelerated RTL simulators, and its value as a practical resource for education in hardware design and computer architecture.

### Soft IP Core Library for SoC Implementation

SoC design methodologies depend on reusable IP cores to reduce development cycles and save time and cost [38], [39]. The CORE dataset serves this methodology by providing its modules as soft IP cores [40], [41] in the form of synthesizable Verilog RTL. Designers integrate these soft IPs with a CPU and other components to create a complete SoC architecture in RTL. As illustrated in Fig. 11, a single, complete SoC design described in RTL can be implemented on multiple, distinct physical targets. It can be synthesized and routed to create a high-performance ASIC for mass production, or it can be synthesized and mapped to a reconfigurable FPGA for rapid prototyping, validation, and lower-volume applications [42], [43], [44].

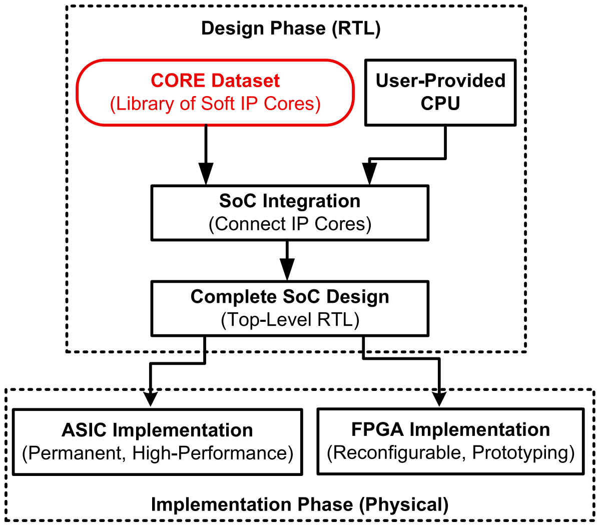

Design Phase (RTL)
CORE Dataset (Library of Soft IP Cores)
User-Provided CPU
SoC Integration (Connect IP Cores)
Complete SoC Design (Top-Level RTL)
ASIC Implementation (Permanent, High-Performance)
FPGA Implementation (Reconfigurable, Prototyping)
Implementation Phase (Physical)

**FIG. 11. SoC implementation flow using the CORE dataset.**

### Benchmarking Hardware-Accelerated RTL Simulators

A primary bottleneck in the VLSI design cycle is the computational expense of RTL simulation, which is often a slow, event-driven, and single-threaded process. Hardware acceleration of this task using platforms like GPUs or FPGAs is a critical area of EDA research [45], [46]. The central challenge for these accelerators is to efficiently handle the full spectrum of digital designs—from parallel, regular structures to irregular, control-dominated logic [47]. A robust benchmark is therefore required to evaluate performance across this entire spectrum. The CORE dataset is well-suited for this benchmarking role due to its structural diversity. As illustrated in Fig. 12, the dataset contains modules with regular, data-parallel architectures (e.g., `configurable_stone_adder.v`, `configurable_fir_filter.v`) that are ideal for execution on SIMD-style hardware like GPUs. It also contains modules dominated by complex, sequential control logic (e.g., `dma_controller.v`, `sequence_detector_fsm.v`) that challenge the limits of parallel execution. By measuring the simulation performance across these distinct categories, researchers can quantitatively assess the strengths and weaknesses of a given hardware accelerator. Furthermore, the extensive parameterization of the modules allows for scalability testing, providing a method to measure how an accelerator's performance changes as the size and complexity of the design increase.

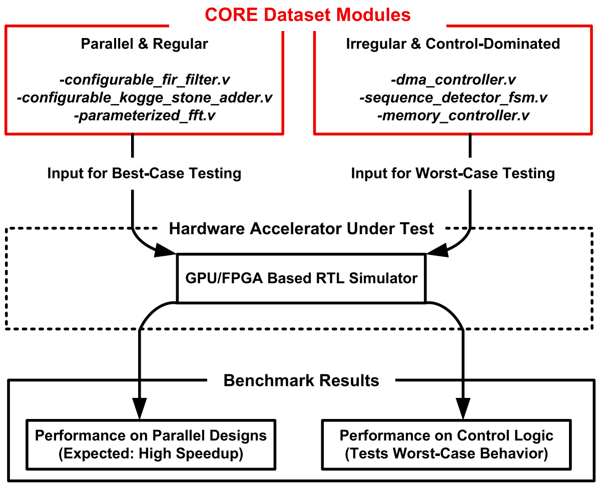

CORE Dataset Modules
Parallel & Regular
-configurable_fir_filter.v
-configurable_kogge_stone_adder.v
-parameterized_fft.v
Irregular & Control-Dominated
-dma_controller.v
-sequence_detector_fsm.v
-memory_controller.v
Input for Best-Case Testing
Input for Worst-Case Testing
Hardware Accelerator Under Test
GPU/FPGA Based RTL Simulator
Benchmark Results
Performance on Parallel Designs (Expected: High Speedup)
Performance on Control Logic (Tests Worst-Case Behavior)

**FIG. 12.** **Benchmarking hardware simulators with CORE.**

### Educational Resource for Practical Hardware Design

As an educational asset, the dataset bridges the gap between theoretical hardware concepts and practical implementation in the field of digital electronics. It enables students to develop practical skills in design, verification, and integration by utilizing Verilog modules and their accompanying testbenches. Moreover, the dataset can serve as a base for case studies in advanced courses such as digital VLSI design [48] and computer architecture [49], discussing how parameterization impacts hardware resources.

## SOURCE CODE AND SCRIPTS

All Verilog source code, testbenches, and analysis scripts are publicly available on GitHub at https://github.com/sjtu-yongfu-research-grp/core/tree/main. The repository's `libraries/` directory houses the Verilog files (`.v`) and their corresponding C++ testbenches (`tb_.cpp`). The root directory contains a `Makefile` and several Python scripts that automate the data processing workflows. The `Makefile` serves as the core automation tool, providing targets to manage verification and synthesis systematically. Key targets include `make verify_<module_name>` to execute a specific module's testbench and `make synth_<module_name>` to synthesize a module with Yosys. The entire suite of testbenches can be executed with `make verify_all`. These `Makefile` targets are orchestrated by the repository's Python scripts. The `report.py` script automates the testing process by invoking the make commands and compiling the results into a comprehensive `REPORT.md` summary. For dataset characterization, `analyze_verilog_dataset.py` performs static analysis and utilizes Yosys to gather synthesis-based metrics, such as gate counts, generating a detailed `DATASET.md` report with statistical plots. Additionally, the `verilog_standard_report.py` script analyzes the source code to determine compliance with various Verilog and SystemVerilog standards, producing the `VERILOG_STANDARDS.md` report. Specific versions of third-party software used in the development and validation process include: Verilog Simulators: Verilator 5.036, Yosys 0.54 + 15. C++ Compiler: GCC 11.4.0, Clang 14.0.0. Python: Python 3.10.12. Git: Git 2.34.1.

## ACKNOWLEDGMENT

K.K.C., F.Q.C., Y.Z., Z.L., Y.J., X.G., H.R., and Y.L. are the main authors of this work. K.K.C., Y.Z., X.G., and Y.L. designed the data collection system, data cleaning, and prepared the data for publication. All the authors reviewed the manuscript.

The authors have declared no conflicts of interest.

## REFERENCES

[1] Y. Zhang and K. L. Low, "Descriptor: MOSFET electrical simulation dataset (MESD)," *IEEE Data Descr.*, vol. 1, pp. 27–32, 2024.
[2] T. Kim, "Challenges on design and technology co-optimization: Design automation perspective," in *Proc. IEEE Int. Midwest Symp. Circuits Syst. (MWSCAS)*, 2023, pp. 212–216.
[3] F. Sheng and R. Tang, "Descriptor: Emerging semiconductor device electrical dataset (ESDED)," *IEEE Data Descr.*, Dec. 2025.
[4] Z. Zhang and R. Wang, "New-generation design-technology co-optimization (DTCO): Machine-learning assisted modeling framework," in *Proc. Silicon Nanoelectronics Workshop (SNW)*, 2019, pp. 1–2.
[5] L. Li and J. Li., "Generalized rapid TFT modeling (GRTM) framework for agile device modeling with thin-film transistors," *IEEE J. Flexible Electron.*, vol. 3, no. 5, pp. 190–196, May 2024.
[6] H. Dixit and V. B. Naik, "TCAD device technology co-optimization workflow for manufacturable MRAM technology," in *Proc. IEEE Int. Electron Devices Meeting (IEDM)*, San Francisco, CA, USA, 2020, pp. 13.5.1–13.5.4.
[7] M. Abedin and S. Khan, "DTCO guided process integration: Case studies from FEOL & BEOL with BSPDN topic: DTCO/DFM," in *Proc. Annu. SEMI Adv. Semicond. Manuf. Conf. (ASMC)*, 2024, pp. 1–5.
[8] C. Wang and Y. Zhang, "D2D-GPT: Leveraging incremental learning GPT for seamless design rule conversion across EDA tools," in *Proc. IEEE Asia Pacific Conf. Circuits Syst. (APCCS)*, 2024, pp. 110–114.
[9] R. Tang and C. Wang, "D2D-LLM+: Unified translation between design rules/manuals and DRC—Bridging inconsistencies for accurate DRC implementation," in *Proc. IEEE Int. Conf. Artif. Intell. Circuits Syst. (AICS)*, 2025, pp. 1–5.
[10] M. Lin and D.-H. Bui, "LogicCraft: LLM-assisted optimization of netlist to layout for complex custom standard cell designs," in *Proc. IEEE Int. Conf. Artif. Intell. Circuits Syst. (AICS)*, 2025, pp. 1–5.
[11] Z. Stanojević and G. Strof, "Cell Designer—A comprehensive TCAD-based framework for DTCO of standard logic cells," in *Proc. Eur. Solid-State Device Res. Conf. (ESSDERC)*, 2018, pp. 202–205.
[12] L. Li and W. Lu, "SCEval: An open-source platform for standardized evaluation and optimization of standard cell libraries in next-generation process nodes," in *Proc. Int. Conf. Electron. Inf. Commun. (ICEIC)*, 2025, pp. 1–4.
[13] C. Ma and Q. Zhang, "IGZO-TFT-PDK: Thin-film flexible electronics design kit, standard cell and design methodology," *IEEE Open J. Circuits Syst.*, vol. 2, pp. 757–765, 2021.
[14] J. Kwak, G. Choe, and S. Yu, "Design-technology co-optimization for stacked nanosheet oxide channel transistors in monolithic 3D integrated circuit design," *IEEE Trans. Nanotechnol.*, vol. 23, pp. 622–628, 2024.
[15] K. Chen and C. Ma, "FreePDK15TFET: An open-source process design kit for 15 nm CMOS and TFET devices," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2021, pp. 1–5.
[16] F. Brglez, P. Pownall, and R. Hum, "Accelerated ATPG and fault grading via testability analysis," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, 1985, pp. 695–698.
[17] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, 1989, pp. 1929–1934.

[18] jpsety, “Verilog Benchmark, EPFL and ISCAS85,” 2020. [Online]. Available: https://github.com/jpsety/verilog_benchmark_circuits

[19] D. K. Houngninou, “benchmark,” (2025). Accessed: Sep. 22, 2022. [Online]. Available: https://github.com/davidkebo/benchmark

[20] G. De Micheli, “The EPFL combinational benchmark suite,” in *Proc. Int. Workshop Logic Synthesis (IWLS),* 2015, pp. 121–134.

[21] OpenCores, “Opencores: Open source hardware IP-cores,” (2024). Accessed: Jun. 5, 2025. [Online]. Available: https://opencores.org

[22] B. S. Group, “Basejump stl,” (2025). Accessed: Jun. 13, 2025. [Online]. Available: https://github.com/bespoke-silicon-group/basejump_stl

[23] M. B. Taylor, “Basejump STL: Systemverilog needs a standard template library for hardware design,” in *Proc. ACM/ESDA/IEEE Des. Automat. Conf. (DAC)*, 2018, pp. 1–6.

[24] O. Kindgren, “FuseSoC,” (2025). Accessed: Jun. 13, 2025. [Online]. Available: https://github.com/olofk/fusesoc

[25] Fusesoc, “Fusesoc Cores,” (2025). Accessed: Jun. 13, 2025. [Online]. Available: https://github.com/fusesoc/fusesoc-cores/tree/master

[26] O. Kindgren, “Invited paper: A scalable approach to IP management with FuseSoC,” (2019). Accessed: Jun. 13, 2025. [Online]. Available: https://osda.gitlab.io/19/kindgren.pdf

[27] S. Thakur and B. Ahmad, “Verigen: A large language model for verilog code generation,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 29, pp. 1–31, 2023.

[28] M. Liu and N. Pinckney, “Invited paper: Verilogeval: Evaluating large language models for verilog code generation,” in *Proc. IEEE/ACM Int. Conf. Comput. Aided Des. (ICCAD),* 2023, pp. 1–8.

[29] J. D. Zamfirescu-Pereira and E. Jun, “Beyond code generation: LLM-supported exploration of the program design space,” in *Proc. CHI Conf. Human Factors Comput. Syst.*, 2025, pp. 143–156.

[30] A. B. Chowdhury and S. Thakur, “Towards the imagenets of ML4EDA,” in *Proc. IEEE/ACM Int. Conf. Computer Aided Des. (ICCAD),* 2023, pp. 1–7.

[31] IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language, *IEEE Standard 1364-1995*, pp. 1–688, 1996.

[32] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language, *IEEE Standard 1800-2012 (Revision IEEE Std 1800-2009)*, pp. 1–1315, 2013.

[33] W. Snyder and V. Committers, “Verilator: High-performance, open-source verilog HDL simulator,” (2025). Accessed: May 21, 2025. [Online]. Available: https://www.veripool.org/verilator/

[34] C. Wolf, “Yosys open synthesis suite,” 2012. [Online]. Available: https://github.com/YosysHQ/yosys

[35] R. Zhong and X. Du, “LLM4EDA: Emerging progress in large language models for electronic design automation,” 2023. [Online]. Available: https://arxiv.org/abs/2401.12224

[36] IEEE Standard Verilog Hardware Description Language, *IEEE Standard 1364-2001*, 2001.

[37] IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language, *IEEE Std 1800-2005*, 2005.

[38] Y. Zorian, E. J. Marinissen, and S. Dey, “Testing embedded-core-based system chips,” *Computer*, vol. 32, no. 6, pp. 52–60, Jun. 1999.

[39] Z. Guiqing and F. Tao, “The SoC design and implementation of digital protective relay based on IP cores,” in *Proc. Int. Conf. Power System Technol.*, vol. 4, 2002, pp. 2580–2583.

[40] M. El-Assal and M. Bayoumi, IIII “MAP decoder architecture: Soft IP for SoC applications,” in *Proc. Midwest Symp. Circuits Syst.*, 2002, pp. 156–178.

[41] X. Pang and D. Yu, “Design and application of IP core in SoC technology,” in *Proc. Int. Symp. Inf. Sci. Eng.*, 2010, pp. 71–74.

[42] F. Abid and N. Izeboudjen, “Technology-independent approach for FPGA and ASIC implementations,” in *Proc. Int. Conf. Elect. Eng. (ICEE),* 2015, pp. 1–4.

[43] F. Abid and N. Izeboudjen, “ASIC implementation of an OpenRISC-based SoC for VoIP application,” in *Proc. Int. Conf. Inf. Commun. Syst. (ICICS),* 2015, pp. 64–67.

[44] M. Hammerquist and R. Lysecky, “Design space exploration for application specific FPGAS in system-on-a-chip designs,” in *Proc. IEEE Int. SOC Conf.*, 2008, pp. 279–282.

[45] Z. Guo and Y. Zhang, “GEM: GPU-accelerated emulator-inspired RTL simulation,” in *Proc. Des. Automat. Conf.* Piscataway, NJ, USA: IEEE, 2025.

[46] R. S. Molina and V. Gil-Costa, “High-level synthesis hardware design for FPGA-based accelerators: Models, methodologies, and frameworks,” *IEEE Access*, vol. 10, pp. 90429–90455, 2022.

[47] B. Reagen and R. Adolf, “MachSuite: Benchmarks for accelerator design and customized architectures,” in *Proc. IEEE Int. Symp. Workload Characterization (IISWC),* 2014, pp. 110–119.

[48] J. Williams, *Digital VLSI Design with Verilog: A Textbook from Silicon Valley Technical Institute.* New York: Springer, 2014.

[49] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. San Mateo, CA, USA: Morgan Kaufmann Publishers, 2019.