VerilogASTBench: Benchmark Construction of Verilog AST Dataset with Dual-Stage AST Semantic Enhancement Framework

LUPING ZHANG, Nanjing University of Posts and Telecommunications, China
CHAO CHEN, Nanjing University of Posts and Telecommunications, China
DAPENG YAN*, Nanjing University of Posts and Telecommunications, China
HUI XU*, University of Electronic Science and Technology of China, China
MINGSHENG CAO, University of Electronic Science and Technology of China, China
JINGKUAN SONG, Tongji University, China
ZHIKUANG CAI, Nanjing University of Posts and Telecommunications, China
YUFENG GUO, Nanjing University of Posts and Telecommunications, China

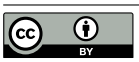
With the increasing complexity and scale of integrated circuit (IC) designs, automated circuit design methods are essential for Verilog implementation. Although Large Language Models (LLMs) perform well in general-purpose coding such as C++ and Python, their performance in Verilog is constrained by domain-specific semantic and structural rules as well as the scarcity of high-quality training datasets. To address these issues, this work proposes a semantically enhanced Verilog code parsing and automatic repair framework based on Abstract Syntax Tree (AST). First, an advanced register-transfer level (RTL) analysis framework was developed to achieve semantic enhancement through static analysis–driven functional role inference and attribute annotation. Second, enhanced AST information is used to construct structured prompts and generate semantically rich module descriptions via an Application Programming Interface (API) for dataset construction. Finally, an AST-guided automatic Verilog repair framework was designed, which leverages enhanced AST analysis for precise defect localization and intelligent repair through compiler feedback loops. Experimental results indicate that the proposed method successfully repaired 15.24% of defective Verilog code, resulting in a high-quality RTL benchmark dataset containing 318,021 samples. Models fine-tuned on this dataset demonstrate significant performance improvements across three benchmarks, achieving average improvements of 13.76% and 16.95% on Eval-Machine pass@1 and pass@5, 8.97% and 14.56% on Eval-Human pass@1 and pass@5, and 15.70% and 12.20% on RTLLM V1.1 Syntax-VCS and Func.

CCS Concepts: • **Hardware → Hardware description languages and compilation**; • **Software and its engineering → Software testing and debugging**; • **Computing methodologies → Language resources**; **Natural language processing**.

Additional Key Words and Phrases: Verilog, Benchmark, Abstract Syntax Tree, Code Repair, LLM

*Dapeng Yan and Hui Xu are the corresponding authors.

Authors' Contact Information: Luping Zhang, Nanjing University of Posts and Telecommunications, Nanjing, China, 2024221209@njupt.edu.cn; Chao Chen, Nanjing University of Posts and Telecommunications, Nanjing, China, 1224228403@njupt.edu.cn; Dapeng Yan, Nanjing University of Posts and Telecommunications, Nanjing, China, dapeng.yan@njupt.edu.cn; Hui Xu, Institute for Advanced Study, University of Electronic Science and Technology of China, Shenzhen, China, huixu.kim@uestc.edu.cn; Mingsheng Cao, University of Electronic Science and Technology of China, Chengdu, China, cms@uestc.edu.cn; Jingkuan Song, Tongji University, Shanghai, China, jingkuan.song@gmail.com; Zhikuang Cai, Nanjing University of Posts and Telecommunications, Nanjing, China, whczk@njupt.edu.cn; Yufeng Guo, Nanjing University of Posts and Telecommunications, Nanjing, China, yfguo@njupt.edu.cn.



ACM 2994-970X/2026/7-ARTFSE045
https://doi.org/10.1145/3797073

ACM Reference Format:
Luping Zhang, Chao Chen, Dapeng Yan, Hui Xu, Mingsheng Cao, Jingkuan Song, Zhikuang Cai, and Yufeng Guo. 2026. VerilogASTBench: Benchmark Construction of Verilog AST Dataset with Dual-Stage AST Semantic Enhancement Framework. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE045 (July 2026), 22 pages. https://doi.org/10.1145/3797073

1 Introduction

In modern information technology, integrated circuits (ICs) constitute the fundamental infrastructure, with their design quality directly determining industrial development and technological progress. With the rapid increase in the complexity and scale of IC designs [16], traditional manual Verilog design methods face significant limitations, driving urgent demand for automated hardware design methodologies. As the mainstream hardware description language, Verilog code serves as the critical bridge between design specifications and hardware implementation [11], playing a core role in the Electronic Design Automation (EDA) [20] toolchain and directly affecting the efficiency of synthesis, simulation, and verification processes [9]. In recent years, Large Language Models (LLMs) have been widely applied to code generation tasks for general-purpose programming languages such as C++ [39], Python [6], and Java [57], achieving substantial progress and contributing to the advancement of software development automation [59]. These developments have led researchers to investigate their application to Verilog code generation for hardware design automation. In contrast to software programs, Verilog is a register-transfer level (RTL) hardware description language (HDL) that describes concurrent hardware behavior with explicit clock-driven state updates and cycle-accurate timing semantics. However, existing LLMs still suffer from low syntax and functionality correctness in Verilog code generation. These challenges primarily stem from the lack of large-scale, high-quality Verilog training data [4], which prevents models from capturing the domain-specific patterns and structural relationships. Consequently, Verilog code generation is highly susceptible to hallucinations, including symbol misinterpretation, knowledge deficiencies, and logical reasoning errors [47].

To address these challenges, researchers have explored large-scale data construction and post-training strategies. For data construction, VeriGen [44] integrates approximately 51,826 pairs of high-quality Verilog modules collected from GitHub with grammar rules and code examples extracted from 70 textbooks, which improves model understanding of Verilog semantics. GEMMV [56] integrates GPT-4 with prompt engineering to automatically construct a high-quality dataset covering 24 types of GEMM hardware designs and multiple hardware optimization techniques. Furthermore, the AutoVCoder framework [12] adopts a two-stage fine-tuning approach, combining 210,000 pairs of filtered Verilog modules with synthetic question-answer pairs generated by ChatGPT-3.5 to enhance model performance on Verilog-related benchmarks. On the post-training side, other studies enhance model performance by introducing stronger supervision and reasoning signals. For example, symbolic interpretation-driven Chain-of-Thought reasoning has been applied to better align model outputs with Verilog design specifications [53]. Similarly, multi-granularity dataset construction combined with a balanced fine-tuning strategy has been shown to enhance model generalization [58]. The integration of Chain-of-Thought annotations with expert-curated labels and curriculum learning further improves the semantic understanding and code generation quality of LLMs [26].

However, the existing benchmarks still exhibit several issues. Their training data is primarily composed of static examples, which limits the ability to model module hierarchies and interconnections. Many approaches also rely on closed-source models, leading to high construction costs and limited reproducibility. Heavy dependence on synthetic data can also result in insufficient domain-specific semantic richness. Moreover, existing methods primarily focus on shallow semantic analysis and

code pattern learning while overlooking the inherent structural hierarchy and functional semantics of Verilog designs [50]. Verilog code is inherently highly structured and functionally semantic [13]. Its design follows a modular hierarchical organization, consisting of modules, interfaces, and signal dependencies. In particular, Verilog functional behaviors often depend on complex contextual information, including cross-module interactions, data-flow dependencies, and state transitions. These structural hierarchies and functional semantics make it challenging for purely text-based processing methods to fully capture the design-level relationships embodied in Verilog code [60].

To address these issues, this paper adopts an Abstract Syntax Tree (AST) based analysis method to better represent the structural and semantic characteristics of Verilog designs. The AST [10] effectively preserves the structural integrity of Verilog code while clearly representing module organization and hierarchical relationships. Nevertheless, existing open-source Verilog parsing tools lack sufficient capabilities for deep structural analysis and fail to extract fine-grained semantic relationships and design patterns. Pyverilog [42], implemented in Python, supports AST construction and data-flow analysis for Verilog HDL, while Surelog [7] targets SystemVerilog and parses the code into the Universal Hardware Data Model (UHDM) for EDA tools. Although these tools are powerful in structural modeling, their outputs are primarily grammar-driven representations (e.g., AST or UHDM) and lack high-level semantic annotations such as port functional roles, signal timing attributes, and cross-module dependencies. This constraint severely restricts their ability to abstract design-level semantics, including hierarchical module instantiation and reset logic patterns, thereby hindering the construction of semantically rich datasets required for effective LLM-based HDL code generation.

Therefore, this work develops a semantically enhanced static analysis benchmark dataset based on AST, which integrates structural parsing and high-level semantic annotation to construct high-quality RTL benchmark datasets, thereby enabling a more comprehensive evaluation and optimization of LLMs for hardware design automation. This work collected 1,035,777 Verilog samples from open-source repositories. After functional verification, 266,585 high-quality samples were retained.

However, this process discarded many samples with recoverable defects. First, open-source Verilog repositories contain numerous syntax errors and incomplete files, leading to substantial sample discard during data cleaning. Second, the lack of efficient code quality assessment and automatic repair mechanisms prevents effective utilization of these defective samples from authentic designs, reducing model learning efficiency and limiting coverage of real-world RTL designs. While code translation techniques [18] can expand the available training corpus, recovering and leveraging authentic open-source designs remains essential for constructing high-reliability benchmarks. Therefore, we propose an agent-driven repair mechanism based on AST analysis to further enhance dataset quality and reliability [49]. By representing code as abstract syntax trees, our approach analyzes code hierarchy and dependencies. Specifically, this method represent error code samples as AST structures, analyze the type and location of each error based on the structured information, and then repair the error code through interactions with EDA tools. The repaired samples are subsequently incorporated into the dataset as supplementary data, thereby enhancing dataset integrity and expanding the coverage of design patterns.

A total of 337,590 error codes were collected after removing duplicates, of which 51,436 were successfully repaired, yielding a repair rate of 15.24%. The repaired codes were eventually merged with the validation data. As a result, a high-quality RTL benchmark dataset containing 318,021 samples was constructed. This dataset provides a solid data foundation for research on LLM-based hardware design automation. The primary contributions of this work encompass:

- Constructing a large-scale, semantically enhanced Verilog benchmark with 318,021 validated RTL samples, offering broader coverage, richer structural annotations, and higher reliability than existing datasets, and establishing a solid foundation for LLM-based hardware design automation.
- Enhancing Verilog code semantics through a dual-stage AST framework that infers functional roles, captures timing attributes, and models cross-module dependencies, while repairing defective samples with a compiler-feedback pipeline that restores 15.24% of errors and improves dataset completeness without sacrificing real-world diversity.
- Fine-tuning small-scale open-source LLMs on the enriched dataset and benchmarking them against state-of-the-art baselines, achieving performance that approaches, and in some cases surpasses, closed-source counterparts, with 9–17% absolute gains on VerilogEval/RTLLM and over 10% improvement on specification-to-RTL conversion tasks.

2 Background

With the rapid increase in design complexity and the continuous advancement of Moore's Law, traditional HDL design processes that rely on manual coding are increasingly unable to meet modern IC design requirements for efficiency, quality, and scalability. RTL represents a fundamental abstraction in digital circuit design [32] that describes hardware behavior through data flow between registers and combinational logic. Verilog is a primary language for RTL modeling and describes concurrent hardware structures rather than sequential software execution. The increasing design complexity and Verilog's inherent characteristics (concurrent execution, timing constraints, and hierarchical organization) present fundamental challenges for automated code generation. Recent advances in LLM, including GPT [34], CodeGen [33] and CodeLlama [37], have improved the accuracy and efficiency of automated hardware design. These models enable mapping natural-language specifications to Verilog implementations.

To systematically evaluate the capabilities of these models, VerilogEval [23] constructed an evaluation dataset based on HDLBits, comprising 156 Verilog programming problems. VerilogEval v2 [35] further expands the dataset by introducing RTL code completion and specification-to-RTL generation tasks, providing more comprehensive evaluation dimensions. RTLLM [27] proposes a benchmark dataset covering 30 diverse digital circuit designs, categorized into two major types: arithmetic and logic. These benchmarks establish evaluation standards while exposing persistent limitations in LLM-based Verilog code generation.

Verilog code generation using LLMs still encounters several critical challenges. Current models typically treat HDL code as plain text sequences, lacking explicit modeling of its inherent hierarchical structure and semantic dependencies. This often results in structural inconsistencies, such as mismatches between module declarations and instantiations, inconsistent signal bit widths, incorrect port connections, and improper multi-clock-domain partitioning. Even code that passes syntax checks can still suffer from logical errors, such as confusion between combinational and sequential logic or inaccurate state-machine transitions. These issues are exacerbated in complex, multi-level modular designs, where inter-module dependencies, hierarchical signal transmissions, and strict interface specifications must be preserved to ensure functional correctness. Existing LLMs struggle to capture these relationships effectively, frequently leading to interface mismatches, signal transmission errors, and module instantiation failures. The fundamental cause of these problems lies in the lack of explicit structural modeling of Verilog code's inherent characteristics, as traditional text-based processing methods cannot effectively capture its hierarchical organization and complex semantic dependencies.

Meanwhile, although some HDL code-repair methods incorporate compiler feedback, they primarily rely on shallow error information and therefore struggle to address deeper structural issues. For high-complexity designs involving state machines, pipeline architectures, or bus protocols, the

functional accuracy of generated code often decreases noticeably. Furthermore, redundant logic, inconsistent naming conventions, and irregular code structures still exist, which can reduce code maintainability and increase verification costs.

In addition, existing works on Verilog dataset construction have alleviated data scarcity. Nonetheless, significant challenges remain in achieving fine-grained semantic enhancement, broader coverage of complex RTL designs, and developing comprehensive, industrial-scale benchmarks. Notably, existing dataset construction approaches lack structured semantic representation, making it difficult to provide models with adequate structural priors and hierarchical understanding necessary for high-quality HDL code generation. These limitations motivate the need for constructing a high-quality, semantically enriched, and structurally comprehensive dataset to better support LLM-based hardware design automation.

To address these challenges, this paper proposes a semantic-driven, AST-enhanced dataset construction framework. By integrating structural extraction, semantic constraints, and agent-assisted repair, this approach constructs a high-quality dataset with guaranteed structural integrity and functional correctness.

3 Study Design

This section outlines the research questions, algorithm benchmark, and implementation methods. These elements are employed to investigate the effectiveness of the Verilog parsing method based on a semantically enhanced AST in constructing the benchmark dataset.

3.1 Research Questions

This paper investigation addresses the following research questions (RQs):

- **RQ-1.** ***How can the structural and semantic properties of Verilog code be systematically characterized and modeled to enable high-quality learning for large language models?***
  This problem investigates the intrinsic characteristics of the dataset derived from multi-dimensional analysis, focusing on scale distribution, complexity stratification, and error pattern analysis. The structural distribution of samples across different complexity tiers was comprehensively evaluated and rigorously examined.
- **RQ-2.** ***How does the incorporation of multi-level semantic constraints—such as Natural language description, AST structures and role annotations—affect large language models' ability to generate functionally correct and synthesizable Verilog code?***
  This study comprehensively investigates the effect of prompt structural complexity and semantic richness on the reliability of Verilog code generation using large language models. The evaluation framework rigorously examines multiple dimensions, including code executability, functional correctness, and structural alignment.
- **RQ-3.** ***To what extent does a high-quality, semantically enhanced Verilog dataset improve model accuracy and cross-model and cross-task generalization in hardware design automation?***
  To address this problem, a dataset-driven fine-tuning experiment was performed. The effectiveness of the proposed dataset in improving model accuracy and enhancing cross-model and cross-task generalization was systematically evaluated through comparisons between the fine-tuned model and the baseline model.

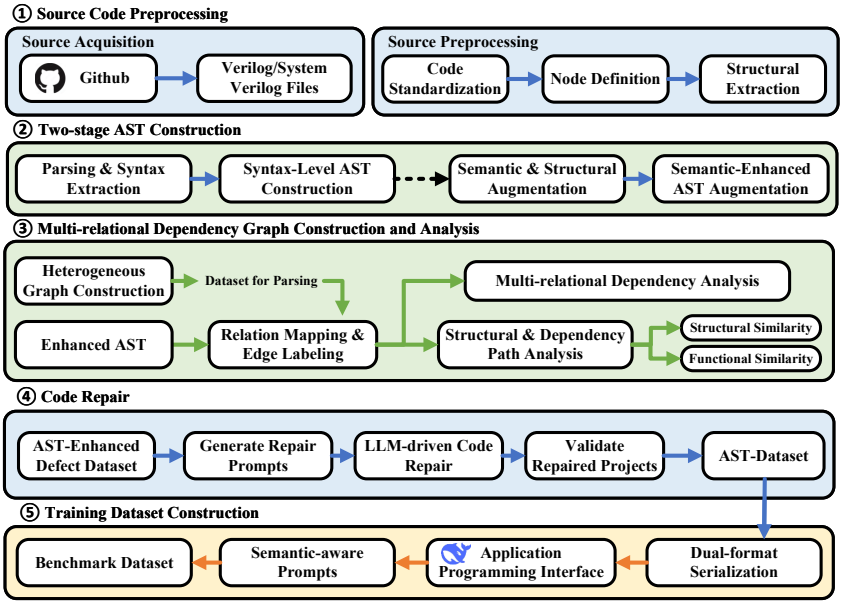


Fig. 1. Enhanced AST-based Large-scale Verilog Dataset Construction Method

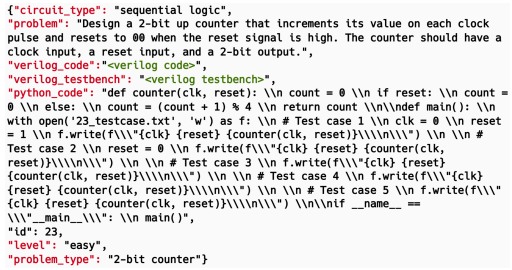
3.2 Algorithm Description

To construct a high-quality RTL benchmark dataset, this paper proposes a comprehensive framework consisting of two core components: dataset construction and defect repair. The overall algorithmic framework is illustrated in Figure 1.

3.2.1 Dataset Construction. To build a high-quality Verilog benchmark dataset, this paper collects original design files from multiple open-source code repositories and systematically acquires Verilog projects on January 1st of each year from 2010 to 2025 from GitHub.

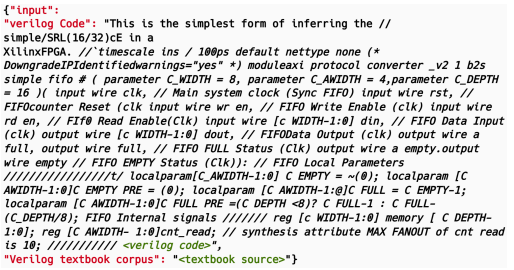
To eliminate data redundancy, this work collected 1,035,777 Verilog source files and removed duplicates using a content-hash-based deduplication method, resulting in 604,175 valid samples. Subsequently, Icarus Verilog [48] was employed for compilation and simulation testing. After dual verification of syntax and semantics, 266,585 high-quality samples were retained, while the remaining 337,590 samples failed verification and were regarded as defective candidates. These failed samples were further processed by the proposed AST-guided repair framework, which successfully repaired 51,436 samples, yielding a sample-level repair rate of 15.24%. The repaired samples were then merged with the verified samples, resulting in a final benchmark dataset containing 318,021 validated RTL samples. Figure 2 compares representative samples from our dataset with those from existing benchmarks. The results demonstrate that this dataset significantly surpasses existing works in task coverage, richness of semantic annotations (e.g., AST hierarchies, role labels, and timing constraints), and overall data quality.

This work employs a content-hash-based deduplication approach to remove duplicate files across directories and data sources, thereby guaranteeing dataset uniqueness. Figure 2 contrasts representative samples from existing Verilog benchmarks with our dataset, which provides substantially broader task coverage and temporal span, more comprehensive semantic annotations (including AST hierarchies, role labels, and timing constraints), and superior data quality ensured through rigorous deduplication and functional validation.



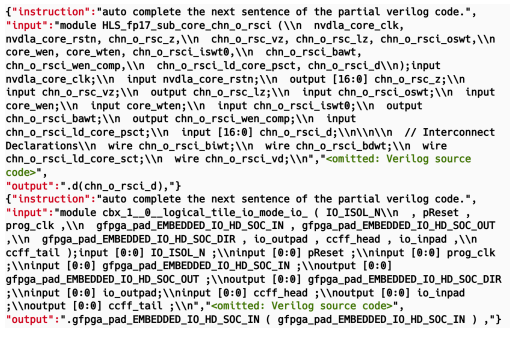
```
{"circuit_type": "sequential logic",
"problem": "Design a 2-bit up counter that increments its value on each clock
pulse and resets to 00 when the reset signal is high. The counter should have a
clock input, a reset input, and a 2-bit output.",
"verilog_code":"<verilog code>",
"verilog_testbench": "<verilog testbench>",
"python_code": "def counter(clk, reset): \\n count = 0 \\n if reset: \\n count =
0 \\n else: \\n count = (count + 1) % 4 \\n return count \\n\\ndef main(): \\n
with open('23_testcase.txt', 'w') as f: \\n # Test case 1 \\n clk = 0 \\n reset
= 1 \\n f.write(f\\\"{clk} {reset} {counter(clk, reset)}\\\\n\\\") \\n \\n #
Test case 2 \\n reset = 0 \\n f.write(f\\\"{clk} {reset} {counter(clk,
reset)}\\\\n\\\") \\n \\n # Test case 3 \\n f.write(f\\\"{clk} {reset}
{counter(clk, reset)}\\\\n\\\") \\n \\n # Test case 4 \\n f.write(f\\\"{clk}
{reset} {counter(clk, reset)}\\\\n\\\") \\n \\n # Test case 5 \\n f.write(f\\\"
{clk} {reset} {counter(clk, reset)}\\\\n\\\") \\n\\nif __name__ ==
\\\"__main__\\\": \\n main()",
"id": 23,
"level": "easy",
"problem_type": "2-bit counter"}
```


(a) AutoVCoder



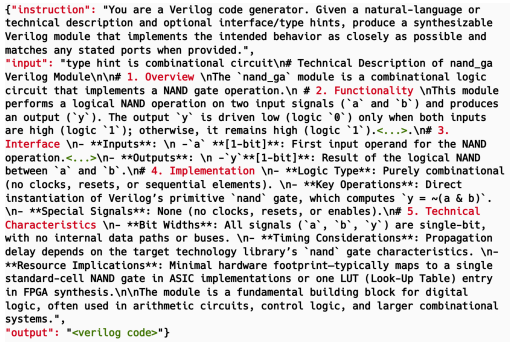
```
{"input":
"verilog Code": "This is the simplest form of inferring the //
simple/SRL(16/32)cE in a
XilinxFPGA. //`timescale ins / 100ps default nettype none (*
DowngradeIPIdentifiedwarnings="yes" *) moduleaxi protocol converter _v2 1 b2s
simple fifo # ( parameter C_WIDTH = 8, parameter C_AWIDTH = 4,parameter C_DEPTH
= 16 )( input wire clk, // Main system clock (Sync FIFO) input wire rst, //
FIFOcounter Reset (clk input wire wr en, // FIFO Write Enable (clk) input wire
rd en, // FIf0 Read Enable(Clk) input wire [c WIDTH-1:0] din, // FIFO Data Input
(clk) output wire [c WIDTH-1:0] dout, // FIFODaTa Output (clk) output wire a
full, output wire full, // FIFO FULL Status (Clk) output wire a empty.output
wire empty // FIFO EMPTY Status (Clk)): // FIFO Local Parameters
//////////////////t/ localparam[C_AWIDTH-1:0] C EMPTY = ~(0); localparam [C
AWIDTH-1:0]C EMPTY PRE = (0); localparam [C AWIDTH-1:@]C FULL = C EMPTY-1;
localparam [C AWIDTH-1:0]C FULL PRE =(C DEPTH <8)? C FULL-1 : C FULL-
(C_DEPTH/8); FIFO Internal signals /////// reg [c WIDTH-1:0] memory [ C DEPTH-
1:0]; reg [C AWIDTH- 1:0]cnt_read; // synthesis attribute MAX FANOUT of cnt read
is 10; //////////// <verilog code>",
"Verilog textbook corpus": "<textbook source>"}
```


(b) VeriGen



```
{"instruction":"auto complete the next sentence of the partial verilog code.",
"input":"module HLS_fp17_sub_core_chn_o_rsci (\\n  nvdla_core_clk,
nvdla_core_rstn, chn_o_rsc_z,\\n  chn_o_rsc_vz, chn_o_rsc_lz, chn_o_rsci_oswt,\\n
core_wen, core_wten, chn_o_rsci_iswt0,\\n  chn_o_rsci_bawt,
chn_o_rsci_wen_comp,\\n  chn_o_rsci_ld_core_psct, chn_o_rsci_d\\n);input
nvdla_core_clk;\\n  input nvdla_core_rstn;\\n  output [16:0] chn_o_rsc_z;\\n
input chn_o_rsc_vz;\\n  output chn_o_rsc_lz;\\n  input chn_o_rsci_oswt;\\n  input
core_wen;\\n  input core_wten;\\n  input chn_o_rsci_iswt0;\\n  output
chn_o_rsci_bawt;\\n  output chn_o_rsci_wen_comp;\\n  input
chn_o_rsci_ld_core_psct;\\n  input [16:0] chn_o_rsci_d;\\n\\n\\n  // Interconnect
Declarations\\n  wire chn_o_rsci_biwt;\\n  wire chn_o_rsci_bdwt;\\n  wire
chn_o_rsci_ld_core_sct;\\n  wire chn_o_rsci_vd;\\n","<omitted: Verilog source
code>",
"output":".d(chn_o_rsci_d),"}
{"instruction":"auto complete the next sentence of the partial verilog code.",
"input":"module cbx_1__0__logical_tile_io_mode_io_ ( IO_ISOL_N\\n  , pReset ,
prog_clk ,\\n  gfpga_pad_EMBEDDED_IO_HD_SOC_IN , gfpga_pad_EMBEDDED_IO_HD_SOC_OUT
,\\n  gfpga_pad_EMBEDDED_IO_HD_SOC_DIR , io_outpad , ccff_head , io_inpad ,\\n
ccff_tail );input [0:0] IO_ISOL_N ;\\ninput [0:0] pReset ;\\ninput [0:0] prog_clk
;\\ninput [0:0] gfpga_pad_EMBEDDED_IO_HD_SOC_IN ;\\noutput [0:0]
gfpga_pad_EMBEDDED_IO_HD_SOC_OUT ;\\noutput [0:0] gfpga_pad_EMBEDDED_IO_HD_SOC_DIR
;\\ninput [0:0] io_outpad;\\ninput [0:0] ccff_head ;\\noutput [0:0] io_inpad
;\\noutput [0:0] ccff_tail ;\\n","<omitted: Verilog source code>",
"output":".gfpga_pad_EMBEDDED_IO_HD_SOC_IN ( gfpga_pad_EMBEDDED_IO_HD_SOC_IN ) ,"}
```


(c) Chipgptseries



```
{"instruction": "You are a Verilog code generator. Given a natural-language or
technical description and optional interface/type hints, produce a synthesizable
Verilog module that implements the intended behavior as closely as possible and
matches any stated ports when provided.",
"input": "type hint is combinational circuit\n# Technical Description of nand_ga
Verilog Module\n\n# 1. Overview \nThe `nand_ga` module is a combinational logic
circuit that implements a NAND gate operation.\n # 2. Functionality \nThis module
performs a logical NAND operation on two input signals (`a` and `b`) and produces
an output (`y`). The output `y` is driven low (logic `0`) only when both inputs
are high (logic `1`); otherwise, it remains high (logic `1`).<...>.\n# 3.
Interface \n- **Inputs**: \n -`a` **[1-bit]**: First input operand for the NAND
operation.<...>\n- **Outputs**: \n -`y`**[1-bit]**: Result of the logical NAND
between `a` and `b`.\n# 4. Implementation \n- **Logic Type**: Purely combinational
(no clocks, resets, or sequential elements). \n- **Key Operations**: Direct
instantiation of Verilog's primitive `nand` gate, which computes `y = ~(a & b)`.
\n- **Special Signals**: None (no clocks, resets, or enables).\n# 5. Technical
Characteristics \n- **Bit Widths**: All signals (`a`, `b`, `y`) are single-bit,
with no internal data paths or buses. \n- **Timing Considerations**: Propagation
delay depends on the target technology library's `nand` gate characteristics. \n-
**Resource Implications**: Minimal hardware footprint—typically maps to a single
standard-cell NAND gate in ASIC implementations or one LUT (Look-Up Table) entry
in FPGA synthesis.\n\nThe module is a fundamental building block for digital
logic, often used in arithmetic circuits, control logic, and larger combinational
systems.",
"output": "<verilog code>"}
```


(d) Our Method

Fig. 2. Diverse Verilog Benchmark Dataset Examples

3.2.2 AST Construction. To construct a high-quality AST-based dataset for Verilog, this section proposes a unified semantic-enhanced parsing pipeline that combines robust preprocessing, dual-stage AST construction, and structured representation generation. This pipeline enables comprehensive semantic modeling while ensuring structural integrity.

Before constructing the AST, the Verilog source files undergo uniform preprocessing, including encoding detection, format normalization, BOM removal, and the unification of line break formats across multiple platforms, to ensure parsing stability and improve the accuracy of semantic analysis. Subsequently, core structural elements such as modules, ports, signals, and parameters are rapidly extracted using regular expressions to efficiently locate key structural elements and provide contextual support for subsequent semantic enhancement.

After preprocessing the Verilog source code, this work adopts a two-stage AST strategy to convert RTL descriptions into structured representations that are suitable for large-model training. In the first stage, Pyverilog is used to generate the baseline AST ($\mathcal{A}_p$). Essential structural elements such as modules, ports, and process blocks are parsed to ensure structural validity and source code traceability, which provides a reliable structural foundation for subsequent semantic enhancement.

In the second stage, this work performs static-analysis–driven semantic enhancement ($\Phi: \mathcal{A}_p \rightarrow \mathcal{A}_c$) explicitly for functional role inference and semantic injection. A set of predefined roles is defined, covering clock, reset (synchronous/asynchronous), enable, control/data signals, FSM states, counters, indices, shift registers, timing categories (Combinational/Sequential Logic) and so on. During depth-first traversal, this section assigns functional roles to AST nodes based on heuristic cues from static analysis, including naming rules, driver-load relationships, sensitivity lists, and

clock-edge information. The enriched nodes are further annotated with supplementary semantic attributes, such as port functionality, signal bit-widths, and timing characteristics. Child nodes are recursively processed to achieve the synchronous integration of semantic information. For each node, a structural summary is computed based on its type, attribute sequence, and child-node hash. This structural summary is used to enable fast equivalence detection and similarity retrieval, which improves semantic expressiveness while maintaining structural integrity.

Based on the semantically enhanced AST ($\mathcal{A}_c$), this section constructs a multi-relational heterogeneous directed graph, as shown in Equation (1). This graph unifies structural hierarchy, semantic roles, and dependency relations into a single representation, enabling fine-grained signal-level dependency reasoning and fast module-level similarity analysis.

$$G = (V, E_{\text{struct}} \cup E_{\text{data}} \cup E_{\text{control}}) \tag{1}$$

E_{struct}: Structural hierarchies among AST nodes, E_{data}: Signal-level data dependencies, E_{control}: Control-flow relationships between conditions and procedural blocks. This graph serves as an intermediate semantic representation that unifies structural hierarchy, signal dependencies, and functional roles within a single framework. By leveraging an optimized depth-first traversal with semantic caching, the graph efficiently captures signal-level data flows, clock/reset domains, and control constraints, while propagating role annotations and timing attributes along edges. These semantic attributes are integrated into the semantically enhanced RTL benchmark dataset, supporting the construction of B2 prompts with richer structural, temporal, and functional information. Furthermore, the graph facilitates module-level similarity computation via structural summaries, enabling dataset consistency checks and module reuse analysis. The specific algorithm flow is shown in Algorithm 1.

Algorithm 1 Semantic-Enhanced RTL Benchmark Dataset Construction

Require: Verilog set $V = \{v_1, \ldots, v_n\}$
Ensure: High-quality dataset D

$D \leftarrow \emptyset$
for all $v \in V$ **do**
  $code \leftarrow \textsc{Preprocess}(v)$ // encoding detection, normalization, regex hints
  $\mathcal{A}_p \leftarrow \textsc{PyverilogParse}(code)$ // basic syntax AST
  $\mathcal{A}_c \leftarrow \textsc{SemanticEnhance}(\mathcal{A}_p)$ // two-stage AST construction via $\Phi : \mathcal{A}_p \rightarrow \mathcal{A}_c$
  $G \leftarrow \textsc{Build heterogeneous directed graph}(\mathcal{A}_c)$ // construct heterogeneous directed graph $G = (V, E_{struct} \cup E_{data} \cup E_{control})$
  $Roles \leftarrow \textsc{InferRoles}(\mathcal{A}_c, G)$ // clk/rst/cnt role mapping
  $Patterns \leftarrow \textsc{RecognizePatterns}(\mathcal{A}_c, G, Roles)$ // FSM/counter/reset pattern recognition
  $Json_{\text{sem}} \leftarrow \textsc{SerializeToJSON}(\mathcal{A}_c, Roles, Patterns)$ // structured representation
  $Sample \leftarrow \{src{:}\ v,\ ast{:}\ \mathcal{A}_c,\ roles{:}\ Roles,\ patterns{:}\ Patterns,\ json{:}\ Json_{\text{sem}}\}$
  $D \leftarrow D \cup \{Sample\}$
end for
return D

The generated semantic descriptions and annotations are then integrated into a semantically enhanced RTL training dataset, providing a standardized representation for subsequent multi-task learning and model training. Based on the DeepSeek Application Programming Interface (API), the semantic enhancement method further refines functional roles across multiple dimensions,

encompassing functional descriptions, interface analysis, implementation details, and technical features.

3.2.3 Semantic-Aware Repair for Dataset Enhancement. This paper proposes an agent-based, structured prompt automatic repair algorithm for Verilog designs. Leveraging a semantically enhanced AST as the core representation, the algorithm provides fine-grained semantic understanding and structural consistency constraints during the repair process. Through a closed loop "compile–fix–verify" iteration, compilation errors are automatically resolved. The overall algorithm flow is illustrated in Figure 3.

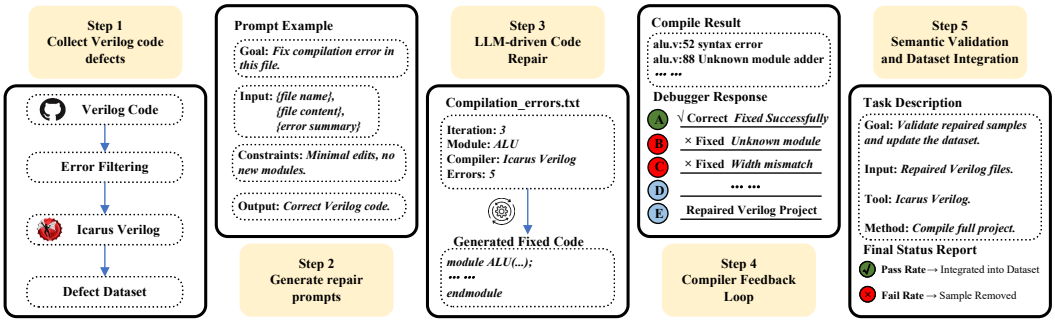


Fig. 3. Compiler Feedback-driven Intelligent Agent Verilog Code Repair System

Given a design composed of n Verilog source files $\mathcal{P} = f_1, \ldots, f_n$, the algorithm operates on an internal working set $\widehat{\mathcal{P}}$ to preserve the integrity of the sources. Compilation and simulation are performed using Icarus Verilog to collect structured error information $\mathcal{E}$, including error locations, categories, and diagnostic messages. Based on $\mathcal{E}$, the repair agent applies a structured prompting mechanism combined with contextual analysis to generate targeted local repair solutions, and a multi-level verification strategy adaptively determines whether to continue iterative fixing or terminate early. The Iterative Repair Workflow, as shown in Equation (2), guides the symbol-based reasoning process using feedback derived from compilation results.

$$\widehat{\mathcal{P}} \xrightarrow{\text{Compile}} \mathcal{E} \xrightarrow{\text{Agent(Prompt)}} \widehat{\mathcal{P}}' \xrightarrow{\text{ReCompile}} \cdots \rightarrow \widehat{\mathcal{P}}^* \tag{2}$$

$\widehat{\mathcal{P}}^*$ denotes the final design version that is both compilable and functionally correct. To constrain the generation behavior of the repair agent and ensure executability and design consistency, a structured prompting template provides constraint-aware and context-rich guidance during each iteration. After the repair, each sample is re-parsed into an AST-consistent semantic representation and written back to the dataset, thereby improving dataset quality, semantic expressiveness, and design coverage.

4 Study Results

4.1 RQ-1: AST-Guided Structural and Semantic Characterization of Verilog Code

This study conducts a multi-faceted analysis to characterize the compositional patterns and quality attributes of the constructed Verilog dataset. Dataset size, ablation experiments, and experimental verification of dataset quality. These analyses provide a comprehensive overview of the dataset's quality, diversity, and robustness.

4.1.1 Complexity Evaluation. To systematically evaluate the complexity of the dataset, this study extracts multiple complexity metrics across key dimensions based on the AST representation of each Verilog sample [3]. For features exhibiting significant skewness, a logarithmic transformation is applied to normalize distributions. Feature scaling is performed using robust normalization to mitigate outlier effects. To improve discriminative power, a weighted aggregation framework integrates dimensional scores, augmented by a nonlinear mapping function. Finally, the samples were classified into three complexity levels using the quantile method: Simple, Intermediate, and Advanced. A simple design corresponds to samples below the 25th percentile (Q1), an intermediate design refers to samples between Q1 and the 75th percentile (Q3), and a complex design includes samples above Q3 [38]. This classification accounts for features such as multi-module instantiation, cross-clock domain design, and parametric generation, enabling a comprehensive and quantitative evaluation of Verilog implementation complexity. The original dataset consists of approximately 318,021 Verilog modules. To enhance training efficiency and reduce computational overhead, samples longer than 4,096 tokens were excluded. After preprocessing, roughly 170,000 samples were utilized for model training.

Table 1. Verilog Module Complexity Metrics

Level	Sample Size	LOC	LOC Range	Modules	Always	Ports	Instance Count	A Width
Simple	42,106	14	2-241	1-10	0-1	4	0-3	30.2
Medium	84,210	42	1-1,815	1-138	0-16	6	0-9	185.5
Hard	42,101	136	10-1,469	1-50	0-69	12	0-175	465.6

As shown in Table 1, the dataset exhibits relatively balanced distributions across complexity levels, with intermediate designs dominating (approximately 50%) and both simple and advanced designs each accounting for approximately 25%. Moreover, design complexity is influenced by multiple factors beyond code size, including procedural control, interface width, and hierarchical instantiations. Compared to simple designs, advanced designs exhibit substantially more procedural blocks and higher instance and expression counts, indicating deeper structural and behavioral complexity. These characteristics suggest higher verification costs and integration challenges for advanced designs, highlighting the necessity of multi-dimensional metrics for evaluating Verilog complexity.

4.1.2 Comparison of Dataset Sizes. To better demonstrate the diversity and scale of datasets used for RTL code generation, this experiment compares the sample sizes of existing benchmarks and our proposed dataset. As shown in Table 2, our dataset significantly exceeds the sizes of AutoVCoder, VeriGen, chipgptseries and GEMM, providing a larger number of high-quality Verilog modules and spec-to-code pairs. This substantial scale offers better coverage of design patterns and enhances the generalization capability of models trained on our dataset.

Since the primary focus of this paper is the construction of benchmark datasets for Verilog code generation tasks, this section primarily compares our dataset with natural language Verilog Generation benchmarks that are also designed for code generation. AutoVCoder and VeriGen focus on high-quality Verilog modules suitable for large-scale model pretraining. Chipgptseries provides a large-scale benchmark for general-purpose code generation. GEMMV-DS contributes specialized data for GEMM accelerators and performance-oriented hardware studies. In comparison, our dataset integrates a large collection of diverse Verilog modules with a substantial amount of structured code, enabling both broad coverage of RTL design patterns and task-driven instruction tuning. This hybrid composition not only supports large-scale domain-adaptive pretraining but

Table 2. Comparison of Dataset Sizes for RTL Code Generation

Dataset	Samples	Type
AutoVCoder [12]	≈ 50,000	Spec-to-Verilog
VeriGen [44]	51,826	Spec-to-Verilog
chipgptseries [5]	124,000	Spec-to-Verilog
GEMMV [56]	12,519	Domain-Specific RTL Generation
Ours	**318,021**	Spec-to-Verilog
Ours	**168,417**	Fine-tuning Dataset

also facilitates fine-grained supervised learning, thereby enhancing semantic understanding and significantly boosting the generalization capability of LLMs for Verilog code generation.

4.1.3 Evaluation of Closed-Loop Automatic Repair Framework. This experiment aims to comprehensively evaluate the effectiveness of the LLM-based closed-loop automatic repair framework for restoring failed samples due to compilation and simulation errors. To evaluate the repair framework, this section uses Repair Rate as the primary metric, which measures the proportion of failed samples successfully repaired, reflecting the system's overall recovery capability. Furthermore, the error distribution across the complete and restored datasets is analyzed to reveal the framework's adaptability across diverse failure scenarios.

Table 3. Error Statistics on Complete vs. Restored Datasets

Error Type [41]	Complete Dataset		Restored Dataset		Repair Rate(%)
	Sample Size	Proportion(%)	Sample Size	Proportion(%)	
Module and instantiation errors	291,661	73.54	44,967	71.63	15.42 ↑
Grammatical and structural errors	45,837	11.56	6,511	10.37	14.20 ↑
Type and width errors	12,795	3.23	2,541	4.05	19.86 ↑
Variable and declaration errors	37,460	9.45	7,704	12.27	20.57 ↑
Expression and operation errors	5,022	1.27	299	0.48	5.95 ↑
Port and connection errors	2,035	0.51	225	0.36	11.06 ↑
Range and index errors	599	0.15	111	0.18	18.53 ↑
Else	1,180	0.30	419	0.67	35.51 ↑
All	396,589	100.00	62,777	100.00	15.83 ↑

As shown in Table 3. The errors in RTL design primarily stem from challenges arising from design complexity. Statistical analysis of the complete dataset reveals that module and instantiation errors are dominant (73.54%), typically caused by cross-module interface mismatches that require precise parameter configuration and port mapping. Type and width errors (3.23%) mainly result from signal bit width mismatches, which can trigger numerous design inconsistency issues.

To address these challenges, this section proposes a structured error identification and repair framework based on AST-enhanced defect analysis. Experimental results demonstrate that this framework achieves remarkable improvements across various error types. Although expression and operation errors are inherently difficult to repair, they still achieve a 5.95% repair success rate, with successful cases benefiting from accurate AST-based reconstruction of localized structural patterns. The fix rate for variable and declaration errors reaches 20.57%, benefiting from AST's ability to construct precise definitions, usage relationship diagrams, and systematic type derivations. Meanwhile, agent-based reasoning resolves most bit-width mismatch problems, achieving a repair rate of 19.86% for type and width errors. For low-complexity samples, approximately 90%

are successfully repaired within the first 20 iterations, while high-complexity samples require 40 iterations to restore grammatical and semantic correctness. To balance repair accuracy with engineering efficiency, this section sets an upper limit of 40 repair iterations to ensure the repaired dataset retains the difficulty distribution and diversity of the original dataset, thereby guaranteeing representativeness for downstream model training and evaluation.

This experiment constructed a large-scale Verilog dataset comprising 318,021 samples. Through multi-dimensional complexity evaluation and semantic enhancement strategies, its effectiveness in hardware design automation was verified. Complexity features were extracted using AST representations, and the quantile method was applied to classify the samples. The resulting distribution accurately reflects the current state of industrial hardware design: medium-complexity functional units dominate, while system-level complex designs are relatively rare due to their high verification costs and integration challenges. Compared with existing benchmarks, this dataset is significantly larger in scale and offers more comprehensive coverage of design patterns for model training.

Answer to RQ-1.

The dataset exhibits balanced complexity distribution (simple, intermediate and advanced) using multi-dimensional AST-based metrics including LOC, module count, and procedural blocks. The repaired defects span multiple error categories, including instantiation, syntactic/structural issues, and type/bit-width mismatches. The closed-loop repair framework achieves 15.83% overall repair success rate, effectively improving dataset quality while preserving difficulty distribution.

4.2 RQ-2: Enhancing Functional Correctness through Multi-Level Semantic Constraints

This study systematically evaluates the impact of different levels of prompt structuring on the quality, consistency, and reliability of Verilog code generation.

This experiment evaluates three distinct prompting approaches: B0 (Natural Language Prompt): Provides only the natural language description and interface requirements for the target module. B1 (Structured JSON Format): Incorporates a JSON-serialized AST skeleton derived from syntactic parsing, containing fundamental structural elements (module names, ports, etc.) while excluding semantic enhancements. B2 (Ours): The complete semantic features, such as signal roles and clock/reset attributes of the enhanced AST are extracted from the module to form a constrained prompt, maximizing the retention of structural and temporal information.

To assess executability and functional correctness, we adopt three key metrics: CR (compilation rate), SR (simulation success rate) and SC (structural consistency) [27]. A stratified balanced sampling strategy is adopted, where each prompting group (B0, B1, B2) contains 2,000 Verilog modules. This sample size was chosen based on statistical calculations to ensure a 95% confidence level (under p=0.5 assumption) and an error margin of approximately ±2% [40], helping to maintain the representativeness and diversity of the dataset and ensuring that the evaluation process is both comprehensive and reliable. Before making API calls, this experiment conducts comprehensive quality control on the B0, B1, and B2 datasets, verifying each sample's structural integrity, detecting potential unintended solution leakage, and flagging excessively long prompts that may exceed the model's context window [52]. This rigorous preprocessing ensures that all evaluated samples are valid and maintains experimental fairness throughout this study.

For each strategy, the unified model is called to generate Verilog code, compiled using Icarus Verilog. Throughout the experiment, random seeds were fixed, and the results were cached to ensure reproducibility. This methodology excludes any intelligent inference or enhancement processing, thereby isolating and accurately measuring the intrinsic effects of different prompt formats on model outputs. The specific results are shown in Figure 4. Among the three prompting strategies,

B0 produces the most SR but suffers from severe hallucination, leading to poor structural and semantic consistency. B1 mitigates this issue by providing a more reliable module skeleton, yet it lacks semantic constraints and still yields insufficient functional correctness. In contrast, B2 combines AST structure and functional role information, thereby constraining model outputs and significantly improving structural consistency and functional reliability. Experimental results show that B2 achieves the highest Avg SC (52.35%), outperforming B1 (29.63%) and B0 (17.60%) by 22.72 and 34.75 percentage points, respectively, while maintaining a high Avg CR of 98.67%.

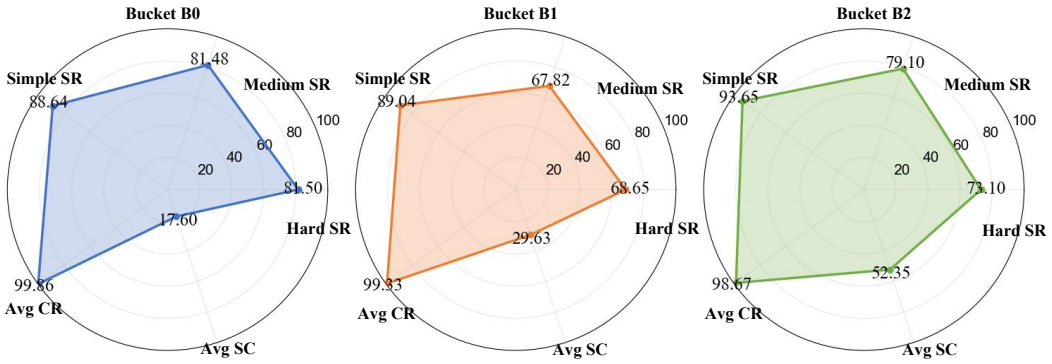


Fig. 4. Bucket Validation Results

This dataset is constructed from the original Verilog source files through Pyverilog parsing and heuristic semantic enhancement. The parsing process directly extracts the module port, bit width, clock/reset semantics and structure, data, and control class dependencies from the source code. The final dataset also contains the original .v file. Therefore, since all structured fields are derived from the automatic parsing results and have not undergone manual secondary modification, the data is naturally consistent with the original RTL code in terms of structure and interface attributes. Thus, there is no need to conduct additional consistency comparison experiments. The experiment demonstrates that prompts with higher completeness and structural organization lead to significantly improved code generation success rates, functional accuracy, and implementation reliability.

Answer to RQ-2.

Prompt structuring impacts code generation quality through three progressive mechanisms. B0 provides only functional descriptions and interface requirements, achieving high success rates but causing severe hallucinations due to lack of structural constraints; structured prompts. B1 improves reliability by providing detailed structural information and module frameworks, but lacks semantic guidance. B2 integrates signal-role and clock/reset constraints, achieving superior structural consistency while maintaining a high compilation rate.

4.3 RQ-3: Cross-Model Evaluation of General-Purpose and Specialized LLMs on Verilog Benchmarks

To enhance the Verilog code generation capability, this experiment fine-tunes the CodeQwen-7B, DeepSeek-Coder-6.7B, Mistral-v0.3-7B, and Qwen3-8B models using the semantically enhanced dataset constructed in this study. The dataset consists of structured problem–code pairs, where each sample includes a formal design requirement description and its corresponding verified Verilog code.

To reduce computational overhead while maintaining training effectiveness, this section adopts LoRA for parameter-efficient, instruction-supervised fine-tuning, optimizing the models with the cross-entropy loss to ensure consistency between the generated and target implementations. During evaluation, the Icarus Verilog simulator is used to verify both syntax and functionality correctness of the generated code. This section further evaluates the effectiveness of our method by comparing the fine-tuned models with state-of-the-art models and existing baseline methods.

4.3.1 Benchmark Evaluation. The evaluation is conducted on multiple benchmarks, including VerilogEval [23] and RTLLM v1.1 [27]. In accordance with standard practice, this experiment focuses on pass@1 and pass@5 metrics, along with syntactic and semantic checks from RTLLM.

As shown in Table 4, our method achieves consistent and substantial improvements in Verilog code generation across diverse baselines.

Table 4. Benchmarking Verilog Code Generation on VerilogEval and RTLLM Benchmarks

Model Type	Evaluated Model	Params	Eval-Machine (%)		Eval-Human (%)		RTLLM V1.1 (%)	
			K=1	K=5	K=1	K=5	Syntax-VCS	Func
Closed-Source Baseline	GPT-4 [1]	N/A	**60.00**	70.60	**43.50**	**55.80**	**100.00**	**65.50**
	ChipNeMo [22]	13B	43.40	N/A	22.40	N/A	N/A	N/A
	VerilogEval [23]	16B	46.20	67.30	28.80	45.90	N/A	N/A
Open-Source Baseline	Codegen2 [33]	16B	5.00	9.00	0.90	4.10	72.40	6.90
	Starcoder [21]	15B	46.80	54.50	18.10	26.10	**93.10**	27.60
	rtl code [43]	16B	44.00	52.60	30.30	43.90	86.20	24.10
	MG-Verilog [58]	7B	52.70	58.50	N/A	N/A	N/A	N/A
	asic [14]	7B	40.60	48.40	N/A	N/A	N/A	N/A
Base Model	CodeQwen1.5 [2]	7B	51.26	74.63	23.97	44.58	83.52	36.03
	DeepSeek-Coder [17]	6.7B	43.84	63.18	23.59	38.44	88.85	35.36
	Mistral-v0.3 [31]	7B	21.61	36.77	5.00	11.89	31.90	8.62
	Qwen3(no think) [51]	8B	43.85	48.36	28.21	31.87	86.48	42.99
Our Method	CodeQwen1.5 [2]	7B	**60.28**	**79.74**	30.58	**51.63**	86.89	46.82
	DeepSeek-Coder [17]	6.7B	48.81	**77.30**	25.45	48.66	90.28	**49.86**
	Mistral-v0.3 [31]	7B	50.77	64.31	25.64	38.92	88.45	31.42
	Qwen3(no think) [51]	8B	55.73	69.38	**35.00**	45.81	87.93	43.68

In the VerilogEval benchmark, the CodeQwen-7B model improves by 9.02% and 5.11% on the Eval-Machine metric. Its performance with K=1 and K=5 reaches 60.28% and 79.74%, respectively, surpassing GPT-4 (60.00% and 70.60%), validating the effectiveness of fine-tuning with high-quality datasets. The most substantial improvement is observed in the Mistral v0.3-7B model. Under the K=1 setting of Eval-Machine, its accuracy rises from 21.61% to 50.77%, achieving a 29.16% improvement. These results demonstrate that our method is particularly effective for general-purpose baseline models that initially perform poorly on Verilog-related tasks.

In the Eval-Human benchmark, CodeQwen-7B achieves absolute improvements of 6.61% and 7.05%. Similarly, the DeepSeek-Coder-6.7B model exhibits strong performance, achieving a 14.12% improvement under the K=5 setting in Eval-Machine, increasing from 63.18% to 77.30%.

In the RTLLM v1.1 benchmark, most baseline models achieve syntactic correctness exceeding 80.00%. While our models may not achieve the highest syntax scores, circuit verification primarily focuses on functional correctness, where our approach demonstrates strong performance, ensuring effective overall results. By fine-tuning on our dataset, CodeQwen improves by 10.79%, and DeepSeek-Coder improves by 14.50% in functional correctness. Notably, Mistral achieves a remarkable improvement of 22.80%. Although it still falls below specialized baselines, this demonstrates the importance of domain-specific fine-tuning.

Overall, the proposed method leverages a high-quality Verilog dataset to enable general-purpose LLMs to achieve substantial performance gains in HDL-related tasks. The fine-tuned open-source baseline models approach the performance of GPT-4 on multiple metrics, demonstrating that, with precisely constructed datasets, open-source models can compete with closed-source models in professional Verilog code generation. This provides an efficient, cost-effective, and high-performance alternative for the real-world deployment of automatic Verilog code generation systems.

4.3.2 Impact of Model Scaling and Task Complexity. To further evaluate the generalizability of our approach across different model architectures and task complexities, we conducted comprehensive comparisons on different model scales and two distinct Verilog-related tasks using VerilogEval v2 [35].

As shown in Table 5, our fine-tuning method demonstrates significant improvements across different model architectures.

Table 5. Model Comparison for Verilog Tasks

Model Name	Model Size	License	Type	Task: Code Completion	Task: Specification to RTL
In-Context Learning Examples:				**1-shot**	**1-shot**
Temperature:				**T=0**	**T=0**
GPT-4 [1]	Undisclosed	Closed	General	51.30%	48.70%
Mistral Large [30]	Undisclosed	Closed	General	44.20%	48.70%
Llama3.1 [29]	70B	Open	General	34.00%	48.00%
Llama3 [29]	70B	Open	General	36.50%	39.50%
Llama2 [29]	70B	Open	General	15.40%	17.80%
CodeLlama [28]	70B	Open	General	41.70%	41.50%
DeepSeek-Coder [17]	33B	Open	General	42.30%	40.10%
Llama 3.1 [29]	8B	Open	General	10.90%	27.60%
Code-Gemma [15]	7B	Open	Coding	19.90%	24.30%
RTL-Coder [25]	6.7B	Open	Verilog RTL	37.20%	34.90%
DeepSeek-Coder [17]	6.7B	Open	Coding	33.30%	27.60%
CodeQwen1.5 [2]	7B	Open	Coding	31.69%	34.62%
Mistral-v0.3 [31]	7B	Open	Coding	18.59%	14.74%
Qwen3(no think) [51]	8B	Open	Coding	39.10%	33.97%
DeepSeek-Coder [17]	**6.7B**	**Fine-tuning**	**Coding**	**36.54%**	**42.95%**
CodeQwen1.5 [2]	**7B**	**Fine-tuning**	**Coding**	**33.97%**	**37.82%**
Mistral-v0.3 [31]	**7B**	**Fine-tuning**	**Coding**	**28.21%**	**30.77%**
Qwen3(no think) [51]	**8B**	**Fine-tuning**	**Coding**	**46.15%**	**43.56%**

In the Code Completion task, while GPT-4 achieved an accuracy of 51.30%, our approach enabled smaller models to deliver remarkable results. Specifically, Qwen3-8B (no think) achieved 46.15% performance, surpassing much larger models including Llama3.1-70B (34.00%) and CodeLlama-70B (41.70%), and even outperforming the closed-source model Mistral Large (44.20%). All baseline models demonstrated substantial improvements: DeepSeek-Coder-6.7B improved from 33.30% to 36.54%, while Mistral-v0.3 surged from 18.59% to 28.21%.

The closed-source model GPT-4 maintains the leading position with an accuracy rate of 51.30%. Among open-source models, large-parameter models demonstrate a significant scale advantage, with DeepSeek-Coder-33B achieving 42.30% performance. The medium and small-sized models fine-tuned using the method proposed in this paper also demonstrated significant performance improvements.

In the Specification to RTL conversion task, the performance improvement is more pronounced. DeepSeek-Coder-6.7B increased substantially from the baseline of 27.60% to 42.95%, achieving an improvement of 15.35%. This performance notably exceeds the specialized RTL-Coder-6.7B (34.90%) and approaches larger general-purpose models such as Llama3.1-70B (48.00%) and Mistral

Large (48.70%). Similarly, Qwen3-8B represents an enhancement of 9.59%, getting closer to the performance levels of GPT-4.

Our fine-tuning method demonstrates consistent and substantial improvements across multiple benchmarks and model architectures. The results show that with high-quality domain-specific datasets, smaller open-source models can achieve competitive performance with larger closed-source models.

Answer to RQ-3.

The fine-tuning method demonstrates stable gains and strong transferability across different model architectures and task complexities. For code-specific models, CodeQwen-7B reaches 60.28% surpassing GPT-4, with some open-source models matching or exceeding closed-source performance. For general-purpose models, Mistral-v0.3 improves by 29.16%, achieving significant enhancement on initially weak-performing models. In cross-task evaluation, smaller models like Qwen3-8B outperform multiple 70B-scale models. The method exhibits strong generalization across model scales and task difficulties.

5 Discussion

This study demonstrates that the performance of large language models in Verilog code generation is governed by a complex interplay of factors, extending beyond model size and architectural choices to include the quality of the training dataset and the model used for fine-tuning. Drawing on the experimental results of RQ-1, RQ-2, and RQ-3, we summarize the main conclusions as follows.

5.1 High-Quality Datasets Analysis

The dataset constructed in this study has been systematically designed with respect to scale, complexity stratification, and semantic enhancement, enabling more comprehensive coverage of diverse Verilog modules and design patterns. In terms of sample size, semantic consistency, and automated repair mechanisms, the dataset offers richer and higher-quality information, thereby improving its usability and stability for subsequent model training and code generation tasks.

By combining an AST-annotated dataset with a data-centric closed-loop repair process, both the proportion of compilable/simulable samples and the consistency of structural and interface definitions have been substantially improved. In RQ-1, error distribution analysis reveals that module/instantiation-related issues constitute the primary source of errors. Among these, the closed-loop repair mechanism demonstrates the most significant effectiveness in addressing module/instantiation and type/bit-width issues, whereas its impact on complex structural or timing-related problems remains relatively limited.

Overall, the proposed methodology establishes a solid foundation for advancements in prompt structuring and model fine-tuning, ultimately enhancing the performance of Verilog code generation models.

5.2 Functional Correctness Analysis

In RQ-2, by comparing the three prompt strategies (B0, B1, and B2), this work observed that B2 (the semantically enhanced AST prompt) consistently outperforms both B0 and B1 overall. Compared with B0, B2 significantly reduces the ambiguity inherent in natural language descriptions by introducing structured information. Compared with B1, B2 further integrates richer semantic features on top of structural representation. These additional semantic cues make the prompt content more comprehensive and instructive, effectively guiding the model toward more precise

logical reasoning, which in turn improves structural consistency while maintaining high compilation and simulation success rates.

5.3 Model Fine-Tuning Analysis

In RQ-3, compared with the unfine-tuned baseline, the correctness of code generation has been significantly improved after tuning. The experimental results reveal several key findings. First, our fine-tuning method enables 6.7B-8B models to match the performance of much larger models, providing significant value for resource-constrained deployments. Second, since our dataset focuses on generation tasks, the improvements are more pronounced on RTL generation, aligning with our design objectives. Third, models with weaker baseline performance achieve substantial gains, demonstrating the method's generalizability and effectiveness.

These experimental results comprehensively validate the effectiveness of the proposed fine-tuning method for Verilog code generation tasks. The approach demonstrates significant advantages in balancing model performance and computational cost, providing a cost-effective solution for practical applications.

6 Threats to Validity

This study proposes a semantically enhanced AST framework, a closed-loop dataset construction method, and a prompt strategy optimization for the Verilog code generation task. To ensure the reliability of our research findings, we analyze the potential threats to validity that may affect the results and propose corresponding mitigation measures.

6.1 Threats to External Validity

External validity concerns the generalizability of research findings across different languages, models, and industrial environments. The semantically enhanced AST framework and the closed-loop dataset construction method proposed in this study natively support two hardware description languages, Verilog and SystemVerilog, and can uniformly extract key information such as syntactic constructs, signal roles, inter-module dependencies, and cross-clock-domain constraints. However, SystemVerilog differs significantly from Verilog in syntax structures, type systems, and modeling paradigms (such as interfaces, assertions, and richer data types). Although the experiment can parse SystemVerilog, the benchmark datasets focus on Verilog since the existing benchmarks are mainly based on Verilog.

6.2 Threats to Internal Validity

Internal validity concerns the potential confounding factors in the experimental process and their influence on causal conclusions. In this study, Icarus Verilog is adopted as the primary compilation and simulation tool. Although widely used in academic research, different tools may vary in syntax analysis, timing checks, and coverage statistics. To mitigate this threat, we fixed the toolchain version to ensure reproducibility of the experimental results. Due to cost constraints, multi-emulator verification was not conducted in this study. In future work, we plan to perform consistency analyses under a multi-emulator environment.

Additionally, this study proposes a closed-loop automatic repair mechanism for large-scale Verilog code. The iterative repair process is achieved through multi-round interactions between the Agent architecture and EDA tools. Theoretically, given unlimited iterations, the mechanism can achieve a near-100% repair rate since each round performs locally optimized fixes based on precise error locations and contextual semantic constraints. Considering engineering efficiency and computational cost, we set an upper bound on the number of repair iterations to balance repair efficiency and resource consumption.

6.3 Construct Validity

Structural effectiveness evaluates whether the selected indicators comprehensively capture the research objectives. In this study, we primarily adopt CR, SR, and SC as the core evaluation metrics, which effectively measure the executability and functional correctness of the generated code. However, the correct implementation of a function does not necessarily indicate a high-quality design. Since this study primarily focuses on code executability and functional correctness rather than circuit-level performance optimization, the current set of indicators does not yet cover comprehensive performance dimensions—including area utilization and power consumption. In future work, we plan to incorporate PPA-related metrics (Performance, Power, Area) to establish a more holistic code quality assessment framework.

6.4 Conclusion Validity

Conclusion validity concerns the reliability and statistical robustness of the research inferences. In this study, we evaluated 6,000 samples in RQ-2, with 2,000 samples allocated to each group. At a 95% confidence level, we controlled the margin of error within ±2%, thereby enhancing the credibility of our statistical conclusions. However, due to the lack of industrial-scale data, certain high-complexity subsets and rare error categories still suffer from limited sample sizes, which may introduce fluctuations in the evaluation results. This issue will be further investigated in future work.

In RQ-3, we assessed the impact of fine-tuning on model performance. The fine-tuned model demonstrates significant improvements over the baseline in both functional correctness and structural consistency. The reliability of this conclusion, however, largely depends on the quality of the training data and the independence of the evaluation data. To avoid overestimating the effect of fine-tuning, we strictly ensured complete separation between the training and test datasets. All test samples were drawn from independent datasets that were never used during fine-tuning.

7 Related Work

With the rapid advancement of LLMs, HDL code generation based on LLMs has emerged as a key research direction in electronic design automation. Current research can be broadly divided into code generation and code repair tasks.

Code Generation Tasks. In terms of benchmark testing and evaluation frameworks, Liu et al. [23] constructed the VerilogEval evaluation framework. By establishing a dataset containing 156 diverse questions, they addressed the issue of insufficient performance evaluation of LLMs in Verilog code generation. Lu et al. [27] developed the RTLLM open-source benchmark, integrating 30 digital designs of varying complexity and assessing design quality. Dehaerne et al. [8] developed a deep learning-based auto-completion method that leverages large-scale pre-training and Verilog-specific fine-tuning to improve module completion efficiency.

In terms of model training optimization, Liu et al. [22] proposed the ChipNeMo, which employs techniques including domain-adaptive pre-training (DAPT), model alignment, and retrieval augmented generation (RAG) to adapt LLMs to industrial chip design tasks. Thakur et al. [44] addressed the problem of unreliable code generation by existing commercial models in hardware design scenarios. By combining open-source GitHub code with Verilog textbook materials, they reduced syntax errors and functional defects in HDL generation. Liu et al. [25] proposed RTLCoder, which employs an automated dataset pipeline and reinforcement learning to address the performance gap among RTL generation tasks.

In terms of improving code generation quality, Thakur et al. proposed AutoChip [45], which improves zero-shot accuracy through iterative compiler-guided feedback. Huang et al. [19] applied

multi-round interaction and chain-of-thought reasoning, mitigating the issues of LLMs neglecting temporal constraints and exhibiting low functional correctness when generating code. Pei et al. [55] proposed BetterV, which integrates instruction tuning with a generative discriminator to improve efficiency in Verilog code generation.

However, existing datasets still lack sufficient diversity and system-level coverage, providing limited support for cross-module instantiation, hierarchical dependencies, and SoC-level designs. As a result, they fail to evaluate LLM performance on complex circuit designs comprehensively. These limitations highlight the need for a large-scale, high-quality, and semantically rich dataset to better support Verilog code generation.

Code Repair Tasks. In Verilog code debugging, LLMs face several challenges [24]. First, Verilog's complex syntax and fragmented compiler diagnostics make it difficult to identify and repair errors. Second, the debugging process often lacks sufficient contextual information and clear correction guidelines, with error logs frequently disconnected from root causes. Moreover, hardware design introduces dynamic issues such as concurrent faults from multi-module interactions and cross-clock-domain signal constraints. These factors are hard to capture through static analysis, forming multi-level obstacles for LLM-based Verilog debugging.

To improve debugging efficiency and handle logical flows and data operations in LLM-generated Verilog, Li et al. [36] proposed a method that decomposes programs into basic blocks and extracts execution trajectories, effectively simulating breakpoint debugging. Tsai et al. [46] developed RTLFixer, which converts LLMs into reasoning agents supported by a human expert knowledge base for step-by-step error analysis, knowledge retrieval, and code revision. Yao et al. [54] proposed HDLDebugger, which reverse-engineers error cases, generates compiler messages, and employs dual-channel retrieval with chain-of-thought fine-tuning to improve repair success rates.

To address these issues, this paper proposes a generation strategy based on semantically enhanced AST and integrates it with the automatic repair method that leverages AST to achieve error detection, targeted repair, and high-quality sample supplementation during dataset construction. This forms a closed-loop data enhancement pipeline, significantly improving both dataset quality and the model's code generation capabilities.

8 Conclusion

This study proposes a semantically enhanced AST framework and an automated repair mechanism for Verilog to address the limitations of existing datasets. This work constructed a large-scale benchmark dataset comprising 318,021 high-quality samples, obtained through rigorous function validation and an AST-based repair process, achieving a 15.24% recovery rate for the error code. This dataset significantly enriches design pattern diversity and provides reliable support for the efficient fine-tuning of open-source large models. Experimental results show that CodeQwen1.5-7B outperforms GPT-4 on the Eval-Machine metrics of VerilogEval, while Qwen3-8B surpasses several 70B-scale open-source models on the Specification-to-RTL task. Other small-sized models, such as Mistral-7B and DeepSeek-Coder-6.7B, also substantially outperform existing open-source baselines across multiple metrics. Overall, domain-specific datasets can effectively empower small and medium-scale models, enabling them not only to rival or even surpass closed-source and ultra-large-scale models, but also to demonstrate clear advantages when compared at the same scale.

Data-Availability Statement

The replication package, including the code and a sample dataset, is available at: https://github.com/shiyiyanqin/Verilog-benchmark. The README in the package provides the link to the complete dataset for download.

Acknowledgements

The work is supported in part by the Natural Science Research Start-up Foundation of Recruiting Talents of Nanjing University of Posts and Telecommunications (Grant No. NY22502), the National Natural Science Foundation of China (Grant No. 62371256), the Self-developed Project of Nanjing University of Posts and Telecommunications Nantong Research Institute Co., Ltd. (Grant No. ZY-001), and the Natural Science Research Project of Higher Education Institutions in Jiangsu Province (Grant No. 22KJA510010).

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, et al. 2023. Gpt-4 technical report. arXiv:2303.08774 [cs.CL] doi:10.48550/arXiv.2303.08774

[2] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Fei Huang, et al. 2023. Qwen technical report. arXiv:2309.16609 [cs.CL] doi:10.48550/arXiv.2309.16609

[3] Dan Braha and Oded Maimon. 2002. The measurement of a design structural and functional complexity. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans* 28, 4 (2002), 527–535. doi:10.1109/3468.686715

[4] Paul E Calzada, Zahin Ibnat, Tanvir Rahman, Kamal Kandula, Danyu Lu, Sujan Kumar Saha, Farimah Farahmandi, and Mark Tehranipoor. 2025. VerilogDB: The Largest, Highest-Quality Dataset with a Preprocessing Framework for LLM-based RTL Generation. arXiv:2507.13369 [cs.AR] doi:10.48550/arXiv.2507.13369

[5] Kaiyan Chang, Kun Wang, Nan Yang, Ying Wang, Dantong Jin, Wenlong Zhu, Zhirong Chen, Cangyuan Li, Hao Yan, Yunhao Zhou, et al. 2024. Data is all you need: Finetuning llms for chip design via an automated design-data augmentation framework. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6. doi:10.1145/3649329.3657356

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] doi:10.48550/arXiv.2107.03374

[7] Alain Dargelas and Henner Zeller. 2020. Universal hardware data model. In *Workshop on Open-Source EDA Technology (WOSET)*. https://woset-workshop.github.io/PDFs/2020/a10.pdf

[8] Enrique Dehaerne, Bappaditya Dey, Sandip Halder, and Stefan De Gendt. 2025. A deep learning framework for verilog autocompletion towards design and verification automation. In *2025 IEEE 38th International System-on-Chip Conference (SOCC)*. 1–6. doi:10.1109/SOCC66126.2025.11235413

[9] Erik H D'Hollander, Ewout Danneels, Karel-Brecht Decorte, Senne Loobuyck, Arne Vanheule, Ian Van Kets, and Dirk Stroobandt. 2024. Exploring large language models for verilog hardware design generation. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 111–115. doi:10.1109/IPDPSW63119.2024.00034

[10] Weiliang Dong, Teng Zhou, and Dapeng Yan. 2022. SolChecker: A practical static analysis framework for ethereum smart contract. In *2022 International Conference on Networks, Communications and Information Technology (CNCIT)*. 179–186. doi:10.1109/CNCIT56797.2022.00036

[11] Peter Flake, Phil Moorby, Steve Golson, Arturo Salz, and Simon Davidmann. 2020. Verilog HDL and its ancestors and descendants. *Proceedings of the ACM on Programming Languages* 4, HOPL (2020), 1–90. doi:10.1145/3386337

[12] Mingzhe Gao, Jieru Zhao, Zhe Lin, Wenchao Ding, Xiaofeng Hou, Yu Feng, Chao Li, and Minyi Guo. 2024. Autovcoder: A systematic framework for automated verilog code generation using llms. In *2024 IEEE 42nd International Conference on Computer Design (ICCD)*. IEEE, 162–169. doi:10.1109/ICCD63220.2024.00033

[13] Jennifer Gillenwater, Gregory Malecha, Cherif Salama, Angela Yun Zhu, Walid Taha, Jim Grundy, and John O'leary. 2010. Synthesizable high level hardware descriptions. *New generation computing* 28, 4 (2010), 339–369. doi:10.1007/s00354-008-0093-1

[14] Emil Goh, Maoyang Xiang, I Wey, T Teo, et al. 2024. From english to asic: Hardware implementation with large language model. arXiv:2403.07039 [cs.AR] doi:10.48550/arXiv.2403.07039

[15] Google. 2024. *CodeGemma 7B (Hugging Face)*. https://huggingface.co/google/codegemma-7b

[16] Mart Graef. 2021. More Than Moore White Paper. In *2021 IEEE International Roadmap for Devices and Systems Outbriefs*. IEEE, 1–47. doi:10.1109/IRDS54852.2021.00013

[17] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. arXiv:2401.14196 [cs.SE] doi:10.48550/arXiv.2401.14196

[18] Charles Hong, Brendan Roberts, Huijae An, Alex Um, Advay Ratan, and Yakun Sophia Shao. 2025. hdl2v: A Code Translation Dataset for Enhanced LLM Verilog Generation. In *7th ACM/IEEE Symposium on Machine Learning for CAD*

(MLCAD 2025). 1–8. doi:10.1109/MLCAD65511.2025.11189055

[19] Hanxian Huang, Zhenghan Lin, Zixuan Wang, Xin Chen, Ke Ding, and Jishen Zhao. 2024. Towards llm-powered verilog rtl assistant: Self-verification and self-correction. arXiv:2406.00115 [cs.PL] doi:10.48550/arXiv.2406.00115

[20] Luciano Lavagno, Igor L Markov, Grant Martin, and Louis K Scheffer. 2017. *Electronic design automation for IC system design, verification, and testing*. CRC Press. doi:10.1201/b19569

[21] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, et al. 2023. Starcoder: may the source be with you! arXiv:2305.06161 [cs.CL] doi:10.48550/arXiv.2305.06161

[22] Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, et al. 2024. Chipnemo: Domain-adapted llms for chip design. arXiv:2311.00176 [cs.CL] doi:10.48550/arXiv.2311.00176

[23] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. In *2023 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. doi:10.1109/ICCAD57390.2023.10323812

[24] Mingjie Liu, Yun-Da Tsai, Wenfei Zhou, and Haoxing Ren. 2025. Craftrtl: High-quality synthetic data generation for verilog code models with correct-by-construction non-textual representations and targeted code repair. arXiv:2409.12993 [cs.AR] doi:10.48550/arXiv.2409.12993

[25] Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2024. Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 44, 4 (2024), 1448–1461. doi:10.1109/TCAD.2024.3483089

[26] Yi Liu, Changran Xu, Yunhao Zhou, Zeju Li, and Qiang Xu. 2025. Deeprtl: Bridging verilog understanding and generation with a unified representation model. In *The Thirteenth International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=2hcfoCHKoB

[27] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2024. Rtllm: An open-source benchmark for design rtl generation with large language model. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, IEEE Press, 722–727. doi:10.1109/ASP-DAC58780.2024.10473904

[28] Meta AI. 2024. *meta-llama/CodeLlama-70b-Instruct-hf*. https://huggingface.co/meta-llama/CodeLlama-70b-Instruct-hf

[29] Meta AI. 2024. *meta-llama/llama-models*. https://github.com/meta-llama/llama-models

[30] Mistral AI. 2024. *Au Large*. https://mistral.ai/news/mistral-large/

[31] Mistral AI. 2024. *Mistral-7B-Instruct-v0.3*. https://www.modelscope.cn/models/LLM-Research/Mistral-7B-Instruct-v0.3

[32] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. 2016. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (2016), 1591–1604. doi:10.1109/TCAD.2015.2513673

[33] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. arXiv:2203.13474 [cs.LG] doi:10.48550/arXiv.2203.13474

[34] OpenAI. 2025. *GPT-5*. https://openai.com

[35] Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. 2025. Revisiting VerilogEval: A Year of Improvements in Large-Language Models for Hardware Code Generation. *ACM Transactions on Design Automation of Electronic Systems* 30, 6, Article 91 (2025), 20 pages. doi:10.1145/3718088

[36] Khushboo Qayyum, Chandan Kumar Jha, Sallar Ahmadi-Pour, Muhammad Hassan, and Rolf Drechsler. 2025. LLM-assisted Bug Identification and Correction for Verilog HDL. *ACM Transactions on Design Automation of Electronic Systems* 30, 6, Article 101 (2025), 28 pages. doi:10.1145/3733237

[37] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2024. Code llama: Open foundation models for code. arXiv:2308.12950 [cs.CL] doi:10.48550/arXiv.2308.12950

[38] Sergio Saponara, Massimo Rovini, Luca Fanucci, Athanasios Karachalios, George Lentaris, and Dionysios Reisis. 2012. Design and comparison of FFT VLSI architectures for SoC telecom applications with different flexibility, speed and complexity trade-offs. *Circuits, Systems, and Signal Processing* 31, 2 (2012), 627–649. doi:10.1007/s00034-011-9332-7

[39] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2024. Learning Performance-Improving Code Edits. arXiv:2302.07867 [cs.SE] doi:10.48550/arXiv.2302.07867

[40] Ana-Maria Simundic. 2008. Confidence interval. *Biochemia Medica* 18, 2 (2008), 154–161. doi:10.11613/BM.2008.015

[41] Sangeetha Sudakrishnan, Janaki Madhavan, E James Whitehead Jr, and Jose Renau. 2008. Understanding bug fix patterns in verilog. In *Proceedings of the 2008 international working conference on Mining software repositories*. 39–42. doi:10.1145/1370750.1370761

[42] Shinya Takamaeda-Yamazaki. 2015. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *International Symposium on Applied Reconfigurable Computing*. Springer, 451–460. doi:10.1007/978-3-319-16214-0_42

[43] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2023. Benchmarking large language models for automated verilog rtl code generation. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6. doi:10.23919/DATE56975.2023.10137086

[44] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2024. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems* 29, 3 (2024), 1–31. doi:10.1145/3643681

[45] Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. 2024. Autochip: Automating hdl generation using llm feedback. arXiv:2311.04887 [cs.PL] doi:10.48550/arXiv.2311.04887

[46] YunDa Tsai, Mingjie Liu, and Haoxing Ren. 2024. Rtlfixer: Automatically fixing rtl syntax errors with large language model. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. San Francisco, CA, USA, Article 53, 6 pages. doi:10.1145/3649329.3657353

[47] Ning Wang, Bingkun Yao, Jie Zhou, Yuchen Hu, Xi Wang, Nan Guan, and Zhe Jiang. 2025. Insights from verification: Training a verilog generation LLM with reinforcement learning with testbench feedback. arXiv:2504.15804 [cs.AR] doi:10.48550/arXiv.2504.15804

[48] Stephen Williams and Michael Baxter. 2002. Icarus verilog: open-source verilog more than a year later. *Linux Journal* 2002, 99 (2002), 3. https://www.linuxjournal.com/article/6001

[49] Dapeng Yan, Kui Liu, Yuqing Niu, Li Li, Zhe Liu, Zhiming Liu, Jacques Klein, and Tegawendé F Bissyandé. 2022. Crex: Predicting patch correctness in automated repair of C programs through transfer learning of execution semantics. *Information and Software Technology* 152 (2022), 107043. doi:10.1016/j.infsof.2022.107043

[50] Dapeng Yan, Wenjie Yang, Zhipeng Gao, Kui Liu, Zhikuang Cai, Xiaoyuan Xie, and Zhiming Liu. 2026. Evolving Trends in Cleanliness of Open Source Projects. *ACM Transactions on Software Engineering and Methodology* (March 2026). doi:10.1145/3800680

[51] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, et al. 2025. Qwen3 technical report. arXiv:2505.09388 [cs.CL] doi:10.48550/arXiv.2505.09388

[52] John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. 2025. SWE-bench Multimodal: Do AI Systems Generalize to Visual Software Domains?. In *The Thirteenth International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=riTiq3i21b

[53] Yiyao Yang, Fu Teng, Pengju Liu, Mengnan Qi, Chenyang Lv, Ji Li, Xuhong Zhang, and Zhezhi He. 2025. Haven: Hallucination-mitigated llm for verilog code generation aligned with hdl engineers. In *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 1–7. doi:10.23919/DATE64628.2025.10993072

[54] Xufeng Yao, Haoyang Li, Tsz Ho Chan, Wenyi Xiao, Mingxuan Yuan, Yu Huang, Lei Chen, and Bei Yu. 2025. Hdldebugger: Streamlining hdl debugging with large language models. *ACM Transactions on Design Automation of Electronic Systems* 30, 6, Article 102 (2025), 26 pages. doi:10.1145/3735638

[55] PEI Zehua, Huiling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. 2024. Betterv: Controlled verilog generation with discriminative guidance. In *Proceedings of the 41st International Conference on Machine Learning*, Vol. 235. 40145–40153. https://proceedings.mlr.press/v235/pei24e.html

[56] Gaoche Zhang, Dingyang Zou, Kairui Sun, Zhihuan Chen, Meiqi Wang, and Zhongfeng Wang. 2025. GEMMV: An LLM-based Automated Performance-Aware Framework for GEMM Verilog Generation. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* (2025). doi:10.1109/JETCAS.2025.3568712

[57] Jingying Zhang and Kang Liu. 2024. JavaLLM: A Fine-Tuned LLM for Java Programming Education. In *2024 8th International Symposium on Computer Science and Intelligent Control (ISCSIC)*. IEEE, 276–280. doi:10.1109/ISCSIC64297.2024.00064

[58] Yongan Zhang, Zhongzhi Yu, Yonggan Fu, Cheng Wan, and Yingyan Celine Lin. 2024. Mg-verilog: Multi-grained dataset towards enhanced llm-assisted verilog generation. In *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 1–5. doi:10.1109/LAD62341.2024.10691738

[59] Zejun Zhang, Yixin Gan, Zhenchang Xing, Tian Zhang, Yi Li, Xiwei Xu, Qinghua Lu, and Liming Zhu. 2026. Still Manual? Automated Linter Configuration via DSL-Based LLM Compilation of Coding Standards. arXiv:2602.07783 [cs.SE] doi:10.48550/arXiv.2602.07783

[60] Zejun Zhang, Zhenchang Xing, Xiaoxue Ren, Qinghua Lu, and Xiwei Xu. 2024. Refactoring to pythonic idioms: A hybrid knowledge-driven approach leveraging large language models. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1107–1128. doi:10.1145/3643776

Received 2025-09-12; accepted 2025-12-22