Development of an summing module with Vivado on RedPitaya board

Fabio A. Ciraci
Under the supervision of
Chris Stougthon

September 26, 2024

Abstract

This project focuses on the development of a summing module using Vivado on the RedPitaya board. The primary objective was to create a Python library that simplifies communication with the FPGA, enabling the summation of two arrays of values. This library is intended to support future projects in advancing the GQuEST experiment. Additionally, a GitHub repository was established to facilitate the learning process for individuals interested in mastering FPGA methods for array summation. The results demonstrate the effectiveness of the developed module and its potential applications in various computational tasks.

Contents

1 Introduction

1.1 Theoretical Background

Over the past century, one of the most significant challenges in theoretical physics has been the development of a quantum description of gravity [1]. Physicists have traditionally understood gravity as the curvature of spacetime, a concept introduced by Einstein's General Theory of Relativity. This curvature is caused by the presence of energy and momentum. On the other hand, quantum mechanics provides a framework for understanding the behavior of particles at the most fundamental level, describing phenomena with remarkable precision.

However, integrating these two powerful formalisms has proven to be exceptionally challenging. Several structural incompatibilities arise when attempting to formulate gravity within a quantum mechanical framework:

- **Non-linearity of Gravity**: Unlike the linear equations governing quantum mechanics, the equations of General Relativity are highly non-linear. This non-linearity complicates the superposition principle, a cornerstone of quantum theory.
- **Background Independence**: General Relativity describes gravity as a dynamic entity that shapes the spacetime fabric itself, making it background-independent. In contrast, quantum mechanics typically operates within a fixed spacetime background, leading to a fundamental discord between the two theories.
- **Scale Discrepancies**: Gravity is significant at macroscopic scales, while quantum effects dominate at microscopic scales. This vast difference in the energy and length scales at which these phenomena are observable poses a significant challenge in modeling gravity's local behavior within a quantum framework.

In recent years, one promising development in this field is the concept of **holography**. Originating from String Theory, the holographic principle suggests that the description of a volume of space can be thought of as encoded on a lower-dimensional boundary to the region. This principle highlights a discrepancy in the degrees of freedom between gravitational models and quantum mechanical models.

Building on these insights, physicists like Verlinde and Zurek have proposed that by applying the holographic principle locally, it is possible to detect fluctuations in longitudinal distances. These fluctuations are hypothesized to arise from variations in vacuum energy, which are influenced by holographic degrees of freedom.

1.2 GQuEST

The GQuEST (Gravity from Quantum Entanglement of Space-Time) collaboration aims to verify this holographic conjecture. This initiative seeks to address the challenge of detecting quantum gravity by exploring the entanglement properties of spacetime itself. By investigating these quantum entanglements, GQuEST hopes to uncover measurable effects that could provide evidence for a quantum theory of gravity.
The core components are the **Michelson-Morley Interferometer**, a well-established tool in experimental physics, renowned for its ability to detect minute changes in distance through the interference of light waves. This makes it ideal for observing the predicted metric fluctuations caused by quantum gravitational effects.

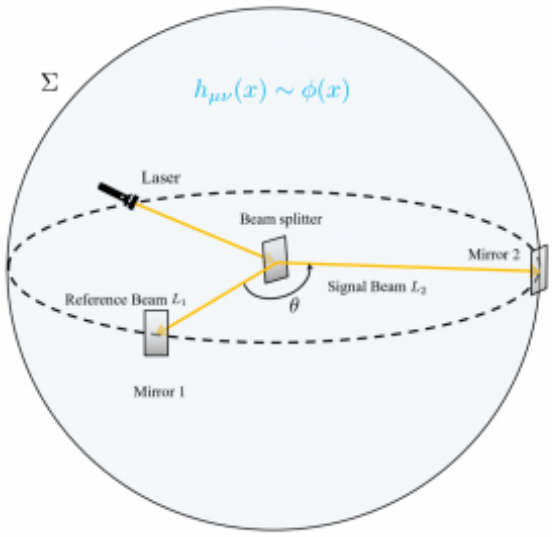


Figure 1: The GQuEST experiment

The configuration includes two co-located interferometers. This dual configuration is crucial for distinguishing between genuine signals correlated with metric fluctuations and uncorrelated noise, which could otherwise obscure the results. When gravitational waves or metric fluctuations pass through the interferometers, they induce slight changes in the lengths of the interferometer arms. These changes cause shifts in the interference pattern of the light beams, which can be precisely measured. The readout is perfomed with Homodyne tecnique.
Homodyne readout is a technique used to measure the phase shift of light waves with high precision. In the context of the GQuEST experiment, it helps in detecting the minute phase changes induced by metric fluctuations.
This method is highly sensitive and can provide continuous real-time data on the phase shifts occurring within the interferometers.

2 FPGA and Vivado Software

In this section we will explain in details what are the FPGA and why are the most suitable approach for this type of problems. In particular we'll explore the most important charasteristics of the FPGA boards in *section* 2.1.Finally in *section* 2.2, we'll describe charateristics of VIVADO, a software used for create designs suitable for FPGAs.

2.1 Field Programmable Gate Arrays (FPGAs)

Field Programmable Gate Arrays (FPGAs) are integrated circuits designed to be configured by the user after manufacturing. Unlike traditional fixed-function devices like Application-Specific Integrated Circuits (ASICs), FPGAs offer the flexibility of hardware reconfiguration, allowing designers to tailor the chip for specific applications. This

adaptability makes FPGAs an ideal solution for tasks that require high performance, parallel processing, and low-latency responses, such as signal processing, real-time control, cryptographic functions, and scientific computing.

2.1.1 Basic Structure and Operation of FPGAs

At the core of an FPGA are programmable logic blocks, interconnects, and I/O (Input/Output) blocks. These components can be configured to perform complex combinatorial and sequential logic operations. The FPGA fabric consists of:

- **Logic Blocks:** These are the fundamental building blocks of FPGAs, typically composed of Look-Up Tables (LUTs), flip-flops, and multiplexers. LUTs allow the FPGA to implement any logical function by storing truth tables, while flip-flops enable sequential logic by storing state information.
- **Interconnects:** Programmable interconnects allow different logic blocks to communicate. This highly customizable routing fabric enables the FPGA to form complex circuits by linking logic blocks together.
- **I/O Blocks:** These blocks interface with external components, such as sensors, memory, or communication systems, and support various I/O standards to accommodate different signal voltages and speeds.

An FPGA's configuration is stored in SRAM (Static Random-Access Memory) or flash memory, which means it can be reprogrammed multiple times. This allows engineers to iterate on their designs rapidly, making FPGAs particularly useful for prototyping and applications that need regular updates or adjustments.

2.1.2 Parallelism and Flexibility of FPGAs

One of the defining features of FPGAs is their ability to execute multiple operations in parallel. Traditional CPUs and GPUs process instructions sequentially, albeit with some degree of parallelism (e.g., through multi-core processors or SIMD instructions). In contrast, FPGAs offer true hardware-level parallelism, where different parts of the FPGA fabric can execute independent or coordinated tasks simultaneously.

For example, in an image processing application, different sections of an image can be processed in parallel on the FPGA, dramatically reducing the overall computation time. Similarly, in communication systems, multiple signal streams can be processed concurrently, making FPGAs ideal for high-throughput tasks.

This parallelism is especially beneficial in real-time systems where processing speed and responsiveness are critical. By leveraging the reconfigurable nature of FPGAs, designers can create custom data paths that precisely match the application's computational needs, ensuring minimal latency and maximum throughput.

2.1.3 Hardware Resource Utilization

FPGAs allow designers to maximize performance by leveraging the vast array of configurable hardware resources. The efficient use of these resources can significantly improve the performance of cross-correlation and other signal processing tasks. A well-optimized design involves striking a balance between computational units, memory resources, and communication paths. Below are key aspects of resource utilization on FPGAs:

- **DSP Slices:** DSP (Digital Signal Processing) slices are specialized hardware blocks designed for arithmetic operations like multiplication and accumulation. In cross-correlation, a large number of multiplications are required for each time-lag. FPGAs can instantiate multiple DSP slices to compute these operations in parallel, allowing for high-speed, low-latency computation. Modern FPGAs, such as those from Xilinx and Intel, contain hundreds or even thousands of DSP slices, enabling massively parallel processing of signal data. By using these DSP slices efficiently, the FPGA can compute complex operations like multiply-accumulate (MAC) in fewer clock cycles. Furthermore, many FPGAs support floating-point operations or fixed-point arithmetic with high precision, depending on the application requirements.

- **Block RAM (BRAM):** Block RAM is a critical resource for buffering and storing input signals, intermediate results, and final cross-correlation outputs. FPGAs provide a distributed block RAM architecture that can be configured to support various memory sizes and access patterns. For cross-correlation, signals often need to be stored for multiple time shifts, requiring efficient memory management. BRAM allows for fast, concurrent access to data, which is vital for maintaining high throughput in parallel processing environments. Designers can also use dual-port BRAM to allow simultaneous read and write operations, further enhancing performance. Efficient memory organization and pipelining techniques can ensure that memory bandwidth does not become a bottleneck during computation.

- **Distributed Memory and LUTs:** In addition to block RAM, FPGAs also feature distributed memory implemented using Look-Up Tables (LUTs). These can be used for smaller, low-latency storage needs, such as storing coefficients or state information. In cross-correlation, LUTs may be used to hold precomputed values or intermediate sums, allowing the FPGA to access this data with minimal delay. While distributed memory offers less storage capacity than BRAM, its flexibility and proximity to computational elements make it suitable for fast, small-scale data operations.

- **Parallelism:** One of the greatest strengths of FPGAs is their ability to implement massive parallelism. Cross-correlation involves repeated calculations across many time-lags, which can be computed simultaneously using parallel hardware architectures. Each lag can be processed in a separate hardware pipeline, ensuring that multiple correlations are computed in parallel. By taking full advantage of the FPGA's parallelism, designers can achieve significant performance gains, enabling real-time processing even for high-bandwidth signals. In contrast to traditional processors, which are bound by sequential instruction execution, FPGAs allow for the simultaneous execution of hundreds or thousands of operations, depending on the available logic resources.

- **Pipelining:** Pipelining is a key technique in FPGA design that allows multiple stages of a computation to overlap in time. By dividing the cross-correlation computation into several stages, each stage can process different parts of the data concurrently. For example, one stage might handle the multiplication, another the accumulation, and a final stage could normalize the results. This reduces the overall computation time and ensures that the FPGA can handle high-throughput data streams with minimal latency. Well-designed pipelines also allow for optimal use of

resources, as different operations can be executed in parallel across multiple pipeline stages.

- **Latency and Throughput Optimization:** FPGAs offer flexibility in controlling both latency and throughput. For applications where real-time performance is critical, minimizing latency is essential. This can be achieved through careful pipeline design and minimizing memory access delays. On the other hand, high-throughput applications benefit from optimizing parallel data paths and using as many processing elements as possible in parallel. Balancing these two factors depends on the specific needs of the application. For example, a radar system might prioritize low latency to ensure timely detection, while a communication system might prioritize high throughput to process large volumes of data efficiently.
- **Clock Management and Power Efficiency:** FPGA designs must carefully manage clock signals to ensure that all components operate efficiently and within timing constraints. FPGAs feature multiple clock domains, and designs can benefit from clock-gating and dynamic clock scaling to reduce power consumption in areas where high-speed operation is not required. Power efficiency is a critical concern in embedded systems, and FPGA designs can be optimized to minimize power usage without sacrificing performance by shutting down unused blocks or lowering clock speeds in certain regions of the design.

By intelligently utilizing the available hardware resources, FPGAs can achieve significant gains in both performance and power efficiency. Careful attention to resource allocation, parallelism, and memory management allows for high-speed computation that is essential for real-time cross-correlation applications. Furthermore, the flexibility of FPGAs allows designers to adjust and optimize the design to meet the specific needs of each application, whether it be for high-performance scientific instruments, communication systems, or real-time signal processing.

2.2 Introduction to Vivado Module Development

Vivado is a comprehensive FPGA design suite from Xilinx, widely used for designing and implementing digital logic systems. It provides a complete environment for FPGA design, from hardware description language (HDL) coding and simulation to synthesis, implementation, and bitstream generation for Xilinx FPGAs. Vivado supports both Verilog and VHDL, the two most common hardware description languages, allowing developers to write custom modules to meet specific design requirements.

Vivado module development involves several key stages, including design entry, simulation, synthesis, implementation, and verification. These steps are critical to ensure that the developed hardware functions as expected within the constraints of timing, power, and resource usage.

VIVADO supports many different types of FPGA boards, the one used during this internship is called RedPitaya 125-14 and which needs some custom modules to be adapted for the VIVADO environment, in this work these are implemented by [2].

2.2.1 Design Entry

The process begins with writing the HDL code, which describes the hardware's functionality at the register-transfer level (RTL). In Vivado, developers can create new modules

from scratch or reuse existing modules from the Xilinx IP library. Each module typically defines a specific hardware function, such as arithmetic units, state machines, or memory controllers. Modules can be connected to form more complex systems, and Vivado's graphical interface allows for easy management of these connections.

2.2.2 Synthesis and Implementation

Once the design is verified in simulation, Vivado translates the HDL description into a netlist of logic gates through a process called synthesis. This process converts the high-level behavioral description into a gate-level implementation suitable for FPGA hardware. Following synthesis, the design is implemented by mapping the logic onto the physical resources of the FPGA, such as lookup tables (LUTs), flip-flops, DSP slices, and block RAMs. Vivado's implementation tools optimize the design for performance and resource utilization, ensuring that the module meets timing constraints.

2.2.3 Timing and Resource Constraints

An essential aspect of module development in Vivado is the management of timing and resource constraints. Timing constraints define the required performance metrics, such as clock frequency and signal propagation delays. These constraints are critical for ensuring that the module operates reliably at the target clock speeds. Similarly, resource constraints ensure that the design fits within the available hardware resources on the FPGA. Vivado provides tools for constraint management and optimization, enabling developers to balance performance, power, and resource usage. In our case the module for time and clock managment was provided by Pau Gómez [2].

2.2.4 Bitstream Generation and Programming

After the synthesis and implementation phases, Vivado generates a bitstream file, which can be loaded onto the FPGA to configure its hardware. This bitstream contains the binary representation of the entire design, including the custom modules. Once the bitstream is programmed into the FPGA, the module becomes part of the FPGA's hardware logic, ready to interact with other system components in real-time.

Vivado module development is a powerful and flexible process, enabling designers to create custom digital hardware tailored to specific applications. From system-on-chip (SoC) designs to high-performance computing, Vivado provides the necessary tools for developing efficient, scalable, and high-performance FPGA modules.

3 Internship Aim

The primary aim of this internship was to gain hands-on experience in FPGA development, focusing on the design and implementation of custom hardware modules using the Vivado Design Suite. This internship has provided me with a descrete understanding of the FPGA design flow, from hardware description language (HDL) coding to the synthesis, implementation, and verification of complex digital systems.

A key object of this internship was the understanding of complex problems in hardware design and their solution, focusing on the module choice and understanding of the VHDL and Verilog languages.

3.1 Structure of the work

The internship work started from a simple blinking led project to understand the logic and hardware structure of the RedPitaya and then focused on a deeper understanding of the hardware.

The main steps were:

(a.) Blinking Led Project: Understanding how to access hardware resources

(c.) DMA Access project: Accessing the FPGA memory using the the DMA method

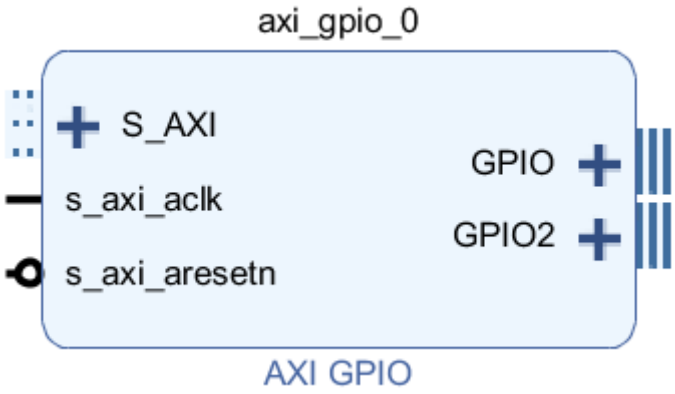
(b.) Sum in Arrays: Performing sums between arrays.

But before going in detail on the projects an explanation of what are the most important and recurring blocks in FPGAs designs is provided in order to easely present the further results.

4 Vivado IP blocks

In this section main Vivado IP blocks, useful for the subsequent analysis and project, are introduced and then described in detail. Reflecting this form, this section is divided into three subsection referred to different blocks.

Axi Gpio The AXI GPIO provides a general purpose input/output interface to the AXI (Advanced eXtensible Interface) interface. This 32-bit soft IP core is designed to interface with the AXI4-Lite interface.



This block is fundamental the interact with external input/output source, it has two channels and it is not possible to add more of them.

Direct Memory Access Direct memory access (DMA) is a feature of computer systems that allows certain hardware subsystems to access main system memory independently of the central processing unit (CPU)[3].

The DMA is a crucial block to store data inside the FPGA memory, the functioning is similar to portal between the external environment and the memory.

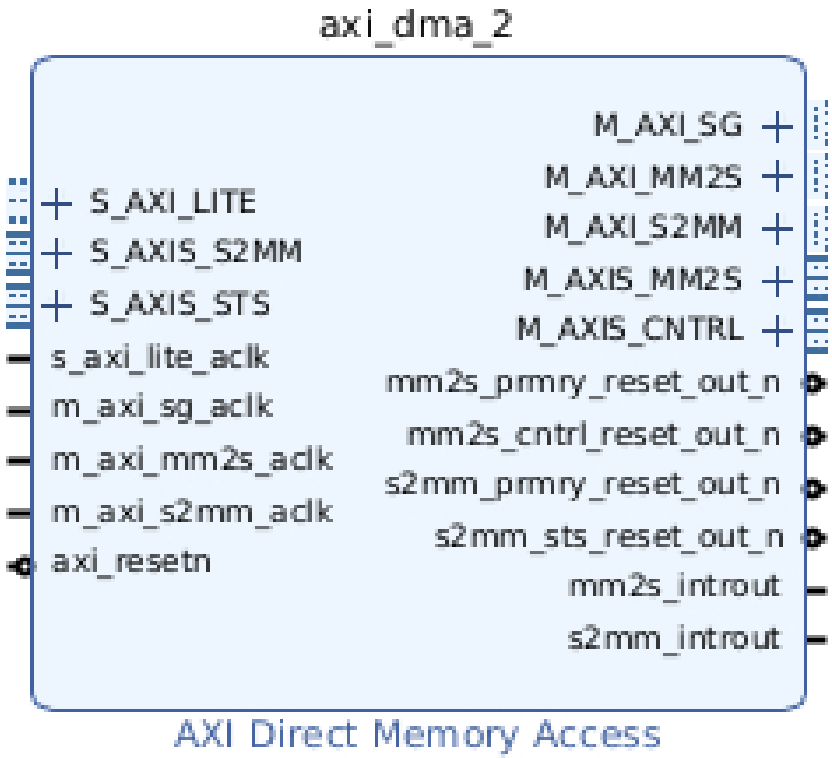


Figure 2: AXI DMA

In the DMA is possible to decide whether it is necessary for reading, writing or both by uncheking/checking the correspondent toggle:

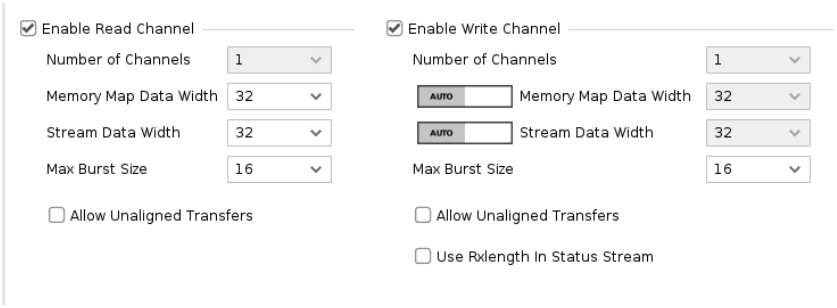


Figure 3: AXI DMA toggles

Processing System AMD provides the Processing System IP Wrapper for the Zynq™ 7000 to accelerate the design and its configuration for the embedded products.

The Processing System IP is the software interface around the Zynq 7000 Processing System. The Zynq 7000 family consists of a system-on-chip (SoC) style integrated processing system (PS) and a Programmable Logic (PL) unit, providing an extensible and flexible SoC solution on a single die.

The Processing System IP Wrapper acts as a logic connection between the PS and the PL while assisting in integrating custom and embedded IPs with the processing system using the Vivado™ IP integrator [4]

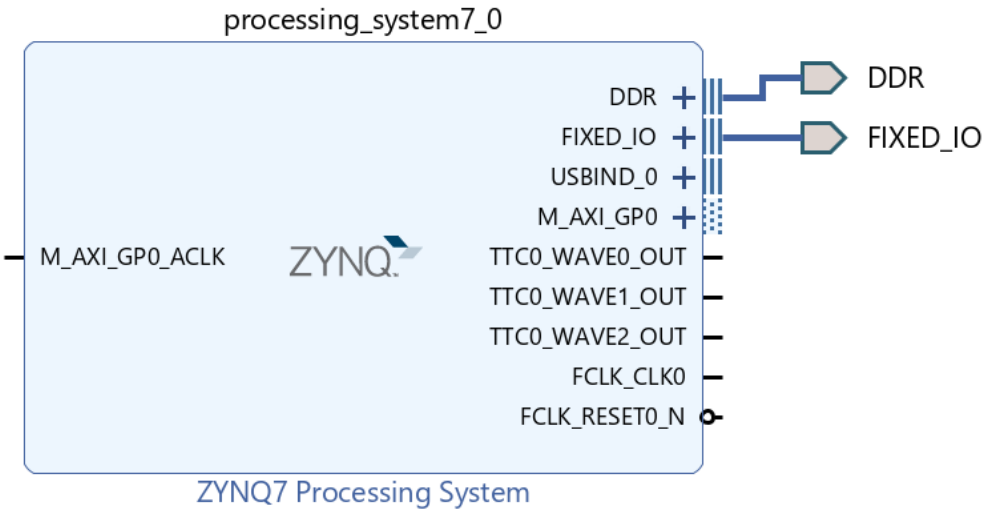


Figure 4: Zynq Processing System

5 Projects

The following section is divided into four main subsection reflecting the structure of the work.

5.1 Blincking Led Project

The first project focuses on accessing the RedPitaya hardware by creating a design that makes the LED blink. This project's purpouse is to access the basic functionality of the FPGA througth the VIVADO design.

During the project creation it is necessary to add the .xdc file to configure the hardware logic of the RedPitaya, this is a crucial step also for the next projects. [2]

The needed blocks are Zynq 7 Processing System and AXI GPIO along with the Concat, Slice and Interconnect.

The first step is to configure the DDR and Fixed IO on the processing system and it can be completed by pressing on the "*Run Block Automation*" tab. To create the required clock, reset and AXI Connect infrastructure it is possible to run the connection automation by pressing on the "*Run Connection Automation Tab*". Only the AXI GPIO needs to be configured manually so it not to be checked in the menu.

After the connection automation the design becomes like this:

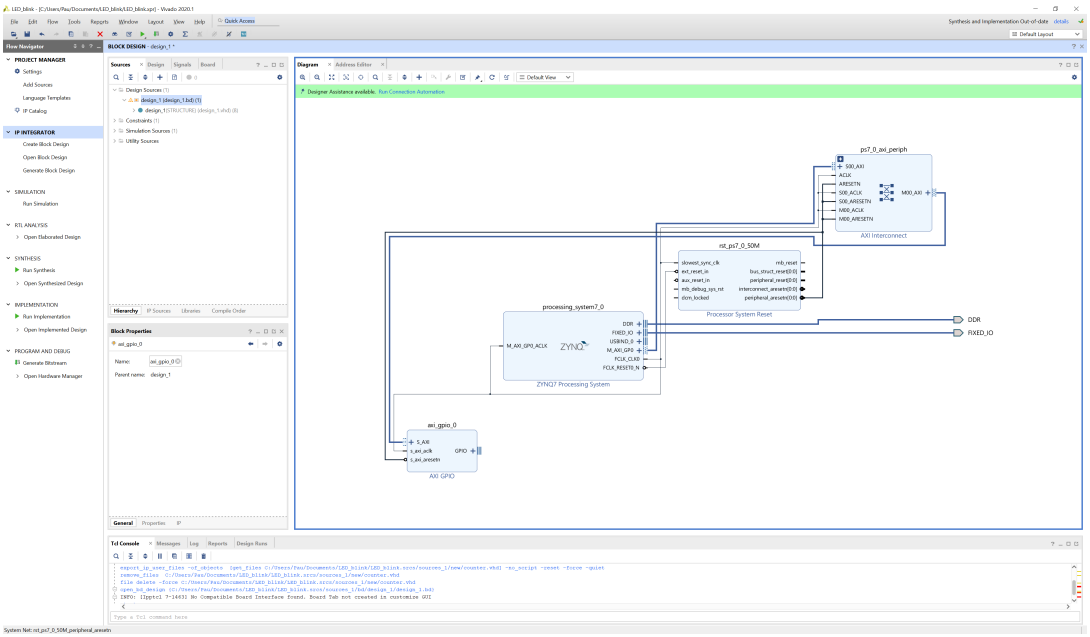


Figure 5: New Design

On the AXI GPIO instance the channel needs to be configured as outputs of 1 and 32 bits, the channel1 is used as a trigger and the channel2 to increment the counter value.

The Zynq instance needs the PL fabric clock to be configured to 50 MHz and usually this parameter is set as default if not it is possible to configure it under *Clock Configuration and then PL Fabric Clocks.*

The next step is to create an HDL counter. It is possible to create a new RTL(*Register Transfer Level*) module by using a file*Add Source -¿ Add or create design source, then naming it and choose the language, the code is available here 7.1.*

To add the module to the design rigth-clik on the design and select *Add Module.* In the constraints file it is necessary to uncomment the lines from 166 to 177 which define and configure the FPGA ports connected to the LEDs.

The information are sended to the LED using a port (it is possible to create a port by pressing simultaneusly *CTRL and k*):

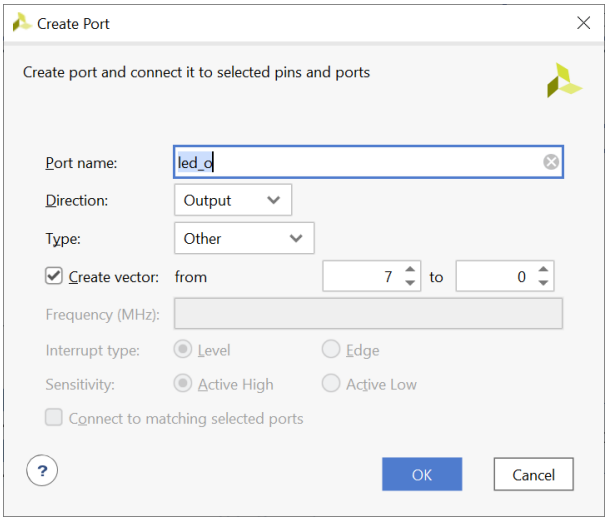


Figure 6: Port Creation

Now it is necessary to connect the MSB of the HDL counter to the output port and for this purpose two blocks are necessary:

- Slice IP
- Concat IP

The Slice IP needs to accept a 32 bit wide input and return the MSB:

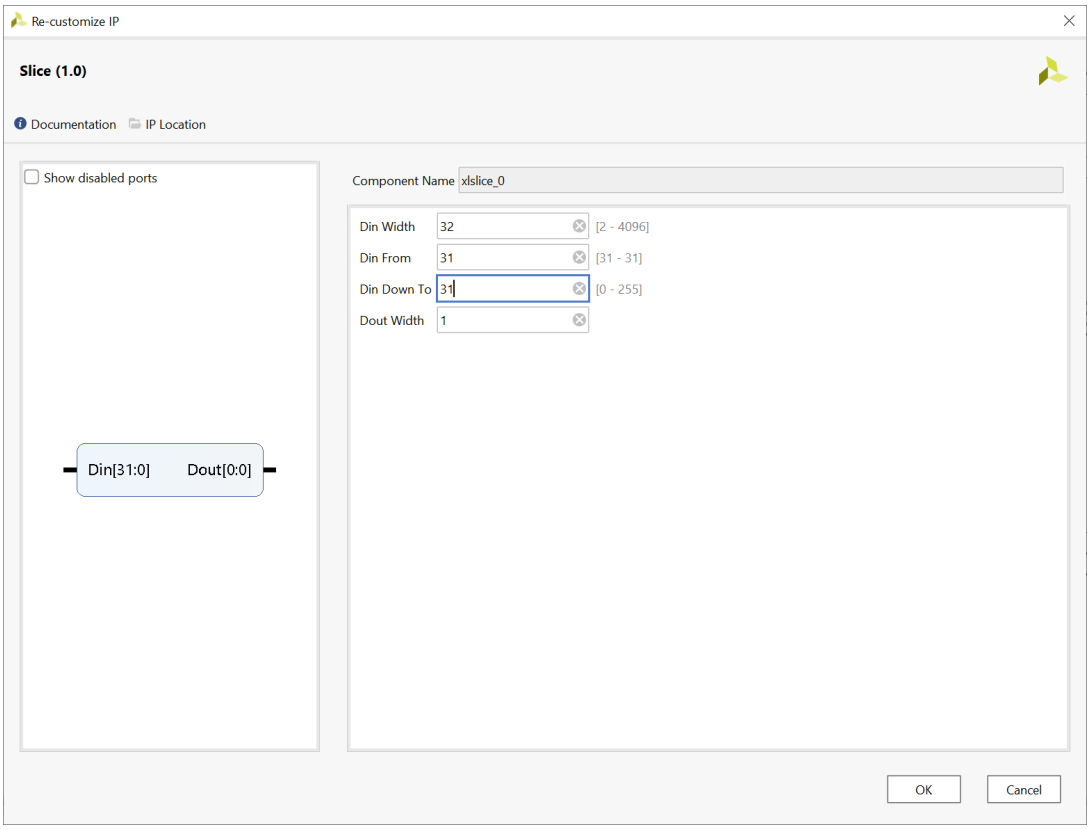


Figure 7: Slice Configuration

The Concat bundle 8 inputs of 1 bit each:

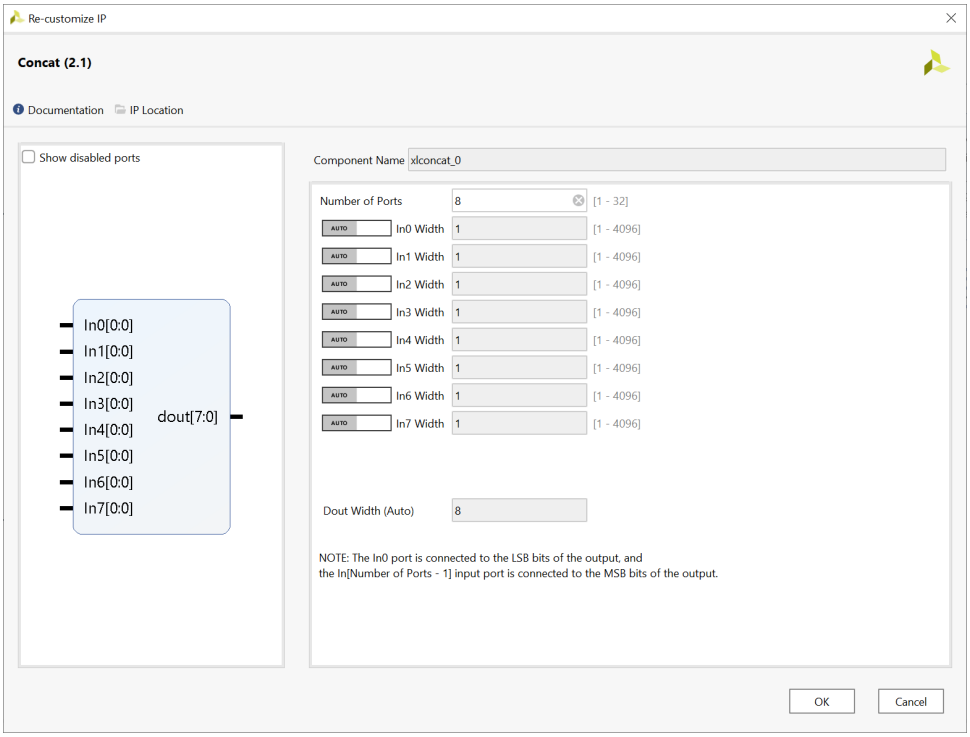


Figure 8: Concat Configuration

Then it is possible to wire the AXI GPIO, HDL counter, Slice IP, Concat IP and *led_o* port. The final desing looks like this:

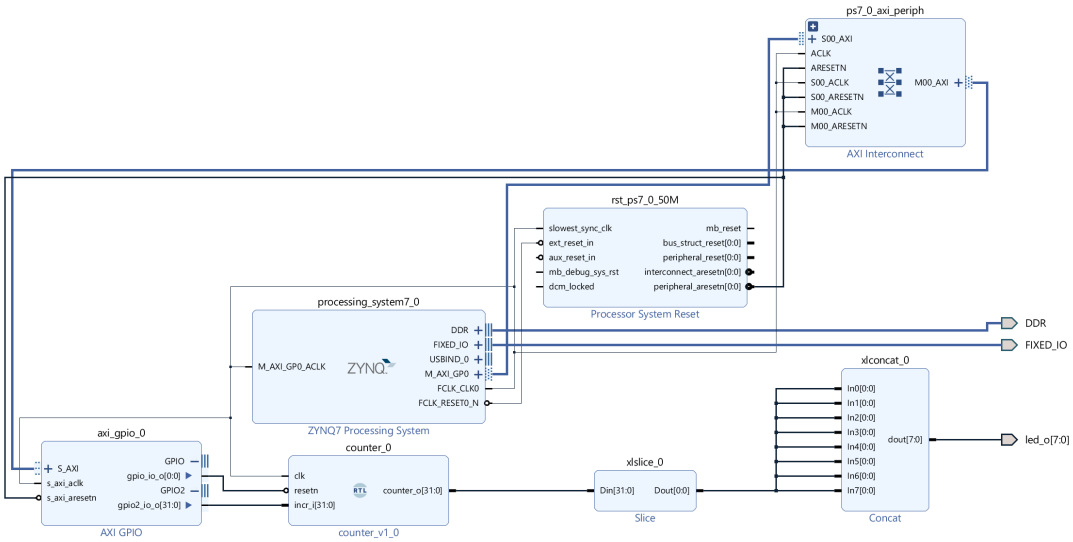


Figure 9: Final Design

After the creation of the HDL design wrapper and bitstream it is possible to transfer the bit, hwh and tcl file into the RedPitaya.

The first action that needs to be performed is the reset of the board, along with the loading of the desing using the bit file.

```
import pynq
pynq.PL.reset()
ol = pynq.Overlay("LED_blink.bit")
```

Then it is possible to design the blinking frequency, the clock frequency and configuring the channels by running:

```
blinkFreq = 1
clkFreq = 50e6
incr = int(2**32 * blinkFreq / clkFreq)

#Enable counter
ol.axi_gpio_0.channel1.write(val=1, mask=0x1)

#Set counter increment
ol.axi_gpio_0.channel2.write(val=incr, mask=0xffffffff)
```

5.2 DMA Access project

This goal of this project is to test the memory resources present on the PS of the FPGA On the RedPitaya is present a 512MB DDR memory, which serves as RAM for the Zynq PS but it can also be used by the Zynq PL to exchange data.

A PYNQ Jupyter Notebook will be used to preload waveforms to RAM memory, simultaneously trigger the waveform generation/acquisition and plot the acquired samples. [2]

The necessary blocks are: Zynq 7 Processing System, Processor System Reset, AXI GPIO, 2x AXI Interconnect, 2x AXI Direct Memory Access, RedPitaya-125-14-clk, RedPitaya-125-14-adc, RedPitaya-125-14-dac.

Once all the instances are created the tab *Block Automation* will automatically create the route to the DDR and the FIXED_IO ports.

The Zynq 7 needs to be configured to accept an HP Slave AXI interface(*double click on Zynq 7 then PS-PL Configuration then HP Slave AXI Interface*) and enable the HP0 and the HP1 with data width 64 bit.

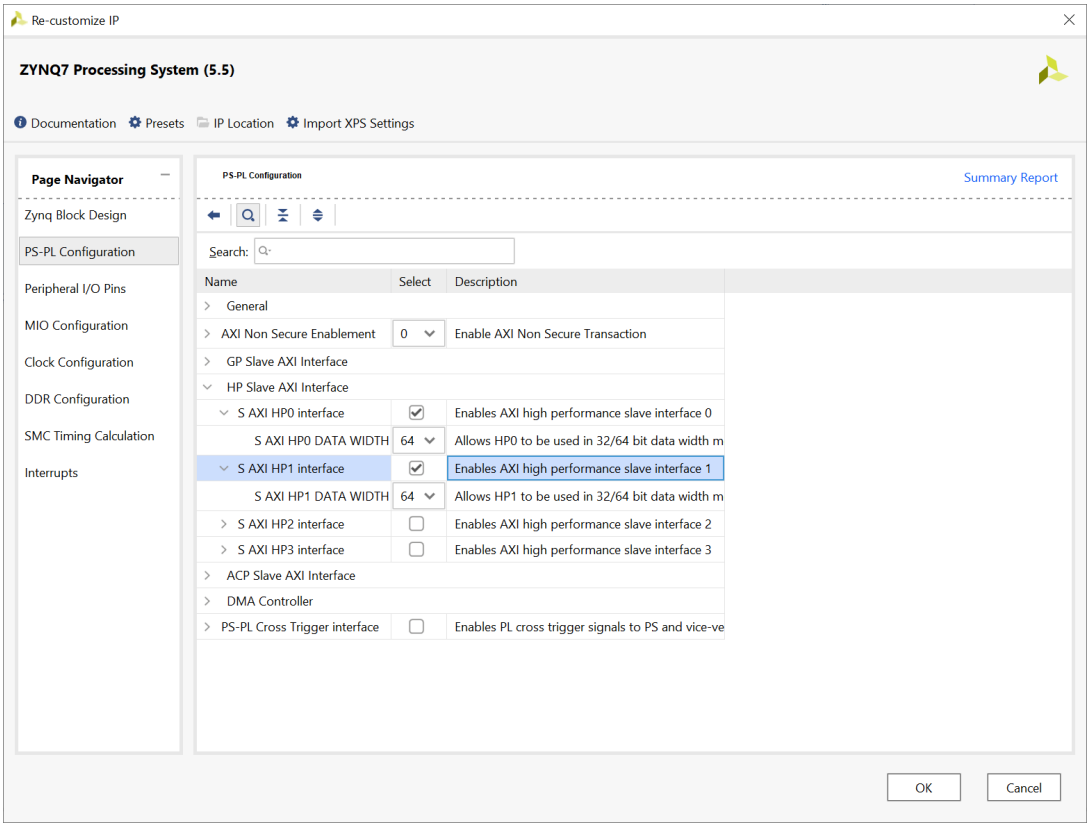


Figure 10: Zynq Processing System Configuration

The AXI GPIO can be configured as the previous project with two channel, the first will accept 1-bit output and the second a 32-bit output. Both of the AXI Interconnect needs to be configured to have 2 slaves and 1 master interface:

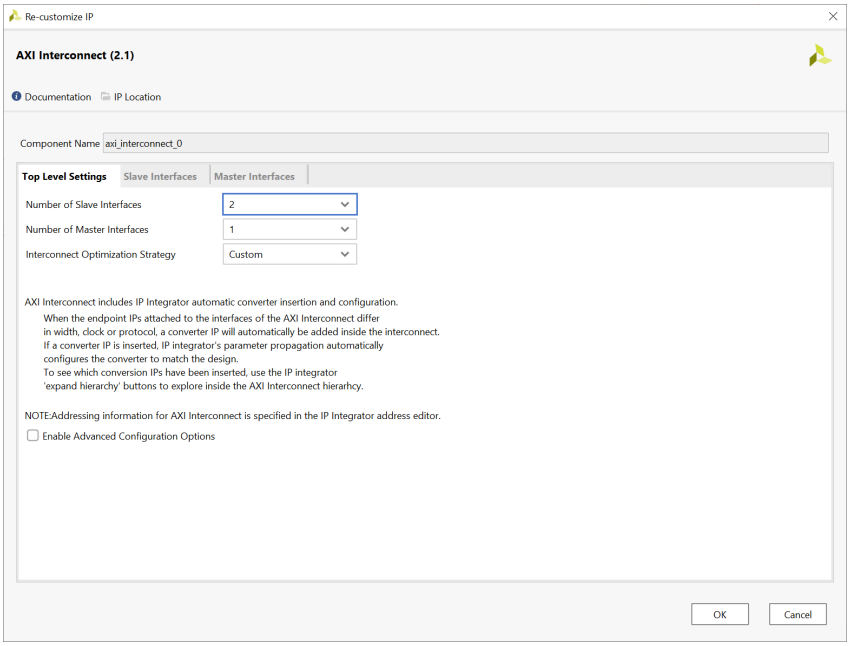


Figure 11: AXI Interconnect Configuration

And also the both the DMAs needs to be configured to accept a read and a write channel.

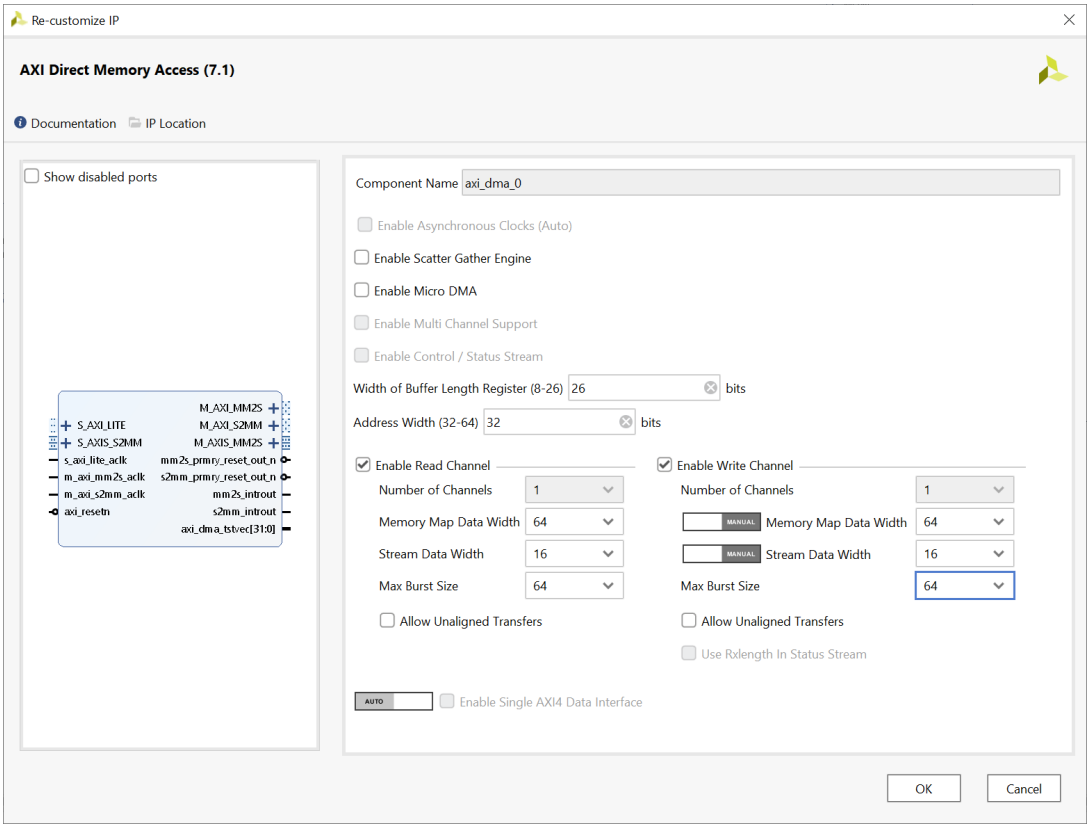


Figure 12: AXI DMA Configuration

Then it is possible to connect the clock and the reset wires.
In order to control the data streams going into and out of the DMA engine a simple HDL module is necessary, this module will be necessary to hold the incoming and outgoing data streams until trigger signal is received a rise the AXI stream tlast line when the last data sample gets acquired.

The module in Verilog is available here 7.2.

On the design four stream controller are necessary and they needs to be connected as follow:

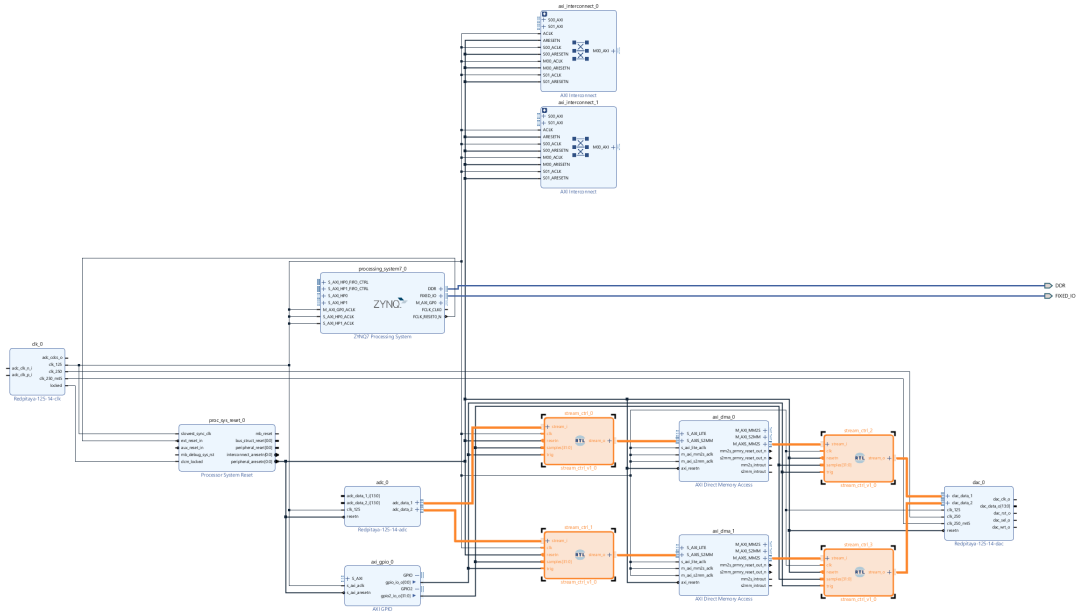


Figure 13: Stream Control New Configuration

In order to access the data by the Zynq 7 to the DMAs the MM2S and S2MM interfaces needs to be connected.

To makes it possible for the FPGA to read the ADC, DAC and Clock constraints is necessary to uncomment the following sections in the xdc file:

- ADC (lines from 8 to 56)
- DAC (lines from 59 to 90)
- Clock (lines from 180 to 183)

At this point is possible to create the inputs and outputs ports for the blocks:

- RedPitaya-125-14-clk
- RedPitaya-125-14-adc
- RedPitaya-125-14-dac

Each port can be created with the shorcut *CTRL+K*. To connect the memory mapped interfaces is possible to run the *Connection Automation* tab and select AXI GPIO, DMA and Zynq Interfaces and the final design becomes like this:

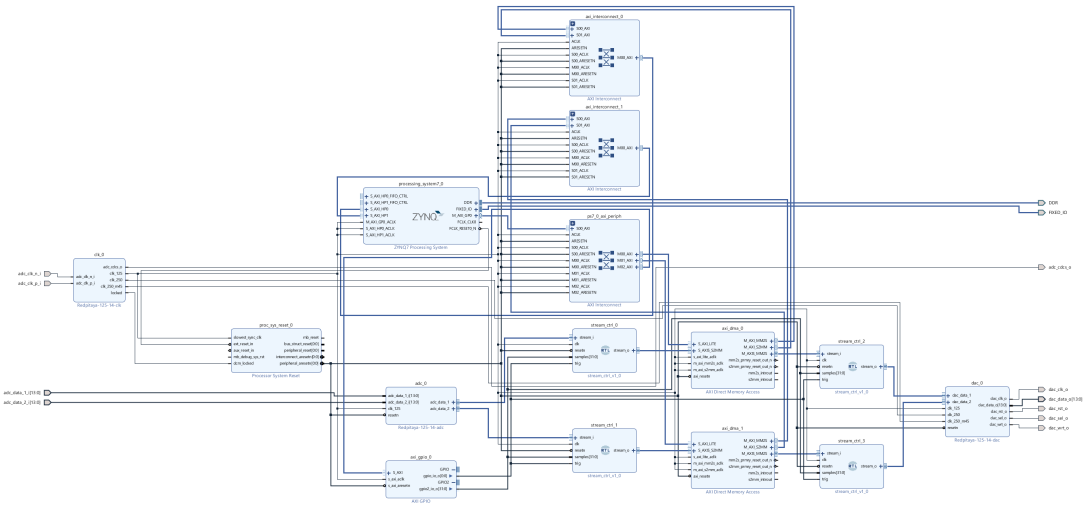


Figure 14: Final Design

The last step before generating the bitstream is to check the address editor and automatically assign an address space to all the interfaces, this can be easely done by rigth-clicking on *Network 0* and select *Assign All*

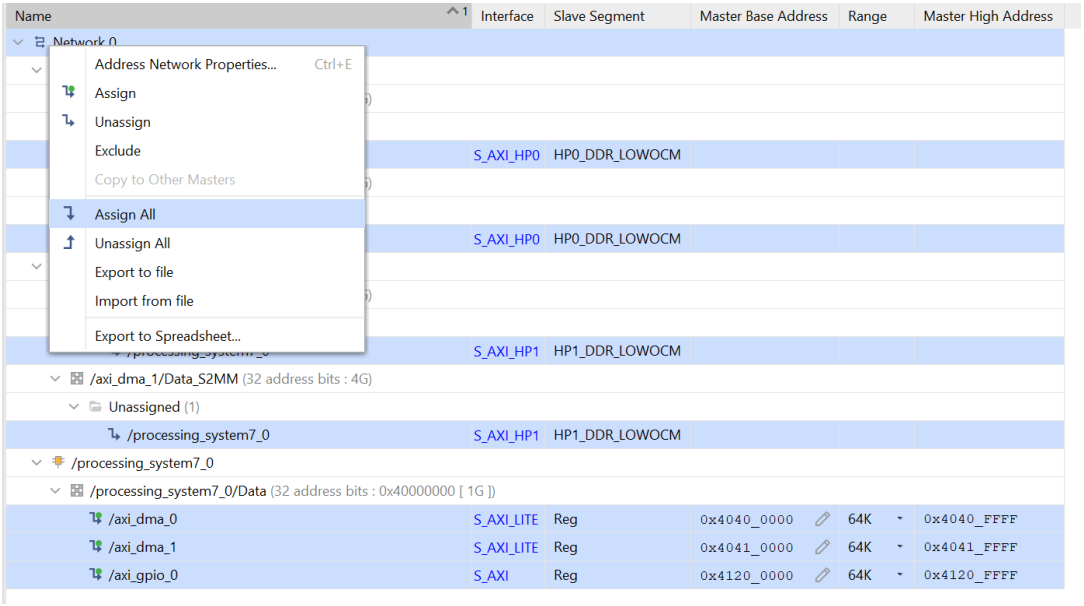


Figure 15: Assigning new addresses to ports

The next step is performed on the Jupyter Notebook, once a new notebook is created and the bit file of the project is already transfered on the FPGA is possible to reset the design:

```
import pynq
import numpy as np
import matplotlib.pyplot as plt
pynq.PL.reset()
ol = pynq.Overlay("DMA_transfer.bit")
```

The is possible to set the parameters:

```
#Parameters
samples = 10000
clk_freq = 125e6

#Time array
t = np.arange(samples) / clk_freq

#Gaussian
t_center = 0.5 * samples / clk_freq
t_width =  0.15 * samples / clk_freq
gaussian = np.exp(-(t - t_center)**2 / (2 * t_width**2))

#Sin/Cos
freq = 200e3
sin = np.sin(2 * np.pi * freq * t)
cos = np.cos(2 * np.pi * freq * t)

#Waveforms
waveform_out_1= np.array((2**15 - 1) * sin * gaussian, dtype=np.int16)
waveform_out_2= np.array((2**15 - 1) * cos * gaussian, dtype=np.int16)
```

and then allocate the memory:

```
#Parameters
samples = 10000
clk_freq = 125e6

#Time array
t = np.arange(samples) / clk_freq

#Gaussian
t_center = 0.5 * samples / clk_freq
t_width =  0.15 * samples / clk_freq
gaussian = np.exp(-(t - t_center)**2 / (2 * t_width**2))

#Sin/Cos
freq = 200e3
sin = np.sin(2 * np.pi * freq * t)
cos = np.cos(2 * np.pi * freq * t)

#Waveforms
waveform_out_1= np.array((2**15 - 1) * sin * gaussian, dtype=np.int16)
waveform_out_2= np.array((2**15 - 1) * cos * gaussian, dtype=np.int16)
```

The previously created channels1 is now set to low to avoid any problems during data transmission, while on the channel2 the number of samples is sended.

```
ol.axi_gpio_0.channel1.write(val=0, mask=0x1) #Trig low
ol.axi_gpio_0.channel2.write(val=samples, mask=0xffffffff) #Set samples

ol.axi_dma_0.recvchannel.transfer(input_buffer_1)
ol.axi_dma_0.sendchannel.transfer(output_buffer_1)
ol.axi_dma_1.recvchannel.transfer(input_buffer_2)
ol.axi_dma_1.sendchannel.transfer(output_buffer_2)

ol.axi_gpio_0.channel1.write(val=1, mask=0x1) #Trig high

ol.axi_dma_0.recvchannel.wait()
ol.axi_dma_0.sendchannel.wait()
ol.axi_dma_1.recvchannel.wait()
ol.axi_dma_1.sendchannel.wait()
```

Finally is possible to plot the data:

```
plt.figure(figsize=(12,6))
plt.plot(t * 10**6, input_buffer_1 / 2**15, label = "ch1")
plt.plot(t * 10**6, input_buffer_2 / 2**15, label = "ch2")
plt.xlabel("Time (us)")
plt.ylabel("Input Voltage (V)")
plt.grid()
plt.legend()
plt.show()
```

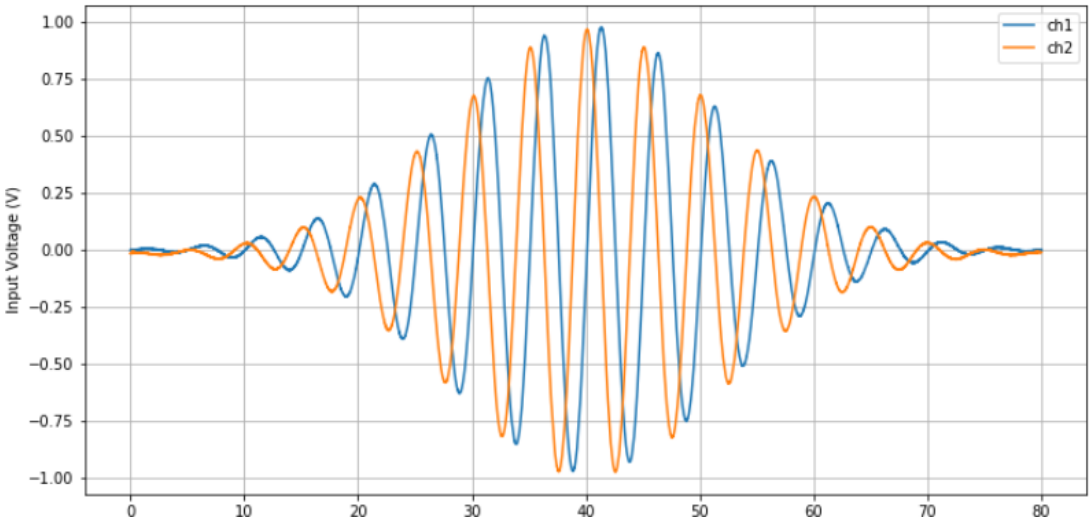


Figure 16: DMA Plot

5.3 Summing Arrays

The next section focuses on the sum of two arrays in an FPGA.
The necessary blocks are: Zynq 7 Processing System, RedPitaya Clock, Processor System Reset, 2 AXI DMA, AXI GPIO.

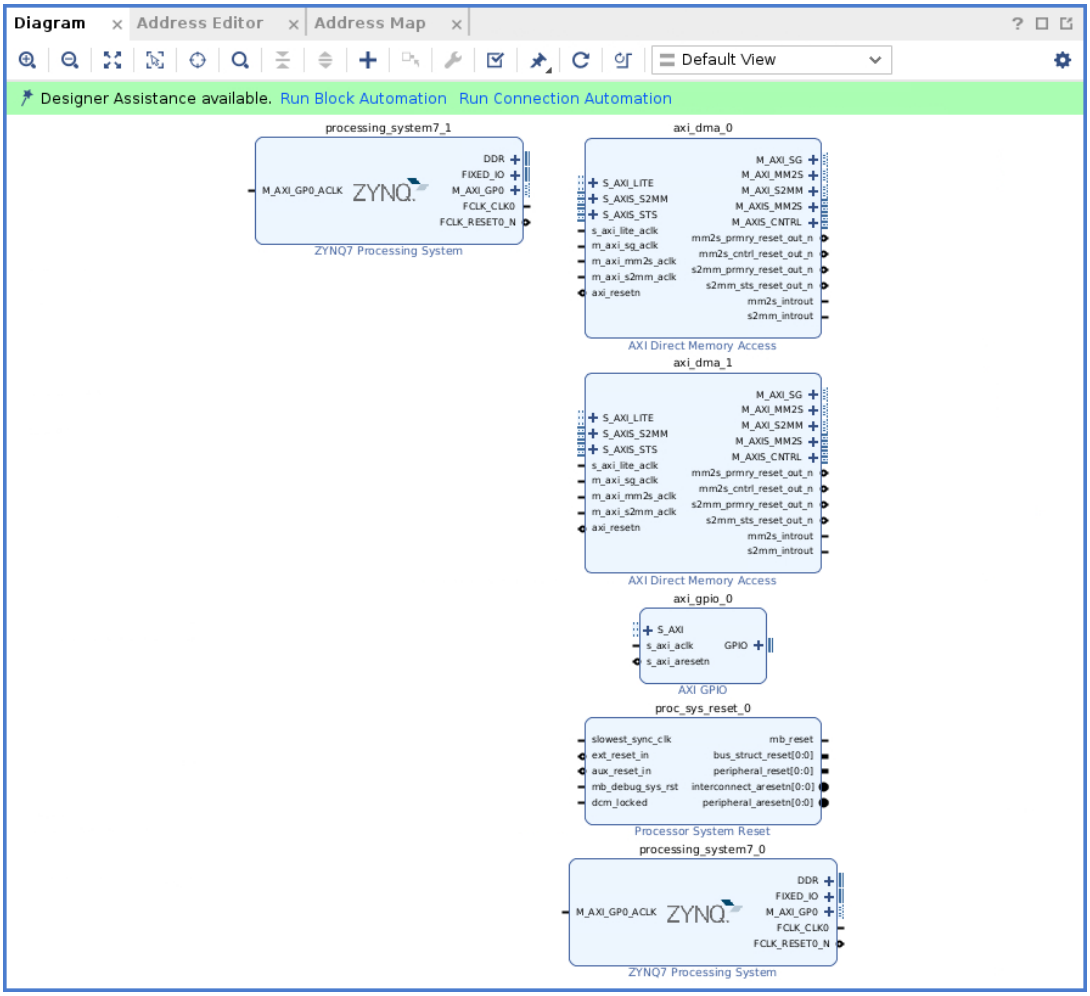


Figure 17: Initial design

The RedPitaya Clock needs to be connected externally so it is necessary to create the correspondent ports, this step can be completed using the *CTRL+K* command.

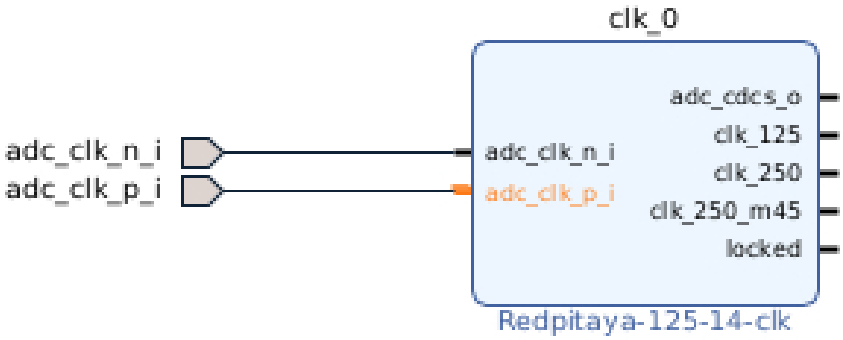


Figure 18: Axi GPIO port connectios

The *Run Block Automation* tab will automatically connect the DDR and FIXED_IO. Next is necessary to configure the DMA(*Direct Memory Access*). The goal is to minimize the number of DMAs used, in this case is possible to use only two. The first DMA will use both the writing and the reading channel like the previous project, while the second one will only need the reading channel configuration.

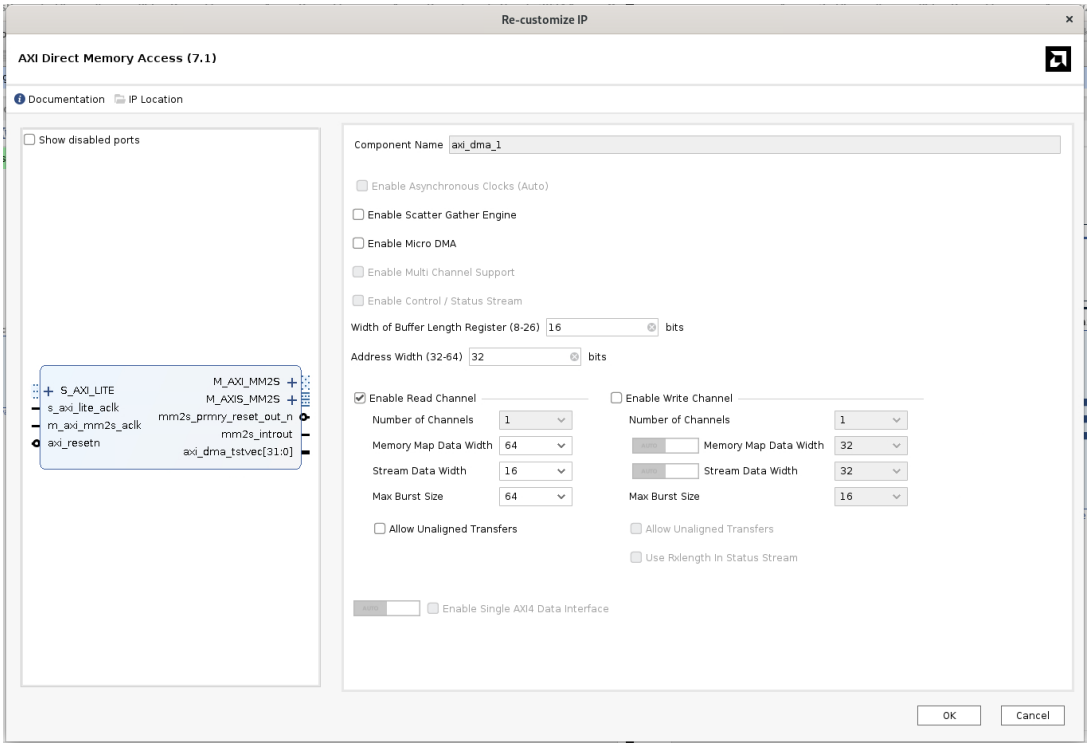


Figure 19: Receiving DMA configuration

The AXI GPIO module synchronize the DMAs using a trigger along with the number of samples which is necessary to communicate to the FPGA the length of the array, to perfom this tasks the AXI GPIO needs to be configured with the usual 1-bit channel and the 32-bit channel.

The first channel has only 1-bit wire (0 or 1), and it will be the trigger, while the second will be the sample number channel.

The HP0 interface on the Zynq7 needs to be set on. To turn it on double-click on the block and select the PS-PL Configuration:

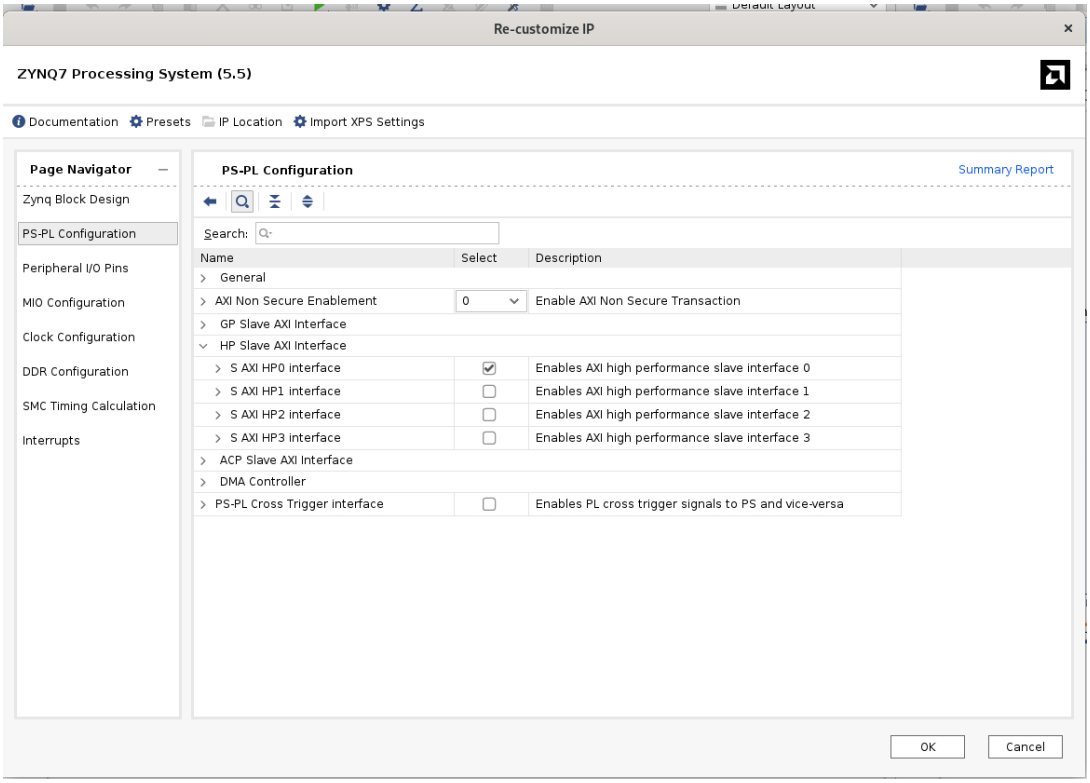


Figure 20: HP0 Configuration

Now the clock and the reset wire can be connected.

The tab *Run Connection Automation* will automatically run all the necessary connections, only the AXI GPIO needs to be wired manually. Following this step would bring the design to look like this:

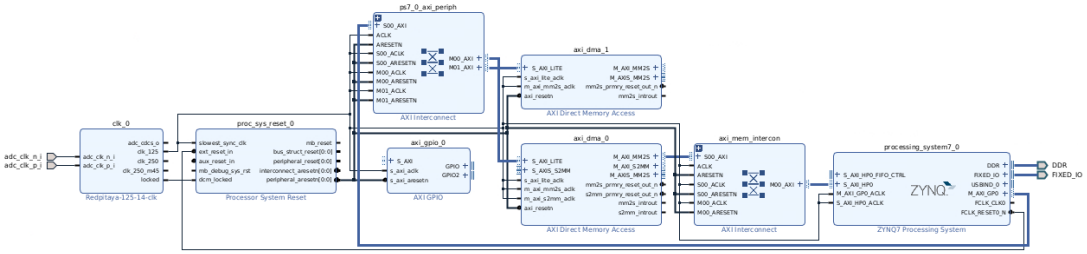


Figure 21: AXI GPIO configuration

The next step is to use the *Stream Controller* to control the flow of the data going into the *Stream Adder*. This task can be performed using a modified version of the previously shown *stream_ctl.vhd*. The code is available here 7.3. While the adder is available here 7.4.
The final design should looks like this:

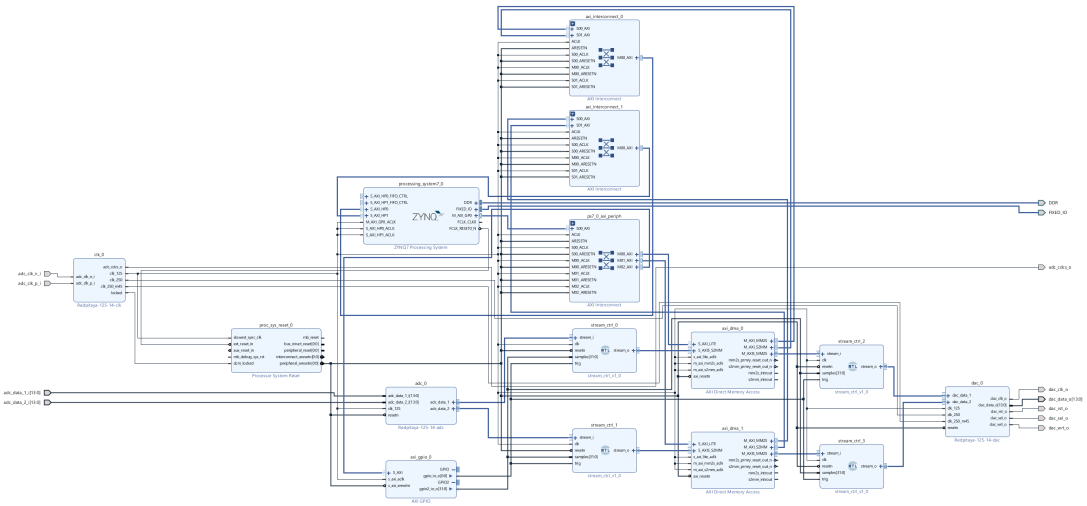


Figure 22: Final Configuration

To test this project once it's on the RedPitya this script returns as a output [3 4 5 6 7 8 9 10 11 12]:

```
import pynq
pynq.PL.reset()
ol = pynq.Overlay("summingArrays.bit")
import numpy as np

array0 = np.array([2,2,2,2,2,2,2,2,2,2])
array1 = np.array([1,2,3,4,5,6,7,8,9,10])

samples = len(array0)

input_buffer_1 = pynq.allocate(shape=(samples,), dtype=np.int16)
output_buffer_1 = pynq.allocate(shape=(samples,), dtype=np.int16)
output_buffer_2 = pynq.allocate(shape=(samples,), dtype=np.int16)

np.copyto(output_buffer_1, array0)
np.copyto(output_buffer_2, array1)

ol.axi_dma_0.sendchannel.transfer(output_buffer_1)
ol.axi_dma_1.sendchannel.transfer(output_buffer_2)
ol.axi_dma_0.recvchannel.transfer(input_buffer_1)

ol.axi_gpio_0.channel1.write(val=1, mask=0x1)
ol.axi_gpio_0.channel2.write(val=10, mask=0xFFFFFFFF)

print(input_buffer_1)
```

On the GitHub repo it is possible to find the library to perform this commands automatically.[5]

5.4 Results

The outcome of this project led to the development of a comprehensive Python library specifically designed to facilitate seamless communication with the FPGA.
This library's primary function is to enable the efficient summation of two arrays of values, thereby streamlining computational processes. The library is intended to serve as a foundational tool for subsequent projects, particularly those aimed at advancing the GQuEST experiment. By providing a robust and user-friendly interface, the library significantly reduces the complexity involved in FPGA programming and operations.
In addition to the library, a detailed GitHub repository was established. This repository includes extensive documentation, example codes, and tutorials all aimed at expediting the learning curve for individuals and teams interested in mastering FPGA methods for array summation. The repository serves as a valuable resource, offering insights and practical guidance to both novice and experienced developers. Through these efforts, the project not only achieved its immediate goals but also laid the groundwork for future innovations and developments in the field.

6 Conclusions and future works

Further works may focus on the implementation and extension of the library to include a correlator module, which could significantly enhance the experiment's analytical capabilities. The addition of a correlator module is anticipated to allow for a more precise and comprehensive analysis of the experiment results, thereby providing deeper insights into the quantum states being studied.

In conclusion, the development of the summing library marks a significant step forward for the GQuEST experiment. By focusing on the implementation of a correlator module in future works, we can enhance the precision and depth of our analyses, ultimately contributing to the advancement of quantum state estimation and the broader field of quantum computing.

7 Codes

In this section all the files used during the tutorial will be displayed.

7.1 counter.vhd

```
module counter (
    input clk,
    input resetn,
    input [31 : 0] incr_i,
    output reg [31 : 0] counter_o
    );

    always @(posedge clk) begin
        if(resetn == 0) begin
```

```
            counter_o <= 0;
        end else begin
            counter_o <= counter_o + incr_i;
        end
    end

endmodule
```

7.2 streamcontroller.vhd

```
module stream_ctrl #(
parameter DATA_WIDTH = 16
)(
input clk,
input resetn,
input [31 : 0] samples,
input trig,
input [DATA_WIDTH - 1 : 0] stream_i_tdata,
input stream_i_tvalid,
output stream_i_tready,
output [DATA_WIDTH - 1 : 0] stream_o_tdata,
output stream_o_tvalid,
output stream_o_tlast,
input stream_o_tready
);

localparam [0:0] IDLE=0, RUNNING=1;
reg state;
reg [31 : 0] counter;
reg trig_old;



always @(posedge clk) begin
    if (resetn == 0) begin
        state <= IDLE;
        counter <= 0;
        trig_old <= 0;
    end else begin
        case(state)
            IDLE: begin
                counter <= 0;
                if((trig == 1) && (trig_old == 0)) begin
                    state <= RUNNING;
                end
            end

            RUNNING: begin
                if((stream_i_tvalid == 1) && (stream_o_tready == 1)) begin
                    counter <= counter + 1;
                    if(counter == (samples - 1)) begin
                        state <= IDLE;
                    end
                end
            end
        endcase
```

```
            trig_old <= trig;
        end
    end

    assign stream_o_tdata = stream_i_tdata;
    assign stream_o_tvalid = (state == RUNNING) ? stream_i_tvalid : 0;
    assign stream_i_tready = (state == RUNNING) ? stream_o_tready : 0;
    assign stream_o_tlast = ((state == RUNNING) && (counter == (samples - 1))) ? 1 : 0;

endmodule
```

7.3 streamcontroller2.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


entity stream_ctrl is
    Generic( DATA_WIDTH : integer := 16);
    Port ( clk : in STD_LOGIC;
           resetn : in STD_LOGIC;
           samples : in STD_LOGIC_VECTOR(31 downto 0);
           trig : in STD_LOGIC;
           stream_i_tdata : in STD_LOGIC_VECTOR(DATA_WIDTH - 1 downto 0);
           stream_i_tvalid : in STD_LOGIC;
           stream_i_tready : out STD_LOGIC;
           stream_o_tdata: out STD_LOGIC_VECTOR(DATA_WIDTH - 1 downto 0);
           stream_o_tvalid : out STD_LOGIC;
           stream_o_tlast : out STD_LOGIC;
           stream_o_tready : in STD_LOGIC

           );
end stream_ctrl;

architecture Behavioral of stream_ctrl is
    type STATE_TYPE is (IDLE, RUNNING);
    signal state : STATE_TYPE := IDLE;
    signal counter : UNSIGNED(31 downto 0) := (others => '0');
    signal trig_old : STD_LOGIC := '0';

    begin
        process(clk)
            begin
                if rising_edge(clk) then
                    if resetn='0' then
                        state <= IDLE;
                        counter <= (others => '0');
                        trig_old <= '0';
                    else

                        case state is
                            when IDLE =>
                                counter <=  (others => '0');
                                if trig = '1' and trig_old = '0' then
```

```
                                state <= RUNNING;
                            end if;

                        when RUNNING =>
                            if stream_i_tvalid = '1' and stream_o_tready = '1' then
                                counter <= counter + 1;
                                if counter = (unsigned(samples) - 1) then
                                   state <= IDLE;
                                end if;
                            end if;
                    end case;

                    trig_old <= trig;

                end if;
            end if;
        end process;

stream_o_tdata <= stream_i_tdata;

stream_o_tvalid <= stream_i_tvalid when state = RUNNING else
                   '0';

stream_i_tready <= stream_o_tready when state = RUNNING else
                   '0';

stream_o_tlast <= '1' when (state = RUNNING) and (counter = (unsigned(samples) - 1)) else
                  '0';

end Behavioral;
```

7.4 Adder.vhd

```
use library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;



entity stream_adder is
    Generic (DATA_WIDTH : integer := 16);
    Port ( clk : in STD_LOGIC;
           resetn : in STD_LOGIC;
           data_0_i_tdata : in STD_LOGIC_VECTOR (DATA_WIDTH - 1 downto 0);
           data_0_i_tvalid : in STD_LOGIC;
           data_0_i_tlast : in STD_LOGIC;
           data_0_i_tready : out STD_LOGIC;

           data_1_i_tdata : in STD_LOGIC_VECTOR (DATA_WIDTH - 1 downto 0);
           data_1_i_tvalid : in STD_LOGIC;
           data_1_i_tlast : in STD_LOGIC;
           data_1_i_tready : out STD_LOGIC;

           data_o_tdata : out STD_LOGIC_VECTOR (DATA_WIDTH - 1 downto 0);
           data_o_tvalid : out STD_LOGIC;
           data_o_tlast  : out STD_LOGIC;
           data_o_tready : in STD_LOGIC);
```

```
end stream_adder;

architecture Behavioral of stream_adder is
    signal sum_reg : signed(DATA_WIDTH - 1 downto 0) := (others => '0');
    signal valid_reg : std_logic := '0';
    signal tlast_reg : std_logic := '0';

begin
    process(clk)
    begin
        if rising_edge(clk) then
            if resetn = '0' then
                sum_reg <= (others => '0');
                valid_reg <= '0';
                tlast_reg <= '0';
            else
                if data_0_i_tvalid='1' and data_1_i_tvalid='1' and data_o_tready='1' then
                sum_reg <= signed(data_0_i_tdata) + signed(data_1_i_tdata);
                valid_reg <= data_0_i_tvalid and data_1_i_tvalid;
                tlast_reg <= data_0_i_tlast and data_1_i_tlast;
                else
                    valid_reg <= '0';
                    tlast_reg <= '0';
--                sum_reg <= signed(data_0_i_tdata) + signed(data_1_i_tdata);
--                valid_reg <= data_0_i_tvalid and data_1_i_tvalid;
                end if;
           end if;
        end if;
    end process;

    data_o_tvalid <= valid_reg;
    data_o_tdata <= std_logic_vector(sum_reg);
    data_o_tlast <= tlast_reg;

    data_1_i_tready <= data_o_tready;
    data_0_i_tready <= data_o_tready;



end Behavioral;
```

References

[1] David Nguyen, Chris Stougthton, *GQuEST Control System Resolution*, FERMILAB-PUB-24-0412-STUDENT, 2024.

[2] Pau Goméz, *FPGA For Scientists*, github repo

[3] *DMA Fundamentals on various PC Platforms*

[4] *Zynq 7*, Xilinx-wiki

[5] *AddingnumberisFPGA*, github repo