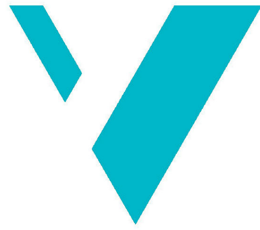


Western Norway
University of
Applied Sciences

BO21E-05

Design of an embedded data storage system using PYNQ

Asbjørn Magnus Midtbø
Harald Træet Lægreid

June 1, 2021

Document control

Report title: Design of an embedded data storage system using PYNQ	*Date/Version* June 1, 2021
	Report number: BO21E-05
Authors: Asbjørn Magnus Midtbø Harald Træet Lægreid	*Course:* EEL18
	Number of pages including appendixes: 69
Supervisor at Western Norway University of Applied Sciences: Svein Haustveit	*Security classification:* Open
Comments: We, the authors, allow publishing of the report.	

Contracting entity: Microelectronics group at the University of Bergen	*Contracting entity's reference:*
Contact at contracting entity: Kjetil Ullaland: Kjetil.Ullaland@uib.no	

Revision	**Date**	**Status**	**Performed by**
1	1/6-2021	Release	AMM, HTL

Preface

The following work was done in the spring of 2021 at the Department of Computer science, Electrical engineering and Mathematical sciences at the Western Norway University of Applied Sciences for the microelectronics group at the University of Bergen. The work was a continuation on the Master's thesis' by *Mats Heigre* and *Alexander Nesse* whom laid the groundwork for the development of our design. At the beginning of the project we had some knowledge of the fundamentals of embedded systems design, and had previously dabbled in Python development in earlier internships and hobby projects. These experiences were quite valuable from the outset of the project

Acknowledgements

We would like to thank our supervisor *Assistant Professor Svein Haustveit*, whom introduced us to embedded system design last spring and guided us through this project. The project would also never have gotten as far as it did without the input of our contractor at the microelectronics group, namely *Professor Kjetil Ullaland*. He helped with his previous experiences with the Xilinx environment and the ALOFT project. He also gave good input on the academic writing of this thesis.

We would also like to thank our friends, and fellow students in *EEL18* whom helped us get through these three last years, and especially this last digitally defined year.

Asbjørn Magnus Midtbø
Harald Træet Lægreid

Summary

The main goal of this project has been to evaluate the use of a Linux based Operating System (OS) for the ALOFT project, namely the PYNQ Operating System.

The conclusion to this is that the project could have good use for the PYNQ Operating System. The reasoning for this is that:

- It is easier to start developing on a Linux based OS like PYNQ compared to a Real Time Operating System (RTOS), which was previously trialed.
- PYNQ utilizes Python as its standard programming language, which is a well known and easy to learn programming language. Python is also already used by the future users.
- PYNQ arrives with several premade hardware drivers which speeds up embedded development.

During this project there has been designed a simple, but effective, means of delivering data from an Analog to Digital Converter (ADC) source to a storage space using PYNQ on a PYNQ-Z1 Circuit board (Z1). This has been accomplished with minimal Central Processing Unit (CPU) intervention, and utilizes both Xilinx and custom Intellectual Property (IP) to reach the bit-width and data throughput requirements of the contracting entity. The design has been created with modularity and standardization in mind as to ease the future development of the project.

On the software side there has been created a class and several programs which manages the hardware. This should lay a good groundwork for future developments. In addition there has been developed a means of managing parts of this hardware from the ground while the Z1 is running.

All in all the system has been reliably tested to surpass the data throughput requirement, with an average rate of 5 MB/s continuously. However, we do believe that the data throughput could be further surpassed if needed. Currently the Achilles heel of the system exists somewhere in the Processing System (PS) someplace between the system memory, and the USB memory. This USB memory was used as storage during testing. We believe that the use of a faster interface protocol could speed up the system considerably. However, this cannot be proved at this point because the only connection to the PS on the Z1 is a USB 2.0 connector. The need for a faster interface also highly depends on the final needs of the ALOFT project.

Contents

1 Introduction **1**
1.1 Contracting entity . . . 1
1.2 Problem description . . . 1
1.3 Main idea and solution . . . 2
1.4 Report layout . . . 2

2 Specifications of requirements **3**

3 Problem analysis **4**
3.1 Interface of the MUX-network . . . 4
3.1.1 Parallel interface bus . . . 5
3.1.2 AXI4-Stream . . . 5
3.1.3 Conclusion on choice of interface . . . 6
3.2 MUX-network to storage device connections . . . 6
3.2.1 A purely hardware based solution . . . 7
3.2.2 A software and DMA based solution . . . 7

4 Realization of selected solution **9**
4.1 PYNQ Z1 . . . 9
4.1.1 PYNQ OS . . . 9
4.1.2 Overlays . . . 10
4.2 Hardware . . . 11
4.2.1 AXI4-Stream accommodation in the ALOFT system . . . 11
4.2.2 Data flow . . . 11
4.2.3 Control . . . 12
4.2.4 Interrupts . . . 12
4.2.5 Xilinx IP . . . 13
4.2.6 Custom hardware modules . . . 15
4.2.7 FPGA resource utilization . . . 20
4.3 Software . . . 21
4.3.1 Configuration . . . 21
4.3.2 Initialization . . . 21

4.3.3 Methods . . . 22
4.3.4 The use of asynchronous programming . . . 25
4.3.5 Logging . . . 25
4.3.6 Signaling . . . 26
4.3.7 Remote control . . . 26

5 Testing **28**
5.1 Testing of custom hardware . . . 28
5.2 Assessing stored data . . . 28
5.3 Data throughput performance . . . 29
5.4 Adding FIFOs to enhance performance and alleviate bursts . . . 30

6 Discussion and further work **31**
6.1 Further development of data monitoring . . . 31
6.2 Last received data-point before overflow . . . 31
6.3 Remote control functionality . . . 31
6.4 A faster storage interface . . . 32
6.5 Adapting to the ALOFT circuit board . . . 32
6.6 Known software bugs . . . 32

7 Conclusion **34**
7.1 The use of PYNQ for further development of the project. . . . 34
7.2 The ALOFT design . . . 34

Appendix A References **i**

Appendix B Acronyms **iii**

Appendix C Project management **v**
C.1 Project organization . . . v
C.2 Project form . . . v

Appendix D Git Repository **vii**

Appendix E User manual **viii**
E.1 Building the hardware in Vivado . . . viii

E.2 Setup of the PYNQ-Z1 board viii
E.3 Running the demos ix
E.3.1 AloftDataPolling.py x
E.3.2 AloftHlsTest.py x
E.4 Using the Aloft class xii
E.4.1 Initializing xii
E.4.2 Predefined parameters xii
E.4.3 Methods xiii
E.4.4 Attributes xiv

Appendix F Building the hardware from scratch xvi

F.1 Creating the Vivado project xvi
F.2 Adding and connecting IPs xvii
F.2.1 Zynq 7 Processing System xvii
F.2.2 AXI Direct Memory Access xx
F.2.3 AXI4-Stream Data FIFO xx
F.2.4 Adding Custom IPs xxi
F.2.5 AXI4-Stream Switch xxii
F.2.6 AXI GPIO xxii
F.2.7 AXI Interrupt Controller xxiv
F.3 Creating hierarchies and routing signals xxiv
F.4 Running implementation and generating bitstream xxv

List of Figures

1 An ER-2 in flight. Earlier versions of the ALOFT project has resided in the pod under the right wing. Pictured by NASA . . . 1
2 A general setup of the finished system . . . 4
3 An example packet transfer using the AXI4-Stream interface. The packet size is set to 9 transfers in this example. . . . 6
4 An abstracted overview of a software and DMA based solution. . . . 8
5 PYNQ Z1 . . . 9
6 PYNQ Z1 base overlay . . . 10
7 The several paths the data can be transferred . . . 12
8 The configuration of several hardware settings is done using a control register . . . 12
9 Interrupt connections . . . 13
10 AXI4-Stream Switch blocks in Vivado . . . 14
11 Counter block diagram . . . 15
12 Counter state diagram . . . 16
13 Bit Compressor block diagram . . . 16
14 Bit Compressor timing diagram . . . 17
15 Bit Compressor simplified state diagram . . . 17
16 Bit Expander simplified state diagram . . . 18
17 Packet Counter block diagram . . . 18
18 Packet Counter timing diagram . . . 19
19 Packet counter state diagram . . . 19
20 FPGA resource utilization (%) . . . 20
21 Flowchart for initialization . . . 21
22 Flowchart for saving algorithm . . . 23
23 Flowchart for HLS test . . . 24
24 The GUI of the software . . . 26
25 UVVM log snippet for the Bit Expander . . . 28
26 5 MB/s AloftDataPolling.py resource usage . . . 29
27 8 MB/s AloftDataPolling.py resource usage . . . 30
28 Gantt scheme . . . vi
29 New project summary . . . xvi

30 Vivado working area xvii
31 The Zynq 7 processing system xviii
32 Enabling high performance port xviii
33 Enabling interrupt ports xix
34 Additions to the Zynq 7 processing system xix
35 Settings for the DMA IP xx
36 Settings for the FIFO IP xxi
37 Data flow to explain AXI4-Stream connections xxi
38 Connection automation for the AXI GPIO IPs xxiii
39 Control register connections xxiii
40 Hierarchy used for the ALOFT system xxiv
41 Connecting button 0 to the AXI GPIO input by selecting correct package pin xxv

List of Tables

1 Data rates stored in ROM . . . 15
2 Buffer sizes stored in ROM . . . 18
3 Time until failure for different data rates . . . 29
4 Time until failure for different data rates with several FIFOs . . . 30
5 Data source options . . . xiii
6 Buffer size options . . . xiii
7 Data rate options . . . xiii
8 Control register for the Custom IPs . . . xxiii

1 Introduction

1.1 Contracting entity

The contracting entity for this project is the microelectronics group at Department of Physics and Technology at the University of Bergen (UiB). The project contains parts of the upcoming Airborne Lightning Obsevatory for FEGS & TGFs (ALOFT) system. The group works with instrumentation for the European Space Agency, medicine, and research at CERN [1]. An earlier system similar to that of ALOFT is the Flys Eye GLM Simulator (FEGS)-Bismuth Germanate Oxide (BGO). This system measured gamma ray flashes from the NASA ER-2 high altitude aircraft pictured in figure 1. ALOFT is the successor of this system.

Figure 1: An ER-2 in flight. Earlier versions of the ALOFT project has resided in the pod under the right wing. Pictured by NASA

1.2 Problem description

The ALOFT system is meant to be an improvement of the FEGS-BGO system which was used on a flight campaign in 2017. The reason and inspiration to make an improved system is that FEGS was originally designed for satellite use, where it is desired to have a system specialised for use on an aircraft and with the limitations that ensues. It is also desirable to integrate the system on an embedded System on a Chip (SoC) platform with an accompanying Field Programmable Gate Array (FPGA). The reason for this, is that an SoC system has better internal communication, occupies less area, and is more reliable in use.

The background for our assignment is that the client wants to know if PYNQ OS is applicable for ALOFT and to other projects in the future. The client wants ease of use and less time consuming introduction to embedded FPGA development. For physics students, with limited to no knowledge in VHDL and embedded development, this can be very useful. PYNQ OS is a modified version of Ubuntu with an accompanying Python library.

The verification work is to be performed on the PYNQ-Z1 Circuit board. This is a board designed for introduction and use of PYNQ OS. The Z1 has an ZYNQ Z-7020 SoC, which is very similar to the ZYNQ Z-7030 which is intended for the ALOFT circuit board. Under the assignment we were also encouraged to come with suggestions of changes to the design of the ALOFT circuit board.

1.3 Main idea and solution

Our assignment is to use PYNQ OS to make parts of the ALOFT system. The main focus will be data storage and simple interaction over low speed Ethernet.

1.4 Report layout

In Section 2 we outline the requirements of the design which we have aimed to create. Sections 3 and 4 is dedicated to an explanation and discussion of our design, with a thorough description of the finalized parts. The results of the tests done to these parts and the design as a whole is described in Section 5. In Section 6 the work that is done is discussed, in addition to further work that could be done. The appendices contain some information which is important for future developers, especially Appendix E and F which contain descriptions of the usage of our systems, and hardware reconstruction steps.

2 Specifications of requirements

The requirements set by the client was split into 3 main parts, in addition to a last eventual part that could be done if we had the time. These requirements were:

1. Evaluate the usability of PYNQ OS for the ALOFT project.
    - Map the underlying structure of the Z1, and figure out the modifications that are needed to be done on the client's own hardware.
    - Implement a watchdog timer on the client's card, if needed.
2. Create a Direct Memory Access (DMA) based storage solution, which stores ADC measurements to a SD card.
    - Design must maintain a data-throughput of at least 2 MB/s
    - Must be able to accept an input of 48-bits as described in Heigre and Nesse's master thesis' [2] [3].
        - Must be able to easily change the bit-width in case of changes to the design.
    - A test data generator should be used to test the storage solution's functionality, e.g. a counter.
3. Create a system for transmitting of the status of the storage solution, and reception of commands from the ground, using an Ethernet to satellite connection.
    - Status of storage solution should be sent on command.
    - Continuously send status of storage solution on command.
4. If time allows, a Digital Signal Processing (DSP) solution should be made as a proof of concept.
    - This was already partially done as a part of the feasibility study.

3 Problem analysis

The storage system is supposed to be a DMA based SoC solution that is highly configurable, therefore the PYNQ ecosystem seems like a good platform to develop from. Because of the data-rate requirement of at least 2 MB/s, the system has to be built quite reliably with limited bottlenecks, this restricts the design considerably. On the input, the system has been able to accept a 48-bit data-bus on earlier versions of the project. This bit-width should be easily configurable by the user, on account of eventual changes to the design. Consequently, a fair bit of modularity has to be integrated into the system.

The system should also be able to accept three different data sources:

1. The ALOFT ADC system (Later referred to as BGO-TOP)
2. A counter, for easy testing of the storage system
3. A data feedback loop to test new High-Level Synthesis (HLS) filters with known data

Furthermore, the user should also be able to bypass the user-generated HLS hardware. This means that the system is going to need some kind of Multiplexer (MUX)-network. A general depiction of a finished system can be seen in figure 2.

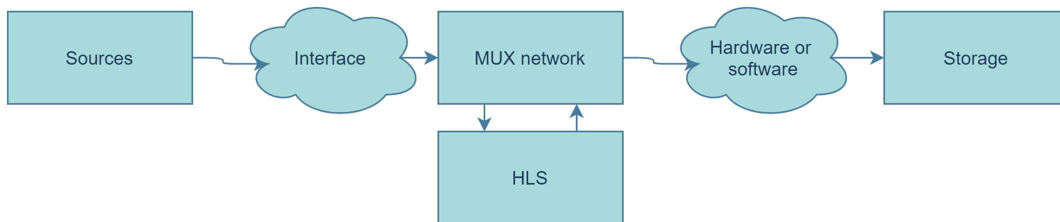


Figure 2: A general setup of the finished system

It is known that the user is going to supply some kind of data source referred to as BGO-TOP, and that the system has to be able to send that signal through a filter of some kind, most likely written in HLS. It is also known that some kind of storage device (SD card, USB drive) should be used to store the data long term. The decisions that are left boils down to what protocol the system should be based on, and if the storage process should be done in hardware or software.

3.1 Interface of the MUX-network

The MUX-network must most likely connect to a DMA controller. This means that the interface has to be converted into an Advanced eXtensible Interface (AXI)4-Stream interface at some point because Xilinx utilizes this interface on their DMA IP. AXI4 is also the standard connection between the Programable Logic (PL) and Processing System (PS) on the Z1. Therefore the question is if the system is going to utilize a standard AXI4-Stream interface on the entire system, or implement a simpler protocol to make it easier to connect supporting hardware like the HLS and BGO-TOP.

3.1.1 Parallel interface bus

A parallel interface bus would contain a data port of 48-bits, and a "new data" flag signifying if the data port has changed that clock cycle. This means that a slave receiving this flag only knows that it can read the data on the data port when the "new data" flag is high. This design's bit-width is not fixed, which means that the requirement of easily being able to expand the data-bus is upheld by this solution.

A downside of this design is the fact that the master does not know if the slave is ready to receive new data. This could lead to loss of data if the system is backlogged. This could be rectified by making the "new data" flag a dual control signal, where the master pulls the signal high when it has new data to deliver, and the slave pulls the signal low when it has received the data. This would complicate the interface considerably, and the development of accompanying hardware would become more difficult.

Eventually the interface would have to interact with the storage part of the system. This would most likely be done by down-scaling the system to a 32-bit system, since 32-bit based hardware and software are more common than 48-bit. This could be done by chopping up two 48-bit numbers into three 32-bit numbers. If the system is to be based on a DMA connection, an AXI4-Stream converter has to be added. This adds a fair bit of complexity to the embedded system, and maintenance and improvement could become quite complicated after some time.

3.1.2 AXI4-Stream

AXI4-Stream is a general-purpose point-to-point transfer protocol, generally used between a master and a slave. The protocol is a standardized and well-documented interface which makes it quite desirable for a project like ALOFT. The interface primarily utilizes three signals:

- tdata, which is a defined width data bus, generally a multiple of two, but can be other widths like 48-bits. This signal is controlled by the master of the connection.
- tvalid, which is a 1-bit signal which denotes that the tdata signal is ready to be read by being set high. This signal is also controlled by the master.
- tready, which is a 1-bit signal which denotes that the slave is ready to receive data by being set high. This signal is controlled by the slave of the connection.

A tlast signal could also be used to utilize packets in transfers. Packets are collections of transfers, and the last transfer of the packet is accompanied by a logical high on the tlast signal. An example of a packet transfer can be seen in figure 3. The AXI4-Stream standard encompasses more signals, but the four named signals are the ones utilized in this project.

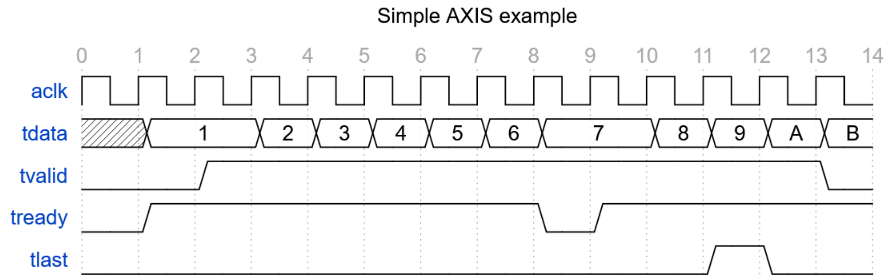


Figure 3: An example packet transfer using the AXI4-Stream interface. The packet size is set to 9 transfers in this example.

Since the source signal has a bit-width of 48 bits, and the existing Xilinx AXI4-Stream modules has bit-widths of multiples of 2 (e.g. 16, 32, 64), the data-bus has to either be up- or downscaled at some point in the system. This can either be done at entry, which means that the entire system is based on a 32-bit data width with upscalers on outputs available to the user and downscalers on the inputs. The system could also be realized with a 48-bit data-bus until the output of the MUX network. This would mean that the existing AXI4-Stream MUXes would have to be replaced by custom hardware. This would add a good amount of complexity, but there would be fewer points-of-failure, because of the lessened amount of up- and down-scaling.

3.1.3 Conclusion on choice of interface

At first, we started developing the parallel interface bus solution, but when testing the system we found that backlogging was a big problem. Therefore, we had to either develop a handshake process for the parallel interface bus solution or pivot to an AXI4-Stream interface which already includes this handshake process. Under consultation with the contractor, we decided that pivoting to a well-documented protocol like AXI4-Stream is preferred for the project over further development of the parallel interface bus solution. We had also gained quite a bit of experience on the AXI4-Stream during development of other parts of the system. The development of this solution is further described in Section 4.2.

3.2 MUX-network to storage device connections

When it comes to the choice between a software- or hardware-based connection between the MUX-network and the long term storage, there are certain criteria to fulfill. Firstly the connection has to move data quickly. Considering that a USB 2.0 thumb drive can save data at speeds between 3-10 MB/s[4] the connection should not be the bottleneck in this solution. The connection should also utilize the processor as little as possible since the processor is quite bad at transferring data. Lastly the connection should produce an OS readable file on the thumb drive. This means that the user should not need a custom reader to access the data on the file from a Windows, Mac, or Linux based computer. Primarily the connection would have to create exFAT(Windows) or ext4(Linux) based files because

of their ability to accommodate larger files (over 4 GB).

3.2.1 A purely hardware based solution

A solution based solely on hardware would indisputably be the fastest compared to a software based connection. Because of its dedicated nature, it can achieve a constant movement of data. However, this solution would also be quite complex. The solution would hinge on a connector interface (USB/SD) connected to the programmable logic, and would need a hardware based driver for this connection. This would make it impossible to change the connection later in the design without discarding this driver and creating a new one. By choosing this solution the project would pivot into creating or buying this driver as either an IP or an external chip. This driver would also have to create exFAT or ext4 based files on the drive because of the OS readability criteria.

Because of the potential speed gain of this solution it was the first avenue that was evaluated. However, after discovering the plethora of difficulties associated with this solution, primarily the difficulty of achieving the OS readability criteria and the disability of not being able to change storage method easily, this solution was scrapped in favour of a software and DMA based solution.

3.2.2 A software and DMA based solution

A software and DMA based solution means that the CPU is involved in the data transfer. This slows down the system considerably. Therefore minimizing this involvement is fundamental. One solution to this could be the utilization of a ping-pong buffer where one buffer is written to by the MUX-Network DMA. Meanwhile another buffer is read by the data-moving-DMA internally in the PS. This would only necessitate CPU intervention when the MUX-network buffer is full for exchanging buffer control. This solution would rely on the PS DMA being quicker than the MUX-network DMA because loss-of-data could occur if the write buffer is filled before the read buffer is completely read. This is accomplished as long as the mean write speed to the MUX-Network buffer is under the storage device write speed, for a USB 2.0 drive this is a maximum of 10 MB/s. This problem could also be rectified by making sure that the storage device buffer is sent to the storage buffer in its entirety before switching buffers. This would have to involve asynchronous functions, and the utilization of internal interrupts in the PS. The data paths for a solution of this type can be seen in figure 4. As it is the DMA and the internal storage connections of the PS that moves the data, the CPU does not have to be overly involved in the data moving part of the system. The CPU is rather placed in a controlling position, which it is far more adept at, and it is only involved when a buffer switch is necessary. This means that two large buffers would be far better than two smaller ones, however Z1 only contains 500 MB of DRAM, so the buffers have to be considerably smaller than half of that as to not hog the entire memory of the PS.

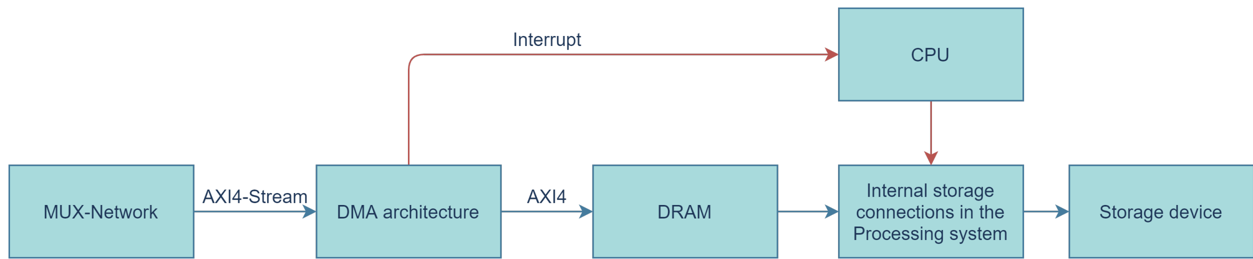


Figure 4: An abstracted overview of a software and DMA based solution.

As it is the CPU that is writing the data to the storage device it would be quite easy to make it write to an ext4 type file, which would accomplish the OS readability criteria. It would also be easy to utilize different types of connectors as it is the OS that mounts the devices. This is the only solution that is seen as a viable solution by the team in both scope and complexity, and it is therefore this design that is further described in Section 4.

4 Realization of selected solution

In this section we will describe the design as given by the specifications mentioned in Section 3, and how they are realized within the PYNQ ecosystem. We will also describe the function of our custom IPs, and how they mix with the pre-made Xilinx IPs.

4.1 PYNQ Z1

A Xilinx Diligent PYNQ Z1 board was used during the development of the storage solution for the ALOFT system. The board has a Z-7020 SoC which closely resembles the Z-7030 SoC intended for use on the ALOFT circuit board. The Z-7020 SoC is equipped with a dual-core ARM Cortex-A9 processor with an integrated Artix-7 FPGA[5]. The FPGA has a clock frequency of up to 100 MHz. The Z1 board has all the features and peripherals needed to develop the storage solution for the ALOFT system(Ethernet, USB, SD, FPGA).

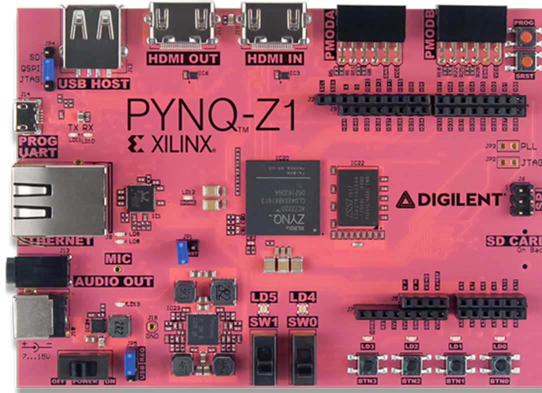


Figure 5: PYNQ Z1

4.1.1 PYNQ OS

The Z1 is a trainer board meant for an easy introduction to PYNQ. The PYNQ OS image is prebuilt for the Z1 board and has a lot of preconfigured capabilities. The board and OS image are made for ease of use for developers with little to no knowledge of the design of an embedded system. PYNQ allows for the configuration of the integrated FPGA from the OS. This is also made easier by the use of Python instead of C for embedded development. A drawback of this, is that the developer does not know the exact functionality of the Python code used, which is inherent when using Python. Another drawback is that there is a need for prebuilt overlays with the correct functionality. This means that the developer needs to build custom overlays for specific functionality. However, one of the main perks of PYNQ is rapid development and testing. It is also very easy to reuse overlays if it has the right capabilities.

4.1.2 Overlays

By the developer of PYNQ, the definition of an overlay is "post bit-stream configurable design" [6]. An overlay object is made when the desired bit-stream is loaded through a class initialization. Bit-streams are made in the Vivado design process.

Once an overlay is loaded it is given a default IP driver if the IP does not already have a driver. With the default IP driver the developer has basic functionality over the IP, such as writing to control registers. The developer may also write custom drivers for their IP.

In Figure 6, an overview of the base overlay, that is included with the PYNQ OS, can be seen. This is an overlay made by PYNQ that enables use of most of the IOs connected to the PL on the Z1 board.

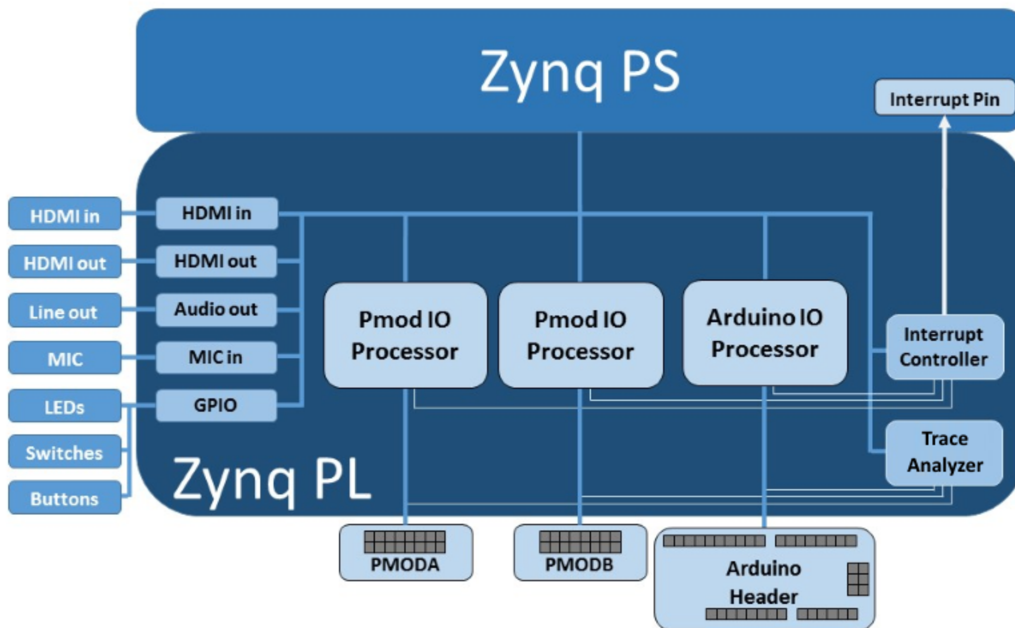


Figure 6: PYNQ Z1 base overlay

4.2 Hardware

The design of the system was mainly done on a block design level, while the design and testing of hardware modules was done on an Hardware Description Language (HDL) level. During concept design, the need for custom functionality, not native to the Vivado ecosystem arose. The functionally of custom hardware was decided in context of the entire system.

4.2.1 AXI4-Stream accommodation in the ALOFT system

Xilinx provided IPs, and the PS, have no support for 48-bit AXI4-Stream, and thus Bit Compressors are needed. Consequently, a Bit Expander is also needed for the HLS ports to make the data usable since the Bit Compressor scrambles the data, as seen in Figure 14 in Section 4.2.6.2.

Tlast is present in all custom modules of the block design, however, it is not active before the data is sent through the Packet Counter. This is because the Packet Counter's sole responsibility is to issue the tlast signal to the DMA.

4.2.2 Data flow

The selected solution results in the following layout of IP blocks as seen in Figure 7. All coloured arrows indicate AXI4-Stream and its direction. Parallel arrows share the same bus. This design provides a wide number of use cases for the user. The feedback loop from the DMA provides support for the testing of HLS modules by supplying it with known data. The path from BGO allows for raw or processed data to be stored. The Counter provides simple testing of the storage system capabilities by supplying a known input with a continuous set speed. The First In First Out (FIFO) buffer is used as a burst buffer in the data flow. This helps alleviate the CPU when it is changing between ping-pong buffers.

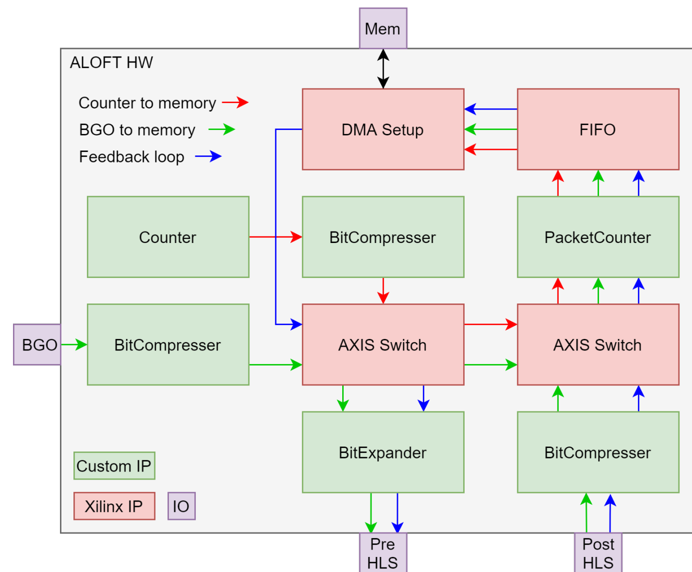


Figure 7: The several paths the data can be transferred

When the ALOFT system is deployed, it is expected that the the BGO path going through the HLS will be used. The HLS module should make the possibility of losing data less likely. The addition of the feedback loop was done to make the testing of said HLS modules easier and more accessible.

4.2.3 Control

Control and surveillance of custom hardware modules are done in a control register. An AXI General Purpose Input Output (GPIO) provided by Xilinx has been used to achieve that. By doing so, the need to use the time to develop and test custom registers was circumvented. The AXI GPIO can be written to as a normal control register in software. It is an 8-bit (can be expanded) register that is split in Vivado to address the correct hardware modules as seen in Figure 8.

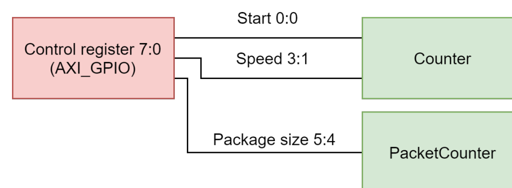


Figure 8: The configuration of several hardware settings is done using a control register

4.2.4 Interrupts

Interrupts are used to signal the CPU of an event that needs immediate attention. The ALOFT system has three different hardware interrupts, see Figure 9, that will trigger individual methods or events in software. The functionality of these will be explained in Section 4.3.

The DMA will issue an interrupt when it has completed a task (e.g. Completed buffer write). The FIFO will assert a flag if there is only one space left in the FIFO to write to. And in the current version, there is a button on the Z1 board connected as well. Chosen functionality for the button, in this solution, is to stop. However, this can easily be changed.

The signals from the FIFO and button are not interrupt signals. But to make use of them as such they are connected to an AXI GPIO. The AXI GPIO activates an interrupt signal when the connected Input Output (IO) is changed. In software, it is possible to check the origin of the interrupt signal.

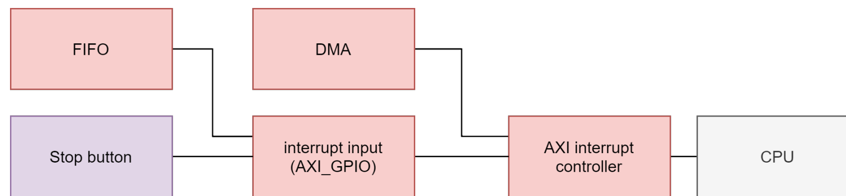


Figure 9: Interrupt connections

The AXI interrupt controller was used since the PYNQ interrupt drivers do not support interrupt signals directly connected to the CPU.

4.2.5 Xilinx IP

In the design of the ALOFT system, there was a fair bit of Xilinx IPs used. When there was a need for hardware with specific functionality, a check of Vivado's IP catalog was always preformed. This is because Xilinx IP is well documented, and the time saved on developing and testing custom hardware modules was greatly valued.

This subsection will explain the basic functionality and settings used with the Xilinx provided IPs. There are more IPs used, however these will not be discussed since they are a byproduct of the use of other IPs.

4.2.5.1 DMA

The DMA utilizes a high-performance slave port connected to the memory controller on the PS. The address register width for DMA buffer was set to the maximum of 26-bit, which results in a maximum memory mapped buffer of 67.1 MB. This is consequently the maximum buffer size the DMA can write to. However, that size was never used since allocating enough memory with continuous address space was a problem. The Xilinx DMA has scatter gather capability. This allows use of the memory with a non-continuous address space. However, scatter gather was never used since the PYNQ DMA drivers does not support this.

4.2.5.2 FIFO

The FIFO is used as a small buffer to smooth out sudden bursts and keeping the system from losing data when the DMA is getting new write instructions. Tlast and an almost full signal are enabled. Packet mode is turned off to make the almost full signal accessible.

The FIFO in the ALOFT system uses 135 KB of block memory. This is the maximum of what the FIFO can be configured with. However, adding several FIFOs in series to expand the burst buffer is possible. Note that this is almost 23% of the available 612 MB of block memory. The use of several FIFOs is further examined in Section 5.

4.2.5.3 AXI4-Stream Switch

The AXI4-Stream Switch is a more advanced form of a MUX that is capable of handling AXI4-Stream. There can be up to 16 in- and outputs. Control of the AXI4-Stream Switches is done via control registers.

In the ALOFT system, there are two AXI4-Stream Switches. One Source switch and one HLS switch. The Source switch selects between three input sources and sends chosen input to either HLS or the HLS switch. The HLS switch chooses between the HLS and the non-HLS inputs, and sends it to the packet counter. These blocks, in reference to the complete system, can be seen in Figure 7.

In Figure 10 the block design of the AXI4-Stream Switches used in the ALOFT system can be seen. The *SXX_AXIS* and *MXX_AXIS* ports are 32-bits AXI4-Stream, where the *S* indicates a slave and the *M* a master. The *S_AXI_CTRL* ports are 32-bits AXI-lite slave ports used to control the switches from PS. The *aclk* and *arstn* are clock and negative reset used by the AXI4-Stream. While the remaining ports is the clock and negative reset used by the AXI-lite control signal. In the ALOFT system both clocks and resets are connected to their same respective source.

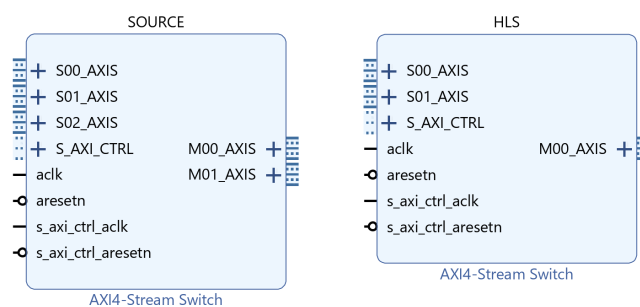


Figure 10: AXI4-Stream Switch blocks in Vivado

4.2.5.4 AXI GPIO

The AXI GPIO is used for control and surveillance of the ALOFT system. They can be configured with outputs, inputs, or both. The number of IOs can also be configured. In the ALOFT system, only

in- and outputs are used. The AXI GPIO is simpler to use with PYNQ drivers when tri-state IOs are avoided.

4.2.6 Custom hardware modules

The following hardware modules were developed using VHDL. The functionality is based on state machines. The modules were tested in Modelsim utilizing an AXI4-Stream package from UVVM light, which is supplied by Inventas [7]. It was used active low reset on the custom modules. This is common on IPs with the AXI or AXI4-Stream interfaces.

4.2.6.1 Counter

This is a counter module with a generic width AXI4-Stream output. This is 48-bit by default and has a start input signal as well as a speed select input. In Figure 11, a block diagram of the counter can be seen. The *aclk* and *arstn* are the clock and negative reset signals used in the Counter. The start input signal is a one-bit input signal that will start the counter if it has a high input, and stop it if the input is low. The speed select signal is a three-bit input signal that can choose between eight different speeds stored in a Read Only Memory (ROM) space, seen in Table 1.

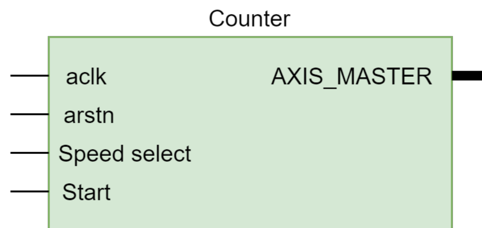


Figure 11: Counter block diagram

Speed select	0	1	2	3	4	5	6	7
Speed MB/s	1	2	3	5	8	10	20	50

Table 1: Data rates stored in ROM

This counter also has a generic value for the clock frequency the IP will experience. This value is set to 100 MHz by default, and is used to calculate the data rate.

This counter is a state machine that counts and operates the AXI4-Stream output. The frequency interval of when to send a new count is decided by a clock counter. This process can be seen in Figure 12. It is the *count* and *send count* states that operate the tvalid signal of the AXI4-Stream master.

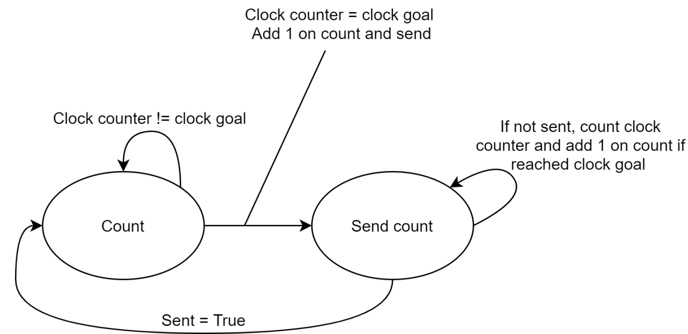


Figure 12: Counter state diagram

While the counter is in the sending state it is important to keep the clock counter running. Because, if the rest of the system is not able to receive, it will postpone the sending until the system is able to receive. This is one of the features of AXI4-Stream that the counter needs to circumvent in order to deliver simulated polling data. Otherwise, the system will never lose data on any selected speed. This will off course not be the real speed of the system. In that situation, the system speed will vary, based on the speed and utilization of the CPU.

4.2.6.2 Bit Compressor and Bit Expander

These are non-generic modules with an AXI4-Stream interface. This section will mainly discuss the Bit Compressor since the Bit Expander is very similar. It is essentially a mirror image of the Bit Compressor with the exact opposite functionality.

The Bit Compressor was made to make the interface compatible with Xilinx IPs and the PS by converting from 48- to 32-bit. While the Bit Expander was made so the data going through the HLS path would be in the intended 48-bit format when interfacing with the HLS modules. There are no settings that can be changed by the end-user when using these modules.

Figure 13 is the block diagram for the Bit Compressor. The *aclk* and *arstn* are clock and negative reset signals. The *AXIS_SLAVE* port is the 48-bits AXI4-Stream slave input while the *AXIS_MASTER* is the 32-bit output. The Bit Expander's block diagram is the same, but with a 32-bit AXI4-Stream slave input and a 48-bits AXI4-Stream master output instead.

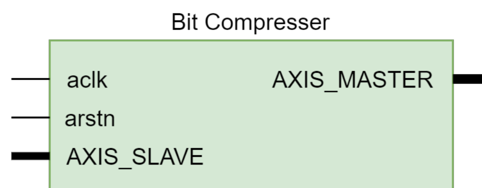


Figure 13: Bit Compressor block diagram

The Bit Compressor converts two 48 bit packets into three 32-bit packets. The two different states, *await*

and *send*, manage the slave tready and the master tvalid signals. This correlation can be seen in Figure 14. The two states changes when different conditions are met. The first state change is done when receiving the "first" 48-bit packet. Then immediately the "first" 32-bit packet is sent onward through the system. When the next 48-bit packet is received the module will send the two "last" 32-bit packets. The timings for this solution can be seen in 14.

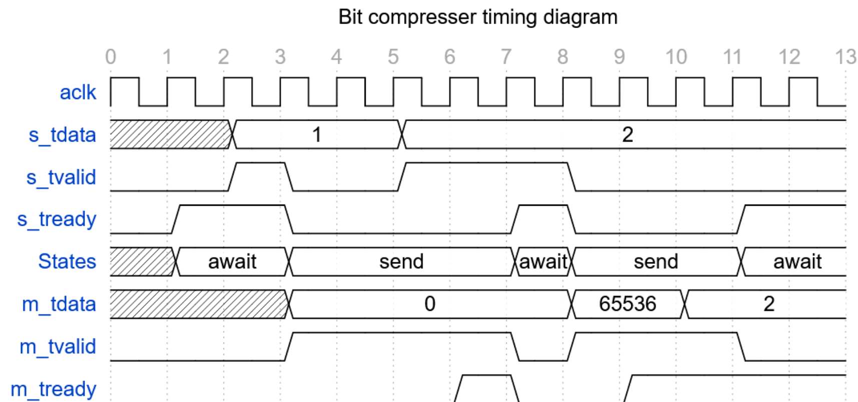


Figure 14: Bit Compressor timing diagram

This solution was decided upon to make use of the time spent waiting for the next 48-bit packet. The minimum time between each time the module can receive data, without stalling it, is only two clock cycles instead of three when waiting for both 48-bit packets before sending. Depending on if the system is able to receive. The difference in maximum speed is quite large, with improvement from 150 MB/s to 200 MB/s(When using a 100 MHz system clock). This is the main limiter of the speed in the PL. If there is a need for higher speeds into an HLS DSP it would be best to change the bit-width of the AXI4-Stream to 64-bit. This way there would be no loss in functionality of the MUX network. The maximum AXI4-Stream speed of the system is 800 MB/s if 64-bit is used.

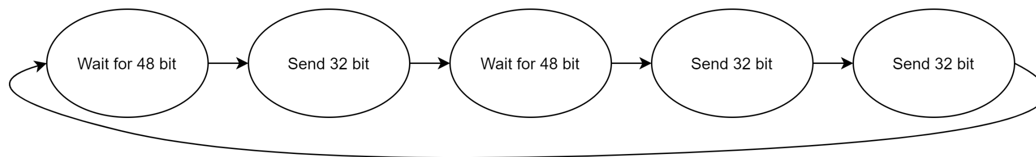


Figure 15: Bit Compressor simplified state diagram

When designing the module it was designed with only two states. One for receiving and one for sending. When making the state diagram it was decided to simplify it so it would be easier to understand. It was realized when studying Figure 15 that the VHDL code would also be easier to understand if the addition of more states was done in the first place. This is also the case of the Bit Expander which simplified state diagram can be seen in Figure 16.

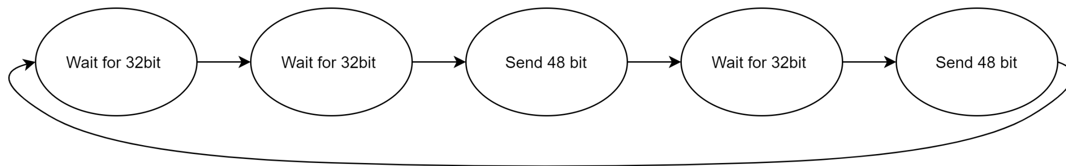


Figure 16: Bit Expander simplified state diagram

4.2.6.3 Packet Counter

This is a generic module made to apply the tlast signal to the stream when the packet size is equal to the desired buffer size. This is the buffer that the DMA writes to, as mentioned in Section 4.2.5.1. It has four different buffer sizes stored in ROM, see Table 2, that can be assigned from software. The *Packet size* input determine the buffer which is chosen. The packet number output is used by the software to note the approximate location of the buffer-address the DMA is writing to.

Packet size	0	1	2	3
32 bit packets	1024	4096	8192	10 137 600
Size KB	4	16	32	40 550

Table 2: Buffer sizes stored in ROM

The three smaller sizes have no real purpose and are only remnants from earlier revisions. They can easily be changed to match desired buffer sizes or be removed entirely. The fourth one, with a size of 40 MB, is the buffer size that sees the most use. The reasoning behind this will be explained in Section 4.3.2.

The block diagram of the Packet Counter can be seen in Figure 17. The *aclk* and *arstn* are clock and negative reset signals. *Packet size* is a two-bits input port used to choose the correct buffer size stored in ROM internally in the IP. The *Packet number* port is a 21-bits GPIO output used to read current approximately placement in the buffer. The *AXIS_SLAVE* port and the *AXIS_MASTER* port is a generic 32-bits AXI4-Stream in- and output.

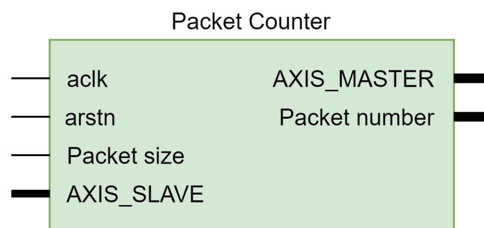


Figure 17: Packet Counter block diagram

The two different states, *await* and *send*, manage the slave tready and the master tvalid signals. This correlation can be seen in Figure 18. Each time a 32-bit packet is received it will immediately send the

packet onward through the system. A counter will add one each time the state goes from *await* to *send*. When sending, the module will check if the current packet size is equal to the set packet size. If this is true it will send tlast with the corresponding tdata and reset the current packet size.

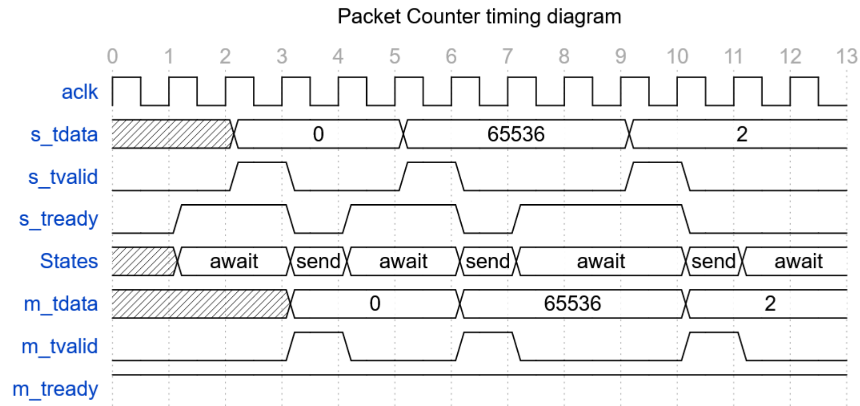


Figure 18: Packet Counter timing diagram

One of the main drawbacks of this approach is that it halves the maximum AXI4-Stream speed. Instead of being capable of sending data each clock cycle, it will only be able to send every second clock cycle.

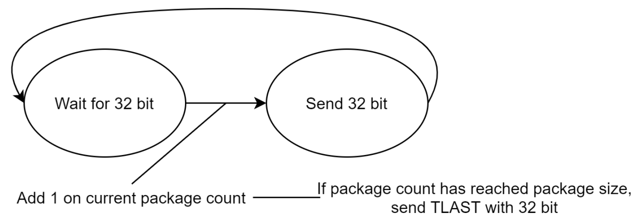


Figure 19: Packet counter state diagram

4.2.7 FPGA resource utilization

The FPGA resources used by the storage solution are minuscule as seen is Figure 20. The only real resource impact is the use of 27% block memory. There are many free resources left that can be used on additional hardware modules, such as an HLS DSP module. The resource use should not pose a problem since the impact is so small and the ALOFT system is to be implemented on an SoC with more FPGA resources available.

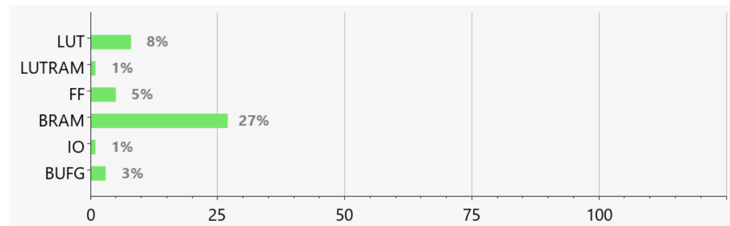


Figure 20: FPGA resource utilization (%)

4.3 Software

When making the software solution for the ALOFT system, it was decided that the drivers should be used in a class. This will make the system easily configurable while it is in use. And it would be harder to maintain smooth operation if a set of functions were used as drivers. The program where the ALOFT class is used is referred to as *main*. During development of the software we have tried to adhere to the PEP 8 Python standard [8].

This chapter will go in depth on the ALOFT class capabilities and how it works. It will explain the setup used to operate the class. There are also other features included in the solution that will be looked into.

The essential libraries used in *main* and the ALOFT class will be motioned in this chapter.

4.3.1 Configuration

The configuration of *main* can be done directly in *main*, or in a separate configuration program. This configuration program would be imported into *main*, and take its values from a *.env* enviroment file. This file could be edited during runtime. However, if the *.env* file is edited, the configuration program has to be re-imported. The configuration program solution is currently used in the ALOFT system. This is to make *main* more manageable and clear.

4.3.2 Initialization

During initialization the overlay is loaded to the class. This gives the class access to the hardware. The next steps includes setting the MUX-Network configuration. By default the signal takes the green path seen in the data flow in Figure 7 in Section 4.2.2. This configuration can be edited in the *.env* configuration file, or directly in *main*. Lastly the counter speed is configured along with the packet size. These are by default set to be respectively 8 MB/s and max buffer size (40 MB), but can also be edited in *main* or *.env*.

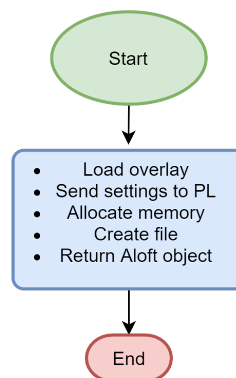


Figure 21: Flowchart for initialization

Allocating memory is necessary to make a protected space in memory where the data buffers can be located. The data type of unsigned 32-bit integer is also chosen. The allocation of memory is done by using a function available from the PYNQ library. This will create a numpy-array like array in memory [9]. There are two buffers allocated so they can be used as ping pong buffers.

The buffers are by default set to be the largest size available. The reasoning behind this choice is that it is the only useful size available in the current system. This is because, there is a greater chance of losing data when using small buffers, due to the PS not being able to keep up with the buffer switching with smaller buffers. This is caused by the FIFO storing data when the PS is making the buffer switch, and is emptying itself when the DMA is writing to memory. Consequently the FIFO does not have enough time to empty itself when using smaller buffers.

The file used to store the incoming data is created along with the class. It is set to *AloftDataPolling.bin* by default. If the chosen file name already exists, it will add an increasing number for each instance the program checks if the file name already exists.

The class uses six parameters in the initialization. Five of these are predefined. The only parameter input that is necessary is the overlay object. The five other parameters are settings for the MUX-network, data rate of the counter, and the size of the buffers used.

4.3.3 Methods

When returning the ALOFT object, *main* will now be able to use different methods and change the settings(e.g. Source, HLS, Counter speed, buffer size). The methods available in the object are used in *main* to decide the functionality of the system. Most of these methods are asynchronous. The reasoning of the use of asynchronous methods will be explained in Section 4.3.4.

4.3.3.1 Saving

The method `run_polling()` is an asynchronous method that will save all data sent to the PS. It utilizes the buffers allocated in memory as ping pong buffers. While the DMA is writing to one buffer, the CPU is saving the other buffer to file. The algorithm used can be seen in Figure 22.

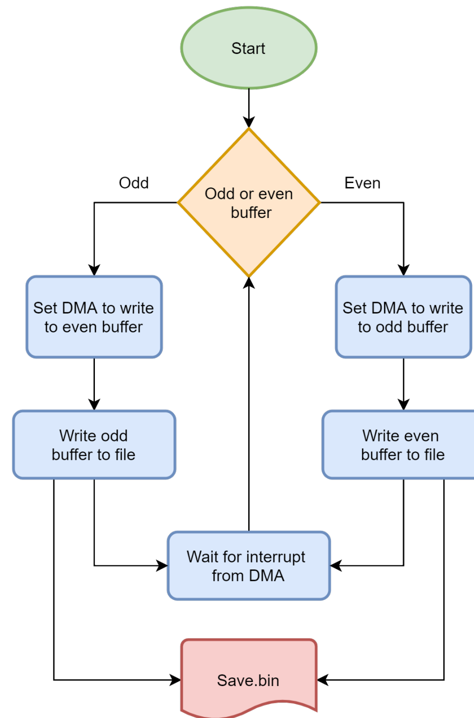


Figure 22: Flowchart for saving algorithm

When the DMA has filled one buffer, the buffer has to be exchanged immediately so that the FIFO does not fill up. Future developers has to make sure that there are as few steps as possible between the DMA giving the buffer full interrupt, and the DMA getting the location of a new buffer. Additionally it is well known that Python is not the fastest executing programming language. Even a simple addition to the code can have a huge impact.

This and the following method uses the provided DMA driver from the PYNQ library. While there are more functions in the DMA library, the ALOFT system uses only the send, receive, and wait for interrupt functions. This method is also responsible for starting the Counter if the MUX-network is configured to receive data from it.

4.3.3.2 HLS testing

`run_hls_test()` is an asynchronous method that is used to send user defined data through the HLS in the ALOFT system. Before using this method, the user needs to add data to the `hls_buffer_in` attribute in the ALOFT object. This data will then be sent through the system when the method is used.

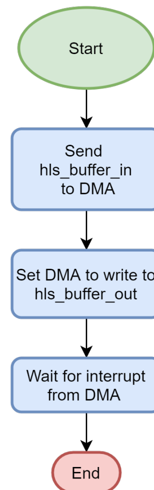


Figure 23: Flowchart for HLS test

As seen in Figure 23, there are two buffers, one input and one output. These are the same buffers already allocated when initializing the ALOFT class. It is only the input buffer that can be set, the output buffer can only be read. The user needs to use the same size and data type of the array used to set the input buffer. When employing this method, it is possible to use one of the smaller buffer sizes since there is no danger of losing any data.

4.3.3.3 Interrupts from FIFO and button

There are two asynchronous methods that handles interrupts from hardware; `full_fifo_listener` and `stop_listener`. The interrupt from the FIFO will log an error message. And the interrupt connected to the button will activate the shut down procedure(Section 4.3.3.5). Both methods can easily be changed to have other functionality that may be needed by the user.

The methods use the PYNQ provided drivers for interrupts from the AXI GPIO. When a interrupt is received, the method will read the inputs on the AXI GPIO and determine if the right conditions for the interrupt action is met.

4.3.3.4 Updating the PL settings

To update the settings while the program is running, and not schedule changes in *main*, the `update_enviroment_variables(signum, frame)` is used. This is a method activated by the SIGUSR1 signal ordered from the remote control software. For further explanation see Section 4.3.6 . The method imports the configuration program and executes the appropriate changes. The only settings that will be updated are for the MUX-network. This method is activated by the remote program, and is meant for deployment use only.

4.3.3.5 Safe shut down

The method `term(signum, frame)` is activated by the termination signal sent from the OS (e.g. `kill main`). This method is assigned in *main* and will execute the safe shut down of *main*. The most important assignment this method has, is that it will save the buffer the DMA is currently writing to. Otherwise, at worst, almost 40 MB of data could be lost. The method will read the current packet number output, and only save data up to the current writing position of the DMA. This is because the buffers will retain a value until it is written to once again.

4.3.4 The use of asynchronous programming

The use of asynchronous programming is paramount for smooth and versatile operation of the system. The system could work without asynchronous programming. However, it could only execute one operation(e.g `run_polling`). If the method was not asynchronous, it would suspend the program until an interrupt was received from the DMA, and then begin the cycle again. However, when asynchronous programming is used, Python is able to switch between which task to execute. Whenever *main* is waiting for the DMA to finish writing, *main* can check on other asynchronous methods used in *main*(e.g. `stop_listener`). A benefit of this approach is also that CPU resources are not used when *main* is doing nothing but waiting.

All of the methods have in common that interrupt is used. This indicates some sort of wait time, where the program can do other tasks or accept inputs. On the other hand, `run_hls_test` could have been synchronous (not use interrupt). But it is not. This comes from the fact that if there is more data going into the HLS than coming out, it will wait indefinitely for the "missing" data. When using asynchronous programming, *main* will be able to stop `run_hls_test` after a desired interval of time. This interval can be very short since the system is able to do an 40 MB in-out transfer in less than a second.

The asynio library is used to achieve the asynchronous functionality[10]. This is an easy to use Python library. It is however not truly asynchronous. It does not utilize several CPU threads, but is instead "juggling" between different tasks on a single thread. It can still be considered asynchronous since it is not executing the program code in sequence.

4.3.5 Logging

The logging library from Python is used for log, debug, info, warning, and error messages [11]. The ALOFT class and *main* uses the logging format defined in the configuration program. It is very easy to add logging messages in the ALOFT class and in *main*. The logging messages are saved to a file(defined in configuration) and are printed to the terminal used. This may be changed by the user.

4.3.6 Signaling

The system relies on signals, which are terminal launchable commands, to update register values and shut down the system. In Linux a signal is sent by using the *kill -signal pid* command, where signal is the type of signal to be sent, i.e. *SIGTERM* terminate, *SIGKILL* kill, *SIGUSR1/2* custom signal, and *pid* is the process id of the process which should receive the signal. These signals allows us to send software interrupts into already running python scripts like *aloftDataPolling.py* from the git repository. This is useful for the remote control software when it wants to update hardware values.

4.3.7 Remote control

The remote control software is written in C#, and utilizes the RenciSSH.NET library for communication over the network. Generally its just a set of Secure Shell (SSH) commands written as strings that is sent using the SSH library with a Graphical User Interface (GUI). Therefore, migrating this program to another programming language should not be complicated. The GUI can be seen in figure 24.

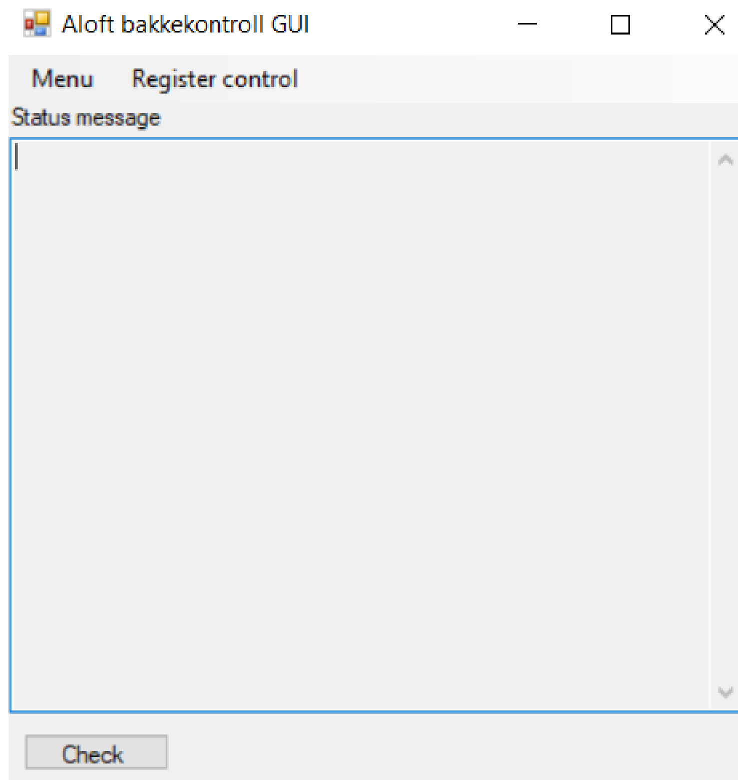


Figure 24: The GUI of the software

The GUI utilizes quite a simple setup with a ribbon menu on the top where settings can be changed, and the software could be started, stopped, or restarted. Most of the space is taken up by a Status message field where messages from the program is presented to the user, lastly there is a Check status button at the bottom. This button does not currently have any functionality because the content of a

check status message has not been decided as of yet. See chapter 6.3 for possible avenues of further improvements to this software.

The remote control software relies heavily on the signal and python-dotenv python packages, as well as the screen app for Linux to run python scripts on "terminals" which are detached from the terminal which the python script was started from. The python-dotenv package is used to edit the environment file which is used by *aloftDataPolling.py* to set the HLS and source register values. After the software has edited this file it sends a SIGUSR1 signal, which is an unassigned signal in Linux, to the *aloftDataPolling.py* program which launches the `change_enviroment_variables` function mentioned in Section 4.3.3.4.

4.3.7.1 Back-end

The program is based on a *userLogIn* class which stores the username, password and hostname of the device the software is connecting to as strings. The class has a standard constructor which sets the username and password to *xilinx*, and hostname to *aloft*, but also contains a constructor which is able to give new values to all of these variables.

The main C# form also contains links to other C# forms named *logIn*, where the user can enter new log in credentials, and *workingPathForm* which asks the user for a new path to the *.env* file on the device it is connecting to.

5 Testing

During our time at HVL, the importance and benefits of testing was made clear. At the start of our work it was decided to use testing rigorously. Due to earlier knowledge of HDL testing, the testing of custom hardware was easier. Meanwhile, the testing of the hardware system and software was done to the best of our ability. This section will go over the testing of custom hardware, the capabilities of the system, and examine possible improvements at the cost of system resources.

5.1 Testing of custom hardware

Modelsim was the software used to simulate and test the custom hardware. Prior familiarity with the software makes it easy to operate. The UVVM Light AXI4-Stream package from Inventas [7] was used with Modelsim to make the testbench development faster, and more robust. The initial setup of the UVVM Light AXI4-Stream package was a little time consuming, however when that was done it was easily transferable to other testbenches. Both manual observation of the wave diagram, and the use of UVVM logging was used to verify the functionality of the hardware. In Figure 25, an example of the UVVM log can be seen.

```
# UVVM: ID_PACKET_INITIATE      4765.0 ns  TB seq.      axistream_transmit(4B)=> 'Send 32bit Package'
# UVVM: ID_PACKET_DATA          4765.0 ns  TB seq.      axistream_transmit(4B)=> Tx x"00", byte# 0. 'Send 32bit Package'
# UVVM: ID_PACKET_DATA          4765.0 ns  TB seq.      axistream_transmit(4B)=> Tx x"00", byte# 1. 'Send 32bit Package'
# UVVM: ID_PACKET_DATA          4765.0 ns  TB seq.      axistream_transmit(4B)=> Tx x"B7", byte# 2. 'Send 32bit Package'
# UVVM: ID_PACKET_DATA          4765.0 ns  TB seq.      axistream_transmit(4B)=> Tx x"00", byte# 3. 'Send 32bit Package'
# UVVM: ID_PACKET_COMPLETE      4772.0 ns  TB seq.      axistream_transmit(4B)=> Tx DONE. 'Send 32bit Package'
# UVVM: ID_PACKET_INITIATE      4775.0 ns  TB seq.      axistream_expect(6B) while executing axistream_receive=> Receive packet. 'Recive 48bit'
# UVVM: ID_PACKET_DATA          4780.0 ns  TB seq.      axistream_expect(6B) while executing axistream_receive=> Rx x"B7" (byte# 0). 'Recive 48bit'
# UVVM: ID_PACKET_DATA          4780.0 ns  TB seq.      axistream_expect(6B) while executing axistream_receive=> Rx x"00" (byte# 1). 'Recive 48bit'
# UVVM: ID_PACKET_DATA          4780.0 ns  TB seq.      axistream_expect(6B) while executing axistream_receive=> Rx x"00" (byte# 2). 'Recive 48bit'
# UVVM: ID_PACKET_DATA          4780.0 ns  TB seq.      axistream_expect(6B) while executing axistream_receive=> Rx x"00" (byte# 3). 'Recive 48bit'
# UVVM: ID_PACKET_DATA          4780.0 ns  TB seq.      axistream_expect(6B) while executing axistream_receive=> Rx x"00" (byte# 4). 'Recive 48bit'
# UVVM: ID_PACKET_DATA          4780.0 ns  TB seq.      axistream_expect(6B) while executing axistream_receive=> Rx x"00" (byte# 5). 'Recive 48bit'
# UVVM: ID_POS_ACK              4782.0 ns  TB seq.      check_value() => OK, for std_logic '0'. 'Recive 48bit'
# UVVM: ID_POS_ACK              4782.0 ns  TB seq.      check_value() => OK, for std_logic '0'. 'Recive 48bit'
# UVVM: ID_POS_ACK              4782.0 ns  TB seq.      check_value() => OK, for std_logic '0'. 'Recive 48bit'
# UVVM: ID_POS_ACK              4782.0 ns  TB seq.      check_value() => OK, for std_logic '0'. 'Recive 48bit'
# UVVM: ID_POS_ACK              4782.0 ns  TB seq.      check_value() => OK, for std_logic '0'. 'Recive 48bit'
# UVVM: ID_POS_ACK              4782.0 ns  TB seq.      check_value() => OK, for std_logic '0'. 'Recive 48bit'
# UVVM: ID_POS_ACK              4782.0 ns  TB seq.      check_value() => OK, for std_logic '0'. 'Recive 48bit'
# UVVM: ID_POS_ACK              4782.0 ns  TB seq.      check_value() => OK, for std_logic '0'. 'Recive 48bit'
```

Figure 25: UVVM log snippet for the Bit Expander

5.2 Assessing stored data

Making sure that no data was lost when storing to the USB drive, was important when evaluating the success of the system. *ReadData.py* was developed to check if the saved counter numbers was continuous. This should have been easy, but since each 32-bit number was saved using little-endian, some extra steps had to be added to make the numbers big-endian. *ReadData.py* is not an optimized program, and therefore is very slow. However, when done, the saved data that was expected to have no data loss was indeed confirmed to have no loss.

5.3 Data throughput performance

Evaluating and testing of the entire system was used to determine if the system could deliver required data throughput. It was also used to evaluate bottlenecks in the system.

The requirement, see Section 2, of sustaining a throughput speed of at least 2 MB/s was successfully met. The maximum speed of the system is between 5 and 8 MB/s. The exact maximum speed is not found since a change in custom hardware has to be done to asses this. It was decided that the effort would not be worth it. In Table 3 failure times can be seen for different data rates. The failure times are averaged and rounded.

Speed (MB/s)	1	2	3	5	8	10	20	50
Ca. Failure time	N/A	N/A	N/A	N/A	16 min	4 min	11 sec	6 sec

Table 3: Time until failure for different data rates

In Figure 26, the resources of a short run of *AloftDataPolling.py* can be seen. The CPU usage shows clearly buffer shifts and the initialization of the system. The time between CPU spikes are used to save from buffer to USB. The transfer rate from memory to USB can be seen maxing out at around 10 MB/s. The USB transfer rate is determined as the system's main bottleneck. The reason the system is not able to store continuously at 8 or 10 MB/s, is probably because of varying transfer rates to the USB. The varying data rates to the USB may be caused by memory bus conflicts, or how the OS utilizes a DMA to transfer the data.

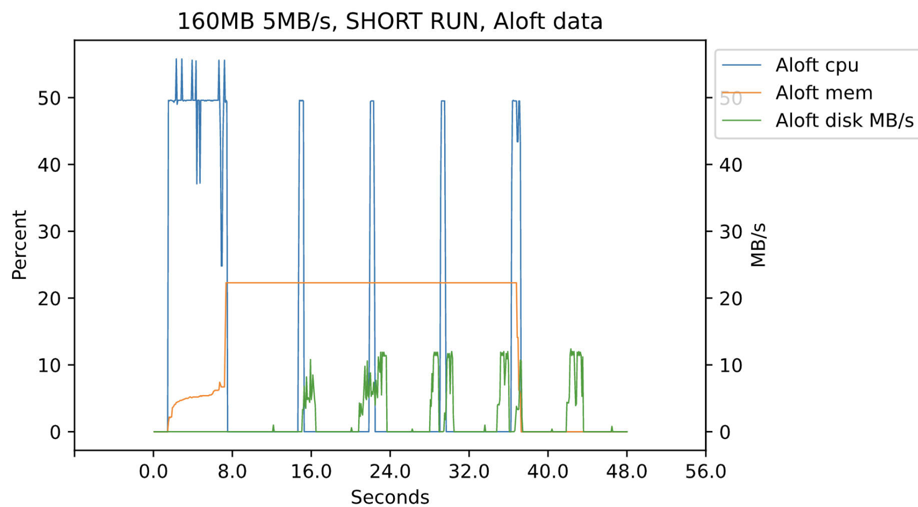


Figure 26: 5 MB/s AloftDataPolling.py resource usage

When the system was tested, without saving the buffers from memory to the USB, it managed to store 50 MB/s to memory without losing data. This indicates that the system is very adept at handling bursts of data. Additionally this is further evidence that it is the transfers to the USB that is the problem. As

seen in Figure 27, the margins become very small at higher speeds. Any disruption could result in loss of data. The loss of data for 8 MB/s occurs usually around 16 minutes into a test run. Why this is so predictable is not known. One could argue that the FIFO in the system slowly fills up and then loses data. However, the test in Section 5.4 shows that the size of the FIFO has no impact on failure times. This behavior may come down to how the OS, and its DMA, operates.

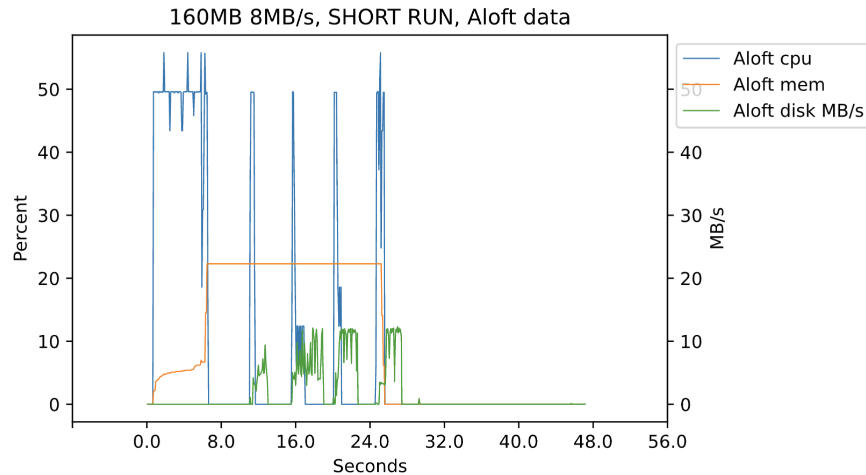


Figure 27: 8 MB/s AloftDataPolling.py resource usage

5.4 Adding FIFOs to enhance performance and alleviate bursts

When examining the causes for the bottlenecks, it was hypothesized that adding several FIFO buffers in series would enhance the performance of the system. Adding them in series was necessary since the FIFO IP provided by Xilinx was already at its maximum capacity of 135 KB. However, as seen in Table 4, the averaged and rounded failure times of three FIFOs is very close to, or the exact same as with one FIFO. This equates to no real difference, and the difference seen may be due to a small sample size.

Speed (MB/s)	8	10	20	50
Ca. Failure time, one FIFO	16 min	4 min	11 sec	6 sec
Ca. Failure time, three FIFOs	15 min	5 min	11 sec	6 sec

Table 4: Time until failure for different data rates with several FIFOs

The real benefit of several FIFOs may however be applicable if expecting large data bursts exceeding the tested 50 MB/s. The main task of the FIFO is to store data when the CPU is executing a buffer switch. Therefore adding FIFOs may make the system less prone to data loss if a large burst of data comes when this switch occurs. However, we do not know how to measure any improvements since the benefit may occur in a very short time span. Additionally the FPGA resources may be more beneficially used in a HLS filter than addressing possible edge cases such as these.

6 Discussion and further work

During the feasibility study we expected the development of the hardware and accompanying software to take 7 weeks in total. This estimate was exceeded by 4 weeks. However, during development we discovered that the Ethernet part of the system, which was planned to be on the PL, had to be developed for the PS. This was due to the Ethernet connector on the Z1 only being connected to the PS. This freed up enough time to make up for the exceeded time used on the storage system design. Other than that the project proceeded as expected.

During the late stages of the design cycle potential additions to the design were discussed. These improvements has to be done at a later stage.

6.1 Further development of data monitoring

Currently there is no way of knowing if any data is lost in the PL part of the system before the FIFO. Therefore, a monitor is needed in order to see if all the data that is put into the system also arrives at the other end. This could be done with a counter, or a more complex bit of logic.

6.2 Last received data-point before overflow

Presently the hardware only gives a message to the software if the data-buffers are filled. This consequently generates an error flag. However, there is no way of knowing where the lost data should have been in the dataset. Future versions of the design could implement a last-message-before-lost-data bit in the last message sent to the buffer. This would, however, claim one bit in every 32 bit message which is quite a bit of lost datarate. Future versions could instead implement a last data register which is readable by the software. The software could then log this data, or try to find it in the dataset, and input a newline symbol in the storage file.

6.3 Remote control functionality

Currently the remote control software is in a state where it is only able to give orders to the storage system using signals and SSH. This could be further developed to use SSH login during operation which would be safer.

The software also needs to be able to check the status of the storage system, which is currently not implemented. This could be achieved by creating a Python script which gathers data from other parts of the system, and compiles this data in a status message. This message could contain the number of *errors* and *warnings* the system has produced, how many buffers which are written, and when the last buffer was written. The content of such a status message is not decided as of yet.

6.4 A faster storage interface

During testing of the system it was discovered that the system could at least handle an average data throughput at up to 5MB/s continuously as mentioned in Section 5.3. However, any more than that would cause loss off data after some time of sustained load. It was hypothesized that this fault was because of the USB 2.0 interface which the data had to traverse in order to store data on the USB drive. This is because any USB 2.0 connection can not handle write speeds at more than between 3 and 10 MB/s [4]. Therefore it is suspected that a faster connection could speed up the system considerably.

One such inteface which is available on the Zynq-7000 series [12], which ALOFT utilizes, is a SD High Speed (SDHS) connection [13]. This interface is capable of transfers of up to 10 MB/s according to the SD standard[14]. The implementation of this interface on future SoC boards could potentially double the data throughput if the storage device interface is the true speed bump in our system.

Additionally there is also a Peripheral Component Interconnect Express (PCIE) Gen 2 x4 interface connected to the PL on certain Zynq-7000 series chips [12]. This could theoretically have a bandwidth of 4 Gb/s in full duplex according to a white paper published by Xilinx [15]. However, this solution would require a PCIE2x4 compatible storage device, and quite a bit of work to get going.

Before any of these interfaces are properly implemented the minimum mean speed has to be decided on, and further research into the low speed between the memory and the storage device has to be researched.

6.5 Adapting to the ALOFT circuit board

When adapting this system from the Z1 to the ALOFT circuit board a few steps has to be completed.

- A hardware board file of the ALOFT circuit board has to be created for the Vivado design. This entails a complete rebuild of the hardware following the steps laid out in Appendix F with the new board file.
- As mentioned in Appendix E a PYNQ system image has to be adapted to the ALOFT circuit board.
- A reset button has to be added to the ALOFT circuit board. This button could also be scrapped as mentioned in Appendix F.

6.6 Known software bugs

- When executing the `term()` function from the ALOFT class the system does not always close as expected. The system can not save to the *.bin* file, and produces an *FIFO full* error before continuing running the program. This is also a problem when stopping the software from the ground control software, but it was rectified by sending *newline* 10 times after sending the stop order. This is not a satisfactory fix, and should be further investigated.

- Due to PYNQ OS being little-endian the 32-bit numbers which are received in the system memory is flipped when written to the *.bin* file on the USB drive. This makes it harder to read the coupled 48-bit numbers afterwards, and a procedure to store the numbers the right way should be developed.

7 Conclusion

During the last few months we have created a system defined by the requirements set in Section 2, and researched the adoption of PYNQ for the contractors. During this time we have reached a few conclusions regarding these topics.

7.1 The use of PYNQ for further development of the project.

PYNQ is a versatile and easy OS to use in an embedded system. The utilization of Python as the standard programming language makes it easier to step into the software side of embedded development. In conjunction with the premade hardware drivers it is quite easy to set up a simple hardware accelerated program in just a manner of hours with little knowledge of embedded systems. However, this simplicity also introduces a few difficulties. Since Python is quite a high level programming language a lot of the intricacies is abstracted away in the code, which makes tasks such as debugging harder for the developer. Conversely the main programming language taught to the students who will utilize this system is mainly Python. So a Python implemented design would make the system easier to handle for the users.

All things considered PYNQ is a fully fledged Linux based operating system which performs the tasks it is given admirably in an embedded, Python based, context.

7.2 The ALOFT design

A system is created that reliably can receive data from an, at this point, undecided source. This data can be received reliably at a rate of 5MB/s in the current 48-bit-to-32-bit setup. Further testing has to be performed to check the data throughput of other setups. Lastly the system offers quite a bit of control to the user. The user is able to select from 3 different sources:

- A counter which supplies the system with a constant flow of data. It is also possible for the user to toggle this counter on and off. The byterate of the constant flow of data is also able to be changed during runtime between 8 different preset rates.
- A data feedback loop which reads data from system memory, and feeds it back into the system. This is meant for testing of the HLS, where you want to check the output against a known input which is not a counter.
- The aforementioned undecided source which is the real data input of the system.

Accompanying this design there has been created a program which is able to connect with the system over SSH. This program is capable of starting, stopping and restarting the system. Additionally it is able to restart the hardware. The program is also capable of changing the data source, and if the data should enter the HLS block during runtime.

Appendix A References

References

[1] UiB, “Mikroelekronikkgruppen ved institutt for fysikk og teknologi ved universitetet i bergen,” University of Bergen, 2020. [Online]. Available: https://www.uib.no/ift/132660/mikroelektronikk-ved-ift-%E2%80%93-fra-cern-til-verdensrommet

[2] M. F. Heigre, “Design of an embedded readout system for the aloft gamma-ray detector instrument,” Master’s thesis, University of Bergen, 2018. [Online]. Available: https://hdl.handle.net/1956/18671

[3] A. N. Nesse, “Soc design of electronic readout system for aloft,” Master’s thesis, University of Bergen, 2018.

[4] How fast is your flash drive? read write speeds of USB flash drives! [Online]. Available: https://www.flashbay.com/support/faq/usb-flash-drive-read-write-speed

[5] Xilinx, “Zynq 7000 soc family overview,” Xilinx, 2019. [Online]. Available: https://www.xilinx.com/support/documentation/selection-guides/zynq-7000-product-selection-guide.pdf

[6] ——, “Overlay source code,” Xilinx, 2021. [Online]. Available: https://github.com/Xilinx/PYNQ/blob/master/pynq/overlay.py

[7] Inventas, “Uvvm light,” Inventas (formerly Bitvis), 2021. [Online]. Available: https://github.com/UVVM/UVVM_Light

[8] G. v. Rossum, B. Warsaw, and N. Coghlan. PEP 8 – style guide for python code. [Online]. Available: https://www.python.org/dev/peps/pep-0008/

[9] NumPy, “Numpy array,” NumPy, 2021. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.array.html

[10] Pyhton, “Asyncio,” Python, 2021. [Online]. Available: https://docs.python.org/3/library/asyncio.html

[11] ——, “Logging,” Python, 2021. [Online]. Available: https://docs.python.org/3/library/logging.html

[12] Xilinx, “Zynq-7000 all programmable SoC and 7 series devices memory interface solutions v4.2 data sheet (v4.2).” [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/mig_7series/v4_2/ds176_7Series_MIS.pdf

[13] ——, “Zynq-7000 SoC technical reference manual (v1.13).” [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[14] SD standard overview | SD association. [Online]. Available: https://www.sdcard.org/developers/sd-standard-overview/

[15] J. Lawley, "Understanding performance of PCI express systems," p. 16.

Appendix B Acronyms

Acronyms

AXIS AXI4-Stream. iv, vii, viii, xx–xxii, xxiv, 4–6, 11, 14–19, 28

ADC Analog to Digital Converter. iii, 3, 4

ALOFT Airborne Lightning Obsevatory for FEGS & TGFs. ii, iii, v, vii, 1–5, 32, 34

AXI Advanced eXtensible Interface. iii, 4, 12–15, 24

BGO Bismuth Germanate Oxide. 1

CERN Conseil européen pour la recherche nucléaire. 1

CPU Central Processing Unit. iii, 7, 8, 11

DMA Direct Memory Access. iv, vii, 3–5, 7, 8

DRAM Dynamic Random Access Memory. 7

DSP Digital Signal Processing. 3

ESA European Space Agency. 1

FEGS Flys Eye GLM Simulator. 1

FIFO First In First Out. 11, 13

FPGA Field Programmable Gate Array. iv, 1, 2, 20

GPIO General Purpose Input Output. vi, viii, xxii, xxiii, xxv, 12–15, 18, 24

GUI Graphical User Interface. 26

HDL Hardware Description Language. 11, 28

HLS High-Level Synthesis. xxiv, 4

HVL Western Norway University of Applied Sciences. 28

IO Input Output. xxii, 10, 13–15

IP Intellectual Property. iii

MUX Multiplexer. iv, 4, 6, 7

NASA National Aeronautics and Space Administration. vii, 1

OS Operating System. iii, viii, 2, 3, 6–8, 33, 34

PCIE Peripheral Component Interconnect Express. 32

PL Programable Logic. 4, 31, 32

PS Processing System. iii, 4, 7, 31

PYNQ Python productivity for Zynq. iii, viii, 2–4, 33, 34

ROM Read Only Memory. 15

RTOS Real Time Operating System. iii

SD Secure Digital. iv, x, 3, 4, 7, 32

SDHS SD High Speed. 32

SoC System on a Chip. 1, 32

SSH Secure Shell. viii, 26, 31, 34

UiB University of Bergen. 1

USB Universal Serial Bus. iii, x, xii, 4, 6, 7, 32, 33

VHDL VHSIC Hardware Description Language. 2

Z1 PYNQ-Z1 Circuit board. iii, 2–4, 7, 31, 32

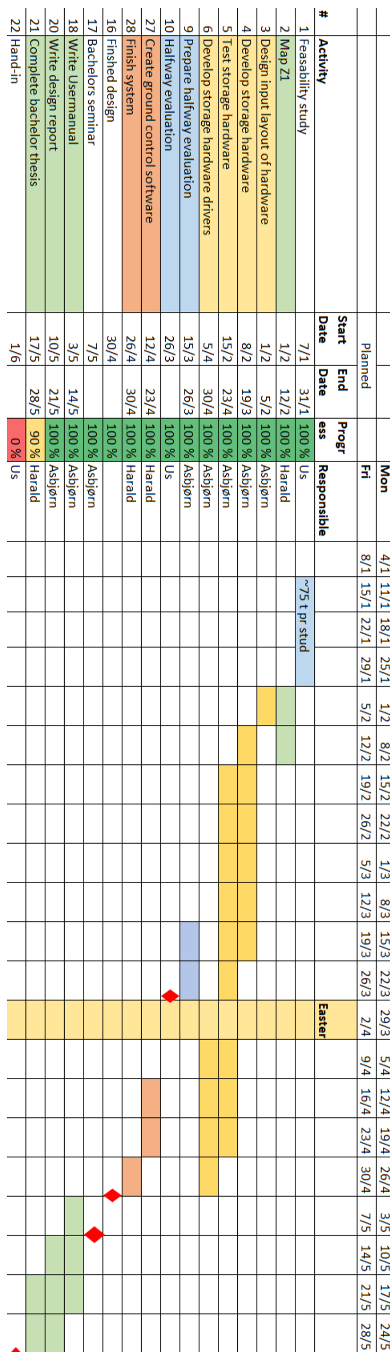
Appendix C Project management

C.1 Project organization

Early on in the project it was intended to have a changing project lead based on what part of the design we were working on. However, this became more loose during the project, and we decided to rather split work between us without a project lead. This worked well seeing as we were only two participants on this project, and we both knew what needed to be done. Frequent internal meetings also lessened the need for a project lead. The communication with the supervisor and contractor were split from the offset. This meant that Harald took charge of communication with the contractors, while Asbjørn communicated with the supervisor.

C.2 Project form

Initially it was decided that the project should have a linear approach. This entailed that we should finish the design of one part of the system before engaging in a new part of the design. This was somewhat upheld until the final weeks of the design where we split our effort into different parts. Asbjørn primarily took charge of the Aloft class, accompanying software and the finishing of the digital design. Meanwhile Harald began designing the ground control software from scratch. This can be seen in Figure 28.



#	Activity	Start Date	End Date	Progress	Responsible
1	Feasability study	7/1	31/1	100 %	Us
2	Map Z1	1/2	12/2	100 %	Harald
3	Design input layout of hardware	1/2	5/2	100 %	Asbjørn
4	Develop storage hardware	8/2	19/3	100 %	Asbjørn
5	Test storage hardware	15/2	23/4	100 %	Asbjørn
6	Develop storage hardware drivers	5/4	30/4	100 %	Asbjørn
9	Prepare halfway evaluation	15/3	26/3	100 %	Asbjørn
10	Halfway evaluation	26/3		100 %	Us
27	Create ground control software	12/4	23/4	100 %	Harald
28	Finish system	26/4	30/4	100 %	Harald
16	Finished design	30/4		100 %	
17	Bachelors seminar	7/5		100 %	Asbjørn
18	Write Usermanual	3/5	14/5	100 %	Asbjørn
20	Write design report	10/5	21/5	100 %	Asbjørn
21	Complete bachelor thesis	17/5	28/5	90 %	Harald
22	Hand-in	1/6		0 %	Us

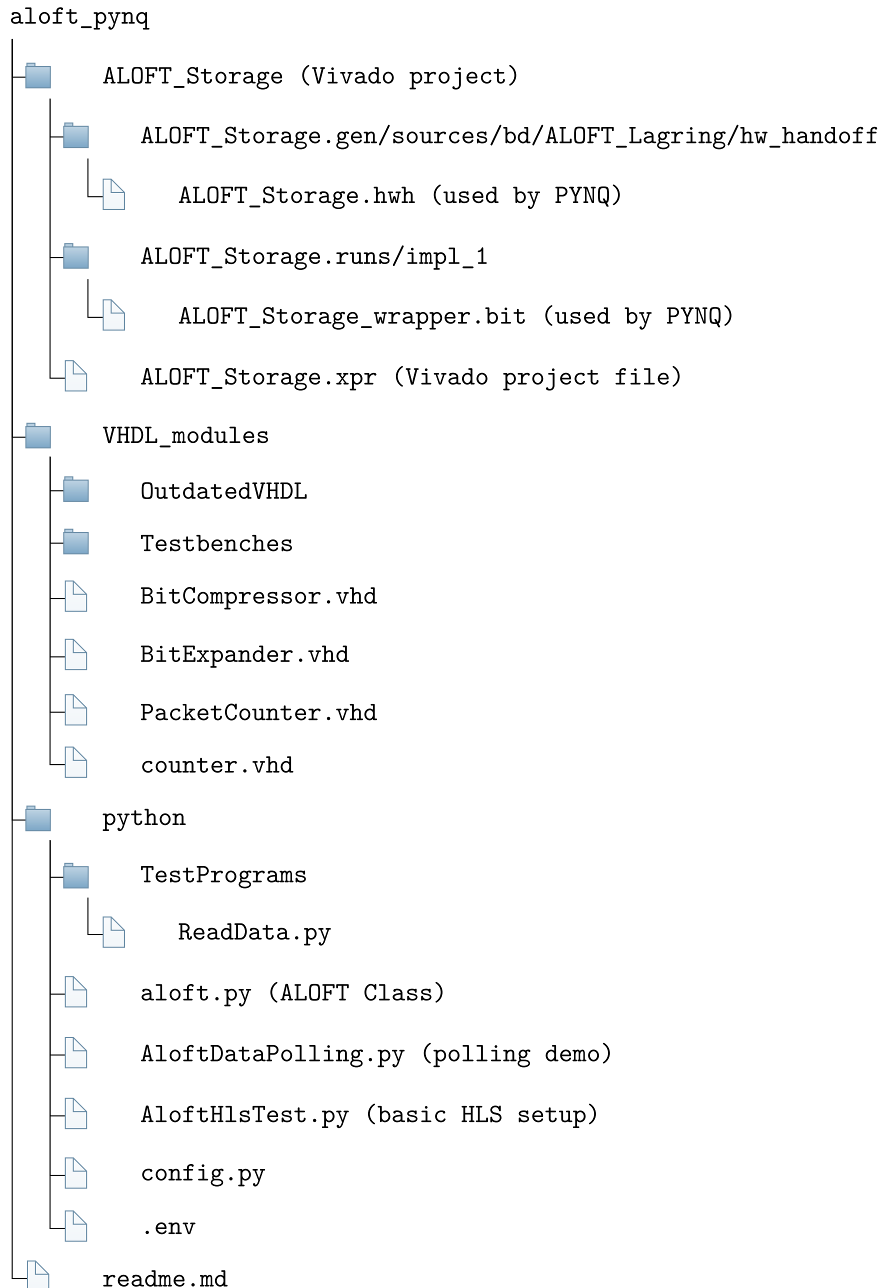
Figure 28: Gantt scheme

Appendix D Git Repository

The following directory tree representing the git repository used in this project was used in sharing and development of the ALOFT storage system. The tree only shows files referred to in this thesis and in the user manuals.

Located at :

```
https://git.app.uib.no/master/aloft_pynq.git
```



Appendix E User manual

This user manual will give an understanding of basic use of the storage solution for the ALOFT system. For use of the ALOFT system, in its current state, a PYNQ-Z1 circuit board is required. The user should also preferably have the Vivado design suite installed. However this is not necessary unless the user wants to change or add hardware.

It is also required to use some sort of SSH enabled terminal, a text editor, and a file explorer during setup and maintenance of the system. During the development of the current build Visual Studio code has been used to accomplish all of these tasks, with its internal file explorer and terminal. Lastly some sort of disk imager is needed to flash the PYNQ Operating System to a SD card.

Section E.1 is only useful if the user wants to add custom hardware to the project (eg. HLS, DSP, BGO TOP). If this is not desired, simply jump to Section E.2 .

Section E.1 assumes the user has some prior knowledge and experience with Vivado. It is to be advised, if the user has little to no knowledge in Vivado, to try the process of building the hardware themselves by following the guide in Appendix F. Note that this project possibly needs to be built from scratch when transitioning to another board.

E.1 Building the hardware in Vivado

Launch Vivado, preferably the 2020.2 version of Vivado. However, the project should be easily forwarded to a new version of Vivado since we have opted to the sharing of all the Vivado project files. Now launch the project that is located in:

```
<aloft_pynq repository>/ALOFT_Storage
```

Note that some hardware modules that are in use by the project are located in the folder listed below. The project will not be able to synthesise if these are missing.

```
<aloft_pynq repository>/VHDL_modules
```

Now add the desired hardware in the block design, for example a FIR filter. Remember that the HLS connections are AXI4-Stream 48 bit. The input for this needs to be connected to `PRE_HLS` and the output to `POST_HLS`. Save the block design and generate HDL wrapper. When that is done, simply hit *Generate Bitstream* and wait for it to finish.

E.2 Setup of the PYNQ-Z1 board

During the following setup of the board the username and password should be set to *xilinx*, which is standard, and the Hostname should be aloft. Otherwise the remote control software will not work as easily as intended.

To setup the Z1 board please go to the website listed below and follow the instructions for basic setup. Guides for several boards are located there including a guide on how to build an image for other

boards.

```
https://pynq.readthedocs.io/en/latest/getting_started.html
```

The correct image for the Z1 board is located at:

```
https://github.com/Xilinx/PYNQ/releases
```

When basic setup is completed the user is advised to open a connection via Samba, for file access and editing, and SSH for terminal control. This could be done in Visual Studio Code, or any other combination of setups.

After setup the user has to give certain privileges to the xilinx user. This is to make the remote control software usable. Firstly the user has to get root access. Open up your terminal, log in to the xilinx user, and enter:

```
su                                  #To enter root
usermod -G root xilinx              #To give xilinx root access
```

After this the user has to gain a few root privileges. This is done by writing *visudo* while still in the root terminal, and pressing *ENTER*. In the *visudo* file the user has to enter two lines under *User privilege specification*:

```
xilinx ALL=(ALL) ALL
xilinx ALL=NOPASSWD: /sbin/shutdown
```

Afterwards you can exit the file by in succession pressing: `Ctrl + X`, y, and `ENTER`

After this, reboot the board:

```
sudo shutdown -r now
```

Lastly the user has to install the screen app, and dotenv module using the following commands in the user terminal:

```
sudo apt install screen
sudo pip install python-dotenv
```

These packages are used to run Python scripts without having a connected terminal, and editing the *.env* file from the command line.

Note: It is not advised to use Jupyter or VS code SSH file transfer when running the storage system. Those options require more memory and the user will end up needing to restart the board after a little while to be able to allocate the required memory of the storage system. Using SSH in the VS code integrated terminal is fine.

E.3 Running the demos

Transfer the python folder located in the repository to an appropriate space on the Z1 board. Preferably this space should be in the xilinx folder.

```
<aloft_pynq repository>/python
```

The user will need to transfer the following files from the Vivado project to the Z1 board.

```
<aloft_pynq repository>/ALOFT_Storage/ALOFT_Storage.gen/sources/bd/
    ALOFT_Storage/hw_handoff/ALOFT_Storage.hwh
<aloft_pynq repository>/ALOFT_Storage/ALOFT_Storage.runs/impl_1/
    ALOFT_Storage_wrapper.bit
```

It is recommended to create a folder to contain the files, and save this folder in PYNQ's overlays folder. The user will also need to remove *_wrapper* from the name of the *.bit* file. The *.bit* file and the *.hwh* file will need to have the same name. The overlays folder is located at:

```
/home/xilinx/pynq/overlays
```

The last step for the setup of the demos is to replace the default path of the *.bit* file in the *.env* file that is located in the python folder. In the *.env* file the basic configuration settings can be found.

E.3.1 AloftDataPolling.py

To run the demo, simply use the following command after all the necessary files and file changes are in place.

```
sudo python3 AloftDataPolling.py
```

This demo is a simple test of storing data that is generated from a counter in hardware. The demo allows for the user to set changes to the run in the *main* function. By default the *main* function starts by setting the count speed to *20 MB/s*, waiting for two seconds, then switching the count speed to *8 MB/s*. There are eight different speeds to choose from. See Section E.5 for more information and for parameters and attributes that can be changed.

Note that the path to a data storage file will have to be set by the user in the *.env* file. This storage file has to be a *.bin* file, and it is adviced to have this file on a storage device like an USB drive or SD card. These devices has to be mounted by the user using the *mount* command beforehand.

Stop the program by using the remote control program or by using the command below. The user may also use `CTRL+C`, however the current buffer that the DMA is writing to will not be saved.

```
sudo kill <pid>
```

Note that the system is limited to lower write speeds continuous. The user will get an error if there is any data loss. See chapter 8 *Testing* for more information.

The user can also use this program as a base build for saving data from BGO TOP.

E.3.2 AloftHlsTest.py

This demo was created to show the software setup of how to test HLS modules made by the user. The user will need to set input data that will be processed by the HLS. The demo shows how to do this

by creating an array with ones and sending it through the system's HLS path. Instead of ones, in the `in_buffer` array, the user should put desired data in the array. After it is gone through the system, the processed data will be copied to a new array. The user will need to add custom functionality to analyze the processed data. Run the demo by using the following command.

```
sudo python3 AloftHlsTest.py
```

Note that the size of the arrays is fixed to four different sizes. Three small arrays and one large. The large buffer, with the size of *10 137 600* 32 bit numbers, is the only one that will go through the system smoothly. However this depends on the output data array (processed) is of the same size as the input data array. Otherwise the system will not stop listening to receive more data. This causes an inconvenience of having to shut down the program manually. There will also be some zeros in the output array since those spots were never written to. More on the aforementioned array sizes are found in E.5.2.

E.4 Using the Aloft class

This subsection will go in to greater detail on the properties and capabilities of the Python class made for the ALOFT system. It is important to read this if the user wants make a program from scratch. Remember to note which methods uses asyncio.

E.4.1 Initializing

The only input parameter that is not predefined and is necessary to define is the overlay object. The user may change predefined parameters when initializing the class. Read PYNQ documentation for more information on overlay. This class is only functional on the given overlay or on other overlays with the exact same structure and naming. The user will get a lot of errors otherwise.

Example of complete simple polling program.

```
import asyncio
from pynq import Overlay
from aloft import Aloft

overlay = Overlay("overlays/ALOFT_Storage.bit")

aloft = Aloft(overlay)

loop = asyncio.get_event_loop()
tasks = [
    asyncio.ensure_future(aloft.full_fifo_listener()),
    asyncio.ensure_future(aloft.run_polling())]
loop.run_until_complete(asyncio.gather(*tasks))
loop.close()
```

Adding asynchronous capabilities is also necessary when using asynchronous methods from the Aloft class.

E.4.2 Predefined parameters

There are a set of predefined parameters in the Aloft class. These may be changed either when initializing the class, or in the case of the demos in the *.env* file.

```
str file_path = "AloftDataPolling.bin"
```

By default the data storage file will be saved in the local directory. The user is free to change this to save anywhere on the system.(Like a mounted USB drive.)

```
int source = 1
```

The source parameter sets where to take input data. By default this is BGO TOP. The feedback loop is used to test HLS modules.

Value	0	1	2
Source	Counter	BGO TOP	Feedback loop

Table 5: Data source options

```
bool hls = False
```

Select if the system should use HLS module to process data before writing to memory buffer.

```
int pckt_cnt = 3
```

Sets the size of the memory buffer. User able to choose from four fixed values.

Value	0	1	2	3
32 bit packets	1024	4096	8192	10 137 600
Size KB	4	16	32	40 550

Table 6: Buffer size options

Note that the user should never change *pcktSize* unless they are using the system for HLS testing. In other usage, the system will not be able to save fast enough to avoid losing data. The user will get an error.

```
int cnt_speed = 4
```

Sets the counting speed of the counter so it will generate the right amount of data per second. The user is able to choose from eight fixed values.

Value	0	1	2	3	4	5	6	7
Speed MB/s	1	2	3	5	8	10	20	50

Table 7: Data rate options

E.4.3 Methods

All of these methods are asynchronous, except the *Term* method, and needs to be added in the program as tasks to the event loop when in use. Them being asynchronous is paramount for the functionality of the system. Example on where to add the method can be found is in E.5.1.

```
async def run_polling()
```

When added this method will save data from the given source. It will run until the program is shut down.

```
async def run_hls_test()
```

Sends memory buffer through the feedback loop and writes received data to another memory buffer. The memory buffer used to send data can be set by setting the hlsBufferIn attribute. The other can be read from the hlsBufferOut attribute.

```
async def full_fifo_listener()
```

Using this method will make the user able to monitor if the system is losing data. It will give an error if that situation arises.

```
async def stop_listener()
```

The program will stop if the system receives a stop interrupt from the system. In current version of the project the stop signal is activated by button 0 on the Z1 board. This can be changed in Vivado to fit the users needs, or be removed.

```
def term(signum, frame)
```

This is a method used by software signaling to shut down the program so it will not lose data already written to memory buffer. This method should only be called by signals. To use this add the following to the program. After that the method will be called upon when terminating the program.

```
import signal
signal.signal(signal.SIGTERM, aloft.term)
```

```
def update_enviroment_variables(self, signum, frame)
```

This method is started by the *SIGUSR1* signal to change the *HLS* and *SOURCE* values when the .env file is called. The following has to be added to the program to use this function.

```
import signal (Does not need to be added if term() is used already)
signal.signal(signal.SIGUSR1, aloft.update_environment_variables)
```

E.4.4 Attributes

```
numpy array uint32 hls_buffer_out
```

Returns output buffer when using feedback loop. This attribute can not be set.

```
numpy array uint32 hls_buffer_in
```

Returns input buffer when using feedback loop.

```
str file_path
```

Returns file path and name of save file. This attribute can not be set.

```
int source
```

Returns current system source.

```
bool hls
```

Returns HLS usage.

```
int pckt_size
```

Returns buffer size. Can be set lower but not higher than current packet size.

```
int cnt_speed
```

Returns counting speed of the counter.

Appendix F Building the hardware from scratch

To follow this guide, the user will need access to Vivado. Preferably the 2020.2 version of the software. Additionally, to have any benefit of this system, the user must have a PYNQ Z1 circuit board at hand. Consequently the need to download and install the Z1 board files to Vivado arises. Board files, and instruction on installation, can be found on the following website:

```
https://pynq.readthedocs.io/en/v2.6.1/overlay_design_methodology/
    board_settings.html
```

If you are using a custom board, you have to supply the board files yourself. If the user is stuck or has a problem, it is advised to review the Vivado project available in the Git repository.

F.1 Creating the Vivado project

The first step in building the hardware is to create a Vivado project. When Vivado is opened, click the *Create Project* option in the quick start menu. The first step in this wizard is to choose a name for the project, and an appropriate folder for the location of the project. The second step is to select the project type; choose the *RTL* option. Third, is to add design sources to the project. The user will need to add the hardware modules located in the git repository.

```
<aloft_pynq repository>/VHDL_modules
```

In the fourth step Vivado asks for constraints files to add to the project; skip this step. Lastly the user is asked to choose the correct Xilinx product. Navigate to the board tab and choose the PYNQ Z1 board. The board should be in the list if it was added correctly. When finishing the creation wizard, a *New Project Summary*, as seen in Figure 29, should appear. Make sure the settings match in your summary and click the finish button.

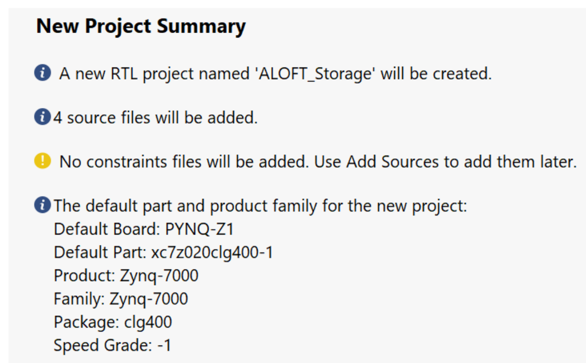


Figure 29: New project summary

When the project is finished loading you are almost ready to start building the system. To build the system you will need a block design. Create a block design by selecting that option form the *Flow*

Navigator on the left hand side. Choose an appropriate name for the block design. When the block design is created, the work area of Vivado should look similar to Figure 30.

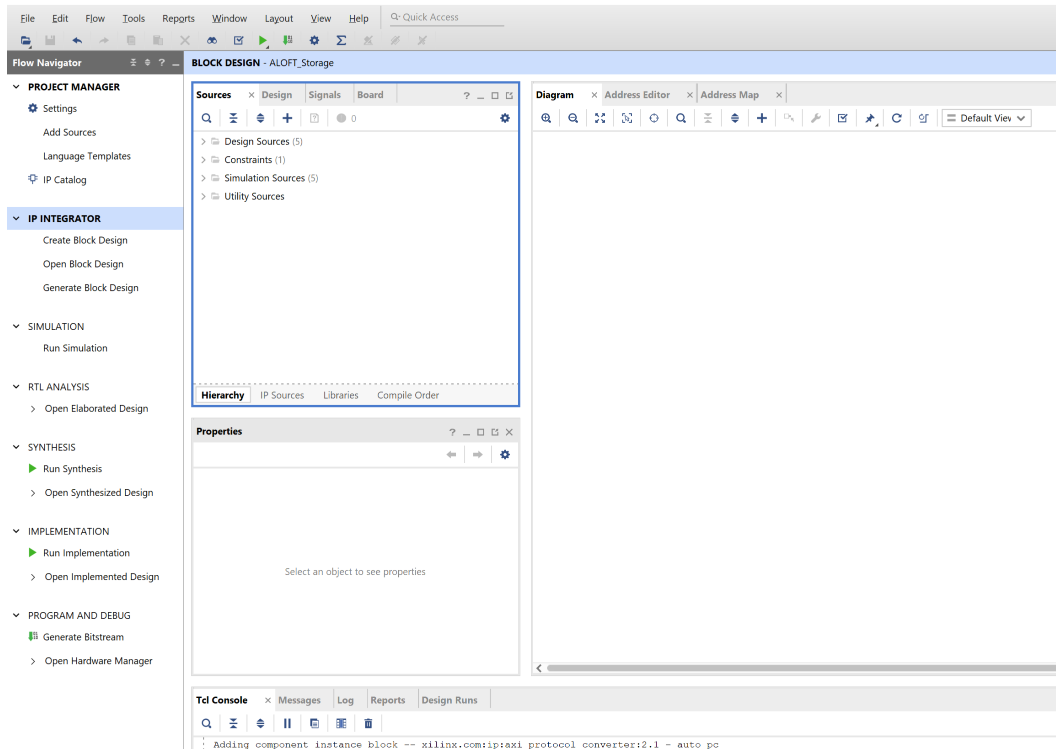


Figure 30: Vivado working area

F.2 Adding and connecting IPs

Now it is time to add IPs to the block diagram. Every Xilinx IP can be added by clicking the plus sign in the diagram toolbar, and searching for the correct IP. When adding an IP, it comes with a default name. The user is free to choose names on the IPs, however when this guide specifies a certain name change(e.g. "set the DMA name to *dataDMA*"), it is very important to follow said name changes. Name changes are done in the *Properties* window seen in Figure 30. The software made for the ALOFT system is dependent on these names. If changed, the user will have to change the corresponding names in the ALOFT class.

F.2.1 Zynq 7 Processing System

The first IP to add is the *Zynq 7 Processing System*. This is the processing system already located on the 7020 SoC. When the IP is added a *Run Block Automation* option should appear in a green bar beneath the diagram toolbar. Click this option and apply board preset. There should be no need to change any of the already applied settings. When this is done, the Zynq 7 IP will look like Figure 31.

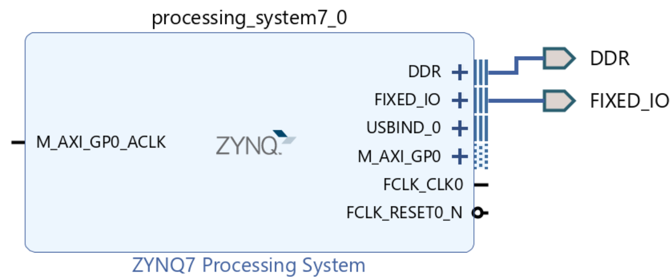


Figure 31: The Zynq 7 processing system

Now the user will need to do some changes to the Zynq 7 IP settings. Double click on the IP and a Zynq 7 IP settings window will appear. Navigate to *PS-PL Configuration* and enable the high performance AXI slave interface as seen in Figure 32.

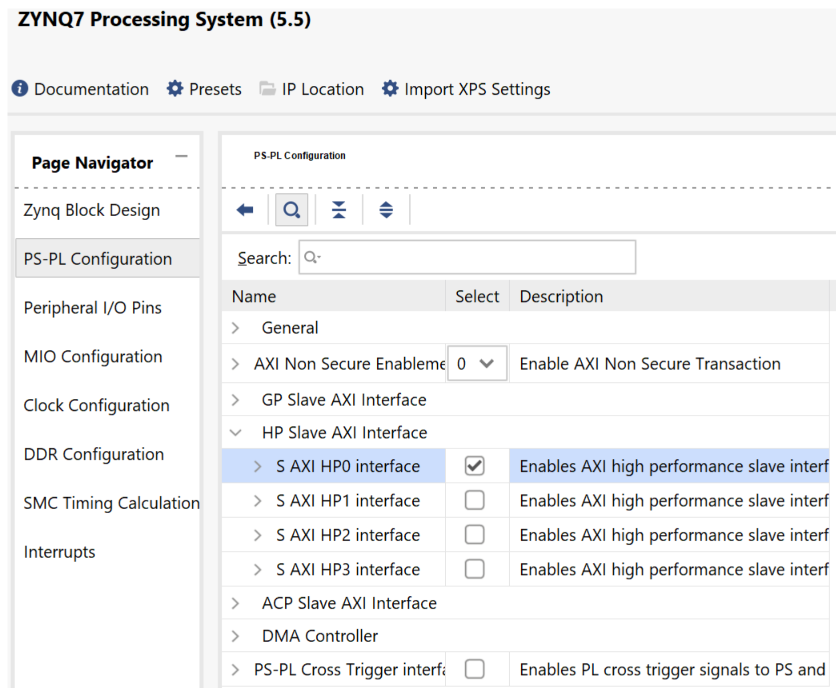


Figure 32: Enabling high performance port

Next, navigate to *Interrupts* and enable fabric interrupts and the *IRQ_F2P* interrupt ports, as seen in Figure 33. Click *Ok* when the changes are done. After that the Zynq 7 IP should look like Figure 34.

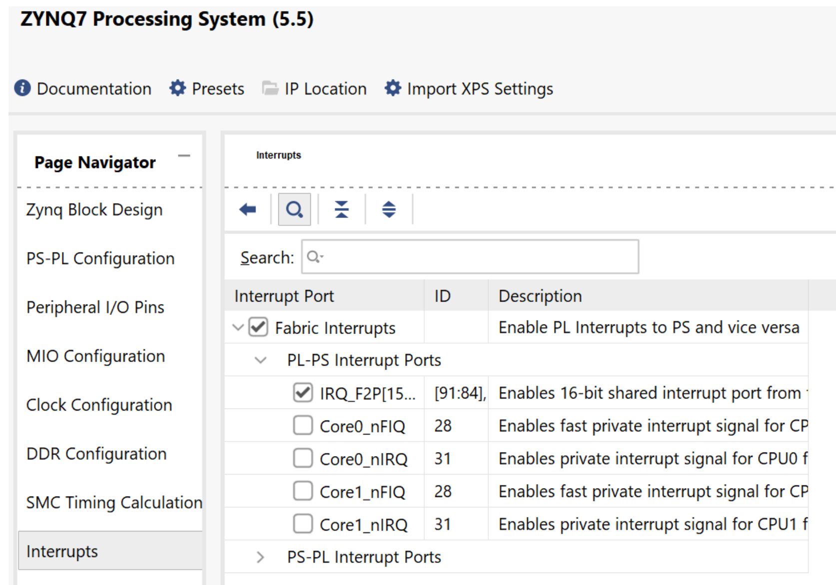


Figure 33: Enabling interrupt ports

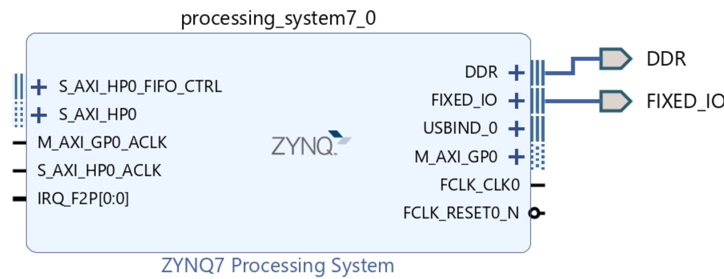


Figure 34: Additions to the Zynq 7 processing system

F.2.2 AXI Direct Memory Access

The next IP to add is the *AXI Direct Memory Access* IP. Change the name of the IP to *dataDMA*. Before running the connection automation, changes to the settings of the IP is necessary. Change all the settings to match the settings in Figure 35. Some of the settings are described in Section 4.2.5.1. After this is done you will need to run connection automation twice. Do not worry about the IPs appearing out of nowhere, those are a byproduct of the DMA IP and is necessary to function properly.

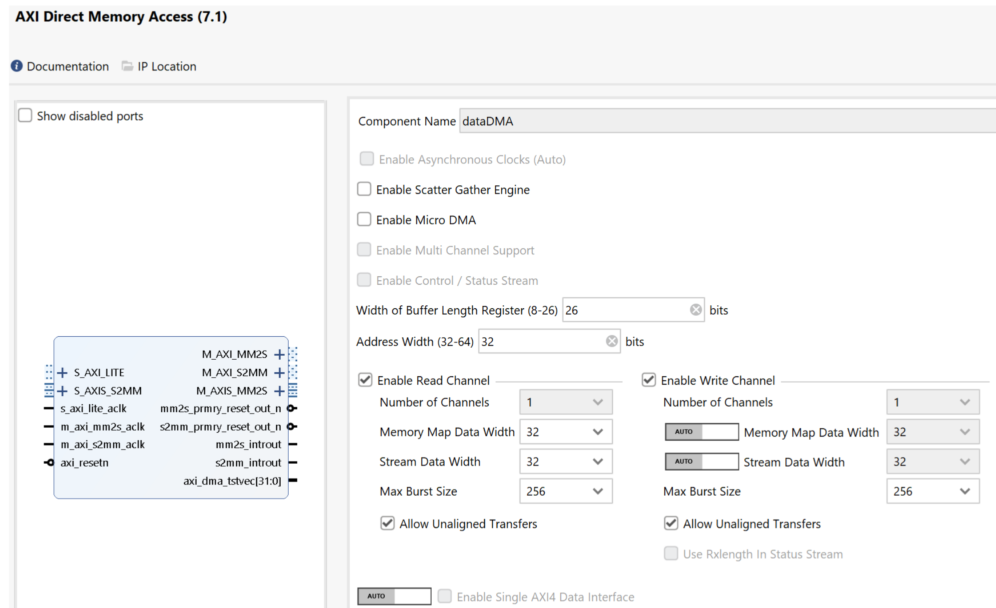


Figure 35: Settings for the DMA IP

F.2.3 AXI4-Stream Data FIFO

The FIFO IP needs some changes in the IP settings as well. In Figure 36 most of the correct settings can be seen. However, the user will also need to enable the *Almost full* signal located under the *Flags* tab.

After running the connection automation the first manual connection is to be made. Connect *M_-AXIS* port on the FIFO to the *S_AXIS_S2MM* port on the DMA. This is the first of many AXI4-Stream connections in the design. It is recommended to study Figure 37 for more context on the upcoming AXI4-Stream connections. The black arrows, except the one between memory and the DMA, are AXI4-Stream connections. The direction of the arrows indicate slave to master configuration.

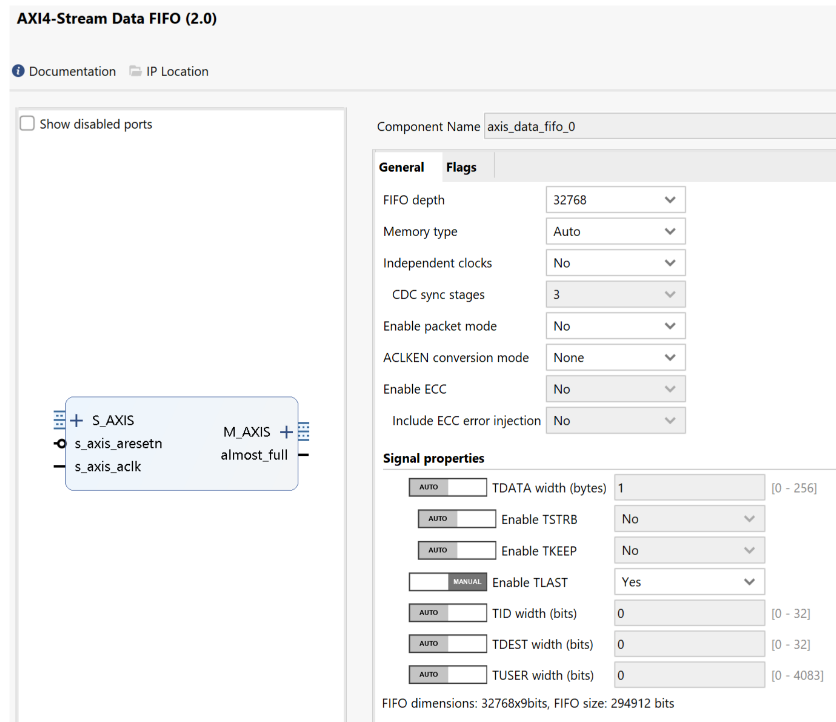


Figure 36: Settings for the FIFO IP

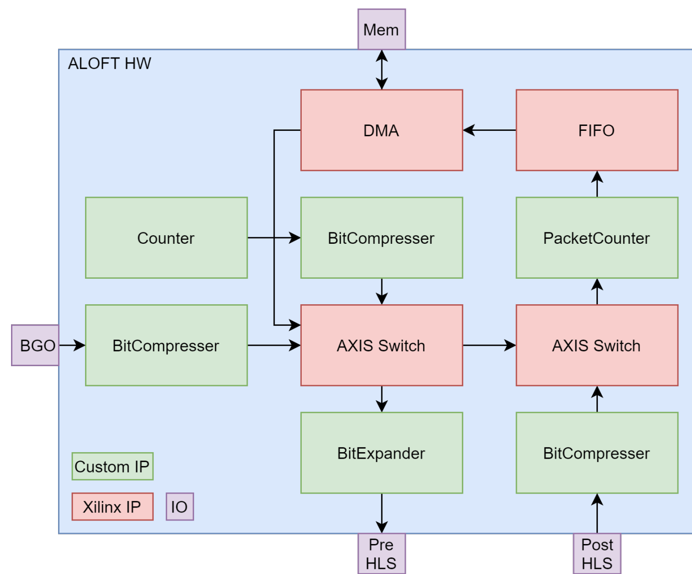


Figure 37: Data flow to explain AXI4-Stream connections

F.2.4 Adding Custom IPs

The custom IPs are easy to use and configure, and so, the main part of adding them to the design is to make the correct connections to the IP. The IPs are added to the design by dragging them from

Design Sources in the highlighted *Sources* tab in Figure 30(Remember to add three instances of the Bit Compressor). Every custom IP uses the same *aclk* and *arstn* as every other IP in the design. However, Vivado has problems to connect these under *Connection Automation*, and therefore needs to be connected manually. After IPs are connected to clock and reset, connect the *M_AXIS* port on the Packet Counter to the *S_AXIS* port on the FIFO. The other IPs will be connected later in the guide. The same holds true for any other port that needs a connection.

F.2.5 AXI4-Stream Switch

As seen in Figure 37, there is the need for two AXI4-Stream Switches in the system. Give the name of one of these to *SOURCE_MUX*, and the other one *HLS_MUX*. The *SOURCE_MUX* will have three to two configuration, while the *HLS_MUX* will have two to one configuration. The configuration change is done in the IP settings. While in the settings, enable *Use Control Register Routing* on both. Now run the *Connection Automation*. There may be some clock and reset ports not connected since there are two inputs for both signal. Connect them if they did not do so.

Connect the *M00_AXIS* port on the *SOURCE_MUX* to the *S00_AXIS* port on the *HLS_MUX*. Connect the *M00_AXIS* port on the *HLS_MUX* to the *S_AXIS* port on the Packet Counter. This is as seen in Figure 37, with the correct ports connected. Connect the *S02_AXIS* port on the *SOURCE_MUX* to the *M_AXIS_MM2S* port on the DMA. The rest of the input-ports need to be connected to a Bit Compressor, while the remaining output-port is to be connected to the Bit Expander. The last AXI4-Stream connection to be made is the one coming from the Counter. This has to be connected to the Bit Compressor connected to the *S00_AXIS* port on the *SOURCE_MUX*. The outstanding AXI4-Stream ports will be addressed in section F.3.

F.2.6 AXI GPIO

The AXI GPIO IP is used for a few different purposes, and consequently the setup differs. Add three AXI GPIOs to the design and name them; *controlRegister*, *pckCntStatus*, and *axiGPIO_intc*. Run the *Connection Automation*, however, uncheck the GPIO options as seen in Figure 38 . These will try to connect to the Z1 board IO. Vivado will prompt the user to run *Connection Automation* until the GPIO ports are connected.

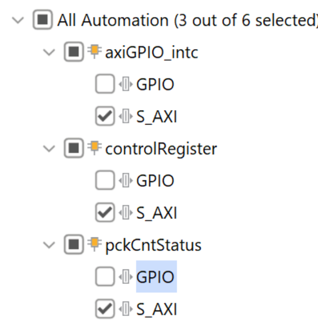


Figure 38: Connection automation for the AXI GPIO IPs

The *controlRegister* is used as a control register, and therefore the output needs to be split up to the right signals. These signals are; *Start* and *SpeedSel* which are connected to the Counter; and *pckSizeSel* connected to the Packet Counter. To achieve the splitting, the slice IP is used. Add three of the slice IPs and name each of them to a corresponding signal. In the *controlRegister* IP settings, set the output width to 8-bits and set the GPIO to be all outputs. Connect the GPIO to the slice IPs and the slice IPs to the corresponding port, see Figure 39. In the slice IP settings the range of each signal is set. The correct ranges can be seen in Table 8.

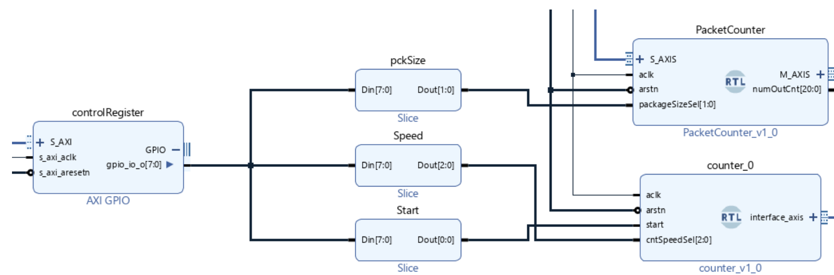


Figure 39: Control register connections

Control Register:							
7	6	5	4	3	2	1	0
UNUSED		packageSizeSel		cntSpeedSel			START

Table 8: Control register for the Custom IPs

In the IP configuration for the *axiGPIO_intc*; enable dual channel; set both to a width of 1-bit; select all inputs on both; and enable interrupt. Connect the GPIO input to the *almost full* port on the FIFO. Right click on the GPIO2 port and select the *Create port* option. Name this port button, and connect it if it is not done automatically.

The *pckCntStatus* GPIO is to be connected to the *numOutCnt* port on the Packet Counter. Enable all inputs and a width of 21-bits in the GPIO settings to do so.

F.2.7 AXI Interrupt Controller

To use interrupt with PYNQ, an AXI interrupt controller IP needs to be added along with a concat IP. In the concat, the two interrupt signals from the DMA(mm2s_introut and s2mm_introut ports) and the interrupt signal form *axiGPIO_intc* needs to be connected. Then the *iqr* port on the interrupt controller is connected to the *IQR_F2P* port on the Zynq 7 processing system.

F.3 Creating hierarchies and routing signals

To make the system clearer and orderly in Vivado, and when accessing hardware from software, it was decided to group the IPs in different hierarchies. To group IPs in hierarchies, select the desired IPs and right click, then choose the *Create Hierarchy* option. The correct hierarchies can be seen in Figure 40. It is important to give the hierarchies the same names as those seen in Figure 40.

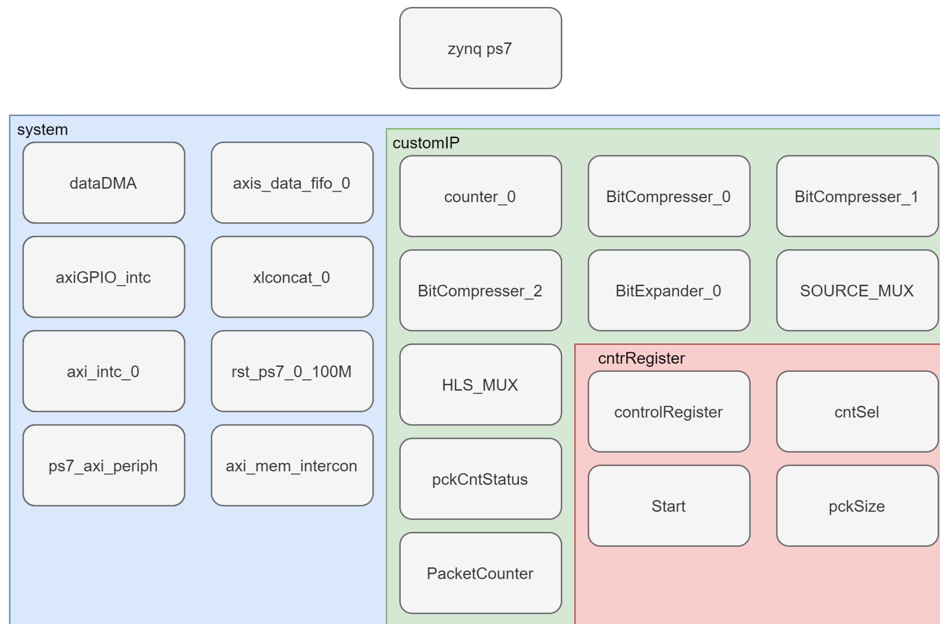


Figure 40: Hierarchy used for the ALOFT system

When the hierarchies are created, double click on the *customIP* hierarchy. Inside the hierarchy, right click on the *M_AXIS* port on the Bit Expander, and select *Create interface port*. Give this the name *PRE_-HLS* and make sure it the AXI4-Stream interface is selected. Do the same to the remaining AXI4-Stream slave ports on the Bit Compressors. Give the one connected to the *HLS_MUX* the name *POST_HLS*, and the one connected to the *SOURCE_MUX* the name *BGO_TOP*. Exit the hierarchy and do the same inside the *system* hierarchy. On the block diagrams top level, connect the *PRE_HLS* port to the *POST_-HLS* port if no HLS is to be used. Otherwise connect chosen HLS solution between the two ports.

F.4 Running implementation and generating bitstream

The block design is now done and only a few steps are left until the hardware can be used by software.

Save the block design(ctrl+s when the block diagram is selected), and locate the *.bd* file in the sources tab, highlighted in Figure 30. Right click on the *.bd* file and select the *Create HDL wrapper* option.

When it is finished generating, click on the *Run implementation* option in the *Flow Navigator*. Open the implemented design when finished. When opened, navigate to, and select, *I/O ports* located under *Window* in the menu bar. Now change the package pin for the button port created earlier to D19 as seen in Figure 41. The D19 pin is connected to button zero on the Z1 board, and is only relevant when using it. The pin can be different on your board. Give it the pin of a button on the card. Save, and give an appropriate name to the constraints file(e.g. btnConstraint). Now click on the *Generate Bitstream* in the *Flow Navigator*.

Figure 41: Connecting button 0 to the AXI GPIO input by selecting correct package pin

This concludes the guide of building the hardware from scratch. For further usage, see Appendix E.