Hochschule Karlsruhe – Technik und Wirtschaft
Faculty of Electrical and Information Engineering
Sensor Systems Technology Master

Project B

Fully Programmable Adaptive-Digital-Filter Architecture implemented on Altera DE2i-150 development board

Submitted by: Luis Eduardo Ardila[ǂ]

Matriculation number: 48121

Supervisor: Prof. Dr. Michael Bantel

Co-supervisor: Prof. Dr. Christian Langen

October of 2015

ǂ Email: arlu1011@hs-karlsruhe.de – luis.ardila@bozica.co

Abstract

In the following report is contained a description of how it is possible to implement a fully programmable architecture from an already established set of modules. The goal of this implementation is to provide a deeper degree of observability of the filter to the designer by means of hardware and software co-design. It is presented how fixed modules can be turn into configurable ones and how the software can modify or configure them over time.

The architecture here presented provides a stepping stone for rapid prototyping and fast digital filter designs verification. Within this document it is demonstrated how adaptive architectures can be an excellent choice in the first debugging stages of a filter design, and how new parameters and coefficients can be configured and analyzed within seconds. No need for recompilation of the hardware is necessary with the proposed implementation.

Finally, a description of the software requirements for the system configuration is shown and some remarks in how to further improve and characterize the filter are presented in the document.

Table of Contents

Illustration Index

Index of Tables

1 Introduction

1.1 Previous work

The work here presented is a continuation of different efforts with the goal of improving the signal quality of Germanium detectors. Semiconductor radiation detectors are very sensitive to mechanical disturbances often produced by pumping motors. The crystalline structure of the detectors makes them prone to exhibit some degree of piezoresistivity when physically stressed and therefore influencing the charge summing amplifier.

In previous work by T. Hasenohr [1] it was shown that if we analyze the signals coming from the Germanium detector and the microphonic noise in the frequency domain, we find that they are widely separated and that the dominant component in the microphonic noise is no larger than 1kHz, whereas the detector signal could vary very rapidly. This consideration gives us a great advantage as already mentioned in [2] and [3], we can undersample the signal coming from the noise source and make the adaptive filter adjust itself to the noise, so then all calculations will be at this reduced frequency.

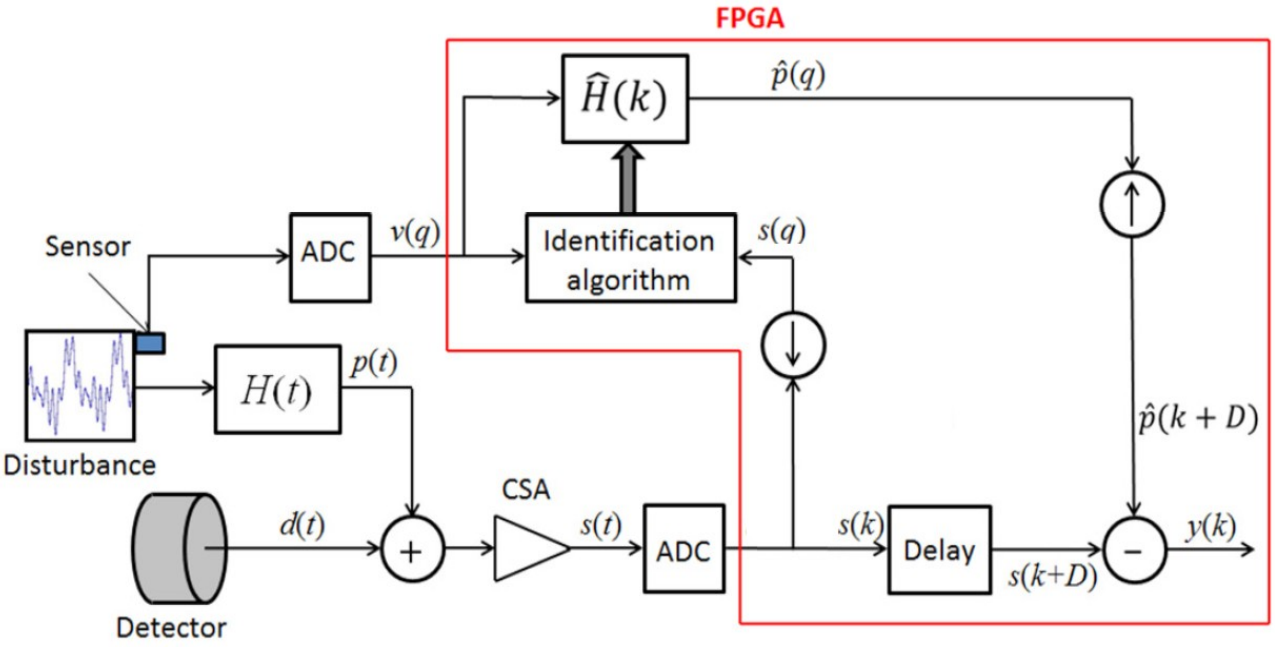

Illustration 1: Block Diagram of the Microphonic Noise Cancellation Strategy[1]

The proposed strategy for noise reduction by adaptive techniques presented in the Illustration 1 was later implemented in hardware by two different approaches which constitute the bases for the work presented in this document; C. Pfeiffer's Master Thesis [4] and L. Ardila's project A report [5]. In both of this implementations it has been shown how the implementation is possible in medium sized FPGAs, and how the algorithm successfully converge after some time of operation.

However, both implementations also pointed out the difficulty at the time of evaluating the performance of the filter, not enough data points for creating reliable statistics or too short filters

designed were the limiting factors. There it is also mention how some features and the overall behavior should be investigated in further detail in order to completely verify it. In future chapters, the intention is to merge the best of both attempts and create a versatile architecture which satisfies the need for debugging at this stage of the implementation.

1.2 FPGA Technology

Field programmable gate arrays (FPGA) are in principle matrix of configurable blocks which can be interconnected between them with the use of configurable routing boxes. In the Illustration 2 we observe a very simple representation of an FPGA architecture [6]. This simple representation helps us identify the five most important components inside an FPGA; the configurable logic blocks (CLB), the RAM memory, DPS blocks, I/O ports and the routing boxes.

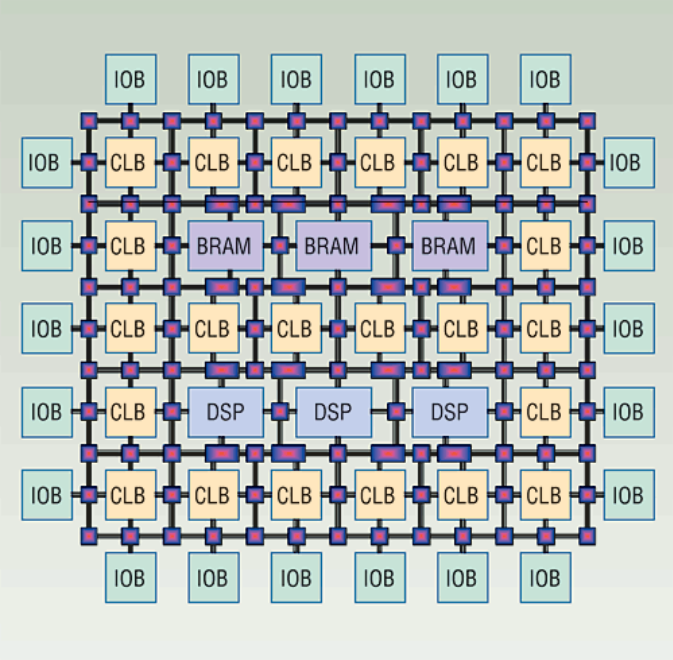

Illustration 2: FPGA Architecture[6]

The CLB uses small look-up-tables (LUTs), flip-flops and multiplexers. In Altera devices the smallest logic unit is called a Logic Element (LE) which contains a 4 input LUT, a programmable register, a carry chain connection, a register chain connection, and a number of possible interconnections between neighboring elements. Several LEs are grouped together to form a logic array block (LAB), showed in Illustration 3, 16 LEs per LAB in the case of Cyclone IV FPGA used in the DE2i-150 Board.

The LABs are called slices in Xilinx FPGAs [7], which could contain two separate 4-input 1-output LUTs, fast-carry dedicated logic, two flip-flops, and some shared control signals. Additionally, each LUT can be used as a 16 x 1 RAM or ROM.

By looking at the Illustration 2, we can identify the feature number one of the FPGAs, which is its reconfigurability. It is build from arrays of fixed multipurpose elements with interconnections around them, this flexibility is the one that allows them to adapt and be useful for a wide range of

applications. This same concept is the one we would like to implement through this project targeting an specific application which is the microphonic noise reduction in germanium detectors.

In the following chapters it would be described the features and arguments for implementing this type of configurable architecture as well as the results obtained in terms of capabilities of the system and data speed transfer.

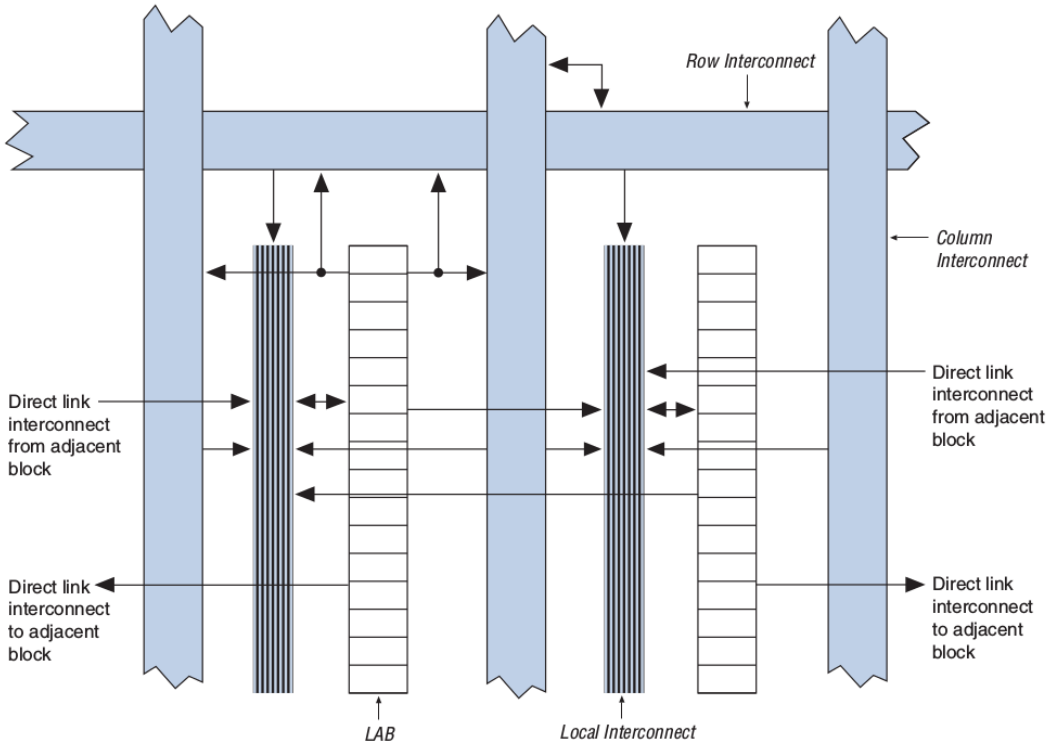

Illustration 3: Cyclone IV Device LAB Structure

2 The Adaptive-Filter Architecture

In this chapter, a detailed description of the implementation of the adaptive-filter architecture is given, even though it has already been described in C. Pfeiffer's Master Thesis [4], we would like to go a little deeper in the implementation features of each block itself and have a more realistic representation. The block diagram of the full microphonic noise cancellation filter proposed by Pfeiffer is shown in Illustration 4, there we can see how the adaptive filter receives a down-sampled and high-passed detector signal <s(q)> along with the mechanical disturbance signal <v(q)> to generate the estimated noise <$\hat{p}$(q)>. The estimated noise <$\hat{p}$(q)> is then interpolated and subtracted from the delayed detector signal <s(k)>.

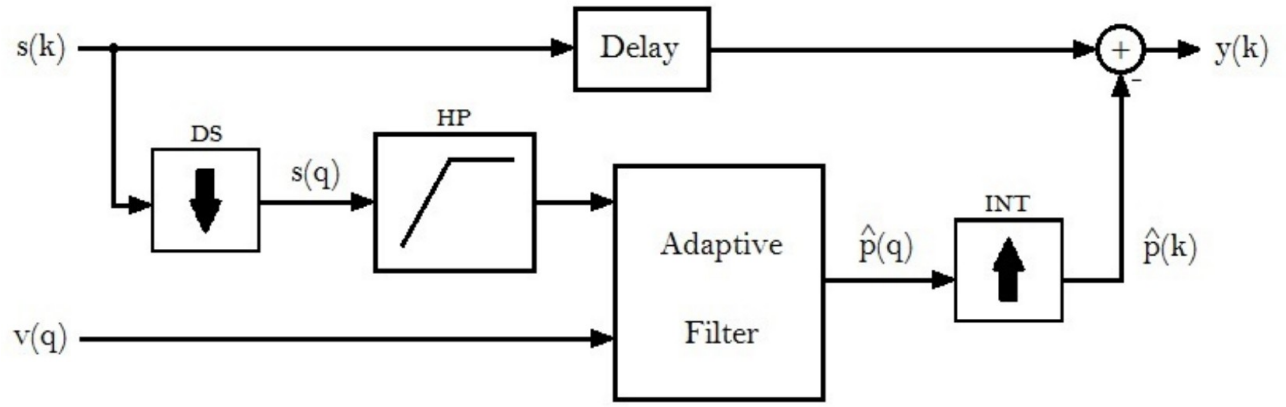

Illustration 4: Adaptive-Filter Architecture[4]

It is worth noting from the Illustration 4 that there are two sampling frequencies (k and q) involved in the diagram, and the down-sample filter (DS) and the interpolation filter (INT) serve as bridges between the two. The factor for the DS and INT filter is determined using a sampling rate of 25 MHz for the detector signal and an analysis of the frequencies involved in the mechanical disturbance, which at 5 KHz are only 1 % of the maximum, therefore 10 kHz was chosen as sampling frequency, giving us a factor of 2500.

2.1 Down-Sample Filter

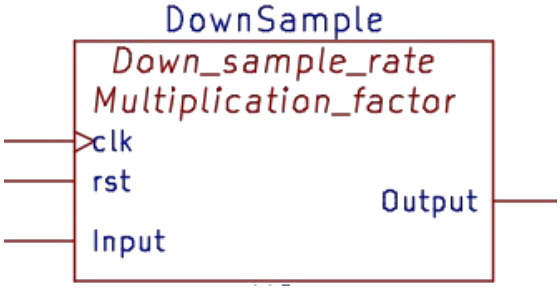

Illustration 5: DownSample Block

From the Illustration 5, we can see how the implementation of the down-sample filter is determined by two parameters which are adjustable, the down-sample_rate and the multiplication_factor. These two properties of the filter are analyzed at the moment of synthesizing the design and are converted to constants within the block. Both parameters need to be correlated between each other to effectively calculate the average depending on the down-sampling rate (r_{ds}). This relation is given by mult_factor = 1 / r_{ds} following the convention of an average filter.

The functioning of the down-sample filter is given by the Illustration 6, where we can see how both parameters are fundamental in the flow control and the calculation of the result. The program starts by initializing the variable *i* representing the order of the system. In the next stage the coefficient[i] of the filter gets multiply by the delayed-input[*i*]. The result is accumulated in the veriable *acc* until *i* has reach the desired down-sampling rate at which point the output y gets updated with the value of *acc*.

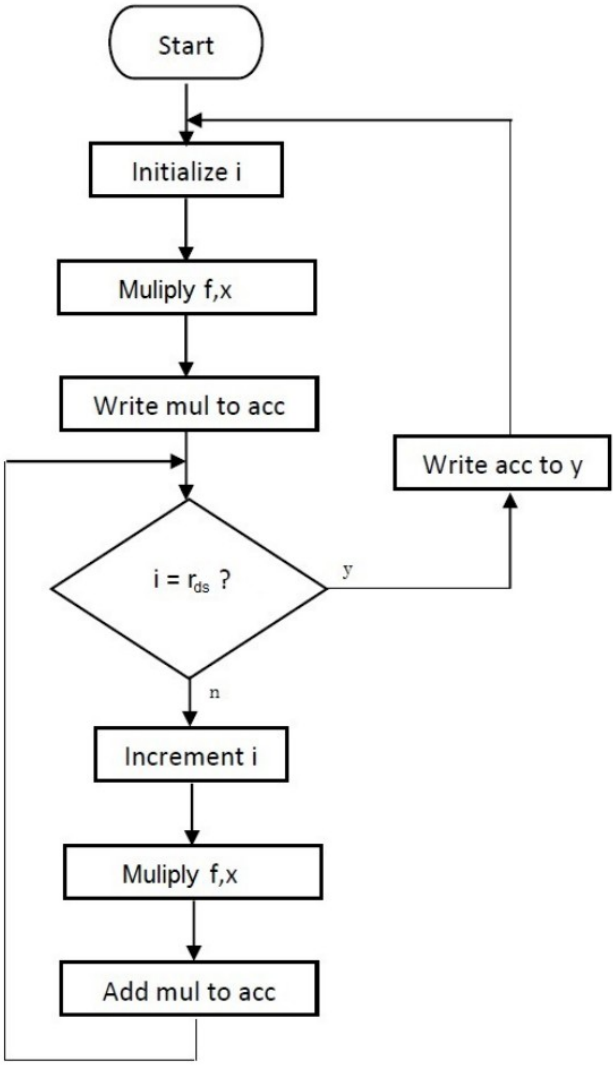

Illustration 6: Flow Diagram of the Downsample, Average Filter [4]

2.1.1 Improvements to the Current Implementation

In the flow diagram of the down-sample average filter (Illustration 6) we can find the first opportunity to optimize the behavior of the filter. Before the index i is evaluated there are three states that are not necessary for the execution of the filter. Those states are already implicit in the way VHDL signals are evaluated and assigned within a process.

Moreover, looking at the implementation we have two independent processes clocked with a phase difference of 180 degrees, this is a practice that is not recommended for complex and fast designs because it uses more clocking networks than needed, and once more it is not necessary due to the fact that VHDL will evaluate the signals in the current edge and then update the ones that are assigned inside the processes.

For example:

```
process(clk)
begin
  if (rising_edge(clk)) then
    acc <= acc + mul;
  end if;
end process;
```

In this example the new value for acc is calculated with the current value of acc and mul, and we do not need to have the double process clocked 180 degrees apart for the update of index i and the update of the accumulator acc.

2.1.2 Programmability of the Down-Sample Filter

As it was shown before, we can have a generic design in the VHDL code implemented based on those two parameters. To make the down-sample average filter programmable and configurable in runtime, we have to implement inputs which contain the values for this parameters. Therefore they stop being constants for the synthesis and now are converted to inputs which we can later manipulate during execution of the code.

The new programmable down-sample filter is shown in Illustration 7. There we can see how we could have any kind of decimation rate and multiplication factor, which is ideal for a testing phase once we have real data from the detector. We have to remember that the decimation rate has to be the same as the interpolation rate, and therefore the clock generation for the the interpolation block has to be adjusted accordingly.

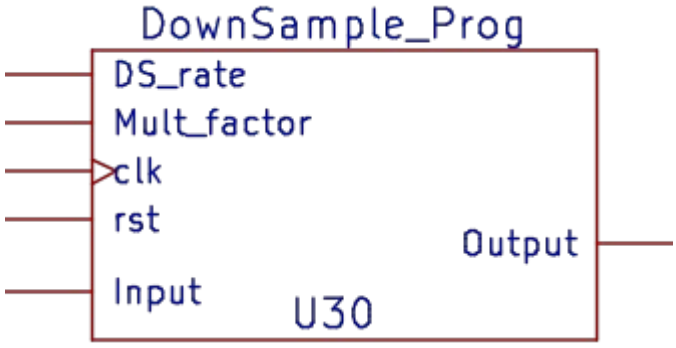

Illustration 7: Programmable DownSample Filter

2.2 High Pass Filter

The implementation of the High Pass filter is necessary to remove the DC steps generated by the active Charge Summing Amplifier CSA as shown in Illustration 9. There, we can see how the negative step size determines the amplitude of a detector hit, and the positive step the discharge of the feedback capacitor.

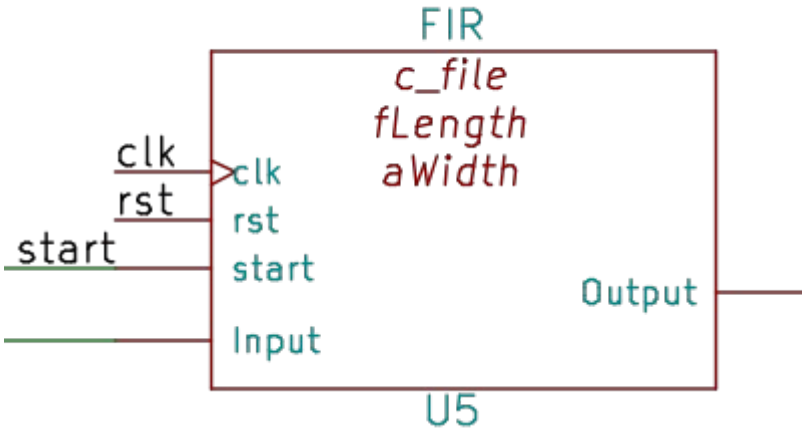

Illustration 8: High-Pass FIR Filter

The implementation of this filter is done by a sequential version of the direct form of the finite impulse response (FIR) filter making use of the decimation rate, we can utilize only one multiplier. The sequential loop is determined by one of the parameters included in the Illustration 8. The other parameter serves simply as an indicator of how wide should be the addressing to gather all the coefficients determined by the length of the filter.

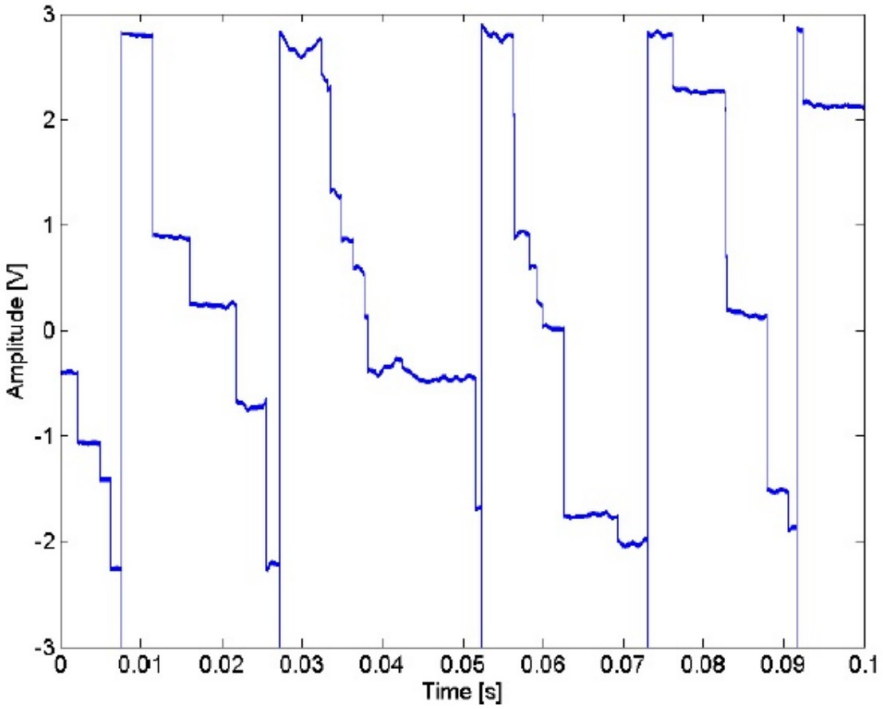

Illustration 9: Detector Signal with an Active CSA [1]

In Illustration 10 it is shown the reduction of the DC components of the signal after the high-pass filter, therefore making the analysis much easier for the subsequent steps. It is necessary that the filter possess the least ripple in the band pass to avoid unwanted scaling of the peaks amplitude.

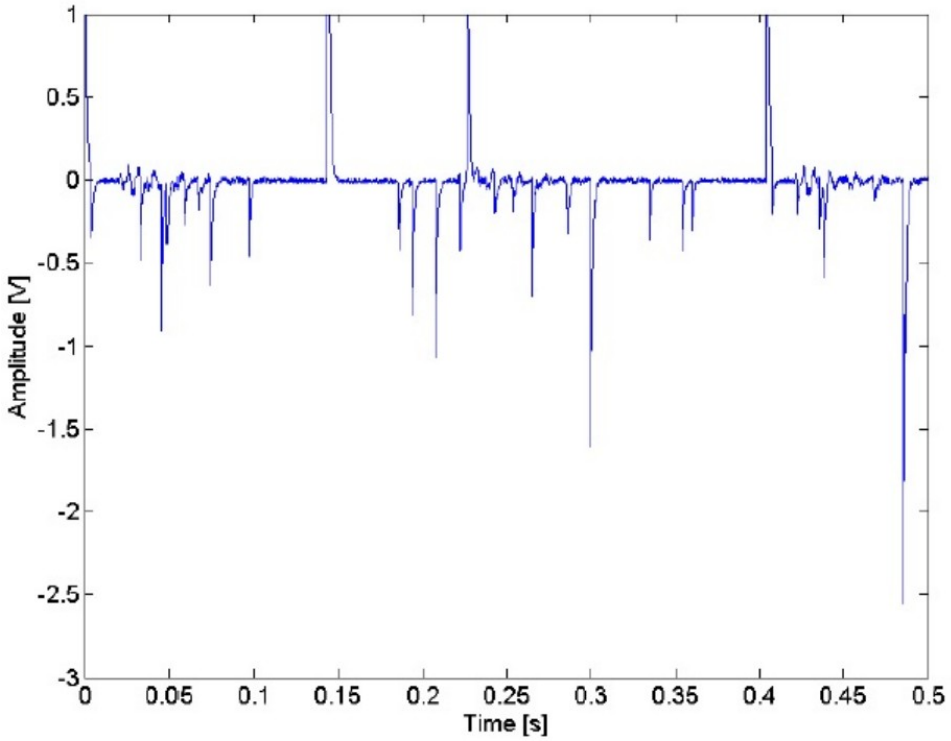

Illustration 10: Detector Signal with an Active CSA after High-Pass Filtering [1]

2.2.1 Programmability of the High-Pass FIR Filter

Now that we have explored the functioning of the filter, and how the parameters determine the final implementation, I would like to propose a way to have the length of the filter being adjusted in run time by a register which is configurable from the user interface in software as explained later in chapter 4.

By implementing a variable size of the filter, we need have to have a fixed size memory declaration for the coefficient and the shift register. In this way, we have a maximum number of taps for the FIR filter which can then be resized down to evaluate the performance.

The implementation is done by integrating a memory block to the Avalon bus declared in the PCIe module. This memory block is addressable through the software side of the PCIe driver by an unique descriptor and therefore we can read and write the coefficients as well as the length during runtime. This ability is a key component to have a quick way to test the dependence of the FIR length to the signal quality.

As explained in chapter 3, the extended architecture modification allows us to have only the down-sample and the high-pass filter in the design. Therefore we are able to analyze the quality of the signal after these two components and determine the relation with the size of the FIR filter.

We can also make quick studies of the ripple in the pass band typical in these kind of filters and

have an optimal size depending on the desired accuracy for different frequencies. Note that we are expecting the detector signal to be relatively fast, therefore having mostly high frequency components.

The implementation of the memory is done with a double port, one for the Avalon bus with the PCIe module and another port so that the values can be read sequentially from the FIR engine. This infrastructure is already in place due to the fact that in [4] the coefficients were read as well from a memory block which was initialized by the non-synthesizeable file_read function in VHDL. This only with the purpose of simulating the system, but made it unsuitable for implementation.

2.3 Adaptive Filter

The implementation of the Adaptive filter in [4] follows the same algorithm proposed in [5] which ultimately will lead to the equation (2.3) for updating the filter coefficients. The Widrow-Hoff LMS algorithm can be found in the following box.

$$h[n+1] = h[n] + \mu e[n]x[n] \qquad (2.3)$$

The Widrow-Hoff LMS Algorithm [8]

1) Initialize the (Lx1) vector $h = x = 0 = [0, 0, \ldots, 0]^T$
2) Accept a new pair of input samples {x[n], d[n]} and shift x[n] in the reference signal vector x[n]
3) Compute the output signal of the FIR filter, via
   $y[n] = h^T[n]x[n]$
4) Compute the error function with
   $e[n] = d[n] - y[n]$
5) Update the filter coefficients according to
   $h[n+1] = h[n] + \mu e[n]x[n]$

Now continue with step 2.

This algorithm we can visualize in the Illustration 12 where it is very clear the implementation of it and we can identify two main data-paths, the one containing the delayed signal and the filter coefficients, and the other in charge of updating the coefficients based on the error and the convergence speed.

Apart from the crude implementation of the filter, as we have seen with other modules, the adaptive filter in [4] was also implemented in a way that we have some parameters which allows for the configuration of the module at the time of synthesis. These are shown in Illustration 11.

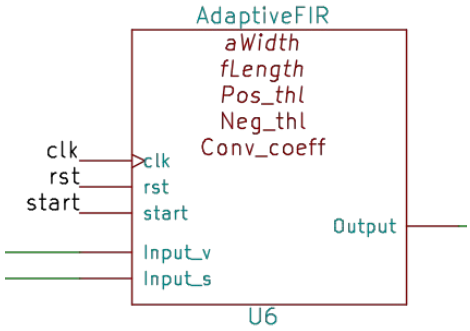

Illustration 11: Adaptive Filter Block

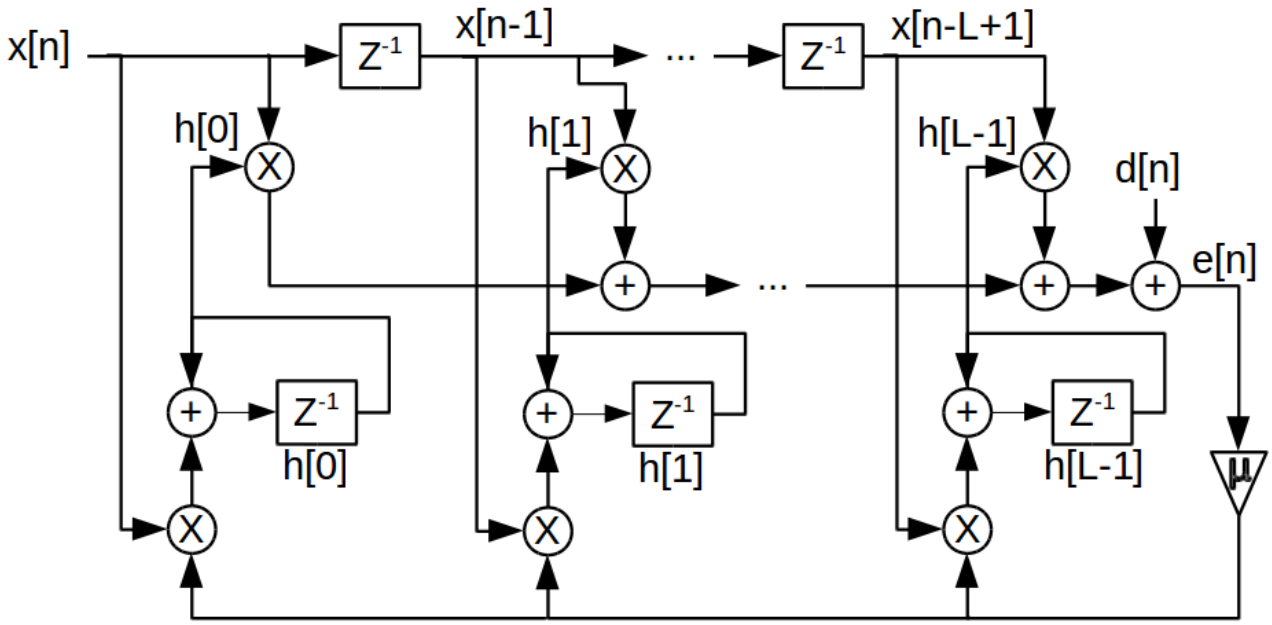

Illustration 12: Signal Flow Graph of the LMS Algorithm [1]

2.3.1 Improvements to the Adaptive filter

There is only one clear improvement that can be done to the filter if we required more coefficients or a faster execution time. At the moment we have only one multiplier assign to this module making it very efficient in resource utilization.

However, if we were to implement the design with tight clocking and fully pipelined signals using the maximum possible room, which is the decimation factor. We will need an extra multiplier in order to compute the updating of the coefficients in parallel to the FIR part of the filter. In this way we no longer need the 2*Length + 7 clock cycles stated in [4] but probably only length + 7 clock cycles.

There might be a possibility to optimize those 7 clock cycles which are due to the waiting times for fetching the values from the memory, but this is really insignificant.

2.3.2 Programmability of the Adaptive Filter

It is clear already which of the parameters we can have to be updated in runtime by software by the

user, these are; the filter soft-lenght, the adjust flag, the convergence speed coefficient, the positive threshold, and the negative threshold. For the filter length once more we would like to implement a fixed size larger than the current implemented and tested design in order to have some room to grow and play with the system.

In addition, we would like to be able to read the coefficients of the filter at any given time, therefore, the same strategy is implemented here, where we have a dual port ram with an Avalon PCIe side and a filter side for storing the coefficients as they are being updated.

By means of the SignalTap module we can observe how the adaptation of the filter coefficients are being done for each sample and then stored in the memory, hence we can access them at any given time by the PCU and make analysis of the filter adaptation over time for a given set of input signals.

The adaptive filter output can be assigned to one of the DAC inputs so that we can see the effect of the adaptation process for a given set of inputs, which are also settable to the other DAC input.

2.4 Interpolation Filter

The interpolation filter works by inserting zeros between samples at a factor of the interpolation rate. Using the design in [4] the filter is divided in different small factors, allowing the overall filter to utilize less resources

Moreover a clever implementation represented in Illustration 13, along with the pre-multiplication of the filter coefficients by the rate factor allows us to not need the final low pass filter stage once the zeros has been added [9, 10]. This reduce even more the complexity and increase the performance of the filter

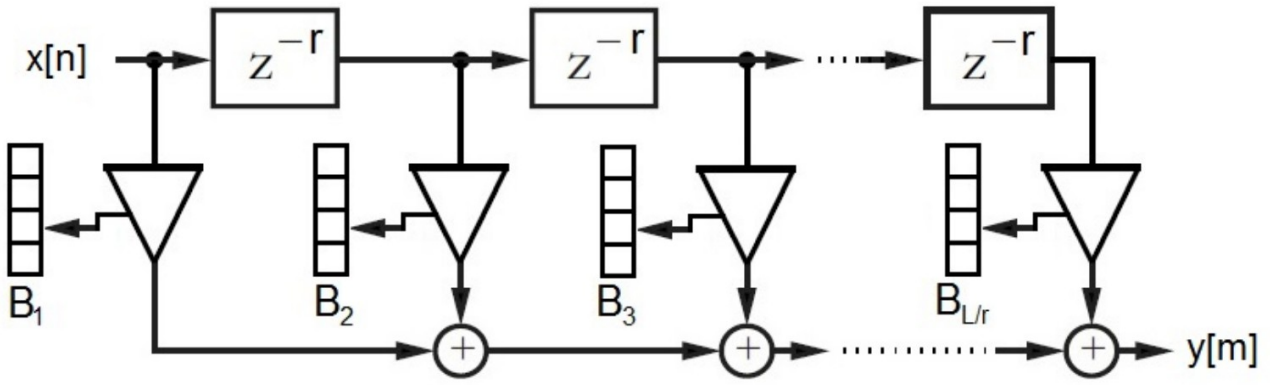

Illustration 13: Interpolation FIR Filter [4]

2.4.1 Programmability of the Interpolation filter

For the interpolation filter, which is really a series of small filters as seen in Illustration 13 we can have the same principles applied as before. A dual port RAM block has been implemented connected to the Avalon PCIe block and the filter itself.

Interpolation filters size are dependent on the interpolation rate, and therefore we would like to keep

the size fixed to the same values as they were implemented in [4].

Now we have the ability to update the filter coefficients at any time during the execution of the filter by using a software tool design along with the FPGA architecture. This tool is capable of converting filters designed in Matlab or Python and then send them to the FPGA via the PCIe link using the DMA engine.

2.5 Delay

The Delay module is used to store the detector signal at the sampling frequency so that once we have the results from the interpolation filter we can subtract one from the other, and therefore eliminating the mechanical noise.

The delay is implemented in the form of a ring buffer using RAM blocks of the FPGA, the architecture is shown in Illustration 14. There we can see how the length of the delay is determined by the size of the buffer, with no room for adapting it.

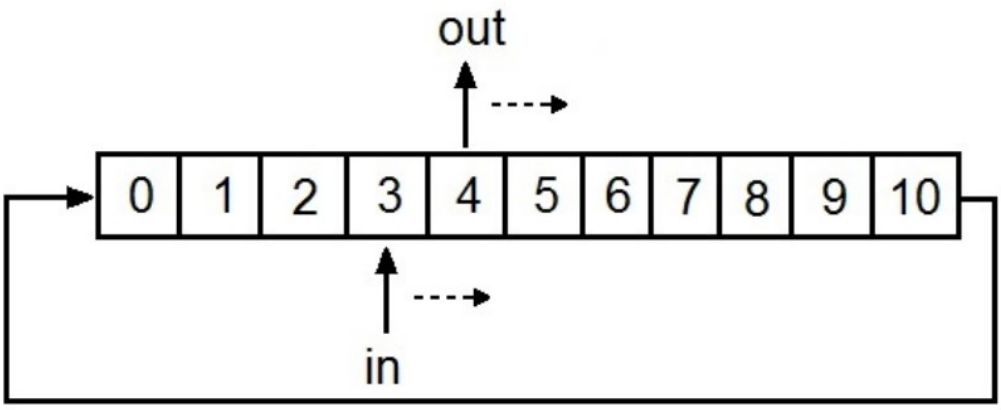

Illustration 14: Ring Buffer Schematic [4]

Even though we can calculate the delay depending on the filter size for each of the components, this value may not be that accurate and we would like to have a possibility to move it around the calculated value to better the alignment.

2.5.1 Programmability of the Delay

Now, if we take into account the variability of all the previous described components, our delay module should not have a fixed amount of delay only configurable at the synthesis time. But, instead, we would like a runtime configurable delay buffer which we can adjust depending on the current configuration of the filter.

This is achievable by simply defining a maximum length of which the delay may be, this taking in consideration the maximum implemented FIR, AdaptiveFIR, decimation and interpolation rates. Then, by using the same principle we simple add an input signal which will adjust the difference in the index for input and output of the delay shown in Illustration 15.

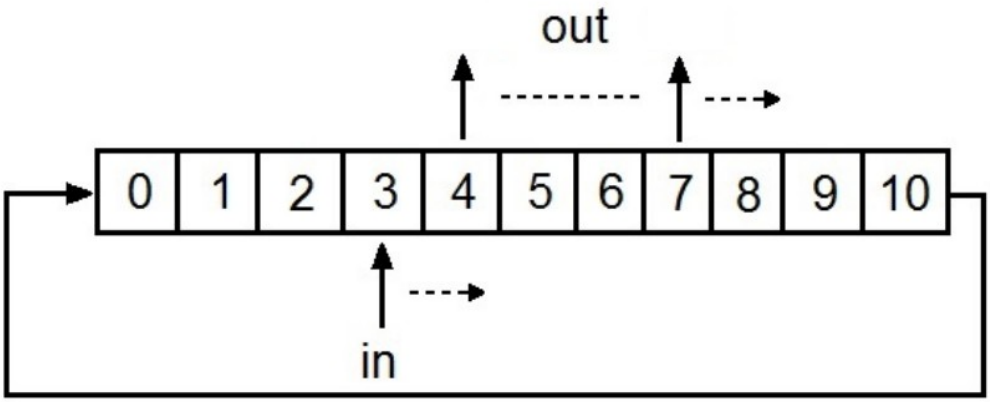

Illustration 15: Programmable Delay (ring buffer)

This will induce some memory space which will be overhead of the module, but it is needed in case we design the other modules also with that extra size. We then would have to make the delay calculations following [4] recommendations for each configuration of the filter.

2.6 Subtractor

This module is a simple subtraction operation with a multiplication factor parameter for the $\hat{p}(q)$ signal. The module is clocked at the detector sampling frequency.

It is important to notice that we would like to have the same mechanical noise amplitude at the end of the whole filter chain, in case this is not achieve we can make use of this multiplication factor to compensate for signal amplitude drop.

2.6.1 Programmability of the Subtractor

There is not very large room for customization apart from having the multiplication factor for $\hat{p}(q)$ to be adjustable, therefore a special register to control this parameter has been implemented and now we can change at any given time by software the value of this factor.

2.7 Detailed Architecture

In the Illustration 16 we can see how the architecture designed by C. Pfeiffer [4] is implemented. We can see in more detail the clock management system, which is a very important part of the design in order to understand the working of the overall filter.

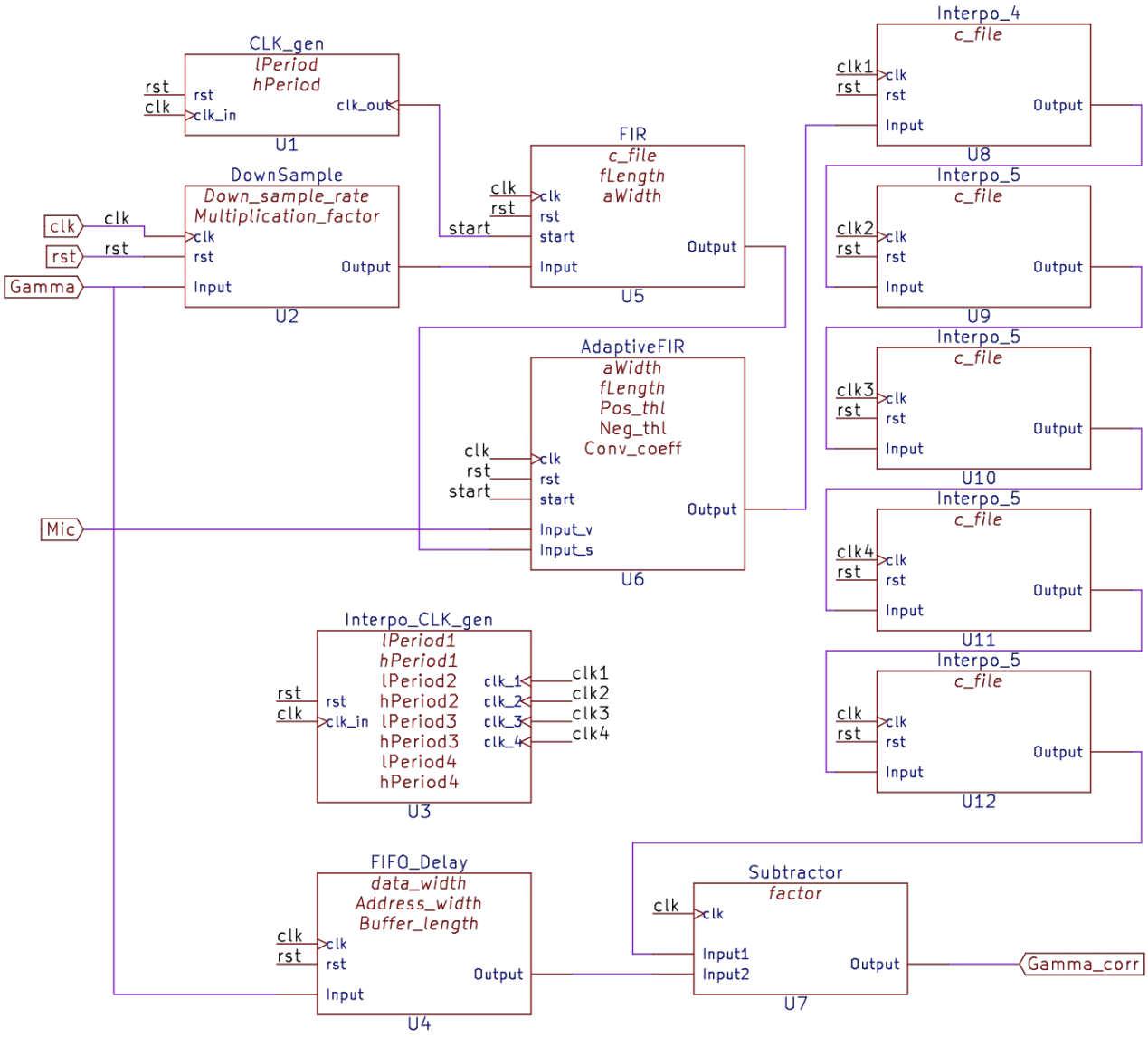

Illustration 16: Adaptive Filter Architecture as in [4]

We can see how the building blocks have been designed with the customization in mind, by implementing the component with general parameters we can easily change the properties and simply recompile the architecture. This, as it has been expressed along this chapter is a good idea, but it is not optimal for FPGA designs. The main reason is that the compilation time required to synthesize the system might be in the order of half an hour. Therefore, the ability to change the architecture for testing different parameters is limited by the compilation times. The extended representation of the architecture of the filter is helpful to understand the multiple interpolation stages that are taking place at different clock speeds in order to slowly get from 10 kHz to 25 MHz. This approach helps reducing the number of coefficients needed to perform such a degree of interpolation.

Along this chapter some ways to improve the reconfigurability of the individual blocks has been mentioned, however nothing has been mentioned about the system as a whole. Here, for the first time we can see how small changes and extra routing options between the blocks can help create a programmable architecture where it is possible to evaluate the performance of single blocks or only a part of the system. This routing options are mentioned in more detail in the next chapter.

3 Programmable Architecture Features

This chapter is intended to show the multiple configurations which can take place using the modified modules described in chapter 2, along with some extra modules for routing the signals. The fully programmable architecture is shown in Illustration 17.

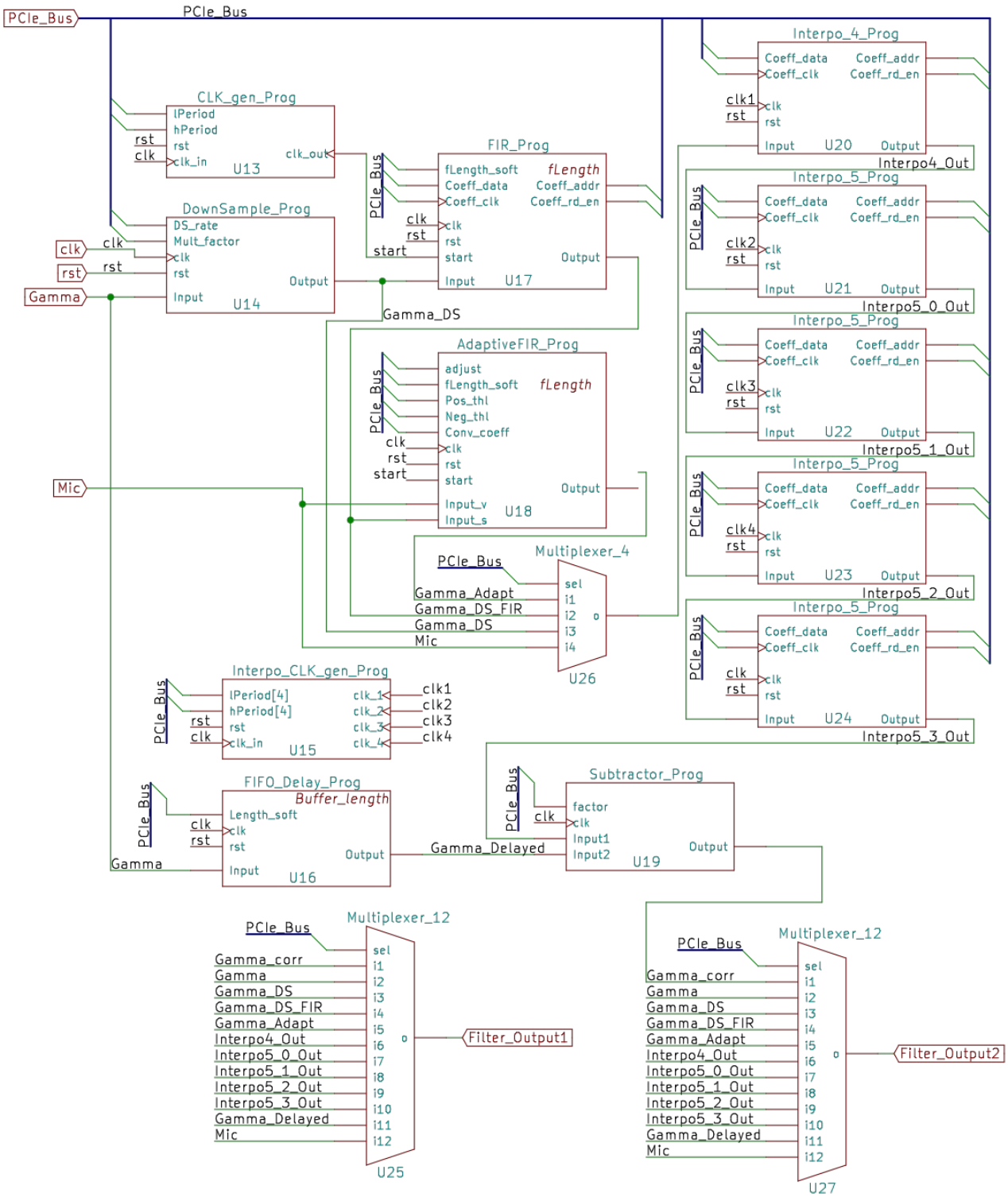

Illustration 17: Full Programmable Adaptive-Digital-Filter Architecture

Digital filters implemented in FPGAs can have some challenging features to manage, first, we have to make sure the fixed point logic can handle the multiplication of the coefficients and the later scaling down of the computed signal. Sometimes the input signals may have high frequency components which do not behave correctly when using fixed point multiplications and ultimately, experimenting with different filter coefficients may be a patience work while waiting between compilations to take place, which in bigger designs could take well around one hour.

The configurable architecture presented in the Illustration 17 is aimed to solving some of the problems that may arise when verifying the correct functioning of a digital filter. The architecture has been designed to be a playground where easy commands can change the overall output of the system.

This architecture could also be used as an educational platform. Different filter coefficients and filter arrangements could be configured withing seconds and signal evaluation could be done as fast as well. By connecting the two supplied outputs to the DAC chip in the ADA Board we have an immediate feedback to the user which can be visualized by means of an oscilloscope. Further in this chapter some of the possible configurations will be mentioned.

By looking at the Illustration 17 we can imaging many different configurations to be possible with the system. And with those we can think of being able of debugging individual pieces of it by selecting the right output. For instance, if we were to choose the input 3 (i3) in the multiplexer for output 1, we are selecting the output of the DownSample filter. And we will be getting an overall architecture as the one shown in Illustration 18.

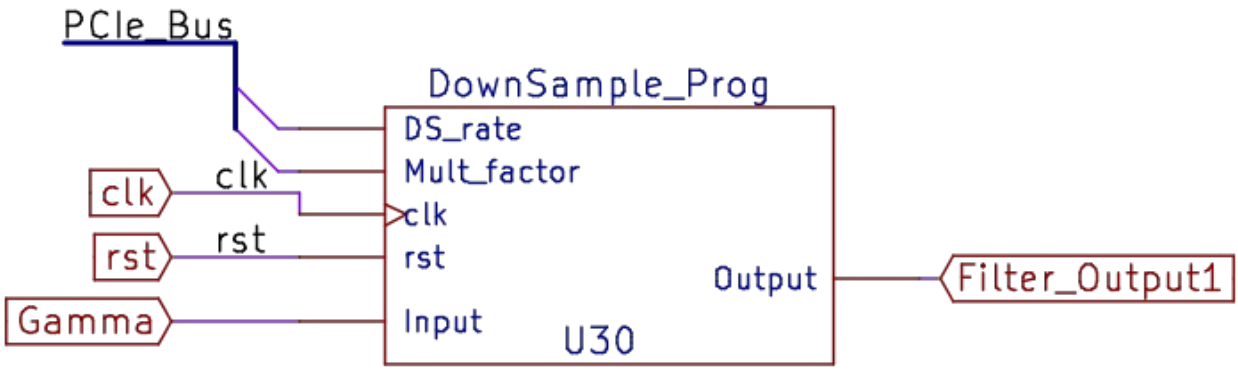

Illustration 18: Configurable DownSample Filter

By selecting the previous options we now have the ability to judge the working of the DownSample filter by it self, with no other intervention of the rest of the components in the system. The same way if we look further down the line. We could have an architecture to debug the working of the FIR filter, which is in charge of doing the high frequency passing. This configuration is shown in Illustration 19.

We could now have the ability to evaluate different filter coefficients and see the behavior instantaneous in the oscilloscope, we can quickly change the size of the filter within the limits and the values of the coefficients. With this, we can configure it to be a low-pass, high-pass and band-pass filter within seconds, counting that we have the filters already predesigned and ready to be loaded to the FPGA.

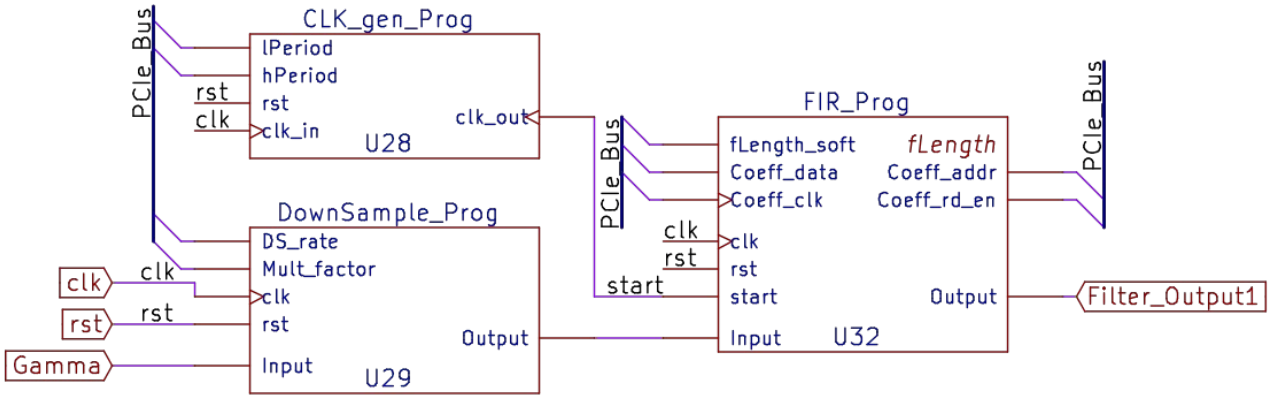

Illustration 19: Configurable DownSample and FIR Filter

In the case of the adaptive filter, it has been configure to have only read capability of the coefficients, therefore they are by no means affected by any external configuration. This, gives us the possibility to read the coefficients as they adapt over time at any moment, and enable us to have the ability to look deeper in the functioning of the filter. One could envision a simple software tool which could plot the adaptive filter coefficients as they adapt to see the evolution in time, and how they (hopefully) converge to a final value.

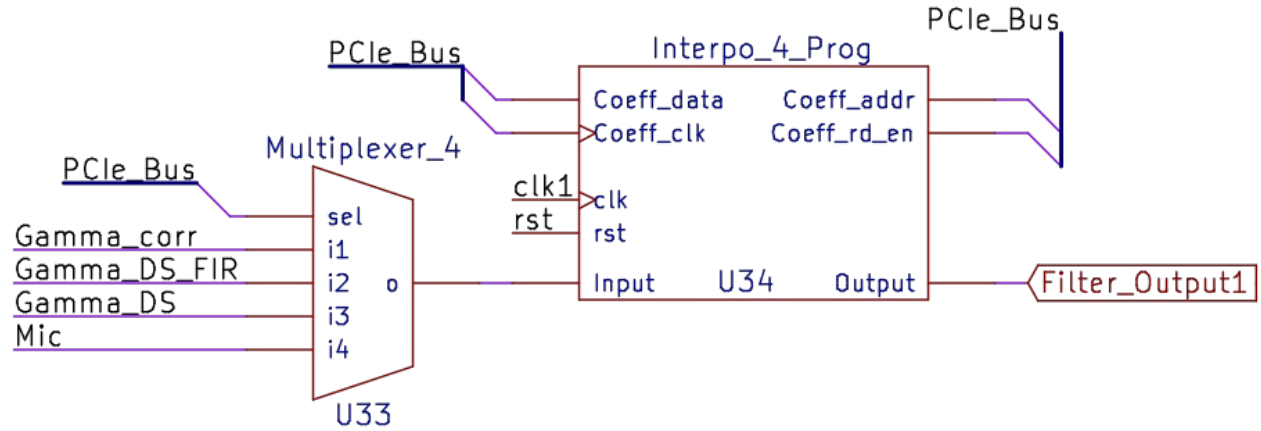

Illustration 20: Interpolation Input Selector

Another interesting feature is the ability to completely bypass the initial filter steps and go directly to the interpolation filter, This is shown in the Illustration 20, where one could analyze the inner working of the interpolation filter by itself. As well as looking the progression of the signal as it goes along the different interpolation stages. This could be selected with the final multiplexer of twelve positions shown in the Illustration 21.

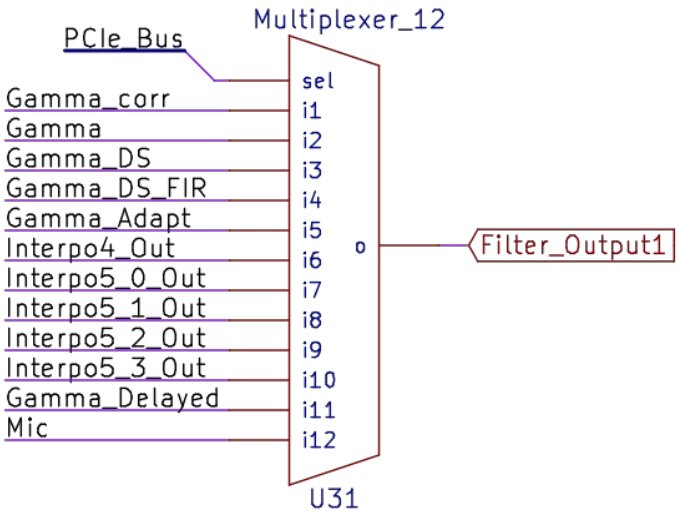

Illustration 21: Output Selector

With this final selector we can have access to the signal at any given point inside the architecture, allowing the designer to have full observability of the system and be able to identify possible problems like rolling over the limits or incorrect handling of the signed or unsigned signals, as well as possible multiplication or scaling error.

This programmable interface is aimed to be a playground for the designer, where he could easily verify the settings and behavior of each component of the filter. It also could be the baseline for more custom programmable architectures with other applications in mind. Each and every component can be sized and adapted to fit the requirements of other applications.

4 Hardware-Software Communication

To explain the communication details between the Hardware (FPGA) and the Software (CPU), it is necessary to understand the topology of our system, and how it is connected by means of physical wires and through which protocol. In the following section we have a small description of the DE2i-150 Development Board used in this project, which also has been described in REF.

4.1 The Altera DE2i-150 Development Board [11]

The board consist of a full featured computer with all the required components to successfully run Linux or Windows operating systems. In addition to this, there is an FPGA chip in the same board connected via a PCIe link to the processor. Thanks to this, we could envision systems with high data transfer rates between the two, where hardware acceleration is open for software implementations and also visualization tools available for hardware acquired signals.

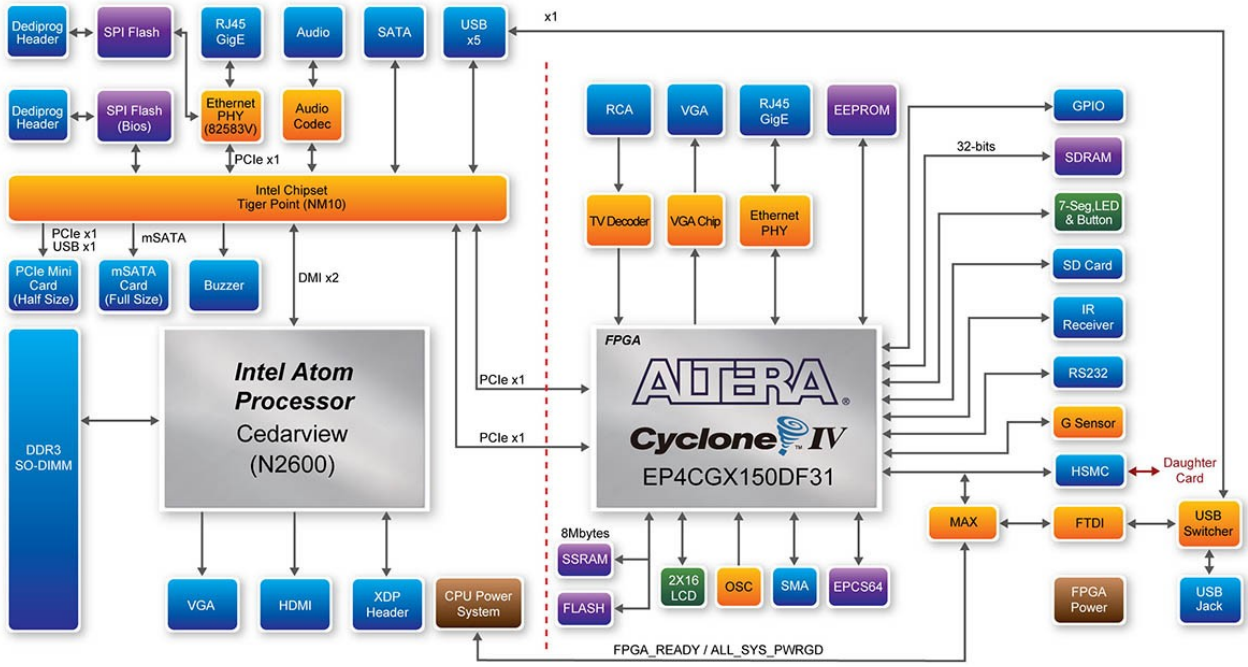

Illustration 22: Block Diagram Altera DE2i-150 Board [11]

In the Illustration 22 we can see how the board is connected with its different peripherals. The processor side we have important components such as the DDR3 memory block or the mSATA port for the solid state disk containing the operating system. We also have a port for a PCIe mini card with a WiFi module, an Ethernet module, five USB and a video connector complete the required features for having a standalone computer.

Also in Illustration 22 we can notice how the FPGA side of the board is designed and how it is

connected via the PCIe lane to the processor. We also have by default some useful components when we want to develop and quickly test different projects. Additionally, we also have the chance to add a mezzanine board with extra features of our interest.

For example, in this case we want to have a fast ADC board capable of sampling the signals at speeds of more than 50 MHz. The chosen board is capable of a sampling frequency up to 150 MHz with 14 bits of resolution.

4.2 The PCIe IP Core

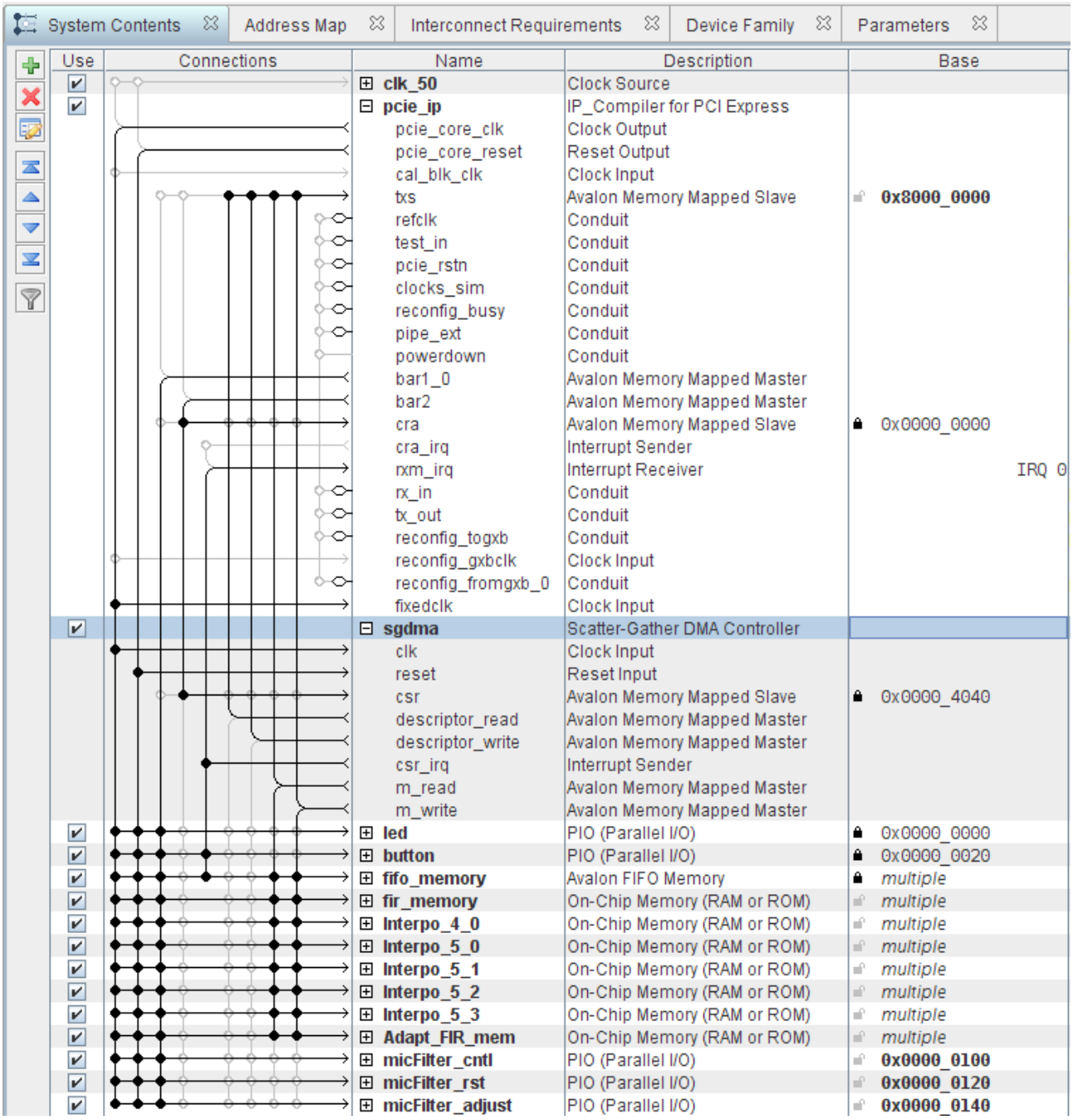

Illustration 23: QSYS System Configuration

In order to use the PCIe link between the processor and the FPGA one must declare and use an IP module from the IP cores library. This module is the one in charge of abstracting the first three layers for the designer. This module will take care of implementing all the required and most of the optional features of the transaction, data link and physical layer of the PCIe specification [12].

Another methodology instead of individually declaring the IP cores is to use the QSYS integration tool. This tool will take care of interconnecting all the declared higher level blocks with the necessary logic so that they can work together. In the Illustration 23 we can take a look at how this looks in the integration tool, here we have a declaration of the PCIe hardcore and some other modules such as the dual port RAM for the filter coefficients and some control registers.

Something important to notice is that the software side of the IP core will have some pre-specified settings that we need to specify in the IP module inside the FPGA. Such settings include the device information registers as shown in Illustration 24. Which have to be equal as the software object to control the device.

Illustration 24: PCIe device parameters

Due to its dual nature, the PCIe IP module must be declared also in the software applications that we write in the host computer. In Illustration 25 we can identify the different components of the system and how we should declare them. In the software side we must include two precompiled objects; a kernel driver and a shared object library. With the use of this two libraries we can do:

- Direct I/O for control and data transmission
- DMA for high speed data transmission

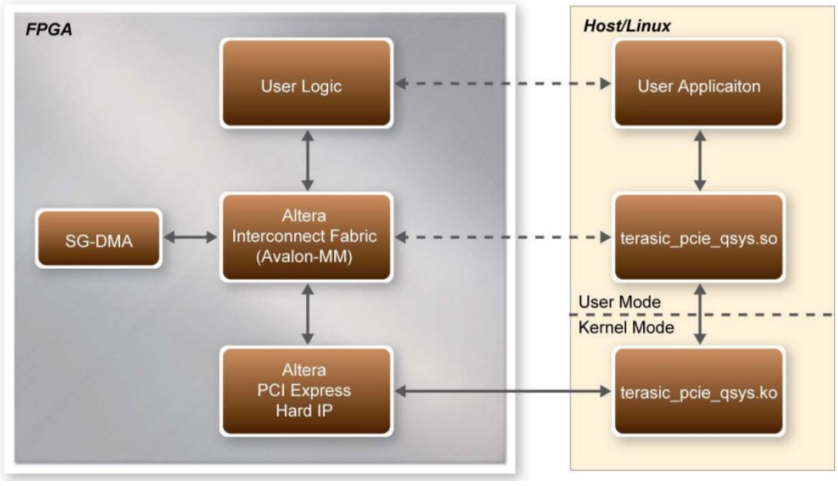

Illustration 25: PCIe system framework [11]

4.3 Software API

Inside the PCIe framework presented in Illustration 25 as part of the shared object (.so) library there is a set of functions that the user application have access to, which are mentioned in the following list.

Thanks to this libraries one could implement a variety of high efficient data transmission programs in which we feed the FPGA constantly with data, and it is capable of handling the data transmission seamlessly.

- ***PCIE_Close*** *(PCIE_HANDLE hFPGA);*
- ***PCIE_Read32*** *(PCIE_HANDLE hFPGA, PCIE_BAR PciBar, PCIE_ADDRESS PciAddress, DWORD *pdwData);*
- ***PCIE_Write32*** *(PCIE_HANDLE hFPGA, PCIE_BAR PciBar, PCIE_ADDRESS PciAddress, DWORD dwData);*
- ***PCIE_Read16*** *(PCIE_HANDLE hFPGA, PCIE_BAR PciBar, PCIE_ADDRESS PciAddress, WORD *pwData);*
- ***PCIE_Write16*** *(PCIE_HANDLE hFPGA, PCIE_BAR PciBar, PCIE_ADDRESS PciAddress, WORD wData);*
- ***PCIE_Read8*** *(PCIE_HANDLE hFPGA, PCIE_BAR PciBar, PCIE_ADDRESS PciAddress, BYTE *pcData);*
- ***PCIE_Write8*** *(PCIE_HANDLE hFPGA, PCIE_BAR PciBar, PCIE_ADDRESS PciAddress, BYTE cData);*
- ***PCIE_DmaRead*** *(PCIE_HANDLE hFPGA, PCIE_LOCAL_ADDRESS LocalAddress, void *pBuffer, DWORD dwBufSize);*
- ***PCIE_DmaWrite*** *(PCIE_HANDLE hFPGA, PCIE_LOCAL_ADDRESS LocalAddress, void *pData, DWORD dwDataSize);*
- ***PCIE_DmaFifoRead*** *(PCIE_HANDLE hFPGA, PCIE_LOCAL_FIFO_ID LocalFifoId, void *pBuffer, DWORD dwBufSize);*
- ***PCIE_DmaFifoWrite*** *(PCIE_HANDLE hFPGA, PCIE_LOCAL_FIFO_ID LocalFifoId, void *pData, DWORD dwDataSize);*

from the list above we can identify the first problem with this API; all functions are processor dependent, all of them work by polling the FPGA, and none of them have access to the interrupt service in the lower abstraction layers. This prevents us from implementing code based on interrupt routines generated from the FPGA.

By having a completely polling operation mode we need to design a system which would be able to gather data at a periodical way without missing any information. Therefore it is necessary to do data analysis inside the FPGA, produce a histogram of the detector hits amplitudes. Which would have a fixed number of bins, and therefore we can issue a PCIE_DmaRead request with a known buffer size (dwBufSize). This is an idea for the next step of this project, which could include some data analysis inside the FPGA and data visualization in the processor.

4.4 Transmission Speed

In the Illustration 26 we can observe the time difference between consecutive data transfers of 32-

bits per read request when we don't use the DMA for completing the transmission. Here we can see that it takes ~800 clock cycles between individual transmissions using the PCIE_API PCIE_Read32 function. This translates to 800 clock cycles at 100 MHz, this leads to a maximum data transfer of 4 Mbits/s. For a 14 bit transfer that is equivalent to only 285 kHz sampling frequency instead of 25 MHz or 50 MHz as intended.

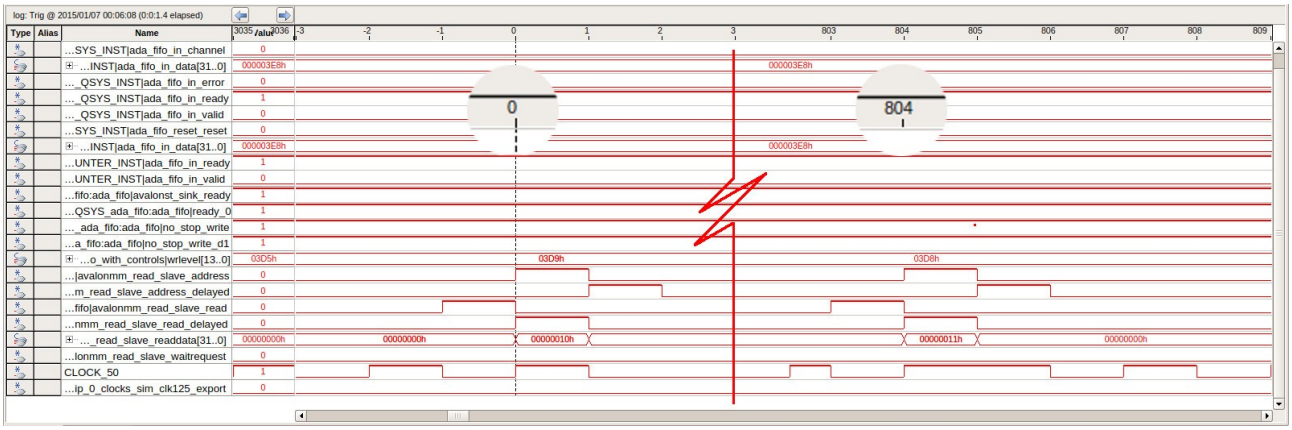

Illustration 26: PCIe Data Transfer without DMA

The data transfer using the DMA engine is much faster, it is capable of transferring 32 bits for every clock cycle of the system, if we have a 50 MHz clock as shown in the Illustration 23 then we are capable of filling the coefficient memory at 1.6 Gbit/s. In Illustration 27 we are observing how the retrieving of the coefficients is done by using the second port of the memory. In this case, we have a 25 MHz clock fetching the information of the coefficients so that they can be loaded to the internal variables in charge of the data calculation. The data transfer goes up to 800 Mbit/s a factor of 200 faster with also a possible sampling frequency of 57.14 MHz, making it in theory possible to fetch all data samples taken at 25 MHz or even 50 MHz.

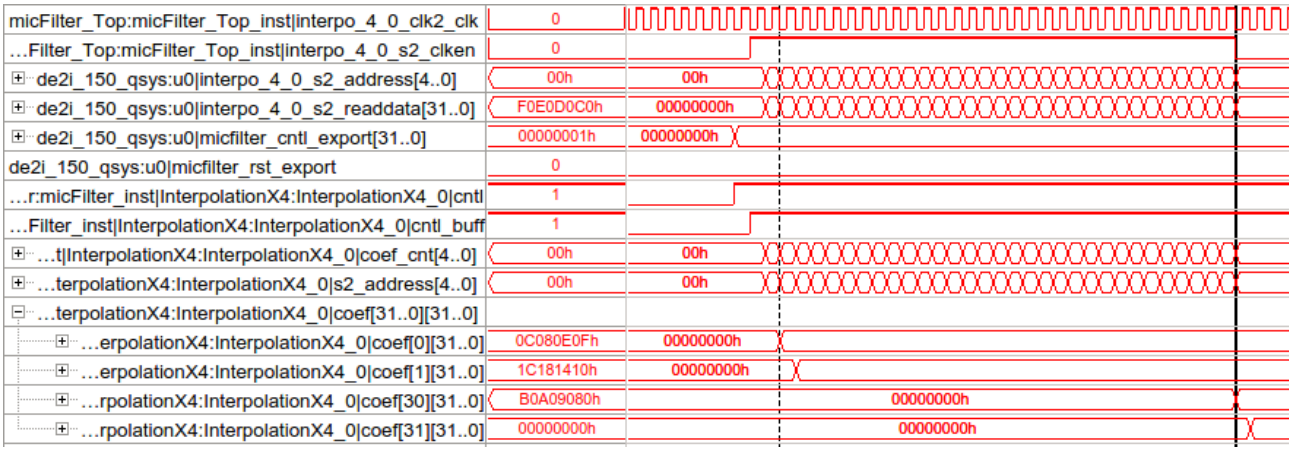

Illustration 27: PCIe Data Transfer with DMA

The DMA transfer is a big improvement with respect to the individual transfers, however it does not solve the problem of having to start every single communication from the CPU side, therefor, as we explained earlier, it is necessary to have a fixed size of the data to be transfer and therefore we could implement a regular poling of the data. The ideal scenario would be having a histogram built inside

the FPGA with the amplitudes of the signals with a fixed number of bins.

4.5 Software Tools

In section 4.3 we mentioned the API provided by Altera which we should use in order to implement our own functions, here, we describe briefly the purpose of each one of those functions which can be found in the appendix. For starting, we need some basic read file functions which will allow us to read the filters designed by other softwares and converted in simple .txt files, then by using the provided API, those values are to be send to the FPGA by means of the DMA transmission.

4.5.1 Filter Generation

Each one of the filters are design by external tools which can have also many different ways of outputting the coefficients of the filters, therefore it is necessary to define a common file format which would be used to store all the filters to be sent to the FPGA. In this first stage, we need a small program which is capable of translating the coefficients designed by Matlab for example, into this common file format which is shown in the Text 1.

The filter generation then can take place in many different software platforms, and therefore not limiting the designer to a certain tool. It is known that Python has a very interesting environment for signal processing design with the simple inclusion of some libraries like SciPy [13], and therefore making it a perfect alternative to the filter design by Matlab.

In both cases, it is not hard to design a small extra code which translate the coefficients in those native environments to a plain .txt file in the form of the defined format.

```
0    0x03020100
1    0x07060504
2    0x0B0A0908
3    0x0F0E0D0C
4    0x13121110
5    0x17161514
6    0x1B1A1918
7    0x1F1E1D1C
8    0x23222120
9    0x27262524
...
```

Text 1: File Format for the Filter Coefficients

4.5.2 File Reader

Every line of the file contains the address expressed in decimal and then, the value for that address express in hexadecimal with a width of 32 bits.

In this way, the software in charge of reading the file knows which kind of format to expect and therefore sends the correct coefficient to the correct address. Due to the fact that the data is transfer

by DMA, we must ensure that the addresses are store in a consecutive way, this format provides also a way to check the correct order.

Then, once the data is correctly read and stored in the CPU RAM, we can make use of the API by Altera to simply push the data by means of PCI DMA to the FPGA.

Inside the FPGA we can verify that the transfer has been done successfully by programming a SignalTap core sensing the address and the data lines of the dual port RAM. This is shown in Illustration 27.

4.5.3 Python Visualization

One of the section 2.3.2 we mentioned how it is possible to access the filter coefficients from the adaptive filter at any given time, and how interesting it would be to have a plot of them overtime showing the convergence speed of the filter. This kind of analysis would be more suitable to be implemented in a higher language such as Python, where there are already some libraries which help in this type of data analysis.

We should also mention the plotting capabilities of Python libraries such as matplotlib [14] and pyqtgraph [15] making it a perfect choice for this type of implementation. In order for it to take place, we would need to compile the current C functions into shared libraries and those could be called and used inside a python script.

This script will have the data at hand to produce graphs and visualization charts as well as sending the filter coefficients to the FPGA via the C implementation. This, however is not yet implemented, but it is a road map for further development concerning the software layers.

5 FPGA Implementation

In order to implement the configurable digital-adaptive-filter inside an FPGA we have to consider the extra components and how they might impact the resource usage. We only have a limited amount of multipliers and logic cells, and even more critical in this design, we are cutting very close to the maximum possible memory allocation due to the fact that we are using memory components in basically every module for the storage of the coefficients as well as for the delay.

5.1 Resources Usage

In the Table 1 we can see a summary of the total resources usage by the complete implementation of the fully configurable architecture described in previous chapters. There, we can see how the utilization rates are not that critical except for the memory.

The implementation of large amounts of fast memory inside FPGAs is reserve for special very large dies, which may cost something in the range of 10,000 USD, it is usual to have a very limited amount of internal memory, and in this case we are using up to 82 % of it.

Family	Cyclone IV GX
Device	EP4CGX150DF31C7
Total logic elements	88,832 / 149,760 (59 %)
Total combinational functions	61,109 / 149,760 (41 %)
Dedicated logic registers	54,165 / 149,760 (36 %)
Total registers	54165
Total pins	467 / 508 (92 %)
Total memory bits	5,431,786 / 6,635,520 (82 %)
Embedded Multiplier 9-bit elements	168 / 720 (23 %)
Total PLLs	2 / 8 (25 %)

Table 1: Total Resources Required for the Configurable Digital-Adaptive-Filter Design

5.1.1 Memory Bits

In our design, memory plays an important role. It is used for the storing of the coefficients for every single filter that we have implemented. The interpolation blocks, the high-pass FIR, the adaptive FIR and even the clock generation have memory blocks associated with them. The reason is that we have a large number of settings which should be stored in memory so that the system can get its configuration properties from.

However, it is important to analyze the memory usage per module, so that we can make an estimation of which part of the design is the critical block and how can we reduce or control the further incrementing of this need.

Looking at the project reports by Quartus, we found that the first contributor for the memory usage is the delay block. It is implemented as a circular memory, or FIFO, it has a contribution of 4,762,114 memory bits, counting for 88 % of the total bill. This is the case because we are saving every measurement taken at 25 MHz of the detector signal, and having it delayed until we are able to subtract the adapted noise from it. The increased sampling rate is responsible for making this module a memory hungry implementation. There is little the designer can do about this problem, and it might be the limiting factor number one for larger implementations with multiple inputs.

It is then necessary to implement an external memory with large capacity and fast speed, like a DDR memory, in this way we have no more restrictions on this mater, but we then introduce an extra component for the hardware design. This might impact the overall cost and complexity of the system. We then need to choose a large and fast enough FGPA capable of communicating with this type of memory.

The second contributor is the in-system analysis tool SignalTap, which is consuming 425,984 memory bits (8 %). This we could consider to be free resources as it is not intended that this module will be present in final implementations, however it is important to have it in order to effectively debug the system when needed.

Finally, the last contributor is the integration tool QSYS where all the memories for the filters are declared and all the configurations for each block are stored, Then we can say that the resource usage by the coefficients storing and configuration registers is not that much compared to the ring buffer implemented.

5.1.2 DSP Modules

Surprisingly, the usage of the DSP modules is quite low at only 23 %, this may be due to the fact that all calculations are taken in a sequential order and therefore using the same multiplier to calculate the result of an entire filter. This approach takes advantage of the decimation rate and allows the system to even have periods of time where no calculations are taken place while we wait for the next sample.

The module that uses the most DSP blocks is the interpolation module, with 32 DPS blocks per interpolation stage. This is due to the fact that we are having 8 multipliers working in parallel to calculate each output.

Then, we have four DSP blocks by the high-pass FIR and also four by the adaptive filter. There it is interesting to see that even though we are having two identical multipliers declared, only one got synthesize as a DSP block and the other as logic elements. This is sometimes dependent on the synthesizer if we are not declaring the embedded multiplier explicitly in the code.

6 Results

An improved configurable architecture has been successfully designed and implemented. This architecture is based in the previous work done by T. Hasenor [1], C. Pfeiffer [4] and myself L. Ardila [5]. This architecture has been designed so that we solve some of the difficulties that previous developments had. For example, it was expressed the difficulty to effectively test different configurations of the system due to a variety of reasons; not enough statistics from stored signals, process speed of the filter using Matlab, and time consumption in the compilation of different coefficients limiting the testing ability.

To tackle the lack of statistics when evaluating the filter, it is necessary that we are able to obtain signals from a high speed ADC integrated to the inputs of the filter. Additionally, with the implemented architecture we are able to diagnose the system in stages and therefore verifying its correct functioning, this is shown in Illustration 27.

Another very important feature which has been implemented is the ability to read and write the filter coefficients at any given time. This feature combined with the observability of the system is key in assuring the designer the optimal behavior at any given point inside the filter. We are able to provide known input signals to the filters and see the output behavior, and therefore characterizing the real performance of each component of the filter.

The final implementation of the design has been the combination of two efforts, the adaptive filter architecture proposed by C. Pfeiffer [4] and the hardware specific modules and communication facilities done by L. Ardila [5]. In the first, we have a complete filter designed entirely in VHDL, and in the second we have a design written only in Verilog. Therefore the final implementation is a top module in Verilog with the required functions for the PCIe communication, the acquisition of new data from the ADC, and an instantiation of the VHDL block. This multi-language implementation is normal to happen in large designs with multiple engineers involved and the correct interface between the two has to be carefully implemented.

Along with the FPGA description by HDL languages, it has been programmed a basic software tool in C. This tool has the goal to effectively communicate the FPGA with the CPU by means of the PCIe link which is given by the DE2-150 Dev Board. By using this provided capability we are guaranteeing a fast communication link between the two and also reducing the need for an external cable to connect them. The software tool is able to read and write the coefficients from the various filters inside the FPGA individually. We are able to read simple .txt files containing the values for each tab of the filters.

Overall, it has been implemented a fully configurable architecture which gives us the possibility to evaluate different components of the filter individually, therefore reducing the debugging time. It also provides a deeper degree of observability and allows the designer to have detailed information of the current state of the filter.

7 Future Work

The current design is a stepping stone for the optimal filter design for microphonic noise cancellation in Germanium detectors. Now, the user has at its hands a fully adjustable platform ready for experimentation with the detector. It is necessary to design an initial set of filter coefficients and configurations which could be used as a starting point for testing and debugging of the quality and performance of the system as a whole.

This architecture due to its programmability is suitable to be implemented in other types of applications, for instance we could think of replacing the decimation and interpolation modules and simply having arrays of FIR filters along with delays and adders to implement wavelet transformations and decompositions, image processing algorithms based on FIR filters and FFT modules. All of which would be benefit by the reconfigurability of the filter coefficients by software.

Regarding the software, we could think of having an effective way of visualizing the data and controlling the system, this could be done in Python. The filter design, and the data visualization are features which could be easily implemented by the declaration of libraries [14,15], and therefore the effort in the implementation is greatly reduced.

Finally, and more important, it is imperative that the detector signal is correctly handled by the front-end electronics and we are able to have it within the ADC range of the expansion board, this means ± 3 V to ± 500 mV. In this way, there is no other limitation for start taking data from the detector and analyzing the behavior of the filter step by step as well as a whole.

1 - *T. Hasenor. Bachelor's thesis: Active microphonic noise cancellation in readiation detectors. Karslruhe University of Applied Sciences, 2014.*

2 - *V. Moeller-Chan, T. Hasenohr, T. Stezelberger, M. Turqueti, S. Zimmermann: Microphonic Noise Cancellation in Radiation Detectors Using Real-Time Adaptive Modeling*

3 - *S. Zimmermann, N. Abgrall, M. Bantel, V. Moeller-Chan, M. Cromaz, C. Grace: Development of Next-Generation Nuclear Physics Integrated Readout Electronics for GRETINA*

4 - *C. Pfeiffer. Master thesis: Hardware-implemented Active Microphonic Noise Cancellation in Radiation Detectors. Karslruhe University of Applied Sciences, 2015*

5 - *L. Ardila. Project A: Adaptive digital filter design and its implementation on Altera DE2i-150 board. Karslruhe University of Applied Sciences, 2015*

6 - Duncan Buell, Tarek El-Ghazawi, Kris Gaj, Volodymyr Kindratenko, "Guest Editors' Introduction: High-Performance Reconfigurable Computing," Computer, vol. 40, no. 3, pp. 23-27, March, 2007

7 - *Xilinx: (1993), “Data book”, XC4000*

8 - *J. Treichler, C. Johnson, M. Larimore: Theory and Design of Adaptive Filters (Prentice Hall, Upper Saddle River, New Jersey, 2001)*

9 - *Mathworks. intfilt function, April 2015. URL http://de.mathworks.com/help/signal/ref/intfilt.html?searchHighlight=intfilt.*

10 - *Iowegian International. dspguru - interpolation, April 2015. URL http://www.dspguru.com/dsp/faqs/multirate/interpolation.*

11 - *Terasic DE2i-150 Development Kit FPGA System User Manual, 2013*

12 - *Altera: “IP Compiler for PCI Express – User Guide”, 2014*

13 - *http://docs.scipy.org/doc/scipy/reference/signal.html*

14 - *http://matplotlib.org/*

15 - *http://www.pyqtgraph.org/*