arXiv:2309.07544v1 [cs.LG] 14 Sep 2023

VerilogEval: Evaluating Large Language Models for Verilog Code Generation

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren
NVIDIA Corporation
{mingjiel, npinckney, bkhailany, haoxingr}@nvidia.com

***Abstract*—The increasing popularity of large language models (LLMs) has paved the way for their application in diverse domains. This paper proposes a benchmarking framework tailored specifically for evaluating LLM performance in the context of Verilog code generation for hardware design and verification. We present a comprehensive evaluation dataset consisting of 156 problems from the Verilog instructional website HDLBits. The evaluation set consists of a diverse set of Verilog code generation tasks, ranging from simple combinational circuits to complex finite state machines. The Verilog code completions can be automatically tested for functional correctness by comparing the transient simulation outputs of the generated design with a golden solution. We also demonstrate that the Verilog code generation capability of pretrained language models could be improved with supervised fine-tuning by bootstrapping with LLM generated synthetic problem-code pairs.**

I. INTRODUCTION

The escalating popularity of Large Language Models (LLMs), characterized by their remarkable capacity to comprehend and generate human-like text, has opened up a realm of possibilities across diverse domains [1]–[3]. LLMs tailored for specific domains have garnered significant attention owing to their impressive performance on both general-purpose benchmarks and specialized tasks within domains like financial engineering [4], biomedical studies [5], [6], and general scientific research [7]. When it comes to coding, LLMs can assist developers by suggesting code snippets, offering solutions to common programming challenges, and even explaining complex concepts in a more accessible manner [8], [9].

In the realm of Electronic Design Automation, LLMs provide the potential to aid engineers in designing and verifying digital systems, providing insights into Verilog coding, optimizing circuits, and automating time-consuming tasks [10], [11]. A number of studies have initiated the evaluation of LLMs' potential in generating Verilog code. Thakur et al. [12] fine-tuned CodeGen [9] models which was evaluated on 17 designs. A later follow up work [10] further demonstrate the ability to design chip-level circuits with ChatGPT. RTLLM [13] propose a benchmark framework with 30 designs, which focus on increasing the benchmark design scalability. The authors further improved the solution quality with simple and effective prompt engineering techniques.

While LLMs have proven to be powerful tools, their pretraining phase, characterized by unsupervised training loss, often lacks alignment with specific tasks. To enhance performance, supervised fine-tuning (SFT) is used [14], involving task-specific data to adapt to requirements. Ensuring model alignment is imperative for achieving improved performance, driving the investigation of increasingly computationally demanding techniques, such as reinforcement learning with human feedback [15], [16]. The cost associated with acquiring labeled data also remains a barrier, prompting a growing interest in alternative annotation-free alignment techniques. Self-Instruct [17] starts with a seed task, using LLMs to create more instructions and instances. WizardLM's Evol-Instruct [18] evolves instructions for a diverse dataset, which is further applied to code generation [19]. Additionally, a recent study utilized GPT-4 to generate a high-quality synthetic textbook dataset, achieving superior coding capabilities at 1/100th of the cost of other models [20]. Within the realm of Verilog coding, there remains a notable gap in the exploration of supervised fine-tuning for model enhancement.

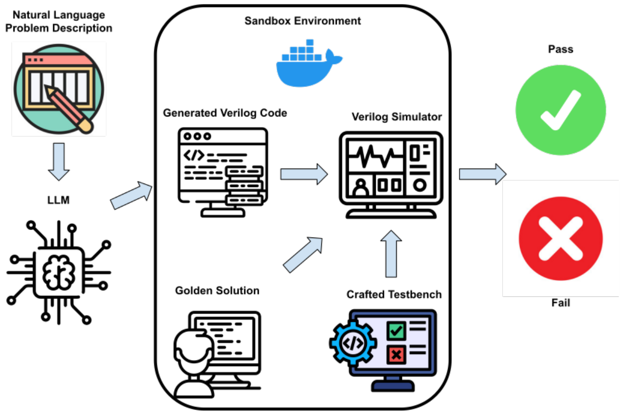


Fig. 1 **VerilogEval** uses a sandbox environment for simple and reproducible evaluation of LLM Verilog code generation

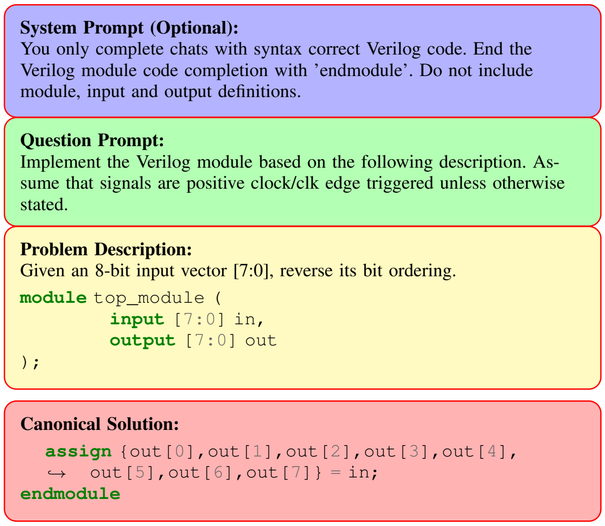
Moreover, notwithstanding commendable endeavors, recent research in Verilog code benchmarking has revealed limitations concerning its comprehensiveness, quantity, and the diversity of problems studied. Effective benchmarks should exhibit diversity, encompassing a wide array of topics, to mitigate testing variance. Furthermore, they should offer unambiguous problem descriptions, ensuring that solutions can be assessed with clear distinctions regarding correctness. In addition, reliability and automation are key factors, enabling the straightforward evaluation of generated code through robust testing procedures.

Our research addresses these gaps through the introduction of **VerilogEval**[1], a open-source benchmark that encompasses a diverse array of questions, offers clear and unambiguous problem descriptions, and incorporates automated, easily reproducible testing procedures. This contribution significantly enhances the robustness and effectiveness of the evaluation framework for Verilog code generation and assessment. Our specific contributions are as follows:

- We present a comprehensive evaluation dataset comprising 156 problems sourced from the HDLBits. These problems have undergone meticulous curation, ensuring both clarity and diversity.
- We developed a benchmarking framework wherein Verilog code completions are subjected to automatic functional correctness testing.
- We constructed a synthetic supervised fine-tuning dataset by leveraging LLMs to generate problem descriptions paired with Verilog code. This dataset is employed in extensive experiments on SFT, further enhancing the model's proficiency in Verilog coding tasks.

II. EVALUATION FRAMEWORK

In this section we discuss the details of our evaluation framework and evaluation dataset collection. Our work closely follows the widely adopted Python coding benchmark HumanEval [21] for best practices. **VerilogEval** is presented in Fig. 1 where we develop a sandbox environment for simple and reproducible evaluation of LLM Verilog code generation.



System Prompt (Optional):
You only complete chats with syntax correct Verilog code. End the Verilog module code completion with 'endmodule'. Do not include module, input and output definitions.

Question Prompt:
Implement the Verilog module based on the following description. Assume that signals are positive clock/clk edge triggered unless otherwise stated.

Problem Description:
Given an 8-bit input vector [7:0], reverse its bit ordering.

```
module top_module (
        input [7:0] in,
        output [7:0] out
);
```

Canonical Solution:

```
  assign {out[0],out[1],out[2],out[3],out[4],
↪   out[5],out[6],out[7]} = in;
endmodule
```

Fig. 2 Example of `vectorr` in **VerilogEval-human**. The **Problem Description** includes both natural language description and module header, input, and output definition.

A. *VerilogEval* Evaluation Set

We evaluate functional correctness on a selected set of problems from the Verilog instructional website HDLBits[2]. HDLBits is a collection of digital circuit design exercises and an online judge for learning digital logic using the Verilog hardware description language. The evalutation set consists of diverse Verilog coding tasks, ranging from module implementation of simple combinational circuits to complex finite state machines, code debugging, and testbench construction.

We focus on generating *self-contained*[3] Verilog modules from natural language text descriptions. We define a Verilog module as *self-contained* if the module implementation does not require instantiation of any other modules. We emphasize the significance of module instantiation as a crucial capability in Verilog, playing an essential role in constructing extensive system-level designs. It's important to note that our evaluation does not delve into this topic. However, while most problems in **VerilogEval** are intentionally concise, they demand the LLM to possess a comprehension of hardware design along with adept problem-solving skills in areas encompassing circuits, Boolean logic, state transitions, and more.

Fig. 2 shows an example of the problem `vectorr`. **Problem Description** includes both natural language description and module header and IO definition. Including the module header removes ambiguity such as the bit width of signals. The **Question Prompt** is concatenated with **Problem Description** and sent to the LLM for inference. **Canonical Solution** is provided as the golden solution for testing.

B. *Problem Descriptions*

Although HDLBits serves as a valuable resource for Verilog coding challenges, a significant portion of the website's problem descriptions are not readily compatible with text-only language models. These problem descriptions rely on various modalities, frequently incorporating circuit schematic images, state transition diagram graphs, Boolean logic tables, and Karnaugh maps. We explore with two methods for generating text-only problem descriptions for these problem sets.

*1) **VerilogEval-machine**:* We completely disregard the descriptions on the website and opt to utilize LLMs for the automated creation of problem descriptions. We employ the prompt template depicted in Fig. 3, employing `gpt-3.5-turbo`. Initially, we create all problem descriptions using zero-shot methods. We validate these descriptions by using the LLM to produce code solutions. Problem descriptions are considered invalid if none of the generated completions succeed across 100 samples, and such descriptions are then discarded. Surprisingly, among the pool of 156 candidate problems, 108 of them yield successful solutions upon initial sampling. Subsequent to this, we consider the valid generated descriptions as few-shot examples (4-shot) and proceed to further sample unsolved problems. In this phase, we iteratively sample descriptions along with their corresponding code completions (8 completions per description). Descriptions are labeled as valid as soon as any of the corresponding code completions pass testing. Sampling for each problem is halted

---

[1] https://github.com/NVlabs/verilog-eval

[2] https://hdlbits.01xz.net/wiki/Problem_sets

[3] Example of a removed question that is not *self-contained*: https://hdlbits.01xz.net/wiki/Module_cseladd.

System Prompt:
Explain the high-level functionality of the Verilog module. Use as many high-level concepts that are directly applicable to describe the code, say at the level of an undergraduate EECS major, but do not include extraneous details that aren't immediately applicable. Use text-based truth tables and state transition graphs when necessary. Speak concisely as if this was a specification for a circuit designer to implement. You should only reply with descriptive natural language and not use any code.

Few-shot Examples (Optional):
Question: Explain the high-level functionality of the Verilog module.

```
module top_module (
        input [7:0] in,
        output [7:0] out
);
  assign {out[0],out[1],out[2],out[3],out[4],
  ↪   out[5],out[6],out[7]} = in;
endmodule
```

Answer: This top Verilog module is a 8-bit unidirectional data port. It takes an 8-bit input in, and outputs an 8-bit signal out. The output of the module is assigned to out based on the value of in. If in = a,b,c,d,e,f,g,h, then out = h,g,f,e,d,c,b,a.

User Prompt:
Question: Explain the high-level functionality of the Verilog module.

```
module top_module (
        input [5:0] y,
        input w,
        output Y1,
        output Y3
);
  assign Y1 = y[0]&w;
  assign Y3 = (y[1]|y[2]|y[4]|y[5]) & ~w;
endmodule
```

GPT-3.5 Response:
Answer: This Verilog module is a combinational logic circuit that takes in 6-bit input vector y and a single bit input w. The module then outputs two signals Y1 and Y3. Y1 is the logical AND of the first bit of the input vector y and the single bit input w. Y3 is the logical AND of the logical OR of the second, third, fifth and sixth bits of the input vector y and the logical NOT of the single bit input w. In other words, Y1 is true if both the first bit of the input vector y and the single bit input w are true, and Y3 is true if at least one of the second, third, fifth and sixth bits of the input vector y is true and the single bit input w is false.

Fig. 3 **VerilogEval-machine** use `gpt-3.5-turbo` to generate problem descriptions for `2012_q2b`.

upon reaching a allocated sampling budget, resulting in an increase of 35 additional solutions. In total we generated 143 valid problem descriptions.

User Prompt:
I want you to write Verilog from a state transition diagram. How can I give you this diagram in a text format?

GPT-3.5 Response:
For a more graphical representation using text, you can specify the states with their output values and the transitions like this:

```
StateA (0) --0--> StateA
StateA (0) --1--> StateB
StateB (1) --0--> StateC
StateB (1) --1--> StateB
StateC (0) --0--> StateA
StateC (1) --1--> StateB
```

Fig. 4 ChatGPT guidance on state transition diagrams.

*2) **VerilogEval-human**:* We engaged in manual review and conversion of problem descriptions from the website into a text-only structure. We dedicated particular attention to addressing ambiguity within the problem descriptions, particularly when precisely determining attributes such as the clock's posedge or negedge triggering, whether reset and enable signals are active high or active low, and whether they operate synchronously or asynchronously. Boolean logic tables and Karnaugh maps were transformed into textual tabular formats. Circuit schematic diagrams were translated into natural language explanations of the connections between logical gates. For sequential waveforms, we meticulously detailed all signal values at each transition edge of the clock, presented in a tabular layout with an added column for time steps. One particular challenge we confronted revolved around the task of converting state transition graphs into a text-based representation. To tackle this, we turned to ChatGPT for guidance, as depicted in Fig. 4. We ultimately adopted the edge list-based format to depict these state transition graphs. Examples of manually converted descriptions are shown in Fig. 5. Initial explorations were conducted regarding Verilog code completion by employing the converted formats. Notably, ChatGPT exhibited the capability to generate meaningful code using these formats for simple problems. We manually converted 156 problem descriptions in total.

Comparing the descriptions between **machine** and **human**, we find that **machine** descriptions are often more verbose (`vectorr` in Figs. 2 and 3). Although the model is directed to generate high-level explanations, produced **machine** descriptions frequently delve into low-level details. These descriptions tend to mirror the code's implementation line by line, rather than focusing on the overarching functionality of the circuit (`2012_q2b` in Figs. 3 and 5). Furthermore, despite that we have taken steps to ensure that all **machine** descriptions are capable of producing passing solutions through LLMs, we cannot guarantee the absence of ambiguity and errors. Nevertheless, **VerilogEval-machine** remains a valuable benchmark, particularly for assessing LLM's competence in comprehending low-level instructions and generating syntactically and functionally accurate Verilog code.

Benchmark	Language	Train	Test
HumanEval [21]	Python	-	164
MBPP [22]	Python	374	500
APPS [23]	Python	5,000	5,000
VerilogEval-machine	Verilog	8,502	143
VerilogEval-human	Verilog	-	156

TABLE I **VerilogEval** in comparison with popular (python) coding benchmarks.

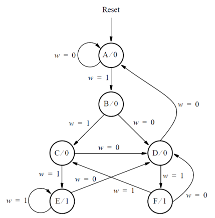
TABLE I summarizes our **VerilogEval** benchmark and compares with existing popular coding benchmarks in python. We further detail our method of generating synthetic supervised fine-tuning dataset in Section III-A.

C. Automated Testing Environment

Unlike software programming languages like Python, the behavior of Verilog modules are typically evaluated through

Consider the state machine shown below:

```
A (0) --1--> B
A (0) --0--> A
B (0) --1--> C
B (0) --0--> D
C (0) --1--> E
C (0) --0--> D
D (0) --1--> F
D (0) --0--> A
E (1) --1--> E
E (1) --0--> D
F (1) --1--> C
F (1) --0--> D
```

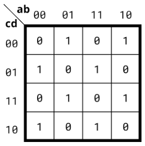


Assume that a one-hot code is used with the state assignment y[5:0] = 000001(A), 000010(B), 000100(C), 001000(D), 010000(E), 100000(F). Write a Verilog for the signal Y1, which is the input of state flip-flop y[1], for the signal Y3, which is the input of state flip-flop y[3]. Derive the Verilog by inspection assuming a one-hot encoding.

(a) `2012_q2b`

Implement the circuit described by the Karnaugh map below.

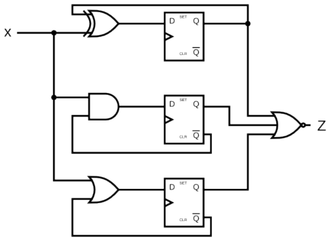
```
      ab
cd    00 01 11 10
 00 | 0 | 1 | 0 | 1 |
 01 | 1 | 0 | 1 | 0 |
 11 | 0 | 1 | 0 | 1 |
 10 | 1 | 0 | 1 | 0 |
```



cd \ ab	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

(b) `kmap4`

Given the finite state machine circuit described below, assume that the D flip-flops are initially reset to zero before the machine begins.
Build this circuit in Verilog.
`Input x goes to three different two-input gates: a XOR, an AND, and a OR gate. Each of the three gates is connected to the input of a D flip-flop and then the flip-flop outputs all go to a three-input XNOR, whose output is Z. The second input of the XOR is its corresponding flip-flop’s output, the second input of the AND is its corresponding flip-flop’s complemented output, and finally the second input of the OR is its corresponding flip-flop’s complementary output.`



(c) `ece241_2014_q4`

Fig. 5 Examples of **VerilogEval-human** descriptions. We show original website descriptions alongside manually converted text format.

simulations. To enable automated testing, we compare simulation results between generated code completions with golden reference solutions. We assert for output signal correctness at clock (posedge and/or negedge) transition edges for sequential circuits, while for combinational circuits, we validate them when any input signals changes. Our testbench incorporates two categories of input signals for each problem: manually crafted test patterns of significance, and randomly generated test patterns. Randomly generated test patterns may span from a few hundred clock cycles for simple problems to several thousand cycles for more complex ones.

We adapted the sandbox environment to safely run untrusted programs from HumanEval [21]. We built and installed the open-source ICARUS Verilog [24] simulator in a docker container. We note that our evaluation of Verilog syntax is limited by the simulator, which might not include all features of Verilog HDL IEEE-1364 standard. Simulation and testing are handled under the hood and results can be produced using just a single line of command.

D. Evaluation Metric

Early works on code evaluations report on match-based metrics such as BLEU score [25]. However, recent works [21], [26] have argued that such metrics does not correlate well with functional correctness. In Fig. 6 we show that Verilog coding exhibit similar issues, where the distributions of correct versus wrong solutions are not clearly seperable based on BLEU score probability densities.

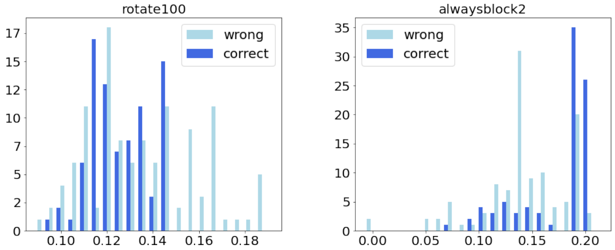


Fig. 6 BLEU score probability densities for correct and wrong solutions from `codegen-16B-verilog` [12] for 2 tasks from **VerilogEval-human**.

We follow recent work in directly measuring code functional correctness through pass@k metric [21], [22], [27], where a problem is considered solved if any of the k samples passes the unit tests. We also suggest using the unbiased estimator from [21]:

$$pass@k := \underset{Problems}{\mathbb{E}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right], \tag{1}$$

where we generate $n \geq k$ samples per task in which $c \leq n$ samples pass testing. In Fig. 7 we show that the number of samples n need to be sufficiently large to produce low variance estimates for pass@k.

III. SUPERVISED FINE-TUNING

This section provides our findings concerning the supervised fine-tuning (SFT) of Large Language Models (LLM) for Verilog coding. We elucidate our approach to the generation of synthetic SFT data, achieved by utilizing LLMs to create problem descriptions, detailed in Section III-A. Subsequently, Section III-B comprises a comprehensive suite of supervised fine-tuning (SFT) experiments, showcasing its potential for improving model performance.

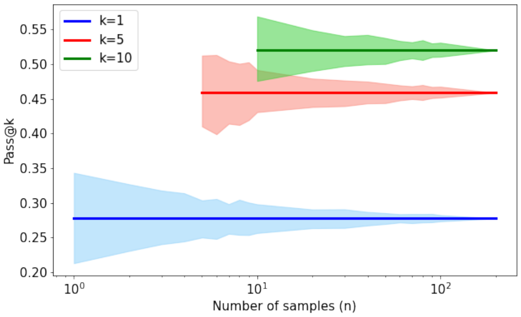


Fig. 7 Variance in estimating pass@k with n. Samples from `codegen-16B-verilog` [12] for **VerilogEval-human**.

A. Synthetic SFT Data Generation

In this work, we investigate the creation of synthetic SFT data through a bootstrapping process involving code descriptions generated by LLMs. To be precise, we undertake the task of identifying and refining *self-contained* Verilog modules sourced from Github data [12]. Subsequently, we employ the prompt template depicted in Fig. 3 to generate corresponding descriptive texts for each of these Verilog modules, effectively creating **machine** descriptions and code pairs. It's worth noting that our approach to synthetic data generation is straightforward in its implementation, and we acknowledge the potential for more advanced techniques as a promising future direction.

We leverage Pyverilog [28] to extract to abstract syntax tree from Verilog code and employ the following filtering process to identify *self-contained* Verilog modules from open-sourced Github Verilog code [12]:

- We verify that the filtered code contain the `module` and `endmodule` keywords, positioned at the beginning and end of the code, respectively.
- We remove Verilog modules more than 200 lines of code or exceeding 1024 tokens.
- We ascertain that the code includes at least one of the essential keywords: `always, assign, always_ff, always_comb, always_latch`.
- We ensure extracted modules are *self-contained* without any module instantiation.

We further perform approximate deduplication based on MinHash algorithm [29] using *Jaccard* similarity threshold of 0.8 as in [30]. We used `gpt-3.5-turbo` to generate code descriptions based on the prompt template in Fig. 3, using **VerilogEval-human** descriptions of `shift18, rule110, lemmings1, fsm3onehot` as *few-shot* examples. We selected these examples with the aim of encompassing a wide range of design instances and the utilization of natural language descriptions, including those presented in tabular formats. In total we generated 8,502 problem description and code pairs.

B. Results on Supervised Fine-tuning

We conduct extensive experiments on fine-tuning with the generated synthetic SFT data. Including both description and code, our SFT data is 11MB in file size, compared with ∼700MB of Github Verilog data used in [12]. For the fine-tuning process, we employed the Adam optimizer with hyperparameters $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$. We set the learning rate to $lr = 2e^{-5}$, effective batch size as 1M tokens, and opted not to apply weight decay. For all our experiments, we sample $n = 20$ code completions for measuring $pass@k = \{1, 5, 10\}$ using Equation (1). We use nucleus sampling [31] with top $p = 0.95$, temperature $temp = 0.8$, and context length of 2048. We used a single NVIDIA DGX node with 8 A100s and 2TB RAM.

Our experimentation primarily focuses on the CodeGen model series [9] and its Verilog-trained counterparts in [12]. These experiments encompass model sizes of `350M`, `2B`, `6B`, and `16B`. We use `-sft` to indicate models fine-tuned with our synthetic SFT data. We clarify our notation for base models as follows:

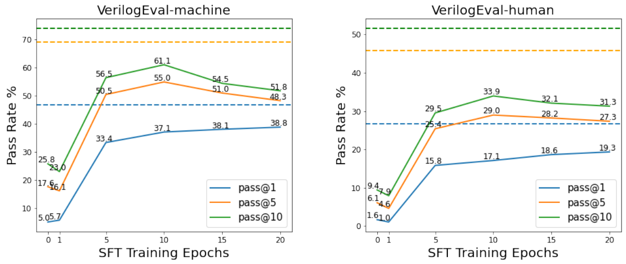
- `codegen-nl` [9] : Natural language model. Trained on ThePile [32] 825.18GB English text corpus.
- `codegen-multi` [9]: Code model. Initialized from `codegen-nl` and continue trained on BigQuery multilingual code dataset consisting of C, C++, Go, Java, JavaScript, and Python.
- `codegen-verilog` [12]: Verilog code model. Initialized from `codegen-multi` and continue trained on ∼300MB of Github Verilog and 400MB of textbook data.

Furthermore, we conducted comparisons with the `gpt-3.5-turbo` and `gpt-4` models through OpenAI APIs [33]. Our analysis specifically involved default `4k` context length models from `0613` checkpoints.

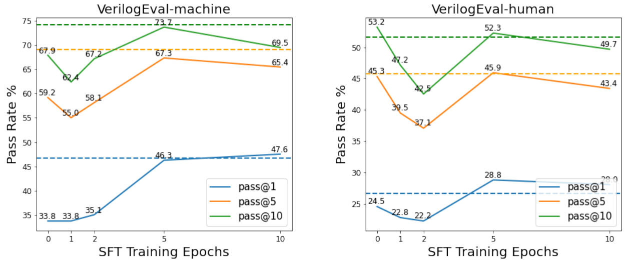
1) Training Epochs: Fig. 8 depicts the pass rate on **VerilogEval** with different SFT training epochs. Dashed lines indicate `gpt-3.5` results. Results show that **machine** descriptions correlate well with **human**, demonstrating that synthetic generated benchmarks could be a good indicator for downstream task performance.

In most cases, we observe that the performance metric $pass@1$ continues to exhibit improvement as the supervised fine-tuning (SFT) training epochs progress, whereas the metrics $pass@5$ and $pass@10$ begin to deteriorate. This trend suggests that with an increase in training epochs, the model tends to overfit to the SFT data, limiting its ability to generate diverse solutions for tackling complex challenges. Interestingly, this overfitting also leads to an increase in the model's confidence and success rate when dealing with simpler problems, highlighting a discernible trade-off between the $pass@1$ and $pass@10$ metrics. Consequently, we encourage future research to report on both of these metrics, particularly for models post-alignment, to provide a more comprehensive assessment of their performance.

Throughout the remainder of this study, we conduct supervised fine-tuning (SFT) using 10 epochs for `multi` and 5



(a) `codegen-16B-multi-sft`



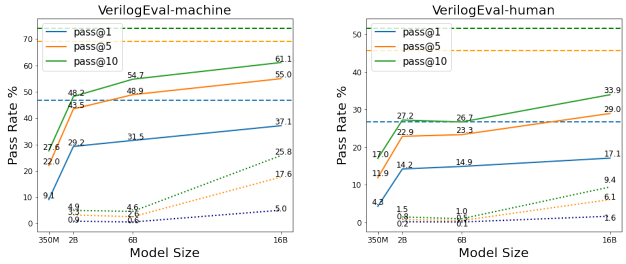
(b) `codegen-16B-verilog-sft`

Fig. 8 SFT training epochs and pass rate on **VerilogEval**. Dashed lines are `gpt-3.5` results.

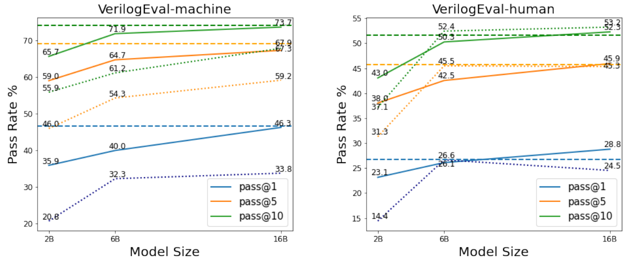
epochs for `verilog` models.

2) Model Size and Base Model: Fig. 9 illustrates the pass rates for the **VerilogEval** task using various model sizes and base models. The base model denotes the initial model checkpoint prior to SFT. It is worth noting that we have omitted the results for models with a size of `350M`, either due to their unavailability or because their pass rates are insufficient to demonstrate statistical significance. Our results suggest that more capable and larger models generally result in better Verilog coding capabilities.

In most instances, SFT using synthetically generated data yields notable enhancements in downstream model performance. These improvements are particularly pronounced, especially in the case of `multi` models, where the original model was not explicitly trained on a substantial corpus of Verilog code. In the case of `verilog` models, **VerilogEval-machine** exhibited significant performance gains, whereas the **VerilogEval-human** approach displayed comparatively less improvement and, at times, even slight deteriorations. Our SFT data is sourced from the GitHub Verilog corpus, and thus does not introduce additional Verilog code that the model did not encounter during its training for `verilog` models. However, by providing problem-code pairs, this data facilitates better alignment of the model, resulting in improved outcomes for **VerilogEval-machine**. Despite incorporating *few-shot* prompting during the generation of SFT data (as discussed in Section III-A), the generated descriptions tend to be primarily low-level, lacking the textual diversity found in **human** examples, such as state transition graphs, waveforms, Karnaugh maps, and similar elements. This "misalignment" between SFT data and **VerilogEval-human** might have caused `verilog-sft` models to degrade slightly in performance. We envision that increasing SFT (and Verilog pretraining) data diversity and quality would further lead to increased performance.



(a) `codegen-multi-sft`



(b) `codegen-verilog-sft`

Fig. 9 **VerilogEval** results on different model size. Solid lines are `sft` models, dotted lines are corresponding base models without SFT, dashed lines are `gpt-3.5` results.

Model	VerilogEval-machine			VerilogEval-human		
	pass@1	pass@5	pass@10	pass@1	pass@5	pass@10
`gpt-3.5`	46.7	69.1	74.1	26.7	45.8	51.7
`gpt-4`	47.9	67.8	72.9	27.0	45.8	52.0
`verilog-sft`	46.2	67.3	73.7	28.8	45.9	52.3

TABLE II Results on `gpt` models, comparing with `codegen-16B-verilog-sft`.

In TABLE II we present the results obtained from both the `gpt-3.5` and `gpt-4` models for the **VerilogEval** task. Additionally, we demonstrate that our top-performing model `codegen-16B-verilog-sft`, exhibits performance that is on par with `gpt` models.

Model	VerilogEval-machine		
	pass@1	pass@5	pass@10
`codegen-16B-nl-sft`	33.9	51.9	58.1
`codegen-16B-multi-sft`	37.1	55.0	61.1

TABLE III Comparing `nl` and `multi` as SFT base models.

In TABLE III we present a comparison of results between `sft` models utilizing two distinct base models: `codegen-nl` and `codegen-multi`. The tokenizer of `codegen-nl` model is inefficient in handling whitespaces, consequently preventing some of the **VerilogEval-human** problems from fitting within the limited context window of `2048` tokens. Thus we only display results for **VerilogEval-machine**. Despite the fact that `multi` models undergo pretraining on an extensive corpus of multi-lingual code data, they exhibit only marginal enhancements of approximately 3% when applied to Verilog coding task. This observation potentially suggests that there is limited positive knowledge transfer between software programming languages like C++ and hardware descriptive languages such as Verilog. This highlights the significance of pretraining on substantial Verilog corpora, as it can significantly enhance model performance in Verilog-related tasks.

3) SFT Data Quality: We conducted a comparative experiment aimed at assessing the significance of data quality in SFT. In this experiment, we introduced a manipulation by shuffling problem descriptions with incongruous Verilog code solutions, resulting in the creation of erroneous problem-code pairs denoted as `sft-error`. The outcomes, as presented in TABLE IV, provide a comparison of the performance results obtained through fine-tuning on the `codegen-2B-verilog` models concerning the **VerilogEval-machine** task. The results clearly demonstrate that the inclusion of incorrect problem-code pairs detrimentally impacts model performance, underscoring the critical importance of maintaining high-quality SFT data.

Model	**VerilogEval-machine**		
	pass@1	pass@5	pass@10
`codegen-2B-verilog`	20.1	46.0	55.9
`codegen-2B-verilog-sft`	35.9	59.0	65.7
`codegen-2B-verilog-sft-error`	21.4	38.8	46.1

TABLE IV Comparative experiment on SFT data quality. Incorrect low-quality SFT data degrades model performance.

IV. Limitations and Future Directions

In **VerilogEval**, our primary focus centers on harnessing Large Language Models (LLMs) to generate *self-contained* Verilog modules directly from natural language text descriptions. While we incorporate a wide array of hardware design topics through human-generated descriptions, it's important to note that our current evaluations are confined to boilerplate code generation for relatively small-scale designs. We emphasize the significance of module instantiation as a crucial capability in Verilog, as it plays a pivotal role in constructing complex system-level designs, albeit currently absent from our benchmark. Recent advancements in LLM-based coding benchmarking, as seen in [34], are starting to explore pragmatic code generation beyond standalone functions. It's worth mentioning that our testing environment solely assesses functional correctness and does not ensure that the generated Verilog code adheres to synthesizable formatting standards. We do not evaluate the performance of downstream circuit implementations, a gap that is addressed by the work presented in [13].

Additionally, it's crucial to recognize that boilerplate Hardware Description Language (HDL) code generation, as currently addressed in our **VerilogEval** and similar endeavors, inherently operates within an exceedingly limited scope within the broader landscape of hardware design. Hardware design, in its entirety, necessitates a multidisciplinary approach that draws on the expertise of domain professionals ranging from transistor device, circuit design, and to hardware system architecture. This holistic understanding is indispensable, as it allows design teams to navigate the intricacies of hardware design effectively. Furthermore, it's important to highlight that a significant portion of the hardware design process revolves around optimizing the Power, Performance, and Area (PPA) metrics. These three factors, power consumption, computational performance, and physical chip area, are paramount considerations in modern hardware design. Achieving the right balance among them is a formidable challenge that requires meticulous planning, advanced simulation, and iterative refinement. Equally critical is the extensive effort invested in design verification, aimed at ensuring the reliability and yield of the hardware. Verifying that a design functions as intended under diverse conditions and corner cases is vital to mitigate the risk of costly errors and to guarantee the final product meets its specifications. In essence, the successful realization of hardware designs hinges on the convergence of domain expertise, PPA optimization, and robust verification practices.

Nonetheless, Large Language Models (LLMs) present an exciting opportunity for future research to revolutionize the hardware design process. This transformative potential lies in their ability to collaborate with domain experts in formulating novel problems and devising innovative solutions. By leveraging the vast knowledge and natural language understanding of LLMs, domain experts can work in tandem with these models to explore uncharted territories in hardware design, potentially leading to breakthroughs that enhance the efficiency, reliability, and agility of the design process. The fusion of human expertise with machine intelligence using LLMs in this collaborative endeavor promises an exhilarating avenue for future research, one that holds the potential to reshape the very fabric of the hardware design research landscape.

V. Conclusion

The growing prominence of Large Language Models (LLMs) has ushered in a new era of their application across various domains. In this paper we introduce a specialized benchmarking framework meticulously designed to assess LLM performance within the realm of Verilog code generation for hardware design. The cornerstone of this contribution lies in the creation of a robust evaluation dataset, comprising of 156 distinct problems sourced from HDLBits. Furthermore, we have demonstrated that the Verilog code generation capabilities of pretrained language models can be enhanced through supervised fine-tuning, facilitated by the generation of synthetic problem-code pairs using LLMs. These findings not only advance the state of the art in Verilog code generation but also underscore the vast potential of LLMs in shaping the future of hardware design and verification.

Acknowledgment

The authors would like to thank Henry Wong (henry@stuffedcow.net) the creator of HDLBits for his invaluable assistance in providing reference solutions and testbenches for the problems used in this paper.

REFERENCES

[1] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[2] OpenAI, "Gpt-4 technical report," 2023.

[3] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg *et al.*, "Sparks of artificial general intelligence: Early experiments with gpt-4," *arXiv preprint arXiv:2303.12712*, 2023.

[4] S. Wu, O. Irsoy, S. Lu, V. Dabravolski, M. Dredze, S. Gehrmann, P. Kambadur, D. Rosenberg, and G. Mann, "Bloomberggpt: A large language model for finance," *arXiv preprint arXiv:2303.17564*, 2023.

[5] H.-C. Shin, Y. Zhang, E. Bakhturina, R. Puri, M. Patwary, M. Shoeybi, and R. Mani, "Biomegatron: Larger biomedical domain language model," *arXiv preprint arXiv:2010.06060*, 2020.

[6] E. Bolton, D. Hall, M. Yasunaga, T. Lee, C. Manning, and P. Liang, "BioMedLM." [Online]. Available: https://crfm.stanford.edu/2022/12/15/biomedlm.html

[7] R. Taylor, M. Kardas, G. Cucurull, T. Scialom, A. Hartshorn, E. Saravia, A. Poulton, V. Kerkez, and R. Stojnic, "Galactica: A large language model for science," *arXiv preprint arXiv:2211.09085*, 2022.

[8] A. Mastropaolo, L. Pascarella, E. Guglielmi, M. Ciniselli, S. Scalabrino, R. Oliveto, and G. Bavota, "On the robustness of code generation techniques: An empirical study on github copilot," *arXiv preprint arXiv:2302.00438*, 2023.

[9] 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," *ICLR*, 2023.

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

[11] 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.

[12] 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 *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[13] 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.

[14] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," *arXiv preprint arXiv:2109.01652*, 2021.

[15] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.

[16] 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.

[17] Y. Wang, Y. Kordi, S. Mishra, A. Liu, N. A. Smith, D. Khashabi, and H. Hajishirzi, "Self-instruct: Aligning language model with self generated instructions," *arXiv preprint arXiv:2212.10560*, 2022.

[18] C. Xu, Q. Sun, K. Zheng, X. Geng, P. Zhao, J. Feng, C. Tao, and D. Jiang, "Wizardlm: Empowering large language models to follow complex instructions," *arXiv preprint arXiv:2304.12244*, 2023.

[19] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "WizardCoder: Empowering code large language models with evol-instruct," *arXiv preprint arXiv:2306.08568*, 2023.

[20] S. Gunasekar, Y. Zhang, J. Aneja, C. C. T. Mendes, A. Del Giorno, S. Gopi, M. Javaheripi, P. Kauffmann, G. de Rosa, O. Saarikivi *et al.*, "Textbooks are all you need," *arXiv preprint arXiv:2306.11644*, 2023.

[21] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code.(2021)," *arXiv preprint arXiv:2107.03374*, 2021.

[22] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[23] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song *et al.*, "Measuring coding challenge competence with apps," *arXiv preprint arXiv:2105.09938*, 2021.

[24] S. Williams. (2023) The ICARUS verilog compilation system. [Online]. Available: https://github.com/steveicarus/iverilog

[25] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[26] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.

[27] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. S. Liang, "Spoc: Search-based pseudocode to code," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[28] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *Applied Reconfigurable Computing: 11th International Symposium, ARC 2015, Bochum, Germany, April 13-17, 2015, Proceedings 11*. Springer, 2015, pp. 451–460.

[29] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*. IEEE, 1997, pp. 21–29.

[30] K. Lee, D. Ippolito, A. Nystrom, C. Zhang, D. Eck, C. Callison-Burch, and N. Carlini, "Deduplicating training data makes language models better," *arXiv preprint arXiv:2107.06499*, 2021.

[31] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," *arXiv preprint arXiv:1904.09751*, 2019.

[32] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima *et al.*, "The pile: An 800gb dataset of diverse text for language modeling," *arXiv preprint arXiv:2101.00027*, 2020.

[33] OpenAI. (2023) OpenAI models api. [Online]. Available: https://platform.openai.com/docs/models

[34] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, T. Xie, and Q. Wang, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," 2023.