Rochester Institute of Technology

RIT Digital Institutional Repository

5-2018

Verification of SD/MMC Controller IP Using UVM

Tejas Pravin Warke
tpw8099@rit.edu

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

VERIFICATION OF SD/MMC CONTROLLER IP USING UVM

by

Tejas Pravin Warke

GRADUATE PAPER

Submitted in partial fulfillment
of the requirements for the degree of
MASTER OF SCIENCE
in Electrical Engineering

Approved by:

_____

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

_____

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

DEPARTMENT OF ELECTRICAL AND MICROELECTRONIC ENGINEERING
KATE GLEASON COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK

MAY, 2018

I would like to dedicate this work to my mother for her love and support during my thesis.

Declaration

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

<div style="text-align: right">

Tejas Pravin Warke

May, 2018

</div>

Acknowledgements

Abstract

Wide spread IP reuse in SoC Designs has enabled meteoric development of derivative designs. Several hardware block IPs are integrated together to reduce production costs, time-to-fab/time-to-market and achieve higher levels of productivity. These block IPs must be verified independently before shipping to ensure proper working and conformance to protocols that they are implementing. But, since the application of these IPs will vary from SoC to SoC, the verification environment must consider the important features and functions that are critical for that application. This may mean, revamping the entire testbench to verify the application critical features. Verification takes a major chunk of the total time of the manufacturing cycle. Thus, Verification IPs are created that can be re-used by making minor modifications to the existing test bench. In this project, an Open Cores IP – "SD/MMC Card Controller" (written in Verilog) is re-used by adding an interrupt line and card-detect feature and is verified using Universal Verification Methodology (UVM). The SD/MMC Card Controller has Wishbone as the Host Controller and SPI Master as the Core Controller. The test environment is layered and can be reused. This means, if this IP is re-designed to be controlled by another Host Controller (AXI for example), the verification environment can be re-used by inserting the BFM of that host controller. This paper discusses SD/MMC, Wishbone bus and SPI protocols, along with SD/MMC Controller and UVM based test-bench architecture.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

System-on-Chip (SoC) is a platform integrated with multiple hardware blocks and one or more processors. The existence of a processor distinguishes an SoC from an Application Specific Integrated Circuit (ASIC)[2][3]. Modern SoCs are extremely advanced thanks to massive reuse of hardware IPs for high level integration and product development[4][5]. A typical SoC integrates one or more microcontrollers, microprocessors or a DSP cores[6]. Advanced hardware modules/peripherals like co-processors, GPUs, memory blocks, timing blocks like phase-locked loops, power management blocks and various industry standard interfaces like USB, UART, SPI, Ethernet, etc. are also connected as peripherals[7][8]. These interfaces are commonly used to enable communication between the microprocessor and an external peripheral like SD cards and sensors[9]. All these blocks are connected using a System Bus such as Wishbone and ARM's AMBA bus[9][10]. A SoC is developed with the intention of achieving end-user's requirements through controlled operation and collaboration of all these blocks[10]. One such block is the SD/MMC Card Controller that enables communication between the SoC processor and an SD/MMC card via System Bus[11][12]. The IP used in this project is Open Cores "SD/MMC Controller IP". This IP has an SPI Master based core controller. It supports SPI Bus Accesses

and SD/MMC memory card accesses. The features include:

1. SD/MMC card initialization, block read and block write support

2. 512 byte Rx and Tx FIFOs.

3. 8-bit Wishbone Slave Interface

4. Configurable SPI Core Logic clock (via Wishbone Bus Interface)

5. Data transfer rates upto 24Mbps.

Two additions, namely – interrupt line and card detect, are made to the existing Core IP.

The verification of this IP is done by the vendor (Open Cores) in Verilog using Directed testbench[13]. The testbench performs a card initialization and a single read and write to the SD/MMC Card[14]. This test is not enough to assure complete functional correctness of the Device Under Test (DUT; here – SD/MMC Card Controller). Hence, a layered testbench is written in SystemVerilog using Universal Verification Methodology. Verifying the IP for all the possible input cases is extremely time consuming[15]. But, UVM allowed driving the DUT using a Constrained Random Stimulus, which not only helps in achieving higher functional coverage, but also checked for corner cases that might be evident to the verification engineer. Constrained Random Stimulus helped in reducing simulation time by making sure only valid cases are checked[16]. Invalid cases are ignored. But, it is advisable to manually insert stimulus to check for invalid cases to understand system's behavior. Assertions (both immediate and concurrent) were inserted to manually do property-checking. This SystemVerilog/UVM based test bench is portable and highly reusable. The system bus used for communication is wishbone. If the customer reuses this IP with another bus like ARM's AMBA, this test bench can be modified by replacing the Wishbone's BFM with AMBA's BFM. UVM library used along with System

Verilog automated and reduced the effort for creating sequences for stimulus generation, and data manipulation using features like packing, copy, etc.

This test bench was implemented to test various scenarios like card initialization, block read, block write, card detect, card error, interrupt generation and handling. The test bench architecture is described in chapter XYZ.

1.1   Research Goals

This project dealt with creation of a complex re-usable verification environment to validate SD/MMC protocol. The intended outcomes were:

- Fully understand SD/MMC protocol

- Understand and modify the SD/MMC controller architecture to add additional features

- Develop a test plan to verify the functional correctness of the IP

- Develop a verification environment based on the test plan using SystemVerilog Constructs (with UVM libraries), Constrained Random Stimulus, Assertions, and Coverage.

1.2   Contributions

The major contributions to the project are as below:

1. A test plan is developed to verify the core IP. Includes description of the test environment, stimulus generation, cases to be covered, property checking, etc.

2. A layered testbench is created in UVM.

3. Once the test environment is setup, the DUT is driven using the stimulus generated with the help of Wishbone's BFM and the data written to the SD card is collected using an SD card model. The data is compared to ensure conformance to SD card protocol. A block of data is written to the SD card and the same block is read by the controlling core to ensure successful transmission of data.

4. The results of these tests are gathered in a log file.

1.3   Organization

The structure of the graduate paper is as follows:

- Chapter 2: This chapter deals with the background, operation, design and commands associated with SD/MMC card and the SD/MMC controller IP. It also discusses the protocols to initialize, write a block of data and read a block of data from the SD card.

- Chapter 3: This chapter describes the Design Under Test. The vendor of this IP has made changes to the protocols described in chapter 2 to suit his needs and applications. It also describes the design changes done to the DUT by the author of this paper.

- Chapter 4: This chapter deals with verification methodology used (UVM), the test plan to verify this DUT and the verification environment architecture.

- Chapter 5: This chapter consists of results and discussions.

- Chapter 6: The conclusion, difficulties encountered and future feature additions to the test environment are briefly discussed in this chapter.

Chapter 2

Bibliographical Research

The main idea of this thesis is to introduce a verification environment in UVM for an SoC module or IP which showcases the advantages, disadvantages and features of the environment. Because, verification takes majority of the time in the entire SoC/ASIC design process (time-to-fab), it is important to understand the basic need for verification, methodologies involved in verification, resources available to the industries and their preferred techniques to implement the environment.

A typical verification environment these days is written in SystemVerilog. Learning SystemVerilog can be daunting if the engineer does not have a background in OOP (Object Oriented Programming). [17] is a book that was used to learn the basics of SystemVerilog, how to implement a testbench in SystemVerilog, and how to add a variety of SystemVerilog features available to the engineer. This book brushes up on language syntax, object oriented programming basics, and ways to create a basic layered verification environment.

The latest trends in semiconductor industry for verification involves the use of Universal Verification Methodology (UVM) libraries along with SystemVerilog. The specification (UVM description) is given by [18]. It is a user manual created by Accelera to aid and assist in the use of UVM. It describes all the aspects of UVM that are needed to create a successful testbench.

[19] was incredibly helpful throughout this project. This book describes in detail all the aspects related to UVM. Taking an example of a simple ALU, this book walks the user through the development of a UVM based test environment - starting from a simple driver -dut - tester and then adding tons of additional features each step of the way. The test environment used in this book was helpful in creating a basic skeleton for the testbench used in this project.

The DUT used in the project is a module in a large SoC. The DUT used is a SD/MMC Card Controller IP that was a part of an FPGA Core. General observation of this SoC introduced the author to the concept of reusable IP in SoCs. [20], and [21]shed light on this area. Increase in complicated standards and protocols have resulted in complex architectures of SoCs. This affects the time-to-market of these semiconductor products. Resultant need of rapid development of SoCs introduced IP reuse in SoC Design. An SoC architecture consists of proprietary modules like a bus IP (Wishbone and ARM AMBA), processor/core IP, various interfaces, controller IPs, etc. Assembling these modules to achieve the functionality desired is easier because these IPs are thoroughly verified, and they just have to be instantiated in the SoC.

Verification of these individual modules is done using industry standard UVM. A typical SoC may consist of over 20-25 blocks communicating with eachother. Hence, even though the IP is thoroughly verified, there comes a need to verify the IP modified to suit the requirements of SoC. All the modules in an SoC communicate through a common interface. The verification effort is decreased substantially if the environment can be reused to verify multiple blocks by reconfiguration of a few components in the testbench. [2], [10] and [16] talk about reconfigurable verification environments. The idea of the testbench architecture used in this project is heavily influenced by the ideas of the authors of the above publications.

The IP block in this project is SD/MMC Controller which implements the SecureDigital protocol v2.0 given by www.sdcard.org [22]. This protocol describes in detail a variety of features available to chose from while implementing this protocol. A software implementation of this

protocol helped the author of this project to understand the flow of the protocol [12]. This paper talks about a software implementation of SD Card protocol on an FPGA. This protocol is then used to verify the operation of an image processing algorithm. [23], [14] and [11] speak about different hardware implementation of the SD Card protocol. These references were useful in understanding the protocol in detail. It helped to bifurcate between critical and the optional parts of the protocol.

The verification environment created for this project was based on the environment explained in [19]. But, this environment needed scaling to verify a complicated IP like SD/MMC Controller IP. [24] and [25] describe verification environments created for similar IPs. They were used to add features to this current environment. The DUT in this environment is driven using a Bus Functional Model of Wishbone. This model helps in creating transactions that can be fed to the DUT. The outputs of the DUT generate responses and commands for the SD card model. [10] describes usage of BFMs in a UVM test environment.

All other works cited here helped understanding the technical (syntactical) aspects of UVM and SystemVerilog, tweaking the test environment, and solving problems associated with UVM verification of an SoC Module,

Chapter 3

SD/MMC Controller IP

This chapter deals with the SD/MMC protocol and controller architecture.

The Secure Digital (SD)/ Multimedia Memory Card (MMC) card protocol is given by SD Association (SDA) as a guideline to develop SD Host controllers[22]. This protocol was collectively developed as a joint effort by SanDisk, Panasonic and Toshiba as an improvement over their MMC cards. It is now an industry standard promoted by SDA to meet market demands for low cost, energy efficient, medium sized non-volatile memory[9].

Natively, SD card supports SPI in low cost embedded systems because SPI is widely used and easy-to-implement interface[12]. The SD Card Controller acts as a mediator between the master controller (processor) and the SD card. It acts as an addressable sector which makes reading and writing data from and to SD cards possible. Secure Digital is divided into four categories based on their capacity and functions viz. Standard Capacity (SDSC), High Capacity (SDHC), eXtended Capacity (SDXC), and SDIO. The latter is used for data storage along with I/O functions. MMC card (Multimedia Memory Card) was developed as an upgraded version of SDSC. This IP (DUT) only supports SDSC – MMC cards. The controller has to be configured during initialization phase to support other type of cards. But due to absence of the need to

support other types, this DUT is run in default SDSC mode.

3.1   SD Card Commands

Table 3.1: SD Card Command words [1]

Command	Argument	Response	Data	Description
CMD0	None	R1	No	Software Reset
CMD1	None	R1	No	Initialize SD card
ACMD41*	*2	R1	No	For SDC: Initialize SD card
CMD8	*3	R7	No	Voltage check
CMD9	None	R1	Yes	Read CSD
CMD10	None	R1	Yes	Read CID
CMD12	None	R1b	No	Stop reading data
CMD16	Blocklength[31:0]	R1	No	Change R/W block size
CMD17	Addr[31:0]	R1	Yes	Read a block
CMD18	Addr[31:0]	R1	Yes	Read multiple blocks
CMD23	No_of_blocks[15:0]	R1	No	For MMC only: Set number of blocks for next R/W
ACMD23*	No_of_blocks[15:0]	R1	No	For SDC only: Set no. of blocks to be erased before next R/W
CMD24	Addr[31:0]	R1	Yes	Write a block
CMD25	Addr[31:0]	R1	Yes	Write multiple blocks
CMD55*	None	R1	No	Leading command of ACMD[i] command
CMD58	None	R3	No	Read OCR
Command* indicates a sequence of CMD55-ACMD command				

The SD Card Controller has a fixed set of activation commands and responses that allows for proper communication with the SD card. Data is transmitted in an 8-bit/ word (byte) format. Each command has a fixed length of 6 bytes. The six bytes are as follows:

1. The first byte is the addition of 64 + the command number. For ex. For resetting the SD card, CMD0 has to be asserted. CMD0 has the command number 0. So 0x40 has to be written in the first byte of CMD0 register to assert a soft reset.

2. The next four bytes are the arguments passed along with the initial byte. These arguments generally contain data address or block length.

3. The last byte is CRC byte. CRC stands for Cyclic Redundancy Check, which looks for transmission errors. Most commands in SPI mode don't require CRC check and hence, it is disabled. For ex. For CMD0 (soft reset) 0x95 is sent, while in all other cases 0xFF is sent.

3.2   SD Card Command frame:

The SD card receives a command whenever the DO (Data out) is set. The CS (chip select) is driven high to low before sending the command, and stays low throughout the operation[14]. The time between the command and the response is NCR (Command Response Time)[26]. To manage clock domain crossing or prevent data getting skipped, 8 pulses are inserted between command and response. While receiving data, DI (Data in) is reset. The responses from the Card are stored in R1 register. The value of this register indicates: 0x01 = Command that was sent prior to response made the card go into Idle state 0x00 = Card has been accepted and the card will await an event to take place. Any other bits set in R1 response indicates an error.

1. R1[1] = Erase Reset

2. R1[2] = Illegal Command

3. R1[3] = Command CRC error

4. R1[4] = Erase Sequence Error

5. R1[5] = Address Error

6. R1[6] = Parameter error

3.3   SD Card Architecture

The SD card contains a controller inherent to it which controls the functionality of the SD card. It handles card initialization protocol, flash memory accesses and communication with the SD card controller. Data transfer between the SD card controller and this inherent controller is performed in clock serial mode and in 512 byte blocks. The file system supported by this SD card controller is FAT12/FAT16[27, 28].

Pinout

Pin 1: Data2
Pin 2: Data 3
Pin 3: CMD I/O
Pin 4: GND
Pin 5: VDD
Pin 6: CLK
Pin 7: GND
Pin 8: Data0
Pin 9: Data1
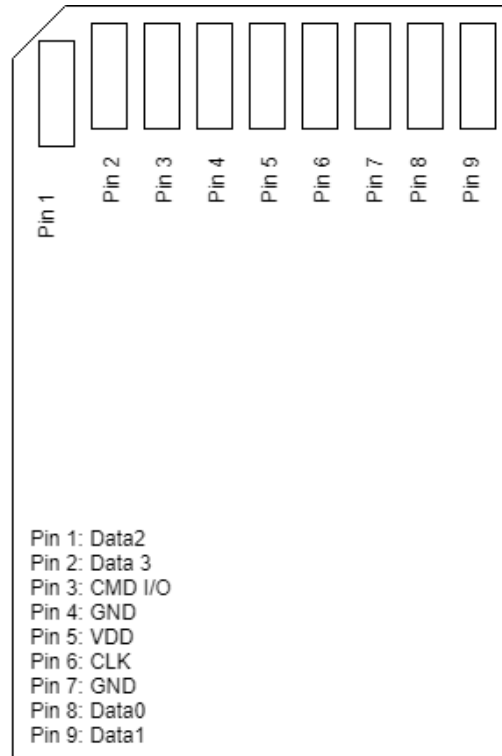
Figure 3.1: SD Card Pinout

Bus protocol

The default mode is standard SD mode, but is can be changed to support a slow SPI mode. This protocol selection is performed with the first reset command after power up. By default, the card starts in SD mode. To change it to SPI mode, a CS signal is asserted when the SD Card Controller asserts a reset command. This protocol selection once done, cannot be undone or changed until the next power cycle.

In SPI mode, card only uses 1 bit data bus. All the commands are in multiple of 1 byte. New response tokens are defined for operation in SPI mode.

3.4   SD Card Protocols

3.4.1   Initializing the card

While initializing the card, the card is most probably in the idle state. Hence, only legal commands are CMD0, CMD1, ACMD41, CMD58 and CMD59. Any other command asserted will set R1[2] bit.

Initialization of SD and MMC card have some primary difference in the command sequence. The steps are mentioned below:

1. SPI clock is set to 400 kHz. This is essential for compatibility between SD card types. (SDSC and MMC in this case).

2. SD Card Controller instructs the SD card to communicate using SPI by setting SS and CS lines. Then, the SDCLK is toggled for 74 clock pulses before sending any command word. This step is necessary for the card to initialize all the internal registers.

3. CMD0 is asserted to reset the card. Bit sequence: 01 000000 00000000 00000000 00000000 00000000 1001010 1

4. CRC check is ignored in SPI mode. But, the card is expecting a CRC check after reset. Hence, a default value of 0x95 is sent with CMD0. To make this simple, this CRC byte is sent with every command.

5. The SD card responds with the message (00000001) which indicates that the card is now in Idle state. This message is received by holding CS line low and MOSI line high.

6. CMD58 is asserted to check the operating conditions register (OCR) of the SD card. This register contains information like voltage, version number, special functions, etc. This command returns a bit field containing allowed operating voltages (generally, between

2.7v to 3.6v). The SD card responds with a 40-bit word (say R2) where the first 8 bits are standard SD card response while the next 32 bits contain information specific to that SD card. R2[39:32] – If successful, the value will either be 00000001 or 00000101 depending on the SD card. R2[31:28] – Version number R2[27:12] – Reserved for additional/special functionality R2[11:8] – Voltage code R3[7:0] – CRC The response to this command tells if the SD card is compatible with this controller.

7. CMD55 and ACMD41 are asserted to initialize the SD card internally.

8. Step 7 may have to be repeated several times depending on the type of the card. The steps mentioned above also vary from card to card, and additional steps or more iterations of the above steps are needed to successfully initiate a card. SD Card initialization routine

3.4.2 Data Transfer

In data transfer transactions, a data block is received or sent after the command response. This data block is transferred in the form of a data packet which consists of Token, Data Block and CRC. The format of this packet is shown below:

Figure 3.2: SD Card Data Packet

There are three data tokens depending on the CMD asserted. If a valid response cannot be sent by the SD card, an error packet is sent; described as the fourth byte in Fig. 2.2 is the registered Error packet. The fifth byte contains CRC errors.

3.4.2.1 Writing a block of data to SD card

Once the card is initialized, data transfer operations are pretty straight forward. For writing a block of data:

1. CMD24 is asserted. Once, a valid response (11111110) is obtained, the host controller sends a data packet to the card.

2. The card sends a data response as soon as it received the data packet from the host controller. This response consists of a busy flag. The host controller must wait until the busy flag is cleared to perform a new transaction[29].

3.4.2.2 Reading a block of data from SD card

For reading a block of data:

1. CMD17 is asserted. Once, a valid response (11111110) is sent to the host controller and the card begins the read operation. The host controller then receives the block of data from the SD card.

2. Once the valid token is detected, the host controller obtains the data field and CRC bytes. The CRC bytes must be transmitted to the controller even though CRC information is not necessary.

3. If an error occurs, an error token is transmitted instead of data packet to the host controller[29].

Chapter 4

Device Under Test

This chapter discuses in detail the DUT Architecture, Design modifications and Communication protocols.

The SD Card Controller IP operates as an SPI Master module. It supports SPI bus accesses and communication with SD/MMC Cards.

This module communicates with the master controller using wishbone interface. The interface is responsible for creating clocks and reset signals for the module. Register block contains all the registers that are to be written to or read from to successfully carry out a task like block_read() or block_write(). The values and the order in which those values are written to the register file is described in the SD protocol. Clock manager creates SPI clock. Power manager creates appropriate VDD signal for the SD card depending on the voltages supported by the card.
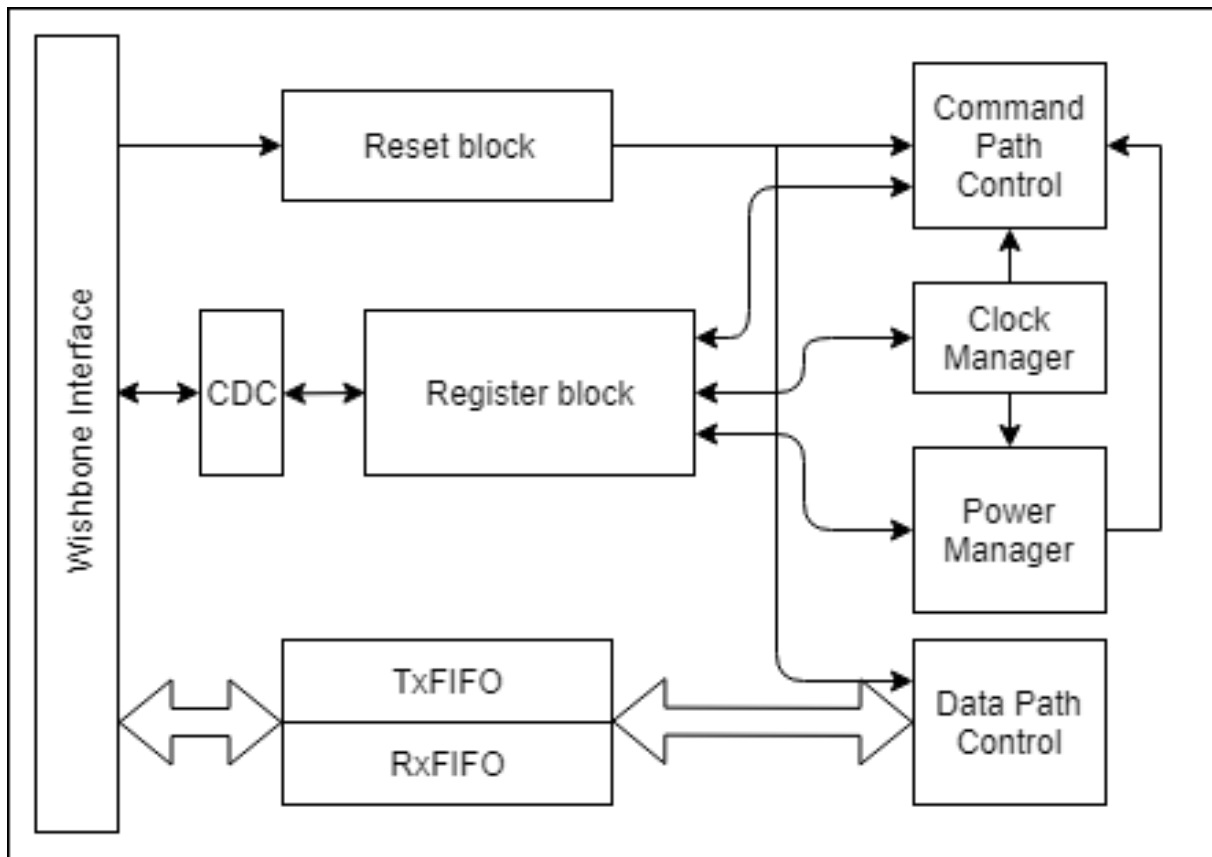
Figure 4.1: SD/MMC Controller Architecture

4.1   IO Ports

Figure 4.2: SD/MMC Controller IO ports

This DUT has a standard wishbone interface to communicate with the master controller (processor/mcu) and an SPI interface to communicate with the SD/MMC card.

4.2   Register File

The register description for each register is given below. A typical SD Card controller DUT consists of all these registers. A few registers are added to this controller especially to relieve the complicated SD card protocols. For eg. Initializing an SD card consists of 10-12 command assertions. This DUT enables the user to do the same operation using less than 3 steps. The simplified protocols for this DUT are described in the following sections.

Table 4.1: SD/MMC Controller Registers

Register Name	Bit Position	Description	Default Value	R/W
TRANS_ERROR_REG	[5:4]	0: WRITE_NO_ERROR 1: WRITE_CMD_ERROR 2: WRITE_DATA_ERROR 3: WRITE_BUSY_ERROR		R
	[3:2]	0: READ_NO_ERROR 1: READ_CMD_ERROR 2: READ_TOKEN_ERROR		R
	[1:0]	0: INIT_NO_ERROR 1: INIT_CMD0_ERROR 2: INIT_CMD1_ERROR		R
DIRECT_ACCESS_DATA_REG	[7:0]	TX_DATA is set prior to starting a direct access transaction. SPI bus does not define a read/write transaction, hence, every direct access transmission begins by data transmission from master and ends by data reception from the slave.	0	W

Table 4.1: SD/MMC Controller Registers

Register Name	Bit Position	Description	Default Value	R/W
TRANS_ERROR_REG	[5:4]	0: WRITE_NO_ERROR 1: WRITE_CMD_ERROR 2: WRITE_DATA_ERROR 3: WRITE_BUSY_ERROR		R
	[7:0]	Read the data obtained from TX_Data by reading contents of this register		R
TRANS_ERROR_REG	[5:4]	0: WRITE_NO_ERROR 1: WRITE_CMD_ERROR 2: WRITE_DATA_ERROR 3: WRITE_BUSY_ERROR		R
	[3:2]	0: READ_NO_ERROR 1: READ_CMD_ERROR 2: READ_TOKEN_ERROR		R
	[1:0]	0: INIT_NO_ERROR 1: INIT_CMD0_ERROR 2: INIT_CMD1_ERROR		R

Table 4.1: SD/MMC Controller Registers

Register Name	Bit Position	Description	Default Value	R/W
TRANS_ERROR_REG	[5:4]	0: WRITE_NO_ERROR 1: WRITE_CMD_ERROR 2: WRITE_DATA_ERROR 3: WRITE_BUSY_ERROR		R
DIRECT_ACCESS_DATA_REG	[7:0]	TX_DATA is set prior to starting a direct access transaction. SPI bus does not define a read/write transaction, hence, every direct access transmission begins by data transmission from master and ends by data reception from the slave.	0	W
	[7:0]	Read the data obtained from TX_Data by reading contents of this register		R

Table 4.1: SD/MMC Controller Registers

Register Name	Bit Position	Description	Default Value	R/W
TRANS_ERROR_REG	[5:4]	0: WRITE_NO_ERROR 1: WRITE_CMD_ERROR 2: WRITE_DATA_ERROR 3: WRITE_BUSY_ERROR		R
SD_ADDR_7_0_REG	[7:0]	Normally, this register is set to 0, because accesses to memory occur on a 512 byte data boundary. A value is written to this register before starting a read/write operations	0	R/W
SD_ADDR_15_8_REG	[7:0]	Normally, SD_ADDR[8] bit is set to 0, because accesses to memory occur on a 512 byte data boundary.	0	R/W
SD_ADDR_23_16_REG	[7:0]	Next 8 bits of SD_ADDR	0	R/W
SD_ADDR_31_24_REG	[7:0]	Next 8 bits of SD_ADDR	0	R/W

Table 4.1: SD/MMC Controller Registers

Register Name	Bit Position	Description	Default Value	R/W
TRANS_ERROR_REG	[5:4]	0: WRITE_NO_ERROR 1: WRITE_CMD_ERROR 2: WRITE_DATA_ERROR 3: WRITE_BUSY_ERROR		R
SPI_CLK_DEL_REG	[7:0]	This register controls the clock frequency of SPI after SD card initialization is successful. To change this frequency before initialization, SLOW_SPI_CLK constant should be modified. SPI_CLK_DEL = (SPI System Clock Freq. / (SPI_CLK * 2)) - 1	0	R/W
RX_FIFO_DATA_REG	[7:0]	Fifo for 512 bytes of block read data. Size of fifo is equal to the SD/MMC block size of 512 bytes		R

Table 4.1: SD/MMC Controller Registers

Register Name	Bit Position	Description	Default Value	R/W
TRANS_ERROR_REG	[5:4]	0: WRITE_NO_ERROR 1: WRITE_CMD_ERROR 2: WRITE_DATA_ERROR 3: WRITE_BUSY_ERROR		R
RX_FIFO_DATA_COUNT_MSB	[7:0]	MSB of FIFO_DATA_COUNT. It indicates the number of entries currently in the FIFO		R
RX_FIFO_DATA_COUNT_LSB	[7:0]	LSB of FIFO_DATA_COUNT. It indicates the number of entries currently in the FIFO		R
FIFO_FORCE_EMPTY	0	1: Forces FIFO EMPTY. Deletes all the entries in the FIFO. Self clearing	0	W
TX_FIFO_DATA_REG	[7:0]	Fifo for 512 bytes of block write data. Size of fifo is equal to the SD/MMC block size of 512 bytes	0	W

Table 4.1: SD/MMC Controller Registers

Register Name	Bit Position	Description	Default Value	R/W
TRANS_ERROR_REG	[5:4]	0: WRITE_NO_ERROR 1: WRITE_CMD_ERROR 2: WRITE_DATA_ERROR 3: WRITE_BUSY_ERROR		R
TX_FIFO_CONTROL_REG	0	1: Forces FIFO EMPTY. Deletes all entries in FIFO. Self clearing.	0	W

4.3 Clocks

This module has two clocks. One is the SPI System Clock (spiSysClk) and the other is Wishbone Bus Clock (clk_i). Both these clocks have a duty cycle of 50%. Wishbone clock is bounded by the value of spiSysClk. Max. frequency of clk_i is (spiSysClk * 5) while the minimum frequency is equal to the frequency of spiSysClk.

4.4 Simplified Protocols for this DUT

The creator of this DUT has internalized all protocol transactions. The protocol is implemented in the form of a request/action handshake. The type of transaction is determined by the value of SPI_TRANS_TYPE_REG. The transaction can initialize the card, write a block of data to the card or read a block of data from the card. The transaction initiates once SPI_TRANS_CTRL_REG

register is set. Once, SPI_TRANS_STS_REG is reset, error check is done by registering the contents of SPI_TRANS_ERROR_REG. The transaction continues or ends depending on the type. Thus, this level automation has simplified the complex SD protocol[30].

4.4.1   SD Card: Initialization

1. Set SPI_TRANS_TYPE_REG = SPI_INIT_SD

2. Set SPI_TRANS_CTRL_REG = SPI_TRANS_START

3. Wait for SPI_TRANS_STS_REG != TRANS_BUSY

4. Check for SPI_TRANS_ERROR_REG [1:0] == INIT_NO_ERROR

4.4.2   SD Card: Block data write

1. Write 512 bytes to SPI_TX_FIFO_DATA_REG

2. Set the SD block address registers: SD_ADDR_7_0_REG, SD_ADDR_15_8_REG, SD_ADDR_23_16_R and SD_ADDR_31_24_REG

3. Set SPI_TRANS_TYPE_REG = SPI_RW_READ_SD_BLOCK

4. Set SPI_TRANS_CTRL_REG = SPI_TRANS_START

5. Wait for SPI_TRANS_STS_REG != TRANS_BUSY

6. Check for SPI_TRANS_ERROR_REG[5:4] == WRITE_NO_ERROR

4.4.3   SD Card: Block data read

1. Set the SD block address registers: SD_ADDR_7_0_REG, SD_ADDR_15_8_REG, SD_ADDR_23_16_RI

   and SD_ADDR_31_24_REG

2. Set SPI_TRANS_TYPE_REG = SPI_RW_READ_SD_BLOCK

3. Set SPI_TRANS_CTRL_REG = SPI_TRANS_START

4. Wait for SPI_TRANS_STS_REG != TRANS_BUSY

5. Check for SPI_TRANS_ERROR_REG[3:2] == READ_NO_ERROR

6. Read 512 bytes from SPI_RX_FIFO_DATA_REG

Chapter 5

Verification Methodology

A test environment is a collection of verification components that are designed to check if the Device Under Test conforms to the design specifications. A typical test environment setup is as follows:

1. Generate stimulus, sequences, transactions based on the DUT input BFM

2. Reset the DUT

3. Feed the generated sequences to the DUT.

4. Run the tests. Insert property checking/assertions.

5. Capture the outputs generated by the DUT

6. Compare them to the expected outputs

7. Verify if they are correct. Verify if the assertions fail.

8. Create a log file for errors/mismatches/failures

9. Provide possible solutions.

All the above things can be done using a single file which merges all the components. But, as the complexity of the DUT increases, verification effort increases exponentially. Hence, the concept of layered test bench is used. Using a layered testbench (in SystemVerilog) involves creating various classes that do a particular task and communicate with each other to achieve the final aim of verifying the DUT.

Creating a layered test bench has various advantages. The testbench has a software like construct which supports OOP. It is good for verification of complex, high level DUTs where a lot of modules communicate with each other. In the test environment, each component (class) can be individually edited/tweaked to make it more feature-rich. The testbench can be reused to test a similar DUT by editing a few components to satisfy the requirements of that DUT. This testbench can interact with modules written in C too enabling a more diverse, inclusive test environment[19].

5.1   Layered Testbenches

5.1.1   Turning simulation into verification

Directed tests using Verilog to test the complete DUT is impractical as the level of abstraction and the complexity of DUTs increases. Hence, a good verification methodology takes into consideration the function DUT is supposed to perform. A test plan or verification plan is derived based on this function as it is broken down property by property. This verification plan includes methods to test each of these properties.

Complex systems should not be verified manually by looking at waveforms. This makes the verification effort difficult and time consuming. Instead, automating the function checking is done. This automation is a collection of verification metrics such as property checking as

described by the verification plan. To ease this process, a functionally correct model is created to form the basis on which the DUT is compared against. Ideally, a verification process should involve directed tests of low level modules to ensure 100% coverage, and then, as the level of abstraction increases, random stimulus should be inserted. Random test vectors test for bugs which would have been otherwise undiscoverable, while directed tests give good overall coverage. As a compromise between the two, constrained random tests are performed which helps achieve high coverage by making use of a series of good engineering judgements[31].

5.1.2   3 C's of verification

Coverage, checkers and constraints play a key role in the verification process. Checkers ensure functional correctness. This is achieved by feeding random stimulus to a DUT to improving coverage. Checkers can be implemented in SystemVerilog as assertions or using procedural code. Recently, assertions are embedded inside the DUT and placed on external interfaces to ease the verification effort.

Coverage also provides a measure of completeness of testing. It indicates that the goals set in the test plan are met. When the simulation is done, SystemVerilog offers mechanism for coverage collection. They could be property based coverage (cover directives) or sample based coverage (covergroups).

Constraints help achieve test plan coverage goals by shaping the random vectors to ensure the DUT is tested for corner cases. Without constraints, it'd be difficult to discover these corner faults as the verification time will increase exponentially. Constraints ensure that corner cases are reached by statistically shaping the randomness of the generated test vectors.

5.1.3   Engineering effort

Using constraints in random stimulus generation, verification effort shifts from writing directed tests to designing automatic checkers and executable coverage models. This enables the random stimulus to efficiently utilize resources to achieve higher coverage goals. This also makes the verification process a little more predictable and transparent.

Using reusable environments saves engineering time. It allows flexibility to configure and use various verification components. Hence, to ensure reusability and take advantage of automation, a layered testbench architecture needs to be set. UVM architecture is thus designed, keeping in mind all the above points.

5.2   UVM

UVM is a library of SystemVerilog constructs. It was created by Accelera based on OVM (Open Verification Methodology v2.1.1). Its a method of verifying the functionality of digital hardware using simulation. This hardware can be verified at behavioral, register transfer (RTL) or transaction level using Verilog, SystemVerilog, SystemC, VHDL, etc. based on the abstraction level desired by the user [32]. UVM acts a bridge which facilitates simulation oriented methodologies, hardware acceleration or emulation. It also supports assertion based verification[18].

Advantages of UVM [33]:

1. UVM Source Code is available.

2. Code reusability as a product, service or a framework.

3. Phasing extensions

4. Sequence Mechanism

5. TLM interfaces

6. Resource Database

7. End of Test Mechanism

8. Command line processor

5.3 UVM based test environment verification of SD/MMC Controller IP

5.3.1 UVM classes

UVM library contains all the blocks that are needed for quick development of a reusable verification environment in SystemVerilog.

Based on the usage and applications, UVM classes are divided into following categories:

Globals   Classes included in this category contain a small list of data types, variables, functions and tasks that are scoped by uvm_pkg.

Base   These classes consist of basic blocks that are needed to build any verification environment. They pass information between components and ports using interfaces. The core base classes used in this testbench are described in the following sections. uvm_object, uvm_component, uvm_transaction and uvm_root are core base classes that facilitate the modularity and reusability of the test bench.

Reporting   These classes provide a functionality that enables the user to issue messages, errors, reports, logs, etc.

Factory   The Factory is used to create UVM objects and components based on the config-uration provided by the user. Using this class type allows the user to configure the hierarchy dynamically without breaking encapsulation.

Phasing   All UVM classes are simulated in phases. The classes in this section describe the phasing capability provided by UVM.

Configuration and Resources   They are a set of classes that provide configuration database. This database is used to store and load properties that are needed at configuration time as well as run time.

Synchronization   These classes are used for synchronization of processes.

Containers   These classes contain parameterized data structures that provide queue and pool services.

Policies   Each class in this type are used to perform a particular task like printing, compar-ing, recording, etc. for a uvm_object-based objects. They are not included in uvm_object to enable user to print and compare without modifying the object class.

TLM   This library provides an elaborate abstract transaction-level interfaces and ports that are used to transport data, which are generally whole transactions (objects) at one time. TLM ports are inherently more reusable and modular.

Components   This library forms the foundation of UVM. They encapsulate behavior or classes like drivers, scoreboards, etc. All the components are derived from this library directly or indirectly.

Sequencers   Sequencers act like data arbiters that control transaction flow from multiple stimuluses.

Sequences   Sequences encapsulate user defined methods that generate multiple sequence based transactions.

Macros   Macros are used to improve productivity.

Register Layers   These are abstract classes that are used to perform read/write operations to memory models inside the DUT.

Command Line Processor   This library provides an interface to decode command line arguments.

5.3.2   Test plan

SD/MMC controller is verified like any other memory. Data is written to a location, and then read back. If the data that is read matches the data that was written, the controller functions properly [34].

The stimulus to the controller is 32 bit address and 8 bit data. Both of these stimuli are randomized. The address is constrained to address only 2 MB (2^{21} locations) of memory, because only that much memory is instantiated in the SD Model. This controller only reads and writes data in multiples of a 512 byte block, an additional constraint that the start address of the data block to be written or read is less than (2^{21} - 512 - 1). This ensures that the data block doesn't go out of bounds.

The operation of this DUT at a very abstract level can be divided into 4 tasks: write_to_controller_fifo, read_from_controller_fifo, write_to_sd_card and read_from_sd_card. Initially, the test plan was

to randomize these 4 tasks, so that they can read/write to any location in the SD card and at the same time, the data inside the FIFO can be manipulated. But, due to absence of an interrupt mechanism, synchronizing these tasks between the stimulus generator (sequencer in UVM) and checker (monitor/scoreboard in UVM) would have been really difficult.

To avoid the above complications, a simpler test plan is created. The stimulus generator generates random address and data values and feed it to the write_to_fifo_task. After receiving an acknowledgement that the data has been successfully written to the TX_FIFO, a command sequence directing the DUT to write the data stored in the FIFO to the SD card is sent. The stimulus generator waits for a fixed amount of time and then, it polls for a TRANS_BUSY flag. Once, the the busy flag is reset, the stimulus generator sends a command sequence to the DUT to read the data block from the SD card back to the RX_FIFO in the DUT. The data from the RX_FIFO is captured in the monitor by calling the task read_from_controller_fifo. To synchronize the two events, a handshaking method is established between the stimulus generator (driver) and the checker (monitor). Two events, run_monitor and run_driver are created. The driver generator waits for run_driver to be triggered before it fetches the next write sequence from the sequencer. At the same time, once the driver has successfully sent a read_from_sd_card command sequence to the controller, run_monitor event is triggered, which tells the monitor to start capturing the data from the controller's RX_FIFO. The data written to the TX_FIFO by the driver and the data captured from the RX_FIFO by the monitor are both compared in scoreboard using TLM FIFOs. The scoreboard keeps a track of successful read and write transaction.

Functional coverage is obtained by creating coverpoints in the driver, where the random data from the sequencer is received.

Figure 5.1: Testbench architecture
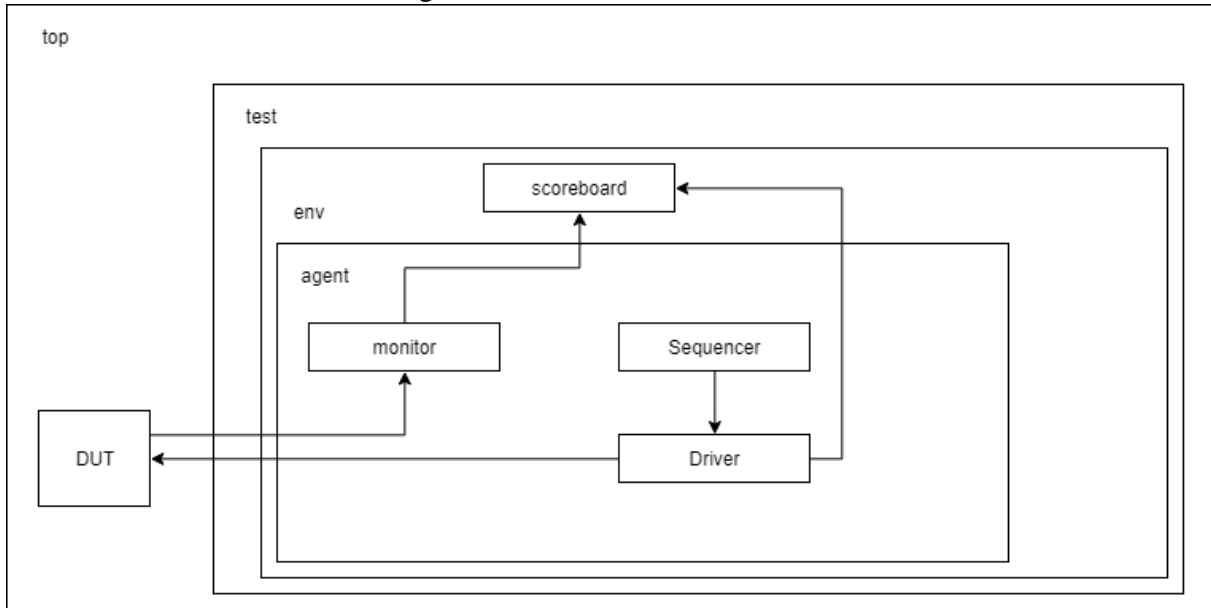
5.3.3   Testbench Architecture

Fig. 4.1 describes the testbench architecture implemented in this project. The following subsections describe each block in detail. Fig. 4.1 is a very broad overview of the architecture. Many more blocks function inside these classes in shadows.

5.3.3.1   DUT

The DUT in this architecture refers to the SD/MMC controller IP from Opencores. This DUT is modified to include an interrupt controller and a card detect which can be read via wishbone by the master core.

5.3.3.2   Top module

This module connects the DUT and the verification environment components. It can be used to instantiate multiple agents to drive the same DUT on different systems.

5.3.3.3   Interface

Interface encapsulates connectivity and functionality. All the modules instantiated within the environment can be connected using a single interface. This adds a level of abstraction and granularity to connections between the modules. This enables addition and deletion of ports throughout the system by just editing one interface.

5.3.3.4   Agent

Agent can be of two types: active or passive. Active agents contain all the three components viz. driver, sequencer and monitor. A passive agent only contains the monitor; it samples DUT signals, but doesn't drive the DUT.

5.3.3.5   Sequencer, Driver and Monitor

A sequencer consists of transaction items that are declared using factory mechanism. These transaction consists of a sequence of stimulus for the DUT to perform a protocol or a functional operation like sd_init(), sd_block_write() and sd_block_read() in this test bench. These sequences contain a series of tasks that are called. The driver drives the DUT using these tasks. The sequencer sequences these tasks for the driver. Monitor samples the data from the DUT for comparison in the scoreboard.

5.3.3.6   Coverage and Scoreboard

Scoreboard is a comparator which compares samples from the monitor against a database or table or vectors to verify if the data processed by DUT is correct. Coverage offers coverpoints inside the test environment to ensure visibility and observability.

5.3.3.7   Environment

Environment consists of static and dynamic components. DUT, interfaces, anything physical are static components. Dynamic components are the components that are generated or modified at run time. The environment encapsulates the agent, scoreboard, coverage collector, sequencers, configuration databases, etc.

5.3.3.8   Test

Test defines the test scenario for the testbench. A UVM testbench is activated when run_test() method is called. run_test() method is specified inside the initial block. A user can add testcases by providing arguments to run_test().

5.3.3.9   Sequence item

This item consists of data fields required for stimulus generation.Stimulus is generated by randomizing sequences. Data in sequence item is defined as rand data type. Constraints are also added.

5.3.3.10   Sequence

Sequence generates series of sequence_items and sends it to driver using sequencer.

5.3.3.11   Wishbone BFM

Wishbone BFM acts as a Verilog module that generates stimulus signals for the SD/MMC controller IP. Typically, this BFM is instantiated inside sd_mmc_tb_top module, but in this environment, this BFM is stripped down and tasks from this BFM are inserted directly in the driver.

5.3.3.12   SD card model

SD card model acts as the output device that connects to the SD/MMC controller. It is a memory-less model.

Chapter 6

Results and Discussion

This chapter presents the results of the test environment. The post-simulation tasks are listed below:

- The DUT is verified using test sequences generated by the test environment and comparing the outputs with the expected data. This data is available in the scoreboard in a 512 byte packet.

The "SD/MMC Controller IP" was Synthesized on a different technology nodes using two different synthesis options, RTL logic synthesis and DFT Synthesis with a full scan methodology. Area, Power, Timing and DFT coverage analysis for the 32nm, 65nm, 180nm is tabulated in 6.1
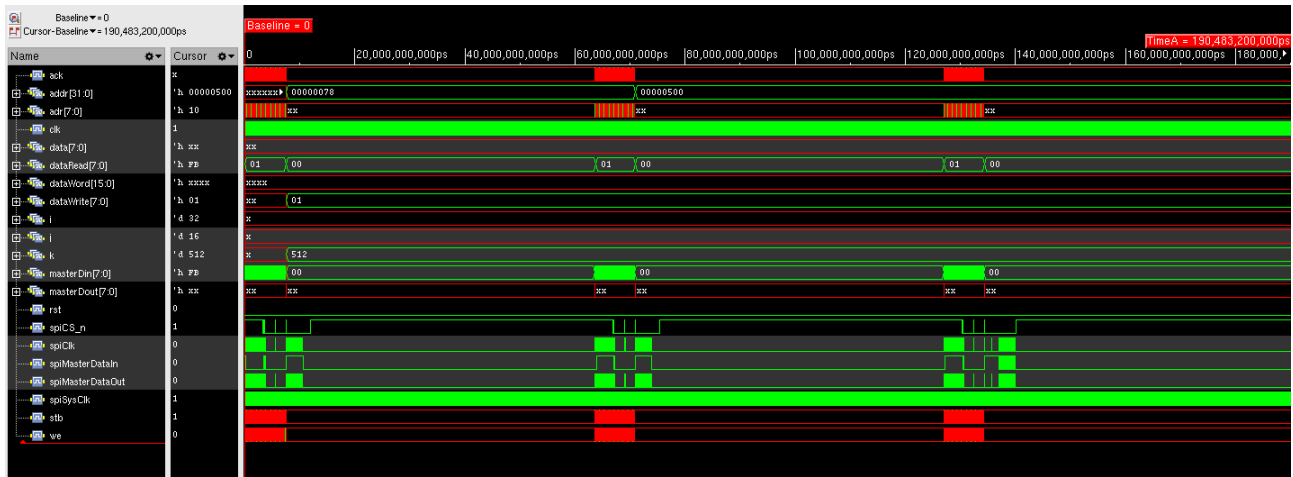
Figure 6.1: DUT Simulation

Table 6.1: Area, Power, Timing and DFT Coverage of AES Encryption

		32nm	65nm	180nm
Area	Combinational Area (μm^2)	145792.78	20903.4	145792.78
	Buf/Inv Area (μm^2)	6935.544	684.72	6935.544
	Non-Combinational Area (μm^2)	778261.15	103228.28	778261.15
	Total Area (μm^2)	924053.94	124231.68	924053.94
Power	Internal Power (W)	0.0200	6.268E-09	0.0200
	Switching Power (W)	5.733E-04	1.755E-07	5.733E-04
	Leakage Power (W)	3.981E-06	1.310E-07	3.981E-06
	Total Power (W)	0.0206	3.128E-7	0.0206
Timing	Slack (ns)	14.3880	17.4700	14.4070
DFT Coverage	(%)	100	100	100

DUT Simulation, Wishbone Simulation, RX_FIFO, TX_FIFO, Console Window and the final Memory Map are given below.

Figure 6.2: Wishbone Simulation

Figure 6.3: Console Window

Figure 6.4: RX_FIFO Memory Map

Figure 6.5: TX_FIFO Memory Map

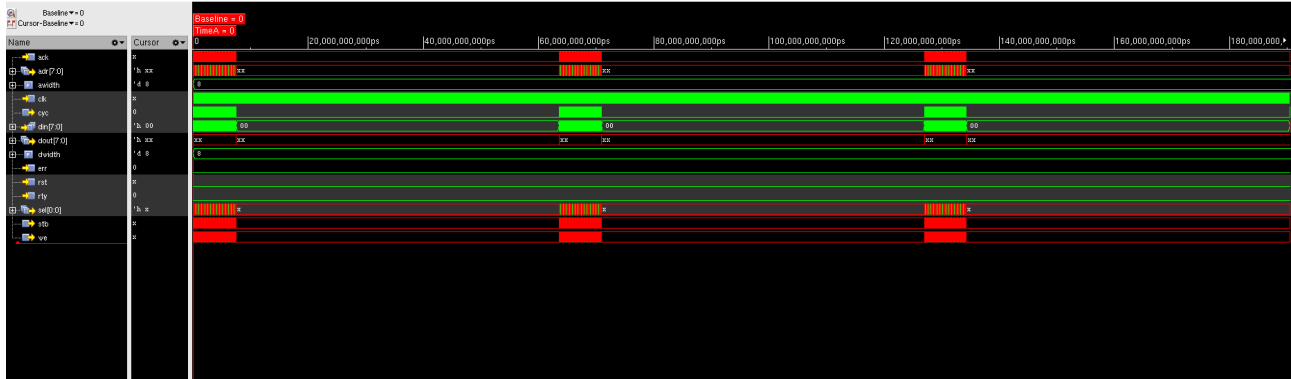|	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51
0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
52	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
104	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	01	04	07	0A	0D	10	13	16	19	1C	1F	22	25	28	2B	2E	31	34	37	3A	3D	40	43	46	49	4C	4F	52	55	58	5B	5E	61	64	67	6A
156	6D	70	73	76	79	7C	7F	82	85	88	8B	8E	91	94	97	9A	9D	A0	A3	A6	A9	AC	AF	B2	B5	B8	BB	BE	C1	C4	C7	CA	CD	D0	D3	D6	D9	DC	DF	E2	E5	E8	EB	EE	F1	F4	F7	FA	FD	00	03	06
208	09	0C	0F	12	15	18	1B	1E	21	24	27	2A	2D	30	33	36	39	3C	3F	42	45	48	4B	4E	51	54	57	5A	5D	60	63	66	69	6C	6F	72	75	78	7B	7E	81	84	87	8A	8D	90	93	96	99	9C	9F	A2
260	A5	A8	AB	AE	B1	B4	B7	BA	BD	C0	C3	C6	C9	CC	CF	D2	D5	D8	DB	DE	E1	E4	E7	EA	ED	F0	F3	F6	F9	FC	FF	02	05	08	0B	0E	11	14	17	1A	1D	20	23	26	29	2C	2F	32	35	38	3B	3E
312	41	44	47	4A	4D	50	53	56	59	5C	5F	62	65	68	6B	6E	71	74	77	7A	7D	80	83	86	89	8C	8F	92	95	98	9B	9E	A1	A4	A7	AA	AD	B0	B3	B6	B9	BC	BF	C2	C5	C8	CB	CE	D1	D4	D7	DA
364	DD	E0	E3	E6	E9	EC	EF	F2	F5	F8	FB	FE	01	04	07	0A	0D	10	13	16	19	1C	1F	22	25	28	2B	2E	31	34	37	3A	3D	40	43	46	49	4C	4F	52	55	58	5B	5E	61	64	67	6A	6D	70	73	76
416	79	7C	7F	82	85	88	8B	8E	91	94	97	9A	9D	A0	A3	A6	A9	AC	AF	B2	B5	B8	BB	BE	C1	C4	C7	CA	CD	D0	D3	D6	D9	DC	DF	E2	E5	E8	EB	EE	F1	F4	F7	FA	FD	00	03	06	09	0C	0F	12
468	15	18	1B	1E	21	24	27	2A	2D	30	33	36	39	3C	3F	42	45	48	4B	4E	51	54	57	5A	5D	60	63	66	69	6C	6F	72	75	78	7B	7E	81	84	87	8A	8D	90	93	96	99	9C	9F	A2	A5	A8	AB	AE
520	B1	B4	B7	BA	BD	C0	C3	C6	C9	CC	CF	D2	D5	D8	DB	DE	E1	E4	E7	EA	ED	F0	F3	F6	F9	FC	FF	02	05	08	0B	0E	11	14	17	1A	1D	20	23	26	29	2C	2F	32	35	38	3B	3E	41	44	47	4A
572	4D	50	53	56	59	5C	5F	62	65	68	6B	6E	71	74	77	7A	7D	80	83	86	89	8C	8F	92	95	98	9B	9E	A1	A4	A7	AA	AD	B0	B3	B6	B9	BC	BF	C2	C5	C8	CB	CE	D1	D4	D7	DA	DD	E0	E3	E6
624	E9	EC	EF	F2	F5	F8	FB	FE	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
676	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
728	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
780	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
832	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
884	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
936	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
988	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
1040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
1092	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
1144	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
1196	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
1248	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	01	04	07	0A	0D	10	13	16	19	1C	1F	22	25	28	2B	2E	31	34	37	3A
1300	3D	40	43	46	49	4C	4F	52	55	58	5B	5E	61	64	67	6A	6D	70	73	76	79	7C	7F	82	85	88	8B	8E	91	94	97	9A	9D	A0	A3	A6	A9	AC	AF	B2	B5	B8	BB	BE	C1	C4	C7	CA	CD	D0	D3	D6
1352	D9	DC	DF	E2	E5	E8	EB	EE	F1	F4	F7	FA	FD	00	03	06	09	0C	0F	12	15	18	1B	1E	21	24	27	2A	2D	30	33	36	39	3C	3F	42	45	48	4B	4E	51	54	57	5A	5D	60	63	66	69	6C	6F	72
1404	75	78	7B	7E	81	84	87	8A	8D	90	93	96	99	9C	9F	A2	A5	A8	AB	AE	B1	B4	B7	BA	BD	C0	C3	C6	C9	CC	CF	D2	D5	D8	DB	DE	E1	E4	E7	EA	ED	F0	F3	F6	F9	FC	FF	02	05	08	0B	0E
1456	11	14	17	1A	1D	20	23	26	29	2C	2F	32	35	38	3B	3E	41	44	47	4A	4D	50	53	56	59	5C	5F	62	65	68	6B	6E	71	74	77	7A	7D	80	83	86	89	8C	8F	92	95	98	9B	9E	A1	A4	A7	AA
1508	AD	B0	B3	B6	B9	BC	BF	C2	C5	C8	CB	CE	D1	D4	D7	DA	DD	E0	E3	E6	E9	EC	EF	F2	F5	F8	FB	FE	01	04	07	0A	0D	10	13	16	19	1C	1F	22	25	28	2B	2E	31	34	37	3A	3D	40	43	46
1560	49	4C	4F	52	55	58	5B	5E	61	64	67	6A	6D	70	73	76	79	7C	7F	82	85	88	8B	8E	91	94	97	9A	9D	A0	A3	A6	A9	AC	AF	B2	B5	B8	BB	BE	C1	C4	C7	CA	CD	D0	D3	D6	D9	DC	DF	E2
1612	E5	E8	EB	EE	F1	F4	F7	FA	FD	00	03	06	09	0C	0F	12	15	18	1B	1E	21	24	27	2A	2D	30	33	36	39	3C	3F	42	45	48	4B	4E	51	54	57	5A	5D	60	63	66	69	6C	6F	72	75	78	7B	7E
1664	81	84	87	8A	8D	90	93	96	99	9C	9F	A2	A5	A8	AB	AE	B1	B4	B7	BA	BD	C0	C3	C6	C9	CC	CF	D2	D5	D8	DB	DE	E1	E4	E7	EA	ED	F0	F3	F6	F9	FC	FF	02	05	08	0B	0E	11	14	17	1A
1716	1D	20	23	26	29	2C	2F	32	35	38	3B	3E	41	44	47	4A	4D	50	53	56	59	5C	5F	62	65	68	6B	6E	71	74	77	7A	7D	80	83	86	89	8C	8F	92	95	98	9B	9E	A1	A4	A7	AA	AD	B0	B3	B6
1768	B9	BC	BF	C2	C5	C8	CB	CE	D1	D4	D7	DA	DD	E0	E3	E6	E9	EC	EF	F2	F5	F8	FB	FE	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
1820	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
1872	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
1924	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
1976	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2028	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Figure 6.6: Memory Map

Chapter 7

Conclusion

A verification IP is developed using UVM to verify a SD/MMC controller IP created by Open-Cores.org. SystemVerilog with UVM classes is studied to create a completely reusable and reconfigurable test environment. This environment thoroughly randomizes the data and address to be written to the SD card. Protocol checking is done by asserting properties. This VIP enables the user to have control on the stimulus generator and monitor to validate the operation of the DUT and debug it incase it fails.

7.1 Future Work

The DUT and the verification environment presented in this paper has a tremendous scope for improvement. Some of the ideas are presented below:

- An interrupt controller for the DUT can be developed to enable the micro-controller to directly read the interrupt register using wishbone bus. This interrupt pin/register would also make the verification job easier. This was one of the goals set before this projects and not achieved.

- The verification environment can be modified to not have hard time constraints to wait for the read and write operations to be finished. Instead of having events across monitor and driver - which is not an ideal - the interrupt pin can be polled.

- The four tasks read_from_fifo, write_to_fifo, read_from_sd_card and write_to_sd_card can be randomized in the sequencer to have completely randomized test sequences for the DUT. This would also reveal corner cases and the behavior of the DUT when it encounters them.

- Functional Coverage can be improved by creating a wide range of stimulus vectors.

- SD card model can be improved to act as an SD as well as MMC card. So that the test sequences can further be randomized. In this environment, the DUT sends both the initialization procedures for SD and MMC card and the model responds the SD card procedure which tells the DUT that the card detected was an SD card.

- The test environment can be further parameterized to make it more reusable.

References

[1] T. Martin, *The Insider's Guide To The NXP LPC2300/2400 based Microcontrollers*, M. Beach, Ed. Hitex (UK) Ltd., 2007.

[2] V. S. Rashmi, G. Somayaji, and S. Bhamidipathi, "A Methodology To Reuse Random Ip Stimuli In An Soc Functional Verification Environment," in *2015 19th International Symposium on VLSI Design and Test*, June 2015, pp. 1–5.

[3] P. J. Ma, Y. Jiang, K. Li, and J. Y. Shi, "Functional Verification Of Network Processor," in *2011 International Conference on Electronics, Communications and Control (ICECC)*, Sept 2011, pp. 1472–1475.

[4] P. Rashinkar, P. Paterson, and L. Singh, *System-on-a-Chip Verification Methodology and Techniques*. Springer US, 2002.

[5] I. Kastelan and Z. Krajacevic, "Synthesizable SystemVerilog Assertions as a Methodology for SoC," in *2009 First IEEE Eastern European Conference on the Engineering of Computer Based Systems*, Sept 2009, pp. 120–127.

[6] S. A. Wadekar, "A RTL Level Verification Method For Soc Designs," in *IEEE International [Systems-on-Chip] SOC Conference, 2003. Proceedings.*, Sept 2003, pp. 29–32.

[7] N. Gupta and C. Harakchand, "Embracing the FPGA Challenge for Processor Design Verification," in *2014 15th International Microprocessor Test and Verification Workshop*, Dec 2014, pp. 39–43.

[8] K. Kayamuro, T. Sasaki, Y. Fukazawa, and T. Kondo, "A Rapid Verification Framework for Developing Multi-core Processor," in *2016 Fourth International Symposium on Computing and Networking (CANDAR)*, Nov 2016, pp. 388–394.

[9] K. Yasufuku, N. Oshiyama, T. Saito, Y. Miyamoto, Y. Nakamura, R. Terauchi, A. Kondo, T. Aoyama, M. Takahashi, Y. Oowaki, and R. Bandai, "A Uhs-ii Sd Card Controller With 240mb/s Write Throughput And 260mb/s Read Throughput," in *2014 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, Nov 2014, pp. 29–32.

[10] A. El-Yamany, "Echoing The "generality Concept" Through The Bus Functional Model Architecture In Universal Verification Environments," in *2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, Dec 2016, pp. 77–80.

[11] Y. Yang, Y. Yang, L. Niu, H. Wang, and B. Liu, "Hardware System Design Of Sd Card Reader And Image Processor On Fpga," in *2011 IEEE International Conference on Information and Automation*, June 2011, pp. 577–580.

[12] Y. Yang, Y. Yang, T. Yu, and Y. Zheng, "Software Design Of Sd Card Reader And Image Processor Based On Fpga," in *2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC)*, Aug 2011, pp. 1864–1867.

[13] R. Sethulekshmi, S. Jazir, R. A. Rahiman, R. Karthik, S. Abdulla M, and S. Sree Swathy, "Verification Of A Risc Processor Ip Core Using Systemverilog," in *2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, March 2016, pp. 1490–1493.

[14] P. Zhou, T. Wang, X. Wang, and Y. Wang, "Hardware Implementation Of A Low Power Sd Card Controller," in *2014 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC)*, Aug 2014, pp. 158–161.

[15] C. E. Cummings and T. Fitzpatrick, "OVM & UVM Techniques for Terminating Tests," 2011. [Online]. Available: http://www.sunburst-design.com/papers/ CummingsDVCon2011_UVM_TerminationTechniques.pdf

[16] Y. N. Yun, J. B. Kim, N. D. Kim, and B. Min, "Beyond UVM for practical SoC verification," in *2011 International SoC Design Conference*, Nov 2011, pp. 158–162.

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

[18] *Universal Verification Methodology (UVM) 1.2 User's Guide - Accellera*.

[19] R. Salemi, *The UVM Primer*. Boston Light Press, October 29, 2013.

[20] S. Sarkar, G. S. Chanclar, and S. Shinde, "Effective IP reuse for high quality SOC design," in *Proceedings 2005 IEEE International SOC Conference*, Sept 2005, pp. 217–224.

[21] D. Stow, I. Akgun, R. Barnes, P. Gu, and Y. Xie, "Cost Analysis And Cost-driven Ip Reuse Methodology For Soc Design Based On 2.5d/3d Integration," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2016, pp. 1–6.

[22] *SD Host Controller Simplified Specification given by www.sdcard.org*.

[23] C. S. Lin, K. Y. Chen, Y. H. Wang, and L. R. Dung, "A NAND Flash Memory Controller for SD/MMC Flash Memory Card," in *2006 13th IEEE International Conference on Electronics, Circuits and Systems*, Dec 2006, pp. 1284–1287.

[24] A. El-Yamany, S. El-Ashry, and K. Salah, "Coverage Closure Efficient UVM Based Generic Verification Architecture for Flash Memory Controllers," in *2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, Dec 2016, pp. 30–34.

[25] G. Visalli, "UVM-based verification of ECC module for flash memories," in *2017 European Conference on Circuit Theory and Design (ECCTD)*, Sept 2017, pp. 1–4.

[26] C. S. Lin, K. Y. Chen, Y. H. Wang, and L. R. Dung, "A NAND Flash Memory Controller for SD/MMC Flash Memory Card," in *2006 13th IEEE International Conference on Electronics, Circuits and Systems*, Dec 2006, pp. 1284–1287.

[27] O. Elkeelany and V. S. Todakar, "Data Archival To Sd Card Via Hardware Description Language," *IEEE Embedded Systems Letters*, vol. 3, no. 4, pp. 105–108, Dec 2011.

[28] ——, "Data Concentration And Archival To Sd Card Via Hardware Description Language," in *2011 IEEE GLOBECOM Workshops (GC Wkshps)*, Dec 2011, pp. 625–630.

[29] S. Kim, C. Park, and S. Ha, "Architecture Exploration of NAND Flash-based Multimedia Card," in *2008 Design, Automation and Test in Europe*, March 2008, pp. 218–223.

[30] S. Fielding, *spiMaster IP Core Specification*, 2008.

[31] M. Courtnoy, "SoC Verification Growing Up Fast," Breker Verification Systems, Tech. Rep., 2013.

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

[33] G. Sharma, L. Bhargava, and V. Kumar, "Automated Coverage Register Access Technology on UVM Framework for Advanced Verification," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2018, pp. 1–4.

[34] K. Khalifa, "Extendable generic base verification architecture for flash memory controllers based on UVM," in *2017 IEEE 21st International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, April 2017, pp. 584–589.

Appendix I

Source Code

I.1 Agent

```
1  class sd_mmc_agent extends uvm_agent;
2    `uvm_component_utils(sd_mmc_agent)
3
4    uvm_event run_mon;
5    uvm_event run_drv;
6
7    sd_mmc_sequencer    sa_seqr;
8    sd_mmc_driver    sa_drvr;
9    sd_mmc_monitor_before sa_mon;
10
11
12   function new(string name, uvm_component parent);
13     super.new(name, parent);
```

```
14    endfunction: new

15

16    function void build_phase(uvm_phase phase);

17      super.build_phase(phase);

18

19

20

21      sa_seqr   = sd_mmc_sequencer::type_id::create(.name("
            sa_seqr"), .parent(this));

22      sa_drvr   = sd_mmc_driver::type_id::create(.name("sa_drvr")
            , .parent(this));

23      sa_mon  = sd_mmc_monitor_before::type_id::create(.name("
            sa_mon"), .parent(this));

24      // sa_mon_after  = sd_mmc_monitor_after::type_id::create(.
            name("sa_mon_after"), .parent(this));

25      endfunction: build_phase

26

27  function void connect_phase(uvm_phase phase);

28      super.connect_phase(phase);

29

30      sa_drvr.seq_item_port.connect(sa_seqr.seq_item_export);

31      endfunction

32

33  endclass: sd_mmc_agent
```

I.2 Config

```
1 class sd_mmc_configuration extends uvm_object;
2   `uvm_object_utils(sd_mmc_configuration)
3
4   function new(string name = "");
5     super.new(name);
6   endfunction: new
7 endclass: sd_mmc_configuration
```

I.3   Driver

```
1 class sd_mmc_driver extends uvm_driver#( sd_mmc_transaction );
2    'uvm_component_utils( sd_mmc_driver )
3 frame a;
4    virtual sd_mmc_if vif;
5    sd_mmc_transaction sd_tx;
6    uvm_put_port #(frame) put_port_h;
7       integer i;
8     integer j;
9     integer k;
10    reg[31:0] addr;
11    reg[31:0] addr1;
12      bit dir;
13      logic [7:0] respByte;
14      logic [7:0] rxByte;
15    logic [1:0] smSt;
16    reg [7:0] cnt;
17    //integer i;
18   uvm_event run_mon;
19   uvm_event run_drv;
20   uvm_event_pool p1;
21
22    // reg spiSysClk;
23
```

```
24    function new( string name, uvm_component parent );
25        super.new( name, parent );
26    endfunction: new
27
28    function void build_phase( uvm_phase phase );
29        super.build_phase( phase );
30        void'( uvm_resource_db#( virtual sd_mmc_if )::
            read_by_name
31        ( .scope( "ifs" ), .name( "sd_mmc_if" ), .val( vif ) ) )
            ;
32
33      p1=uvm_event_pool::get_global_pool();
34     run_drv=p1.get("run_drv");
35     run_mon=p1.get("run_mon");
36     put_port_h = new("put_port_h",this);
37
38    endfunction: build_phase
39
40    task run_phase( uvm_phase phase );
41        phase.raise_objection(.obj(this));
42
43
44        begin
45
46            vif.rst_i = 0;
```

```
47          @(posedge vif.clk_i);
48           vif.rst_i = 1;
49          @(posedge vif.clk_i);
50           vif.rst_i = 1;
51          @(posedge vif.clk_i);
52           vif.rst_i = 1;
53          @(posedge vif.clk_i);
54           vif.rst_i = 1;
55          @(posedge vif.clk_i);
56           vif.rst_i = 1;
57          @(posedge vif.clk_i);
58           vif.rst_i = 1;
59          @(posedge vif.clk_i);
60           vif.rst_i = 1;
61          @(posedge vif.clk_i);
62           vif.rst_i = 1;
63          @(posedge vif.clk_i);
64           vif.rst_i = 0;
65
66
67          config_dut(vif.data_o);
68          $display("Before Init");
69
70
71          init(vif.data_o);
```

```
72
73
74
75          empty_fifo();
76
77          forever begin
78              seq_item_port.get_next_item(sd_tx);
79              addr1 = sd_tx.address;
80              $display("addr = %x", sd_tx.address);
81              seq_item_port.item_done();
82              wb_block_write(sd_tx.address);
83              rd_cmd_from_sd(vif.data_o);
84
85              put_port_h.put(a);
86
87          end
88
89
90
91
92          $write("Tests done");
93          $stop;
94          end
95
96      phase.drop_objection(.obj(this));
```

```
97      endtask : run_phase
98
99
100
101  task  wb_block_write ( ) ;
102  input  [31:0]  address ;
103
104  begin
105     init ( vif . data_o ) ;
106     empty_fifo ( ) ;
107     set_addr ( addr ) ;
108     for ( i  =  0;  i  <  512;  i  =  i  +  1)
109        begin
110           seq_item_port . get_next_item ( sd_tx ) ;
111           w_to_fifo ( sd_tx . data ) ;
112
113           a [ i ]= sd_tx . data ;
114              seq_item_port . item_done ( ) ;
115        end
116     wr_cmd_to_sd ( vif . data_o ) ;
117  end
118  endtask :  wb_block_write
119
120  task  wb_block_read ( ) ;
121  input  [31:0] address ;
```

```
122
123  begin
124    init ( vif . data_o );
125    set_addr ( addr );
126    for ( i = 0;  i < 512;  i = i + 1)
127      begin
128        r_from_fifo ();
129      end
130    rd_cmd_from_sd ( vif . data_o );
131    run_mon . trigger ();
132    run_drv . wait_trigger ();
133  end
134  endtask :  wb_block_read
135
136
137
138
139  task  wb_read ;
140      input    delay ;
141      integer  delay ;
142
143      input   [7:0]   a ;
144      output  [7:0]  d ;
145
146      begin
```

```
147
148         // wait initial delay
149         repeat(delay) @(posedge vif.clk_i);
150
151         /// assert wishbone signals
152         // #1;
153         vif.address_i  = a;
154         vif.data_i = {8{1'bz}};
155
156         vif.strobe_i  = 1'b1;
157         vif.we_i    = 1'b0;
158
159         @(posedge vif.clk_i);
160
161         // wait for acknowledge from slave
162         while(~vif.ack_o) @(posedge vif.clk_i);
163
164         // negate wishbone signals
165         // #1;
166
167         vif.strobe_i  = 1'bz;
168         vif.address_i  = {8{1'bz}};
169         vif.data_i = {8{1'bz}};
170         vif.we_i    = 1'hz;
171
```

```verilog
172            d     = vif.data_o;
173
174        end
175  endtask
176
177  task wb_write;
178      input   delay;
179      integer delay;
180
181      input [7:0] a;
182      input [7:0] d;
183
184      begin
185
186        // wait initial delay
187        repeat(delay) @(posedge vif.clk_i);
188
189        // assert wishbone signal
190        // #1;
191        vif.address_i  = a;
192        vif.data_i = d;
193  ;
194        vif.strobe_i  = 1'b1;
195        vif.we_i    = 1'b1;
196
```

```verilog
197        @(posedge vif.clk_i);
198
199        // wait for acknowledge from slave
200        while (~vif.ack_o) @(posedge vif.clk_i);
201
202        // negate wishbone signals
203        // #1;
204
205        vif.strobe_i  = 1'bz;
206        vif.address_i  = {8{1'bz}};
207        vif.data_i = {8{1'bz}};
208        vif.we_i    = 1'hz;
209
210
211     end
212   endtask
213
214
215
216
217 task config_dut;
218 input [7:0] dataRead;
219
220
221 begin
```

```
222   /*
223   $write("Testing register read/write\n");
224     wb_write(1, `CTRL_STS_REG_BASE+`SPI_CLK_DEL_REG , 8'h10);
          //b
225   wb_write(1, `CTRL_STS_REG_BASE+`SD_ADDR_7_0_REG , 8'h78); //7
226   wb_write(1, `CTRL_STS_REG_BASE+`SD_ADDR_15_8_REG , 8'h0); //8
227     wb_write(1, `CTRL_STS_REG_BASE+`SD_ADDR_23_16_REG , 8'h0);
          //9
228     wb_write(1, `CTRL_STS_REG_BASE+`SD_ADDR_31_24_REG , 8'h0);
          //a
229     */
230
231
232   $write("Testing SPI bus direct access\n");
233   wb_write(1, `CTRL_STS_REG_BASE+`TRANS_TYPE_REG , {6'b000000,
          `DIRECT_ACCESS}); //2
234   wb_write(1, `CTRL_STS_REG_BASE+`DIRECT_ACCESS_DATA_REG , 8'
          h5f); //6
235     wb_write(1, `CTRL_STS_REG_BASE+`TRANS_CTRL_REG , {7'
          b0000000, `TRANS_START}); //3, 1
236     wb_read(1, `CTRL_STS_REG_BASE+`TRANS_STS_REG , dataRead);
          //4
237     while (dataRead[0] == `TRANS_BUSY)
238       wb_read(1, `CTRL_STS_REG_BASE+`TRANS_STS_REG , dataRead);
            //4
```

```
239
240      // write one byte to spi bus, and wait for complete
241      wb_write(1, 'CTRL_STS_REG_BASE+'DIRECT_ACCESS_DATA_REG , 8'
             haa); // 6
242      wb_write(1, 'CTRL_STS_REG_BASE+'TRANS_CTRL_REG , {7'
             b0000000, 'TRANS_START}); // 3
243      wb_read(1, 'CTRL_STS_REG_BASE+'TRANS_STS_REG , dataRead);
244      while (dataRead[0] == 'TRANS_BUSY)
245        wb_read(1, 'CTRL_STS_REG_BASE+'TRANS_STS_REG , dataRead);
246
247
248 end
249 endtask
250
251 task init;
252 input [7:0]dataRead;
253
254
255
256 begin
257
258   $write("Testing SD init \n");
259   wb_write(1, 'CTRL_STS_REG_BASE+'TRANS_TYPE_REG , {6'b000000,
           'INIT_SD});
```

```
260     wb_write(1, 'CTRL_STS_REG_BASE+'TRANS_CTRL_REG , {7'
            b0000000 , 'TRANS_START});

261     #60000;

262     wb_read(1, 'CTRL_STS_REG_BASE+'TRANS_STS_REG , dataRead);

263     while (dataRead[0] == 'TRANS_BUSY)

264       wb_read(1, 'CTRL_STS_REG_BASE+'TRANS_STS_REG , dataRead);

265     wb_read(1, 'CTRL_STS_REG_BASE+'TRANS_ERROR_REG , dataRead);

266

267     if (dataRead[1:0] == 'INIT_NO_ERROR)

268       $write("SD init test passed\n");

269     else

270       $write("—— ERROR: SD init test failed. Error code = 0x
            %01x\n", dataRead[1:0] );

271

272 end

273 endtask

274

275

276

277 task empty_fifo;

278     wb_write(1, 'TX_FIFO_BASE+'FIFO_CONTROL_REG , 8'h01);

279 endtask

280

281

282 task w_to_fifo;
```

```verilog
283  input [7:0] dataWrite;
284
285  begin
286    wb_write(1, `TX_FIFO_BASE+`FIFO_DATA_REG , dataWrite);
287  end
288  endtask
289
290
291  task r_from_fifo;
292  reg [7:0]dataRead;
293  begin
294    wb_read(1, `RX_FIFO_BASE+`FIFO_DATA_REG , dataRead);
295  end
296  endtask
297
298
299  task set_addr;
300  input [31:0] addr;
301
302  begin
303    wb_write(1, `CTRL_STS_REG_BASE+`SD_ADDR_7_0_REG , addr[7:0]);
         //7
304      wb_write(1, `CTRL_STS_REG_BASE+`SD_ADDR_15_8_REG , addr
           [15:8]); //8
```

```
305     wb_write(1, 'CTRL_STS_REG_BASE+'SD_ADDR_23_16_REG , addr
            [23:16]); //9
306     wb_write(1, 'CTRL_STS_REG_BASE+'SD_ADDR_31_24_REG , addr
            [31:24]); //a
307     end
308 endtask
309
310
311 task wr_cmd_to_sd;
312
313 input [7:0]dataRead;
314
315 begin
316
317     wb_write(1, 'CTRL_STS_REG_BASE+'TRANS_TYPE_REG , {6'b000000
            , 'RW_WRITE_SD_BLOCK});
318     wb_write(1, 'CTRL_STS_REG_BASE+'TRANS_CTRL_REG , {7'
            b0000000 , 'TRANS_START});
319     #100000;
320     #8000000;
321     #8000000;
322     #8000000;
323     #8000000;
324     #8000000;
325     #8000000;
```

```verilog
326        #8000000;
327        wb_read(1, `CTRL_STS_REG_BASE+`TRANS_STS_REG , dataRead);
328
329        if (dataRead[0] == `TRANS_BUSY) begin
330          $write("—— ERROR: SD block write failed to complete\n")
                    ;
331        end
332        else begin
333          wb_read(1, `CTRL_STS_REG_BASE+`TRANS_ERROR_REG , dataRead
                    );
334          if (dataRead[5:4] == `WRITE_NO_ERROR)
335             $write("SD block write passed\n");
336          else
337             $write("—— ERROR: SD block write failed. Error code
                        = 0x%01x\n", dataRead[5:4] );
338      end
339  end
340  endtask
341
342
343  task rd_cmd_from_sd;
344
345  input [7:0]dataRead;
346
347  begin
```

```
348    wb_write(1, 'CTRL_STS_REG_BASE+'TRANS_TYPE_REG , {6'b000000,
          'RW_READ_SD_BLOCK});
349    wb_write(1, 'CTRL_STS_REG_BASE+'TRANS_CTRL_REG , {7'
          b0000000, 'TRANS_START});
350    #100000;
351  #8000000;
352    #8000000;
353    #8000000;
354    #8000000;
355    #8000000;
356    #8000000;
357    #8000000;
358    wb_read(1, 'CTRL_STS_REG_BASE+'TRANS_STS_REG , dataRead);
359
360    if (dataRead[3:2] == 'READ_NO_ERROR) begin
361      $write("SD block read passed\n");
362      for (j=0; j< 16; j=j+1) begin
363        $write("Data 0x%0x = ",j*32);
364        for (i=0; i < 32; i=i+1) begin
365          wb_read(1, 'RX_FIFO_BASE+'FIFO_DATA_REG , dataRead);
366            $write("0x%0x ",dataRead);
367        end
368        $write("\n");
369      end
370    end
```

```
371      else
372         $write("———  ERROR:  SD  block  read  failed.  Error  code = 0x
            %01x\n", dataRead[3:2] );
373
374
375 end
376 endtask
377
378
379 endclass: sd_mmc_driver
```

I.4 Environment

```
1  class sd_mmc_env extends uvm_env;
2    'uvm_component_utils(sd_mmc_env)
3
4    sd_mmc_agent sa_agent;
5    sd_mmc_scoreboard sa_sb;
6    uvm_tlm_fifo #(frame) fifo_h1;
7
8    uvm_tlm_fifo #(frame) fifo_h2;
9    function new(string name, uvm_component parent);
10     super.new(name, parent);
11   endfunction: new
12
13   function void build_phase(uvm_phase phase);
14     super.build_phase(phase);
15     sa_agent  = sd_mmc_agent::type_id::create(.name("sa_agent")
         , .parent(this));
16     sa_sb     = sd_mmc_scoreboard::type_id::create(.name("sa_sb")
         , .parent(this));
17       fifo_h1=new("fifo_h1",this);
18       fifo_h2=new("fifo_h2",this);
19   endfunction: build_phase
20
21   function void connect_phase(uvm_phase phase);
```

```
22        super.connect_phase(phase);

23     sa_agent.sa_drvr.put_port_h.connect(fifo_h1.put_export);

24     sa_agent.sa_mon.put_port_h1.connect(fifo_h2.put_export);

25     sa_sb.get_port_h1.connect(fifo_h1.get_export);

26     sa_sb.get_port_h2.connect(fifo_h2.get_export);

27

28     endfunction: connect_phase

29   endclass: sd_mmc_env
```

I.5 Interface

```systemverilog
1  interface sd_mmc_if ;
2    logic clk_i ;
3    logic rst_i ;
4    logic [7:0] address_i ;
5    logic [7:0] data_i ;
6
7    logic [7:0] data_o ;
8
9    logic strobe_i ;
10   logic we_i ;
11
12   logic ack_o ;
13
14  // SPI logic clock
15   logic spiSysClk ;
16
17  // SPI bus
18   logic spiClkOut ;
19   logic spiDataIn ;
20   logic spiDataOut ;
21   logic spiCS_n ;
22
23
```

```
24      logic scan_in0;
25    logic scan_en;
26    logic test_mode;
27      logic scan_out0;
28
29  //    clocking sd_mmc_cb @ ( posedge clk_i);
30  //        default input #1step output #1ns;
31  //        output data_o, ack_o, scan_out0,  spiClkOut, spiDataOut
      , spiCS_n;
32  //        input   rst_i, address_i, data_i, strobe_i, we_i,
      spiSysClk, spiDataIn, scan_in0, scan_en, test_mode;
33  //    endclocking: master_cb
34
35  //    modport master_mp( input clk_i, rst_i, address_i, data_i,
      strobe_i, we_i, spiSysClk, spiDataIn, scan_in0, scan_en,
      test_mode, output  data_o, ack_o, scan_out0,  spiClkOut,
      spiDataOut, spiCS_n);
36  //    modport master_sync_mp( clocking master_cb );
37
38  //  wb_master_model   wb1    (clk_i,rst_i,address_i,data_i,
      data_o,1'b1,strobe_i,we_i,1'b1,ack_o,1'b0,1'b0);
39
40  endinterface: sd_mmc_if
```

I.6 Monitor

```
1  class sd_mmc_monitor_before extends uvm_monitor;
2    `uvm_component_utils(sd_mmc_monitor_before)
3  frame a;
4  uvm_put_port #(frame) put_port_h1;
5    uvm_event_pool p1;
6    virtual sd_mmc_if vif;
7    uvm_event run_mon;
8    uvm_event run_drv;
9
10   integer i;
11
12   function new(string name, uvm_component parent);
13     super.new(name, parent);
14   endfunction: new
15
16   function void build_phase(uvm_phase phase);
17     super.build_phase(phase);
18
19     void'(uvm_resource_db#(virtual sd_mmc_if)::read_by_name
20       (.scope("ifs"), .name("sd_mmc_if"), .val(vif)));
21
22       p1=uvm_event_pool::get_global_pool();
23     run_drv=p1.get("run_drv");
```

```
24        run_mon=p1.get("run_mon");
25        put_port_h1 = new("put_port_h1",this);
26
27     endfunction: build_phase
28
29     task run_phase(uvm_phase phase);
30         forever begin
31           reg [7:0] dataRead;
32           i = 0;
33            run_mon.wait_trigger();
34           repeat(512) begin
35             $display("Before Init_ monitor");
36             r_from_fifo(dataRead);
37             a[i] = dataRead;
38             i = i + 1;
39
40          end
41          put_port_h1.put(a);
42          run_drv.trigger();
43          end
44
45     endtask: run_phase
46
47
48
```

```
49
50
51
52
53
54  task  r_from_fifo;
55  output  [7:0]dataRead;
56
57  begin
58    wb_read(1 , 'RX_FIFO_BASE+'FIFO_DATA_REG , dataRead);
59  end
60  endtask
61
62
63  task  wb_read;
64      input    delay;
65      integer  delay;
66
67      input   [7:0]   a;
68      output  [7:0]  d;
69
70      begin
71
72        // wait  initial  delay
73        repeat(delay) @(posedge  vif.clk_i);
```

```
74
75          /// assert wishbone signals
76          // #1;
77          vif.address_i  = a;
78          vif.data_i = {8{1'bx}};
79
80          vif.strobe_i  = 1'b1;
81          vif.we_i    = 1'b0;
82
83        @(posedge vif.clk_i);
84
85          // wait for acknowledge from slave
86          while(~vif.ack_o) @(posedge vif.clk_i);
87
88          // negate wishbone signals
89          // #1;
90
91          vif.strobe_i  = 1'bx;
92          vif.address_i  = {8{1'bx}};
93          vif.data_i = {8{1'bx}};
94          vif.we_i    = 1'hx;
95
96          d    = vif.data_o;
97
98      end
```

```
 99   endtask
100
101
102
103
104
105
106
107
108
109
110   endclass: sd_mmc_monitor_before
```

I.7   Package

```systemverilog
1  package sd_mmc_pkg;
2
3    import uvm_pkg::*;
4
5    `include "sd_mmc_sequencer.sv"
6    `include "sd_mmc_monitor.sv"
7    `include "sd_mmc_driver.sv"
8    `include "sd_mmc_agent.sv"
9    `include "sd_mmc_scoreboard.sv"
10   `include "sd_mmc_config.sv"
11   `include "sd_mmc_env.sv"
12   `include "sd_mmc_test.sv"
13
14
15 endpackage: sd_mmc_pkg
```

I.8   Scoreboard

```
1
2
3  class sd_mmc_scoreboard extends uvm_scoreboard;
4    `uvm_component_utils(sd_mmc_scoreboard)
5
6
7    uvm_get_port #(frame) get_port_h1;
8    uvm_get_port #(frame) get_port_h2;
9
10   frame a;
11
12   frame b;
13
14   function new(string name, uvm_component parent);
15     super.new(name, parent);
16     endfunction: new
17
18   function void build_phase(uvm_phase phase);
19     super.build_phase(phase);
20
21     get_port_h1 = new("get_port_h1",this);
22
23     get_port_h2 = new("get_port_h2",this);
```

```
24
25
26    endfunction: build_phase
27
28    task run();
29      forever begin
30      integer i = 0;
31      get_port_h1.get(a);
32      get_port_h2.get(b);
33      for(i=0;i<511;i=i+1)
34      if(a[i]!=b[i+1])
35      $display("Error");
36
37        end
38    endtask: run
39
40 endclass: sd_mmc_scoreboard
```

I.9   Sequencer and Sequence Item

```
1  class sd_mmc_transaction extends uvm_sequence_item;

2

3

4      //rand bit dir;

5

6

7  //   rand dir_e dir;

8    rand bit[31:0] address;

9      rand bit[7:0] data;

10

11

12     rand int create_delay;

13

14     constraint default_delay_values {

15   soft create_delay inside {[0:10]};

16   }

17

18   constraint max_write_address

19   {

20   address < (2^21−513) ; }

21

22

23   function new(string name = "" );
```

```systemverilog
24        super.new(name);
25    endfunction
26
27    `uvm_object_utils_begin(sd_mmc_transaction)
28        // `uvm_field_enum(dir_e, dir, UVM_ALL_ON)
29    //    `uvm_field_int(dir, UVM_ALL_ON)
30        `uvm_field_int(address, UVM_ALL_ON)
31        `uvm_field_int(data, UVM_ALL_ON)
32        `uvm_field_int(create_delay, UVM_ALL_ON)
33    `uvm_object_utils_end
34
35
36 endclass: sd_mmc_transaction
37
38
39 class sd_mmc_init_sequence extends uvm_sequence#(
      sd_mmc_transaction);
40
41
42    `uvm_object_utils(sd_mmc_init_sequence)
43
44    function new(string name = "");
45       super.new(name);
46    endfunction: new
47
```

```systemverilog
48    task  body ( ) ;
49      begin
50      sd_mmc_transaction  sd_tx ;
51      // sd_mmc_transaction :: sd_frame  sd_frame_tx ;
52
53
54        sd_tx = sd_mmc_transaction :: type_id :: create (. name (" sd_tx "
                ) , . contxt ( get_full_name ( ) ) ) ;
55
56        start_item ( sd_tx ) ;
57        assert ( sd_tx . randomize ( ) ) ;
58        finish_item ( sd_tx ) ;
59      end
60
61    endtask :  body
62 endclass :  sd_mmc_init_sequence
63
64
65 typedef  bit  [ 7 : 0 ]  [ 511 : 0 ]  frame ;
66
67 typedef  uvm_sequencer #( sd_mmc_transaction )  sd_mmc_sequencer ;
```

I.10   Top

```
1
2  `include "uvm_macros.svh"
3
4  `include "sd_mmc_pkg.sv"
5  // `include "sd_mmc.v"
6  `include "sd_mmc_if.sv"
7
8  module test;
9     import uvm_pkg::*;
10    import sd_mmc_pkg::*;
11
12    // Interface declaration
13    sd_mmc_if vif();
14
15    // Connects the Interface to the DUT
16    sd_mmc top (
17        vif.scan_in0,
18        vif.scan_en,
19        vif.test_mode,
20        vif.scan_out0,
21        vif.clk_i,
22        vif.rst_i,
23        vif.address_i,
```

```
24        vif.data_i,

25        vif.data_o,

26        vif.strobe_i,

27        vif.we_i,

28        vif.ack_o,

29        // SPI logic clock

30        vif.spiSysClk,

31        //SPI bus

32        vif.spiClkOut,

33        vif.spiDataIn,

34        vif.spiDataOut,

35        vif.spiCS_n

36         );

37

38   spi_sd_model U1(

39        .rstn(vif.rst_i),

40        .sclk(vif.spiClkOut),

41        .mosi(vif.spiDataOut),

42        .miso(vif.spiDataIn),

43        .ncs(vif.spiCS_n)

44   );

45

46

47   initial begin
```

```
48      // Registers the Interface in the configuration block so
            that other
49      // blocks can use it
50      uvm_resource_db#(virtual sd_mmc_if)::set
51        (.scope("ifs"), .name("sd_mmc_if"), .val(vif));
52
53      // Executes the test
54      run_test();
55    end
56
57    // Variable initialization
58    initial begin
59      vif.clk_i <= 1'b0;
60      vif.spiSysClk <= 1'b0;
61    end
62
63    // Clock generation
64    always
65      #20 vif.clk_i = ~vif.clk_i;
66
67    always
68      #10 vif.spiSysClk = ~vif.spiSysClk;
69
70
71 endmodule
```

I.11 Test

```
1
2        class sd_mmc_test extends uvm_test;
3                  `uvm_component_utils(sd_mmc_test)
4
5
6
7              sd_mmc_env sa_env;
8
9              function new(string name = "sd_mmc_test",
                  uvm_component parent = null);
10                     super.new(name, parent);
11           endfunction: new
12
13           function void build_phase(uvm_phase phase);
14                   super.build_phase(phase);
15                   sa_env = sd_mmc_env::type_id::create(.
                          name("sa_env"), .parent(this));
16           endfunction: build_phase
17
18           task run_phase(uvm_phase phase);
19                   sd_mmc_init_sequence sa_seq;
20
21                   phase.raise_objection(.obj(this));
```

```
22                              sa_seq = sd_mmc_init_sequence ::
                                   type_id :: create (. name (" sa_seq
                                   ") ,  . contxt ( get_full_name ()))
                                   ;
23                              assert ( sa_seq . randomize ());
24                              sa_seq . start ( sa_env . sa_agent .
                                   sa_seqr );
25                       phase . drop_objection (. obj ( this ));
26              endtask :  run_phase
27 endclass :  sd_mmc_test
```

I.12   SPI Model

```
 1  //SD Card, SPI mode, Verilog simulation model
 2  //
 3  //Version history:
 4  //1.0 2016.06.13  1st released by tsuhuai.chan@gmail.com
 5  //        Most of the Card information is referenced
 6  //        from Toshiba 2G and 256MB SD card
 7  //
 8
 9  //parsed commands:
10  // CMD0, CMD8, CMD9, CMD10, CMD12, CMD13, CMD16, CMD17, CMD18,
11  // CMD24, CMD25, CMD27, CMD55, CMD51, CMD58, ACMD13, ACMD51
12  //Not parsed command: (still responses based on spec)
13  // CMD1, CMD6, CMD9, CMD10, CMD30, CMD32, CMD33, CMD42, CMD56,
14  // CMD59, ACMD22, ACMD23, ACMD41, ACMD42
15
16  //Memory size of this model should be 2GB, however only 2MB is
17  // implemened to reduce system memory required during
        simulation.
18  // The initial value of all internal memory is word_address+3.
19
20  //Detail command status
21  // 1. card response of ACMD51: not sure
22  // 2. lock/unlock: not implemented
```

```verilog
23  // 3. erase: not implemented
24  // 4. read multiple block: seems verify OK
25  // 5. write single block: seems verify OK
26  // 6. write multiple block: not verified
27  // 7. partial access: not implemented
28  // 8. misalign: no check
29  // 9. SDHC address: not verified
30
31
32  `define UD 1
33  module spi_sd_model (rstn, ncs, sclk, miso, mosi);
34  input rstn;
35  input ncs;
36  input sclk;
37  input mosi;
38  output  miso;
39
40  parameter tNCS    = 1; //0 ~
41  parameter tNCR    = 1; //1 ~ 8
42  parameter tNCX    = 0; //0 ~ 8
43  parameter tNAC    = 1; //from CSD
44  parameter tNWR    = 1; //1 ~
45  parameter tNBR    = 0; //0 ~
46  parameter tNDS    = 0; //0 ~
47  parameter tNEC    = 0; //0 ~
```

```verilog
48  parameter tNRC      = 1; //

49

50  parameter MEM_SIZE  = 2048*1024; // 2M

51  parameter PowerOff  =  0;

52  parameter PowerOn =  1;

53  parameter IDLE      =  2;

54  parameter CmdBit47  =  3;

55  parameter CmdBit46  =  4;

56  parameter CommandIn =  5;

57  parameter CardResponse  =  6;

58  parameter ReadCycle =  7;

59  parameter WriteCycle  =  8;

60  parameter DataResponse  =  9;

61  parameter CsdCidScr =  10;

62  parameter WriteStop =  11;

63  parameter WriteCRC  =  12;

64

65  integer  i  =  0; // counter index

66  integer  j  =  0; // counter index

67  integer  k  =  0; // for MISO (bit count of a byte)

68  integer  m  =  0; // for MOSI (bit count during CMD12)

69

70

71  reg  miso;

72  reg  [7:0]  flash_mem  [0:MEM_SIZE−1];
```

```verilog
73  reg [7:0] token; // captured token during CMD24, CMD25
74  reg [15:0] crc16_in;
75  reg [6:0] crc7_in;
76  reg [7:0] sck_cnt; // 74 sclk after power on
77  reg [31:0] csd_reg;
78  reg [31:0] block_cnt;
79  reg init_done; // must be defined before ocr.v
80  reg [3:0] st; // SD Card internal state
81  reg app_cmd; //
82  reg [7:0] datain;
83  reg [511:0] ascii_command_state;
84  reg [2:0] ist; // initialization stage
85  reg [45:0] cmd_in, serial_in;
86  wire [5:0] cmd_index = cmd_in[45:40];
87  wire [31:0] argument = cmd_in[39:8];
88  wire [6:0] crc = cmd_in[7:1];
89  wire read_single = (cmd_index == 17);
90  wire read_multi = (cmd_index == 18);
91  wire write_single = (cmd_index == 24);
92  wire write_multi = (cmd_index == 25);
93  wire pgm_csd = (cmd_index == 27);
94  wire send_csd = (cmd_index == 9);
95  wire send_cid = (cmd_index == 10);
96  wire send_scr = (cmd_index == 51) && app_cmd;
97  wire read_cmd = read_single | read_multi;
```

```verilog
 98  wire write_cmd = write_single | write_multi;

 99  wire mem_rw = read_cmd | write_cmd;

100  reg [31:0] start_addr;

101  reg [31:0] block_len;

102  reg [7:0] capture_data; // for debugging

103  reg [3:0] VHS; // Input VHS through MOSI

104  reg [7:0] check_pattern; // for CMD8

105  wire [3:0] CARD_VHS = 4'b000; // SD card accept voltage range

106  wire VHS_match = (VHS == CARD_VHS);

107  reg [1:0] multi_st; // for CMD25

108  reg [45:0] serial_in1; // for CMD25

109  wire [5:0] cmd_in1 = serial_in1[45:40]; // for CMD25

110  wire stop_transmission = (cmd_in1 == 12); // for CMD25

111

112  // Do not change the positions of these include files

113  // Also, ocr.v must be included before csd.v

114  wire CCS = 1'b0;

115  wire [31:0] OCR = {init_done, CCS, 5'b0, 1'b0, 6'b111111, 3'
         b000, 12'h000}; // 3.0~3.6V, no S18A

116

117  wire [1:0] DAT_BUS_WIDTH = 2'b00; // 1bit

118  wire SECURE_MODE = 1'b0; // not in secure mode

119  wire [15:0] SD_CARD_TYPE = 16'h0000; // regular SD

120  wire [31:0] SIZE_OF_PROTECTED_AREA = 32'd2048; //

121  // protected area = SIZE_OF_PROTECTED_AREA * MULT * BLOCK_LEN
```

```
122  wire  [7:0]  SPEED_CLASS  =  8'h4; // class  10

123  wire  [7:0]  PERFORMANCE_MOVE  =  8'd100; // 100MB/sec

124  wire  [3:0]  AU_SIZE  =  7; // 1MB

125  wire  [15:0]  ERASE_SIZE  =  16'd100; // Erase  100  AU

126  wire  [5:0]  ERASE_TIMEOUT  =  16'd50; // 50  sec

127  wire  [1:0]  ERASE_OFFSET  =  0; // 0  sec

128

129  wire  [511:0]  SSR  =  {DAT_BUS_WIDTH,  SECURE_MODE,  6'b0,  6'b0,
        SD_CARD_TYPE,  SIZE_OF_PROTECTED_AREA,  SPEED_CLASS,
        PERFORMANCE_MOVE,  AU_SIZE,  4'b0,  ERASE_SIZE,  ERASE_TIMEOUT,
        ERASE_OFFSET,  400'b0};

130

131

132  wire  [7:0]  MID  =  8'd02;

133  wire  [15:0]  OID  =  16'h544D;

134  wire  [39:0]  PNM  =  "SD02G";

135  wire  [7:0]  PRV  =  8'h00;

136  wire  [31:0]  PSN  =  32'h6543a238;

137  wire  [11:0]  MDT  =  {4'd15,8'h12};

138  wire  [6:0]  CID_CRC  =  7'b1100001; // dummy

139  wire  [127:0]  CID  =  {MID,  OID,  PNM,  PRV,  PSN,  4'b0,  MDT,  CID_CRC
        ,  1'b1};

140

141

142
```

```
143

144

145  wire  [1:0]  CSD_VER  =  2'b00; //Ver1.0

146  wire  [7:0]  TAAC  =  {1'b0,  4'd7,  3'd2}; //3.0*100ns

147  wire  [7:0]  NSAC  =  8'd101;

148  wire  [7:0]  TRAN_SPEED  =  8'h32;

149  wire  [3:0]  READ_BL_LEN  =  4'd11; //2^READ_BL_LEN,  2048 bytes

150  wire  READ_BL_PARTIAL  =  1'b1; //always  1  in  SD  card

151  wire  WRITE_BLK_MISALIGN  =  1'b0; //crossing  physical  blocak
         boundaries  is  invalid

152  wire  READ_BLK_MISALIGN  =  1'b0; //crossing  physical  blocak
         boundaries  is  invalid

153  wire  DSR_IMP  =  1'b0; //no  DSR  implemented

154  wire  [11:0]  C_SIZE  =  2047;

155  wire  [2:0]  VDD_R_CURR_MIN  =  3'd1; //1mA

156  wire  [2:0]  VDD_R_CURR_MAX  =  3'd2; //10mA

157  wire  [2:0]  VDD_W_CURR_MIN  =  3'd1; //1mA

158  wire  [2:0]  VDD_W_CURR_MAX  =  3'd2; //10mA

159  wire  [2:0]  C_SIZE_MULT  =  3'd7; //MULT=512

160  wire  ERASE_BLK_EN  =  1'b0; //Erase  in  unit  of  SECTOR_SIZE

161  wire  [6:0]  SECTOR_SIZE  =  7'd127; //128  WRITE BLOCK

162  wire  [6:0]  WP_GRP_SIZE  =  7'd127; //128

163  wire  WP_GRP_ENABLE  =  1'b0; //no  GROUP  WP

164  wire  [2:0]  R2W_FACTOR  =  3'd0;

165  wire  [3:0]  WRITE_BL_LEN  =  READ_BL_LEN;
```

```
166  wire  WRITE_BL_PARTIAL  =  1'b0;//
167  wire  iFILE_FORMAT_GRP  =  1'b0;
168  wire  iCOPY  =  1'b0;
169  wire  iPERM_WRITE_PROTECT  =  1'b0;//DISABLE  PERMENTAL  WRITE
         PROTECT
170  wire  iTMP_WRITE_PROTECT  =  1'b0;//
171  wire  [1:0]  iFILE_FORMAT  =  1'b0;//
172  wire  [6:0]  iCSD_CRC  =  7'b1010001;//dummy
173
174  reg  FILE_FORMAT_GRP;
175  reg  COPY;
176  reg  PERM_WRITE_PROTECT;
177  reg  TMP_WRITE_PROTECT;
178  reg  [1:0]  FILE_FORMAT;
179  reg  [6:0]  CSD_CRC;
180
181  wire  v1sdsc  =  (CSD_VER  ==  0)  &  ~CCS;//Ver  1,  SDSC
182  wire  v2sdsc  =  (CSD_VER  ==  1)  &  ~CCS;//Ver  2,  SDSC
183  wire  v2sdhc  =  (CSD_VER  ==  1)  &  CCS;//Ver  2,  SDHC
184  wire  sdsc  =  ~CCS;
185
186
187  wire  [127:0]  CSD  =  {CSD_VER,  6'b0,  TAAC,  NSAC,  TRAN_SPEED,  12'
         b0101_1011_0101,  READ_BL_LEN,
```

```
188                    READ_BL_PARTIAL ,  WRITE_BLK_MISALIGN ,
                       READ_BLK_MISALIGN ,  DSR_IMP,
189                    2'b0 ,  C_SIZE ,  VDD_R_CURR_MIN,  VDD_R_CURR_MAX,
                       VDD_W_CURR_MIN,
190                    VDD_W_CURR_MAX,  C_SIZE_MULT ,  ERASE_BLK_EN ,
                       SECTOR_SIZE ,
191                    WP_GRP_SIZE ,  WP_GRP_ENABLE,  2'b00 ,  R2W_FACTOR,
                       WRITE_BL_LEN ,
192                    WRITE_BL_PARTIAL ,  5'b0 ,  FILE_FORMAT_GRP,  COPY,
                       PERM_WRITE_PROTECT ,
193                    TMP_WRITE_PROTECT ,  FILE_FORMAT ,  2'b0 ,  CSD_CRC,  1'
                       b1 } ;
194
195
196
197 wire OUT_OF_RANGE = 1'b0 ;
198 wire ADDRESS_ERROR = 1'b0 ;
199 wire BLOCK_LEN_ERROR = 1'b0 ;
200 wire ERASE_SEQ_ERROR = 1'b0 ;
201 wire ERASE_PARAM = 1'b0 ;
202 wire WP_VIOLATION = 1'b0 ;
203 wire CARD_IS_LOCKED = 1'b0 ;
204 wire LOCK_UNLOCK_FAILED = 1'b0 ;
205 wire COM_CRC_ERROR = 1'b0 ;
206 wire ILLEGAL_COMMAND = 1'b0 ;
```

```verilog
207  wire CARD_ECC_FAILED = 1'b0;

208  wire CC_ERROR = 1'b0;

209  wire ERROR = 1'b0;

210  wire CSD_OVERWRITE = 1'b0;

211  wire WP_ERASE_SKIP = 1'b0;

212  wire CARD_ECC_DISABLE = 1'b0;

213  wire ERASE_RESET = 1'b0;

214  wire [3:0] CURRENT_ST = 1; // ready

215  wire READY_FOR_DATA = 1'b1;

216  wire APP_CMD = 1'b0;

217  wire AKE_SEQ_ERROR = 1'b0;

218  wire IN_IDLE_ST = (CURRENT_ST == 4'b1);

219

220  wire [15:0] CSR = {OUT_OF_RANGE, ADDRESS_ERROR, BLOCK_LEN_ERROR
         , ERASE_SEQ_ERROR,

221                        ERASE_PARAM, WP_VIOLATION, CARD_IS_LOCKED,
                              LOCK_UNLOCK_FAILED,

222                        COM_CRC_ERROR, ILLEGAL_COMMAND,
                              CARD_ECC_FAILED, CC_ERROR,

223                        ERROR, 2'b0, CSD_OVERWRITE, WP_ERASE_SKIP,
                              CARD_ECC_DISABLE,

224                        ERASE_RESET, CURRENT_ST, READY_FOR_DATA, 2'
                              b0, APP_CMD, 1'b0,

225                        AKE_SEQ_ERROR, 3'b0};

226
```

```
227  wire [3:0] SCR_STRUCTURE = 4'd0; // Ver1.0
228  wire [3:0] SD_SPEC = 4'd2; // Ver2.0 or 3.0
229  wire DATA_STAT_AFTER_ERASE = 1'b1;
230  wire [2:0] SD_SECURITY = 3'd4; // Ver3.00
231  wire [3:0] SD_BUS_WIDTHS = 4'b0001; ///1 bit
232  wire SD_SPEC3 = 1'b1; // Ver3.0
233  wire [13:0] CMD_SUPPORT = 14'b0; //
234  wire [63:0] SCR = {SCR_STRUCTURE, SD_SPEC,
         DATA_STAT_AFTER_ERASE, SD_SECURITY, SD_BUS_WIDTHS, SD_SPEC3,
         13'b0, CMD_SUPPORT, 32'b0};
235
236
237
238  //
239  task R1;
240  input [7:0] data;
241  begin
242     // $display("   SD R1: 0x%2h at %0t ns", data, $realtime);
243     k = 0;
244     while (k < 8) begin
245        @(negedge sclk) miso = data[7-k];
246        k = k + 1;
247     end
248  end
249  endtask
```

```verilog
250
251 task R1b;
252 input [7:0] data;
253 begin
254   // $display ("   SD R1B: 0x%2h at %0t ns",data, $realtime);
255   k = 0;
256   while (k < 8) begin
257     @(negedge sclk) miso = data[7-k];
258     k = k + 1;
259   end
260 end
261 endtask
262
263 task R2;
264 input [15:0] data;
265 begin
266   // $display ("   SD R2: 0x%2h at %0t ns",data, $realtime);
267   k = 0;
268   while (k < 16) begin
269     @(negedge sclk) miso = data[15-k];
270     k = k + 1;
271   end
272 end
273 endtask
274
```

```verilog
275  task  R3;
276  input  [39:0]  data;
277  begin
278    // $display("   SD R3: 0x%10h  at  %0t  ns",data,  $realtime);
279    for  (k =0;  k <  40;  k = k + 1)  begin
280        @(negedge  sclk)  ;
281      miso  =  data[39 − k];
282      end
283  end
284  endtask
285
286  task  R7;
287  input  [39:0]  data;
288  begin
289    // $display("   SD R7: 0x%10h  at  %0t  ns",data ,  $realtime);
290    k  =  0;
291    while  (k <  40)  begin
292        @(negedge  sclk)  miso  =  data[39−k];
293        k  = k  +  1;
294      end
295  end
296  endtask
297
298  task  DataOut;
299  input  [7:0]  data;
```

```verilog
300  begin
301    // $display("   SD DataOut 0x%2H at %0t ns",data, $realtime);
302    k = 0;
303    while (k < 8) begin
304       @(negedge sclk) miso = data[7 - k];
305       k = k + 1;
306    end
307  end
308  endtask
309
310  task DataIn;
311  begin
312    for (k = 7; k >= 0; k = k - 1) begin
313    // $display("capture_data = %d", capture_data);
314       @(posedge sclk) capture_data[k] = mosi;
315    end
316    // $display("   SD DataIn: %2h at %0t ns",capture_data,
            $realtime);
317  end
318  endtask
319
320  always @(*) begin
321    if (pgm_csd) csd_reg = argument;
322  end
323
```

```verilog
324 task CRCOut;
325 input [15:0] data;
326 begin
327    // $display("   SD CRC Out 0x%4H at %0t ns",data, $realtime);
328    k = 0;
329    while (k < 16) begin
330       @(negedge sclk) miso = data[15 - k];
331       k = k + 1;
332    end
333 end
334 endtask
335
336 task TokenOut;
337 input [7:0] data;
338 begin
339    // $display("   SD TokenOut 0x%2H at %0t ns",data, $realtime)
         ;
340    k = 0;
341    while (k < 8) begin
342       @(negedge sclk) miso = data[7 - k];
343       k = k + 1;
344    end
345 end
346 endtask
347
```

```verilog
348
349 always @(*) begin
350     if (~pgm_csd) begin
351         FILE_FORMAT_GRP = iFILE_FORMAT_GRP;
352         COPY = iCOPY;
353         PERM_WRITE_PROTECT = iPERM_WRITE_PROTECT ;
354         TMP_WRITE_PROTECT = iTMP_WRITE_PROTECT;
355         FILE_FORMAT = iFILE_FORMAT;
356         CSD_CRC = iCSD_CRC;
357     end
358     else begin
359         FILE_FORMAT_GRP = argument[15];
360         COPY = argument[14];
361         PERM_WRITE_PROTECT = argument[13];
362         TMP_WRITE_PROTECT = argument[12];
363         FILE_FORMAT = argument[11:10];
364         CSD_CRC = argument[7:1];
365     end
366 end
367
368
369 always @(*) begin
370     if (~rstn) app_cmd = 1'b0;
371     else if (cmd_index == 55 && st == IDLE) app_cmd = 1;
372     else if (cmd_index != 55 && st == IDLE) app_cmd = 0;
```

```verilog
373  end
374
375  always @(*) begin
376      if (sdsc && mem_rw) start_addr = argument;
377      else if (v2sdhc && mem_rw) start_addr = argument * block_len
             ;
378  end
379
380  /* always @(*) begin
381      if (v2sdhc) block_len = 512;
382      else if (sdsc && cmd_index == 0) block_len = (READ_BL_LEN ==
             9) ? 512 : (READ_BL_LEN == 10) ? 1024 : 2048;
383      else if (sdsc && cmd_index == 16) block_len = argument
             [31:0];
384  end
385  */
386  always @(*) begin
387      if (cmd_index == 8) VHS = argument[11:8];
388      if (cmd_index == 8) check_pattern = argument[7:0];
389  end
390
391  always @(*) begin
392      if (ist == 0 && cmd_index == 0) begin
393          $display("iCMD0 at %0t ns", $realtime);
394          ist <= 1;
```

```verilog
395        end
396
397
398      // if ( ist  ==  1 && cmd_index  ==  8) begin
399       if ( ist  ==  1)  begin
400          // $display ("iCMD8 at  %0t  ns", $realtime );
401          ist  <=  2;
402       end
403
404       if ( ist  ==  2 && cmd_index  ==  0)
405          begin
406             $display ( " .. " );
407
408          end
409      // if ( ist  ==  2 && cmd_index  ==  58) begin
410
411       if ( ist  ==  2)  begin
412          // $display ("iCMD58 at  %0t  ns", $realtime );
413          ist  <=  3;
414       end
415
416      // if ( ist  ==  3 && cmd_index  ==  55) begin
417       if ( ist  ==  3)  begin
418          // $display ("iCMD55 at  %0t  ns", $realtime );
419          ist  <=  4;
```

```
420        end
421        if (ist == 4 && cmd_index == 1) begin
422            $display("iCMD1 at %0t ns",$realtime);
423            ist <= 5;
424        end
425        if (ist == 5 && cmd_index == 58 && CSD_VER == 1) begin
426            $display("iCMD58 at %0t ns",$realtime);
427            ist <= 6;
428        end
429        else if (ist == 5 && CSD_VER == 0) ist <= 6;
430        if (ist == 6 && st == IDLE) begin
431            $display("Init Done at %0t ns",$realtime);
432            if (v2sdhc) $display("Ver 2, SDHC detected");
433            else if (v2sdsc) $display("Ver 2, SDSC detected");
434            else if (v1sdsc) $display("Ver 1, SDSC detected");
435            init_done = 1;
436            ist <= 7;
437        end
438 end
439
440 always @(*) begin
441    if (st == ReadCycle) begin
442        // $display("readcycle");
443        case (multi_st)
444    0:
```

```
445        begin
446            @(posedge sclk) if (~ncs && ~mosi) multi_st = 1; else
                   multi_st = 0;
447        end
448    1:
449        begin
450            @(posedge sclk); if (mosi) multi_st = 2; else multi_st =
                   1;
451        end
452    2:
453        begin
454           m = 0;
455                      while (m < 46) begin
456               @(posedge sclk) serial_in1[45-m] = mosi;
457                          #1 m = m + 1;
458            end
459              multi_st = 0;
460        end
461          endcase
462      end
463  end
464
465  always @(*) begin
466  // $monitor("cmd_in = %d", cmd_in[45:40]);
467      case (st)
```

```
468    PowerOff:
469      begin
470        @(posedge rstn) st <= PowerOn;
471      end
472    PowerOn:
473      begin
474        if (sck_cnt < 73) begin
475          @(posedge sclk) sck_cnt = sck_cnt + 1;
476        end
477                if (sck_cnt == 73) st <= IDLE; else st <=
                        PowerOn;
478      end
479    IDLE:
480      begin
481        @(posedge sclk) if (~ncs && ~mosi) st <= CmdBit46; else
              st <= IDLE;
482      end
483    CmdBit46:
484      begin
485        @(posedge sclk); if (mosi) st <= CommandIn; else st <=
              CmdBit46;
486      end
487    CommandIn: // capture command input -> NCR
488      begin
489        for (i = 0; i < 46; i = i + 1)  begin
```

```
490              @(posedge sclk) ; serial_in[45−i] = mosi;
491         end
492                   cmd_in = serial_in;
493                   //$monitor("cmd_in = %d", serial_in[45:40]);
494         repeat (tNCR*8) @(posedge sclk);
495         st <= CardResponse;
496      end
497  CardResponse: //CardResponse −> delay
498      begin
499         if (~app_cmd) begin
500
501            case (cmd_index)
502            6'd0: R1(8'b0000_0001);
503            6'd41: R1(8'b0);
504            6'd17,
505            6'd1,
506            6'd6,
507            6'd16,
508            6'd18,
509            6'd24,
510            6'd25,
511            6'd27,
512            6'd30,
513            6'd32,
514            6'd33,
```

```
515             6'd42,
516             6'd55,
517             6'd56,
518             6'd59:   R1(8'b0110_1010);
519                      6'd9,
520                      6'd10:      if (init_done) R1(8'b1010_0001)
                              ; else R1(0000_0100);
521             6'd12,
522             6'd28,
523             6'd29,
524             6'd38:   R1b(8'b0011_1010);
525             6'd8:    if (VHS_match) begin
526                                      $display("   VHS match");
527                                      R7({8'h01 | (VHS_match ? 8'
                                          h04 : 8'h00), 20'h00000,
                                          VHS, check_pattern});
528                                  end
529                                  else begin
530                                      $display("   VHS not match")
                                          ;
531                                      R7({8'h01 | (VHS_match ? 8'
                                          h04 : 8'h00), 20'h00000,
                                          4'b0, check_pattern});
532                                  end
```

```
533            6'd13:    R2({1'b0,  OUT_OF_RANGE,  ADDRESS_ERROR,

                   ERASE_SEQ_ERROR,  COM_CRC_ERROR,

534                                    ILLEGAL_COMMAND,

                                        ERASE_RESET,  IN_IDLE_ST,

                                        OUT_OF_RANGE  |

                                        CSD_OVERWRITE,

535                                    ERASE_PARAM,  WP_VIOLATION,

                                        CARD_ECC_FAILED,  CC_ERROR

                                        ,  ERROR,

536                                    WP_ERASE_SKIP  |

                                        LOCK_UNLOCK_FAILED,

                                        CARD_IS_LOCKED});

537            6'd58:    R3({8'b0000_0000,  OCR});

538                    default:    R1(8'b0000_0100); // illegal

                       command

539          endcase

540        end

541        else if (~read_multi) begin

542            case (cmd_index)

543            6'd22,

544            5'd23,

545            6'd41,

546            6'd42,

547                    6'd51:    R1(8'b0000_0000);
```

```
548                 6'd13:   R2({1'b0, OUT_OF_RANGE, ADDRESS_ERROR,
                        ERASE_SEQ_ERROR, COM_CRC_ERROR,
549                                         ILLEGAL_COMMAND, ERASE_RESET
                                                , IN_IDLE_ST, OUT_OF_RANGE
                                                | CSD_OVERWRITE,
550                                         ERASE_PARAM, WP_VIOLATION,
                                                CARD_ECC_FAILED, CC_ERROR,
                                                ERROR,
551                                         WP_ERASE_SKIP |
                                                LOCK_UNLOCK_FAILED,
                                                CARD_IS_LOCKED});
552                     default:   R1(8'b0000_0100);// illegal
                                command
553             endcase
554         end
555
556                 @(posedge sclk);
557
558             // $display("read_cmd = %b", read_cmd);
559         if (read_cmd && init_done /*&& ~stop_transmission*/)
                begin
560
561             // $display("just after");
562                         miso = 1;
563                         repeat (tNAC*8) @(posedge sclk);
```

```verilog
564                              st <= ReadCycle;
565                          end
566          else if (read_cmd && init_done && stop_transmission)
                begin
567                              miso = 1;
568                              repeat (tNEC*8) @(posedge sclk);
569                              st <= IDLE;
570                          end
571          else if ((send_csd || send_cid || send_scr) && init_done
                ) begin
572                              miso = 1;
573                              repeat (tNCX*8) @(posedge sclk);
574                              st <= CsdCidScr;
575                          end
576          else if (write_cmd && init_done) begin
577                              miso = 1;
578                              repeat (tNWR*8) @(posedge sclk);
579                              st <= WriteCycle;
580                          end
581          else begin
582                              repeat (tNEC*8) @(posedge sclk);
583                              st <= IDLE;
584                          end
585      end
586    CsdCidScr:
```

```
587        begin
588            if (send_csd) begin
589                DataOut(CSD[127:120]);
590                DataOut(CSD[119:112]);
591                DataOut(CSD[111:104]);
592                DataOut(CSD[103:96]);
593                DataOut(CSD[95:88]);
594                DataOut(CSD[87:80]);
595                DataOut(CSD[79:72]);
596                DataOut(CSD[71:64]);
597                DataOut(CSD[63:56]);
598                DataOut(CSD[55:48]);
599                DataOut(CSD[47:40]);
600                DataOut(CSD[39:32]);
601                DataOut(CSD[31:24]);
602                DataOut(CSD[23:16]);
603                DataOut(CSD[15:8]);
604                DataOut(CSD[7:0]);
605                        end
606            else if (send_cid) begin
607                DataOut(CID[127:120]);
608                DataOut(CID[119:112]);
609                DataOut(CID[111:104]);
610                DataOut(CID[103:96]);
611                DataOut(CID[95:88]);
```

```
612              DataOut(CID[87:80]);
613              DataOut(CID[79:72]);
614              DataOut(CID[71:64]);
615              DataOut(CID[63:56]);
616              DataOut(CID[55:48]);
617              DataOut(CID[47:40]);
618              DataOut(CID[39:32]);
619              DataOut(CID[31:24]);
620              DataOut(CID[23:16]);
621              DataOut(CID[15:8]);
622              DataOut(CID[7:0]);
623                      end
624          else if (send_scr) begin
625              DataOut(SCR[63:56]);
626              DataOut(SCR[55:48]);
627              DataOut(SCR[47:40]);
628              DataOut(SCR[39:32]);
629              DataOut(SCR[31:24]);
630              DataOut(SCR[23:16]);
631              DataOut(SCR[15:8]);
632              DataOut(SCR[7:0]);
633                        end
634                        @(posedge sclk);
635          repeat (tNEC*8) @(posedge sclk);
636          st <= IDLE;
```

```
637        end
638    ReadCycle: // Start  Token  -> Data  -> CRC(stucked  at  16'hAAAA)
           -> NEC( or  NAC)
639    begin
640
641        if (read_single) begin
642
643                    TokenOut(8'hFE); // Start  Token
644            for (i = 0; i < 512; i = i + 1) begin
645                DataOut(flash_mem[start_addr+i]);
646            end
647                    CRCOut(16'haaaa); //CRC[15:0]
648            @(posedge  sclk);
649            repeat  (tNEC*8) @(negedge  sclk);
650            st  <= IDLE;
651        end
652
653
654
655        /*
656
657
658
659
660
```

```
661
662
663            EDITING  j < block_len
664
665
666
667
668
669
670
671
672
673
674        */
675
676
677        else if (read_multi) begin:loop_1
678            for (j = 0; j > 0 ; j   = j + 1) begin
679                          TokenOut(8'hFE); // Start Token
680                          i = 0;
681                          while (i < block_len) begin
682                DataOut(flash_mem[start_addr+i+block_len*j]);
683                            i = i + 1;
684                          end
685                          CRCOut(16'haaaa);
```

```
686                                    if (stop_transmission) begin //check
                                          stop_tx at end of each data block?
687                                    repeat (tNEC*8) @(posedge sclk);
688                                    $display("STOP transmission");
689                                    @(posedge sclk) begin
690                                         R1(8'b0000_0000);
691                                         repeat (tNEC*8) @(posedge
                                               sclk);
692                                         st <= IDLE;
693                                         disable loop_1;
694                                       end
695                                     end
696                                     else repeat (tNAC*8) @(negedge sclk);
697                                 end
698                                 repeat (tNEC*8) @(posedge sclk);
699                                 st <= IDLE;
700             end
701
702       end
703    WriteCycle: // Start Token -> Data
704       begin
705                       i = 0;
706                       while (i < 8) begin
707            @(posedge sclk)    token[7-i] = mosi;
708                         i = i + 1;
```

```
709             end
710             if (token == 8'hfe && write_single) $display ("Single
                    Write Start Token OK");
711             else if (token != 8'hfe && write_single) $display ("
                    Single Write Start Token NG");
712             if (token == 8'hfc && write_multi)  $display ("Multiblock
                     Write Start Token OK");
713             else if ((token != 8'hfc && token != 8'hfd) &&
                    write_multi)  $display ("Multiblock Write Start Token
                    NG");
714             if (token == 8'hfd && write_multi) begin
715                 $display ("Multiblock Write Stop Token");
716                 st <= WriteStop;
717             end
718                         i = 0;
719                         while (i < block_len) begin
720             DataIn;
721             flash_mem[start_addr + i] = capture_data;
722                         i = i + 1;
723                     end
724         st <= WriteCRC;
725     end
726 WriteCRC: // Capture incoming CRC of data
727     begin
728         i = 0;
```

```verilog
729                          while (i < 16) begin
730              @(posedge sclk) crc16_in[15−i] = mosi;
731                           i = i + 1;
732          end
733          st <= DataResponse;
734      end
735  DataResponse: // All clock after data response CRC
736      begin
737          DataOut(8'b00000101);
738                      @(negedge sclk); miso = 0;
739          repeat (tNEC∗8) @(negedge sclk);
740
741          repeat  (tNDS∗8) @(negedge sclk);
742          miso = 1'bz;
743          @(negedge sclk);
744          miso = 1'b0;
745          repeat (100) @(negedge sclk);
746          miso = 1;
747          @(negedge sclk);
748          miso = 1;
749                      repeat (5) @(posedge sclk);
750          if (write_single) st <= IDLE;
751          else if (write_multi) st <= WriteCycle;
752      end
753  WriteStop:
```

```
754          begin
755              repeat (tNBR*8) @(posedge sclk);
756              miso = 0;
757              repeat (tNEC*8) @(posedge sclk);
758              repeat (tNDS*8) @(posedge sclk) miso = 1'bz;
759              @(posedge sclk) miso = 1'b0;
760              #1000000; //1ms processing time for each block
                         programming
761              @(posedge sclk) miso = 1'b1;
762              repeat (tNEC*8) @(posedge sclk);
763                         @(posedge sclk);
764              st <= IDLE;
765          end
766      default: st <= IDLE;
767      endcase
768  end
769
770  always @(st) begin
771      case (st)
772      PowerOff: ascii_command_state = "PowerOff";
773      PowerOn:  ascii_command_state = "PowerOn";
774      IDLE:     ascii_command_state = "IDLE";
775      CmdBit47: ascii_command_state = "CmdBit47";
776      CmdBit46: ascii_command_state = "CmdBit46";
777      CommandIn: ascii_command_state = "CommandIn";
```

```verilog
778        CardResponse : ascii_command_state = "CardResponse";
779        ReadCycle :   ascii_command_state = "ReadCycle";
780        WriteCycle : ascii_command_state = "WriteCycle";
781        DataResponse : ascii_command_state = "DataResponse";
782        CsdCidScr :   ascii_command_state = "CsdCidScr";
783        WriteStop :   ascii_command_state = "WriteStop";
784        WriteCRC : ascii_command_state = "WriteCRC";
785        default :     ascii_command_state = "ERROR";
786      endcase
787  end
788
789  initial
790      begin
791          sck_cnt = 0;
792          cmd_in = 46'h3fffffffffff;
793          serial_in = 46'h0;
794          crc16_in = 16'h0;
795          crc7_in = 7'h0;
796          token = 8'h0;
797          st <= PowerOff;
798          miso = 1'b1;
799          init_done = 0;
800          ist = 0;
801          capture_data = 8'h0;
802          start_addr = 32'h0;
```

```
803           VHS = 4'h0;
804           serial_in1 = 46'h0;
805           multi_st = 0;
806           block_len = 512;
807           for (i = 0; i < MEM_SIZE − 1; i = i + 1) begin
808               flash_mem[i] = 0;
809           end
810       end
811
812 endmodule
```

I.13   Wishbone Model

```
1  //
    //////////////////////////////////////////////////////////////////////

2  // //


      ////
3  // //    wb_master_model.v
                                                        ////
4  // //


      ////
5  // //    This  file  is  part  of  the  SPI  IP  core  project
                         ////
6  // //    http://www.opencores.org/projects/spi/
                               ////
7  // //


      ////
8  // //    Author(s):
                                                  ////
9  // //        - Simon Srot (simons@opencores.org)
                          ////
```

```
10   ////


        ////
11   ////   Based on:
                                                         ////
12   ////        −  i2c / bench / verilog / wb_master_model . v
                              ////
13   ////          Copyright (C)  2001  Richard  Herveille
                            ////
14   ////


        ////
15   ////   All  additional  information  is  avaliable  in  the  Readme . txt
          ////
16   ////   file .
                                                          ////
17   ////


        ////
18   //
        //////////////////////////////////////////////////////////////////


19   ////


        ////
```

```
20  ////  Copyright (C) 2002 Authors

                                           ////

21  ////


       ////

22  ////  This source file may be used and distributed without

                ////

23  ////  restriction provided that this copyright statement is not

           ////

24  ////  removed from the file and that any derivative work

       contains   ////

25  ////  the original copyright notice and the associated

       disclaimer.  ////

26  ////


       ////

27  ////  This source file is free software; you can redistribute it

           ////

28  ////  and/or modify it under the terms of the GNU Lesser General

           ////

29  ////  Public License as published by the Free Software

       Foundation;  ////

30  ////  either version 2.1 of the License, or (at your option) any

           ////
```

```
31  //// later version.
                                              ////
32  ////

     ////
33  //// This source is distributed in the hope that it will be
          ////
34  //// useful, but WITHOUT ANY WARRANTY; without even the implied
          ////
35  //// warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
          ////
36  //// PURPOSE.   See the GNU Lesser General Public License for
       more ////
37  //// details.
                                              ////
38  ////

     ////
39  //// You should have received a copy of the GNU Lesser General
          ////
40  //// Public License along with this source; if not, download it
          ////
41  //// from http://www.opencores.org/lgpl.shtml
                        ////
```

```verilog
42  // //


      ////
43  //
      ///////////////////////////////////////////////////////////////


44

45

46  module wb_master_model(clk, rst, adr, din, dout, cyc, stb, we,
        sel, ack, err, rty);

47

48    parameter dwidth = 32;

49    parameter awidth = 32;

50

51    input                     clk, rst;

52    output [awidth    -1:0] adr;

53    input  [dwidth    -1:0] din;

54    output [dwidth    -1:0] dout;

55    output                    cyc, stb;

56    output                    we;

57    output [dwidth/8  -1:0] sel;

58    input                     ack, err, rty;

59

60    // Internal signals

61    reg     [awidth    -1:0] adr;
```

```verilog
62    reg     [dwidth    -1:0] dout;
63    reg                      cyc, stb;
64    reg                      we;
65    reg     [dwidth/8  -1:0] sel;
66
67    reg     [dwidth    -1:0] q;
68
69    // Memory Logic
70    initial
71      begin
72        adr  = {awidth{1'bx}};
73        dout = {dwidth{1'bx}};
74        cyc  = 1'b0;
75        stb  = 1'bx;
76        we   = 1'hx;
77        sel  = {dwidth/8{1'bx}};
78        // #1;
79      end
80
81    // Wishbone write cycle
82    task wb_write;
83      input   delay;
84      integer delay;
85
86      input [awidth  -1:0] a;
```

```verilog
87        input [dwidth −1:0] d;

88

89     begin

90

91        // wait initial delay
92        repeat(delay) @(posedge clk);

93

94        // assert wishbone signal
95        // #1;
96        adr  = a;
97        dout = d;
98        cyc  = 1'b1;
99        stb  = 1'b1;
100       we   = 1'b1;
101       sel  = {dwidth/8{1'b1}};
102       @(posedge clk);

103

104       // wait for acknowledge from slave
105       while(~ack) @(posedge clk);

106

107       // negate wishbone signals
108       // #1;
109       cyc  = 1'b0;
110       stb  = 1'bx;
111       adr  = {awidth{1'bx}};
```

```verilog
112            dout = {dwidth{1'bx}};
113            we    = 1'hx;
114            sel   = {dwidth/8{1'bx}};
115
116        end
117    endtask
118
119    // Wishbone read cycle
120    task wb_read;
121        input    delay;
122        integer  delay;
123
124        input  [awidth -1:0]  a;
125        output [dwidth -1:0] d;
126
127        begin
128
129           // wait initial delay
130           repeat(delay) @(posedge clk);
131
132           /// assert wishbone signals
133           // #1;
134           adr  = a;
135           dout = {dwidth{1'bx}};
136           cyc  = 1'b1;
```

```
137          stb   = 1'b1;
138          we    = 1'b0;
139          sel   = {dwidth/8{1'b1}};
140        @(posedge clk);
141
142          // wait for acknowledge from slave
143          while(~ack) @(posedge clk);
144
145          // negate wishbone signals
146          // #1;
147          cyc   = 1'b0;
148          stb   = 1'bx;
149          adr   = {awidth{1'bx}};
150          dout  = {dwidth{1'bx}};
151          we    = 1'hx;
152          sel   = {dwidth/8{1'bx}};
153          d     = din;
154
155        end
156    endtask
157
158    // Wishbone compare cycle (read data from location and
           compare with expected data)
159    task wb_cmp;
160      input    delay;
```

```verilog
161       integer delay;

162

163       input [awidth −1:0] a;

164       input [dwidth −1:0] d_exp;

165

166       begin

167         wb_read (delay, a, q);

168

169       if (d_exp !== q)

170         $display("Data compare error. Received %h, expected %h at
                  time %t", q, d_exp, $time);

171       end

172     endtask

173

174 endmodule
```