5-2018

UVM Verification of an SPI Master Core

Deepak Siddharth Parthipan
dp9040@rit.edu

UVM VERIFICATION OF AN SPI MASTER CORE

by

Deepak Siddharth Parthipan

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, 2018

I would like to dedicate this work to my family, my father Parthipan Kempanna Gowder, my mother Malarmathy Parthipan, my sister Vaishnavi Parthipan, and friends for their love and support during my thesis.

Declaration

I hereby state that except where explicit references are made to the work of others, that all work and contents of this Graduate Paper are original and have not been submitted in part or whole for consideration for any other qualification in this, or any other University. This UVM Verification of an SPI Master Core Graduate Paper is the result of my work and not a collaborative work, except where explicit references are mentioned.

Deepak Siddharth Parthipan

May, 2018

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

In today's world, more and more functionalities in the form of IP cores are integrated into a single chip or SOC. System-level verification of such large SOCs has become complex. The modern trend is to provide pre-designed IP cores with companion Verification IP. These Verification IPs are independent, scalable, and reusable verification components. The SystemVerilog language is based on object-oriented principles and is the most promising language to develop a complete verification environment with functional coverage, constrained random testing and assertions. The Universal Verification Methodology, written in SystemVerilog, is a base class library of reusable verification components. This paper discusses a Universal Verification Methodology based environment for testing a Wishbone compliant SPI master controller core. A multi-layer testbench was developed which consists of a Wishbone bus functional model, SPI slave model, driver, scoreboard, coverage analysis, and assertions developed using various properties of SystemVerilog an the UVM library. Later, constrained random testing using vectors driven into the DUT for higher functional coverage is discussed. The verification results shows the effectiveness and feasibility of the proposed verification environment.

Contents

Contents v

List of Figures xi

List of Tables xii

1 Introduction 1

1.1 Research Goals 2

1.2 Contributions 3

1.3 Organization 3

2 Bibliographical Research 5

3 System Verification 8

3.1 State of the art 8

3.2 UVM Overview 9

3.3 UVM Class Hierarchy 9

3.3.1 UVM Testbench Top 10

3.3.2 UVM Test 11

3.3.3 UVM Environment 11

3.3.4 UVM Agent 11
3.3.5 UVM Sequence Item 12
3.3.6 UVM Sequence 12
3.3.7 UVM Driver 13
3.3.8 UVM Sequencer 13
3.3.9 UVM Monitor 13
3.3.10 UVM Scoreboard 14
3.4 UVM Transaction Level Communication Protocol 14
3.4.1 Basic Transaction Level Communication 14
3.4.2 Analysis ports and Exports 15
3.5 UVM Phases 15
3.5.1 Build Phase 15
3.5.2 Connect Phase 16
3.5.3 End of Elaboration Phase 17
3.5.4 Start of Simulation Phase 17
3.5.5 Normal Run Phase 17
3.5.6 Scheduled Run Phase 17
3.5.6.1 Pre Reset Phase 17
3.5.6.2 Reset Phase 18
3.5.6.3 Post Reset Phase 18
3.5.6.4 Pre Configure Phase 18
3.5.6.5 Configure Phase 18
3.5.6.6 Post Configure Phase 18
3.5.6.7 Pre Main Phase 18
3.5.6.8 Main Phase 18

3.5.6.9 Post Main Phase 19
3.5.6.10 Pre Shutdown Phase 19
3.5.6.11 Shutdown Phase 19
3.5.6.12 Post Shutdown Phase 19
3.5.7 Extract Phase 19
3.5.8 Check Phase 19
3.5.9 Report Phase 20
3.5.10 Final Phase 20
3.6 UVM Macros 20

4 System Architecture **21**
4.1 WISHBONE Interface 21
4.2 WISHBONE I/O Registers 22
4.3 Serial Peripheral Interface 24
4.4 Data Transmission 25
4.5 Hardware Architecture 27
4.5.1 Design of Clock Generation module (spi_clk_gen) 27
4.5.2 Serial data transfer module design (spi_shift) 29
4.5.3 Top-level module (spi) 29
4.6 SPI Registers 29
4.6.1 RxX Register 29
4.6.2 TxX Register 30
4.6.3 ASS Register 30
4.6.4 DIVIDER Register 31
4.6.5 SS Register 31

4.6.6 IE Register 31
4.6.7 LSB Register 31
4.6.8 Tx_NEG Register 32
4.6.9 Rx_NEG Register 32
4.6.10 GO_BSY Register 32
4.6.11 CHAR_LEN Register 32
4.7 Limitation of Standard SPI and Advancements 33

5 Test Methodology and Results **34**
5.1 Testbench Components 34
5.1.1 Test top 34
5.1.2 spi_interface 35
5.1.3 spi_package 36
5.1.4 spi_test 36
5.1.5 spi_environment 36
5.1.6 spi_agent 36
5.1.7 spi_sequence_item 37
5.1.8 spi_sequence 37
5.1.9 spi_sequencer 37
5.1.10 spi_driver 38
5.1.11 spi_monitor 39
5.1.12 spi_scoreboard 39
5.1.13 wishbone_bfm 39
5.2 Testbench Results 40
5.2.1 SPI Master Controller Synthesis Benchmarking 40

5.2.2 Data Transactions 41
5.2.2.1 WISHBONE to SPI Master communication using BFM 41
5.2.2.2 SPI Master-Slave communication 41
5.2.3 Coverage 42
5.2.3.1 Code Coverage 43
5.2.3.2 Functional Coverage - Signal Level 44
5.2.3.3 Functional Coverage - Transaction Level 45

6 Conclusion **48**
6.1 Future Work 49

References **50**

I Source Code **54**
I.1 SPI Top 54
I.2 SPI Clock 69
I.3 SPI Shift 73
I.4 Defines 83
I.5 Test Top 90
I.6 Interface 96
I.7 Package 98
I.8 Test 100
I.9 Environment 103
I.10 Agent 105
I.11 Sequence Item 108
I.12 Sequence 110

I.13 Sequencer 113
I.14 Driver 114
I.15 Monitor 117
I.16 Wishbone Bus Funtion Model 120
I.17 Scoreboard 124
I.18 Coverage 131
I.19 SPI Slave Model 133
I.20 Test defines 141

List of Figures

3.1 UVM hierarchy 10

3.2 UVM Phases 16

4.1 Wishbone Interface 22

4.2 SPI Protocol 25

4.3 Shift Register 27

4.4 SPI Master Architecture 28

5.1 UVM Testbench model 35

5.2 UVM Sequencer Driver Communication 38

5.3 WISHBONE to SPI communication 42

5.4 SPI Master - Slave communication 43

5.5 Top Level Code Coverage 43

5.6 Clock Level Code Coverage 44

5.7 Shift Level Code Coverage 44

5.8 Signal Coverage 45

5.9 Transaction Coverage 46

List of Tables

4.1 WISHBONE I/O Ports 23

4.2 SPI Master core registers 30

5.1 Synthesis Report 40

5.2 Test Configuration 41

Chapter 1

Introduction

The rapid development of modern integrated circuits not only increased the complexity of integrated circuit (IC) design, but also made the IC verification equally challenging. Around 70% to 80% of the entire design cycle time is allotted to verification, and traditional verification methodologies are no longer able to support current verification requirements [1]. In 2002, the Accellera Systems Initiative released SystemVerilog (SV) a a unified hardware design and verification language. SystemVerilog language was an amalgamation of constructs from different languages such as Vera, Super Log, C, Verilog and VHDL languages. Moreover, in 2005 IEEE standardized (1800-2005) SystemVerilog. SystemVerilog supports behavioral, register transfer level, and gate level descriptions. SystemVerilog also supports testbench development by the inclusion of object-oriented constructs, cover groups, assertions, constrained random constructs, application specific interface to other languages [2].

Universal Verification Methodology (UVM) is a standardized verification methodology for testbench creation an is derived form the Open Verification Methodology (OVM), and also inherits some features from Verification Methodology Manual (VMM). Use of the UVM standard enables an increase in verification productivity by creating a reusable verification platform and verification components. The verification results of this work show the effectiveness and feasibility of the proposed verification environment [3]

System on Chip (SoC) is used for intelligent control feature with all the integrated components connected to each other in a single chip. To complete a full system, every SoC must be linked to other system components in an efficient way that allows a faster error-free communication. Data communication between core controller modules and other external devices like external EEPROMs, DACs, ADCs. is critical. Different forms of communication protocols exist such as high throughput protocols like Ethernet, USB, SATA, PCI-Express which are used for data exchanges between whole systems. The Serial Peripheral Interface (SPI) is often considered as light weight communication protocol. The primary purpose of the protocol is that it is suited for communication between integrated circuits for low and medium data transfer rates with onboard peripherals and the serial bus provides a significant cost advantage.

1.1 Research Goals

The goal of this research work is to build a effective test bench that validates the SPI master controller with the help of the WISHBONE bus function model and SPI slave model. The goal is achieved with the following objectives:

- To understand SPI protocol architecture and WISHBONE specific requirements, to establish a connection between the test bench components and core controller.
- To apply advanced verification techniques such as Universal Verification Methodology and Coverage Driven Functional Verification.
- To develop a reusable Verification IP for WISBONE compliant SPI master core.

1.2 Contributions

The major contributions if this work include:

1. Research the SPI sub-system architecture, the Universal Verification Methodology, and SystemVerilog.

2. Development of a WISBONE bus function model acting as an interface between the test bench and the SPI master device under test (DUT) and SPI slave model in order to make the verification closed loop testing.

3. Build hierarchical testbench components using UVM libraries and SystemVerilog constructs, constrained random stimulus, coverage and assertions.

4. Verify transmission of data with different character width and data formats.

1.3 Organization

The structure of the thesis is as follows:

- Chapter 2: This chapter consists majorly of articles/journals/books that are referred to provide a foundation for building a layered test bench. It also discusses some of the new methodologies and techniques for controller verification.

- Chapter 3: This chapter briefly describes the system verification, various components and methodology associated with it.

- Chapter 4: The system architecture, theory of operation, controller configuration registers of both WISHBONE and SPI described.

- Chapter 5: SPI test methodology, test bench components and bus function model are discussed in this chapter.
- Chapter 6: This chapter comprises of the verification results, conclusion and possible future work.

Chapter 2

Bibliographical Research

SPI protocol is one of the widely used serial protocols used in a SoC compared to other protocols such UART and I2C simply because SPI can operate in higher bandwidth and throughput [4]. SPI Protocol typically provides communication between the hosts side microcontroller and slave devices. It is widely used owing to fewer control signals to operate with [5]. At the host side, the specific SPI core studied in this work acts like a WISHBONE compliant slave device. The SPI master core controller consists of three main parts, Serial shift interface, clock generator and WISHBONE interface. The SPI core controller has five 32-bit registers which can be configured through the WISHBONE interface. The serial interface consists of slave select lines, serial clock lines, as well as input and output data lines. The data transfers are full duplex in nature and number of bits per transfer is programmable [6].

It is possible to have high speed SPI Master/Slave Implementation of range 900 – 1000 MHz. The core can be designed with greater ways to control SPI-bus such as the flexibility of handling two slaves at a time. One important feature is configured by programming the control register of the core through which the SPI module can be made to either operate in master or slave mode. During operation, the SPI status register gives information such as the current position of the data transfer operation, whether the data transfer has completed or not, etc. [7]. Another key feature is the flexibility of designing the SPI Interface IPs for multiple devices using parameterization method. Advanced design techniques, such as Time Sharing Multiplex (TSM), is used to automatically identify the master/slave devices and achieve multi-master devices. Using TSM the disadvantage of communication among multiple devices are overcome [8].

Owing to the increasing complexity of the modern SoC, the verification has become more challenging. In fact 70% of the product development time is spent on complex SoC verification. Reducing the verification effort is the key for time to market challenge. In order to cater to such growing complexity advanced verification methodologies are employed. IP verification requires in depth functional coverage with constraint random simulation technique. Various components such as coverage monitors and scoreboards are used for this purpose [9]. For a communication protocol like the SPI communication protocol, it has to be verified as per the design specifications. Applying constrained random technique for higher functional coverage provides effective verification result [10].

For many years, EDA vendors have been proposing newer verification methodologies and languages. For any system level verification methodology and language to be successful, the key lies in the scalability and reusability of the verification components developed. SystemVerilog with object-oriented programming is considered as one of the most promising techniques for high level function verification for current complex SOC designs. SystemVerilog provide complete verification environment, with direct and constrained random generation, assertion based verification and coverage driven metrics [11].

The Universal Verification Methodology (UVM) is the latest functional verification methodology, it uses base class libraries coded in SystemVerilog. UVM is built upon previous methodology libraries such as Mentor's AVM, Mentor & Cadence's OVM, Verisity's eRM, and Synopsys's VMM-RAL. This standardization allows users to implement verification modules that are portable and highly compatible. Such modules are called as Verification components. They are encapsulated and made ready to use configurable verification environments for full systems, submodules, or protocols. The comprehensive base class library forms the foundation for such applications. It is simulation-oriented, and performs coverage-driven constrained random verification, assertion-based verification, hardware acceleration or emulation [12].

Pre-designed and pre-verified is the corner stone of any new modern SoC development. IP blocks developed are reusable in nature and for most blocks one or more bus protocols play a very important role to make these IPs to adapt to a plug and play concept thereby increasing the productivity with a reduction in design time. The WISHBONE System on Chip interconnection is a method to connect different IP cores to form integrated circuits. The core objective behind the WISHBONE bus is to create a standard, portable interface that supports both ASIC and FPGA and technology independent [13]. The SPI protocol is developed using other bus protocols such as On-Chip Peripheral Bus [14]. A Bus Function Model (BFM) is use to verify IPs that are compatible with bus protocol such as the WISHBONE bus. The need for such models is to create a standalone interface that can receive transaction from the test bench from one side and on the other side operate as a master device on the bus an behave and send commands to the device under test [15].

Chapter 3

System Verification

3.1 State of the art

Hardware description languages are tools used by engineers to specify abstract models of digital circuits to translate them into real hardware, as the design progresses towards completion, hardware verification is performed using Hardware verification languages like SystemVerilog. The purpose of verification is to demonstrate the functional correctness of a design. Verification is achieved by means of a testbench, which is an abstract system that provides stimulus to the inputs of design under test (DUT). Functional verification shows that design implementation is in correspondence to the specification. Typically, the testbench implements a reference model of the functionality that needs to be verified and compare the results from that model with the results of the design under test. The role of functional verification is to verify if the design meets the specification but not to prove it [16].

The traditional approach to functional verification relies on directed tests. Verification engineers conceive and apply a series of critical stimulus directly to the device under test, and check if the result is the expected one. This approach produces quick initial results because little effort is required for setting up the verification infrastructure. But as design complexity grows, it becomes a tedious and time-consuming task to write all the tests needed to cover 100% of the design. Random stimuli help to cover the unlikely cases and expose the bugs. However, in order to use random stimuli, the test environment requires automating process to generate random stimulus, there is a need of a block that predicts, keeps track of result and analyses them: a scoreboard. Additionally, functional coverage is a process used, to check what cases of the random stimulus were covered and what states of the design have been reached. This kind of testbench may require a longer time to develop, however, random based testing can actually promote the verification of the design by covering cases not achieved with directed tests [16].

3.2 UVM Overview

The UVM methodology is as a portable, open-source library from the Accellera Systems Initiative, and it should be compatible with any HDL simulator that supports SystemVerilog. UVM is also based on the OVM library which provides some background and maturity to the methodology. A key feature of UVM includes re-usability though the UVM API and guidelines for a standard verification environment. The environment is easily modifiable and understood by any verification engineer that understands the methodology behind it [17].

3.3 UVM Class Hierarchy

Figure 3.1 shows a simple UVM testbench class hierarchy. The following UVM components make up the hierarchy.

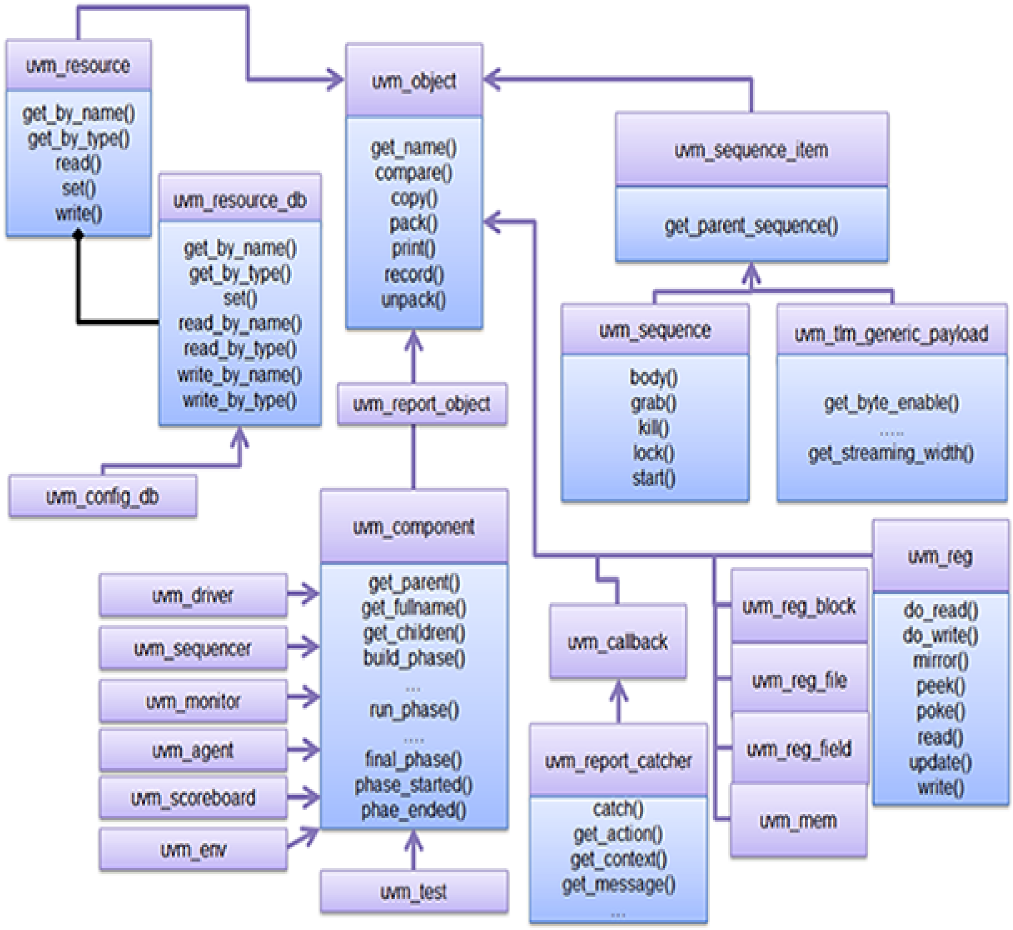

Figure 3.1: UVM hierarchy

3.3.1 UVM Testbench Top

The UVM testbench typically includes one or more instantiations design under test modules and interfaces which connect the DUT with the testbench. Transaction Level Modeling (TLM) interfaces in UVM provide communication methods for sending and receiving transactions between components. A UVM Test is dynamically instantiated at run-time, allowing the UVM testbench to be compiled once and run with many different tests [18].

3.3.2 UVM Test

The UVM test is the top-level UVM component class under UVM testbench. The UVM Test typically performs keys tasks like: configures values in config class and apply appropriate stimulus by invoking UVM sequences through the environment to the DUT. Base test class instantiates and configure the top-level environment; further individual tests will extend the base test to define scenario-specific environment configurations such as which sequences to run, coverage parameters, etc [18].

3.3.3 UVM Environment

The UVM environment is a container component class that groups together interrelated UVM verification components such as scoreboards, agents or even other environments. The top-level environment is a reusable component that encapsulates all the lower level verification components are targeting the DUT. There can be multiple tests that can instantiate the top-level environment class to generate and send different traffic for the selected configuration. UVM Test can override the default configuration of the top-level environment. Master UVM environment can also instantiate other child environments. Each interface to the DUT can have the separate environment. For example, UVM would be used to create reusable interface environments such as PCIe environment, USB environment, cluster environments, e.g., a CPU environment, IP interface environment, etc [18].

3.3.4 UVM Agent

The UVM agent is a container component class. Agent groups together different verification components that are dealing with a particular interface of DUT. The Agent includes other components such as sequencer that manages stimulus flow, the driver that applies stimulus to the

DUT input and monitor that senses the DUT outputs. UVM agents can also include other components, like a TLM model, protocol checkers, and coverage collectors. The sequencer collects the sequences and sends to the driver. The driver then converts a transaction sequence into signal-level at DUT interface. Agent can operate in two kinds of mode active agent and passive agent. Active agent can generate stimulus, whereas passive agents only sense the DUT (sequencer and driver are disabled). Driver has a bidirectional interface to the DUT, while the Monitor has only unidirectional interface[18].

3.3.5 UVM Sequence Item

A UVM sequence item is the lowest object present under the UVM hierarchy. The sequence-item defines the transaction data items and constraints imposed on them; for example, AXI transaction and it is used to develop sequences. The concept of the transaction was created to isolate Driver from data generation but to deal with DUT interface pin wiggling activities at the bit level. UVM sequence items can include variables, constraints, and even function call for operating on themselves[18].

3.3.6 UVM Sequence

After creating a UVM sequence item, the verification environment has to generate sequences using the sequence item that could be sent to the sequencer. Sequences are a collection of ordered sequence items. The transactions are generated based on the need. Since the sequence item variables are typically random type, sequence helps to constrain or restrict the set of values sent to the DUT. Ultimately helps is reducing simulation time [18].

3.3.7 UVM Driver

A UVM Driver is a component class where the transaction-level sequence item meets the DUT clock/ bit/ pin-level activities. Driver pulls sequences from sequencer as inputs, then converts those sequences into bit-level activities, and finally drive the data onto the DUT interface according to the standard interface protocol. The functionality of driver is restricted to send the appropriate data to the DUT interface. Driver can well off course monitor the transmitted data, but that violates modularity aspects of UVM. Driver uses TLM port (seq_item_port) to receive transaction items from sequencer and use interface to drive DUT signals[18].

3.3.8 UVM Sequencer

The UVM sequencer controls request and response flow of sequence items between sequences generated and the driver component. UVM sequencer acts like an arbiter to control transaction flow from multiple sequences. UVM sequencer use TLM interface method seq_item_export and UVM driver use TLM interface method seq_item_import to connect with each other [18].

3.3.9 UVM Monitor

The UVM monitor does things opposite to that of UVM driver. Monitor takes the DUT signal-level/bit-level values and converts into transactions to needs to be sent to the rest of the UVM components such as scoreboard for analysis. Monitor uses analysis port to broadcasts the created transactions. In order to adhere to the modularity of the UVM testbench, comparison with expected output is usually performed in a different UVM component usually scoreboard. UVM monitor can also perform processing on post converted transaction such as collecting the coverage, recording, logging, checking, etc. or delegate the work to other components using monitor's analysis port [18].

3.3.10 UVM Scoreboard

The UVM scoreboard implements checker functionality. The checker usually verifies the DUT response against an expected DUT response. The scoreboard receives output transactions from the monitor through agent analysis ports, and can also receive expected output from a reference module. Finally, the scoreboard compares both received DUT output data versus expected data. A reference model can be written in C, C++, SystemC, or simply a SystemVerilog model. The SystemVerilog Direct Programming Interface (SystemVerilog-DPI) API is used integrate reference models written in C, C++, etc., and allows them to communicate with the scoreboard [18].

3.4 UVM Transaction Level Communication Protocol

Transaction refers to a class object that includes necessary information needed for communication between two components. Simple example could be a read or write transaction on a bus. Transaction-level modeling (TLM) is an approach that consists of multiple processes communication with each other by sending transaction back and forth through channels. The channels could be FIFO or mailbox or queue. The advantages of TLM are it abstracts time, abstracts data and abstracts function.

3.4.1 Basic Transaction Level Communication

TLM is basis for modularity and reuse in UVM. The communication happens through method calls. A TLM port specifies the API or function call that needs to be used. A TLM export supplies the implementation of the methods. Connections are between ports and exports and not between components. The ports and exports are parameterized by the transaction type being communicated. TLM supports both blocking (put, get/peek) and non-blocking (try_put, try_get/

try_peek) methods. If there are multiple transaction that needs to be communicated TLM FIFO are used. In this way the producer need not wait until consumer consumes each transaction.

3.4.2 Analysis ports and Exports

Analysis ports supports communication between one to many components. These are primarily used by coverage collectors and scoreboards. The analysis port contains analysis exports connected to it. When a UVM component class calls analysis port write method, then the analysis port iterates through the lists and calls write method of appropriate connected export. Similar to that of TLM FIFO Analysis ports also extends the feature to support multiple transaction.

3.5 UVM Phases

All the UVM classes in section 3.3 have different simulation phases. UVM uses phases as ordered steps of execution. Phases are implemented as methods. When deriving a new component class, the testbench simulation will go through different steps to connect, construct and configure each components of the testbench component hierarchy. Moreover, if a particular phase is not needed in some of the component class, it is possible to ignore that particular phase, and the compiler will include in its compilation process. UVM phases are represented in Figure 3.2 [19].

3.5.1 Build Phase

The build phase instantiate UVM components under the hierarchy. Build phase is the only top-down phase among all other UVM phases. For example, the build phase of the env class will construct the classes for the agent and scoreboard [19].

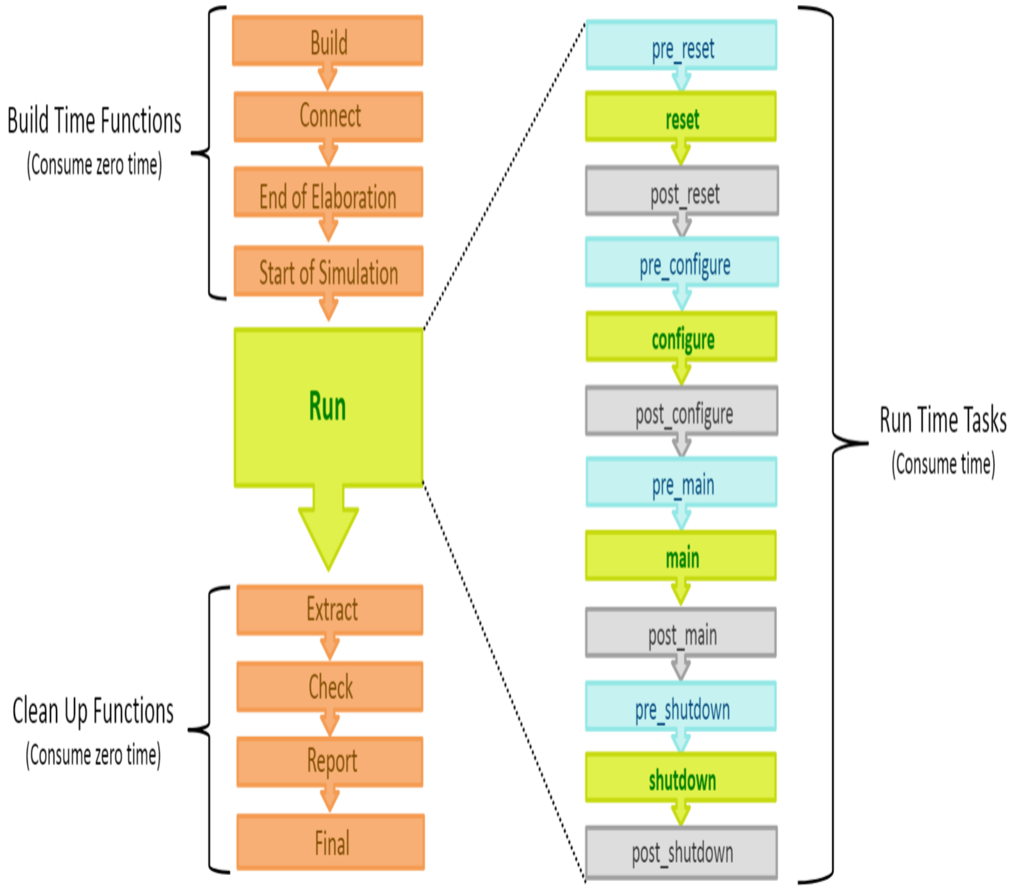

Figure 3.2: UVM Phases

3.5.2 Connect Phase

The connect phase connects UVM subcomponents of a class. Connect phase is executed from the bottom up. In this phase, the testbench components are connected using TLM connections. Agent connect phase would connect the monitor to the scoreboard.

3.5.3 End of Elaboration Phase

Under this phase actions such as checking connections, setting up address range, initializing values or setting pointers and printing UVM testbench topology etc. are performed.

3.5.4 Start of Simulation Phase

During start of simulation environment is already configured and ready to simulate. In this phase actions such as setting initial runtime configurations, setting verbosity level of display statements, orienting UVM testbench topology to check for correctness etc., are performed.

3.5.5 Normal Run Phase

The run phase is the main execution phase, actual simulation of code will happen here. Run phase is a task and it will consume simulation time. The run phases of all components in an environment run in parallel. Any component can use either the run phase or the 12 individually scheduled phase. This phase starts at time 0. It is a better practice to use normal run phase task for drivers, monitors and scoreboards.

3.5.6 Scheduled Run Phase

Any component can use either the run phase or the 12 individually scheduled phase.

3.5.6.1 Pre Reset Phase

Actions that need to be performed before the DUT is reset are done in this phase. Starts at 0ns and coincides with the run phase start time.

3.5.6.2 Reset Phase

In this phase, the actual reset of the DUT occurs. This can be accomplished by running a sequence at the reset interface agent. Often, the reset logic is driven from the top level itself.

3.5.6.3 Post Reset Phase

Post reset actions are done in this phase, like verifying that the device under test is in a specific state.

3.5.6.4 Pre Configure Phase

This phase determines the configuration of the device under test.

3.5.6.5 Configure Phase

Sets the device under test to the desired state as determined in pre configure phase. This would typically be register writes, table writes, memory initialization required for the device under test.

3.5.6.6 Post Configure Phase

Follows the configure phase.

3.5.6.7 Pre Main Phase

This phase executes before the main phase.

3.5.6.8 Main Phase

This phase executes and runs the actual test cases.

3.5.6.9 Post Main Phase

Post main phase performs additional tests to verify that device under test behaved correctly based on the main phase.

3.5.6.10 Pre Shutdown Phase

This phase gets ready for shutdown.

3.5.6.11 Shutdown Phase

Shutdown phase performs all end of test checks.

3.5.6.12 Post Shutdown Phase

This phase performs anything that needs to happen after the end of checks are done. Components running in the run phase would end at the same time as the post-shutdown phase of components running in the scheduled phase mode.

3.5.7 Extract Phase

In this phase, actions such as extracting data from scoreboard and DUT (zero-time back door), preparing final statistics and closing file handlers etc. are performed.

3.5.8 Check Phase

Check phase checks the emptiness of the scoreboard, expected FIFOs and any backdoor accesses to memory content.

3.5.9 Report Phase

The reporting phase is used to furnish simulation results, also write the outputs to file.

3.5.10 Final Phase

Finally, this phase closes all file handles and display any messages.

3.6 UVM Macros

UVM macros are important aspect of the methodology. It is basically implemented methods that are useful in classes and in variables. Some of the most commonly used Marcos are:

- ‘uvm_component_utils - This macro registers is used when new ‘uvm_component classes are derived.
- ‘uvm_object_utils – Similar to ‘uvm_component_utils but instead used with ‘uvm_object.
- ‘uvm_field_int - Registers a variable into factory. And implements functions like compare(), print(), and copy().
- ‘uvm_info – During simulation time this macro is used to print useful messages from the UVM environment .
- ‘uvm_error - Sends messages with an error tag to the output log.

Chapter 4

System Architecture

4.1 WISHBONE Interface

The WISHBONE System-on-Chip Interconnection Architecture shown in Figure 4.1 for portable and flexible IP Cores enables a design methodology for use with semiconductor IP cores. The WISHBONE interface alleviates System-on-Chip integration problems and results in faster design reuse by allowing different IP cores are connected to form a System-on-Chip. As defined, the WISHBONE bus uses both MASTER and SLAVE interfaces as part of the architecture. IP cores with MASTER interfaces initiate bus cycle transactions, and the participating IP cores with SLAVE interfaces can receive the designated bus cycles transactions. MASTER and SLAVE IP cores communicate through an interconnection interface called the INTERCON. The INTERCON is best thought of as a cloud that contains circuits and allows the communication with SLAVEs. INTERCON includes Point-to-point interconnection, Data flow interconnection, Shared bus interconnection and Crossbar switch interconnection [6]. WISHBONE Bus protocols include the implementation of an arbitration mechanism in centralized or distributed bus arbiters. The bus contention issue during the configuration of WISHBONE bus protocol is settled with

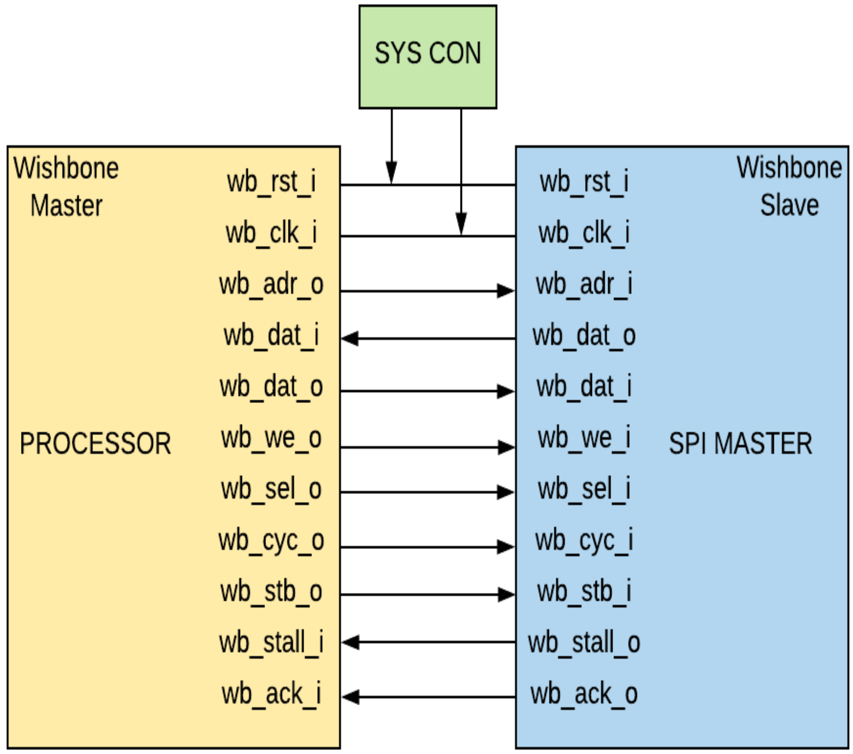

Figure 4.1: Wishbone Interface

the help of a Handshaking protocol and through the deployment of various arbitration schemes such as TDMA, Round Robin, CDMA, Token Passing, Static Priority etc. These strategies are applied based on the specific application in WISHBONE Bus [20].

4.2 WISHBONE I/O Registers

Table. 4.1 refers to the wishbone interface signals used for our Serial Peripheral Interface communication.

- wb_clk_i: All internal WISHBONE logic are sampled at the rising edge of the wb_clk_i clock input.

Table 4.1: WISHBONE I/O Ports

Port	Width	Direction	Description
wb_clk_i	1	Input	Master clock input
wb_rst_i	1	Input	Asynchronous active low reset
wb_int_o	1	Output	Interrupt signal request
wb_cyc_i	1	Input	Valid bus cycle
wb_stb_i	1	Input	Strobe/core select
wb_adr_i	32	Input	Address bit
wb_we_i	1	Input	Write enable
wb_dat_i	32	Input	Data input
wb_dat_o	32	Output	Data output
wb_ack_o	1	Output	Normal bus termination
wb_stall_o	1	Output	Stall communication

- wb_rst_i: wb_rst_i is active low asynchronous reset input and forces the core to restart. All internal registers are preset, to a default value and all state-machines are set to an initial state.

- wb_int_o: The interrupt request output is asserted back to the host system when the core needs its service.

- wb_cyc_i: When the cycle input wb_cyc_i is asserted, it indicates that a valid bus cycle is in progress. It needs to become true on (or before) the first wb_stb_i clock and stays true until the last wb_ack_o. The logical AND function of wb_cyc_i and wb_stb_i indicates a valid transfer cycle to/from the core. This logic is usually taken care of by the bus master.

- wb_stb_i: The strobe input wb_stb_i is true for any bus transaction request. While wb_stb_i is true, the other wishbone slave inputs wb_we_i, wb_addr_i, wb_data_i, and wb_sel_i are valid and reference the current transaction. The transaction is accepted by the slave core any time when wb_stb_i is true, and at the same time, wb_stall_o is false.

- wb_adr_i: The address array input wb_adr_i passes the binary coded address to the core. The MSB is at the higher number of the array. Of the all possible 32 address lines, the slave might only be interested in the relevant slave address

- wb_we_i: When the signal wb_we_i asserted, it indicates that the current bus cycle is a write cycle. When de-asserted, it indicates that the current bus cycle is a read cycle.

- wb_dat_i: The data array input wb_dat_i is used to pass binary data from the current WISHBONE Master to the core.

- wb_dat_o: The data array output wb_dat_o is the data returned by the slave to the bus master as a result of any read request.

- wb_ack_o: When asserted, the acknowledge output wb_ack_o indicates the normal termination of a valid bus cycle. There must be only one clock cycle with wb_ack_o high.

- wb_stall_o: Controls the flow of data into the slave. It will be true in any cycle when the slave can't accept a request from the bus master, and false any time a request can be accepted. It allows the slave core to control the flow of requests that need to be serviced based on master inputs.

4.3 Serial Peripheral Interface

A Serial Peripheral Interface (SPI) module allows synchronous, serial and full duplex communication between a Microcontroller unit and peripheral devices and was developed by Motorola in the mid 1980s. Figure 4.2 represents the structural connection between master and slave core. The SPI bus is usually used to send and receive data between microcontrollers and other small peripherals units such as shift registers, sensors, SD cards, etc. When compared to other proto-

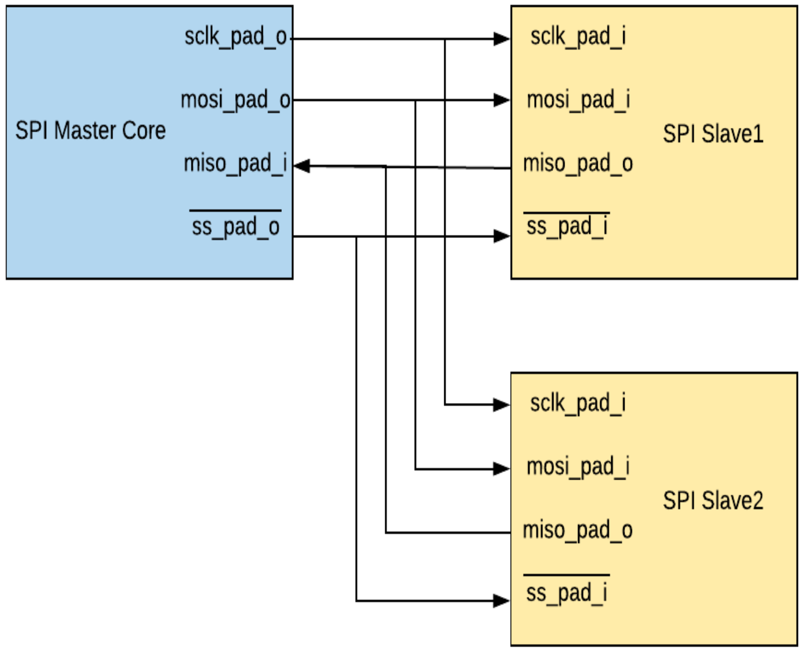

Figure 4.2: SPI Protocol

cols, the SPI protocol has the advantage of relatively high transmission speed, simple to use, an uses a small number of signal pins. Usually, the protocol divides devices into master and slave for transmitting and receiving the data. The protocol uses a master device to generate separate clock and data signal lines, along with a chip-select line to select the slave device for which the communication has to be established. If there is more than a slave device present, the master device must have multiple chip select interfaces to control the devices [21].

4.4 Data Transmission

The SPI bus interface consists of four logic signals lines namely Master Out Slave In (MOSI), Master In Slave Out (MISO), Serial Clock (SCLK) and Slave Select (SS).

Master Out Slave In (MOSI) - The MOSI is a unidirectional signal line and configured as an output signal line in a master device and as an input signal line in a slave device. It is responsible for transmission of data in one direction from master to slave.

Master In Slave Out (MISO) - The MOSI is a unidirectional signal line and configured as input signal line in a master device and as an output signal line in a slave device. It is responsible for transmission of data in one direction from slave to master. When a particular slave is not selected, the MISO line will be in high impedance state.

Slave Select (SS) - The slave select signal is used as a chip-select line to select the slave device. It is an active low signal and must stay low for the duration of the transaction.

Serial Clock (SCLK) - The serial clock line is used to synchronize data transfer between both output MOSI and input MISO signal lines. Based on the number of bytes of transactions between the Master and Slave devices, required number of bit clock cycles are generated by the master device and received as input on a slave device [3].

In the standard SPI protocol, when the communication is initiated, the master device configures the system clock (known as SCLK) to a frequency less than or equal to the maximum possible frequency the slave device supports. The usual frequencies for the communication are in the range of 1-100 MHz. Standard SPI protocol supports single master and multiple devices. The master then transmits appropriate chip-select bit to Logic 0 to select the slave device, since the chip-select line is active low. Thus the communication between master and slave is established, unless the current communication cycle is discarded by the master controlling of slave devices are not possible. The clock (SCLK) is used by all the SPI signals to synchronize. The transmissions involve two shift register of a pre-configured word size are present one each at master and slave ends. As shown in Figure 4.3 both the shift registers act as a ring buffer [22]. While shifting out the data usually the least significant bit from the master is sent to the most significant bit position of the slave receive register, and at the same time, the least significant bit of the slave goes to the vacant least significant bit. Both master and slave register acting in a left

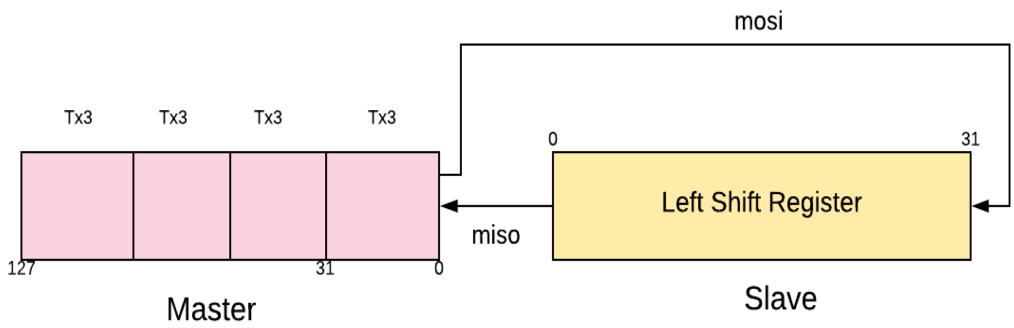

Figure 4.3: Shift Register

shift register fashion and the register values are exchanged with respect to SCLK [6]. If more data needs to be exchanged, then the shift registers are loaded with new data, the and the process is repeated. Finally, after the data values are transmitted then master stops toggling the SCLK and it deselects the slave [22].

4.5 Hardware Architecture

The designed SPI Master IP core is compatible with the SPI protocol and bus principle. At the host side, the design is equivalent to the slave devices of wishbone bus specification complaint. The overall structure of the Wishbone complaint SPI Master core device can be divided into three functional units(Figure 4.4): Clock generator, Serial Interface and Wishbone Interface [23].

4.5.1 Design of Clock Generation module (spi_clk_gen)

The clk_gen is responsible for the generation of the clock signal from the external system clock wb_clk_i, in accordance with different frequency factor of the clock register and produce the output signal s_clk_o. Since there is no response mechanism for Serial Peripheral Interface, in

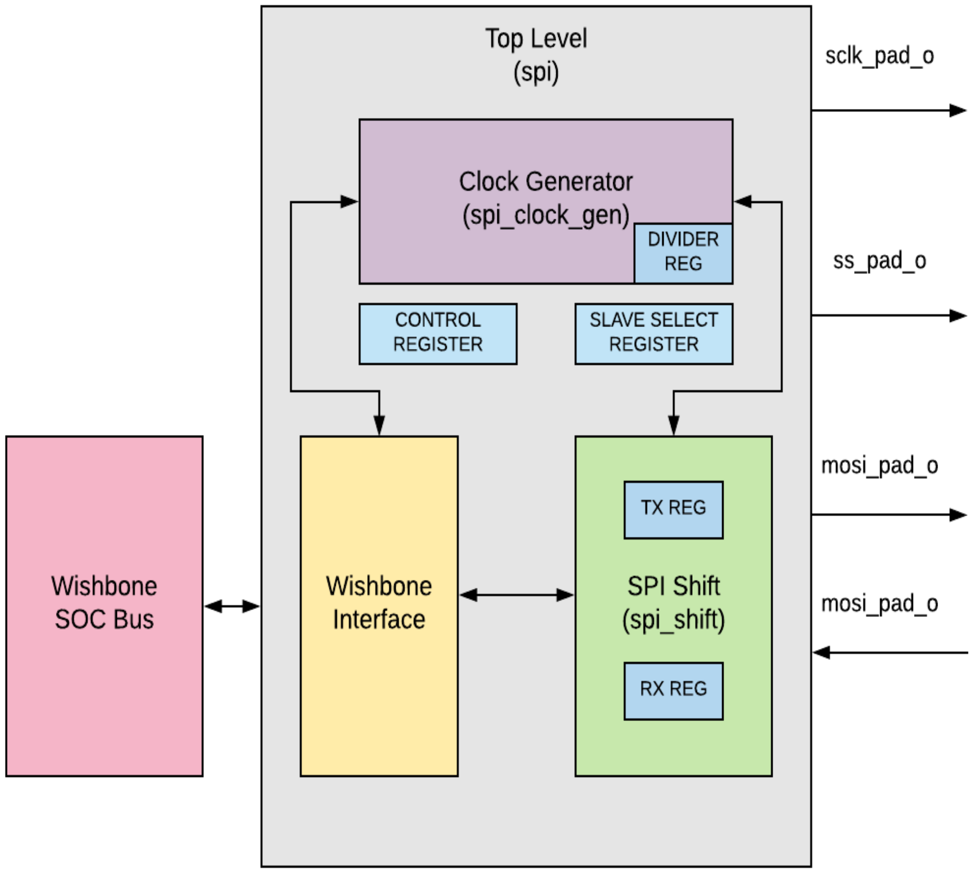

Figure 4.4: SPI Master Architecture

order to ensure the reliability of timing, the clk_gen module can generate reliable serial clock transmission with odd or even frequency division in the register. Clock divider is essential part of digital ASIC and FPGA design, the idea here is to produce frequency relevant to the communication system. Even frequency division is achieved in order to save resources. The core generates the s_clk_o by dividing the wb_clk_i; Arbitrary clock output frequency is achieved by changing the value of the divider. The expression of s_clk_o and wb_clk_i is as follows [22].

$$f_{sclk} = f_{wbclk}/(DIVIDER+1)*2$$

4.5.2 Serial data transfer module design (spi_shift)

Serial data transfer module forms the data transfer core module. It is responsible for converting input parallel data into serial output data to transmit at MOSI and convert input MISO serial data into parallel out. The Receive and Transmit register share same flip-flops. It means that what data is received from the input data line in one data transfer will be transmitted on the output line in the next transfer if no write access to the transmit register was performed between the transfers. The advantage of this is it uses fewer hardware resources, therefore, lesser power consumption. [27] SPI Master core in host side acts as a slave device to receive input data, and at the same time as the master device transmits output data [22].

4.5.3 Top-level module (spi)

The role of the top-level module is to get the basic structure of high-speed reusable SPI bus sub-components to work smoothly. Therefore, the top-level of the SPI module controls normal operation of clock generator module and serial data transmission module [22].

4.6 SPI Registers

The SPI master core uses the register [24] mentioned in the Table 4.2

4.6.1 RxX Register

The Data Receive registers hold the value of data received from the last executed transfer. CTRL register holds the character length field for example if CTRL [9:3] is set to 0x10, bit RxL[15:0] holds the received data. Registers Rx1, Rx2 and Rx3 are not used If character length is less or equal to 32 bits, likewise Registers Rx2 and Rx3 are not used if character length is less than 64

Table 4.2: SPI Master core registers

Name	Address	Width	Access	Description
Rx0	0x00	32	R	Data receive register 0
Rx1	0x04	32	R	Data receive register 1
Rx2	0x08	32	R	Data receive register 2
Rx3	0x0C	32	R	Data receive register 3
Tx0	0x00	32	R/W	Data transmit register 0
Tx1	0x04	32	R/W	Data transmit register 1
Tx2	0x08	32	R/W	Data transmit register 2
Tx3	0x0C	32	R/W	Data transmit register 3
CTRL	0x10	32	R/W	Control and status register
DIVIDER	0x14	32	R/W	Clock divider register
SS	0x18	32	R/W	Slave select register

bits and so on.

4.6.2 TxX Register

The Data Receive registers hold the value of data transmitted from the transfer. CTRL register holds the character length field for example if CTRL [9:3] is set to 0x10, bit TxL[15:0] holds the received data. Registers Tx1, Tx2 and Tx3 are not used If character length is less or equal to 32 bits, likewise Registers Tx2 and Tx3 are not used if character length is less than 64 bits and so on.

4.6.3 ASS Register

If ASS bit is set, the ss_pad_o signal is generated automatically. When the transfer is started by setting CTRL[GO_BSY], the slave select signal which is selected in SS register is asserted by the SPI controller and is de-asserted after the transfer is finished. If ASS bit is cleared, then the slave select signals are asserted and de-asserted by writing and clearing the bits in SS register.

4.6.4 DIVIDER Register

The value in this field divides the frequency of the system clock (wb_clk_i) to generate the serial clock(s_clk) on the output sclk_pad_o. The desired frequency is obtained according to equation1.

4.6.5 SS Register

When CTRL[ASS] bit is cleared, writing 0x1 to any of the bit locations of this field sets the proper ss_pad_o line to an active state and writing 0x0 sets the line back to the inactive state. When CTRL [ASS] bit is set, writing 1 to any bit location of this field will select appropriate ss_pad_o line to be automatically driven to an active state for the duration of the transfer, and will be driven to an inactive state for the rest of the time.

4.6.6 IE Register

When this bit is set, the interrupt output is set active once after a transfer is finished. The Interrupt signal is cleared after a Read or Write to any register.

4.6.7 LSB Register

When LSB bit is set to 0x1, the least significant bit is sent first on the line (bit TxL[0]), and the first bit received from the line will be put in the least significant bit position in the Rx register (bit RxL[0]). When this bit is cleared, the MSB is transmitted /received first (CHAR_LEN field in the CTRL register selects which bit in TxX/RxX register).

4.6.8 Tx_NEG Register

When Tx_NEG bit is set, the mosi_pad_o signal is sent on the falling edge of a sclk_pad_o clock signal, or otherwise, the mosi_pad_o signal is sent on the rising edge of sclk_pad_o.

4.6.9 Rx_NEG Register

When Rx_NEG bit is set, the miso_pad_i signal is received on the falling edge of a sclk_pad_o clock signal, or otherwise, the miso_pad_i signal is received on the rising edge of sclk_pad_o.

4.6.10 GO_BSY Register

Writing 0x1 to this bit starts the transfer and remains set during the transfer. Automatically cleared after the transfer is finished. Writing 0x0 to this bit has no effect.

4.6.11 CHAR_LEN Register

This field specifies the number of bits to be transmitted in one transfer. Can send up to 64 bits in one transfer.

CHAR_LEN = 0x01 … 1 bit

CHAR_LEN = 0x02 … 2 bits

…

CHAR_LEN = 0x7f … 127 bits

CHAR_LEN = 0x00 … 128 bits

4.7 Limitation of Standard SPI and Advancements

Standard SPI communication is a single-master communication. Therefore all the communication can only have one master device active at any time. This limits the functional aspects of the devices that are connected to the SPI topology. To overcome this more advanced designs adopt the parameterization method, identify the master/slave devices automatically and use Time Sharing Multiplex (TSM) technology to control the same slave device at the same time [25].

Chapter 5

Test Methodology and Results

5.1 Testbench Components

The SPI master core is verified along with the SPI slave model. Initially, the SPI master and slave have configured appropriately (for example at the master end no. of bits-32, transmit-posedge, receive-negedge). The basic idea of the verification is to send data from both master and slave ends. And after the transfer is completed verify the exchanged data at both the ends. The Figure. 5.1 shows the testbench module approach. Below each of the components is explained.

5.1.1 Test top

The top-level module is responsible for integrating the testbench module with the device under test. This module instantiates two interfaces, one for the master and another for the slave. Then the master interface is wired with SPI master core and likewise slave interface with SPI slave model. The top module also generates the clock and registers the interface into the config database so that other subscribing blocks can retrieve. Finally, the module calls the run_test function which starts to run the uvm_root.

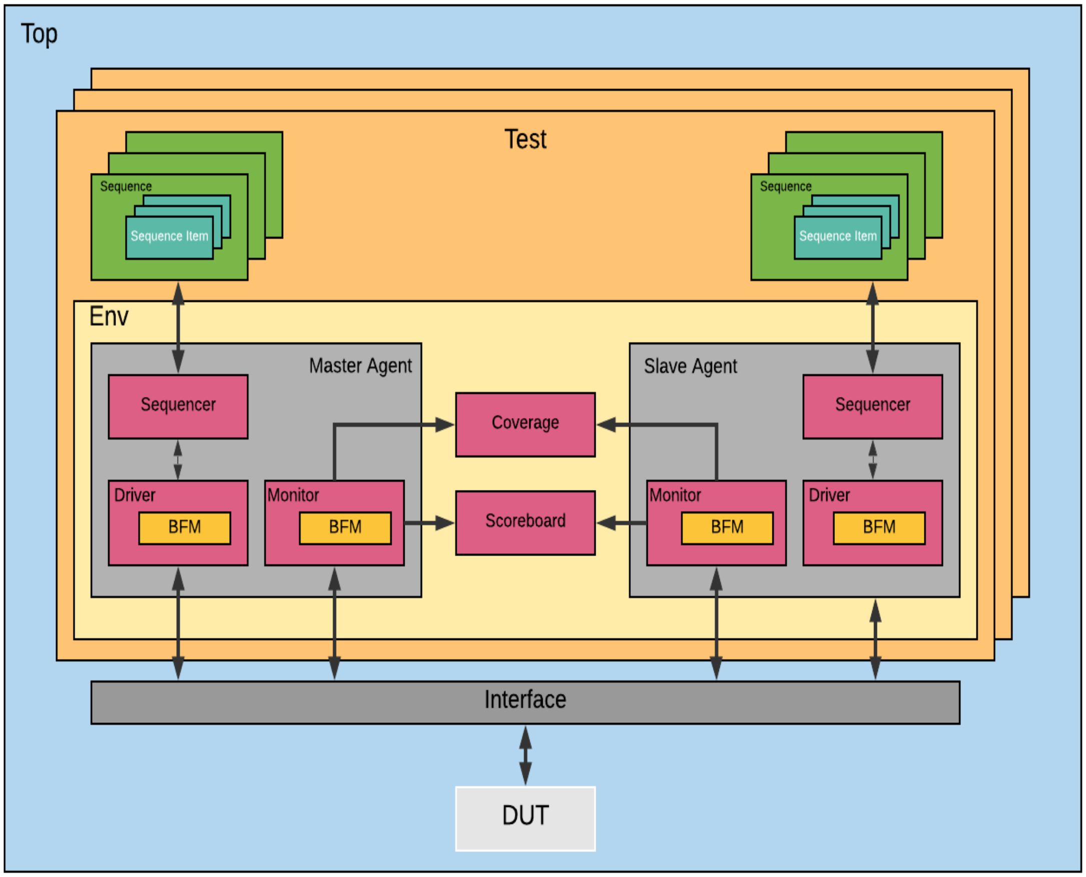

Figure 5.1: UVM Testbench model

5.1.2 spi_interface

The interface block declares all the WISHBONE slave logic signals. The communication with the master and slave core happens through WISHBONE bus function model. The block also samples the input and output signals using two different clocking blocks, one for driver and another for the monitor. Clocking block helps to synchronize all logic signals to a particular clock. It also helps to separate the timing details from the structural, functional and procedural elements of the testbench.

5.1.3 spi_package

The package class typically includes all SystemVerilog testbench components and make the scope available to the entire build process.

5.1.4 spi_test

The test class is created by extending the uvm_test class. Then the class is registered to factory using uvm_component_utils macro. In the build phase, the lower level SPI environment class is created and configured. Instead of the run phase, the test class contains two of the twelve scheduled phases. Reset phase typically resets the device under test. The main phase used to create the sequences and start running the sequencer for the required number of tests. Whenever there needs to be a blocking phase execution, phase raise objection is invoked and like to unblock phase drop objection is used.

5.1.5 spi_environment

SPI environment is a container component containing the agent and scoreboard. It is created using uvm_env virtual base class. In the build phase components within the environment are instantiated. And in the connect phase, the connections are made between components.

5.1.6 spi_agent

Currently, there is only one agent container component is used within the project. The SPI agent container is configured as an active component. SPI agent is created using uvm_agent virtual base class. In the build phase, the agent builds Sequencer, Driver and Monitor components. In the connect phase, the driver and sequencer are connected.

5.1.7 spi_sequence_item

The data flows through the testbench from component to component in the form of packets called as transaction class or sequence item. The SPI sequence item class is created by extending the uvm_sequence_item class. The transaction packet consists of register configuration items (control, divider, and slave select) and data items (input, output and expected) for both master and slave. Then register the class and properties to factory using uvm_object_utils macro. A constructor function is defined for the sequence item. Randomization is applied to sequence items.

5.1.8 spi_sequence

The user-defined SPI sequence class uses uvm_sequence as its virtual base class. This class is a parameterized class with the parameter being the SPI sequence item associated with this sequence. Body() method is called, and code within this method gets executed when the sequence is run. Objections are typically raised and dropped in the pre_body() and post_body() methods of a sequence. Within the body() method the register sequence items and the data sequence items are constrained randomized.

5.1.9 spi_sequencer

SPI sequencer is the component that runs the sequences. The sequencer has a built-in port called sequence_item_export to communicate with the driver. Through this port, the sequencer can send a request item to the driver and receive a response item from the driver. This class is parameterized with SPI sequence item.

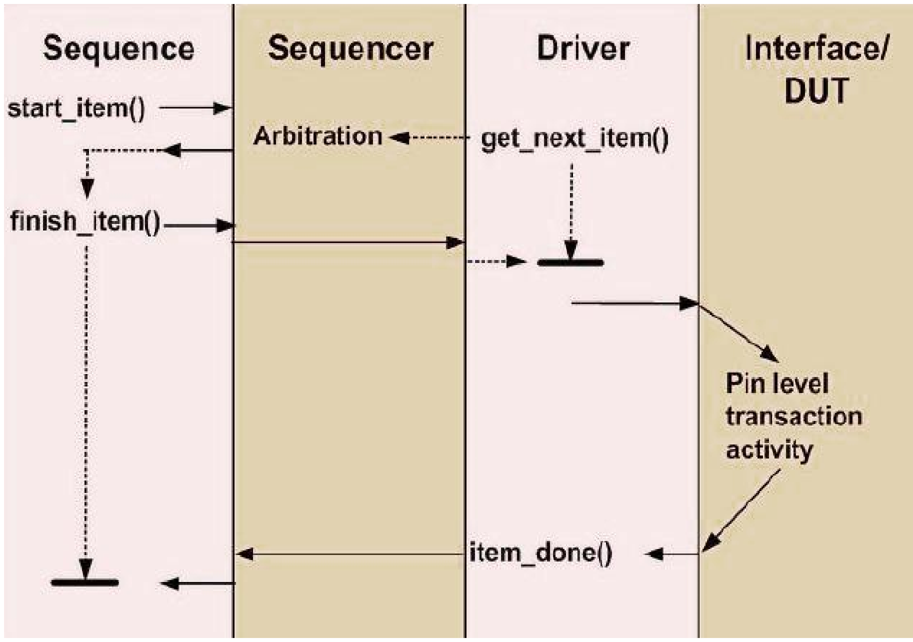

Figure 5.2: UVM Sequencer Driver Communication

5.1.10 spi_driver

SPI driver is the component along with WISHBONE bus function model that takes the generated sequence item from the sequencer and drives it into the DUT according to WISHBONE protocol. The driver is created extending uvm_driver. In order to drive the data virtual interface handle is passed to the driver during the build phase. The SPI driver initially calls the WISHBONE reset method. Then a forever thread is created. In this thread initially, the driver gets the next sequence item from sequencer using the seq_item_port method. This synchronizes with the body function of the sequence as given in the Figure 5.2 and packet is driven into the DUT using the bus function model. In the end, the driver waits for transfer complete interrupt to repeat the thread loop.

5.1.11 spi_monitor

SPI monitor senses the response from the DUT. In order to monitor the data, virtual interface handle is passed to monitor during the build phase. The monitor is created extending uvm_monitor. Initially, the monitor waits for the first SPI data transfer to begin. Then In the forever thread, the monitor waits for the SPI data transfer to complete. SPI monitor uses WISHBONE bus function model to read the response data from DUT. The sequence-item data packet containing the actual and expected output is now broadcast to the environment using analysis write port. The monitor then waits again for a new transfer to being, and this process repeats in a loop.

5.1.12 spi_scoreboard

SPI scoreboard is the component which has transaction level checkers and coverage collectors to verify the functional correctness of a given DUT. Scoreboard class is extended from the uvm_scoreboard base class. TLM analysis FIFOs to connect to the monitor. In the run phase, the input packet is retrieved from the driver, while the output packet is retrieved from the monitor. Then the transaction level functional coverage method is performed using a sampling method to get the coverage. In the end, then when the report phase is invoked the results are displayed.

5.1.13 wishbone_bfm

The WISHBONE bus function model at the driver side transfers the transaction level packets into WISHBONE specific pin level data. At the monitor side, it receives the pin level activities WISHBONE and wraps into transaction packets for higher level modules to use. WISHBONE bus function module implements three methods write, read and reset. The bus function module is non-synthesizable code and written using SystemVerilog.

5.2 Testbench Results

The functional verification of the SPI core controller was carried out successfully with the following results.

5.2.1 SPI Master Controller Synthesis Benchmarking

The project aims to create a functional verification environment for SPI controller. For this purpose the IP core was reused from Opencores, but with some modification. The logic synthesis of the module was performed in the TSMC 180nm, 65nm and SAED 32nm technology. Area, Power and Timing of the final module were captured Table 5.1

Table 5.1: Synthesis Report

Type	Technology node	32 nm	65 nm	180 nm
Area	Sequential Area (μm^2)	2096.68	2520.35	18990.41
	Combinational Area (μm^2)	2527.97	2209.68	17071.08
	Buf/Inv Area (μm^2)	314.37	71.28	1862.78
	Total Area (μm^2)	5847.47	4730.03	36061.50
Power	Internal Power (μW)	32.59	47.34	335.80
	Switching Power (μW)	1.844	3.58	74.86
	Leakage Power (μW)	452.2	0.189	0.145
	Total Power (μW)	486.6	51.11	410.8
Timing	Slack (ns)	18.375	17.958	12.983
DFT Coverage		100%	100%	100%
Latency (Clock cycles)				

5.2.2 Data Transactions

The results published are for below Table 5.2 configuration for a regression run of 10 Million tests.

Table 5.2: Test Configuration

Data Transfer	Sent First	Transmit	Receive
32bit	MSB	posedge	negedge

5.2.2.1 WISHBONE to SPI Master communication using BFM

The communication between the WISHBONE and SPI master is performed using WISHBONE bus function model. The model mainly implements read, write and reset functionalities w.r.t WISHBONE B.3 protocol. In the below Figure. 5.3 shows the WISHBONE protocol. Initially when there is a write data is involved cycle, strobe and write enable signals along with select lines of WISHBONE are asserted to 0x1 by the bus master. The WISHBONE address and data at the same time is placed on the bus. The bus model waits until a receive acknowledgment from the slave is received. Then the bus master frees the bus by terminating the cycle signal to 0x0. For example, if the control register needs to be configured, then control register address 0x10 is sent along with the data value 0x2200, referred at reference 1 in the Figure. 5.3. Correspondingly, the SPI control select flag is selected, and in the next cycle, the value is written to the local control register of the device under test.

5.2.2.2 SPI Master-Slave communication

The master and slave communication in Figure. 5.4 is synchronized to sclk_pad clock, which is synchronized to the wb_clk base clock. Before the start of transfer, the master and slave configure its control register. Control register contains flags like tx_negedge/rx_posedge, which

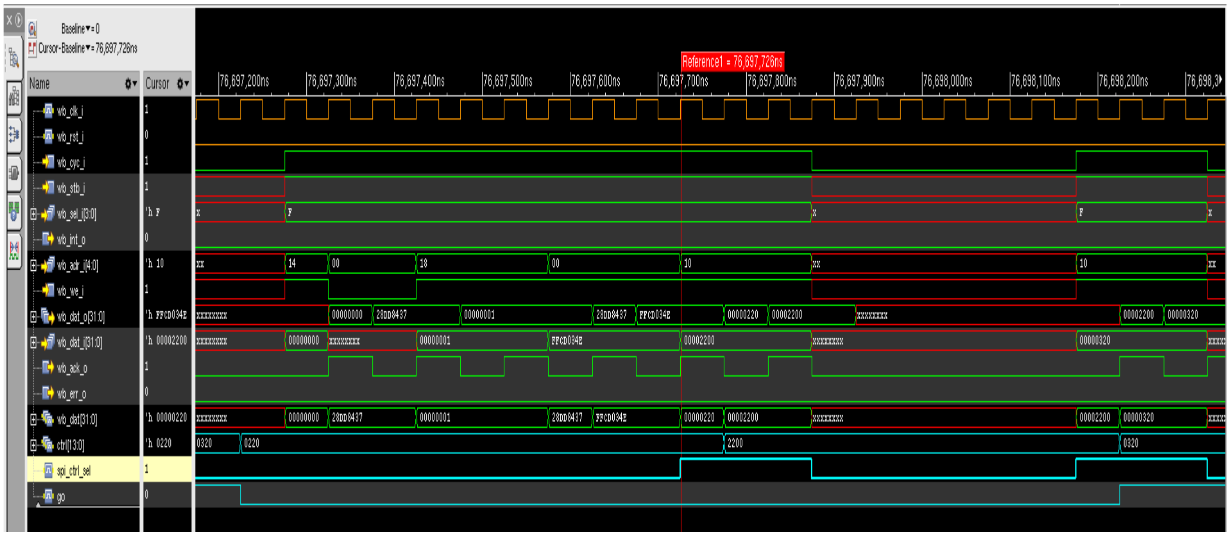

Figure 5.3: WISHBONE to SPI communication

determines the sampling edge of send and receive signal. These two flags should have opposite values to each other since the SPI read input and write output takes place at the same single buffer in a shift register fashion. The master also configures its divider register and slave select register. Once all SPI registers are initially set up, then go flag of the control signal is asserted, which starts the transfer. The testbench uses the flag transfer in progress to synchronize driver and monitor respective forever loop part. Finally as given in Figure. 5.4 after 32 clock cycles, the transfer in progress signal is de-asserted and thus informs the end of communication for the WISHBONE interface to collect the data.

5.2.3 Coverage

Functional coverage is essential to any verification plan, in the project it the coverage is retrieved using Cadence Integrated Metrics Centre tool. Functional coverage is a way to tell the effectiveness of the test plan. Functional coverage infers results such if an end to end code checked if an important set of values corresponding to interface or design requirement and boundary

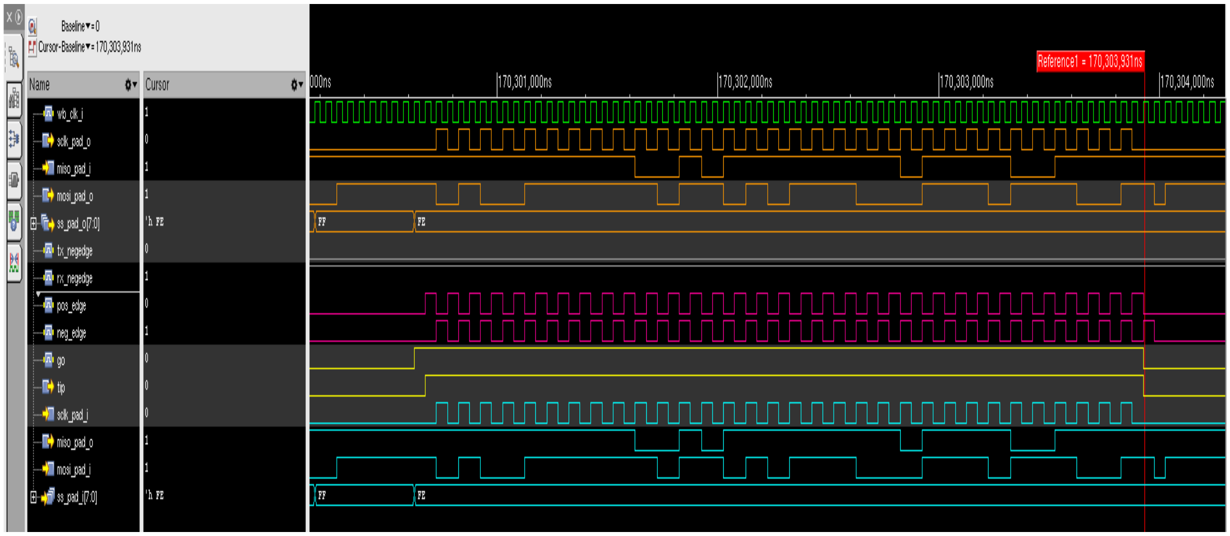

Figure 5.4: SPI Master - Slave communication

conditions have been exercised or not. 100% Functional coverage combined with 100% Code coverage indicates the exhaustiveness of the verification plan coverage.

5.2.3.1 Code Coverage

Tools such as Cadence Integrated Metrics Centre can automatically calculate the code coverage metric. Code coverage tracks information such what lines of code or expression or block have been exercised. However, code coverage is not exhaustive and cannot detect conditions that or not present in the code. To address these deficiencies, we go for functional coverage.

Ex	UNR	Name	Overall Average Grade	Overall Covered
		Overall	66.74%	425 / 887 (47.91%)
		Code	66.74%	425 / 887 (47.91%)
		Block	75.07%	83 / 132 (62.88%)
		Expression	89.7%	46 / 51 (90.2%)
		Toggle	35.45%	296 / 704 (42.05%)

Figure 5.5: Top Level Code Coverage

Figure. 5.5 shows the code coverage for the SPI Top level module. Block coverage is not

100% because not all sections of the code are covered for example for transactions above 32bit higher order SPI receive buffers are not covered. Expression coverage is 100% except for the WISHBONE interrupt acknowledgment section. Finally, toggle coverage is low because for all the input, output wires and registers possible inputs zero's and ones are not covered.

Ex	UNR	Name	Overall Average Grade	Overall Covered
		Overall	66.74%	425 / 887 (47.91%)
		Code	66.74%	425 / 887 (47.91%)
		Block	75.07%	83 / 132 (62.88%)
		Expression	89.7%	46 / 51 (90.2%)
		Toggle	35.45%	296 / 704 (42.05%)

Figure 5.6: Clock Level Code Coverage

Figure. 5.6 shows the code coverage for the SPI Top level module.

Ex	UNR	Name	Overall Average Grade	Overall Covered
		Overall	56.13%	183 / 434 (42.17%)
		Code	56.13%	183 / 434 (42.17%)
		Block	42.86%	30 / 70 (42.86%)
		Expression	86.36%	19 / 22 (86.36%)
		Toggle	39.18%	134 / 342 (39.18%)

Figure 5.7: Shift Level Code Coverage

Figure. 5.7 shows the code coverage for the SPI Top level module. Block coverage is less because not all possible data transfer rates are exercised.

5.2.3.2 Functional Coverage - Signal Level

Signal level functional coverage at Figure. 5.8 is usually applied in the monitor component of the UVM test bench. Signal level exercise the checking at the DUT output pin level. At SPI signal level below three coverpoints are incorporated:

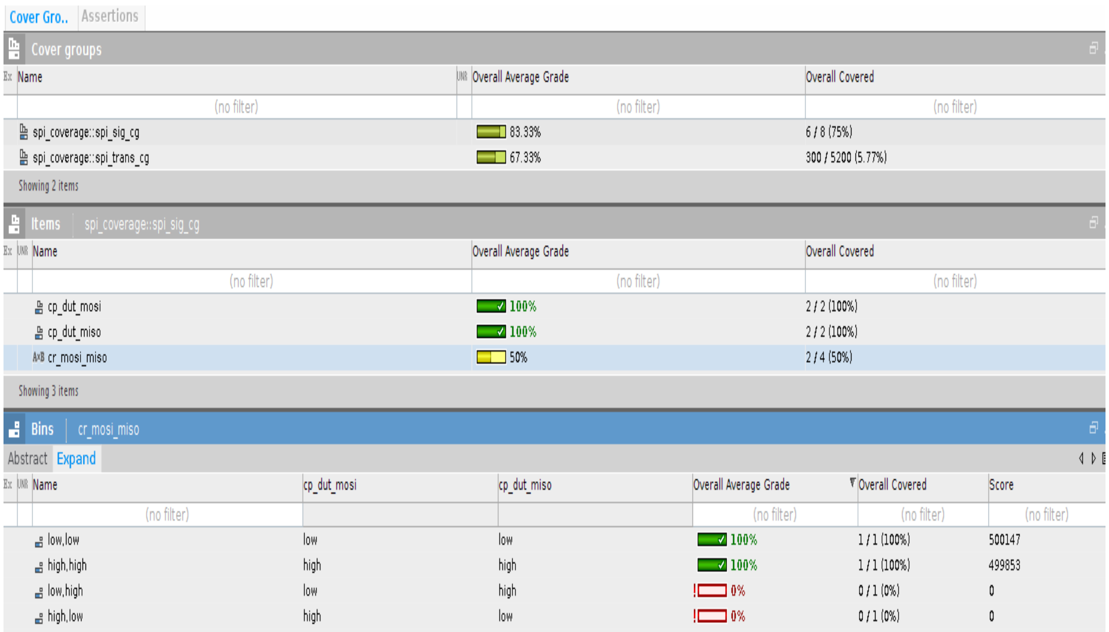

Figure 5.8: Signal Coverage

- cp_dut_mosi: In this coverpoint mosi output line between the master and slave is checked. It has two bins of low bit(0x0) and high bit(0x1). Both the bins are covered 100%
- cp_dut_miso: In this coverpoint miso output line between the master and slave is checked. It has two bins of low bit(0x0) and high bit(0x1). Both the bins are covered 100%
- cp_mosi_miso: This coverpoint gives the cross cover of the both cp_dut_mosi and cp_dut_mosi. It results in total of 2x2 bins. However, only 50% of the bins are hit because the sampling for cross cover happens at the wb_clk master clock and not the sclk clock signal.

5.2.3.3 Functional Coverage - Transaction Level

Transaction level functional coverage at Figure. 5.9 is usually applied in the scoreboard component of the UVM test bench. Signal level exercises the checking at the DUT transaction class outputs. At SPI signal level below six coverpoints are incorporated:

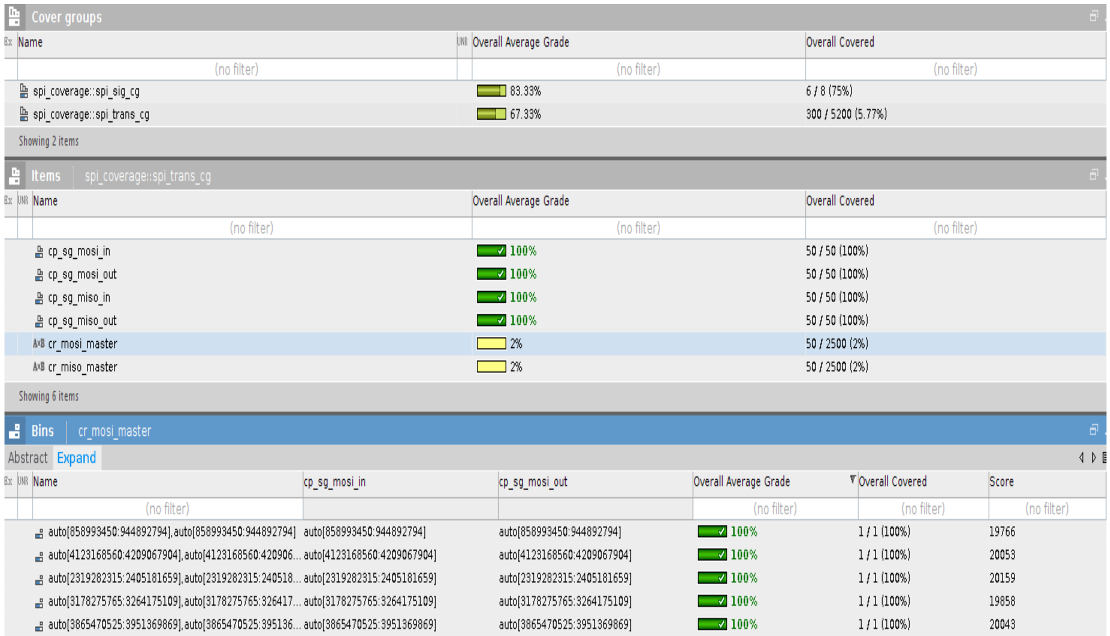

Figure 5.9: Transaction Coverage

- cp_sg_mosi_in: This coverpoint exercises input packets expected master data. Auto bin max value of 50 for this coverpoint owing to reduced regression time availability. Ideally, this should be auto bin max.

- cp_sg_mosi_out: This coverpoint exercises output packets expected master data. Auto bin max value of 50 for this coverpoint owing to reduced regression time availability. Ideally, this should be auto bin max.

- cp_sg_miso_in: This coverpoint exercises input packets expected slave data. Auto bin max value of 50 for this coverpoint owing to reduced regression time availability. Ideally, this should be auto bin max.

- cp_sg_miso_out: This coverpoint exercises output packets expected slave data. Auto bin max value of 50 for this coverpoint owing to reduced regression time availability. Ideally, this should be auto bin max.

- cr_mosi_master: Cross cover of cp_sg_mosi_in and cp_sg_mosi_out is checked in this coverpoint. It verifies if the actual DUT output is equal to expected DUT output. Only 2% of the bins are covered because between actual and expected only one of the 50 bins would be covered and also 50/50*50=2%.

- cr_miso_master: Cross cover of cp_sg_miso_in and cp_sg_miso_out is checked in this coverpoint. It verifies if the actual DUT output is equal to expected DUT output. Only 2% of the bins are covered because between actual and expected only one of the 50 bins would be covered and also 50/50*50=2%.

Chapter 6

Conclusion

In this work, a reusable SystemVerilog based UVM environment is created for an SPI master core controller. The verification environment is built around WISHBONE System on Chip bus thus making both core IP, and verification IP easy to integrate. Configuration capability is provided to configure the testbench to suit different protocol characteristics. The testbench enables to verify and validate the full duplex data transfer between the master core and slave core for various character lengths and data formats respectively.

An SPI slave model was created to enhance the SPI master core verification as end to end feasible. In addition, a WISHBONE BFM was successfully established to form the link between the testbench components and the device under test. The WISHBONE BFM provides basic read and write functionalities. Functional coverage was successfully integrated into the testing environment in order to achieve coverage driven verification metrics.

6.1 Future Work

- The SPI master controller can be enhanced to include First In-First-Out buffers to accept data at different clock rates.
- The SPI master controller can be extended to advanced WISHBONE B4 specification.
- The tests can be further extended to other configurations of SPI master controller so that 100% code coverage can be achieved.

References

[1] W. Ni and J. Zhang, "Research of reusability based on UVM verification," in *2015 IEEE 11th International Conference on ASIC (ASICON)*, Nov 2015, pp. 1–4.

[2] K. Fathy and K. Salah, "An Efficient Scenario Based Testing Methodology Using UVM," in *2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, Dec 2016, pp. 57–60.

[3] P. Rajashekar Reddy, P. Sreekanth, and K. Arun Kumar, "Serial Peripheral Interface-Master Universal Verification Component using UVM," *International Journal of Advanced Scientific Technologies in Engineering and Management Sciences*, vol. 3, p. 27, 06 2017.

[4] R. Prasad and C. S. Rani, "UART IP CORE VERIFICATION BY USING UVM," *IRF International Conference*, 15 2016.

[5] P. Roopesh D, P. Siddesha K, and B. M. Kavitha Narayan, "RTL DESIGN AND VERIFICATION OF SPI MASTER-SLAVE USING UVM," *International Journal of Advanced Research in Electronics and Communication Engineering*, vol. 4, p. 4, 08 2015.

[6] K. Aditya, M. Sivakumar, F. Noorbasha, and P. B. Thummalakunta, "Design and Functional Verification of A SPI Master Slave Core Using SystemVerilog," *International Journal Of Computational Engineering Research*, 05 2018.

[7] N. Anand, G. Joseph, S. S. Oommen, and R. Dhanabal, “Design and implementation of a high speed Serial Peripheral Interface,” in *2014 International Conference on Advances in Electrical Engineering (ICAEE)*, Jan 2014, pp. 1–3.

[8] T. Liu and Y. Wang, “IP design of universal multiple devices SPI interface,” in *Anti-Counterfeiting, Security and Identification (ASID), 2011 IEEE International Conference on*. IEEE, 2011, pp. 169–172.

[9] D. Ahlawat and N. K. Shukla, “DUT Verification Through an Efficient and Reusable Environment with Optimum Assertion and Functional Coverage in SystemVerilog,” *International Journal of Advanced Computer Science and Applications*, vol. 5, no. 4, 2014.

[10] N. Gopal, “SPI Controller Core: Verification,” *SSRG International Journal of VLSI & Signal Processing*, vol. 2, 09 2015.

[11] Z. Zhou, Z. Xie, X. Wang, and T. Wang, “Development of verification envioronment for SPI master interface using SystemVerilog,” in *Signal Processing (ICSP), 2012 IEEE 11th International Conference on*, vol. 3. IEEE, 2012, pp. 2188–2192.

[12] J. Francesconi, J. A. Rodriguez, and P. M. Julian, “UVM based testbench architecture for unit verification,” in *Micro-Nanoelectronics, Technology and Applications (EAMTA), 2014 Argentine Conference on*. IEEE, 2014, pp. 89–94.

[13] A. K. Swain and K. Mahapatra, “Design and verification of WISHBONE bus interface for System-on-Chip integration,” in *India Conference (INDICON), 2010 Annual IEEE*. IEEE, 2010, pp. 1–4.

[14] A. K. Oudjida, M. L. Berrandjia, A. Liacha, R. Tiar, K. Tahraoui, and Y. N. Alhoumays, “Design and test of general-purpose SPI Master/Slave IPs on OPB bus,” in *Systems Signals and Devices (SSD), 2010 7th International Multi-Conference on*. IEEE, 2010, pp. 1–6.

[15] Mahendra.B.M and Ramachandra.A.C, "Bus Functional Model Verification IP Development of AXI Protocol," *International Conference on Engineering Technology and Science*, vol. 3, 02 2014.

[16] P. Araujo, "Development of a reconfigurable multi-protocol verification environment using UVM methodology," *FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO*, pp. 1 – 149, 06 2014.

[17] C. Spear and G. Tumbush, *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*, 3rd ed. Springer Publishing Company, Incorporated, 2012.

[18] A. B. Mehta, *ASIC/SoC Functional Design Verification: A Comprehensive Guide to Technologies and Methodologies*, 1st ed. Springer Publishing Company, Incorporated, 2017.

[19] IEEE, "IEEE Standard for Universal Verification Methodology Language Reference Manual," *IEEE Std 1800.2-2017*, pp. 1–472, May 2017.

[20] M. Sharma and D. Kumar, "Wishbone bus Architecture - A Survey and Comparison," *CoRR*, vol. abs/1205.1860, 2012. [Online]. Available: http://arxiv.org/abs/1205.1860

[21] IEEE, "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, Feb 2018.

[22] D. V. Veda Patil, Vijay Dahake and E. Pinto, "Implementation of SPI Protocol in FPGA," *International Journal Of Computational Engineering Research*, vol. 3, pp. 142 – 147, 01 2013.

[23] S. Ananthula, M. K. Kumar, and J. K. Bhandari, "Design and Verification of Serial Periph-

eral Interface," *International Journal of Engineering Development and Research (IJEDR)*, vol. 1, pp. 130 – 136, Dec. 2014.

[24] Srot and Simon, *SPI Master core specification*, 2004. [Online]. Available: https://opencores.org/project/spi

[25] R. Herveille, *SPI Core Specifications*, 2003. [Online]. Available: https://opencores.org/project/simple_spi

Appendix I

Source Code

I.1 SPI Top

```
/*
 * Author:  Deepak Siddharth Parthipan
 *          RIT, NY, USA
 * Module:  spi
 */
//--------------------------------------------------------------------------------------------

`include "src/spi_defines.v"
`include "src/timescale.v"
//--------------------------------------------------------------------------------------------

module spi
(
```

```
    /* Wishbone signals */
    wb_clk_i, wb_rst_i, wb_adr_i, wb_dat_i, wb_dat_o, wb_sel_i,
    wb_we_i, wb_stb_i, wb_cyc_i, wb_ack_o, wb_err_o, wb_int_o,

    /* SPI signals */
    ss_pad_o, sclk_pad_o, mosi_pad_o, miso_pad_i,

    /* Scan Insertion */
    scan_in0, scan_en, test_mode, scan_out0, tip //,reset, clk
);
/*--------------------------------Wishbone signals------------------------------------------*/
    input                    wb_clk_i;        // master clock input
    input                    wb_rst_i;        // synchronous active high reset
    input          [4:0]     wb_adr_i;        // lower address bits
    input     [32-1:0]       wb_dat_i;        // databus input
    output    [32-1:0]       wb_dat_o;        // databus output
    input          [3:0]     wb_sel_i;        // byte select inputs
```

```
input                    wb_we_i;      // write enable input
input                    wb_stb_i;     // stobe/core select signal
input                    wb_cyc_i;     // valid bus cycle input
output                   wb_ack_o;     // bus cycle acknowledge output
output                   wb_err_o;     // termination w/ error
output                   wb_int_o;     // interrupt request signal output
/*---------------------------------SPI signals ------------------------------------------*/
output [`SPI_SS_NB-1:0]  ss_pad_o;     // slave select
output                   sclk_pad_o;   // serial clock
output                   mosi_pad_o;   // master out slave in
input                    miso_pad_i;   // master in slave out
//input                  reset;        // system reset
```

```
//input                          clk;           // system clock
  input                          scan_in0;      // test scan mode data input
  input                          scan_en;       // test scan mode enable
  input                          test_mode;     // test mode select
  output                         scan_out0;     // test scan mode data output
  output                         tip;
/*---------------------------------------------------------------------------

  reg                  [32-1:0]  wb_dat_o;
  reg                  [32-1:0]  wb_dat;
  reg                            wb_ack_o;
  reg                            wb_int_o;
  reg      [`SPI_CTRL_BIT_NB-1:0] ctrl;
  reg      [`SPI_DIVIDER_LEN-1:0] divider;
  reg            [`SPI_SS_NB-1:0] ss;
  reg                            scan_out0;
  // Internal signals
  wire      [`SPI_MAX_CHAR-1:0]  rx;            // Rx register
```

```
wire                                rx_negedge;       // miso is sampled on negative edge
wire                                tx_negedge;       // mosi is driven on negative edge
wire   [`SPI_CHAR_LEN_BITS-1:0]     char_len;         // char len
wire                                go;               // go
wire                                lsb;              // lsb first on line
wire                                ie;               // interrupt enable
wire                                ass;              // automatic slave select
wire                                spi_divider_sel;  // divider register select
wire                                spi_ctrl_sel;     // ctrl register select
wire                         [3:0]  spi_tx_sel;       // tx_l register select
wire                                spi_ss_sel;       // ss register select
reg                                 tip;              // transfer in progress
wire                                pos_edge;         // recognize posedge of sclk
```

```
wire                              neg_edge;          // recognize negedge of sclk
wire                              last_bit;          // marks last character bit
//----------------------------------------------------------------------------------------

spi_clock_gen clock_gen (.clk_in(wb_clk_i), .rst(wb_rst_i), .go(go), .enable(tip), .last_clk(last_bit),
                          .divider(divider), .clk_out(sclk_pad_o), .pos_edge(pos_edge),
                          .neg_edge(neg_edge));
                          //.scan_in0(scan_in0), .scan_en(scan_en), .test_mode(test_mode), .scan_out0(scan_out0), .reset(reset), .clk(clk));
//----------------------------------------------------------------------------------------

spi_shift shift (.clk_shift(wb_clk_i), .rst(wb_rst_i), .len(char_len[`SPI_CHAR_LEN_BITS-1:0]),
                  .latch(spi_tx_sel[3:0] & {4{wb_we_i}}), .byte_sel(wb_sel_i), .lsb(lsb),
                  .go(go), .pos_edge(pos_edge), .neg_edge(neg_edge), .rx_negedge(rx_negedge),
                  .tx_negedge(tx_negedge), .tip(tip), .last(last_bit),.p_in(wb_dat_i), .p_out(rx),
```

```
                         .s_clk(sclk_pad_o), .s_in(miso_pad_i), .s_out(mosi_pad_o));
                         //.scan_in0(scan_in0), .scan_en(scan_en), .test_mode(test_mode), .scan_out0(scan_out0), .reset(reset), .clk(clk));
/*----------------------------------------------Address decoder----------------------------------------------*/
  assign spi_divider_sel = wb_cyc_i & wb_stb_i & (wb_adr_i[`SPI_OFS_BITS] == `SPI_DIVIDE);
  assign spi_ctrl_sel    = wb_cyc_i & wb_stb_i & (wb_adr_i[`SPI_OFS_BITS] == `SPI_CTRL);
  assign spi_tx_sel[0]   = wb_cyc_i & wb_stb_i & (wb_adr_i[`SPI_OFS_BITS] == `SPI_TX_0);
  assign spi_tx_sel[1]   = wb_cyc_i & wb_stb_i & (wb_adr_i[`SPI_OFS_BITS] == `SPI_TX_1);
  assign spi_tx_sel[2]   = wb_cyc_i & wb_stb_i & (wb_adr_i[`SPI_OFS_BITS] == `SPI_TX_2);
  assign spi_tx_sel[3]   = wb_cyc_i & wb_stb_i & (wb_adr_i[`SPI_OFS_BITS] == `SPI_TX_3);
  assign spi_ss_sel      = wb_cyc_i & wb_stb_i & (wb_adr_i[`SPI_OFS_BITS] == `SPI_SS);
/*---------------------------------------Read from registers---------------------------------------*/
  always @(wb_adr_i or rx or ctrl or divider or ss)
  begin
```

```
    case (wb_adr_i[`SPI_OFS_BITS])
`ifdef SPI_MAX_CHAR_128
            `SPI_RX_0:    wb_dat = rx[31:0];
            `SPI_RX_1:    wb_dat = rx[63:32];
            `SPI_RX_2:    wb_dat = rx[95:64];
            `SPI_RX_3:    wb_dat = {{128-`SPI_MAX_CHAR{1'b0}}, rx
              [`SPI_MAX_CHAR-1:96]};
`else
`ifdef SPI_MAX_CHAR_64
            `SPI_RX_0:    wb_dat = rx[31:0];
            `SPI_RX_1:    wb_dat = {{64-`SPI_MAX_CHAR{1'b0}}, rx
              [`SPI_MAX_CHAR-1:32]};
            `SPI_RX_2:    wb_dat = 32'b0;
            `SPI_RX_3:    wb_dat = 32'b0;
`else
            `SPI_RX_0:    wb_dat = {{32-`SPI_MAX_CHAR{1'b0}}, rx
              [`SPI_MAX_CHAR-1:0]};
            `SPI_RX_1:    wb_dat = 32'b0;
            `SPI_RX_2:    wb_dat = 32'b0;
            `SPI_RX_3:    wb_dat = 32'b0;
`endif
`endif
            `SPI_CTRL:    wb_dat = {{32-`SPI_CTRL_BIT_NB{1'b0}},
              ctrl};
```

```
            `SPI_DIVIDE:   wb_dat = {{32-`SPI_DIVIDER_LEN{1'b0}}, divider};
            `SPI_SS:       wb_dat = {{32-`SPI_SS_NB{1'b0}}, ss};
        default:
      wb_dat = 32'bx;
      endcase
    end
/*--------------------------------Wb data out--------------------------------*/
    always @(posedge wb_clk_i or posedge wb_rst_i)
    begin
      if (wb_rst_i)
        wb_dat_o <=  32'b0;
      else
        wb_dat_o <=  wb_dat;
    end
/*--------------------------------Wb acknowledge--------------------------------*/
    always @(posedge wb_clk_i or posedge wb_rst_i)
    begin
      if (wb_rst_i)
        wb_ack_o <=  1'b0;
      else
        wb_ack_o <=  wb_cyc_i & wb_stb_i & ~wb_ack_o;
    end
```

```
/*------------------------------------------Wb error------------------------------------------*/
  assign wb_err_o = 1'b0;
/*------------------------------------------Interrupt------------------------------------------*/
  always @(posedge wb_clk_i or posedge wb_rst_i)
  begin
    if (wb_rst_i)
      wb_int_o <= 1'b0;
    else if (ie && tip && last_bit && pos_edge)
      wb_int_o <= 1'b1;
    else if (wb_ack_o)
      wb_int_o <= 1'b0;
  end
/*----------------------------------------Divider register----------------------------------------*/
  always @(posedge wb_clk_i or posedge wb_rst_i)
  begin
    if (wb_rst_i)
        divider <= {`SPI_DIVIDER_LEN{1'b0}};
    else if (spi_divider_sel && wb_we_i && !tip)
      begin
        `ifdef SPI_DIVIDER_LEN_8
          if (wb_sel_i[0])
              divider <= wb_dat_i[`SPI_DIVIDER_LEN-1:0];
```

```
        `endif
        `ifdef SPI_DIVIDER_LEN_16
          if (wb_sel_i[0])
              divider[7:0] <=  wb_dat_i[7:0];
          if (wb_sel_i[1])
              divider[`SPI_DIVIDER_LEN-1:8] <=  wb_dat_i[`
                SPI_DIVIDER_LEN-1:8];
        `endif
      `ifdef SPI_DIVIDER_LEN_24
          if (wb_sel_i[0])
              divider[7:0] <=  wb_dat_i[7:0];
          if (wb_sel_i[1])
              divider[15:8] <=  wb_dat_i[15:8];
          if (wb_sel_i[2])
              divider[`SPI_DIVIDER_LEN-1:16] <=  wb_dat_i[`
                SPI_DIVIDER_LEN-1:16];
        `endif
        `ifdef SPI_DIVIDER_LEN_32
          if (wb_sel_i[0])
              divider[7:0] <=  wb_dat_i[7:0];
          if (wb_sel_i[1])
              divider[15:8] <= wb_dat_i[15:8];
          if (wb_sel_i[2])
              divider[23:16] <=  wb_dat_i[23:16];
          if (wb_sel_i[3])
```

```
            divider[`SPI_DIVIDER_LEN-1:24] <= wb_dat_i[`SPI_DIVIDER_LEN-1:24];
    `endif
    end
end
/*-----------------------------------Ctrl register---------------------------------------------*/
always @(posedge wb_clk_i or posedge wb_rst_i)
begin
  if (wb_rst_i)
    ctrl <= {`SPI_CTRL_BIT_NB{1'b0}};
  else if(spi_ctrl_sel && wb_we_i && !tip)
    begin
      if (wb_sel_i[0])
        ctrl[7:0] <= wb_dat_i[7:0] | {7'b0, ctrl[0]};
      if (wb_sel_i[1])
        ctrl[`SPI_CTRL_BIT_NB-1:8] <= wb_dat_i[`SPI_CTRL_BIT_NB-1:8];
    end
  else if(tip && last_bit && pos_edge)
    ctrl[`SPI_CTRL_GO] <= 1'b0;
end
/*-----------------------------------Ctrl register decode-------------------------------------*/
assign rx_negedge = ctrl[`SPI_CTRL_RX_NEGEDGE];
```

```
  assign tx_negedge = ctrl[`SPI_CTRL_TX_NEGEDGE];
  assign go         = ctrl[`SPI_CTRL_GO];
  assign char_len   = ctrl[`SPI_CTRL_CHAR_LEN];
  assign lsb        = ctrl[`SPI_CTRL_LSB];
  assign ie         = ctrl[`SPI_CTRL_IE];
  assign ass        = ctrl[`SPI_CTRL_ASS];
/*----------------------------------Slave select register----------------------------------*/
  always @(posedge wb_clk_i or posedge wb_rst_i)
  begin
    if (wb_rst_i)
      ss <= {`SPI_SS_NB{1'b0}};
    else if(spi_ss_sel && wb_we_i && !tip)
        begin
          `ifdef SPI_SS_NB_8
            if (wb_sel_i[0])
                ss <= wb_dat_i[`SPI_SS_NB-1:0];
          `endif
          `ifdef SPI_SS_NB_16
            if (wb_sel_i[0])
                ss[7:0] <= wb_dat_i[7:0];
            if (wb_sel_i[1])
                ss[`SPI_SS_NB-1:8] <= wb_dat_i[`SPI_SS_NB-1:8];
          `endif
```

```
              `ifdef SPI_SS_NB_24
                if (wb_sel_i[0])
                    ss[7:0] <= wb_dat_i[7:0];
                if (wb_sel_i[1])
                    ss[15:8] <= wb_dat_i[15:8];
                if (wb_sel_i[2])
                    ss[`SPI_SS_NB-1:16] <= wb_dat_i[`SPI_SS_NB
                       -1:16];
              `endif
              `ifdef SPI_SS_NB_32
                if (wb_sel_i[0])
                    ss[7:0] <= wb_dat_i[7:0];
                if (wb_sel_i[1])
                    ss[15:8] <= wb_dat_i[15:8];
                if (wb_sel_i[2])
                    ss[23:16] <= wb_dat_i[23:16];
                if (wb_sel_i[3])
                    ss[`SPI_SS_NB-1:24] <= wb_dat_i[`SPI_SS_NB
                       -1:24];
              `endif
          end
      end
//------------------------------------------------------------------------
```

```
  assign ss_pad_o = ~((ss & {`SPI_SS_NB{tip & ass}}) | (ss & {`SPI_SS_NB{!ass}}));
//----------------------------------------------------------------------------------------

endmodule
//----------------------------------------------------------------------------------------
```

I.2 SPI Clock

```
/*
 * Author:  Deepak Siddharth Parthipan
 *          RIT, NY, USA
 * Module:  spi_clock
 */
//-----------------------------------------------------------------------------------------
`include "src/spi_defines.v"
`include "src/timescale.v"
//-----------------------------------------------------------------------------------------
module spi_clock_gen (clk_in, rst, go, enable, last_clk,
    divider, clk_out, pos_edge, neg_edge);
// scan_in0, scan_en, test_mode, scan_out0 ,reset, clk);
  input                    clk_in;   // input clock (
      system clock)
  input                    rst;      // reset
  input                    enable;   // clock enable
  input                    go;       // start transfer
  input                    last_clk; // last clock
```

```
  input      [`SPI_DIVIDER_LEN-1:0] divider;  // clock divider (output clock is divided by this value)
  output                            clk_out;  // output clock
  output                            pos_edge; // pulse marking positive edge of clk_out
  output                            neg_edge; // pulse marking negative edge of clk_out

  reg                               clk_out;
  reg                               pos_edge;
  reg                               neg_edge;
  reg        [`SPI_DIVIDER_LEN-1:0] cnt;      // clock counter
  wire                              cnt_zero; // conter is equal to zero
  wire                              cnt_one;  // conter is equal to one
//--------------------------------------------------------------------------------------------
  assign cnt_zero = cnt == {`SPI_DIVIDER_LEN{1'b0}};
  assign cnt_one  = cnt == {{`SPI_DIVIDER_LEN-1{1'b0}}, 1'b1};
/*--------------------------------Counter counts half period--------------------------------*/
  always @(posedge clk_in or posedge rst)
  begin
```

```
    if(rst)
      cnt <= {`SPI_DIVIDER_LEN{1'b1}};
    else
      begin
        if(!enable || cnt_zero)
          cnt <= divider;
        else
          cnt <=  cnt - {{`SPI_DIVIDER_LEN-1{1'b0}}, 1'b1};
      end
  end
/*----------------clk_out is asserted every other half period ----------------------*/
  always @(posedge clk_in or posedge rst)
  begin
    if(rst)
      clk_out <= 1'b0;
    else
      clk_out <= (enable && cnt_zero && (!last_clk || clk_out)) ? ~clk_out : clk_out;
  end
/*------------------------ Pos and neg edge signals ---------------------------------*/
  always @(posedge clk_in or posedge rst)
  begin
    if(rst)
```

```
        begin
          pos_edge  <=  1'b0;
          neg_edge  <=  1'b0;
        end
      else
        begin
          pos_edge  <=  (enable && !clk_out && cnt_one) || (!(|
             divider) && clk_out) || (!(|divider) && go && !enable
             );
          neg_edge  <=  (enable && clk_out && cnt_one) || (!(|
             divider) && !clk_out && enable);
        end
    end
//
   ------------------------------------------------------------------------
endmodule
//
   ------------------------------------------------------------------------
```

I.3 SPI Shift

```
/*
 * Author:   Deepak Siddharth Parthipan
 *           RIT, NY, USA
 * Module:   spi_shift
 */
//
    ______________________________________________________________

`include "src/spi_defines.v"
`include "src/timescale.v"
//
    ______________________________________________________________

module spi_shift (clk_shift, rst, latch, byte_sel, len, lsb, go
    , pos_edge, neg_edge, rx_negedge, tx_negedge, tip, last,
                        p_in, p_out, s_clk, s_in, s_out); //scan_in0,
                            scan_en, test_mode, scan_out0 ,reset, clk)
                            ;
//
    ______________________________________________________________

  input                              clk_shift;      // system clock
  input                              rst;            // reset
```

```
input                    [3:0] latch;      // latch signal for storing the data in shift register
input                    [3:0] byte_sel;   // byte select signals for storing the data in shift register
input [`SPI_CHAR_LEN_BITS-1:0] len;        // data len in bits (minus one)
input                          lsb;        // lbs first on the line
input                          go;         // start stansfer
input                          pos_edge;   // recognize posedge of sclk
input                          neg_edge;   // recognize negedge of sclk
input                          rx_negedge; // s_in is sampled on negative edge
input                          tx_negedge; // s_out is driven on negative edge
output                         tip;        // transfer in progress
output                         last;       // last bit
input                   [31:0] p_in;       // parallel in
output     [`SPI_MAX_CHAR-1:0] p_out;      // parallel out
input                          s_clk;      // serial clock
input                          s_in;       // serial in
```

```
  output                                s_out;        // serial out
  reg                                   s_out;
  reg                                   tip;
  reg     [`SPI_CHAR_LEN_BITS:0]        cnt;          // data bit count
  reg     [`SPI_MAX_CHAR-1:0]           data;         // shift register
  wire    [`SPI_CHAR_LEN_BITS:0]        tx_bit_pos;   // next bit position
  wire    [`SPI_CHAR_LEN_BITS:0]        rx_bit_pos;   // next bit position
  wire                                  rx_clk;       // rx clock enable
  wire                                  tx_clk;       // tx clock enable
//----------------------------------------------------------------------------

  assign p_out = data;
  assign tx_bit_pos = lsb ? {!(|len), len} - cnt : cnt - {{`SPI_CHAR_LEN_BITS{1'b0}},1'b1};
  assign rx_bit_pos = lsb ? {!(|len), len} - (rx_negedge ? cnt + {{`SPI_CHAR_LEN_BITS{1'b0}},1'b1} : cnt) :
                            (rx_negedge ? cnt : cnt - {{`SPI_CHAR_LEN_BITS{1'b0}},1'b1});
```

```

  assign last = !(|cnt);
  assign rx_clk = (rx_negedge ? neg_edge : pos_edge) && (!last || s_clk);
  assign tx_clk = (tx_negedge ? neg_edge : pos_edge) && !last;
/*--------------------------------Character bit counter--------------------------------*/
  always @(posedge clk_shift or posedge rst)
  begin
    if(rst)
      cnt <=  {`SPI_CHAR_LEN_BITS+1{1'b0}};
    else
      begin
        if(tip)
          cnt <=  pos_edge ? (cnt - {{`SPI_CHAR_LEN_BITS{1'b0}}, 1'b1}) : cnt;
        else
          cnt <=  !(|len) ? {1'b1, {`SPI_CHAR_LEN_BITS{1'b0}}} : {1'b0, len};
      end
  end
/*--------------------------------Transfer in progress--------------------------------*/
  always @(posedge clk_shift or posedge rst)
  begin
```

```
      if(rst)
        tip <=  1'b0;
    else if(go && ~tip)
      tip <=  1'b1;
    else if(tip && last && pos_edge)
      tip <=  1'b0;
    end
/*------------------------Sending bits to the line----------------------------------*/
    always @(posedge clk_shift or posedge rst)
    begin
      if (rst)
        s_out   <=  1'b0;
      else
        s_out <=  (tx_clk || !tip) ? data[tx_bit_pos[`SPI_CHAR_LEN_BITS-1:0]] : s_out;
    end
/*--------------------Receiving bits from the line----------------------------------*/
    always @(posedge clk_shift or posedge rst)
    begin
      if (rst)
        data   <=  {`SPI_MAX_CHAR{1'b0}};

`ifdef SPI_MAX_CHAR_128
```

```
    else if (latch[0] && !tip)
      begin
        if (byte_sel[3])
          data[31:24] <= p_in[31:24];
        if (byte_sel[2])
          data[23:16] <= p_in[23:16];
        if (byte_sel[1])
          data[15:8] <= p_in[15:8];
        if (byte_sel[0])
          data[7:0] <= p_in[7:0];
      end
    else if (latch[1] && !tip)
      begin
        if (byte_sel[3])
          data[63:56] <= p_in[31:24];
        if (byte_sel[2])
          data[55:48] <= p_in[23:16];
        if (byte_sel[1])
          data[47:40] <= p_in[15:8];
        if (byte_sel[0])
          data[39:32] <= p_in[7:0];
      end
    else if (latch[2] && !tip)
      begin
        if (byte_sel[3])
```

```
            data[95:88] <= p_in[31:24];
          if (byte_sel[2])
            data[87:80] <= p_in[23:16];
          if (byte_sel[1])
            data[79:72] <= p_in[15:8];
          if (byte_sel[0])
            data[71:64] <= p_in[7:0];
        end
      else if (latch[3] && !tip)
        begin
          if (byte_sel[3])
            data[127:120] <= p_in[31:24];
          if (byte_sel[2])
            data[119:112] <= p_in[23:16];
          if (byte_sel[1])
            data[111:104] <= p_in[15:8];
          if (byte_sel[0])
            data[103:96] <= p_in[7:0];
        end
`else

`ifdef SPI_MAX_CHAR_64
      else if (latch[0] && !tip)
        begin
          if (byte_sel[3])
```

```
          data[31:24] <= p_in[31:24];
        if (byte_sel[2])
          data[23:16] <= p_in[23:16];
        if (byte_sel[1])
          data[15:8] <= p_in[15:8];
        if (byte_sel[0])
          data[7:0] <= p_in[7:0];
      end
    else if (latch[1] && !tip)
      begin
        if (byte_sel[3])
          data[63:56] <= p_in[31:24];
        if (byte_sel[2])
          data[55:48] <= p_in[23:16];
        if (byte_sel[1])
          data[47:40] <= p_in[15:8];
        if (byte_sel[0])
          data[39:32] <= p_in[7:0];
      end
`else
    else if (latch[0] && !tip)
      begin
      `ifdef SPI_MAX_CHAR_8
        if (byte_sel[0])
          data[`SPI_MAX_CHAR-1:0] <= p_in[`SPI_MAX_CHAR-1:0];
```

```
    `endif
    `ifdef SPI_MAX_CHAR_16
      if (byte_sel[0])
        data[7:0] <= p_in[7:0];
      if (byte_sel[1])
        data[`SPI_MAX_CHAR-1:8] <= p_in[`SPI_MAX_CHAR-1:8];
    `endif
    `ifdef SPI_MAX_CHAR_24
      if (byte_sel[0])
        data[7:0] <= p_in[7:0];
      if (byte_sel[1])
        data[15:8] <= p_in[15:8];
      if (byte_sel[2])
        data[`SPI_MAX_CHAR-1:16] <= p_in[`SPI_MAX_CHAR
           -1:16];
    `endif
    `ifdef SPI_MAX_CHAR_32
      if (byte_sel[0])
        data[7:0] <= p_in[7:0];
      if (byte_sel[1])
        data[15:8] <= p_in[15:8];
      if (byte_sel[2])
        data[23:16] <= p_in[23:16];
      if (byte_sel[3])
```

```
                data[`SPI_MAX_CHAR-1:24] <= p_in[`SPI_MAX_CHAR
                    -1:24];
          `endif
          end
`endif
`endif
      else
          data[rx_bit_pos[`SPI_CHAR_LEN_BITS-1:0]] <= rx_clk ?
              s_in : data[rx_bit_pos[`SPI_CHAR_LEN_BITS-1:0]];
   end
//
  ---------------------------------------------------------------------

endmodule
//
  ---------------------------------------------------------------------
```

I.4 Defines

```
/*
 * Author:   Deepak Siddharth Parthipan
 *           RIT, NY, USA
 * Module:   spi_defines
 */
//----------------------------------------------------------------------------------------
/*
 Number of bits used for divider register. If used in system with
 low frequency of system clock this can be reduced.
 Use SPI_DIVIDER_LEN for fine tuning the exact number.
*/

//`define SPI_DIVIDER_LEN_8
`define SPI_DIVIDER_LEN_16
//`define SPI_DIVIDER_LEN_24
//`define SPI_DIVIDER_LEN_32

`ifdef SPI_DIVIDER_LEN_8
  `define SPI_DIVIDER_LEN       8    // Can be set from 1 to 8
`endif
```

```
`ifdef SPI_DIVIDER_LEN_16
  `define SPI_DIVIDER_LEN       16    // Can be set from 9 to 16
`endif
`ifdef SPI_DIVIDER_LEN_24
  `define SPI_DIVIDER_LEN       24    // Can be set from 17 to 24
`endif
`ifdef SPI_DIVIDER_LEN_32
  `define SPI_DIVIDER_LEN       32    // Can be set from 25 to 32
`endif
//

/*
 Maximum nuber of bits that can be send/received at once.
 Use SPI_MAX_CHAR for fine tuning the exact number, when using
 SPI_MAX_CHAR_32, SPI_MAX_CHAR_24, SPI_MAX_CHAR_16, SPI_MAX_CHAR_8.
*/

`define SPI_MAX_CHAR_128
//`define SPI_MAX_CHAR_64
//`define SPI_MAX_CHAR_32
//`define SPI_MAX_CHAR_24
```

```
//`define SPI_MAX_CHAR_16
//`define SPI_MAX_CHAR_8

`ifdef SPI_MAX_CHAR_128
  `define SPI_MAX_CHAR           128  // Can only be set to 128
  `define SPI_CHAR_LEN_BITS      7
`endif
`ifdef SPI_MAX_CHAR_64
  `define SPI_MAX_CHAR           64   // Can only be set to 64
  `define SPI_CHAR_LEN_BITS      6
`endif
`ifdef SPI_MAX_CHAR_32
  `define SPI_MAX_CHAR           32   // Can be set from 25 to 32
  `define SPI_CHAR_LEN_BITS      5
`endif
`ifdef SPI_MAX_CHAR_24
  `define SPI_MAX_CHAR           24   // Can be set from 17 to 24
  `define SPI_CHAR_LEN_BITS      5
`endif
`ifdef SPI_MAX_CHAR_16
  `define SPI_MAX_CHAR           16   // Can be set from 9 to 16
  `define SPI_CHAR_LEN_BITS      4
`endif
```

```
`ifdef SPI_MAX_CHAR_8
  `define SPI_MAX_CHAR           8     // Can be set from 1 to 8
  `define SPI_CHAR_LEN_BITS      3
`endif
//----------------------------------------------------------------------------

/*
 Number of device select signals. Use SPI_SS_NB for fine tuning the
 exact number.
*/
`define SPI_SS_NB_8
//`define SPI_SS_NB_16
//`define SPI_SS_NB_24
//`define SPI_SS_NB_32

`ifdef SPI_SS_NB_8
  `define SPI_SS_NB              8     // Can be set from 1 to 8
`endif
`ifdef SPI_SS_NB_16
  `define SPI_SS_NB              16    // Can be set from 9 to 16
`endif
`ifdef SPI_SS_NB_24
```

```
  `define SPI_SS_NB            24    // Can be set from 17 to 24
`endif
`ifdef SPI_SS_NB_32
  `define SPI_SS_NB            32    // Can be set from 25 to 32
`endif
//----------------------------------------------------------------------------

/*
 Bits of WISHBONE address used for partial decoding of SPI registers.
*/
`define SPI_OFS_BITS           4:2
//----------------------------------------------------------------------------

/* Register offset */
`define SPI_RX_0               0
`define SPI_RX_1               1
`define SPI_RX_2               2
`define SPI_RX_3               3
`define SPI_TX_0               0
`define SPI_TX_1               1
```

```
`define SPI_TX_2            2
`define SPI_TX_3            3
`define SPI_CTRL            4
`define SPI_DIVIDE          5
`define SPI_SS              6
//

/* Number of bits in ctrl register */
`define SPI_CTRL_BIT_NB     14
//

/* Control register bit position */
`define SPI_CTRL_ASS        13
`define SPI_CTRL_IE         12
`define SPI_CTRL_LSB        11
`define SPI_CTRL_TX_NEGEDGE 10
`define SPI_CTRL_RX_NEGEDGE 9
`define SPI_CTRL_GO         8
`define SPI_CTRL_RES_1      7
`define SPI_CTRL_CHAR_LEN   6:0
//
```

I.5 Test Top

```
/*
 * Author:  Deepak Siddharth Parthipan
 *          RIT, NY, USA
 * Module:  tb_top
 */
//-----------------------------------------------------------------------
`include "uvm_macros.svh"
`include "spi_pkg.sv"
`include "spi_if.sv"
//-----------------------------------------------------------------------
module test;
  import uvm_pkg::*;
  import spi_pkg::*;

  spi_if master(clock);      //Interface declaration
    spi_if slave(clock);      //Interface declaration
/*-----------------SPI master core-------------------------*/
  spi top (
    /*tb to DUT connection*/
    .wb_clk_i(clock),
    .wb_rst_i(rstn),
    .wb_adr_i(master.adr[4:0]),
```

```
    .wb_dat_i(master.dout),
    .wb_sel_i(master.sel),
    .wb_we_i(master.we),
    .wb_stb_i(master.stb),
    .wb_cyc_i(master.cyc),
    .wb_dat_o(master.din),
    .wb_ack_o(master.ack),
    .wb_err_o(master.err),
    .wb_int_o(master.intp),
    .scan_in0(scan_in0),
    .scan_out0(scan_out0),
    .scan_en(scan_en),
    .test_mode(test_mode),
    /*master to slave connection*/
    .ss_pad_o(ss),
    .sclk_pad_o(sclk),
    .mosi_pad_o(mosi),
    .miso_pad_i(miso),
    .tip(master.pit)
    );
/*------------------------SPI slave core-----------------------------------*/
  spi_slave spi_slave (
    /*tb to DUT connection*/
    .wb_clk_i(clock),
    .wb_rst_i(rstn),
```

```
      .wb_adr_i(slave.adr[4:0]),
      .wb_dat_i(slave.dout),
      .wb_sel_i(slave.sel),
      .wb_we_i(slave.we),
      .wb_stb_i(slave.stb),
      .wb_cyc_i(slave.cyc),
      .wb_dat_o(slave.din),
      .wb_ack_o(slave.ack),
      .wb_err_o(slave.err),
      .wb_int_o(slave.intp),
      .scan_in0(scan_in0),
      .scan_en(scan_en),
      .test_mode(test_mode),
      .scan_out0(scan_out0),
       /*slave to master connection*/
      .ss_pad_i(ss),
      .sclk_pad_i(sclk),
      .mosi_pad_i(mosi),
      .miso_pad_o(miso)
   );
//------------------------------------------------------------
   initial begin
         $timeformat(-9,2,"ns", 16);
         $set_coverage_db_name("spi");

```

```
            `ifdef SDFSCAN
                $sdf_annotate("sdf/spi_tsmc18_scan.sdf", test.top);
            `endif
            generate_clock();
            reg_intf_to_config_db();
            initalize_dut();
            //reset_dut();            //could also be carried out inside pre_reset_phase
        run_test();
    end
//--------------------------------------------------------------------------------

    task generate_clock();
    fork
        forever begin
            clock = `LOW;
            #(CLOCK_PERIOD/2);
            clock = `HIGH;
            #(CLOCK_PERIOD/2);
        end
    join_none
    endtask : generate_clock
//--------------------------------------------------------------------------------
function void reg_intf_to_config_db();
```

```
//Registers the Interface in the configuration block so that other blocks can use it retrived using get
    uvm_config_db#(virtual spi_if)::set(null,"*","m_if",master);
        uvm_config_db#(virtual spi_if)::set(null,"*","s_if",slave);
endfunction : reg_intf_to_config_db
//--------------------------------------------------------------------
function void initalize_dut();
    test_mode  = 1'b0;
    scan_in0   = 1'b0;
    scan_in1   = 1'b0;
    scan_en    = 1'b0;
endfunction : initalize_dut
//--------------------------------------------------------------------
task reset_dut();
    rstn <= `LOW;
    repeat(RESET_PERIOD) @(posedge clock);
    rstn <= `HIGH;
    repeat(RESET_PERIOD) @(posedge clock);
    rstn = `LOW;
    //->RST_DONE;
endtask : reset_dut
//--------------------------------------------------------------------
endmodule : test
```

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

I.6 Interface

```
/*
 * Author:  Deepak Siddharth Parthipan
 *          RIT, NY, USA
 * Module:  Package
 */
//------------------------------------------------------------
interface spi_if(input bit clk);
//------------------------------------------------------------
  // Wishbone signals

  logic                 [4:0] adr;      // lower address
      bits
  logic              [32-1:0] din;      // databus input
  logic              [32-1:0] dout;     // databus output
  logic                 [3:0] sel;      // byte select
      inputs
  logic                       we;       // write enable
      input
  logic                       stb;      // stobe/core
      select signal
  logic                       cyc;      // valid bus
      cycle input
```

```
logic                                ack;        // bus cycle acknowledge output
logic                                err;        // termination w/ error
logic                                intp;        // interrupt request signal output  input
logic                                pit;
//--------------------------------------------------------------------------
   clocking drive_cb @(posedge clk);
   input  din, ack, err, intp, pit;
   output  adr, dout, sel, we, stb, cyc;
   endclocking : drive_cb
//--------------------------------------------------------------------------
   clocking monitor_cb @(posedge clk);
   input  din, ack, err, intp, pit;
   output  adr, dout, sel, we, stb, cyc;
   endclocking : monitor_cb
//--------------------------------------------------------------------------
endinterface : spi_if
//--------------------------------------------------------------------------
```

I.7 Package

```
/*
 * Author:  Deepak Siddharth Parthipan
 *          RIT, NY, USA
 * Module:  Package
 */
//---------------------------------------------------------------
package spi_pkg;
//---------------------------------------------------------------
  import uvm_pkg::*;

    //`include "uvm_macros.svh"
    `include "spi_tb_defines.sv"
    `include "spi_sequence_item.sv"
    `include "wb_bfm.sv"
  `include "spi_driver.sv"
  `include "spi_monitor.sv"
  `include "spi_sequencer.sv"
  `include "spi_agent.sv"
    `include "spi_coverage.sv"
  `include "spi_scoreboard.sv"
    `include "spi_sequence.sv"
  `include "spi_env.sv"
  `include "spi_test.sv"
```

```
//------------------------------------------------------------------------
endpackage: spi_pkg
//------------------------------------------------------------------------
```

I.8 Test

```
/*
 * Author:  Deepak Siddharth Parthipan
 *          RIT, NY, USA
 * Module:  Test
 */
//-----------------------------------------------------------------------
class spi_test extends uvm_test;
//-----------------------------------------------------------------------
    `uvm_component_utils(spi_test)
    spi_env env;
    spi_sequence h_seq;
//-----------------------------------------------------------------------
    function new(string name="spi_test",uvm_component parent);
        super.new(name,parent);
    endfunction: new
//-----------------------------------------------------------------------
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        `uvm_info(get_full_name(),"Build phase called in
           spi_test",UVM_LOW)
        /* Build environment component*/
        env = spi_env::type_id::create("env",this);
    endfunction: build_phase
```

```
//------------------------------------------------------------------
    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        `uvm_info(get_full_name(),"Connect phase called in
            spi_test",UVM_LOW)
    endfunction: connect_phase
//------------------------------------------------------------------
    task reset_phase(uvm_phase phase);
        phase.raise_objection(this);
        rstn <= `LOW;
        repeat(RESET_PERIOD) @(posedge clock);
        rstn <= `HIGH;
        repeat(RESET_PERIOD) @(posedge clock);
        rstn = `LOW;
        phase.drop_objection(this);
    endtask: reset_phase
//------------------------------------------------------------------
    virtual task main_phase(uvm_phase phase);
        `uvm_info(get_full_name(),"in main phase",UVM_LOW)
        phase.raise_objection(this);
        h_seq=spi_sequence::type_id::create("h_seq");
        repeat(100)
        h_seq.start(env.agent.sequencer);
        phase.drop_objection(this);
    endtask: main_phase
```

```
//----------------------------------------------------------------------------
endclass: spi_test
//----------------------------------------------------------------------------
```

I.9 Environment

```
/*
 * Author:   Deepak Siddharth Parthipan
 *           RIT, NY, USA
 * Module:   Environment
 */
//------------------------------------------------------------
class spi_env extends uvm_env;
//------------------------------------------------------------
    `uvm_component_utils(spi_env)
    spi_agent agent;
    spi_scoreboard scoreboard;
//------------------------------------------------------------
    function new(string name="spi_env",uvm_component parent);
        super.new(name,parent);
    endfunction: new
//------------------------------------------------------------
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
       `uvm_info(get_full_name(),"Build phase called in
         spi_environment",UVM_LOW)
        /* Build agent and scoreboard components*/
        agent = spi_agent::type_id::create("agent",this);
```

```
        scoreboard = spi_scoreboard::type_id::create("scoreboard",this);
    endfunction: build_phase
//------------------------------------------------------------------------
    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        `uvm_info(get_full_name(),"Connect phase called in spi_environment",UVM_LOW)
        /* Connect the analysis port for monitor and driver respectively with scorboard */
        agent.monitor.dut_out_pkt.connect(scoreboard.mon2sb);
        agent.driver.dut_in_pkt.connect(scoreboard.drv2sb);
    endfunction: connect_phase
//------------------------------------------------------------------------
endclass: spi_env
//------------------------------------------------------------------------
```

I.10 Agent

```
/*
 * Author:  Deepak Siddharth Parthipan
 *          RIT, NY, USA
 * Module:  Agent
 */
//-----------------------------------------------------------
class spi_agent extends uvm_agent;
//-----------------------------------------------------------
    `uvm_component_utils(spi_agent)
    spi_sequencer sequencer;
    spi_monitor monitor;
    spi_driver driver;
    spi_vif m_vif,s_vif;
//-----------------------------------------------------------
    function new(string name="spi_agent",uvm_component parent);
        super.new(name,parent);
    endfunction: new
//-----------------------------------------------------------
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        `uvm_info(get_full_name(),"Build phase called in
           spi_agent",UVM_LOW)
```

```
        if(!uvm_config_db#(virtual spi_if)::get(this, "", "m_if", m_vif))
        `uvm_fatal("NO_VIF",{"virtual interface must be set for: ",get_full_name(),".m_vif"})
        if(!uvm_config_db#(virtual spi_if)::get(this, "", "s_if", s_vif))
        `uvm_fatal("NO_VIF",{"virtual interface must be set for: ",get_full_name(),".s_vif"})
        sequencer = spi_sequencer::type_id::create("sequencer",this);
        driver = spi_driver::type_id::create("driver",this);
        driver.m_vif = m_vif;
        driver.s_vif = s_vif;
        monitor = spi_monitor::type_id::create("monitor",this);
        monitor.m_vif = m_vif;
        monitor.s_vif = s_vif;
    endfunction: build_phase
//--------------------------------------------------------------------------
    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        `uvm_info(get_full_name(),"Connect phase called in spi_agent",UVM_LOW)
        driver.seq_item_port.connect(sequencer.seq_item_export);
    endfunction: connect_phase
```

```
//------------------------------------------------------------------------------
endclass: spi_agent
//------------------------------------------------------------------------------
```

I.11 Sequence Item

```
/*
 * Author:   Deepak Siddharth Parthipan
 *           RIT, NY, USA
 * Module:   Sequence Item
 */
//------------------------------------------------------------
class spi_sequence_item extends uvm_sequence_item;
//------------------------------------------------------------
    /*Register configuration*/
    rand logic [31:0] master_ctrl_reg;
    rand logic [31:0] slave_ctrl_reg;
    rand logic [31:0] divider_reg;
    rand logic [31:0] slave_select_reg;
    rand logic [31:0] start_dut_reg;
    /*DUT output*/
    logic [31:0] out_master_data;
    logic [31:0] out_slave_data;
    /*Expected data*/
    rand logic [31:0] exp_master_data;
    rand logic [31:0] exp_slave_data;
    /*DUT input*/
    rand logic [31:0] in_master_data;
    rand logic [31:0] in_slave_data;
```

```
    logic [31:0] q;
//----------------------------------------------------------------
    `uvm_object_utils_begin(spi_sequence_item)
      `uvm_field_int(master_ctrl_reg,UVM_ALL_ON)
      `uvm_field_int(slave_ctrl_reg,UVM_ALL_ON)
      `uvm_field_int(divider_reg,UVM_ALL_ON)
      `uvm_field_int(slave_select_reg,UVM_ALL_ON)
      `uvm_field_int(start_dut_reg,UVM_ALL_ON)
      `uvm_field_int(out_master_data,UVM_ALL_ON)
      `uvm_field_int(out_slave_data,UVM_ALL_ON)
      `uvm_field_int(exp_master_data,UVM_ALL_ON)
      `uvm_field_int(exp_slave_data,UVM_ALL_ON)
      `uvm_field_int(in_master_data,UVM_ALL_ON)
      `uvm_field_int(in_slave_data,UVM_ALL_ON)
      `uvm_field_int(q,UVM_ALL_ON)
    `uvm_object_utils_end
//----------------------------------------------------------------
    function new(string name="spi_sequence_item");
        super.new(name);
    endfunction: new
//----------------------------------------------------------------
endclass: spi_sequence_item
//----------------------------------------------------------------
```

I.12 Sequence

```
/*
 * Author:  Deepak Siddharth Parthipan
 *          RIT, NY, USA
 * Module:  Sequence
 */
//-----------------------------------------------------------------
class spi_sequence extends uvm_sequence #(spi_sequence_item);
//-----------------------------------------------------------------
    `uvm_object_utils(spi_sequence)
//-----------------------------------------------------------------
    function new(string name="spi_sequence");
        super.new(name);
    endfunction: new
//-----------------------------------------------------------------
    virtual task body();
        req=spi_sequence_item :: type_id :: create("req");
        start_item(req);
        //configure_dut_register();
        set_dut_data();
        finish_item(req);
    endtask: body
//-----------------------------------------------------------------
    virtual function void configure_dut_register();
```

```
    assert(req.randomize() with {  req.master_ctrl_reg == 32'h00002208;
                                   req.slave_ctrl_reg == 32'h00000200;
                                   req.divider_reg == 32'h00000000;
                                   req.slave_select_reg == 32'h00000001;
                                   req.start_dut_reg == 32'h00000320;
                                 });
  endfunction: configure_dut_register
//-----------------------------------------------------------------
  virtual function void set_dut_data();
    assert(req.randomize() with {
                                   req.divider_reg == 32'h00000000;
                                   req.master_ctrl_reg == 32'h00002208;
                                   req.slave_ctrl_reg == 32'h00000200;
                                   req.slave_select_reg == 32'h00000001;
                                   req.start_dut_reg == 32'h00000320;
```

```
                                        //req.in_master_data == 32'h87654321;
                                        //req.in_slave_data == 32'h11223344;
                                        req.exp_master_data == req.in_slave_data;
                                        req.exp_slave_data == req.in_master_data;
                                    });
    endfunction: set_dut_data
//---------------------------------------------------------------------------
endclass: spi_sequence
//---------------------------------------------------------------------------
```

I.13 Sequencer

```
/*
 * Author:  Deepak Siddharth Parthipan
 *          RIT, NY, USA
 * Module:  Sequencer
 */
//------------------------------------------------------------
class spi_sequencer extends uvm_sequencer #(spi_sequence_item);
//------------------------------------------------------------
    `uvm_component_utils(spi_sequencer)
//------------------------------------------------------------
    function new(string name="spi_sequencer",uvm_component
        parent);
        super.new(name,parent);
    endfunction: new
//------------------------------------------------------------
endclass: spi_sequencer
//------------------------------------------------------------
```

I.14 Driver

```
/*
 * Author:   Deepak Siddharth Parthipan
 *           RIT, NY, USA
 * Module:   Driver
 */
//------------------------------------------------------------
class spi_driver extends uvm_driver #(spi_sequence_item);
//------------------------------------------------------------
    `uvm_component_utils(spi_driver)
     spi_vif m_vif, s_vif;
     spi_sequence_item packet;
     uvm_analysis_port #(spi_sequence_item) dut_in_pkt;
//------------------------------------------------------------
    function new(string name="spi_monitor",uvm_component parent
      );
        super.new(name,parent);
        dut_in_pkt = new("dut_in_pkt",this);
    endfunction: new
//------------------------------------------------------------
     function void build_phase(uvm_phase phase);
        super.build_phase(phase);
       `uvm_info(get_full_name(),"Build phase called in
          spi_driver",UVM_LOW)
```

```
        if(!uvm_config_db#(virtual spi_if)::get(this, "", "m_if", m_vif))
        `uvm_fatal("NO_VIF",{"virtual interface must be set for: ",get_full_name(),".m_vif"})
        if(!uvm_config_db#(virtual spi_if)::get(this, "", "s_if", s_vif))
        `uvm_fatal("NO_VIF",{"virtual interface must be set for: ",get_full_name(),".s_vif"})
    endfunction: build_phase
//------------------------------------------------------------------------
    task run_phase(uvm_phase phase);
        packet = spi_sequence_item :: type_id :: create("packet");
        wb_bfm::wb_reset(m_vif);
        wb_bfm::wb_reset(s_vif);
        fork
            forever begin
            seq_item_port.get_next_item(req);
            drive_transfer(req);
            $cast(packet,req.clone());
            packet = req;
            dut_in_pkt.write(packet);
            seq_item_port.item_done();
            wait(m_vif.monitor_cb.pit==1'b0);
            end
```

```
        join_none
    endtask: run_phase
//-----------------------------------------------------------------------------
    task drive_transfer(spi_sequence_item seq);
      wb_bfm::wb_write(m_vif, 0, SPI_DIVIDE, seq.divider_reg);       // set divider register
      wb_bfm::wb_write(m_vif, 0, SPI_SS, seq.slave_select_reg);      // set ss 0
      wb_bfm::wb_write(m_vif, 0, SPI_TX_0, seq.in_master_data);      // set master data register
      wb_bfm::wb_write(m_vif, 0, SPI_CTRL, seq.master_ctrl_reg);     // set master ctrl register
      wb_bfm::wb_write(s_vif, 0, SPI_CTRL, seq.slave_ctrl_reg);      // set slave ctrl register
      wb_bfm::wb_write(s_vif, 0, SPI_TX_0, seq.in_slave_data);       // set slave data register
      wb_bfm::wb_write(m_vif, 0, SPI_CTRL, seq.start_dut_reg);       // start data transfer
    endtask: drive_transfer
//-----------------------------------------------------------------------------
endclass: spi_driver
//-----------------------------------------------------------------------------
```

I.15 Monitor

```
/*
 * Author:  Deepak Siddharth Parthipan
 *          RIT, NY, USA
 * Module:  Monitor
 */
//------------------------------------------------------------------
class spi_monitor extends uvm_monitor;
//------------------------------------------------------------------
    `uvm_component_utils(spi_monitor)
    spi_vif m_vif, s_vif;
    spi_sequence_item packet;
    uvm_analysis_port #(spi_sequence_item) dut_out_pkt;
//------------------------------------------------------------------
    function new(string name="spi_monitor",uvm_component parent
      );
        super.new(name,parent);
        dut_out_pkt = new("dut_out_pkt",this);
    endfunction: new
//------------------------------------------------------------------
     function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        `uvm_info(get_full_name(),"Build phase called in
           spi_monitor",UVM_LOW)
```

```
        if(!uvm_config_db#(virtual spi_if)::get(this, "", "m_if", m_vif))
         `uvm_fatal("NO_VIF",{"virtual interface must be set for: ",get_full_name(),".m_vif"})
         if(!uvm_config_db#(virtual spi_if)::get(this, "", "s_if", s_vif))
         `uvm_fatal("NO_VIF",{"virtual interface must be set for: ",get_full_name(),".s_vif"})
    endfunction: build_phase
//-----------------------------------------------------------------------------------
    task run_phase(uvm_phase phase);
        packet = spi_sequence_item :: type_id :: create("packet");
        wait(m_vif.monitor_cb.pit==1'b1)     //wait_to_start
        forever begin
           wait(m_vif.monitor_cb.pit==1'b0) //wait_to_complete
           wb_bfm::wb_read(m_vif, 1, SPI_RX_0, packet.out_master_data);
           wb_bfm::wb_read(s_vif, 1, SPI_RX_0, packet.out_slave_data);
           dut_out_pkt.write(packet);
           wait(m_vif.monitor_cb.pit==1'b1); //wait_to_start
        end
    endtask: run_phase
//-----------------------------------------------------------------------------------
```

```
endclass: spi_monitor
//---------------------------------------------------------------------------
```

I.16 Wishbone Bus Funtion Model

```
/*
 * Author:   Deepak Siddharth Parthipan
 *           RIT, NY, USA
 * Module:   wishbone bus function
 */
//------------------------------------------------------------------------
class wb_bfm extends uvm_object;
//------------------------------------------------------------------------
    `uvm_object_utils(wb_bfm)
//------------------------------------------------------------------------
    function new(string name = "wb_bfm");
        super.new(name);
    endfunction: new
//------------------------------------------------------------------------
    static task wb_reset;
        input spi_vif vif;
        vif.adr  <= {awidth{1'bx}};
        vif.dout <= {dwidth{1'bx}};
        vif.cyc  <= 1'b0;
        vif.stb  <= 1'bx;
        vif.we   <= 1'hx;
        vif.sel  <= {dwidth/8{1'bx}};
    endtask: wb_reset
```

```
/*-------------------Wishbone read cycle----------------------------*/
  static task wb_read;
    input spi_vif vif;
    input integer delay;
    input logic [awidth -1:0] a;
    output logic [dwidth -1:0] d;

    begin
      // wait initial delay
      repeat(delay) @(vif.monitor_cb);
      // assert wishbone signals
      repeat(1) @(vif.monitor_cb);
      vif.monitor_cb.adr  <= a;
      vif.monitor_cb.dout <= {dwidth{1'bx}};
      vif.monitor_cb.cyc  <= 1'b1;
      vif.monitor_cb.stb  <= 1'b1;
      vif.monitor_cb.we   <= 1'b0;
      vif.monitor_cb.sel  <= {dwidth/8{1'b1}};
     @(vif.monitor_cb);
      // wait for acknowledge from slave
      wait(vif.monitor_cb.ack==1'b1)
      // negate wishbone signals
      repeat (1) @(vif.monitor_cb);
      vif.monitor_cb.cyc  <= 1'b0;
      vif.monitor_cb.stb  <= 1'bx;
```

```
        vif.monitor_cb.adr  <= {awidth{1'bx}};
        vif.monitor_cb.dout <= {dwidth{1'bx}};
        vif.monitor_cb.we   <= 1'hx;
        vif.monitor_cb.sel  <= {dwidth/8{1'bx}};
                      d     = vif.monitor_cb.din;

    end
  endtask : wb_read
/*------------------------Wishbone write cycle------------------------------------
  */
  static task wb_write;
    input spi_vif vif;
    input integer delay;
    input logic [awidth -1:0] a;
    input logic [dwidth -1:0] d;

    begin
      // wait initial delay
      repeat(delay) @(vif.drive_cb);
      // assert wishbone signal
      vif.drive_cb.adr  <= a;
      vif.drive_cb.dout <= d;
      vif.drive_cb.cyc  <= 1'b1;
      vif.drive_cb.stb  <= 1'b1;
      vif.drive_cb.we   <= 1'b1;
```

```
        vif.drive_cb.sel   <= {dwidth/8{1'b1}};
       @(vif.drive_cb);
        // wait for acknowledge from slave
        //@(vif.drive_cb);
        wait(vif.drive_cb.ack==1'b1)
        // negate wishbone signals
        repeat (2)
     @(vif.drive_cb);
        vif.drive_cb.cyc   <= 1'b0;
        vif.drive_cb.stb   <= 1'bx;
        vif.drive_cb.adr   <= {awidth{1'bx}};
        vif.drive_cb.dout  <= {dwidth{1'bx}};
        vif.drive_cb.we    <= 1'hx;
        vif.drive_cb.sel   <= {dwidth/8{1'bx}};
      end
   endtask : wb_write
//----------------------------------------------------------------------
endclass: wb_bfm
//----------------------------------------------------------------------
```

I.17 Scoreboard

```
/*
 * Author:  Deepak Siddharth Parthipan
 *          RIT, NY, USA
 * Module:  Scoreboard
 */
//------------------------------------------------------------
class spi_scoreboard extends uvm_scoreboard;
//------------------------------------------------------------
    `uvm_component_utils(spi_scoreboard)
    `uvm_analysis_imp_decl(_exp_pkt)
    `uvm_analysis_imp_decl(_act_pkt)
    uvm_analysis_imp_exp_pkt#(spi_sequence_item,spi_scoreboard)
        drv2sb;
    uvm_analysis_imp_act_pkt#(spi_sequence_item,spi_scoreboard)
        mon2sb;
    spi_sequence_item drv_pkt[$];
    spi_sequence_item mon_pkt[$];
    spi_sequence_item ip_pkt;
    spi_sequence_item op_pkt;
    static string report_tag;
    spi_coverage spi_covg;
    int pass = 0;
    int fail = 0;
```

```
//--------------------------------------------------------------------------
    function new(string name="spi_scoreboard",uvm_component parent);
        super.new(name,parent);
        report_tag = $sformatf("%0s",name);
        drv2sb = new("drv2sb",this);
        mon2sb = new("mon2sb",this);
    endfunction: new
//--------------------------------------------------------------------------
     function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        `uvm_info(get_full_name(),"Build phase called in spi_scoreboard",UVM_LOW)
        spi_covg = spi_coverage::type_id::create("spi_covg",this);
    endfunction: build_phase
//--------------------------------------------------------------------------
    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        `uvm_info(get_full_name(),"Connect phase called in spi_scoreboard",UVM_LOW)
    endfunction: connect_phase
//--------------------------------------------------------------------------
    function void write_exp_pkt(spi_sequence_item tmp_pkt);
        spi_sequence_item pkt;
```

```
            $cast(pkt,tmp_pkt.clone());
            //`uvm_info(report_tag,$sformatf("Received packet from driver %0s ",pkt.sprint()),UVM_LOW)
            drv_pkt.push_back(pkt);
            uvm_test_done.raise_objection(this);
        endfunction: write_exp_pkt
//-----------------------------------------------------------------------
        function void write_act_pkt(spi_sequence_item tmp_pkt);
            spi_sequence_item pkt;
            $cast(pkt,tmp_pkt.clone());
           //`uvm_info(report_tag,$sformatf("Received packet from DUT %0s ",pkt.sprint()),UVM_LOW)
            mon_pkt.push_back(pkt);
        endfunction: write_act_pkt
//-----------------------------------------------------------------------
        task run_phase(uvm_phase phase);
            //fork
            forever begin
            wait(mon_pkt.size()!=0);
            op_pkt = mon_pkt.pop_front();
            ip_pkt = drv_pkt.pop_front();
            //if(drv_pkt.size()==0)
              //`uvm_error("Expected packet was not received in scoreboard",UVM_LOW)
            perform_check(ip_pkt,op_pkt);
```

```
            perform_coverage(ip_pkt);
            uvm_test_done.drop_objection(this);
            end
            //join_none
            //disable fork;
        endtask: run_phase
//------------------------------------------------------------------------
        function void perform_coverage(spi_sequence_item pkt);
              spi_covg.perform_coverage(pkt);
        endfunction: perform_coverage
//------------------------------------------------------------------------
        function void perform_check(spi_sequence_item ip_pkt,
          spi_sequence_item op_pkt);
            if(ip_pkt.exp_master_data==op_pkt.out_master_data &&
              ip_pkt.exp_slave_data==op_pkt.out_slave_data)
            begin
            //`uvm_info(get_full_name(),"Master PASSED",UVM_MEDIUM)
            //`uvm_info(get_full_name(),"Slave PASSED",UVM_MEDIUM)
            pass++;
            end
            else
            begin
            `uvm_info(get_full_name(),$sformatf("Slave FAILED: exp
              data=%0h and out data=%0h",ip_pkt.exp_slave_data,
              op_pkt.out_slave_data),UVM_MEDIUM)
```

```
        `uvm_info(get_full_name(),$sformatf("Master FAILED: exp
            data=%0h and out master data=%0h",ip_pkt.
            exp_master_data,op_pkt.out_master_data),UVM_MEDIUM)
        fail++;
        end
    endfunction: perform_check
//------------------------------------------------------------------
    function void extract_phase(uvm_phase phase);
    endfunction: extract_phase
//------------------------------------------------------------------
    function void report_phase(uvm_phase phase);
    if(fail==0)
    begin
     $display
    ("----------------------------------------32bit--MSB First--TX:
       posedge--RX:negedge------------------------------------");
    $display
    ("-------------------------------------------------------TEST
       PASSED-------------------------------------------------------");
     $display
    ("
       ***********************************************************************
       ");
uvm_report_info("Scoreboard Report",$sformatf("Trasactions PASS
    = %0d FAIL = %0d",pass,fail),UVM_MEDIUM);
```

```
    $display
("
 **********************************************************************
 ");
$display
("
 ----------------------------------------------------------------------
 ");
  $display
("
 ----------------------------------------------------------------------
 ");
end
else
begin
    $display
("----------------------------------32bit --MSB First --TX:
 posedge --RX: negedge ----------------------------------");
$display
("--------------------------------------------------TEST
 FAILED--------------------------------------------------");
  $display
("
 **********************************************************************
 ");
```

```
uvm_report_info("Scoreboard Report",$sformatf("Trasactions PASS
    = %0d FAIL = %0d",pass,fail),UVM_MEDIUM);
      $display
    ("
      **********************************************************************
      ");
    $display
    ("
      ----------------------------------------------------------------------
      ");
     $display
    ("
      ----------------------------------------------------------------------
      ");
    end
    endfunction: report_phase
//----------------------------------------------------------------------
endclass: spi_scoreboard
//----------------------------------------------------------------------
```

I.18 Coverage

```
/*
 * Author:  Deepak Siddharth Parthipan
 *          RIT, NY, USA
 * Module:  coverage
 */
//-----------------------------------------------------------------
class spi_coverage extends uvm_component;
//-----------------------------------------------------------------
    `uvm_component_utils(spi_coverage)

    spi_sequence_item c_pkt;
//-----------------------------------------------------
    covergroup spi_trans_cg;

    cp_dut_mosi: coverpoint c_pkt.exp_master_data
    {
        bins byte7   = {[0:255]};
        bins byte15  = {[256:65535]};
        bins byte23  = {[65536:16777215]};
        bins byte31  = {[16777216:$]};
    }
    endgroup : spi_trans_cg
//-----------------------------------------------------
```

```
    function new(string name="spi_covg", uvm_component parent=
       null);
    super.new(name,parent);
    spi_trans_cg = new();
    endfunction : new
//-----------------------------------------------------------------
    function void perform_coverage(spi_sequence_item pkt);
        this.c_pkt=pkt;
        spi_trans_cg.sample();
    endfunction : perform_coverage
//-----------------------------------------------------------------
endclass: spi_coverage
//-----------------------------------------------------------------------------
```

I.19 SPI Slave Model

```verilog
/*
 * Author:   Deepak Siddharth Parthipan
 *           RIT, NY, USA
 * Module:   spi_slave_model
 */
//----------------------------------------------------------------------------------
`include "src/spi_defines.v"
`include "src/timescale.v"
//----------------------------------------------------------------------------------
module spi_slave (
  // Wishbone signals
  wb_clk_i, wb_rst_i, wb_adr_i, wb_dat_i, wb_dat_o, wb_sel_i,
  wb_we_i, wb_stb_i, wb_cyc_i, wb_ack_o, wb_err_o, wb_int_o,

  // SPI signals
  ss_pad_i, sclk_pad_i, mosi_pad_i, miso_pad_o,

  //Scan Insertion
  scan_in0, scan_en, test_mode, scan_out0); //,reset, clk);
```

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

  // Wishbone signals
  input              wb_clk_i;     // master clock input
  input              wb_rst_i;     // synchronous active high reset
  input     [4:0]    wb_adr_i;     // lower address bits
  input     [32-1:0] wb_dat_i;     // databus input
  output    [32-1:0] wb_dat_o;     // databus output
  input     [3:0]    wb_sel_i;     // byte select inputs
  input              wb_we_i;      // write enable input
  input              wb_stb_i;     // stobe/core select signal
  input              wb_cyc_i;     // valid bus cycle input
  output             wb_ack_o;     // bus cycle acknowledge output
```

```
  output                            wb_err_o;          //
     termination w/ error
  output                            wb_int_o;          //
     interrupt request signal output

// SPI signals
  input          [`SPI_SS_NB-1:0]   ss_pad_i;          // slave
     select
  input                             sclk_pad_i;        // serial
     clock
  input                             mosi_pad_i;        // master
     out slave in
  output                            miso_pad_o;        // master
     in slave out

  input                             scan_in0;          // test
     scan mode data input
  input                             scan_en;           // test
     scan mode enable
  input                             test_mode;         // test
     mode select
  output                            scan_out0;         // test
     scan mode data output

```

```
wire                              rx_negedge;      // slave receiving on negedge
wire                              tx_negedge;      // slave transmiting on negedge
wire                              spi_tx_sel;      // tx_l register select

reg                  [32-1:0]     wb_dat_o;
reg                  [32-1:0]     wb_dat;
reg                               wb_ack_o;
reg                               wb_int_o;
reg   [`SPI_CTRL_BIT_NB-1:0]      ctrl;
reg                               miso_pad_o;

//----------------------------------------------------------------------------
// Address decoder
assign spi_ctrl_sel    = wb_cyc_i & wb_stb_i & (wb_adr_i[`SPI_OFS_BITS] == `SPI_CTRL);

assign rx_negedge = ctrl[`SPI_CTRL_RX_NEGEDGE];
assign tx_negedge = ctrl[`SPI_CTRL_TX_NEGEDGE];
assign char_len   = ctrl[`SPI_CTRL_CHAR_LEN];
assign ie         = ctrl[`SPI_CTRL_IE];
```

```

  assign spi_tx_sel   = wb_cyc_i & wb_stb_i & (wb_adr_i[
     `SPI_OFS_BITS] == `SPI_TX_0);
//
     ---------------------------------------------------------------------------------------------------

  // Wb data out
  always @(posedge wb_clk_i or posedge wb_rst_i)
  begin
    if (wb_rst_i)
      wb_dat_o <=  32'b0;
    else
      wb_dat_o <=  wb_dat;
  end
//
     ---------------------------------------------------------------------------------------------------

  // Wb acknowledge
  always @(posedge wb_clk_i or posedge wb_rst_i)
  begin
    if (wb_rst_i)
      wb_ack_o <=  1'b0;
    else
      wb_ack_o <=  wb_cyc_i & wb_stb_i & ~wb_ack_o;
  end
```

```
//------------------------------------------------------------------------
  // Wb error
  assign wb_err_o = 1'b0;

  // Interrupt
/*  always @(posedge wb_clk_i or posedge wb_rst_i)
  begin
    if (wb_rst_i)
      wb_int_o <= 1'b0;
    else if (ie && !ss_pad_i && last_bit && pos_edge) // there needs to be rising edge detector
      wb_int_o <= 1'b1;
    else if (wb_ack_o)
      wb_int_o <= 1'b0;
  end*/
//------------------------------------------------------------------------
  // Ctrl register
  always @(posedge wb_clk_i or posedge wb_rst_i)
  begin
    if (wb_rst_i)
      ctrl <= {`SPI_CTRL_BIT_NB{1'b0}};
```

```
    else if(spi_ctrl_sel && wb_we_i && (!(&ss_pad_i))) //!ss_pad_i Because during no transfer we go to tristate mode
      begin
        if (wb_sel_i[0])
          ctrl[7:0] <= wb_dat_i[7:0] | {7'b0, ctrl[0]};
        if (wb_sel_i[1])
          ctrl[`SPI_CTRL_BIT_NB-1:8] <= wb_dat_i[`SPI_CTRL_BIT_NB-1:8];
      end
  end
//
  always @(posedge(sclk_pad_i && !rx_negedge) or negedge(sclk_pad_i && rx_negedge) or posedge wb_rst_i or posedge(wb_clk_i && (&ss_pad_i)))
  begin
    if (wb_rst_i)
      wb_dat <= 32'b0;
    else if (!(&ss_pad_i))
      wb_dat <= {wb_dat[30:0], mosi_pad_i};
    else if ((&ss_pad_i) && spi_tx_sel)
      wb_dat <= wb_dat_i;
     else
```

```
        wb_dat <=  wb_dat;
  end
//------------------------------------------------------------------------
  always @(posedge(sclk_pad_i && !tx_negedge) or negedge(sclk_pad_i && tx_negedge))
  begin
      miso_pad_o <=  wb_dat[31];
  end
//------------------------------------------------------------------------
endmodule
 //------------------------------------------------------------------------
```

I.20 Test defines

```
//-----------------------------------------------------------------------
/*
 *
 * Author:  Deepak Siddharth Parthipan
 *          RIT, NY, USA
 * Module:  spi tb defines
 *
 */
//-----------------------------------------------------------------------
    `define LOW 0
    `define HIGH 1

    parameter CLOCK_PERIOD = 50;
    parameter RESET_PERIOD = 25;

    parameter dwidth = 32;
    parameter awidth = 32;

    parameter SPI_RX_0   = 5'h0;
    parameter SPI_RX_1   = 5'h4;
    parameter SPI_RX_2   = 5'h8;
    parameter SPI_RX_3   = 5'hc;
    parameter SPI_TX_0   = 5'h0;
```

```
    parameter SPI_TX_1    = 5'h4;
    parameter SPI_TX_2    = 5'h8;
    parameter SPI_TX_3    = 5'hc;
    parameter SPI_CTRL    = 5'h10;
    parameter SPI_DIVIDE  = 5'h14;
    parameter SPI_SS      = 5'h18;

    logic scan_in0, scan_in1, scan_en, test_mode;
    logic clock, rstn;
    logic  [7:0] ss;
    logic  [31:0] q;
    logic sclk, mosi, miso;
    logic tip;

    typedef virtual spi_if spi_vif;
//-----------------------------------------------------------------------------------
```