LLM4DV: Using Large Language Models for Hardware Test Stimuli Generation

Zixi Zhang^{1,*}, Balint Szekely^{2,*}, Pedro Gimenes^{2}, Greg Chadwick^{3},
Hugo McNally^{3}, Jianyi Cheng^{4}, Robert Mullins^{1} and Yiren Zhao^{2}
^{1}University of Cambridge, UK; ^{2}Imperial College London, UK; ^{3}lowRISC, UK; ^{4}University of Edinburgh, UK
* Equal Contribution
Email: zz458@cam.ac.uk, robert.mullins@cl.cam.ac.uk, {balint.szekely20, pedro.gimenes19, a.zhao}@imperial.ac.uk,
{gac, hugom}@lowrisc.org, jianyi.cheng@ed.ac.uk

***Abstract*—Hardware design verification (DV) is a process that checks the functional equivalence of a hardware design against its specifications, improving hardware reliability and robustness. A key task in the DV process is the test stimuli generation, which creates a set of conditions or inputs for testing. These test conditions are often complex and specific to the given hardware design, requiring substantial human engineering effort to optimize. We seek a solution of automated and efficient testing for arbitrary hardware designs that takes advantage of large language models (LLMs). LLMs have already shown promising results for improving hardware design automation, but remain under-explored for hardware DV. In this paper, we propose an open-source benchmarking framework named LLM4DV that efficiently orchestrates LLMs for automated hardware test stimuli generation. Our analysis evaluates six different LLMs involving six prompting improvements over eight hardware designs and provides insight for future work on LLMs development for efficient automated DV.**

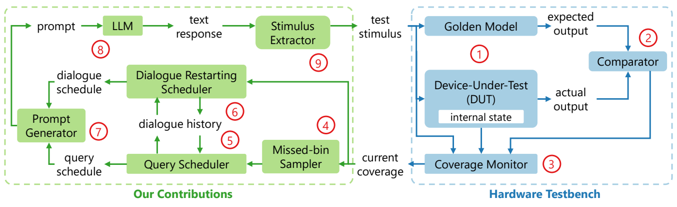


Fig. 1: An overview of LLM4DV framework. The *right part* shows a traditional DV process. DV engineers need to manually interact with the DV process by tailoring various stimulus and observing the coverage. Such a manual process is often iterative. The *left part* highlights our contributions, which adds the stimulus generation agent for automated guidance.

I. Introduction

Large Language Models (LLMs) [1], [2], [3] have gained significant attention in recent years due to their language generation and comprehension capabilities on tasks such as language translation [4], question answering [1], and sentiment analysis [5]. Recently, there has been interest in exploiting LLMs to improve hardware design generation [6], [7], [8]. Arguably, hardware design verification (DV), which checks the correctness of hardware designs, ranks among the most crucial and time-consuming tasks in hardware development. Hardware DV is often ***time-consuming***, usually taking up to 60%-70% of the development time [9], and requires significant ***human guidance and expertise*** due to the complexity of both hardware design and its corresponding testing requirements [10].

On the other hand, existing work on LLMs has been studied for software testing. For example, Codex [11] can produce functionally correct bodies of code from natural language docstring descriptions. LLaMA [3], an LLM using instruction tuning and Reinforcement Learning with Human Feedback (RLHF) [12], [13] for fine-tuning, emerges impressive generalization and external tool usage ability. However, these approaches are not directly applicable due to the following two challenges. First, unlike software programming languages, there is a scarcity of high-quality, open-source hardware designs and testing code available online for training LLMs. This limitation is critical because Hardware Description Languages (HDLs) possess ***distinct semantics*** that differ fundamentally from software programming languages. These unique characteristics make HDLs considerably more challenging for LLMs to interpret and learn from, as the models cannot simply transfer their knowledge from conventional programming contexts without substantial modifications. Second, the testing space for a hardware design design is typically large, leading to a ***scalability*** problem. Existing approaches on hardware DV require human guidance to reduce search space, such as adding heuristics to guide tests of a particular hardware design. This raises an important question: ***can LLMs effectively minimize the amount of human effort involved in hardware DV?***

In this work, we specifically focus on hardware test stimuli generation, which generates test inputs for hardware DV. A good stimuli discovers new hardware states during testing, increasing the test coverage; while a bad stimuli only tests existing states, leaving the coverage the same. Finding good stimuli becomes particularly arduous when encountering hard-to-hit points within the coverage plan. In order to find a path to LLM solutions, we present a novel benchmarking framework named LLM4DV (Large Language Model for Design Verification), that utilizes LLMs for ***test stimuli generation***, and make the following contributions:

- We introduce and construct LLM4DV, a framework that employs prompted LLMs to generate test stimuli for hardware DV. We show automated DV requires a complex prompting strategy and also propose six prompt enhancements to

TABLE I: Comparison to related work applying LLMs in the field of digital hardware.

Name	Task	Number of models	Testing space
RTLFixer[14]	Verilog syntax correction	1	212 syntax errors
NSPG[15]	Repairing security-relevant bugs in Verilog	4	10 designs (10 bugs)
ChipNeMo[16]	Bug analysis and summarisation	3	30 bugs
Kande et al.[17]	Generating security assertions	4	10 designs (10 assertions)
Thakur et al.[18]	Generating Verilog code	6	17 problems
RTLLM [19]	Generating Verilog code	4	30 designs
LLM4DV	Stimulus generation for functional verification	5	8 designs (3883 coverage bins)

establish strong baselines for the LLM4DV framework. We believe this provides an attractive testbed for experimenting the agentic behavior of LLMs.

- We design and construct three DUT modules: a Primitive Data Prefetcher Core, an Ibex CPU Instruction Decoder, and an Ibex CPU. We also select five open-source designs, obtaining a set of DUTs with different testing difficulties. We evaluate LLM4DV using these eight DUT modules and introduce a set of evaluation metrics. LLMs, with optimized prompt enhancements, achieve coverage rates (a primary metric for measuring verification effectiveness) ranging from 89.74% to 100% in a realistic setting. We open-source LLM4DV, the workflow and our optimized prompts, alongside these modules to allow the community to experiment with their ideas.

II. Background and Related Work

A traditional hardware DV process is illustrated in the right of Figure 1. For each hardware design, also known as device-under-test (DUT), the hardware designer provides a functionally equivalent golden model in software to the DUT [20]. The DV process takes a set of inputs, or test stimuli, and sends them to both the DUT and its golden model (①), leading to two sets of results. The results are then compared between the DUT and its golden model (②). A DV process typically tests the DUT iteratively on a large set of stimuli defined by the hardware designer in advance, which is known as the ***coverage plan***, and is normally in the form of ***coverage bins***. A coverage bin is a specific condition or scenario that the verification environment tracks to determine whether a particular aspect of the design has been exercised or tested. The coverage monitor (③) inspects the DUT's inputs, outputs, and internal states; determines whether there are hits of coverage bins; updates the current coverage and returns it to the stimulus generation agent for the next stimulus. The procedure in the right of Figure 1 typically follows an iterative approach, often executed tens of thousands of times, in which a human DV engineer applies various stimuli to achieve comprehensive coverage specified in the coverage plan.

Effective test stimuli generation has been a major challenge especially if one wants to meet 100% coverage [20]. For a simple design, verification can be done with individual directed tests, in which test stimuli (inputs for the DUT) are manually generated. For more complex designs, a large number of stimuli is required for exercising as much of the design's functionality as possible. Traditionally, ***constrained-random testing (CRT)*** has been used to generate vast random but valid test stimuli and to attempt to "hit" the bins. However, CRT is inefficient to hit as many bins as human effort for hardware states with complicated conditions.

In hardware design verification, assertion-based verification (ABV) is also widely adopted. ABV inserts assertions into the DUT HDL source to detect violations of predefined design properties. However, ABV requires test patterns (i.e. input test stimuli) to activate given assertions and therefore reveal vulnerabilities. For simulation-based ABV approaches, traditional test generation that uses random or constrained-random tests cannot guarantee to activate assertions with complex conditions in a reasonable time, even with optimizations [21], [22], [23], [24], and can face the complexity explosion problems [20]. We present an alternative solution ito this issue by utilizing LLM's pre-trained knowledge to reason about the given coverage plan and guide the test stimuli generation. Other advanced testing techniques, such as coverage-directed generation and mutating tests [25], [26], [27], have been studied to improve the performance of CRT.

Recently, the application of LLMs for hardware design and verification purposes has started to gain traction [28]. Table I provides a summary of recent benchmarks that focus on applying LLMs within this domain. In particular, *there are currently no benchmarks that evaluate the stimuli generation capabilities of LLMs.* Among the recent contributions, RTL-Fixer [14] enables the automated correction of Verilog syntax errors. In contrast, NSPG [15] is designed to extract security properties by analyzing hardware documentation. ChipNeMo [16] has been assessed on tasks related to bug summarization and analysis. Additionally, Kande et al. [17] proposed a methodology for automatically generating SystemVerilog assertions (SVAs) using LLMs to enhance hardware security. Meanwhile, Thakur et al. [18] and RTLLM [19] have explored the generation of Register Transfer Level (RTL) code using LLMs. While it is challenging to directly compare the scale of these benchmarks with that of LLM4DV due to the different abilities assessed, it should be noted that LLM4DV's scope of 3883 coverage bins across 8 devices, tested with six different off-the-shelf models, represents a significant contribution to this emerging field.

TABLE II: A list of input options and output evaluation metrics for the proposed LLM4DV framework.

	Names	**Descriptions**
Input Options	DUT	The target DUT to be tested.
	Model	The LLM used for stimulus generation.
	Prompting Configurations	The prompting strategy for stimulus generation.
	Coverage Plan	The coverage plan specified for the target DUT.
Evaluation Metrics	Max Coverage	Maximum recorded number of coverage bins (defined by the coverage plan) covered. A higher number indicates better performance.
	Effective Message Count	The minimum number of messages an LLM produces across several trials in an experiment achieving maximum coverage; a lower count indicates better performance.
	Average Message Count	Average number of query messages per experiment $\pm$ standard deviation of messages. As the usage of LLMs is costly, a faster convergence to maximum coverage is preferred.

III. LLM4DV Benchmarks

A. LLM4DV Framework

In this work, the proposed LLM4DV framework automates the DV process by exploiting LLMs for test stimuli generation, shown in the left of Figure 1. This reduces human involvement in the hardware DV loop and effectively guides tests to increase coverage rates. In each generation cycle, the prompt generator produces a prompt based on a template (⑦) and the current coverage feedback from the coverage monitor. LLM4DV allows customization of prompts inside a dialogue, this means each ***query message*** can receive different prompts, as managed by the query scheduler (⑤) shown in Figure 1. This is explained in Section III-C.

The LLM takes in the prompt and generates a natural language response, from which the test stimulus values are extracted and sent to the DV flow in the right of Figure 1. The DV framework then produces current coverage which is considered as input for the LLM-based stimulus generation agent (⑧) shown in Figure 1. The processes of test stimuli generation and hardware testbench simulation are executed in parallel asynchronously. A buffer is placed between the stimulus generation agent interfaces to balance the rate of the test stimuli generation and consumption. In every timestep when the stimulus generation agent is requested for a test stimulus, it takes out the oldest value in its buffer; if the buffer is empty, the agent takes in a new request and generates new stimuli.

In LLM4DV, each DV process is viewed as a "***trial***", where there would be multiple dialogues made in a single trial, as illustrated in Figure 1, which are controlled by the dialogue scheduler (⑥). A *trial* stops if certain user-set conditions are met (eg. max token counts reached), and the agent is considered "exhausted".

B. Evaluation setup: DUTs and models

The proposed LLM4DV benchmark contains eight DUT modules. Three of the devices were developed by the authors, and the other five are open-source designs. These DUTs are selected because they are commonly seen in most representative computer architectures such as CPUs, GPUs and other hardware accelerators. They are:

- *Primitive Data Prefetcher Core*: Detects stride patterns in a series of integers.
- *Ibex Instruction Decoder*: Decodes RISC-V instructions.
- *Ibex CPU*: A RISC-V CPU core.
- *Asynchronous FIFO* [29]: A dual clock FIFO.
- *AMPLE Prefetcher Weight Bank* [30]: A component of AMPLE, a graph neural network accelerator. It is responsible for fetching data from memory, storing it in a FIFO.
- *AMPLE Prefetcher Fetch Tag* [30]: Another component of AMPLE. Similar to the weight bank, its basic purpose is to fetch data from memory.
- *SDRAM Controller* [31]: a simple SDRAM controller.
- *MIPS CPU* [32]: A MIPS CPU core.

We use six different commercially available LLMs: GPT-3.5 Turbo, Llama v2 70B Chat, Claude 3 Sonnet, CodeLlama 70B Instruct, Llama 3 70B Instruct, and Claude 3.5 Sonnet. To evaluate the effectiveness of these LLMs, we observe the testing performance based on three evaluation metrics, as listed in the lower part of Table II. We have limited each trial to 700 messages. The design choices of all parameters and prompting are based on our ablation runs.

C. LLM4DV prompting enhancements

LLM4DV supports custom user prompts at various interaction stages. We present the following prompting strategies that are already integrated in LLM4DV. These prompting techniques have been empirically observed to influence final performance to varying extents.

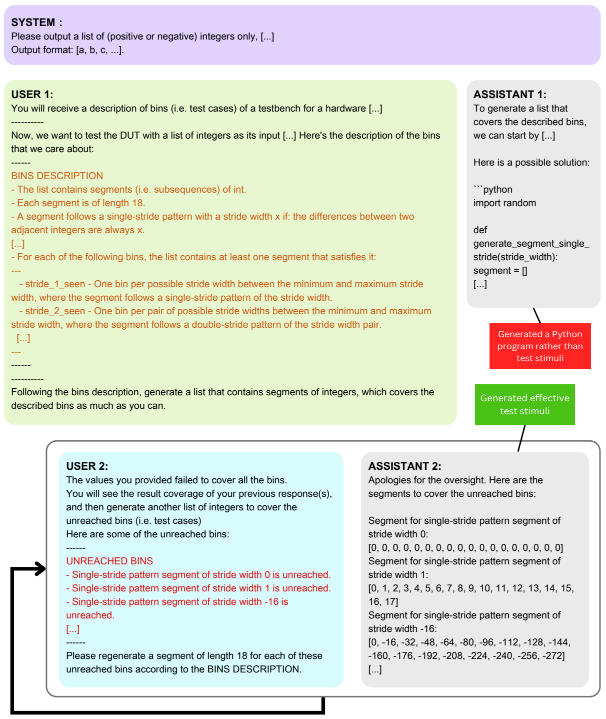
Missed-bin sampling This optimization is ④ in Figure 1, and is later used in the query scheduler. We propose missed-bin sampling, which samples a number of bins from all uncovered bins to be included in the differences part of iterative queries.

Best-iterative-message sampling The LLM needs previous query messages to learn about what has happened. However, as the dialogue grows, the length of input may exceed the LLM's input limit. We propose to randomly keep recent messages and "successful" previous messages that hit the largest number of bins. These strategies are used in our Query Scheduler in ⑤.

Dialogue restarting LLMs sometimes behave stubbornly, repeating mistakes they made previously. To overcome this behavior, we introduce a dialogue restarting scheduler (⑥). When the LLM hits less than three new bins within t responses, we clear the dialogue record and restart from the system message and initial query.

Best-iterative-message buffer reset When the dialogue record is reset, the buffer for "successful" previous messages in *Best-iterative-message sampling* can also be cleared or kept. These two strategies display a trade-off between "effectively forgetting past mistakes" and "faster task re-learning after restart". This is incorporated in the restarting scheduler (⑥).

Providing the DUT code We include the DUT source code in the initial message to allow the LLM to leverage the HDL code to directly correlate specific features and functions with the corresponding coverage bins. Due to limited context length, this technique is applicable only when the device's code fits within the LLM's context window.



SYSTEM :
Please output a list of (positive or negative) integers only, [...]
Output format: [a, b, c, ...].

USER 1:
You will receive a description of bins (i.e. test cases) of a testbench for a hardware [...]

Now, we want to test the DUT with a list of integers as its input [...] Here's the description of the bins that we care about:

BINS DESCRIPTION
- The list contains segments (i.e. subsequences) of int.
- Each segment is of length 18.
- A segment follows a single-stride pattern with a stride width x if: the differences between two adjacent integers are always x.
[...]
- For each of the following bins, the list contains at least one segment that satisfies it:

  - stride_1_seen - One bin per possible stride width between the minimum and maximum stride width, where the segment follows a single-stride pattern of the stride width.
  - stride_2_seen - One bin per pair of possible stride widths between the minimum and maximum stride width, where the segment follows a double-stride pattern of the stride width pair.
  [...]

Following the bins description, generate a list that contains segments of integers, which covers the described bins as much as you can.

ASSISTANT 1:
To generate a list that covers the described bins, we can start by [...]

Here is a possible solution:

~~~
```python
import random

def
generate_segment_single_
stride(stride_width):
segment = []
[...]
~~~

USER 2:
The values you provided failed to cover all the bins.
You will see the result coverage of your previous response(s), and then generate another list of integers to cover the unreached bins (i.e. test cases)
Here are some of the unreached bins:

UNREACHED BINS
- Single-stride pattern segment of stride width 0 is unreached.
- Single-stride pattern segment of stride width 1 is unreached.
- Single-stride pattern segment of stride width -16 is unreached.
[...]

Please regenerate a segment of length 18 for each of these unreached bins according to the BINS DESCRIPTION.

ASSISTANT 2:
Apologies for the oversight. Here are the segments to cover the unreached bins:

Segment for single-stride pattern segment of stride width 0:
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Segment for single-stride pattern segment of stride width 1:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
Segment for single-stride pattern segment of stride width -16:
[0, -16, -32, -48, -64, -80, -96, -112, -128, -144, -160, -176, -192, -208, -224, -240, -256, -272]
[...]

Generated a Python program rather than test stimuli
Generated effective test stimuli

Fig. 2: Example prompts and responses on the Primitive Data Prefetcher Core module. The purple box is the system message, containing a general format instruction. The green box is an initial query, containing a coverage plan summary (orange text). The blue box is an interactive query, containing differences i.e. coverage feedback (red text).

Few-shot prompting As task-specific fine-tuning is outside the scope of this study, we employ few-shot prompting to improve coverage metrics. Examples of stimuli with their corresponding bin hits are included in the initial prompt to facilitate the LLMs to assimilate the hardware verification context and the DUT information. To avoid skewing the experimental results, this technique is applicable only when the coverage plan includes more than 30 coverage bins.

Example Figure 2 demonstrates the LLM4DV flow works with several prompts and responses. The agent (USER) introduces the task and coverage plan in the initial message, and then provides coverage feedback in iterative messages. The LLM (ASSISTANT) generates textual responses according to the description and feedback.

IV. Results and Analysis

Table III presents the best results achieved for each LLM-DUT pair, compared with naive CRT and formal methods serving. In the CRT methodology, we generate 100,000 combinations within the valid input range without additional constraints. The formal baseline utilizes the cover mode of the SymbiYosys tool [33], where all bins of the coverage plans correspond to specific SystemVerilog cover statements, and each formal verification run is limited to a 48-hour timeout.

TABLE III: Best results achieved for each LLM-DUT pair. Each reported experiment employs the best applicable configuration of the prompting techniques described in Section IV. Experiments marked with * have used few-shot prompting, and experiments marked with † have included the DUT source code in the initial prompt. We highlight the **best** results for each DUT. Note that trials were limited to 700 messages.

		Primitive Data Prefetcher Core	Asynchronous FIFO	AMPLE Prefetcher Weight Bank	AMPLE Prefetcher Fetch Tag
gpt-3-turbo	Max coverage	1016 (98.26%)*	10 (100%)	324 (100%)†	**10 (100%)**
	Eff. msg. count	350	16	36	**2**
	Avg. msg. count	509.0 ± 129.4	19.7±3.9	37.7±1.2	**22.0±14.1**
llama-2-70b-chat	Max coverage	431 (41.68%)*	10 (100%)†	324 (100%)	10 (100%)
	Eff. msg. count	700	1	36	22
	Avg. msg. count	470.7±189.9	10.5±7.9	41.3±7.5	27.7±6.0
claude-3-sonnet	Max coverage	801 (77.47%)*	**10 (100%)**	324 (100%)	10 (100%)
	Eff. msg. count	700	**1**	36	8
	Avg. msg. count	676.3±33.5	**1.0**	36.0	19.3±8.0
codellama-70b-instruct	Max coverage	82 (7.93%)*	10 (100%)	324 (100%)	6 (60.00%)
	Eff. msg. count	154	1	44	34
	Avg. msg. count	102.0±50.3	3.7±3.1	52.3±8.5	28.3±4.0
llama-3-70b-instruct	Max coverage	710 (68.67%)*	10 (100%)†	**324 (100%)**	10 (100%)
	Eff. msg. count	700	1	**26**	15
	Avg. msg. count	700.0	1.3±0.5	**32.7±4.7**	20.0±3.6
claude-3.5-sonnet	Max coverage	**1022 (98.84%)***	**10 (100%)**	324 (100%)	9 (90%)
	Eff. msg. count	**321**	**1**	36	25
	Avg. msg. count	**329.3±32.3**	**1.0**	36.7±0.6	25.0
Formal verification	Max coverage	1030 (99.61%)	10 (100%)	3 (0.93%)	10 (100%)
CRT	Max coverage	0 (0%)	10 (100%)	324 (100%)	10 (100%)

		SDRAM Controller	Ibex CPU Instruction Decoder	Ibex CPU	MIPS CPU
gpt-3-turbo	Max coverage	7 (100%)	1466 (69.58%)*	39 (19.90%)*	84 (43.08%)*
	Eff. msg. count	7	700	102	211
	Avg. msg. count	22.3±11.0	432.0±228.3	88.0±21.2	111.0±72.8
llama-2-70b-chat	Max coverage	6 (85.71%)†	402 (19.08%)*	22 (11.22%)*	68 (34.87%)*
	Eff. msg. count	32	186	26	55
	Avg. msg. count	28.3±2.6	125.7±61.1	33.3±10.4	45.7±13.2
claude-3-sonnet	Max coverage	7 (100%)†	1512 (71.76%)*	141 (71.94%)*	159 (81.54%)*
	Eff. msg. count	2	700	315	299
	Avg. msg. count	2.3±0.5	700.0	287±19.9	277.7±35.2
codellama-70b-instruct	Max coverage	7 (100%)†	417 (19.79%)*	25 (12.76%)*	91 (46.67%)*
	Eff. msg. count	8	182	31	142
	Avg. msg. count	29.3±15.1	126.3±57.6	34.3±6.9	113.7±20.4
llama-3-70b-instruct	Max coverage	**7 (100%)**	1135 (53.89%)*	94 (47.96%)*	98 (50.26%)*
	Eff. msg. count	**1**	700	172	175
	Avg. msg. count	**2.3±1.2**	700	180.3±20.9	141±24.1
claude-3.5-sonnet	Max coverage	7 (100%)†	**2006 (95.21%)***	**196 (100%)***	**175 (89.74%)***
	Eff. msg. count	2	**651**	**31**	**176**
	Avg. msg. count	2.0	**683.7±28.3**	**37.0±5.29**	**174.7±41.0**
Formal verification	Max coverage	7 (100%)	2106 (99.95%)	100%	100%
CRT	Max coverage	7 (100%)	1154 (54.77%)	30 (15.31%)	28 (14.36%)

Across all DUTs, each configuration demonstrates that LLM4DV can either match or exceed the coverage rates achieved via naive CRT. This signifies not only the adaptability of LLMs to varied hardware testing contexts but also their potential to streamline certain aspects of verification by reducing reliance on extensive random input generation.

V. Conclusion: Gimmick or Trend?

The hardware design community is now starting to see debates regarding the effectiveness of LLMs for automated chip design, questioning whether their use is merely a gimmick or represents a future trend. Our particular take on this problem is that there is a need to set up open datasets and benchmarks for problems in chip design, so that the effectiveness and potential use of LLMs can be fully understood and quantified.

LLM4DV fits exactly in this category, and we target, in our opinion, the most human labor-intensive part (in terms of engineering) of the chip design process. Our baseline results have demonstrated that LLMs can achieve satisfactory coverage rates on straightforward designs, but they struggle with more complex ones, suggesting that LLMs do hold promise within the specific context of automated hardware DV.

REFERENCES

[1] Z. Yang, N. Garcia, C. Chu, M. Otani, Y. Nakashima, and H. Takemura, “Bert representations for video question answering,” in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2020, pp. 1556–1565.

[2] “Openai: Introducing chatgpt,” https://openai.com/blog/chatgpt, 2020.

[3] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, “Llama 2: Open foundation and fine-tuned chat models,” *arXiv preprint arXiv:2307.09288*, 2023.

[4] F. Feng, Y. Yang, D. Cer, N. Arivazhagan, and W. Wang, “Language-agnostic bert sentence embedding,” *arXiv preprint arXiv:2007.01852*, 2020.

[5] J. Liu, D. Shen, Y. Zhang, B. Dolan, L. Carin, and W. Chen, “What makes good in-context examples for gpt-3?” *arXiv preprint arXiv:2101.06804*, 2021.

[6] J. Blocklove, S. Garg, R. Karri, and H. Pearce, “Chip-chat: Challenges and opportunities in conversational hardware design,” in *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*. IEEE, 2023, pp. 1–6.

[7] Y. Fu, Y. Zhang, Z. Yu, S. Li, Z. Ye, C. Li, C. Wan, and Y. C. Lin, “Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models,” in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.

[8] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, “Rtllm: An open-source benchmark for design rtl generation with large language model,” in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2024, pp. 722–727.

[9] H. Shin, “Efficient bug discovery with machine learning for hardware verification,” https://community.arm.com/arm-research/b/articles/posts/efficient-bug-discovery-with-machine-learning-for-hardware-verification, 2024.

[10] ——, “Data-centric machine learning pipeline for hardware verification,” in *2022 IEEE 35th International System-on-Chip Conference (SOCC)*. IEEE, 2022, pp. 1–2.

[11] M. Chen, J. Tworek, H. Jun, Q. Yuan, J. K. Henrique Ponde de Oliveira Pinto, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.

[12] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, “Deep reinforcement learning from human preferences,” *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[13] N. Stiennon, L. Ouyang, J. Wu, D. Ziegler, R. Lowe, C. Voss, A. Radford, D. Amodei, and P. F. Christiano, “Learning to summarize with human feedback,” *Advances in Neural Information Processing Systems*, vol. 33, p. 3008–3021, 2020.

[14] Y. Tsai, M. Liu, and H. Ren, “Rtlfixer: Automatically fixing rtl syntax errors with large language models,” *arXiv preprint arXiv:2311.16543*, 2023.

[15] X. Meng, A. Srivastava, A. Arunachalam, A. Ray, P. H. Silva, R. Psiakis, Y. Makris, and K. Basu, “Unlocking hardware security assurance: The potential of llms,” *arXiv preprint arXiv:2308.11042*, 2023.

[16] M. Liu, T.-D. Ene, R. Kirby, C. Cheng, N. Pinckney, R. Liang, J. Alben, H. Anand, S. Banerjee, I. Bayraktaroglu, B. Bhaskaran, B. Catanzaro, A. Chaudhuri, S. Clay, B. Dally, L. Dang, P. Deshpande, S. Dhodhi, S. Halepete, E. Hill, J. Hu, S. Jain, A. Jindal, B. Khailany, G. Kokai, K. Kunal, X. Li, C. Lind, H. Liu, S. Oberman, S. Omar, G. Pasandi, S. Pratty, J. Raiman, A. Sarkar, Z. Shao, H. Sun, P. P. Suthar, V. Tej, W. Turner, K. Xu, and H. Ren, “Chipnemo: Domain-adapted llms for chip design,” *arXiv preprint arXiv:2311.00176*, 2023.

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

[18] 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,” *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, 2023.

[19] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, “Rtllm: An open-source benchmark for design rtl generation with large language model,” *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 722–727, 2023.

[20] H. Witharana, Y. Lyu, S. Charles, and P. Mishra, “A survey on assertion based hardware verification,” *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–33, 2022.

[21] B. Pal, A. Banerjee, A. Sinha, and P. Dasgupta, “Accelerating assertion coverage with adaptive testbenches,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 5, p. 967–972, 2008.

[22] L. Ferro, L. Pierre, Y. Ledru, and L. du Bousquet, “Generation of test programs for the assertion-based verification of tlm models,” *2008 3rd International Design and Test Workshop*, p. 237–242, 2008.

[23] J. G. Tong, M. Boule, and Z. Zilic, “Airwolf-tg: A test generator for assertion-based dynamic verification,” *2009 IEEE International High Level Design Validation and Test Workshop*, p. 106–113, 2009.

[24] Y. Lyu and P. Mishra, “Automated test generation for activation of assertions in rtl models,” *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, p. 223–228, 2020.

[25] S. Fine and A. Ziv, “Coverage directed test generation for functional verification using bayesian networks,” in *Proceedings of the 40th annual Design Automation Conference*, 2003, pp. 286–291.

[26] O. Guzey and L.-C. Wang, “Coverage-directed test generation through automatic constraint extraction,” in *2007 IEEE International High Level Design Validation and Test Workshop*. IEEE, 2007, pp. 151–158.

[27] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, “Rfuzz: Coverage-directed fuzz testing of rtl on fpgas,” in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[28] R. Zhong, X. Du, S. Kai, Z. Tang, S. Xu, H.-L. Zhen, J. Hao, Q. Xu, M. Yuan, and J. Yan, “Llm4eda: Emerging progress in large language models for electronic design automation,” *arXiv preprint arXiv:2401.12224*, 2023.

[29] D. Pretet, “Asynchronous dual clock fifo,” https://github.com/dpretet/async_fifo.

[30] P. Gimenes, “Ample: Accelerated message passing logic engine,” https://github.com/pgimenes/ample.

[31] S. Horne, “Sdram memory controller,” https://github.com/stffrdhrn/sdram-controller.

[32] TrivialMIPS, “Nontrivialmips,” https://github.com/trivialmips/nontrivial-mips.

[33] SymbiYosys, “Front-end for yosys-based formal verification flows,” https://github.com/YosysHQ/sby.