International Journal of Science and Research (IJSR)
ISSN: 2319-7064
ResearchGate Impact Factor (2018): 0.28 | SJIF (2019): 7.583

Automated Testing Framework for Functionality of Configuration of Complex Modern Architectures

Ankit Chandankhede

Abstract: ***Modern compute architectures have been architected to be scalable for different applications such as low power chips for mobile, wearable devices and high-performance chips for servers and scientific computers. The scalability and configurability of architecture and features [3] are controlled by the register writes through firmware and parameters to scale the design for instancing multiple pipelines or cores or cache size. Further register writes are also used for fallback option if a feature or fix does not function as expected. Hence verifying the register writes to enable such feature crossing with different design parameters becomes paramount of interest. The number of registers and combinations of parameters grows exponential for such architectural complexities. Thus, verification cycles can prolong and time to production can be exponentially large. This paper proposes an automated approach to randomly test the register combinations through stimulus along with design parameters by parsing feature and register documentations.***

Keywords: scalable architectures, low power chips, high performance chips, register verification, automated testing

1. Introduction

Growing applications of chips and computer architecture has significantly increased the complexity of architecture because of new feature, scalability and configurability. In modern chip design, an architecture is thought to be designed to scalable for different applications and features. Design scalability includes cache sizes, multiple pipelines for same type of workload such as compute pipeline in a graphics architecture, multiple core or execution units, instruction caches and rendering pipelines which can be scaled using synchronized parametrized design for specific applications [1] [3]. Features within such architectures are enabled or disabled through register writes and to process same type of workload to throttle the performance or often improve the depth of processing units. Such parameterized and features complexity can be bug prone and often are developed with a fallback option which is again controlled through register writes. With this architectural shift, verification strategies have been changed significantly such that an architecture or design verification is carried out versus a traditional approach of verifying a design with single application point of view. Thus increasing the complexity of the verification.

Currently verification of perceived functionality of register read and write operation along with feature has been added on top of the scalability of the architecture or design and hence permutation of scenarios have exponentially exploded and hence random or directed testing is not sufficient.

Further certain registers in design are write or read only registers which are traditionally test through Register Abstraction Layer in UVM or normal stimulus of Systemverilog [4]. However, doesn't provide the coverage metric for register testing.

This paper addresses the issues of verifying these larger set of combinations through automated register configuration constraints and parameters of design by parsing the design and architectural documentation using script. These constraints thus can be used on top of the existing testcases which allows reusability and scalability of current test suites. Further script also defined the automated cover points for each register crossing with design parameters to provide quality metric of testcases which is currently a manual process and has longer feedback loop for the validity of the register combinations and design parameters, may result in prolonged verification cycle.

2. Current Verification Methodology

Current methodology for verification of any feature including register is defined as below

a) Starts with testplanning by reading the architectural and design definitions of feature or registers and testplanner draws out scenarios to be verified. This step is prone to human error and skip over a major scenario to be verified.
b) Manually developing the test sequences for each scenario and developing coverpoint through painstaking manual process
c) Analyzing functional coverage and converging on non hit coverpoint can be protracting.
d) Unrealized coverpoint may result in silicon bugs

Different type of registers makes the presilicon verficiation even harder. Following are the type of verification extensively occurs at Unit level

1) Functional Verification
- Functional verification includes the register write and reads and testing the properties of the registers as write only, read only or write and read permitted along with reserved fields and hence it is important to test the decoder by error injecting or attempt to write a read only register and vice versa.
- Fault Injection and Reliability Testing
- While writing into the registers from AXI interface, error could be injected while writing into the registers to test the decoding

2) Security Testing
- Attempt to write a register a read only register and vice versa
- This paper extensively cover the automated framework for functional verification
- Performance testing:
- Moreover, the test framework mentioned in this paper can also be applied to performance test counters and hence covering Performance Testing

Licensed Under Creative Commons Attribution CC BY

3) Stress Testing
- Test framework can be scaled to multiple write or read with each byte enable and data clusters such double word or word or partial write of registers

Other type of testing can be tested based on the temperature or voltage variation that would dappen the clock or register state itself and testing frameworks for such verification or validation are beyond scope of this paper.

- Burn - in Testing - Is beyond scope of this paper.
- Penetration Testing**: Attempting to exploit potential vulnerabilities to assess security robustness.
- Side - channel Analysis
- Electromagnetic Interference (EMI)
- Voltage and Clock Manipulation

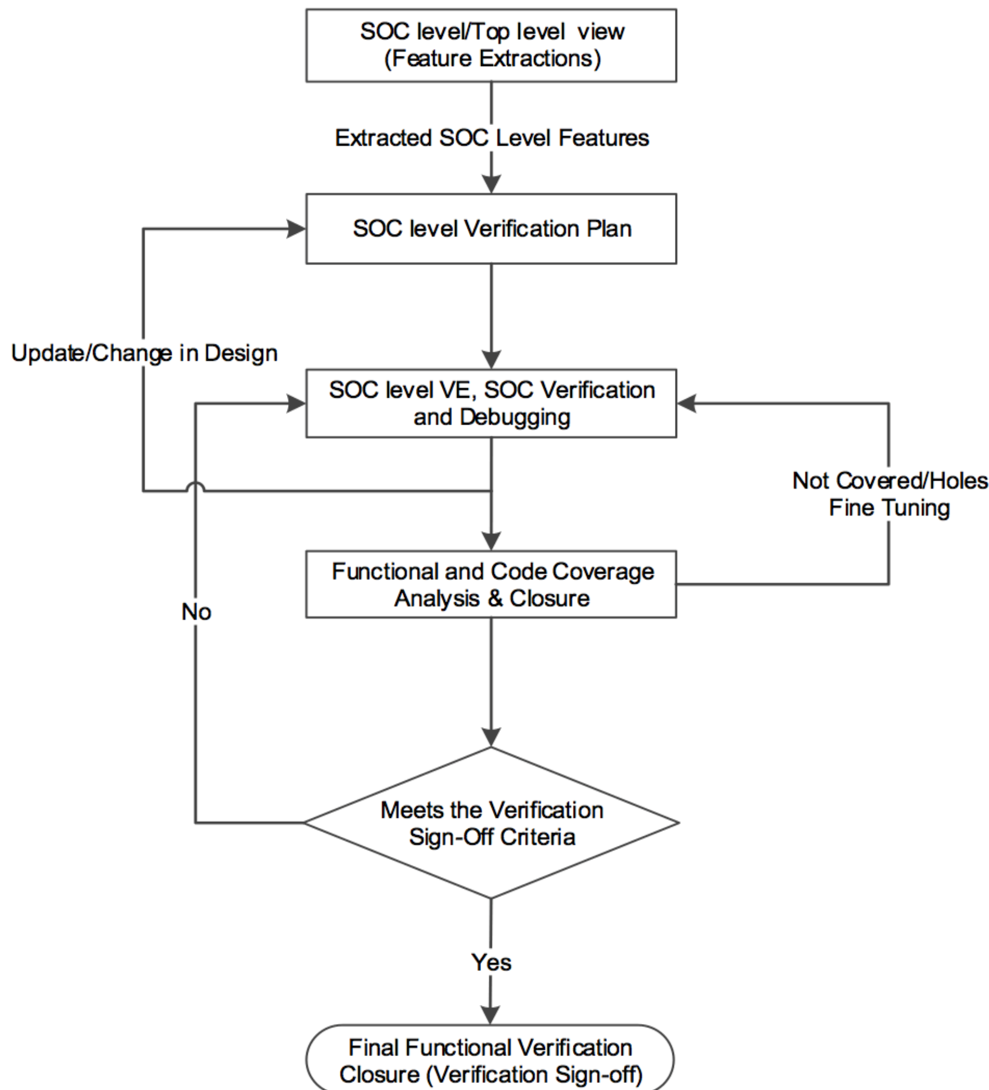

Figure 1: Shows the traditional cycle of presilicon verification

3. Proposed Automated Testing Framework

Aforementioned challenges are address in this paper unique approach by automating testing framework and effectively reducing the verification efforts on testing combination of registers along with different features and configurations. Approach is devised by a script to parse the architectural and design documentation in specific format to understand the register information and parameters of design for each product to derive stimulus with constraint of registers and parameters. This approach allows legal combinations to be applied on pre - existing testcases. Thus brings the testcase suite quality, scalability and portability to different DUTs.

These constraints of registers are derived in SystemVerilog and files of these constraints can be easily included in any stimulus, promoting portability across DUTs from IP to SoC. Thus, reducing the similar implementation of test setup across DUTs. Additionally, if the unified register/RAL model is used by all DUTs, the stimulus of register testing can be ported at any DUT.

Further testing of registers is complemented by functional cover point which are automatically generated based off the constraint derived by scripts and thus engineering effort for coding of the cover point is saved.

Proposed approach is providing end to end approach of verification of register and design parameters using existing testcases and helps reduce human error on determining critical scenarios and covers cross coverage of registers with parameters.

Licensed Under Creative Commons Attribution CC BY

4. Implementation Details

The implementation of the automated testing framework involves:

Script Development:
Script is used to parse architectural documents and derive the constraints in a tabular form as mentioned below for HVP mapping and constraint mapped internally by script are utilized to create the SystemVerilog/UVM constraints and generate the stimulus to either write or read the register using RAL or direct stimulus on register interface of a unit.

- **Automated HVP mapping created by Script**

Table created by script

Table 1: HVP mapping for register coverpoint

Register name	Register field	IP	Width	Read/Write permissions	Valid values	Cross valid	Default values
A_register	A_len	Cache	4	Write only	0 - 10	A_type_security 0 - 7 B_type_security 0 - 4	0
Error_register	Context_id_err	Cache	7	Read only	0 - 127		0
Security_register	A_type_security	Cache Controller	1	Write & read	0 - 1		0
Security_register	B_type_secuirty	Cache Controller	1	Write & read	0 - 1		0

Table 1. shows the name of IP (cache / cache controller) with register name with its register fields, its width, permission of register field, valid values and the constraint if there is cross dependency on registers and its values expected.

For example: A_len is a register field of A_register from IP name “Cache” which has width of 4, is write only with valid value of 0 - 10 but valid value is 0 - 7 if A_type_security register field is set and 0 - 4 if B_type_security_register is set.

- **Automated generated SystemVerilog and UVM constraints**

Although UVM framework provide the register stimulus however does not constraint across the units and register map. The same script uses the map created on possible values of the registers to create SystemVerilog or UVM constraints for each IP as follows:
Following code shows the constraint generated by the script based on the register mapping and property per Unit.

```
class cache_register;
rand bit [3: 0] A_len;
rand bit [6: 0] context_id_err;
constraint A_len_c {
 A_len inside { [0: 10]};
}
constraint context_id_err_c {
 context_id_err inside { [0: 127]};
}
endclass

class cache_controller;
rand bit A_type_security;
rand bit B_type_security;
constraint A_len_c {
 A_type_security inside { [0: 1]};
}
constraint context_id_err_c {
B_type_security inside { [0: 1]};
}
endclass
```

Following code shows the cross constraint generated by the script based on the register mapping and property across the DUT.

```
class cross_dut_register;
 rand bit [3: 0] A_len;
rand bit [6: 0] context_id_err;
rand bit A_type_security;
rand bit B_type_security;
constraint cross_A_len {
 solve A_type_security before A_len;
 solve B_type_security before A_len;
 if (B_type_security) { A_len inside { [0: 3]}; }
 else if (A_type_security) {A_len inside { [0: 7]}; }
 else { A_len inside { [0: 10]}; }
 }
Endclass
```

Stimulus generation
These constraints are used in stimulus of SystemVerilog or UVM sequence are used to test registers with following type of sequences

1) Check if the default value of the register is as expected after the reset
2) Double word or word is written by selective byte enable (partial) writes and the values are reflecting as expected using a read after write sequence.
3) Register writes are followed by reset of IP or unit and read back the register values of register to be either default value for resettable registers and to be the same value as previously written value for non resettable registers.
4) Cross register constraints are used in the DUT while enabling features across the units. These constraints provides an effective way of integrating on top of any testcase and thus promoting the scalability of the testcases and allowing different combinations of registers to be automatically created.

This technique has eliminated the effort to create different stimulus to enable a feature and cross with other features across Units and thus improving coverage by 70%.

Automated Register Functional Coverage
Script further generates the coverpoint based off of these constraints to complement the HVP mapping as well as stimulus generated. Thus reducing the effort of coding coverpoints and complementing the test framework with functional coverage metric. Although UVM framework provides the coverage at transaction level, however they are

Licensed Under Creative Commons Attribution CC BY

also possible to analyzed through code coverage. Current Methodologies doesn't provide the cross but coverage for registers which may have dependency across Units and hence this automated approach based off the prior table 1 mapping provides a unique opportunity to automate coverage creating and provide comprehensive DUT view.

Following automated coverages are created by script based off prior mapping and shows the crossing of A_len and type of security enabled in DUT.

```
covergroup A_len_group;
A_len_c: coverpoint A_len { bins A_len_bin = { [0: 10]}; }
context_id_c: coverpoint context_id_err { bins context_id_err_bin = { [0: 127]}; }
A_len_cross_context_id: cross A_len_c, context_id_c;
A_type_security_c: coverpoint A_type_security { bins A_type_security_bin = { [0: 1]}; }
B_type_security_c: coverpoint B_type_security { bins B_type_security_bin = { [0: 1]}; }
A_type_security_cross_A_len_c: cross A_type_security_c, A_len_c;
B_type_security_cross_A_len_c: cross B_type_security_c, A_len_c;
endgroup
```

5. Benefits and Future Directions

Automated framework modelled by a script is providing end to end verification of registers testing along with feature enabling capabilities across DUT.

a) Efficiency:
   Automated approach has eliminated register code coverage, testplanning, HVP, test constraints and stimulus generation, which is cumulatively accelerating the verification process.
b) Accuracy:
   Since the script is parsing the architectural definition of the feature or registers and extracting the dependencies of the register, is further improving the quality of the test plan and removing chance of the human error.
c) Scalability:
   Since the constraint generated by script are across the DUT, it can be applied to existing test sequences and hence allowing the scalability. Proposed frameworks can be scaled to different architectures and DUT level of verification.
d) Future Enhancements:
   Feature enabling has different stages such as overall system level testplanning, coverage coding, generation of test stimulus for data flow, data integrity checks, convergence of code and functional coverages. This paper only cover the register testing part of verification education and can be scaled similar to other verification strategies.

6. Conclusion

End to end testing through this automated testing framework proposed in this present provides significant improvement in reducing verification cycle for register testing, enabling features and provides path ahead for improvements in verification process. Frameworks capability to scale to diverse complex architecture including Graphics, CPU and AI accelerators and provides This approach not only reduces project timelines but also improves the overall quality and reliability through automated register constraints and scaling constraints on existing testcase and provides a complementing metric of functional coverage. This framework can be further scaled to protocols such AXI and AMBA protocols [6] and thus reducing the verification effort even further.

References

[1] Matthieu Tuna and Mounir Benabdenbi "Software Based Self - Test of Register Files in RISC Processor Cores using March Algorithms"
[2] Jaume Abella Pedro Chaparro Javier Carretero Jaume Abella "End - to - end register data - flow continuous self - test"
[3] C. Ebeling; D. C. Cronquist; P. FranklinConfigurable computing: the catalyst for high performance architectures
[4] Verification academy general "RAL in UVM: https: //verificationacademy. com/verification - methodology - reference/uvm/docs_1.1b/html/files/reg/uvm_reg - svh. html
[5] AXI and AMBA protocol https: //developer. arm. com/documentation/ihi0022/latest/

Licensed Under Creative Commons Attribution CC BY