5-2019

UVM Verification of an I2C Master Core

Shravani Balaraju
sxb5692@rit.edu

UVM verification of an I2C Master Core

by

Shravani Balaraju

Graduate Paper

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in Electrical Engineering

Approved by:

Mr. Mark A. Indovina, Lecturer
Graduate Research Advisor, Department of Electrical and Microelectronic Engineering

---

Dr. Sohail A. Dianat, Professor
Department Head, Department of Electrical and Microelectronic Engineering

Department of Electrical and Microelectronic Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York

May, 2019

I dedicate this work to my mother Gouri Manthena, my father Dhananjaya Raju, my grandmother Siromani Manthena, my grandfather Narasa Raju, my partner Karthik Pythireddi, my best friend Veenadhari Polkam and my colleagues for their love and support during my thesis.

Declaration

I hereby declare that except where specific reference is made to the work of others, that all content of this Graduate Paper are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This Graduate Project is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Shravani Balaraju

May, 2019

Acknowledgements

I would like to thank my advisor Professor Mark A. Indovina for his support, guidance, feedback, and encouragement which helped in the successful completion of my graduate research.

Abstract

With the increasing complexity of IP designs, verification has become quite popular yet is still a significant challenge for a verification engineer. A proper verification environment can bring out bugs that one may never expect in the design. On the contrary, a poorly designed verification environment could give false information about the functioning of the design and bugs may appear on the consumer's end. Hence, the verification industry is continually looking for more efficient verification methodologies. This paper describes one such efficient methodology implemented on an Inter-Integrated Circuit (I2C) system. I2C packs in itself the powerful features of the Serial Peripheral Interface (SPI) and the universal asynchronous receiver-transmitter (UART), but is comparatively more efficient and uses less hardware for implementation. Also, it can establish secure communication between multiple masters and multiple slaves with minimal wiring. In this project, from a design perspective, the master is a hardware block, and the slave is a verification IP. The methodology used for verification is based on the Universal Verification Methodology (UVM), a class library written in the SystemVerilog language. The paper describes how the verification of an I2C system uses the powerful tools of UVM. The master core has been successfully verified and the coverage goals are met. The effort has been documented in this paper in detail.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Back when fabrication of devices with dimensions in microns was a wonder, designs were not as intricate, and the prime focus was on design more than verification. However, now, with rapid advancement in technology scaling, verification has become more of a challenge. As the designs became smaller, more space became available on the chip, and it gave the designer a chance to add new features and capabilities to the design. As a result, many sensors were built right onto the chip instead of connecting it with external sensors.

With these new possibilities in technology and more complex designs, a straightforward verification plan of toggling a few pins and observing the output no longer works. Each design has so many pins connected to it that the connections alone take up multiple lines of code and it is a misuse of time and intelligence to keep all the verification related code in one file and navigate through it. A layered test-bench helps to maintain modular code such that all the wiring is in one file, the input stimulus in one and so forth. This way, when a verification engineer needs to add new features to the verification environment, navigation through all the files is not needed, and change in the code of one file causes little/no change in other files.

Thus it helps in maintainability of the code.

This paper discusses one such efficient and re-usable verification environment applied to an Inter-Integrated Circuit (I2C) system with a Master as hardware and slave as a verification IP. The environment uses the useful features of Universal Verification Methodology (UVM) such as sequences, transaction level modeling, object-oriented programming, advanced data types, and functional coverage to verify the system thoroughly. The sections below describe the motivation, research goals and the organization of this paper.

1.1 Research Goals

The primary intent of setting up this verification environment is to research and implement a self-checking, constrained random, layered test-bench using the UVM framework and to observe its effectiveness. Shown below is a summary of the leading research goals:

- To understand the I2C protocol and integrate the master core with multiple slave VIPs to validate it.
- To observe the effectiveness of a self-checking test-bench and a random sequence generator through functional coverage and assertions.
- To come up with an effective verification methodology that is applicable not only to I2C, but could also easily be applied to other communication protocols. More on this is discussed in the following chapters.

1.2 Contributions

The significant contributions to the projected are listed below.:

1. For I2C master hardware, an accompanying I2C slave is modeled as a Verification IP in UVM.

2. The master core was integrated with multiple slaves with each having a unique address.

3. The master-slave is verified using a self-checking test-bench written in UVM using several standard UVM components like the driver, monitor, sequencer, scoreboard, and so forth.

4. Wherever required, the repetitive code is generated using scripting.

5. The effectiveness of the random sequences and the test-plan is measured using functional coverage metrics such as cover-groups and assertions.

6. The obtained information is analyzed and is presented using graphs and charts.

1.3 Organization

The structure of the thesis is as follows:

- Chapter 2: This chapter discusses about the UVM methodology using references to journals/articles wherever required. It also discusses how it helped verify large, complex systems in several projects.

- Chapter 3: This chapter explains about the typical verification components in a UVM framework, their importance and hierarchy.

- Chapter 4: The chapter goes deeper into the I2C protocol with a brief history, details on the functioning of an I2C system.

- Chapter 5: This chapter explains the proposed verification architecture.

- Chapter 6: This chapter goes over the test plan and implementation details of verification components.
- Chapter 7: This chapter discusses about the obtained results in detail and the drawn observations from the recorded results.
- Chapter 8: This chapter outlines the conclusion of the study and possible ways of extending it.

Chapter 2

Bibliographical Research

2.1 A brief history of Verification

Functional verification is given the most time and effort starting from the design's architecture planning to its tape out. Since it has such enormous importance in the design cycle, it became essential to empower the verification engineers with more advanced verification strategies, and thus SystemVerilog was developed. However, it is not just SystemVerilog or UVM that the verification engineers use. Below is the list of resources used in the Design verification phase of designs in the Industry[1]:

- OVM (Open Verification Methodology) / Verification Methodology Manual (VMM) / UVM
- UPF (Unified Power Format) verification
- AMS (Analog/Mixed Signal) verification
- SVA (SystemVerilog Assertions)
- CDV (Coverage-driven Verification)

- Static Formal Verification
- LEC (Logic Equivalency Check)
- Electronic system level (ESL) - A virtual platform for developing reference models in software
- Hardware-Software (HW-SW) co-verification
- Emulation

With technology scaling, the design complexity is rapidly growing, and the pressure falls very much on the verification engineer to analyze and ensure that the design performs its functions well and strictly adheres to the design specification. It is crucial to extensively validate a design Pre-Silicon so that the Post-Silicon succeeds in the first pass.

2.2 Design and Verification Challenges

The biggest challenge in the Industry is the short time between the design phase and the delivery of a working silicon chip to the customer. Around 40-50% of the total project resources are utilized in verifying the design. This fact alone is enough to give a numerical analysis of the impact of verification. At the same time, even with technology scaling, the designs and their complexities are improving drastically thus making it harder to verify them in such a short time under stringent constraints.

A few years ago, more than 50% of the chips needed a re-spin due to functional bugs. Even though these designs are tested before taping out, a majority of them were not thoroughly tested. It became essential to know thus when the verification is complete, and this gave rise to the concept of coverage in SystemVerilog.

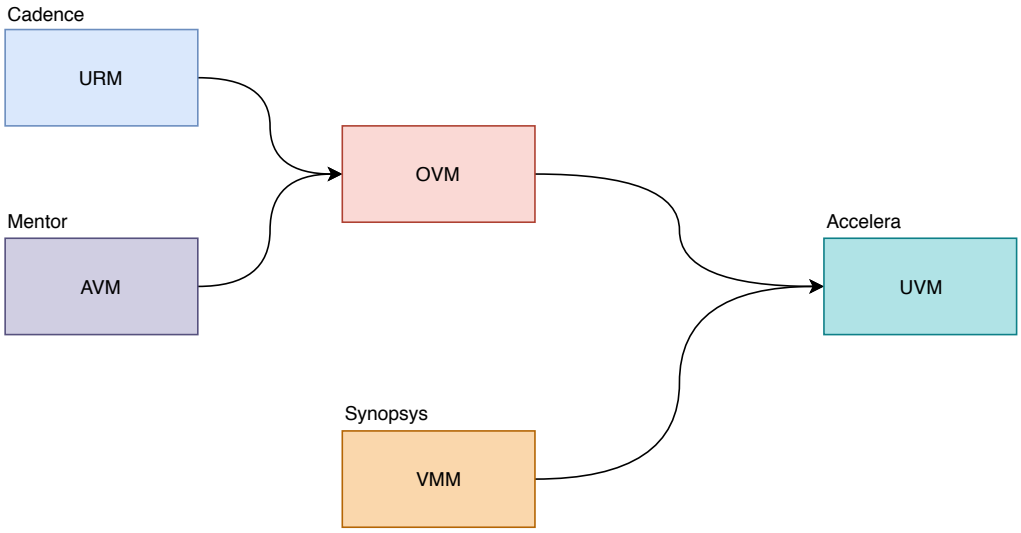


Figure 2.1: Evolution of Verification Methodologies

2.3 Verification Methodologies

Verification methodologies were introduced to ease the verification effort for large designs by developing modular and reusable environments[5]. OVM was first released in the year 2008 and is an enhanced library adapted from URM (Universal Reuse Methodology) and AVM (Advanced Verification Methodology)[6]. Based on the OVM library and addition of helpful features from VMM, a new library to support the growing verification needs, was developed by Accelera and came to be known as the Universal Verification Methodology(UVM). The Fig.2.1 summarizes the evolution of UVM precisely.

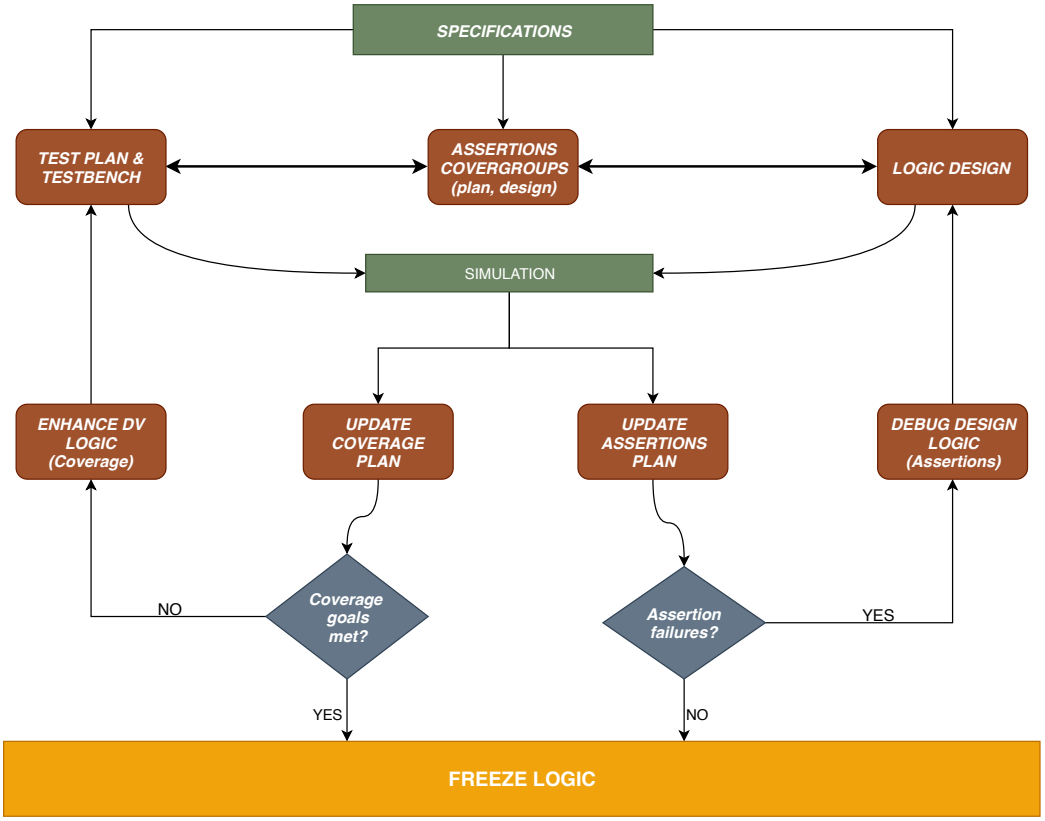


Figure 2.2: Coverage and Assertion driver verification methodology[1]

2.4 Functional Coverage and Assertions

2.4.1 Different coverage metrics

2.4.1.1 Code Coverage

Code coverage is a measure of the extent to which the design code has been exercised using the test bench. However, it does not provide information on whether the design intent has been exercised or not. Code coverage includes the coverage on expressions, toggle of pins and finite state machines. This coverage is tool specific and is generated automatically alongside functional coverage[7].

2.4.1.2 Functional Coverage

Functional coverage provides an analysis of whether the design follows its functional specification and if there are any deviations from the specification. Assertions also could be used to contribute toward functional coverage metrics. Similar to asserting them, the 'cover' keyword can be used for this purpose[3, 8].

2.4.2 Assertions

SVA is one of the most crucial components of the language and finds extensive use in design verification. Its use has become so widespread that many small-medium sized IPs are being verified using this concept alone. It is a language in itself, and it helps to check the corner cases in the design with minimal logic.

To start with, an assertion is simply a check in the design specification that should be true in all conditions. If the check does not hold even once, the assertion fails and reports the same to

the validator. Though there are numerous advantages of using SVA over standard SystemVerilog checks, the biggest of all is that SVA is very user friendly and is readable[9][10]. The methods used in SVA have names which are very true to their function thus making the code self-explanatory. Assertions can also be made part of the design as they are synthesizable. Following are a few examples of the type of assertions that could be used in the design and verification environment.

- RTL assertions to check the intent of the design.
- Block interface assertions for protocol verification and to check the presence of any illegal combinations of ports.
- Chip functionality assertions to check the intent of the block at SoC level.
- Performance implication assertions to check the performance deliverables[1].

2.4.2.1 Immediate Assertions

Immediate assertions are non-temporal and are executed just like statements in a procedural block. They could be compared to an if-else statement but are written with fewer lines of code[11].

2.4.2.2 Concurrent assertions

Concurrent assertions are temporal and are generally used for sophisticated checks that span over multiple clock cycles[12]. They are written using combinations of boolean expressions, sequences, and properties. The 'sequence' of a concurrent assertion is very different from the sequence of UVM test bench, and the prior is just a combination of multiple boolean expressions. A group of these sequences can be written as properties and properties can be 'asserted' or 'covered' as desired.

Chapter 3

UVM Verification Hierarchy

UVM is a class library based on transaction-level modeling and all the verification components communicate via transactions. In any ideal UVM test-bench, the hierarchy is made up of the following components.

1. UVM Test-bench
2. UVM Test
3. UVM Environment

3.1 UVM Environment

A top-level UVM environment encompasses agent and scoreboard components and most often, other environments in its hierarchy[13]. It groups several of the critical components of the test bench so that they could easily be configured at one place in any stage if needed. It can have multiple agents for different interfaces and multiple scoreboards for checks on the different type of data transactions. This way, the environment can enable/disable different verification components for specific tasks all in one place.

3.1.1 UVM Agent

A UVM agent comprises of the sequencer, driver and the monitor of an interface. Multiple agents could be used to drive multiple interfaces, and they are all connected to the test-bench through the environment component. A UVM agent could be active or passive. Active agents include a driver and have the ability to drive signals, but passive agents only have the monitor and cannot drive pins. Even though a passive agent consists of the monitor only, it is vital to maintaining the level of abstraction that UVM promises and to maintain its structure by having all agents in the environment and not sub-components like monitors by themselves. By default, the agent is considered active, but this could be changed using the *set()* method of the UVM configuration database.

3.1.1.1 UVM sequence item

A UVM sequence item is the most fundamental component of the UVM hierarchy[14, 15]. It is a transaction that contains data items, methods, and may also contain the constraints imposed on them. A sequence item is the smallest transaction that can happen in a verification environment.

3.1.1.2 UVM sequence

A UVM sequence is a collection of transactions, also called the sequence items. A sequence gives us the ability to use the sequence item as per our requirements and to use as many sequence items as we want. The main job of a sequence is to generate transactions and pass them to the sequencer.

3.1.1.3 UVM sequencer

A sequencer acts as a medium between the sequence items and the driver to control the flow of transactions from multiple sequences. A TLM interface enables communication between the driver and the sequencer.

3.1.1.4 UVM driver

A driver converts the transactions received from the sequencer into pin level activity on the DUT. To do so, it uses methods such as- get_next_item(), try_next_item(), item_done(), put(). It extends from uvm_driver which inherits from uvm_components. The driver class is generally parameterized with the type of request and response sequence items.

3.1.1.5 UVM Monitor

A UVM monitor looks at the pin level activity of the DUT and converts it back into transactions to send it to other components for further analysis. Generally, the monitor processes transactions like coverage collection and logging before sending them to scoreboards.

3.1.2 UVM Scoreboard

The scoreboard collects the transactions from the monitor and performs checks to verify if the collected data matches the expectation or not. The expectation generally comes from a golden reference model often written in languages such as C/C++ and interfaced with the test bench through Direct Programming Interface(DPI).

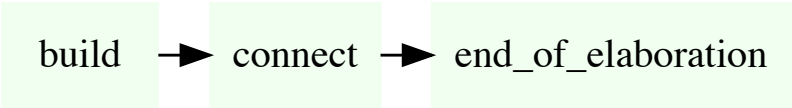


Figure 3.1: UVM Build Phases[2]

3.1.3 UVM Phasing

Phasing is an essential feature of UVM methodology where different phases collect, run and process data to avoid run-time conflicts. Phases are a group of callback methods which could be tasks or functions. All the phases in UVM can be grouped into three main categories which are discussed below[16].

3.1.3.1 Build

The methods in a build phase enable us to build all the components and connect them. These are executed at the start of the simulation. All methods in this phase are functions only and hence execute in zero simulation time. The methods of this phase are shown in Figure.3.1.

3.1.3.2 Run

The stimulus for the test bench is generated and executed in this phase. All the methods in this phase other than *start_of_simulation()* are tasks and are run in parallel. However, the most commonly used ones are the *reset*, *configure* and *main* methods. A detailed list of all the methods in this phase is shown in Figure.3.2.

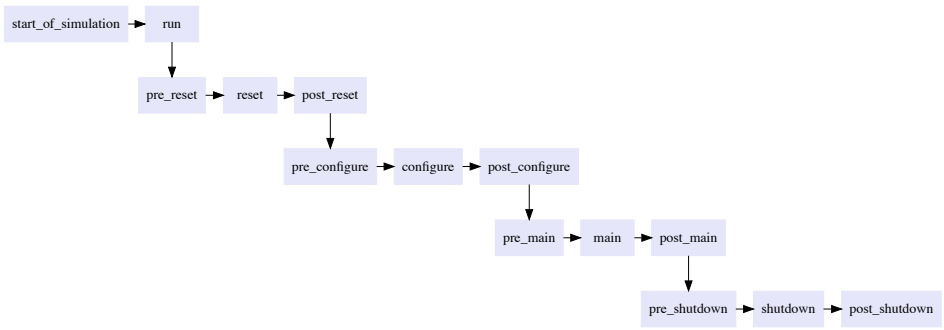


Figure 3.2: UVM Run Phases[2]

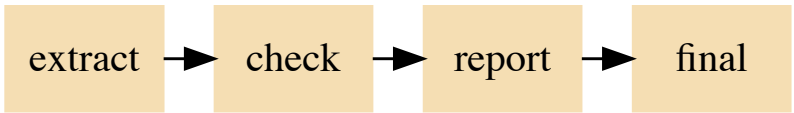


Figure 3.3: UVM Cleanup Phases[2]

3.1.3.3 Cleanup

This phase is mainly to collect the information from scoreboards and to report/present the information extracted towards the end of the test. This phase contains only functions as its methods which take zero simulation time and is executed bottom up. The methods present in this phase are shown in Figure.3.3.

3.2 UVM Test

A UVM test is a top-level component that encapsulates all the test-specific information. The functions of the test component are- instantiating the environment, configuring it and invoking

sequences through it. In test benches where there are multiple focus tests, a base test is first written extending from the *uvm_test* class that instantiates the top-level environment and all the other focus tests are extended from this base test for more specific testing.

3.3 UVM Test bench

The UVM test-bench instantiates the DUT and the UVM test and configures their connections. This is important as most of the times, the UVM test is passed at run-time and needs to be dynamically instantiated. Since all the tests are registered with UVM Factory, the test-bench can instantiate any test that is already registered.

3.4 UVM Debugging

UVM provides the user with excellent built-in debug facilities that come in the form of *'plusargs'* and other features. However, most of these are not explicitly mentioned in the UVM LRM. However, the UVM cookbook mentions some useful features and is the primary source of reference for this section of the paper.

For all of the classes that are derived from the uvm_component, two functions namely *print_config()* and *print_config_with_audit()* are useful for debugging needs. These functions can be called in any phase in the child classes, and they print all of the configuration information. This feature is useful to maintain trackers to have more information on the component itself.

The list of components that are registered with the factory, the Configuration Database table, and all the transaction information can be dumped for review whenever required in the

test-bench. Also, the UVM Register Abstraction Layer (RAL) is also an excellent feature of the library[17]. The results section discusses more on UVM debugging features.

Chapter 4

I2C Overview

Often in an extensive electronic system, there is circuitry for intelligence control, a few converters, and application specific chips. Communication protocols were first introduced to help establish communication between these different blocks in consumer electronics.

Serial interconnects such as a Universal Asynchronous Receiver/Transmitter (UART) were first used, and it came with its set of advantages for different applications. However, the downside of traditional communication between serial ports was that they do not provide any data control. This downside led to the development of Serial Peripheral Interface (SPI).

This interface bus was used to facilitate data communication between micro-controllers and other peripherals with the controller acting like a master and the peripherals behaving like a slave. The SPI bus was based on a push/pull technology and could run at very high speeds making it ideal for high-speed communications[18, 19].

Though the concept was very successful, as more slaves were connected to the master, it needed more circuitry and the bus being a single master and multi-slave slave bus was still not

suitable for all applications. Thus, it led to the development of the Inter-Integrated Circuit or the I2C bus which supported a multi-master and multi-master slave configuration and came with less circuitry. Soon after its introduction, the bus was used for short distance communication extensively.

4.1 A brief history of I2C

The I2C bus was originally developed by the then Philips Semiconductors (now NXP semiconductors) to simplify a two-way communication using just two communication lines. One, the serial Data line (SDA) and the other, a serial clock line (SCL). This bi-directional bus was later called I2C as it helped for inter-IC control that facilitates data transfers up to a speed of 5 M bps in unidirectional mode. Apart from this, bidirectional transfers of data can also be made in different speed modes. A standard mode for 100 K bps transfers, a fast mode for transfers between 400 K bps- 1 M bps and an ultra-fast mode for up-to 3.4 M bps transfers.

An important property of an I2C bus is clock stretching. In an I2C bus, instead of the master and slave agreeing to a predefined baud rate, the master controls the clock speed. But in situations where the slave is not able to cooperate with the clock speed set by the master, it has the capability to pull the clock line low till it catches up-to the speed[20, 21]. The master, on the other hand waits till the clock goes high again. However, when the system is in high speed mode, clock stretching is allowed only after receiving an ACK from slave[22].

The main advantage of this protocol is that it facilitates communication between multiple masters and multiple slaves without any loss of data. This is possible by establishing a safe communication path between one master and one slave at one instant of time so that the master

controls only one slave at one point of time. The slaves each have a unique address that is used by the master for a secure connection.

Since the I2C bus is capable of communicating with only one slave at a time, it is more efficient for short-distance communication between many peripherals and finds a good application in SoC architecture. In this project, the slave of an I2C is modeled as a verification IP and is used to validate a master core.

4.1.1 Advantages

- Less circuitry as only two lines are used for communication between multiple masters and multiple slaves
- Unique address to each device that is in the design scope.
- Prevents data corruption

4.2 I2C Protocol

An I2C system typically uses two lines for data transfers. One serial clock line also called SCL, and one serial data line also called SDA. The data transfer using an I2C protocol is done with only these two lines irrespective of the number of masters or slaves connected to the system as shown in Fig.4.1. This also makes the protocol a little slow and limits its use in high-speed data transfers. The communication can be divided into four parts to understand its operation:

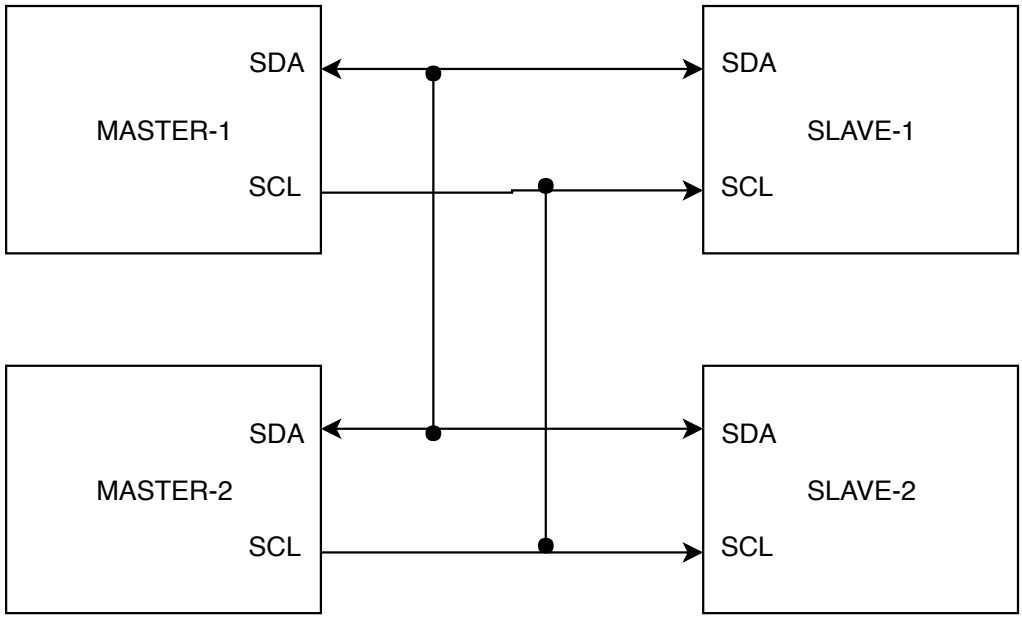


Figure 4.1: I2C Master-slave Communication[3]

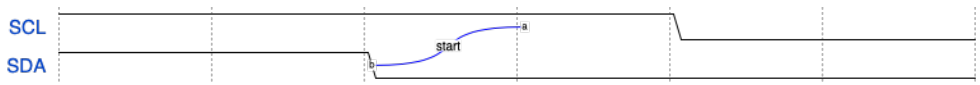


Figure 4.2: I2C START Condition[4]

4.2.1 Generation of START signal

A start condition is when the I2C is ready for communicating with a slave. To indicate to the slaves that communication is about to start, the master generates a start condition[23]. In terms of logic, this condition occurs when the SDA transitions from high to low while the SCL is high. It can be seen in Fig.4.2. If the master needs to complete data cycles before generating a final stop condition, it can do so using the repeated start condition. This is used to re-establish the master-slave communication without generating a stop condition after the previous data transfer. In this case, the SCL goes from low to high while the SDA is high. It can be seen in Fig.4.3.

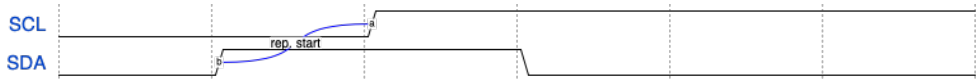


Figure 4.3: I2C Repeated START condition[4]

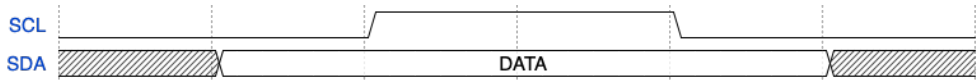


Figure 4.4: Data transfer condition[4]

4.2.2 Transfer of data

Immediately after a Start condition is detected, all the slaves anticipate an address byte. This byte helps the slaves to detect if the master wants to establish a connection with one among them and which one exactly. As soon as the slave with the matching address is detected, the slave immediately responds by acknowledging that it is the right slave and the master can start sending data. The slave acknowledges by pulling the SDA low and immediately making it high again. The master core must have the ability to detect this feature, and it has been tested as a part of this study.

4.2.3 Generation of STOP signal

After the master establishes a connection with the desired slave and transfers data, the master has to generate a stop condition to terminate the connection or to connect with a different slave. This condition is crucial if other masters wish to use the SDA and SCL lines. Fig.4.5 shows the stop condition and as it can be seen, the SDA line must go from high to low while the SCL is low to generate a stop condition.

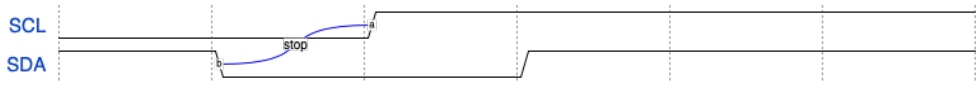


Figure 4.5: I2C STOP Condition

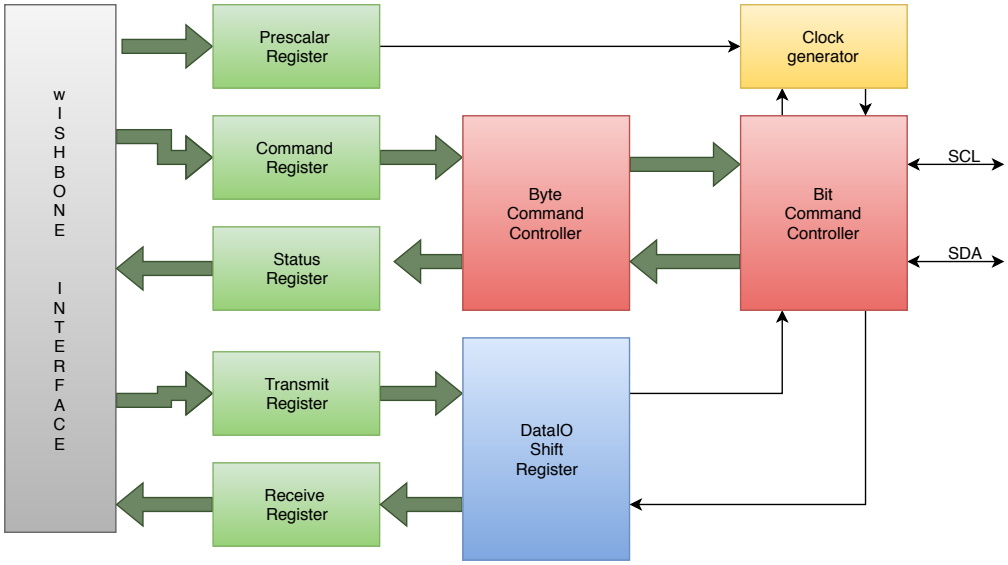


Figure 4.6: I2C master architecture

4.3 I2C Master Core

The I2C master core for the current study has been taken from Opencores[24], an open-source software platform for IP source codes. The architecture of the I2C master core is shown in Fig.4.6. The source code for the master core is distributed among multiple Verilog files which are - the master top module, the bit command controller, the byte command controller. The relationship between these modules is shown in Fig.4.6.

4.3.1 Design Features

- Compatible with multiple masters.
- The clock frequency could be easily programmed by software.
- Includes clock stretching.
- Supports wait state generation.

- The acknowledge bit is software programmable.
- The core has interrupt driven byte-by-byte data transfers.
- The core supports different modes of operating conditions like- start, stop, repeated stop and detects these conditions.
- Support to detect if the bus is busy processing other requests.
- Provides support for both 7-bit and 10-bit addressing modes.
- The design is completely synthesizable.

Chapter 5

I2C Detail Design

5.1 Wishbone Interface Signals

The pin level details of all the wishbone signals in the interface are outlined in Table.5.1.

5.2 Register Model

The I2C design implements a register model which is shown in the table as shown here. This Chapter also explains the in-depth significance of every register and all its fields. The Table.5.2 provides a brief description of what the register is and it's reset value, but each register has its description in the Register description section that expands on bit level details.

The Prescale register is for scaling the SCL clock line. The core uses a 5*SCL internally due to it's architecture and hence this register must be programmed to work at this frequency[19, 25, 26]. This value is changed only when the EN bit of the control register is cleared.

Table 5.1: Wishbone signal details

Port	Width	Direction	Description
wb_clk_i	1	Input	Master clock
wb_rst_i	1	Input	Active high synchronous reset
arst_i	1	Input	Asynchronous reset
wb_adr_i	3	Input	Lower address bits
wb_dat_i	8	Input	Data going to core
wb_dat_o	8	Output	Data coming from the core
wb_we_i	1	Input	Write enable
wb_stb_i	1	Input	Strobe/Core select input
wb_cyc_i	1	Input	Valid bus cycle
wb_ack_o	1	Output	Bus cycle acknowledge
wb_inta_o	1	Output	Interrupt signal output

Table 5.2: Registers

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

Table 5.3: Control Register description

Bit	Access	Description
7	RW	EN- Enable Core bit
6	RW	IEN- Interrupt Enable bit
5:0	RW	*Reserved*

Table 5.4: Transmit Register description

Bit	Access	Description
7:1	W	Next byte to transfer via I2C
0	W	Set to '1' when reading from slave and '0' if writing to it

5.3 Register Description

This section expands on the details of the registers that were outlined in Table.5.2.

5.3.1 Control Register

The control register controls the core operation. It has an 'EN' bit which has to be set in order for the core to respond to any new commands. This bit is cleared only when there are no tranfers in progress which is usually after a 'STOP' condition is generated. This register also has an Interrupt enable bit, IEN which is set to '1' whenever an interrupt is enabled and is '0' when the interrupt is disabled.

5.3.2 Transmit Register

The transmit register contains the data to be sent over the bus as well as the direction of the data transfer. The '0'th bit of this register represents the LSB of data in case of a data transfer but if the transfer is an address transfer, this bit denotes whether the data is written to the slave or read from it.

Table 5.5: Receive Register description

Bit	Access	Description
7:0	R	Last byte received via I2C

Table 5.6: Command Register description

Bit	Access	Description
7	W	STA,generate start/rep. start condition
6	W	STO, generate stop condition
5	W	RD, read from slave
4	W	WR, write to slave
3	W	ACK, when receiver sends ACK (=0) or NACK (=1)
2:1	W	*Reserved*
0	W	IACK, Interrupt acknowledge. Clears pending interrupts if set

5.3.3 Receive Register

The receive register contains the byte that is received over the I2C bus.

5.3.4 Command Register

The command register specifies which conditions to generate and tells the Core what commands to do next. All the bits of this register are automatically cleared and are usually read as Zeros.

5.3.5 Status Register

The status register gives information about the status of the core and if any data transfer is in progress. It has the Busy bit which shows if the core is active and is high as long as there a STOP condition is detected. There is an AL bit which is set if the core loses its arbitration under conditions like an illegal STOP condition detection or if master drives SDA high, but it stays low. The 'TIP' bit is useful to see if there are any data transfers in progress. Finally, this

Table 5.7: Status Register description

Bit	Access	Description
7	R	RxACK, Received acknowledge from slave
6	R	Busy, '1' after a START and '0' after a STOP
5	R	AL, Arbitration lost
4:2	R	*Reserved*
1	R	TIP, Transfer in progress. '1' when transferring data. '0' otherwise
0	R	IF, Interrupt flag.

register has an Interrupt flag that is set when there is an arbitration loss or if one byte of data transfer is complete.

Chapter 6

I2C Verification components

This chapter discusses about the architecture of the proposed verification architecture in detail. Fig.6.1 shows the architecture and the various components that are a part of it. A custom script for the current environment helps to generate the required UVM components from a UVM template.

6.1 I2C Interface

The I2C interface consists of the clock, reset and scan insertion signals but is not limited to just the above. It also consists of SCL and SDA signals along with wishbone signals for communication with the wishbone bus functional model and data transfer.

6.1.1 Master Interface

Apart from the standard clock, reset and scan signals, the SDA and SCL signals play a key role in the functioning of the I2C communication protocol. These signals are a part of interface thus making the communication between the slave, master and the bus functional model (BFM)

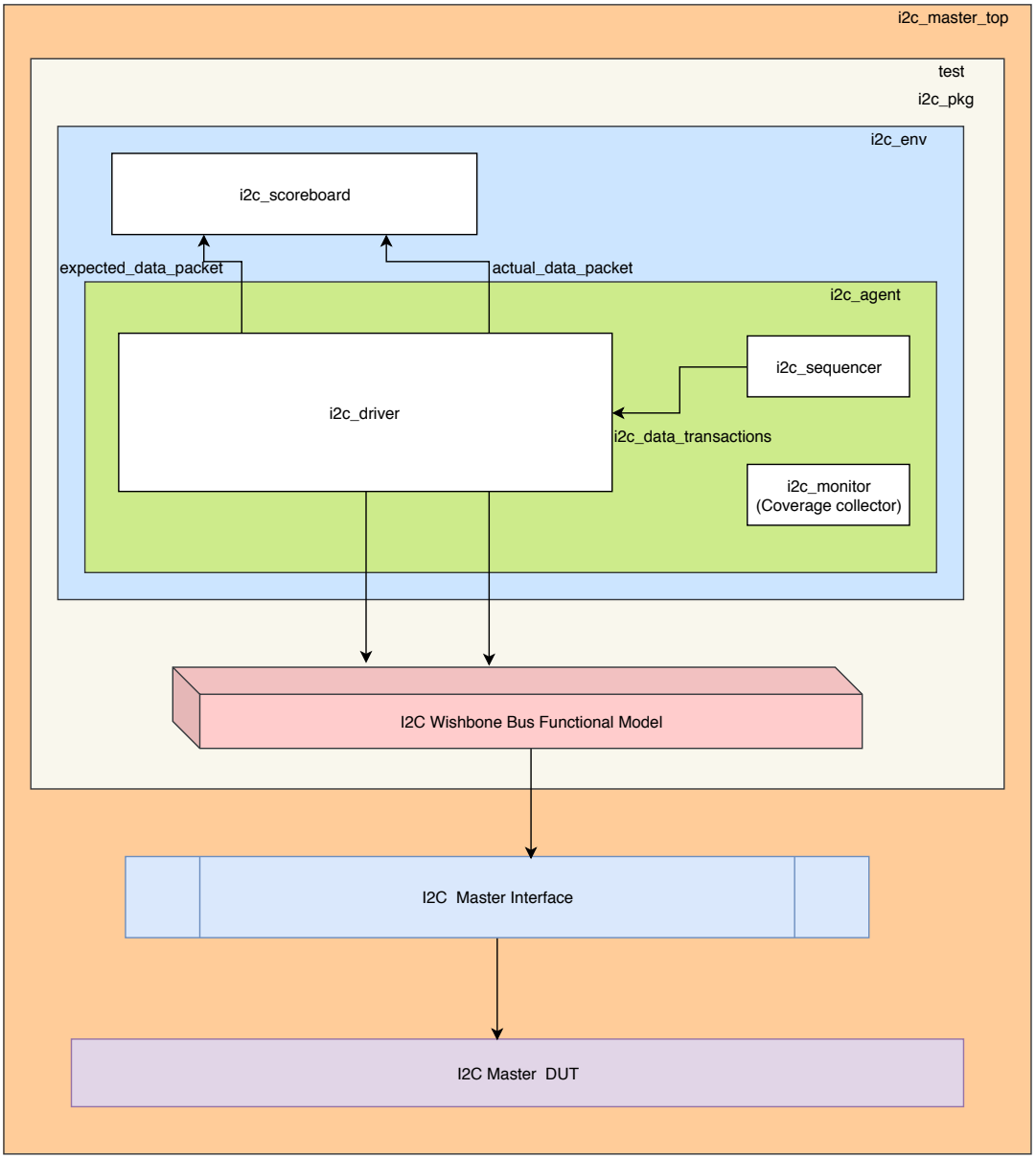


Figure 6.1: I2C Verification Architecture

much simpler.

6.1.2 Wishbone signals

A list of all the wishbone signals can be seen in Table.5.1.

6.2 I2C Agent

The agent encompasses the sequencer, driver and the monitor classes of the I2C test bench architecture. This class connects the TLM interface between the sequencer and the driver that is vital to allow the flow of transactions to the driver.

6.3 I2C Sequences

6.3.1 I2C data transaction

The I2C data transaction is the smallest entity in the I2C test bench architecture and contains the following ports/values as a part of each instance of it:

- An 8-bit value to select from the available slave addresses. The possible slave values are added as a constraint in the transaction.
- Three random data bytes that are later combined to form an array when sent to the driver. It was observed that the randomization is better when data bytes are separated instead of randomizing an array of them. The variables were also made random cyclic to collect a different set of data and to observe how the nature of the random variable affects functional coverage.

- An random integer which when used with a constraint, acts like a watermark to control the flow of sequences to the driver.
- A 2-bit random variable to decide the buffer depth for each iteration of the driver. It is contained in the transaction and could be any value from 1-3.

6.3.2 Low and mid traffic sequences

The low and mid traffic sequences take an *i2c_data_transaction* as a parameter but differ with the data buffer depth. A low traffic sequence is only one byte deep whereas a mid traffic sequence could be 2 or even 3 bytes deep for data transfers.

6.3.3 Data sequence

The data sequence uses instances of the low and mid traffic sequences and repeats them for a certain watermark that is set in the transaction.

6.4 I2C Wishbone BFM

6.4.1 Working of the BFM

Traditionally, before verification methodologies like OVM and UVM came into existence, BFM were used as drivers to drive the design interfaces. But as more improved methods were developed, a driver handles most of the functions that were previously handled by the BFM. However, the driver is kept untimed and all the behaviors that is timed is preferably kept separate from the driver code and thus BFM came into use again.

The driver controls the BFM anyway and calls the tasks in BFM depending on the protocol

and the interface. To summarize, the driver drives the BFM and the BFM wiggles the device under test (DUT) pins. Similarly, a monitor can call a BFM task that reads from the DUT and return the data read by still keeping all the pin level information hidden from the monitor.

6.4.2 Tasks in the BFM

6.4.2.1 Reset task

The reset task of the wishbone BFM is a static task which resets all the wishbone signals into their default values. A description of the default values and the details of the pins can be seen in the table.

6.4.2.2 Write task

The write task takes in the arguments delay, the master interface handle, the address and the data that needs to be written to the wishbone. After waiting for the number of clock cycles as given by the delay parameter, the write task sends the address and data on the wishbone signals and assigns default values to the other wishbone signals.

6.4.2.3 Read task

The read task is similar to the write task in terms of the parameter it takes in but instead of writing data to the wishbone, the task read data from the given address from the wishbone and outputs that data instead.

6.5 I2C Driver & Monitor

The I2C driver handles all the pin level transactions of the DUT and uses a wishbone BFM to write and read address and data to/from the wishbone. As the test bench does not use any handshaking between components other than the sequencer and the driver, all the transaction handling is done by the driver and the monitor monitors all the coverage on ports except the data ports.

6.6 I2C Scoreboard

The I2C scoreboard receives data packets from the driver for both the expected and the actual transaction. Each packet has the slave address and accompanying data. The data that is written to a slave is the data that is expected to come out of the bus and is thus called as expected data. The data that is read from the bus after every write is the actual data coming from the bus.

Every packet is stored in a queue and comparisons are made to make sure both the slave address and the data match for a successful check.

6.7 I2C Environment

The environment class has instances of the agent, scoreboard and handles the analysis port connections between the driver and the scoreboard for transfer of packets.

6.8 I2C Test

The test class decides which sequence or which group of sequences should be used for the current simulation cycle. A standard UVM test bench can have many such tests and a run time knob could be set to decide which test executes for that current simulation run.

6.9 I2C Top

The *i2c_master_top_test.sv* file contains the top module that instantiates the master core, the 3 I2c slaves with addresses 16,1 and 2 in decimal system respectively. The top module also handles clock generation of the master core. All the assertions pertaining to the I2C bus are in this module[15]. This module contains the clock generator for a 5 MHz clock.

Chapter 7

Results and Discussion

The results of the simulations and analyses of the results is provided in this chapter. The chapter discusses the simulation results of both the register-transfer level (RTL) and gate level simulation and also goes deeper into the use of UVM specific features and their effectiveness during the debug phase.

7.1 RTL and Gate level Simulations

The simulations at both the RTL and Gate-level are error free and the screen captures of the simulation waveforms can be found in Fig.7.1 and Fig.7.2 respectively. The simulations ran for up to 100,000 times at both RTL and Gate-level and the test successfully passed in both the cases.

The Table.7.1 shows the synthesis results when the master core RTL was synthesized in different technology nodes- 32nm, 65nm and 180nm.

Figs.7.3,7.4,7.5 and 7.6 show the screen captures of the coverage as seen in the IMC (Integrated Metrics Center) tool[27]. When the test was run for a high number of transactions, the

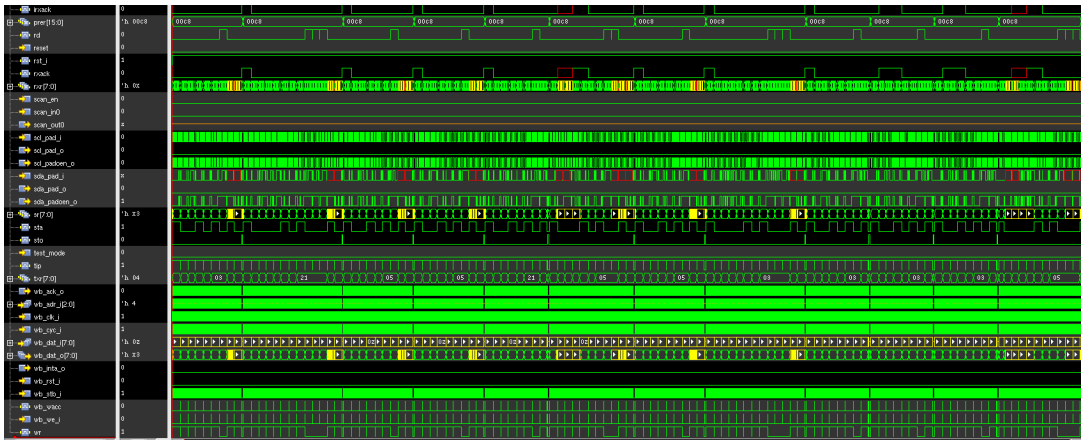


Figure 7.1: RTL simulation Waveforms

Figure 7.2: Net-list simulation Waveforms

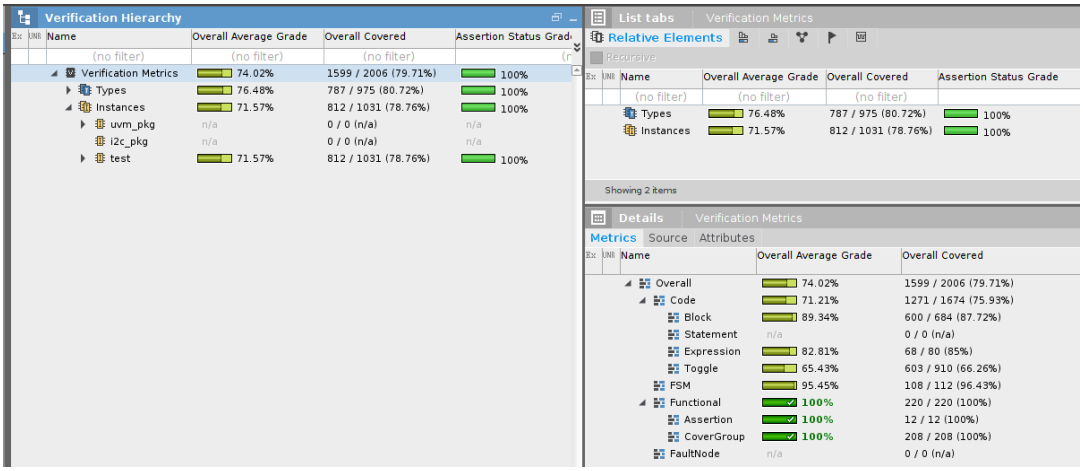


Figure 7.3: Overall Coverage

functional coverage of *100%* was hit.

7.2 Simulation time dependence on transaction count

It is a known fact that the simulation time depends on the number of transaction sent to the DUT. To provide an accurate representation of this dependency, a simulation experiment was done with varying transactions for each run and the run time was captured using C-DPI. The Fig.7.7shows the captured results through histograms.

7.3 Measure of randomization effectiveness

The SystemVerilog language provides many features to assist a verification engineer to generate random stimulus. To test the effectiveness of a random and a random cyclic variable, an experiment was done to check the functional coverage metric for varying number of transactions. In one case, the variables were made random and in the other, the variables were made random cyclic. The results are summarized in Table.7.2 and shown in Fig.7.8.

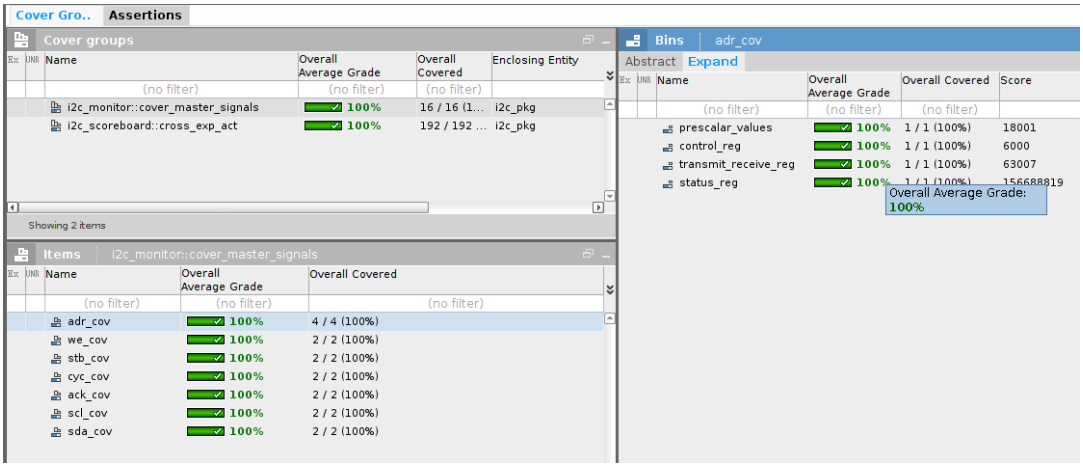


Figure 7.4: Functional Coverage of ports

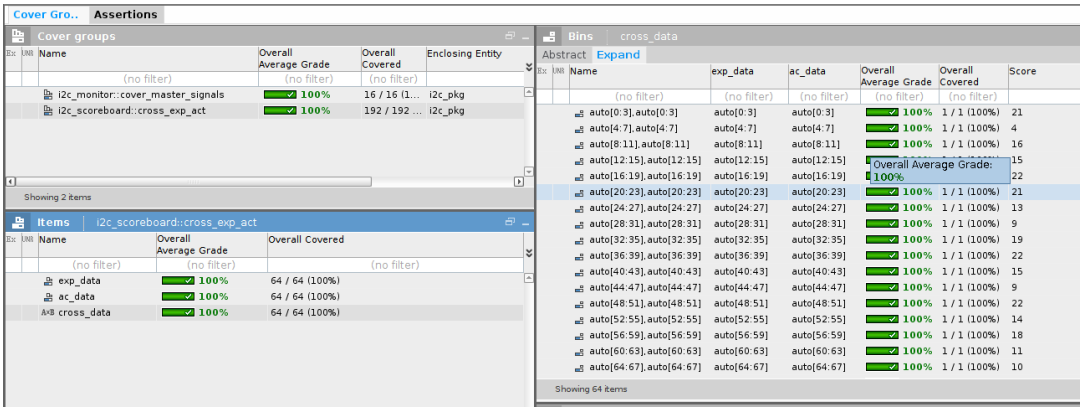


Figure 7.5: Functional Coverage of data(including cross)

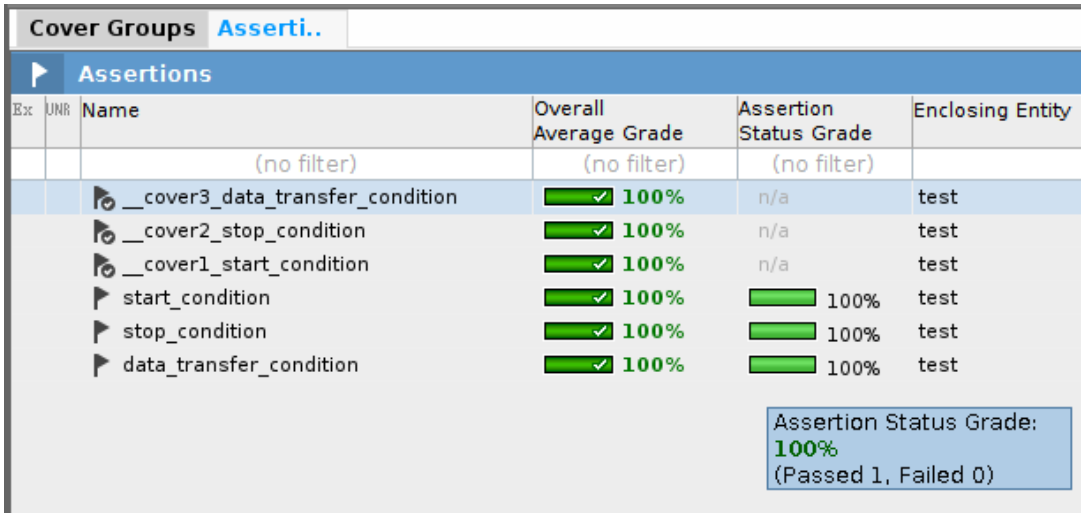


Figure 7.6: Coverage on Assertions

Table 7.1: Synthesis results

Technology		180 nm	65 nm	32 nm
Area (μm^2)	Combinational area	6869.01	954.72	974.889
	Buf/Inv area	1031.18	101.88	123.00
	Sequential area	10335.12	1227.60	1084.68
	Total cell area	17204.14	2182.32	2059.58
	Number of Gates	1725	1515	1354
Power (μW)	Cell Internal Power	90.31	4.98	3.96
	Net Switching Power	40.59	2.02	1.23
	Cell Leakage Power	0.69	0.78	217.64
	Total Power	131	7.09	222.84
Timing	Worst path delay	16.14	18.65	18.85
Test Coverage		100%	100%	100%

Table 7.2: Data sets for coverage with rand and randc variables

Data set	1	2	3
Transactions	20	200	2000
rand variables	76.56%	98.44%	95.31%
randc variables	78.12%	99.22%	100%

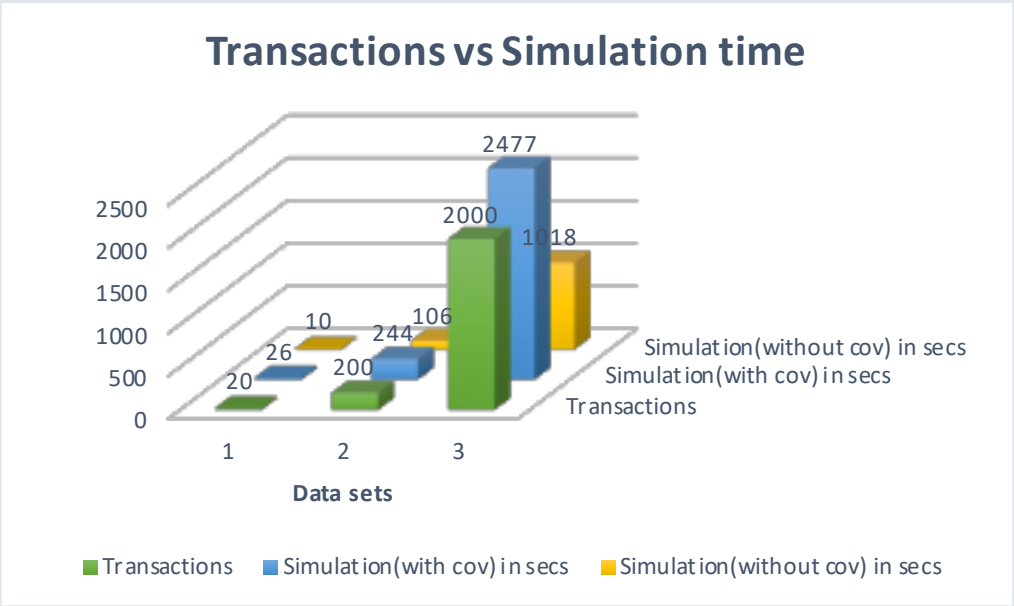
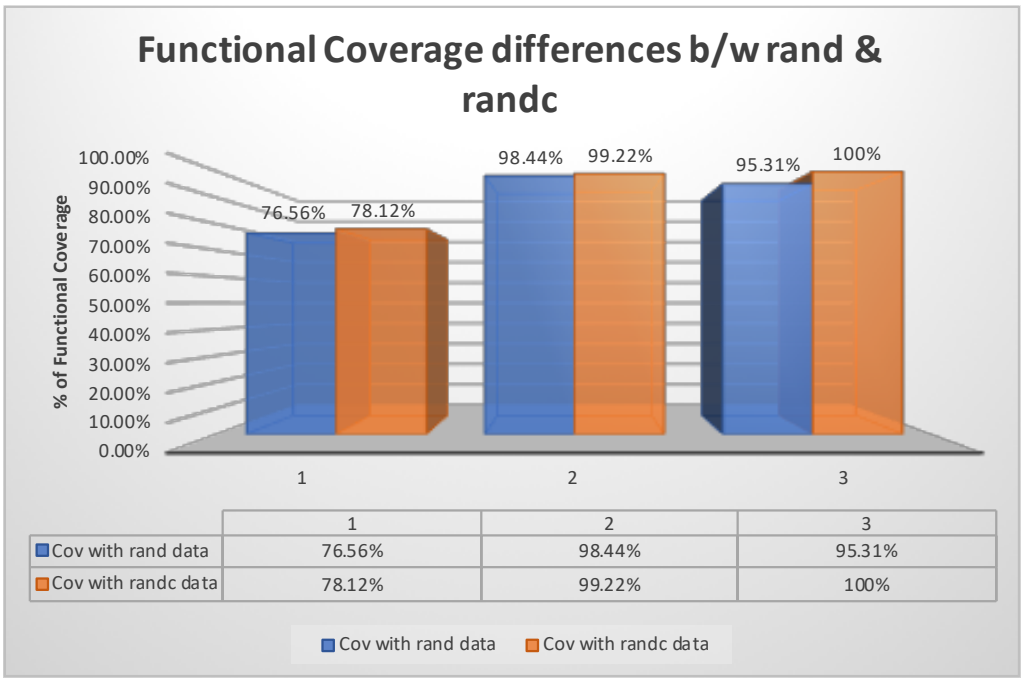


Figure 7.7: Transactions and simulation time relationship



	1	2	3
Cov with rand data	76.56%	98.44%	95.31%
Cov with randc data	78.12%	99.22%	100%

Figure 7.8: Rand and randc variables

```
xcelium> uvm_factory -print -all_types
UVM_INFO /tools/rhel6/cadence/xcelium/current/tools/methodology/UVM/CDNS-1.2/sv/src/base/uvm_factory.svh(1645) @ 452571901.00ns: reporter [UVM/FACTORY/PRINT]
#### Factory Configuration (*)

  No instance or type overrides are registered with this factory

All types registered with the factory: 65 total
  Type Name
  ---------
  cdns_sst2_uvm_recorder
  cdns_uvm_sst2_tr_stream
  i2c_agent
  i2c_base_slave
  i2c_data_sequence
  i2c_data_transaction
  i2c_driver
  i2c_env
  i2c_low_traffic_sequence
  i2c_master_top_test
  i2c_medium_traffic_sequence
  i2c_monitor
  i2c_scoreboard
  i2c_wb_master_bfm
(*) Types with no associated type name will be printed as <unknown>

####
```

Figure 7.9: UVM factory components

7.4 UVM features for debugging

The concept of UVM Factory and Configuration Database are the major features of UVM that make it stand out from the already successful OVM methodology. The Figs.,7.9,7.10 and 7.11 show the screen captures of the Factory components and database dump as seen in the simulator.

All the components of the test bench are registered with the UVM Factory so that their types could be overridden at any time including run-time. This is a useful feature as we can dynamically change the type of a class or instance without any changes in code by simply using run time UVM switches. This is also the reason why the objects in UVM are created using a type_id instead of a *'new()'* constructor. A dump of the factory components in the current scope can be seen in Fig.7.9.

The UVM configuration database works like a look up table and helps to store values or types that can be accessed throughout the environment. Using this method, a value can be *"set"* by any component in the resource database and any component can retrieve or *"get"* its value. A dump of this database for the current verification environment is shown in Fig.7.11.

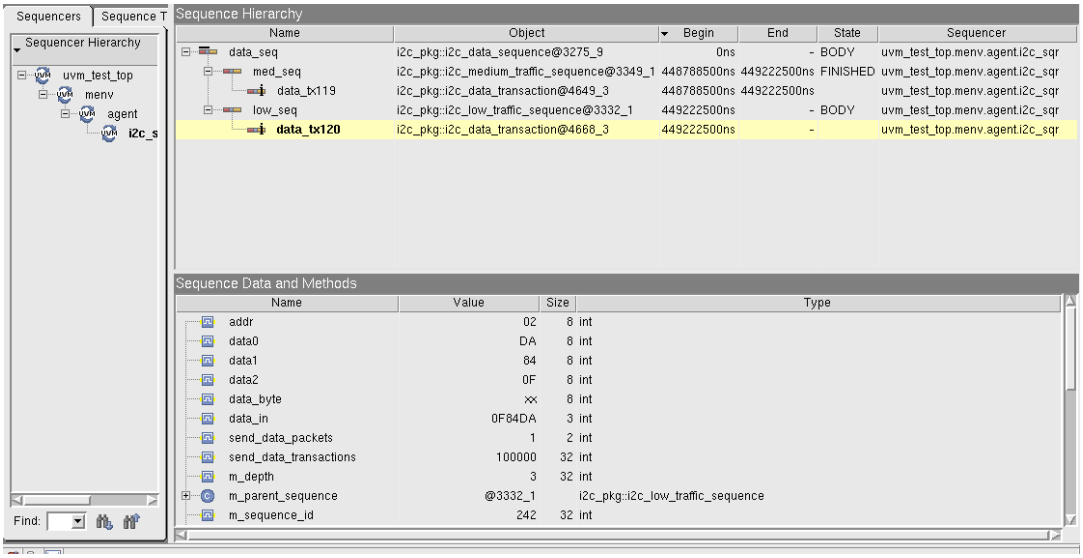


Figure 7.10: UVM Sequence viewer

Figure 7.11: UVM Configuration DB

7.5 Observations

After a successful verification of the master core under several constraints, the following can be concluded:

1. The master core can only read/write up-to 3 bytes of data after a start condition is seen and before a stop is generated. The number of bytes for processing was handled by a random variable in the transaction which can take any value between 1-3.

2. The master core responds well for both 3-bit and 7-bit addressing modes. Both the cases were verified successfully.

3. The master core works well in the normal mode for data processing.

4. The simulation time is proportional to the number of transactions driven by the driver to

the DUT and this can be seen in the Fig.7.7.

5. The functional coverage improves as the number of transactions increase and this can be concluded from the Fig.7.2.

6. Using the results obtained by using random and random cyclic variables in transactions, it can be said that using random cyclic variables provide better coverage even if the total transactions are not too high.

7. The synthesis results were as expected as chip area decreased with technology advancement as seen in Table.7.1.

8. The leakage power was low in higher technology nodes and the increased as the technology node reduces as seen in Table.7.1.

Chapter 8

Conclusion

The master core was successfully verified under various constraints, and the effort has been documented in this paper. The functional coverage goal of 100% was successfully met, and more on it has been elaborated in the previous chapters. Overall, the master core was tested with multiple slaves, and the simulation results showed no data mismatches, and it could be said that the design works well for normal speed mode. Looking at the coverage metrics, it is safe to say that the design functions well and delivers what it promises.

8.1 Future Work

The verification effort to validate the master core was successful, and the coverage goals were met. There are other claims in the design document that could be validated through a further study on the same topic and by enhancing the current test-bench. A few suggestions for a continued effort in this area are presented here:

- The master core was tested with multiple slaves and was found to work well. However, the test bench can be extended to study if it works well with a multi-master configuration.

- The current verification environment successfully tested the master core in the normal speed mode. However, the design document claims that the core works well for high-speed modes which has not been tested in this study.

- Clock stretching was also not tested in the current environment as only the normal speed mode was verified.

References

[1] Ashok B. Mehta. *ASIC/SoC Functional Design Verification*. Springer, 2018.

[2] Emden R. Gansner, eleftherios Koutsofios, and Stephen North. *Drawing graphs with dot*, January 2015.

[3] Pallavi Atha Anuja Dhar, Ekta Dudi and Hema Tiwari. Coverage driven verification of I2C Protocol using system Verilog. *International Journal of Advanced Research in Engineering and Technology*, 7(3):103–113, 2016.

[4] Wavedrom. URL: `https://wavedrom.com`.

[5] Deepa Kaith, Janakkumar B. Patel, and Neeraj Gupta. An Introduction to Functional Verification of I2C Protocol using UVM. *International Journal of Computer Applications*, 121(13), 07 2015.

[6] B. Santosh Kumar, L. Ravi Chandra, A. L. G. N. Aditya, and Fazal Noor Basha & T. Praveen Blessington. Design and functional verification of i2c master core using ovm, 05 2012.

[7] K. Fathy, K. Salah, and R. Guindi. A proposed methodology to improve UVM-based test generation and coverage closure. In *2015 10th International Design Test Symposium (IDT)*, pages 147–148, Dec 2015. `doi:10.1109/IDT.2015.7396754`.

[8] IEEE Approved Draft Standard for System Verilog–Unified Hardware Design, Specification, and Verification Language. *IEEE P1800/D6, August 2012*, pages 1–1312, Dec 2012.

[9] G. Tumbush and Chris Spear. *SystemVerilog for Verification, Third Edition: A Guide to Learn- ing the Testbench Language Features*. Springer, 2012.

[10] P. D. Mulani. SoC Level Verification Using System Verilog. In *2009 Second International Conference on Emerging Trends in Engineering Technology*, pages 378–380, Dec 2009. doi:10.1109/ICETET.2009.205.

[11] Ray Salemi. *The UVM Primer: An Introduction to the Universal Verification Methodology*. Number 4. Boston Light Press, 2013.

[12] IEEE Standard for Universal Verification Methodology Language Reference Manual. *IEEE Std 1800.2-2017*, pages 1–472, May 2017. doi:10.1109/IEEESTD.2017.7932212.

[13] W. Ni and J. Zhang. Research of reusability based on UVM verification. In *2015 IEEE 11th International Conference on ASIC (ASICON)*, pages 1–4, Nov 2015. doi:10.1109/ASICON.2015.7517189.

[14] K. Salah. A UVM-based smart functional verification platform: Concepts, pros, cons, and opportunities. In *2014 9th International Design and Test Symposium (IDT)*, pages 94–99, Dec 2014. doi:10.1109/IDT.2014.7038594.

[15] N. Sudhish, B. R. R., and H. Yagain. An Efficient Method for Using Transaction Level Assertions in a Class Based Verification Environment. In *2011 International Symposium on Electronic System Design*, pages 72–76, Dec 2011. doi:10.1109/ISED.2011.32.

[16] K. D. Larson. Translation of an existing VMM-based SystemVerilog testbench to OVM. In *2008 45th ACM/IEEE Design Automation Conference*, pages 237–237, June 2008. doi:10.1145/1391469.1391529.

[17] T. Lins and E. Barros. The development of a hardware abstraction layer generator for system-on-chip functional verification. In *2010 VI Southern Programmable Logic Conference (SPL)*, pages 41–46, March 2010. doi:10.1109/SPL.2010.5483004.

[18] NXP Semiconductors. *I2C-bus specification and user manual*, v.6 edition, April 2014.

[19] Purvi Mulani, Jignesh Patoliya, Hitesh Patel, and Dharmendra Chauhan. Verification of I2C DUT using System Verilog. *International Journal of Advanced Engineering Technology*, 1, 2010.

[20] L. M. Kappaganthu, M. D. Prakash, and A. Yadlapati. I2C protocol and its clock stretching verification using system verilog and UVM. In *2017 International Conference on Inventive Communication and Computational Technologies (ICICCT)*, pages 478–480, March 2017. doi:10.1109/ICICCT.2017.7975245.

[21] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner. Information flow isolation in I2C and USB. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 254–259, June 2011.

[22] I2C Primer. URL: https://www.i2c-bus.org/i2c-primer/.

[23] Jonathan Valdez and Jared Becker. Understanding the I2C Bus. 2015.

[24] Opencores. URL: https://opencores.org.

[25] Richard Herveille. *I2C- Master Core Specification*. Opencores, July 2003.

[26] M. Sukhanya and K. Gavaskar. Functional verification environment for I2C master controller using system verilog. In *2017 Fourth International Conference on Signal Processing, Communication and Networking (ICSCN)*, pages 1–6, March 2017. doi:10.1109/ICSCN.2017.8085732.

[27] V. Viswanathan, J. Runhaar, D. Reed, and J. Zhao. Tough Bugs vs. Smart Tools - L2/L3 Cache Verification Using System Verilog, UVM and Verdi Transaction Debugging. In *2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, pages 25–29, Dec 2016. doi:10.1109/MTV.2016.15.

Appendix I

Source Code

I.1 Interface

```
interface i2c_master_interface;
  //port declarations
  logic clk;
  logic reset;
  logic test_mode;
  logic scan_in0;
  logic scan_out0;
  logic scan_en;
  var logic wb_clk_i;
  var logic wb_rst_i;
  logic rstn;
  logic [31:0] wb_adr_i;
  logic [7:0] wb_dat_o;
```

```
14    wire [7:0] wb_dat_i;
15    logic wb_we_i;
16    logic wb_stb_i;
17    logic wb_cyc_i;
18    wire wb_ack_o;
19    logic wb_inta_o;
20    wire scl;
21    wire scl0_o;
22    wire scl0_oen;
23    wire sda;
24    wire sda0_o;
25    wire sda0_oen;
26 endinterface : i2c_master_interface
```

I.2 Sequence Item

```
1  //
   ************************************************************************
2  //
3  //   Author      ::   Shravani Balaraju
4  //   Date Created   ::   2019-01-19
5  //   Module name    ::   i2c_data_transaction
6  //
7  //
   ************************************************************************
8  `define TRANSACTION_WATERMARK 100
9
10 class i2c_data_transaction extends uvm_sequence_item ;
11
12 //-----------------------------------------------
13 //     Class variables
14 //-----------------------------------------------
15   rand logic[7:0] addr;
16   randc logic[7:0] data0;
17   randc logic[7:0] data1;
18   randc logic[7:0] data2;
19   rand logic[2:0][7:0] data_in;
```

```
20    rand bit[1:0] send_data_packets;
21    rand integer send_data_transactions;
22    logic[7:0] data_byte;
23
24
25    constraint c1 {
26      send_data_packets > 0;
27
28    }
29
30    constraint c2{
31      send_data_transactions == `TRANSACTION_WATERMARK;
32    }
33
34    constraint c3{
35      addr inside {7'b0010_000, 7'b0000_001, 7'b0000_010};
36    }
37
38    constraint c4{
39      data0 != data1;
40      data0 != data2;
41      data1 != data2;
42    }
43
44  //-----------------------------------------------------
```

```
45 //      Analysis ports
46 //-----------------------------------------------------------
47
48 //-----------------------------------------------------------
49 //      Covergroups
50 //-----------------------------------------------------------
51
52 //-----------------------------------------------------------
53 //      new function
54 //-----------------------------------------------------------
55   function new(string name = "");
56     super.new(name);
57
58   endfunction: new
59
60 //-----------------------------------------------------------
61 //      UVM Phases
62 //-----------------------------------------------------------
63
64
65
66 //-----------------------------------------------------------
67 //      Tasks/Functions
68 //-----------------------------------------------------------
69
```

```
70    `uvm_object_utils_begin(i2c_data_transaction)
71      `uvm_field_int(addr, UVM_ALL_ON)
72      `uvm_field_int(data0, UVM_ALL_ON)
73      `uvm_field_int(data1, UVM_ALL_ON)
74      `uvm_field_int(data2, UVM_ALL_ON)
75      `uvm_field_int(data_byte, UVM_ALL_ON)
76    `uvm_object_utils_end
77
78  endclass: i2c_data_transaction
```

I.3 Sequencer

```
//
    ************************************************************
//
//   Author      ::   Shravani Balaraju
//   Date Created   ::   2019-01-19
//   Module name    ::   i2c_data_sequence
//
//
    ************************************************************

class i2c_low_traffic_sequence extends uvm_sequence#(
    i2c_data_transaction);
  `uvm_object_utils(i2c_low_traffic_sequence)
  integer loop_dut;
  i2c_data_transaction data_tx;
  integer l;

  function new(string name = "");
    super.new(name);
    l = 0;
  endfunction: new
```

```
19
20
21  task body();
22
23    begin
24    data_tx = i2c_data_transaction::type_id::create(.name(
        $sformatf("data_tx%0d",l)),.contxt(get_full_name()));
25    start_item(data_tx);
26    assert(data_tx.randomize() with {data_tx.send_data_packets
        == 1 ;});
27    loop_dut = data_tx.send_data_transactions;
28    uvm_config_db #(integer) :: set(null,"uvm_low_traffic_seq"
        ,"looping",loop_dut);
29    uvm_config_db #(integer) :: set(null,"uvm_low_traffic_seq"
        ,"low_seq_count",l);
30    _set_transaction_data_as_array;
31    finish_item(data_tx);
32    l++;
33    end
34  endtask : body
35  task _set_transaction_data_as_array;
36      data_tx.data_in[0] = data_tx.data0;
37      data_tx.data_in[1] = data_tx.data1;
38      data_tx.data_in[2] = data_tx.data2;
39
```

```
  endtask : _set_transaction_data_as_array

endclass : i2c_low_traffic_sequence

class i2c_medium_traffic_sequence extends uvm_sequence#(
    i2c_data_transaction);
  `uvm_object_utils(i2c_medium_traffic_sequence)
    i2c_data_transaction data_tx;
    integer m;

  function new(string name = "");
    super.new(name);
    m = 0;
  endfunction: new


  task body();

    begin
    data_tx = i2c_data_transaction::type_id::create(.name(
        $sformatf("data_tx%0d",m)),.contxt(get_full_name()));
    start_item(data_tx);
    assert(data_tx.randomize() with {  data_tx.
        send_data_packets <= 3;});
    _set_transaction_data_as_array;
```

```
62      uvm_config_db #(integer) :: set(null,"uvm_med_traffic_seq"
            ,"med_seq_count",m);
63      finish_item(data_tx);
64     m ++;
65      end
66   endtask : body
67   task _set_transaction_data_as_array;
68        data_tx.data_in[0] = data_tx.data0;
69        data_tx.data_in[1] = data_tx.data1;
70        data_tx.data_in[2] = data_tx.data2;
71
72   endtask : _set_transaction_data_as_array
73
74 endclass : i2c_medium_traffic_sequence
75
76
77
78
79
80 class i2c_data_sequence extends uvm_sequence#(
     i2c_data_transaction);
81   `uvm_object_utils(i2c_data_sequence)
82
83 //————————————————————————————————————————
84 //      Class variables
```

```
85  //-----------------------------------------------------------
86
87  int loop_data;
88  int loop_dut;
89  i2c_data_transaction data_tx;
90  i2c_low_traffic_sequence low_seq;
91  i2c_medium_traffic_sequence med_seq;
92  integer l;
93  integer m;
94
95  //-----------------------------------------------------------
96  //      Analysis ports
97  //-----------------------------------------------------------
98
99  //-----------------------------------------------------------
100 //      Covergroups
101 //-----------------------------------------------------------
102
103 //-----------------------------------------------------------
104 //      new function
105 //-----------------------------------------------------------
106   function new(string name = "");
107     super.new(name);
108     l = 0;
109     m = 0;
```

```
  endfunction: new

//-----------------------------------------------------
//      UVM Phases
//-----------------------------------------------------

//-----------------------------------------------------
//      Tasks/Functions
//-----------------------------------------------------

  task body();
    data_tx = i2c_data_transaction::type_id::create(.name("data_tx"),.contxt(get_full_name()));
    low_seq = i2c_low_traffic_sequence::type_id::create(.name("low_seq"),.contxt(get_full_name()));
    med_seq = i2c_medium_traffic_sequence::type_id::create(.name("med_seq"),.contxt(get_full_name()));

    m_sequencer.set_arbitration(UVM_SEQ_ARB_USER);
      data_tx.randomize();
      loop_dut = data_tx.send_data_transactions;

  repeat(loop_dut)
    begin
    low_seq.start(m_sequencer,this,400);
```

```
      med_seq.start(m_sequencer,this,200);
      end
   `uvm_info(get_name(),$sformatf("status: %t Testbench complete ",$time()),UVM_LOW)

   endtask : body

   task post_body();
   uvm_config_db #(integer) :: set(null,"uvm_low_traffic_seq","low_seq_count",l);
   uvm_config_db #(integer) :: get(null,"uvm_med_traffic_seq","med_seq_count",m);
   $display("\n\n \t\tTotal transactions = %0d",l + m + 1);
   endtask : post_body

endclass: i2c_data_sequence

typedef uvm_sequencer#(i2c_data_transaction) i2c_data_sequencer;
```

I.4 Driver

```
//
    ************************************************************
//
//   Author      ::   Shravani Balaraju
//   Date Created   ::   2019-01-19
//   Module name    ::   i2c_driver
//
//
    ************************************************************

class i2c_driver extends uvm_driver#(i2c_data_transaction) ;
  `uvm_component_utils(i2c_driver)

//-----------------------------------------------
//     Class variables
//-----------------------------------------------
  virtual i2c_master_interface mif;
  parameter PRER_LO = 3'b000;
  parameter PRER_HI = 3'b001;
  parameter CTR     = 3'b010;
  parameter RXR     = 3'b011;
```

```
20    parameter TXR      = 3'b011;
21    parameter CR       = 3'b100;
22    parameter SR       = 3'b100;
23
24    parameter TXR_R    = 3'b101; // undocumented / reserved
         output
25    parameter CR_R     = 3'b110; // undocumented / reserved
         output
26
27    parameter RD       = 1'b1;
28    parameter WR       = 1'b0;
29    logic [7:0] SADR;
30    logic [7:0] q, data_read_from_wb;
31    logic [7:0]sending_random_data;
32    logic [7:0] storing_data_sent[$];
33    int i = 0;
34    int j = 0;
35    int loopit;
36    int data_loop;
37    i2c_data_transaction data_txn;
38
39 //-----------------------------------------------
40 //     Analysis ports
41 //-----------------------------------------------
42
```

```
  uvm_analysis_port #(i2c_data_transaction) data_to_bfm;
  uvm_analysis_port #(i2c_data_transaction) data_from_bfm;

//----------------------------------------------------------
//      Covergroups
//----------------------------------------------------------

//----------------------------------------------------------
//      new function
//----------------------------------------------------------
  function new(string name, uvm_component parent);
    super.new(name, parent);
    data_to_bfm = new("data_to_bfm",this);
    data_from_bfm = new("data_from_bfm",this);
  endfunction: new

//----------------------------------------------------------
//      UVM Phases
//----------------------------------------------------------

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    void'(uvm_resource_db#(virtual i2c_master_interface)::
        read_by_name
```

```
66          (.scope("ifs"), .name("i2c_master_interface"), .val(mif)
            ));
67    endfunction: build_phase
68
69    task run_phase(uvm_phase phase);
70    reset();
71    drive();
72    endtask: run_phase
73
74    function void final_phase(uvm_phase phase);
75 //   $finish;
76    endfunction
77
78 //-----------------------------------------------
79 //      Tasks/Functions
80 //-----------------------------------------------
81
82 virtual task reset();
83 begin
84 fork
85    begin
86       mif.clk = 1'b0;
87       mif.reset = 1'b0;
88       mif.scan_in0 = 1'b0;
89       mif.scan_en = 1'b0;
```

```
90      mif.test_mode = 1'b0;
91
92      //resetting the core
93      mif.rstn = 1'b1;
94  #2;
95      mif.rstn = 1'b0;
96      repeat(1) @(posedge mif.clk);
97      mif.rstn = 1'b1;
98
99    //  @(posedge mif.clk);
100   end
101 join
102 end
103
104 endtask
105
106 virtual task drive();
107
108   begin
109     reset_wb_signals;
110
111
112     forever begin
113       seq_item_port.get_next_item(data_txn);
114     //  sending_random_data = data_txn.data_in;
```

```
115     //   storing_data_sent[i] = sending_random_data;
116       data_loop = data_txn.send_data_packets;
117       storing_data_sent[0] = data_txn.data0;
118       storing_data_sent[1] = data_txn.data1;
119       storing_data_sent[2] = data_txn.data2;
120       SADR = data_txn.addr;
121
122
123
124
125     load_i2c_prescalar_values;
126           //`uvm_info(get_name(),$sformatf("status: %t
              programmed pre-scalar registers",$time()),UVM_LOW
              )
127     enable_master_core;
128     write_slave_address_to_wb(SADR);
129     set_start_command;
130     check_tip_bit;
131     present_slave_memory_address;
132     set_write_command;
133     check_tip_bit;
134     write_data_to_slave;
135     write_slave_address_to_wb(SADR);
136     set_start_command;
137     check_tip_bit;
```

```
138         present_slave_memory_address;
139         set_write_command;
140         check_tip_bit;
141         set_read_bit_for_slave01(SADR);
142         set_start_command;
143         check_tip_bit;
144         read_data_from_slave;
145         acknowledge_slave_read;
146         check_slave_addr;
147         set_stop_command;
148         repeat(2000) @(posedge mif.clk);
149         seq_item_port.item_done();
150
151         end
152 end
153 endtask : drive
154
155
156
157 task reset_wb_signals;
158             i2c_wb_master_bfm::wb_reset(mif);
159             //`uvm_info(get_name(),$sformatf("status: %t Wishbone
                signals reset. Starting TB",$time()),UVM_LOW)
160 endtask : reset_wb_signals
161
```

```
162
163
164
165 task acknowledge_slave_read;
166         // read data from slave
167         i2c_wb_master_bfm::wb_write(1,mif, CR,       8'h20); //
                set command (read, mif.wb_ack_o_read)
168         //`uvm_info(get_name(),$sformatf("status: %t read +
               wb_ack ",$time()),UVM_LOW)
169
170 endtask : acknowledge_slave_read
171
172
173
174
175 task set_read_bit_for_slave01(logic [7:0] slaveaddr);
176         // drive slave address
177         i2c_wb_master_bfm::wb_write(1,mif, TXR, {slaveaddr,RD}
                ); // present slave's address, set read-bit
178 endtask : set_read_bit_for_slave01
179
180
181
182
183
```

```
184
185 task set_stop_command;
186         i2c_wb_master_bfm::wb_write(1,mif, CR,       8'h40); //
                set command (stop)
187         //`uvm_info(get_name(),$sformatf("status: %t
               Generating stop condition",$time()),UVM_LOW)
188
189 endtask : set_stop_command
190
191
192
193
194
195 task present_illegal_slave_address;
196     //TRX changed to CR here
197           // send memory address
198           i2c_wb_master_bfm::wb_write(1,mif, TXR,      8'h88);
                // present slave's memory address
199     set_write_command;
200     //`uvm_info(get_name(),$sformatf("status: %t Write address
           - 10 in slave memory",$time()),UVM_LOW)
201           check_tip_bit;
202           // slave should have send Nack
203     //`uvm_info(get_name(),$sformatf("status: %t Check for
           nack",$time()),UVM_LOW)
```

```
            //if(!q[7])
        //`uvm_info(get_name(),$sformatf("status: %t expected Nack, received ack",$time()),UVM_LOW)
endtask : present_illegal_slave_address



task check_tip_bit;
        // check tip bit
        i2c_wb_master_bfm::wb_read(1,mif, SR, q);
        while(q[1])
            i2c_wb_master_bfm::wb_read(1,mif, SR, q); // poll it until it is zero
    //`uvm_info(get_name(),$sformatf("status: %t tip bit back to '0'",$time()),UVM_LOW)
endtask : check_tip_bit



task present_slave_memory_address;
        // send memory address
```

```
226         i2c_wb_master_bfm::wb_write(1,mif, TXR,      8'h01); //
                present slave's memory address
227 endtask : present_slave_memory_address
228
229
230 task check_slave_addr;
231     write_slave_address_to_wb(SADR);
232     set_start_command;
233     check_tip_bit;
234     present_illegal_slave_address;
235     check_tip_bit;
236
237 endtask : check_slave_addr
238
239
240 task load_i2c_prescalar_values;
241         i2c_wb_master_bfm::wb_write(1,mif, PRER_LO, 8'hfa); //
                load prescaler lo-byte
242         i2c_wb_master_bfm::wb_write(1,mif, PRER_LO, 8'hc8); //
                load prescaler lo-byte
243         i2c_wb_master_bfm::wb_write(1,mif, PRER_HI, 8'h00); //
                load prescaler hi-byte
244 endtask : load_i2c_prescalar_values
245
246
```

```
247
248
249 task enable_master_core;
250                 // Enable master core
251         i2c_wb_master_bfm::wb_write(1,mif, CTR,      8'h80); //
                enable core
252         //`uvm_info(get_name(),$sformatf("status: %t Master
               core enabled",$time()),UVM_LOW)
253 endtask : enable_master_core
254
255
256
257
258 task write_data_to_slave;
259     i2c_data_transaction data_packet_to_dut ;
260
261
262     // sending/writing data to the slave
263       repeat(data_loop) begin
264           // send memory contents
265           i2c_wb_master_bfm::wb_write(1,mif, TXR,
                storing_data_sent[i]); // present data
266           set_write_command;
267     //  `uvm_info(get_name(),$sformatf("status: %t Writing
          random data = %0h",$time(),storing_data_sent[i]),
```

```
UVM_LOW)
        data_packet_to_dut = i2c_data_transaction::type_id::
            create("data_packet");

        //$cast(data_txn,data_packet);
        data_packet_to_dut.data_byte = storing_data_sent[i];
        data_packet_to_dut.addr = SADR;
    //   $display("data_packet = %0h",data_packet_to_dut.
         data_byte);
        data_to_bfm.write(data_packet_to_dut);
        i = i +1;
          check_tip_bit;
        end
        i = 0;


endtask : write_data_to_slave




task read_data_from_slave;
    i2c_data_transaction data_packet_from_dut;
```

```
290
291
292
293         //Comparing/reading the data sent earlier
294     repeat(data_loop) begin
295             i2c_wb_master_bfm::wb_write(1,mif, CR,        8'h20)
                    ; // set command (read, mif.wb_ack_o_read)
296             //`uvm_info(get_name(),$sformatf("status: %t read
                    + wishbone ack",$time()),UVM_LOW)
297       check_tip_bit;
298             // check data just received
299             i2c_wb_master_bfm::wb_read(1,mif, RXR,
                    data_read_from_wb);
300       data_packet_from_dut = i2c_data_transaction::type_id::
             create("data_packet1");
301       data_packet_from_dut.data_byte = data_read_from_wb;
302       data_packet_from_dut.addr = SADR;
303       data_from_bfm.write(data_packet_from_dut);
304       j = j + 1;
305     end
306       j = 0;
307 endtask : read_data_from_slave
308
309
310
```

```
311
312
313
314
315 task write_slave_address_to_wb(logic [7:0] slaveaddr);
316         i2c_wb_master_bfm::wb_write(1,mif, TXR, {slaveaddr,WR}
            ); // present slave address, set write-bit
317 endtask : write_slave_address_to_wb
318
319
320
321
322
323
324 task verify_i2c_prescalar_values;
325         // Verify the pre-scalar values using wb compare
            function
326         i2c_wb_master_bfm::wb_cmp(0,mif, PRER_LO, 8'hc8); //
            verify prescaler lo-byte
327         i2c_wb_master_bfm::wb_cmp(0,mif, PRER_HI, 8'h00); //
            verify prescaler hi-byte
328         //`uvm_info(get_name(),$sformatf("status: %t Verified
            prescalar values",$time()),UVM_LOW)
329 endtask : verify_i2c_prescalar_values
330
```

```
331
332
333
334
335 task set_write_command;
336         i2c_wb_master_bfm::wb_write(0,mif, CR,        8'h10); //
            set command (write)
337 endtask : set_write_command
338
339
340
341
342
343 task set_start_command;
344         i2c_wb_master_bfm::wb_write(0,mif, CR,        8'h90 );
            // set command (start, write)
345         //`uvm_info(get_name(),$sformatf("status: %t Setting
            start + write commands",$time()),UVM_LOW)
346 endtask : set_start_command
347
348
349
350 endclass: i2c_driver
```

I.5 Monitor

```
//
    ************************************************************************
//
//   Author      ::   Shravani Balaraju
//   Date Created   ::   2019-01-19
//   Module name    ::   i2c_monitor
//
//
    ************************************************************************

import "DPI" function void start_time();
import "DPI" function void end_time();



class i2c_monitor extends uvm_monitor ;
  `uvm_component_utils(i2c_monitor)

//-----------------------------------------------
//      Class variables
//-----------------------------------------------
```

```
20    virtual i2c_master_interface mif;
21  //------------------------------------------------------------
22  //      Analysis ports
23  //------------------------------------------------------------
24
25  //------------------------------------------------------------
26  //      Covergroups
27  //------------------------------------------------------------
28  covergroup cover_master_signals @(mif.wb_adr_i,mif.wb_dat_o,
      mif.wb_dat_i,mif.wb_we_i,mif.wb_stb_i,mif.wb_cyc_i,mif.
      wb_ack_o,mif.scl,mif.sda);
29    adr_cov : coverpoint mif.wb_adr_i
30    {
31      bins prescalar_values = {0,1};
32      bins control_reg = {'b10};
33      bins transmit_receive_reg = {'b11};
34      bins status_reg = {'b100};
35      ignore_bins reserved_regs = {'b101,'b110};
36    }
37    we_cov : coverpoint mif.wb_we_i;
38    stb_cov : coverpoint mif.wb_stb_i;
39    cyc_cov : coverpoint mif.wb_cyc_i;
40    ack_cov: coverpoint mif.wb_ack_o;
41    scl_cov : coverpoint mif.scl;
42    sda_cov : coverpoint mif.sda;
```

```
endgroup
//-----------------------------------------------
//      new function
//-----------------------------------------------
  function new(string name, uvm_component parent);
    super.new(name, parent);
    cover_master_signals = new();
  endfunction: new

//-----------------------------------------------
//      UVM Phases
//-----------------------------------------------

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    void'(uvm_resource_db#(virtual i2c_master_interface)::
        read_by_name
      (.scope("ifs"), .name("i2c_master_interface"), .val(mif)
          ));
  endfunction: build_phase


  task run_phase(uvm_phase phase);

  endtask: run_phase
```

```
66
67    function void start_of_simulation_phase(uvm_phase phase);
68      start_time();
69    endfunction : start_of_simulation_phase
70
71    function void final_phase(uvm_phase phase);
72      end_time();
73
74    endfunction : final_phase
75
76  //-----------------------------------------------------
77  //      Tasks/Functions
78  //-----------------------------------------------------
79
80
81
82  endclass: i2c_monitor
```

I.6 Scoreboard

```
//
    ************************************************************
//
//   Author      ::   Shravani Balaraju
//   Date Created   ::   2019-03-19
//   Module name    ::   i2c_scoreboard
//
//
    ************************************************************

class i2c_scoreboard extends uvm_scoreboard ;
  `uvm_component_utils(i2c_scoreboard)

//-----------------------------------------------
//     Class variables
//-----------------------------------------------
  i2c_data_transaction expected[$],actual[$];
  logic pass;
  i2c_data_transaction ac;
  i2c_data_transaction ex;
  int i;
```

```
20    int a;
21    int e;
22    int p;
23
24  //-----------------------------------------------
25  //      Analysis ports
26  //-----------------------------------------------
27
28  `uvm_analysis_imp_decl(_expected_pkt)
29  `uvm_analysis_imp_decl(_actual_pkt)
30
31  uvm_analysis_imp_expected_pkt #(i2c_data_transaction,
      i2c_scoreboard) pkt_written_to_core;
32  uvm_analysis_imp_actual_pkt #(i2c_data_transaction,
      i2c_scoreboard) pkt_read_from_core;
33  //-----------------------------------------------
34  //      Covergroups
35  //-----------------------------------------------
36
37
38  covergroup cross_exp_act ;
39    exp_data: coverpoint ex.data_byte;
40    ac_data: coverpoint ac.data_byte;
41    cross_data: cross exp_data, ac_data
42    {
```

```
    ignore_bins ignore_when_values_unequal = cross_data with (
        exp_data != ac_data);
  }
endgroup

//-----------------------------------------------------
//     new function
//-----------------------------------------------------
  function new(string name, uvm_component parent);
    super.new(name, parent);
    i = 0;
    e = 0;
    a = 0;
    p = 0;
    pass = 1;
    cross_exp_act = new();
    pkt_written_to_core = new("pkt_written_to_core",this);
    pkt_read_from_core = new("pkt_read_from_core",this);
    ex = i2c_data_transaction::type_id::create("ex_data");
    ac = i2c_data_transaction::type_id::create("ac_data");
  endfunction: new

//-----------------------------------------------------
//     UVM Phases
//-----------------------------------------------------
```

```
67
68    function void build_phase(uvm_phase phase);
69      super.build_phase(phase);
70    endfunction: build_phase
71
72    task run_phase(uvm_phase phase);
73      //$display("\n\n \t\tTest running ...");
74       forever begin
75        wait(expected.size() != 0 && actual.size() != 0);
76        ex = expected.pop_front();
77        ac = actual.pop_front();
78        if(ex.addr == ac.addr)
79        compare_data(ex.data_byte,ac.data_byte);
80        else begin
81          expected.push_front(ex);
82          actual.push_front(ac);
83      end
84      end
85    endtask: run_phase
86
87    function void final_phase(uvm_phase phase);
88      if(pass == 1 && p != 0)
89        $display("\n\n \t\tTest passed! \n\n");
90      else begin
```

```
91          $display("\n\n \t\tTest failed with %0d data mismatches
               \n\n",i);
92        end
93      endfunction : final_phase
94
95  //-----------------------------------------------------
96  //      Tasks/Functions
97  //-----------------------------------------------------
98
99  function void write_expected_pkt(i2c_data_transaction exp_pkt)
       ;
100     //$display("Received write data = %0h, addr = %0b",exp_pkt.
          data_byte,exp_pkt.addr);
101     expected.push_back(exp_pkt);
102     e = e + 1;
103 endfunction : write_expected_pkt
104
105 function void write_actual_pkt(i2c_data_transaction act_pkt);
106     //$display("Received read data = %0h, addr = %0b",act_pkt.
          data_byte,act_pkt.addr);
107     actual.push_back(act_pkt);
108     a = a + 1;
109 endfunction : write_actual_pkt
110
111 function void compare_data(logic [7:0] ex, logic [7:0] ac);
```

```
int d = 0;
  if(ex == ac)
  begin
    `uvm_info(get_name(),$sformatf("DATA MATCH:: Expected data
        matches received data %0h",ac),UVM_LOW)
    cross_exp_act.sample();
    p = p + 1;
  end
  else if(ex != 'bx && ac != 'bx)
  begin
    `uvm_warning(get_name(),$sformatf("DATA MISMATCH::
       Expected data matches = %0h, received data %0h",ex,ac))
    cross_exp_act.sample();
    pass = 0;
    i = i + 1;
  end
  else
    d = 0;
    //do nothing


endfunction : compare_data

endclass: i2c_scoreboard
```

I.7 Wishbone BFM

```
//
    ************************************************************************
//
//   Author      ::   Shravani Balaraju
//   Date Created   ::   2019-02-03
//   Module name    ::   i2c_wb_master_bfm
//
//
    ************************************************************************
class i2c_wb_master_bfm extends uvm_component ;
  `uvm_component_utils(i2c_wb_master_bfm)

//-----------------------------------------------
//      Class variables
//-----------------------------------------------
virtual i2c_master_interface mif;

//-----------------------------------------------
//      Analysis ports
//-----------------------------------------------

```

```
20 //-----------------------------------------------------
21 //      Covergroups
22 //-----------------------------------------------------
23
24 //-----------------------------------------------------
25 //      new function
26 //-----------------------------------------------------
27   function new(string name, uvm_component parent);
28     super.new(name, parent);
29
30   endfunction: new
31
32 //-----------------------------------------------------
33 //      UVM Phases
34 //-----------------------------------------------------
35
36   function void build_phase(uvm_phase phase);
37     super.build_phase(phase);
38     void'(uvm_resource_db#(virtual i2c_master_interface)::
         read_by_name
39       (.scope("ifs"), .name("i2c_master_interface"), .val(mif)
           ));
40   endfunction: build_phase
41
42   task run_phase(uvm_phase phase);
```

```
43
44    endtask: run_phase
45
46
47
48  //-----------------------------------------------------
49  //      Tasks/Functions
50  //-----------------------------------------------------
51
52  static task wb_reset;
53    input virtual interface i2c_master_interface mif;
54    mif.wb_adr_i  = 1'bx;
55    mif.wb_dat_o  = 1'bx;
56    mif.wb_cyc_i  = 1'b0;
57    mif.wb_stb_i  = 1'bx;
58    mif.wb_we_i   = 1'bx;
59    #1;
60
61
62  endtask
63
64  static task wb_write;
65    input integer delay;
66    input virtual interface i2c_master_interface mif;
67    input logic[7:0] addr;
```

```
68    input logic[7:0] data;
69      repeat(delay) @(posedge mif.clk);
70    #1;
71    begin
72    mif.wb_adr_i  = addr;
73    mif.wb_dat_o  = data;
74    mif.wb_cyc_i  = 1'b1;
75    mif.wb_stb_i  = 1'b1;
76    mif.wb_we_i   = 1'b1;
77 //   $display("Writing data %0h to address %0h",data,addr);
78   @(posedge mif.clk);
79    wait(mif.wb_ack_o == 1'b1)
80    repeat(2) @(posedge mif.clk);
81    mif.wb_cyc_i  = 1'b0;
82    mif.wb_stb_i  = 1'b0;
83    mif.wb_adr_i  = 'bz;
84    mif.wb_dat_o  = 'bz;
85    mif.wb_we_i = 1'h0;
86    end
87 endtask : wb_write
88
89 static task wb_read;
90 input integer delay;
91    input virtual interface i2c_master_interface mif;
92    input logic[7:0] addr;
```

```
93    output logic[7:0] data;
94      repeat(delay) @(posedge mif.clk);
95
96    begin
97    repeat(delay) @(posedge mif.clk);
98    #1;
99    mif.wb_adr_i  = addr;
100   mif.wb_dat_o  = 1'bz;
101   mif.wb_cyc_i  = 1'b1;
102   mif.wb_stb_i  = 1'b1;
103   mif.wb_we_i   = 1'b0;
104 //   $display("Q from read is %0h",data);
105
106   wait(mif.wb_ack_o == 1'b1)
107
108   repeat(2) @(posedge mif.clk)
109   mif.wb_cyc_i  = 1'b0;
110   mif.wb_stb_i  = 1'b0;
111   mif.wb_adr_i  = 'bz;
112   data     = mif.wb_dat_i;
113   mif.wb_we_i = 1'b0;
114 //   $display("reading data %0h from address %0h",data,addr);
115   end
116 endtask : wb_read
117
```

```
static task wb_cmp;
  input integer delay;
  input virtual interface i2c_master_interface mif;
  input [31:0]  addr;
  input [31:0]  data;
  logic [31:0] q;

  begin
  //  $display("Reading q");
  //  wb_read (delay,mif, addr, q);
  //  $display("Value of q  =%h",q);

    if (data !== q)
      $display("Data compare error. Received %h, expected %h
          at time %t", q, data, $time);
  end
endtask
endclass: i2c_wb_master_bfm
```

I.8 Agent

```
1  //
   ************************************************************************
2  //
3  //   Author      ::   Shravani Balaraju
4  //   Date Created   ::   2019-02-05
5  //   Module name    ::   i2c_agent
6  //
7  //
   ************************************************************************
8
9
10 class i2c_agent extends uvm_agent ;
11   `uvm_component_utils(i2c_agent)
12
13 //-----------------------------------------------
14 //      Class members
15 //-----------------------------------------------
16 i2c_data_sequencer i2c_sqr;
17 i2c_driver i2c_dv;
18 i2c_monitor i2c_mon;
19
```

```
20 //i2c_base_slave base_slave;
21
22
23 //------------------------------------------------
24 //      Analysis ports
25 //------------------------------------------------
26
27 //------------------------------------------------
28 //      Covergroups
29 //------------------------------------------------
30
31 //------------------------------------------------
32 //      new function
33 //------------------------------------------------
34   function new(string name, uvm_component parent);
35     super.new(name, parent);
36
37   endfunction: new
38
39
40
41 //------------------------------------------------
42 //      UVM Phases
43 //------------------------------------------------
44
```

```
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
  i2c_sqr = i2c_data_sequencer::type_id::create(.name("i2c_sqr"),.parent(this));
  i2c_dv = i2c_driver::type_id::create(.name("i2c_dv"),.parent(this));
//  base_slave = i2c_base_slave::type_id::create(.name("base_slave"),.parent(this));
  i2c_mon = i2c_monitor::type_id::create(.name("i2c_mon"),.parent(this));


  endfunction: build_phase

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);

    i2c_dv.seq_item_port.connect(i2c_sqr.seq_item_export);
  endfunction : connect_phase



//------------------------------------------------------------
//      Tasks/Functions
//------------------------------------------------------------
```

```
endclass: i2c_agent
```

I.9 Environment

```
//
    ************************************************************************
//
//   Author      ::   Shravani Balaraju
//   Date Created   ::   2019-01-19
//   Module name    ::   i2c_env
//
//
    ************************************************************************


class i2c_env extends uvm_env ;
  `uvm_component_utils(i2c_env)

//-----------------------------------------------
//     Class variables
//-----------------------------------------------
  i2c_agent agent;
i2c_scoreboard i2c_sb;
//-----------------------------------------------
//     Analysis ports
```

```
//-----------------------------------------------------------

//-----------------------------------------------------------
//      Covergroups
//-----------------------------------------------------------

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

  endfunction: new

//-----------------------------------------------------------
//      UVM Phases
//-----------------------------------------------------------

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    agent = i2c_agent::type_id::create(.name("agent"),.parent(
        this));
  i2c_sb = i2c_scoreboard::type_id::create(.name("i2c_sb"),.
      parent(this));
  endfunction: build_phase
```

```
43
44    task run_phase(uvm_phase phase);
45
46    endtask: run_phase
47
48
49    function void connect_phase(uvm_phase phase);
50      super.connect_phase(phase);
51      agent.i2c_dv.data_to_bfm.connect(i2c_sb.
          pkt_written_to_core);
52      agent.i2c_dv.data_from_bfm.connect(i2c_sb.
          pkt_read_from_core);
53
54    endfunction : connect_phase
55
56  //-----------------------------------------------
57  //      Tasks/Functions
58  //-----------------------------------------------
59
60
61
62  endclass: i2c_env
```

I.10 Test

```
//
    ************************************************************
//
//   Author      ::   Shravani Balaraju
//   Date Created   ::   2019-02-06
//   Module name    ::   i2c_test
//
//
    ************************************************************


class i2c_master_top_test extends uvm_test ;
  `uvm_component_utils(i2c_master_top_test)

//-----------------------------------------------
//      Class variables
//-----------------------------------------------
  i2c_env menv;
//-----------------------------------------------
//      Analysis ports
//-----------------------------------------------
```

```

//-----------------------------------------------
//      Covergroups
//-----------------------------------------------

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

//-----------------------------------------------
//      UVM Phases
//-----------------------------------------------

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    menv = i2c_env::type_id::create(.name("menv"),.parent(this
        ));
  endfunction: build_phase

  task run_phase(uvm_phase phase);
    i2c_data_sequence data_seq;
    phase.raise_objection(.obj(this));
```

```
44      data_seq = i2c_data_sequence::type_id::create(.name("
            data_seq"),.contxt(get_full_name()));
45      assert(data_seq.randomize());
46      data_seq.start(menv.agent.i2c_sqr);
47      phase.drop_objection(.obj(this));
48    endtask: run_phase
49
50
51
52  //-----------------------------------------------
53  //      Tasks/Functions
54  //-----------------------------------------------
55
56
57
58  endclass
```

I.11 Slave

```
//
    ************************************************************
//
//      File: i2c_slave_model.sv
//
//
    ************************************************************


module i2c_slave_model (scl, sda);

  //
  // parameters
  //
  parameter I2C_ADR = 7'b000_0000;

  //
  // input && outpus
  //
  input scl;
  inout sda;
```

```
20
21  //
22  // Variable declaration
23  //
24  wire debug = 1'b1;
25
26  reg [7:0] mem [3:0];
27  reg [7:0] memory_address;
28  reg [7:0] memory_data_out;
29
30  reg sta, detect_start;
31  reg sto, detect_stop;
32
33  reg [7:0] shift_reg;
34  reg       read_write;
35
36  wire      my_adr;
37  wire      i2c_reset;
38  reg [2:0] bit_cnt;
39  wire      acc_done;
40  reg       load_count;
41
42  reg       sda_o;
43  wire      sda_dly;   // delayed version of sda
44
```

```
45  // statemachine declaration
46  parameter idle        = 3'b000;
47  parameter slave_ack   = 3'b001;
48  parameter get_mem_adr = 3'b010;
49  parameter gma_ack     = 3'b011;
50  parameter data        = 3'b100;
51  parameter data_ack    = 3'b101;
52
53  reg [2:0] state;
54
55  //
56  // module body
57  //
58
59  initial
60  begin
61     sda_o = 1'b1;
62     state = idle;
63  end
64
65  // generate shift register
66  always @(posedge scl) begin
67    shift_reg <=   {shift_reg[6:0],sda};
68  end
69  //detect my_address
```

```
assign my_adr = (shift_reg[7:1] == I2C_ADR);
always @(my_adr) begin
end

//generate bit-counter
always @(posedge scl)
  if(load_count)
    bit_cnt <=  3'b111;
  else
    bit_cnt <=  bit_cnt - 3'h1;

//generate access done signal
assign acc_done = !(|bit_cnt);

assign   sda_dly = sda;


//detect start condition
always @(negedge sda)
  if(scl)
    begin
        sta   <=  1'b1;
  detect_start <=  1'b0;
  sto   <=  1'b0;

```

```
95            if(debug)
96              $display("DEBUG i2c_slave; start condition
                  detected at %t", $time);
97        end
98      else
99        sta <=   1'b0;
100
101   always @(posedge scl)
102     detect_start <=   sta;
103
104   // detect stop condition
105   always @(posedge sda)
106     if(scl)
107       begin
108           sta <=   1'b0;
109           sto <=   1'b1;
110
111           if(debug)
112             $display("DEBUG i2c_slave; stop condition detected
                  at %t", $time);
113       end
114     else
115       sto <=   1'b0;
116
117   //generate i2c_reset signal
```

```
assign i2c_reset = sta || sto;

// generate statemachine
always @(negedge scl or posedge sto)
  if (sto || (sta && !detect_start) )
    begin
        state <=   idle; // reset statemachine

        sda_o <=   1'b1;
        load_count    <=   1'b1;
    end
  else
    begin
        // initial settings
        sda_o <=   1'b1;
        load_count    <=   1'b0;

        case(state) // synopsys full_case parallel_case
            idle: // idle state
              if (acc_done && my_adr)
                begin
                    state <=   slave_ack;
                    read_write <=   shift_reg[0];
                    sda_o <=   1'b0; // generate i2c_ack

```

```
143                     #2;
144                     if(debug && read_write)
145                       $display("DEBUG i2c_slave; command
                             byte received (read) at %t", $time)
                             ;
146                     if(debug && !read_write)
147                       $display("DEBUG i2c_slave; command
                             byte received (write) at %t", $time
                             );
148
149                     if(read_write)
150                       begin
151                           memory_data_out <=   mem[
                                 memory_address];
152
153                           if(debug)
154                             begin
155                                 #2 $display("DEBUG i2c_slave
                                       ; data block read %x from
                                        address %x (1)",
                                       memory_data_out,
                                       memory_address);
156                                 #2 $display("DEBUG i2c_slave
                                       ; memcheck [0]=%x, [1]=%x
                                       , [2]=%x", mem[4'h0], mem
```

```
                                        [4'h1], mem[4'h2]);
                                end
                            end
                        end

                slave_ack:
                  begin
                      if(read_write)
                        begin
                            state <=   data;
                            sda_o <=   memory_data_out[7];
                        end
                      else
                        state <=   get_mem_adr;

                      load_count    <=   1'b1;
                  end

                get_mem_adr: // wait for memory address
                  if(acc_done)
                    begin
                        state <=   gma_ack;
                        memory_address <=   shift_reg;
                        sda_o <=   !(shift_reg <= 15);

```

```
                    if(debug)
                        $display("DEBUG i2c_slave; address received. adr=%x, ack=%b", shift_reg, sda_o);
                end

            gma_ack:
              begin
                  state <=  data;
                  load_count  <=  1'b1;
              end

            data: // receive or drive data
              begin
                  if(read_write)
                    sda_o <=  memory_data_out[7];

                  if(acc_done)
                    begin
                        state <=  data_ack;
                        memory_address <= #2 memory_address + 8'h1;
                        sda_o <=  (read_write && (memory_address <= 15) );

```

```
202                     if(read_write)
203                       begin
204                           #3 memory_data_out <= mem[
                                 memory_address];
205
206                           if(debug)
207                             #5 $display("DEBUG i2c_slave
                                   ; data block read %x from
                                    address %x (2)",
                                   memory_data_out,
                                   memory_address);
208                       end
209
210                     if(!read_write)
211                       begin
212                           mem[ memory_address[3:0] ] <=
                                   shift_reg; // store data
                                 in memory
213
214                           if(debug)
215                             #2 $display("DEBUG i2c_slave
                                   ; data block write %x to
                                   address %x", shift_reg,
                                   memory_address);
216                       end
```

```
217                     end
218                 end
219
220             data_ack:
221                 begin
222                     load_count <=   1'b1;
223
224                     if(read_write)
225                         if(shift_reg[0])
226                             begin
227                                 state <=   idle;
228                                 sda_o <=   1'b1;
229                             end
230                         else
231                             begin
232                                 state <=   data;
233                                 sda_o <=   memory_data_out[7];
234                             end
235                     else
236                         begin
237                             state <=   data;
238                             sda_o <=   1'b1;
239                         end
240                 end
241
```

```
242             endcase
243         end
244
245     // read data from memory
246     always @(posedge scl)
247       if(!acc_done && read_write)
248         memory_data_out <=   {memory_data_out[6:0], 1'b1};
249
250     // generate tri-states
251     assign sda = sda_o ? 1'bz : 1'b0;
252
253
254
255
256 endmodule
```

I.12 Top

```
1  `include "uvm_macros.svh"
2  `include "i2c_pkg.sv"
3  `include "i2c_master_interface.sv"
4
5  module test;
6    import uvm_pkg::*;
7    import i2c_pkg::*;
8
9    //Interface declarations
10   i2c_master_interface mif();
11   parameter SADR     = 7'b0010_000;
12   parameter SADR1    = 7'b0000_001;
13   parameter SADR2    = 7'b0000_010;
14
15     // hookup wishbone_i2c_master core
16   i2c_master_top top (
17     .reset(mif.reset),
18            .clk(mif.clk),
19            .scan_in0(mif.scan_in0),
20            .scan_en(mif.scan_en),
21            .test_mode(mif.test_mode),
22            .scan_out0(mif.scan_out0),
23     .wb_clk_i(mif.clk),
```

```
24        .wb_rst_i(1'b0),
25        .arst_i(mif.rstn),
26        .wb_adr_i(mif.wb_adr_i[2:0]),
27        .wb_dat_i(mif.wb_dat_o),
28        .wb_dat_o(mif.wb_dat_i),
29        .wb_we_i(mif.wb_we_i),
30        .wb_stb_i(mif.wb_stb_i),
31        .wb_cyc_i(mif.wb_cyc_i),
32        .wb_ack_o(mif.wb_ack_o),
33        .wb_inta_o(mif.wb_inta_o),
34        .scl_pad_i(mif.scl),
35        .scl_pad_o(mif.scl0_o),
36        .scl_padoen_o(mif.scl0_oen),
37        .sda_pad_i(mif.sda),
38        .sda_pad_o(mif.sda0_o),
39        .sda_padoen_o(mif.sda0_oen)
40    );
41
42
43
44    // hookup i2c slave model
45    i2c_slave_model #(SADR) i2c_slave (
46        .scl(mif.scl),
47        .sda(mif.sda)
48    );
```

```
49
50    i2c_slave_model #(SADR1) i2c_slave1 (
51      .scl(mif.scl),
52      .sda(mif.sda)
53    );
54
55    i2c_slave_model #(SADR2) i2c_slave2 (
56      .scl(mif.scl),
57      .sda(mif.sda)
58    );
59
60          // create i2c lines
61    delay m0_scl (mif.scl0_oen ? 1'bz : mif.scl0_o, mif.scl),
62          m0_sda (mif.sda0_oen ? 1'bz : mif.sda0_o, mif.sda);
63    pullup p1(mif.scl); // pullup mif.scl line
64    pullup p2(mif.sda); // pullup mif.sda line
65
66 initial
67 begin
68      $timeformat(-9,2,"ns", 16);
69 `ifdef SDFSCAN
70      $sdf_annotate("sdf/i2c_master_top_tsmc18_scan.sdf", test.
          top);
71 `endif
72      void'(uvm_resource_db#(virtual i2c_master_interface)::set
```

```
73        (.scope("ifs"), .name("i2c_master_interface"), .val(mif)
            ));
74          force i2c_slave.debug = 1'b0; // disable i2c_slave
              debug information
75          force i2c_slave1.debug = 1'b0;
76          force i2c_slave2.debug = 1'b0;
77          $set_coverage_db_name("i2c_master_top");
78          run_test();
79
80 end
81
82
83 property start_condition;
84 @(posedge mif.clk) mif.scl |-> ##[1:$] !mif.sda;
85 endproperty
86
87 assert property (start_condition);
88
89 cover property (start_condition);
90
91 property stop_condition;
92 @(posedge mif.clk) !mif.scl |-> ##[1:$] $fell(mif.sda);
93 endproperty
94
95 assert property (stop_condition);
```

```
96
97  cover property (stop_condition);
98
99  property data_transfer_condition;
100 @(posedge mif.clk) mif.scl |-> ##[1:$] $rose(mif.sda) ##[1:$]
        $fell(mif.sda) ##[1:$] !mif.scl;
101 endproperty
102
103 assert property (data_transfer_condition);
104
105 cover property (data_transfer_condition);
106
107
108 always #100 mif.clk = ~mif.clk;
109 endmodule
110
111 module delay (in, out);
112   input  in;
113   output out;
114
115   assign out = in;
116
117   specify
118     (in => out) = (600,600);
119   endspecify
```

```
120 endmodule
```

I.13 UVM component generator script

```
#!/bin/csh -f
#
# Example: /uvm_component_generator.sh <class_name> <uvm_component_name>

setenv UVM_FILE "i2c_$1.sv"

#copy template to a new file
cp uvm_template.sv $UVM_FILE

echo $UVM_FILE
#sed command to change the class name
sed "s/--classname--/i2c_$1/g" $UVM_FILE | tee $UVM_FILE.1
sed "s/--uvmobject--/$2/g" $UVM_FILE.1 | tee $UVM_FILE.2
setenv DATE `date +%Y-%m-%d`
sed "s/--date--/$DATE/g" $UVM_FILE.2 | tee $UVM_FILE

/bin/rm -f $UVM_FILE.1 $UVM_FILE.2

if ($3 == "c") then
  mv $UVM_FILE ../src/
endif
```

I.14 UVM Template

```
//
    ************************************************************************
//
//   Author      ::   Shravani Balaraju
//   Date Created   ::   --date--
//   Module name    ::   --classname--
//
//
    ************************************************************************


class --classname-- extends --uvmobject-- ;
  `uvm_component_utils(--classname--)

//-----------------------------------------------
//      Class variables
//-----------------------------------------------

//-----------------------------------------------
//      Analysis ports
//-----------------------------------------------
```

```
20
21 //-----------------------------------------------------------
22 //      Covergroups
23 //-----------------------------------------------------------
24
25 //-----------------------------------------------------------
26 //      new function
27 //-----------------------------------------------------------
28   function new(string name, uvm_component parent);
29     super.new(name, parent);
30
31   endfunction: new
32
33 //-----------------------------------------------------------
34 //      UVM Phases
35 //-----------------------------------------------------------
36
37   function void build_phase(uvm_phase phase);
38     super.build_phase(phase);
39
40   endfunction: build_phase
41
42   task run_phase(uvm_phase phase);
43
44   endtask: run_phase
```

```
45
46
47
48 //-----------------------------------------------------------
49 //      Tasks/Functions
50 //-----------------------------------------------------------
51
52
53
54 endclass: --classname--
```

Appendix II

Wavedrom Help Guide

Wavedrom is a JavaScript application that helps to describe digital timing diagrams. Table.II.1 shows a list of common notations and the result of that notation on the final waveform. The description of the waveform is written in terms of simple combinations of the notations outlined in the Table. A simple example for drawing a waveform along with the syntax is given below:

```
{ "signal" : [

  { "name": "SCL", "wave": "0.1.0." },
  { "name": "SDA", "wave": "x=...x",data: [ "DATA", ""] }],

 config: { hscale: 3 }

            }
```

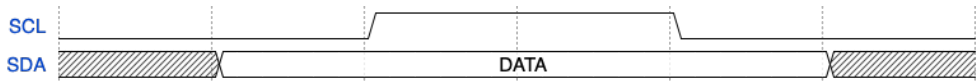
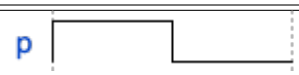
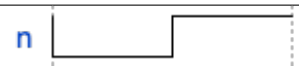
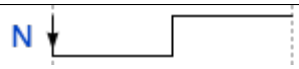
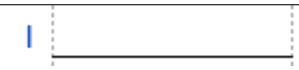
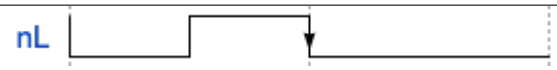
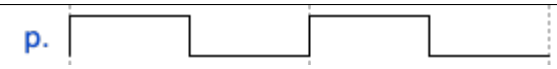
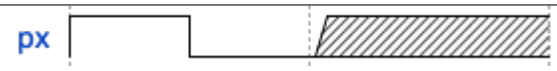
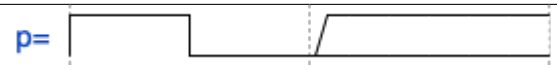


Figure II.1: Example Waveform

Table II.1: Wavedrom Instructions and Waves

Notation	Waveform
p	
P	
n	
N	
h	
H	
l	
L	
’.’	
’x’	
’=’	

The above code gives the following resulting waveform.