POLITECNICO DI TORINO

Corso di Laurea Magistrale in
MECHATRONIC ENGINEERING (INGEGNERIA MECCATRONICA)

Tesi di Laurea Magistrale

Study and development of a real-time monitoring system for parametric experimental testbed

Supervisors

Prof. Luca Sterpone

……………………………..

Prof. Boyang Du

……………………………..

Student

Patricia Arribas Guerrero

…………………………………….

Acknowledgement

First, I would like to express my special thanks of gratitude to mi supervisors, Luca Sterpone and Boyang Du, for the guidance throughout this project and for the countless hours dedicated to ensuring its success.

Secondly, I would also like to thant all the students and doctorands from Lab3 and the new Lab7. It is being an amazing experience to be able to work with you. Truly, thank you for welcoming and helping me in the development this project.

Finally, I would like to dedicate this project to my family. They have been supporting me all the way despite the distance. Thank you for supporting me unconditionally and for always believing in me. And I also want to express my very profound gratitude to Adrián, for the countless sleepless hours spent cheering me up and pushing me to be my best.

General Index:

Figure Index:

Table Index:

Abbreviations and acronyms

ATPG	Automatic Test Pattern Generation
ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BER	Bit Error Rate
BSP	Board Support Package
CLB	Configurable logic blocks
CLK	Clock
DFT	Design for Testability
DUT	Device Under Test
EMIO	Extended Multipurpose Input Output
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input Output
HDL	Hardware Description Language
MIO	Multipurpose Input Output
PL	Programmable Logic
PS	Processing System
RTL	Register-Transfer Level
SDK	Software Development Kit
SEU	Single Event Upsets
SoC	System on Chip
UUT	Unit Under Test
VHDL	Combination of VHSIC and HDL
VHSIC	Very High-Speed Integrated Circuit

Chapter 1

Introduction

In the last decades, technological growth has resulted in a great development of the automatic and electronic sector. Within these areas, there have been different milestones, and one that can be highlighted is the creation in eighties of the FPG, that stands for Field-Programmable Gate Array.

FPGAs are semiconductor devices based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. (1) These devices can be reprogrammed to adapt to any desired application or functionality desired by the user or designer after it has been manufactured. This is one of the characteristics that distinguish this device from the ASICs (Application Specific Integrated Circuits), which are custom manufactured for a specific application, so the design is focused on carrying out certain tasks which cannot be changed during its operational life.

The FPGA configuration is generally specified using a hardware description language (HDL), similar to the one used for an ASIC device. This HDL configures the logic blocks to perform from simple logic gates to more complex combinational functions. It is also possible to add memory elements, which can be as simple as flip-flops or more complete units. All these blocks are then connected by the reconfigurable interconnects that wired them together.

An FPGA can be used to solve any problem which is computable, although they are especially useful for some applications due to their parallel nature and optimality in terms of number of gates used for certain processes.

The capacity of the FGPAs to adapt to different situations and needs and their programmable nature make them a perfect fit for multiple applications and for different markets. These areas go from equipment for video and imaging, to circuitry for computer, automotive, aerospace and military applications. FPGAs are particularly useful for prototyping ASICs or processors, as the device can be reprogrammed until the circuit is bug-free and optimized. After this, it is more secure to manufacture the ASIC or processor as it has been tested. FPGAs are not only useful for the industry, but also for the individuals and enterprise business, as they can be programmed to carry out tasks as data analytics or encryption.

As it has been said, FPGAs are useful for prototyping purposes. The circuits that are being developed can be programmed in this type of device before manufacturing it in order to test their behaviour. This practice allows detecting errors before producing the ASIC.

It is easy to deduce that testing this circuits correctly is key in the process of developing circuits and has as a possible consequence saving money. One of the reason that push engineers to invest time and money in these tests is to detect the errors that have been committed during the design of the circuits, that have as a consequence that the circuit does not behave as expected. Other reason is to optimize the design, once the device works as expected, it is possible that the timing or the consumption of resources is too elevated, and these tests can analyse the performance.

As it can be seen, not all the testing has the same purpose, and depending on the objective of the designer, the test used for the circuit can be different. Two of the most used are the following ones.

The first one is the simulation. In this kind of test, the designers used a specific software, which can be different depending on the FPGA tested and the purpose of the examination. The simulation allows to, as it name implies, simulate the behaviour of a determinate HDL block depending on the environment created by the designer. This tool has as advantage that does not require any specific material, not even the FPGA, as it is done in the computer. It allows the user to detect problems with the signals or the design.

The disadvantage of this testing method is that it cannot be tested in the hardware and some conditions cannot be programmed in the simulation, so there are some errors that may not be detected, as it can be the interaction of the environment in the FPGA.

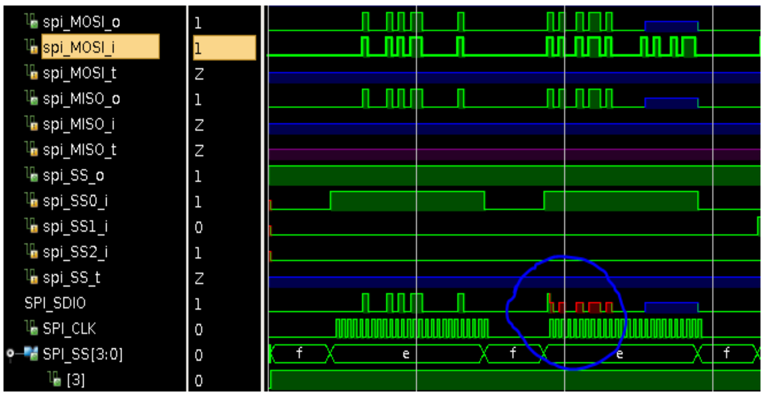


Figure 1. Simulation test

Another type of testing is called real-time condition. As its name implies, it consists on trying the design directly on the hardware, this is on in this case, in the FPGA. Evidently, this test requires more resources than the other one explained, and it may not always be a possibility for all the cases.

This method is called real-time as “it is a system where the result correctness depends not only on the logical result, but also at what time the results arrive.” (2) Therefore, the outcome of this type of tests is not only the logical result, but also the performance of the circuit in terms of timing and sequentially of the tasks.

The conclusion is that the designer is the one that must decide which kind of test is needed for each situation and circuit and can even choose multiple options, but what cannot be doubt is the importance of this step in the process of creating new circuits.

1.1. Motivation of the project

This project is born from the need to create a low-cost system capable of testing circuits in determined situations.

As it has been said, in order to confirm that a certain circuit works as desired, it is necessary to test it. This step of the design can become a big expense and may not be affordable for all the projects. Although the simulation can be considered cheap, as it does not require from any equipment but the computer and the timing that it lasts, sometimes it does not offer the conditions that are needed, and the results are not complete.

There are situations in which the designer has no other option but to test the designed circuit in real conditions, implementing it in an FPGA. This can happen when the environment in which the circuit will be display presents an anormal characteristic.

This is the case of this project, as the board that must be tested is situated in a radiated environment. The real-time condition test is the solution for this situation; however, this technique requires more resources, time and hardware, and this raises the price of the test.

In the market, there are available different professional testing tools. However, most of the ones that can provide some of the necessities for the project, be a real-time benchmarking with high accuracy and customizability, are usually very expensive and are not usually affordable.

The solution is, as already proposed, to develop a low-cost testbed system that allows testing the desired circuits in real-time conditions.

The advantage of this resolution is that it gives the possibility of creating a system according to the needs of the future projects of this department, but also allows it to be as general as it is desired.

1.2. General objectives

The aim of this project is, as it has been said, to develop a low-cost testbed for real-time conditions. However, this objective is too general, and it is necessary to specify more and stablish some other objectives.

First, developing a testbed requires time and effort, and it is not possible to design a specific one each time it is required for a different circuit. For this reason, it is desired that this system can be reused for as many circuits and situations as possible, independently of the design or the environment.

Testing a circuit requires a great amount of inputs and outputs, but this number may be different from one design to another. This complicates the task of developing a reusable testbed, as it must be modified each time that the circuit that must be tested changes. It is necessary to find a way of reconciling all in one, and this is done by stablishing a parametric testbed.

Parametrization is a mean of achieving re-usability in hardware design. By imposing a general structure of the number of the inputs, outputs and even the internal structure of the testbed, it can be achieved the reusability. Independently of the situation, it is possible to modify the number of signals or even its configuration by only changing the generics. This does not take a lot of time, as it only requires changing some characteristics of the HDL code instead of the lines of the process.

The other imposition of the system structure is derived by the possible environment in which the tests must be perform. This system is designed, initially, for tests in which the circuits are exposed to hard environmental conditions. This project is thought for the situation in which the testbed has to be exposed to a big amount of radiation.

This condition difficulties the task of monitoring the results in real-time conditions. The solution that is proposed is to divide the system in two different boards, that are connected in order to share the information. In this situation, one of the boards is the Device Under Test (DUT), which contains the circuit to be tested, the Unit Under Test (UUT), and the modules that are required to communicate with the other board. The second FPGA is the monitoring system, the one in charge of the data, both the inputs of the test and the outputs. It is also the one that is connected to the computer and that can share the information of the situation of the test. For this purpose, it is possible to create an interface that improves the experience.

The following figure shows the general structure of the complete system with its connections and main modules.

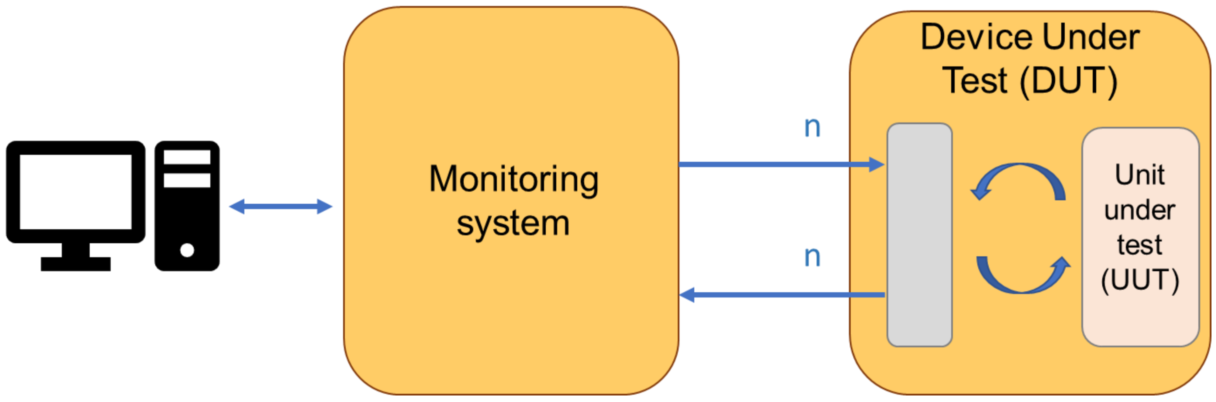


Figure 2. General structure of the system

As a summary of the main objectives of the system

1. Design a low-cost system for testing circuits. This means that the all the materials used, from FPGAs to the connectors, must be as cheap as possible.

2. The system must be implemented in two different boards, one will act as the monitor and the other will be the device under test

3. The structure of the DUT board is the one that follows. There will be a module which will contain the circuit that must be tested, the other module will contain the required HDL to control the communication.

4. The monitor system is the board that controls the complete system. It is the one in charge of giving the inputs to the DUT using the channel available between them and receiving the outputs in the same way.

5. It must be implemented a fluid communication between the boards. Regardless of the number of inputs/outputs of the test, the monitor has to send all the data to the DUT in packages of n bits.

In order to fulfill the objectives of the project, here is a proposal of the steps that have to be followed:

- Research the existing systems that completely or partially meet the requirements just proposed.
- Choose the FPGAs to be used both for the monitor and the DUT.
- Determine how the channel between the boards is going to be, this is, number of buses and their width.
- Design the communication modules that allow the systems to exchange the desired information
- Search the possible UUT that can be used to test the system
- Test the designed system and examine the results

Chapter 2

Theoretical basis

This chapter is dedicated to the explanation of the theoretical ground of the project, as well as the study of the existing systems. First, it is analysed the general structure of the design that is going to be implemented and the reasons of it. Then it is studied the FPGA used for both systems, the Pynq-Z2. It is also explained the reasons of this decision. Last, some projects which have similar systems to the one that is being developed in this one are analysed and commented in order to find the useful information.

2.1. Testbed

The purpose of this project is to develop a real-time monitoring system to test circuits or prototypes, in other words, a testbed.

Testbeds are composite abstractions of systems and are used to study their components and interactions to gain further insight into the essence of the real system. (3) In other words, it is a composition of real parts of the system and the prototypes that want to be studied and analyzed in order to determine whether the functioning is the desired and the performance is also the expected one.

The testbed provides a realistic hardware-software environment that allows to test components in a real-time condition and allows the designers to make decisions based on the behavior of the system.

One of the disadvantages of the testbed is that in general they are limited due to their cost of design and implementation. The aim of this project is to reduce these costs in future occasions by creating a system that can be used with no limitations. The decision of using a testbed over other methods for testing depends on the balance between the cost and the benefits derived from the results that can be obtained.

Testbeds are composed of three different subsystems, with which it is possible to experiment with a wide range of different stimuli, configurations and applications.

- An experimental subsystem. This is the collection of real-world system components and/or prototypes that are wished to be modeled and experimented with.

- Monitoring subsystem. Consists of the interfaces to the experimental system to extract the data of the test and analyze this information in order to understand the results and the circuit that is being tested.
- Simulation-stimulation subsystem. Provides the tools necessary to provide the experimenter with real-world system inputs and outputs to provide a realistic experimentation environment.

In the system that is developed in this project, the DUT board acts as the experimental subsystem, as it is the one in which the test circuit is implemented. The monitor FPGA acts as the monitoring subsystem, as it is the one in charge of collecting the data from the test. The simulation-stimulation subsystem does not belong exactly to only one board. The monitor is the one that creates and sends the inputs to the DUT, however, the DUT is the one in charge of collecting the data and feeding it to the UUT.

In the following image, it can be seen the structure of the testbed applied to the one already proposed in the last chapter for the system, that consists of two different boards connected by some channels.

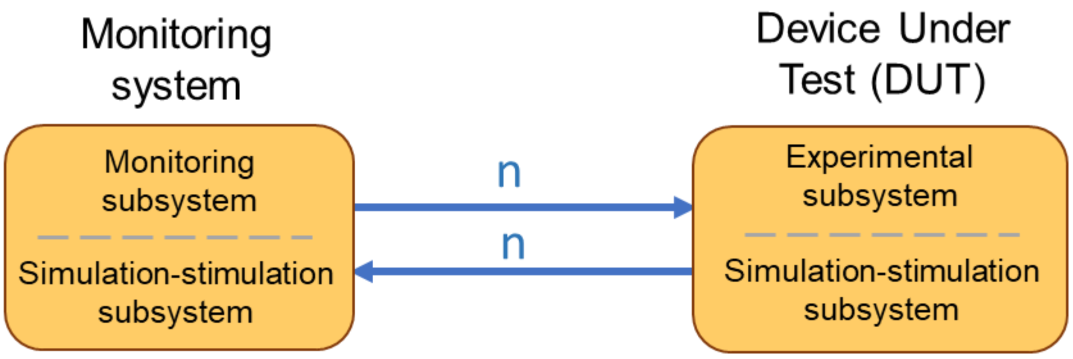


Figure 3.Structure of the testbed in this project

It is important to remark that the results of the testbed depend on the capacity of the monitor subsystem to extract and analyze the results considering the real-world phenomena that may be interacting with them.

2.2. Pynq-Z2

As it has been already explained, it is a requisite for the project to design a testbed composed by two boards, so that each subsystem is implemented in one different FPGA. However, nothing has been said about the need for these two boars to be a different model or the same, and this is a decision that has been taken during the development of the project. There is a large number of different FPGAs available in the market, and each of them have different characteristics.

At the start of the project, it has been decided the choice for the monitoring system board, which is a PYNQ-Z2. For this reason, the only decision to be taken is the FPGA used as DUT, which, as it has been said, can be the same one or different. For the sake of simplicity, the DUT board is also a PYNQ-Z2. The main advantage of choosing the same board is that the functioning is identical, which leads to a situation in which the problem of coordination is minimized, and only one FPGA has to be study. It also easies the communication between both boards, since the same number of pins are available.

PYNQ-Z2 is a TUL board based on Xilinx Zynq SoC (System On Chip), which is a chip that is composed by all the modules that are necessary to make the system work. This board is designed to support PYNQ framework, which is an open-source project from Xilinx designed to easier the use of its platform. PYNQ stands for Python Productivity for Zynq, and by the use of the Python language, libraries and overlays, designers can optimize and improve their use of the programmable logic and microprocessors to develop their electronic systems. (4)

It is easy to deduce that this platform is intended to be used by software developers wanting to avoid designing hardware or by hardware designers wanting to create a product that can reach the widest possible audience or wanting to easy software interface. Since this thesis project is non on any of these situations, this PYNQ framework has not been used, and all of the software has been created from scratch.

Apart from this Zynq SoC, the board is composed by many other elements, such as USB, Ethernet port, two HDMIs, four LEDs, four push-buttons, two slide switches, and many other components. In the following figure, the board is shown and some of this elements can be seen. (5)

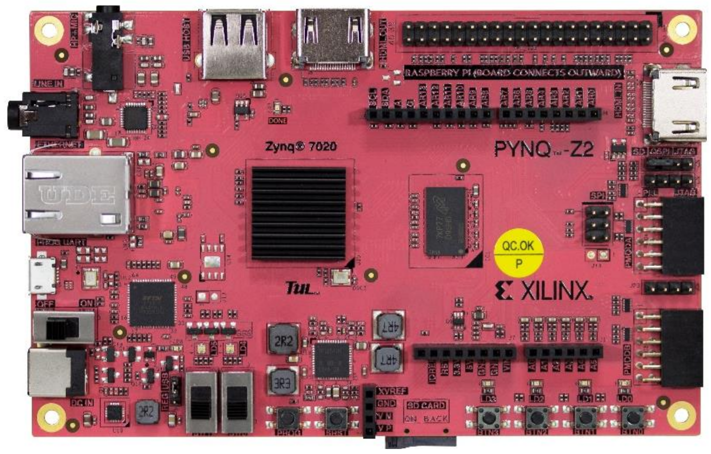


Figure 4. PYNQ-Z2 board

Apart from the components already mentioned, there are some other elements that are found specifically useful for this project. First, as it can be seen in the previous image, there is a big variety of connector pins available on the board, which are needed to connect both boards and allow the communication between them. In the top left-hand corner, there is a Raspberry Pi header connector, which can be used as one element, or each pin can be used individually as a GPIO (General Purpose Input Output). Underneath this element, there is an Arduino header, which can be used in the same way as the Raspberry Pi one. Apart from this two, there are available two standard Pmod ports on the right part of the board and under de HDMI-in. combining all these elements, the number of pins present on this board is high.

There are two ways of powering this board, the first one is from USB, connecting the FPGA to a compute. The other way is by connecting it to an external power source of 7V-15V. The power source selection of the board is done by a jumper located on the left of the slide switches.

Another advantage from this board is that it has multiple ways of being programmed. It can be done from JTAG, which is through a cable connected to a computer, Quad-SPI flash, or with a MicroSD card, which allows the system to be independent in terms of being connected to a computer. The jumper for boot mode selection is located between the HDMI-in and the upper Pmod.

As for this project, the board is programmed through the JTAG, being always connected to the computer. For this reason, and taking advantage of the situation, it is powered via USB, connecting it to the computer as well.

2.2.1. ZYNQ SoC

As it has been said, the PYNQ-Z2 is based on the ZYNQ SoC, which is the chip that controls the functioning of the FPGA, in other words, it is the heart of the board. In order to get a successful system running on these two boards, it is necessary to understand the functioning of this chip, as a way of assuring that it is programmed as desired.

The ZYNQ chip is considered an all programmable SoC, and this is due to its software, hardware and IO (Input and Output) programmability, all in only one single chip. The architecture of this component is divided in two principle parts: the processing system, PS, and the programmable logic (PL). The following figure shows the architecture of this component, displaying as well as the two different parts already mentioned. (6)

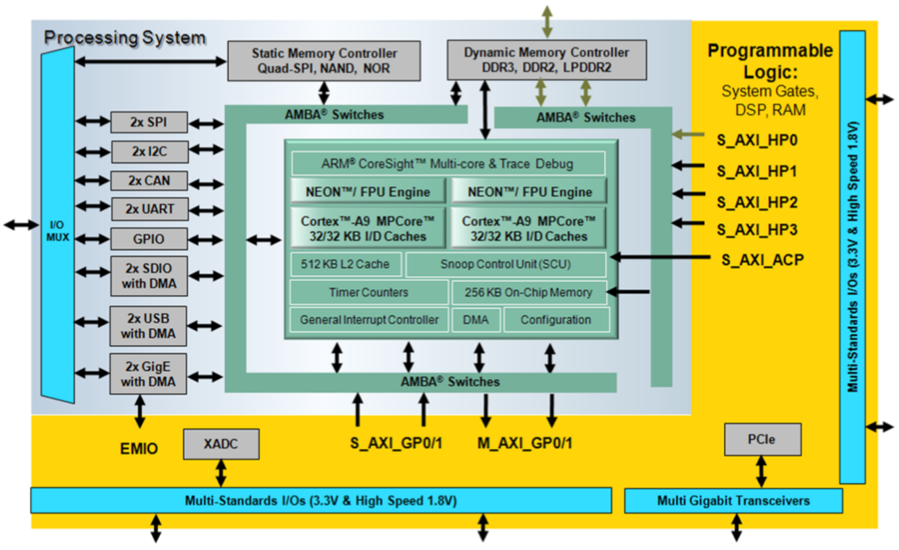

Processing System
Static Memory Controller Quad-SPI, NAND, NOR
Dynamic Memory Controller DDR3, DDR2, LPDDR2
AMBA® Switches
AMBA® Switches
Programmable Logic: System Gates, DSP, RAM
I/O MUX
2x SPI
2x I2C
2x CAN
2x UART
GPIO
2x SDIO with DMA
2x USB with DMA
2x GigE with DMA
ARM® CoreSight™ Multi-core & Trace Debug
NEON™/ FPU Engine
NEON™/ FPU Engine
Cortex™-A9 MPCore™ 32/32 KB I/D Caches
Cortex™-A9 MPCore™ 32/32 KB I/D Caches
512 KB L2 Cache
Snoop Control Unit (SCU)
Timer Counters
256 KB On-Chip Memory
General Interrupt Controller
DMA
Configuration
S_AXI_HP0
S_AXI_HP1
S_AXI_HP2
S_AXI_HP3
S_AXI_ACP
AMBA® Switches
EMIO
XADC
S_AXI_GP0/1
M_AXI_GP0/1
PCIe
Multi-Standards I/Os (3.3V & High Speed 1.8V)
Multi-Standards I/Os (3.3V & High Speed 1.8V)
Multi Gigabit Transceivers

Figure 5. Architecture of Zynq SoC *(8)*

As it has been said, the chip has software programmability, which is offered through a complete ARM based processing system. All Zynq-7000 devices have the same processing system, which easies the migration of applications from on device to another. The specific chip included in the PYNQ-Z2 board is the Z-7020.

This processor has a centric architecture, which means that Zynq 7000 devices boot and operate like any other processor and does not need the programmable logic to be configured in order to operate. In other words, there is no need of any Intellectual Property (IP) block from the programmable logic part to run the processing system. Furthermore, since both parts operate in different power planes, the programmable logic does not even have to be powered.

The hardware programmability aspect is added with the programmable logic, which is used to extend the capability of the processing system by adding peripherals or accelerators. This characteristic gives the designers the possibility to create their own version of the processor for their own applications. The programmable logic is always configured by the processing system, once this one boots. This configuration is performed by application software which access the hardware device configuration unit.

As it has been said, the processing system is the same in all the Zynq-700 family, however the difference between them lies in the programmable logic. Its complexity scales in order to accommodate more or less customization depending on the application, and the number of inputs and outputs available.

The preferred option to exchange data through programmable logic and processing system is the AXI (Advanced eXtensible Interface). There are nine AXI interfaces, where master and slave are considered form the point of view of the processing system. There are two General Purpose masters (M_AXI_GP), 2 General Purpose slaves (S_AXI_GP), four High-Performance slaves (S_AXI_HP) and one Accelerator Coherency Port slave (S_AXI_ACP).

The M_AXI_GP interfaces are mainly used to control IP cores in PL. Each one has a memory range to originate AXI transactions. The CPU is the one in charge of data movements, which is simpler from a software perspective and simplifies the PL design because a slave is implemented. However, the disadvantage of this interface is the spent of clock cycles moving the data and that the amount of data is limited. (7)

Using this interface for transferring data makes it simpler, since any programmable logic block connected to the processing system through an AXI interface is mapped in the memory and therefore accessible in a simple way.

The IO programmability is achieved through a large set of inputs and outputs. One of the main characteristics of the Zynq, as it is of the FPGAs, is its flexibility to adapt to almost any interface available in the market, which enables the chip to communicate directly to almost any element of the system. To connect the Zynq to other components, there are two possibilities.

The first is through the fifty-four dedicated peripheral IO part of the processing system, which are called MIO (Multipurpose Input Output). This means that these peripherals are connected regardless of whether the programmable logic is configured, and this IOs are running as soon as the processing system boots. In the PYNQ-Z2, these MIO pins are already used and connected to other components of the board, and therefore cannot be used for the communication purpose. These pins can be extended into the programmable logic and provide up to sixty-four GPIOs which are called EMIO (Extended Multipurpose IO).

The other possibility is to use the IOs from the programmable logic, which is done by connecting the peripherals to this last one. The programmable logic is surrounded by IOs that can be configured to interface with almost any component outside the chip.

It is important to remember that all the connectors that have been named, as the Arduino or the Raspberry Pi headers, are already mapped and connected to the chip, but, in the case of the monitoring subsystem, some of this signals must be connected to other modules in order to enable a communication between boards, and for this reason it is important to understand how the inputs and outputs are handled by the processor.

Apart from what has been explained, the Zynq SoC is also in charge of generating the clock used by the programmable logic. This task is important for the project, because the modules that control the communication between the boards must be synchronous. The clocks for the processing system and the programmable logic are different, but at the same time are synchronized.

For the processing system, there is a master or primary clock, which is produced by the oscillator available in the board and there are three PLL (Phase-Locked Loop) that can be used to generate other clocks. As it can be deduced, the generated clock is a version of the primary one which is modified by being multiplied, divided or phase-shifted. (8)

The programmable logic can be sourced from an external clock capable pins, but for this project this is not the solution taken. The other option is to use one of the four processing system clocks as a source. This option is not available in the other direction, which means that the programmable logic cannot supply clock source to the processing system.

By controlling the generated clock of the processing system and connecting it to the programmable logic, this part can get a clock whose frequency is the one desired for the designed modules.

2.3. Related works

Before starting with the methodology, it is advisable to do some research to find the different systems that have already been designed. This can be useful to get ideas of how to face the development of this project. Although it may not be possible to find systems as similar to the one that is going to be created as it is desired, it may be interesting studying different implementations in the FPGA selected, the Pynq-Z2, and finding a real-time monitoring system as well.

First, it is interesting to analyze why there are systems that are implemented in more than one board, as well as how this is done. As it has been explained, sometimes the circuits are designed to work in an environments in which the conditions may change the way of behaving of the hardware or even be dangerous. In this situations, it is desired to minimize the amount of components that are exposed to this environment, and the way of doing it is by using more than one board. The first one will contain the circuit or design that has to be tested in the real-environment, the second one contains the rest of the necessary hardware to carry out that exam.

One example of this situation is aging tests. Modern FPGAs are built using the most advanced technology to meet performance and power demands. However, this makes them susceptible to various reliability challenges at nano-scale. It is important to study the parameters and phenomena than influence in this degradation of the FPGAs. The way of doing it is by the aging test that consists on subjecting the FPGA to a situation of high temperature and high voltage to facilitate the degradation. (9)

As it can be supposed, those conditions are hard to stand for any hardware, and it would difficult the task of monitoring the results of the test in real-time. Also, it would made necessary to change the device frequently, as it the degradation suffered would worsen the performance. The solution proposed for this system in a determined project is to use one board for the tests of the FPGAs, and another one in charge of controlling the test and collecting the results. Also, it uses another board for the power supply of both boards and another one for the task of connecting to a computer to transmit the data collected. (10)

In the following image, it can be seen the structure proposed for this aging test.

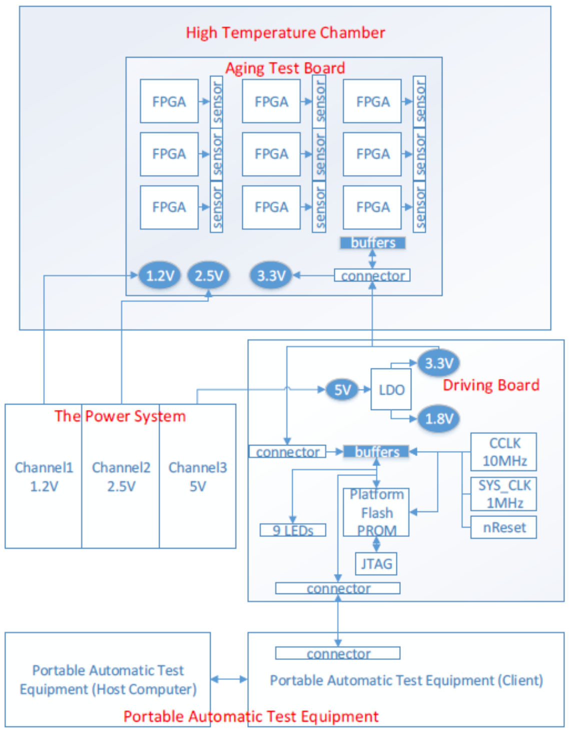


Figure 6. System architecture of aging test (10)

From this type of tests, it can be concluded that sometimes this is the necessary schematic and that it can be implemented. However, the scheme is not exactly the one that is used for this project, as the connection between the driving board and the Aging test board is not as critical. In the project, the communication is essential for a correct test development, and therefore requires more attention.

This communication is one of the most complicated and time-requiring parts of the design of the testbed. The connection cannot be made with a connector, like in this example, and the modules that must assure the interaction between boards are also needed.

Although the structure may be similar, there are a lot of differences with the system of this project, starting from the data flow just explained and continuing with the boards that are used. It is necessary research for some projects in which the board used is similar to the one of this system.

As it has been explained, the Pynq-Z2 is divided in two parts, the processing system and the programmable logic, and it is necessary to understand how this division works and how it has to be connected to develop an efficient design that can be implemented. In the following images, it is shown the designed structure implemented for two different objectives.

The first one is a testbed for experimental benchmarking of algorithms competing in cryptographic contests. The idea is that the message, or input data, is sent from the PS, where it is received, to the PL, where the hash core can do the necessary calculations, and after that the hash value is transferred back to the ARM core in the PS. The two parts are interconnected by AXI interface blocks, which transfer the data by the memory mapped mode. (11)

This is a first approach of how to implement the communication between the two boards. It is possible to develop the modules for the programmable logic, and then connect them to the PS using the AXI interface in order to use them in the software application.

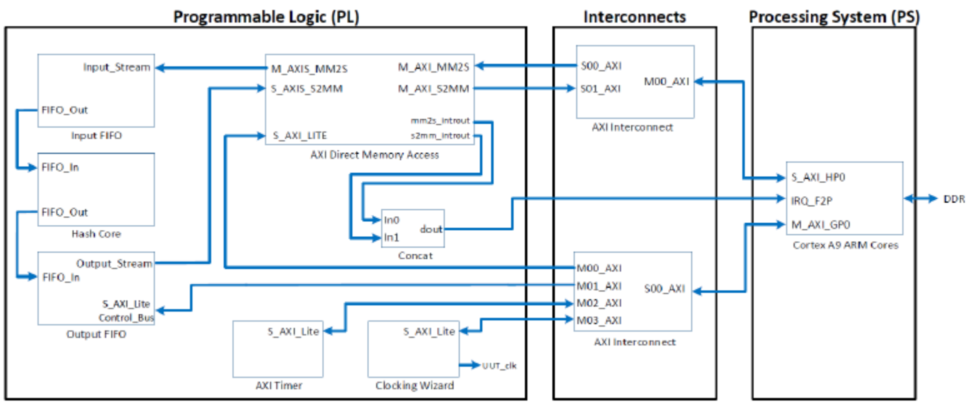


Figure 7. Block diagram of the testbed (11)

The second system is developed to test and simulate SEU (Single Even Upsets) in Zynq BRAM and flip-flops. The design of this testbed is like the last one, the built-in self-test is designed for the PL side on the Zynq, and the PS is the controller. (12)

The PS oversees transmitting the computer control commands and the test results. The PL contains the test subsystem and the UUT.

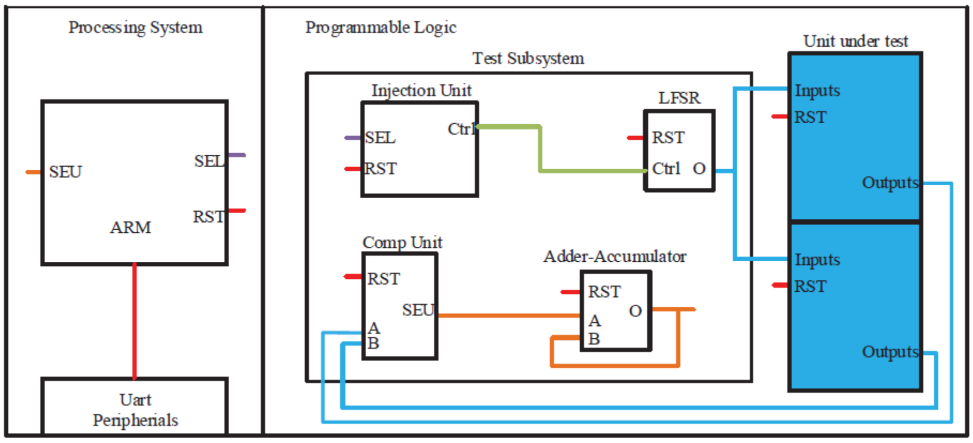


Figure 8. Built-in self-test scheme (12)

It is obvious that the general structure of all the designs developed for a Zynq follow a similar structure. The PS is the part that oversees, in general, the test and controls it, while the PL contains the blocks of the test. The communication between both parts may or may not be necessary.

Chapter 3

Methodology

This chapter is devoted to the explanation of the development of the project itself. First, the tools that have been used are presented, which refers to the software, which is the program in which it has been done the design to control the boards. After this, it is explained the operating principle of the testbed system, focusing in each individual part that composes both subsystems, the monitoring board and the DUT, and making an exhaustive analysis of it. Last, it is explained the software application created for the monitor system.

3.1. Software tools: Vivado and SDK

As it has been explained, the main reason of using FPGAs is their flexibility to be programmed and configured in multiple ways in order to meet the requirements of the applications established by the designers. For this purpose, it is necessary to use a software which allows to create the design desired and after that, program it to the FPGA. For this project, it has been necessary to use two different programs: Vivado Design Suite and Vivado SDK.

3.1.1. Vivado Design Suite.

It is a software suite produced by Xilinx and designed for the synthesis and analysis of HDL (Hardware Description Language) designs. It was presented in 2012 and is an IDE (Integrated Design Environment).

HDL is a specialized computer language used to describe the layout and behavior of an electronic circuit or digital logic circuits. It is also used to stimulate the circuit and check whether or not the response is the one desired. (13) There are many HDL languages available, but the most popular ones are VHDL, that comes from the combination of two acronyms, VHSIC (Very High-Speed Integrated Circuit) and HDL, and Verilog. For this project, VHDL is the language used for both the synthesis and the simulation.

Although HDL looks similar to any other programming language as C, it is important to remember that by using this type of code what it is really being done is describing real hardware and it is a concurrent programming language. Therefore, it is necessary to identify which portions are combinational logic, which ones are sequential circuits and when it is being used a state machine.

One of the advantages of the Vivado Design Suite software is that it is possible to choose the board which will be used for the project, which is required when creating a new project in Vivado. The PYNQ-Z2 board is not available, but it can be easily downloaded from the official Pynq website and imported to the software. This file allows the software to configure the Zynq processing system settings

The Vivado Design Suite is used to synthesize and simulate and for this reason is composed by different components that easies the designing task:

- Vivado IP Integrator. This tool allows the designers to quickly integrate and configure different IP blocks. These units can be selected from the ones available in the XILINX library, but also allows to add IP blocks created by the designer. This modules are added as sources that are written in Verilog or VHDL language.

  One of the advantages of using the Vivado software is within its library a Processing System IP is available for the Zynq-7000 SoC, which can be customized to meet the requirements of the designer.

  In addition to all of this, there are some tools available for the block design such as the "Validate Design" tool that check the correctness of the connections and IP block configuration, or the "Run Connection Automation" which, as it names implies, connects de IP blocks automatically.

  In the following figure, it can be seen the view of the IP integrator with some blocks interconnected.

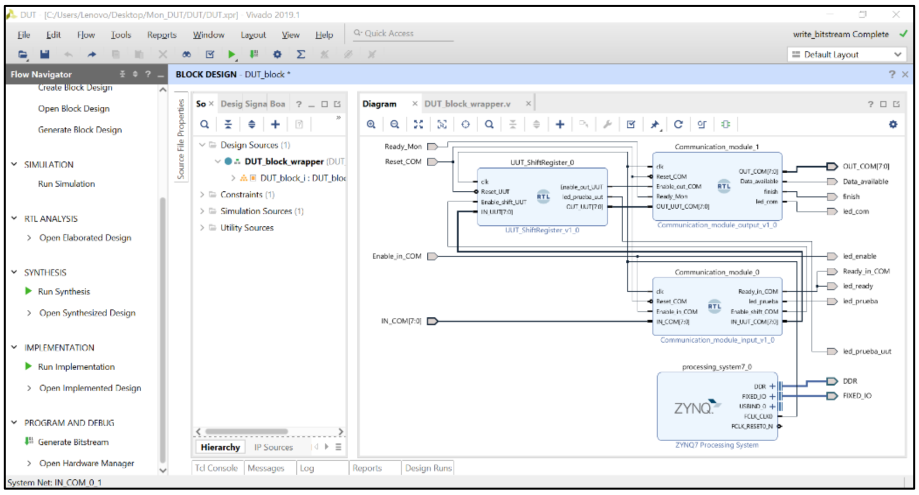


Figure 9. Vivado IP integrator window

- Vivado Simulator. This tool allows the engineer to simulate the block design or HDL module programmed. By creating a simulation stimulus for the inputs of the system in HDL language, it can be seen the reaction of the system, showing the result of the outputs.

In order to learn how to use this tool, there is a big amount of documentation available in the website of Xilinx, and in particular there is one that explains step by step how to run a simulation. (14)

In the following figure, it is shown the result of a simulation in Vivado Design Suite. All the input and output signals of the design are available to be added and displayed. But it must be taken into consideration that the simulation does not contemplate the time of driving the signals from one point of the FPGA to another. For this reason, it is possible that a design that works in the simulation, may not work in the FPGA.

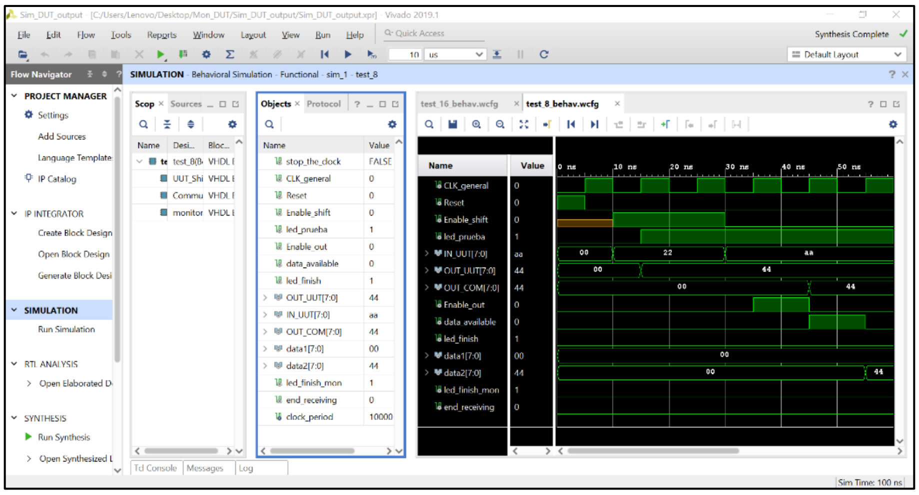


Figure 10. Vivado Simulation window

- Vivado Synthesis. The synthesis is the process of transforming RTL (Register-Transfer Level) design into a gate-level representation. (15) In other words, is the process of turning a specification of the desired circuit behavior, specified in HDL, into a design implementation in terms of logic gates that can be implemented in an FPGA.

  For this purpose, it is necessary to add the constrains of the design. This constrains are used to connect the signals of the design to the physical pins of the board. There is a file available in the board's producer website which contain the constrains and is ready to be modified with the name of the signals of the design. The extension of this file must be XDC, since no other extension is permitted in this software.

Once the synthesis and the implementation of the design has been made, it is possible to generate the bitstream, which is a file that contains the programming information for an FPGA. Once this file is available, it can be uploaded to the board in order to program it. For this project, this can be done through the hardware manager and connecting the FPGA via JTAG to the computer, since there is no desire of using an SD card to program the board.

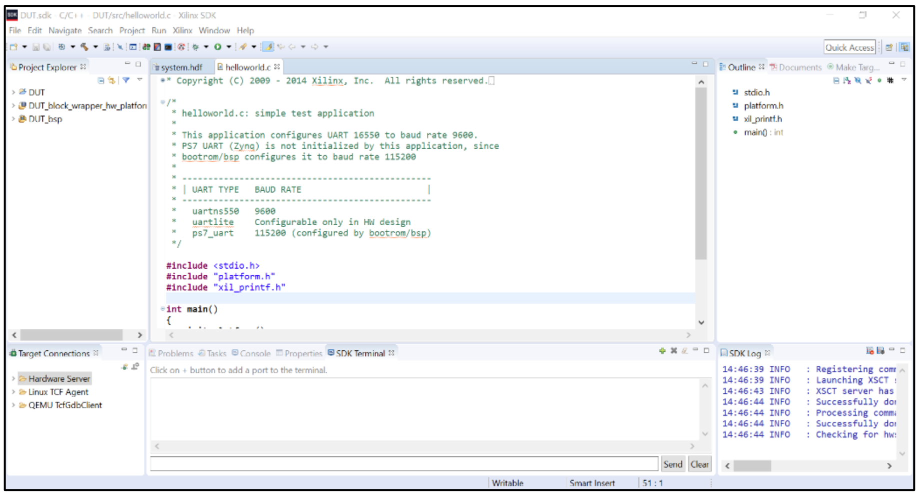
Once the bitstream is programmed in the FPGA, this board is configured as desired, and it can be tested the functioning of it. If all the process of designing and creating the constrains has been done correctly, then the FPGA should behave as expected.

However, in case it is needed software development to control some of the IP blocks or functionalities of the board, it is necessary to use another program, which is XILINX SDK.

3.1.2. XILINX SDK

The acronym stands for software development kit and is an IDE for development of embedded software application for Xilinx embedded processors.

The way of proceeding is by first exporting the bitstream generated to SDK, which can be done directly in the Vivado Design Suite program. This allows the system to create the hardware platform specification, which captures all the information and files of the hardware design, in other words, all the information required for the software development for that specific hardware.



```
 * Copyright (C) 2009 - 2014 Xilinx, Inc.  All rights reserved.

/*
 * helloworld.c: simple test application
 *
 * This application configures UART 16550 to baud rate 9600.
 * PS7 UART (Zynq) is not initialized by this application, since
 * bootrom/bsp configures it to baud rate 115200
 *
 * ------------------------------------------------
 * | UART TYPE   BAUD RATE                        |
 * ------------------------------------------------
 *   uartns550   9600
 *   uartlite    Configurable only in HW design
 *   ps7_uart    115200 (configured by bootrom/bsp)
 */

#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"

int main()
{
```

```
14:46:39 INFO   : Registering comm
14:46:39 INFO   : Launching XSCT s
14:46:43 INFO   : XSCT server has
14:46:44 INFO   : Successfully dor
14:46:44 INFO   : Processing comma
14:46:44 INFO   : Successfully dor
14:46:44 INFO   : Checking for hw:
```

Figure 11. Vivado SDK window

After this, it is necessary to create a new application project, which contains the source files in C or C++ language, the executable output file, with extension elf., and associated files. When this project is created, it is necessary to choose the hardware platform used. For the standalone projects, each software application project must be linked to a Board Support Package (BSP), which is a collection of libraries and drivers that form the lowest layer of the application software. The available libraries for the project depend on the hardware design chosen.

Once the application has been created, so are all the files necessary to create the software for the application. As it has been explained, the BSP includes the libraries with the functions and definitions necessary to program all the IPs. It is also created the file which contains the main function. This is the file that must be modified by the designer. For this project, the language used for this purpose is C.

Once the software is created, it is possible to program the FPGA with the bitstream from SDK, and following this, run the software application in the hardware to test its functioning. If the board does not need any software application, it is not necessary to run the FPGA from SDK, as it can be done from Vivado Design Suite.

Another useful aspect of SDK is that it is possible to communicate with the FPGA, sending or receiving information from it in real time. This is done through the SDK terminal. By connecting this terminal to the board and programming the software application, the information can be shown in the computer screen.

3.2. Communication design

As it has already been explained, the objective of this project is to design a testbed system composed of two separate parts, the monitor and the DUT. The first subsystem is the one in charge of controlling the tests, in other words, is the one that sends the inputs to the UUT, and the one that receives and stores the outputs. The DUT is the one that contains the UUT. But these two subsystems must be connected in order to reach the final goal, get a real-time monitoring testbed.

The intention of the project is to get as final result a system that can perform as many cycles of test as possible, depending on the frequency of the boards. In other words, it is desired to ger as many rounds of sending information from the monitor system to the DUT, performing the test, and sending back the information as it is possible. For this purpose, a good communication between the two parts must be assured.

Such is the importance of this task that, as a first step of this project, it has been decided to develop a communication system to be simulated and tested for one single cycle and board. The idea is to develop the VHDL modules which send and receive the information, regardless of the data used, that can be a constant. Once this modules are created and tested, they can be implemented in the boards and all the adjustments required can be done.

This section of the document is entirely dedicated to the explanation of the basic communication module. As explained, the system is simplified, and the monitoring system only sends simple signals and constant data, in other words, there is no real interaction between the two boards. As a consequence, in this stage there is not a real-time monitoring system.

In the following figure, a generic communication system is proposed between the two subsystems, which are connected by an undetermined number of pins. The quantity of signals (n) that can be transmitted each time depends on the boards used for the subsystems, the monitoring and the DUT, which are in both cases, as already said, Pynq-z2

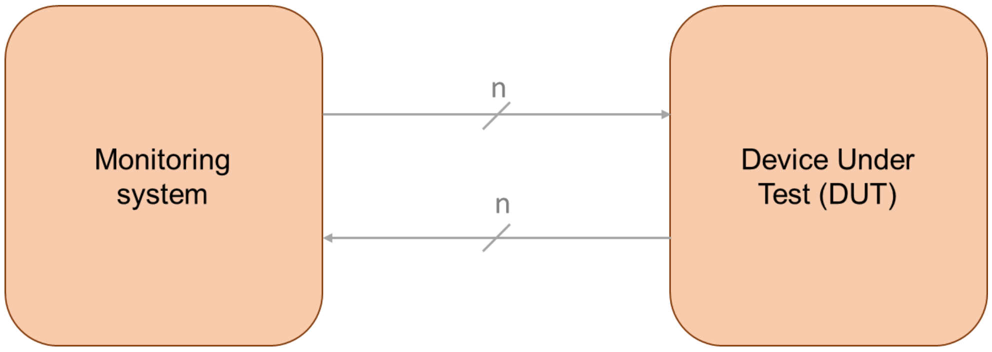
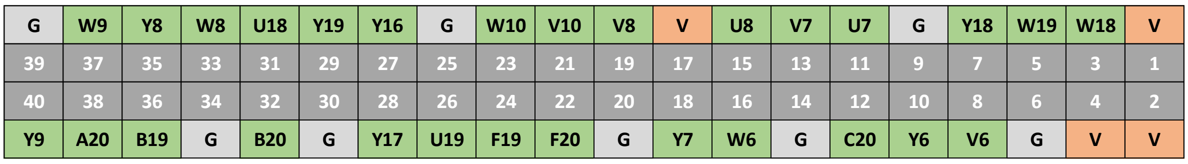


Figure 12. Scheme of the complete system

The Pynq-Z2 board is composed by different headers which can be used to interconnect the two systems. Since the desire is to maximize the width of the communication's channel, but simplifying the amount of connections, the Raspberry Pi header has been chosen from above the other possibilities. Only one of the headers is used, dismissing all the other possibilities presented in the board, as the Arduino pins.

As it can be seen in the following image, the Raspberry Pi header is composed by forty pins, but according to the schematic, which is available in the manufacturer's website (16), only twenty-eight are available to be connected and used as GPIO (General Purpose Input Output). The rest of them are pins used as ground of the system or power supply. It is important to take into account that some of those pins are shared with other elements of the system, as the Pmod, and they can be used as long as those components are not. Considering all the



G	W9	Y8	W8	U18	Y19	Y16	G	W10	V10	V8	V	U8	V7	U7	G	Y18	W19	W18	V
39	37	35	33	31	29	27	25	23	21	19	17	15	13	11	9	7	5	3	1
40	38	36	34	32	30	28	26	24	22	20	18	16	14	12	10	8	6	4	2
Y9	A20	B19	G	B20	G	Y17	U19	F19	F20	G	Y7	W6	G	C20	Y6	V6	G	V	V

Figure 13. Layout of RaspberryPi header: in green the available pins

information just given, the conclusion is that the maximum amount of data that can be transferred each time is limited by this twenty-eight and are the pins colored in green. The nomenclature used to name those pins is not casual, as it is the one used in the schematic and the constrains file necessary to implement the design in Vivado, .

However, the communication needs some other signals apart from the pins transferring data, as a enables or resets. Also, it must be considered that, in case there is a desire of having different pins for each communication's direction, the width of the channels is considerately decreased. On the other hand, having the same pins used as input and output complicates the task of designing the functioning of the communication, as it is needed this signals to be tristate. It is for the combination of all the past reasons exposed that the communication width has been set to sixteen pins. This decision leaves enough pins for other purposes, as the ones mentioned.

From this point, two possibilities are presented. As explained, these sixteen pins can be used as bidirectional, which means that will be used as input and output at the same time. The other option is to divide them in eight pins for the input and the remaining eight for the output. In terms of efficiency, the first option is the best one, since it assures a faster communication between the systems, but in terms of execution it complicates the development of the module. The other case, although it is slower concerning the communication, its implementation is more direct, and it is for this reason that it has been chosen as the final configuration.

This decision supposes some challenges. First, eight pins are a low number of signals, which means that the process of sending and receiving information needs to be done, probably, in multiple phases. This leads to the second problem, as both subsystems need to be synchronized in order to successfully fulfil this task.

The next decision is related with the module that oversees the communication, and several options are within reach. The code can be developed either on the monitoring system or the DUT, but before deciding, it is necessary to explore the challenges of the communication system.

As it has been already mentioned, the number of pins is low for the purposes of this system, as it only allows to have eight inputs/outputs for each test. The way of solving this issue is by dividing the transfer of data in several packages, which size is set to the maximum available. Two different scenarios are then presented.

First, if the number of signals to be sent is fewer than eight. In this case, all the information can be sent in only one package, and the transmission is fulfilled in the less possible time, as it only takes one cycle.

The second scenario is presented when the number of signals is greater than eight. In this situation, the data must be divided in packages and sent one after the other. The timing in this situation is crucial, as there must be enough space between each lot so that it can be prepared, sent, received and store. This scenario is the one that requires more time, and its duration depends on the number of packages that have to be sent. Also, it is the one that requires more control in order to achieve a correct transmission.

As the system must be prepared for any type of test, regardless of the number of inputs or outputs, the communication requires a control module which carries out the task of splitting and colleting these lots of signals, and it can be implemented in either one of the two subsystems.

In order to simplify the control module of the monitoring system, the unit in charge of the communication tasks is placed in the DUT, and it is the interface between the Unit Under Test (UUT) and the monitoring system. The language used for this purpose is VHDL, as already stated. In the following figure, it can be appreciated the final scheme.

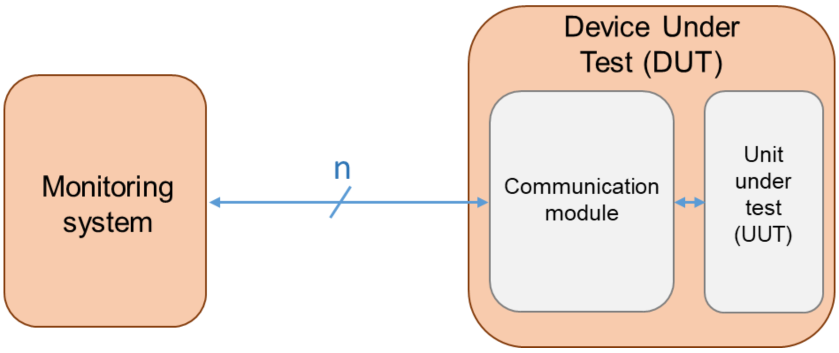


Figure 14. Scheme with details of communication module

From this point forward, the communication will be referred to the DUT point of view, since it is where the module will be implemented.

3.2.1. Communication module: general structure

The first step in the path of achieving a successful communication system is to design a module that can be simulated and therefore, tested. In the following figure, it is presented the basic structure of the communication module. This unit is composed by the following inputs and outputs:

- CLK: This signal is the clock of the system and it is generated, as already explained, by the processing system of the Zynq. It is required in order to make the modules synchronous, which means that the tasks are only carried out when there is a rising or falling edge of this signal.

  This input is the one that has to be modified to study the behavior of the system with respect to the frequency, aim of this project.

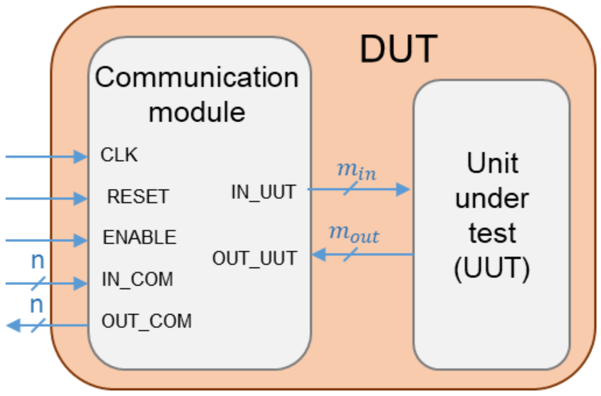


Figure 15. Communication module with inputs and outputs

  One of the decisions to be made is if this signal is shared by the two boards, in other words, if it is created by the monitoring system and then sent to the DUT as a signal, using a cable to join the pins. By doing it this way, both boards would be synchronized, which would ease the communication, but since the cables used are not well-isolated, there could be a lot of noise in the clock used by the DUT. Since this noise can be dangerous, each board generates its own clock, but always with the same frequency.

  It is necessary to clarify that for the simulations this signal is generated as a stimulus in the corresponding file.

- ENABLE: This signal is sent by the control module of the monitoring system, and it indicates when the communication process is about to begin, so this last module can prepare for the tasks. For the VHDL code, two enables have been used: enable_in, that triggers the input communication and enable_out, that does the same as the one before, but with the output transmission of information. Their working principle will be explained later.
- RESET: As its name implies, this signal stops all the transmission procedure and sets the communication system to its initial values. This signal is asynchronous, as it will be explained.
- IN_COM and OUT_COM: These two signals are the ones used for the communication between the monitoring system and the DUT. Their width is always lower than n=8, as it has been set like this.
- IN_UUT: After all the packages have arrived at the communication module, the data is then gathered in this signal and it is later sent to the UUT. Its width (m_{in}) depends on the number of inputs of the system.

- OUT_UUT: After the test is carried out, the output data is sent to the communication module, which is in charge of splitting it into packages and sending it over the monitoring system. Its width (m_{out}) depends on the number of outputs of the total system.

Although the general structure of the module is defined for every test that can be carried out in the UUT, the total number of inputs and outputs differ from one case to another. It is for this reason that is necessary to create generics in order to define the component. With these generics, m_{in} and m_{out} are set in each case manually before starting the test, as it is necessary to establish the size of IN_UUT and OUT_UUT vectors. Even though the system is being programmed for a maximum size of a package of eight bits, this value is set as a generic as well, this way it is easy to change this parameter in case there are more pins available in the future.

The development of the code for the communication module can be divided into two parts: first, receiving split information and gathering it together in IN_UUT, and second, taking the information from OUT_UUT and it in order to create packages that can be send through the bus OUT_COM. Each of these parts are handled in two different processes, but both are triggered by these signals: the clock, reset and its corresponding enable. It is important to mention that the reset is the only asynchronous signal, and when its value is set to zero it triggers the processes independently of the value of clock, setting all the signals to their initial value and this is due to safety reasons.

The next chapters will explain in depth how the receiving and sending processes are programmed.

3.2.2. Receiving information

As it was just explained, the procedure of receiving information is dealt within a process. The sensitivity list of this one includes, as already mentioned, the clock (as it is a synchronous process), one of the enables, the one that triggers the input transmission, and the reset. In order to carry out an action, there has to be a rising edge of the clock when the enable is high, which activates the tasks of reading the data and storing it, or a falling edge of the reset, which initializes the values.

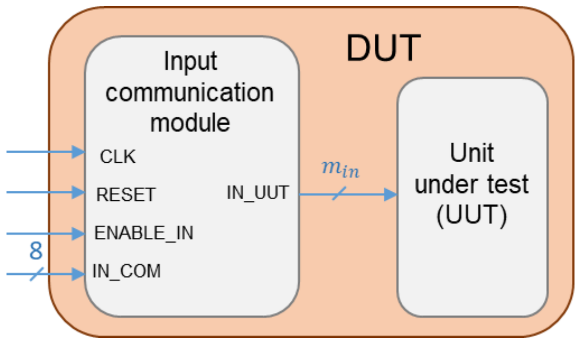


Figure 16. Scheme of the input communication module

It is important to remark that before starting a test, it is necessary to stablish the number of inputs and outputs. One of the reason has already been explained, however, the process of receiving information needs this value in advance as well. By giving the dimensions of the input and output vectors, the code is able to calculate the number of packages expected, and this value simplifies the process of communication.

The process of incoming communication can be summarized in three steps:

1. To start a process of communication, multiple things must happen at the same time. First, the reset of the system must not be connected, since, as it has been explained, this signal is asynchronous and has priority above all the other signals in the process. If the signal is activated, the process will be interrupted and stopped, the data is lost, and the system gets ready to start a new cycle. Second, the enable (enable_in) must be set high, this signal is equivalent to a start button. The last condition is that the clock needs to have a rising edge. When these two last signals are set, then the transfer of information may start.

   Obviously, the duration of the enable in the simulation has to be longer than a clock cycle, or the system can lose the detection of this signal if it is not set exactly when the clock is set to one. This problem is not given when the system is implemented in the boards, since, regardless of the method of setting to one the enable, the duration of this signal is always equal or bigger than one clock cycle. In the simulation, it is the designer the one that chooses the interval of time of the enable, and it can be mistaken, generating a non-realistic situation.

2. Once the communication module is enabled, then the data of the packages that are sent through IN_COM must be saved into another signal, whose size must be the total number of inputs of the test. It is easy to conclude that the most important part of the process is gathered in this step and it is the one that requires to be explained.

   The working principle is based on counting the number of packages that have been already processed with an internal variable of the code. Since the total number of lots can be calculated, it is possible to know whether it is the last package of the communication or not. This condition is the one that allows dividing the VHDL code in two cases.

   The first situation is when there are more lots of data to be received, therefore the package is certainly complete. All the bits contain useful information and are then copied to an auxiliary signal. The other situation is when the last package is being received. This one can be complete or incomplete, and this condition divides the code in other two scenarios. By analyzing the remainder of the division of the total number of inputs by the maximum size of a package, it can be known whether or not the lot is complete. If the remainder is zero, then all the data is useful and stored in the auxiliary signal. If it is different than zero, then not all the bits contain information, and only the correct number of signals must be stored in the signal. The difference between the maximum size of the package and the size of the last one is completed with zeros.

   The conclusion is that this part of the communication has a strong dependence of the number of inputs:

   - Number of inputs is bigger than the width of the channel (settled to eight). In this situation, it is required more than one cycle to send all the data, and so the duration of the process also increases. The time is directly connected to the difference between the number of inputs and width of the channel, since the number of packages required grows. In this case, there is an internal package counter which keeps the module informed of the progress of the communication.

   - Number of inputs is equal or smaller than the width of the channel (settled to eight). This is the easiest situation, since only one package is needed to send all the information and is also the fastest communication.

   In order to finish this step of the module, an auxiliary signal is used. This flag is set to one in order to enable the next part of the code to be executed. The code of this process can be found in the appendix.

3. When the last package has been sent, the code has access to the final condition, since the signal that points the end of the receiving process has been set to one in the past step and the enable value is always low. In this situation, the information that has been stored in the auxiliary signal must be copied to IN_UUT in order to conclude the process.

   Also, it is necessary to set to initial value all the signals that have been used in the code in order to be able to start another communication cycle. The variable that counts the number of packages processed has to be set to one and all the flags used must be set to zero.

After one clock cycle, the information of the signal IN_UUT is deleted in order to assure that the UUT does not take the information more than one time.

The following figure shows an example of how the communication process is expected to be carried out in a timing diagram. In this case, the total number of inputs is thirty, which are divided in four packages. The first three lots are complete and composed by eight signals, but the last one is incomplete, and only the first four bits must be considered and copied to the auxiliary signal. While the process is being carried out, then there is not an input to copy to IN_UUT, and this signal must be set to a certain value. When the process finishes, then the signals are sent to the UUT, and the information is stored for only one clock cycle before it is set to zero again.

The decision of transferring the data to the signal IN_UUT only when the process has finished is in order to avoid interferences. In case the UUT is enable before its timing and takes the

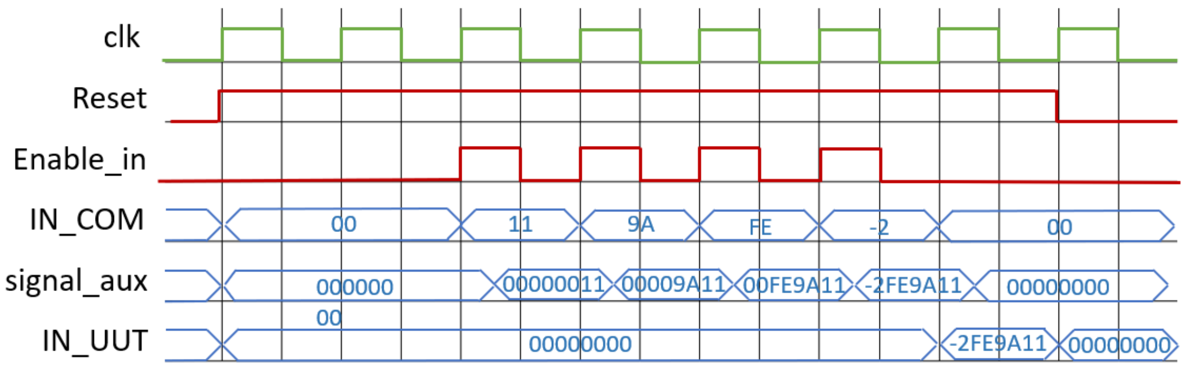


Figure 17. Receiving information: Timing diagram

non-complete data, there could be errors in the test, which must be avoided. In the figure showed below, if every time the auxiliary signal was modify, also IN_UUT was, then this global output would be swinging its value for four times, and the UUT could take any of these intermediate values as the good ones. This is the situation that is avoided by waiting for the process to finish.

Once the module is generated, it is possible to test it to see if the results are the ones expected. The inputs given to the simulation are the ones of the timing diagram, and the result obtained can be seen in the following picture.

Analyzing the results, both signals aux_in and IN_UUT have the expected value, however it should be noted that the data is stored in the auxiliary signal one clock cycle after the enable is set to one. In other words, the task of copying the information into the auxiliary signal is not carried out immediately, as it was expected. This situation does not affect the transmission of data from aux_in to IN_UUT, which is done in the first rising edge of the clock after the last package. The reason is related to the “for” loops that have to be carried out within the process of storing the data in the auxiliary signal, which does not allow the signal to take its value until the next edge.

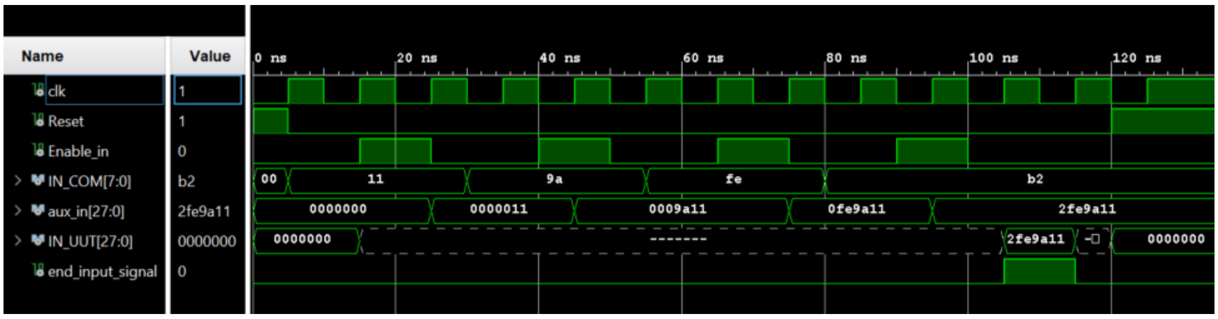


Figure 18. Receiving information: simulation in Vivado

There is another signal to analyze, the one that is called end_in_signal. As its name implies, marks when the process is carrying out its closing condition. In this moment, the auxiliary signal is copied to IN_UUT, and the rest of the signals are being set to their initial value. This signal can be used as an enable of the following module, if necessary, since its duration in time is one clock cycle, which is enough to be detected. Once this process has been executed, both IN_UUT and end_input_signal are set to zero in the default case, which is when none of the other conditions are satisfied.

After analyzing the simulation, the conclusion of this test is that the receiving information module works as expected, and independently of the number of packages, or if they are complete or incomplete, the data is stored correctly and then sent to the UUT through the signal IN_UUT.

3.2.3. Sending information

The process of sending data to the monitoring system presents several similarities to the receiving information one. In both cases the sensitivity list depends on three signals: the clock, that activates the process when its rising edge, the general reset that is asynchronous in this case as well, and the corresponding enable, the one that triggers sending out the information (enable_out). This last enable is sent from the UUT whenever the test is finished.

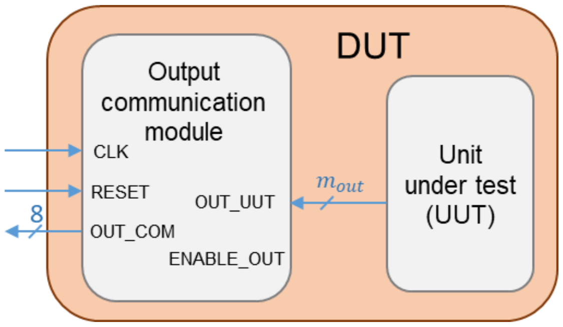


Figure 19. Scheme of the input communication module

As already explained, it is necessary to know ahead of time the number of output signals due to two reasons: first to declare the corresponding generic signal in VHDL for the communication module, and second because it easies the task of splitting the information into packages, as the number of lots is known in advance.

The outcoming communication process can be divided in three stages in order to explain it:

1. The process of sending information can only be carried out whenever the reset is not active, and the process starts whenever two conditions are fulfilled. First, as with the receiving process, the enable (enable_out) must be active, otherwise no task will be executed, and this means that the test has finished, and the resulting signals are available for the communication module. Once this condition is satisfied, then the

process starts when the clock is activated, this is when a rising edge of this signals happens.

As it happened with the input communication module, the enable of this process must be long enough to assure that it is active when there is a rising edge of the clock. This does not present an issue in the physical implementation in the board; however, it must be taken into account when the designer is creating the stimulus for the test.

2. Whenever the process is triggered and the communication is enabled, the code oversees splitting the total number of outputs from the test into packages with a maximum size of eight bits. This information is then sent through the bus OUT_COM to the monitoring system, and whenever the last package is sent, the process in then finished.

   The first obstacle found in this module is that the enable that triggers this process usually lasts one clock cycle, which is not enough to send all the packages, not even in the case of only having one. The way of solving this issue is to create a signal that can be a flag to this enable. Whenever the process is triggered for the first time, this flag is set to one, and as long this is its value, the process will be active, always depending on the value of the clock.

   As it can be noticed, the main difference between the two communication processes lies in the moment when the information is transferred to the global signal, that is, when it is sent to the next module. In the receiving information process, the data is sent to the next module once all the packages have been gathered in an auxiliary signal. But in the output communication process, this action takes place every time a package is created, since the bus connecting this module to the monitoring system is the bottle neck that slows down the procedure.

   The working principle of sending the data is similar to the receiving one. It is based on counting the number of packages that have already been created and delivered, and this is done through a variable in the process. In this case, this signal is even more important than on the other process, since it is the one that allows keeping track of the signals that have already been sent and the ones that have to be transferred next, in other words, it works as a pointer that tags the signal OUT_UUT.

   The other reason for using this variable is that, since the total number of outputs is known, then it is possible to calculate the amount of packages to be sent and control the situation depending on it. The difference between the two processes lies in the way of dividing the scenarios and depends on the number of signals in the package to be created.

   The first situation is when the package that has to be generated is complete, and this condition is always fulfilled if it is not the last one of a transmission. However, it is also true when this last package is complete, in other words, when the remainder of the division of the total number of outputs by the maximum size of a package is equal to zero. In this situation, and with the help of the variable that points the next signals to be sent, this eight bits are transferred to the monitoring system.

   The second scenario is when the last package to be created is incomplete. In this case, only the rest of the signals have to be copies to OUT_COM and the rest of the eight bits that have no value must be set to zero. It is important to keep controlling the value of those signals, even though nor the DUT nor the monitoring system will take them into account.

As the opposite procedure, the process of transmitting the information from the UUT to the monitoring system can be divided in two scenarios depending on the number of outputs:

- Number of outputs is equal or lower than the width of the channel. In this case, only one package must be sent to the monitoring system, and so only one cycle is required to fulfill the transmission of information. This package may or may not be completed with eight signals, but this doesn't affect the duration of the process.
- Number of outputs is bigger than the width of the channel. Due to the number of signals, the communication cannot be done in just one cycle and more packages are required. The time needed by this process is increased as the number of lots of information does. At this point, the benefit of knowing beforehand the number of outputs does not result in a decrease of time, as all the data is saved in the communication module and from the beginning the number of packages is known.

Once all the packages have been sent, this step is executed one more time to set the enable flag, already explained, to zero. This way this step is finished and will not take place until the next communication cycle. This moment is also used to set high the auxiliary signal that triggers the last stage of the module.

3. This last step is executed when the communication has finished. This stage is used to set all the signals and variables to their initial value so that the system is prepared for the next communication cycle. Also, as it is done with the receiving process, when the last package is sent to the monitoring system, the process shall wait one clock cycle and afterwards the signal OUT_COM must be deleted, this is set it to zero in order to avoid any errors.

On the following image it is shown an example of how the output communication process is expected to work in a timing diagram. In this case, the number of outputs is twenty-eight, and for this reason, four packages are needed in order to send all the information. The first three lots are complete, this is, they have eight signals, but the fourth one is incomplete, and only half of it has useful information. The rest of the signals of this package are not considered and are set to 'don't care'.

It can also be seen that every time a package is created, it is directly sent to OUT_COM, which is the opposite behavior to the other communication process, in which all the information is gathered and send at once.

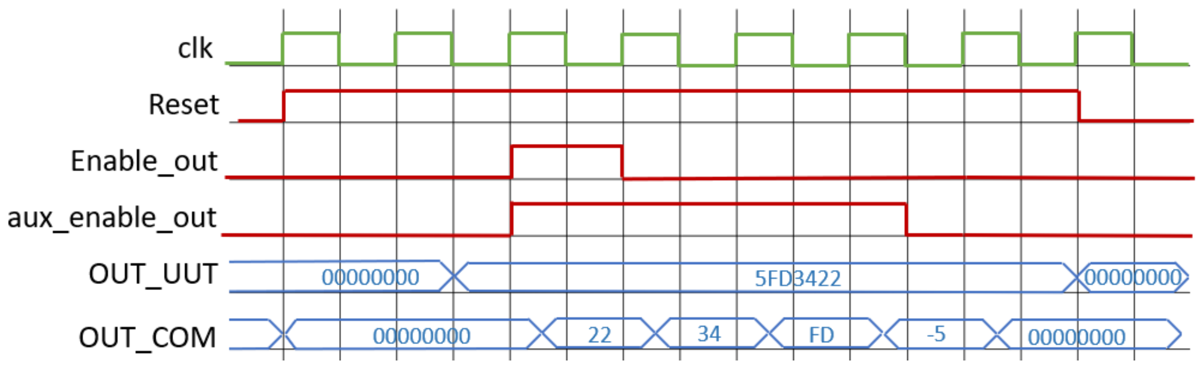


Figure 20. Sending information: timing diagram

In this process, in order to avoid errors, the data that comes from the UUT is transferred to an internal auxiliary signal. This assures that, even if OUT_UUT changes, the communication module will still have the correct information. It is from this auxiliary signal the one that copies it to OUT_COM. Also, when all the information has been trasnfered to the monitoring system in packages, the signal OUT_COM is set to zero in order to avoid getting the last package two times.

As it was explained, the enable signal is only active for one clock cycle, and as it can be seen it is not long enough to trasnmit all the outputs to the moitoring system. The solution adopted is to set another auxiliary signal which can be active as long as the process needs. Whenever the commuication ends, this signal is set to zero and the process ends.

When the module is already prepared, it is necessary to test it in order to see if the behavior is the one desired. The outputs of the test, which are the data to be divided in packages and sent, are the same as the ones of the timing diagram just shown. The results can be seen in the following image.

By analyzing the results at first sight, the conclusion is that the functioning of the module is the one expected, as the information is divided in the desired packages and sent to the monitoring system in the right order. But examining the result more deeply, it can be seen that the timing problem that was present at the receiving test appears in this one as well. First, the auxiary enable is set a clock cycle later than expected, however this does not supose a problem for the functioning of the system.

This situation is the same with the signal OUT_COM, but the delay this time is of two cycles. The reason of having one more cycle is because of the use of an internal auxiliary signal in which the packages are copied, and then, from this one, the data is tranferres to OUT_COM.

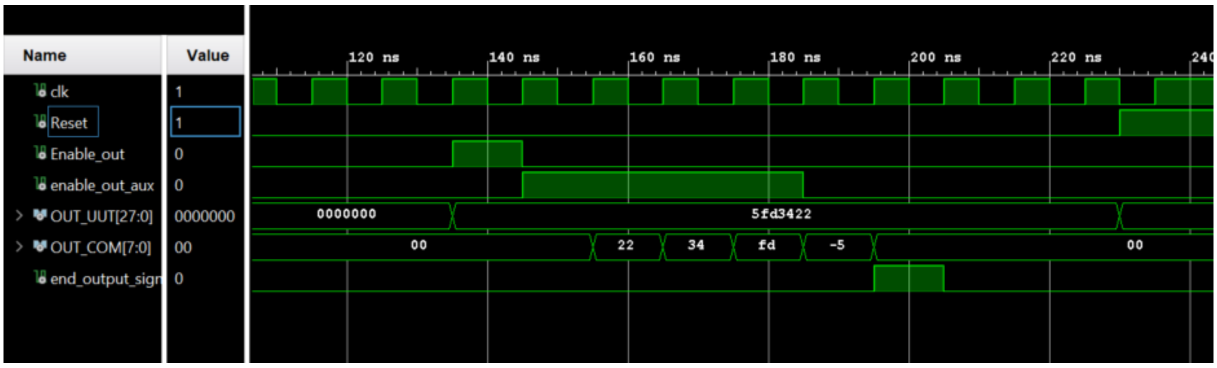


Figure 21. Sending information: simulation in Vivado

The idea of using this intermediary signal is to avoid errors, such as copying twice the information in the output.

The last signal of the test is the one that points out that the process has reached its end. The duration of this variable is of one clock cicle, which is enough to trigger any other process if necessary. At the same time as this signal is set, the data contained in OUT_COM is deleted, resetting the system for the next communication.

3.2.4. Communication test

After the development of the two processes required for the communication, receiving and sending information, it is necessary to test them together in order to assure their correct behavior. It has been proved that they work as expected when they are separated modules. It is necessary to simulate the behavior of the complete DUT board, and for this purpose, it is necessary to introduce a unit under test in between the two processes.

3.2.5. Unit under test

The UUT is another module that has to be programmed in VHDL and introduced to the block design. It has been chosen a shift register as the test implemented, and the reason of this decision of circuit is due to its simplicity. First, its working principle is easy. The idea is as simple as taking the inputs given in IN_UUT and copying them in OUT_UUT moving the bits one position to the left. Having a simple UUT assures that the resources of the FPGA will not be used for this module. Second, its simplicity is also translated in an easy implementation, which requires easy code.

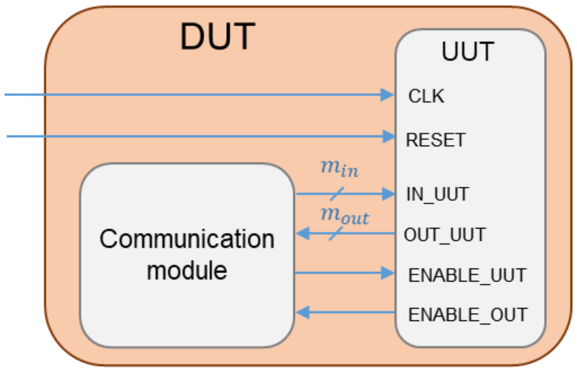


Figure 22. Scheme of the unit under test structure

As the other two modules, this one is composed of sequential actions and therefore the tasks are run in a process. The sensitivity list is composed of three signals: the reset, the clock and an enable signal (enable_UUT). The first two are shared with the communication module. The last one, the enable, is sent by the receiving information process. This signal is activated when all the data has been gathered and is ready to be sent to the test. As it was seen in the chapter of the document referred to the receiving information process, the signal that could be used as this enable is the flag that points the end of this process.

The unit under test process can be divided in the following three steps:

1. First, in order to activate the module, three conditions must be satisfied at the same time. As with the other modules, the reset has to be disabled. Second, the enable must be active, this is, the receiving process has to have finished, since this is the moment in which the ending signal is active. This signal is the one that is also used as the UUT enable. The last condition is to have a rising edge of the clock, as the process is synchronous.

   Since the simulation that is going to be carried out is of the complete DUT, there is no problem with the duration of the enable signal. The reason is that it is assured that it will last, at least, one clock cycle. This duration is enough to be noticed by the system and the process can start.

2. When the three conditions are satisfied, then the test can begin. As it has been explained, the idea is as simple as shifting all the data one position to the left. In the other two processes, the number of inputs or outputs was necessary in order to calculate the amount of packages required. For this one it is necessary to have these two numbers because the shifting process depends on them.

There are two different scenarios depending on the difference between the number of inputs and outputs:

- Number of outputs is bigger than number of inputs. In this case, there are not enough bits to fill the output signal, and the left part of the vector which has no data is set to zero. All the input data is copied to OUT_UUT but shifting their position one to the left. The bit zero of the output is not filled with any input information, since the bit zero of the input is located in the position one of the output and is also set to zero.
- Number of outputs is equal or smaller than the number of inputs. In this situation, some of the input bits cannot be stored in the output and are lost. All the output vector is filled with useful data except to the right position, which is also set to zero, as explained in the other case.

As it has been done in the other module, the data that has to be shifted is copied into an auxiliary signal, in order to avoid overlapping of the information. This procedure is also done with the output signal. The shifted data is stored in another auxiliary variable, and it is only copied to the output one when the process has finished.

3. When the data has been shifted and copied into the auxiliary output signal, the process has reached its end. This is pointed out by an ending signal which triggers the finishing condition of the process. In this branch of the code, the output of the test is copied to OUT_UUT and the enable used to trigger the sending process to the monitoring system is set high.

After one clock cycle, and since no other condition is satisfied, the process runs the default one, which is in charge of resetting all of the signals and variables, including the enable of the process of sending communication. This is the reason why the signal only lasts one clock cycle, and an auxiliary one was necessary. Also, as done it before, the signal OUT_COM is set to default value, in order to avoid sending the last package more than once to the monitoring system.

In the following image, it can be seen a timing diagram with the expected behavior of the shift register. The input to the test is a signal of twenty-eight bits and the output has the same number of signals. When the enable is high, the module is expected to be able to transfer all the data in one clock cycle. However, as it has been said, in order to avoid errors or overlapping the information, auxiliary signals have been used, and transferring the data to these ones requires clock cycles.

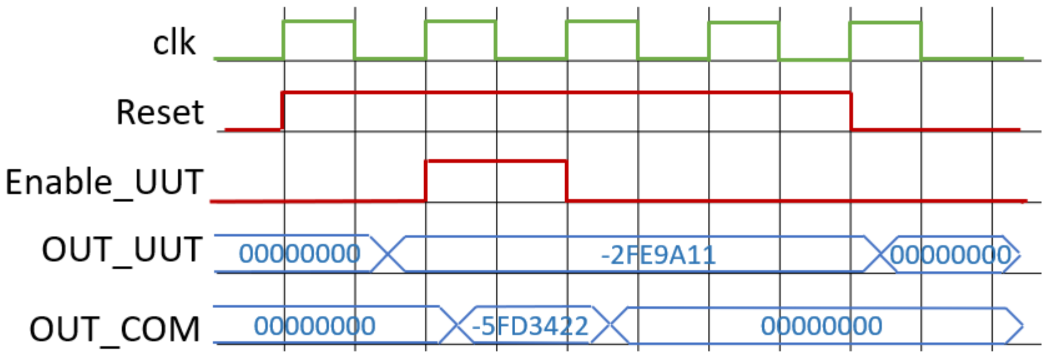


Figure 23. UUT: timing diagram

This becomes an issue, since the duration of the enable of the process is not long enough to copy the data into the auxiliary signals and vice versa. To solve this problem, it has been used an auxiliary enable, as it has been done in the sending information process as well. Whenever the process is triggered, this auxiliary enable is set to one and allows the system to have more clock cycles. When the test is finished, then it is set back to zero.

In order to test if this solution is the correct one, it is necessary to simulate the module of the UUT before adding it to the complete system. The inputs feed to this module are the same ones of the timing diagram.

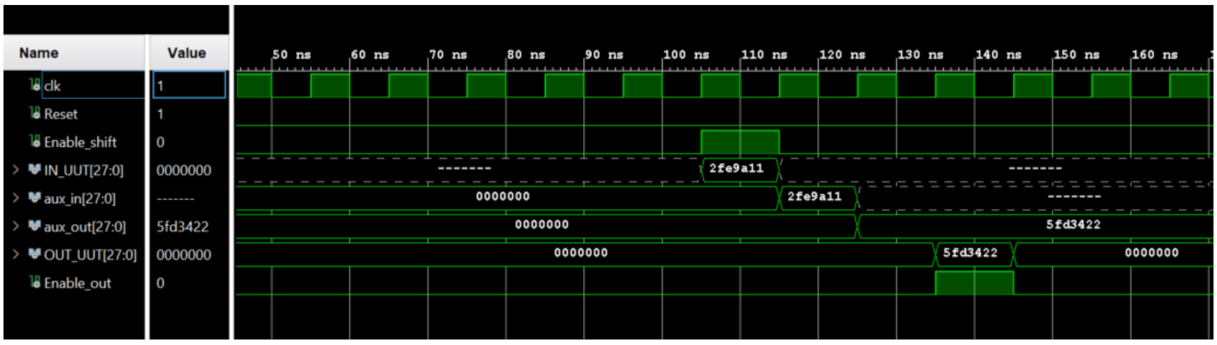


Figure 24. UUT: simulation in Vivado

As it can be seen in the result of the simulation, shown in the past figure, the result of the shift register is the one expected, and it can be concluded that the operation itself of the test is correct. Also, as it was anticipated, the process of the test is not carried out in just one clock cycle, but the use of the auxiliary enable allows the module to be activated as long as it is necessary to carry out all the operations. This widening of the duration of the process is due to the use of auxiliary signals. As it can be seen, the first clock cycle is used to store the data in the auxiliary input, the next one is used to carry out the test and store it in the internal output, and in the last one it is finally transferred to the signal OUT_UUT. Also, in this last clock cycle, the enable of the following process is activated.

When the last clock cycle is finished, the enable and the output signal are set to zero, however the auxiliary signals are not reset. The reason is that, since this process has to be enabled to be carried out, there is no danger of sending the wrong information to the communication module, in other words, there is no danger of swinging the output signal. Also, even if the internal signals are modified without the process being enabled, the auxiliary flag that points the ending of the process has to be set in order to copy the information to OUT_UUT. The conclusion is that the danger that supposes the changing of value of these two internal signals to the output signal is negligible and for this reason they are not set to zero.

3.2.6. Testing results

After having the UUT prepared and tested, it is time to add all the modules created to a block design and connect them together so that the information can go from one to another. The idea of this part is to only simulate its behavior, and for this reason it is not necessary to add the processing system IP, since the clock signal is generated as a stimulus. However, in order to design the complete DUT block design, it has been added and connected as well. In the following image, the block design of the board is shown.

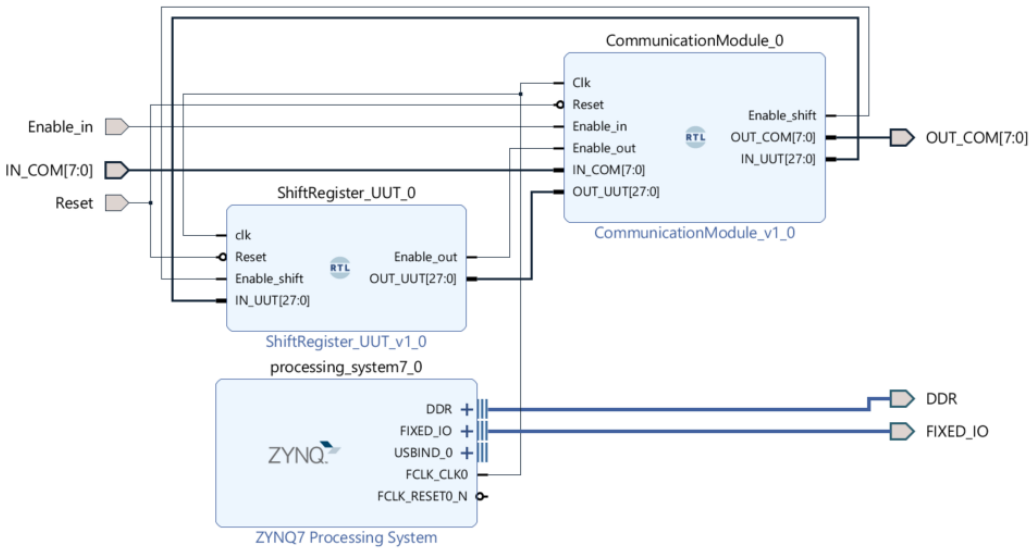


Figure 25. DUT: signal connections

In order to generate a correct simulation source, first it is necessary to declare the two components that form the block design, the shift register and the communication module, and the inputs and outputs that are form of. After this, the signals that link one module to the other have to be connected, and this is done by declaring internal signals of the simulation source and linking them to the signals of both components. Also, this is the moment to give a value to the generics created, that are the number of inputs and outputs of the system. By modifying this value in the simulation source and on the VHDL modules, the test can be done for different situations in which the number of packages differ from one to another.

Once all the modules are declared, and the signals are connected, then it is time to create the stimulus of the test, this is, the signals that may be modify by the designer in order to cause reactions in the system. In this case, the inputs of the simulation are the reset, the enable of the receiving information process (Enable_in), and the input data. At this point, it is important to be aware of the timing of the stimulus. In other words, if the enable is set for too long or for not enough time, then the test may fail even if the system is correctly designed. The same happens with the input data, and the situation is more critical when several packages have to be sent, since the duration of these packages have to be long enough to assure the module can store them, but not enough to allow it to take this value more than one time.

When the code has been developed, it is time to test the system. In order to prove the correct functioning of the DUT, independently of the situation, several testbenches have been prepared and the only purpose of these are to check the correct functioning of the communication. By modifying the values of the reset, enables and inputs, the outputs are analyzed and compared to the expected result. But not only the results are checked, as it is also interesting to analyze the intermediate states of the system and its timing performance.

As already explained, the system depends on two variables: number of inputs and number of outputs., as these two determine the number of packages, and therefore, the complexity and duration of the communication. The tests have been classified depending on the number of inputs, and are divided in three: eight, twelve and sixteen inputs. This allows to examine different situations, first, when there is available just one package that is complete, second, when there are two lots. This last scenario is divided in two, as in one the last package is incomplete and in the other it is not.

Each case given by a number of inputs is then divided in other three, that differ from one to another depending on the number of outputs. The situations chosen are the same as the ones explained, eight twelve and sixteen, and the reasons are also the same.

In the following table, it is shown the two possible input packages given to the DUT, depending on the scenario. This data is expressed in binary, since it is easy to do the shifting of the data with this format, but also in hexadecimal, since it is the way in which the simulation will show the information. It should be noted that in the case of having twelve bits as input to the system, the second package is fed as it was complete, and the reason is to test if the system will only take the first four bits, dismissing the others.

The rest of the table shows the expected outputs for each scenario, both in hexadecimal and binary as well.

INPUT SIGNALS				OUTPUT SIGNALS					
Number	Package	Data		8		12		16	
		Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
8	1	0x11	0001 0001	0X22	0010 0010	0X22	0010 0010	0X22	0010 0010
						0X-0	---- 0000	0X00	0000 0000
12	1	0x11	0001 0001	0X22	0010 0010	0X22	0010 0010	0X22	0010 0010
	2	0x4F	0100 1111			0x-E	---- 1110	0x1E	0001 1110
16	1	0x11	0001 0001	0X22	0010 0010	0X22	0010 0010	0X22	0010 0010
	2	0x4F	0100 1111			0x-E	---- 1110	0x9E	1001 1110

Table 1. Test results: inputs and expected outputs

- Number of inputs=8

In this first situation, all the information is sent in one package, which is complete. As it was explained, from the three cases studied, this is the only one in which the sending information process is the shortest it can be, as transferring the information only takes one cycle.

In the UUT there are two different situations. When the number of outputs is bigger that the inputs, not only all the information is useful and shifted, but it is necessary to add zeros to the left part of the vector in order to fill it. In the case in which the number of inputs and outputs is the same, only the last bit of the input is lost. For the first package, it is a zero what is dismissed.

Once the test has been carried out, it is time for the information to be divided in packages and sent to the monitoring system. Depending on the number of outputs chosen, there are one or two packages available.

It is easy to conclude that the shortest cycle of DUT (which is the time taken to receive information, carry out the test and send back the data to the monitoring system) is when the number of inputs and outputs is eight. The worst cases are when both inputs and outputs are over eight, as there are two packages in both communications.

Following this, it can be seen the result of these three simulations. The signals shown are, in order, number of inputs, number of outputs, maximum size of a package, clock, reset, enable_in, which is for the receiving communication module, enable_shift, that activates the UUT process, enable_out, used for the sending communication module, IN_COM, sends the data from the monitoring system, IN_UUT, stores all the data, OUT_UUT, stores the results of the shift register and OUT_COM, sends the packages back to the monitoring system.

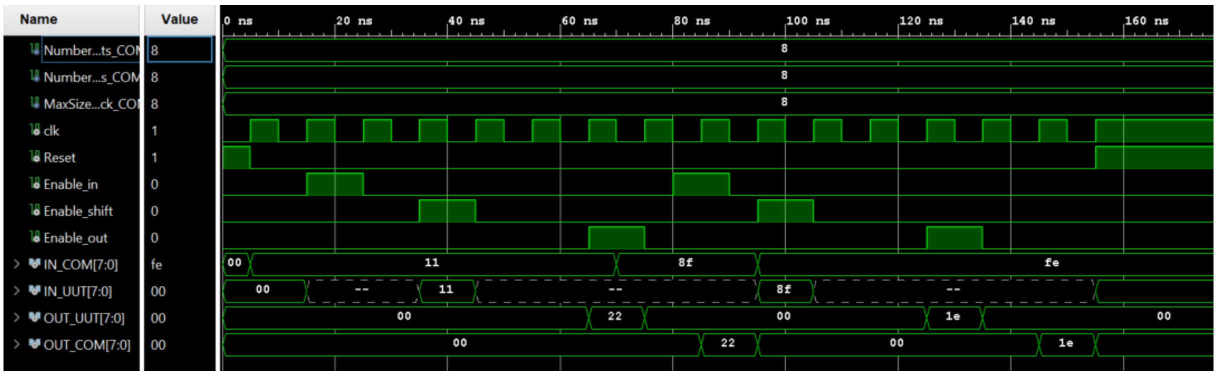


Figure 28. Simulation of communication: input=8, output=8

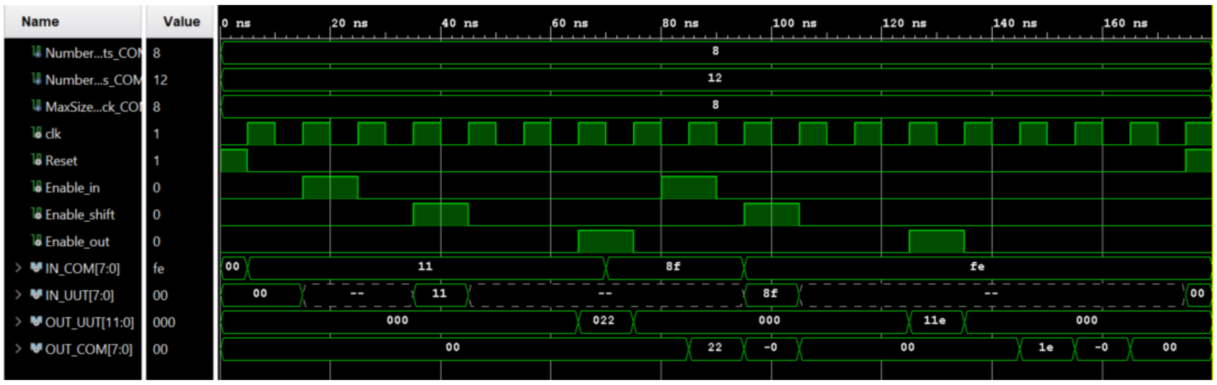


Figure 27. Simulation of communication: input=8, output=12

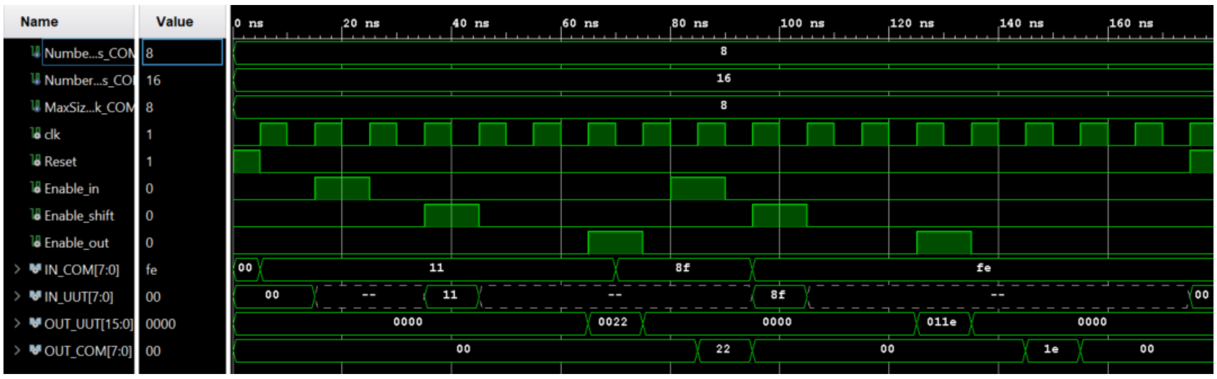


Figure 26. Simulation of communication: input=8, output=16

The first conclusion of the simulations is that the system works as expected, since the output is the one calculated. Also, the behavior of the system is the one desired, the input is fed through IN_DUT whenever the communication is enabled and then sent to the shift register (test module) through IN_UUT, taking as many clock cycles as seen in the individual tests. The output of the test is showed in OUT_UUT and after this, is sent to the monitoring system in packages using OUT_DUT signal.

Taking advantage of the speed of these three scenarios, in which there is only one input package, it has been decided to verify whether or not the system is able to carry out more than one DUT cycle. For this reason, once the first input has been tested and sent back, there is a second cycle with new information fed as input. Independently of this situation, the system is able to calculate a new output and send it, without mixing or overlapping the data.

It is important to point that on the simulation of twelve outputs, it can be seen that two packages are send to the monitoring system, with the expected values. However, when the number of outputs is sixteen, it is only shown one package. This is due to the value of the second one, which is 0x00. Since this value is the same as the default one, there is no change and the simulation shows no separation between the two cases. But, since the case of twelve outputs works correctly, it can be supposed that this last one works as well when the value is different.

- Number of inputs=12

In this case, the number of inputs is bigger than the maximum signals than can be transmitted in a package, and so two lots are necessary. The first one is complete, however the second one is filled with only four useful bits. Despite this, it has been chosen to feed as input of the second package a complete one in order to test whether or not the system is able to disregard the non-desired information.

Since this situation requires more time, as both input and output communication modules have to send two packages in some scenarios, only one DUT cycle has been simulated, in other words, each process is activated only once.

The signals that are tested in the simulation are the same ones already explained. The stimulus controls the clock, reset, receiving communication enable and the data that is fed for the test. The expected results are shown in Table 1.

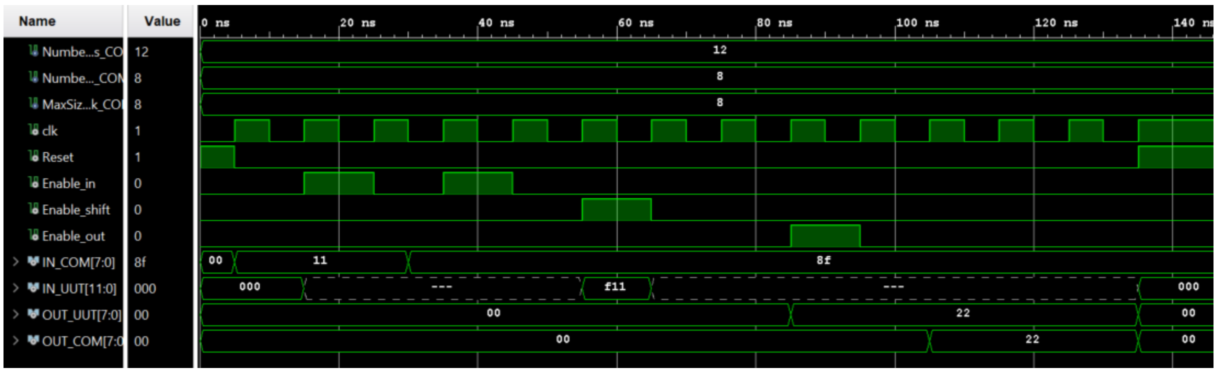


Figure 29. Simulation of communication: input=12, output=8

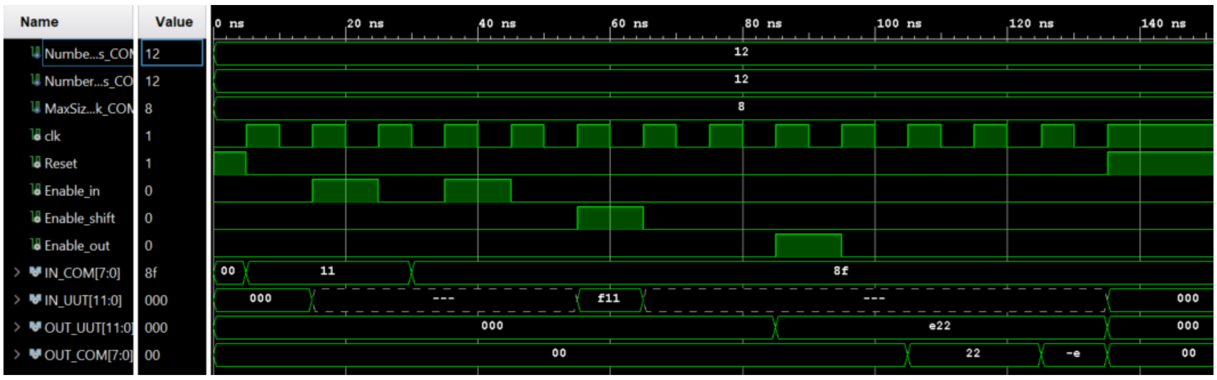


Figure 30. Simulation of communication: input=12, output=12

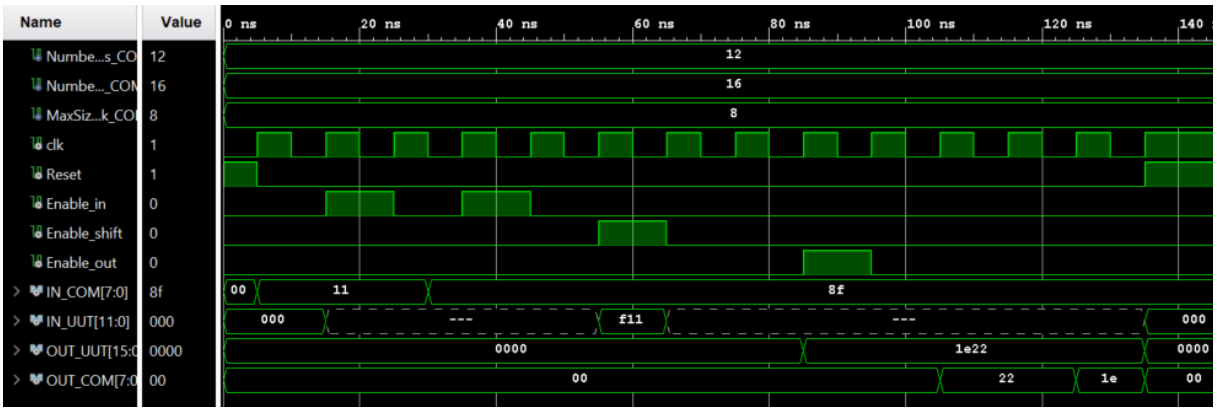


Figure 31. Simulation of communication: input=12, output=16

By analyzing the past images, it can be concluded that the system works as expected when the number of inputs is twelve. The outputs of the tests are in the three cases the ones calculated with anticipation. Also, the behavior of the system is the one desired. Even though the data fed for the second package is composed of sixteen bits, the receiving information process only takes the first four signals, dismissing the other ones, leaving IN_UUT with a value of 0x-F11.

It must be noted that, in this situation, the stimulus produced by the simulation has to set the Enable_in signal two times in order to allow the system to take the two-input data necessary for the packages.

The other thing to take into consideration, is that, for the first time, it has been simulated the worst case in terms of timing from the scenarios studied. When the number of outputs is greater than the maximum size of a package, then both the receiving and the sending communication processes have to manage two lots with information. Comparing the performance of the best case, which is eight inputs and outputs, and one of the worst, it can be seen that the difference of duration of the tests are, more or less, forty nanoseconds. Although this number may sound small, it can be seen that it is no if it is compared with the duration of the best-case scenario, that is seventy-five nanoseconds. This means that the worst case increases its duration with respect to the best one by a 50%.

There is an intermediate situation in which only the input has to send two packages, whereas the output only needs one lot. The time required for this DUT cycle is one hundred nanoseconds, which is obviously in between the other two cases. As a curiosity, this situation is closer the worst case in terms of duration.

- Number of inputs=16

In this last case, the number of inputs also exceeds the size of the package, and although two packages are also required, in this case both of them contain useful information. This means that to the UUT arrives an amount of information that in any case can be transferred completely to the output. The reason is that the number of outputs is always smaller or at least equal to the number of inputs. As it was explained, some bits are lost because there is no space for them, in the case of being different the number of inputs and outputs, or because when they are shifted they get lost, which is the case of the bit on the left when the number of inputs and outputs are equal.

As it was done in the last tests, the simulation is done for only one DUT cycle, since two of the three scenarios are the worst case on timing performance, and this allows a clearer image of the simulation.

As it happened in the other cases, the signals that are controlled are the reset, clock, the Enable_in, that has to be activated twice to allow the system to take two different data for the packages, and the data itself, IN_UUT.

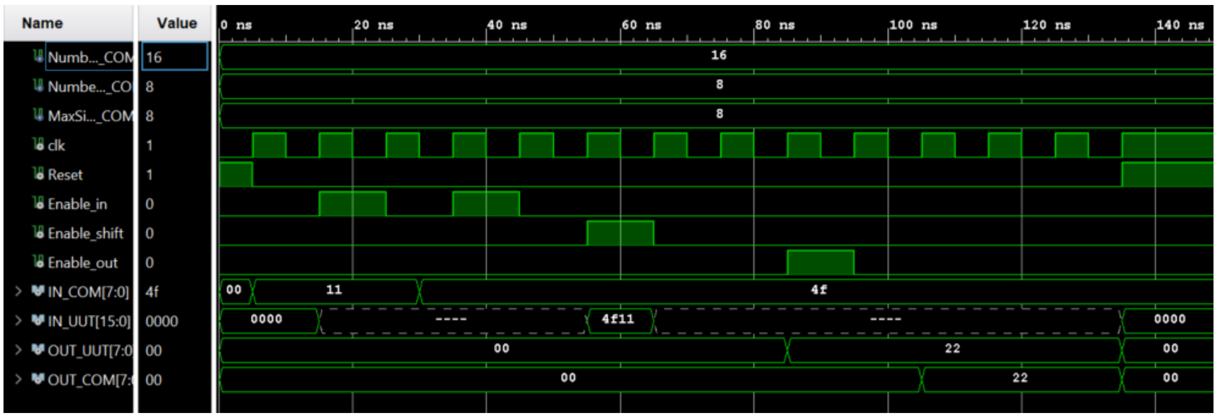


Figure 32. Simulation of communication: Input=16, Output=8

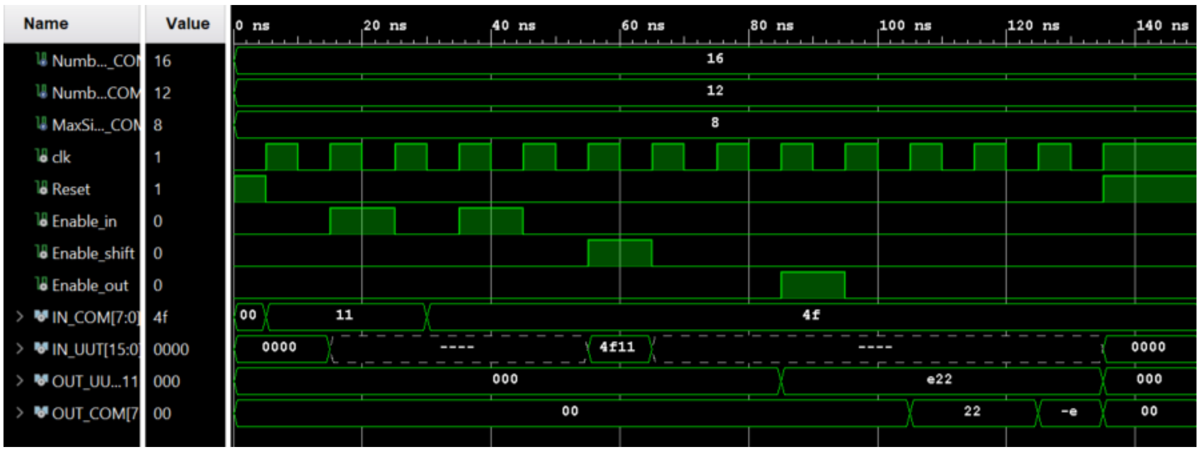


Figure 33. Simulation of communication: Input=16, Output=12

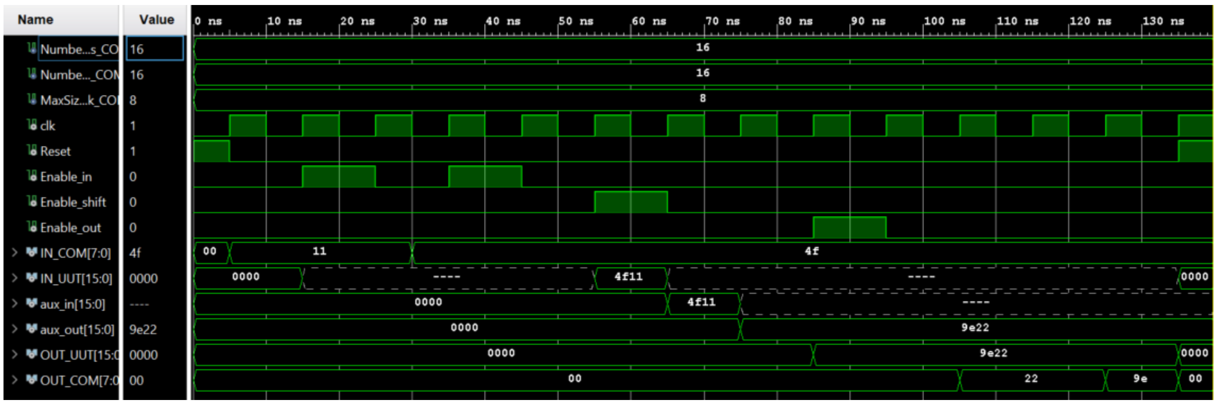


Figure 34. Simulation of communication: Input=16, Output=16

By analyzing the results obtained, the first conclusion is that the system behaves correctly, as once more, the results are the ones predicted. Independently of the number of packages used for the transmission of the input and the output data, the information is correctly gathered or divided, and the UUT is able to carry the shift register test.

With the scenario of sixteen inputs and outputs, it is possible to analyze the last situation, which is that a complete second package arrives to the DUT module, is shifted and then sent back to the monitoring system. It can be seen that the second output lot is sent with all the information, which means that the supposition made at the test with eight inputs and sixteen outputs was correct, and the second package works correctly.

With respect of the timing performance, no differences are noted between the cases examined in the last scenario, and the time required for all the cases is, more or less, the same.

With this last set of tests, it is possible to conclude that the communication module works for every case than can be presented, as it has been proved that can send and receive either one or more packages, giving the correct results.

3.3. Communication implementation

Once the design of the communication board has been done, it is the moment to do the implementation for both boards. As it has been already explained, the two FPGAs of the project are necessarily connected, and it is done through the jumpers that link the correspondent pins and through the communication modules that were explained in the last section, which are implemented in VHDL. However, this units were designed for a specific scenario and for simulation, where the inputs were fed manually with the stimulus, assuring that the timing was the right one. But in this part of the project, it is the monitor system the one that has to send the right signals in the corresponding moment in order to have a successful communication. It can be said that the modules were designed only taken into account the DUT board and acting like the monitor was not there. This is the reason why the communication modules must be adapted to this new situation.

The aspects that were not taken into account when designing the basic modules were the following ones. First, as it has been said, the input data was fed through the simulation source, adapting its timing to assure that the receiving process of the DUT was ready to take more data. This has been done by controlling the enable of this process, which was not set until the system had processed the previous package. But this time cannot be known easily beforehand, as it depends on the number of inputs and outputs of the test. Also, the data was stored in IN_UUT some time before the enable was set in order to assure the DUT would take the correct information.

In this new situation the monitor system is the one in charge of feeding the data and triggering the process of receiving it in the DUT, and this condition presents some problems. First, it has to take the correct information from the memory and send it in a package at the correct timing, this is whenever the DUT is prepared to receive it. In case there are more than one lot, the monitor must wait as much time as the DUT requires to process and store the data, before sending the next package. As it was said, this time is not exact and can variate from one case to another, so it cannot be a solution to make the process wait a certain amount of time.

This situation is similar in the other communication process. When this process receives the information from the unit under test, it directly starts sending packages to the monitor system. This behavior presents an issue since this last board may not be ready for the package and the reason is that it may not be expecting it. Also, in case there are more than one lot to be sent, the DUT does it one after the other, and the monitor may not have time to take the information, process it and store it. This all leads to a situation is which information may be lost or misplaced.

It is easy to conclude that it is necessary to add some other module in the monitor board in order to get more control of the communication process, and to modify the existing one in the DUT.

3.3.1. Communication protocol

The main problem in the communication system can be identified by analyzing the issues of the processes of sending and receiving information just explained. All can be summed up in that the two boards are out of synchronization, in other words, whenever one is sending information, the other one is not waiting for it nor requesting it.

The solution to this situation is to introduce a communication protocol that can assure this necessary synchronization. There are plenty of available protocols with different levels of complexity, but for this project it is needed a simple one which doesn’t require a lot of resources nor time. For all the given reason, the protocol chosen for the communication process is a variation of the one known as handshaking.

The handshaking is an automated process that sets parameter for communication between two different devices before normal communication begins. This computer handshake provides both devices with the basic rules for the way data is to be shared between them. (17) This process of setting the rules begins every time there is a new connection between two components, as the computer with a printer. Extrapolating this example to the situation of the project, every time that one of the boards wants to communicate with the other, it is necessary to do a “handshake” or to stablish the channel of communication.

The difference between the original protocol and the one that should be implemented in the project is that the first is used to set rules as the transfer rate or the coding alphabet used. In our case, the protocol used is simpler, and it only has to work as an alarm that wakes up the other side of the communication, which is sleeping until it is told that the information is available.

This handshaking protocol has to be implemented in the two direction of the communication. But, although the idea for both is the same, each process requires different adaptation and signals. For this reason, these two cases are going to be explained separately in the two following sections.

3.3.2. Handshaking protocol: from Monitor to DUT

This process is the one in charge of taking the input data from the memory and sending it to the DUT, this process has to be repeated as many times as packages are needed. In the first design, explained in the last chapter, the process was triggered by its own enable, which was controlled as a stimulus in the simulations.

In this new situation, the enable has to be controlled by the monitoring system and has to be in charge of starting the process of receiving information in the DUT every time that new data is available. This procedure becomes more difficult as the number of packages increases, since the second board may or may not be prepared when the information is available and sent. So, it is necessary to introduce some way in which the DUT can show its state.

The way of introducing the handshake protocol in this process is by using two new signals that are used as indicators of the state of the two boards.

The first one is an enable which is used as the one in the original module. The idea is that this signal is the alarm that wakes up the device under test, warning it that new data is coming its way and it must be taken and stored. This signal is an output of the monitoring system and an input to the DUT.

The other signal is the one used to inform of the state of the DUT. It is called ‘Ready’ signal, as it indicates when this board has finished taking the information from the channel and is ready for new information, if it exists.

As it was done with the communication module designed before, the working principle of the information transmission is based on counting the number of packages that have already been sent by the monitoring system, and comparing this number with the total amount necessary to transfer all the data for a test. However, in this case there are two modules that are implemented one in the monitor and the other in the DUT. The way of implementing the handshaking protocol with the new signals is divided in three steps, that are shown in the following figure.

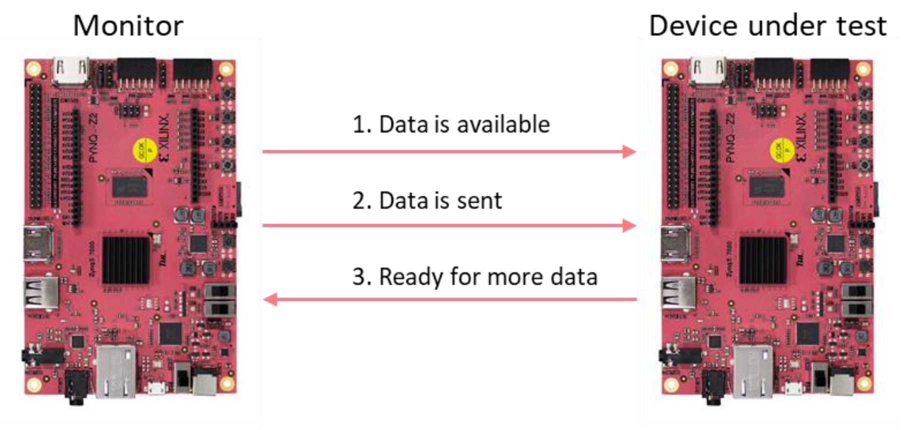


Figure 35. Receiving information handshake protocol

The first cycle is developed as follows. When a new cycle is started, independently on the way of doing it, the monitor loads the data in the corresponding signal that connects the two boards. Once this is done, the system waits one clock cycle in order to assure that the information has been uploaded. After this, the enable signal is set to one, warning the DUT that new information is available. This only happens if the ready signal is not set to one, and it is done to assure that the system has had time to set it back to zero.

Having the enable activated triggers the process that is located in the module of the DUT, which is in charge of getting the package and storing it in a signal. As it was already explained, the VHDL code differentiates the situation depending on how full the lot is. As long as the enable is set to one, the system is taking the package as a new one and storing it, this is the reason why it has been used a signal as a flag to assure that this condition is executed only once every time the enable is set to one. While this is happening in the DUT, in the monitoring system the process is set to a default case in which the enable is set back to zero, however in case this takes longer than one clock cycle it is necessary to use the flag already explained.

Once the data has been taken and stored, the next condition of the DUT process executed depends on the amount of packages left to be received. If this lot of information is the last one, then the end flag is set to one and the finishing condition is carried out. This one is in charge of enabling the next module, which is the test, and resetting all the system for the next cycle of communication, in other words, setting to the initial value the variable that counts the number of packages and all the flags used. After this, the system goes to the default case, which sets this enable back to zero.

If there are more packages left, then the branch executed is the one that sets the signal ready to one in order to inform that the system is ready to receive more information. This is only executed if the enable is back to zero. After this, the system goes to the default case in which the ready is set back to zero. At this point, the monitor system is ready to start again, which means that has to load new information into the corresponding signal and send it to the DUT following the same procedure until the last package is sent.

In order to show what has been explained, which is the functioning of the communication module, in a graphic way, it has been created the following timing diagram with the most relevant signals of the process. In this case, the number of inputs is sixteen, which means that two packages are sent from the monitoring system to the DUT.

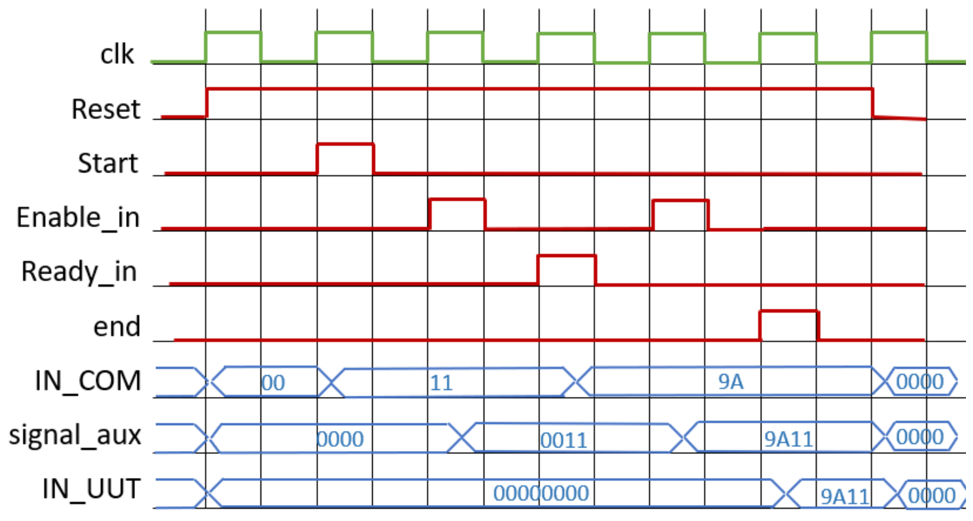


Figure 36. Receiving information handshaking protocol: timing diagram

The monitoring system begins the process of transmitting information when the signal ‘Start’, which is considered in a general way since it has not been decided yet how it works, is set to one. The first thing done is taking the data from the memory and transferring it to IN_COM, which is the signal shared by the monitor and the DUT. As it was explained, the system waits one clock cycle before setting the enable, in order to assure that the data is copied before the DUT takes it. Once the enable is high, the information is copied into the auxiliary signal of the process, where all the inputs are stored until the ending signal is set.

Since this case is set for two packages, the enable has to be set to one two times, one for each lot. However, the ready signal is set only once, as only one more package is missing from the monitor system.

It can be seen that the information is only transferred to the test when the end signal is high. This signal can also be used as the enable of the following process, the test, as it is active when the communication is over and only lasts for one clock cycle.

As it can be seen in the diagram, the timing is really important for the handshaking communication, and it is the reason why all the flags are used. If the signal enable is set before the ready_in is back to zero or vice versa, the code could get caught in a loop which would make impossible the communication, or it could generate a situation in which one of the system misses the signal of the protocol.

For this reason, it is important to make both signals as short in time as possible, this is, restricting them to a clock cycle if possible, which will avoid the overlapping of the enable and ready.

3.3.3. Handshaking protocol: from DUT to monitor

This part of the communication module is in charge of taking the outputs produced by the test and dividing this information in as many packages as required. After this, the process sends them one by one to the monitoring system. This module was designed to be triggered by a signal produced by the test, which works as an enable of the process. This part of the design is works as desired and does not have to be modified. However, the problem of this procedure comes to the part in which the monitor system has no way of knowing that the packages are being sent, and so it is not prepared to receive information.

In order to solve this issue, it is necessary to introduce a new signal, data_available, which will warn the monitor system that the packages are going to be sent. In other words, this signal is in charge of waking up the process of taking the information to store it in the correspondent part of the memory.

This signal is enough in case that all the information can be sent in one package, but as the number of lots increases, so does the complexity of the communication. The reason is that the monitor system has to have enough time in between transmission to take the data, store it and prepare for the next one. If the DUT is capable of sending the packages faster than the monitor is able to process them, then some information may be lost. The solution is introducing a second signal, as already done with the other communication process, which is controlled by the monitor and used to inform about the state of this board. It informs about whether or not the monitor is ready to take another package.

The conclusion is that the way of introducing the handshake protocol in this process is by using two new signals which indicates the state of the boards: data_available for the DUT and ready_out for the monitor.

The working principle of the sending information process is the same as before. The packages already prepared and sent are counted by both boards and this number is compared to the total amount of lots that are necessary in order to know the state of the process. The main difference is that in the original design only one module was implemented, and it was for the DUT board, but in order to introduce the handshaking protocol, it is necessary to develop one new module for the monitor board in addition to the other one.

The communication protocol working principle is divided in three steps that are summarized in the following image.

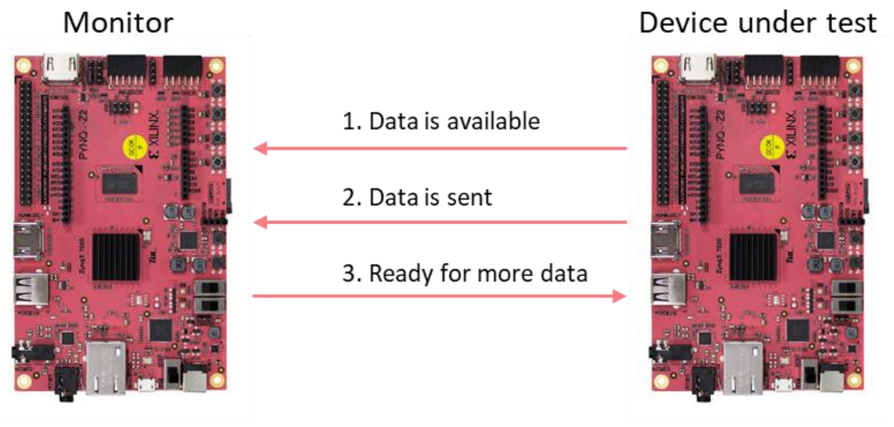


Figure 37. Sending information handshake protocol

Here is a deeper explanation about how the process is carried out with the handshaking protocol and the new signals introduced. The cycle starts whenever the unit under test has finished its tasks. The last step of this process is to set the signal used as an enable of the sending information module, which triggers it. It is important to stress that this is the only way of starting this module.

When the process is triggered by the enable, the first step is to check whether or not there are packages left to send, and since it has just begun and there must be at least one lot, the information is taken from the output of the test and stored in OUT_COM. At the same time, the package counter is increased. As it was done in the other process, it is necessary to assure that this branch of the code is executed only once each time so that no information is lost, and for this purposed, a flag is used. This signal is also used in order to activate the condition of the next part of the code.

After one clock cycle, when the flag signal is high, and as long as the ready signal is low, data_available is set in order to inform the monitor that there is available information in the channel shared by the two boards. In this part of the code, the flag is disabled so that this branch is not executed more than once. Since the ready is set to one yet and no other condition is satisfied, the process executes the default condition in which the signal data_available is set to zero.

On the other board, when data_available is set to one, the process is activated. As done in the DUT module, it is check whether or not there is a package. Then the data is taken from the channel and it is stored in the monitor board. At the same time, a flag is activated. The reason is the same as before, it gives more time to the system to process the information and it is assured that there is no errors in this task. After a clock cycle, the next condition to be satisfied depends on this flag, that must be set, and the signal data_available, that has to be low. This avoids endless loops and assures that the process has ended in the DUT before starting a new one. When these two conditions are verified at the same time, the signal ready is set to one, but only for one clock cycle, since there is another flag that sends the process to default case in which ready is set to zero.

This cycle is repeated until the moment in which the module of the DUT is enabled by the ready signal but there are no more packages to send. In this situation, the system sets the end process flag that activates for the last time the signal data_available. After a clock cycle, the finish condition is executed. Here, only the signal data_available is set to zero. The rest of the signals are set to their initial values at the beginning of the module. When the enable from the test activates the process, the first thing done is to set the counter for the packages and the flags used to their respective starting values.

In the monitor board, when the last data_available warn gets to the module, the system executes the finishing condition as well. This time, the ready signal is not set high in order to avoid an infinite loop. All the flags and the package counter is also set to tis initial value.

The procedure just explain has more complexity that the receiving information one, and it is due to the needs of the software to control the restart of a new complete cycle. But this will be explained in the next section.

In order to clarify the process that has been explained, it is shown a timing diagram with the most relevant signals of the communication protocol. The example used is for sixteen outputs,

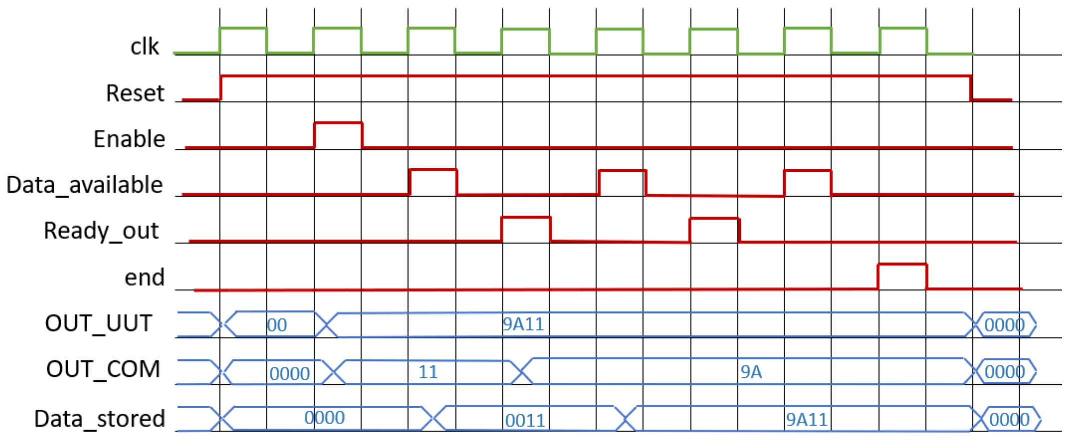


Figure 38. Sending information handshaking protocol: timing diagram

which means that there are two complete packages. The number of inputs or the data used is not relevant for this diagram, so the information is taken from the other examples. Since it has not been explained yet where or how the data is stored when is received in the monitoring system, for this timing diagram it has been introduced a signal which takes the values of the packages.

At this point, it is important to remark the most important events in the process. As it was said, the process of transmitting information starts only when the enable that comes from the unit under test is set. This situation triggers the action of loading the data in the channel, and after this, the data_available signal is set. At this moment, the monitor system is the one that takes the information and stores it. Since there are more packages left, the ready is set, and the process starts again, loading new data.

The complex part of the process comes when the second, and last package arrives to the monitoring system. The data is then stored, and even though there are no lots left, the ready_out is set again. The reason is that this last cycle in the DUT is used to finish the process. Although it is not pointed in the timing diagram, at the same time of the last data_available set, the ending signal of the DUT communication process is also set. However, the communication is not over yet, since there is one cycle left in the monitoring system.

This last activation of the process is used to reset the module as well, and to set the end of the total transmission signal. It is important to assure that the code does not activate the ready signal, since it is easy to conclude that the module would be in a never-ending loop, and this is not desired.

This communication process is more complex and takes more time than the one that transmits the information from the monitoring system to the DUT. Comparing the situation in which there are two packages, for the process that is being explained, the enable, called data_available, has to be set three times, one more than the other module. The difference is the same with the ready signal, which is set two times instead of one from the other process. In other words, the process of receiving information from the DUT needs one more cycle of the protocol to finish that the sending information to it.

The conclusion of this two last sections is that the communication between the two boards is more reliable after introducing the handshaking protocol. At this point, the modules are not only implemented in the DUT board, but they are also needed in the monitor one.

3.4. Boards design

Once the communication module has been conceived, programmed and tested, the next step of the project is to design the rest of the system that has to be implemented in the DUT and the monitor board. Obviously, these two parts of the testbed require different designs as their jobs are not the same, and for this reason, it is necessary to develop two different, and in some way independent, projects with their own block design.

At this point, it is important to stablish the general requirements in order to apply them on the design of these two systems. Some of them are directly derived from the general objectives of the project, and some other are obtained from the purpose of doing this testbed as useful and general as possible.

The first requirement, which is also an objective of the project, is that the testbed must be carry out in a cyclical way. In other words, once the system has been run for the first time, which means that the data has been sent from the monitor to the DUT, has been gathered, tested and then returned to the first system, the next cycle of these actions must begin again, without any human intervention. The reason for this requirement that it is desired to check at which frequency these cycles do not work as expected.

There are several ways to tell the system that the first cycle should begin. The easiest way of doing it is by having it running as soon as the board is connected to power and programmed. However, this is not a good idea since it is not possible to control the system. The other two options are to trigger the cycle by interacting with some hardware from the FPGA, as a switch or a push-button, or by sending a signal from the computer to the board that will indicate the start of the process. The easiest way is to implement a start button, also is the simplest one in terms of hardware and software design. The only disadvantage is the possible bouncing of the signal, and the designer must be aware of this situation and implement a solution in order to avoid any problems that it may cause.

The second requirement is the way of feeding the input data to the testbed. As it has been said, the system must be run cyclically as fast as it can, and for this reason the data cannot be fed manually by the tester through a computer. The information must be available for the monitor system in a way that it can be taken when needed and sent to the DUT. The way of doing this is by enabling a memory in the monitor board in which it can be stored all the information.

The third requirement is related to the second one. The data that is received from the monitor system, and which is the result of the test carried out, must be stored in a memory. The reason is that this information must be examined once all the cycles have been carried out, and not during the process since any interaction would make it slower.

The last condition is related with the clock that governs the boards. As it has been explained in the section referred to the Pynq-Z2 and its microprocessor, it is possible to use the processing system as a source for the clock. The advantage of this implementation is that the frequency can be easily controlled and modified by just adjusting the configuration of this IP, whoever the range of available frequencies is limited from 0.1 to 250MHz. This is not a problem, since it is expected that the system will work and also failed in this range of values, so it works for the project.

The other decision is related to whether or not the communication modules have to share the same clock source, this is, if the signal has to be sent from one board to the other. Although this issue was already addressed, is important to remember that the decision is that each board will have as clock source their own processing system, which have to be at the same frequency in order to assure that the FPGAs are synchronized.

As a conclusion, and before starting the design of each board, it is important to clarify which are the task of the two subsystems. The monitor FPGA is the one in charge of managing the starting of a new cycle, regardless of whether it is the first cycle or any other. This board is also in charge of taking the input data from the memory and sending it to the DUT whenever it is required, so it has to be synchronized with the start process and with the other board. It is also in charge of receiving the packages that contain the output of the test and storing the in the correct part of the memory. Last, whenever a cycle is over, it is in charge of triggering a new one.

The device under test board is in charge of receiving the information from the monitor and gather it together in order to send it to the test. It is also the one that contains the test that has to be carried out, and whenever it finishes, it must send all the data split in packages to the monitor board.

3.4.1. DUT board

In this section, it is explained the block designed for this FPGA. As it has been explained, the device under test board is the one in charge of two tasks: first, is the one that contains the unit under test, the most important module of the system, and second, the communication, the packages must be received, gathered in one signal and fed to the UUT, and the output must be divided in packages that have to be sent to the monitor.

One thing that must be taken into account is that once the information has arrived to the DUT, all the tasks that are carried out by this board do not require the interaction with the monitor nor with the computer, as they modules are executed sequentially, independently of the state of the other FPGA. Also, it doesn’t require from any interaction with its hardware, as buttons or switches.

Although no interaction is needed with the computer, it is possible to send information about the state of the board in real time in order to check how the process is being developed. However, this decision it not advisable since it is time consuming and it creates difficulties for making the design in terms of hardware, since more IP blocks would be required, and in terms of software, since it is required in order to control those blocks and get the information to display it in the computer. Moreover, all the information about the status and the data of the process can be read more easily from the monitor board, since it is expected that this board will have to use more hardware and software in order to store the results.

These details are important since it suppose a great simplification of the design since it is not necessary to introduce any other IP block apart for the processing system and the modules that have already been designed. Also, it is concluded that it is not necessary to develop any software application to control this board.

After the analysis of the system, it can be concluded that the design of the DUT is composed by four modules that are explained next.

3.4.2. Processing system IP block

Processing system IP block. As it was explained in the chapter related to the Vivado Design Suite software, one of the advantages of using this program with the Pynq-Z2 is that there is available an IP block in the library which easies the process of configuring the microprocessor of the board. This module is called ZYNQ7 Processing System and is the one used for this purpose. By using the wizard tool available, configuring the functionalities desired is as easy as selecting the ones needed and choosing the options.

It has been said that from the point in which the communication process is activated, all the modules are run sequentially, and this means that there is no need of any interaction with the processing system, as it is not required any interruption or any exchange of information. For this reason, all the AXI interfaces are not used and they must be disabled in order to avoid errors.

The only thing that is required from the processing system is the clock that has to run all the synchronized processes of the modules. As it was explained, it is possible to generate a clock from the masters one, choosing the frequency at which it must work. Although it can be generated for multiple purposes, the one used is the programmable logic fabric clock, as the modules generated in VHDL are considered part of the PL.

Obviously, it is not necessary to generate a clock for each module, moreover, it is better to use only one signal for all of them. This assures the synchronization between modules and avoids errors of setting different frequencies for each clock generated.

3.4.3. Communication modules.

These blocks, that are the ones that have been explained in the last chapter, are designed and programmed for this project. As it was said, each board has two processes that control the communication and are in charge of sending and receiving information, always from the point of view of the DUT. Since this two processes are independent from one another, it has been decided to divide them in two different modules, this is, two different blocks. This is done to ease the task of designing the process as the layout resulting from this decision is cleaner. This has as a consequence that the connections between modules and boards is facilitated.

The way of adding this VHDL module to the project is by creating a design source in this language and adding the code. After this, it is as easy as dragging the source to the window of the block design once it is created.

The first module is the one in charge of receiving the data from the monitor board. The VHDL source is named communication_module_input and it is composed by eight signals. The nomenclature given in this section is the one used in the block design and the modules.

It is important to differentiate between the internal ones and the externals. The first type are the one used to connect two modules of the same board. As they are not connected to any hardware, they are not declared in the constrain file. The second one are the ones that must be connected to the pins of the Raspberry Pi header or to any other hardware as leds or buttons, and they must appear in the constrain file in order to have a successful implementation of the design.

The inputs are the clock, which is an internal signal that comes from the processing system IP block. Al already explained, runs the majority of the process except for the reset condition. This last one is another input signal. However, it is a global one and is send through the Raspberry Pi connectors from the monitor board. The reason is that when a reset of the system is desired, both boards have to go thought it at the same time. This signal is different from the reset button that is available in the Pynq-Z2. By pressing that button, the FPGA is set to its default value and the implementation and software application are deleted, whereas the other reset only sets all the modules to the starting situation without deleting them.

The other two inputs are also global ones and are directly used for the communication. The first one is IN_COM and is the channel between the two boards used to send the packages. The other one is the enable of the process, enable_in_COM, and its use is explained in the section dedicated to the handshaking protocol.

The outputs of the module can also be divided in these two types. The two internal signals are the enable of the process of the test, enable_shift_com, and IN_UUT, which is the signal that gathers all the information from the different packages and sends all the inputs to the UUT.

The external outputs are Ready_in_COM, which is also used for the handshaking protocol that has been explained before. The last signal is led_input, and it is connected, as it name implies, to a LED. This piece of hardware turns of once the process is over, and it is turned off when the process starts again. The idea of using this LED is to determine where is the failure in case of an error. By introducing this signs in all the processes, the LED that is not on is the process that did not finish. As it is reset in the beginning of the process, this procedure can be used even when the number of cycles is bigger than one.

This system of LEDs is also introduced for the handshaking protocol, and it is done by connecting the signals enable_in_COM and Ready_in_COM to two different leds, that turn on when the signals are set. If the process fails because of one of this signals, it can be easily discovered.

The second module is the one in charge of sending the output result from the test to the monitoring system. The VHDL source is called Communication_module_output, and is composed, as the other one, of nine signals which are divided in inputs and outputs and internal and externals.

Out of this nine signals, five are inputs, and two of them are global. The first one is the reset. This signal is shared with all the modules that have been designed and this excludes the processing system, which has its own reset in the board. It is generated in the board of the monitoring system and then processed and sent to the DUT. The other signal is called Ready_MON and it is used for the handshaking communication protocol. It is the one that informs the DUT that the monitor board is ready to receive a new package, if there is any left.

The other three input signals are internal. The first is the clock, and, as it has been explained, it is generated by the microprocessor and is the same for both communication modules. The other two signals come from the UUT. These are the enable of the sending information process, called Enable_out_COM, and the signal which stores all the outputs of the test, OUT_UUT_COM.

There are four outputs of the module, and all of them are global. Only two of them are used for the communication purpose. The other two are used, as explained in the other process, to ease the process of detecting errors as they are connected to LEDs.

First, the explanation of the ones that are used for the communication purpose. One of them is Data_available, which is the signal that informs the monitoring system that the DUT is ready to send a package, if there is one left, and that it is already loaded in the channel. The second signal is this channel that connects the two boards together, and its name is OUT_COM.

The other two signals are called finish and led_com. The first one is used to verify whether or not this process reaches its end in every cycle. The other one is added due to the complexity of this module. Since there are a lot of communications between both boards, it has been decided to use another led at some points in the code to assure that the code is executed in the desired order. By setting and resetting this led, it is possible to discover where the boards struggle in the communication.

3.4.4. Block design

Once all the modules are created and added to the block design, it is time to connect them together in order to finish it. In this point is where it becomes easier to distinguish which signals are internal and which are considered external. The first type are the ones that are connected between modules, the output of one unit is the input of the following one. The global ones have to be externalized in order to be included in the constrain file and connect it to its corresponding hardware.

For this block design, it has been used the shift register created in the last chapter of the document as UUT. This block is the one that has to be changed in case different circuits are wanted to be tested with this testbed system.

In the following page, it is shown an image of the final block design for the DUT board with all the modules and signals connected.

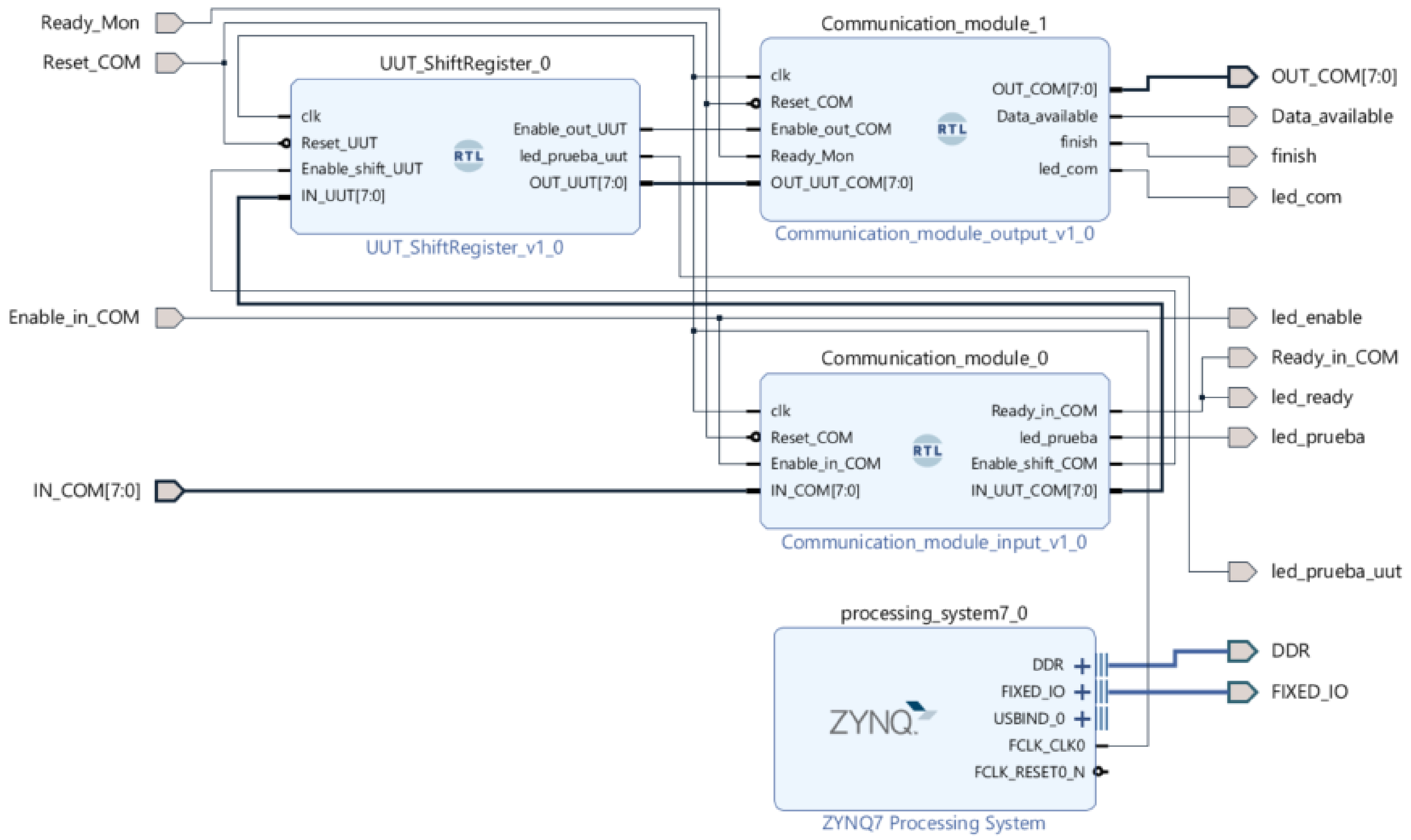


Figure 39. DUT board: block design

3.4.5. Monitor board

In this section is going to be explained the design and implementation done for the monitor FPGA. As it has been explained before, although the DUT is the board that contains the test, which is the most important module of the system, this board is the one that actually controls the testbed. It is the one in charge of most of the tasks of the complete system, including allowing the functioning of the DUT. The principal tasks of the monitor system are the following ones.

First, this board is the one in charge of taking the input information for the test from a memory, which stores data for multiple cycles. At the beginning of each cycle, the system has to be able to select the correct inputs for all the ones available and send them through the channel to the DUT. It also is the one in charge of storing the corresponding outputs in the memory. This task may seem simple, but the information has to be stored in a way that does not produce overlapping and in which it is easy to find the desired results.

Second, as it have been said, this FPGA is the one that contains the hardware for the internal reset of the testbed. This signal has to be associated to a switch or a push-button, and to a Raspberry pin from the connector in order to send it to the DUT board.

It is also in charge of the start button or switch, which is the one that allows the first cycle of the testbed. However, the rest of the cycles must not depend from the state of this signal, and another solution has to be found. This is a task of the monitoring system, implementing the procedure to restart the system once a cycle has finished.

The last responsibility of the monitor board is the communication. As it was said before, the exchange of information between the boards is not depend only on the DUT, but also in the monitor. For this reason, this system counts with a communication module that is in charge of taking the information that comes from the memory and giving it to the DUT, and it is also in charge of storing the output packages from the test in the corresponding spaces of the memory installed in the FPGA.

As it can be seen, the complexity of this board is greater than the one from the DUT, and some of the tasks cannot be carried out without the interaction and exchange of information between the PL and the PS. For this reason, it is needed to use some AXI blocks, as well as some memory block.

The final conclusion is that it is necessary to develop both a block design and its corresponding software application for the correct functioning and complete compliance with the tasks of the monitor board.

3.4.6. Operating principle

Before starting with the design, it is necessary to decide how some of the tasks are going to be implemented, in order to know which blocks are needed for the design and the software application.

From the five tasks that were just described, not all of them require the same attention. The communication duty is already explained and designed in a previous section of the document. Also, the start and reset implementation do not require from any special block, as they can be used as inputs to the desired modules. The tasks that need a deeper explanation are loading and storing information in the memory and re-starting a cycle after the previous one has ended. First it is going to be explained how the process of taking or saving information from the memory is managed.

It is important to decide beforehand the implementation of the memory in the board, since this will affect the interaction between this block and the others. There are two options. The first one is to design this memory in VHDL code. The procedure would be the same followed for the communication modules, that is creating a source file in which it can be added the code and after this, put this module in the block design. The advantage of this solution is that it can be designed for the specific needs of this system and would assure a perfect fit. Also, the connection between the memory and the communication module would be as easy as letting the information go from one to another, but this memory would have to be in charge of selecting the correct information from all the available one. The disadvantage is that the idea of the project is to have as many different information for input of the cycles as possible, and this requires creating an efficient and effective memory, which is not as simple.

The second option is to use one of the blocks that is available in the library given by Vivado. The IP block is named Block Memory Generator and, as its name implies, it creates an optimized memory. The way of storing the data and loading is by developing the correspondent instructions in the software application. In order to control the memory block, it is necessary to add another block, which is the AXI BRAM controller, It allows the designer to use, in an easy way, the memory generated, as it connects.

The next step is to find a way to transfer the information from the memory to the communication module and vice versa. As it has been said, the control of the memory is done through the software application, so it is possible to load the information desired in a signal. However, this signal has to be connected in some way to the block design in order to be fed to the DUT board. This can be done by adding the AXI_GPIO block. In the software application side, the memory can load the information, and through the AXI, it can be used in the PL part of the block design by connecting this module to the one desired.

The option of implementing the memory by using the IP block implies more complexity in the design, however it is compensated by the facility of use. Also, using this block implies saving the time that would require developing the memory specifically for this project.

However, this implementation presents one unexpected issue, the time that requires loading or storing the information in the memory delays the communication procedure. In other words, the information that is sent to the DUT or stored may not be de correct one. When the start button is pressed, the first package is loaded in the AXI_GPIO, but at the same time all the process of the communication is enabled and the first package is sent, even though the information has not arrived yet. For this reason, the information of the first package is sent in the second one and so on. The same happens with the information that arrives from the DUT. The other problem is that it is not possible to know from the software application when it is the moment to load the next package, or to store the information. It is necessary to modify the communication module that is on the monitor board.

First, it is going to be analyze the communication module in charge of sending the information to the DUT. The problem here is that the system has to know, beforehand, when the data has to be taken from the memory, and also has to assure that the package is loaded in the channel before activating the signal that warns the DUT that the data is available. The way of doing this is by adding the signals of the handshaking protocol, enable and ready, to the software application, in order to interact with the memory in the right time. For this purpose, AXI_GPIO blocks are needed. However, it is also needed the signal that starts the cycle for the first time, start, which is the one that comes from the switch.

The way of proceeding with the communication is the following one. When the start is pressed, then the software application gets an interruption that loads the package in the correspondent AXI_GPIO and sets a signal that informs of the finalization of the task. This signal has to arrive to the communication module of the block design, and so another AXI_GPIO is used. The data is then copied to the channel that is shared by the boards, but in order to assure that the DUT gets the information, it is waited another clock cycle to enable the process of the board. This is done through another AXI_GPIO signal, that is set when the start is pressed, that allows, as long as it is set and the ready signal is disabled, the condition that sets the enable. This last signal is disabled as soon as the software application finishes the start condition.

It is necessary to remark that the AXI_GPIO connected to the start is allowed to interrupt the processing system. This is important, as the software application cannot be checking all the time the state of the signals. Also, it is needed to note that the AXI_GPIO that sends the data form the memory is not directly connected to the channel, this is done through the communication module, and it is like this for two reasons. First, it is controlled when the data is sent to the DUT, avoiding errors, and second, the modules were designed like this and it has been maintained.

In case there is more than one package, the ready signal works as the start one, and interrupts the software application, generating the same behavior of the modules as the one just explained.

In total, there are five AXI_GPIO blocks used for the process of sending information from the monitor to the DUT board. This increases the complexity of the design, but also assures that the correct information arrives to the test every time. In the following image, it is shown a simplified process flow. This diagram only describes the process of sending data, without adding all the part of counting the packages or ending the communication.

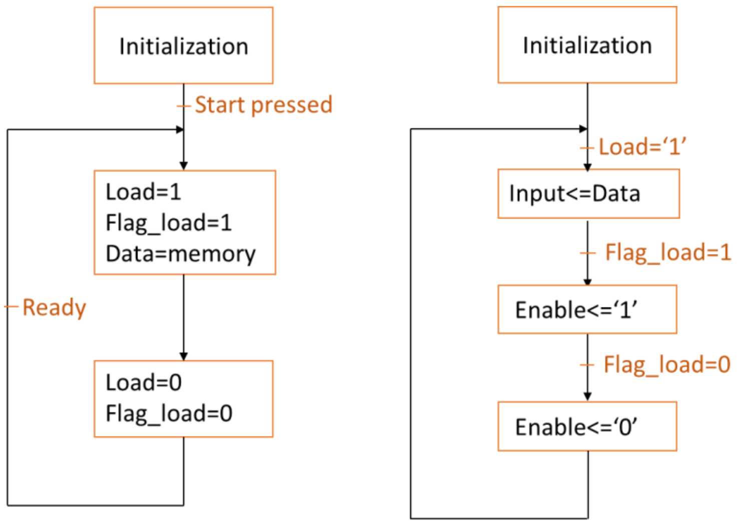


Figure 40. Process Flow for Monitor to DUT communication: left software application, right VHDL module

Once this module is explained, it is time to analyze the communication module that allows sending the output of the test from the DUT to the monitor board. The problem in this transmission is that the software application has no way of knowing when the packages are coming its way, and therefore the information cannot be stored in the memory. The solution is to add some AXI_GPIO signals that are set in the VHDL code of the block design and can be used in the software application. In this case, it is not necessary to use the ones from the handshaking protocol, and new ones may be created.

The new operation of the process is as follows. When the DUT has created the package, that is filled with output information, and it is loaded in the channel, the signal data_available is set and triggers the process of the monitor board. This, in turn, sets a signal that is connected to an AXI_GPIO, which produces an interruption of the software application. The data from the channel is then read and saved in the corresponding place of the memory, and also a flag that warns of the finalization of this process is set. This signal is used in the VHDL to activate the next condition, and therefore is used as an AXI_GPIO. The signal of the interruption is disabled in this moment and the ready used for the handshaking protocol is set, informing the DUT that the monitor is ready for more packages, in case there are any left. After a clock cycle, the signal ready is disabled, and the process waits for the next set of data_available.

For this process, it is only needed three AXI_GPIO, and only one is able to interrupt the software application. In the following image, it is shown a simplified process flow of the updates of the communication module. This diagram only describes the process of receiving and storing the data in the memory, without adding all the part of counting the packages or ending the communication.

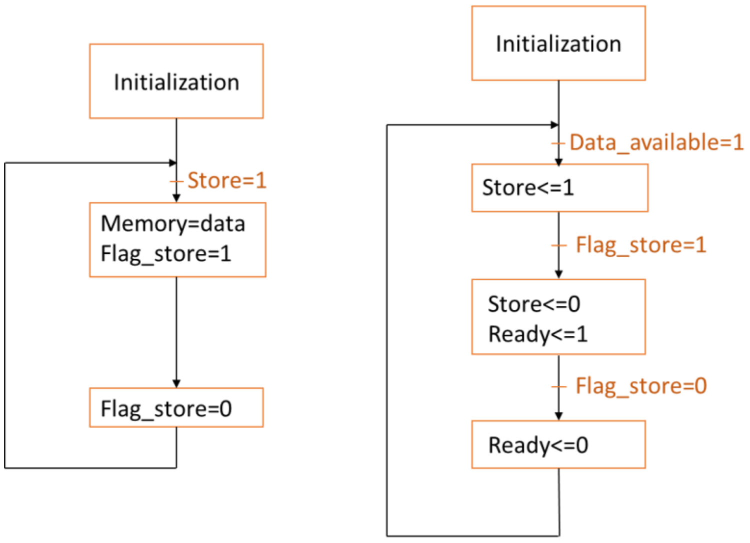


Figure 41. Process Flow for DUT to Monitor communication: left software application, right VHDL module

The last task of the monitoring system that has to be implemented is that whenever a cycle is finished, another one has to start right away. This functionality can be easily implemented after studying the communication modules. The process of sending information from the monitor to the DUT is triggered by the start signal, which depends on a switch, and for this reason it cannot be reused. The other option is to create a new signal that can commence the communication as the start does. Implementing the solution, the beginning of a test cycle depends on two signals which are the state of the start for the first cycle, and the sate of this new variable for the rest of them.

The way of implementing this procedure is by finding a signal that informs of the finalization of the previous cycle, as this is the moment in which the system has to prepare the next round of test. When the communication module than sends the information from the DUT to the monitor was designed, a signal that is set when the process has finished was enable. This signal is named end_transmission.

When this signal is set, it means that the last package with output information has been received and stored in the memory, so the next cycle is ready to be started. There are two ways of implementing this re-start. The first option is to do it in VHDL, however this solution does not sound useful, since the process that sends the information from the monitor to the DUT sending information is controlled from the software application. Also, the other problem is the way of loading the first package in the communication channel.

For this reason, the best solution is to implement the re-start process using the software application, and the way of doing this is by connecting the end_ transmission signal to an AXI_GPIO block. Since the system cannot be constantly checking the value of this variable, it must be able to interrupt the software application, as this is the only way to ensure that the signal is not missed without having the FPGA constantly blocked.

The re-start process works as follows. When the signal end_ transmission is set, it produces an interrupt and the software application executes the function associated to it. The code contained in the function is similar to the one of the start interruption, as the tasks are more or less the same ones. First, the new inputs are loaded in the data signal which is send to the communication module. At the same time, the load and flag_load signals are set, and they trigger the different stages of the communication process.

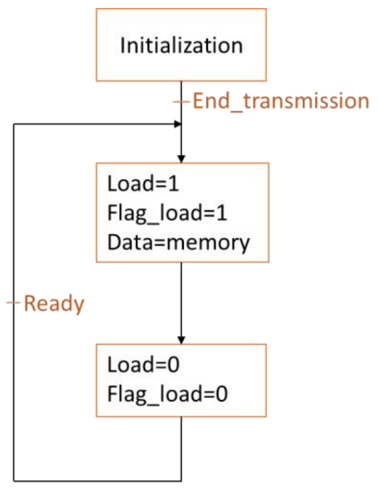


Figure 42. Process Flow for re-start the cycle

After the function is executed, both signals are immediately reset to zero and the rest of the cycle is executed in the same way as it was explained before. This is, when the DUT is prepared for the next package, the ready signal is set, and the sending package process is carries out again.

3.4.7. Block Design

Once all the issues that were found to implement the tasks have been solved, it is the moment of developing the block design for the monitor board. Unlike the one done for the DUT, this one is more complex due to the number of IP blocks required for the communication and for the memory.

In the following page, it can be seen the final block design developed in Vivado Design Suite. After it, each block that forms part of the design and its configuration will be explain.

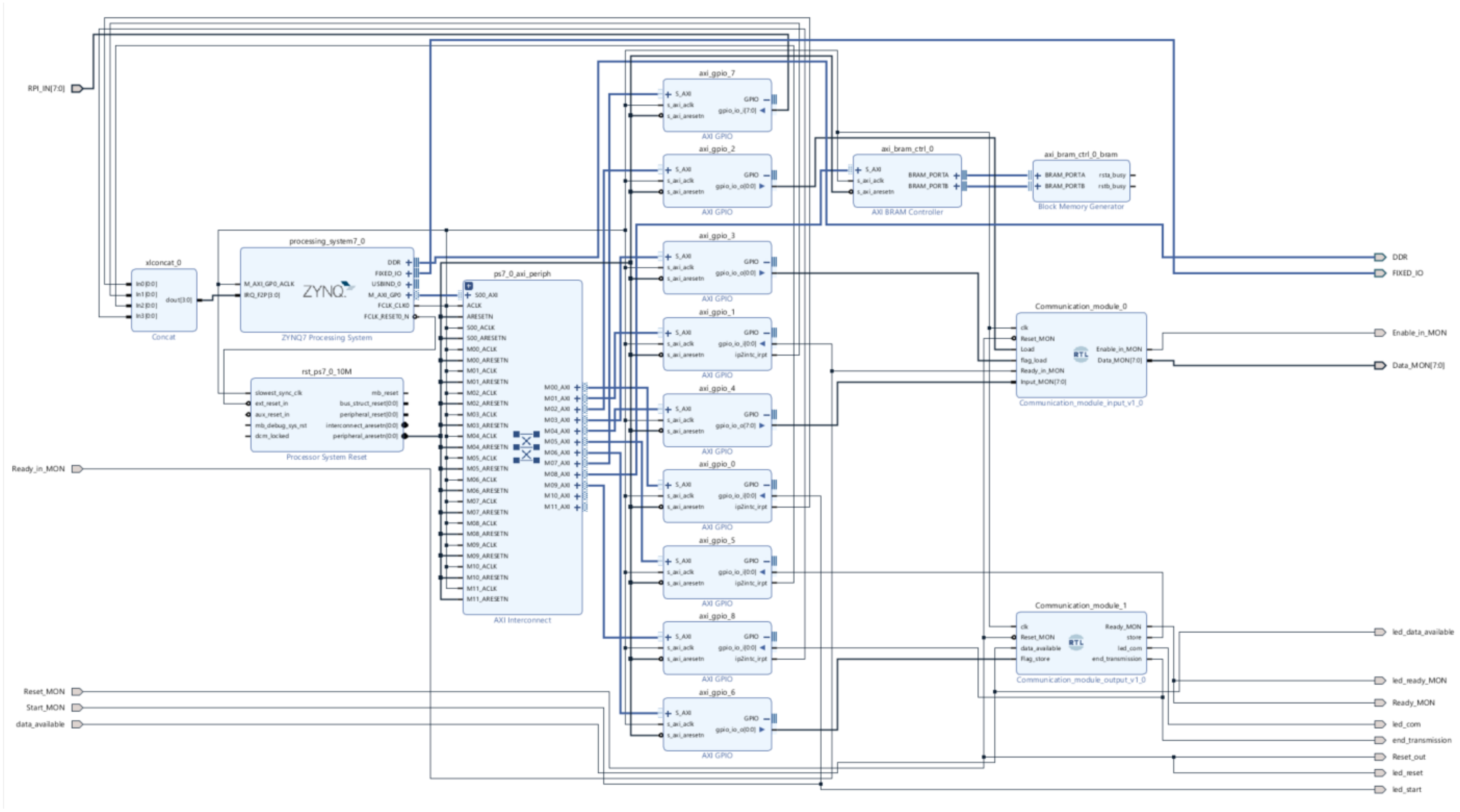


Figure 43. Monitor board: block design

- Processing system IP block.

As it was done with the DUT board, the module used is the ZYNQ7 Processing System, which must be customized to meet the requirements of the monitor board.

First, this block is the one that generates the clock signal which is used by the rest of the modules of the board. As it was explained, each FPGA generates its own clock from the primary one, and the frequency is the parameter that must be chosen. It is important to mention that in this board this clock is not only used by the modules that have been designed, as the communication ones, but also by the AXI_GPIOs, the AXI communication, the memory and so on, and all of them share the same one, FCLK_CLK0.

Also, in this board there are two different resets. The first one is the one controlled by the user, which is implemented via a switch or a push-button, and controls the modules that have been designed for the project: the communication ones and the UUT. But it is also necessary another reset signal which can be used for the other modules, such as the AXI. This clock reset is generated by the processing system to be used by the PL and is connected to a block called Processor System Reset. This core is the one that provides the customized reset used by all the interconnects and peripherals, such as the memory controller or the AXI GPIO.

As it has been explained in the last section, some of the blocks used in the PL part have to be able to interrupt the PS. For this purpose, in the ‘Fabric Interrupts’ windows it has been activated the IRQ_F2P, that enables sixteen-bit shared interrupt ports from the PL to PS. This means that all the interruptions signals that come from the AXI GPIO blocks must be gathered in just one vector. In order to create this vector, it has been used the Concat IP Block which can be configured to indicate the number of inputs that are fed.

The last configuration is related with the AXI interface that interconnects the PS and the PL of the microprocessor. It consists on a communication between two parts, in which one is the master who controls the process and the other is the slave, which actuates as the master orders. These two parts communicate via five different channels: read address, read data, write address, write data and write response. . In this situation, the processing system block is considered the master in control, and for this reason it is necessary to activate the general purpose AXI master interface. This activates at the same time the clock signal for this communication interface, which is connected to the same clock signal as the rest of the block design.

The problem of this situation is that there is only one master, but there is more than one PL component that uses the AXI interface as a slave, and there is no way of defining which order goes to each one. For this purpose, it is necessary to add another block that is called AXI Interconnect. This module is in charge of interconnecting all the blocks in the correct way, assuring that the master communicates with the desired slave each time, and that no information is lost.

- Communication module.

These two modules are the ones that were explained in the chapter dedicated to this topic. As it was explained, each board counts with two process that are completely independent from one another, and each one controls one side of the communication: sending information form the monitor to the DUT and vice versa. The VHDL code of these two processes can be stored in one file, which would have as a result only one module, or in two different sources. For the sake of clearness, it has been decided that the second option is the best one for the project.

The first module to be explained is the one in charge of sending the input data from the monitor board to the DUT. The VHDL source is called Communication_module_output and it is composed by eight signals. As it was done in the block design explanation of the other board, these signals are classified depending on their direction, input or output, and if they are internal signals, that are used inside the board, or external, which are the ones that are sent to the DUT or that are connected to any component of the hardware, as it can be a switch, led or pins of a header.

There are six different inputs available in the module. The first one is the clock, clk, which is the one in charge of running the majority of the process, except the reset condition. As it was just explained, is produced by the processing system block, and for this reason it must be considered internal. The second one is the reset, Reset_MON. This signal is generated by the user through the hardware, push-button or switch, and it is different from the one generated by the processing system. It can be considered external and has to be present in the constrain file.

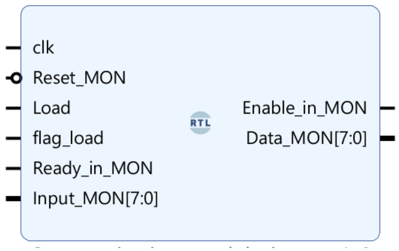


Figure 44. Monitor to DUT communication block

The two following signals are Load and flag_load and, as it was explained the operation principle section, are used to control the communication process from the software application. These two signals are connected internally to AXI_GPIO blocks. The next signal is Ready_in_MON and it is used for the handshaking protocol implementation. It is sent from the DUT to inform that the board is prepared for the next input package, and for this reason it is considered external. The last signal is Input_MON and it is in charge of carrying the input information from the memory to the channel that is between the two boards. The data is loaded in this signal in the software application, and it is used as an AXI GPIO, for this reason this last signal is also an internal one.

The other two signals are outputs of the module, and both of them can be considered externals. The first one is Enable_in_MON and is the other signal together with Ready_in_MON used for the handshaking protocol. It is the one in charge of triggering the DUT communication protocol, and this is the reason why it is external. The last signal to be explained is Data_MON and is the vector that sends the input information from the monitor to the DUT.

As it was done in the DUT board, it is useful to connect some of the signals to the hardware of the FPGA, such as the LEDs to be able to analyze how the process is being developed in case of an error. For this communication process, the important signals are Ready_in_MON and Enable_in_MON, the ones used for the communication protocol.

The other module of the communication is the one in charge of collecting the output data that comes in packages from the DUT and storing it in the memory. The VHDL source is called Communication_module_output, as it is always from the point of view of the DUT. This module has eight available signals that have to be connected to other modules or hardware.

Out of this eight signals, four of them are inputs from the point of view of the module. Two of them are the ones already mentioned in the other process, and that are available in all the modules designed. These are the clock, clk, and the reset, Reset_MON. the first one is an internal signal, as it comes from the processing system, and the second one is external, as it comes from a component of the hardware.

Data_available is an input signal used in the communication protocol. It is sent by the DUT to inform that another package has been sent and is available in the channel, and for this reason it is considered an external signal. The last one is Flag_store. This is an AXI GPIO signal that is used to control, from the software application, that the process of taking the information from the channel and storing it in the memory is driven correctly.

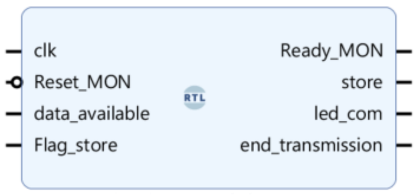


Figure 45. DUT to Monitor communication block

The outputs of the module are the following ones. First, Ready_MON, which is the other signal used to implement the handshaking protocol. It is sent from the monitor board to warn he DUT communication module that the system is ready to receive a new package with information. As it is a signal that goes from one board to another, it is external. The store signal is used to execute the part of the software application in charge of storing the data in the memory, and it is connected to an AXI GPIO, for this reason is internal. End_transmission is a signal that indicates the moment in which the communication reaches its end. It is also used in the software application to re-start the cycle of the test, and for this reason it is connected to an AXI_GPIO.

The last signal, which is led_com, is used for the validation of the code of this process. Since this module is complex, it is used to verify that an intermediate point of the code is executed by setting and resetting this signal in different moments. it is connected to a component of the board, such as a LED, and allows the designer to identify how the process is being developed. Also, the signal end_transmission is connected to a LED in order to warn, in a visual way, that the process has ended in a successful way.

- Memory Block.

As it was explained in the operating principle section, the memory for this project is implemented with a block that is available in the library of Vivado. The memory generator block is an advance memory constructor, that creates optimized memories in terms of performance using embedded block RAM resources from the FPGA. It can generate a single-port or a dual-port RAM, depending on the configuration of the IP. The advantage of the dual one, is that it is possible to read and write at the same time from different directions of the memory

It is possible to configure the size of the memory as well as the width of the words. This means that the user can decide the amount of data that is stored in the FPGA depending on the needs of the test by changing this parameter. The width of the word is the number of bits that can be read or written in each cycle. For our case, with eight is enough.

It is important to pay attention to the read latency, that is the time required for the processor from the request of data until it is retrieved. For the block memory used in this project, it is one clock cycle. This latency is the reason why it was important to add signals in the software application to assure the data was loaded in the signal before enabling the DUT to read from the channel.

The interface used by this RAM memory is not compatible with the AXI, which is the one used by the processing system. It is necessary to add another block that can be used as an intermediary between the two blocks. The name is AXI BRAM controller and must be configure as single or dual port depending on the memory block.

- AXI GPIO.

This block provides a general-purpose input/output interface to the AXI one, in other words, it allows the designer to use GPIOs in the PL part of the microprocessor connecting them to the PS through the AXI interface. As it was explained in the section of hardware tools, the number of inputs/outputs available (MIO) in the microprocessor is limited and most of them are already used by other components of the board. It is possible to extend those inputs/outputs by using the EMIOs, but since the number of signals is big and the AXI interface is already used for the memory, it has been decided to use this type of block.

Before using the AXI GPIO, it is necessary to configure it to meet the designer's needs. First, each block can have one or two channels, in the case of this project, all the IPs have only one channel. After this, the GPIO width, which is the number of bits available in each channel, must be configure. The maximum width of this GPIO is of thirty-two pins. Finally, by default the channel is configured as an inout signal, however it is possible to set the direction of each block. Although this is not necessary, it is recommendable as it avoids errors in the software application, since it only allows to read from the in GPIOs and to write in the out ones.

It has been said before that these blocks can interrupt the processing system from the programmable logic part. For this it is necessary to enable the interrupt of the AXI GPIO. This new port that is added is the one that has to be connected to the Concat module that feeds the interrupt port of the processing system IP. This means that whenever the data of the AXI GPIO channel changes, the block produces an interrupt signal that arrives to the processing system. The type of variation that produces the interruption can be configured in the software application.

In the block design there are eight different AXI GPIO blocks, and it is necessary to summarize all the signals that are connected to these IPs and divide them by the different processes to which they belong.

First, the communication module in charge of sending the information from the memory to the DUT. This process requires five different signals that are connected to an AXI GPIO, and two of them require the interrupt to be enable. In the following table, it can be seen the name of the signal, the type, if it is input or output, whether or not the interrupt is required, and last the name of the GPIO to which it is connected. This last information is really important, as this name is the one used in the software application, and it cannot be mixed.

SIGNAL NAME	I/O	INTERRUPT	TAG AXI GPIO
START	INPUT	INTERRUPT0	AXI_GPIO0
READY	INPUT	INTERRUPT1	AXI_GPIO1
LOAD	OUTPUT	-	AXI_GPIO2
FLAG_LOAD	OUTPUT	-	AXI_GPIO3
DATA	OUTPUT	-	AXI_GPIO4

Table 2. AXI GPIO: Communication from monitor to DUT

The signals that require the interruption are, as explained, the start and ready, which are the moments in which the data has to be uploaded in the channel to transfer it to the communication module. These are also the only inputs, as the rest of the signals are set in the software application.

The communication module in charge of receiving the data from the DUT to store it in the memory requires three AXI GPIO blocks. Only one of those IPs has the interrupt enabled and is the one that starts the process of saving the information once it has arrived at the module, so it has to be an input. The other two GPIOs are used to set the signal that informs that the module is ready to receive more information (flag_stiore), so it is an output, and to send the information from the communication module to the memory.

SIGNAL NAME	I/O	INTERRUPT	TAG AXI GPIO
STORE	INPUT	INTERRUPT2	AXI_GPIO5
FLAG_STORE	OUTPUT	-	AXI_GPIO6
RPI_IN	INPUT	-	AXI_GPIO7

Table 3. AXI GPIO: Communication from DUT to monitor

The last AXI GPIO block needed is the one used to communicate the end of the test cycle to the software application, and for this reason it is an input GPIO. The IP is connected to the signal end_trasnmission that is an output of the communication module. This signal, as it was explained, has to be able to interrupt the processing system, and for this reason the option is enabled.

SIGNAL NAME	I/O	INTERRUPT	TAG AXI GPIO
END_TRANSMISSION	INPUT	INTERRUPT3	AXI_GPIO8

Table 4. AXI GPIO: re-start cycle

3.4.8. Software application

Once the block design has been finished, and it has been successfully synthetized and implemented, the bitstream can be generated and exported to the SDK to start programming the software application. It is important to remark that the implementation cannot be done without the constrain file, which is explained in the next section with this name.

The software application is where the following tasks are implemented. First, here is where the parts of the programmable logic that require it, as the AXI GPIO, the memory block or the interrupts, are initialized and configured, since the bitstream only boots the processing system. Also, it is where the designer can add, manually and as part of the initialization of the memory, the values that are stored here and that, later, are fed to the test in each cycle as inputs. By changing this initialization, the designer can adapt the information contained in the memory to its needs.

Also, part of the communication process takes part in the software application as it was explained before. The inputs must be loaded from the memory and copied in a signal that is later send to the communication module, and the outputs of the test that are sent from the DUT must be stored in the memory in this part of the software application. Since it is important to control the timing of each action, it is necessary to use some signals to control the process, as store or load. Some of this AXI GPIOs can interrupt the processing system, and this interruptions have to be initialized and configured. Also, it has to be programmed what the system has to do when the interruption happens. Finally, the implementation of the re-start cycle is done in this software application as well.

All of this takes are programmed in one file of the project, which has the extension of '.c'. This file contains the principle function, which is called 'main'. When the board is programmed and the software application is run, this function is the one that is executed, and for this reason, all the tasks that the designer wants to implement have to be in this function or in an interruption one. This is, all the initializations and configurations of the blocks have to be carried out in the main function, and all the actions that comes as a consequence of an interrupt of an AXI GPIO must be programmed in their correspondent function.

As it was explained, when the bitstream is exported to Vivado SDK, a hardware platform is generated which depends on the information given by that bitstream. When a project is created with that hardware platform, some libraries associated to PL blocks are available as sources. This files contains functions and parameters which can ease the task of initializing these components and even using them in the code. The ones that are used for the software application are the following ones:

- xil_io.h. This file contains the interface for the general I/O components. This interface is composed of the input/output functions used by the processor that do not require any special I/O handling.
- xparameters.h. As its names implies, this file contains the system parameters for the Xilinx device driver environment. It contains the number of each device in the system as well as the parameters and memory map for each block. The designer can use this file to see all the devices that are contained in the system and its parameters.
- xgpio.h. This resource includes the software API definition of the AXI GPIO device driver. In other words, contains the functions to initialize and use this block in the software application.
- xscugic.h This file contains the necessary functions and drivers to manage the interruptions introduced by the interrupt controller component.
- xbram.h. It is form by the collection of functions that are necessary to initialize and control the AXI BRAM controller block.
- Xil_exception.h. This last file contains the specific exceptions related APIs.

The content of these libraries, parameters, functions…, can be used in the main.c file once they are added to it. This is done by using the '`#include`' header followed by the exact name of the library in square brackets.

As it has been said, when the board is programed with the bitstream, the processing system is configured, but this does not happen automatically to some blocks of the programmable logic. The configuration is done through the software application. It is necessary to explain how this initialization is done for the blocks that require it.

- AXI BRAM controller block. One of the libraries just explained is created for this device. It includes some functions that can be used to configure, xBram_LookupConfig, and to initialize, XBram_CfgInitialize the block. It includes as well the definition of the structures XBram and XBram_Config. The first one is composed by two different signals, config, which its type is the second structure mentioned, and a ready. The configuration signal is, in turn, another structure composed by all the variables that are needed to configure the block in order to use the memory. The ready signal informs when the XBram has been correctly configured and is prepared to be used.

This type of variables is extremely useful, since they already contain all the information needed, and the programmer only has to provide them with the correct values.

The way of configuring the block is by following the next steps. First, a variable of the class XBram has to be declared, in this case it is named Bram, and another of the type XBram_Config, which is ConfigPtr. When this is done, it is time to fill the configuration structure with all the data that is needed, and this is done with the function xBram_LookupConfig, that has as argument the identifier of the device, the AXI BRAM controller. It this is successfully done, then the last step is to initialize the Bram structure with the other function available, XBram_CfgInitialize, that has as arguments the XBram structure, the configuration structure and the memory address space.

Once this is all done, the AXI BRAM controller block is initialized and ready to be used, which means that the data can be written or read from the memory.

- AXI GPIO block. The library xgpio.h contains the drivers needed to configure this type of block. The way of proceeding with this device is similar to the AXI BRAM controller. The library contains two functions that simplifies the process of initializing the component. The first one is named XGpio_Initialize, and the second one is called XGpio_SetDataDirection.

  Also, the library contains two structures types. The first one is XGpio and its composed by four signals: the device base address, the ready, which indicates that the device is initialized and can be used, another signal that notifies whether this AXI GPIO has enabled the interruptions and last, the signal that informs if the block has one or two channels. The other structure is XGpio_Config and is composed in turn by another four signals: the first one is the unique ID of the device, and the other three are the same ones of the XGpio, excluding the ready signal.

  In order to initialize each AXI GPIO it is necessary to first declare a variable of the type XGpio, GPIO. It is important to remark that each AXI GPIO must have its own XGpio variable declared. After this, the function XGpio_initialize must be executed. The arguments given are the structure GPIO and the ID of the device. If this function is run correctly, then the device is initialized. But before using it, it must be declared the direction of this GPIO, and this is done with the function XGpio_SetDataDirection. The arguments are the GPIO structure, the channel that is being configured and last, the direction of the pins. In order to set one signal as output, the correspondent bit has to be set to zero, and to configure it as input, it has to be one.

  Once all this process has been done, the device is configured as input or output and it can be used.

- Interrupt controller. When an interruption is enabled in any of the AXI GPIO used in the block design, it has to be configured in order to act as the designer desires. This process is eased by the library created for this purpose xscugic.h. It contains the functions that have to be used for this process as well as the data types.

  The two structs defined for the interrupt controller are XScuGic and XScuGic_Config. The first one is composed by three signals, one from the type XScuGic_Config, another one that informs that the device has been initialized and is ready to be used and another signal stores the statistics of the unhandled interrupts. The XScuGic _Config struct contains in turn four signals, the first one is the ID of the device, the second is the base address of the CPU interface register, the third is the distributor register base address and the last one is a vector table of the interrupt handlers.

In this case, it is necessary to use three different functions to configure each interruptions, which are XScuGic_CfgInitialize, XScuGic_SetPriorityTriggerType and XScuGic_Connect.

Before starting with the process of configurating each interruption, it is important that the AXI GPIO associated to this signal is already successfully initialized and configured before doing the same process with the interrupt. .

The way of configurating each interruption is as follows. First, a variable of the class XScuGic and another of the type XScuGic_Config have to be declared. Independently of the number of interruptions of the system, it is only needed one of the type XScuGic. Once this is done, the function XScuGic_CfgInitialize has to be executed. The arguments of this functions are, first the XScuGic struct, then XScuGic_Config and the last one is the device base address in the virtual memory address space. This configures the interruption.

When this initialization has been done and no errors have been detected, it is time to link the correspondent interruption with its function and GPIO device. This is done by using the function XScuGic_Connect. The arguments of this function are the XScuGic variable, the ID of the interrupt source, the name of the function that has to be executed when the interruption is produced and last the ID of the XGpio device.

The last step of this procedure is to configure the priority of the interruption, in case there are more than one in a certain moment, and how it is triggered. This is done with the function XScuGic_SetPriorityTriggerType. The arguments are first the XScuGic variable, then the ID of the interrupt source, the level of priority, there are thirty-eight and go from zero, highest level, to 248, the lowest, in steps of eight. The last argument is the way the signal is triggered, which can be by rising edge, falling edge, active high level sensitive… For this project all the interruptions have the same level of priority 0xA0 and are triggered by active edge.

Once all of this is done, the interrupt is configured correctly, and it is necessary to enable this interruptions. This is done with three different functions. The first two are XGpio_InterruptEnable and XGpio_InterruptGlobalEnable, which allow the GPIO interrupts to interrupt. The argument of this function is the correspondent XGpio, and for each AXI GPIO block that has the interruption enabled, there has to be both of this functions. The other function is XScuGic_Enable, which enables the interrupt source provided by the argument. This function requires two arguments, the first one is the XScuGic variable, and the other one is the ID of the interruption source.

As it was said, each interruptions is linked to a function which contains the code that is executed when the device interrupts the processing system. Although the content of the function has to be designed by the programmer, the general structure must be the following one. First, the interruptions of this device have to be disabled in order to avoid having more than one in the same time. This is done with the function XGpio_InterruptDisable, which has as arguments the XGpio variable and the mask that disables the corresponding bits. Once the interrupts are disabled, the programmer can add the statements that are required to fulfill the tasks.

Once all the code has been added, the pending interrupts of the GPIO device have to be cleared with the function XGpio_InterruptClear, which has as arguments the XGpio variable and the mask that clears the bits of the interrupt desired. Last, the interruptions have to be enabled again with XGpio_InterruptEnable.

All these last steps that were explained are the ones that have to be done to initialize the blocks of the programmable logic. Each of the functions has to be executed in the main function before starting with any other task of the monitor board, such as loading data from the memory. Once this has been done, it is possible to start using the devices in order to transfer the data from the memory to the AXI GPIOs and vice versa. These libraries that contain all the functions used to configure the blocks, also contain the ones that are used for transferring the data.

- AXI BRAM controller. This device needs two different functions that are declared in the xbram.h file, one for writing in a determined part of the memory and the other one to read the information stored in an address.

  The first function is `XBram_Out8`. This one is used to store the desired information in a determined address. The number eight in the function implies the amount of bits that can be written in one execution. The function is also available for sixteen and thirty-two bits. This function requires two arguments:

  - The first one is the address in which the designer or user wants to store the input signals. The base address of the AXI BRAM controller can be found In the file xparameters.h. Since for this project it is needed more than one address to store all the inputs, the way of acceding to other parts of the memory is by writing the base address and summing an integer number to move the pointer to the following direction. Each direction can store up to eight bits, and for this reason, if the function for more than eight signals is used, the number of spaces that the address has to be moved from the base address has to take this into account.
  - The second parameter is the data that has to be stored in the address. This data is written in hexadecimal in this project.

  The other function is `XBram_in8`. This one is used to read the data that is stored in a determined address of the memory. As it happened with the other function, this one can only read eight bits, but it is possible to read more by changing the eight for a sixteen or thirty-two. When calling the function, it is only required one argument, which is the address from which the designer wishes to read the data. As before, it can be used the base address for this project, moving it a determined quantity of spaces.

- AXI GPIO. As for the memory controller, these blocks have two different functions declared in the file xgpio.h, one of them reads the state of the GPIO and the other writes the value of it. However, only one of them can be used for each bit of a device. The reason for this situation is that each pin is declared as an input or output when the GPIO device is configured and initialized. When a bit is declared as input, it is not possible to write a value in this pin, so only the read function can be used. In the opposite situation, when the bit is declared as output, the system is the one in charge of setting the value of this pin, and the write function has to be used. However, in this last case it is possible to use the read function for this pin, but it is not useful since the value that will be read is the one that was written in a previous line, when the write function was executed.

  This project has the particularity that those blocks whose width is bigger than one has all the pins declared as inputs or outputs, and no block has mixed signals.

The first function is `XGpio_DiscreteRead`, and as its name implies, it is the one used to read the state of an GPIO. When this function is used, the programmer must provide two arguments:

- The first one is a pointer to the AXI GPIO device, which is a variable of the type XGpio. This device has to be initialized before using it.
- The second parameter is the channel of the device which wants to be read. As it was explained, the AXI GPIO can have two channels, but for this project all these blocks have only one.

The second function is `XGpio_DiscreteWrite`, and it is used to write the value of an GPIO declared as output. This function has three arguments:

- The pointer to the AXI GPIO device, which is linked to a variable of the type XGpio.
- The channel in which the programmer wants to write the data. As it was said, in the case of this project, there is only one channel available.
- The data that has to be written in this AXI_GPIO. The amount of bits of this data has to match the width of the GPIO that was configured in the block design.

These are all the special functions of the devices that are declared in the available libraries. Now, it is the moment to design and program the software application. For this purpose, it has been chosen the C language. It is important to structure in the correct way how and when the different tasks of the monitor board are going to be carried out, and which functions have to be used in each moment. Following this, it is explained the structure of the main function and the interruptions. The code of the software application can be found in the corresponding appendix I.

As it has been previously said, the first that the software application has to do is to initialize devices that require it, which are the AXI BRAM controller and the AXI GPIOs. For the first device, two routines have been created, as this is the way of keeping the main function simple and with the smaller number of lines as possible. The first one is init_bram() and contains all the functions explained in order to initialize the AXI BRAM controller. The second one is init_value(), which is used to fill the memory with the inputs for the test with the aid of the XBram_out8 function. This routine can be implemented as many times as desired in order to have a big variety of data for the test.

Once the AXI BRAM controller is ready, and the memory has been initialized with data, it is time to configure the AXI GPIO, and the function used for this is init_GPIOnumber(), in which the tag number has to be substituted by the number of the tag of the AXI GPIO in the design block. In this function it is executed the initialization of each device and the direction of the pins. This function has to be executed for the nine different blocks that are used in the block design.

Some of this AXI GPIO blocks have the interrupt function enabled, and these interruptions have to be configured and connected to their respective routines and GPIO devices. This task is carried out by the routine init_interrupt() that is called from the main. This function calls, in turn, to another one named IntcInitFunction(), which has as arguments the ID of the interrupt device and the XGpio variables created for each AXI GPIO block that can generate an interrupt. In this routine, all the steps that were explained in the section of initializing interrupts are followed.

For this project, there are four AXI GPIO that can produce interruptions of the process, and for this reason, there are four functions to which are linked:

- START_Intr_Handler. This function is executed when the start button is pressed, and it is charge of preparing the monitor board for the beginning of a cycle. For this purpose, the first package is loaded in the correspondent GPIO and the flags 'load' and 'flag_load' are set. The way of following which package has to be loaded from the memory is by using a counter variable as a pointer summed to the base address of the block.

  The problem of using some hardware for the start signal is that there are rebounds, and for this reason this interrupt can be produced more than one despite the configuration of the trigger. This is a problem, since the counter can increase its value by more than one, and some information of the memory may be missed. The way of avoiding this is by using a flag that assures that all the lines are executed just one time. Since this interruption has to be ran just one time in the software application, this flag does not have to be reset.

- READY_Intr_Handler. This routine is linked to the signal ready of the AXI GPI, which is used in the handshaking protocol to warn the monitor that the DUT is ready to receive another package. The lines of this function generate the same behavior of the board as the START_Intr_Handler, this is, the inputs are loaded in the GPIO and the flags load and flag_load are set.

  In this case it is not necessary the used of a flag, since the ready signal is set for one clock cycle, and there is no danger of missing memory information by increasing the variable.

- STORE_Intr_Handler. This routine is executed when the GPIO store becomes active and is the one in charge of storing the incoming information from the DUT in the correspondent address of the memory. For this purpose, it is used another counter variable that tracks the address of the outputs. This function also sets the signal flag_store used to continue with the communication process.

  Although it may not be necessary, this function also uses a flag that assures the function code is executed only once. This flag is reset at the same time as flag_store, which is in the main function.

- END_TRANSMISSION_Intr_Handler. This last function is executed when the cycle of the test has finished, and another one has to start. It is in charge of loading new input data from the memory for the first package of the communication. Also, it sets the signals Load and flag_load, as it is done in the start and ready interruption functions and increases the package counter to have the correct address.

  As the number of inputs for the test is limited, and the desire is to try how many cycles can be carried without errors by the testbed, it is necessary to assure that the input data that is fed is always controlled. For this reason, in this function it checked whether or not the counter has reached the maximum number of inputs introduced in the memory, and in the case that it has, the counter is reset. The same happens with the output counter, as the memory is not unlimited.

In order to assure that the end_trasnmission interruption function is executed only once each time, it is used a flag, as it was done in the others. This signal is then reset in the main function.

Once all the devices have been configured and initialized, it is necessary to set an initial value for the output GPIOs that are used in the communication process as flags in order to assure that they start with the desired state. This signals are load, flag_load and flag_store. After these code lines, the system is ready to be used for the testbed.

As it was explained, the interrupt functions are the ones in charge of setting the variables in charge of the communication process, but then it is the main routine the one in charge of resetting them. As it is not desired to use more interruptions and since the main function has no other duty to do, the way of programming this reset is by checking constantly the value of this variables. In other words, it is used a while loop whose condition is always true to have this main function active all the time.

In this while loop, the value of load, flag_load and flag_store is constantly being checked. When they are set in their respective interruption function, the if condition associated to their value is activated, and the flags used to assure that the interruptions are executed only once, and the signals are reset.

This code is designed for the situation in which the number of inputs and outputs is not relevant, which means that it works exactly in the same way independently of the case. The only thing that must be considered is the number of inputs that are stored in the memory. When the number input packages increases, the number of cycles that can be done with a quantity of inputs decreases, and for this reason more information is needed if the user wants the test to be varied.

3.4.9. Connections and constrain file

Once the designs of both boards have been done and the HDL wrappers have been created and are ready to be synthesized and implemented, it is necessary first to create the constrains of the project. This constrain files is the way of communicating Vivado the connections between the external signals of the VHDL and the hardware of the FPGA. In reality, what it is being done is connecting those signals to the pins of the microprocessor which are, in turn, connected to the components.

Once the file has been added to the project, the way of creating a constrain is done through two statements:

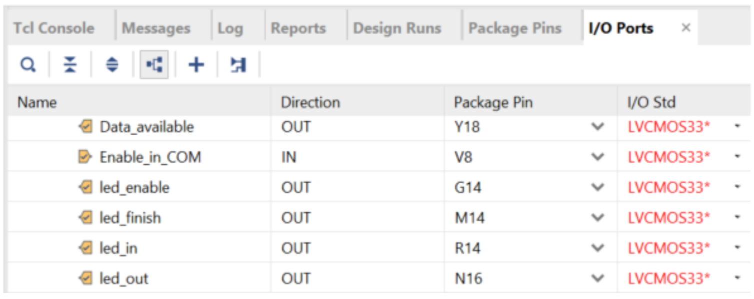
```
set_property PACKAGE_PIN R14 [get_ports {name_signal}]
set_property IOSTANDARD LVCMOS33 [get_ports { name_signal ]}]
```

The first one is used to connect the signal with the pin of the microprocessor. This name is found in the schematics of the board. (6) This document contains all the connections between the components and the microprocessor. The way of proceeding is searching the element desired, for example a switch, and seeing the tag used for that pin. After that, the name has to be searched in the bank of pins of the microprocessor, where it will be found the name that it is being looked for.

The second statement is the one used to declare what type of pin is used, in this case is a LVCMOS and its supply voltage, 3.3V.

One thing that must be taken into account is that the name of the signal is the tag used when it is externalized in the block design, and not the name in the VHDL source. Moreover, this name has to be exactly the same, all the capital letters have to be respect as well as all the signs.

These lines don't have to be written directly by the designer since there are two ways of automatically creating these constrain file. The first one is done through the I/O planning tool of Vivado Design Suite that is shown in the following figure. With this window, it can be easily assigned a pin to every signal and its type. Once this is done, the software is in charge of creating the file that will be saved and attached to the project.



Name	Direction	Package Pin	I/O Std
Data_available	OUT	Y18	LVCMOS33*
Enable_in_COM	IN	V8	LVCMOS33*
led_enable	OUT	G14	LVCMOS33*
led_finish	OUT	M14	LVCMOS33*
led_in	OUT	R14	LVCMOS33*
led_out	OUT	N16	LVCMOS33*

Figure 46. I/O planning tool window

The second option is not always available. Some boards have their own XDC file already created with all the lines written and commented, so it is ready to be used. The designer only has to add the file to the project, choose the line with the desired pin, uncomment it and write the exact name of the signal. The Zynq-z2 board has this constrain file available in the manufacturer's webpage, which makes this process faster and easier.

The last sections have been used to describe the modules of the design and the signals used for transferring the information. The following step is to assign them to the different components of the hardware, these are the LEDs, switches, buttons and connector.

First, both boards use signals to verify the state of the processes, and they were thought to be used as illuminated marks. The Pynq-z2 has four available LEDs and two RGB, or three state, LEDs, and this sums ten available ports. Each board needs only six of them, and although the RGB can be used for different signals, it makes sense to assign only one to each LED.

The switches are used only in the monitor board. As it was explained, the start and reset have to be controlled with this kind of hardware. The other possibility would be to use the push-buttons, however the switches seem a better option as they are more stable. As there are two available in the FPGA, there is no need of using the pushbuttons.

The exact assignment of these signals to the hardware can be seen in the following figure.

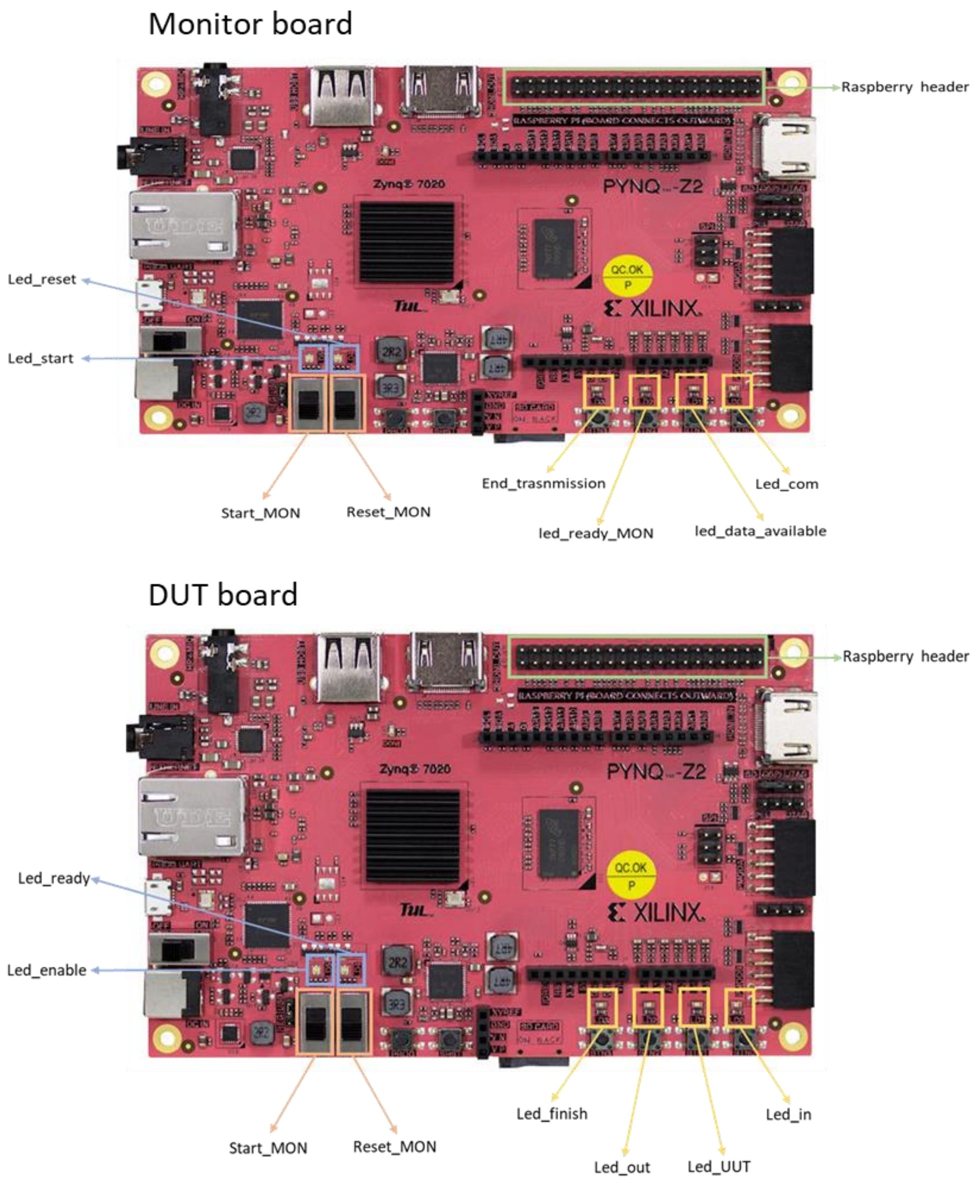


Figure 47. Signals assigned to LEDs and switches

Using the datasheet as explained, the correspondent pins for the components used are shown in the following table. As it has been said, the RGB LED has three signals, red blue and green, and it has been chosen the blue for both boards. Also, another thing to know is that all of the pins are LVCMOS33.

Component	LED0	LED1	LED2	LED3	RGB4	RGB5	Switch0	Switch1
Package_pin	R14	P14	N16	M14	L15	G14	M20	M19

Table 6. Correspondence between components and pin

The last assignment that must be done is the one for the signals used for the communication between the boards. It was decided that it would only be used the Raspberry header, and there are twenty-eight available pins for nineteen signals. The assignment strategy is to group the vectors as closed together as possible. Also, in order to help the process of connecting the boards with jumper, it has been decided the boards must face each other, allowing the connectors to be as close as possible.

As the length of the jumpers is limited, and it is desired to avoid connections that lead to the cables being tangled up, the task of locating the signals on the pins has to be done carefully. The way of proceeding is to assign all the signals from one board, and from this point, the other FPGAs pins have to be given to the signals corresponding to the ones in front of them.

In the following page, it is shown the Raspberry header for both the boards, and the signals that have been assigned to each pin.

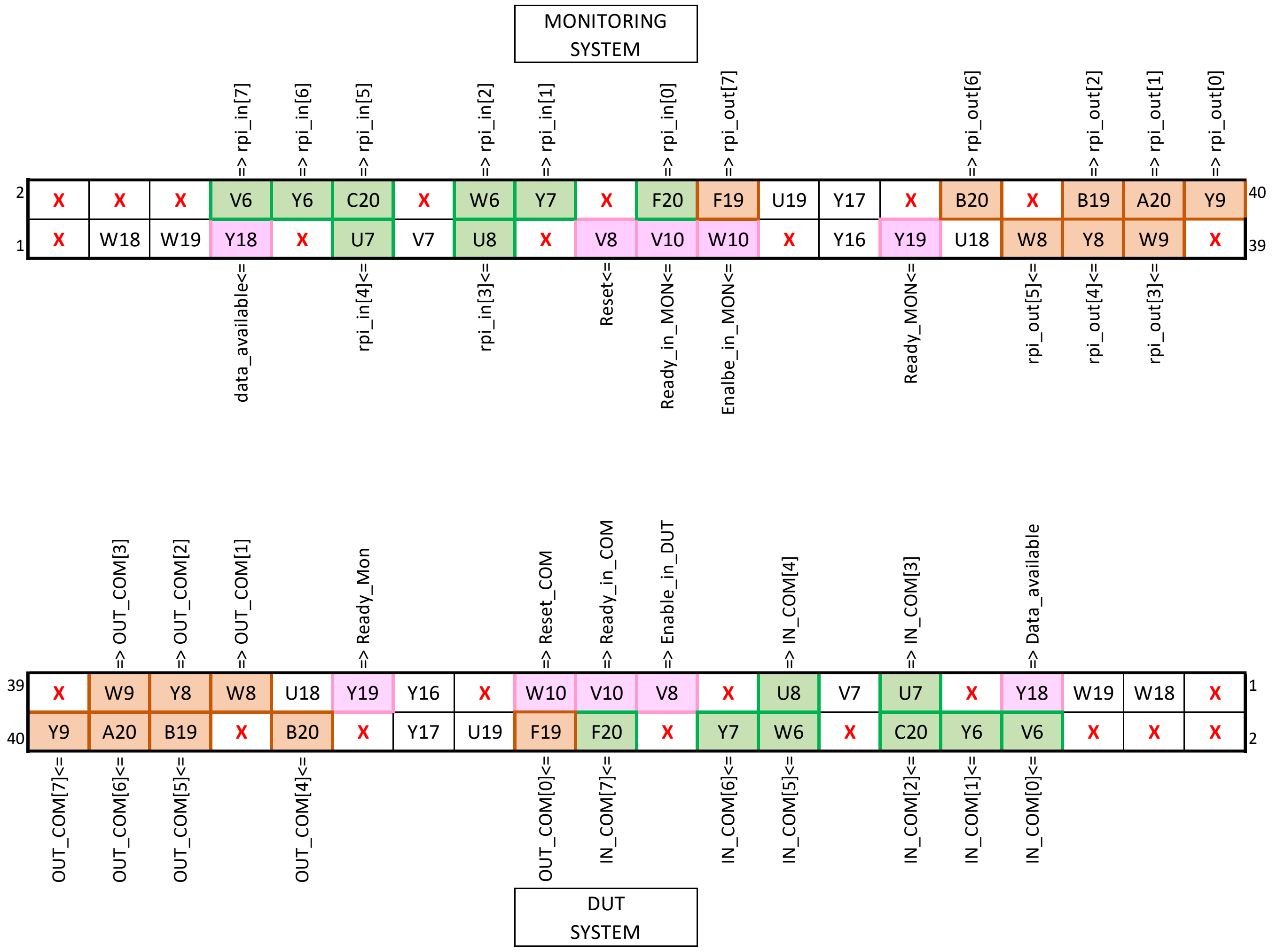


Figure 49.RPi header connections between boards

Chapter 4

Results

This chapter is dedicated to explaining the tests that the system has undergone in order to check its operational performance in general and with respect to the frequency, as well as the analysis of the results obtained and the possible causes for these. Finally, a brief conclusion is drawn from the final system.

4.1. Setup of the test

Before running any test in the system to check its performance, it is necessary first to prepare these tests. Some of the tasks for the setup of the test includes preparing the boards, connecting them together and choosing the desired UUT. Also, the decision of the type of tests that are carried out have to be made.

4.2. Board setup

First of all, the two boards have independent designs and implementations, but as it has been explained several times, they must communicate with each other, and for this purpose they must be connected. As it was explained in the last section, the only part of the hardware that has to be linked to the other board, are the chosen pins of the Raspberry Pi header. So, the first decision is to choose the method used to connect the two boards.

The first option is by using jumpers. This object is a wire which has in its ends an adaptor to ease the connection with the pins. As this connector fits with the pin of the RPi header, it is coupled and makes it difficult for the cable to come loose. This is a great advantage, as the boards may be moved from one place to another by the users, and the connections must be well secure. The other advantage of this solution is its cost. These jumpers are a cheap solution and they are easily found in the market.

This jumpers can be independent wires which have to be connected one by one by the user, or a string of forty cables connected to a header that fits the one that is in the Pynq-z2. This last solution is better for the user due to two reasons: first, it is safer as there is no possibility of wrongly connecting two pins that don’t belong together, and second, it avoids the tiring task of connecting each cable to the pin.

There are two main disadvantages when using jumpers. First of all, the length of this wires is typically small, which forces the two boards to be close one to the other. This difficult the task of fulfilling one of the requirements of the testbed, which is the capacity to have the two boards away from each other in order to test the DUT in certain environments. The second one is the quality of this jumpers. They are designed with no protections to the noise that can be generated when they are surrounded by other jumpers or components, and this can lead to a situation in which the signals are disturbed, and this may even get to a point in which the testbed fails. This situation is worsened as the length of the jumpers increases, there is more surface to be disturbed.

One of the solutions is to use wires that are more expensive but that have isolation and can minimize the noise generated by the other cables. This wires are, as expected, more expensive than the normal jumpers, and for this project a cheap solution is being searched. The other solution is to use another wire connected to ground rolled up around the jumper that connects the pins of the two boards. Although this last solution is economically possible, as it would only require some more wires which are cheap, in increases the difficulty of connecting all the pins together.

For all of this reasons, the first attempt is using simple jumpers connecting the pins of the two boards and without taking into account their length or the possible noise that may be created and that could get in the system. The following image shows the setup of the two boards when they are connected.

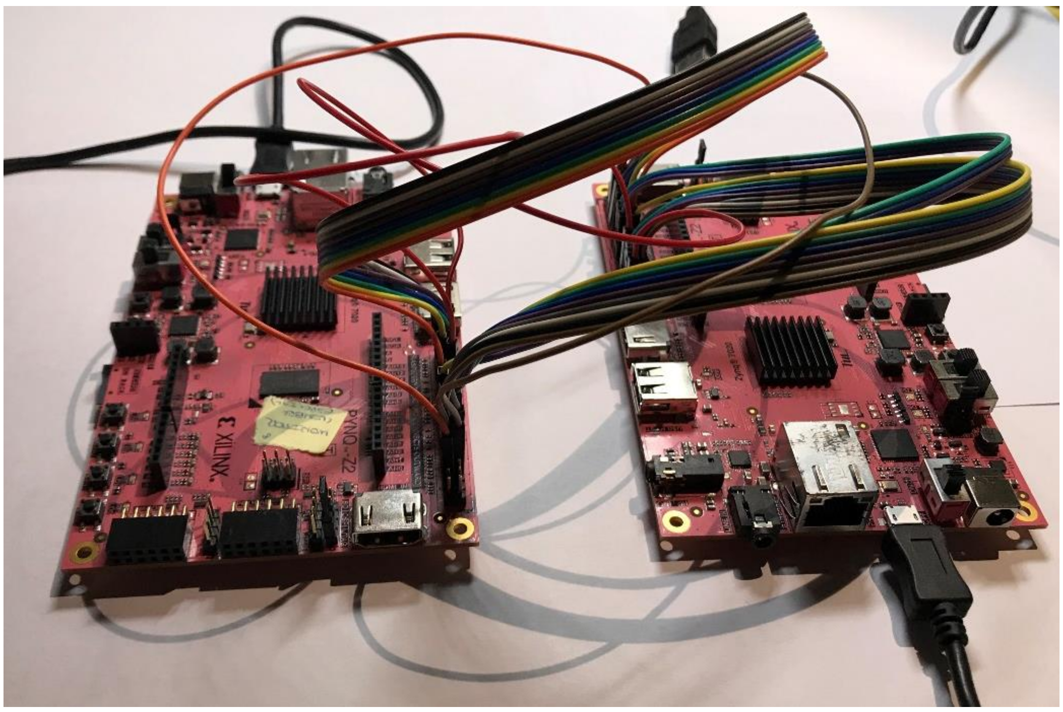


Figure 50. Boards setup

4.2.1. Unit Under Test

This block is the most important one of the design, since it is the circuit that the designer or engineer wants to test. In order to be a useful system, this module has to be easily changeable by any other.

At this point of the project, it is time to decide which module is going to be implemented here in order to run the tests and get the results of the system. At this point there are two options.

The first one is to find just one circuit that has the capability of adapting to different situations, which means that must be able of changing the number of inputs and outputs easily. The shift register used in the past for this project meets this criteria. The advantages of this solution is that it is already created and designed to fit in as UUT. Also, as generics where used to define the size of the vectors and the conditions of the code, it can be adapted to any number of inputs and outputs. The disadvantage is that it has not been tested in any other platform, and therefore it is not possible to compare its performance in this system with respect to other implementations. Also, the number of processes of the module are always the same, one process, and cannot be changed. This circuit, in certain situations, can be too simple.

The other option is to find more than one circuit that can be used to test different situations and analyze the performance of the system. These circuits are called benchmarks and are used for the evaluation and performance of different testing approaches. There is a big amount of different benchmarks available on the internet, but for this project the search has been narrowed to the ones of the ITC’99.

The ITC99 benchmarks are a set of circuits from many companies of the electronics business that are meant to be used for experimentation of Design for Testability (DFT) and Automatic Test Pattern Generation (ATPG). (18) The DFT is used to increase a system’s testability, resulting in improving quality while reducing time to market and test costs. It is necessary to remark that the testability is defined with respect to a fault, which is testable if there exists a procedure to expose it, and it is implementable with a reasonable cost. (19)

Some of the designs of this ITC99 benchmarks are from business companies and other are from different universities. More specifically, some of the benchmarks that are being considered are from Politecnico di Torino, which are the I99T<1-22> or Torino benchmarks.

Torino benchmarks consist of twenty-two RTL lever sequential benchmarks, ranging from 45 gates and 5 FFs (Flip-Flops) to 98.000 gates and 6.600FFs, where the larger designs are compositions of the smaller ones. The advantage of this circuits is that they are simple in the sense of being simple clock designs without any internal memory or tristate busses. (18) In case it is chosen this option for the UUT of the project, it is necessary to select some of the circuits from the twenty-two available, taking into account that they must be varied in terms of number of inputs and outputs and of processes.

These three are valid options: b05, b07 and b13. The first circuit has three inputs and thirty-six outputs, which means that needs only one package to transfer all the information from the monitoring system to the DUT and five for the opposite direction. The other important characteristic is the number of processes, which in this case is 3.

The b07 circuit has four inputs and six outputs, which means that only needs one package for the input and output communication. This benchmark would test the simplest cycle communication available. The number of processes is of one.

The last circuit is the b13. It has twelve inputs, which are transferred in two packages, and ten outputs, that also need of two lots to transfer the complete information back to the DUT. The number of processes in this case if five.

The reason of choosing these three circuits is due to two reasons. First, as it can be seen they gather together the three different timing scenarios of the communication. Benchmark b13 is the worst case, since both for the input and the output are required more than one package, b07 is the best for the opposite reason. In the middle is b05, which for one of the communication uses just a package, but for the other uses more.

The other reason is related with the processes, as all of the circuits have different processes, and for from one to five.

The advantages of choosing benchmarks as UUT is that they are already tested and can be used to check how is the performance of the system. However, the implementation of this circuits requires more time than the shift register, since it is necessary to adapt the communication module to this units.

The advantage of having the shift register already tested and used as UUT before, makes it easier to, as a start, choose this circuit for the implementation. In the future it is possible to change it for any of the ones explained to do more tests.

4.2.2.

Once the UUT has been chosen, it is possible to design the type of test that is desired. In order to check how the performance of the system without depending on the communication, it is necessary to develop at least two different tests. The reason is that the performance of the platform depends directly from the performance of the communication between the two boards, and this, in turn, depends on the number of inputs and outputs of the UUT.

As it was explained in the sections related to the communication modules, the complexity of the process of transferring data increases as the number of packages required does. And for this reason, the easiest communication is when there is only one input and one output package. This is the first situation that is tested in the platform, and the number of inputs and outputs is the same, eight bits. The second situation that is tested is when it is required two packages for both the sending and the receiving information processes.

The idea is to first check the performance of the platform itself, and then to analyze the difference of this performance depending on the number of inputs and outputs. It is expected a difference in the performances, as the communication protocol is more complicated when more than one package is required.

In the following tables it is shown the inputs that are fed to the test in each situation, one and two input/output packages. These inputs have been selected to be as different as possible.

Also, it is shown the outputs that should come out from the UUT after shifting all the bits one position to the left. Although the idea of these tests is not to check that the UUT works correctly, it is important to verify that no information is lost and the communication works as it should, and this is done by analyzing the results that are stored in the memory.

Input	Output
0x04	0X08
0x02	0X04
0x0A	0X14
0X01	0X02
0X08	0X10
0X03	0X06
0X0B	0X16
0X05	0X0A
0X07	0X0C
0X13	0X26
0X09	0X12
0XF1	0XE2

Input		Output	
0x02	0x04	0x04	0x08
0x01	0x0A	0x02	0x14
0x03	0x08	0x06	0x10
0x05	0x0B	0x0A	0x16
0x13	0x07	0x26	0x0E
0xF1	0x09	0xE2	0x12
0x06	0xE2	0x0D	0xC4
0x25	0x43	0x4A	0x86
0xB6	0x2F	0x6C	0x5E
0x87	0x1A	0x0E	0x34
0x79	0x35	0xF2	0x6A
0xBA	0x97	0x75	0x2E

Table 7. Inputs and expected outputs

.

4.3. Results

After having developed the block design, the software application and having setup the test, it is the moment to start testing the performance of the designed system. For this purpose, the boards must be connected with the jumpers in the corresponding pins, and the selectors of the programming and power mode In the correct position.

When the boards are connected to the computer via USB, they can be programmed from SDK via JTAG. Both the monitor and the DUT must be programmed with the bitstream to boot the processing system, however only the monitor has software application that must be uploaded after booting the PS.

The idea of the test is to analyze the performance of the testbed with respect to the frequency of the clocked used for the PS and the PL. For this purpose, it is necessary to modify this parameter and examine the behavior of the system, this is, the communications and the results of the test. Every time a test is launched, the system starts a cycle of loading the inputs, sending the information to the UUT, carrying out the test, returning the information to the monitor and storing it in the memory. Once it is over, another cycle starts immediately. Theoretically, below a certain frequency, this loop can go an unlimitedly number of times, as the system has enough time in each clock cycle to carry out all the tasks without missing any part of it. Get a frequency is reached, there is a chance that the system may not be able to finish all the tasks, which can lead to an error that stops the execution.

However, there is a lower limit in the frequency imposed by the design. As it was explained in the chapter dedicated to the board and to the designs, the processing system can fabricate a clock withing the range of 0.3 and 200MHz. Although 0.3MHZ may sound like a low frequency, this system is a low-cost implementation, and the communication protocol robustness may not be enough. For this reason, it is expected that the real functioning of the system is not the theoretical, but that there will be errors, and the objective is to determine at which frequencies those errors are represent a minimum percentage with respect to the cycles successfully achieved.

The problem of this new objective is that every time the system misses one important task, the process is then blocked, and the cycles stop. In order to start a new try, it is necessary to reset the system and reprogram the monitor board, which takes a lot of time. The solution would be to implement an option of re-starting a cycle whenever it is blocked for longer than expected. However, this was discovered once the tests were being developed.

The best way of getting the most accurate results is by repeating the process as many times as it is possible, however, it was just explained that each time there is an error the system has to be re-started again manually. This situation limits the number of times that the test can be carried out, as the time that implies each one is too high. It cannot be forgotten that each test has to be carried first for the situation of eight input/output and some frequencies and then the same for sixteen inputs/outputs.

The final decision of how to proceed with the tests is the following one. First, for each frequency, the number of repetitions is of thirty. Each time the system fails, it is annotated the number of cycles than have been carried out before stopping the system and the reason for the failure. However, they might be a frequency for which the system does stop at a really high number of cycles. As for each cycle sixteen signals are transmitted, eight from the monitor to de DUT and another eight in the other direction, for over a number of cycles the amount of bits transmitted is big enough to stop. This quantity is set to thirty thousand of cycles, as the quantity of bits is almost half a million.

This processes is repeated for eight different frequencies that go from 0.3 to 100MHz. This range is wide enough to assure that the points that are being looking for will be there. Those points are first a frequency in which it can be said that the system works with minimal number of errors. The second one is the frequency in which it can be said that the system does not work anymore. It is expected to be found in the middle of these two a zone of frequencies in which the behavior is still good but deteriorates as the frequency increases.

The process of changing the frequency also requires time, as it has to be modified in the block design of both projects by changing the configuration of the processing system block. After this, the bitstream has to be created again and exported to SDK. After this, it is possible to program and test again the boards.

Trying the system for eight different frequencies, for two scenarios of number of input and outputs gives as a result a great amount of tests to be carried out. As said, there are two results that are annotated: the number of cycles successfully completed and, in case the system stops due to an error, the process in which the system got blocked. There are three types of errors that can occur, and their tags are: in, out and re-start.

- *In*. This error takes place when the process of sending information from the monitor to de DUT is not achieved completely, the reason is that one of the protocol signals get lost without the other board noticing its activation, and therefore the process gets blocked waiting for the enable or the ready.
- *Out*. It is produced by the block of the receiving information process, in other words, when the output of the test is transmitted from the DUT to the monitor board. The reason for this error is the same as in the previous situation.
- *Re-start*. When the cycle is finished, the application software in interrupted by a signal controlled by the monitor system. However, if the duration of this signal is not long enough, or it does not get activated, the interruption routine is not executed and therefore the system gets blocked.

Whenever a test is stopped, it is necessary to figure the reason. The way of doing it is by checking the state of the signals that are connected to LEDs in the block design. In the following table it is summarized the three situations with the value of the LEDs. It is necessary to explain that in the monitor board, the LED one two and three are used for signals that last one clock cycle, and for this reason they are always off.

		IN	OUT	RE-START
Monitor	LED0	✓	✓	✓
	LED1			
	LED2			
	LED3			
DUT	LED0		✓	✓
	LED1	✓	✓	✓
	LED2	✓	✓	✓
	LED3	✓		✓

Table 8. State of LEDs during an error

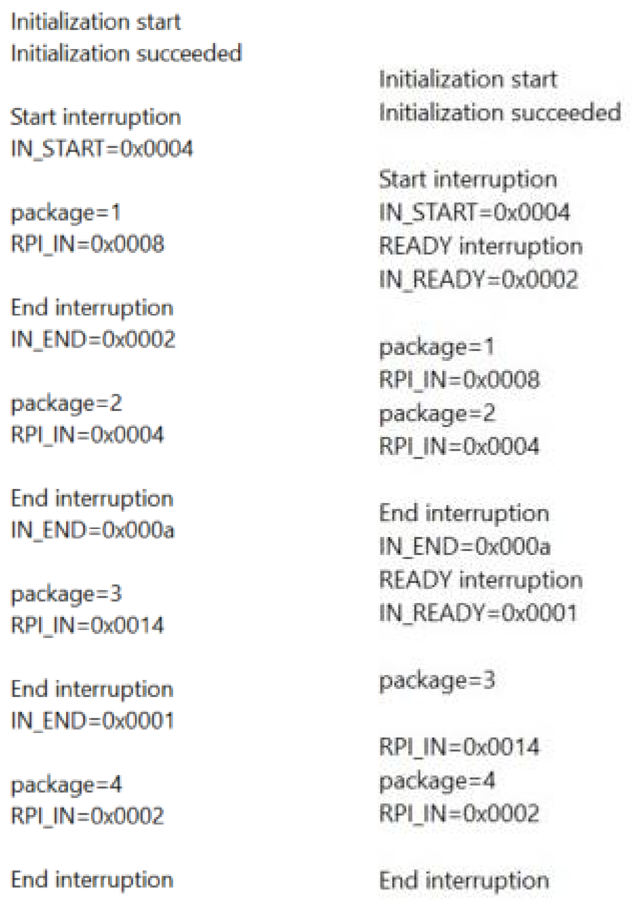
As it can be seen, the three situations can be easily distinguishable, as there is always one LED different. However, the situation in which the system fails in the re-start process all the LEDs that are on for more than one clock cycle are activated. There is another way of differentiating this three situations, and it is using the SDK terminal. This tool allows the user to print information of the status of the process by adding the correspondent instructions in the software application.

For this project, it is useful to print the information related to the number of cycles that have successfully finished and also to notify the interruptions that are executed. This way, when the process is blocked, it is possible to analyze the last task executed by the system and conclude what is the error.

In the following figure, it can be seen the SDK terminal for both situations: eight and sixteen inputs/outputs. In the case of eight I/O, the only interruptions that are executed are the start and re-start cycle and store in the memory. When the start button is pushed, the start interruption is executed, and the first inputs are loaded in the only package that is required, the SDK terminals prints it with the name of IN_START, which is not the signal. When the DUT is sending the outputs of the test, the store interruption is executed and stores the input signals in the memory, and it is printed with the name of RPI_IN. At this point, the process has almost finished, the information has been transmitted and the number of cycles successfully finished is printed with the name of package. When the package has arrived, the re-start process starts with the end_trasmission interruption. The new inputs are loaded in the signal, and it is shown in the terminal with the name of IN_END. Then the rest of the process is repeated continuously.

The case for sixteen I/O looks like the one of eight, but with one more interruption, which is used in the sending information to the DUT. It is the one in charge of sending all the packages except for the first one, which is managed by the start o re-start interruptions. This way, when the start switch is activated, the first package is sent and it is shown in the terminal with

IN_START, and the second one is sent in the ready interruption and printed with IN_READY. The store routine is executed two times in order to save the two packages of the communication. And after this, the re-start interruption loads the first lot of the following cycle, starting the process again.



```
Initialization start
Initialization succeeded

Start interruption
IN_START=0x0004

package=1
RPI_IN=0x0008

End interruption
IN_END=0x0002

package=2
RPI_IN=0x0004

End interruption
IN_END=0x000a

package=3
RPI_IN=0x0014

End interruption
IN_END=0x0001

package=4
RPI_IN=0x0002

End interruption
```

```
Initialization start
Initialization succeeded

Start interruption
IN_START=0x0004
READY interruption
IN_READY=0x0002

package=1
RPI_IN=0x0008
package=2
RPI_IN=0x0004

End interruption
IN_END=0x000a
READY interruption
IN_READY=0x0001

package=3

RPI_IN=0x0014
package=4
RPI_IN=0x0002

End interruption
```

Figure 51. SDK terminal: On the left eight i/o on the right sixteen i/o

In the right part of the figure, it can be seen that for each cycle, there are two packages counted. For this reason, it is necessary to divide by two this number to calculate the real number of cycles successfully finished.

4.3.1. Case one: eight I/O

The first test to be carried out is the one that corresponds to eight inputs and outputs. In the following page it is shown the collected data from all the frequencies, showing the number of cycles and the errors that caused the system to be stopped.

	0.3MHz	Error type	1MHz	Error type	2MHz	Error type	5MHz	Error type	10MHz	Error type	25MHz	Error type	50MHz	Error type	100MHz	Error type
1	30000	none	23579	in	1071	in	1119	in	825	in	128	out	13	out	1	-
2	30000	none	12892	in	4379	out	2511	re-start	722	out	129	re-start	92	out	1	-
3	30000	none	10797	in	13513	in	527	in	582	out	31	out	55	out	1	-
4	26922	in	18333	in	6315	out	5965	in	470	out	12	out	58	out	1	-
5	30000	none	14517	in	1896	in	1941	in	769	in	21	out	68	out	1	-
6	30000	none	23179	in	21592	in	1860	out	231	out	135	in	22	out	1	-
7	30000	none	25270	out	14843	re-start	2093	in	371	out	158	out	29	out	1	-
8	14854	in	20979	in	5383	in	2365	in	833	out	82	out	22	out	1	-
9	15853	out	27153	in	5093	in	1070	re-start	497	re-start	19	out	7	out	1	-
10	30000	none	10632	out	9247	in	823	in	254	in	12	out	31	re-start	1	-
11	23560	in	18245	in	8551	in	1087	out	1486	in	67	out	35	re-start	1	-
12	21364	in	10382	re-start	14253	out	711	in	1250	out	75	in	136	out	1	-
13	16230	out	11463	in	6327	in	1039	out	1401	out	105	out	91	out	1	-
14	30000	none	14455	in	8336	in	1186	in	2413	in	21	out	15	out	1	-
15	30000	none	23457	in	29067	out	909	in	2099	out	8	out	15	out	1	-
16	22446	in	30000	in	22428	in	3573	re-start	1367	out	108	out	75	in	1	-
17	30000	none	12031	in	2730	in	1411	in	152	out	102	re-start	105	out	1	-
18	30000	none	11187	out	4736	out	2232	in	675	in	107	out	21	out	1	-
19	23943	in	15436	in	8094	in	1685	in	425	out	64	out	173	out	1	-
20	29685	in	20740	in	9971	in	2215	in	931	in	34	out	70	out	1	-
21	18573	in	8402	in	21495	in	2357	out	674	out	28	out	37	re-start	1	-
22	30000	none	8679	in	9421	in	914	in	1538	re-start	49	out	112	out	1	-
23	28116	in	14862	out	2259	out	1562	re-start	861	out	100	in	20	re-start	1	-
24	30000	none	15567	out	4862	out	1947	in	119	in	12	out	49	out	1	-
25	30000	none	15617	re-start	1605	in	1209	in	257	in	13	out	104	out	1	-
26	30000	none	16946	out	3986	out	1705	out	915	out	95	in	81	out	1	-
27	30000	none	18095	in	14276	re-start	1922	in	635	out	7	out	38	in	1	-
28	30000	none	29134	in	6254	in	1673	out	254	in	14	out	11	out	1	-
29	30000	none	9919	in	10883	in	536	in	874	out	20	out	60	out	1	-
30	30000	none	15265	in	2865	in	491	in	521	in	38	re-start	33	re-start	1	-
Cycle average	27052		16907		9191		1688		813		60		56		1	
bit error	88		232		240		240		240		240		240		240	
bit trasnsmitted	12984736		8115408		4411696		810208		390416		28704		26848		240	
Bit error rate	6,77719E-06		2,85876E-05		5,44008E-05		0,00029622		0,000614729		0,008361204		0,008939213		1	

Table 9. Test results for eight I/O

The first conclusion that can be drawn from the table with the results is that the functioning with respect to the frequency is the expected one. The number of cycles that the system can complete decreases as this parameter increases. This results particularly visible with the parameter of average cycles, which goes from over twenty-five thousands of rounds with low frequencies to almost none above fifty megahertz.

Analyzing this behavior, it is easy to understand the reason why the cycles decrease in this way. When the frequency increases, the period of the clock, which is inversely proportional, is decreased. As a consequence, the system has less time between each rising edge of the clock. However, the number of tasks that have to be executed each time does not change, and this means that the system is more likely to fail, as it may not complete all the duties of the process before it is executed again.

This behavior is especially dangerous during the communication process. As it has been explained, the two boards must be synchronized to exchange the data between them, and for this purpose two signals are used, the ready and the enable. However, if during an execution, the system has not enough time to set/reset them in the correct moment, this may lead to a block of the communication and consequently to an error.

There are two cases that deserve special attention. The first one is 0.3MHz. At this frequency, the system is capable of doing in average over twenty-seven thousands of cycles, which is translated in a great amount of bits exchanged. The reliability of the system for this number of input/outputs and frequency has as a consequence that it can be said that the system works as desired.

The opposite situation happens when the frequency is 100MHz. The system is not capable of sending any information to the DUT, and at this frequency the system does not work. However, this situation may not start at this point, but in a frequency between 50 and 100MHz. Furthermore, from 25MHz the systems behavior degrades and the cycle average decreases considerably.

The second observation that can be made about the results is related with the variability of the number of cycles successfully finished for the same value of the frequency. For example, for the case of 10MHz, the maximum amount of cycles is around two thousands and the minimum is less than one hundred and fifty rounds. But the worst case is for 1MHz, in which the difference between the two values is greater than twenty-five thousands of cycles. The reason for this variability has to do, in part, with the way of facing these tests. As it was said, the system is started, and as soon as it fails, the process gets blocked and it has to be started again. There is a certain probability of having an error at one frequency, but this fault can happen at the beginning of the test or after a big amount of rounds. As this probably increases, the variability is the results decreases and the values are more alike. Also, it has to be taken into account that this is a low-cost system, which means that the investment in materials such as the jumpers, increase the probability of errors for a certain frequency.

The way of improving these results would be implementing in the system the option to continue with a new cycle when an error occurs, counting the number of faults that occur in a certain number of cycles. When these type of tests are carried out, it is possible to analyze the percentage of errors with a ratio which is called Bit Error Rate (BER).

The BER is a measure of the number of bit errors that occur in a given number of bit transmissions. It is a measure of the quality of the quality of the communication, as it examines the quality of the transmitter, the receiver, the transmission path and the environment. (20) The way of calculating this ratio is by applying a continuous bit stream to the system and counting the bit errors.

This ratio cannot be used directly for this system as it doesn't have a bit stream applied continuously. However, it can be adapted to this situation accepting that every time the system stops and re-starts, the information is fed again. This way, whenever the system is blocked, it is considered an error in the transmission, and all the successful cycles are the transmitted bits. This ratio is calculated and shown in the table where the results are. The BER with respect to the frequency is shown in the following graphic. It must be noted that the abscissa is in logarithmic scale to represent the frequency.

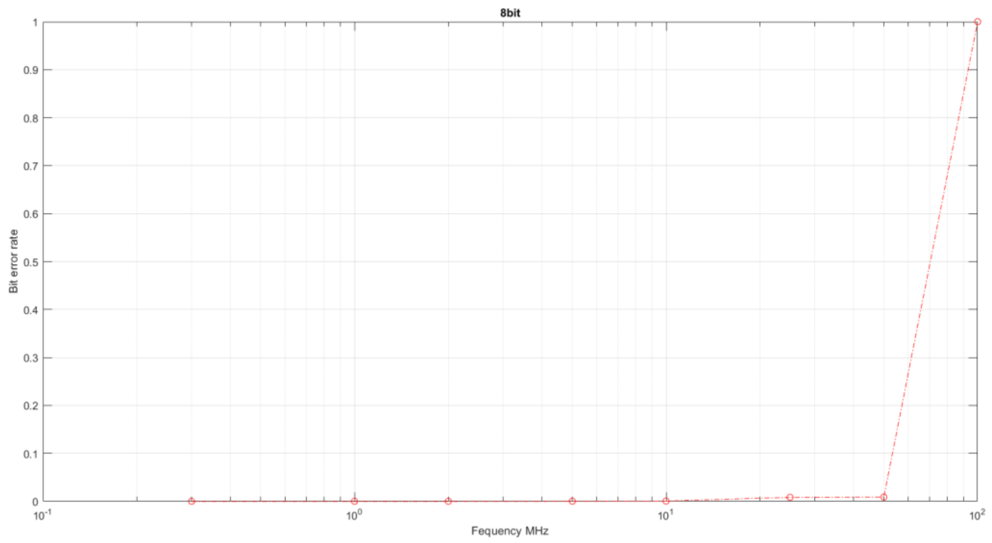


Figure 52. BER with respect to the frequency for 8I/O

The BER shows what it has been explained, the system is more reliable as the frequency decreases. For frequencies below fifty megahertz, the ratio is nearly zero. However, this situation changes when the parameter goes above the mentioned value, until it reaches its maximum amount at around one hundred megahertz. Whenever the BER is one, it means that no information is transmitted without errors, in other words, at this point the system is not working.

It can be noted that the increase in the value of the BER occurs significantly in a shot space of frequencies. Under this values, the system is relatively reliable, however, in this range the system performance deteriorates rapidly, and it is dangerous to work in this zone of frequencies.

This ratio can be employed to determine at which frequencies the system can be used depending on the needs of the user and the tests that have to be carried out for the system. Since not all the needs are the same, it is not possible to decide whether or not the testbed works.

The last observation is related with the nature of the errors of the system. It can be noticed that for low frequencies, the most part of the faults of the system are due to an error of communication in the process of sending information from the monitor to the DUT, however the end-transmission interruption is executed, as it can be seen in the SDK terminal. So, the conclusion is that the issue must be that the DUT does not get in time the signal that warns that there is available information in the channel, and therefore the system is blocked waiting for a signal that has already been sent by the monitor.

However, the behavior changes for higher frequencies, where the most common error is due to the other communication process, the one that sends the results of the test to the monitor system. In this situation, the system is never interrupted to execute the re-start function, and the reason is that the communication process is never finished, as the monitor never receives the package with the information. The problem is the same as the other case, the signal that notifies that the lot is available is not noticed by the monitor, and the system is blocked waiting for it.

The re-start issue is the one that appears the least compared with the other two. The highest incidence is at 50MHz, but there are not enough results to assure that this result is repeated for all the high frequencies. This fault appears when the communication process has finished, and the end_transmission should be set. This signal is the one that triggers the re-start interruption; however, the system does not have enough time to set the signal, and therefore the interruption never executed. The system gets blocked waiting for a new cycle that cannot start.

4.3.2. Case two: sixteen I/O

The other case studied is the one in which the number of inputs and outputs of the system is sixteen. In this situation, the system must use two packages for both direction of the communication to send all the data. As it was explained in the chapter related to the communication process, this situation is the worst case in terms of time, as it requires more executions and interruptions to finish a cycle. It is expected that this condition will affect the performance of the system.

In the following table, it is shown the results of the tests carried out. The information contained is the same as in the other case, the number of successful cycles for each try in a determined frequency, the cause of the error and then the BER ratio and the average cycles per frequency value.

	0.3MHz	Error type	1MHz	Error type	2MHz	Error type	5MHz	Error type	10MHz	Error type	25MHz	Error type	50MHz	Error type	100MHz	Error type
1	2986	in	370	in	358	in	587	re-start	148	out	13	out	18	out	1	-
2	901	out	2700	re-start	276	re-start	544	out	118	out	140	out	33	out	1	-
3	5090	in	1572	in	88	in	167	out	357	out	113	out	79	re-start	1	-
4	1612	in	707	in	570	in	154	out	149	out	11	out	3	out	1	-
5	426	in	974	in	230	in	930	out	427	out	126	out	1	out	1	-
6	3139	in	1402	re-start	173	out	542	out	52	in	121	in	23	out	1	-
7	2625	out	414	in	432	out	197	out	765	out	12	out	12	out	1	-
8	9418	in	1284	in	717	out	24	out	106	out	127	out	59	out	1	-
9	3941	in	336	re-start	109	out	863	out	10	out	113	out	32	out	1	-
10	4324	in	440	in	1009	in	213	out	1090	out	141	out	27	out	1	-
11	25618	in	1268	out	1696	in	729	out	525	out	14	out	7	out	1	-
12	7072	out	424	in	110	out	313	out	134	out	176	out	9	out	1	-
13	1696	in	841	out	1020	out	187	out	293	out	125	out	14	out	1	-
14	2766	in	196	in	418	in	124	in	574	out	126	out	41	re-start	1	-
15	1074	in	2252	in	242	out	39	out	6	out	37	out	15	out	1	-
16	5616	re-start	1317	re-start	139	out	56	out	216	out	138	re-start	153	in	1	-
17	3586	in	320	in	311	out	138	out	113	out	77	re-start	23	out	1	-
18	8792	out	5716	in	794	in	212	out	393	out	155	out	42	in	1	-
19	1470	in	204	in	1177	out	72	out	28	out	13	out	45	out	1	-
20	870	in	226	in	278	in	91	out	331	out	107	out	47	out	1	-
21	1754	in	924	in	1134	out	296	out	188	out	119	out	28	re-start	1	-
22	12320	in	424	in	113	out	272	out	14	out	128	out	6	out	1	-
23	2974	out	488	in	172	out	182	out	150	out	110	out	8	out	1	-
24	1568	re-start	2720	in	132	in	138	out	38	in	117	out	13	out	1	-
25	2780	re-start	582	in	2119	in	615	out	12	out	17	out	100	out	1	-
26	4982	out	277	out	198	out	235	out	256	out	144	out	60	out	1	-
27	8750	re-start	336	in	454	out	120	re-start	352	out	136	out	15	in	1	-
28	1626	in	2894	out	603	in	1426	out	370	out	71	out	6	out	1	-
29	878	in	690	in	747	in	204	re-start	88	re-start	13	out	10	out	1	-
30	2626	in	1512	in	106	in	54	out	160	out	28	re-start	18	re-start	1	-
	4443		1127		531		324		249		92		32		1	
Bit error	480		480		480		480		480		480		480		480	
Bit trasnsmitted	2132480		540960		254800		155584		119408		44288		15152		480	
Bit error rate	0,00022509		0,000887311		0,00188383		0,00308515		0,004019831		0,01083815		0,031678986		1	

Table 10. Test results for sixteen I/O

As it was expected the performance of the system in this case is worst that the case of eight inputs and outputs, however it is necessary to analyze this scenario independently from the other one.

The first conclusion from the results presented in the table is that the system works as expected in this situation as well. This means that as the frequency value increases, the average number of cycles successfully completed decreases. As it was explained, this agrees with the explanation given about how the duration of the clock cycle is lessen. This situation complicates the execution of all of the tasks in each board and increases the probability of missing a duty undone, being specially dangerous when the communication is being carried out between the boards.

Analyzing the average cycles of each frequency, the first conclusion is that the system's performance is clearly deteriorated when the frequency is above one megahertz, not being able to successfully finish more than five hundred cycles each time the system is tested. More important is no analyze where the system stops working. When the frequency value is higher than ten megahertz, the number of cycles completed is below one hundred. It can be concluded that for these values, the system barely can function.

The best way to analyze deeply the performance of the system is by using the BER ratio, which compares the bits error with the amount of bits correctly transferred. It is important to remark that this ratio is more accurate when the number of transmitted bits is bigger, as it is more complete. However, in this scenario of sixteen inputs and outputs, the amount of data transferred for each frequency is lower due to a worst functioning of the system. Also, it is reminded that the calculation of the BER requires from a continuous transfer of information, and the approximation made for this project is to accept that every time the system is blocked and initialized again, it counts as the same bit stream.

The ratio for each frequency is calculated and shown in the table that contains the results. In the following graphic this value is represented with respect to the frequency. This last parameter, which is located in x-axis, is represented in logarithmic scale, while the BER is not scaled.

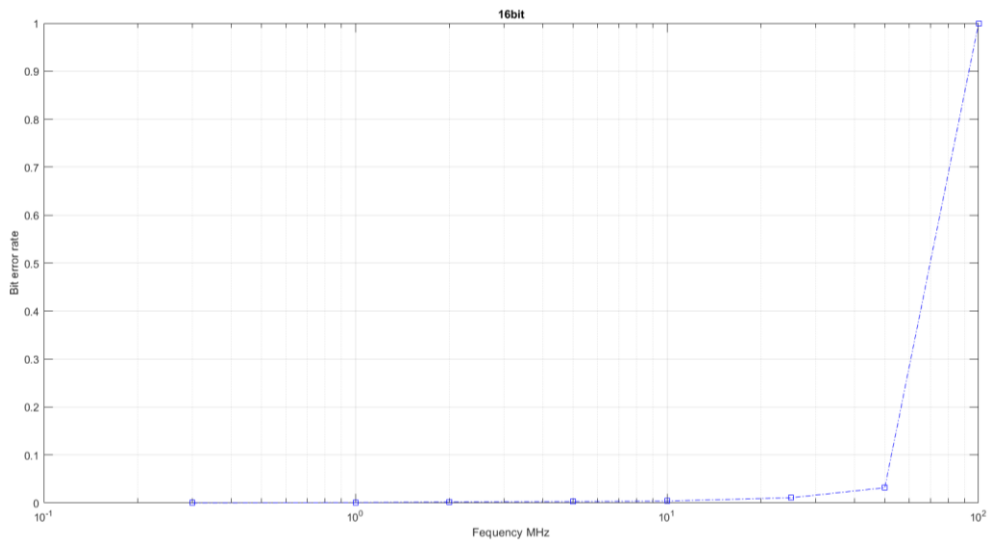


Figure 53. BER with respect to the frequency for 16 I/O

From the graphic, it is possible to see that for frequencies below ten megahertz, the ratio is nearly zero and the performance of the system is good. However, the average cycles that are completed without blocking the system variates from over five thousand to two hundred and fifty. It is the user the person that has to determine whether or not the amount of cycles is enough for the used given. When the frequency is increased above fifty megahertz, the BER starts increasing, getting to values in which is not possible to not take it into account. In other words, the performance is deteriorated.

As it happen with the other scenario, the worst case is produced when the frequency is near one hundred megahertz. In this situation, the system is not able to send any transmission correctly, and therefore the ratio value is one. At this point, it can be said that the system does not work. For values of the parameter close to this one, although the BER is not one, it cannot be negligible, and it the responsibility of the user to choose correctly the frequency used depending on the system and the test.

The second observation is the variability of the number of cycles per test at the same frequency. As it occurred in the other scenario, for some cases the difference between the maximum and the minimum is too big to be neglected. The reason for this variability is related to the percentage of errors for a determined frequency. However, since the system is stopped every time it is blocked, this errors can get to the system at any point. As the frequency increases, the variability decreases until a point in which the errors are so recurrent that the system fails constantly.

Last, it must be analyzed the type of error that blocks the system at each frequency. The behavior of the system is like the one in the other scenario. For low frequencies, the system is more likely to fail for a fault in the communication process that send the data to the DUT from the monitor. However, in this case there are more re-start faults that in the other case. For higher frequencies, the system is more sensible to the errors in the communication process that sends the outputs of the test to the monitor system. The re-start error does not seam to follow a pattern and it affects at all the frequencies tested.

These communication errors are produced for the lack of time to execute all the tasks of the process each time the clock has a rising edge. This has as a consequence that a signal that triggers a condition may be lost or is never set. Since the system is not capable of continuing the cycle without that signal, the process is blocked, and it has to be reset to start again, losing the bit stream.

Chapter 5

Conclusions

The principal objective of this project was to develop a low-cost parametric testbed system which could be used to test the behavior of other circuits in harsh environment conditions. For this purpose, it was stablished that it is necessary two boards, one for the monitor system and the other one for the UUT that is contained in the DUT, that must be interconnected in order to be able to exchange the information.

After developing all the necessary modules and designs for the two boards, the system was tested to check its performance in different situations, this is different number of inputs and outputs and frequencies. In the last chapter, the two scenarios have been analyzing independently, it is necessary to compare the results in order to draw conclusions of the system.

The first observation comes from the comparation of the results shown in both tables. It is easy to see that for the same values of the frequency, the number of cycles successfully finished is different. As expected, the scenario of eight I/O has better results than the other one. The reason lies in the communication process, in the first case it is much simples as it only requires one package in each direction. This decreases the probability of the system to fail in these processes, which are the critic point in the system. The difference can be seen comparing the average cycles from each frequency.

These differences are also seen in the BER, which is the ratio used to examine the performance of the systems. In the following graphic, it can be seen the results of this ratio for both situations.

First of all, from the graphic it can be seen that for frequencies below ten megahertz the BER is so small that can be negligible for both cases. Even though the behavior of the system in terms of number of cycles finished is different, the ratio is so small that it can be concluded that they are alike in terms of performance.

The difference comes for higher frequencies, and specially above twenty-five megahertz. In this situation, the graphic shows a deviation of the curves. The BER increases faster for the case of sixteen I/O, than for the other case. However, both systems stop working in a frequency in between fifty an a hundred of megahertz. This deviation of the BER is due to the complexity added when the communication requires more than one package. Each transaction requires more code execution, and therefore it is more likely that the system will fail. When the

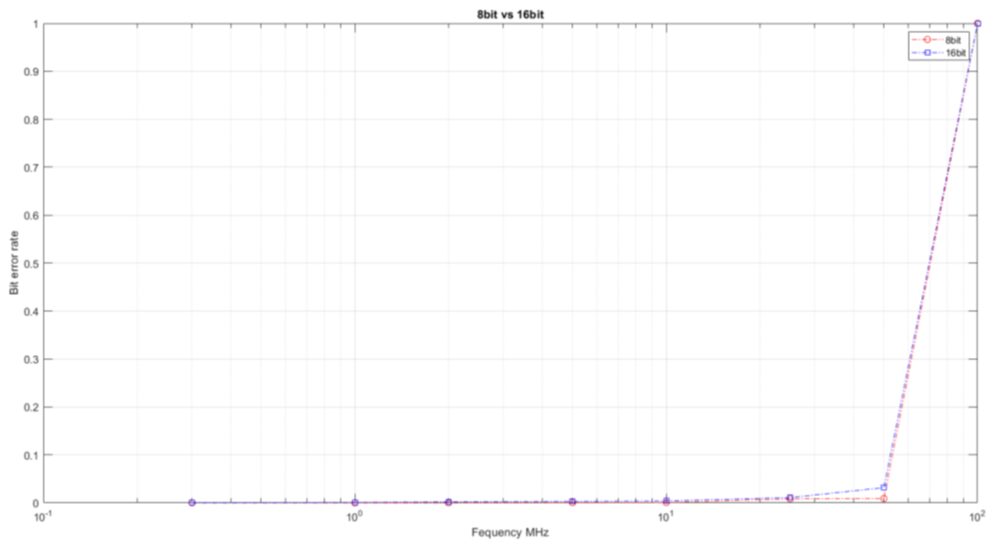


Figure 54. BER with respect to the frequency for 8 and 16 I/O

frequency is increased, the time between rising edges of the clock is decreased, leaving less time to execute all the tasks. Having to do more steps in the execution has as a consequence the increase of the BER.

However, the reason of this faults during the transaction of bits are the same in both scenarios for the different cases of frequencies. For the low values, the system tends to fail more often when the monitor is sending the information, and for high values when the DUT is the one sending the data to the monitor. The re-start fault can be found typically at all the frequencies with more or less the same assiduousness.

The conclusion that can be drawn from this tests is that the performance of the system is improved as the frequency and number of packages required decrease. For really low frequencies, the BER proves that in both situations the system is reliable. However, the number of achieved transmissions do change from one case to another, and this parameter must be taken into account when deciding if this platform performance is good enough to test a determined circuit.

Comparing these results and the design carried out for both boards with the initial objectives of this thesis, the following conclusions can be drawn. First, the main objective was to develop a testbed that could be implemented separately in two boards. One of the boards must be the monitor system, in charge of controlling the functioning and sending information to the other board, the DUT that contains the circuit to be tested. This condition has been respected in the thesis, and the final design follows this structure. As it was asked to be a low-cost system, it was chosen jumpers to interconnect the boards instead of a more sophisticated method. This decision may save money; however, the performance of the system is compromised due to the low quality of these wires. As they are not isolated, it is possible that the information that is transmitted may be distorted by the noise. It is important for this reason to keep the grounds of both boards connected to minimize this effect.

About the second objective, that is related with the communication between the boards, it has been developed a series of VHDL modules that are implemented in both boards to assure this task. The protocol chosen is a variation of the handshaking one. This is done implementing two signals that allow each modules to inform about the readiness of the system to receive the data, or to warn the other that the data is available in the channel.

Analyzing the final results, the behavior and performance of the system was the one expected, however it is possible to improve the quality of the system. First it is necessary to detect the possible issues that could be change for this objective.

The VHDL modules were design from scratch, and for this reason they were continuously modified in order to solve the issues and meet the criteria of the thesis. However, this lead to a situation in which the communication protocol is a little complex. The number of cycles required for each process is above the necessary ones, and it is necessary to improve this modules in order to improve the communication.

Also, in the design were used AXI GPIO IPs. However, this blocks require the AXI interface to connect the PL with the PS, and this interconnection requires more time to exchange the information than others like the EMIO. These extended MIO pins can be used from the programable logic side with the use of some libraries that are included in the platform in SDK. There are up to sixty-four of them available, which is enough for the number of bits that is used in this project.

Last, although it was possible to carry out the tests, it was a time-requiring task, as every time the system failed, it had to be reprogrammed and started again manually. This also has as a consequence that there is not the possibility of sending a bit stream of information into the UUT, which difficulties the calculus of the BER ratio, used to analyze how is the performance of the communication.

As a conclusion of this project, it is necessary to remark that the system that has been created is a first prototype. The development of this generic testbed allows users to carry out tests in hard environmental conditions with a minimized cost. This is really important, since the electronic market is strongly competitive, and the race to develop the systems in the least possible time and minimizing the costs, push the designers to the limits. However, this kind of systems allows to test the designs made with a low cost.

5.1. Future lines

As it has been explained, this system is a first prototype, and there are multiple improvements that can be introduced in order to get better results and performance of the system. Here are mentioned some of the possibilities for the future lines.

First, it has been explained that the connections between the boards are simple jumpers with no isolation, and this may interfere with the correct functioning of the system at determined frequencies. It is possible to improve this connections, trying to respect the low-cost aspect. One possibility includes using wires surrounded by another wires connected to ground. This isolation can be hand-made, decreasing the cost of this improvement, and may isolate the communication in order to get better results.

The second future line is related to the block design. As it was said, although the communication between the boards is assured, it is possible to improve this system to get a more reliable and stable exchange of data. For this purpose, the protocol used can be improved, as it depends on two signals that have to be detected in the correct timing. If those signals are not set nor no detected, then the system is blocked. Also, the block design can be modified in order to try if the system works faster with EMIOs connecting the PL to the PS instead of the AXI GPIOs.

Last, it is required to improve the testing methods to analyze in a better way the real performance of the system. As it has been explained, it is desired to implement the possibility of feeding the test with a constant bit stream, and for this it is necessary to implement this option in the software application. Whenever the system is blocked due to a failing communication, it must be able to detect it and re-start a new cycle, giving this one as a fail in the transmission.

Introducing this last option will allow to test in a more efficient way, and as a consequence, the results will be more complete and for more frequencies and scenarios. This will allow to study in a better way the performance of the system and the possible errors that may produce the block of the communication.

BIBLIOGRAPHY

1. **XILINX.** Field Programmable Gate Array (FPGA). [Online] https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html.

2. **Stallings, William.** *Operating systems: Internals and Design Principles.*

3. **Fortier , Paul J. and Howard E., Michel.** *Computer Systems Performance Evaluation and Prediction.* s.l. : Digital Press, 2003.

4. **PYNQ: PYTHON PRODUCTIVITY FOR ZYNQ. [Online]** http://www.pynq.io/.

5. TUL. Product: FPGA. [Online] http://www.tul.com.tw/ProductsPYNQ-Z2.html.

6. XILINX. Zynq-7000 SoC Data Sheet: Overview. [Online] https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

7. *Study of the data exchange between Programmable Logic and Processor System of Zynq-7000 devices.* Melo, Rodrigo A., et al. 2019.

8. XILINX. Zynq Architecture. [Online] 2012. http://www.ioe.nchu.edu.tw/Pic/CourseItem/4468_20_Zynq_Architecture.pdf.

9. Aging effects in FPGAs: An experimental analysis. Conference Digest - 24th International Conference on Field Programmable Logic and Applications. **Amouri, Abdulazim & Bruguier, Florent & Kiamehr, Saman & Benoit, Pascal & Torres, Lionel & Tahoori, Mehdi. 2014.**

10. A System for FPGA Aging Test. **Zong Jie Xiang, Wei Liu, Li heng Wang, Lan Lai Wang. 2018.**

11. A Zynq-based Testbed for the Experimental Benchmarking of the Algorithms Competing in Cryptographic Contests. **Farahmand,, Farnoud, Homsirikamol,, Ekawat and Gaj, Kris. 2007.**

12. A SEU test and simulation method for Zynq BRAM and flip-flops. **Yao, Ji, Shaojun, Wang and Yu, Peng. 2017.**

13. Harris, Sarah L. and Harris, David Money. Digital Design and Computer Architecture. 2016.

14. XILINX. Vivado Design Suite User Guide: Logic Simulation. [Online] 2018. https://china.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug900-vivado-logic-simulation.pdf.

15. XILINX. Vivado Desing Suite User Guide: Synthesis. [Online] 2013. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_2/ug901-vivado-synthesis.pdf.

16. XILINX. PYNQ-Z2 schematics. [Online] 2018. http://www.tul.com.tw/download/TUL_PYNQ%20Schematic_R12.pdf.

17. Warycka, Andy. What is meant by a hanshaking protocol? [Online] https://smallbusiness.chron.com/meant-handshaking-protocol-73134.html.

18. ITC99 Benchmark Home Page. [Online] https://www.cerc.utexas.edu/itc99-benchmarks/bench.html.

19. IIT Kharagpur, Department of Electrical Engineering. Testing of Embedded System. [Online] https://nptel.ac.in/content/storage2/courses/108105057/Pdf/Lesson-39.pdf.

20. FrenzelJr, Louis E. Handbook of Serial Communications Interfaces. 2016.