DEVELOPMENT OF OPEN VERIFICATION IP FOR I2C CONTROLLER

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF BACHELOR OF TECHNOLOGY IN
ELECTRONICS AND COMMUNICATION ENGINEERING

SUBMITTED BY:

SANTOSH KUMAR PATRO (107EI033)

JYOTI PRAKASH SAHOO (107EI020)

Under the guidance of
Dr. D.P. ACHARYA
ASSOCIATE PROFESSOR

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY
ROURKELA - 769008

National Institute of Technology, Rourkela

CERTIFICATE

This is to certify that the thesis entitled **"DEVELOPMENT OF OPEN VERIFICATION IP FOR I2C CONTROLLER"** submitted by Jyoti Prakash Sahoo (107EI020, Dept of ECE), Santosh Kumar Patro (107EI033, Dept of ECE)) in partial fulfilment of the requirements for the award of Bachelor of Technology degree in Electronics and Instrumentation Engineering at the National Institute of Technology, Rourkela is an authentic work carried out by them under my supervision and guidance.

To the best of my knowledge, the contents described and analysed in the thesis has not been submitted to any other University/Institute for the award of any Degree.

Date:

Dr. D.P. Acharya
Associate Professor
Dept. of Electronics and Communication Engg
National Institute Of Technology
Rourkela – 769008

ACKNOWLEDGEMENT

We take thank all the individuals without whose support and guidance we could not have completed our project in this stipulated period of time. First we would like to express our deepest gratitude to our Project Supervisor, Dr. D. P. Acharya(Associate Professor), Department of Electronics and Communication Engineering, for his invaluable support, guidance, references, motivation and encouragement throughout the period this work was carried out. His readiness for consultation at all times, his educative comments and inputs, his concern and assistance even with practical things have been extremely helpful.

We would also like to thank all professors and lecturers and the lab faculty members and other industrial experts for their generous help in various ways for the completion of this thesis. We also extend our thanks to our fellow students and friends who were there all the time for all kinds of help and support .We would like to thank everyone for their friendly co-operation.

SANTOSH KUMAR PATRO	JYOTI PRAKASH SAHOO
107EI033	107EI020
DEPARTMENT OF ECE	DEPARTMENT OF ECE
NIT,ROURKELA	NIT, ROURKELA

ABSTRACT

Before any IC is fabricated it is desired to check whether the required functionalities are preserved or not. Otherwise this may lead to a huge loss to the company in case of any failure in during the design/coding stage. Verification engineers have to make sure that before fabrication all the properties of the IC can be successfully implicated. So functional verification provides a lot of benefits to the IC designers. Today, testing and verification are alternatively used for the same thing. Testing of a large design using FPGA consumes longer compilation time in case of debugging and committing small mistakes. Simulation based testing is faster and also provides capability to check all the signals buried under the design. But due to the increasing complexity in design and the concurrency behavior of the design it has become very difficult to verify the functionality using traditional testbenches. So new languages called Hardware Verification Languages (HVL) are introduced. System Verilog is an IEEE standard Verification language. The library and package oriented feature provide an efficient way of writing testbenches. The Open Verification Methodology **(OVM)** Class Library provides the building blocks needed to quickly develop reusable and well-constructed verification components and test environments using SystemVerilog. In this paper we have developed testing environment using system Verilog implementation of OVM for I2C controller core. Our work introduces an automated stimulus generating testing environment for the design and checks the functionality of the I2C bus controller.

Keywords: System Verilog, *OOP, OVM, , I2C bus , SDA, SCL, ovm_test, ovm_env*.

CONETNTS

CHAPTER 1

INTRODUCTION

1.1 I2C Bus

I2C bus was developed by PHILIPS SEMICONDUCTOR in 1980s for use in television. It is a simple bus having only two wires required for data transmission. I2C is a simple, two-wire interconnect comprising a serial data channel plus a clock. The protocol supports device addressing, which can simplify board design and also allows extra devices to be added easily - up to the maximum 112 devices allowable with the constraints of the standard 7-bit address space. The I2C bus is cost-effective and easy to use, so it's widely adopted as a simple interconnect. It's good for use in case of slow transmission rate over a small distance between different ICs and it finds its application in consumer electronic products.

1.2 Motivation

OVM is new concept in the verification field. We are eager to explore this methodology for developing Verification environment for different IPs. The object oriented and coverage driven verification feature of ovm are efficient for functional verification. The System Verilog clocking block implementation addresses the synchronisation issue regarding sampling and driving the signals. So that functional verification can be effectively carried out .

1.3 Verification

Verification is process used for ensuring that the properties of the design unit are preserved prior to the mapping into the silicon. This process is implemented in parallel to the design process. So at each stage of the design we can check out the bugs.

Verification is same as testing but fundamentally different from designing. So design and verification are two different areas of experts in the VLSI market. System Verilog and Open Vera are the widely used HVLs. Generally design and verification engineers work together for debugging as well for problem solving. It's better to use same language for design, testbench and assertion because testbench has the direct access to every part of the environment without requiring any external. Here in this paper we are concerned about the protocol verification of the I2C bus controller.

1.4 The Need Of Verification

Verification is different from design but it requires complete knowledge of the design. A primary purpose of functional verification is about finding failures identifying bugs and correcting before they are mapped into the IC. As electronics market is changing swiftly and its growth being enormous it induces designers to go for complex IC design and packing them into small spaces. So systems on chip (SOC) are developed. 70 % of design effort goes to verification. Checking of complex design, preserving intellectual property (IP), testing of SOC makes verification a difficult task. So in industry the number of verification engineers is much more than RTL designers.

1.5 Traditional Testbenches And OVM Verification

Testbench the environment in which the design will reside. It checks whether the RTL Implementation meets the design specification or not. This Environment creates valid and expected as well as invalid and unexpected conditions to test the design.

Linear Testbench is the simplest, easiest and fastest way of writing testbenches .But it' the slowest way to execute stimulus. These are written in the VHDL or Verilog. Here simple sequence of test vectors is mentioned. Small models or simple state machine can be verified with this approach. It's not the best approach for complex designs also it's very hectic to consider all the possible scenarios.

Testbench supplies input to the design and watches the output, but verification determines the pattern of input that should be given to the design and checks whether the design is properly working or not [21].

Problems Faced In General Testbenches Are:

- The tests are generated by human and hence all possible scenarios are not covered. Even if in case of Random tests 100% coverage of all possible values are not ensured.
- Concurrency can't be verified
- Code coverage couldn't catch bugs because of concurrency nature of the design

Advantage of verification environments:

- Testbench automation by means of SystemVerilog l enables Constrained Random Verification
- more state space can be covered than directed tests
- Assertion/Property Based Verification ensures the expected behavior of the design unit
- Formal Verification validates all features of the hardware
- Total Coverage Model indicates the end of the verification

1.6 Verification Plan

Verification plan is based on System Verilog language constructs and Coverage driven verification methodology.

Feature Extraction:

All the desired features of the design that are to be verified are listed to ensure the proper functioning of the design.

Stimulus Generation Plan**:**
Random stimulus is automatically generated by the test environment depending upon the design specification and requirement. All possible types of input conditions are generated to avoid bugs in code/functional coverage.

Coverage Plan:

The coverage plan describes the functional coverage of the design features. The functional coverage plan is used for the implementation coverage points in verification environment. Generally, it is better if the coverage block is broken based on the design logical blocks. It will list various Coverage groups and assertion.

Implementing The Verification Environment:

The verification environment should be ready prior to finishing of the RTL design. In Most of the cases the IC design consists of multiple modules. And they are designed one after other according to the design hierarchy. So in all stages the desired features can be verified. This helps to produce a bug free design.

Response Checking:

In traditional testbenches the stimulus as well as the functionality is known. So systems performance can be checked by using hard-coded conditional statements in the testbench. But in randomly generated test cases although the systems functionality is known but the stimulus is not known .So the expected response has to be computed and compared with the actual response of the system.

Assertion:

A System Verilog assertion specifies and validates the behavior of a design. In addition, they are used to provide functional coverage and generate input stimulus for validation. The evaluation of the assertions is guaranteed to be equivalent between formal verification, which is cycle-based and simulation, which is event-based . SystemVerilog allows assertions to communicate information to the test bench and allows the test bench to react to the status of assertions [10].

Accuracy:

The verification environment should accurately validate the system functionality. So the desired output has to be computed cycle-by-cycle basis and compare to the actual output. This is the toughest job for a verification engineer. From the specs the behavior of the system has to be inferred.

Score-boarding:

Score boarding is used to predict the output of the system dynamically and compare with the actual output. It’s good for comparing end to end results and integrity of output data .But it’s not good for finding the cause of error.

1.7 Methods Of Verification

Mentor Graphics and cadence are providing some of the leading advanced methodologies. Three basic methodologies used are:
Assertion based Verification:

- Assertions actively monitor a design (or testbench) to ensure the correct functionality and detect design errors. Hence they greatly increase the observability and decrease debug time.

- ABV methodology with assertion based languages like SVA (a module of System Verilog) is supported by a wide number of tools and they can be implemented with Verilog, SystemVerilog, and VHDL designs.
- The ABV flow leverages the assertion libraries and user defined assertions. Those are used in simulation, functional verification, and emulation or FPGA supported prototyping, and useful for coverage collection in a coverage database file like Questa's Unified Coverage Data Base (UCDB).

Universal Verification Methodology /Open Verification Methodology:

- The OVM provides class library (including source code and documentation) that is used to construct such test environment and generate stimulus. The OVM class library is written using standard IEEE 1800 SystemVerilog.
- The documentation provides the details about constructing the testbench and reusable VIP components. SystemVerilog language is not by itself sufficient. Without the building blocks and methodology to describe how to use them, interoperability could not happen.

Processor driven Verification:

- A processor-driven testbench generates desired stimulus for the design and monitor the DUT signals using fully functional processor model. The test instructions provide stimulus to the design when they are executed and monitor the DUT signals, but by some other set of instructions. The processor driven tests provide greatest re-usability in the project life.
- However, drawback of this model is the tests do not cover the entire interaction between the external logic and the processor. Since the end product is processor driven, the approach of executing instructions on a fully-functional processor model and driving stimulus into the design is better than traditional bus-functional models.

Chapter 2

2.1Introduction To OVM

For large chips, the days of hand-writing hundreds or thousands of tests and running them one at a time are long over. Modern verification projects make use of constrained-random stimulus generation, in which the simulator does most of the work to generate the tests, and rely on functional coverage metrics to assess progress. Verification is complete only when all coverage points for all chip functions have been exercised. The OVM supports this approach intrinsically as the best way to verify complex designs.We have used OVM version 2.0 here.

OVM allows us to achieve coverage-driven verification (CDV). CDV is combination of automatic test generation, self-checking testbenches, and coverage metrics to significantly reduce the time spent in verifying the design. The purpose of CDV is to:

■ reduce effort and time in creating tests.
■ goal of the verification can be easily achieved.
■ Receive early error notifications and run-time checking and error analysis for debugging.

With CDV, we start by setting verification goals. Then we create a testbench to generate legal stimuli and send to the DUT. Cover groups are added to the monitor to measure progress and detect non-exercised functionality. Checkers can be added to identify undesired character of the DUT. Simulations are launched after the implementation of both the coverage model and testbench. Verification can be achieved in this way.

Using CDV, verification of the design can be thoroughly done by changing testbench parameters or by changing the randomization seed. Test constraints can be added. CDV support both directed as well constrained-random testing. However, the preferred approach is to let constrained-random as it’s very difficult to include all possible scenarios in case of directed tests.

2.2 OVM Testbenches And environment

An OVM testbench is composed of reusable verification environments called Open verification components (OVCs). An OVC is a ready-to-use, encapsulated and configurable verification environment for a small design, design sub-module, or a full system. Each OVC follows a standard architecture and consists of a set of elements for collecting coverage information and checking for a specific protocol or design also for stimulating. The OVCs are connected to the design under test (DUT) for verifying the implementation protocol or design architecture. Data passes between different OVCs as transaction as in Transaction Level Modelling (TLM).

The interface OVCs are instantiated and configured for a desired operational mode. The verification environment contains a multiple channel sequence mechanism the virtual sequencer which synchronizes the timing and the data between the different interfaces and allows fine control over the test environment for a particular test.

The OVM Class Library System Verilog provides the building blocks which are needed to develop the transaction based and well-constructed and reusable verification test environments and components in SystemVerilog [2].

OVM library contains:

- Component classes for building testbench components like generator/driver/monitor etc.
- Reporting classes for logging,
- Factory for object substitution.
- Synchronization classes for handling concurrent process.
- Classes for printing the report, comparing results, packing, and unpacking the data fields of ovm_object based classes.
- TLM Classes for transaction level passing of data through interface.
- Sequencer and Sequence classes for generating realistic stimulus.
- Macros which are used for shorthand representation of complex implementation.

2.3 The OVM Library

The OVM library and methodology provides all necessary features the technologies for constructing reusable, constrained-random and coverage-driven test-benches. It provides the TLM-based modelling for building modular and reusable verification components which communicate through transaction-level interfaces.

The OVM class library allows us creating sequential constrained-random stimulus which helps to collect and analyse the functional coverage and include assertions configured as members of those test-bench environments.

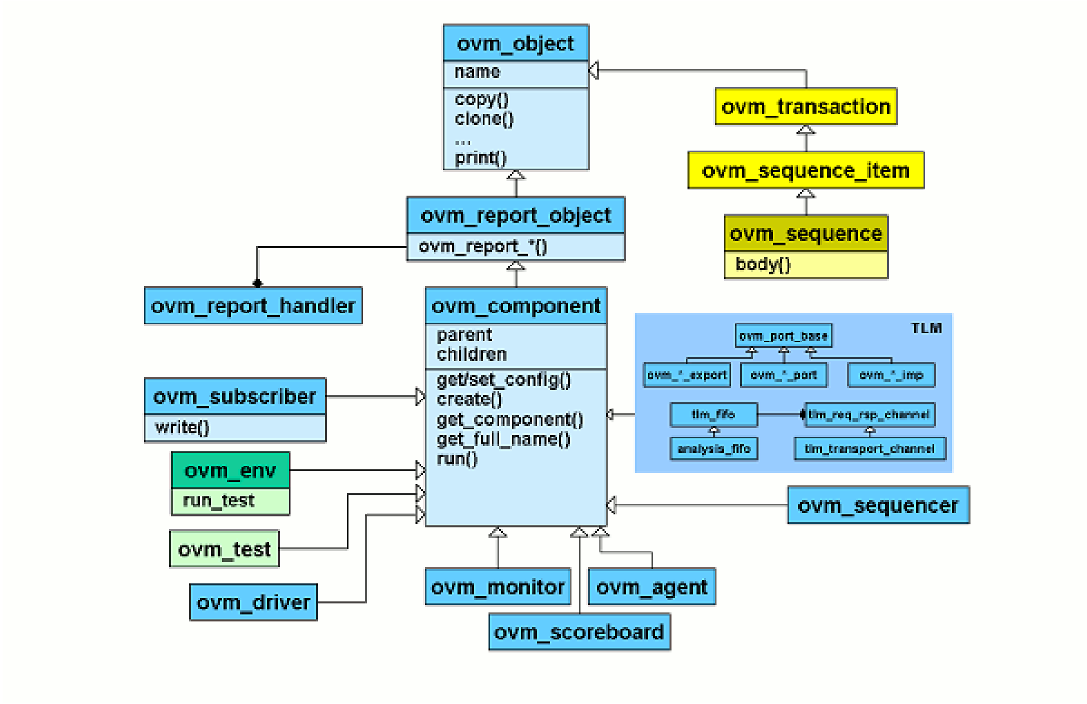


Figure 2.2 OVM class hierarchies [1]

2.4 Components Of OVM

The following are the required and important components of OVM based verification.

2.4.1 Design Under Test

This describes the design that is intended to be verified .This is generally RTL description in any of the HDL (Verilog, VHDL and System Verilog).This completely describes the functionality of the design as well the features to be verified.

2.4.2 Interface

Interface serves as the actual link between the design-under-verification and the verification environment. It is a SystemVerilog interface. The interface describes the pin-level description of the DUT. An interface is basically a bundle of nets or wires.

In Verilog, different modules are connected through module ports. For large modules or complex designs this is not productive as manually connecting hundreds of ports is not an easy task. This is prone to error .Also this demands the detailed knowledge of all the pins.

And on changing the design features it is required to change the pin connection in the interface also. But these problems can be overcome by using interface. Interface can also be used to connect the DUT to other sub-modules or components. The advantages are given below:

- It is passed as a single item.
- Port definitions are independent of modules
- Easy to maintain the signals
- Structured information flows between blocks.
- Interface can include tasks and functions as well protocol checking using assertions.

Virtual Interfaces

Virtual interfaces provide a mechanism for separating abstract models from the actual signals of the design. A virtual interface allows the same instance or the subprogram to operate on different parts of the design. It dynamically controls the set of signals associated with the subprogram, this allows passing the same data over all the components. Instead of referring to the actual signals directly, we can manipulate a set of virtual signals [6]. Their use will be discussed in further sections.

2.4.3 Transactions

Interfaces represent the input to the DUT. The fields and attributes of transactions are derived from the transaction's specification. In a test, many data items are generated and those are sent to the DUT via driver. Generally data item fields are randomized using System Verilog constraints many number of tests can be created.

2.4.4 Sequence And Sequence

A sequence is the series of transaction and sequencer is used to control the flow of transaction generation. A sequence is extended from ovm_sequence class. Ovm_sequencer does the generation of this sequence of transaction. Driver (extension of ovm_driver) takes the transactions from Sequencer and processes the packets of data or drives them to other component or to the DUT [20].

2.4.5 Driver

Driver is defined by extending ovm_driver. Driver takes the transactions from the sequencer by using seq_item_port. These transactions will be driven to DUT as per the interface signal specifications. Then it sends the transaction to scoreboard using ovm_analysis_port. Task for resetting DUT and configuring the DUT are also declared here. An instance of the driver class is created in the environment class and the sequencer is connected to it. The following figure shows the connection between the ovm_sequencer and ovm-driver and connection between ovm_driver to ovm_scoreboard.

2.4.6 Monitor

Monitor samples DUT signals but does not drive them. Monitors collect coverage information and perform protocol checking. Even though reusable drivers and sequencers drive bus traffic, they are not used for coverage and checking. Monitor does the following:

■ extracts signal information from a bus and translates the information into a transaction that can be made available to other components and to the test writer.

■ performs checking and coverage.

Checking typically consists of protocol and data checkers to verify that the DUT Output meets the protocol specification. Coverage is collected in the monitor. It is implemented by extending the ovm_monitor class and an instance is created in the environment for hooking it up with DUT signals.

2.4.7 Scoreboard

Scoreboard is implemented by extending ovm_scorboard. Scoreboard has 2 analysis imports. One is used to for getting the packets from the driver and other from the receiver (extended class of ovm_component class). Then the packets are compared and if they don't match, then error is asserted. Compare function of transaction class is used for comparision.

2.4.8 Environment

Environment class is used to implement verification environments in OVM. It is extension on ovm_env class. The testbench simulation needs some systematic flow like building the components, connection the components, starting the components etc. ovm_env base class has methods formalize the simulation steps. All the methods inside environment class are declared virtual. Virtual interface is created in the environment and all other virtual functions of environment class are extended.

2.4.9 Testcases

The ovm_test class defines the test scenario for the testbench for the DUT and is specified in the test. Testcase contains the instance of the environment class. This testcase creates an Environment object and defines the required test specific functionality. Verification environment contains the declarations of the virtual interfaces. These virtual interfaces are pointed to the physical interfaces which are declared in the top module. These virtual interfaces are made to point to physical interface in the testcase.

2.4.10 Top Module

System Verilog interface instance is created in this module. DUT instance is created and hooked up with the interface instance. Clock generator is implemented here. run_test method is called. The test name can be implicitly passed or can be passed as a command line argument during simulation. The implementation will be discussed in the further sections.

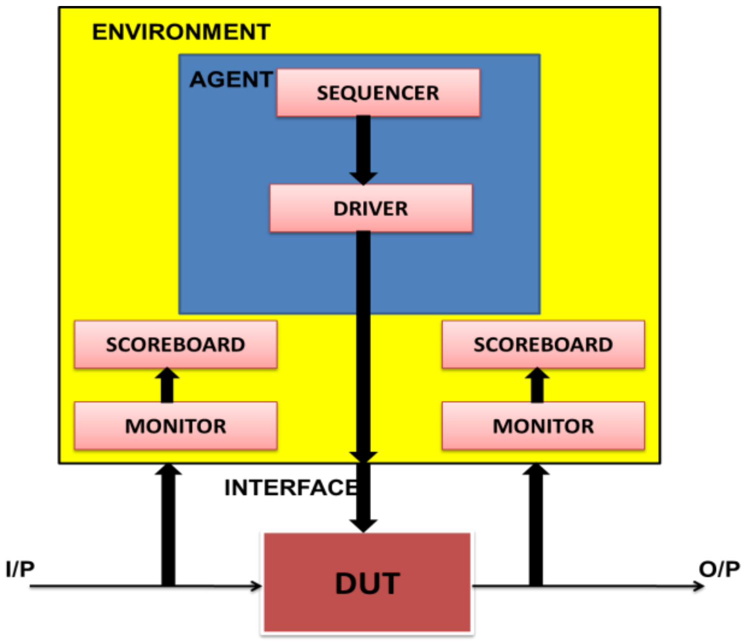


Figure 2.8: Verification Components (All of them are inside the Testclass)

2.5 Advantage Of Transaction Based Verification

In Transaction-Based Verification (TBV at each stage of the verification cycle the desired functionalities of the design are specified and, checked. The growing market demand results many designs with incomplete verification. Signal level verification creates longer debugging cycles which is not practical for today's complex system on chip (SOC) and system design as they contain thousands of signals. The design operates at the signal level (ones and zeros). System architects do not consider about signal level. They start their design considering data flows through the system and with knowledge about how and where to store.

2.6 **Advantage Of Using OVM**

The following are the some of the advantages of using OVM for verification:

Open Source:

It is distributed under the standard open-source Apache™ 2.0 license. The OVM code is supported and runs on any SystemVerilog compliant simulator [15]

Transaction-Level Modelling:

Using SystemVerilog implementation of TLM in the OVM, a component may be connected via its interface to other components those implement that interface. Thus, one component can be connected to others components implemented at many levels of abstraction using TLM at the transaction level [15].

OVM Automation:

OVM provides access to advanced automation for simplifying environment creation. Macros dramatically reduce coding and improve readability. The same features can be availed by function calls. [15].

Sequential Stimulus:

OVM provides class feature for building hierarchical sequences. Defining sequences as objects, the OVM separates the test behavior from the testbench, allowing greater modularity and reuse of stimulus scenarios [15].

Sophisticated Virtual Sequences:

Virtual sequences are used to coordinate the execution of multiple sequences on different interfaces; this allows more sophisticated tests to exercise more complex cases of the design under test (DUT) [15].

Simple Test Writer:

The test writer specifies procedurally the type of transactions to be generated, along with an optional set of constraints. The transaction is automatically randomized and sent to the driver.

CHAPTER 3

I2C BUS

3.1 Introduction

A bus connects the components of a system , e.g .- connection between cpu and main memory, memory and I/O ports or between peripheral devices . Different types of bus are available for data transfer. They are I2C, PCI, WISHBONE, AMBA, XBUS, SPI, and USB. Each bus has different protocol and bus speed. A faster bus speed allows faster data transfer. In a typical computer or SOC bus is used for address transfer and data transfer.

The I2C bus was designed by Philips in the early '80s to allow easy communication between components which reside on the same circuit board. The name I2C translates into "Inter IC". Sometimes the bus is called IIC or I²C bus [16]. I2C is used on single boards and to connect components which are linked via cable. Key characteristics that make this bus attractive to many applications are simplicity and flexibility.

- I2C bus has three speeds:

  -Slow (under 100 Kbps)

  -Fast (400 Kbps)

  -High-speed (3.4 Mbps) – I2C v.2.0 [16]

3.2 I2c bus protocol

3.2.1 I2C Bus Terminology [11],[12]

• **Transmitter** - It sends data to the bus. It can either put data on the bus of its own accord (a 'master as transmitter'), or in response to a request from data from another devices (a slave as transmitter) [11].

• **Receiver** - the device that receives data from the bus.

• **Master** - initializes a transfer, generates the clock signal, and terminates the transfer. A master can be either a transmitter or a receiver [11].

• **Slave** - the device addressed by the master. A slave can be either receiver or transmitter.

• **Multi-master** - To incorporate more masters who can co-exist on the bus at the same time without collision or data loss [11].

• **Arbitration** – The procedure to decide which device will act as master in multi master system [11].

•.**Synchronization**- Procedure to provide clock to components of bus and synchronize them.

• **SDA** – data signal line (Serial Data)

• **SCL** – clock signal line (Serial Clock) [11].

3.2.2 I^2C Communication Procedure [11]

One IC that wants to talk to another must:

1) Check whether there is any bus activity is occurring or not. If both SDA and SCL line are high then bus is free. If the bus is available master generates START condition.

2) SCL provides clock signal to all the ICs connected through the bus as reference clock signal. The data on the data wire (SDA) must be valid at the time the clock wire (SCL) switches from 'low' to 'high' voltage [11].

4) Address of each device is put on serial form on the SDA line.

5) One bit signal is put on the SDA line to know whether data is to be transmitted or received from the slave.

6) One bit represents acknowledgement bit to inform the master that slave is ready to receive or transmit data.

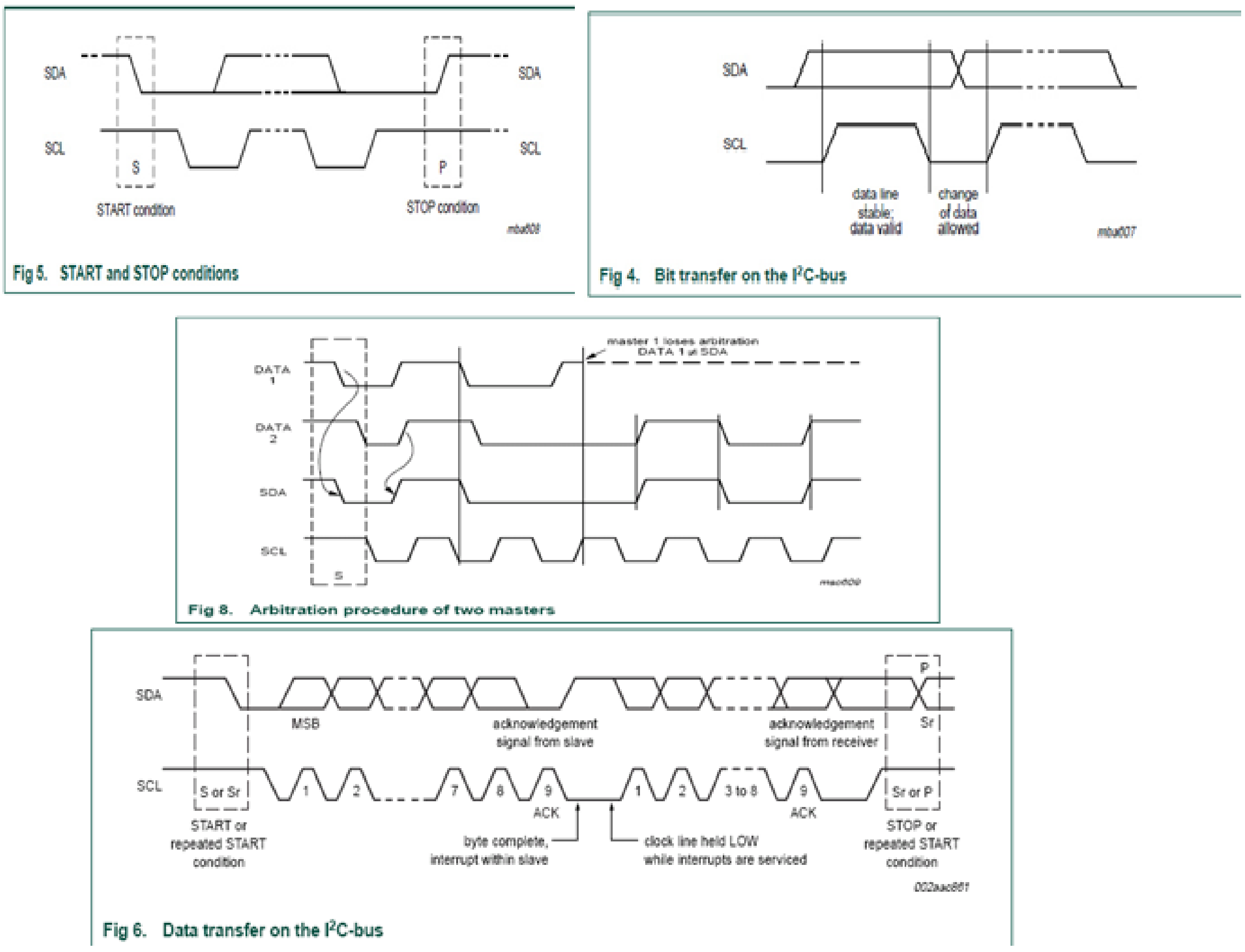
7) After the acknowledgement bit is received by the master it puts data serially on the SDA line.

8) The first IC sends or receives as many 8-bit words of data as it wants. After every 8-bit data word the sending IC expects the receiving IC to acknowledge the transfer is going OK [11].

9) When all data is received STOP condition is generated and the bus is again free.

Only two chips are involved in any one communication - the Master that initiates the signals and the Slave responds when addressed.

4.3.3 I2c protocol diagram



SDA
SCL
S
P
START condition
STOP condition
mba608
Fig 5. START and STOP conditions
SDA
SCL
data line stable; data valid
change of data allowed
mba607
Fig 4. Bit transfer on the I²C-bus
master 1 loses arbitration DATA 1 ≠ SDA
DATA 1
DATA 2
SDA
SCL
S
mac609
Fig 8. Arbitration procedure of two masters
SDA
MSB
acknowledgement signal from slave
acknowledgement signal from receiver
P
Sr
SCL
S or Sr
1
2
7
8
9
ACK
3 to 8
Sr or P
START or repeated START condition
byte complete, interrupt within slave
clock line held LOW while interrupts are serviced
STOP or repeated START condition
002aac861
Fig 6. Data transfer on the I²C-bus

Figure 3.1: I2C protocol [17]

3.3: I2c bus controller

Properties of i2c bus controller (dut) [18]

1) Baud rate of 100kbps or 400kbps and it is determined by the speed bit.
2) Slave pulls SCL low to suspend master
3) Busy bit shows module is busy or not
4) Single master controller , arbitration is not supported, only supports master mode
5) It supports 3 bit addressing mode and 1 byte data transmission or receive

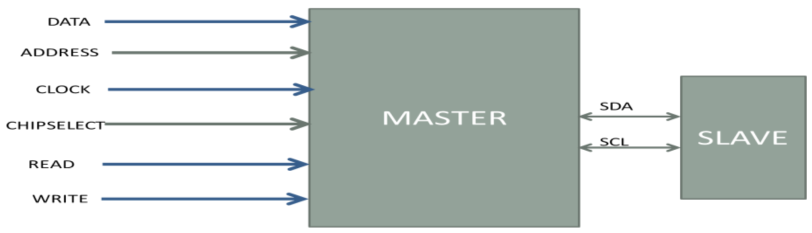


Figure 3.2: I2C signals

3.4 Verification plan

3.4.1 Features to be verified

1. Response of the DUT in different states: idle, read, write.
2. START and STOP condition.
3. Clock Synchronization between the master and slave.
4. 3- bit addressing validity.
5. All possible Master – Slave data transfer formats.

3.4.2 Stimulus Generation Plan

1. Generate slave address randomly excluding reserved address.

2. Generate random data sequence of 1 byte length.

3. Generate random delay to insert wait states and check Transmitter’s response.

4. Single master, but with different clock speeds to check synchronization .

5. No arbitration, so transactions are generated only one master.

6. Transmit address and data randomly in different operation (read or write or no operation)
7. Sample the data and address via bus monitor and display the report in scoreboard.

3.5 Designing individual open verification components

3.5.1 DUT

Open core sources for both master and slave are used.

3.5.2 Interface

Interface for master is used. Slaves are memory /registers.

3.5.3 Transaction

The transaction consists of 3 bit address and 1 byte data for the master.

3.5.4 Sequence

This class is extended from ovm_seq_item class. Randomized transactions are generated using `ovm_do and `ovm_do_with macros with inline constraints to check specific features of the i2c master controller.

3.5.5 Sequencer

This class is extended from ovm_sequencer class. An object of this class is defined by connecting it to the virtual interface in the i2c_env class by enabling automatic sequence library updating. The sequencer's constructor begins with the `super.new()` call, followed by a `` `ovm_update_seqeunce _lib _and _item `` macro. This macro expands to a function call that copies all of the statically registered sequences into the sequencer's local sequence library, which contains all of the sequences available for execution by this sequencer [1].

3.5.6 Driver

This class is extended from ovm_driver. It collects transactions from the sequencer through the seq_item_export and seq_item_port. It drives the DUT signals. It gets the transaction through the virtual interface in the i2c_env class. The primary role of the driver is to drive (in a master) or respond (in a slave) on the I2C bus according to the signal-level protocol. This is done in the run ( ) task. This task is a built in part of OVM's simulation phasing and is invoked automatically [1].

3.5.7 Monitor

Monitor is defined for both master and slave for monitoring the respective inputs. The i2c bus monitor collects i2c_transfers seen on the i2c signal-level interface and emits status updates via a state transaction, indicating different activity on the bus. The i2c bus monitor has class checks and collects coverage if checks and coverage collection is enabled. The i2c bus monitor is instantiated within the i2c environment. The i2c_env build () function has a control field called has_bus_monitor, which determines whether the i2c_bus_monitor is created or not. The bus monitor will be created by default since the default value for this control field is one [1].

3.5.8 Environment

Here all the above OVCs are instantiated along with a virtual interface. The virtual interface is passed as parameter of the constructor passed through which all the OVCs gets connected. All the virtual methods of class ovm_env are defined. The build () function of the i2c_env creates the master agents, slave agents, and the i2c bus monitor. Three properties control whether these are created [1].

3.5.9 Test

In OVM, test-cases are implemented by extending ovm_test. ovm_test provides the ability to select test to execute using the OVM_TESTNAME command line option or argument to the ovm_root::run_test task. An instance of the environment class is created and is registered with the ovm factory in the build function using type_id::create function. Inside the new ( ) constructor i2c_env class is constructed. The run () method is the only task which consumes time for execution. After completing the start_of_simulation() phase , this method is called. To stop the simulation global_stop_request() is called after a specific time instant. The print method of ovm library is called inside the run ( ) method [1].

3.5.10 Scoreboard

Scoreboard is implemented by extending ovm_scorboard. Scoreboard has 2 analysis imports. One is used to for getting the packets from the i2c_master_driver and other from the salve_driver or a slave_receiver (extended class of ovm_component class). Then the packets are compared and if they don't match, then error is asserted. Compare function of transaction class is used for comparision [1].

3.5.11 Top module

Here the DUT is connected to the test environment through the interface. OVM global task run_test () is called from here. Clock signal for the DUT is also generated in the top module. The i2c testbench (i2c_demo_tb) is instantiated in a top-level module to create a class-based simulation environment. The top module contains the typical HDL constructs and a SystemVerilog interface. This interface is used to connect the class-based testbench to the DUT. The i2c bus environment (i2c_env) inside the testbench (i2c_demo_tb) uses a virtual interface variable to refer to the actual interface [1]. The DUT instance signals are connected to the interface variable xi0 in i2c_top.sv file. Argument to run_test () method is passed in the command line using +OVM_TESTANAME= enc_base_test .

Chapter 4

RESULT

```
# OVM_INFO @ 0 [RNTST] Running test test_read_modify_write...
# OVM_INFO @ 0: ovm_test_top [test_read_modify_write] Printing the test
topology :
# ----------------------------------------------------------------------
# Name                      Type                 Size              Value
# ----------------------------------------------------------------------
# ovm_test_top              test_read_modify_w+ -          ovm_test_top@2
#   i2c_demo_tb0            i2c_demo_tb          -          i2c_demo_tb0@4
#     i2c0                  i2c_env              -                  i2c0@6
#       bus_monitor         i2c_bus_monitor      -          bus_monitor@10
#       masters[0]          i2c_master_agent     -           masters[0]@19
#       slaves[0]           i2c_slave_agent      -            slaves[0]@21
#       has_bus_monitor     integral             1                     'h1
#       num_masters         integral             32                    'h1
#       num_slaves          integral             32                    'h1
#       intf_checks_enable  integral             1                     'h1
#       intf_coverage_ena+  integral             1                     'h1
#       recording_detail    ovm_verbosity        32               OVM_FULL
#     scoreboard0           i2c_demo_scoreboard -            scoreboard0@8
#       item_collected_ex+  ovm_analysis_imp     -
item_collected_export@111
#       disable_scoreboard  integral             1                     'h0
#       num_writes          integral             32                    'd0
#       num_init_reads      integral             32                    'd0
#       num_uninit_reads    integral             32                    'd0
#       recording_detail    ovm_verbosity        32               OVM_FULL
#     recording_detail      ovm_verbosity        32               OVM_FULL
# ----------------------------------------------------------------------
#
# OVM_INFO @ 0: ovm_test_top.i2c_demo_tb0.i2c0.bus_monitor
[i2c_bus_monitor] Transfer collected :
# ----------------------------------------------------------------------
# Name                      Type                 Size              Value
# ----------------------------------------------------------------------
# i2c_transfer_inst         i2c_transfer         -    i2c_transfer_inst@13
#   addr                    integral             3                     'h0
#   read_write              i2c_read_write_enum 32                     NOP
#   size                    integral             32                    'h1
#   data                    da(integral)         1                       -
#     [0]                   integral             8                     'h0
#   wait_state              da(integral)         0                       -
#   error_pos               integral             32                    'h0
#   transmit_delay          integral             32                    'h0
#   master                  string               0
# ----------------------------------------------------------------------
#
```

Inserting another portion of the result

```
# OVM_INFO @ 100: ovm_test_top.i2c_demo_tb0.scoreboard0
[i2c_demo_scoreboard]  existing address...: 5 with data : 30
# OVM_INFO @ 100: ovm_test_top.i2c_demo_tb0.i2c0.bus_monitor
[i2c_bus_monitor] Transfer collected :
# ----------------------------------------------------------------------
# Name                      Type                 Size              Value
# ----------------------------------------------------------------------
# i2c_transfer_inst         i2c_transfer         -    i2c_transfer_inst@13
#   addr                    integral             3                     'h6
#   read_write            i2c_read_write_enum 32                   READ
#   size                  integral            32                    'h1
#   data                  da(integral)        1                       -
#     [0]                 integral            8                     'h0
#   wait_state            da(integral)        0                       -
#   error_pos             integral            32                    'h0
#   transmit_delay        integral            32                    'h0
#   master                string              0
#   end_time              time                64                    100
# ----------------------------------------------------------------------
#
# OVM_INFO @ 120: ovm_test_top.i2c_demo_tb0.scoreboard0
[i2c_demo_scoreboard]  existing address...: 1 with data : fe
# OVM_INFO @ 120: ovm_test_top.i2c_demo_tb0.i2c0.bus_monitor
[i2c_bus_monitor] Transfer collected :
# ----------------------------------------------------------------------
# Name                    Type                Size                Value
# ----------------------------------------------------------------------
# i2c_transfer_inst       i2c_transfer        -    i2c_transfer_inst@13
#   addr                  integral            3                     'h0
#   read_write            i2c_read_write_enum 32                   READ
#   size                  integral            32                    'h1
#   data                  da(integral)        1                       -
#     [0]                 integral            8                     'h0
#   wait_state            da(integral)        0                       -
#   error_pos             integral            32                    'h0
#   transmit_delay        integral            32                    'h0
#   master                string              0
#   end_time              time                64                    120
# ----------------------------------------------------------------------
#
```

Inserting another portion of the result

```
# OVM_INFO @ 1950: ovm_test_top.i2c_demo_tb0.scoreboard0
[i2c_demo_scoreboard]  existing address...: 3 with data : c6
# OVM_INFO @ 1950: ovm_test_top.i2c_demo_tb0.i2c0.bus_monitor
[i2c_bus_monitor] updating data and address
# OVM_INFO @ 1960: ovm_test_top.i2c_demo_tb0.i2c0.bus_monitor
[i2c_bus_monitor] Transfer collected :
# ----------------------------------------------------------------------
# Name                    Type                Size                Value
# ----------------------------------------------------------------------
# i2c_transfer_inst       i2c_transfer        -    i2c_transfer_inst@13
#   addr                  integral            3                     'h0
#   read_write            i2c_read_write_enum 32                  WRITE
#   size                  integral            32                    'h1
#   data                  da(integral)        1                       -
#     [0]                 integral            8                     'h0
#   wait_state            da(integral)        0                       -
#   error_pos             integral            32                    'h0
#   transmit_delay        integral            32                    'h0
#   master                string              0
#   end_time              time                64                   1960
# ----------------------------------------------------------------------
#
# OVM_INFO @ 1970: ovm_test_top.i2c_demo_tb0.scoreboard0
[i2c_demo_scoreboard]  existing address...: 0 with data : 9f
# OVM_INFO @ 1970: ovm_test_top.i2c_demo_tb0.i2c0.bus_monitor
[i2c_bus_monitor] updating data and address
# OVM_INFO @ 1980: ovm_test_top.i2c_demo_tb0.i2c0.bus_monitor
[i2c_bus_monitor] Transfer collected :
# ----------------------------------------------------------------------
# Name                       Type                Size              Value
# ----------------------------------------------------------------------
# i2c_transfer_inst          i2c_transfer        -    i2c_transfer_inst@13
#   addr                     integral            3                   'h0
#   read_write               i2c_read_write_enum 32                WRITE
#   size                     integral            32                  'h1
#   data                     da(integral)        1                     -
#     [0]                    integral            8                   'h0
#   wait_state               da(integral)        0                     -
#   error_pos                integral            32                  'h0
#   transmit_delay           integral            32                  'h0
#   master                   string              0
#   end_time                 time                64                 1980
# ----------------------------------------------------------------------
#
# OVM_INFO @ 1990: ovm_test_top.i2c_demo_tb0.scoreboard0
[i2c_demo_scoreboard]  existing address...: 2 with data : bd
# OVM_INFO @ 1990: ovm_test_top.i2c_demo_tb0.i2c0.bus_monitor
[i2c_bus_monitor] updating data and address
# OVM_INFO @ 2000: ovm_test_top [test_read_modify_write] Calling
global_stop_request() to end the run phase
# OVM_INFO @ 2000: ovm_test_top.i2c_demo_tb0.scoreboard0
[i2c_demo_scoreboard] Reporting scoreboard information...
# ----------------------------------------------------------------------
# Name                       Type                Size              Value
# ----------------------------------------------------------------------
# scoreboard0                i2c_demo_scoreboard -          scoreboard0@8
#   item_collected_export    ovm_analysis_imp    -
item_collected_export@111
#     recording_detail       ovm_verbosity       32             OVM_FULL
#   disable_scoreboard       integral            1                   'h0
#   num_writes               integral            32                  'd0
#   num_init_reads           integral            32                 'd89
#   num_uninit_reads         integral            32                  'd0
#   recording_detail         ovm_verbosity       32             OVM_FULL
# ----------------------------------------------------------------------
#
#
# --- OVM Report Summary ---
#
# ** Report counts by severity
# OVM_INFO :  258
# OVM_WARNING :    0
# OVM_ERROR :    1
# OVM_FATAL :    0
# ** Report counts by id
# [RNTST               ]     1
# [i2c_bus_monitor     ]   165
# [i2c_demo_scoreboard ]    90
# [read_modify_write_seq]     1
# [test_read_modify_write]     2
# ** Note: $finish    : /home/NIS/BTECH0812/jyoti/ovm-
2.0/src//base/ovm_root.svh(304)
#    Time: 2 ps  Iteration: 8  Instance: :i2c_top:ovm_root::run_test
```

CHAPTER 5

ANALYSIS OF RESULT OBTAINED FROM OVM OUTPUT

At beginning of the simulation which test module is called from top module is displayed. Here test_read_modify_write test is called from i2c_top module.Then associate files are loaded.

The 1st table displays the variables and data types created by OVM .According to this table one master agent and one slave agent is created. Num_masters represents number of master present during the transaction which is one. Similarly num_slaves represent number of slave participating in each transaction. One scoreboard instance is created of type i2c_demo_scoreboard. Intf_checks_enable enables scoreboard and monitor to check size of each transaction and validity of slave address. Slave agent creates random memory address .

There after each table contains information about each transaction. Each transaction contains address, data, read write or no operation performed ,transmission delay time, error position. The operations are generated randomly. At first bus state is in idle case, it waits for start operation to start transaction. So for first cases no operation is done. During no operation time no memory is accessed, no data transfer occurs , hence no transaction is performed. Time taken for each transaction is zero. We have taken ideal data transmission case where no time delay is incorporated and no error position have been transmitted. Addition of error position and time delay depends upon the DUT and the DESIGNER.

Now after no operation READ or WRITE operation is done. The operation is performed randomly. **Addr** represents address, data represents data represents data.**OVM** show**s** time of each transaction .As previously mentioned each transaction has no transmit delay each transaction takes 20 unitsecond for transfer. During Reading operation address containing data is displayed. Master reads data from memory. Some cycles of Read operation is performed. After read operation Write operation is done. Data to be written in the slave(memory) is provided randomly, it has no relation with the Read Operation. Hence it shows bidirectional nature of SDA line. For further improvement(Future development in real time operation) a microprocessor or microcontroller can be interfaced with the master to retrieve data from a memory location and vice versa.

Scoreboard shows what transaction is done on which memory location. Notice the address and data size mentioned in the dut is 3 bit and 8 bit respectively. The transactions made are all within the size limit. If there is an error in size overflow it will show an error.

Similarly if sent data item and received data item mismatch then error message will be displayed. After each successful transaction address and data are updated. If address mismatch occur, or data transaction fails due to error in transaction then address and data will not be updated. As there is not any such type of scenario, data transaction is occurring smoothly, so there is no error in the DUT and transaction. One ovm error report is generated due to simultaneous on of both read and write pin, which is an error due to randomization.

After a number of transactions OVM summarizes all the transactions. It shows number of times the transaction is done and how many times error has occurred, number of instances of scoreboard ,monitor created and shows run method of which test case is being called and total time consumed during the transaction.

CHAPTER-6

CONCLUSION AND FUTURE WORK

Automated test-cases are generated and applied to our DUVs. Functionality of The I2C Bus Controller Was Verified From The tabular representation. Developing the Verification IP for any design (DUT) becomes very simple by using OVM. OVM verifies the design in most effective way.

For further implementation this controller can be interfaced with any microprocessor or microcontroller who can generate data, address and other signals in real time designs or SOCs. OVM can be used to verify the system operation as test cases are easier to create in OVM and it can be verified of the system’s response to real time data through scoreboard. Any data mismatch will be considered as error and the error position can be determined.

References:

1. Cadence Designs Systems and Mentor Graphics Inc., “Open Verification Methodology User Guide” Version 2.0, Sept.2008 available from http://www.ovmworld.org
2. [Online] http://www.testbench.in/OT_01_INTRODUCTION.html
3. [Online] http://www.ovmworld.org/white_papers.php
4. [Online] http://www.mentor.com/products/fv/methodologies/abv/
5. [Online] http://www.mentor.com/products/fv/methodologies/pdv/
6. [Online] http://www.testbench.in/IF_04_CLOCKING_BLOCK.html
7. [Online] http://www.testbench.in/IF_05_VIRTUAL_INTERFACE.html
8. [Online]http://www.testbench.in/CM_06_PHASE_3_ENVIRONMENT_N_TESTCASE.html
9. [Online] http://www.systemverilog.org/products/products_solu.html
10. [Online] http://testbench.in/AS_01_INTRODUCTION.html
11. [Online] http://www.coursehero.com/file/5762297/PhillipsI2Cmanual/
12. [Online] http://www.coursehero.com/file/3904209/PhillipsI2Cmanual/
13. [Online] http://www.testbench.in/TS_24_VERIFICATION_PLAN.html
14. [Online] http://www.testbench.in/SL_03_VERIFICATION_PLAN.html
15. [Online] http://www.ovmworld.org/datasheets.php
16. [Online] http://www.i2c-bus.org/
17. Irazabal Jean-Marc, Blozis Steve. AN10216-01 I2C MANUAL : Philips Semiconductors, 2003
18. [Online] http://read.pudn.com/downloads163/sourcecode/embed/740675/I2C_Master.v__.htm
19. [Online] http://www.testbench.in/UT_01_INTRODUCTION.html
20. [Online] http://www.mentor.com/products/fv/abv/questa_sv/low-power.cfm
21. [Online] http://www.testbench.in/TS_05_LINEAR_TESTBENCH.html
22. [Online] http://www.scribd.com/doc/50646205/3/Coverage-Driven-Verification-CDV

APPENDIX A

Top module

```
`include "i2c_master.v"//dut
`include "i2c_if.sv"//interface

module i2c_top;

  `include "i2c.svh"//file should
exist
  `include "i2c_test.sv"

  i2c_if xi0();

  i2c_master dut(
    xi0.addr,
    xi0.in_data,
    xi0.out_data,
    xi0.sda,
    xi0.scl,
    xi0.cs,
    xi0.rd,
    xi0.wr,
    xi0.clk

  );

  initial begin
    run_test();
  end

  initial begin

       xi0.clk <= 1'b1;
    xi0.cs=1'b1;//cs alays 1 for
both read write process


  //Generate Clock
  always
    #5 xi0.clk = ~xi0.clk;

endmodule
```

Data item

```
typedef enum { NOP,
               READ,
               WRITE
             } i2c_read_write_enum;


class i2c_transfer extends
ovm_sequence_item;

  rand bit [2:0]            addr;
  rand i2c_read_write_enum read_write;
  rand int unsigned         size;
  rand bit [7:0]            data[];
  rand bit [3:0]            wait_state[];
  rand int unsigned         error_pos;
  rand int unsigned         transmit_delay
= 0;
  string                    master = "";
  string                    slave = "";
  constraint c_read_write {
    read_write inside { READ, WRITE };
  }
  constraint c_size {
    size inside {1,2,4,8};
}
  constraint c_data_wait_size {
    data.size() == size;
    wait_state.size() == size;
  }
  constraint c_transmit_delay {
    transmit_delay <= 10 ;
  }

  `ovm_object_utils_begin(i2c_transfer)
    `ovm_field_int      (addr,
OVM_ALL_ON)
    `ovm_field_enum
(i2c_read_write_enum, read_write,
OVM_ALL_ON)
    `ovm_field_int      (size,
OVM_ALL_ON)
    `ovm_field_array_int(data,
OVM_ALL_ON)
    `ovm_field_array_int(wait_state,
OVM_ALL_ON)
    `ovm_field_int      (error_pos,
OVM_ALL_ON)
    `ovm_field_int      (transmit_delay,
OVM_ALL_ON)
    `ovm_field_string   (master,
OVM_ALL_ON|OVM_NOCOMPARE)
    //`ovm_field_string   (slave,
```

Interface

```
interface i2c_if ();//interface
 bit                has_checks =
1;
  bit                 has_coverage
= 1;

logic              clk;
logic          [2:0]addr;   wire
logic          [7:0]in_data;
logic           [7:0]out_data;
wire logic              sda;
wire logic              scl;
logic              cs;
logic              rd;
logic              wr;
endinterface
```

```
OVM_ALL_ON|OVM_NOCOMPARE)
`ovm_object_utils_end

  // new - constructor
  function new (string name =
"i2c_transfer_inst");
    super.new(name);
  endfunction : new

endclass : i2c_transfer

`endif
```

APPENDIX B:

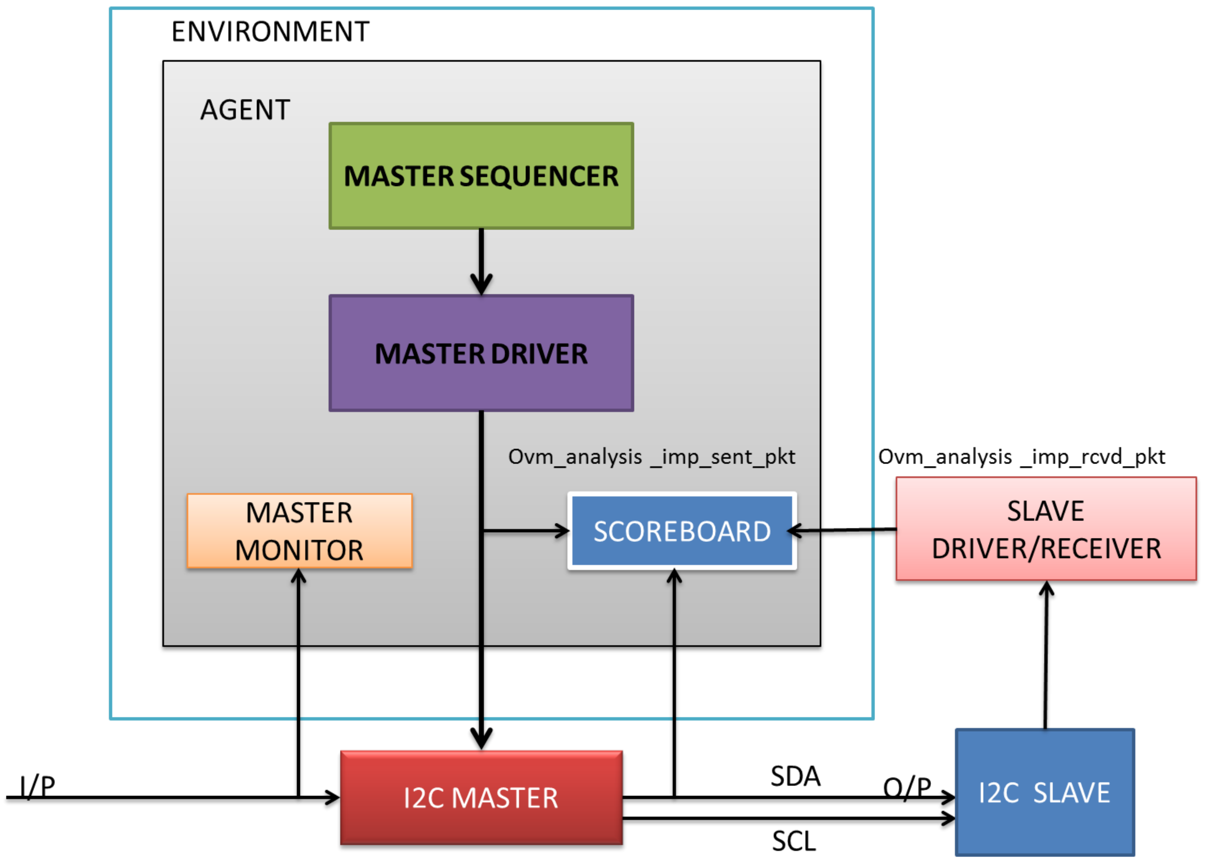


Figure A: Verification Environment For I2C Bus

Master sequencer generates the data and is sent to the Master core through the Driver. Other necessary master signals are also generated. Serial data that is to be transmitted by the master are given to the master by the master driver and the driver sends the same data to scoreboard as the sent_pkt through ovm mail box (ovm_component object). The data received by the slave are feed back to the scoreboard via slave_driver for comparision as rcvd_pkt through the mail box. Then here the sent and received data item are compared.