ANALYSIS OF SOFT-DMAS ON LOW-COST ZYNQ DEVICES FOR IOT PLATFORMS RUNNING BARE-METAL APPLICATIONS

by
GERARD PLANELLA FONTANILLAS
Internet Technologies & Storage Research Group
Engineering and Architecture La Salle
Ramon Llull University

Submitted in accordance with the requirements for the Degree of
Electronics and Communication Engineering - Minor in Robotics

Supervised by:

Joaquim Porte and Eduard Fernández

[May 22, 2020 at 8:49 – classicthesis Version 0.1]

ABSTRACT

Since its official naming in 1999, Internet of Things has become ever so relevant in today's society. Our need for higher bandwidth and smarter machines, accompanied by the many advances in the technology industries, has allowed many ideas that seamed unfeasible 20 years ago part of our day to day lives. This project has both scientific and ethical goals. Ways of increasing performance on an NVIS sensor network deployed in both Antarctica and Peru have been explored using a ZYNQ SoC running a bare-metal application. The use of different DMA IPcores was also applied to different data acquisition scenarios and studied. Achieving a higher performance in IoT systems is vital when the only way of accessing information is through these platforms.

Keywords: ZYNQ, DMA, CDMA, IoT, PS, PL, CPU, FPGA, bare-metal.

ABSTRACT

Desde su nombramiento oficial en 1999, el Internet of Things se ha vuelto más relevante que nunca en nuestra sociedad actual. Nuestra necesidad para anchos de banda mayores y maquinas más inteligentes, acompañado por los muchos avances de las empresas tecnológicas, han permitido que muchas ideas que parecían imposibles hace 20 años formen parte de nuestro día a día. Este proyecto tiene metas tanto éticas como científicas. Se han explorado diferentes maneras de aumentar el rendimiento de una red de sensores desplegada en la Antártida y en Perú, comunicados por NVIS. Se ha utilizado un ZYNQ SoC corriendo una aplicación bare-metal. También se ha estudiado el uso de diferentes IPcores de DMA, aplicado a diferentes escenarios de adquisición de datos. Adquirir un rendimiento mayor en sistemas IoT es vital cuando la única fuente de acceso a cierta información es a través de estas plataformas.

Palabras Clave: ZYNQ, DMA, CDMA, IoT, PS, PL, CPU, FPGA, bare-metal.

La energía sigue al pensamiento,
Juan Carlos.

ACKNOWLEDGMENTS

"Vull donar les gràcies a la meva mare Gemma, al meu pare Rafael i a la meva germana Nidia pel seu suport incondicional. També dono gràcies als meus companys pel suport moral que m'han donat sempre. Finalment, agraeixo l'ajuda dels meus professors, en Joaquim i l'Eduard. Encara que potser no ho saben, han sigut una gran font de motivació per a mi, especialment en els moments més difícils d'aquest projecte." G. Planella.

"I want to thank my mother Gemma, my father Rafael and my sister Nidia for the unconditional support they have given to me. I also want to thank my friends for the moral support they have and always have given to me. Finally, I appreciate the help that my professors, Joaquim and Eduard, have given to me. Even if they are not conscious of it, they have been a great source of motivation for me, especially in the hardest moments of this project" G. Planella.

CONTENTS

1 INTRODUCTION 1
1.1 Relevance and Motivation behind the study . . . 1
1.2 Objectives . . . 2
1.3 Structure of the Document . . . 2
2 THEORETICAL BACKGROUND 5
2.1 Application Domain . . . 5
2.2 Zynq . . . 8
2.2.1 PS . . . 8
2.2.2 PL . . . 11
2.3 Chosen Development Board . . . 11
2.4 AXI Protocol . . . 14
2.5 DMA . . . 14
2.5.1 AXI DMA . . . 16
2.5.2 Other DMAs . . . 16
2.6 Other Relevant IPcores used . . . 17
2.6.1 Processor System Reset . . . 17
2.6.2 Slice . . . 18
2.6.3 Concat . . . 18
2.6.4 Const . . . 18
2.6.5 Clocking Wizard . . . 18
2.6.6 FIFO Generator and AXI-Stream Data FIFO . . . 19
2.6.7 AXI GPIO . . . 20
2.6.8 Traffic Generator . . . 21
2.6.9 Block Memory Generator . . . 21
2.6.10 BRAM Controller . . . 22
2.7 IDE . . . 22
2.8 Related Work . . . 22
3 DESIGN AND TECHNICAL IMPLEMENTATION 25
3.1 Clock Generation . . . 26
3.2 Scenario 1: DMA and FIFO . . . 27
3.2.1 PL . . . 27
3.2.2 PS . . . 33
3.3 Scenario 2: CDMA . . . 40
3.3.1 PL . . . 41
3.3.2 PS . . . 47
3.4 Scenario 3: GRITS NVIS RX Channel . . . 52
3.4.1 PL . . . 53
3.4.2 PS . . . 62
4 METHODOLOGY 73
5 RESULTS 75
5.1 Scenario 1 . . . 75
5.1.1 Effect of Data Cache . . . 75
5.1.2 Effect of Direct Memory Access (DMA) Burst Size . . . 77

5.1.3 Effect of Programmable Logic (PL) Frequency 81
5.2 Scenario 2 84
5.2.1 Effect of Data Cache 84
5.2.2 Effect of CDMA Burst Size 86
5.2.3 Effect of PL Frequency 87
5.3 Scenario 3 90
5.3.1 Effect of traffic generator delay and frequency 90
5.3.2 Effect of traffic generator length and frequency 92
5.3.3 Maximum Throughput 94
5.3.4 NVIS real-time throughput 95
6 CONCLUSIONS 97
7 FUTURE LINES OF RESEARCH 101

BIBLOGRAPHY 103

I APPENDIX 107
A DMA SCATTER GATHER MODE 109
B SCENARIO 1 VIVADO VIEW 111
C SCENARIO 2 VIVADO VIEW 115
D SCENARIO 3 VIVADO VIEW 119
E GITHUB REPOSITORY 125

LIST OF FIGURES

Figure 1 Red Pitaya development board . . . 6
Figure 2 Long Term Evolution (LTE)/4G standards adapted to Internet of Things (IoT)[9] . . . 7
Figure 3 Simplified diagram of a Zynq device[10] . . . 8
Figure 4 Zynq-7000 diagram[15] . . . 9
Figure 5 Zynq-7000 diagram provided by the Vivado Integrated Development Environment (IDE) . . . 10
Figure 6 Comparison of different Zynq-7000 devices . . . 12
Figure 7 Meaning of the Zynq device name. . . . 12
Figure 8 Cora Z7-07S development board . . . 13
Figure 9 Cora board booting modes[19] . . . 13
Figure 10 DMA and Processor conceptual connection. . . . 15
Figure 11 DMA Maximum frequencies for different speed grades. . . . 16
Figure 12 Processor System Reset Block . . . 17
Figure 13 Slice block . . . 18
Figure 14 Concat block. . . . 18
Figure 15 Const block. . . . 18
Figure 16 Clocking Wizard Block . . . 19
Figure 17 Advanced eXtensible Interface (AXI) Interconnect block . . . 19
Figure 18 FIFO Generator block configured to act as an AXI-Stream Data FIFO . . . 20
Figure 19 AXI General Purpose Input Output (GPIO) Block . . . 20
Figure 20 Traffic Generator block configured in streaming mode . . . 21
Figure 21 dual-port Block Random Access Memory (BRAM) . . . 21
Figure 22 BRAM controller . . . 22
Figure 23 Scenario 1:PL Fabric clock . . . 26
Figure 24 PL Fabric clock Generation . . . 27
Figure 25 General view of Scenario 1 . . . 28
Figure 26 Scenario 1:Zynq Configuration, Enable Master GPO interface . . . 30
Figure 27 Scenario 1:Zynq configuration, Enable Zynq Interrupts . . . 30
Figure 28 Scenario 1:Zynq configuration, check UART is enabled . . . 31
Figure 29 Scenario 1: FIFO configuration . . . 31
Figure 30 Scenario 1: DMA Configuration . . . 32
Figure 31 Scenario 1: Implementation view . . . 33
Figure 32 Scenario 1: General Flow chart . . . 34
Figure 33 Scenario 1: MM2S Test Flow chart . . . 35
Figure 34 Scenario 1: MM2S Latency Test Flow chart . . . 36
Figure 35 Scenario 1: MM2S Throughput test Flow chart . . . 37
Figure 36 Scenario 1: S2MM Test Flow chart . . . 38
Figure 37 Scenario 1: S2MM Latency Test Flow chart . . . 39
Figure 38 Scenario 1: S2MM Throughput Test Flow chart . . . 40
Figure 39 Scenario 2: General view . . . 41

Figure 40 Scenario 2: Zynq Interface Configuration . . . 43
Figure 41 Scenario 2: Zynq Interrupt Configuration . . . 44
Figure 42 Scenario 2: Central Direct Memory Access (CDMA) Configuration 44
Figure 43 Scenario 2: BRAM Configuration . . . 45
Figure 44 Scenario 2: BRAM Controller Configuration . . . 46
Figure 45 Scenario 2: Implementation view from Vivado . . . 47
Figure 46 Scenario 2: General Flow Chart . . . 48
Figure 47 Scenario 2: General Test Flow Chart . . . 49
Figure 48 Scenario 2: Latency and Configuration Test Flow Chart . . . 50
Figure 49 Scenario 2: Throughput and Drop Rate Test Flow Chart . . . 51
Figure 50 Scenario 3: General view of the scenario . . . 52
Figure 51 Scenario 3: Clock Domains . . . 53
Figure 52 Scenario 3: Interface configuration for the ZYNQ Processing system . . . 55
Figure 53 Scenario 3: Interrupt configuration for the ZYNQ Processing system . . . 56
Figure 54 Scenario 3: DMA Configuration . . . 57
Figure 55 Scenario 3: FIFO Configuration . . . 58
Figure 56 Scenario 3: Traffic Generator Configuration (1) . . . 59
Figure 57 Scenario 3: Traffic Generator Configuration (2) . . . 59
Figure 58 Scenario 3: AXI GPIO Interrupt configuration . . . 60
Figure 59 Scenario 3: FPGA Implementation . . . 61
Figure 60 Scenario 3: General Flow Chart for the Latency test . . . 63
Figure 61 Scenario 3: Flow Chart for the Latency test . . . 64
Figure 62 Scenario 3: General Flow Chart . . . 66
Figure 63 Scenario 3: Mode Test Flow Chart . . . 67
Figure 64 Scenario 3: Mode Test S2MM IRQ Flow Chart . . . 69
Figure 65 Scenario 3: Mode Test GPIO IRQ Flow Chart . . . 71
Figure 66 Effect of Data Cache on Total Time, Memory Mapped to Stream (MM2S) transfers. . . . 75
Figure 67 Effect of Data Cache on Latency and Configuration Time, MM2S transfers. . . . 76
Figure 68 Effect of Data Cache on Total Time, Stream to Memory Mapped (S2MM) transfers. . . . 76
Figure 69 Effect of Data Cache on Latency and Configuration Time, S2MM transfers. . . . 77
Figure 70 Effect of DMA Burst Size on Total Time, MM2S transfers. . . . 78
Figure 71 Effect of DMA Burst Size on Latency, MM2S transfers. . . . 78
Figure 72 Effect of DMA Burst Size on Configuration Time, MM2S transfers. 79
Figure 73 Effect of DMA Burst Size on Total Time (1/2), S2MM transfers. . 79
Figure 74 Effect of DMA Burst Size on Total Time (2/2), S2MM transfers. . 80
Figure 75 Effect of DMA Burst Size on Latency, S2MM transfers. . . . 80
Figure 76 Effect of DMA Burst Size on Configuration Time, S2MM transfers. 81
Figure 77 Effect of PL frequency on throughput, MM2S and S2MM transfers. 81
Figure 78 Effect of PL frequency on Latency, MM2S and S2MM transfers. . 82
Figure 79 Effect of PL frequency on Configuration Time, MM2S and S2MM transfers. . . . 82

Figure 80 Effect of Data Cache on Total Time . . . 84
Figure 81 Effect of Data Cache on Latency . . . 85
Figure 82 Effect of Data Cache on Configuration Time . . . 85
Figure 83 Effect of CDMA Burst Size on Total Time . . . 86
Figure 84 Effect of CDMA Burst Size on Latency . . . 86
Figure 85 Effect of CDMA Burst Size on Configuration Time . . . 87
Figure 86 Effect of PL Frequency on Throughput . . . 87
Figure 87 Effect of PL Frequency on Latency . . . 88
Figure 88 Effect of PL Frequency on Configuration Time . . . 88
Figure 89 Effect of Traffic Generator delay and frequency on Total Time . . . 90
Figure 90 Effect of Traffic Generator delay and frequency on Configuration Time Time . . . 91
Figure 91 Effect of Traffic Generator delay and frequency on the number of FIFO Overflows . . . 91
Figure 92 Effect of Traffic Generator delay and frequency on Drop Rate . . . 92
Figure 93 Effect of Traffic Generator length and frequency on Total Time . . . 92
Figure 94 Effect of Traffic Generator length and frequency on Configuration Time Time . . . 93
Figure 95 Effect of Traffic Generator length and frequency on the number of FIFO Overflows . . . 93
Figure 96 Effect of Traffic Generator delay and frequency on Drop Rate . . . 94
Figure 97 Effect of FIFO Depth on drop rate at 65MHz . . . 94
Figure 98 Vivado Interface view of Scenario 1 . . . 112
Figure 99 Scenario 1 implemented in Vivado . . . 113
Figure 100 Vivado Interface view of Scenario 2 . . . 116
Figure 101 Scenario 2 implemented in Vivado . . . 117
Figure 102 Vivado Interface view of Scenario 3 . . . 120
Figure 103 Scenario 3 implemented in Vivado . . . 121
Figure 104 Scenario 3 implemented in Vivado (Part 1 of 2) . . . 122
Figure 105 Scenario 3 implemented in Vivado (Part 2 of 2) . . . 123

LIST OF TABLES

Table 1 Address map for environment 1 . . . 29
Table 2 Scenario 1: FPGA Resources . . . 33
Table 3 Scenario 2: Address Map . . . 43
Table 4 Scenario 2: Resources used . . . 46
Table 5 Scenario 3: Address Map . . . 55
Table 6 Scenario 3: FPGA Resources . . . 61
Table 7 Results for a 240MB transfer with an 8MHz Traffic Generator Frequency . . . 95

ACRONYMS

IoT Internet of Things
DMA Direct Memory Access
CDMA Central Direct Memory Access
VDMA Video Direct Memory Access
MCDMA Multi-Channel Direct Memory Access
CPU Central Processing Unit
DDR Double Data Rate
FPGA Field-Programmable Gate Array
PL Programmable Logic
PS Processing System
SRAM Synchronous Dynamic Random Access Memory
BRAM Block Random Access Memory
RAM Random Access Memory
FIFO First In First Out
AXI Advanced eXtensible Interface
MM2S Memory Mapped to Stream
S2MM Stream to Memory Mapped
MM2MM Memory Mapped to Memory Mapped
FPU Floating-Point Unit
GRITS Internet Technologies and Storage Research Group
NVIS Near Vertical Incidence Skywave
RF Radio Frequency
ADC Analog to Digital Converter
DAC Digital to Analog Converter
SMA SubMiniature version A
RX Receive
TX Transmit
LTE Long Term Evolution
DSP Digital Signal Processor
HP High Performance
GP General Purpose
GPIO General Purpose Input Output
ACP Accelerated Coherency Port

RTOS Real Time Operating System
CAN Controller Area Network
UART Universal asynchronous receiver-transmitter
SPI Serial Peripheral Interface
SDIO Secure Digital Input Output
USB Universal Serial Bus
JTAG Joint Test Action Group
AMBA Advanced Microcontroller Bus Architecture
MIO Multiplexed I/O
EMIO Extended Multiplexed I/O
CLB Configurable Logic Block
LUT Look-up Table
HDL Hardware Description Language
SCU Snoop Control Unit
PLL Phase-Locked Loop
IP Intellectual Property
GUI Graphic User Interface
ISR Interrupt Service Rountine
MCMM Multi-Corner Multi-Mode
IDE Integrated Development Environment
FSBL First Stage Boot Loader
IRQ Interrupt Request
MIG Memory Interface Generator

1 INTRODUCTION

There is an increasing demand for high-speed embedded devices which forces engineers of the field to stay up-to-date with new up and coming technologies more than ever. This study aims to analyse different scenarios that can apply to the field of IoT.

The scenarios implemented and analysed in this study are mainly used for data acquisition and data transfers, this implies that not only we must take into account the speed at which the system manages data, but also how reliable it is. This has played a crucial role when analysing the different scenarios as a balance between speed and reliability had to be met for success.

1.1 RELEVANCE AND MOTIVATION BEHIND THE STUDY

In order to properly appreciate why this project has been developed, it is important to understand the motivation behind it and the relevance it has in today's world.

This study started as a collaboration with the Internet Technologies and Storage Research Group (GRITS), a research group from **La Salle, Ramon Llull University**. One of the projects that GRITS works on is the Near Vertical Incidence Skywave (NVIS) ionospheric communications[1] project, which works on the deployment of a sensor network at a low cost and with the minimal infrastructure required. This project has been deployed in Antarctica [2], where a variety of different sensors transfer data to the **Juan Carlos I Spanish Antarctic Base**, where scientists can obtain measurements from different locations without needing a line of sight between the sensor and the base station.

Nevertheless, the project's scope is not only limited to Antarctica, but GRITS has also deployed this system in *Valle Sagrado*, Peru[3], to communicate villages where traditional Radio Frequency (RF) links would not be feasible due to the mountainous topography in that region.

We can see that achieving high throughputs in any of these two situations is crucial, especially in *Valle Sagrado*, where a higher data transfer rate would allow essential messages such as medical emergencies[4] to be received much quicker which has distinct benefits.

Studying in La Salle for the past four years has allowed me to develop a keen interest in the NVIS project. Seeing how the project has evolved and improved over the years. Helping to develop this platform further seems like a great way of applying all the knowledge acquired through these years and also learning about how to used Zynq devices, the integrated component that I used in the NVIS project for the data acqui-

sition and processing. The combination of developing in both C code and VHDL in one same component seemed very daunting at the beginning, but after completing this project, invaluable skills have been acquired that has allowed me to further develop as a future engineer.

1.2 OBJECTIVES

One of the features that had some space for improvement in the NVIS project was the throughput of the established RF links between the sensors and the Antarctic base. This is where this project comes into place. In order to maximise efficiency and trough put, it was essential to find out where the bottlenecks of the system were, to then try and eliminate them. Nevertheless, due to the complexity of the system, it was necessary to understand its underlying technologies in detail.

Therefore, this project not only tries to improve the existent NVIS project, but it also aims to help new students and engineers which may be inexperienced with the technology used in it, a Xilinx ZYNQ device(refer to section 2.2). This second point is of great importance as reducing the learning curve required to correctly understand the inner workings of the hardware used in the system, allows the future students that want to be part of GRITS to adapt to the project faster—enabling both the knowledge of the students and the technology of the NVIS project to grow together.

In order to improve the existing platform, the PS-PL throughput will be increased. Due to not knowing the exact location of a bottleneck in the existing platform, it was decided to migrate the software running on a Linux OS to a bare-metal application programmed in C. Requiring no use of an operating system meant that more control was had for controlling the hardware in the Zynq's FPGA. Three different scenarios were implemented, varying in their hardware configurations. The main block used to move data in these scenarios was a DMA. So if we want to increase the throughput of data between the PS-PL, we had to control this peripheral as efficiently as possible and configuring it to maximise its performance.

1.3 STRUCTURE OF THE DOCUMENT

This Document is Structured in six different parts:

1. Theoretical Background
2. Design and Implementation
3. Methodology
4. Results
5. Conclusions
6. Future Lines of Work

Firstly, we will find a detailed explanation of the required background knowledge needed to perform this project. Here the reader will be able to gain a better understanding of the technology used in the NVIS project after first exploring the application

domain of this study. Furthermore, examples of related work that have served as stepping stones for the development of this study will be explained and contrasted with the different implemented scenarios.

Secondly, the design of each scenario will be discussed, separating each of the three distinct ones into software and hardware to explain these two parts separately. Here, some of the limitations and problems found when developing each scenario will also be explained, as these can make it easier to understand the logic followed when implementing each of the scenarios.

The third point will explain how the results were acquired and analysed, which leads up to the set of results which will be shown and explained after this point together with the conclusions extracted from the obtained results.

After this, the study's conclusions will be explained. These conclusions differ from the result's conclusions as they are more linked to the project as a whole and how useful it has been to follow this project as a student.

Finally, future lines of work will be discussed, showing the reader possible improvements to be made or things that could be done to expand and improve the project but could not have been done due to various reasons.

2 THEORETICAL BACKGROUND

In order to properly understand the key concepts that this project tackles on, we will first explain the background knowledge that was required to be able to implement each scenario properly. This will include the application domain of this project, what a Zynq Device is and the different IPcores used to implement each scenario.

The NVIS project requires knowledge of many different disciplines, from Electronics to Telecommunications going through Computer Science and Hardware programming. This project mainly focuses on the last two. For more information on the Telecommunications aspect of the project, Carles Vilella i Parra's thesis[1] and Joaquim Porte's final degree project[5] explain the concepts applied in detail.

The different scenarios studied in this project will allow analysing the performance of the DMA in different scenarios, where each could be used for different applications where data wants to be transferred at high speed.

2.1 APPLICATION DOMAIN

The current version of the NVIS project developed by Joaquim Porte et al. [6], uses the Red Pitaya development board [7] in order to transmit and receive data to build up the sensor network. The benefit of utilising such platform is that it boasts a Xilinx Zynq device, which will be explained in further detail in section 2.2. The Zynq allows the system to integrate what would have traditionally required many different hardware components into one robust and compact board.
The Red Pitaya also has a high-speed 14-bit Analog to Digital Converter (ADC) and Digital to Analog Converter (DAC) which make it perfect for the proposed application as through these you can modulate or demodulate the signals that are to be sent or being received[7].

The board in figure 1 is the Red Pitaya board. From the image, we can see that it has four SubMiniature version A (SMA) connectors for antennas. The onboard ADC and DAC are connected to two SMA connectors each. In the NVIS project, only two of these are used.
Nevertheless, in the NVIS project, one of the SMA connectors is used for a Receive (RX) channel using the ADC and the other is used for a Transmit (TX) channel with the DAC. Both of the converters in the board work with the system's clock frequency of 125MHz, which limit the frequency of the signals that can be sent and received.

The NVIS project uses the Red Pitaya to transmit and receive data from different sensors. The Red Pitaya is connected to a Raspberry PI, which will receive the data read by the Red Pitaya, and it will send to the Red Pitaya the data that has to be transmitted. This allows data received from different sensors like a GPS module connected to the

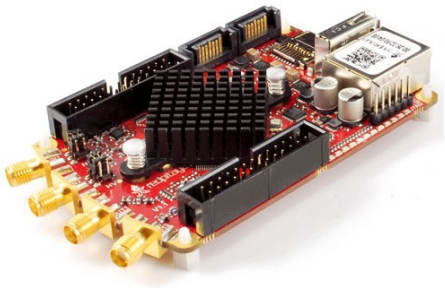


Figure 1: Red Pitaya development board

Raspberry PI to be sent through the Red Pitaya. Using many of these allows to build up a sensor network, without requiring a cabled connection between remote sensors. For more information on how the exact configurations of the hardware used for each node in the sensor network, refer to Joaquim Porte's master's thesis[6].

In the NVIS project, the Red Pitaya has two very distinct modes. Firstly we have the normal mode. In this mode the Red Pitaya has two jobs:

- RX: The Red Pitaya will read from its ADC, then the data will be demodulated and processed by the Zynq. If a frame is detected, the Zynq will temporarily store it in the SRAM. When all the frame has been received the Zynq will send this frame from the Synchronous Dynamic Random Access Memory (SRAM) to a Raspberry PI through a point-to-point Ethernet connection from its onboard Ethernet connector.
- TX: The Raspberry PI will send the data to the Red Pitaya through the same Ethernet connection, where the data will be temporarily stored in the SRAM until it has to be framed, modulated and sent using the DAC.

The second mode is the real-time mode[8]. In this mode, most of the data processing is moved to the Raspberry PI. This means that the Red Pitaya does not have to wait to receive a whole frame for it to be sent to the Raspberry PI, it only has to worry about the demodulation and modulation of the frames and to transfer these to the Raspberry PI or to the DAC. This enables the system to work in real-time.

This project is going to try to increase the throughput in the used Zynq between PS and PL by improving the existent hardware design, developed in the Zynq's Field-Programmable Gate Array (FPGA) and trying to efficiently control it using a bare metal application. This will allow for a greater bitrate to be acquired in future upgrades to theNVIS project. The current PS-PL throughput will be now calculated:

1. Normal Mode: In order to transfer one frame, 10 packets of 100K samples are sent for a period of one second. Each sample consisting of 64 bits.

$$10 \cdot 100K \cdot 64b = 64Mb = 8MB \tag{1}$$

This is the total amount of data transferred.

$$\frac{64Mb}{1s} = 64Mbps = 8MBps \tag{2}$$

A throughput of 64Mbps is achieved.

2. real-time Mode: To transfer one frame, 600 packets of 100K samples of 64 bits are transferred in 60 seconds.

$$600 \cdot 100K \cdot 64b = 3.94Gb = 480MB \tag{3}$$

A total of 480MB are transferred.

$$\frac{3.9Gb}{60s} = 64Mbps = 8MBps \tag{4}$$

A throughput of 64Mbps is achieved.

Furthermore, being an IoT application, the bitrate of the system is used to characterise it. This value should not be confused by the number of bits transmitted per unit of time. In IoT, the bitrate refers to the amount of actual data that is being received and transmitted by the IoT platform. Which is dependent on the modulation used for transmitting such data. The NVIS platform has bitrate that can vary from 3kbps to 15kbps in real-time mode[8], the specific bitrate will depend on the conditions set.

A way of characterising an IoT device is using the LTE/4G standards adapted to IoT.

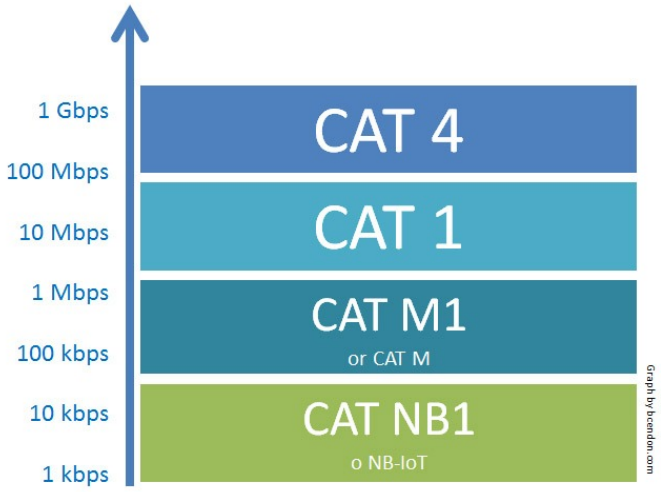


Figure 2: LTE/4G standards adapted to IoT[9]

Observing figure 2 it can be seen that the current NVIS project in real-time mode is found between the category CAT NB-IoT or CAT M1 on the LTE/4G scale depending on the conditions.

Zynqs are Integrated components produced by the company Xilinx that encompass both the software programmability of a Central Processing Unit (CPU)(typically an ARM-based processor) and the hardware programmability of an FPGA. As can be imagined, this combination can be advantageous as it allows for hardware-accelerated platforms which can drastically increase the performance of many applications.
A Zynq device can be divided into two very distinct parts, the Processing System (PS) and the PL. The PS normally contains the CPU and other peripherals; it is programmed similarly to other ARM processors. The PL contains the FPGA, Digital Signal Processor (DSP) and BRAM.

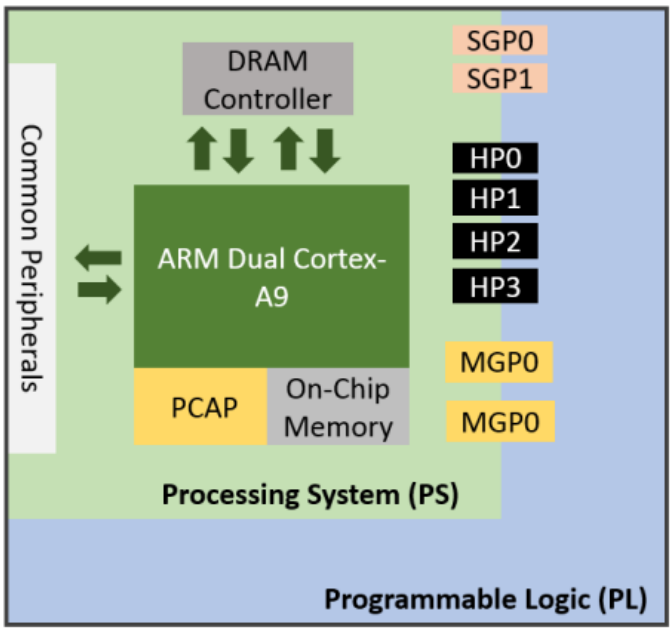


Figure 3: Simplified diagram of a Zynq device[10]

Observing figure 3 we can see this division between PS and PL. It can also be observed that PS-PL communication is performed through a series of interfaces, General Purpose (GP) and High Performance (HP). Aside from these two, there are also more; these interfaces will be explained in section 2.2.2.
We will proceed by explaining in detail one specific type of Zynq family called Zynq-7000[11]. There are other more complex families of Zynq devices such as the Zynq UltraScale + family[12] but this will not be explained in this project.

2.2.1 *PS*

As mentioned in 2.2 the PS consists of a CPU, different peripherals, and memory controllers.

In the Zynq-7000 family, the CPU consists of an ARM Cortex A9 multiprocessor core, depending on the exact model of the Zynq device you will find more or fewer cores.

The Red Pitaya has a dual-core ARM Cortex A9, which allows you to run PetaLinux[13] on it for your embedded applications with no problem. This can be helpful as it means you can run programs using high-level languages like Python or Java for non-time-critical applications or if more precision is needed C language. Real Time Operating System (RTOS) or a bare-metal application can be run which will be used for time-critical applications. A RTOS is an operating system that is specialised for real-time applications that process data as it is being received, for more information on RTOS refer to FreeRTOS's website[14] a widely used free RTOS. This project has implemented bare-metal applications for each of the proposed scenarios in order to maximise performance and reliability as not having an operating system allows you to have more control over the performance of the PS.

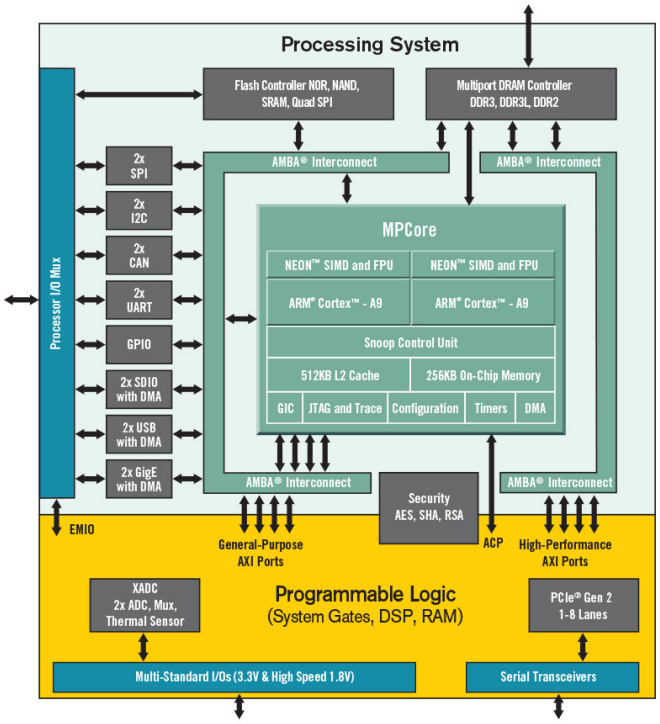


Figure 4: Zynq-7000 diagram[15]

From figure 4 we can observe some of the peripherals that the PS has.

- Serial communication Protocols: Such as Universal Serial Bus (USB) Controller Area Network (CAN), Universal asynchronous receiver-transmitter (UART), I2C and Serial Peripheral Interface (SPI). These can be very useful when your Zynq device has to communicate with other peripherals outside the integrated component. We also have Gigabit Ethernet interfaces and an Secure Digital Input Output (SDIO) interface for an SD card. The SD Card can be used to store the PetaLinux image for booting or for simple data storage. We can also see that the peripherals are connected to the Multiplexed I/O (MIO) interface, which allow the peripherals to be mapped to different pins as needed.

- Debugging interface: The ARM Cortex A9 contains a Joint Test Action Group (JTAG) debugging module which is used for both debugging and programming the PS and PL.
- GPIO: These allow the PS to read and write to digital pins.
- Memory Controllers: The Flash and Double Data Rate (DDR) memory controllers abstract PS and PL from the access protocols the controllers have to implement to generate memory accesses.

Furthermore, the PS has an Advanced Microcontroller Bus Architecture (AMBA) AXI Interconnect to allow the PL to access some of the peripherals in the PS.To learn more about interconnects refer to section**??**.

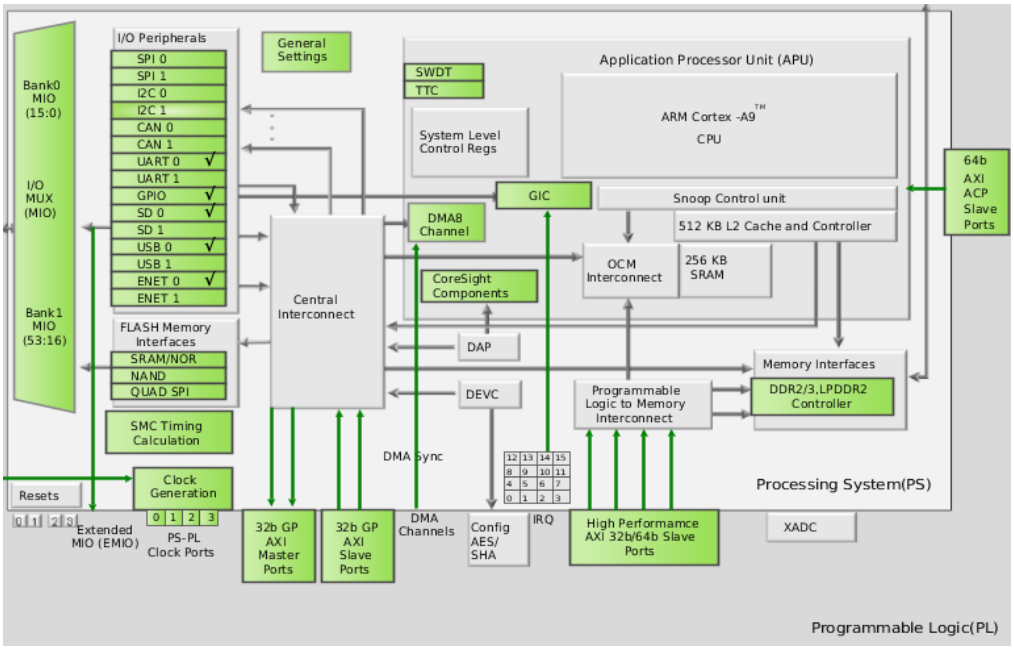


Figure 5: Zynq-7000 diagram provided by the Vivado IDE

Observing figure 5 we can see a more detailed version of a Zynq device, this one in specific has a single-core CPU. It can be appreciated that the CPU has an L1 and L2 Cache that is connected to a Snoop Control Unit (SCU). This SCU maintains Cache coherency and also acts as a semaphore for the L2 cache. The cache is a memory that stores the data that is most frequently accesses by a CPU.

Caches normally have a very low latency which means that for data that has to be accessed frequently, accesses to a low latency memory will increase the efficiency of data use. There exist different levels of caches. Level 1 Cache refers to a cache found in the same die as a CPU. On the other hand, a level 2 caches can be either external or found with the core. L2 Cache will be slower than L1 cache but of a greater size. We can also find L3 Caches, which again, are slower than L2 Caches but have a greater capacity.
A problem arises when different memories with copies of the same data are found. Data Coherency has to be ensured, if data in one cache is modified and that data is

also found in the other caches, all caches will have to be updated and so will the main memory. This is what the SCU ensures.

2.2.2 *PL*

The PL is made up by an FPGA. The Zynq-7000 family usually has an Artix-7 or Kintex-7 FPGA, as it can be seen in figure **??**. For a more detailed description of the internal structure of an FPGA, you can refer to the overview of the 7-series FPGAs by Xilinx[16].

If we look back at figure 5 we can see that the PL has many different connections to the PS, some of these will be explained:

1. Extended Multiplexed I/O (EMIO): This interface connects the PL to the MIO in the PS, which allows the PS and PL to share peripherals.

2. GP: This type of interface has master and slave variants and is used for general purpose accesses between PS and PL that do not require an excessive bus bandwidth.

3. HP: This 64/32 bit interface is used to perform accesses that require a high bus bandwidth, or to perform transfers where minimal latencies are needed.

4. Accelerated Coherency Port (ACP): 64-bit slave interface that provides connectivity between the PS and the PL. The ACP directly connects the PL to the snoop control unit SCU of the ARM Cortex-A9 processors, enabling cache-coherent access to CPU data in the L1 and L2 caches.

5. PL clock ports: These connect the clock generator to the PL. The clock generator contains a Phase-Locked Loop (PLL) to produce different clock frequencies. A PLL can also be instantiated in the PL as an alternative.

Another thing we can find in the PL is BRAM. An FPGA only has a discrete amount of BRAM. This memory can be used to implement things like a First In First Out (FIFO) or a dual-port memory. The benefit of using BRAM is the low latency needed to access it from the PL. Nevertheless, the reduced amount in most low-cost Zynq devices means it must be used with care.

The PL allows you to develop hardware as easy as an AND gate or as complex as an actual CPU. Xilinx offers its clients Intellectual Property (IP) cores or IPcores. These are seen by the user as Hardware blocks and can be instantiated in Hardware Description Language (HDL) or through Graphic User Interface (GUI). The blocks used to implement the scenarios will be explained in section 2.6.

2.3 CHOSEN DEVELOPMENT BOARD

Initially, the board this project was going to use was the Red Pitaya, but due to a faulty development board and no spare Red Pitayas, it was needed to choose a new development board. To do this, knowing the differences between the Zynq-7000 devices was required.

		Cost-Optimized Devices						Mid-Range Devices			
	Device Name	Z-7007S	Z-7012S	Z-7014S	Z-7010	Z-7015	Z-7020	Z-7030	Z-7035	Z-7045	Z-7100
	Part Number	XC7Z007S	XC7Z012S	XC7Z014S	XC7Z010	XC7Z015	XC7Z020	XC7Z030	XC7Z035	XC7Z045	XC7Z100
Processing System (PS)	Processor Core	Single-Core ARM® Cortex™-A9 MPCore™ Up to 766MHz			Dual-Core ARM Cortex-A9 MPCore Up to 866MHz			Dual-Core ARM Cortex-A9 MPCore Up to 1GHz(1)			
	Processor Extensions	NEON™ SIMD Engine and Single/Double Precision Floating Point Unit per processor									
	L1 Cache	32KB Instruction, 32KB Data per processor									
	L2 Cache	512KB									
	On-Chip Memory	256KB									
	External Memory Support(2)	DDR3, DDR3L, DDR2, LPDDR2									
	External Static Memory Support(2)	2x Quad-SPI, NAND, NOR									
	DMA Channels	8 (4 dedicated to PL)									
	Peripherals	2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO									
	Peripherals w/ built-in DMA(2)	2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO									
	Security(3)	RSA Authentication of First Stage Boot Loader, AES and SHA 256b Decryption and Authentication for Secure Boot									
	Processing System to Programmable Logic Interface Ports (Primary Interfaces & Interrupts Only)	2x AXI 32b Master, 2x AXI 32b Slave 4x AXI 64b/32b Memory AXI 64b ACP 16 Interrupts									
Programmable Logic (PL)	7 Series PL Equivalent	Artix®-7	Artix-7	Artix-7	Artix-7	Artix-7	Artix-7	Kintex®-7	Kintex-7	Kintex-7	Kintex-7
	Logic Cells	23K	55K	65K	28K	74K	85K	125K	275K	350K	444K
	Look-Up Tables (LUTs)	14,400	34,400	40,600	17,600	46,200	53,200	78,600	171,900	218,600	277,400
	Flip-Flops	28,800	68,800	81,200	35,200	92,400	106,400	157,200	343,800	437,200	554,800
	Total Block RAM (# 36Kb Blocks)	1.8Mb (50)	2.5Mb (72)	3.8Mb (107)	2.1Mb (60)	3.3Mb (95)	4.9Mb (140)	9.3Mb (265)	17.6Mb (500)	19.2Mb (545)	26.5Mb (755)
	DSP Slices	66	120	170	80	160	220	400	900	900	2,020
	PCI Express®	—	Gen2 x4	—	—	Gen2 x4	—	Gen2 x4	Gen2 x8	Gen2 x8	Gen2 x8
	Analog Mixed Signal (AMS) / XADC(2)	2x 12 bit, MSPS ADCs with up to 17 Differential Inputs									
	Security(3)	AES & SHA 256b Decryption & Authentication for Secure Programmable Logic Config									
	Speed Grades Commercial	-1			-1			-1			-1
	Speed Grades Extended	-2			-2,-3			-2,-3			-2
	Speed Grades Industrial	-1, -2			-1, -2, -1L			-1, -2, -2L			-1, -2, -2L

Notes:
1. 1 GHz processor frequency is available only for -3 speed grades in Z-7030, Z-7035, and Z-7045 devices. See DS190, *Zynq-7000 SoC Overview* for details.
2. Z-7007S and Z-7010 in CLG225 have restrictions on PS peripherals, memory interfaces, and I/Os. Please refer to UG585, *Zynq-7000 SoC Technical Reference Manual* for more details.
3. Security block is shared by the Processing System and the Programmable Logic.

XILINX

Figure 6: Comparison of different Zynq-7000 devices

Figure 6 shows different Zynq devices at different price levels. The Red Pitaya uses the Zynq Z-7010. Since a bare-metal application was going to be run and only one of the two cores was going to be used, it was decided that having a dual-core CPU was not crucial. Therefore one of the Zynq Devices with the 766MHz CPU was to be selected.

Another essential feature to take into consideration was the speed grade of the PL.In Xilinx FPGAs, the speed grade is an indicator of how fast can the PL work to meet timing requirements. Some IPcores will allow different maximum frequencies depending on the board's speed grade[17].
But how do we check our boards speed grade? We can get this value from the Zynq's device name[18]: In the case of the Red Pitaya, its device name is *XC7C010CLG400-1*.

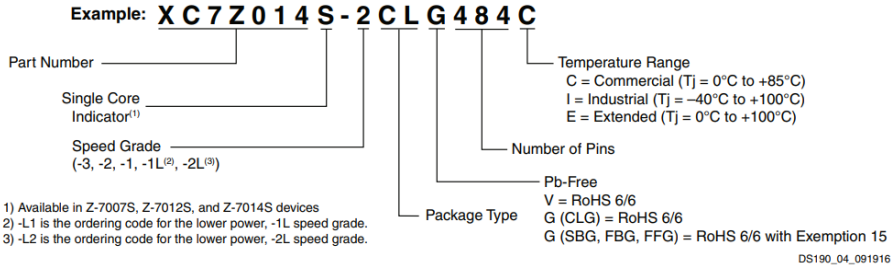


Figure 7: Meaning of the Zynq device name.

From its name and referring to figure 7 we can see that the speed grade is -1, the highest speed grade is -3. Therefore we have to select a Zynq device with a speed grade between -1 and -3.

After searching the market for low-cost development boards, it was decided to use the Cora Z7-07S[19]. This board produced by Digilent is a great sub-100€[20] Zynq Development board. If we look at figure 6 we can see that it has a single-core CPU and a little less Look-up Table (LUT), Flip-Flops and BRAM than the Red Pitaya but enough to implement the project's scenarios. Other board that offer great options are the parallella board by Parallella[21] and at a higher price range Digilent's Zybo board[22]. Even though the parallella board was said to cost 99$, it could not be found for delivering to Spain for less than 120$ without including shipping costs.

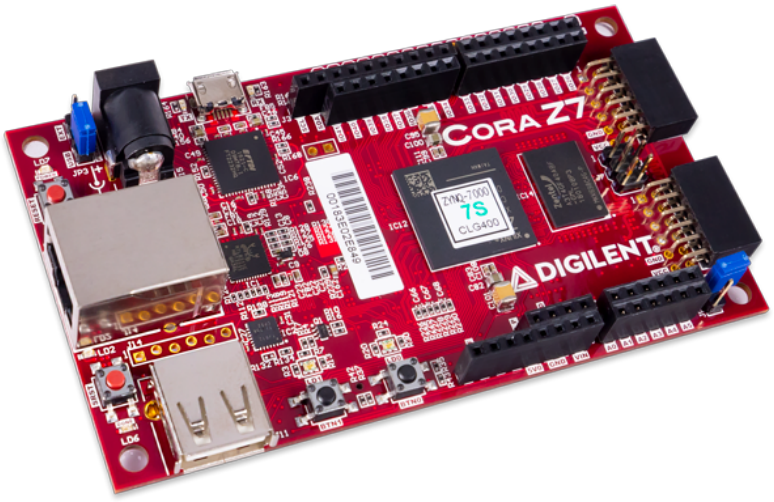


Figure 8: Cora Z7-07S development board

Figure 8 shows the board that was chosen. The Cora board also has a DDR3 Interface to connect to the onboard 512MB 16 bit SRAM.

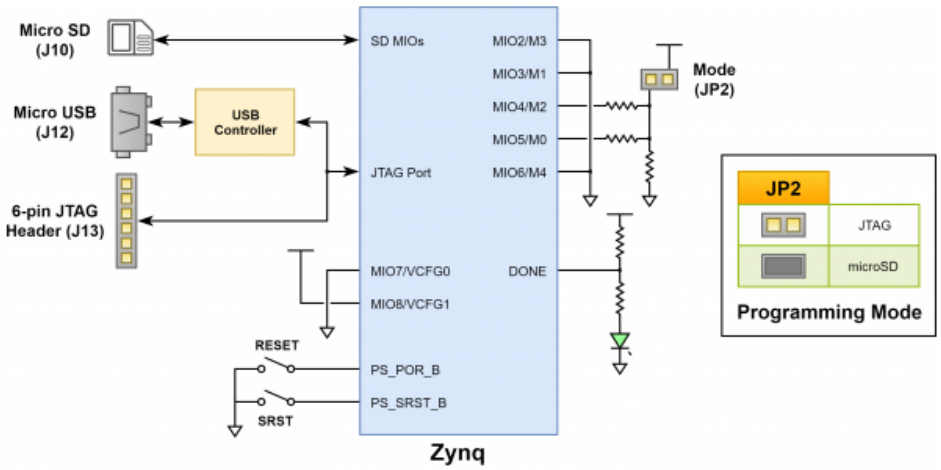


Figure 9: Cora board booting modes[19]

Furthermore, the Cora board has two different booting modes as shown in figure 9. Through the jumper JP2 we can select if we want to boot from the SD Card slot or through the JTAG interface. For this project, we exclusively booted through the JTAG interface. Another exciting feature is that the Micro USB port J12 can act as a Serial Interface and JTAG Interface, while also supplying power to the board. This dramatically

reduces the number of cables needed to program the Cora board.

The Cora Board has a *XC7Z007S-1CLG400* with a speed grade of -1, the same as in the Red Pitaya. Nevertheless, we can see that the Cora board is drastically different from the Red Pitaya in the sense that it lacks any SMA and onboard DAC and ADC. But due to budget limitations, it was decided to use this board and simulate the data traffic that would be seen in the Red Pitaya using a Traffic Generator**??** in the PL to generate a stream of data.

2.4 AXI PROTOCOL

The AXI protocol is part of the AMBA specification. It is a parallel, high performance, high-frequency protocol normally used for on-chip communication. It is based on a Master-Slave model of communication and burst transfers. The different types of AXI interfaces[23] we can find in the Zynq device are the following:

- AXI3
- AXI4
- AXI4-Lite
- AXI-Stream

These different types are explained in detail in the AXI reference guite[23]. Nevertheless, to understand some concepts applied when designing the scenarios, we must explain some key concepts.
AXI interfaces can be subdivided into two different categories: Memory-mapped and Stream interfaces. Memory-mapped interfaces allow read and write operations with different destination addresses, which may be used for example to access to a specific address in memory or to configure a specific register in a peripheral. We can think of Stream interfaces as a one way, one-lane road. Operations are only performed between the two endpoints and in one direction, so there is no need to use addresses. This makes the AXI-Stream protocol much more straightforward than the full AXI protocol.

We will refer to the specific signals the AXI-Stream protocol uses in different points of this report, the AXI4-Stream Adoption and Support chapter on page 45 of the AXI reference guide has a description of each of the signals the AXI-Stream protocol uses. The main ones to be discussed are TVALID, TREADY, TDATA and TLAST.

2.5 DMA

A DMA Controller is a component that allows the movement of blocks of data between a peripheral and memory, peripheral and peripheral or memory and memory.

A DMA is generally used in conjunction with a Processor, as shown in figure **??**. If used correctly, it can drastically improve the performance and reduce the power consumption of a system as the DMA is specifically designed to transfer data. Both the DMA and processor share the same buses.

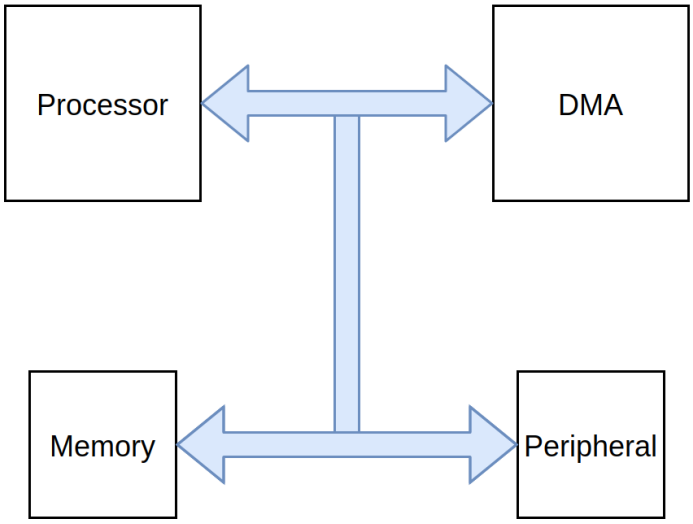


Figure 10: DMA and Processor conceptual connection.

In order to properly configure the DMA, the processor must specify an origin and a destination address to the DMA, which can typically be configured through the on-board Registers. Furthermore, a transfer length will also be configured. After the CPU indicates the DMA it can start a transfer, it has to be able to know when the DMA has finished said transfer. There are two methods to check this: Polling or Interrupts.

When polling, the processor will have to continuously check the DMA's status registers to find if the transfer has been completed. In some applications where CPU time is critical, this method may not be advisable.
When using interrupts, the DMA will perform an interrupt request once it has finished the transfer. This interrupt request will be sent to the interrupt controller which will decide depending on the interrupt's priority when to assert it. When the CPU is interrupted, its program counter will be moved to the Interrupt Service Rountine (ISR); here the processor will be able to reconfigure the DMA if needed or it will perform any other action. When the ISR finishes, the program counter will return to the code address that was being executed when the interruption occurred. For time-critical systems, this method is more advisable as the processor does not have to wait to check the DMA registers periodically. There is no magic rule to know what method to use, it depends on the application, and each case must be analysed separately.

The Cora board has a DMA in the PS, but this one is not used in this project, you can instantiate various DMA IPcores in the PL to move data from different regions of memory, accessing to the PS through the interfaces specified in 2.2.2. These have been used to implement the scenarios in this project. Some of the IPcores Xilinx offers will be explained in section 2.6.

2.5.1 *AXI DMA*

The AXI DMA is an IPcore[24] produced by Xilinx that enables you to instantiate a DMA in the PL. This IPcore allows you to perform transfers from an AXI-Stream interface to a memory-mapped interface or from a memory-mapped interface to an AXI-Stream Interface. When Configuring this block, you can modify several parameters to adapt it to your specific application:

- Width of buffer length register: Specifies the number of bits valid to configure the RX length.
- Address width: Specifies width of the Address bus.
- Burst Size: Maximum size of burst cycles for RX or TX.
- Unaligned Transfers: Enabling this allows byte-level alignment for transfers to or from memory-mapped peripherals.

When speaking about the AXI DMA, the RX channel refers to the channel that performs S2MM transfers and the TX channel refers to MM2S transfers.

Family	Speed Grade	Fmax (MHz)		
		AXI4	AXI4-Stream	AXI4-Lite
Virtex®-7	-1	200	200	180
Kintex®-7		200	200	180
Artix®-7		150	150	120
Virtex-7	-2	240	240	200
Kintex-7		240	240	200
Artix-7		180	180	140
Virtex-7	-3	280	280	220
Kintex-7		280	280	220
Artix-7		200	200	160

Figure 11: DMA Maximum frequencies for different speed grades.

Figure 11 has been extracted from the AXI DMA's datasheet[24]. We can see that for the Cora board, that has an Artix-7 FPGA with a speed grade of -1 the DMA can not be run at more than 150Mhz.

The DMA IPcores Xilinx offers can work on a mode called Scatter Gather, for more information about this mode consult appendix a.

2.5.2 *Other DMAs*

We will briefly describe three more DMA IPcores that Xilinx offers as they have many similarities to the AXI DMA. In this project only the AXI DMA and AXI CDMA were

used as the others didn't provide features that were useful for the proposed scenarios explained in section 3.

The DMA IPcores are the following:

1. AXI CDMA: This IPcore[25] is very similar to the AXI DMA, the main difference is that this one is exclusively for Memory Mapped to Memory Mapped (MM2MM) transfers. All the configuration parameters are equivalent to those explained in section 2.5.1.
2. AXI Video Direct Memory Access (VDMA): The VDMA is especially designed to transfer video data providing transfers from AXI-Stream video interfaces to memory-mapped interfaces or from memory-mapped interfaces to AXI-Stream Video interfaces.
3. AXI Multi-Channel Direct Memory Access (MCDMA): As the name indicates, this IPcore allows to perform the same type of transfers as the normal DMA but with more channels. If we refer to appendix a, the limitation of working with an MCDMA in Scatter-Gather mode are explained.

2.6 OTHER RELEVANT IPCORES USED

This section will explain the different IPcores used to implement the different scenarios in order to understand how they can be configured for achieving good performance and to avoid errors.

2.6.1 *Processor System Reset*

The Processor System Reset block is used to handle the reset for a given system; it can handle various input reset signals and can be useful to perform resets on other blocks. When a reset is detected in one of its input pins, it will perform resets to the peripherals connected to it for a configured period of time.

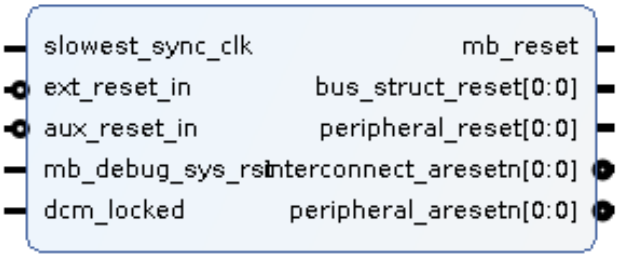


Figure 12: Processor System Reset Block

From figure 12, we can see that the IPcore can perform resets on different types of blocks. Furthermore, we can see that it receives as an input to the system's clock.

2.6.2 *Slice*

The slice block could be described as a demultiplexer of variable input and output length where the select pins can only be configured when the block is instantiated (not at run time).

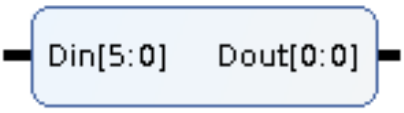


Figure 13: Slice block

We can see in figure 13 that the slice is used for normal non-AXI signals.

2.6.3 *Concat*

The Concat block can be used to concatenate various signals of different lengths.

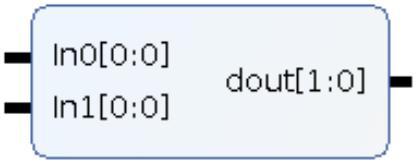


Figure 14: Concat block.

2.6.4 *Const*

The Const block is used to produce a constant logic level, either '0' or '1'. The output length can be modified to produce a constant value on a bus.

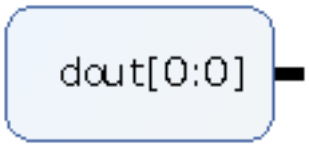


Figure 15: Const block.

2.6.5 *Clocking Wizard*

The Clocking Wizard works with the PL's PLLs or Multi-Corner Multi-Mode (MCMM). It receives one or various input clocks of a given frequency to produce output clocks of different or equal frequencies. Different parameters, such as output Jitter and phase, can be configured.

The *locked* output signal shows the status of the PLL and will activate when the output clocks are locked to the configured frequency.

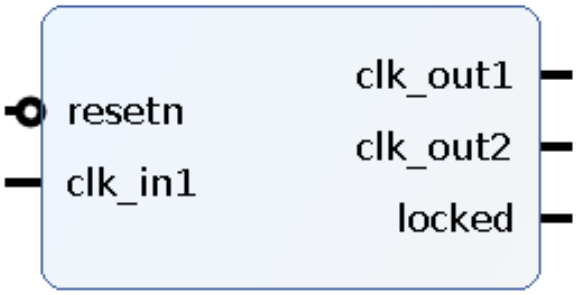


Figure 16: Clocking Wizard Block

2.6.5.1 *AXI Interconnect*

The AXI Interconnect is used to connect one or more AXI Memory-mapped masters to one or more AXI Memory-mapped slaves. Different priority labels for the different masters can be configured in this block.

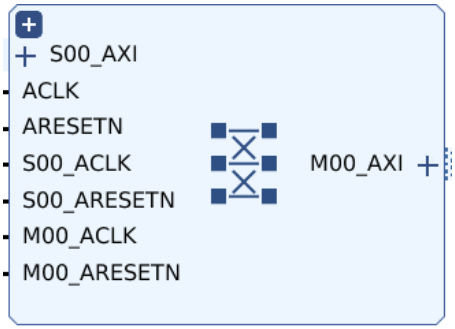


Figure 17: AXI Interconnect block

Figure 17 shows us how the AXI Interconnect needs each clock and reset signals for each of the AXI interfaces connected to it. It also has its own clock and reset signals.

2.6.6 *FIFO Generator and AXI-Stream Data FIFO*

The FIFO Generator is a block that allows configuring various types of FIFOs, whilst the AXI-Stream Data FIFO[26] is a specific type of FIFO that has two AXI-Stream interfaces, one for reading and another one for writing to the FIFO. One can use a FIFO Generator block to build a FIFO that acts like an AXI-Stream Data FIFO. Configuring the depth and width of TDATA allows configuring the size of the FIFO.
One use of the FIFO blocks is to separate clock domains. Both read and write channels can share the same clock or can have different clocks. When one only clock domain it defines, we can read the amount of data stored in the FIFO in bytes when the option is configured. Nevertheless, when using different clock domains, we will have to read the *write data count* and *read data count* registers to calculate the amount of data stored in the FIFO.

Furthermore, we can see that in figure 18, we also have enabled the overflow and programmable empty signals. The overflow signal will activate when the FIFO Overflows and the programmable empty signal activate when the amount of data falls behind

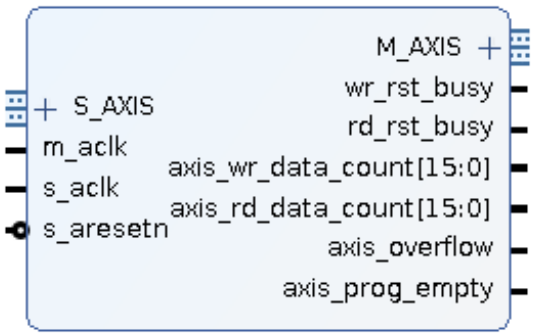


Figure 18: FIFO Generator block configured to act as an AXI-Stream Data FIFO

a specific value. This value can be set when instantiating the block, or a configuring bus can be enabled to configure this threshold from a different block. The Underflow and programmable full signals are also present in the FIFO but can not be activated simultaneously with the overflow and programable empty signals.

Another option we have for configuring the FIFO is the type of memory to use. There are two available options: BRAM or Distributed Random Access Memory (RAM). The latter uses LUTs as storage devices. This limits the amount of data that can be stored and also limits the frequency that the memories can be run at due to potential problems with clock path delays when high frequencies are needed.

2.6.7 *AXI GPIO*

The AXI GPIO block[27] is used to read or write signals from an AXI interface. An example of this could be reading a FIFO's overflow bit in the PL from the PS, as the PS can only interface the PL through AXI interfaces as discussed in section 2.2.2.

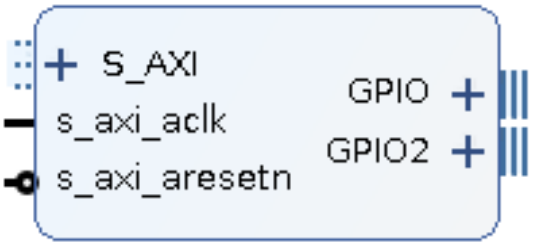


Figure 19: AXI GPIO Block

As shown in figure 19, the block can be configured to have two GPIO channels and can be used with the Slice or Concat blocks to interface multiple blocks with one only GPIO instance. Furthermore, an interrupt out signal can be configured for input GPIO channels. This interrupt signal will activate when in a GPIO input channel, a bit changes state.

2.6.8 *Traffic Generator*

The Traffic Generator[28] can be used to simulate different types of traffic. In this project it has been used in streaming mode to simulate AXI-Stream traffic.

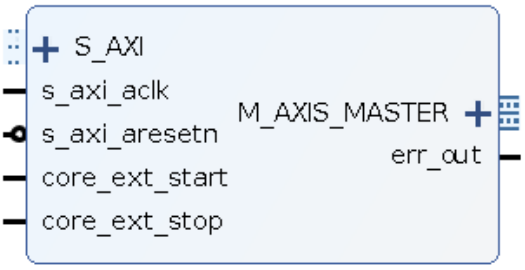


Figure 20: Traffic Generator block configured in streaming mode

In streaming, master only mode, the traffic generator will transfer a series of packets. Each packet consists of beats of data where each beat of data has the same width as TDATA. There are a series of parameters that are important when configuring the Traffic Generator for performing transactions:

- Length: The length defines how many beats of data will be transferred in one packet. Where a value of 0 refers to 1 beat of data, a value of 1 refers to two beats of data and so on.
- Delay: Defines the delay in clocks that will be applied between packets.
- Transfer Count: Defines the total number of packets to be transferred, limited to 16 bits.

Starting or stopping the traffic generator can be done either through the AXI slave interface or through the external *core_ext_start* and *core_ext_stop* signals.

2.6.9 *Block Memory Generator*

Through the block memory generator[29] we can instantiate a BRAM with different features such ad dual-port capabilities. We can also configure the depth of the memory and the width of TDATA to change the size of the BRAM.

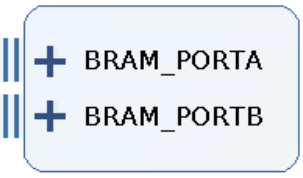


Figure 21: dual-port BRAM

By looking at figure 22, we can see that the interfaces offered by the block memory generator are BRAM native interfaces, which means that we need a block that can access these.

2.6.10 *BRAM Controller*

The BRAM controller[30] allows to interface a BRAM from an AXI interface, allowing peripherals such as the CDMA to access the BRAM seamlessly.

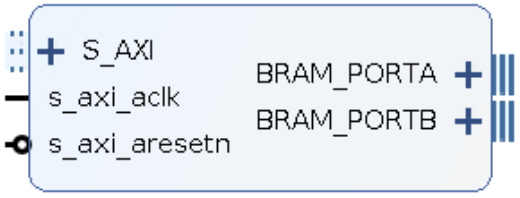


Figure 22: BRAM controller

2.7 IDE

To develop the different scenarios, two different IDEs were used: Vivado Design suite[31] and the Vitis Unified Software Development Platform[32]. Being released in November of 2019, it has been challenging to adapt to the new up-scaled version of the Xilinx SDK and some bugs were found and communicated to Xilinx during the implementation of the project. Nevertheless, it has been rewarding to work with such a new tool.

Vivado is used to program the PL, while Vitis is used to program the PS. It is important to mention that previous versions of the 2019 Vivado will not work with the Vitis development platform. This is because from the 2018 version of Vivado to the 2019 version of Vivado, the types of files created when exporting hardware were changed from .hdf to .xsa. Vitis can only work with .xsa files.

In order to view how Vivado should be used, it is recommended to read Joaquim Porte's master's thesis[6]. Furthermore, to work with the Vitis platform, we can refer to the Vitis guide [32] and a guide for embedded programming with Xilinx SDK[33]. Due to being both based on Eclipse, both the Xilinx SDK and Vitis platform behave very similarly so there should not be any significant difference when transitioning from one to the other.

2.8 RELATED WORK

Before starting the development of this project, it was beneficial to look into previous research papers on similar topics to see what was being done in the area of bare-metal Zynq programming for IoT. Some of the research papers found will be discussed.

In the paper, **DMA implementations for FPGA-based data acquisition systems**[34] written by Wojciech M. Zabołotny, two different scenarios for data acquisition are proposed. One of them for video acquisition and the other for data acquisition. The PS firmware was run in Linux, but it was still useful to see the different strategies for implementing the hardware in the two proposed scenarios. Furthermore, in this research paper, all the scenarios were simulated.

Another paper that was read was **Performance Evaluation PL330 DMA Controller for Bulk Data Transfer in Zynq SoC**[35] by Apurva Choudhary et al. In this paper, Choudhary et al. analyse a DMA driver with the PS firmware running Petalinux. Also, the results are simulated using the QEMU simulator and using the actual Zynq device. An exciting part of this paper is the comparison between transfer rates with or without the DMA. Their results end up showing how using a DMA drastically increases the throughput of data but what is even more peculiar is that when they use the DMA, the total amount of that that they transfer does not have a linear relationship with the data rate. In the conclusions, they mention how data transfers are time-critical but it would've been interesting to see a comparison when using the Linux operating system or bare-metal.

Finally, **The performance comparison of the DMA subsystem of the Zynq SoC in bare metal and Linux applications**[36] by Michał Fularz et al. compare the DMA's performance in bare metal and Linux operating system applications. In the paper, it is concluded that for small amounts of data the bare-metal application is superior but for large amounts of data software architecture must be chosen with care as the time taken for configuring the DMA can drastically affect the performance of the system. Furthermore, the smaller time required to configure the DMA in a bare-metal application makes it more suitable than when in the Linux operating system. Furthermore, it is stated that for vision-based systems using Linux is preferable, but both Linux and bare-metal applications reach the maximal obtainable performance for large amounts of data.

3 DESIGN AND TECHNICAL IMPLEMENTATION

This chapter will explain how the different scenarios were configured and implemented. The implementation of each scenario will be described separately. Moreover, the PS(Software) and PL(Hardware) will also be described in separate sections to ease the comprehension of each scenario. A total number of three scenarios were implemented and analysed. The first two serve as a comparison between the DMA and CDMA respectively. In these, data will be only moved from one region of memory to another and different metrics will be extracted for analysis in section 5.

The third scenario makes up the RX channel of the GRITS NVIS project and will try to improve on the performance of the existing design. As mentioned in section 2.3, to emulate the flow of data from the third scenario a traffic generator will be used. This traffic generator will be configured in different ways, varying its length and delay parameters in order to simulate different flows of data. Two working modes will be defined for the traffic generator: Data peaks and Constant data flow. These two modes enable the implemented scenario to be tested in different conditions.

It has been very useful to divide this project into different scenarios of increasing complexity as it has allowed for a natural progression and comfort when using Xilinx's tools.

What all the scenarios have in common is the metrics measured and the general structure of the hardware and software, except for the third scenario which needed a more complex design in order to correctly function. These include:

- Total time: total time taken to complete the transfer measured in clocks
- Configuration time: Total time spent configuring the DMA or CDMA, measured in clocks.
- Total data: Total number of bytes configured to be transferred.
- Dropped bytes and Drop Rate: A measure of how much data is being lost.
- Latency: Time in clocks required to transfer one word of data(4 bytes). Measured from the time the DMA is started to when the transfer ends with the last DMA interrupt.
- FPGA resources: PL resources needed to implement the different scenarios.

For the first two scenarios, the effect that enabling or disabling the data cache has on the performance of the system was also measured. Furthermore, for the third scenario, additional metrics were measured: Total number of overflows and Number of DMA configurations.

The different scenarios implemented will be now described, explaining the PS and PL separately. The PL was designed and implemented using Vivado's IPcores and the PS was implemented programming in C using the Vitis platform. Xilinx offers some bare-metal libraries to control different peripherals. These were used as needed.

Furthermore, for all the scenarios UART0 was used as the stdout for the PS, meaning that we could output debugging messages and test results through the onboard micro-USB port and visualise the sent data on a computer. The software used to do this was picocom for Linux but any software that can read from a USB port can be used. UART0 was used in MIO[14..15] for these scenarios.

3.1 CLOCK GENERATION

To produce different clock frequencies, the clocking wizard IPcore was used. A problem that appeared was that at frequencies near 150MHz, setup timing violations were being produced in the internal paths of the different DMA and CDMA blocks. This was resolved by using the Zynq PL Fabric clock when these timing violations were being produced.

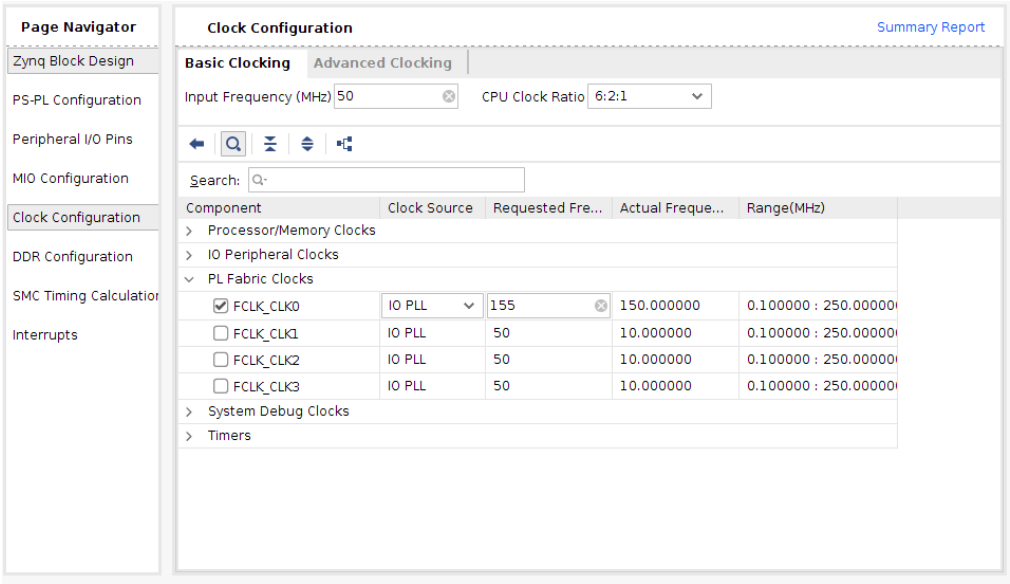


Figure 23: Scenario 1:PL Fabric clock

In figure 23 it can be seen how the PL Fabric clock is configured through the ZYNQ Processing system block.

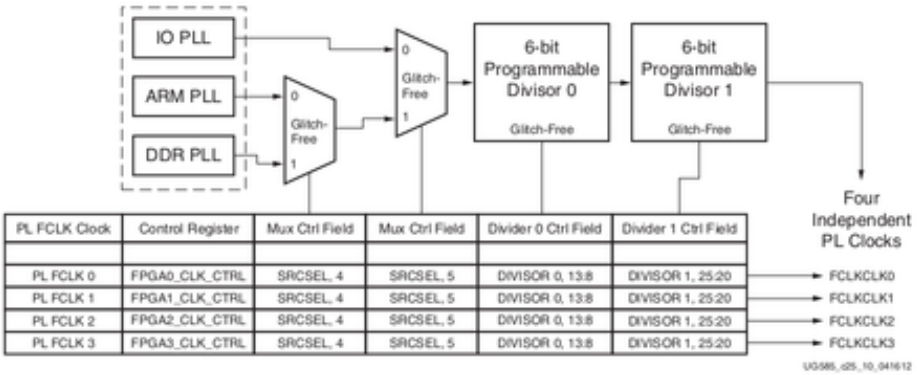


Figure 24: PL Fabric clock Generation

Figure 24 shows how the PL Fabric clock is generated, from figure 23 we can see that the IO PLL clock source has been selected, which is a 500MHz clock. This frequency was calculated by setting the PL Fabric clock to output a certain frequency and then checking the value of the dividers from the First Stage Boot Loader (FSBL) generated after exporting the hardware to Vivado, then calculating the IOPLL frequency. The limitation of the PL Fabric clock is that we only have two 6-bit divisors[11], this means that many of the frequencies that were needed to be tested were not available through the PL Fabric clock. For this reason, at lower frequencies, the Clocking wizard was used when it did not produce timing violations. This occurred in all of the scenarios.

3.2 SCENARIO 1: DMA AND FIFO

The first scenario performs two types of transfers between an AXI-Stream FIFO and the onboard SRAM. These are MM2S and S2MM transfers. A MM2S transfer will move a specific amount of data from the SRAM through the DDR interface to the FIFO in the PL. On the other hand, the S2MM transfer will move data from the previous full FIFO in the PL to the SRAM.

Figure 25 shows the general view of the system. Due to the way the Cora board is built, to access the SRAM from the PL we have to cross the PS. Some boards have direct connections from the PL to the SRAM, which allows you to instantiate a Memory Interface Generator block[37] in the PL to access the SRAM but this is not the case in the Cora board. We will continue explaining in further detail the implementation of the scenario.

3.2.1 *PL*

This scenario has two different clock domains, the PS clock and the PL clock. To provide the PL with a clock we used the Clocking Wizard block to produce a 150MHz signal from the systems 125MHz system clock. If we look back at figure 25, the connection between the DMA and SRAM has to cross the PS as previously explained. To do this we use the interface HP0 configured with a width of 64 bits.
Furthermore, we can see that there are counters instantiated in the PL, these counters run at the PL's frequency and are used to perform the time measurements needed to analyse the system.

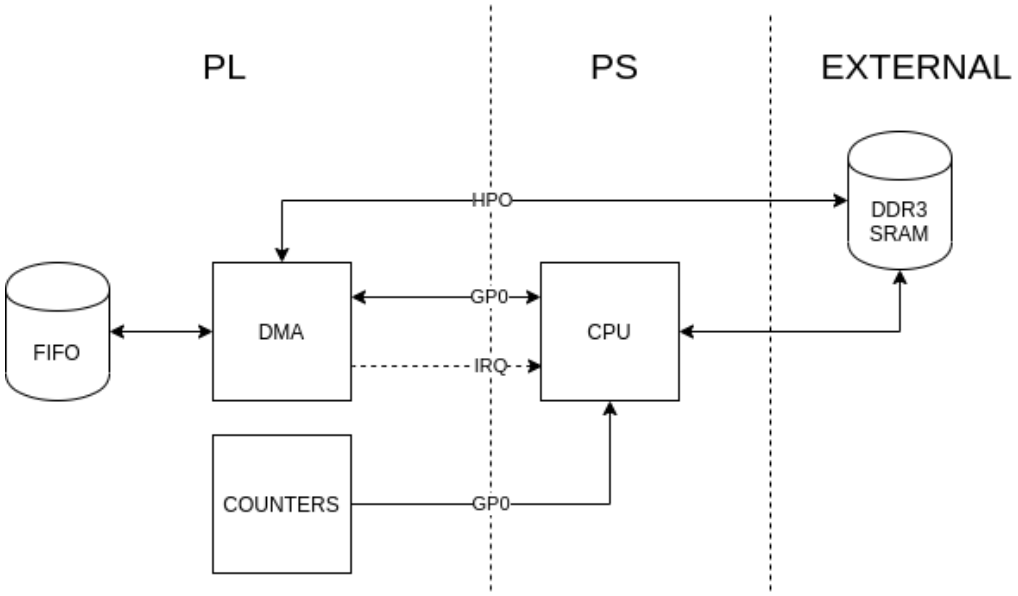


Figure 25: General view of Scenario 1

Figure 98 in appendix b shows the general interface view in Vivado. The first thing that should be mentioned is that the block called ZYNQ Processing System allows us to configure features of the PS that we will need to properly design the PL. This includes activating interrupts, configuring interfaces like the HP0 and GP0 or configuring the peripherals to route them to different MIOs. This figure shows us a reduced version of the system only showing the AXI interfaces in the design. If we look closely at the AXI DMA instance, we can see that the two master interfaces for the MM2S and S2MM transfers are connected to the slave HP0 interface in the Zynq PS through an AXI Interconnect, which allows the slave interface to be shared by the two masters.

We can also see that both the Master and Slave AXI-Stream interfaces of the DMA are connected to the Slave and Master AXI-Stream interfaces of the FIFO respectively. Furthermore, there are four instances of AXI GPIO blocks. These are uses to control and read the different counters, to control different signals in the design including the on-board LEDs (used for debugging purposes) and to read the amount of data stored in the FIFO. To configure the different AXI GPIOs and the DMA, we need to be able to access them from the PS. For this, we used the Master GP0 interface in the Zynq Processing System. Connecting these interfaces to the different Slave interfaces through another AXI Interconnect.

If we take a look at figure 25 in appendix b we can see the complete view of the implemented design. It can now be seen that to be able to control the different signals in the counters Slice blocks have to be used. Furthermore, the interrupts lines for the DMA transfers have been connected to the PS through the Concat block.
To perform resets, we have two processor system reset blocks, one for the FIFO and

one for the rest of the design.

We will now show the GPIO map of the system to gain a better understanding of how the design has been built:

- GPIO 0
  - Channel 1 [IN]: FIFO data count [15..0]
- GPIO 1
  - Channel 1 [IN]: Counter 1 Value [31..0]
  - Channel 2 [IN]: Counter 2 Value [31..0]
- GPIO 2
  - Channel 1 [OUT]

[0]: Counter 1 Enable [2]: Counter 2 Enable [4]: FIFO NReset
[1]: Counter 1 Reset [3]: Counter 2 Reset

  - Channel 2 [IN]

[0]: FIFO Read Busy [2]: FIFO Underflow
[1]: FIFO Write Busy [3]: FIFO Overflow

- GPIO 3
  - Channel 1 [OUT]: RGB LEDs [5..0]

Cell	Slave Interface	Slave Segment	Offset Address	Range	High Address
axi_dma_0					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_dma_0	S_AXI_LITE	Reg	0x4040_0000	16K	0x4040_3FFF
axi_gpio_0	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
axi_gpio_1	S_AXI	Reg	0x4122_0000	64K	0x4122_FFFF
axi_gpio_2	S_AXI	Reg	0x4123_0000	64K	0x4123_FFFF
axi_gpio_3	S_AXI	Reg	0x4124_0000	64K	0x4124_FFFF

Table 1: Address map for environment 1

Table 1 shows the address map for the implemented design. It shows how different peripherals are mapped to memory, from the DMA and the PS. This will be used to access and configure the different peripherals. For example, we can see that AXI GPIO0 has been mapped to address 0x41200000 and has a range of 64K which means its end

address is 0x4120FFFF. Furthermore, we can see a section called *HP0_DDR_LOWOCM*. This section maps the on-chip memory and SRAM. If we look at the address map in chapter 4 of the Zynq reference manual[11] we will see that in order to access the SRAM through the DDR and CPU we can use the addresses 0x00100000 to 0x1FFFFFFF.

The configurations of the individual blocks used will be now explained:

3.2.1.1 *ZYNQ Processing System*

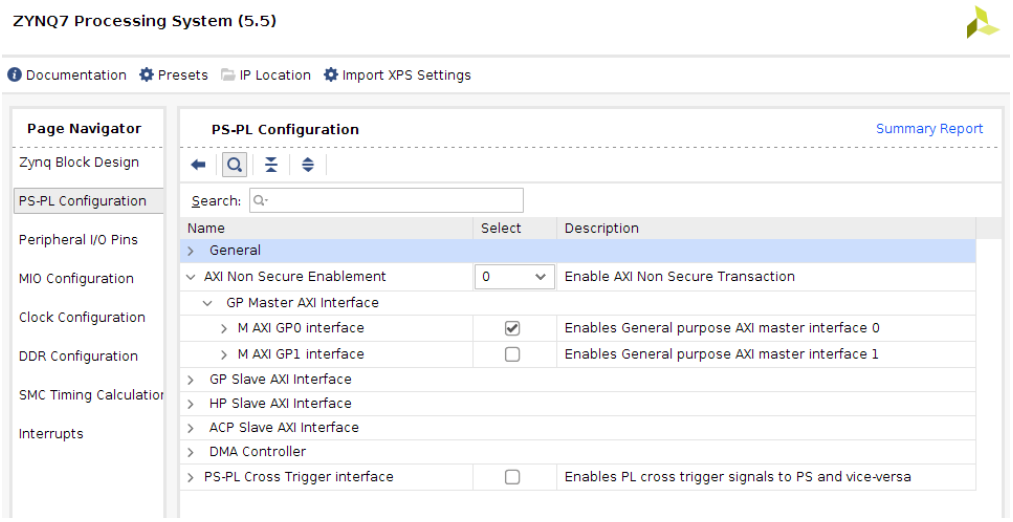


Figure 26: Scenario 1:Zynq Configuration, Enable Master GPO interface

Firstly, the Master GP0 interface was enabled as shown in figure 26.

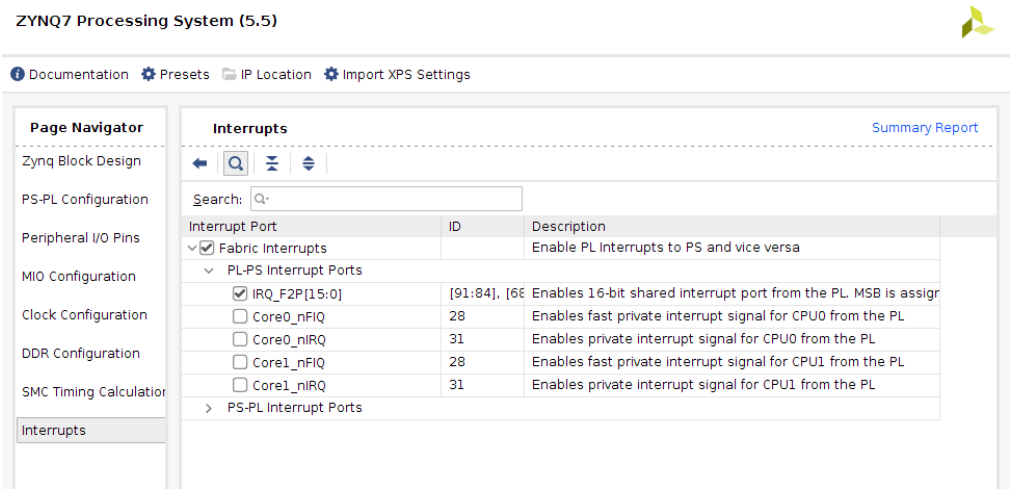


Figure 27: Scenario 1:Zynq configuration, Enable Zynq Interrupts

Secondly, we activated PL-PS interrupts as shown in figure 27. We will use two interrupt lines, one for S2MM interrupts and the other one for MM2S interrupts. Furthermore, it is good practice to verify that UART is enabled.

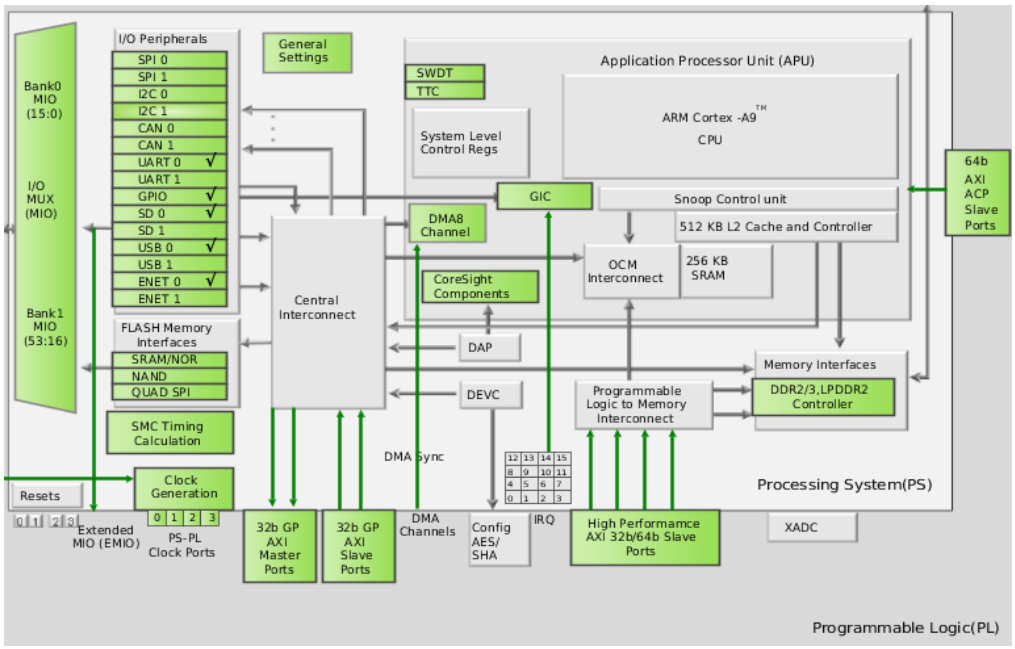


Figure 28: Scenario 1:Zynq configuration, check UART is enabled

In this case and all other scenarios, UART0 was used as seen in figure 28.

3.2.1.2 *FIFO*

FIFO Generator Summary

Selected Simulation Model

Interface Type :	AXI Stream
Model Generated :	Behavioral Model

Clocking Summary

Clocking Scheme:	Common Clock

AXI Stream Summary

Configuration Type :	FIFO	Memory Type: Block RAM
Application Type	Data FIFO	BRAM Resource (s) (18K/36K) : 0/30
Width/Depth	33 / 32770	Latency : 2

AXI Stream Additional Features Summary

Occupancy Data Count	Selected
Interrupt Flag (UnderFlow/OverFlow)	Selected / Selected

Figure 29: Scenario 1: FIFO configuration

Figure 29 shows the summary for the FIFO's configuration, where it can be seen that we configured it with a depth of 32K and a width of 4 bytes. We can also see that the overflow and underflow signals, as well as the occupancy count, were enabled.

3.2.1.3 *DMA*

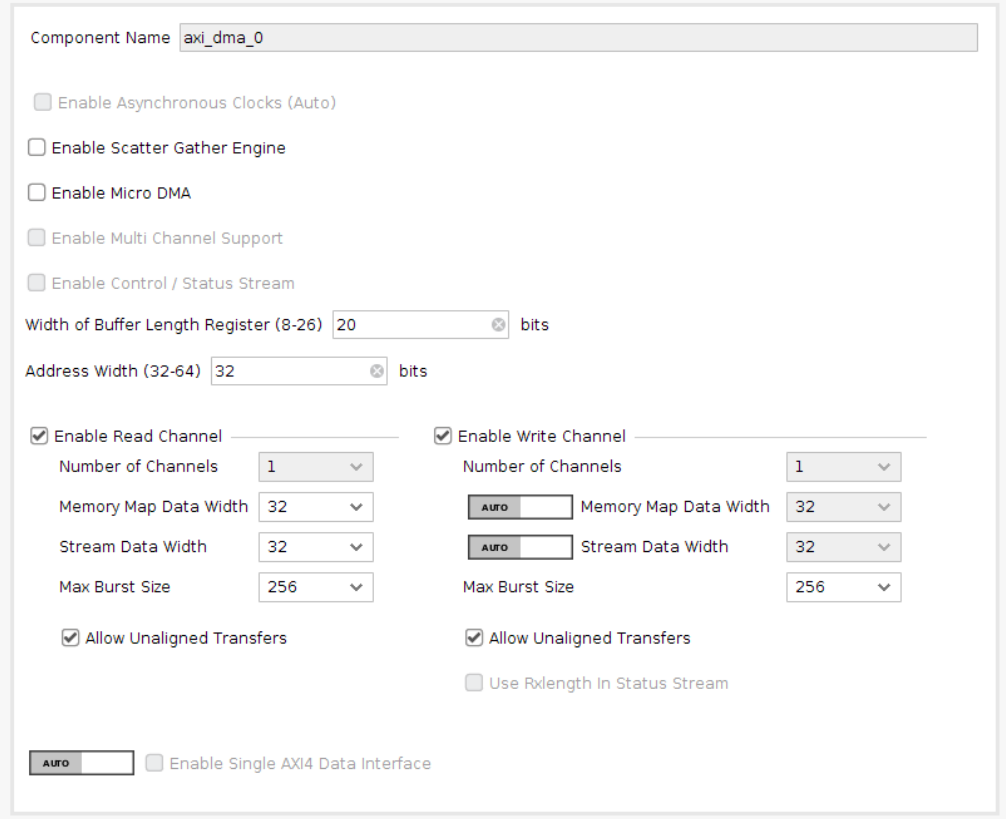


Figure 30: Scenario 1: DMA Configuration

From figure 30, we can see that the DMA was configured to allow unaligned transfers, with a burst size of 256 and a data width of 4 bytes. We also configured the buffer length register to 20 bits even though less could have been assigned. Some of these parameters will be modified when performing different tests.

3.2.1.4 *Counters*

The counters were configured to include Enable and Reset control signals, where the priority of the reset is higher than that of the enable. They were all configured to have a width of 32 bits, which at a maximum frequency of 150MHz would overflow at an on-time of 28.63 seconds which is more than enough.

3.2.1.5 *FPGA Resources*

After finishing the design of our scenario, it must be then synthesised and implemented before a bitstream file can be generated. The bitstream file is used to program the PL. After the process of implementation we get the next view:

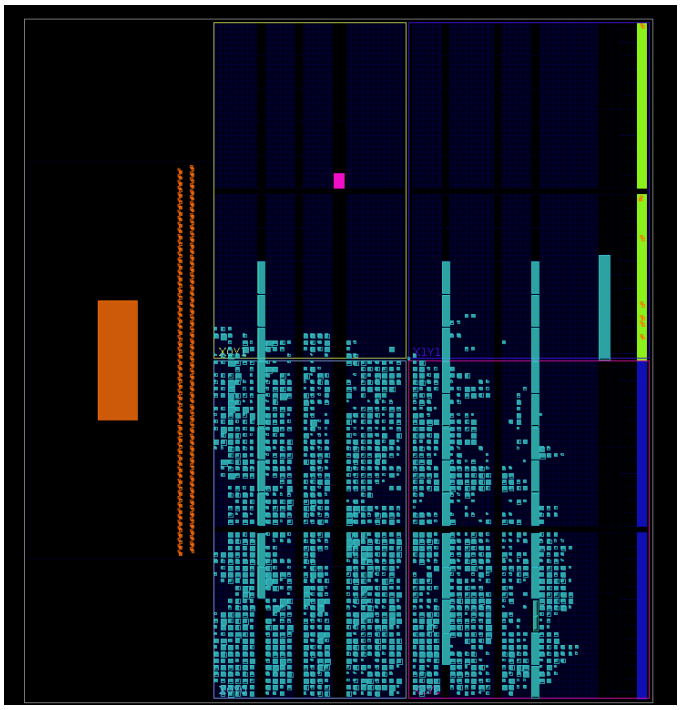


Figure 31: Scenario 1: Implementation view

Figure 31 shows us how our design has been implemented in the FPGA. If we zoomed in enough, each Configurable Logic Block (CLB) and LUT could be seen. The light blue regions represent the parts that have been assigned. Furthermore, at this stage, we also get information about the resources our design uses which in this scenario's case can be seen in table 2.

Resource	**Utilisation**	**Available**	**Utilisation %**
LUT	3663	14400	25.437502
LUTRAM	239	6000	3.9833333
FF	5342	28800	18.54861
BRAM	35	50	70
IO	7	100	7
BUFG	2	32	6.25
MMCM	1	2	50

Table 2: Scenario 1: FPGA Resources

3.2.2 *PS*

The software created to perform the different transactions in this scenario will be described through the use of flow charts. In the following flow charts, regions delimited by a dotted rectangle mean that a specific interrupt is enabled and so the code may jump to an ISR.

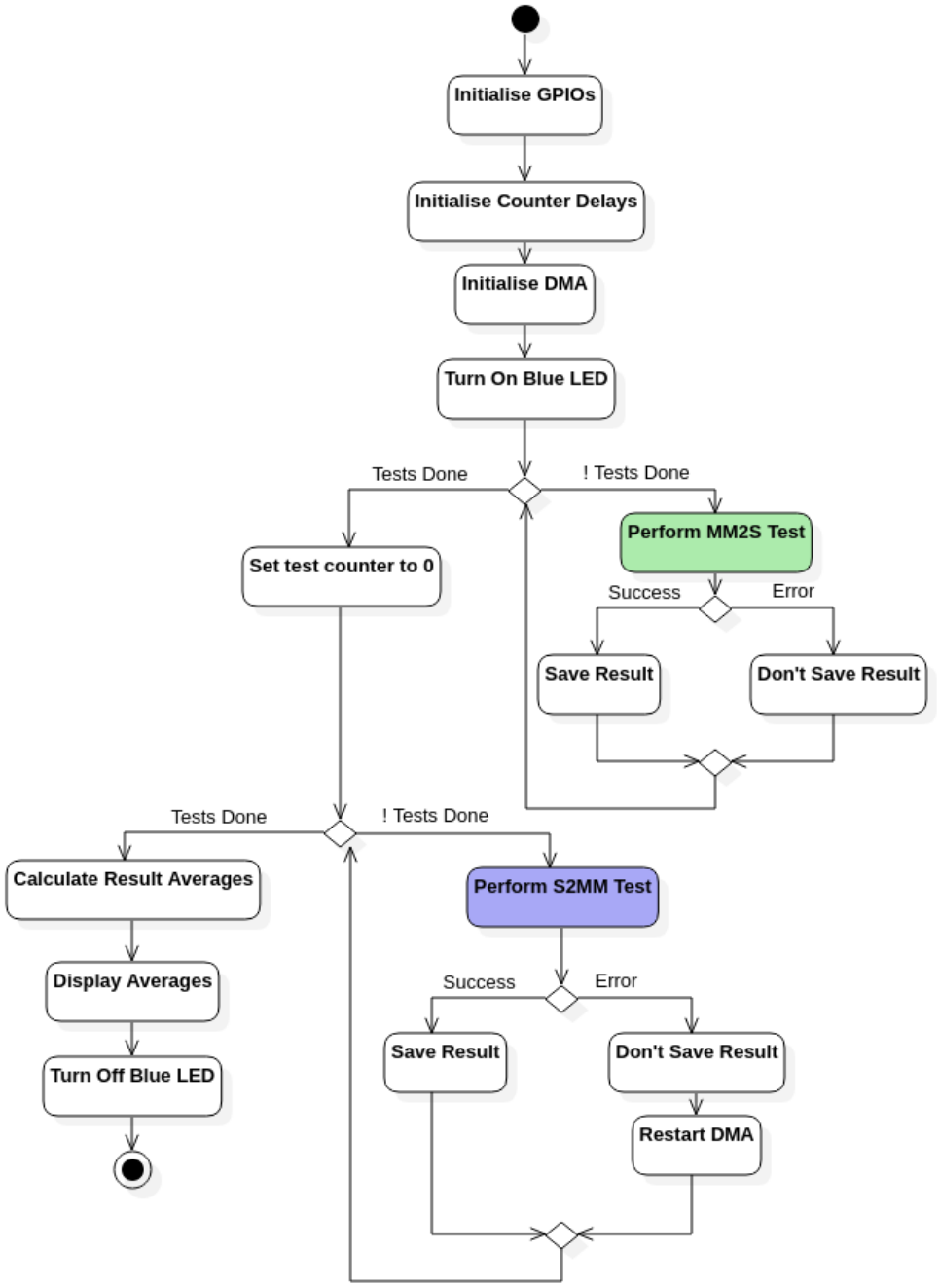


Figure 32: Scenario 1: General Flow chart

The flow chart in figure 32 shows the execution of the software required to perform different tests. The execution starts by initialising the AXI GPIOs in order to be able to interface with the counters and other blocks. After this, the counter delays will be calculated. The counter delay is a value that measures the number of clocks required to start and immediately stop the counter. This value is calculated 10 times and then averaged for each counter. This way, when time is measured with the counter we will not carry the clock's latency error.
Next, we will initialise the DMA, this includes configuring the interrupts by assigning the specific ISRs for each MM2S and S2MM interrupts. In order to configure the inter-

rupts the SCUGIC driver provided by Xilinx has been used.

After initialising the DMA we will turn on the blue LED, which indicates when a test is running. Subsequently, MM2S tests will be performed and the results will be saved, we will do the same for the S2MM test.
Once all the tests have been completed the result averages will be calculated and displayed. At this point, the blue LED will be turned off.
Each of the tests performed is composed of two sub-tests.

1. Latency and Configuration test: Latency and configuration times will be measured by transferring one word of data, which is equal to four bytes.
2. Throughput and Drop Rate test: The total time is taken and the amount of dropped bytes will be measured. The throughput will also be calculated using the total time taken.

3.2.2.1 *MM2S Test*

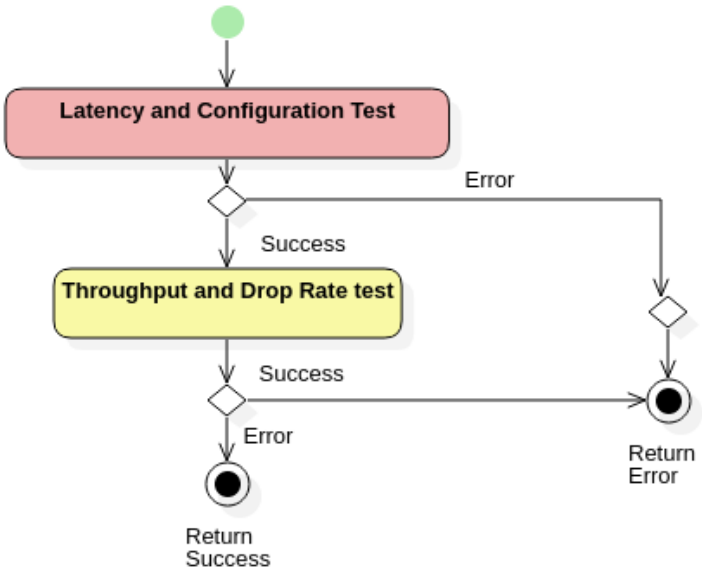


Figure 33: Scenario 1: MM2S Test Flow chart

As previously mentioned we can see from figure 33 how the tests are divided into two sub-tests.

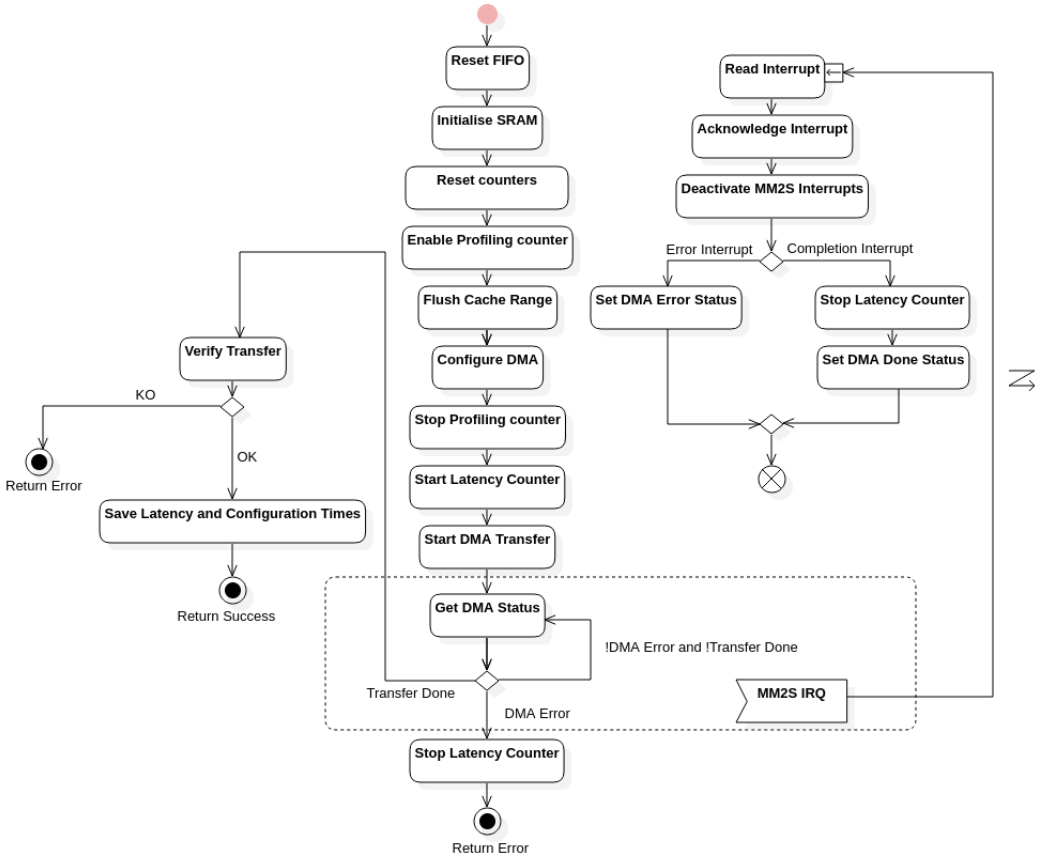


Figure 34: Scenario 1: MM2S Latency Test Flow chart

Figure 34 shows the steps taken to perform a latency test. To transfer one word of data from the FIFO to the SRAM we will first fill the FIFO with one word of data and we will clean the SRAM by writing 0s to each address. Next, the two counters will be restarted after which we will enable one of the counters which we will call the profiling counter for reference. This counter will measure the configuration time. After this, we will flush the Cache range needed. If we do not do this, we may find that when we need to read the data from the PS to ensure a correct transfer, the data read won't be the one transferred. After this, we will configure the DMA and stop the profiling counter, which will give us the configuration time. Next, the Latency counter and the DMA transfer will be started.

As we can see from the diagram we enter a region where a MM2S Interrupt Request (IRQ) can happen at any moment. At this point, we will loop waiting for the transfer to finish or for an error to happen. Instead of waiting, in other applications, we could use this spare time to perform other actions. When an interrupt occurs, the code will jump to the ISR where the interrupt will be read and acknowledged. If the interrupt type is detected to be an error interrupt a DMA error flag will be activated, which will end the loop and finish the test returning an error. But if the interrupt indicates a correct completion, the latency counter will be stopped and a Transfer Done flag will be activated.

Once the Transfer Done flag has been activated and the transfer finishes, the SRAM will be read by the CPU to verify the transfer. Furthermore, the DMA status registers and the FIFO error signals will be checked too. If there has been no mistake found we will proceed to save the latency and configuration times from the counters returning Success.

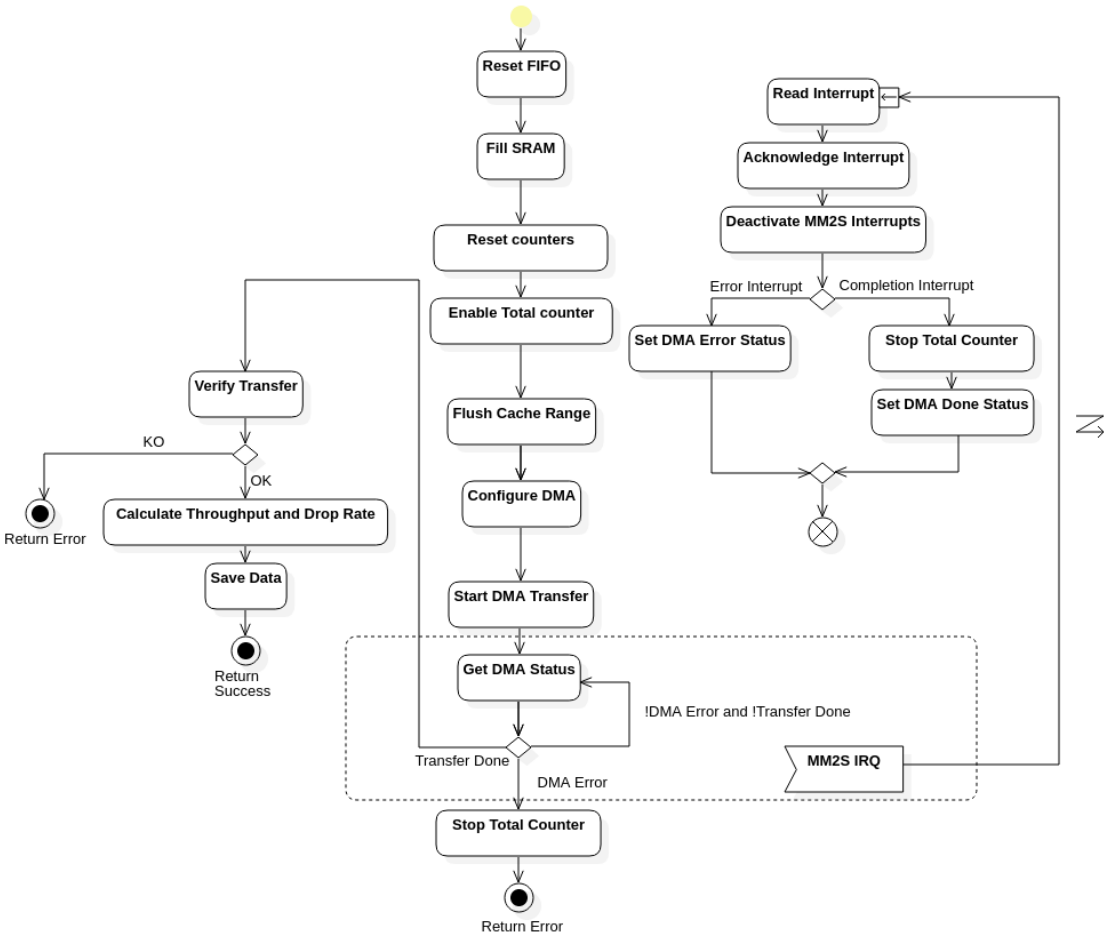


Figure 35: Scenario 1: MM2S Throughput test Flow chart

Figure 35 shows the flow diagram for the throughput and drop rate test. This is very similar to the Latency test. The main difference is the way the transfer is that instead of transferring one word of data we transfer 10KB of data. We can see that where we before enabled the counter for measuring the configuration time we now enable the counter to measure the total time taken. This counter will be stopped in the ISR when the DMA has completed the transfer successfully.

In order to verify the transfer we check that the correct amount of data was written to the FIFO using its occupancy signals, this data should be composed of an increasing sequence of numbers as this is with what the SRAM was initialised with. Once the transfer has been verified we can calculate the throughput as:

$$\text{THRP} = \frac{\text{NBYTES}}{\text{COUNT} \cdot \frac{1}{f_{PL}}} \tag{5}$$

Where NBYTES is the total amount of data transferred. The drop rate is calculated as:

$$DR = \frac{1}{NBYTES_{dropped}} \tag{6}$$

The calculation of the different parameters was performed in the ARM host. The ARM possesses an Floating-Point Unit (FPU) and was enabled using the compiler options *-mfloat-abi=hard -mfpu=vfpv3*. Nevertheless, after checking the disassembly of the produced code it seemed that the floats were being eliminated. To overcome this a floating-point function was implemented that was based on using two 32 bit variables to store the upper and lower parts of the number.

3.2.2.2 *S2MM Test*

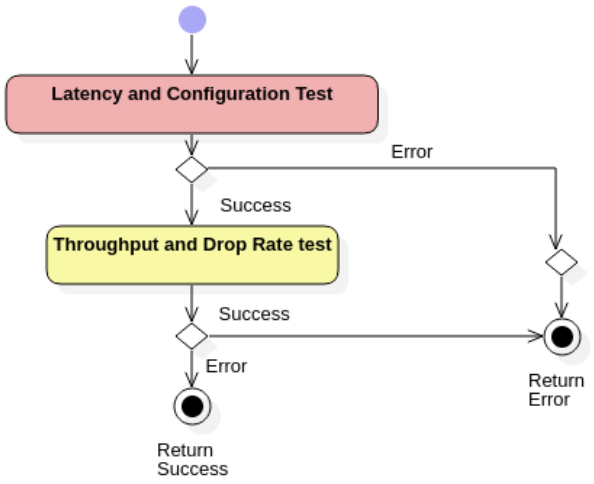


Figure 36: Scenario 1: S2MM Test Flow chart

For the S2MM tests, the same structure was followed, performing two sub-tests as shown in figure 36.

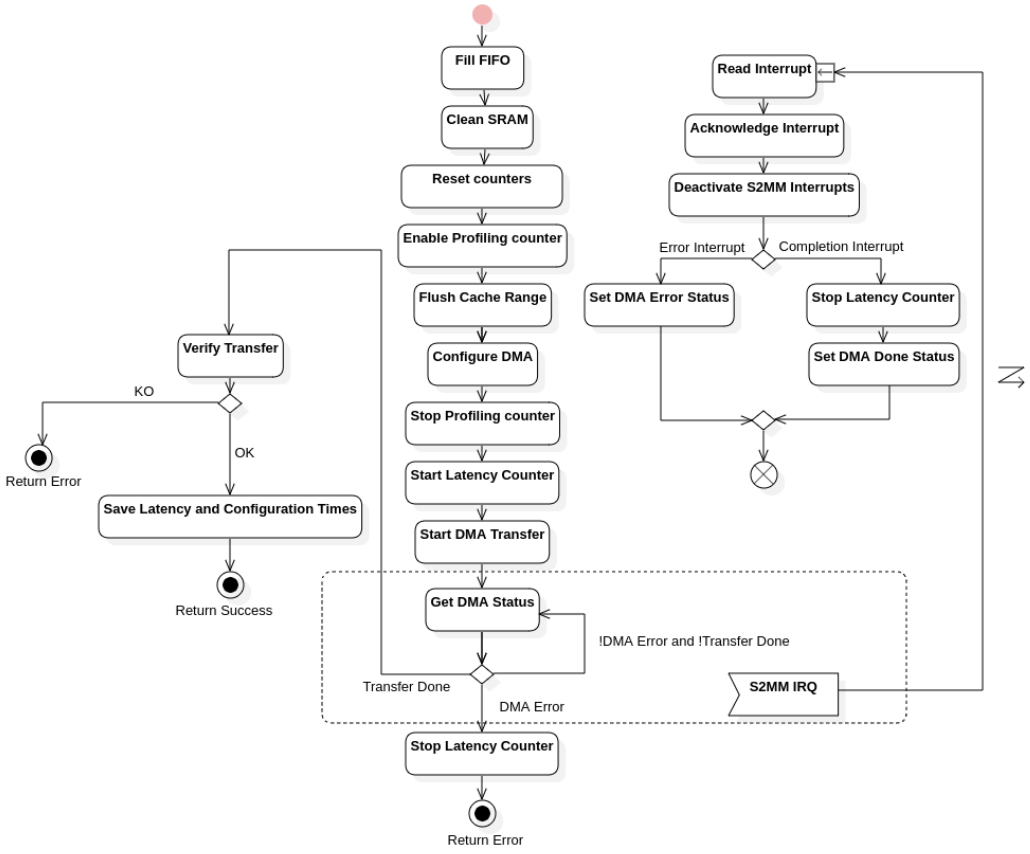


Figure 37: Scenario 1: S2MM Latency Test Flow chart

The structure for the S2MM Latency and throughput tests is very similar to the ones for the MM2S tests. If we look at figure 37 it can be seen that we will start by filling the FIFO with the amount of data to be transferred, which will be composed of an increasing sequence of numbers. After the transfer finishes, we will be able to read the SRAM to check if the data has been transferred correctly and we will also verify that the FIFO is empty.

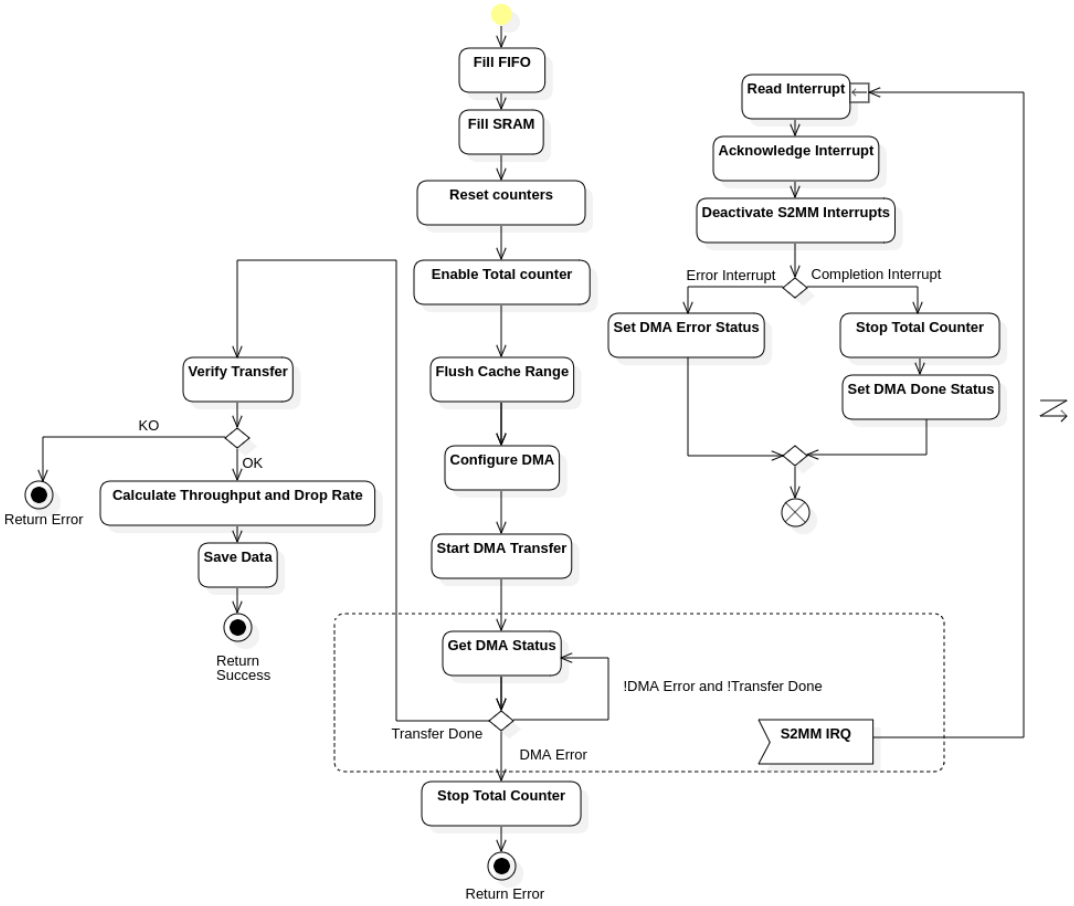


Figure 38: Scenario 1: S2MM Throughput Test Flow chart

Looking at figure 38 it can be seen how this test is analogous to the other throughput sub-test. Nevertheless, there's a slight difference that needs to be taken into account. When configuring the DMA for an MM2S transfer, the number of bytes we configure for the transfer will be the exact amount of data that the DMA will send. This is not the case for S2MM transfers as we can not ensure that all the data will be received. To check the number of bytes read after an S2MM transfer the LENGTH register in the DMA can be read. After completion of an S2MM transfer, the LENGTH register will be updated by the DMA to hold the total number of bytes read by the DMA. Furthermore, if in this case, the DMA reads more bytes than it was configured to, it will halt, needing CPU intervention for it to be restarted. This is the reason that for S2MM tests we perform a reset to the DMA if the test fails, as shown in figure 32.

3.3 SCENARIO 2: CDMA

In this second environment, theCDMA is used to perform transfers between a BRAM and an SRAM. Like in the first scenario, there is only one clock domain. This clock is generated from the Clocking Wizard block, which has the 125MHz system clock as an input and if timing violations occur the clock source is changed to the PL Fabric.

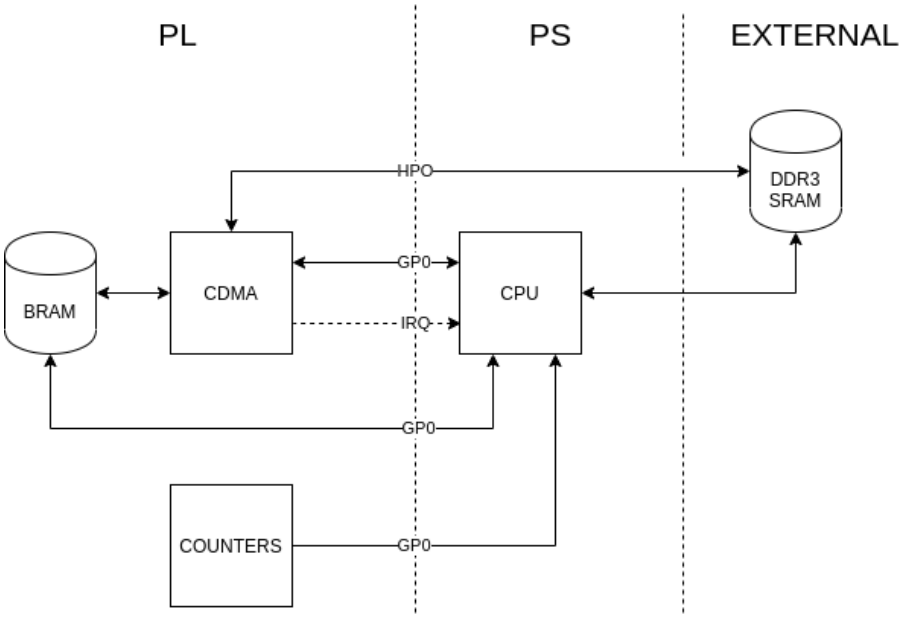


Figure 39: Scenario 2: General view

Figure 39 shows the general view of the scenario. It can be seen that both the CDMA and CPU have access to the BRAM, this has been done to initialise the BRAM from the CPU for different transfers and to be able to verify the transfers directly from the CPU. Furthermore, it can be seen that the CDMA will produce interrupt requests towards the CPU.

Four directions of transfer will be studied in this scenario:

- BRAM-BRAM Transfers
- SRAM-SRAM Transfers
- BRAM-SRAM Transfers
- SRAM-BRAM Transfers

The PS and PL will be described separately.

3.3.1 *PL*

If we look at figure 39 we can predict that BRAM-BRAM transfers will achieve a higher performance as there is no need to cross from the PL to the PS through the HP0 interface. It will be interesting to find out whether SRAM-BRAM transfers and BRAM-SRAM transfers pose any difference.

Appendix c shows the view of the implemented scenario from Vivado. Firstly, figure 100 illustrates the interface view of the scenario. Showing only the AXI interfaces. It can be seen that like in the first environment, AXI GPIOs have been used, these are used to control the same clocks as in the first scenario and to control the onboard LEDs. In addition, it can be seen that the AXI CDMA only has one interface which is an AXI4 Master interface, through which all transfers will be made. The configuration interface

of the CDMA and the AXI GPIOs have been connected to the GP0 interface in the Zynq Processing System block through one only AXI Interconnect.

An instance of a BRAM controller can also be seen. This controller is connected to the AXI CDMA, and both the HP0 and GP0 ports of the Zynq device through another Interconnect. Since we have to separate master interfaces trying to access the instantiated BRAM, a dual-port BRAM was configured and connected to the BRAM controller. Allowing both the PS and CDMA to perform simultaneous accesses to the BRAM. Nevertheless, it must be said that this can cause problems of data integrity when both perform operations in the same region of memory. This will have to be taken into account when designing the software for the PS.

Furthermore, in figure 99 we can see the complete view of the scenario. One only Processor System Reset block has been used, connecting the interconnects to the interconnect_aresetn output, and the rest of devices to the peripheral_aresetn output. The same clock structure has been used to implement Scenario 2, using the Slice blocks to access the clocks. We can also see that in this case no Concat block is needed for the interrupt lines as the CDMA only outputs one interrupt. This line will be activated when a transfer is finished or when an error occurs.

The AXI GPIO map for this scenario was the following:

- GPIO 0
  - Channel 1 [IN]: Counter 1 Value [31..0]
  - Channel 2 [IN]: Counter 2 Value [31..0]
- GPIO 1
  - Channel 1 [OUT]

    [0]: Counter 1 Enable [2]: Counter 2 Enable
    [1]: Counter 1 Reset [3]: Counter 2 Reset

- GPIO 2
  - Channel 1 [OUT]: RGB LEDs [5..0]

Cell	Slave Interface	Slave Segment	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_bram_ctrl_0	S_AXI	Memo	0x5000_0000	128K	0x5001_FFFF
axi_cdma_0					
axi_gpio_0	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
axi_gpio_1					
axi_gpio_2					
axi_cdma_0					
Data (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Memo	0x5000_0000	128K	0x5001_FFFF
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
axi_gpio_3	S_AXI	Reg	0x4124_0000	64K	0x4124_FFFF

Table 3: Scenario 2: Address Map

Table 3 shows the address map for environment 2, which shows how both the PS and CDMA have access to the BRAM Controller and consequently to the BRAM.

The different configurations for the blocks will be now described. Finishing with the resources used to implement the design in the FPGA.

3.3.1.1 *Zynq Processing System*

To configure the Zynq Processing system we had to enable the HP0 Slave interface and the GP0 Master interface as seen in figure 40.

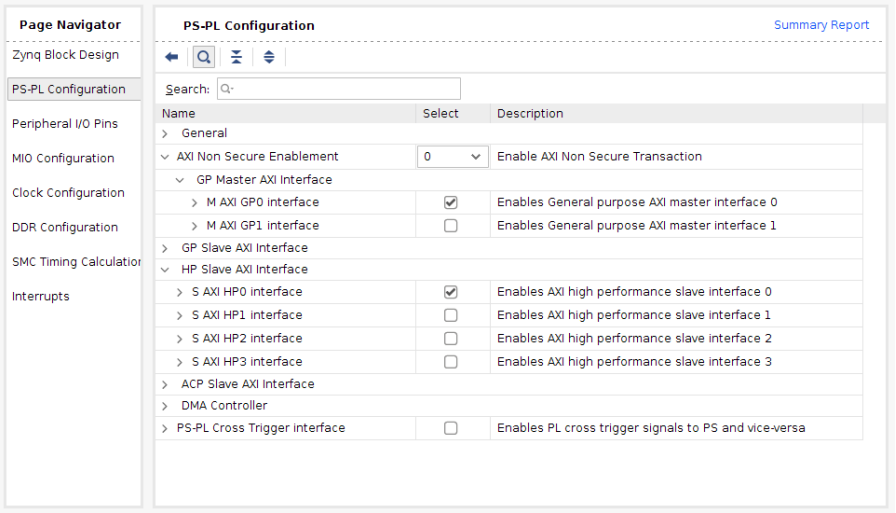


Figure 40: Scenario 2: Zynq Interface Configuration

Next, the PL-PS interrupt ports were enabled, out of which only one was used. This can be seen in figure 41.

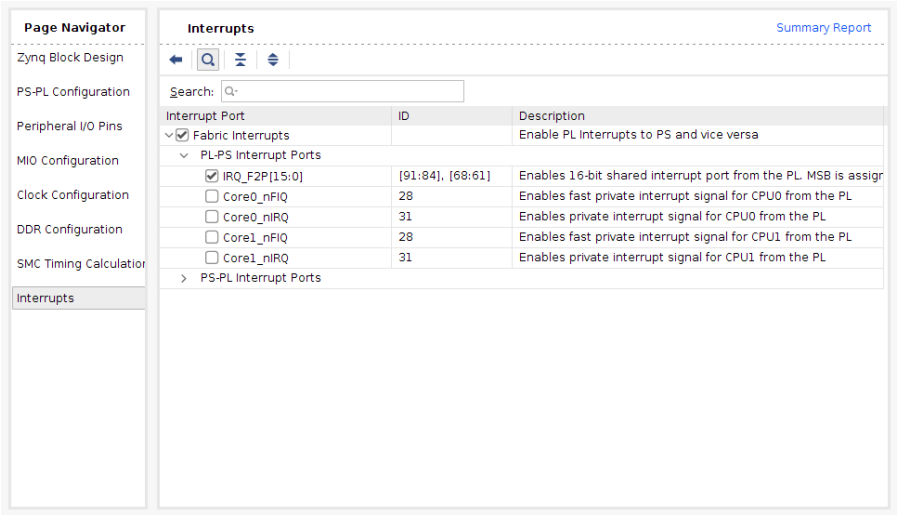


Figure 41: Scenario 2: Zynq Interrupt Configuration

Furthermore, it was verified that UART0 has been enabled.

3.3.1.2 *CDMA*

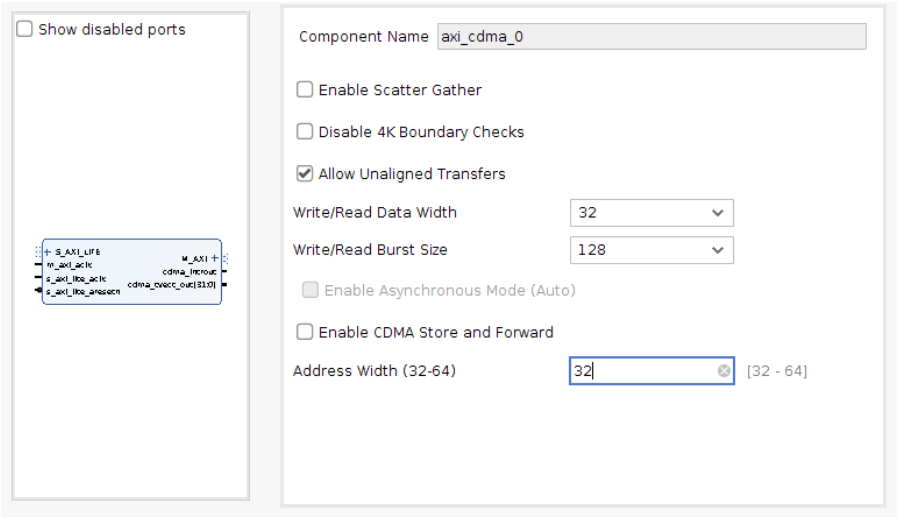


Figure 42: Scenario 2: CDMA Configuration

To configure the CDMA like in figure 42, Unaligned transfers were enabled which meant that the CDMA could write to an address that did not have to be aligned for two-byte words. Furthermore, the data width and address width were set to four bytes and the burst size was set to 128. As different combinations of burst sizes were to be tested, the burst size that was initially configured did not have any importance.

Like in Scenario 1, the PL Fabric clock was used instead of the clocking wizard when high enough frequencies were to be used. Figure 23 shows how to activate it.

3.3.1.3 *Counters*

The counters were configured as in the first scenario. Where Enable and Reset signals were activated, and the Reset had a higher priority than the enable. The counters were also 32 bits wide as the same maximum frequency was going to be reached due to the Cora board's speed grade.

3.3.1.4 *BRAM*

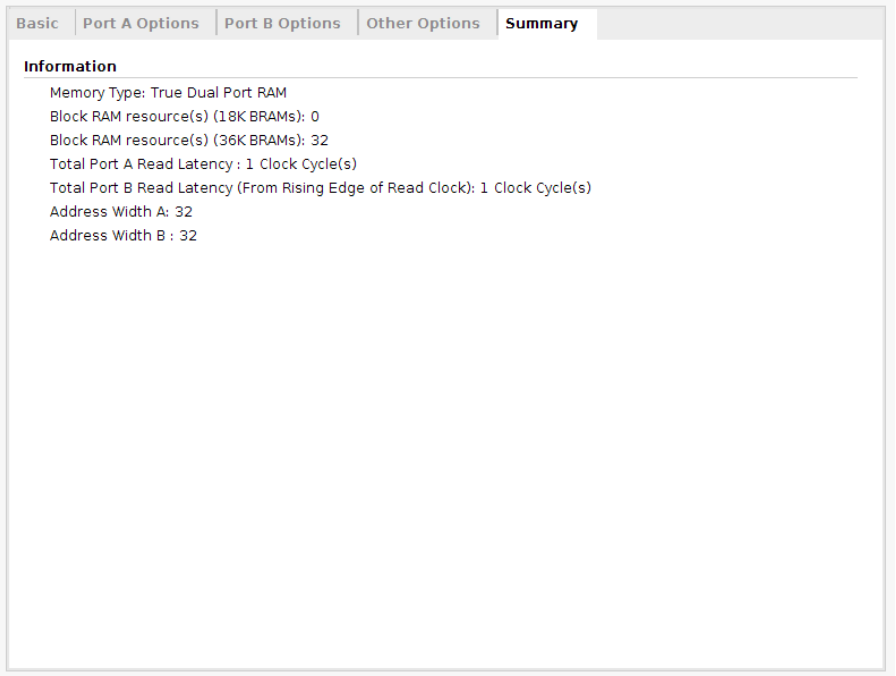


Figure 43: Scenario 2: BRAM Configuration

As previously mentioned, the BRAM was configured to be a Dual Port BRAM. The BRAM was 32K Deep with a width of 4 Bytes.

3.3.1.5 *BRAM Controller*

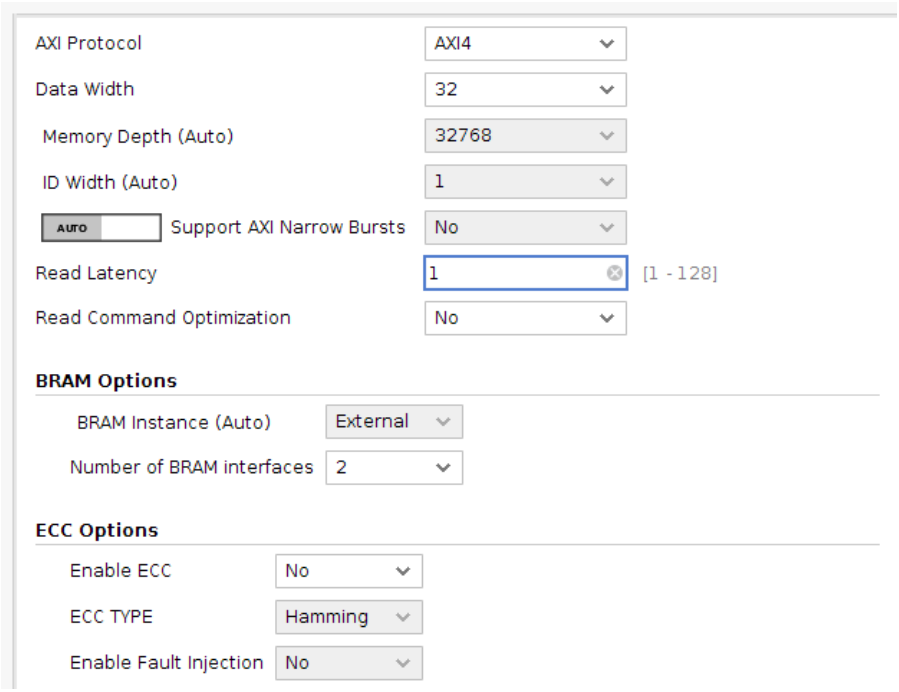


Figure 44: Scenario 2: BRAM Controller Configuration

The BRAM does not natively have an AXI interface, for this reason, a BRAM controller was needed. Figure 44 shows the configuration for this controller. The data width was set to 4 bytes to match that of the BRAM and the AXI Protocol was set to be AXI-4 to match the CDMA and Zynq interfaces. Furthermore, the number of BRAM interfaces was set to two due to having a dual-port BRAM. The other values were left as default.

3.3.1.6 *FPGA Resources*

After implementing the design, we can see how it will be fitted in the FPGA in figure 45. In addition, we can also check how many resources have been used to implement the Scenario.

Resource	**Utilisation**	**Available**	**Utilisation %**
LUT	3886	14400	26.98611
LUTRAM	249	6000	4.1499996
FF	4661	28800	16.184027
BRAM	32	50	64
IO	6	100	6
BUFG	1	32	3.125

Table 4: Scenario 2: Resources used

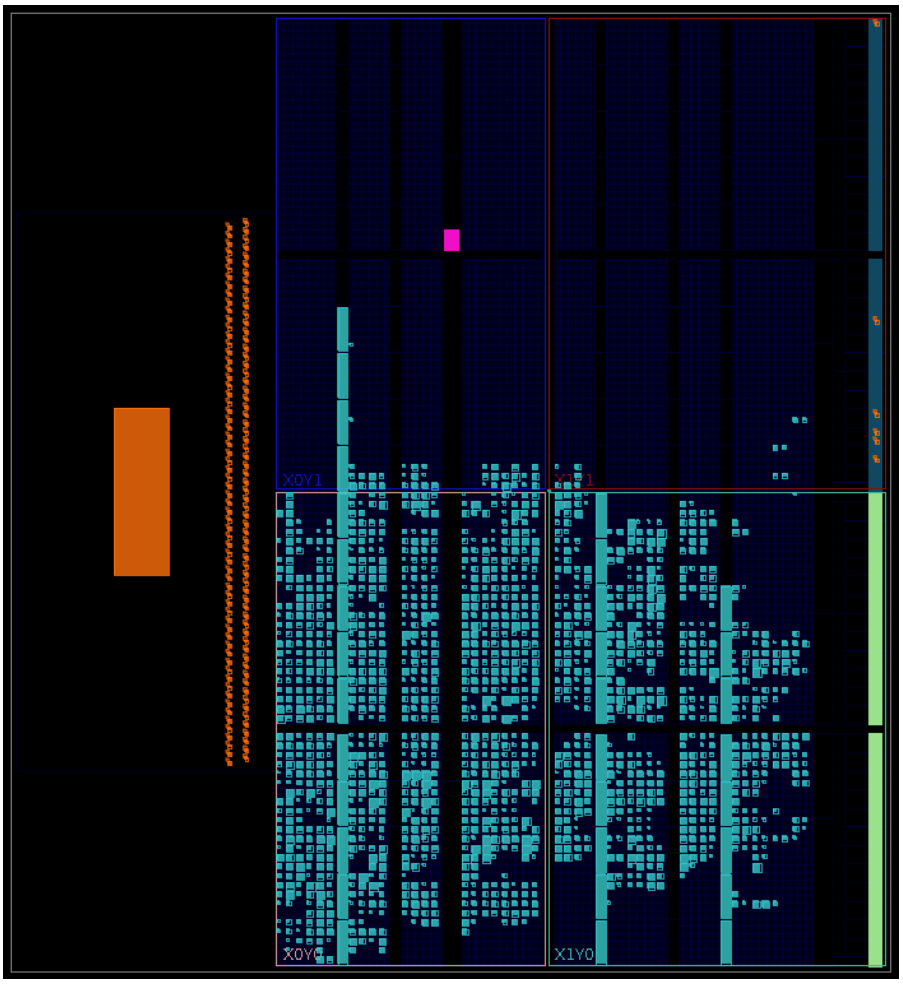


Figure 45: Scenario 2: Implementation view from Vivado

3.3.2 *PS*

As described in section 3.3, this scenario has four different tests. These tests were executed separately, one after the other. Like in the first scenario, the software created will be explained through the use of flowcharts.

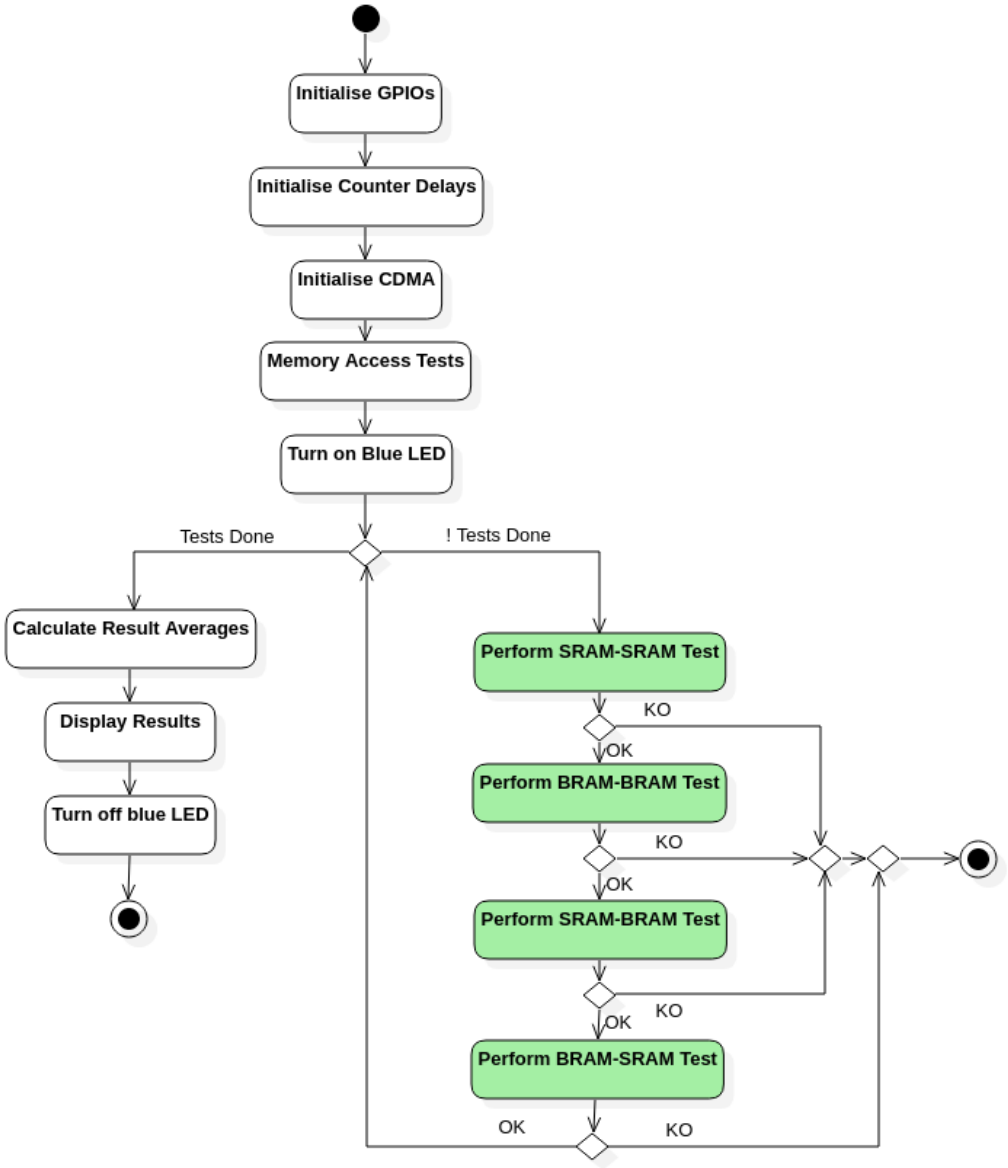


Figure 46: Scenario 2: General Flow Chart

Stepping through the flow chart in figure 46, we start by initialising the different AXI GPIOs. After which we calculate the Counter delays in the same way as in the first scenario. After this, we initialise the CDMA, configuring the interrupt and the ISR that will be executed when an interrupt request from the CDMA occurs.

Following the initialisation of the CDMA, memory tests are performed. Here, the CPU will try to access to the four different memory blocks that have been assigned for the tests. Two in the BRAM and two in the SRAM. Data will be written and read to verify that the hardware is working correctly. After the memory has been tested, the blue LED will be turned on, indicating that the tests are going to be performed.
The tests will be performed in the following order:

1. SRAM-SRAM

2. BRAM-BRAM
3. SRAM-BRAM
4. BRAM-SRAM

After each test has been completed, we will verify that the test was successful. If the test fails, the execution of the code will be terminated. This loop will save the test results of each sub-test until all the tests needed have been performed.
At this point, the result averages will be calculated and the results will be displayed and the blue LED will be turned off, indicating that the tests have ended.

Like in the first scenario, each test is divided into two subtests, but due to all the transfers being MM2MM, there is no need to distinguish between the direction of the transfer. As all the addresses are mapped as shown in table 3, we just need to check the xparameters.h file to choose the correct addresses. The fact that some of these are mapped to the SRAM and others are mapped to the BRAM is transparent to us when performing the tests. So we only need to choose the correct addresses for performing the tests.

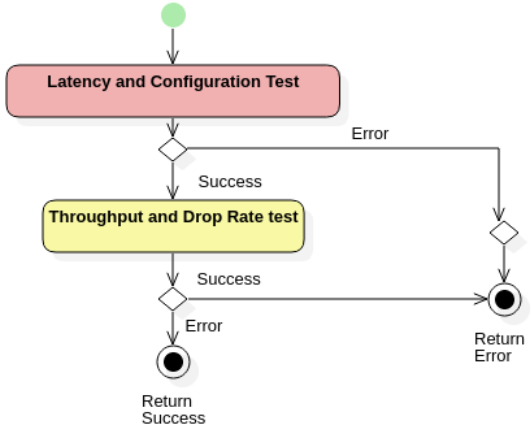


Figure 47: Scenario 2: General Test Flow Chart

The flow chart in figure 47 shows what has already been described. Clearly showing how each test is divided into two sub-tests, where first the latency and configuration times will be measured and in the next test the total time, configuration time and drop rate will be measured.

3.3.2.1 *Latency and Configuration Test*

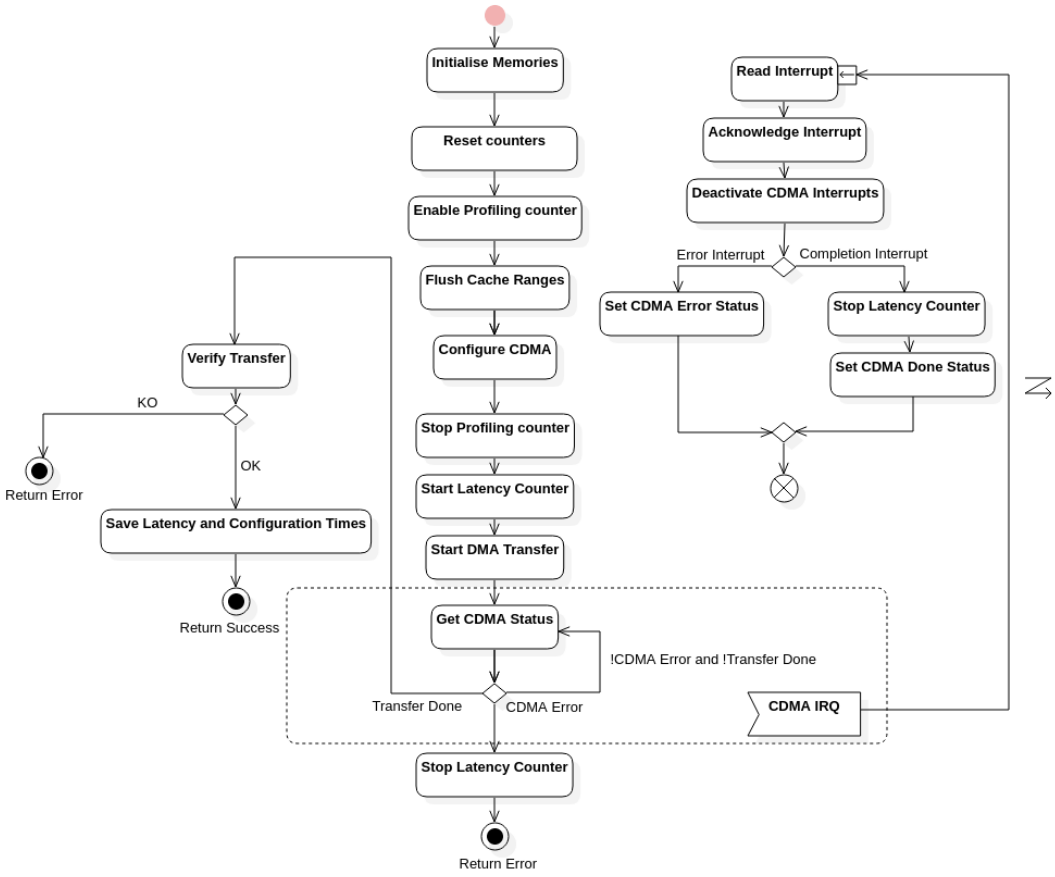


Figure 48: Scenario 2: Latency and Configuration Test Flow Chart

The flow chart in figure 48 shows the execution flow for the tests. Firstly, the memories will be initialised. This will clear the destination memory block and write a word of data to the origin memory block. Next, the counters will be restarted and the Profiling counter will be enabled, this counter will be used to measure the configuration time. After enabling the counter, the cache will be flushed and the CDMA will be configured to perform a transfer. After which point we will stop the profiling counter and start the latency counter straight away. Then the CDMA transfer will be started.

Here we enter a region where an interrupt request from the CDMA will send the code execution to the ISR. The code will loop, reading the CDMA status until either an error has occurred or the transfer has ended. If an error occurs, the latency counter will be stopped and an error will be returned. On the other hand, if the transfer occurs successfully the transfer will be verified, checking in the destination memory block that the word of data has been transferred. If the transfer is successful, the latency and configuration times will be saved and the sub-test will return success. If when the transfer is verified the number of bytes is not equal to the ones sent, the test will fail, this is because with only one word is being sent the data must be sent properly.

When the code is looping, waiting for the transfer to complete, an interrupt will direct the execution to the ISR. Here the interrupt will be read and acknowledged. After verifying that the interrupt confirms a correct completion of the transfer the latency counter will be stopped, and a Transfer Done flag will be activated. If the interrupt indicates an error, an error flag will be activated.

3.3.2.2 *Throughput and Drop Rate Test*

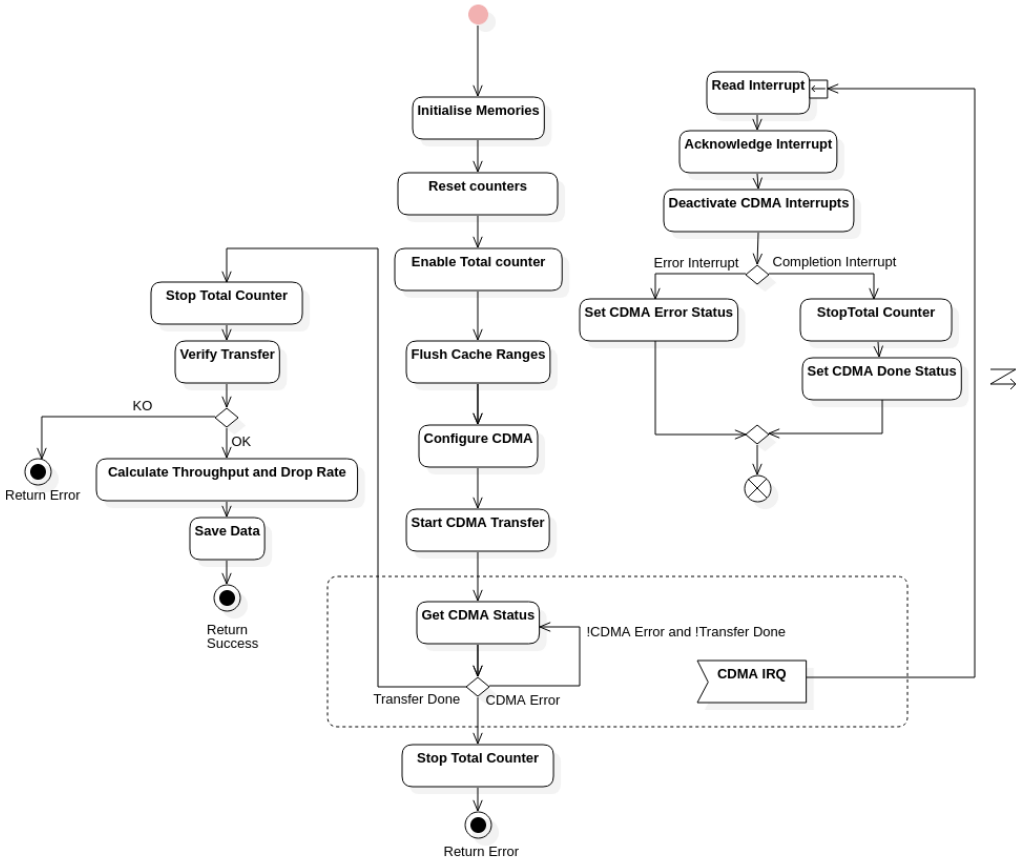


Figure 49: Scenario 2: Throughput and Drop Rate Test Flow Chart

Figure 49 shows the flowchart for the throughput and drop rate test, which starts by initialising the origin memory block with an ascending sequence of numbers and clearing the destination memory block. Then the counters will be reset and the Total time counter will be enabled. This counter will be stopped in the ISR once the transfer has ended successfully. The main difference between the throughput test and the latency test is the verification of the transfer. Due to initialising the origin memory with a sequence of numbers, we can check the destination memory block by reading it directly through the CPU, this will allow us to check the number of bytes that have been dropped. The throughput is calculated using equation 5 and the drop rate is calculated using equation 6.

The third and last scenario implemented made up the RX Channel of GRITS's NVIS project. For the NVIS project, there was a problem with the RX Channel where some bytes were being lost. This only happened for the RX channel as for this channel, the data fills up the FIFO, and if the DMA can not cope up with the RX data rate, bytes will be lost due to the FIFO overflowing. On the other hand, there is more slack for the TX channel, as the data has to first be stored in the SRAM before it is moved by the DMA to another FIFO for it to be sent[6]. This means that data will never be lost for the TX channel.

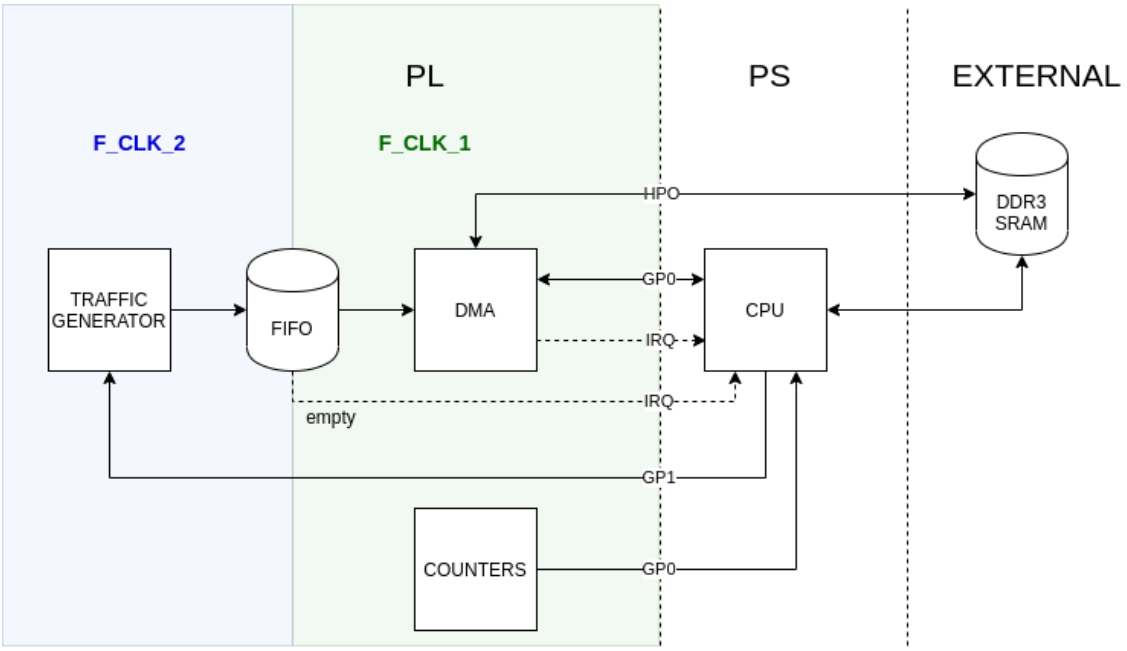


Figure 50: Scenario 3: General view of the scenario

The general view of the scenario can be seen in Figure 50. The different colours represent the clock domains in the scenario, where a FIFO is used to separate these clock domains. The traffic generator works at a lower frequency than the rest of the system, and it generates data that will be stored in the FIFO. This FIFO works as an intermediate buffer. The DMA reads the data from the FIFO and saves it in the SRAM. Depending on the efficiency of the implemented design, more or fewer bytes will be lost at different frequencies. Furthermore, the size of the FIFO will also be critical to reducing the number of lost bytes. The goal is to avoid a loss of data at the 8MBps that were calculated in equation 2 and to also find what maximum data rate the design can manage without losing an excessive amount of bytes. Furthermore, it will be interesting to study the effects that the different system parameters can have on the system.
From figure 50, it can also be seen that counters were used to measure the same metrics as in the other scenarios.

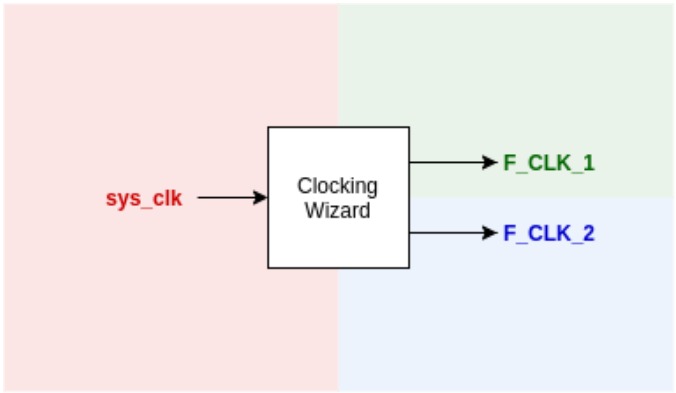


Figure 51: Scenario 3: Clock Domains

Figure 51 shows how the clocking wizard was used to produce the two different clock frequencies, allowing the Traffic Generator and the rest of the system to work at different frequencies. Furthermore, we can see from figure 50 that the traffic generator is configured through the interface GP1.

It can also be seen that two different types of interrupts are needed, one coming from the DMA and the other coming from the empty signal of the FIFO.

Two different working modes for the traffic generator will be used to analyse the system:

1. Data Peaks mode: In this mode, the delay parameter of the traffic generator will be varied between tests to produce different bursts of data.
2. Constant Data Flow: The traffic generator delay will be set to zero, and the length parameter of the traffic generator will be varied to observe how the activation of the TLAST signal affects the performance of the system.

3.4.1 *PL*

Referring to Appendix d, we can see the interface view and complete view of the design in Vivado. For readability, the complete design has been separated into two figures.

Figure 102 shows the interface view, where only the AXI interfaces can be seen, a similar structure has been followed to the one in the first scenario, the main difference is that in this case only a write channel is needed for the DMA as the FIFO is filled by the traffic generator. Furthermore, we can see how the Traffic Generator will be configured through the Master GP1 interface in the Zynq.

The complete view of the scenario can be seen in figure 103, 105 and 105. By looking at the complete scenario, we can see that the same setup for the counters has been used as in the last two scenarios, using the slices and AXI GPIOs to control them. The GPIOs

have also been used to reset the FIFO and the traffic generator through two separate Processor system reset blocks.

Due to the FIFO having two separate clocks, the data count displayed is also divided into two signals, these are fed to GPIOs for calculating the amount of data in the FIFO in the PS. Furthermore, we can see that to produce an interrupt from the FIFO; the prog_empty signal was fed to an AXI GPIO that produces an interrupt. This interrupt is concatenated with the DMA S2MM interrupt using a Concat block as the first scenario.

If we look at the FIFO, we can see that the overflow, write busy, and read busy signals were also connected to an AXI GPIO. These were used for debugging purposes.

The GPIO map for the scenario is the following:

- GPIO 0
  - Channel 1 [IN]: FIFO read count [13..0]
  - Channel 2 [IN]: FIFO write count [13..0]
- GPIO 1
  - Channel 1 [IN]: Counter 1 Value [31..0]
  - Channel 2 [IN]: Counter 2 Value [31..0]
- GPIO 2
  - Channel 1 [OUT]

[0]: Counter 1 Enable [2]: Counter 2 Enable [4]: FIFO NReset
[1]: Counter 1 Reset [3]: Counter 2 Reset [5]: TrafGen NReset

  - Channel 2 [IN]

[0]: FIFO Read Busy [2]: FIFO Prog Empty
[1]: FIFO Write Busy [3]: FIFO Overflow

- GPIO 3
  - Channel 1 [OUT]: RGB LEDs [5..0]
- GPIO4
  - Channel 1 [IN + INTERRUPT]: [0]: FIFO prog empty

We can see that AXI GPIO4 was used to generate the interrupt towards the PS. In addition, the resulting address map of the scenario can be seen in the following table:

Cell	Slave Interface	Slave Segment	Offset Address	**Range**	**High Address**
axi_dma_0					
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
processing_system7_0					
Data (32 address bits : 0x40000000 [1G] ,0x80000000 [1G])					
axi_dma_0	S_AXI_LITE	Reg	0x4040_0000	32K	0x4040_7FFF
axi_gpio_0	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
axi_gpio_1	S_AXI	Reg	0x4122_0000	64K	0x4122_FFFF
axi_gpio_2	S_AXI	Reg	0x4123_0000	64K	0x4123_FFFF
axi_gpio_3	S_AXI	Reg	0x4124_0000	64K	0x4124_FFFF
axi_gpio_4	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF
axi_traffic_gen_0	S_AXI	Reg0	0x8000_0000	64K	0x8000_FFFF

Table 5: Scenario 3: Address Map

Table 5 shows how using the Master GP1 interface for accessing the traffic generator will be completely transparent to the PS. GP0 is mapped to address 0x40000000 and has a range of 1G; on the other hand, GP1 is mapped to address 0x80000000 and has a range of 1G. The only difference that will be seen from the PS is this difference in the address range.

The configuration of the different used IPcores will be now explained.

3.4.1.1 *ZYNQ Processing System*

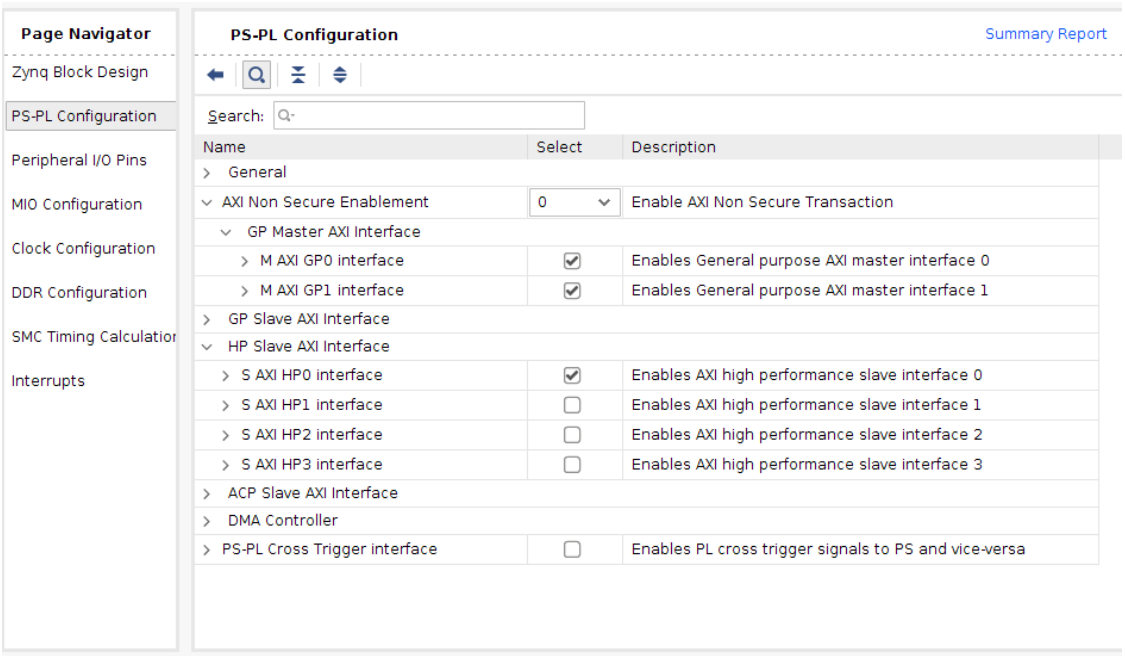


Figure 52: Scenario 3: Interface configuration for the ZYNQ Processing system

To configure the interfaces, GP0, GP1 and HP0 were enabled as shown in figure 52

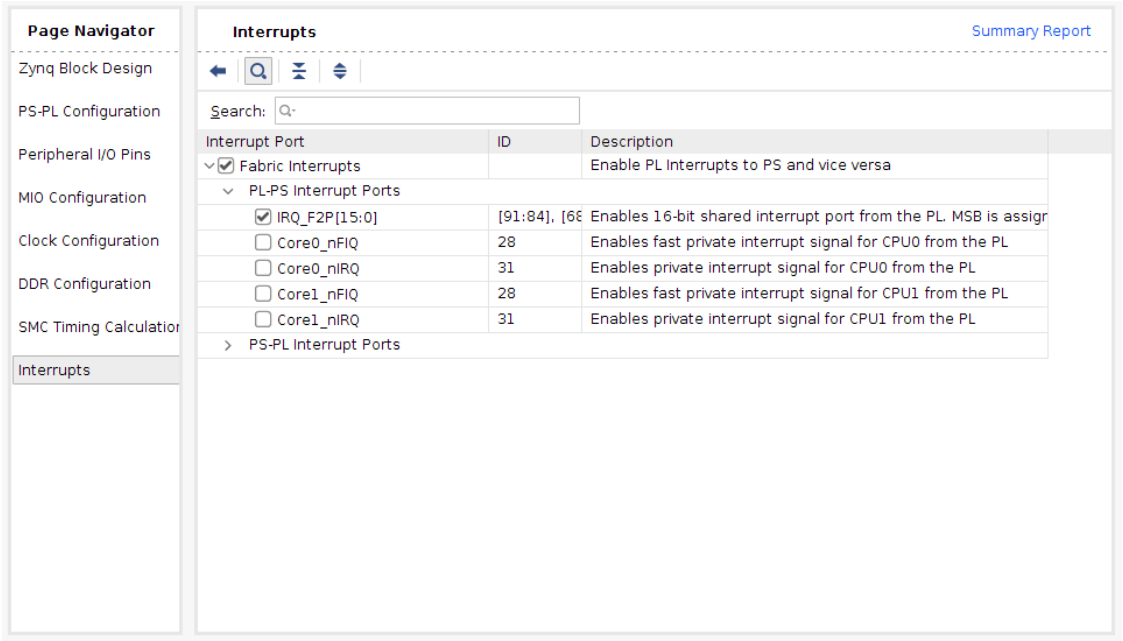


Figure 53: Scenario 3: Interrupt configuration for the ZYNQ Processing system

After this, the interrupt ports were configured as seen in figure 53.

3.4.1.2 *DMA*

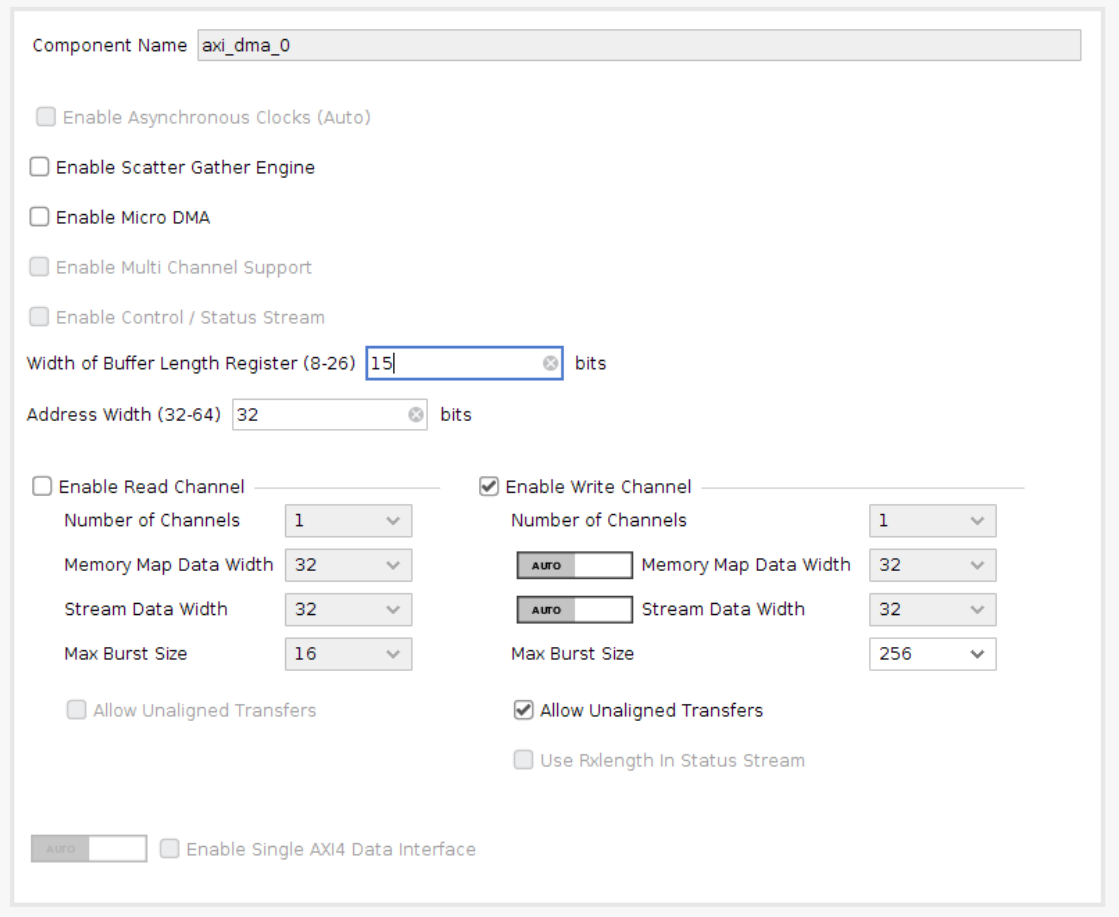


Figure 54: Scenario 3: DMA Configuration

As seen in figure 54, only the write channel was enabled. Unaligned transfers were needed for this environment to work correctly. This is because if the FIFO is filled with 7 bytes, for example, the DMA has to be able to transfer these bytes properly. Moreover, the address width was configured to be 32 bits, and the burst size was set to 256. The buffer length register was set to a width of 15 as $2^{15} - 1 \approx 32\text{K}$, this value was changed depending on the FIFO size, but for a FIFO Depth of 8K, $8\text{K} \cdot 4\text{Bytes/word} = 32\text{KB}$ a buffer length width of 15 bits were used.

3.4.1.3 *FIFO*

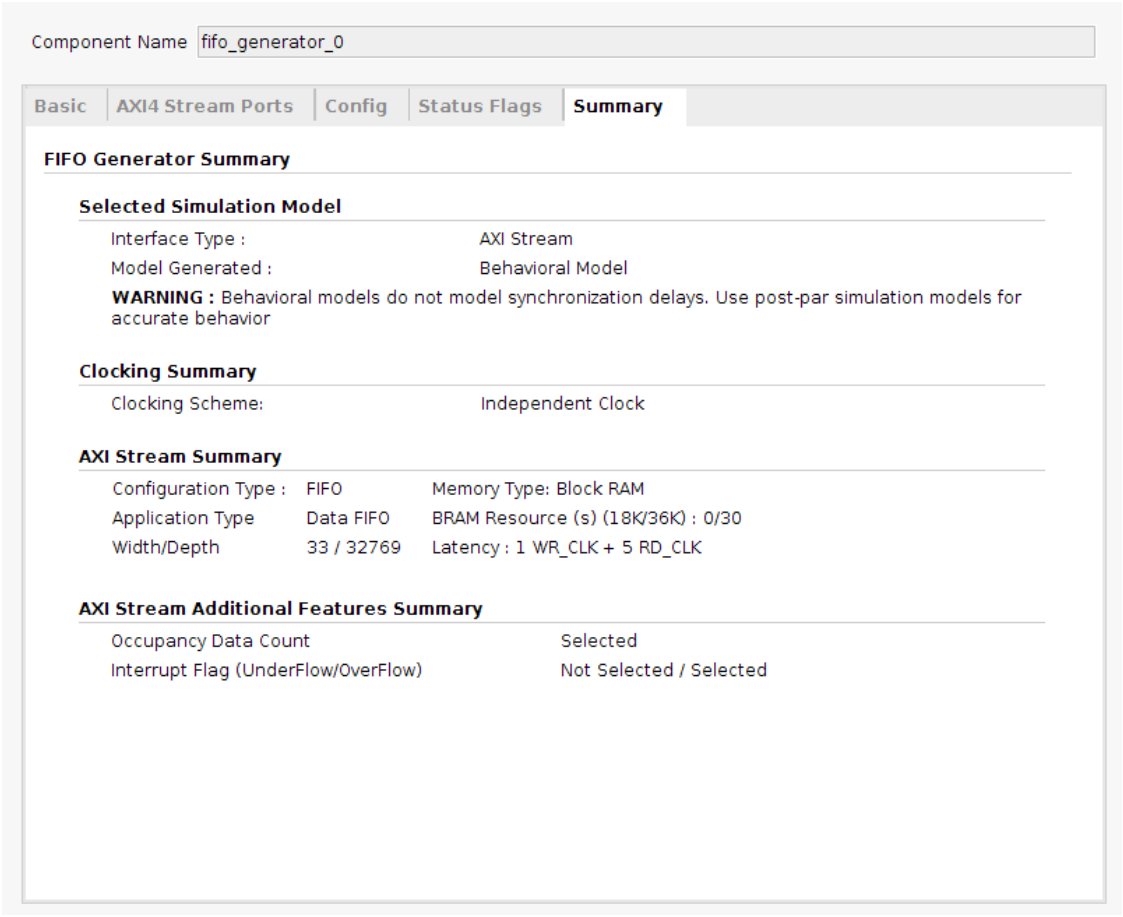


Figure 55: Scenario 3: FIFO Configuration

The FIFO was configured as shown in figure 55. Where an AXI-Stream interface was selected as independent clocks were chosen. Figure 55 was taken when using a FIFO depth of 32K and a width of 4 bytes. Additionally, the data occupancy count and the overflow flag was enabled.

Another flag that was enabled was the programmable empty flag which was set to a value of 4. The programmable empty flag was used instead of the underflow flag because the underflow flag would only activate when there was a read request, and the FIFO was empty, not when the FIFO was empty, but no read requests were made.

3.4.1.4 *Traffic Generator*

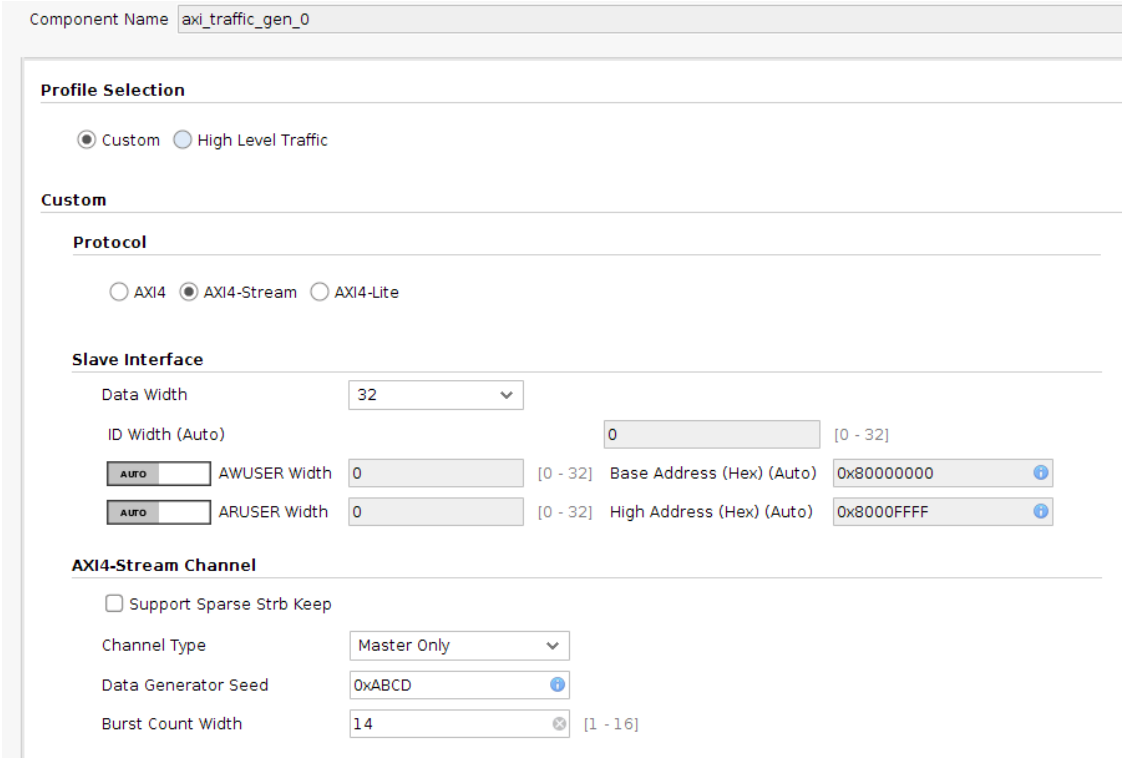


Figure 56: Scenario 3: Traffic Generator Configuration (1)

The traffic generator was configured to produce data in the AXI-Stream protocol, as seen in figure 56, a 4-byte data width was set. We disabled the *sparse strb keep* function as if TKEEP and TSTRB[23] were used, the effective data rate that the traffic generator produced would not be constant. Furthermore, a burst count width of 14 was set to prevent the FIFO from overflowing if the Traffic Generator was configured to produce random data lengths. Nevertheless, this random mode was not implemented, and so the configured width was more than enough.

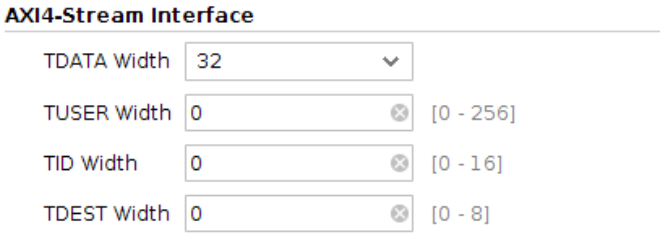


Figure 57: Scenario 3: Traffic Generator Configuration (2)

TDATA was set to 4 bytes like in the other scenarios and no TUSER, TID and TDEST were needed as only one AXI-Stream master had been connected to the DMA.
If we look at figure 105 it can be seen that the TREADY signal was forced to '1' using a Const block. This was done to accurately emulate the flow of data that would be experienced in the NVIS project. If we connect traffic generator's TREADY to the FIFO's

TREADY, what happens is that when the FIFO is full, it asserts the TREADY signal to low, and the traffic generator does not produce any data. This means that no bytes would be ever lost due to the FIFO overflowing. In the real world, what would happen when the FIFO is full is that the data would keep on coming and bytes would be lost, therefore by forcing TREADY to '1' the traffic generator will output data even when the FIFO is full, correctly emulating the flow of data in the NVIS project.

3.4.1.5 *Interrupt AXI GPIO*

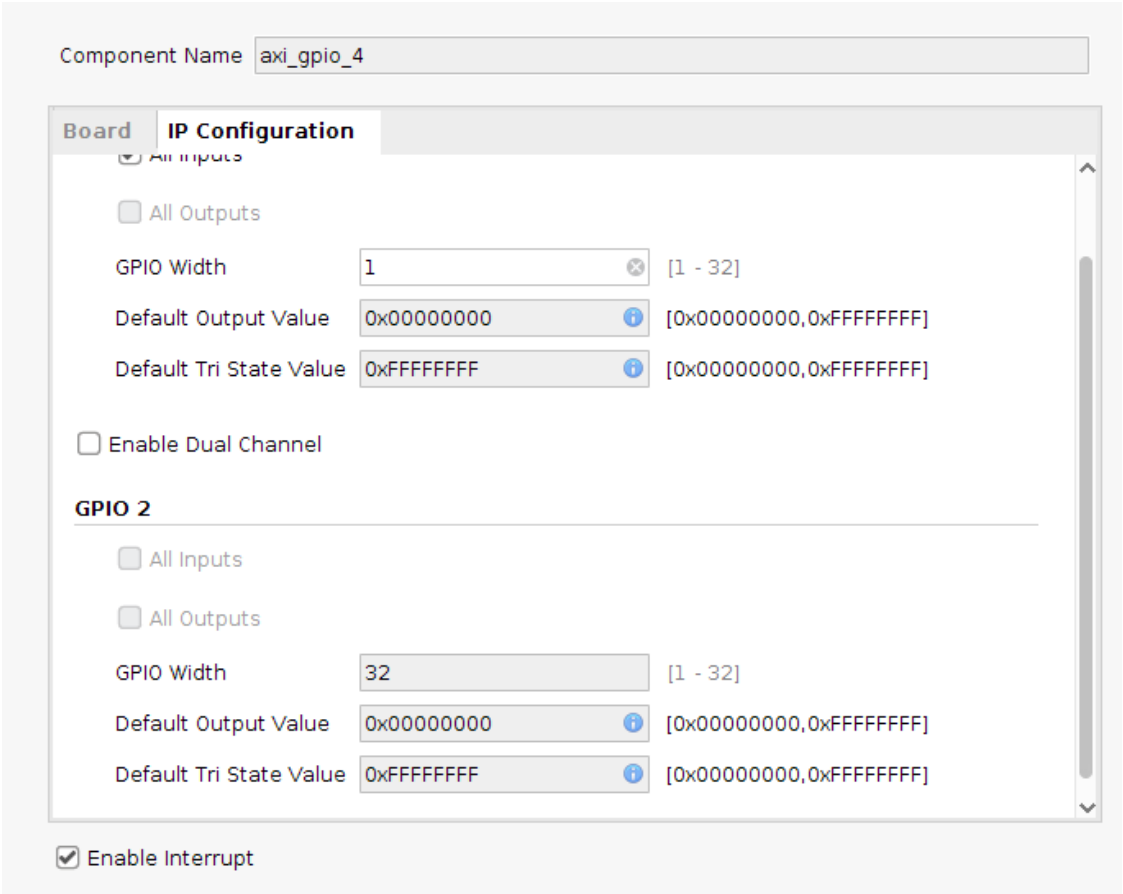


Figure 58: Scenario 3: AXI GPIO Interrupt configuration

To generate an interrupt when the prog_empty signal changed of state, the interrupts were enabled for GPIO4 as shown in figure 58.

3.4.1.6 *FPGA Resources*

The implementation results can be seen in figure 59 where it can be appreciated that this scenario takes up more space in the FPGA than the others.

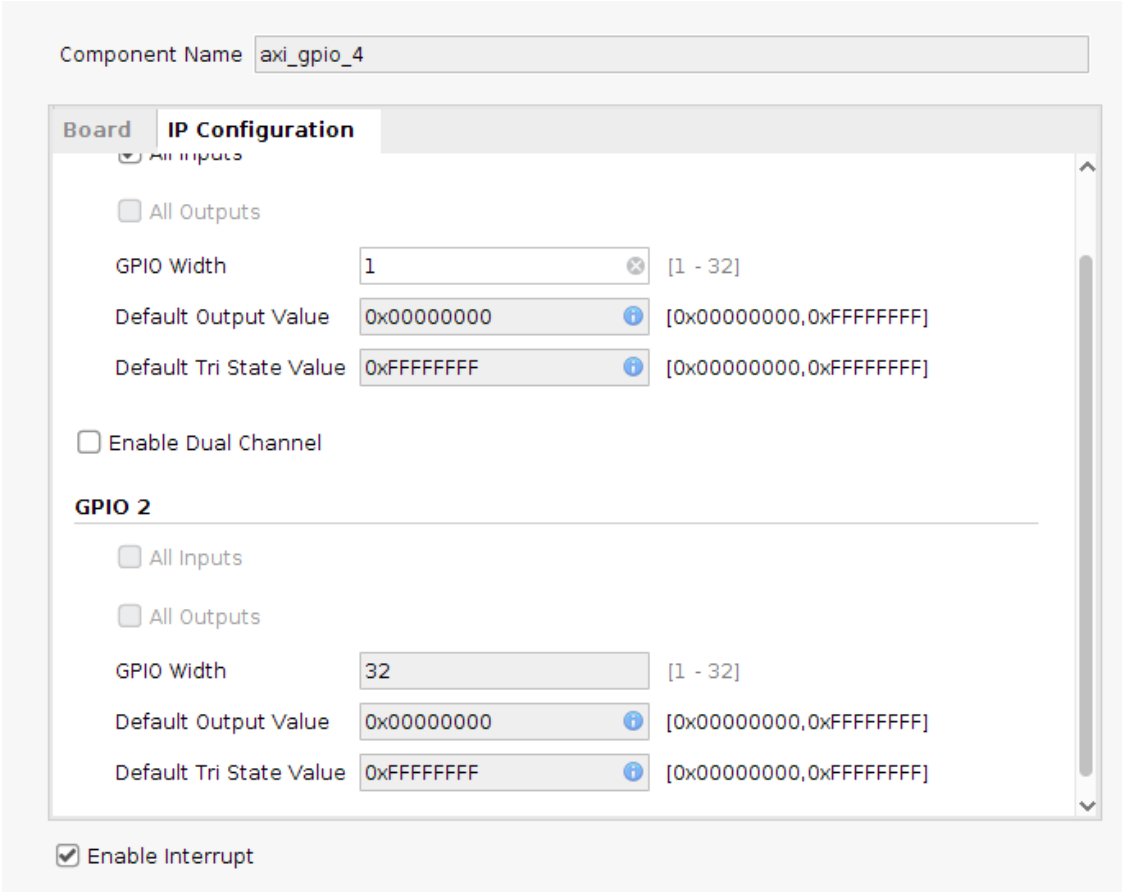


Figure 59: Scenario 3: FPGA Implementation

Resource	Utilisation	Available	Utilisation %
LUT	3695	14400	25.659721
LUTRAM	374	6000	6.2333336
FF	5052	28800	17.541666
BRAM	10	50	20
IO	7	100	7
BUFG	3	32	9.375
MMCM	1	2	50

Table 6: Scenario 3: FPGA Resources

Furthermore, table 6 shows the resources utilised by the design. It can be seen that the clocking wizard used two MCMMs to generate the clock outputs instead of PLLs. It can also be seen that there is still more than enough space to expand this design to implement the TX channel of the NVIS project too.

3.4.2 *PS*

As with the other scenarios, the software created will be explained through different flowcharts. Which will also make it easier to explain how the interrupts were used in this scenario. This scenario was divided into two tests:

- Latency test: The traffic generator is configured to transfer one word of data. The time is taken for the word to be transferred from the traffic generator to the SRAM is measured.
- Mode test: The traffic generator is configured to first produce data in the Data Peaks mode and then in the Constant Data Flow mode. The delay and length parameters for the traffic generator are modified, and the following metrics are measured:
  - Total time taken
  - Bytes dropped and drop rate
  - Configuration Time
  - number of DMA configurations
  - FIFO Overflows
  - number of bytes transferred

Each test is performed ten times to then perform averages of the results. To execute the different tests preprocessor directives were used. This allowed to quickly change from performing the latency test to the mode test. As these two tests are independent of each other, they will be explained separately.

3.4.2.1 *Latency Test*

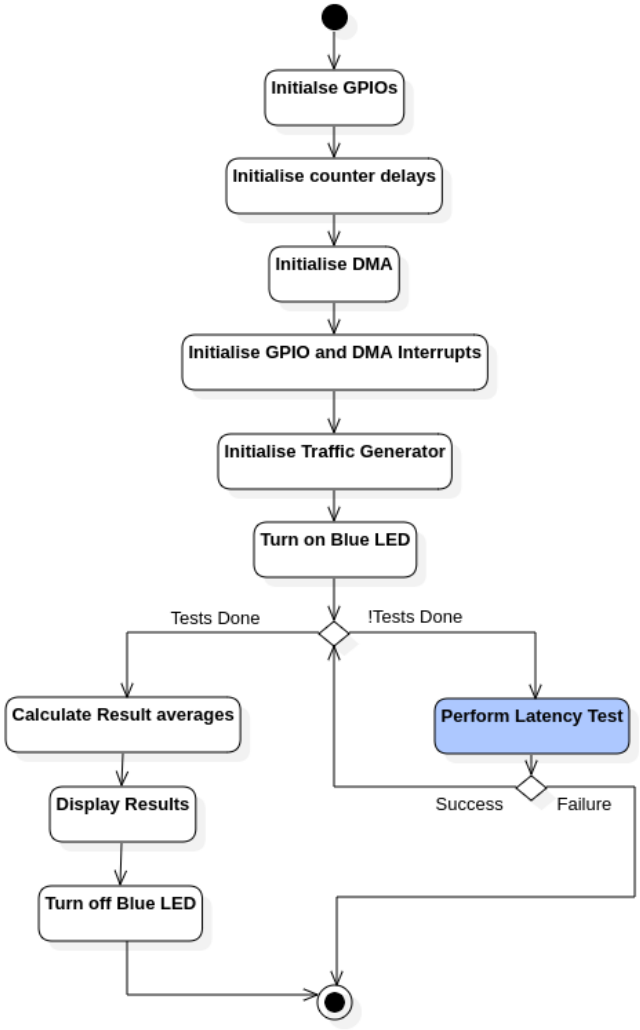


Figure 60: Scenario 3: General Flow Chart for the Latency test

The flow chart in figure 60 shows the general flow for performing the latency tests. Firstly, the AXI GPIOs are initialised following the counter delays. After this, the DMA is initialised, and the GPIO and DMA interrupts are configured and assigned to their different ISRs. Moreover, the traffic generator is initialised and the blue LED is turned on, indicating that the tests will begin.

The Latency tests will be performed until ten results have been obtained if any of these tests fail the code execution will stop. Differently, if all the tests succeed, the resulting average will be calculated, and both the results and the average will be displayed. Then the blue LED turns off, indicating that the test has finished correctly.

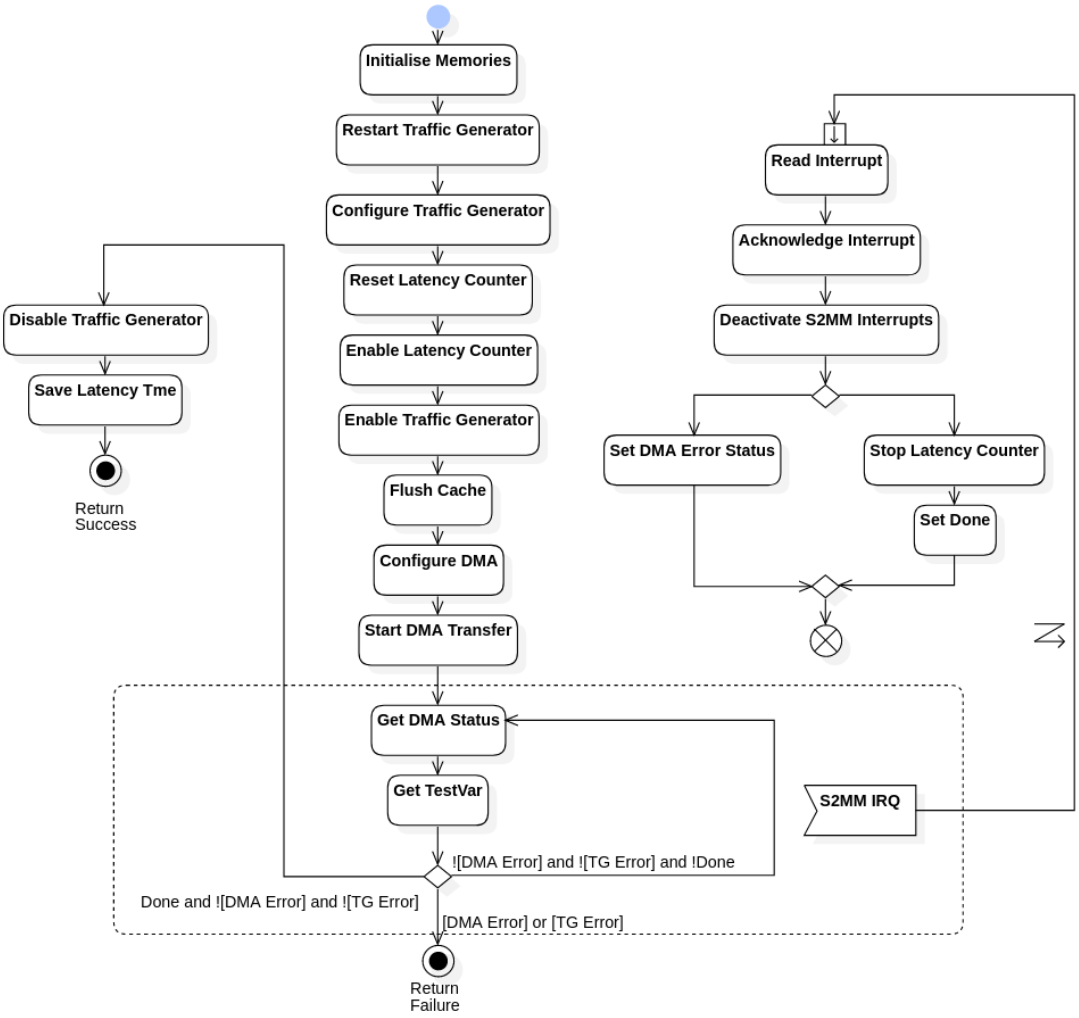


Figure 61: Scenario 3: Flow Chart for the Latency test

The Latency test can be seen in figure 61. Firstly, the memories are initialised by resetting the FIFO and clearing the SRAM. Next, the traffic generator is restarted to return it to its default settings after which it is configured to transfer one 4 byte word of data. The latency counter is then restarted and enabled. At this point, the latency counter is enabled, and the traffic generator is started. Then the cache is flushed, and the DMA is configured to transfer the data from the FIFO to the SRAM, here the S2MM interrupt is also enabled.

We then enter a loop when the DMA Status and a structure called TestVar is checked. The structure of TestVar is the following:

```
typedef struct{
    u32  t_total;
    u32  t_profiling;
    u32  n_configs_dma;
    u32  total_length_dma;
    u32  fifo_rd;
    u32  fifo_wr;
    u32  fifo_ovf_cnt;
```

```
    char test_running;
}TestVar;
```

If there is an error detected in the DMA or the traffic generator, the test will end returning Failure. Nevertheless, if the test ends successfully, the loop will break, and the traffic generator will be disabled. Next, the latency time will be saved, and the test will return success.

While in the loop, if an S2MM interrupt occurs, the code will jump to the ISR, where the interrupt will be read and acknowledged. Furthermore, S2MM interrupts will be disabled. If the interrupt is found to indicate an error, an error flag will be activated and when the ISR finished the loop will break returning Failure. Always, if the interrupt indicates a correct completion of the transfer, the latency counter is stopped and a done flag is enabled to indicate the loop that the transfer has been performed successfully. The attribute total_length_dma from the TestVar variable will contain the number of bytes read.

3.4.2.2 *Mode Test*

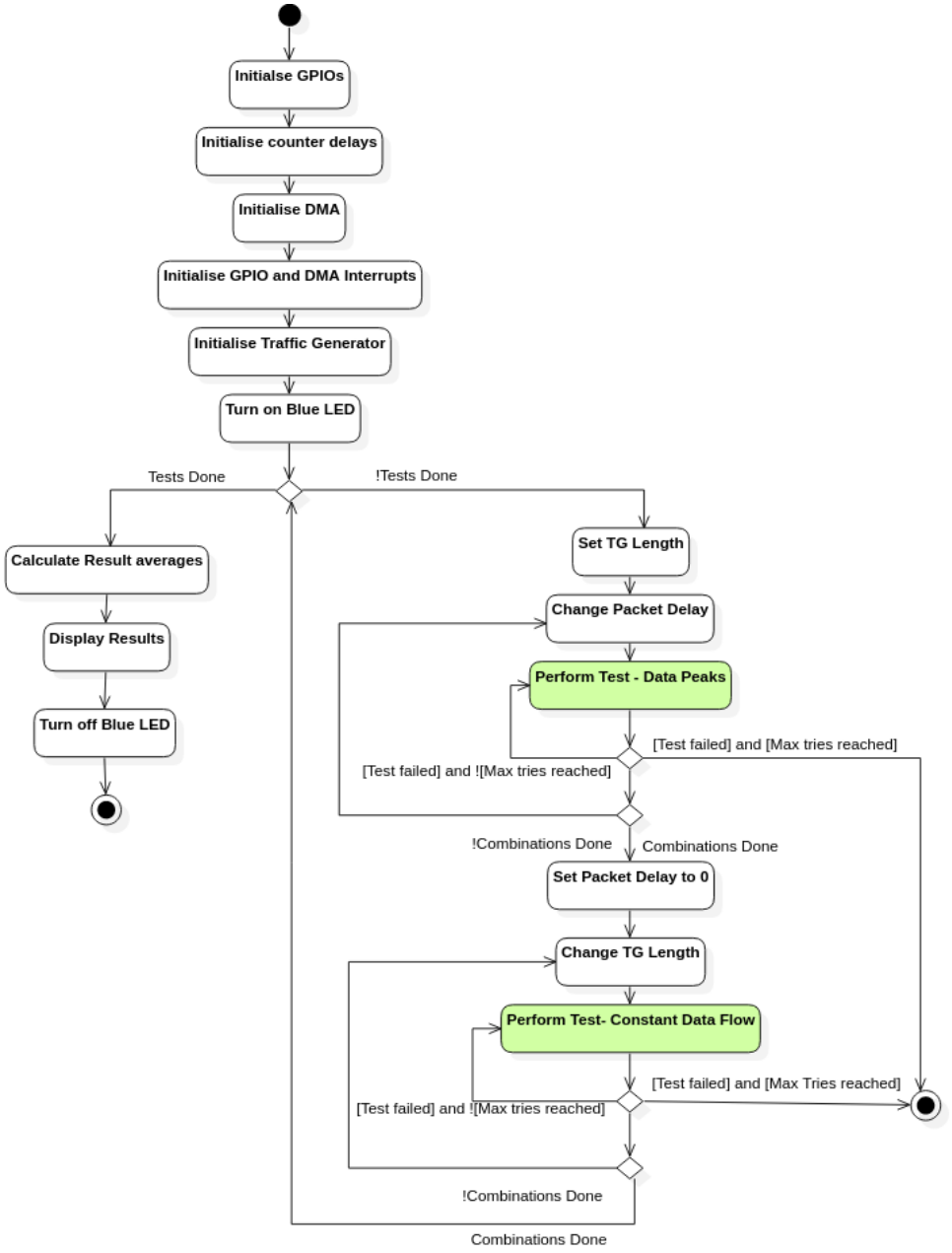


Figure 62: Scenario 3: General Flow Chart

The general flow of the mode test can be seen in figure 62. The initial structure of the code is the same as in the Latency test. The differences appear after the Blue LED is turned on, indicating the tests will begin. Firstly, the traffic generator length parameter is set; all the Data Peaks tests will use the same traffic generator length. Then the packet delay parameter is changed; different packet delays will be tested in the Data Peaks test. Each combination is tested ten times. After the test is performed if the test fails it will be executed again until a maximum number of tries has been reached, if all the combinations have not been tested, the packet delay will be changed, and the test will be performed again. Once all the combinations have been tested the code will

proceed to perform the Constant Data Flow test.

The general structure for performing this test is the same as in the Data Peaks test, but now the packet delay will always be set to 0, and the Length parameter will be varied with each combination.

Once all combinations have been tested, and all tests have been completed, the result averages will be calculated and displayed, finishing by turning the blue LED off.

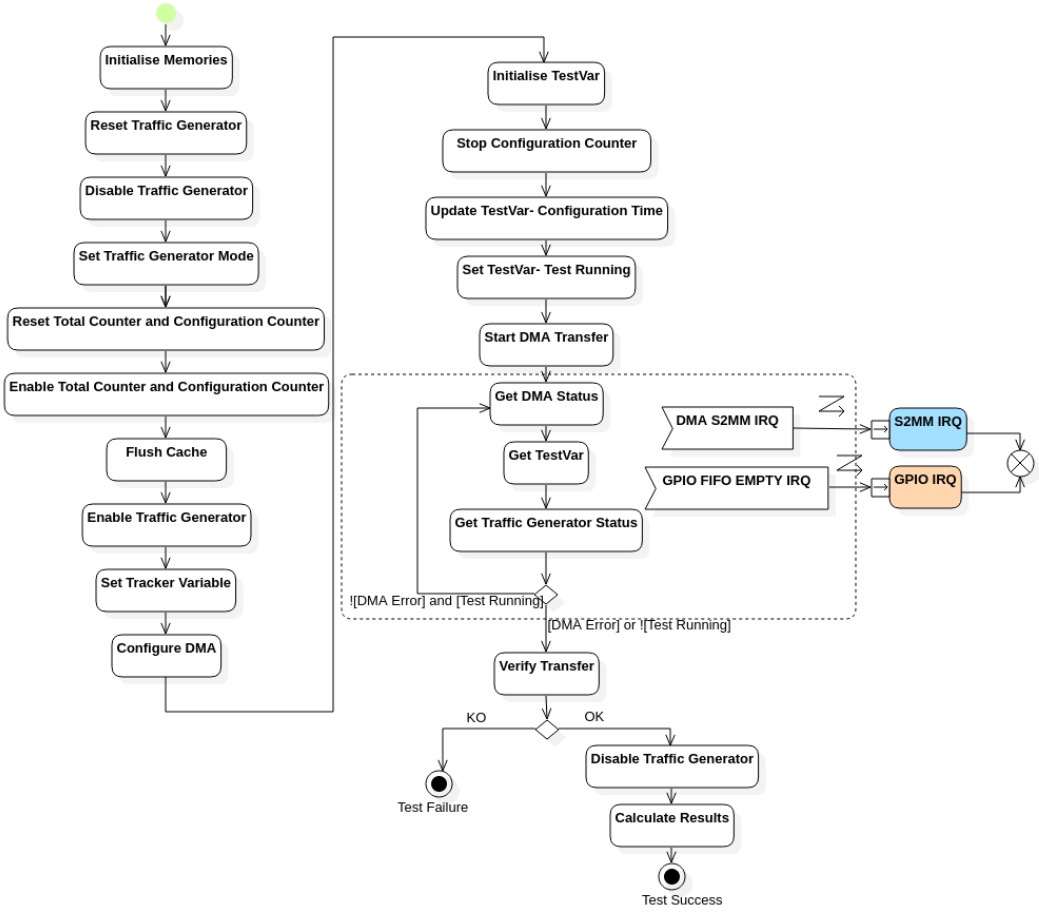


Figure 63: Scenario 3: Mode Test Flow Chart

The structure of the Mode test can be seen in figure 63 and will be now explained. After initialising the memories by resetting the FIFO and clearing the used region of the SRAM, the traffic generator is reset and disabled. Next, the Traffic Generator is configured by setting the chosen length and delay parameters. After this, the total time counter or Total counter will be reset and enabled. This will also be done to the configuration counter.

At this point, the cache will be flushed, and the traffic generator will be enabled. After this, the tracker variable is initialised; this variable has the following structure:

```
typedef struct{
    u32 address; // address to be written to
    u32 nBytes; //Total Number of bytes in the test
    u32 initialDmaLength; //Used to ensure bytes from the last test are not
        counted
    char emptyWait; //Indicates that a GPIO interrupt is expected
}DmaTracker;
```

The first parameter in this structure is used to save the address that was written to by the DMA. As many transfers will occur, the destination address has to be updated to not overwrite the bytes that were written from a previous transfer. The other parameters will be explained as the flowchart is described.

After setting the Tracker variable, the DMA will be configured to receive the maximum number of bytes it can read, and the S2MM interrupts will be enabled. TestVar will be now initialised. At this point, the configuration counter will be stopped, and TestVar will be updated by increasing the configuration time and count and by setting the test running flag to high.

Following this, the DMA will be started, and we will loop acquiring the DMA Status, the Traffic Generator Status and refreshing the TestVar. When a DMA Error is detected, or the test running flag from TestVar goes low, the loop will be broken. While in the loop, two possible interrupts may occur, S2MM and GPIO. The GPIO is enabled in the S2MM ISR when needed.

After the loop breaks, the transfer is verified, checking the status of the DMA and Traffic generator and verifying from the total_length_dma that the number of bytes reads is coherent (not more significant than the total bytes set for the test and not equal to '0' and checking that the FIFO is empty through the fifo_rd and fifo_wr parameters which are updated using the FIFO's read and write count signals. If the test has been performed successfully, the traffic generator will be disabled, and the different results will be calculated using the TestVar structure.

The two ISRs will be now explained:

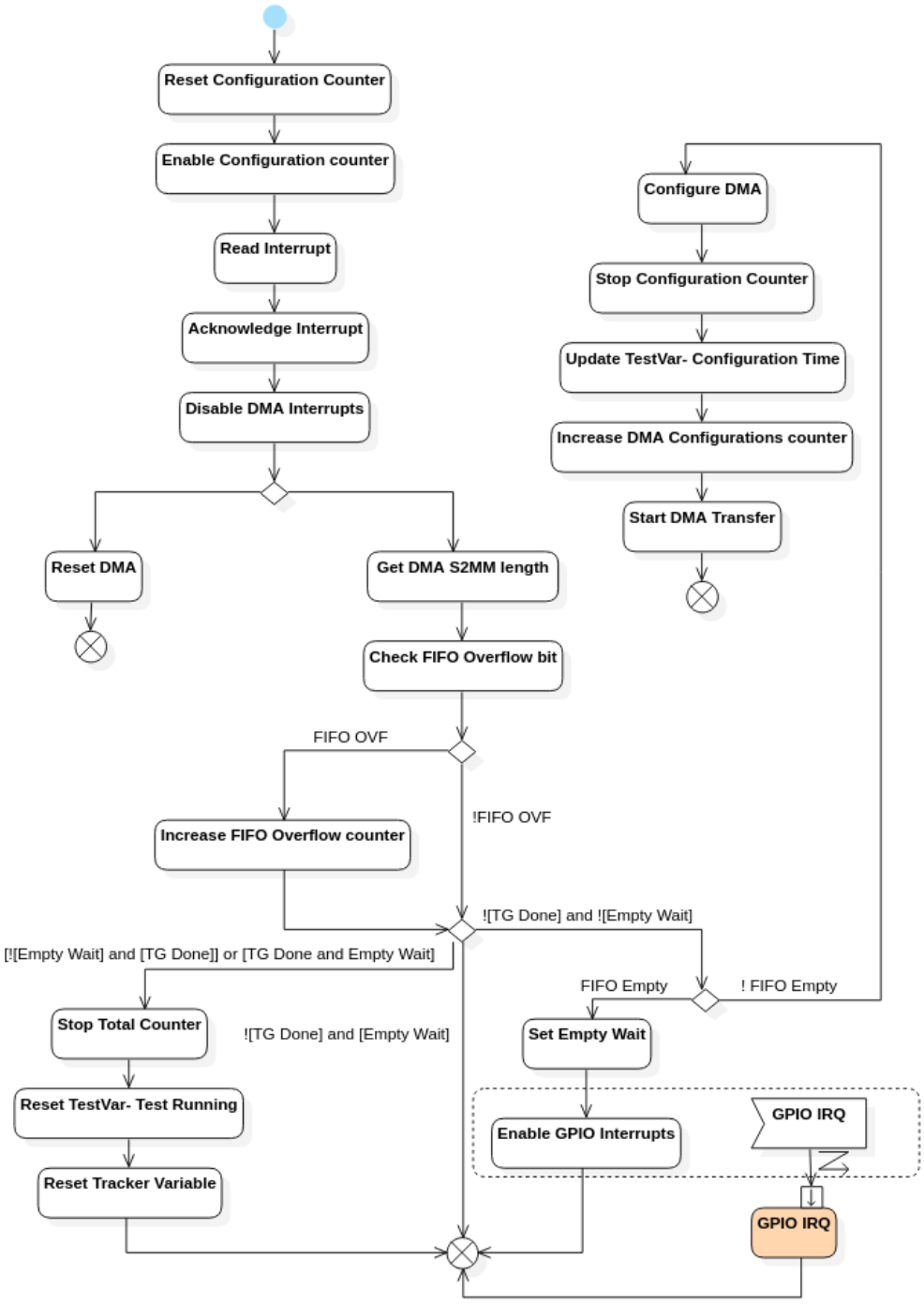


Figure 64: Scenario 3: Mode Test S2MM IRQ Flow Chart

Figure 64 shows the ISR for the S2MM interrupt. The interrupt begins by resetting the configuration counter and enabling it. Then the interrupt is read and acknowledged. If an error occurs the DMA is reset and the ISR ends, furthermore an error flag is activated for the DMA. If the interrupt is successful, the DMA S2MM register will be read. This register contains the number of bytes that were read during the last transfer. The overflow bit from the FIFO is also read. If an overflow is detected, the fifo_ovf_cnt variable in the TestVar structure is increased.

The empty_wait variable from the tracker variable indicates that in the last transfer, the FIFO was empty so no data could be read. In the case that this flag is activated and the traffic generator has not finished producing data, the ISR will end. If the traffic generator has not finished and the empty wait flag is deactivated, the FIFO empty flag will be checked. If the FIFO is empty, the empty wait flag will be activated, and the GPIO interrupt will be enabled. Ending the ISR. On the other hand, if the FIFO is not empty, the DMA will be configured to read all the data from the FIFO. The tracker variable is also updated when the DMA is configured. Next, the configuration counter will be stopped, and the TestVar structure will be updated by increasing the configuration time and number of configurations, finally, a new DMA Transfer will be started.

When none of these options happens, it will mean that the test has ended. In this case, the total counter will be stopped, and the test_running variable will be set to 0.

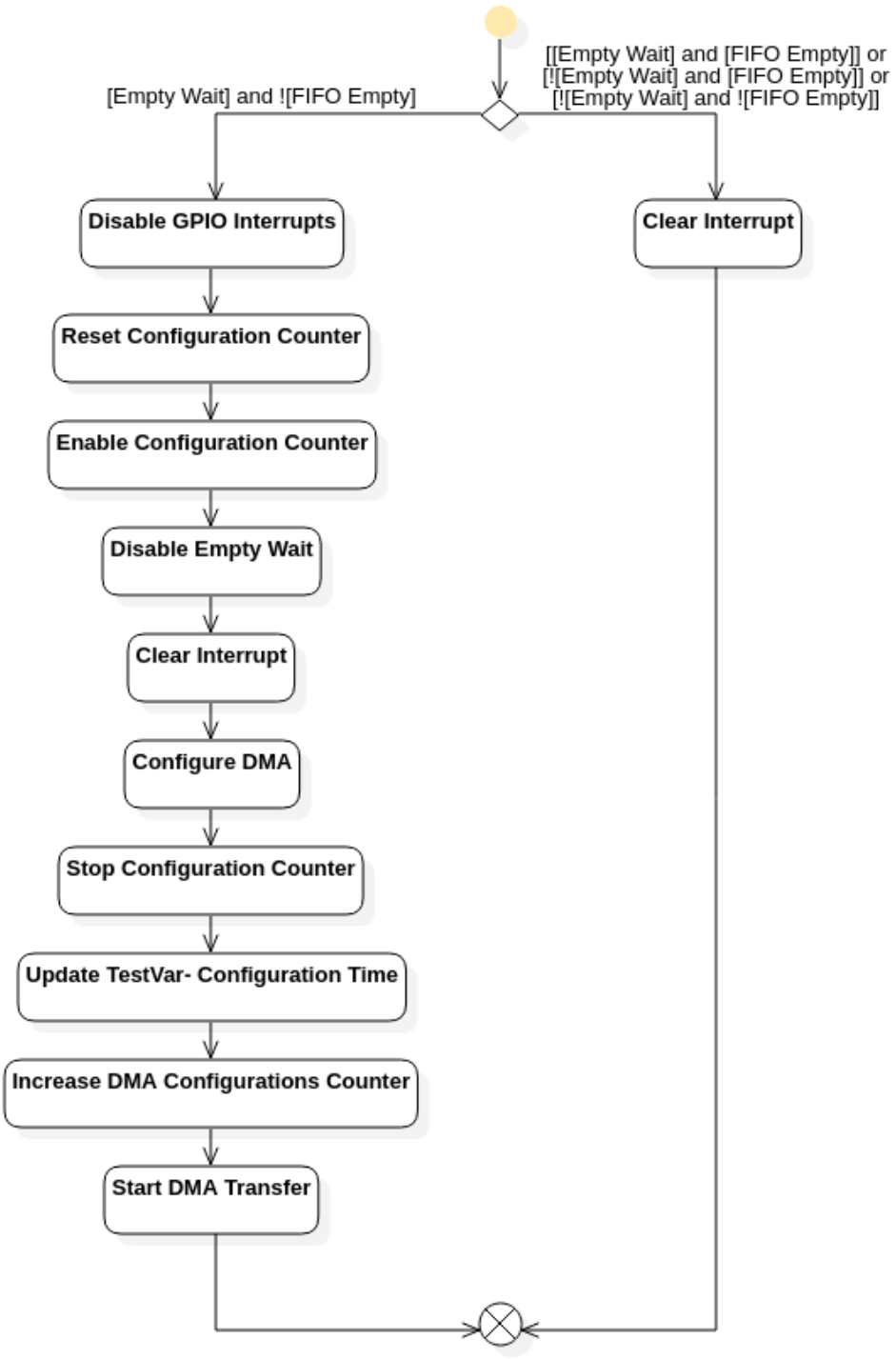


Figure 65: Scenario 3: Mode Test GPIO IRQ Flow Chart

Figure 65 shows the ISR for the GPIO interrupt. The first thing to be checked is is the empty wait flay is activated. If it is activated and the FIFO is still emptied, the interrupt will be cleared, and the ISR will end. If there is data in the FIFO, the GPIO interrupts will be disabled, and the configuration timer will be reset and enabled. At this point, the empty wait flag will be disabled, and the interrupt will be cleared. The interrupt

is cleared after performing all of this just in case another interrupt from the DMA appears. Next, the DMA will be configured, updating the tracker variable too. After the configuration, the configuration counter will be stopped, and the configuration time and counter from TestVar are updated. Finally, a new transfer is started, and the ISR ends.

One problem that was found when implementing the system was that when the FIFO was empty, and the CPU started a DMA transfer, the DMA would return an error and halt instead of waiting for data to arrive. For this reason, the GPIO interrupt was used, allowing the DMA to remain inactive if the FIFO was empty. For this reason, too the underflow flag from the FIFO was not used, as to activate it, the DMA had to read from the already empty FIFO which would return an error.

4 METHODOLOGY

The methodology followed for extracting the results will be explained.
Before the results are analysed, it is important to envision the different variables for the test, separating them into dependent and independent variables.

The variables for the first two scenarios are the same, so these will be explained together.

Scenario 1 & Scenario 2 variables:

- Dependent Variables
  - Throughput
  - Total Time
  - Drop Rate
  - Configuration Time
  - Latency
- Independent Variables
  - Clock Frequency
  - DMA/CDMA Burst Size
  - Use of Data Caches

Scenario 3 variables:

- Dependent Variables
  - Throughput
  - Total Time
  - Drop Rate
  - Configuration Time
  - DMA Configurations
  - FIFO Overflows
  - Latency
- Independent Variables
  - Traffic Generator Frequency
  - Traffic Generator Length
  - Traffic Generator Delay
  - FIFO Depth

When all tests were performed, every combination was tested ten times in order to account for random error. As all the tests were being executed, the results were transferred to a spreadsheet. When all the results were transferred to the spreadsheet, graphs and charts were created in order to visualise the data more clearly. For the first two scenarios, a total of 10KB of data were to be transferred.

For the third scenario, when all tests were performed for the first time, it was seen that no bytes were being dropped from the FIFO. This was solved by asserting the TVALID signal of the traffic generator to '1' as explained in section 3.4.1. It was advantageous to visualise the data through graphs and charts as it shows a clear representation of how each scenario works in different conditions and it also allowed to solve problems that had not been spotted during the implementation of the scenarios. The total amount of data transferred in each test was 33MB; this was done to be able to test the low values of the Traffic Generator Length parameter. If a low length is used, the traffic generator has to do more transfers. Due to having a 16-bit limitation for configuring the transfer count value, the total amount of bytes had to be reduced to accommodate each traffic generator length. On the third scenario though, a total transfer of 240MB was done in the last test to validate the NVIS scenario, sending half the amount of data as in both the real-time mode and normal modes due to a limitation in the transfer count registers of the traffic generator.

When describing the results for scenario 3, the frequency of the traffic generator will be used. This value can be easily converted to the throughput of data produced through the following equation:

$$\mathrm{THRP}[\mathrm{MBps}] = f_{TG} \cdot \mathrm{TDATA}[\mathrm{B}] \tag{7}$$

So for example, having the traffic generator working at 100MHz would mean having a throughput of $100\mathrm{MHz} \cdot 4\mathrm{B} = 400\mathrm{MBps}$. Equation **??** will only be true when the traffic generator delay is equal to 0. If a delay is configured, the average throughput can be calculated as:

$$\mathrm{THRP}[\mathrm{MBps}] = \frac{f_{TG} \cdot \mathrm{TDATA}[\mathrm{B}]^2 \cdot \mathrm{LENGTH}_{TG}[\mathit{WORDS}]}{\mathrm{TDATA}[\mathrm{B}] \cdot \mathrm{LENGTH}_{TG} + \mathrm{DELAY}_{TG}[\mathrm{CLK}]} \tag{8}$$

The spreadsheet with the different results obtained from all the tests may be found at the projects GitHub repository. The link can be seen in appendix e.

5 RESULTS

This chapter focuses on the quantitative aspect of this project. The effect that some internal parameters of each scenario have on the performance of the system will be studied; this includes variations of the working frequency and the DMA burst size, to name a few. The results obtained for the different scenarios will be shown and explained scenario by scenario; trying to reach conclusions on what the results mean and what they imply when using a DMA or CDMA IPcore in a Zynq device. Furthermore, they will allow spotting any weaknesses in the generated software that controls these IPcores and finding what specific configurations aid in achieving higher performances. Each set of results will be displayed through different graphs to facilitate comprehension, and each graph will be discussed separately.

5.1 SCENARIO 1

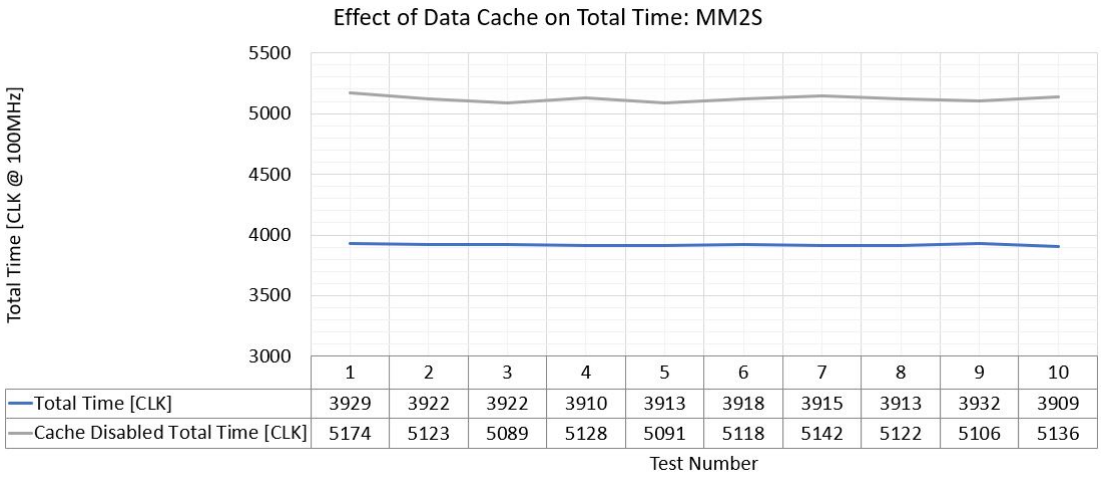
For the first scenario, different graphs will be shown that will allow viewing the relationship between different parameters:

1. Effect of disabling the Data Cache on: Total time, Latency, Configuration Time.
2. Effect of DMA Burst Size on: Total time, Latency, Configuration Time.
3. Effect of PL frequency on: Throughput, Latency, Configuration Time.

Each of these will be done for MM2S and S2MM transfers. All tests for the first scenario were configured to transfer 10KB of data.

5.1.1 *Effect of Data Cache*

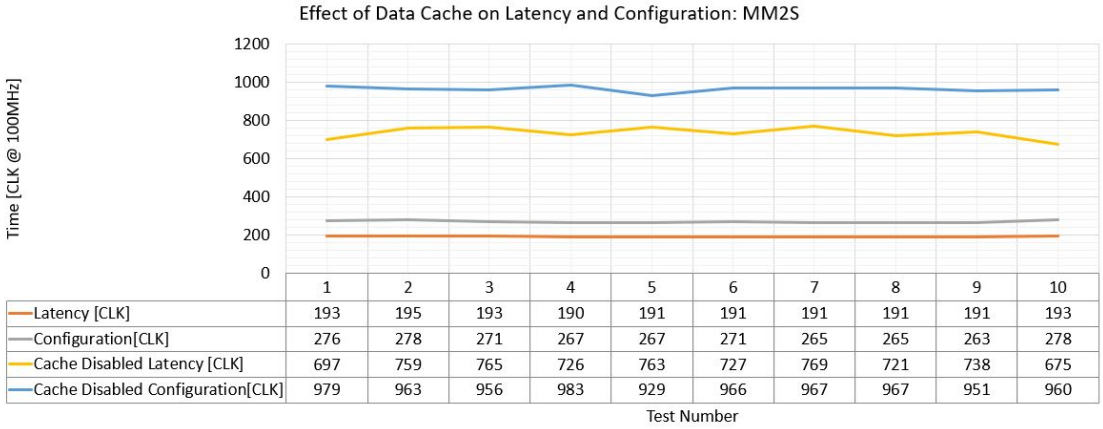
We will first see how the data cache affects MM2S transfers.



	1	2	3	4	5	6	7	8	9	10
Total Time [CLK]	3929	3922	3922	3910	3913	3918	3915	3913	3932	3909
Cache Disabled Total Time [CLK]	5174	5123	5089	5128	5091	5118	5142	5122	5106	5136

Figure 66: Effect of Data Cache on Total Time, MM2S transfers.

Figure 66 shows how the total time varies when enabling or disabling the Data Cache. The blue line represents the different tests carries out with the data cache enabled. It can be seen that there is an incredible difference when the data cache is enabled. This is representative of how efficient the Cache is and how much it can improve a system's performance.

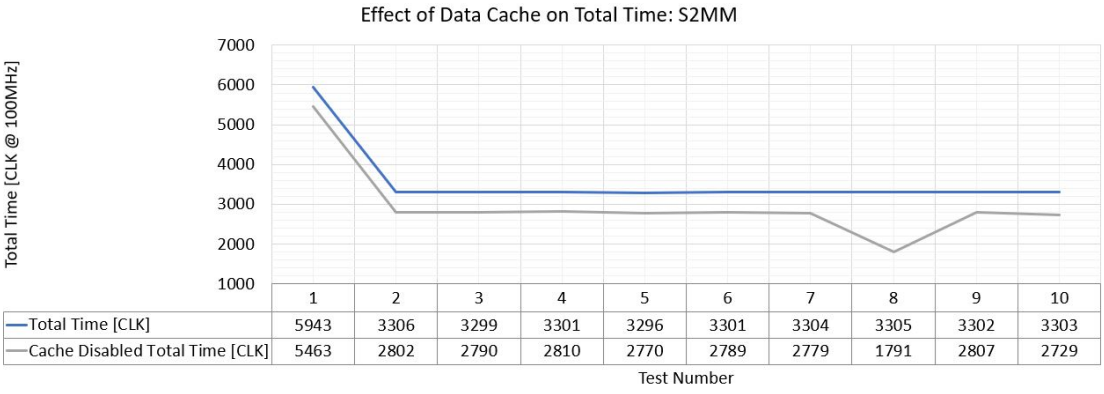


	1	2	3	4	5	6	7	8	9	10
Latency [CLK]	193	195	193	190	191	191	191	191	191	193
Configuration[CLK]	276	278	271	267	267	271	265	265	263	278
Cache Disabled Latency [CLK]	697	759	765	726	763	727	769	721	738	675
Cache Disabled Configuration[CLK]	979	963	956	983	929	966	967	967	951	960

Figure 67: Effect of Data Cache on Latency and Configuration Time, MM2S transfers.

Through the graph in figure 67 the effect of the using a Data Cache can be seen for both Latency and configuration times. The latency drastically improves, which is to be expected with the results from figure 66. Moreover, the configuration time decreases when the data cache is enabled. Again, it can be seen that the use of the Data Cache can increase the system's efficiency. As both of these times improve when using the Data Cache, it makes sense that the total time taken to perform the transfers will also improve. What has the most significant effect on the total time taken is the latency time, as only one configuration is done per test in the first scenario.

The effect of the data cache will be now shown for S2MM transfers.

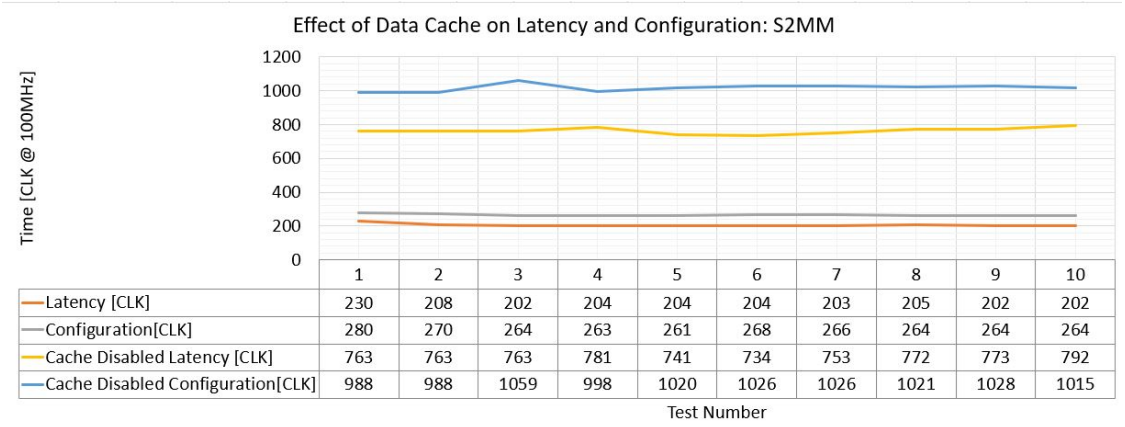


	1	2	3	4	5	6	7	8	9	10
Total Time [CLK]	5943	3306	3299	3301	3296	3301	3304	3305	3302	3303
Cache Disabled Total Time [CLK]	5463	2802	2790	2810	2770	2789	2779	1791	2807	2729

Figure 68: Effect of Data Cache on Total Time, S2MM transfers.

It can be seen how, for the first test, there appears to be an increase in time for both tests. This test was repeated multiple times with the same result. It will be marked as an anomalous result since the other tests did not experience this. We can also see that

there is not such a vast difference between the total time when the Cache is enabled or disabled. This may be because data is being written to the SRAM and not read.



	1	2	3	4	5	6	7	8	9	10
Latency [CLK]	230	208	202	204	204	204	203	205	202	202
Configuration[CLK]	280	270	264	263	261	268	266	264	264	264
Cache Disabled Latency [CLK]	763	763	763	781	741	734	753	772	773	792
Cache Disabled Configuration[CLK]	988	988	1059	998	1020	1026	1026	1021	1028	1015

Figure 69: Effect of Data Cache on Latency and Configuration Time, S2MM transfers.

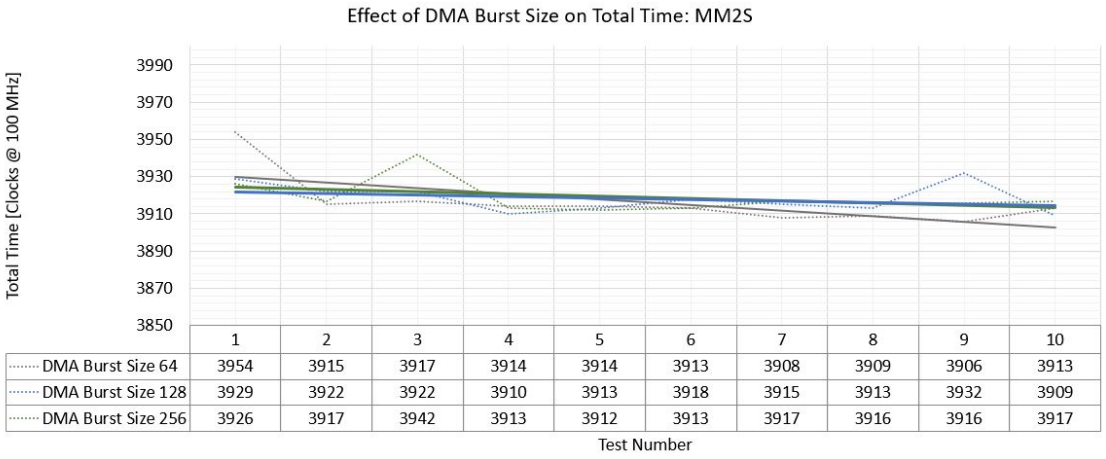
As seen in figure 69, the results are quite similar to the ones obtained in the MM2S test. Which is quite peculiar as the total time does not seem to be affected in such a way. As in the MM2S test, here, the configuration time will not have a large effect on the total time taken. It can also be seen that the latency time decreases below the one measured in the MM2S test. This implies that there could be a bottleneck somewhere between the FIFO and the SRAM. It is probably not due to the SRAM, as this memory works at a much higher frequency than 100MHz. Meaning that it may be due to the way the memory drivers in the PS perform write transactions to the SRAM. It also is probably not due to the HP0 port not having enough bandwidth as it can withstand much higher frequencies and it would have the same effect in MM2S transfers.

These results can make us see than when implementing the third scenario; we may think that disabling the data cache would be the best decision as only S2MM transfers are being made. While the last part is right, the data cache was enabled for scenario 3 as when the TX channel would be implemented, having the Cache disabled would significantly reduce the performance of the system.

5.1.2 *Effect of* DMA *Burst Size*

The effect of the DMA burst size will now be studied. Three values will be tested: 64, 128 and 256 with 256 being the largest value permitted by the DMA block. Due to the variability of the results, trend lines were used to spot any pattern that may not be seen viewing the results directly. The dotted lines represent the results whereas the filled lines represent the trend lines.

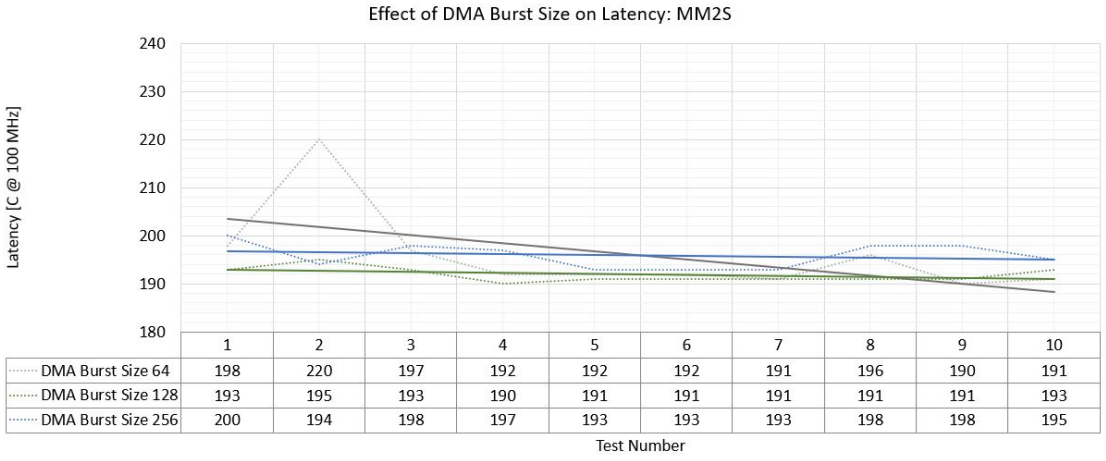
First, we will look at the MM2S transfers, after which point we will look at the S2MM transfers.



	1	2	3	4	5	6	7	8	9	10
DMA Burst Size 64	3954	3915	3917	3914	3914	3913	3908	3909	3906	3913
DMA Burst Size 128	3929	3922	3922	3910	3913	3918	3915	3913	3932	3909
DMA Burst Size 256	3926	3917	3942	3913	3912	3913	3917	3916	3916	3917

Figure 70: Effect of DMA Burst Size on Total Time, MM2S transfers.

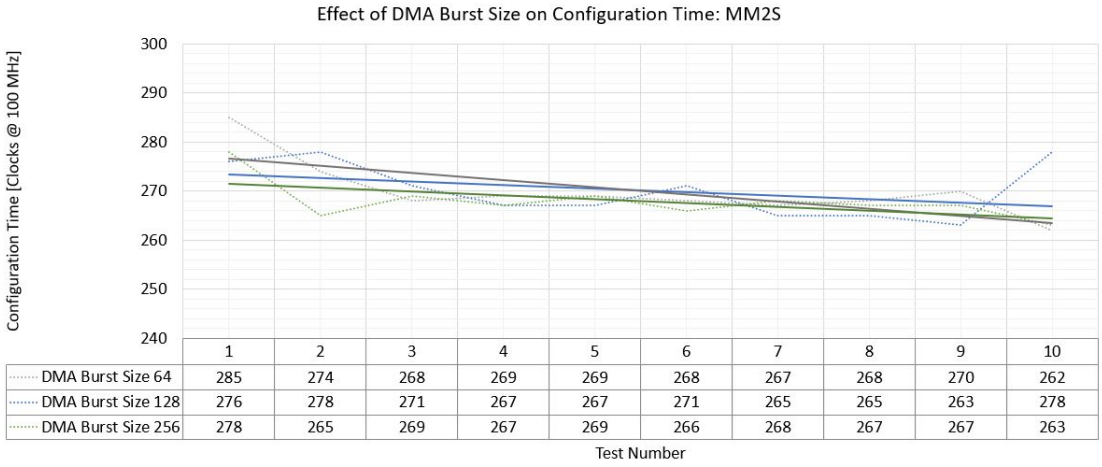
Figure 70 shows the effect of the burst size on the total time taken to perform the tests. It can be seen that there is no significant effect on the total time taken when changing the burst size. Nevertheless, looking at the dotted lines, we can see that for the highest value of 256, there is less variability than in the other two combinations. With the value of 64 showing the most considerable variability in the results shown. Nevertheless, this variability does not surpass the 100 clocks.



	1	2	3	4	5	6	7	8	9	10
DMA Burst Size 64	198	220	197	192	192	192	191	196	190	191
DMA Burst Size 128	193	195	193	190	191	191	191	191	191	193
DMA Burst Size 256	200	194	198	197	193	193	193	198	198	195

Figure 71: Effect of DMA Burst Size on Latency, MM2S transfers.

For the latency time, shown in figure 71, it can be seen how the larger burst size minimally improves on the latency. Furthermore, it can also be seen that for smaller burst sizes, the latency time varies more.

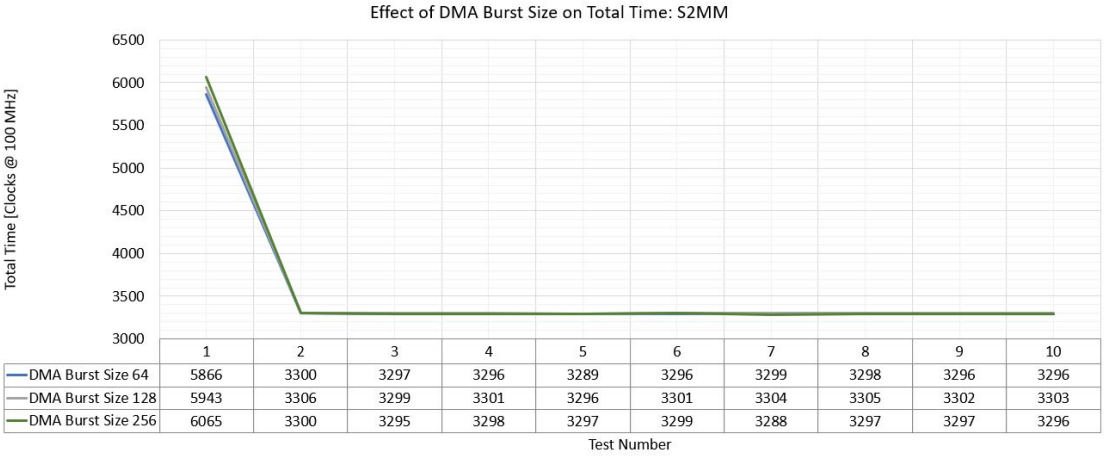


	1	2	3	4	5	6	7	8	9	10
DMA Burst Size 64	285	274	268	269	269	268	267	268	270	262
DMA Burst Size 128	276	278	271	267	267	271	265	265	263	278
DMA Burst Size 256	278	265	269	267	269	266	268	267	267	263

Figure 72: Effect of DMA Burst Size on Configuration Time, MM2S transfers.

As seen in figure 72, it seems that larger burst sizes reduce the configuration time, but it does not seem that increasing the burst size reduces the variability of the results. Again, we have to take into consideration that the variability shown in this graph is minimal and that as expected, the configuration time does not seem to depend on the DMA Burst size.

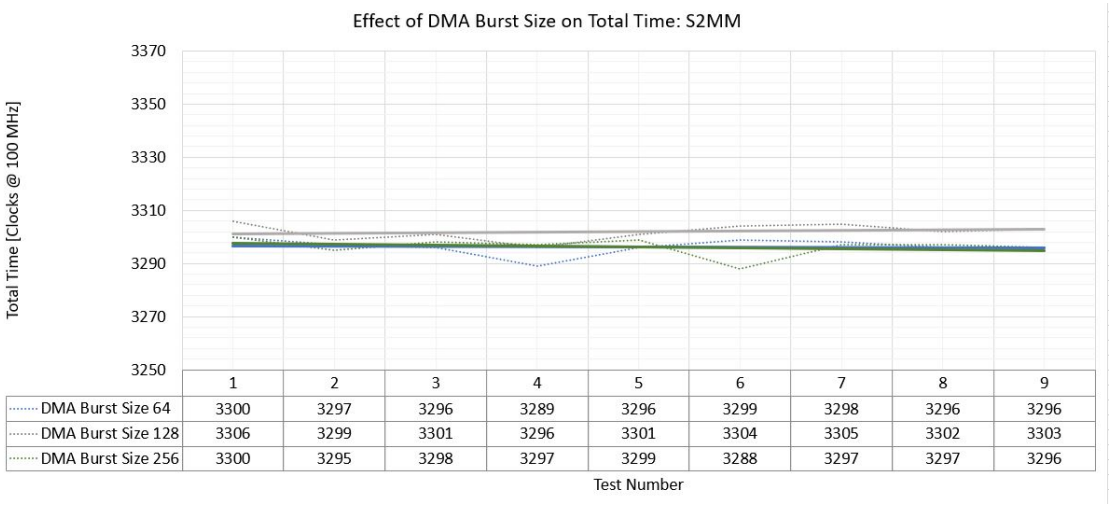
The effects on the S2MM tests will be now studied.



	1	2	3	4	5	6	7	8	9	10
DMA Burst Size 64	5866	3300	3297	3296	3289	3296	3299	3298	3296	3296
DMA Burst Size 128	5943	3306	3299	3301	3296	3301	3304	3305	3302	3303
DMA Burst Size 256	6065	3300	3295	3298	3297	3299	3288	3297	3297	3296

Figure 73: Effect of DMA Burst Size on Total Time (1/2), S2MM transfers.

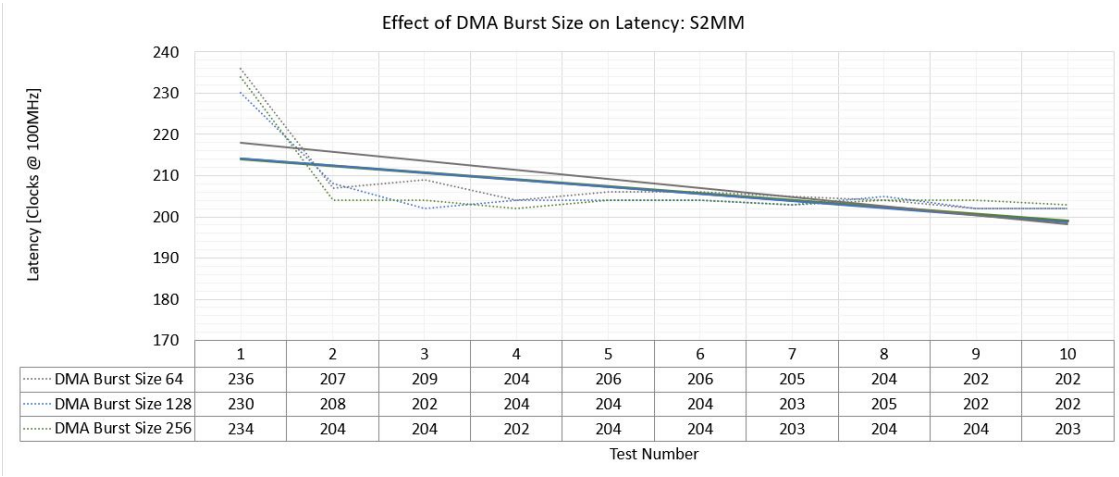
As in the previous S2MM tests, we can see in figure 74 that the first value is anomalous, and it avoids the results from being seen clearly. For this reason, the first value was eliminated, and the graph was plotted again.



	1	2	3	4	5	6	7	8	9
DMA Burst Size 64	3300	3297	3296	3289	3296	3299	3298	3296	3296
DMA Burst Size 128	3306	3299	3301	3296	3301	3304	3305	3302	3303
DMA Burst Size 256	3300	3295	3298	3297	3299	3288	3297	3297	3296

Figure 74: Effect of DMA Burst Size on Total Time (2/2), S2MM transfers.

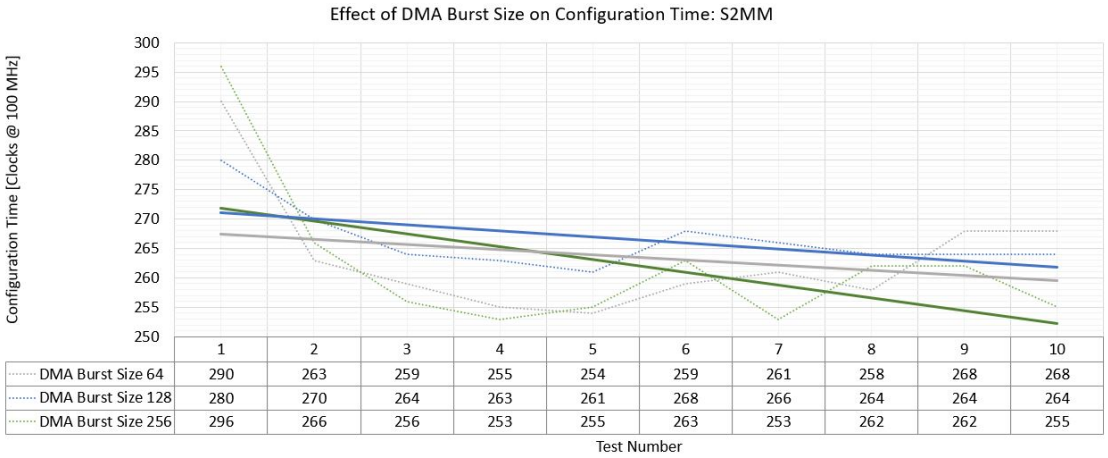
Figure 74 shows the graph plotted without the anomalous point. It can be seen that there is a notable difference between using a value of 64 or a larger value. Furthermore, we can see again that increasing the DMA burst size has reduced the variability of the results.



	1	2	3	4	5	6	7	8	9	10
DMA Burst Size 64	236	207	209	204	206	206	205	204	202	202
DMA Burst Size 128	230	208	202	204	204	204	203	205	202	202
DMA Burst Size 256	234	204	204	202	204	204	203	204	204	203

Figure 75: Effect of DMA Burst Size on Latency, S2MM transfers.

Figure 75 shows how on average, using a larger burst size reduces the latency. Nevertheless, there appears to be no correlation with the variability of the results and the Burst size used.



	1	2	3	4	5	6	7	8	9	10
DMA Burst Size 64	290	263	259	255	254	259	261	258	268	268
DMA Burst Size 128	280	270	264	263	261	268	266	264	264	264
DMA Burst Size 256	296	266	256	253	255	263	253	262	262	255

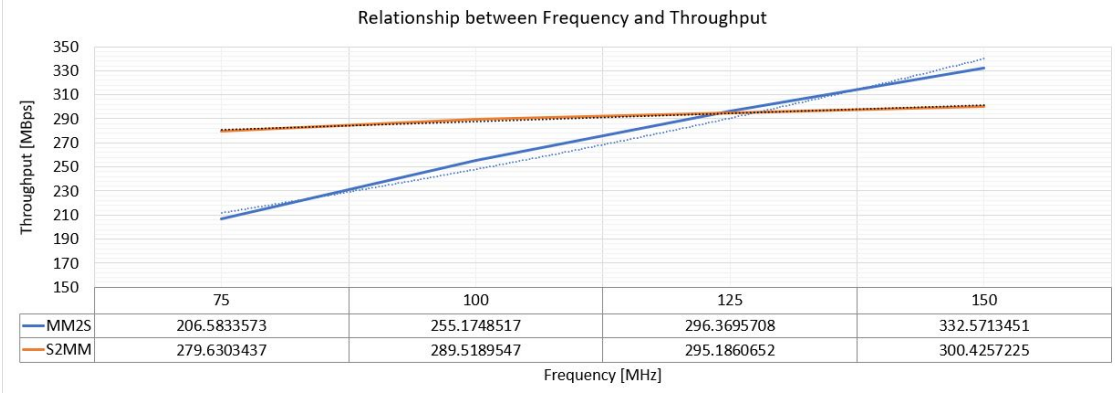
Figure 76: Effect of DMA Burst Size on Configuration Time, S2MM transfers.

From the trend lines in figure 76, it may seem that increasing the burst size can decrease the configuration time. Nevertheless, this could be attributed to the first anomalous result. We can also see how the variability of the results is unaffected by the burst size in the DMA.

From the results obtained, it was concluded that using a larger DMA burst size would be beneficial. As even though there was only a small increase in performance, this increase could make a difference when thousands of transfers are to be performed. Furthermore, reducing the variability of the scenario's performance may also help to increase its reliability.

5.1.3 *Effect of* PL *Frequency*

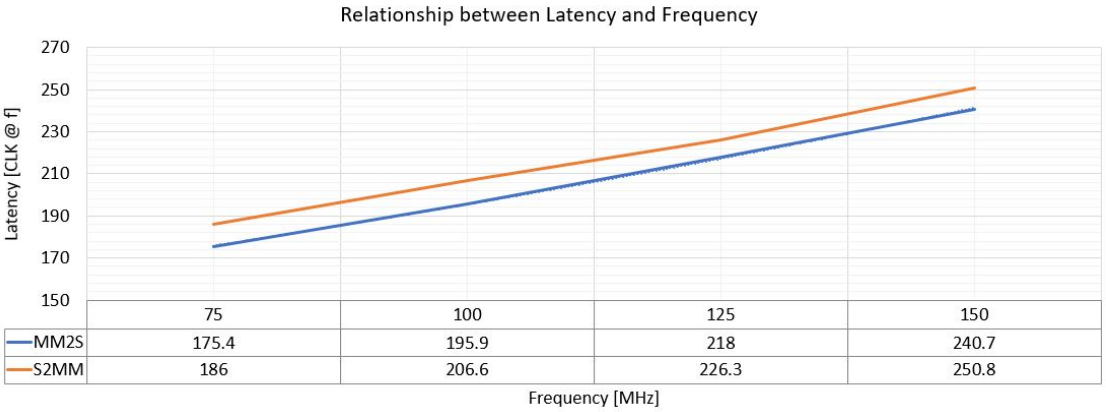
The effect of the PL frequency will now be studied. The time will be measured in clocks at the selected frequencies; this is useful as if any nonlinearities are spotted, they may be attributed to nonlinearities in the IPcores used. This time, trend lines were plotted on dotted lines. Both S2MM and MM2S tests were plotted on the same graphs.



	75	100	125	150
MM2S	206.5833573	255.1748517	296.3695708	332.5713451
S2MM	279.6303437	289.5189547	295.1860652	300.4257225

Figure 77: Effect of PL frequency on throughput, MM2S and S2MM transfers.

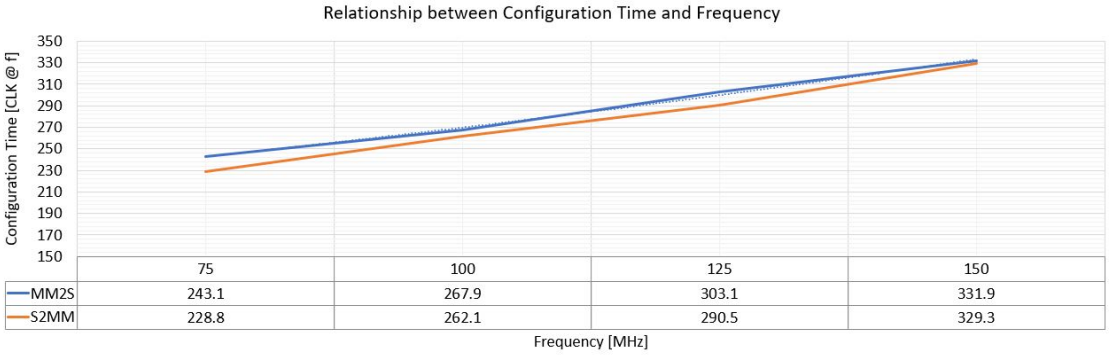
Figure 79 shows the effect that the PL frequency has on the throughput between PS and PL. It can be seen that for MM2S tests, the throughput increases with the frequency as expected, but as higher frequencies are achieved, the line's gradient decreases. This shows some of the nonlinearities mentioned. Furthermore, for the S2MM tests, what is curious is how the throughput increases to a lesser extent than in the MM2S tests and shows a stagnation at higher frequencies which could be attributed to the DMA IPcore.



	75	100	125	150
MM2S	175.4	195.9	218	240.7
S2MM	186	206.6	226.3	250.8

Figure 78: Effect of PL frequency on Latency, MM2S and S2MM transfers.

Figure 79 shows how in both S2MM and MM2S tests there is a linear relationship between the PL's frequency and the latency time. It also shows how the latency for S2MM tests is always higher than in MM2S tests.



	75	100	125	150
MM2S	243.1	267.9	303.1	331.9
S2MM	228.8	262.1	290.5	329.3

Figure 79: Effect of PL frequency on Configuration Time, MM2S and S2MM transfers.

The graph in figure 79 shows how the configuration time is greater for MM2S transfers than for S2MM transfers. We also see that the relationship between PL frequency and configuration time is not completely linear.

This last result could mean that the nonlinearity found in figure 79 may not be caused by the DMA as also the latency time is linear. This could mean that some other intermediate block is causing this nonlinearity. It could be due to the FIFO or due to an AXI Interconnect block. Nevertheless, there is not enough information to reach a conclusion with these graphs.

Moreover, the number of bytes dropped was measured in all the tests, no bytes were dropped in any of the tests. This was expected for this scenario as data was being moved from a full FIFO to the SRAM and the other way round. The only way for a byte to be dropped would be if the Xilinx IPcores were not working correctly, which was certainly not the case.

The results obtained in the second scenario will be now shown. The same metrics as in the first scenario will be modified, and their effects on the system's performance will be studied.

Different graphs showing the relationship between different parameters will be shown and analysed:

1. Effect of disabling the Data Cache on: Total time, Latency, Configuration Time.
2. Effect of CDMA Burst Size on: Total time, Latency, Configuration Time.
3. Effect of PL frequency on: Throughput, Latency, Configuration Time.

These graphs will reflect the impact on the design's performance for the four different types of transfers present in scenario 2. The results shown are calculated by averaging a series of 10 results for each test. All the tests shown for scenario 2 were configured to transfer 10KB of data.

5.2.1 *Effect of Data Cache*

The effect that enabling or disabling the data cache has on the performance of the scenario will be studied.

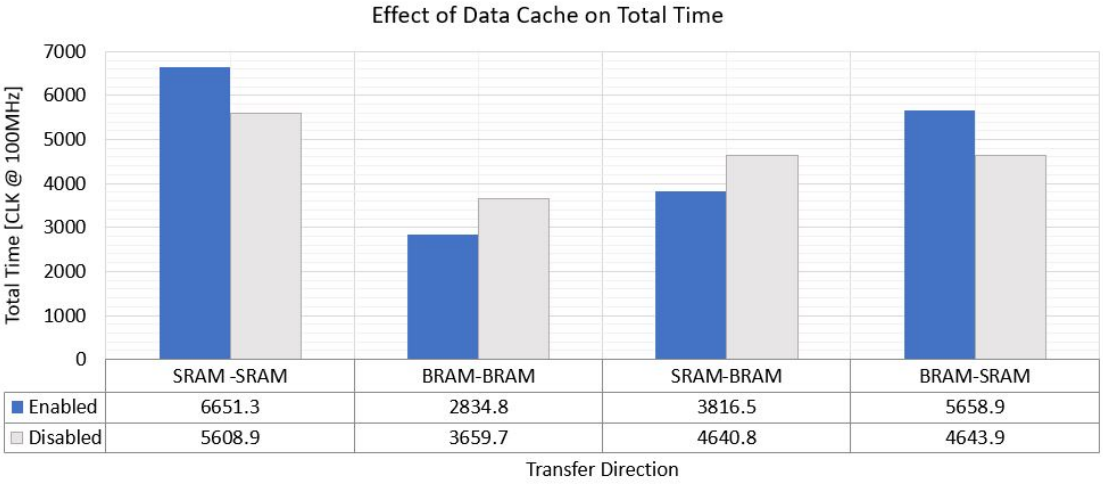


Figure 80: Effect of Data Cache on Total Time

Figure 80 shows the effect the Data Caches have on the total time taken to perform the test. It can be seen that for transfers where the destination was the SRAM, the Data Cache actually reduced the performance of the design, increasing the total time taken for the transfer. We see the opposite effect for the other two transfers, where the use of the Data Cache reduces the total time taken for the transfer to be completed. It is also interesting to see how BRAM to BRAM transfers are the quickest, as no data has to cross from the PL to the PS. On the other hand, for SRAM to SRAM transfers, the total time taken is the largest, the reason for this is that the data has to cross the PS-PL boundary twice, drastically increasing the time taken to perform a transfer. Similarly

to the S2MM transfers, we can see that BRAM-SRAM transfers take much longer than SRAM-BRAM transfers. This could be caused by the increased time taken to write in the SRAM or due to the CDMA having asymmetrical read and write latencies.

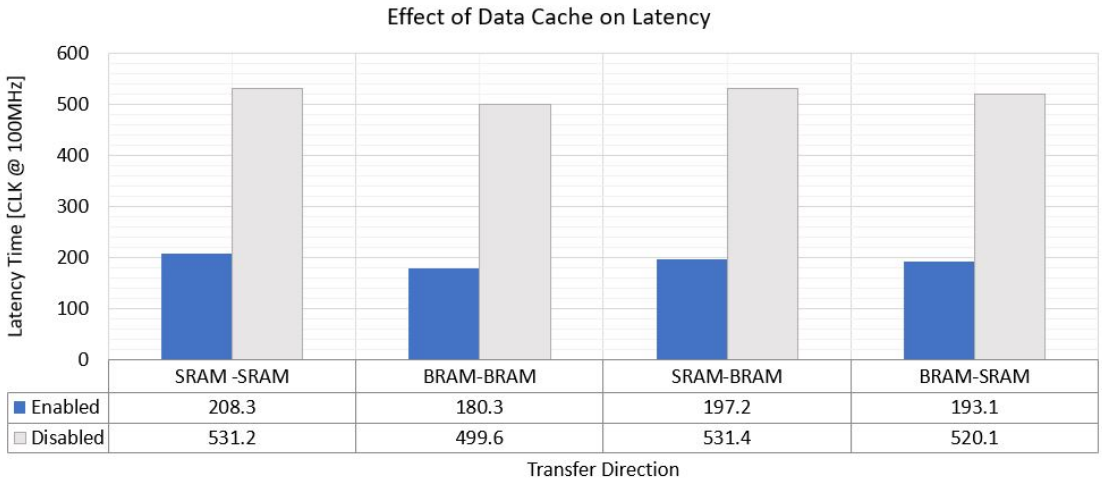


Figure 81: Effect of Data Cache on Latency

From figure 81 it can be seen that in fact the latency is not significantly affected by the direction of the transfer, therefore the asymmetry described previously may come from different read and write latencies in the SRAM caused by either the hardware or the PS's memory driver. Furthermore, it can be seen how the use of the data cache reduces the latency in all transfers, with the SRAM-SRAM transfer having the most substantial latency.

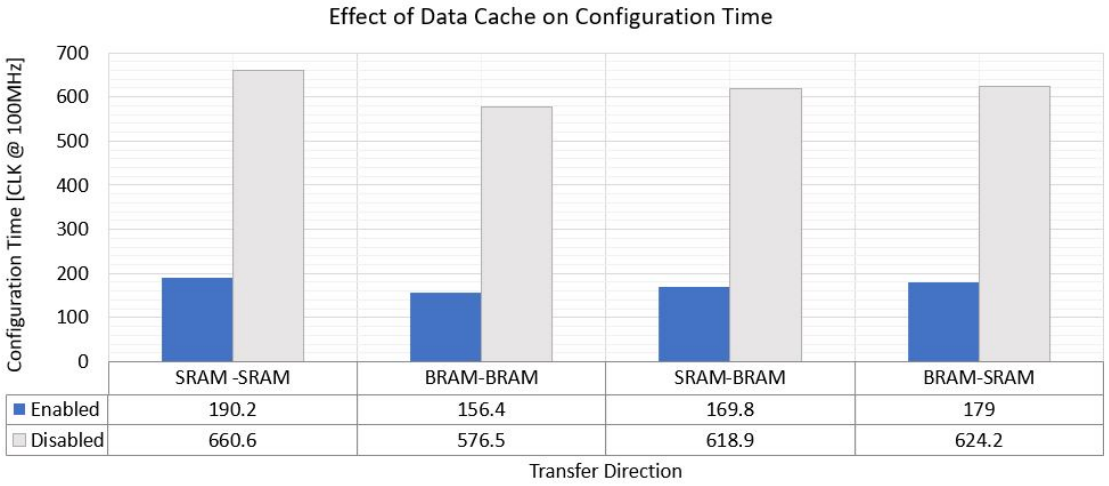
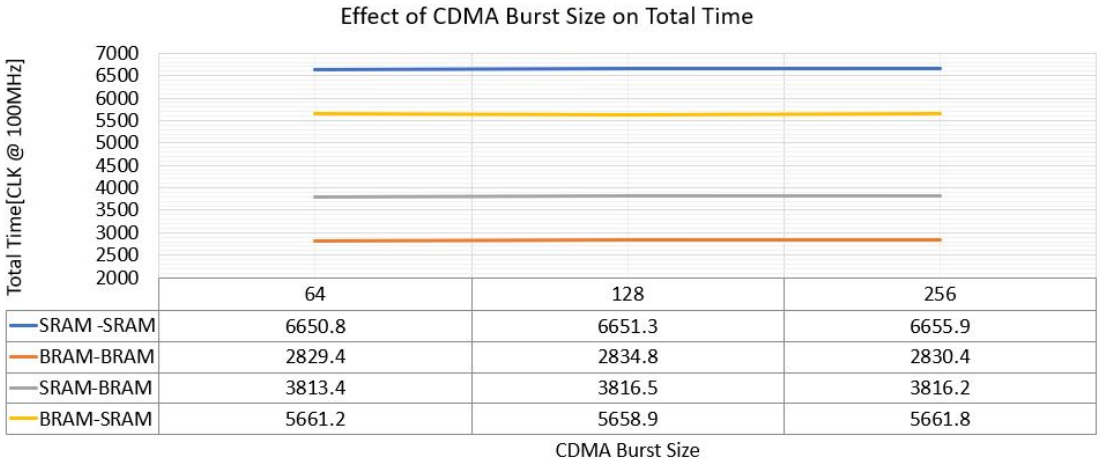


Figure 82: Effect of Data Cache on Configuration Time

Finally, a similar effect can be seen in figure 82, where the configuration time decreases when caches are used in all the transfers. This is quite interesting as both latency and configuration times always benefit from the use of the Data Cache, but the total time taken doesn't, which supports the idea that there may be an inefficiency somewhere in the Zynq. More information would be needed to deduce the exact location of this inefficiency.

5.2.2 *Effect of CDMA Burst Size*

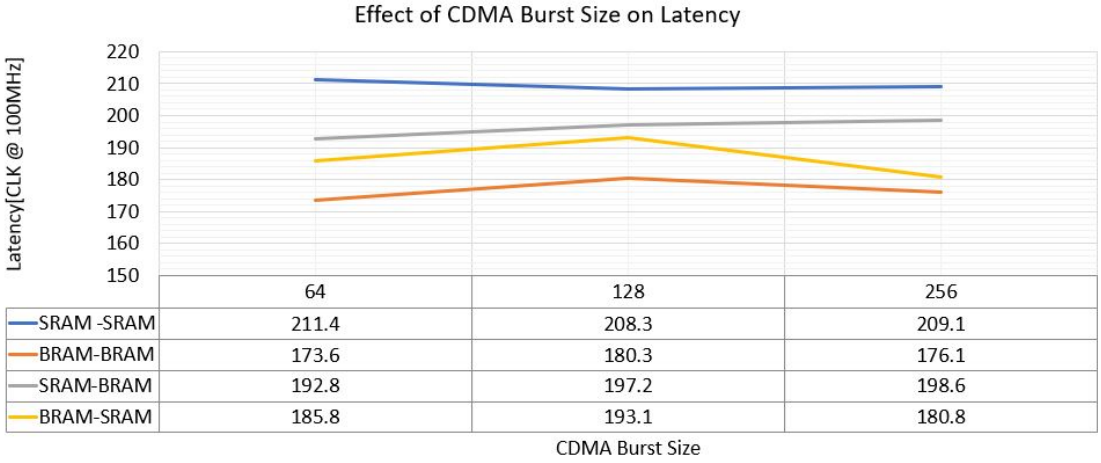
Three different values for the CDMA burst size will be used as in the first scenario.



	64	128	256
SRAM -SRAM	6650.8	6651.3	6655.9
BRAM-BRAM	2829.4	2834.8	2830.4
SRAM-BRAM	3813.4	3816.5	3816.2
BRAM-SRAM	5661.2	5658.9	5661.8

Figure 83: Effect of CDMA Burst Size on Total Time

The same pattern that was seen when studying the effect of the Cache on this scenario can be seen in figure 83. Where SRAM-SRAM transfers are the slowest and BRAM-**BRAM,!** (**BRAM,!**) transfers are the quickest. Furthermore, BRAM-SRAM are slower than SRAM-BRAM transfers. If we look at the result table in the figure, it can be seen that increasing the Burst Size reduces the performance of the design, increasing the time taken to perform the transfer. This increase is minimal and does not make much of a difference when transferring 10KB of data. Nevertheless, it is interesting to see that the effect that the CDMA burst size has on the total time is opposite to what was expected.

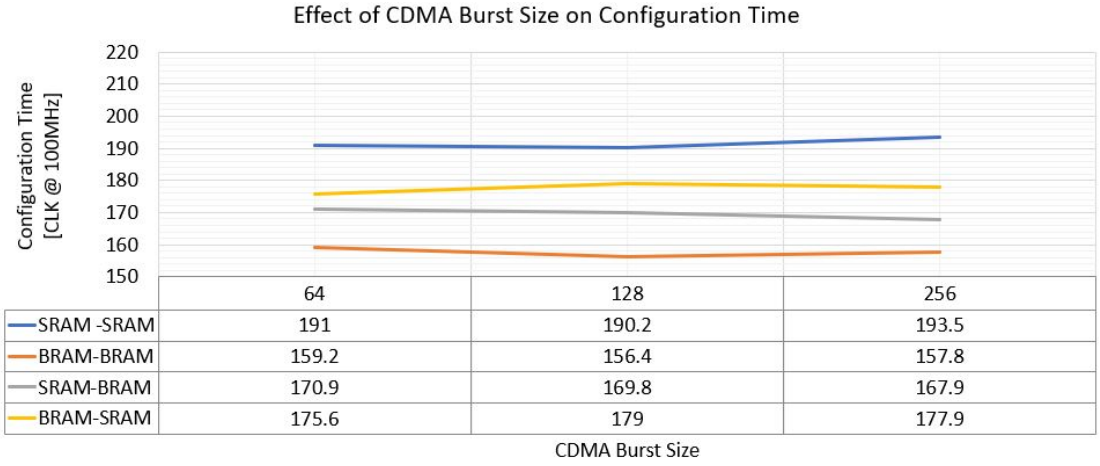


	64	128	256
SRAM -SRAM	211.4	208.3	209.1
BRAM-BRAM	173.6	180.3	176.1
SRAM-BRAM	192.8	197.2	198.6
BRAM-SRAM	185.8	193.1	180.8

Figure 84: Effect of CDMA Burst Size on Latency

Figure 84 shows the effect the burst size has on the latency. It seems like as the burst size is increased, the latency time increases too. It is notable to mention that for a value of 128, the latency peaks in both BRAM to BRAM transfers and BRAM to SRAM transfers. For SRAM-BRAM transfers, the latency increases with the burst size.

Moreover, with SRAM to SRAM transfers, a value of 128 gives the lowest latency time. No conclusions can be extracted directly from this data, but it shows that common sense may not always apply when working with a Zynq device.



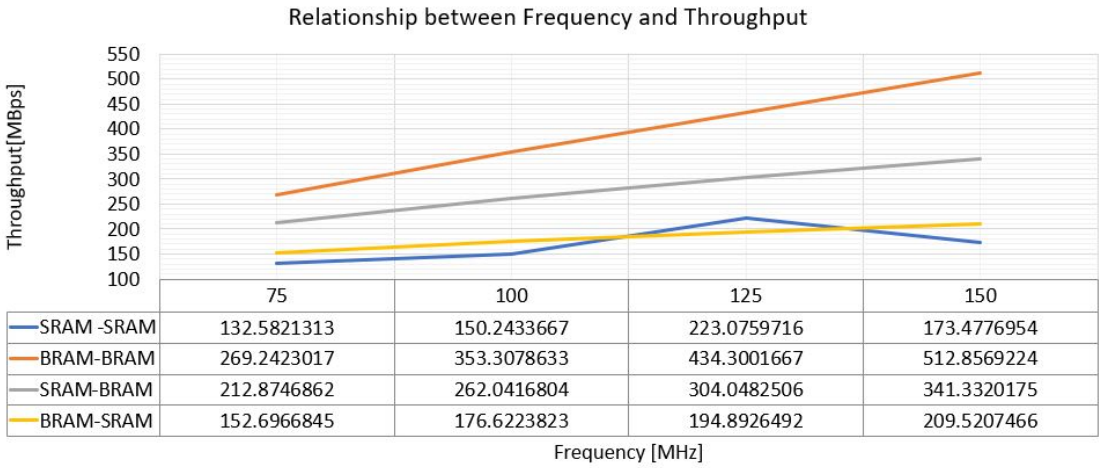
	64	128	256
SRAM -SRAM	191	190.2	193.5
BRAM-BRAM	159.2	156.4	157.8
SRAM-BRAM	170.9	169.8	167.9
BRAM-SRAM	175.6	179	177.9

Figure 85: Effect of CDMA Burst Size on Configuration Time

Finally, we see how the CDMA burst size has no apparent relationship with the configuration time. However, there seem to be differences when using different Burst Sizes. There should be no relationship between the configuration time and the CDMA burst size as no transfers are being made during this time. More information is needed to conclude with these results.

5.2.3 *Effect of PL Frequency*

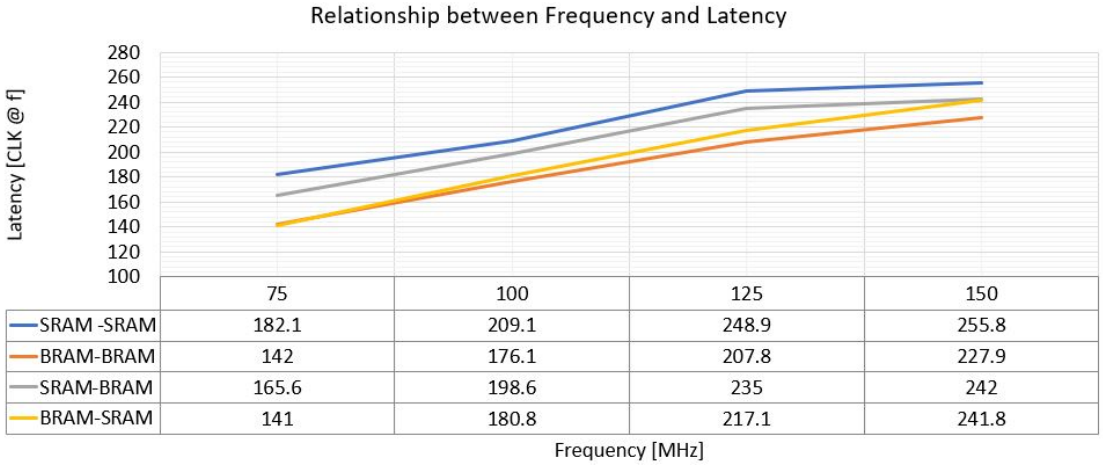
The PL frequency will be varied in order to see how the scenario responds to different frequencies. As the clocks are referenced to each frequency, it is useful to focus on the gradient of the graphs and not the numerical values, allowing to spot nonlinearities in the design.



	75	100	125	150
SRAM -SRAM	132.5821313	150.2433667	223.0759716	173.4776954
BRAM-BRAM	269.2423017	353.3078633	434.3001667	512.8569224
SRAM-BRAM	212.8746862	262.0416804	304.0482506	341.3320175
BRAM-SRAM	152.6966845	176.6223823	194.8926492	209.5207466

Figure 86: Effect of PL Frequency on Throughput

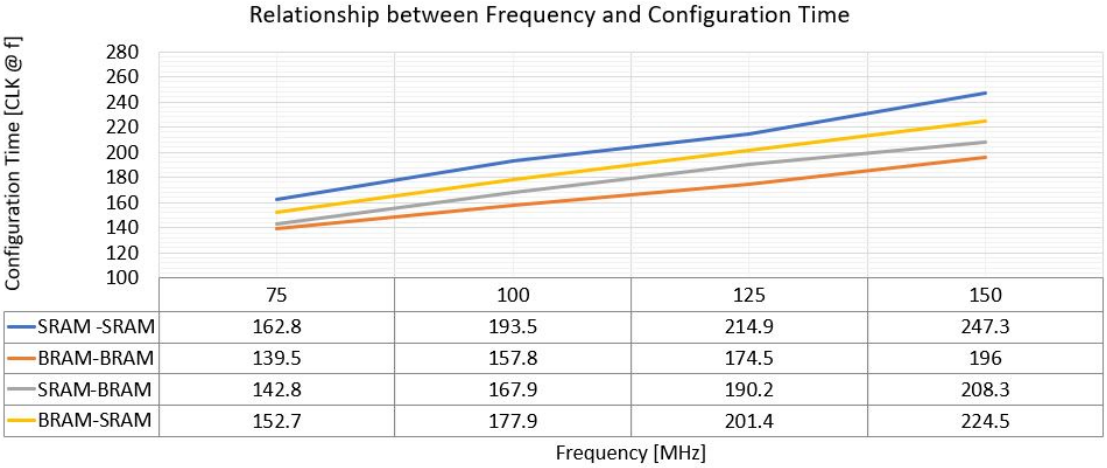
Figure 86 shows the effect that the PL frequency has on the throughput. From this figure, if we look at the gradients of the different lines, we can see that BRAM-BRAM transfers have the biggest gradient. This means that the PL and CDMA behave well at these frequencies, so as expected, the throughput increases linearly with the frequency. It can also be seen that when the data has to cross the PS-PL domain, this gradient decreases, which may be caused due to the HP0 port not having enough bandwidth for the transfers or due to the SRAM. Furthermore, for SRAM-SRAM transfers, it can be seen how at 125MHz, there is a peak in throughput, which seems counter-intuitive and could be an anomalous behaviour. It may also be because the SRAM does not manage simultaneous read and writes correctly, which is amplified when the frequency of the transfers is increased.



	75	100	125	150
SRAM -SRAM	182.1	209.1	248.9	255.8
BRAM-BRAM	142	176.1	207.8	227.9
SRAM-BRAM	165.6	198.6	235	242
BRAM-SRAM	141	180.8	217.1	241.8

Figure 87: Effect of PL Frequency on Latency

From figure 87 it can be seen that at more than 125MHz, there is a decrease in the gradient of the lines, which could mean that we are getting close to the 150MHz maximum frequency accepted by the CDMA[25]. Furthermore, for low frequencies, it can be seen that BRAM-SRAM and SRAM-BRAM have similar latencies, which supports the idea that there appears to be a bottleneck somewhere between the CDMA and the SRAM.



	75	100	125	150
SRAM -SRAM	162.8	193.5	214.9	247.3
BRAM-BRAM	139.5	157.8	174.5	196
SRAM-BRAM	142.8	167.9	190.2	208.3
BRAM-SRAM	152.7	177.9	201.4	224.5

Figure 88: Effect of PL Frequency on Configuration Time

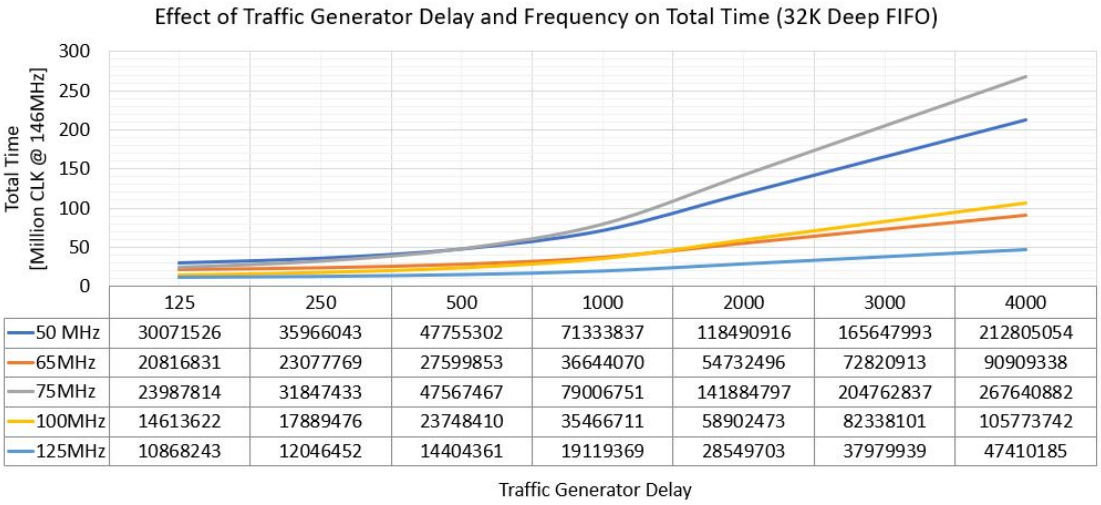
Finally, through figure 88, it can be seen that even though the relationship between frequency and configuration time is not entirely linear, there seem to be no anomalies or sudden dips in the gradient. This means that the GPo port is working well at these different frequencies, which matches what was expected.

The results obtained for the third scenario will be shown through different graphs and tables. The relationships graphed and analysed are the following:

1. Effect of traffic generator length, and frequency on: Total time, Configuration time, Number of FIFO overflows, drop rate. With a FIFO Depth of 32K and a traffic generator length of 1024.
2. Effect of traffic generator delay, and frequency on: Total time, Configuration time, Number of FIFO overflows, drop rate. With a FIFO Depth of 32K and a traffic generator delay of 0.

The tests were performed configuring a transfer of 33MB as explained in section 4, allowing to test small traffic generator length values. The DMA's frequency was maintained at 146MHz for all the tests. Furthermore, the average latency of the scenario was shown to be 270CLK @146MHz.

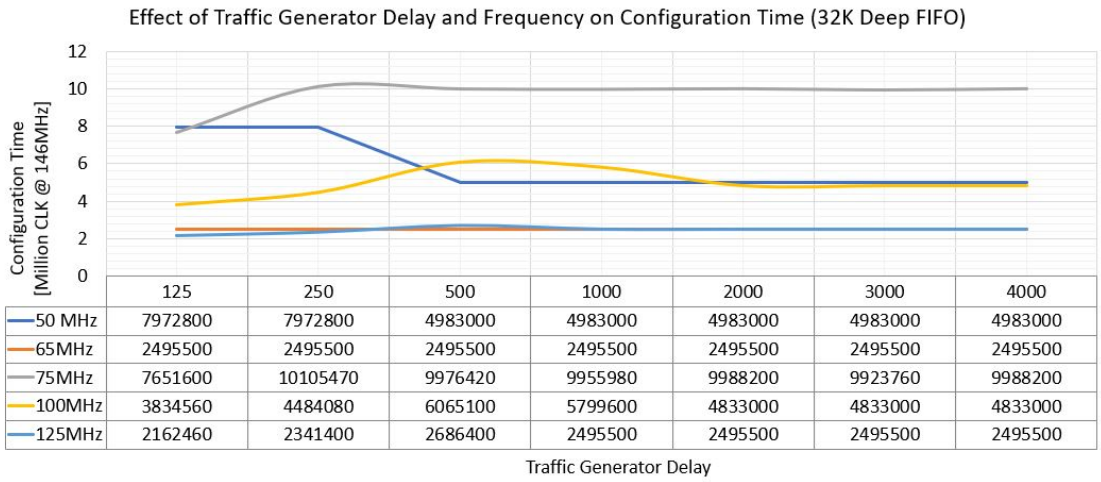
5.3.1 *Effect of traffic generator delay and frequency*



	125	250	500	1000	2000	3000	4000
50 MHz	30071526	35966043	47755302	71333837	118490916	165647993	212805054
65MHz	20816831	23077769	27599853	36644070	54732496	72820913	90909338
75MHz	23987814	31847433	47567467	79006751	141884797	204762837	267640882
100MHz	14613622	17889476	23748410	35466711	58902473	82338101	105773742
125MHz	10868243	12046452	14404361	19119369	28549703	37979939	47410185

Figure 89: Effect of Traffic Generator delay and frequency on Total Time

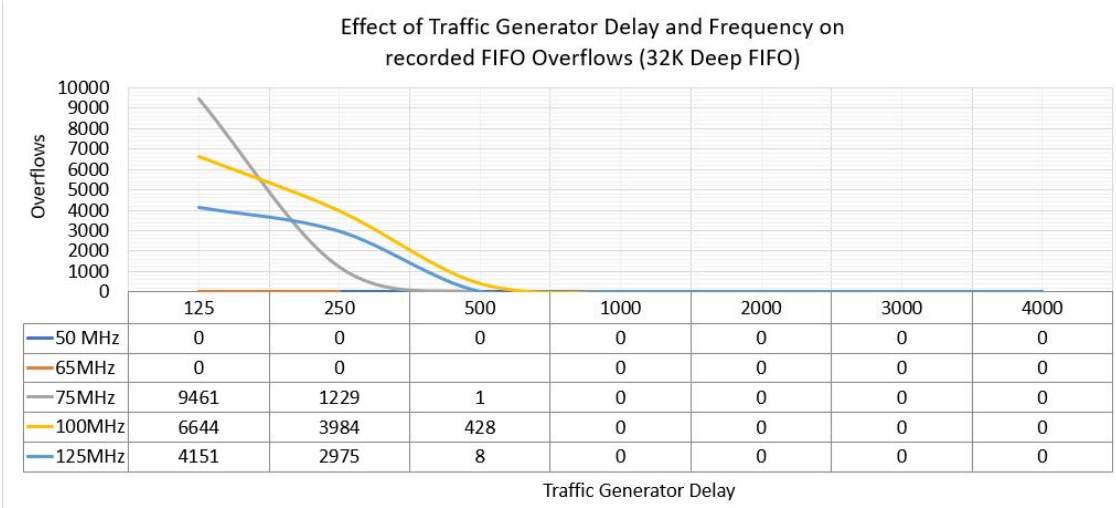
Figure 89 shows how the traffic generator delay affects the total time taken to transfer the 33MB of data. This simulates how the scenario responds to bursts of data. The different delays are shown as clocks in the frequency of the traffic generator. As expected, it can be seen how having more considerable delays increases the total time taken. If we look at equation 8, we can see how increasing the delay decreases the average throughput, meaning that more time will be needed to transfer all the data.



	125	250	500	1000	2000	3000	4000
50 MHz	7972800	7972800	4983000	4983000	4983000	4983000	4983000
65MHz	2495500	2495500	2495500	2495500	2495500	2495500	2495500
75MHz	7651600	10105470	9976420	9955980	9988200	9923760	9988200
100MHz	3834560	4484080	6065100	5799600	4833000	4833000	4833000
125MHz	2162460	2341400	2686400	2495500	2495500	2495500	2495500

Figure 90: Effect of Traffic Generator delay and frequency on Configuration Time Time

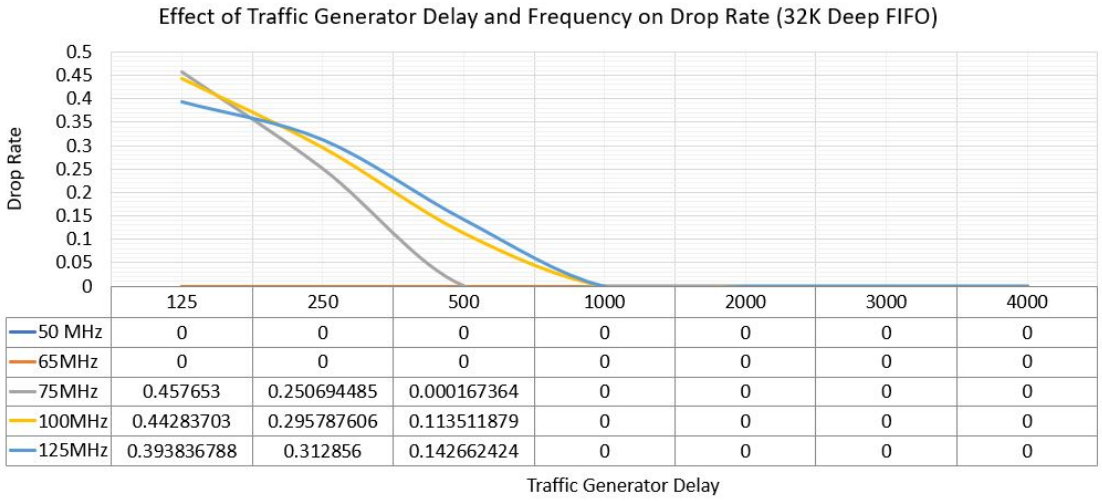
From figure 90 it can be seen how the configuration time is affected by the traffic generator frequency, this time is the total time the CPU spends configuring the DMA for transferring 33MB. As the Traffic Generator IPcore only has one clock input that is shared with the configuration interface. This results in the time taken to configure the traffic generator to vary in time. Furthermore, it can be seen that the configuration time at different frequencies does not increase linearly, with the information provided by the measured data, there is not enough information to explain why this happens. Nevertheless, we can see that the configuration time increases as the traffic generator delay increases. This may happen because, with higher delays, the DMA may have to make more transfers of a smaller volume, which will increase the total configuration time.



	125	250	500	1000	2000	3000	4000
50 MHz	0	0	0	0	0	0	0
65MHz	0	0		0	0	0	0
75MHz	9461	1229	1	0	0	0	0
100MHz	6644	3984	428	0	0	0	0
125MHz	4151	2975	8	0	0	0	0

Figure 91: Effect of Traffic Generator delay and frequency on the number of FIFO Overflows

Figure 91 shows a behaviour that was expected, increasing the delay parameter results in fewer FIFO overflows as the DMA has more time to empty the FIFO. Furthermore, it can be seen that for delays superior to 500 clocks, there are no data losses caused by the FIFO overflowing.



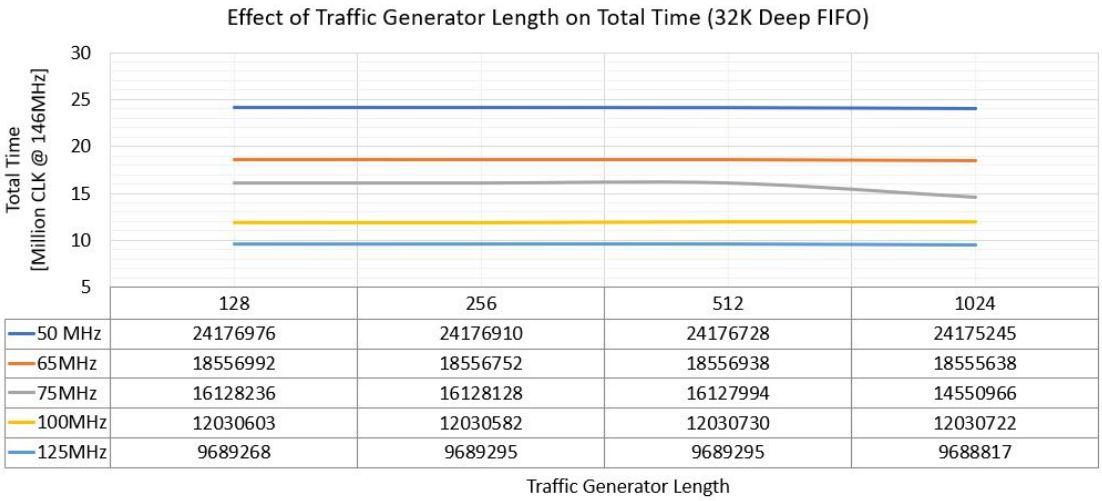
	125	250	500	1000	2000	3000	4000
50 MHz	0	0	0	0	0	0	0
65MHz	0	0	0	0	0	0	0
75MHz	0.457653	0.250694485	0.000167364	0	0	0	0
100MHz	0.44283703	0.295787606	0.113511879	0	0	0	0
125MHz	0.393836788	0.312856	0.142662424	0	0	0	0

Figure 92: Effect of Traffic Generator delay and frequency on Drop Rate

Figure 92 shows a similar pattern, with delays above 500 CLK, there are no bytes dropped, which makes us believe that all bytes dropped are caused to the FIFO overflowing. Moreover, it can be seen that at 65MHz, no bytes are being dropped even for the lowest delay. At 65MHz, with a delay of 125 CLK, the traffic generator is producing data at an average throughput of 252MBps, which is calculated using equation 8. We can not suppose this throughput is the maximum allowed by the system as it just represents an average throughput.

5.3.2 *Effect of traffic generator length and frequency*

Different traffic generator lengths will be testes to see the effect they have on the scenario. These tests simulate a constant flow of data, where the TLAST signal is asserted after a different number of bytes, forcing the DMA to do more or fewer transfers.

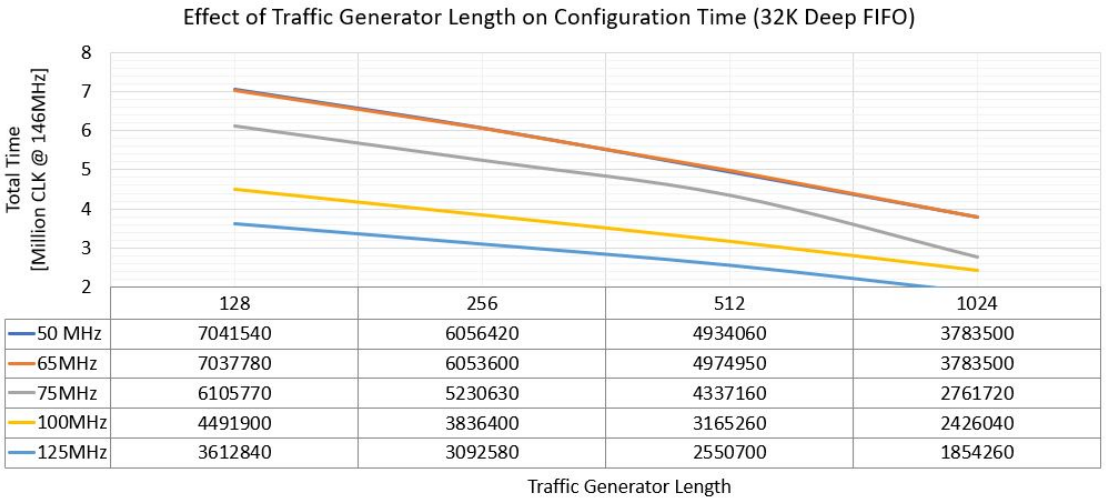


	128	256	512	1024
50 MHz	24176976	24176910	24176728	24175245
65MHz	18556992	18556752	18556938	18555638
75MHz	16128236	16128128	16127994	14550966
100MHz	12030603	12030582	12030730	12030722
125MHz	9689268	9689295	9689295	9688817

Figure 93: Effect of Traffic Generator length and frequency on Total Time

Through figure 93, it can be seen that increasing the frequency reduces the total time taken, as expected. Furthermore, we can see that less time is needed when using length

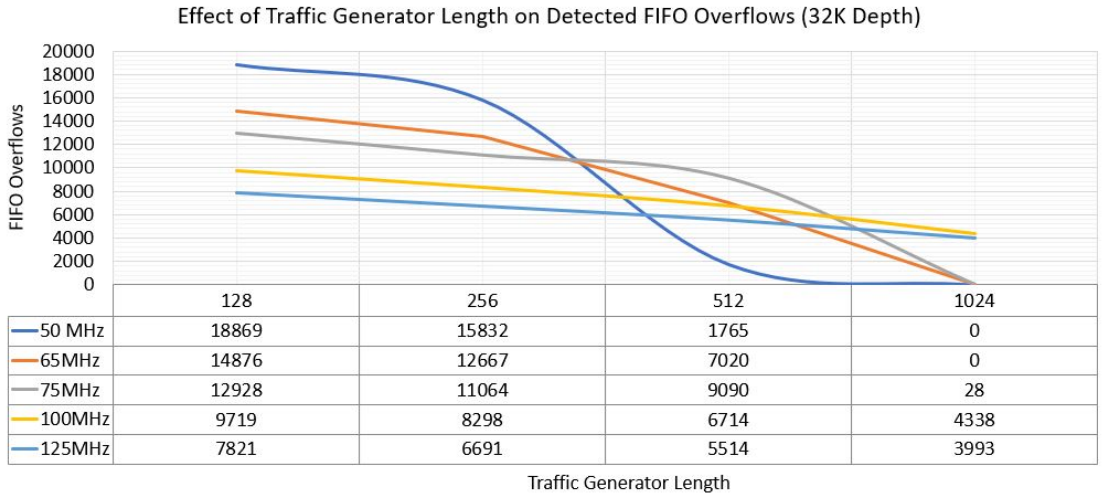
values closer to 1024, which make sense as we allow the DMA to perform more massive transfers at once.



	128	256	512	1024
50 MHz	7041540	6056420	4934060	3783500
65MHz	7037780	6053600	4974950	3783500
75MHz	6105770	5230630	4337160	2761720
100MHz	4491900	3836400	3165260	2426040
125MHz	3612840	3092580	2550700	1854260

Figure 94: Effect of Traffic Generator length and frequency on Configuration Time Time

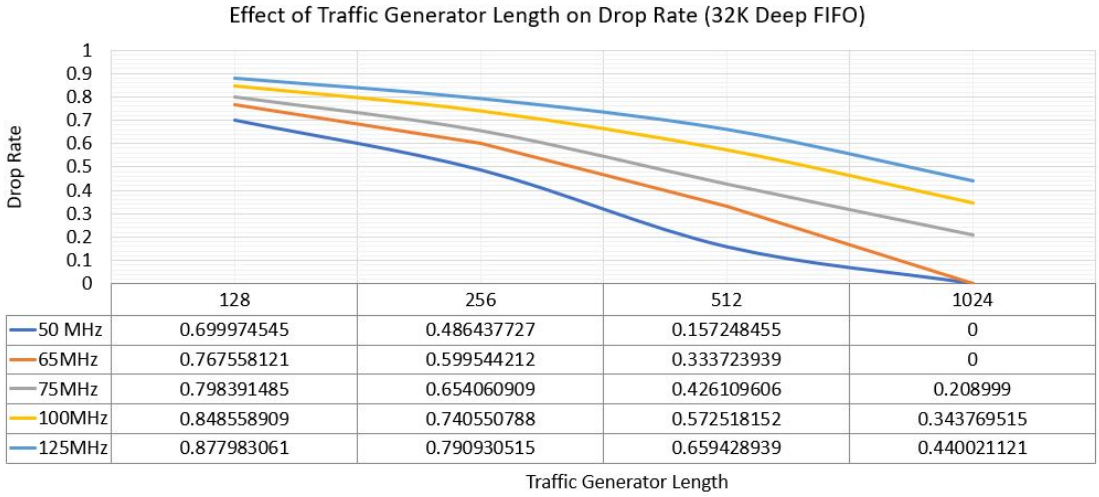
Figure 94 shows how the total configuration time decreases with time. This is also an expected result as with higher values for the traffic generator's length. Less DMA transfers have to be made, which in turn reduces the total configuration time as the DMA has to be configured fewer times.



	128	256	512	1024
50 MHz	18869	15832	1765	0
65MHz	14876	12667	7020	0
75MHz	12928	11064	9090	28
100MHz	9719	8298	6714	4338
125MHz	7821	6691	5514	3993

Figure 95: Effect of Traffic Generator length and frequency on the number of FIFO Overflows

From the graph in figure 95, it can be seen that increasing the length parameter reduces the amount of FIFO overflows. Also, it can be seen that for frequencies below 75MHz, the number of overflows reduces to zero when using a length of 1024.



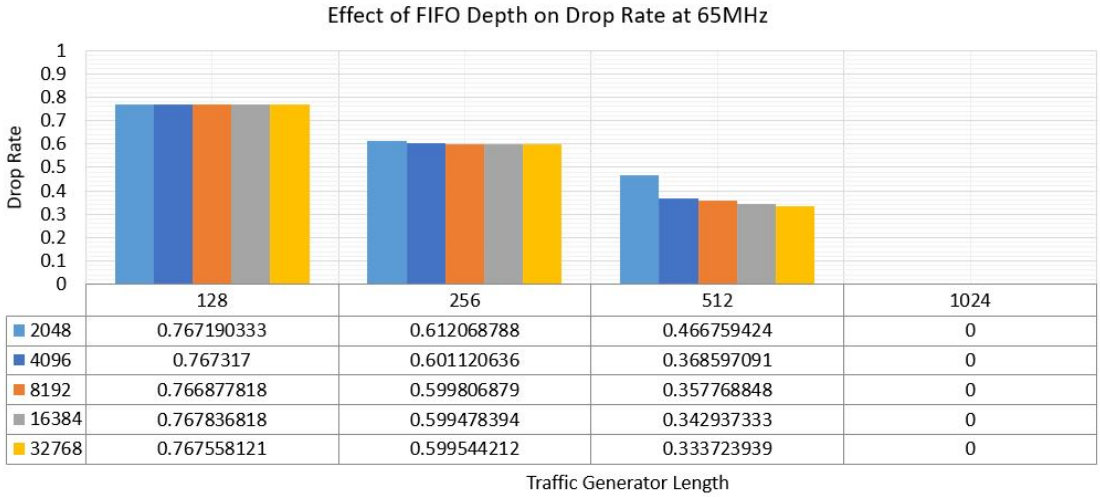
	128	256	512	1024
50 MHz	0.699974545	0.486437727	0.157248455	0
65MHz	0.767558121	0.599544212	0.333723939	0
75MHz	0.798391485	0.654060909	0.426109606	0.208999
100MHz	0.848558909	0.740550788	0.572518152	0.343769515
125MHz	0.877983061	0.790930515	0.659428939	0.440021121

Figure 96: Effect of Traffic Generator delay and frequency on Drop Rate

Figure 96 shows that increasing the length parameter also reduces the drop rate. Nevertheless, it can also be seen that for frequencies above 65MHz bytes are always lost. At 65MHz, we can see that when using a length of 1024 (asserting TLAST every 1024 4-byte words) we lose no bytes.

5.3.3 *Maximum Throughput*

As shown in figure 96, we can see that our design can handle a throughput of $65\text{MHz} \cdot 4\text{B} = 260\text{MBps}$. Nevertheless, this value has been acquired using a FIFO with a depth of 32K, so different values will be tested to see how the drop rate is affected.



	128	256	512	1024
2048	0.767190333	0.612068788	0.466759424	0
4096	0.767317	0.601120636	0.368597091	0
8192	0.766877818	0.599806879	0.357768848	0
16384	0.767836818	0.599478394	0.342937333	0
32768	0.767558121	0.599544212	0.333723939	0

Figure 97: Effect of FIFO Depth on drop rate at 65MHz

It can be seen from figure 97 that when using a value of 1024 for the traffic generator length, the FIFO Depth can be brought down to 2K, which is equal to a FIFO of $2\text{K} \cdot 4 = 8\text{KB}$. We can also see that for small lengths, a much higher FIFO would be needed to avoid the loss of bytes. This happens because, for small lengths, the DMA will perform more transfers of fewer bytes. If it were needed to use a small length

value, the solution would be increasing the DMA frequency or the FIFO depth.

5.3.4 *NVIS real-time throughput*

The throughput from PL-PS in the NVIS real-time mode was calculated to be 8MBps as calculated in equation 4. We can see how the implemented scenario has increased this by a very significant amount while also avoiding any data loss and reducing the size of the FIFO used. The small size of the FIFO means that if the TX channel were to be implemented too, there would be no problem implementing the design in the current Zynq device or the Red Pitaya's superior Zynq.

Due to the way the clocking has been generated in the PL, in order to simulate a constant data flow of 8MBps, the traffic generator should be configured to a frequency of 2MHz. Sadly, the clocking wizard and PL Fabric clocks do not allow frequencies below 4MHz. Nevertheless, a value of 8MHz was used to verify that this scenario could handle a slower throughput of data reliably. This frequency would simulate a throughput of $8\mathrm{MHz} \cdot 4 = 32\mathrm{MBps}$.

A transfer of 240MB was configured and the results were measured, a transfer of 480MB could not be configured due to the limitation in the transfer count register length of the traffic generator. The results were the following:

Traffic Generator Frequency [MHz]	8
Total Time [CLK @146MHz]	1094417806
Dropped Bytes	0
Detected Overflows	0
DMA Configurations	58590

Table 7: Results for a 240MB transfer with an 8MHz Traffic Generator Frequency

It can be seen that when transferring half the amount of data as in the NVIS project at a higher frequency results in no data being lost, which shows how the implemented scenario could be implemented for the NVIS.

6 CONCLUSIONS

The main goal of this project has been to increase the performance and reliability of the NVIS project by improving the Zynq's PL-PS communication and explaining different scenarios that could be used in many other projects. Moreover, the background knowledge given in section 2, the explanation of each scenario in section 3 and the analysis of the effects of different configuration parameters in section 5 will enable future students to become much more confident in the field of Zynq development, allowing them to tackle many different projects using Zynq platforms.

From the obtained results in the first scenario, it can be concluded that the use of the data cache will be beneficial for increasing the performance of the scenario. Furthermore, the use of a larger DMA burst size may improve the system's performance and reliability, although, from the obtained results, this increase in performance will not be significant. It can also be deduced that an increase in the PL frequency will not be directly proportional an increase in the system's performance as this will depend on the responses of the different intermediate blocks the required data has to pass through. Even though the relationship is not directly proportional it can be simplified to a linear relationship as the PL's limit frequency is reached before any significant deviation from a linear relationship is shown.

The second scenario allows us to analyse the use of a different data transfer block, the CDMA. This scenario shows that depending on the nature of the transfers being made, it may be more beneficial to enable or disable the data cache. Besides, no clear relationship has been found between the CDMA burst size and the system's performance. Finally, it has been seen that different transfer directions will respond differently to an increase in the PL frequency, this variation will not only be dependent on the Zynq Device used, it will also be dependent on any external memory used. This means that when using a CDMA, it is crucial to analyse the nature of the transfers that will be made before deciding the system's configuration.

The third and last scenario allowed to find the best configuration to increase the NVIS project's RX channel's Pl-PS performance. The conclusions found for the first scenario also apply to this one, as the third scenario can be seen as an expansion of the first scenario. Furthermore, it has been found that asserting the TLAST signal after a more significant number of bytes are transferred will increase both the performance and reliability of the system drastically. In addition, the way that this scenario has been analysed could be applied to many other scenarios and implementation, as the system's maximum operating frequency will not only depend on the configuration of each of the scenario's parameters but also on the Zynq used, the types of memories im-

plicated and how optimal the software is. This shows how complex it can be to study the performance of an SoC without any simulation accurately. Moreover, as the FIFO is working as an intermediate buffer, a larger FIFO will allow for a lower drop rate in most cases. Nevertheless, we must remember that the FIFO uses block memory, which usually is a scarce resource in most FPGAs. We have also seen that in this specific scenario, varying the size of the FIFO did not always result in a reduction of the drop rate. This was said to be due to a bottleneck in the HP0 interface of the Zynq, or the onboard SRAM, due to its chip or drivers.

This project has allowed me to gain experience in the field of both Embedded C programming for ARM Cores and hardware development on Xilinx FPGAs. Furthermore, having to work in the Vivado and Vitis environments have made me more comfortable learning to use new IDEs, which I find to be a precious skill.

Working with Zynq devices has been very challenging at some times, especially when debugging both hardware and software simultaneously. I also value a lot the fact that I had to find a new development board on the fly, as it forced me to gain knowledge on how to select different Zynq devices, which I believe is a skill that will help me in the future when I develop more projects on Zynq platforms.

Also, knowing that results were going to be extracted from the scenarios that were being implemented forces me to try and make my programming as efficient as possible, which I believe has improved my programming skills. Having good communication with my professors has also been a critical factor for developing this project successfully and efficiently.

From a research point of view, I have been able to see how vital planning things in advance is as sometimes you may not have time to keep on making changes due to bad planning. I believe this project had made me value even more the jobs that researchers due, as it not only requires technical skills but also writing, data collection, planning, critical thinking and critical analysis between many others.

Moreover, having to develop a project during the COVID-19 quarantine has required a lot of self-discipline and organisation to meet the required deadlines consistently. This is what I believe to be the most valuable and challenging skill I have worked on during the development of this project.

To conclude, I hope that the work developed in this project can help future students or engineers that, like me, started with not much of the background knowledge required to work with Zynq platforms and can find this project useful.

7 FUTURE LINES OF RESEARCH

Through the development of this project, many things that could allow future improvements have been found, but not developed due to time constraints.

Firstly, in order to analyse if some of the behaviours described in section 5 were caused by the memory driver in the Zynq's PS, it would be interesting to repeat the tests using a Zynq device that had the SRAM connected to the PL, requiring the use of a Memory Interface Generator (MIG) IPcore to interface the SRAM. Having a direct connection from the memory to the PL would significantly reduce the latency times necessary to move data from and to the SRAM through the AXI DMA which would probably increase the performance of the system, allowing for much higher throughputs to be reached.

Also, implementing both the TX and RX channels on one same design would be the next logical step for this project, analysing how the performance of the design changes when simultaneous reads and writes have to be performed by the AXI DMA. This would also be much more useful for the GRITS research group, only having to add to the design the different IPcores used for treating the data before sending or receiving it from the Raspberry PI.

Moreover, using the scatter-gather mode for other applications could be very interesting, and it would also teach me more things about how to use DMAs. This could be combined with having to use other DMA blocks, such as the MCDMA or the VDMA. I have developed much curiosity for this last block, so studying different uses of the VDMA could be very interesting for applications where a video input/output is needed. Furthermore, the use of video data requires much control of the data latencies so it would be an exciting challenge.

Finally, it would have been beneficial to use the onboard SD card to save the data acquired from the different tests. This could have been done to automate the data extraction process by, for example, writing the data in the SD card as a JSON object and then extracting such data through a Python script. I believe this would be an essential upgrade to the project as the data extraction resulted in much more tedious than it was anticipated and allows for less space for human error.

BIBLOGRAPHY

[1] Carles Vilella i Parra. "Comunicacions avancades d ' HF entre la Base Antartica Espanyola i l ' Observatori de l ' Ebre : caracteritzacio de canal i transmissio de dades Realitzada per : Carles Vilella i Parra." In: (2007).

[2] David Badia Joaquim Porté, Joan Lluis Pijoan, Josep Masó and Rosa Maria Alsina-Pagès Zaballos, Agustín. "Advanced HF Communications for Remote Sensors in Antarctica." In: *Intech* (2018).

[3] Marta Miret. "Digital Emergency Communications for the isolated High- Andean communities in Valle Sagrado." In: (2018).

[4] Joaquim Porte et al. *Education and e-health for developing countries using NVIS communications*. https://ieeexplore.ieee.org/document/8629842. (Accessed on 05/09/2020). 2018.

[5] Joaquim Porte. "[TFG]: Estudi canal comunicació NVIS Barcelona - Cambrils." In: (2016), p. 82.

[6] Joaquim Porte. "[TFM]: Desenvolupament d'una plataforma de baix cost per a sensors remots amb tecnologia NVIS." In: (2018), p. 90.

[7] *Red Pitaya home page*. Accessed: 08-05-2020. URL: https://www.redpitaya.com/.

[8] Guillem Garrofé Montoliu. "Receptor de comunicacions NVIS en temps real i disseny d'una xarxa de comunicacions per les comunitats remotes del Valle Sagrado." In: (2019).

[9] *Las redes más usadas en IoT (Sigfox, LoRa, NB-IoT, LTE Cat M)*. http://www.bcendon.com/las-redes-mas-usadas-en-el-iot/. (Accessed on 05/08/2020).

[10] *Zynq Architecture showing PS, PL and the interfaces*. https://www.researchgate.net/figure/Zynq-Architecture-showing-PS-PL-and-the-interfaces_fig2_303810600. (Accessed on 05/08/2020).

[11] Xilinx. "Zynq-7000 SoC Technical Reference Manual - UG585 (v1.12.2) July 1, 2018." In: *UG585* 585 (2018), pp. 1–1843.

[12] Xilinx. *Zynq UltraScale+ MPSoC*. https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html. (Accessed on 05/09/2020).

[13] *PetaLinux Tools Documentation: Reference Guide*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug1144-petalinux-tools-reference-guide.pdf. (Accessed on 05/08/2020).

[14] FreeRTOS. *FreeRTOS: Quality RTOS Embedded Software*. https://www.freertos.org/about-RTOS.html. (Accessed on 05/09/2020).

[15] Xilinx. *Zynq-7000 SoC*. https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html. (Accessed on 05/09/2020). 2016.

[16] Xilinx and Inc. *7 Series FPGAs Data Sheet: Overview (DS180)*. Tech. rep. 2018.

[17] *Speed Grade*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx10/isehelp/pta_p_hid_options_speed.htm. (Accessed on 05/09/2020).

[18] Xilinx Inc. "Zynq-7000 SoC Data Sheet: Overview." In: (2018).

[19] *Cora Z7 Reference Manual [Reference.Digilentinc]*. https://reference.digilentinc.com/reference/programmable-logic/cora-z7/reference-manual?_ga=2.245666500.1110325775.1588933889-1536654346.1580921996. (Accessed on 05/09/2020).

[20] *Cora Z7 Low Cost Zynq-7000 Development Board - Digilent - Xilinx*. https://store.digilentinc.com/cora-z7-zynq-7000-single-core-and-dual-core-options-for-arm-fpga-soc-development/. (Accessed on 05/09/2020).

[21] Parallella. *Parallella board home page*. https://www.parallella.org/. (Accessed on 05/09/2020).

[22] Digilent. *Digilent Zybo board*. https://reference.digilentinc.com/reference/programmable-logic/zybo/start. (Accessed on 05/09/2020).

[23] Xilinx. "UG1037, AXI Reference Guide, v4.0." In: 1037 (2017), p. 175. URL: https://www.xilinx.com/support/documentation/ip{_}documentation/axi{_}ref{_}guide/latest/ug1037-vivado-axi-reference-guide.pdf.

[24] Xilinx. "AXI DMA Reference Guide v7.1." In: *PG021* (2019), pp. 1–7. DOI: 10.1895/wormbook.1.67.2.

[25] Xilinx. "AXI Video Direct Table of Contents." In: *PG430* (2018), pp. 1–60.

[26] Xilinx. "AXI4-Stream Infrastructure IP Suite v3.0 LogiCORE IP Product Guide Vivado Design Suite." In: (2018). URL: www.xilinx.com.

[27] Xilinx and Inc. *AXI GPIO v2.0 LogiCORE IP Product Guide (PG144)*. Tech. rep. 2016.

[28] Xilinx. "AXI Traffic Generator Table of Contents." In: (2019).

[29] Xilinx Inc. *Block Memory Generator v8.4 LogiCORE IP Product Guide Vivado Design Suite*. Tech. rep. 2019.

[30] Xilinx and Inc. *AXI Block RAM (BRAM) Controller v4.1 LogiCORE IP Product Guide Vivado Design Suite*. Tech. rep. 2019.

[31] Xilinx Inc. "Vivado Design Suite User Guide." In: *Ug903* 4 (2019).

[32] Xilinx. "Vitis Unified Software Platform Documentation." In: *UG1400* 1393 (2019), pp. 1–407.

[33] Xilinx. "Zynq-7000 All Programmable SoC: Embedded Design Tutorial." In: *Ug 1165* 1165 (2015), pp. 1–110.

[34] Wojciech M. Zabołotny. "DMA implementations for FPGA-based data acquisition systems." In: *Photonics Applications in Astronomy, Communications, Industry, and High Energy Physics Experiments 2017* 10445 (2017), p. 1044548. ISSN: 1996756X. DOI: 10.1117/12.2280937.

[35] Apurva Choudhary, Jaimin B. Chavda, Amit P. Ganatra, and Rikin J. Nayak. "Performance evaluation PL330 DMA controller for bulk data transfer in Zynq SoC." In: *2016 IEEE International Conference on Recent Trends in Electronics, Information and Communication Technology, RTEICT 2016 - Proceedings* (2016), pp. 1811–1815. DOI: 10.1109/RTEICT.2016.7808147.

[36] M. Fularz, M. Kraft, and D. Pieczyński. "The performance comparison of the DMA subsystem of the Zynq SoC in bare metal and Linux applications." In: *Measurement Automation Monitoring* 63.5 (2017), pp. 189–191. ISSN: 2450-2855.

[37] Xilinx and Inc. *Zynq-7000 SoC and 7 Series Devices Memory Interface Solutions v4.2, User Guide (UG586)*. Tech. rep. 2018.

Part I

APPENDIX

a

DMA SCATTER GATHER MODE

When the scatter-gather mode is enabled in the DMA, the CPU does not have to configure the addresses accesses by the DMA. Instead, the DMA will read a series of descriptors from a specified region of memory through a specific scatter-gather AXI interface. These descriptors contain the information needed by the DMA to perform a transaction. Therefore, the CPU will only have to fill the memory (normally a small BRAM in the PL) with as many descriptors as needed.

This means that the CPU will only have to indicate the DMA to start the transfer, and the DMA will execute each descriptor one by one. It can be deduced that this greatly reduces the interaction needed with the DMA. Furthermore, if the DMA is not a MCDMA, the DMA can be configured to Cyclic Scatter gather mode. In this mode, when the DMA finished cycling through the descriptors, it will go back to the first descriptor, cycling through them continuously.

When in cyclic mode, a complete lack of feedback can be problematic as if the DMA halts, the CPU may not realise it. To resolve this, we can use interrupt coalescing in cyclic scatter-gather mode. Each time the DMA starts a transfer, an internal timer will be started, if the time surpasses a programmed threshold, the CPU will be interrupted so it can evaluate what has occurred and what to do.

b

SCENARIO 1 VIVADO VIEW

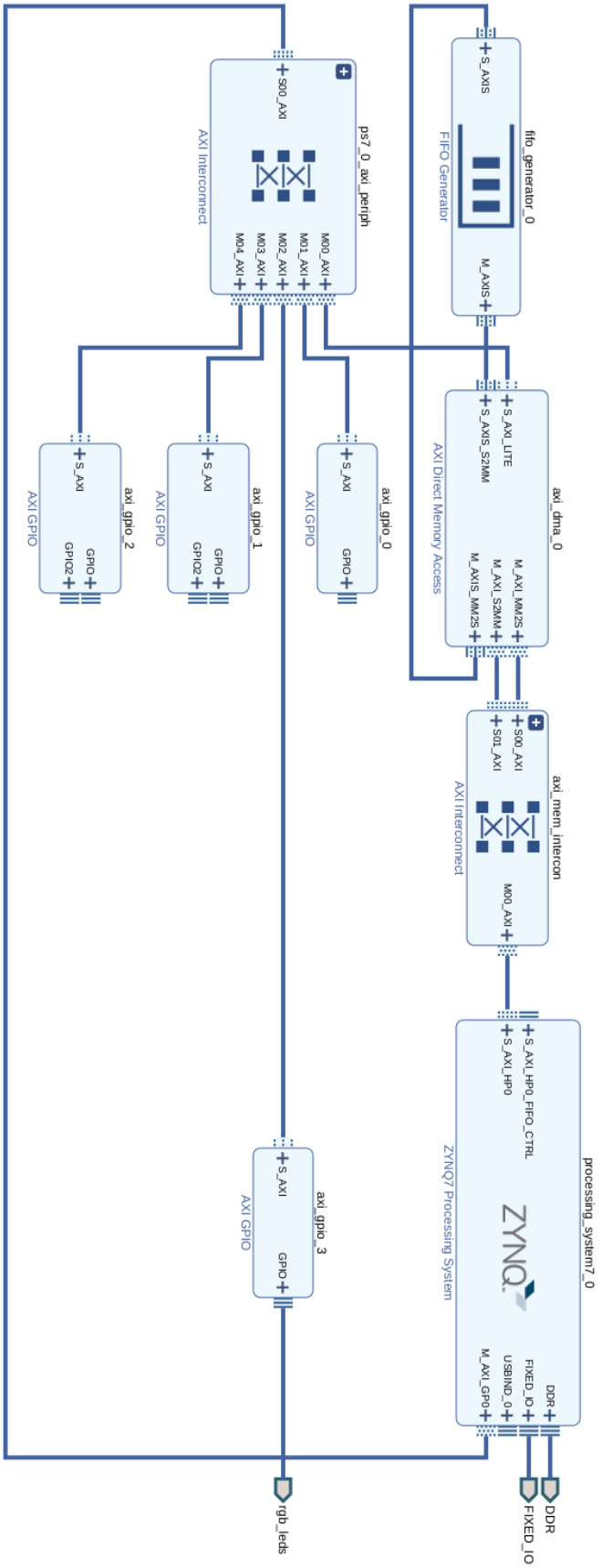


Figure 98: Vivado Interface view of Scenario 1

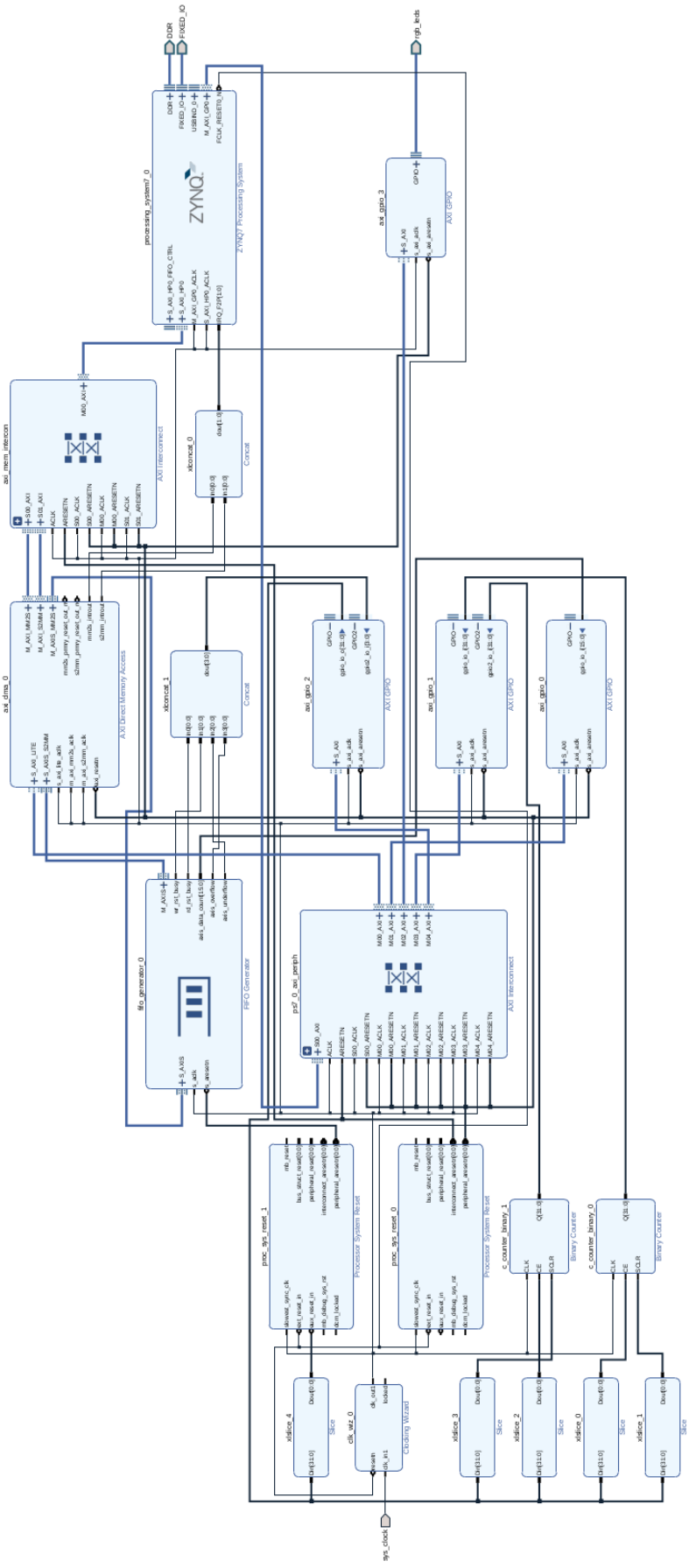


Figure 99: Scenario 1 implemented in Vivado

C

SCENARIO 2 VIVADO VIEW

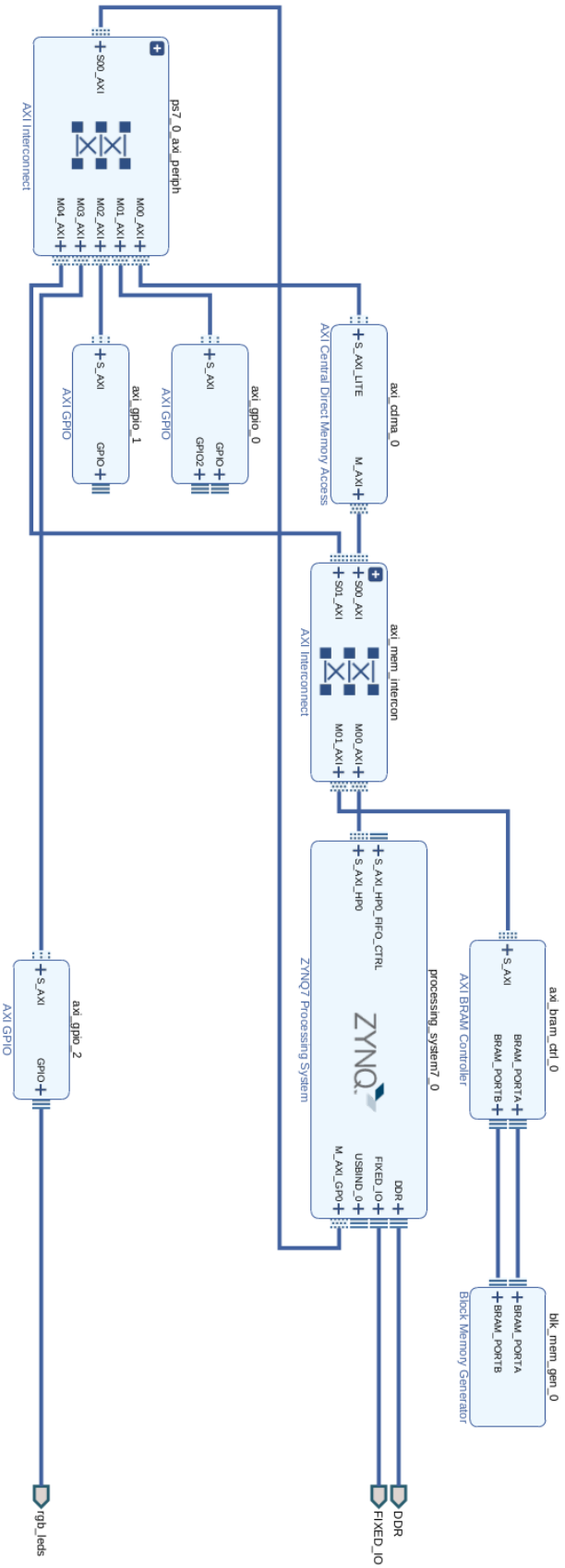


Figure 100: Vivado Interface view of Scenario 2

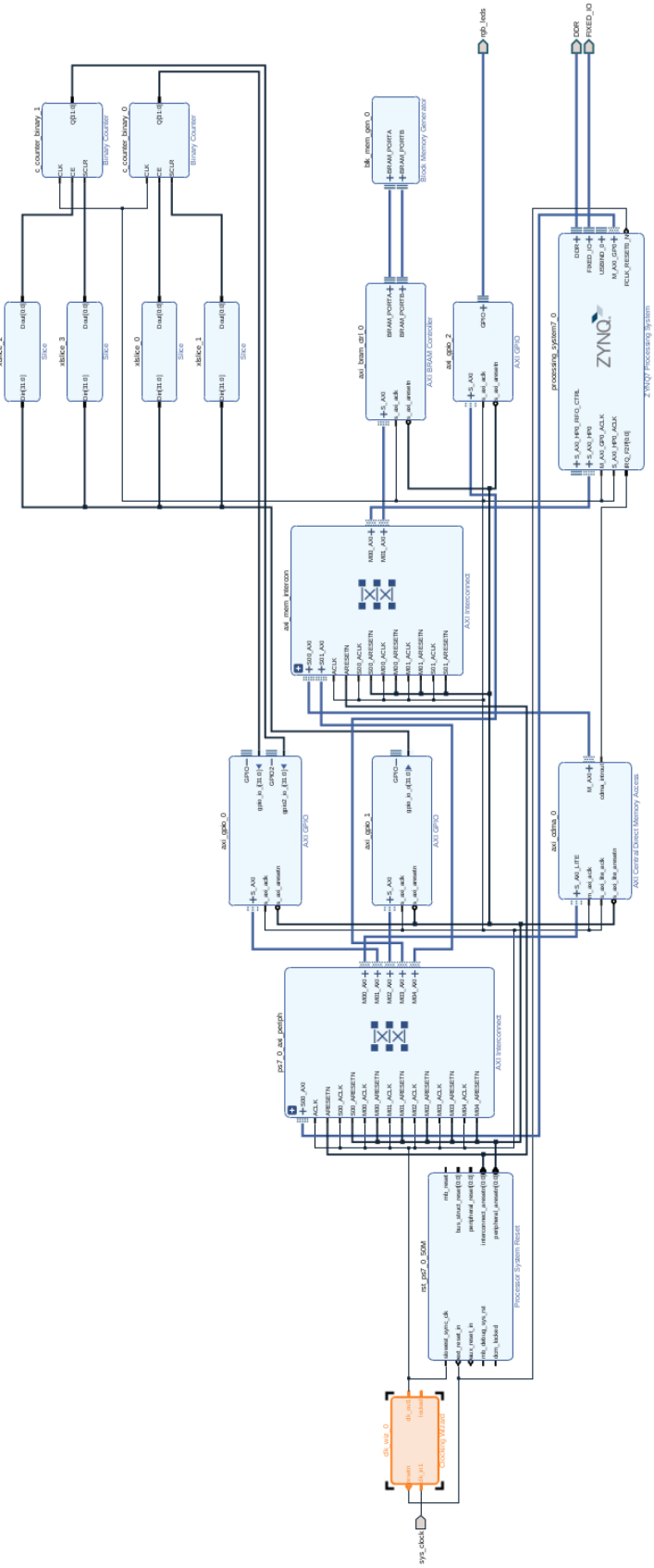


Figure 101: Scenario 2 implemented in Vivado

SCENARIO 3 VIVADO VIEW

d

Figure 102: Vivado Interface view of Scenario 3

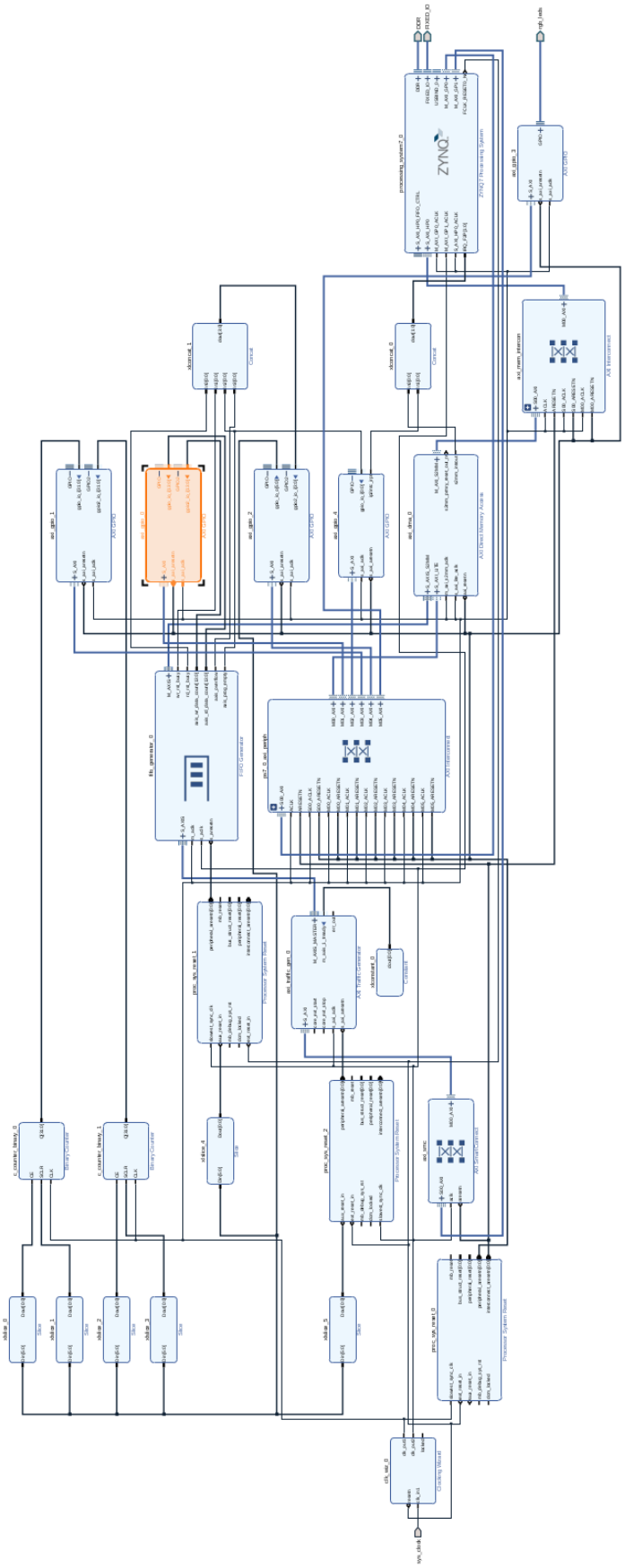


Figure 103: Scenario 3 implemented in Vivado

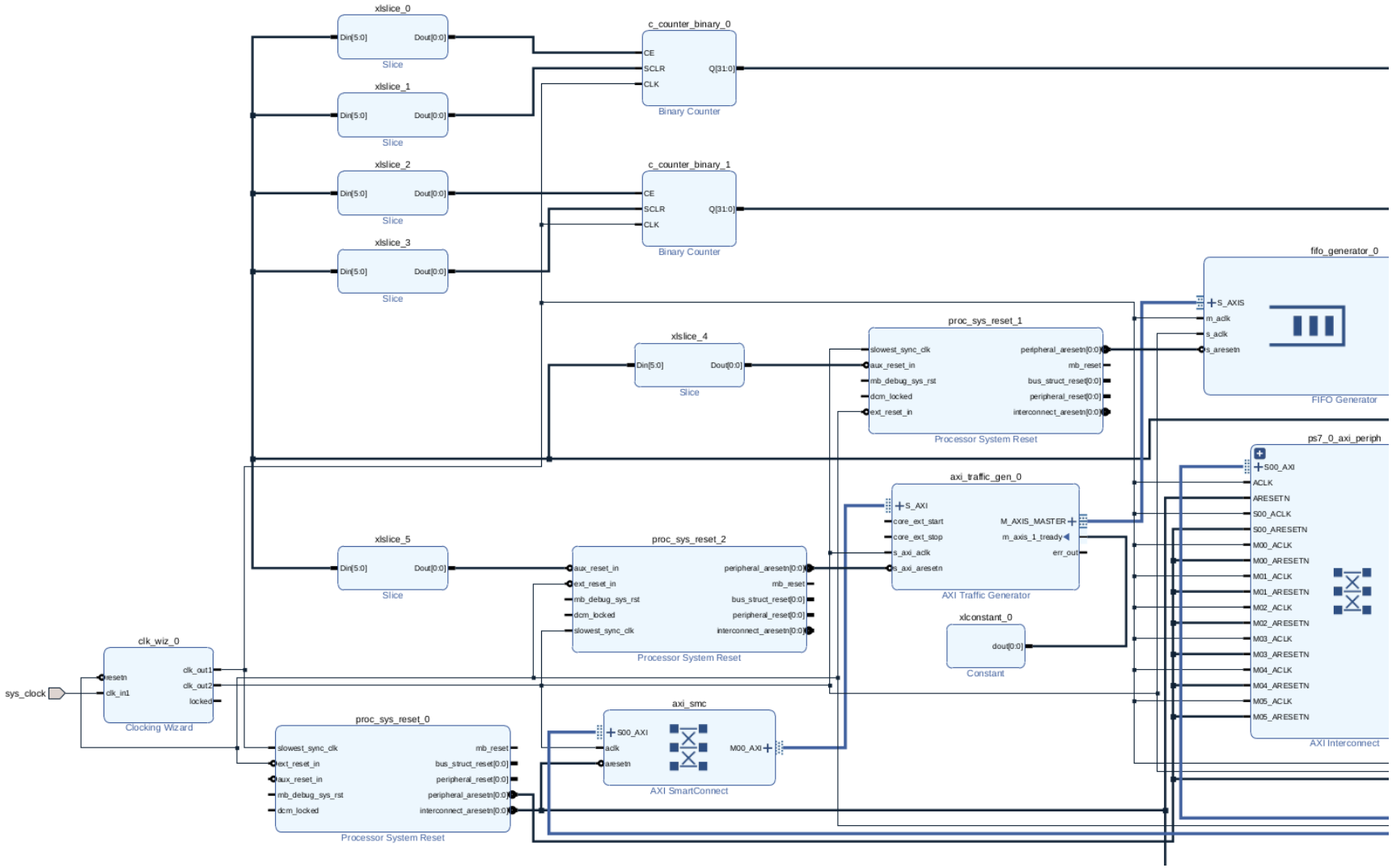


Figure 104: Scenario 3 implemented in Vivado (Part 1 of 2)

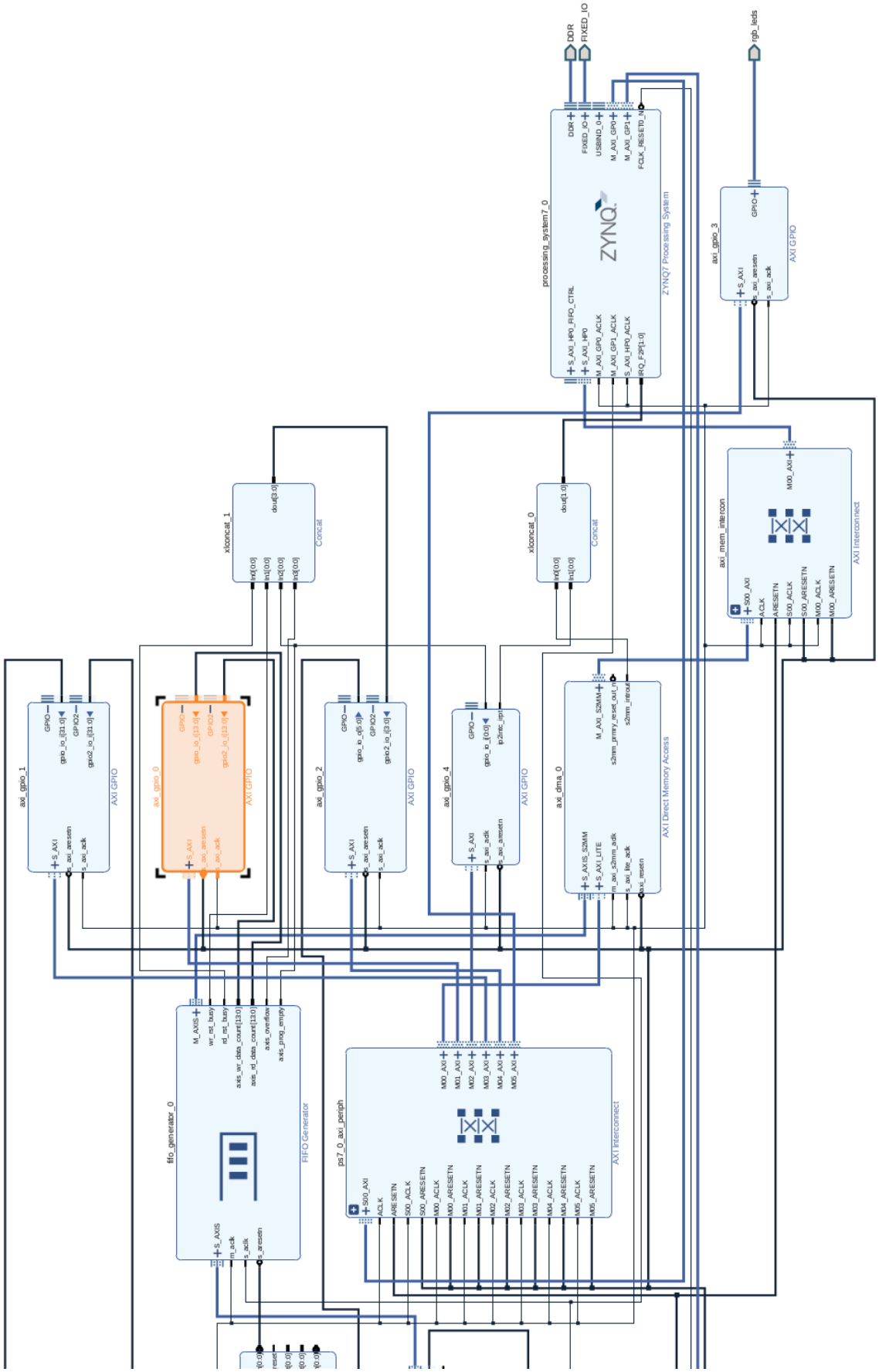


Figure 105: Scenario 3 implemented in Vivado (Part 2 of 2)

e

GITHUB REPOSITORY

In order to view the project files, you can access the GitHub repository for the project. Here the .tcl files for generating each scenario and the source code for the scenario's software will be found. Furthermore, a spreadsheet with all the extracted results can also be found there.