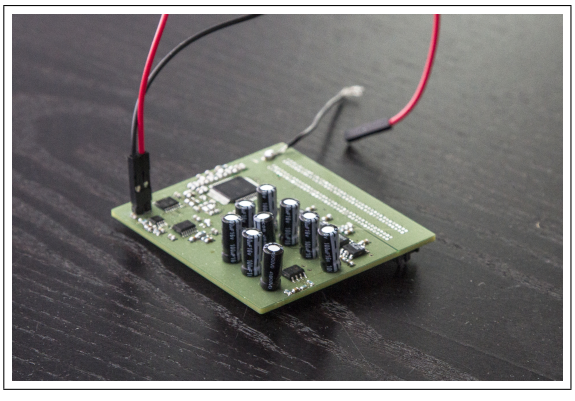
LULEÅ UNIVERSITY OF TECHNOLOGY

Master's Thesis in Engineering Physics and Electrical Engineering

Towards Self-Learning Sensors: FPGA-Based ADC Front End

Author:
Joakim Nilsson

Supervisor:
Jens Eliasson



August 29, 2013

(last updated February 8, 2014)

Abstract

With the amount of sensors in the environment ever increasing, high demands are being posed on today's sensor systems in terms of power consumption, data compression and cost. This thesis presents the work of constructing and evaluating an FPGA-based ADC front end suitable for running demanding signal processing algorithms on it with data rate reduction as its primary goal. A prototype of the front end is built and demonstration software is written demonstrating the feasibility of it being able to handle a matching pursuit-based algorithm which allows for sparse representations of signals for data rates over 1 MHz. This assessment is done by evaluating the front end in terms of noise, power consumption and speed and also by the construction of a test application, an FIR filter bank which is related and compared to an FPGA implementation of matching pursuit. It is also concluded that for the system described in this thesis, an ASIC design may be more suitable than an FPGA design because of the higher power consumption, lower speed and higher per-unit-cost of FPGAs in comparison to ASICs.

Preface

I would like to thank my supervisor, Jens Eliasson, and his colleague, Fredrik Sandin, for inspiring me to do this project, for discussing it, for their opinions on this thesis and for being inexhaustible sources of information in general.

I also want to thank my classmate, Kim Albertsson, for doing a related project and for many interesting discussions about our both projects (and about loads of other stuff as well).

Moreover, I want to thank Johan Borg for his willingness to share his exceptionally good knowledge about electronic construction.

Other people I want to thank include everyone who is a part of SKF University Technology Centre, which is the place at which most of the work carried out in this thesis took place. Of those people, Sergio Martin del Campo Barraza and Fredrik Häggström deserve special mentions.

I also want to thank the nice people at the ZedBoard forums for their guidance in the field of FPGA programming.

Last but not least, I want to thank my family, Thomas, Marita and Jenny Nilsson for supporting me during my studies and, more importantly, for being a great family to grow up in.

Joakim Nilsson, 2013

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many of today's sensors transmit their information wirelessly as a wireless communication link is often easier to set up physically than a communication link through cables. For instance, a sensor can be mounted on a rotating part, making it very hard to draw cables to it. There are two main problems with cable-free sensors. The first problem is that they need to either be battery powered or gather their energy from the environment. Because of this problem, such sensors should be very energy efficient as to require as little maintenance as possible. The second problem is the amount of data that is being sent by them. This data congestion problem has been examined in an article by Hull, Jamieson and Balakrishnan in 2004 [1]. The naïve way to send data would be to just send raw samples directly. However, one can also send compressed data to reduce the data rate by doing some processing on the sensor system. Because of the huge numbers of sensors used in some applications, the problem of high amounts of data also exists for the wired case, though.

"Matching pursuit", introduced by Mallat and Zhang in 1993 [2], is an algorithm which allows for succinct representations of signals and may therefore be an appropriate way of compressing sensor data. Matching pursuit describes a signal as a weighted sum of generic functions, called kernels. This algorithm has proved to be an efficient way of encoding both natural sounds and images. In 2004, Olshausen and Field [3] showed that matching pursuit with dictionary learning can be used to sparsely encode natural images, and in 2006, Smith and Lewicki [4] showed how to sparsely encode mammalian sounds using matching pursuit with dictionary learning. The dictionary learning part of matching pursuit means that a probabilistic method is used to generate kernels that together provides a means of accurately describing the signal that is being processed. A field-programmable gate array (FPGA) implementation of matching pursuit with dictionary learning has been simulated in a parallel project made by Albertsson [5].

The international company SKF [6], which among other things manufacture bearings, mechatronic products and lubrication systems, uses acoustic emission analysis for condition monitoring of machine components like bearings. They have found that acoustic emission signals from some of their bearings have interesting features at frequencies of several hundreds of kilohertz that are not well understood [7]. Due to the high frequencies and non-trivial structure of these signals, a system to efficiently encode and transmit data produced from

an acoustic emission sensor could be of good use to SKF.

A sensor system with the capabilities of performing the event-based matching pursuit algorithm described by Smith and Lewicki [4] might achieve low power consumption and low data rates and might therefore be suitable for monitoring the bearings just mentioned. An idea arising from this observation is that a "more intelligent" analog-to-digital converter (ADC) or an analog-to-feature converter (AFC), can possibly be created. This component would take as input an analog signal and output a digital representation of the input signal that is a compressed version of what an ordinary ADC would output. Using matching pursuit would enable the AFC to describe an analog signal in terms of its features instead of the signal's raw digital representation as an ordinary ADC would. (For more information about these ideas, see [8].)

Due to matching pursuit's significant requirements of processing power, a microcontroller-based AFC seem not to be appropriate for the task. An application specific-integrated circuit (ASIC) component may be more appropriate. To demonstrate the concept and to lay the groundwork for producing an ASIC that is able to do what was just described, this thesis focuses on a sensor front end based on a FPGA that is capable of running matching pursuit.

Some questions thus arise; Can a sensor front end be developed with embedded FPGA-based processing capabilities that is capable of performing signal processing algorithms like matching pursuit on high frequency signals? Can such algorithms be run on such a system on signals with interesting frequency content of as high frequencies as hundreds of kilohertz or even in the megahertz range? Can such a system achieve a low power consumption? How accurately can such a system sample data?

If an AFC can be developed, it would have implications in several fields of research. The development cost and per-unit-cost of condition monitoring systems could be reduced while at the same time the power efficiency could be increased. Improved power efficiency would in turn benefit the environment. Although this thesis has condition monitoring of bearings through analysis of acoustic emission as its reference application, there are several other fields of research that would benefit from this too, two such areas being image and speech recognition which could also see reduced costs and improved efficiencies.

This thesis demonstrates the work of building an ADC front end interfacing an FPGA with the purpose of being a general purpose signal processing platform. This front end is then evaluated and test applications are written for it to demonstrate its processing capabilities in order to investigate the questions listed above.

1.1 Goals

The primary objective of this thesis is to construct an ADC front end that is capable of performing advanced signal processing algorithms, in particular the event-based matching pursuit algorithm that Smith and Lewicki described in 2006 [4], on an FPGA. A demonstration of the system running an FIR filter bank should be made so that its processing capabilities can be estimated.

1.1.1 Scope

There are some limitations regarding the goals of this thesis; These are the following:

- While the ADC front end should leave the possibility open to design for low power, the primary goal of all demonstrations in this thesis is computational speed.
- A prototype of the ADC front end and not a ready-for-production unit should be produced. The step should be small, though, to go from the prototype to a final product.

1.2 Motivation

There are several possible applications for the proposed ADC front end. Some of them are presented in this section and all of the presented ones are using the event based matching pursuit algorithm described by Smith and Lewicki in 2006 [4] as this is the reference algorithm used to measure the processing capabilities of the system.

1.2.1 Condition monitoring of bearings

Acoustic emission

A possible application for the ADC front end proposed in this thesis is within the field of condition monitoring of bearings through analysis of acoustic emission. SKF [6], a company producing, among other things, bearings and sensors for bearings, examines the acoustic signal emitted from their bearings in order to detect defects so that a bearing can be replaced before it breaks. By doing this they can prevent it from causing damage to surrounding machinery or people. A problem within this field is that the acoustic signal is of very high frequency and therefore, if the data is transmitted uncompressed, very high data rates are reached. It is also desirable to place the senor as close to the bearing as possible as the signal is otherwise misrepresented due to the attenuation and distortion introduced in the signal while it propagates through different materials. Therefore the sensor is often placed inside the bearing and the sensor signal must therefore be transmitted wirelessly. With the ADC front end proposed in this thesis and the event-based matching pursuit algorithm described by Smith and Lewicki in 2006 [4], the data rate can be greatly reduced which in turn reduces demands on the routing system responsible for forwarding the sensor data to a computer as well as lowering the power consumption of the wireless transmitters.

Vibration analysis

Analysis of lower frequency vibration measurements using matching pursuit have been proved in 2002 by Liu, Ling and Gribonval [9] to also be a good method for performing condition monitoring on bearings. Therefore, even though the sensor front end would prove to be too weak to be able to do matching pursuit to analyze acoustic emission, it may still very well be suitable for vibration analysis to detect bearing faults.

1.2.2 Earthquake detection

In 2001, Alperovich, Zhelude and Hayakawa mentioned a use case of matching pursuit in the field of earthquake detection. By implementing the same algorithm as for the condition monitoring of bearings example, the sensor system proposed in this thesis could possibly be used to detect earthquakes before they strike. Several sensors could be placed in the area of interest and, if sufficiently low power consumption can be achieved, be powered by solar panels. Earthquake detection is very useful because it enables precautions to be taken a short period of time before the earthquake strikes. For example in a mine, workers could move to safety rooms just before the strike.

1.2.3 Speech recognition

Wang and Goblirsch showed in 1997 [10] that matching pursuit can be used for speech recognition. Speech recognition implemented on the ADC front end proposed in this thesis could produce a system able to detect and forward calls for help in environments where there are no people to hear the calling person. This can be useful for many workers who work in loud or solitary environments.

1.3 Thesis structure

This thesis is organized as follows:

- Chapter 1 (this chapter) provides an introduction to the subject and presents some motivation and goals for this thesis as well as the thesis structure and a license notice.
- Chapter 2 presents a way in which the ADC front end can be designed as well as the theory behind the design.
- Chapter 3 presents the resulting ADC front end implementation and test applications written for it.
- Chapter 4 discusses and evaluates the different features of the ADC front end test applications.
- Chapter 5 contains a glossary of the abbreviations used in this thesis.
- Chapter 6 contains references to literature referred to in this thesis.
- Appendix A contains a specification of the ADC front end.
- Appendix B contains parts of the source code developed in this thesis.

1.4 Licensing of work

The source code that is part of the work presented in this thesis is released under the GNU General Public License, version 3 [11]. All other work presented in this thesis, including schematics, is free for anyone to use in any way they want as long as credit is given. The full source code and design files of this thesis are available upon request to the author.

Chapter 2

Theory

The ADC front end that this thesis demonstrates the construction of consists of two main parts. The first is an analog part which performs amplification and low pass filtering on an incoming signal. The second part consists of an ADC and an FPGA. The analog filtered signal is sampled by the ADC, a digital low pass filter is applied on it from within the ADC and then a digital representation of the signal is sent to the FPGA. The FPGA then performs processing on the digital signal and forwards its results through a universal asynchronous receiver/transmitter (UART) port which can either be read by a computer directly or forwarded to a computer by a wireless communication device.

2.1 FPGA versus other computing methods

It is not obvious that an FPGA is the optimum choice of computing device. Therefore, some alternatives are discussed in this section.

2.1.1 Microcontroller

Due to the high demands of processing power of the algorithm described by Smith and Lewicki in 2006 [4] (which is mentioned in one of the design goals of the ADC front end, see Section 2.2.1) a microcontroller lacks the computational capabilities required for this project. An attempt has been made in a parallel project by Albertsson [5] on a laptop with a 1.8 GHz dual-core Intel Core i5 (Turbo Boost up to 2.8 GHz) with 3MB shared L3 cache (which is far more powerful than most microprocessors). During this attempt information could only be processed with a sample rate no higher than around 27 kHz which is too slow for many applications.

2.1.2 Digital signal processor

A digital signal processor (DSP) is specifically designed to run signal processing algorithms like digital filters fast. This property makes them suitable for performing the algorithm described by Smith and Lewicki in 2006 [4]. However, an FPGA solution has another advantage; Converting from an FPGA design to an ASIC is relatively easy and could increase the processing speed and decrease the

power consumption significantly. An FPGA is also suitable for a wider range of applications compared to a DSP due the high flexibility of FPGAs.

2.2 ADC front end

In this section, goals for the ADC front end are first presented and then follows an overview of the design and some theory of its different parts and how they interact with each other.

2.2.1 Design goals

For the design goals of this project the acoustic emission example described in Section 1.2.1 was chosen as a reference application. This means that one goal of the ADC front end was set to be that it be able to handle the matching pursuit event-based algorithm described by Smith and Lewicki in 2006 [4] on signals with interesting frequency content of up to 1 MHz. Other design goals are:

- Because the signal to be fed into the system can be of varying strength, the front end should contain variable amplification functionality so that the signal can be amplified before it is sampled by the ADC.
- To avoid aliasing, high frequency content (above 1 MHz) must be filtered out from the signal before it is sampled by the ADC.
- The front end must be able to handle differential input so that a common ground between the front end and the sensor does not need to be established.
- The front end must provide output samples with at least 16 noise-free bits of accuracy at an output data rate of at least 2 million samples per second.

2.2.2 Amplification system

Because the source resistance of the device that is producing the input signal is unknown, the first stage in the ADC front end is a buffering amplifier. The output of the buffering amplifier, let it be denoted S_0, is fed to a programmable gain amplifier (PGA) which is controlled by the FPGA. The PGA allows for FPGA controlled gain-control as the PGA amplifies S_0 by a factor controlled by the FPGA.

Let the output of the PGA be denoted S_1. Both S_0 and S_1 are fed to a multiplexer (MUX) which is also controlled by the FPGA. The multiplexer is there so that the system quickly can switch from an amplified signal to an unamplified one.

The output of the multiplexer is then fed to a first-order active analog low pass filter which in turn sends the signal to the ADC. The ADC communicates with the FPGA trough a parallel digital interface. An abstract overview of how the different components are connected is presented in Figure 2.1.

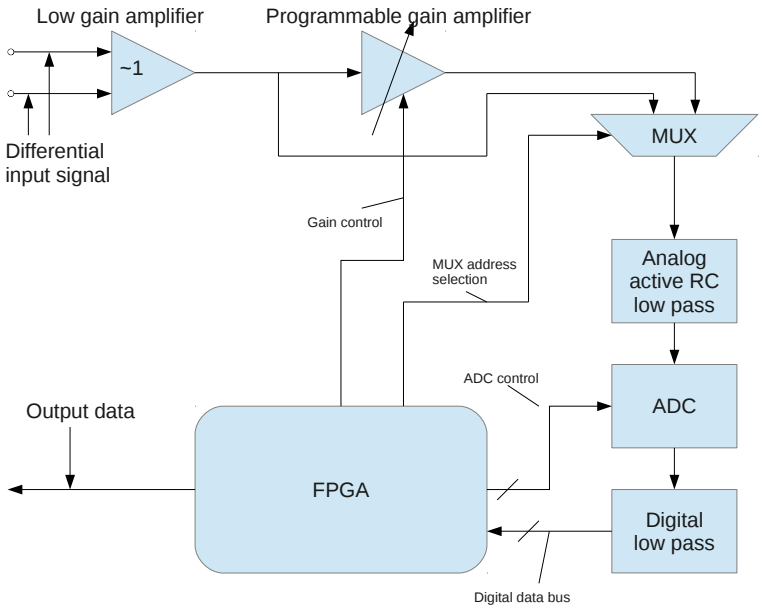


Figure 2.1: Block diagram of the ADC frontend.

2.2.3 Filters

To avoid the effects of aliasing, the input signal must be low pass filtered before it is used. Filtering can be done both analogically and digitally with both methods having their advantages and disadvantages. One advantage with an analog filter is that it can filter out all frequencies lower than an arbitrary break frequency. This is not the case with a digital filter whose highest filterable frequency is dependent on the sampling frequency. This is a result of the fact that a digital signal is a representation of the original signal *after* it has been sampled. On the other hand, a steep frequency response with an analog filter is hard to achieve without high amounts of electronic components. (Steep frequency responses are desirable because non-steep frequency responses will distort the signal at frequencies near the break frequency.) This is not the case with digital filters with which it is easy to achieve steep frequency responses. Due to the just mentioned advantages and disadvantages of each filter type, a combination of an analog and a digital filter is suitable for keeping the component count low (which is beneficial for cost and size) while still achieving a steep frequency response. The break frequency of the digital filter would in that case be set at the desired data rate while the break frequency of the analog filter would be set so that all the aliasing effects introduced by sampling would be filtered out by the digital filter (which could be at the same frequency as in the case of the digital filter depending on the shape of the frequency response of the analog filter). A conceptual illustration of the frequency response of such a combination of an analog and digital filter is presented in Figure 2.2.

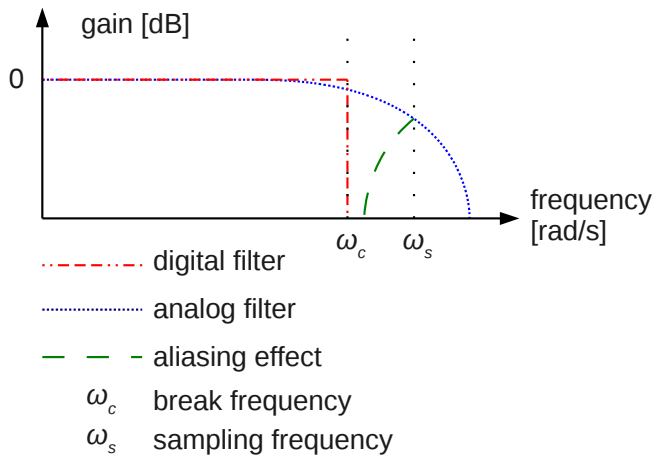


Figure 2.2: Frequency responses of an analog and a digital low pass filter both having break frequency ω_c. The effects of aliasing on the frequency response is also shown for a sampling frequency of ω_s. The figure illustrates that if ω_s is chosen appropriately, the aliasing effects arising from an analog filter can be filtered out by a digital filter.

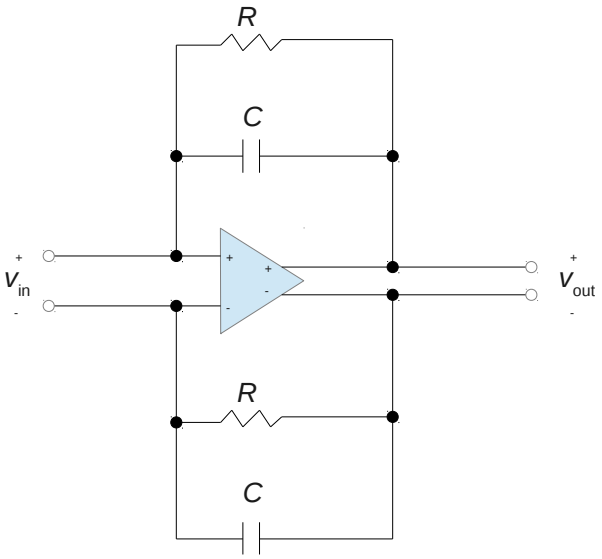


Figure 2.3: Schematic diagram of a matched electronic first-order active low pass filter with fully differential signal path having break frequency, $f_c = 1/(2\pi RC)$. v_{in} is the incoming voltage signal and v_{out} is the outgoing voltage signal.

Analog filter

A matched active low pass filter can be constructed according to the schematic diagram shown in Figure 2.3. Such a low pass filter would have break frequency, f_c, according to Equation 2.1 as any first order low pass RC filter would [12]. Here, R and C are the resistance and the capacitance, respectively, of the resistors and the capacitors shown in Figure 2.3. Due to the fact that the filter is matched, it is important that its resistors and capacitors are of high precision.

$$f_c = \frac{1}{2\pi RC} \tag{2.1}$$

Digital filter

A finite impulse response (FIR) filter is capable of low pass filtering digital data. An FIR filter consists of a number of coefficients, b_n, which are multiplied by the input signal delayed by n samples to produce its output. The number of coefficients equals the filter's order. An FIR filter with a given break frequency is easy to generate with computer tools like GNU Octave [13] or MATLAB [14]. The mathematical definition of an FIR filter is described in Equation 2.2. Here, $y[n]$ is output sample n, $x[n]$ is input sample n, b_n is coefficient n and k is the filter order.

$$y[n] = \sum_{i=0}^{k} x[n-i]b_i \tag{2.2}$$

2.3 FPGA FIR filter bank

To demonstrate that the ADC front end being constructed in this thesis is suitable for applications like the acoustic emission example described in Section 1.2.1, an FIR filter bank can be implemented on the FPGA as such a bank can be compared in relation to the event-based matching pursuit algorithm described by Smith and Lewicki in 2006 [4]. (For information on FIR filters, see Section 2.2.3.)

2.3.1 Relation to matching pursuit

The innermost loop of the event-based matching pursuit dictionary learning algorithm described by Smith and Lewicki in 2006 [4] takes care of projections of kernel functions. These kernel functions can be carried out in parallel through a bank of FIR filters. The gains obtained from implementing an FIR filter bank on an FPGA should therefore approximate the gains achieved by implementing event-based matching pursuit on an FPGA.

2.3.2 FIR filter design

Highly parallel FIR filters can be constructed on an FPGA by building a separate multiplier for each value-coefficient pair and feeding the resulting products into an adder tree which sums them up. An effect of this would be that one iteration of an FIR filtration can be done in one cycle with a delay introduced from the pipelined structure of the adder tree. This delay will be of as many cycles as the height of the adder tree. Assume there are n value-coefficient pairs, i.e. that the order of the FIR filter is n. Then the adder tree's height equals $\lceil \log_2 n \rceil$ since an adder tree is built the same way a balanced binary tree is (for an introduction to balanced binary trees, see [15]). Since the entire FIR filter structure is pipelined, the clock frequency could be set as high as it could be for one multiplication instruction (as multiplication is the most time consuming operation of this structure).

This method of implementing an FIR filter on an FPGA is shown graphically in Figure 2.4. A bank of FIR filters could simply be implemented by putting several FIR filters in parallel.

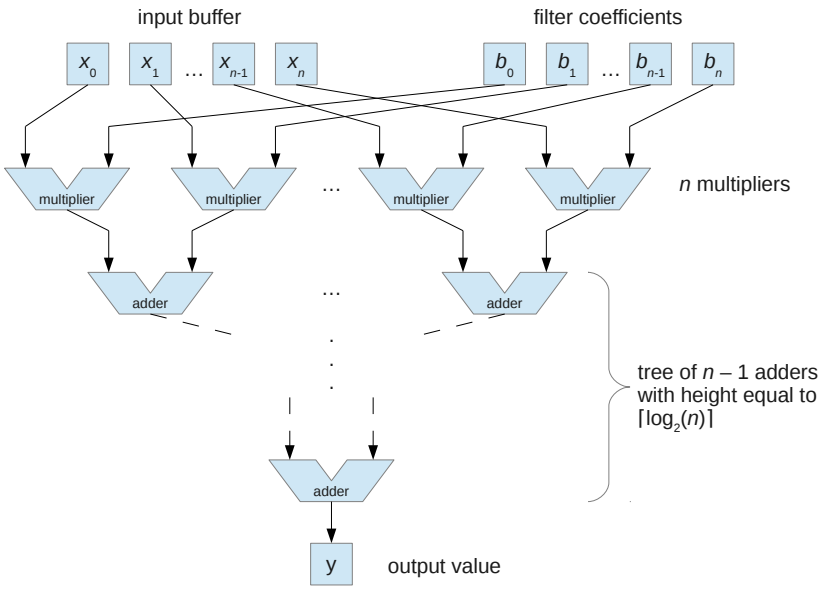


Figure 2.4: FPGA FIR filter implementation block diagram. This figure is also available in [8].

Chapter 3

Implementation

In this chapter, the resulting implementation of the theory from Chapter 2 is presented.

3.1 ADC front end

The ADC front end board was designed with KiCad which is an open source suite of electronic design automation (EDA) tools. The resulting design and performance measurements are presented in this section.

3.1.1 Design

An overview of how the front end was designed is presented in Figure 3.1 while the detailed specification is presented in Appendix A. A board was constructed to interface with an FPGA development board. In this section, the designs of the different parts of that board are presented.

Amplification system

All references to net names and components in this section refer to the schematic presented in Figure A.1.

The amplification system is the first stage of the ADC front end and consists of two main components: an instrumentation amplifier and a PGA. The input signal is first fed to the instrumentation amplifier whose primary task is to isolate the source resistance from the load resistance of the front end. It is also amplifying the input signal with a gain of 1.99. The gain of the instrumentation amplifier can be changed by replacing resistor `Rg1` (for details on how, see the datasheet of the instrumentation amplifier listed in Table A.1). The output signal from the instrumentation amplifier, let it be denoted S_0, is fed to the PGA. A pulse width modulated (PWM) signal, `pwm`, is used to control the PGA. `pwm` is generated by the FPGA. `pwm` is converted from a PWM signal to an analog voltage signal through the low pass filter consisting of resistor `R1` and capacitors `C1` and `C2`. This analog voltage signal is then used to control the PGA. The higher the voltage, the more the PGA will amplify S_0. Call the output from the PGA S_1. S_0 and S_1 are connected to the differential nets, {`v+_pga`, `v-_pga`} and {`v+_instr_amp`, `v-_instr_amp`}, respectively.

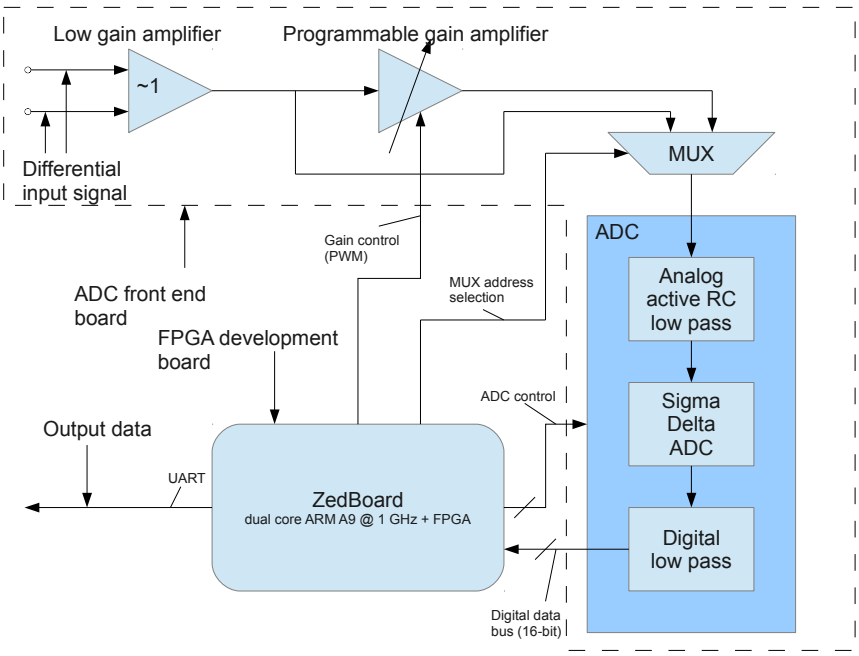


Figure 3.1: Overview of the ADC front end.

ADC system

All references to net names and components in this section refer to the schematic presented in Figure A.2.

The outputs, S_0 and S_1, connected to the differential nets, {`v+_pga`, `v-_pga`} and {`v+_instr_amp`, `v-_instr_amp`}, are connected to a multiplexer. This multiplexer is controlled by the digital signal, `mux_sel`, which is generated by the FPGA. This signal selects which of S_0 and S_1 is fed to the active low pass filter consisting of a built in amplifier inside the ADC component, resistors `R11` and `R12`, and capacitors `C25` and `C26`. The break frequency of the low pass filter was set with the values presented in Table 3.1 resulting in a break frequency of 1.012 MHz according to Equation 2.1, as it was designed according to the schematic presented in Figure 2.3. The output of the low pass filter is sampled by the ADC and sent to the FPGA through a 16-bit digital interface consisting of a control bus, `fpga_adc_ctrl`, and a data bus, `fpga_adc_data`. Before it is sent to the FPGA it is filtered by a digital low pass filter with a break frequency of 1 MHz that is built into the ADC.

FPGA

Component The FPGA component, a Xilinx Zynq Z7020, used in this thesis is the one that is mounted on the ZedBoard FPGA development board which is the development board used in this thesis to interface the FPGA to the ADC. Xilinx claims that this FPGA has 85,000 equivalent logic cells of FPGA fabric and an embedded hard dual-core ARM processor Cortex A9.

Table 3.1: Resistor and capacitor values for the low pass filter consisting of resistors `R11` and `R12` and capacitors `C25` and `C26` in the schematic describing the ADC system of the ADC front end being presented in Figure A.2.

Component	Value
`R11`	655 Ω
`R12`	655 Ω
`C25`	240 pF
`C26`	240 pF

Designs for this FPGA were implemented with a tool suite provided by Xilinx named ISE Design Suite [16].

Schematics All references to net names and components in this section refer to the schematic presented in Figure A.3.

The FPGA component shown in the schematic is not an actual FPGA, but an expansion connector which can be connected to the FPGA development board described in the previous paragraph. The FPGA (which is located on the development board) is generating a PWM signal, `pwm`, controlling the PGA. It is also generating control signals, `fpga_adc_ctrl`, to perform read/write operations on the ADC through the data bus, `fpga_adc_data` and it is generating a clock signal, `mclk`, to the ADC and a digital signal, `mux_sel`, to control which of the signals S_0 and S_1 that is passed through the multiplexer. A voltage level shifter, `LevelShifter1`, translates the 2.5 V signals the FPGA is generating to 5 V signals, `mux_sel` and `mclk`, that the multiplexer and the PGA are being controlled with.

Power supply

All references to net names and components in this section refer to the schematic presented in Figure A.4.

An input voltage of 12 V, `D12V`, is generated from the FPGA development board. This voltage is used to generate new voltages on the board. These voltages are:

- `+5V`, used to provide the analog components with a voltage of 5 V.
- `+2.5V`, used to provide the analog components with a voltage of 2.5 V.
- `D5V`, used to provide the digital components with a voltage of 5 V.
- `D2.5V`, used to provide the digital components with a voltage of 2.5 V.
- `2V5ref`, used to provide a stable voltage reference of 2.5 V so that the ADC can take accurate samples.

Board design

Design files for a 57 × 65 mm, four layer board was sent in to Cogra [17] who manufactured the board. The top layer contained two ground planes (analog and digital). These can be seen in Figure A.5. The upper ground plane is the

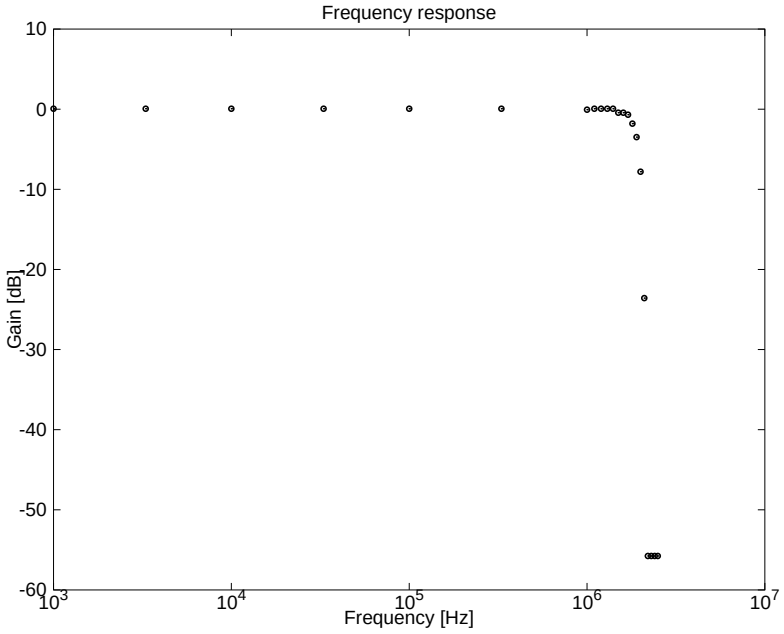


Figure 3.2: Measured frequency response of the ADC front end.

digital one and the other is the analog. All the components were placed at the top layer except the expansion connector, `U1` (see Figure A.3), which was placed at the bottom layer. For more details about the board, see Appendix A.3.

3.1.2 Performance

In this section, measured properties like frequency response, noise and power consumption are presented as well as an erratic behavior of the system at high amplitudes of the input signal.

Frequency response

The gain was measured at different frequencies between the input signal and the input to the ADC. In order to measure the gain at different frequencies, different sinusoids of different frequencies and known amplitude was fed to the input of the front end by a signal generator. The amplitude of the resulting output sinusoids (measured at the input to the ADC with an oscilloscope) was then noted. To obtain the gain, the output amplitudes were divided by the input amplitude and then converted to decibels by taking the logarithm of the result and then multiplying by 20. The gain was normalized to be 0 dB at its peak value. More measurements were taken close to the break frequency than at the other frequencies as the gain remained nearly constant at those other frequencies. The resulting gain data was plotted with GNU Octave [13] and the resulting frequency response plot is presented in Figure 3.2. From the plot, one can see that the break frequency is close to 2.1 MHz.

Table 3.2: Noise introduced from the amplifiers.

Component	Noise per $\sqrt{\text{Hz}}$	Noise below 2.1 MHz
Instrumentation amp.	3.2 nV	4.6 µV
Programmable gain amp.	5.0 nV	7.2 µV

Noise

Noise estimation Due to the complexity involved in measuring noise, no noise measurements have been conducted. However, by reading the noise specifications from the datasheets of the amplifiers, an estimation of the noise can be made. A list of the noise specifications for the two amplifiers used is presented in Table 3.2. From the table it can be seen that 4.6 µV of noise that will not be filtered out by the low pass filter is introduced from the instrumentation amplifier and 5.0 V from the PGA. However, the signal path to the PGA goes via the instrumentation amplifier so that the noise present at the output of the PGA is the sum of the two individual noise levels, that is 11.8 µV.

Bit resolution For the worst-case signal path, 11.8 µV of noise is introduced. This voltage divided by the maximum voltage sampled by the ADC, 5 V, yields a ratio that can be converted to the same binary format that the ADC uses; Q0.24. That ratio equals $2.36 \cdot 10^{-6}$ and when converted to Q0.24 it can be written as 0.0000 0000 0000 0000 0010 1000_2, with the subscript, 2, denoting the base of the number format. Since there are 18 '0's preceding the first '1' in that number, the noise will not affect the first 18 bits. Hence the ADC will provide output samples with 18 bits of resolution.

Power consumption

The current consumption for the ADC front end excluding the FPGA development board was found to be 18 mA at the 12 V supply when outputting samples at a rate of 2.5 MHz. This yields a power consumption of 220 mW.

Distortion at high amplitudes

The ADC system suffered problems when subject to high amplitudes on the input signal resulting in a distorted output signal. No explanation to this behavior has been found. For low amplitudes (less than about 200 mV), the system seems to behave as expected. Addressing this issue is considered future work.

3.2 FPGA interface

In this section the internal and external interfaces of the FPGA are described.

3.2.1 ADC communication

The FPGA interface to the ADC consists of a control bus and a data bus which sends samples at a rate of 2.5 MHz, low pass filtered with a break frequency of 1 MHz. Because of this high speed, reading from the ADC was implemented

on the FPGA fabric and not in the embedded processor. The writes to the ADC are controlled from the processor. A VHDL module, `ad7760Reader` (see Listing B.1 for source code), was created to forward processor writes to and reading samples from the ADC. Read samples are stored in memory and the processor can fetch them from there.

3.2.2 Processor - FPGA communication

Communications between processor and FPGA fabric was done through a AMBA AXI4 Interface Protocol [18]. This protocol is capable of burst reads, which means that it can read large chunks of memory in a single read operation, yielding high throughput. On the FPGA fabric side, this interface is implemented in a file automatically generated by Xilinx XPS, which is part of ISE Design Suite [16]. This file was modified to contain modules for implementing the ADC communication and the FIR filter bank described in Section 3.3. On the processor side, functions to communicate with the FPGA side was generated by Xilinx XPS.

3.3 FPGA FIR filter bank

An FIR filter bank demonstration application was implemented on the ADC front end (for source code see Listing B.2, Listing B.3 and Listing B.4). The implementation follows the design described in Section 2.3. Implementations with various parameters were tested for speed and usage of FPGA resources. These parameters were:

- *Word size* - The number of bits used to represent integers in the design.
- *Filter order* - The order of the FIR filters. All filters in the bank were always of the same order because of the simplicity of running the tests that way and no reason was found to why mixing filter orders would yield more interesting test results or even different test results.
- *Number of filters* - The number of FIR filters in the bank.

3.3.1 Computational speed

The FIR filter bank was able to operate at a rate of 77 MHz, that is outputting samples at that rate. This is true for all the variations of parameters.

3.3.2 Resource utilization

For the resource utilization diagrams presented in this section (Figure 3.3, Figure 3.4 and Figure 3.5), the two most interesting resources and at the same time the resources whose degree of utilization vary the most are LUT and DSP48E1. LUT are the lookup tables constituting the central part of the FPGA fabric and DSP48E1 are hardware digital signal processing blocks like multipliers and adders.

For too large values of the implementation parameters, the tools synthesizing the FPGA bitstream stopped with an error message stating that to little RAM was available on the computer that was running the tools. This thus limited the

experiments. This computer was running a 32-bit version of Microsoft Windows which limits the memory per application to 2 GB. Had the synthesis been run on 64-bit Windows, this would likely not have been an issue and parameters with larger values could have been tested. This error was seen for:

- When the word size exceeded 16 bits.
- When the filter order exceeded 256.
- When the number of filters exceeded 32.

Word size

Two FIR filter bank implementations with different word sizes were generated and the resulting resource utilization is presented in Figure 3.3. It can be seen that more DSP48E1 blocks are used in the case of a word size of 8 while more LUT blocks were used in the case of a word size of 16. This makes it hard to estimate the resource utilization for varying word sizes, as there is no known correlation between DSP48E1 blocks and LUT blocks.

Filter order

Three FIR filter bank implementations with different orders of the filters were generated and the resulting resource utilization is presented in Figure 3.4. It can be seen that varying the filter order does not affect the resource utilization. Why this is the case is hard to answer, but one possible explanation is that some of the 25 % DSP48E1 blocks are consumed as a continuous block, that is that no value of utilization in between some lower bound and 25 % is possible.

Number of filters

Three FIR filter bank implementations with different numbers of filters per bank were generated and the resulting resource utilization is presented in Figure 3.5. It can be seen that for low amounts of filters per bank many DSP48E1 blocks are utilized while LUT blocks are utilized instead for higher amounts of filters per bank. As in the case for varying word size, this makes it hard to estimate resource utilization.

3.3.3 Power consumption

The power consumption of the FPGA was measured when the FPGA was idle and when FIR filter banks with different parameters were running on it. The processor was stopped and the ADC front end board was disconnected during both measurements. No notable difference in power consumption was found in any of the parameter variations compared to when the FPGA was idle; The measurements always showed a power consumption of 370 mW. This is quite high standby consumption, and is probably a result of the fact that the measurements were taken on a development board with lots of peripherals whose power consumption is unknown. If the measurements could have been taken on the FPGA component only, a more accurate estimation of the power consumption of the FIR filter bank could have been done.

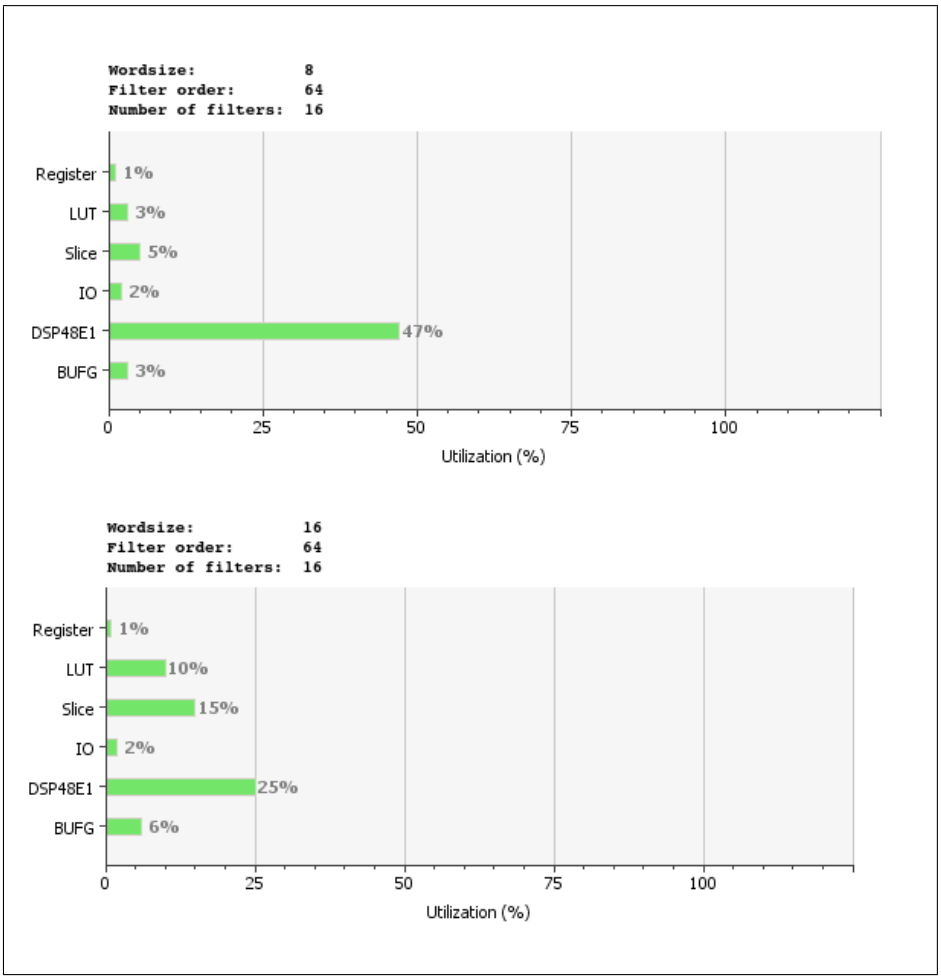


Figure 3.3: Resource utilization diagrams for an FIR filter bank for varying word size

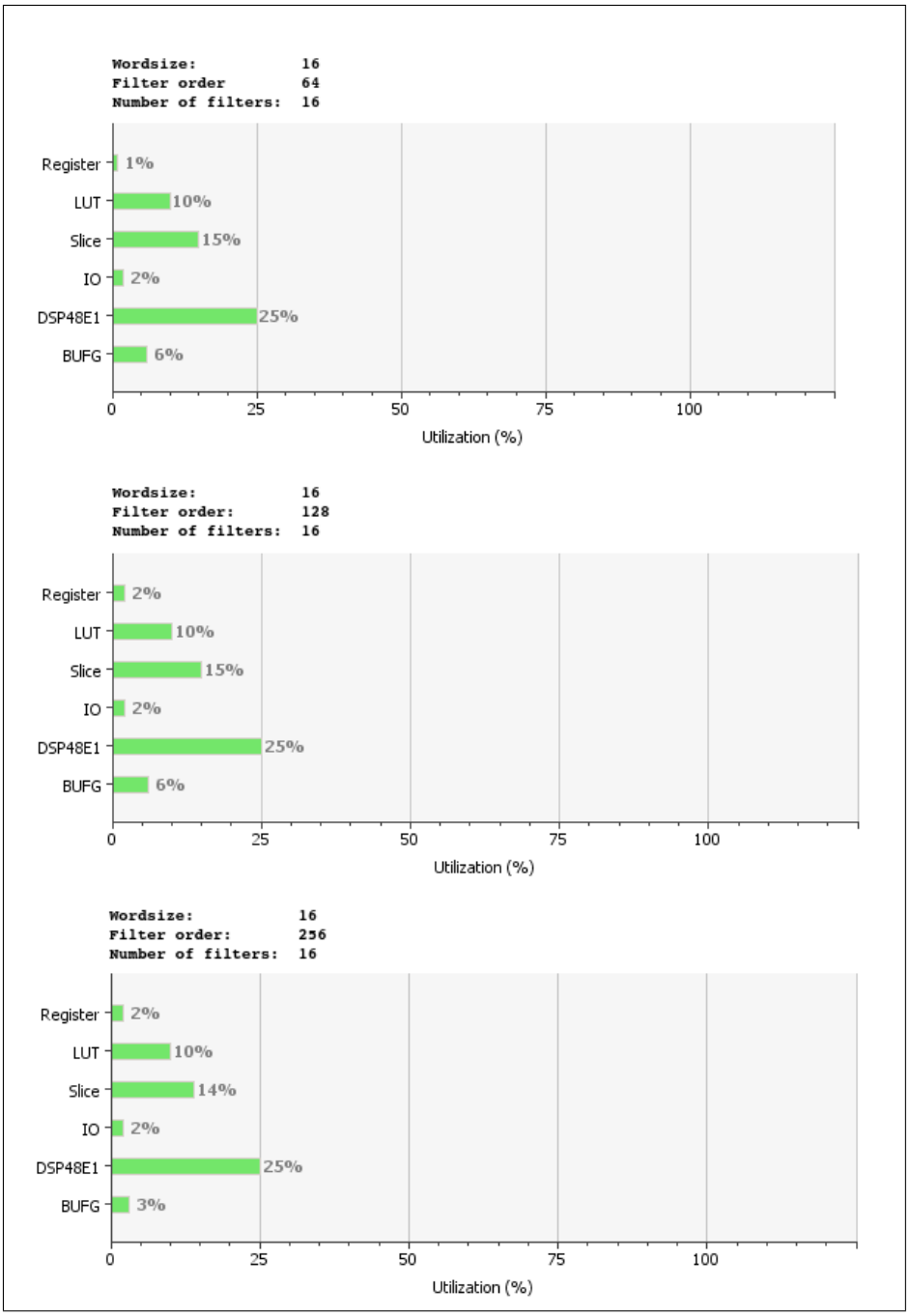


Figure 3.4: Resource utilization diagrams for an FIR filter bank for varying filter order.

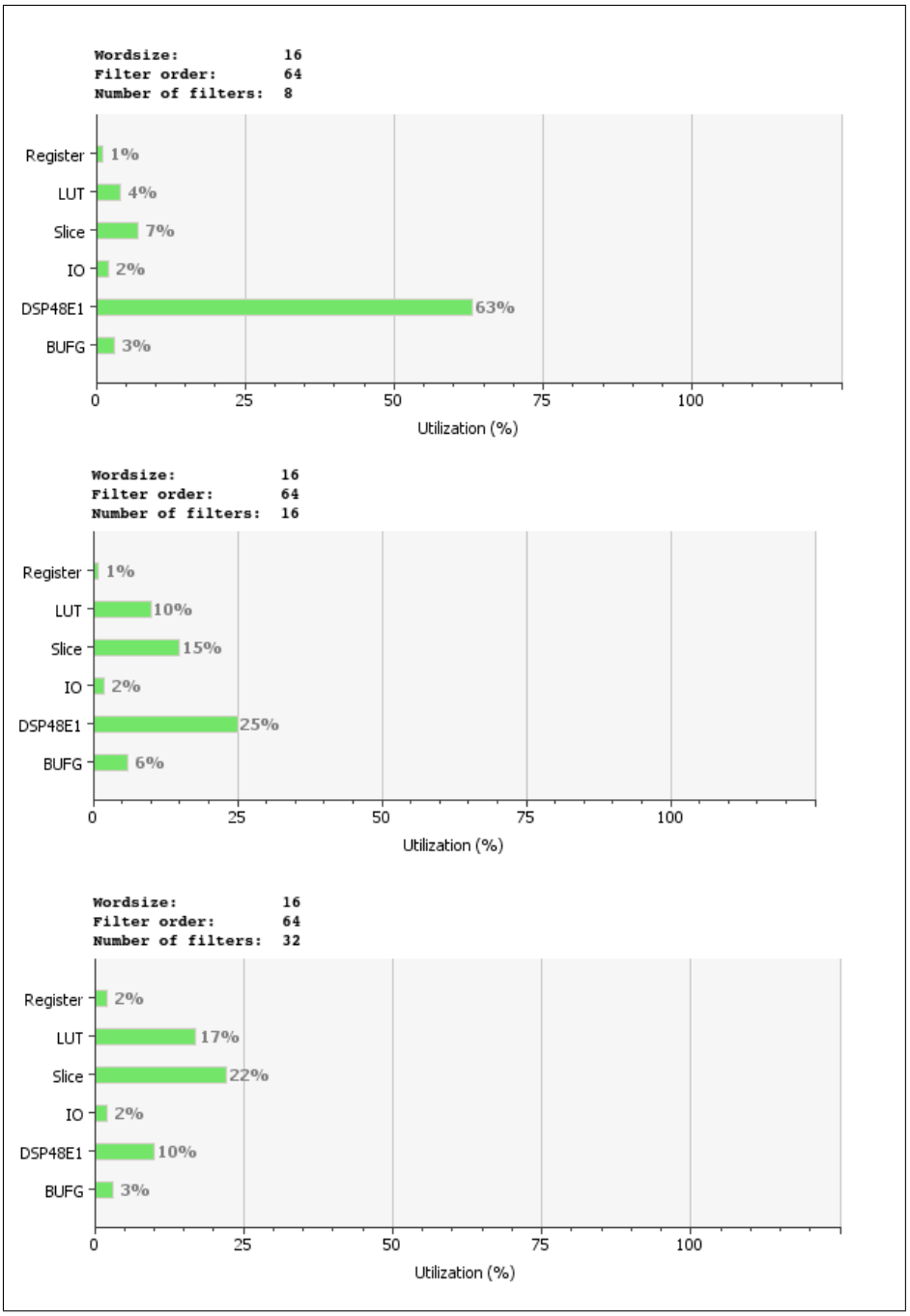


Figure 3.5: Resource utilization diagrams for an FIR filter bank for varying numbers of filters per bank.

3.4 FPGA Matching pursuit

In a parallel project made by Albertsson [5] a VHDL implementation of the event-based matching pursuit described by Smith and Lewicki in 2006 [4] was developed. In that project kernels were modeled in the same way as FIR filters were modeled in this project. An attempt was made to use the code developed in that project on the FPGA from this thesis, but it was found that the code was not synthesizable by ISE Design Suite [16]. The code could probably have been modified to be synthesizable, but due to time constraints of both Albertsson and the author, this was not done.

Chapter 4

Discussion

In this chapter, an evaluation of the different sub-projects of this thesis is presented. Conclusions are drawn from those evaluations and a discussion is held of what can be done with the result. Possible future work is also presented.

4.1 ADC front end evaluation

An evaluation of the ADC front end was performed, based on measured data and calculated estimations. The discussion that follows is the result of that evaluation.

4.1.1 Power consumption

The 220 mW of power that the ADC front end consumes when operating at full speed is quite high. A standard lithium ion battery may have stored energy on the order of 10 kJ. Assuming a 10 kJ battery, then the on-time of the front end would equal 10 kJ / 220 mW $\approx$ 13 hours and here the power consumed by the FPGA is excluded. Due to such a short battery life, continuous sampling may be inappropriate for battery powered applications. However, if sampling is done in intervals, say every minute of an hour, then change of batteries need only to happen once a month. For certain applications, for example condition monitoring of bearings described in Section 1.2.1, this approach can be used to reduce the power consumption as a bearing often breaks slowly and therefore small defects need not to be detected immediately. While this is the case for bearings, for other applications like the earthquake detection example described in Section 1.2.2 the output of a sufficiently large solar cell might be enough to keep the sensor system continuously powered.

4.1.2 Noise

If the noise estimations are correct then a quite high signal-to-noise ratio (SNR) can be achieved even for very small signals. For example, for a signal of as small amplitude as 1 mV, a signal to noise ratio for the worst-case path (via both the instrumentation amplifier and the PGA) the SNR would equal 1 mV / 11.8 μV = 85 = 58 dB.

Also, 18 noise free bits from the ADC is a quite good result and might very well be applicable to running an event-based matching pursuit algorithm like the one described by Smith and Lewicki in 2006 [4]. This has actually been demonstrated by Albertsson [5] who has run matching pursuit with good results with 16-bit fixed-point integers as primary data type.

4.2 FIR filter bank evaluation

The FIR filter bank implementation was evaluated and some conclusions were drawn from that evaluation. The discussion that follows is a result of that evaluation.

4.2.1 Resource utilization

It is hard to draw conclusions from the resource usage results presented in Section 3.3.2 as DSP48E1 and LUT blocks seem to be used interchangeably and no correlation between their value is known. It would be interesting to see diagrams for implementations where no DSP48E1 blocks were available, but no such option in ISE Design Suite [16] has been found.

However, the resource usage seem to be low even for the largest filter bank that could be generated, that is the one with 16 filters in which each filter is of order 256 and has a word size of 16 bits. Because of the large size of that filter, the author deems it likely that large applications like the VHDL matching pursuit code developed by Albertsson [5] could fit in the FPGA used in this thesis.

4.2.2 Computational speed

The FIR filter bank could be run at a sample rate of 77 MHz. In the parallel project by Albertsson [5], frequencies on the order of 10 MHz is required to run the VHDL event-based matching pursuit algorithm developed in that project. Due to the similarity in design between the two projects (see Section 2.3.1 for more information on this), the author deems it likely that event-based matching pursuit (as described in [4]) could be implemented on the system constructed in this thesis.

4.2.3 Power consumption

As the power consumption of the FIR filter bank is negligible compared to the power consumption of the FPGA development board in stand-by mode, it is hard to estimate the energy efficiency of the FIR filter bank. However, it would be interesting to see how much the power consumption can be increased by converting to an ASIC design. For a deeper discussion on this topic, see Section 4.3.2.

4.3 Future work

In this section possible future work is presented that was not included in this thesis due to time and scope constraints.

4.3.1 Low power consumption design

As mentioned in Section 4.2.3, it would be interesting to see how much the power consumption can be lowered by changing the designs of the applications. To do this, a method for calculating the power consumption of the actual application running on the FPGA (power consumption excluding stand-by consumption of the development board) needs to be established.

4.3.2 FPGA to ASIC

FPGAs demand a higher per-unit-cost for large quantities of devices than do ASICs. Also, FPGAs are inherently less power and speed efficient than ASICs. Not counting the use of "hard blocks" in FPGAs (that is, for example, a multiplier made directly in hardware which can be connected to the FPGA fabric), the power consumption can be lowered by a factor of approximately 17 by converting from an FPGA design to an ASIC design [19]. Still not counting hard blocks, the computational speed can be increased by a factor of approximately 35 by converting to an ASIC [19]. These approximate improvement factors are based off certain benchmarking tests run in [19]. The FPGA that was used in this thesis has 25 hard multipliers. That is far fewer than the multipliers used in all the FIR filters in the demos presented in this thesis. This means that merely by changing to an ASIC design (not modifying the designs), the power consumption and computational speed of the ADC front end (excluding the analog parts) could probably be reduced by factors of around 17 and 35, respectively. Going from an FPGA to an ASIC design should be relatively easy when assuming the VHDL and C code developed in this project.

4.3.3 Implementing Albertsson's matching pursuit

The C and VHDL implementation of matching pursuit developed by Albertsson [5] can be adjusted as to be synthesizable on the ADC front end constructed in this thesis. If this is done successfully, it would prove that event-based matching pursuit described by Smith and Lewicki in 2006 [4] is implementable on the ADC front end developed in this thesis. It would also make the ADC front end ready to be used for most, if not all, the applications described in Section 1.2.

4.4 Conclusion

Based on all the measurements, estimations and calculations done in this thesis, it is deemed likely that the ADC front end is capable of processing algorithms with similar demands as event-based matching pursuit described by Smith and Lewicki [4] in a sampling rate on the order of 1 MHz. It is also concluded that a power consumption suitable for some applications mentioned in this thesis (see Section 1.2) may be achieved. In any case, the power consumption of the front end constructed in this thesis can most likely be improved by altering its design and/or changing its components. This would also extend the ADC front end's usability and make it suitable for even more applications. The front end that was built in this thesis demonstrates that if the noise estimations done in this thesis are correct, then a system can be built which samples in the megahertz range with a precision of at least 18 noise free bits.

Chapter 5

Glossary

Below follows a table with meanings and explanations for the abbreviations used in this thesis.

Abbrev.	Meaning	Explanation
ADC	*Analog-to-Digital Converter*	Component that converts an analog signal to a digital one
AFC	*Analog-to-Feature Converter*	Component that converts an analog signal to a digital one which describes the original signal in terms of features
ASIC	*Application-Specific Integrated Circuit*	Very small electronic circuit performing a specific task
C	*C*	A programming language
DSP	*Digital Signal Processor*	Processor more suitable for signal processing than an ordinary microprocessor
EDA	*Electronic Design Automation*	The field of automating the creation of electronic systems, for example PCBs
FIR	*Finite Impulse Response*	Digital filter with the property that it eventually produces a zero output when a single impulse is input to it
FPGA	*Field-Programmable Gate Array*	Programmable logic circuitry
GNU	*GNU's Not UNIX*	A free operating system
HDL	*Hardware Description Language*	Language describing hardware
LUT	*Look-Up Table*	Table made of digital electronics used to realize logic functions

MUX	*MUltipleXer*	Electronic component that selects as its output signal one out of many input signals
PCB	*Printed Circuit Board*	Board containing electronic circuits
PGA	*Programmable Gain Amplifier*	Electrical amplifier with variable gain
PWM	*Pulse Width Modulation*	Method for creating digital control signals with known period but variable pulse width
RAM	*Random-Access Memory*	Volatile memory with short access time
SKF	*Svenska KullagerFabriken AB*	An international bearing-oriented company originating from Sweden [6]
SNR	*Signal-to-Noise ratio*	The ratio between signal amplitude and noise amplitude
UART	*Universal Asynchronous Receiver/Transmitter*	Commonly used simple communication protocol
VHDL	*VHSIC Hardware Description Language*	A hardware description language
VHSIC	*Very High Speed Integrated Circuit*	Integrated circuit designed for very high clock frequencies

Chapter 6

References

[1] Hull, Jamieson, and Balakrishnan. Mitigating congestion in wireless sensor networks. *SenSys '04 Proceedings of the 2nd international conference on Embedded networked sensor systems*, 2004.

[2] Stéphane Mallat and Zhifeng Zhang. Matching pursuit with time-frequency dictionaries. *IEEE Transactions on Signal Processing*, 1993.

[3] Bruno A. Olshausen and David J. Field. Sparse coding of sensory inputs. *Current Opinion in Neurobiology*, 2004.

[4] Evan C. Smith and Michael S. Lewicki. Efficient auditory coding. *Nature*, 2006.

[5] Kim Albertsson. Towards self-learning sensors: Matching pursuit with dictionary learning on FPGA. Master's thesis, Luleå University of Technology, to appear 2013.

[6] SKF. URL `http://www.skf.com/`. Visited on May 24, 2013.

[7] A W Lees, Z Quiney, A Ganji, and B. Murray. The use of acoustic emission for bearing condition monitoring. *Journal of Physics: Conference Series*, 2011.

[8] Sergio Martin del Campo Barraza, Kim Albertsson, Joakim Nilsson, Jens Eliasson, and Fredrik Sandin. FPGA prototype of machine learning analog-to-feature converter for event-based succinct representation of signals. *IEEE International Workshop on Machine Learning for Signal Processing*, 2013, accepted for publication.

[9] N Liu, S.F Ling, and R Gribonval. Bearing failure detection using matching pursuit. *NDT & E International 35*, 2002.

[10] Kuansan Wang and David M. Goblirsch. Extracting dynamic features using the stochastic matching pursuit algorithm for speech event detection. *IEEE Workshop on Automatic Speech Recognition and Understanding*, 1997.

[11] Free Software Foundation. GNU licenses. URL `http://www.gnu.org/licenses/`. Visited on August 23, 2013.

[12] Adel S. Sedra and Kenneth C. Smith. *Microelectronic circuits*, chapter 12. Oxford University Press Inc., fifth edition, 2004. ISBN 0-19-514252-7.

[13] Free Software Foundation. GNU Octave. URL `http://www.gnu.org/software/octave/`. Visited on June 16, 2013.

[14] MathWorks. MATLAB. URL `http://www.mathworks.se/products/matlab/`. Visited on May 22, 2013.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*, chapter 13. MIT press, third edition, 2010. ISBN 978-0-262-03384-8.

[16] Xilinx. ISE design suite. URL `http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm`. Visited on May 22, 2013.

[17] Cogra. URL `http://www.cogra.se/`. Visited on May 24, 2013.

[18] Xilinx. AMBA AXI4 Interface Protocol. URL `http://www.xilinx.com/ipcenter/axi4.htm`. Visited on May 23, 2013.

[19] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2006.

Appendix A

ADC front end specification

A.1 Component list

A list of all non pin header connectors and active components that were used is presented in Table A.1.

A.2 Schematics

All schematics of the ADC front end are presented in the following figures:

- Amplification system: Figure A.1
- ADC system: Figure A.2
- FPGA interface: Figure A.3
- Power supply: Figure A.4

A.3 Board design

The dimensions of the board are 57×65 mm.

The board masks of the ADC front end are presented in the following figures:

- Front layer: Figure A.5
- Back layer: Figure A.6
- Inner layer 1: Figure A.7
- Inner layer 2: Figure A.8

A.4 Pictures

Pictures of the ADC front end board is preseneted in the following Figures:

- Front: Figure A.9
- Back, beside FPGA development board: Figure A.10
- Mounted on FPGA development board: Figure A.11

Table A.1: List of all non pin header connectors and active components and that were used to design the ADC front end

Part	Part name	Amount
Amplifiers		
Instrumentation amplifier	AD8421ARMZ	1
Programmable gain amplifier	AD8330ARQZ	1
Voltage generators		
Voltage reference, 2.5 V	CD4053BPWRG3	1
Voltage regulator, 2.5 V	LT1763CS8-2.5#PBF	2
Voltage regulator, 5 V	LT1763CS8-5#PBF	2
Other		
Analog-to-digital converter	AD7760BSVZ	1
Multiplexer	CD4053BPWRG3	1
Level shifter	TXB0102DCUR	1
Expansion connector	ASP-134606-01	1
FPGA development board	ZedBoard Rev. C.1	1

A.5 Errata

Mistakes that were made during the making of the board are listed here:

- The pads of the voltage level shifter, `LevelShifter1`, shown in Figure A.3, are too short and should be extended. It is still possible to solder `LevelShifter1`, but it is difficult.
- The holes for the expansion connector, `U1`, shown in Figure A.3, are of too small diameter so that the expansion connector pins will not fit. If a steady hand is available, the expansion connector can be soldered on top of the holes.
- For unknown reasons, the PGA control signal, `pwm`, could not be programmed to output a signal. As a workaround, a patch cable between pins `H20` and `C6` of `U1` can be soldered to the board.

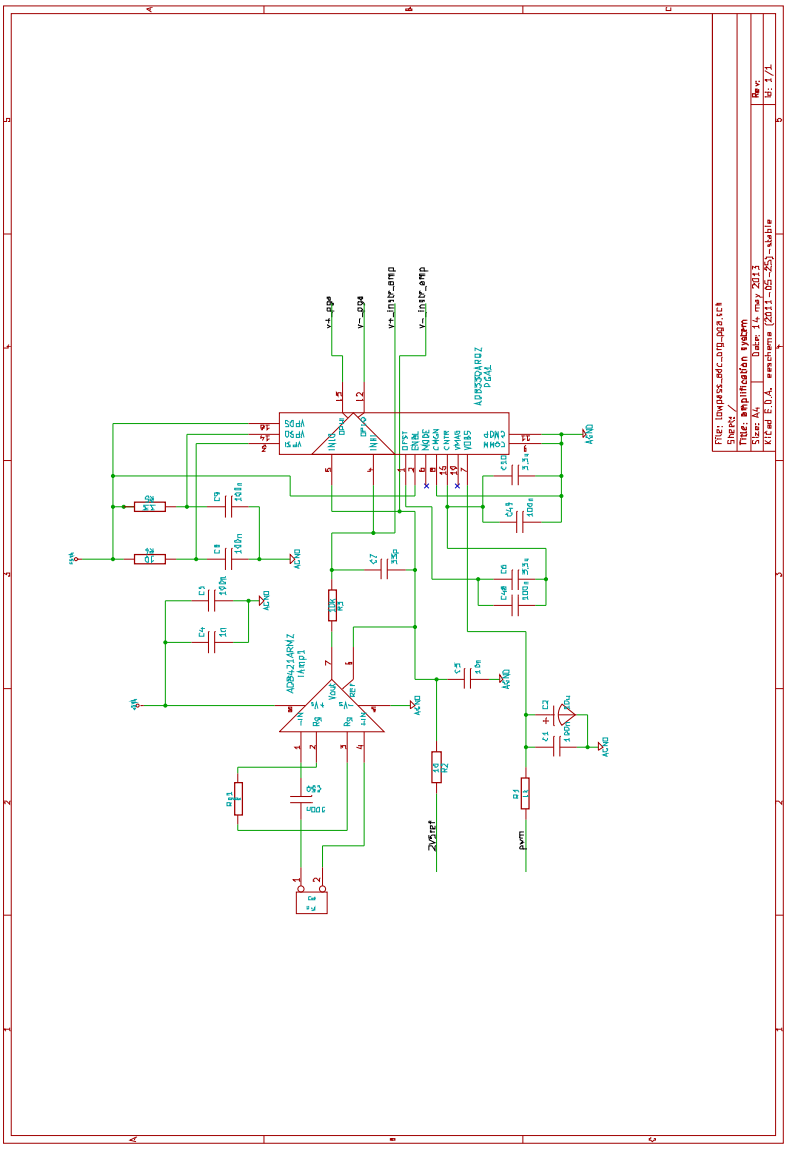


Figure A.1: Schematics of the amplifier system of the ADC front end.

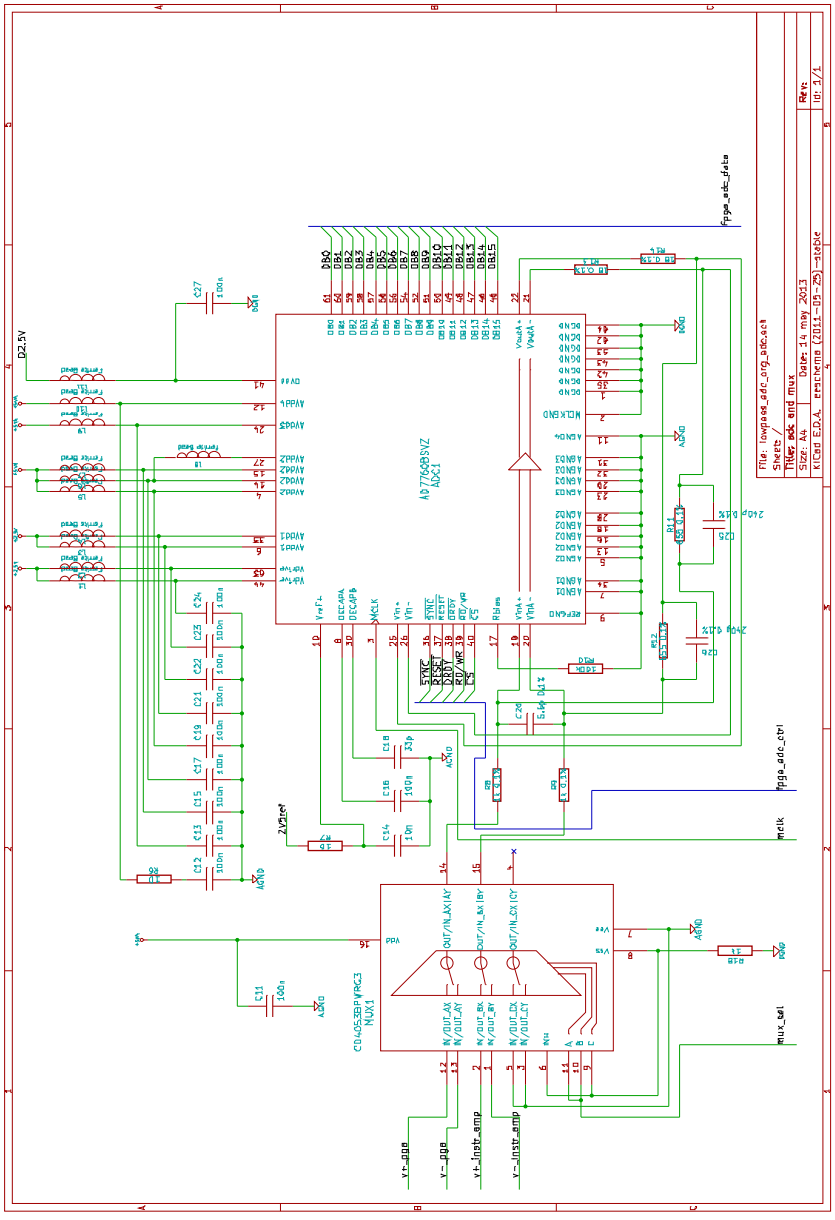


Figure A.2: Schematics of ADC system of the ADC front end.

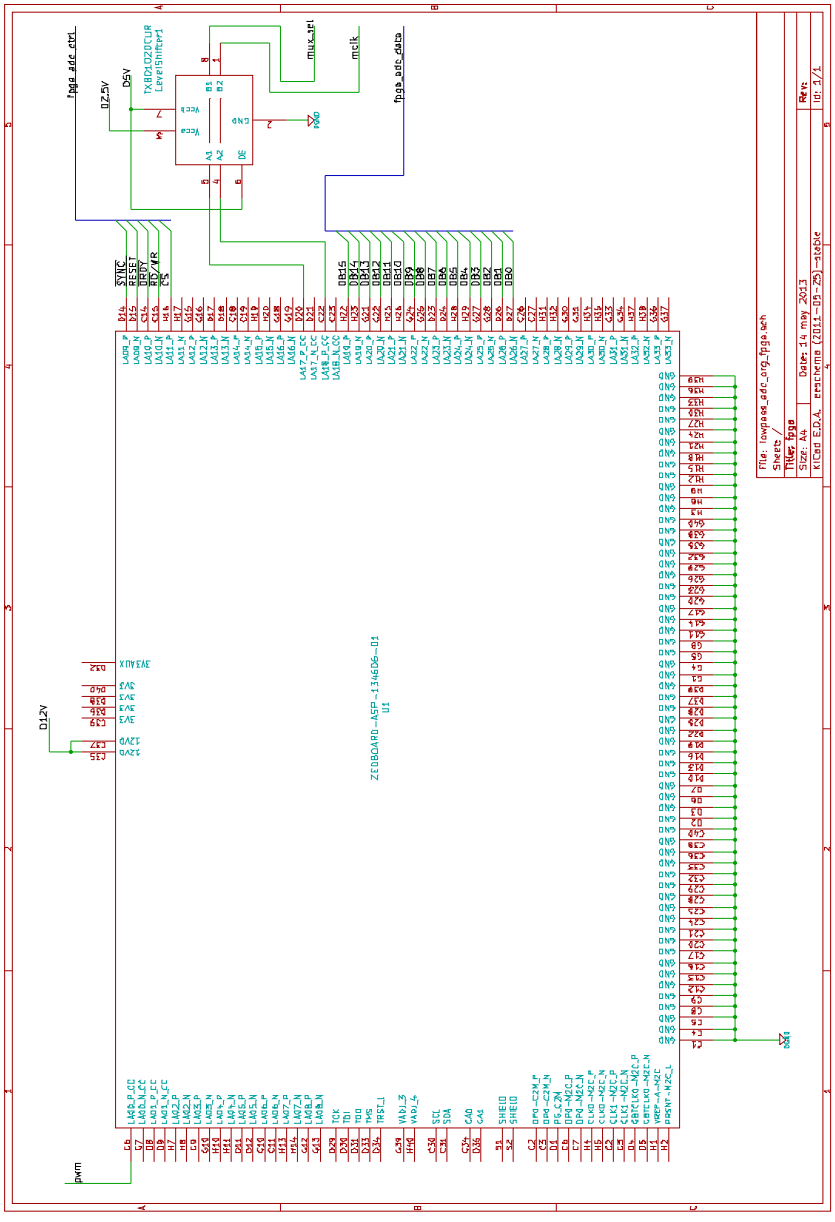


Figure A.3: Schematics of the FPGA interface of the ADC front end.

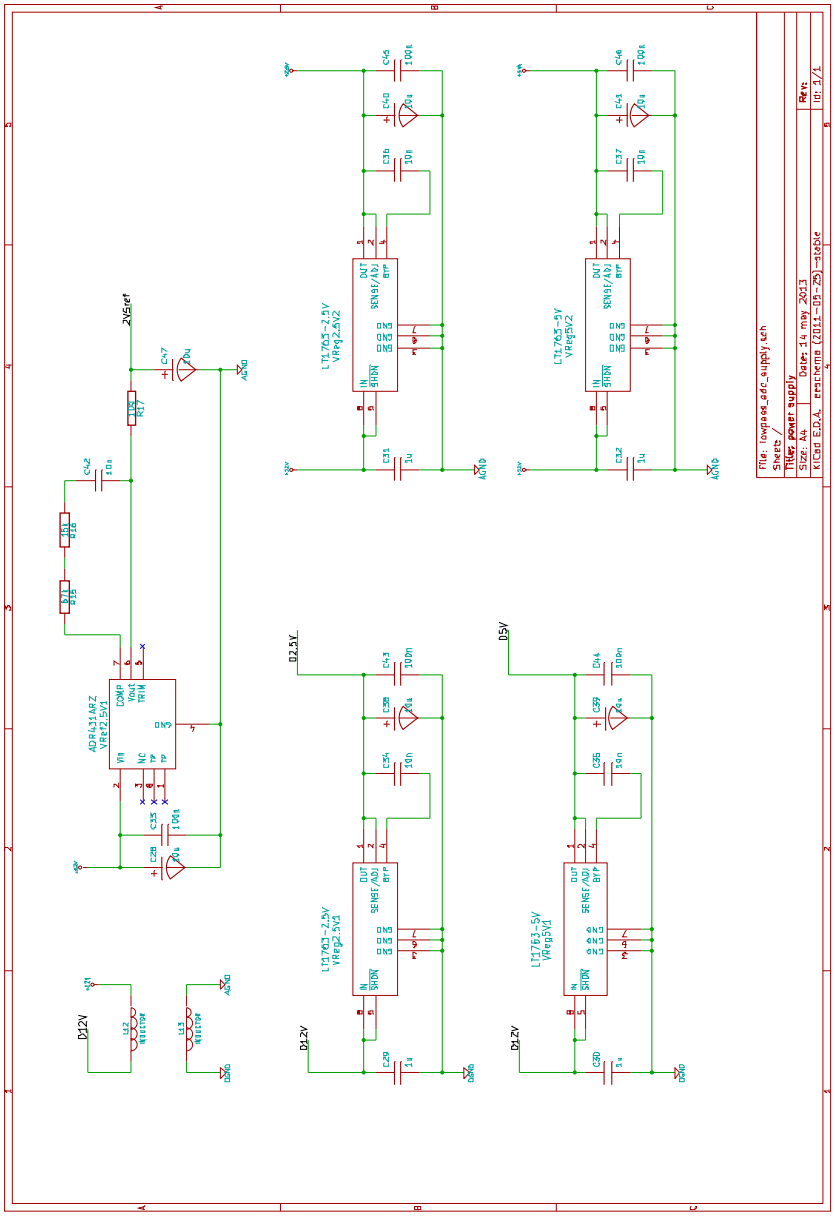


Figure A.4: Schematics of the power supplies of the ADC front end.

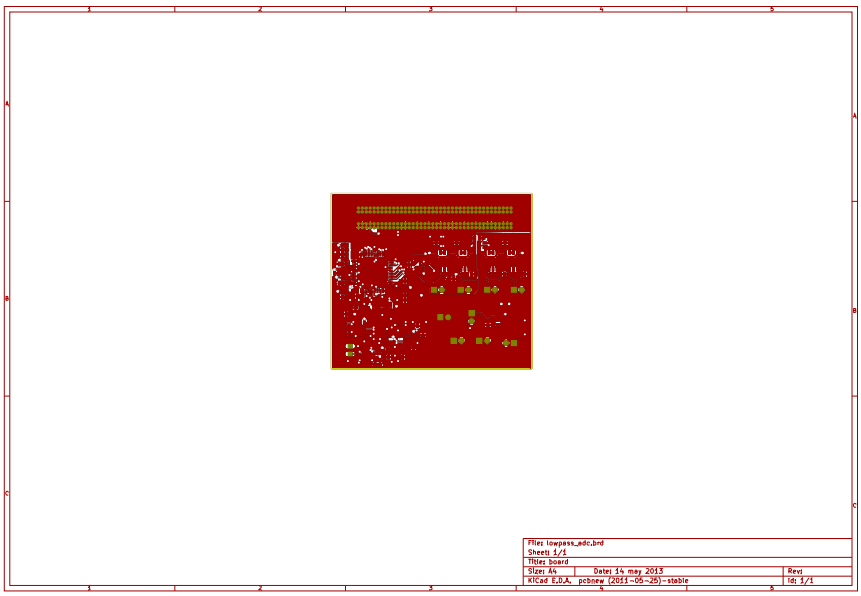


Figure A.5: Front layer of the mask of the ADC front end board.

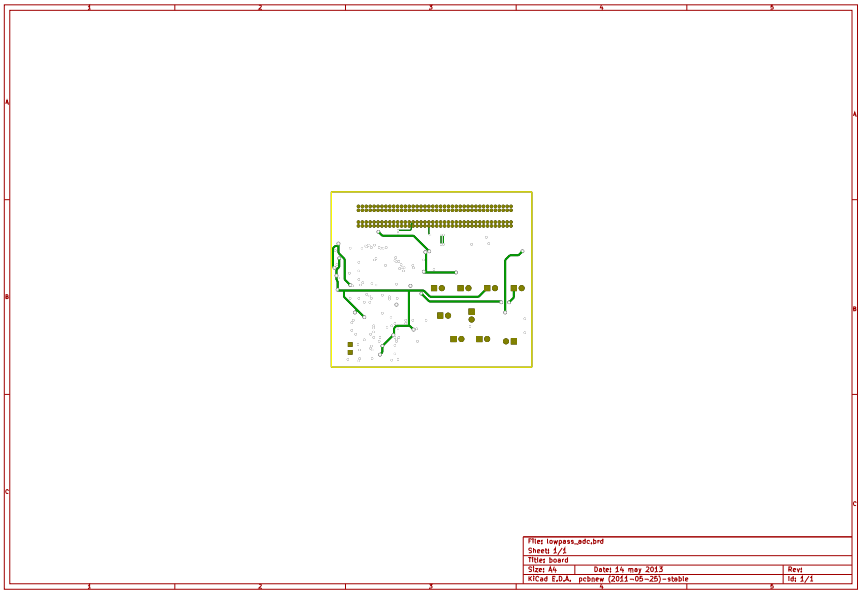


Figure A.6: Back layer of the mask of the ADC front end board.

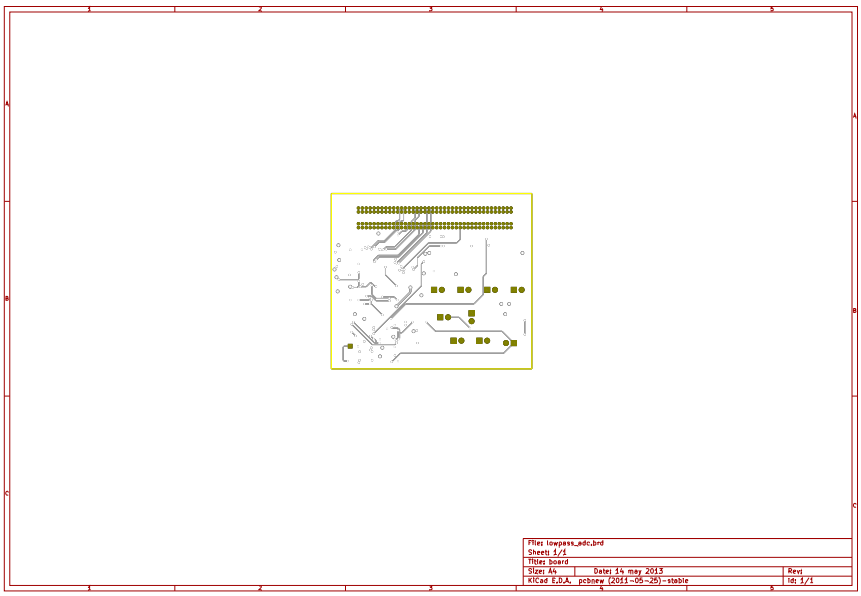


Figure A.7: First inner layer of the mask of the ADC front end board.

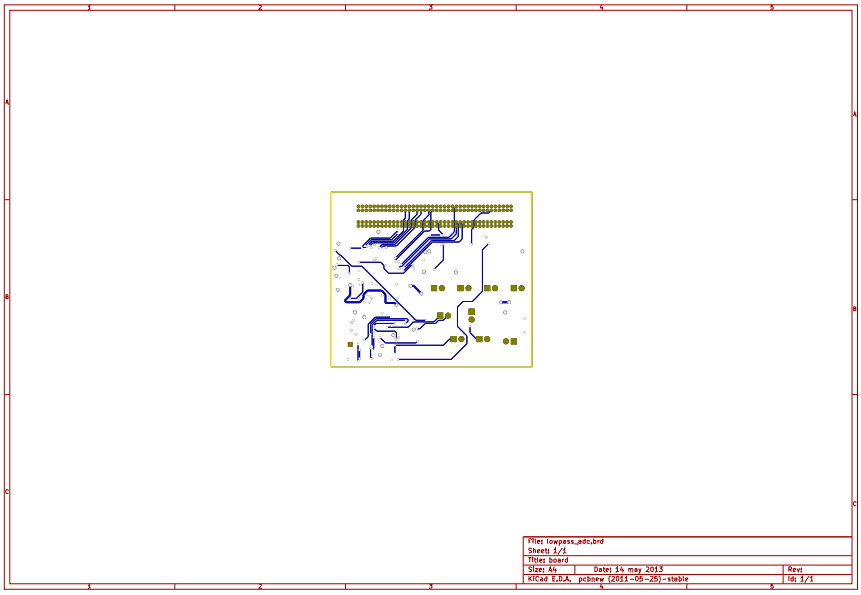


Figure A.8: Second inner layer of the mask of the ADC front end board.

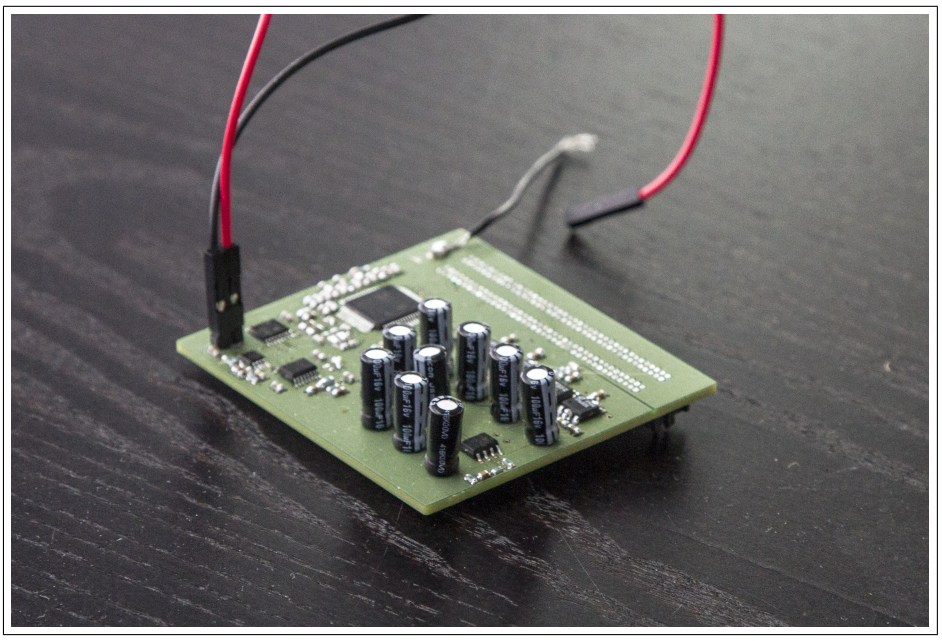


Figure A.9: The front side of the ADC front end board.

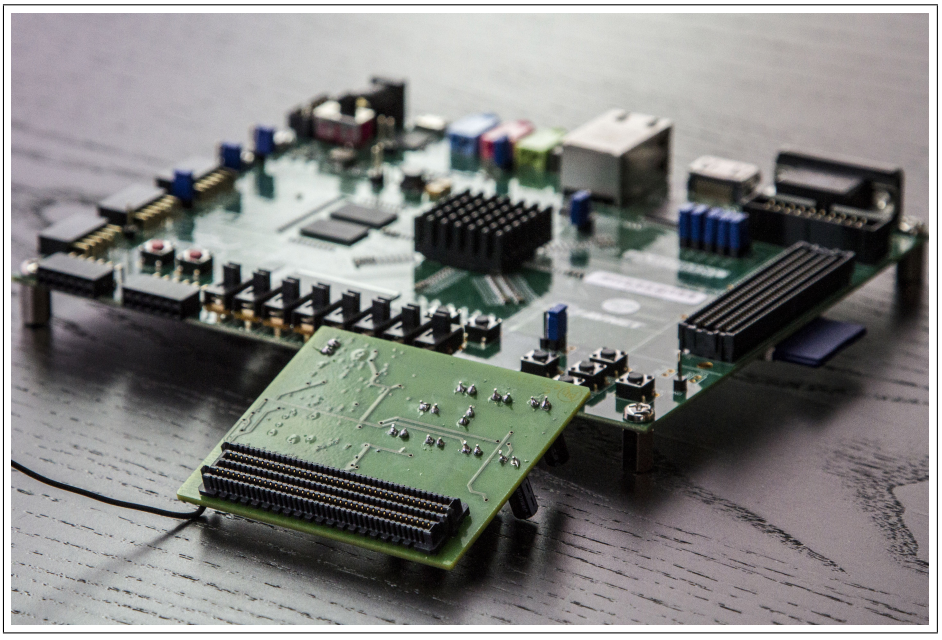


Figure A.10: The back side of the ADC front end board laying beside the FPGA development board.

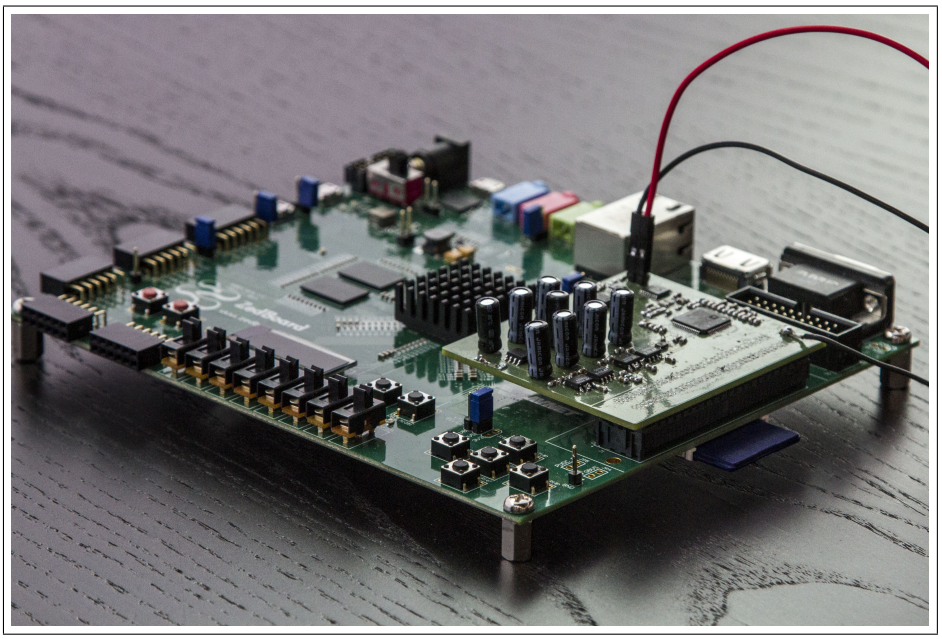


Figure A.11: The ADC front end board mounted on top of the FPGA development board.

Appendix B

Source code

B.1 VHDL source code

VHDL source code for selected files developed for this thesis is presented in the following listings:

- `ad7760Reader.vhd`: Listing B.1
- `FirFilterBank.vhd`: Listing B.2
- `FirFilter.vhd`: Listing B.3
- `AdderTree.vhd`: Listing B.4

Listing B.1: `ad7760Reader.vhd` - VHDL module describing a reader for an AD7760BSVZ ADC.

```
1  -- Copyright 2013 Joakim Nilsson
2  --
3  -- This program is free software: you can redistribute it and/or modify
4  -- it under the terms of the GNU General Public License as published by
5  -- the Free Software Foundation, either version 3 of the License, or
6  -- (at your option) any later version.
7  --
8  -- This program is distributed in the hope that it will be useful,
9  -- but WITHOUT ANY WARRANTY; without even the implied warranty of
10 -- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
11 -- GNU General Public License for more details.
12 --
13 -- You should have received a copy of the GNU General Public License
14 -- along with this program.  If not, see <http://www.gnu.org/licenses/>.
15
16 library ieee;
17 use ieee.std_logic_1164.all;
18 use ieee.numeric_std.all;
19
20 entity ad7760Reader is
21     port (
22         clk          : in std_logic;
23         rst          : in std_logic;
24         read_en      : in std_logic;
25         drdy         : in std_logic;
26         cs_in        : in std_logic;
27         cs_out       : out std_logic;
28         db           : in std_logic_vector(15 downto 0);
29         res          : out std_logic_vector(31 downto 0);
30         resrdy       : out std_logic
31     );
32 end ad7760Reader;
33
34 architecture arch of ad7760Reader is
35     -- Types
36     type state_t is (
37         IDLE,
38         CSLOW1,
39         READ1,
40         CSHIGH,
41         CSLOW2,
42         READ2,
43         RESR
44     );
45
46     -- Signals
47     signal state_cur     : state_t;
48     signal state_next    : state_t;
49     signal msbdata       : std_logic_vector(15 downto 0);
50
51     begin
52
53         -- Update state and reset process
54         process (clk, rst) begin
55             if rst = '1' then
56                 state_cur <= IDLE;
57                 elsif rising_edge(clk) then
58                 state_cur <= state_next;
59             end if;
60         end process;
61
62         -- State machine process
63         process (state_cur, drdy, cs_in, read_en) begin
64             if read_en = '0' then
65                 cs_out <= cs_in;
66             else
67                 case state_cur is
68                     when IDLE =>
69                         if drdy = '0' then
70                             state_next <= CSLOW1;
71                         else
72                             state_next <= IDLE;
73                         end if;
74                         cs_out <= '1';
75                         resrdy <= '0';
76                         res <= (others => '0');
77
78                     when CSLOW1 =>
79                         state_next <= READ1;
80                         cs_out <= '0';
81
82                     when READ1 =>
83                         state_next <= CSHIGH;
84                         msbdata <= db;
85
86                     when CSHIGH =>
87                         state_next <= CSLOW2;
88                         cs_out <= '1';
89
90                     when CSLOW2 =>
91                         state_next <= READ2;
92                         cs_out <= '0';
93
94                     when READ2 =>
95                         state_next <= RESR;
```

```
96                              res <= msbdata & db;
97
98                          when RESR =>
99                              state_next <= IDLE;
100                             resrdy <= '1';
101                             cs_out <= '1';
102
103                         when others =>
104                             state_next <= IDLE;
105                 end case;
106             end if;
107         end process;
108
109 end arch;
```

Listing B.2: `FirFilterBank.vhd` - VHDL module describing a bank of FIR filters.

```
1  -- Copyright 2013 Joakim Nilsson
2  --
3  -- This program is free software: you can redistribute it and/or modify
4  -- it under the terms of the GNU General Public License as published by
5  -- the Free Software Foundation, either version 3 of the License, or
6  -- (at your option) any later version.
7  --
8  -- This program is distributed in the hope that it will be useful,
9  -- but WITHOUT ANY WARRANTY; without even the implied warranty of
10 -- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
11 -- GNU General Public License for more details.
12 --
13 -- You should have received a copy of the GNU General Public License
14 -- along with this program.  If not, see <http://www.gnu.org/licenses/>.
15
16 library ieee;
17 use ieee.std_logic_1164.all;
18 use ieee.numeric_std.all;
19
20 entity FirFilterBank is
21     generic (
22         WORDSIZE : integer;
23         ORDER : integer;
24         BANKS : integer
25     );
26     port (
27         clk, halt    : in std_logic;
28         rst          : in std_logic_vector(BANKS - 1 downto 0);
29         coeff_wen    : in std_logic_vector(BANKS - 1 downto 0);
30         ival         : in std_logic_vector(WORDSIZE - 1 downto 0);
31         ovals_flat   : out std_logic_vector(WORDSIZE * BANKS - 1 downto 0)
32     );
33 end FirFilterBank;
34
35 architecture arch_FirFilterBank of FirFilterBank is
36     -- Types
37     subtype SWord is signed(WORDSIZE - 1 downto 0);
38     type Arr_banks is array(0 to ORDER - 1) of SWord;
39
40     -- Signals
41     signal ovals : Arr_banks;
42
43     -- Components
44     component FirFilter
45         generic (
46             WORDSIZE : integer;
47             ORDER : integer
48         );
49         port (
50             clk, halt, rst   : in std_logic;
51             coeff_wen        : in std_logic;
52             ival             : in signed(WORDSIZE - 1 downto 0);
53             oval             : out signed(WORDSIZE - 1 downto 0)
54         );
55     end component;
56
57     begin
58
59         firs: for i in 0 to BANKS - 1 generate
60             -- FIR filters
61             fir: FirFilter
62             generic map (
63                 WORDSIZE => WORDSIZE,
64                 ORDER => ORDER
65             )
66             port map (
67                 clk => clk,
68                 halt => halt,
69                 rst => rst(i),
70                 coeff_wen => coeff_wen(i),
71                 ival => signed(ival),
72                 oval => ovals(i)
73             );
74
75             -- Unpack output values
76             ovals_flat( WORDSIZE * (i + 1) - 1  downto WORDSIZE * i) <=
                   std_logic_vector( ovals(i) );
77         end generate;
78
79 end arch_FirFilterBank;
```

Listing B.3: `FirFilter.vhd` - VHDL module describing an FIR filter.

```
1  -- Copyright 2013 Joakim Nilsson
2  --
3  -- This program is free software: you can redistribute it and/or modify
4  -- it under the terms of the GNU General Public License as published by
5  -- the Free Software Foundation, either version 3 of the License, or
6  -- (at your option) any later version.
7  --
8  -- This program is distributed in the hope that it will be useful,
9  -- but WITHOUT ANY WARRANTY; without even the implied warranty of
10 -- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
11 -- GNU General Public License for more details.
12 --
13 -- You should have received a copy of the GNU General Public License
14 -- along with this program.  If not, see <http://www.gnu.org/licenses/>.
15
16 library ieee;
17 use ieee.std_logic_1164.all;
18 use ieee.numeric_std.all;
19 use ieee.math_real.all;
20
21 entity FirFilter is
22     generic (
23         WORDSIZE : integer;
24         ORDER : integer
25     );
26     port (
27         clk, halt, rst   : in std_logic;
28         coeff_wen        : in std_logic;
29         ival             : in signed(WORDSIZE - 1 downto 0);
30         oval             : out signed(WORDSIZE - 1 downto 0)
31     );
32 end FirFilter;
33
34 architecture arch_FirFilter of FirFilter is
35     -- Constants
36     constant LOG_WORDSIZE : integer := integer( ceil( log2( real( WORDSIZE ) ) ) );
37
38     -- Types
39     subtype SWord is signed(WORDSIZE - 1 downto 0);
40     type Arr_order is array(0 to ORDER - 1) of SWord;
41
42     -- Signals
43     signal ptr              : unsigned(LOG_WORDSIZE - 1 downto 0) := (others => '0')
           ;
44     signal coeffs, vals     : Arr_order;
45     signal terms_flat       : std_logic_vector(WORDSIZE * ORDER - 1 downto 0);
46     signal termsExt_flat    : std_logic_vector(2 * WORDSIZE * ORDER - 1 downto 0);
47
48     -- Components
49     component AdderTree
50         generic (
51             WORDSIZE : integer;
52             ELEMENTS : integer
53         );
54         port (
55             clk, halt   : in std_logic;
56             terms_flat  : in std_logic_vector(WORDSIZE * ORDER - 1 downto 0);
57             sum         : out signed(WORDSIZE - 1 downto 0)
58         );
59     end component;
60
61     begin
62
63         -- Clock process
64         process (clk) is begin
65             if rising_edge(clk) then
66                 if rst = '1' then
67                     -- Reset pointer
68                     ptr <= (others => '0');
69
70                     -- Reset coefficients
71                     resetCoeffs: for i in 0 to ORDER - 1 loop
72                         coeffs(i) <= (others => '0');
73                     end loop;
74                 elsif (halt = '0') then
75                     -- Increment pointer
76                     ptr <= ptr + 1;
77
78                     -- Fetch input value
79                     vals( to_integer(ptr) ) <= ival;
80
81                     -- Write coefficient
82                     if coeff_wen = '1' then
83                         coeffs( to_integer(ptr) ) <= ival;
84                     end if;
85
86                 end if;
87             end if;
88         end process;
89
90         -- Multiply coeffs with vals
91         multiply: for i in 0 to ORDER - 1 generate begin
92             termsExt_flat( 2 * (i + 1) * WORDSIZE - 1 downto 2 * i * WORDSIZE ) <=
                   std_logic_vector( vals( to_integer(to_unsigned(i, ptr'length) + ptr
                   ) ) * coeffs(ORDER - 1 - i) );
```

```
93              terms_flat( (i + 1) * WORDSIZE - 1 downto i * WORDSIZE ) <=
                    termsExt_flat( 2 * (i + 1) * WORDSIZE - WORDSIZE - 1 downto 2 * i *
                     WORDSIZE );
94          end generate;
95
96          -- Add the terms
97          at: AdderTree
98          generic map (
99              WORDSIZE => WORDSIZE,
100             ELEMENTS => ORDER
101         )
102         port map (
103             clk => clk,
104             halt => halt,
105             terms_flat => terms_flat,
106             sum => oval
107         );
108
109 end arch_FirFilter;
```

Listing B.4: `AdderTree.vhd` - VHDL module describing an adder tree.

```
1  -- Copyright 2013 Joakim Nilsson
2  --
3  -- This program is free software: you can redistribute it and/or modify
4  -- it under the terms of the GNU General Public License as published by
5  -- the Free Software Foundation, either version 3 of the License, or
6  -- (at your option) any later version.
7  --
8  -- This program is distributed in the hope that it will be useful,
9  -- but WITHOUT ANY WARRANTY; without even the implied warranty of
10 -- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
11 -- GNU General Public License for more details.
12 --
13 -- You should have received a copy of the GNU General Public License
14 -- along with this program.  If not, see <http://www.gnu.org/licenses/>.
15
16 library ieee;
17 use ieee.std_logic_1164.all;
18 use ieee.numeric_std.all;
19
20 entity AdderTree is
21     generic (
22         WORDSIZE : integer;
23         ELEMENTS : integer
24     );
25     port (
26         clk, halt   : in std_logic;
27         terms_flat  : in std_logic_vector(WORDSIZE * ELEMENTS - 1 downto 0);
28         sum         : out signed(WORDSIZE - 1 downto 0)
29     );
30 end AdderTree;
31
32 architecture arch_AdderTree of AdderTree is
33     -- Types
34     subtype SWord is signed(WORDSIZE - 1 downto 0);
35     type Arr_2elementsM1 is array(0 to 2 * ELEMENTS - 2) of SWord;
36
37     -- Signals
38     signal heap : Arr_2elementsM1;
39
40     begin
41
42         -- Pack flat terms
43         packFlatTerms: for i in 0 to ELEMENTS - 1 generate begin
44             heap(ELEMENTS + i - 1) <= signed( terms_flat((i + 1) * WORDSIZE - 1
                    downto i * WORDSIZE) );
45         end generate;
46
47         -- Add terms tree-wise
48         addTerms: for i in 1 to ELEMENTS - 1 generate begin
49             heap(i - 1) <= heap(2 * i - 1) + heap(2 * i) when (rising_edge(clk) and
                    (halt = '0') );
50         end generate;
51
52         -- First element in heap is result
53         sum <= heap(0);
54
55 end arch_AdderTree;
```