Methodology for Design Verification

Major Project

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology
in
Electronics & Communication Engineering
(Embedded Systems)

By

Amreen D Charaniya
(12mece02)

Electronics & Communication Engineering Branch
Electrical Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382 481

May 2014

Methodology for Design Verification

Major Project

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology
in
Electronics & Communication Engineering
(Embedded Systems)

By

Amreen D Charaniya
(12mece02)

Under the guidance of

External Project Guide:

Vishal Jain
Sr. Engg Specialist(IBP Dept),
Mr. Abhishek Jain
Technical Manager(IBP Dept),
STMicroelectronics,
Greater Noida.

Internal Project Guide:

Dr. N.P.Gajjar
Sr. Associate Professor (EC Dept.),
Institute of Technology,
Nirma University,
Ahmedabad.

Electronics & Communication Engineering Branch
Electrical Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382 481
May 2014

Declaration

This is to certify that

1. 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.

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

- Amreen D Charaniya

Certificate

This is to certify that the Major Project entitled **"Methodology for Design Verification"** submitted by **Amreen D Charaniya (12mece02)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Embedded Systems, 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

Dr N.P.Gajjar
Internal 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

Acknowledgements

I would like to express my sincere gratitude to **Dr.Ketan Kotecha** (Director, Nirma University, Ahmedabad) for his continuous guidance, support and enthusiasm. I would take this opportunity to thank **Dr.(Prof.)P.N.Tekwani** (Head of Department, Electrical Engineering), **Dr.N.P.Gajjar** (Professor and Program Coordinator,M.Tech-EC(Embedded System)) and all the faculties at **Nirma University (Embedded System)**, for their vision and relentless effort, support, and encouragement to undertake this thesis work and for their guidelines during the review process.

I am deeply indebted to my thesis supervisors **Dr. N.P.Gajjar**, Sr. Associate Professor, E.C.Dept., Nirma University and **Mr. Abhishek Jain**, Technical Manager at STMicroelectronics Pvt. Ltd. for their constant guidance and motivation. I also wish to thank **Mr. Vishal Jain**, Sr. Engineering Specialist, STMicroelectronics Pvt. Ltd., **Mr. Soyeb Khanusiya**, Design Engg. and all other team members at STMicroelectronics for their constant help and support. Without their experience and insights, it would have been very difficult to do quality work.

I wish to thank my friends of my class for their delightful company which kept me in good humor throughout the year.

Last, but not the least, no words are enough to acknowledge constant support and sacrifices of my family members because of whom I am able to complete the degree program successfully.

- Amreen D Charaniya
12mece02

Abstract

The Imaging group in ST mainly deals with two types of devices - sensors and processors. The main function of sensors is to convert the viewed scene into a data stream. The companion processor function will manage the sensor so that it can produce the best possible pictures and to process the data stream into a form which is easily handled by upstream mobile baseband or MMP (Multi-Media Processor) chipsets.

Image signal processing algorithms are developed and evaluated using Reference models before RTL implementation. After finalizing the algorithm, Reference models are used as a golden model for the IP development. The register configuration of an IP is done by the ST internal bus. Driver(component of UVC) is used to drive the signal to the IP, and the same signal is provided to the reference model. The monitor (component of UVC) senses the signal traffic going to and from the IP. The output from the IP is now obtained by the other UVC. Scoreboard compares the output of the RTL (that was captured by the UVC) with the output of python reference model. Output of RTL and Pyhon Model can be status or/and Image. For IP/soC level verification, System Verilog UVM based Verification Environment will be used. Universal Verification Environment is the first standard, open, interoperable, and proven verification re-use methodology for System Verilog.

For easier verification of register's of DUT and to automate register related activities, The UVM_REG register and memory package was released with UVM 1.1 release. The register model can be written by the user but as almost all devices have thousand's of registers with few memories, witting of register model becomes tedious time consuming task and hence IPXACT tool is used for the generation of register model and almost all of the SV-UVM verification environment files. The IPXACT tool takes specification file as an input to the script and reads the register and memory information from specification file and hence generates the register model according to specification. Since IPXACT tool generates files quickly it has emerged to be an efficient way of generating verification environment files.

Contents

List of Figures

Chapter 1

Introduction

With the projects schedules being so tight, it is important to have a strong verification methodology which contributes to First Silicon Success for all complex design in the semiconductor industries. And so is the need of methodology which enforce full functional coverage and verification of all the corner cases through pseudo random test scenarios.

In 2000, Verisity Design (now Cadence Design Systems, Inc.) introduced a Verification Advisor (vAdvisor), which was targeted for the e-users. In 2002, Verisity announced the first verification library the e-Reuse Methodology(eRM). In 2003, Synopsys announced the Reuse Verification Methodology library (RVM) for the Vera verification language. Over time, RVM was converted to the System Verilog (SV) Verification Methodology Manual (VMM). The Advanced Verification Methodology (AVM) from Mentor was introduced in 2006. Cadence' acquired Verisity in 2005, and began developing a SV version of eRM. The Universal Reuse Methodology (URM) was introduced in early 2007. In January 2008, Cadence and Mentor joined forces to release OVM. The Universal Verification Methodology (UVM) was announced by Accellera on December 23rd, 2009 and introduced to the user on May 17,2010. The Universal Verification Methodology (UVM) has emerged to be a standardized methodology for verifying complex design in the semiconductor industry. UVM has full industry wide support and standardized under the Accellera Systems Initiative. The UVM offers a single cross-industry solution to all the challenges of design verification. It delivers simulator, verification IP and high-level language interoperability within and across companies. A well-defined build flow allows creation of hierarchical reusable environments and hence Universal Verification Methodology (UVM) is said to be the first standard, open, interoperable, and proven verification re-use methodology. UVM is explicitly simulation-oriented, but UVM can also be used alongside assertion-based verification, hardware acceleration or emulation.

The UVM_REG register and memory package was introduced in release of Accellera's UVM 1.0 release for easier verification of register's of DUT and automate register related activities. The UVM_REG register and memory package derives the register level API from VMM and the use model, register sequenced, register operation items, layering concepts, and more from the UVM_RGM register and memory package contributed to UVM World by Cadence. The use of UVM_REG register and memory package has been proven more satisfactory than use of UVM_RGM register and memory package, and hence migration from UVM_RGM register and memory package to UVM_REG register and memory package, becomes necessary for more efficient verification of register and memory of DUT.

Accellera IP-XACT standard is used for capturing register's specifications of IP/SoC. The code generators can be used(that uses IP-XACT standard) for generation of the SV-UVM code for the verification environment at IP level as well as for SoC level verification. As almost all devices have thousand's registers with few memories, which also needs to be verified and to have built Register and memory model for such device is very tedious and time consuming task, building these large amount of register's for a given model may lead to some of the human error, and hence result to inefficient way. So to generate these register model from the specification given in .docx/.mif format is converted to .xml format (with help of spec2spirit script internal to ST). These specification in .xml format is then used as an input to spirit2uvm script which is internal to ST (that uses IP-XACT Standard given by Accellera)to generate the final SV-UVM files to built register model and also those standard sequence such as register read-write sequence, register read sequence and register write sequence are generated by the script to ease the work of verification engineer with the sureness of having the 100% correctness of register model with the specification. As the specification are in standard format it leads to less ambiguities on writing specification of design, and hence provides higher levels of automation. Work on improving this script (spirit2uvm script) is done so as to have not only register model ans standard sequence but also those other SV-UVM files (such as test-case file, top-environment file, virtual sequence, virtual sequencer file and invalid address map files) used in verification for the Design.

1.1 Motivation

An open standard that would deliver verification productivity within design teams and across multi-company is required. Universal verification Methodology is the one open standard interoperable and proven verification re-use methodology. It provide the power to find bugs and benefits of having constrained random testcases. UVM plays an important role in reducing time to market.

The main aim of adopting universal verification methodology (UVM) over other methodology is its reusability through test bench reuse and verification IP allowing plug and play. It provides advanced verification capability and it contains a well maintained Register Package and many predefined test cases for register and memory verification. Also it is a proven methodology with industry wide support and Vendor independent i.e. it does not lock users into a single vendor solution.

UVM_REG register and memory model provides various features to ease verification of register's of DUT and automate register related activities. Compared to UVM_RGM it provides enhanced features of having automatic comparsion on use read and write API, it provides support for little endian and big-endian feature where as in UVM_RGM, externally support had to be given. UVM_REG supports all ipXact 1.5 access policies as well. Also as UVM_REG is gien by Accellera it is suppose to be updated with upgrading of UVM itself.

A register model can be written by hand, However, with more than a few registers this can become a tedious task and has always a potential of having errors in model. So the best practice for using the register & memory package (UVM_RGM or UVM_REG Package) is to use it with IP-XACT. There are a number of other reasons which motivates to use a generator such as :

1. The register model can be generated efficiently without errors.

2. The register model can be re-generated whenever there is a change in the register definition,

3. Multiple format register definitions for different design blocks can be merged together into an overall register description.

4. IPXACT flow is so designed to fit the requirement of the designer expecting to reduce the time-to-market.

5. IPXACT flow is independent of the design language and design tool.

1.2 Thesis Organization

The rest of the thesis is organized as follows.

Chapter-1, (Introduction) provides the introduction of the Universal Verification Methodology (UVM) and its main features. Then, the motivation of choosing Universal Verification Methodology (UVM) over other methodology is given. Also with the purpose of using IP-XACT script and Migration from using UVM_RGM

register and memory package to UVM_REG package.

Chapter-2, (SV-UVM based Verification Environment) provides an overview of main blocks of System Verilog UVM based IP Level Verification Environment. The flow of IP level Verification Environment is described in detail with the execution flow. The information on connection to the DUT interface with that of the verification environment is also provided. And at the end Advantages of Adopting UVM is provided.

Chapter-3 (Register Model) describes main concepts of UVM_REG/UVM_RGM register and memory model and then, usage of UVM_REG/UVM_RGM register model for programming of registers of designs is described. then the common API used are described in detail. The concept of coverage collection is also described and at the end the basic built-in sequences of UVM_REG/UVM_RGM is explained in detail.

The chapter-4, (IPXACT Flow) gives an overview of the SPIRIT script and IP-XACT flow. Then, IP-XACT flow used in imaging group is described. The three scripts (spec2verilog, spirit2uvm, spec2uvm) that are used to automatically generate UVM Verification Environment file are described with detail description of work done on the spirit2uvm script.

Chapter- 5,(Work at ST) provides the overall work done. First Migration from use of UVM_RGM register and memory package to UVM_REG register and memory package is described. Then the modification done to the spirit2uvm script (IPXACT script) is explained with its usage and use of python script for testbench generation is described. Also the modification to the FEKIT (internal Tool) GUI is shown.

Chapter- 6, (Running UVM Simulation) describes the integration of Enterprise Manager (using Verification Cockpit) for running regressions and coverage analysis. And the verification cockpit flow is described in detail.

Finally, in **chapter 7**, (Conclusion) concluding remarks and scope for future work is presented.

Chapter 2

SV-UVM based Verification Environment

This chapter provides the introduction of the Universal Verification Methodology (UVM) and its main features. Then, main components of Verification Environment are described. The flow of IP level Verification Environment is described in detail with the execution flow. The information on connection to the DUT interface with that of the verification environment is also provided. And at the end its main advantages are described.

2.1 Introduction

The Universal Verification Methodology (UVM) has emerged to be a standardized methodology for verifying complex design in the semiconductor industry. UVM has full industry wide support and standardized under the Accellera Systems Initiative.It delivers simulator, verification IP and high-level language interoperability within and across companies and so Universal Verification Methodology (UVM) is said to be the first standard, open, interoperable, and proven verification re-use methodology.UVM is explicitly simulation-oriented, but UVM can also be used alongside assertion-based verification, hardware acceleration or emulation.

The Imaging group in ST mainly deals with two types of devices - sensors and processors. The main function of sensors is to convert the viewed scene into a data stream. The companion processor function will manage the sensor so that it can produce the best possible pictures and to process the data stream into a form which is easily handled by upstream mobile base-band or MMP (Multi-Media Processor) chipsets.
Image signal processing algorithms are developed and evaluated using Reference mod-

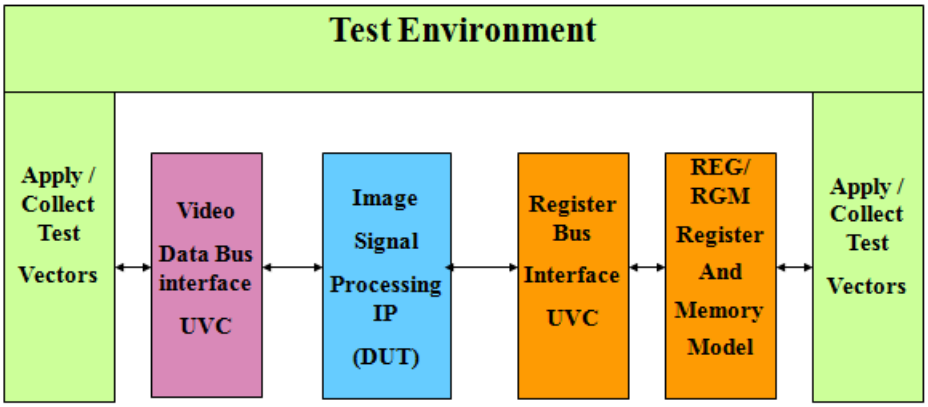

Figure 2.1: IP Level UVM Verification Environment

els before RTL implementation. After finalizing the algorithm, Reference models are used as a golden model for the IP development. the register configuration of an IP is done by the ST internal bus. Driver(component of UVC) is used to drive the signal to the IP,and the same signal is provided to the reference model. The monitor (component of UVC) senses the signal traffic going to and from the IP. The output from the IP is now obtained by the other UVC. Scoreboard compares the output of the RTL (that was captured by the UVC) with the output of python reference model. Output of RTL and Pyhon Model can be status or/and Image.

2.2 Basic blocks of SV-UVM based IP Level Verification Environment

In an image signal processing IP, there are A input video data interfaces, C output video data interfaces, B memory interfaces, D output Interrupts and E register interfaces, where A, B, C, D and E values can be from 0 to any arbitrary number.

For verifying these interfaces, dedicated UVCs are used [1]. In case of register interface(s), STBus UVC and RGM/REG register model are used. Similarly for video data interface(s), video data interfaces UVCs (IDP/ISB/VDB) are used. There can be multiple instances of these UVCs in a verification environment. Each agent is configured separately and any combination of agent configurations can coexist in the same environment. Therefore in above case, E instances of register interface UVC agents, M (M = max (A, C)) instances of video data interface UVC agents and D instances of interrupt checker are used to interface with a DUT. Figure below illustrates the basic blocks of System Verilog UVM based IP Level Verification Environment.

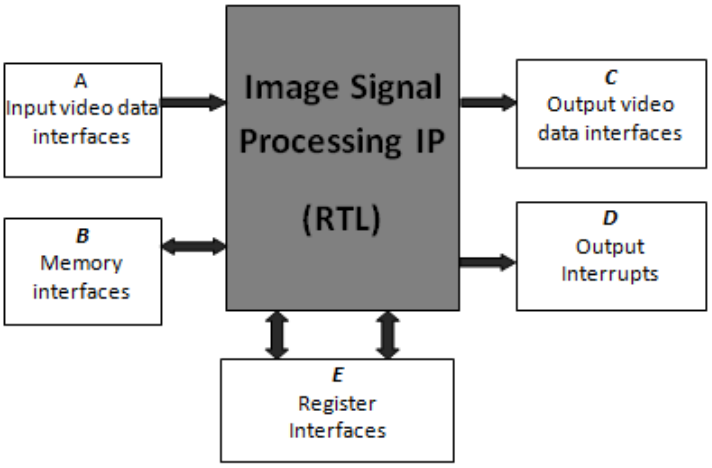

Figure 2.2: Basic blocks of SV-UVM based IP Level Verification Environment

2.3 UVM Verification Environment

An UVM verification Environment is composed of reusable verification component called universal verification components. A verification component is an encapsulated, ready-to-use, configurable verification environment for an interface protocol, a design submodule, or a full system [3]. The verification component is applied to the device under test (DUT) to verify the implementation of the protocol or design architecture.

The UVM testbench architecture is modular to facilitate the reuse of groups of verification components either in different projects (horizontal reuse) or at a higher level of integration in the same project (vertical reuse) [1]. The main component of the environment are :

- **Transaction** : A bundle of data items, which may be distributed over time and space in the system, and which form a communication abstraction such as a handshake, bus cycle, or data packet

- **Sequence** : An ordered collection of transactions or of other sequences. Sequences are assembled from transactions and are used to build realistic sets of stimuli. A sequence could generate a specific pre-determined set of transactions, a set of randomized transactions, or anything in between. Transactions and sequences together represent the domain of dynamic data within the verification environment.

  In terms of class inheritance, the uvm_sequence inherits from the uvm_sequence_item which inherits from the uvm_object. Both base classes are known as objects

rather than components.Sequences are the primary means of generating stimulus in the UVM. The fact that sequences and sequence_items are objects means that they can be easily randomized to generate interesting stimulus. Their object orientated nature also means that they can be manipulated in the same way as any other object.

In the UVM sequence architecture, sequences are responsible for the stimulus generation flow and send sequence_items to a driver via a sequencer component. The driver is responsible for converting the information contained within sequence_items into pin level activity. The sequencer is an intermediate component which implements communication channels and arbitration mechanisms to facilitate interactions between sequences and drivers. The flow of data objects is bidirectional, request items will typically be routed from the sequence to the driver and response items will be returned to the sequence from the driver. The sequencer end of the communication interface is connected to the driver end together during the connect phase.

- **Sequence Items** : As sequence_items are the foundation on which sequences are built, some care needs to be taken with their design. Sequence_item content is determined by the information that the driver needs in order to execute a pin level transaction; ease of generation of new data object content, usually by supporting constrained random generation; and other factors such analysis hooks. By convention sequence_items should also contain a number of standard method implementations to support the use of the object in common transaction operations, these include copy, compare and convert2string.

- **A Driver** :A driver is an active entity that drives the DUT. The driver is responsible for converting the data inside a series of sequence_items into pin level transactions. The driver pulls transactions from its sequencer and controls the signal-level interface to the DUT. The transaction-level interface between the sequencer and the driver is a fixed feature of UVM, and is unusual in the sense that both the port and the export required for TL- communication are implicit.

- **A Sequencer** : A sequencer is a stimulus generator that controls the sequence items provided to/from the driver for execution. Sequencer runs sequences and sends them downstream to drivers or to other sequencers. The role of the sequencer is to route sequence_items from a sequence where they are generated to/from a driver. At its simplest a sequencer looks like any other component, except that it has an implicit transaction-level export for connection to a driver. The transfer of request and response sequence items between sequences and their target driver is facilitated by a bidirectional TLM communication mechanism implemented in the sequencer. The uvm_driver class contains an uvm_seq_item_pull_port which should be connected to an uvm_seq_item_pull_export

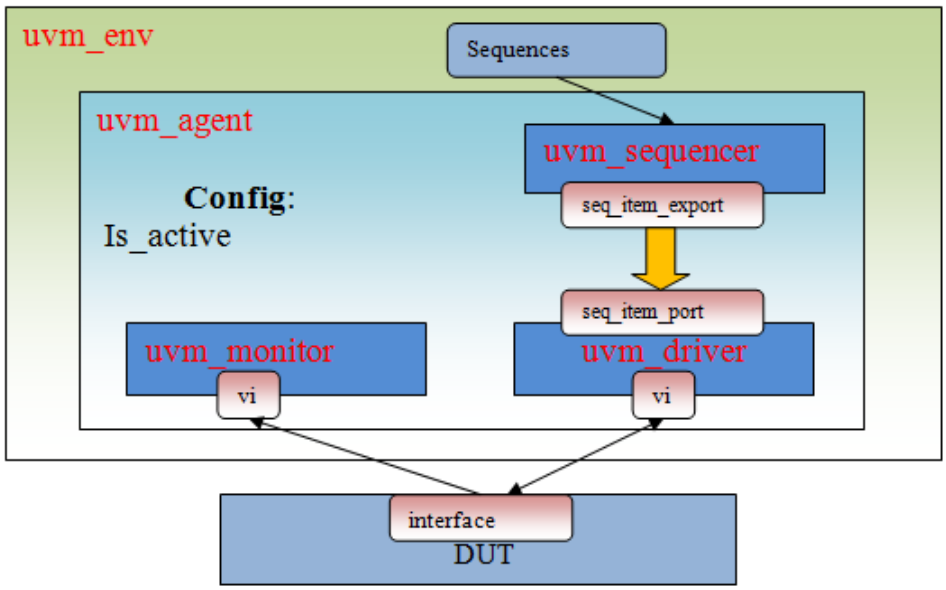

Figure 2.3: Different Components of Universal Verification Component (UVC)

in the sequencer associated with the driver. The port and export classes are parameterised with the types of the sequence_items that are going to be used for request and response transactions. Once the port-export connection is made, the driver code can use the API implemented in the export to get request sequence_items from sequences and return responses to them. By default, a sequencer behaves similarly to a simple stimulus generator and returns a random data item upon request from the driver. This default behavior leads us to add constraints to the data item class in order to control the distribution of randomized values.

- **A Monitor** :A monitor is a passive entity that samples DUT signals but does not drive them. Monitors collect coverage information and perform checking. Even though reusable drivers and sequencers drive bus traffic, they are not used for coverage and checking. Monitors are used instead.The monitor observes pin level activity and converts its observations into sequence_items which are sent to components such as scoreboards which use them to analyse what is happening in the testbench.

- **The Agent** : Sequencers, drivers, monitors, and collectors can be reused independently, but this requires the environment integrator to learn the names, roles, configuration, and hookup of each of these entities. To reduce the amount of work and knowledge required by the test writer, UVM recommends that environment developers create a more abstract container called an agent. Agents encapsulate a driver, sequencer, and monitor. Verification components can con-

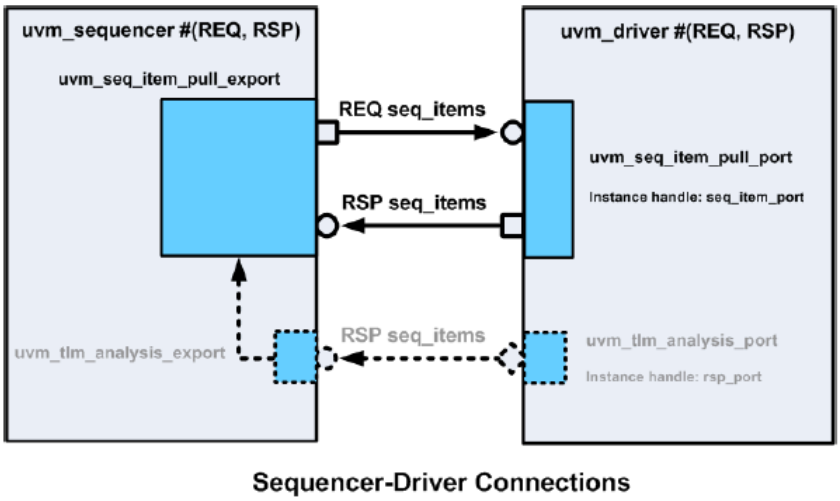

Figure 2.4: Sequencer

tain more than one agent. Some agents initiate transactions to the DUT, while other agents react to transaction requests. A UVM agent can be thought of as a verification component kit for a specific logical interface. The agent is developed as package that includes a SystemVerilog interface for connecting to the signal pins of a DUT, and a SystemVerilog package that includes the classes that make up the overall agent component. The agent class itself is a top level container class for a driver, a sequencer and a monitor, plus any other verification components such as functional coverage monitors or scoreboards. The agent also has an analysis port which is connected to the analysis port on the monitor, making it possible for a user to connect external analysis components to the agent without having to know how the agent has been implemented. The agent is the lowest level hierarchical block in a testbench and its exact structure is dependent on its configuration which can be varied from one test to another via the agent configuration object. Agents should be configurable so that they can be either active or passive. Active agents drive transactions according to tests whereas Passive agents only monitor DUT activity.

- **Scoreboards** : A scoreboard is an analysis component that checks whether the DUT is behaving correctly or not. UVM scoreboards use analysis transactions from the monitors implemented inside agents. Scoreboard compares the output of the RTL with the output of Python Model. Output of RTL and Python Model can be status or/and Image.

- **Virtual Sequence** : A virtual sequence is a sequence which controls stimulus generation using several sequencers. Since sequences, sequencers and drivers are focused on point interfaces, almost all testbenches require a virtual sequence to co-ordinate the stimulus across different interfaces and the interactions between

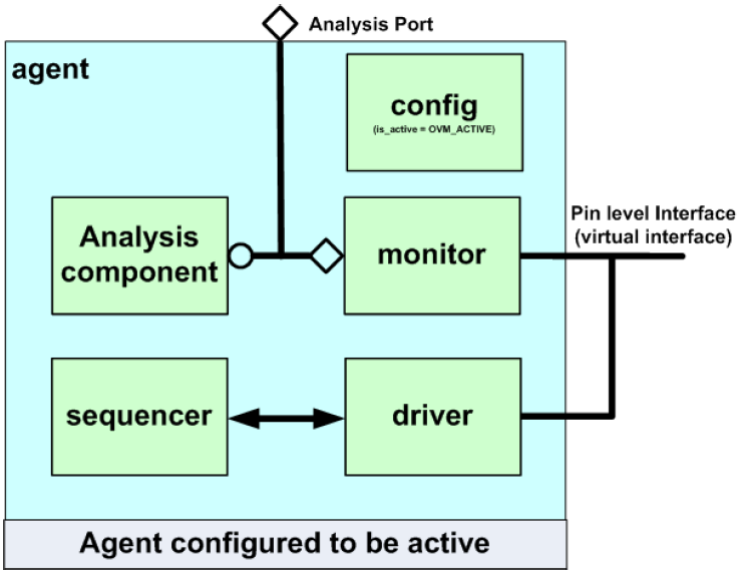

Figure 2.5: Agent

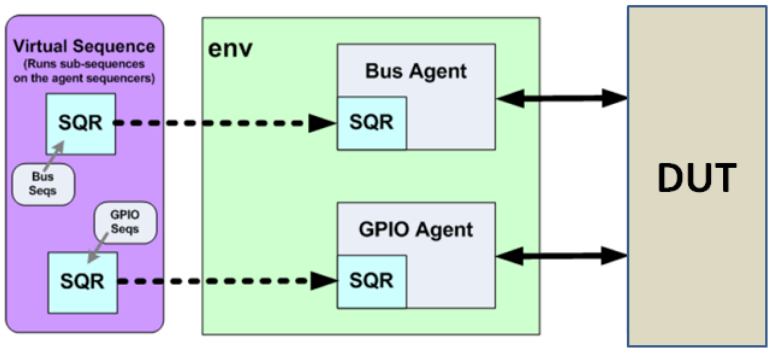

Figure 2.6: Virtual Sequence

them. A virtual sequence is often the top level of the sequence hierarchy. A virtual sequence might also be referred to as a 'master sequence' or a 'co-ordinator sequence'.A virtual sequencer is used in the stimulus generation process to allow a single sequence to control activity via several agents. Virtual sequences runs on virtual sequencer to drive UVC sequencer instances. A virtual sequence differs from a normal sequence in that its primary purpose is not to send sequence items. Instead, it generates and executes sequences on different target agents. To do this it contains handles for the target sequencers and these are used when the sequences are started.

- **Functional Coverage Monitors** : A functional coverage monitor analysis component contains one or more covergroups which are used to gather functional coverage information related to what has happened in a testbench during a test

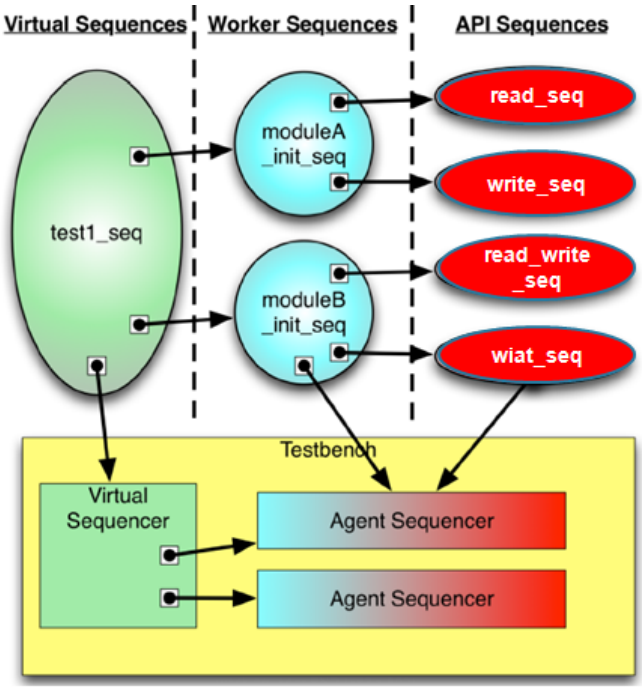

Figure 2.7: Working of virtual Sequencer

case. A functional coverage monitor is usually specific to a DUT.

2.4 Running a sequence

Some sequences are used as part of a reusable component and others are created to have a test for specific corner case for DUT. There are three steps to run the sequence first is to create the sequence, then configuring it and then starting the sequence to the sequencer.

The sequence is derived from the uvm sequence base class and the request and response item type parameters are specified. In the example code, only the request type is specified(interface_type). This will result in the response type also being of type interface_type. The 'uvm_sequence_utils macro is used to associate the sequence with the relevant sequencer type and to provide the various automation utilities. This macro also adds a p_sequencer variable that is a pointer to the specific sequencer invoking that sequence. Static sequencer properties such as hierarchical path, end-of-test control and more are accessible through the p_sequencer variable. Now implement the sequence's body () task with the specific scenario that is needed by the sequence to execute. In these body task, data items and other sequences are executed using "'uvm_do" and "'uvm_do_with".

```
1    Class example_seq extends uvm_sequence#(interface_type);
2// Constructor and UVM automation macros
3    'uvm_sequence_utils(example_seq, example_master_sequencer)
4      function new(string name="example_seq");
5          super.new(name);
6      endfunction : new
7      virtual task body();
8          'uvm_do(req)
9       endta&k : body
10   endclass : example_seq
```

Figure 2.8: Sequence creation Code

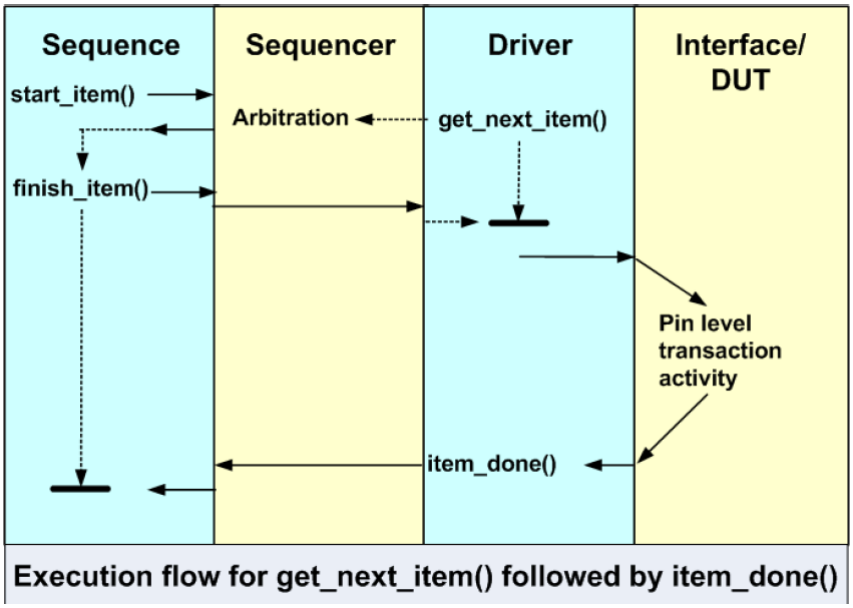

Figure 2.9: Execution Flow

2.4.1 Execution Flow

The purpose of the driver sequencer API is for the driver to receive a series of sequence_items from sequences containing the data required to initiate transactions, and for the driver to communicate back to the sequence that it has finished with the sequence_item and that it is ready for the next item. This use model allows the driver to get a sequence item from a sequence, process it and then pass a hand-shake back to the sequence using item_done(). No arguments should be passed in the item_done() call. This is the preferred driver-sequencer API use model, since it provides a clean separation between the driver and the sequence.

The corresponding sequence implementation would be a start_item() followed by a finish_item(). Since both the driver and the sequence are pointing to the same sequence_item, any data returning from the driver can be referenced within the sequence

via the sequence_item handle. In other words, when the handle to a sequence_item is passed as an argument to the finish_item() method the drivers get_next_item() method call completes with a pointer to the same sequence_item. When the driver makes any changes to the sequence_item it is really updating the object inside the sequence. The drivers call to item_done() unblocks the finish_item() call in the sequence and then the sequence can access the fields in the sequence_item, including those which the driver may have updated as part of the response side of the pin level transaction.

2.5 Connections to DUT Interfaces

The Device Under Test (DUT) is typically a Verilog module or a VHDL entity/architecture while the testbench is composed of SystemVerilog class objects.

The DUT and testbench belong to two different SystemVerilog instance worlds. The DUT belongs to the static instance world while the testbench belongs to the dynamic instance world. Because of this the DUT's ports can not be connected directly to the testbench class objects so a different SystemVerilog means of communication, which is virtual interfaces, is used. The DUT's ports are connected to an instance of an interface. The Testbench communicates with the DUT through the interface instance. Using a virtual interface as a reference or handle to the interface instance, the testbench can access the tasks, functions, ports, and internal variables of the SystemVerilog interface. As the interface instance is connected to the DUT pins, the testbench can monitor and control the DUT pins indirectly through the interface elements.

When using virtual interfaces the location of the interface instance is supplied to the testbench so its virtual interface properties may be set to point to the interface instance. The recommended approach for passing this information to the testbench is to use either the configuration database usingthe config_db API or to use a package.

2.6 Advantages of Adopting UVM

The main advantage of adopting universal verification methodology (UVM) over other methodology is its reusability through test bench reuse and verification IP allowing plug and play. Methodology provides full functional coverage and verification of all the corner cases through pseudo random test scenarios and hence help in finding bug more efficiently and in reduced time. The UVM has UVM_REG register and memory model library as an open source library, making it to support verification of hardware registers and memory blocks. UVM_REG register and memory package, being an part of the UVM library, makes UVM more efficient methodology for verification. Also it is a proven methodology with industry wide support and Vendor independent i.e. it

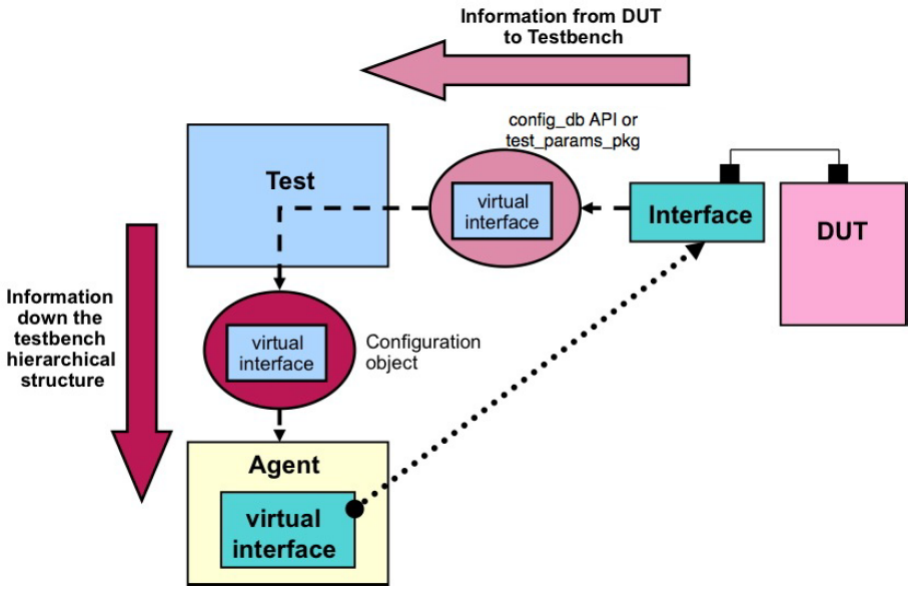

Figure 2.10: Virtual Interface

does not lock users into a single vendor solution.

Chapter 3

Register Model

This chapter provides the overview of the main concepts of UVM_REG/UVM_RGM register and memory model and and then, usage of UVM_REG/UVM_RGM register model for programming of registers of designs is described. then the common API used are described in detail. The concept of coverage collection is also described and at the end the basic built-in sequences of UVM_REG/UVM_RGM is explained in detail.

3.1 Introduction

Almost all devices have registers and memories that need to be controlled, checked, and covered as part of the verification task. Verifying the behavior of registers and memory blocks is always an essential part of the verification process. In verifying a device under test (DUT), one often needs to Capture registers attributes and dependencies, Randomize the device configuration and the initial register values, Execute bus transactions to write the initial configuration to the DUT, Read and write registers and memories as part of the normal run-time operation and Debug and analyze register activities. And hence comes the use of Register Model.

The UVM register model provides a way of tracking the register content of a DUT and a convenience layer for accessing register and memory locations within the DUT. Standard Register and Memory model is used for efficient register and memory verification. Register model allows the features of Address mapping, Modeling registers and memory blocks, Front door and back-door access to Device under Verification (DUV), Implicit and explicit prediction of registers and memory blocks values and Coverage model API. The UVM register model is designed to ease efficient verification of programmable hardware. Main purpose of the register model is to make it easier to write reusable register/memory sequences that access hardware registers and memory areas.

3.2 Register Model

In the testbench, the register model object needs to be constructed and a handle needs to be passed around the testbench environment using either the configuration and/or the resource mechanism. In order to drive an agent from the register model an association needs to be made between it and the target sequencer so that when a sequence calls one of the register model methods a bus level sequence_item is sent to the target bus driver. 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.

The UVM register package contains built-in test sequences library which is used to perform most of the basic register and memory tests, such as testing of register reset values and testing of the register and memory data paths. The model supports both front door and back door access to the DUT registers. Front door access uses the control bus agent in the test bench and register accesses use the normal control bus transfer protocol. Back door access bypass the normal bus interface logic and uses simulator data base access routines to directly force or observe the register hardware bits in zero simulation time.

3.2.1 RGM Model

The uvm_rgm package provides a methodology to enable productive and reusable register-related verification logic. The basic Architecture for RGM MODEL is shown below.

3.2.1.1 The Register Database (RGM_DB)

RGM database (RGM_DB) component contains the entire register and memory model. All the information is placed within the RGM_DB component after capturing the device memory and register model. Unlike the address map and register files, which are UVM objects that can be allocated dynamically, the RGM_DB is an UVM quasi-static component which can be placed in testbench (uvm_env).

3.2.1.2 The Register Sequencer and Sequences

The uvm_rgm package uses the familiar UVM sequence mechanism to randomize and drive register and memory sequences. In a sequence, you can randomly select a register object from the RGM_DB, randomize it, set the access direction (read or write), and perform the operation. Register/memory operation sequences look much like any other UVM sequence. Using the sequence mechanism allows you to create reusable sequences to support different configuration modes, use an existing sequence as a sub-sequence,

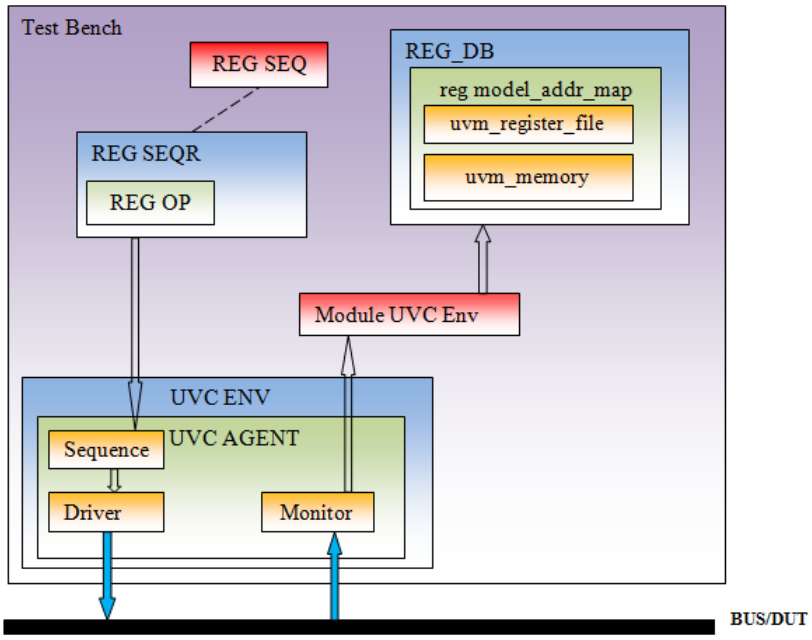

Figure 3.1: RGM Model

traverse through all the register in the addresses range, and much more. An API is provided to perform read and write operations.

3.2.1.3 The Bus Interface UVC

The RGM sequencer is layered on top of an existing bus master sequencer. The bus master emulates the CPU as it programs, controls, and supervises other devices on the bus. Every read and write operation is translated to protocol-specific bus transactions. This isolation between register operation and protocol-specific bus transactions allows reuse of the same register operation sequences, even if the specification changes to use a different bus. The bus interface is extended to support register sequence operations using the factory.

3.2.1.4 The Interface UVC Monitor

As mentioned before, the monitoring path is independent from the injection facilities. Again, we use the protocol-specific bus monitor to detect a bus transaction. At that point, the transaction information is sent to the module UVC. The interface UVC monitor as a reusable component is unaware of whether the transaction information collected is general bus traffic or a specific register access. This knowledge should be partitioned into the Module UVC.

3.2.1.5 Updating the Model

Every operation (READ/WRITE) on the DUTs registers through the bus is sensed by the monitor, which then does corresponding update or compare (and update) of the shadow model. That way, there is constant alignment between the DUTs status and the registers model which lies inside the RGM_DB. This task should be done by the integrator, in charge of connecting the register model with the UVC. Keeping the model up to date with the DUT is achieved by following these steps:

- Create the module UVC (or extend the existing module UVC), which is in charge of updating the register model after each transaction that is captured by the interface UVC monitor.
- Connect the interface UVC monitor to the module UVC such that the transactions collected are available to the module UVC.
- When each WRITE transaction is received by the module UVC. The module UVC should update the register model by calling the update() method providing the address and data (and any mask if applicable for say byte enables). This will keep the register model in sync with what was written into the DUT. When each READ transaction is received by the module UVC, the module UVC should compare the data to what is contained in the register model by calling the compare_and_update() method. compare_and_update(), as indicated by the name, also updates the register model to the data that was provided after the compare has taken place.

3.2.1.6 The Module UVC

The Module UVC takes the transactions collected by the interface UVC monitor and decides what action to execute on the register and memory model. In the case of a write access to a register or memory location, the shadow register, or memory location is updated in the RGM_DB structure. Therefore, the content of the shadow register in the RGM_DB structure is synchronized with the DUT registers. When a read access is detected on the bus, the monitor accesses the RGM_DB structure and compares the read result from the DUT to the value in the RGM_DB shadow model. If it does not match, a DUT error is issued. In addition, the RGM_DB structure can collect coverage on the register accesses and values (when coverage collection is enabled).

- **Creating the Module UVC** : The module UVC is a component which is in charge of updating the register model after each interface UVC transaction related to the registers is detected. This task should be done by the integrator. The module UVC keeps the shadow model aligned with the DUTs registers status by notifying the register model after each UVC register transaction. This is done by calling the update() method after each WRITE transaction and calling

```
class xbus_parallel_mem extends uvm_rgm_bank_memory #(64,20,2);
        `uvm_object_utils(xbus_parallel_mem)
        function new(input string name="unnamed-xbus_parallel_mem");
          super.new(name);
            set_reset_value(uvm_rgm_hard_reset, 64'h5a5a5a5a_5a5a5a5a);
            set_reset_value(uvm_rgm_soft_reset, 0);
            set_memory_bank_type(PARALLEL_MEM); // Bank memory kind
            reset(uvm_rgm_hard_reset);
        endfunction
endclass
```

Figure 3.2: Memory Definition

compare_and_update() after each READ transaction.

3.2.1.7 Defining Memory Bank

UVM_RGM supports serial and parallel memory banks. Serial specifies the first item is located at the banks base address. Each subsequent item is located at the previous items address, plus the range of that item (adjusted for LAU and bus width considerations, rounded up to the next whole multiple). This allows the user to specify only a single base address for the bank and have each item line up correctly. Parallel specifies each item is located at the same base address with different bit offsets. The bit offset of the first item in the bank always starts at 0, the offset of the next items in the bank is equal to the widths of all the previous items. In terms of implementation, both serial and parallel memories are implemented as flat, continuous memory. Handling backdoor operation is the only implementation difference between a normal and bank memory. Bank memory is an extension of uvm_rgm_bank_memory class.

3.2.2 REG Model

The UVM register library (UVM_REG) is an open source library, being part of the UVM library, allows an easy modeling and verification of hardware registers and memory blocks. The UVM_REG combines elements from multiple proprietary solutions (e.g. Synopsys RAL, and Cadence UVM_RGM) with new code from Mentor for tight alignment with the UVM BCL and methodology. The UVM 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

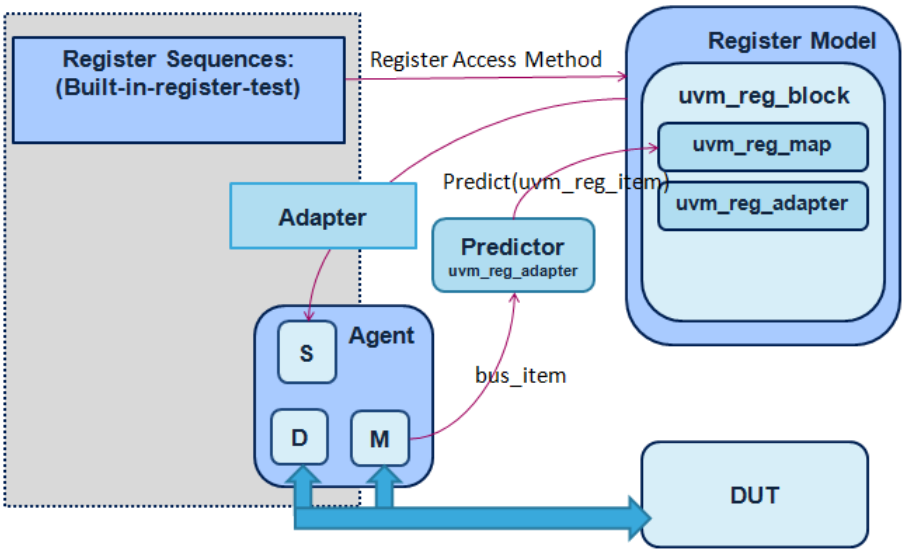

Figure 3.3: REG Model

items, and to be able to convert bus sequence item responses back to bus sequence items. The adapter should be implemented by extending the uvm_reg_adapter base class.

The overview of the different UVM classes used to build register database is described below:

- **Field** : A group of bits providing specific functionality in a hardware register. It represents different register field within the register. It is modeled in the UVM register library using the uvm_reg_field class, and configured using the uvm_reg_field::configure() method. And uvm_reg_field is the lowest register abstraction layer. The uvm_reg_field has several properties :

  1. reset["HARD"] (property) stores a hard reset value.

  2. mirrored (property) stores the value of what we think in our design under test (DUT).

  3. desired_value (property) stores the value of what we want to set to the DUT.

  4. value (property) stores the value to be sampled in a functional coverage, or the value to be constrained when the field is randomized.
  Among these properties, only the value property is public. The other properties are local, thus we cannot access them directly. To access these local properties use of register access methods is done .

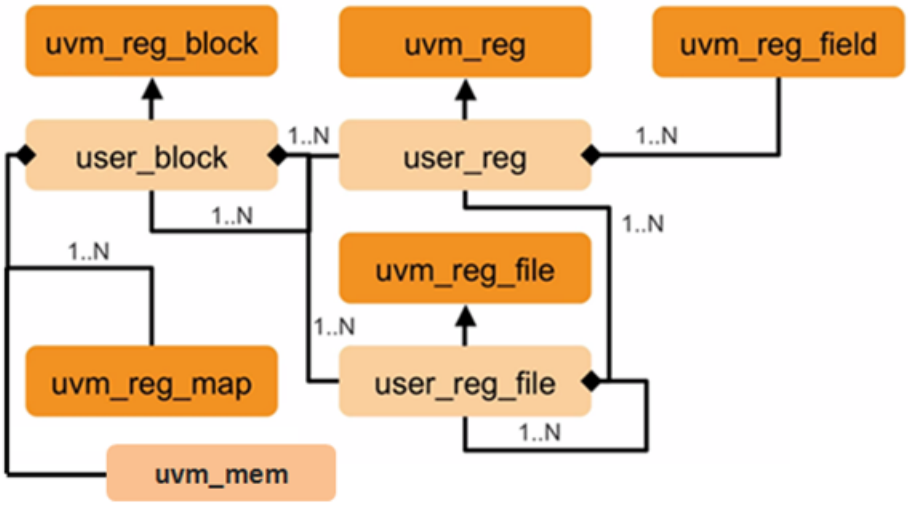

Figure 3.4: Register Model Hierarchy

Reserved Fields:There is no pre-defined field access policy for reserved fields.Reserved fields are left unmodelled i.e. they will be assumed to be RO fields filled with 0s.

- **Register** : A hardware register model grouping fields at different offsets within the register. It is modeled in the UVM register library by extending the uvm_reg base class adding rand objects of uvm_reg_field type, and configured using the uvm_reg::configure() method.
  An example of uvm_reg definition is shown below:

- **Register File Types** : A register file type is constructed using a class extended from the uvm_reg_file class. Register files can contain registers and other register files also. There must be one class per unique register file type. The name of the register file type is created by the register model generator. The name of the register file type class must be unique within the scope of its declaration. The register file type class must include an appropriate invocation of the ‘uvm_object_utils() macro.

- **Memory** : A memory block with well-defined address range. It is modeled in the UVM register library by extending the uvm_mem base class defining the memory block specifications inside the constructor new(), and configured using the uvm_mem::configure() method.

- **Block** : Groups registers, memories and sub-blocks. It is modeled in the UVM register library by extending the uvm_reg_block base class, then instantiating and configuring registers, memories and sub-blocks inside its build() method.

- **Map** : Locates the address offset of registers, memories and sub-blocks within a block. It is modeled in the UVM register library by instantiating an object of

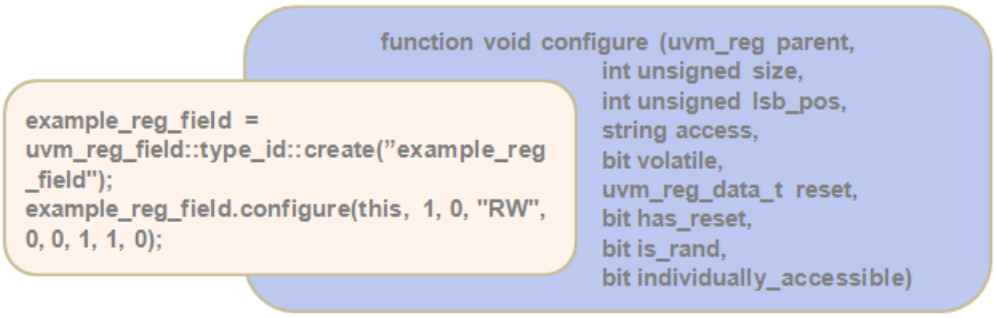

Figure 3.5: UVM reg field Configuration

```
class example_reg extends uvm_reg;
`uvm_object_utils(example_reg)
rand uvm_reg_field example_field;
function new (...);
virtual function void build();
example_field = uvm_reg_field::type_id::create("example_field");
example_field.configure(this, 32, 0, "RO", 0, 32'h00004444, 1, 1, 1);
endfunction
endclass
```

Figure 3.6: UVM Reg Example

uvm_reg_map class in a block. Registers and memories are added to the address map using uvm_reg_map::add_reg() and uvm_reg_map::add_mem() respectively.

The corresponding register block only has to create the enclosing register and memory and then add it to the required address maps, as shown:

Registers are defined by extending uvm_reg class. Each field of the register is defined as uvm_reg_field then configured in the build_configuration function. Memories are defined by extending uvm_mem class. The reg_block contains the registers and memories defined above and a address map. Address maps can be composed into higher-level address maps.Registers and register files are an excellent vertical reuse (module-to-system) opportunity, as sub-systems configuration logic is valid and reusable at the system integration level. Designs can be packaged with their configuration sequences allowing the system integrator smooth operations without the need to learn all the sub-system configuration details. The register model is a hierarchal reference model for a specific DUT and captures the DUT memories and registers structure and at-

```
class my_reg_block extends uvm_reg_block;
virtual function void build();
...
example_reg =
example_reg::type_id::create("example_reg");
example_reg.configure(this, null, "example_reg");
example_reg.build();
default_map.add_reg(example_reg, 'h24, "RW");
...
```

Figure 3.7: UVM Reg Block Example

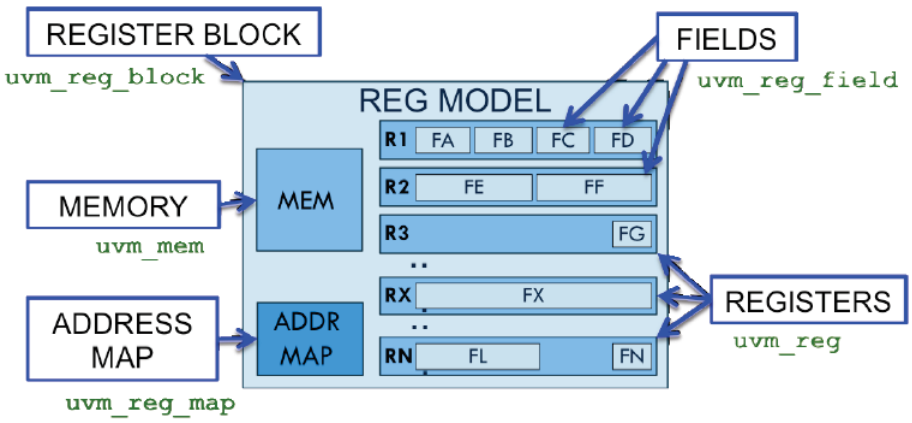

Figure 3.8: Register Model Structure

tributes. It contains nested objects of register blocks, registers and their field class that are derived from the uvm_reg classes and specialized to the specifications at hand. The model allows the randomization of configuration values, checking of the DUT register values for correctness and collection of coverage.

There are two parts to the register adaption layer, the first part implements the sequence based stimulus layering and the second part implements the analysis based update of the register model using a predictor component.

3.2.2.1 Register Sequence Adaption Layer

The register sequence layering adaption should be done during the UVM connect phase when the register model and the target agent components are known to have been built. The register layering for each target bus interface agent supported by the

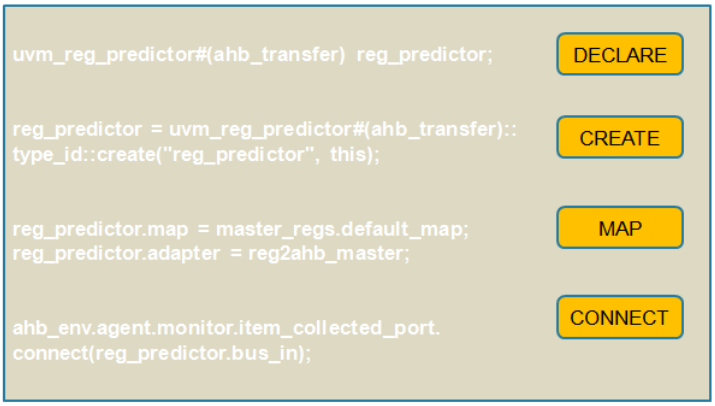

Figure 3.9: Predictor

register model should only be done once for each map. In a block level environment, this will be in the env, but in environments working at a higher level of integration this mapping should be done in the top level environment. In order to determine whether the particular env is at the top level the code should test whether the parent to its register block is null or not - if it is, then the model and therefore its env is at the top level.

3.2.2.2 Register Prediction

By default, the register model uses a process called explicit_prediction to update the register data base each time a read or write transaction that the model has generated completes. Explicit prediction requires the use of a uvm_reg_predictor component. uvm_reg_adapter converts between register model read and write methods and the interface-specific transactions. uvm_reg_predictor updates the register model based on observed transactions published by a monitor.

3.2.2.3 Field Access Policies

The UVM provides a comprehensive set of pre-defined field access policies [1], which are summarized in Table shown below. The field access policy is normally setup during ”build” by the ”configure” method.

Field access policies should not be confused with the access rights declared when a register is added to a particular address map. Registers can be added to more than one map (corresponding to different interfaces in the DUT) with different access rights (RW, RO or WO are the only choices); whether a register field can be read or written depends on both the fields configured access policy and the registers rights in the map being used to access the field.

	NO WRITE	WRITE VALUE	WRITE TO CLEAR	WRITE TO SET	WRITE TO TOGGLE	WRITE ONCE
NO READ	-	WO	WOC	WOS	-	WO1
READ VALUE	RO	RW	WC W1C W0C	WS W1S W0S	W1T W0T	W1
READ TO CLEAR	RC	WRC	-	WSRC W1SRC W0SRC	-	-
READ TO SET	RS	WRS	WCRS W1CRS W0CRS	-	-	-

Figure 3.10: Register Field Policies

3.2.2.4 Register Access Methods for uvm_reg model

Each field has a corresponding predicted and mirrored value, as well as reset state and hooks for additional operations such as randomization via a value field, as shown in Figure.

- **reset()** : The reset() method resets the properties of a register field, if the m_reset[kind] exists. The default kind is "HARD". If the m_reset[kind] does not exist, the reset() method does nothing. Note that the reset() method does not reset a register in the DUT. It only resets the properties of a register-field object.

- **set()** : The set() method sets the desired value of a register field. The set() method does not set the value to a register in the DUT. It only sets the value to the m_desired and the value properties of a register-field object.

- **get()** : The get() method gets the desired value of a register field. The get() method does not get the value from a register in the DUT. It only gets the value of the m_desired property. To actually get the value from the DUT, use read() or mirror() methods. These methods will be explained later. Similarly to the get() method, there are two more getters to access the local properties. The get_reset() retrieves the value of the m_reset[kind] property, while the get_mirrored_value() method retrieves the value of the m_mirrored property.

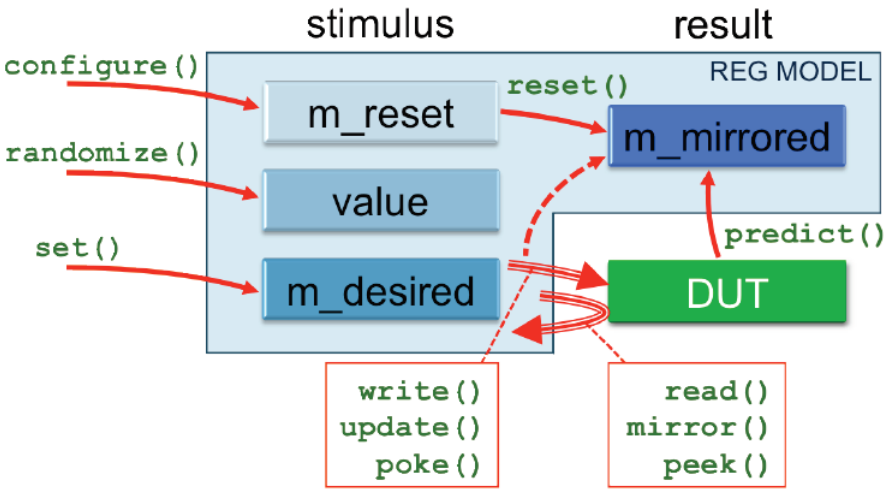

Figure 3.11: Register Access API

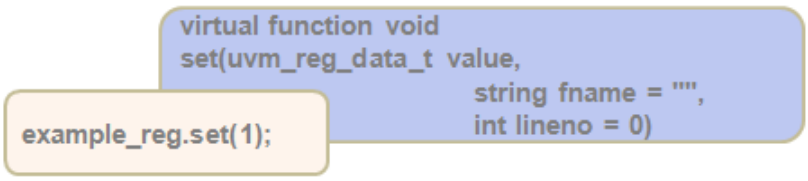

Figure 3.12: Set Value to Register in Register Model

Figure 3.13: Get Value of Register from Register Model

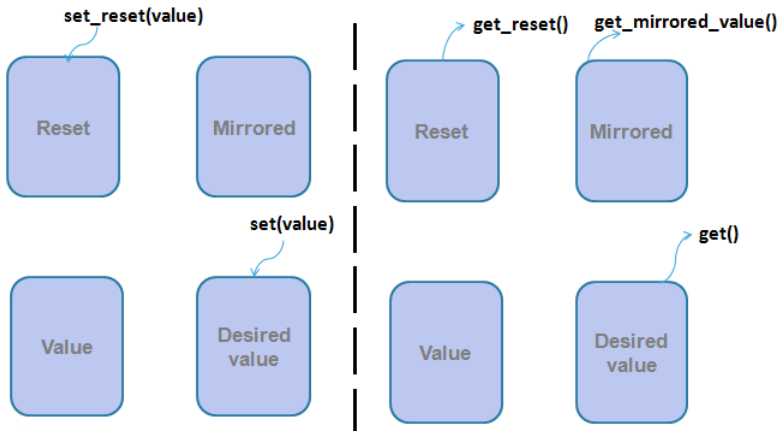

Figure 3.14: Working of SET and GET Function

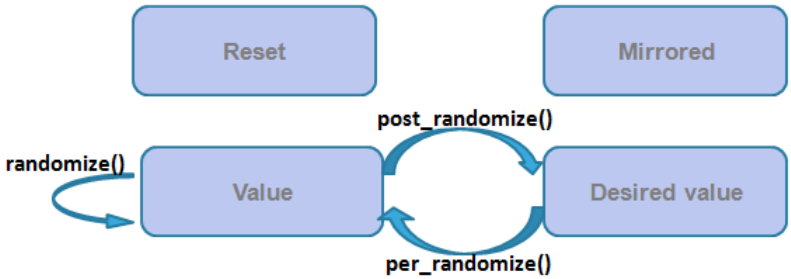

Figure 3.15: Working of Randomize Function

- **randomize()** : The randomize() method is a System-Verilog method. It randomizes the value property of a register-field object. After the randomization, the post_randomize() method copies the value of the value property to the m_desired property. Note that the pre_randomize() method copies the value of the m_desired to the value property if the rand_mode of the value property is OFF.

- **write()** : The write() method actually writes a value to the DUT. The write() method involves multiple steps.
  1. A uvm_reg_item object corresponding to the write operation is created.

  2. The uvm_reg_adapter converts the write operation to a corresponding bus transaction.

  3. The uvm_driver executes the bus transaction to the DUT.

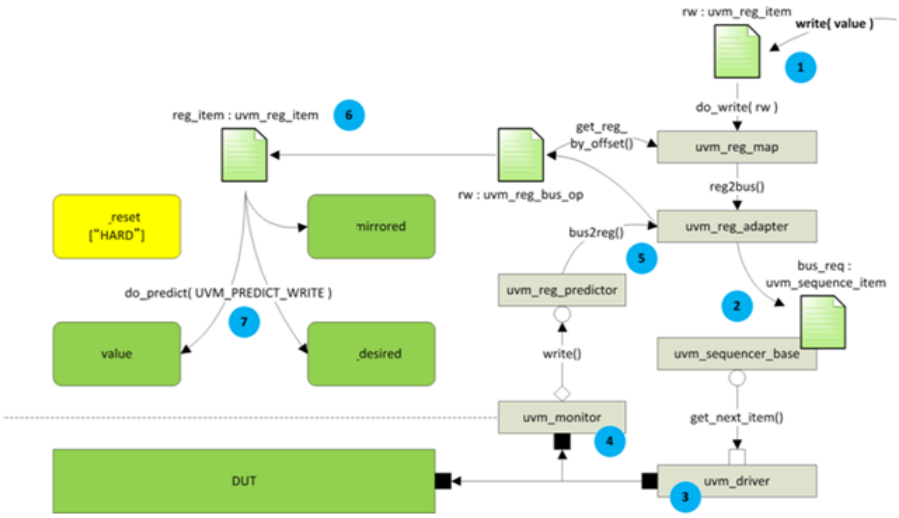

Figure 3.16: Working of Write Function

```
virtual task write (output uvm_status_e  status,
                    input uvm_reg_data_t  value,
                    input uvm_path_e  path = UVM_DEFAULT_PATH,
                    input uvm_reg_map  map = null,
                    input uvm_sequence_base  parent = null,
                    input int prior = -1,
                    input uvm_object  extension = null,
                    input string fname = "",
                    input int lineno = 0)
```

```
enable.write( status,
              enable.get(),
              .parent(this) );
```

Figure 3.17: Code for Write Function

4. The uvm_monitor captures the bus transaction.

5. The uvm_reg_predictor asks the uvm_reg_adapter to convert the bus transaction to a corresponding register operation.

6. The register operation is converted to a uvm_reg_item.

The uvm_reg_item is used to update the value, m_mirrored, and m_desired properties.

- **read()** : The read() method actually reads a register value from the DUT. Also

if the individually_accessible argument was 0 when the register field was configured, the entire register containing the field is read. In this case, the m_mirrored values are updated for the other fields as well.

Similarly to the write() method, the read() method involves multiple steps.

1. A uvm_reg_item object corresponding to the read operation is created.

2. The uvm_reg_adapter converts the read operation to a corresponding bus transaction.

3. The uvm_driver executes the bus transaction to the DUT.

The uvm_reg_apapter converts the bus transaction with read data to a register operation.

4. The read() method returns the read value to the caller.

5. In the mean time, the uvm_monitor captures the bus transaction.

6. The uvm_reg_predictor asks the uvm_reg_adapter to convert the bus transaction to a corresponding register operation.

7. The register operation is converted to a uvm_reg_item.

8. The uvm_reg_item is used to update the value, m_mirrored, and m_desired properties.

- **update()** : The update() method actually writes a register value to the DUT. The update() method belongs to the uvm_reg class. The uvm_reg_field class does not have the update() method.

  The differences between the write() method and the update() method are:

  1. The write() method takes a value as its argument, while the update() method uses the value of the m_desired property as the value to write.

  2. The update() method writes the value only if the m_mirrored and the m_desired are not equal.

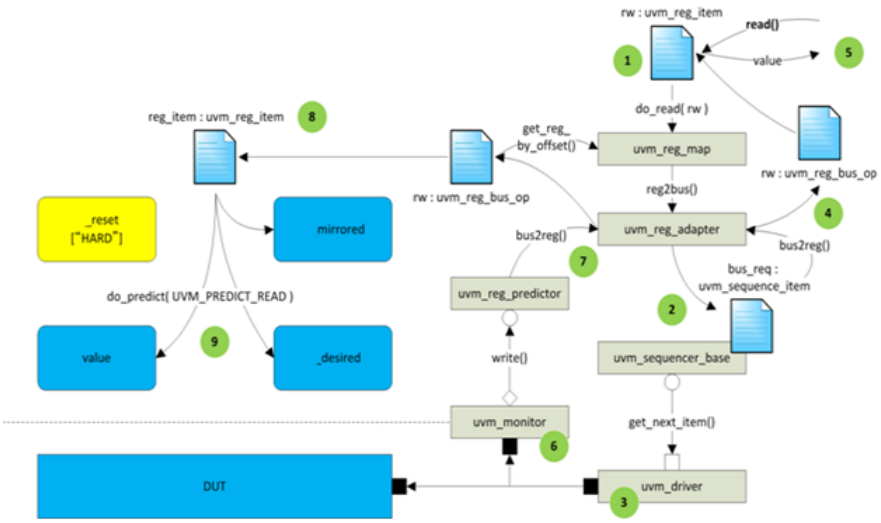

Figure 3.18: Working of Read Function

```
virtual task read (output uvm_status_e  status,
                   output uvm_reg_data_t  value,
                   input uvm_path_e  path = UVM_DEFAULT_PATH,
                   input uvm_reg_map  map = null,
                   input uvm_sequence_base  parent = null,
                   input int prior = -1,
                   input uvm_object  extension  = null,
                   input string fname = "",
                   input int lineno = 0 )
```

```
uvm_status_e    status;
uvm_reg_data_t  rdata;
enable.read(.status(status),
            .value(rdata),
            .parent(this));
```

Figure 3.19: Code for Read Function

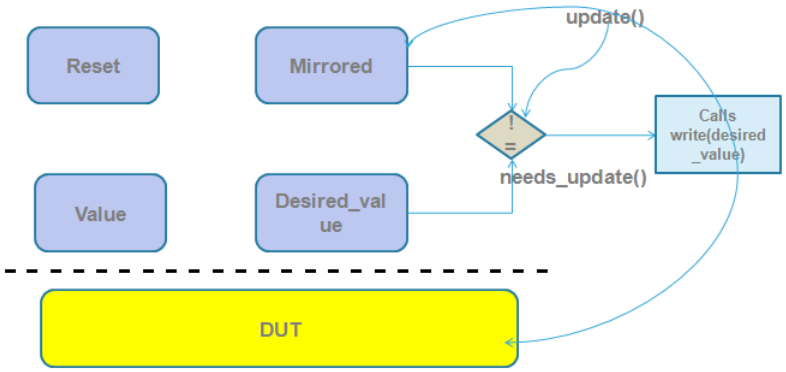

Figure 3.20: Update Value of Register

3. The update() method internally calls the write(.value(m_desired)). Because of this, the value of the m_mirrored will be updated as well, after the update.

- **mirror()** : The mirror() method actually reads a register from the DUT. The differences between the read() method and the mirror() method are:

  1. The read() method returns the register value to the caller, while the mirror() method does not return the register value. The mirror() method only updates the value of the m_mirrored property.

  2. The mirror() method compares the read value against the m_desired if the value of the check argument is UVM_CHECK.

  3.The mirror() method internally calls do_read() method. This is the same method the read() method internally calls. Because of this, the mirror() method will update the value and the m_desired properties, in addition to the m_mirrored property.

Register operations can be summarized as:

1. write, poke, set-update and randomize-update are all active operations which update both the mirrored and DUT register values.

2. read, peek and mirror are all active operations which update the mirrored value based on DUT register values.

3. reset and predict are passive operations which update the mirrored value independent of active model stimulus

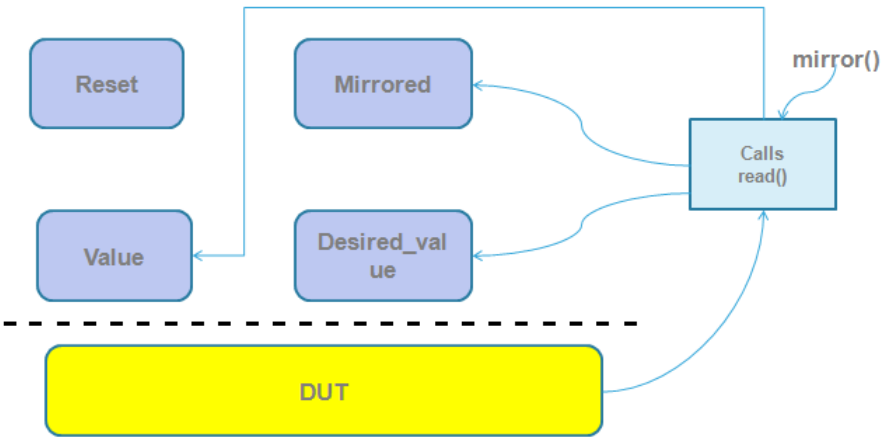

Figure 3.21: Read of Mirror Value

3.3 Coverage Model

Coverage Driven Verification is a methodology where verification goals are defined in terms of functional coverage points. Each area of functionality required to be tested is described in terms of values and events. With this philosophy of verification, everything in the test centers around getting the DUT's functionality to include these values. Coverage collectors are essential for Coverage Driven Verification. A coverage collector is an analysis component that collects coverage information through SystemVerilog covergroups. This information should be stored in a persistent UCDB database. The database is used to determine how much of the overall verification goals have been achieved.

The UVM register library classes do not include any coverage models as a coverage model for a register will depend on the fields it contains and the layout of those fields, and a coverage model for a block will depend on the registers and memories it contains and the addresses where they are located. The UVM register library classes provide the necessary API for a coverage model to sample the relevant data into a coverage model. Functional coverage is implemented using SystemVerilog covergroups. The details of the covergroup (that is, what to make coverpoints, when to sample coverage, and what bins to create) is planned and decided before implementation begins.

Due to the significant memory and performance impact of including a coverage model in a large register model, the coverage model needs to handle the possibility that specific cover groups will not be instantiated or to turn off coverage measurement even if the cover groups are instantiated. Therefore, the UVM register library classes provide the necessary API to control the instantiation and sampling of various coverage models.

Coverage collection for the test is performed by covergroups instantiated inside the

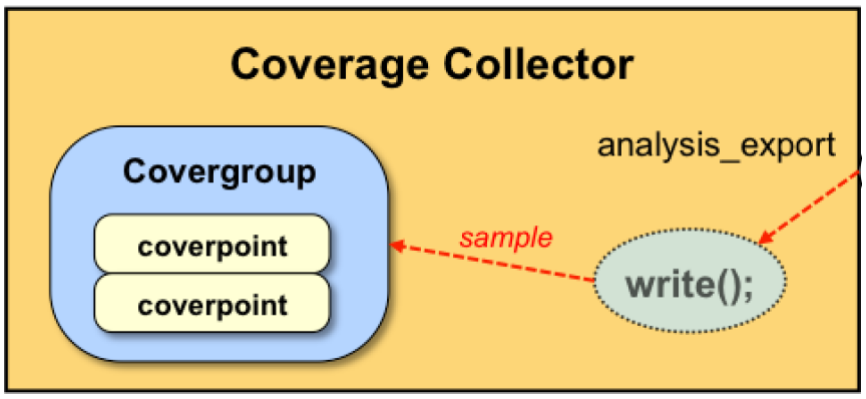

Figure 3.22: Coverage Collector

coverage collector. The covergroup samples the data that has been monitored by the Monitor from interfaces tied to the DUT modules.

Covergroups are built within a register or register block. Coverage is initialised by a build_coverage() call within the constructor of the register or register block. The sample() method is called automatically for each register and register block access. The various methods used to control covergroup build and their effects are summarized below:

For Overall Control

1. uvm_reg::include_coverage(uvm_coverage_model_e) : This is the static method that sets up a resource with the key "include_coverage". Used to control which types of coverage are collected by the register model.

For Build Control

2. build_coverage(uvm_coverage_model_e) : Used to set the local variable m_has_cover to the value stored in the resource database against the "include_coverage" key.

3. has_coverage(uvm_coverage_model_e) : Returns true if the coverage type is enabled in the m_has_cover field.

4. add_coverage(uvm_coverage_model_e) Allows the coverage type(s) passed in the argument to be added to the m_has_cover field.

For Sample Control

5. set_coverage(uvm_coverage_model_e) : Enables coverage sampling for the cover-

```
class example_reg extends uvm_reg;
rand uvm_reg_field example_field;
function new(string name = “example_reg");
super.new(name, 32,
build_coverage(UVM_CVR_ALL));
endfunction: new
virtual function void build( );
...
endfunction: build
`uvm_object_utils(example_reg)
endclass: example_reg
```

Figure 3.23: Example for Coverage

Identifier	Description
UVM_NO_COVERAGE	No coverage models.
UVM_CVR_REG_BITS	Coverage models for the bits read or written in registers.
UVM_CVR_ADDR_MAP	Coverage models for the addresses read or written in an address map.
UVM_CVR_FIELD_VALS	Coverage models for the values of fields.
UVM_CVR_ALL	All coverage models.

Figure 3.24: Coverage Identifier

age type(s), sampling is not enabled by default.

6. get_coverage(uvm_coverage_model_e) : Returns true if the coverage type(s) are enabled for sampling.

3.3.1 Predefined Coverage Identifiers

The UVM library has several predefined functional coverage model identifiers. UVM_NO_COVERAGE specifies no coverage model, UVM_CVR_REG_BITS specifies Coverage models for the bits read or written in registers, UVM_CVR_ADDR_MAP specifies Coverage models for the addresses read or written in an address map, UVM_CVR_FIELD_VALS Coverage models for the values of fields and UVM_CVR_ALL specifies for all coverage models.

```
class base_test extends uvm_test;
...
`uvm_component_utils(base_test)
function new(string name, uvm_component parent);
super.new(name, parent);
uvm_reg::include_coverage("*", UVM_CVR_ALL);
endfunction: new
...
endclass: base_test
```

Figure 3.25: Code to Include Coverage

```
#include_coverage not located
#  did you mean disable_scoreboard?
#  did you mean dut_name?
#include_coverage not located
#  did you mean disable_scoreboard?
#  did you mean dut_name?
```

Figure 3.26: Message to Include Coverage

3.3.2 Controlling Coverage Model Construction and Sampling

By default, coverage models are not included in a register model when it is instantiated. To be included, they must be enabled via the uvm_reg::include_coverage() method. If include_coverage() is not used and coverage is instantiated, simulation displays following message. Furthermore, the sampling for a coverage model is implicitly disabled by default. To turn the sampling for specific coverage models on or off, use the uvm_reg_block::set_coverage(), uvm_reg::set_coverage(), and uvm_mem::set_coverage() methods.

3.4 Integrating UVM Registers in the testbench environment

UVM register models are integrated in the testbench environment by doing the following steps:
1. Build register database by constructing the register blocks in the test and pass their handles to testbench components via configuration objects.

2. Build a register adaption layer; a component to translate register transactions to bus transactions and vice versa. This can be achieved by extending the uvm_reg_adapter base class and providing an implementation for reg2bus() and bus2reg() methods.

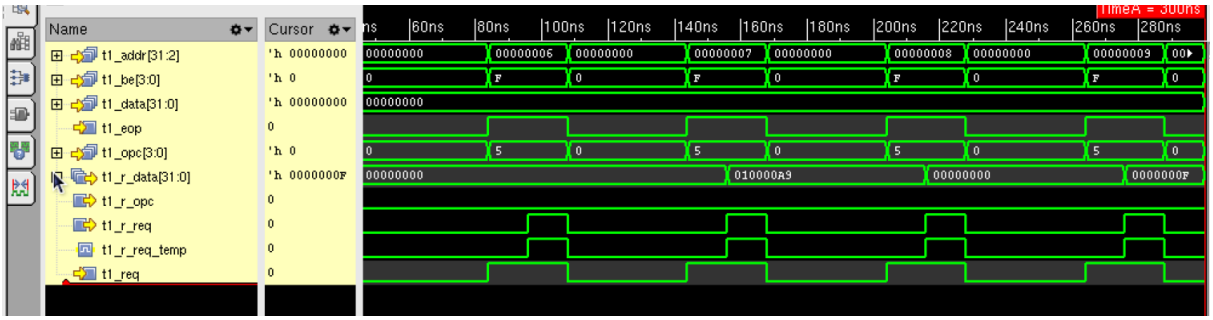

Figure 3.27: Hardware Reset Test

3. Construct the register adapter object in the testbench environment and connect it, as well as the agent sequencer, to the register map via the set_sequencer() method.

4. Build a predictor component acting as a listener on the bus by extending the uvm_reg_predictor class, implementing its write() method. The predictor is used to convert bus transactions to register transactions then update the corresponding register model, or if desired compare the register model value to the actual hardware register value.

5. Construct the predictor object in the testbench environment, and connect it to the bus agent monitor analysis port using normal UVM Transaction Level Modeling (TLM) analysis port connections.

3.5 Built-in Sequence

3.5.1 REG-MODEL

Built-in register test cases allow user to execute pre-defined register testcases. The uvm_reg package has built-in sequence library.
The basic built-in sequences are as follow:

1. uvm_reg_hw_reset_seq: This is used to test the hard reset value of the register. Here it first resets the DUT and then, reads all the register in the block via all the available address map and check their value with the specified reset value. If NO_REG_TEST or NO_REG_HW_RESET_TEST bit type resource is specified in REG:: namespace then that block or register is not tested.

2. uvm_reg_single_bit_bash: This is used to verify the implementation of single register by writing 1s or 0s to all the bits via address map, checking whether it is correctly set

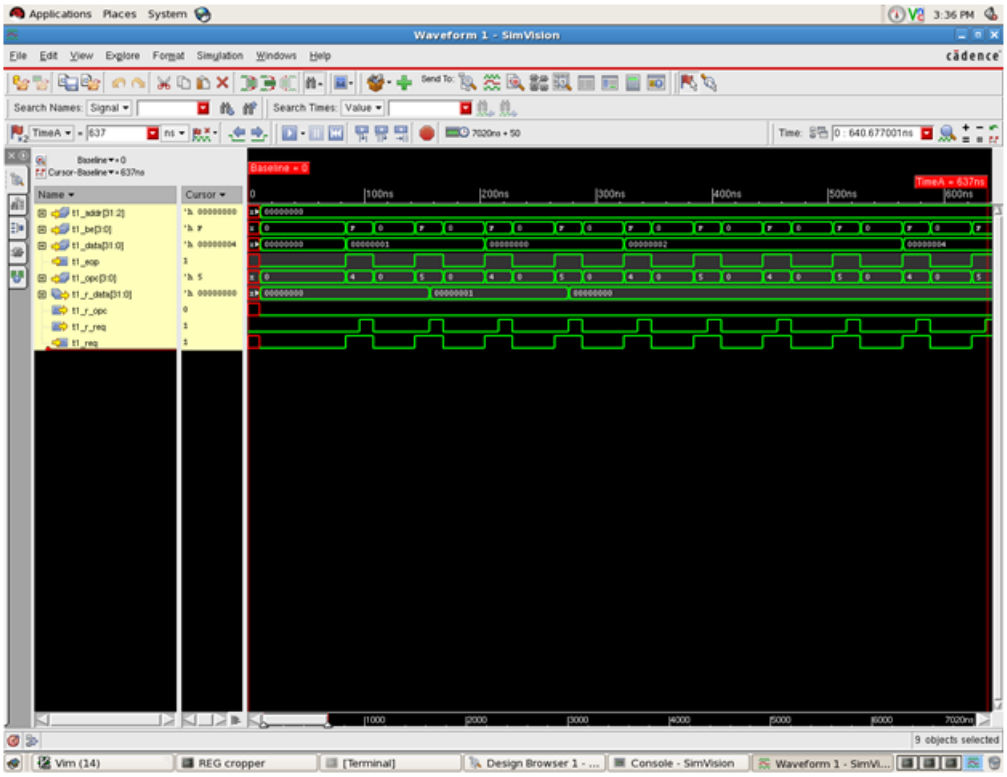

Figure 3.28: Reg bit bash test

or cleared, based on field access policy specified for field containing the target bit. If NO_REG_TEST or NO_REG_BIT_BASH_TEST bit type resource is specified in REG:: namespace then that register is not tested.

3. uvm_reg_bit_bash: This test verifies the implementation of every register in the block by executing uvm_reg_single_bit_bash sequence on it. If NO_REG_TEST or NO_REG_BIT_BASH_TEST bit type resource is specified in REG:: namespace then that block or register is not tested.

4. uvm_reg_single_access_seq: This test is used to verify the accessibility of the register. First it writes to register then reads the value via back-door so as to confirm that the value was written correctly. And similarly it writes through backdoor and the read the value of the register via address map so as to confirm the accessibility of the register. If NO_REG_TEST or NO_REG_ACCESS_TEST bit type resource is specified in REG:: namespace then that register is not tested. Also those register with no back-door or those with read-only field or with unknown access policies cannot be tested.

5. uvm_reg_access_seq: This test is used to verify the accessibility of all the regis-

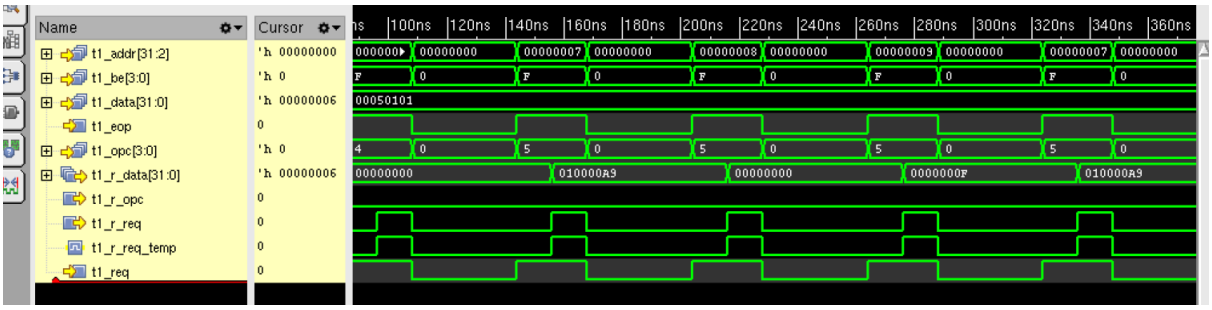

Figure 3.29: Reg Aliasing Sequence

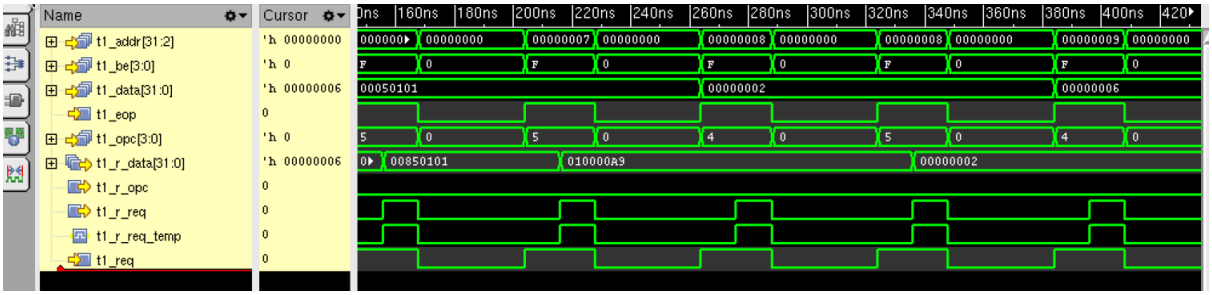

Figure 3.30: Reg Write Follow Read Sequence

ter in a block by executing uvm_reg_access_seq on all register. If NO_REG_TEST or NO_REG_ACCESS_TEST bit type resource is specified in REG:: namespace then that block or register is not tested.

Additional built-in test contributed for uvm_reg Package by Cadence (so as to be similar sequence as in uvm_rgm package)are as follow:

1. uvm_reg_built_in_aliasing_seq : Goes through all the registers, writes a random value to a register and reads all the other registers to make sure the write did not affect them. It does this for all selected registers inside container.

2. uvm_reg_built_in_write_all_regs_seq : Writes all registers inside register-block while ignoring constraints (fully random value).

3. uvm_reg_built_in_read_all_regs_seq : Reads all registers inside a register-block.(Usually this sequence is used for reset checks)

4. uvm_reg_built_in_wr_follow_rd_seq : Writes a constrained random value to the register and reads back to make sure the value is written correctly. The sequence does this for each register in the register block.

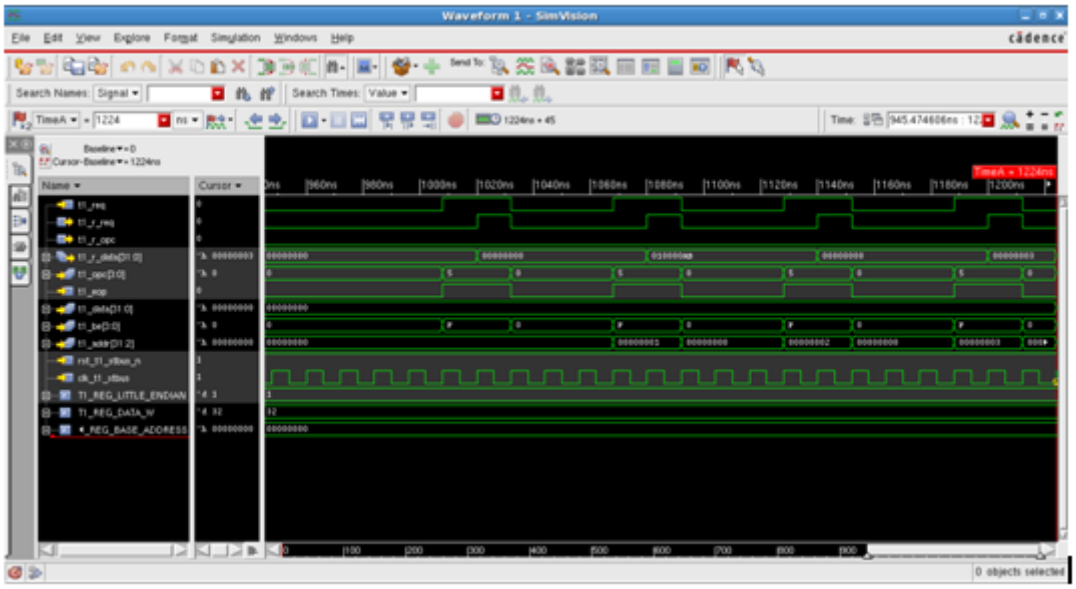

Figure 3.31: Read all Sequence

3.5.2 RGM-MODEL

Built-in register test cases allow user to execute pre-defined register testcases. The uvm_reg package has built-in sequence library.
The basic built-in sequences are as follow:

List of Built-In Sequences
1. uvm_rgm_read_all_reg_seq Reads all registers inside a container.(Usually this sequence is used for reset checks)

2. uvm_rgm_write_all_reg_seq Writes to all registers inside container. The user can chose to write directed or controlled random value (register constraints are followed).

3. uvm_rgm_any_write_all_reg_seq Writes all registers inside container while ignoring constraints (fully random value).

4. uvm_rgm_wr_rd_all_reg_seq Writes a constrained random value to the register and reads back to make sure the value is written correctly. The sequence does this for each register in the container.

5. uvm_rgm_walking_one_zero_seq Writes a pattern (0xffff_fffe in case of walking zero and 0x0000_0001 in case of walking one) to a register, and checks that the value is correctly written. It then right shifts the pattern by 1 and writes to the register followed by a read to check the value. It does this for all selected registers inside container.(We

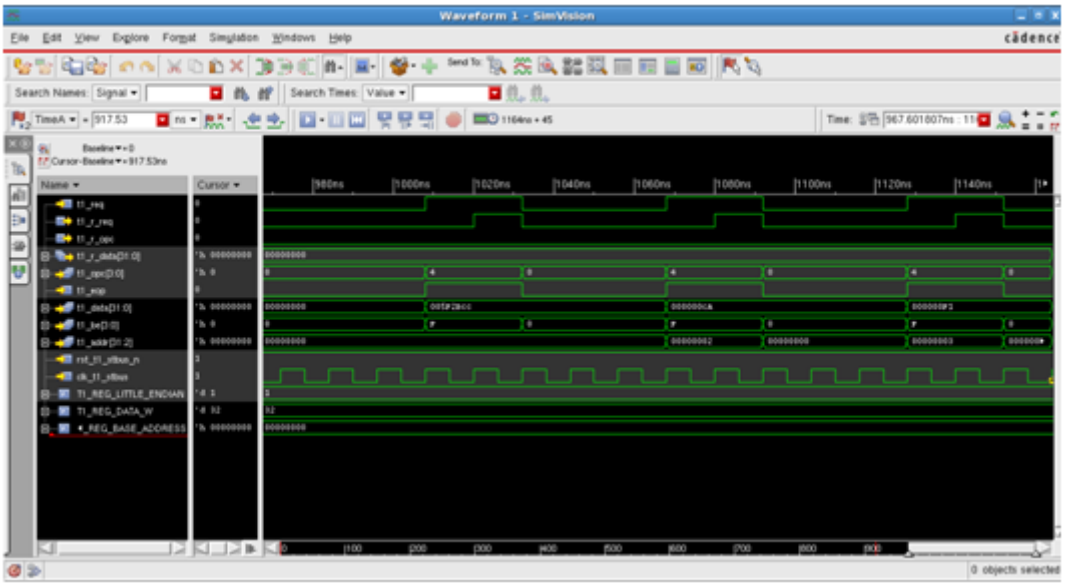

Figure 3.32: Any write Sequence

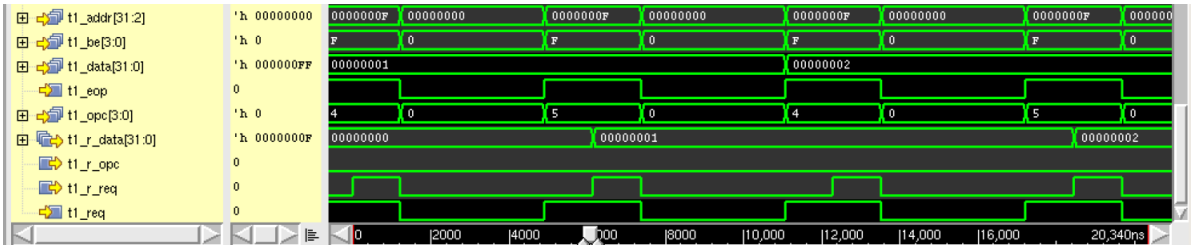

Figure 3.33: Walking one-zero Sequence

are using this sequence for bit bashing).

6. uvm_rgm_aliasing_seq Goes through all the registers, writes a random value to a register and reads all the other registers to make sure the write did not affect them. It does this for all selected registers inside container.

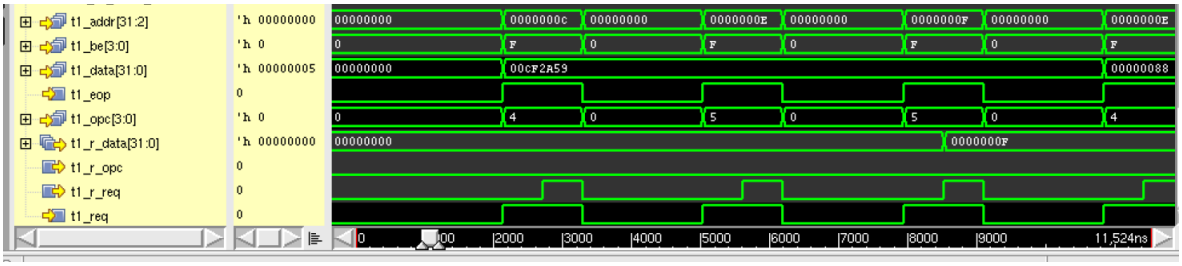

Figure 3.34: Aliasing Sequence

Chapter 4

IPXACT Flow

This chapter provides the overview of the SPIRIT script and IP-XACT flow. Then, IP-XACT flow used in imaging group is described. The three scripts (spec2verilog, spirit2uvm, spec2uvm) that are used to automatically generate UVM Verification Environment file are described in detail.

4.1 Introduction

SPIRIT stands for Structure for Packaging, Integrating and Re-using IP within Toolflows. It is Standard based on XML open format. IPXACT is developed by The SPIRIT Consortium (http://spiritconsortium.org). An IP-XACT description of a design or component consists of a set of XML documents referring to one another. It is a description of components and designs written in a standard data exchange format (XML), which is both machine process-able and human readable. It describes electronic system designs and the interconnection of IP interfaces in a standard specification to provide IP suppliers, design integrators and Electronic Design Automation (EDA) vendors, with a common representation. In this way, IP-XACT provides a mechanism by which design reuse has been made a practical reality.

Accellera IP-XACT standard is used for capturing register's specifications of IP/SoC. The code generators can be used(that uses IP-XACT standard) for generation of the SV-UVM code for the verification environment at IP level as well as for SoC level verification. IPXACT flow is so designed to fit the requirement of the designer expecting to reduce the time-to-market. IPXACT flow is independent of the design language and design tool and is also very efficient.

In System Verilog UVM based Verification Environment, register description file for

```
- <spirit:memoryMap>
    <spirit:name>memory_map_name</spirit:name>
- <spirit:addressBlock>
    <spirit:name>regfile_name</spirit:name>
    <spirit:baseAddress>0</spirit:baseAddress>
    <spirit:range>16</spirit:range>
    <spirit:width>8</spirit:width>
- <spirit:register>
    <spirit:name>register_name</spirit:name>
    <spirit:addressOffset>0x0</spirit:addressOffset>
    <spirit:size>8</spirit:size>
    <spirit:access>read-write</spirit:access>
- <spirit:reset>
    <spirit:value>0x00</spirit:value>
    <spirit:mask>0xff</spirit:mask>
  </spirit:reset>
- <spirit:field>
    <spirit:name>field_name</spirit:name>
    <spirit:bitOffset>0</spirit:bitOffset>
    <spirit:bitWidth>2</spirit:bitWidth>
    <spirit:access>read-write</spirit:access>
- <spirit:values>
    <spirit:value>0</spirit:value>
    <spirit:name>LEFT</spirit:name>
    <spirit:description>0</spirit:description>
  /spirit:values>
- <spirit:values>
```

Figure 4.1: Example of XML file

register model, address map file, sequences file, functional Coverage file, data checker file to compare the output of RTL with output of Reference(Python) model are IP/SoC specific which need to be modified for every IP/SoC. Therefore, IP-XACT based tools are used for generation of these files and hence reduce the quality amount of time writing these files.

XML-based IP-XACT view is automatically generated from the Register Specification Document. In Data checker file, there is invocation of executable of Python model containing attributes thus; automatic generation of data checker file requires the mapping between the registers/register-fields/parameters of RTL and the attributes of Python model.

The structure of the IP-XACT input file consist of the following:

- A header with some document detail such as vendor name, SPIRIT IP-XACT version, project name, and so on
- An array of memory maps that specifies the memory and registers description. Each memory map is an address space that can hold memory blocks register-files, registers, or fields.

IP-XACT is not complete for register automation needs. Some of the missing capabilities include constraints, coverage directives, backdoor paths, and so on. That said, the standard supports extensions to enable the missing functionality.

```
<spirit:vendorExtensions>
  <vendorExtensions type>ua_cr_c</vendorExtensions:type>
  <vendorExtensions hdl_path>ua_cr_reg</vendorExtensions:hdl_path>
  <vendorExtensions:constraint>c1 {tx_en!= value.rx_en;}
    </vendorExtensions:constraint>
  </spirit:vendorExtensions>
```

Figure 4.2: IP-XACT Vendor Extensions in XML file

4.2 IPXACT Flow

To generate the System-Verilog files for the verification environment, spirit2uvm script is used. spirit2uvm script takes XML file as its input. XML file is just used as a structure to store and transport information. The major advantage with XML file is that we can have our own tags, as XML has no pre-defined tags. The XML file description complexity depends on the complexity of the IP. The IP-XACT description contains register descriptions, address block and memory map description. Therefore XML file complexity fully depends on the complexity of the IP to be verified. And so the generation of these XML file should be automated in order to reduce the time-to-market and hence practically error free file. The XML file is so generated from the script spec2verilog which reads the .docx file or .mif file (whichever available) of the IP. The .docx/.mif file of the IP is provided by the designer to the IP verification team. The .docx file includes the description of the IP and the description of the register access policies and address and memory mapping information which is then read by spec2verilog. IP-XACT flow provides the automation in setting up the verification environment and thus reducing the Time-to-Market. The spirit script are described below in detail.

4.2.1 spec2verilog

spec2verilog script generates the Xml files and then Register_bank files of the particular IP . This will be generated from the Specification file (mif/docx) of that particular IP taken as input to the script. The script Generates xml file and this generated xml file is used by spec2verilog script to generate register description file for register model, sequences file, functional Coverage file, data checker file which are IP dependent.

To run spec2verilog script following command is used:

spec2verilog.sh $-file < file-name > .mif/docx[-out < XML_FileName > .xml][-log < logName > .log][-inter][-version]$

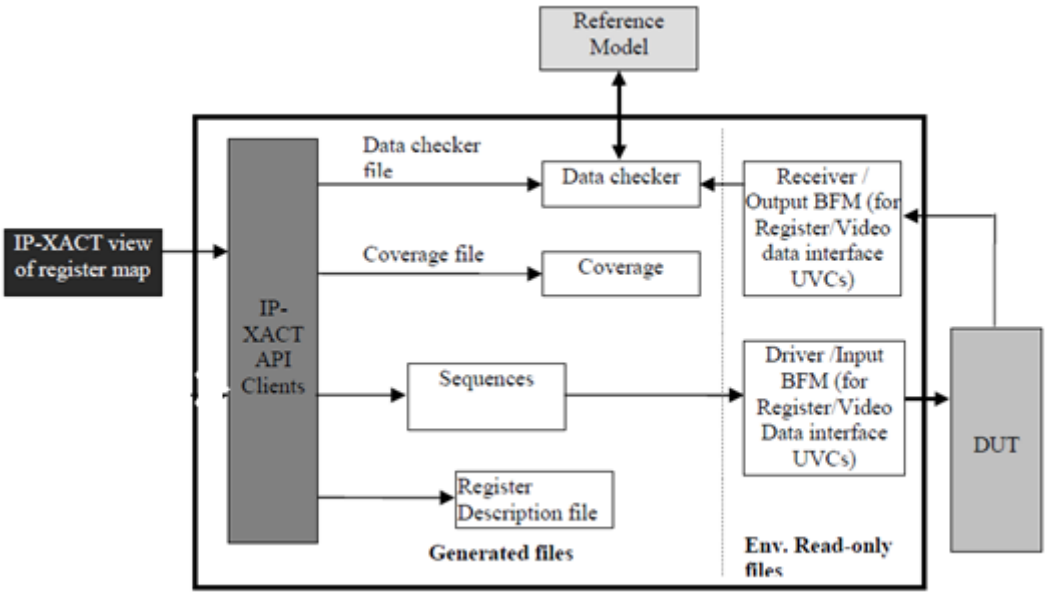

Figure 4.3: IP-XACT flow

spec2verilog converts a spec (.docx/.mif) file describing registers into :

1. Four verilog register banks (8-bits big-endian, 8/16/32-bits little endian T1 data bus)

2. Four verilog register banks (8-bits big-endian, 8/16/32-bits little endian T1 data bus)

3. Two Verilog header file containing respectively registers offsets and registers values

4. Two Corresponding C header file

Desciption of Options supported for spec2verilog are:

1. **file <file-name><.mif or .docx>**

Give the input specificstion file in .mif or .docx format.

2. **out <XML-FileName>.xml**

Define the name of the XML file which will contain the Spirit description of the register bank.
(Default : filename.xml)

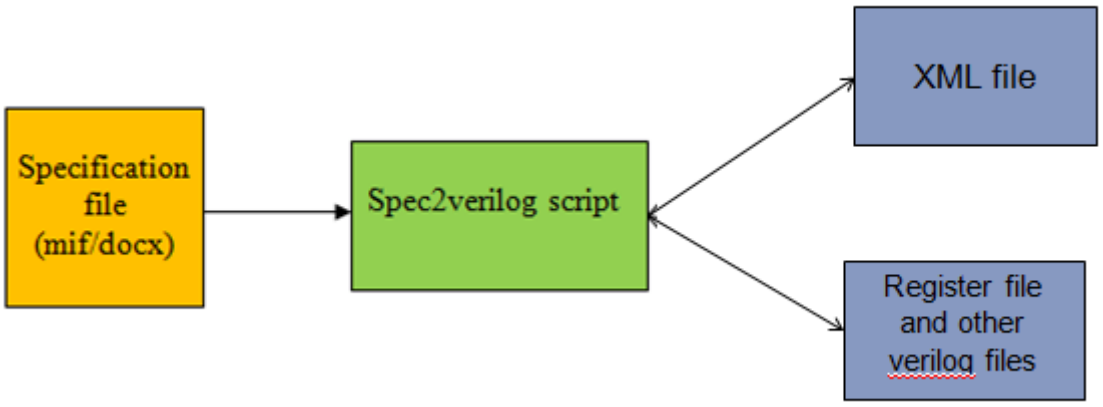

Figure 4.4: spec2verilog flow

3. **log <log-fileName>.log**

Define the name of the log file generated by the .MIF parser
(Default: display on screen)

4. **inter**

Full script becomes interactive (user prompt) and step-by-step process.
(Default: not interactive)

5. **version**

Displays the version of each internal tool (ds2spirit, spirit2verilog, ...)

4.2.2 spec2uvm

spec2uvm script is using the spec (mif/docx) file and generating System Verilog files. spec2uvm.csh script internally calls spec2verilog.sh script to generate the xml file and then calls spirit2uvm script to generate the System Verilog files in the vrad_verif directory using generated xml file. spec2uvm.csh script should be run from verif directory and it will also create the tests, VC, test_bench etc. sub-directories in the verif directory. Generated System Verilog files in $< output\_dir >$ will be copied into the proper sub-directories in the verif directory.

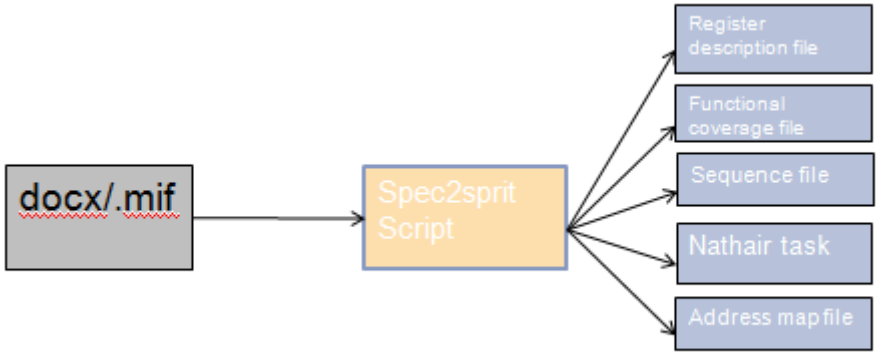

Figure 4.5: spec2uvm flow

4.2.3 spirit2uvm

The xml file which is been generated from spec2verilog will be used as input file here. The xml file and the map file will be used as input files for running spirit2uvm script and the output files which will be generated will be Register Definition Files, Functional Coverage files, Sequence Files, Nathair files, Address Map Files. Following are the generated file names with there naming conventions:

IPXACT flow is so designed to fit the requirement of the designer expecting to reduce the time-to-market time. The spirit2uvm script generates the register description files along with sequences file, functional Coverage file, and data checker file.

1. **IP_*_uvm_reg_def file**
2. **IP_*_seq_lib file**
3. **IP_*_derived_seq_lib file**
4. **IP_*_addr_map file**

To run this spirit2uvm script following Command Line Options are used:

$< tool-path > /bin/spirit2uvm.sh$ -input_spirit_file <input-spirit-file-path> [-output_dir output-directory][-output_files <Output-file-selection>] [-invalid_address option][-relax_checks option][-unique_mmaps option][-uvm option][-h help][-v version]

Following options are supported in spirit2uvm.

1. **-input_spirit_file <Input-spirit-file-path>**

IPXACT file with absolute or relative path will be used as an input parameter to

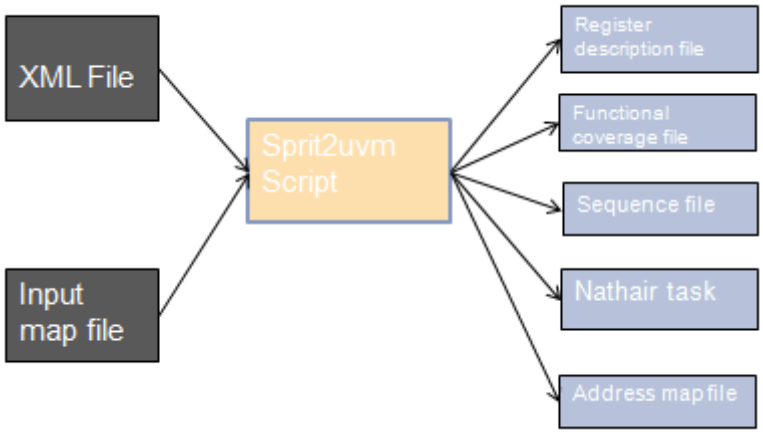

Figure 4.6: sprit2uvm flow

generate verification files. It is a mandatory parameter.

2. **-output_dir <Output-directory-Path>**

Specify path of the directory where output files will be stored. In case of absence of this parameter, vrad_verif directory will be created in the current working directory and all output files are stored in this directory. If the specified directory does not exist then it will be created.

3. **-output_files <Output-file-selection>**

Output file selection option provides control over generation of output files in the specified destination directory. User can generate only required files as needed. Input to this option is given as names of files, colon separated inside quotes.

4. **-invalid_address Invalid address generation**

Invalid address option generates a file which contains a set of registers which address lies outside the valid address boundary, specified by memory map of register bank (valid for rgm files).

5. **-relax_checks Option to relax checks**

Relax checks option with 1 value allow tool to relax certain register level checks while building internal data structure and subsequent output file generation.

6. **-unique_mmaps Unique memory maps generation**

This option is useful when input IPXACT file has more than one memory maps. User may avoid naming conflict by selecting 1 as an argument. In case of 0, tool will not change name of the conflicted memory maps during systemVerilog file generation. By default, argument 1 is used. It is an optional argument.

7. **-uvm Option to generate systemVerilog files**

This option controls format of output verification files. Option 1 is used to generate systemVerilog files and 0 is used to generate e files. By default, option 0 is used by the tool. It is an optional argument.

8. **-h help**

User can access the information of scripts option usage through this option.

9. **-v version**

User can know current tool version through this option.

Chapter 5

Work at ST

In this chapter the work done is explained in detail. first Migration from use of UVM_RGM register and memory package to UVM_REG register and memory package is described. Then the modification done to the spirit2uvm script (IPXACT script) is explained with its usage. Also the modification to the FEKIT (internal Tool) GUI is shown.

5.1 Why to Migrate from UVM_RGM to UVM_REG

In the Above chapter 3 of Register model the Architecture of the Register model is described in detail. UVM_REG register and memory model provides various features to ease verification of register's of DUT and hence automate register related activities. Compared to UVM_RGM register and memory model, it provides enhanced features of having automatic comparsion on use read and write API, Also provides support for little endian and big-endian feature where as in UVM_RGM, externally support had to be given. UVM_REG supports all ipXact 1.5 access policies as well. Also as UVM REG is given by Accellera it is suppose to be updated with upgrading of UVM itself. This Above advantage of UVM_REG over UVM_RGM makes one want to migrate from use of UVM_RGM register and memory model to UVM_REG register and memory model.

The UVM_REG register and memory package was introduced in release of Accellera's UVM 1.0 release for easier verification of register's of DUT and automate register related activities. The UVM_REG register and memory package derives the register level API from VMM and the use model, register sequenced, register operation items, layering concepts, and more from the UVM_RGM register and memory package contributed to UVM World by Cadence. The use of UVM_REG register and memory package has been proven more satisfactory than use of UVM_RGM register and mem-

ory package, and hence migration from UVM_RGM register and memory package to UVM_REG register and memory package, becomes necessary for more efficient verification of register and memory of DUT.

5.2 Writting the Basic Register Model Architecture

The Register definition class is derived from the base-library register class and the instance of this register definition class is used in register-file class which is extended from base-library register-file class. and then the instance of this register-file class is used in register-block class which is extended from base-library register-block class. and then finally a register map class(extended from base-library register-map class) is defined which has instance of the register-block class.

Both the Register Model has register definition that includes the declarations of the register field with its access policies of field, position of field in the register and the size of field. The difference lies in the way they are defined. Also in RGM model the unused bits of the register are defined but in REG model there is no need of defining the unused bits of the register. There is no pre-defined field access policy for reserved fields. Reserved fields should be left unmodelled (where it will be assumed to be RO fields filled with 0s. The RGM model has register definition class extended from the uvm_rgm_sized_register class and the fields are defined in a packed struct, and the REG model has register definition class extended from the uvm_reg class and the fields are not defined in a packed struct. The figure 5.1 shows the how to define the basic register field for both RGM and REG model. The register definition in RGM model shown has $example\_type$ register definition with $example\_field\_type$ and $unused\_1$ register field whereas its equivalent structure for REG model is shown where only $example\_reg\_field$ is defined.

The register-level coverage model is always defined and instantiated in the register type class. Coverage is taken for read and write operations for field values on each instance of that register type. The uvm_reg::sample() is used trigger the sampling of a coverage point as mentioned in covergroup of the register type instance. The sampling of the coverage model in the register definition file is done only if uvm_reg::get_coverage() method returns 1. All the coverage models that are included

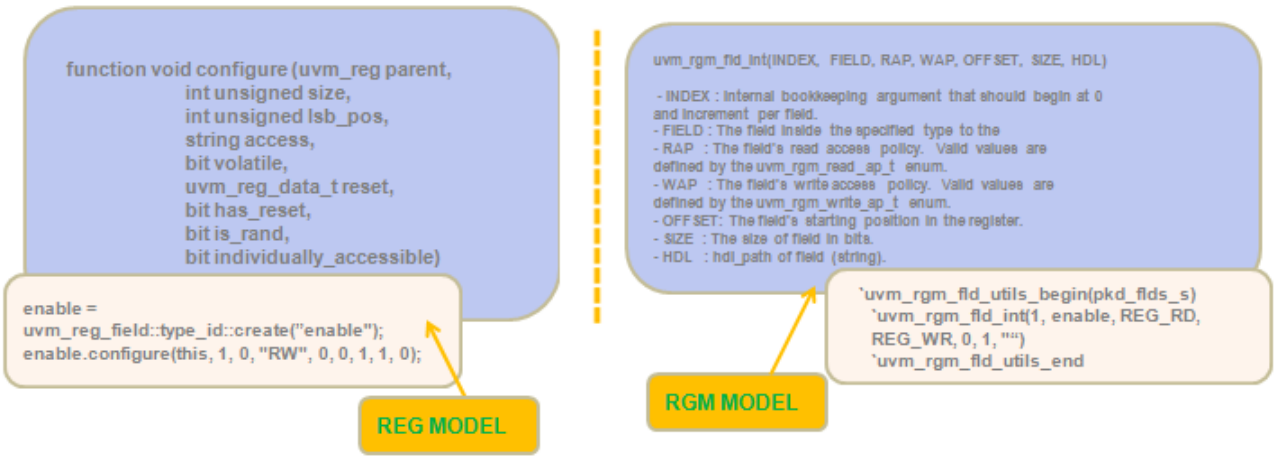

Figure 5.1: Register Field Definition

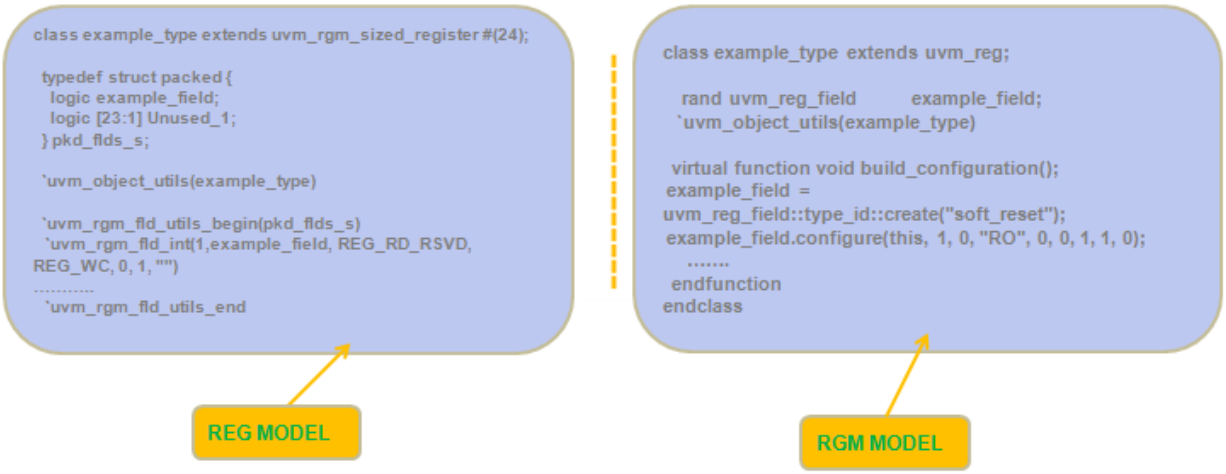

Figure 5.2: Register Definition

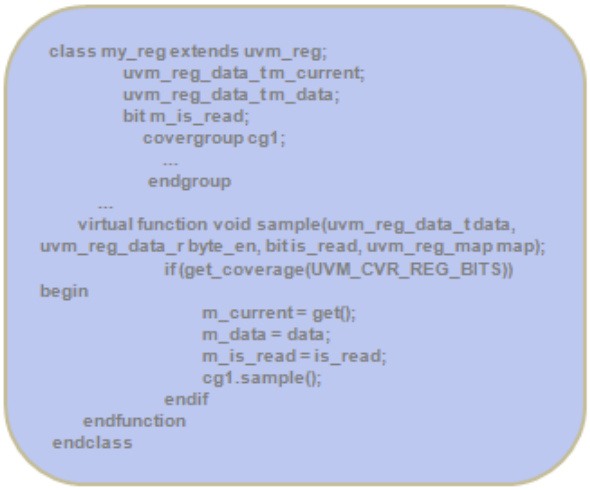

Figure 5.3: Sampling Covergroup

Figure 5.4: Constructing Covergroup

in the environment are build using the uvm_reg::build_coverage() method when super.new() is called or the uvm_reg::add_coverage() method. So the construction of coverage model is done if the uvm_reg::has_coverage() method returns 1.

The instance of register definition class is instantiated in register file. The instance of register definition class can be instantiated directly to the register block but it is recommended to use the reg-file hierarchy in the register model. Register types are instantiated in the build() method of the register file types. The constructor uvm_reg::new() method is called with appropriate argument values for the register type. In REG Model Register files can have other register files as well. The build() method calls the configure() method for all register and register file specifying get_block() for the parent block and "this" for the parent register file. The map() method calls uvm_reg_map::add_reg() for all register class properties, adding the value

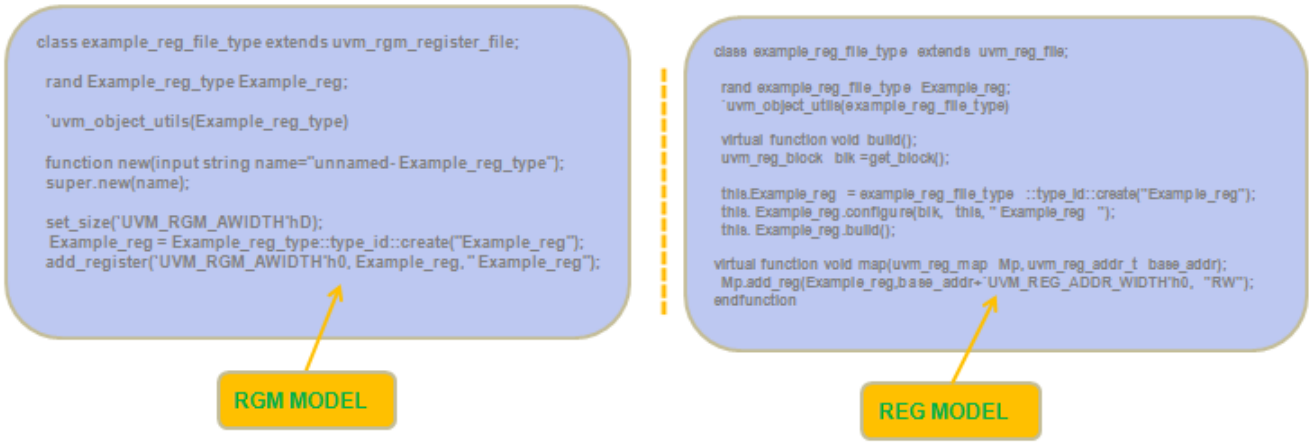

Figure 5.5: Register file Definition

of the address offset argument to the offset of the register in the register file. The register file definition class for both RGM model and REG model is shown in figure 5.5

The register block is created in the case of REG model which can have instance of register or/and the register file instance or/and memory instance or/and another register block inside a register block. While in case of the RGM model there is rgm_map that contains instance of the register file and/or memory block. A block-level coverage model is defined and instantiated in the block type class.The uvm_reg_block::sample() methods is used to trigger the sampling of a coverpoint, based on the data provided as an argument. The sampling of the coverage model is done only if sampling for the corresponding coverage model has been turned on so that the uvm_reg_block::get_coverage() method returns 1. The memory model for both models are shown in figure 5.7 and the register block for both model is shown in figure

Test sequences defined needs a verification environment to get executed. here in this environment the register model needs to be instantiated so as to be used by the tests to access registers and memories in the DUT. The register model is explicitly built by calling its build() method, then it calling the uvm_reg_block::lock_model() method. The register model is integrated with the bus agents that perform and monitor the actual read and write operations. Implicit prediction and Explicit prediction any one can be used for updating the model with that of the DUT. Implicit prediction only requires the integration of the register model with bus sequencers. After completion of the read, write, peek, or poke operations it would automatically update the register model. This will lead to update register model for all observed bus opera-

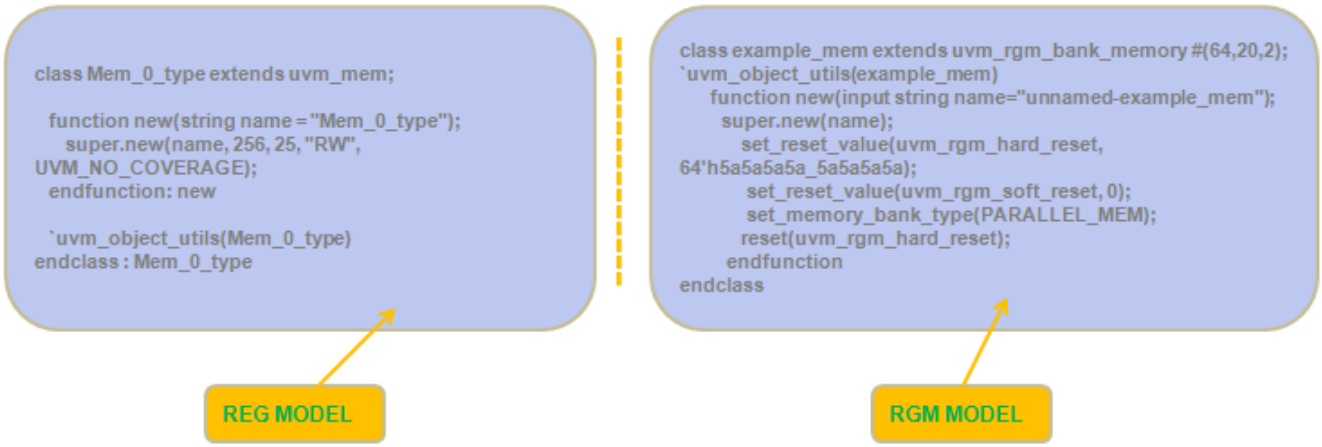

Figure 5.6: Memory Definition

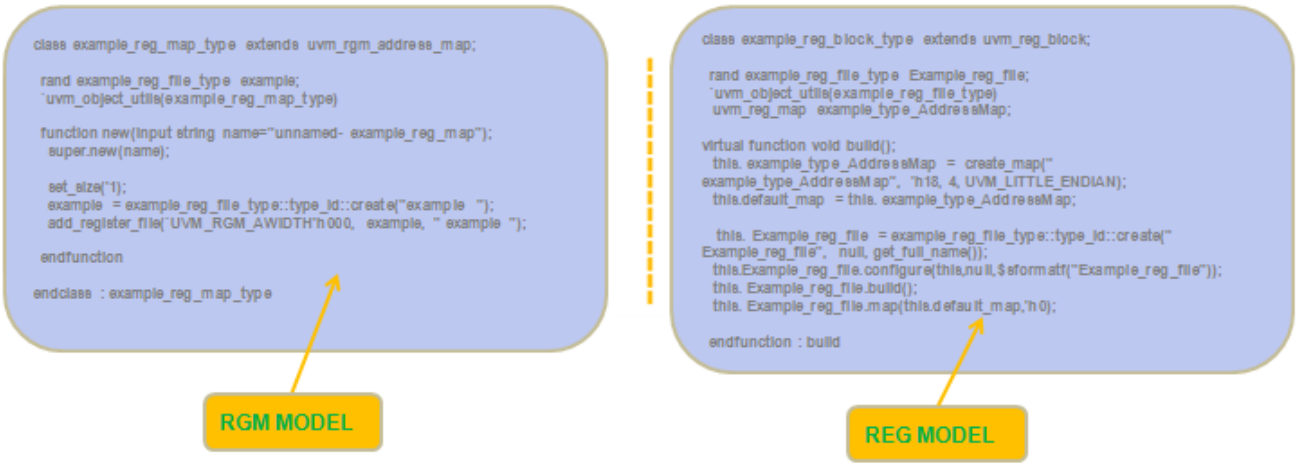

Figure 5.7: Register Block

```
class my_blk extends uvm_reg_block;
      covergroup cg_vals;
        ...
      endgroup
  ...
    virtual function void sample_values();
      super.sample_values();
      if (get_coverage(UVM_CVR_FIELD_VALS))
          cg_vals.sample();
    endfunction
  ...
endclass : my_blk
```

```
class my_blk extends uvm_reg_block;
    covergroup cg_vals;
      ...
    endgroup
  ...
    function new(string name = "my_blk");
      super.new(name,
build_coverage(UVM_CVR_FIELD_VALS));
      if (has_coverage(UVM_CVR_FIELD_VALS))
        cg_vals = new();
    endfunction: new
  ...

endclass : my_blk
```

Figure 5.8: Register Block Coverage

```
class top_env extends uvm_env;
    block_reg_model regmodel;
    virtual function void connect();
      ...
        if (regmodel.get_parent() == null) begin
          reg2apb_adapter reg2apb0=
reg2apb_adapter::type_id::create("reg2apb",,get_full
_name());
          ...
        end
    endfunction
endclass
```

```
class top_env extends uvm_env;
    block_reg_model regmodel;

    virtual function void build();
      if (regmodel == null) begin
          regmodel =
block_reg_model::type_id::create("regmodel", this);
          regmodel.build();
          regmodel.lock_model();
      end
    endfunction
endclass
```

Figure 5.9: Top-Env Register Model Connection

tion originate from register model and not others. The explicit prediction updates the register model for all observed operation. Here implicit prediction is turned off and the uvm_reg_predictor component is used. The predictor receives the bus operations observed by a connected bus monitor, and thus appropriately update the corresponding register in model. The adapter implemented is used by the predictor to know which register transaction had occured. The adapter is implemented by extending the uvm_reg_adapter class and ii has reg2bus() and bus2reg() methods to convert the register transaction to the bus operation and to convert bus operation to register transaction respectively. The adapter is instantiated in the connect() method of the top environment. In the connect_phase function, the sequencer and the register adapter of the agent are associated with the reg_map by calling set_sequencer function . The set_sequencer() is called before starting any sequences based on uvm_reg_sequence.

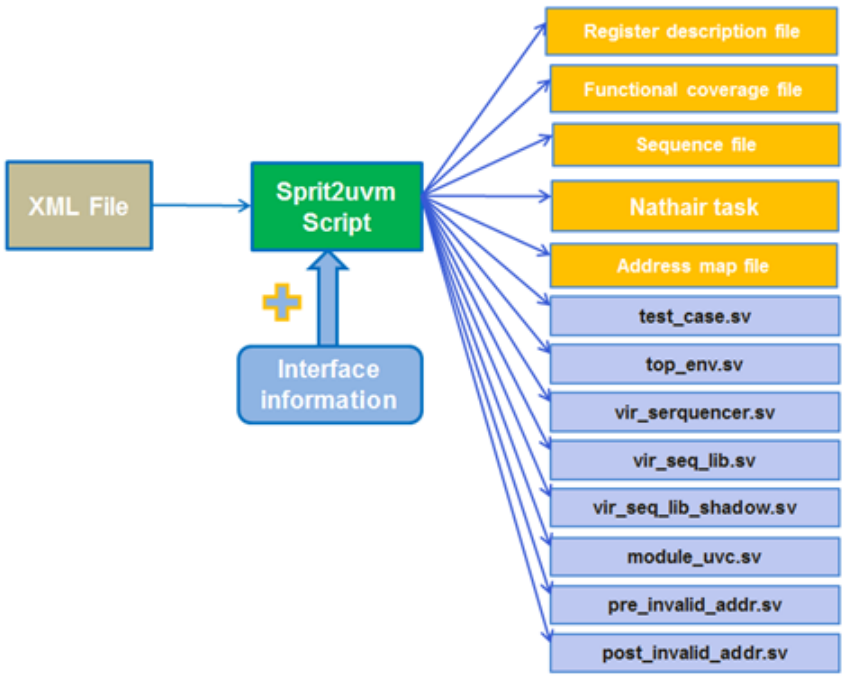

Figure 5.10: New sprit2uvm flow

5.3 Introduction to IPXACT

Although register models could be built-up manually, as described above but typically register models are automatically generated using IPXACT flow for register model generators, which not only prevents manual coding errors but also reduce the quality amount of time writing these files.

The chapter-4 (IPXACT flow)describes the spirit2uvm script is used for automatic generation of the SV-UVM environment files through xml specification file of the IP/subsysytem/SoC. There are many other files to be written by the verification engineer so as to set the verification environment. To write these files we need interface information i.e the type of interface used and the no of input and output interfaces needed. The spirit2uvm script is modified to take extra input argument of interface information and hence generate the remaining files to set the verification environment. The flow for this script is shown below. The modified spirit2uvm script generates the register description files which has register and memory model in the hierarchy that has reg_file and hirarchy without reg_file for UVM_REG register Model generation where as for UVM_RGM regiser definition file has hierarchy of rgm_file in register model Also the memory block specified in xml is now readable through script and register-definition file builds the memory block as specified, sequences file where the standard sequences like read-write sequence, read sequence and write sequences are defined, functional Coverage file which is the extended file for register definition file

where the coverage is defined in a standard format also this can be easily manipulated by the user so as to have the specific coverage scenario , and data checker file which is the nathair file that generates the python script for the input to the model, along with these above mentioned files extra files that modified spirit2uvm generates are invalid register map file that is used as an definition file for the dummy pre and post invalid regiser above and below the valid range of registers, test-case file where the actual test case scenarios are written here only the register built-in sequence test are provided along with one streaming test-case is generated, top-environment file, virtual sequence file where the sequences that are defined in sequence file are created and send to the sequencer, virtual sequencer file is where the sequence are started and module-uvc file used for score-boarding are all generated by the spirit2uvm script.

To run this spirit2uvm script following Command Line Options are used:

$< tool-path > /bin/spirit2uvm.sh$ -input_spirit_file <input-spirit-file-path> [-output_dir output-directory][-output_files <Output-file-selection>] [-invalid_address option][-relax_checks option][-unique_mmaps option][-uvm option][-rgm option][-no_of_input_interface option][-no_of_output_interface option][-type_of_input_interface option][-no_of_output_interface option][-h help][-v version]

Following options are extra options that is now supported in spirit2uvm.

1. **-rgm** $< 0 - 1 >$

This option gives the support for generating the rgm files as well as reg files. if given 1 it generated the rgm files and if given 0 it generates the reg files.

2. **-no_of_input_interface integer**

Gives the no of input interface for the DUTBy default it generates file for having 1 input interface.

3. **-no_of_output_interface integer**

Gives the no of output interface for the DUT. By default it generates file for having 1 output interface.

4. **-type_of_output_interface integer**

Gives the type of output interface for the DUT . By default it generates IDP type output interface.

5. **-type_of_input_interface integer**

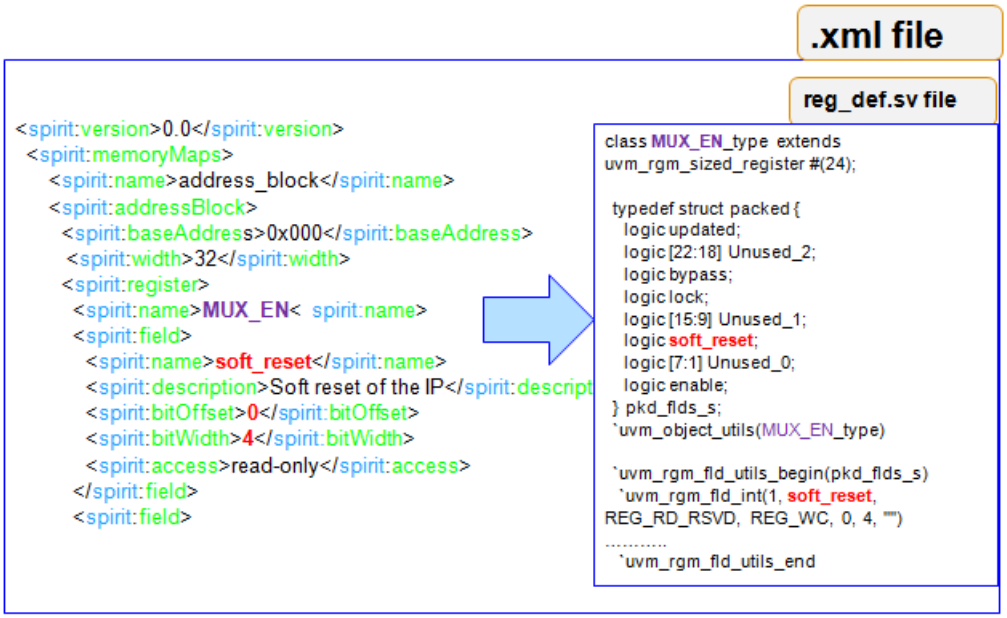

Figure 5.11: XML to reg def (RGM MODEL)

Gives the type of input interface for the DUT. By default it generates IDP type input interface.

6. **-reg_file**

This option takes information on whether or not to generate the reg_file hierarchy in register model for UVM_REG model this is not needed in UVM_RGM model generation. Default it generates the register model with reg_file hierarchy.

7. **-invalid_address Invalid address generation**

Invalid address option generates a file which contains a set of registers which address lies outside the valid address boundary, specified by memory map of register bank (valid for rgm and reg files).

The below given figure shows sample example of input .xml file and output reg_def file. The .xml file gives all the basic information of the IP such as IPs name (<spirit:name>MUX_EN<spirit:name>), data bus size (<spirit:width>32</spirit:width>), base address (<spirit:baseAddress>0x000</spirit:baseAddress>), etc. also the register definition with its fields such as register bit width

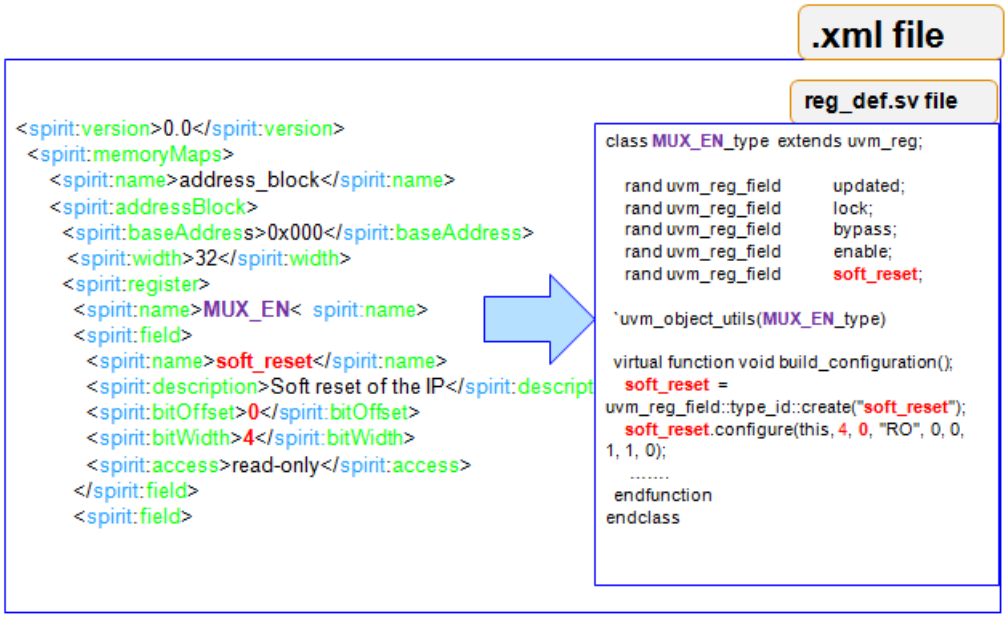

Figure 5.12: XML to reg def (REG MODEL)

(<spirit:bitWidth>4</spirit:bitWidth>), register offset bit (<spirit:bitOffset>0</spirit:bitOffset>), register accessibility (<spirit:access>read-only</spirit:access>) are provided. From the above given information corresponding ref_def file is generated.

5.4 Generating Test-bench

The IP-XACT tool mentioned above is used to generate the SV-UVm environment files using the specification(.mif/.docx/.xml). The testbench file is written for connecting to the DUT so needs the RTL file the write the test-bench. To ease the work of user the testbench file is generated using the RTL file. The Python script is written to read the rtl file also with the information of interface and address width used the script generates test-bench file.

To run this test-bench-python script following Command Line Options are used:

python $< tool{-}path > script.py < interface\_type > < address\_width > < rtl\_file\_path > < Register\_model >$

Following options are extra options that is now supported in test-bench-python script.

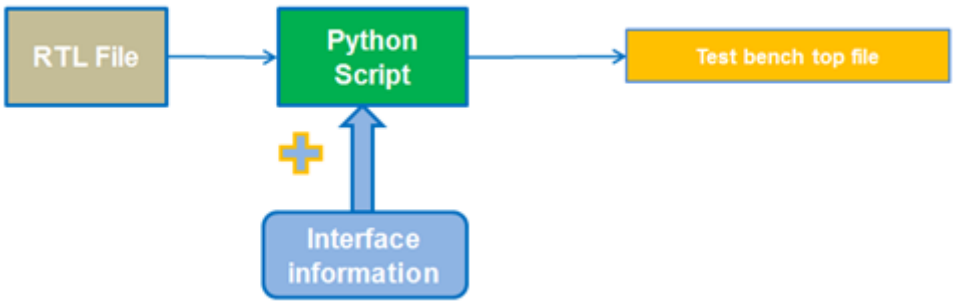

Figure 5.13: Generating Test-bench File

1. $< interface\_type >$

This option takes interface information.

2. $< address\_width >$

This option takes the information for address width used.

3. $< rtl\_file\_path >$

This option takes the input RTL file.

4. $< Register\_model >$

This option takes information Register model used.

5.5 Introduction to FEKIT

FEKIT(Front-End Kit Tool) GUI is software made on JAVA. It is an encapsulation of all the scripts required to perform basic tasks. It is developed to provide ease to user, so that he can perform all the basic tasks required at the front end from this software. Now user is not required to use terminal and run scripts by moving to different directories. Now user only require a jar file and he can populate fekit and it will automatically change directories and will run tasks. To run FEKIT GUI user should have jdk1.6 or more installed on his system, otherwise it shows error message.

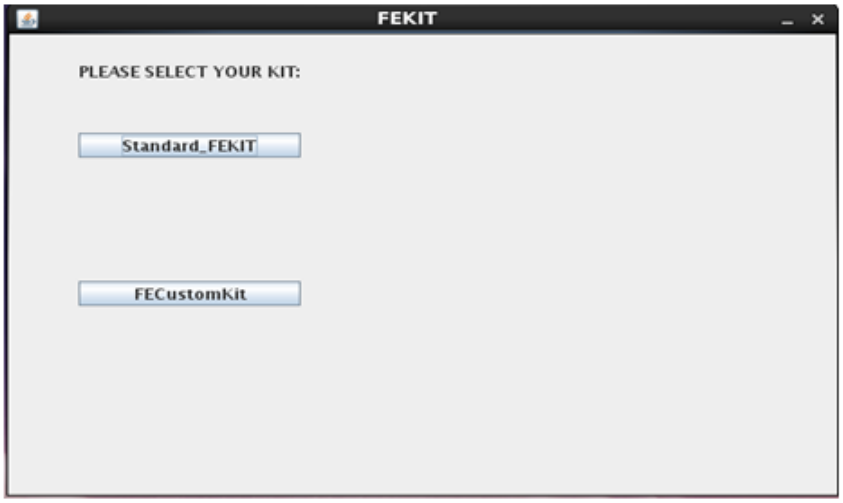

Figure 5.14: KIT-Option Window

5.5.1 How to run FEKIT GUI

FEKIT GUI provides two modes of running:

1. If user already have setup stored in his area then user only needs to go to SETUP directory of feKit_setup and run the command shown below:

java jar fekit_gui.jar

If user has placed jar file in any other directory then setup then user needs to provide path to that directory from current directory as shown below:

java jar ../path to jar file/fekit_gui.jar

2. If user dont have setup he has to run jar file as shown below:

java jar fekit_gui.jar

Then user will populate and FEKIT gui will automatically move to setup directory.

First window which appears when user launches fekit gui from setup directory is setup window as shown in fig 5.14

Next window which appears to the user is to choose standard or specific path for using FEKIT (these option is for both FeKit and FeCustomKit) this is shown in figure 6.15

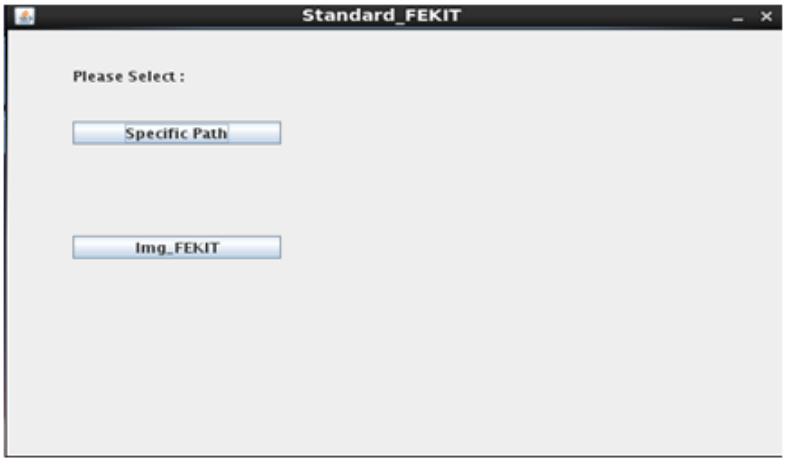

Figure 5.15: Setup Window

This window appears when user choose Img_FeKit option. This is shown in figure 6.16

Window of figure 6.17 appears when user chooses any technology option.

Window of figure 6.18 appears when user chooses option of specific Path instead of IMG-fekit. this would lead to show all technology file options with some default path and it lets the user to give the specific path for those files.

Fekit Gui will reads Project_Variables.tcl file and according to that it will show design unit, design paths, clocks, resets. User can manually add or remove design paths, clocks, resets. As user will click Submit button this will update Project_Variables.tcl and constraints.tcl files and close this window.

In window of fig 6.19 user can change LSF command according to which commands will be fired on LSF, by default LSF command will be long as set in setup.csh. As user will click Link_Design link_design script will be run and a label will appear which will show current status and kill button to kill the task and user can see log file by clicking on View Log button. The running status will be shown in text area. User can run only one task at a time among Link Design, Recital and Spyglass Compile.

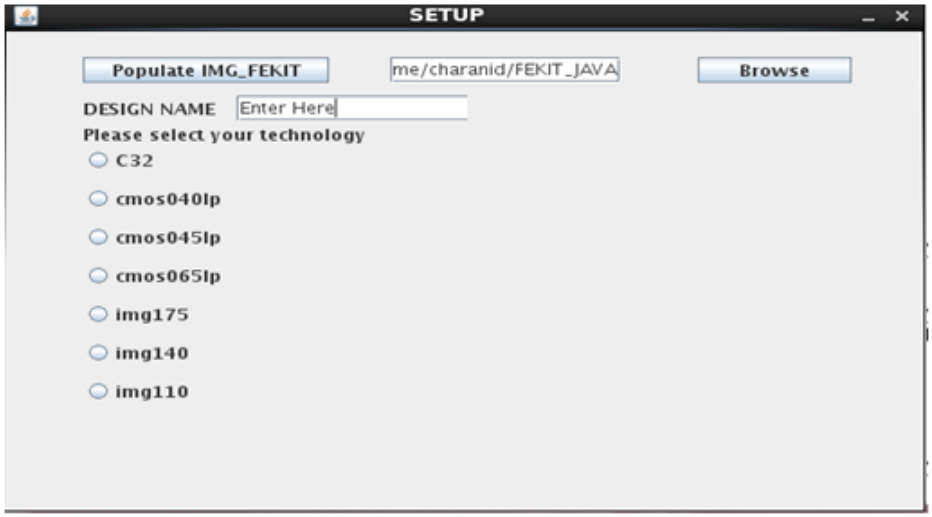

Figure 5.16: Technology option for Img-Fekit

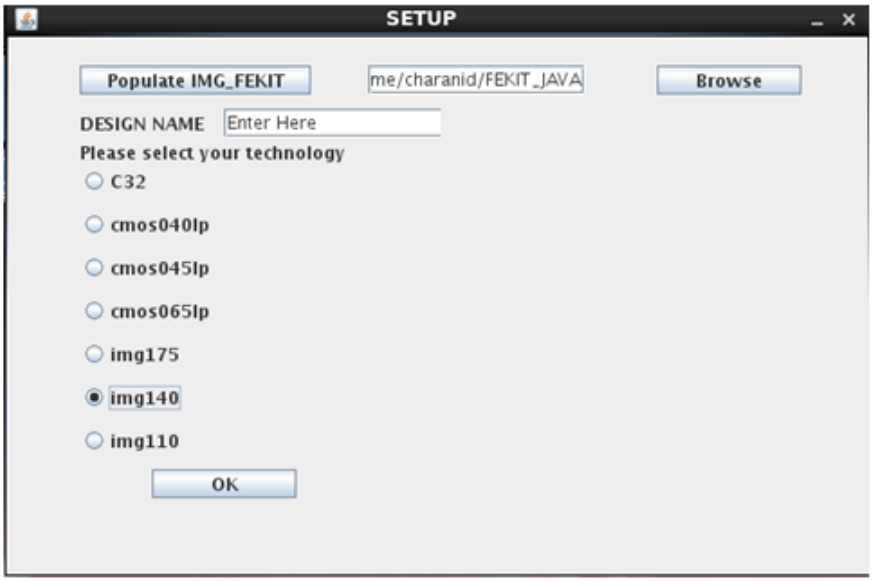

Figure 5.17: Checking of Technology option for Img-Fekit

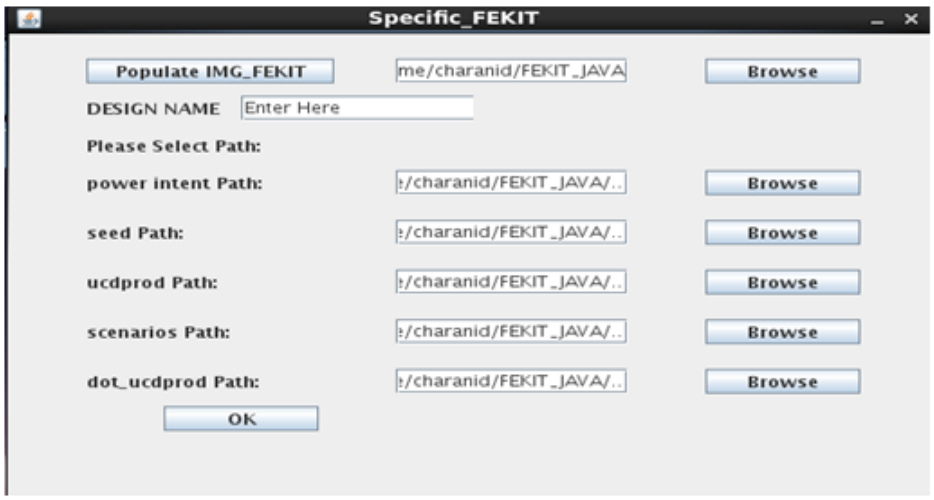

Figure 5.18: Window for Specific Path option

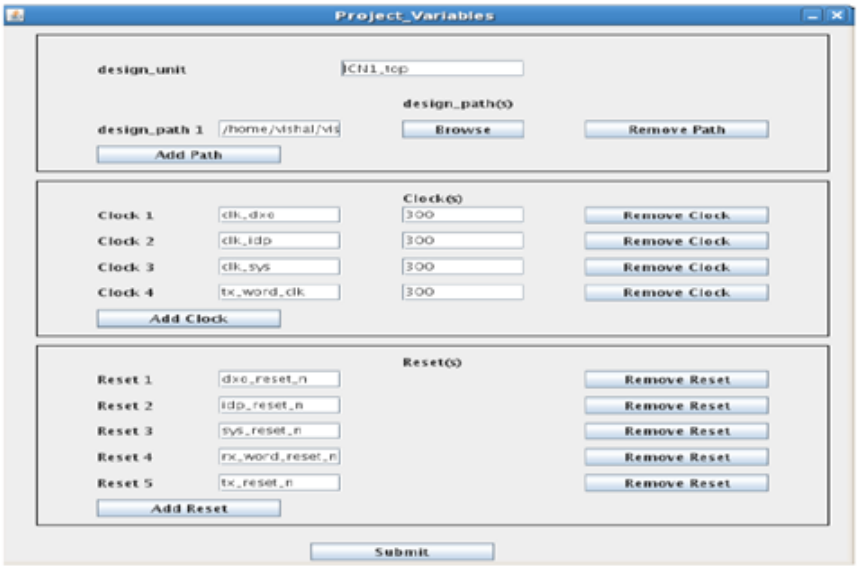

Figure 5.19: Window for Reading Design

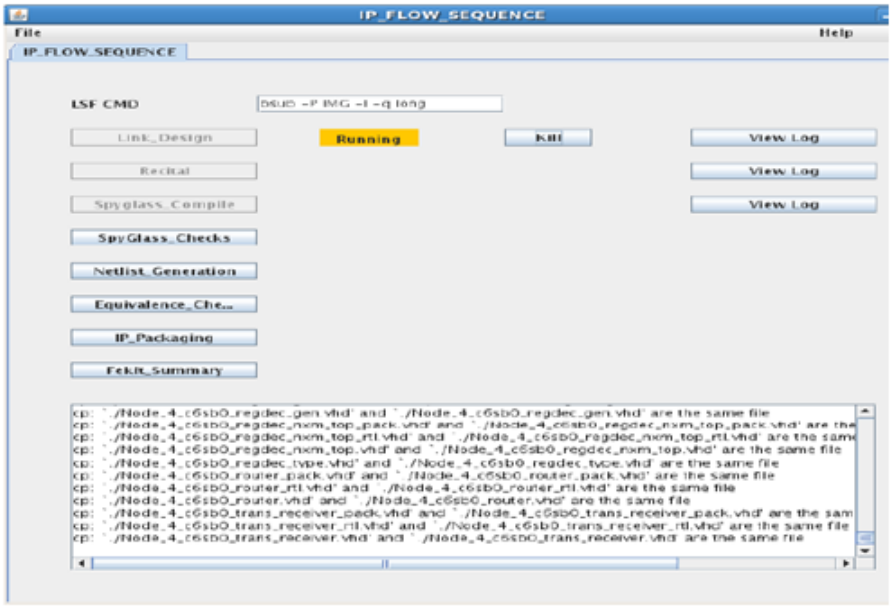

Figure 5.20: IP FLOW Window

Chapter 6

Running UVM Simulation

This chapter provides the information to run the simulation and describes the integration of Enterprise Manager (using Verification Cockpit) for running regressions and coverage analysis. And the verification cockpit flow is described in detail.

6.1 Running Simulation

To understand how UVM simulations works within the System-Verilog testbench environment, it is useful to have a big-picture view of the entire simulation flow. The design and testbench are first compiled, then the design and testbench are elaborated. Elaboration happen before the start of simulation at time-0.
At time-0, the procedural blocks (initial and always blocks) in the top-level module and in the rest of the design start running.

The first phase of an UVM testbench is the build phase. During this phase the uvm_component classes that make up the testbench hierarchy are constructed into objects. The construction process works top-down with each level of the hierarchy being constructed before the next level is configured and constructed. This approach to construction is referred to as deferred construction.

The UVM testbench is is activated when the run_test() method is called in an initial block in the top level test module. This method is an UVM static method, and it takes a string argument that defines the test to be run and constructs it via the factory. Then the UVM infrastructure starts the build phase by calling the test classes build method During the execution of the tests build phase, the testbench component configuration objects are prepared and assignments to the testbench module interfaces are made to the virtual interface handles in the configuration objects. The next step is for the configuration objects to be put into the test's configuration table. Finally the

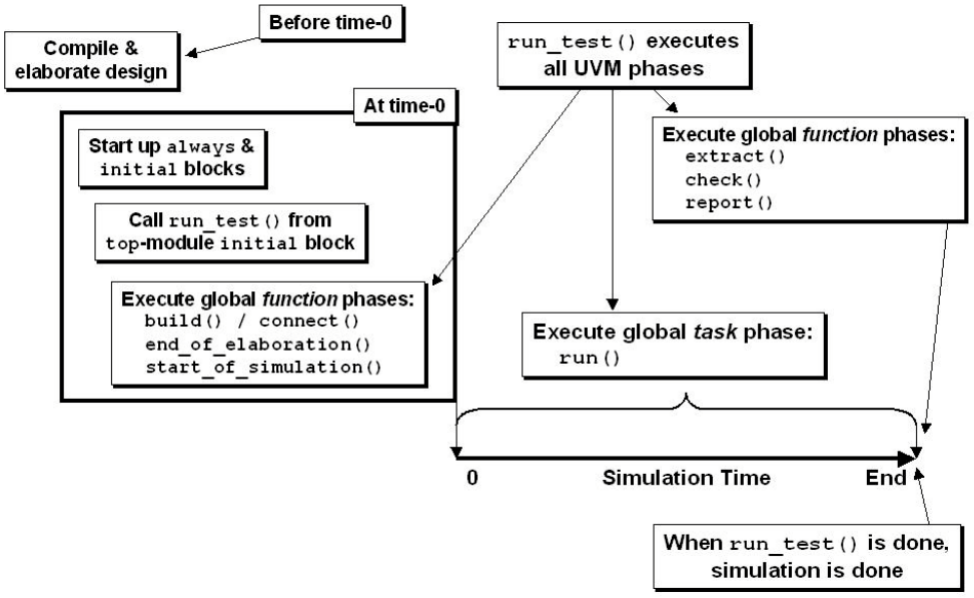

Figure 6.1: Running Simulation

next level of hierarchy is built. At the next level of hierarchy the configuration object prepared by the test is "got" and further configuration may take place, before the configuration object is used to guide the configuration and conditional construction of the next level of hierarchy. This conditional construction affects the topology or hierarchical structure of the testbench. The build phase works top-down and so the process is repeated for the each successive level of the testbench hiearchy until the bottom of the hierarchical tree is reached. After the build phase has completed, the connect phase is used to ensure that all intra-component connections are made. The connect phase works from the bottom to the top of the testbench hierarchy. Following the connect phase, the rest of the UVM phases run to completion before control is passed back to the testbench module.

The build process for an UVM testbench starts from the test class and works top-down. The test class build method is the first to be called during the build phase and what the test method sets up determines what gets built in an UVM testbench. The function of the tests build method is to:

- Set up any factory overrides so that configuration objects or component objects are created as derived types
- Create and configure the configuration objects required by the various sub-components
- Assign any virtual interface handles put into configuration space by the testbench module

- Build up a nested env configuration object which is then set into configuration space
- Build the next level down in the testbench hierarchy, usually the top-level env.

For a given design verification environment most of the work done in the build method will be the same for all the tests, so it is recommended that a test base class is created which can be easily extended for each of the test cases.

In the top-level module is an initial block that calls the run_test() task from uvm_top. When run_test() is called at time-0, the UVM pre-run() global function phases (build(), connect(), end_of_elaboration(), start_of_simulation()) all execute and complete. After the pre-run() global function phases complete (still at time-0), the global run() phase starts. The run() phase is a task-based phase that executes the entire simulation, consuming all of the simulation time. When the run() phase stops, the UVM post-run() global function phases (extract(), check(), report()) all run in the last time slot before simulation ends. Graceful termination of the run() phase often requires the use of UVM built-in termination commands, such as global_stop_request().

The uvm_test is a collection of one or more sequences that are started on a uvm_sequencer and hence what we typically call a test can really be thought of as a single test executing a single sequence, or a group of sequences executed as separate tests within the top-level test.

UVM provides run_test() command so as to start all of the UVM phases. The run_test() command is passed with the a valid test name. There are two ways to pass a valid test name to the run_test() command:

a. Coded into the top module

b. Passed to the UVM test bench through the command line switch +UVM_TESTNAME.

1. **Coded into the top module**

The test name can be written into the argument of the run test so as to have that test run. The typical example of coding the test name into the top_module file is shown below: The inline coded method is not typically recommended. This is so because this will lead to run the only test which is been mentioned. And hence will result in modifying the top module every time the user wants to run a new test. And hence need to compile the entire code repeatedly i.e. every time the new test is run. This scenario leads to increase the time of verifying the IP as this would take long to compile code and hence ultimately result in increase in entire time to verify the IP.

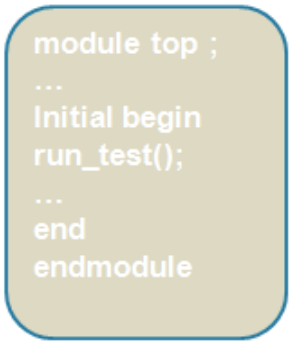

Figure 6.2: Run Code

2. **Passed to the UVM test bench through the command line switch +UVM_TESTNAME.**

UVM provides the+UVM_TESTNAME command line switch to perform the test. Using this command line switch the user work is now eased as compared to earlier use of inline codding method.

With this method the user need to compile the file for once and then run the simulation for any no of test cases as required. Since this method require to compile the design for once is reduces significant of time of verification.

Due to ease of use and reduce in time of verification, this method of passing testname through command line switch is recommended.

Below is shown an example command line switch :
At command line:

For Questa simulator:
vsim -c -do "run -all" top +UVM_TESTNAME=test1

For NCsim simulator:
irun c uvm -top testbench_top +UVM_TESTNAME=test1

```
class base_test extends uvm_test;
   `uvm_component_utils(base_test)

   dut_env            env;
   dut_env_cfg        env_cfg;

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      env_cfg = dut_env_cfg::type_id::create("env_cfg");

      uvm_config_db#(dut_env_cfg)::set(this, "*", "dut_env_cfg",
                                                   env_cfg);

      env = dut_env::type_id::create("env", this);
   endfunction

endclass: base_test
```

Figure 6.3: Base Test Class

6.2 UVM testbench Build And Connection Process

The recommended method in UVM for creating testbench components or transaction objects is to use the built-in method ::type_id::create command.Using the ::type_id::create command makes a call to the factory to extract the requested component or transaction type and then uses the new() constructor that is included in the class type to build a copy of the class-type object, all of which is done at run time. Whatever class type is stored in the factory look-up table at the requested type_id location, is extracted and created. The factory makes it possible to allow a compatible type to be stored at the desired location and therefore a compatible substitute can be automatically requested when the ::type_id::create command is executed.

6.2.1 Factory Overrides

The UVM factory allows an UVM class to be substituted with another derived class at the point of construction. This facility can be useful for changing or updating component behaviour or for extending a configuration object. The factory override must be specified before the target object is constructed, so it is convenient to do it at the start of the build process. The factory permits a top-level test to make a substitution for one of the component or transaction types in the factory at run-time, before building the entire testbench environment using factory overrides.

```
virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    seq = random_sequence::type_id::create("seq");
    seq.start(env.agent.sequencer);
    phase.drop_objection(this);
endtask
```

6.3 Managing the End of Test

A UVM testbench, if is using the standard phasing, has a number of zero time phases to build and connect the testbench, then a number of time consuming phases, and finally a number of zero time cleanup phases. End of test occurs when all of the time consuming phases have ended. Each phase ends when there are no longer any pending objections to that phase. So end-of-test in the UVM is controlled by managing phase objections

The UVM objection mechanism is designed to coordinate activities of test termination. For example, there is a need to run entire read and write operations before it can be considered complete. In this example, the various UVM verification components (UVCs) can raise objections assuring that they are able to complete before it declares that it is done. When those other UVCs complete their work, they clear their objection flags enabling the simulation to continue.

In short Component raises an objection when it starts a transaction with the DUT.And Component drops that objection when the transaction is completed Or Component expecting a response from the DUT will keep an objection raised until the response is received.

6.3.1 Objection Control

The uvm_objection class has three APIs Methods for raising and dropping objections and for setting the drain time.

- **raise_objection** (uvm_object obj = null, string description = "" , int count = 1).
  Raises the number of objections for the source object by count, which defaults to 1. The raise of the objection is propagated up the hierarchy.

- **drop_objection** (uvm_object obj = null, string description = "" , int count = 1).
  Drops the number of objections for source object by count, which defaults to 1. The drop of the objection is propagated up the hierarchy. If the objection count drops to 0 at any component, an optional drain_time and that component's all_dropped() callback is executed first. If the objection count is still 0 after this, propagation proceeds to the next level up the hierarchy.

- **set_drain_time** (uvm_object obj = null, time drain).
  Sets the drain time on the given object.

6.4 Verification Cockpit Flow

It is important to concentrate on debugging design issues than spending time on debugging tool integration and script maintenance. Verification Cockpit flow is Linux compatible product and a infrastructure tool that bridges other tools such as NC-sim,eManager(These are Cadence Tools) etc. VC cockpit flow (ST internal framework) thus helps user to eliminate wasting time on tool integration The main backbone of verification cockpit is 2 scripts

a. Test case file (testcase.csv) which is generated automatically.

b. Setup script (vc_setup.csh) which sets the entire variable we use in the environment.

The user need to create the setup script file that lists all the variable pointing to the certain directories and script to be used in environment. User need to create the test-case file using generic command vc_generate_test this will to generate a file with all the testcase names in it. This file is then used at the time of simulation to choose the testcase to be run.

The verification cockpit flow is described below:

a. **Build and Compile** :

   After setting up the entire verification environment, the design is build. To do so vc_build command is used. This command is invoked in the directory pointed by variable $VE_BUILD_DIR and will execute a script pointed by variable $VE_BUILD_SCRIPT. These variables are already been set in the setup file.

   After the successful build, next step is to compile the design. To compile the IP environment compilation command are used. This command basically runs the script. The script is generally list of steps that the user uses for compilation.

After compiling, if any error occurs, then it can be viewed in the log file which will be generated in directory pointed by variable $VE_BUILD_DIR.

b. **Running Standalone Test case** :

After compiling the Environment, simulation is carried out for the same environment. Here first built-in register test-cases are run (so as to verify the register/memory configuration) and then the streaming tests. The command to run the test-cases are launched in the directory pointed by variable $VE_RUN_DIR and will execute a script pointed by variable $VE_TEST_RUNNER. This variables are already been set in the setup file. A tcl script is used for pointing the input values. The results of simulation can be observed and the waveforms can be checked for verifying the functionality of the IP.

Since UVM methodology is interoperable i.e. it is vendor independent, simulation can be run with any simulator available. Generally NCsim simulator is used for simulating the design but modelsim simulator and vcs simulator can be used as well.

The script used to run simulation has various options for running the test.An example of command line to run simulation is as follow:

vc_run_test -t test1 -b
This leads to run simulation at command line.

vc_run_test -t test1 -probe all
This leads to run simulation in GUI mode as shown below.

After running the Stand-alone test, check for coverage is carried out. There is a option for NCSim to have direct coverage through IMC (Incisiv Metric Center). An example for coverage of registers(through register-derived-class) for a given test is shown below with IMC.

c. **Running Regression** :

The Cadence Tool, Emanager is used for running regressions. When regression is run a .vsif file is generated from the test-case file and this vsif file will be input to the Emanager. The main aim over running regression is to have the coverage information such as function coverage, code coverage etc. of the IP to be verified.

d. **Customize Regression** :

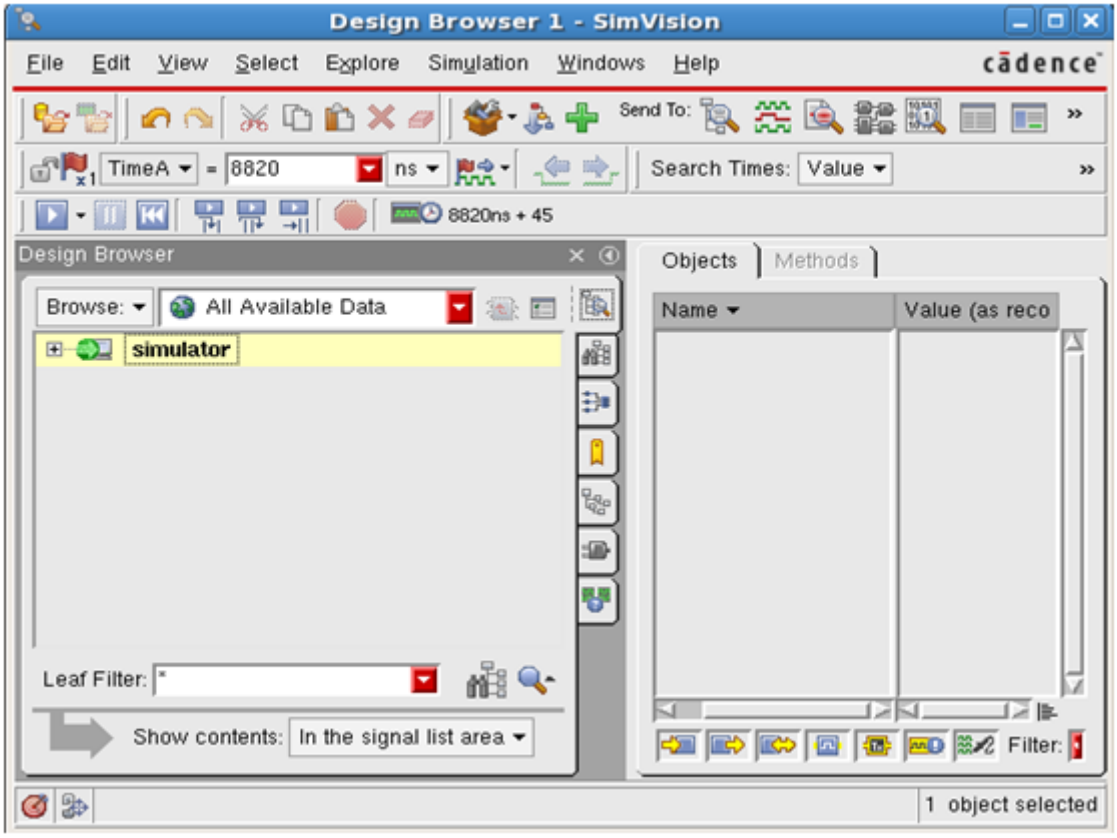

Figure 6.4: NCSIM simulator

To run customized regression vc_server is used -

1. This command which will invoke a terminal and provide us with a link to open in your web browser (Internet Explorer or Google Crome etc...).

2. This will open a GUI based interface of the test-case fie where selection of the number of test-case to be run is done also dumping of a VSIF file for running later or checking of status of runs on GUI window can be done.

3. Standalone test-case can be run from the GUI either in the same run directory or new directory.

4. Probes can be enabled to run standalone test-case.

5. Filtering can be applied in case we have huge number of testcase and can save the filter to apply it whenever needed.

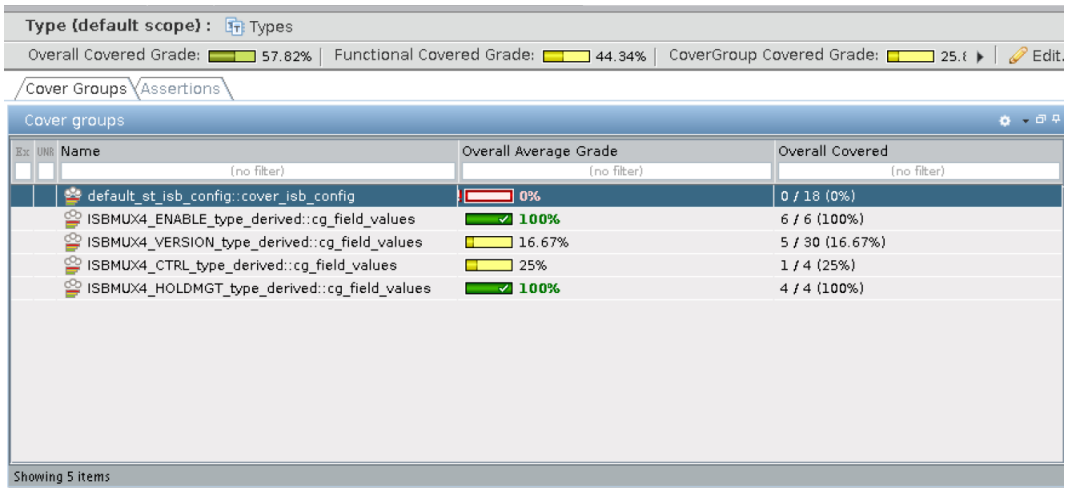

Figure 6.5: Coverage in IMC

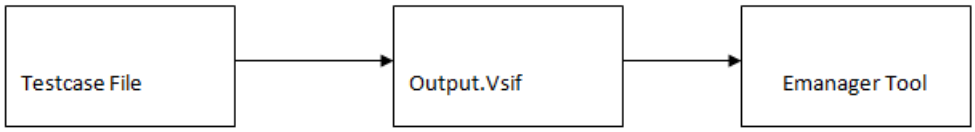

Figure 6.6: Regression

Regression can be run on a SQL server for which server setting has to be done on the setup.csh script and then the results can be extracted by pinging the SQL server and have a report as required.

6.5 Make-file

Make utility automates the entire verification cockpit flow. Once the suitable make file exist, each time you change the source file, you now just need to give the command make. This leads to reduce further time to run test. Make command will now need argument MODE so as to determine the running of test in GUI or in Batch mode. Along with the name of test to run.
Eg:
>> **make** <test_case_name> MODE=<GUI/BATCH>

This command will compile the source code and run the given test in GUI/BATCH mode.

Chapter 7

Conclusion and Future Scope

7.1 Conclusion

The System-Verilog Universal Verification Methodology (SV-UVM) based reusable verification environment is efficiently used for verifying the imaging IP/SoC. The SV-UVM methodology helped in overcoming many drawbacks of earlier methodologies. As compared to earlier methodologies, where verification flows were disjoint at IP, SoC and Validation level, SV-UVM has emerged to be a methodology helping in saving verification cost and effort. Also Migration from uvm_rgm to uvm_reg register Model usage for register verification has lead to many advantages of having more in-built test-case for verification and automatic check while having use of basic read -write API's.

IP-XACT based tools are used for the automatic generation of IP/SoC dependent system Verilog files. IPXACT flow is so designed to fit the requirement of the designer expecting to reduce the time-to-market. Now with the help of IPXACT flow 90% of the IP/SoC dependent System Verilog files are automatically generated. IPXACT flow is independent of the design language and design tool and hence is a very efficient tool.

Use of FEKIT tool save the time of the user as all the flow is automated. i.e without having change directory for running different commands and also easing the pain of remembering all the commands to run. This tool will be very useful for the users because this will make them to run all the options automatic and no need do much manual changes.

7.2 Future work

In Future, more Improvement in existing verification environment in terms of flexibility, coverage and reusability from IP level to SOC level can be done. And addition of more feature to IPXACT script (spirit2uvm) for automatic generation of all the files (i.e for Sub-System level and SoC level) from xml specification file.

References

[1] Abhishek Jain, Giuseppe Bonanno, Dr. Hima Guptaand Ajay Goyal (2012) "Generic System Verilog Universal Verification Methodology based Reusable Verification Environment for Efficient Verification of Image Signal Processing IPs/SOCs

[2] Accellera Organization, Inc. Universal Verification Methodology (UVM) May 2012

[3] http://www.doulos.com/knowhow/sysverilog/uvm/

[4] http://testbench.in/

[5] ClueLogic, UVM Tutorial for Candy Lovers - 16. Register Access Methods, www.cluelogic.com

[6] ST Microelectronics internal documents regarding UVM Methodology