DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS
CHENNAI – 600036

Verification of Serial Peripheral Interface Used in Shakti Processor

A Project Report

Submitted by

MARRI K L MOUNIKA

(EE20M076)

In the partial fulfilment of requirements

For the award of the degree

Of

MASTER OF TECHNOLOGY

May 2022

ACKNOWLEDGEMENTS

A successful project work is the result of inspiration, support, guidance, motivation and cooperation of facilities during study. I feel obliged to express my deep sense of gratitude and indebtedness to my guide **Prof. Veezhinathan Kamakoti** for giving me an opportunity to work on this project and for his invaluable guidance and valuable suggestions throughout this work.

I wish to express my wholehearted gratitude to my co-guide **Dr. Janakiraman Viraraghavan** for being my guide from Department of Electrical Engineering which helped me to take up this project outside the EE Department.

I am grateful to my mentor **Lavanya Jagadeeswaran**, Project Officer for providing timely suggestions, valuable guidance and constant involement in the project. I am very thankful to **Divya**, Project Assistant for her detailed guidance, constant support and immense help whenever I was struck with an issue.

I extend my thanks to **Kotteeswaran Ekambaram** for helping me out in case of mismatches and bugs. I am also thankful to **Sriram Rajagopalan** and **Shreyas** for their help in setting up the VIP machine and in SV based verification.

ABSTRACT

Electronic devices became ingrained in our daily lives. It is becoming increasingly impossible for us to work without the use of electronic devices. For electronics to function properly and cater to the need of people, verification of the design in it is of utmost importance. Verification ensures accuracy of design and confirms that design caters to the specifications.

This project work is about verification of Serial Peripheral Interface design integrated with AXI4 bus. It is a full duplex, serial interface, used to communicate within short distances. Speed of this interface is higher than that found in I2C. This interface is commonly used for communication between the microcontrollers and small peripherals. Python-based verification environment CoCoTb is used in development of testbenches for verification of SPI and Verilator is used to perform design simulations.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

AXI Advanced eXtensible Interface.

CPHA Clock Phase.

CPOL Clock Polarity.

DUT Design under Test.

FIFO First In, First Out.

I2C Inter-Integrated Circuit.

MISO Master In Slave Out.

MISO Chip Select.

MOSI Master Out Slave In.

RTL Register Transfer Level.

RX Receive.

SCLK Serial Clock.

SPI Serial Peripheral Interface.

SV System Verilog.

TX Transmit.

UVM Universal Verification Methodology.

VIP Verification IP.

CHAPTER 1

VERIFICATION FRAMEWORK

Electronic devices became an imperative a part of our daily lives. Working without the utilization of electronic devices has grown increasingly challenging. We board a technologically advanced generation where robots and computing can perform human tasks with greater ease and efficiency. As the size and complexity of designs increases, the difficulty to test and verify the design also increases in scope. Day by day, we are reducing the size of electronics to the smallest conceivable size and hence verification is now a very important stage in the product development.

1.1 INTRODUCTION TO VERIFICATION

Verification is a technique that checks to see if the design fulfils the criteria and meets the specifications. The most significant part of the product development stage is verification, which takes up to 80 percent of the total development time. Verification is carried out along with design development. Main goal of verification is to ensure functional correctness before the design is taped out into chip. Verification is a crucial stage because if a bug is discovered after the chip has been released, it can be very costly. The best example of the significance of verification is intel floating point bug i.e, Pentium FDIV bug. Intel announced a pre-tax charge of 475 million dollars against earnings, ostensibly the total cost associated with replacement of the flawed processors. In 2019, this equates to 743 million dollars (about 5500 crores).

1.2 TYPES OF VERIFICATION

There are two types of verification testing:

- Directed Verification
- Constrained Random Verification

Each method has its own pros and cons and depending on the design and its specification, verification engineer should take a call on which type of verification to go with.

1.2.1 Directed Verification

In directed verification, a directed test will be written which has been designed to exercise specific functionality in the design. In this type of verification, engineers spend good amount of time to understand the functionality of design and identify different verification scenarios to cover functionality. Once identification of scenarios is done, directed test bench architecture is defined. This type of verification with a set of directed tests is extremely time-consuming and difficult to maintain for more complex designs. Directed tests only cover cases that the verification team has foreseen by going over specifications, which might result in costly re-spins. Traditionally, Verification IP(VIP) works in a directed test environment.

1.2.2 Constrained Random Verification

In constrained random verification, a random scenario is created with certain constraints which restrict the parameters to values within the specification range to test the design. Constraint random tests can cover a good number of scenarios and/or multiple configurations. This methodology provides an efficient method for achieving coverage targets while also assisting in the detection of corner case issues. Engineers do not need to write as many test cases since a smaller set of constrained-random scenarios combined with a few complete random test scenarios is sufficient to meet coverage targets (functional as well as code coverage).

1.3 VERIFICATION FLOW

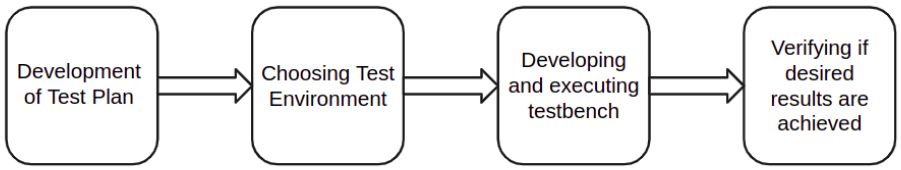

Figure 1.1: Verification Flow

1.3.1 Testplan Development

A test plan is a road map for carrying out verification. To develop a testplan, you'll need a thorough understanding of the design specifications. A test plan is a document that outlines the features to be tested, how to test each feature, and the design parameters to keep in mind while testing each feature. The preparation of a test plan is an essential step in verification because it directs the entire testing process.

1.3.2 Test Environment

Some of the test environments available are:

- Verilog
- System Verilog
- UVM
- Cocotb

Verilog

Back within the 1990's, Verilog was the primary language to verify functionality of styles that were small, not very complex and had less features. Verilog was unable to match the requirements as design complexity rose.

System Verilog

System Verilog, a Hardware Verification Language, is a Verilog extension containing many such verification features that allow engineers to verify their designs in simulation utilising complex testbench structures and random stimuli.

UVM

The Universal Verification Methodology(UVM) is a standardized methodology for verification. It also makes reusing verification components easy.

CoCoTb

CoCoTb is a COroutine based COsimulation TestBench environment for verifying design RTL using Python.Its free and open source.

1.3.3 Testbench Development

A test case is identified based on the test plan. Then, the testbench is created in our chosen environment for each test scenario. A testbench basically provides the input sequence that is needed for the specific test case. The testbench is then run, and the result is observed. If the results are as expected, the test passes; if they are not, the test is considered a failure.

CHAPTER 2

PROJECT GOAL

The purpose of this project is to create an effective testbench for validating the SPI design with AXI4 bus in Cocotb environment. For SPI verification, Cocotb and Verilator are employed. Other goal is to integrate the design with the Verification IP in SV based environment and run a sample test on it to get a comparision between the SV based Verification and Cocotb based verification. The following objectives help to attain the goal:

- Understanding the SPI architecture and design specifications.
- Understanding the open source verification resources such as Cocotb, Verilator and Coverage driven verification.
- Connect the components of the testbench to the design under test.
- Understanding the Verification IP interface.
- Connect the Verification IP to the design under test and developing a sample test to run.

CHAPTER 3

DESIGN SPECIFICATIONS

The goal of this project work is to verify the functionality of the **Serial Peripheral Interface** integrated with **AXI4** bus.

3.1 INTRODUCTION TO SPI

The Serial Peripheral Interface (SPI) is a short-distance synchronous serial communication interface specification. This interface bus is commonly used to send data between microcontrollers and small peripherals such as sensors, ADCs, DACs, SRAM, shift registers and others. It uses separate clock and data lines, along with a select line to choose the device we want to communicate with. The speed of the bus range is substantially higher than that found in I2C; speeds up to 80 MHz are not uncommon. A master-slave architecture is used by SPI devices to communicate in full duplex mode. Sometimes SPI is called a four-wire serial bus. Since SPI is a "synchronous" data bus, it means that it uses separate lines for data and a "clock" that keeps both sides in perfect sync.

3.2 SPI DESIGN FEATURES

The design under verification SPI has the following features:

- The output enables of NCS and SCLK decides whether they are generated by the SPI controller or received from the device connected to the controller.
- LSB/MSB first transmission can be programmed.
- The length of the bit stream to be transferred or received can be programmed.
- 32-level receive and transmit FIFO is provided to reduce servicing overhead.

- 32-bit memory mapped tx and rx registers are used to enqueue and dequeue data to and from FIFO respectively.
- Delayed transmit control i.e, setup and hold time for slave can be achieved.
- Required bit rate can be achieved by programming the prescaller register.
- Supports four clock modes - the idle state of SCLK and the edge at which output has to be transmitted can be varied.
- Five communication statuses are available - busy, transmit enable, receive not enable, FIFO thresholds and overrun.
- FIFO statuses like empty, quad, half and full levels are provided in FIFO status register and interrupts can also be generated for these events.

3.3 INTERFACE

The SPI bus specifies four logic signals:

- SCLK
- MOSI(Master Out Slave In) : Data output from master
- MISO(Master In Slave Out) : Data ouput from slave
- NCS(Chip Select) : Active low output from master

SCLK

SCLK stands for Serial Clock. This signal is the clock pulse which synchronizes data transmission. This is output from master.

MOSI

MOSI stands for Master Out Slave In. This signal is data output from master. It is used for transfer of data from master to slave.

MISO

MISO stands for Master In Slave Out. This signal is data output from slave. It is used for transfer of data from slave to master.

NCS

NCS is Chip Select Signal. N means negative to indicate that the signal is active low signal. This signal is used by master to select the device for communication. It is slave select output from master.

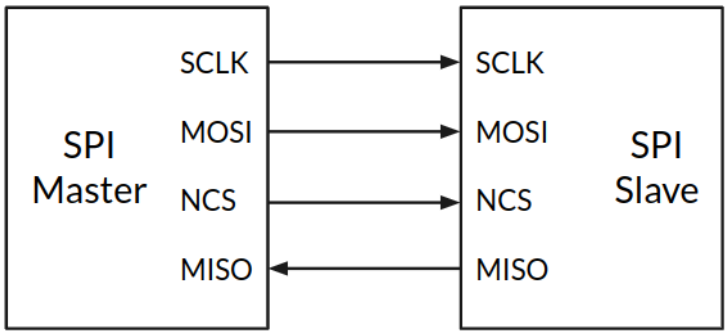

Figure 3.1: SPI Communication with one Master and one Slave

MOSI and MISO on a master connects to MISO and MOSI on a slave respectively. NCS is used instead of an addressing concept and has the same functionality as chip select signal in general.

To start SPI communication, the master must send the clock signal and enable the NCS signal to choose the slave. Since NCS is an active low signal, the master must send a logic 0 on this signal to select the slave. Because SPI is a full-duplex interface, both the master and slave can send data via the MOSI and MISO channels at the same time. During SPI communication, the data is simultaneously transmitted (shifted out serially onto the MOSI) and received (the data on the bus MISO is sampled). The SCLK signal is used to synchronise data shifting and sampling.

3.4 SERIAL CLOCK CONFIGURATION

By providing a precise choice of clock phase and clock polarity, the SPI interface allows the user to sample and/or shift data on the rising or falling edge of the clock.

Table 3.1: The different combinations of CPHA, CPOL and the edge used to input/output the data

CPHA	CPOL	Edge to input/output data
0	0	In idle state, SCLK is 0. Data to be transmitted is sent out on falling edge of SCLK and data to be received is sampled on rising edge of SCLK.
0	1	In idle state, SCLK is 1. Data to be transmitted is sent out on falling edge of SCLK and data to be received is sampled on rising edge of SCLK.
1	0	In idle state, SCLK is 0. Data to be transmitted is sent out on rising edge of SCLK and data to be received is sampled on falling edge of SCLK.
1	1	In idle state, SCLK is 1. Data to be transmitted is sent out on rising edge of SCLK and data to be received is sampled on falling edge of SCLK.

3.4.1 Clock Phase and Clock Polarity

The clock phase and polarity can be chosen by the master. During the idle state, the CPOL bit controls the polarity of the clock signal. The idle state is defined as the time when NCS is high at the start of transmission and transitions to low at the end. The CPHA bit selects the clock phase. Depending on the CPHA bit, the rising or falling clock edge is used to input/output the data. Table 3.1 shows the various combinations of CPHA and CPOL, as well as the edges that are utilised to transmit and receive data.

Figure 3.2 through Figure 3.5 show examples of communication in four different CPHA, CPOL configurations in SPI. The data is displayed on the MOSI and MISO lines in these examples. The dotted green line indicates the start and end of the SPI transaction, orange indicates the edge at which data is received, and the edge at which data is transmitted is indicated in blue.

SCLK Configuration 1: CPHA=0, CPOL=0

Figure 3.2 shows the timing diagram for SPI transaction when CPHA=0 and CPOL=0. The clock polarity is 0 in this mode, indicating that the clock signal's idle condition is low. In this mode, the clock phase is 0, indicating that data is sampled on the rising edge of the SCLK signal (shown by the orange dotted line), and data is transmitted on the falling edge (shown by the dotted blue line).

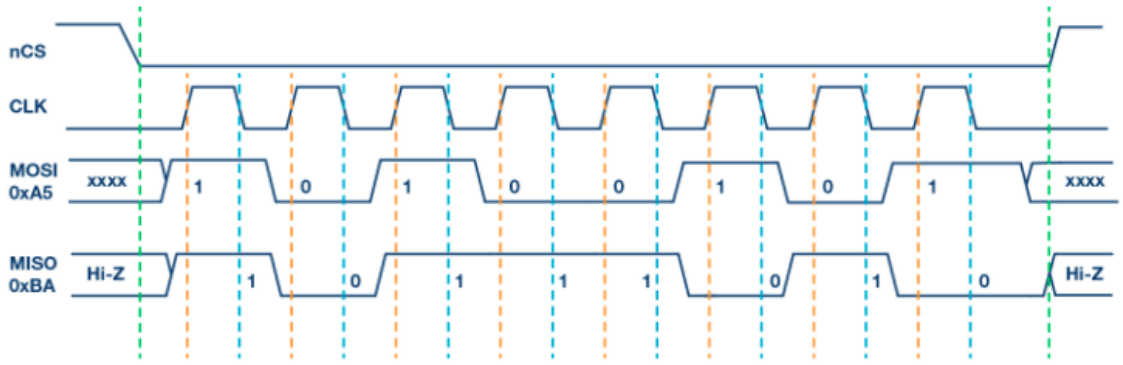

Figure 3.2: SPI Transaction with CPHA=0 and CPOL=0

SCLK Configuration 2: CPHA=0, CPOL=1

Figure 3.3 shows the timing diagram for SPI transaction when CPHA=0 and CPOL=1. The clock polarity is 1 in this mode, indicating that the clock signal's idle state is high. In this mode, the clock phase is 0, indicating that data is sampled on the rising edge of the SCLK signal (shown by the orange dotted line), and data is transmitted on the falling edge (shown by the dotted blue line).

SCLK Configuration 3: CPHA=1, CPOL=0

Figure 3.4 shows the timing diagram for SPI transaction when CPHA=1 and CPOL=0. The clock polarity is 0 in this mode, indicating that the clock signal's idle condition is low. In this mode, the clock phase is 1, indicating that data is sampled on the falling edge of the SCLK signal (shown by the orange dotted line), and data is transmitted on the rising edge (shown by the dotted blue line).

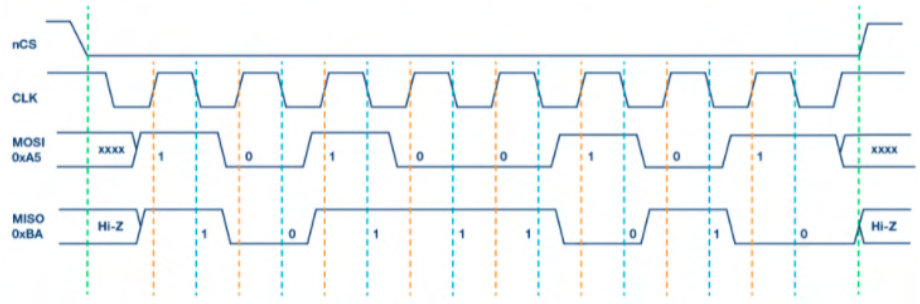

Figure 3.3: SPI Transaction with CPHA=0 and CPOL=1

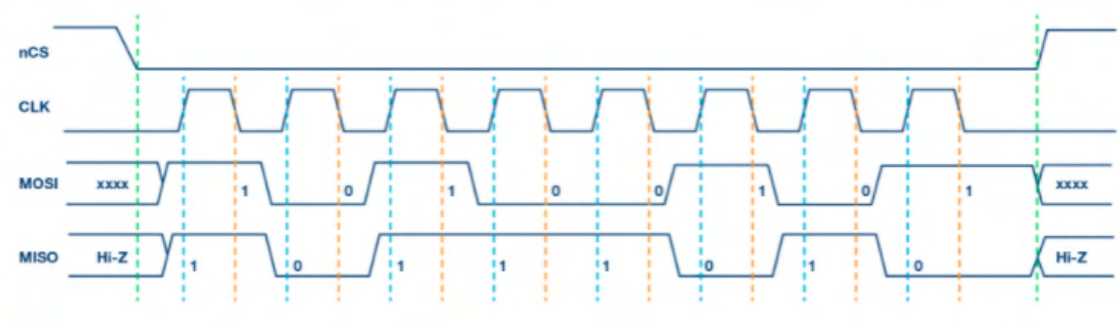

Figure 3.4: SPI Transaction with CPHA=1 and CPOL=0

SCLK Configuration 4: CPHA=1, CPOL=1

Figure 3.5 shows the timing diagram for SPI transaction when CPHA=1 and CPOL=1. The clock polarity is 1 in this mode, indicating that the clock signal's idle state is high. In this mode, the clock phase is 1, indicating that data is sampled on the falling edge of the SCLK signal (shown by the orange dotted line), and data is transmitted on the rising edge (shown by the dotted blue line).

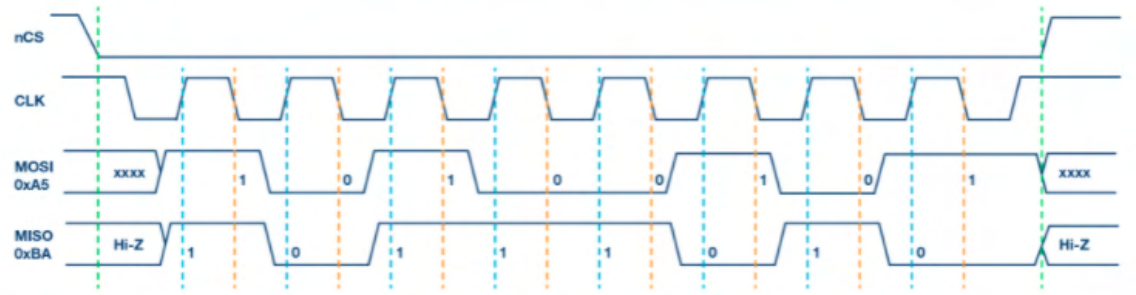

Figure 3.5: SPI Transaction with CPHA=1 and CPOL=1

3.5 COMMUNICATION MODES

SPI has four possible communication modes:

- Only Transmit Mode
- Only Receive Mode
- Transmit and Immediate Receive Mode(Half Duplex)
- Simultaneous Transmit and Receive Mode(Full Duplex)

3.6 CONFIGURATION REGISTERS

SPI has the following configuration registers :

- Communication Control Register
- Clock Control Register
- Transmit Data Register
- Receive Data Register
- Interrupt Enable Register
- FIFO Status Register
- Communication Status Register
- Input Qualification Control Register

Table 3.2: Offset address and access permission about each SPI register

SPI Register Name	**Offset Address**	**Size**	**Access Permission**
Communication Control Register	‘h00	32 bits	Read and Write
Clock Control Register	‘h04	32 bits	Read and Write
Transmit Data Register	‘h08	32 bits	Read and Write
Receive Data Register	‘h0C	32 bits	Read and Write
Interrupt Enable Register	‘h10	16 bits	Read and Write
FIFO Status Register	‘h14	8 bits	Read only
Communication Status Register	‘h18	16 bits	Read only
Input Qualification Control Register	‘h1C	3 bits	Read and Write

3.6.1 Communication Control Register

Communication control register is a 32-bit register used to enable SPI communication, to make SPI module act as master or slave, to set LSB/MSB first transmission, to set the communication mode , to set the number of bits to be transmitted/received and to enable SCLK,MISO,MOSI,NCS pins act as output or input. This register is described in table 3.3 .

Reset Value = 0x02C00000

3.6.2 Clock Control Register

Clock control register is a 32-bit register used to set the clock polarity and clock phase, set a prescaler value to achieve the desired bit rate, to set slave select active to transmit begin delay and transmit end to slave select disable delay. This register is described in table 3.4 .

Reset Value = 0x00000000

Table 3.3: Communication Control Register Field Descriptions

Bit	Field	Description
31-26	Reserved	Reads return 0 and writes have no effect.
25	OUTENMOSI	Output enable for MOSI pin. 0 = MOSI will be an input pin. 1 = MOSI will be an output pin.
24	OUTENMISO	Output enable for MISO pin. 0 = MISO will be an input pin. 1 = MISO will be an output pin.
23	OUTENSCLK	Output enable for SCLK pin. 0 = SCLK will be an input pin. 1 = SCLK will be an output pin.
22	OUTENCS	Output enable for NCS pin. 0 = NCS will be an input pin. 1 = NCS will be an output pin.
21-14	TOTALBITSRX	Total number of bits to be received. The values vary from 0 to 32.
13-6	TOTALBITSTX	Total number of bits to be transmitted. The values vary from 0 to 32.
5-4	COMMMODE	SPI Communication Mode. 00 - Only Transmit Mode. 01 - Only Receive Mode. 10 - Transmit and Immediate Receive Mode(Half Duplex). 11 - Simultaneous Transmit and Receive Mode(Full Duplex).
3	IMMRX	Immediate Receive. 0 = Immediate response is not needed from the slave immediately after the transmission of command. 1 = Immediate response is needed from the slave immediately after the transmission of command.
2	LSBFIRST	LSB or MSB first. 0 = Data is transmitted or received with MSB first. 1 = Data is transmitted or received with LSB first.
1	ENABLE	SPI Enable. This bit enables the SPI transfers. 0 = SPI communication is disabled. 1 = SPI communication is enabled.
0	MASTER	SPI Master. This bit enables master mode. 0 = SPI module acts as slave. 1 = SPI module acts as master.

Table 3.4: Clock Control Register Field Descriptions

Bit	Field	Description
31-26	Reserved	Reads return 0 and writes have no effect.
25-18	TX2NCSDELAY	Transmit end to slave select disable delay. It is the hold time for the slave device. It takes values from 0 to 255 indicating 0 to 255 SCLK cycles.
17-10	NCS2TXDELAY	Slave select active to transmit begin delay. It is the setup time for the slave device. It takes values from 0 to 255 indicating 0 to 255 SCLK cycles.
9-2	PRESCALER	These bits sets the prescaler value. Internal clock frequency is divided by the prescaler value to attain required bit rate. It is a 8-bit value and hence can provide maximum of 255 different data rates. Required Bit Rate = Internal clock frequency/(Prescaler value+1)
1	CLKPHASE	This bit gives the clock offset at which the data to be transmitted or received. 0 = SPI receives data in the first clock transition. 1 = SPI receives data in the second clock transition.
0	CLKPOLARITY	This bit holds the value of SCLK in idle state.

3.6.3 Transmit Data Register

Transmit data register can be a 32-bit register or 16-bit register or 8-bit register which is used to hold the data which is to be transmitted.

3.6.4 Receieve Data Register

Receive data register can be a 32-bit register or 16-bit register or 8-bit register which is used to hold the data which is received.

3.6.5 Interrupt Enable Register

Interrupt enable register is a 32-bit register used to enable TX FIFO and RX FIFO interrupts - interrupts are sent when FIFO is empty, 1/4th full, half-full or full and when overrun occurs. This register is described in table 3.5 and table 3.6 .

Reset Value = 0x00000000

3.6.6 FIFO Status Register

FIFO status register is a 8-bit register used to know the status of TX FIFO and RX FIFO. Each bit of this register will be set according to the status of TX and RX FIFO i.e, empty or quarter-full or half-full or full. This register is described in table 3.7 .

Reset Value = 0x00

3.6.7 Communication Status Register

Communication Status Register is a 8-bit register used to enable SPI transmission and SPI receive, used to know the status of communication and number of entries in FIFO. This register is described in table 3.8 .

Reset Value = 0x0004

3.6.8 Input Qualification Control Register

FIFO status register is a 3-bit register used to enable input qualification control in case of receive mode.

Reset Value = 0

Table 3.5: Interrupt Enable Register Field Descriptions

Bit	Field	Description
15-9	Reserved	Reads return 0 and writes have no effect.
8	RXFIFOOVERRUNERREN	Overrun occurs when the received data cannot be enqueued because the RX FIFO is full. 0 = Overrun interrupt is not sent to PLIC. 1 = Overrun interrupt is sent to PLIC.
7	RXFIFOFULLINTREN	RX FIFO full interrupt enable. 0 = RX FIFO full interrupt is not sent to PLIC. 1 = RX FIFO full interrupt i.e, when RX FIFO is full(has 32 entries), interrupt is sent to PLIC.
6	RXFIFOHALFINTREN	RX FIFO half interrupt enable. 0 = RX FIFO half interrupt is not sent to PLIC. 1 = RX FIFO half interrupt i.e, when RX FIFO is half full(has 16 entries), interrupt is sent to PLIC.
5	RXFIFOQUADINTREN	RX FIFO quad interrupt enable. 0 = RX FIFO quad interrupt is not sent to PLIC. 1 = RX FIFO quad interrupt i.e, when RX FIFO is quad full(has 8 entries), interrupt is sent to PLIC.
4	RXFIFOEMPTYINTREN	RX FIFO empty interrupt enable. 0 = RX FIFO empty interrupt is not sent to PLIC. 1 = RX FIFO empty interrupt i.e, when RX FIFO is empty(has 0 entries), interrupt is sent to PLIC.
3	TXFIFOFULLINTREN	TX FIFO full interrupt enable. 0 = TX FIFO full interrupt is not sent to PLIC. 1 = TX FIFO full interrupt i.e, when TX FIFO is full(has 32 entries), interrupt is sent to PLIC.

Table 3.6: Interrupt Enable Register Field Descriptions Cont.

Bit	Field	Description
2	TXFIFOHALFINTREN	TX FIFO half interrupt enable. 0 = TX FIFO half interrupt is not sent to PLIC. 1 = TX FIFO half interrupt i.e, when TX FIFO is half full(has 16 entries), interrupt is sent to PLIC.
1	TXFIFOQUADINTREN	TX FIFO quad interrupt enable. 0 = TX FIFO quad interrupt is not sent to PLIC. 1 = TX FIFO quad interrupt i.e, when TX FIFO is quad full(has 8 entries), interrupt is sent to PLIC.
0	TXFIFOEMPTYINTREN	TX FIFO empty interrupt enable. 0 = TX FIFO empty interrupt is not sent to PLIC. 1 = TX FIFO empty interrupt i.e, when TX FIFO is empty(has 0 entries), interrupt is sent to PLIC.

3.7 CONFIGURATION OF SPI

To start the SPI transmit or receive, we need to configure the SPI configuration registers as per our requirements in a prescribed order. In this section, we will look into how to configure SPI for transmit and receive.

3.7.1 SPI Transmission

To transmit data, we have to :

1. First, write the data to be transmitted into the transmit data register. Since data bus width is 32 bits according to the specification, if we want to transmit more than 32 bits then we have to write multiple times into the transmit data register and every write will push the contents into the TX FIFO.

2. Next, we have to write into the clock control register as per our specifications.

3. Then, as per our requirement, we have to write into the interrupt enable register.

4. Then , we have to write into the input qualification control register as per our criteria.

Table 3.7: FIFO Status Register Field Descriptions

Bit	Field	Description
7	RXFIFOFULL	RX FIFO full status bit. 0 = RX FIFO is not full. 1 = RX FIFO is full(has 32 entries).
6	RXFIFOHALF	RX FIFO half status bit. 0 = RX FIFO is not half full. 1 = RX FIFO is half full(has 16 entries).
5	RXFIFOQUAD	RX FIFO quad status bit 0 = RX FIFO is not quad full. 1 = RX FIFO is quad full(has 8 entries).
4	RXFIFOEMPTY	RX FIFO empty status bit. 0 = RX FIFO is not empty. 1 = RX FIFO is empty(has 0 entries).
3	TXFIFOFULL	TX FIFO full status bit. 0 = TX FIFO is not full. 1 = TX FIFO is full(has 32 entries).
2	TXFIFOHALF	TX FIFO half status bit. 0 = TX FIFO is not half full. 1 = TX FIFO is half full(has 16 entries).
1	TXFIFOQUAD	TX FIFO quad status bit. 0 = TX FIFO is not quad full. 1 = TX FIFO is quad full(has 8 entries).
0	TXFIFOEMPTY	TX FIFO empty status bit. 0 = TX FIFO is not empty. 1 = TX FIFO is empty(has 0 entries).

5. Lastly, we have to write into the communication control register accordingly and enable SPI transmit.

3.7.2 SPI Receive

To receive data, we have to :

1. First, we have to write into the clock control register as per our criteria.

2. Then, as per our requirement, we have to write into the interrupt enable register.

3. Next , we have to write into the input qualification control register as per our requirement.

4. Now, we have to write into the communication control register accordingly and enable SPI receive.

5. The received data will be available in the receive data register. Since data bus width is 32 bits according to the specification, if we want to receive more than 32 bits then we have to read multiple times from the receive data register and every read will pop the contents from the RX FIFO.

3.8 OPERATION

SPI can operate as a master or as a slave.

3.8.1 SPI Master Mode

As soon as SPI communication is enabled via communication control register, the following happens sequentially,

1. After one cycle, slave select signal NCS goes low.

2. In the next cycle, SCLK counter starts and upcounts till prescale value. The SCLK signal value inverts or remains the same depending on the clock phase and clock polarity set in the clock control register. This SCLK signal generation happens throughout the SPI transaction.

3. If any setup delay set is set in the clock control register, SCLK signal is not sent as output till the setup delay is met. If there is no setup delay, SCLK signal will be sent as output immediately in the next cycle after NCS signal goes low.

4. Once the setup delay is met, transmit from the controller happens when SCLK counter is zero and that value is held till the counter reaches the prescaler value.

5. The controller reads an input value when SCLK counter is equal to half the prescaler value.

6. Only transmit, only receive, transmit and immediate receive or simultaneous transmit and receive happens based on the communication mode set in the communication control register.

7. The transmit/receive/ both transmit and receive continues till the number of bits to be transmitted or the number of bits to be received is done according to the configuration in the communication control register.

8. Once the transmit - receive is done, the SCLK signal stops but NCS signal will go high only after the hold delay is met according to the configuration in the clock control register. This completes the SPI transaction and SPI enable signal will also be reset to zero. To start the next transaction we need to configure the SPI configuration registers accordingly again.

3.8.2 SPI Slave Mode

Slave mode is similar to the master mode except that NCS signal and SCLK signal will be received from the SPI device connected to the controller. The SPI enable signal is expected to be set before the NCS signal goes low and once the NCS signal goes low, the SPI state changes happens same as in master mode i.e, SPI enable signal can be changed to SPI enable bit. In slave mode, SCLK counter will not be used, instead the rising or falling edge is detected and transmit and receive edge is computed. Once the NCS signal goes high, SPI enable signal is reset by the controller.

3.9 SPI DUT BLOCK DIAGRAM

Fig 3.6 shows the block diagram of the design under test in this project i.e, SPI.

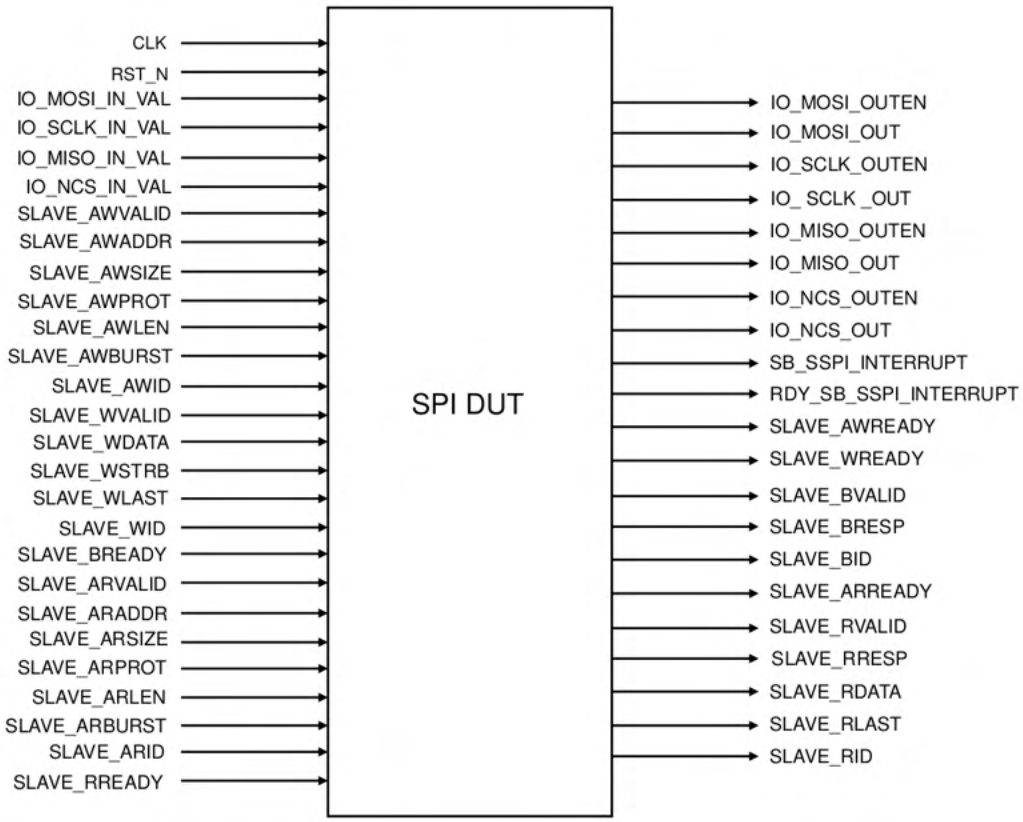

Figure 3.6: Block Diagram of the SPI DUT

Table 3.8: Communication Status Register

Bit	Field	Description
7	OVERRUN	This bit is set when RX FIFO is full(has 32 entries) and there is a write to the RX data register.
6-5	RXFIFO	RX FIFO Level. 00 - RX FIFO is empty. 01 - RX FIFO is quad full. 10 - RX FIFO is half full. 11 - RX FIFO is full.
4-3	TXFIFO	TX FIFO Level. 00 - TX FIFO is empty. 01 - TX FIFO is quad full. 10 - TX FIFO is half full. 11 - TX FIFO is full.
2	RXNE	Receiver Not Enable. When this is reset(0), the data is being received. Once the receive operation starts this bit will be reset and after completion of receive operation this bit will be set.
1	TXE	Transmitter Enable. When this is set(1), the data is being transmitted. Once transmit operation starts this bit will be set and after completion of reset operation this bit will be reset. Reset value is 0.
0	BUSY	SPI Busy. 1 = SPI communication is being carried out. 0 = SPI is in Idle state. When NCS goes low, this bit will be set and when NCS goes high this bit will be reset. Reset value is 0.

CHAPTER 4

TESTPLAN

Table 4.1: Test plan for Verification of SPI-1

Features	**Sub-Features**	**Testplan**
MOSI OUTPUT ENABLE REGISTER	Reserved	1. Set this to 1 and check if output is being transmitted or not. 2. Set this to 0 and check if data is being received or not.
MISO OUTPUT ENABLE REGISTER	1 bit register storing the MISO pin's output enable. If set, output is transmitted through this pin else it is read from this pin. Reset value is 0.	1. Set this to 1 and check if output is being transmitted or not. 2. Set this to 0 and check if data is being received or not.
NCS OUTPUT ENABLE REGISTER	1 bit register storing the NCS pin's output enable. If set, the controller generates NCS(Master mode) else NCS is expected from the SPI device(Slave mode). Reset value is 1.	1. Set this to 1 and check if NCS is being internally generated or not. 2. Set this to 0 and check if NCS is being received or not.
SCLK OUTPUT ENABLE REGISTER	1 bit register storing the SCLK pin's output enable. If set, the controller generates the SCLK(Master mode) else SCLK is expected from the SPI device(Slave mode). Reset value is 1.	1. Set this to 1 and check if SCLK is being internally generated or not. 2. Set this to 0 and check if SCLK is being received or not."

Table 4.2: Test plan for Verification of SPI-2

Features	**Sub-Features**	**Testplan**
TOTAL BIT TX REGISTER	8 bit register storing the total number of bits to be transmitted in a SPI transaction. Reset value is 0.	1. Assign a random number to this register (less than 32 as data bus is only 32 bits wide to perform data transfer in a single transaction) and check if the number of bits transmitted is matching the value stored by this register. 2.Assign a random number to this register (greater than 32 as data bus is only 32 bits wide to perform data transfter in mutiple transactions) and check if the number of bits transmitted is matching the value stored by this register.
TOTAL BIT RX REGISTER	8 bit register storing the total number of bits to be received in a SPI transaction. Reset value is 0.	Assign a random number to this register and check if the number of bits received is matching the value stored by this register.
COMM MODE REGISTER	2 bit register stroring the communication mode of the SPI transaction. 00 indicates only transmit mode, 01 indicates only receive mode, 10 indicates transmit and immediate receive mode and 11 indicates transmit and receive mode. Reset value is 0.	1. Set this to 00 and confirm that transmission is going on and nothing is being received. 2. Set this to 01 and confirm that receiving is going on and nothing is being transmitted. 3. Set this to 10 and confirm that receiving happens only after completion of transmission. 4. Set this to 11 and see if both transmission and receiving are happening simultaneously."
LSBFIRST REGISTER	1 bit register storing whether the SPI transaction is LSB first. If set, it means LSB first else it means MSB first. Reset value is 0.	1. Set this to 1 and check if LSB is first. 2. Set this to 0 and check if MSB is first."
SPI EN REGISTER	1 bit register storing the SPI enable control. Once this bit is set, the SPI transaction will start and it will be reset at the end of SPI transaction. Reset value is 0.	1. Set this to 1 and see if SPI transaction starts. 2. Also check that this bit should be 0 after completion of the transaction."

Table 4.3: Test plan for Verification of SPI-3

Features	**Sub-Features**	**Testplan**
T CS DELAY REGISTER	8 bit register storing the hold delay. Reset value is 0.	Store a random value in this register and check that NCS goes high only after this delay..
CS T DELAY REGISTER	8 bit register storing the setup delay. Reset value is 0.	Store a random value in this register and check that SCLK is sent only after this delay.
PRESCALLER REGISTER	8 bit register storing the prescaller value of the SCLK. Reset value is 0.	Store a random value in this register and check that SCLK generation only happens after the SCLK counter counts till prescaller value.
CLK PHASE REGISTER	1 bit register storing the clock phase. Reset value is 0.	Store a random value and check for sclk inversion.
CLK POLARITY REGISTER	1 bit register storing the clock polarity. Reset value is 0.	Store a random value and check for sclk inversion.
TX DATA REGISTER	32 bit register storing the data to be transmitted. This register is written by the AXI write request and once written the data is transffered to TX FIFO. Reset value is 0.	Check that AXI write request properly writes data to this register and also that the data is being transferred to TX FIFO.
RX DATA REGISTER	32 bit register storing the data to be received. This register is read by the AXI read request. The data is written from the RX FIFO. Reset value is 0.	Check that data is properly written from RX FIFO to this register and AXI read request reads this data.
RX OVER RUN ERR INTR ERR REGISTER	1 bit register storing overrun interrupt enable bit. Overrun occurs when the received data cannot be enqueued because the RX FIFO is full. When set, ovverrun interrupt is sent to PLIC. Reset value is 0.	Store 32 entries into RX FIFO and enable this bit. Check that overrun interrupt is sent to PLIC.
RX FIFO FULL INTR EN REGISTER	1 bit register storing RX FIFO full interrupt enable bit. When set, interrupt is sent to PLIC when RX FIFO is full i.e, 32 entries. Reset value is 0.	Store 32 entries into RX FIFO and enable this bit. Check that RX FIFO full interrupt is sent to PLIC.
RX FIFO HALF INTR EN REGISTER	1 bit register storing RX FIFO half interrupt enable bit. When set, interrupt is sent to PLIC when RX FIFO is half-full i.e, 16 entries. Reset value is 0.	Store 16 entries into RX FIFO and enable this bit. Check that RX FIFO half interrupt is sent to PLIC.

Table 4.4: Test plan for Verification of SPI-4

Features	**Sub-Features**	**Testplan**
RX FIFO QUAD INTR EN REGISTER	1 bit register storing RX FIFO full interrupt enable bit. When set, interrupt is sent to PLIC when RX FIFO has 8 entries. Reset value is 0.	Store 8 entries into RX FIFO and enable this bit. Check that RX FIFO quad interrupt is sent to PLIC.
RX FIFO EMPTY INTR EN REGISTER	1 bit register storing RX FIFO full interrupt enable bit. When set, interrupt is sent to PLIC when RX FIFO is empty. Reset value is 0.	Store 0 entries into RX FIFO and enable this bit. Check that RX FIFO empty interrupt is sent to PLIC.
TX FIFO FULL INTR EN REGISTER	1 bit register storing TX FIFO full interrupt enable bit. When set, interrupt is sent to PLIC when TX FIFO is full i.e, 32 entries. Reset value is 0.	Store 32 entries into TX FIFO and enable this bit. Check that TX FIFO full interrupt is sent to PLIC.
TX FIFO HALF INTR EN REGISTER	1 bit register storing TX FIFO half interrupt enable bit. When set, interrupt is sent to PLIC when TX FIFO is half-full i.e, 16 entries. Reset value is 0.	Store 16 entries into TX FIFO and enable this bit. Check that TX FIFO half interrupt is sent to PLIC.
TX FIFO QUAD INTR EN REGISTER	1 bit register storing TX FIFO full interrupt enable bit. When set, interrupt is sent to PLIC when TX FIFO has 8 entries. Reset value is 0.	Store 8 entries into TX FIFO and enable this bit. Check that TX FIFO quad interrupt is sent to PLIC.
TX FIFO EMPTY INTR EN REGISTER	1 bit register storing TX FIFO full interrupt enable bit. When set, interrupt is sent to PLIC when TX FIFO is empty. Reset value is 0.	Store 0 entries into TX FIFO and enable this bit. Check that TX FIFO empty interrupt is sent to PLIC.
OVER RUN REGISTER	1 bit register storing overrun bit. When there is an overrun during receive operation, this bit will be set. Reset value is 0.	Store 32 entries in RX FIFO and perform one more receive operation. Check that this bit should be set.

Table 4.5: Test plan for Verification of SPI-5

Features	**Sub-Features**	**Testplan**
RX FIFO TH REGISTER	2 bit register storing RX FIFO threshold bits to know the number of entires in the RX FIFO. Reset value is 0.	Set random number of entries in RX FIFO and check if no of entries<8, rg_rx_fifo_th=0, if no of entries is 8-15 then rg_rx_fifo_th=1, if no of entries is 16-23 then rg_rx_fifo_th=2 and if no of entries>24 then rg_rx_fifo_th = 3.
TX FIFO TH REGISTER	2 bit register storing TX FIFO threshold bits to know the number of entires in the TX FIFO. Reset value is 0.	Set random number of entries in TX FIFO and check if no of entries<8, rg_tx_fifo_th=0, if no of entries is 8-15 then rg_tx_fifo_th=1, if no of entries is 16-23 then rg_tx_fifo_th=2 and if no of entries>24 then rg_tx_fifo_th = 3.
RXNE REGISTER	1 bit register storing receive not enable bit. Once the receive operation starts this bit will be reset and after completion of receive operation this bit will be set. Reset value is 1.	Check that when receive operation starts this bit should be 1 and after completion of receive this should be 0.
TXE REGISTER	1 bit register storing transmit enable bit. Once transmit operation starts this bit will be set and after completion of transmit operation this bit will be reset. Reset value is 0.	Check that when transmit operation starts this bit should be 1 and after completion of transmission this should be 0.
BUSY REGISTER	1 bit register storing SPI busy bit. When NCS goes low, this bit will be set and when NCS goes high this bit will be reset. Reset value is 0.	Check that after NCS goes low this bit sets to indicate that SPI is busy and a transmission/receiving is going on.
RX FIFO FULL REGISTER	1 bit register storing the RX FIFO full status bit. This bit will be set when RX FIFO is full i.e, 32 entries. Reset value is 0.	Store 32 values in RX FIFO and check if this bit is set or not. Store a random number of values other than 32 in RX FIFO and check if this bit is 0 or not.

Table 4.6: Test plan for Verification of SPI-6

Features	**Sub-Features**	**Testplan**
RX FIFO HALF REGISTER	1 bit register storing the RX FIFO half status bit. This bit will be set when RX FIFO is half-full i.e, 16 entries. Reset value is 0.	Store 16 values in TX FIFO and check if this bit is set or not. Store a random number of values other than 16 in TX FIFO and check if this bit is 0 or not.
RX FIFO QUAD REGISTER	1 bit register storing the RX FIFO quad status bit. This bit will be set when RX FIFO has 8 entries. Reset value is 0.	Store 8 values in RX FIFO and check if this bit is set or not. Store a random number of values in RX FIFO and check if this bit is 0 or not.
RX FIFO EMPTY REGISTER	1 bit register storing the RX FIFO empty status bit. This bit will be set when RX FIFO is empty . Reset value is 0.	Store some value in RX FIFO and check the value of this bit(should be 0). Empty RX FIFO and check the value of this bit(should be 1).
TX FIFO FULL REGISTER	1 bit register storing the TX FIFO full status bit. This bit will be set when TX FIFO is full i.e, 32 entries. Reset value is 0.	Store 32 values in TX FIFO and check if this bit is set or not. Store a random number of values other than 32 in TX FIFO and check if this bit is 0 or not.
TX FIFO HALF REGISTER	1 bit register storing the TX FIFO half status bit. This bit will be set when TX FIFO is half-full i.e, 16 entries. Reset value is 0.	Store 16 values in TX FIFO and check if this bit is set or not. Store a random number of values other than 16 in TX FIFO and check if this bit is 0 or not.
TX FIFO QUAD REGISTER	1 bit register storing the TX FIFO quad status bit. This bit will be set when TX FIFO has 8 entries. Reset value is 0.	Store 8 values in TX FIFO and check if this bit is set or not. Store a random number of values other than 8 in TX FIFO and check if this bit is 0 or not.
TX FIFO EMPTY REGISTER	1 bit register storing the TX FIFO empty status bit. This bit will be set when TX FIFO is empty . Reset value is 0.	Store some value in TX FIFO and check the value of this bit(should be 0). Empty TX FIFO and check the value of this bit(should be 1).
NCS REGISTER	1 bit NCS register. This register will be set by the controller in master mode whereas it will be set from ncs io input in slave mode. Reset value is 1.	Set this to 1 and check if controller is generating NCS or not. Set this to 0 and check if NCS is being received or not.

Table 4.7: Test plan for Verification of SPI-7

Features	**Sub-Features**	**Testplan**
SCLK REGISTER	1 bit SCLK register. This register will be set by the controller in master mode whereas it will be set from sclk to input in slave mode. Reset value is 0.	Set this to 1 and check if controller is generating SCLK or not. Set this to 0 and check if SCLK is being received or not.
PREV SCLK REGISTER	1 Bit previous SCLK register. This register holds the previous value of sclk register to detect the edges in slave mode. Reset value is 0.	Perform two transactions and see if this register is correctly getting the sclk.
CLK COUNTER REGISTER	8 bit clock counter register used in master mode to generate SCLK which continously upcounts till clock prescale value and resets. This counting operation will be active throughout the SPI transaction. Reset value is 0.	Store a random value in prescaller register and check that this register should count till the prescale value and after the counter reaches prescale value SCLK is generated.
TXDATA TO TXFIFO REGISTER	1 bit register used to enable the transfer of TX data from tx_data register to TX FIFO. When this bit is set, the contents from TX data is read and enqueued to TX FIFO. The ARSIZE/AWSIZE from AXI read and write request is used to decide the enqueue length. Reset value is 0.	Set this bit to 1 and check the enqueuing of data to TX FIFO with different AWSIZE/ARSIZE values.
RXFIFO TO RXDATA REGISTER	1 bit register used to enable the transfer of data from RX FIFO to rx_data register. When this bit is set, the contents from RX FIFO is transferred to rx_data register. The ARSIZE/AWSIZE from AXI read and write request is used to decide the dequeue length. Reset value is 0.	Set this bit to 1 and check the dequeuing of data from RX FIFO with different AWSIZE/ARSIZE values.
TXFIFO ENQ SIZE REGISTER	3 bit register used to hold the AWSIZE from tx_data AXI write request. This decides the number of elements that will be enqueued to TX FIFO. Reset value is 0.	In above to set different values, store a random number in this register.

Table 4.8: Test plan for Verification of SPI-8

Features	**Sub-Features**	**Testplan**
CURR TX BYTE REGISTER	8 bit register storing the current TX byte that is being transmitted. Reset value is 0.	Simply stores current byte.
CURR RX BYTE REGISTER	8 bit register storing the current RX byte that is being received. Reset value is 0.	Simply stores current byte.
COUNT TX DATA BITS REGISTER	8 bit register storing the count of number of TX bits transmitted in a SPI transaction. Reset value is 0.	Simply stores count of TX bits. Check that count is being stored properly.
COUNT RX DATA BITS REGISTER	8 bit register storing the count of number of RX bits received in a SPI transaction. Reset value is 0.	Simply stores count of RX bits. Check that count is being stored properly.
COUNT TX DATA REGISTER	3 bit register storing the count of number of bits transmitted in the current TX byte. Reset value is 0.	Simply stores count of TX bits in current byte. Check that count is being stored properly.
COUNT RX DATA REGISTER	3 bit register storing the count of number of bits transmitted in the current RX byte. Reset value is 0.	Simply stores count of RX bits in current byte. Check that count is being stored properly.
TRANSMIT DATA REGISTER	1 bit register storing the current bit that is being transmitted. Reset value is 0.	Simply stores the bit.
TRANSMIT STATE REGISTER	ENUM based register storing data type transmit_state. This register stores the transmit state of the transmitter. Reset value is IDLE.	Storage register, nothing to check.
RECEIVE STATE REGISTER	ENUM based register storing data receive_state. This register stores the receive state of the receiver. Reset value is IDLE.	Storage register, nothing to check.
ACTIVE REGISTER	ENUM based register storing data type spi_state. This register stores the SPI state. Reset value is IDLE.	Storage register, nothing to check.

CHAPTER 5

TOOLS AND PACKAGES USED

5.1 VERILATOR

Verilator is run with parameters similar to Synopsys's VCS. It reads the provided Verilog or SystemVerilog code, does lint checks, and optionally inserts assertion checks and coverage-analysis points, and then "Verilates" it. Verilator in this project is used for design simulation and is also used to collect code coverage.

5.2 COCOTB

CoCoTb is COroutine based COsimulation TestBench environment. It is based on python and is used for System Verilog RTL and HDL verification. It is open-source and completely free. CoCoTb follows UVM's design reuse and randomised testing principles, however it's written in Python. CoCoTb is provided as a python library.

5.2.1 Advantages of CoCoTb

- Tests are automatically discovered by CoCoTb and hence we need not take the pain and add a test to regression.
- Python is used for entire verification and has multiple advantages over HDL and SV. HDL is not suitable for verification of complex designs. SV or UVM can be used, but along with being powerful languages they are very complicated too. Python, on the other hand is easy to learn and use, includes a large library of ready-to-use code which can be re-used and mainly without needing to recompile the design or exit the simulator GUI, tests can be changed and re-run.

5.2.2 Architecture

A standard simulator is used to execute the design under test (DUT). In the simulator, it is instantiated as the toplevel. CoCoTb acts as a bridge between the simulator and Python and employs the Verilog Procedural Interface (VPI) or the VHDL Procedural

Interface(VHPI) for the same. It applies the stimulus to the DUT's inputs and keeps track of the output from Python.

Python testbench is nothing but simple python function known as a coroutine. It can change values in the DUT hierarchy, wait for simulation time to elapse and also keep an eye on the rising edge and falling edge of the signals.

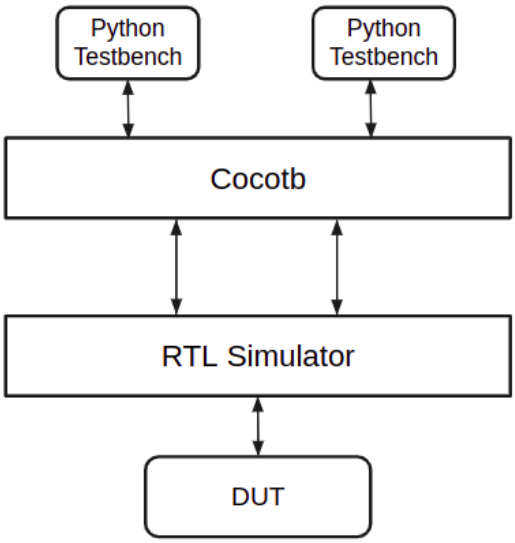

Figure 5.1: Basic CoCoTb Architecture

5.2.3 Using CoCoTb

To indicate which files to include in simulation, each CoCoTb project requires a Makefile. Fig 5.2 shows the Makefile used in this project for verification of SPI.

Components of the Makefile are :

- SIM : Sets the simulator to use. In this project, we are using Verilator.
- TOPLEVEL_LANGUAGE : Used to tell the simulator, the top level language of RTL. This can be Verilog or VHDL. In this project, the toplevel language is Verilog.
- COCOTB_HDL_TIMEUNIT : Used to set the time unit of the timescale. Time unit is the unit of measurement for delays and simulation time.

- COCOTB_HDL_TIMEPRECISION : Used to set the time precision of the timescale. Time precision defines how delay quantities are rounded off before they are used in simulation.
- EXPORT PARAM : This is used to set the AXI4 bus parameters in the design.
- COMPILE_ARGS : This is used to provide compile arguments.
- PWD : PWD is the present work directory.
- TOPDIR : This is the top directory and this is passed from the terminal in the run time.
- VERILOG_SOURCES : This is used to tell the RTL files to include.
- TOPLEVEL : This is used to tell the top level RTL module to instantiate.
- MODULE : This is used to tell which Python testbench to use.

5.2.4 Testbench Structure

Fig 5.3 shows the CoCoTb testbench architecture. We will discuss the testbench components in detail in this section.

Input Transaction

Input Transaction has the stimulus which has to be driven to the DUT.

CoCoTb Driver

CoCoTb Driver takes in the stimulus from the input transaction and drives it to the DUT. The driver is in charge of serialising transactions to the interface's physical pins.

Input Monitor

Input Monitor is responsible for capturing the input signal activity to the DUT. It also calls the reference model and sends the input information to it.

Output Monitor

Output Monitor is responsible for capturing the output signal activity from the DUT. The obtained result from DUT taken by the output monitor and sends to the scoreboard.

```
SIM ?= verilator
TOPLEVEL_LANG ?= verilog

COCOTB_HDL_TIMEUNIT = 1ns
COCOTB_HDL_TIMEPRECISION = 1ns

export PARAM_DATA_WIDTH ?= 32
export PARAM_ADDR_WIDTH ?= 64
export PARAM_STRB_WIDTH ?= $(shell expr $(PARAM_DATA_WIDTH) / 8 )
export PARAM_ID_WIDTH ?= 4
export PARAM_AWUSER_WIDTH ?= 1
export PARAM_WUSER_WIDTH ?= 1
COMPILE_ARGS += -GRUSER_WIDTH=$(PARAM_RUSER_WIDTH)

PWD=$(shell pwd)

VERILOG_SOURCES = $(PWD)/top.v
VERILOG_SOURCES += $(TOPDIR)/mkdummy.v
VERILOG_SOURCES += $(TOPDIR)/BRAM2BELoad.v
VERILOG_SOURCES += $(TOPDIR)/ClockInverter.v
VERILOG_SOURCES += $(TOPDIR)/FIFO1.v
VERILOG_SOURCES += $(TOPDIR)/FIFO2.v
VERILOG_SOURCES += $(TOPDIR)/FIFO10.v
VERILOG_SOURCES += $(TOPDIR)/FIFO20.v
VERILOG_SOURCES += $(TOPDIR)/MakeClock.v
VERILOG_SOURCES += $(TOPDIR)/MakeReset0.v
VERILOG_SOURCES += $(TOPDIR)/ResetEither.v
VERILOG_SOURCES += $(TOPDIR)/SyncFIFO1.v
VERILOG_SOURCES += $(TOPDIR)/SyncReset0.v


TOPLEVEL = top

export TOPLEVEL_LANG
MODULE=test_sspi_txtoncsdelay

include $(shell cocotb-config --makefiles)/Makefile.sim
```

Figure 5.2: Makefile used for Verification of SPI

Reference Model

Reference model is a test model of the DUT. It is a golden model that predicts the correct results from the provided stimulus. This takes in the inputs from the input monitor and generates the expected data.

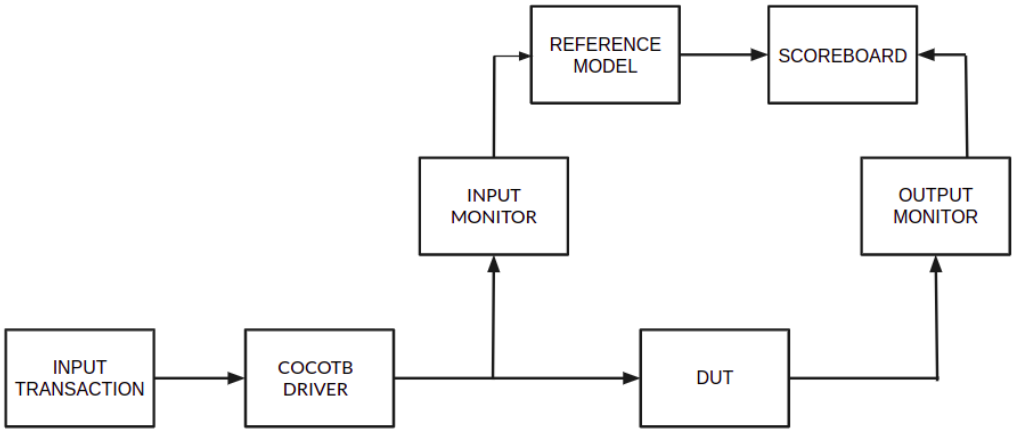

Figure 5.3: CoCoTb Testbench Structure

Scoreboard

Scoreboard checks the design's functionality. It compares the expected data from the reference model with the data collected from the output monitor to determine whether or not the DUT is performing as intended.

5.3 OTHER PYTHON MODULES AND CLASSES

5.3.1 Modules Used

Random

The Random module is employed for generation of random numbers.

Sys

The Sys module provides us with access to system-specific parameters and functions.

Math

The Math module contains a set of methods and constants that can be used to solve mathematical problems.

PPrint

Pprint module provides a capability to arbitrary Python data structures.

Logging

Logging module is used to log messages that we want to see.

Yaml

YAML parser and emitter for Python.

OS

OS module provides functions for creating and removing a directory, fetching its contents, changing and identifying the present directory.

Pytest

PyTest is a testing framework that permits users to write down test codes using Python programming language.

Itertools

Itertools is a module which implements a number of iterator building blocks.

5.3.2 Classes Used

Coroutine

Coroutine is class from cocotb.decorators. It is a decorator class that allows us to provide common coroutine mechanisms.

Timer

Timer is a class from cocotb.triggers. cocotb.triggers is a collections of triggers which a testbench can await. Timer is used to elapse specified simulation time period.

RisingEdge

RisingEdge is a class from cocotb.triggers. Fires on the rising fringe of signal, on a transition from 0 to 1.

FallingEdge

FallingEdge is a class from cocotb.triggers. Fires on the falling fringe of signal, on a transition from 1 to 0.

BusMonitor

BusMonitor is a class from cocotb_bus.monitors. It is a wrapper providing common functionality for monitoring buses.

BusDriver

BusDriver is a class from cocotb_bus.drivers. It is a wrapper around common functionality for buses which have a list of signals, optional_signals, a clock, a name and an entity.

BinaryValue

BinaryValue is a class from cocotb.binary. It is used for representation of values in binary format.

TestFactory

TestFactory is a class from cocotb.regression. cocotb.regression has all things relating to regression capabilities. Testfactory is a factory to automatically generate tests.

Scoreboard

Scoreboard is a class from cocotb_bus.scoreboard. It is a generic scoreboarding class.

TestFailure

TestFailure is a class from cocotb.result. It is used to show that test was completed with severity failure.

Clock

Clock is a class from cocotb.clock. It is a simple 50-50 duty cycle clock driver.

AxiBus, AxiMaster, AxiRam

AxiBus, AxiMaster, AxiRam are classes from cocotbext.axi. They are the AXI modules.

Cocotb_coverage.coverage

cocotb_coverage.coverage helps us to use functional coverage techniques.

CHAPTER 6

RESULTS

In this chapter, we will see simulation results for a few test-cases, functional coverage information and code coverage information.

6.1 TOTAL_BIT_TX

Feature:

8 bit register storing the total number of bits to be transmitted in a SPI transaction.

Testplan:

1. Assign a random number to this register (less than 32 as data bus is only 32 bits wide to perform data transfer in a single transaction) and check if the number of bits transmitted is matching the value stored by this register.

2. Assign a random number to this register (greater than 32 as data bus is only 32 bits wide to perform data transfter in mutiple transactions) and check if the number of bits transmitted is matching the value stored by this register.

Single Transmit Transaction

Fig 6.1 shows the simulation of testcase for single transaction. As sspi_rg_clk_phase is 0 and sspi_rg_clk_polarity is 1, data to be transmitted is sent out on rising edge of io_sclk_out on the io_mosi_out line. sspi_rg_total_bit_tx shows the total number of bits to be transmitted and the data to be transmitted is stored in sspi_rg_tx_data. sspi_rg_count_tx_databits counts the bits of data getting transmitted. The data in the sspi_rg_tx_data register is compared with the data getting transmitted on io_mosi_out line manually and then verified.

Fig 6.2 shows the functional coverage information for this testcase. The functional coverage is taken for the total bits to be transmitted in a single transaction(bits to be

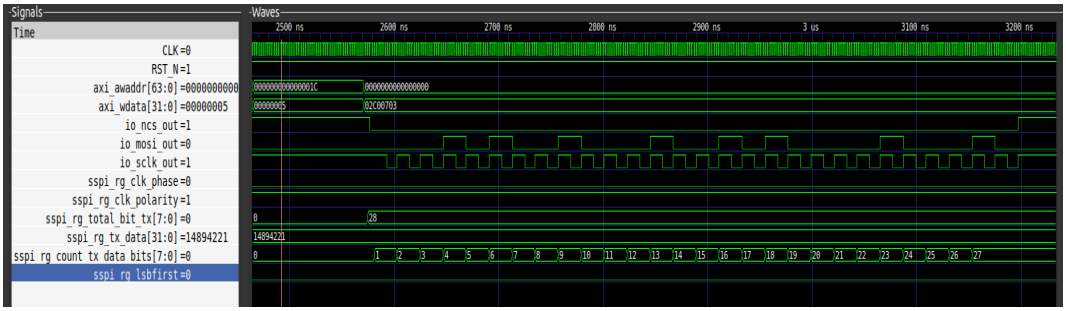

Figure 6.1: Simulation for transmit transaction for checking total_bit_tx

transmitted are between 0 and 32).

Functional coverage for total_bit_tx in single transmit transaction = 100 percent

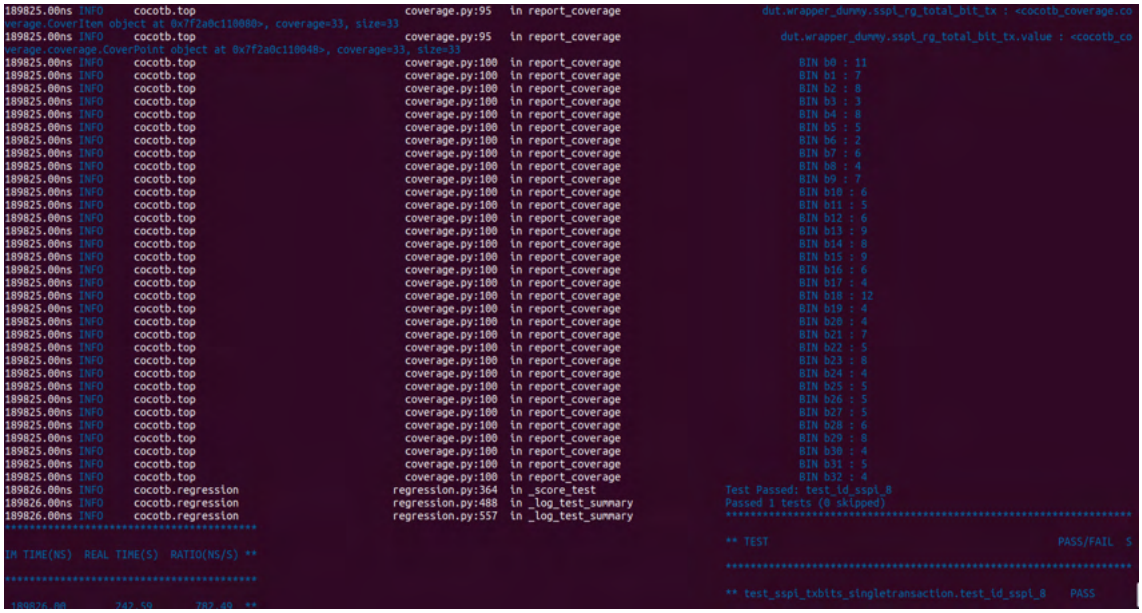

Figure 6.2: Functional coverage for total_bit_tx in single transmit transaction

Two Transmit Transactions

Fig 6.3 shows the functional coverage information for this testcase. The functional coverage is taken for the total bits to be transmitted in two transactions(bits to be transmitted are between 33 and 64).

Functional coverage for total_bit_tx in two transmit transactions = 100 percent

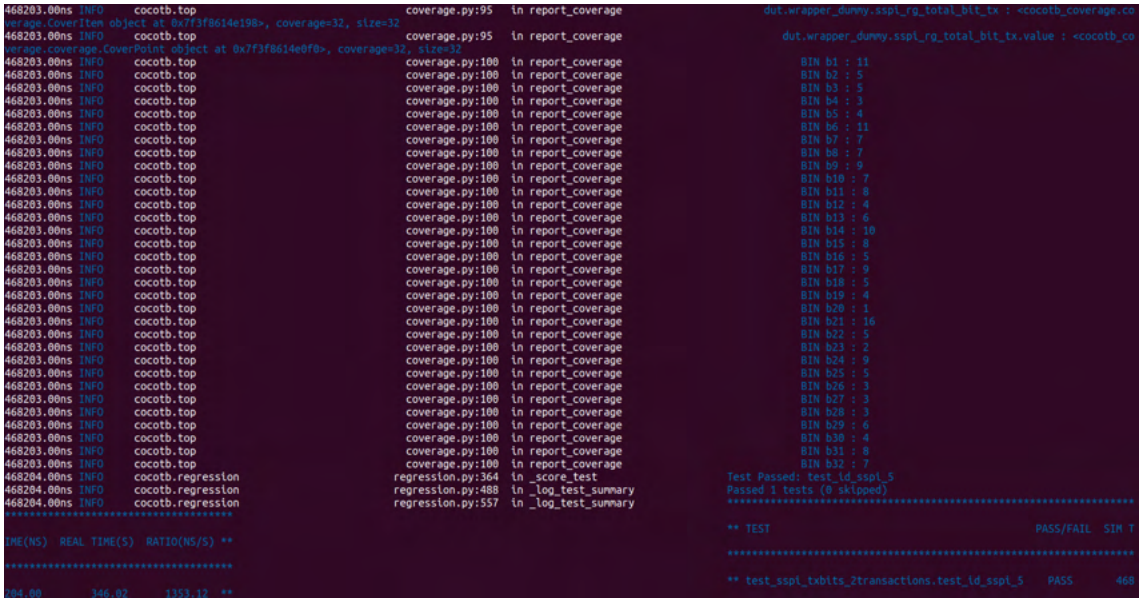

Figure 6.3: Functional coverage for total_bit_tx in two transmit transactions

Three Transmit Transactions

Fig 6.4 shows the functional coverage information for this testcase. The functional coverage is taken for the total bits to be transmitted in three transactions(bits to be transmitted are between 65 and 96).

Functional coverage for total_bit_tx in three transmit transactions = 100 percent

Four Transmit Transactions

Fig 6.5 shows the functional coverage information for this testcase. The functional coverage is taken for the total bits to be transmitted in four transactions(bits to be transmitted are between 97 and 128).

Functional coverage for total_bit_tx in four transmit transactions = 100 percent

Five Transmit Transactions

Fig 6.6 shows the functional coverage information for this testcase. The functional coverage is taken for the total bits to be transmitted in five transactions(bits to be transmitted are between 129 and 160).

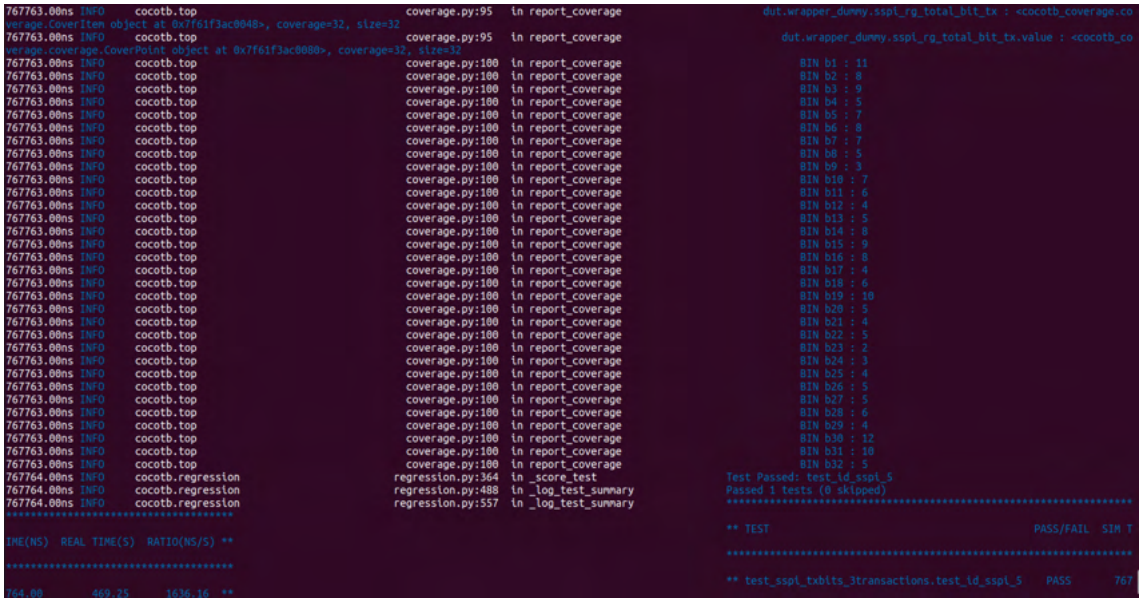

Figure 6.4: Functional coverage for total_bit_tx in three transmit transactions

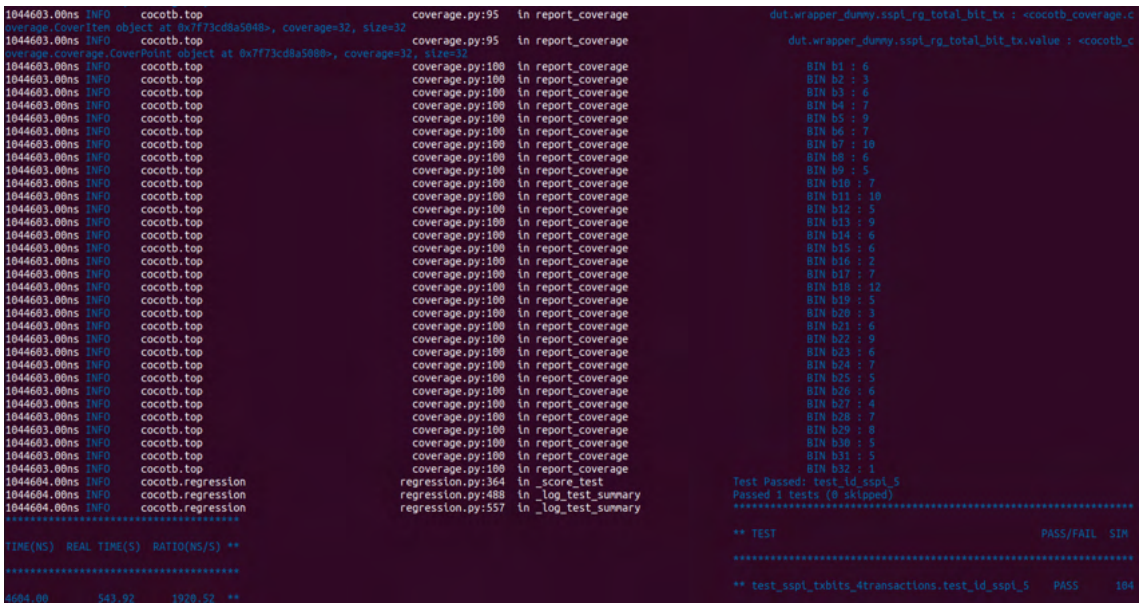

Figure 6.5: Functional coverage for total_bit_tx in four transmit transactions

Functional coverage for total_bit_tx in five transmit transactions = 100 percent

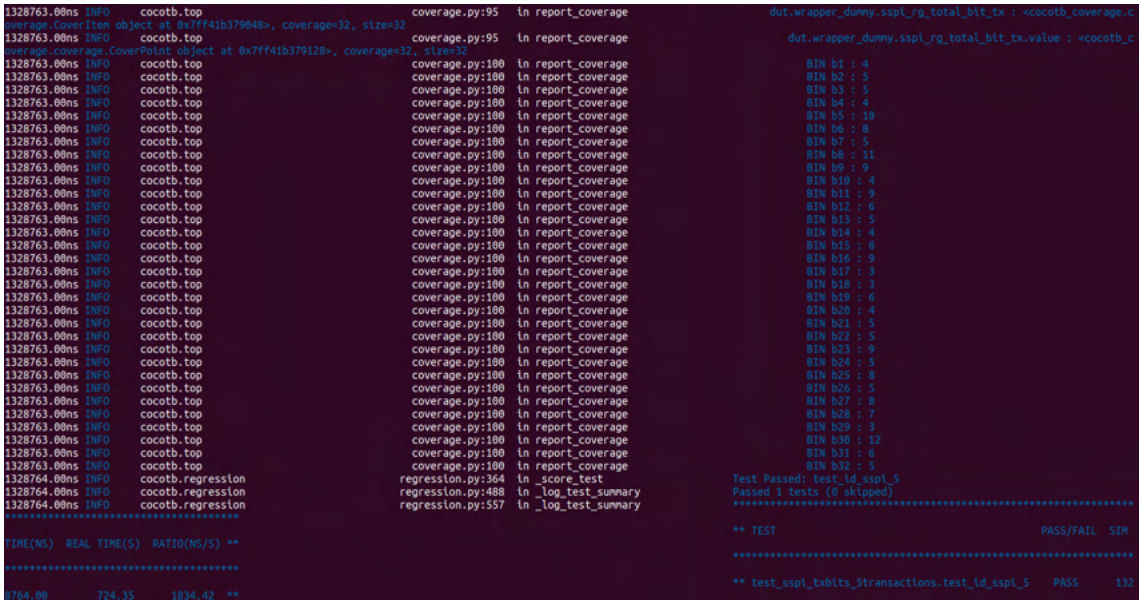

Figure 6.6: Functional coverage for total_bit_tx in five transmit transactions

Six Transmit Transactions

Fig 6.7 shows the functional coverage information for this testcase. The functional coverage is taken for the total bits to be transmitted in six transactions(bits to be transmitted are between 161 and 192).

Functional coverage for total_bit_tx in six transmit transactions = 100 percent

Seven Transmit Transactions

Fig 6.8 shows the functional coverage information for this testcase. The functional coverage is taken for the total bits to be transmitted in seven transactions(bits to be transmitted are between 193 and 224).

Functional coverage for total_bit_tx in seven transmit transactions = 100 percent

Eight Transmit Transactions

Fig 6.9 shows the functional coverage information for this testcase. The functional coverage is taken for the total bits to be transmitted in eight transactions(bits to be transmitted are between 224 and 256).

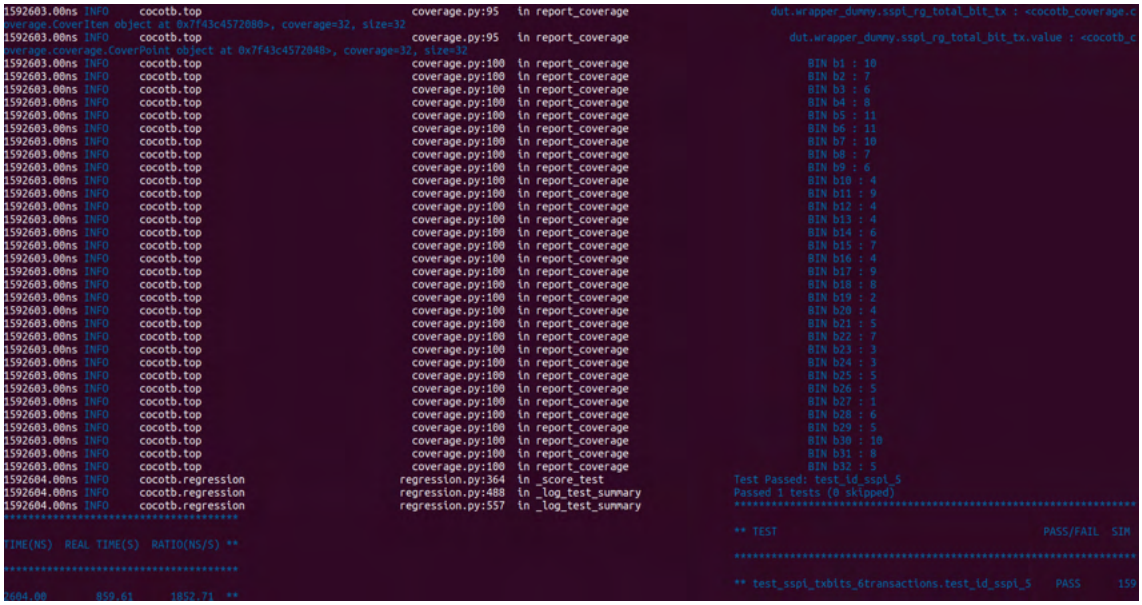

Figure 6.7: Functional coverage for total_bit_tx in six transmit transactions

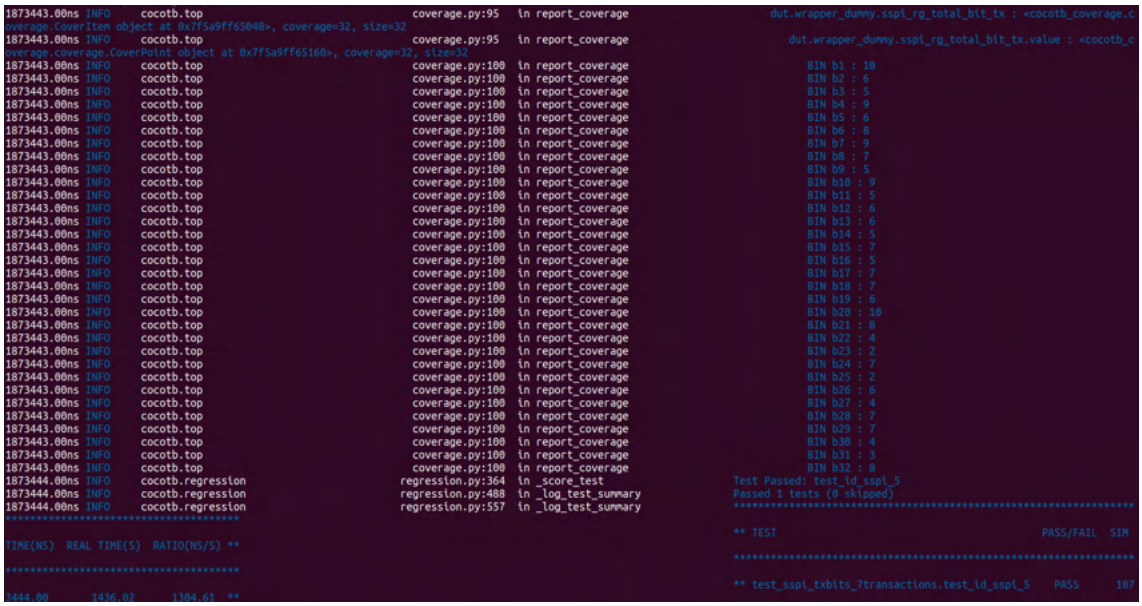

Figure 6.8: Functional coverage for total_bit_tx in seven transmit transactions

Functional coverage for total_bit_tx in eight transmit transactions = 100 percent

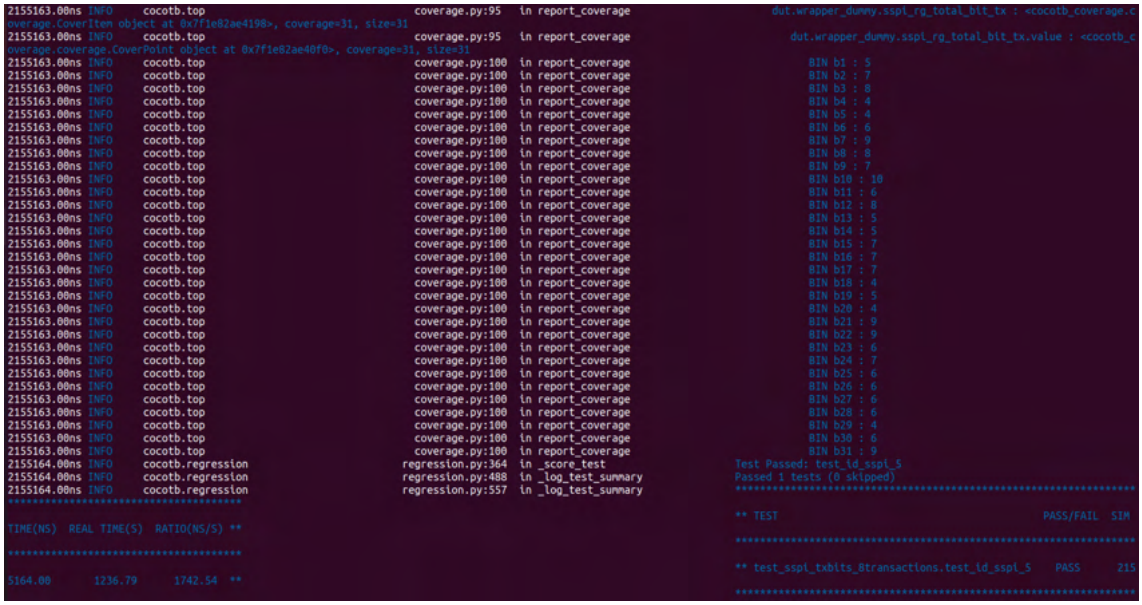

Figure 6.9: Functional coverage for total_bit_tx in eight transmit transactions

6.2 TOTAL_BIT_RX

Feature:

8 bit register storing the total number of bits to be received in a SPI transaction.

Testplan:

Assign a random number to this register and check if the number of bits received is matching the value stored by this register.

Single Receive Transaction

Fig 6.10 shows the simulation of testcase for single receive transaction. As sspi_rg_clk_phase is 0 and sspi_rg_clk_polarity is 1, data to be received is sampled on falling edge of io_sclk_out. Data is sampled from io_miso_in_val line. sspi_rg_total_bit_rx shows the total number of bits to be received and the data received is stored in sspi_rg_rx_data. The data in the sspi_rg_rx_data register is compared with the data being received from io_miso_in_val line manually and then verified.

Fig 6.11 shows the functional coverage information for this testcase. The functional

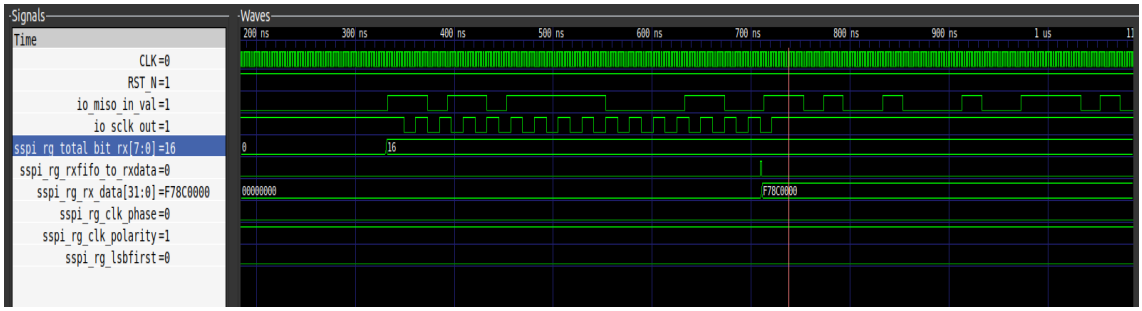

Figure 6.10: Simulation for receive transaction for checking total_bit_rx

coverage is taken for the total bits to be received in a single transaction(bits to be received are between 0 and 32).

Functional coverage for total_bit_rx in single receive transaction = 100 percent

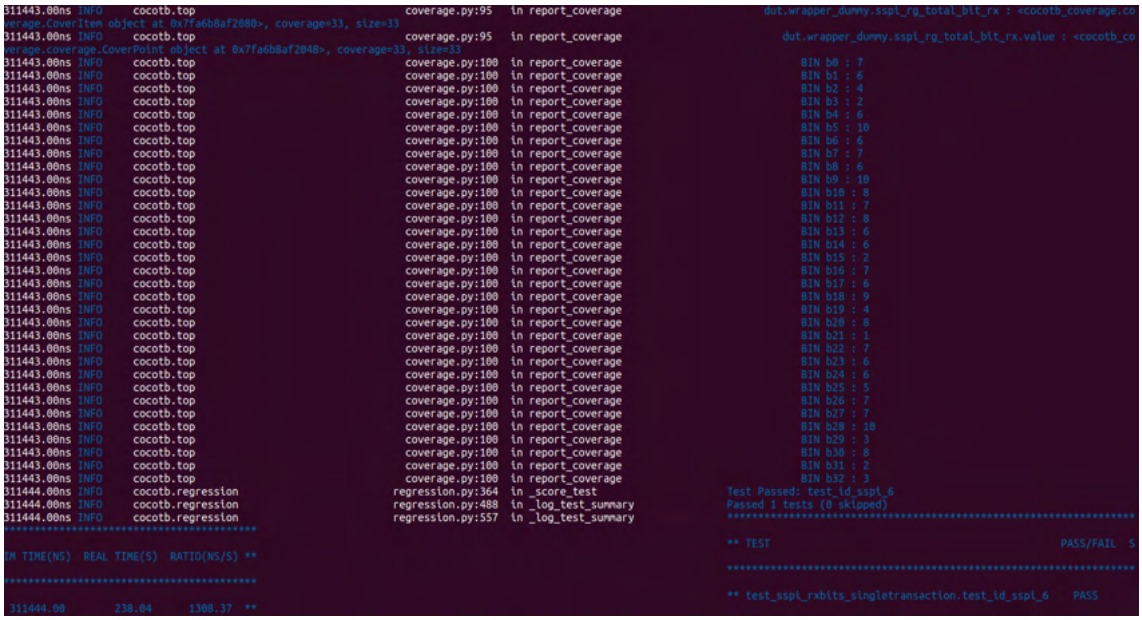

Figure 6.11: Functional coverage for total_bit_rx in single receive transaction

Two Receive Transactions

Fig 6.12 shows the functional coverage information for this testcase. The functional coverage is taken for the total bits to be received in two transactions(bits to be received are between 33 and 64).

Functional coverage for total_bit_rx in two receive transactions = 100 percent

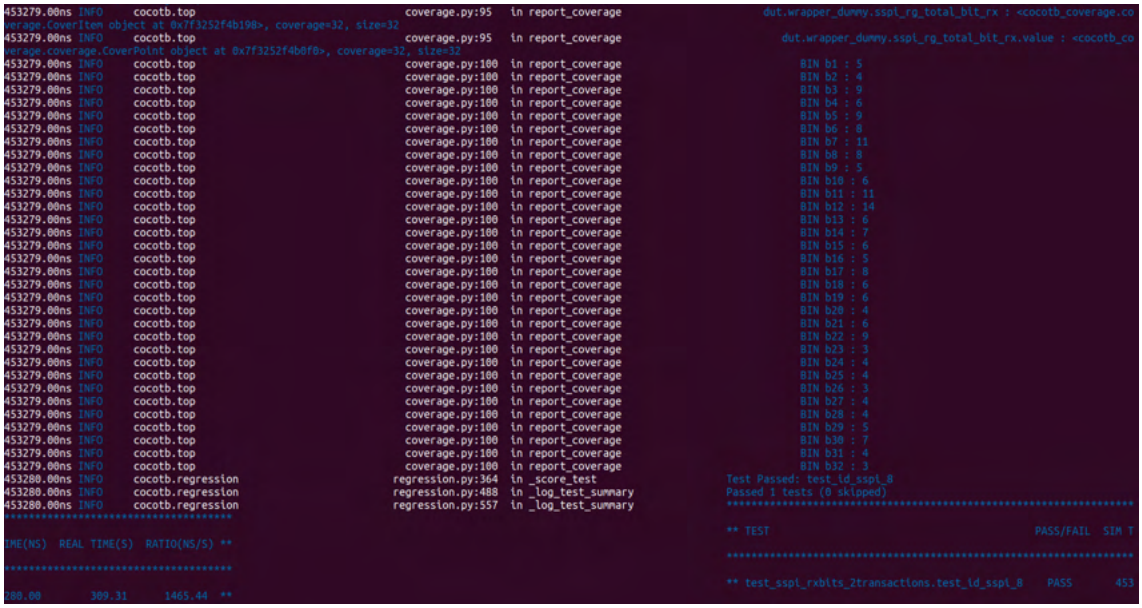

Figure 6.12: Functional coverage for total_bit_rx in two receive transactions

Three Receive Transactions

Fig 6.13 shows the functional coverage information for this testcase. The functional coverage is taken for the total bits to be received in three transactions(bits to be received are between 65 and 96).

Functional coverage for total_bit_rx in three receive transactions = 100 percent

Four Receive Transactions

Fig 6.14 shows the functional coverage information for this testcase. The functional coverage is taken for the total bits to be received in four transactions(bits to be received are between 97 and 128).

Functional coverage for total_bit_rx in four receive transactions = 100 percent

Five Receive Transactions

Fig 6.15 shows the functional coverage information for this testcase. The functional coverage is taken for the total bits to be received in five transactions(bits to be received

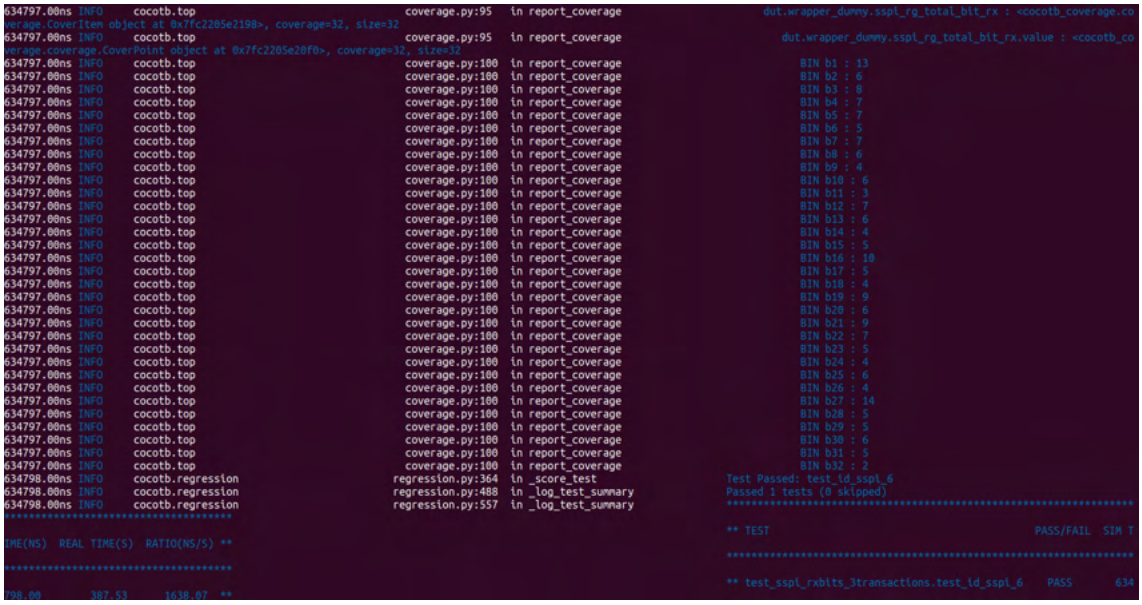

Figure 6.13: Functional coverage for total_bit_rx in three receive transactions

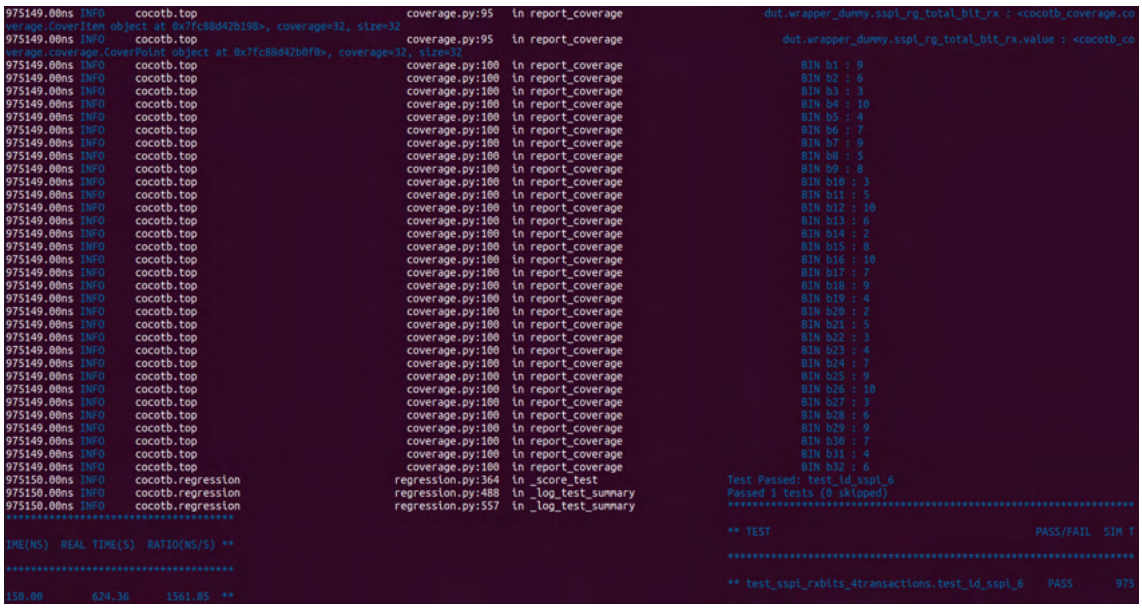

Figure 6.14: Functional coverage for total_bit_rx in four receive transactions

are between 129 and 160).

Functional coverage for total_bit_rx in five receive transactions = 100 percent

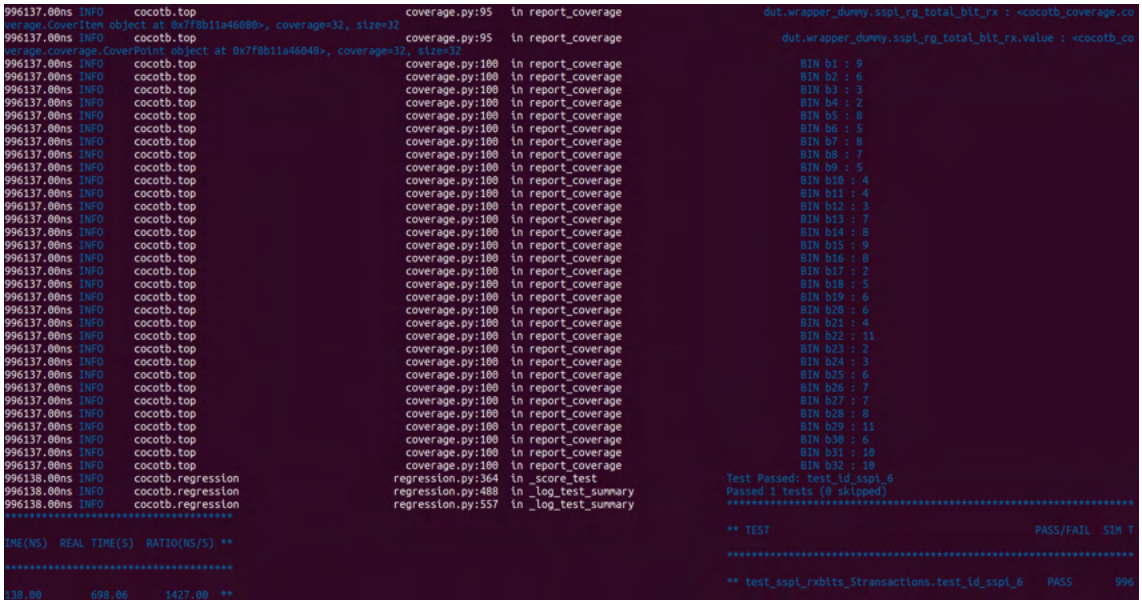

Figure 6.15: Functional coverage for total_bit_rx in five receive transactions

Six Receive Transactions

Fig 6.16 shows the functional coverage information for this testcase. The functional coverage is taken for the total bits to be received in six transactions(bits to be received are between 161 and 192).

Functional coverage for total_bit_rx in six receive transactions = 100 percent

Seven Receive Transactions

Fig 6.17 shows the functional coverage information for this testcase. The functional coverage is taken for the total bits to be received in seven transactions(bits to be received are between 193 and 224).

Functional coverage for total_bit_rx in seven receive transactions = 100 percent

Eight Receive Transactions

Fig 6.18 shows the functional coverage information for this testcase. The functional coverage is taken for the total bits to be received in eight transactions(bits to be received are between 224 and 256).

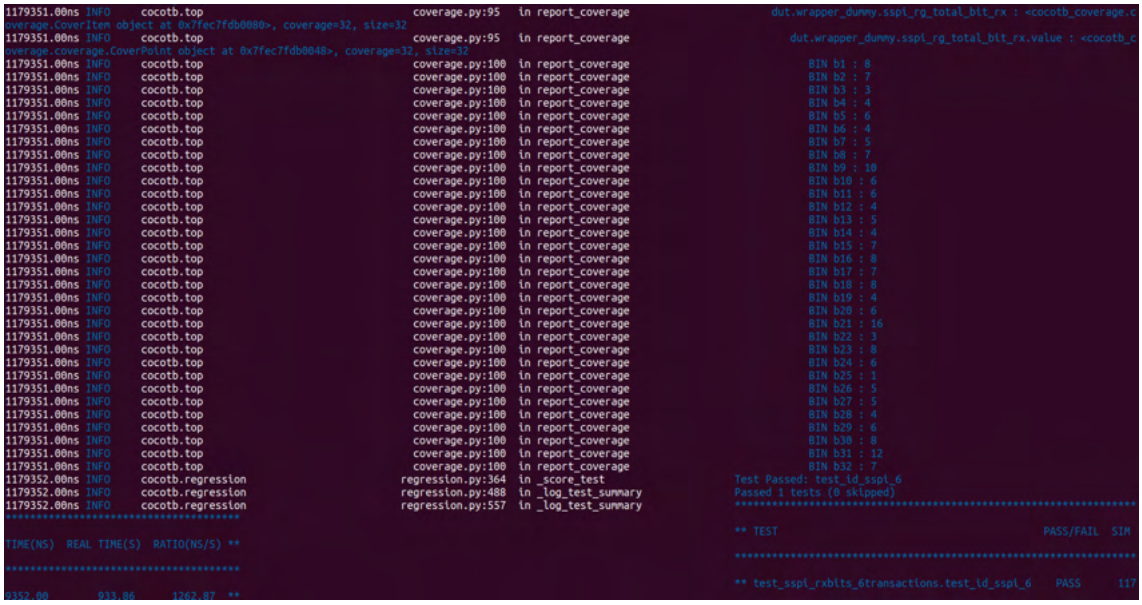

Figure 6.16: Functional coverage for total_bit_rx in six receive transactions

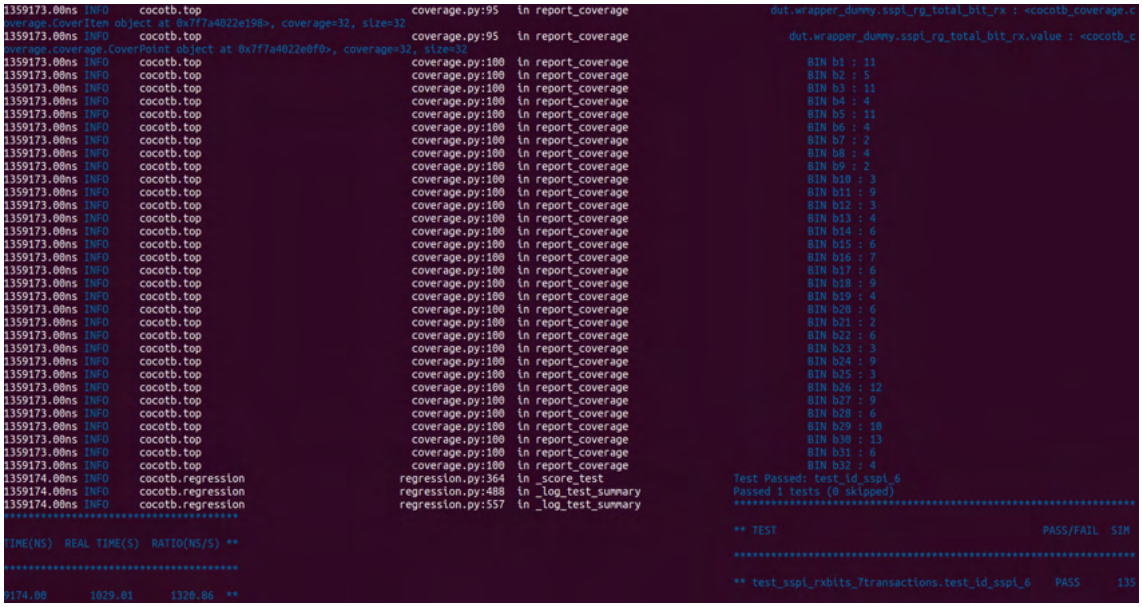

Figure 6.17: Functional coverage for total_bit_rx in seven receive transactions

Functional coverage for total_bit_rx in eight receive transactions = 100 percent

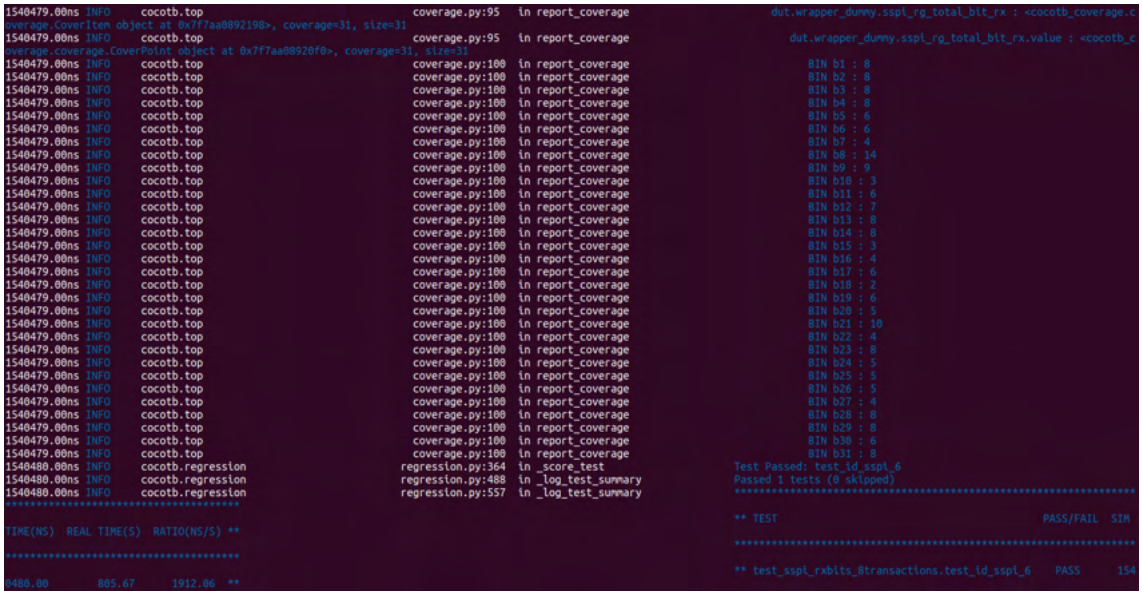

Figure 6.18: Functional coverage for total_bit_rx in eight receive transactions

6.3 LSBFIRST

Feature:

1 bit register storing whether the SPI transaction is LSB first. If set, it means LSB first else it means MSB first.

Testplan:

1. Set this to 1 and check if LSB is first getting transmitted/received.
2. Set this to 0 and check if MSB is first getting transmitted/received.

6.3.1 Transmit

LSB First Transmit

Fig 6.19 shows the simulation of testcase for LSB first transmit transaction. As sspi_rg_lsbfirst is 1, the transmit should start from LSB. A bug is identified here and will be discussed in detail in the next chapter.

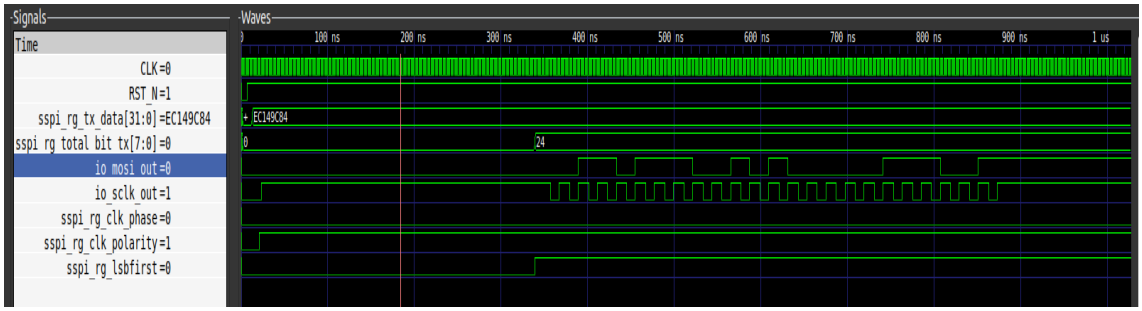

Figure 6.19: Simulation for checking LSB first transmit transaction

MSB First Transmit

Fig 6.20 shows the simulation of testcase for MSB first transmit transaction. As sspi_rg_lsbfirst is 0, the transmit should start from MSB.

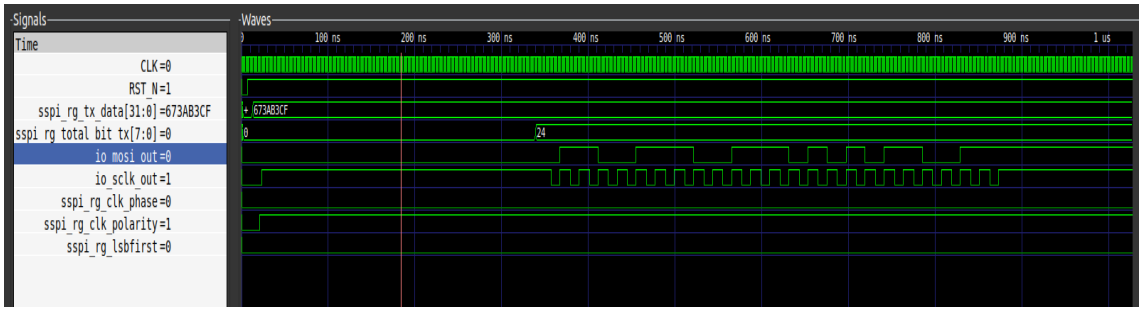

Figure 6.20: Simulation for checking MSB first transmit transaction

6.3.2 Receive

LSB First Receive

Fig 6.21 shows the simulation of testcase for LSB first receive transaction. As sspi_rg_lsbfirst is 1, the bits getting received on MISO line should be taken as LSB first.

MSB First Receive

Fig 6.22 shows the simulation of testcase for MSB first receive transaction. As sspi_rg_lsbfirst is 0, the bits getting received on MISO line should be taken as MSB first.

Figure 6.21: Simulation for checking LSB first receive transaction

Figure 6.22: Simulation for checking MSB first receive transaction

6.4 SCLK CONFIGURATION

6.4.1 Transmit

CPHA=0 and CPOL=0

Fig 6.23 shows the simulation of testcase for SCLK configuration in transmit mode with CPHA=0 and CPOL=0. As sspi_rg_clk_phase is 0 and sspi_rg_clk_polarity is 0, in idle state SCLK is 0 and the bits getting are getting transmitted on every falling edge of SCLK.

CPHA=0 and CPOL=1

Fig 6.24 shows the simulation of testcase for SCLK configuration in transmit mode with CPHA=0 and CPOL=1. As sspi_rg_clk_phase is 0 and sspi_rg_clk_polarity is 1, in idle state SCLK is 1 and the bits getting are getting transmitted on every rising edge of SCLK.

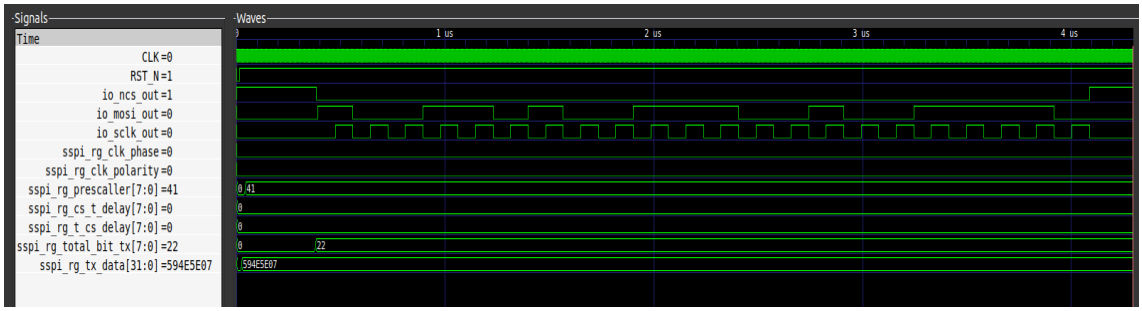

Figure 6.23: SCLK configuration in transmit mode with CPHA=0 and CPOL=0

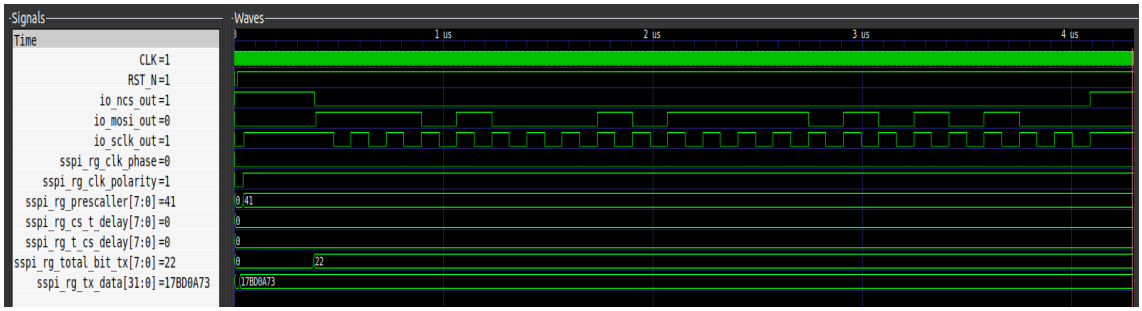

Figure 6.24: SCLK configuration in transmit mode with CPHA=0 and CPOL=1

CPHA=1 and CPOL=0

Fig 6.25 shows the simulation of testcase for SCLK configuration in transmit mode with CPHA=1 and CPOL=0. As sspi_rg_clk_phase is 1 and sspi_rg_clk_polarity is 0, in idle state SCLK is 0 and the bits getting are getting transmitted on every rising edge of SCLK.

CPHA=1 and CPOL=1

Fig 6.26 shows the simulation of testcase for SCLK configuration in transmit mode with CPHA=1 and CPOL=1. As sspi_rg_clk_phase is 1 and sspi_rg_clk_polarity is 1, in idle state SCLK is 1 and the bits getting are getting transmitted on every falling edge of SCLK.

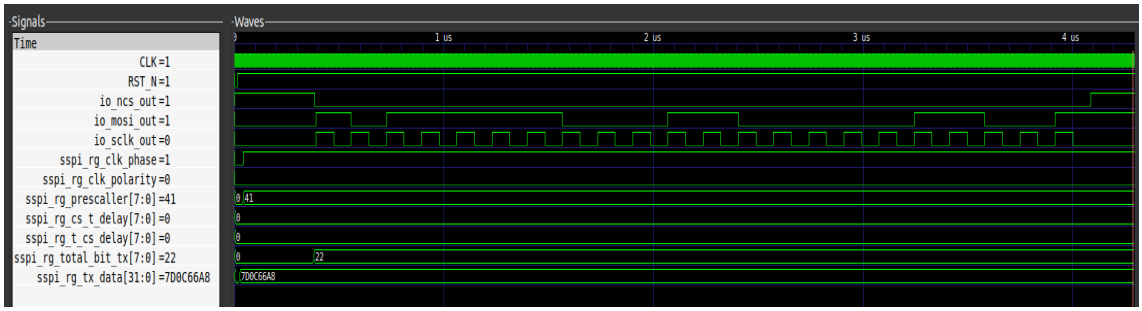

Figure 6.25: SCLK configuration in transmit mode with CPHA=1 and CPOL=0

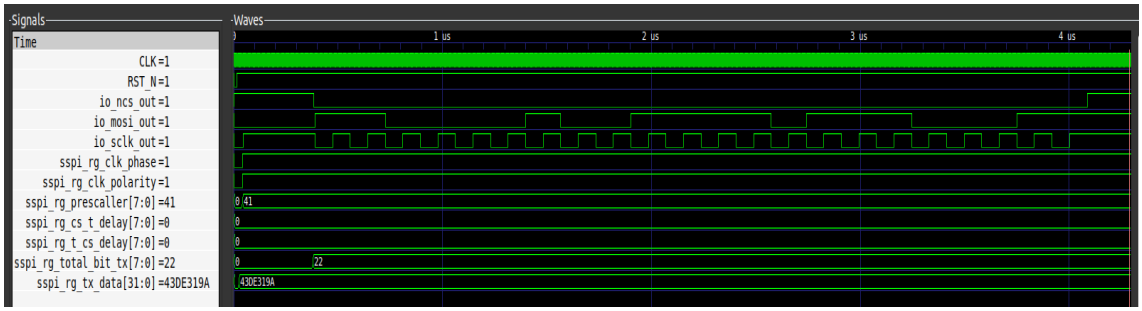

Figure 6.26: SCLK configuration in transmit mode with CPHA=1 and CPOL=1

6.4.2 Receive

CPHA=0 and CPOL=0

Fig 6.27 shows the simulation of testcase for SCLK configuration in receive mode with CPHA=0 and CPOL=0. As sspi_rg_clk_phase is 0 and sspi_rg_clk_polarity is 0, in idle state SCLK is 0 and the bits to be received is getting sampled on every rising edge of SCLK.

CPHA=0 and CPOL=1

Fig 6.28 shows the simulation of testcase for SCLK configuration in receive mode with CPHA=0 and CPOL=1. As sspi_rg_clk_phase is 0 and sspi_rg_clk_polarity is 1, in idle state SCLK is 1 and the bits to be received is getting sampled on every falling edge of SCLK.

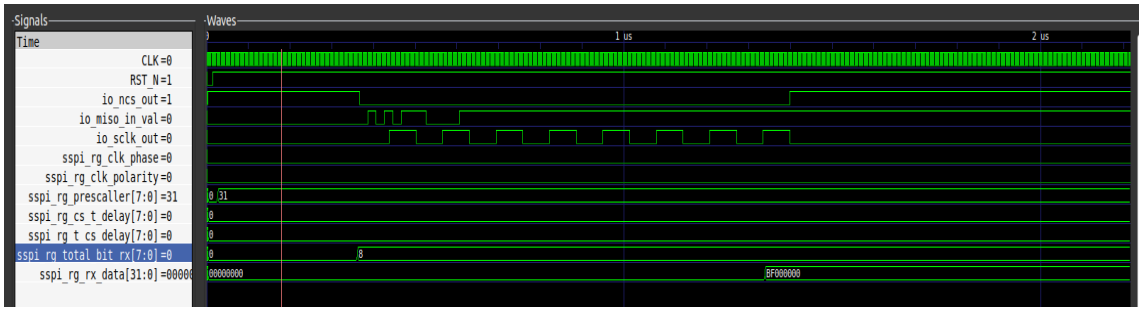

Figure 6.27: SCLK configuration in receive mode with CPHA=0 and CPOL=0

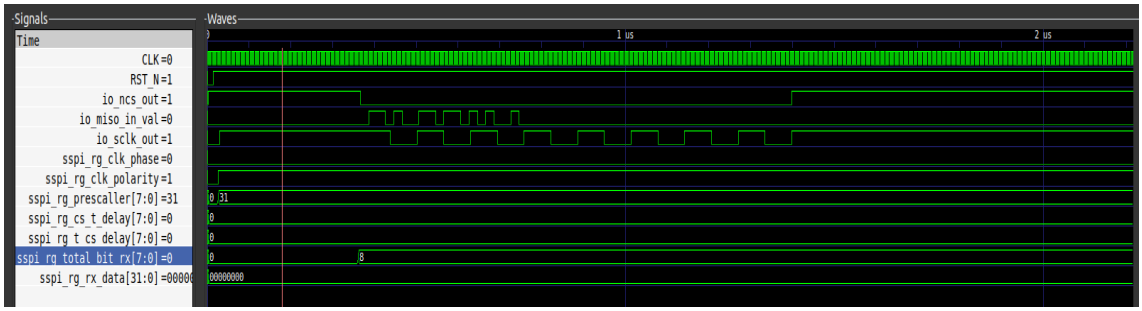

Figure 6.28: SCLK configuration in receive mode with CPHA=0 and CPOL=1

CPHA=1 and CPOL=0

Fig 6.29 shows the simulation of testcase for SCLK configuration in receive mode with CPHA=1 and CPOL=0. As sspi_rg_clk_phase is 1 and sspi_rg_clk_polarity is 0, in idle state SCLK is 0 and the bits to be received is getting sampled on every falling edge of SCLK. A bug is identified here in SCLK generation and will be discussed in detail in the next chapter.

CPHA=1 and CPOL=1

Fig 6.30 shows the simulation of testcase for SCLK configuration in receive mode with CPHA=1 and CPOL=1. As sspi_rg_clk_phase is 1 and sspi_rg_clk_polarity is 1, in idle state SCLK is 1 and the bits to be received is getting sampled on every rising edge of SCLK. A bug is identified here in SCLK generation and will be discussed in detail in the next chapter.

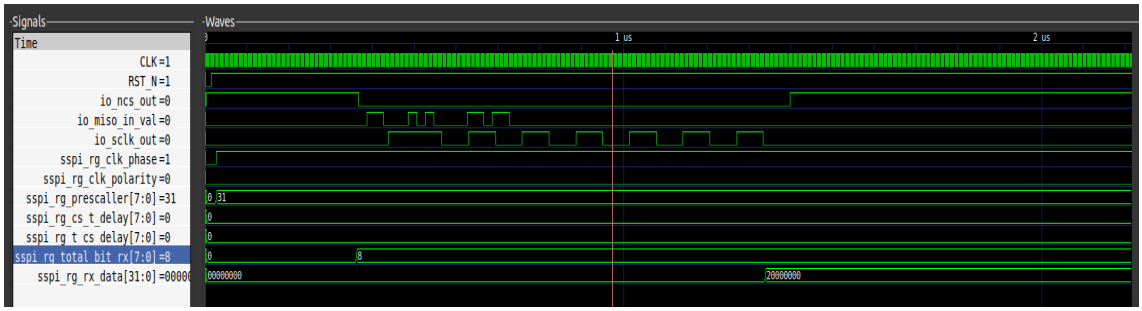

Figure 6.29: SCLK configuration in receive mode with CPHA=1 and CPOL=0

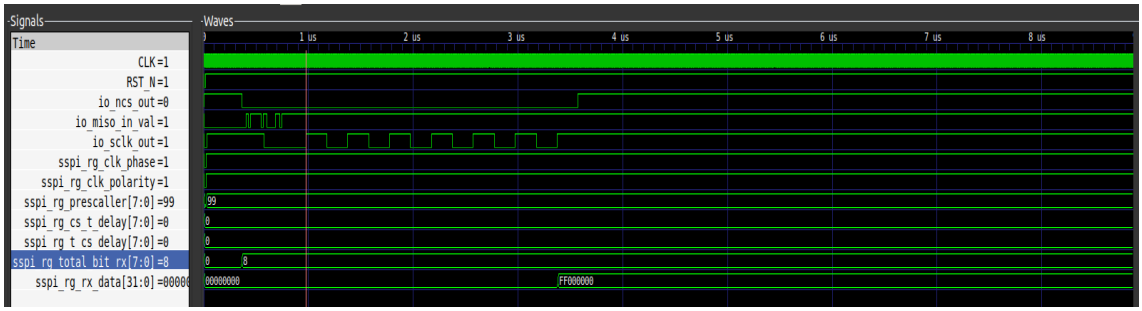

Figure 6.30: SCLK configuration in receive mode with CPHA=1 and CPOL=1

6.5 SETUP DELAY

Setup delay is nothing but the delay between NCS going low and the transmit or receive transaction starting. It is the sspi_rg_cs_t_delay register configured using clock control register.

Fig 6.31 and 6.32 shows the simulation of testcase for setup delay in transmit mode. Fig 6.31 shows the time when NCS is going low and fig 6.32 shows the time when transmit is starting. A bug is identified here and will be discussed in detail in the next chapter.

Fig 6.33 shows the functional coverage information for the above testcase. The functional coverage is taken for sspi_rg_cs_t_delay register in 32 bins. Each bin has 32 values.

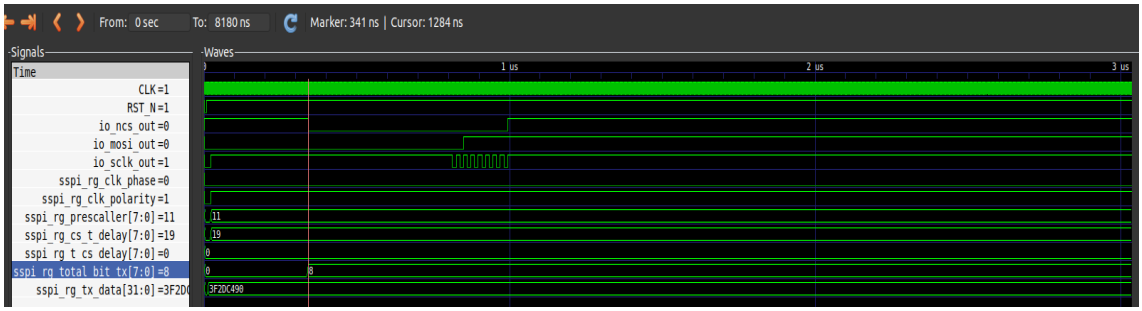

Figure 6.31: Simulation for setup delay indicating time when NCS is going low in transmit mode.

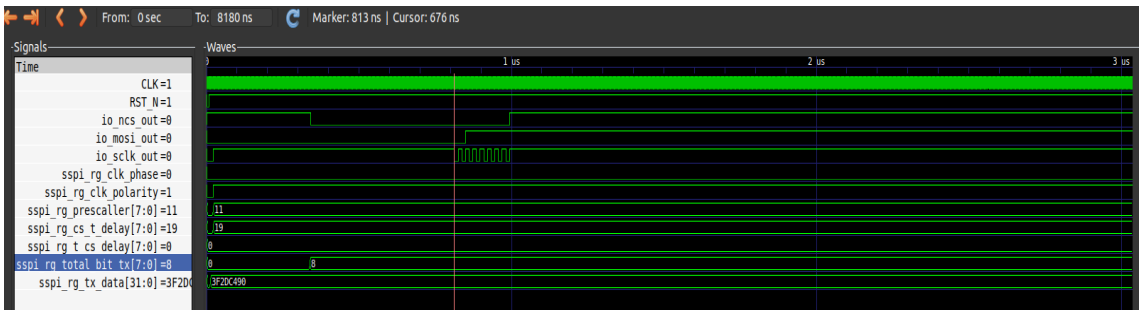

Figure 6.32: Simulation for setup delay indicating time when transmit is starting

6.6 HOLD DELAY

Hold delay is nothing but the delay between the transmit or receive transaction ending and the NCS going high. It is the sspi_rg_t_cs_delay register configured using clock control register.

Fig 6.34 and 6.35 shows the simulation of testcase for hold delay in transmit mode. Fig 6.34 shows the time when transmit is ending and fig 6.35 shows the time when NCS is going high.

From the figures we see that transmit is ending at 1610ns and NCS is going high at 1896ns. The sspi_rg_t_cs_delay is 13 and sspi_rg_prescaller is 10. The CLK time period is 2ns.

Expected delay = (Prescaller+1)*Hold*CLK period = (10+1)*13*2 = 286ns

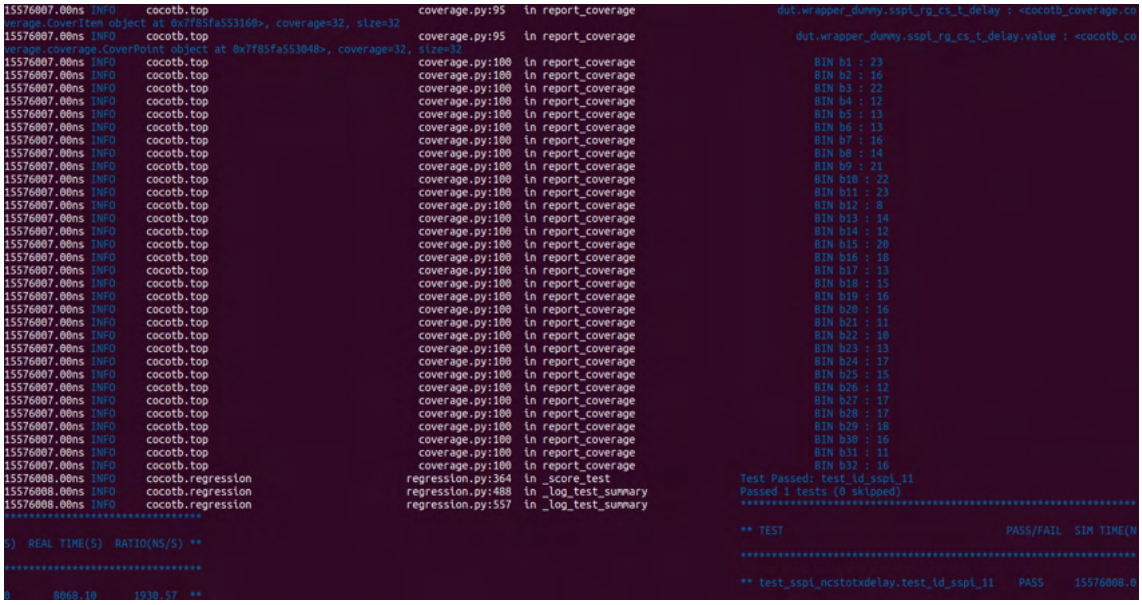

Figure 6.33: Functional coverage for sspi_rg_cs_t_delay in transmit mode.

Figure 6.34: Simulation for hold delay indicating time when transmit is ending.

Obtained delay = 1896-1610 = 286ns

Fig 6.36 shows the functional coverage information for the above testcase. The functional coverage is taken for sspi_rg_t_cs_delay in in transmit mode register in 32 bins. Each bin has 32 values.

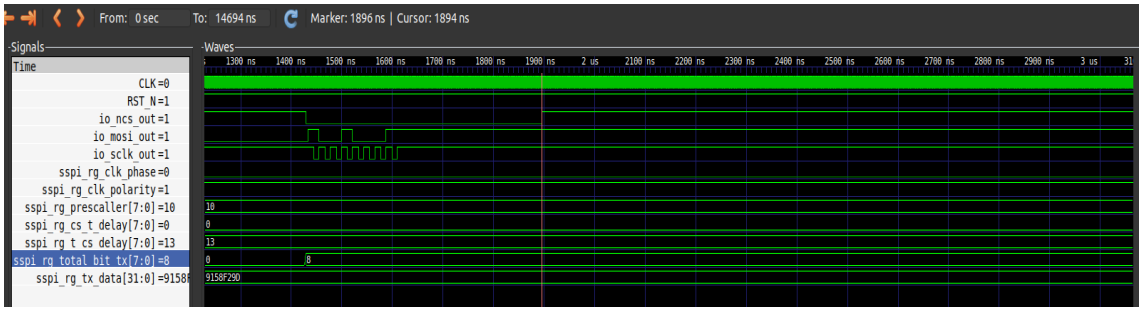

Figure 6.35: Simulation for setup delay indicating time when NCS is going high in transmit mode.

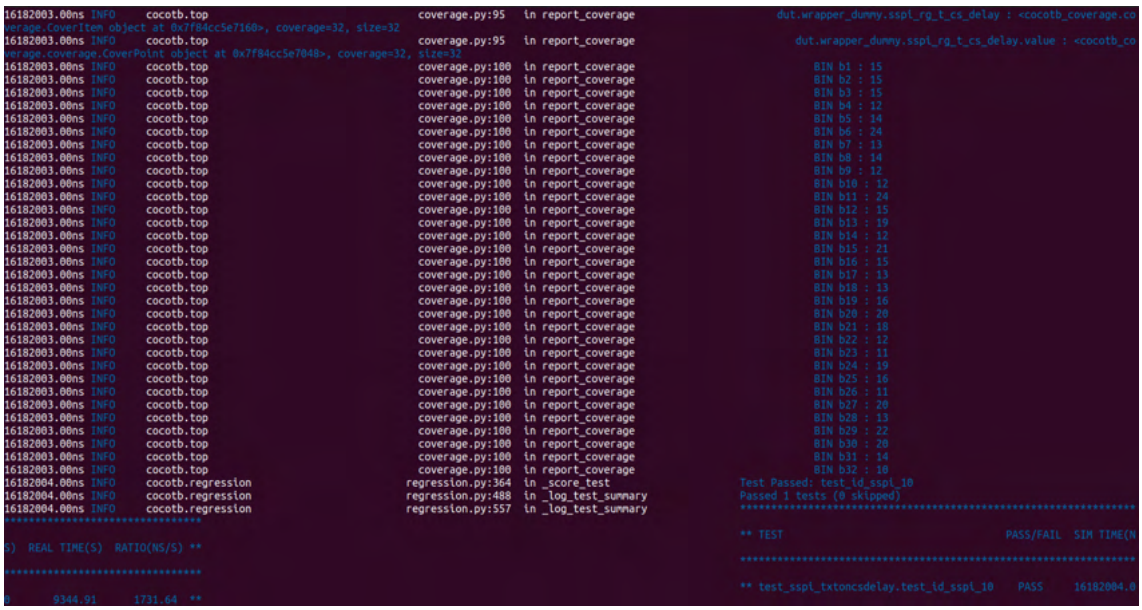

Figure 6.36: Functional coverage for sspi_rg_t_cs_delay in transmit mode.

6.7 REQUIRED BIT RATE GENERATION - PRESCALLER

Internal clock frequency is divided by the prescaller value to attain the required bit rate. It is the sspi_rg_prescaller register configured using clock control register. It is a 8-bit value and hence can provide maximum of 255 different bit rates.

Required Bit Rate = Internal clock frequency/(Prescaler value+1)

So as per this,

Time period of SCLK = Time period of internal clock * (Prescaler value+1)

6.7.1 Transmit

Fig 6.37 and 6.38 shows the simulation of testcase for prescaller value in transmit mode.

Fig 6.37 shows the one edge of SCLK and fig 6.38 shows the other edge of SCLK.

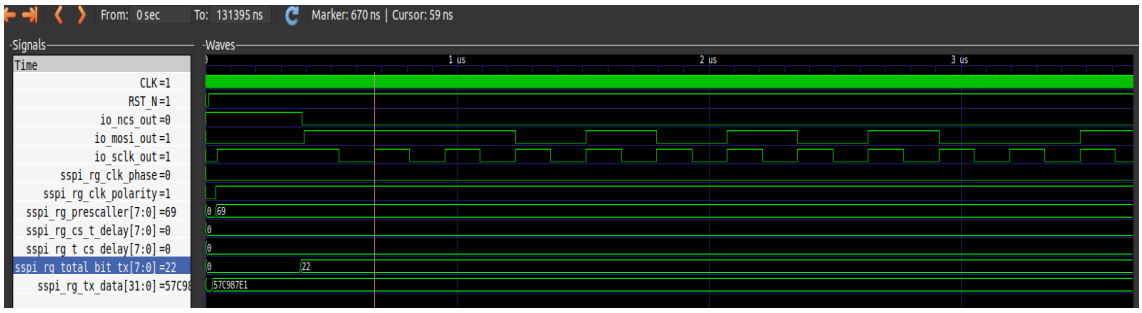

Figure 6.37: Simulation showing one edge of SCLK for required bit rate generation in transmit mode.

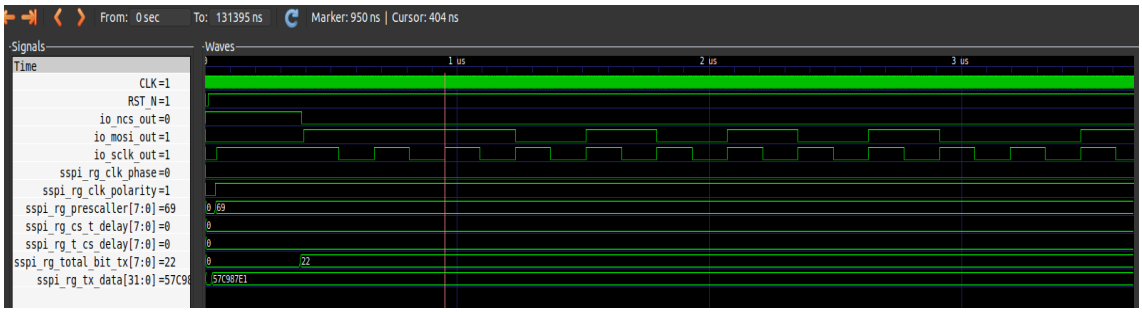

Figure 6.38: Simulation showing other edge of SCLK for required bit rate generation in transmit mode.

From the figures we see that one edge is at 670ns and other edge is at 950ns. The sspi_rg_prescaller is 69 and internal clock time period is 4ns.

Expected time period = Time period of internal clock * (Prescaler value+1) = (69+1)*4 = 280ns

Obtained delay = 950-670 = 280ns

Fig 6.39 shows the functional coverage information for the above testcase. The functional coverage is taken for sspi_rg_prescaller register in transmit mode in 32 bins. Each bin has 32 values.

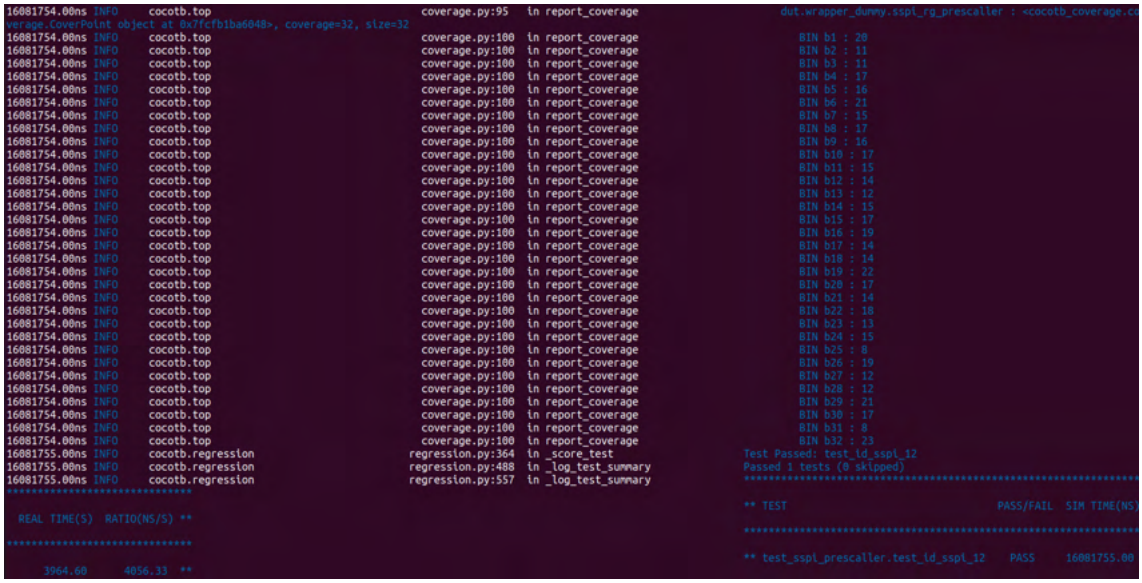

Figure 6.39: Functional coverage for sspi_rg_prescaller in transmit mode.

6.7.2 Receive

Fig 6.40 and 6.41 shows the simulation of testcase for prescaller value in receive mode.

Fig 6.40 shows the one edge of SCLK and fig 6.41 shows the other edge of SCLK.

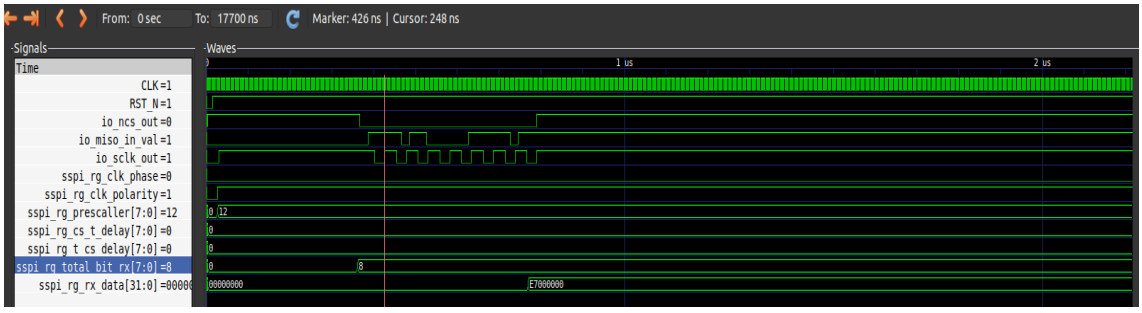

Figure 6.40: Simulation showing one edge of SCLK for required bit rate generation in receive mode.

From the figures we see that one edge is at 426ns and other edge is at 478ns. The sspi_rg_prescaller is 12 and internal clock time period is 4ns.

Expected time period = Internal clock period * (Prescaler value+1) = (12+1)*4 = 52ns

Obtained delay = 478-426 = 52ns

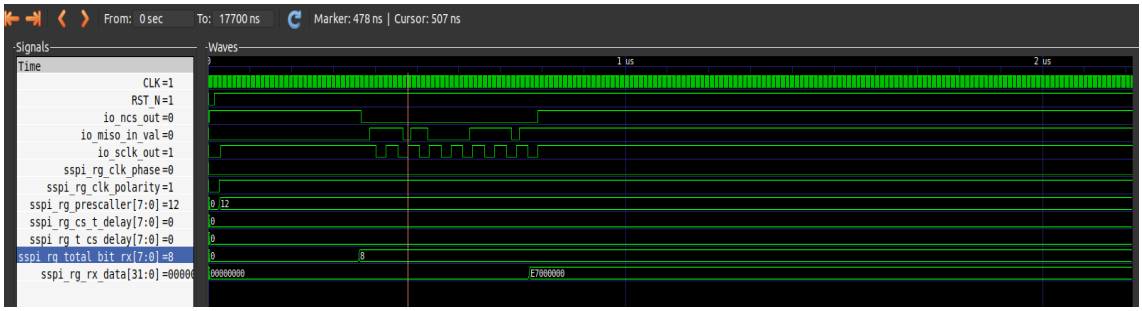

Figure 6.41: Simulation showing other edge of SCLK for required bit rate generation in receive mode.

Fig 6.42 shows the functional coverage information for the above testcase. The functional coverage is taken for sspi_rg_prescaller register in transmit mode in 32 bins. Each bin has 32 values.

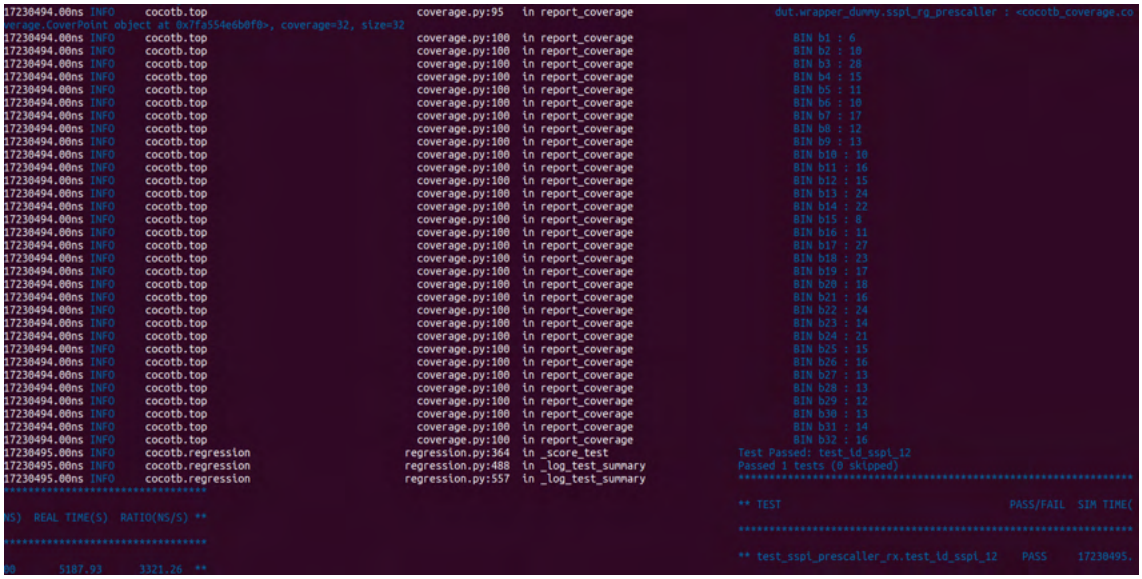

Figure 6.42: Functional coverage for sspi_rg_prescaller in receive mode.

6.8 CODE COVERAGE

This project has achieved a code coverage of 92.8 percent as shown in fig 6.43.

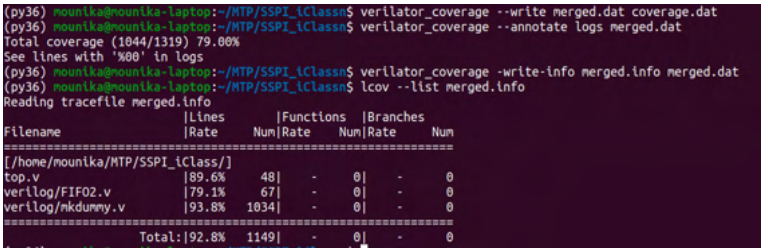

Figure 6.43: Code Coverage acheived.

CHAPTER 7

DESIGN MISMATCHES, BUGS AND CHALLENGES

In this chapter we will discuss the design mismatches found, bugs in the design which were found out during the verification and the challenges faced during verification.

7.1 DESIGN MISMATCHES

7.1.1 LSB First Transmit Mode Mismatch

Fig 7.1 shows the simulation of testcase for LSB first transmit transaction. As sspi_rg_lsbfirst is 1, the transmit should start from LSB.

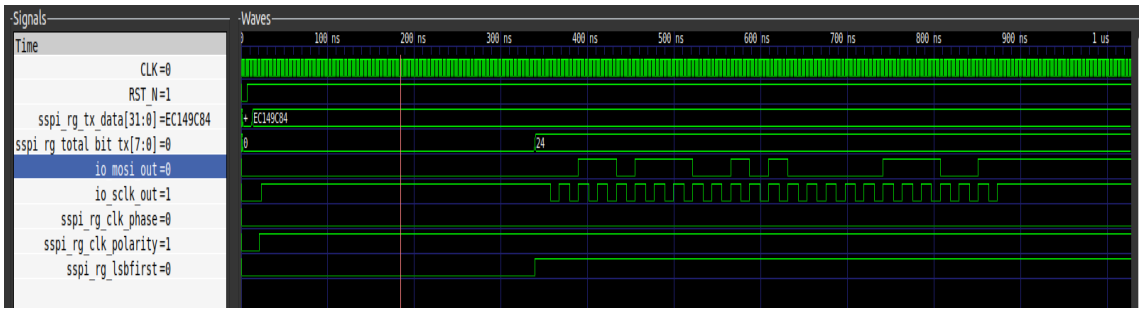

Figure 7.1: Simulation for checking LSB first transmit transaction

sspi_rg_total_bit_tx is 24 which means we are transmitting 24 bits. The transmit data in sspi_rg_txdata is EC149C84 (in Hex) and 1110 1100 0001 0100 1001 1100 1000 0100 (in Binary). So, the expected order of transmission on the io_mosi_out line is 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0 i.e, from bit[0] to bit[23].

Since, sspi_rg_clk_phase is 0 and sspi_rg_clk_polarity is 1, data is getting transmitted for every rising edge of io_sclk_out. From the waveform if we see, the order in which bits are getting transmitted is 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1. Expected and obtained are not matching. Data is getting transmitted byte by byte and

this is the mismatch. Data is getting transmitted as bit[24] to bit[31] first, then bit[16] to bit[23] next and then bit[8] to bit[15].

7.1.2 Setup Delay Mismatch

Setup delay is nothing but the delay between NCS going low and the transmit or receive transaction starting. It is the sspi_rg_cs_t_delay register configured using clock control register.

Transmit Mode

Fig 7.2 and 7.3 shows the simulation of testcase for setup delay in transmit mode. Fig 7.2 shows the time when NCS is going low and fig 7.3 shows the time when transmit is starting.

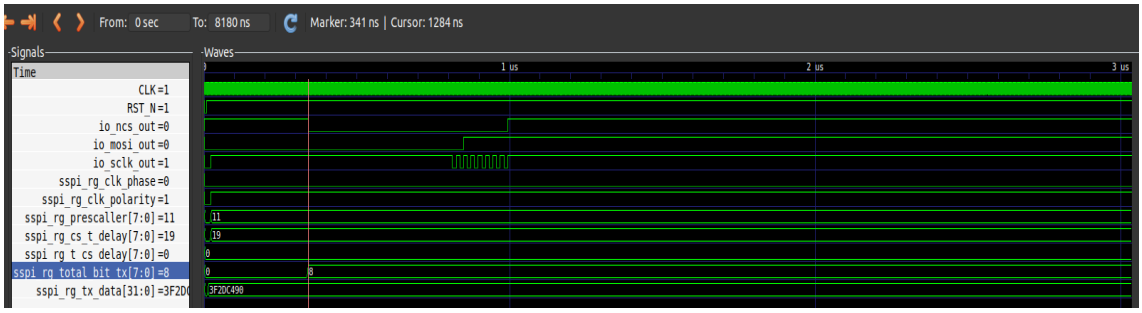

Figure 7.2: Simulation for setup delay indicating time when NCS is going low in transmit mode.

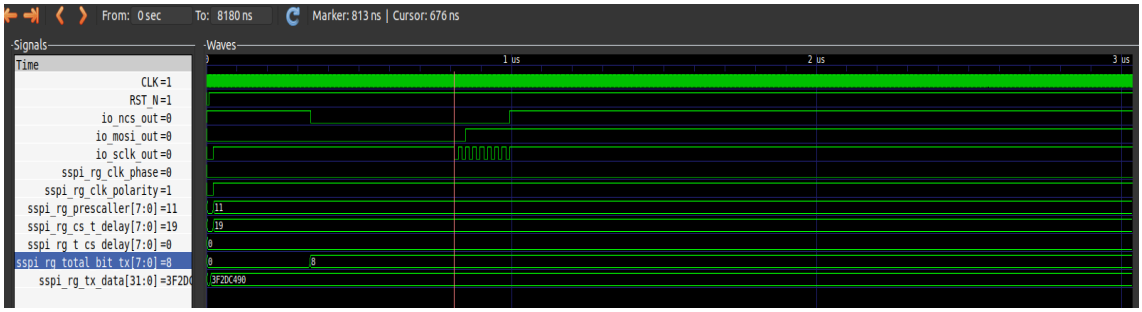

Figure 7.3: Simulation for setup delay indicating time when transmit is starting

From the figures we see that NCS is going low at 341ns and transmit is starting at 813ns.

Table 7.1: Different cases of extra delay in transmit mode

Prescaller	**Setup**	**Expected Delay**	**Obtained Delay**	**Difference**
1	148	931-333=598	148*2*2=592	6
2	65	731-333=398	65*3*2=390	8
3	185	1821-333=1488	185*4*2=1480	8
4	24	583-333=250	24*5*2=240	10
5	235	3163-333=2830	235*6*2=2820	10
6	115	1955-333=1622	115*7*2=1610	12
7	197	3497-333=3164	197*8*2=3152	12
8	61	1445-333=1112	61*9*2=1098	14
9	146	3267-333=2934	146*10*2=2920	14
10	35	1119-333=786	35*11*2=770	16
11	85	2389-333=2056	85*12*2=2040	16
12	48	1599-333=1266	48*13*2=1248	18
13	105	3291-333=2958	105*14*2=2940	18
14	172	5513-333=5180	172*15*2=5160	20
15	246	8225-333=7892	246*16*2=7872	20

The sspi_rg_cs_t_delay is 19 and sspi_rg_prescaller is 11. The CLK time period is 2ns.

Expected delay = (Prescaller+1)*Hold*CLK period = (11+1)*19*2 = 456ns

Obtained delay = 1896-1610 = 472ns

Expected and obtained delays are not matching. There is an extra delay.Few more cases are given in the table 7.1 .

The setup delay is applicable only for transmit data and is applicable only in master mode. The slave will see only the clock mode and edges as it is not aware of set up time etc. There may be delays based on system clock period, AXI latencies when SPI is enabled. Until it affects the transmission and reception from other side, we can ignore the additional delay. For all the cases in table 7.1, the additional delay is not affecting the transmission and hence can be ignored.

7.2 BUGS IDENTIFIED

7.2.1 SCLK Edge Bug

CPHA=0 and CPOL=1

Transmit

Fig 7.4 shows the simulation of testcase for transmit only mode with SCLK configuration as CPHA=0 and CPOL=1.

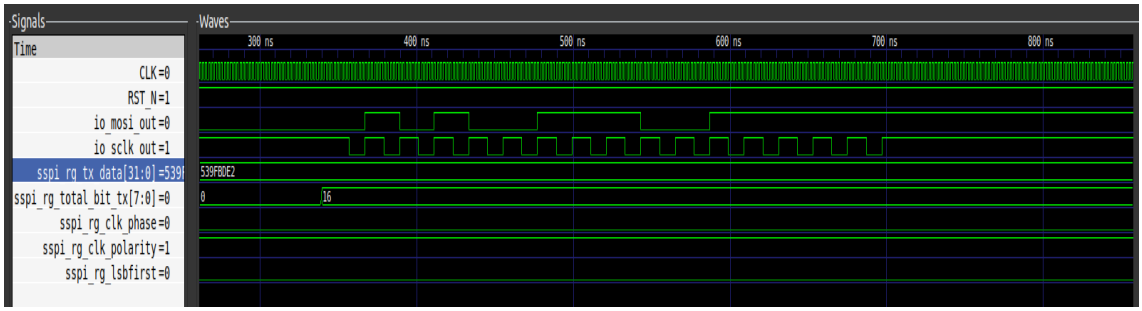

Figure 7.4: Transmit only mode simulation with CPHA=0 and CPOL=1

As CPHA=0 and CPOL=1, in idle state SCLK should be high and data should be transmitted on every falling edge of SCLK. In the simulation if we see, data is getting transmitted on every rising edge of SCLK. This is the bug. It is raised and fixed.

Receive

Fig 7.5 shows the simulation of testcase for receive only mode with SCLK configuration as CPHA=0 and CPOL=1.

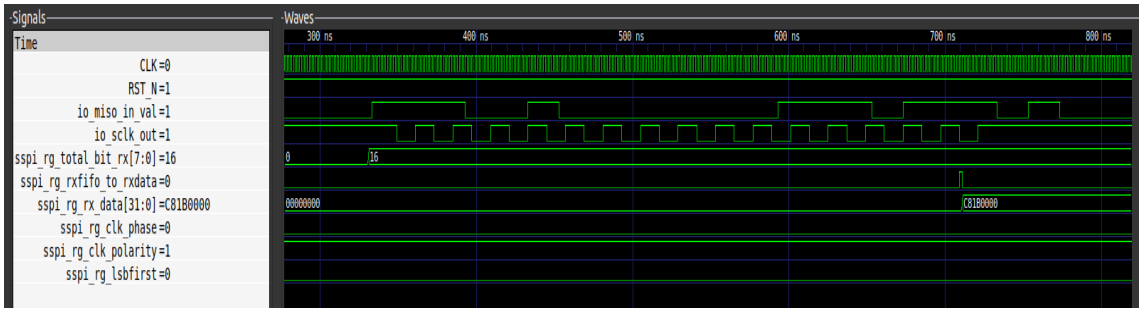

Figure 7.5: Receive only mode simulation with CPHA=0 and CPOL=1

As CPHA=0 and CPOL=1, in idle state SCLK should be high and data to be received should be sampled on every rising edge of SCLK. If the data is sampled on every rising edge of SCLK, the rx_data should be E01F0000 and if the data is sampled on every falling edge of SCLK, the rx_data should be C81B0000. In the simulation we can see that the sspi_rx_data has C81B0000 which means that the data got sampled on every falling edge. This is the bug. It is raised and fixed.

CPHA=1 and CPOL=1

Transmit

Fig 7.6 shows the simulation of testcase for transmit only mode with SCLK configuration as CPHA=1 and CPOL=1.

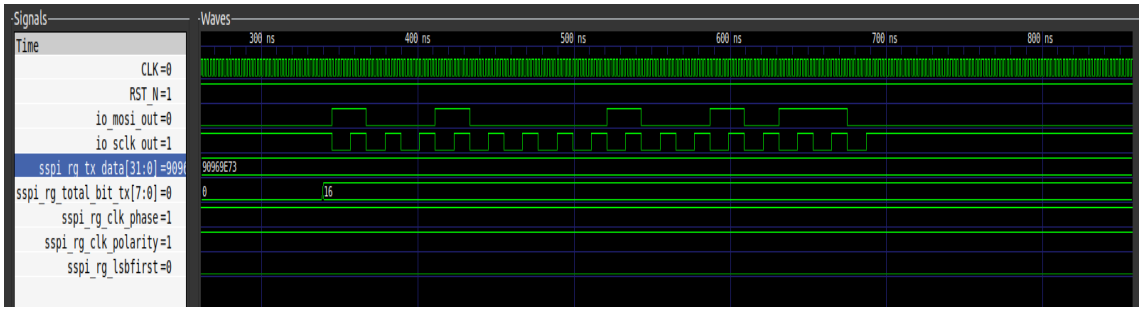

Figure 7.6: Transmit only mode simulation with CPHA=1 and CPOL=1

As CPHA=1 and CPOL=1, in idle state SCLK should be high and data should be transmitted on every rising edge of SCLK. In the simulation if we see, data is getting transmitted on every falling edge of SCLK. This is the bug. It is raised and fixed.

Receive

Fig 7.7 shows the simulation of testcase for receive only mode with SCLK configuration as CPHA=1 and CPOL=1.

As CPHA=1 and CPOL=1, in idle state SCLK should be high and data to be received should be sampled on every falling edge of SCLK. If the data is sampled on every falling edge of SCLK, the rx_data should be DB990000 and if the data is sampled on every

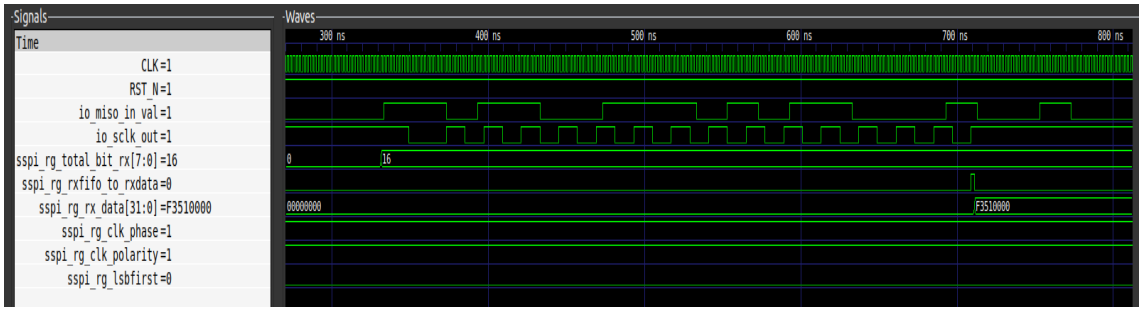

Figure 7.7: Receive only mode simulation with CPHA=1 and CPOL=1

rising edge of SCLK, the rx_data should be F3510000. In the simulation we can see that the sspi_rx_data has F3510000 which means that the data got sampled on every rising edge. This is the bug. It is raised and fixed.

7.2.2 Receive Mode SCLK Generation Bug

CPHA=1 and CPOL=0

Fig 7.8 shows the simulation of testcase for SCLK configuration in receive mode with CPHA=1 and CPOL=0.

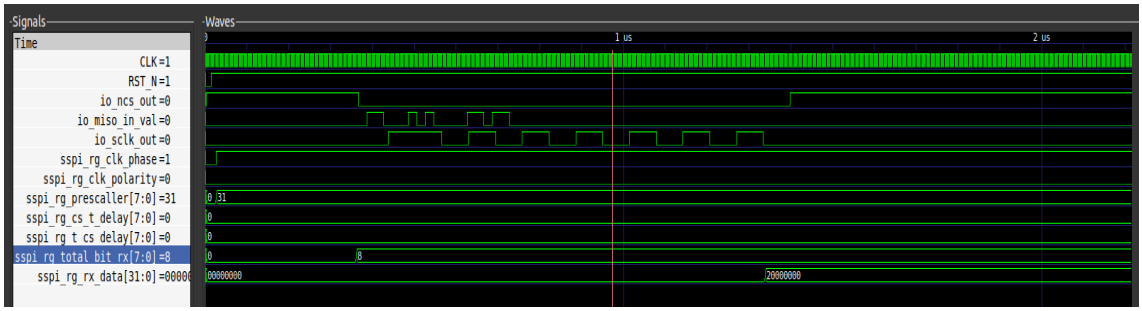

Figure 7.8: SCLK configuration in receive mode with CPHA=1 and CPOL=0

In the waveform we can see that the first pulse of SCLK is not of 50 percent duty cycle. To understand it in much details, let us see two cases, one is odd prescaller value and other is even prescaller value.

Case 1: Say the prescaller value is 112 internal clock time period is 2ns which means expected time period of SCLK is ((112+1)*2 = 226ns). What we get from the waveforms

is that, in the first pulse, SCLK is HIGH for 226ns and LOW for 112ns. From second pulse onwards, SCLK is HIGH for 114ns and LOW for 112ns.

Case 2: Say the prescaller value is 31 and internal clock time period is 4ns which means expected time period of SCLK is ((31+1)*4 = 128ns). What we get from the waveforms is that, in the first pulse, SCLK is HIGH for 128ns and LOW for 64ns. From second pulse onwards, SCLK is HIGH for 64ns and LOW for 64ns.

For the first SCLK pulse, SCLK is HIGH for the time of one expected time period duration of SCLK and LOW for the time of half of expected time period duration of SCLK. This is the bug. It is raised and fixed in IISU. Issue number: 155.

CPHA=1 and CPOL=1

Fig 7.9 shows the simulation of testcase for SCLK configuration in receive mode with CPHA=1 and CPOL=1.

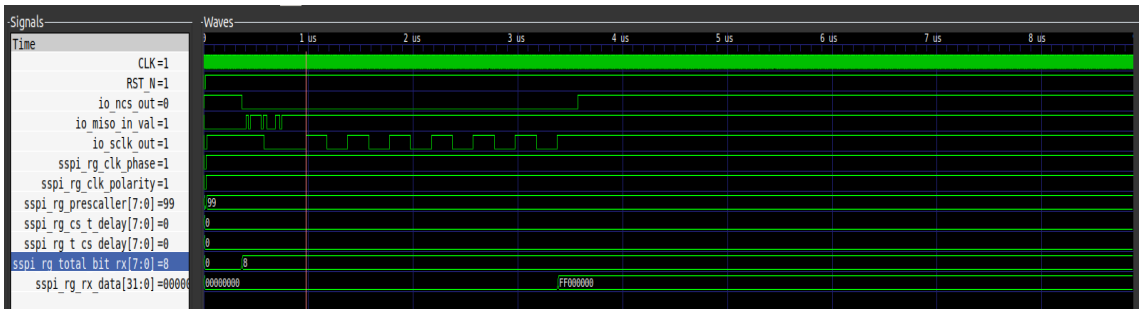

Figure 7.9: SCLK configuration in receive mode with CPHA=1 and CPOL=1

In the waveform we can see that the first pulse of SCLK is not of 50 percent duty cycle. To understand it in much details, let us see two cases, one is odd prescaller value and other is even prescaller value.

Case 1: Say the prescaller value is 99 and internal clock time period is 4ns which means expected time period of SCLK is ((99+1)*4 = 400ns). What we get from the waveforms is that, in the first pulse, SCLK is LOW for 400ns and HIGH for 200ns. From second pulse onwards, SCLK is HIGH for 200ns and HIGH for 200ns.

Case 2: Say the prescaller value is 207 and internal clock time period is 2ns which means

expected time period of SCLK is ((207+1)*2 = 416ns). What we get from the waveforms is that, in the first pulse, SCLK is LOW for 416ns and HIGH for 208ns. From second pulse onwards, SCLK is LOW for 208ns and HIGH for 208ns.

For the first SCLK pulse, SCLK is HIGH for the time of one expected time period duration of SCLK and LOW for the time of half of expected time period duration of SCLK. This is the bug. It is raised and fixed in IISU. Issue number: 155.

7.2.3 RX FIFO Popping Bug

Fig 7.10 shows the simulation of testcase for RX FIFO popping issue in receive mode.

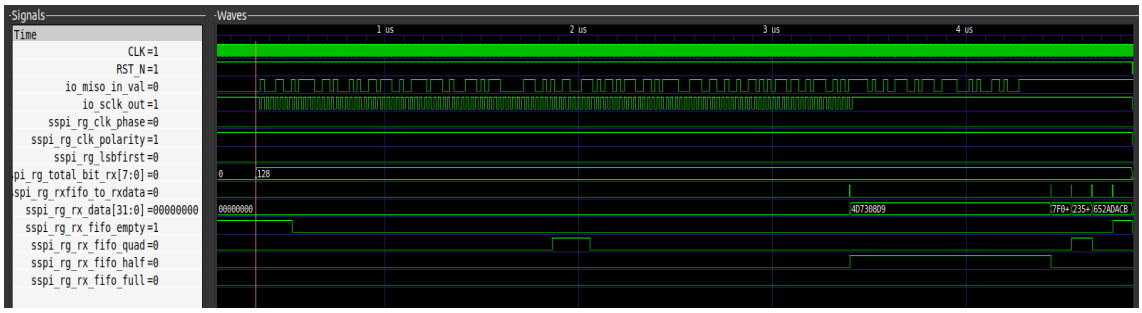

Figure 7.10: Simulation of testcase for RX FIFO popping bug.

In the waveform we can see that we are performing a receive operation of 128 bits. Initially RX FIFO is empty and hence sspi_rg_rx_fifo_empty is 1. After 8 bits are sampled, the first push happens to RX FIFO and hence sspi_rg_rx_fifo_empty now becomes 1. Similarly, after 64 bits are sampled, RX FIFO will have 8 entries and hence sspi_rg_rx_fifo_quad is 1 to indicate that RX FIFO is 1/4th full. After 128 bits are sampled, RX FIFO will have 16 entries and hence sspi_rg_rx_fifo_half is 1 to indicate that RX FIFO is 1/2 full. sspi_rg_rxfifo_to_rxdata is a signal which when goes to 1 means that data is being transferred into sspi_rg_rx_data. According to the specifications and the comments in design code, when the signal sspi_rg_rxfifo_to_rxdata is asserted, data should be copied from FIFO to the rx_data register but the data should not be popped out from FIFO.

In the waveforms we can see that whenever the signal sspi_rg_rxfifo_to_rxdata is asserted, data is popping out from the FIFO and is getting transferred into the rx_data register. This is the bug. It is raised and fixed in IISU. Issue number: 155.

7.2.4 Overrun Bug

Fig 7.11 shows the simulation of testcase for overrun issue in receive mode. Overrun occurs when when the receiver FIFO is full and there is a write to receive data register.

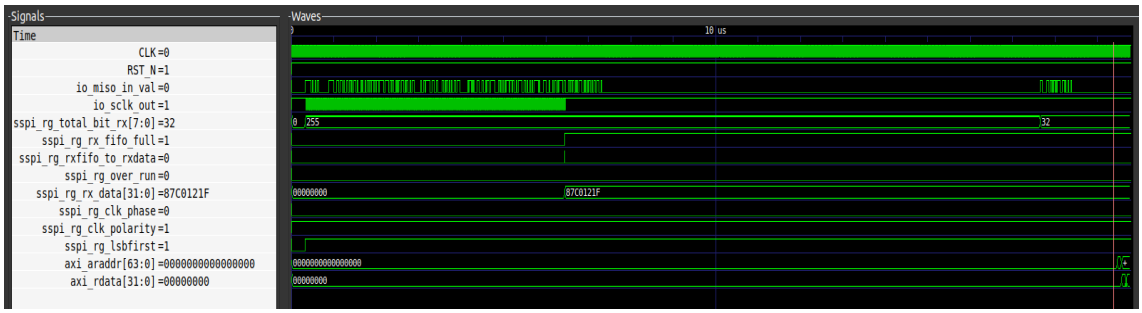

Figure 7.11: Simulation of testcase for overrun bug.

In this testcase, we are first performing a receive of 255 bits. After this receive, we can see that the RX FIFO is full. Then again we are performing a receive of 32 bits. Overrun should now get asserted because after the RX FIFO is full we are again trying to write to the receive data register. But in the waveforms we can see that the signal sspi_r_over_run is 0.

Fig 7.12 shows the simulation of the same testcase and it focuses on the last read operation from the communication status register.

On reading from communication status register we see that the communication status register has the value of 8181 (1000 0001 1000 0001). Communication status register is a 16-bit register whose bit[7] corresponds to the overrun bit. From 8181 we get the bit[7] is 1 which means there is an overrun. The issue here is that the configuration register is showing overrun occurs but the internal overrun register is not getting asserted. This is

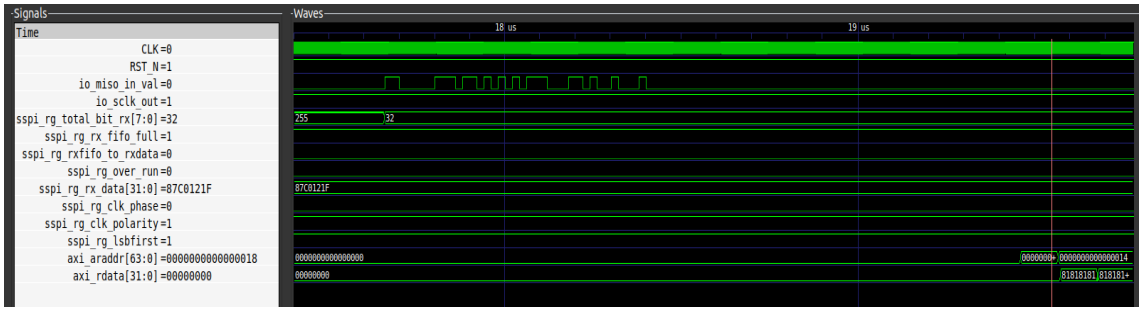

Figure 7.12: Simulation of testcase for overrun bug showing read from communication status register.

the bug. It is raised and fixed in IISU. Issue number: 156.

7.3 CHALLENGES

One of the major challenge in this project is verification in slave mode. The problem is to verify slave mode configuration, we need one master and one slave. If we make two instances of design and connect them via top module, problem is writing to the registers. We have only AXI Master instance and we dont have any AXI slave instance. So this can't be done. I also tried to use only one instance, configure in slave mode and give NCS, SCLK externally. But even this didn't work. One more possibility is to use the cocotb-spi but then I will not be able to see waveforms and I'm verifying using waveforms only. So this also can't be done. Other challenge is that for some testcases, the input configurations were not proper in the documentation. So I had to go through the entire design code and figure out the input configuration needed for that particular testcase.

CHAPTER 8

COMPARISION BETWEEN SV BASED AND COCOTB BASED VERIFICATION

In this chapter we will see an analysis of Verification of SPI using Verification IP(SV based) and Verification of SPI Using CoCoTb.

8.1 SETUP

Setup and installation of VIP is a complicated process. CoCoTb and Verilator is to install and setup. Also, VIP is accessible through a paid license which is costly. CoCoTb is open-source and completely free.

8.2 DESIGN SPECIFICATIONS

VIP is designed for specific design specifications. We have to make additional changes as required to use it for our design. Some specifications of our design may not be present in VIP and hence we can't test the design properly. Using CoCoTb we can test any design as there is no bound of design specifications. In this case, our design has an AXI bus whereas the design based on which the VIP is designed doesnt have any AXI bus. So the AXI integration and testing using VIP is very challenging and complicated.

8.3 TESTPLAN

One of the major advantages of verification using VIP is that the testplan is already designed and it comes along with VIP. The verification engineer doesnt have to take the pain to go through the design specifications and design a testplan manually. In CoCoTb based verification, the verification engineer first needs a thorough understanding of the design specifications. It has to be done very carefully as its a very crucial step in

verification because it directs the entire process of verification.

8.4 TESTBENCH DEVELOPMENT

Another advantage of verification using VIP is that the testbench for each testcase is already developed. The verification engineer just needs to make some changes, configure, enable and run the testbench. But if we want to change any testbench more as per our design specifications, it is again a very complicated process as the testbenches are not straight forward and have many classes which are very interdependent to many other codes. In CoCoTb based verification, the verification engineer has to develop the entire testbench from scratch based on the testcase.

8.5 COVERAGE

In CoCoTb based verification, we can get the functional coverage using CoCoTb by simply adding bins and code coverage can be obtained from verilator. Getting coverage in CoCoTb based verification is pretty simple and straight-forward, whereas in SV based verification using VIP, coverage is a bit complicated. There are many parameters which we need to understand and configure for obtaining coverage.

CHAPTER 9

CONCLUSION

CoCoTb and Verilator were used to successfully verify the Serial Peripheral Interface. Python was used to create all of the testbenches. The correctness of the design was validated by manually comparing DUT output data with expected data according to the specifications using waveforms. To achieve good coverage, constrained random input generation was used. In an SV-based environment, the design was also integrated with the SPI Verification IP and a sample test was done on it.

9.1 FUTURE WORK

- Verification of SPI in slave mode.
- As the code coverage is 92.7 percent, we can still try to increase it more by adding some more tests. Verification in slave mode will definitely increase the coverage by a significant amount.
- Bugs should be resolved from the design end and then re-verify the buggy testcases.
- Verification of SPI using the Verification IP.
- Although this project verified almost the entire SPI, verification in this project is done manually and hence takes a lot of time. This can be improved by making it automatic using assertions and then can be used in regressions.

CHAPTER 10

REFERENCES

1. https://en.wikipedia.org/wiki/Serial_Peripheral_Interface

2. https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html

3. https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html

4. https://indico.cern.ch/event/776422/attachments/1769690/2874927/cocotb_talk.pdf