RTLCoder: Fully Open-Source and Efficient LLM-Assisted RTL Code Generation Technique

Shang Liu, Wenji Fang *Graduate Student Member, IEEE*, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang *Member, IEEE*, Zhiyao Xie *Member, IEEE*

***Abstract*—The automatic generation of RTL code (e.g., Verilog) using natural language instructions and large language models (LLMs) has attracted significant research interest recently. However, most existing approaches heavily rely on commercial LLMs such as ChatGPT, while open-source LLMs tailored for this specific design generation task exhibit notably inferior performance. The absence of high-quality open-source solutions restricts the flexibility and data privacy of this emerging technique. In this study, we present a new customized LLM solution with a modest parameter count of only 7B, achieving better performance than GPT-3.5 on all representative benchmarks for RTL code generation. Especially, it outperforms GPT-4 in VerilogEval Machine benchmark. This remarkable balance between accuracy and efficiency is made possible by leveraging our new RTL code dataset and a customized LLM algorithm, both of which have been made fully open-source. Furthermore, we have successfully quantized our LLM to 4-bit with a total size of 4GB, enabling it to function on a single laptop with only slight performance degradation. This efficiency allows the RTL generator to serve as a local assistant for engineers, ensuring all design privacy concerns are addressed.**

I. Introduction

In recent years, large language models (LLMs) such as GPT [1] have demonstrated remarkable performance in natural language processing (NLP). Inspired by this progress, researchers have also started exploring the adoption of LLMs in agile hardware design [2]. Many new LLM-based techniques emerge and attract wide attention in 2023. For example, LLM-based solutions are proposed to generate design flow scripts to control EDA tools [3], [4], design AI accelerator architectures [5], [6], design quantum architectures [7], hardware security assertion generation [8], fix security bugs [9], and even directly generate the target design RTL [4], [10]–[20].

Among the above explorations, a promising direction that perhaps attracts the most attention is automatically generating design RTL based on natural language instructions [4], [10]–[18]. Specifically, given design functionality descriptions in natural language, LLM can directly generate corresponding hardware description language (HDL) code[1] such as Verilog, VHDL, and Chisel from scratch. Compared with well-explored *predictive* machine learning (ML)-based solutions in EDA [23], such *generative* methods benefit the hardware design and optimization process more directly. This LLM-based design generation technique can potentially revolutionize the existing HDL-based VLSI design process, relieving designers from the tedious HDL coding tasks.

Works	New Training Dataset	New LLM Model	Outperform GPT-3.5
Prompt Engineering [10]–[12], [15], [16]	N/A	N/A	N/A
Thakur et al. [14] MG-Verilog [17] Goh et al. [18]	**Open-Source**	**Open-Source**	No
VerilogEval [13] ChipNeMo [4] ChipGPT-FT [21]	Closed-Source	Closed-Source	Comparable
BetterV [22]			**Yes**
RTLCoder	**Open-Source**	**Open-Source**	**Yes**

TABLE I: LLM-based works on automatic design RTL (e.g., Verilog) generation based on natural language instructions.

Table I summarizes existing works in LLM-based design RTL generation. Some works [10]–[12], [15], [16] focus on prompt engineering methods based on commercial LLMs like GPT, without proposing new datasets or models for RTL code generation. As we will discuss later, reliance on commercial LLM tools limits in-depth research exploration and incurs serious privacy concerns in industrial IC design scenarios. Thakur et al. [14] generate a large unsupervised training[2] dataset by collecting Verilog-based projects from online resources like GitHub, then fine-tuning its own model. However, this unsupervised dataset is quite unorganized with a mixture of code and text. Evaluations on a third-party benchmark [12] show that the performance of its fine-tuned model is still inferior to commercial tools like GPT-3.5. The VerilogEval [13] from the NVIDIA research team proposes its own labeled training dataset and benchmark, then fine-tunes its own new model. This may be the first non-commercial model that claims comparable performance with GPT-3.5, but according to their authors, neither the training dataset nor

Manuscript received XXXX; revised XXXX; accepted XXXX. Date of publication XXXX; date of current version XXXX. This work is partially supported by Hong Kong Research Grants Council (RGC) ECS Grant 26208723, National Natural Science Foundation of China (62304192, 92364102), and ACCESS – AI Chip Center for Emerging Smart Systems, sponsored by InnoHK funding, Hong Kong SAR. (*Corresponding author: Zhiyao Xie.*)

Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang and Zhiyao Xie are with the Department of Electronic and Computer Engineering at the Hong Kong University of Science and Technology (HKUST), Hong Kong SAR, China (email: sliudx@connect.ust.hk; wfang838@connect.ust.hk; yludf@connect.ust.hk; jwangjw@connect.ust.hk; qzhangcs@connect.ust.hk; hongcezh@ust.hk; eezhiyao@ust.hk).

Hongce Zhang is also with the Microelectronics Thrust at the Hong Kong University of Science and Technology (GZ), Guangzhou, China.

[1]Most existing works focus on generating design RTL in Verilog code. In this work, we also choose Verilog, while the method should be general and applicable to other HDL types like VHDL. We will use terms *RTL code* and *Verilog code* interchangeably.

[2]Most customized LLM solutions (including RTLCoder) are developed by fine-tuning pre-trained LLMs based on a training dataset about the specific task. In this paper, we use the terms *training* and *fine-tuning* interchangeably.

fine-tuned LLM model will be released to the public in the near future [13]. Goh et al. [18] proposed an open-source dataset with Verilog instruction-code samples. However, their instructions have only module name information, without code functionality descriptions. A recently released dataset MG-Verilog [17] proposed a multi-grained dataset consisting of descriptions of various level details with code to enhance the model's instruction-following ability. But the model performance of [17], [18] also cannot outperform GPT3.5. Besides these customized RTL-generation solutions, according to our study, all other software code (e.g., Python) generation models like CodeGen2 [24], StarCoder [25], and Mistral [26] are significantly inferior to GPT-3.5 in this RTL generation task.

Compared with solutions based on closed-source commercial LLM tools like GPT, the open-source LLM solution is vitally important from both research and application perspectives: 1) For research purposes, obviously, closed-source commercial tools prevent most in-depth studies and customizations of this emerging technique. 2) For realistic applications, users of commercial LLM tools unavoidably have data privacy concerns, since all instructions have to be uploaded to LLM providers like OpenAI. The privacy concern is especially critical in the VLSI design industry, where information leakage of intellectual property (IP) or key technical innovations can seriously hurt the competitive advantage of users' companies. In comparison, each user's own local LLM developed based on an open-source solution can eliminate all privacy concerns and also ensure a reliable service.

However, as mentioned, high-performance open-source RTL generation models are currently unavailable. According to our study, a major challenge is the unavailability of high-quality circuit design data for training: 1) Organized design data is mostly owned by semiconductor companies, who are almost always unwilling to share design data. 2) Design data directly collected online is messy and unorganized, either leading to inferior model performance or requiring prohibitive human efforts to clean the dataset.

In this work, we finally fill this gap with our new open-source LLM solution named **RTLCoder**[3]. To the best of our knowledge, it is the first non-commercial and open-source LLM method that clearly outperforms GPT-3.5 in design RTL code generation. We validate this on two representative benchmarks [12], [13] and observe consistent trends. To build this RTLCoder, we first propose an automated data generation flow and have generated a dataset with over 27,000 instruction-code samples for the RTL generation task.

RTLCoder obviously achieves state-of-the-art trade-offs between performance and efficiency. Besides demonstrating unprecedented RTL generation correctness in non-commercial solutions, it only has 7 billion (B) parameters and can be trained with only 4 consumer-level GPU cards. After further quantizing the parameters to 4 bits, the RTLCoder-4bit takes only 4GB of memory and can work on a laptop with limited accuracy loss. As a result, our open-source lightweight RTL-Coder solution is accessible to almost every research group and it is friendly to be implemented and improved by researchers. The contributions of RTLCoder can be summarized below:

- Targeting Verilog code generation, we propose an automated flow to generate a large dataset with over 27 thousand diverse Verilog design problems and answers. It addresses the serious data availability challenge in IC design-related tasks, and its potential applications are not limited to LLMs. The LLM directly trained on it can already achieve comparable accuracy with GPT-3.5.
- We introduce a new LLM training scheme based on code quality feedback. It further boosts the ultimate model performance to outperform GPT-3.5, being comparable with GPT-4. We further revised the training process from an algorithm perspective to reduce its GPU memory consumption. The training process only requires 4 commercial-level GPU cards.
- We designed RTLCoder to be a lightweight solution with only 7B parameters. After quantizating its parameters into 4 bits, it takes only 4GB of memory, allowing it to serve as a local assistant for engineers without privacy concerns.
- RTLCoder has been fully open-sourced, including our data generation flow, complete generated dataset, LLM training algorithm, and the fine-tuned model. Considering RTLCoder's lightweight property and low hardware barrier, it allows anyone to easily replicate and further improve based on our existing solution.

II. Automatic Dateset Generation

In this work, we first propose a new automated training dataset generation flow. Based on this flow, we have generated over 27 thousand training samples, with each sample being a pair of design description instruction (i.e., model input) and the corresponding 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 answer from LLMs, implementing the circuit functionality in Verilog code. We observe that these generated training samples exhibit high diversity and complexity in the RTL-generation domain, encompassing a diverse spectrum of difficulty levels.

We build this automated generation flow by taking full advantage of the powerful general text generation ability of the commercial tool GPT. Please notice that GPT is only used for dataset generation in this work, and we adopt GPT-3.5 in this data generation task. The automated dataset generation flow is illustrated in Figure 1, which includes three stages: 1) RTL domain keywords preparation, 2) instruction generation, and 3) reference code generation. We designed several general prompt templates to control GPT generating the desired outputs in each stage.

A. Stage 1: Keywords Preparation

The first stage of our data generation flow targets preparing RTL domain keywords for subsequent stages. At process ❶ shown in Figure 1, we request GPT to generate keywords

[3]Our RTLCoder has been fully open-source at https://github.com/hkust-zhiyao/RTL-Coder. It includes the data generation flow, the complete generated training data set, the model training flow, and the final fine-tuned models (based on both Mistral and DeepSeek).

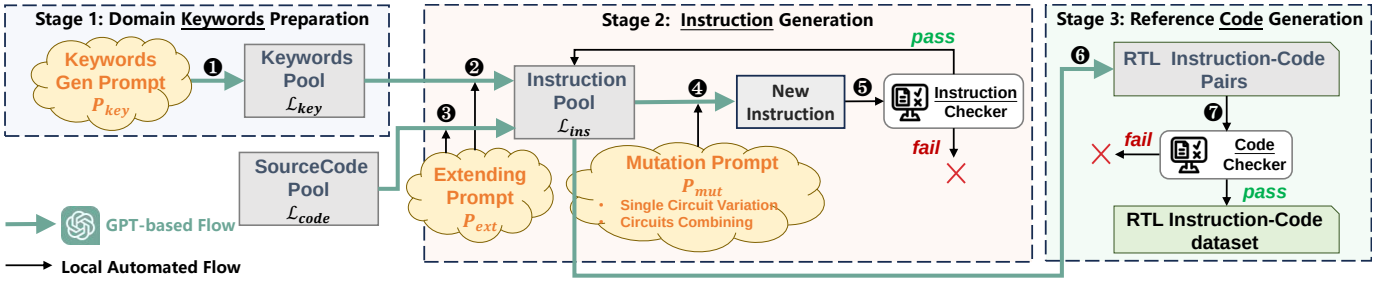


Fig. 1: Our proposed automated training dataset generation flow.

Prompt:
Please act as a professional Verilog designer. List the categories and examples of blocks that are frequently used in Verilog coding tasks.

Fig. 2: An example of Prompt P_{key} in ❶[4]

Response:
1. Combinational Logic Blocks: Arithmetic blocks (adders, subtractors, multipliers, dividers)...
2. Sequential Logic Blocks: Shift Registers (serial-in/serial-out, serial-in/parallel-out, parallel-in/serial-out, parallel-in/parallel-out), Sequence detectors...
3. Finite State Machines (FSMs): Mealy FSM, Moore FSM, One-hot FSM, Gray-code FSM...
4. **Digital Signal Processing (DSP) Blocks:** Filters (FIR, IIR), Fast Fourier Transform (FFT)...
5. Communication Protocol Blocks
...

Fig. 3: A GPT response example to Prompt P_{key} in ❶

related to digital IC design (i.e., commonly used logic components) based on a set of prompts P_{key}. We obtain a keyword pool $\mathcal{L}_{key}$ with hundreds of digital design keywords.

Specifically, in this process ❶, to collect a comprehensive range of RTL design task topics, we utilize a tree-like structure with multiple branches to issue queries to GPT. We first prompt GPT at the root node to provide categories and examples of frequently used block keywords in RTL design as Figure 2 illustrated. The response from GPT has a tree structure that consists of some subfields as Figure 3 shows. With the response, we could use the categories and examples as branches to continue prompting GPT for more design keywords within each topic. For example, we can use scripts to ask GPT about more types of the block "multiplier", it will return more specific design names such as "Booth multiplier, Wallace tree multiplier, etc.". After this process, we obtain hundreds of keywords related to RTL design in the Keywords pool $\mathcal{L}_{key}$.

Besides using our introduced keywords generation flow, we believe manually collecting initial Verilog instructions from existing IP families is also feasible and can help to produce a comprehensive keyword pool, as long as users devote manual efforts to collecting the IP document resources and checking their coverage and diversity.

This obtained keyword pool with about 350 instances will be extended into around 5K instruction samples in process ❷ and these instructions will be further mutated and combined with each other and those obtained from source code in stage 2 to generate more new descriptions.

B. Stage 2: Instruction Generation

The second stage targets generating sufficient instructions based on the initial keywords and Verilog source code. At process ❷, we extend existing keywords from $\mathcal{L}_{key}$ to complete design instructions. Specifically, we randomly sample one or two keywords from $\mathcal{L}_{key}$ each time, combined with prompts P_{ext}, and feed them into GPT. The output is a complete RTL design instruction.

In addition to keyword-based instruction generation in process ❷, we also propose to generate instructions based on existing source code collected by us, as shown in process ❸. This is partially inspired by the work of [27]. By providing GPT with either part or a complete Verilog code $\mathcal{L}_{code}$ collected by [14], we can inspire it to create a related Verilog design problem. By adopting this new ❸ together with ❷, we further enhance the diversity of our dataset by utilizing a vast and varied collection of source code.

Process ❷ and ❸ help generate the initial design instruction pool $\mathcal{L}_{ins}$ based on our customized prompt P_{ext}. Two types of prompt P_{ext} are proposed for processing $\mathcal{L}_{key}$ and $\mathcal{L}_{code}$, denoted as P_{ext}^{key} and P_{ext}^{code}, respectively. As shown in Figure 4, our prompt P_{ext}^{key} in process ❷ adopts the few-shot prompting technique, which means we provide an example of the question (i.e., keyword) and answer (i.e., instruction) in the input prompt. Figure 5 shows an example of GPT's corresponding response. As for the prompt P_{ext}^{code} used in process ❸, an example of prompt and the response of GPT are provided in Figure 6 and Figure 7. The prompt P_{ext}^{code} asks GPT to convert the given Verilog code snippet to the corresponding description instruction.

For process ❸, instructions obtained through code snippets from $\mathcal{L}_{code}$ focus more on describing detailed circuit behavior (such as the specific behavior in different cycles of a signal). On the other hand, the instructions obtained from the keyword pool $\mathcal{L}_{key}$ in process ❷ tend to provide an overall summary of the circuit's functionality in a high-level manner. For example, based on the keyword "traffic light" in the keyword pool, a

[4]We use *red* text boundary to denote GPT *input* examples, and *green* text boundary to denote GPT *output* examples in this work. Please notice that some *green* GPT *output* in this data generation flow are instructions, which will be the input of LLMs.

You should **create a task** that only requires one Verilog module related to the given topic.
Here is an example for you.

[Given Topic]
UART transmitter

[Instruction]
Create a Verilog module for a UART transmitter that can send data at a baud rate of 9600. The module should have a single input for the data to be transmitted (8 bits) and ...

```
module uart_transmitter(
    input clk, // system clock
    input reset_n, // active low asynchronous reset
    input [7:0] data_in, // data to be transmitted
    output reg tx, // UART transmit pin
    output reg tx_busy // UART transmit busy signal
);
```

Fig. 4: An example of Prompt P_{ext}^{key} in ❷

series of traffic light circuit instructions can be generated. These overall descriptions can hardly be obtained solely through code snippets if the signal name is not related to something like 'red_light' or 'yellow_light'. The instructions generated through keywords not only describe circuit behavior but also link to commonly used digital design terminologies and higher-level abstracts. Hence, both initial instruction generation methods are crucial and complementary. They describe circuit behavior from different levels of description (highly abstract and detailed to the behavior of each signal).

In the extension process of ❷ and ❸, the prompt for generating the instruction responses should be carefully designed. We provide Figure 6 for the reader's reference. The instructions returned by GPT should contain detailed and unambiguous requirements to ensure that LLM can generate only one functionally unique Verilog code based on this instruction. What's more, the design complexity should be limited so that the corresponding Verilog code can be implemented in hundreds of code lines.

After generating the initial instruction pool $\mathcal{L}_{ins}$ with hundreds of initial instructions, we will iteratively use mutation methods to significantly augment the scale and complexity of this pool[5]. At ❹, we use P_{mut} to apply two types of mutation operations on instructions sampled from the design instruction library $\mathcal{L}_{ins}$. The process ❹ would check every new design instruction using a set of rules and only passed valid instructions are added to $\mathcal{L}_{ins}$. We cover more details of this iterative process below.

For the mutation operation in ❹, we propose two types of prompts P_{mut}^{s} and P_{mut}^{c}, performing single circuit variation and circuits combination, respectively.

- The P_{mut}^{s} focuses on changing the single circuit module by requiring different functionalities while keeping the same hardware type, or implementing the same functionality but using different hardware. Figure 8 is an example of P_{mut}^{s} and Figure 9 is a GPT's response example with given instructions.
- The P_{mut}^{c} focuses on combining two circuit designs together. They are combined by either adding control logic to select using one of the two functionalities (i.e., parallel combination) or feeding one output into the other's input (i.e., serial combination). Figure 10 gives a sample of prompt P_{mut}^{c} and Figure 11 is a GPT's

[5]The scale and complexity in this context describe the target circuit. Circuit scale can be reflected by the total number of logic gates in the circuit after implementation. Circuit complexity can be reflected in its functionality complexity, which may be reflected by the number of lines in its implementation. Such metrics are also used in prior benchmarking works [12]. As an example, in Figure 10, we illustrate how combining a binary counter and a comparator can result in a circuit with a larger area and more complex functionality.

[Given Topic]
Pulse width modulator (PWM).

[Instruction]
Please act as a professional Verilog designer. Your task is to create a Verilog module that implements a pulse width modulator (PWM).
You need to design a PWM module that takes a clock signal, a reset signal, and a user-defined duty cycle value as inputs, and outputs a PWM signal. The duty cycle value should be an integer ranging from 0 to 100, representing the percentage of the PWM signal being high during each cycle.
Your module should meet the following **requirements**:

- The PWM signal should have a fixed frequency, determined by a parameter called PWM_FREQUENCY. You can assume that the input clock frequency is 100 MHz.
- The duty cycle of the PWM signal should be adjustable by changing the duty_cycle input value.
- When the rst input is high, the PWM output should be reset to low, and the duty cycle should start from the beginning of the next cycle when the reset signal goes low.
- The PWM output should transition from low to high at the beginning of each cycle and should stay high for a duration proportional to the duty_cycle input value.
- You can assume that the input clock signal is a perfect square wave with a 50% duty cycle, and the reset signal is synchronous to the input clock. The solution should be synthesizable and follow good coding practices.

```
module PWM(
    input clk,
    input rst,
    input [7:0] duty_cycle, // 8-bit duty cycle
value (0-100)
    output pwm_out
);
```

Fig. 5: A GPT response example (green) to Prompt P_{ext}^{key} when given topic: Pulse width modulators (PWM) (red) in ❷

Prompt:
Please gain inspiration from the following random code snippet to create a high-quality Verilog coding problem.
Verilog code snippet for inspiration:

```
if ((C_REG_CONFIG == 32'h00000001) || (C_REG_CONFIG
==  32'h00000008))  begin  reg  [C_DATA_WIDTH-1:0]
m_payload_i;  reg  [C_DATA_WIDTH-1:0]  skid_buffer…
has_valid_storage_i = 1'b1; // Clear the value if
it's set and Master side completes the transaction
but we don't have a new slave side.
```

The Verilog coding problem should be completely self-contained, providing all the contextual information one needs to understand and write code to implement this module. Assume common programming knowledge, but ensure that any specific context, variables, or code snippets pertinent to this description are explicitly included. You should describe the behavior of the circuit in detail.

Fig. 6: An example of Prompt P_{ext}^{code} in ❸

Response:
This Verilog module implements a configurable slave interface with three different modes of operation. The module has the following inputs and outputs…The module has three modes of operation, selected by the value of C_REG_CONFIG: Mode 1 (C_REG_CONFIG == 32'h00000001 or C_REG_CONFIG == 32'h00000008): This mode is a pass-through mode where the slave interface forwards data from the master interface to the slave interface and vice versa…

```
module configurable_slave_interface (
    input ACLK,
    input ARESET,
    input [31:0] C_REG_CONFIG,
    input C_DATA_WIDTH = 32,
    input S_VALID…)
```

Fig. 7: A GPT response example to Prompt P_{ext}^{code} in ❸.

The **rewritten task** should achieve different circuit functionality but require similar methods or components in the given instruction. Here is one example for you.

[Given Instruction]
Please act as a professional Verilog coder. Create a module that implements a finite state machine (FSM), used for 5-bit sequence detection.

[New Instruction]
Please act as a professional Verilog designer. Create a module that implements a finite state machine (FSM), used for 5-bit sequence detection, but with a twist. The module should detect two different 5-bit sequences, "01010" and "10101", and output a signal indicating which sequence was detected. The output signal should be '0' for "01010" and '1' for "10101". The FSM should transition to the initial state whenever a reset signal is asserted. Implement the FSM using a Mealy machine, where the output is a function of both the current state and the input.

Fig. 8: An example of Prompt P_{mut}^{s} in ❹

[Given Instruction]
Build a **decade counter** that counts from 0 through **9**, inclusive, with a period of **10**. The reset input is synchronous, and should reset the counter to 0. We want to be able to **pause the counter** rather than always incrementing every clock cycle, so the slowena input indicates when the counter should increment...

Response:
[New Instruction]
Build a **16-bit binary counter** that counts from 0 through **15**, inclusive, with a period of **16**... Additionally, include a count_down input to control the **direction of the counter (count up or down)**...

Fig. 9: A GPT response example (green) to Prompt P_{mut}^{s} with Given Instruction (red) in ❹

response example with given instructions.

It is important to note that we need to carefully design the instruction generation prompt considering the following two aspects in the mutation process:

- Basic instruction validity[6]. For example, GPT may provide an instruction of implementing a physical temperature regulator or a speedometer, which are not directly related to Verilog coding.

[6]"Validity" here refers that the instruction can be implemented solely by Verilog code, requiring no other tools or languages.

You should create a **new task** based on the two given instructions. The output of one module could be the input of the other. Here is an example for you.

[Given instructions]
1. Create a module that implements a **binary counter**...
2. Create a module that implements a **comparator**...

[New instruction]
Create a module that implements a **binary counter** that counts from 0 to 15, ..., Additionally, include a **comparator** that compares the value of the counter to a given value, and outputs a signal indicating whether the counter value is greater than or equal to the given value.

Fig. 10: An example of Prompt P_{mut}^{c} in ❹

- Level of circuit behavior detail for LLM to generate correct code. Other studies [14], [17] have also indicated that the level of circuit description has a significant impact on the code generation quality. When descriptions are overly vague, LLMs trained on this dataset struggle to align the functionality of the code with the general instruction. Conversely, if descriptions are too detailed, focusing on intricate RTL circuit specifics, the RTL generation effectively becomes a form of "code translation", which can also not boost the general generative abilities of trained LLMs.

Moreover, we request GPT to generate its reasoning steps (i.e., how it analyzes the code generation task step-by-step). These reasoning steps further enhance the detailed information of our instruction pool. In addition to incorporating these principles in the query prompt, we also use the one-shot technique (provide a query-response sample for GPT's reference) to demonstrate our requirements. As long as following these principles and adopt basic prompt engineering techniques, we believe other alternative prompts may achieve similar response quality.

For the instruction checking in ❺, we automatically check the correctness and diversity of each new instruction generated in the mutation process, based on a series of rules. The rules include 1) invalid-word checking, and 2) similarity measurement with existing $\mathcal{L}_{ins}$. For example, in invalid-word checking, if words such as "image" and "text" are detected, it indicates that GPT has likely provided a problem that cannot be implemented solely with RTL code. As for similarity rate checking, we calculate the similarity between the new instruction and existing $\mathcal{L}_{ins}$. The new instruction is discarded if the similarity is above a certain threshold since it can't add enough new information to $\mathcal{L}_{ins}$. Such checking of instructions in ❺ is imperfect, but it is fully automated and accurate enough to generate an ultimate instruction pool $\mathcal{L}_{ins}$, including over 50,000 instructions.

C. Stage 3: Reference Code Generation

The third stage targets generating the reference code. In the third stage, as shown in ❻, we feed each instruction from $\mathcal{L}_{ins}$ into GPT, generating the corresponding reference design code as the solution. Specifically, we will ask GPT to provide at least five answers. After that, in ❼, we will evaluate these answers using a code checker. In this work, we adopt an automated syntax checker to verify the correctness

[Given Instruction]
1. Please act as a professional Verilog coder. Create a module that implements a **finite state machine (FSM),** used for **5-bit sequence detection**...
2. You are tasked with designing a **4-bit binary to BCD (Binary Coded Decimal) converter** using Verilog. The binary input should be in two's complement notation, and the output should be in BCD format.

Response:
[New Instruction]
Create a module that combines a **5-bit sequence detection finite state machine (FSM)** and **a 4-bit binary to BCD converter**. The FSM module should detect a specific 5-bit sequence in the input data stream. When the sequence is detected, the FSM should output a 'match' signal. The binary to BCD converter should accept a 4-bit two's complement binary input and convert it to BCD format. The BCD conversion should only be performed when the 'match' signal from the FSM is high.

Fig. 11: A GPT response example (green) to Prompt P^c_{mut} with Given Instruction (red) in ❹

of RTL code syntax. Only syntax-correct design code can be kept. If all five answers fail the syntax checking, this instruction will be discarded. Finally, only valid instruction-code pairs are saved as our dataset. Ideally, process ❼ should also check whether the functionality of the generated RTL code is consistent with the instruction, but currently generating testbenches for functionality verification cannot be automated. Similar to the checker in stage 2, this imperfect automated checking can already filter out the most serious mistakes in the dataset.

D. Dataset Generation Flow Discussion

After going through all three proposed stages, we generate the ultimate training dataset named RTLCoder-27K with more than 27,000 instruction-code data samples. In this subsection, we will discuss some important topics about our proposed data generation flow and the output dataset. Multiple experiments are conducted to provide more insights about our RTLCoder dataset.

Statistics of keyword pool. In the keyword generation process, we adopt the hierarchical prompting structure to query GPT, resulting in around 350 keywords. Experienced engineers have recognized the keyword pool as covering a vast majority of commonly used circuit design problems. We also adopted GPT to analyze the keyword types and illustrate the categories and distribution of this keyword pool in Figure 12(a).

Comparison with other RTL dataset. We compare RTLCoder with other recent RTL code datasets [13], [14], [17], [18] in the following. In Thakur et al. [14], the training dataset only consists of code without instructions. Such code-only datasets do not align well with the RTL generation tasks based on natural language instructions. After removing exactly the same duplicated code in the dataset, there are approximately 25K data entries. In VerilogEval [13], each data entry comprises an instruction-code pair, totaling 8.5K entries in their dataset. However, their dataset is not open-sourced. In the MG-Verilog dataset [17], there are 11K instances with each one consisting of descriptions at various levels of detail and corresponding code. In the Goh et al. [18], there are a total of 60K samples. However, their instruction only provides the module name information without any functionality descriptions. The performance of LLMs [13], [14], [17], [18] trained on these 4 datasets will be presented in Table III in the following Section IV and all cannot clearly outperform GPT3.5. RTLCoder significantly outperforms all these models, indicating the overall quality of our open dataset.

Evaluation of design type distribution. We also utilized GPT for annotating the categories of the generated 27K-dataset. The circuit type distribution statistics is depicted in Figure 12(b). The distribution statistics for the two benchmarks, RTLLM-1.1 [12] and VerilogEval [13] are also plotted in Figure 12(c) and (d) respectively. The generated dataset consists of a wide range of circuit designs including Arithmetic Units, Signal Processing Units, Encryption Units, etc. It is important to note that a single data sample may implement functionalities of multiple circuit types, and we only adopt one circuit type to represent each instruction. As the majority of circuits contain submodules like Multiplexers, Decoders, and selectors, this may result in a relatively higher proportion of the Data Path type.

Evaluation of dataset diversity. To further check the diversity of our proposed training dataset, RTLCoder-27K, we utilized two diversity measures: Compression Ratios (CR) and Part-of-Speech Compression Ratio (CR: POS) which are suggested the best lexical diversity metrics by [28] among Homogenization Score (BERTScore), Self-BLEU, Homogenization Score (ROUGE-L), N-Gram Diversity Score, Hypergeometric Distribution, etc. CR is calculated utilizing text compression algorithms which can identify redundancy in the whole contents. We constructed a file D containing all the text data in RTLCoder-27K following [28]. Subsequently, we compressed the entire dataset using gzip. The Compression Ratio (CR) is as follows:

$$\mathrm{CR}(D) = \frac{\text{size of } D\oplus}{\text{compressed size of } D\oplus}$$

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 [28] and used NLTK POS tagger and the Penn Treebank set of 36 tags to extract the tag sequences [28].

The results are shown in Table II. Additionally, we also provide the CR and CR:POS values for three representative fine-tuning datasets: Goh et al. [18], MG-Verilog [17], and Magicoder-OSS-Instruct-75K [27] for comparison. A smaller CR and CR-POS value indicates a higher level of diversity in the dataset. We observe that RTLCoder-27K exhibits similar diversity to the widely used Python benchmark and higher diversity compared to the Verilog code datasets.

Imperfection in data functionality correctness. Our dataset generation flow ensures the syntax correctness of all instruction-code data samples, but cannot guarantee that every data sample is functionally correct (i.e., code implements the same functionality as described in instruction). This is because automatically checking the functionality correctness

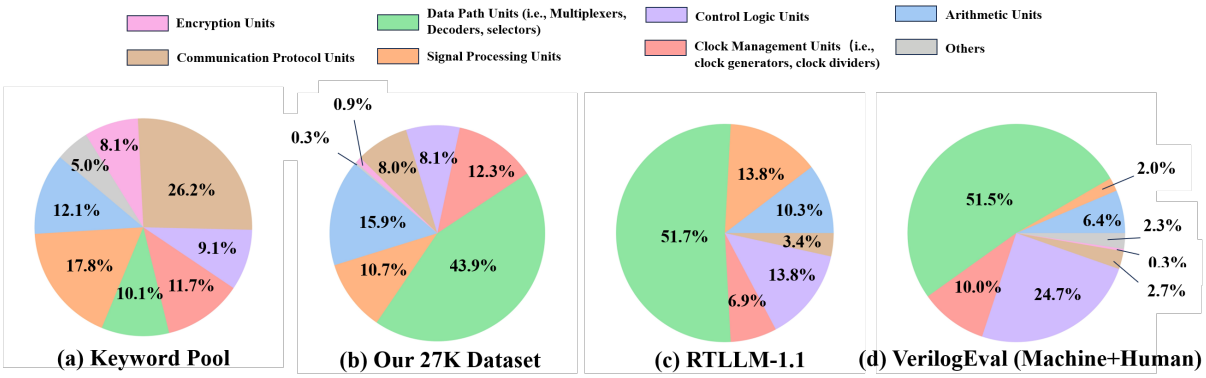


Fig. 12: The circuit type distribution of keyword pool, the obtained dataset RTLCoder-27K, RTLLM-1.1 [12] and VerilogEval [13].

	RTLCoder-27K	Goh et al. [18]	MG-Verilog [17]	Magicoder-OSS-Instruct-75K [27]
CR	4.41	5.27	5.80	4.02
CR: POS	7.61	10.1	9.16	6.67

TABLE II: The diversity scores of 4 representative fine-tuning instruct datasets. There are three Verilog datasets, including our RTLCoder-27K, Goh et al. [18] and MG-Verilog [17]. We also evaluate a widely used software (mainly Python) dataset, Magicoder-OSS-Instruct-75K [27]. The lower value of CR and CR: POS, the higher dataset diversity. RTLCoder-27K exhibits a satisfactory level of diversity compared with the other three datasets.

of training samples is inherently a very challenging task. Functionality checking for the Verilog code is practically hardware verification, which has been studied for decades, relies on human engineers, and is difficult to get guaranteed results. Although the code in our proposed dataset has only undergone syntax checks, finetuning the model using this dataset can still lead to improved model performance on the benchmarks as Table III shows in section IV. This indirectly demonstrates the contribution of the proposed dataset.

Relation between RTLCoder and GPT-3.5. An interesting observation is that, although we generate our training dataset based on GPT-3.5, RTLCoder turns out to outperform the GPT-3.5 baseline on representative benchmarks [12], [13]. One important reason is that, for each instruction, we have employed a syntax checker to evaluate the reference code generated based on GPT-3.5. Therefore, among all correct and incorrect code from GPT-3.5, we filter out the obviously incorrect ones and retain the largely correct ones for training RTLCoder. This process can be viewed as a refinement of GPT-3.5's Verilog generation capabilities.

III. New Training Scheme Incorporating Code Quality Feedback

Besides the new training dataset, we propose a new LLM training scheme that incorporates code quality scoring. It significantly improves the RTLCoder's performance on the RTL generation task. Also, we revised the training process from the algorithm perspective to reduce the GPU memory consumption of this new training method, allowing implementation with limited hardware resources.

A. Existing Supervised Training on LLMs

This part will first introduce the existing supervised training method for LLMs. Then we will further discuss its limitations in RTL generation tasks. Suppose we have a training data dateset $\{x_i, y_i\}$ for $i = 1, ..., N$, where x_i represents an design instruction, y_i represents the corresponding correct reference code. Each sample of data will be split into a sequence of tokens by certain rules during the preprocessing process. In this paper, we use $x_i = \{x_i^t\}$ and $y_i = \{y_i^t\}$ for $t = 1, 2, ..., T$ to represent the tokenized sequence.

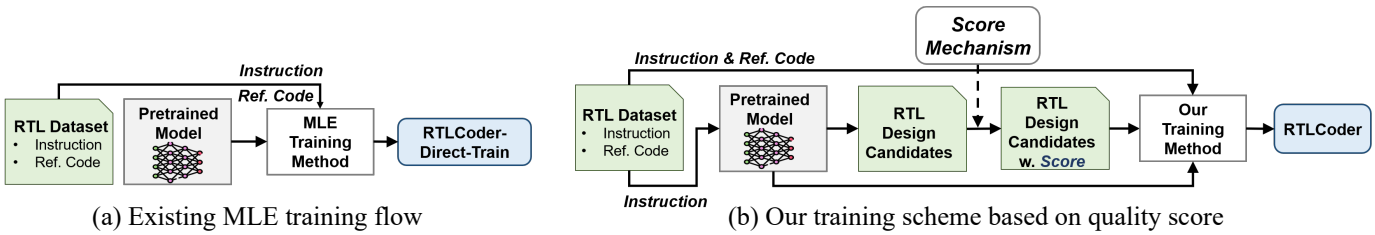
LLMs generate a sequence by continuously predicting the next token based on the already generated previous ones. For a decoder-only language model, which is the mainstream LLM architecture, the probability of producing the next token depends only on the previous output tokens and the input instruction. We denote the probability of generating the t-th token r_t (r_t can be any single token in the vocabulary) as $P_\pi\left(r_t \mid x_i, y_i^{<t}\right)$ where π represents the model parameters and $y_i^{<t}$ denotes the already generated previous tokens $\left\{y_i^1, .., y_i^{t-1}\right\}$. Then the log probability of generating the whole sequence can be written as: $\sum_{t=1}^{T} \log P_\pi\left(y_i^t \mid x_{i,}, y_i^{<t}\right)$.

In the existing training method, Maximum Likelihood Estimation (MLE) is commonly used to find the best parameters π that maximize the log probability. The training flow is shown in Figure 13(a). The loss is usually defined as below:

$$loss_{mle} = -\sum_{t=1}^{T} \log P_\pi\left(y_i^t \mid x_{i,}, y_i^{<t}\right)$$

However, there exists a phenomenon named *exposure bias* [29], [30]. Since the above sequence generation is autoregressive, which means the model always predicts the next token based on its own generated previous ones $r_i^{<t}$ rather than the reference tokens $y_i^{<t}$. Therefore, even though the probability of producing y_i^t is high when given $y_i^{<t}$ in the training, it can still result in a huge deviation from the reference code in the generation process.

We have also observed this phenomenon in our experiments. After the supervised training, the qualities of multiple generated code candidates for the same instruction may diverse greatly in the performance aspect. They can include correct code while at the same time including many low-quality an-



(a) Existing MLE training flow (b) Our training scheme based on quality score

Fig. 13: Comparison between (a) existing MLE-based LLM training flow and (b) our proposed LLM training flow.

swers. Some candidates exhibit serious nonsense duplication[7].

To alleviate the *exposure bias* phenomenon, we suggest that in addition to the reference code y_i, the model's generation should also be considered in the training process. Since the generation may be different from the reference code, it is necessary to introduce a scoring mechanism to judge the quality of generated candidates. We will give our detailed solution in Section III-B.

B. Our Proposed Training Method

Our proposed training scheme is illustrated in Figure 13(b). For each instruction, we will now collect multiple code candidates generated by the initial pre-trained model. Then, we pack these candidates and the original reference code y_i together as $\mathbf{y}_i = \{y_{i,k}\}$, $k = 1, 2, .., K$, where K represents the number of generated code for one instruction. Next, all these candidates will be scored by the scoring mechanism $R(x_i, y_{i,k})$ which could be a syntax checker or unit test for functionality check. We will then obtain a set of score $\mathbf{z}_i = \{z_{i,k}\}$, $k = 1, 2, .., K$, denoting the quality for the code sample $\{y_{i,k}\}$. In the training process, we aim to make the model learn to assign relatively higher generation probabilities to answers with higher scores. In this way, the model not only learns from the reference code, but also from the new information introduced by the quality score feedback.

The conditional log probability (length-normalized) of generating the entire code $y_{i,k}$ is commonly written as:

$$p_{i,k} = \frac{\sum_t \log P_\pi \left(y_{i,k}^t \mid x_i, y_{i,k}^{<t} \right)}{\|y_{i,k}\|}$$

We calculate $p_{i,k}$ for all code candidates $\mathbf{y}_i = \{y_{i,k}\}$, $k = 1, 2, .., K$, then we normalize these $p_{i,k}$ values using a $softmax$ function, defining the probability of each code being selected as:

$$s_{i,k} = \frac{e^{p_{i,k}}}{\sum_{\tau=1}^{K} e^{p_{i,\tau}}}$$

This $s_{i,k}$ reflects the model's tendency to output the k^{th} code candidate, with higher probabilities indicating a greater likelihood that the model will generate it.

To encourage the model to assign higher probability scores to high-quality code, we can define a new loss function term:

[7]We notice that this duplication couldn't be simply dealt with by adding repetition penalty to the decoding process like other works in natural text generation. Because some correct RTL design code also contain similarly repetitive expressions.

Algorithm 1 Training scheme using gradients splitting

Input: The single data sample $\{x_i, \mathbf{y}_i, \mathbf{z}_i\}$. Model forward function $s_{i,k} = f_\pi(x_i, y_{i,k}, z_{i,k})$. Loss calculation function $L_\pi(\mathbf{s}_i, \mathbf{z}_i)$. GPU affordable batch size J. Model parameters w.
Output: The derivative of the loss with respect to model parameters: g_i.

Group the sample $\{x_i, y_{i,k}\}$ for $k = 1, 2, ..., K$ into Q parts based on batch size J.
initialize empty vector list $temp$. Initialize the gradients $g_i = 0$.
for $q \in Q$ **do**
    Calculate $s_{i,k} = f_\pi(x_i, y_{i,k}, z_{i,k})$, for $k \in q$.
    Empty the computation graph
Calculate $loss = L_\pi(\mathbf{s}_i, \mathbf{z}_i)$ //$\mathbf{s}_i = \{s_{i,k}\}$ for $k = 1, .., K$
Backward process: $temp_k = \partial \, \text{loss} / \partial s_{i,k}$, for $k = 1, ..., K$
for $q \in Q$ **do**
    Calculate $s_{i,k} = f_\pi(x_i, y_{i,k}, z_{i,k})$, for $k \in q$
    Backward process: $g_i = g_i + \sum_{k \in q} temp_k \partial s_{i,k} / \partial w$
    Empty the computation graph
Return g_i

$$loss_{compare} = \sum_{z_{i,k} < z_{i,\tau}} max\left(s_{i,k} - s_{i,\tau} + \lambda, 0\right)$$

where λ is a threshold value.

To provide an intuitive explanation of this loss function term, we provide a simple example. Suppose we have the i^{th} instruction and only two code candidates with initial selection probability $s_{i,1}$ and $s_{i,2}$ with $s_{i,1} + s_{i,2} = 1$ and $s_{i,1} > s_{i,2}$. But the first candidate has a lower quality score, i.e., $z_{i,1} < z_{i,2}$. Then the positive loss would drive model parameters to update until the model assigns a new set of $s_{i,1}^*$ and $s_{i,2}^*$ so that $s_{i,2}^* - s_{i,1}^* \geq \lambda$ is satisfied.

It is worth noting that this loss only depends on the relative scores among multiple code candidates, so it can still be used when answer quality cannot be precisely quantified. Finally, We define the total loss as:

$$loss = loss_{compare} + loss_{mle}$$

C. Reduced Memory by Splitting Gradients

Directly calculating our new $loss$ function even with 1 batch size would still require forwarding all code candidates in a sample at once to maintain all the activation values. This will lead to the $O(K)$ space complexity and make the GPU memory consumption prohibitively high in many large language model training scenarios.

We propose a gradient-splitting approach for model training based on quality score from an algorithm perspective. It can achieve a $O(1)$ space complexity as illustrated in Algorithm 1.

The gradients of $loss$ with respect to w can be computed as below:

$$\frac{\partial \text{loss}}{\partial w} = \sum_k \frac{\partial \text{loss}}{\partial s_{i,k}} \frac{\partial s_{i,k}}{\partial w}$$

The property of the chain rule indicates that we can decompose the gradient updates into several parts. Assume J is the maximum allowable batch size for GPU consumption. We divide the K candidates into Q groups based on the batch size J. Firstly, we pass these groups through the forward function separately and collect the obtained $\mathbf{s}_i$ values as lines 1-5 illustrate. In the second step, we calculate the loss function and compute the derivative of the loss with respect to $\mathbf{s}_i$ in lines 6-7, storing the temporary results in vector $temp$. In the third step, we perform the forward operation on the original Q groups again and for each forward operation, the obtained $s_{i,k}$ is multiplied by $temp_k$ in a dot product, followed by a backward pass to accumulate the gradient in lines 9-12.

IV. Experimental Results

A. Evaluation Benchmark and Metric

To evaluate the performance of Verilog code generation, there are two representative benchmarks VerilogEval [13] and RTLLM [12].

The VerilogEval [13] benchmark consists of two parts, EvalMachine and EvalHuman, each including more than 100 RTL design tasks. We follow the original paper [13] and use the widely-adopted $pass@k$ metric in code generation tasks:

$$\text{pass@k} = E_i \left(1 - \frac{C_{n-c_i}^k}{C_n^k}\right)$$

where n is the total number of trials for each instruction and c_i is the number of correct code generations for task i. We set $n = 20$ in this experiment. If any code in the k trials could pass the test, then this task is considered to be addressed and the $pass@k$ metric reflects the estimated proportion of design tasks that could be solved.

The RTLLM V1.1 [12] benchmark contains 29 RTL design tasks at a larger design scale. We mostly follow the testing method in the original paper [12], but further proposes two slightly different metrics for evaluating syntax correctness, using either Synopsys VCS [31] or Design Compiler [32]. They are denoted as Syn-VCS and Syn-DC, respectively. 1) For the Syn-VCS metric, VCS not only requires the design to comply with the Verilog syntax rules, but also requires that the interface of the design correspond to the testbench, so that the circuit can be simulated. 2) For the Syn-DC metric, DC requires the design to be physically synthesizable. The functionality result is obtained by VCS simulation. We calculate the scores of the design syntax part and design functionality part separately. In both parts, following the original benchmark [12], each task is counted as success as long as *any* of 5 trials passes the test. This can be interpreted as pass@5 metric.

In the generation process, we set $top_p = 0.95$ and $temperature = \{0.2, 0.5, 0.8\}$. For all tested models (i.e., baselines, RTLCoder, and ablation studies), we evaluate all 3 $temperature$ conditions and report the best performance for each model.

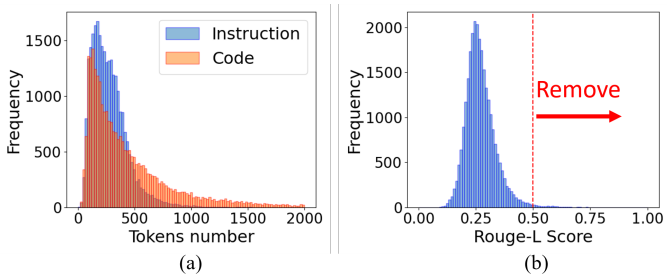


Fig. 14: Training dataset analysis. (a) Tokens number distribution of instruction and code part. (b) Similarity measurement between training dataset and two benchmarks based on Rouge-L metric.

B. Examine Training Set for Fair Evaluation

To ensure a fair evaluation of our proposed RTLCoder, before training, we explicitly examined the similarity between samples in our proposed training dataset and those test cases in benchmarks [12], [13], then we get rid of our training samples that are similar to test cases during the training process.

To measure the similarity between two text sequences, we employed the Rouge-L metric, which is a widely-used similarity calculation scheme in the LLM domain such as by OpenAI [1]. The Rouge-L score $\in [0, 1]$, with values closer to 1 indicating higher similarity between the two sequences. For each instruction-code concatenated sample in the training dataset, we computed its Rouge-L value with all test cases in the benchmarks. In addition, we also separately analyzed the distribution of token counts for instructions and code in the dataset. The resulting statistic is in Figure 14.

From Figure 14 (a), we can see that a sample that consists of one instruction and one code candidate is generally within 2048 token length. So we can set 2048 as the max length in our finetuning. In Figure 14 (b), we observed that the majority of training samples in the dataset have a low overlap compared with the benchmark, with Rouge-L scores < 0.3. However, there are still a small number of samples with higher similarity. To ensure fair evaluation of the RTLCoder, we get rid of all training samples with Rouge-L values > 0.5 which counts about 100 samples.

C. Model Training

Based on our generated dataset with 27K instruction-code pairs, we choose the latest Mistral-7B-v0.1 [26] and DeepSeek-Coder-6.7b [33] as the basic pre-trained model for finetuning. In all experiments, we opted for the Adam optimizer with β_1 = 0.9, β_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 only afford 2×2048 context length using DeepSpeed stage-2 [34]. Under the hardware constraint, the training is impossible without the proposed gradient-splitting method.

Model Type	Evaluated Model	Num of Params	VerilogEval Benchmark [13] (using pass@k metric)						RTLLM V1.1 [12] (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* [4]	13B	43.4	N/A	N/A	22.4	N/A	N/A	N/A	N/A
	VerilogEval* [13]	16B	46.2	67.3	73.7	28.8	45.9	52.3	N/A	N/A
	BetterV* [22]	7B	64.2	75.4	79.1	40.9	50.0	53.3	N/A	N/A
Open-Source Baseline	Codegen2 [24]	16B	5.00	9.00	13.9	0.90	4.10	7.25	72.4	6.90
	Starcoder [25]	15B	46.8	54.5	59.6	18.1	26.1	30.4	93.1	27.6
	Thakur et al. [14]	16B	44.0	52.6	59.2	30.3	43.9	49.6	86.2	24.1
	MG-Verilog et al. [17]	7B	52.7	58.5	60.9	N/A	N/A	N/A	N/A	N/A
	Goh et al. [18]	7B	40.6	48.4	54.4	N/A	N/A	N/A	N/A	N/A
Base Model	Mistral-7B-v0.1 [26]	7B	36.9	48.8	57.4	4.49	12.6	18.6	72.4	20.7
	DeepSeek-Coder-6.7b [33]	6.7B	54.1	63.8	67.5	30.2	42.2	46.2	89.6	34.5
Less Training Data (10K Samples)	RTLCoder-Mistral-10k	7B	56.5	66.6	69.4	31.7	42.2	46.5	86.2	34.5
	RTLCoder-DeepSeek-10k	6.7B	55.3	70.4	76.2	36.7	47.0	50.4	79.3	37.9
Direct Training	RTLCoder-Mistral-Direct	7B	58.9	70.0	74.1	34.4	42.3	45.1	89.7	41.4
	RTLCoder-DeepSeek-Direct	6.7B	59.8	73.6	77.2	39.1	48.3	51.3	86.2	44.8
RTLCoder	**RTLCoder-Mistral-4bit**	7B * 4bit	59.5	72.2	76.9	33.8	42.3	47.1	86.2	41.4
	RTLCoder-DeepSeek-4bit	6.7B * 4bit	56.5	73.2	78.4	37.5	50.5	55.5	93.1	37.9
	RTLCoder-Mistral	7B	62.5	72.2	76.6	36.7	45.5	49.2	96.6	48.3
	RTLCoder-DeepSeek	6.7B	61.2	76.5	81.8	41.6	50.1	53.4	93.1	48.3

*We cannot directly evaluate VerilogEval [13], ChipNeMo [4] and BetterV [22] on RTLLM Benchmark due to closed-source models. We fully understand and respect the authors' privacy concerns. The accuracy values of VerilogEval [13], ChipNeMo [4], BetterV [22], GPT-3.5, and GPT-4 on the VerilogEval Benchmark [13] are directly cited from the original publication [4], [13], [22].

TABLE III: Performance comparison of RTL code generators on VerilogEval Benchmark [13] and RTLLM Benchmark [12]. The top scores ranked 1st, 2nd, and 3rd in each column are marked in Green, Blue, and Red, respectively. RTLCoder outperforms GPT-4 on EvalMachine of [13]. It is only second to GPT-4 on the other benchmarks (EvalHuman of [13] and RTLLM [12]), outperforming GPT-3.5 and all others.

To implement our proposed training scheme, we first generated 3 code candidates for each instruction using the pre-trained model with the Beam search method. Then we use Pyverilog [35] as the syntax checker to score the code candidates. Specifically, we assigned a full score (i.e., 1) for the reference code from the dataset and those candidates who can pass the syntax check. For those who failed syntax checks, we used the Rouge-L metric to assign the code similarity between the candidate and reference code as its score.

In addition, considering GPU memory consumption is a crucial factor that limits the applicability of LLMs, based on quantization methodologies [36], we further quantize the parameters of the obtained RTLCoder into 4 bits, generating RTLCoder-DeepSeek-4bit and RTLCoder-Mistral-4bit, consuming only 4GB memory.

D. Experiment Results Overview

Table III summarizes the comparison of all relevant RTL generation solutions, including commercial models GPT3.5/GPT4, models customized for Verilog generation [13], [14], [22], software code generators [24]–[26], [33], our proposed RTLCoder and quantized version RTLCoder-4bit, and ablation studies of RTLCoder. In addition, we further visualize key results on VerilogEval benchmark in Figure 15.

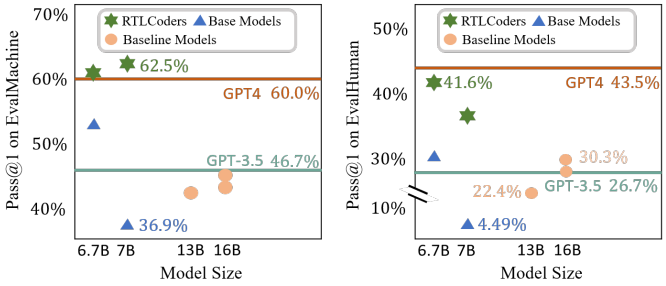


Fig. 15: Visualization of key accuracy comparisons from Table III, selecting pass@1 metric on EvalMachine and EvalHuman of [13]. RTLCoder includes both RTLCoder-Mistral and RTLCoder-DeepSeek. The baseline models include Thakur et al. [14], ChipNeMo [4] and VerilogEval [13].

In the VerilogEval benchmark [13], for both EvalHuman and EvalMachine categories, RTLCoder-DeepSeek scores 61.2 and 41.6 respectively. It clearly outperforms GPT-3.5 and is only inferior to GPT-4 among all the models in EvalHuman. Specifically, in the EvalMachine part, RTLCoder-DeepSeek and RTLCoder-Mistral even outperforms GPT4 by an absolute value of 1.2% and 2.5%. A similar trend can be observed in the RTLLM benchmark V1.1 [12]. RTLCoder is also second only to GPT-4. In summary, RTLCoder outperforms GPT-3.5 and all non-commercial baseline models in all metrics on both benchmarks. It is surprising that the lightweight RTLCoder

TABLE IV: Detailed Syntax and Functionality Evaluation Results using sampling generation method in RTLLM V1.1 [12]

Design	GPT-3.5			GPT-4			Thakur et al. [14]			StarCoder[25]			RTLCoder-Mistral-4bit			RTLCoder-Mistral		
	Syn-VCS	Syn-DC	Func	Syn-VCS	Syn-DC	Func	Syn-VCS	Syn-DC	Func	Syn-VCS	Syn-DC	Func	Syn-VCS	Syn-DC	Func	Syn-VCS	Syn-DC	Func
accu	2	2	✔	5	5	✔	4	4	✘	3	4	✘	5	5	✘	4	4	✘
adder_8bit	3	3	✔	4	4	✔	3	3	✔	2	4	✘	5	5	✔	5	5	✔
adder_16bit	1	0	✘	3	3	✔	3	4	✘	2	3	✘	0	0	-	3	3	✘
adder_32b	0	0	-	2	2	✔	1	0	✘	1	3	✘	1	0	✘	1	0	✘
adder_pipe_64b	5	5	✘	5	5	✔	0	0	-	0	0	-	1	1	✘	3	2	✘
multi_booth_8b	5	2	✘	5	5	✘	3	3	✘	4	3	✘	5	5	✔	5	5	✔
multi_16b	5	0	✔	5	5	✔	3	3	✘	3	4	✘	4	2	✘	5	5	✔
multi_pipe_4b	0	0	-	2	2	✘	1	0	✘	3	1	✘	4	1	✘	2	0	✘
multi_pipe_8b	2	0	✘	5	5	✘	3	1	✘	2	3	✘	0	0	-	2	0	✘
div_8bit	3	1	✘	5	1	✘	0	1	-	3	0	✘	3	1	✘	4	1	✘
div_16bit	4	0	✘	5	4	✔	1	2	✘	1	1	✘	0	0	-	0	0	-
JC_counter	5	5	✘	5	5	✘	3	3	✘	4	5	✘	5	5	✔	5	4	✔
right_shifter	4	4	✔	5	5	✔	0	2	-	3	3	✔	5	5	✔	5	5	✔
synchronizer	5	5	✔	4	4	✔	4	4	✔	5	5	✔	4	4	✔	5	5	✔
counter_12	5	5	✔	5	5	✔	2	4	✔	2	4	✔	5	5	✔	5	5	✔
freq_div	5	5	✔	5	5	✔	4	4	✔	4	4	✘	5	5	✔	5	3	✔
signal_gen	5	5	✔	5	5	✔	4	5	✘	4	4	✘	5	5	✘	5	5	✘
serial2parallel	4	4	✘	5	5	✔	4	4	✘	4	4	✘	5	3	✘	5	3	✘
parallel2serial	2	2	✘	5	5	✘	1	2	✘	3	4	✔	3	3	✘	3	2	✔
pulse_detect	4	4	✘	5	3	✘	4	3	✘	3	3	✘	5	5	✘	2	2	✘
edge_detect	5	5	✔	5	5	✔	4	5	✔	3	4	✔	4	2	✔	5	4	✔
FSM	5	4	✘	5	2	✘	4	4	✘	5	5	✘	4	4	✘	5	5	✘
width_8to16	4	3	✔	5	5	✔	4	1	✔	3	4	✘	5	5	✔	5	4	✔
traffic_light	4	0	✘	4	3	✔	5	2	✘	5	3	✘	4	0	✔	4	3	✔
calendar	5	5	✘	5	5	✔	2	1	✘	5	4	✔	1	0	✘	5	5	✘
RAM	4	0	✔	5	2	✔	5	5	✔	2	0	✔	3	0	✔	3	0	✔
asyn_fifo	0	0	-	3	2	✘	0	0	-	0	0	-	0	2	-	1	3	✘
ALU	2	0	-	5	4	-	2	2	✘	1	0	✘	2	1	✘	1	0	✘
PE	5	5	✔	5	5	✔	3	3	✘	3	5	✔	1	1	✔	5	5	✔
Success rate	89.7%	65.5%	11/29	100%	100%	19/29	86.2%	86.2%	7/29	93.1%	82.8%	8/29	86.2%	75.9%	12/29	96.6%	79.3%	14/29

with only 7 billion parameters could achieve such impressive accuracy despite its smaller size.

Furthermore, we validate the effectiveness of our proposed dataset and algorithm through an ablation study. The RTLCoder-Mistral-Direct and RTLCoder-DeepSeek-Direct are directly trained with the existing method mentioned in Figure 13(a). Using our training dataset, they can already significantly outperform the base model and even GPT-3.5 on part of these indexes. Then the RTLCoders trained with our proposed training scheme further outperform those using Direct training method on all benchmarks, indicating that our training method greatly further improves the model performance.

In addition, although the quantized model RTLCoder-DeepSeek-4bit shows a slight performance degradation compared to the original model, it is still superior to GPT-3.5 on the VerilogEval benchmark and comparable to it on RTLLM V1.1 with only 4GB size. Such RTLCoder-4bit can work on a simple laptop, allowing it to serve as a local assistant for engineers, addressing privacy concerns.

We also randomly selected 10K samples from the 27K training dataset to finetune the base models and obtained RTLCoder-Mistral-10k and RTLCoder-DeepSeek-10k respectively. Compared with the two models, RTLCoders trained on a 27K dataset are clearly superior on all metrics. Increasing the size of the training dataset and enhancing its diversity clearly further improves the model performance.

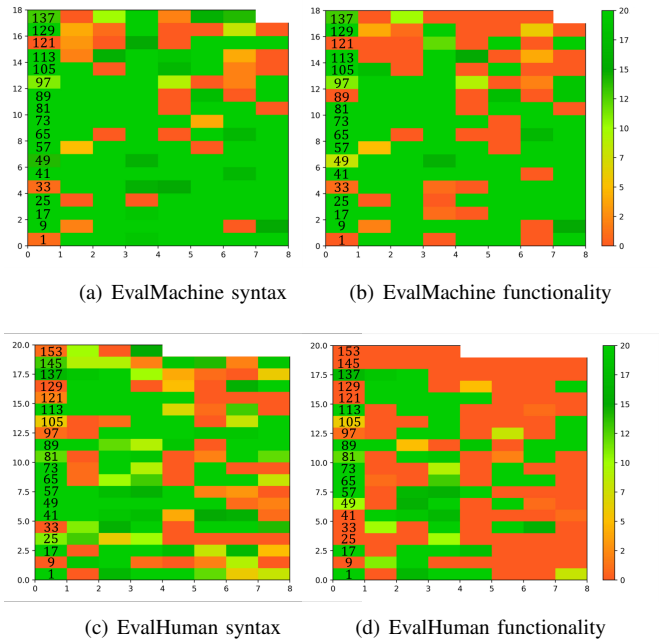
As for the pre-trained model selection, we can see that different base model also has a significant impact on the performance of the fine-tuned model. On one hand, RTLCoder-DeepSeek slightly outperforms RTLCoder-Mistral in accuracy on most benchmarks. This trend is consistent with the base model's relative accuracy (i.e., DeepSeek outperforms Mistral in most benchmarks). On the other hand, the inference speed of RTLCoder-Mistral is considerably faster than RTLCoder-DeepSeek, largely because of the Grouped Query Attention and Rolling Buffer KV Cache techniques used in Mistral.

E. Experiment Results in Detail

To further examine the performance in detail, for both benchmarks [12], [13], we report RTLCoder's performance on each individual design case in both syntax and functionality correctness.

We list the test results of RTLCoder-Mistral and available baseline models on the RTLLM V1.1 benchmark for each design task in Table IV. Given 5 trials of generation, here we counted the number of passed cases in terms of Syn-VCS, Syn-DC, and Functionality. As introduced, for both syntax and functionality, we count one success if any of the 5 trials pass the test. Generally, Syn-VCS is easier to pass than Syn-DC.

We further inspect the wrong answers in Table IV. We observed that the overall code structures of wrong answers from GPT-3.5, GPT-4, and RTLCoder-Mistral exhibit no obvious mistakes, despite the functionality incorrectness. In



(a) EvalMachine syntax (b) EvalMachine functionality

(c) EvalHuman syntax (d) EvalHuman functionality

Fig. 16: Detailed syntax and functionality results of RTLCoder-Mistral on VerilogEval Benchmark [13], reporting EvalMachine and EvalHuman separately. Each sub-figure has 8 columns, and thus cell at (i, j) represents the $((j-1)\times 8+i)^{\text{th}}$ task. The color of each cell indicates the count of correct cases among 20 trials. EvalMachine contains 143 tasks, so the last 1 cell is empty. EvalHuman contains 156 tasks, so the last 4 cells are empty.

comparison, the code generated by other open-source baselines occasionally contains obviously redundant content or deviates considerably from the given description. In terms of syntax, we observed that both GPT and RTLCoder-Mistral frequently assign 0 directly to two-dimensional arrays, resulting in syntax errors. Regarding functionality, we noticed that for more complex combinational logic circuits such as multi_pipe_4bit and multi_pipe_8bit, and sequential logic circuits like pulse_detect and FSM, some of the logical behaviors described in the instructions are not adequately captured by all LLM solutions, leading to functional errors.

The RTLCoder-Mistral's results on VerilogEval Benchmark are reported in Figure 16. Each cell in the image represents one design case, with color indicating the number of successful ones among all 20 trails. There are 8 columns in each image. The location of cell (i, j) represents the $((j-1)\times 8+i)^{\text{th}}$ design case in the provided description file. So we used white cells to fill the cells in the last row (18^{th} row for EvalMachine and 20^{th} row in the EvalHuman) that do not correspond to a design task.

During the process of generating text sequences, the model continuously repeats the behavior of predicting the next token. For all models in our experiment, we adopt the sampling method, which randomly selects the next token from the vocabulary dictionary based on the probability distribution. Here we further add an ablation study based on the beam search method. A beam of the top "beam size" sub-sequences with the highest generation probabilities is maintained and updated during the generation process. We conduct experiments using beam search method with a beam size 5 on RTLLM V1.1 for RTLCoder-Mistral and open source baselines. The results are shown in Table V. The accuracies of all methods drop after adopting beam search. RTLCoder-Mistral is still superior to all the open-source baselines with beam search.

TABLE V: Ablation study of different decoding methods in RTLLM V1.1 Benchmark [12]. The result of the sampling decoding method is adopted and reported in the Table III.

Model	Sampling decoding [used in experiment]			Beam search decoding [for ablation study]		
	Syn-VCS	Syn-DC	Func	Syn-VCS	Syn-DC	Func
Thakur et al. [14]	86.2	86.2	24.1	69.0	51.7	17.2
StarCoder[25]	93.1	82.8	27.6	58.6	58.6	17.2
RTLCoder-Mistral-4bit	86.2	75.9	41.4	75.9	65.5	31.0
RTLCoder-Mistral	96.6	79.3	48.3	75.9	72.4	37.9

	Verilog	Python	Cpp	Sh
Mistral-7B-v0.1	4.49	25.2	30.1	9.07
DeepSeek-Coder-6.7b-v1	30.2	66.8	63.9	36.3
RTL-Coder Mistral	36.7	25.3	21.4	1.99
RTL-Coder DeepSeek	41.6	66.7	63. 0	32.6

TABLE VI: The pass@1 results of the trained RTLCoder and two base models on different programming tasks

We also investigate the performance of the RTLCoder fine-tuned on the Verilog task compared to its pre-train models on other code-generation tasks. Table VI shows pass@1 results on Veilog tasks from VerilogEval-Human [13], python tasks from HumanEval+ [37], Cpp and Sh programming tasks from [38]. RTLCoder-DeepSeek shows a significant improvement over its base model in the Verilog task and performs similarly to the base model in the other three programming tasks. RTLCoder-Mistral, compared to its base model, also shows better performance in the Verilog task but experiences more performance degradation in the Cpp and Sh benchmarks. This observation indicates that different pre-train models when fine-tuned on the same specific task, exhibit varying degrees of forgetting when applied to other benchmarks. Overall, such a degradation in other programming tasks does not affect our target application (i.e., circuit RTL generation).

V. Limitation and Future Work

The results of GPT4 on VerilogEval-Human and RTLLM 1.1 are very impressive in Table III which indicates the significant superiority of GPT4 to all the other reported models. In our opinion, GPT-4, the most powerful large-scale model by OpenAI to date, outperforms existing open-source models for mainly three main reasons: 1) The size of the pre-training dataset used by GPT-4 is orders of magnitude larger than our adopted pre-trained models. The generation of a huge dataset that enables high-performance LLM consumes a significant amount of manpower and resources. 2) The GPT-4 model is much larger in scale than any existing open-source model. The scaling law [39] suggests that as the model parameters increase, the model's generalization and learning capabilities also improve. 3) The alignment technique such as RLHF [40]

employed by GPT-4 can enhance the quality of the model's outputs, reducing incorrect answers, which is also one of the motivations for us proposing the scoring-based training approach in our paper.

Given limited resources, we believe there are several key strategies for the open-source model to outperform GPT4 in specific tasks.

- The diversity and coverage of the dataset can be further improved. Due to limited manual efforts during dataset generation, the training set primarily relies on guiding GPT to create instructions. To further enhance the coverage of the dataset for Verilog design problems, manual inspection and creation for high-quality data sample generation can be conducted.
- The checking of functional correctness of the Verilog code should be further explored if possible, using either manpower or a reliable automated verification process. Although our fine-tuned models have a significant performance improvement and even outperform GPT-3.5, reliable filtering based on functionality checking can certainly further boost the model's generation ability.
- The automated functionality checking for hardware is challenging. A possible solution is to apply LLMs such as GPT to also help generate verification assertions based on instructions [41]. Then the code combined with assertions can be verified by verification platforms such as Cadence Jasper. However, the correctness of assertions, since they are also automatically generated by LLMs, is not guaranteed either. This may result in wrongly filtering out many correct samples.
- The scoring-based training scheme can be improved by applying automated functionality checking. This technique can hopefully enhance the model's instruction alignment capability.

VI. Conclusion

This work proposes a new LLM solution named RTLCoder for RTL code generation, achieving state-of-the-art performance in non-commercial solutions and outperforming GPT-3.5. We contribute a new data generation flow and a complete dataset with over 27 thousand instruction-answer samples, addressing the serious data availability problem in hardware-design-related tasks. Also, we contribute a new training scheme based on design quality scoring. It greatly boosts the model's performance. Importantly, RTLCoder has been fully open-sourced. RTLCoder's lightweight property and low hardware barrier allow anyone to easily replicate and further improve based on our existing solution. We expect more brilliant LLM-based solutions in this agile hardware design direction.

References

[1] OpenAI, "GPT-4 Technical Report," *arXiv preprint arXiv:2303.08774*, 2023.

[2] L. Chen, Y. Chen, Z. Chu, W. Fang, T.-Y. Ho, Y. Huang, S. Khan, M. Li, X. Li, Y. Liang *et al.*, "The dawn of ai-native eda: Promises and challenges of large circuit models," *arXiv preprint arXiv:2403.07257*, 2024.

[3] Z. He, H. Wu, X. Zhang, X. Yao, S. Zheng, H. Zheng, and B. Yu, "Chateda: A large language model powered autonomous agent for eda," in *MLCAD Workshop*, 2023.

[4] M. Liu, T.-D. Ene, R. Kirby, C. Cheng, N. Pinckney, R. Liang, J. Alben, H. Anand, S. Banerjee, I. Bayraktaroglu *et al.*, "Chipnemo: Domain-adapted llms for chip design," *arXiv preprint arXiv:2311.00176*, 2023.

[5] Y. Fu, Y. Zhang, Z. Yu, S. Li, Z. Ye, C. Li, C. Wan, and Y. Lin, "Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models," *arXiv preprint arXiv:2309.10730*, 2023.

[6] Z. Yan, Y. Qin, X. S. Hu, and Y. Shi, "On the viability of using llms for sw/hw co-design: An example in designing cim dnn accelerators," *arXiv preprint arXiv:2306.06923*, 2023.

[7] Z. Liang, J. Cheng, R. Yang, H. Ren, Z. Song, D. Wu, X. Qian, T. Li, and Y. Shi, "Unleashing the potential of llms for quantum computing: A study in quantum architecture design," *arXiv preprint arXiv:2307.08191*, 2023.

[8] R. Kande, H. Pearce, B. Tan, B. Dolan-Gavitt, S. Thakur, R. Karri, and J. Rajendran, "Llm-assisted generation of hardware assertions," *arXiv preprint arXiv:2306.14027*, 2023.

[9] B. Ahmad, S. Thakur, B. Tan, R. Karri, and H. Pearce, "Fixing hardware security bugs with large language models," *arXiv preprint arXiv:2302.01215*, 2023.

[10] K. Chang, Y. Wang, H. Ren, M. Wang, S. Liang, Y. Han, H. Li, and X. Li, "Chipgpt: How far are we from natural language hardware design," *arXiv preprint arXiv:2305.14019*, 2023.

[11] J. Blocklove, S. Garg, R. Karri, and H. Pearce, "Chip-chat: Challenges and opportunities in conversational hardware design," *arXiv preprint arXiv:2305.13243*, 2023.

[12] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "Rtllm: An open-source benchmark for design rtl generation with large language model," *arXiv preprint arXiv:2308.05345*, 2023.

[13] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "Verilogeval: Evaluating large language models for verilog code generation," *arXiv preprint arXiv:2309.07544*, 2023.

[14] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg, "Benchmarking large language models for automated verilog rtl code generation," in *DATE*, 2023.

[15] S. Thakur, J. Blocklove, H. Pearce, B. Tan, S. Garg, and R. Karri, "Autochip: Automating hdl generation using llm feedback," *arXiv preprint arXiv:2311.04887*, 2023.

[16] M. Nair, R. Sadhukhan *et al.*, "Generating secure hardware using chatgpt resistant to cwes," *Cryptology ePrint Archive*, 2023.

[17] Y. Zhang, Z. Yu, Y. Fu, C. Wan *et al.*, "Mg-verilog: Multi-grained dataset towards enhanced llm-assisted verilog generation," *arXiv preprint arXiv:2407.01910*, 2024.

[18] E. Goh, M. Xiang, I. Wey, T. H. Teo *et al.*, "From english to asic: Hardware implementation with large language model," *arXiv preprint arXiv:2403.07039*, 2024.

[19] S. Liu, Y. Lu, W. Fang, M. Li, and Z. Xie, "Openllm-rtl: Open dataset and benchmark for llm-aided design rtl generation," in *2024 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE/ACM, 2024.

[20] M. Li, W. Fang, Q. Zhang, and Z. Xie, "Specllm: Exploring generation and review of vlsi design specification with large language model," *arXiv preprint arXiv:2401.13266*, 2024.

[21] K. Chang, K. Wang, N. Yang, Y. Wang, D. Jin, W. Zhu, Z. Chen, C. Li, H. Yan, Y. Zhou *et al.*, "Data is all you need: Finetuning llms for chip design via an automated design-data augmentation framework," *arXiv preprint arXiv:2403.11202*, 2024.

[22] Z. Pei, H.-L. Zhen, M. Yuan, Y. Huang, and B. Yu, "Betterv: Controlled verilog generation with discriminative guidance," *arXiv preprint arXiv:2402.03375*, 2024.

[23] M. Rapp, H. Amrouch, Y. Lin, B. Yu, D. Z. Pan, M. Wolf, and J. Henkel, "Mlcad: A survey of research in machine learning for cad keynote paper," *IEEE TCAD*, 2021.

[24] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "Codegen2: Lessons for training llms on programming and natural languages," *arXiv preprint arXiv:2305.02309*, 2023.

[25] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[26] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023.

[27] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Source code is all you need," *arXiv preprint arXiv:2312.02120*, 2023.

[28] C. Shaib, J. Barrow, J. Sun, A. F. Siu, B. C. Wallace, and A. Nenkova, "Standardizing the measurement of text diversity: A tool and a comparative analysis of scores," *arXiv preprint arXiv:2403.00553*, 2024.

[29] Y. Liu, P. Liu, D. Radev, and G. Neubig, "Brio: Bringing order to abstractive summarization," *arXiv preprint arXiv:2203.16804*, 2022.

[30] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer, "Scheduled sampling for sequence prediction with recurrent neural networks," in *NeurIPs*, 2015.

[31] Synopsys, "VCS® functional verification solution," https://www.synopsys.com/verification/simulation/vcs.html, 2021.

[32] ——, "Design Compiler® RTL Synthesis," https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-nxt.html, 2021.

[33] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[34] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *KDD*, 2020.

[35] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *Applied Reconfigurable Computing*, 2015.

[36] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "GPTQ: Accurate post-training compression for generative pretrained transformers," *arXiv preprint arXiv:2210.17323*, 2022.

[37] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[38] L. Ben Allal, N. Muennighoff, L. Kumar Umapathi, B. Lipkin, and L. von Werra, "A framework for the evaluation of code generation models," https://github.com/bigcode-project/bigcode-evaluation-harness, 2022.

[39] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[40] Y. Bai, A. Jones, K. Ndousse, A. Askell, A. Chen, N. DasSarma, D. Drain, S. Fort, D. Ganguli, T. Henighan *et al.*, "Training a helpful and harmless assistant with reinforcement learning from human feedback," *arXiv preprint arXiv:2204.05862*, 2022.

[41] W. Fang, M. Li, M. Li, Z. Yan, S. Liu, H. Zhang, and Z. Xie, "Assertllm: Generating and evaluating hardware verification assertions from design specifications via multi-llms," *arXiv preprint arXiv:2402.00386*, 2024.

Shang Liu received the B.E. degree in Automation Science and Electrical Engineering from Beihang University, Beijing, China, in 2023. He is currently pursuing the Ph.D. degree with the Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology, Hong Kong. His research interests include agile VLSI design methodologies and Artificial Intelligence.

Wenji Fang is currently a Ph.D. student with the Department of Electronic and Computer Engineering at the Hong Kong University of Science and Technology. He received his M.Phil. degree in Microelectronics from the Hong Kong University of Science and Technology (Guangzhou) in 2024, and his B.Eng. degree from Nanjing University of Aeronautics and Astronautics in 2021. His research interests include Electronic Design Automation (EDA) and VLSI design verification.

Yao Lu received the B.E. degree from the School of Electronic Science and Engineering, Southeast University, Nanjing, China, in 2020, and the master degree from the School of Microelectronics, Fudan University, Shanghai, China, in 2023. She is currently pursuing the Ph.D. degree with the Department of Electronic and Computer Engineering, The Hong Kong University of Science and Technology, Hong Kong. Her current research interests focus on machine learning applications in EDA.

Jing Wang received the B.S. degree in Electrical Information Engineering from Peking University, Beijing, China, in 2022, and the master degree in Artificial Intelligence from the Department of Statistics and Actuarial Science, The University of Hong Kong, China, in 2023. He is currently pursuing the Ph.D. degree with the Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology, Hong Kong. His research interests include agile VLSI design methodologies and Artificial Intelligence.

Qijun Zhang received the B.Eng. degree from Tongji University, Shanghai, China, in 2022. He is currently a Ph.D. student in the Department of Electronic and Computer Engineering (ECE) at the Hong Kong University of Science and Technology (HKUST). His research interests include Computer Architecture and Electronics Design Automation.

Hongce Zhang (Member, IEEE) received the B.S. degree in microelectronics from Shanghai Jiao Tong University, Shanghai, China, in 2015, and the Ph.D. degree from the Electrical and Computer Engineering Department of Princeton University, NJ, USA, in 2021.

He is currently an Assistant Professor with the Microelectronics Thrust, Function Hub of Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China, and is also affiliated with the Electronic and Computer Engineering Department of the Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong SAR. His research interests include formal verification and hardware model checking.

Zhiyao Xie is an Assistant Professor of the Department of Electronic and Computer Engineering (ECE) at the Hong Kong University of Science and Technology (HKUST). Zhiyao received his Ph.D. degree from Duke University in 2022 and B.Eng. from City University of Hong Kong in 2017. His research interests include machine learning algorithms for EDA and VLSI design. He has received multiple prestigious awards, including the IEEE/ACM MICRO 2021 Best Paper Award, ACM SIGDA SRF Best Research Poster Award 2022, ASP-DAC 2023 Best Paper Award, ACM Outstanding Dissertation Award in EDA 2023, EDAA Outstanding Dissertation Award 2023, and the 2023 Early Career Award from Hong Kong Research Grants Council (RGC).