Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2007

MSc THESIS

High speed reconfigurable computation for electronic instrumentation in space applications

Dimitrios Lampridis

CE-MS-2007-20

Abstract

Small, light structures, with low power consumption are the key to success for future electronic instrumentation in space applications. Future missions will have to rely on lighter payloads to reduce the costs, and higher levels of integration to put more instruments in the confined space of a small spacecraft. At the same time, recent developments in the space industry have introduced radiation-hardened FPGAs, making one step forward towards the use of re-programmable hardware in space, and the European Space Agency is actively promoting System-on-Chip (SoC) design methodologies for future highly-integrated space electronics. In this thesis, we set off to investigate the benefits of a managed SoC approach in future space electronic instrumentation. To this end, we study a digital pulse detector. Such an instrument is often found on-board on planetary exploration spacecrafts, because of its two-fold role: its primary function is to monitor the radiation levels of the spacecraft's environment, but it can also classify the detected radiation pulses to perform γ-ray digital spectroscopy. The pulse detector is designed as an AMBA IP core that can be interfaced to many SoC libraries. We perform all processing associated to pulse detection, including shaping, pulse height determination, and pile-up rejection, in real-time with zero dead-time. To achieve this, we use a shallow-pipelined serial design, with a spesialised computational block for digital trapezoidal shaping, based on Carry-Save Adder reduction trees, followed by a custom peak detection algorithm. Our pulse detector is able to maintain a constant high throughput and low latency, independent of the number of samples under consideration. Additionally, the entire pulse detection mechanism is supervised by the on-chip processor through embedded software. We developed a prototype using a Xilinx XC3S1500 FPGA, the LEON3 on-chip processor and a set of IP cores from the GRLIB library. An external 8-bit ADC and the pulse detector were clocked at 100MHz, while the rest of the system was running at 40MHz. Preliminary experimental results obtained with our prototype are very promising and demonstrate the correctness of the design.

Faculty of Electrical Engineering, Mathematics and Computer Science

High speed reconfigurable computation for electronic instrumentation in space applications
empty

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Dimitrios Lampridis
born in Athens, Greece

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

High speed reconfigurable computation for electronic instrumentation in space applications

by Dimitrios Lampridis

Abstract

Small, light structures, with low power consumption are the key to success for future electronic instrumentation in space applications. Future missions will have to rely on lighter payloads to reduce the costs, and higher levels of integration to put more instruments in the confined space of a small spacecraft. At the same time, recent developments in the space industry have introduced radiation-hardened FPGAs, making one step forward towards the use of reprogrammable hardware in space, and the European Space Agency is actively promoting System-on-Chip (SoC) design methodologies for future highly-integrated space electronics. In this thesis, we set off to investigate the benefits of a managed SoC approach in future space electronic instrumentation. To this end, we study a digital pulse detector. Such an instrument is often found on-board on planetary exploration spacecrafts, because of its two-fold role: its primary function is to monitor the radiation levels of the spacecraft's environment, but it can also classify the detected radiation pulses to perform γ-ray digital spectroscopy. The pulse detector is designed as an AMBA IP core that can be interfaced to many SoC libraries. We perform all processing associated to pulse detection, including shaping, pulse height determination, and pile-up rejection, in real-time with zero dead-time. To achieve this, we use a shallow-pipelined serial design, with a spesialised computational block for digital trapezoidal shaping, based on Carry-Save Adder reduction trees, followed by a custom peak detection algorithm. Our pulse detector is able to maintain a constant high throughput and low latency, independent of the number of samples under consideration. Additionally, the entire pulse detection mechanism is supervised by the on-chip processor through embedded software. We developed a prototype using a Xilinx XC3S1500 FPGA, the LEON3 on-chip processor and a set of IP cores from the GRLIB library. An external 8-bit ADC and the pulse detector were clocked at 100MHz, while the rest of the system was running at 40MHz. Preliminary experimental results obtained with our prototype are very promising and demonstrate the correctness of the design.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2007-20

Committee Members :

Advisor:	Sorin D. Cotofana, CE, TU Delft
Member:	Rene van Leuken, CAS, TU Delft

To someone...

Contents

List of Figures

List of Tables

Acknowledgements

This text is the result of more than nine months of work that was carried out as partial fulfillment for a degree of Master of Science in Computer Engineering. During these months I have spent most of my time at cosine Research BV in Leiden (yes, there is no capital letter in the word "cosine"), and I would like to take the opportunity to acknowledge all the support that I received while doing my research at their premises.

I would like to thank in particular Alex Palacios and Erik Maddox, my on-site supervisors. Alex has been a tremendous help during the course of this work, and he offered invaluable advice in the design of the electronics. Erik on the other hand, provided all the necessary background in γ-ray spectroscopy, and made me appreciate a bit more the field of high-energy physics. Both of them were always there when I needed them, for technical support, but also as friends.

I would also like to thank Stefan Kraft, general manager of cosine Research BV, for giving me the chance to work on this project and placing his faith on me, as well as providing support and advice in critical moments of the project. Last but not least, I would like to thank every single employee of the company for making me feel like home from the very first day, and accepting me as equal among equals.

My gratitude goes to my dear professor and advisor Sorin Cotofana, for supporting my decisions all the way, even when I contradicted myself. I do believe that his patience with me has been ...monumental. Thank you Sorin for letting me do my own, but also for pushing the "right buttons" to keep me on track.

Having said that, I would have never reached my goal if it wasn't for the unconditional love of my family and my beloved girlfriend, so my greatest "thank you" goes to them, for simply making me feel alive.

As far as the dedication is concerned, it was while I was working on this project, that the Computer Engineering laboratory of TU Delft suffered the very sudden loss of our beloved professor Stamatis Vassiliadis. Stamatis has been a great man and scientist, a true inspiration from the first time that we met. I would have promptly dedicated my work to his memory, but I believe that he deserves much more than my master thesis. I prefer to leave the dedication empty, and honour him in my own way.

Dimitrios Lampridis
Delft, The Netherlands
November 20, 2007

Introduction 1

Small, light structures with low power consumption are the key to success for electronic instrumentation in space applications [19]. Lighter payloads reduce the mission costs and allow us to put more instruments in the confined space of a small spacecraft.

In the future, the ever-increasing demands for high processing performance, low mass and power on board the spacecraft, will demand for very high integration levels of instrumentation and electronics. Following the current trends in the rest of the electronics industry, as the number of available resources on silicon increases, the design of electronic instrumentation for space applications will have to move away from the use of traditional components to more advanced and complex systems within a single device [17].

To develop such complicated multi-functional systems the design methodology will have to change from being gate-level oriented to the integration of complex building blocks, with verified functionality. These blocks should also be accompanied by detailed documentation and testing methodology.

As the number of instruments on-board the spacecraft increases, so does the amount of data generated during the mission. It is highly unlikely that the slow down-link from the spacecraft to Earth will be able to transfer all this information in time. Furthermore, apart from their scientific purpose, many of the on-board instruments play a vital, safety-critical role in the spacecraft's reaction to the environment and need real-time processing. The solution is to do on-board processing, in order to reduce the amount of output data, and to respond faster to environmental changes.

System-on-Chip (SoC) approaches offer a small, light single-chip solution, fitting the above-stated requirements of high integration, managed design methodology with large building blocks, and on-board processing. An SoC is usually developed using Field Programmable Gate Arrays (FPGAs), but the final product can also be manufactured into an Application Specific Integrated Circuit (ASIC).

With the capacity and performance of FPGAs increasing every year, and the manufacturing costs of an ASIC still very high, it is becoming a popular solution to actually fly the FPGAs in space, instead of just using them during design development only. The role of the FPGAs is also rapidly changing, from simple "glue" logic between other silicon chips, to a complete SoC, comprising of processors, peripherals, memories, and dedicated hardware.

Reprogrammable FPGAs offer a new dimension for space applications, because they allow the modification of on-board electronics during the mission. Possible utilizations of the FPGA reprogramming ability include replacing of faulty design modules, updates to processing algorithms, adaptation to new mission requirements, and switching to different operation profiles, optimized for area, power, performance or a combination of the above.

Programmable hardware has been flying on board spacecrafts for more than a decade.

However, most of the FPGAs used are still one-time programmable, because reprogrammable FPGAs are more sensitive to involuntary reconfiguration due to Single Event Upsets (SEU) induced by radiation [7]. The space environment is very hostile, and high amounts of radiation can cause bit flips in memory elements. This poses an additional threat to the on-chip configuration memory of reprogrammable FPGAs.

Recent developments in the defense and space industry have introduced radiation-hardened FPGAs, making one step forward towards the use of reprogrammable hardware in future space missions. Space applications present new challenges in the use of reconfigurable computing, particularly due to the effects of incident radiation, and a new field is emerging to provide answers and solutions.

In this thesis, we set off to investigate the benefits of a managed SoC approach in future space electronic instrumentation. For our investigation, we choose a high-count digital pulse detector. Such an instrument is often found on board planetary exploration spacecrafts, because of its two-fold versatile role: its primary function is to monitor the radiation levels of the spacecraft's environment. The system continuously monitors the levels of the detected radiation pulses, and sends out an alarm signal to the spacecraft when the irradiation becomes too intense and threatens the spacecraft's integrity. This kind of application is time-critical and demands for high responsiveness. With minor modifications, namely the recording and classification of the detected pulse heights, the instrument may also be extended with the secondary function of γ-ray digital spectroscopy. This kind of spectroscopy is very popular in planetary missions, because γ-ray sensing is an established technique to study the composition of the outer layers of planets.

We follow a reconfigurable approach, suitable for SoC design, to process the digitized signals and calculate the pulse height. The digital pulse detector is designed as an AMBA [9] IP core that can be combined with other cores (processors, memories, other peripherals) into a single SoC solution. We keep the computational part of our design separated from the AMBA interface, within different clock domains, to remove the need for matching the speed of the computation with that of the interconnection bus, and to maximize the reusability of the IP core.

Our approach combines the high performance of dedicated computational hardware, with the flexibility of a complementary on-chip processor, to produce a complete, compact solution for future electronic instrumentation in space applications. The only parts of the system that are external to the FPGA are the analog pre-amplification and the Analog-to-Digital Converter (ADC).

Inside our pulse detector IP core, linear trapezoidal filtering is applied to the digitized samples. The filters can keep a high and constant throughput, independent of the number of past samples under consideration. The filtered output is further processed by a smart on-line peak detection algorithm that discards false events and pile-ups. Both the filter and peak detector parameters are fully configurable via the AMBA APB interface. The detected peaks are transferred back to the on-chip processor, where the embedded software creates pulse height histograms and transmits them back to base.

The design was implemented in IEEE-compliant VHDL, without any manufacturer-specific hardware structures and macros, allowing us to synthesise and place the IP core into any of the available FPGAs. For experimental purpose we programmed the resulting bitstream on a Xilinx Spartan3 XC3S1500 FPGA, using the 32-bit LEON3 Sparc V8 compatible [6] synthesisable processor, together with a minimal set of AMBA IP cores from Gaisler Research [1]. The ADC and filter/peak detector are clocked at 100MHz, while the rest of the system is running at 40MHz. The system was configured and controlled using our own software, written in C and cross-compiled for the Sparc architecture.

The remaining of this thesis is a detailed discussion of our proposal, experimental setup, and obtained results, structured as follows: in Chapter 2 we present popular methods for digital pulse processing that we also make use of, like trapezoidal filtering and dual filter setup. Near the end of Chapter 2, we provide an overview of related work in the field of digital pulse processing. Armed with the necessary background knowledge, we focus next on the system architecture of our digital pulse detector, presented in Chapter 3. In Chapter 4, we discuss the experimental setup with the LEON3 processor and the GRLIB library, which we used to test our design. We also go briefly over the various tools that we used during development, and present our preliminary results, which we obtained with thay setup. Finally, our work concludes in Chapter 5, with a discussion of the problems that we encountered during development, the lessons learned, and an overview of the numerous possibilities for further work on the subject.

2 Background & Related Work

Over the past 15 years, the introduction of fast Analog-to-Digital Converters (ADCs) in ever-increasing speeds has brought digital processing into fields that used to be dominated by analog solutions. It was not long before the research community and the industry came up with a variety of digital solutions for pulse detection and spectroscopy. Today, with low-power ADCs able to convert an analog input into several million samples per second, the range of solutions spans from purely analog to almost completely digital (apart from source conditioning circuits and the ADC itself).

Although pulse detectors may exist in many different flavours, they often operate under a common idea: a triggering system detects the pulse, signalling a second stage that calculates the height of the pulse, or some linear function of it. In this chapter we provide details on how we have chosen to implement these mechanisms in our design. We use well-tested and compact methods, based on their suitability for fast on-line filtering and peak detection.

In the last part of this chapter we present an overview of related work in the field. We do not attempt to perform an in-depth analysis of the benefits of digital processing over traditional analogue methods in high-count pulse detection and spectroscopy. We refer the interested readers to [24], [23] and [25] for a more detailed discussion on the subject. Instead, our intent is to offer the reader a better understanding of the context of this work, and to define the aspects that make our design unique.

2.1 Trapezoidal Filtering

Many digital pulse detectors use triangular and/or trapezoidal functions to filter their input. The reason in that relates to the fact that those functions are easy to understand and implement in digital logic. Triangular weighting is useful for very fast detector channels, while trapezoidal weighting is preferred for slow, good resolution channels. If the flat-top of the trapezoid shape is set to zero, the function is identical to triangular weighting. Therefore, the same trapezoidal function with different parameters (weights) can be used for both channels. When compared to Gaussian shaping, a trapezoidal function has comparable resolution but needs less processing time [13]. These facts make trapezoidal functions a good all-around choice for high-speed digital spectrometers.

A trapezoidal function considers two data-sets (windows) of input samples at a time. Between the two windows exists an optional “gap”, represented by the flat-top of the trapezoid. Figure 2.1 illustrates a radiation-induced pulse event and the segmentation of its digitised samples in windows. Both windows must have the same width, and the sum of window widths and possible gap must not exceed the filter’s sample memory.

Every clock period, the filtering function averages the samples inside each window and subtracts the two resulting sums. That is, if $O[n]$ is the output of the filter at time

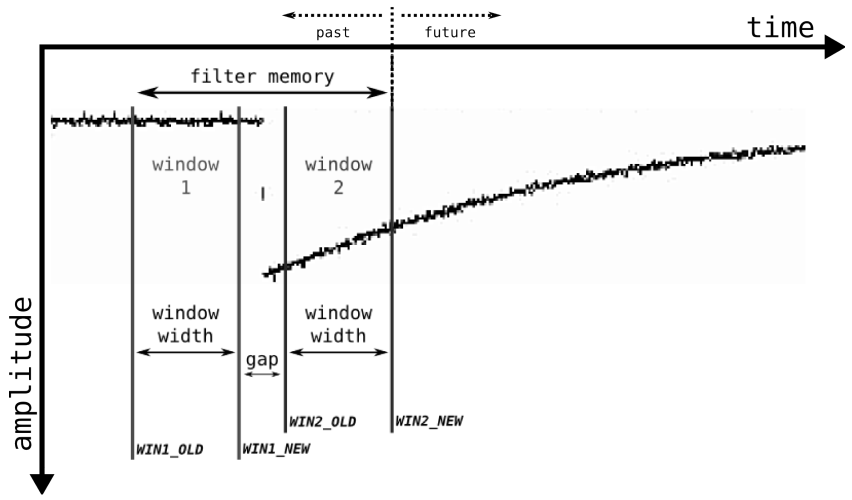


Figure 2.1: Segmentation of input samples for trapezoidal filtering

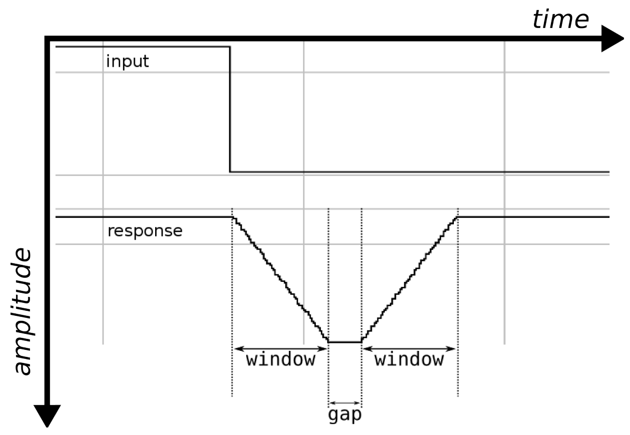
unit n, $I[n]$ is the input of the filter at the same moment, w is the window width, and g is the gap, then:

$$O[n] = \frac{1}{w} \times \left[\left(\sum_{k=n-w}^{n} I[k] \right) - \left(\sum_{k=n-2w-g}^{n-w-g} I[k] \right) \right]$$

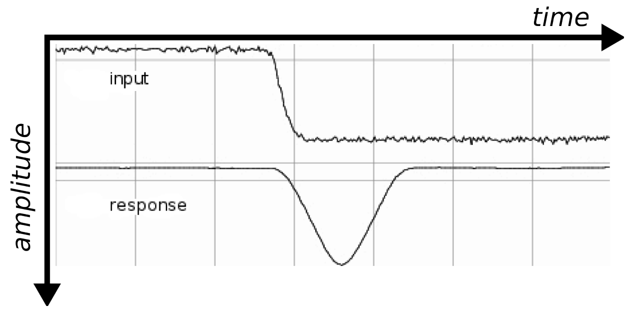
Figure 2.2a depicts a simulated example output of the trapezoidal filter, given an inverted step function as input. We can see that as the step function samples flow through the first window, the output amplitude increases monotonically. Then as the samples continue through the gap, they create a flat-top. Finally, samples going through the second window cause the output amplitude to decrease monotonically. The resulting shape is a perfectly symmetrical (as long as the two windows have the same width) trapezoid shape, hence the name of the filter. It is also interesting to see how altering the filter parameters affect the output shape: bigger window widths reduce the slope on the sides of the trapezoid, while a bigger gap increases the width of the flat-top.

Figure 2.2b presents again a simulated output of the filter, this time for a realistic radiation trace as input (zoomed in at the time of an event). These trace samples were captured with a digital oscilloscope and used as input to the simulation. The response is a symmetrical bell-like shape, a fact that will later simplify the process of calculating the maximum reached height. Another important aspect is of course the evident reduction in noise, a result of the averaging function of the filter.

In the above examples, the input samples were driven into the simulated IP core at a rate of 100 MHz (one sample every 10 ns), and filtered with a window width of 48 samples and a gap of 16 samples, resulting in a 480 ns window and a 160 ns gap. For best performance, the gap should always be greater than the event rise time of the input.



(a) Step function input



(b) Radiation trace input

Figure 2.2: Simulated filter response for various inputs

2.2 Dual-Channel Filters

Modern pulse detectors, both digital and analog, process their input using two channels simultaneously. A “fast” channel is used to detect incoming particles, while a “slow” channel takes more time to evaluate, in order to extract high-resolution information about the pulse height. In this setup, an event in the fast channel acts like a trigger signal for the slow channel. Using this scheme, we combine the quick response of a fast filter, with the improved resolution of a slower filter. In our implementation, we use two identical trapezoidal filters, one per channel, with different parameters (the fast channel uses a smaller window width).

Depending on the filter parameters, it might be that more pulses arrive while the slow channel is still evaluating the first coming one. In that case, we can use the fast channel to detect multiple input events while the slow channel is still evaluating (pile-up rejection) [25]. It follows that the parameters of the fast filter should be chosen based on the expected behaviour of the events we wish to detect. Ideally, the fast filter parameters should be small enough to not allow any pile-ups while the fast filter is still evaluating.

Figure 2.3 depicts the output of such a dual-channel configuration: From top to bottom, we have the input signal followed by the slow and fast channel responses. We can see how the input events A and B are captured by both channels, but the arrival

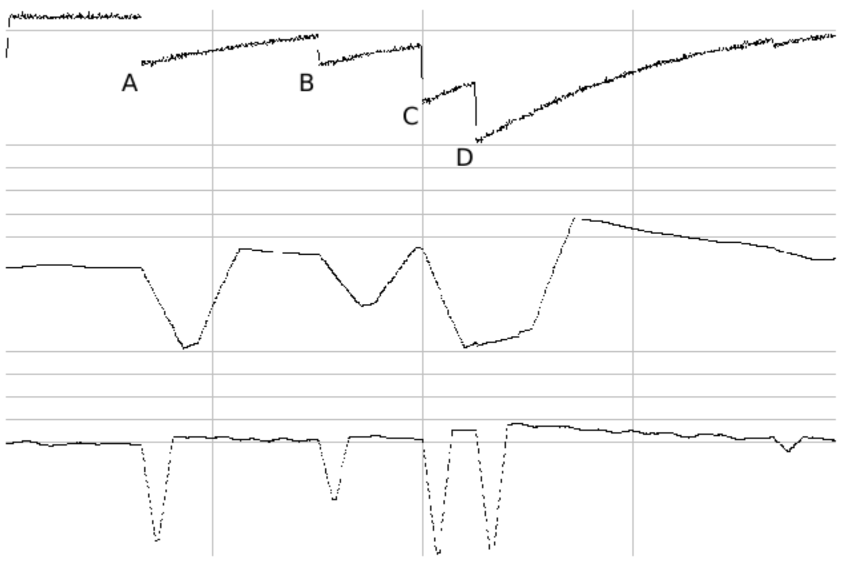


Figure 2.3: Pile-up inspection using dual-channel filtering

times of events C and D are close enough to cause a pile-up on the slow channel. The fast channel on the other hand has no problem to detect all four distinct events. We can use this combined information to register the first two clean events and reject the third one as pile-up. Pulse height determination is performed using the enhanced resolution values of the slow filter.

2.3 Related Work

We have discussed the basic common steps involved in pulse detection, popular methods for digital pulse shaping and discrimination, as well as the increasing trend of digital processing in FPGAs. Existing proposals in the field vary in the way they distribute the task to the available hardware (and software) resources.

One way to classify the existing work is to look at the point when the digitisation takes place. One could digitise the input samples and do all subsequent processing in digital, or one could do some initial processing while the signal is still in analogue form and then do the digitisation. The authors of [12] take this idea even further and propose an analog processing circuitry that only digitises the detected pulse peaks for storage. Their analogue part is based on the dual slow/fast filter concept we discussed in the previous section, but uses semi-Gaussian shaping. For the digital part, they use a 10MHz, 12-bit ADC and an FPGA to control the process, store results in on-chip memory, and transfer histograms to a host computer for visualisation.

Other researchers propose purely digital fast data acquisition systems, coupled with off-line processing blocks [20]. The speed of the data acquisition ensures good resolution, while off-line processing relaxes the need for an equally fast processing block, at the cost of a very large (tens of megabytes) sample memory for intermediate storage. A similar approach is proposed in [10], but this time a single FPGA solution with an embedded on-chip processor does the off-line processing in software to increase the flexibility of the device. Moreover, in [15], a very fast 200MHz, 14-bit ADC is used for data acquisition, but the large memory requirement is removed by using an FPGA to compress and store the results for later software processing. In the latter case, there is no embedded processor, and the software is running on a host PC.

Yet another group of researchers has been investigating purely hardware solutions using FPGAs for on-line pulse processing ([21], [18], [14], and [8]). One thing all these proposals have in common is that they rely on external chips (DSPs and/or microcontrollers) to assist the FPGA in the calculations and system control. This releases valuable resources on the FPGA, but increases the size of the resulting PCB.

We feel that our work shares the most with that in [11]. Its authors propose a hardware solution with an FPGA and an external DSP. However, all calculations are performed on-line within the FPGA, and the DSP is only used for storage and transmission of results via a serial port. The source signal is sampled at 60MHz with an 8-bit ADC.

To the best of our knowledge, our idea is unique in that it proposes a single-chip solution, using an FPGA and no other supporting chips. We take advantage of our managed SoC approach to embed the complementary processor and interconnection bus within the FPGA, resulting in a small and lightweight implementation, capable of doing on-line pulse detection at the speed of the ADC (100MHz in our experimental setup). We suggest a modular IP core with an AMBA bus interface that can be easily connected to many of the available on-chip processors, both synthesisable and hard-coded.

This concludes our discussion on the subject of background knowledge and related work. In the next chapter we look into the design of the pulse detector and the implementation of the ideas presented in this chapter.

System Architecture 3

This chapter presents a detailed description of the system architecture used for the digital pulse detector IP core. Since our modular approach allows us to connect our pulse detector to a variety of Systems-on-Chip, we describe the IP core in isolation from the rest of the system, and leave the discussion of the complete system for the next chapter.

We begin in Section 3.1 by presenting an overview of the proposed architecture. In Section 3.2 we examine the filter datapath block, and the way we implemented digital pulse shaping and discrimination. We then move to the AMBA control block (Section 3.3), and look closer at the configuration process of the filters and the transmission of processed data. The chapter concludes in Section 3.4 with a discussion on keeping the two blocks under separate clock domains and the implications of this choice.

3.1 Architectural Overview

The pulse detector IP core we propose consists of two main blocks: the AMBA control block and the filter datapath. The goal of splitting the design into two major blocks is to decouple computation from communication. By keeping these two blocks separated, we can easily modify the AMBA block to match another protocol if needed, without affecting the way computation is done. Furthermore, we would like to run the external ADC and the filters at a clock speed that might not match the one used over the communication bus. Our approach allows to easily define separate clock domains per block.

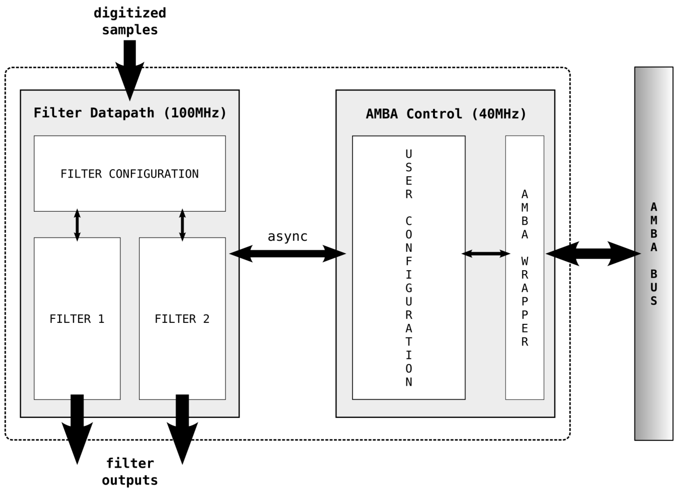


Figure 3.1: IP core functional diagram

Figure 3.1 presents the functional diagram of the IP core. The AMBA block controls the communication between the filters and the AMBA bus. The device appears as a memory-mapped set of registers, accessible over the system bus. Thus, we can alter the filter parameters and query the status of the device by using embedded software that accesses those registers.

Once the desired configuration is in place, the control block transfers the new parameters for both filters to the filter datapath over an asynchronous link, and processing begins. The detected pulse heights are transmitted back to the AMBA block for temporary storage, until they are retrieved by the embedded software. The output of both filters is connected to the pins of the FPGA and can be driven into a digital-to-analog converter for reconstruction and inspection.

3.2 Filter Datapath

The filter datapath is in charge of storing the incoming samples from the external ADC, performing trapezoidal filtering calculation to shape the input, and extracting the pulse heights with a peak detection algorithm, while rejecting pile-ups.

We first elaborate on the theory behind trapezoidal filtering, in order to arrive at a hardware-implementable form. Recall from Section 2.1 on page 4 that the output $O[n]$ of a digital trapezoidal filter at time unit n is given by:

$$O[n] = \frac{1}{w} \times \left[\left(\sum_{k=n-w}^{n} I[k] \right) - \left(\sum_{k=n-2w-g}^{n-w-g} I[k] \right) \right],$$

where w is the window width and g is the gap. It follows that in the next time unit, the output $O[n+1]$ is given by:

$$O[n+1] = \frac{1}{w} \times \left[\left(\sum_{k=n+1-w}^{n+1} I[k] \right) - \left(\sum_{k=n+1-2w-g}^{n+1-w-g} I[k] \right) \right]$$

It is not efficient to average all samples within both windows every clock cycle. Such an approach would require a great amount of resources and would provide limited flexibility, since it would be hard to adapt the calculation to different window and gap sizes.

The solution is to serialise the calculation, by relating two consecutive outputs to each other. It is easy to spot the relation between $O[n+1]$ and $O[n]$:

$$O[n+1] = O[n] + \frac{I[n+1] - I[n-w]}{w} - \frac{I[n+1-w-g] - I[n-2w-g]}{w}$$

In other words, each clock cycle the output of the filter is equal to its previous value, with the addition of a new sample and the removal of the oldest one inside each window. The new serial form is much more suitable for hardware implementation. It only involves four of the input samples and the last output in feedback. This way we save on hardware resources and increase throughput, as it is often the case with serial implementations.

In the above equations, we can also safely ignore for hardware implementation purposes the division by w. The window width is a parameter of the filter, and as such, it will be stored in the configuration registers. The division will be later performed by the application controlling the filter, which has access to those registers. With this in mind:

$$O[n+1] = O[n] + (I[n+1] - I[n-w]) - (I[n+1-w-g] - I[n-2w-g])$$

Our choice is justified by the fact that division is the slowest of basic operations, and performing it every clock cycle consumes valuable resources and reduces the performance of our design. Instead, we delay the operation until the detected pulse heights are transferred to the complementary on-chip processor. Given the fact that the on-chip processor is only in charge of accumulating the detected pulses and transferring them via a serial port, we can use the remaining processing power for the division. This way we avoid dividing every intermediate filter output, and only divide once per detected pulse.

Finally, following the notation used in Figure 2.1 on page 5, the previously established relation between two consecutive outputs can be rewritten as:

$$O[n] = O[n-1] + WIN2_{NEW} - WIN2_{OLD} - WIN1_{NEW} + WIN1_{OLD} \tag{3.1}$$

However, a straight-forward implementation of Equation (3.1) using behavioural VHDL leads to slow cycle times. A 16-bit ripple-carry adder takes 16 full-adder (FA) delays to produce a result. Optimised adders take less, but having five operands (four samples plus the previous output) requires at least two levels of addition/subtraction. We could add pipeline registers between these levels, at the cost of increasing input to output latency.

There exists a better solution, one that takes slightly more than one 16-bit full-adder delay, to calculate Equation (3.1), without the need for pipelining. We take advantage of the dedicated, fixed nature of the calculation to propose a tree-like structure of Carry-Save Adders (CSA, more information in [22]). This structure reduces within a few (three) FA delays the five operands of Equation (3.1) into one number in redundant sum and carry form. The final result is obtained by a single optimized 16-bit full adder. We explain our solution in detail in Section 3.2.2. Such an approach maintains a high-throughput while keeping latency to a minimum, resulting in a fast, responsive system, suitable for time-critical applications (such as radiation level monitoring).

The proposed filtering calculation requires that every clock cycle we provide four new operands from the filter's memory. One option is to use an external memory to store and retrieve the samples. This however would add the unnecessary delay of accessing the external chip and extra logic to handle the memory. A better solution would be to use the dedicated memory blocks often to be found inside FPGAs, but we would like to keep the IP core free of FPGA-specific structures, and we would still need the extra memory-controlling logic.

In our design, past samples are stored in a simple FIFO memory structure, made out of flip-flops. Incoming samples from the external ADC are driven directly into the first

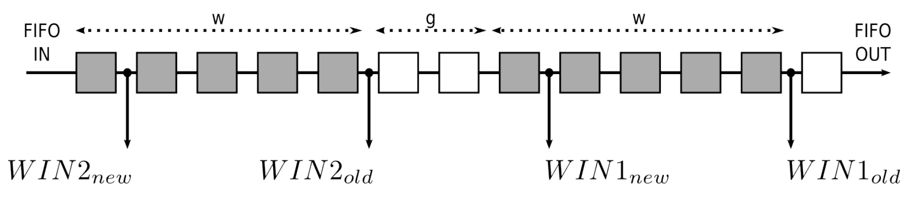


Figure 3.2: Selection of operands from FIFO memory

memory element of the FIFO, at the clock rate of the ADC. Every clock cycle, a new sample from the ADC is inserted and the oldest one is dropped.

During an initial configuration step, the user sets the desired values of w and g. These parameters are then translated by the embedded application into four FIFO "pointers" and stored inside the configuration registers. The pointers are used to select four cells of the FIFO, representing the samples at the windows boundaries $(WIN2_{NEW}, WIN2_{OLD}, WIN1_{NEW}, WIN1_{OLD})$. During runtime, we keep track of the previous result and add/subtract to it the sampled values as they move through these four cells. Figure 3.2 depicts how the values of w and g translate to positions inside the FIFO memory.

As mentioned in Section 2.2 on page 6, we opted to use a dual-filter setup for enhanced pulse detection. Consequently, we must consider two quads of FIFO values, one for the slow and another one for the fast filter. Each of the two filters performs its own calculation using a quad of values. Their results are combined and processed in a serial way by a peak detection algorithm, concurrently with pile-up inspection. The filter results "flow" constantly on the output, and it is in the responsibility of the peak detection algorithm to signal whether a given output is a peak to be captured, a pile-up, or just an intermediate calculation value.

We have already identified the three vital sub-blocks of the filter datapath: the operand selection block provides every cycle four new samples, the filtering calculation block performs trapezoidal shaping of the input, and the peak detection block determines the pulse heights and rejects pile-ups. Each block feeds into the next one, in the given order, much like a traditional processing pipeline. We add registers in between these blocks, to arrive at a three-stage pipeline, with a high throughput and low (three clock cycles) latency.

Our idea is further illustrated in Figure 3.3, with the filter datapath split by intermediate registers into three stages. The result is a clean, shallow-pipelined serial design.

The main advantage of our proposal is that we keep the hardware complexity of the calculation independent of the filter parameters, since the computation always includes five operands (the old output plus four samples). We will discuss further advantages and disadvantages of our approach in the following sections as we look into the design of each pipeline stage in detail.

3.2.1 Operand Selection

The first stage of the pipeline consists of the input FIFO and the mechanism that selects the appropriate operands for both the slow and the fast filter. As it can be seen in

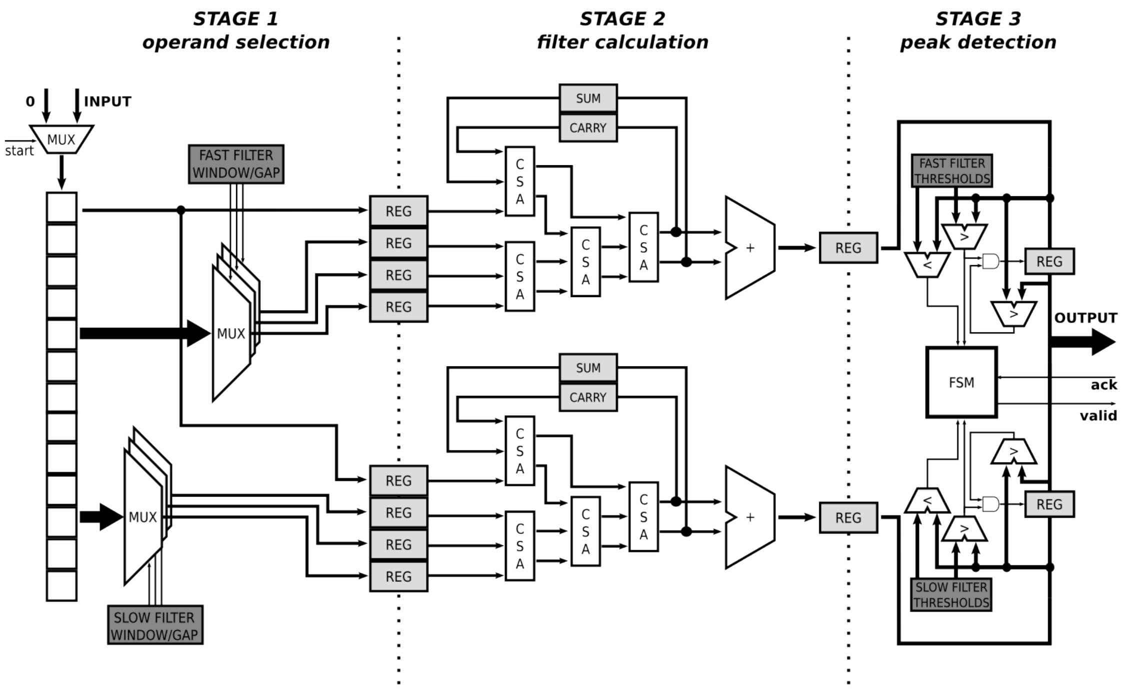


Figure 3.3: Three-stage pipelined design of the filter datapath

Figure 3.4, two configuration registers provide for user control over the operands selection. These registers are 32-bit wide and are controlled by the AMBA block over an asynchronous interface.

The input FIFO is 8-bit wide (to accommodate the 8-bit samples coming from the external ADC) and 256-sample deep. Given a 100 MHz clock, the FIFO can keep up to 2560 ns of past samples in memory. The external ADC encodes the digitised samples using a 128-biased representation (most negative number corresponds to all binary digits being zero). We would like to use a complement representation, because it makes it easier to handle calculations. A straightforward solution is to invert the most significant bit of each sample, leading to two's complement numbers. We do this before injecting the samples in the FIFO, and we keep all further values in two's complements. More on the binary representations of signed numbers can be found in [22].

When a sample reaches the end of the FIFO, it is dropped. In Figure 3.4, this corresponds to the "vertical" flow of samples within the FIFO. At the same time, all FIFO elements are multiplexed for the operand selection (horizontal flow). As mentioned before, we need four operands for each filter, leading to eight 256-to-1 multiplexers, combined in two groups of four. These multiplexers are formed by two levels of smaller 16-to-1 multiplexers.

We can reduce the number of multiplexers down to six, by noting that one of the operands ($WIN2_{NEW}$) is always the most recent sample, which corresponds to the topmost FIFO element for both filters. Thus, we can remove the two multiplexers and directly forward the first FIFO element to the next stage.

Generally speaking, even a single 256-to-1 multiplexer for 8-bit values is a very wide

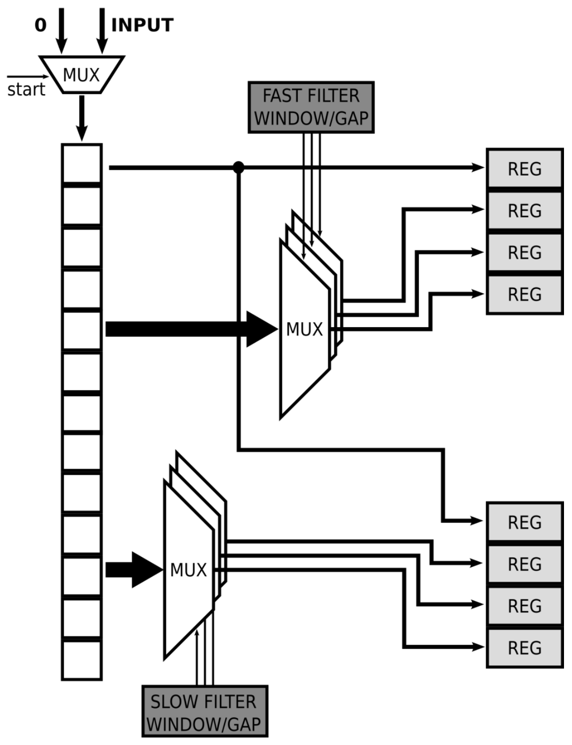


Figure 3.4: First pipeline stage: operand selection

and slow structure that should almost always be avoided. In our application however, the multiplexers do not have such an impact on performance, since the values in the configuration registers controlling the multiplexers are only set once before a measurement, during a configuration step, and remain stable while the device is producing results. Therefore, the path is static and the multiplexers do not introduce any delay in the operational cycle.

Regarding the large area consumed by the multiplexers, our choice is justified by the fact that the design should be as generic and FPGA-agnostic as possible. This requirement prohibits the use of modern runtime (dynamic) reconfiguration techniques, a solution that would have removed the multiplexers altogether, at the cost of tying the design to a specific FPGA manufacturer.

3.2.2 Filtering Calculation

Two pairs of four operands enter the second pipeline stage as 8-bit two's complement numbers. Since we are not averaging the outputs by dividing the result over the window width, we expect the output value to grow over the maximum representable number within 8 bits. To accommodate for this, we increase all operands to 16 bits by sign-extending them and do all subsequent calculations in 16 bits. In the worst case scenario, the input is a step function that immediately rises to the maximum representable value

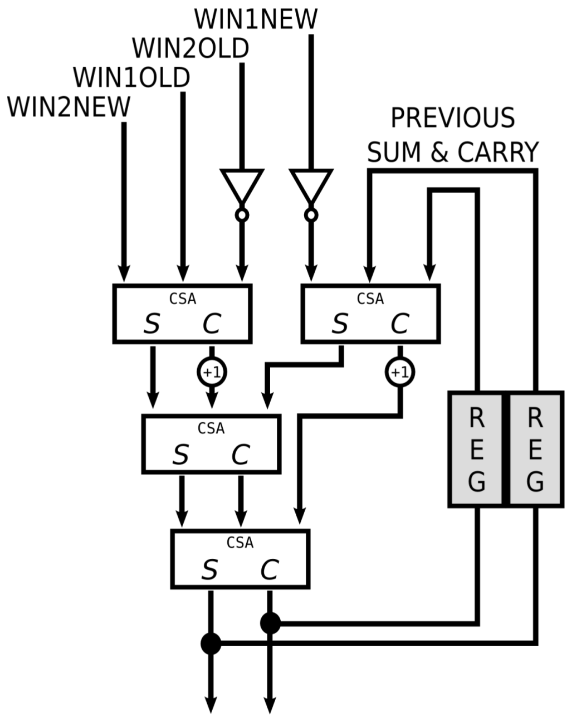


Figure 3.5: Six-to-two CSA reduction tree with two inverted operands

within 8 bits ($2^8 = 256$), a zero gap, and a 128-sample window width[1]. The sum when all 128 (2^7) samples are equal to 256 is representable within 15 bits ($2^8 \times 2^7 = 2^{15}$). The extra (16th) bit allows us to double the FIFO memory (if required) without the need to redesign the calculation logic.

For the calculation itself, we make use of the fact that we have serialised the computations and propose a reduction tree of Carry-Save Adders (CSAs). Every cycle, we get the previous output in sum/carry redundant form and add it to the four new operands, producing a new pair of sum and carry. This leads to a 6-to-2 reduction tree, depicted in Figure 3.5. The final result is obtained by adding the two outputs of the tree. The final addition is described in behavioural VHDL, in order to let the synthesis tool optimise it in any way possible. Using this scheme, each CSA takes one FA delay, leading to three FA delays for the complete reduction tree, plus the delay of a single slow addition.

In order to perform the subtractions present in Equation (3.1), we simply invert the corresponding operands ($WIN2_{OLD}$ and $WIN1_{NEW}$). In two's complement form, inverting a number produces its negative, minus one[2]. Thus, apart from inverting, we also need to add one carry in per inverted operand. Since we are using CSAs, the carry outputs must be shifted left by one before they enter the next stage of the reduction tree, leaving an "empty" space at the least significant bit position. We can use these empty spaces to inject "+1" and complete the negation of the two operands ($WIN2_{OLD}$ and

[1]We need symmetrical window widths, therefore we can only partition our 256-samples memory into two windows of 128 samples each.

[2]Example: inverting number $9 = 01001_b$ leads to $10110_b = -10$ instead of -9.

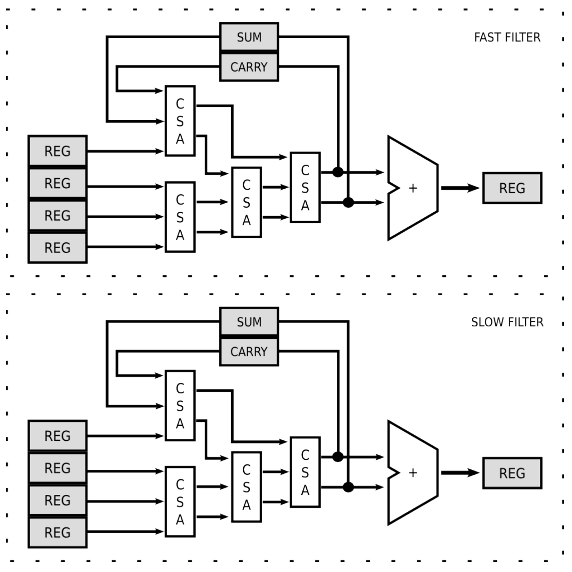


Figure 3.6: Second pipeline stage: filtering calculation

$WIN1_{NEW}$) without resorting to slow ripple-carry adders.

The final division is performed later in software by reading the contents of the configuration registers and extracting the window width. As a beneficial side-effect, we provide in this way increased 16-bit output resolution for the detected peak levels.

The complete pipeline stage that is responsible for the trapezoidal calculation is sketched in Figure 3.6. Both filters use identical structures, consisting of four registers with the operands from the previous pipeline stage, the CSA reduction tree, the final adder, and two registers to hold the previous output in redundant form.

3.2.3 Peak Detection

The final stage of the pipeline combines the outputs of both the fast and the slow filter to detect input pulses. The filter outputs carry energy information. Upon detecting an event (a pulse peak), the peak detection algorithm, implemented in hardware as a Finite State Machine (FSM), stores the energy information at that moment. Figure 3.7 depicts the third pipeline stage: it is composed of the FSM, input pipeline registers, threshold registers, plus three comparators, one two-input AND gate and one register per filter.

At configuration time, the user sets two thresholds for each filter. An upper threshold defines the value that the output must reach to consider an event, while a lower threshold comes into effect after the upper one has been crossed, to mark the end of the event. By keeping separate upper/lower thresholds, we reduce noise effects: even small fluctuations of the output immediately after crossing the upper threshold may drop the output below the threshold and cause the device to erroneously end the detected event before reaching its true maximum. We avoid this problem by setting a second lower threshold.

Each filter uses a 32-bit configuration register to hold the two 16-bit thresholds. It should be noted that since we do not perform the required division in hardware, the user

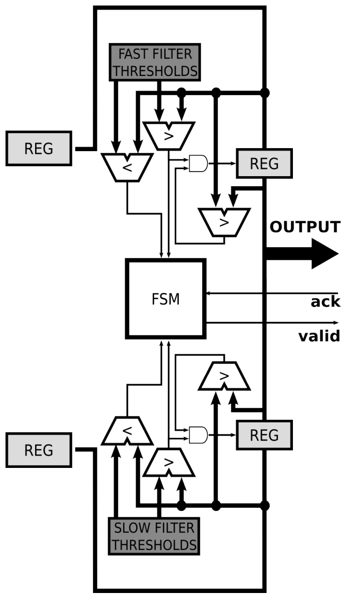


Figure 3.7: Third pipeline stage: peak detection

should provide all threshold values multiplied by the window width.

During runtime, the outputs of the filters are constantly compared against their two thresholds. One comparator is used to check whether an output is greater than the upper threshold, while another comparator is used to check whether the output is smaller than the lower threshold. The outputs of both comparators feed into the peak detection FSM.

The concept of using a dual-filter setup was introduced in Section 2.2. Using the combined four comparator outputs of both filters, we build an FSM that is able to detect clean events and discard pile-ups. Our FSM design is presented in Figure 3.8. When idle, the FSM rests at state 0. In case of an event, the first threshold to be crossed is the upper threshold of the fast filter ($FF \uparrow$). The output of the corresponding comparator will be set to one, triggering the transition to state 1. Depending on the filters setup, the next event will be either from the slow filter crossing the upper threshold ($SF \uparrow$) as well, or from $FF \downarrow$, leading to states 3 and 2, respectively. Whichever event happens first ($SF \uparrow$ or $FF \downarrow$), the other one should be the one to follow, in which case the FSM makes a transition to state 4. To make a complete event (E), the slow filter must return too ($SF \downarrow$). Any other series of events results in a pile-up (P) and are discarded. In any case, the FSM always returns to state 0.

To actually detect a peak within an event, we make use of the third comparator in Figure 3.7. A register is used to store the last maximum output value. Every cycle, we compare the new filter output to the maximum value. If we detect a new maximum *and* we are over the upper threshold, we store and output the new value. This way, after crossing the upper threshold, the final register keeps track of the maximum value, and will keep it until a new event has started. When the FSM detects a new event, it signals to the AMBA block that the output of the filter datapath is valid. The final output of the datapath is a 32-bit value combining the 16-bit peaks of both filters.

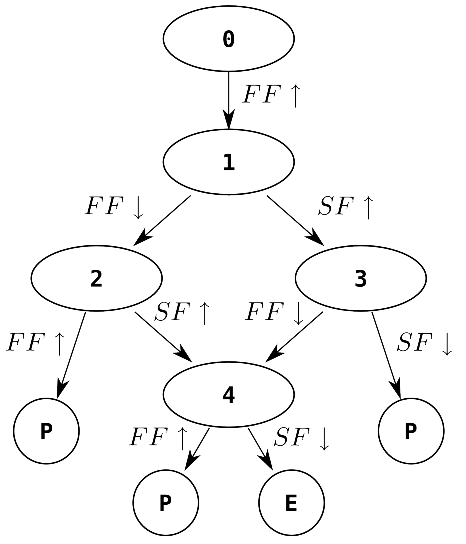


Figure 3.8: Peak detection finite state machine

3.3 AMBA Control Block

The second half of the pulse detector is the AMBA [9] control block. This block forms the communication link between the filter datapath and the controlling application that is running on the embedded on-chip processor.

Our pulse detector is designed to communicate over the AMBA Advanced Peripheral Bus (APB). AMBA is a popular interconnection protocol, especially in SoC implementations. APB is a slow single-master non-pipelined bus that is usually attached to a faster AHB bus. The AHB/APB bridge acts like the master and all APB peripherals are slaves.

In our implementation we can afford a slow bus because we perform real-time calculations and reduce the amount of output data; for every pulse event, only a single 32-bit value representing the pulse height is produced. Using APB has several advantages over faster interconnections like AHB. The simplicity of the protocol reduces the complexity of the interface, resulting in a more compact design with less wires and logic. Furthermore, APB is optimised for low power consumption, a much appreciated attribute when designing for space applications.

The diagram of the AMBA block can be seen in Figure 3.9. A set of 32-bit configuration/status registers appear as a 32-bit aligned memory-mapped region, accessible over the APB bus. Communication with the device consists of read and write operations inside this I/O region. Major components inside the block, apart from the registers, include the AMBA signal decoding logic and two FSMs; one is programmed to start, stop, and reconfigure the filter datapath, while the other one is in charge of retrieving the pulse heights.

The registers are used to store the configuration and to control the filter. Configurable parameters include the window width and gap of the trapezoidal filters, and the peak detection thresholds. Another register is used to hold the last detected pulse height. For

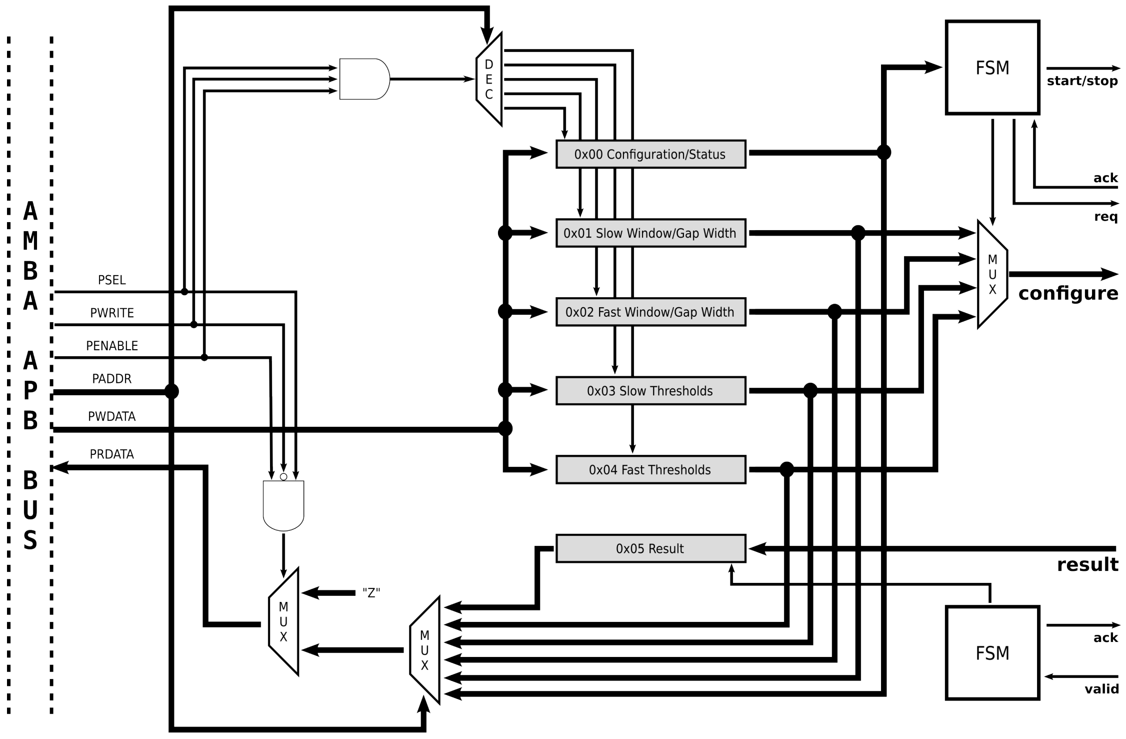
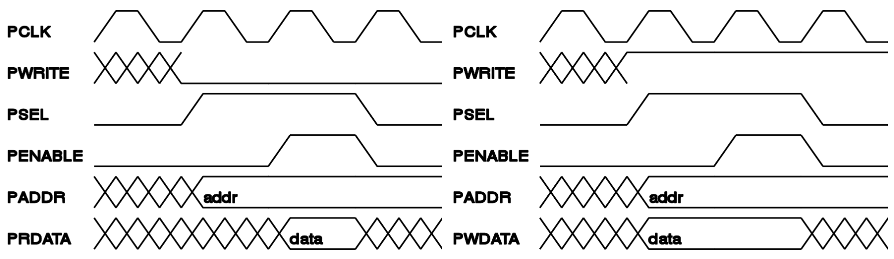


Figure 3.9: Block diagram of the AMBA interface



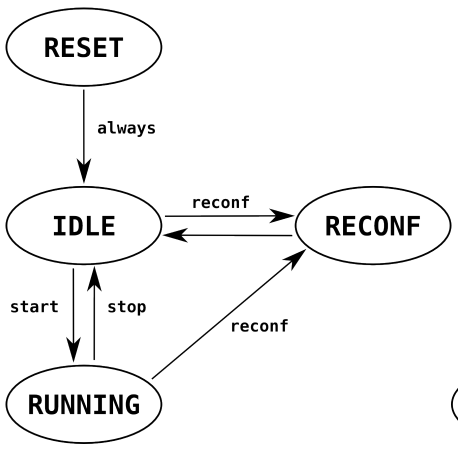
(a) Read (b) Write

Figure 3.10: AMBA APB read and write bus cycles

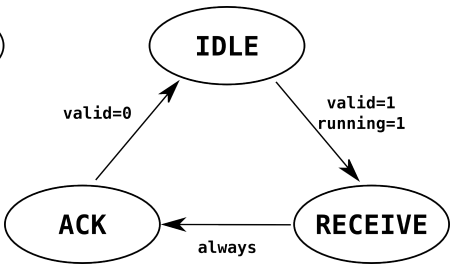
a complete list of the available registers and their purpose, refer to Appendix A.

The AMBA APB bus uses a typical set of control and data signals. Figure 3.10 presents the timing diagrams of the read and write bus cycles. Based on this information, we have built a simple decoding mechanism that controls the loading signal to all registers and returns the correct register contents.

Using the AMBA APB bus, the application accesses the configuration registers to set the desired window and gap widths, as well as the thresholds for both filters. After choosing and setting up the desired values, the application then sends a command to reconfigure the filters, which triggers the asynchronous transition of the configuration values to the filter datapath. Following that, a command to start the computation



(a) Filter control FSM



(b) Pulse retrieval FSM

Figure 3.11: States and transitions of the AMBA control block

enables the processing of incoming samples, while a command to stop terminates the process.

Figure 3.11 presents the available states and transitions of both FSMs. Upon reset, the system enters the “reset” state for a short period of time, before moving to the “idle” state. From there, a command from the user to start processing will move the system to the “running” state, while a command to reconfigure will move the system to the “reconfiguration” state (until the new setup is loaded and the system returns to idle state). While running, a user command to stop will return the system to the idle state, while a reconfiguration command will move again the system to the “reconfiguration” state. If upon completion of the reconfiguration step the system is still in operational mode (in other words, the START bit of the configuration/status register is still ‘1’), then the filters will resume operation (after a quick transition through the “idle” state), otherwise the system will rest in the “idle” state. This corresponds to the filter control FSM illustrated in Figure 3.11a.

The second FSM (Figure 3.11b) is programmed to communicate with the filter datapath block and retrieve the pulse heights. If a “valid” signal is received from the filter datapath while the device is running, the second FSM moves to the next state and loads the new value in the result register. It then issues a signal to acknowledge the reception and waits for the valid signal to be deasserted before returning to the idle state.

3.4 Multiple Clock Domains

Very often in the design of embedded systems that interact with the environment, the circumstances dictate the use of multiple clocks. Data acquisition blocks may need to sample a natural event at a high rate to ensure the required resolution. This rate is often much higher than the one we can achieve and/or want for the rest of our design.

In our case, another important argument in favour of using multiple clocks is that we design a modular IP core. Our module can be interfaced to a number of SoC solu-

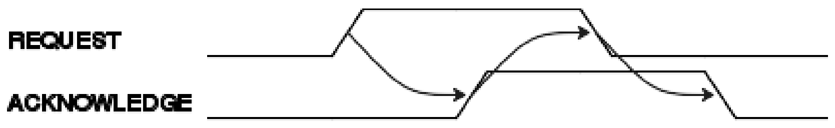


Figure 3.12: Four-phase handshaking protocol

tions, and the complete design can be programmed in a variety of silicon chips. These facts introduce uncertainty concerning the clock rate of the complete system. However, we would like our pulse detector to function with a predefined speed and resolution, decoupled from the performance of the supporting system.

Multiple clock domains is the solution to the above-stated problems, but not without pitfalls. When designing using multiple clocks with different frequencies and/or phases, one must pay special attention to minimise the number of signals which cross clock domains, and to properly synchronise those that do so.

Synchronisation is needed to avoid metastability issues [26]. If not properly synchronised, a signal has the chance to violate setup and hold times on the receiving memory element, and arrive at exactly the wrong moment to trigger a metastable state for an unknown length of time.

Several solutions are available for reducing the risk of metastability, with varying degrees of protection. For the first version of our design we chose to implement the simplest of those: we place a pair of flip-flops in series on the receiving side of every signal that crosses clock domains. By doing so, we greatly reduce the chance of a metastable signal reaching the receiving logic.

However, duplicating flip-flops increases the area footprint and introduces latency. Therefore, we must keep the number of signals which cross clock domains to a bare minimum. Furthermore, for multi-bit signals, it is not efficient to duplicate flip-flops for every single bit. The solution is to add two handshaking lines to properly synchronise the delivery of the multi-bit value. Of course, the signals of the handshaking lines themselves should be protected from metastability, but this introduces only two more flip-flops. For our handshake mechanism we use the four-phase protocol depicted in Figure 3.12.

The complete set of cross-clock domain signals is illustrated in Figure 3.13. Gray boxes represent metastability protection flip-flops. A reset and a start signal cross the clock domains to allow us to control the filter datapath from the control block side. Both of them correspond to bits in the configuration/status register (refer to Appendix A.1). Two sets of four-phase handshaking lines protect the configuration and result buses, both 32-bits wide. The configuration bus is used to transfer the configuration values to the filter datapath, while the result bus transfers the detected peaks back to the control block.

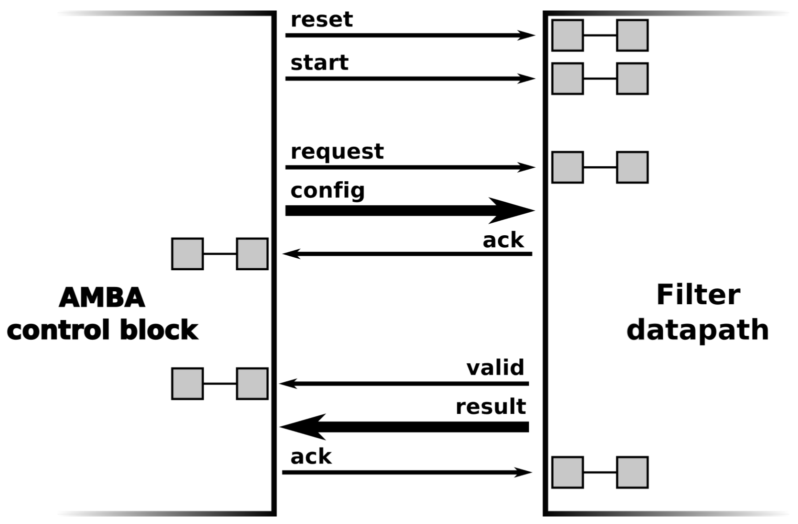


Figure 3.13: Set of cross-clock domain signals

Experimental Setup & Results 4

In the previous chapters we argued in favour of a System-On-Chip approach to digital pulse detection and discussed the system architecture of the proposed IP core. In this chapter we present the experimental setup that was used to develop and test the functionality of our module within a complete, self-sustained single-chip system. We also presented the preliminary results that we obtained using this setup.

In the following sections, we examine the library of IP cores that was used to assemble our SoC, including the on-chip processor, interconnection bus, and interface to host PC (Section 4.1). We also look into the embedded C code that we wrote to control the entire process. In Section 4.2 we present the development board and FPGA used to program our design, together with a custom mezzanine board that hosts the analog readout electronics, up to and including the ADCs. The mezzanine board is used to digitise the input signal, and in Section 4.3 we briefly discuss our setup for generating and capturing radiation-induced pulses. Section 4.4 contains a review of the tools that were used to develop, simulate, synthesise, debug, and process the results. The obtained results are presented in the last part of this chapter, in Section 4.5.

4.1 GRLIB IP Core Library

The GRLIB IP library of reusable cores is centered around the AMBA on-chip interconnect bus. The library is developed and maintained by Gaisler Research [1]. It is provided under the GNU GPL open-source license and it is largely vendor independent. Our FPGA-agnostic IP core, combined with a selection of IP cores from GRLIB, can be programmed in almost any FPGA available on the market, provided that it fits inside the target chip.

GRLIB uses the AMBA AHB as its main interconnect, with an optional AHB/APB bridge to connect a slave APB bus. The library includes cores for the LEON3 general purpose processor, 32-bit PC133 SDRAM controller, 32-bit PCI bridge with DMA, 10/100 Mbit ethernet MAC, 8/16/32-bit PROM and SRAM controller, CAN controller, TAP controller, UART with FIFO, modular timer unit, interrupt controller, and a 32-bit GPIO port. Memory and pad generators are available for Virage, Xilinx, UMC, Atmel, Altera, Actel, and Lattice.

In our SoC implementation, apart from our own pulse detector core, we also make use of the LEON3 processor, AHB bus controller, memory controller, a debug support unit for the LEON3 over a serial port, and the AHB/APB bridge. On the APB bus, we connect another UART port, the interrupt controller, and our core. The block diagram of the complete SoC can be seen in Figure 4.1.

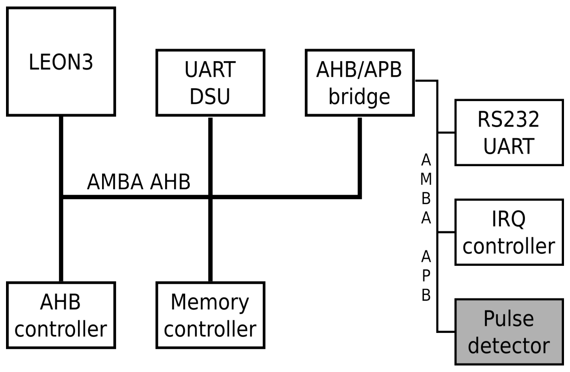


Figure 4.1: Complete System-On-Chip block diagram

4.1.1 The LEON3 Processor

The LEON3 is a synthesisable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture. It is a Symmetric Multi-Processing (SMP) capable CPU, with a 7-stage pipeline and hardware MUL, MAC, and DIV units. It also supports an additional IEEE-754 FPU. According to the manufacturer, the processor can reach up to 125 MHz on an FPGA, or 400 MHz on a 0.13 um ASIC technology.

The LEON3 follows the "Harvard architecture", using a single main memory but separate instruction and data caches. Caches are configurable as either direct-mapped or 2- to 4-way set associative, with 256 KB per set. Supported replacement strategies include random, least-recently used (LRU), and least-recently replaced (LRR, also known as FIFO).

In our system, the processor has a complementary role, mostly control of the entire pulse detection process, since the majority of the calculations are performed by our dedicated hardware. Hence we would like to use a minimal instance of the processor. To this end, we configured the LEON3 with a single CPU, direct-mapped (1-way set associative) instruction and data caches, and no hardware floating-point support. We do however keep the DIV unit, in order to quickly perform the final division of detected pulse heights (see also Section 3.2 on page 10). To facilitate debugging, we've also included the non-intrusive Debug Support Unit (DSU), that offers access via the serial port to all on-chip registers and memory, together with trace buffers of both executed instructions and AMBA bus traffic.

LEON3 is also available in a non-free fault-tolerant (FT) version that offers Single Event Upset (SEU) immunity with no timing penalty compared to the non-FT version. This fact makes the LEON3 a very attractive solution for space-oriented SoC designs.

4.1.2 Plug & Play Capability

All GRLIB cores use the same data structures to declare the AMBA interfaces, and can easily be connected together. Figure 4.1.2 lists the available APB input/output sets of signals, while Figure 4.1.2 presents an example of a typical declaration of an APB slave device using these records.

```
-- APB slave inputs
  type apb_slv_in_type is record
    psel    : std_logic_vector(0 to NAPBSLV-1);          --  slave select
    penable : std_ulogic;                                --  strobe
    paddr   : std_logic_vector(31 downto 0);             --  address bus (byte)
    pwrite  : std_ulogic;                                --  write
    pwdata  : std_logic_vector(31 downto 0);             --  write data bus
    pirq    : std_logic_vector(NAHBIRQ-1 downto 0);      --  interrupt result bus
  end record;

-- APB slave outputs
  type apb_slv_out_type is record
    prdata  : std_logic_vector(31 downto 0);             -- read data bus
    pirq    : std_logic_vector(NAHBIRQ-1 downto 0);      -- interrupt bus
    pconfig : apb_config_type;                           -- memory access reg.
    pindex  : integer range 0 to NAPBSLV -1;             -- diag use only
  end record;
```

Figure 4.2: GRLIB APB slave input/output records

```
library grlib;
use grlib.amba.all;

library ieee;
use ieee.std_logic.all;

entity apbslave is
  generic (
     pindex : integer := 0;
     paddr  : integer := 0;
     pmask  : integer := 0;
     pirq   : integer := 0;
     imask  : integer := 0);
  port (
     rst    : in  std_ulogic;
     clk    : in  std_ulogic;
     apbi   : in  apb_slv_in_type;  -- APB slave inputs
     apbo   : out apb_slv_out_type  -- APB slave outputs
  );
end entity;
```

Figure 4.3: Typical GRLIB APB slave entity definition

The *pconfig* field in the output record of Figure 4.1.2 adds “plug & play” capability to the AMBA APB bus. It includes information like vendor and device ID, address mapping information and assigned irq line. This information is forwarded to the APB bus master (the AHB/APB bridge), so that it can be later retrieved by the controlling application (or operating system) to automatically configure itself to “talk” to the hardware.

4.1.3 PC Interface

The complete SoC interfaces via the serial port to a host PC. During normal operation, the pulse detector captures the heights of the detected pulse events and transfers them to the LEON3 processor. Software running on processor assembles a histogram (number of detected events per pulse height unit) out of the received values and transmits it to the host PC for further processing and visualisation.

Communicating over the serial port is a simple task: the system is configured to redirect all output to the serial port, so transmission is as simple as printing values on "screen". On the host PC side, a terminal application with logging capabilities is all that is needed to receive the transmitted values. Appendix B contains example embedded C code, showing how to transmit results to the host PC.

Future versions of our system will drop the serial interface in favour of SpaceWire (SpW) [16]. The European Space Agency offers ready to use SpaceWire cores, and there is even an SpW core with an AMBA interface. Thanks to the SoC approach, adding a new core that already supports our interconnect should be a trivial job.

4.2 Development Board

The complete design (LEON3, AMBA bus, peripherals, and digital pulse detector) was programmed on a Xilinx FPGA, which was hosted by the GR-XC3S development board by Pender Electronics GmbH [5]. The GR-XC3S is a 15x9.5cm low-cost development board that was designed with the LEON3/GRLIB system in mind.

Figure 4.4 presents a top view of the GR-XC3S board. The FPGA is a 1.5 million gate XC3S1500 Spartan3 from Xilinx [27]. Just below the FPGA chip, one can find the on-board 64MB SDRAM memory, as well as the 8MB flash memory. On the top left, we can also see the two serial ports, one for the Debug Support Unit, and another one for interfacing to the host PC. The existing 20-pin headers were used to interface to the analogue electronics mezzanine board (see below, Section 4.2.1). The FPGA is clocked by a 50MHz crystal that is part of the development board.

Aside from the parts already mentioned, the board also contains an Ethernet MAC and PHY, 24-bit video DAC, USB PHY controller, and PS2 mouse and keyboard interfaces. These components were not used in our system. By removing them (and their respective connectors) we can arrive to a very small and effective solution.

4.2.1 The Mezzanine Board

Although not part of this research, the analogue electronics that deliver data to our digital pulse detector are very important to the quality of the result. Cosine Research BV kindly offered a custom-built mezzanine board which we can easily plug into our development board. Figure 4.5 presents the development board with the mezzanine plugged in.

The mezzanine hosts analogue signal conditioning circuits and the Analog-to-Digital converter. The ADC is 8-bit wide, and can be operated at 100MHz maximum. The board also includes two DACs, also 8-bit, 100MHz. The DACs are used to reconstruct the input and outputs of the system, for debugging purposes. The mezzanine expects to receive its clock from the development board. Distributing the clock to the system is our next topic.

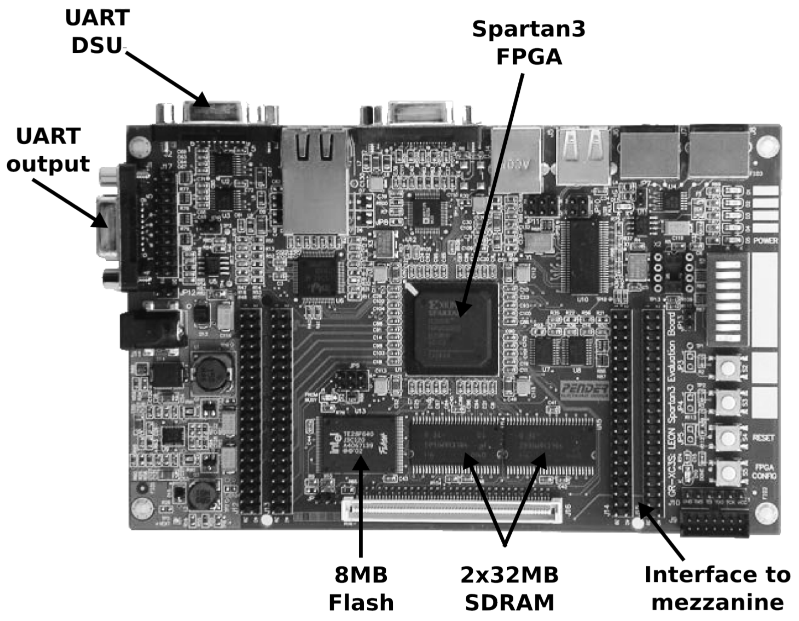


Figure 4.4: Top view of the development board

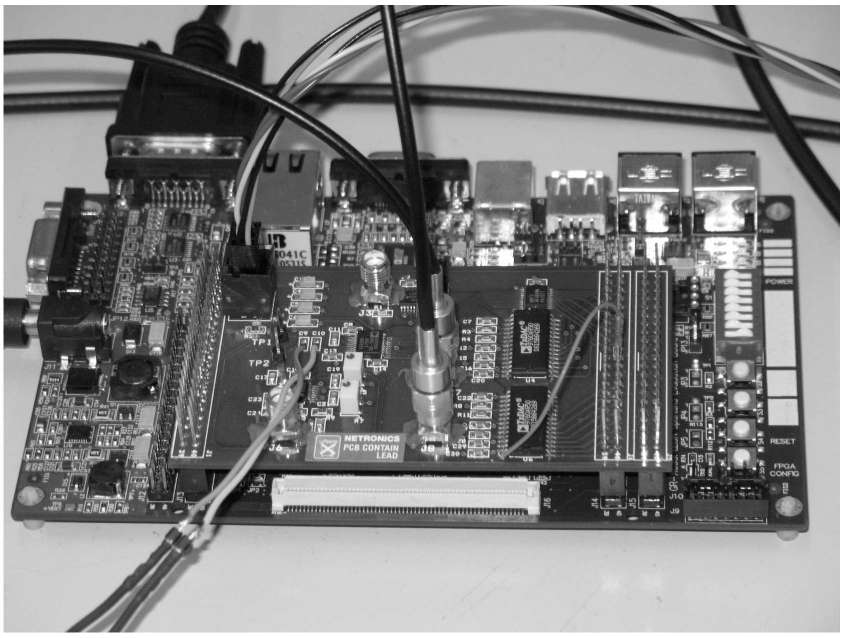


Figure 4.5: Development board with fitted mezzanine

4.2.2 Clock Distribution

The complete system needs a diverse set of clock frequencies. The LEON3 processor and AMBA bus can be safely clocked at 40MHz using the XC3S1500 Spartan3 FPGA. The external SDRAM memory also needs the same 40MHz clock. The mezzanine board

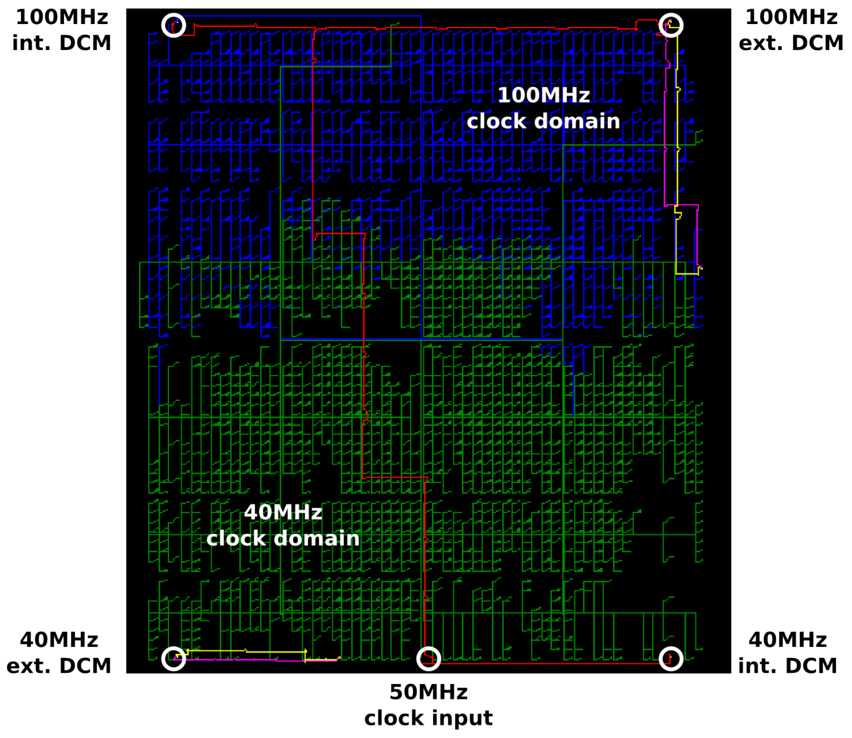


Figure 4.6: Clock distribution network

requires a 100MHz clock, and our pulse detector requires both the 40MHz clock for the AMBA block and the 100MHz for the filter datapath.

All required clocks are generated inside the FPGA from the 50MHz input clock signal, using the embedded Digital Clock Managers (DCM) of the chip [28]. This particular FPGA has four DCMs, placed on the four corners of the chip. Figure 4.6 illustrates how we make use of these four DCMs to synthesise all the required clock signals from the main 50MHz board clock (represented by the red line). The two bottom DCMs are instantiated by GRLIB to generate the 40MHz internal system clock (green lines) and the external clock to the SDRAM memory. The two DCMs on the top provide the 100MHz clock to the filter datapath block of our design (blue lines) and the external clock to the mezzanine board. Both external clocks make use of the Delay-Locked-Loop (DLL) mechanism of the DCM which automatically adjusts the phase of the generated clock to take into account for clock skew.

4.3 Pulse Generation

Our experimental setup would not be complete without a source of pulses. This source was again provided by cosine Research BV. The pulse events were generated by a Cobalt-60 source and captured by a scintillator $LaBr_3$ cylinder, coupled to a Hamamatsu R6231 photomultiplier tube with a pre-amplifier. This kind of source emits approximately 6000 γ-rays per second, one third of which is captured by the scintillator and converted to light. The emitted light is then converted to electrical current and is amplified before entering the digitisation circuit on the mezzanine board. Such a pulse event has a rise time of 500ns and fall time in the order of 100us.

4.4 Development Tools

Another important aspect of this project is the set of tools that we used for development and testing. We decided that all selected software tools should be available for free and if possible open-source. As a starting point, a 32-bit x86 GNU/Linux laptop was used as the main development platform, and GNU/Emacs was the default VHDL and C editor throughout the project.

For the needs of simulating the VHDL code, we selected GHDL [2]. GHDL uses GCC and conforms fully with the IEEE VHDL standards. GHDL creates testbench executable files and can output to Value Change Dump (VCD) ascii file format, as well as its own GHW compressed ascii file format. The GHW format is preferable for long-length simulations because it greatly reduces the output file size. The resulting file was visualized using GTKWave [4], a free wave viewer with the ability to read GHW and VCD files (among many others). Both GHDL and GTKWave are open-source software, released under the GPL.

In order to synthesize the complete SoC and transfer the bitstream to the FPGA, we used the scripts provided by GRLIB. These scripts internally use the command-line Xilinx synthesis and place/route tools. For this purpose, we downloaded and used the ISE WebPack from the Xilinx website. The ISE WebPack is free but not open-source. Apart from the standard tools already mentioned, the WebPack also includes a variety of tools that proved to be very useful: the FPGA editor is a tool that visualises the placement and routing of the synthesised system, and helped us debug problems with very long paths. Figures 4.6 and 4.8, on pages 28 and 31 respectively, where created using the FPGA editor. Another tool that is part of the WebPack, the timing analyser, was also very important for debugging, because it reported all paths which failed the timing constraints and was able to highlight them on the FPGA editor tool.

For the embedded C code, we used the BCC LEON3 cross-compiler from Gaisler Research. BCC is a complete cross-compilation toolchain for the LEON3 processor. It is based on GCC3 and includes the Newlib Embedded C library. Gaisler Research also offers a small tool to pack the compiled executable into a PROM file that we can upload to the on-board flash memory.

Uploading the PROM file or directly accessing the memory of the system is accomplished with GRMON, a debug monitor for the LEON3 processor that uses the Digital Support Unit. GRMON allows access to all registers and memory, instruction and AMBA transaction trace buffers, downloading and execution of applications, breakpoint insertion, remote connection to GDB and flash programming. GRMON is not free, but there is an evaluation version. All tools mentioned from Gaisler Research are available at the company's website [1].

On the host PC side, we received the results via the serial port and captured them using a standard open-source terminal application. We stored the data in a text file and visualized it using Gnuplot [3], an established open-source data plotting software.

4.5 Results

In this section we present the results from the experimental setup that we described in the previous sections. These results are preliminary and only serve to verify the correctness of the design. A complete characterisation of the device requires further testing, but falls outside the scope of this master thesis project.

4.5.1 Area

We first look into the hardware resources. We have already mentioned that we used a Xilinx XC3S1500 FPGA. Figure 4.5.1 lists the usage of the FPGA resources, taken from the reports of the Xilinx tools. We can see that we are using 73% of the available slices for our SoC. This report however does not offer any insight on the resource usage of the individual SoC modules.

In order to have a more in-depth view of the allocated resources, we use the FPGA editor tool from Xilinx and colour the nets of various parts of our design. Figure 4.8a presents the basic output of the FPGA editor tool, corresponding to the 73% usage of the chip. In Figure 4.8b we can see how the resources are divided between GRLIB (white) and our IP core (blue for the datapath, red for the amba block). A more detailed view of the GRLIB area is given in Figure 4.8c, where we see that the LEON3 processor takes most of the space, followed by the Debug Support Unit.

Noticing the large amount of resources needed for the filter datapath of our design, we tried to provide an explanation. If we compare Figure 4.8d to 4.8b, we can see that the main problem behind the increased area usage is the existence of the 256:1 multiplexers. This was foreseen and discussed already in Section 3.2.1, where we also argued in favour of our choice, mainly because one of the requirements was to keep the design FPGA-agnostic, a fact that ruled out the use of dynamic run-time reconfiguration. Since our SoC fits the FPGA and the timing constraints, there is no reason to change our design. However, should it be needed, we can always replace the 256:1 multiplexers with 64:1 versions. This would save a lot of resources, but on the downside, it would allow the user to choose only one position in every four samples, thus reducing the resolution of the filter parameters from 10ns steps (with a 100MHz clock) to 40ns steps.

4.5.2 Performance

Regarding the performance of our system, the initial requirements stated that the SoC should work at the same frequency that it worked without our modifications (40MHz), while the filtering and peak detection should ideally match the speed of the ADC (100MHz). Figure 4.5.2 presents the relevant part from the timing report of the Xilinx tools. It is clear that our design has managed to reach the given targets.

However, it is not equally clear whether our design could actually go faster, since the above requirements were given to the tools as timing constraints, and the tools terminate their "exploration" of available solutions as soon as they find something that satisfies the given requirements. We do know that an early version of the filter datapath alone, without GRLIB, the AMBA control block or assignment of signals to input/output pins of the FPGA, could reach speeds above 300MHz, almost touching the limit of the FPGA

```
Logic Utilization:
  Number of Slice Flip Flops:        5,048 out of  26,624   18%
  Number of 4 input LUTs:           15,898 out of  26,624   59%

Logic Distribution:
  Number of occupied Slices:         9,744 out of  13,312   73%

Total Number of 4 input LUTs:       16,104 out of  26,624   60%
  Number used as logic:             15,898
  Number used as a route-thru:         131
  Number used as Shift registers:       43
  Number of bonded IOBs:               125 out of     333   37%
    IOB Flip Flops:                     45
  Number of Block RAMs:                 18 out of      32   56%
  Number of MULT18X18s:                  1 out of      32    3%
  Number of GCLKs:                       3 out of       8   37%
  Number of DCMs:                        4 out of       4  100%

Total equivalent gate count for design:  1,373,808
Additional JTAG gate count for IOBs:         6,000
```

Figure 4.7: Post-Map FPGA resource utilisation

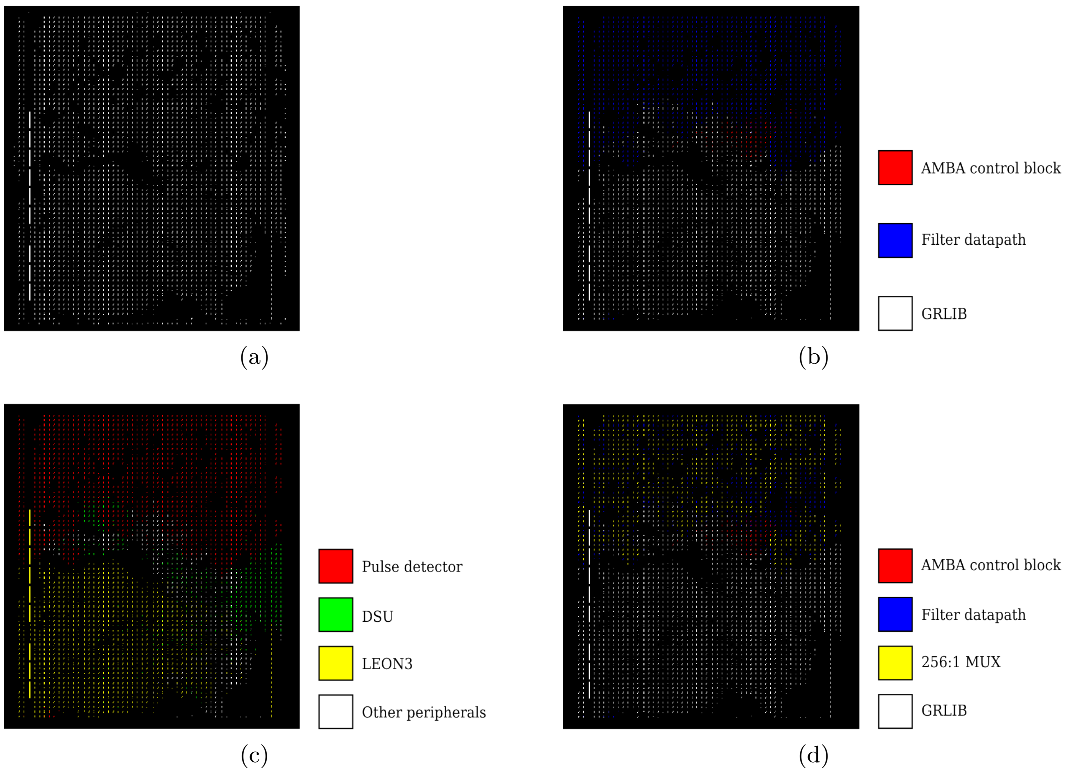


Figure 4.8: XC3S1500 Spartan3 FPGA resource usage

```
---------------------------------------------------------------------------------------
  Constraint                          | Check   | Worst Case |  Best Case  | Timing
                                      |         |   Slack    | Achievable  | Errors
---------------------------------------------------------------------------------------
  Filter datapath and ADC clock       | SETUP   |    0.657ns |    9.343ns  |    0
                                      | HOLD    |    0.907ns |    ~107MHz  |    0
---------------------------------------------------------------------------------------
  AMBA clock                          | SETUP   |    2.378ns |   22.622ns  |    0
                                      | HOLD    |    0.901ns |     ~44MHz  |    0
---------------------------------------------------------------------------------------
```

Figure 4.9: Timing report

itself. But this cannot serve as concrete evidence, since we were using much less resources and thus the tools had an easier task in finding the optimal placement. This effect is verified by the fact that a later attempt to measure the speed of the device using the same design, with the addition of constraints to I/O pins, resulted in a 150MHz maximum clock frequency. We believe the cause was that in the latter case, the logic had to be placed relevant to the I/O pins, some of them positioned on one side of the chip, while others on the opposite side, forcing the tools to create long paths, and reducing the scope of available solutions.

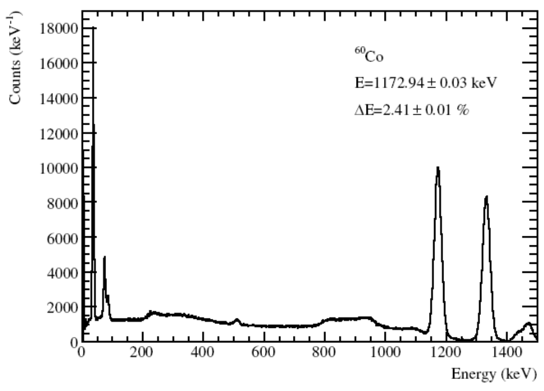
4.5.3 Histograms

A discussion on the obtained results would not be complete without presenting the resulting histograms. In Figure 4.10 we can see five similar histograms. The first one (4.10a) is our reference spectrum, obtained from a Cobalt-60 source using a well-tested detector and processing setup. The rest of the figures (4.10b to 4.10e) are histograms built from data collected using our system and the same Cobalt-60 source as in 4.10a.

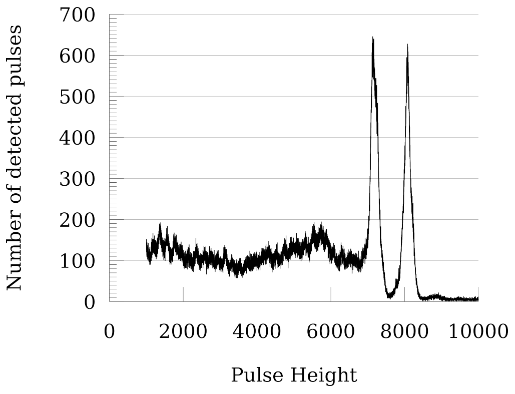
The differences between the histograms in 4.10b to 4.10e are due to different filtering parameters. These parameters are given for reference under each figure, following the format *Window/Gap/Upper Threshold/Lower Threshold* (F stands for Fast filter, S for Slow). These results show how important it is to properly configure the filters before the measurements.

The closest match to 4.10a is that of 4.10c. It should be noted that although the reference figure is a proper spectrum, our results are simple pulse height distribution histograms. In other words, we directly plot the output values, representing the heights of the detected pulses, instead of going through the linear transformation to get the energy carried by the pulses. However, since this transformation is linear, the shape of the reference spectrum and our histograms should be almost identical.

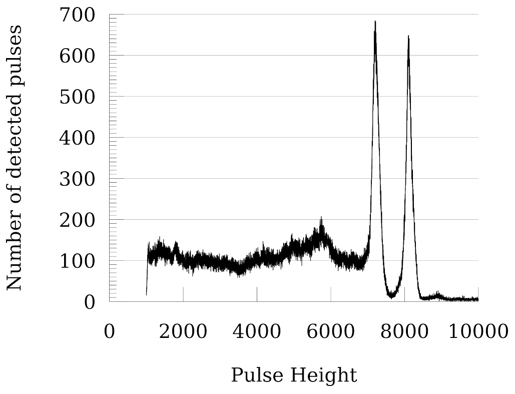
All our presented histograms use the same thresholds for the slow filter (1000/1000). It is due to this fact that all plots start from position 1000 in the horizontal axis.



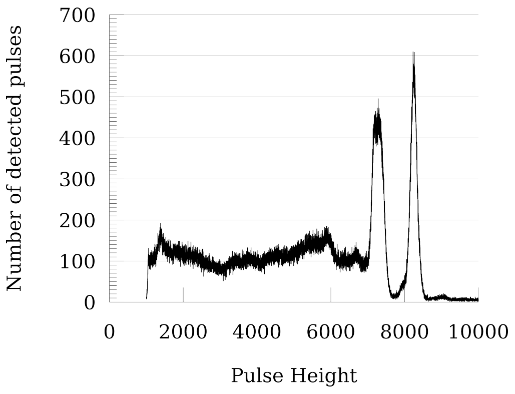
(a) Reference spectrum



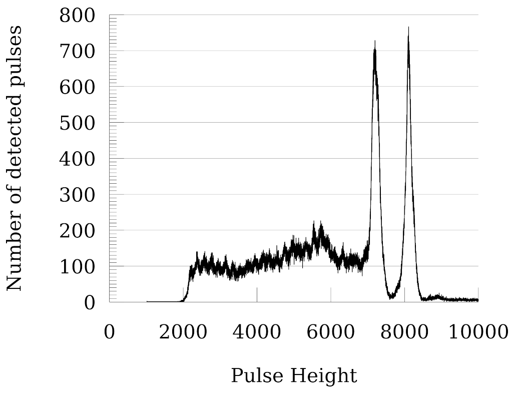
(b) F:50/15/256/128, S:100/15/1000/1000



(c) F:50/15/512/256, S:100/15/1000/1000



(d) F:50/50/512/256, S:100/50/1000/1000



(e) F:25/15/256/128, S:100/15/1000/1000

Figure 4.10: Pulse height distribution histograms for one million events from a Cobalt-60 source

Conclusion 5

In this thesis we described the principles, the design, and the implementation of a fully-configurable digital pulse detector. We studied existing methods for digital pulse shaping, height analysis, and pile-up rejection. We proposed a high throughput, low latency serial computation of the trapezoidal shaping function, using a dual filter setup and a custom algorithm for peak detection. We arranged the complete filter datapath in a three stage pipeline to further increase the performance of the system. We encapsulated the pulse detector into an AMBA IP core, and combined it with an on-chip processor, to arrive at a single-chip SoC solution. We created a small, lightweight device, for real-time, zero dead-time radiation monitoring and *gamma*-ray spectroscopy, keeping in mind the needs of current and future space exploration missions. The ability to run additional software on the embedded on-chip processor of the SoC adds to the flexibility and reusability of the design.

Although we chose to develop a digital pulse detector as a representative example of electronic instrumentation in space applications, our methodology is applicable to many other cases as well. Indeed, as the capacity of FPGAs increases, it is not hard to imagine a multitude of electronic instruments embedded within a single chip, under the same interconnection bus and the control of one or more on-chip processors. The recent introduction of mixed-signal FPGAs with embedded analog electronics takes the single-chip idea even further: such FPGAs embed the ADCs into the chip, and offer high-voltage tolerant analog inputs, effectively eliminating the need for external analog circuits.

Another recent trend in space electronics is the study of Micro-Electro-Mechanical Systems (MEMS) to further reduce the size of payloads. MEMS components integrate mechanical elements, sensors, actuators, and electronics in a single silicon chip. Together with mixed-signal FPGAs and Systems-on-Chip, MEMS technology completes the picture of, perhaps tomorrow's, truly single-chip highly-integrated payloads.

We believe our work is important because it highlights the potential benefits in the use of reconfigurable hardware in future space missions; by compacting multiple functions inside a complete SoC design, higher levels of integration can be achieved, while increasing the re-usability of the design and decreasing development time, effectively reducing the overall cost of the mission.

We conclude our discussion in the following sections with some final words on the problems that we encountered (Section 5.1), lessons learned (Section 5.2), and possible directions for future research on the subject (Section 5.3).

5.1 Problems Encountered

One of the most persisting problems throughout the various phases of the development process was to maintain the design FPGA-agnostic. Even if the developer strives to provide a clean, generic VHDL source code, he cannot avoid the addition of manufacturer-specific macros at the low level description of the system. A typical example is that of the embedded clock managers (detailed in Section 4.2.2) on the Xilinx Spartan3 FPGA. These DCMs play a vital role in generating the required clock signals for the various parts of our system. However, not all FPGAs provide such facilities, and even when they do, their instantiation might have to be done in a different way. The existence of the DCMS is unavoidable in this case, but they effectively "tie" our design to the specific FPGA family.

GRLIB, the IP core library of our prototype, deals with this problem by providing generic configurable instantiation blocks, together with a lower set of libraries for different FPGAs and development boards. This approach solves the issue, but requires constant updating of the low level libraries to support newer FPGAs. The only true solution to the problem would be a standardisation process, where different manufacturers would agree on a common way of providing the underlying resources. This process would also enable for portable, dynamically-reconfigurable designs, giving complete freedom to the developer to choose his/her hardware and tools, based on the requirements of the target system. By not having to go over a learning curve when switching FPGA models, and re-using existing libraries, one can also reduce the development time, and build on his/her previous knowledge to produce better designs every time.

Another cause for "tainting" the code with non-generic expressions is the current confusion when it comes to standard VHDL libraries. VHDL is defined by IEEE, together with a set of standard libraries, namely the packages "std_logic_1164" defined by IEEE 1164, "numeric_bit" and "numeric_std" defined by IEEE 1076.3, and "vital_timing" and "vital_primitives" defined by IEEE 1076.4. However, there exist other popular additions to the IEEE library, like the packages "std_logic_arith", "std_logic_signed", "std_logic_unsigned", and "std_logic_textio", that have been added without permission from IEEE and are not part of the standard. The definitions of symbols and operators between the different packages overlap, and can cause very obscure and hard to find bugs in the resulting design. When combining different cores, one should pay attention to the IEEE libraries used in each core.

Adding IP cores to existing libraries, or combining cores from different sources, is not a trouble-free process either. Apart from the problem of interfacing the different signal types (e.g. std_logic_vectors versus signed/unsigned, or bit vectors), once too often the developer has to go deep into the code of cores that he/she has not written, to track down compatibility problems. The automated tools that usually accompany a library, work with the predefined set of building blocks, but require modifications to include external cores, leading to increased development cycles.

An equally important issue is that of debugging the resulting system. It is common practice to develop code for FPGAs with less attention to detail and proper design procedures, because there is always the possibility to reprogram the device. However, a bug in a modern SoC, within a medium-sized FPGA, can be very difficult to trace and fix. As newer FPGAs offer greater capacities, the problem will only become worse. The proper tools can be invaluable in dealing with bugs, but the process can be very time-consuming. It is better to think of debugging during the initial steps of the design. We further discuss solutions for debugging in Section 5.3.

During the last steps of the development of our experimental system, when most of the components were in place, we noticed that it would take over 45 minutes to produce a single new version of the system, including synthesis, mapping, placement, and routing. This imposes an upper limit to the number of different alternatives one can try every day. In practice, if we also take into consideration the time that it takes to implement the changes and test the system, one cannot usually attempt more than two or three alternatives per day. The long waiting times also make the developer more impatient and prone to errors.

We have already mentioned the use of DCMs to provide the necessary clock signals to the FPGA and external analog circuits. Another problem related to the clock network is that of calculating and balancing the clock skew, particularly when it comes to external circuits. In our system, a 100MHz clock signal was provided by a single DCM to the external ADC, but also to some DACs that reconstructed the digital filter outputs for inspection on the oscilloscope. The ADC is "before" the FPGA, and thus requires the clock signal to have some negative skew to accommodate for the transmission delay, while the DACs are "after" the FPGA, and require the clock signal to have some positive skew. The DCM has only one output and a single skew adjustment. The solution is to place time constraints on the pins of the FPGA connected to the ADC and DACs, but the process of manually determining the correct skew is also time-consuming.

5.2 Lessons Learned

It seems there is never "enough" when it comes to careful thinking and planning of a complex project. Many times did we rush into implementing a new idea, only to find ourselves later spending long hours into trying to debug the problems. Nevertheless, the overall approach was organised and managed, and the result was indeed a functional digital pulse detector.

Performing simulations at every step of the development process, even when we expect to see the obvious, can be a very rewarding habit. We discovered that it is more time-efficient to create our own VHDL testbench files for every entity in a hierarchical way, and manage the execution of the testbenches within scripts. Establishing an automated simulation process from the early development steps, is also a good motive to run the simulations more often.

Another important lesson was on the use of timing and placement constraints. A design may exist as an idea within the lines of VHDL code, but it is the use of constraints that makes it possible to create a functional, real device. In our example, a synthesis tool would never succeed in achieving a 10ns clock period with the critical path passing through the 256:1 multiplexers, without the extra information that the select signals of the multiplexers are actually static.

In another case, regarding placement constraints this time, the orientation of the four DCMs on the corners of the chip greatly affects the overall arrangement, and thus the performance of the system, since the GRLIB AMBA IP cores will always be placed by the tools close to the 40MHz DCM, while our IP core will be closer to the 100MHz DCMs. The physical placement of the ADC, DACs and external SDRAM memory also affect our choice of arranging the DCMs.

Our approach to the development was a hardware/software co-design one. The basic idea behind our partitioning of the available functionality into hardware and software blocks, was that the hardware should provide the device mechanisms, while the software should dictate the policy of accessing these mechanisms. This kind of approach is also very rewarding in the long run, because it allows to use the hardware in ways that were not imagined during the initial planning.

Last but not least, our approach involved a development toolchain that was free and, in most of the cases, open-source. Throughout the project we gained experience in the use of these tools, and our result is proof of the fact that today it is possible to design and build prototypes of complex systems without investing a fortune in software licences.

5.3 Further Work

The outcome of our work was a functional digital pulse detector prototype. The obtained results using our prototype are very promising but there is always room for improvement.

For future work, the first issues to address would involve improving the hardware mechanisms. The current way of delivering results to the on-chip processor is rather "weak", with only a single 32-bit register available for intermediate storage. If the on-chip processor fails to read back fast enough, valuable results will be dropped. In a more mature approach, a dual-clock FIFO could be placed in between the two clock domains of our design, with the filter datapath producing results, and the AMBA block consuming them. To further improve the efficiency of the read-back process, an interrupt could be generated to signal the arrival of new data to the on-chip processor, eliminating the need for polling, and reducing the traffic over the shared interconnection bus.

An interesting addition to the available hardware mechanisms would be a form of boundary scan to ease the process of debugging. For this to happen, we need to connect all the registers of the pulse detector into a long chain and provide a serial output, together with a signal to select between normal and scan operation. We can then pause the execution of the datapath's pipeline at will, switch to scan mode, and examine the contents of all registers with the help of a logic analyser, or even transfer the contents back to the embedded software and out of the system, through the UART port.

In terms of software improvements, our system still needs a calibration mechanism, one that will allow the user to quickly determine the proper filtering parameters (window widths, gap widths, and thresholds) for the application at hand. Another improvement would be to offer to possibility to acquire a histogram without any source of events, and subtract it later from the real measurements, to accommodate for background noise. Other software improvements concern the ease-of-use of the control application.

Yet another direction for further work involves the complete system instead of the developed IP core only. In the future, we would like to add more modules to the SoC, ones that complement the existing functionality, as well as others for completely different scientific measurements, in order to evaluate the performance and needs of such a multi-functional system. Of course, we would have to use a larger FPGA for this purpose.

One module that we would like in particular to add to our SoC is that of a SpaceWire [16] port. SpaceWire is the new interconnection standard for cables and PCB tracks in space applications. It is essentially a revised version of the popular FireWire protocol, with minor improvements and radiation-hardened manufacturing. The European Space Agency provides a SpaceWire port IP core with an AMBA interface, which we can easily interface to our existing system, as a drop-in replacement for the UART port. SpaceWire will offer a high-bandwidth link (up to 400 Mbps), permitting multiple on-chip instruments to communicate with the main spacecraft computer at a high rate. Because of our managed SoC approach, we can replace the UART communication port of the system with a SpaceWire one, without any modifications in the rest of the hardware, and only update the running software.

Finally, instead of integrating more modules on-chip, another research initiative could look further into the usage of MEMS technology and mixed-signal FGPAs, to create truly single-chip instruments, with embedded sensors and analog circuits, placed side by side with the digital SoC on the same silicon substrate.

It is a safe bet to predict that in the future the capacity of new FPGAs will continue to grow. Therefore, the real question is not whether we can fit our existing designs into a given FPGA, but how to effectively use the ever-increasing available resources in the upcoming years, without having to undergo through increased development times. We hope that the ideas and facts presented in this text will serve as a good starting point for future, more complex, miniaturised systems.

Bibliography

[1] *Gaisler Research LEON3 synthesizable processor and GRLIB portable IP library*, [Online]. Available: `http://www.gaisler.com`.

[2] *GHDL, a complete VHDL simulator*, [Online]. Available: `http://ghdl.free.fr/`.

[3] *Gnuplot, a command-line driven interactive data and function plotting utility*, [Online]. Available: `http://www.gnuplot.info/`.

[4] *GTKWave,a fully featured GTK+ based wave viewer*, [Online]. Available: `http://home.nc.rr.com/gtkwave/`.

[5] *PENDER Electronic Design GmbH*, [Online]. Available: `http://www.pender.ch`.

[6] *The SPARC Architecture Manual Version 8*, [Online]. Available: `http://www.sparc.org/standards/V8.pdf`.

[7] *Suitability of reprogrammable FPGAs in space applications*, Feasability report n FPGA-002-01, Gaisler Research (2002).

[8] R. Abbiati, A. Geraci, and G. Ripamonti, *Self-configuring digital processor for online pulse analysis*, IEEE Transactions on Nuclear Science **51** (2004), no. 3 Part 3, 826–830.

[9] ARM limited, *AMBA specification, IHI 0011A*, 2.0 ed., 1999.

[10] M. Bautista-Palacios et al., *Configurable hardware/software architecture for data acquisition implementation on FPGA*, International Conference on Field Programmable Logic and Applications (2005), 241–246.

[11] M. Bolic and V. Drndarevic, *Digital gamma-ray spectroscopy based on FPGA technology*, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **482** (2002), no. 3, 761–766.

[12] S. Buzzetti, M. Capou, C. Guazzoni, A. Longoni, R. Mariani, and S. Moser, *High-Speed FPGA-Based Pulse-Height Analyzer for High Resolution X-Ray Spectroscopy*, IEEE Transactions on Nuclear Science **52** (2005), no. 4, 854–860.

[13] CANBERRA, *Performance of digital signal processors for gamma spectrometry*, [Online]. Available: `http://www.canberra.com/pdf/Literature/a0338.pdf`.

[14] JM Cardoso et al., *A high performance reconfigurable hardware platform for digital pulse processing*, IEEE Transactions on Nuclear Science **51** (2004), no. 3 Part 3, 921–925.

[15] B. Esposito, M. Riva, D. Marocco, and Y. Kaschuck, *A digital acquisition and elaboration system for nuclear fast pulse detection*, Nuclear Inst. and Methods in Physics Research, A **572** (2007), no. 1, 355–357.

[16] European Cooperation for Space Standardization, *Spacewire - links, nodes, routers and networks, ecss-e-50-12a*, January 2003.

[17] S. Habinc, *Designing Space Applications Using Synthesisable Cores*, MAPLD 1999-Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference, 2nd, Johns Hopkins Univ, APL, Laurel, MD (1999).

[18] VT Jordanov, DL Hall, M. Kastner, GH Woodward, and RA Zakrzewski, *Portable radiation spectrometer using low power digital pulseprocessing*, Nuclear Science Symposium, 1999. Conference Record. 1999 IEEE **1** (1999).

[19] S. Kraft et al., *On the study of highly integrated payload architectures for future planetary missions*, Proceedings of 11th SPIE International Symposium on Remote Sensing (2005).

[20] CFM Loureiro and C Correia, *Innovative modular high-speed data-acquisition architecture*, IEEE Transactions on Nuclear Science **49** (2002), no. 3 Part 2, 858–860.

[21] DMC Odell, BS Bushart, LJ Harpring, FS Moore, and TN Riley, *Zero deadtime spectroscopy without full charge collection*, 9. symposium on radiation measurements and applications, Ann Arbor, MI (United States) (1998).

[22] B. Parhami, *Computer arithmetic*, Oxford University Press (2000), 128–131.

[23] G. Ripamonti, A. Pullia, and A. Geraci, *Digital vs. analogue spectroscopy: a comparative analysis*, Instrumentation and Measurement Technology Conference, 1998. IMTC/98. Conference Proceedings. IEEE **1** (1998).

[24] W.K. Warburton and P.M. Grudberg, *Current trends in developing digital signal processing electronics for semiconductor detectors*, Nuclear Inst. and Methods in Physics Research, A **568** (2006), no. 1, 350–358.

[25] W.K. Warburton, M. Momayezi, B. Hubbard-Nelson, and W. Skulski, *Digital pulse processing: new possibilities in nuclear spectroscopy*, Applied Radiation and Isotopes **53** (2000), no. 4-5, 913–920.

[26] N.H.E. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective, 3rd edition*, Pearson/Addison-Wesley (2005), 454–463.

[27] Xilinx, *DS099 – Spartan-3 FPGA Family: Complete Data Sheet*, [Online]. Available: `http://direct.xilinx.com/bvdocs/publications/ds099.pdf`.

[28] Xilinx, *XAPP462 – Using Digital Clock Managers (DCMs) in Spartan-3 FPGAs*, [Online]. Available: `http://www.xilinx.com/bvdocs/appnotes/xapp462.pdf`.

A List of registers

This is a list of all registers and commands implemented on the radiation monitor.

A.1 Configuration/Status register

field	*reserved*	**DATA_RDY**	**STATE**	**RECONF**	**START**	**RESET**
bits	31 to 6	5	4 to 3	2	1	0
access	-	R	R	R/W	R/W	R/W

Table A.1: Configuration/Status register, offset 00000H

This register is used for configuration and status retrieval of the device. Its main use is to reset and start/stop the device, but also to monitor its state (ex. reading back the contents of the register after a reconfiguration command, while waiting for the system to return to the "idle" state).

[**RESET**] Setting this bit to 1 initiates a soft reset.

[**START**] Setting this bit to 1 enables processing of the input samples. Writing a 0 stops the processing.

[**RECONF**] Setting this bit to 1 updates the filter settings using the values stored in the slow and fast filter configuration and thresholds registers. Upon completion of the process, this bit is automatically set back to 0.

[**STATE**] This read-only field reflects the state of the AMBA control block, as follows:

RESET	=	00
IDLE	=	01
RUNNING	=	10
RECONFIGURING	=	11

[**DATA_RDY**] A one in this bit position marks the arrival of new data in the results register. Reading from the results register will clear this bit.

A.2 Slow filter configuration register

field	**WIN1_OLD**	**WIN1_NEW**	**WIN2_OLD**	*reserved*
bits	31 to 24	23 to 16	15 to 8	7 to 0
access	R/W	R/W	R/W	-

Table A.2: Slow filter configuration, offset 00001H

This register holds the parameters of the slow filter. All fields are 8-bit wide. Writing to this register does not change the filter parameters on the fly. The user should issue a reconfiguration command after changing the fields of this register, for the changes to take effect.

[**WIN2_OLD**] Position of oldest sample in second window.

[**WIN1_NEW**] Position of newest sample in first window.

[**WIN1_OLD**] Position of oldest sample in first window.

The newest sample in the second window is omitted (reserved bits 7 downto 0), since it always corresponds to a zero value (first sample in FIFO memory).

A.3 Fast filter configuration register

field	**WIN1_OLD**	**WIN1_NEW**	**WIN2_OLD**	*reserved*
bits	31 to 24	23 to 16	15 to 8	7 to 0
access	R/W	R/W	R/W	-

Table A.3: Fast filter configuration, offset 00010H

The mechanics of this register and meaning of its fields are identical to the slow filter configuration register.

A.4 Slow filter thresholds register

field	**upper threshold**	**lower threshold**
bits	31 to 16	15 to 0
access	R/W	R/W

Table A.4: Slow filter thresholds, offset 00011H

This register holds the upper and lower thresholds of the slow filter. Both fields are 16-bit wide. All values should be in two's complement format. Writing to this register does not change the filter parameters on the fly. The user should issue a reconfiguration command after changing the fields of this register, for the changes to take effect.

The lower threshold should always be equal to or smaller than the upper threshold (in absolute value), otherwise no pulses will be detected.

A.5 Fast filter thresholds register

field	**upper threshold**	**lower threshold**
bits	31 to 16	15 to 0
access	R/W	R/W

Table A.5: Fast filter thresholds, offset 00100H

The mechanics of this register and meaning of its fields are identical to the slow filter thresholds register.

A.6 Result register

field	**slow filter peak**	**fast filter peak**
bits	31 to 16	15 to 0
access	R	R

Table A.6: Result register, offset 00101H

This read-only register holds the last valid result obtained from the pulse detector. It includes two fields, the slow and fast filter peaks. All fields are 16-bit signed numbers. Reading from this register will clear the **DATA_RDY** bit of the configuration/status register.

Under normal circumstances only the slow filter peak value is useful, since it contains the same information but with better resolution. The fast filter peak value is included for testing purposes and might be removed in future versions.

B Sample embedded C code

The following code snippet is from the embedded C code that controls the pulse detection process. We present here three functions: *conf_filter()* uses a default set of values to configure the filter parameters, *get_events()* asks the user how many events should be captured and then proceeds to retrieve them from the filter and form the histogram. Finally the *transfer_results()* function prints the histogram data on the output, which is redirected to the system's serial port.

```
#define DFILTER_BASEIO 0x80000500
#define DFILTER_REGNUM 5
#define CHANNELS 32768

unsigned long events;
unsigned long event_matrix[CHANNELS];

unsigned long *amba_regs = DFILTER_BASEIO;

void conf_filter()
{
  int slow_win_width, slow_win_gap;
  int slow_up_thres, slow_dn_thres;
  int fast_win_width, fast_win_gap;
  int fast_up_thres, fast_dn_thres;

  unsigned long slow, fast, s_th, f_th;

  slow_win_width =  100;
  slow_win_gap   =   30;
  fast_win_width =   50;
  fast_win_gap   =   30;

  slow_up_thres  = 2048;
  slow_dn_thres  = 1024;
  fast_up_thres  = 1024;
  fast_dn_thres  =  512;

  slow  = 0;
  slow |= slow_win_width << 8;
  slow |= (slow_win_width   + slow_win_gap) << 16;
  slow |= (2*slow_win_width + slow_win_gap) << 24;
  fast  = 0;
  fast |= fast_win_width << 8;
  fast |= (fast_win_width   + fast_win_gap) << 16;
  fast |= (2*fast_win_width + fast_win_gap) << 24;

  s_th  = slow_dn_thres;
  s_th |= slow_up_thres << 16;
  f_th  = fast_dn_thres;
  f_th |= fast_up_thres << 16;

  // Update register values
  amba_regs[1] = slow;
  amba_regs[2] = fast;
  amba_regs[3] = s_th;
  amba_regs[4] = f_th;

  while ((amba_regs[1] != slow) &&
         (amba_regs[2] != fast) &&
         (amba_regs[3] != s_th) &&
         (amba_regs[4] != f_th));

  // Send command to start reconfiguration
  amba_regs[0] = 4;
}


void get_events(void)
{
  long conf_status, result;
  int i, times;

  printf("how many events? : ");
  scanf("%d", &times);
  printf("%d\n", times);

  for (i=0; i < CHANNELS; i++)
    event_matrix[i]=0;

  events = 0;

  while(events < times){
    conf_status = amba_regs[0];

    if ((conf_status & 0x20) == 0x20){
      result = amba_regs[5];

      event_matrix[result >> 16]++;

      events++;
    }
  }
}

void transfer_results(void)
{
  int i;

  for (i=0; i < CHANNELS; i++)
    if (event_matrix[i] != 0)
      printf("%d, %d\n", i+1, event_matrix[i]);
}
```