UVM Verification of an SPI Master Core

Deepak Siddharth Parthipan
dp9040@rit.edu

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation
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
<table>
<thead>
<tr>
<th>Contents</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.6.9 Post Main Phase</td>
<td>19</td>
</tr>
<tr>
<td>3.5.6.10 Pre Shutdown Phase</td>
<td>19</td>
</tr>
<tr>
<td>3.5.6.11 Shutdown Phase</td>
<td>19</td>
</tr>
<tr>
<td>3.5.6.12 Post Shutdown Phase</td>
<td>19</td>
</tr>
<tr>
<td>3.5.7 Extract Phase</td>
<td>19</td>
</tr>
<tr>
<td>3.5.8 Check Phase</td>
<td>19</td>
</tr>
<tr>
<td>3.5.9 Report Phase</td>
<td>20</td>
</tr>
<tr>
<td>3.5.10 Final Phase</td>
<td>20</td>
</tr>
<tr>
<td>3.6 UVM Macros</td>
<td>20</td>
</tr>
<tr>
<td><strong>4 System Architecture</strong></td>
<td>21</td>
</tr>
<tr>
<td>4.1 WISHBONE Interface</td>
<td>21</td>
</tr>
<tr>
<td>4.2 WISHBONE I/O Registers</td>
<td>22</td>
</tr>
<tr>
<td>4.3 Serial Peripheral Interface</td>
<td>24</td>
</tr>
<tr>
<td>4.4 Data Transmission</td>
<td>25</td>
</tr>
<tr>
<td>4.5 Hardware Architecture</td>
<td>27</td>
</tr>
<tr>
<td>4.5.1 Design of Clock Generation module (spi_clk_gen)</td>
<td>27</td>
</tr>
<tr>
<td>4.5.2 Serial data transfer module design (spi_shift)</td>
<td>29</td>
</tr>
<tr>
<td>4.5.3 Top-level module (spi)</td>
<td>29</td>
</tr>
<tr>
<td>4.6 SPI Registers</td>
<td>29</td>
</tr>
<tr>
<td>4.6.1 RxX Register</td>
<td>29</td>
</tr>
<tr>
<td>4.6.2 TxX Register</td>
<td>30</td>
</tr>
<tr>
<td>4.6.3 ASS Register</td>
<td>30</td>
</tr>
<tr>
<td>4.6.4 DIVIDER Register</td>
<td>31</td>
</tr>
<tr>
<td>4.6.5 SS Register</td>
<td>31</td>
</tr>
</tbody>
</table>
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
## Contents

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
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.13</td>
<td>Sequencer</td>
<td>113</td>
</tr>
<tr>
<td>I.14</td>
<td>Driver</td>
<td>114</td>
</tr>
<tr>
<td>I.15</td>
<td>Monitor</td>
<td>117</td>
</tr>
<tr>
<td>I.16</td>
<td>Wishbone Bus Function Model</td>
<td>120</td>
</tr>
<tr>
<td>I.17</td>
<td>Scoreboard</td>
<td>124</td>
</tr>
<tr>
<td>I.18</td>
<td>Coverage</td>
<td>131</td>
</tr>
<tr>
<td>I.19</td>
<td>SPI Slave Model</td>
<td>133</td>
</tr>
<tr>
<td>I.20</td>
<td>Test defines</td>
<td>141</td>
</tr>
</tbody>
</table>
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) as 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 and is derived from 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
1.1 Research Goals

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 cornerstone 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 used 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 and 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 ef-
fort 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.
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 UVM Class Hierarchy

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 UVM Class Hierarchy

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].
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 UVM Phases

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 UVM Phases

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_utills - This macro registers is used when new ‘uvm_component classes are derived.

- ‘uvm_object_utills – Similar to ‘uvm_component_utills 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
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.

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: WISHBONE I/O Ports

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

- **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-
4.4 Data Transmission

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
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
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].

\[ f_{sclk} = \frac{f_{wbclk}}{(\text{DIVIDER} + 1) \times 2} \]
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

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

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 equation 1.

### 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.
5.1 Testbench Components

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 Testbench Components

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.
5.1 Testbench Components

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.

Figure 5.2: UVM Sequencer Driver Communication
5.1 Testbench Components

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>
<thead>
<tr>
<th>Table 5.1: Synthesis Report</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>Area</td>
</tr>
<tr>
<td>Sequential Area (μm²)</td>
</tr>
<tr>
<td>Combinational Area (μm²)</td>
</tr>
<tr>
<td>Buf/Inv Area (μm²)</td>
</tr>
<tr>
<td>Total Area (μm²)</td>
</tr>
<tr>
<td>Power</td>
</tr>
<tr>
<td>Internal Power (μW)</td>
</tr>
<tr>
<td>Switching Power (μW)</td>
</tr>
<tr>
<td>Leakage Power (μW)</td>
</tr>
<tr>
<td>Total Power (μW)</td>
</tr>
<tr>
<td>Timing</td>
</tr>
<tr>
<td>Slack (ns)</td>
</tr>
<tr>
<td>DFT Coverage</td>
</tr>
<tr>
<td>100%</td>
</tr>
<tr>
<td>Latency (Clock cycles)</td>
</tr>
</tbody>
</table>
5.2 Testbench Results

5.2.2 Data Transactions

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

<table>
<thead>
<tr>
<th>Data Transfer</th>
<th>Sent First</th>
<th>Transmit</th>
<th>Receive</th>
</tr>
</thead>
<tbody>
<tr>
<td>32bit</td>
<td>MSB</td>
<td>posedge</td>
<td>negedge</td>
</tr>
</tbody>
</table>

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
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
5.2 Testbench Results

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.

<table>
<thead>
<tr>
<th>Ex</th>
<th>Name</th>
<th>Overall Average Grade</th>
<th>Overall Covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Overall</td>
<td>66.74%</td>
<td>425 / 867 (49.01%)</td>
</tr>
<tr>
<td>1</td>
<td>Code</td>
<td>66.74%</td>
<td>425 / 987 (47.91%)</td>
</tr>
<tr>
<td>2</td>
<td>Block</td>
<td>75.67%</td>
<td>83 / 132 (62.88%)</td>
</tr>
<tr>
<td></td>
<td>Expression</td>
<td>69.7%</td>
<td>46 / 51 (90.2%)</td>
</tr>
<tr>
<td></td>
<td>Toggle</td>
<td>35.45%</td>
<td>296 / 704 (42.05%)</td>
</tr>
</tbody>
</table>

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.

![Figure 5.6: Clock Level Code Coverage](image)

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

![Figure 5.7: Shift Level Code Coverage](image)

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:

- **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.

**Figure 5.8: Signal Coverage**

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
5.2 Testbench Results

outputs. At SPI signal level below six coverpoints are incorporated:

- **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.

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.
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


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


Appendix I

Source Code

I.1 SPI Top

1 /*
2  * Author: Deepak Siddharth Parthipan
3  * RIT, NY, USA
4  * Module: spi
5  */
6 //---

7 `include "src/spi_defines.v"
8 `include "src/timescale.v"
9 //---

10 module spi
11 (
I.1 SPI Top

/* 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
29  input            wb_we_i;       // write
   enable input
30  input            wb_stb_i;    // stobe/
core select signal
31  input            wb_cyc_i;    // valid
   bus cycle input
32  output           wb_ack_o;    // bus
cycle acknowledge output
33  output           wb_err_o;    //
termination w/ error
34  output           wb_int_o;    //
   interrupt request signal output
35 /*----------------------------------------SPI signals ----------------------------------------*/
36  output           [ 'SPI_SS_NB–1:0] ss_pad_o; // slave
        select
37  output           sclk_pad_o; // serial
clock
38  output           mosi_pad_o; // master
   out slave in
39  input            miso_pad_i; // master
   in slave out
40  // input            reset;       // system
   reset
```vhdl
I.1 SPI Top

41 // input
   clk;   // system
   clock
42 input scan_in0;  // test
   scan mode data input
43 input scan_en;   // test
   scan mode enable
44 input test_mode; // test
   mode select
45 output scan_out0; // test
   scan mode data output
46 output tip;

47 */

48 reg [32−1:0] wb_dat_o;
49 reg [32−1:0] wb_dat;
50 reg wb_ack_o;
51 reg wb_int_o;
52 reg [‘SPI_CTRL_BIT_NB−1:0] ctrl;
53 reg [‘SPI_DIVIDER_LEN−1:0] divider;
54 reg [‘SPI_SS_NB−1:0] ss;
55 reg scan_out0;
56 // Internal signals
57 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 clk
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(
                            clk_pad_o), .pos_edge(pos_edge),
                        .neg_edge(neg_edge));

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_128[1'b0]}, rx[SPI_MAX_CHAR_128-1:96]};
  else
    ifdef SPI_MAX_CHAR_64
      'SPI_RX_0:    wb_dat = rx[31:0];
      'SPI_RX_1:    wb_dat = {{64-'SPI_MAX_CHAR_64[1'b0]}, rx[SPI_MAX_CHAR_64-1:32]};
      'SPI_RX_2:    wb_dat = 32'b0;
      'SPI_RX_3:    wb_dat = 32'b0;
    else
      'SPI_RX_0:    wb_dat = {{32-'SPI_MAX_CHAR_32[1'b0]}, rx[SPI_MAX_CHAR_32-1:0]};
      '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};
I.1 SPI Top

116  `SPI_DIVIDE : wb_dat = {{32-`SPI_DIVIDER_LEN{1'b0}},
           divider};
117  `SPI_SS :    wb_dat = {{32-`SPI_SS_NB{1'b0}}, ss};
118  default:
119    wb_dat = 32'b0;
120  endcase
121  end
122  `*----------------------------------------------------------Wb data out
123  `*----------------------------------------------------------*/
124  always @(posedge wb_clk_i or posedge wb_rst_i)
125  begin
126    if (wb_rst_i)
127      wb_dat_o <= 32'b0;
128    else
129      wb_dat_o <= wb_dat;
130  end
131  `*----------------------------------------------------------Wb acknowledge
132  `*----------------------------------------------------------*/
133  always @(posedge wb_clk_i or posedge wb_rst_i)
134  begin
135    if (wb_rst_i)
136      wb_ack_o <= 1'b0;
137    else
138      wb_ack_o <= wb_cyc_i & wb_stb_i & ~wb_ack_o;
139  end
```verilog
138 //----------------------------------Wb error
139 assign wb_err_o = 1'b0;
140 //-----------------------------------Interrupt
141 always @(posedge wb_clk_i or posedge wb_rst_i)
142 begin
143 if (wb_rst_i)
144 wb_int_o <= 1'b0;
145 else if (ie && tip && last_bit && pos_edge)
146 wb_int_o <= 1'b1;
147 else if (wb_ack_o)
148 wb_int_o <= 1'b0;
149 end
150 //------------------------------Divider register
151 always @(posedge wb_clk_i or posedge wb_rst_i)
152 begin
153 if (wb_rst_i)
154 divider <= `SPI_DIVIDER_LEN{1'b0};
155 else if (spi_divider_sel && wb_we_i && !tip)
156 begin
157 ifdef SPI_DIVIDER_LEN_8
158 if (wb_sel_i[0])
159 divider <= wb_dat_i[`SPI_DIVIDER_LEN−1:0];
160 ```
160  `endif
161  `ifdef SPI_DIVIDER_LEN_16
162   if (wb_sel_i[0])
163     divider[7:0] <= wb_dat_i[7:0];
164   if (wb_sel_i[1])
165     divider[`SPI_DIVIDER_LEN-1:8] <= wb_dat_i[`SPI_DIVIDER_LEN-1:8];
166  `endif
167  `ifdef SPI_DIVIDER_LEN_24
168   if (wb_sel_i[0])
169     divider[7:0] <= wb_dat_i[7:0];
170   if (wb_sel_i[1])
171     divider[15:8] <= wb_dat_i[15:8];
172   if (wb_sel_i[2])
173     divider[`SPI_DIVIDER_LEN-1:16] <= wb_dat_i[`SPI_DIVIDER_LEN-1:16];
174  `endif
175  `ifdef SPI_DIVIDER_LEN_32
176   if (wb_sel_i[0])
177     divider[7:0] <= wb_dat_i[7:0];
178   if (wb_sel_i[1])
179     divider[15:8] <= wb_dat_i[15:8];
180   if (wb_sel_i[2])
181     divider[23:16] <= wb_dat_i[23:16];
182   if (wb_sel_i[3])
183   divider[24:16] <= wb_dat_i[24:16];
184   if (wb_sel_i[4])
185     divider[31:16] <= wb_dat_i[31:16];
186  `endif

divider[‘SPI_DIVIDER_LEN−1:24] <= wb_dat_i[‘SPI_DIVIDER_LEN−1:24];
end

always @(posedge wb_clk_i or posedge wb_rst_i)
begins
if (wb_rst_i)
ctrl <= {'SPI_CTRL_BIT_NB{1'b0}};
elself (spi_ctrl_sel & & wb_we_i & & !tip)
begins
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
assign rx_negedge = ctrl[‘SPI_CTRL_RX_NEGEDGE];
assign tx_nedge = 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'];

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
  end
end
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
I.2 SPI Clock

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

22 reg clk_out;
23 reg pos_edge;
24 reg neg_edge;
25 reg [ 'SPI_DIVIDER_LEN-1:0] cnt; // clock counter
26 wire cnt_zero; // counter is equal to zero
27 wire cnt_one; // counter is equal to one

29 assign cnt_zero = cnt == { 'SPI_DIVIDER_LEN{1'b0} };
30 assign cnt_one = cnt == {{ 'SPI_DIVIDER_LEN-1{1'b0} }, 1'b1};
31 /*--------------------------------Counter counts half period--------------------------------*/
32 always @(posedge clk_in or posedge rst)
33 begin

34     if (rst)
35         cnt <= {'SPI_DIVIDER_LEN[1'b1]};
36     else
37         begin
38             if (!enable || cnt_zero)
39                 cnt <= divider;
40             else
41                 cnt <= cnt - {'SPI_DIVIDER_LEN-1[1'b0]}, 1'b1;
42         end
43     end
44     /*---------------------------------clk_out is asserted every other half period---------------------------------*/
45     always @(posedge clk_in or posedge rst)
46     begin
47         if (rst)
48             clk_out <= 1'b0;
49         else
50             clk_out <= (enable && cnt_zero && (!last_clk || clk_out))
51                 ? ~clk_out : clk_out;
52     end
53     /*--------------------------------- Pos and neg edge signals ---------------------------------*/
54     always @(posedge clk_in or posedge rst)
55     begin
56         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
I.3 SPI Shift

15  input [3:0] latch;   // latch signal for storing the data in shift register
16  input [3:0] byte_sel;  // byte select signals for storing the data in shift register
17  input ['SPI_CHAR_LEN_BITS−1:0] len;   // data len in bits (minus one)
18  input lsb;     // lsb first on the line
19  input go;      // start transfer
20  input pos_edge;    // recognize posedge of sclk
21  input neg_edge;    // recognize negedge of sclk
22  input rx_negedge;  // s_in is sampled on negative edge
23  input tx_negedge;  // s_out is driven on negative edge
24  output tip;      // transfer in progress
25  output last;     // last bit
26  input [31:0] p_in;   // parallel in
27  output ['SPI_MAX_CHAR−1:0] p_out; // parallel out
28  input s_clk;      // serial clock
29  input s_in;       // serial in
I.3 SPI Shift

30 output s_out;       // serial out
31 reg s_out;
32 reg tip;
33 reg [‘SPI_CHAR_LEN_BITS:0] cnt;      // data bit
34 reg [‘SPI_MAX_CHAR–1:0] data;    // shift register
35 wire [‘SPI_CHAR_LEN_BITS:0] tx_bit_pos;  // next bit position
36 wire [‘SPI_CHAR_LEN_BITS:0] rx_bit_pos;  // next bit position
37 wire rx_clk;            // rx clock enable
38 wire tx_clk;            // tx clock enable
39 //

40 assign p_out = data;
41 assign tx_bit_pos = lsb ? {!llen, len} – cnt : cnt – {{
   ‘SPI_CHAR_LEN_BITS{1’b0},1’b1};
42 assign rx_bit_pos = lsb ? {!llen, len} – (rx_negege ? cnt
   + {{‘SPI_CHAR_LEN_BITS{1’b0},1’b1} : cnt) :
   (rx_negege ? 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

‘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])
I.3 SPI Shift

111     data[95:88] <= p_in[31:24];
112     if (byte_sel[2])
113     data[87:80] <= p_in[23:16];
114     if (byte_sel[1])
115     data[79:72] <= p_in[15:8];
116     if (byte_sel[0])
117     data[71:64] <= p_in[7:0];
118           end
119     else if (latch[3] && !tip)
120           begin
121           if (byte_sel[3])
122           data[127:120] <= p_in[31:24];
123           if (byte_sel[2])
124           data[119:112] <= p_in[23:16];
125           if (byte_sel[1])
126           data[111:104] <= p_in[15:8];
127           if (byte_sel[0])
128           data[103:96] <= p_in[7:0];
129           end
130       else
131
132       ifdef SPI_MAX_CHAR_64
133       else if (latch[0] && !tip)
134           begin
135           if (byte_sel[3])
136           data[127:120] <= p_in[31:24];
137           if (byte_sel[2])
138           data[119:112] <= p_in[23:16];
139           if (byte_sel[1])
140           data[111:104] <= p_in[15:8];
141           if (byte_sel[0])
142           data[103:96] <= p_in[7:0];
143       end
144     endif
145     else
146     begin
147     end
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
#if SPI_MAX_CHAR_8
if (byte_sel[0])
data[SPI_MAX_CHAR-1:0] <= p_in[SPI_MAX_CHAR-1:0];
#endif
161  `endif
162  `ifdef  SPI_MAX_CHAR_16
163   if (byte_sel[0])
164    data[7:0] <= p_in[7:0];
165   if (byte_sel[1])
166    data[SPI_MAX_CHAR−1:8] <= p_in[SPI_MAX_CHAR−1:8];
167  `endif
168  `ifdef  SPI_MAX_CHAR_24
169   if (byte_sel[0])
170    data[7:0] <= p_in[7:0];
171   if (byte_sel[1])
172    data[15:8] <= p_in[15:8];
173   if (byte_sel[2])
174    data[SPI_MAX_CHAR−1:16] <= p_in[SPI_MAX_CHAR−1:16];
175  `endif
176  `ifdef  SPI_MAX_CHAR_32
177   if (byte_sel[0])
178    data[7:0] <= p_in[7:0];
179   if (byte_sel[1])
180    data[15:8] <= p_in[15:8];
181   if (byte_sel[2])
182    data[23:16] <= p_in[23:16];
183   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
I.4 Defines

```
21  `ifdef SPI_DIVIDER_LEN_16
22   `define SPI_DIVIDER_LEN = 16 // Can be set from 9 to 16
23  `endif
24  `ifdef SPI_DIVIDER_LEN_24
25   `define SPI_DIVIDER_LEN = 24 // Can be set from 17 to 24
26  `endif
27  `ifdef SPI_DIVIDER_LEN_32
28   `define SPI_DIVIDER_LEN = 32 // Can be set from 25 to 32
29  `endif
30  `/*
31   Maximum number of bits that can be send/received at once.
32   Use SPI_MAX_CHAR for fine tuning the exact number, when using
33   SPI_MAX_CHAR_32, SPI_MAX_CHAR_24, SPI_MAX_CHAR_16,
34   SPI_MAX_CHAR_8.
35 */
36
37  `define SPI_MAX_CHAR_128
38  // `define SPI_MAX_CHAR_64
39  // `define SPI_MAX_CHAR_32
40  // `define SPI_MAX_CHAR_24
```
/* Define SPI_MAX_CHAR_16 */
#define SPI_MAX_CHAR 16  // Can only be set from 9 to 16
#define SPI_CHAR_LEN_BITS 4

/* Define SPI_MAX_CHAR_8 */

#ifdef SPI_MAX_CHAR_128
#define SPI_MAX_CHAR 128  // Can only be set to 128
#define SPI_MAX_CHAR_LEN_BITS 7
#endif

#ifdef SPI_MAX_CHAR_64
#define SPI_MAX_CHAR 64  // Can only be set to 64
#define SPI_MAX_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_MAX_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
I.4 Defines

64  `ifdef SPI_MAX_CHAR_8
65  `define SPI_MAX_CHAR 8  // Can be set from 1 to 8
66  `define SPI_CHAR_LEN_BITS 3
67  `endif
68  //

69  /*
70  Number of device select signals. Use SPI_SS_NB for fine tuning
    the
71  exact number.
72  */
73  `define SPI_SS_NB_8
74  // `define SPI_SS_NB_16
75  // `define SPI_SS_NB_24
76  // `define SPI_SS_NB_32
77
78  `ifdef SPI_SS_NB_8
79  `define SPI_SS_NB 8  // Can be set from 1 to 8
80  `endif
81  `ifdef SPI_SS_NB_16
82  `define SPI_SS_NB 16  // Can be set from 9 to 16
83  `endif
84  `ifdef SPI_SS_NB_24
I.4 Defines

85 define SPI_SS_NB 24   // Can be set from 17 to 24
86 endif
87 ifdef SPI_SS_NB_32
88 define SPI_SS_NB 32   // Can be set from 25 to 32
89 endif
90

/*
91 Bits of WISHBONE address used for partial decoding of SPI registers.
92 */
93
94 define SPI_OFS_BITS 4:2
95

/* Register offset */
96 define SPI_RX_0 0
97 define SPI_RX_1 1
98 define SPI_RX_2 2
99 define SPI_RX_3 3
100 define SPI_TX_0 0
101 define SPI_TX_1 1
I.4 Defines

```
103  `define SPI_TX_2     2
104  `define SPI_TX_3     3
105  `define SPI_CTRL     4
106  `define SPI_DIVIDE   5
107  `define SPI_SS       6
108  //

109  /*! Number of bits in ctrl register */
110  `define SPI_CTRL_BIT_NB 14
111  //

112  /*! Control register bit position */
113  `define SPI_CTRL_ASS   13
114  `define SPI_CTRL_IE    12
115  `define SPI_CTRL_LSB   11
116  `define SPI_CTRL_TX_NEGEDGE 10
117  `define SPI_CTRL_RX_NEGEDGE 9
118  `define SPI_CTRL_GO     8
119  `define SPI_CTRL_RES_1  7
120  `define SPI_CTRL_CHAR_LEN 6:0
121  //
```
I.5 Test Top

1 /*
2 * Author: Deepak Siddharth Parthipan
3 * RIT, NY, USA
4 * Module: tb_top
5 */
6 //---------------------------------------------
7 `include "uvm_macros.svh"
8 `include "spi_pkg.sv"
9 `include "spi_if.sv"
10 //---------------------------------------------
11 module test;
12 import uvm_pkg::*;
13 import spi_pkg::*;
14
15 spi_if master(clock); // Interface declaration
16 spi_if slave(clock); // Interface declaration
17 /*---------------------------------------SPI master core---------------------------------------*/
18 spi_top ( 
19     /*tb to DUT connection*/
20     .wb_clk_i(clock),
21     .wb_rst_i(rstn),
22     .wb_adr_i(master.adr[4:0]),
I.5 Test Top

```verilog
    .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");
I.5 Test Top

```vhdl
    'ifdef SDFSCAN
        $sdf_annotate("sdf/spi_tsmcl8_scan.sdf", test.top);
    'endif

    generate_clock();
    reg_intf_to_config_db();
    initialize_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 retrieved 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 initialize_dut();
test_mode = 1'b0;
scan_in0 = 1'b0;
scan_in1 = 1'b0;
scan_en = 1'b0;
endfunction : initialize_dut

//-------------------------------------------------------------
task reset_dut();
rsn <= 'LOW;
repeat(RESET_PERIOD) @(posedge clock);
rsn <= 'HIGH;
repeat(RESET_PERIOD) @(posedge clock);
rsn = 'LOW;
//-->RST_DONE;
endtask : reset_dut

//-------------------------------------------------------------
endmodule : test
I.6 Interface

1 /*
2 * Author: Deepak Siddharth Parthipan
3 * RIT, NY, USA
4 * Module: Package
5 */
6 //---------------------------------------------
7 interface spi_if(input bit clk);
8 //---------------------------------------------
9 // Wishbone signals
10
11 logic [4:0] adr; // lower address bits
12 logic [32-1:0] din; // databus input
13 logic [32-1:0] dout; // databus output
14 logic [3:0] sel; // byte select inputs
15 logic we; // write enable input
16 logic stb; // stobe/core select signal
17 logic cyc; // valid bus cycle input
I.6 Interface

18  logic ack;       // bus cycle
    acknowledge output
19  logic err;       // termination w/ error
20  logic intp;      // interrupt request signal output input
21  logic pit;
22  //-----------------------------------------------
23  clocking drive_cb @(posedge clk);
24  input  din, ack, err, intp, pit;
25  output adr, dout, sel, we, stb, cyc;
26  endclocking : drive_cb
27  //-----------------------------------------------
28  clocking monitor_cb @(posedge clk);
29  input  din, ack, err, intp, pit;
30  output adr, dout, sel, we, stb, cyc;
31  endclocking : monitor_cb
32  //-----------------------------------------------
33  endinterface : spi_if
34  //-----------------------------------------------
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"
24 //___________________________________________________________
25 endpackage: spi_pkg
26 //___________________________________________________________
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)
endfunction: build_phase

/* 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
47 //______________________________________________________________
48 endclass: spi_test
49 //______________________________________________________________
I.9 Environment

1 /*
2 * Author: Deepak Siddharth Parthipan
3 * RIT, NY, USA
4 * Module: Environment
5 */
6
7 class spi_env extends uvm_env;
8
9 'uvm_component_utils(spi_env)
10   spi_agent agent;
11   spi_scoreboard scoreboard;
12
13 function new(string name="spi_env", uvm_component parent);
14   super.new(name, parent);
15 endfunction: new
16
17 function void build_phase(uvm_phase phase);
18   super.build_phase(phase);
19   'uvm_info(get_full_name(), "Build phase called in
20     spi_environment", UVM_LOW)
21       /* Build agent and scoreboard components */
22   agent = spi_agent::type_id::create("agent", this);
I.9 Environment

*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 scoreboard */

agent.monitor.dut_out_pkt.connect(spi_scoreboard.mon2sb);

agent.driver.dut_in_pkt.connect(spi_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)
```cpp
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

1 / *
2  * Author: Deepak Siddharth Parthipan
3  * RIT, NY, USA
4  * Module: Sequence Item
5  */
6 //---
7 class spi_sequence_item extends uvm_sequence_item;
8 //---
9     /*Register configuration*/
10     rand logic [31:0] master_ctrl_reg;
11     rand logic [31:0] slave_ctrl_reg;
12     rand logic [31:0] divider_reg;
13     rand logic [31:0] slave_select_reg;
14     rand logic [31:0] start_dut_reg;
15     /*DUT output*/
16     logic [31:0] out_master_data;
17     logic [31:0] out_slave_data;
18     /*Expected data*/
19     rand logic [31:0] exp_master_data;
20     rand logic [31:0] exp_slave_data;
21     /*DUT input*/
22     rand logic [31:0] in_master_data;
23     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

//

class: spi_sequence_item

//
I.12 Sequence

1 /*
2 * Author: Deepak Siddharth Parthipan
3 * RIT, NY, USA
4 * Module: Sequence
5 */
6 //
7 class spi_sequence extends uvm_sequence #(spi_sequence_item);
8 //
9 'uvm_object_utils(spi_sequence)
10 //
11 function new(string name="spi_sequence");
12 super.new(name);
13 endfunction: new
14 //
15 virtual task body();
16 req=spi_sequence_item :: type_id :: create("req");
17 start_item(req);
18 // configure_dut_register();
19 set_dut_data();
20 finish_item(req);
21 endtask: body
22 //
23 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;
});

endefunction: 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;
I.12 Sequence

39 // req.in_master_data == 32'
    h87654321;
40 // req.in_slave_data == 32'
    h1123344;
41 req.exp_master_data == req.
    in_slave_data;
42 req.exp_slave_data == req.
    in_master_data;
43 });
44 endfunction: set_dut_data
45 //---------------------------------------------
46 endclass: spi_sequence
47 //---------------------------------------------
I.13 Sequencer

```cpp
/*
 * 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
end
I.14 Driver

42                 join_none
43            endtask: run_phase
44          //
45      task drive_transfer(spi_sequence_item seq);
46            wb_bfm::wb_write(m_vif, 0, SPI_DIVIDE, seq.divider_reg);
47            // set divider register
48            wb_bfm::wb_write(m_vif, 0, SPI_SS, seq.slave_select_reg);
49            // set ss 0
50            wb_bfm::wb_write(m_vif, 0, SPI_TX_0, seq.in_master_data);
51            // set master data register
52            wb_bfm::wb_write(s_vif, 0, SPI_CTRL, seq.master_ctrl_reg);
53            // set master ctrl register
54            wb_bfm::wb_write(s_vif, 0, SPI_CTRL, seq.slave_ctrl_reg);
55            // set slave ctrl register
56            wb_bfm::wb_write(s_vif, 0, SPI_TX_0, seq.in_slave_data);
57            // set slave data register
58            wb_bfm::wb_write(m_vif, 0, SPI_CTRL, seq.start_dut_reg);
59            // start data transfer
60      endtask: drive_transfer
61          //
62      endclass: spi_driver
63          //
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"})
else 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 Function 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'b0}};
    vif.dout <= {dwidth{1'b0}};
    vif.cyc <= 1'b0;
    vif.stb <= 1'b0;
    vif.we  <= 1'hx;
    vif.sel <= {dwidth/8{1'b0}};
endtask: wb_reset
I.16 Wishbone Bus Function Model

24 /*-----------------------------Wishbone read cycle-----------------------------*/
25 static task wb_read;
26 input spi_vif vif;
27 input integer delay;
28 input logic [awidth -1:0] a;
29 output logic [dwidth -1:0] d;
30
31 begin
32    // wait initial delay
33    repeat (delay) @(vif.monitor_cb);
34    // assert wishbone signals
35    repeat (1) @(vif.monitor_cb);
36    vif.monitor_cb.adr <= a;
37    vif.monitor_cb.dout <= {dwidth{1'bx}};
38    vif.monitor_cb.cyc <= 1'b1;
39    vif.monitor_cb.stb <= 1'b1;
40    vif.monitor_cb.we <= 1'b0;
41    vif.monitor_cb.sel <= {dwidth/8{1'b1}};
42    @(vif.monitor_cb);
43    // wait for acknowledge from slave
44    wait(vif.monitor_cb.ack==1'b1)
45    // negate wishbone signals
46    repeat (1) @(vif.monitor_cb);
47    vif.monitor_cb.cyc <= 1'b0;
48    vif.monitor_cb.stb <= 1'bx;
I. 16 Wishbone Bus Funtion Model

49 `vif.monitor_cb.adr <= {awidth(1'bx)};
50 vif.monitor_cb.dout <= {dwidth(1'bx)};
51 vif.monitor_cb.we <= 1'hx;
52 vif.monitor_cb.sel <= {dwidth/8(1'bx)};
53 d = vif.monitor_cb.din;
54
55 end
56 endtask : wb_read
57
58 static task wb_write ;
59 input spi_vif vif;
60 input integer delay;
61 input logic [awidth -1:0] a;
62 input logic [dwidth -1:0] d;
63
64 begin
65 // wait initial delay
66 repeat(delay) @(vif.drive_cb);
67 // assert wishbone signal
68 vif.drive_cb.adr <= a;
69 vif.drive_cb.dout <= d;
70 vif.drive_cb.cyc <= 1'b1;
71 vif.drive_cb.stb <= 1'b1;
72 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

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;
```verbatim
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;
```
$\text{cast}(\text{pkt}, \text{tmp\_pkt\_clone})$

// `\text{uvm\_info}(\text{report\_tag}, \text{\$sformatf}("\text{Received packet from driver}\ %0s", \text{pkt\_sprint()}), \text{UVM\_LOW})`

drv\_pkt\_push\_back(pt);

\text{uvm\_test\_done\_raise\_objection}(\text{this});

function void write\_act\_pkt(spi\_sequence\_item tmp\_pkt);

\text{spi\_sequence\_item pkt;}

$\text{cast}(\text{pkt}, \text{tmp\_pkt\_clone})$

// `\text{uvm\_info}(\text{report\_tag}, \text{\$sformatf}("\text{Received packet from DUT}\ %0s", \text{pkt\_sprint()}), \text{UVM\_LOW})`

\text{mon\_pkt\_push\_back(pt)};

endfunction: write\_act\_pkt

task run\_phase(uvm\_phase phase);

// fork

\text{for\_ever\ begin}

wait(\text{mon\_pkt\_size()!}\text{=}0);

op\_pkt = mon\_pkt\_pop\_front();

ip\_pkt = drv\_pkt\_pop\_front();

// if(drv\_pkt\_size()\text{=}0)

// `\text{uvm\_error}("\text{Expected packet was not received in scoreboard}\", \text{UVM\_LOW})`

perform\_check(ip\_pkt, op\_pkt);
65 perform_coverage(ip_pkt);
66 uvm_test_done.drop_objection(this);
67 end
68 // join_none
69 // disable fork;
70 endtask: run_phase
71 //
72 function void perform_coverage(spi_sequence_item pkt);
73 spi_covg.perform_coverage(pkt);
74 endfunction: perform_coverage
75 //
76 function void perform_check(spi_sequence_item ip_pkt,
77 spi_sequence_item op_pkt);
78 if (ip_pkt.exp_master_data==op_pkt.out_master_data &&
   ip_pkt.exp_slave_data==op_pkt.out_slave_data)
79 begin
80 // 'uvm_info(get_full_name(),"Master PASSED",UVM_MEDIUM)
81 // 'uvm_info(get_full_name(),"Slave PASSED",UVM_MEDIUM)
82 pass++;
83 end
84 else
85 begin
86 '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)
I.17 Scoreboard

86     '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)
87             fail ++;
88     end
89     endfunction: perform_check
90     //
91     function void extract_phase(uvm_phase phase);
92     endfunction: extract_phase
93     //
94     function void report_phase(uvm_phase phase);
95     if (fail ==0)
96         begin
97             $display
98                 ("_________________________________________32bit—MSB First—TX:
99                         posedge—RX: negege—_____________________________________")
100            $display
101                 ("_________________________________________TEST
102                         PASSED_________________________________________")
103            $display
104                 ("  
105                         ********************************************************************************
106                 ")
107     uvm_report_info("Scoreboard Report",$sformatf("Trasactions PASS
108                     = %0d FAIL = %0d",pass,fail),UVM_MEDIUM);
$display

("                                                                                                     
                                                                                                           
")

$display

("                                                                                                     
                                                                                                           
")

$display

("                                                                                                     
                                                                                                           
")

end

else

begin

$display

("                                                                                                     
                                                                                                           
")

$display

("                                                                                                     
                                                                                                           
")

else

begin

$display

("                                                                                                     
                                                                                                           
")

$display

("                                                                                                     
                                                                                                           
")

end
uvm_report_info("Scoreboard Report", $sformatf("Transactions PASS = %0d FAIL = %0d", pass, fail), UVM_MEDIUM);

$display
$display(
"**********************************************************************************************************"
);
$display
$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;

'using 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

/*

* 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
output   [32-1:0]  wb_dat_o;  // databus
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_nedge; // slave receiving on negedge
wire tx_nedge; // slave transmitting 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_nedge = ctrl['SPI_CTRL_RX_NEGEDGE];
assign tx_nedge = 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 <= 32'0;
    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 <= 1'0;
    else
        wb_ack_o <= wb_cyc_i & wb_stb_i & ~wb_ack_o;
end
85  //

86  // Wb error
87  assign wb_err_o = 1'b0;
88
89  // Interrupt
90  /* always @(posedge wb_clk_i or posedge wb_rst_i)
91    begin
92      if (wb_rst_i)
93        wb_int_o <= 1'b0;
94      else if (ie && !ss_pad_i && last_bit && pos_edge) // there
95        needs to be rising edge detector
96        wb_int_o <= 1'b1;
97      else if (wb_ack_o)
98        wb_int_o <= 1'b0;
99    end */
100  //

100  // Ctrl register
101  always @(posedge wb_clk_i or posedge wb_rst_i)
102  begin
103    if (wb_rst_i)
104      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
//

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
I.19 SPI Slave Model

123    wb_dat <= wb_dat;
124    end
125    //

126    always @(posedge sclk_pad_i && !tx_negedge) or negedge(
127                sclk_pad_i && tx_negedge))
128    begin
129        miso_pad_o <= wb_dat[31];
130    end
130    //

131    endmodule
132    //
I.20 Test defines

```plaintext
//______________________________________________________________

/*
 *
 * 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;
```
I.20 Test defines

24    *parameter* SPI_TX_1 = 5'h4;
25    *parameter* SPI_TX_2 = 5'h8;
26    *parameter* SPI_TX_3 = 5'hc;
27    *parameter* SPI_CTRL = 5'h10;
28    *parameter* SPI_DIVIDE = 5'h14;
29    *parameter* SPI_SS = 5'h18;

30
31    *logic* scan_in0, scan_in1, scan_en, test_mode;
32    *logic* clock, rstn;
33    *logic* [7:0] ss;
34    *logic* [31:0] q;
35    *logic* sclk, mosi, miso;
36    *logic* tip;
37
38    *typedef* virtual spi_if spi_vif;
39    //---------------------------------------------------------------