AssertionBench: A Benchmark to Evaluate Large-Language Models for Assertion Generation for Hardware Design

Anonymous ACL submission

Abstract

Assertions have been the de facto collateral for hardware verification for over a decade. The verification quality, *i.e.*, detection and diagnosis of corner-case design bugs, is critically dependent on the assertion quality. There has been a considerable amount of research to generate high-quality assertions from hardware design source code and design execution trace data. With recent advent of generative AI techniques such as Large-Language Models (LLMs), there has been a renewed interest in deploying LLMs for assertion generation. However, there is little effort to quantitatively establish the effectiveness and suitability of various LLMs for assertion generation. In this paper, we present AssertionBench, a novel benchmark to evaluate LLMs' effectiveness for assertion generation quantitatively. AssertionBench contains 100 curated Verilog hardware designs from OpenCores and formally verified assertions for each design, generated from GoldMine and HARM. We use AssertionBench to compare state-of-the-art LLMs to assess their effectiveness in inferring functionally correct assertions for hardware designs. Our experiments comprehensively demonstrate how LLMs perform relative to each other, the benefits of using more in-context examples in generating a higher fraction of functionally correct assertions, and the significant room for improvement for LLM-based assertion generators.

1 Introduction:

System-on-Chip (SoC) designs are the building blocks for many safety-critical computing systems, including vehicular systems, infrastructure, military, and industrial automation. It is crucial to establish that the SoCs are functionally correct, safe, and secure to ensure that the SoC designs function as intended and are free from errors and vulnerabilities.

Assertions or *design invariants* are mathematical encoding (in Boolean logic) of desired design properties that should hold True in the hardware design. Assertion-based Verification (ABV) (Witharana et al., 2022) has long emerged as the de facto standard to verify security and functional correctness of SoCs. However, crafting a succinct set of assertions for hardware designs is a tedious and time-consuming task, often requiring considerable human ingenuity. Too many assertions can negatively affect verification performance with a prolonged verification closure, whereas too few assertions may result in insufficient design coverage causing corner case design bugs to escape to production and mass manufacturing. ***Consequently, it is crucial to develop automated and scalable techniques to rapidly generate a succinct set of hardware design properties targeting design functionality, safety, and security***.

There has been a considerable amount of research work for assertion generation using two different paradigms – lightweight static analysis of design source code and formal verification (Tiwari et al., 2001; Păsăreanu and Visser, 2004), and data-driven statistical analysis, *e.g.*, data mining (Ernst et al., 2000; Hangal et al., 2005; Pinter and Majzik, 2005; DeOrio et al., 2009; Chang and Wang, 2010). GoldMine, for the first time developed a static analysis guided statistical analysis-based technique to generate hardware assertions in Linear Temporal Logic (Pnueli, 1977) in a fully automated way. While GoldMine and follow-up research made assertions accessible beyond the elite group of hardware verification engineers, most of those methods still failed to scale to large industry-scale designs due to the algorithmic complexity of the underlying static analysis.

With recent advances in generative AI mod-

els, especially Large-Language Models (LLMs), there is a renewed interest in the research community to harness the power of LLMs for assertion generation task. Most recent assertion generation approaches (c.f., Appendix A.2) treat a LLM model as a ***black-box*** and use *prompt engineering* to iteratively refine the set of generated assertions. However, there is no in-depth study nor a dataset to evaluate on how well different state-of-the-art (SOTA) LLM models perform on generating correct set of assertions without designer-developed prompts.

To address this gap, in this work, we propose AssertionBench, the first comprehensive benchmark to quantify the efficacy of SOTA LLMs for the assertion generation task. The benchmark contains 100 curated designs of varying complexity encompassing a broad spectrum of design types, *e.g.*, UART, encoders, decoders, and arithmetic operations such as addition, multiplication and 2's complement in Floating Point Unit, along with their formally verified assertions facilitating future research in exploring applicability of LLMs in assertion generation. Our primary focus is to quantify the quality of the generated assertions using LLMs and a collection of labeled designs and their formally verified assertions as in-context (IC) examples.

2 The AssertionBench Benchmark

AssertionBench contains IC and and test designs from OpenCores (OpenCores, 2024). The benchmark contains combinational and sequential hardware designs containing up to 1250 lines of codes (LoCs) (Danial, 2021) excluding comments and blank lines.

In our benchmark, each IC examples consists of a Verilog design and its formally verified assertions, generated from GoldMine (Pal et al., 2020) and HARM (Germiniani and Pravadelli, 2022b), and verified using Cadence JasperGold (Cadence, 2024). The IC set comprises fundamental sequential designs such as Arbiter, T Flip-Flop, and combinational designs such as Half Adder, Full Adder, and Full Subtractor among others. The IC set assertions contains both overlapped and non-overlapped implication operators. Our test design set contains 100 Verilog designs that are up to 20× bigger than those in the IC set in term of LoCs, to evaluate LLMs' 1-shot and 5-shot learning.

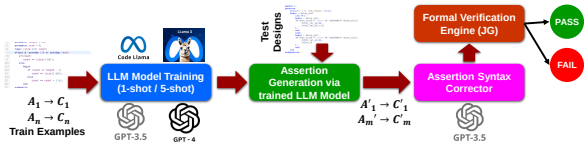

Figure 1: **Our evaluation framework**. **JG**: JasperGold Formal Property Verification Engine.

```
You are an expert in SystemVerilog Assertions.
Your task is to generate the list of assertions to the given verilog design. An example is shown below. Generate only the list of assertions for the test program with no additional text.
Program 1: module arb2(clk, rst, req1, req2, gnt1, gnt2); input clk, rst; ...
Assertions 1: (state == 1 & req2 == 1) |-> (gnt1 == 0);...
Test Program:
module fifo_mem #(parameter DEPTH=8, DATA_WIDTH=8, PTR_WIDTH=3) ( input wclk, w_en, rclk, r_en, input [PTR_WIDTH:0] b_wptr,  ...
Test Assertions:
```

Figure 2: **An example of the prompt for 1-shot learning**. The example consists of a tuple, a Verilog design (`Program 1`) and a set of formally verified assertions for the design (`Assertions 1`). The `Test Program` is the Verilog design for which we generate assertions using the trained LLM.

3 Experimental Setup

Figure 1 shows our evaluation framework. We evaluated four SOTA LLMs GPT-3.5 (Ye et al., 2023), GPT-4o (OpenAI et al., 2024), CodeLLaMa 2 (Rozière et al., 2024), and LLaMa3-70B (Meta, 2024) using the proposed AssertionBench on the task of predicting correct or valid assertions.

Compute Platform: We use UIUC NCSA's public Delta Cluster (NCSA, 2024) for our experiments. We use GPU nodes containing 1-way, 4-way, and 8-way NVIDIA A40 (with 48GB GDDR6) and A100 (with 40GB SXM) GPUs to perform k-shot learning and inference. Each 1-way and 4-way GPU computing node has 256 GB RAM and each 8-way GPU computing node has 2 TB RAM.

Hyperparameters: We use default hyperparamaters for all the LLM models under consideration. Specifically, the number of maximum output tokens has been set at 1024, employing a greedy decoding strategy and maintaining a *temperature* of 1.0, *top_p* of 0.95. The *random seed* has been configured to 50.

Pre-trained Models and Verification Tool: We use pre-trained LLMs from the HuggingFace model repository (HuggingFace, 2024) for AssertionBench evaluation. We have also used Python 3.11 and Cadence JasperGold (JG) version 2022.06p002 for formally verifying the assertions generated from the test Verilog designs. We use two SOTA classical tools GoldMine (Pal et al., 2020) and HARM (Germiniani and Pravadelli, 2022b) to generate *example assertions* from different Verilog designs in the IC set. Below, we summarize the four LLMs that we evaluate using AssertionBench.

- ***GPT-3.5*** is a commercial LLM built using the Generative Pre-trained Transformer (GPT) architecture. It is part of OpenAI's GPT series of models designed to understand and generate text based on the input it receives.
- ***GPT-4o*** accepts any combination of input audio, image, video, and text and responds with an output consisting of image, audio, video, and text. GPT-4o is a unified end-to-end model for text, vision, and audio.
- ***CodeLLaMa 2*** is a collection of pre-trained and fine-tuned generative text models. It is an auto-regressive language model based on transformer architecture with the large 70B model using Grouped-Query Attention for improved inference scalability.
- ***LLaMa3-70B*** has the context window length of 8192 and is pre-trained with 15 Trillion tokens of public data. LLaMa3-70B excels at language nuances, translation, contextual understanding, and dialogue generation.

Evaluation Protocol: To evaluate effectiveness of the different LLMs, the few-shot testing regime consists of 1-shot and 5-shot IC examples. Each example is a tuple consisting of a Verilog design source code and a set of formally verified assertions containing up to 10 assertions. Figure 2 shows our prompt consisting of four parts – (i) an English language description of the task, (ii) an IC Verilog design, (iii) an assertion in SystemVerilog Assertion (SVA) (SystemVerilog, 2024), and (iv) a test Verilog design. Next, we provide each IC learned model 100 test Verilog designs to infer assertions. In our experiments, we have found all of the LLM models generate syntactically erroneous assertions, *i.e.*, each LLM fails to learn the SVA syntax from the IC examples. Consequently, we use a syntax corrector (SC) using GPT-3.5 and feed the output of the SC to JG to evaluate the quality of the generated assertions. Any other SVA-compatible formal property verifier (FPV) will work as well.

Metrics: We evaluate the assertions generated from the test programs using following three metrics for each IC learning for each LLM – (i) **Pass** quantifies the fraction of generated assertions that FPV attests as valid for the design; (ii) **Fail** quantifies the fraction of generated assertions that FPV attests as wrong with a counterexample trace (CEX); and (iii) **Error** quantifies the fraction of generated assertions for which the FPV identifies syntactic errors even after syntax correction. We have not reported any other metric, *e.g.*, assertion coverage (Athavale et al., 2014), as that is meaningful only when an assertion is valid and one wants to quantify the quality of the assertion or would like to induce a ranking on assertions (Pal et al., 2020). In current work, we did not target to quantify the quality of the assertions neither we wanted to induce a rank on them. Rather we focused on the fitness of the SOTA LLMs on generating correct assertions.

4 Experimental Results

We show our overall experimental results in Figure 3. We make following observations.

Observation 1: **Most LLMs generate valid assertions with increasing number of IC examples** (c.f., Figure 3a-3d). GPT-3.5, GPT-4o, and CodeLLaMa 2 show on average an improvement of 2×, 1.2×, and 1.12× for valid assertion generation, respectively, when moved from 1-shot learning to 5-shot learning. However, LLaMa3-70B model loses accuracy from 31% to 24% on the same dataset. Our analysis shows in many cases, LLaMa3-70B either fails to generate assertions or generates syntactically wrong assertions or tries to generate codes in a new programming language (*e.g.*, Java). This experiment shows that there is a considerable scope for improving LLaMa3-70B model for this task, likely via fine-tuning the pre-trained LLaMa3-70B model.

Observation 2: **An enhanced model with higher number of model parameters does not necessarily ensure a better semantic or syntactic understanding**. For GPT-3.5

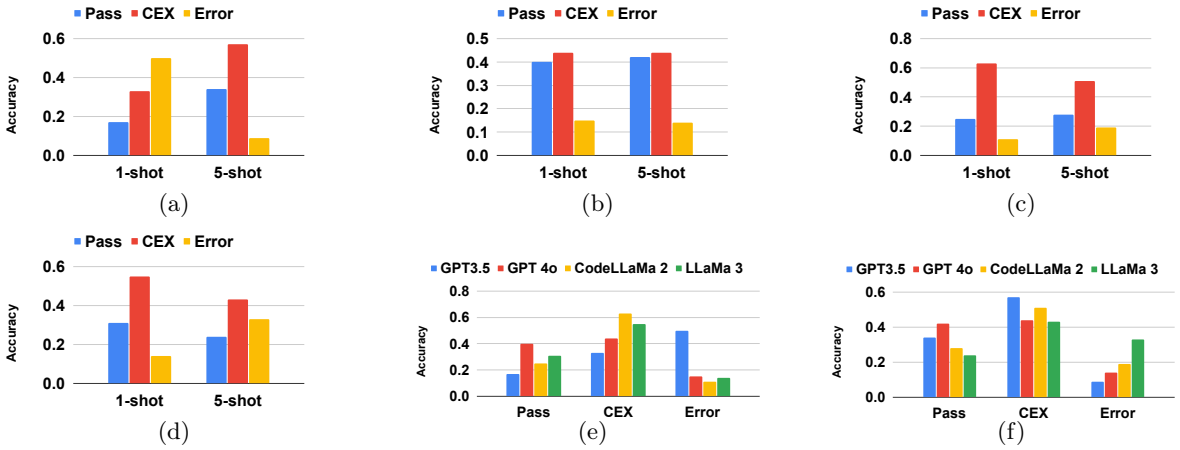

Figure 3: **Comparison of accuracy of generated assertions**. (a) Assertion accuracy comparison for GPT-3.5. (b) Assertion accuracy comparison for GPT-4o. (c) Assertion accuracy comparison for CodeLLaMa 2. (d) Assertion accuracy comparison for LLaMa3-70B. (e) $k = 1$-shot assertion accuracy. (f) $k = 5$-shot assertion accuracy. **CEX**: Counter Example trace.

(c.f., Figure 3a), with increase in the number of IC examples, the LLM was able to produce more syntactically correct assertions, however, majority of corrected assertions (on average up to 24%) generated a CEX when verified with JG. For GPT-4o, the results were more consistent in terms of syntactically correct and failing assertions from 1-shot and 5-shot learning (c.f., Figure 3b). For CodeLLaMa 2 and LLaMa3-70B, with increase in the number of IC examples, the fraction of failed assertions decreased (on average up to 12% for CodeLLaMa 2 and LLaMa3-70B, c.f., Figure 3c and Figure 3d), however, both models generated more syntactically wrong assertions (on average up to 19% more for LLaMa3-70B). Our analysis shows that with 1-shot, the variation in types of assertions in the IC examples were limited. Consequently, LLaMa3-70B learned the syntax. However, in 5-shot learning, we have more variations in assertion syntax which made LLaMa3-70B's learning task difficult. This experiment suggests that increasing the IC examples will not necessarily improve LLM's consistency in generating passing, failing, and syntactically correct assertions.

Observation 3: **GPT-4o is relatively more consistent and superior for assertion generation task** (c.f., Figure 3e-3f). Our experiments show that GPT-4o generates on an average up to 15.6% more valid assertions as compared to other LLMs for both 1-shot and 5-shot learning. Additionally, GPT-4o produced less CEX generating assertions and syntactically incorrect assertions as compared to other LLMs. This experiment shows that GPT-4o is more beneficial for asertion generation as compared to the other LLMs.

Observation 4: **All LLM models need considerable improvement for assertion generation task** (c.f., Figure 3). Our analysis show that none of the LLM models can generate valid assertions on average of no more than 44% accuracy whereas up to 63% generated assertions produces CEX and on average up to 33% of generated assertions are syntactically wrong. Clearly, for LLMs to be of practical usage for any realistic industrial-scale design, considerable improvement needs to be made. Specifically, the LLMs need to capture the semantic meaning of the Verilog designs for automatically producing higher fraction of valid assertions without iterative human prompting.

5 Conclusion and Future Work

This work introduces AssertionBench to evaluate the current and future commercial and open-source LLMs for the assertion generation task. Although there is no LLM that consistently outperform other LLMs, we notice several promising trends and research direction to enhance practical applicability of LLMs for assertion generation task which will further accelerate SoC and hardware design verification. As LLM research is growing at a tremendous pace both in commercial and academic research, we plan to maintain the benchmark and augment its learning and test set with more complex designs and their formally verified assertions to further stress test the LLMs.

6 Limitations

We identify the following limitations of this work in terms of the dataset and the evaluation methodology.

- **Dataset**: In the scope of this study, our primary focus is on Verilog designs, given its status as the predominant hardware design language. Moving forward, it will be intriguing to develop benchmarks for assertions in alternative hardware languages, *e.g.*, VHDL, SystemVerilog, SystemC, thereby expanding the scope of our analysis to encompass a broader range of design paradigms. Additionally, AssertionBench considers only a few temporal assertions with shallow temporality. It would be interesting to increase the temporal depth to capture design behaviors that cut across multiple clock cycles and evaluate LLM's ability to learn and generate assertions to capture such behaviors succinctly.

- **Assertion Objective**: In this work, we primarily focused on the assertions that capture design functionality. It would be interesting to enhance and augment AssertionBench with security assertions to evaluate the LLM's ability to capture and summarize security violations/concerns from hardware design source code.

- **Quantitative Assertion Ranking**: In this work, we primarily focused on correctness of an assertions without quantifying and ranking the subtlety of the captured design behavior (Pal et al., 2020). It would be interesting to include such ranking in the IC learning and evaluate LLM's capability to automatically rank generated assertions to quantify captured design behavior.

- **Modeling**: In this paper, we assessed the few-shot assertion generation capabilities of SOTA language models. In future work, it will be interesting to fine-tune language models for assertion generation and evaluate their performance on AssertionBench.

- **Evaluation**: In future work, it will be valuable to conduct a more detailed evaluation of model errors to better understand the specific limitations of each LLM for assertion generation.

References

Google Gemini. https://gemini.google.com/app. Accessed: October 16, 2024.

Viraj Athavale, Sai Ma, Samuel Hertz, and Shobha Vasudevan. 2014. Code Coverage of Assertions Using RTL Source Code Analysis. *Design Automation Conf. (DAC).*

Avinash Ayalasomayajula, Nusrat Farzana, Debjit Pal, and Farimah Farahmandi. 2024. Prioritizing Information Flow Violations: Generation of Ranked Security Assertions for Hardware Designs. *IEEE Int'l Symp. on Hardware Oriented Security and Trust (HOST).*

Saddek Bensalem, Yassine Lakhnech, and Hassen Saidi. 1996. Powerful Techniques for The Automatic Generation of Invariants. *Int'l Conf. on Computer-Aided Verification (CAV).*

Cadence. 2024. JasperGold Apps. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html. Accessed: October 16, 2024.

Michel Caplain. 1975. Finding Invariant Assertions for Proving Programs. *Int'l Conf. on Reliable Software (ICRS).*

Po-Hsien Chang and Li.-C Wang. 2010. Automatic Assertion Extraction via Sequential Data Mining of Simulation Traces. *Asia and South Pacific Design Automation Conf. (ASP-DAC).*

Chih-Neng Chung, Chia-Wei Chang, Kai-Hui Chang, and Sy-Yen Kuo. 2011. Applying Verification Intention for Design Customization via Property Mining Under Constrained Testbenches. *Int'l Conf. on Computer Design: VLSI in Computers and Processors, (ICCD).*

Alessandro Danese, Nicolò Dalla Riva, and Graziano Pravadelli. 2017. A-TEAM: Automatic Template-based Assertion Miner. *Design Automation Conf. (DAC).*

Albert Danial. 2021. cloc: v1.92.

Andrew DeOrio, Adam B. Bauserman, Valeria Bertacco, and Beth C. Isaksen. 2009. Inferno: Streamlining Verification With Inferred Semantics. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD).*

Calvin Deutschbein, Andres Meza, Francesco Restuccia, Ryan Kastner, and Cynthia Sturton. 2021. Isadora: Automated Information Flow Property Generation for Hardware Designs.

Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. 2000. Quickly Detecting Relevant Program Invariants. *Int'l Conf. on Software Engineering (ICSE).*

Wenji Fang, Mengming Li, Min Li, Zhiyuan Yan, Shang Liu, Hongce Zhang, and Zhiyao Xie. 2024. AssertLLM: Generating and Evaluating Hardware Verification Assertions from Design Specifications via Multi-LLMs. *arXiv.*

Samuele Germiniani and Graziano Pravadelli. 2022a. Exploiting Clustering and Decision-Tree Algorithms to Mine LTL Assertions Containing Non-boolean Expressions. *IFIP/IEEE Int'l Conf. on Very Large Scale Integration (VLSI-SoC).*

Samuele Germiniani and Graziano Pravadelli. 2022b. HARM: A Hint-Based Assertion Miner. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD).*

Tara Ghasempouri and Graziano Pravadelli. 2015. On The Estimation of Assertion Interestingness. *IFIP/IEEE Int'l Conf. on Very Large Scale Integration (VLSI-SoC).*

GoldMine. 2024. GoldMine: An Automatic Assertion Generation Tool. http://goldmine.csl.illinois.edu/. Accessed: October 16, 2024.

Sudheendra Hangal, Sridhar Narayanan, Naveen Chandra, and Sandeep Chakravorty. 2005. IODINE: A Tool to Automatically Infer Dynamic Invariants for Hardware Designs. *Design Automation Conf. (DAC).*

Mohammad Reza Heidari Iman, Jaan Raik, Maksim Jenihhin, Gert Jervan, and Tara Ghasempouri. 2021. A Methodology for Automated Mining of Compact and Accurate Assertion Sets. *IEEE Nordic Circuits and Systems Conference (NorCAS).*

Mohammad Reza Heidari Iman, Jaan Raik, Gert Jervan, and Tara Ghasempouri. 2022. IMMizer: An Innovative Cost-Effective Method for Minimizing Assertion Sets. *Euromicro Conference on Digital System Design (DSD).*

A. Hekmatpour and A. Salehi. 2005. Block-based Schema-driven Assertion Generation for Functional Verification.

Samuel Hertz, Debjit Pal, Spencer Offenberger, and Shobha Vasudevan. 2019. A Figure of Merit for Assertions in Verification. *Asia and South Pacific Design Automation Conf. (ASP-DAC).*

Samuel Hertz, David Sheridan, and Shobha Vasudevan. 2013. Mining Hardware Assertions With Guidance From Static Analysis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD).*

HuggingFace. 2024. Model Repository. https://huggingface.co/. Accessed: October 16, 2024.

Rahul Kande, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Shailja Thakur, Ramesh Karri, and Jeyavijayan Rajendran. 2023. LLM-assisted Generation of Hardware Assertions. *arXiv.*

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. *arXiv*.

Jan Malburg, Tino Flenker, and Görschwin Fey. 2017. Property Mining Using Dynamic Dependency Graphs. *Asia and South Pacific Design Automation Conf. (ASP-DAC)*.

Bhabesh Mali, Karthik Maddala, Sweeya Reddy, Vatsal Gupta, Chandan Karfa, and Ramesh Karri. 2024. ChIRAAG: ChatGPT Informed Rapid and Automated Assertion Generation. *arXiv*.

Meta. 2024. Introducing Meta Llama 3: The most capable openly available LLM to date. https://ai.meta.com/blog/meta-llama-3/. Accessed: October 16, 2024.

Jayadev Misra. 1977. Prospects and Limitations of Automatic Assertion Generation for Loop Programs. *SIAM Journal on Computing*.

NCSA. 2024. NCSA Delta. https://www.ncsa.illinois.edu/research/project-highlights/delta/. Accessed: October 16, 2024.

OpenAI, Josh Achiam, and et al. 2024. GPT-4 Technical Report. *arXiv*.

OpenCores. 2024. https://opencores.org/. Accessed: October 16, 2024.

Marcelo Orenes-Vera, Margaret Martonosi, and David Wentzlaff. 2023. Using LLMs to Facilitate Formal Verification of RTL. *arXiv*.

Debjit Pal, Spencer Offenberger, and Shobha Vasudevan. 2020. Assertion Ranking Using RTL Source Code Analysis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*.

Corina S. Păsăreanu and Willem Visser. 2004. Verification of Java Programs Using Symbolic Execution and Invariant Generation. *Int'l SPIN Workshop on Model Checking of Software (SPIN)*.

Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can Large Language Models Reason About Program Invariants? *Int'l Conf. on Machine Learning (ICML)*.

G. Pinter and I. Majzik. 2005. Automatic Generation of Executable Assertions for Runtime Checking Temporal Requirements.

Amir Pnueli. 1977. The Temporal Logic of Programs. *Annual Symp. on Foundations of Computer Science (SFCS)*.

Mohammad Reza Heidari Iman, Gert Jervan, and Tara Ghasempouri. 2024. ARTmine: Automatic Association Rule Mining with Temporal Behavior for Hardware Verification. *Design, Automation, and Test in Europe (DATE)*.

Frank Rogin, Thomas Klotz, Gorschwin Fey, Rolf Drechsler, and Steffen Rulke. 2008. Automatic Generation of Complex Properties for Hardware Designs. *Design, Automation, and Test in Europe (DATE)*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code.

SystemVerilog. 2024. 1800-2017 - IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. https://ieeexplore.ieee.org/document/8299595. Accessed: October 16, 2024.

A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. 2001. A Technique for Invariant Generation. *Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *arXiv*.

L.-C. Wang, M.S. Abadir, and N. Krishnamurthy. 1998. Automatic Generation of Assertions for Formal Verification of PowerPC/sup TM /microprocessor Arrays Using Symbolic Trajectory Evaluation. *Design Automation Conf. (DAC)*.

Hasini Witharana, Aruna Jayasena, Andrew Whigham, and Prabhat Mishra. 2023. Automated Generation of Security Assertions for RTL Models. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*.

Hasini Witharana, Yangdi Lyu, Subodha Charles, and Prabhat Mishra. 2022. A Survey on Assertion-based Hardware Verification. *ACM Comput. Surv. (CS)*.

Junjie Ye, Xuanting Chen, Nuo Xu, Can Zu, Zekai Shao, Shichun Liu, Yuhan Cui, Zeyang Zhou, Chao Gong, Yang Shen, Jie Zhou, Siming Chen, Tao Gui, Qi Zhang, and Xuanjing Huang. 2023. A Comprehensive Capability Analysis of GPT-3 and GPT-3.5 Series Models. *arXiv*.

```
1  module arb2(clk, rst, req1, req2,
2  gnt1, gnt2);
3  input clk, rst, req1, re2;
4  output gnt1, gnt2;
5  reg gnt_, gnt1, gnt2;
6  always @(posedge clk or posedge rst
     )
7    if(rst)
8       gnt_ <= 0;
9    else
10      gnt_ <= gnt1;
11 always @(*)
12   if (gnt_)
13   begin
14      gnt1 = req1 & req2;
15      gnt2 = req2;
16   end
17   else
18   begin
19     gnt1 = req1;
20     gnt2 = req2 & ~req1;
21   end
22 endmodule
```

Figure 4: **A Verilog code for a 2-port Arbiter** (Pal et al., 2020).

A Appendix

A.1 Background

In this section, we provide a few definitions necessary to understand assertion generation and AssertionBench. For our discussion, we consider a Verilog RTL design $\mathcal{D}$. $\mathcal{D}$ is a parallel composition of a set of concurrent processes P_i (`always` and `assign` blocks), and let V be the set of design signals. Let $\mathcal{I} \subset V$ be the set of input signals, $\mathcal{O} \subset V$ be the set of output signals, and $\mathcal{R} \subset V$ be the set of internal registers. In Figure 4, we show a Verilog design $\mathcal{D}$ of a 2-port Arbiter, consisting of two concurrent `always` processes P_1 (line 6) and P_2 (line 11), and $V = \{clk, rst, req1, req2, gnt1, gnt2, gnt_\}$, $\mathcal{I} = \{clk, rst, req1, req2\}$, $\mathcal{O} = \{gnt1, gnt2\}$, and $\mathcal{R} = \{gnt_\}$.

We define an ***assertion*** in a restricted manner as that of (Athavale et al., 2014). An assertion is a temporal formula in LTL (Pnueli, 1977) of the format $P = \mathcal{G}(A \rightarrow C)$ where the antecedent A is of the form $A = A_0 \wedge \mathcal{X}(A_1) \wedge \mathcal{X}\mathcal{X}(A_2) \wedge \ldots \wedge \mathcal{X}^m(A^m)$ and consequent C is of the form $C = \mathcal{X}^n(C^n)$, where $n \geq m$. Each A_i (C_i) is a proposition and is a (v, val) pair where $v \in V$ and $val \in \{0, 1\}$. $\mathcal{X}$ is called the next-cycle operator and $\mathcal{X}^n (n \geq 0)$ is equal to a delay of n clock cycles $\underbrace{\mathcal{X}\mathcal{X}\ldots\mathcal{X}}_{n \text{ times}}$. We say an assertion P is True if $\mathcal{D} \models P$ (read as $\mathcal{D}$ models P), in other words the assertion holds True globally (indicated by $\mathcal{G}$ operator) on design $\mathcal{D}$. Otherwise, the assertion is False, *i.e.*, $\mathcal{D} \not\models P$ and there exists a Boolean value assignment to design signals known as *cex* (counter example) that shows a refutation of asssertion P on $\mathcal{D}$. The implication operator $\rightarrow$ in P could be of two types, *overlapped* and *non-overlapped*. The *overlapped* implication operator denoted as $\rightarrow$ implies *if there is a match on the antecedent A, then the consequent C is evaluated in the same clock cycle.* The *non-overlapped* implication operator denoted as $\Rightarrow$ implies *if there is a match on the antecedent A, then the consequent C is evaluated in the next clock cycle.* Additionally, based on the evaluation condition of the pre-condition and the post-condition, an assertion can have different evaluation. In Figure 5, we show the assertion evaluation status. Note $A \rightarrow C$ can be re-written as $\neg A \vee C$. Consequently, if pre-condtion A is **unreachable** (or False), then the assertion P is vacuously True (*i.e.*, $\neg False \vee C$). If pre-condition A is True and post-condition C is True as well, the assertion is reported to be Valid (*i.e.*, $\mathcal{D} \models P$), otherwise, if the post-condition C is False, then a counter example *cex* is generated.

		Pre-condition	
		Covered	Unreachable
Post-Condition	True	Valid	Vacuous
	False	CEX	

Figure 5: **Assertion status based on pre-condition and post-condition evaluation**. **CEX**: Counter example.

For the Arbiter design of Figure 4, consider two assertions $P1 : \mathcal{G}((req1 == 1 \wedge req2 == 0) \rightarrow (gnt1 == 1))$ and $P2 : \mathcal{G}((req2 == 0 \wedge gnt_ == 1) \wedge \mathcal{X}(req1 == 1) \rightarrow \mathcal{X}(gnt1 == 1))$. The assertion $P1$ will evaluate to True if $req1$ is `1'b1` and $req2$ is `1'b0` at the current clock cycle, then $gnt1$ is `1'b1` in the current clock cycle. Note that $P1$ contains the *overlapped* implication operator $\rightarrow$. The assertion $P2$ will evaluate to True if $req2$ is `1'b0` and $gnt_$ is `1'b1` in the current cycle, $req1$ is `1'b1` in the next cycle, then $gnt1$ is

`1'b1` in the cycle after (*i.e.*, in the 2nd cycle). Note that $P2$ can be re-written using the *non-overlapped* implication operator in the following way $P2 : \mathcal{G}((req2 == 0 \wedge gnt_ == 1) \wedge \mathcal{X}(req1 == 1) \Rightarrow (gnt1 == 1))$ where the $\Rightarrow$ subsumes the $\mathcal{X}$ operator in the consequent. When we try to discharge a proof for $P1$ and $P2$ using a formal property verification (FPV) engine, such as Cadence JasperGold (Cadence, 2024), we find $P1$ is a valid assertion, *i.e.*, $\mathcal{D} \models P1$ whereas JasperGold finds a counterexample (CEX) for $P2$, *i.e.*, $\mathcal{D} \not\models P2$. We will use the assertion evaluation criterion of Figure 5 to define our *goodness* metric while evaluating SOTA LLMs for assertion generation task.

A.2 Related Work

Automatic generation of assertions in software and hardware has been an active area of research since seventies. Assertion generation via static analysis of design source code has been studied in the context of deductive program verification (Bensalem et al., 1996; Caplain, 1975; Misra, 1977). Static analysis techniques have been used to learn and generate likely program invariants for software verification (Tiwari et al., 2001; Păsăreanu and Visser, 2004). Data Mining and data-driven dynamic analysis have also been used to generate software invariants. DAIKON is one of the earliest and pioneering work on generating likely program invariants for software programs (Ernst et al., 2000). IODINE is one of the earliest work for hardware assertion generation by analyzing dynamic program behavior with respect to a set of standard property templates (Hangal et al., 2005). Prior works (Pinter and Majzik, 2005; Wang et al., 1998; Hekmatpour and Salehi, 2005) have used static analysis for assertion generation whereas (Rogin et al., 2008; DeOrio et al., 2009; Chang and Wang, 2010; Chung et al., 2011) have used dynamic simulation execution data for assertion generation. GoldMine introduced the novel concept of data-driven statistical analysis guided by the lightweight static analysis of design source code to automatically generate assertions for a broad spectrum of hardware designs (GoldMine, 2024; Hertz et al., 2013). Following GoldMine, researchers have developed a wide variety of assertion generation techniques using a combination of lightweight analysis of design source code and data-driven dynamic analysis targeting design functionality (Hertz et al., 2019; Pal et al., 2020; Malburg et al., 2017; Reza Heidari Iman et al., 2024; Heidari Iman et al., 2021; Danese et al., 2017; Germiniani and Pravadelli, 2022a; Heidari Iman et al., 2022; Germiniani and Pravadelli, 2022b) and design security (Deutschbein et al., 2021; Witharana et al., 2023). Additionally, researchers have proposed approaches to evaluate the quality of numerous assertions that automatic methods generate in terms of captured design functionality and security to induce a rank order to aid the downstream verification tasks (Ayalasomayajula et al., 2024; Pal et al., 2020; Heidari Iman et al., 2022; Ghasempouri and Pravadelli, 2015). ***However, all these works suffer from following shortcomings*** – they (i) do not scale for industrial-scale designs, (ii) often require massive amount of trace data to generate properties, (iii) generate too many design properties, often redundant, without any explanation on their usability for verification tasks, (iv) fail to generalize the properties beyond what is seen in the trace, and (v) encompass an extremely small subset of SystemVerilog (SVA) grammar (SystemVerilog, 2024), limiting the expressibility and richness of the generated design properties.

Recently, massive success of Large-Language Models (LLMs), *e.g.*, GPT (OpenAI et al., 2024), LLaMa (Touvron et al., 2023; **?**), Gemini (gem), in diverse applications have led researchers to investigate application of LLMs for hardware property generation (Liu et al., 2023; Orenes-Vera et al., 2023; Kande et al., 2023; Fang et al., 2024; Mali et al., 2024; Pei et al., 2023). ***However, almost all recent works on property generation using LLMs suffer from the following shortcomings*** – they (i) require considerable human efforts and deep understanding of the target hardware designs to devise *hand-crafted prompts* to generate and refine hardware properties, (ii) do not generalize the assertions, and (iii) do not consider execution traces, risking to potentially miss subtle incorrect design behaviors or security vulnerabilities that are not obvious in the design source code. In fact, there does not exist a systematic study in comparing the effectiveness of different commercial and open-source LLMs in generating valid assertions for an input

Verilog hardware design.

In this paper, we address this gap and propose AssertionBench to evaluate the goodness of four different widely-used LLMs on generating valid assertions. AssertionBench can be used to evaluate the assertion generation capabilities of any future LLMs as well.