VerifLLMBench: An Open-Source Benchmark for Testbenches Generated with Large Language Models

Nishanth Somashekara Murthy*, Eldon Nelson†, Sachin S. Sapatnekar*, John Sartori*
*Department of ECE, University of Minnesota, Twin Cities, USA
† Synopsys Inc., Sunnyvale, USA
Email: somas026@umn.edu, eldon.nelson@synopsys.com, sachin@umn.edu, jsartori@umn.edu

Abstract

The recent capabilities of generating RTL designs from natural language provide a possible glimpse into the future of design and coding. Benchmarking, used in the context of generative design and verification, offers a method to evaluate the effectiveness of a generator. A benchmark, especially one that is open source, well known and maintained, is a crucial part of the generative process. Previous work has explored the generation of RTL designs from natural language descriptions. Our paper seeks to extend this work by providing a benchmark and methodology to evaluate the effectiveness of UVM testbench generation. We present a structured methodology for benchmarking UVM testbench generation from natural language using state-of-the-art LLMs. Our approach incorporates verified DUTs, LLM-based UVM testbench generation, iterative refinement of prompts to resolve syntax errors, and thorough coverage analysis. The integration of UVM linting ensures that the generated testbenches meet industry standards, providing a solid framework to evaluate the quality and reliability of the LLM-driven verification code. We evaluate LLM performance based on build success rates, coverage metrics, and lint errors and warnings. Although the designs used to demonstrate coverage metrics are relatively simple, we demonstrate the challenges and gaps encountered by LLMs in generating robust UVM testbenches that provide high functional coverage. The benchmarking results highlight key limitations of LLM-generated UVM testbenches, emphasizing the need for refined methodologies and advanced training to enhance coverage, adaptability, and reliability in verification environments. Future advancements in LLMs, including those with larger context windows and UVM-specific training, hold significant potential to bridge these gaps and drive more robust and reliable generative solutions for design and verification.

Index Terms

Large-Language Model (LLM), Universal Verification Methodology (UVM), SystemVerilog, Design Under-Test (DUT), Test-bench (TB), Benchmarking

I. Introduction

Automated RTL generation based on natural language prompts has become feasible in recent years, and some recent works have focused on the use of Large Language Models (LLMs) to generate RTL designs [1], [2]. Verification of such designs requires the generation of a testbench (TB) that evaluates the quality of the generated RTL. Building upon the foundation of previous work on RTL generation, this paper leverages LLMs to synthesize UVM TBs and introduces a benchmarking methodology to evaluate their effectiveness. The proposed approach enables a multidimensional quality assessment of generated TBs, allowing a systematic comparison of different verification generators.

Our work is motivated by [1], which explores the use of LLMs to generate RTL. The authors of [2] supplement their RTL descriptions with handwritten implementations in SystemVerilog (SV) as well as simple, manually written TBs to verify them. Generative tools were used to assess the quality of the SV designs generated.

This paper presents a UVM TB benchmark that can be used to compare generative methods in a consistent way. Our work provides a benchmark suite under MIT-license, promoting a standard for evaluating generated UVM TBs. In both [1] and [2], the primary goal was to create a series of natural language descriptions with the objective of generating RTLs for various designs using an LLM. The goal of our work is to extend this concept to address the needs of verification engineers, who need a benchmark to evaluate the effectiveness of tools that generate UVM verification code.

II. Background

A. Universal Verification Methodology (UVM)

UVM offers a structured and reusable approach to design-verification through constrained random testing and coverage-driven methods, although it requires significant setup and expertise. In contrast, formal verification provides exhaustive, mathematically rigorous checks to prove properties across all scenarios, excelling in finding corner-case bugs but facing scalability issues with large designs. Unit testing, while simple and effective for early verification of individual modules, lacks the coverage and depth needed for full system interaction testing. A balanced verification strategy often combines UVM's comprehensive capabilities, formal methods for critical properties, and unit testing for early-stage validation. This work focuses on using LLMs to generate and benchmark UVM TBs for HDL designs.

B. Large Language Models

The selection of an LLM for the generation of UVM test benches is based on optimizing the relevance of training data, the efficiency of parameters, the computational feasibility, and the cost effectiveness. A model that aligns well across these dimensions offers a strategic advantage, enabling high-quality, standards-aligned TB generation that supports robust HDL design verification.

We consider the following LLMs.

1) **ChatGPT-4o (omni)** is optimized for conversational and multimodal tasks and offers fast response times and high versatility in both general-purpose and programming contexts. It leverages OpenAI's proprietary dataset with strong performance across a wide range of language tasks.
2) **Gemma2** [3] is an open source LLM developed by Google. At its largest, with $27\mathbb{B}$[1] parameters, it delivers benchmark performance that exceeds models more than twice its size.
3) **Gemini 1.5 Pro** is part of a new generation of multimodal models that integrate sparse and dense scaling alongside significant advancements in training, distillation and serving infrastructure that allow the model to excel in efficiency, reasoning, planning, multilingual capabilities, function calling, and long-context performance.
4) **Llama3.1** [4] is a highly parameterized transformer with $405\mathbb{B}$ parameters designed for deep learning research and adaptable for complex coding and language tasks. Developed by Meta, it is used primarily in academic and industrial research settings and is known for its customizable quantization.
5) **Llama3.2** is the latest model from Meta offering advanced image reasoning, multilingual text generation, and tool calling capabilities, with larger models excelling in visual-linguistic tasks and smaller models enabling private applications on the device.

A comparison of various LLMs is shown in Table I.

TABLE I: Comparison of LLMs

Metric	**ChatGPT-4o**	**Gemma2 [3]**	**Gemini 1.5 Pro**	**Llama 3.1 [4]**	**Llama 3.2**
License	Proprietary	Open	Proprietary	Open	Open
Parameters	$\sim$20 $\mathbb{B}$	27 $\mathbb{B}$	$\sim$200 $\mathbb{B}$	405 $\mathbb{B}$	90 $\mathbb{B}$
Context Window	128k	8k	1 M	128k	128k
Quantization	N/A	4-bit, 8-bit, 16-bit, 32-bit	N/A	4-bit, 8-bit	1-bit, 2-bit, 3-bit, 4-bit, 5-bit, 6-bit, 7-bit, 8-bit
Price (USD/1M tokens)	$4.38	$5.13	$2.19	$0.80	$4.50

Choosing an LLM for the task of generating UVM TBs for HDL designs depends on several metrics, as outlined below.

1) Licensing: Licensing plays a role in the selection of an LLM. Open-source models such as Google Gemma, Meta Llama3.2, and Llama3.1 provide greater flexibility for customization and fine-tuning, with permissive licenses that can be suitable for both commercial and noncommercial purposes. However, proprietary models such as OpenAI's ChatGPT4o or Google's Gemini 1.5 Pro often come with stricter usage terms and higher costs, which might restrict deployment in sensitive or resource-constrained settings. When choosing an LLM, it is important to align the licensing terms with the project budget, scope of usage, and compliance requirements.

2) Number of Parameters: Number of parameters is a primary determinant of an LLMs ability to capture intricate patterns in language, directly impacting the effectiveness in generating complex and structured output. For UVM TBs, a model with a substantial parameter count can provide the depth needed to interpret nuanced verification requirements, improving its capacity to generate detailed and contextually accurate TBs. However, while higher parameter counts generally yield greater precision, they also increase computational demands, necessitating a balance between model complexity and operational efficiency to ensure responsiveness and practicality.

3) Context Window: Context window determines how much information the model can process and "remember" within a single query or session. For UVM TB generation, a longer context window is critical to handle large HDL designs and intricate verification requirements without losing coherence or omitting crucial details. Models with long context windows, such as Google's industry-leading 1 Million tokens, can maintain continuity across complex interactions, allowing them to process detailed specifications and produce comprehensive TBs. However, longer context windows may also be susceptible to errors in the generated UVM TB, such as logical inconsistencies or misinterpretation of requirements, necessitating a balance between input length and success rate.

[1]$\mathbb{B}$ = Billion

4) Compute Power: The computational requirements of a model impact both inference time and responsiveness, particularly for resource-intensive LLMs with high parameter counts, directly affecting the feasibility of real-time or iterative TB generation. For large-scale or continuous verification needs, computing efficiency is paramount to maintain productivity while managing resource costs. In addition, fine-tuning an open-source LLM for UVM-specific tasks can further amplify computational demands, highlighting the need to match a model's requirements with available hardware resources.

For a given LLM having a parameter count P and quantization mode Q (4-bit, 8-bit, 16-bit, 32-bit), the amount of GPU memory M required to load the model is given by $M = \frac{P \times 4\ bytes}{32/Q} \times 1.2/10^9$, where 1.2 accounts for 20% overhead for common use cases of auxiliary operations, buffers, and inefficiencies encountered during training and inference. Table II shows the memory requirements for open source LLMs and is followed by examples of GPUs that can run one of the models.

TABLE II: GPU memory requirements (in GB)

	Models		
Quantization	**Gemma**	**Llama3.2**	**Llama3.1**
4	16.2	54	243
8	32.4	108	486
16	-	216	972
32	-	432	1944

Example of GPUs that can run Llama3.1 (405$\mathbb{B}$ parameters):

- 8 $\times$ AMD MI300 (192GB) GPUs in 16-bit mode.
- 8 $\times$ NVIDIA A100/H100 (80GB) GPUs in 8-bit mode.
- 4 $\times$ NVIDIA A100/H100 (80GB) GPUs in 4-bit mode.

5) Cost: Cost considerations encompass both licensing expenses and operational costs for compute resources. High-performance LLMs with extensive training often come with substantial licensing fees, while hardware costs to run large models add further financial implications. Balancing cost with projected efficiency gains of the model, such as reduced time to generate and debug TBs or higher verification quality, allows an informed assessment of return on investment for frequent UVM TB generation.

C. Prompt Engineering

Prompt engineering plays a crucial role in LLM performance optimization. Specifically, it focuses on crafting structured prompts to guide LLMs in creating correct, reusable UVM verification components. UVM relies on modular components like agents, drivers, sequences, etc. Prompt engineering enables LLMs to generate these components effectively by embedding UVM-specific terminology, structure, and best practices into the prompts, ensuring that generated code adheres to UVM's complex syntax, encouraging modularity and configurability, and aligning with verification needs for scalability and re-usability.

We use prompt refinement in this work, where an initial prompt is submitted to an LLM, and successive adjustments are made through prompts based on the LLM's responses until a satisfactory output is achieved. This approach can iteratively improve the quality and accuracy of responses, although it requires additional human interaction. An iterative feedback loop is used, using the model output as feedback. Subsequent prompts to the LLM are designed to correct errors or improve clarity and format. The cycle continues until the output meets the required standards, such as being free from syntax errors and complying with the desired format or content requirements.

III. Methodology

Benchmarking a design and benchmarking a TB take different approaches. To benchmark a design, the output response for a given input stimulus must be compared against the expected output, a common check performed by verification engineers. However, benchmarking a TB is a less familiar and more challenging ideology; this involves evaluating the quality of the stimulus and its ability to exercise the functionality of the Design Under Test (DUT). This task is not straightforward, in part because it involves proposing a solution to the recursive task of *testing the tester*.

A. Experimental Setup

To ensure consistency and emphasize the implementation of UVM TB components, a uniform starting point is provided for all LLM runs. Defining the interface and the top-level module in advance addresses several challenges. LLMs have limitations in managing extensive context, so skipping the implementation of these components allows more context capacity to generate and refine UVM TB, which is the primary focus of this work. By narrowing the scope to UVM class-based TB code, we enable direct comparison of TB generated by different LLMs. Without a standardized top-level module and DUT interface, variations would complicate functional coverage analysis. Hence, a stable predefined TB harness provides a reliable foundation for consistent performance evaluations.

We begin with a synthesized verified RTL model (1), i.e. the DUT as shown in Figure 1. The DUT is a verified RTL design taken directly from [1]. Beginning with a verified design enables us to isolate potential issues within the testbench or generator. An interface is created consisting of all I/O ports of the DUT. SystemVerilog cover-groups and cover-points are included to facilitate coverage analysis.

We considered five DUTs with varying complexity and design features. For each design, a TB wrapper, LLM TB prompt, and coverage-enabled interface were created.

1) `accu` - Accumulates 8-bit data and output after 4 inputs
2) `adder_8bit` - An 8-bit adder
3) `adder_16bit` - A 16-bit adder implemented with full adders
4) `fsm` - FSM sequence detection circuit
5) `alu` - An ALU for 32-bit MIPS-ISA CPU

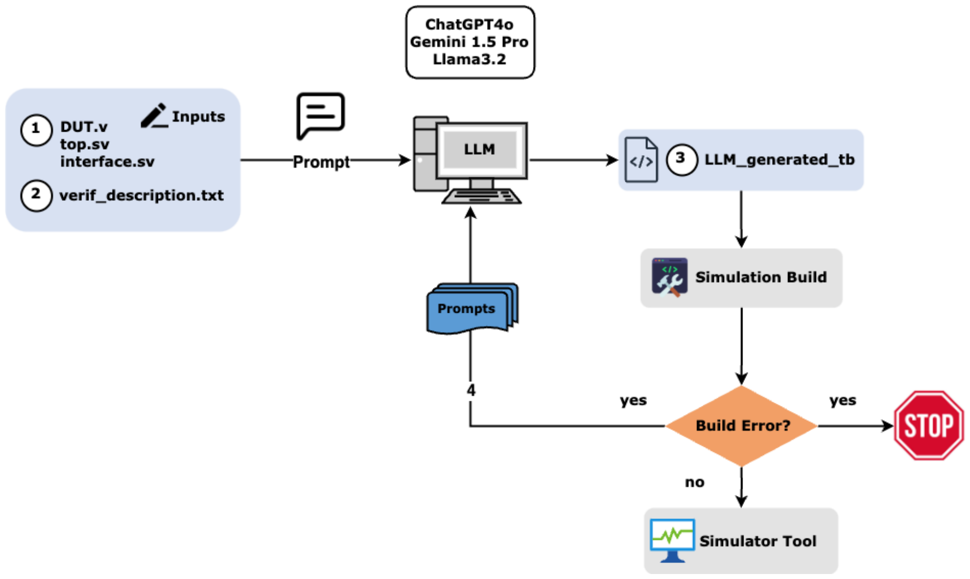

Fig. 1: LLM TB Generation to Build Flow

The top module instantiates the DUT and the interface, as shown in Fig. 2. An accompanying virtual interface facilitates seamless communication between the TB and the DUT. We take advantage of `uvm_config_db` to integrate the virtual interface into the UVM TB, which enhances the reusability and modularity of the test. The verification process includes waveform dumping for simulation traceability and the execution of a comprehensive UVM-based test suite.

```
// Declare the virtual interface

// Instantiate the Interface and pass it to the design

// DUT wrapper instantiation

initial begin
    // Set the interface as config object in UVM database

    $dumpfile("dump.vcd");
    $dumpvars;
    // Run Test
end
```

Fig. 2: `top.sv`

```
`include "uvm_macros.svh"
import uvm_pkg::*;

`ifdef GPTt1
    `include "../../_chatgpt4o/t1/accu_tb/accu_pkg.sv"
`elsif GPTt2
    `include "../../_chatgpt4o/t2/accu_tb/accu_pkg.sv"
`elsif GPTt3
    `include "../../_chatgpt4o/t3/accu_tb/accu_pkg.sv"
`elsif GPTt4
    `include "../../_chatgpt4o/t4/accu_tb/accu_pkg.sv"
`elsif GPTt5
    `include "../../_chatgpt4o/t5/accu_tb/accu_pkg.sv"
`endif
```

Fig. 3: `include.svh`

A header file (Fig. 3) is created consisting of all preprocessor macros and conditional compilation directives to include different versions of a package file, based on the LLM target and iteration defined during compilation.

Early experiments with LLM-based test generation revealed various issues in the prompts. The balance between explicit instructions and the possibility of LLM inference proved to be crucial. Omitting a comprehensive list of components or phases of UVM TB often led to incomplete implementations, such as generating a stimulus without a scoreboard for verification. Overly detailed prompts caused responses to truncate, highlighting processing limitations of the LLM with certain prompting styles. After extensive engineering efforts, a general framework for the description of natural language (2) was created, as shown in Fig. 4.

```
Please act as a professional verification engineer.

Verilog design of Accumulator (DUT) is defined as:
// Brief description of DUT functionality

// DUT I/O ports

// Interface
accu_if() Interface is defined.

Instance of the interface, and the virtual interface is set in the TOP module which wraps the DUT.

Utilize the existing top module and interface to handle signal interactions.

Required:

// brief description of expectations from each class, ex: constructors, build_phase, connect_phase, and run_phase
Transaction class
Sequence class
Sequencer class
Driver classs
Monitor class
Agent class
Scoreboard class
Environment class
Test class

Make sure the connections between the sequencer, driver, monitor, and scoreboard are correctly established.

Give me the complete code.
```

Fig. 4: `verif_description.txt` simplified outline

B. LLM testbench Generation

The `verif_description.txt` prompts the LLM to generate various components of a UVM TB. The prompts include

- A brief description of the DUT behavior is used to guide the LLM to develop a reference model of the DUT in the scoreboard.
- I/O ports of the DUT, which is used to model transactions packets.
- UVM phasing requirements (e.g. *build_phase*, *connect_phase*, *run* etc.)
- TLM interface ports and directionality.

From the state-of-the-art LLMs listed in Table I, we selected three – ChatGPT-4o, Gemini 1.5 Pro, and Llama3.2 – based on considerations of cost efficiency and computational resource constraints. All LLM runs were performed on the cloud using the chat-based feature. All LLMs are run without memory, ensuring that the response on each iteration is independent and is not influenced by previous iterations.

C. Compilation (Build) Process

The LLM-generated TB, along with the provided TB components (interface and top), is compiled using Synopsys VCS® (2022.06). Upon a compilation fail due to syntax errors, an iterative correction cycle is initiated. During each iteration, prompts are crafted to guide the LLM towards generating a TB (3) free of syntax errors, and the compilation process is repeated. Iterative refinement is arbitrarily capped at four attempts. If syntax errors persist beyond four iterations, the process is deemed unsuccessful.

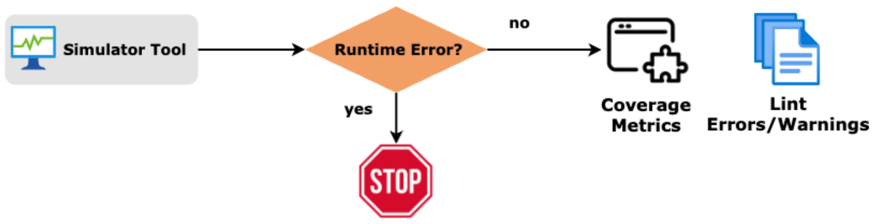

Fig. 5: Simulation - Coverage Collection, Lint Report Flow

D. Simulation and Coverage Collection

Coverage collection is enabled during both the build and simulation phases to collect coverage data, including line, condition, toggle, finite state machine (FSM), branch, and functional coverage (covergroups and coverpoints). We include cover-groups and cover-points in the interface, as shown in Fig 6. Upon a successful build, the simulation process is initiated as shown in Fig 5. If a simulation results in a runtime error, it is indicated but excluded from coverage analysis to maintain the accuracy of the results; only data from successful simulation runs are analyzed, allowing for the generation of precise coverage reports. The Unified Report Generator (URG) consolidates coverage data from multiple runs and produces HTML reports for review. By merging data from various simulation runs, a more comprehensive and insightful coverage analysis is achieved.

```
interface accu_if(input logic clk, input logic rst_n);

    // Declare I/O ports

    // Declare cover-group
    covergroup cg_inputs () @ (posedge clk && rst_n);
        // Add cover-point 1

        // Add cover-point 2
        .
        .
        .

    endgroup

    cg_inputs cg_inputs_inst = new();

endinterface
```

Fig. 6: `interface.sv`

E. UVM Linting

UVM linting is performed using Synopsys Euclide® (2024.09) to identify coding inconsistencies, usage violations, and improper phase management, to mitigate errors in complex verification environments. This process, commonly known as code linting, plays a crucial role in ensuring the reliability and maintainability of verification code. By integrating Euclide into the verification workflow, we gain a quantitative measure of code quality, enabling the evaluation of whether generated TBs align with industry standards and best practices.

Rules are applied to lint only the LLM-generated UVM code and not to the DUT, interface, or top. For clarity and simplicity, the counts of *Fatal* and *Error* – both representing critical levels of lint severity – are combined under a single category labeled Errors, which serves as the primary metric for our analysis. This is reported along with the count of warnings.

The entire process, from build to coverage collection and linting, is automated through the use of Makefiles. This approach improves workflow efficiency and simplifies the execution of tasks on different LLMs, ensuring consistent and reproducible runs.

IV. Results and Conclusion

Table III presents the consolidated results of five individual test runs conducted on different DUTs for ChatGPT-4o, Gemini 1.5 Pro and Llama3.2. The column "Build" indicates the number of successful simulation builds following the LLM prompt refinement process to achieve the final result. For example, an entry of 5 signifies that the LLM-generated UVM TB produced a successful simulation build on every attempt.

TABLE III: Syntax, Coverage and Lint Metrics for Different Designs

Design	**ChatGPT-4o**			**Gemini 1.5 Pro**			**Llama3.2**		
	Build	Coverage	Lint Errors/Warnings	Build	Coverage	Lint Errors/Warnings	Build	Coverage	Lint Errors/Warnings
accu	3	80.1%	7/1	5	78.2%	0/0	3	55.1%	0/12
adder_8bit	5	58.2%	0/2	5	66.3%	8/5	5	42.0%	0/4
adder_16bit	5	52.9%	0/0	5	78.3%	4/5	5	73.9%	0/4
FSM	3	93.3%	0/27	2	91.1%	0/0	2	9.9%	0/14
ALU	5	68.6%	2/3	5	82.4%	23/4	1	52.9%	0/15
Average	84%	70.6%	1.8/6.6	88%	79.3%	7/2.8	64%	46.8%	0/9.8

The problems encountered during the development process mainly involved missing *connect_phase* methods in the UVM `agent` and `environment`, as well as incorrect use of variable types within the UVM scoreboard, leading to build errors. These problems were systematically addressed through iterative debugging and a careful examination of the code. Furthermore, a recurring source of runtime errors was due to improper handling of TLM ports in the `scoreboard`, where mismatches between the expected and actual port types led to functional inconsistencies. Resolving these issues required refining the interface connections and ensuring the correct use of TLM communication protocols.

The “Coverage” column represents the coverage score reported by Synopsys VCS®, which is an average of all nonzero values of line, conditional, toggle, FSM, and branch coverage. Since zero values are excluded from the calculation, this averaging method can obscure individual coverage metrics and potentially lead to misleading interpretations. The “Lint” column indicates the count of lint errors and warnings specific to the LLM-generated UVM code, offering a quantitative measure of its quality.

A. *Coverage Metrics*

A detailed analysis of individual coverage metrics is presented in Figure 7, evaluating the ability of the LLM-generated UVM TBs to go beyond syntax correctness and benchmarking their depth in achieving functional closure. For each DUT, the figure shows the minimum, maximum, and average coverage achieved by each LLM across the 5 simulation builds.

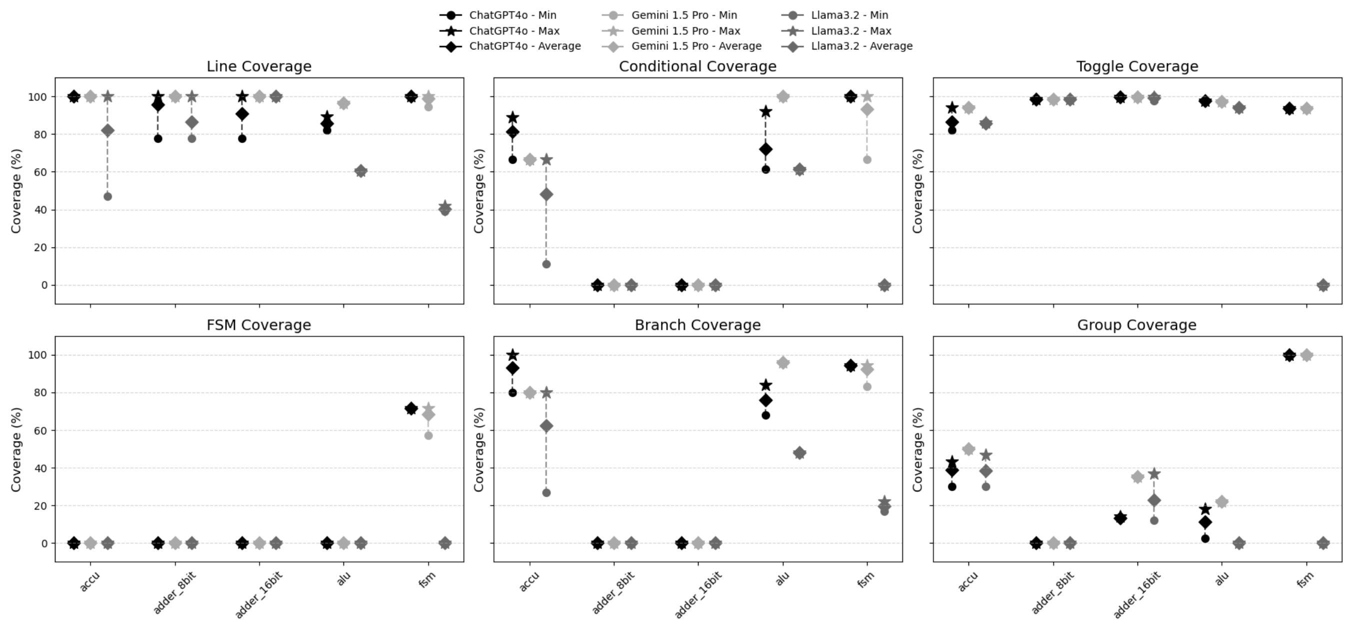

Fig. 7: Coverage Metrics across all LLMs and DUTs

1) Line: Line coverage achieved by LLM-generated TBs consistently outperforms other metrics, frequently reaching full or nearly full coverage. This suggests that the LLMs effectively generate TBs capable of exercising the majority of RTL. High line coverage highlights the LLMs’ strength in producing broad test scenarios that traverse most execution paths.

2) Conditional: The LLM-generated TBs show a gap between line and conditional coverage, emphasizing that while code execution is comprehensive, logical conditions may remain under-explored. For designs such as `adder_8bit` and `adder_16bit`, conditional coverage is 0% because these designs do not contain conditional logic. In benchmarking, it is important to note that no coverage for a metric may indicate the absence of that design feature, rather than a failure of the TB generator. For other designs, conditional coverage is relatively high in some cases, but in others, low coverage shows that the LLMs lack the contextual understanding to generate test scenarios that adequately exercise all logical conditions. Addressing this limitation could involve refining the LLM’s prompts or integrating additional knowledge about a design’s structure to encourage more diverse and precise branching tests using either constrained randomization or directed test-cases.

3) Toggle: Toggle coverage for the LLM-generated TBs closely aligns with line coverage, indicating that the LLMs are effective in generating stimuli that exercise individual bits of the design. For example, designs such as `alu` and `adder_16bit` achieve nearly perfect toggle coverage, showcasing the LLMs’ ability to excite low-level hardware mechanisms. However, the results reveal a limited scope. LLMs excel at handling straightforward execution paths but struggle with more intricate logic testing. To expand the scope of testing, enhancements in TB generation could aim to integrate more nuanced bit-level toggling that targets uncovered conditional or FSM elements.

4) FSM: Benchmarking indicates that FSM coverage may be the most challenging metric for LLMs. FSM coverage is not applicable for designs that lack FSM features (e.g., `accu`, `adder_8bit`, `adder_16bit`). For designs with FSM features, LLM-generated TBs struggle to create targeted stimuli to exercise all possible state transitions. This shortfall highlights a weakness in the current TB generation approach; it lacks mechanisms to recognize and systematically exercise FSMs. Future improvements could involve increasing understanding of FSM structures and improving the ability of LLMs to generate state-specific stimuli.

5) Branch: Branch coverage results reveal a pattern similar to conditional coverage, where LLM-generated TBs are unable to effectively explore some execution paths with branching logic. For example, `alu` demonstrates a significant gap between line coverage (96%) and branch coverage (76%), reflecting the difficulty in generating test cases that adequately evaluate all the branches of the design. This gap indicates a need for better alignment between the LLM's output and the structural and behavioral nuances of the design. Improving branch coverage would require more sophisticated TB generation that identifies and targets specific branches within the code.

6) Group: Group coverage, which relies on manually added cover-points and cover-groups, exhibits high variability and presents unique challenges for LLM-generated TBs. To demonstrate this variability, a total of 135,705 bins were created for the `adder_8bit` design, with only 0.18% and 0.20% of bins covered, on average, by the sequences generated by ChatGPT4o and Gemini 1.5 Pro, respectively. In contrast, 165 bins were created for the `adder_16bit` design, resulting in average group coverage of 22.7% and 33.5% from sequences generated by Gemini 1.5 Pro and Llama3.2, respectively. ChatGPT4o achieves 39% group coverage for the `accu` design, significantly higher than its FSM coverage, but lower than its line coverage.

This variability underscores a challenge for LLMs in systematically addressing manually defined bins. To improve group coverage, the TB generation process should prioritize incorporating explicit knowledge of cover-point definitions and intelligently targeting specific cover-groups. Evaluating TBs based solely on metrics such as group coverage highlights the strengths and limitations of different generators in addressing nuanced, manually defined criteria.

Assertion: Assertion coverage is inherently tied to the presence of assertions in the DUT, which are typically crafted by the designer to validate specific conditions or behaviors during simulation. In the DUTs evaluated, no assertions were implemented, so benchmarking shows 0% assertion coverage in all test cases. Without assertions defined in the DUT or added during testbench generation, this metric becomes irrelevant for evaluation.

B. LLM Performance

1) ChatGPT4o: ChatGPT4o outperforms other models in most coverage metrics, demonstrating good line, branch, and toggle coverage. It achieves near-perfect line coverage in most designs, with averages reaching 100% for `accu` and `fsm`. ChatGPT4o's ability to handle complex scenarios, such as the `fsm` design, with high branch and group coverage, makes it one of the most versatile and reliable LLMs in this comparison, often matching or exceeding the performance of the other models.

2) Gemini 1.5 Pro: Gemini shows strong performance in specific areas, particularly in toggle coverage, where it matches, slightly trails, or slightly exceeds ChatGPT4o for most designs. It also achieves consistent line coverage, maintaining 100% in simpler designs such as `accu` and `adder_8bit`. Despite slightly lower FSM and branch coverage compared to ChatGPT4o, Gemini shows promise in targeted scenarios, making it a solid but perhaps less comprehensive option.

With its huge input token size ($1M$), Gemini 1.5 Pro excels at understanding the complex requirements of UVM TBs. TBs are consistently generated with minimal variability across multiple runs, demonstrating the LLM's potential to achieve higher scores in coverage benchmarks through effective prompt engineering and iterative prompting. Gemini 1.5 Pro outperforms ChatGPT4o by a small margin in overall coverage and build convergence benchmarking.

3) Llama3.2: Llama3.2 exhibits inconsistent performance in all coverage metrics, with significant strengths in toggle coverage but notable weaknesses in others, particularly FSM and group coverage. It achieves perfect line coverage for designs like `adder_16bit` but falters with more complex scenarios, such as FSM, where performance drops drastically. Its low branch and conditional coverage highlight the challenges in handling intricate logic and decision-making scenarios, making it the least competitive of the three LLMs in this evaluation.

Llama3.2 attempts to improve coverage metrics by incorporating constraint randomization into its transactions. However, this approach hinders the convergence of the build, ultimately lowering the overall coverage scores.

These results indicate similar benchmark performance for Gemini 1.5 Pro and ChatGPT4o. Llama3.2 exhibits performance gaps in both build convergence and functional coverage, indicating the need for further refinement to handle complex scenarios effectively. The results do not reveal a clear correlation between the number of model parameters and their performance on the metrics evaluated.

C. Challenges

1) Functional - Correctness: During multiple test runs, the LLM-generated scoreboard frequently reported that the tests were failing, despite the fact that the DUT was fully verified. Often due to incorrect coupling of interface objects, resulting in high-impedance (z) values in the scoreboard. This underscores a limitation of the LLMs in generating scoreboards that accurately model the complex state-dependent behavior of DUTs. These challenges highlight the need for specialized methodologies to improve the precision and reliability of LLM-generated scoreboards in verification environments.

Additionally, LLMs face difficulties in interpreting nonstandard data formats, providing reliable error diagnostics, and adapting to evolving DUT specifications – all of which are essential for effective verification. These limitations underscore the need for further refinement of LLMs and UVM-specific model training to improve the accuracy and reliability of the LLM-based scoreboard to reflect DUT behavior.

SystemVerilog Assertions (SVA) are a powerful tool for verifying the behavior of a design and can be used to validate the correctness of the model's output in the `scoreboard` by specifying constraints and conditions that must hold throughout simulation. This ensures that the reference model is consistently checked against defined expectations, improving the reliability and accuracy of the scoreboard. Including a detailed description of the DUT's behavior significantly increases the number of tokens processed by the LLM, which can degrade the quality of the generated TB, due to the constraints of current context-window limits.

2) Functional - Coverage: Coverage metrics, such as FSM and group coverage, were very low across all designs. This highlights a critical issue with the current sequence generation methodology. Currently, TBs and sequences generated by an LLM, which, while efficient in automating initial TB development, remain rudimentary in terms of test strategy sophistication. Specifically, it lacks implementation of constrained randomization and does not include directed test cases.

Developing aspects such as restricted randomization and directed tests in the prompts can degrade output quality, often resulting in syntax errors and inconsistencies. This occurs because the LLM, while adept at pattern recognition and basic code generation, struggles to meet the nuanced requirements of complex test constructs without introducing flaws.

The advent of LLMs with larger context windows could alleviate both of these problems by enabling the model to handle extensive, detailed input descriptions without compromising output quality, thereby supporting the generation of more accurate and comprehensive TBs.

Several efforts have been made to achieve functional closure and sign-off using automated coverage-driven test adaptation [5], feedback loops [6], and Artificial Intelligence/Deep Reinforcement Learning [7], [8]. The achievement of functional closure summarizes the end-to-end flow of automation, that is, the generation of the TB to the functional sign-off proposed in this work.

D. Conclusion

Benchmarking provides valuable information in the strengths and limitations of UVM TB generated by LLM. Through detailed coverage analysis and performance evaluations, we learned that while LLMs can produce efficient and comprehensive TBs in terms of line and toggle coverage, they struggle with more complex aspects such as FSM, branch and conditional coverage. These challenges underscore current limitations in the generation of sophisticated test strategies, particularly when it comes to using intricate design features and adapting to evolving DUT specifications.

The results emphasize the need for further refinement in LLM-based TB generation, especially in the areas of constrained randomization and directed test cases. The coverage gaps in critical areas highlight the importance of improving the model's understanding of complex design behavior and the specific nuances of verification environments. Furthermore, the benchmarking process revealed that issues such as improper interface handling, incorrect variable types, and difficulties with TLM ports are recurring obstacles that must be addressed to improve the reliability and precision of LLM-generated TB.

Ultimately, the findings of this benchmarking effort stress the importance of careful integration and configuration of TB components. Although LLMs show promise in automating portions of the TB creation process, achieving full functional closure and sign-off requires more advanced methodologies and deeper model training. Prompt engineering can address the core issues by improving modularity and offering a flexible framework that supports human intervention, allowing users to guide and refine the process, ensuring that the system reaches its desired functionality and performance with greater precision and adaptability. The benchmark results suggest that future LLM advancements, particularly those with larger context windows and UVM-specific training, may mitigate current limitations and lead to more accurate, robust, and reliable verification solutions.

REFERENCES

[1] L. Yao, L. Shang, Z. Qijun, and X. Zhiyao, "RTLLM: An open-source benchmark for design rtl generation with large language model," *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024.
[2] L. Mingjie, P. Nathaniel, K. Brucek, and R. Haoxing, "VerilogEval: Evaluating large language models for Verilog code generation," *International Conference on Computer Aided Design (ICCAD)*, 2023.
[3] Google, Gemma Team, "Gemma 2: Improving open language models at a practical size," 2024.
[4] Meta, Llama team, "The Llama 3 herd of models," 2024.
[5] A. Nazi, Q. Huang, H. Shojaei, H. Asghari Esfeden, A. Mirhosseini, and R. Ho, "Adaptive test generation for fast functional coverage closure," USA, 2022.
[6] A. Vintila and S. Duda, "UVM-SV feedback loop – the foundation of self-improving testbenches," San Jose, USA, 2023.
[7] E. Ohana, "Closing functional coverage with deep reinforcement learning: A compression encoder example," San Jose, USA, 2023.
[8] S. Vadanaparthi, P. Ganesh, D. Mahay, and M. Ganai, "Accelerating functional verification through stabilization of testbench using AI/ML," San Jose, USA, 2023.

Appendix

```
`timescale 1ns/1ns
module accu(
    input               clk         ,
    input               rst_n       ,
    input       [7:0]   data_in     ,
    input               valid_in    ,
    output  reg         valid_out   ,
    output  reg [9:0]   data_out
);
   reg [1:0] count;
   wire add_cnt;
   wire ready_add;
   wire end_cnt;
   reg [9:0]   data_out_reg;

   assign add_cnt = ready_add;
   assign end_cnt = ready_add && (count == 'd3);

   //count
   always @(posedge clk or negedge rst_n) begin
       if(!rst_n) begin
          count <= 0;
       end
       else if(end_cnt) begin
          count <= 0;
       end
       else if(add_cnt) begin
          count <= count + 1;
       end
   end

    //data_out_reg
   always @(posedge clk or negedge rst_n) begin
      if(!rst_n) begin
        data_out_reg <= 0;
      end
      else if (add_cnt && count == 0) begin
          data_out_reg <= data_in;
      end
      else if (add_cnt) begin
          data_out_reg <= data_out_reg + data_in;
      end
   end

    //data_out
   always @(posedge clk or negedge rst_n) begin
      if(!rst_n) begin
        data_out <= 0;
      end
      else if (add_cnt && count == 0) begin
          data_out <= data_in;
      end
      else if (add_cnt) begin
          data_out <= data_out + data_in;
      end
   end

   //ready_add
   assign ready_add = !valid_out | valid_in;

   //valid_out
   always @(posedge clk or negedge rst_n) begin
       if(!rst_n) begin
           valid_out <= 0;
       end
       else if(end_cnt) begin
           valid_out <= 1;
       end
       else begin
           valid_out <= 0;
       end
   end
endmodule
```

Fig. 8: `accu.v`

```
interface accu_if(input logic clk, input logic rst_n);

  logic   [7:0]   data_in;
  logic           valid_in;
  logic           valid_out;
  logic   [9:0]   data_out;

  // input covergroup
  covergroup cg_inputs () @ (posedge clk && rst_n);
    data_in_stim: coverpoint data_in {
        bins max = { 2**8 - 1 };
        bins min = { 0} ;
        bins all[5] = { [0:2**8 - 1] };
    }
    valid_in_stim: coverpoint valid_in {
        bins max = { 1 };
        bins min = { 0 };
        bins all[] = { 0,1 };
    }
    full_stim: cross data_in_stim, valid_in_stim {
        bins all = binsof(data_in_stim.all) &&
        binsof(valid_in_stim.all);
    }
  endgroup
  cg_inputs cg_inputs_inst = new();
endinterface
```

Fig. 9: `accu_if.sv`

```
`include "uvm_macros.svh"
import uvm_pkg::*;

`ifdef Llamat1
    `include "../../_llama3/t1/accu_tb/accu_pkg.sv"
`elsif Llamat2
    `include "../../_llama3/t2/accu_tb/accu_pkg.sv"
`elsif Llamat3
    `include "../../_llama3/t3/accu_tb/accu_pkg.sv"
`elsif Llamat4
    `include "../../_llama3/t4/accu_tb/accu_pkg.sv"
`elsif Llamat5
    `include "../../_llama3/t5/accu_tb/accu_pkg.sv"
`elsif Geminit1
    `include "../../_gemini/t1/accu_tb/accu_pkg.sv"
`elsif Geminit2
    `include "../../_gemini/t2/accu_tb/accu_pkg.sv"
`elsif Geminit3
    `include "../../_gemini/t3/accu_tb/accu_pkg.sv"
`elsif Geminit4
    `include "../../_gemini/t4/accu_tb/accu_pkg.sv"
`elsif Geminit5
    `include "../../_gemini/t5/accu_tb/accu_pkg.sv"
`elsif GPTt1
    `include "../../_chatgpt4o/t1/accu_tb/accu_pkg.sv"
`elsif GPTt2
    `include "../../_chatgpt4o/t2/accu_tb/accu_pkg.sv"
`elsif GPTt3
    `include "../../_chatgpt4o/t3/accu_tb/accu_pkg.sv"
`elsif GPTt4
    `include "../../_chatgpt4o/t4/accu_tb/accu_pkg.sv"
`elsif GPTt5
    `include "../../_chatgpt4o/t5/accu_tb/accu_pkg.sv"
`endif
```

Fig. 10: `accu_includes.svh`

```
module top;
  import accu_pkg::*;

  // Declare clock and reset signals here and initialize them here
  bit clk, rst_n;

  // Clock generation
  always #5 clk = ~clk;

  // Reset generation
  initial begin
    rst_n = 0;
    #20 rst_n = 1;
  end

  virtual accu_if vif;
  // Instantiate the Interface and pass it to the design
  accu_if accu_if_inst (.clk(clk), .rst_n(rst_n));

  // DUT wrapper instantiation
  accu uut(
    .clk(accu_if_inst.clk),
    .rst_n(accu_if_inst.rst_n),
    .data_in(accu_if_inst.data_in),
    .valid_in(accu_if_inst.valid_in),
    .valid_out(accu_if_inst.valid_out),
    .data_out(accu_if_inst.data_out)
  );
 initial begin
    // Set the interface as config object in UVM database
    uvm_config_db #(virtual accu_if)::set(null, "*", "vif", accu_if_inst);
    $dumpfile("dump.vcd");
    $dumpvars;
    run_test("accu_test");
  end
endmodule
```

Fig. 11: `top.sv`

```
Please act as a professional verification engineer.

Verilog design of Accumulator (DUT) is defined as:
Achieve serial input data accumulation output, input is 8bit data. The valid_in will be set to 1 before the first data
↪  comes in. Whenever the module receives 4 input data, the data_out outputs 4 received data accumulation results and
↪  sets the valid_out to be 1 (will last only 1 cycle).

Module name:
    accu
Input ports:
        clk: Clock input for synchronization.
        rst_n: Active-low reset signal.
        data_in[7:0]: 8-bit input data for addition.
        valid_in: Input signal indicating readiness for new data.
Output ports:
    valid_out: Output signal indicating when 4 input data accumulation is reached.
    data_out[9:0]: 10-bit output data representing the accumulated sum.

accu_if() Interface is defined.

Instance of the interface, and the virtual interface is set in the TOP module which wraps the DUT.
Utilize the existing top module and interface to handle signal interactions.
Required:
Transaction class - Write a UVM sequence item to represent a transaction. All inputs should be randomized except clk and
↪  rst_n, outputs are just bits. Write macros for all i/O.
Sequence class - Write a UVM sequence that randomizes inputs, generates multiple transactions in a loop for high
↪  functional coverage.
Sequencer class
Driver classs - Write a UVM driver that takes transactions from the sequencer, assign values to inputs through the
↪  virtual interface. Use anlysis port to send transaction to scoreboard.
Monitor class - Write a UVM monitor that connects to the virtual interface. The monitor should sample the values of I/O
↪  during each transaction, package them into a UVM transaction object, and forward the data to an analysis port for
↪  scoreboard verification.
Agent class - write a UVM agent. it should include a sequencer, driver, monitor and the virtual interface.
Scoreboard class - Write a UVM scoreboard that should receive transactions from the monitor through an analysis port,
↪  compute the expected result, and compare it to the output from the interface. Report mismatches.
Environment class - Write a UVM environment class that includes the agent and the scoreboard. Connect the monitor to the
↪  environment. Connect the environment to the scoreboard.
Test class - Write a UVM test class, declare environment and sequence,
             build_phase - instantiate environment and sequence
             run_phase - phase.raise_objection, start_sequencer, phase.drop_objection

Make sure the connections between the sequencer, driver, monitor, and scoreboard are correctly established.
Give me the complete code.
```

Fig. 12: `verif_description.txt`

```systemverilog
// UVM Transaction Class
class accu_trans extends uvm_sequence_item;
  `uvm_object_utils(accu_trans)

  rand bit [7:0] data_in;
  bit valid_in;
  bit [9:0] data_out; // output from DUT, not randomized
  bit valid_out;

  function new(string name = "accu_trans");
    super.new(name);
  endfunction
endclass
```

Fig. 13: `accu_transaction.sv`

```systemverilog
// UVM Sequence Class
class accu_sequence extends uvm_sequence #(accu_trans);
  `uvm_object_utils(accu_sequence)

  function new(string name = "accu_sequence");
    super.new(name);
  endfunction

  virtual task body();
    accu_trans trans;
    repeat (10) begin
      // Generate multiple transactions for high
      ↪  functional coverage
      trans = accu_trans::type_id::create("trans");
      assert(trans.randomize());
      start_item(trans);
      finish_item(trans);
    end
  endtask
endclass
```

Fig. 14: `accu_sequence.sv`

```systemverilog
// UVM Monitor Class
class accu_monitor extends uvm_monitor;
  `uvm_component_utils(accu_monitor)

  virtual accu_if vif;
  uvm_analysis_port #(accu_trans) ap;

  function new(string name = "accu_monitor",
  ↪  uvm_component parent = null);
    super.new(name, parent);
    ap = new("ap", this);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (!uvm_config_db #(virtual accu_if)::get(this, "",
    ↪  "vif", vif))
      `uvm_fatal("NOVIF", "Virtual interface not found")
  endfunction

  virtual task run_phase(uvm_phase phase);
    accu_trans trans;
    trans = accu_transaction::type_id::create("trans");
    forever begin
      @(posedge vif.clk);
      trans.data_in = vif.data_in;
      trans.valid_in = vif.valid_in;
      trans.data_out = vif.data_out;
      trans.valid_out = vif.valid_out;
      ap.write(trans);
    end
  endtask
endclass
```

Fig. 15: `accu_monitor.sv`

```systemverilog
// UVM Sequencer Class
class accu_sequencer extends uvm_sequencer #(accu_trans);
  `uvm_component_utils(accu_sequencer)

  function new(string name = "accu_sequencer",
  ↪  uvm_component parent = null);
    super.new(name, parent);
  endfunction
endclass
```

Fig. 16: `accu_sequencer.sv`

```systemverilog
// UVM Driver Class
class accu_driver extends uvm_driver #(accu_trans);
  `uvm_component_utils(accu_driver)

  virtual accu_if vif;
  uvm_analysis_port #(accu_trans) ap;

  function new(string name = "accu_driver", uvm_component
  ↪  parent = null);
    super.new(name, parent);
    ap = new("ap", this);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (!uvm_config_db #(virtual accu_if)::get(this, "",
    ↪  "vif", vif))
      `uvm_fatal("NOVIF", "Virtual interface not found")
  endfunction

  virtual task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(req);
      vif.data_in <= req.data_in;
      vif.valid_in <= req.valid_in;
      @(posedge vif.clk);
      seq_item_port.item_done();
      ap.write(req);
    end
  endtask
endclass
```

Fig. 17: `accu_driver.sv`

```systemverilog
// UVM Agent Class
class accu_agent extends uvm_agent;
  `uvm_component_utils(accu_agent)

  accu_sequencer seqr;
  accu_driver drv;
  accu_monitor mon;
  virtual accu_if vif;

  function new(string name = "accu_agent", uvm_component
  ↪  parent = null);
    super.new(name, parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    seqr = accu_sequencer::type_id::create("seqr", this);
    drv = accu_driver::type_id::create("drv", this);
    mon = accu_monitor::type_id::create("mon", this);

    if (!uvm_config_db #(virtual accu_if)::get(this, "",
    ↪  "vif", vif))
      `uvm_fatal("NOVIF", "Virtual interface not found")

    drv.vif = vif;
    mon.vif = vif;
  endfunction

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    drv.seq_item_port.connect(seqr.seq_item_export);
    mon.ap.connect(drv.ap);
  endfunction
endclass
```

Fig. 18: `accu_agent.sv`

```
// UVM Scoreboard Class
class accu_scoreboard extends uvm_scoreboard;
  `uvm_component_utils(accu_scoreboard)

  uvm_analysis_imp #(accu_trans, accu_scoreboard) ap;

  function new(string name = "accu_scoreboard", uvm_component parent = null);
    super.new(name, parent);
    ap = new("ap", this);
  endfunction

  virtual task write(accu_trans trans);
    static bit [9:0] expected_sum = 0;
    static int count = 0;
    if (trans.valid_in) begin
      expected_sum += trans.data_in;
      count++;
    end

    if (count == 4) begin
      if (trans.valid_out && trans.data_out !== expected_sum) begin
        `uvm_error("SCOREBOARD", $sformatf("Mismatch! Expected: %0d, Got: %0d", expected_sum, trans.data_out))
      end else if (trans.valid_out) begin
        `uvm_info("SCOREBOARD", "Match found", UVM_LOW)
      end
      expected_sum = 0; count = 0;
    end
  endtask
endclass
```

Fig. 19: accu_scoreboard.sv

```
// UVM Environment Class
class accu_env extends uvm_env;
  `uvm_component_utils(accu_env)

  accu_agent agt;
  accu_scoreboard sb;

  function new(string name = "accu_env", uvm_component
  ↪  parent = null);
    super.new(name, parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    agt = accu_agent::type_id::create("agt", this);
    sb = accu_scoreboard::type_id::create("sb", this);
  endfunction

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    agt.mon.ap.connect(sb.ap);
  endfunction
endclass
```

Fig. 20: accu_environment.sv

```
// UVM Test Class
class accu_test extends uvm_test;
  `uvm_component_utils(accu_test)

  accu_env env;
  accu_sequence seq;

  function new(string name = "accu_test", uvm_component
  ↪  parent = null);
    super.new(name, parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env = accu_env::type_id::create("env", this);
    seq = accu_sequence::type_id::create("seq");
  endfunction

  virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    seq.start(env.agt.seqr);
    phase.drop_objection(this);
  endtask
endclass
```

Fig. 21: accu_test.sv