ISTANBUL TECHNICAL UNIVERSITY

ELECTRICAL-ELECTRONICS FACULTY

VERIFICATION OF A SERIAL PERIPHERAL INTERFACE INTELLECTUAL PROPERTY BY USING UNIVERSAL VERIFICATION METHODOLOGY

SENIOR DESIGN PROJECT

Berkay TURGAY

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

FEBRUARY 2021

ISTANBUL TECHNICAL UNIVERSITY

ELECTRICAL-ELECTRONICS FACULTY

VERIFICATION OF A SERIAL PERIPHERAL INTERFACE INTELLECTUAL PROPERTY BY USING UNIVERSAL VERIFICATION METHODOLOGY

SENIOR DESIGN PROJECT

Berkay TURGAY

040150097

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

Project Advisor: Prof. Dr. Sıddıka Berna ÖRS YALÇIN

FEBRUARY 2021

İSTANBUL TEKNİK ÜNİVERSİTESİ

ELEKTRİK-ELEKTRONİK FAKÜLTESİ

BİR SERİ ÇEVRESEL ARAYÜZ FİKRİ MÜLKİYETİNİN EVRENSEL DOĞRULAMA METHODU İLE DOĞRULANMASI

LİSANS BİTİRME TASARIM PROJESİ

Berkay TURGAY

040150097

Proje Danışmanı: Prof. Dr. Sıddıka Berna ÖRS YALÇIN

ELEKTRONİK VE HABERLEŞME MÜHENDİSLİĞİ BÖLÜMÜ

ŞUBAT, 2021

We are submitting the Senior Design Project Report entitled as “VERIFICATION OF A SERIAL PERIPHERAL INTERFACE INTELLECTUAL PROPERTY BY USING UNIVERSAL VERIFICATION METHODOLOGY”. The Senior Design Project Report has been prepared as to fulfill the relevant regulations of the Electronics and Communication Engineering Department of Istanbul Technical University. We hereby confirm that we have realized all stages of the Senior Design Project work by ourselves and we have abided by the ethical rules with respect to academic and professional integrity .

Berkay TURGAY ....

040150097

FOREWORD

I would like to thank to my mentor Prof. Dr. Sıddıka Berna Örs Yalçın who helped me to find this project and who allowed me to collaborate with ANKASYS and guided me throughout this project. Secondly, I would like to offer my gratitude to my other mentor Ekrem Şahin from ANKASYS who guided me in all of my mistakes and who helped me finish this project successfully. Without them, I would not finish this project properly. Finally, I would like to thank to my family and friends who supported me my entire life.

February 2021 Berkay TURGAY

TABLE OF CONTENTS

Page

FOREWORD ... **v**
TABLE OF CONTENTS ... **vii**
ABBREVIATIONS ... **ix**
LIST OF FIGURES ... **x**
SUMMARY ... **xiii**
ÖZET ... **xiv**
1. INTRODUCTION ... **1**
2. BACKGROUND INFORMATION ... **2**
2.1 Verification ... 2
2.2 Universal Verification Methodology ... 2
2.2.1 Verification Environment ... 3
2.2.2 UVM Classes ... 4
2.2.3 UVM Phases ... 5
2.2.4 Transaction Level Modeling ... 6
2.2.5 Top Block and Interface ... 7
2.2.6 Components ... 8
2.2.6.1 Transaction ... 8
2.2.6.2 Sequence and Sequencer ... 9
2.2.6.3 Driver ... 10
2.2.6.4 Monitor ... 11
2.2.6.5 Agent ... 12
2.2.6.6 Environment ... 13
2.2.6.7 Subscriber ... 14
2.2.6.8 Scoreboard ... 14
2.2.6.9 Test ... 16
2.3 Serial Peripheral Interface ... 17
3. VERIFICATION AND SIMULATION TOOLS ... **22**
3.1 Design Verification Kit ... 22
3.2 QuestaSim ... 23
4.SERIAL PERIPHERAL INTERFACE INTELLECTUAL PROPERTY **24**
4.1 SPI Clock Generation ... 24
4.2 Parallel Write Interface ... 25
4.3 Parallel Read Interface ... 26
5. VERIFICATION OF SERIAL PERIPHERAL INTERFACE INTELLECTUAL PROPERTY BY USING UNIVERSAL VERIFICATION METHODOLOGY ... **27**
5.1 Top Block and Interface ... 27
5.2 Transaction ... 29
5.3 Sequence and Sequencer ... 30
5.4 Driver ... 32
5.5 Monitor ... 40
5.6 Agent ... 42
5.7 Environment ... 45
5.8 Subscriber ... 46
5.9 Config ... 48

5.10 Test 50
5.11 Simulation and Test Results 52
6. REALISTIC CONSTRAINTS AND CONCLUSIONS 63
6.1 Practical Application of this Project 63
6.2 Realistic Constraints 63
6.2.1 Social, environmental and economic impact 64
6.2.2 Cost analysis 64
6.2.3 Standards 64
6.2.4 Health and safety concerns 64
6.3 Future Work and Recommendations 64
REFERENCES 65
CURRICULUM VITAE 69

ABBREVIATIONS

UVM : Universal Verification Methodology

SPI : Serial Peripheral Interface

DUT : Device Under Test

HDL : Hardware Description Language

OVM : Open Verification Methodology

I2C : Inter-Integrated Circuit

UART : Universal Asynchronous Receiver Transmitter

FSM : Finite State Machine

FPGA : Field Programmable Gate Array

RAM : Random Access Memory

DVKit : Design Verification Kit

IDE : Integrated Development Environment

IP : Intellectual Property

VIP : Verification Intellectual Property

LIST OF FIGURES

Page

Figure 2.1 : UVM Testbench Structure........ **3**
Figure 2.2 : UVM Class Tree........ **4**
Figure 2.3 : UVM Phases........ **5**
Figure 2.4 : Transaction transfer via TLM ports........ **6**
Figure 2.5 : Connection between testbench and the DUT via interface **7**
Figure 2.6 : Transaction Transfer Scheme........ **8**
Figure 2.7 : Sequence-driver communication via sequencer........ **9**
Figure 2.8 : Transaction transfer from sequence to driver........ **10**
Figure 2.9 : Transaction transfer from driver to DUT **10**
Figure 2.10 : Monitor in a testbench........ **11**
Figure 2.11 : Agent class........ **12**
Figure 2.12 : Environment class **13**
Figure 2.13 : Scoreboard’s connections........ **15**
Figure 2.14 : Test class **16**
Figure 2.15 : SPI Mode 0, CPOL=0, CPHA=0, CLK idle state=low........ **17**
Figure 2.16 : SPI Mode 1, CPOL=0, CPHA=1, CLK idle state=low........ **18**
Figure 2.17 : SPI Mode 2, CPOL=1, CPHA=0, CLK idle state=high........ **18**
Figure 2.18 : SPI Mode 3, CPOL=1, CPHA=1, CLK idle state=high........ **18**
Figure 2.19 : SPI diagram with all modes........ **19**
Figure 2.20 : Regular SPI mode structure........ **19**
Figure 2.21 : Daisy-chain mode........ **20**
Figure 3.1 : DVKit interface. **22**
Figure 3.2 : QuestaSim interface........ **23**
Figure 4.1 : Parallel write sequence **25**
Figure 4.2 : Parallel read sequence. **26**
Figure 5.1 : Interface........ **27**
Figure 5.2 : Top block parameter and variable definitions........ **27**
Figure 5.3 : Top block adaptation layer and assertion instantiations........ **28**
Figure 5.4 : Top block adaptation layer and assetion instantiations continued. **28**
Figure 5.5 : Transaction. **29**
Figure 5.6 : Slave sequence class constructor........ **30**
Figure 5.7 : Slave sequence body task. **30**
Figure 5.8 : Master sequence **31**
Figure 5.9: Master driver class constructor and build phase........ **32**
Figure 5.10 : Master driver driverOff and driver Clk functions. **33**
Figure 5.11 : Master driver run phase........ **33**
Figure 5.12 : Master driver run phase continued. **34**
Figure 5.13 : Master driver SPI mode 0........ **34**
Figure 5.14: Master driver SPI mode 1........ **35**
Figure 5.15: Master driver SPI mode 2........ **36**
Figure 5.16: Master driver SPI mode 3........ **36**
Figure 5.17 : Slave driver class constructor and build phase........ **37**
Figure 5.18 : Slave driver run phase. **38**
Figure 5.19 : Slave driver SPI mode 0 and 1. .... **38**
Figure 5.20 : Slave driver SPI mode 2 and 3. .... **39**

Figure 5.21 : Monitor class constructor and build phase. **40**
Figure 5.22 : Monitor run phase and collect_transfer function.......... **40**
Figure 5.23 : Monitor SPI mode 0 and 1. .... **41**
Figure 5.24 : Monitor SPI mode 2 and 3. .... **41**
Figure 5.25 : Master agent class constructor and declarations. **42**
Figure 5.26 : Master agent build phase **42**
Figure 5.27 : Master agent connect phase. **43**
Figure 5.28 : Slave agent class constructor and declarations. **43**
Figure 5.29 : Slave agent build phase. **43**
Figure 5.30 : Slave agent connect phase. **44**
Figure 5.31 : Environment connections. **45**
Figure 5.32 : Subsciber write_master function. **46**
Figure 5.33 : Subscriber write_slave function. **47**
Figure 5.34 : Config class. **48**
Figure 5.35 : Config class continued. **49**
Figure 5.36 : Test class constructor and build phase. **50**
Figure 5.37 : Test first sequence. **51**
Figure 5.38: Test second sequence. **51**
Figure 5.39 : Test base. **52**
Figure 5.40 : Design loopback. **52**
Figure 5.41 : SPI mode 0, design=master testbench=slave. **53**
Figure 5.42 : SPI mode 0, design=slave testbench=master. **53**
Figure 5.43 : SPI mode 0, design=master testbench=slave results. **54**
Figure 5.44 : SPI mode 0, design=slave testbench=master results. **55**
Figure 5.45 : SPI mode 1, design=master testbench=slave. **56**
Figure 5.46 : SPI mode 1, design=slave testbench=master. **56**
Figure 5.47 : SPI mode 1, design=master testbench=slave results. **57**
Figure 5.48 : SPI mode 1, design=slave testbench=master results. **58**
Figure 5.49 : SPI mode 2, design=master testbench=slave. **58**
Figure 5.50 : SPI mode 2, design=slave testbench=master. **59**
Figure 5.51 : SPI mode 2, design=master testbench=slave results. **59**
Figure 5.52 : SPI mode 2, design=slave testbench=master results. **60**
Figure 5.53 : SPI mode 3, design=master testbench=slave. **60**
Figure 5.54 : SPI mode 3, design=slave testbench=master. **61**
Figure 5.55 : SPI mode 3, design=master testbench=slave results. **61**
Figure 5.56 : SPI mode 3, design=slave testbench=master results. **62**

VERIFICATION OF A SERIAL PERIPHERAL INTERFACE INTELLECTUAL PROPERTY BY USING UNIVERSAL VERIFICATION METHODOLOGY

SUMMARY

Due to increasing complexity of designed electronic systems, electronic system design industry is trying to handle the challenges of marketing time and the quality of the design. It is common to choose one over the other, but this compromise can be avoided. Focusing on “verifying correctly” can increase both productivity and quality. From an Intellectual Property (IP) to a System on Chip (SoC), successful design of any system depends on the correct verification. From chip level to card level and finally to system level, the cost of detecting an incorrect chip increases 10 times with each level pass. Sometimes, deciding a correct verification methodology is not easy, and choosing the wrong one can cause problems later. Verification IP (VIP) provides tools to make the correct assumption.

VIP is nothing but a model that provides a user interaction tool at different levels of abstraction of the basic design. Choosing the correct VIP, which includes identification and verification methodology, is as difficult as choosing a design IP. Correct methodology increases reusability. Debugging and error correcting is simple with the correct methodology. It also improves the ability to abstract the underlying complexity. Understanding how to build a permanent VIP is a challenge due to changing technologies and market conditions.

Serial Peripheral Interface (SPI) is a synchronous serial communication interface. SPI devices communicate in full duplex mode using a master-slave structure. The data from the master or the slave is adjusted on the rising or falling edge of the clock. Master and slave can send data simultaneously. Data is transferred between the master and the slave and that transfer is synchronized to the clock that is created by the master. SPI has 4 signals. These are Clock(SCLK,CLK), chip select/slave select(CS,SS), master-in slave-out(MISO) and master-out slave-in(MOSI). MOSI and MISO are the data lines. Throughout the communication, the data is synchronously sent and acquired via MOSI and MISO lines, since SPI is a full-duplex interface. In SPI, one master can communicate with multiple slaves. Master can select the specific slave to send data or receive data from, by using the Slave Select (SS) signal.

In this project, an SPI Master-Slave Interface design will be used. For the verification of this design, a VIP will be created by using UVM and the SPI interface will be verified by the VIP.

BİR SERİ ÇEVRESEL ARAYÜZ FİKRİ MÜLKİYETİNİN EVRENSEL DOĞRULAMA METHODU İLE DOĞRULANMASI

ÖZET

Günümüzde tasarlanan elektronik sistemlerin karmaşıklığının artması sebebiyle, elektronik sistem tasarım endüstrisi pazara sunma süresi ve tasarım kalitesi zorlukları ile baş etmeye çalışmaktadır. Birini diğerinin pahasına elde etmek çok yaygındır, ancak bu ödün verme kaçınılmaz değildir. "Doğru doğrulamaya" odaklanmak hem üretkenliği hem de kaliteyi artırabilir. Bireysel bir fikri mülkiyet (Intellectual Property - IP) bloğundan bir yonga üstü sisteme (System on Chip – SoC) herhangi bir uygulamanın başarılı bir şekilde tasarlanması, doğru doğrulamaya bağlıdır. Yonga düzeyinden kart düzeyine ve son olarak sistem düzeyine her geçişte, hatalı bir yongayı tespit etmenin maliyeti 10 kat artar. Bazen doğru doğrulama metodolojisini yargılamak kolay değildir ve yanlış olanı seçmek daha sonra sorunlara neden olabilir. Doğrulama IP'si (Verification IP - VIP) doğru yargıya varabilmek için araçlar sağlar.

VIP, temel tasarımın farklı soyutlama düzeylerinde bir kullanıcı etkileşimi aracı sağlayan bir modelden başka bir şey değildir. Tanımlamasını ve doğrulama metodolojisini değerlendirmeyi içeren doğru VIP'yi seçmek, bir tasarım IP'si seçmek kadar zordur. Doğru metodoloji, yeniden kullanılabilirliği, hata ayıklama ve hata düzeltme kolaylığını ve bakımı artırırken altta yatan karmaşıklığı soyutlama yeteneğini geliştirir. Kalıcı bir VIP'nin nasıl oluşturulacağını anlamak, değişen teknolojiler ve pazar koşulları göz önüne alındığında bir zorluktur.

Seri Çevresel Arayüz (Serial Peripheral Interface - SPI),senkron bir seri haberleşme arayüzüdür. SPI, bir usta/çırak ilişkisi içinde çift yollu bir haberleşme düzeni sağlar. Bilgi, ustanın ürettiğin bir saat sayesinde ustadan çırağa ya da çıraktan ustaya iletilecek şekilde aktarılır. Bu iletişim, ustanın ürettiği saatin yükselen ya da alçalan kenarlarına entegre edilmiştir. SPI, yükselen ya da alçalan kenarlarda bilgi iletimi sağlaması için 4 farklı modda çalışmaktadır. Bilgi, Usta Giriş-Çırak Çıkış(MISO) ve Usta Çıkış-Çırak Giriş(MOSI) hatları üzerinden, eşzamanlı olarak aktarılır. Ayrıca birden fazla çırak ile çalışmak SPI protokolünde mümkündür. Usta, Çırak Seçim(Slave Select-SS) biti sayesinde hangi çırağa bilgi aktaracağını ya da bilgi toplayacağını seçebilir.

Bu projede, daha önceden tasarlanmış bir SPI Master-Slave arayüzü kullanılacaktır. Daha sonra bu tasarlanmış modülün doğrulanması için bir VIP tanımlanacak ve VIP kullanılarak SPI modülü doğrulanacaktır.

1. INTRODUCTION

In this project, a digital circuit design is verified by using Universal Verification Methodology (UVM) [1]. UVM is a regulated methodology and has many advantages for the engineers and it is very reliable. Main purpose of this project is to learn and perform the UVM methodology and create a verification environment. Since the main goal is to create a verification environment, a previously designed interface is used as Serial Peripheral Interface Intellectual Property (SPI IP) in this project [2]. This SPI IP is an SPI Master-Slave interface and is designed by using Verilog [3] and SystemVerilog [4] hardware languages. In order to create a verification environment, SystemVerilog and UVM is learnt and implemented. The design code is examined and the signals are defined and connections are made accordingly to the verification environment.

The design is simulated in a simulation tool, QuestaSim [5] and tested by defining input values and signals. Design's simulation behavior is explained in the SPI IP section.

After that, by using UVM, a verification environment is created and the design is tested by using the Design Verification Kit (DVKit) program [6]. QuestaSim simulation tools is also used and two simulation results are compared. UVM's class library, hierarchy and UVM's phases, as well as all the components that are created and used in the verification environment, all the verification steps and the simulation results are explained in the Verification of SPI IP by using UVM section.

2. BACKGROUND INFORMATION

2.1. Verification

The method of checking and confirming the digital circuit design works as intended is called verification. The verification process includes building a test environment by using hardware description languages (HDL) and simulating the design by using various simulators [7].

The main goal of the verification is detecting mistakes. Each digital circuit design has its task, verification engineers have to check and verify that the task is completed as expected. For that, verification engineers build a test environment to show that expected and simulated results are matching.

Checking and confirming the digital circuit design for its accuracy and verifying its reliability are crucial. Therefore, it is inevitable that digital system designs need to be verified.

2.2. Universal Verification Methodology (UVM)

Universal Verification Methodology (UVM) is a regulated methodology for verifying digital circuit designs. UVM is a set of class libraries characterized using the syntax and semantics of SystemVerilog. Its primary goal is to create verification environments that are reusable and arranged well. Before UVM, verification languages like SystemVerilog, e [8] and Vera [9] were used to verify designs. Also, Open Verification Methodology (OVM) were used as verification methods [10]. However, as the complexity of the digital circuit designs increased, verification of these systems with these methods and languages became more and more challenging. Initially, UVM is derived from OVM and promoted by vendors like Synopsys [11], Cadence [12] and Mentor [13]. The main advantage of UVM is the UVM class library. UVM class library is written in SystemVerilog language and it provides particular mission for each component. For instance, a driver class is only in charge of driving signals to the design whereas a monitor class just monitors the design interface and does not drive signals. This feature can be achieved with UVM's class hierarchy. With class hierarchy, each component and sub-component have different responsibilities and can updated separately. Creating a verification environment by using SystemVerilog language is simple but updating and changing each component can be hard. However, with UVM, creating and changing the verification environment are both uncomplicated and easy. Thus, UVM class library and class hierarchy is making the verification environments reusable and making them neat, not complex for the users.

2.2.1. Verification Environment

For the verification of a digital circuit design, a verification environment is needed. This environment is called testbench and it includes all the verification components. This testbench will communicate with the DUT (Device Under Test) by using interface. In Figure 2.1, UVM testbench structure is shown. It also includes the interface connection between DUT and the testbench. All the components of this testbench will be explained later. UVM class library, class hierarchy, UVM phases will be used for the creation of the verification environment [20].

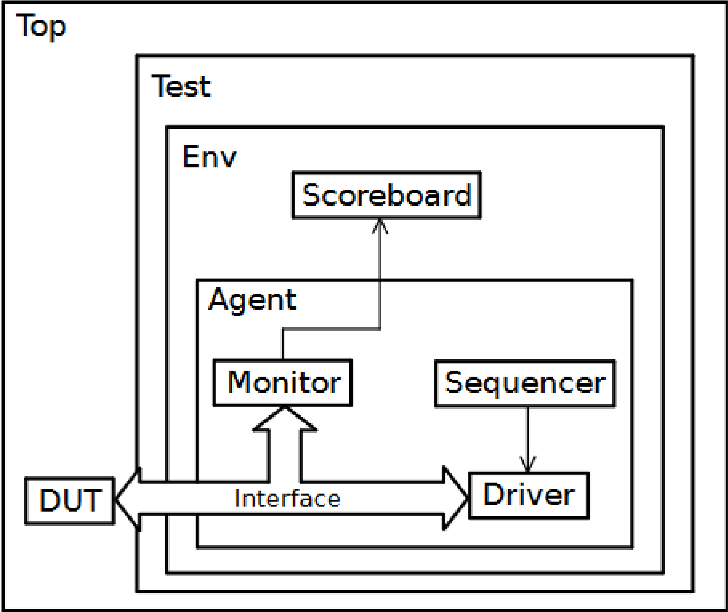

Figure 2.1: UVM Testbench Structure [20]

2.2.2. UVM Classes

As previously mentioned, UVM class library provides great advantages. The library makes it easy to edit and update each component separately, without having to change the whole testbench. A class tree of UVM classes can be seen down below in Figure 2.2.

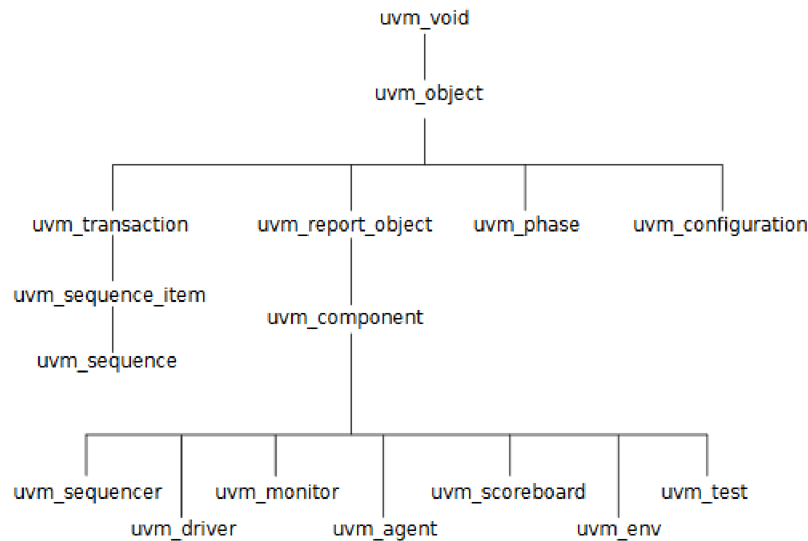

Figure 2.2: UVM Class Tree [20]

Uvm_object class is the core class for the all UVM data and hierarchical classes. All the components and transactions are derived from uvm_object. Its main duty is to give description to the methods for universal applications like create, copy, compare, print and record [21].

Uvm_component class is the origin core class for UVM components. All UVM components are derived from uvm_component. Uvm_component provides hierarchy interface to search and travel through component hierarchy. All the verification components are derived from uvm_component class such as uvm_driver, uvm_agent, uvm_monitor etc. Besides hierarchy interface, uvm_component also grants phasing, reporting, transaction recording and factory interfaces [22].

UVM components and the UVM transaction classes will be explained later.

2.2.3. UVM Phases

UVM phases are coordinated structures for the verification environment. Every verification component pass through UVM phases. Every time a new component is created, the simulation of the verification environment runs through UVM phases to build, compose and connect the testbench component hierarchy [20]. The most essential phases can be seen in Figure 2.3.

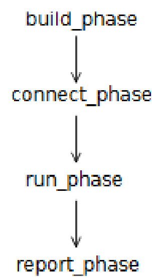

Figure 2.3: UVM Phases [20]

Build phase creates testbench components and establish their instances.

Connect phase connects verification components, for instance, it connects driver component to sequencer component.

Run phase is the primary phase where the simulation is carried out.

Report phase demonstrates the result of the simulation [20].

2.2.4. Transaction Level Modeling (TLM)

Transaction Level Modeling (TLM) is an interface structure that transfers the created transactions between the components. This transfer is achieved by exclusive ports called TLM interfaces. This modeling sets apart components from each other, if one component is updated, other components that are connected to this component are not affected [44].

TLM library comes with transaction-level interface, ports, imports, exports and analysis ports. In Figure 2.4, a simple connection between ports of components can be seen.

As an example, driver component uses a TLM port, seq_item_port, to initiate communication with sequencer. With functions like get_next_item and item_done, this communication can be started and finished.

Another example is in the monitor class. Monitor class uses the analysis port (uvm_analysis_port) to call the write function. This write function essentially writes the data that is observed in monitor to the analysis port. Any component that is connected to this analysis port via analysis export can read the data. When monitor class calls the analysis_port.write(), it checks all the connected exports and calls their write functions. This way scoreboard or subscriber components can read and write the data that is observed in monitor.

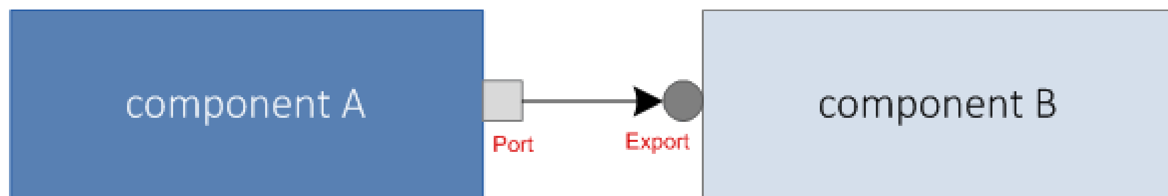

Figure 2.4: Transaction transfer via TLM ports [44]

2.2.5. Top Block and Interface

In order to connect the testbench and DUT, two components are required. These are top block and the interface. The top block establishes instances of the DUT and the testbench and the interface connects them. The interface contains all the signals of the DUT. Other components such as driver and monitor, as well as the DUT will be attached to the interface [23]. Figure 2.5 shows the simple connection between testbench and the DUT via interface.

In order to connect to interface to the DUT, virtual interface object is established in top block, then the virtual interface is linked with the DUT that is instantiated in the top block as well.

As mentioned before, interface holds the signals, and allows the user to file and check the transactions transferred between the testbench and the DUT in top block. Since all the data covered in the interface, it is a simple way to establish communication this way.

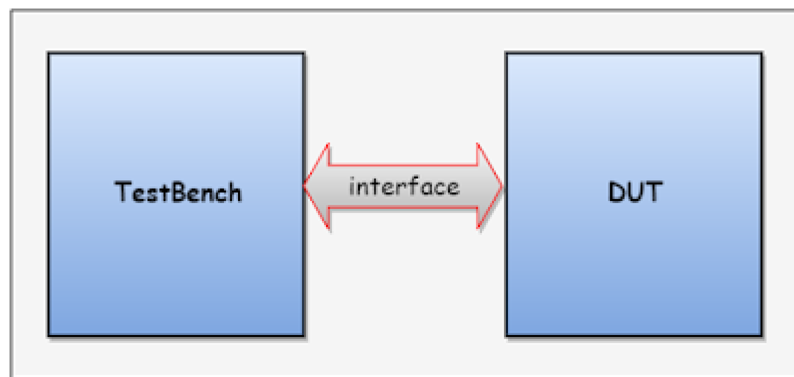

Figure 2.5: Connection between testbench and DUT via interface [24]

2.2.6. Components

2.2.6.1. Transaction

One of the basics of the verification of a digital circuit design is to create data packages. These data packages are sent to the DUT to be processed. The smallest of these data packages or items are called transactions. Usually, the driver class handles signal transfers at the bit layer, but when an 8-bit of a package is needed to be transferred to the DUT, a transaction class is needed [25].

A transaction is a class object and it develops from uvm_transaction or uvm_sequence_item classes. Transaction class carries all the data necessary to create the transmissions between the other components. Transactions are the least possible amount of data can be transmitted and can be carried out by the verification environment as can be seen in Figure 2.6.

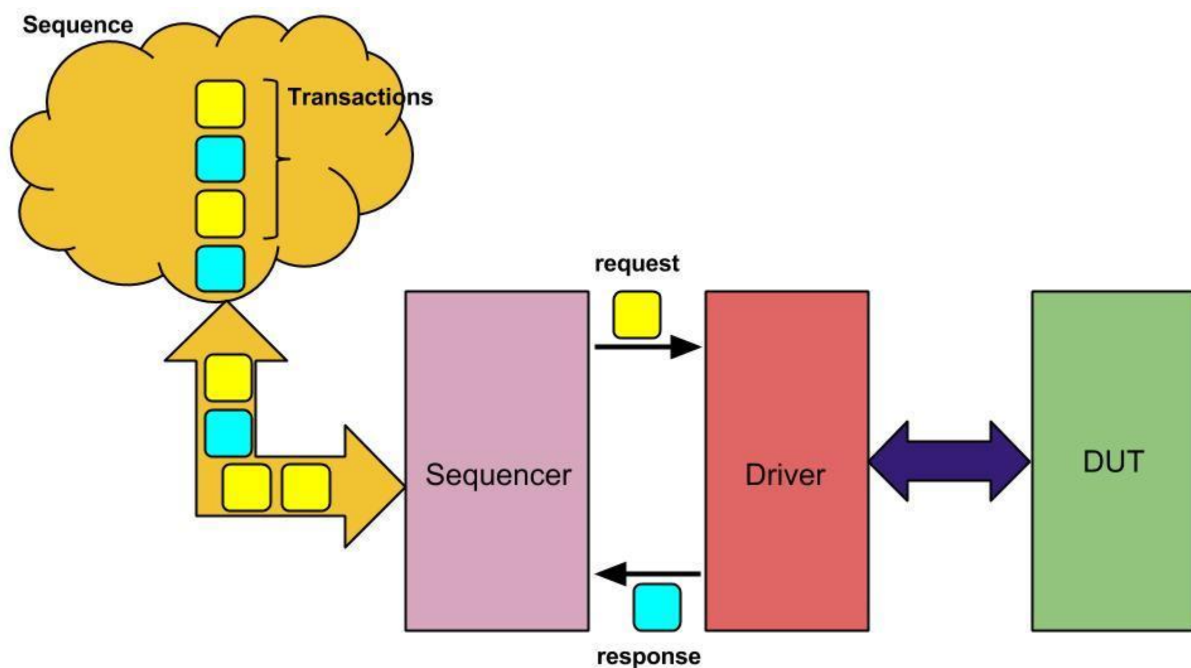

Figure 2.6: Transaction Transfer Scheme [26]

2.2.6.2. Sequence and Sequencer

Sequence class is the class that creates sequences from the previously generated transactions. It extends from uvm_sequence class. These sequences are basically data item combinations, sequence class combines transactions in many ways to generate different outcomes. After that, these sequences are transferred to the driver via sequencer class. Sequencer class extends from uvm_sequencer in order to establish a communication with the driver [25]. This communication establishment can be seen in Figure 2.7.

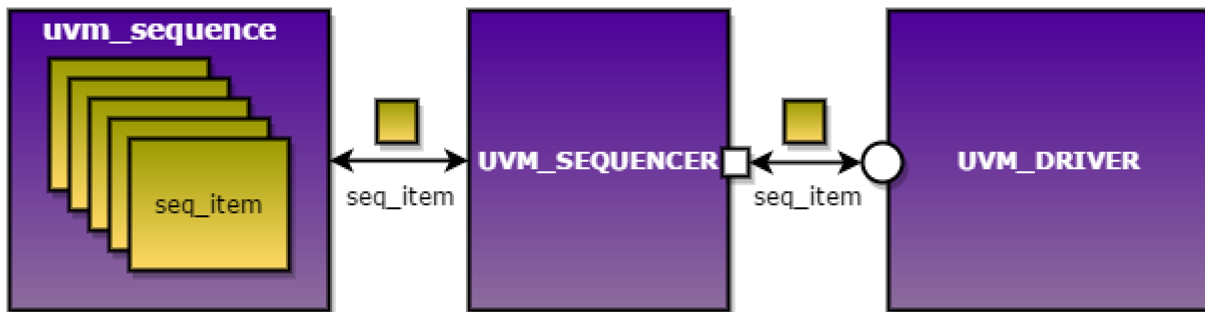

Figure 2.7: Sequence-driver communication via sequencer [27]

2.2.6.3. Driver

Driver is responsible for sending created transactions to the DUT. Driver gets those transactions, changes them into bit-level actions, and drives the data into the DUT [28]. This structure can be seen in Figures 2.8 and 2.9.

As seen in Figure 2.7, driver receives the transactions or sequence items from sequencer. This communication is established via some techniques. Driver class has a uvm_seq_item_pull port to receive transactions from sequencer. First, driver calls the get_next_item function to start the communication. This function runs until all the transactions are transferred. After that, item_done function is called by the driver in order to stop the communication.

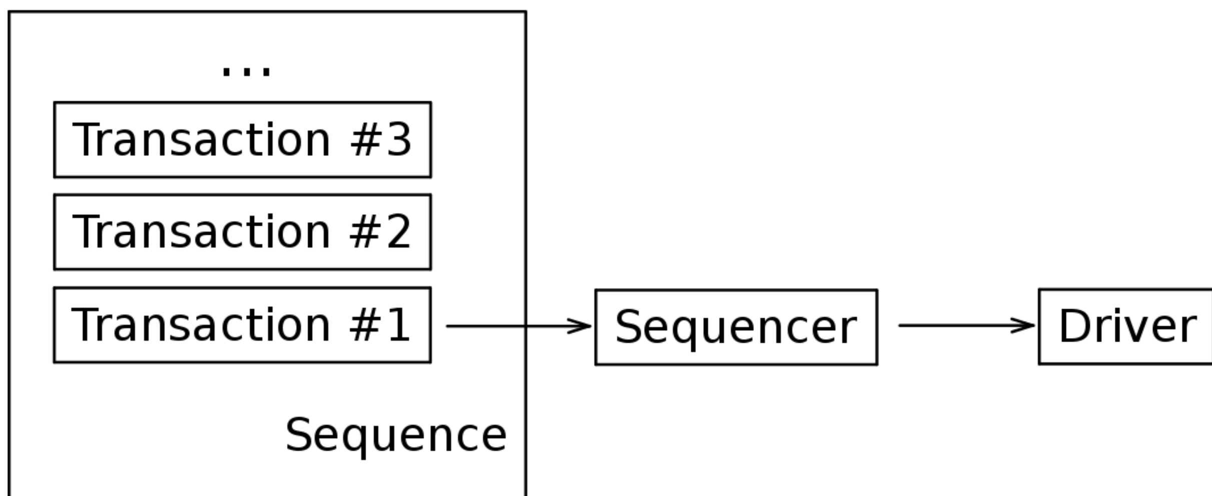

Figure 2.8: Transaction transfer from sequence to driver [25]

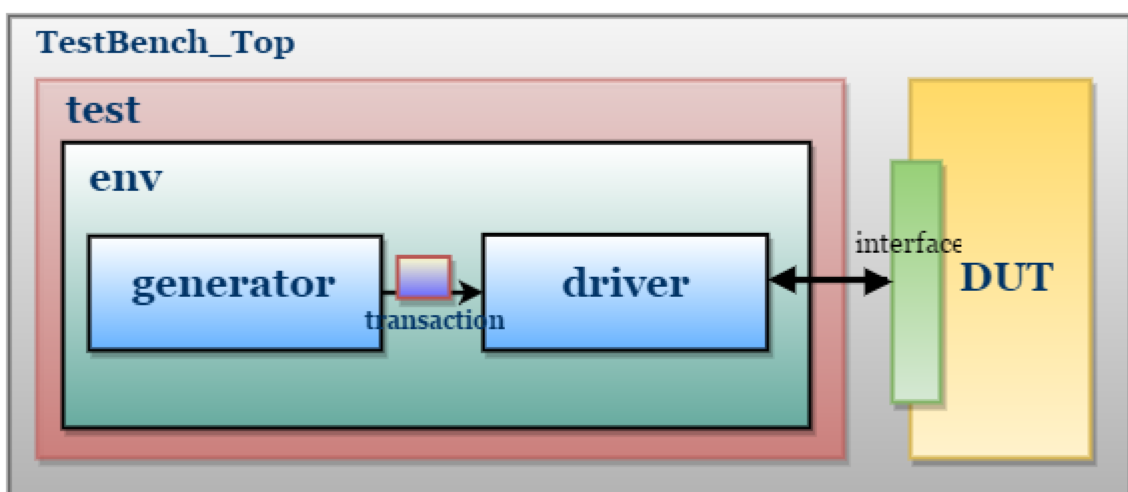

Figure 2.9: Transaction transfer from driver to DUT [29]

2.2.6.4. Monitor

Monitor is a class that inspects and monitors the communication between the DUT and the testbench. It shows the results from the DUT and displays them to the user. Monitor is a static component, that means it does not send any transactions to any component, its only job is to convert the signals that are sent from the DUT to purposeful data, and then these data will be assessed by other components [30]. Monitor's location and its connections between other components can be seen in Figure 2.10.

Monitor class has an analysis port to allow the users to declare the data to other components such as subscriber and scoreboard. Data is not driven by the monitor, translated data is simply read by the subscriber or scoreboard, in order to compare and verify the result. This communication is established via the analysis port.

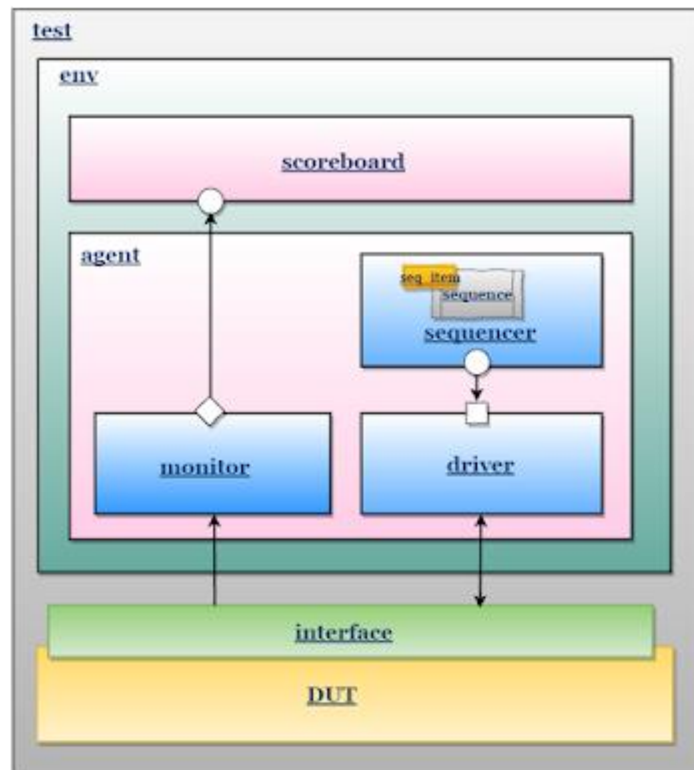

Figure 2.10: Monitor in a testbench [31]

2.2.6.5. Agent

Agent is class that contains all the previous components, driver, monitor and transaction. In Figure 2.11, agent class connects these components to each other and makes the connection to the scoreboard component. It defines the component ports, such as slave and master driver ports, it also allows the user the select the specific driver. Agent class does not need a run phase. All the components established in the build phase and connected in the connect phase. When the agent class is activated by UVM_ACTIVE, it starts an if statement to choose the slave or master driver. Then the transactions are sent or received accordingly to the selected driver [32].

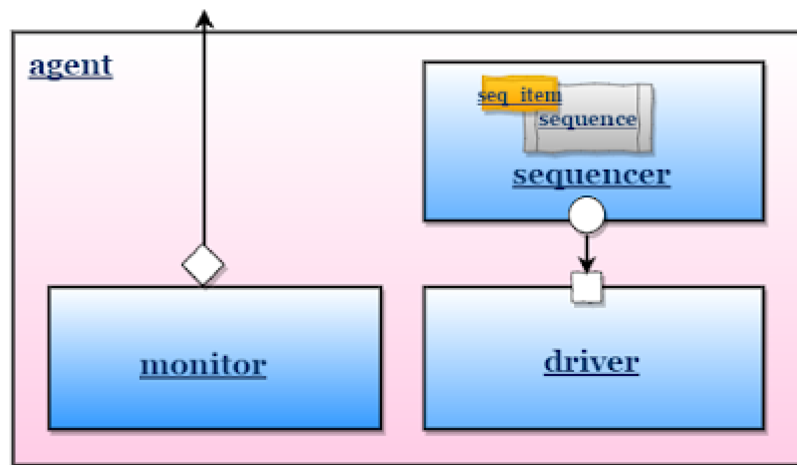

Figure 2.11: Agent class [33]

2.2.6.6. Environment

Environment class is a very simple class that acts like a capsule, it consists of agents, drivers and all the components that are created for the verification environment as can be seen in Figure 2.12.

Agents and scoreboards/subscribers can be incorporated in test class directly but it is not an ideal solution. Environment class is defined because of the reusability, it is easy to update the structure, if incorporation is done in test class, it would take a lot time to upgrade the files. Created environments can be used for multiple tests. For each test, there would be no need to build the environment and its subcomponents from scratch [34].

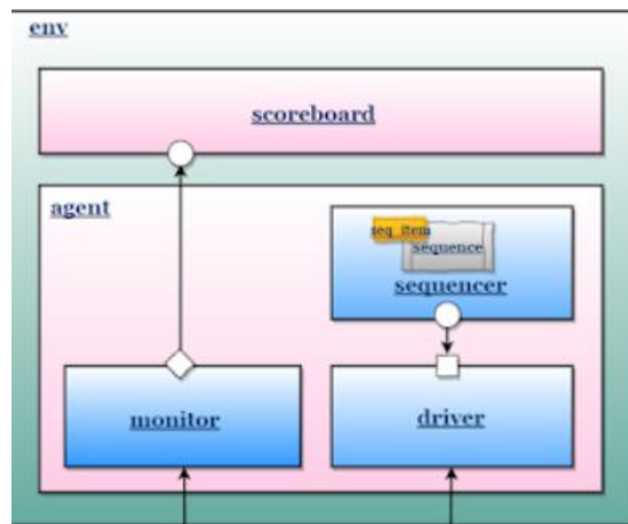

Figure 2.12: Environment class [35]

2.2.6.7. Subscriber

Subscriber is class that has a built-in analysis port named analysis_export which provides access to the write method for receiving transactions. Previously created monitor class has an analysis port which contains write function. With this analysis port, transactions are transferred to analysis components like scoreboard or subscriber. Subscriber basically acts as an audience for the analysis port, they “subscribe” to a communicator and acquires and then prints the results of the received transactions whenever they are published [36].

2.2.5.8. Scoreboard

Scoreboard class is created to validate and verify the design. It examines the results from the DUT, and compares them with the anticipated results. Scoreboard obtains the transactions by checking the monitor’s analysis port [40]. Scoreboard’s location in the testbench structure, as well as its connection to the monitor and DUT can be seen in Figure 2.13.

Scoreboard class uses its export port to read the data from monitor’s analysis port. Monitor class calls the analysis_port.write(), it checks all the connected exports and calls their write functions. After that, scoreboard can analyze and verify the data.

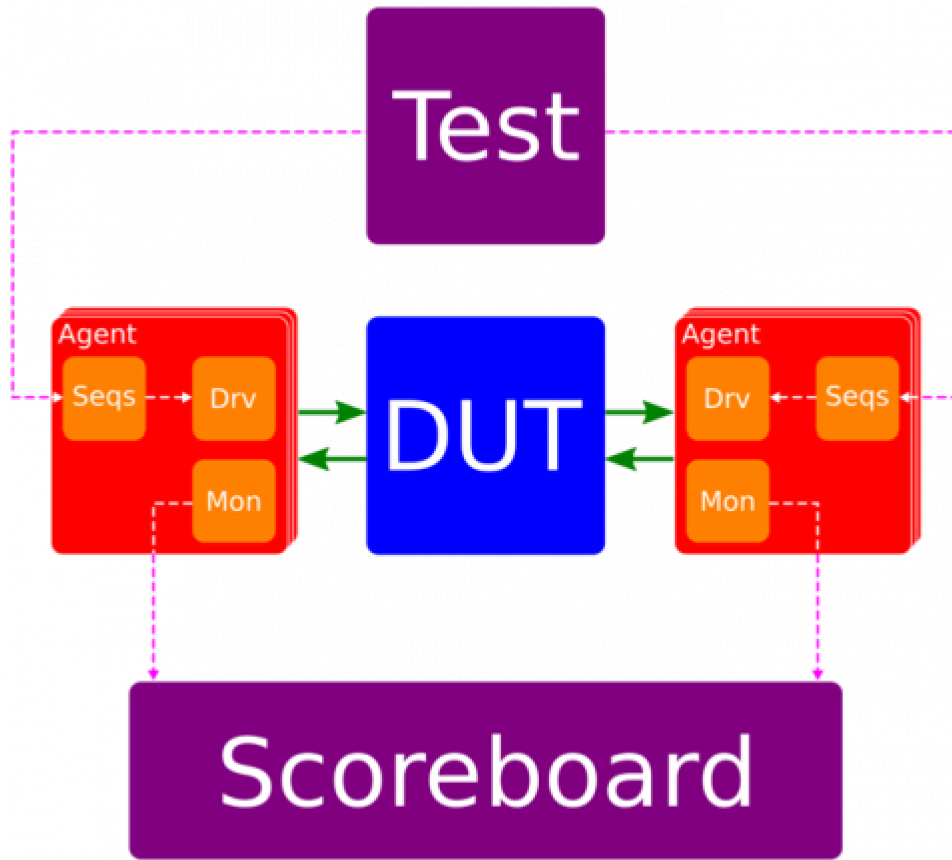

Figure 2.13: Scoreboard's connections [43]

2.2.6.9. Test

Test class is created to run the tests for the design. These tests are built by user and many different tests can be applied to the design since test class contains environment class and all the other components. That is why this method is chosen since it allows the user to reuse the verification environment without changing the all structure [41].

As mentioned, test class capsules the environment class so many different sequences can be carried out by the user, either in the same test via using different sequencers or operating different tests. That way, users validate the design by running different sequence combinations.

Created tests called and carried out in top block of the testbench by using run_test function. As specified before, with this arrangement, more than one test and different sequence combinations can be achieved for different outcomes and the verification of the design would be more valid. Test class encapsulates environment and other classes as can be seen in Figure 2.14.

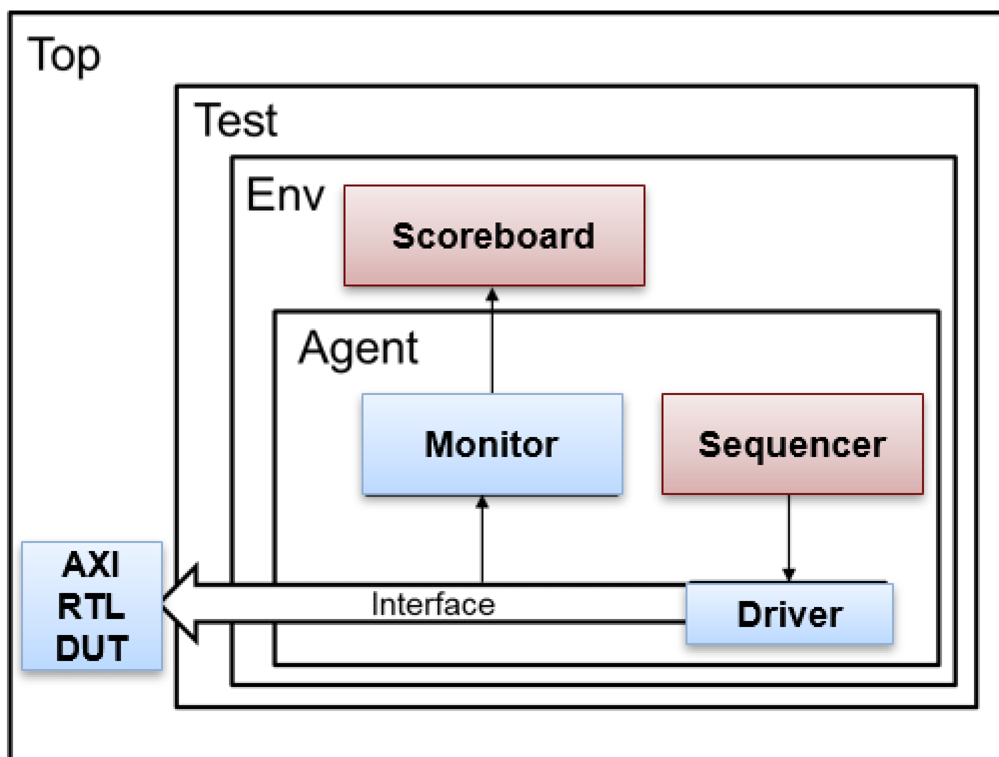

Figure 2.14: Test class [42]

2.3. Serial Peripheral Interface (SPI)

Serial Peripheral Interface (SPI) is a synchronous serial communication interface. SPI devices communicate in full duplex mode using a master-slave structure. The data from the master or the slave is adjusted on the rising or falling clock edge. Master and slave can send data simultaneously. SPI has 4 signals. These are Clock (SCLK, CLK), chip select/slave select (CS, SS), master-in slave-out (MISO) and master-out slave-in (MOSI). MOSI and MISO are the data lines. MOSI sends data from the master to the slave and MISO sends data from the slave to the master. Clock signal is produced by the master. Data is carried between the master and the slave, and synchronized to the clock that is created by the master. To start the SPI communication, the master sends the clock signal and chooses the slave by activating the chip select signal. Throughout the communication, the data is synchronously sent and acquired via MOSI and MISO lines, since SPI is a full-duplex interface. There are 4 modes of the SPI communication. The master can choose the clock polarity (CPOL) or the clock phase (CPHA) to determine these modes. The CPOL bit configures the polarity of the clock signal during the ineffective case. The CPHA bit picks the clock phase. The rising or falling clock edge is used to examine and/or move the data [37]. SPI modes with CPOL and CPHA can be seen down below in Figures 2.15, 2.16, 2.17 and 2.18.

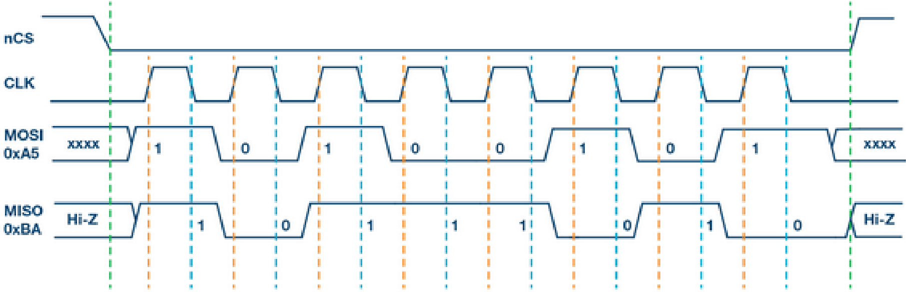

Figure 2.15: SPI Mode 0, CPOL=0, CPHA=0, CLK idle state=low [37]

In SPI Mode 0, clock polarity is 0, which displays that idle state of the clock signal is low. Clock phase is 0, which implies that the data is examined on the rising edge and then the data is transferred on the falling edge of the clock signal as can be seen in Figure 2.15.

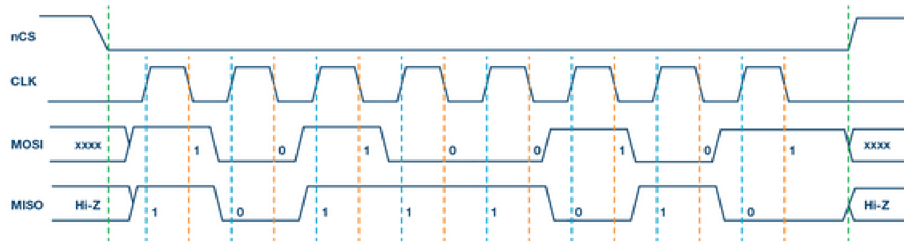

Figure 2.16: SPI Mode 1, CPOL=0, CPHA=1, CLK idle state=low [37]

In SPI Mode 1, clock polarity is 0, which displays that idle state of the clock signal is low. Clock phase is 1, which implies that the data is examined on the falling edge and then the data is transferred on the rising edge of the clock signal as can be seen in Figure 2.16.

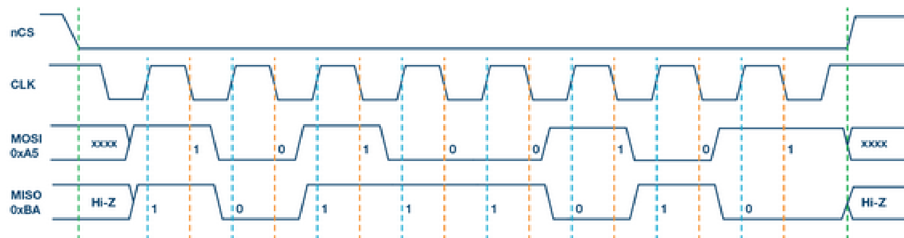

Figure 2.17: SPI Mode 2, CPOL=1, CPHA=0, CLK idle state=high [37]

In SPI Mode 2, clock polarity is 1, which displays that idle state of the clock signal is high. Clock phase is 0, which implies that the data is examined on the falling edge and then the data is transferred on the rising edge of the clock signal as can be seen in Figure 2.17.

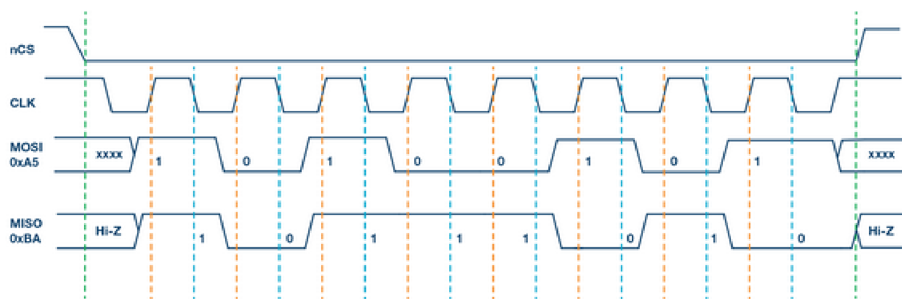

Figure 2.18: SPI Mode 3, CPOL=1, CPHA=1, CLK idle state=high [37]

In SPI Mode 2, clock polarity is 1, which displays that idle state of the clock signal is high. Clock phase is 1, which implies that the data is examined on the rising edge and then the data is transferred on the falling edge of the clock signal as can be seen in Figure 2.18.

An SPI diagram, including all the modes can be seen down below in Figure 2.19.

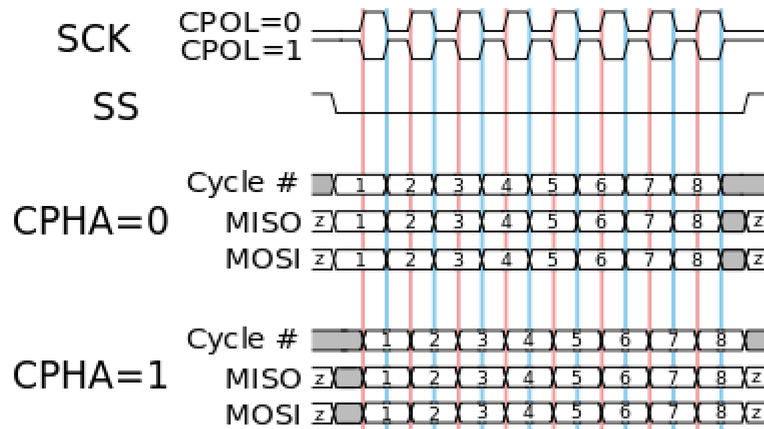

Figure 2.19: SPI diagram with all modes [38]

One SPI master can work with numerous slaves. These slaves can be linked in regular mode or daisy-chain mode. Figure 2.20 shows the regular mode structure. In regular mode, every slave has its own chip select signal from the master. When the chip select signal is activated by the master, MOSI and MISO lines are ready to use for the clock and the data transfer for chosen slave. If master activates more than one chip select, it cannot recognize which slave is sending data and the data on MISO line becomes distorted [37].

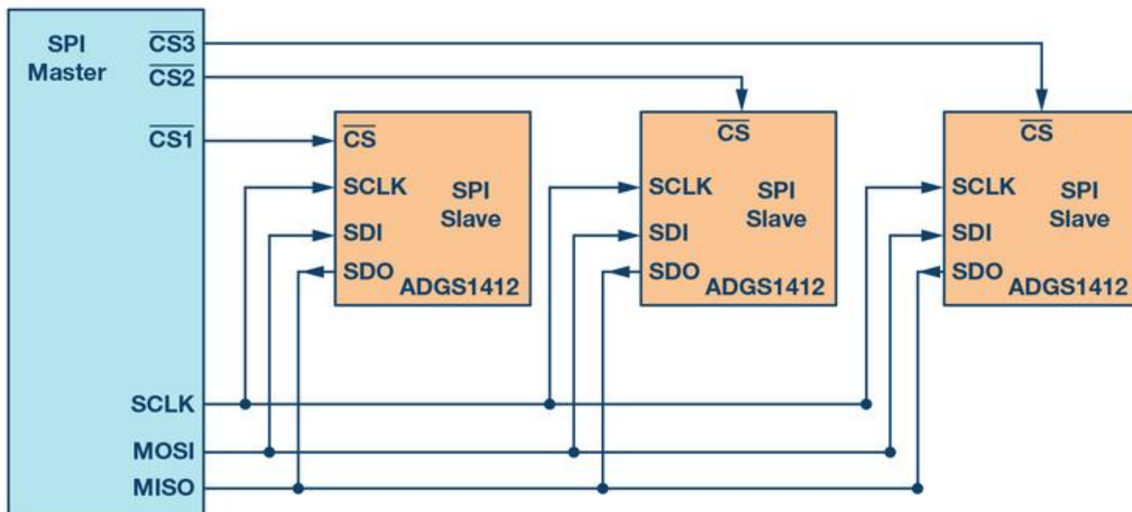

Figure 2.20: Regular SPI mode structure [37]

In daisy-chain mode, in Figure 2.21, the slaves arranged such that the chip select signal for the slaves is connected to each other and data flows from one slave to the next. In this setup, each slave gets the exact SPI clock simultaneously. First slave obtains the data from the master and passes on the next slave until all slaves get the data. It is not as efficient as regular mode, because number of the clock cycles needed to send data is significantly more in daisy-chain mode [37].

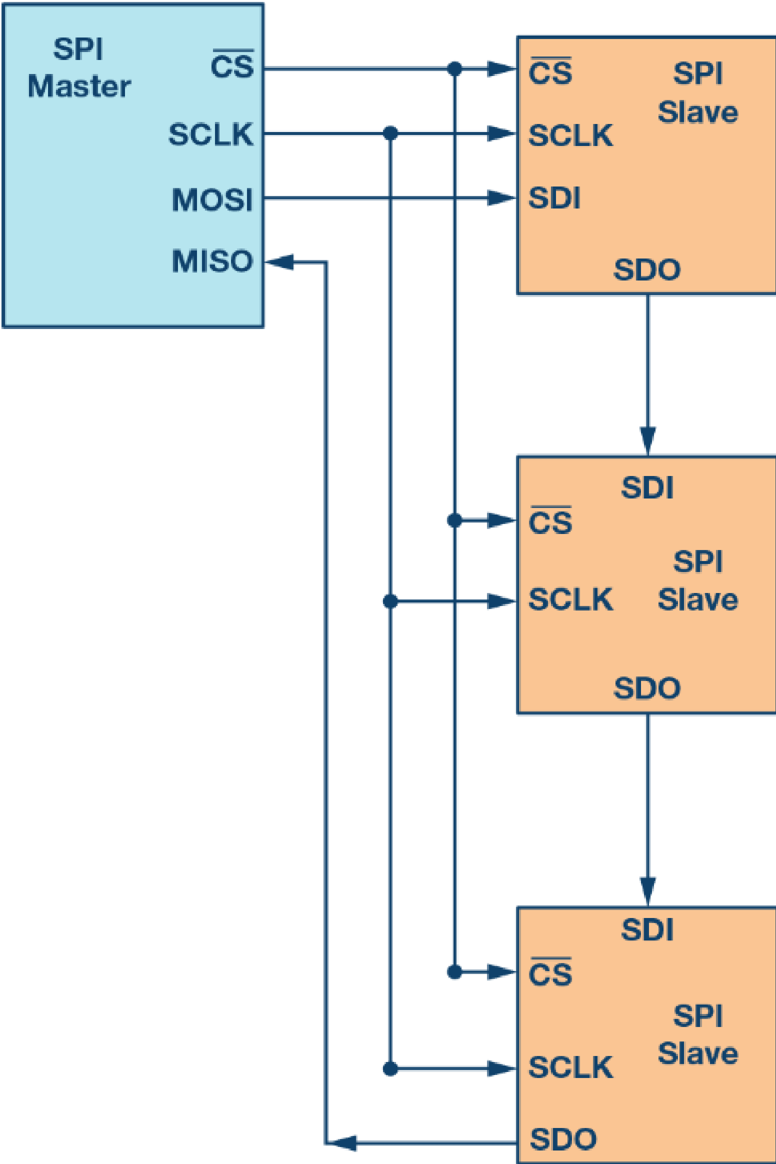

Figure 2.21: Daisy-chain mode [37]

There are some advantages and disadvantages of SPI, compared to other communication protocols like I2C (Inter-Integrated Circuit) [14] and UART (Universal Asynchronous Receiver Transmitter) [15]. First, SPI has some advantages over its counterparts. Unlike UART, SPI has no start and stop bits, data is transmitted repeatedly without disruption. SPI's slave communication method is not as complex as I2C's slave communication method. SPI has greater data transfer percentage that I2C. Also, SPI has MISO and MOSI lines, both of these lines can be used for transmitting and receiving data simultaneously. However, SPI has some disadvantages as well. SPI uses four wires whereas I2C and UART use two wires. Since SPI has no start or stop bits, no confirmation on the delivery of the data, I2C has that feature. Also, unlike UART, SPI has no parity bit so that it cannot inspect the mistakes in the data. Finally, although SPI supports multiple slaves, it only permits one master [39].

3. Verification and Simulation Tools

Creating a verification environment, simulating the design and comparing the results require some tools. In this project, an Integrated Development Environment (IDE) is needed for developing the verification environment and for the simulations, a simulation environment is needed. For IDE, DVKit is used and for simulator, QuestaSim is chosen.

3.1. Design Verification Kit (DVKit)

DVKit is an IDE for design and verification engineers. IDE provides engineers with extensive tools for the software development. DVKit helps the verification engineers to handle the verification environment which consist of SystemVerilog and UVM. It comes with an editor, SystemVerilog Editor (SVEditor), for Verilog and SystemVerilog as well as UVM. It also comes with C development tools and Webtools for development but these are not necessary to create a verification environment. These tools can be activated by using the Eclipse plugin since DVKit's IDE is Eclipse-based [6]. Main interface of the DVKit can be seen in Figure 3.1.

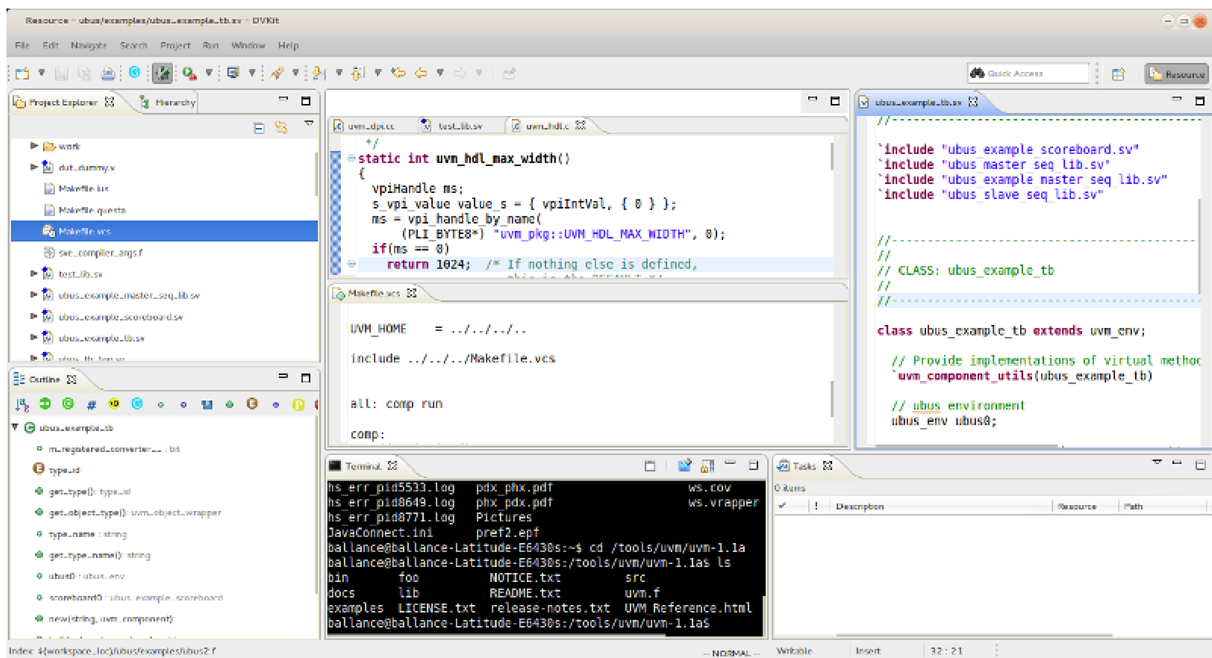

Figure 3.1: DVKit interface [6]

3.2. QuestaSim

Questa is a verification tool that includes an integrated platform called QuestSim. Questa is owned and developed by Mentor Graphics. QuestaSim can perform advanced verification of electronic systems with high efficiency. Administration and debugging facilities are embedded in QuestaSim structure [5]. It is based on ModelSim [16] tool, which is also owned by Mentor Graphics. Main difference between ModelSim and QuestaSim is, QuestaSim supports SystemVerilog and UVM, but ModalSim does not, which is why in this project, QuestaSim is used for the simulations -.

QuestaSim's main interface can be seen in Figure 3.2.

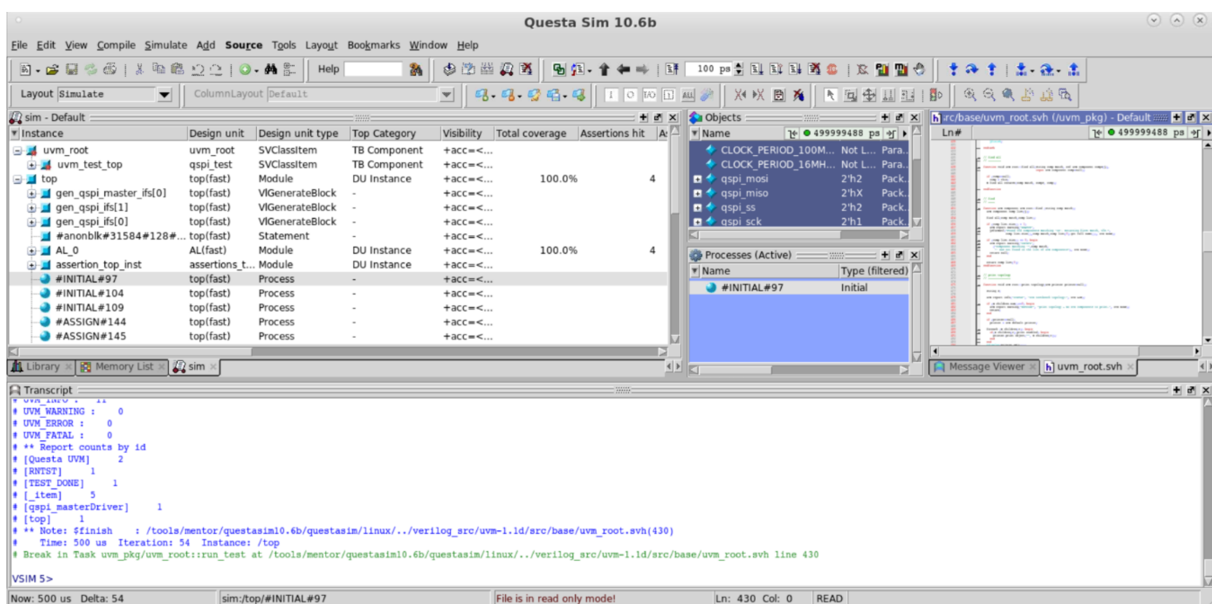

Figure 3.2: QuestaSim interface

4. Serial Peripheral Interface Intellectual Property (SPI IP)

An SPI Master-Slave Interface is used as SPI IP to verify in this project. This design includes an SPI clock generation, parallel read interface and parallel write interface. Read and write sequences can be seen down below.

The actions in the design are synchronous to two clocks, “sclk_i” and “pclk_i”. These clocks are not parallel, they are not occurring at the same time. All actions within the center are simultaneous to “sclk_i”. By using “sclk_i”, SPI core clock can be created. “sclk_i” is divided by the frequency that is double the amount of the SPI SCLK line frequency. All coordinated I/O connection actions are simultaneous to “pclk_i”. This clock is also known as high speed clock.

This design is uncomplicated and very easy to use. It has parallel inputs and outputs that behave like a synchronous memory I/O. It is parameterizable via generics for the data width (N), SPI modes (CPHA and CPOL), prefetch signaling (‘PREFATCH’) and the SPI base clock division from “sclk_i” (SPI_2X_CLK_DIV) by the user.

4.1. SPI Clock Generation

The clock creation is obtained from the high-speed “sclk_i” clock for the SPI SCLK. To establish the SPI base clock, the center divides the source clock by the default parameter “SPI_2X_CLK_DIV”. For the SPI 2X clock, the user must adjust the divider value, which is twice the desired SCLK frequency.

All registers in the center are clocked by the high-speed clocks and clock enables makes the Finite State Machine (FSM) [17] and other logics to execute at lower rates. Field Programmable Gate Array (FPGA) [18] clock utilities such as global clock buffers are preserved by this architecture and path delays that are created by combinational clock divider outputs are averted. To control asynchronous clocks for the SPI and parallel interfaces, this center has asynchronous clock domain circuitry.

4.2. Parallel Write Interface

The parallel interface has an input port “di_i” and an output port “do_o”. Three signals are used to control the parallel load: “di_i, “di_req_o” and “wren_i”. “di_req_o” is a look ahead data request line, it is used to adjust the “PREFETCH” clock cycles up front to synchronize a pipelined memory, to give the next input data at “di_i” directly to have continuous clock at SPI bus, to grant back-to-back continuous load.

For a pipelined sync Random Access Memory (RAM) [19], a PREFETCH of 2 cycles admits an address creator to give the new address to the RAM in one cycle, and the RAM to answer back in one extra cycle, in time for “di_i” to be latched by the shifter. If the user wants to use the sequencer with a different value for PREFETCH, the default setting can be changed at instantiation time. The “wren_i” write enable strobe must be accurate at least one setup time before the rising edge of the last SPI clock cycle, if continuous communication is planned. If “wren_i” is not accurate 2 SPI clock cycles after the last carried bit, the interface goes in an idle state and declares SSEL. When the interface is idle, “wren_i” write strobe carries the data and starts communication. “di_req_o” will strobe when arriving idle state, if a previously carried data has already been submitted. Parallel write sequence, with “pclk_i”, “di_req_o”, “di_i” and “wren_i”, can be seen down below in Figure 4.1.

Figure 4.1: Parallel write sequence

4.3. Parallel Read Interface

To clone the internal shift register data to send the “do_o” port, an internal buffer is utilized. The central shift register is sent to the buffer at the rising edge of the SPI clock, “spi_clk”, when a full word is obtained. “do_valid_o” signal is configured one clock after “spi_clk” to send a synchronous memory instantly. “do_valid_o” and “pclk_i” are concurrent, on rising edges of “pclk_i”, “do_valid_o” alters. The data at the “do_o” port contains the last word obtained while the interface is idle. Parallel read sequence, with “spi_clk”, “pclk_i”, “do_o” and “do_valid_o” can be seen down below in Figure 4.2.

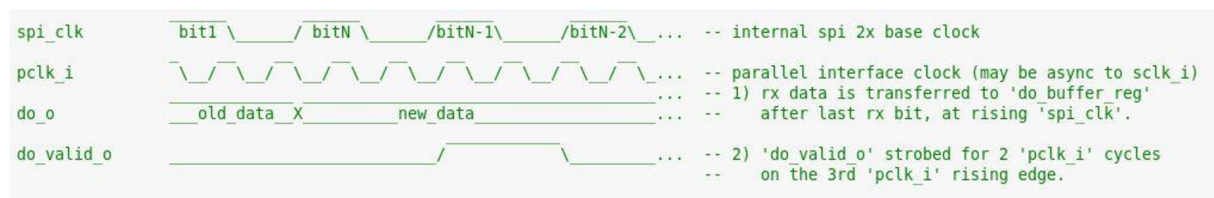

Figure 4.2: Parallel read sequence

The propagation delay of “spi_sck_o” and “spi_mosi_o”, referenced to the internal clock is compensated by identical path delays, but for full duplex application, the “spi_miso_i” sampling delay establishes an setup time referenced to the “sck” signal that restricts the interface’s high frequency.

5. VERIFICATION OF SERIAL PERIPHERAL INTERFACE INTELLECTUAL PROPERTY (SPI IP) BY USING UNIVERSAL VERIFICATION METHODOLOGY (UVM)

5.1. Top Block and Interface

For this project, the interface unit includes SCLK, MOSI, MISO, SS, RST and external clock. These signals are defined as logics, because they are 4-state data types and also SystemVerilog suggests that these signals should be declared as logics by the definition. Interface unit can be seen down below in Figure 5.1.

External clock and RST are used in master driver. External clock is used for creating a clock for the master driver and RST is used for resetting the chip/slave select signal for the master driver.

```
1 interface ifQspi;
2   logic SCLK;
3   logic MOSI;
4   logic MISO;
5   logic SS;
6   logic RST;
7   logic externalClock;
8 endinterface: ifQspi
```

Figure 5.1: Interface

For the top block, parameter and variable definitions such as master and slave memory are defined in Figure 5.2.

```
36 /***********************************************************************************
37  ****************************PARAMETER DEFINITIONS**********************************
38  ***********************************************************************************/
39     parameter CLOCK_PERIOD_100MHZ = 10;
40     parameter CLOCK_PERIOD_16MHZ = 62.5;
41
42 /***********************************************************************************
43  *****************************VARIABLE DEFINITIONS**********************************
44  ***********************************************************************************/
45     bit clk100Mhz;
46     bit clk16Mhz;
47     bit rstn;
48     bit assertion_check_clk125M;
49     bit m_wren_i;
50     bit s_wren_i;
51     bit [11:0] master_memory [5] = {12'h345, 12'hABC, 12'hBBB, 12'h159, 12'h222};
52     bit [11:0] slave_memory [5] = {12'h678, 12'hDEF, 12'hCCC, 12'h260, 12'h333};
53     bit [11:0] m_di_i;
54     bit [11:0] s_di_i;
55     bit m_di_req_o;
56     bit s_di_req_o;
57     int counter;
```

Figure 5.2: Top block parameter and variable definitions

Also, adaptation layer and assertion instantiations are done. Adaptation layer is need because the design is created by using VHDL but verification environment is created by using SystemVerilog and UVM. Because of this, an adaptation layer is implemented. In top block, instances of the DUT signals are created and connected to the interface. Those instance declarations can be seen in Figures 5.3 and 5.4.

```
58 /*****************************************************************************************
59  ******************ADOPTATION LAYER and ASSERTION INSTANTIATION*********************
60  *****************************************************************************************/
61     AL AL_0(
62         .m_clk_i(clk100Mhz),
63         .s_clk_i(clk100Mhz),
64         .m_pclk_i(clk100Mhz),
65         .m_rst_i (m_rstn),
66         .m_spi_miso_i(qspi_miso[0]),
67         .s_spi_miso_o(qspi_master_miso[0]),
68         .m_di_i(m_di_i),
69         .s_di_i(s_di_i),
70         .m_wren_i(m_wren_i),
71         .s_wren_i(s_wren_i),
72         .m_spi_ssel_o(qspi_ss[0]),
73         .m_spi_sck_o(qspi_sck[0]),
74         .m_spi_mosi_o(qspi_mosi[0]),
75         .s_spi_ssel_i(qspi_master_ss[0]),
76         .s_spi_sck_i(qspi_master_sck[0]),
77         .s_spi_mosi_i(qspi_master_mosi[0]),
78         .m_di_req_o(m_di_req_o),
79         .s_di_req_o(s_di_req_o),
80         .m_wren_ack_o(),
81         .s_wr_ack_o(),
82         .m_do_valid_o(),
```

Figure 5.3: Top block adaptation layer and assertion instantiations

```
83         .m_do_o(),
84         .s_do_valid_o(),
85         .s_do_o(),
86         .m_do_transfer_o(),
87         .m_wren_o(),
88         .m_rx_bit_reg_o(),
89         .m_state_dbg_o(),
90         .m_core_clk_o(),
91         .m_core_n_clk_o(),
92         .m_sh_reg_dbg_o(),
93         .s_do_transfer_o(),
94         .s_wren_o(),
95         .s_rx_bit_reg_o(),
96         .s_state_dbg_o()
97         );
98
99     assertions_top assertion_top_inst();
```

Figure 5.4: Top block adaptation layer and assertion instantiations continued

5.2. Transaction

In this SPI VIP, data packages are created with the width of 8 bits. Each data has their unique address. These addresses are defined as 7-bits. Transaction class also has 3 modes, Read (RD), Write (WR) and Error (ERR). MOSI and MISO data are also defined as 8-bit packages in Figure 5.5.

```
class qspi_transaction #(datawidth=12,
                         addrwidth=7) extends uvm_sequence_item;

    typedef enum {WR,RD,ERR} mod;

    rand bit [datawidth-1:0] MOSI_DATA;
    rand bit [datawidth-1:0]  MISO_DATA;

    randc mod MODE;

    int m_bit_number;



  `uvm_object_utils_begin(qspi_transaction)
      `uvm_field_int(MOSI_DATA, UVM_ALL_ON|UVM_NOPACK)
      `uvm_field_int(MISO_DATA, UVM_ALL_ON|UVM_NOPACK)

  `uvm_object_utils_end

  function new (string name = "");
    super.new(name);
    `uvm_info(get_full_name(),$psprintf(uvm_object_value_str(this)),0);
  endfunction

endclass
```

Figure 5.5: Transaction

5.3. Sequence and Sequencer

In Figures 5.6, 5.7 and 5.8, sequence class is seen. Here, created transactions are taken and put into miso_data and mosi_data as random 12-bit packages. Then, in body task, these sequences are requested by "req" and put into MISO_DATA and MOSI_DATA. MISO_DATA is sent to slave driver by sequencer class. MOSI_DATA is sent to master driver by sequencer class.

For this project, two sequences classes are made for master and slave drivers. Each sequence creates the same 12-bit packages as mentioned above. Each driver is getting data from their unique sequences by sequencers.

```
1  class qspi_sequence extends uvm_sequence  #(qspi_transaction);
2    `uvm_object_utils(qspi_sequence)
3
4    qspi_transaction # (`DATAWIDTH,`ADDRWIDTH) req;
5
6    int num_transactions; // number of transactions, the value set from spi_test
7
8    randc bit [11:0] miso_data;
9    int bitstosent = 12;
10
11   ///// Class Constructor //////
12
13   function new (string name = "");
14     super.new(name);
15
16     req = qspi_transaction# (`DATAWIDTH,`ADDRWIDTH)::type_id::create("req");
17     req.print();
18   endfunction
19
20   ///// Body Task /////
21
```

Figure 5.6: Slave sequence class constructor

```
22   task body;
23     begin
24         req.MISO_DATA = miso_data;
25         req.m_bit_number = bitstosent;
26         `uvm_send(req)
27     end
28
29         end
30     end
31    endtask:body
32
33 endclass:qspi_sequence
```

Figure 5.7: Slave sequence body task

```
1 class qspi_sequence_master extends uvm_sequence  #(qspi_transaction);
2   `uvm_object_utils(qspi_sequence_master)
3 
4   qspi_transaction # (`DATAWIDTH,`ADDRWIDTH) req;
5 
6   int num_transactions; // number of transactions, the value set from spi_test
7 
8   randc bit [11:0] mosi_data;
9   int bitstosent = 12;
10 
11  ///// Class Constructor //////
12 
13  function new (string name = "");
14    super.new(name);
15 
16    req = qspi_transaction# (`DATAWIDTH,`ADDRWIDTH)::type_id::create("req");
17    req.print();
18  endfunction
19 
20  ///// Body Task /////
21 
22  task body;
23    begin
24        req.MOSI_DATA = mosi_data;
25        req.m_bit_number = bitstosent;
26        `uvm_send(req)
27    end
```

Figure 5.8: Master Sequence

5.4. Driver

Driver is responsible for sending created transactions to the DUT. Driver gets those transactions, changes them into bit-level actions, and drive the data into the DUT. For this project, two drivers are needed for master and slave respectively.

Master driver is responsible for generating a clock to start data transmission. For each SPI mode, master driver sends data from master to slave and disables and enables the clock and slave select in between.

```
1  class qspi_masterDriver extends uvm_driver #(qspi_transaction);
2      `uvm_component_utils(qspi_masterDriver);
3
4      virtual ifQspi      m_ifQspi;
5      qspiConfig          m_config;
6      qspi_transaction    m_qspi_transaction;
7
8      //// Class Constructor ////
9
10     function new(string name, uvm_component parent);
11         super.new(name, parent);
12     endfunction:new
13
14     //// Build Phase ///////
15
16
17     function void build_phase(uvm_phase phase);
18
19         uvm_config_db#(virtual ifQspi)::get(this, "", "m_ifQspi", m_ifQspi);
20
21     endfunction:build_phase
22
```

Figure 5.9: Master driver class constructor and build phase

In Figure 5.9, connections to interface, config class and transaction class are established. In build phase, uvm_config_db is used to establish the configuration for the virtual interface, this interface will be used in master driver to make connections to the interface signals as well as to the DUT.

```
23      //// driverOff to reset the Chip Select ////
24
25      task driverOff();
26
27          m_ifQspi.SS <= '1;
28          m_ifQspi.MOSI <= 'b0;
29          m_ifQspi.RST <= 'b0;
30      endtask:driverOff
31
32      ///// Master Driver's Clock ////
33
34      task driverClk;
35          m_ifQspi.RST=0;
36          m_ifQspi.externalClock <= 0;
37          #(m_config.clk_period*2*1ns) m_ifQspi.RST <= ~m_ifQspi.RST;
38          forever begin
39              #(m_config.clk_period*2ns) m_ifQspi.externalClock <= ~m_ifQspi.externalClock; end
40      endtask:driverClk
41
```

Figure 5.10: Master driver driverOff and driverClk functions

In Figure 5.10, driverOff and driverClk tasks are defined. driverOff task is used to reset the chip/slave select signal and is called at the end of the run phase, after the transaction transfer is done. driverClk task is the master driver's clock generator, it switches the RST and externalClock signals by waiting them 2ns times clk_period amount of time.

```
42      ////// Run Phase //////
43
44      task run_phase(uvm_phase phase);
45
46
47          fork
48              begin
49                  wait(m_ifQspi.RST);
50                  `uvm_info(get_type_name(), $psprintf("reset finished"), UVM_LOW)
51                  m_ifQspi.SS <='b1;
52
53                  if(m_config.m_cpol == 0)
54                      m_ifQspi.SCLK <='b0;
55                  else
56                      m_ifQspi.SCLK <='b1;
57
```

Figure 5.11: Master driver run phase

In Figure 5.11, run phase is started. Fork method is used in order to execute the functions under it parallelly. RST signal is waited and then slave select signal is enabled as default. For SCLK, if the CPOL is 0, then SCLK signal is low or 0 by default, if it is not, SCLK is high or 1.

```
58                forever begin
59                    seq_item_port.get_next_item(req);
60
61                    if (!$cast(m_qspi_transaction, req))
62                        `uvm_fatal(get_type_name(), "Failed to cast the requested transaction")
63
64                    data_transfer_master_driver;
65                    seq_item_port.item_done();
66                end
67            end
68            driverClk;
69        join_none
70    endtask:run_phase
71
```

Figure 5.12: Master driver run phase continued

In Figure 5.12, run phase continues as seq_item_port is used to start the communication and receive the transaction from sequencer by using get_next_item. data_transfer_master_driver function is called, that is the main function of the master driver and will be mentioned in the next figure. After all the transactions are received, item_done request is called on the seq_item_port to stop the communications. As mentioned before, driverClk task is called at the end of the run phase.

```
72    task data_transfer_master_driver();
73
74        case (m_config.m_cpol)
75            0: begin
76                case(m_config.m_cpha)
77                    0: begin
78                            @(posedge m_ifQspi.externalClock);
79                            m_ifQspi.SCLK <='b0;
80                            #1ns;
81                            m_ifQspi.SS <= 'b0;
82                            #1ns;
83                            for(int i = m_qspi_transaction.m_bit_number - 1; i >= 0; --i) begin
84                                m_ifQspi.MOSI <=m_qspi_transaction.MOSI_DATA[i];
85                                @(negedge m_ifQspi.externalClock);
86                                m_ifQspi.SCLK <='b1;
87                                #1ns;
88                                m_qspi_transaction.MISO_DATA[i] <= m_ifQspi.MISO;
89                                @(posedge m_ifQspi.externalClock);
90                                m_ifQspi.SCLK <='b0;
91                            end
92                            @(negedge m_ifQspi.externalClock);
93                            m_ifQspi.SS <= 'b1;
94                            #1ns;
95
96                    end
```

Figure 5.13: Master driver SPI mode 0

In Figure 5.13, data_transfer_master_driver task is defined. For each SPI mode, SCLK and SS signals are enabled and disabled with propagation delay, MOSI_DATA is written and MOSI_DATA is read. When CPOL is 0, SCLK and externalClock are not parallel, the behavior of these clocks will be seen in the simulation results chapter.

For SPI mode 0, CPOL and CPHA are 0. First, positive edge of the external clock is waited, as mentioned before, external clock and SCLK are opposite so at the positive edge of the external clock, SCLK is 0 or low. After that SS signal is brought down to low or 0, with that MOSI_DATA is sent. While in for loop, negative edge of the external clock is waited and SCLK is brought up to high or 1. Then, MISO_DATA is read. After the last data, positive edge of the external clock is waited once again in order to bring down the SCLK signal. Finally, in order to bring up the SS signal to high, negative edge of the external clock is waited. For SPI mode 0, MOSI_DATA is written and MISO_DATA is read.

```
97                      1: begin
98                              @(posedge m_ifQspi.externalClock);
99                              m_ifQspi.SCLK <='b0;
100                             #1ns;
101                             m_ifQspi.SS <= 'b0;
102                             #1ns;
103                             for(int i = m_qspi_transaction.m_bit_number - 1; i >= 0; --i) begin
104                                 @(negedge m_ifQspi.externalClock);
105                                 m_ifQspi.SCLK <='b1;
106                                 #1ns;
107                                 m_ifQspi.MOSI <= m_qspi_transaction.MOSI_DATA[i];
108                                 @(posedge m_ifQspi.externalClock);
109                                 m_ifQspi.SCLK <='b0;
110                                 #1ns;
111                                 m_qspi_transaction.MISO_DATA[i] <= m_ifQspi.MISO;
112                             end
113                             @(negedge m_ifQspi.externalClock);
114                             m_ifQspi.SS <= 'b1;
115                     end
116                 endcase
117             end
```

Figure 5.14: Master driver SPI mode 1

In Figure 5.14, SPI mode 1 can be seen. In SPI mode 1, CPOL is 0 and CPHA is 1. First, positive edge of the external clock is waited, as mentioned before, external clock and SCLK are opposite so at the positive edge of the external clock, SCLK is 0 or low. After that SS signal is brought down to low or 0. In SPI mode 1, data is not written at the positive edge of the external clock. For that, negative edge of the external clock is waited and SCLK is brought up to high or 1. After the propagation delay, MOSI_DATA is written. At the positive edge of the external clock, SCLK is brought down to low or 0 and MISO_DATA is read. In order to bring up the SS signal to high or 1, negative edge of the external clock is waited. For SPI mode 1, MOSI_DATA is written and MISO_DATA is read.

```
118                 1: begin
119                     case(m_config.m_cpha)
120                         0: begin
121                                 @(posedge m_ifQspi.externalClock);
122                                 m_ifQspi.SCLK <='b1;
123                                 #1ns;
124                                 m_ifQspi.SS <= 'b0;
125                                 #1ns;
126                                 for(int i = m_qspi_transaction.m_bit_number - 1; i >= 0; --i) begin
127                                     m_ifQspi.MOSI <= m_qspi_transaction.MOSI_DATA[i];
128                                     @(negedge m_ifQspi.externalClock);
129                                     m_ifQspi.SCLK <='b0;
130                                     #1ns;
131                                     m_qspi_transaction.MISO_DATA[i] <= m_ifQspi.MISO;
132                                     @(posedge m_ifQspi.externalClock);
133                                     m_ifQspi.SCLK <='b1;
134                                 end
135                                 @(negedge m_ifQspi.externalClock);
136                                 m_ifQspi.SS <= 'b1;
137                         end
```

Figure 5.15: Master driver SPI mode 2

In Figure 5.15, SPI mode 2 can be seen. In SPI mode 2, CPOL is 1 and CPHA is 0. As mentioned before, SCLK and external clock are now parallel since CPOL is 1. Positive edge of the external clock is waited and SCLK is brought up to high or 1. After that SS signal is brought down to low or 0 in order to start the writing of MOSI_DATA. At the negative edge of the external clock, SCLK is brought down to low or 0 and MISO_DATA is read. In order to bring up the SCLK to high, positive edge of the external clock is waited. After that, in order to bring up the SS to high, negative edge of the external clock is waited. And with that for SPI mode 2, MOSI_DATA is written and MISO_DATA is read.

```
138                         1: begin
139                                 @(posedge m_ifQspi.externalClock);
140                                 m_ifQspi.SCLK <='b1;
141                                 #1ns;
142                                 m_ifQspi.SS <= 'b0;
143                                 #1ns;
144                                 for(int i = m_qspi_transaction.m_bit_number - 1; i >= 0; --i) begin
145                                     @(negedge m_ifQspi.externalClock);
146                                     m_ifQspi.SCLK <='b0;
147                                     #1ns;
148                                     m_ifQspi.MOSI <= m_qspi_transaction.MOSI_DATA[i];
149                                     @(posedge m_ifQspi.externalClock);
150                                     m_ifQspi.SCLK <='b1;
151                                     #1ns;
152                                     m_qspi_transaction.MISO_DATA[i] <= m_ifQspi.MISO;
153                                 end
154                                 @(negedge m_ifQspi.externalClock);
155                                 m_ifQspi.SS <= 'b1;
156                         end
157                     endcase
158                 end
159             endcase
160         endtask
161 endclass
```

Figure 5.16: Master driver SPI mode 3

In Figure 5.16, SPI mode 3 can be seen. In SPI mode 3, CPOL and CPHA are both 1. Positive edge of the external clock is waited to bring up SCLK to high or 1 and bring down the SS to low or 0. Data reading or writing does not occur at positive edge of the external clock so negative edge of the external clock is waited. SCLK is brought down to low or 1 and MOSI_DATA is written. After that positive edge of the external clock is waited, SCLK is brought up to high or 1, and MISO_DATA is read. In order to complete the communication SS signal is brought up to high or 1 at the negative edge of the external clock. And with that, for all SPI modes, reading MISO_DATA and writing MOSI_DATA are achieved.

```
1  class qspi_slaveDriver extends uvm_driver #(qspi_transaction);
2      `uvm_component_utils(qspi_slaveDriver)
3  
4      virtual ifQspi      m_ifQspi;
5      qspiConfig          m_config;
6      qspi_transaction    m_qspi_transaction;
7  
8      //// Class Constructor ////
9  
10     function new(string name, uvm_component parent);
11         super.new(name, parent);
12     endfunction:new
13 
14     //// Build Phase ///////
15 
16 
17     function void build_phase(uvm_phase phase);
18 
19         uvm_config_db#(virtual ifQspi)::get(this, "", "m_ifQspi", m_ifQspi);
20 
21     endfunction:build_phase
22 
23 
```

Figure 5.17: Slave driver class constructor and build phase

Afterwards, the slave driver is created. In master driver, MISO_DATA is read. That MISO_DATA is written in slave driver. In Figure 5.17, same definitions and configurations from master driver can be seen.

```
24    ////// Run Phase //////
25
26    task run_phase(uvm_phase phase);
27        forever begin
28            seq_item_port.get_next_item(req); //get sequence item and send data to DUT
29
30            if (!$cast(m_qspi_transaction, req))
31                `uvm_fatal(get_type_name(), "Failed to cast the requested transaction")
32
33            sent_data_from_driver;
34            seq_item_port.item_done();
35        end
36    endtask:run_phase
37
```

Figure 5.18: Slave driver run phase

In Figure 5.18, at the run phase, transactions or sequence items are gathered from seq_item_port by the get_next_item function. Later, sent_data_from_driver task is called. When this task is completed, with the item_done function, driving data is done and finished.

```
43        if (!m_ifQspi.SS) begin
44        case (m_config.m_cpol)
45            0: begin
46                case(m_config.m_cpha)
47                    0: begin
48                        for(int i = m_qspi_transaction.m_bit_number - 1; i >= 0; --i) begin
49                            #(m_config.m_holdTime);
50                            m_ifQspi.MISO= m_qspi_transaction.MISO_DATA[i];
51                            @(posedge m_ifQspi.SCLK);
52                            #(m_config.m_holdTime);
53                            @(negedge m_ifQspi.SCLK);
54                        end
55                    end
56                    1: begin
57                        for(int i = m_qspi_transaction.m_bit_number - 1; i >= 0; --i) begin
58                            @(posedge m_ifQspi.SCLK);
59                            #(m_config.m_holdTime);
60                            m_ifQspi.MISO= m_qspi_transaction.MISO_DATA[i];
61                            @(negedge m_ifQspi.SCLK);
62                            #(m_config.m_holdTime);
63                        end
64                    end
65                endcase
66            end
```

Figure 5.19: Slave driver SPI mode 0 and 1

Like in the master driver, for each SPI mode, data is written in positive or negative edge of the clock, depending of the SPI mode. Each edge also has a delay even if they do not sent data in that case. For SPI mode 0, CPOL and CPHA are both 0. When SS is brought down to low, MISO_DATA is written from transaction class as can be seen in Figure 5.19. After that positive and negative edges of the SCLK are waited with delay time.

For SPI mode 1, CPOL is 0 and CPHA is 1. At the positive edge, after the delay, MISO_DATA is written from transaction class as can be seen in Figure 5.6c.

```
67            1: begin
68                case(m_config.m_cpha)
69                    0: begin
70                        for(int i = m_qspi_transaction.m_bit_number - 1; i >= 0; --i) begin
71                            #(m_config.m_holdTime);
72                            m_ifQspi.MISO= m_qspi_transaction.MISO_DATA[i];
73                            @(negedge m_ifQspi.SCLK);
74                            #(m_config.m_holdTime);
75                            @(posedge m_ifQspi.SCLK);
76                        end
77                    end
78                    1: begin
79                        for(int i = m_qspi_transaction.m_bit_number - 1; i >= 0; --i) begin
80                            @(negedge m_ifQspi.SCLK);
81                            #(m_config.m_holdTime);
82                            m_ifQspi.MISO= m_qspi_transaction.MISO_DATA[i];
83                            @(posedge m_ifQspi.SCLK);
84                            #(m_config.m_holdTime);
85                        end
86                    end
87                endcase
88            end
89        endcase
90        end
91    endtask
```

Figure 5.20: Slave driver SPI mode 2 and 3

For SPI mode 2, CPOL is 1 and CPHA is 0. When SS is brought down to low, MISO_DATA is written from transaction class as can be seen in Figure 5.20. After that positive and negative edges of the SCLK are waited with delay time.

For SPI mode 3, CPOL and CPHA are both 1. At the negative edge, after the delay, MISO_DATA is written from transaction class as can be seen in Figure 5.20.

Finally, both master and slave drivers are completed, all the modes are covered, data transfer from master to slave and from slave to master are both achieved.

5.5. Monitor

All configurations defined in master and slave driver are also defined in monitor class as can be seen Figure 5.21. In addition, a monitor port is defined from uvm_analysis_port. After all the MISO and MOSI data is read and obtained, this port will connect the monitor to the subscriber and or scoreboard.

```
class qspi_monitor extends uvm_monitor;
  `uvm_component_utils(qspi_monitor)


  virtual ifQspi m_ifQspi;

  uvm_analysis_port #(qspi_transaction) mon_port;
  qspi_transaction #(`DATAWIDTH,`ADDRWIDTH) m_qspi_transaction;
  qspiConfig m_config;

  //// Class Constructor ////

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


  ///// Build Phase /////

  function void build_phase (uvm_phase phase);
    mon_port= new("mon_port",this);

    uvm_config_db#(virtual ifQspi)::get(this, "", "m_ifQspi", m_ifQspi);

    endfunction:build_phase

```

Figure 5.21: Monitor class constructor and build phase

In Figure 5.22, run phase can be seen. In run phase, collect_transfer task is called. After that collect_transfer task is written. Transaction definitions are done and bit number is derived from config class.

```
  //////// Run Phase //////

  virtual task run_phase (uvm_phase phase);
    forever begin
        collect_transfer();
    end
  endtask


  task collect_transfer();
    qspi_transaction #(`DATAWIDTH,`ADDRWIDTH) m_qspi_transaction= qspi_transaction #(`DATAWIDTH,`ADDRWIDTH)::type_id::create("m_qspi_transaction")

    @(negedge m_ifQspi.SS);
    m_qspi_transaction.m_bit_number = m_config.m_bit_number;
    #(m_config.m_holdTime);

```

Figure 5.22: Monitor run phase and collect_transfer function

In Figure 5.23, data reading for SPI mode 0 and 1 is achieved. In order to start the data reading, SS needs to be low or 0. For SPI mode 0, CPOL and CPHA is 0. Positive edge of the SCLK is waited, MISO_DATA and MOSI_DATA is read. For SPI mode 1, CPOL is 0 and CPHA is 1. Negative edge of the SCLK is waited, MISO_DATA and MOSI_DATA is read.

```
43    if(!m_ifQspi.SS) begin
44        case (m_config.m_cpol)
45            0:begin
46                case(m_config.m_cpha)
47                    0:begin
48                        for(int i = m_qspi_transaction.m_bit_number - 1; i >= 0; --i) begin
49                            @(posedge m_ifQspi.SCLK);
50                            m_qspi_transaction.MISO_DATA[i]=m_ifQspi.MISO;
51                            m_qspi_transaction.MOSI_DATA[i]=m_ifQspi.MOSI;
52                        end
53                    end
54                    1:begin
55                        for(int i = m_qspi_transaction.m_bit_number - 1; i >= 0; --i) begin
56                            @(negedge m_ifQspi.SCLK);
57                            m_qspi_transaction.MISO_DATA[i]=m_ifQspi.MISO;
58                            m_qspi_transaction.MOSI_DATA[i]=m_ifQspi.MOSI;
59                        end
60                    end
61                endcase
62            end
```

Figure 5.23: Monitor SPI mode 0 and 1

In Figure 5.24, data reading for SPI mode 2 and 3 is achieved. For SPI mode 2, CPOL is 1 and CPHA 0. Negative edge of the SCLK is waited, MISO_DATA and MOSI_DATA is read. For SPI mode 3, CPOL and CPHA are both 1. Positive edge of the SCLK is waited, MISO_DATA and MOSI_DATA is read. At the end of the monitor class, with write function, read transactions are sent to the monitor's analysis port, mon_port.

```
63            1:begin
64                case(m_config.m_cpha)
65                    0:begin
66                        for(int i = m_qspi_transaction.m_bit_number - 1; i >= 0; --i) begin
67                            @(negedge m_ifQspi.SCLK);
68                            m_qspi_transaction.MISO_DATA[i]=m_ifQspi.MISO;
69                            m_qspi_transaction.MOSI_DATA[i]=m_ifQspi.MOSI;
70                        end
71                    end
72                    1:begin
73                        for(int i = m_qspi_transaction.m_bit_number - 1; i >= 0; --i) begin
74                            @(posedge m_ifQspi.SCLK);
75                            m_qspi_transaction.MISO_DATA[i]=m_ifQspi.MISO;
76                            m_qspi_transaction.MOSI_DATA[i]=m_ifQspi.MOSI;
77                        end
78                    end
79                endcase
80            end
81        endcase
82    end
83
84    mon_port.write(m_qspi_transaction);
85
86  endtask
87 endclass
```

Figure 5.24: Monitor SPI mode 2 and 3

5.6. Agent

While drivers can be selected and activated by using UVM_ACTIVE in one agent, it is better to create two agents for master and slave specifically. Master and slave agents are very similar, only difference is the connected driver. In Figure 5.25, master agent can be seen. An analysis port is defined for each agent. For master agent, master driver, monitor, sequencer as well as virtual interface and config class is defined.

```
1  class qspi_masterAgent extends uvm_agent;
2    `uvm_component_utils(qspi_masterAgent)
3
4    uvm_analysis_port#(qspi_transaction) masterAgent_port;
5
6    qspi_monitor monitor;
7    qspi_masterDriver masterDriver;
8    uvm_sequencer #(qspi_transaction) m_sequencer;
9    virtual ifQspi       m_ifQspi;
10   qspiConfig           m_qspiConfig;
11
12   /// Class Constructor ///
13
14   function new(string name,uvm_component parent);
15     super.new(name,parent);
16     m_qspiConfig = qspiConfig::type_id::create("m_qspiConfig");
17     m_qspiConfig.setConfig();
18   endfunction
19
20
```

Figure 5.25: Master agent class constructor and declarations

```
21   ////// Build Phase /////
22
23   function void build_phase(uvm_phase phase);
24
25     masterAgent_port=new("masterAgent_port", this);
26     monitor=qspi_monitor::type_id::create("monitor", this);
27     monitor.m_config = m_qspiConfig;
28     monitor.m_ifQspi = m_ifQspi;
29
30     if(is_active == UVM_ACTIVE) begin
31         m_sequencer=uvm_sequencer #(qspi_transaction)::type_id::create("m_sequencer",this);
32             masterDriver=qspi_masterDriver::type_id::create("masterDriver",this);
33             masterDriver.m_config = m_qspiConfig;
34             masterDriver.m_ifQspi = m_ifQspi;
35
36     end
37   endfunction: build_phase
38
39
```

Figure 5.26: Master agent build phase

In Figure 5.26, build phase is seen. Previously defined master agent port, config class, interface and monitor are registered. With UVM_ACTIVE, sequencer and master driver are registered.

```
40   //// Connect Phase /////
41
42
43   function void connect_phase(uvm_phase phase);
44     masterDriver.seq_item_port.connect(m_sequencer.seq_item_export); //connect m_sequencer to the driver
45     monitor.mon_port.connect(masterAgent_port); //// connect monitor to agent port
46
47   endfunction:connect_phase
48
49
50 endclass
```

Figure 5.27: Master agent connect phase

In Figure 5.27, connect phase is seen. In connect phase, sequencer and master driver are connected and also monitor's analysis port is connected to master agent's analysis port.

```
1  class qspi_slaveAgent extends uvm_agent;
2    `uvm_component_utils(qspi_slaveAgent)
3
4    uvm_analysis_port#(qspi_transaction) slaveAgent_port;
5
6    qspi_slaveDriver slaveDriver;
7    qspi_monitor monitor;
8
9    uvm_sequencer #(qspi_transaction) m_sequencer;
10   virtual ifQspi      m_ifQspi;
11   qspiConfig          m_qspiConfig;
12
13   /// Class Constructor ///
14
15   function new(string name,uvm_component parent);
16     super.new(name,parent);
17     m_qspiConfig = qspiConfig::type_id::create("m_qspiConfig");
18     m_qspiConfig.setConfig();
19   endfunction
20
21
```

Figure 5.28: Slave agent class constructor and declarations

In Figure 5.28, same definitions are made and an analysis port created for the slave agent.

```
22   ////// Build Phase /////
23
24   function void build_phase(uvm_phase phase);
25
26     slaveAgent_port=new("slaveAgent_port", this);
27     monitor=qspi_monitor::type_id::create("monitor", this);
28     monitor.m_config = m_qspiConfig;
29     monitor.m_ifQspi = m_ifQspi;
30
31     if(is_active == UVM_ACTIVE) begin
32         m_sequencer=uvm_sequencer #(qspi_transaction)::type_id::create("m_sequencer",this);
33             slaveDriver=qspi_slaveDriver::type_id::create("slaveDriver",this);
34             slaveDriver.m_config = m_qspiConfig;
35             slaveDriver.m_ifQspi = m_ifQspi;
36
37     end
38   endfunction: build_phase
39
40
```

Figure 5.29: Slave agent build phase

In Figure 5.29, registrations that are done in master agent, implemented for slave agent as well.

```
41    //// Connect Phase /////
42
43
44    function void connect_phase(uvm_phase phase);
45      slaveDriver.seq_item_port.connect(m_sequencer.seq_item_export); //connect m_sequencer to the driver
46      monitor.mon_port.connect(slaveAgent_port); //// connect monitor to agent port
47
48    endfunction:connect_phase
49
50
51  endclass
```

Figure 5.30: Slave agent connect phase

In Figure 5.30, connect phase is seen. In connect phase, sequencer and slave driver are connected and also monitor’s analysis port is connected to the slave agent’s analysis port.

5.7. Environment

For environment class, preexisting environment base class which is created by ANKASYS [45] for their VIP projects, is used. In this environment class, registration of DUT agents and subscriber are done in build phase. In connect phase, DUT agents are connected to the slave or master agents' analysis port and also to the subscriber's analysis port. DUT agents are also created by ANKASYS and act like the testbench agents, their responsibilities are identical.

In Figure 5.31, connections between DUT agents, master and slave agents and subscriber's master and slave write functions can be seen.

```
129    m_dut_agents.m_qspi_agent[0].slaveAgent_port.connect(m_qspi_subscriber_sample.analysis_imp_slave);
130    m_dut_agents.m_qspi_master_agent[0].masterAgent_port.connect(m_qspi_subscriber_sample.analysis_imp_master);
```

Figure 5.31: Environment connections

5.8. Subscriber

Multiple connections can be established, in this project, two agents and one monitor are created, both agents are connected to the subscriber via environment class and also monitor’s analysis port which includes write function is also connected. With write method, whenever there are transactions in the monitor’s analysis port and write method is called, all the components that are connected to the monitor’s analysis port can read the data. With that, subscriber’s write method is activated and received transactions are printed as MOSI_DATA and MISO_DATA as seen in Figure 5.32.

```
46 function void qspi_subscriber_sample::write_master(qspi_transaction t);
47     bit [11:0] slave_memory [5] = {12'h678, 12'hDEF, 12'hCCC, 12'h260, 12'h333};
48     if (!$cast(m_qspi_transaction_master, t))
49         return;
50     if(m_qspi_transaction_master.MISO_DATA==slave_memory[master_counter])
51         `uvm_info(get_full_name(), $psprintf("Transfer is successful"), UVM_LOW)
52         else
53             `uvm_error(get_full_name(), $psprintf("Transfer failed"))
54         master_counter++;
55 
56         if (master_counter == 5)
57             master_counter = 0;
58 
59     `uvm_info(get_full_name(), $psprintf("m_qspi_transaction_master.MISO_DATA = %h",m_qspi_transaction_master.MISO_DATA), UVM_LOW)
60     `uvm_info(get_full_name(), $psprintf("m_qspi_transaction_master.MOSI_DATA = %h",m_qspi_transaction_master.MOSI_DATA), UVM_LOW)
61 endfunction
62 
```

Figure 5.32: Subscriber write_master function

In Figure 5.32, write_master function can be seen. In this structure, design is the slave and testbench is the master. Subscriber checks if MISO_DATA matches the slave memory. If they match, “Transfer is successful” message is printed. Finally, both MISO and MOSI data are also printed.

```
63 function void qspi_subscriber_sample::write_slave(qspi_transaction t);
64     bit [11:0] master_memory [5] = {12'h345, 12'hABC, 12'hBBB, 12'h159, 12'h222};
65
66     if (!$cast(m_qspi_transaction_slave, t))
67         return;
68     if(m_qspi_transaction_slave.MOSI_DATA==master_memory[slave_counter])
69         `uvm_info(get_full_name(), $psprintf("Transfer is successful"), UVM_LOW)
70         else
71             `uvm_error(get_full_name(), $psprintf("Transfer failed"))
72         slave_counter++;
73
74         if (slave_counter == 5)
75             slave_counter = 0;
76
77     `uvm_info(get_full_name(), $psprintf("m_qspi_transaction_slave.MISO_DATA = %h",m_qspi_transaction_slave.MISO_DATA), UVM_LOW)
78     `uvm_info(get_full_name(), $psprintf("m_qspi_transaction_slave.MOSI_DATA = %h",m_qspi_transaction_slave.MOSI_DATA), UVM_LOW)
79 endfunction
```

Figure 5.33: Subscriber write_slave function

In Figure 5.33, write_slave function can be seen. In this structure, design is the master and testbench is the slave. Subscriber checks if MOSI_DATA matches the master memory. If they match, “Transfer is successful” message is printed. Finally, both MISO and MOSI data are also printed.

In this project, scoreboard class is not needed since both printing and comparing data are done in subscriber class.

5.9. Config

A configuration class is needed to store the bit definitions, period and delay definitions. Bit number amount is also defined in this class. In Figures 5.34 and 5.35, config class can be seen.

```
11 `ifndef qspiConfig
12 `define qspiConfig
13
14 class qspiConfig extends uvm_object;
15   bit m_cpol = 0;
16   bit m_cpha = 0;
17   int m_bit_number = `DEFAULT_DATA_TRANSFER_SIZE;
18   time m_holdTime = 60ns;
19   time clk_period = 10ns;
20   `uvm_object_utils_begin(qspiConfig)
21     `uvm_field_int(m_cpol,       UVM_ALL_ON)
22     `uvm_field_int(m_cpha,       UVM_ALL_ON)
23     `uvm_field_int(m_holdTime,  UVM_ALL_ON)
24     `uvm_field_int(m_bit_number, UVM_ALL_ON)
25     `uvm_field_int(clk_period, UVM_ALL_ON)
26   `uvm_object_utils_end
27
28   extern function new(string name = "qspiConfig");
29
30   extern virtual function void setConfig(
31     bit a_cpol = 0,
32     bit a_cpha = 0,
33     int a_bit_number = `DEFAULT_DATA_TRANSFER_SIZE,
34     time a_holdTime = 60ns,
35     time a_clk_period = 10ns
36     );
37 endclass : qspiConfig
```

Figure 5.34: Config class

```
38
39 function qspiConfig::new (string name = "qspiConfig");
40   super.new(name);
41 endfunction : new
42
43 function void qspiConfig::setConfig(
44   bit a_cpol = 0,
45   bit a_cpha = 0,
46   int a_bit_number = `DEFAULT_DATA_TRANSFER_SIZE,
47   time a_holdTime = 60ns,
48   time a_clk_period = 10ns
49   );
50
51   m_cpol = a_cpol;
52   m_cpha = a_cpha;
53   m_bit_number = a_bit_number;
54   m_holdTime = a_holdTime;
55   clk_period = a_clk_period;
56 endfunction : setConfig
57
58 `endif
```

Figure 5.35: Config class continued

5.10. Test

In test class, two sequences are in one test. In Figure 5.36, master and agent sequences are defined and registered in test class.

```
1 class qspi_test extends test_base;
2   `uvm_component_utils(qspi_test)
3
4   qspi_sequence seq;
5   qspi_sequence_master seq_master;
6
7
8   ///// Class Constructor ////
9
10  function new(string name, uvm_component parent);
11    super.new(name,parent);
12  endfunction
13
14  ///// Build Phase /////
15
16  function void build_phase(uvm_phase phase);
17    super.build_phase(phase);
18  endfunction
19
20  ///// Run Phase ////
21
22  task run_phase(uvm_phase phase);
23
24    phase.raise_objection(this);
25
26    seq =qspi_sequence::type_id::create("seq");
27    seq_master =qspi_sequence_master::type_id::create("seq_master");
28
```

Figure 5.36: Test class constructor and build phase

In Figures 5.37 and 5.38, run phase is seen. Inside the run phase, fork method is used so sequence transfers do not have to wait for each other. In Figure 5.37, 12 bits of data is sent from slave to master. With seq.miso_data lines, 12 bits of 5 data are defined, in this case they are ‘h111,’h222, ‘h333, ‘h000 and ‘h147. In this structure verification environment acts as slave and the design acts as master.

```
29        fork
30            repeat(1) begin
31                seq.randomize();
32                seq.bitstosent = 12;
33                seq.miso_data = 'h111;
34                seq.start(m_anka_env.m_env.m_dut_agents.m_qspi_agent[0].m_sequencer);
35                seq.miso_data = 'h222;
36                seq.start(m_anka_env.m_env.m_dut_agents.m_qspi_agent[0].m_sequencer);
37                seq.miso_data = 'h333;
38                seq.start(m_anka_env.m_env.m_dut_agents.m_qspi_agent[0].m_sequencer);
39                seq.miso_data = 'h000;
40                seq.start(m_anka_env.m_env.m_dut_agents.m_qspi_agent[0].m_sequencer);
41                seq.miso_data = 'h147;
42                seq.start(m_anka_env.m_env.m_dut_agents.m_qspi_agent[0].m_sequencer);
43            end
44
```

Figure 5.37: Test first sequence

```
45            repeat(1) begin
46                seq_master.randomize();
47                seq_master.bitstosent = 12;
48                seq_master.mosi_data = 'h444;
49                seq_master.start(m_anka_env.m_env.m_dut_agents.m_qspi_master_agent[0].m_sequencer);
50                seq_master.mosi_data = 'h555;
51                seq_master.start(m_anka_env.m_env.m_dut_agents.m_qspi_master_agent[0].m_sequencer);
52                seq_master.mosi_data = 'h666;
53                seq_master.start(m_anka_env.m_env.m_dut_agents.m_qspi_master_agent[0].m_sequencer);
54                seq_master.mosi_data = 'h777;
55                seq_master.start(m_anka_env.m_env.m_dut_agents.m_qspi_master_agent[0].m_sequencer);
56                seq_master.mosi_data = 'h888;
57                seq_master.start(m_anka_env.m_env.m_dut_agents.m_qspi_master_agent[0].m_sequencer);
58            end
```

Figure 5.38: Test second sequence

In Figure 5.38, verification environment acts as master and the design acts as slave. 12 bits of 5 data is sent from master to slave.

In Figure 5.2, master and slave memories are defined as variables. If design acts as slave, its response to mosi_data that is sent from test class, must be from slave memory. For example, ‘h444 is sent from master to slave, in this case design is the slave, its response must be from slave memory. First data in slave memory is the ‘h678, so master receives ‘h678 while sending ‘h444 to the slave. If design acts as master, master sends ‘h345 from master memory to slave and receives ‘h111 from slave.

5.11. Simulation and Test Results

In order to start the simulation for each mode, SPI mode needs to be configured both in test base class and design. In Figure 5.39, master and slave configuration are done separately. Under setConfig, first two numbers represent CPOL and CPHA respectively and 12 is the bit number. In this case, SPI mode 2 is activated, CPOL is 1 and CPHA is 0.

```
316 `ifdef USED_QSPI
317     function void test_base::do_qspi_config();
318         foreach (m_env_config.m_qspi_config[i]) begin
319             `ifdef USED_ANKA_VIP
320                 m_env_config.m_qspi_config[i].setConfig
321                     (1,
322                         0,
323                         12,
324                         1ns,
325                         10ns);
326             `endif
327         end
328 
329         foreach (m_env_config.m_qspi_master_config[i]) begin
330             `ifdef USED_ANKA_VIP
331             m_env_config.m_qspi_master_config[i].setConfig
332                 (1,
333                     0,
334                     12,
335                     1ns,
336                     10ns);
337             `endif
338         end
339     endfunction
```

Figure 5.39: Test base

In Figure 5.40, CPOL and CPHA configurations for SPI mode 0 can be seen. This configuration is done in loopback class of the design.

```
28 entity spi_loopback is
29     Generic (
30         N : positive := 12;
31         CPOL : std_logic := '0';
32         CPHA : std_logic := '0';
33         PREFETCH : positive := 2;
34         SPI_2X_CLK_DIV : positive := 5
35         );
```

Figure 5.40: Design loopback

After mode selection, it is time to verify the design. All signals of design that are defined in top block are added to the wave. In Figure 5.41, added signals for SPI mode 0 can be seen. In this structure, testbench is the slave and design is the master. m_di_o represents the MISO data. These MISO data are sent by the testbench in test class. m_do_o represents the master memory and it is a response to the MISO data. In Figure 5.41, m_di_i is ‘ABC and m_do_o is ‘222. That is the expected result, since ‘222 is the second sequence that is sent in test class and ‘ABC is in the second place of the master memory.

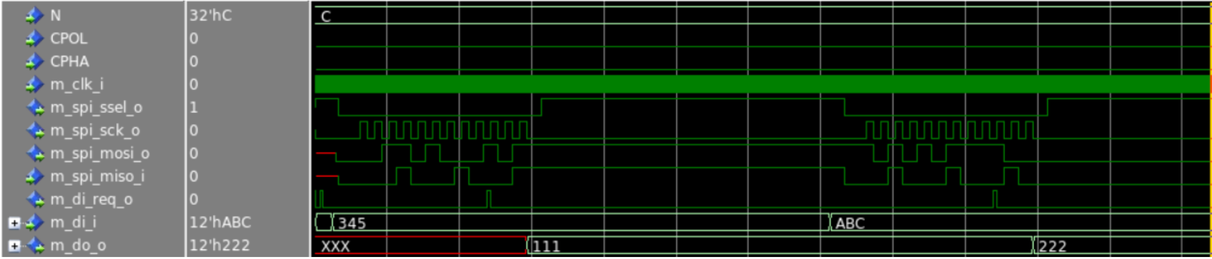

Figure 5.41: SPI mode 0, design=master testbench=slave

In Figure 5.42, testbench is the master and the design is the slave. m_di_o represents the MOSI data. These MOSI data are sent by the testbench in test class. m_do_o represents the slave memory and it is a response to the MOSI data. In Figure 5.42, m_di_i is ‘DEF and m_do_o is ‘555. That is the expected result, since ‘555 is the second sequence that is sent in test class and ‘DEF is in the second place of the slave memory. It has a slight shift in the simulation but console results show that the transfer is successful and communication between master and slave is achieved.

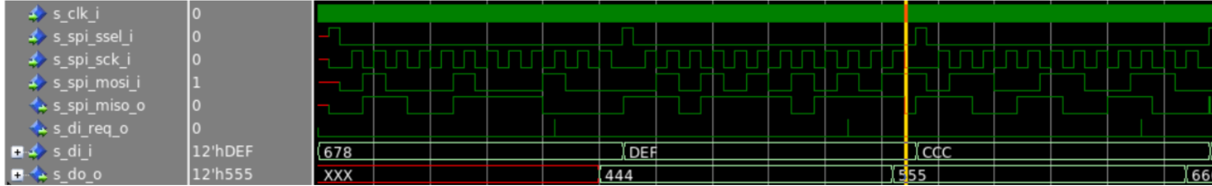

Figure 5.42: SPI mode 0, design=slave testbench=master

Results are verified with the help of subscriber class, as Figure 5.43 and Figure 5.44 show, there is no error and all data transfer between master and slave is successful.

```
nka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification,
nka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave MISO_DATA = 111
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification,
nka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave MOSI_DATA = 345
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification,
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification,
nka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification,
nka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave MISO_DATA = 222
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification,
nka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave MOSI_DATA = abc
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification,
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification,
nka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification,
nka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave MISO_DATA = 333
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification,
nka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave MOSI_DATA = bbb
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification,
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification,
anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification
anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MISO_DATA = 000
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification
anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MOSI_DATA = 159
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification,
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification,
anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification,
anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave MISO_DATA = 147
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification
anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave MOSI_DATA = 222
```

Figure 5.43: SPI mode 0, design=master testbench=slave results

```
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MISO_DATA = 678
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MOSI_DATA = 444
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VII
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MISO_DATA = def
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MOSI_DATA = 555
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VII
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MISO_DATA = ccc
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MOSI_DATA = 666
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VII
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MISO_DATA = 260
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MOSI_DATA = 777
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VII
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MISO_DATA = 333
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MOSI_DATA = 888
```

Figure 5.44: SPI mode 0, design=slave testbench=master results

In Figure 5.45, added signals for SPI mode 1 can be seen. In this structure, testbench is the slave and design is the master. m_di_o represents the MISO data. These MISO data are sent by the testbench in test class. m_do_o represents the master memory and it is a response to the MISO data. In Figure 5.45, m_di_i is ‘345 and m_do_o is ‘111. That is the expected result, since ‘111 is the first sequence that is sent in test class and ‘345 is in the first place of the master memory.

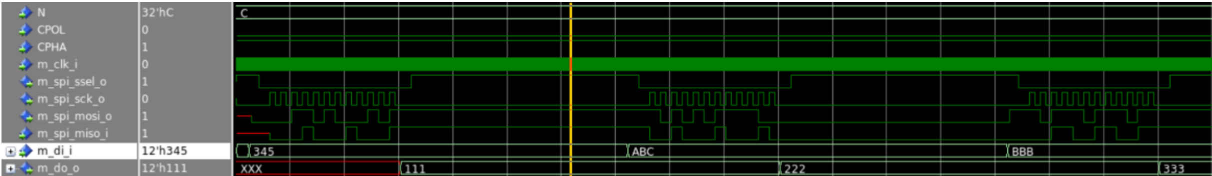

Figure 5.45: SPI mode 1, design=master testbench=slave

In Figure 5.46, testbench is the master and the design is the slave for SPI mode 1. m_di_o represents the MOSI data. These MOSI data are sent by the testbench in test class. m_do_o represents the slave memory and it is a response to the MOSI data. In Figure 5.46, m_di_i is ‘678 and m_do_o is ‘444. That is the expected result, since ‘444 is the first sequence that is sent in test class and ‘678 is in the first place of the slave memory.

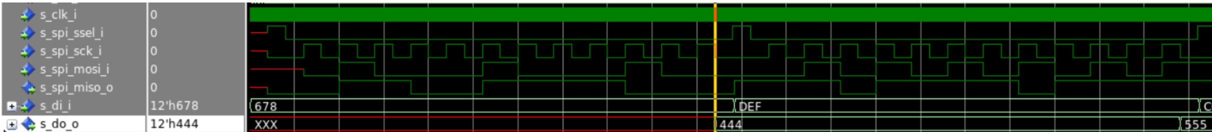

Figure 5.46: SPI mode 1, design=slave testbench=master

Results are verified with the help of subscriber class, as Figure 5.47 and Figure 5.48 show, there is no error and all data transfer between master and slave is successful.

```
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MISO_DATA = 111
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MOSI_DATA = 345
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MISO_DATA = 222
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MOSI_DATA = abc
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MISO_DATA = 333
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MOSI_DATA = bbb
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MISO_DATA = 000
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MOSI_DATA = 159
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MISO_DATA = 147
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MOSI_DATA = 222
```

Figure 5.47: SPI mode 1, design=master testbench=slave results

```
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
.m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MISO_DATA = 678
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MOSI_DATA = 444
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP/QSP
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MISO_DATA = def
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MOSI_DATA = 555
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP/QSP
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MISO_DATA = ccc
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MOSI_DATA = 666
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP/QSP
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MISO_DATA = 260
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MOSI_DATA = 777
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP/QSP
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MISO_DATA = 333
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MOSI_DATA = 888
```

Figure 5.48: SPI mode 1, design=slave testbench=master results

In Figure 5.49, added signals for SPI mode 2 can be seen. In this structure, testbench is the slave and design is the master. m_di_o represents the MISO data. These MISO data are sent by the testbench in test class. m_do_o represents the master memory and it is a response to the MISO data. In Figure 5.49, m_di_i is ‘345 and m_do_o is ‘111. That is the expected result, since ‘111 is the first sequence that is sent in test class and ‘345 is in the first place of the master memory

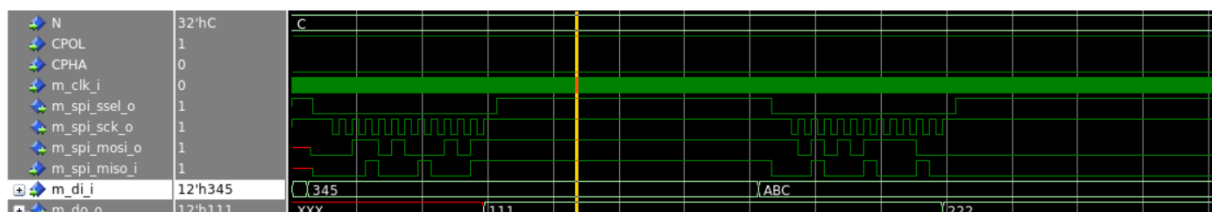

Figure 5.49: SPI mode 2, design=master testbench=slave

In Figure 5.50, testbench is the master and the design is the slave for SPI mode 2. m_di_o represents the MOSI data. These MOSI data are sent by the testbench in test class. m_do_o represents the slave memory and it is a response to the MOSI data. In Figure 5.50, m_di_i is ‘678 and m_do_o is ‘444. That is the expected result, since ‘444 is the first sequence that is sent in test class and ‘678 is in the first place of the slave memory.

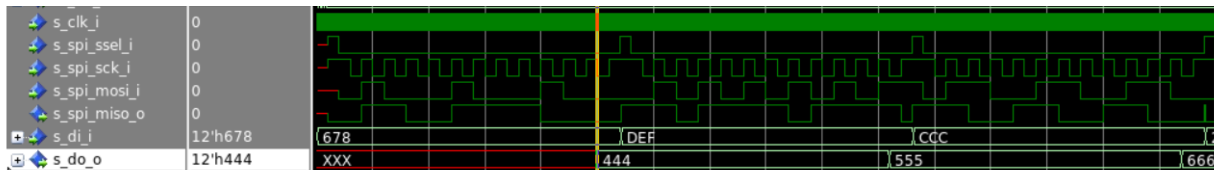

Figure 5.50: SPI mode 2, design=slave testbench=master

Results are verified with the help of subscriber class, as Figure 5.51 and Figure 5.52 show, there is no error and all data transfer between master and slave is successful.

```
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Du
_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_q
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MISO_DATA = 111
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Du
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MOSI_DATA = 345
# UVM_INFO /storage/projects/verification_projects/santez/santez_q
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Du
_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_q
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MISO_DATA = 222
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Du
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MOSI_DATA = abc
# UVM_INFO /storage/projects/verification_projects/santez/santez_q
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Du
_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_q
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MISO_DATA = 333
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Du
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MOSI_DATA = bbb
# UVM_INFO /storage/projects/verification_projects/santez/santez_q
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Du
m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qs
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MISO_DATA = 000
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Du
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MOSI_DATA = 159
# UVM_INFO /storage/projects/verification_projects/santez/santez_qs
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Du
m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qs
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MISO_DATA = 147
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Du
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MOSI_DATA = 222
```

Figure 5.51: SPI mode 2, design=master testbench=slave results

```
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SC
.m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SC
.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MISO_DATA = 678
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SC
.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MOSI_DATA = 444
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP/QS
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SC
p.m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SC
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MISO_DATA = def
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SC
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MOSI_DATA = 555
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP/QS
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SC
p.m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SC
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MISO_DATA = ccc
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SC
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MOSI_DATA = 666
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP/QS
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SC
p.m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SC
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MISO_DATA = 260
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SC
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MOSI_DATA = 777
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP/QS
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SC
p.m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SC
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MISO_DATA = 333
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SC
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master.MOSI_DATA = 888
```

Figure 5.52: SPI mode 2, design=slave testbench=master results

Finally, in Figure 5.53, added signals for SPI mode 3 can be seen. In this structure, testbench is the slave and design is the master. m_di_o represents the MISO data. These MISO data are sent by the testbench in test class. m_do_o represents the master memory and it is a response to the MISO data. In Figure 5.53, m_di_i is ‘BBB and m_do_o is ‘333. That is the expected result, since ‘333 is the third sequence that is sent in test class and ‘BBB is in the third place of the master memory

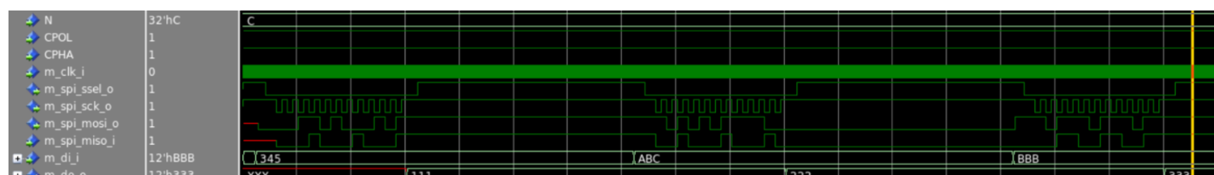

Figure 5.53: SPI mode 3, design=master testbench=slave

In Figure 5.54, testbench is the master and the design is the slave for SPI mode 2. m_di_o represents the MOSI data. These MOSI data are sent by the testbench in test class. m_do_o represents the slave memory and it is a response to the MOSI data. In Figure 5.54, m_di_i is 'CCC and m_do_o is '666. That is the expected result, since '666 is the third sequence that is sent in test class and 'CCC is in the third place of the slave memory.

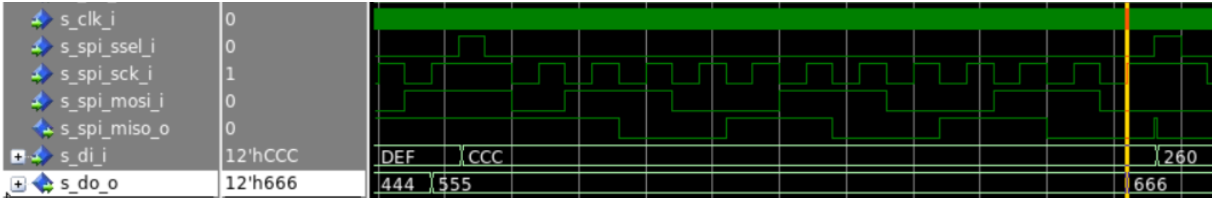

Figure 5.54: SPI mode 3, design=slave testbench=master

Results are verified with the help of subscriber class, as Figure 5.55 and Figure 5.56 show, there is no error and all data transfer between master and slave is successful.

```
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/S(
_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/S(
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MISO_DATA = 111
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/S(
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MOSI_DATA = 345
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP/QS
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/S(
_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/S(
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MISO_DATA = 222
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/S(
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MOSI_DATA = abc
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP/QS
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/S(
_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/S(
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave MISO_DATA = 333
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/S(
_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave MOSI_DATA = bbb
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP/QS
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/S(
m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/S(
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MISO_DATA = 000
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/S(
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MOSI_DATA = 159
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP/QS
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/S(
m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/S(
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MISO_DATA = 147
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/S(
m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_slave.MOSI_DATA = 222
```

Figure 5.55: SPI mode 3, design=master testbench=slave results

```
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
.m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master MISO_DATA = 678
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master MOSI_DATA = 444
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP/QSP
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master MISO_DATA = def
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master MOSI_DATA = 555
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP/QSP
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master MISO_DATA = ccc
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master MOSI_DATA = 666
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP/QSP
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master MISO_DATA = 260
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master MOSI_DATA = 777
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/VIP/QSP
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] Transfer is successful
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master MISO_DATA = 333
# UVM_INFO /storage/projects/verification_projects/santez/santez_qspi/verification/Duv/SCO
p.m_anka_env.m_env.m_qspi_subscriber_sample] m_qspi_transaction_master MOSI_DATA = 888
```

Figure 5.56: SPI mode 3, design=slave testbench=master results

6. REALISTIC CONSTRAINTS AND CONCLUSIONS

Verification of a digital circuit design is crucial in microelectronic industry. It is a challenge for engineers, designing an electronic system is already a complicated process and with the verification necessity, it becomes a burden. But verification can not be ignored, since to be able to correct and validate a design before its release is essential. And with this verification process, quality of the design is significantly increased.

For this objective, a verification environment is created by using SystemVerilog and UVM. Then, an SPI Master/Slave interface is put under test. This interface is already created, since designing and verification operations are both equally complicated. This SPI design is used as an IP, and with the created verification environment, this whole structure becomes a VIP. The design is tested and compared with the expected results. With the acquired results, VIP is completed and the design is successfully verified with the created verification environment.

6.1 Practical Application of this Project

This project successfully implements a VIP for a relatively complex SPI interface. By analyzing the design, both master and slave interfaces, connections are made in verification environment, this created structure can be reusable and adaptable for different designs. Connection ports can be changed in the verification environment accordingly to the design. Main purpose of the UVM is reusability, so in this project this task is achieved.

6.2 Realistic Constraints

Most important aspect of this project is the reusability. For SPI protocol, many different designs can be verified by using this VIP. With small changes, time and resource efficiency are achieved. And because it is separate process from designing, both design and verification engineers work very efficiently and they can be very productive this way. Design engineers do not have to use their time to validate their design and can focus on only designing process.

6.2.1 Social, environment and economic impact

Time and resources are crucial aspects in our lives. In order to save time and resources, work sharing is done in businesses. Each individual has its own responsibilities, resources and time period to do the tasks. With this project, efficiency is achieved, since designing and verification processes are both time and resource consuming. Design engineers do not have to lost in the design by checking for small mistakes and finding them. Verification engineers can simply put the design under test and point the mistakes to the design engineers. That way both time and resources are saved and it has a big economic impact on the microelectronic industry.

6.2.2 Cost analysis

This design can be implemented and verified on a FPGA board. This FPGA board's cost is the only vital cost factor of this project. Normally, QuestaSim simulation tool licence needs to be purchased but since ANKASYS helps the project with their licenses and tools, this aspect is not a cost factor.

6.2.3 Standards

For implementing the verification environment, SystemVerilog and UVM standards are followed. Connection between design and verification environment is done accordingly to SPI protocol, since design is an SPI interface. Also, the engineering code of conduct is adopted in this project.

6.2.4 Health and safety concerns

Creating this project does not possess any danger to any human. There is no health or safety risks in the development process.

6.3 Future Work and Recommendations

In this project, default SPI protocol is followed. Default SPI protocol is a full-duplex model, that means communication is established on 2 lines. In the future, this verification environment can be altered to establish the communication on 4 lines. That means changing the default duplex mode to quadruple mode or changing SPI to QSPI. Furthermore, this verification environment can be altered for the I2C or UART communication protocols. I addition to one slave interface, more than one slaves can be attached to master, since SPI protocol allows multiple slaves. Finally, the design and verification environment can be implemented to a FPGA board to verify the functionality of the design.

REFERENCES

[1] Vasudevan, S. (2020). Practical UVM: Step by step with IEEE 1800.2. R. R. Bowker.

[2] Frenzel, L. (2015). Handbook of serial communications interfaces: A comprehensive compendium of serial digital input/Output (I/o) standards. Newnes.

[3] Thomas, D. E., Donald E., T., & Moorby, P. R. (1998). The Verilog® hardware description language. Springer.

[4] Sutherland, S., Davidmann, S., & Flake, P. (2006). SystemVerilog for design second edition: A guide to using SystemVerilog for hardware design and modeling. Springer Science & Business Media.

[5] Questa® advanced simulator. (n.d.). Siemens EDA is a leader in electronic design automation-Siemens EDA.

Retrieved January 23, 2021, from https://www.mentor.com/products/fv/questa/

[6] DVKit. (n.d). Retrieved January 23, 2021, from https://dvkit.sourceforge.net/

[7] Ciletti, M. D. (2011). Advanced digital design with the Verilog HDL.

[8] Original-E: Foundations for social virtual realities. (n.d.). Welcome to ERights.Org. Retrieved January 23, 2021, from https://erights.org/history/original-e/index.html

[9] Vera. (2019,). Semiconductor Engineering. Retrieved January 23, 2021, from https://semiengineering.com/knowledge_centers/languages/vera/

[10] Glasser, M. (2009). Open verification methodology cookbook. Springer.

[11] Synopsys (2021). | EDA Tools, Semiconductor IP and Application Security Solutions. Retrieved January 23, 2021, from https://www.synopsys.com/

[12] *Cadence*. (n.d.). Cadence | Computational Software for Intelligent System Design™. Retrieved January 23, 2021, from https://www.cadence.com/en_US/home.html

[13] Simulation & verification. (n.d.). Siemens EDA is a leader in electronic design automation- Siemens EDA.

Retrieved January 23,2021, from https://www.mentor.com/products/fpga/verification-simulation/

[14] Paret, D., & Fenger, C. (1997). The I2C bus: From theory to practice. John Wiley & Son.

[15] Osborne, A., & Bunnell, D. (1982). An introduction to microcomputers. Osborne/McGraw-Hill.

[16] ModelSim®. (n.d.). Siemens EDA is a leader in electronic design automation - Siemens EDA.

Retrieved January 23, 2021, from https://www.mentor.com/products/fpga/verification-simulation/modelsim/

[17] Wang, J., & Tepfenhart, W. (2019). Formal methods in computer science. CRC Press.

[18] Simpson, P. A. (2015). FPGA design: Best practices for team-based reuse. Springer.

[19] RAM. (2020). Computer Hope's Free Computer Help. https://www.computerhope.com/jargon/r/ram.htm

[20] Chapter 2 – Defining a verification environment. (n.d.). https://colorlesscube.com/uvm-guide-for-beginners/chapter-2-defining-the-verification-environment/

[21] Uvm_object. (n.d.). Verification Academy - The most comprehensive resource for verification training. | Verification Academy. Retrieved January 23, 2021, from https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1c/html/files/base/uvm_object-svh.html

[22] Uvm_component. (n.d.). Verification Academy - The most comprehensive resource for verification training. | Verification Academy. Retrieved January 23, 2021, from https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1b/html/files/base/uvm_component-svh.html

[23] SystemVerilog interface. (n.d.). ChipVerify. Retrieved January 23, 2021, from https://www.chipverify.com/systemverilog/systemverilog-interface

[24] SystemVerilog interface construct. (2020). Verification Guide. Retrieved January 23, 2021, from https://verificationguide.com/systemverilog/systemverilog-interface-construct/

[25] Chapter 4 – Sequences and sequencers. (n.d.). Pedro Araújo – Random thoughts about hardware design. Retrieved January 23, 2021, from https://colorlesscube.com/uvm-guide-for-beginners/chapter-4-transactions-sequences-and-sequencers/

[26] UVM sequences and transactions application. (2015). Universal Verification Methodology. Retrieved January 23, 2021, from https://www.learnuvmverification.com/index.php/2015/07/29/uvm-sequences-and-transactions-application/

[27] UVM sequence. (2020). Verification Guide. Retrieved January 23, 2021, from https://verificationguide.com/uvm/uvm-sequence/

[28] Chapter 5 – Driver. (n.d.). Pedro Araújo – Random thoughts about hardware design. Retrieved January 23, 2021, from https://colorlesscube.com/uvm-guide-for-beginners/chapter-5-driver/

[29] SystemVerilog TestBench. (2020). Verification Guide. Retrieved January 23, 2021, from https://verificationguide.com/systemverilog-examples/systemverilog-testbench-example-01/

[30] Chapter 6 – Monitor. (n.d.). Pedro Araújo – Random thoughts about hardware design. Retrieved January 23, 2021, from https://colorlesscube.com/uvm-guide-for-beginners/chapter-6-monitor/

[31] UVM TestBench architecture. (2020). Verification Guide. Retrieved January 23, 2021, from https://verificationguide.com/uvm/uvm-testbench-architecture/

[32] Chapter 7 – Agent. (n.d.). Pedro Araújo – Random thoughts about hardware design. Retrieved January 23, 2021, from https://colorlesscube.com/uvm-guide-for-beginners/chapter-7-agent/

[33] UVM agent. (2020). Verification Guide. Retrieved January 23, 2021, from https://verificationguide.com/uvm/uvm-agent/

[34] UVM environment [uvm_env]. (n.d.). ChipVerify. Retrieved January 23, 2021, from https://www.chipverify.com/uvm/uvm-environment

[35] UVM environment example. (2020). Verification Guide. Retrieved January 23, 2021, from https://verificationguide.com/uvm/uvm-environment-example/

[36] Uvm_subscriber. (n.d.). Verification Academy - The most comprehensive resource for verification training. | Verification Academy. Retrieved January 23, 2021, from https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1c/html/files/comps/uvm_subscriber-svh.html

[37] Dhaker, P. (n.d.). Introduction to SPI interface. Retrieved January 23, 2021, from https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html

[38] SPI communications – Slave core VHDL. (n.d.). Daniel Álvarez's Blog | Robotics, Electronics and some Open Source Weblog. Retrieved January 23, 2021, from https://dani.foroselectronica.es/spi-communications-slave-core-vhdl-137/

[39] Basics of the SPI communication protocol. (2018). Circuit Basics. Retrieved January 23, 2021, from https://www.circuitbasics.com/basics-of-the-spi-communication-protocol/

[40] UVM scoreboard. (n.d.). ChipVerify. Retrieved January 24, 2021, from https://www.chipverify.com/uvm/uvm-scoreboard

[41] UVM test. (2020). Verification Guide. Retrieved January 24, 2021, from https://verificationguide.com/uvm/uvm-test/

[42] Replace behavioral DUT with AXI-based RTL DUT in UVM test bench. (n.d.). MathWorks. Retrieved January 24, 2021, from https://es.mathworks.com/help/hdlverifier/ug/uvm_replace_behavioral_dut_with_axi_dut.html;jsessionid=4fe8ad05c385418434977bfe9dd6

[43] From UVM to VUnit: Test benches in VHDL | ITDev. (n.d.). ITDev |. Retrieved January 24, 2021, from https://www.itdev.co.uk/blog/uvm-vunit-test-benches-vhdl

[44] Uvm tlm. (n.d.). ChipVerify. Retrieved January 25, 2021, from https://www.chipverify.com/uvm/tlm-preface

[45] ANKASYS. (n.d.). Retrieved January 28, 2021, from https://ankasys.com/

CURRICULUM VITAE

Name Surname : Berkay Turgay

Place and Date of Birth : Eminönü / 12.04.1997

E-mail : berkayturgay1@gmail.com

Berkay Turgay finished primary and high school in Istanbul. He is currently a senior year student at Electronics and Communication Engineering in Istanbul Technical University Electrical-Electronics Faculty. He completed his internship in ANKASYS.