IMPROVEMENT IN VERIFICATION EFFICIENCY BY USING REGISTER ABSTRACTION LAYER

Major Project Report

submitted in partial fulfillment of

degree

of

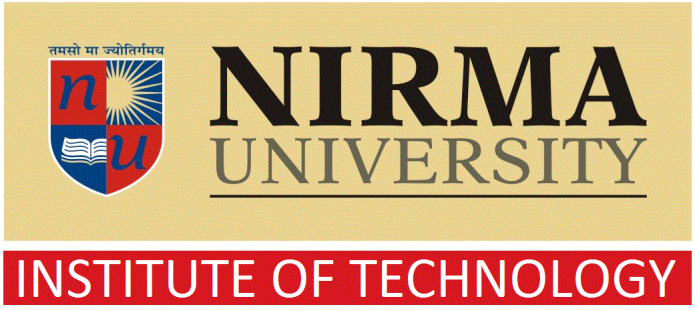
MASTER OF TECHNOLOGY

in

EMBEDDED SYSTEM

submitted by

ABHIJEET C VIBHANDIK



DEPARTMENT OF ELECTRICAL ENGINEERING
INSTITUTE OF TECHNOLOGY, NIRMA UNIVERSITY, AHMEDABAD.

December,2014

IMPROVEMENT IN VERIFICATION EFFICIENCY BY USING REGISTER ABSTRACTION LAYER

Major Project Report

submitted in partial fulfillment of

degree

of

MASTER OF TECHNOLOGY

in

EMBEDDED SYSTEM

submitted by

ABHIJEET C VIBHANDIK

under the guidance of

MR. VIJAY SAVANI

DEPARTMENT OF ELECTRICAL ENGINEERING
INSTITUTE OF TECHNOLOGY, NIRMA UNIVERSITY, AHMEDABAD.

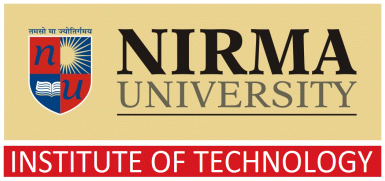
December,2014

Declaration

This is to certify that, the thesis comprises my original work towards the degree of **Master of Technology** in **Embedded Systems** at **Nirma University** and has not been submitted elsewhere for a degree.

Due acknowledgment has been made in the text to all other material used.

- Abhijeet C Vibhandik



CERTIFICATE

This is to certify that the Major Project entitled "IMPROVEMENT VERIFICATION EFFICIENCY BY USING IMPLIMENTATION OF REGISTER ABSTRACTION LAYER" submitted by Vibhandik Abhijeet Chandrashekhar (13MECE23), towards the partial fulfillment of the requirements for the degree of "Master of Technology" in "Embedded Systems" at "Nirma University, Ahmedabad"; is the record of work carried out by him under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.The results embodied in this major project, to the best of our knowledge,haven't been submitted to any other university or institution for award of any degree or diploma.

Date: Place: Ahmedabad

Mr. Vijay Savani
Project Guide

Dr. N.P. Gajjar
Program Coordinator

Dr.D.K. Kothari
Section Head, EC

Dr. P.N.Tekwani
Head of EE Dept.

Dr. K. Kotecha
Director, IT-NU

CERTIFICATE

This is to certify that the Major Project "IMPROVEMENT VERIFICATION EFFICIENCY BY USING IMPLEMENTATION OF REGISTER ABSTRACTION LAYER "submitted by Vibhandik Abhijeet Chandrashekhar 13$MECE$23, towards the fulfillment of the requirements for the degree of Master of Technology in Embedded Systems, to Institute Of Technology, Nirma University, Ahmedabad is the record of work carried out by him under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.

Date: May 13, 2015

Mr. Vishal Dewan

Technical Lead
Intel India Technology pvt Ltd,
Banglore

Mr. Vishal Namshiker

Manager
Intel India Technology pvt Ltd,
Banglore

Mr. Saish Amonkar
Manager
Intel India Technology pvt Ltd,
Banglore

Abstract

Registers and memories defines software interface to device, and resembles large part of programmer's guide. As registers define HW/SW interface for device, it's very important that register perform as expected for correct operation of design. Thus registers need to be verified through product life cycle.

With increased design size and complexity, today's design contains several hundreds to thousands of registers. From specification, documentation to design implementation, verification of each register, each bit and there property involves lot of effort and complexities, which consumes time. Moreover maintaining separate register specification document for each purpose, and manually updating these documents for multiple iterations through product life cycle is error prone and tedious task as number of register increases above few hundred.

Use of single source written in high level language like SystemRDL help reduce complexity for generating documents of software, design and verification.

Thesis describe a methodology which provides low maintenance, almost zero time and reusable register design and verification environment. Project leads to stable and promising OVM based register verification environment, and Register Abstraction Layer (RAL) is discussed. A methodology provides a complete solution form register specification in .xls format to RAL files to reusable verification components and environment.

Thesis presents useful RDL constructs for modelling scalable register descriptions, like registers arrays, reg-files and register field instantiation. A sample example for RDL specification helps understand SystemRDL constructs, field, register, register file and addressmap instantiation.

Thesis also describes Register Model, lists main components in register model with overview of each. Register Model use flow specified in thesis helps understanding how RAL is integrated into existing environment.

At the end results are discussed, for register verification test with use of BFM (insted of processor). The results are compared with simulation results of C test run on processor. Result are impressive with 21.46% reduction in CPU time with specified methodology. Also, use of methodology results large improvement in verification efficiency by reducing register verification time to period of week instead of month.

Acknowledgements

With immense pleasure, I would like to take this opportunity to thank all those who helped me for the successful completion of the first phase of the dissertation and for providing valuable guidance throughout the project work.

I would first of all like to offer thanks to Dr. N. P. Gajjar, Program coordinator M. Tech. Embedded System, Institute of Technology, Nirma University, Ahmedabad whose keen interest and excellent knowledge base helped us to finalize the seminar topic. I would offer thanks to Asst. Prof. Vijay Savani, Guide; Institute of Technology, Nirma University, Ahmedabad. His constant support and interest in the subject equipped me with a great understanding of different aspects of the required architecture for the project work. He has shown keen interest in this dissertation work right from beginning and has been a great motivating factor in outlining the flow of my work.

Thanks to my associates at Intel Mr. Saish Amonkar, Mr. Vishal Namshikar and Mr. Vishal Dewan for continuous guidance and support. Mr. Varun Gupta for consistent advice and support for GLS activities. I would also like to thank Mr. Jatan Bhatt and Mr. Vinu Bhaskar for their advice and help in register abstraction layer integration and their inputs regarding feasibility of implementation. And thanks to all who have directly or indirectly helped us making this work successful.

- Abhijeet C Vibhandik
13MECE23

Abbreviations

RAL	Register Abstraction Layer
SRDL/RDL	SystemRDL,register description language, it is an ieee standard maintained by Acellera
OCP	Open Core Protocol is on chip interface standerd
BFM	Base Fuction Module
SV	System-Verilog
OVM	Open Verification Methodology
UVM	Universal Verification Methodology
OVC	Open Verification Component
UVC	Universal Verification Component
HDL	Hardware Description Language
HTML	Hypertext Markup Language
IP	Intellectual Property
LSB	Least Significant Bit
MSB	Most Significant Bit
RTL	Register Transfer Level

Contents

Declaration i

Certificate ii

Abstract iv

Acknowledgements v

Abbreviations vi

1 Introduction 2

1.1 Verification . . . 3

1.2 Basic Verification Principle . . . 3

1.3 Verification Methodology . . . 4

1.3.1 Simulation Based Verification . . . 4

1.3.2 Formal Method Based Verification . . . 5

1.4 Verification Challenges . . . 6

2 Register Verification 7

2.1 Need For Register Verification . . . 7

2.2 Legacy Flow . . . 8

2.3 Basic Register Verification Principle . . . 9

2.3.1 Test plan development . . . 9

2.3.2 Verification Strategies . . . 10

2.3.3 Test Case Development . . . 10

2.3.4 Considering Corner Cases 11

3 Register Model And Integration Overview 12

3.1 Register Specification 12

3.2 SystemRDL Overview 13

3.3 Register Model Use Flow 14

3.4 Register Model Integration Overview 15

4 SystemRDL 17

4.1 SystemRDL 17

4.1.1 Key Concepts in SRDL 18

4.2 Register Model 19

4.2.1 Defining Components 19

4.2.2 Instantiating Components 21

4.2.3 Component Properties 21

4.2.4 Assigning Default Values 22

4.2.5 Dynamic Assignment 22

4.2.6 Property Assignment Precedence 23

4.2.7 Scoping 24

4.2.8 Universal Properties 24

4.3 Example with systemRDL 25

5 Implimentation 28

5.1 Adapter 28

5.2 Bus Agent 30

5.3 Predictor 30

5.4 Front Door and Backdoor Access 31

5.5 Test and Test sequence 31

5.6 Considering Corner Cases 31

5.6.1 Lock Bit 31

5.6.2 Write Only Register 32

5.6.3 Special Bits 32

5.7 Implementation Algorithms 32

6 Simulation and Results **35**

6.1 Simulation 35

6.2 Results 35

7 Conclusion and Future Scope **39**

7.1 Conclusion 39

7.2 Future Scope 39

References **41**

A Register Description **42**

B Example 1 Complete SRDL **47**

B.1 RDL Top 47

B.2 ACV Register File 47

B.3 SPI Register File 48

B.4 I2C Register File 50

B.5 RAL Files 52

List of Tables

3.1 Register, Field and Block basic attributes 13
3.2 Basic Constructs Required For Register Model Integration 16

4.1 Component Types 20
4.2 Universal Properties 25

A.1 ACV Register Memory Map 42
A.2 SPI Register Memory Map 42
A.3 I2C Register Memory Map 43
A.4 acv_ctrl_0 Register Field Description 44
A.5 acv_RSVD_0 Register Field Description 44
A.6 spi_ctrl_0 Register Field Description 45
A.7 spi_ssienr Register Field Description 45
A.8 SR Register Field Description 46
A.9 IMR Register Field Description 46

List of Figures

1.1 Design Flow 2

2.1 Legacy Flow[3] 9

3.1 Fields in Register 13
3.2 Steps Involved In Using Register Model 15

5.1 Register Abstraction Layer [2] 29

6.1 Simulation waveform for C based register verification test 37
6.2 Simulation waveform for SV test or RAL based register verification test 37
6.3 CPU time taken for C based register verification test execution 38
6.4 CPU time taken for SV test or RAL based register verification test execution 38

Chapter 1

Introduction

Design verification is the process which ensures that a design will meet its required specifications. This chapter introduces the basic concepts of hardware design and verification, and will cover the methods/techniques of design verification and verification challenges.

During a design process we transforms a set of specifications into an implementation that satisfy all specification. Specifications gives an idea about functionality of design but doesnt specify how to be achieve the functionality. Lower abstraction layer, specify more of design details. The increase in details with decrease in level of abstraction can be seen in below figure [1.1].

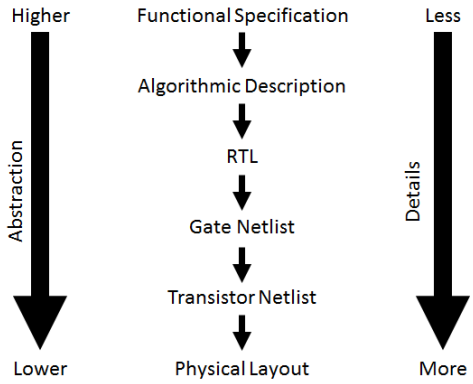


Figure 1.1: Design Flow

1.1 Verification

Design verification is the reverse process of design, it starts with implementation to prove that design meets the given specifications. For every step of deign there will be at least one verification step. For example, for design step that converts functional specifications into an algorithmic implementation, will require a verification check to ensure that the algorithm performs the specified functionality. In general verification can be performed in various areas like functional verification, timing verification, layout verification, electric verification etc. In this chapter we discuss only functional verification and refer to it as design verification.

At lower abstraction verification can be classified in two types. First type verifies that two versions of design are functionally equivalent and is called as equivalence checking. One common scenario of equivalence checking is comparing two versions of circuits at the same abstraction level. For example RTL and Net-list equivalence checking.

1.2 Basic Verification Principle

There can be two types of design error. One, the error may exist in design, not in specifications. May be because of wrong implementation process. Second, the error is in specifications itself. A human error may result in any of these errors. Common human error may be in interpreting design functionality. To avoid such errors, we may use a software program to synthesize an implementation, directly from the specifications. Error may still result, even with this approach. This synthesis approach is limited in practice due to two reasons. First, as most of specifications and functionality is in form of conversational language, such as English instead of usual mathematical language such as Verilog or C++. Second, at such high abstraction level functional requirements may not represent timing behavior, instead timing requirements are clearer at lower abstraction level (implementation level).

Another more widely used method to reduce such errors is through redundancy. With this approach, same specifications are implemented two or more times using different approaches and results of approaches are compared. Although in practice more than two approaches are rarely used, as more errors may occur with each alternative verification approach also cost and time can be insurmountable.

Second type of error which exists in the specifications. It can be unspecified functionality, conflicting requirements, and unrealized features. One way to catch this the type of error is with use of redundancy, since specifications are already at the top of the abstraction hierarchy and so there is no reference/golden model, which can be used to compaire.

1.3 Verification Methodology

A good verification methodology starts with a test-plan which details the specific functionality which is to be verified, so that the specifications are satisfied. A test plan consists of various features, operations, corner cases, transactions etc. So if we verify these items then we can say that we have verified specifications. To track progress over a test plan, a "score boarding" scheme is used. In score boarding, the items from the test plan are marked as done when they are fully verified. Its necessary that we measure a quality of verification. Most commonly used coverage standard of measurement are functional coverage and code coverage. Functional coverage approximates the percentage of functionality that is verified, whereas code coverage represents the percentage of code simulated.

Apart from test plan, a method of verification is to decide between, what language should be used. For example, arithmetic operators are often in verification code, but rarely in design codes. Moreover, in contrast to verification code, design code generally has to be synthesized. So, ideal verification language must resemble a software language more than hardware language. Vera, Verilog, C/C++, Java and System-verilog are most popular verification languages.

Mainly there are two verification methodologies, simulation based verification and formal method based verification. The differentiating factor between methodologies is vectors. Simulation based verification methodologies depends on vectors, while formal method based verification methodologies doesnt. Other factor that distinguish simulation-based and formal method based verification, is that formal method verification is output oriented where output properties are provided by designer, and simulation is input oriented where input is provided by designer. Whereas, the methodology which takes input vectors and verifies formally around the neighborhood of vector is known as Semi-formal verification. Since semi-formal methodology is a mixture of formal based and simulation based technology, its not explained separately here.

1.3.1 Simulation Based Verification

The most widely used verification method is simulation based verification. In simulation based verification, the RTL is integrated in a test bench, test bench is provided with input to be applied to DUT, and design output is compared with golden/ref. output. A test bench include a code that supports functionality of the design, and quiet sometimes TB generates input stimulus and measures output against the reference output, too. The stimulus to be provided as input can be generated prior and in design it can be read from database while execution/simulation. Or it can be generated on fly while simulation run. In a same way, the golden/reference output is either generated in before in advance or during simula-

tion. Afterwards, a golden/reference model is simulated in lock step with the design, and results from both models are compared.

There can be event driven simulator or cycle based software or hardware simulator. Whenever an input of gate or the variable to which the block is sensitive, changes a value event based simulator will evaluate the block of statement or gate. A change in value is known as an event. While circuit is partitioned according to clock domains, in cycle based simulation. And each sub circuit is evaluated once at each edge of clock in cycle based simulator. That is why, simulator speed gets affected by event count. Lower the event count in circuit faster will be event based simulation. Whereas cycle based simulator can run faster for circuit with high event count. In practice, most of circuits will have enough large number of events that cycle-based simulators performs much better than event-driven simulator. However, cycle-based simulation has its own drawbacks. A clock domains must be well defined in circuit to be simulated with cycle based simulator. For example, a circuit which doesnt have clear clock domain for example asynchronous circuit cannot be simulated in a cycle-based simulator because, we does not have a transparent clock domain definition since there is no clock involved.

1.3.2 Formal Method Based Verification

The formal method-based verification methodology does not require the generation of test vectors like in simulation based verification methodology; otherwise, it is similar. Formal verification can be classified into two major categories:

- Formal equivalence checking
- Formal property verification

Functional equivalence between two implementations is determined by equivalence checking, whereas the other type of formal verification is property checking. Property checking mainly takes two inputs a design and a properties which are a partial specification of the design, and proves or disproves that the design holds the property. A property is essentially mimics a design description, and it acts to confirm the design through redundancy. "Model checker" is a program that checks a property. The idea behind property checking is to search the entire state space for points that fail the property.

Some practical problems with property verifiers embody long iteration times in decisive the right constraining parameters, and debugging failures within a properties and design. Since only portion of design in provided to property verifier, an environment around the portion of design must be modeled properly. Most of practical experience shows that an outsized share of time (appr. 70%) is spent in obtaining the right constraints during verification. Second, debugging a property is

tough once the property is written in a language apart from the one used in design. For example, the properties may be written in System-Verilog while design is written with Verilog.

1.4 Verification Challenges

From a cycle of project development, we can perceive the problem in design verification. A design engineer sometimes constructs a design based on provided specifications. However, a verification engineer should verify a design in all possible cases, usually in practice for complex designs we have huge number of cases. Even with a significant investment of resources in verification, it's commonplace for a fairly complicated chip to go through more than one tape-outs before it goes to market for revenue.

The impact of design verification cant be ignored. A faulty chip not solely drains budget through re-spin prices, it conjointly increases time-to-market, affects overall revenue, shrinks market shares, and forces the corporate into taking part in the catch-up game. To scale back verification time and price is one amongst major challenges in SOC or ASIC development process.

Majorly there are four areas where a design projects face verification challenges. One, there is no specific decent way to study a functional situations that must be tasted. Second, additionally it takes an excessive amount of effort to make a random test-bench. Third, its hard to attain coverage closure. Random testing is good to achieve up to first 80% of coverage. however the last 20% of coverage is specifically tough to attain. It conjointly needs several manual iterations, consuming time and efforts.

There are lot of efforts going on in industry to reduce, verification cost both in terms of money and time. Lots of efforts are being done for automation of verification environment, to address these challenges.

Here in this project I have implemented Register Abstraction Layer with proprietary system-verilog class library, for automation in Register Verification. Implementation has much of resemblance to UVM register model. So much of part is explained with reference of UVM register model.

Chapter 2

Register Verification

Sometimes the tiniest things will do the largest tasks when it comes to verification. Configuration registers is good samples for this. Configuration registers are very important for correct functioning of design, so we'd like to verify every bit/field of each register in SOC or IP. Registers and memory components represent large share of todays complex designs. On-chip registers outline the software system interface to the chip, and typically represent the most important portion of the chip specification or programmer's guide. Its an increasing challenge to maintain documentation, implementation and maintenance with increase in number of registers. Probability of introducing errors in process and productivity gets affected, if we have to manage these components manual.[3]

Most times, though, registers have a regular structure, defined by their field attributes. Using this characteristic, it is possible to define a flow where the register architecture is defined in a high level register description language like SystemRDL, which in turn is used to generate the design, documentation and verification components. This helps to reduce the often tedious and error-prone task of managing registers, and enables design, verification and firmware teams to work more efficiently from consistent and synchronized views of the chip design.

2.1 Need For Register Verification

To understand importance or need for register verification, let's take an example. Most of the host interface bus protocols contain a base address register (BAR) that is generally used for memory mapping. Assume the size of the BAR is 32 bits, and the number of read-only bits (with default value 0) within the BAR represents the size of memory the device requires. Now, per protocol, the host writes 0xFFFF_FFFF to the BAR and then reads back to calculate the required memory / I/O space for the device. If the read value is 0xFFC0_0000, then that means the memory /

I/O space allocated to that device is 2MB where the last 22 bits are ready-only bits. Lets say Read/Write attributes for the BAR have not been verified properly and the design has a bug where the 22nd bit is writable instead of read-only. In this case, the host writes 0xFFFF_FFFF to the BAR; when it reads back, it finds 0xFFE0_0000, which represents 1MB of the memory size while the intended size is 2MB. Incomplete attribute verification of a single bit can reduce memory allocation to half. Sometimes such errors consumes lot of time to debug and uncover the main cause for fault, if detected later in design development cycle.

2.2 Legacy Flow

Register definition starts as architect prepare specifications. As specifications become ready verification engineer, software engineer and hardware engineer starts coding different views of register description based on function specifications. And as soon as design is completed verification engineer and software engineer start running tests. As an when any bug is discovered design and specifications must be changed, but by mistake or due to scenarios and priorities, it may happen that designer might would have updated design but document is not updated or vice versa. This process iterated multiple times during the course of project. Bug is not only source of change here marketing request may also come in between adding or removing any feature, requiring specifications to be modified and accordingly downstream code to be modified. As we see this flow is prone to error, documents, design, verification and software may go out of sync during this process. (ref. Fig:2.1)

Generally coding an RTL for registers is long effort for designer (lets say 3 weeks), as there in complex industrial design there can be hundreds s of registers with each register having numerous fields with each field having different attribute. Once designer is done with his work, verification engineer will develop a reusable randomized verification environment with tests like reset value check and read write or attribute check tests, lets say verification engineer takes more 2 weeks (at least). Moreover closure based on verification feedback is time intensive process. Overall there is huge effort in months plus maintenance overhead, when address mapping is modified or register is added or removed.

This flow is prone to errors, as there could be disconnect in between design-document, design, verification environment and software. The automated register design and verification (DV) flow streamlines this process.

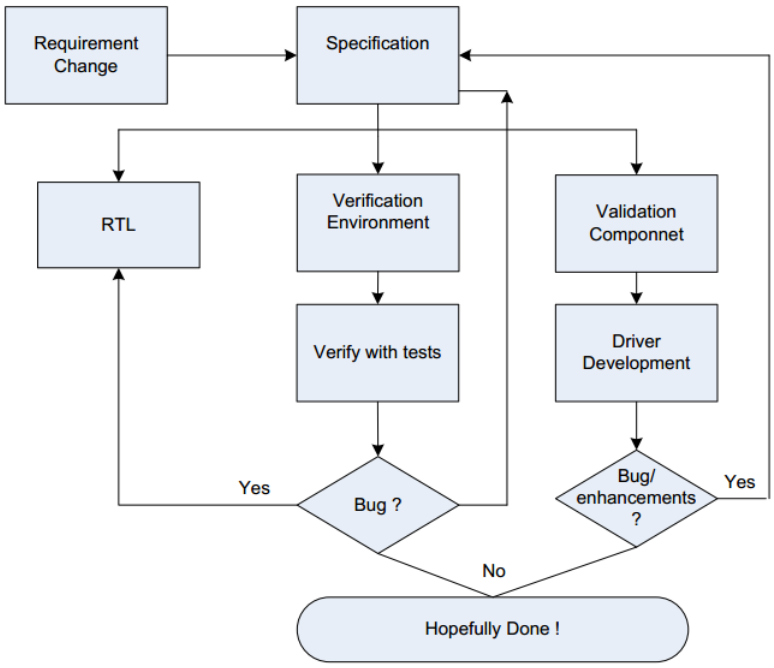


Figure 2.1: Legacy Flow[3]

2.3 Basic Register Verification Principle

There are various methods to reduce efforts required in verification of registers, make it risk free and less time consuming. Register Verification strategies are well explained in Cadence's article on "Incisive Verification Article"[6] . In below section we will see at few methodologies which we can apply to any kind of register verification. We will see how small pre-planning can help us make register verification more effective and much easier.

2.3.1 Test plan development

For any verification first thing to do is to create verification plan. Register verification plan is devidied into three major test groups.

1. Reset value or default value verification
   Tese tests must cover reset value or default value verification of each field of each register.

2. Attribute verification
   These test are intended to perform attribute testing of each field of each register. Attribute can be RW (read write), RO(read only), RW/1C(read write

one to clear), WO(write only) etc. Moreover each field in register can have different attribute.

3. Functional verification
   Registers are not just read write, but most of times there will be some functionality to be performed based on current value of register or some FSM might be expected to get executed. Functional register verification tests covers these kind of cases and make sure that desired or expected functionality is getting executed.

2.3.2 Verification Strategies

For register verification we write to register then read it and based on write value and attribute/mask of register we will generate expected value for that particular transaction and compare expected and actual read value. In register verification each bit of register must be exercised for both 1 and 0 value. In terms of verification methodology we must get 100% functional coverage for register attribute and default value verification. With directed testing its very difficult and time consuming any may lead to bug later on in future. Example explained in Sec:2.1, puts real value in putting together a good strategy.

2.3.3 Test Case Development

If we develop test case for each register, we will land up with hundreds or thousands test cases, which will be difficult to manage. Moreover, during test run each time design need to be initialized before we start with register verification, which takes time. Also time for design initialization depends on design complexity. We may also want to run regression several times, as we have random test cases. If we consider time for design initialization be 1 minute, with thousands of such tests we can get idea of amount of resources and time we will require to finish the register verification.

A quite good approach is to have one test covering default value and/or attribute check for all registers. In this case we will be able to save time require for design initialization. So we will reduce regression and simulation time significantly. It may even reduce regression time from a couple of days to few hours. For functional verification we may require to have separate test for each functionality based on scenario. But for reset/default value verification and attribute verification its better to reduce no of test cases, as we are not doing much in it.

2.3.4 Considering Corner Cases

We must consider all possible scenarios while developing test for verification. Similarly there can be various corner cases for register verification lets discuss few most common.

For example if there is error in encoding/decoding of register address, that is if in design both address locations are swapped. Lets say first address as AX and other address as BX. Now due to bug in design whenever you try to write on address Ax, you are actually writing on to address BX. And when we read form Ax design read from Ax and returns data. It's difficult to catch such bugs. Even with help of all register verification utility, its quite difficult to catch. Such scenario requires manual debugging.

Its better to use register modeling utilities for register verification so as to achieve 100% coverage. Also register model reduces tedious and error prone manual process of maintaining verification environment. Also register model provides quite impressive way to synchronize various teams during ASIC development cycle. One can use third party register models, if there are no project specific modification to be done in register modeling utility. Even if there are few modifications to be done we can write wrapper around such third party models to add project specific features.

Register model must be hooked up to proper location in verification environment. We can integrate register model as independent utility for register verification or can hook up it like any other monitor to perform auto verification of all configuration registers.

Chapter 3

Register Model And Integration Overview

Register specification involves various attributes like base address, default value, access modes, security attributes etc. With all those various attributes changing repeatedly during design process, register verification process is becoming growing challenge. For every change we need to propagate the change down the complete environment and various TB components. There has to be modification in TB for each change register or field or memory is tedious and time costly task. Moreover older manual approach may cause defects or errors as number of registers to verify exceeds two hundreds and so.

This chapter consist of brief introduction about various types of register, register attributes, register model and basic register access flow using register model.

3.1 Register Specification

Hardware functional blocks connected to host processors are managed via memory mapped registers. This means that each bit in the software address map corresponds to a hardware flip-flop. In order to control and interact with the hardware, software has to read and write the registers and so the register description is organized using an abstraction which is referred to as the hardware-software interface, or as the register description.

H/w s/w interface maps addresses in Input output memory to registers, where registers are identified by mnemonics. Each register is divided into fields, where each field may have more than one bit, refer (fig.3.1). That is field may have single bit, or can be large enough as register. Moreover each field has different attributes, it can be R/O (read only), R/W (read write), RW/1C (read write 1 to clear) etc. One may have reserved registers or fields, to be used in future or might not been

used at all in design.

Considering IP we can say registers can be grouped in to two types. First, registers those can be accessed by host. These are configuration registers mostly. Second, registers those are internal to IP and not visible to host. Usually controlstatus registers fall under second category. Although IP may have sub IP's in it, and same applies to them.

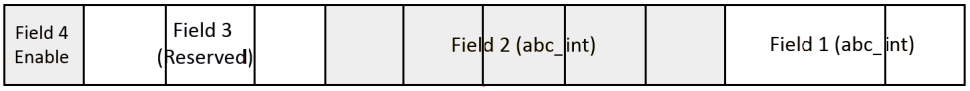


Figure 3.1: Fields in Register

Each register has it's own attributes, for instance address offset, name, description and size. There can be more complex attributes for each register, like key and lock value, mirror or shadow registers and hierarchical path to register etc. We will see them in followed sections. Apart from all these attributes each register or field may have some user defined attributes. Further we can block, it's a group of registers. Block also contains it's own attributes, like base address, disc and name. Table 3.1 groups various attributes related to register, field and block.

Table 3.1: Register, Field and Block basic attributes

Field Attributes	**Register Attributes**	**Reg. Block Attributes**
Description	Description	Description
Name	Name	Name
Starting bit	Offset Address	Base Address
Size	Reset Value	-
Access	Width	-

Register specifications are maintained by RTL team, in form of .xls sheet. Which then used to write register description in SystemRDL language, generating .rdl file. We will see more about SystemRDL in next section

3.2 SystemRDL Overview

SystemRDL is industry de-facto standard, by SPIRIT consortium and maintained by Accellera. It was specifically developed describe and implement a various control status registers. The register description then can be used for generating various register description views for specifications, h/w design, s/w development and documentation. Which can then help synchronizing verification, design and s/w teams.

Information about the registers in a circuit design is required throughout its lifetime, from initial architectural specification, through creation of an HDL description, verification of the design, post-silicon testing, to deployment of the circuit.Register specification must be accurate and in sync. As the register specified by the architects and the registers used by user for programming must be same. SystemRDL descriptions are used as inputs to software tools that generate circuit logic, test programs, printed documentation, and other register artifacts. Generating all of these from a single source ensures their consistency and accuracy. The description of a register may correspond to a register in an preexisting circuit design, or it can serve as an input to a synthesis tool that creates the register logic and access interfaces.

For verification register description in systemRDL language must be converted to language which can be used to read register description. SystemRDL descriptions are used as inputs to software tools to generate SV classes. Next chapter will describe systemRDL in detail.

Most engineers use third part tools, those converts .rdl into desired format instead of SV classes. Aim is to get register description into format that is easy to integrate into working verification environment. Usually output will be classes extended from predefined base classes. Base classes are nothing but derived classes, extended from OVM or UVM components.These derived classes are part of RAL model.

3.3 Register Model Use Flow

Register model can be used as tool for implementing control registers and memory in verification environment. Register model is an application package. Used to automatically create object oriented abstract model for control registers and memories inside design. RAL model has predefined tests and may also have functional coverage model to make sure that every bit of all registers has been exercised.

Register model is completely based on concept of register abstraction layer(RAL). RAL can support automation of register verification in complex designs, with large register sets. With RAL we can generate verification components automatically, which improves verification productivity. It also provides tests, assertions, backdoor paths and coverage, from high level specs. Components need to be generated once only and then we can compile them with other components. Then we can verify front-door access to all registers and verify functionality. Also provides fast backdoor register access which can be used to speed up verification, once basic tests are passed.

The Register Abstraction Layer (RAL) file is description of registers accessible to host in form of System Verilog classes. That is RAL files are set of system-Verilog classes, that represents description of registers. Set of RAL files describing registers, sometimes is referred as **“register map”** or **“model”**.

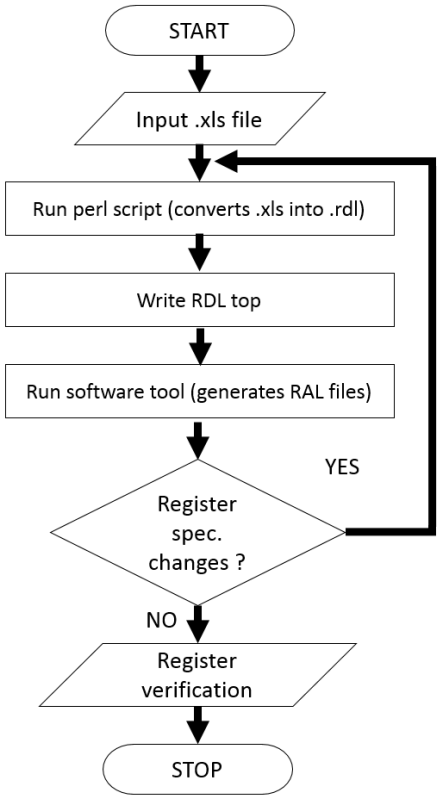


Figure 3.2: Steps Involved In Using Register Model

Flow chart 3.2 above shows register model use flow. Whenever there is any change in specification user don't have to make changes to model manually, instead user has to rerun script and software tool and register model get automatically updated. In verification environment register's are accessed with there names using RAL predefined methods, so environment also get updated with RAL file data implicitly. In next chapter we will see more about register model in detail.

3.4 Register Model Integration Overview

To be able to use register model, one must have an adaptor class available for the bus agent that is going to be used to interact with the DUT bus interface. If adaptor class is available, then the register model object needs to be constructed and a handle needs to be passed around the test bench environment using either the configuration and/or the resource mechanism. To perform read and write on to physical register transaction need to be generated on to bus. Which is done by bus agent, so there has to be bus agent which can send transaction to bus driver. Note

that information in RAL files is not in form of transaction packet, which need to be generated in order to access physical register. Adapter class then used to interpret RAL file data and generate sequence item that can be send to bus driver.

There can be more that one masters trying to access physical register. The register model is kept updated with the current hardware register state via the bus agent monitor, and a predictor component is used to convert bus agent analysis transactions into updates of the register model. So updater is another object, which converts observed bus transactions into RAL file updates. Following table (3.2) groups basic constructs required for integration of register model.

Table 3.2: Basic Constructs Required For Register Model Integration

Construct	**Description**
Register Model	Register Map in form of SV classes
Adapter	SV class, that can interpret register map and call sequence with bus compatible sequence item.
Bus Agent	SV class, that can send interpreted sequence item to driver
Predictor	SV class that observes bus agent's transactions and interpret transaction to update RAL update

Above table (3.2) groups basic constructs required for integration of register model. We will see more about register model integration in Implementation chapter. For SV class based implementation and more about agent, monitor, sequencer and other OVM concepts one can ref [1] and for more vedios on OVM or UVM refer Verification Academy [2]

Chapter 4

SystemRDL

Register model is description of all registers in design. We saw in earlier chapter that we use systemRDL language to write register model. In this chapter we will discuss details of register model and systemRDL.

4.1 SystemRDL

We have already seen register specification and it's importance. We have seen that each register block, register and field has it's own attributes.SystemRDL is a language for the design and delivery of intellectual property (IP) products used in designs. SystemRDL semantics supports the entire life-cycle of registers from specification, model generation, and design verification to maintenance and documentation. Registers are not just limited to traditional configuration registers, but can also refer to register arrays and memories.SystemRDL is designed to increase productivity, quality, and reuse during the design and development of complex digital systems. It can be used to share IP within and between groups, companies, and consortium's. This is accomplished by specifying a single source for the register description from which all views can be automatically generated, which ensures consistency between multiple views. A view is any output generated from the SystemRDL description, e.g., RTL code or documentation like .html.xml.

SystemRDL was created to minimize problems encountered in describing and managing registers. Typically in a traditional environment the system architect or hardware designercreates a functional specification of the registers in a design. This functional specification is most often text and lacks any formal syntactic or semantic rules. This specification is then used by other members of the team including software, hardware, and design verification. Each of these parties uses the specification to create representations of the data in the languages which they use in their aspect of the chip development process. These languages typically include

Verilog, VHDL, C, C++, Vera, e, and SystemVerilog. Once the engineering team has an implementation in a HDL and some structures for design verification, then design verification and software development can begin. During these verification and validation processes, bugs are often encountered which require the original register specification to change. When these changes occur, all the downstream views of this data have to be updated accordingly. This process is typically repeated numerous times during chip development. In addition to the normal debug cycle, there are two additional aspects thatcan cause changes to the register specification. First, marketing requirements can change, which require changes to a registers specification. Second, physical aspects, such as area and timing constraints can drive changes to the registers specification. There are clearly a number of challenges with this approach:

1. The same information is being replicated in many locations by many individuals.
2. Propagating the changes to downstream customers is tedious, time-consuming, and error-prone.
3. Documentation updates are often postponed until late in the development cycle due to pressures to

complete other more critical engineering items at hand. These challenges often result in a low-quality productand wasted time due to having incompatible register views. SystemRDL was designed to eliminate these problems by defining a rich language that can formally describe register specifications. Through application of SystemRDL and a SystemRDL compiler, users can save time and eliminate errors by using a single source of specification and automatically generating any needed downstream views.

Syntax for systemRDl is much resembling to verilog. This report skips discussion on lexical conventions in SRDL like white spaces, comment, identifiers, strings and numbers; however one can refer SystemRDL v1.0 manual by SPIRIT Consortium[8]. Separating SRDL from register model is very hard, since SRDL is standard for writing register model. So in next section we start with register model, during which SRDL symantics are covered. Before we start with SRDL and register model let's see basic cocepts of SRDL.

4.1.1 Key Concepts in SRDL

1. **component:** A basic building block in SystemRDL that acts as a container for information. Similar to a structur class in programming languages.
2. **property:** A characteristic, attribute, or a trait of a component in SystemRDL.

3. **field:** The most basic componentobject. Fields serve as an abstraction of hardware storage elements.

4. **register:** A set of one or more fields which are accessible by softwareat a particular address.

5. **register file:** A grouping of registers and other register files. Registerfiles can be organized hierarchically.

6. **address map:** Defines the organization of the registers, register files, and address maps into a software addressable space. Address maps can be organized hierarchically.

4.2 Register Model

This subclause describes the key concepts of SystemRDL and documents general rules about how to use the language to define hardware specifications. Subsequent clauses contain overview about working with each of the individual components in SystemRDL.

A componentin SystemRDL is the basic building block or a container which contains properties that further describe the components behavior. There are four structural components in SystemRDL: field, reg, regfile, and addrmap. Additionally, there are two non-structural components: signal and enum.Components can be defined in any order, as long as each component is defined before it is instantiated. All structural components (and signals) need to be instantiated before being generated.

4.2.1 Defining Components

To define components in SystemRDL, each definition statement shall begin with the keyword corresponding to the component object being defined. All components need to be defined before they can be instantiated. Various component types and keywords in systemRDL are listed in table4.1.

SRDL components can be defined in two ways: definitively or anonymously.

- Definitive defines a named component type, which is instantiated in a separate statement. The definitive definition is suitable for reuse.

- Anonymous defines an unnamed component type, which is instantiated in the same statement. The anonymous definition is suitable for components that are used once.

Table 4.1: Component Types

Component	Keyword
Field	field
Register	reg
Register File	regfile
Address Map	addrmap
Signal	signal
Enumeration	enum

A definitive definition of a component appears as follows.
`component type_name [property;]*;`
An anonymous definition (and instantiation) of a component appears as follows.
`component [property;]* instance_name;`

And each property is specified as a `name=value` pair, e.g., `name=foo`.
The component definition body (specified within the curly braces) is comprised of one or more of following

1. Default property assignments
2. Property assignments
3. Component instantiations
4. Nested component definitions

Example:

- Definitive field component definition for myField
  `field myField ;`
- anonymous field component definition for myField
  `field  myField;`

Here, in definitive field definition “myfield”is user defined type name of component, which can be instantiated multiple times with any instance name. While in anonymous definition “myfield”is instance name.

4.2.2 Instantiating Components

Similar to defining components, SRDL components can be instantiated in two ways definative and anonymous.

1. A definitively defined component is instantiated in a separate statement, as follows.
   `type_name instance_name [[number] | [number : number]];`
   where,
   (a) type_name is the user-specified name for the component.
   (b) instance_name is the user-specified name for instantiation of the component.
   (c) number is a simple decimal or hexadecimal number.
   - `[number]` specifies the size of the instantiated component array.
   - `[number : number]` specifies the specific indices of the array. This form of instantiation can only be used for field or signal components
2. An anonymously defined component is instantiated in the statement that defines it.

Components need to be defined before they can be instantiated. In some cases, the order of instantiation impacts the structural implementation, e.g., for the assigning of bit positions of fields in registers. Also in case of defining regfiles in addrmap order of register impacts the structural implementation.

4.2.3 Component Properties

Component properties define the specific function and purpose of a component, as well as its interaction with other instantiated components. Property types include boolean, string, numeric, sizedNumeric,unsizedNumeric, accessType (enum), addressingType (enum), precedenceType (enum).

Each component type has its own set of pre-defined properties. Properties may be assigned in any order. User-defined properties can also be specified to add additional properties to a component that are not predefined by the SystemRDL specification. A specific property shall only be set once per scope. All component property assignments are optional. A property assignment appears as follows.
`property_name [= value];`
When value is not specified, it is presumed the property_name is of type boolean and set to true.
Example:

```
0    field myField {
1              rclr; // Bool property assign, set implicitly to true
2              woset = false; // Bool property assign, set explicitly
                                // to false
3              name = my field; // string property assignment
4              sw = rw; // accessType property assignment
6              };
```

4.2.4 Assigning Default Values

Default values for a given property can be set within the current or any parent scope. Any components defined in the same or lower scope as the default property assignment shall use the default values for properties in the component not explicitly assigned in a component definition. A specific property default value shall only be set once per scope.A default property assignment appears as follows.
`default property_name [= value];`
When value is not specified, it is presumed the property_name is of type boolean and the default value is set to true. The descriptions for the types of values that are legal for each property_name.
Example:

```
0    reg {
1              default name = ‘‘default name’’;
2              field {} f1; // assumes the name ‘‘default name’’from
                              //above
3              field { name = ‘‘new name’’} f2; // name assignment
                              // overrides ‘‘default name’’
4         } some_reg;
```

4.2.5 Dynamic Assignment

Some properties may have their values assigned or overridden on a per-instance basis. When a property is assigned after the component is instantiated, the assignment itself is referred to as a dynamic assignment. Properties of a referenced instance shall be accessed via the arrow operator (-¿). A dynamic assignment appears as follows.
`instance_name -> property_name [= value];`
In the case where instance_name is an array and if the component type is field or signal, the fact the component is an array does not matter, the assignment is treated

as if the component were a not an arrray. While if the component is an array and the component type is reg, regfile, or addrmap The user can dynamically assign the property for all elements of the array by eliminating the square brackets ([ ]) and the array index from the dynamic assignment.
`array_instance_name -> property_name [= value];`
Or user can dynamically assign the property for an individual index of the array by using square brackets ([ ]) and specifying the index to be assigned within the square brackets.
`array_instance_name [index] -> property_name [= value];`

Example 1:
This example assigns a simple scalar.

```
0     reg {
1          field {} f1;
2          f1->name = ‘‘New name for Field 1’’;
3          } some_reg;
```

Example 2:
This example assigns an array.

```
0     reg {
1          field {} f1;
2          f1->name = ‘‘New name for Field 1’’;
3          } some_reg[8];
4          some_reg->name = ‘‘This value is applied to all elements in
                                 // the array’’;
5          some_reg[3]->name = ‘‘This value is only applied to the 4th
                                 // item in the array of 8’’;
```

4.2.6 Property Assignment Precedence

There are several ways to set values on properties. The precedence for resolving them is (from highest to lowest priority):

1. dynamic assignment

2. property assignment
3. default property assignment
4. SystemRDL default value for property type

4.2.7 Scoping

SystemRDL is a statically scoped language, where the root scope is the outermost scope. The body of any defined component is its own scope. All component names within a given scope shall be unique. All instance names within a given scope shall be unique. However, there can be a component and instance with the same name in the same scope.

The only component definitions visible at any scope shall be those defined in the current scope and any parent scope, up to and including the root scope. To resolve a component name, SystemRDL searches from the current scope to the outer scope until it finds the first matching component name.

The root scope shall only contain component definitions and signal instantiations. No other component instantiations shall be allowed in the root scope. Therefore, all component instantiations shall occur within an addrmap component definition. The roots of an addrmap hierarchy are those addrmaps that are defined, but not subsequently instantiated. Only instances instantiated in the current scope can be referenced within that scope. A child instance can be referenced via the dot operator (.). A instance reference appears as follows.

```
instance_name [.  child_instance_name]*
```

Dynamic assignments that are specified at an outer scope override those that are specified at an inner scope. No more than one assignment of a property per scope is allowed in SystemRDL.

4.2.8 Universal Properties

The **name** and **desc** properties can be used to add descriptive information to the SystemRDL code. The use of these properties encourages creating descriptions that help generate rich documentation. All components have a instance name already specified in SystemRDL; name can provide a more descriptive name and desc can specify detailed documentation for that component. Universal properties are listed as follows in table(4.2). Where, Dynamic indicates whether a property can be assigned dynamically.

Table 4.2: Universal Properties

Property	ImplementationApplication	Type	Dynamic
Name	Specifies a more descriptive name (for documentation purposes).	string	Yes
Desc	Describes the components purpose.	string	Yes

4.3 Example with systemRDL

Let's take example that will explain register model and how to use systemRDL language to create it.

Example 1 Let's assume we have IP named ACV which has instances of subIP's. Consider ACV has single instance of I2C and two instances SPI each. Now register specifications of ACV will have registers of SPI, I2C and it's own configuration and controlsignal registers. Since SPI has 2 instances SPI_0 and SPI_1,we will have same registers but with different base address. Following tables (A.1, A.2 and A.3)summarizes the register memory map for IP ACV. For Register Field Description refer Appendix[A].

Now, we want to build a single register model for our IP, that will have disctiption of all registers of IP and it's subIP's. To start with we need SRDL for each subIP as well as for ACV, since root RDL can't have register and addrmap in one scope. i.e. root RDL or top addrmap can have only addrmap's which are not instantiated anywhere and dynamic assignments in it. Let's write SRDL for ACV first.

SRDL for ACV: RDL for ACV can be written as follows,

```
addrmap ACV_reg_file
{
    name="ACV_reg_file";
    desc=" This is register file/block, contains
           description of config or control/status
           registers of ACV. We can write register
           defination in saperate file and then
           include that file in register discription
           using 'include OR we can write everything
           in one single file as we are doing here
           since complexity is lees and it's more
           redable in one page and better for
           understanding.";
```

```
    reg acv_ctrl_0_def  //reg defination
    {
        name="acv_ctrl_0_reg";
        desc="Controle Register";
        regwidth=32;     //type numeric
        shared=FALSE;    //type boolean
        //anonymously defining fields since not reused
        field
        {
            name="SCPOL";
            desc="";
            hw=R/W; //hardware access ability
            sw=R/W; //software access ability
        }SCPOL[31:21];

        field
        {
            name="FRF";
            desc="";
            hw=R/W; //hardware access ability
            sw=R/W; //software access ability
        }FRF[20:0];
    };

    reg acv_RSVD_0_def
    {
        name="acv_RSVD_0";
        desc="";
        regwidth=32;     //type numeric
        shared=FALSE;    //type boolean
        field
        {
            name="RSVD1";
            desc="";
            hw=R/W; //hardware access ability
            sw=R/W; //software access ability
        }RSVD1[31:0];
    };
            //instantiating registers
    acv_ctrl_0_def acv_ctrl_0 @ 0x0000 ;
            //use @ for static address offset assignment
    acv_RSVD_0_def acv_RSVD_0 @ 0x0004 ;
};
```

Similarly, we can write SRDL code for all other subIP's. For SPI we can instantiate I2C_reg_file twice; each with different base address. No need to write same RDL twice. We also need to write one intermediate addrmap for SPI. where we can instantiate register blocks, and then instantiate address map in top address map. SRDL for all subIP's and top SRDL code is given in appendixB.

So we saw systemRDL basics and how to use systemRDL for creating register model. We have already discussed in brief about how to use register model for register verification in previous chapters. We will see how we integrated register model in environment and used it for register verification.

Chapter 5

Implimentation

In this chapter we will discus, about our implementation of RAL for register verification. Through which one can expect to understand flow of integration, and components that are required for integration as well as how register access happens for verification perpose.

Table 3.2 summarises basic components required for register verification using RAL framework/ model. Let's see discuss on each component about it's role and implementation in verification environment. We saw register model in previous chapter in brief. Starting with adapter.

5.1 Adapter

The register model access methods generate bus read and write cycles using generic register transactions. These transactions need to be adapted to the target bus sequence_item. The adapter needs to be bidirectional in order to convert register transaction requests to bus sequence items, and to be able to convert bus sequence item responses back to bus sequence items. The adapter should be implemented by extending it from one of specific base class from RAl model, for UVM RAL model it must be extended from uvm_reg_adapter base class. Fig.(5.1) shows existence of adapter in register access flow.

Adapter in this project implementation is is unidirectional and it converts register description in RAL file into bus agent transaction, when register model access method is called. In this project adapter checks for user defined attribute "access_path" which represents set of ovm(in this project) components, which are involved in current transaction. As environment maintains list of components in correlation with key which is access_type here. Based on the value of access_path we select which agent to be used.

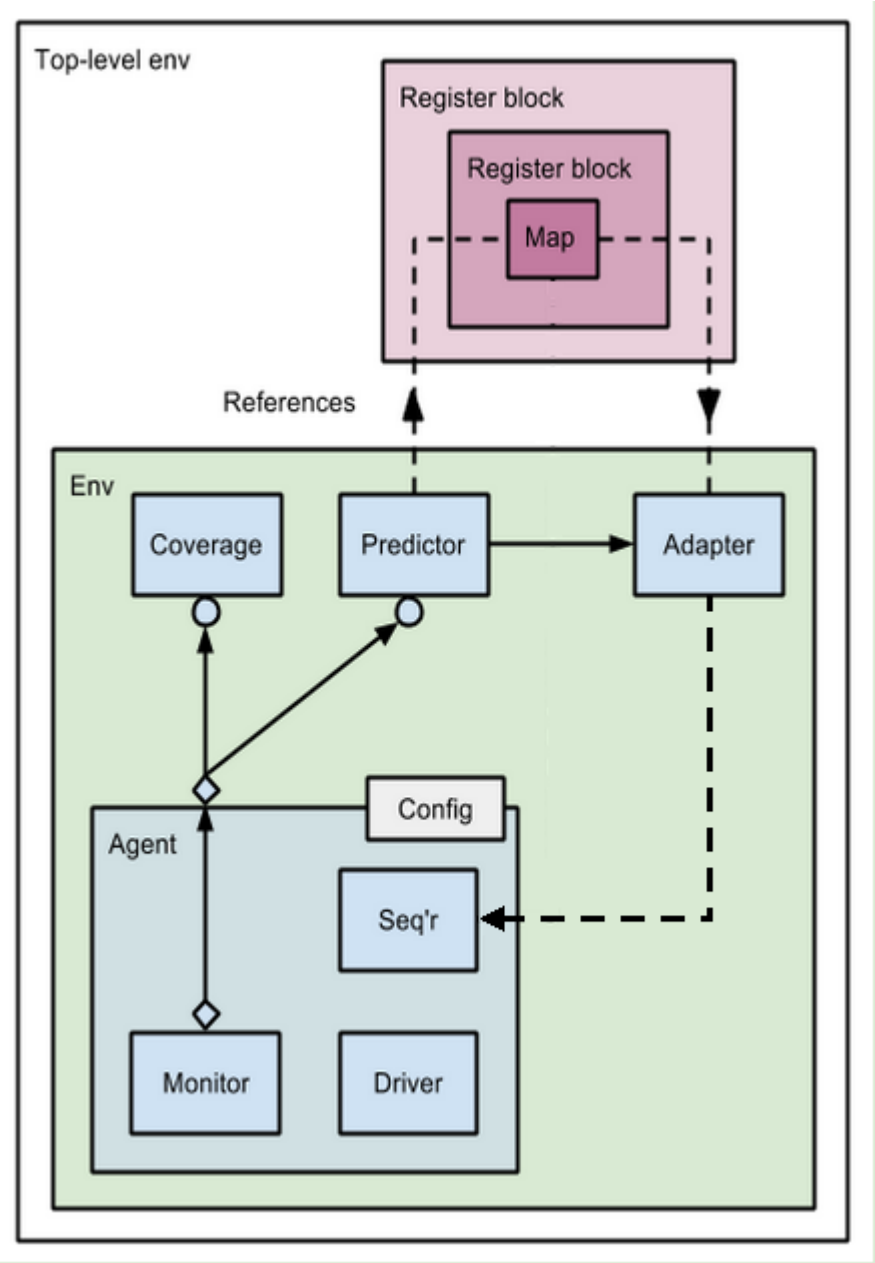


Figure 5.1: Register Abstraction Layer [2]

So, basically adapter will be called by UVM/OVM internal method for reg access whenever there is call to register model's method ex. read, write or read_and_check etc. Upon call to adapter will read register name which is to be accessed and looks for existence of particular register in regs queue. regs queue is simply an associative array which is maintained by RAL model internally in base class, which has all register in it. Where register name is key and value is pointer to particular regis-

ter class object. This queues are generated dynamically in build phase of ovm(in this project). Then adapter calls extended ral class method to get address of register. Once it has and address it calls OVM/UVM api 'ovm_do_with, and constrains bus_agent's transaction package to desired value. And hence data gets converted to bus transaction packet. This can be done in many other ways based on programmers logic and project needs.

5.2 Bus Agent

Bus agent is just like any other OVM(in this project) agent, mainly it has sequencer, driver and monitor. In this case since this agent components are build to communicate with bus it's called as bus agent. bus agent's driver must be capable of communicating to DUT, so it generate pin level transactions to communicate with DUT. In this project we call an agent which intern calls an agent which calls bus driver. Since there are other operations to be performed before writing on to bus.

For simplicity of understanding let's assume we have only single agent. In previous section we saw adapter calls Bus_sequence with constraints. Bus_equence then sends transaction packet, while bus_sequencer pass modified transaction to bus_driver driver writes/reads on/from bus. If transaction is read transaction then once transaction is complete, read value is returned via calling function, in test where we have called read api of UVM/OVM RAL model.

5.3 Predictor

RAL model internally maintains mirrored value for each register. which is updated on each transaction to register. This update of mirror value can be done manually by calling update and predict methods along with read and write operations respectively or, it can be updated manually using monitor class generated by same software tool which is used to convert systemRDL into RAL files. This monitor keeps track of bus_agents transactions and on each transaction it updates mirror values accordingly.

It based on project which method to choose either manual or automatic update. If project has more than one masters who can access register mutually. Then in this case RAL will not be aware of any changes or transactions happened by other master on registers, and register verification will fail. In such case it's recommended to use monitor based mirror value update. But if project has only one master, then manual update can be used. For predictor we can simplly created method in RAL base class.

5.4 Front Door and Backdoor Access

There are two ways of accessing design registers in a verification environment: front door and back door. Front door is by using the design register bus. This consumes cycles and follows the register bus protocol. Backdoor is a zero simulation time access by mapping to the design register directly using the HDL path and allows quick configuration of registers. Thus, configuring by backdoor saves simulation time, especially useful for full-chip and sub-chip simulations where several registers need to be setup.

Backdoor access also helps in uncovering address decode design bugs. Since the mirror register gets updated while doing front-door or back-door writes, writing a register in front door and reading it from the backdoor flags any mismatch between design and the mirror register.

5.5 Test and Test sequence

In test we can override environment configuration in build phase, to setup environment for register verification. Along with that we will call user sequence in test run phase, which is our test sequence. In test sequence we will use RAL base class methods to access register in sequence body. The simplest flow for register verification can be think as read, write and then read and compare with previous value in register to verify attributes or reset value. Here attributes are register access attributes, like r/w or r/o etc.

Tests in this project are be made to perform default value check and register access attribute check. For functionality check we have different tests.

5.6 Considering Corner Cases

We seen simple sequence for register verification, usually it won't be the case. Since there can be many register hows functionality will be different that usual.

5.6.1 Lock Bit

There can be register who's one or more bit's may depend on bit/bit's of other register. Bit's other register on which attribute of bit depends are called lock bits. For lock bits we can specify, an user defined attribute and then implementing methods in base class to handle the scenario. This is one time effort and reduces time and efforts required later in test development stage.

Or we can take them as corner case and address them separately. Here in this project we have implemented methods in base class to take care of lock bit's and used lock as user defined attribute for each filed.

5.6.2 Write Only Register

Write only register can't be verified using front door sequences, since we can't read them as it will always return zero. In this project we have used backdoor access to verify such registers. To use backdoor access we need to define hierarchical path to register in RDL. And at time of erification we can write oin register using front door sequence and read using backdoor sequence.

Back door sequences also reduce verification time since it avoids important system clock cycles involved in bus transaction.

5.6.3 Special Bits

Bit's like reset or enable bit can't be verified since on reset design will start initializing again, similarly in case of enable bit if we disable it subIP or module or block of design will be disabled. We need to avoide writing on such bit's in test sequence.

Apart from corner cases discussed, there can be other corner cases. Like there can be sequence which writes on to register during design initialization process.

5.7 Implementation Algorithms

Test Algorithm

Configuration of environment is optional here, it is completely need based. But mostly there will be one or other component or sequence in environment which need to be disabled for scope of test.

1. START
2. Configure Environment
3. Call register verification's user test sequence
4. STOP

Test Sequence Algorithm

To reduce maintenance of register verification, we will be using base class methods to get pointer to reg_file or register by it's name. So even if there is any change, changes will be automatically reflected in environment after regeneration of RAL files.

We need to update RAL before we do write in attribute check, and update method does read operation on register. Since we are performing read before write then we can do reset value check in same test before attribute check. This method will reduce one extra read operation. But care must be taken for R/W1C attribute registers, which are read and write to clear i.e. write 1 to clear.

We can also perform all register check in single file but this is not recommended, as we discussed in sec(2.3.3). So we have implemented separate test sequence for each subIP. And there are such 15 subIP's in this project.

1. START
2. Get top RAL class pointer
3. Get required file pointer
4. Get Register pointer
   ++++++++++++ Reset Value Check Started ++++++++++++
5. Read register value and Check
   ++++++++++++ Reset Value Check Done ++++++++++++
   ++++++++++++ Attribute Check Started ++++++++++++
6. Update RAL
7. Get write data
8. write register
9. read and check register val
   ++++++++++++ Attribute Check Done ++++++++++++
10. Repeat step 4 to 9 for each register in file
11. Get next file pointer and repeat step 4 to 10
12. STOP

RAL access sequence Algorithm

RAL access sequence is smiler to adaptor in UVM RAL methodology. It takes register name, write data and operation op-code as parameters. Then in step 1 it calls user defined get_addr function. Since address format and address translation strategy night be different from register to register, this function is user defined so one can design address translation according to project. Once we get address, this sequence will choose op-code to constrain with bus_seq based of access_path value and other user defined attributes passed for readwrite function call. Then it executes sequence with inferred constrain on registered sequencer. Agent does all read/write on bus and returns read data, if it is read operation.

1. START
2. get register address
3. get command or op-code
4. initiate sequence with data, address and command constraints
5. STOP

Chapter 6

Simulation and Results

6.1 Simulation

This chapter describes simulation of test for register verification. Also we will discuss and compare result of simulation of register verification test using C SV handshake with processor as RTL and using RAL without processor (that is we use BFM instead of processor). For simulation and debugging we use Synopsiss VCS tool. For debugging and coverage VCS provides various features like DVE and URG etc.

For simulation setup, since we are using UNIX HDL simulation front-end environment for project specific customization and creating compilation and simulation flow. We have created compilation and simulation flow where we define parameters needed for compilation and simulation with register model and BFM, included flow specific include directories and file and other customization. we also need to modify RTL to exclude processor and connect BFM, we have used pre-processor directives to keep integrity of other compilation and simulation flows intact.

6.2 Results

Register model is readily available by third party vendors. With use of register model its just one time effort to integrate register model and setup the environment. But it gives tremendous benefits, as it helps reduce verification time and efforts, other benefits include;

- Reduced maintenance cost:
  During development cycle there can be change in base address or register offset due to addition or removal of feature or module as part of bug fix or change in specification. For such changes, we will have to make change at every place in environment where we are accessing register by its address. But with register model we only need to update RDL files and re-generate RAL files, provided we are using standard APIs provided by register model for accessing register by its name. As described in section 2.3.3, weeks of time to maintain and add new tests has been reduced into hours. During project time to develop new test was approx. 90 min., if all corner cases are already known.

- Ease of use:
  To access registers of any nodule, one dont have to know about agent to be used and all other deep stuff. Instead one can use API for read write to register buy its name, address or type by providing all required arguments.

- Reuse:
  RDL and RAL files are reusable at SOC level. And since register model is same throughout project APIs for register access are same, which reduces efforts of test developer. Again developer doesnt need to know which agent to be used and how APIs are implemented. Although register access sequences and RAL integration to local IP environment is not reusable at SOC. Since IP goes as a RTL into SOC, and SOC environment will use standard host to slave protocol to communicate with IP.

- Scalability:
  Test sequences with use of register model are scalable and can accommodate addition/removal of register automatically as per updated RAL file. The resone why we call implementation as automated register verification, cause for simple addition or removal of register there is no need to modify test sequence except to regenerate RAL files. Test sequence need to be modified only in scenario of corner cases.

- Better control over test scenario:
  With register model and BFM we avoid use of RTL of processor. With processor we would have to write C program where we will access registers, but accessing register in particular scenario we need to have some communication between C and SV domain. Special scenarios may include initialization of IP, entering to some power down mode etc. Simplest way is to communicate by read and write to register which is time consuming. On other hand with register model and BFM (instead of processor), complete test can be written in SV. Which provides better control as more SV and OVM constructs are available for test development.

- Improvement in simulation time:
  As processor consumes significant cycles for stuff other than register access, like reading data from ROM/SRAM, servicing interrupts etc. With use of BFM we can save those cycles during test simulation. With the use of register model and BFM time for test simulation has been reduced by 21.46% for test used to generate results, also improvement in simulation time depends on scenario to be exercised and no C to SV domain handshakes involved.

  Fig.6.1 shows simulation wave form with C test and processor, while fig.6.2 shows simulation wave form with register model and BFM for register verification of same module.

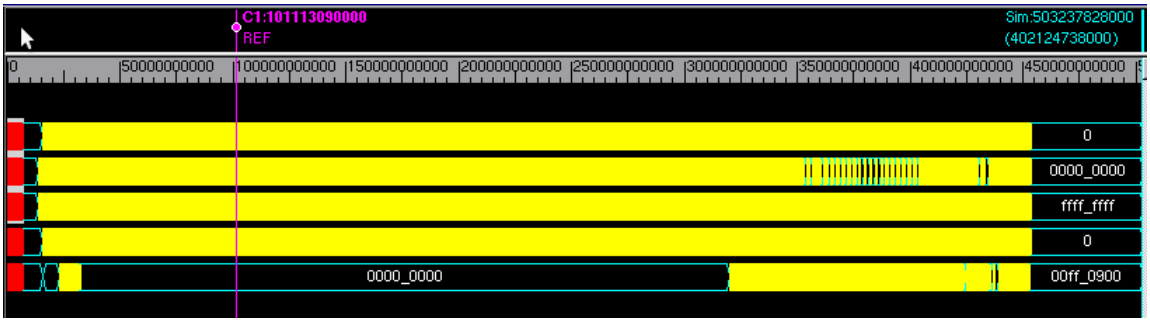


Figure 6.1: Simulation waveform for C based register verification test

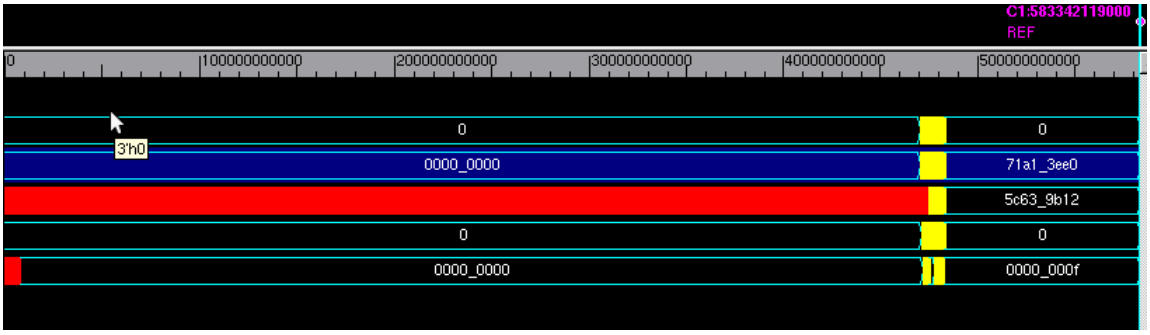


Figure 6.2: Simulation waveform for SV test or RAL based register verification test

Fig.6.3 and Fig.6.4 shows snippet from simulation logs of both tests, with CPU time taken for simulation of two tests for same module.

Time for C test execution = 1160 sec.
Time for SV or RAL based test execution = 911 sec.
Percentage of time saving = $(((1160911)/1160) * 100)\% = 21.4\%$

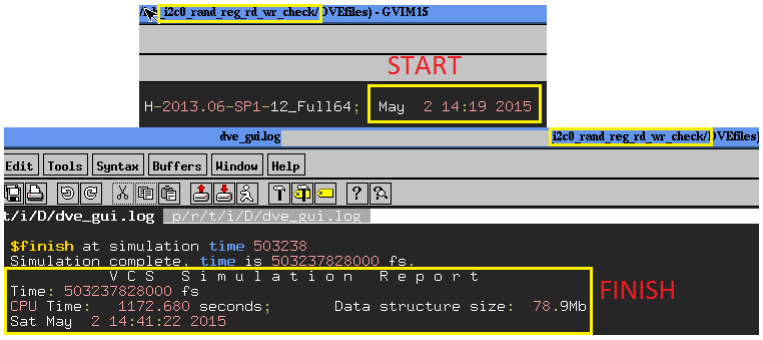


Figure 6.3: CPU time taken for C based register verification test execution

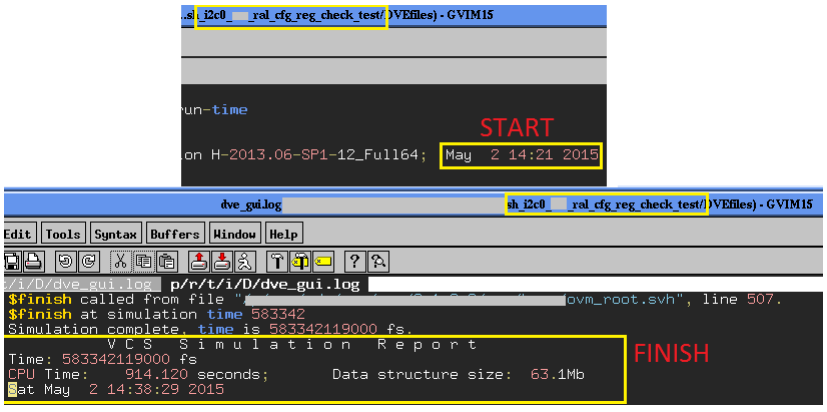


Figure 6.4: CPU time taken for SV test or RAL based register verification test execution

Chapter 7

Conclusion and Future Scope

7.1 Conclusion

Use of register model eliminates error prone and time consuming process, and it provides an almost zero-time, low maintenance, and reusable register design and verification system. Allows design, verification and firmware teams to work more efficiently from consistent and synchronized views of the chip design.

Flow presented in thesis is repeatable and can be used across various blocks, SOC and chip level. The flow with register model provides lot of saving of efforts, enhanced productivity, firm and stable design with better verification. As seen earlier, greater than a months effort can be cut down to less than a weeks by this approach, provides significant improvement in test execution time(about 21.46%).

Use of RAL methodology can successfully reduce register verification time from month to week duration with implementation of RAL. Which will help reduce time to market the product and improves the IP quality, by adding an important feature to IP.

7.2 Future Scope

- Backdoor Access:
  Backdoor access provides fast and easy access to registers. Once basic tests are passed, that is once we are sure of register front door access is working correctly we can use backdoor access everywhere else. This methodology helps reduces execution time, in further process of verification.
- Monitor:

There can be more than one agents in IP, trying to access register; register may get modified by hardware; set value by strap or can be forced with some value. Monitor can be developed with backdoor accesses to keep mirror copy of register in sync with actual register value. This also helps to cover more range of register attributes for example: W/O (write only), as with W/O we cant read back value we have written, so in order to verify W/O we need to read it via backdoor.

- Coverage:
  Current version of implementation of RAL model doesnt support coverage and checkers. Register toggle coverage can be good measure to determine an amount of block verification, as most of activities turn down to register level bit changes. So no of bits toggled collectively in functional verification tests, can predict how much functionality of block has been verified by tests.

Bibliography

[1] Verification Academy. *OVM Cook Book*. 2014.

[2] Verification Academy. *UVM Cook Book*. 2014.

[3] Silpa Naidu Ballori Banerjee Subashini Rajan. "Automated approach to Register Design and Verification of complex SOC". In: DVCon, 2011.

[4] Mentor Graphics. *Verification Academy*. URL: `https://verificationacademy.com/`.

[5] Gopikrishna Naresh Maddipati. URL: `http://www.testbench.in/`.

[6] Kunal Shah. "Verification Of Configuration Registers: Don't Take It Easy!" In: Incisive Verification Article.

[7] Deepak Kumar Tala. URL: `http://www.asic-world.com/`.

[8] Register Description Working Group of The SPIRIT Consortium. *SystemRDL v1.0: A specification for a Register Description Language*. 2009.

Appendix A

Register Description

Appendix contains register discription for example 1, explained in chapter (4).

ACV

Register Block Name: acv_reg_blk
Base address: 0x0000

Table A.1: ACV Register Memory Map

Name	Address Offset	Width	Description
acv_ctrl_0	0x0000	32 bits	Control register 0 **Reset Val:** 0x00
acv_RSVD_0	0x0004	32 bits	Reserved Register Reserved location for future use, write has no effect, read returns 0 **Reset val:** 0x00

SPI

Register Block Name: s0_reg_blk
Base address: 0x0100

Register Block Name: s1_reg_blk
Base address: 0x0200

Table A.2: SPI Register Memory Map

Name	Address Offset	Width	Description
CTRLR0	0x0000	32 bits	Control register 0 **Reset Val:** 0x00
SSIENR	0x0004	1 bits	SSI Enable Register **Reset val:** 0x00

I2C

Register Block Name: i2c_reg_blk
Base address: 0x0500

Table A.3: I2C Register Memory Map

Name	Address Offset	Width	Description
SR	0x0000	8 bits	Status register 0 **Reset Val:** 0x06
IMR	0x0004	4 bits	Interrupt Mask Register **Reset val:** 0x03

ACV Register Discription

acv_ctrl_o

Name: ACV Control Register 0
Size: 32bits
Address Offset: 0x0000
ReadWrite Access: R/W
Reset Val: 0x0000

Table A.4: acv_ctrl_0 Register Field Description

bits	Name	Access	Description
31:21	SCPOL	R/W	-
20:0	FRF	R/W	-

acv_RSVD_0

Name: ACV Reserved Register 0
Size: 32bits
Address Offset: 0x0004
ReadWrite Access: R/w
Reset Val: 0x0000

Table A.5: acv_RSVD_0 Register Field Description

bits	Name	Access	Description
31:0	RSVD1	R/W	Write has no effect, Read returns 0

SPI Register Discription

spi_ctrl_o

Name: SPI Control Register 0
Size: 32bits
Address Offset: 0x0000
ReadWrite Access: R/w
Reset Val: 0x0000

Table A.6: spi_ctrl_0 Register Field Description

bits	Name	Access	Description
31:21	sctrl_2	R/W	-
20:16	sctrl_1	R/O	-
15:0	sctrl_0	R/W	-

ssi_e_o

Name: SPI SSIENR 0
Size: 32bits
Address Offset: 0x0004
ReadWrite Access: R/w
Reset Val: 0x0000

Table A.7: spi_ssienr Register Field Description

bits	Name	Access	Description
31:1	reserved3	R/W	Reserved for future use
0:0	EN	R/W	Enable bit

I2C Register Discription

SR

Name: Status Register
Size: 32bits
Address Offset: 0x0000
Reset Val: 0x0006

Table A.8: SR Register Field Description

bits	Name	Access	Description
31:8	RSVD0	R/W	reserved for future use
7:0	inptr_sts	R/W	-

IMR

Name: Interupt Mask Register 0
Size: 32bits
Address Offset: 0x0004
ReadWrite Access: R/w
Reset Val: 0x0003

Table A.9: IMR Register Field Description

bits	Name	Access	Description
31:4	RSVD1	R/W	Write has no effect, Read returns 0
3	int_3	R/W	interrupt 3 mask bit
2	int_2	R/W	interrupt 2 mask bit
1	int_1	R/W	interrupt 1 mask bit
0	int_0	R/W	interrupt 0 mask bit

Appendix B

Example 1 Complete SRDL

B.1 RDL Top

ex_top.rdl

```
`include "lib_udp.rdl"
addrmap top_addrmap {
        `include "ex_i2c.rdl"
        `include "ex_spi.rdl"
        `include "ex_acv.rdl"
        name="top_addrmap";
        desc="tp";
        acv_reg_file ACV_reg_file_i@0x0000;
        spi_reg_file spi_reg_file_i_0@0x0100;   //0x100
        spi_reg_file spi_reg_file_i_1@0x0200;   //0x200
        I2C_reg_file I2C_reg_file@0x0500;
                     };
```

B.2 ACV Register File

ex_acv.rdl

```
       `include "lib_udp.rdl"
addrmap acv_reg_file{
        name="acv_reg_file";
        desc=" ";
```

```
reg acv_ctrl_0_def{  //reg defination
    name="acv_ctrl_0_reg";
    desc="Controle Register";
    regwidth=32;     //type numeric
//anonymously defining fields since not reused
    field{
        name="SCPOL";
        desc="";
        AccessType = "RW"; //hardware access ability
         //software access ability
    }SCPOL[31:21]=11'b0;

    field{
        name="FRF";
        desc="";
        AccessType = "RW"; //hardware access ability
         //software access ability
    }FRF[20:0]=21'b0;
};

reg acv_RSVD_0_def{
    name="acv_RSVD_0";
    desc="";
    regwidth=32;     //type numeric
    field{
        name="RSVD1";
        desc="";
        AccessType = "RW"; //hardware access ability
         //software access ability
    }RSVD1[31:0]=32'b0;
};
acv_ctrl_0_def acv_ctrl_0 @ 0x0000 ;
acv_RSVD_0_def acv_RSVD_0 @ 0x0004 ;
};
```

B.3 SPI Register File

ex_spi.rdl

```
  `include "lib_udp.rdl"
addrmap spi_reg_file{
```

```
name="spi_reg_file";
desc="  This is register file/block, contains
        description of config or control/status
        registers of SPI.";

reg CTRLR0_def { //reg defination
    name="acv_ctrl_0_reg";
    desc="Controle Register";
    regwidth=32;     //type numeric
//anonymously defining fields since not reused
    field{
        name="sctrl_2";
        desc="";
       AccessType = "RW"; //hardware access ability
    }sctrl_2[31:21]=11'b0;

    field{
        name="sctrl_1";
        desc="";
        AccessType = "RO"; //hardware access ability
    }sctrl_1[20:16]=5'b0;

    field{
        name="sctrl_1";
        desc="";
        AccessType = "RW";//hardware access ability
    }sctrl_0[15:0]=16'b0;
};

reg SSIENR_def{
    name="acv_RSVD_0";
    desc="";
    regwidth=32;     //type numeric
   field{
        name="reserved3";
        desc="";
        AccessType = "RW"; //hardware access ability
         }reserved3[31:1]=31'b0;

    field{
        name="EN";
        desc="";
        AccessType = "RW"; //hardware access ability
```

```
            }EN[0:0]=1'b0;
    };
            //instantiating registers
    CTRLR0_def CTRLR0@0x0000 ;
    //use @ for static address offset assignment
    SSIENR_def SSIENR@0x0004 ;
    };
```

B.4 I2C Register File

ex_i2c.rdl

```
`include "lib_udp.rdl"

addrmap I2C_reg_file{
    name="I2C_reg_file";
    desc="  This is register file/block, contains
            description of config or control/status
            registers of I2C. ";

    reg SR_def  //reg defination
    {
        name="SR_reg";
        desc="Controle Register";
        regwidth=32;     //type numeric
    //anonymously defining fields since not reused
        field
        {
            name="RSVD0";
            desc="";
           AccessType = "RW"; //hardware access ability
        }RSVD0[31:8]=24'b0;

        field
        {
            name="intr_sts";
            desc="";
           AccessType = "RW"; //hardware access ability
        }intr_sts[7:0]=8'b0110;
    };
```

```
reg IMR_def
{
    name="IMR";
    desc="";
    regwidth=32;     //type numeric
    field
    {
        name="RSVD1";
        desc="";
        AccessType = "RW"; //hardware access ability
    }RSVD1[31:4]=28'b0;
    field
    {
        name="int_3";
        desc="interrupt 3 mask bit";
        AccessType = "RW"; //hardware access ability
    }int_3[3:3]=1'b0;
    field
    {
        name="int_2";
        desc="interrupt 2 mask bit";
        AccessType = "RW"; //hardware access ability
    }int_2[2:2]=1'b0;
    field
    {
        name="int_1";
        desc="interrupt 1 mask bit";
        AccessType = "RW";//hardware access ability
    }int_1[1:1]=1'b1;
    field
    {
        name="int_0";
        desc="interrupt 0 mask bit";
        AccessType = "RW"; //hardware access ability
    }int_0[0:0]=1'b1;
    };
```

B.5 RAL Files

Output of software tool consist of SV class files, one for each RDL file and one for top rdl. Along with RAL files it also generates monitor, which can be used for updating RAL using predictor if one has specified backdoor path to all registers. In this example we haven't considered backdoor access. Out put is as follow:
my_exoutput:
outputovmI2C_reg_file_regs.svh
outputovmacv_reg_file_regs.svh
outputovmspi_reg_file_regs.svh
outputovmtop_addrmap_ral_env.svh
outputovmtop_addrmap_ral_rtl_monitor.svh

For environment, systemverilog and ovm related other resources, one can refer ([4], [5] and [7]).