# Enhanced Automatic UVM Testbench Generator for RTL Design Verification Using Python

Preeti Metre
*Department of Electronics and Communication Engg.*
*Bangalore Institute of Technology,*
Bangalore, India
*Preetilaxman2001@gmail.com*

Dr. Niranjan E
*Associate Professor*
*Dept. of Electronics and Communication Engg.*
*Bangalore Institute of Technology*,
niranjan_e@bit-bangalore.edu.in

***Abstract-Automatic verification techniques have become important in modern VLSI functional verification due to increasing digital circuit complexity and the effort required for manual UVM testbench development. This paper presents an Enhanced Automatic UVM Testbench Generator (AUTG) using Python-based RTL parsing and System Verilog UVM methodology for efficient RTL design verification. The proposed framework automatically extracts DUT information from RTL files and generates reusable UVM components such as interface, driver, monitor, scoreboard, and testbench modules. Assertion-based verification and functional coverage techniques are implemented to improve bug detection capability and verification completeness. Full Adder, D Flip-Flop (DFF), and FIFO designs are verified using the generated UVM environment. Simulation results and waveform analysis confirm successful verification and correct DUT functionality. The proposed methodology reduces manual coding effort, improves verification productivity, and provides a reusable and scalable verification solution for different RTL designs. The framework also minimizes human errors and supports faster development of UVM-based verification environments for modern VLSI systems.***

***Keywords— UVM, RTL Verification, Python Automation, Functional Verification, Assertions, Functional Coverage, FIFO Verification, D Flip-Flop, Full Adder.***

## I. INTRODUCTION

The rapid growth of modern VLSI digital systems has significantly increased the complexity of hardware design and verification processes. Functional verification is one of the most important stages in the VLSI design cycle, as it ensures that the Design Under Test (DUT) operates according to the required specifications before fabrication. Verification consumes a major portion of the overall design time because engineers must create testbenches, generate test cases, analyze simulation results, and debug functional errors. Therefore, improving verification productivity and reducing development time have become important challenges in modern VLSI design.

Universal Verification Methodology (UVM) is a widely adopted verification standard used for creating reusable and scalable verification environments. UVM provides a structured verification approach using components such as sequence item, sequencer, driver, monitor, scoreboard, agent, and environment. These components help perform transaction-level verification and improve verification quality. However, manual development of UVM testbenches requires significant coding effort, verification expertise, and debugging time. The repetitive creation of UVM components for every new RTL design increases development complexity and reduces productivity.

To overcome these limitations, automation techniques are increasingly being used in RTL verification. Python provides a flexible and efficient platform for automating verification tasks such as RTL parsing, file generation, and verification environment creation. Automatic generation of UVM components can significantly reduce manual effort and improve reusability of verification environments. The integration of automation with UVM methodology enables faster verification development and better verification efficiency. This paper presents an Enhanced Automatic UVM Testbench Generator (AUTG) using Python-based RTL parsing and System Verilog UVM methodology. The proposed framework automatically extracts DUT information from RTL design files and generates reusable UVM verification components such as interface, sequence item, driver, monitor, scoreboard, and testbench modules. Assertion-based verification and functional coverage techniques are integrated to improve bug detection capability and verification completeness. The framework is validated using Full Adder, D Flip-Flop (DFF), and FIFO designs. Simulation results and waveform analysis confirm successful verification and demonstrate the effectiveness of the proposed automated verification framework for modern VLSI functional verification applications.

## LITERATURE REVIEW

The increasing complexity of VLSI digital systems has created a growing demand for efficient functional verification methodologies. Several researchers have proposed automated testbench generation techniques, reusable verification environments, and assertion-based verification methods to improve verification productivity and reduce manual effort. UVM has become one of the most widely adopted verification methodologies due to its scalability, reusability, and support for coverage-driven verification.

Various studies have explored automatic testbench generation using different programming languages and automation frameworks. Python-based verification techniques have gained significant attention because of their flexibility in RTL parsing, script generation, and automation of repetitive verification tasks. These approaches reduce coding complexity and improve verification efficiency by automatically generating verification components and test scenarios.Assertion-based verification and

functional coverage techniques have also been widely used to improve verification quality.

Assertions help detect design errors automatically during simulation, while functional coverage ensures that all important DUT scenarios and input combinations are exercised successfully. These techniques reduce debugging effort and increase confidence in design correctness.

Although several verification automation frameworks have been proposed, many existing approaches still require significant manual customization and have limited support for generating complete UVM verification environments. To address these limitations, the proposed Automatic UVM Testbench Generator (AUTG) integrates Python-based RTL parsing, automatic UVM component generation, assertions, and functional coverage into a unified framework. The proposed approach improves verification productivity, reduces manual coding effort, and supports verification of combinational, sequential, and memory-based RTL designs such as Full Adder, D Flip-Flop (DFF), and FIFO.

## III METHODOLOGY

The proposed methodology focuses on developing an Enhanced Automatic UVM Testbench Generator (AUTG) using Python automation and System Verilog UVM methodology for efficient RTL verification. The framework automatically analyzes RTL design files and extracts important DUT information such as module name, input signals, output signals, clock signals, reset signals, and bus widths. This extracted information is used to generate reusable UVM verification components automatically, reducing manual coding effort and verification complexity.

The proposed system consists of a Python parser, interface generator, sequence item generator, driver generator, monitor generator, scoreboard generator, and testbench modules. The Python parser reads the RTL design and extracts DUT signal information. Based on the extracted information, the framework automatically generates verification files such as **interface.sv, seq_item.sv, driver.sv, monitor.sv, and scoreboard.sv**. These generated files form a complete UVM verification environment for the DUT.

The generated driver applies randomized input transactions to the DUT through the interface, while the monitor continuously captures DUT activity during simulation. The scoreboard compares actual DUT outputs with expected outputs generated from the Golden Reference Model and reports any functional mismatches. This automated verification flow improves debugging efficiency and ensures functional correctness of the design.

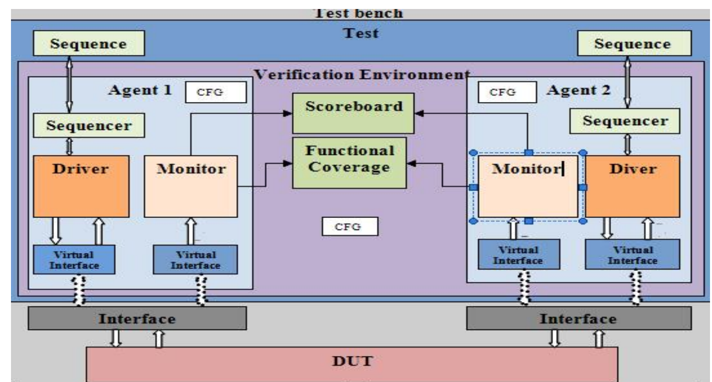
The proposed framework also integrates Assertion-Based Verification (ABV) and Functional Coverage techniques to improve verification reliability and completeness. Assertions automatically detect invalid DUT behavior and output mismatches during simulation, while functional coverage verifies different DUT scenarios, signal transitions, and input combinations. The framework is validated using Full Adder, D Flip-Flop (DFF), and FIFO designs, demonstrating successful verification of combinational, sequential, and memory-based circuits.

## IV PROPOSED ARCHITECTURE

The proposed system consists of an Enhanced Automatic UVM Testbench Generator (AUTG) developed using Python automation and System Verilog UVM methodology for efficient RTL verification. The proposed architecture includes a Python parser, interface generator, sequence item generator, driver, monitor, scoreboard, environment, and testbench modules for automatic generation of reusable UVM verification environments

The overall system is controlled by the Python parser, which reads RTL design files and extracts DUT information such as module name, input signals, output signals, clock signals, and reset signals. Based on the extracted information, the framework automatically generates UVM verification components required for functional verification. The generated verification environment reduces manual coding effort and improves verification productivity.

**Block diagram**




Proposed UVM Verification Architecture for Automatic RTL Verification

illustrates the proposed UVM verification architecture used in the Automatic UVM Testbench Generator (AUTG) framework for efficient RTL verification. The architecture contains major UVM components such as Test, Sequence, Sequencer, Driver, Monitor, Scoreboard, Functional Coverage, Interface, and DUT.

The test module controls the complete verification flow and initializes the verification environment. The sequence generates randomized transactions, which are transferred to the sequencer. The sequencer manages transaction flow and sends input stimulus data to the driver.The driver applies DUT input signals through the virtual interface and interface module. The DUT

processes the applied inputs and generates output responses during simulation. The monitor continuously captures DUT activity and sends collected transaction data to the scoreboard and functional coverage modules.

The scoreboard compares actual DUT outputs with expected outputs to detect functional mismatches during simulation. Functional coverage measures verification completeness by

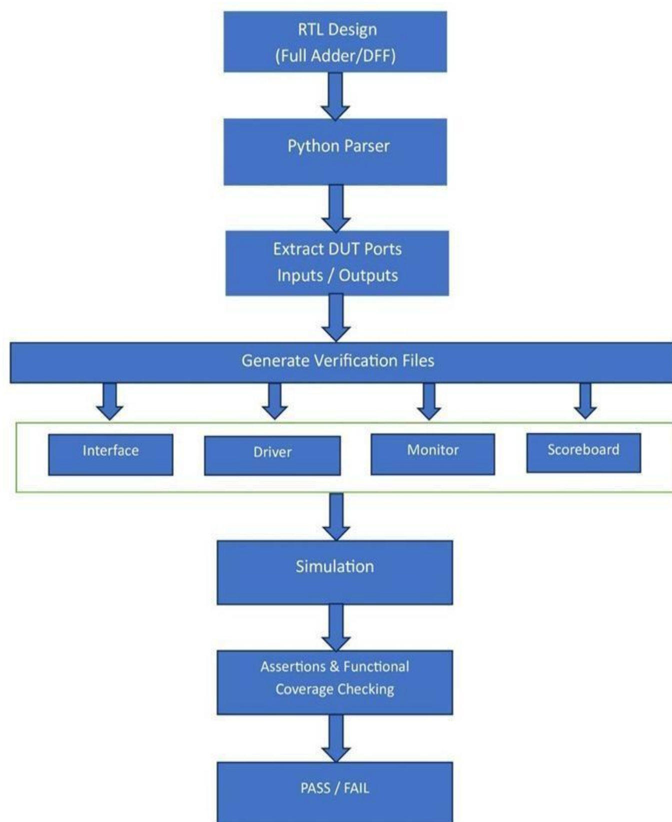
analyzing different DUT scenarios, signal transitions, and input combinations.

The interface module acts as a communication bridge between the DUT and UVM verification components. The proposed architecture supports reusable and scalable verification flow for combinational, sequential, and memory-based RTL designs. The virtual interface mechanism provides connectivity between System Verilog interfaces and class-based UVM components. This separation improves flexibility and allows the same verification components to be reused across multiple DUT configurations. The configuration (CFG) blocks are used to distribute verification settings and control parameters to different UVM components during simulation.

Functional coverage and scoreboard modules are placed at the environment level to collect verification information from multiple monitors. Functional coverage tracks DUT behavior and verifies whether all important verification scenarios have been exercised successfully. Coverage analysis helps identify untested conditions and improves verification completeness.

## RTL Parsing Technique

shows the working flow of the proposed Automatic UVM Testbench Generator (AUTG). The process starts with the RTL design file such as Full Adder, D Flip-Flop (DFF), or FIFO. The RTL file is given to the Python parser, which automatically reads the design and extracts important DUT




RTL Parsing Flowchart

information such as input signals, output signals, clock signals, and reset signals. Based on the extracted information, the framework automatically generates verification files including interface, driver, monitor, and scoreboard modules.

After generating the verification files, simulation is performed using the generated UVM environment. The driver applies input stimulus to the DUT, while the monitor captures DUT outputs during simulation. The scoreboard compares actual outputs with expected outputs to verify DUT functionality.

Assertion-based verification and functional coverage technique are then used to detect errors and measure verification completeness. Finally, the verification results are analyzed and the design is reported as PASS if all checks are successful or FAIL if mismatches are detected.

The proposed flow reduces manual coding effort, improves verification productivity, and supports automated verification of Full Adder, DFF, and FIFO designs.

The proposed flow automates the complete verification environment generation process and provides an efficient, reusable, and scalable solution for verifying combinational, sequential, and memory-based RTL designs such as Full Adder, D Flip-Flop (DFF), and FIFO.

## Assertion-Based Verification

Assertion-Based Verification (ABV) is implemented in the proposed Automatic UVM Testbench Generator (AUTG) to improve functional correctness checking and automatic bug detection during simulation. Assertions are special verification statements that continuously monitor DUT behavior and verify whether the generated outputs satisfy the expected design conditions. If any invalid condition, functional error, or output mismatch occurs, the assertion immediately reports an error message, making it easier to identify and debug design issues.

In the proposed framework, assertions are integrated with the generated UVM verification environment and operate automatically during simulation. Assertions help verify important DUT functionalities without requiring manual waveform analysis. For Full Adder verification, assertions check the correctness of sum and cout outputs. For D Flip-Flop (DFF) verification, assertions validate clock-triggered output updates and reset behavior. For FIFO verification, assertions verify correct read/write operations, synchronization behavior, and FIFO status conditions. Thus, assertion-based verification improves verification reliability, reduces debugging effort, and increases confidence in DUT functionality

## Functional Coverage

Functional Coverage is implemented in the proposed Automatic UVM Testbench Generator (AUTG) to measure verification completeness and ensure that all important DUT scenarios are tested successfully during simulation. It helps verify different input combinations, signal transitions, reset conditions, and operating modes of the DUT. Functional coverage provides information about

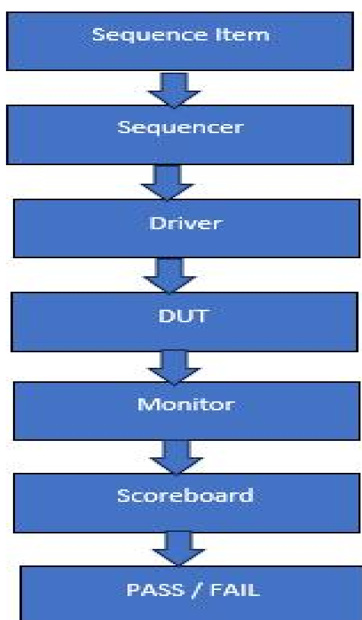
which design functionalities have been tested and helps identify any untested scenarios that require additional verification.

In the proposed framework, functional coverage is implemented using covergroups, coverpoints, and cross coverage techniques. The monitor continuously captures DUT activity during simulation and updates coverage information automatically. For Full Adder verification, coverage verifies all possible combinations of A, B, and Cin inputs. For D Flip-Flop (DFF) verification, coverage checks clock transitions, reset conditions, and output behavior, while FIFO verification validates read/write operations, data transfer conditions, and FIFO status signals. Thus, functional coverage improves verification reliability, increases confidence in DUT functionality, and ensures complete validation of the design.

**Automatic UVM Component**

Generation Automatic UVM Testbench Generator (AUTG). The verification process begins with the Sequence Item, which contains randomized transaction data and DUT input values. These transactions are forwarded to the Sequencer, which controls the flow of transactions and sends them to the driver The Driver receives the

transactions from the sequencer and applies the corresponding input stimulus to the Design Under Test (DUT) through the interface. The DUT processes the applied inputs and generates output responses based on its functionality.




UVM Verification Flow

The Monitor continuously observes the DUT inputs and outputs during simulation and collects transaction information. The captured data is then transferred to the Scoreboard for functional verification. The scoreboard compares the actual DUT outputs with the expected outputs generated from the Golden Reference Model and checks for any mismatches.

If the actual outputs match the expected outputs, the verification result is reported as PASS. Otherwise, the scoreboard reports an error and the verification result is marked as FAIL. This verification flow enables automatic checking of DUT functionality and improves verification efficiency, reliability, and debugging capability.

**IMPLEMENTATION AND RESULTS**

The generated verification environment demonstrated successful interaction between all UVM components throughout the verification process. The sequencer generated transaction data and transferred it to the driver, which applied stimulus to the DUT through the generated interface. The monitor continuously observed DUT inputs and outputs and collected transaction information during simulation. This information was forwarded to the scoreboard for functional correctness checking and result analysis.

The proposed AUTG framework automatically generated reusable UVM verification files, reducing the need for manual testbench development. The generated files were successfully

used for multiple DUTs without significant modifications, demonstrating the reusability and scalability of the framework. Waveform analysis confirmed correct signal transitions and expected DUT behavior for all test cases.

The functional coverage reports indicated that all important verification scenarios, signal transitions, and input combinations were exercised successfully during simulation. The assertion results confirmed that no functional mismatches or invalid DUT behaviors were detected. These results demonstrate that the proposed framework provides an efficient, reliable, and automated solution for modern RTL functional verification while significantly reducing verification effort and development time.

Overall, the proposed Automatic UVM Testbench Generator (AUTG) achieved successful generation of reusable verification components and demonstrated reliable verification performance across all tested designs. The integration of Python automation, assertion-based verification, and functional coverage techniques significantly enhanced verification quality while reducing development time and verification complexity.

**Verification Results of Full Adder, DFF, and FIFO**

| DUT | Type | Files Generated | Coverage | Result |
| --- | --- | --- | --- | --- |
| Full Adder | Combinational | 5 | 100% | PASS |
| DFF | Sequential | 5 | 100% | PASS |
| FIFO | Memory-Based | 5 | 100% | PASS |

presents the overall verification summary of the proposed Automatic UVM Testbench Generator (AUTG). The framework was evaluated using three different RTL designs: Full Adder, D Flip-Flop (DFF), and FIFO. These designs represent combinational, sequential, and memory-based circuits respectively, allowing the framework to be tested across different categories of digital systems.

The results indicate that the proposed framework successfully generated the required UVM verification files for all DUTs. A total of five verification files were generated automatically for each design, including interface, sequence item, driver, monitor, and scoreboard modules. The automatic generation of these files significantly reduced manual coding effort and simplified the development of the verification environment.

Functional coverage results achieved **100% coverage** for all tested designs, indicating that all important verification scenarios, signal combinations, and operating conditions were exercised successfully. This demonstrates that the generated verification environment was able to provide complete validation of DUT functionality.

The verification results for Full Adder, DFF, and FIFO were reported as **PASS**, indicating that no functional mismatches were detected during simulation. The successful verification of combinational, sequential, and memory-based circuits demonstrates the flexibility and scalability of the proposed AUTG framework.

Furthermore, the framework improved verification productivity by reducing testbench development time and minimizing the possibility of human errors. The generated verification components were reusable and could be applied to different RTL designs with minimal modifications. Overall, the results confirm that the proposed AUTG framework provides an efficient, scalable, and reliable solution for modern RTL functional verification applications.

The generated UVM environment successfully supported stimulus generation, DUT monitoring, output comparison, and functional checking during simulation. Assertion-based verification also confirmed correct DUT behavior and helped detect errors automatically.

**Main Python Code for Automatic UVM File Generation**

```
import os
from parser import parse_ports
from interface_gen import generate_interface
from seq_item_gen import generate_seq_item
from driver_gen import generate_driver
from monitor_gen import generate_monitor
from gref_parser import parse_gref
from scoreboard_gen import generate_scoreboard

# Select design
design_name = input("Enter design name: ")

# Paths
design_path = f"../design/{design_name}.sv"
gref_path = f"../gref/{design_name}_gref.sv"

# Create separate output folder
output_dir = f"../output/{design_name}"
os.makedirs(output_dir, exist_ok=True)

# Step 1: Parse DUT
inputs, outputs = parse_ports(design_path)

print("Inputs:", inputs)
print("Outputs:", outputs)

# Step 2: Generate Interface
interface_code = generate_interface(inputs, outputs)

with open(f"{output_dir}/interface.sv", "w") as f:
    f.write(interface_code)


# Step 3: Generate Sequence Item
seq_code = generate_seq_item(inputs, outputs)

with open(f"{output_dir}/seq_item.sv", "w") as f:
    f.write(seq_code)

print("Sequence item generated!")

# Step 4: Generate Driver
driver_code = generate_driver(inputs)

with open(f"{output_dir}/driver.sv", "w") as f:
    f.write(driver_code)
print("Driver generated!")

# Step 5: Generate Monitor
monitor_code = generate_monitor(inputs, outputs)

with open(f"{output_dir}/monitor.sv", "w") as f:
    f.write(monitor_code)

print("Monitor generated!")

# Step 6: Parse Golden Reference
gref_lines = parse_gref(gref_path)

# Step 7: Generate Scoreboard
scoreboard_code = generate_scoreboard(gref_lines)
with open(f"{output_dir}/scoreboard.sv", "w") as f:
    f.write(scoreboard_code)

print("Scoreboard generated!")
```

shows the main Python script used in the proposed Automatic Functional coverage was implemented using covergroups and cross coverage techniques to verify all possible combinations of A, B, and Cin input signals. Coverage analysis confirmed successful verification of all important DUT scenarios and input conditions.

UVM Testbench Generator (AUTG). The script acts as the central controller of the framework and coordinates the automatic generation of UVM verification files. It imports the required Python modules for RTL parsing, interface generation, sequence item generation, driver generation, monitor generation, Golden Reference parsing, and scoreboard generation.

The user enters the design name, and the script automatically identifies the corresponding RTL design file and Golden Reference Model. The parser analyzes the RTL code and extracts important DUT information such as input signals, output signals, clock signals, and reset signals. This information is then used to generate the required UVM verification components automatically.

The script creates a separate output directory and generates verification files including **interface.sv, seq_item.sv, driver.sv, monitor.sv, and scoreboard.sv**. After successful generation of each file, a status message is displayed to inform the user about the progress of the generation process.

The generated verification files are reused for simulation, assertion-based verification, and functional coverage analysis. The implementation of the main Python script significantly reduces manual coding effort, improves verification productivity, and provides a scalable solution for automated RTL verification of Full Adder, D Flip-Flop (DFF), and FIFO designs.

**Full Adder Verification Result**

The generated UVM verification environment was first tested using a Full Adder DUT to validate combinational circuit verification functionality. The Full Adder performs binary addition using three input signals namely A, B, and Cin, and generates two output signals Sum and Cout. The proposed framework automatically generated interface, driver, monitor, scoreboard, sequencer, and testbench components required for verification.

The driver module applied different randomized input combinations of A, B, and Cin to the DUT through the generated interface. The DUT processed the input signals and generated corresponding Sum and Cout outputs based on Full Adder functionality. The monitor continuously captured DUT input and output signals during simulation and transferred the collected data to the scoreboard for comparison.

The scoreboard compared actual DUT outputs with expected outputs generated from the Golden Reference Model. The comparison results confirmed correct arithmetic functionality of the Full Adder without any mismatches during verification.

Assertions were also implemented to validate output correctness automatically during simulation.

Example assertion used:

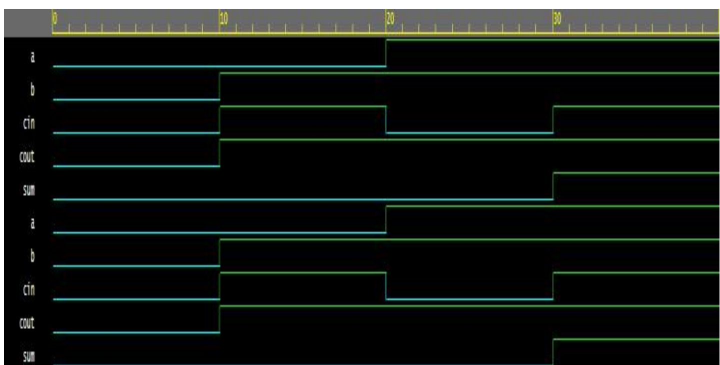
```
assert (Sum == (A ^ B ^ Cin));
```

Functional coverage was implemented using covergroups and cross coverage techniques to verify all possible combinations of A, B, and Cin input signals. Coverage analysis confirmed successful verification of all important DUT scenarios and input conditions.

```
import os

from parser import parse_ports
from interface_gen import generate_interface
from seq_item_gen import generate_seq_item
from driver_gen import generate_driver
from monitor_gen import generate_monitor
from gref_parser import parse_gref
from scoreboard_gen import generate_scoreboard


# Select design
design_name = input("Enter design name: ")

# Paths
design_path = f"../design/{design_name}.sv"
gref_path = f"../gref/{design_name}_gref.sv"

# Create separate output folder
```

```
Monitor generated!
Scoreboard generated!
PS D:\AUTG_Project\tool> python main.py
Enter design name: full_adder
Inputs: ['a', 'b', 'cin']
Outputs: ['sum', 'cout']
Interface generated successfully!
Sequence item generated!
Driver generated!
Monitor generated!
```

Automatic UVM File Generation Using Python Parser



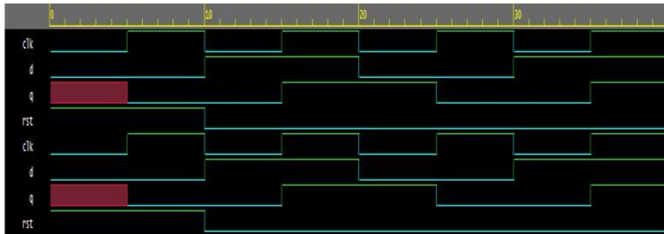

Full Adder Waveform

**D Flip-Flop (DFF) Verification Result**

```
import os

from parser import parse_ports
from interface_gen import generate_interface
from seq_item_gen import generate_seq_item
from driver_gen import generate_driver
from monitor_gen import generate_monitor
from gref_parser import parse_gref
from scoreboard_gen import generate_scoreboard


# Select design
design_name = input("Enter design name: ")

# Paths
design_path = f"../design/{design_name}.sv"
gref_path = f"../gref/{design_name}_gref.sv"

# Create separate output folder
```

```
PS D:\AUTG_Project\tool> python main.py
Enter design name: dff
Inputs: ['clk', 'rst', 'd']
Outputs: ['reg']
Interface generated successfully!
Sequence item generated!
Driver generated!
Scoreboard generated!
```

Automatic UVM File Generation for DFF




D Flip-Flop (DFF) Waveform

shows the DFF verification results obtained using the proposed Automatic UVM Testbench Generator (AUTG). The main Python script successfully parsed the RTL design and extracted the DUT signals including **clk**, **rst**, and **d**. Based on the extracted information, the framework automatically generated the required UVM verification files such as **interface.sv, seq_item.sv, driver.sv, monitor.sv, and scoreboard.sv**.

The generated UVM environment was used to perform functional verification of the DFF design. During simulation, the driver applied input stimulus to the DUT, while the monitor continuously observed the DUT signals and collected transaction information. The scoreboard compared the actual DUT outputs with the expected outputs to verify functional correctness.

The waveform analysis confirms that the output signal **q** follows the input signal **d** on the positive edge of the clock signal. When the reset signal is asserted, the output is initialized correctly according to the DFF functionality. The observed waveform behavior matches the expected operation of a D Flip-Flop, indicating correct DUT functionality.

The assertion results confirmed that no functional mismatches or invalid conditions were detected during simulation. Functional coverage results verified clock transitions, reset operations, and output updates successfully. The successful verification results demonstrate the effectiveness of the proposed AUTG framework in automatically generating reusable UVM verification environments for sequential circuit verification, resulting in an overall **PASS** status.

## FIFO Verification Result

Automatic Generation of UVM Verification Files for FIFO

The proposed UVM verification environment was tested using a FIFO DUT to validate memory-based and sequential circuit verification capability. FIFO supports synchronized read and write operations using clock and control signals. The generated verification environment successfully verified FIFO data transfer, buffer control, and synchronization behavior.

The Python parser automatically extracted DUT signals such as clk, rst, wr_en, and rd_en from the RTL design file. Based on the extracted DUT information, the framework generated interface, sequence item, driver, monitor, and scoreboard modules automatically.

During simulation, the driver applied randomized write and read operations to the DUT through the generated interface. The monitor captured FIFO input and output behavior during simulation and transferred the collected information to the scoreboard for verification analysis.

The scoreboard compared actual FIFO outputs with expected outputs generated from the Golden Reference Model. Functional coverage and assertion-based verification techniques were used to verify read/write operations, synchronization behavior, and FIFO status conditions. The results confirmed successful FIFO verification using the proposed AUTG framework.

## CONCLUSION AND FUTURE SCOPE

This paper presented an **Enhanced Automatic UVM Testbench Generator (AUTG)** developed using Python automation and SystemVerilog UVM methodology for efficient RTL verification. The proposed framework automatically parses RTL design files, extracts DUT information, and generates reusable UVM verification components such as interface, sequence item, driver, monitor, and scoreboard modules. This automation significantly reduces manual coding effort and simplifies the development of verification environments.

The proposed framework was successfully validated using **Full Adder, D Flip-Flop (DFF), and FIFO** designs representing combinational, sequential, and memory-based circuits. Simulation results, waveform analysis, assertion-based verification, and functional coverage confirmed correct DUT functionality and successful verification of all tested designs. The framework achieved **100% functional coverage** and generated PASS results for all DUTs.

The automatic generation of verification files improved verification productivity, reduced development time, and minimized the possibility of human errors. The generated UVM environment demonstrated reusability and scalability across different RTL designs without requiring significant modifications. Furthermore, the integration of assertions and functional coverage enhanced verification reliability and completeness.

Overall, the proposed AUTG framework provides an efficient, reliable, and automated solution for modern VLSI functional verification. The successful verification results demonstrate its effectiveness in reducing verification complexity while improving verification quality, making it a practical approach for future RTL verification applications.

## REFERENCES

[1] G. Hunter, *Advanced UVM: Universal Verification Methodology*. New York, NY, USA: Springer, 2013.

[2] M. Glasser and R. Mathews, *A Practical Guide to Adopting the Universal Verification Methodology (UVM)*. Boston, MA, USA: Pearson Education, 2012.

[3] C. Spear and G. Tumbush, *System Verilog for Verification: A Guide to Learning the Testbench Language and Functional Coverage*, 3rd ed. New York, NY, USA: Springer, 2012.

[4] J. Bergeron, *Writing Testbenches Using SystemVerilog*, 2nd ed. New York, NY, USA: Springer, 2006.

[5] Accellera Systems Initiative, *Universal Verification Methodology (UVM) 1.2 User Guide*. Accellera Standards Organization, 2015.

[6] S. Sutherland, S. Davidmann, and P. Flake, *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling*. New York, NY, USA: Springer, 2013.

[7] V. Iyengar and M. Vachharajani, *Verification Methodology Manual for SystemVerilog*. New York, NY, USA: Springer, 2006.

[8] S. Ray and J. Bhadra, "Automated Verification Environment Generation Using Python for RTL Designs," *International Journal of Computer Applications*, vol. 180, no. 42, pp. 12–18, 2018.

[9] A. Habibi and S. Tahar, "Design Verification Using Assertion-Based Verification Techniques," *IEEE Design & Test of Computers*, vol. 21, no. 1, pp. 56–65, Jan.–Feb. 2004.

[10] H. Foster, A. Krolnik, and D. Lacey, Assertion-Based Design, New York, NY, USA: Springer, 2004.