OVM COMPLIANT VERIFICATION FOR A WISHBONE COMPATIBLE I^2C MASTER CONTROLLER CORE

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

BY

ABINASH MOHANTY
10609002

MANORANJAN XESS
10609004

MADHURITA MAHAPATRA
10607031

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 **"DESIGN OF OVM COMPLIANT VERIFICATION FOR A WISHBONE COMPATIBLE I^2C MASTER CONTROLLER CORE"** submitted by Abinash Mohanty (10609002, Dept of ECE), Manoranjan Xess (10609004, Dept of ECE) and Madhurita Mahapatra (10607031, Dept of ECE) in partial fulfillment of the requirements for the award of Bachelor of Technology degree in Electronics and Communication 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 matter embodied in the thesis has not been submitted to any other University/Institute for the award of any Degree or Diploma.

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

ACKNOWLEDGEMENT

We take this opportunity to thank all individuals without whose support and guidance we could not have completed our project in this stipulated period of time. First and foremost 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, 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 for their generous help in various ways for the completion of this thesis. We also extend our thanks to our fellow students for their friendly co-operation.

ABINASH MOHANTY
Roll No. 10609002
Department ECE

MANORANJAN XESS
Roll No. 10609004
Department ECE

MADHURITA MAHAPATRA
Roll No. 10607031
Department ECE

ABSTRACT

Increasing design complexity and concurrency of Integrated Circuits has made traditional directed testbenches an unworkable solution for testing. Today, testing as a word has been substituted with verification. Verification engineers have to ensure what goes to the factory for manufacturing is an accurate representation of the design specification. Inter Integrated Circuit (I^2C) bus is a very widely used communication protocol in embedded system design due to its hardware simplicity and high data transfer rates capability. Most ICs incorporate I^2C interface. Thus the ASIC design process of these ICs calls for robust, independent and exhaustive verification to reduce the risks of their failures. Open Verification Methodology (OVM) is an open source verification methodology library intended to run on multiple platforms and be supported by multiple EDA vendors. This thesis attempts to study and hence introduces a comprehensive verification environment for the latest specifications of the I^2C bus protocol realized in the OVM platform, a new industry standard for comprehensive verification due to its rich base classes and OOP features. This work has been challenging since very few work has been reported in this domain for reference.

Keywords: *OVM, I^2C bus protocol, code coverage, functional coverage, OOP*

CONTENTS

INTRODUCTION

An embedded system is a computer system designed to perform one or a few dedicated functions often with real-time computing constraints. Embedded systems are controlled by one or more main processing cores that are typically either a microcontroller or a digital signal processor (DSP). Embedded Systems talk with the outside world via peripherals (1). All the peripherals are connected to the processing cores using bus lines following a specific protocol. Thus bus and its protocol are among the most important parts of the system.

To maximize hardware efficiency and circuit simplicity, Philips Semiconductors (now NXP Semiconductors) developed a simple bidirectional 2-wire bus for efficient inter-IC control. This bus is called the InterIC or I^2C-bus. Through years of continuous improvement the I^2C-bus has become a world standard that is now implemented in over 1000 different ICs manufactured by more than 50 companies. All I^2C-bus compatible devices incorporate an on-chip interface which allows them to communicate directly with each other via the I^2C-bus (2).

Logical and functional flaws continue to be the leading cause of costly design respins (3). A primary purpose for functional verification is to detect failures so that bugs can be identified and corrected before it gets shipped to costumer (4). Thus there lies a requirement of an efficient verification IP for I^2C bus interface since it's implemented on a large number of ICs.

In this thesis, an efficient and comprehensive verification environment has been implemented for I^2C bus interface using a new methodology called OVM.

Chapter 1

Verification

1.1 What is verification?
1.2 The importance of verification
1.3 Test-benches vs. verification
1.4 The verification process
1.5 The verification plan
1.6 Methodologies for functional verification

1.1 WHAT IS VERIFICATION?

Verification is a process used to demonstrate that the intent of a design is preserved in its implementation (5). Verification is always done in parallel to the design creation process that is verification is carried out at each step of manufacturing process (6).

Verification is generally viewed as a fundamentally different activity from design. This split has led to the development of narrowly focused language for verification and to the bifurcation of engineers into two largely independent disciplines. This specialization has created substantial bottlenecks in terms of communication between the two groups. SystemVerilog addresses this issue with its capabilities for both camps. Neither team has to give up any capabilities it needs to be successful, but the unification of both syntax and semantics of design and verification tools improves communication. For example, while a design engineer may not be able to write an object-oriented test-bench environment, it is fairly straightforward to read such a test and understand what is happening, enabling both the design and verification engineers to work together to identify and fix problems. Likewise, a designer understands the inner workings of his or her block, and is the best person to write assertions about it, but a verification engineer may have a broader view needed to create assertions between blocks.

Another advantage of including the design, test-bench, and assertion constructs in a single language is that the test-bench has easy access to all parts of the environment without requiring specialized APIs. The value of an HVL is its ability to create high-level, flexible tests, not its loop constructs or declaration style. SystemVerilog is based on the Verilog constructs that engineers have used for decades (5).

1.2 THE IMPORTANCE OF VERIFICATION:

Verification is one of the most vital sections of any design industry. Its importance is stated as follows (5):

- ***70% of design effort goes to verification***

  Today the Integrated Circuits (IC) design industry is associated with very complex designs, reusable intellectual property (IP), and system-on-a-chip (SoC) designs; this makes verification a very vital part of the industry and takes about 70% of the design effort. The number of engineers dedicated to verification can be up to twice that of RTL designers.

- ***Verification is on the critical path***

  Given the amount of effort demanded by verification, the shortage of quality hardware design and verification engineers, and the quantity of code that must be produced, it is no surprise that in all projects, verification rest separately in the critical path.

- ***Verification time can be reduced through abstraction***

  Providing higher abstraction levels has enabled us to work more efficiently forgetting about the low-level details.

- ***Using abstraction reduces control over low level details***

  Higher abstraction levels are usually accompanied by a reduction in control and thereafter must be chosen wisely. Higher abstraction levels require additional training for understanding the abstraction mechanism and how the desired effect is produced. The verification process can use higher abstraction levels by working transaction-or-bus-cycle levels, instead of always dealing with low-level zeroes and ones.

- ***Verification time can be reduced through automation***

  By automation we can do something else and let the machine complete the task automatically, faster and gives predictable results. Requirement of automation are standard processes with well-defined inputs and outputs. All processes cannot be automated.

- ***Randomization can be used as an automation tool***

  Randomization is very useful in the verification process. By the use of constrained random generator one can produce valid inputs within the bounds of a particular domain taking all corner cases into consideration.

1.3 TEST-BENCH VS. VERIFICATION:

The term "test-bench" refers to simulation code used to create a predetermined input sequence to a design, then optionally to observe the response. A test-bench is commonly implemented using VHDL, Verilog, e or OpenVera, but it may also include external data files or C routines.

Verification is different from a test-bench. It is a process used to determine the extent to which the intent of a design is preserved in its implementation.

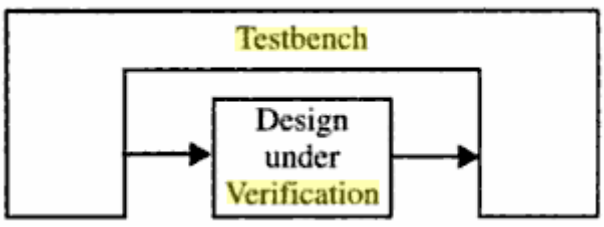


Figure 1.1 - A generic structure of test-bench and design under verification.

As far as design is concerned, test-bench provides a model for the design. It provides input to the design and watches any output. But verification determines what input pattern has to be

supplied to the design and what is the expected output of a properly working design when submitted to those inputs (6).

Challenges faced due to directed tests:

- Traditional directed tests are written by humans and humans can't anticipate all possible scenarios
- Directed tests can't stimulate and verify concurrency
- Code/Structural Coverage don't catch bugs due to concurrency

Solutions via Verification Environment constitute:

- Testbench automation (SystemVerilog) enables Constrained Random Verification, covering more state space than directed tests
- Assertion/Property Based Verification ensures the design operates as expected
- Formal Verification validates hardware exhaustively
- Total Coverage Model lets you know when you're done

1.4 THE VERIFICATION PROCESS:

Hardware designs create devices that perform a required task based on a design specification. The purpose of a verification engineer is to make sure the device can accomplish that task successfully — that is, the design is an accurate representation of the specification. Due to bugs we get discrepancy.

The hardware specification for a block, usually the human language description, is interpreted by the designer to create the corresponding logic in a machine-readable form which is usually RTL code. For this the designer needs to understand the input format, the transformation function, and the format of the output. A verification engineer also reads the hardware specification to create a verification plan.

After this, the one looks for discrepancies at boundaries between blocks. Interesting problems arise when two or more designers read the same description yet have different interpretations. For a given protocol, what signals change and when?

To simulate a single design block, we need to create tests that generate stimuli from all the surrounding blocks. The benefit is that these low-level simulations run very fast. The demerit is that the multiple block simulations may uncover more bugs, but they also run slower.

At the highest level of the DUT, the entire system is tested, but the simulation performance is greatly reduced. All I/O ports are active, processors are crunching data, and caches are being refilled. Because of all this action, data alignment and timing bugs occur. At this level one is

able to run sophisticated tests that have the DUT executing multiple operations concurrently so that as many blocks as possible are active

After verifying that the DUT performs its designated functions correctly, we need to see how it operates when there are errors.

Commercially successful verification IP solutions must include the following key elements:

- Expertise and participation in developing interface standards.
- A flexible approach to encapsulate domain expertise and enable modeling of vendor-specific subsets of the interface standard (includes BFM).
- Complete assertion libraries (checkers or monitors) for verifying compliance with the interface specification.
- Compliance test suites to exercise the design with compliant and noncompliant traffic.
- Application-specific traffic generation to support system-level verification.
- A coverage engine to identify corner-cases and test completeness.
- Portability across testbench environments and verification languages to enable quality through a single, reusable solution for an industrywide user base.
- Easy integration to other tools for coverage, transaction analysis and debug.

As the IP market develops to support SoC design, a parallel market for verification IP is accelerating. Verification IP is essential for the success of the IP industry and SoC design in general. Commercial verification IP is being leveraged successfully in IP development and deployment, as well as system integration and verification (7).

1.5 THE VERIFICATION PLAN:

It is a fact that in design and verification, the latter task that dominates time scales. More effort is required to verify a design than to write the RTL code for it. A lot of effort goes into specifying the requirements of the design. Given that verification is a larger task, even more effort should go into specifying how to make sure the design is correct. The verification plan is that specification. And that plan must be based on the intent of the design, not its implementation.

Every design is specified in stages: first requirements, then architecture and finally detailed implementation. The verification planning process follows similar steps.

Verification Implementation Plan:

The primary aim of implementing the functional verification plan is to ensure that the implementation culminates in exhaustive coverage of the design and its functionality within the project time scales. The implementation is based on the requirements of the verification environments. The implementation plan should start as early as possible in the project lifecycle. Ideally, it should be completed before the start of the RTL-coding phase of the project and before any verification test bench code is written. This step is necessary to produce a design with a high degree of probability of being bug-free.

Response Checking:

The enumeration of all errors must be detected by the verification environment. These detection mechanisms are required to have a strategy for predicting the expected response and to compare the observed response against those expectations. With traditional directed test cases, because the stimulus and functionality of the design are known, the expected response may be intellectually derived up front and hard-coded as part of the directed test. With random stimulus, although the functionality is known, the applied stimulus is not. The expected response is computed based on the configuration and functionality of the design. The observed response is then compared against the computed response for correctness.

Assertions:

Assertion means that a statement that is true. When verification is concerned, an assertion is a statement of the expected behavior. A detected discrepancy in the observed behavior leads to an error. Thus the entire test bench can be considered as one big assertion: thus it's a statement of the expected behavior of the design. But in design verification assertion refers to a property expressed using a temporal expression.

Accuracy:

The comparison functions in its simplest form compare the observed output of the design with the predicted output on a cycle-by-cycle basis. This approach requires the response to be accurately predicted down to the cycle level, which is a difficult task.

Score boarding:

A scoreboard predicts the response of the design dynamically. The stimulus applied to the design is provided to a transfer function. It is the transfer function that performs all transformation operations on the stimulus in order to produce the form of the final response then inserts it in a data structure. Scoreboarding works well for verifying the end-to-end response of a design and the integrity of the output data. It is not well suited for detecting errors whose symptoms of failures are not obvious at the granularity of a single response (8).

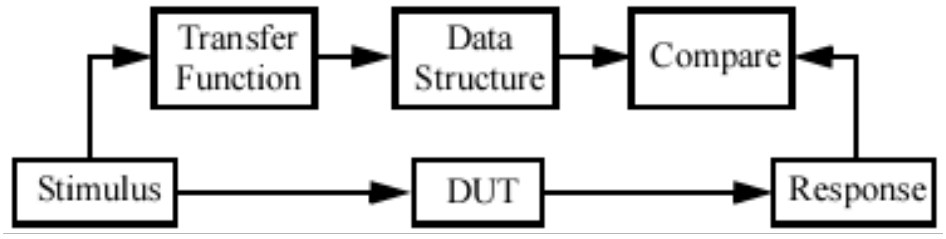


Figure 1.2 - Score boarding

1.6 Methodologies for Functional Verification:

Backbone of a solid verification strategy is Verification Methodologies. Mentor Graphics is actively driving advanced methodologies and their standardization across the industry.

Three basic methodologies used are:

- ***Assertion based Verification:***

  Here the designers use assertions to capture specific design intent and through simulation or formal verification, or emulation of these assertions, verify that the design correctly implements that intent (9).

- ***Open Verification Methodology:***

  The OVM is the first truly open, interoperable, and proven verification methodology based on the SystemVerilog IEEE 1800 language and delivers an open and unified class library and methodology for interoperable verification IP (VIP) (10).

- ***Processor driven Verification:***

  Current techniques of applying test vectors from an HDL test-bench only begin to mimic processor bus behavior. The introduction of processor-driven test benches into the existing verification methodology enables real-world verification and extensive reuse of testbench software throughout the project (11).

Chapter 2

OVM(open verification methodology)

2.1 Introduction to OVM
2.2 Advantages of OVM
2.3 OVM components or OVC

2.1 INTRODUCTION TO OVM:

The Open Verification Methodology (OVM) is developed as a joint initiative between Mentor Graphics and the Cadence Design Systems which provides the first open and interoperable, SystemVerilog verification methodology in the design industry. The OVM provides a library of base classes allows users to create modular and reusable verification environments in which components can talk via standard transaction-level modeling or TLM interfaces. It also enables intra- and inter-company reuse through a very common methodology which uses classes for development of stimulus sequences and block-to-system reuse.

Multiple verification platforms support it. It is ideally suited to speed verification for both expert and non-expert verification engineers. It has been built on the success of the Advanced Verification Methodology (AVM) from Mentor Graphics and Cadence developed Universal Reuse Methodology (URM), the OVM brings the combined power of these companies together to deliver using SystemVerilog. The OVM offers interoperability mechanisms for verification IP (VIP), RTL models, and integration with commonly used languages in production flows.

Figure 2.1 – SystemVerilog Hierarchy

In general the major EDA vendors adopt an approach which is similar to as given in Figure 2.1, but the details differed. Each vendor had its own methodology, a different class library, and language with different features used in the library and recommended by its methodology. When different simulators which supported different subsets of different languages were coupled with that some methodologies which were proprietary and restricted which implied that it was not possible to run the VIP and verification code on different or multiple simulators.

Even assuming that all those major simulators would have eventually supported the same set of language features, the existence of this multiple class libraries and methodologies implied that VIP was not at all interoperable. Since the different methodologies as shown in Figure 2.1 SystemVerilog Verification defined different mechanisms for communicating Hierarchy

between VIP and test-benches, combining components from different vendors was such a challenge that it offset the benifit of licensing pre-verified VIP .

The OVM spans the class library and layers of methodology as shown in the Figure 2.2 which gives the SystemVerilog verification hierarchy,

The class library is supported by both the Cadence Incisive verification and Mentor Graphics Questa platforms. Any VIP or test-benches that have been built using this library will run on either platform with certain conversions, translations, or extra effort. OVM is delivered in open-source format which is under the terms of the Apache2.0 license agreement, the code can run on any simulator that can support the SystemVerilog standard.

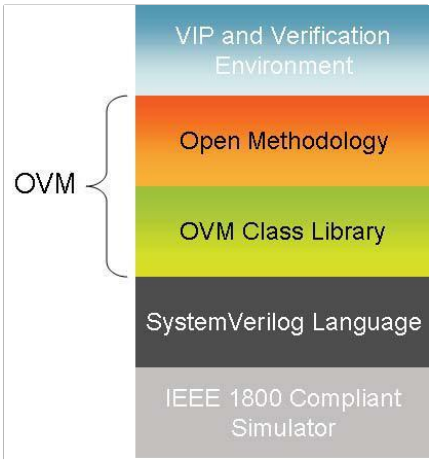


Figure 2.2 - OVM Verification Hierarchy

The OVM is “new” in as it was recently announced. However, the OVM has already been proven because it was built on well-established verification technologies and methodologies, which reflects more than ten years of industry best practices. The OVM is based on, and backward compatible with, the Cadence URM 6.2 versions and Mentor Graphics AVM 3.0 (12).

THE OVM LIBRARY

Figure 2.3 is a Unified Modeling Language (UML) diagram of the library given by OVM. The library and methodology provides all necessary tools and the technologies to construct, reuse the constrained-random and coverage-driven test-benches. The OVM provides the TLM-based infrastructure for modeling and building modular verification components which are reusable and that communicate through transaction-level interfaces.

The OVM class library allows users in the creation of sequential constrained-random stimulus which helps collect and analyze the functional coverage and the information obtained, and include assertions as members of those configurable test-bench environments. Specific features include:

- TLM communication which provides foundation for the connection of verification components in order to facilitate reusability and modularity
- User-extensible phasing commonly used to coordinate the execution activity of all environment components.

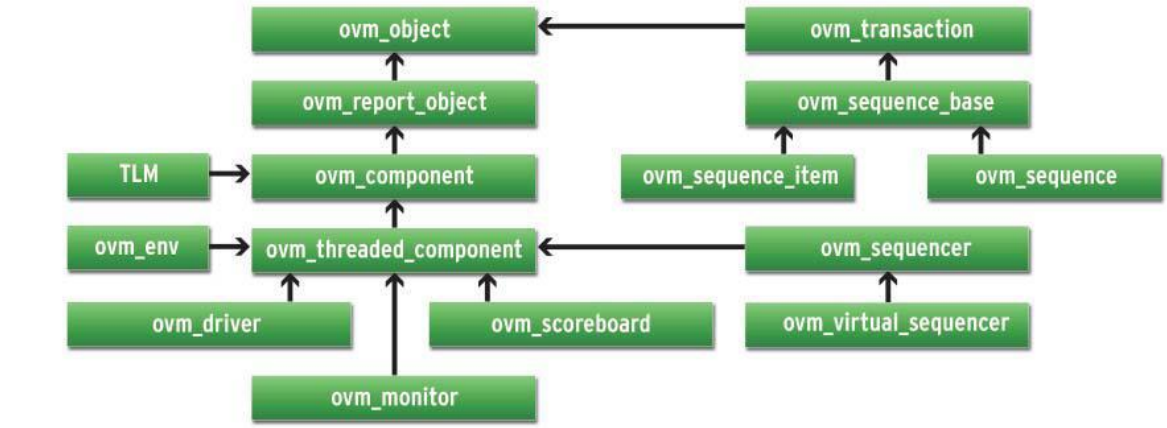


Figure 2.3 - The OVM Class Library

- Ability to transform test-bench environments on-the-fly and write several and multiple tests from the same base environment usage of minimal code and code changes
- Common configuration interface where all components may be customized on a per-type or per-instance basis by reusing the underlying code
- Message reporting and common formatting interface

Phasing and Execution Management:

The OVM defines a series of phases for the execution of simulation and through which all verification components have to go through . Once the top-level environment has been constructed in the new() phase, child components are declared and instantiated, and configured hierarchically during the post_new() method. The connections between components which are defined during the elaboration() phase, and the connections obtained are checked and resolved in the post_elaboration() phase. At the end of the post_elaboration(), all components in the environment are noted, allocated, connected, and made ready for use.

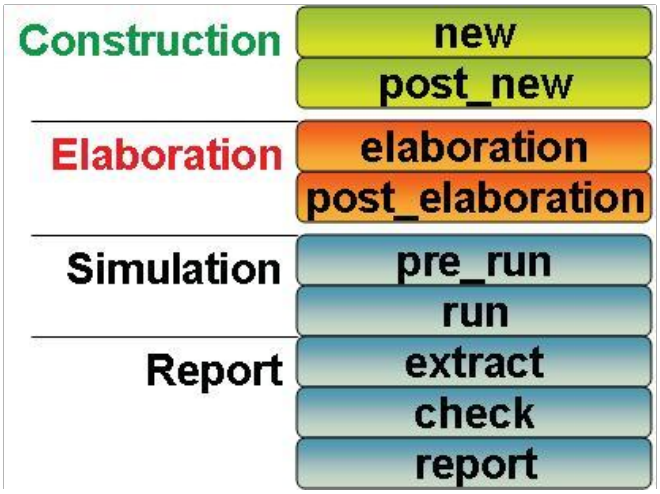


Figure 2.4 - Simulation Phases of the OVM

To further customize and configure verification components of the test and/or the design under test (DUT) allowed by the pre_run() prior to executing the test in the run() phase. At the conclusion of run(), which is a task, simultaneously three reporting phases are executed: extract() which allows results to be gathered to and from specific components, check() which validates the results to determine the pass/fail status of the test, and report() that lets each component report and its results and status to a log file or the display, and using the message severity and formatting routines.

For successful verification, the projects require more than a just a standard language. A sophisticated methodology is required to build leading-edge test-benches to ensure interoperability, and support verification reuse. The co-development and approval by Mentor and Cadence has given the OVM credibility and viability to answer the industry's concerns. The OVM is clearly the only interoperable, open, and proven verification methodology (12).

2.2 BENEFITS OF OVM:

- ***Interoperability***

  It is jointly developed and tested on both the Mentor Graphics Questa® and Cadence Incisive® verification platforms which utilize standard TLM interfaces to improve modularity and reuse. It also provides architecture, utilities and infrastructure, including specialized base classes, to create higher-level verification components for block-to-system and project-to-project reuse. Moreover it can be adopted incrementally to enhance existing module-based test-bench methodologies. It is also backward compatible with AVM 3.0 and URM 6.2 (13).

- ***Advanced Capabilities***

  The unsurpassed flexibility and configurability of OVM allows test case customization without writing the underlying code again. The standard test phases for coordinating activities of multiple components, which are customizable to suit any

organizational needs. Unified customizable message formatting and reporting is added along with powerful and flexible sequence specification for test scenarios, from simple stimulus to complex multi-layer protocols (13).

- ***Open Source***

  Distributed under the standard open-source Apache™ 2.0 license, the OVM base classes and examples can be downloaded and distributed freely. Access to the source code simplifies debug, enhances code quality, facilitates collaboration, and ensures adherence to the IEEE-1800 SystemVerilog standard. The OVM code runs on any SystemVerilog compliant simulator, and the open-source distribution greatly accelerates the adoption of SystemVerilog throughout the verification ecosystem (13).

- ***Verification Infrastructure***

  The OVM provides all the building blocks needed to construct a suitable test environment. Components may be encapsulated and instantiated hierarchically and are controlled through an extendable set of phases to initialize, run, and complete each test. These phases are defined in the base class library but can be extended to meet specific project needs (13).

- ***Transaction-Level Modeling***

  The OVM components can communicate via standard a TLM interface, which improves reuse. The TLM Standard was originally developed by OSCI, and was defined as a standard set of interface methods to define communication semantics and implementations. SystemVerilog implementation of TLM is used in the OVM, a component may use its interface to communicate with other component. Hence, one component can be connected at the transaction level to many others that have been implemented at multiple levels of abstraction (13).

- ***OVM Messaging***

  It facilitates debugging, results checking, and overall consistency. The OVM includes a standard set of methods for reporting messages to s*end out* and to many text files. Messages may be controlled using customizable filters and verbosity on an individual or hierarchical basis (13).

- ***OVM Automation***

  OVM users have a facility to simplify environment creation through advanced automation. Macros drastically and dramatically reduce coding (thus making it faster) and improve readability. Otherwise, function calls can be used by users (13).

- ***Flexible Configuration***

  To decouple the “test” from the “testbench.”, the OVM uniquely provides three forms of configuration .to perform verification the test-bench is the complete topology for the specific components while the test becomes a straight forward configuration of

that topology. These three forms of configuration those are all available at run time (13).

- ***Sequential Stimulus***

  The OVM provides supports building of hierarchical sequences through classes thus making it easy to specify complex layered stimulus. OVM separates the test behavior from the structure of the test-bench and hence allowing greater modularity and reuse. At every level of abstraction, sequences can enable simple test writer interface which include built-in semantics for controlled randomization of transactions (13).

2.3 OVM COMPONENTS OR OVC:

An OVM test-bench is composed of reusable verification environments called OVM verification components (OVCs). Each OVC follows a consistent architecture and consists of a complete set of elements for stimulating, checking, and collecting coverage information for a specific protocol or design. The OVC is applied to the device under test (DUT) to verify the implementation of the protocol or design architecture. OVCs expedite creation of efficient test-benches for your DUT and are structured to work with any hardware description language and high-level verification language including Verilog, VHDL, e, SystemVerilog, and SystemC.

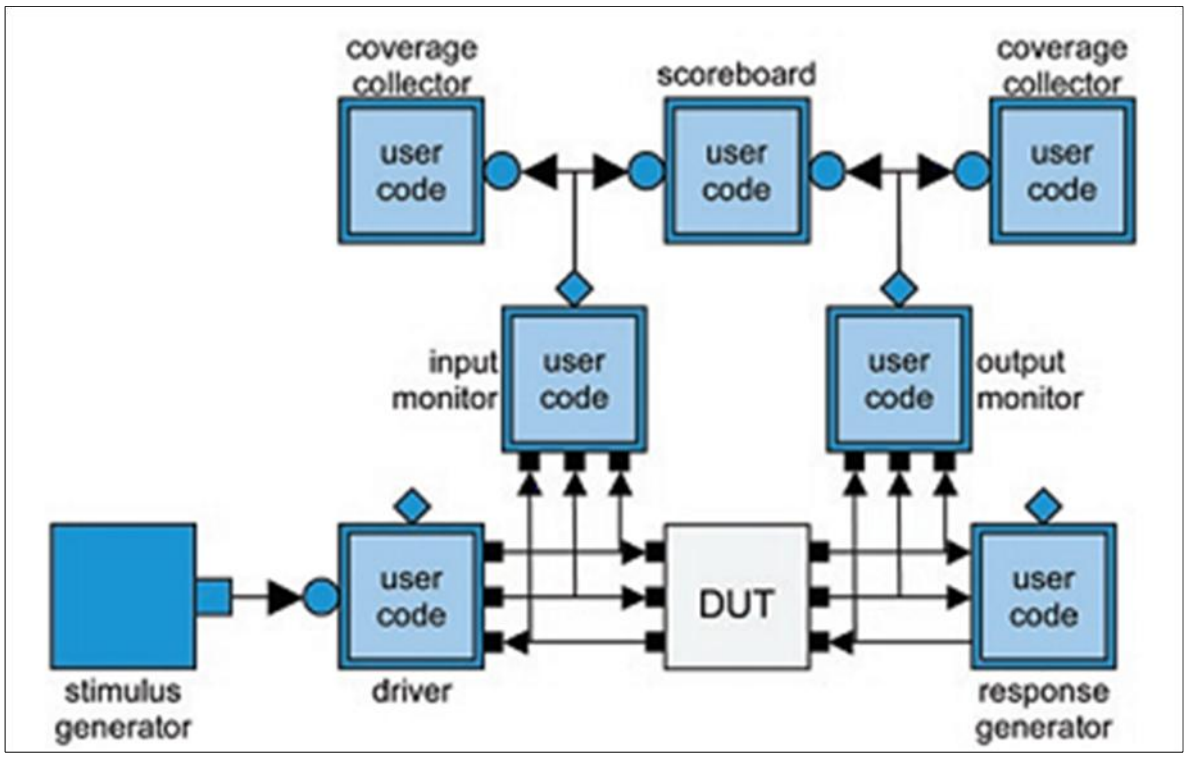


Figure 2.5 - OVM Verification Components in a generalized Verification Environment

In Figure 2.5 are most of the OVCs used in an OVM compliant verification environment. The OVCs written in SystemVerilog code is structured as follows (14):

— Interface to the design-under-test

— Design-under-test (or DUT)

— Verification environment (or test bench)

    — Transaction

    — Sequencer (stimulus generator)

    — Driver

    — Top-level of verification environment

        — Instantiation of sequencer

        — Instantiation of driver

— Response checking

    — Monitor

    — Scoreboard

— Top-level module

    — Instantiation of interface

    — Instantiation of design-under-test

    — Test, which instantiates the verification environment

    — Process to run the test

2.3.1 Interface:

Interface is the actual link between the design-under-test and the verification environment. It is defined in the same way as is defined in a SystemVerilog interface. The interface encapsulates all the pin-level connections that are made to the DUT. An interface is a bundle of nets or variables.

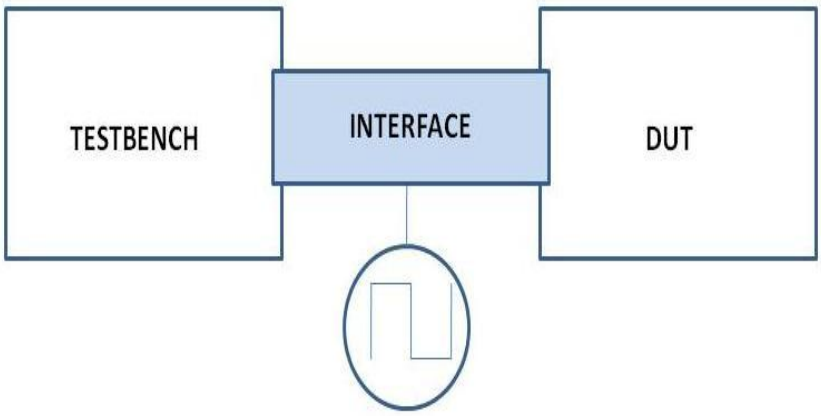


Figure 2.6 - Schematic of use of an Interface

2.3.2 Design Under Test (DUT):

DUT completely describes the working model which has to be tested and verified.

2.3.3 Transaction:

A transaction is a collection of related data items that are to be passed as a single unit around the verification environment. The sequencer which creates the random transactions are then retrieved by the driver and hence used to stimulate the pins of the DUT. Since we use a sequencer, the transaction class has to be derived from the ovm_sequence_item class, which is a subclass of ovm_transaction.

As transaction may need to be copied, compared, printed, packed and unpacked when objects are passed around the verification environment. The ovm_object_utils and ovm_field macros do the above said necessary things automatically.

The given field should be copied, printed, included in any comparison for equality between two transactions is indicated by the OVM_ALL_ON flag. The flags OVM_DEC and OVM_BIN indicate the radix of these fields to be used when printing the given field.

2.3.4 Sequencer:

A sequencer is an advanced stimulus generator that controls the transactions that are provided to the driver for execution. It allows the addition of constraints to the data item generated in the sequence, thus bringing forth the corner cases.

2.3.5 Driver:

A driver is an active entity that emulates logic to drive transactions into the DUT. A typical driver repeatedly receives a data item and drives it to the DUT by sampling and driving the DUT signals.

The run method is one of the standard hooks called back in each of the phases of elaboration and simulation. It contains the main behavior of the component to be executed during simulation. This run method contains an infinite loop to wait for some time, get the next

transaction from the seq_item_port, then wiggle the pins of the DUT through the virtual interface mentioned above.

2.3.6 Monitor:

A monitor is a passive entity that samples DUT signals but doesn't drive them. A monitor:

- Collects transactions (data items).
- Extracts events, performs checking and coverage.
- Optionally prints trace information.

2.3.7 Scoreboard:

It is a very crucial element of a self-checking environment . Typically, a scoreboard verifies whether there has been proper operation of your design at a functional level.

2.3.8 Environment:

The environment (env) is the top-level component of the OVC. The environment class (ovm_env) is architected to provide a flexible, reusable, and extendable verification component. The main function of the environment class is to model behaviour by generating constrained-random traffic, monitoring DUT responses, checking the validity of the protocol activity, and collecting coverage.

The build method creates instances of the driver and sequencer components using the factory, which performs polymorphic object creation. The arguments that are passed to the method create are: one, the local instance name being instantiated; and two, a reference to the parent component. The connect method is used to connect ports to exports.

2.3.9 Test:

The test configures the verification environment to apply a specific stimulus to the DUT.

2.3.10 TOP-Level Module:

A single top-level module couples the verification environment and the design-under-test together.

Chapter 3

I2C Bus

3.1 Introduction to I2C bus protocol
3.2 Functioning of I2C bus
3.2.1 I^2C Bus Configuration
3.2.2 I^2C Protocol
3.2.3 START/STOP
3.2.4 ADDRESSING
3.2.5 CLOCK STRETCHING
3.2.6 ARBITRATION
3.2.7 BUS CLEAR

3.1 INTRODUCTION TO I2C BUS PROTOCOL

A bus connects all the internal computer components to the CPU and Main memory. Every bus has a clock speed measured in MHz. A fast bus allows data to be transferred faster, which makes applications run faster. On PCs, the old ISA bus is being replaced by faster buses such as PCI. A System bus is used to transfer data from one location or device on the board to the central processing unit where all calculations take place.

Two different parts of a Computer Bus

- Address bus-transfers information about where the data should go
- Data bus-transfers the actual data

Bus Examples: SPI, I2C, PCI, PCI Xpress, AMBA,CAN, Wishbone, ISA …

SYSTEM BUS: this bus connects the CPU, memory and Cache.

- Address Bus
- Data Bus
- Control Bus

I/O BUS: Connecting to the above three buses is the "good old" standard I/O bus, used for slower peripherals (mice, modems, regular sound cards, low-speed networking) and also for compatibility with older devices. On almost all modern PCs this is the Industry Standard Architecture (ISA) bus.

INTER INTEGRATED CIRCUIT BUS:

Inter integrated circuits or I2C is one of the most widely used buses these days. I2C devices include EEPROMs, thermal sensors, and real-time clocks. Used as a control interface to signal processing devices which have separate data interfaces, e.g. RF tuners, video decoders and encoders, and audio processors.

I2C bus has three speeds:

- Slow (under 100 Kbps)
- Fast (400 Kbps)
- High-speed (3.4 Mbps) – I2C v.2.0

The distance is limited to about 10 feet for moderate speeds.[7]

3.2 FUNCTIONING OF I2C BUS:

3.2.1 I^2C BUS CONFIGURATION

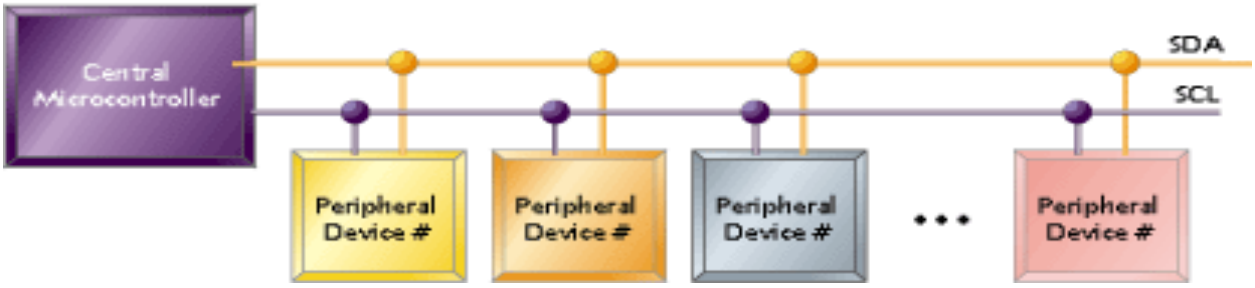


Figure 3.1 - General structure of I2C bus configuration

The basic structure of an I2C bus protocol is shown above in Figure 3.1. As can be seen there are 2 lines - Serial data (SDA) and Serial clock (SCL). All the required control signals and data are transferred with these two lines. Thus I2C bus is a half-duplex, synchronous, multi-master bus. No chip select required. The lines are pulled high via resistors, pulled down via open-drain drivers (wired-AND).

3.2.2 I^2C PROTOCOL

The basic procedure followed during the data transfer between the master and the slave in seven bit addressing mode.

- Master sends start condition (S) and controls the clock signal
- Master sends a unique 7-bit slave device address
- Master sends read/write bit (R/W) – 0 - slave receive, 1 - slave transmit
- Receiver sends acknowledge bit (ACK)
- Transmitter (slave or master) transmits 1 byte of data

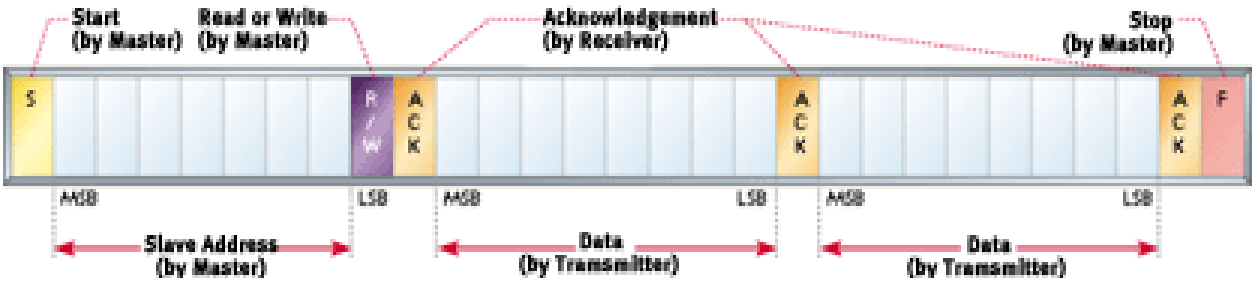


Figure 3.2 - Data frames

3.2.3 START/STOP Conditions, DATA TRANSFER

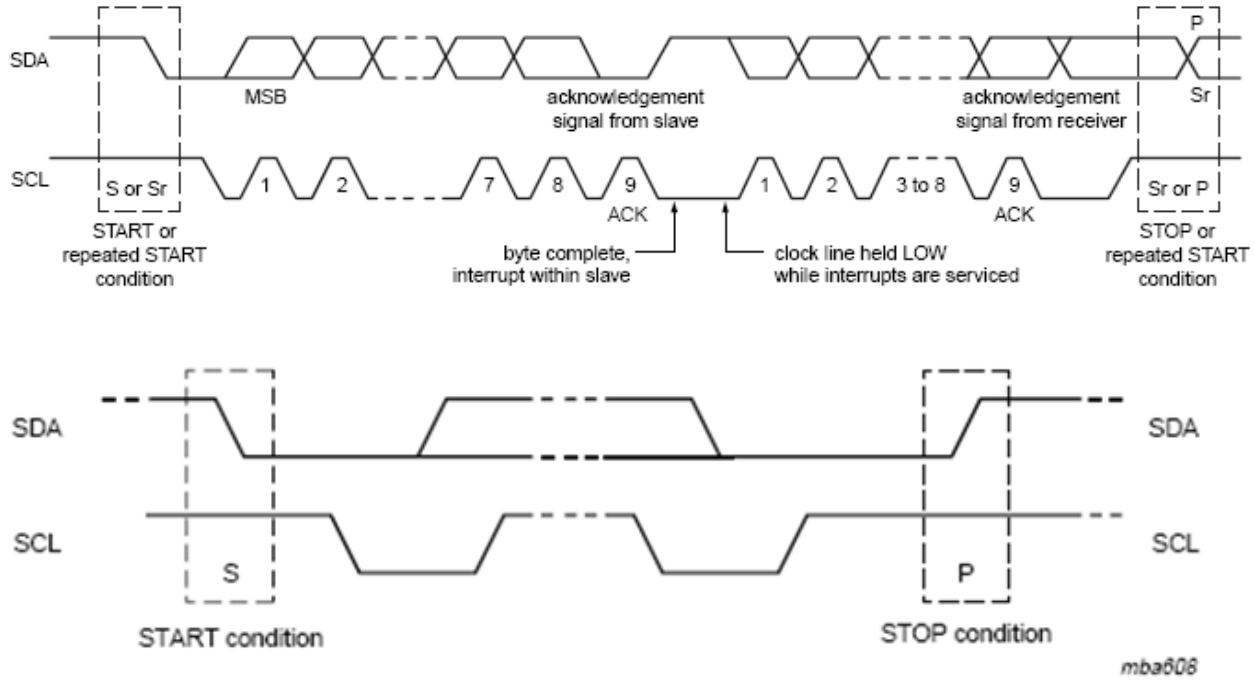


Figure 3.3 - Start, Stop and Data transfer

The various control signals in I2C bus protocol are defined as follows. They are also shown in Figure 3.3.

- Start – high-to-low transition of the SDA line while SCL line is high
- Stop – low-to-high transition of the SDA line while SCL line is high
- Ack – receiver pulls SDA low while transmitter allows it to float high
- Data – transition takes place while SCL is slow, valid while SCL is high

3.2.4: ADDRESSING

I2C bus supports 2 addressing modes. They are – 7 bit addressing and 10 bit addressing. The functions of the various bits in the two addressing modes are shown in figure below.

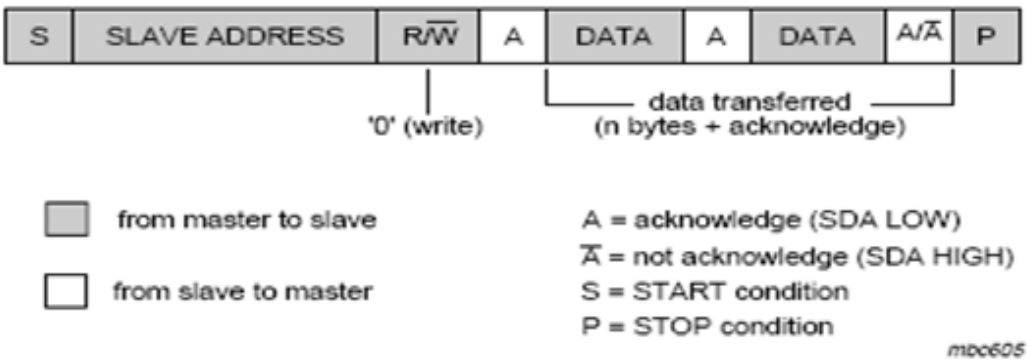


Figure 3.4 - 7 bit addressing

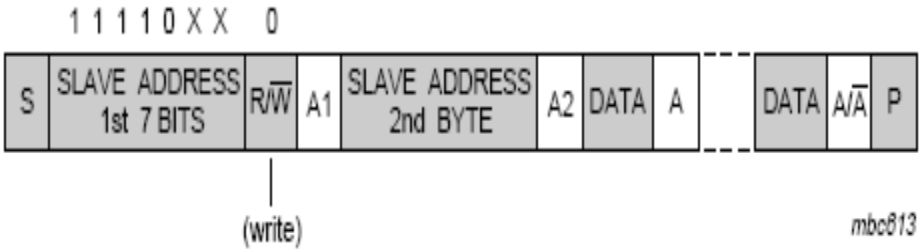


Figure 3.5 - 10 bit addressing

3.2.5 CLOCK STRETCHING

Clock stretching pauses a transaction by holding the SCL line LOW. The transaction cannot continue until the line is released HIGH again. Clock stretching is optional and in fact, most slave devices do not include an SCL driver so they are unable to stretch the clock.

A device may be able to receive bytes of data on a byte level at a fast rate, but may need more time to store a received byte or put in order another byte to be transmitted. Slaves can then hold the SCL line LOW after reception and acknowledgment of a byte to depower the master into a wait state until the slave is ready for the next byte transfer (15).

3.2.6 ARBITRATION

Arbitration is a part of the protocol required only if more than one master will be used in the system. Slaves are not involved in the arbitration procedure. A master may start a transfer only if the bus is free. Two masters may generate a START condition within the minimum hold time of the START condition which results in a valid START condition on the bus. Arbitration is then required to determine which master will complete its transmission.

Arbitration proceeds bit by bit. During every bit, while SCL is HIGH, each master checks to see if the SDA level matches what it has sent. This process may take many bits. Two masters can actually complete an entire transaction without error, as long as the transmissions are identical. The first time a master tries to send a HIGH, but detects that the SDA level is LOW, the master knows that it has lost the arbitration and will turn off its SDA output driver. The other master goes on to complete its transaction.

No information is lost during the arbitration process. A master that loses the arbitration can generate clock pulses until the end of the byte in which it loses the arbitration and must restart its transaction when the bus is idle.

If a master incorporates a slave function and loses arbitration during the addressing stage, it may be possible that the winning master is trying to address it. The losing master should hence switch over immediately to its slave mode.

The control of the I2C-bus is decided exclusively on the address and data sent by two or more competing masters, there is no central master, nor any order of priority given on the bus (15).

3.2.7 BUS CLEAR

In the unlikely event where the clock (SCL) is stuck LOW, the preferential procedure is to reset the bus using the HW reset signal if your I2C devices have HW reset inputs. If the I2C devices do not have HW reset inputs, cycle power to the devices to activate the mandatory internal Power-On Reset (POR) circuit.

If the data line (SDA) is LOW and stuck, the master should send 9 clock pulses. The device that holds the bus LOW should release in sometime within these 9 clocks. Else use the cycle power or hardware reset to clear the bus (15).

Chapter 4

Development of OVM Verification Environment for I2C

4.1 Specifications of the Design under Test:

The Design under Test is an I^2C master controller core. It produces SDA and SCL signals as per the configuration of its internal registers. The internal registers are configured using a WISHBONE interface (16).

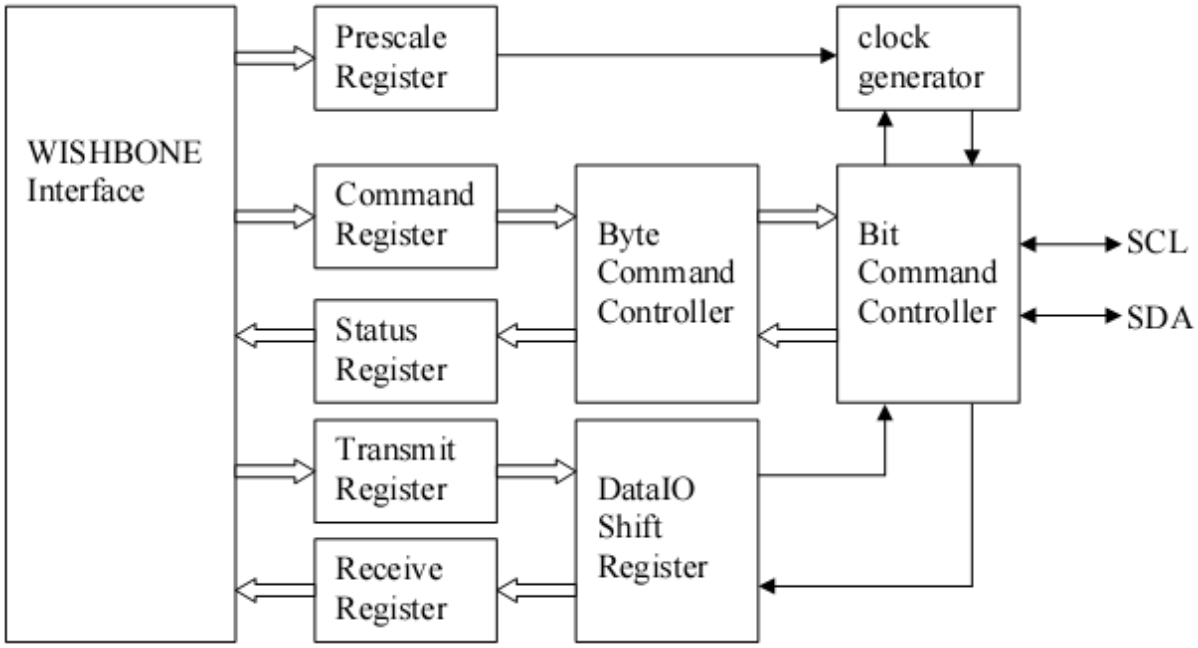


Figure 4.1 – Block diagram of the master controller core

IO Ports

The following table shows WISHBONE Interface signals to and from the I^2C master controller core:

Port	Width	Direction	Description
wb_clk_i	1	Input	Master clock
wb_rst_i	1	Input	Synchronous reset, active high
arst_i	1	Input	Asynchronous reset
wb_adr_i	3	Input	Lower address bits
wb_dat_i	8	Input	Data towards the core
wb_dat_o	8	Output	Data from the core
wb_we_i	1	Input	Write enable input
wb_stb_i	1	Input	Strobe signal/Core select input

wb_cyc_i	1	Input	Valid bus cycle input
wb_ack_o	1	Output	Bus cycle acknowledge output
wb_inta_o	1	Output	Interrupt signal output

These WISHBONE interface signals are WISHBONE RevB.3 compliant. Each access takes 2 clock cycles. Here access refers to the read/write operation to the core through the WISHBONE interface. arst_i is not a WISHBONE compatible signal.

The I^2C interface uses a serial data line (SDA) and a serial clock line (SCL) for data transfers. All devices connected to these two signals have open drain or open collector outputs. Both lines are pulled-up to VCC by external resistors.

The tri-state buffers for the SCL and SDA lines are added at a higher hierarchical level. Connections were made according to the following figure:

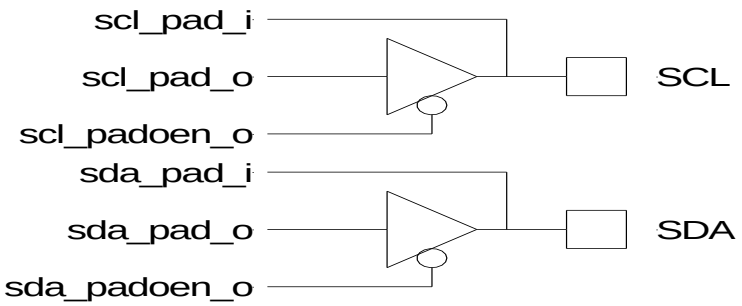


Figure 4.2 – SDA and SCL connections

The following table shows I^2C Interface signals to and from the I^2C master controller core:

Port	Width	Direction	Description
scl_pad_i	1	Input	Serial Clock line input
scl_pad_o	1	Output	Serial Clock line output
scl_pad_oe	1	Output	Serial Clock line output enable
sda_pad_i	1	Input	Serial Data line input
sda_pad_o	1	Output	Serial Data line output
sda_pad_oe	1	Output	Serial Data line output enable

Registers

The following table shows the list of registers in the I^2C master controller core:

Name	Address	Width	Access	Description
PRERlo	0x00	8	RW	Clock Prescale register lo-byte
PRERhi	0x01	8	RW	Clock Prescale register hi-byte
CTR	0x02	8	RW	Control register
TXR	0x03	8	W	Transmit register
RXR	0x03	8	R	Receive register
CR	0x04	8	W	Command register
SR	0x04	8	R	Status register

These registers need to be programmed according to the following description for the proper functioning of the master controller core.

Register Description

Prescale Register

This register is used as a prescalar to the SCL clock line. This core uses a 5*SCL clock internally. The prescale register needs to be programmed to 5*SCL frequency (minus 1). The value of the prescale register is changed only when the 'EN' bit is cleared.

Used wb_clk_i = 1.12GHz, desired SCL = 100KHz

prescale = $\frac{1.12GHz}{5\times100KHz} - 1$ = 0x08C6

Reset Value: 0xFFFF

Control Register

The following table shows different bits of the control register of the master controller core:

Bit #	Access	Description
7	RW	EN, I^2C core enable bit. When set to '1', the core is enabled.

		When set to ‘0’, the core is disabled.
6	RW	IEN, I^2C core interrupt enable bit. When set to ‘1’, interrupt is enabled. When set to ‘0’, interrupt is disabled.
5:0	RW	*Reserved*

Reset Value: 0x00

When the ‘EN’ bit is set, the core responds to new commands. Pending commands are finished first. The ‘EN’ bit is cleared only when no transfer is in progress, i.e. after a STOP command, or when the command register has the STO bit set. When halted during a transfer, the core can hang the I^2C bus.

Transmit register

Bit #	Access	Description
7:1	W	Next byte to transmit via I^2C
0	W	In case of a data transfer this bit represent the data’s LSB. In case of a slave address transfer this bit represents the RW bit. ‘1’ = reading from slave ‘0’ = writing to slave

Reset value: 0x00

Receive register

Bit #	Access	Description
7:0	R	Last byte received via I^2C

Reset value: 0x00

Command register

Bit #	Access	Description
7	W	STA, generate (repeated) start condition
6	W	STO, generate stop condition
5	W	RD, read from slave
4	W	WR, write to slave
3	W	ACK, when a receiver, sent ACK (ACK = '0') or NACK (ACK = '1')
2:1	W	*Reserved*
0	W	IACK, Interrupt acknowledge. When set, clears a pending interrupt.

Reset Value: 0x00

The STA, STO, RD, WR, and IACK bits are cleared automatically. These bits are always read as zeros.

Status register

Bit #	Access	Description
7	R	RxACK, Received acknowledge from slave. This flag represents acknowledge from the addressed slave. '1' = No acknowledge received '0' = Acknowledge received
6	R	Busy, I^2C bus busy '1' after START signal detected '0' after STOP signal detected
5	R	AL, Arbitration lost This bit is set when the core lost arbitration. Arbitration is lost when: • a STOP signal is detected, but non requested • The master drives SDA high, but SDA is low. See *bus-arbitration* section for more information.
4:2	R	*Reserved*

1	R	TIP, Transfer in progress. '1' when transferring data '0' when transfer complete
0	R	IF, Interrupt Flag. This bit is set when an interrupt is pending, which will cause a processor interrupt request if the IEN bit is set. The Interrupt Flag is set when: • one byte transfer has been completed • arbitration is lost

Reset Value: 0x00

*Please note that all **reserved bits** are read as zeros.*

4.2 Verification Plan:

The Verification Plan defines exactly what needs to be tested, and drives the coverage criteria. The completeness of a verification plan and its accurate implementation lead to success of the verification project in hand. Detailed goals using measurable metrics, along with optimal resource usage and realistic schedule estimates are the contents of a good plan (17).

Feature extraction, Stimulus generation plan, Checker plan and Coverage plan are the important parts of a verification plan.

4.2.1 Feature Extraction

It contains the list of all the features to be verified. For the present DUT, it is the following:

• Generation of START condition

• Generation of STOP condition

• Clock Stretching

• Clock Synchronization and Arbitration

• Addressing modes 7 - bit/10 – bit

• All possible Master – Slave data transfer formats

• Generation of ACK and NACK

4.2.2 Stimulus Generation Plan

It contains information about different types of transactions, sequences of transactions and various scenarios generated as per the specification.

- •Generate slave address randomly excluding reserved address
- •Generate random data sequence of variable length
- •Generate random ACK/NACKs to check Transmitter's response
- •Generate random delay to insert wait states and check Transmitter's response
- •Instantiate multi – masters with different clock speeds to check for synchronization and arbitration

Checker Plan is for checking expected results, implemented by monitors and scoreboards based on the protocol.

Coverage Plan explains the functional coverage of the features. A functional coverage plan should be built to help implement coverage points in the verification environment.

4.2.3 Verification Environment

It is the implementation of the verification plan to verify the DUT. It consists of various sub parts which when put together act as the verification environment to the DUT.

4.3 Development of individual OVCs

INTERFACE

It contains all the pin level connections to the DUT to and from the verification environment. Here, all the WISHBONE signals as well as all the I^2C signals are mentioned along with their correct data types. A modport is defined showing connections with respect to the verification environment.

TRANSACTION

It is required for transaction level modeling. Transactions are passed along different blocks of the verification environment as a bundled unit except for pin level connections to and from the DUT. It enables easy recording of information generated in the environment.

Usually all the signals or values that need to be generated randomly are contained in a transaction. It may also contain values which are not generated randomly but need to be

recorded always. It also contains the constraints within which each value needs to be generated.

The transaction used here contains the following:

7/10 I^2C slave address which needs to be generated randomly.

Direction of transfer initiated by the master is also generated randomly.

8 bit data is always generated randomly.

Prescale register values need to be generated randomly to choose between different speed modes supported by the core.

Control register values also need to be randomly generated to enable or disable the core.

Since registers of the core are being written and read frequently by the environment, core register addresses are included in the transaction to enable easy recording.

SEQUENCER

It needs to be defined by connecting it to the interface and enabling automatic sequence library updating.

SEQUENCE

It lies at the core of a verification environment. Any number of sequences can be defined to test each different operation of the DUT.

`ovm_do_with is used to randomize a transaction with inline constraints for specific functionality of the DUT to be checked.

DRIVER

It contains the underlying logic to drive the pins of the DUT according to transactions provided to it from the sequencer. The write operations stimulate the pins of the DUT to read particular value at the output pins of the DUT using the output monitor. The read operations are meant to record response from the DUT as and when generated.

MASTER MONITOR

It is used to monitor the signals sent to the pins of the DUT from the driver which in turn it receives the transaction (stimuli sequence) from the sequencer. It collects the stimulus and their corresponding response for each read/write access of the core using the WISHBONE interface.

It also gives coverage values according to the covergroups implemented in it.

BUS MONITOR

It is used to keep a track of the changes occurring in the SDA and the SCL lines. It verifies the functional correctness of the bus protocol by keeping a record of every bit sent/received by the master and corresponding bit in the line which is further used for calculating coverage.

ENVIRONMENT

It creates instances of all the above mentioned OVCs and links them with each other for the functioning of the verification environment. Default sequence is registered with the OVM factory. ovm_random_sequence is the default sequence to be used by the sequencer. set_config_string() is used to register a user defined sequence as a default sequence.

All the OVC instances are registered with the OVM factory for printing using the type_id::create function in the build function.

TEST

It creates an instance of the environment to invoke the environment. The environment is also registered for printing using the type_id::create function.

The environment is connected to the interface using the interface instance. Also the ovm_printer is invoked here.

TOP

This module connects the DUT with the verification environment through the interface instance. Global clock pulses are created here. run_test is used to run the verification process. global_stop_request is used to stop the verification process after a specified period of time or number of iterations or after a threshold value of coverage.

Chapter 5

Results

FOLLOWING IS THE SIMULATION OUTPUT:

```
[manoranjan@node5 ~]$ qverilog /home/NIS/BTECH_06-
10/manoranjan/Desktop/ibus_new.sv +define+CVC -R +OVM_TESTNAME=ibus_demo_tb
QuestaSim-64 qverilog 6.5b Compiler 2009.05 May 21 2009
/cad/Mentor2009/modeltech/bin/../linux_x86_64/qverilog /home/NIS/BTECH_06-
10/manoranjan/Desktop/ibus_new.sv +define -R +OVM_TESTNAME=ibus_demo_tb
-- Compiling interface ibus_if
-- Compiling package ibus_new_sv_unit
-- Compiling module top
** Warning: /home/NIS/BTECH_06-10/manoranjan/Desktop/ibus_new.sv(207):
(qverilog-2223) In-line constraints for hierarchical call to
class::randomize() will be resolved with respect to the current scope
** Warning: /home/NIS/BTECH_06-10/manoranjan/Desktop/ibus_new.sv(244):
(qverilog-2223) In-line constraints for hierarchical call to
class::randomize() will be resolved with respect to the current scope
-- Compiling module i2c_master_top
-- Compiling module i2c_master_byte_ctrl
-- Compiling module i2c_master_bit_ctrl
-- Compiling module i2c_slave_model

Top level modules:
        top
+ /cad/Mentor2009/questasim/v6.5b/linux_x86_64/vsim -lib work
+OVM_TESTNAME=ibus_demo_tb top -c -do run -all; quit -f -appendlog -l
qverilog.log -vopt
Reading /cad/Mentor2009/questasim/v6.5b/tcl/vsim/pref.tcl

# 6.5b

# vsim +OVM_TESTNAME=ibus_demo_tb -appendlog -do {run -all; quit -f} -l
qverilog.log -lib work -c -vopt top
# ** Note: (vsim-3812) Design is being optimized...
# ** Warning: /home/NIS/BTECH_06-10/manoranjan/Desktop/ibus_new.sv(207):
(vopt-2223) In-line constraints for hierarchical call to class::randomize()
will be resolved with respect to the current scope
# ** Warning: /home/NIS/BTECH_06-10/manoranjan/Desktop/ibus_new.sv(244):
(vopt-2223) In-line constraints for hierarchical call to class::randomize()
will be resolved with respect to the current scope
# //  QuestaSim-64 6.5b May 21 2009 Linux 2.6.9-55.0.2.ELsmp
# //
# //  Copyright 1991-2009 Mentor Graphics Corporation
# //             All Rights Reserved.
# //
# //  THIS WORK CONTAINS TRADE SECRET AND
# //  PROPRIETARY INFORMATION WHICH IS THE PROPERTY
# //  OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
# //  AND IS SUBJECT TO LICENSE TERMS.
# //
# Loading sv_std.std
# Loading work.ibus_new_sv_unit(fast)
# Loading work.top(fast)
# Loading work.ibus_if(fast)
# Loading work.i2c_master_top(fast)
# Loading work.i2c_master_byte_ctrl(fast)
# Loading work.i2c_master_bit_ctrl(fast)
# Loading work.i2c_slave_model(fast)
# run -all
# ----------------------------------------------------------------
# OVM-2.0.2
# (C) 2007-2009 Mentor Graphics Corporation
```

```
# (C) 2007-2009 Cadence Design Systems, Inc.
# ----------------------------------------------------------------
# OVM_INFO @ 0: reporter [RNTST] Running test ibus_demo_tb...
# OVM_INFO @ 0: ovm_test_top [ovm_test_top] START of build test
# OVM_INFO @ 0: ovm_test_top [ovm_test_top] END of build test
# OVM_INFO @ 0: ovm_test_top.ibus0 [ovm_test_top.ibus0] START of build env
# ------------------------------------------------------------------------
# Name                      Type                Size                 Value
# ------------------------------------------------------------------------
# sequencer                 ibus_master_sequen+ -              sequencer@11
#   rsp_export              ovm_analysis_export -            rsp_export@13
#     recording_detail      ovm_verbosity       32                OVM_FULL
#   seq_item_export         ovm_seq_item_pull_+ -       seq_item_export@37
#     recording_detail      ovm_verbosity       32                OVM_FULL
#   recording_detail        ovm_verbosity       32                OVM_FULL
#   default_sequence        string              14          write_byte_seq
#   count                   integral            32                      -1
#   max_random_count        integral            32                    'd10
#   sequences               array               5                       -
#     [0]                   string              19     ovm_random_sequence
#     [1]                   string              23    ovm_exhaustive_sequ+
#     [2]                   string              19     ovm_simple_sequence
#     [3]                   string              13           read_byte_seq
#     [4]                   string              14          write_byte_seq
#   max_random_depth        integral            32                    'd4
#   num_last_reqs           integral            32                    'd1
#   num_last_rsps           integral            32                    'd1
# ------------------------------------------------------------------------
# ------------------------------------------------------------------------
# Name                      Type                Size                 Value
# ------------------------------------------------------------------------
# driver                    ibus_master_driver  -                driver@41
#   rsp_port                ovm_analysis_port   -              rsp_port@45
#     recording_detail      ovm_verbosity       32                OVM_FULL
#   sqr_pull_port           ovm_seq_item_pull_+ -         sqr_pull_port@43
#     recording_detail      ovm_verbosity       32                OVM_FULL
#   recording_detail        ovm_verbosity       32                OVM_FULL
# ------------------------------------------------------------------------
# OVM_INFO @ 0: ovm_test_top.ibus0 [ovm_test_top.ibus0] END of build env
# ------------------------------------------------------------------------
# Name                      Type                Size                 Value
# ------------------------------------------------------------------------
# ovm_test_top              ibus_demo_tb        -           ovm_test_top@1
#   ibus0                   ibus_env            -                   ibus0@3
#     bus_monitor           ibus_bus_monitor    -            bus_monitor@5
#       item_collected_po+ ovm_analysis_port    -    item_collected_por+
#       coverage_enable     integral            1                     'h1
#       num_transactions    integral            32                    'h0
#       recording_detail    ovm_verbosity       32                OVM_FULL
#     driver                ibus_master_driver  -                driver@41
#       rsp_port            ovm_analysis_port   -              rsp_port@45
#       sqr_pull_port       ovm_seq_item_pull_+ -         sqr_pull_port@43
#       recording_detail    ovm_verbosity       32                OVM_FULL
#     monitor               ibus_master_monitor -               monitor@47
#       item_collected_po+ ovm_analysis_port    -    item_collected_por+
#       coverage_enable     integral            1                     'h1
#       recording_detail    ovm_verbosity       32                OVM_FULL
#     sequencer             ibus_master_sequen+ -              sequencer@11
#       rsp_export          ovm_analysis_export -            rsp_export@13
#       seq_item_export     ovm_seq_item_pull_+ -       seq_item_export@37
#       recording_detail    ovm_verbosity       32                OVM_FULL
```

```
#       default_sequence   string              14         write_byte_seq
#       count              integral            32                     -1
#       max_random_count   integral            32                   'd10
#       sequences          array               5                      -
#         [0]              string              19    ovm_random_sequence
#         [1]              string              23   ovm_exhaustive_sequ+
#         [2]              string              19    ovm_simple_sequence
#         [3]              string              13          read_byte_seq
#         [4]              string              14         write_byte_seq
#       max_random_depth   integral            32                   'd4
#       num_last_reqs      integral            32                   'd1
#       num_last_rsps      integral            32                   'd1
#     has_bus_monitor      integral            1                    'h1
#     recording_detail     ovm_verbosity       32               OVM_FULL
# ---------------------------------------------------------------------
# OVM_INFO @ 0: ovm_test_top.ibus0 [] Called ibus_env::run
# OVM_INFO @ 100: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 100: ovm_test_top.ibus0.driver [] Got Transaction prelo= `x30,
prerhi = `x2
# OVM_INFO @ 200: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 200: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x18
# OVM_INFO @ 300: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x18
# OVM_INFO @ 400: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 500: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x1
# OVM_INFO @ 600: ovm_test_top.ibus0.driver [] Got Transaction ctr= `x8
# OVM_INFO @ 700: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 800: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x4
# OVM_INFO @ 900: ovm_test_top.ibus0.driver [] Got Transaction addr= `x4e,
data = `xf5
# OVM_INFO @ 1100: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 1300: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x48
# OVM_INFO @ 1600: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 1900: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x7a
# OVM_INFO @ 2100: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x8
# OVM_INFO @ 2400: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 2700: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x28
# OVM_INFO @ 2900: ovm_test_top.ibus0.driver [] Got Transaction prelo=
`xc6, prerhi = `x8
# OVM_INFO @ 2900: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 3200: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x63
# OVM_INFO @ 3300: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x63
# OVM_INFO @ 3700: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x4
# OVM_INFO @ 3800: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x4
# OVM_INFO @ 3800: ovm_test_top.ibus0.driver [] Got Transaction ctr= `x8
# OVM_INFO @ 4100: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x4
# OVM_INFO @ 4200: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x4
# OVM_INFO @ 4200: ovm_test_top.ibus0.driver [] Got Transaction addr= `x4e,
data = `xf5
# OVM_INFO @ 4600: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 4700: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 5000: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x48
# OVM_INFO @ 5100: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x48
# OVM_INFO @ 5500: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 5600: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 5900: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x7a
# OVM_INFO @ 6000: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x7a
# OVM_INFO @ 6400: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x8
# OVM_INFO @ 6500: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x8
# OVM_INFO @ 6800: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 6900: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 7300: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x28
```

```
# OVM_INFO @ 7400: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x28
# OVM_INFO @ 7400: ovm_test_top.ibus0.driver [] Got Transaction prelo=
`xc6, prerhi = `x8
# OVM_INFO @ 7800: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x63
# OVM_INFO @ 7900: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x63
# OVM_INFO @ 8300: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x4
# OVM_INFO @ 8300: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x4
# OVM_INFO @ 8400: ovm_test_top.ibus0.driver [] Got Transaction ctr= `x8
# OVM_INFO @ 8700: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x4
# OVM_INFO @ 8800: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x4
# OVM_INFO @ 8800: ovm_test_top.ibus0.driver [] Got Transaction addr= `x4e,
data = `xf5
# OVM_INFO @ 9200: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 9200: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 9600: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x48
# OVM_INFO @ 9700: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x48
# OVM_INFO @ 10100: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 10100: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 10500: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x7a
# OVM_INFO @ 10600: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x7a
# OVM_INFO @ 11000: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x8
# OVM_INFO @ 11000: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x8
# OVM_INFO @ 11400: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 11500: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 11900: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x28
# OVM_INFO @ 11900: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x28
# OVM_INFO @ 12000: ovm_test_top.ibus0.driver [] Got Transaction prelo=
`xc6, prerhi = `x8
# OVM_INFO @ 12400: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x63
# OVM_INFO @ 12500: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x63
# OVM_INFO @ 12800: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x4
# OVM_INFO @ 12900: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x4
# OVM_INFO @ 12900: ovm_test_top.ibus0.driver [] Got Transaction ctr= `x8
# OVM_INFO @ 13300: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x4
# OVM_INFO @ 13400: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x4
# OVM_INFO @ 13400: ovm_test_top.ibus0.driver [] Got Transaction addr=
`x4e, data = `xf5
# OVM_INFO @ 13700: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 13800: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 14200: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x48
# OVM_INFO @ 14300: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x48
# OVM_INFO @ 14600: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 14700: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 15100: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x7a
# OVM_INFO @ 15200: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x7a
# OVM_INFO @ 15500: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x8
# OVM_INFO @ 15600: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x8
# OVM_INFO @ 16000: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 16100: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 16400: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x28
```

```
# OVM_INFO @ 16500: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x28
# OVM_INFO @ 16600: ovm_test_top.ibus0.driver [] Got Transaction prelo=
`xc6, prerhi = `x8
# OVM_INFO @ 17000: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x63
# OVM_INFO @ 17100: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x63
# OVM_INFO @ 17400: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x4
# OVM_INFO @ 17500: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x4
# OVM_INFO @ 17500: ovm_test_top.ibus0.driver [] Got Transaction ctr= `x8
# OVM_INFO @ 17900: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x4
# OVM_INFO @ 18000: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x4
# OVM_INFO @ 18000: ovm_test_top.ibus0.driver [] Got Transaction addr=
`x4e, data = `xf5
# OVM_INFO @ 18300: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 18400: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 18800: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x48
# OVM_INFO @ 18900: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x48
# OVM_INFO @ 19200: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 19300: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 19700: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x7a
# OVM_INFO @ 19800: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x7a
# OVM_INFO @ 20100: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x8
# OVM_INFO @ 20200: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x8
# OVM_INFO @ 20600: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 20700: ovm_test_top.ibus0.monitor [] Got Transaction addr= `x0
# OVM_INFO @ 21000: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x28
# OVM_INFO @ 21100: ovm_test_top.ibus0.monitor [] Got Transaction addr=
`x28
#
# --- OVM Report Summary ---
#
# ** Report counts by severity
# OVM_INFO :  117
# OVM_WARNING :    0
# OVM_ERROR :    0
# OVM_FATAL :    0
# ** Report counts by id
# []   112
# [RNTST]     1
# [ovm_test_top]     2
# [ovm_test_top.ibus0]     2
# ** Note: $finish    : /home/NIS/BTECH_06-
10/manoranjan/ovm/src/base/ovm_root.svh(488)
#    Time: 1 ms  Iteration: 8  Instance:
:ibus_new_sv_unit::ovm_root::run_test
[manoranjan@node5 ~]$
```

Initially all the connections of the OVCs are shown in the tabular form using the ovm_printer function.

One can see the monitor keeping track of every input to the controller from the driver as well as the sequences received by the driver.

Chapter 6

Conclusion

CONCLUSION

Development of a Verification IP for any design (DUV) becomes very simple with the use of OVM. I^2C Verification IP has been designed for I^2C master/slave with Wishbone Interface (16). It can be easily extended for any kind of feasible I^2C interface.

Chapter 7

References

REFERENCES

1. Embedded system - Wikipedia, the free encyclopedia. [Online] http://en.wikipedia.org/wiki/Embedded_system.

2. **manual, UM10204 I2C-bus specification and user.** [Online] http://www.nxp.com/documents/user_manual/UM10204.pdf.

3. **Fitzpatrick, Tom.** [Online] www.mentor.com.

4. Functional Verification Need. [Online] http://www.testbench.in/TS_03_FUNCTIONAL_VERIFICATION_NEED.html.

5. **Bergeron, Janick.** *Writing testbenches: functional verification of HDL models.* s.l. : Springer, 2003.

6. **Spear, Chris.** *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features.* s.l. : Springer, 2006.

7. **Lin, David.** Verification IP for IP verification. [Online] http://www.design-reuse.com/articles/8243/verification-ip-for-ip-verification.html.

8. **Bergeron, Janick, et al.** *Verification Methodology Manual for SystemVerilog.* s.l. : Springer, 2005.

9. [Online] http://www.mentor.com/products/fv/methodologies/abv/.

10. [Online] http://www.mentor.com/products/fv/methodologies/_3b715c/.

11. [Online] http://www.mentor.com/products/fv/methodologies/pdv.

12. OVM White Papers. [Online] http://www.ovmworld.org/white_papers.php.

13. OVM Datasheets. [Online] http://www.ovmworld.org/datasheets.php.

14. Getting Started with OVM - Tutorial 1 - A First Example. [Online] http://www.doulos.com/knowhow/sysverilog/ovm/tutorial_1/.

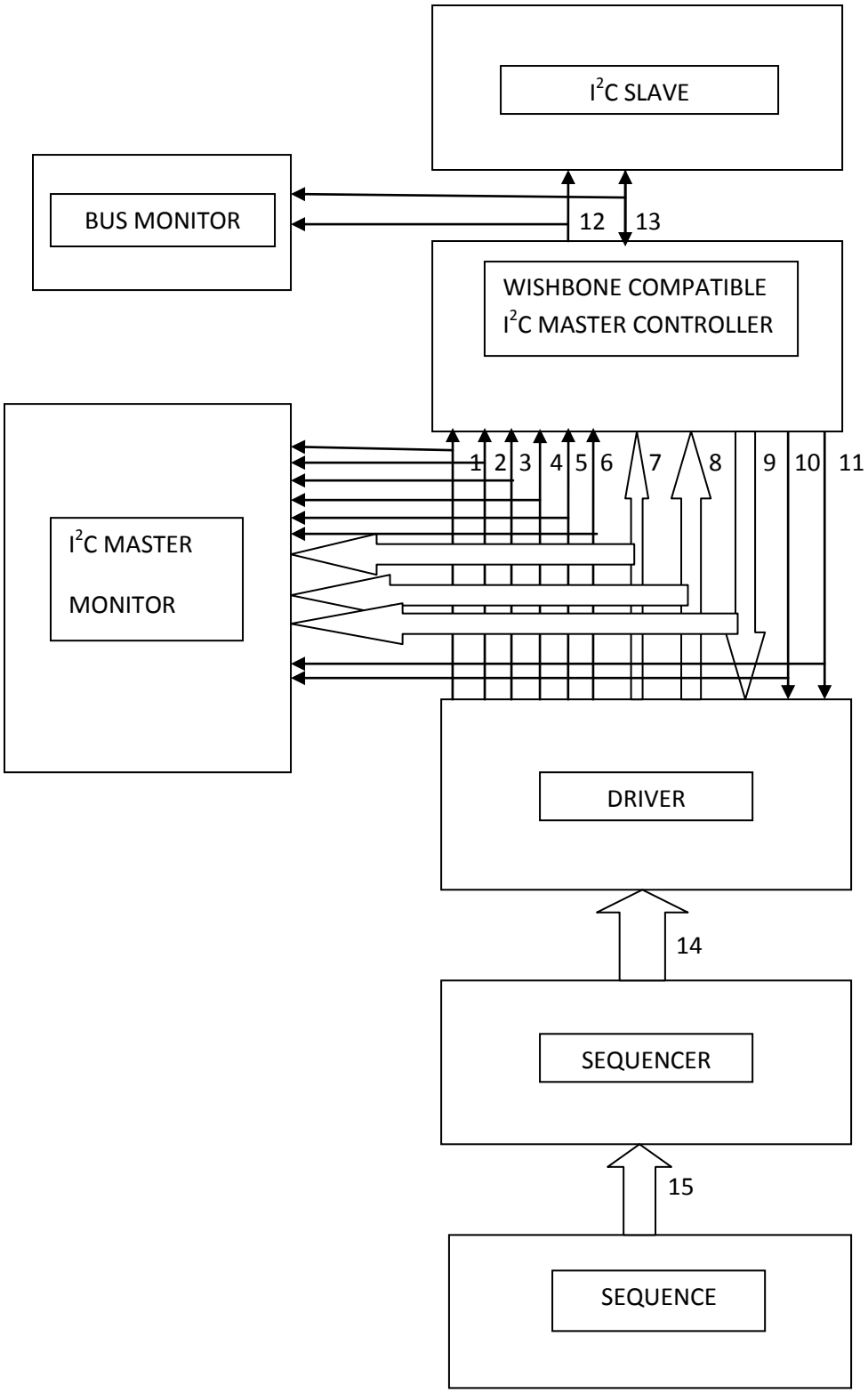
15. Application Note. [Online] http://www.nxp.com/documents/application_note/AN10216.pdf.

16. **Herveille, Richard.** [Online] www.opencores.org.

17. Verification Plan. [Online] www.testbench.in.

APPENDIX

Verification Environment for aWISHBONE compatible I^2C Master Controller Core



INTERNAL SIGNALS GENERATED

1. wb_clk_i
2. wb_rst_i
3. arst_i
4. wb_stb_i
5. wb_we_i
6. wb_cyc_i
7. wb_adr_i
9. wb_dat_o
10. wb_ack_o
11. wb_inta_o
12. scl
13. sda
14. Transaction (sequencer to driver)
15. Transaction (sequence to sequencer)