OpenLLM-RTL: Open Dataset and Benchmark for LLM-Aided Design RTL Generation

Invited Paper

Shang Liu[*], Yao Lu[*], Wenji Fang[*], Mengming Li, Zhiyao Xie[†]

Hong Kong University of Science and Technology (HKUST)

{sliudx, yludf, wfang838, mengming.li}@connect.ust.hk, eezhiyao@ust.hk

ABSTRACT

The automated generation of design RTL based on large language model (LLM) and natural language instructions has demonstrated great potential in agile circuit design. However, the lack of datasets and benchmarks in the public domain prevents the development and fair evaluation of LLM solutions. This paper highlights our latest advances in open datasets and benchmarks from three perspectives: (1) RTLLM 2.0, an updated benchmark assessing LLM's capability in design RTL generation. The benchmark is augmented to 50 hand-crafted designs. Each design provides the design description, test cases, and a correct RTL code. (2) AssertEval, an open-source benchmark assessing the LLM's assertion generation capabilities for RTL verification. The benchmark includes 18 designs, each providing specification, signal definition, and correct RTL code. (3) RTLCoder-Data, an extended open-source dataset with 80K instruction-code data samples. Moreover, we propose a new verification-based method to verify the functionality correctness of training data samples. Based on this technique, we further release a dataset with 7K verified high-quality samples. These three studies are integrated into one framework, providing off-the-shelf support for the development and evaluation of LLMs for RTL code generation and verification. Finally, extensive experiments indicate that LLM performance can be boosted by enlarging the training dataset, improving data quality, and improving the training scheme.

CCS CONCEPTS

• **Hardware → Hardware description languages and compilation**; • **Computing methodologies → Natural language processing**.

KEYWORDS

LLM-assisted circuit design, electronic design automation

ACM Reference Format:

Shang Liu[*], Yao Lu[*], Wenji Fang[*], Mengming Li, Zhiyao Xie[†]. 2024. OpenLLM-RTL: Open Dataset and Benchmark for LLM-Aided Design RTL Generation: Invited Paper. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '24), October 27–31, 2024, New York, NY, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3676536.3697118

ICCAD '24, October 27–31, 2024, New York, NY, USA

ACM ISBN 979-8-4007-1077-3/24/10
https://doi.org/10.1145/3676536.3697118

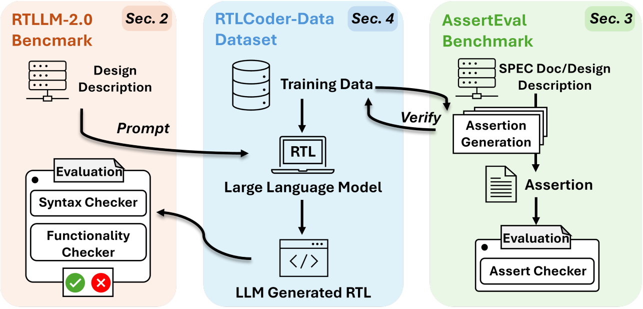


Figure 1: This paper presents open-source benchmarks and dataset for LLM-assisted RTL generation and verification.

1 INTRODUCTION

In recent years, large language models (LLMs) such as GPT [34] have demonstrated remarkable performance in natural language processing (NLP). Inspired by this progress, researchers have started exploring the adoption of LLMs in agile hardware design [8]. A promising direction that attracts the most attention is automatically generating design RTL based on natural language instructions [4, 7, 15, 25, 27–29, 31, 32, 38, 45, 46, 53, 55, 57]. In modern VLSI design flow, design teams typically exert great effort to implement precise design functionality in design RTL using hardware description languages (HDLs). Now given design functionality descriptions in natural language (e.g., specification), LLM solutions target directly generating corresponding HDL code such as Verilog, VHDL, and Chisel from scratch. This LLM-based design generation technique can potentially revolutionize the existing HDL-based VLSI design process, relieving designers from the tedious HDL coding tasks. Compared with well-explored *predictive* machine learning (ML)-based solutions in EDA [39, 51], such *generative* methods may benefit the hardware design process more directly.

In addition to the generation of RTL (i.e., HDL code) itself, the verification of RTL correctness is equally important and challenging in modern VLSI design. Functional verification ensures the RTL implementation satisfies its specification. Assertion-based verification (ABV) [50] employs assertions derived from specifications to verify the functional behavior of RTL designs. ABV can be conducted through either simulation or formal property verification (FPV), with assertions often expressed using SystemVerilog Assertions (SVAs). However, a major challenge in ABV is to obtain sufficient, high-quality assertions. Existing research on automating assertion generation includes dynamic assertion mining based on simulation traces [10, 14, 48], static generation using predefined design-specific templates [11, 35], and the direct translation of natural language

[*] Equal Contribution
[†] Corresponding Author

LLM-Assisted RTL Generation	
Prompt Engineering	[4, 7, 29, 31, 32, 46]
Closed Dataset	VerilogEval [27], BetterV [38], ChipNemo [25] Chang et al. [6], OriGen [9], CodeV [57]
Open Benchmark	RTLLM [29], VerilogEval [27], RTL-repo [2] **RTLLM 2.0 (Section 2)**
Open Dataset (code only)	Thakur et al. [45], Wang et al. [49]
Open Dataset (instruction-code)	RTLCoder [28], MG-Verilog [55], Goh et al. [15] **RTLCoder-Data (Section 4)**

LLM-Assisted RTL Verification & Debugging	
Prompt Engineering	[3, 12, 18, 20, 26, 30, 36, 42, 47, 52]
Closed Dataset	HDLdebugger [54]
Open Benchmark	**AssertEval (Section 3)**

Table 1: Existing explorations in LLM-aided design RTL generation and verification, with a focus on works that adopt or propose new datasets and benchmarks.

specifications into assertions [1, 13, 17, 20–23, 36, 37, 42, 56]. LLM solutions [3, 12, 20, 26, 30, 36, 42] turn out to be also promising in generating assertions for design RTL verification.

Many existing works directly prompt commercial LLMs like GPT-3.5/GPT-4 for RTL code generation [4, 7, 29, 31, 32, 46] or verification [3, 12, 18, 47, 52, 53], without proposing new datasets or models. However, reliance on commercial LLM tools limits in-depth research exploration and further model customization. More importantly, users of commercial LLM solutions unavoidably have data privacy concerns, since all instructions have to be uploaded to LLM providers like OpenAI. Such privacy concerns are especially critical in the IC design industry. In addition, commercial LLMs may not ensure reliable service with a low response latency.

To develop our own customized or open-source LLM solutions for RTL generation or verification, a primary challenge is the limited availability of circuit data. Unlike the huge amount of text and image resources in the public domain, circuit designs are the most important intellectual property (IP) of semiconductor companies, who typically strongly oppose sharing their designs. Such limited circuit data sharing is a long-standing issue not only for academia but also among different design teams within a single company. This data availability problem leads to a lack of datasets and benchmarks, preventing both the development and fair evaluation of LLM solutions in hardware design.

Table 1 summarizes existing efforts in LLM-assisted RTL generation and verification, with a focus on open-source datasets and benchmarks. Open benchmarks [2, 27, 29] are vitally important for a fair evaluation of LLM solutions. In addition to prompting GPT, many works tried to construct their own LLMs with either open-source [15, 28, 46, 49, 55] or closed-source [6, 9, 25, 27, 38, 57] datasets. Among the open-sourced dataset, several of them [46, 49] only provide RTL code, without alignment with the RTL generation tasks based on natural language instructions. In comparison, some open datasets [15, 28, 55] provide a pair of natural language instruction (i.e., LLM input) and code (i.e., expected LLM output) as one data sample. These datasets are better aligned with the RTL generation task and benefit the LLM fine-tuning process.

In this paper, as summarized in Table 1 and Figure 1, we highlight our latest advances in open datasets and benchmarks for LLM-assisted design and integrate them into a unified framework. It consists of three major components.

(1) In Section 2, we present an open-source **benchmark named RTLLM 2.0 for evaluating the performance of LLM-assisted RTL generation**[1]. It provides 50 RTL designs. It is an extension of our proposed benchmark RTLLM [29], which originally provided 30 designs. For each design, we provide the functionality description, test cases, and correct RTL design handcrafted by human engineers.
(2) In Section 3, we present an open-source **benchmark named AssertEval for LLM-assisted RTL verification**[2]. It provides 18 designs to evaluate the generation of assertions by LLMs. These designs cover a diverse spectrum of applications. For each design, we provide the specification document, golden RTL code, and the script for FPV.
(3) In Section 4, we present an open-source **dataset named RTLCoder-Data for training the LLM for RTL generation**[3]. This dataset provides 80K (thousand) samples, with each sample being a code generation instruction and corresponding RTL code. This is an extension of the dataset released in our proposed RTLCoder [28], which originally provided 27K samples.

In addition, a challenge in dataset generation is the difficulty in checking the correctness of data samples. RTLCoder [28] has proposed both *instruction checker* and *code checker*, evaluating the diversity introduced by new instructions and the syntax correctness of new code, respectively. However, no data generation method can automatically check whether the code has the correct functionality (i.e., same functionality as described in the instruction). In this paper, we explore an innovative method to verify training data correctness by generating assertions for each sample. In this way, we further generate and release a **verified 7K-sample dataset for training LLM for RTL generation**, which is also introduced in Section 4. Finally, we trained and compared various LLM solutions to study the factors that affect LLM performance in RTL generation.

2 RTLLM 2.0: OPEN BENCHMARK FOR RTL GENERATION

2.1 Overview of RTLLM 2.0

Our previously proposed RTLLM [29] is a comprehensive open-source benchmark for design RTL generation with natural language. It supports the evaluation of any generated HDL format, including Verilog, VHDL, and Chisel, as long as it supports logic synthesis and RTL simulation.

RTLLM [29] consists of 30 designs with a wide coverage of design complexities and scales. In RTLLM-2.0, we have expanded this collection to include 50 designs, of which the ones highlighted in **bold** in Table 2 are newly added. This enlargement allows for a more thorough evaluation of different design types and sizes, offering a more comprehensive understanding of how RTL code generation performs across a wider variety of benchmarks. By increasing the number of designs, we can now explore a broader range of scenarios,

[1] RTLLM 2.0 is in https://github.com/hkust-zhiyao/RTLLM.
[2] AssertEval is in https://github.com/hkust-zhiyao/AssertLLM.
[3] RTLCoder-Data (both 80K and 7K) is in https://github.com/hkust-zhiyao/RTL-Coder.

Arithmetic Modules	
Design	**Description**
adder_8bit	An 8-bit adder
adder_16bit	A 16-bit adder implemented with full adders
adder_32bit	A 32-bit carry-lookahead adder
adder_pipe_64bit	A 64-bit ripple carry adder based on 4-stage pipeline
adder_bcd	A BCD adder for decimal arithmetic operations
sub_64bit	A 64-bit subtractor for high-precision arithmetic
multi_8bit	An 8-bit multiplier based on shifting and adding operation
multi_16bit	A 16-bit multiplier based on shifting and adding operation
multi_booth_8bit	An 8-bit booth-4 multiplier
multi_pipie_4bit	A 4-bit unsigned number pipeline multiplier
multi_pipie_8bit	An 8-bit unsigned number pipeline multiplier
div_16bit	A 16-bit divider based on subtraction operation
radix2_div	An 8-bit radix-2 divider
comparator_3bit	A 3-bit comparator for comparing binary numbers
comparator_4bit	A 4-bit comparator for comparing binary numbers
accu	Accumulates 8-bit data and output after 4 inputs
fixed_point_adder	A fixed-point adder for arithmetic operations with fixed precision
fixed_point_substractor	A fixed-point subtractor for precise fixed-point arithmetic
float_multi	A floating-point multiplier for high-precision calculations

Control Modules	
Design	**Description**
fsm	FSM detection circuit for specific input
sequence_detector	Detect specific sequences in binary input
counter_12	Counter module counts from 0 to 12
JC_counter	A 4-bit Johnson counter with specific cyclic state sequence
ring_counter	An 8-bit ring counter for cyclic state sequences
up_down_counter	A 16-bit counter that can increment or decrement based on control signals

Memory Modules	
Design	**Description**
asyn_fifo	An asynchronous FIFO 16×8 bits
LIFObuffer	A Last-In-First-Out buffer for temporary data storage
right_shifter	Right shifter with 8-bit delay
LFSR	A Linear Feedback Shift Register for generating pseudo-random sequences
barrel_shifter	A barrel shifter for rotating bits efficiently
RAM	8x4 bits true dual-port RAM
ROM	A Read-Only Memory module for storing fixed data

Miscellaneous Modules	
Design	**Description**
clkgenerator	A clock generator for providing timing signals
instr_reg	An instruction register module for holding and processing CPU instructions
signal_generator	Generate various signal patterns
square_wave	A generator for producing square wave signals
alu	An ALU for 32bit MIPS-ISA CPU
pe	A Multiplying Accumulator for 32bit integer
freq_div	Frequency divider for 100M input clock, outputs 50MHz, 10MHz, 1MHz
freq_divbyeven	Frequency divider that divides input frequency by even numbers
freq_divbyodd	Frequency divider that divides input frequency by odd numbers
freq_divbyfrac	Frequency divider that divides input frequency by fractional values
calendar	Perpetual calendar with seconds, minutes, and hours
traffic_light	Traffic light system with three colors and pedestrian button
width_8to16	First 8-bit data placed in higher 8-bits of the 16-bit output
synchronizer	Multi-bit mux synchronizer
edge_detect	Detect rising and falling edges of changing 1-bit signal
pulse_detect	Extract pulse signal from the fast clock and create a new one in the slow clock
parallel2serial	Convert 4 input bits to 1 output bit
serial2parallel	1-bit serial input and output data after receiving 6 inputs

Table 2: RTLLM-2.0 benchmark description. The benchmark includes 50 designs across various applications, with bold designs representing newly added designs relative to RTLLM.

better understand design intricacies, and assess performance more effectively in the context of RTL code generation.

Unlike RTLLM, which broadly classifies designs into *Arithmetic* and *Logic* types, RTLLM-2.0 takes a closer look by categorizing designs based on their specific functions and applications. By breaking down designs in RTLLM-2.0 according to their unique purposes, these detailed categories allow for better comparisons of how different models perform across various design types.

2.2 Detailed Inspection of the Benchmark

The benchmark RTLLM-2.0 dataset is meticulously categorized into four primary module classes: Arithmetic Modules, Memory Modules, Control Modules, and Miscellaneous Modules. Each class encompasses a variety of functional units pertinent to diverse computational and control tasks, as delineated in Table 2. This structured classification facilitates a comprehensive analysis and application of the dataset across multiple domains in digital system design.

Arithmetic Modules comprise various adders, subtractors, multipliers, dividers, comparators, accumulators, and other specialized units like fixed-point arithmetic components. For instance, the adder subcategory includes 8-bit, 16-bit, 32-bit, and 64-bit pipelined adders, along with a BCD adder, addressing both general and specific arithmetic operations. Similarly, subtractors, multipliers, and comparators are available in different bit widths and configurations, such as a 64-bit subtractor, an 8-bit multiplier, and both 3-bit and 4-bit comparators, respectively.

Memory Modules are designed to handle data storage and retrieval with FIFO (First-In, First-Out) and LIFO (Last-In, First-Out) buffers, alongside various shifters including right shifters, LFSRs (Linear Feedback Shift Registers), barrel shifters, as well as RAM and ROM. The inclusion of asynchronous FIFO and LIFO buffers highlights the benchmark's capability to manage different memory access patterns efficiently.

Design Type	Cryptographic Unit			Processor Core			Arithmetic Unit			Communication Protocol			Memory Controller		
Design Name/ # Page/ # Signal	AES	15	11	amber	26	14	ecg	9	12	ethernet	42	54	hpdmc	8	4
	sha3	17	9	lxp32	59	22	mac	24	34	i2c	15	24	sdc	26	53
	tiny_aes	17	4	minsoc	22	14	pairing	13	8	sockit	29	15	sdr_ctrl	28	46
							tiny_pairing	17	10	uart	10	11			

Table 3: AssertEval benchmark description. The benchmark includes 18 open-source designs across various applications. For each design's specification document, we list the number of file pages and the number of architectural signals under verification.

Control Modules focus on state management and counting mechanisms, featuring finite state machines (FSMs), sequence detectors, and various counters such as a 12-bit counter, Johnson counter (JC_counter), ring counter, and an up/down counter. These modules are crucial for controlling the flow of operations and ensuring sequential logic execution within digital systems.

Miscellaneous Modules cover a broad spectrum of functionalities including signal generation, RISC-V components, frequency dividers, and other essential units. Signal generation features modules like a signal generator and a square wave generator. The RISC-V category includes clock generators, instruction registers, ALU, and processing elements, essential for constructing RISC-V-based architectures. Frequency dividers are detailed with modules that divide by even, odd, and fractional values. Additionally, there are modules for specific applications such as calendars, traffic lights, data width converters, synchronizers, and various signal detection and conversion units.

This classification framework facilitates a nuanced understanding of the benchmark, highlighting its versatility and applicability across different domains of digital system design and analysis.

2.3 Benchmark Evaluation

2.3.1 Overview of Test Files. For each design, RTLLM-2.0 provides the following information in three separate files.

Description (*design_description.txt*): A natural language description of the target design's functionality, serving as a prompt for LLMs to generate RTL code. It includes the module name and all input/output (I/O) signals with names and widths, enabling automatic functionality verification with the provided testbench.

Testbench (*testbench.v*): A testbench containing multiple test cases with input and expected output values. It corresponds to the module name and I/O signals in *design_description.txt* and is used to verify design functionality.

Correct Design (*designer_RTL.v*): A reference design Verilog hand-crafted by human designers. By comparing with this reference design, we can quantitatively evaluate the design qualities of the automatically generated design.

2.3.2 Evaluation Metrics. To systematically evaluate the generated design RTL, we summarize three progressive goals, which can all be evaluated with our benchmark. The first and basic goal is the **syntax goal**. It means the syntax of the generated RTL design should at least be correct. It can be verified by checking whether the design can be correctly synthesized into netlist by synthesis tools [43]. The second is **functionality goal**. It requires the generated RTL design to function as expected, verified by passing all test cases provided in *testbench.v*. While the testbench samples a reasonable number of cases, passing them doesn't guarantee 100% functionality correctness. If the design is correct in both syntax and functionality, it is considered successful. However, for practical use, its design qualities, including performance, power, and area (PPA), should also be desirable. This is the **quality goal**, verified by measuring PPA values after synthesis and layout.

3 ASSEREVAL: OPEN FRAMEWORK AND BENCHMARK FOR RTL VERIFICATION

3.1 Assertions Generation and Evaluation Framework

Inspired by the potential of LLMs, translating natural language specifications into assertions has gained significant attention. Some works[20, 30] leverage LLMs to convert human-extracted or human-written specification sentences into corresponding assertions. Other approaches, like AssertLLM [12], process entire specification documents directly, using LLMs to automatically extract assertion-related information from highly unstructured, multi-modal data, including descriptive text and waveform diagrams.

Despite the growing interest in LLM-based assertion generation, a universal evaluation method and benchmark are still unavailable. To address this challenge, we propose AssertEval, a benchmark and framework designed to evaluate the quality of LLM-based assertion generation across various VLSI designs.

The generation and evaluation flow is demonstrated in Figure 2. For the generation process, we provide entire specification documents as input, users then generate assertions for each architectural signal with their own assertion generation methods. These assertions are then evaluated against our provided golden RTL implementations using formal property verification techniques.

After the assertion generation process, the framework automatically evaluates the quality of the generated assertions. Bug-free golden RTL implementations are provided for this evaluation. Based on these golden RTL designs, the generated assertions are verified through formal property verification (FPV) techniques. After performing FPV, the following metrics are computed to evaluate the quality of generated assertions:

- **Syntax**: checks if generated assertions have syntax errors.
- **FPV pass/fail**: when RTL designs are bug-free, an assertion that passes the FPV check is considered semantically correct, and conversely, a failure indicates an incorrect assertion.
- **COI coverage**: cone of influence (COI) coverage measures the percentage of design logic that is structurally connected to the properties. It is a common metric to evaluate the quality and usefulness of the generated properties.

3.2 AssertEval Benchmark Description

The benchmark AssertEval consists of 18 open-source designs that cover a diverse array of applications, including cryptographic units, processor cores, arithmetic units, communication protocols, and memory controllers. Considering the capability of existing LLM-based generation methods, we have collected specification documents that are fewer than 60 pages and contain fewer than 60

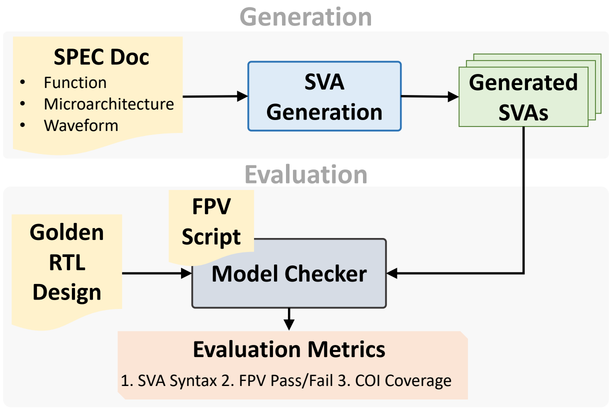


Figure 2: Evaluation of generated assertions using our benchmark. We provide natural language specification documents as input for the assertion generation process. The generated assertions are then evaluated against the provided golden RTL designs using the FPV technique. Three key metrics are employed to assess the quality of the generated assertions.

architecture-level signals. We list the detailed statistics for each design in Table 3. Additionally, we provide an FPV script for Cadence JasperGold [5], which can be executed with a single button click for ease of use. For each design within the benchmark, our benchmark provides the following three files:

- **Specification document**: This file contains the natural language specification for the design, providing a detailed description of the design architecture and functionality.
- **Golden RTL implementation**: This file comprises the RTL design implementations that are strictly implemented according to the specification. The designs are verified to ensure it is free from bugs, serving as a reliable standard for evaluating the correctness of generated assertions.
- **FPV script**: This script automatically executes FPV, allowing users to execute the verification with a single click.

The specification document is highly unstructured, with assertion-related information dispersed across various sections. Additionally, it includes multi-modal data (e.g., descriptive text and waveform diagrams), making the extraction of relevant details challenging.

Our provided specification typically includes seven key sections: 1) Summary: outlines the design's concepts and features; 2) IO ports: provides detailed information for the interface; 3) Registers: describes all the architecture-level registers in the design; 4) Operation: explains the operational procedures for dataflow and control; 5) Architecture: the high-level workflow and dataflow of the design; and 6) Usage examples: offers basic usage scenarios for the design. For signals, the specification may only define critical architecture-level IO ports and registers, leaving the designers to detail internal signals for RTL implementations. 7) Waveform diagram: describe behaviors for different signals.

4 OPEN DATASET FOR RTL GENERATION

In this Section, we present an open-source dataset named RTLCoder-Data for training the LLM for RTL generation. It provides a large 'raw dataset' with 80K samples, tripling the one previously released in RTLCoder [28]. Moreover, we propose an innovative verification-based method to check the functionality correctness of each instruction-code data sample. Applying both the functionality checker and syntax checker, we further generate and release a high-quality verified dataset with 7K 'mostly-correct' samples.

4.1 Basic Dataset Generation Flow

Our prior work RTLCoder [28] has proposed an automated training dataset generation flow and generated 27K training samples, with each sample being a pair of design instruction (i.e., model input) and the reference RTL code (i.e., expected model output). The instruction can be viewed as the input question for LLMs, describing the desired circuit functionality in natural language. The reference code is the expected Verilog code that implements the functionality. This flow takes advantage of the powerful general text generation ability of the commercial tool GPT with several prompt templates. As Figure 3 shows, the flow includes three stages, which are summarized below.

Stage 1: Keywords Preparation. The first stage of the data generation flow targets preparing RTL domain keywords for subsequent stages. At process ❶ in Figure 3 shows, GPT is requested to generate keywords related to digital IC design (i.e., commonly used logic components) based on a set of prompts. We obtain a keyword pool $\mathcal{L}_{key}$ with hundreds of digital design keywords.

Stage 2: Instruction Generation. The second stage targets generating sufficient instructions based on the initial keywords and Verilog source code. At process ❷, existing keywords are extended from $\mathcal{L}_{key}$ to complete instructions. In addition to keyword-based instruction generation, we also generate instructions based on existing source code collected by us, as shown in process ❸. By providing GPT with either part or a complete Verilog code $\mathcal{L}_{code}$ collected by [45], we inspire it to create a related Verilog design problem.

Process ❷ and ❸ help generate the initial design instruction pool $\mathcal{L}_{ins}$. After that, we iteratively augment this pool with mutation. Process ❹ applies two types of mutation operations on instructions sampled from the design instruction library $\mathcal{L}_{ins}$. The process ❺ would check every new design instruction using a set of rules and only passed valid instructions are added to $\mathcal{L}_{ins}$.

Stage 3: Reference Code Generation. The third stage targets generating the reference code corresponding to each instruction. As shown in ❻, we feed each instruction from $\mathcal{L}_{ins}$ into GPT, generating corresponding reference design code. Then depending on whether process ❼ is applied, we generate two types of datasets in this flow, as we will introduce in Section 4.2 and 4.3, separately.

4.2 80K Raw Dataset in RTLCoder-Data

To collect a large dataset, we continued to execute the basic data generation flow. Compared with the previous dataset from [28], we further enlarge the source code pool in $\mathcal{L}_{code}$ in process ❸ and continue the mutation in process ❹ during the generation process. In addition, we slightly relaxed the instruction checking conditions in process ❺. Previously in [28], each new instruction is compared with all existing instructions in $\mathcal{L}_{ins}$ to check whether it introduces diversity. However, it takes a long time to compare each new instruction with all existing instructions. We removed this time-consuming diversity checking process, and only checked the basic instruction content in process ❺. As we will introduce in Section 5.2, results demonstrate that removing such diversity checking does not impair the overall diversity in the ultimate dataset. Finally, we accumulate and release a dataset of 80K samples, tripling the previous dataset in RTLCoder [28].

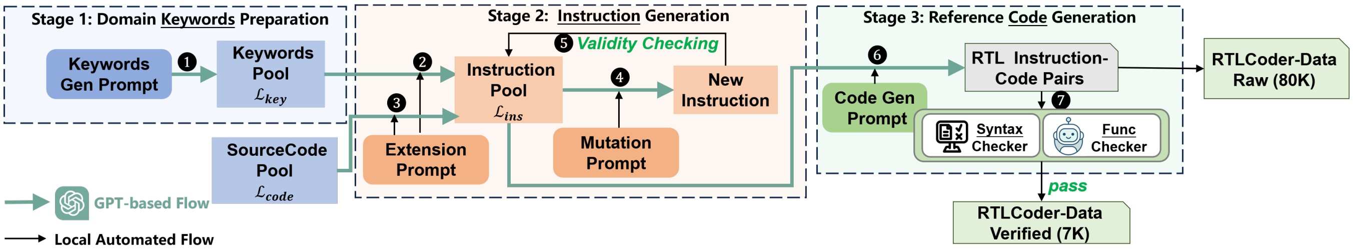


Figure 3: The automated training dataset generation flow to generate RTLCoder-Data. The framework is based on prior RTLCoder [28], but we proposed an innovative automated functionality checking method in Stage 3.

However, since the overall generation process of this 80K dataset relies on prompting commercial LLMs, we cannot guarantee the correctness of all samples. Therefore, we also refer it as a ‘raw’ 80K dataset in RTLCoder-Data. To evaluate the effectiveness this ‘raw’ dataset, we have trained LLMs with different numbers of data samples and evaluated results will be introduced in Section 5.3. Results indicate that a larger dataset leads to better model performance, and the performance is not saturated when 80K samples are used for training. Despite possible incorrectness in data samples, a larger dataset still clearly boosts model performance and proves useful.

4.3 7K Verified Dataset in RTLCoder-Data

As introduced in Section 4.2, we have accumulated a raw dataset with 80K samples, but it is difficult to verify the correctness of each data sample. Specifically, it is feasible to automatically check the syntax correctness of the code in each sample with tools like VCS [44] or iVerilog, but it is very challenging to check whether the code has the correct functionality (i.e., code functionality matches the description in instruction). This functionality checking task is exactly hardware verification, which has been studied for decades, relies on human engineers, and is difficult to get guaranteed results. To the best of our knowledge, there is no prior work on automatic examination of code functionality correctness in dataset generation.

In this work, we made an innovative exploration to enable the automatic functionality checking of each instruction-code data sample. It is shown as the *functionality checker* in process ❼ of Figure 3. The solution is based on the LLM-assisted verification method introduced in Section 3. First, based on the functionality description from instruction, we prompt commercial LLMs to generate corresponding assertions. The prompt techniques are from LLM-assisted verification works such as AssertLLM [12]. Second, we combine the code and generated assertions, and feed them to verification platforms (e.g., JasperGold [5]) to check whether the code violates any assertions. If all assertions are passed, it is likely that the code correctly implements the functionality described in the instruction. Still, this is not 100% guarantee of sample correctness, but this process is fully automated and it leads to sufficiently high-quality samples for model training.

A problem in this verification-based functionality checking is, the assertions for verification are also generated by LLMs [12], thus the correctness of assertions is not guaranteed either. As a result, incorrect assertions may be generated for actually correct samples, making the correct samples fail the verification process. Therefore, this functionality checking method is conservative: Samples passing all assertions are likely to be correct, but correct data samples may fail the checking due to wrong assertions. Applying both functionality checking and syntax checking, as indicated by ❼, we collected 7K high-quality verified samples. As we will introduce in Section 5.3, the 7K verified dataset leads to better LLM performance compared with models trained with even 50K raw data.

5 RTL GENERATION EXPERIMENT RESULTS

In this Section, we train and evaluate various LLM solutions with our 80K raw dataset and 7K verified datasets from RTLCoder-Data. In addition to extensive comparisons with various other LLM solutions, we studied the impact of training data amount, training scheme, and training data quality on LLM performance.

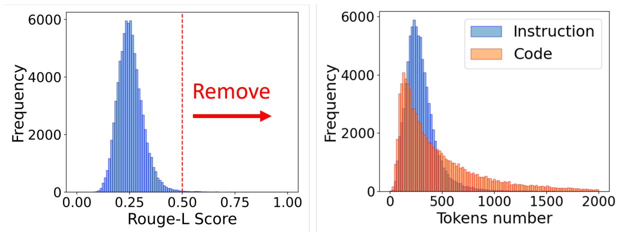
5.1 LLM Training and Evaluation Setup

To evaluate the performance of LLM-assisted RTL generation, we adopt two representative benchmarks named VerilogEval [27] and RTLLM [29]. For RTLLM, following the original benchmark [29], each task is counted as success as long as *any* of 5 trials passes the test. This can be interpreted as pass@5 metric. For all tested models, we evaluate all 3 *temperature* conditions $\{0.2, 0.5, 0.8\}$ and report the best performance for each model.

We choose the Mistral-7B-v0.1 [19] and DeepSeek-Coder-6.7b-Instruct [16] as the basic pre-trained model for finetuning. In all experiments, we opted for the Adam optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and learning rate γ = 1e-5, while abstaining from the use of weight decay. Concurrently, we established a context length of 2048 and a global batch size of 256. We trained the model on only 4 consumer-level RTX 4090 GPUs (24GB each), each of which could afford 2×2048 context length using DeepSpeed stage-2 [40].

5.2 Evaluation of Dataset

To prevent information leakage, for each instruction-code concatenated sample in the training dataset, we computed its maximum similarity with all test cases in the benchmarks. We employed the



(a) (b)

Figure 4: Training dataset analysis for the obtained 80K dataset. (a) Similarity measurement between training dataset and two benchmarks based on Rouge-L metric. (b) Tokens number distribution of instruction and code part.

	RTLCoder-Data Raw (80K)	RTLCoder-Data Verified (7K)	MG-Verilog [55]	Goh et al. [15]
CR	4.21	4.32	5.80	5.27
CR: POS	7.33	7.45	9.16	10.1

Table 4: Diversity scores (CR, CR:POS) of RTLCoder-Data Raw (80K), RTLCoder-Data Verified (7K), and other RTL datasets [15, 55]. Lower CR and CR:POS mean higher dataset diversity. Both datasets from RTLCoder-Data exhibit satisfactory diversity compared with others.

Rouge-L[4], a widely used similarity calculation metric in the LLM domain. As Figure 4 (a) shows, most training samples have a low Rouge-L Value of around 0.25 and this indicates a low semantic overlap with the benchmarks. There are a small number of samples with higher similarity, and we get rid of these samples with Rouge-L > 0.5 during training. In addition, Figure 4 (b) shows that an instruction-code sample is generally within 2048 token length. So we can set 2048 as the max length in our finetuning.

To check the diversity of our proposed training dataset RTLCoder-Data-Raw (80K) and RTLCoder-Data-Verified (7K), we utilized two diversity measures: Compression Ratios (CR) and Part-of-Speech Compression Ratio (CR: POS) which are suggested best lexical diversity metrics by [41]. CR is calculated utilizing text compression algorithms which can identify redundancy in the whole contents. The CR-POS can capture the repeated syntactic redundancy by compressing the part-of-speech (POS) tag sequences of the original text. We also followed the method utilized in [41] to extract the tag sequences of the dataset. The results are illustrated in Table 4. Our proposed two datasets have lower CR and CR:POS than other existing open-source Verilog instruction-code datasets. This indicates that RTLCoder-Data-Raw (80K) and RTLCoder-Data-Verified (7K) have a satisfactory diversity.

5.3 Result of Trained LLM in RTL Generation

Table 5 summarizes the comparison of various LLM-assisted RTL generation solutions, including commercial models GPT3.5/GPT4, both closed- and open-source LLMs customized for Verilog generation [27, 38, 45], general software code generators [19, 24, 33], and our fine-tuned models based on DeepSeek-Coder-6.7b-Instruct [16] and Mistral-7B-v0.1 [19] with different amount of training data and training schemes. Relevant results are also presented in Figure 5, which shows LLM performance versus the amount of training data.

Overall performance based on RTLCoder-Data. We train the base model directly on the RTLCoder-Data Raw (80K) through instruction-supervised fine-tuning which is referred to as "basic direct training" in Table 5. We can observe that DeepSeek-Direct (80K data samples) outperforms all other baseline models in Eval-Machine and is only inferior to GPT-4 in Eval-Human and RTLLM V1.1. Specifically, in the Eval-Machine part, it even outperforms GPT4 by an absolute value of 4.7% in the pass@1 metric. In summary, DeepSeek-Direct (80K data samples) outperforms GPT-3.5 and all non-commercial baselines in all metrics. It is surprising that the lightweight model with only 7 billion parameters could achieve such impressive accuracy despite its smaller size.

Impact of training data amount. To further investigate the impact of dataset size on model performance, we sampled subsets of 5K, 27K, and 50K samples from the RTLCoder-Data Raw (80K) and then conducted direct finetuning on these subsets. The results are shown in Table 5 and also plotted in Figure 5. We can observe that as the training data volume increases, the overall performance of the model on the benchmarks also improves. For instance, as the training data size increases from 5K to 80K, the model's performance on the Eval-Machine pass@1 metric rises from 53.7% to 64.7%. Additionally, as illustrated in Figure 5, even with 80K data samples, there are still no signs of model performance saturation. This indicates that enlarging the training dataset size can significantly boost the model's code generation capabilities.

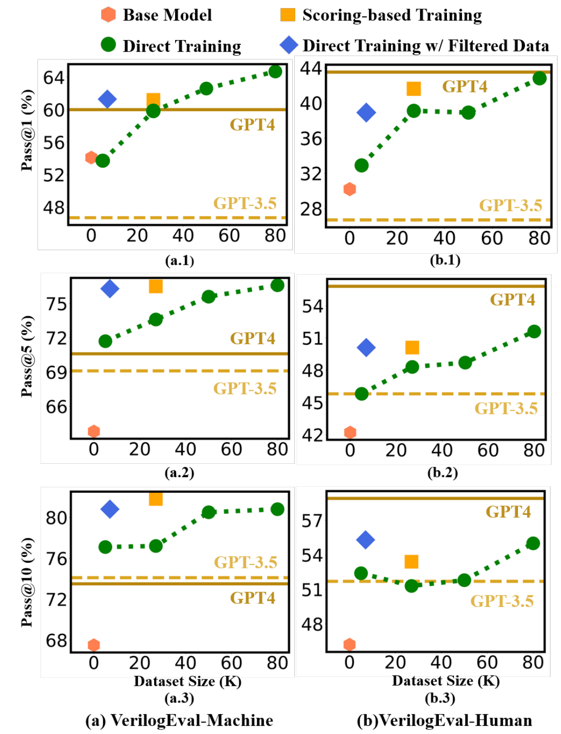


Figure 5: The pass@k performance on VerilogEval benchmarks versus the amount of training data from RTLCoder-Data. The performance improves as the data size increases.

Impact of training scheme. We extracted a 27K subset from the RTLCoder-Data Raw (80K) and employed the code quality feedback-based training scheme proposed in RTLCoder [28] to obtain models named Mistral-Scoring and DeepSeek-Scoring. Their performance is presented in Table 5 and Figure 5 titled 'Scoring-based Training'. Compared with DeepSeek-Direct (27K data samples) and Mistral-Direct (27K data samples), models trained with the scoring-based scheme are better on all benchmarks, indicating that this better scoring-based training method [28] improves model performance.

Impact of training data quality. In the 'Verified Dataset' part of Table 5, we directly trained the DeepSeek-Coder using the dataset RTLCoder-Data Verified (7K). DeepSeek-Direct (7K verified) outperforms DeepSeek-Direct (27K) across all benchmarks and even surpasses DeepSeek-Direct (50K) on 6/8 metrics. Moreover, DeepSeek-Direct (7K verified) only uses < 20% of the training time of DeepSeek-Direct (50K). This demonstrates that enhancing the quality of the training dataset can improve the model performance and reduce the LLM training cost. It indicates the great potential of our proposed assertion-based functionality checking technique.

[4] The Rouge-L score $\in [0, 1]$, with values closer to 1 indicating higher similarity between the two sequences.

Model Type	Evaluated Model	Num of Params	VerilogEval Benchmark [27] (using pass@k metric)						RTLLM V1.1 [29] (using pass@5 metric)	
			Eval-Machine (%)			Eval-Human (%)			Syntax-VCS(%)	Func (%)
			k=1	k=5	k=10	k=1	k=5	k=10		
Closed-Source Baseline	GPT-3.5	N/A	46.7	69.1	74.1	26.7	45.8	51.7	89.7	37.9
	GPT4	N/A	60.0	70.6	73.5	43.5	55.8	58.9	100	65.5
	ChipNeMo [25]	13B	43.4	N/A	N/A	22.4	N/A	N/A	N/A	N/A
	VerilogEval [27]	16B	46.2	67.3	73.7	28.8	45.9	52.3	N/A	N/A
	BetterV [38]	7B	64.2	75.4	79.1	40.9	50.0	53.3	N/A	N/A
Open-Source Baseline	Codegen2 [33]	16B	5.00	9.00	13.9	0.90	4.10	7.25	72.4	6.90
	Starcoder [24]	15B	46.8	54.5	59.6	18.1	26.1	30.4	93.1	27.6
	Thakur et al. [45]	16B	44.0	52.6	59.2	30.3	43.9	49.6	86.2	24.1
	Mistral-7B [19]	7B	36.9	48.8	57.4	4.49	12.6	18.6	72.4	20.7
	DeepSeek-Coder [16]	6.7B	54.1	63.8	67.5	30.2	42.2	46.2	89.6	34.5
Scoring-based Training [28]	Mistral-Scoring (27K data samples)	7B	62.5	72.2	76.6	36.7	45.5	49.2	96.6	48.3
	DeepSeek-Scoring (27K data samples)	6.7B	61.2	76.5	81.8	41.6	50.1	53.4	93.1	48.3
Basic Direct Training	Mistral-Direct (27K data samples)	7B	58.9	70.0	74.1	34.4	42.3	45.1	89.7	41.4
	DeepSeek-Direct (5K data samples)	6.7B	53.7	71.7	77.1	32.9	45.8	52.4	93.1	41.4
	DeepSeek-Direct (27K data samples)	6.7B	59.8	73.6	77.2	39.1	48.3	51.3	86.2	44.8
	DeepSeek-Direct (50K data samples)	6.7B	62.6	75.6	80.5	38.9	48.7	51.8	89.7	55.2
	DeepSeek-Direct (80K data samples)	6.7B	64.7	76.6	80.8	42.8	51.6	55.0	93.1	48.3
Verified Dataset	DeepSeek-Direct (7K verified data samples)	7B	61.3	76.3	80.8	38.9	50.1	55.3	100	48.3

Table 5: Performance comparison of RTL code generators on VerilogEval Benchmark [27] and RTLLM Benchmark [29]. The top scores ranked 1st, 2nd, and 3rd in each column are marked in Green , Blue , and Red , respectively.

6 LIMITATION AND CHALLENGES

Finally, we would like to discuss some challenges and questions we encountered during the development of the dataset or benchmark for LLM-assisted design automation solutions, and share our thoughts about these questions.

When building the open-source benchmark for RTL generation, we encountered several challenges:

(1) *Shall we include more complex designs in the benchmark?* Due to the limited abilities of existing LLMs, almost all LLMs encounter difficulty in generating 'correct' RTL design code for very complex designs. As a result, overly complex designs often fail to differentiate the capabilities of the models. In addition, it is difficult to precisely describe complex designs with natural languages.

(2) *How detailed should the description be?* When descriptions are overly vague or general, LLMs struggle to produce designs that meet expected functionality, making it difficult to assess model capabilities. Conversely, if descriptions are too detailed, focusing on intricate RTL circuit specifics, the RTL generation effectively becomes a form of 'code translation', which also fails to demonstrate the general generative abilities of LLMs. Therefore, the level of detail in the description for benchmarking requires careful consideration.

(3) *How to alleviate the influence of training data leakage on the benchmark scores?* The overlap between the training dataset and benchmarks should always be carefully examined because an overfitted LLM cannot generalize well in practice. Overfitted LLM can easily lead to unfair comparisons and misleading conclusions. However, the text similarity approximation we used based on Rouge-L metric may not be perfect. In addition, leakage during the LLM pre-training process is difficult to control. How to define and evaluate the data leakage in RTL generation is still a challenging open problem.

The main challenge in LLM-based assertion generation centers around improving the quality of the generated assertions. We break down this challenge into two key questions:

(1) *How to better quantify the assertion quality?* Existing metrics like syntax/semantics correctness and COI coverage are useful but inadequate for complex verification scenarios, such as capturing state transitions or ensuring different assertions cover distinct properties. More precise evaluation techniques are worth exploring in future works.

(2) *What limits the generation of high-quality assertions?* High-quality assertions depend not only on LLM capabilities but also on the richness of the specification documents. Specifications that lack detailed functionalities or connectivities will limit the effectiveness of assertion generation, regardless of the capability of LLM.

7 CONCLUSION

In this work, we present our latest advances in open-source benchmarks and datasets for developing LLMs to assist in design RTL generation and verification. We fully open-sourced 1) RTLLM 2.0, an updated benchmark for the evaluation of LLM-assisted RTL generation; 2) AssertEval, a benchmark or the evaluation of LLM-assisted assertion generation for verification; and 3) RTLCoder-Data, an extended open-source dataset for training LLMs for RTL generation. It provides 80K instruction-code data samples, as well as a 7K verified high-quality dataset. These open-source circuit data are provided as off-the-shelf resources, targeting more democratized and reproducible AI for EDA research.

ACKNOWLEDGMENTS

This work is supported by Hong Kong Research Grants Council (RGC) ECS Grant 26208723, National Natural Science Foundation of China 92364102, and ACCESS – AI Chip Center for Emerging Smart Systems, sponsored by InnoHK funding, Hong Kong SAR.

REFERENCES

[1] Fnu Aditi and Michael S Hsiao. 2022. Hybrid Rule-based and Machine Learning System for Assertion Generation from Natural Language Specifications. In *Asian Test Symposium (ATS)*.

[2] Ahmed Allam and Mohamed Shalan. 2024. RTL-Repo: A Benchmark for Evaluating LLMs on Large-Scale RTL Design Projects. *arXiv preprint arXiv:2405.17378* (2024).

[3] Jitendra Bhandari, Johann Knechtel, Ramesh Narayanaswamy, Siddharth Garg, and Ramesh Karri. 2024. LLM-Aided Testbench Generation and Bug Detection for Finite-State Machines. *arXiv preprint arXiv:2406.17132* (2024).

[4] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. 2023. Chip-Chat: Challenges and Opportunities in Conversational Hardware Design. *arXiv preprint arXiv:2305.13243* (2023).

[5] Cadence. 2023. Jasper Formal Verification Platform . https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html.

[6] 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. *arXiv preprint arXiv:2403.11202* (2024).

[7] Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. 2023. ChipGPT: How far are we from natural language hardware design. *arXiv preprint arXiv:2305.14019* (2023).

[8] Lei Chen, Yiqi Chen, Zhufei Chu, Wenji Fang, Tsung-Yi Ho, et al. 2024. The dawn of AI-native EDA: Promises and challenges of large circuit models. *arXiv preprint arXiv:2403.07257* (2024).

[9] Fan Cui, Chenyang Yin, et al. 2024. OriGen: Enhancing RTL Code Generation with Code-to-Code Augmentation and Self-Reflection. *arXiv preprint arXiv:2407.16237* (2024).

[10] Alessandro Danese, Nicolò Dalla Riva, and Graziano Pravadelli. 2017. A-team: Automatic template-based assertion miner. In *DAC*.

[11] Wenji Fang, Guangyu Hu, and Hongce Zhang. 2023. r-map: Relating Implementation and Specification in Hardware Refinement Checking. *IEEE TCAD* (2023).

[12] Wenji Fang, Mengming Li, Min Li, Zhiyuan Yan, Shang Liu, Hongce Zhang, and Zhiyao Xie. 2024. AssertLLM: Generating and Evaluating Hardware Verification Assertions from Design Specifications via Multi-LLMs. *arXiv preprint arXiv:2402.00386* (2024).

[13] Steven J Frederiksen, John Aromando, and Michael S Hsiao. 2020. Automated Assertion Generation from Natural Language Specifications. In *ITC*.

[14] Samuele Germiniani and Graziano Pravadelli. 2022. Harm: a hint-based assertion miner. *IEEE TCAD* (2022).

[15] Emil Goh, Maoyang Xiang, I Wey, T Hui Teo, et al. 2024. From English to ASIC: Hardware Implementation with Large Language Model. *arXiv preprint arXiv:2403.07039* (2024).

[16] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[17] Christopher B Harris and Ian G Harris. 2016. Glast: Learning formal grammars to translate natural language specifications into hardware assertions. In *DATE*.

[18] 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 preprint arXiv:2406.00115* (2024).

[19] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).

[20] Rahul Kande, Hammond Pearce, et al. 2024. (Security) Assertions by Large Language Models. *IEEE Transactions on Information Forensics and Security (TIFS)* (2024).

[21] Oliver Keszocze and Ian G Harris. 2019. Chatbot-based assertion generation from natural language specifications. In *Forum for Specification and Design Languages*.

[22] Rahul Krishnamurthy and Michael S Hsiao. 2019. Controlled natural language framework for generating assertions from hardware specifications. In *ICSC*.

[23] Rahul Krishnamurthy and Michael S Hsiao. 2019. Ease: Enabling hardware assertion synthesis from english. In *Rules and Reasoning: Third International Joint Conference*.

[24] Raymond Li, Loubna Ben Allal, Yangtian Zi, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[25] Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, et al. 2023. ChipNeMo: Domain-Adapted LLMs for Chip Design. *arXiv preprint arXiv:2311.00176* (2023).

[26] Mingjie Liu, Minwoo Kang, Ghaith Bany Hamad, Syed Suhaib, and Haoxing Ren. 2024. Domain-Adapted LLMs for VLSI Design and Verification: A Case Study on Formal Verification. In *2024 IEEE 42nd VLSI Test Symposium (VTS)*. IEEE, 1–4.

[27] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. *arXiv preprint arXiv:2309.07544* (2023).

[28] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2023. RTLCoder: Outperforming GPT-3.5 in Design RTL Generation with Our Open-Source Dataset and Lightweight Solution. *arXiv preprint arXiv:2312.08617* (2023).

[29] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2023. RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model. *arXiv preprint arXiv:2308.05345* (2023).

[30] Bhabesh Mali, Karthik Maddala, Sweeya Reddy, Vatsal Gupta, Chandan Karfa, and Ramesh Karri. 2024. ChIRAAG: ChatGPT Informed Rapid and Automated Assertion Generation. *arXiv preprint arXiv:2402.00093* (2024).

[31] Madhav Nair, Rajat Sadhukhan, et al. 2023. Generating secure hardware using chatgpt resistant to cwes. *Cryptology ePrint Archive* (2023).

[32] Andre Nakkab, Sai Qian Zhang, Ramesh Karri, and Siddharth Garg. 2024. Rome was Not Built in a Single Step: Hierarchical Prompting for LLM-based Chip Design. *arXiv preprint arXiv:2407.18276* (2024).

[33] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309* (2023).

[34] OpenAI. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).

[35] Marcelo Orenes-Vera, Aninda Manocha, et al. 2021. AutoSVA: Democratizing Formal Verification of RTL Module Interactions. In *DAC*.

[36] Marcelo Orenes-Vera, Margaret Martonosi, and David Wentzlaff. 2023. Using LLMs to Facilitate Formal Verification of RTL. *arXiv e-prints* (2023), arXiv–2309.

[37] Ganapathy Parthasarathy, Saurav Nanda, Parivesh Choudhary, and Pawan Patil. [n. d.]. SpecToSVA: Circuit Specification Document to SystemVerilog Assertion Translation. In *2021 Second Document Intelligence Workshop at KDD*.

[38] Zehua Pei, Hui-Ling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. 2024. BetterV: Controlled Verilog Generation with Discriminative Guidance. *arXiv preprint arXiv:2402.03375* (2024).

[39] Martin Rapp, Hussam Amrouch, Yibo Lin, Bei Yu, David Z Pan, Marilyn Wolf, and Jörg Henkel. 2021. MLCAD: A survey of research in machine learning for CAD keynote paper. *IEEE TCAD* (2021).

[40] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *KDD*.

[41] Chantal Shaib, Joe Barrow, Jiuding Sun, Alexa F Siu, Byron C Wallace, and Ani Nenkova. 2024. Standardizing the measurement of text diversity: A tool and a comparative analysis of scores. *arXiv preprint arXiv:2403.00553* (2024).

[42] Chuyue Sun, Christopher Hahn, and Caroline Trippel. 2023. Towards Improving Verification Productivity with Circuit-Aware Translation of Natural Language to SystemVerilog Assertions. In *International Workshop on Deep Learning-aided Verification*.

[43] Synopsys. 2023. Design Compiler® RTL Synthesis. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-nxt.html.

[44] Synopsys. 2023. VCS® functional verification solution. https://www.synopsys.com/verification/simulation/vcs.html.

[45] 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 *DATE*.

[46] Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. 2023. AutoChip: Automating HDL Generation Using LLM Feedback. *arXiv preprint arXiv:2311.04887* (2023).

[47] YunDa Tsai, Mingjie Liu, and Haoxing Ren. 2023. Rtlfixer: Automatically fixing rtl syntax errors with large language models. *arXiv preprint arXiv:2311.16543* (2023).

[48] Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, and Daniel Johnson. 2010. Goldmine: Automatic assertion generation using data mining and static analysis. In *DATE*.

[49] Ning Wang, Bingkun Yao, Jie Zhou, Xi Wang, Zhe Jiang, and Nan Guan. 2024. Large Language Model for Verilog Generation with Golden Code Feedback. *arXiv preprint arXiv:2407.18271* (2024).

[50] Hasini Witharana, Yangdi Lyu, Subodha Charles, and Prabhat Mishra. 2022. A survey on assertion-based hardware verification. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–33.

[51] Zhiyao Xie. 2022. *Intelligent Circuit Design and Implementation with Machine Learning*. Ph. D. Dissertation. Duke University.

[52] Ke Xu, Jialin Sun, et al. 2024. MEIC: Re-thinking RTL Debug Automation using LLMs. *arXiv preprint arXiv:2405.06840* (2024).

[53] Sichao Yang and Ye Yang. 2024. FormalEval: A Method for Automatic Evaluation of Code Generation via Large Language Models. In *ISEDA*.

[54] Xufeng Yao, Haoyang Li, Tsz Ho Chan, Wenyi Xiao, Mingxuan Yuan, Yu Huang, Lei Chen, and Bei Yu. 2024. Hdldebugger: Streamlining hdl debugging with large language models. *arXiv preprint arXiv:2403.11671* (2024).

[55] Yongan Zhang, Zhongzhi Yu, et al. 2024. MG-Verilog: Multi-grained Dataset Towards Enhanced LLM-assisted Verilog Generation. *arXiv preprint arXiv:2407.01910* (2024).

[56] Junchen Zhao and Ian G Harris. 2019. Automatic assertion generation from natural language specifications using subtree analysis. In *DATE*.

[57] Yang Zhao, Di Huang, et al. 2024. CodeV: Empowering LLMs for Verilog Generation through Multi-Level Summarization. *arXiv preprint arXiv:2407.10424* (2024).