University of Oulu

DEGREE PROGRAMME IN ELECTRICAL ENGINEERING

MASTER'S THESIS

MULTIPURPOSE SYNTHESIZABLE SYSTEMVERILOG SPI-BUS PROTOCOL VERIFICATION SYSTEM

Author	Markku Raappana
Supervisor	Jukka Lahti
Second Examiner	Juha Häkkinen
Technical advisor	Teemu Sirviö

December 2016

Raappana M. (2016) Multipurpose Synthesizable SystemVerilog Spi-Bus Protocol Verification System. University of Oulu, Degree Programme in Electrical Engineering. Master's Thesis, 50p.

ABSTRACT

The complexity of System-on-a-Chip (SoC) is continuing to increase due to the shrinking die size, increase in the number of sub-modules, power efficiency, performance, higher functionality and used protocols. This has an impact on the verification process related to the overall design process.

For the verification process, there are commercial products that can be applied in order to verify and test certain Intellectual Properties (IP) but also platforms that lack these tools. This thesis focuses on the issue where a system has to be constructed that helps the verification and testing process of a data bus protocol used by the Device Under Testing (DUT).

The study of the Serial Peripheral Interface (SPI) gives the examples of some issues that can be faced during applying this data bus protocol to any given system.

Different kinds of testing and verifying methods are addressed in order to show what the tools can be when applying new DUT to a system or examining the data bus protocol it uses.

The flow of a design process is studied by showing the iterations of a particular system that was to be created to meet the need that were introduced while examining the issues relating this subject. This flow can be said to start from ground level and end to the final iteration where the system could be created from.

The functionality and structure of a Multipurpose Verification System that was created during this thesis are explained. The proofing process of this system is showed by examining the simulation and synthesis reports.

The outcomes and future development ideas are discussed as well. This thesis showed that the study in hand has benefits to Nokia as the applying company and the system could be added to the company tool library after modifying it to be used as a stand-alone IP.

Key words: verification, synthesis, prototyping, SPI, SystemVerilog, testing.

Raappana M. (2016) Monikäyttöinen syntetisoituva SystemVerilog SPI-väyläprotokollan verifiointijärjestelmä. Oulun yliopisto, sähkötekniikan koulutusohjelma. Diplomityö, 50s.

TIIVISTELMÄ

Järjestelmäpiirien (SOC) kompleksisuus on jatkuvassa kasvussa johtuen johdinpiirien pienenemisestä, alijärjestelmien määrän kasvusta, vaatimuksista tehonkulutuksessa, suorituskyvyn kasvusta, toiminnallisuuden kasvusta ja käytettävistä protokollista. Näillä attribuuteilla on vaikutusta kokonaissuunnitteluprosessin verifiointiprosessiin.

Verifiointiprosessiin on tällä hetkellä saatavilla kaupallisia sovelluksia, joita voidaan hyödyntää kolmannen osapuolen suunnitteleman systeemin testaukseen ja verifioitiin. Samalla on myös olemassa testausalustoja, joista nämä sovellukset puuttuvat. Tämä diplomityö keskittyy tilanteeseen, jossa täytyy rakentaa systeemi joka helpottaa testattavassa laitteessa (DUT) käytettävän tietoväyläprotokollan verifiointi- ja testausprosessia.

Serial Peripheral Interface (SPI)-väyläprotokollan tutkimus tuo esiin esimerkkejä, joihin voidaan törmätä, kun kyseistä väyläprotokollaa käytetään missä tahansa sitä hyödyntävässä systeemissä.

Diplomityössä tutkitaan erilaisia testaus- ja verifiointimetodeja, jotta voidaan osoittaa mitä erilaisia työkaluja voidaan hyödyntää, kun uusi testattava laite lisätään olemassa olevaan systeemiin tai tutkitaan tämän käyttämää väyläprotokollaa.

Kokonaissuunnitteluprosessia on tutkittu esittelemällä tietyn järjestelmän iteraatiovaiheita, joka kehitettiin ratkaisemaan aiemmin tarkasteltuja, tähän aiheeseen liittyviä ongelmia. Suunnitteluprosessin voidaan katsoa alkaneen tilanteesta, jossa mitään konkreettista ei ollut vielä valmiina ja päättyvän tilanteeseen jossa järjestelmän viimeinen iteraatio voitiin alkaa konkretisoimaan.

Monikäyttöisen syntetisoituvan verifiointijärjestelmän funktionaalisuus ja rakenne esitellään. Tutkimalla simulointi- ja synteesiraportteja näytetään tämän järjestelmän varmennusprosessi.

Diplomityön toteutumista ja tulevaisuuden jatkokehitysideoista keskustellaan. Tämä diplomityö osoittaa, että Nokia soveltajayrityksenä pystyy hyödyntämään tämän tutkielman lopputulemia. Lisäksi työn tulokset voidaan lisätä, modifioinnin jälkeen, yrityksen komponenttikirjastoon toimimaan itsenäisenä instanssina.

Avainsanat: verifiointi, synteesi, prototypointi, SPI, SystemVerilog, testaus.

TABLE OF CONTENTS

ABSTRACT
TIIVISTELMÄ
TABLE OF CONTENTS
FOREWORD
LIST OF ABBREVIATIONS AND SYMBOLS
1. INTRODUCTION 7
2. SERIAL PERIPHERAL INTERFACE 9
   2.1. Serial Peripheral Interface 9
   2.2. SPI structure 9
   2.3. Modes 13
3. TESTING AND VERIFICATION 17
   3.1. Testbench 17
   3.2. Prototyping 18
   3.3. Design checks based on assertions 20
4. DEVELOPMENT OF THE MULTIPURPOSE VERIFICATION SYSTEM 23
   4.1. Ground Zero 23
   4.2. Multipurpose Verification System 24
   4.3. Hardware versus Software 25
   4.4. General Purpose Model and Softcore 26
   4.5. Final Approach 27
   4.6. Related Solutions 29
5. SYNTHESIZING AND FPGA PROTOTYPING 31
   5.1. Declaring the Designs Main and Submodules 31
   5.2. Verification 33
   5.3. Synthesis 35
6. DISCUSSION 36
7. SUMMARY 38
8. REFERENCES 39
9. APPENDICES 42

FOREWORD

This Master's Thesis was done for the Nokia Mobile Network SoC Prototyping and Qualification team during the autumn 2016. The aim of this work was to create a synthesizable SPI-bus protocol testing and verification system that could be used in different platforms, i.e in simulator-, emulator- and FPGA-platforms.

I would like to thank my Technical supervisors M.Sc. Teemu Sirviö and M.Sc. Esa-Matti Turtinen for taking time to listen and evaluating the original idea for this thesis and giving me the opportunity to do this thesis. The appreciation also goes to Nokia Mobile Networks that gave me the possibility and resources to actualize this thesis. I would also like to give my gratitude to M.Sc. Aku Mikkonen and B.Sc. Ari Hautala who had the necessary knowledge, insight, expertize and patience to help me during this process. Thank you Dr. Jukka Lahti for giving me advice and guidance and for supervising this thesis.

The last but not the least, kudos and love go to my family just for being there for me.

Oulu December 2016

Markku Raappana

LIST OF ABBREVIATIONS AND SYMBOLS

ACE	Advanced Extensible Interface Coherency Extensions
AIP	Assertive Intellectual Property
AIP	Assertion Based Intellectual Property Checker
AMBA	Advanced Microcontroller Bus Architecture
ASIC	Application-Specific Integrated Circuit
AXI4-lite	Advanced Extensible Interface 4 Lite
BRAM	Block Random Access Memory
CAN	Controller Area Network
CPHA	Clock Phase
CPOL	Clock Polarity
CPU	Central Processing Unit
DORD	Data Order
DUT	Device under Testing
EDA	Electronic Design Automation
EEPROM	Electrically Erasable Programmable Read-Only Memory
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
I^2C	Inter-Integrated Circuit
IP	Intellectual Property
LSB	Least Significant Bit
LUT	Look up Table
MSB	Most Significant Bit
MVS	Multipurpose Verification System
OOP	Object Oriented Programming
PCB	Printed Circuit Board
RAM	Random Access Memory
RTL	Register-Transfer Language
SOC	System on Chip
SPI	Serial Peripheral Interface
SVA	SystemVerilog Assertion
UVM	Universal Verification Methodology
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VIP	Verification Intellectual Property
mW	Milliwatt

1. INTRODUCTION

While the designing of a System-on-a-Chip (SoC) becomes more and more complicated, because the number of gates is increasingly getting bigger and bigger as predicted by Moore's law [1], the importance of the system verification is evenly increasing. The overall challenges are getting more complex due to the shrinking die size, increase in the number of sub-modules, power efficiency, performance, higher functionality and used protocols.[2]

An American freethinker H. L. Menchen once said in the 1910's, ”There is always a well-known solution to every human problem─neat, plausible and wrong.”.[3] When this thesis was started, it appeared that this aphorism hold the very truth for the problem how to ensure that the bus protocol used holds its specification where it cannot be ensured with tools available?

The problem arose while examining an entity, using the Serial Peripheral Interface (SPI) that did not have any counterpart to test it against to. While appearing completely valid, the data transmitted via this entity, did not hold the specification when used in a proper context. At the time, an ad hoc solution was used in a form of adding an external system to the design in hand. This solution seemed to be useful in this case but a need for more robust, versatile, visible and generic solution was clearly seen.

A system that could be used to monitor and verify the used bus protocol in different platforms would be the most suited for these kind of situations. The problem is shown in Figure 1, where the bus protocol used between the Central Processing Unit (CPU) and the Device under Testing (DUT) is verified but the bus protocol between the DUT and the Verification Intellectual Property (VIP) is not.

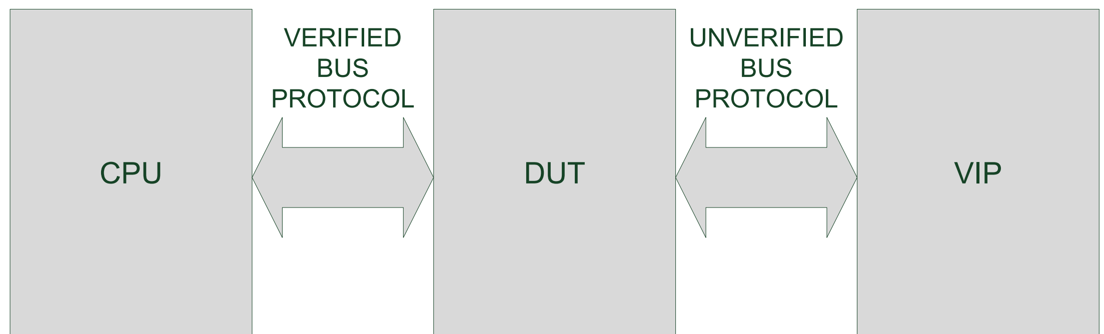


Figure 1. A graphical presentation of a bus verification problem.

It was studied if existing commercial solutions could be used such as a VIP, an Assertive Intellectual Property (AIP) or a bus protocol analyzer but they were limited to be used in specific platforms and didn't hold all the functionality needed. The AIP and bus protocol analyzer would analyze the data bus itself but do not take a stand what happens on the other end of the bus. A need for company specific tool was inevitable and the company needs in its repertoire a tool that can make this analysis at the receiving end. Thus ,the quest of a multipurpose tool was started.

A plausible solution was manifested when SystemVerilog Assertions (SVA) were introduced to the author of this thesis. The SystemVerilog had evolved via iterations

to a point where it could be reasonably utilized to solve some aspects of the given problem.[4][5][6][7][8]

The goal was to design a system that could be used to verify the used bus protocol with a reasonable degree of certainty with selected preconditions. The system would have to be designed so that it could be fitted to be used in simulation, emulation as well as in circuit implementation by logic synthesis. In this work it was studied what is the most suitable approach for creating the previously mentioned system. The system configuration and functionality are studied and they are iterated to meet the specification. The SPI bus protocol characteristics are explained and the main testing and verification methods are addressed. Chapter 2 is dedicated to explaining to the reader how the SPI bus protocol is constructed and used. The theory gives to the reader basic understanding why the bus has to be verified and monitored. Also the reasons for a potential error are explained.

The reader is familiarized with the commonly used testing and verification methods in Chapter 3. This chapter also explains some of the issues when applying different kinds of methods to the existing DUT. In Chapter 4 it is explained what was the path when the systems configuration was studied. All the iterations are explained to the reader. The ideas behind these iterations are addressed, as well as the reasons why they were discarded or taken to a further study. Chapter 5 explains to the reader how the final iteration was created using SystemVerilog and how it was simulated and tested in order to verify its functionality. In Chapters 6 and 7, it is discussed what were the outcomes of this work. It is also addressed what could be the further ideas for developing the system and what could be the alternative approaches towards to the original problem.

2. SERIAL PERIPHERAL INTERFACE

This chapter is designated to SPI. The basic functionality of this bus protocol is explained and the structure of this bus protocol is shown.

2.1. Serial Peripheral Interface

Serial Peripheral Interface, or SPI for short, was originally introduced by the Motorola Company in the late 1980's. The reason for adapting this new protocol was to replace parallel interfaces inside in any suitable system. Using this serial interface instead of parallel interfaces would inevitably reduce the volume of wires needed for the routing without reducing the data transfer speeds.[9]

The simplicity of the interfacing and the theoretical as well as concrete data transfer speeds made SPI easy to adopt throughout the electric industry. Currently SPI can be stated to be a *de facto* standard in short distance data transfer primarily in embedded systems.[9]

The loosely standardized protocol gives the user, in a default version, a full duplex communication with flexible bit range and with low power consumption. The synchronous data transfer is using only one clock provided by the SPI-master unit so the SPI-slave units can be designed without expensive precision oscillators. However, this does not apply to units that can act both SPI-master and –slave.

Even though the SPI is a widely used protocol, there is no official formal, separate specification of the SPI-bus. The specification can be obtained from separate documents provided by the component vendors that use the SPI in their products.[9]

2.2. SPI structure

The SPI was introduced in its default form as a four-wire serial communication interface. These wires included all the information that the devices needed for the device-to-device communication between the SPI-master and -slave. The signal wires include SCKL- (Master Clock), MOSI- (Master out Slave In), MISO- (Master in Slave Out) and SS-signal (Slave Select) in the default form. Any system using the SPI-protocol can be implemented with additional wiring for extra signals; however these wires are not part of the SPI-protocol but can be implemented if needed.[9]

The SPI-master is acting always in the dominant role during the communication. The master decides when and what the slave component will be addressed in order to start the synchronized communication by enabling the slave select signal for that particular device. The slave select is usually used as zero-active-signal. Due to the loose standardization of the SPI-bus protocol, it can be said that usually the slave select signal is pulled low in this addressing state of the communication. This method releases the system from using unit addresses when addressing the SPI-slave counterparts. In theory, the number of slaves is not limited in any way. The negative side of this method is that every SPI-slave has to have its own slave select line if more than one slave is driven by the SPI-master shown in Figure 2.[10]

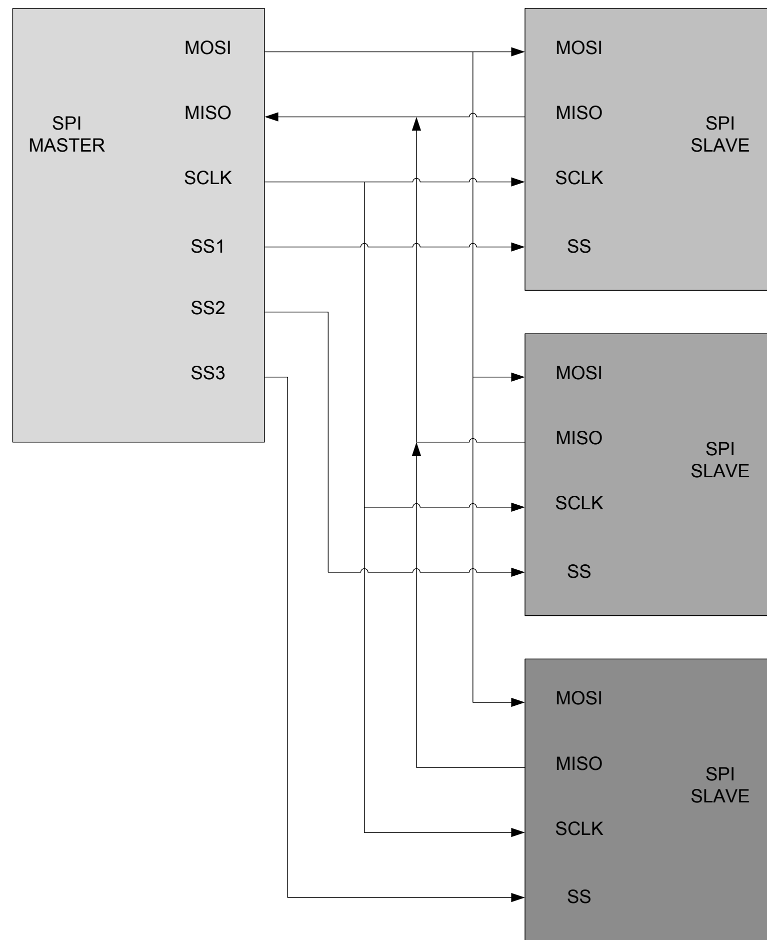


Figure 2. SPI implementation.

SPI uses pull-up and pull-down resistors in signal lines in order to gain their idle status. Depending on the configuration, either logical 1 or 0 is then considered as the active state. If the slave-select signal is not in an active state, all the output signal lines for the SPI-slaves are driven to a high impedance state. Due to this, all the slaves can use the same signal lines, because a separate slave unit does not have an effect on the signal line status. Since the SPI is considered as default, a full-duplex synchronized communication protocol, the transmission state, is divided using the opposite edges of the clock signal between the SPI-master and –slave. In practice, the system is using shift registers in order to reach this functionality, as shown in Figure 3.[9][10]

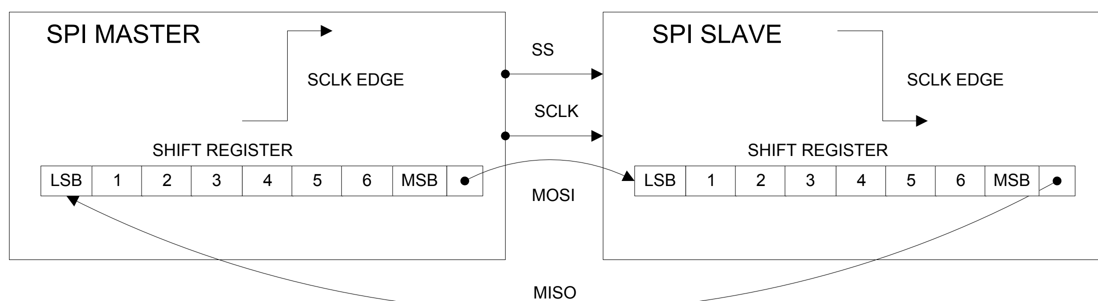


Figure 3. SPI shift register hardware setup for master-slave transaction.

However, the full-duplex communication does not include any form of acknowledgement between the counterparts. This can be said to be built-in functionality of the SPI-protocol. After the transmission is started, by activating the slave select signal, the SPI-master sends the data Most Significant Bit (MSB) bit firsts to the SPI-slave shift registers Least Significant Bit (LSB), evicting or shifting that bit. This starts the transmission in the SPI-slaves shift register where the MSB-bit is sent to the SPI-master shift register LSB, evicting that bit or shifting it. This when Data Order 0 (DORD) is in use. If the MSB and LSB order is reversed, the DORD is considered to be 1. Even if the SPI-slave is transmitting data towards the SPI-master, the SPI-master may ignore this transmission. This is not problematic per se since the master may ignore the incoming transmission if it does not affect the SPI-master's functionality. This functionality can be utilized when the SPI-slaves are used in daisy chain-configuration shown in Figure 4.[9][10]

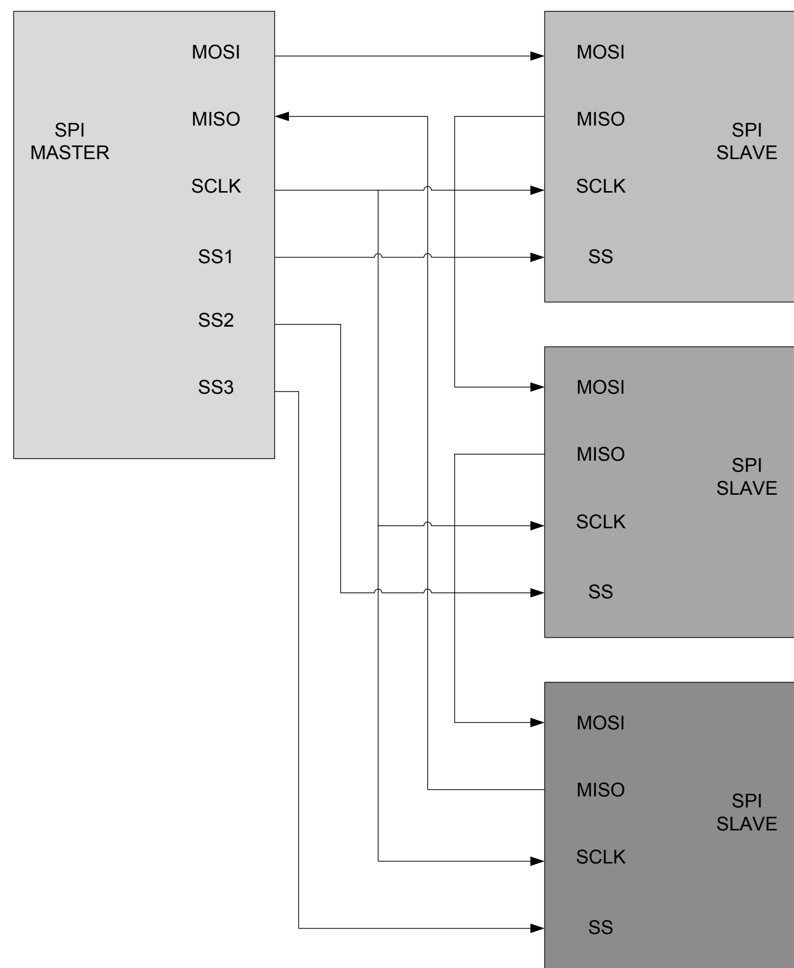


Figure 4. SPI daisy chain-implementation

However, the possibility to ignore the transmission in either direction can be problematic because neither of the counterparts do not have the knowledge whether the transmission is received or not. The "in-build" lack of acknowledgement makes the system faster but vulnerable to making a transmission to the void. This is one the main issues that is addressed in this thesis in coming chapters.[9]

The SPI-protocol is highly flexible regarding the length of the data word, transmission length and transmission periodization. The new features such as quad- or double-protocol even broaden the spectrum of these parameters. In Figure 5, it is shown how MOSI-signal (PIN0) and the following MISO-signals (PIN 1 to 3) are behaving during a read operation. In the extended-protocol SPI-master is addressing the slave first via MOSI-line (PIN0) and receiving the response via MISO-line (PIN1) in consecutive order. In the dual- and quad-protocol, all the lines are used by partitioning both excitation and response evenly between the signal lines.[11]

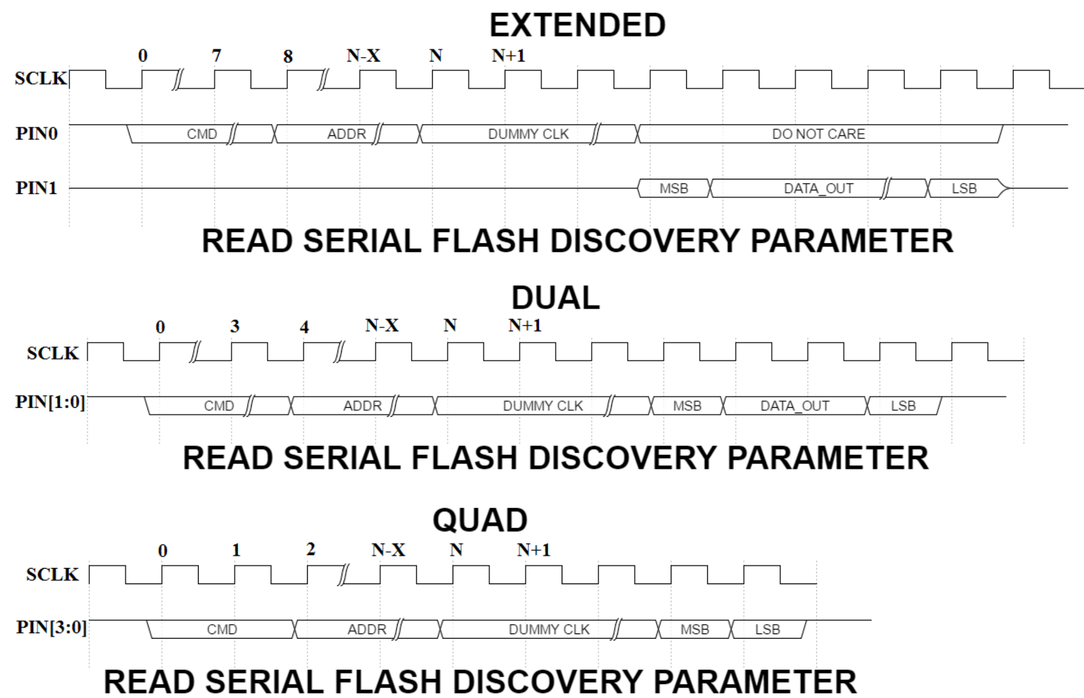


Figure 5. An example from extended-, dual and quad-protocol serial read operation.

In Figure 5, the slave-signal is not shown. Where the extended protocol is using two unidirectional lines the dual and quad are using bidirectional lines for data transfer. For the data transfer length calculations in clock cycles, the following formulas can be used. For the extended protocol, Formula (1), for the dual protocol, Formula (2) and for the quad protocol, Formula (3) respectively.

$$N - x = 7 + (Address[max] + 1) \tag{1}$$

$$N - x = 3 + \frac{3 + (Address[max] + 1)}{2} \tag{2}$$

$$N - x = 3 + \frac{1 + (Address[max] + 1)}{4} \tag{3}$$

The SPI-protocol in its default form uses 8-bit data or byte manifold as the base length in transmissions. However, this is not limiting any system to use a different kind of widths in data words. Because the basic communication relies on use of shift

registers, both of the instances using the SPI-bus, should be using the same size shift registers and the protocol should also support that size. Whereas this feature might help the hardware usage in some extension, it also makes the system testing more complicated.[10]

2.3. Modes

In section 2.2, it was mentioned that the transmission between SPI-master and –slave could be problematic due the lack of acknowledgement. This can be due to the modes that the instances are using. By default, devices have to use the same mode in order to be able to communicate with each other. The SPI-protocol is using four different kind of modes. The modes differ from each other depending on the Clock Phase (CPHA) and Clock Polarity (CPOL) they are using. An additional feature is DORD that defines whether the MSB or LSB is sent first. These combinations are shown in Figure 6 and in Figure 7, respectively.[9][12]

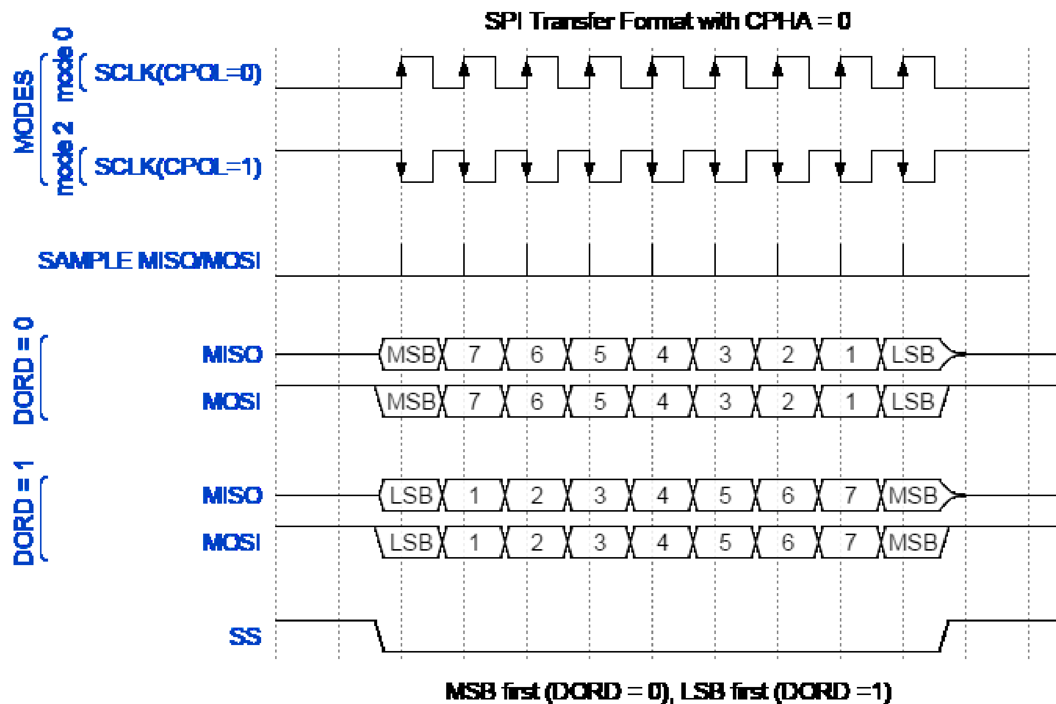


Figure 6. SPI-bus protocol modes 0 and 2 with CPHA 0. The slave select is zero active.

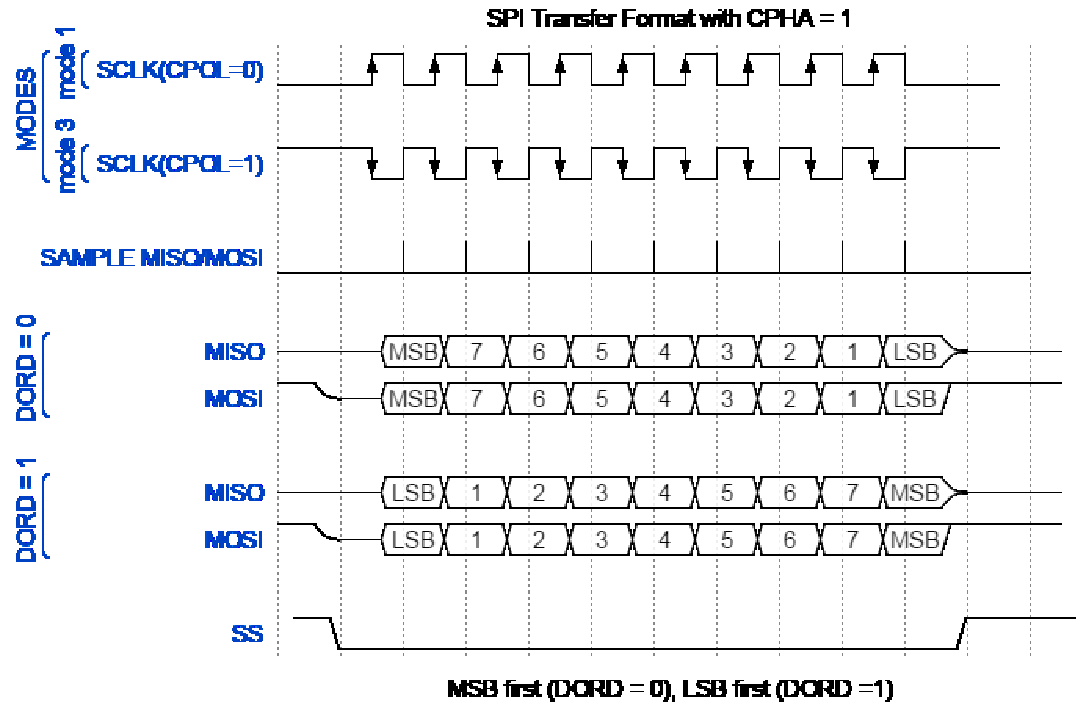


Figure 7. SPI-bus protocol modes 1 and 3 with CPHA 1. The slave select is zero active.

The CPOL defines the idle and active state of the clock. The CPHA defines which edge of the clock phase is used to sample the signal. These parameters set the number of modes to four as explained in Table 1 and Figure 8.

Table 1. CPOL and CPHA functionality in different SPI-modes

	Leading Edge	**Falling Edge**	**SPI Mode**
CPOL = 0, CPHA = 0	Sample (Rising)	Setup (Falling)	0
CPOL = 0, CPHA = 1	Setup (Rising)	Sample (Falling)	1
CPOL = 1, CPHA = 0	Sample (Falling)	Setup (Rising)	2
CPOL = 1, CPHA = 1	Setup (Falling)	Sample (Rising)	3

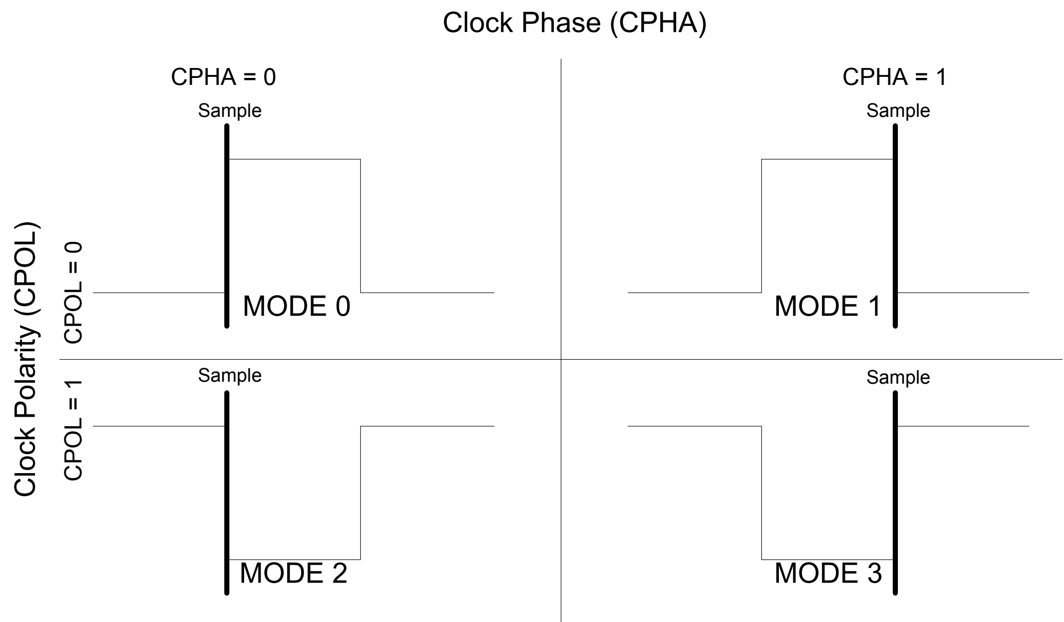


Figure 8. The four SPI-modes corresponding to the states of CPHA and CPOL parameters.

If the DORD is taken into account, the number of unofficial modes rises to eight. This results in a situation where the designer has to have encompassing knowledge of the functionality of the instances used. This will be addressed more thoroughly in section 4.1.

The mode dictates the basic functionality of the SPI-instance. This means that the received data is always literally interpreted by the receiving counterpart. The ambiguity of the data is not taken into any account. This subject is addressed in more details in coming sections.

Usually the modes can be changed for the master when initializing the instance. The SPI-slave components are on the other hand more or less static when it comes to the mode selection. This means that the SPI-master component has to adapt to the used protocol between SPI-master and –slave. Because there is no standardized hand-shaking-protocol for the SPI-bus, the initial hand-shaking is made via trial-and-error if the mode used is not known or additional signaling is configured to co-exist aside the SPI-bus.[13] As mentioned earlier, the data from SPI-bus is interpreted literally by the receiver, the received data can cause the SPI-instance to react to the incoming data. The response on the other hand might not be as desired. This is why the mode selection is vital and has to be the same for both instances in order to avoid plausible communication.

To clarify the last clause in the previous sentence, it has to be understood that when the data is received it will be translated by the receiver. If the translation is then in any executable form, the decoded instruction (previously referred to as data) will be executed. The execution is the literal result of the interpreted instruction. If the interpretation is close enough to the “right one”, the executed instruction might result in a response that is plausible but not right. The following analogue could explain the situation. When asked ”Name a domesticated feline?”, if the answer was “A tiger.” the answer would be plausible but wrong. This kind of an error could be spotted, for

example, from thoroughly examining the response from waveforms or using design language specific property checks and assertions. If the designers do not use these kinds of methods but rather use assumptions, such as “If a response is gained, then the system is working.”, the plausible functionality stays and the wrong mode might be left unfixed. This subject will be addressed more thoroughly in Chapter 4.

3. TESTING AND VERIFICATION

The good design practice indicates that in order to pass design criteria a new design has to be tested and verified.[14] The design has the desired properties only in paper if it is not tested in any way. This is a common problem when new designs are created. If a new Intellectual Property (IP) is created but there is no counterpart to test it against to, the IP's functionality can be only guessed.[15] This can be solved either by using testbenches with verification IP's, real components or using logic analyzers. Each of these has their own advantages and disadvantages.

In frontend testing, the designer has to evaluate what is the most efficient way to make these tests in order to see how the design is behaving. The following sub sections try to give some points of view what to expect when using these different techniques.

3.1. Testbench

A common way to test the new design is to test it in a testbench that is created specifically to fit the design in hand. This practice can be utilized by using highly complex Hardware Description Language (HDL) structures. A commonly used HDL language is Universal Verification Methodology (UVM). A testbench topology created with UVM is shown in Figure 9.

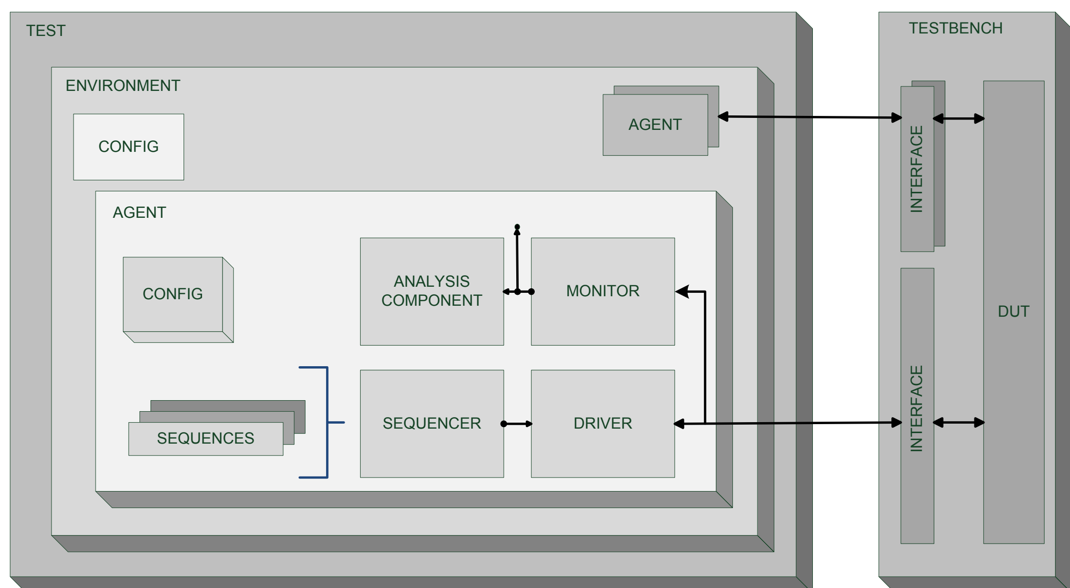


Figure 9. UVM-testbench layout.

The other way to proceed is by feeding the DUT's inputs with desired data and monitoring the outputs in a simple manner, as shown in Figure 10.[16][17]

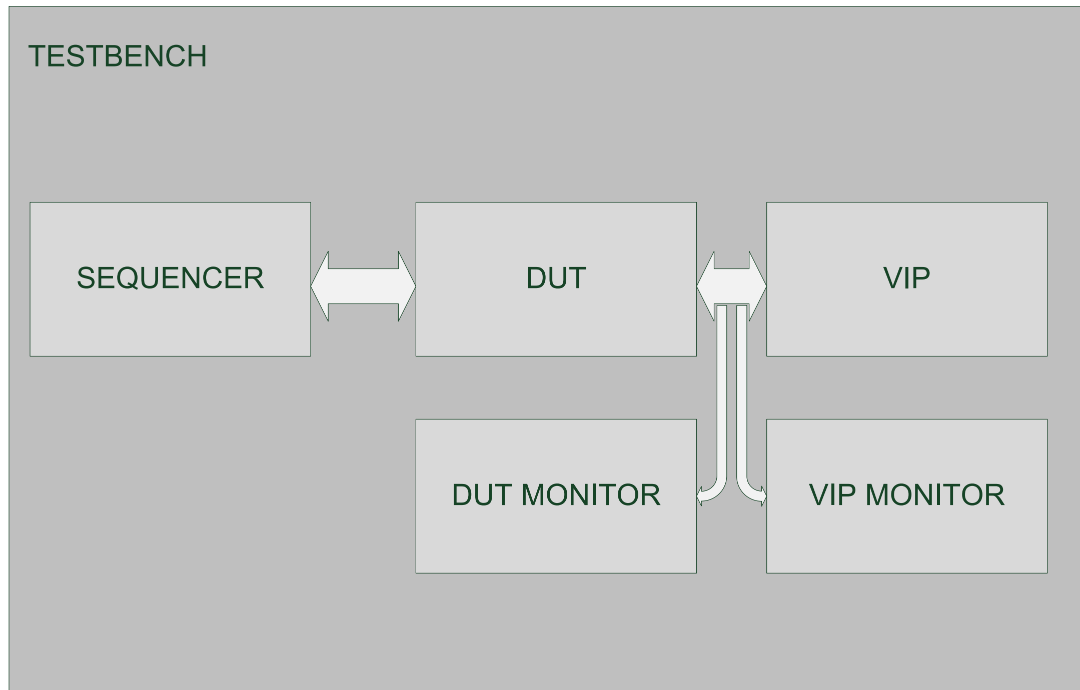


Figure 10. Simple HDL-testbench layout.

These testbenches can usually be used in simulations and emulators. If these were to be used in Field Programmable Gate Array (FPGA) environment, the RTL/HDL has to be fully synthesizable or some actions have to be done in order to implement the design to the FPGA, i.e. the use of transactors and other instances alike is vital. The transactor is a component that actively interacts with the DUT interfaces. The testbenches are configurable and can be fed with large data patterns in order to cover all the corner cases inside the system.

The creation of the testbench can be even more time consuming than the DUT design. So in order to use testbenches, the designer has to be sure that all the specifications around the DUT are valid. Usually testbenches are created separately from the DUT. This is advisable in order to reduce the error margin.

The testbench can be fitted to include automatic evaluating properties, aka assertions, that indicate if the functionality of the DUT is compromised due inconsistencies in internal or external signals. Assertions however can be only utilized in simulations and emulations this is due assertions do not compile in synthesis tools. Adding assertions to the RTL design is highly advisable and using SVA is an ever increasing method in RTL designing.[18]

Using testbenches gives many possibilities to the designer to check the functionality of the system via tools provided by the simulator, emulator and third party vendor analyzing tools.

3.2. Prototyping

The prototyping process can include implementing highly complex devices in to the design. If the interoperability is to be tested using real components, there is a need to

minimize the possible problems that might occur while the SOC is in ramp up-stage with the attached components.

Using actual components such as Electrically Erasable Programmable Read-Only Memories (EEPROM) driven with SPI- or Inter-Integrated Circuit (I^2C) –busses can be justifiable when time is of essence and the component can be attached to the system with relative ease. This might be the situation where parts of the design can be driven separately using emulator or FPGA. This method is even recommendable if it enables the usage of the same components that are present in the final product. Usually the vendors provide accurate and specific datasheets that help the designer to use the DUT as it should in a real situation.[19]

The downside is the need for PCB's or equivalent systems to implement these components to the existing design shown in Figure 11. If the bring-up connection board can be designed along the new design, it is very usable throughout the design flow. This could prevent problems that are caused by thermal, connectivity, mechanical or electrical mismatches between the used design and the implemented circuits. This approach to testing and verifying the DUT does not exactly follow the main principles in the good design practice and is considered to be more of a shortcut or ad-hoc than a proper testing practice if it doesn't follow the given design rules and specification regarding the design in hand.[19]

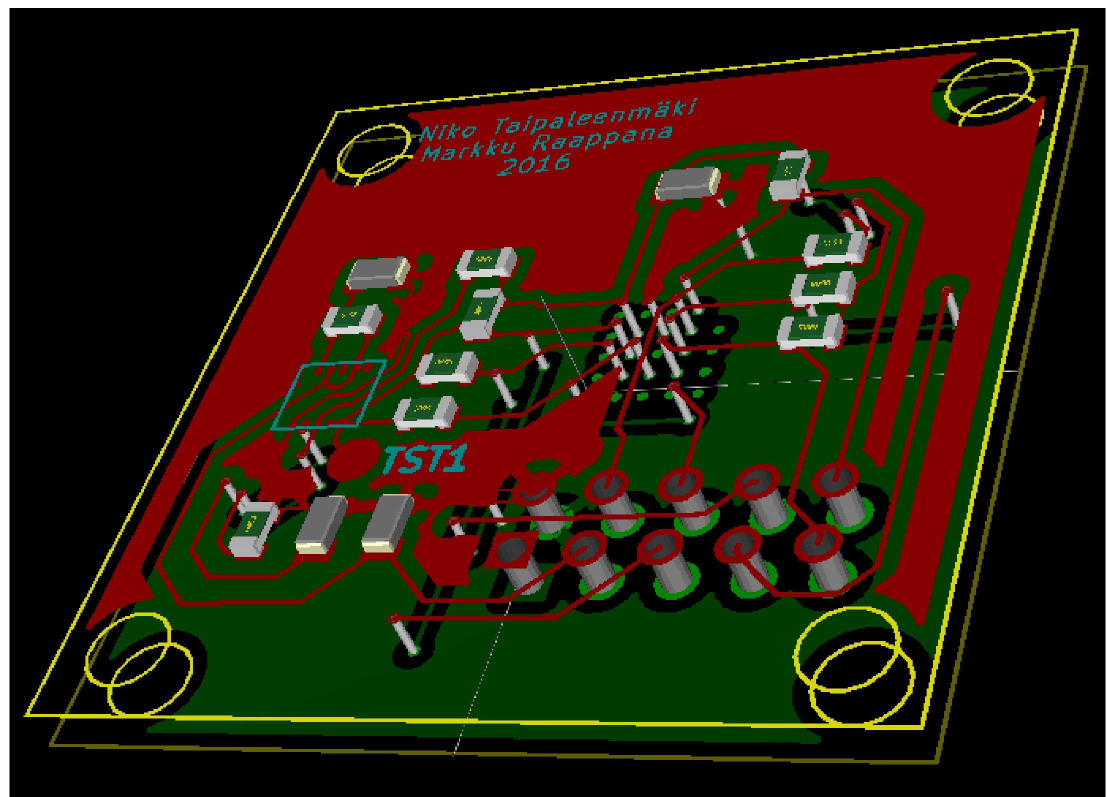


Figure 11. 3-D model of a daughter board to be used in testing environment.

One has to take into account also the fact that if visibility to the DUT is of importance, the use of the actual component can be problematic. The emulator vendors provide tools for checking these properties via internal waveforms with ease but when using FPGA-environment, the designer has to have good knowledge over the design

in order to use ILA's (Integrated Logic Analyzer) in right nets [20]. An alternative approach is to use an EDA vendor tool that enables the designer to choose the desired signals that are to be monitored with limitations such as sample length and number of signals. [21] If the design (DUT) can be transferred to FPGA-environment as it is after synthesis, the nets can be checked from the netlist. If the design is compiled and downloaded to FPGA with changes the RTL has to be constructed so that there are points created where probes can be mapped to. This method enables the test engineer to use a logic analyzer with multiple channels and good resolution to check the actual signals generated by the DUT or the real component. Where an emulator can be used, the instances like the FPGA ILA's, are just a stack memories that can be analyzed. It has to be stated that with emulators there are no means to check the external signals without using additional equipment such as an oscilloscope.

3.3. Design checks based on assertions

A growing trend among companies that are involved in RTL-design is to use RTL-language specific design property checks and assertions to check and verify the produced RTL-code. By using assertions and property checkers, the design procedure is under constant observation design checking-wise. The main reason for using assertions and property checks along the RTL-coding is to reduce the overall design time.[18]

The assertion can be implemented directly to the RTL-code as an immediate assertion for example to check certain randomization casting values or used as a separate assertion module where concurrent assertions are used to check any variation of single or multiple signal behavior.[16][18][22]

Using assertions to check the design during RTL-coding helps the design engineers to make pre-verification testing in order to ensure that the RTL-code produced meets the given specification. The RTL-code is basically tested while it is written because the use of assertions makes the designer to follow his/her interpretation of the design specification to the full. This means that if the designer writes an assertion that he/she knows to fail every time, there has to be a fault in the original RTL-code and vice versa.[16][18][22]

The assertion-based methodology has proven to be a very useful tool for the design engineers to make good code. The problem, however, has been that although the methodology is recognized to be useful the verbosity has been the most negative key feature for the designers to adapt the methodology in broader use. This has been identified by the EDA-vendors and has been addressed by means of simplifying the syntax for example in SystemVerilog. An example can be presented to clarify this situation. The three following RTL examples have the same functionality but the verbosity is lessened with every subsequent example. The property statement evaluates sequential expressions that are to be asserted true or false.[18][22]

```
property A1;
@(sampling_signal) disable iff  (expression)
property_sequence_or_expression
endproperty

assert property (A1) optional_action_block;

Example

property A1;
@(posedge sclk) disable iff  (!rst_n)
a |-> ##1 b |-> ##1 a
endproperty

assert property (A1)
else $error("Wrong sequence %h did not follow %h in given time or %h did
not follow %h in given time", b, a, a, b);
```

```
ERROR_b_did_not_follow_a_or_a_did_not_follow_b:
assert property (@(posedge sclk) disable iff  (!rst_n)
(a |-> ##1 b |-> ##1 a));
```

In this example, the lines were reduced significantly and the verbosity lessened considerably. Even this improvement has shown to be insufficient from the designer's point of view so the assertions can be presented as macros, hence the verbosity is set to almost minimum.

After initializing the macro where sampling signals "if and only if"-expressions are stated the use of assertions is made quite easy syntax-wice. The structure of the assertions is however present and to be solved by the designer. The macro can then be utilized with multiple assertions.

```
`define assert_clk(arg) \
assert property (@(posedge sclk) disable iff (!rst_n) arg)

ERROR_b_did_not_follow_a_or_a_did_not_follow_b:
`assert_clk(a |-> ##1 b |-> ##1 a);

ERROR_a_did_not_follow_c_or_b_did_not_follow_a:
`assert_clk(c |-> ##1 a |-> ##1 b);
```

After defining the macro, the number of lines is not actually reduced so much but the syntax is simpler and number of characters is less than in the previous example. It is worth mentioning that the designer has less concern about the use of brackets in this last example.[22]

Using assertions makes it easy for the designer to follow any possible faults inside the design if the labeling or optional assertion action block is stated properly. This is a very useful way of using assertions since after error has occurred most EDA-vendors provide an option to follow the signals in their waveform displays. If the previous syntax is used, all the signals stated in the assertion will be visible and no tracing is needed outside the context, an example of this is shown in Figure 12.

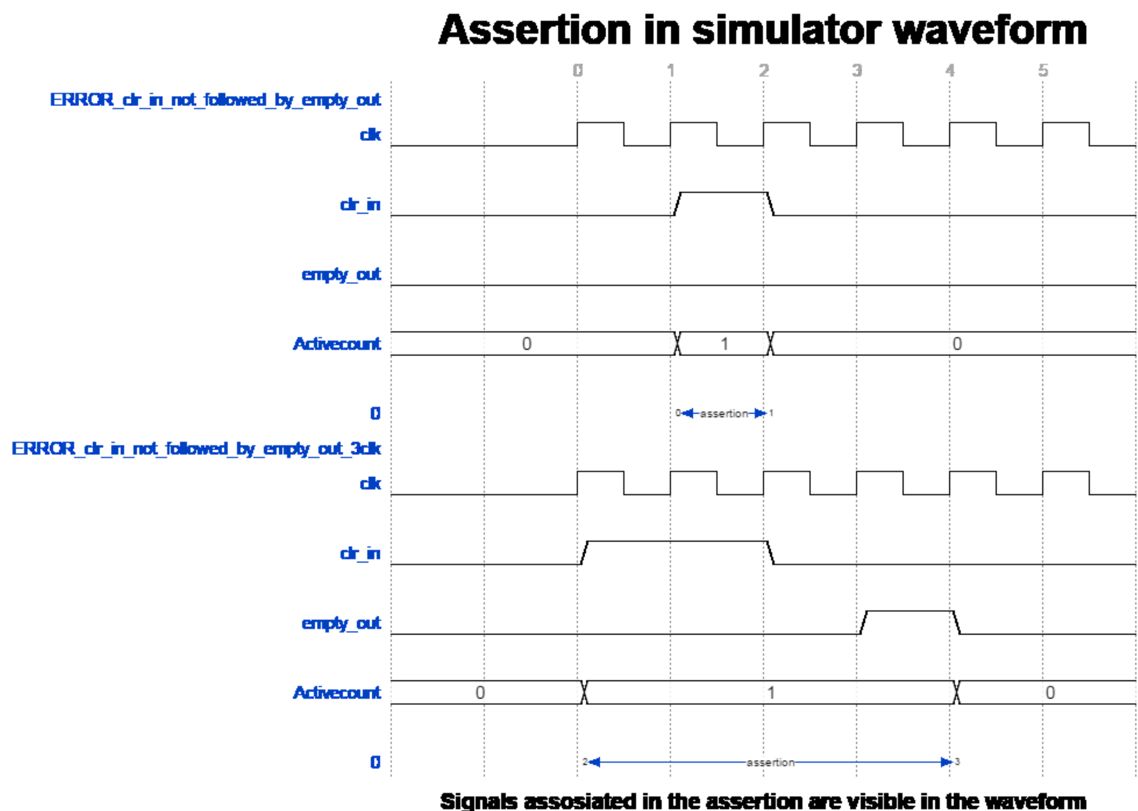


Figure 12. An example from assertion signal dependency visibility in a simulator waveform generator. The assertion length is shown after the zero label as blue lines.

Since the assertions and property checkers are used mainly in simulation environment, it is advisable to use them in a separate module that can be implemented to the design via bind-statement. It can be said that every minute used to make assertions for a new design is gained back in manifold during the verification.[18]

The same principle is utilized by the EDA-vendors in their component libraries. Quite recently the vendors have released AIP's (Assertion based Intellectual Property checker) that use property checks and assertions to verify that the used bus protocols behavior is as desired. These AIP's (Assertion based Intellectual Property checker) are UVM based instances that use assertions and property checks to verify that the used buses follow all the given specifications and that their behavior is as desired.

4. DEVELOPMENT OF THE MULTIPURPOSE VERIFICATION SYSTEM

In this chapter it is explained what where the stepping points in this thesis project while choosing the approach angle to the final design. The original problem and the initial solution for this problem are also elaborated.

4.1. Ground Zero

A fundamental question can arise when testing a new DUT against a VIP that has not been verified, namely, whether the failed test is caused by the DUT or by the VIP. This can be the situation in pre-verification when new modules are implemented to immature design. Every new driver and DUT has to be tested and verified before they can be implemented to the design.

If the situation is as mentioned previously, it is impossible to determine where the problem is if the designer is using the assumption that he or she is using a valid DUT with valid configuration against a valid VIP. In a case like this the designer has to detect if the DUT, the configuration or the VIP is faulty or any combination of these. An example can be presented using simple SPI-bus protocol read operation between a SPI-master-component (which can be interpreted to act as a DUT in a simulation) and a SPI-slave-component (which can be interpreted to act as a VIP in a simulation) as shown in the waveform in Figure 13.

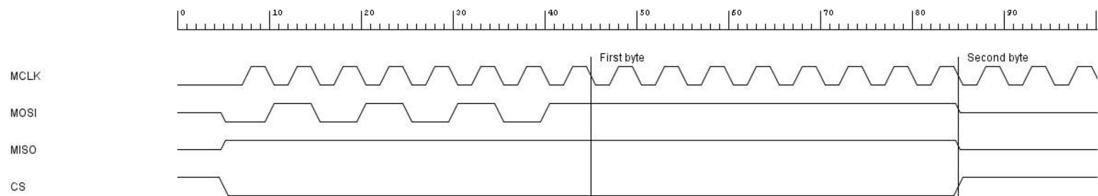


Figure 13. A fault in the initial handshake between the SPI-master- and SPI-slave-component.

Without examining the datasheets of the SPI-master and SPI-slave, the waveform above cannot be unambiguously explained. In this case, the SPI-master is trying to make a one byte read operation using little-endian where the LSB is sent first. However, the SPI-slave is using big-endian, where the MSB is sent first, so the SPI-slave is translating the transmitted data as write function request of three byte-length to a certain address. In a way both components are functioning completely correct however, the result is unwanted.

The scenario above is made by mimicking real components but in pre-verification state, the components are created in a simulator from RTL-code. The problem is that these RTL-generated components do not have documentation ready that specify their behavior and functionality, but are merely constructed by applying architectural guidelines and design specifications. If there are any mismatches between the documents and the RTL, there is a great risk that the design does not work as it should.

When facing a situation where the system is not working as it is presumed, the designer has to pinpoint the origin of error in order to fix the problem and to get the desired functionality to the system. If the designer is confident, that he or she has done

everything right with the DUT configuration and that the DUT is constructed right, it is easy to point the finger towards the VIP. But how to be sure that the VIP is constructed wrongly? Alternatively, is the origin of error inside the DUT or in the configuration? To understand the behavior of some particular instance, the designer would have to have access to the RTL source code and further-more have the knowledge how to interpret that code.

4.2. Multipurpose Verification System

As a result to the problem manifested in the previous section, a concept of Multipurpose Verification System (MVS) was introduced. It seemed that this concept could have answers for the questions about the bus protocol current state. It would also use assertions and property checks to monitor the transmitted data from the DUT. A general configuration of this system is shown in Figure 14.

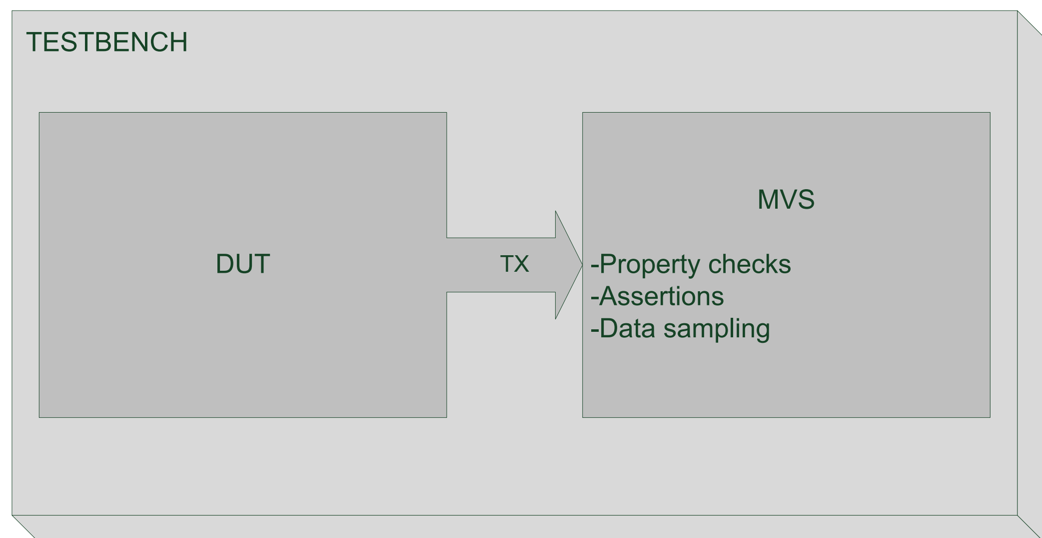


Figure 14. Conceptual model of the multipurpose verification system.

The concept was drafted based on the problems faced while testing similar system. The idea was based only on the fact that the bus protocol has to be checked before making any further changes to the test environment. The MVS would make property checks that would further on be asserted. The system would also hold some additional properties such as data sampling. The concept also holds the idea about the synthesizability and the use of SystemVerilog as the design language. The synthesizability would also grant possibility to make initial testing locally, with partial design in a single FPGA, instead of using the whole design with multiple FPGA's.

After initial evaluation the MVS was supposed to be a monitoring unit where transmitted data was evaluated and based on that evaluation, the user was to be informed of how the transmitted data was seen from the receiving part point-of-view. This manner of approach seemed to be quite straightforward and hold only one part of the interaction between two instances.

More studies about the subject had to take place in this point. Further discussions about the subject revealed also that mere monitoring was not sufficient for the designer

needs. The multipurpose verification system needed more functionality in order to be useful. This lead to a situation where the concept had to be revised.

4.3. Hardware versus Software

The project was started from the hardware designer point-of-view. Even though this approach would have been sufficient for some fields, the overall benefit was not comprehensive enough. Where the hardware designer could examine one particular IP module, a SPI-master module, in this context, as a DUT, the software designer needed a more complex design as a DUT, in order to get the system tested see Figure 15.

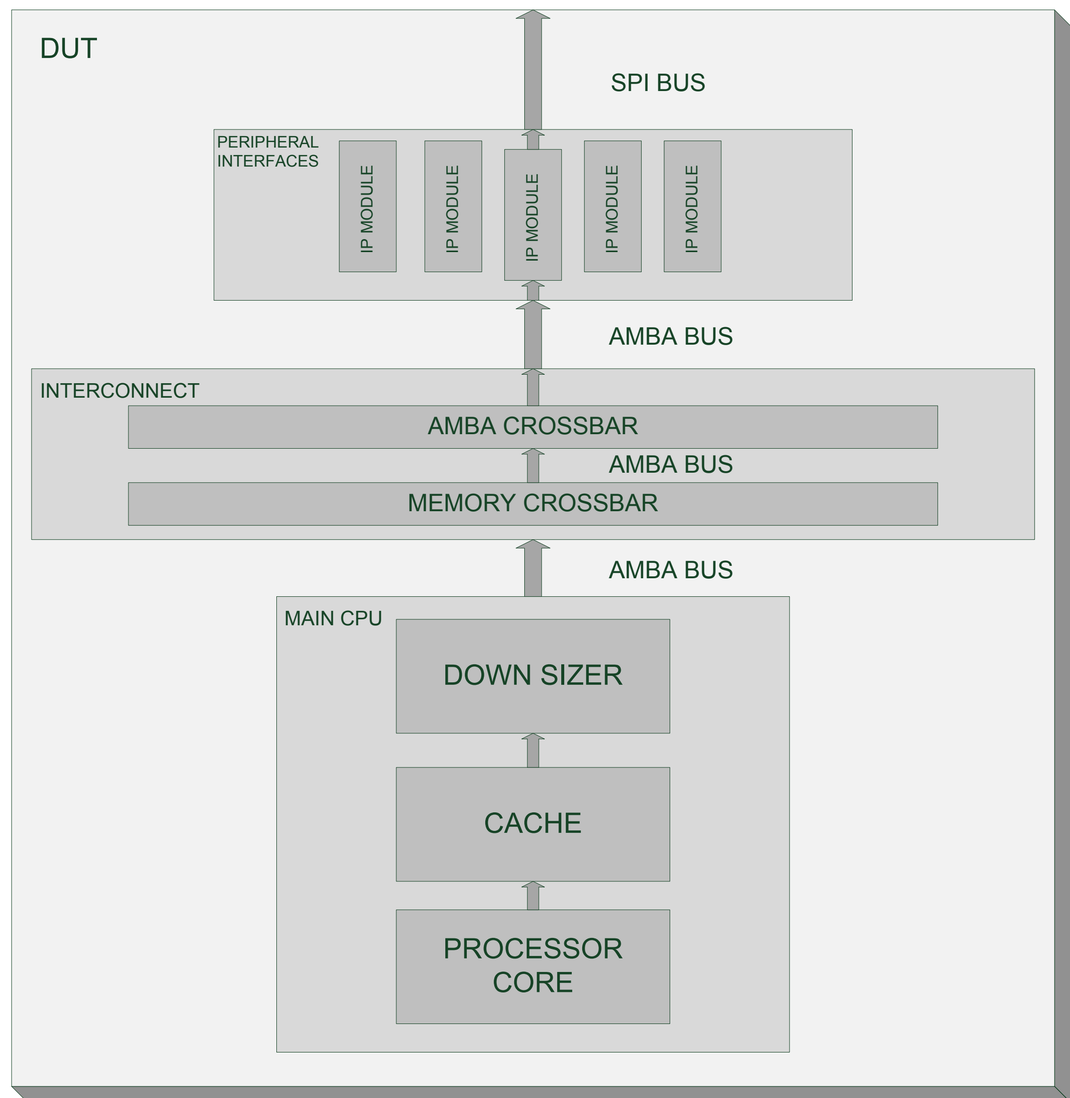


Figure 15. The module block for software designers DUT

In order to test a driver for peripheral interfaces the software designer needs to utilize the main Central Processing Unit (CPU) and all the intermediate instances, AMBA-buses (Advanced Microcontroller Bus Architecture) and AMBA-crossbars, before reaching the final IP to be tested. This makes the testbench bigger in a sense of

the number of gates used. The size would inevitably limit the usage of it in laboratory facilities due to the mere size of the design. The premeditated FPGA board would hold 4 million ASIC gates but if the needed design size would exceed that number, the near personal workstation usage would be highly limited.

Whereas the software designer would treat the peripheral interface as a black-box and the only interest is the output from that instance, the hardware designer would treat that same instance as a white-box where the interest is in the functionality of that particular instance. In both cases, there is a need for a data logger that would enable the inspection of the output later in time. At this point the idea of a command logger was born. This idea contained a concept where output data would be analyzed so that the designer would have a clear view of the commands that he or she had written. Instead of, from waveforms, binaries or hexadecimals the designer could read the created command list in a text form.

To transform the data into readable form includes recording the DUT's output data, processing the data, comparing processed data to Look up Table (LUT) or alike and finally creating a text file. The task would be a straightforward procedure if there were only a few instances that the RAM would have to mimic, however the number of instances would be far greater. Furthermore, if this concept needed to be scalable, the system would have to be constructed with even more functionality than the latest iteration implied at the beginning.

4.4. General Purpose Model and Softcore

The idea was to divide the multipurpose verification system into smaller separate sections so that the design could be more controllable and generic. Instead of adding all the functionality to one module, the design would be sectioned into generic register bank section and to more detailed softcore section, a microprocessor implemented using logic circuits, where the data from the registers could be processed as shown in Figure 16.

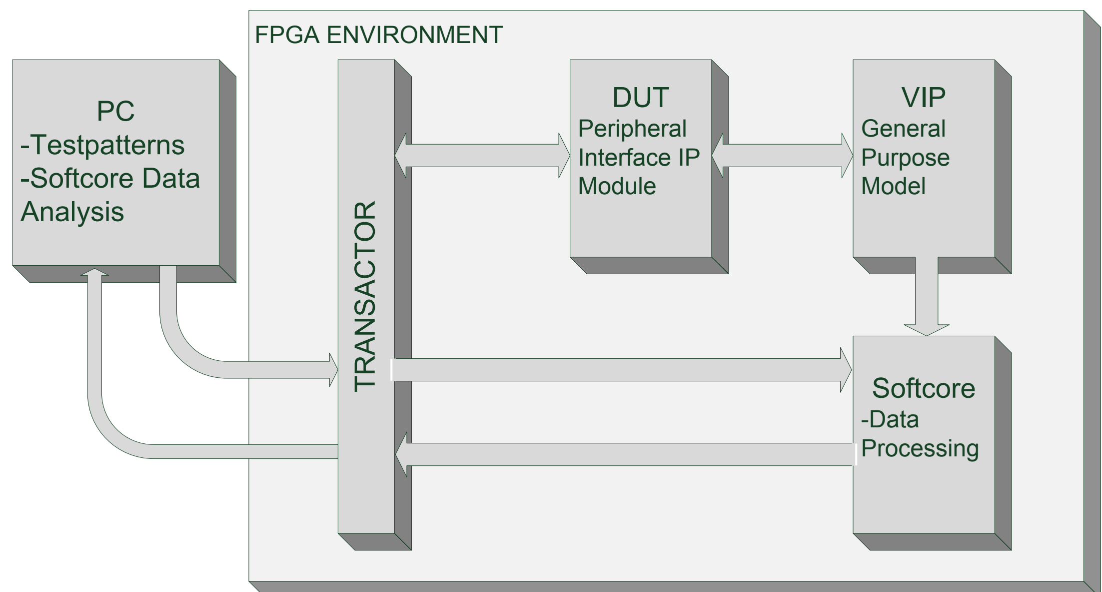


Figure 16. The multipurpose verification system in FPGA environment.

The general-purpose model, referred to as VIP in Figure 16, would act as a counter part for the DUT. As the SPI-bus protocol is constructed from sequential read and write operations, this section doesn't need to have any complex logic build inside it. If simplified, it needs only to have configurable shift registers, register banks and decoding logic. If functionality is needed, a look up table can be added in order to mimic functionality of a SPI-slave.

By applying softcore to the system it is possible to make the system more flexible to future changes to the system. If the arithmetic part of the design is separate from the rest of the system, it is more easily configurable. The softcore can be driven through a transactor if some particular functions are needed from the design and it can be reconfigured at run time.

After this approach was chosen, a check was started of what are the options for using the softcore as a part of this particular design. If the softcore was to be implemented to the design, it had to be compatible to the used FPGA-family. This would initially limit the use of the available softcores.[23][24] After starting the survey, an idea arose that the logic inside the design could be brought out from the design. All the data could be processed outside of the system since no real time processing would be used. One of the possibilities was to use scripting language to analyze and process the available data. This would mean that instead of using softcore, the design would be implemented with a monitoring unit that could be accessed via a transactor or even skip the monitoring unit and access the registers directly.[25]

Using this approach would minimize the area used for the synthesis. Instead of using any logic inside the synthesized design, the system would be created so that the use of hardware was as minimal as possible. This would eventually make the design more flexible for future development.

4.5. Final Approach

In order to minimize the synthesizable area and to maximize the flexibility the design was reorganized once more. As the need for the softcore was compromised, it was removed from the final version. The decision was made to utilize Object Oriented Programming (OOP) for the data analysis and driving the system. The OOP would enable fast data processing and test data generation. Using the OOP would also benefit from both software and hardware testing and verification, by giving them the same accessibility to the test platform. The synthesizable section was to be made as small and simple as possible and as much as possible of the functionality was to be transferred out of the system to be processed with OOP.

The use of an EDA-tools component library would enable to generate most of the instances inside the design automatically. This would mean that the main task was to design an instance that would act as a MVS. The instance should be as generic as possible. In order to achieve this goal, a configuration register would be needed. Using this approach makes it possible to build an instance that could be used in various tasks without making major changes to its base functionality. The block diagram of this instance is presented in Figure 17. [26] To simplify the diagram, only a four wire SPI-system with optional input signals is presented. The speed- and quad-SPI-mode are excluded.

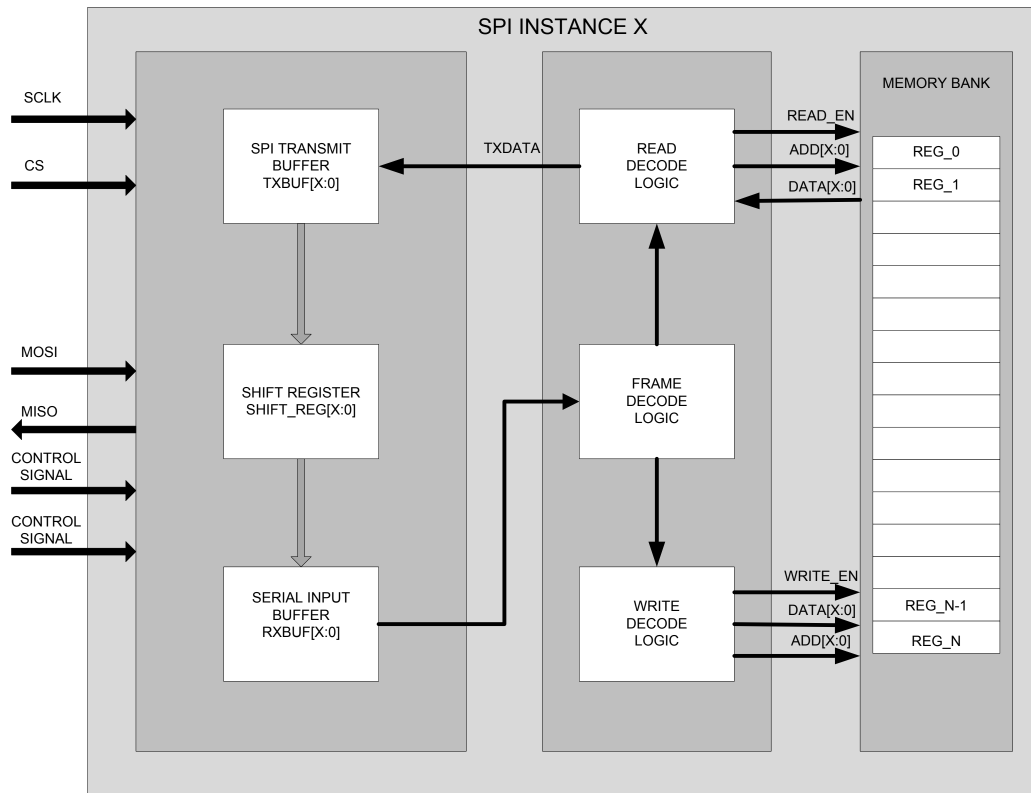


Figure 17. SPI-module block diagram.

The system had to be constructed in a fashion that would enable it to be used in different environments. This meant in practise that certain modules would be implemented to the design. These modules make it possible to use data gathered from the top level designs capture points or from the testbench and synchronize the use of the data. The synchronization is handled by adding binary counter and pulse generator to the system. Bus crossbars and bus bridges are configurable, which makes it possible to change the I/O-ports to suit the DUT. The MVS is reconfigurable via the register bank that hold the information for the parametrized instantiations inside the MVS as well as other configurations for the system. The MVS is integrated into a unique closed loop configuration. The generated memory block is accessible for both the MVS and the master entity via a separate data access point. This enables the master entity to make a comparison between the intended and actual actions through the testbench instances such as DUT, transactor, crossbar, et cetera. In other words, a comparison can be made whether the intended action, stimulus, reaches the MVS in correct form. Since this is a conceptual approach, the more detailed information about the system is not vital. The system level module block diagram is shown in Figure 18.

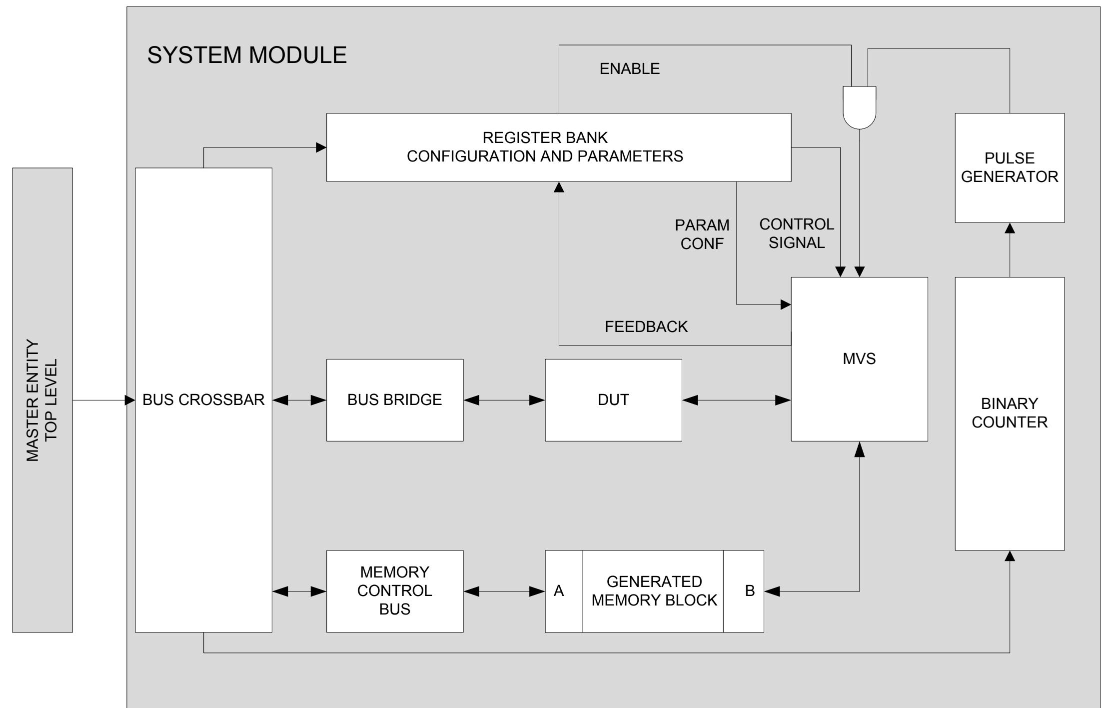


Figure 18. The system level module block diagram.

4.6. Related Solutions

The use of a commercial verification system was considered [27]. Many of the EDA vendors had verification intellectual property products in their portfolio that could be utilized in this project that could enable an alternative approach for the construction [28][29][30][31]. It would be possible to use these instances as black boxes inside the system.

By studying further this subject, the advantage of using these products was that the products were already verified by the providing EDA vendor. All the available assertion and verification IP's are monitoring the functionality of a bus protocol and giving indication if an error occurs during transmissions. A drawback in this solution was that the provided VIP's and AIP's were suitable only in certain types of environments i.e. in simulators and emulators. This was mainly due to the fact that these instances were UVM-based. Nevertheless, these instances could be used to test them in simulations that could help to evaluate them for further use.

Since there was possibility to use a ready-made environment created for the FPGA, using commercial verification products seemed appealing. Negotiations were started with the vendor to use their instances also in the synthesized environment. The vendor would have to create an assertion and verification instance that could be synthesized. It would be possible to reduce the amount of work to create a working system if the vendor could provide the needed modules. The possibility to monitor the internal signals for the VIP and AIP would also be highly appreciated and useful since it would allow using the assertions via SVA bind-statement.

The preliminary testing for the commercial VIP and AIP would be done by utilizing a SystemVerilog testbench with a small amount of write and read operations. If the

result would be promising, these instances were to be implemented to the existing environment mentioned earlier. The benefit to use the commercial products is that the product support would be left to the vendor.

Further studies regarding this subject were not done due to the workload for making an evaluation that is more detailed to be included in this thesis, proved too great. This meant that the preceding stage was going to be the platform for the coming SystemVerilog-model.

In the following chapter it is explained how the models shown in Figure 17 and Figure 18 were crafted into a synthesized models.

5. SYNTHESIZING AND FPGA PROTOTYPING

This chapter explains how the main principles and guidelines were adopted to the creation of the design in hand. In addition, some key issues about the problems that arose during this stage of the project are addressed as well.

5.1. Declaring the Designs Main and Submodules

The design had to be self-built from the very beginning, this meant that there were no specifications or design rules that could be used as guidelines for the design. The earlier research, documented in Chapter 2, had given some clues as to how to approach the situation. Based on the developed block level model, presented in section 4.5, a RTL-model had to be created for simulation and synthesis purposes.

One of the key issues was to decide whether to use the SPI-masters clock signal as the designs main clock or choose a faster clock signal for the designed SPI-instance. In the very beginning, the former alternative was chosen due to the properties that some of the SPI-instances have as mentioned in Chapter 2. This decision made the design to take a step towards being a product specific model rather than a general purpose model.

The design was almost completed using only the SPI-master clock as the design clock but the problems with dummy clock cycles needed for the data decoding and register transfers led using a faster clock as the design clock would be more advantageous. After the decision about the faster clock was made the base functionality from the previous design was adopted to the new design with minor changes. It was clear that some additional signals and finite state machines were needed due to the faster clock.

The design was originally constructed under a single module but the need of integrating this design into the existing modules, provided by numerous EDA vendors, the design was split into three main parts shown in Figure 19.

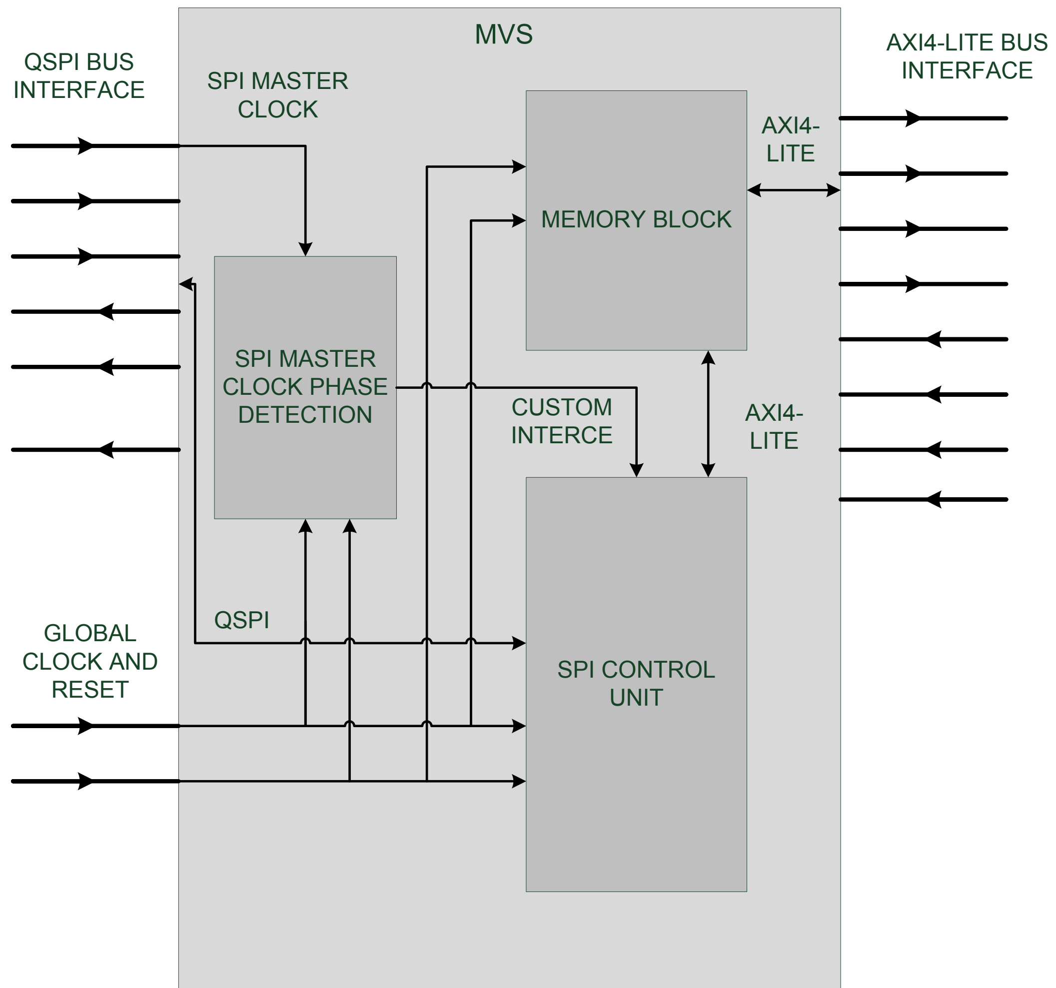


Figure 19. The module level schematics of the designed multipurpose verification system instance.

The spi_control_unit-module would hold all the logic needed for the decoding and signal processing where the phase_detection-module would handle the timing regarding the SPI-master clock and the memory_block-module would act as a register bank for the design. The lastly presented module would eventually be replaced with another instance and therefore the functionality of this module was reduced to the minimum and would hold only a limited address space. The design holds also a SystemVerilog-file that enables the reconfiguring of the module via defining the parameter values inside the SPI-module.

The memory_block was to be replaced with a register bank made with a specified macro provided by the EDA vendor. This meant that the access interface was to be modified to use the AXI4-lite interface instead of a custom interface used during the preliminary configuration. The use of AXI4-lite interface would enable the designer to change a memory block without making major changes to the RTL. Since there was only read and write operations between the SPI-instance and the generated register bank, the AXI4-lite interface did not hold all the functionality it possesses by default, but the interfaces were to be customized to meet the specification needed for the operations mentioned earlier.

The original design was modified so that it holds all the necessary signals for it to be used as a master AXI4-lite-instance memory access-wise. The main principles for these changes were obtained by following the guidelines in the AMBA® AXI® and ACE® Protocol Specification provided by ARM®. [32] A BRAM-module (Block Random Access Memory) with AXI4-lite-slave-interface was generated with IP-generator in order to test the system in a simulator environment to the reasonable extent. The conversion from the SPI-protocol to the AXI4-lite-protocol proved to be a time-consuming process, however. This was because the conversion to this direction was unorthodox. The variable parametrization was also a concern due to the big differences between these two bus protocols. As an example, the data width used in the SPI bus was eight bits whereas the AXI4-lite had fixed 32 bit data width. This resulted in the fact that the used address space would have unused data storage. This actually proved be a beneficial attribute because the SPI bus data width could be adjusted without compromising the use of the fixed data width in AXI4-lite bus because the data width in the SPI bus would not exceed the 32 bit.

5.2. Verification

The design had to be tested and verified before implementing it to FPGA-environment. The simulations were run simultaneously using directed test randomization with two different simulators, i.e. Mentor Graphics' QuestaSim and Synopsys' VCS. This was done because it was shown that the simulators used different a kind of algorithms while compiling the RTL. In a very early stage, there was a clear mismatch between waveforms generated by these simulators. Each of the waveforms is shown in Appendices Appendix 1 and Appendix 2. Thus, both were used in order to pre-verify the RTL before applying a lint checker.

The RTL coding was generated following the basic guidelines of SystemVerilog [8]. The design of the Finite State Machines (FSM) was the most time consuming part of this work. In order to build the FSM functionality properly, all the documentation for the used bus protocols had to be carefully familiarized. While debugging FSMs it was shown that a minor error in document interpretation caused major faults in the results. An example is shown in Appendix 3 in order to clarify the one hot FSM coding style.

The SPI-bus uses a serial interface, whereas the AXI4-lite has a parallel bus interface. This resulted in the fact that the SPI-bus would be the more limiting interface and all the timing issues had to be addressed accordingly. The system was verified using multiple sequential write and read operations that would give evidence that the system is functioning as desired. The clocking frequencies ratios were also variated in order to check whether to system can tolerate changes in the frequencies used. This was one of the critical paths in design verification. The waveforms from these operations are shown in Figure 20 and in Figure 21, respectively. A signal level AXI4-lite write operation is explained in more details in Appendix 4.

By examining the waveforms, it was shown that the system had the desired functionality. After lint checking, it was tested for the synthesizability.

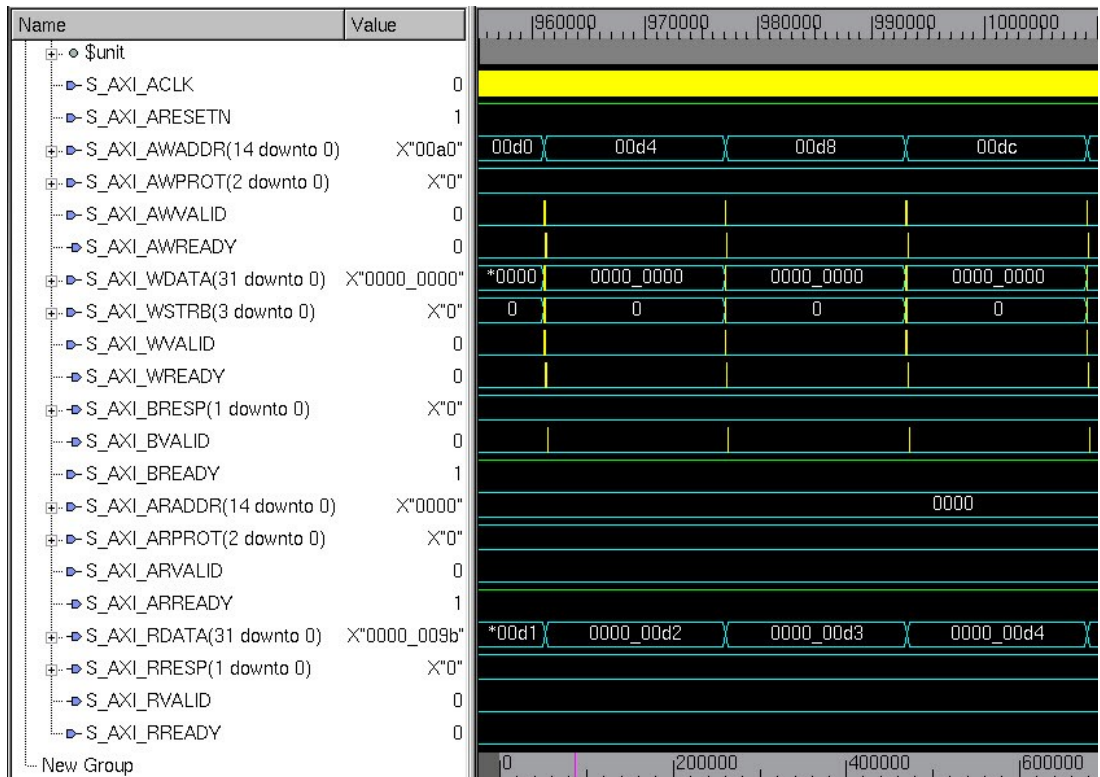


Figure 20. SPI to AXI4-lite write operation.

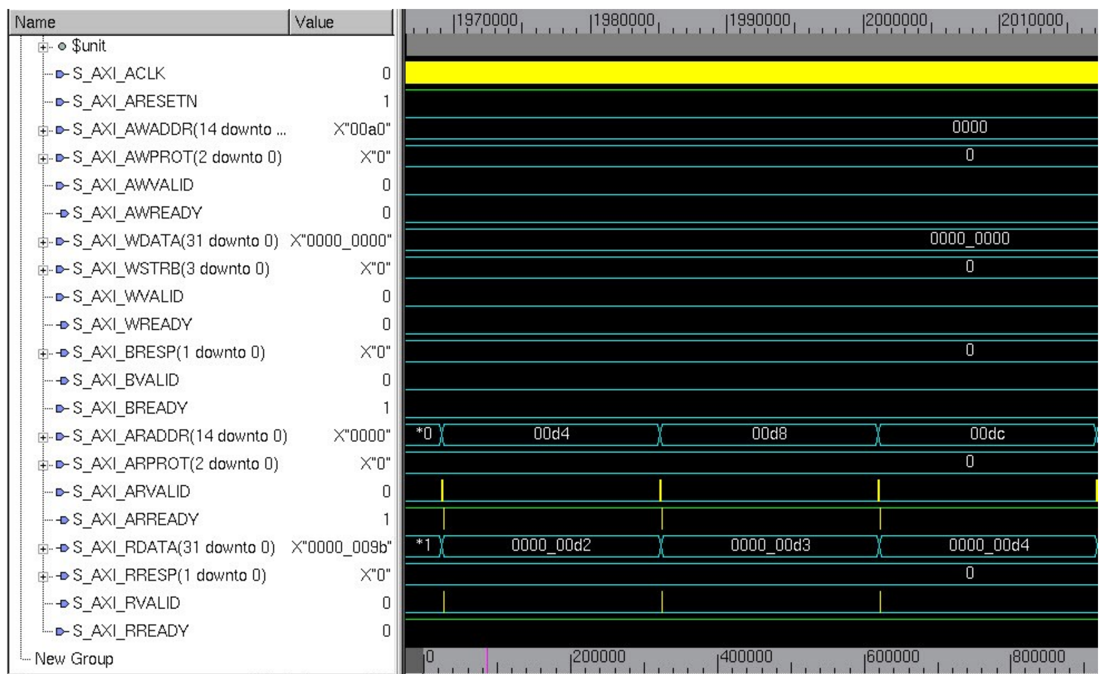


Figure 21. SPI to AXI4-lite read operation.

5.3. Synthesis

As the goal for this thesis was to create a system that could be used in synthesis, the design had to be checked for synthesizability. The design was checked using Xilinx's EDA tool Vivado. The design was checked as a stand-alone instance. The synthetization report can be seen in Appendix 5. The synthesis report shows that the design is synthesizable and it could be used in a FPGA platform.

After the synthesis, the design could be analyzed. The reports show that the design used only parts of mil compared to the complete synthesizable area. The design uses only a few hundreds of flip-flops and IO-buffers. This analysis does not take into account the parts that are consumed by the register bank. The size of the register bank is configurable and it is yet to be decided what is the optimal size and is there for not addressed.

As in this state, the system is using up to 330 mW with medium confidence level. As the setup and hold time restrictions were not defined the timing summary is only partially examined. There were no timing violations present and for the system clock overall slack, it could be stated to be 56% smaller than needed. As said earlier this statement lacks some of the vital parameters needed for a conclusive analysis. The MVS can be implemented to FPGA without having issues with used clock frequencies nor used area. The physical level optimization of the design is out of the scope of this thesis and is to be addressed in a future development state.

The scope for this study had been fulfilled at this point. Due to the nature of this thesis the study was frozen. The future development would be to make the design usable in the testbench library as a separate IP that could be implemented to any suitable testbench. The configuration was introduced in section 4.5. All the needed components would be implemented using available library components to the design in hand.

6. DISCUSSION

This thesis focuses on the problem that arose when observing SPI-bus where the stimulus provided towards the SPI-slave-component did not result in the desired response. While examining the bus signals, it appeared that the stimulus was correct but there was no clear way to confirm this assumption. The question, after this was how to verify the SPI-bus protocol with the tools available?

The goal was to study what were the options to get a solution to this problem. Preceding this thesis, a study was made that showed there were not tools available that could be applied to complete this thesis. This study was vital for avoiding the situation where a ready commercial system would already have been available but not utilized. During this study, it was found out that there are multiple solutions for a designer to test a bus protocol. There are commercial solutions that can be utilized in a simulator, emulator or FPGA environment. All these solutions were, however, more or less platform specific and cannot be used as such as a generic tool. This issue has been introduced to the EDA vendors.

The use of a commercial product has clearly many benefits such as usability, controllability and product support. At this point, if there is a need for a multipurpose SPI verification system, it has to be manufactured locally.

This thesis focuses on the problem where the solution has to be made locally. At the starting point, there was no documentation what-so-ever that could be used to create an instance that could provide all the functionality desired. This meant in practise that it had to be studied how this could be done and all the solutions had to be self-made. To clarify this further, there was no guidance how to build this specific IP but all the steps had to be studied and implemented during this thesis.

Before this thesis work was started, the original problem was faced by the thesis worker and based on this problem, a conceptual idea was introduced to the employer. The conceptual idea was then taken into closer evaluation. The evaluation resulted in the final requirements for the thesis. This idea was then carefully studied and after several iterations and side roads, the final approach was decided. As a result, a multipurpose verification system was created. The specifications for the MVS were created alongside the thesis, as they were not academically available.

The creation of MVS showed that a synthesizable IP is a good solution for a problem in hand. The MVS holds enough functionality to be a stand-alone IP against a DUT. It is not directly a specific verification tool, which can verify the full functionality of a design, but can be used in pre-verification in a good level of confidence. To get better confidence level the MVS has to be verified more thoroughly using more a complex testbench with randomized stimuli and timing. This could be done for example using UVM. The goal would be pointing out corner cases and possible issues with timing.

The SystemVerilog enables using functional modules in FPGA-synthesis. The synthesis of the MVS showed that the memory bank excluded, it took only parts of per mil from the total synthesis board area. The MVS was constructed using only a few hundreds of flip-flops and IO-buffers and it was a relatively simple design. The clocking issues should be negligible since the SPI-bus uses relatively low clocking frequencies and is asynchronous compared to the system clock. [10]

This study proved that creating an instance like the MVS could help the designer to test parts of a design without having a specific verification tool. The need of a verification system is still present and the MVS cannot replace it in a final verifications process. The use of the MVS or alike is highly recommended when applying

prototyping since it is a white box-type system and can be monitored and used with ease.

The downside for using the MVS is the need for in-house support that would inevitably affect the used resources. A good documentation helps in this situation and version control should be applied according to the company policy.

After evaluating the results and how they compared with the requirements, a conclusion could be made that all essential goals were met during this thesis.

The future development idea is to implement the MVS to a more complex configuration where it can be used as a company component library IP. This configuration was introduced in Figure 18. This would enable the company research and development to re-use this IP in their current and future projects. As an example, the functionality if a new driver can be tested using MVS as corresponding instance or in signal level testing with multiple SPI-slave SPI-configuration could be mentioned here.

The MVS functionality can be enhanced in the future by applying separate ID checkers, status registers, assertions and property checkers via bind statement and additional register banks in order to make it mimic specified instances.

It can be also considered if this concept can be applied with other smaller data bus protocols such as Controller Area Network (CAN) or I^2C.

7. SUMMARY

The thesis focus is on a problem, where a reliable verification component is needed to be used with an unverified DUT in different stages of SoC-development. While SPI-bus protocol is used, the bus protocol can be observed but there is a possibility that the end-point functionality and visibility cannot be unveiled.

The solution was to create a system that could be used to monitor, analyze and verify the used SPI-protocol with a reasonable level of confidence. The solution would have to be able to be used in simulation, emulation and FPGA-prototype. The use of SystemVerilog as RTL-language would enable the solution to be used in any of these platforms.

There were no specifications or guidelines that could be followed in order get a solution for the problem. This resulted in a study of multiple solutions via iterations of the original conceptual idea. The final solution was to create a Multipurpose Verification System that could be used as a stand-alone instance in any suitable testbench. The MVS was constructed using SystemVerilog. The MVS was constructed as a closed loop and was to be used with SPI for transactions and for monitoring and analyzing the transactions AXI4-lite interface backdoor was created.

The MVS was verified using Mentor Graphics' QuestaSim and Synopsys' VCS simulators. While observing the simulation results it was shown that the MVS could be used in this context with a reasonable level of confidence. After lint checks with Synopsys' Spyglass the MVS was tested for synthesizability with Xilinx's Vivado. The synthesis reports showed that the MVS could be synthesized and therefore used in FPGA environment.

This thesis proved that a synthesizable verification system can be created using SystemVerilog. In its current form, the MVS holds all the required functionality and transparency. The MVS needs to be fine-tuned before it can be implemented into the company component library. After the fine-tuning, the MVS can be used as a stand-alone pre-verification or verification tool. This asset benefits the company in testing and verification processes.

The MVS functionality can be enhanced in the future by applying separate ID checkers, status registers, assertions and property checkers via bind statement and additional register banks in order to make it mimic specified instances. The future tasks include testing the MVS in FPGA- and emulator-environments. The MVS is scalable memory bank size-wise, but in its current form, the memory resizing has to be done by hand. This feature could be automated in the future.

8. REFERENCES

[1] Mollick, E. (2006). Establishing Moore's Law. IEEE Annals of the History of Computing, 28(3), 14 p.

[2] Bamford , N. et al (2006). Challenges in System on Chip Verification. Seventh International Workshop on Microprocessor Test and Verification (MTV'06) (p. 52-60). Austin, Texas: IEEE.

[3] Mencken, H. L. (1949). A Mencken chrestomathy. Michigan: A. A. Knopf.

[4] Sutherland, S., & Mills, D. (2013). Synthesizing SystemVerilog Busting the Myth that SystemVerilog is only for Verification. http://www.sutherland-hdl.com/papers/2013-SNUG-SV_Synthesizable-SystemVerilog_paper.pdf, 45 p.

[5] International Standard. (2007). IEEE 1800™-2005 Standard for SytemVerilog - Unified Hardware Design, Specification, and Verification Language. New York: The Institute of Electrical and Electronics Engineers, Inc, 668 p.

[6] IEEE Computer Society. (2005). *1364™-2005 - IEEE Standard for Verilog® Hardware Descrition Language.* New York: Design Automation Standards Committee, 590 p.

[7] IEEE Computer Society. (2009). IEEE Standard 1800™-2009 for SystemVerilog - Unified Hardware Design, Spcification, and Verification Language. New York: IEEE, 1285 p.

[8] IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group. (2012). IEEE Std 1800™-2012: IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language. New York: IEEE, 1315 p.

[9] Leens, F. (2009). An introduction to I2C and SPI protocols. *IEEE Instrumentation & Measurement Magazine*, ss. 8-13.

[10] Motorola, Inc. (Accessed 14.7.2016). *Motorola's Spi Block Guide V04.01*. Retrieved from www.nxp.com: http://www.nxp.com/files/microcontrollers/doc/ref_manual/S12SPIV4.pdf.

[11] Micron. (Accessed 4.11.2016). Micron Serial NOR Flash Memory N25Q. Retrieved from https://www.micron.com: https://www.micron.com/products/datasheets/df5e85a8-35ef-42b5-882d-1f632ed53e4a.

[12] Prasad, M. (Accessed 9.6.2016). *Serial Pheripheral Interface - SPI Basics*. Retrieved from www.maxEmbedded.com: http://maxembedded.com/2013/11/serial-peripheral-interface-spi-basics/.

[13] Byte Paradigm sprl. (Accessed 8.11.2016). Introduction to I²C and SPI protocols. Retrieved from www.byteparadigm.com: http://www.byteparadigm.com/applications/introduction-to-i2c-and-spi-protocols/.

[14] IEEE Computer Society. (Accessed 10.11.2016). *www.computer.org*. Retrieved from Software Engineering Code of Ethics: https://www.computer.org/web/education/code-of-ethics.

[15] Esteve, E. (Accessed 10.11.2016). *Semiconductor IP would be nothing without VIP…*. Retrieved from www.semiwiki.com: https://www.semiwiki.com/forum/content/404-semiconductor-ip-would-nothing-without-vip%C2%85.html.

[16] Lahti, J. (2016). Digitaalitekniikka 3.(in Finnish) Oulu: University of Oulu, 170 p.

[17] Verification Academy. (2013). UVM Cookbook - Testbench Guide. Online publication: Mentor Graphics, 53 p.

[18] Foster, H. D.;& Krolnik, A. C. (2008). Creating Assertion-Based IP. New York: Springer Science+Business Media, LLC, 328 p.

[19] Amos, D.;Lesea, A.;& Richter, R. (2011). FPGA-Based Prototyping Methdology Manual. Mountain View, California: Synopsys, Inc.

[20] Xilinx, Inc. (Accessed 11.11.2016). *Integrated Logic Analyzer v6.1 LogiCORE IP Product Guide.* Retrieved from www.xilinx.com: http://www.xilinx.com /support/documentation/ip_documentation/ila/v6_1/pg172-ila.pdf.

[21] Synopsys, Inc. (2016). HAPS® ProtoCompiler Debugger User Guide. Mountain View, California: Synopsys, Inc.

[22] Cummings, C. E. (2010). SystemVerilog Assertions Design Tricks and SVA Bind Files. Sunburst Design SystemVerilog Assertion Techniques, 71. Beaverton, Oregon, USA: Sunburst Design.

[23] https://en.wikipedia.org/wiki/Soft_microprocessor Accessed October 2016.

[24] https://en.wikipedia.org/wiki/List_of_HDL_simulators Accessed October 2016.

[25] Asanovic, K. (2007). Transactors for Parallel Hardware and Software Co-Design. IEEE International High Level Design Validation and Test Workshop (ss. 140-142). Irvine, CA: IEEE.

[26] Shah, A. K. (2015). High Speed SPI Slave Implementation in FPGA using Verilog HDL. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET), Volume 4*(Issue 12), 4365-4369.

[27] Cerny, E.;Bergeron, J.;Thottaseri, M. K.;Anderson, T.;& Synopsys Inc. (Accessed 30.11.2016). Design of SystemVerilog Assertion IP. Retrieved from Desing&Resuse: http://www.design-reuse.com/articles/9875/design-of-systemverilog-assertion-ip.html.

[28] Cadence Inc. (Accessed 26.11.2016). Assertion-Based Verification IP. Retrieved from Candence : https://www.cadence.com/content/cadence-www/global/ en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/assertion-based-verification-ip.html.

[29] Synopsys Inc. (2016). VC Formal Verification, Version L-2016.06-SP2. Mountain View, CA: Synopsys Inc.

[30] Yeung, P. (2015). The Four Pillars of Assertion-Based Verification. Wilsonville, Oregon: Mentor Graphics.

[31] Xilinx Inc. (2016). AXI Protocol Checker. San Jose: Xilinx Inc.

[32] ARM. (Accessed 10.11.2016). AMBA specification. Retrieved from www.arm.com: https://www.arm.com/products/system-ip/amba-specifications.

9. APPENDICES

Appendix 1. An example of the waveforms generated with QuestaSim

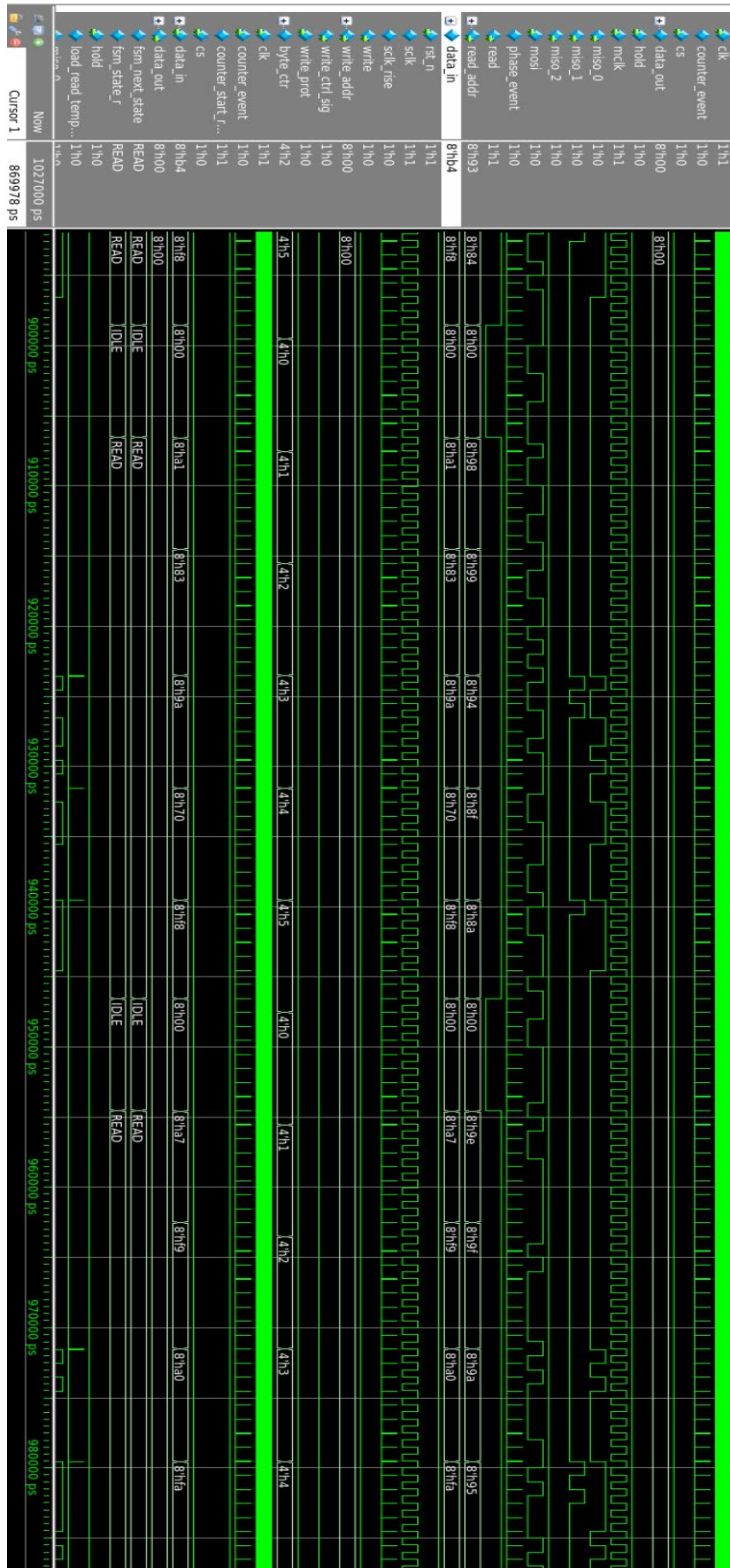
Appendix 2. An example of the waveforms generated with VCS

Appendix 3. An example of an one hot FSM coding style

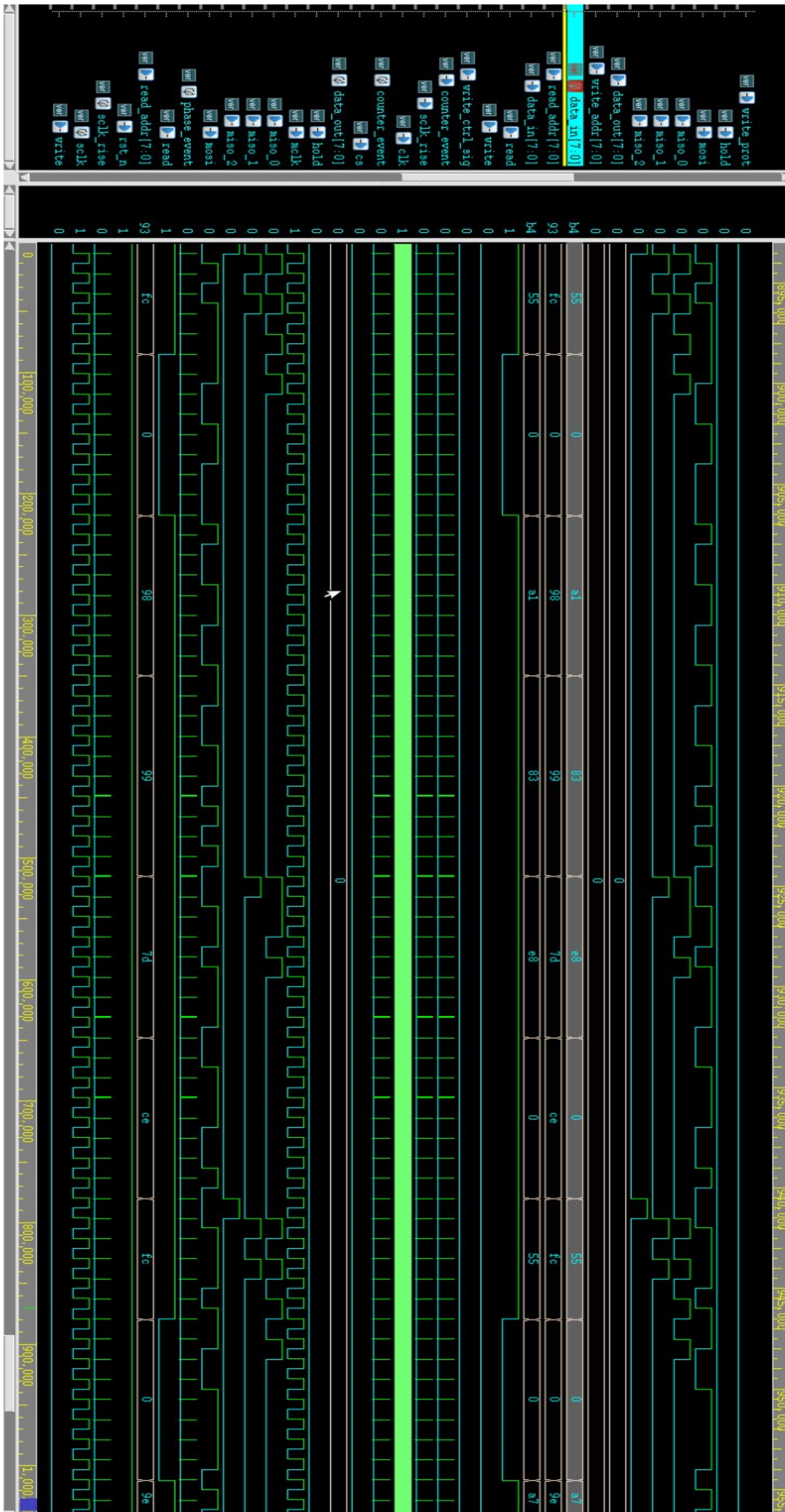
Appendix 4. A signal level AXI4-lite write operation explained

Appendix 5. The synthesis report from Vivado

Appendix 1. An example of the waveforms generated with QuestaSim



Appendix 2. An example of the waveforms generated with VCS

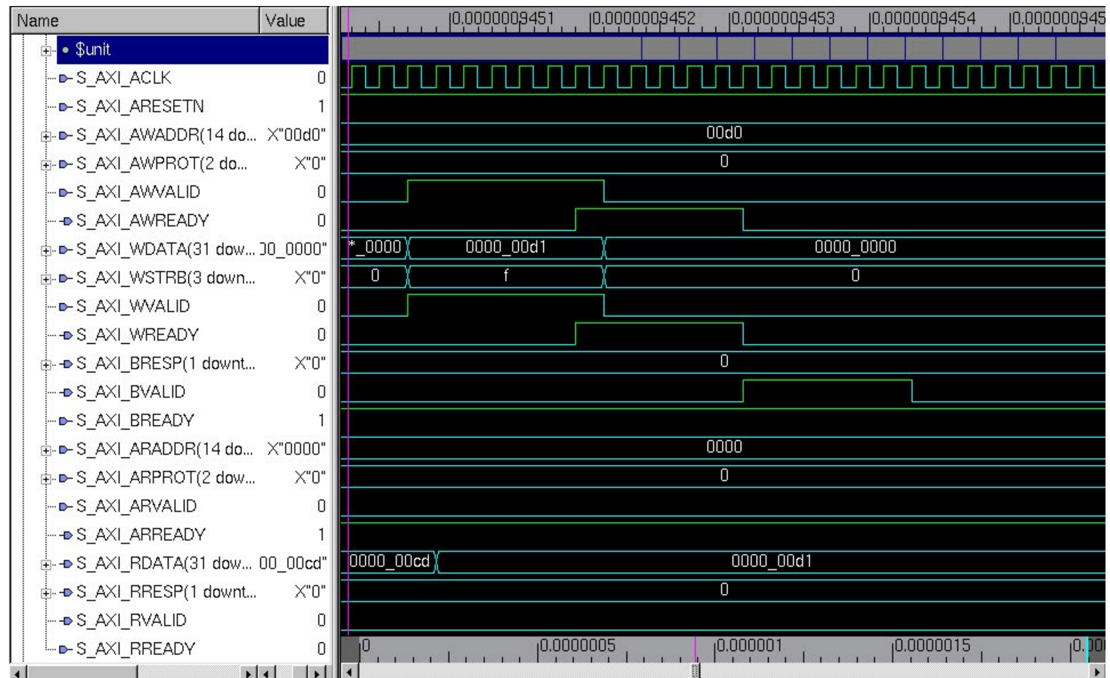


Appendix 3. An example of an one hot FSM coding style

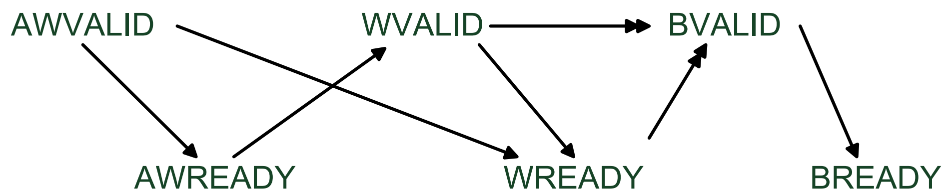
```
   enum logic[1:0] {BANK_IDLE = 2'b01, BANK_READ = 2'b10} bank_logic_r,
bank_logic_next;

    always_ff@(posedge clk or negedge rst_n)
     begin
       if(!rst_n)
        begin
          temp_bank <= 0;
          bank_logic_r <= BANK_IDLE;
        end
       else
        begin
          temp_bank <= temp_bank_next;
          bank_logic_r <= bank_logic_next;
        end
     end
    always_comb
     begin:reg_bank_fsm
       temp_bank_next = temp_bank;
       bank_logic_next = BANK_IDLE;
       case(bank_logic_r)
        BANK_IDLE:
         begin
           if(cs == 1'b1)
               begin
                 bank_logic_next = BANK_IDLE;
               end
           else
               begin
                 if(shift_ctr == BYTE_LENGTH-2 && byte_ctr >= 0 &&
counter_event == 1'b1)
                  begin
                      bank_logic_next = BANK_READ;
                  end
                 else
                  begin
                      bank_logic_next = BANK_IDLE;
                  end
               end
         end
        BANK_READ:
         begin
           bank_logic_next = BANK_IDLE;
           temp_bank_next = shift_reg;
         end
       endcase
     end
```

Appendix 4. A signal level AXI4-lite write operation explained



The AXI4-lite signals have dependencies that affect to their behavior. Basically the states for the signals are either ACTIVE (high) or IDLE (low) and the signals in ACTIVE state are further on allocated as WAIT or ASSERTED. The dependencies, the transaction handshake, in a write action between signals are shown in a graph below.



The single headed arrow points to signals that can be asserted before or after the signal at start of the arrow. The double-headed arrows point to signals that must be asserted only after assertion of the start of the arrow. Some exclusions exists such as the BREADY that can be set to HIGH as default in AXI4-lite since there are only one master- and slave-instances related to this transaction.[32]

Appendix 5. The synthesis report from Vivado

Copyright 1986-2016 Xilinx, Inc. All Rights Reserved.

```
-----------------------------------------------------------------
--------------------------------------------
| Tool Version : Vivado v.2016.2 (lin64) Build 1577090 Thu Jun  2
16:32:35 MDT 2016
| Date         : Thu Dec  1 14:23:09 2016
| Host         : quagmire3 running 64-bit Red Hat Enterprise
Linux Server release 7.2 (Maipo)
| Command      : report_utilization -file
spi_slave_utilization_synth.rpt -pb
spi_slave_utilization_synth.pb
| Design       : spi_slave
| Device       : 7vx690tffg1930-2
| Design State : Synthesized
-----------------------------------------------------------------
--------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Memory
3. DSP
4. IO and GT Specific
5. Clocking
6. Specific Feature
7. Primitives
8. Black Boxes
9. Instantiated Netlists

1. Slice Logic
--------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |  152 |     0 |    433200 |  0.04 |
|   LUT as Logic          |  152 |     0 |    433200 |  0.04 |
|   LUT as Memory         |    0 |     0 |    174200 |  0.00 |
| Slice Registers         |  118 |     0 |    866400 |  0.01 |
|   Register as Flip Flop |  118 |     0 |    866400 |  0.01 |
|   Register as Latch     |    0 |     0 |    866400 |  0.00 |
| F7 Muxes                |    0 |     0 |    216600 |  0.00 |
| F8 Muxes                |    0 |     0 |    108300 |  0.00 |
+-------------------------+------+-------+-----------+-------+
* Warning! The Final LUT count, after physical optimizations and
full implementation, is typically lower. Run opt_design after
synthesis, if not already completed, for a more realistic count.
```

1.1 Summary of Registers by Type

Total	Clock Enable	Synchronous	Asynchronous
0	_	-	-
0	_	-	Set
0	_	-	Reset
0	_	Set	-
0	_	Reset	-
0	Yes	-	-
4	Yes	-	Set
113	Yes	-	Reset
0	Yes	Set	-
1	Yes	Reset	-

2. Memory

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	0	0	1470	0.00
RAMB36/FIFO*	0	0	1470	0.00
RAMB18	0	0	2940	0.00

* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E1

3. DSP

Site Type	Used	Fixed	Available	Util%
DSPs	0	0	3600	0.00

4. IO and GT Specific

Site Type	Used	Fixed	Available	Util%
Bonded IOB	27	0	1000	2.70

Bonded IPADs	0	0	74	0.00
Bonded OPADs	0	0	48	0.00
PHY_CONTROL	0	0	20	0.00
PHASER_REF	0	0	20	0.00
OUT_FIFO	0	0	80	0.00
IN_FIFO	0	0	80	0.00
IDELAYCTRL	0	0	20	0.00
IBUFDS	0	0	960	0.00
GTHE2_CHANNEL	0	0	24	0.00
PHASER_OUT/PHASER_OUT_PHY	0	0	80	0.00
PHASER_IN/PHASER_IN_PHY	0	0	80	0.00
IDELAYE2/IDELAYE2_FINEDELAY	0	0	1000	0.00
ODELAYE2/ODELAYE2_FINEDELAY	0	0	1000	0.00
IBUFDS_GTE2	0	0	12	0.00
ILOGIC	0	0	1000	0.00
OLOGIC	0	0	1000	0.00

5. Clocking

Site Type	Used	Fixed	Available	Util%
BUFGCTRL	1	0	32	3.13
BUFIO	0	0	80	0.00
MMCME2_ADV	0	0	20	0.00
PLLE2_ADV	0	0	20	0.00
BUFMRCE	0	0	40	0.00
BUFHCE	0	0	240	0.00
BUFR	0	0	80	0.00

6. Specific Feature

Site Type	Used	Fixed	Available	Util%
BSCANE2	0	0	4	0.00
CAPTUREE2	0	0	1	0.00
DNA_PORT	0	0	1	0.00
EFUSE_USR	0	0	1	0.00
FRAME_ECCE2	0	0	1	0.00
ICAPE2	0	0	2	0.00
PCIE_3_0	0	0	3	0.00
STARTUPE2	0	0	1	0.00
XADC	0	0	1	0.00

7. Primitives

Ref Name	Used	Functional Category
FDCE	113	Flop & Latch
LUT6	56	LUT
LUT5	42	LUT
LUT4	36	LUT
LUT2	30	LUT
LUT3	27	LUT
OBUF	22	IO
IBUF	5	IO
FDPE	4	Flop & Latch
LUT1	2	LUT
FDRE	1	Flop & Latch
BUFG	1	Clock

8. Black Boxes

Ref Name	Used
axi_bram_ctrl_0	1

9. Instantiated Netlists

Ref Name	Used