Verilog_Coder: Verilog Code Generation using LLM and Evaluation using DC and ICC2 tools

Sri Samadarsini Vemulapalli
Department of Computer Engineering
Univeresity of Maryland Baltimore county
Email: jv29859@umbc.edu

***Abstract*—This paper delves into the growing knowledge of artificial intelligence into the hardware design. It explores the application of large language models (LLMs) in generation of Verilog code, a very first step in VLSI design. Specifically, the study compares the performance of GPT-3.5, GPT-4, and the Verilog-Coder LLM in producing functionally and syntactically correct RTL code. Through an experimental approach, the Verilog code generated by these models is synthesized and evaluated with Design Compiler (DC) and (ICC2) tools to evaluate their performance in terms of area and power consumption metrics. Also, Synopsys VCS simulator is used to evaluate its functionality correctness. The results indicate LLMs potential in RTL code generation which reduces the manual coding errors and their efforts.**

1. Introduction

In the rapidly growing field of Very Large Scale Integration (VLSI) design, the automation of hardware description language (HDL) coding, specifically Verilog, is a key step in the beginning of the design process. Traditional methods of Verilog coding are manual, time-consuming, and prone to human error, which can affect the design cycle of ICs. Artificial intelligence (AI), especially in the domain of Large Language Models (LLMs), has opened new platforms for automating complex tasks such as summarization, prediction, and the generation of Verilog code.

This paper explores LLMs and has designed a language model named Verilog_Coder for this project, which is designed specifically to automate the generation of Verilog Code based on prompts given in natural language descriptions. Verilog_Coder is compared against advanced language models such as GPT-3.5 and GPT-4, to assess their capability in generating accurate synthesizable Verilog code. These different versions of Verilog code for the same prompt will be collected and evaluated using DC and ICC2 tools to evaluate the generated code in terms of area and power.

1.1. Large Language Model (LLM)

A Large Language Model (LLM) is an advancement of artificial intelligence which has the ability to generate human-like text by analyzing the large set of corpora. Language models like GPT, which is a well-known version these days, predict and assemble words for the given natural language description. This LLMs are useful in tasks like text summarization, writing or reviewing articles, language translations, and even in coding in different programming languages. These language models are very powerful because they can handle complex tasks that require human intelligence.

- In this project, I designed a *Verilog_Coder* using a newly developed training scheme which is based on code quality feedback and generates RTL code for the given prompt in natural language description.
- For the same prompt, I have collected different versions of Verilog code generated by language models GPT 3.5 and GPT 4.
- All these three versions of RTL files are synthesized and evaluated using DC and ICC2 tools.
- Some of the generated codes from the benchmark RTLLM V1.1 are taken to evaluate their functionality in Synopsys VCS simulator from an open-source EDA playground.

The following paper includes the sections of the work involved in the project implementation. In section 2, I discussed the background work done to this project before implementing the Verilog_Coder. In section 3, the proposed work for this project will be discussed with the implementation process. The following sections includes the experimental setup, results, and Conclusion for this paper.

2. Background Work

Advancement of technology in natural language processing, large language models such as GPT have shown its remarkable performance in this field. Researchers inspired by this have started using LLM-based techniques in automation. To cite an instance, LLM-based solutions are proposed to design AI accelerator architectures, designing quantum architectures, fixing security bugs, and even generating the RTL code. Among these, the RTL code generation attracts and grasps my attention towards its ability and efficiency.

Many research works are going through the automation of Verilog code generation.

Some of the existing works in this field are focusing on prompt engineering methods without creating datasets or models needed for code generation. With a deep exploration, I found work on generating a large training dataset by collecting numerous Verilog-based projects and then fine-tuning their own model [4]. The limitation of this model is evaluating on another benchmark, this fine-tuned model is inferior to GPT 3.5. NVIDIA also proposes its own dataset for model fine-tuning and it also proposed its own benchmark called *Verilog_Eval* [3]. This is the first model to attain comparable results with the GPT 3.5. However, the limitation of this model is neither a dataset nor a developed model is open source to the public. Also, other models that are efficient in the software code generation process like CodeGen, StarCoder, and Mistral are inferior to GPT 3.5 in RTL code generation.

As mentioned, good performance and efficient RTL generation models are currently unavailable. According to my study, in recent times to fill the gap, researchers have come up with innovative work in this field. They developed an LLM model named RTL Coder [1] to generate the Verilog code for the given instruction. To the best of my knowledge, it is the first model that outperforms the GPT 3.5. Their work involves the generation of a high-quality labeled dataset over various samples for the RTL task. RTL Coder is trained with their own developed dataset which achieves outstanding results when compared to existing results.

2.1. Existing Training Scheme

The existing works on LLM-based Verilog code generation involve the development of a model by training and fine-tuning on a specific Verilog dataset. However, most of the existing works lack their efficacy in generating Verilog code. This is because of the training scheme they are used during their implementation. If *Verilog_Coder* uses a basic existing training scheme, it will follow the process flow illustrated below [1].

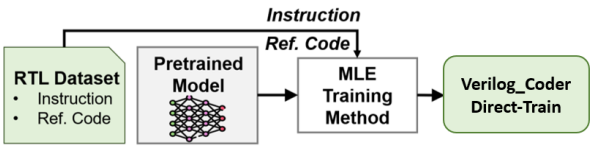

Figure 1. Flow of the training process for Verilog_Coder.

Figure 1 depicts the flow of the existing direct training process for the *Verilog_Coder*. The process starts with the RTL dataset which contains Instruction and Reference code which are paired together as samples. This RTL dataset is very crucial because the models start learning the syntax and structure of Verilog code from the foundational examples provided by this dataset. The next component present in the flow is the pre-trained model. This model is based on the neural network architecture such as transformer which has been initially trained on a wide range of corpus of text data to understand the meaning of natural language. For any model like *Verilog_Coder*, the pre-trained model undergoes further training, basically called as fine-tuning using the specific RTL dataset. Here, the fine-tuning uses the Maximum Likelihood Estimation training method which adjusts the model's parameters to minimize the difference between predicted and actual data.

This method helps in accurate predictions by the model as described in the instructions. Finally, the output of the flow is Directed trained Verilog]_Coder which is capable of generating Verilog codes for the given instruction.

2.2. Training Data and Loss Function

Consider a training dataset $\{x_i, y_i\}$ for $i = 1, \ldots, N$, where x_i represents a design instruction and 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 $t = 1, 2, \ldots, T$ to represent the tokenized sequence [1].

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 (where r can be any single token in the vocabulary) as $P(r_t|x_i, y_i^{<t}; \pi)$, where π represents the model parameters and $y_i^{<t}$ denotes the already generated previous sequence $y_i^1, \ldots, y_i^{t-1}$. Then the log probability of generating the whole sequence can be written as:

$$\sum_{t=1}^{T} \log P(y_i^t|x_i, y_i^{<t}; \pi)$$

In the existing training method, Maximum Likelihood Estimation (MLE) is commonly used to find the best parameters that maximize the log probability. The training flow is shown in Figure 12(a). The loss is usually defined as below:

$$\text{loss}_{\text{mle}} = -\sum_{t=1}^{T} \log P(y_i^t|x_i, y_i^{<t}; \pi)$$

However, there are limitations to this existing method called exposure bias. Exposure bias, which means the bias occurs when the model continually predicts the next token based on its own previous outputs rather than the reference tokens, which can lead to significant deviations from the intended output. To reduce the exposure bias, I suggest to consider the reference code with the instruction and previous token. Now, that the generation of model's code is different from the reference code, it is necessary and useful to introduce the scoring mechanism to evaluate the quality of generated code candidates.

3. Proposed Work

As discussed in previous section 2, my work involves model training with the scoring mechanism. The proposed workflow is shown in Fig 2.

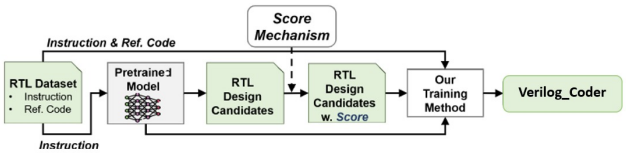

Figure 2. Proposed training scheme based on scoring mechanism

In this flow, multiple code snippets will be collected which are generated by the pre-trained model. Then, these code snippets will be paired with the original reference code together as $y_i = \{v_{i,k}\}$ for $k = 1, 2, \ldots, K$ where K number of code snippets generated for each instruction. Next, all these candidates will be scored by the scoring mechanism represented by $R(x_i, v_{i,k})$ which could be a syntax checker or functionality check. We will then obtain a set of score $z_i = \{z_{i,k}\}$, $k = 1, 2, \ldots, K$, denoting the quality for the code sample $\{y_{i,k}\}$. This process helps the model to learn new information along with reference code. In this way, it assigns the higher probabilities to answers with higher scores.

The conditional log probability (length-normalized) of generating the entire code $y_{i,k}$ is commonly written as [1]:

$$p_{i,k} = \frac{\sum_t \log P(y_{i,k}^t | x_i, y_{i,k}^{<t}; \pi)}{\|y_{i,k}\|}$$

where $\|y_{i,k}\|$ represents the length of the sequence $y_{i,k}$, and π denotes the model parameters, ensuring that the probability is normalized by the sequence length.

We calculate $p_{i,k}$ for all code candidates $Y_i = \{y_{i,k}\}, k = 1, \ldots, K$. These are then normalized using a softmax function to define the probability of each candidate being selected:

$$s_{i,k} = \frac{\exp(p_{i,k})}{\sum_{r=1}^{K} \exp(p_{i,r})}$$

This $s_{i,k}$ reflects the model's tendency to output the kth 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 as:

$$\text{loss}_{\text{compare}} = \sum_k \max(s_{i,k} - s_{i,t} + \lambda, 0)$$

where λ is a threshold value, and $s_{i,t}$ represents the score of a specific reference or target sequence against which comparisons are made.

3.1. Model Training

In the model training phase, the first step is to consider the Verilog dataset which consists of instruction code pairs [RTL_coder]. For this proposed method, the basic pre-trained model for fine-tuning on this dataset is considered the DeepSeek-Coder-6.7b. In all experiments, we used the Adam optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and learning rate $\gamma = 1 \times 10^{-5}$. The context length is 2048 and a global batch size is 256. To implement the proposed training scheme using a pre-trained model, we use a syntax checker for scoring. For reference code, the assigned score is 1. The code candidates will be scored by comparing them with the reference code in the dataset. Finally, the Verilog_Coder is trained using a scoring mechanism such that it is ready to provide the Verilog code for the given instruction in natural language description.

3.2. Workflow of Project

The overall workflow involved in this project is defined in this current section and is shown in Figure 3.

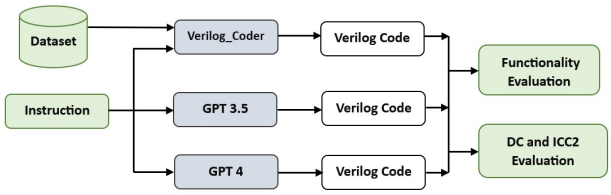

Figure 3. Workflow of the project

As shown in the above figure, the process is about the generation of Verilog code using three different Large Language Models (LLMs) — Verilog_Coder, GPT-3.5, and GPT-4. A predefined dataset contains specific instructions and a reference code such that Verilog_Coder will adapt the reference code from this dataset for the given instruction. For all three LLMs, the same instruction will be passed such that three versions of Verilog codes will be collected for evaluation. The two types of evaluations used in this project to assess the generated Verilog codes are: Firstly, the particular prompt the generated codes are evaluated using DC and ICC2 tools to report the area and power metrics.

The evaluation phase also incorporates the use of an open-source platform EDA playground along with the selection of Synopsys VCS tool simulator. In this evaluation, some of the generated Verilog codes are taken and a respective testbench is written for them to simulate. Now, the Verilog code is simulated with the testbench, and a waveform is generated to observe its functionality.

4. Experimental Setup

4.1. Implementation

The model Verilog_Coder designed in this paper is implemented using Python and its in-built libraries which sup-

port the language processing and machine learning process. The workstation used is an AMD Ryzen 7 processor with 16GB RAM which provides the computational strength. These experiments were carried out in the Google Colab Pro platform with an extended GPU, which supports the optimizing performance for model training and evaluation.

4.2. Benchmarks

For testing the Verilog_Coder, the benchmarks used in this paper are Verilog_eval and RTLLM V1.1. In Verilog_eval there is machine_eval and human_eval. In this project, I used in machine_eval benchmark and generated the output successfully. For the RTLLM V1.1 benchmark, the RTL codes are generated for the respective instructions presented in the benchmark.

4.3. Evaluation

The evaluation of the generated code files involves two types, both synthesis and simulation. Design compiler (DC) and ICC2 tools are used to synthesize the Verilog code generated by three LLMs. This includes evaluating the gate-level optimizations, area, and power usage. Another evaluation is to check the functionality of the code. Synopsys VCS simulator available on EDA playground was used to simulate the Verilog code functionality. Here, respective test benches are written for some of the generated verilog codes and verified their functionality.

5. Experimental Results

By the end of the project, I am able to generate the Verilog code for the instruction given in natural language description using the developed Verilog_Coder.

Prompt: Please act as a professional Verilog designer and design a 2-bit half adder including a clock.
The generated output is shown below.

```
module half_adder
(input [1:0]a1,
input [1:0]a2,
input clk,
output [1:0]sum,
output carry);

reg [1:0]sum;
reg carry;

always @(posedge clk) begin
    sum <= a1 + a2;
    carry <= (a1 & a2);
end

endmodule
```

Listing 1. Half Adder Module

5.1. Area and Power Metrics Evaluation

Moving to DC and ICC2 results, all three versions of Verilog codes from three LLMs, GPT-3.5, GPT-4, and Verilog_Coder are evaluated with these tools for area and power metrics. According to the reports obtained, the following table is constructed to show the comparison between their performance. These designs are generated from Verilog_Coder by giving different prompts as shown before.

TABLE 1. COMPARISON OF AREA METRICS ACROSS DESIGNS

Design	GPT 3.5 (um^2)	GPT 4 (um^2)	Verilog_Coder (um^2)
half_adder	29.73	30.49	30.49
2to4 Decoder	44.73	44.73	44.73
updown_counter	352.24	284.64	296.49
serial_adder	282.10	178.97	53.88
comparator	233.30	105.17	165.49
ring_counter	145.66	89.60	96.37
alu	629.01	488.73	430.01

TABLE 2. COMPARISON OF POWER METRICS ACROSS DESIGNS

Design	GPT 3.5 (pW)	GPT 4 (pW)	Verilog_Coder (pW)
half_adder	3.00e+07	3.07e+07	3.05e+07
2to4 Decoder	4.00e+07	4.00e+07	4.00e+07
updown_counter	1.69e+08	1.37e+08	1.45e+08
serial_adder	9.39e+06	1.09e+07	2.64e+07
comparator	7.21e+07	1.59e+07	3.76e+07
ring_counter	7.81e+07	4.37e+07	5.62e+07
alu	9.13e+07	5.94e+07	7.32e+07

In addition, Synopsys VCS simulator from an open-source EDA playground is used to evaluate its functionality. Some of the generated files are taken and written a test-bench for respective design and generated the waveform for checking its functionality. The observed functionality details are shown in the table below.

TABLE 3. FUNCTIONALITY CORRECTNESS OF THE GENERATED CODES

Design	Functionality Correctness
adder_16bit	✓
counter_12	✓
Right shifter	✓
accu	×
signal_generator	✓
calendar	✓
FSM	✓
multi_16bit	×

These designs have been taken from the codes generated by Verilog_Coder during the test of the RTLLM V1.1 benchmark. Some of the designs are taken and written a respective testbench to evaluate every design using Synopsys VCS simulator in an open-source EDA playground. Out of eight designs simulated, six designs generated the correct output and passes the functionality test. So, we can assume the model efficiency in generating the functionally correct codes is 75%. This may vary positively or negatively while the number of designs changes.

5.2. Results Comparison

In this section, the efficiency of Verilog_Coder is compared with GPT 3.5 and GPT 4. Verilog_Coder is more efficient than GPT 3.5 in both area and power metrics. It is efficient than GPT 3.5 area by 26.96%, and power usage by 10.41%. While in the context of GPT 4, Verilog_Coder is not much efficient than the GPT 4. But it got comparable results in terms of area and a significant decrease in efficiency of power.

TABLE 4. COMPARISON OF AREA AND POWER METRICS FOR VERILOG_CODER OVER GPT3.5 AND GPT4

Metric	Verilog_Coder over GPT3.5	Verilog_Coder over GPT4
Area	26.96%	-2.13%
Power	10.41%	-17.96%

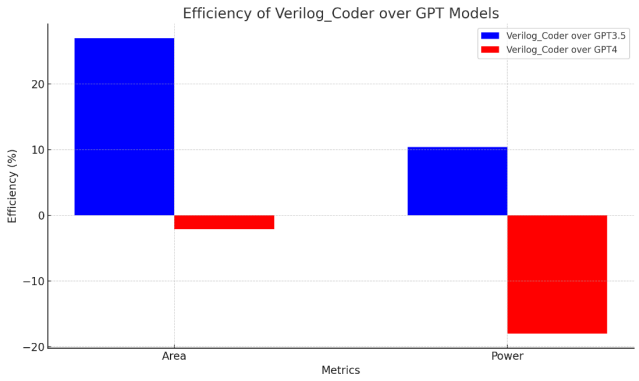

Figure 4. Efficiency of Verilog_Coder over GPT3.5 and GPT4

6. Conclusion

This paper has explored the capabilities of large language models (LLMs), especially focusing on designing Verilog_Coder to automate the generation of verilog code which is a first step in the design process. Through involving scoring mechanism in the training process with Maximum Likelihood Estimation method the Verilog_Coder has shown its efficiency in converting the natural language instructions into the verilog code. The use of standard tools such as DC and ICC2 gives its performance in terms of area and power metrics. Verilog_Coder is more efficient than GPT 3.5 in both area and power metrics. However, when compared to GPT 4 the verilog_coder got comparable results in terms of area and its efficiency is decreased in power metrics. In addition, synopsys VCS simulator from an open source EDA playground is used to observe its functionality correctness. Future research could focus on enhancing the model's ability to translate more complex design instructions into syntactically correct code by improving its training algorithms.

References

[1] Liu, S., Fang, W., Lu, Y., Zhang, Q., Zhang, H., & Xie, Z. (2023). Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution. *arXiv preprint arXiv:2312.08617*.

[2] Lu, Y., Liu, S., Zhang, Q., & Xie, Z. (2023). RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model. *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 722-727.

[3] Liu, M., Pinckney, N., Khailany, B., & Ren, H. (2023, October). Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)* (pp. 1-8). IEEE.

[4] Thakur, S., Ahmad, B., Pearce, H., Tan, B., Dolan-Gavitt, B., Karri, R., & Garg, S. (2024). Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems*, 29(3), 1-31.