RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model

Yao Lu*, Shang Liu*, Qijun Zhang, Zhiyao Xie†

Hong Kong University of Science and Technology

{yludf, sliudx, qzhangcs}@connect.ust.hk, eezhiyao@ust.hk

***Abstract*—Inspired by the recent success of large language models (LLMs) like ChatGPT, researchers start to explore the adoption of LLMs for agile hardware design, such as generating design RTL based on natural-language instructions. However, in existing works, their target designs are all relatively simple and in a small scale, and proposed by the authors themselves, making a fair comparison among different LLM solutions challenging. In addition, many prior works only focus on the design correctness, without evaluating the design qualities of generated design RTL. In this work, we propose an open-source benchmark named RTLLM, for generating design RTL with natural language instructions. To systematically evaluate the auto-generated design RTL, we summarized three progressive goals, named syntax goal, functionality goal, and design quality goal. This benchmark can automatically provide a quantitative evaluation of any given LLM-based solution. Furthermore, we propose an easy-to-use yet surprisingly effective prompt engineering technique named self-planning, which proves to significantly boost the performance of GPT-3.5 in our proposed benchmark.**

I. Introduction

In recent years, machine learning (ML) for EDA, or named ML for circuit/hardware design, has become a trending topic [1], [2]. By learning from prior design solutions, ML models can perform fast circuit quality evaluations or even optimizations. Most existing ML for EDA solutions can be categorized into two main types, *predictive* models and *generative* models. *Predictive* ML models are trained to provide early predictions on circuit qualities. In contrast, *generative* models are supposed to generate design solutions directly, which is more useful while challenging.

Recently, natural language processing (NLP) researchers realize that when the scale of model parameters exceeds a certain level, these enlarged language models can achieve a significant performance improvement over small-scale language models like BERT [3]. The most remarkable progress of large language models (LLMs) is reflected by the popularity of commercial products GPT-3.5 and GPT-4 [4].

Inspired by this recent success of LLMs, researchers start to explore the adoption of LLMs for agile hardware design. One intuitive and promising direction is to generate the target design RTL directly with natural language instructions. This new paradigm is expected significantly reduce the barrier of hardware design and improve the design productivity. Such natural-language-based design method may revolutionize existing design methods based on hardware description language (HDL), including Verilog, VHDL, Chisel, C++/SystemC with high-level synthesis (HLS), etc.

There have been some most recent explorations [5]–[7] on this topic. Thakur et al. proposes to fine-tune open-source LLMs like CodeGen [8] to generate Verilog code for target designs [5]. Then Chip-Chat [6] further discusses the challenges and opportunities in hardware design based on LLMs. It indicates an obviously superior performance of ChatGPT over open-sourced LLMs. Another work Chip-GPT [7] study a similar task, proposing to perform RTL design based on ChatGPT. We expect more explorations in natural-language based hardware design based on LLMs in the future.

Works	Num of Designs	Num of HDL Lines	Num of Cells in Netlist[1]
		{Medium, Mean, Max, Total}	
Thakur et al. [5]	17	{16, 19, 48, 0.3K}	{9.5, 45, 335, 0.7K}
Chip-Chat [6]	8	{42, 42, 72, 0.3K}	{37, 44, 110, 0.4K}
Chip-GPT [7]	8	Not released to public	
RTLLM	30	{52, 86, 518, 2.5K}	{121, 408, 2435, 11.8K}

TABLE I: The statistics of designs evaluated in prior works [5]–[7] and in RTLLM. We quantify the design complexity with the number of HDL lines in each design RTL, and the design scale with the number of cells in the post-synthesis netlist. RTLLM is an obviously more comprehensive benchmark compared with other datasets.

However, in these existing works [5]–[7], their target designs are all relatively simple and in a small circuit scale, as summarized in Table I. As a result, the performance and scalability of LLM solutions are not thoroughly evaluated. In addition, these small designs are proposed by the authors themselves, making a fair comparison among different LLM solutions challenging. More importantly, even for the same design, the natural language description from different human designers can be largely different. Using a unified natural language design description is necessary in fair LLM evaluations.

In this work, we propose a comprehensive open-source benchmark for design RTL generation with natural language. It is named RTLLM. 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 consists of 30 designs with a wide coverage of design complexities and scales. To systematically and quantitatively evaluate the quality of each auto-generated design RTL, we summarize three progressive goals, named syntax goal, functionality goal, and design quality goal. Based on our provided design automation scripts, the benchmark can automatically evaluate any given LLM solution with respect to all three goals. More importantly, RTLLM provides ground-truth design RTLs crafted by human designers, providing a standard baseline to evaluate the design quality goal.

Our contributions in this work are summarized below.

- We propose a comprehensive open-source benchmark[2]

*Equal Contribution

†Corresponding Author

[1]Excluding the pseudo RAM design implemented with a large matrix of D flip-flop, because its complexity completely depends on the number of wordlines and bitlines as parameters. Also, realistic SRAMs should consist of SRAM cells instead of flip-flops and be generated by memory compilers.

[2]It will be open-sourced in https://github.com/hkust-zhiyao/RTLLM

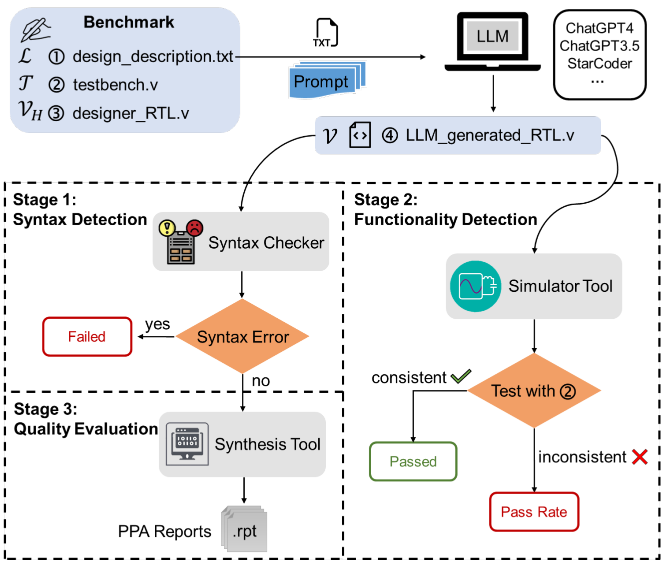

Fig. 1: The workflow of adopting RTLLM for completely automated design RTL generation and evaluation. The user only need to provide their LLM as input. It evaluates whether each generated design satisfies the syntax goal, functionality goal, and quality goal.

dedicated to the automatic design RTL generation with natural languages. Compared with released dataset in recent works, our benchmark includes many more designs, also with higher design scale and complexity.

- We systematically evaluated state-of-the-art commercial and academic solutions with our benchmark. In addition to assessing syntax and functionality, we also evaluate the design PPAs of the generated RTL by comparing it with our human-crafted designs provided in RTLLM.
- Besides providing the benchmark, we also propose an innovative new prompt engineering technique named self-planning, without requiring any human interference. Combining self-planning and GPT-3.5 can well outperform the performance of GPT-3.5 and get close to GPT-4's state-of-the-art performance.

II. Problem Formulation

In this section, we provide a general formulation of the RTL generation task based on natural language instructions. Given a natural language description of desired design functionality named $\mathcal{L}$, the target is to develop an ML model F to generate the RTL of this design $\mathcal{V}$, with $\mathcal{V} = F(\mathcal{L})$. To achieve this goal, currently the model F is based on LLMs.

However, the generation directly based on the LLM F may not be successful. Therefore, prompt engineering techniques P can be applied to revise the design functionality description in natural language $\mathcal{L}$, generating $\mathcal{L}_P = P(\mathcal{L})$, which is feed into LLMs F as input. In addition, this LLM output may be further manually revised by human engineers H, making the ultimate output $\mathcal{V} = H(F(\mathcal{L}_P))$.

III. RTLLM: An RTL Generation Benchmark

A. Evaluation Metrics for RTL Generation Task

To systematically evaluate the generated design RTL $\mathcal{V}$, we summarize three progressive goals of the $\mathcal{V}$. Our benchmark enables automatic evaluation of these three goals as three metrics. These goals are summarized as below.

We name the first and the most fundamental goal as the **syntax goal**. It means the syntax of generated RTL design $\mathcal{V}$ should at least be correct. It can be verified by checking whether the design can be correctly synthesized into netlist by synthesis tools [9] without syntax errors.

After ensuring the syntax correctness, we name the second goal as the **functionality goal**. It means the functionality of generated RTL design $\mathcal{V}$ should be exactly the same as designers' expectation. It can be verified by checking whether the generated design passes all test cases in a comprehensive testbench. Of course, exhausting all possible test cases will make the testbench file extremely cumbersome. Our benchmark only samples a reasonable number of test cases. Passing all test cases does not necessarily mean the functionality is 100% correct.

If the generated design RTL $\mathcal{V}$ proves correct in both syntax and functionality, the design can be viewed as successful. But in order to make $\mathcal{V}$ practically useful, its design qualities including performance, power, area (PPA) should also be desirable. We name this goal as **quality goal**. It can be verified by measuring the PPA values after the synthesis and layout of generated $\mathcal{V}$. This quality goal is not explicitly evaluated in prior works [5]–[7].

B. An Overview of the Design Generation Benchmark

RTLLM collects 30 common designs with various design scales and complexities. For each design, the benchmark provides the following information in three separate files.

- **Description** (*design_description.txt*) denoted as $\mathcal{L}$: A natural language description of the target design's functionality. The criteria is, a human designer can write a correct design RTL $\mathcal{V}$ after reading the description $\mathcal{L}$. This description $\mathcal{L}$ also includes an explicit indication of the module name, all input and output (I/O) signals with signal name and width. These pre-defined module and I/O signal information enables automatic functionality verification with our provided testbench.
- **Testbench** (*testbench.v*) denoted as $\mathcal{T}$: A testbench with multiple test cases, each with input values and correct output values. The testbench corresponds to the pre-defined module name and I/O signals in $\mathcal{L}$. It can be applied to verify the correctness of design functionality.
- **Correct Design** (*designer_RTL.v*) denoted as $\mathcal{V}_H$: A reference design Verilog hand-crafted by human designers. By comparing with this reference design $\mathcal{V}_H$, we can quantitatively evaluate design qualities of the automatically generated design $\mathcal{V}$. Also, these correct designs have all passed our proposed testbenches.

Fig. 1 shows a complete workflow of RTL generation and evaluation using this benchmark, including three straightforward stages. In stage ❶, users feed each natural language description $\mathcal{L}$ into their target LLM F, generating the design RTL $\mathcal{V} = F(\mathcal{L})$. If an LLM solution requires additional prompt techniques P, it will switch the natural language description $\mathcal{L}$ to actual input prompts $\mathcal{L}_P$, with the output design RTL being $\mathcal{V} = F(\mathcal{L}_P)$. If necessary, additional human engineers' efforts can also be introduced, generating $\mathcal{V} = H(F(\mathcal{L}_P))$.

TABLE II: Benchmark Descriptions and Scales

	Design	**Description**	**Lines of Code**	**Circuit Scale (Cells)**
Arithmetic	accu	Accumulates 8-bit data and output after 4 inputs	64	195
	adder_8bit	An 8-bit adder	26	58
	adder_16bit	A 16-bit adder implemented with full adders	137	130
	adder_32bit	A 32-bit carry lookahead adder	181	312
	adder_64bit	A 64-bit ripple carry adder based on 4-stage pipeline	197	1340
	multi_8bit	An 8-bit booth-4 multiplier	84	34
	multi_16bit	An 16-bit multiplier based on shifting and adding operation	65	817
	multi_pipe_4bit	A 4-bit unsigned number pipeline multiplier	43	120
	multi_pipe_8bit	An 8-bit unsigned number pipeline multiplier	92	578
	div_8bit	An 8-bit radix-2 divider	72	94
	div_16bit	A 16-bit divider based on substraction operation	45	1855
Logic	JC_counter	4-bit Johnson counter with specific cyclic state sequence	22	134
	right_shifter	Right shifter with 8-bit delay	17	466
	mux	Multi-bit mux synchronizer	46	19
	counter_12	Counter module counts from 0 to 12	37	38
	freq_div	Frequency divider for 100M input clock, outputs 50MHz, 10MHz, 1MHz	51	64
	signal_generator	Signal generator produces square, sawtooth, and triangular waveforms	52	135
	serial2parallel	1-bit serial input and output data after receiving 6 inputs	62	66
	parallel2serial	Convert 4 input bits to 1 output bit	41	24
	pulse_detect	Extract pulse signal from the fast clock and create a new one in the slow clock	38	6
	edge_detect	Detect rising and falling edges of changing 1-bit signal	39	7
	FSM	FSM detection circuit for specific input	77	24
	width_8to16	First 8-bit data placed in higher 8-bits of the 16-bit output	50	117
	traffic_light	Traffic light system with three colors and pedestrian button	106	117
	calendar	Perpetual calendar with seconds, minutes, and hours	37	121
	RAM	8x4 bits true dual-port RAM	50	1834
	asyn_fifo	An asynchronous FIFO 16×8 bits	149	686
	ALU	An ALU for 32bit MIPS-ISA CPU	111	2435
	PE	A Multiplying Accumulator for 32bit integer	27	1439
	risc_cpu	Simplified RISC_CPU with clock generator, instruction register, accumulator, arithmetic logic unit, data controller, state controller, etc.	518	407

In stage ❷, the framework will test the functionality of generated design RTL $\mathcal{V}$ using our provided testbench $\mathcal{T}$. In stage ❸, the generated design RTL $\mathcal{V}$ is synthesized into netlist to analyze the design qualities in terms of PPA values. They will be compared with the design qualities of the provided reference designs $\mathcal{V}_H$. This whole process is automated.

C. Detailed Inspection of the Benchmark

Table II shows the detailed description of all 30 designs in our provided benchmark. It provides 30 common digital designs, with 11 arithmetic designs and the other 19 logic designs implementing various functionalities. All reference designs from human engineers $\mathcal{V}_H$ are coded in Verilog. To give more information about the design complexity and design scale, Table II also provides the number of lines in the HDL code and the number of cells in synthesized gate-level netlist. Intuitively, a design with more HDL code tends to be more complex to implement by human designers, and a design with more cells in netlist is naturally larger in the design area and power consumption. These statistics are collected based on the correct reference designs $\mathcal{V}_H$ from designers.

For the 11 arithmetic designs in RTLLM, they cover the most common design types including accumulators, adders, multipliers, and dividers, with all common bit widths from 4 bits to 64 bits. For each design type, we cover different implementation requirements. Take the adder as an example, the benchmark includes the basic version without any requirement (i.e., adder_8bit), the adder implemented with 1-bit full adders (i.e., add_16bit), the lookahead adder (i.e., adder_32bit), and the ripple adder with pipelines (i.e., adder_64bit). Both design complexity and scale increase progressively in these adders.

The 19 logic designs in RTLLM include designs with more variations in their target functionalities. It includes simpler designs like counters (i.e., counter_12) and finite state machines (i.e., FSM), and more complex designs like the simplified RISC CPU design (i.e., risc_cpu) and a processing element (i.e., PE) performing multiply–accumulate operations.

In summary, RTLLM proposes 30 common designs with rich diversities in their functionalities, implementation requirements, design complexities, and design scales. The overall scale of RTLLM is significantly larger than the data released in prior works [5], [6], as already summarized in Table I.

IV. Self-Planning Technique

To further explore the capabilities of LLMs, in this work, we also propose a highly effective prompt engineering technique named self-planning. It is extremely easy to use and works surprisingly well. Instead of directly generating design RTL in one query, self-planning decomposese this enquiry into a two-step process, without requiring any extra efforts from human users or any existing design data.

The first step requests the LLM to *plan* how to write the target design $\mathcal{V}$. Specifically, the model is required to

```
1 #Implement the design of unsigned 16bit multiplier based on shifting and adding opration.
2 module multi_16bit(
3 // ...I/O details omitted...
4 );
5 #Please act as a professional verilog designer, try to understand the requirements above and give reasoning steps in natural language to achieve it.
6 #In addition, try to give advice to avoid syntax error.
```

Code 1: Part of the input of the first step in self-planning. Besides the design description with the definition of module I/O, it requests the LLM to first generate the reasoning steps (line 5) and advices to avoid errors (line 6).

output natural language-level analysis and reasoning steps of the target task. Additionally, considering that language models sometimes overlook the syntax requirements in Verilog generation, we also require the model to provide the syntax errors it needs to avoid during the code generation process. The LLM outputs are collected for the second step.

Code. 1 shows a partial input example in the first step of self-planning when generating the multi_16bit design. Lines 5 and 6 at the end give the planning instructions. Part of the LLM output is shown in Code. 2. As requested, the output in Code. 2 includes both a detailed plan with reasoning steps, and advice to avoid potential syntax errors.

In the second step, both the original design description $\mathcal{L}$ and the *plan* and *advice* collected in the first step will be fed into the LLM to generate the final design RTL $\mathcal{V}$. Compared with directly asking for the result, the design RTL generated by this two-step self-planning is obviously less error-prone.

Code 3 and 4 present the wrong RTL designs generated by GPT-3.5 without using self-planning, targeting generating multi_16bit and adder_32bit respectively. After using self-planning with GPT-3.5, both the syntax error in Code 3 and the functionality error in Code 4 are solved, generating correct design RTL for both designs.

Our proposed self-planning prompt engineering technique is actually similar to how we human beings solve a challenging task, like taking an exam or writing a complex algorithm. When we make a good plan ourselves before actually starting, we tend to perform better. Such similarity may imply certain humanlike "intelligence" of existing LLMs and can inspire further prompt engineering techniques in the future.

```
The task is to implement ... The modules has an ...
To achieve this, we will use a state machine to control the multiplication process. The steps involved are as follows:
IDLE: The initial state where the module waits for the start signal to be asserted.
MULTIPLY: The state where the multiplication operation is performed.
FINISH: The final state indicating the multiplication operation is complete.
Create the necessary registers to hold the state, the inputs (ain and bin), and the intermediate results during the multiplication process.
// ...details omitted...
Potential Syntax Error Advice:
Ensure that all the signals ... are declared ...
Make sure to use blocking ... non-blocking ...
Ensure that you use the correct parameter size for states. Since we have three states, we need at least two bits to represent them ...
Check for any typos or misspellings ...
```

Code 2: Part of the output of the first step from self-planning.

```
module multi_16bit(
// ...I/O details omitted...
);
// ...details omitted...
always @(posedge clk or negedge rst_n) begin
    integer i;
    reg [15:0] a_shifted;
    for (i = 0; i < 16; i = i + 1) begin
        // ......
    end
end
// ......
```

Code 3: A wrong design RTL of multi_16bit, generated by GPT-3.5 without self-planning. There are syntax errors in this design. It wrongly defines new variables (i.e., **i**, **a_shifted**) in the always block. After adopting self-planning, GPT-3.5 can generate correct design, likely benefiting from the syntax error advice from the first step.

V. Experimental Results

A. Experiment Setup

Given a design RTL, the design quality can be evaluated using synthesis tools. We perform logic synthesis with Synopsys Design Compiler® [9], using the advanced 'compile_ultra' synthesis option. We set the frequency to be extremely large to ensure a negative slack in all designs for a easier timing comparison. For functionality verification, the RTL simulation is performed with Synopsys VCS®.

In the experiment, we evaluated five LLMs with our proposed RTL generation benchmark:

1) GPT-3.5: the free commercial solution.
2) GPT-4: the state-of-the-art commercial solution.
3) Thakur et al. [5]: an academic model with 16 billion parameters developed by fine-tuning the CodeGen model [11] with Verilog data.
4) StarCoder [10]: a recent general academic model with 15 billion parameters for code generation, without being fine-tuned for Verilog.
5) GPT-3.5 + self-planning: adopting our proposed self-planning technique when using GPT-3.5.

Since there can be randomness in many LLM's outputs, for each test design in RTLLM, we query each LLM five times in five parallel sessions, with exactly the same description $\mathcal{L}$,

```
module adder_32bit(
// ...I/O details omitted...
);
// ...details omitted...
    for (i = 1; i <= 32; i = i + 1) begin : cla_block
        wire P, G, carry;
        //......
        assign carry = (i == 1) ? 1'b0 : (P & {carry, G});
        assign S[i] = A[i] ^ B[i] ^ {carry, G};
        //......
    end
end
```

Code 4: A wrong design RTL of adder_32bit, generated by GPT-3.5 without self-planning. Despite correct syntax, the functionality of this design is wrong, epecially reflected in its usage of the **carry** variable. After adopting self-planning, GPT-3.5 can generate correct design, likely benefiting from the reasoning steps from the first step.

TABLE III: The Syntax and Functionality Correctness Verification for Different LLMs

Design	**GPT-3.5**		**GPT-4**		**Thakur et al. [5]**		**StarCoder [10]**		**GPT-3.5 + Self-planning**	
	Syntax	Func.	Syntax	Func.	Syntax	Func.	Syntax	Func.	Syntax	Func.
accu	4	✔	5	✔	0	-	0	-	4	✔
adder_8bit	4	✔	5	✔	0	-	0	-	4	✔
adder_16bit	5	✘	5	✔	5	✔	0	-	5	✔
adder_32bit	5	✘	5	✘	0	-	0	-	5	✔
adder_64bit	2	✘	3	✘	0	-	0	-	4	✘
multi_8bit	3	✘	4	✘	0	-	0	-	5	✘
multi_16bit	0	-	5	✔	5	✔	0	-	2	✔
multi_pipe_4bit	0	-	2	✔	0	-	5	✔	1	✘
multi_pipe_8bit	0	-	4	✘	0	-	5	✔	3	✘
div_8bit	0	-	0	-	0	-	0	-	4	✘
div_16bit	0	-	5	✘	0	-	0	-	0	-
JC_counter	5	✔	5	✔	5	✘	5	✘	4	✔
right_shifter	5	✔	5	✔	5	✔	0	-	5	✔
mux	0	-	4	✔	5	✔	0	-	4	✘
counter_12	5	✔	5	✔	5	✘	5	✘	4	✔
freq_div	5	✘	5	✘	5	✘	0	-	5	✘
signal_generator	5	✔	5	✔	0	-	5	✘	5	✔
serial2parallel	5	✔	5	✔	0	-	5	✘	5	✔
parallel2serial	5	✘	4	✘	0	-	0	-	3	✘
pulse_detect	1	✘	5	✘	5	✘	0	-	5	✘
edge_detect	5	✔	5	✔	5	✘	0	-	5	✔
FSM	5	✘	5	✘	5	✘	0	-	5	✘
width_8to16	5	✔	5	✔	0	-	5	✔	5	✔
traffic_light	5	✘	5	✔	0	-	0	-	5	✘
calendar	0	-	5	✘	0	-	0	-	5	✔
RAM	0	-	0	-	5	✔	5	✔	0	-
asyn_fifo	0	-	0	-	0	-	0	-	2	✘
ALU	0	-	5	✘	0	-	0	-	0	-
PE	4	✔	5	✔	5	✘	5	✔	5	✔
risc_cpu	0	-	0	-	0	-	0	-	0	-
Success rate	55%	10/30	81%	15/30	40%	5/30	27%	5/30	73%	14/30

then collect all five outputs $\mathcal{V}$, which may be different from each other. In our experiment results, we will evaluate the correctness of all five outputs for each test case. There is *no* extra fixing of any incorrect output by human engineers or another round of query to LLMs.

B. RTL Generation Correctness

Table III summarizes the quantitative evaluation of both syntax and functionality correctness of all five evaluated LLMs using RTLLM. The syntax part counts the number of generated design RTLs $\mathcal{V}$ with correct syntax, out of the five trials. Then the functionality part (i.e., Func.) will count a success ✔ as long as there is one generated RTL successfully passing the testbench $\mathcal{T}$, out of the ones already with correct syntax.

According to Table III, GPT-4, the state-of-the-art commercial LLM, achieves the highest performance with 81% correct syntax and $15/30$ correct functionalities. In comparison, the GPT-3.5 alone degrades to 55% correct syntax and $10/30$ correct functionalities. After using our self-planning together with GPT-3.5, the performance rise back to 73% and $14/30$, which is close to the GPT-4's performance. It clearly validates the effectiveness of the self-planning technique.

In comparison, the academic LLMs perform significantly worse, with 40% syntax for Thakur et al. [5] and 27% for StarCoder [10], both with $5/30$ functionality correctness.

As demonstrated in this design correctness example, using our proposed RTLLM, we can automatically evaluate the performance of all LLMs in design RTL generation. In summary, the performance rank is GPT-4 > GPT-3.5 + self-planning > GPT-3.5 > Thakur et al. [5] >= StarCoder [10].

C. RTL Generation Quality

After evaluating design correctness, our RTLLM further supports evaluating the design qualities in power, timing, and area. Table IV summarizes the design qualities of generated design RTL from different LLMs[3]. These quality values are measured on each post-synthesis netlist. We report the worst negative slack (WNS) as the timing metric. It also presents the qualities of our designer-generated reference design $\mathcal{V}_H$ in RTLLM. All these reference designs are functionally correct.

For each generated design RTL $\mathcal{V}$, as long as it can be correct in syntax, we can perform the logic synthesis and report its design qualities in Table IV. We then mark the design RTLs with correct syntax but wrong functionality (i.e., fail to pass testbench) in Table IV as red color. Those unsynthesizable designs with wrong syntax are left blank in Table IV.

For each design from RTLLM, we mark the generated design with the best power, performance, and area among all candidates in green color. Then we count the number of best qualities achieved by each LLM method. Of course, only designs that are both syntax and functionality correct are eligible for this comparison and can be colored green.

According to the last row of Table IV, the GPT-4 achieves the highest number of best qualities. GPT-3.5 + self-planning

[3]The worst LLM StarCoder is not presented due to space limitation.

TABLE IV: The Design Qualities of Gate-Level Netlist, Synthesized with Design Compiler

Design	Designer Reference ($\mathcal{V}_H$)			ChatGPT-3.5			ChatGPT-4			Thakur et al. [5]			GPT-3.5 + Self-planning		
	Area (μm^2)	Power (μW)	Timing (ns)	Area (μm^2)	Power (μW)	Timing (ns)	Area (μm^2)	Power (μW)	Timing (ns)	Area (μm^2)	Power (μW)	Timing (ns)	Area (μm^2)	Power (μW)	Timing (ns)
accu	239	19K	-0.42	298	24K	-0.43	304	21K	-0.39	-	-	-	231	18K	-0.37
adder_8bit	65	34	-0.62	38	14	-0.14	15	5.8	-0.12	-	-	-	74	42	-0.63
adder_16bit	128	68	-1.21	157	91.0	-0.33	126	68	-1.19	189	106	-0.31	163	94	-0.33
adder_32bit	571	298	-0.72	58	17	-0.04	65	26	-0.13	-	-	-	337	199	-0.43
adder_64bit	2.9K	296K	-0.48	2.5K	242K	-0.60	2.4K	187K	-0.48	-	-	-	2.3K	220K	-0.32
multi_8bit	52	6.1K	-0.08	640	45K	-0.43	494	33K	-0.49	-	-	-	259	23K	-0.27
multi_16bit	749	75K	-0.91	-	-	-	531	79K	-0.50	7.5K	384K	-1.76	-	-	-
multi_pipe_4bit	198	19K	-0.34	-	-	-	193	22K	-0.33	-	-	-	146	17K	-0.30
multi_pipe_8bit	961	78K	-0.65	-	-	-	1.1K	80K	-0.99	-	-	-	443	42K	-0.14
div_8bit	158	8.4K	-0.38	-	-	-	-	-	-	-	-	-	-	-	-
div_16bit	1.8K	2.4K	-4.20	-	-	-	1.5K	1.8K	-4.84	-	-	-	-	-	-
JC_counter	380	45K	-0.13	380	45K	-0.13	42	4.7K	-0.26	29	4.6K	-0.23	195	21K	-0.22
right_shifter	42	4.2	-0.14	40	3.8K	-0.12	46	5.7K	-0.13	40	3.8K	-0.12	40	3.8K	-0.12
mux	68	6.5	-0.08	-	-	-	90	9.5	-0.08	64	13	-0.08	144	14	-0.08
counter_12	49	4.3K	-0.31	79.0	8.0K	-0.25	46	4.4K	-0.26	35	4.0K	-0.24	76	8.4K	-0.26
freq_div	124	16K	-0.29	911	66K	-0.45	118	16K	-0.32	226	16K	-0.4	667	53K	-0.41
signal_generator	178	14K	-0.36	72	9.2K	-0.23	98	11K	-0.26	-	-	-	101	11K	-0.27
serial2parallel	135	13K	-0.29	168	16K	-0.30	100	9.8K	-0.28	-	-	-	155	14K	-0.33
parallel2serial	55	8.6K	-0.23	35	6.2K	-0.21	20	3.8K	-0.19	-	-	-	1.06	0	0
pulse_detect	25	2.8	-0.13	42	2.8	-0.12	40	4.3	-0.08	25	2.8	-0.12	28	3.4	-0.08
edge_detect	19	2.6K	-0.14	24	3.3K	-0.16	19	2.6K	-0.14	1.06	0	0	19	2.6K	-0.14
FSM	44	3.5K	-0.18	26	2.7K	-0.21	34	2.7K	-0.25	27	2.7K	-0.24	45	4.1K	-0.2
width_8to16	219	23K	-0.26	214	21K	-0.20	219	23K	-0.26	-	-	-	144	14K	0.24
traffic_light	178	18K	-0.35	147	14K	-0.34	138	11K	-0.38	-	-	-	-	-	-
calendar	199	16K	-0.36	-	-	-	460	31K	-0.51	-	-	-	227	16K	-0.37
RAM	3.5K	248K	-0.35	-	-	-	-	-	-	353	27K	-0.26	-	-	-
asyn_fifo	1.3K	107	-0.23	-	-	-	-	-	-	-	-	-	0	0	0
ALU	2.4K	1.0K	-0.76	-	-	-	3.3K	1.4K	-0.71	-	-	-	-	-	-
PE	2.4K	363K	-1.03	2.5K	359K	-1.08	2.6K	366K	-1.06	2.2K	275K	-0.07	2.5K	358K	-1.08
risc_cpu	634	6.2K	-0.30	-	-	-	-	-	-	-	-	-	-	-	-
Best Quality Num	3	7	5	2	2	5	8	5	6	2	1	2	5	7	5

ranks the second, with 5, 7, 5 designs achieving the best area, power, and timing, respectively. Both of them perform better than the designer-crafted reference designs $\mathcal{V}_H$. This trend of design quality is similar to the trend of design correctness, indicating GPT-4 $>$ GPT-3.5 + self-planning $>$ GPT-3.5 $>$ Thakur et al. [5]. Please notice that, since there is a strong trade-off between different design objectives, this summation of individual best design quality leads to a straightforward but less rigorous comparison.

VI. Conclusion

In this work, we propose a comprehensive open-source benchmark for design RTL generation with natural language instructions. Compared with the datasets released in recent works, our benchmark includes more designs, also with higher design scale and complexity. We also propose an effective prompt engineering technique named self-planning. In our future work, we will first keep extending and maintaining this benchmark. We will also keep validating the self-planning technique. In addition, we will fine-tune our own open-source models to achieve better performance in our RTLLM benchmark.

Acknowledgement

This work is partially supported by the Hong Kong Research Grants Council (RGC) ECS Grant 26208723, National Natural Science Foundation of China (NSFC) 62304192, and ACCESS – AI Chip Center for Emerging Smart Systems, sponsored by InnoHK funding, Hong Kong SAR.

References

[1] G. Huang, J. Hu, Y. He, J. Liu, M. Ma, Z. Shen, J. Wu, Y. Xu, H. Zhang, K. Zhong *et al.*, "Machine learning for electronic design automation: A survey," *ACM TODAES*, 2021.

[2] 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," *TCAD*, 2021.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

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

[5] 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.

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

[7] 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.

[8] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," *arXiv preprint arXiv:2203.13474*, vol. 30, 2022.

[9] Synopsys, "Design Compiler® RTL Synthesis," 2022.

[10] 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.

[11] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.