

OPEN-SOURCE PYTHON BASED HARDWARE VERIFICATION TOOL

by

Debasmita Aich, M.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Engineering
August 2021

Committee Members:

Aslan Semih, Chair

Damian Valles Molina

William A Stapleton

COPYRIGHT

by

Debasmita Aich

2021

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

This work is protected by the Copyright Laws of the United States (Public Law 94-553, section 107). Consistent with fair use as defined in the Copyright Laws, brief quotations from this material are allowed with proper acknowledgment. Use of this material for financial gain without the author's express written permission is not allowed.

Duplication Permission

As the copyright holder of this work, I, Debasmita Aich, authorize duplication of this work, in whole or in part, for educational or scholarly purposes only.

ACKNOWLEDGEMENTS

I would like to thank my guide, Dr. Semih Aslan, whose expertise was invaluable in formulating the research questions and methodology. Your insightful feedback pushed me to sharpen my thinking and brought my work to a higher level.

I would also like to thank my committee members Dr. Damian Valles and Dr. William A Stapleton for their time and cooperation.

This thesis is dedicated to my parents, family, and Texas State University who have supported me throughout.

TABLE OF CONTENTS

| | Page |
|---|-------------|
| ACKNOWLEDGEMENTS | iv |
| LIST OF FIGURES | vii |
| LIST OF ABBREVIATIONS | ix |
| ABSTRACT | x |
| CHAPTER | |
| 1. INTRODUCTION | 1 |
| 2. LITERATURE SURVEY | 5 |
| 2.1 Previous Research | 5 |
| 2.2 Work Focus | 6 |
| 3. PROBLEM DESCRIPTION | 11 |
| 3.1 Verification bottleneck | 15 |
| 3.2 Industry-level verification methods | 17 |
| 4. PROPOSED FLOW | 21 |
| 4.1 Backend overview | 21 |
| 4.2 Files and their roles | 24 |
| 5. VERIFOG OVERVIEW | 26 |
| 5.1 More about verifog tool | 27 |
| 5.2 Verifog files used for top-level module | 32 |
| 5.3 Configuration | 33 |
| 6. TESTING THE TOOL | 35 |
| 6.1 Testing example_adder.v | 35 |
| 6.2 Testing leading_one.v: Partial Verification | 40 |
| 6.3 Testing leading_one.v: Full Verification | 47 |
| 6.4 Tool limitation | 53 |

| | |
|-------------------------------------|-----|
| 7. CONCLUSION AND FUTURE WORK | 55 |
| APPENDIX SECTION | 57 |
| REFERENCES | 124 |

LIST OF FIGURES

| Figure | Page |
|--|------|
| 1-1: Design Process Flowchart | 2 |
| 2-1: cocotb Flowchart | 9 |
| 3-1: Industrial Verification Flow | 13 |
| 3-2: Proposed Verification Flow | 14 |
| 3-3: Design and Verification gaps | 16 |
| 4-1: Verifog System Flow Chart..... | 22 |
| 5-1: Verifog Tool designed by Qt Designer | 26 |
| 6-1: Adder Golden File | 35 |
| 6-2: Adder TVF..... | 36 |
| 6-3: Adder Verify Step..... | 38 |
| 6-4: Adder Makefile..... | 39 |
| 6-5: Adder Report | 39 |
| 6-6: Adder Regression | 40 |
| 6-7: Constrained Inputs for Leading One Detector (Partial) | 41 |
| 6-8: Golden File for Leading One Detector (Partial)..... | 42 |
| 6-9: Generating TVF for Leading One Detector (Partial) in Verifog..... | 43 |
| 6-10: TVF Inputs for Leading One Detector (Partial) | 44 |
| 6-11: Verify Step for Leading One Detector (Partial) in Verifog..... | 45 |
| 6-12: Regression for Leading One Detector (Partial) | 46 |

| | |
|---|----|
| 6-13: Makefile for Leading One Detector (Partial) | 46 |
| 6-14: Report for Leading One Detector (Partial) | 47 |
| 6-15: Constrained Inputs for Leading One Detector (Full) | 48 |
| 6-16: TVF Generation for Leading One Detector (Full) in Verifog | 49 |
| 6-17: TVF for Leading One Detector (Full). | 50 |
| 6-18: Verify Step for Leading One Detector (Full) in Verifog | 51 |
| 6-19: Regression for Leading One Detector (Full). | 52 |
| 6-20: Makefile for Leading One Detector (Full) | 52 |
| 6-21: Report for Leading One Detector (Full). | 53 |

LIST OF ABBREVIATIONS

| Abbreviation | Description |
|---------------------|--|
| FPGA | Field Programmable Gate Array |
| ASIC | Application Specific Integrated Circuit |
| SoC | System on Chip |
| IP | Intellectual Property |
| NoC | Network on Chip |
| DUT | Design Under Test |
| IC | Integrated Circuit |
| OVM | Open verification methodology |
| UVM | Universal verification methodology |
| RTL | Register Transfer Level |
| GUI | Graphical User Interface |
| VPI | Virtual Path Identifier |
| OFDM | Orthogonal Frequency-Division Multiplexing |

ABSTRACT

Today a need for more efficient and time-effective hardware/chip verification has become a necessity due to the increasing size and complexity of several electronic devices. Everyone wants to develop new technology, and for this urge, every day, something new is planned and designed in the Silicon Industry. Due to the rapid growth of technology and competition in between the industry, more and more complex and sophisticated electronic devices are being developed for various areas like medical, entertainment, defense industry, Space centers, etc.

The creation of these electronic devices needs complex designs and a high level of verification on time. Many verification methodologies use timing simulations, but unfortunately, it is the most time-consuming for several designs. The main purpose of chip verification is to catch bugs in the designs in the most convenient way, and the earliest the bug is found, the project can be ready for TAPE-OUT. The purpose of this research is to design a Verification tool using Python and cocotb to reduce the need of building multiple testbenches and to reduce some of the hurdles like not able to catch bugs at the earliest stages of design phase which comes in the way of chip Verification. This tool called Varifog catches the bugs at the earliest stages of the design phase without using any testbenches and hence can save a lot of time for Verification Engineers who write multiple basic tests for the designs just to check if the design is generating expected outputs or if any chain/fublet is broken. Verifog is tested with simple as well as complex Verilog design files and is efficiently catching RTL bugs at the earliest design phases.

1. INTRODUCTION

A design process as shown in Figure 1-1 [1] transforms a set of specifications into an implementation of the specifications. At the specification level, the specifications state the functionality that the design executes but does not indicate how it executes. An implementation of the specifications spells out the details of how the functionality is provided. Both a specification and implementation are a form of description of functionality, but they have different levels of concreteness or abstraction. A design process refines a set of specifications and produces various levels of concrete implementations. Design verification is the reverse process of design. Design verification starts with implementation and confirms that the implementation meets its specifications. Thus, at every step of the design, there is a corresponding verification step. For example, a design step that turns a functional specification into an algorithmic implementation requires a verification step to ensure that the algorithm performs the functionality in the specification. Similarly, a physical design that produces a layout from a gate netlist must be verified to ensure that the layout corresponds to the gate netlist.

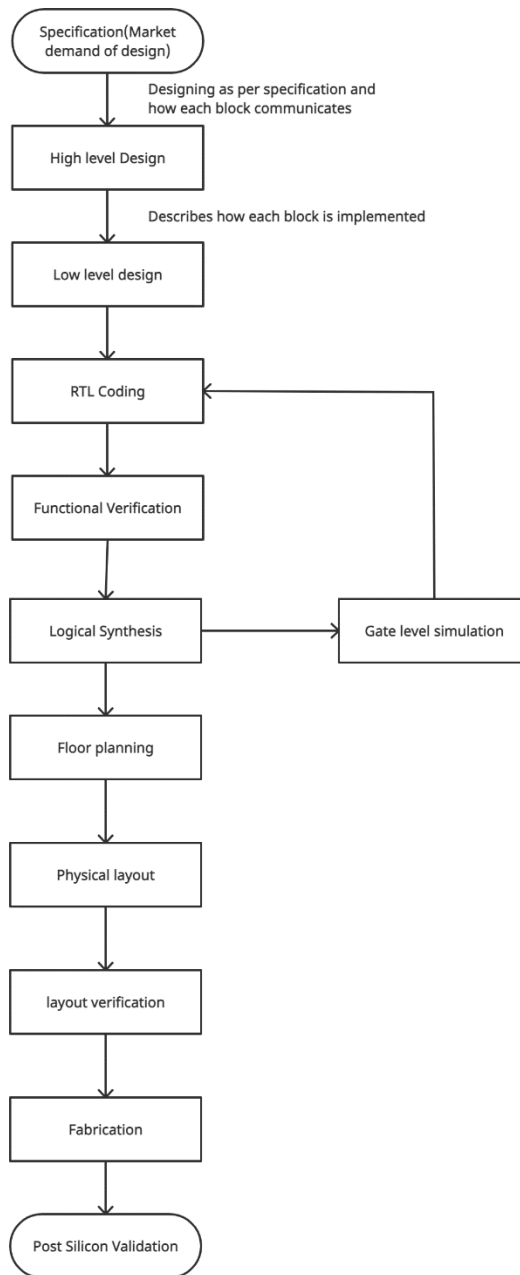


Figure 1-1 Design Process Flowchart

Verification ensures that the design meets the specification and has become an indispensable part of a product development cycle of a digital hardware design. With the increasing design complexity of digital hardware, verification has become increasingly on the critical path of the cycle[2]. While verification is just one word, it is a must step to

be followed in a product's TAPE-OUT process. Whether it is an FPGA, or ASIC, or Pre/Post Silicon, Mixed Signal, or SoC, every single platform needs to be verified. Taking about SoC's, with increasing numbers of CPU cores, multimedia subsystems, and communication IP's with complex BUS connectivity's in today's SoCs, the main SoC interconnects, crossbars, or NoC fabrics become key components of the system. In Addition, IP reuse and NoC generation solutions have enabled the conception of new SoC architectures within a few months. The verification of the SoC bus interconnects faces the challenge of verifying the correct routine of transactions as well as security and protection modes, power management features, virtual address space, and bus protocol translations while still reaching project milestones [3] [4].

Foundation of Verification:

- Checking the correctness of the transactions (scoreboard)
- Checking that various protocols are being obeyed.
- Adding additional verification constraints on to aspects of the interconnect (corner cases)
- Ways to measure the quality of the verification performed (coverage)

To ensure that all these steps are achieved, DV engineers write effective testbenches, and the goal is to catch the bugs while the steps are being implemented. The existence of a tool at this point that can catch the design bugs at the early stages will make the verification process a relief for the engineers. Only a design file and golden file need to be uploaded in the tool, and the input of the design must be constrained by the user. The tool then automatically following few easy steps will give a report on the mismatched values hence finding any design bugs which is causing the mismatches in the

values.

In this process of verifying several complex designs, the DV engineer's role is to find as many bugs as possible in less time. The purpose of this paper is to develop a tool that will help the DV engineers achieve their goal of catching maximum bugs effectively and in no time. The engineers can select between partial and full verification in the tool and can upload their design file and golden value file in the tool to check the correctness of the DUT and to find as many bugs as possible. The work of a golden file is described further below in this paper. Details of the tool and how it works are also mentioned in chapters 4 and 5.

2. LITERATURE SURVEY

2.1 Previous Research

The design verification industry has leaped forward from its simplistic roots when a directed stimulus was used to perform verification with HDL. The approach has now shifted to using advanced programming techniques to create richer and reusable models; directed random stimulus to reduce test case writing efforts, and functional and code coverage to measure verification progress objectively. While these techniques and methodologies have improved design quality and verification efficiency, there is still much development to accomplish, especially for complex chip designs [5].

In these complex designs, one of the important things to accomplish is to understand the importance of Formal Verification because it ensures the correctness of the design[6].

The system Verilog Assertions were developed to catch the bugs more easily and efficiently in the early stages of the design flow, and later, those were used to reduce other verification bottlenecks[7]. Several researches have proposed a unified flow for functional verification using assertion-based and coverage-based verification techniques [8]. Verification of System-On-Chip (SoC) is again a field where verification complexity is increasing every day, so nowadays a hardware extension is developed that allows to efficiently synthesize the checkers[9]. Previous researches have been done on proposing a more powerful and efficient approach for writing fully self-checking tests using a transaction-based verification method[10] [11]. These approaches from previous research talk about assertions, building powerful testbenches, coverage, which is currently being used in the hardware industry. None of these methods guarantee to catch bugs at the early stages. Also, these methods use System Verilog/Verilog/VHDL testbench creation. This

paper talks about catching bugs faster, without the use of traditional System Verilog testbenches in UVM environment and implementing python-based testbenches using cocotb environment which gives more flexibility in the entire verification process and its faster than the traditional methods.

2.2 Work Focus

This paper focuses on the development of a tool, “Verifog” in a GUI-based system that is able to catch the bugs in the early stages of the design phase.

The GUI of “Verifog” is designed using PyQt [12]. PyQt has a tool called Qt Designer[13] [14] which is required to compile it into a usable GUI [15] [16]. “Verifog” takes into consideration the RTL file (design file in Verilog) and a golden file (a file developed by the Design Engineers containing all the expected outputs of the design file) and creates a “TVF” (test vector file). A TVF is a file containing all the constraints, test vectors, and every expected result from the golden file. The testing environment is build using “cocotb”[17] [18]. The python/cocotb testbench parses the TVF and compares it with the golden file. The details on how the entire flow of Verifog works is mentioned in chapter 4. Few of the reasons for selecting Python as the scripting language for this research are[19] [20] [21] [22]:

- Readable and maintainable code.
- Compatible with major platforms and systems.
- Robust Standard Library.
- Many open-source frameworks and tools.

GUI Design

The GUI is designed with Qt Designer, and to compile it into a usable GUI that used PyQt5. Figure 6-2 shows how the Verifog UI looks like.

Need for Qt Designer

Using Qt Designer can dramatically improve the productivity of the developer else, the developer ends up hand-coding the GUI in plain Python code. GUI applications often consist of the main window and several dialogs.

To create any graphical components in an efficient and user-friendly way, Qt Designer is the tool. Qt Designer provides a “what-you-see-what-you-get” user interface to create GUIs for PyQt5 applications productively and efficiently. With this tool creating GUIs can be done by simply dragging and dropping QWidget objects on an empty form. After that they can be arranged into a coherent GUI using different layout managers. Qt Designer also allows previewing the GUIs using different styles and resolutions. Connect signals and slots, create menus and toolbars, and more. It is a platform and programming language independent. It does not produce code in any programming language, but it creates .ui files. These files are XML files with detailed descriptions of how to generate Qt-based GUIs. Qt is a tool that makes the GUI code more automated. For Qt Designer to compile into a usable GUI PyQt5 is used. Here the GUI will be compiled to python code[12].

PyQt5

PyQt is a library that lets the user use the Qt GUI framework. Qt itself is written in C++. By using PyQt, one can build applications much more quickly while not sacrificing much of the speed of C++.

PyQt5 refers to the most recent version 5 of Qt. It's a comprehensive set of Python bindings for Qt v5. It is implemented as more than 35 extension modules and enables Python to be used as an alternative application development language to C++ on all supported platforms[23] [24]. It gives good separation between code and GUI, so a core logic and GUI can be designed separately and then combine them.

Generic Testbench using cocotb

- cocotb is a coroutine-based co-simulation testbench environment for verifying VHDL and System Verilog RTL using Python.
- An HDL design can be simulated using cocotb, but cocotb will require a simulator to do this.
- Like UVM, cocotb also encourages design re-use and randomized testing but is implemented in python.
- SystemVerilog or VHDL are used for the design itself, not the testbench with cocotb.
- cocotb lowers the overhead of creating a test.
- cocotb can discover the tests by itself, and hence no additional step is required for adding tests to a regression.
- All the verification can be done with python, which has various advantages over using System Verilog or VHDL for verification:
 - Writing Python is fast.
 - It is easy to interface with other languages from Python.
 - Python has a huge library of existing code to reuse.

- In Python, tests can be edited and rerun without having to recompile the design or exit the simulator GUI. In other words, Python is interpreted.

Implementation of cocotb

No additional RTL code is required for a cocotb testbench to work. The DUT is instantiated as the top level in the simulator without any wrapper code. Stimulus is driven into the inputs through cocotb to the DUT and monitors the outputs directly from Python. cocotb cannot instantiate HDL blocks so the DUT in use must be complete[25]. Figure 2-1 below is a pictorial representation of how it works.

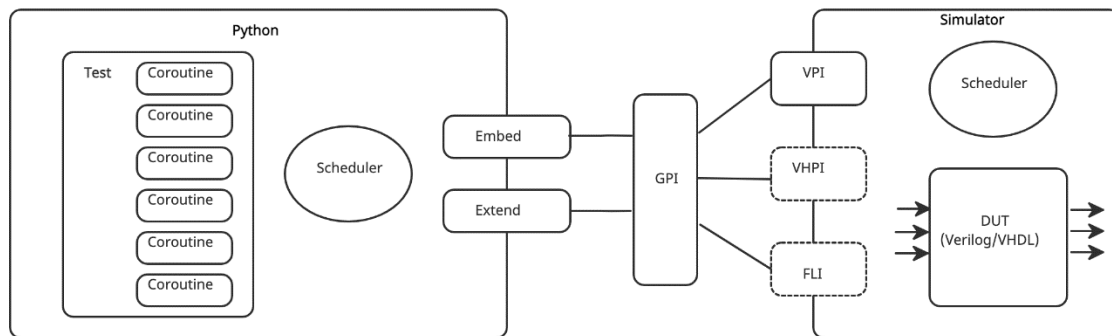


Figure 2-1: cocotb Flowchart

When cocotb initializes, it finds the toplevel instantiation in the simulator and creates a handle called `dut`. Top-level signals can be accessed using the “dot” notation used for accessing the object attribute in python. For example:

```
clk = dut.clk
```

Values can be assigned to the signals using the *value* property. For example:

```
Clk = dut.clk
```

```
Clk.value = 1
```

Testbenches built using cocotb are used coroutines. While the coroutine is executing, the simulation stops. The coroutine uses the *await* keyword to block another coroutine's execution. cocotb contains a library called GPI written in C++ that is an abstraction layer of VPI, VHPI, and FLI simulator interfaces. The interaction between Python and GPI is via a Python extension module called *simulator*. Embed is to embed a shell within an existing test where it can inspect local variables. Extend means extending existing build flows.

3. PROBLEM DESCRIPTION

In general, design verification encompasses many areas, such as functional verification, timing verification, layout verification, and electrical verification, just to name a few[26]. This research will be focusing on hardware verification issues. The first thing any DV engineer sees after the design has been made is the DUT which they have to verify. There are several steps to be followed while the verification is taking place, like creating a Test Plan, deciding which features of the DUT need to be verified, how to verify, and then creating the verification environment using UVM or OVM to create testbenches that checks the functionality of the DUT features. Design Engineers like to see testbenches that have code reusability, efficiency, verifies corner cases, catch as many bugs as possible. Also, the testbench should be ready to be modified at any point of time during the project life cycle if the design changes. The testbenches should be written in a way so that they can be reused in the future or in the current project for the same feature, which might be in a different hierarchy (for example, the same Duty Cycle Detector can be present in core0/1/2/3 and Bus).

To simulate the design, both the DUT and the stimulus provided by the test bench are needed. A test bench is a Verilog code that allows providing a documented, repeatable set of stimuli that is portable across different simulators. A test bench can be as simple as a file with clock and input data or a more complicated file that includes error checking, file input and output, and conditional testing[27] [28]. There are different types of testbenches in use: Simple testbench, self-checking testbench, self-checking testbench with test vectors[29].

Few Verification challenges are:

- Simulation Complexity- Increasing IP count of SoC's means slower simulation, and an increasing number of external interfaces means increasing testbench complexity.
- Functional coverage.
- Catching as many bugs as possible.

The current silicon industry uses complex testbenches written by the DV engineers to ensure that the maximum number of bugs is caught. They write several tests for individual scenarios of the RTL and run regressions to check the corner cases.

Industry-based verification flow is illustrated in Figure 3-1.

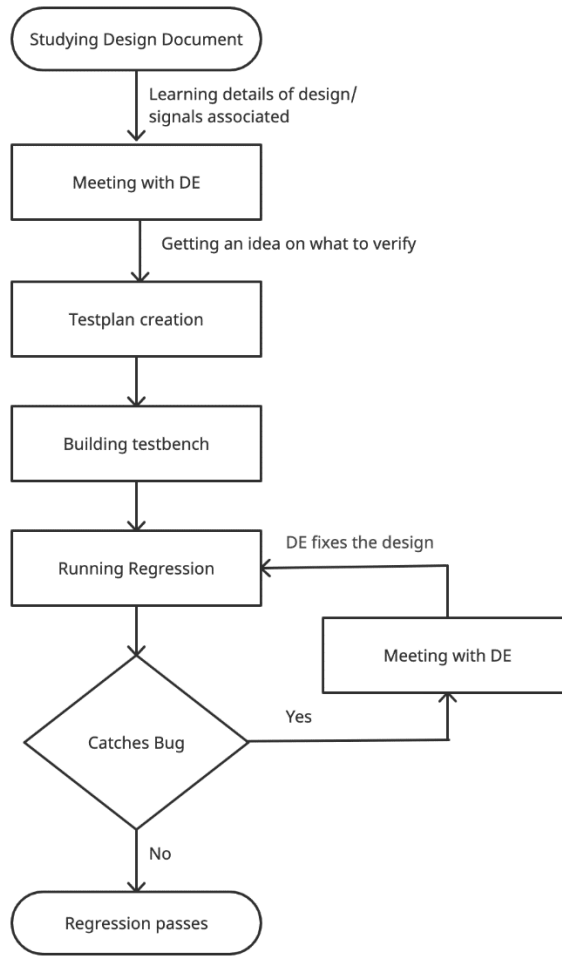


Figure 3-1: Industrial Verification Flow

However, this process does not guarantee the catching of bugs all the time. Also, several testbenches need to be built for checking every case. During times when design changes (which happens almost every time), building testbenches in no time becomes a tough job, and most of the time, reusing previous testbenches might not work as new IPs or cores have been introduced in the new design. This thesis works on this issue of verification challenge (catching bugs at early stages without developing testbenches) by building a tool, “Verifog” which not only catches bugs at an early design phase but also lessens the hurdles of creating System Verilog testbenches. Also, Verifog provides a

proof-of-concept that Python is strong enough to replace legacy used verification languages and environment. This is due to its flexibility, portability, and its vast community, which is wider than design/verification engineering ones. In the proposed system, the verification flow is shown in Figure3-2 below.

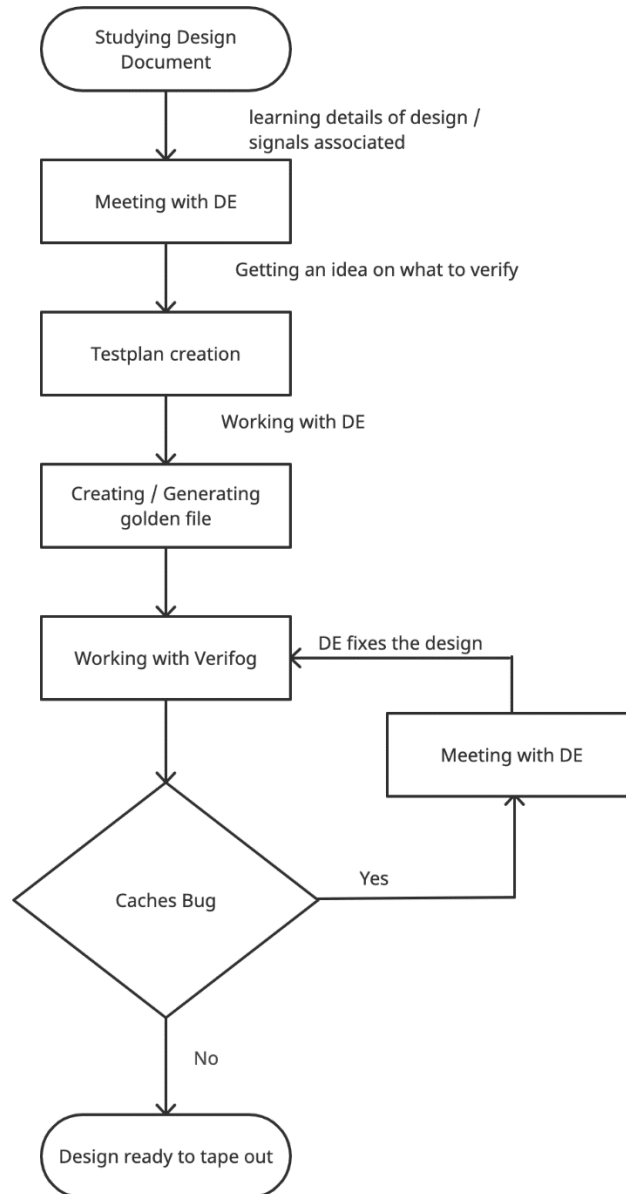


Figure 3-2: Proposed Verification Flow

The proposed system shows that after the test plan creation, the engineer is creating a golden value file. In the Verifog tool, this golden file, along with the RTL code (design file) can be uploaded to catch bugs in the RTL at the early stages. How this entire tool works with these two files to catch the bugs is described in the below sections.

3.1 Verification bottleneck

Chip Verification is one of the most significant parts of building and making the finest chips. Every company in the semiconductor industry around the globe goes through a ‘verification phase’ during a project’s life cycle. Design productivity growth continues to remain lower than complexity growth, but this time around, it is verification time, not design time, that poses the challenge. A recent statistic showed that 60-70% of the entire product cycle for a complex logic chip is dedicated to verification tasks. Verification of complex functions that we can build using new design tools poses a challenge to reducing the total product time [30]. According to Moore’s Law number of transistors in a dense IC double about every two years. Considering this law, a graph is shown in Figure 3-3. It shows the design and verifications gaps based on transistors per month. A design gap is always there between what Moore’s law states and design productivity. In the same way verification gap always exists between Moor’s law and verification productivity. This means if Moore’s law holds good still today, the existence of the verification gap is staying on the verification productivity. Many parts of the system are not getting verified at all.

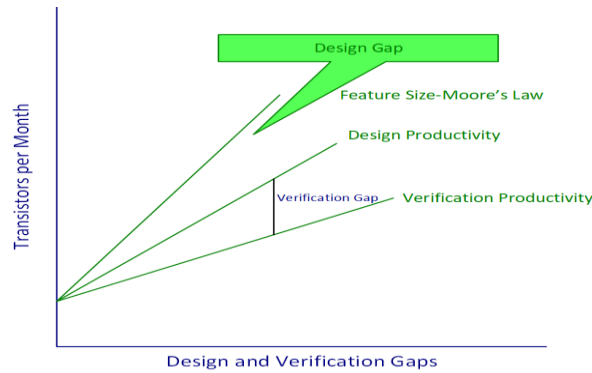


Figure 3-3: Design and Verification gaps

The verification bottleneck is an effect of raising the design abstraction level for the following reasons:

- Designing at a higher abstraction level allows us to build complex functions with ease. This increase in design complexity then results in almost doubling the verification effort. The functional complexity has been doubled and hence its verification scope.
- Using a higher level of abstraction for design, transformation, and eventual mapping to the product is not without information loss and misinterpretation. For instance, synthesis takes an HDL-level design and transforms it to the gate level. Verification is needed at this level to ensure that the transformation was indeed correct and that design intent was not lost. Raising the level of abstraction also brings about the question of interpretation of the code that is used to describe the design during simulation[31].

Options companies have for tackling verification bottlenecks:

- *Reducing chip complexity.* Practically, this is not possible because of customer demand for more functionality.

- *Reduce the number of designs.* This solution affects a company's long-term goal of being profitable.
- *Increase verification productivity.* One of the first things to do is to catch maximum bugs in the early stages of the design phase. This has obvious potential for gains in productivity. The rest of this paper concentrates on this last point[32].

3.2 Industry-level verification methods

Need for hardware verification

Hardware verification has been one of the biggest drivers for all kinds of verification and has seen the greatest practical impact of its results.

The use of formal techniques has not been uniformly successful, with equivalence checking widely used, assertion-based verification seeing increased adoption, and general property checking algorithm proving seeing only limited use. A major challenge in verification techniques providing scalable solutions is the inherent unbending of the problem. This only gets worse with increasing complexity and is reflected in the increasing number of bug escapes into silicon [33].

Functional Verification

In the whole process of Verification, Functional Verification plays a very important role since it tests for the functional capabilities of the RTL. The most widespread method of functional verification is dynamic in nature. The reason it is also called "dynamic" is because input patterns/stimulus are generated and applied over several clock cycles to the design, and the corresponding result is collected and compared against a reference/golden model for conformance with the specification[34]. A simulator

is used to compute all the values of the signals in the DUT and compare the same with the specified expected values and the calculated ones. The two types of simulators found these days are:

- Cycle-based simulator- does not care about what happens within a clock and evaluates signals once per cycle. This simulator is typically faster because of the low execution time.
- Event-based simulator- here events are taken into consideration (by the event it means within a clock or at the clock boundary), and I propagated through the design until a steady-state is achieved.

Random/Directed Functional Verification

When the logic gets more complex, the verification space increases. This brings about random dynamic simulation, which provides random stimulus to the design to maximize the functional space that can be covered[35]. The problem with random testing is that for very large and complex designs, it can be an unbounded problem. to solve this problem, the EDA industry introduced higher-level verification languages like the system C verification library. These introduced concepts such as constrained random stimulus, random stimulus distribution, and reactive testbenches. While implementing this verification methodology, the industry typically once include the following points:

- number of lines of code that were verified (Line coverage)
- how many of the logical expressions were tested (expression coverage)
- how many states in an FSM design were reached (FSM coverage)?
- The number of ports and registers that were toggled both ways during a simulation run (toggle coverage)

- The number of logical paths in the design code that were covered (path coverage).

Coverage in Verification

Every possible design verification methodology requires a coverage matrix to gauge the process, assess the effectiveness and help determine when the design is robust enough for tape out. At every step of the way and with every bug-finding technology and tool, verification ingeniously assesses coverage results and makes critical decisions on the next steps. For the verification of large complex system-on-chip designs, coverage metrics and the responses to them guide the entire flow. The term “coverage driven verification” describes the methodology built around coverage metrics as the primary way to manage verification. The different kinds of coverage are as follows:

- Code coverage- being one of the most traditional form, RTL code coverage has migrated from specialized add-on tools directly into the more advanced simulators providing much better performance and ease of use. Coverage is very helpful at identifying holes in verification; that is if a section of code has not been exercised then it has not been verified. However, High Code coverage metrics do not necessarily mean that a design is bug-free at that the verification effort is complete and thorough.
- Functional Coverage- this track whether the verification process has:
 - Filled and emptied every FIFO in the design.
 - Transmitted all package types across a particular channel of the design.
 - Transmitted all packet types across all channels that are cross coverage.

Every verification type can have further two types of verifications:

- Partial Verification- Testing certain values required by the designer.

- Full Verification- Testing every possible value.

Assertions

Assertions are used by designers as placeholders to describe assumptions and behavior associated with the design. Assertions get triggered during a dynamic simulation if the design meets or fails the specification or exemption. Assertions can also be used in a formal verification environment.

4. PROPOSED FLOW

4.1 Backend overview

Figure 4-1 shows the pictorial representation of the backend work of Verifog. To get started with Verifog, it needs three inputs to work with:

- The Verilog top module (RTL file).
- The golden file (which defines the ground truth outputs FOR ALL testcases to be tested).
- Input ports constraints.

The tool parses this all through (core.py). A TVF file is generated, which digests all the parsed info and inputs all into one file. This file is the core file in this process as the cocotb (generic_tester.py, which is signaled to run through core_tester.py) as it is used to compare the results with (as it also digests the golden file), applies ports constraints. In short, this TVF file is a key player in the tool. Next, after the TVF, comes the cocotb work, which uses this file to test using the TVF parsed files. cocotb generic_tester.py runs all testcases specified, gets their output, and compare, providing a table of mismatches in the form of a simple HTML report displayed for a user on request (by clicking “Report button”).

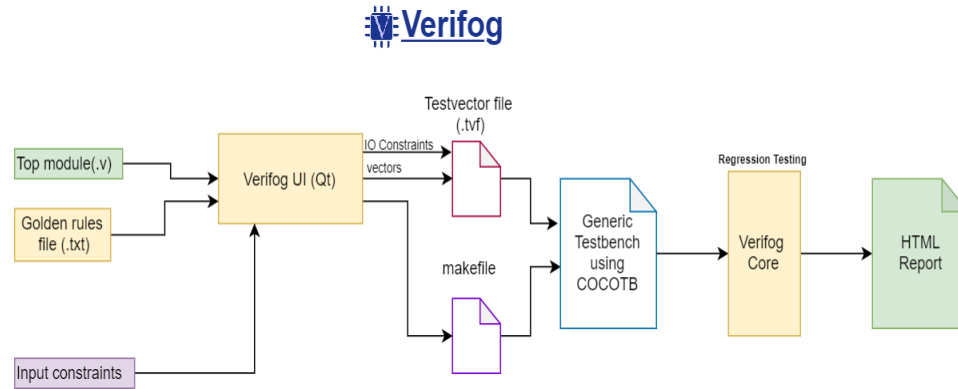


Figure 4-1: Verifog System Flow Chart

Verifog UI (Qt)

The backend code to design the UI (User Input) is in “verifog_ui.py”.

verifog_ui.py: This is the UI part, designed in Qt Designer and transferred into Python using Qt tools (so it is now a code). Most of it (~ 75%) is auto-generated and the rest of 25% is adding code logic to utilize the other scripts above.

- The Verifog UI takes the IO constraints and the vectors to create the TVF (test vector file). The code which works in the background to generate this TVF is in the “core.py” file.
- Another file is also generated in the background, which is the “makefile.”

“core_tester.py” generates the makefile and runs the makefile.

Generating the auto-generated codes from the ui file PyQt has a tool called (Qt Designer). The flow of GUI design in this Tool is simple. A GUI is designed by drag and drop in Qt Designer, then convert it to Python code to add the core logic, ui logic, etc. Qt Designer is normally installed with *pip install PyQt5*. The following step in this part is for generating verifog_ui.py, which is already modified and has lots of things added (as discussed earlier, the code is self generated). To generate it, the below command is used

in the “Anaconda command prompt”:

```
pyuic5 -x verifog2.ui -o verifog_ui.py
```

This will give an executable python code called “verifog_ui.py”.

Generic testbench using cocotb.

Just like UVM (Universal Verification Methodology) is used to build an environment for System Verilog testbenches, “cocotb” is built to create an environment for Python testbenches. Since Python is used as the main scripting language and the testbench is also developed in Python, it was best to build the testbench environment in cocotb for verifying VHDL and System Verilog RTL. Using the cocotb testbench environment allows VHDL or System Verilog for design purpose only, not the testbench itself.

Files used for the python testbench creation using cocotb are mentioned in detail below:

- “generic_tester.py” is the backbone of the backend, which is a cocotb testbench. It takes the .TVF file into account, parses it, applies constraints, runs the Verilog, takes the output, and compares them. In the tool, the “verify” button is doing this in the frontend. It also uses the Makefile. For cocotb to use generic_tester.py, makefile must be ready.
- “core_tester.py” is the one that interacts with the GUI directly. So, it creates the makefile, then runs the make command to run the “generic_tester.py”. Then “generic_tester.py” outputs the report. When the report is ready, “core_tester.py” informs GUI so that when the user clicks the “Report” button in the tool, it fetches the report for the user.

- The “report” button in the tool is comparing the outputs from and puts them in the report.

Verifog Core (Regression) Testing

The regression testing happens inside the “generic_tester.py”. Regression means running all the testcases which happens inside the generic tester code. There is a function cocotb runs to generate the regressions. It’s the “@cocotb.test() def generic_test(dut)”. It sits inside the main test function executed by cocotb.

HTML Report

- This gets the report from cocotb as Jason and uses Python module to convert it into an HTML report.
- The two files it's comparing are coming from the golden file and the cocotb testbench (generic_tester.py). This cocotb testbench gets the output from the Verilog code (RTL code).

4.2 Files and their roles

In this section, we present an overview of the project files and their roles inside the whole image. While developing the Verifog flow, the files are divided into two sets. One is the files related to core (backend) code and the files related to GUI development. Here are the files and their roles:

- Core files related to GUI logic
core.py: This is the main core file; it provides the GUI with all required functions such as parsing Verilog, generating test vector files, etc. In short, it is the hand for the GUI which does anything not related directly to GUI and not related to the testing process itself.

strings.py: This is a file mainly for grouping all string messages presented to a user to keep things organized.

logman.py: This is a file mainly for holding logging procedures and functionality.

- Core files related to testing process

core_tester.py: This is the main joint between the GUI and the testing environment “cocotb”. In short, it prepares it, runs its make file, and notifies the GUI when regression is done and the report is ready.

generic_tester.py: This is the heart of our tool. It applies what the user has specified in his constraints, makes it in the form of a testbench, gets HDL output from the “DUT” and compares it with the golden file input to the tool. Then prepares a report of all mismatches it has seen.

- GUI files

Verifog2.ui: This is the GUI sketch designed in Qt Designer software

verifog_ui.py: This is the main entry file for the tool and its main GUI. This file is autogenerated from the above file as discussed in the previous section of this paper. However, it was then modified to add the interactions with another tool components.

5. VERIFOG OVERVIEW

In this section, a walk-through of the process of Verifog has been provided for a better understanding of how it works. Figure 5-1 represents the Verifog tool. The GUI is divided into several parts:

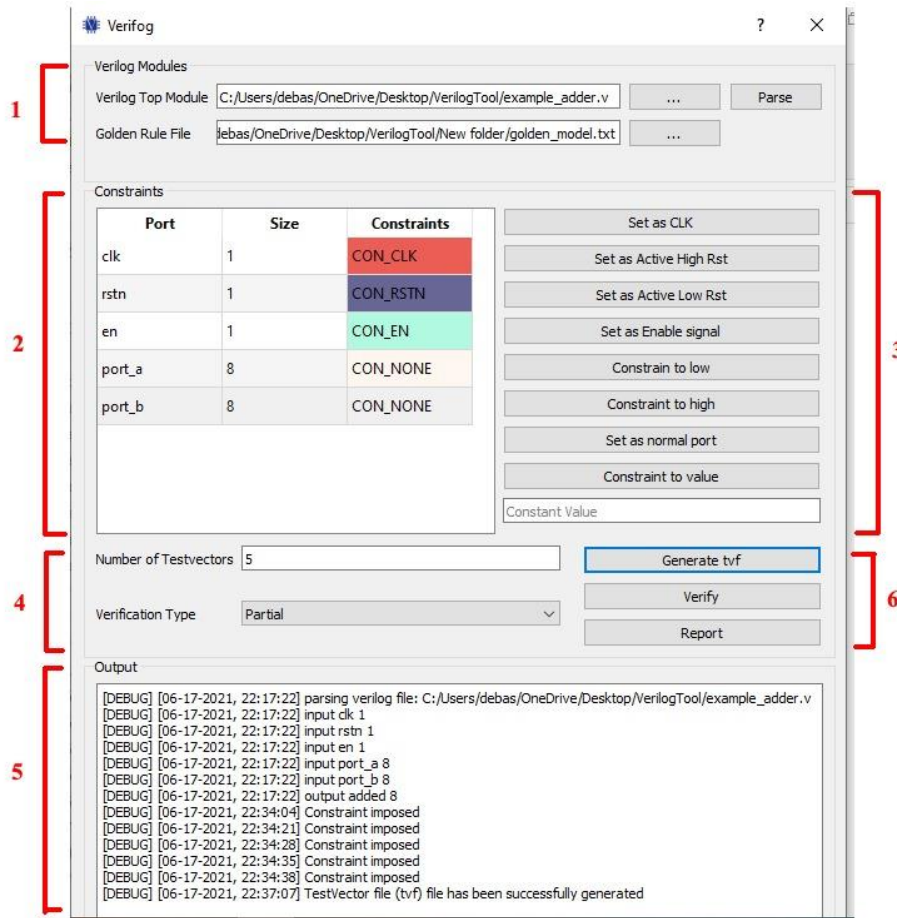


Figure 5-1: Verifog Tool designed by Qt

Designer

1. This part is for reading/parsing Verilog and golden files.
2. This part is for Input's visualization with their own constraints applied.
3. Constraint's area. There are several constraints that can be enforced on a given port.

4. Testing options. The user can select a full testing (all possibilities) or a partial one (limited number of test cases)
5. Output Log.
6. This part is for generating the TVF file (test vector file), applying the verification, and getting reports.

Steps to be followed in the tool:

1. Upload the RTL (design) Verilog file.
2. Upload the Golden file.
3. Parse Verilog file.
4. Set the constraints.
5. Generate the TVF.
6. Verify
7. Report

5.1 More about verifog tool

Detailed description of each of the tabs in Verifog tool as shown in the above section in Figure 5-1 is provided below-

Verifog Top Module: This tab is to upload the Verilog file, which needs to be tested. In this case, “example_adder.v” and “leading_one.v” are used for testing the tool.

Parse tab: After clicking the ‘parse’ tab, inputs of the Verilog testbench will be visible on the ‘Constraint’ window.

- Only inputs will be visible, not output.
- Parsing is parsing the Verilog top module by taking its input to constraint them.

- In DUTs, there are some inputs that may not enter the test vector generation.
Either they are constants or some special signals like clk, rst, enable, etc.
- So, the values (constraint) are specified so that the tool knows which input holds what value and which input will plug into the test vector generation.

Golden Rule File: The tester makes the file through a ‘model’ which models the RTL in any high-level language like python. For example, if a mux is designed, a ‘model’ in python needs to be built, a golden file needs to be generated, and then can be proceeded to the next steps, or the golden file can be taken from anywhere else as well.

- Here in Verifog, a JSON file is being created by hard coding the results of the top module file. NO model in python has been created separately to generate the golden values by itself.
- The golden rule is the file that is used to test the outputs.
- In the digital design flow, modeling is a very beginning step. This is where golden files are usually generated.

Output Window: This window is to show the output log. It’s just for messages for debugging when there is an issue.

Constraints Area

- *Set as CLK/Active high rst/Active low rst/enable signal:* It constraints this port as CLK/ High reset/ Low reset/ enable. So, the testbench will make this signal as CLK/ High reset/ Low reset/ enable signal, and cocotb python code will understand this signal is a CLK/ High reset/ Low reset/ enable signal. Inside the ‘TVF’ the inputs are categorized according to the constraint. The one made as a CLK/ High reset/ Low reset/ enable is marked as a clock/high reset/low

reset/enable. So, the backend responsible for generating the testbench will make this signal as a clock/high reset/low reset or enable. For the cocotb/python to understand which constraint/inputs in the Verilog file are what, the ‘Constraint Area’ is used in Verifog so that cocotb backends knows which are real inputs and which are clocks, resets, or enables.

- *Constraint to High*: Makes the signal logic 1 during all the tests.
- *Constraints to Low*: Constraint to low is the same but with logic low. Both are for logic 1 but signal.
- Buses are ‘*Constraints to values*’ so that the buses will have the value throughout all the tests.
- *Constraint to value*: It is used for constraining an input as a constant value. A user can specify constraint 1-bit value to high or low and constraint buses with “constant bus values”. For example, if an adder is needed to test, but the user has fixed one input on 0X02 and will make the other inputs normal inputs. So, the user is testing 0X02 + testcases. So, the user will constraint the input 1 to a constant value of 0X02 (in binary).

➤ How to do this in the tool

Mark the port, write the value in the value in the textbox being asked about, and click “Set as Constrained value.”

So, all the tabs in the Constraint Area are not used for every Verilog file. It is needed as per the Verilog file. It varies with the DUT and its requirements. In some cases, none of the options in the Constraint Area is required. In those cases, just “set as

normal input” is used, and this is the default for any input which is unconstrained. Few important points to be considered here are:

- This should be matching with the golden file being used for this Verilog file. So, if for example, three inputs are set as normal input, then the golden file should be matched with them.
- So, it has every output port defined for every possible input “set as normal input”.

After constraining the input values from the Verilog file, the “*Output Log*” window shows “constraints imposed.”

Reason for using JSON file over text file[36] [37]

- JSON is well-established standard for parsing and reading data.
- It's more readable parseable with the self-libraries in nearly all languages.
- JSON is a format. It can be written in any extension the user wants.
- Instead of using or making its own format, which is not portable, JSON has a standard format for reading and writing objects.

JSON File (Golden Rule File/ golden_model.txt)

- It consists of the hardcoded results of the top module file (Verilog file). All the possible inputs of the test determines all outputs.
- For example, a line of code-

```
“00000000_00000001”:  
  {“added”: “00000001”};
```

This means when input 1= 000....0 and input 2= 000....1, the output values at these inputs are added: added 00...1. If there are multiple outputs, then it needs to be defined for every possible input.

- So, if the Verilog file will have a block with two inputs, three outputs, and ten test cases for example, then for each test case, the output value needs to be hardcoded.

Hence, the testbench applies an input to HDL, gets output from HDL, opens the golden file, gets the output corresponding to the same input given to the HDL, and then compares both outputs.

Partial Verification/Full Verification

- Assuming having two inputs, each has 8 bits. So, combining them into one vector will be 16 bits. 16 bits = 65536 possible values. So, full verification means testing every possible value for the inputs. Partial is testing a certain amount.
- In Verifog, both Full and Partial verification can be done, but the golden file should match this depending on which verification is required. If FULL, then the golden file should also have FULL golden results for all possible values.

The number of Test Vectors

For any kind of verification (full/partial), how many test cases are required is sent through this tab to the backend.

Test Vector File (TVF)

It's a file holding every option being selected and every constraint being made. This file also contains every test vector being used and, finally every expected result from the golden file.

Why is it generated

- To use in the cocotb Python generic testbench.
- This cocotb/Python testbench will read this file, which is also a JSON file, then interprets it. So, if input is constrained to be a clock, then it takes this input and

applies a clock. If one input is constrained to high, it will take it and make it high, etc. Then takes the test vectors and apply them one by one to the HDL.

- This TVF file is just a juice made from all the inputs the user gave to the tool till now. It is not generating anything new.

Report

The report shows if there is any bug in the Verilog testbench or not. In an actual environment, the design engineer will write his code and will write his golden file and plug them into the tool. Except for preparing the golden files, this tool, in fact, can be extended to a full environment. But this is away from this thesis scope. Since here test benches are written in Python, anything imaginable can be done, including generating complex test cases for most complex scenarios.

5.2 Verifog files used for top-level module

Simple Example: example_adder.v

- Induced errors intentionally in the golden file to see it in the report.
- The report showed two mismatches.

Complex Example: leading_one.v

- Golden file does not have any error in full verification with 256 cases. The report shows no mismatch.
- Partial verification with 32 cases is showing no mismatch.

Leading One detector [38] [39]-

- Detects the position of first one in binary number. For example-

0011 0111 gives output.

first_one_o = 010 (i.e 2)

$\text{no_ones_o} = 0$ (i.e valid point)

- It starts from the most significant bit. Count the position of the first one the user sees.

Reason for considering “Leading_One” as the complex test for this thesis

- LOD is an electronic circuit commonly found in the central processing unit and especially their ALU’s.
- It is used to detect whether the leading bit in a computer word is zero or one, which is used to perform basic binary tests.
- Used in floating-point multiplication, addition, subtraction, and in algorithmic converters.
- Leading one detector is not a (trivial) block-like addition. The addition seems simple while testing a clocked design.

So, two tests have been done to test the tool’s correctness and effectiveness:

- Simple clocked design (Addition)
- More complex combinational design (leading_one). It grabs the position of the first one it sees.

5.3 Configuration

To run the tool, first, a list of software is required to install in the system:

- Anaconda for Python, including Anaconda prompt.
- Python 3.7
- PyQt5
- Qt Designer (It is a tool to make the GUI code more automated. Here, the GUI is used for a usable purpose, so Qt Designer is required along with PyQt5).

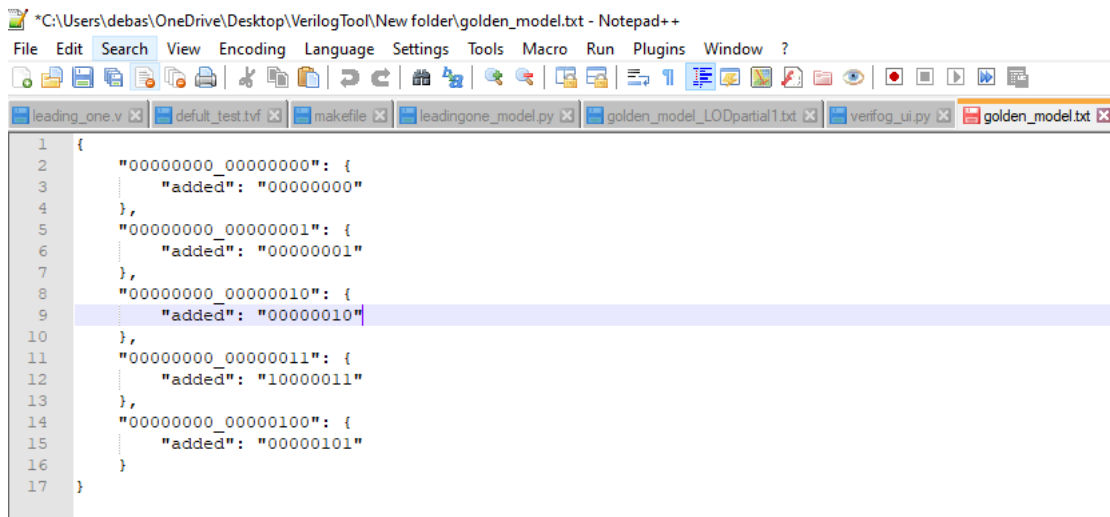
Running the tool

- Open Anaconda Prompt.
- Cd Onedrive\Desktop\VerilogTool (VerilogTool is a folder name where all the files are stored)
- Python verifog_ui.py (Command to open the Verifog_ui.py file)

6. TESTING THE TOOL

6.1 Testing example_adder.v

- Open the Verifog tool.
- Upload the Verilog file (example_adder.v).
- Parse it.
- Constrained the input values-
 - clk – Set as CLK
 - rstn – Set as active low reset.
 - en – Set as enable signal.
 - port_a/port_b – Set as a normal port.
- Upload the golden rule file (golden_model.txt). Figure 6-1 below shows the golden file used for verifying the adder. The golden file consists of the hardcoded results of the top module file (Verilog file). All the possible inputs of the test determines all outputs.



```
1 {
2     "00000000_00000000": {
3         "added": "00000000"
4     },
5     "00000000_00000001": {
6         "added": "00000001"
7     },
8     "00000000_00000010": {
9         "added": "00000010"
10    },
11    "00000000_00000011": {
12        "added": "10000011"
13    },
14    "00000000_00000100": {
15        "added": "00000101"
16    }
17 }
```

Figure 6-1: Adder Golden File

- Set the number of test vectors as 5.
- Generate the TVF. Figure 6-2 below shows the generation of the TVF in the tool.

As shown, all the inputs are constrained, the number of test vectors and verification type is set in Verifog, and then TVF is generated. The output log window shows all the messages associated with the tool.

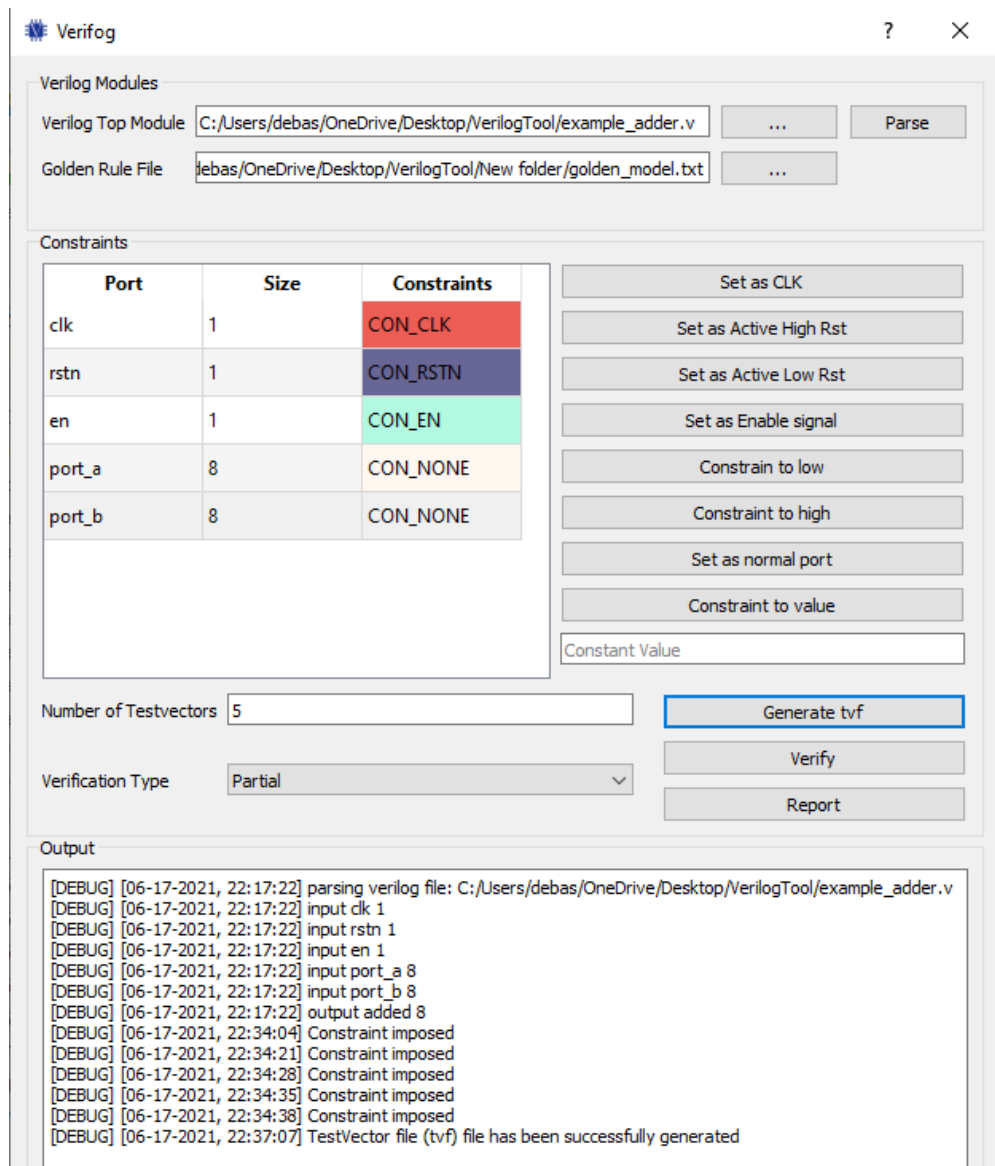


Figure 6-2: Adder TVF

- Verify: This step runs the regression and generates the makefile. Figure 6-3 shows the successful regression run after clicking the ‘verify’ tab.

+

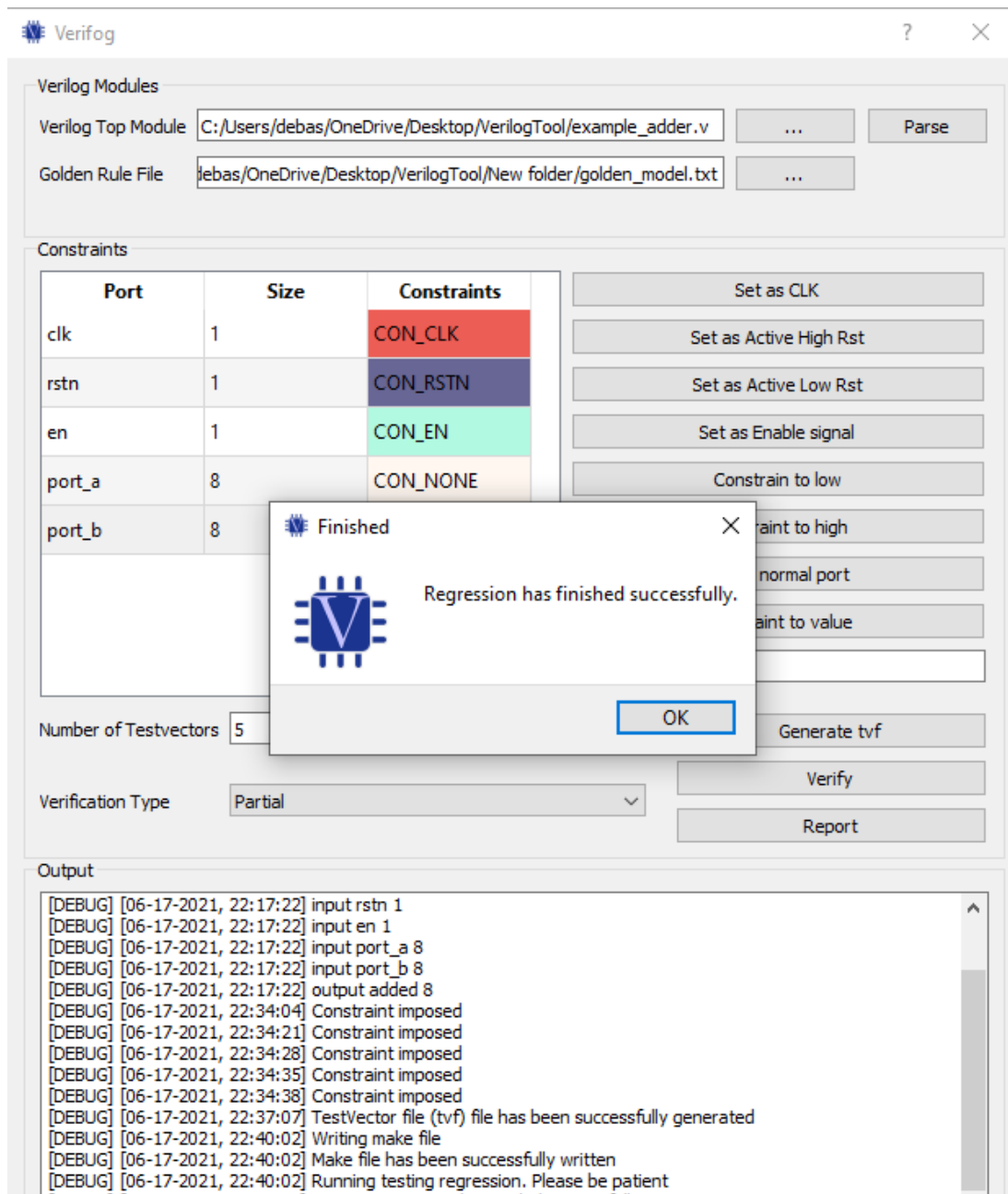


Figure 6-3: Adder Verify Step

- Figure 6-4 shows the auto generated Makefile after a successful regression run.

For cocotb to use generic_tester.py, makefile must be ready.

```

1 SIM ?= icarus
2 MODULE = generic_tester
3 TOPLEVEL = example_adder
4 TOPLEVEL_LANG ?= verilog
5 EXTRA_ARGS =
6 VERILOG_SOURCES =C:/Users/debas/OneDrive/Desktop/VerilogTool/example_adder.v
7 VHDL_SOURCES =
8 include $(shell cocotb-config --makefiles)/Makefile.inc
9 include $(shell cocotb-config --makefiles)/Makefile.sim
10

```

Figure 6-4: Adder Makefile

- The report is showing two mismatches. Figure 6-5 below shows the last report, which tells whether the design has any bugs or not. Here, clearly, the mismatch is shown.

| | |
|---------------------|--|
| n_mismatches | 2 |
| | case_no expected_val output_val |
| mismatches | 3 00000011 10000011 |
| | 4 00000100 00000101 |

Figure 6-5: Adder Report

- Regression Summary. Figure 6-6 below shows the cocotb regression happening in the background in the anaconda shell prompt.

```

Anaconda Powershell Prompt (anaconda3)

*****
TEST                PASS/FAIL  SIM TIME(NS)  REAL TIME(S)  RATIO(NS/S)  **
*****
generic_tester.generic_test  PASS          85.00        0.02         5664.62      **
*****

85.00ns INFO      cocotb.regression                regression.py:565  in _log_sim_summary
*****

                ERRORS : 0                                **
*****

                SIM TIME : 85.00 NS                        **
                REAL TIME : 0.12 S                         **
                SIM / REAL TIME : 689.22 NS/S              **
*****

85.00ns INFO      cocotb.regression                regression.py:255  in tear_down
tear_down...
make[1]: Leaving directory '/c/Users/debas/OneDrive/Desktop/VerilogTool'
QThread: Destroyed while thread is still running
(base) PS C:\Users\debas\OneDrive\Desktop\VerilogTool>

```

Figure 6-6: Adder Regression

6.2 Testing leading_one.v: Partial Verification

- Open the Verifog tool.
- Upload the Verilog file (leading_one.v).
- Parse it.
- Constrained the input values.
 - Set the “in_i” port to “set as normal port”.

Figure 6-7 below shows the constrained inputs of the leading One design.

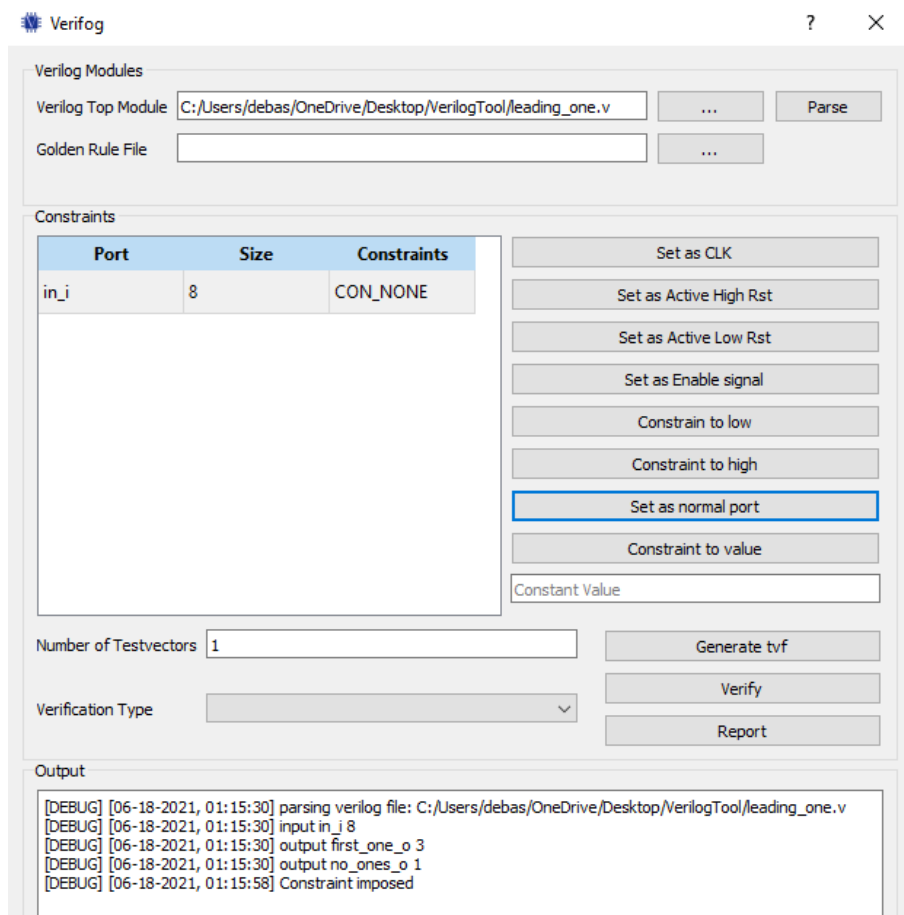
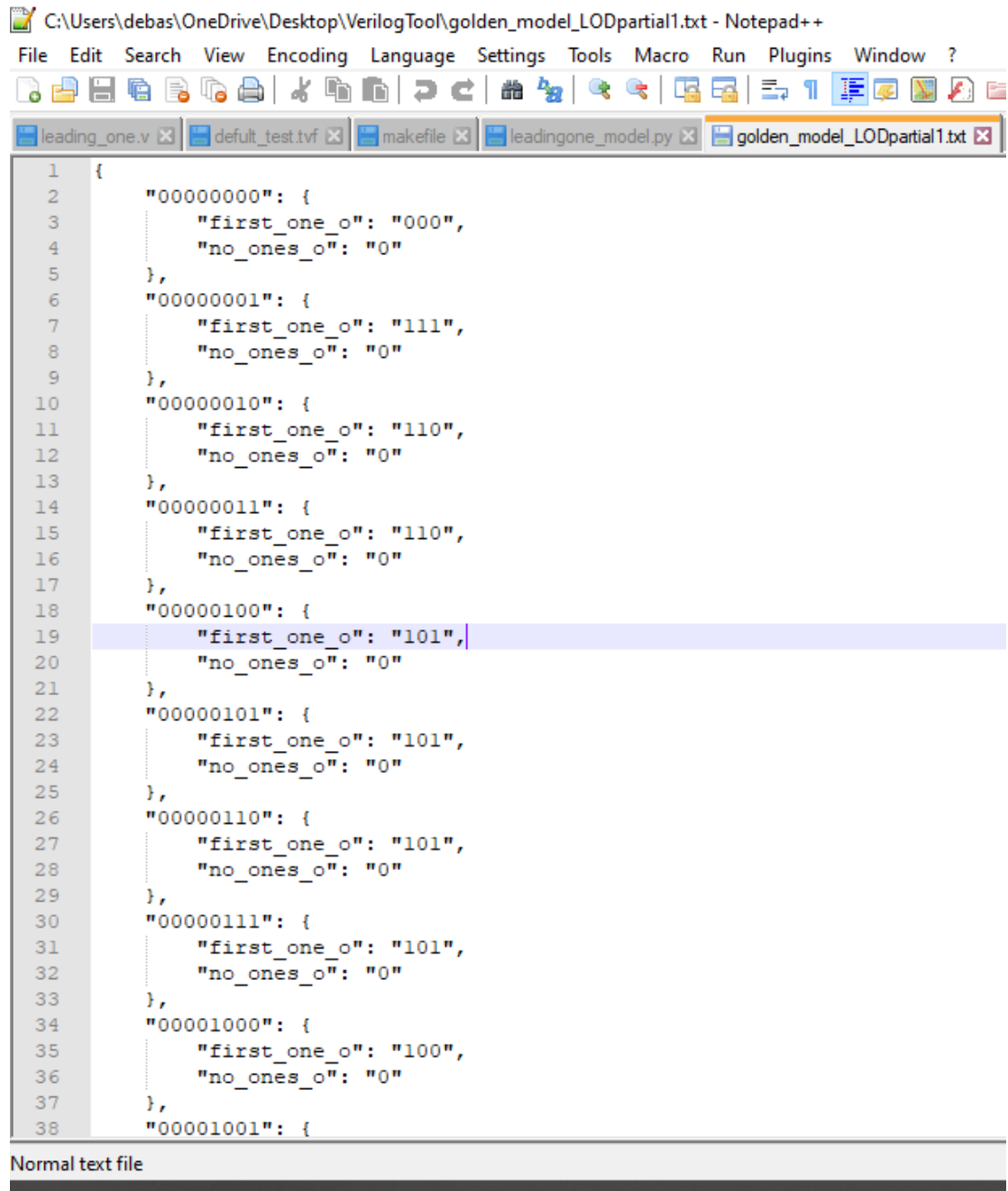


Figure 6-7: Constrained Inputs for Leading One Detector (Partial)

- Upload the golden rule file (golden_model_LODpartial1). Figure 6-8 below shows the leading_one golden file for partial verification. The below screenshot is not the full file.



```
1 {
2     "000000000": {
3         "first_one_o": "000",
4         "no_ones_o": "0"
5     },
6     "000000001": {
7         "first_one_o": "111",
8         "no_ones_o": "0"
9     },
10    "000000010": {
11        "first_one_o": "110",
12        "no_ones_o": "0"
13    },
14    "000000011": {
15        "first_one_o": "110",
16        "no_ones_o": "0"
17    },
18    "000000100": {
19        "first_one_o": "101",
20        "no_ones_o": "0"
21    },
22    "000000101": {
23        "first_one_o": "101",
24        "no_ones_o": "0"
25    },
26    "000000110": {
27        "first_one_o": "101",
28        "no_ones_o": "0"
29    },
30    "000000111": {
31        "first_one_o": "101",
32        "no_ones_o": "0"
33    },
34    "00001000": {
35        "first_one_o": "100",
36        "no_ones_o": "0"
37    },
38    "00001001": {
```

Figure 6-8: Golden File for Leading One Detector (Partial)

- Set the number of test vectors as logic 1.
- Generate the TVF. Figure 6-9 below shows the generation of the TVF in the tool.

As shown, all the inputs are constrained, a number of test vectors and verification

type is set in Verifog, and then TVF is generated. The output log window shows all the messages associated with the tool.

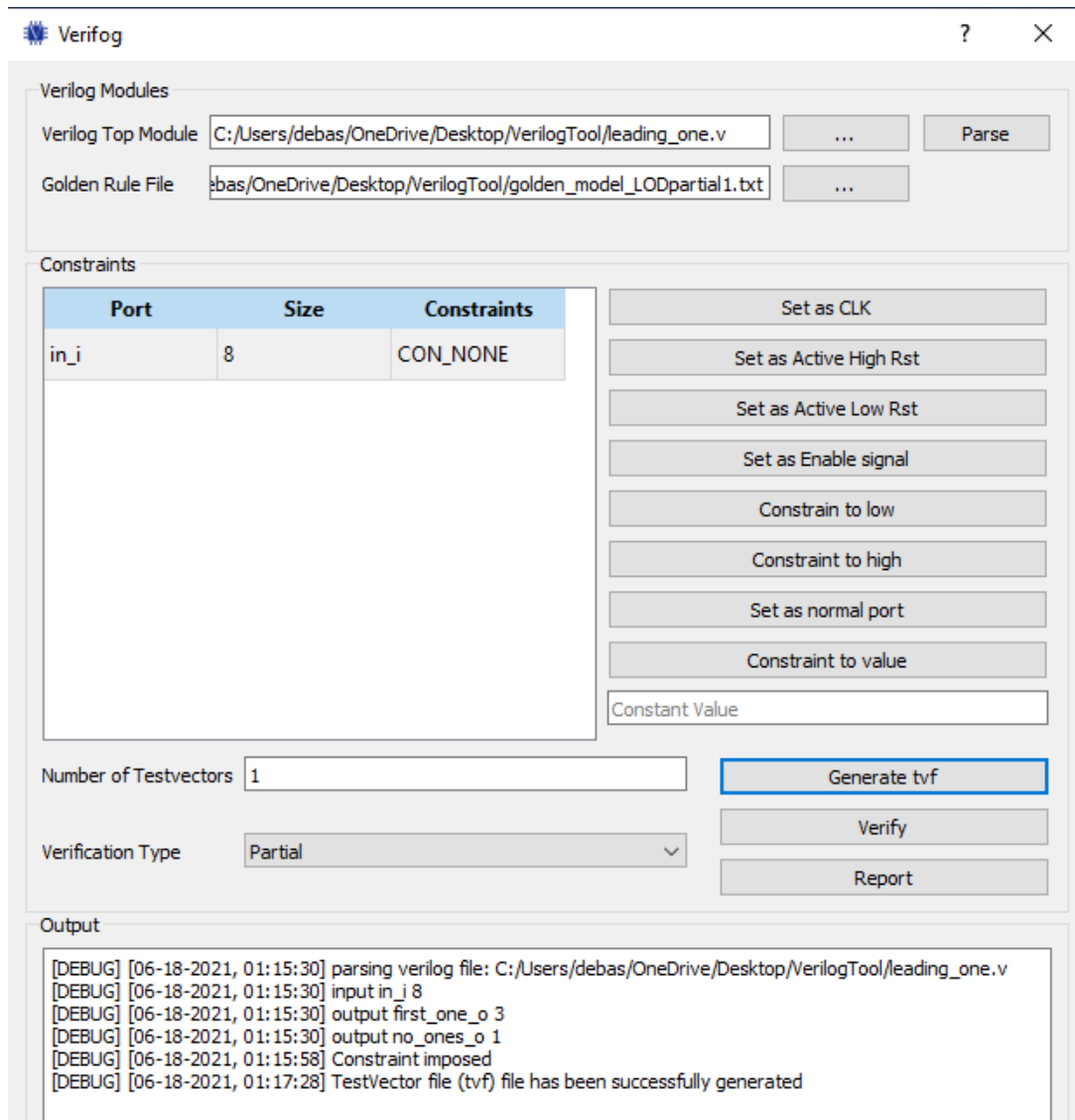
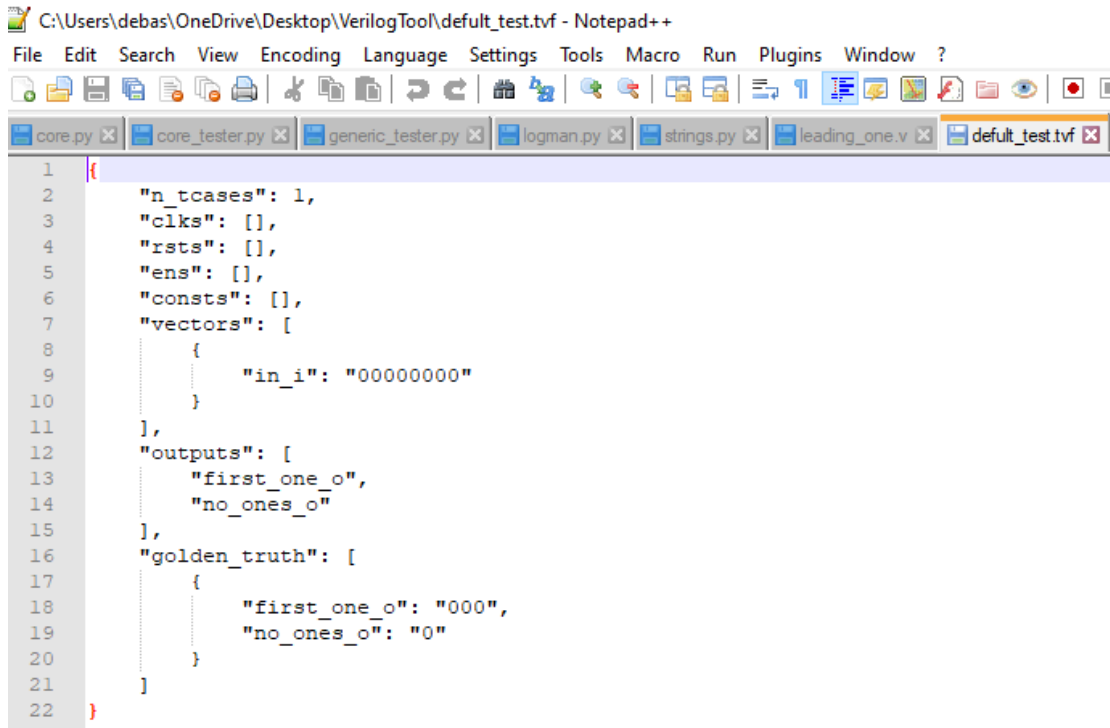


Figure 6-9: Generating TVF for Leading One Detector (Partial) in Verifog

Figure 6-10 shows the auto-generated TVF with 32 cases as partial verification has been chosen.



```
1 {
2   "n_tcases": 1,
3   "clks": [],
4   "rst": [],
5   "en": [],
6   "consts": [],
7   "vectors": [
8     {
9       "in_i": "00000000"
10    }
11  ],
12  "outputs": [
13    "first_one_o",
14    "no_ones_o"
15  ],
16  "golden_truth": [
17    {
18      "first_one_o": "000",
19      "no_ones_o": "0"
20    }
21  ]
22 }
```

Figure 6-10: TVF Inputs for Leading One Detector (Partial)

- Verify. This step runs the regression and generates the makefile. Figure 6-11 shows regression running successfully after clicking the ‘verify’ tab.

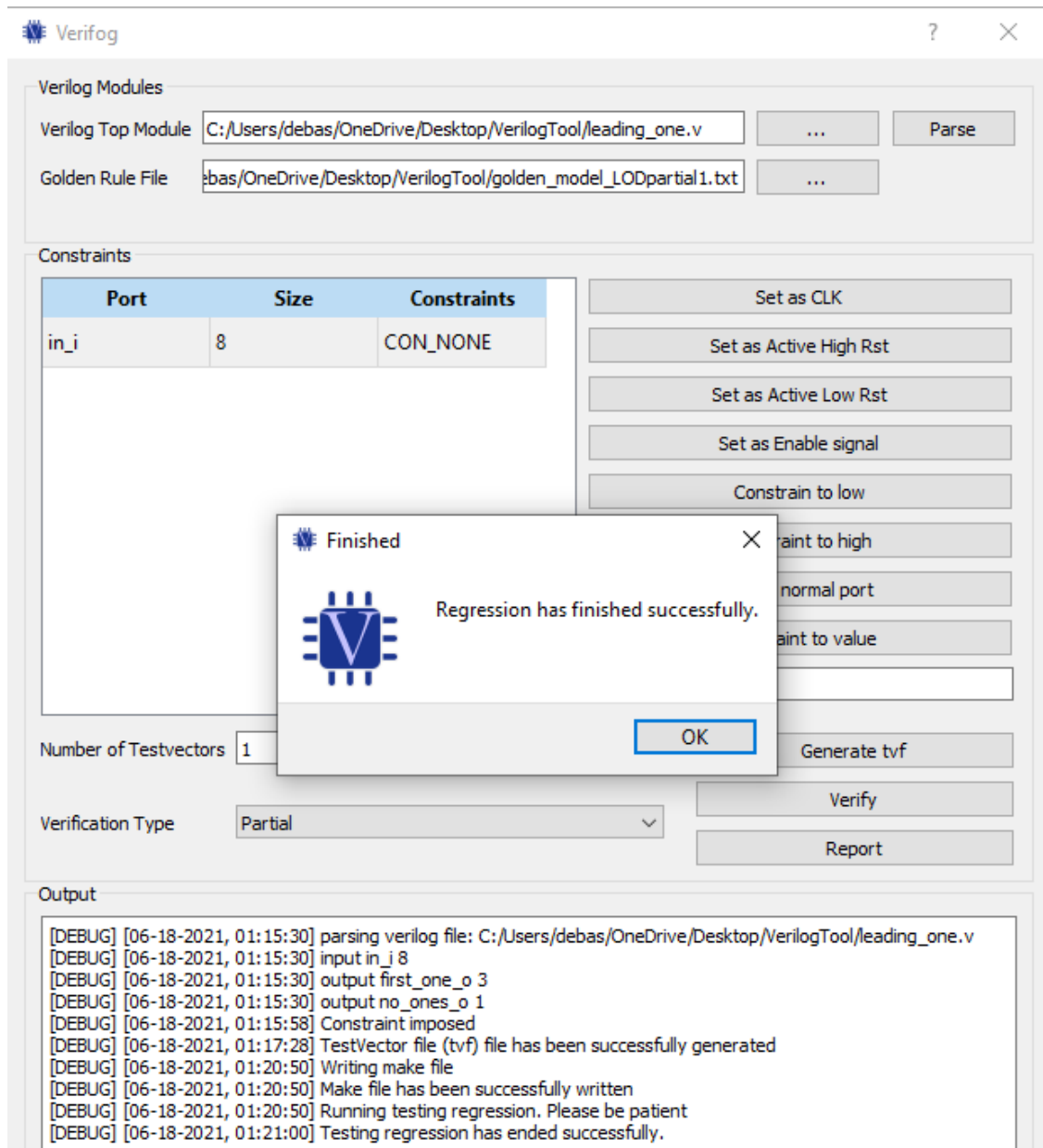


Figure 6-11: Verify Step for Leading One Detector (Partial) in Verifog

- Regression running- Figure 6-12 below shows the cocotb regression happening in the background in the anaconda shell prompt.

```

Anaconda Powershell Prompt (anaconda3)

TEST                PASS/FAIL  SIM TIME(NS)  REAL TIME(S)  RATIO(NS/S)  **
*****
generic_tester.generic_test  PASS          85.00        0.02        5664.62  **
*****

85.00ns INFO      cocotb.regression      regression.py:565  in _log_sim_summary
*****
ERRORS : 0  **
*****

SIM TIME : 85.00 NS  **
REAL TIME : 0.12 S  **
SIM / REAL TIME : 689.22 NS/S  **
*****

85.00ns INFO      cocotb.regression      regression.py:255  in tear_down
tting down...
make[1]: Leaving directory '/c/Users/debas/OneDrive/Desktop/VerilogTool'
QtThread: Destroyed while thread is still running
(base) PS C:\Users\debas\OneDrive\Desktop\VerilogTool>

```

Figure 6-12: Regression for Leading One Detector (Partial)

Figure 6-13 shows the automated generated Makefile after a successful regression run. For cocotb to use generic_tester.py, makefile must be ready.

```

C:\Users\debas\OneDrive\Desktop\VerilogTool\makefile - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

generic_tester.py logman.py strings.py leading_one.v default_test.tvf makefile leadingone_model.py

1 SIM ?= icarus
2 MODULE = generic_tester
3 TOPLEVEL = leading_one
4 TOPLEVEL_LANG ?= verilog
5 EXTRA_ARGS =
6 VERILOG_SOURCES =C:/Users/debas/OneDrive/Desktop/VerilogTool/leading_one.v
7 VHDL_SOURCES =
8 include $(shell cocotb-config --makefiles)/Makefile.inc
9 include $(shell cocotb-config --makefiles)/Makefile.sim
10

```

Figure 6-13: Makefile for Leading One Detector (Partial)

- The report is showing one mismatch. Figure 6-14 below shows the last report which tells whether the design has any bugs or not. Here, clearly, the mismatch is shown.

| | | | |
|---|----------------|---------------------|-------------------|
| <div> <div> <div>←</div> <div>→</div> <div>↻</div> <div>🏠</div> </div> <div> <div>📄</div> <div>File</div> </div> <div> <div>C:/Users/debas/OneDrive/Desktop/VerilogTool/report_html.html</div> </div> </div> <div> <div> <div>📁</div> <div>Apps</div> </div> <div> <div>★</div> <div>Bookmarks</div> </div> <div> <div>🔍</div> <div>Coding Guidelines - ...</div> </div> <div> <div>🔄</div> <div>HEXADECIMAL to B...</div> </div> <div> <div>📄</div> <div>cadence</div> </div> <div> <div>🏠</div> <div>Home - Circuit [For...</div> </div> </div> | | | |
| n_mismatches | 1 | | |
| | case_no | expected_val | output_val |
| mismatches | 0 | 111 | 000 |
| | 0 | 1 | 0 |

Figure 6-14: Report for Leading One Detector (Partial)

6.3 Testing *leading_one.v*: Full Verification

- Open the Verifog tool.
- Upload the Verilog file (example_adder.v).
- Parse it.
- Constrained the input values.
 - Set the “in_i” port to “set as normal port”.

Figure 6-15 shows the constrained inputs of the leading One design for full verification.

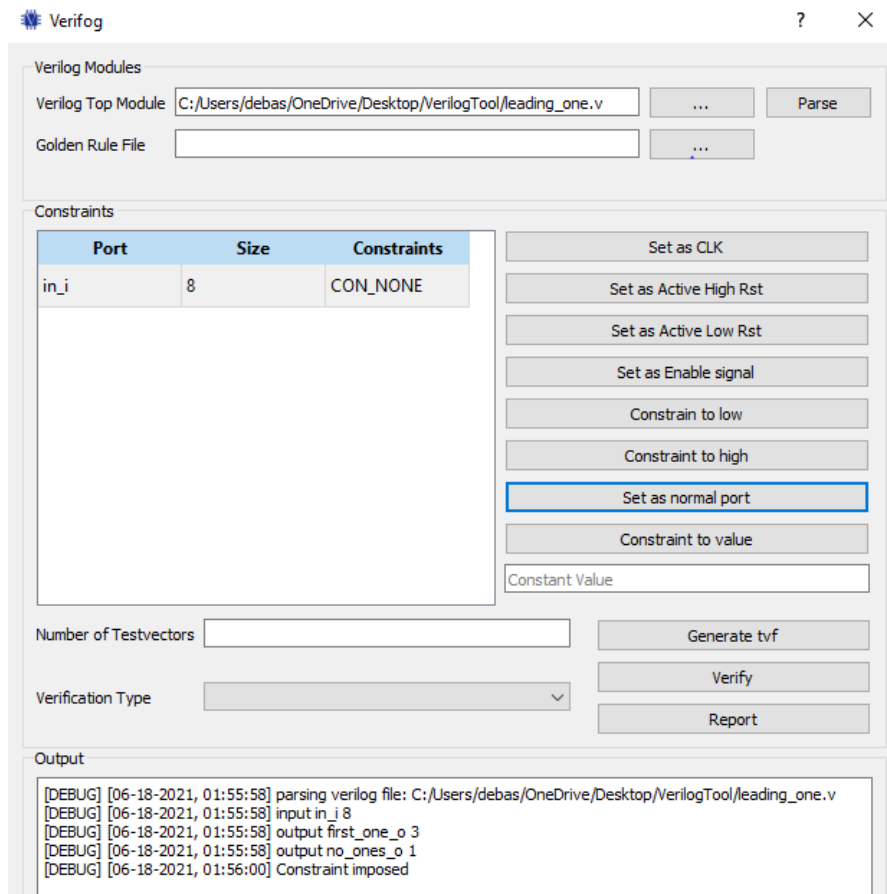


Figure 6-15: Constrained Inputs for Leading One Detector (Full)

- Upload the golden rule file (golden_model_LODfull).
- Set the number of test vectors as 1.
- Generate the TVF. Figure 6-16 shows the generation of the TVF in the tool. As shown, all the inputs are constrained, the number of test vectors and verification type is set in Verifog, and then TVF is generated. The output log window shows all the messages associated with the tool.

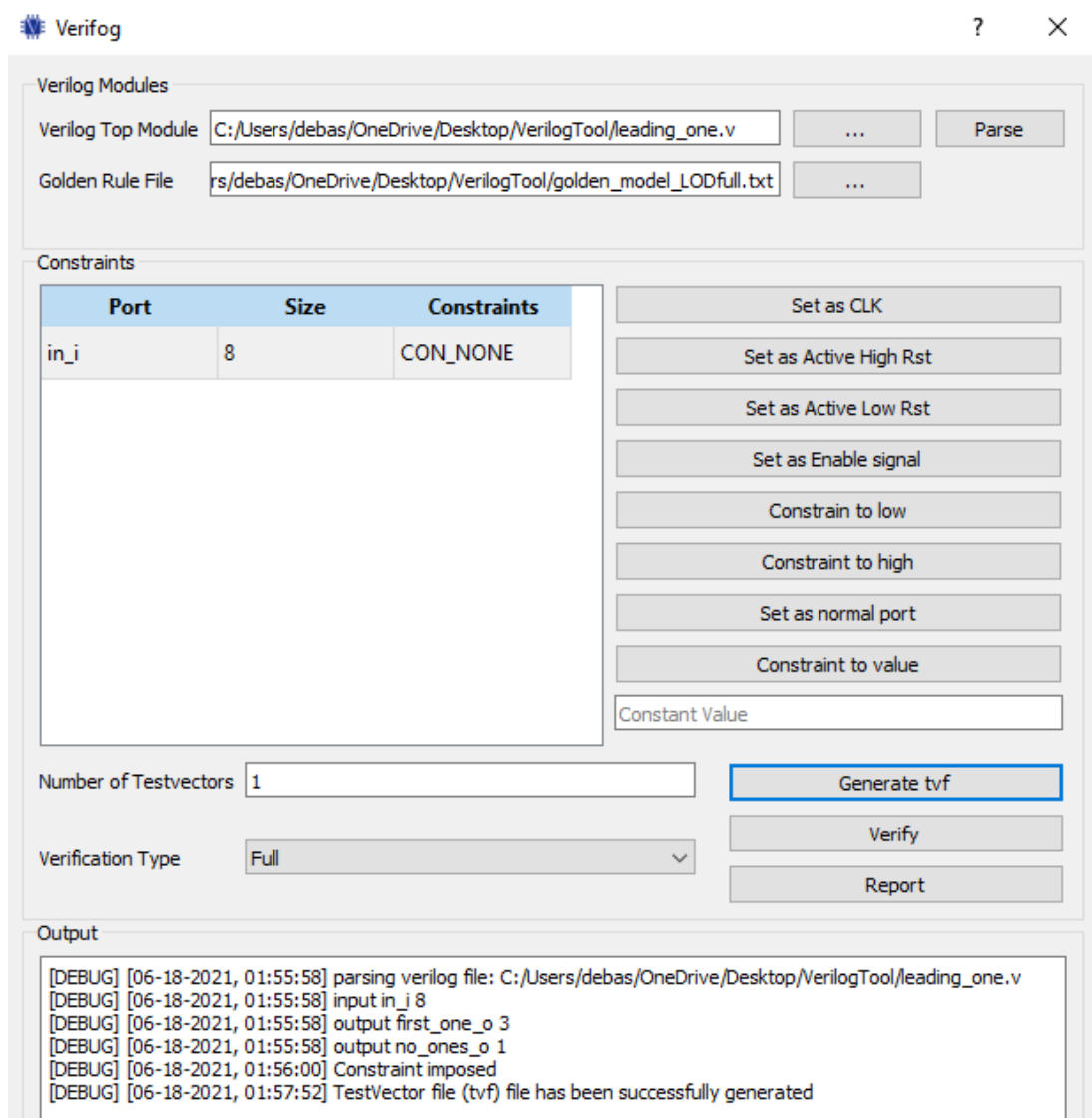
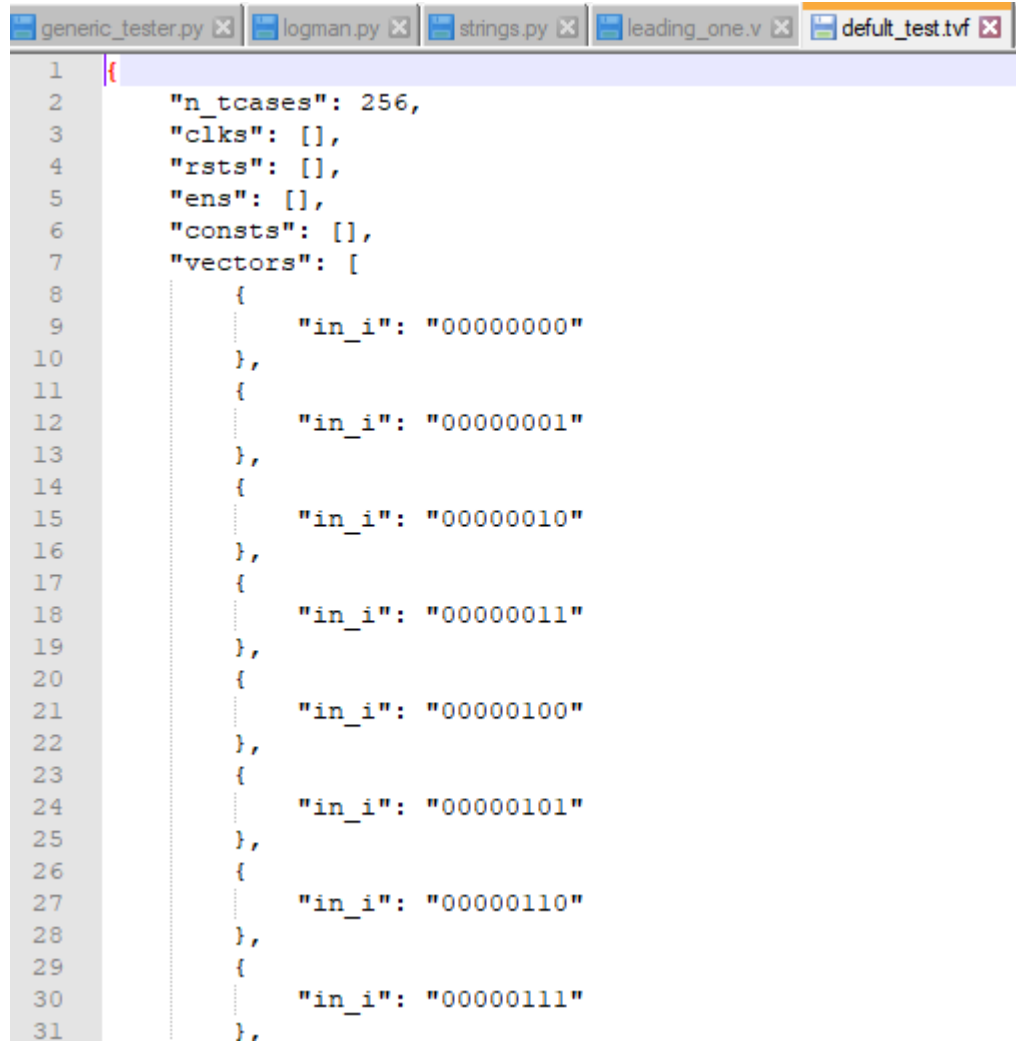


Figure 6-16: TVF Generation for Leading One Detector (Full) in Verifog

Figure 6-17 shows the auto-generated TVF with 256 cases as full verification has been chosen.



```
1 {
2     "n_tcases": 256,
3     "clks": [],
4     "rst": [],
5     "ens": [],
6     "consts": [],
7     "vectors": [
8         {
9             "in_i": "00000000"
10        },
11        {
12            "in_i": "00000001"
13        },
14        {
15            "in_i": "00000010"
16        },
17        {
18            "in_i": "00000011"
19        },
20        {
21            "in_i": "00000100"
22        },
23        {
24            "in_i": "00000101"
25        },
26        {
27            "in_i": "00000110"
28        },
29        {
30            "in_i": "00000111"
31        },
32    ],
33 }
```

Figure 6-17: TVF for Leading One Detector (Full).

- Verify. This step runs the regression and generates the makefile. Figure 6-18 shows regression running successfully after clicking the ‘verify’ tab.

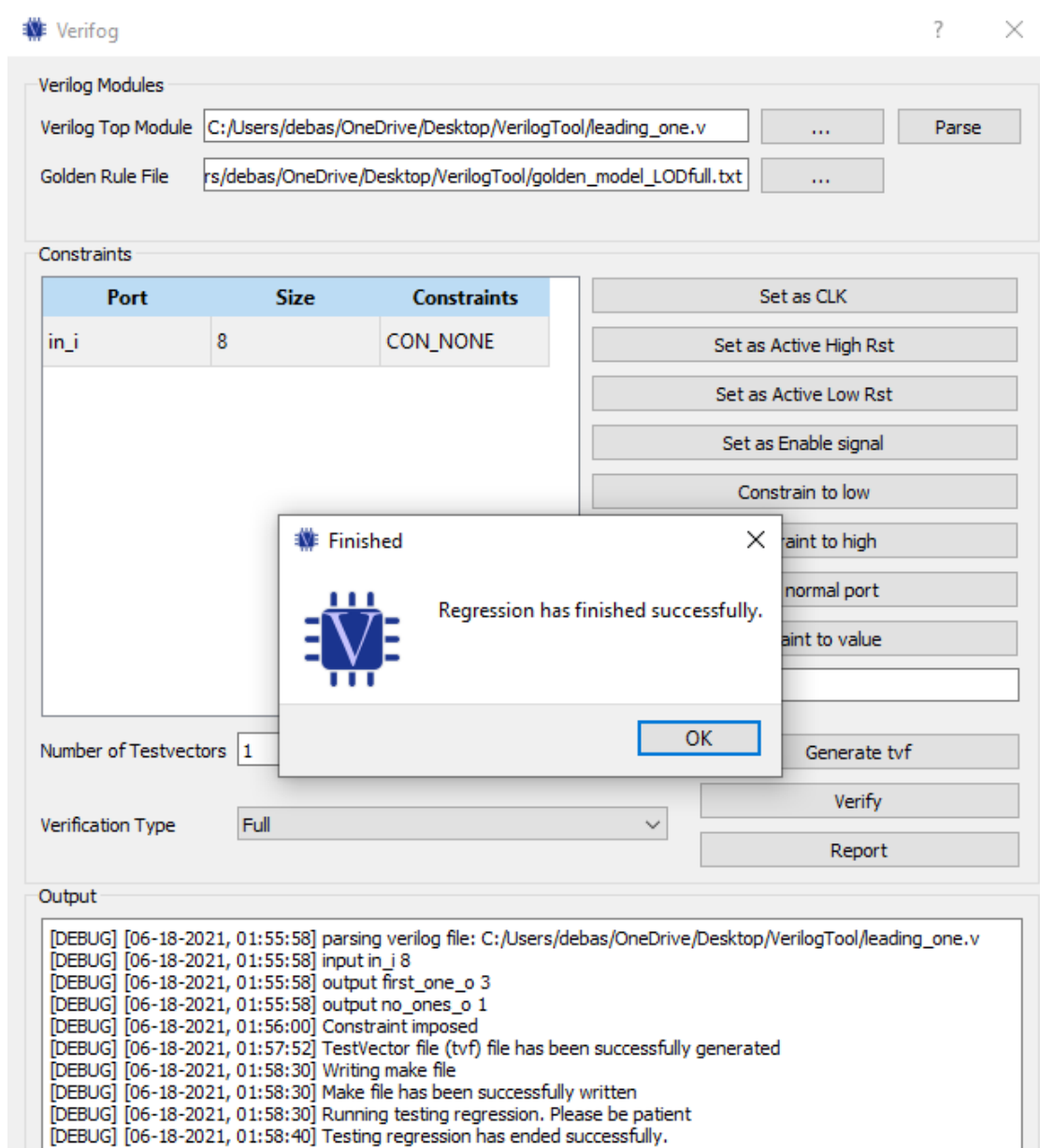


Figure 6-18: Verify Step for Leading One Detector (Full) in Verifog

- Regression running- Figure 6-19 shows the cocotb regression happening in the background in the anaconda shell prompt.

```

Anaconda Prompt (anaconda3) - python verifog_ui.py
3840.00ns INFO cocotb.regression regression.py:548 in _log_test_summary
*****
TEST          PASS/FAIL  SIM TIME(NS)  REAL TIME(S)  RATIO(NS/S)  **
*****
generic_tester.generic_test  PASS          3840.00      0.15      25055.70  **
*****
3840.00ns INFO cocotb.regression regression.py:565 in _log_sim_summary
*****
ERRORS : 0  **
*****
SIM TIME : 3840.00 NS  **
REAL TIME : 0.16 S  **
SIM / REAL TIME : 23943.30 NS/S  **
*****
3840.00ns INFO cocotb.regression regression.py:255 in tear_down
tear down...
make[1]: Leaving directory '/c/Users/debas/OneDrive/Desktop/VerilogTool'

```

Figure 6-19: Regression for Leading One Detector (Full).

Figure 6-20 shows the automated generated Makefile after a successful regression run.

For cocotb to use generic_tester.py, makefile must be ready.

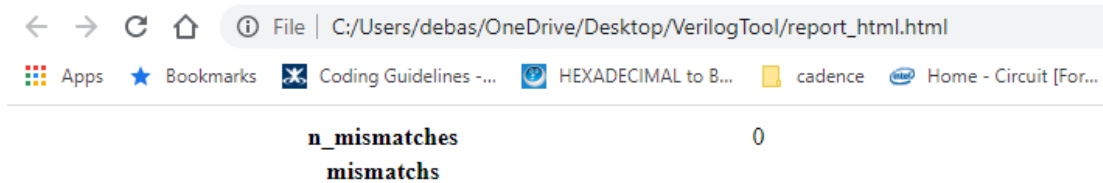
```

C:\Users\debas\OneDrive\Desktop\VerilogTool\makefile - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
generic_tester.py logman.py strings.py leading_one.v default_test.tvf makefile leadingone_model.py
1 SIM ?= icarus
2 MODULE = generic_tester
3 TOPLEVEL = leading_one
4 TOPLEVEL_LANG ?= verilog
5 EXTRA_ARGS =
6 VERILOG_SOURCES =C:/Users/debas/OneDrive/Desktop/VerilogTool/leading_one.v
7 VHDL_SOURCES =
8 include $(shell cocotb-config --makefiles)/Makefile.inc
9 include $(shell cocotb-config --makefiles)/Makefile.sim
10

```

Figure 6-20: Makefile for Leading One Detector (Full)

- The report showing 0 mismatches. Figure 6-21 below shows the last report, which tells whether the design has any bugs or not. In this case, no mismatch is found.



The screenshot shows a web browser window with the address bar displaying 'C:/Users/debas/OneDrive/Desktop/VerilogTool/report_html.html'. The browser's bookmark bar is visible with several icons. Below the browser window, a table displays the report results.

| | |
|---------------------|---|
| n_mismatches | 0 |
| mismatch | |

Figure 6-21: Report for Leading One Detector (Full).

6.4 Tool limitation

While working on Verifog tool, the intension was to make an industry standard debug tool to benefit the DV engineers. But there are certain things which are beyond this tool's scope as of now. Listing below the limitations of Verifog:

- To debug in Verifog, outputs should only depend on current input. A code that has dependencies between cycles, cannot be tested in Verifog. For example- In a code an address has been entered and the output is designed to come after 3 clock cycles (may be due to a reset in the deisgn). Most of the times, these resets are an integral part of the hardware and cant be removed. So DV engineers have to create testbenches in a way so that it vefiryes the design without having the need to remove the reset. In Verifog there is no way to test a design where there are delays in between cycles.
- Verifog is not designed to create the golden file . Here the DV engineers has to either create his own golden file or can run a script to buid the file. But a great

thing will be if Verifog can have a feature which runs a script to generate a golden file based on the design file (Verilog file).

- If there is a design file where the RTL is coded to generate Xs or any random value which the user wont know in the output, it cannot be tested in Verifog. Verifog can only test design files whose outputs are known and deterministic. So, for every set of inputs, the user knows exactly the outputs and those outputs depend on these cycle input.

7. CONCLUSION AND FUTURE WORK

As discussed in the previous section, there are few features which if implemented in future will enhance the tool's capability of debugging to another level. Some of the work that can be done in future are:

- If there is way to test a design with delays (whether a reset delay, or any hardware delay or a delay caused by another block of the design) then no limitations will be set on the designs to be tested. Any design can be tested in that case.
- The tool's capability to create its own golden file can help the DV engineers further in the long run and they don't have to rely on DE's to get the golden file or need not have to build the file by themselves. The tool will be able to generate the file and the DV engineer will only need the Verilog file (design file) to work with the tool.
- Another feature that needs to be added in future work is a way to handle unknown or random output values coming from known inputs of the Verilog file.

Currently the industry standard verification language is Verilog/ System verilog using UVM environment. Its evident that verification language should be real, powerful, highlevel general purpose language. C and C++ are too low level and building environments with them is too time consuming and complex. They tend to 'BUS error' a lot. It can be agreed that the language in which the design is expressed, does not have to be the one that verification is built. Moreover, if the languages are different, recompilation time of the DUT is spared. Verilog code compilation time is much shorter because RTL code compiles really fast and no overhead of test bench compilation occurs.

Combining Python with Verilog opens up an opportunity to use free tools throughout the whole process. It also provides easy switch between commercial simulators, the simulator just sees the DUT which by most part is plain RTL. Today there is a connection between all commercial and free simulators and Python and it is implemented using VPI. Checkers and coverages can be easily combined. Python is rich, powerful, succinct and clear language. It is very stable and practically has no bugs. When connected to Verilog, powerful math libraries can be used which are as good as MATLAB for all the OFDM algorithms. Graphical libraries can also be used to plot results on the fly. Lastly debugging in high level language is so much productive.

Another feature of a good language is code reusability which means the same of code can either work in conjunction with running Verilog or post-processing or pre-processing- thus saving licenses and splitting the work among servers. For example- a model of CPU can be used to verify the RTL of that processor, but also works as a software development tool[40].

So, this tool provides a POC (proof-of-concept) that Python is strong enough to replace legacy used verification languages and environment. This is due to its flexibility, portability, and its vast community, which is wider than the design/verification engineering ones. If more effort is put in, “Verifog” it can be extended to handle more complex scenarios which deal with complex designs.

APPENDIX SECTION

CODES USED

1. Core.py:

```
from enum import Enum

from datetime import datetime

import hdlparse.verilog_parser as vlog

from logman import *

import cocotb

import json

from strings import *

from traceback import print_exc

# Enum class for the constraints

class constr(Enum):

    con_clk      = 'con_clk'.upper()  # clk signal

    con_rstn     = 'con_rstn'.upper() # active low rst signal

    con_rst      = 'con_rst'.upper()  # active hight rst signal

    con_en       = 'con_en'.upper()   # enable signal

    con_high     = 'con_high'.upper() # high

    con_low      = 'con_low'.upper()  # low

    con_val      = 'con_val'.upper()  # some constant

    con_none     = 'con_none'.upper() # not constrained

    con_non_valid = 'con_non_valid'.upper() # for indicating non valid constraints

# Enum class for verification types
```

```

class verification_types(Enum):

    verif_partial=0

    verif_full = 1

# This is the class responsible for parsing verilog/ generating TVF

class testbench_gen:

    def __init__(self,conFigures={ },inputs={ },outputs={ }):

        # holding inputs and outputs with their options (i.e.: sizes, constraints, values if
needed)

        self.inputs=inputs

        self.outputs= outputs


        # holding conFigureurations

        self.conFigures=conFigures

        # verilog extracting

        self.vlog_ex = vlog.VerilogExtractor(

# parsing verilog module into inputs and outputs

    def parse_verilog(self,module_path):

        try:

            self.inputs={}

            self.outputs={}

            # parsing the verilog

            module_objects= self.vlog_ex.extract_objects(module_path)

            #parsing ports to fill both self.inputs and self.outputs

```

```

        for port in module_objects[0].ports:

            if port.mode == 'input':

                self.inputs[port.name]

            ={'constr':constr.con_none,'size':self.__getSize__(port.data_type),'val':None}

                logman.log(loglevel.debug,"{ } { } { }".format(port.mode, port.name,
self.__getSize__(port.data_type)))

            else :

                self.outputs[port.name]

            ={'constr':constr.con_non_valid,'size':self.__getSize__(port.data_type),'val':None}

                logman.log(loglevel.debug,"{ } { } { }".format(port.mode, port.name,
self.__getSize__(port.data_type)))

            return self.inputs

        except Exception as e:

            pass

    # set the conFigures

    def set_conFigures(self,conFigures):

        self.conFigures=conFigures

        return

    # imposing constraint on the input port

    def impose_constraint(self,input_port,constraint_type,constraint_value):

        try:

            if input_port not in list(self.inputs.keys()):

```

```

        # [TODO: exc// adding non existent port]

        return

# If constraint is set to value, then check if the given value from the ui == size of
port

    if (constraint_type == constr.con_val):

        if (self.inputs[input_port]['size'] != len(constraint_value)):

            # [TODO: exc// constraint is not same in length]

            return False,msg_constraint_invalid_size

        # else, all other ports holds other constraints (excpet for normal port) should be 1
bit width

        elif constraint_type != constr.con_none and self.inputs[input_port]['size'] !=1:

            return False,msg_constraint_invalid_size

        self.inputs[input_port]['constr'] = constraint_type

        self.inputs[input_port]['val'] = constraint_value

        return True,"Constraint imposed"

    except Exception as e:

        pass

# generating testvector

def gen_testvectors(self,golden_file='ground_truth.txt'):

    try:

        un_constrained_inputs=[]

        vector_length=0

        # this dict holds all the data to be writting into json TVF

```

```

final_TVF={'n_tcases':0,

          'clks':[],

          'rst':None,

          'ens': [],

          'consts':[],

          'vectors':[],

          'outputs':[],

          'golden_truth':[]

}

# Adding outputs

for output_port,v in self.outputs.items():

    final_TVF['outputs'].append(output_port)

# Adding inputs

for input_port,v in self.inputs.items():

    # Adding clocks

    if v['constr'] ==constr.con_clk:

        final_TVF['clks'].append(input_port)

    # Adding resets (active high and low)

    elif v['constr'] ==constr.con_rst:

        final_TVF['rst'].append((input_port,1,'async'))

    elif v['constr'] ==constr.con_rstn:

        final_TVF['rst']=(input_port,0,'async')

```

```

# Adding enables

elif v['constr'] == constr.con_en:

    final_TVF['ens'].append(input_port)


# Adding constants (High/Low/Value)

elif v['constr'] == constr.con_high:

    final_TVF['consts'].append([input_port, '1'])


elif v['constr'] == constr.con_low:

    final_TVF['consts'].append([input_port, '0'])


elif v['constr'] == constr.con_val:

    final_TVF['consts'].append([input_port, v['val']])


# Finally, normal unconstrained inputs

elif v['constr'] == constr.con_none:

    unconstrained_inputs.append((input_port, v['size']))

    vector_length += v['size']


# Getting the number of testcases

if self.conFigures['verif_type'] == verification_types.verif_full:

    # if full verification, then get all possible values (i.e. 2^total inputs length)

    N = 2**(vector_length)

```



```

else:

    # else, use the number given in the ui (found at self.conFigures)

    N= min( self.conFigures['verif_n'] ,2** (vector_length))

# adding number of testcases and outputs

final_TVF['n_tcases']=N

number_of_outputs=len(final_TVF['outputs'])

# reading the golden file

golden_json = self.read_golden_file(golden_file)

# generating testcases as many as the required in the verification type

for i in range (N):

    # in the next part, we have one vector, holding all inputs concatenated, and we
increment this long vector in binary

    binary= bin(i)[2:].zfill(vector_length)

    # Then partitioning this long binary vector on the un-constrained inputs

    marker=0

    tcase={ }

    for inp in un_constrained_inputs:

        input_name=inp[0]

        input_size=inp[1]

        val = binary[marker:marker+input_size]

        marker+=input_size

        tcase[input_name]=val

    final_TVF['vectors'].append(tcase)

```

```

        # adding the golden file result of the current testcase generated to the TVF dict
        if binary not in golden_json:

            logman.log(loglevel.err,msg_golden_not_all_ips)

            golden_case=golden_json[binary]

            # checking if the number of outputs in the golden file = the number of outputs
already found in the verilog module under-test

            if len(list(golden_case.keys())) !=number_of_outputs:

                logman.log(loglevel.err,msg_golden_not_all_ops)

            # adding to TVF

            final_TVF['golden_truth'].append(golden_case)

        # writing TVF to file

        self.write_tstvector_file(final_TVF)

        return True

    except Exception as e:

        print_exc()

# reading golden file and remove any '_' added for readability
def read_golden_file(self,golden_name):

    try:

        with open(golden_name) as f:

            golden_file=json.load(f)

            # removing any "_" used for readability

            golden_file = { key.replace("_","") : val for key,val in golden_file.items() }

            return golden_file

```

```

except Exception as e:

    print_exc()

# writing TVF, writing a normal json file.

def write_tstvector_file(self,TVF,name='default_test.TVF'):

    try:

        with open(name,'w') as rpt:

            json.dump(TVF,rpt,indent=4)

    except Exception as e:

        pass

# some private functions

# get the size as integer from the data given by the verilog parser. It is better to debug
and add breakpoints to understand it in case u are interested

def __getSize__(self,size_string):

    try:

        if len(size_string)>0:

            arr= size_string.split '[')

            #data_type = arr[0]

            width = arr[1].split(':')[0]

            width = 1+int(width)

        else:

            width=1

        return width

    except Exception as e:

```

```
pass
```

2. Core_tester.py

```
import subprocess
```

```
import json
```

```
from PyQt5 import QtCore, QtGui, QtWidgets
```

```
from logman import loglevel
```

```
from strings import *
```

```
# this class is made for handling the usage of cocotb through make files from the gui
```

```
class tester(QtCore.QObject):
```

```
# signals to gui when done testing, report is done and when we want to write a log
```

```
tester_signal_done = QtCore.pyqtSignal()
```

```
tester_signal_rprt = QtCore.pyqtSignal(dict)
```

```
tester_signal_log = QtCore.pyqtSignal(loglevel,str)
```

```
def __init__(self):
```

```
    super().__init__()
```

```
    self.report=None
```

```
# preparing the make file used with cocotb and writing it in the same folder
```

```
def prepare(self,verilog_path):
```

```
    self.tester_signal_log.emit(loglevel.debug,msg_writing_makefile)
```

```

        verilog_abs_path=verilog_path.replace(" ",r"\ ")

        makefile_str="""SIM ?= icarus

MODULE = generic_tester

TOPLEVEL = example_mux

TOPLEVEL_LANG ?= verilog

EXTRA_ARGS =

VERILOG_SOURCES={}

VHDL_SOURCES =

include $(shell cocotb-conFigure --makefiles)/Makefile.inc

include $(shell cocotb-conFigure --makefiles)/Makefile.sim

""".format(verilog_abs_path)

        with open('makefile','w')as makefile:

            makefile.write(makefile_str)

        self.testersignal_log.emit(loglevel.debug,msg_done_writing_makefile)

# actual run of cocotb through running "make" command in the cmd

def run_make(self):

    self.testersignal_log.emit(loglevel.debug,msg_regression_running)

    process = subprocess.Popen(["make","sim"])

    process.wait()

    if process.returncode==0:

        self.testersignal_log.emit(loglevel.debug,msg_regression_done)

    else:

        self.testersignal_log.emit(loglevel.debug,msg_regression_err)

```

```

# read the report and informing GUI about it

def get_report(self):

    with open('test_report.rpt') as f:

        self.report=json.load(f)

        self.testersignal_rprt.emit(self.report)

# this is a function which runs all above ones in order

# 1- prepare make file, 2- make and finally, 3- notify the gui when report is ready.

def run_regression(self,verilog_path):

    try:

        self.prepare(verilog_path)

        self.run_make()

        self.get_report()

    except Exception as e:

        pass

```

3. Generic_tester.py

```

import cocotb

from cocotb.clock import Clock

from cocotb.triggers import Timer, RisingEdge,FallingEdge,Edge

from cocotb.result import TestFailure

import json

```

```
# This is the hidden backbone of the process. generic cocotb testbench.  
# you won't get it the best way unless you read cocotb documentations.
```

```
class generic_driver:  
    def __init__(self,dut,units='ns'):  
        self.dut=dut  
        self.units=units  
        self.map_ports={'clock':None,'enable':None,'reset':(None,0,'sync'),  
                        'constants':[],  
                        'outputs':[],  
                        }  
  
        self.n_testvectors=0  
        self.report_list=[]  
  
    # parsing TVF and translating into digital logic things  
    def parse_TVF(self,path='default_test.TVF'):  
        with open(path) as f:  
            TVF=json.load(f)  
  
        self.TVF=TVF  
        self.n_testvectors =TVF['n_tcases']
```

```

# parsing clocks

if len(TVF['clks'])>0:

    self.map_ports['clock'] = getattr(self.dut,TVF['clks'][0])


# parsing resets

if len(TVF['rst'])==3:

    self.map_ports['reset'] = ( getattr(self.dut,TVF['rst'][0]), TVF['rst'][1]
,TVF['rst'][2])


# parsing enables

if len(TVF['ens'])>0:

    self.map_ports['enable'] = getattr(self.dut,TVF['ens'][0])


# parsing constants

if len(TVF['consts'])>0:

    for const in TVF['consts']:

        port_name,value =const[0],const[1]

        port    = getattr(self.dut,port_name)

        value_int = int(value,2)

        self.map_ports['constants'].append( (port,value_int) )


# parsing outputs

self.map_ports['outputs'].extend(TVF['outputs'])

```



```

# This function will take care of reset part (if found)

# i.e.: asserts the rests if the design has any pin constrained as rst signal

# [TODO] use rst_type (sync/async)

@cocotb.coroutine

def reset_dut(self,duration):

    cocotb.log.info("-----")

    if self._is_resetable_design() ==False:

        return

    rst_type =self.map_ports['reset'][2]

    rst_level=self.map_ports['reset'][1]

    RST_ASRT = rst_level

    RST_DSRT = 1 if RST_ASRT==0 else 0

    rst = self.map_ports['reset'][0]

    rst <= RST_DSRT

    yield Timer(duration/2, units=self.units)

    rst <= RST_ASRT

    yield Timer(duration, units=self.units)

    rst <= RST_DSRT

```

```

# Taking care of clock part

# i.e.: applies a clock on the pin constrained as clock pin

async def clock_dut(self, clock_period):

    if self._is_clocked_design() == False:

        return

    # create the clock

    self.clock = Clock(self.map_ports['clock'], clock_period, self.units)

    # starting it

    cocotb.fork(self.clock.start()) # Start the clock


# Taking care of constant inputs and enables

# [TODO] make enable value arbitrary instead of one

async def constants_dut(self):

    # handling enables

    if self._is_enableable_design():

        ENABLE_LEVEL=1

        enable_port = self.map_ports['enable']

        enable_port <= ENABLE_LEVEL


    # handling constants

    for port, value_int in self.map_ports['constants']:

```

```

        port <= value_int

# adding inputs of the normal ports from the testcase

# it takes testcase order, gets it from the TVF and applies it
async def apply_testcase(self,i):

    # [TODO] more exception handling

    tcase = self.TVF['vectors'][i]

    for port_name,bin_val in tcase.items():

        port = getattr(self.dut,port_name)

        port <= int(bin_val,2)

# compare using the golden truth part in the TVF

def compare_against_golden(self,i):

    gcase = self.TVF['golden_truth'][i]

    case_match=True

    for port_name,bin_val in gcase.items():

        out_port = getattr(self.dut,port_name)

        if str(out_port.value) != bin_val:

            case_match=False

self.report_list.append({'case_no':i,'expected_val':str(out_port.value),'output_val':bin_val
    })

```

```

        return case_match

        #port <= int(bin_val,2)

# checks if this design is clocked by checking if there is a ping marked as clock
def _is_clocked_design(self):

    return self.map_ports['clock'] is not None

# checks if this design is rst-able by checking if there is a ping marked as rst
def _is_resetable_design(self):

    return self.map_ports['reset'][0] is not None

# checks if this design has enables by checking if there is a ping marked as enable
def _is_enableable_design(self):

    return self.map_ports['enable'] is not None

# just makes output from the verilog code as string for logging into cocotb
def _get_outputs_string(self):

    s = [ "{}".format(getattr(self.dut,o).value) for o in self.map_ports['outputs'] ]

    return " ".join(s)

# this is the main test function executed by cocotb

```

```

@cocotb.test()

def generic_test(dut):

    mismatches=0

    #10 ns

    CLK_PERIOD=10

    # main driver holding all objects

    driver= generic_driver(dut)

    # parsing the file resulting from the GUI

    driver.parse_TVF()


    # apply clock signal (if the design has a clock)

    yield driver.clock_dut(CLK_PERIOD)

    # this will reset the design (if the design has reset signal) for 2.5 clock cycles

    yield driver.reset_dut(CLK_PERIOD*2.5)


    # Now, applying constants (if any)

    yield driver.constants_dut()


    # Now, we can apply the inputs. we will wait a clock rising edge if the design is
    clocked. Otherwise, we can wait for small time

```

```

if driver._is_clocked_design():
    time_align = RisingEdge(driver.map_ports['clock'])
else:
    time_align = Timer(CLK_PERIOD, units=driver.units)

# Applying inputs
for i in range(driver.n_testvectors):
    # wait for the time
    yield time_align
    # apply inputs
    yield driver.apply_testcase(i)
    # wait again for half period
    yield Timer(CLK_PERIOD/2, units=driver.units)
    # get the result and compare
    match = driver.compare_against_golden(i)
    if match==False:
        mismatches=mismatches+1
    cocotb.log.info("outputs= {}".format(driver._get_outputs_string()))

#reporting
report={
    'n_mismatches':mismatches,
    'mismatches': driver.report_list

```

```
}
```

```
with open('test_report.rpt','w') as rpt:
```

```
    json.dump(report,rpt,indent=4)
```

4. Verifog_ui.py:

```
# -*- coding: utf-8 -*-
```

```
# Form implementation generated from reading ui file 'Verfog2.ui'
```

```
#
```

```
# Created by: PyQt5 UI code generator 5.15.0
```

```
#
```

```
# WARNING: Any manual changes made to this file will be lost when pyuic5 is
```

```
# run again.
```

```
# most of this file is auto-generated.
```

```
from PyQt5 import QtCore, QtGui, QtWidgets
```

```
from core import *
```

```
from core_tester import *
```

```
from strings import *
```

```

from functools import partial

import logman

import json2table

import os


class Ui_Dialog(QtCore.QObject):

    # signal from the ui to the object responsible for doing testing with cocoth

    start_regression = QtCore.pyqtSignal(str)


    # auto generated

    def setupUi(self, Dialog):

        self.dialog =Dialog

        Dialog.setObjectName("Dialog")


        Dialog.resize(619, 728)

        self.gb_output = QtWidgets.QGroupBox(Dialog)

        self.gb_output.setGeometry(QtCore.QRect(10, 490, 601, 231))

        self.gb_output.setObjectName("gb_output")

        self.txt_output = QtWidgets.QTextEdit(self.gb_output)

        self.txt_output.setGeometry(QtCore.QRect(10, 20, 581, 201))

        self.txt_output.setObjectName("txt_output")

        self.gb_input = QtWidgets.QGroupBox(Dialog)

        self.gb_input.setGeometry(QtCore.QRect(10, 10, 601, 101))

```



```

self.gb_input.setObjectName("gb_input")

self.gridLayoutWidget = QtWidgets.QWidget(self.gb_input)

self.gridLayoutWidget.setGeometry(QtCore.QRect(10, 20, 581, 111))

self.gridLayoutWidget.setObjectName("gridLayoutWidget")

self.gridLayout = QtWidgets.QGridLayout(self.gridLayoutWidget)

self.gridLayout.setContentsMargins(0, 0, 0, 0)

self.gridLayout.setObjectName("gridLayout")

self.btn_parse = QtWidgets.QPushButton(self.gridLayoutWidget)

sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Fixed,
QtWidgets.QSizePolicy.Fixed)

sizePolicy.setHorizontalStretch(0)

sizePolicy.setVerticalStretch(0)

sizePolicy.setHeightForWidth(self.btn_parse.sizePolicy().hasHeightForWidth())

self.btn_parse.setSizePolicy(sizePolicy)

self.btn_parse.setObjectName("btn_parse")

self.gridLayout.addWidget(self.btn_parse, 0, 3, 1, 1)

self.btn_browse_verilog = QtWidgets.QPushButton(self.gridLayoutWidget)

sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Fixed,
QtWidgets.QSizePolicy.Fixed)

sizePolicy.setHorizontalStretch(0)

sizePolicy.setVerticalStretch(0)

sizePolicy.setHeightForWidth(self.btn_browse_verilog.sizePolicy().hasHeightForWidth())

```

```

)

self.btn_browse_verilog.setSizePolicy(sizePolicy)

self.btn_browse_verilog.setObjectName("btn_browse_verilog")

self.gridLayout.addWidget(self.btn_browse_verilog, 0, 2, 1, 1)

self.txt_goldeb_path = QtWidgets.QLineEdit(self.gridLayoutWidget)

self.txt_goldeb_path.setObjectName("txt_goldeb_path")

self.gridLayout.addWidget(self.txt_goldeb_path, 1, 1, 1, 1)

self.lbl_verilog_top_module = QtWidgets.QLabel(self.gridLayoutWidget)

self.lbl_verilog_top_module.setObjectName("lbl_verilog_top_module")

self.gridLayout.addWidget(self.lbl_verilog_top_module, 0, 0, 1, 1)

self.lbl_golden_file = QtWidgets.QLabel(self.gridLayoutWidget)

self.lbl_golden_file.setObjectName("lbl_golden_file")

self.gridLayout.addWidget(self.lbl_golden_file, 1, 0, 1, 1)

self.btn_browse_golden = QtWidgets.QPushButton(self.gridLayoutWidget)

sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Fixed,
QtWidgets.QSizePolicy.Fixed)

sizePolicy.setHorizontalStretch(0)

sizePolicy.setVerticalStretch(0)

sizePolicy.setHeightForWidth(self.btn_browse_golden.sizePolicy().hasHeightForWidth())

)

self.btn_browse_golden.setSizePolicy(sizePolicy)

self.btn_browse_golden.setObjectName("btn_browse_golden")

```

```

self.gridLayout.addWidget(self.btn_browse_golden, 1, 2, 1, 1)

self.txt_verilog_path = QtWidgets.QLineEdit(self.gridLayoutWidget)

self.txt_verilog_path.setObjectName("txt_verilog_path")

self.gridLayout.addWidget(self.txt_verilog_path, 0, 1, 1, 1)

spacerItem = QtWidgets.QSpacerItem(20, 40, QtWidgets.QSizePolicy.Minimum,
QtWidgets.QSizePolicy.Expanding)

self.gridLayout.addItem(spacerItem, 3, 2, 1, 1)

self.gb_constraints = QtWidgets.QGroupBox(Dialog)

self.gb_constraints.setGeometry(QtCore.QRect(10, 110, 601, 381))

self.gb_constraints.setObjectName("gb_constraints")

self.horizontalLayoutWidget_3 = QtWidgets.QWidget(self.gb_constraints)

self.horizontalLayoutWidget_3.setGeometry(QtCore.QRect(10, 20, 579, 261))

self.horizontalLayoutWidget_3.setObjectName("horizontalLayoutWidget_3")

self.horizontalLayout_3 =
QtWidgets.QHBoxLayout(self.horizontalLayoutWidget_3)

self.horizontalLayout_3.setContentsMargins(0, 0, 0, 0)

self.horizontalLayout_3.setObjectName("horizontalLayout_3")

self.tableWidget = QtWidgets.QTableWidget(self.horizontalLayoutWidget_3)

sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Expanding)

sizePolicy.setHorizontalStretch(0)

sizePolicy.setVerticalStretch(0)

sizePolicy.setHeightForWidth(self.tableWidget.sizePolicy().hasHeightForWidth())

```

```

self.tableWidget.setSizePolicy(sizePolicy)

self.tableWidget.setFrameShape(QtWidgets.QFrame.StyledPanel)

self.tableWidget.setFrameShadow(QtWidgets.QFrame.Plain)


self.tableWidget.setSizeAdjustPolicy(QtWidgets.QAbstractScrollArea.AdjustToContents
)

self.tableWidget.setAlternatingRowColors(True)

self.tableWidget.setShowGrid(True)

self.tableWidget.setRowCount(0)

self.tableWidget.setObjectName("tableWidget")

self.tableWidget.setColumnCount(3)

item = QtWidgets.QTableWidgetItem()

self.tableWidget.setHorizontalHeaderItem(0, item)

item = QtWidgets.QTableWidgetItem()

self.tableWidget.setHorizontalHeaderItem(1, item)

item = QtWidgets.QTableWidgetItem()

self.tableWidget.setHorizontalHeaderItem(2, item)

self.tableWidget.horizontalHeader().setCascadingSectionResizes(False)

self.tableWidget.verticalHeader().setVisible(False)

self.tableWidget.verticalHeader().setCascadingSectionResizes(True)

self.horizontalLayout_3.addWidget(self.tableWidget)

self.verticalLayout_4 = QtWidgets.QVBoxLayout()

self.verticalLayout_4.setObjectName("verticalLayout_4")

```

```

self.btn_con_clk = QtWidgets.QPushButton(self.horizontalLayoutWidget_3)

self.btn_con_clk.setObjectName("btn_con_clk")

self.verticalLayout_4.addWidget(self.btn_con_clk)

self.btn_con_rst_high = QtWidgets.QPushButton(self.horizontalLayoutWidget_3)

sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Fixed)

sizePolicy.setHorizontalStretch(0)

sizePolicy.setVerticalStretch(0)

sizePolicy.setHeightForWidth(self.btn_con_rst_high.sizePolicy().hasHeightForWidth())

self.btn_con_rst_high.setSizePolicy(sizePolicy)

self.btn_con_rst_high.setObjectName("btn_con_rst_high")

self.verticalLayout_4.addWidget(self.btn_con_rst_high)

self.btn_con_rst_low = QtWidgets.QPushButton(self.horizontalLayoutWidget_3)

self.btn_con_rst_low.setObjectName("btn_con_rst_low")

self.verticalLayout_4.addWidget(self.btn_con_rst_low)

self.btn_con_en = QtWidgets.QPushButton(self.horizontalLayoutWidget_3)

self.btn_con_en.setObjectName("btn_con_en")

self.verticalLayout_4.addWidget(self.btn_con_en)

self.btn_con_low = QtWidgets.QPushButton(self.horizontalLayoutWidget_3)

self.btn_con_low.setObjectName("btn_con_low")

self.verticalLayout_4.addWidget(self.btn_con_low)

self.btn_con_high = QtWidgets.QPushButton(self.horizontalLayoutWidget_3)

```

```

self.btn_con_high.setObjectName("btn_con_high")
self.verticalLayout_4.addWidget(self.btn_con_high)

self.btn_con_none = QtWidgets.QPushButton(self.horizontalLayoutWidget_3)
self.btn_con_none.setObjectName("btn_con_none")
self.btn_con_none.setText("Set as normal port")
self.verticalLayout_4.addWidget(self.btn_con_none)

self.btn_con_val = QtWidgets.QPushButton(self.horizontalLayoutWidget_3)
self.btn_con_val.setObjectName("btn_con_val")
self.verticalLayout_4.addWidget(self.btn_con_val)

self.txt_con_val = QtWidgets.QLineEdit(self.horizontalLayoutWidget_3)
self.txt_con_val.setObjectName("txt_con_val")
self.txt_con_val.setPlaceholderText("Constant Value")
self.verticalLayout_4.addWidget(self.txt_con_val)
self.horizontalLayout_3.addLayout(self.verticalLayout_4)
self.formLayoutWidget_3 = QtWidgets.QWidget(self.gb_constraints)
self.formLayoutWidget_3.setGeometry(QtCore.QRect(10, 290, 371, 83))
self.formLayoutWidget_3.setObjectName("formLayoutWidget_3")
self.formLayout_3 = QtWidgets.QFormLayout(self.formLayoutWidget_3)

```

```

self.formLayout_3.setContentsMargins(0, 0, 0, 0)

self.formLayout_3.setObjectName("formLayout_3")

self.lbl_tst_vectors = QtWidgets.QLabel(self.formLayoutWidget_3)

self.lbl_tst_vectors.setObjectName("lbl_tst_vectors")

self.formLayout_3.setWidget(0, QtWidgets.QFormLayout.LabelRole,
self.lbl_tst_vectors)

self.txt_n_testvectors = QtWidgets.QLineEdit(self.formLayoutWidget_3)

self.txt_n_testvectors.setObjectName("txt_n_testvectors")

self.formLayout_3.setWidget(0, QtWidgets.QFormLayout.FieldRole,
self.txt_n_testvectors)

self.lbl_verif_type = QtWidgets.QLabel(self.formLayoutWidget_3)

self.lbl_verif_type.setObjectName("lbl_verif_type")

self.formLayout_3.setWidget(2, QtWidgets.QFormLayout.LabelRole,
self.lbl_verif_type)

self.cmb_verif_type = QtWidgets.QComboBox(self.formLayoutWidget_3)

self.cmb_verif_type.setObjectName("cmb_verif_type")

self.cmb_verif_type.addItem("")

self.cmb_verif_type.addItem("")

self.cmb_verif_type.addItem("")

self.formLayout_3.setWidget(2, QtWidgets.QFormLayout.FieldRole,
self.cmb_verif_type)

spacerItem1 = QtWidgets.QSpacerItem(20, 40, QtWidgets.QSizePolicy.Minimum,
QtWidgets.QSizePolicy.Expanding)

```

```

self.formLayout_3.setItem(3, QtWidgets.QFormLayout.FieldRole, spacerItem1)

spacerItem2 = QtWidgets.QSpacerItem(20, 40, QtWidgets.QSizePolicy.Minimum,
QtWidgets.QSizePolicy.Expanding)

self.formLayout_3.setItem(1, QtWidgets.QFormLayout.FieldRole, spacerItem2)

self.verticalLayoutWidget = QtWidgets.QWidget(self.gb_constraints)

self.verticalLayoutWidget.setGeometry(QtCore.QRect(399, 290, 191, 80))

self.verticalLayoutWidget.setObjectName("verticalLayoutWidget")

self.verticalLayout_2 = QtWidgets.QVBoxLayout(self.verticalLayoutWidget)

self.verticalLayout_2.setContentsMargins(0, 0, 0, 0)

self.verticalLayout_2.setObjectName("verticalLayout_2")


self.btn_gen_testvector_file = QtWidgets.QPushButton(self.verticalLayoutWidget)

self.btn_gen_testvector_file.setObjectName("btn_gen_testvector_file")

self.btn_gen_testvector_file.setText("Generate TVF")

self.verticalLayout_2.addWidget(self.btn_gen_testvector_file)


self.btn_verify = QtWidgets.QPushButton(self.verticalLayoutWidget)

self.btn_verify.setObjectName("btn_verify")

self.verticalLayout_2.addWidget(self.btn_verify)


spacerItem3 = QtWidgets.QSpacerItem(20, 40, QtWidgets.QSizePolicy.Minimum,
QtWidgets.QSizePolicy.Expanding)

self.verticalLayout_2.addItem(spacerItem3)

```



```

self.btn_report = QtWidgets.QPushButton(self.verticalLayoutWidget)

self.btn_report.setObjectName("btn_report")

self.verticalLayout_2.addWidget(self.btn_report)

spacerItem4 = QtWidgets.QSpacerItem(20, 40, QtWidgets.QSizePolicy.Minimum,
QtWidgets.QSizePolicy.Expanding)

self.verticalLayout_2.addItem(spacerItem4)

self.tableWidget.setSelectionMode(QtWidgets.QAbstractItemView.SingleSelection)

self.tableWidget.setSelectionBehavior(QtWidgets.QAbstractItemView.SelectRows)

self.retranslateUi(Dialog)

QtCore.QMetaObject.connectSlotsByName(Dialog)


# auto generated

def retranslateUi(self, Dialog):

    _translate = QtCore.QCoreApplication.translate

    Dialog.setWindowTitle(_translate("Dialog", "Dialog"))

    self.gb_output.setTitle(_translate("Dialog", "Output"))

    self.gb_input.setTitle(_translate("Dialog", "Verilog Modules"))

    self.btn_parse.setText(_translate("Dialog", "Parse"))

    self.btn_browse_verilog.setText(_translate("Dialog", "..."))

    self.lbl_verilog_top_module.setText(_translate("Dialog", "Verilog Top Module"))

    self.lbl_golden_file.setText(_translate("Dialog", "Golden Rule File"))

    self.btn_browse_golden.setText(_translate("Dialog", "..."))

    self.gb_constraints.setTitle(_translate("Dialog", "Constraints"))

```

```

item = self.tableWidget.horizontalHeaderItem(0)

item.setText(_translate("Dialog", "Port"))

item = self.tableWidget.horizontalHeaderItem(1)

item.setText(_translate("Dialog", "Size"))

item = self.tableWidget.horizontalHeaderItem(2)

item.setText(_translate("Dialog", "Constraints"))

self.btn_con_clk.setText(_translate("Dialog", "Set as CLK"))

self.btn_con_rst_high.setText(_translate("Dialog", "Set as Active High Rst"))

self.btn_con_rst_low.setText(_translate("Dialog", "Set as Active Low Rst"))

self.btn_con_en.setText(_translate("Dialog", "Set as Enable signal"))

self.btn_con_low.setText(_translate("Dialog", "Constrain to low"))

self.btn_con_high.setText(_translate("Dialog", "Constraint to high"))

self.btn_con_val.setText(_translate("Dialog", "Constraint to value"))

self.lbl_tst_vectors.setText(_translate("Dialog", "Number of Testvectors"))

self.lbl_verif_type.setText(_translate("Dialog", "Verification Type"))

self.cmb_verif_type.setItemText(0, _translate("Dialog", ""))

self.cmb_verif_type.setItemText(1, _translate("Dialog", "Partial"))

self.cmb_verif_type.setItemText(2, _translate("Dialog", "Full"))

self.btn_verify.setText(_translate("Dialog", "Verify"))

self.btn_report.setText(_translate("Dialog", "Report"))


# setup core logic instance

def setupCore(self):

```

```

# logging options

logman.LOG_DIRECTION="qt"

logman.LOG_QWIDGET = self.txt_output


self.report=None


# setup connections and events between the GUI slots (in case you didn't understand
this, you need to read about PyQt5, no other way)

self.setupConnections()


# the core objects

self.testbench_gen = testbench_gen()

self.testster = testster()


# conditions that should be satisfied from GUI user to proceed to testing

self.ready_to_work=False

self.verilog_parsed=False

self.golden_parsed=False

self.TVF_ready=False

self.verifcation_known=False

self.results_ready=False


# threading the core

```

```

self.tester_thread = QtCore.QThread(self)

# inter-thread signal-slot

self.start_regression.connect(self.tester.run_regression)

self.tester.tester_signal_log.connect(self.fcn_log_from_thread)

self.tester.tester_signal_rpvt.connect(self.fcn_report_ready)


self.tester.moveToThread(self.tester_thread)

# starting the core object into separate thread (to not hang the gui)

self.tester_thread.start()


# setup connections from gui to slots, go study Qt5 and PyQt5 to understand this, no
other way

def setupConnections(self):

    self.btn_parse.clicked.connect(self.fcn_parse_verilog)

    self.btn_browse_verilog.clicked.connect(self.fcn_browse_verilog)

    self.btn_browse_golden.clicked.connect(self.fcn_browse_goldenrule)

    self.btn_verify.clicked.connect(self.fcn_verify)

    self.btn_report.clicked.connect(self.fcn_report)

    self.btn_gen_testvector_file.clicked.connect(self.fcn_generate_TVF_file)

```

```

self.cmb_verif_type.currentTextChanged.connect(self.fcn_verification_type_changed)

        self.btn_con_clk.clicked.connect    (partial(self.fcn_set_constraint,constr.con_clk))
        self.btn_con_en.clicked.connect    (partial(self.fcn_set_constraint,constr.con_en))
        self.btn_con_high.clicked.connect
        (partial(self.fcn_set_constraint,constr.con_high))
        self.btn_con_low.clicked.connect
        (partial(self.fcn_set_constraint,constr.con_low))
        self.btn_con_val.clicked.connect    (partial(self.fcn_set_constraint,constr.con_val))
        self.btn_con_rst_high.clicked.connect
        (partial(self.fcn_set_constraint,constr.con_rst))
        self.btn_con_rst_low.clicked.connect
        (partial(self.fcn_set_constraint,constr.con_rstn))
        self.btn_con_none.clicked.connect
        (partial(self.fcn_set_constraint,constr.con_none))

    # connections

    # browse button to browse verilog top module

    @QtCore.pyqtSlot()
    def fcn_browse_verilog(self):

        fileName, _ = QtWidgets.QFileDialog.getOpenFileName(self.dialog, 'Single File',
        '*,v')

        if len(fileName)<0:

```

```

        self.verilog_parsed=False

    return

    self.txt_verilog_path.setText( fileName)

# logging

@QtCore.pyqtSlot(logman.loglevel,str)

def fcn_log_from_thread(self,lv1,msg):

    logman.logman.log(lvl,msg)

# brosse button to browse golden rule file

@QtCore.pyqtSlot()

def fcn_browse_goldenrule(self):

    try:

        self.golden_parsed=False

        self.golden_parsed=True

        fileName, _ = QtWidgets.QFileDialog.getOpenFileName(self.dialog, 'Single File'

, '*.txt')

        if len(fileName)<0:

            self.golden_parsed=False

            return

        self.txt_goldeb_path.setText( fileName)

        self.golden_file_path= fileName

```

```

except Exception as e:

    pass

# reading the verilog after we got its path, then using the core object to parse it to get
its inputs

# then, display them into the gui

@QtCore.pyqtSlot()

def fcn_parse_verilog(self):

    try:

        self.verilog_parsed=False

        # reading verilog path from user

        verilog_path=self.txt_verilog_path.text()

        if len(verilog_path)==0:

            logman.logman.log(logman.loglevel.err,msg_invalid_verilog_path)

            logman.logman.log_msgbox(logman.loglevel.err,msg_invalid_verilog_path)

            return

        logman.logman.log(logman.loglevel.debug,"parsing verilog file:

        {}".format(verilog_path))

        # asking core object to parse it

        inputs = self.testbench_gen.parse_verilog(module_path=verilog_path)

        # populating the table

```

```

while self.tableWidget.rowCount() > 0:

    self.tableWidget.removeRow(0)

for key,val in inputs.items():

    newIndex = self.tableWidget.rowCount()

    self.tableWidget.setRowCount(newIndex + 1)


    self.tableWidget.setItem(newIndex,0,QtWidgets.QTableWidgetItem(key))


self.tableWidget.setItem(newIndex,1,QtWidgets.QTableWidgetItem(str(val['size'])))


self.tableWidget.setItem(newIndex,2,QtWidgets.QTableWidgetItem(val['constr'].value))


    self.verilog_path=verilog_path

    self.verilog_parsed=True

except Exception as e:

    pass


# imposing a constraint from GUI.

# this handles constraints buttons in the gui and asks the core object to impose the
constraint for us

# then giving the port a color according to the type of constraints.


@QtCore.pyqtSlot(constr)

```



```

def fcn_set_constraint(self,constraint):

    for index in self.tableWidget.selectionModel().selectedRows():

        i= index.row()

        self.tableWidget.item(i,2).setText(constraint.value)


    port=self.tableWidget.item(i,0).text()


    if constraint==constr.con_val:

        val=self.txt_con_val.text().replace("_","")

    else:

        val=""

    done,msg=self.testbench_gen.impose_constraint(port,constraint,val)

    if not done:

        logman.logman.log(logman.loglevel.err,msg)

        logman.logman.log_msgbox(logman.loglevel.err,msg)

    else:

        logman.logman.log(logman.loglevel.debug,msg)


    #some fancy colors

    if constraint==constr.con_clk:

        self.tableWidget.item(i,2).setBackground(QtGui.QColor("#EC5D55"))

    elif constraint==constr.con_high:

        self.tableWidget.item(i,2).setBackground(QtGui.QColor("#C2E358"))

```

```

elif constraint==constr.con_low:

    self.tableWidget.item(i,2).setBackground(QtGui.QColor("#36AB9D"))

elif constraint==constr.con_en:

    self.tableWidget.item(i,2).setBackground(QtGui.QColor("#B1F9E1"))

elif constraint in [constr.con_rst,constr.con_rstn]:

    self.tableWidget.item(i,2).setBackground(QtGui.QColor("#676694"))

else:

    self.tableWidget.item(i,2).setBackground(QtGui.QColor("#FFF7F0"))


# generate the TVF button.

@QtCore.pyqtSlot()

def fcn_generate_TVF_file(self):

    try:

        self.TVF_ready=False

        if self.golden_parsed==False:

            logman.logman.log(logman.loglevel.err,msg_golden_not_parsed)

            return


        golden_path=self.txt_goldeb_path.text()

        done=self.testbench_gen.gen_testvectors(golden_file=golden_path)

        if done:

            logman.logman.log (logman.loglevel.debug,msg_TVF_file_generated)

```

```

        self.TVF_ready=True

    except Exception as e:

        pass

# verify button

@QtCore.pyqtSlot()

def fcn_verify(self):

    if self.verilog_parsed and self.golden_parsed and self.TVF_ready and
self.verification_known:

        self.results_ready=False

        self.start_regression.emit(self.verilog_path)

    else:

        logman.logman.log(logman.loglevel.err,msg_fill_req_data)

# this is executed when the tester informs us that report is ready

@QtCore.pyqtSlot(dict)

def fcn_report_ready(self,report_dict):

    self.report = report_dict

    self.results_ready=True

    msg = QtWidgets.QMessageBox.about(self.dialog, "Finished", "Regression has
finished successfully.")

# this when we try to open the report. it opens it in the browser (as it is html report)

```

```

@QtCore.pyqtSlot()

def fcn_report(self):

    with open ('test_report.rpt','r') as rpt:

        jsonfile = json.load(rpt)

    html_table= json2table.convert(jsonfile,

        build_direction="LEFT_TO_RIGHT",

        table_attributes={ "style": "width:50%",

            "class" : "table table-striped"

        })

    with open ('report_html.html','w') as html_rpt:

        html_rpt.write(html_table)

    os.system('report_html.html')

# when we select verification type, this affects the conFigures inside the core.

# So, we are making this conFigures according to the type before informing the core
with it

@QtCore.pyqtSlot(str)

def fcn_verification_type_changed(self,verificatio_type):

    self.verification_known=False

    if verificatio_type.lower() == "full":

        self.testbench_gen.set_conFigures({

            "verif_type": verification_types.verif_full,

```

```

        'verif_n':0
    })

elif verificatio_type.lower() == "partial":

    if len(self.txt_n_testvectors.text()) ==0:

        logman.logman.log(logman.loglevel.err,msg_n_testvectors_empty)

    try:

        n_tvs = int(self.txt_n_testvectors.text())

        self.testbench_gen.set_conFigures({

            "verif_type": verification_types.verif_partial,

            'verif_n':n_tvs

        })

        self.verification_known=True

    except Exception as e:

        logman.logman.log(logman.loglevel.err,msg_n_testvectors_lessthanone)

else:

    logman.logman.log(logman.loglevel.err,msg_select_valid_option)

```

Running the above main class, most of this is auto generated

```

if __name__ == "__main__":

    import sys

    app = QtWidgets.QApplication(sys.argv)

```

```

Dialog = QtWidgets.QDialog()

ui = Ui_Dialog()

ui.setupUi(Dialog)

ui.setupCore()

Dialog.setWindowTitle("Verifog")

Dialog.setWindowIcon(QtGui.QIcon('verifog.ico'))

Dialog.show()

sys.exit(app.exec_())

```

5. Verifog2.ui: Autogenerated file.

```

<?xml version="1.0" encoding="UTF-8"?>

<ui version="4.0">

<class>Dialog</class>

<widget class="QDialog" name="Dialog">

<property name="geometry">

<rect>

<x>0</x>

<y>0</y>

<width>619</width>

<height>728</height>

</rect>

</property>

<property name="windowTitle">

<string>Dialog</string>

```

```

</property>

<widget class="QGroupBox" name="gb_output">

  <property name="geometry">

    <rect>

      <x>10</x>

      <y>490</y>

      <width>601</width>

      <height>231</height>

    </rect>

  </property>

  <property name="title">

    <string>Output</string>

  </property>

  <widget class="QTextEdit" name="txt_output">

    <property name="geometry">

      <rect>

        <x>10</x>

        <y>20</y>

        <width>581</width>

        <height>201</height>

      </rect>

    </property>

  </widget>

```

```

</widget>

<widget class="QGroupBox" name="gb_input">

  <property name="geometry">

    <rect>

      <x>10</x>

      <y>10</y>

      <width>601</width>

      <height>101</height>

    </rect>

  </property>

  <property name="title">

    <string>Verilog Modules</string>

  </property>

  <widget class="QWidget" name="gridLayoutWidget">

    <property name="geometry">

      <rect>

        <x>10</x>

        <y>20</y>

        <width>581</width>

        <height>111</height>

      </rect>

    </property>

    <layout class="QGridLayout" name="gridLayout">

```



```

<item row="0" column="3">

  <widget class="QPushButton" name="btn_parse">

    <property name="sizePolicy">

      <sizepolicy hsize="Fixed" vsize="Fixed">

        <horstretch>0</horstretch>

        <verstretch>0</verstretch>

      </sizepolicy>

    </property>

    <property name="text">

      <string>Parse</string>

    </property>

  </widget>

</item>

<item row="0" column="2">

  <widget class="QPushButton" name="btn_browse_verilog">

    <property name="sizePolicy">

      <sizepolicy hsize="Fixed" vsize="Fixed">

        <horstretch>0</horstretch>

        <verstretch>0</verstretch>

      </sizepolicy>

    </property>

    <property name="text">

      <string>...</string>

```

```

</property>

</widget>

</item>

<item row="1" column="1">

  <widget class="QLineEdit" name="txt_goldeb_path"/>

</item>

<item row="0" column="0">

  <widget class="QLabel" name="lbl_verilog_top_module">

    <property name="text">

      <string>Verilog Top Module</string>

    </property>

  </widget>

</item>

<item row="1" column="0">

  <widget class="QLabel" name="lbl_golden_file">

    <property name="text">

      <string>Golden Rule File</string>

    </property>

  </widget>

</item>

<item row="1" column="2">

  <widget class="QPushButton" name="btn_browse_golden">

    <property name="sizePolicy">

```

```

<sizepolicy hsize="Fixed" vsize="Fixed">

<horstretch>0</horstretch>

<verstretch>0</verstretch>

</sizepolicy>

</property>

<property name="text">

<string>...</string>

</property>

</widget>

</item>

<item row="0" column="1">

<widget class="QLineEdit" name="txt_verilog_path"/>

</item>

<item row="3" column="2">

<spacer name="verticalSpacer_2">

<property name="orientation">

<enum>Qt::Vertical</enum>

</property>

<property name="sizeHint" stdset="0">

<size>

<width>20</width>

<height>40</height>

</size>

```

```

    </property>

    </spacer>

</item>

</layout>

</widget>

</widget>

<widget class="QGroupBox" name="gb_constraints">

    <property name="geometry">

        <rect>

            <x>10</x>

            <y>110</y>

            <width>601</width>

            <height>381</height>

        </rect>

    </property>

    <property name="title">

        <string>Constraints</string>

    </property>

<widget class="QWidget" name="horizontalLayoutWidget_3">

    <property name="geometry">

        <rect>

            <x>10</x>

            <y>20</y>

```

```

<width>579</width>

<height>261</height>

</rect>

</property>

<layout class="QHBoxLayout" name="horizontalLayout_3">

<item>

<widget class="QTableWidget" name="tableWidget">

<property name="sizePolicy">

<sizepolicy hsize="Expanding" vsize="Expanding">

<horstretch>0</horstretch>

<verstretch>0</verstretch>

</sizepolicy>

</property>

<property name="frameShape">

<enum>QFrame::StyledPanel</enum>

</property>

<property name="frameShadow">

<enum>QFrame::Plain</enum>

</property>

<property name="sizeAdjustPolicy">

<enum>QAbstractScrollArea::AdjustToContents</enum>

</property>

<property name="alternatingRowColors">

```

```

    <bool>true</bool>

  </property>

  <property name="selectionMode">

    <enum>QAbstractItemView::ContiguousSelection</enum>

  </property>

  <property name="showGrid">

    <bool>true</bool>

  </property>

  <property name="rowCount">

    <number>0</number>

  </property>

  <attribute name="horizontalHeaderCascadingSectionResizes">

    <bool>false</bool>

  </attribute>

  <attribute name="verticalHeaderVisible">

    <bool>false</bool>

  </attribute>

  <attribute name="verticalHeaderCascadingSectionResizes">

    <bool>true</bool>

  </attribute>

  <column>

    <property name="text">

      <string>Port</string>

```

```

    </property>
</column>
<column>
    <property name="text">
        <string>Size</string>
    </property>
</column>
<column>
    <property name="text">
        <string>Constraints</string>
    </property>
</column>
</widget>
</item>
<item>
    <layout class="QVBoxLayout" name="verticalLayout_4">
        <item>
            <widget class="QPushButton" name="btn_con_clk">
                <property name="text">
                    <string>Set as CLK</string>
                </property>
            </widget>
        </item>
    </layout>
</item>

```

```

<item>

<widget class="QPushButton" name="btn_con_rst_high">

  <property name="sizePolicy">

    <sizepolicy hsize="Expanding" vsize="Fixed">

      <horstretch>0</horstretch>

      <verstretch>0</verstretch>

    </sizepolicy>

  </property>

  <property name="text">

    <string>Set as Active High Rst</string>

  </property>

</widget>

</item>

<item>

<widget class="QPushButton" name="btn_con_rst_low">

  <property name="text">

    <string>Set as Active Low Rst</string>

  </property>

</widget>

</item>

<item>

<widget class="QPushButton" name="btn_con_en">

  <property name="text">

```



```

    <string>Set as Enable signal</string>

  </property>

</widget>

</item>

<item>

  <widget class="QPushButton" name="btn_con_low">

    <property name="text">

      <string>Constrain to low</string>

    </property>

  </widget>

</item>

<item>

  <widget class="QPushButton" name="btn_con_high">

    <property name="text">

      <string>Constraint to high</string>

    </property>

  </widget>

</item>

<item>

  <widget class="QPushButton" name="btn_con_val">

    <property name="text">

      <string>Constraint to value</string>

    </property>

```

```

    </widget>

  </item>

</layout>

</item>

</layout>

</widget>

<widget class="QWidget" name="formLayoutWidget_3">
  <property name="geometry">
    <rect>
      <x>10</x>
      <y>290</y>
      <width>371</width>
      <height>83</height>
    </rect>
  </property>
  <layout class="QFormLayout" name="formLayout_3">
    <item row="0" column="0">
      <widget class="QLabel" name="lbl_tst_vectors">
        <property name="text">
          <string>Number of Testvectors</string>
        </property>
      </widget>
    </item>

```

```

<item row="0" column="1">

  <widget class="QLineEdit" name="txt_n_testvectors"/>

</item>

<item row="2" column="0">

  <widget class="QLabel" name="lbl_verif_type">

    <property name="text">

      <string>Verification Type</string>

    </property>

  </widget>

</item>

<item row="2" column="1">

  <widget class="QComboBox" name="cmb_verif_type">

    <item>

      <property name="text">

        <string>Partial</string>

      </property>

    </item>

    <item>

      <property name="text">

        <string>Full</string>

      </property>

    </item>

  </widget>

```

```

</item>

<item row="3" column="1">

  <spacer name="verticalSpacer_3">

    <property name="orientation">

      <enum>Qt::Vertical</enum>

    </property>

    <property name="sizeHint" stdset="0">

      <size>

        <width>20</width>

        <height>40</height>

      </size>

    </property>

  </spacer>

</item>

<item row="1" column="1">

  <spacer name="verticalSpacer_4">

    <property name="orientation">

      <enum>Qt::Vertical</enum>

    </property>

    <property name="sizeHint" stdset="0">

      <size>

        <width>20</width>

        <height>40</height>

```

```

    </size>

    </property>

    </spacer>

    </item>

    </layout>

    </widget>

    <widget class="QWidget" name="verticalLayoutWidget">

        <property name="geometry">

            <rect>

                <x>399</x>

                <y>290</y>

                <width>191</width>

                <height>80</height>

            </rect>

        </property>

        <layout class="QVBoxLayout" name="verticalLayout_2">

            <item>

                <widget class="QPushButton" name="btn_verify">

                    <property name="text">

                        <string>Verify</string>

                    </property>

                </widget>

            </item>

```

```

<item>

<spacer name="verticalSpacer_5">

  <property name="orientation">

    <enum>Qt::Vertical</enum>

  </property>

  <property name="sizeHint" stdset="0">

    <size>

      <width>20</width>

      <height>40</height>

    </size>

  </property>

</spacer>

</item>

<item>

<widget class="QPushButton" name="btn_report">

  <property name="text">

    <string>Report</string>

  </property>

</widget>

</item>

<item>

<spacer name="verticalSpacer">

  <property name="orientation">

```

```

        <enum>Qt::Vertical</enum>

    </property>

    <property name="sizeHint" stdset="0">

        <size>

            <width>20</width>

            <height>40</height>

        </size>

    </property>

</spacer>

</item>

</layout>

</widget>

</widget>

</widget>

<resources/>

<connections/>

</ui>

```

6. Logman.py

```

import enum

from datetime import datetime

```

```

# This file contains logging functions.

# logging direction (to console or to gui)

LOG_DIRECTION= "console"

# the widget in gui used for writing logs if direction != console

LOG_QWIDGET=None


# log levels

class loglevel(enum.Enum):

    info = '[INFO]'

    err = '[ERROR]'

    debug = '[DEBUG]'


# Logging class

class logman:

    @staticmethod

    def log(log_level,msg):

        # formating the log message

        msg_formated='{LVL} [{DT}] {MSG}'.format(LVL=log_level.value,

                                                DT=datetime.now().strftime("%m-%d-%Y, %H:%M:%S"),

                                                MSG=msg)

        # log to console if needed

        if LOG_DIRECTION== "console":

```



```

        print(msg_formatted)

    else:

        # log to gui if needed

        if LOG_QWIDGET is None:

            pass

        LOG_QWIDGET.append(msg_formatted)

# not implemented and not used.

@staticmethod

def log_msgbox(log_level,msg):

    pass

```

7. Strings.py

this file groups all messages in one place for more readability and organization

msg_invalid_verilog_path= "Invalid verilog path"

msg_constraint_invalid_size = "Size of Value is not equal port size"

msg_n_testvectors_empty = "Number of testvectors cannot be empty"

msg_n_testvectors_lessthanone = "Number of testvectos should be greater than zero"

msg_select_valid_option = "Select a valid option for verification type"

msg_golden_not_parsed = "Please select the golden file"

msg_golden_not_all_ops = "At least one case in the golden file is not fully determined"

msg_golden_not_all_ips = "At least one input not determined"

msg_fill_req_data = "Please fill all required data to start the regression"

```
msg_TVF_file_generated = "TestVector file (TVF) file has been successfully generated "
```

```
msg_writing_makefile = "Writing make file"
```

```
msg_done_writing_makefile = "Make file has been successfully written"
```

```
msg_regression_running = "Running testing regression. Please be patient"
```

```
msg_regression_done    = "Testing regression has ended successfully."
```

```
msg_regression_err     = "Testing regression has ended with errors."
```

8. Example.adder.v

```
module example_adder
```

```
(
```

```
input clk,rstn,en,
```

```
input[7:0] port_a,
```

```
input[7:0] port_b,
```

```
output[7:0] added
```

```
);
```

```
initial
```

```
begin
```

```
    $dumpfile("test.vcd");
```

```
    $dumpvars;
```

```
end
```

```
assign added = port_a+port_b;
```

```
endmodule
```

9. Leading_one.v

```
module leading_one
(
    input [7:0]    in_i,
    output [2:0] first_one_o,
    output        no_ones_o
);

    localparam NUM_LEVELS = $clog2(8);

    wire [8-1:0][NUM_LEVELS-1:0]    index_lut;
    wire [2**NUM_LEVELS-1:0]        sel_nodes;
    wire [2**NUM_LEVELS-1:0][NUM_LEVELS-1:0] index_nodes;

    wire [7:0] in_tmp;

    genvar i;

    generate
    for (i = 0; i < 8; i=i+1) begin: GEN0
        assign in_tmp[i] = in_i[7-i] ;
    end

    endgenerate

    genvar j;
```

```

        generate
    for (j = 0; j < 8; j=j+1) begin: GEN1
        assign index_lut[j] = j;
    end

    endgenerate

    genvar level,k,l;

    generate
    for (level = 0; level < NUM_LEVELS; level=level+1) begin

        if (level < NUM_LEVELS-1) begin
            for (l = 0; l < 2**level; l=l+1) begin:GEN2
                assign sel_nodes[2**level-l+1] = sel_nodes[2**(level+1)-l+1*2] |
sel_nodes[2**(level+1)-l+1*2+1];

                assign index_nodes[2**level-l+1] = (sel_nodes[2**(level+1)-l+1*2] == 1'b1) ?
                    index_nodes[2**(level+1)-l+1*2] : index_nodes[2**(level+1)-l+1*2+1];
            end
        end

    end

    if (level == NUM_LEVELS-1) begin
        for (k = 0; k < 2**level; k=k+1) begin:GEN3

            // if two successive indices are still in the vector...

            if (k * 2 < 8-1) begin

```

```

        assign sel_nodes[2**level-1+k] = in_tmp[k*2] | in_tmp[k*2+1];

        assign index_nodes[2**level-1+k] = (in_tmp[k*2] == 1'b1) ? index_lut[k*2]
: index_lut[k*2+1];

    end

    // if only the first index is still in the vector...

    if (k * 2 == 8-1) begin

        assign sel_nodes[2**level-1+k] = in_tmp[k*2];

        assign index_nodes[2**level-1+k] = index_lut[k*2];

    end

    // if index is out of range

    if (k * 2 > 8-1) begin

        assign sel_nodes[2**level-1+k] = 1'b0;

        assign index_nodes[2**level-1+k] = 'd0;

    end

end

end

end

endgenerate

assign first_one_o = NUM_LEVELS > 0 ? index_nodes[0] : 'd0;

assign no_ones_o = NUM_LEVELS > 0 ? ~sel_nodes[0] : 1'b1;

endmodule

```

REFERENCES

- [1] “VLSI Design - Digital System - Tutorialspoint.”
https://www.tutorialspoint.com/vlsi_design/vlsi_design_digital_system.htm
(accessed Jun. 28, 2021).
- [2] M. K. Ganai and A. Gupta, Eds., “Design Verification Challenges,” in *SAT-Based Scalable Formal Verification Solutions*, Boston, MA: Springer US, 2007, pp. 1–16.
doi: 10.1007/978-0-387-69167-1_1.
- [3] “SoC Interconnect Verification Challenge,” *Design And Reuse*. <https://www.design-reuse.com/articles/31993/soc-interconnect-verification-challenge.html> (accessed Jun. 18, 2021).
- [4] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L.-C. Wang, “Challenges and Trends in Modern SoC Design Verification,” *IEEE Des. Test*, vol. 34, no. 5, pp. 7–22, Oct. 2017, doi: 10.1109/MDAT.2017.2735383.
- [5] E. Staff, “Dealing with the challenges of integrating hardware and software verification,” *Embedded.com*, Jan. 04, 2008. <https://www.embedded.com/dealing-with-the-challenges-of-integrating-hardware-and-software-verification/> (accessed Apr. 07, 2021).
- [6] G. S. Chandar and S. Vaideeswaran, “Addressing verification bottlenecks of fully synthesized processor cores using equivalence checkers,” in *Proceedings of the ASP-DAC 2001. Asia and South Pacific Design Automation Conference 2001 (Cat. No.01EX455)*, Feb. 2001, pp. 175–180. doi: 10.1109/ASPDAC.2001.913300.

- [7] Z. B. H. Amor, L. Pierre, and D. Borriane, “System-on-chip verification: TLM-to-RTL assertions transformation,” in *2014 10th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*, Jun. 2014, pp. 1–4. doi: 10.1109/PRIME.2014.6872713.
- [8] A. Hany, A. Ismail, A. Kamal, and M. Badran, “Approach for a unified functional verification flow,” in *2013 Saudi International Electronics, Communications and Photonics Conference*, Apr. 2013, pp. 1–6. doi: 10.1109/SIEPC.2013.6550753.
- [9] R. Drechsler, “Synthesizing checkers for on-line verification of System-on-Chip designs,” in *Proceedings of the 2003 International Symposium on Circuits and Systems, 2003. ISCAS '03.*, May 2003, vol. 4, p. IV–IV. doi: 10.1109/ISCAS.2003.1206281.
- [10] E. Zhang and E. Yogev, “Functional verification with completely self-checking tests,” in *Proceedings of Meeting on Verilog HDL (IVC/VIUF'97)*, Mar. 1997, pp. 2–9. doi: 10.1109/IVC.1997.588525.
- [11] EETimes, “EETimes - How transaction-based verification works,” *EETimes*, Mar. 22, 2002. <https://www.eetimes.com/how-transaction-based-verification-works/> (accessed Jul. 05, 2021).
- [12] R. Python, “Qt Designer and Python: Build Your GUI Applications Faster – Real Python.” <https://realpython.com/qt-designer-python/> (accessed Apr. 07, 2021).
- [13] “PyQt - Using Qt Designer - Tutorialspoint.” https://www.tutorialspoint.com/pyqt/pyqt_using_qt_designer.htm (accessed Jun. 14, 2021).

- [14] L. P. R. R. time 12:52 C. applications with Q. Designer, “Build GUI layouts with Qt Designer for PyQt5 apps,” *Martin Fitzpatrick*, Jan. 09, 2020.
<https://www.mfitzp.com/tutorials/qt-designer-gui-layout/> (accessed Jun. 14, 2021).
- [15] R. C. Limited, *PyQt5: Python bindings for the Qt cross platform application toolkit*. Accessed: Apr. 07, 2021. [Online]. Available:
<https://www.riverbankcomputing.com/software/pyqt/>
- [16] “PyQt5 tutorial 2021: Create a GUI with Python and Qt.” <https://build-system.fman.io/pyqt5-tutorial> (accessed Apr. 07, 2021).
- [17] “Welcome to cocotb’s documentation! — cocotb 1.6.0.dev0+g0872e05c.d20210320 documentation.” <https://docs.cocotb.org/en/stable/index.html> (accessed Apr. 07, 2021).
- [18] “Using Cocotb for design re-use and randomized testing in Riviera-PRO - Application Notes - Documentation - Resources - Support - Aldec.”
<https://www.aldec.com/en/support/resources/documentation/articles/1905> (accessed Jun. 14, 2021).
- [19] “Python For Beginners,” *Python.org*. <https://www.python.org/about/gettingstarted/> (accessed Apr. 07, 2021).
- [20] “What is Python Used For?,” *Treehouse Blog*, Jul. 08, 2014.
<https://blog.teamtreehouse.com/what-is-python> (accessed Apr. 07, 2021).
- [21] J. I. Villar, J. Juan, M. J. Bellido, J. Viejo, D. Guerrero, and J. Decaluwe, “Python as a hardware description language: A case study,” in *2011 VII Southern Conference on Programmable Logic (SPL)*, Apr. 2011, pp. 117–122. doi: 10.1109/SPL.2011.5782635.

- [22] M. Solutions, “Python: 7 Important Reasons Why You Should Use Python,” *Medium*, Oct. 03, 2017. <https://medium.com/@mindfiresolutions.usa/python-7-important-reasons-why-you-should-use-python-5801a98a0d0b> (accessed Apr. 07, 2021).
- [23] R. C. Limited, *PyQt5: Python bindings for the Qt cross platform application toolkit*. Accessed: Jun. 18, 2021. [Online]. Available: <https://www.riverbankcomputing.com/software/pyqt/>
- [24] “PyQt5 tutorial 2021: Create a GUI with Python and Qt.” <https://build-system.fman.io/pyqt5-tutorial> (accessed Jun. 18, 2021).
- [25] “Welcome to cocotb’s documentation! — cocotb 1.6.0.dev0+gec99a877.d20210503 documentation.” <https://docs.cocotb.org/en/stable/> (accessed Jun. 18, 2021).
- [26] P. Tiwari, R. Mitra, M. Chopra, and A. Jain, “Tutorial T4B: Formal Assertion-Based Verification in Industrial Setting,” in *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID ’07)*, Jan. 2007, pp. 7–7. doi: 10.1109/VLSID.2007.169.
- [27] “Tutorial - What is a Testbench (simulation).” <https://www.nandland.com/articles/what-is-a-testbench-fpga.html> (accessed Jun. 18, 2021).
- [28] “Testbench,” *Semiconductor Engineering*. https://semiengineering.com/knowledge_centers/eda-design/verification/testbench/ (accessed Jun. 18, 2021).

- [29] “9. Testbenches — FPGA designs with Verilog and SystemVerilog documentation.”
<https://verilogguide.readthedocs.io/en/latest/verilog/testbench.html> (accessed Jun. 18, 2021).
- [30] “Addressing the Verification Bottleneck | EE Times.”
<https://www.eetimes.com/addressing-the-verification-bottleneck/> (accessed Apr. 08, 2021).
- [31] A. MOLINA and O. Cadenas, “Functional verification: Approaches and challenges,” *Lat. Am. Appl. Res. Pesqui. Apl. Lat. Am. Investig. Apl. Latinoam.*, vol. 37, Jan. 2007.
- [32] “How to choose a verification methodology | EE Times.”
<https://www.eetimes.com/how-to-choose-a-verification-methodology/> (accessed Apr. 08, 2021).
- [33] S. Malik, “Hardware Verification: Techniques, Methodology and Solutions,” in *Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Heidelberg, 2008, pp. 1–1. doi: 10.1007/978-3-540-78800-3_1.
- [34] A. Hany, A. Ismail, A. Kamal, and M. Badran, “Approach for a unified functional verification flow,” in *2013 Saudi International Electronics, Communications and Photonics Conference*, Apr. 2013, pp. 1–6. doi: 10.1109/SIEPCPC.2013.6550753.
- [35] S. Karlapalem and S. Venugopal, “Scalable, Constrained Random Software Driven Verification,” in *2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, Dec. 2016, pp. 71–76. doi: 10.1109/MTV.2016.19.

- [36] “Working with JSON - Learn web development | MDN.”
<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON> (accessed Jun. 17, 2021).
- [37] “xml - Why json and not txt,” *Stack Overflow*.
<https://stackoverflow.com/questions/44509942/why-json-and-not-txt> (accessed Jun. 17, 2021).
- [38] K. Kunaraj and R. Seshasayanan, “Leading one detectors and leading one position detectors - An evolutionary design methodology,” *Can. J. Electr. Comput. Eng.*, vol. 36, no. 3, pp. 103–110, Summer 2013, doi: 10.1109/CJECE.2013.6704691.
- [39] D. Nandan, “AN EFFICIENT ARCHITECTURE OF LEADING ONE DETECTOR,” *Int. J. Pure Appl. Math.*, vol. 118, Feb. 2018.
- [40] “(20) SV out, Python in, do You dare? | LinkedIn.”
<https://www.linkedin.com/pulse/sv-out-python-do-you-dare-ilia-greenblat/>
(accessed Jun. 23, 2021).