2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 15-16, 2024

Comparative Study of Multiple Frameworks in Register Verification and Finding a More Efficient Solution for Early Verification Sign-off

Sougata Bhattacharjee, Samsung Semiconductor India Research (SSIR), Senior Staff Engineer, Bangalore, India,

sougata.b@samsung.com

***Abstract*—Register Verification is one of the important aspects of Verification in IP, Subsystem, and SOC-based designs. With the increasing complexity of the design and keeping time to market in mind along with the project needs, verifying huge chunks of registers within the specified time limit is an enormous challenge. The Register Abstraction Layer (RAL) is a way of automating the register verification process thereby increasing the reusability of the testbench. RAL is used to abstract or separate the tests from the register-level details. Therefore it is independent of the protocol information, address, and interface but in spite of all the advantages mentioned, there are certain challenges associated with the RAL. The motivation behind writing this paper is to address the set of challenges in UVM RAL and also to come up with unique solutions to solve the given set of problems and improve the quality and efficiency of the testbench.**

Keywords—UVM, RAL, Python, March-Bash, Chisel, Cocotb

I. INTRODUCTION

As the verification process became more stringent due to the complex nature of the chips, bringing more automation in the DV flows is the urgent need of the hour to streamline the project execution, meet time to market, and deliver bug-free chips. Among several other aspects of Verification, Register Verification also needs attention because otherwise, it will end up consuming a lot of time depending on the number of registers and the complexity of the chip.

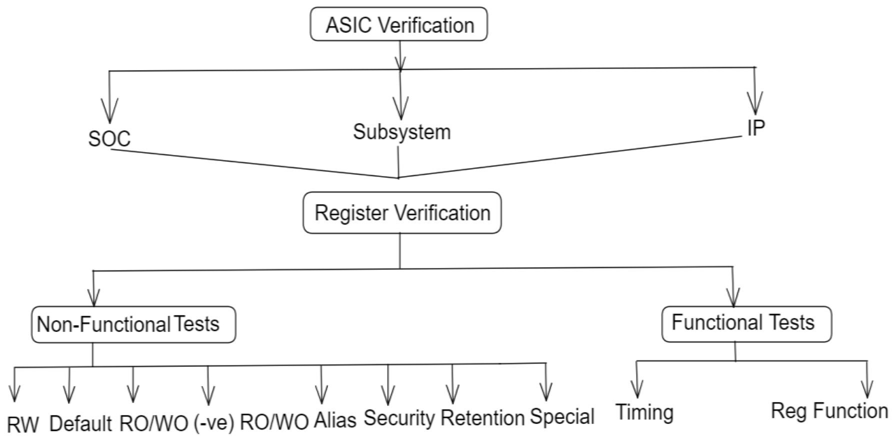


Figure 1: Register Tests Classification

Although all the above tests (mentioned in the above diagram) can be performed with the standard automation techniques provided by UVM RAL [1] infrastructure and its different inbuilt sequences, there are certain challenges associated with it which make it inefficient.

II. UVM RAL FLOW AND CHALLENGES

A. *UVM RAL Flow*

The Register model is generated with the help of a Register Assistant which contains the base address of the block, the corresponding offset of the sub-block, and the Register spec in the form of XML as input. The generated register model contains the hierarchical combination of a single register, a block of register, and the top register block which has all the information of access attribute, volatility, size, lsb, randomness, and reset value related to a particular register.

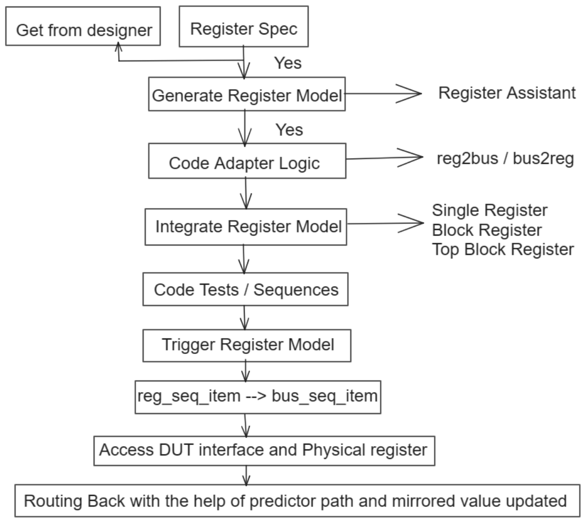


Figure 2: UVM RAL Framework / Flow

B. *UVM RAL Infrastructure*

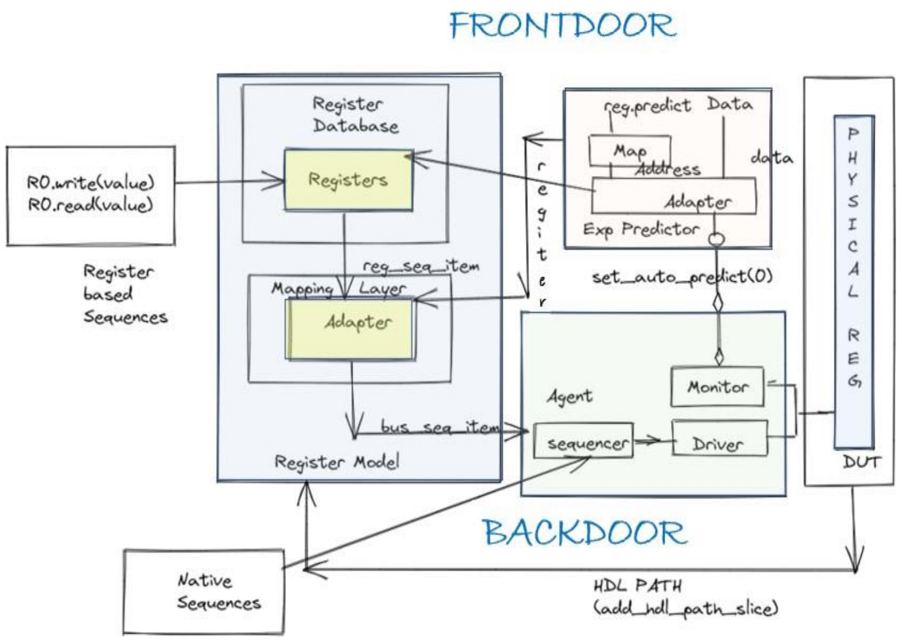


Figure 3: UVM RAL Infrastructure

The register has been called by its name instead of the address from the sequence and when it is triggered it maps the content of the register through the address map in the register model. The register model has all the information regarding the attribute and other access functionalities of the register through the register database.

It then produces the reg_sequence_item and with the help of an adapter converts it into a bus sequence item. The adapter is bidirectional and has two functionalities in the name of bus2reg and reg2bus. The transaction then reaches the Agent interface through which it will be able to access the contents of the physical register from the DUT. The response then routes back with the help of the predictor path and reaches the adapter to update the contents of the mirror value. The mirror value is sitting in the register model and contains the current state of the DUT register and is updated by the predictor after each writes and read cycle. It is important to note that the mirror value should not be out of date. Apart from the mirrored value, the desired value of a register can be written and retrieved back with the help of set() and get().

The process described above can be termed as Frontdoor. But apart from that, there is one more path through which the register contents can be accessed directly by skipping the entire process of the adapter–predictor path with the help of HDL paths by setting up the add_hdl_path_slices in the reg model, and this type of access is known as Backdoor access.

C. Challenges in UVM RAL:

- If there are more registers (For Ex: ~50k to 1L), the bit bash sequence of UVM RAL will consume more simulation time in regression due to its exhaustive execution in the form of walking 0 / 1. Due to that disk space increases and the size of the simulation logs also increases making it immensely difficult to find any error register from the log. Moreover, the bit bash sequence might stop intermittently or the test might not finish due to a timeout issue, disk space issue, or the inability of Simulation to complete.
- Unit-level or Module-level testing is not possible with the UVM RAL infrastructure.
- The verbosity and the complexity of the UVM RAL are more and hence any new Engineer from a different background may find it hard to deal with the code.
- Less Versatile and concise due to dependence on SystemVerilog and hence due to this programming style, RAL can not perform efficiently on immutable data structures.

III. SOLUTIONS TO CHALLENGES IN UVM RAL

A. Use of Custom Algorithm (Solution to Challenge – 1):

The diagram below shows the implementation of the March-Bash algorithm and it nullifies some of the disadvantages of the Bit Bash sequence as mentioned above and additionally has some extra features that make it more efficient and can be termed as Bit-Bash +- (i.e. All the good features of Bit Bash – time-consuming + Additional features)

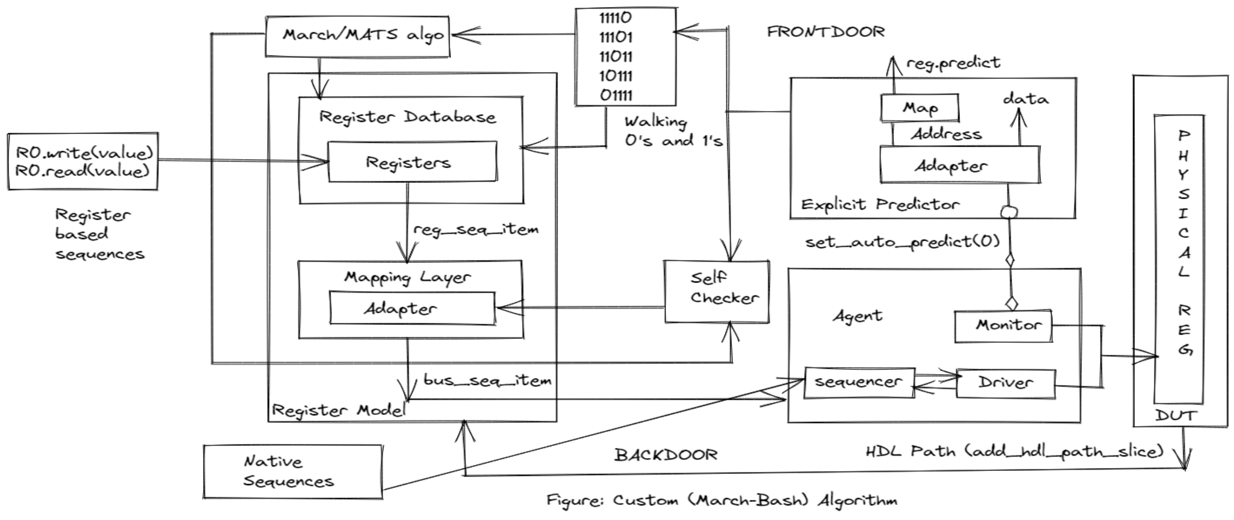


Figure 4: March-Bash Algorithm

Execution steps:

- Writing data in the register triggered in the register database in the register model.
- The register sequence item is converted to a bus sequence item with the help of the Adapter.
- Map each content of the register offset with the corresponding address map.
- Write data in the form of a reglist pattern to make sure each register within the set is covered.
- The registers are grouped in the form of clusters by categorizing them based on address and offset.
- The data is passed through the DUT interface and accesses the physical register
- Compare the written data with the read data with the help of a predictor
- The checked data is taken as input and passed on to the bus for accuracy.
- Write 0 in increasing & decreasing addressing order and Read 0 / Write1 in increasing addressing order
- Read 1 and Write 0 in decreasing addressing order and Read 0 in both increasing/decreasing addressing order
- The corresponding data is checked with predicted data & compared and if it is correct update the mirror value

Advantages:

- Each bit or field of the register is verified and checked.
- Each signal of RTL is being checked for Stuck at Fault 0 and 1
- Cost and Time-Saving in terms of the issues being addressed at Pre Silicon Verification itself.
- Resolves the problem of Transition and Addresses decoding bugs.
- Reduces the debug effort and improves the efficiency of the testbench.
- Registers can be claimed as regions in the form of clusters.

B. Use of SVUnit for Register Verification (Solution to Challenge – 2):

- Directed Test cases are easy to code as their development time is less but covers fewer scenarios while in constrained random, development time is more but covers more scenarios. SVUnit comes up as an alternative to bridge the gap between the two.

- Used for unit-level testing and can be used by both design and verification engineers and is relatively fast as used for only unit level.

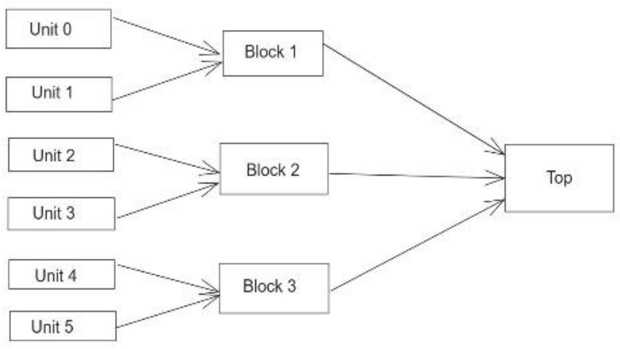


Figure 5: SVUnit Flow

SVUnit as Register Verification:

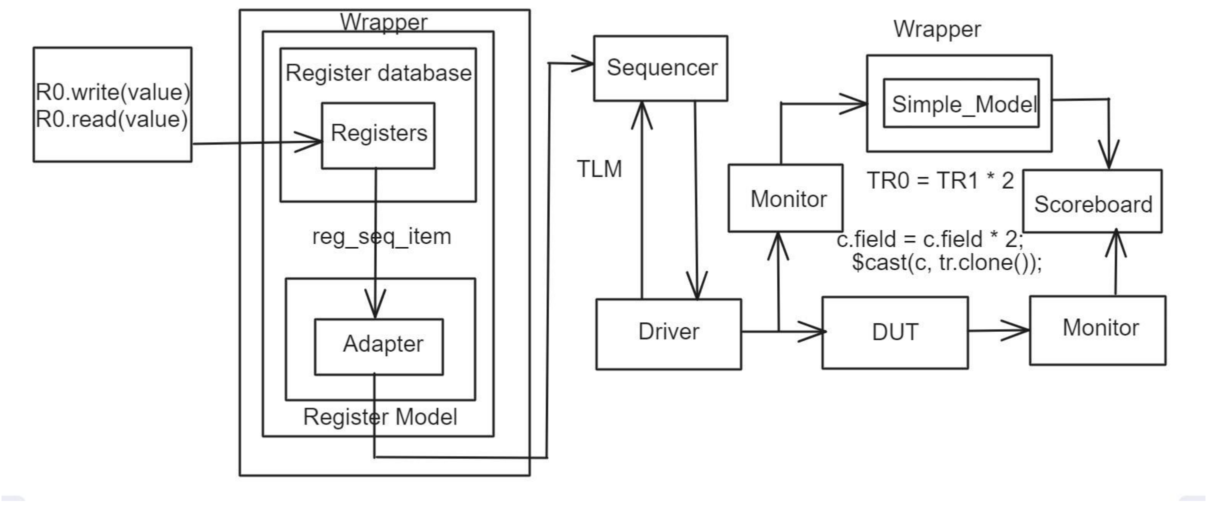


Figure 6: Register Verification Framework for SVUnit

C. Use of Python (Cocotb) testbench in Register Verification (Solution to Challenge – 3):

Cocotb [3] is a co-routine co-simulation environment that enables us to write verification codes like software with the help of Python test benches. It connects the RTL codes implemented in any HDL with the help of VPI or VHPI called cocotb and finally, it emulates the real UVM behavior with the help of Python libraries pyuvm.

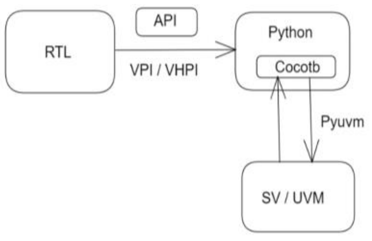


Figure 7: Cocotb as an API

But there are a few questions that need to be answered before moving on with cocotb:

[a] When there is already UVM RAL, then why Cocotb?
[b] How to connect a UVM testbench to Python?

The answers to the above questions are given below sequentially:

Advantages of Python over UVM:

[a] No involvement of macros in Python and it is less verbose and the usage of the construct is light.
[b] Open source and hence user friendly.
[c] Configurations, factory registrations, and factory overrides are simple to handle.
[d] No type is involved in Python as it directly copies the handles and is easily integrated with Python tools like numpy, scipy, etc.
[e] Reporting mechanisms are simpler with the addition of FIFO_DEBUG which helps to debug FIFO-related scenarios.
[f] Run phase has been refactored and can be used with minimal effort.

Communication of UVM TB with Python:

There are various ways through which UVM and Python can interact with each other:
[a] Using Python Client Server and Socket Connection [2]
[b] Using Foreign Language Interface (FLI)
[c] Using File-based approach.

Client Server Socket execution steps:

[a] Python server script that sends the request on the socket and processes them.
[b] DPI-C Interface in the UVM TB to communicate with Python server.
import "DPI-C" function void request(input string request_msg, output string response);
[c] Implement the DPI-C function to send requests to the python server
[d] The header file for implementing the helper function needs to be integrated for communication.

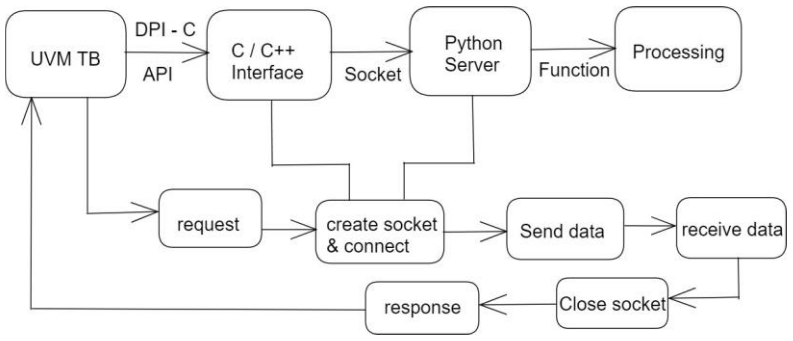


Figure 8: Client Server Socket

Foreign Language Interface (FLI) execution steps:

[a] The UVM TB calls the DPI-C function and the FLI transfers the call from the UVM TB to the Python wrapper.

[b] The Python wrapper calls the Python function from the module and then returns the result.

[c] The Python wrapper sends the result back to UVM TB through FLI

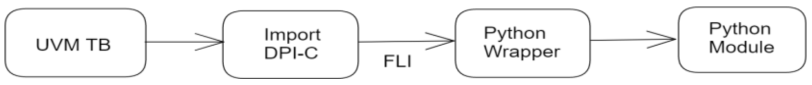


Figure 9: FLI Interface

File-based approach:

File communication involves writing data to a file from one source and reading it from another source.

Considering the above three approaches, the second approach fits the best to emulate the behavior of UVM.

With the help of Cocotb, all the inbuilt UVM RAL sequences like bit bash, reg access, and reg shared sequences can be implemented in Python with less overhead and complexity by leveraging the pyuvm library which is quite helpful in emulating the real UVM RAL behavior in Python Testbenches.

SINGLE ALGORITHM FOR COCOTB RAL (REG_MEM_PTH_SEQ):

- ❑ Divide the set of inbuilt sequences of registers and memories.
- ❑ Execute each of the register sequences like hard reset, bit bash, etc for registers and mem walk, and mem access sequences for memories.

- ❑ Enable customization features like forming clusters of groups of address based on offset for bit bash/mem walk, frontdoor write and read, and reduced simulation time for reg/mem access.
- ❑ Combine all the sequences and code the reg/mem combo sequences with conditional compilation switch and merge them.
- ❑ Implement the reg/mem shared access sequence in Python and plugin with the reg and mem builtin seq.

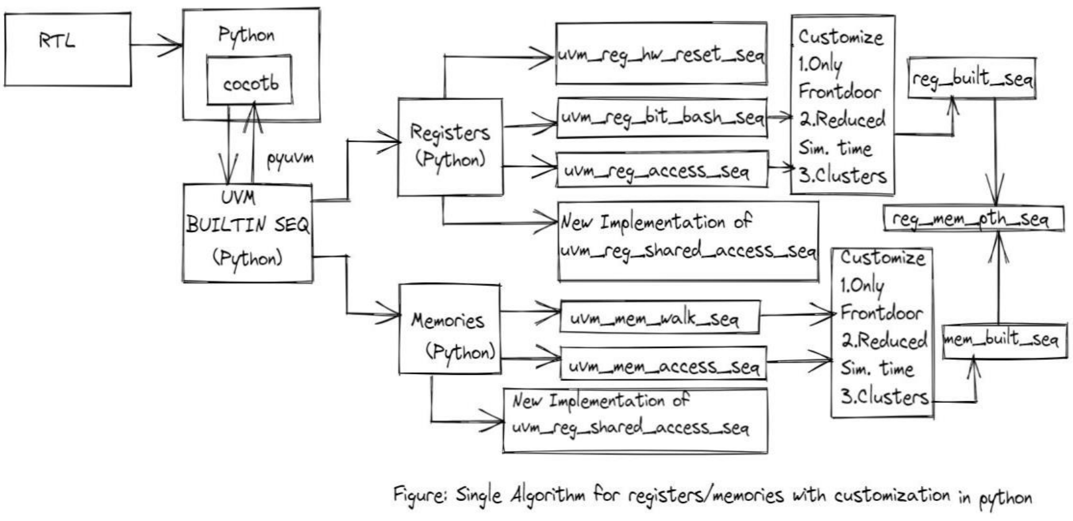

RTL
Python
cocotb
pyuvm
UVM BUILTIN SEQ (Python)
Registers (Python)
Memories (Python)
uvm_reg_hw_reset_seq
uvm_reg_bit_bash_seq
uvm_reg_access_seq
New Implementation of uvm_reg_shared_access_seq
uvm_mem_walk_seq
uvm_mem_access_seq
New Implementation of uvm_reg_shared_access_seq
Customize 1.Only Frontdoor 2.Reduced Sim. time 3.Clusters
Customize 1.Only Frontdoor 2.Reduced Sim. time 3.Clusters
reg_built_seq
reg_mem_oth_seq
mem_built_seq
Figure: Single Algorithm for registers/memories with customization in python

Figure 10: Single RAL implementation in Python

D. Use of Chisel in Register Verification (Solution to Challenge – 4):

Chisel is a hardware construction language embedded in Scala and can be used as a Design and Verification language. It can be useful to perform verification as software for register verification due to less verbosity and simplicity. Moreover, due to its concise syntax, it offers more flexibility.

The advantage of using Chisel is that since it is based on ScalaTest, it supports multithreaded behavior so that the test classes can run in parallel. Chisel uses peek, poke, expect, and step methods to operate on the DUT and can enhance the HW to SW interactions based on the project needs.

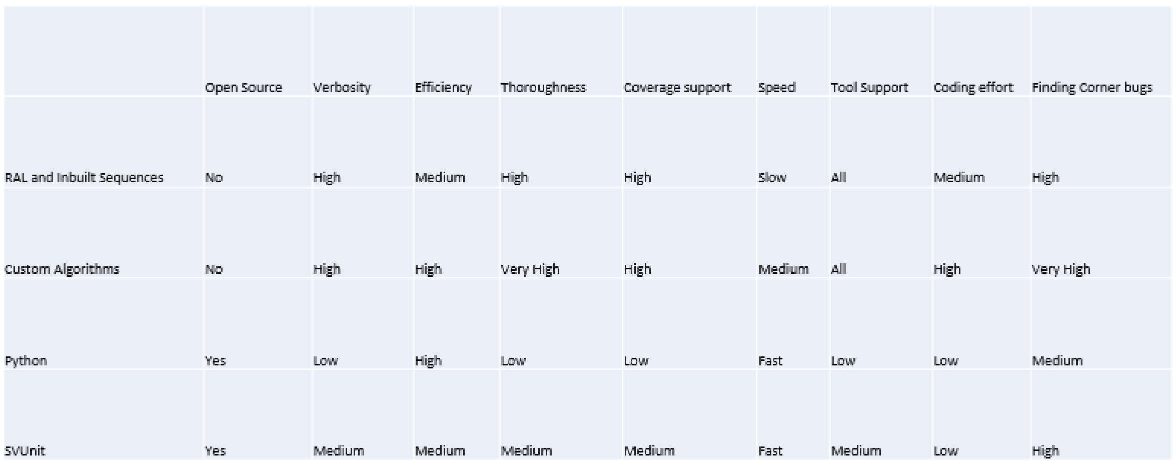
But the main advantage of using Chisel as a register verification language is working with the immutable data structure possible with "case classes".

```
case class Config(txDepth: Int, rxDepth: Int, width: Int, resetvalue: BigInt)

// Create immutable registers
  value regA = Register("RegA", 32, 0.U)
  value regB = Register("RegB", 8, 1.U)

// Use Chisel's Reg construct with immutable data
  value regC = RegInit(regA.resetValue.U)
  value regD = RegInit(regB.resetValue.U)
```

IV. RESULTS

[1] The efficiency of the register verification in frontdoor is increased by roughly 40% (*calculated based on features an algorithm supports*) by using the March-Bash algorithm because it provides additional benefits of hitting more corner case bugs, finding the transition, stuck-at-fault, and addressing decoding bugs.

[2] The process of coverage closure is improved due to the March-Bash algorithm as the method is more exhaustive and hits more areas of the code.
[3] The reusability within the testbench is increased by 20% (*calculated based on the plug-and-play feature and change in address space*) by adopting the March-Bash Algorithm.
[4] Python-based test benches using cocotb have reduced the verbosity of the test environment by 20% since the use of the construct is simple and easy and the coding and debugging effort is reduced by 40%.
[5] SVUnit is very helpful in unit testing verification and is relatively fast in execution as compared to other HVL and Methodologies due to its lightweight nature and less setup and configuration overhead.
[6] Chisel works well with immutable data structures in comparison with SV and UVM and due to its versatility and flexibility, it works well with Register verification where there is a need to use a common Design and Verification Language and open source simulators like Verilator and waveform viewer like GTKWave. Moreover, it can be used as a gateway for software professionals to learn verification faster.



	Open Source	Verbosity	Efficiency	Thoroughness	Coverage support	Speed	Tool Support	Coding effort	Finding Corner bugs
RAL and Inbuilt Sequences	No	High	Medium	High	High	Slow	All	Medium	High
Custom Algorithms	No	High	High	Very High	High	Medium	All	High	Very High
Python	Yes	Low	High	Low	Low	Fast	Low	Low	Medium
SVUnit	Yes	Medium	Medium	Medium	Medium	Fast	Medium	Low	High

V. CONCLUSION

There are multiple Register Verification frameworks discussed in this paper with each one of them having their capabilities. Depending on the project's needs, each of the frameworks can be deployed in Testbench to achieve the desired result.

REFERENCES

[1] Universal Verification Methodology (UVM) 1.2 User's Guide by Accellera

[2] How to connect SystemVerilog with Python by Amiq Consulting

[3] Cocotb: a Python-based digital logic verification framework by Benn Roser, University of Pennsylvania.