Off-chip data acquisition system
for high density CMOS microelectrode arrays

Author:
Michelangelo Carrozzo

Professor:
Yusuf Leblebici

Project supervisors:
Alexandre Schmid
Neil Joye
Fabrizio Pirri

Lausanne, August 13, 2009
Acknowledgements

During my Master’s Thesis I’ve been supported by many individuals. Neil Joye provided all the informations on the MEA developed by LSM, guided me throughout the whole project, and revised my manuscript. Hossain Afshari dedicated me a lot of time to give me his vital support for the FPGA programming. Alexandre Schmid gave me many advices about the algorithm design and supported me in other administrative issues. Alain Vachoux guided me through the simulation using the Xilinx libraries on Modelsim.
I would also like to thank my family that supported me during all my stay abroad.
Abstract

CMOS neurochips, which consist in Microelectrode Array (MEA) manufactured on top of CMOS circuitry, allow the recording of the electrical activity of neural networks in-vitro. As CMOS technology scales down, the number of electrodes increases, so does the amount of data to be processed. Hence the need for an off-chip acquisition system which allows data compression prior to visualization on a computer. Thus, a data compression algorithm based on the delta compression scheme is implemented on a FPGA. This FPGA processes incoming neural signals form the CMOS-based MEA before sending it to a personal computer. An Ethernet interface is therefore also developed in order to communicate with the PC. Finally, an artificial spike generation system is designed in order to perform experimental verification of the data compression algorithm.

I neurochip CMOS, che consistono in una serie di microelettrodi integrati (MEA) con circuiti CMOS, permettono di rilevare l’attività elettrica di una rete neuronale in-vitro. Con la riduzione della scala della tecnologia CMOS, il numero di elettrodi aumenta così come la quantità di dati da processare. Dunque la necessità di un sistema di acquisizione off-chip che permetta di comprimere i dati prima che vengano visualizzati su un computer. Si è dunque implementato su una FPGA un algoritmo per la compressione basato sulla compressione delta. La suddetta FPGA processa i segnali provenienti dal sensore CMOS-MEA prima di inviarli verso un computer. Un’interfaccia Ethernet è stata dunque sviluppata per permettere questa comunicazione. Infine, per verificare sperimentalmente il funzionamento dell’algoritmo, è stato sviluppato un sistema per la generazione artificiale dei segnali.

Los neurochip CMOS, consistentes en una serie de electrodos integrados (MEA) con circuitos CMOS, permiten relevar las actividades eléctricas de una red neural in-vitro. Con la reducción de la escalera de la tecnología CMOS, el número de electrodos aumenta así como la cantidad de datos para procesar. Por esta razón se crea la necesidad de un sistema de adquisición off-chip que permite comprimir los datos antes de que sean visualizados en el ordenador. Ha sido implementado, sobre una FPGA, un algoritmo por la compresión basado en la compresión delta. Esta FPGA procesa las señales que vienen desde el sensor CMOS-MEA antes de enviarlas al ordenador. Para permitir esta comunicación, se ha desarrollado una interfaz Ethernet. Por fin, para verificar experimentalmente el funcionamiento del algoritmo, se ha creado un sistema para la generación artificial de las señales.
Les neuropuces CMOS, qui sont constituées d’une matrice de micro électrodes (MEA) fabriquée au-dessus d’un circuit CMOS, permettent d’enregistrer l’activité électrique d’un réseau de neurone. Cependant, la réduction d’échelle de la technologie CMOS entraîne une nette augmentation du nombre d’électrodes et donc de la quantité de données à traiter. Par conséquent, un système d’acquisition de données extérieur au dispositif, permettant la compression des données avant d’être visualisée par un ordinateur, est requis. Un algorithme de compression de données basé sur la compression delta a donc été implémenté sur un FPGA. Ce dernier traite les signaux provenant du capteur CMOS-MEA avant d’être envoyer à l’ordinateur. Afin de permettre cette communication, une interface Ethernet a été conçue. Finalement, un générateur de signaux artificiels a été développé afin d’accomplir la vérification expérimentale de l’algorithme de compression de données.
B Algorithm using a double clock RAM: VHDL code 38
   B.1 Delta compression algorithm 38
   B.2 Dual clock RAM block 42

C Serial to parallel interface: VHDL code 44
   C.1 Testbench that simulates the ADC output 45

D MAC wrapper interface: VHDL code 48
   D.1 Top level block 48
   D.2 User constraint file 59
List of Figures

1.1 Commercial microelectrode array sensor ........................................... 2
1.2 SEM image of a cell culture lying on a CMOS-based MEA [1] .................. 3
1.3 SEM image of the three-dimensional electrodes ................................. 4
1.4 The packaged MEA measuring 5 cm in edge ........................................ 4
1.5 Conceptual cross-section of a CMOS-based MEA ................................. 4

2.1 MEA overview .................................................................................... 6
2.2 System overview and data flow ........................................................ 7
2.3 FPGA component overview .............................................................. 8
2.4 Experimental setup ............................................................................ 9

3.1 Train of neural spikes ....................................................................... 11

4.1 Data stream ....................................................................................... 13
4.2 Simulink model used to obtain input data for the testbench ................. 14
4.3 Single channel algorithm flow chart ................................................. 14
4.4 Zoom over a single spike ................................................................ 15
4.5 Flow chart of the algorithm using the RAM ...................................... 16
4.6 Timing of the algorithm ................................................................. 17
4.7 Multichannel algorithm block scheme .............................................. 18
4.8 Multichannel algorithm flow chart .................................................. 19

5.1 Fully parallel FIR filter ................................................................. 21
5.2 Shared resources FIR filter .............................................................. 21
5.3 Bode plot of the filter transfer function ........................................... 22

6.1 Pinout of the ADCS7478 ................................................................. 24
6.2 Timing diagram of the serial output .................................................. 25
6.3 PCB board with the ADC ............................................................... 26

7.1 MAC frame ...................................................................................... 27
7.2 MAC wrapper architecture blocks ................................................... 31
7.3 Frame transfer through the Local Link interface ............................... 32
7.4 MAC wrapper design top level block .............................................. 33
List of Tables

4.1 Algorithm block utilization summary ........................................ 17
5.1 FIR filter specifications ......................................................... 22
5.2 SNR due to quantization ......................................................... 23
5.3 Device utilization summary ..................................................... 23
7.1 Transmit FIFO Local Link interface signals ............................. 32
7.2 Main signals of MAC wrapper design top level block ............. 34
Chapter 1

Introduction

In order to understand the dynamics of large neural networks, where information is widely distributed over thousands of cells, one of today’s challenges is to successfully record the simultaneous activities of as many neurons as possible. These observations of the neural culture activity are very useful for the study of the brain functionality. They are vital for branches of neuroscience such as brain engineering and cellular neuroscience.

Nevertheless, how is it possible to record the electric signals generated by a neural culture? Nowadays, Microelectrode arrays (MEAs) are the answer. These devices measure the extracellular voltage of neural cells generated by action potentials. An action potential (or nerve impulse) is a transient alteration of the transmembrane voltage (or membrane potential) across an excitable membrane generated by the activity of voltage-gated ion channels embedded in the membrane. Action potentials play multiple roles in several types of excitable cells such as neurons, myocytes, and electrocytes.

1.1 Microelectrode arrays

MEAs are available for all kinds of extracellular multichannel recording systems. Moreover in culture chambers, it is possible to cultivate tissues or cell cultures directly on the surface of MEAs. They enable stepping from the observation of the electrical behaviour of single cells toward the simultaneous analysis of population of neural cells [2]. Thus, cell cultures on MEAs are suitable for the development of sensors for:
• Drug development through screening of pharmacological effects of compounds [3].

• Building hybrid systems, containing neural cells and an electronic part [4]. The system takes advantage of the learning capability of the cell culture and the computation power of the electronic circuitry.

• Understanding how the brain is operating by studying the electrical behavior of cell cultures. For example, learning [2] and epilepsy [5] can be examined using cell cultures on MEAs.

An example of a commercial MEA is depicted in Figure 1.1. These MEAs usually contain 60-120 electrodes, with electrode sizes ranging from 10 to 50 µm, and inter-electrode spacing ranging between 30 and 200 µm [6]. These dimensions are much larger than the 5-20 µm typical size of vertebrate neurons used during electrophysiological experiments [7]. Thus, commercial MEAs suffer from a very low spatial resolution.

![Figure 1.1: Commercial microelectrode array sensor, after Multi Channel System MCS GmbH (from [6])](image)

However recently, a new generation of CMOS-based MEAs containing a high density and a large number of electrodes has been proposed [1]-[8]. These new devices consist of an electrode array post-processed on top of a CMOS circuit which performs data processing such as amplification, addressing, and analog to digital conversion. A SEM image of a cell culture lying on the surface of a CMOS-based MEA is depicted in Figure 1.2. These new devices have a pitch dimension
as low as 7.8 μm with an electrode diameter of 4.5 μm [9]. Electrophysiological experiments at subcellular resolution are now therefore possible.

![SEM image of a cell culture lying on a CMOS-based MEA](image)

Figure 1.2: SEM image of a cell culture lying on a CMOS-based MEA [1]

### 1.2 Neurochip developed by LSM

The goal of the MEA project in LSM is to manufacture a CMOS-based MEA having a high spatial resolution. Thus, the idea is to have an inter-electrode spacing of about 5 μm or less.

A first version has already been manufactured in CMI, the clean room of EPFL. These MEAs implement an innovative three-dimensional tip electrode array technology, as depicted in Figure 1.3. For small electrodes, an improvement of the electrical coupling up to 20 dB is observed compared to planar electrodes, in simulation [10]. Thus, recording neural activity at subcellular resolution is believed to be feasible. These first MEAs are however passive. They have no CMOS circuitry implemented on-chip. Thus, only sixty electrodes are manufactured on this first MEA version. The packaged MEA, as depicted in Figure 1.4, is compatible with the off-chip data acquisition system provided by Multi Channel Systems MCS GmbH [6].

A second MEA version is currently developed, where CMOS processing is implemented on-chip, as depicted in Figure 1.5. The array is expected to have approximately 10,000 electrodes. The amplification, addressing, and A/D converters
are currently under development.

However, one of the main limitations of CMOS-based MEAs is due to their large number of electrodes (in the order of several thousands). In order to read the whole array of electrodes at a correct sampling frequency, the amount of data which is sent out of the MEAs can be larger than 1 GB/s [8]-[9]. Real-time processing of the neural signals by a personal computer becomes therefore very difficult to perform. Thus, data compression has to be accomplished before sending the data to a PC.

In this project, data compression is implemented on a FPGA. It is the main goal of this master project. The project description and the FPGA overview are introduced in Chapter 2. The spike generator, which is used to create artificial neural signals, is presented in Chapter 3. This signal generator is used during the physical verification of the data compression algorithm. Chapter 4 then describes
in detail the data compression algorithms that have been developed during this project. A FIR filter, which is also implemented on the FPGA, is presented in Chapter 5. The A/D converter used during the experimental verification of the data compression algorithm is described in Chapter 6. The development of the interface between the FPGA and the computer is described in detail in Chapter 7. Finally, Chapter 8 summarizes the achievements of the project, explains future work and gives various ideas to further improve the presented design.
Chapter 2

Project description

2.1 CMOS-based MEA

The targeted MEA for this project is currently being developed and is sketched in Figure 2.1. It hosts a matrix of 128 x 128 electrodes with a pitch of 4-6 µm manufactured on top of a CMOS chip. This CMOS circuit performs data processing directly on-chip (amplification, addressing, etc) as shown in [1]-[9].

Neural activity information to be read from the electrodes has a frequency range from 100 Hz to 10 kHz [10], leading to a full readout frame of 20 kfps (kilo frames per second). This sampling frequency $f_s$ should be at least 20 kHz (the double of the maximum frequency in the range) in order to comply with the Nyquist rule. Moreover, it has been shown in simulation that the extracellular voltage that is being sensed is approximately 2-3 order of magnitude smaller than the intracellular voltage of the cell [10]. The signal that is being sensed is therefore very small. Thus, low-noise amplification stages are required. However, due to
the limited silicon space under each electrode, the amplification stage has to be placed on the sides of the array, as depicted in Figure 2.1. This means that each operational amplifier will be connected to more than one electrode and that some kind of multiplexing will have to be undertaken.

As shown in Figure 2.1, each amplifier can be connected to all electrodes of one zone. A zone is defined as an array of 64 x 4-6 electrodes. However, while reading the extracellular activity of a cell culture, only 25 electrodes in one zone will be sensed. Reading more than 25 electrodes is expected to become very tricky. Each operational amplifier output is then connected to an 8-bit A/D converter [11]. The output rate $R_{out}$ of an A/D converter can therefore be calculated as:

$$R_{out} = f_s \cdot N_{bit} \cdot N_{electrodes} = 4Mb/s$$

### 2.2 System architecture

As already briefly explained in Chapter 1, the amount of data produced by the MEA is very large. Considering zones of 64 x 4 electrodes, which corresponds to a total number of zone equal to 64, the total data output rate of the MEA is equal to 256 Mb/s ($64 \cdot 4Mb/s$). Thus, data compression has to be performed before sending the data to a workstation. This task will be implemented on a FPGA, as depicted in Figure 2.2. It has to be noticed that the number of output pins of the CMOS-based MEA is not yet defined. The digital block described in Figure 2.2 has not yet been developed.

![Figure 2.2: System overview and data flow](image-url)
2.2.1 FPGA overview

The targeted board for this project is a ML505 evaluation board from Xilinx, Inc. implementing a Xilinx Virtex 5 FPGA. The ISE tool from Xilinx will be used for synthesizing, placement and routing of the model. An overview of the components to be implemented on the FPGA is shown in Figure 2.3.

The output of the A/D converter is serial, so a simple serial to parallel 8-bit component has to be implemented on the FPGA, as described in detail in Chapter 6. Then, once the signal is synchronized with the FPGA, it is necessary to remove the input noise generated by the CMOS circuitry. For this purpose, a FIR filter is designed and implemented on the FPGA, as described in Chapter 5. After filtering, an algorithm is needed to detect the occurrence of an incoming spike for each electrode, and transmit only a segment of the recorded data that contains it. This algorithm is based on the delta compression principle, as explained in Chapter 4. This implies that when a spike is detected, the component that implements the algorithm must transmit a given amount of samples before and after the event. Both the delta threshold and the amount of transmitted samples have to be programmable. Furthermore, the data on the output has to be read by a workstation, so an interface is needed. For this application, an Ethernet interface at MAC level is the best choice in terms of simplicity and speed performance. The ML505 board hosts a Marvell 88e1111 gigabit Ethernet transceiver that can be configured and used for our purposes.

![Figure 2.3: FPGA component overview](image_url)
2.3 Experimental setup

In order to test the functionality of the designed system, a reduced version of the system using only one channel will be considered. The input signal will be coming from a programmable signal generator (in our case an Agilent 33250A) preloaded with the data provided by a MATLAB spike generator (see Chapter 3). This signal emulates the one coming from the MEA sensor and carries also the noise generated by it. Then, the signal goes through a PCB hosting the ADCS7478 analog to digital converter in order to be digitalized prior to entering the FPGA. The PCB board pins are showed in detail in Figure 6.3. Each block of the system has to be tested and synchronized with the others in order to communicate with each other.

To resume, the experimental setup is composed by a signal generator, an ADC, the ML505 board and an Ethernet interface system on a PC in order to read the output as described in Figure 2.4.

![Experimental setup](image)

Figure 2.4: Experimental setup
Chapter 3

Spike generator

3.1 Neural spike generation in MATLAB

The CMOS-based MEA that is being developed in LSM (see Section 1.2) is not yet operational. Therefore, in order to experimentally verify the functionality of the data acquisition system implemented on the FPGA, an artificial source which simulates the output signal of the MEA is requested. Noisy spike trains are needed in order to simulate the kind of signals that an extracellular electrode records from a neural culture. Fortunately, a group in the department of computer science and mathematics of the University of Stirling in Scotland has developed a set of MATLAB functions which can perform this task [12]. This complex tool allows adjusting several parameters such as the number of spikes, the event duration, or the number of jittered trains. The main function is called `generatenoisysamples.m` and can be called as follows:

```matlab
[ signals target r1] = generatenoisysamples('parameter1', value, 'parameter2', value)
```

This function generates three outputs. The first one is a 1-dimensional array containing the noisy signal, while the second one is a structure containing amongst other things the actual spike times. It includes both the peaks of the (intracellular) spikes and the spike times used to generate the spike train. The last one is a structure used by the MATLAB random number generator. The parameters of interest are:

- **Duration** This is the duration of the spike train in seconds. It is by default set to 0.1 seconds.
- **SampleRate** The number of samples/second. It is by default set to 100 ksamples/second.

- **Targets** The number of target neurons. It is by default set to 2.

- **N_Jitter** The number of jittered spike trains modelling the effect of nearby neurons whose spike times are correlated with (but not identical to) one of the original spike times.

- **N_Uncorr** The number of uncorrelated (independent) spike trains modelling the effect of nearby neurons whose spike times are not correlated with the original spike times. It is by default set to 15.

In addition, it is also possible to modify the m-file to tune the SNR in dB. This value is then passed to the *awgn* function which adds Gaussian white noise to the generated signal in order to mimic the effect of the electronic and thermal noise sources. This can be accomplished by modifying the parameter `noise_snr` at line 268 of the file. However, for a more detailed description of this function please refer to [13].

A train of spikes generated by this function is showed in Figure 3.1.

![Figure 3.1: Train of neural spikes](image-url)
As it can be seen, the signal over noise ratio is quite high (100 in this case) and the spikes are confined between the value 0 and 2. However by increasing the number of uncorrelated spikes the noise becomes more significant. In the meantime, only high SNRs are considered for testing the system. Moreover, the FIR filter implemented on the FPGA cuts out most of the noise before being dealt by the processing component. The frequency of the spikes generated by this software is around 200 kHz.

### 3.2 Waveform export to signal generator

The signal generated by MATLAB has then to be stored on a programmable signal generator in order to be used in the experimental setup. The Agilent 33250A signal generator comes with the software ™IntuiLink Waveform Editor that allows to create arbitrary waveforms and save them on the instrument memory. The software accepts data organized in column within a text file that can be directly generated by MATLAB. However, the values in the file should be within the range of -1.0 to 1.0.

Once the signal is loaded into the software, a connection with the instrument via RS-232 needs to be established. After that, several parameters can be tuned. The followings are the most important:

- **Frequency** The frequency by which the arbitrary signal will be repeated.

- **Amplitude** Peak to peak amplitude.

- **Offset** The DC offset to the waveform.

The frequency has to be set to the inverse of the time duration of the spikes generated by the MATLAB function. Furthermore, the ADCS7478 component allows only positive signal on its input pin so the peak to peak amplitude can be the same, but an offset of 1 volt has to be applied.
Chapter 4

Data compression

As explained in Section 1.2 the amount of data transmitted to the computer is significantly large. Thus, a compression algorithm is needed. However, most of the information about the neural activity is contained in the spike events. Therefore, most of the information between two successive spikes could be discarded.

The parameters that characterize a spike event are the spike shape, its duration, and the time interval between the spikes. The spike shape exhibits steep rise and fall edges over the mean value. Thus, the spike event can be located by looking at two successive samples. In fact, the idea for the compression algorithm is the following: the difference between a sample and its previous version, which is called delta, is compared to a threshold value. The samples that contain the useful information are then those whose delta exceeds the threshold.

The samples are unsigned integer number represented on 8-bit, and the stream of data entering the algorithm block is formed by subsequent samples belonging to different channel, as described in Figure 4.1.

![Figure 4.1: Data stream](image)

The input file for the testbench is generated by the Simulink model showed in Figure 4.2. It takes from the workspace the variable generated by the MATLAB function described in Chapter 3 nd simulates the FIR filter designed in chapter Section 5.2. The output generated is a vector of data which ranges from 0 to 256 with a sampling frequency of 24 kHz.
4.1 Algorithm for a single electrode

As a first approximation step, only one electrode is considered. It has then been chosen to process the data on the FPGA and send out only the useful data without storing them on the FPGA. In this way, the data is processed and sent almost in real-time. This is the basic idea adopted for each channel of data. The simplest algorithm which implements delta compression is sketched in Figure 4.3.

As it can be seen in Figure 4.3, the information about the spike is complete. Its shape and the original timing are kept. Only the useless data are discarded. Implementing this algorithm in VHDL (see Appendix A), its behaviour can be simulated using Modelsim. The test bench reads the input from a file of binary values and writes the output to another file. Analyzing the results with MATLAB, it can be seen that the data is reconstructable, as depicted in Figure 4.4. The red dots correspond to the output data and the blue line to the original data. The spike shape and the timing are preserved. However, this algorithm only works with a single channel. When it comes to have 25 channels multiplexed on one
pin, a RAM is needed to store the previous samples and the mean value of each channel.

The VHDL code tested is reported in Appendix A.

![Figure 4.4: Zoom over a single spike](image)

**4.1.1 Using a double clock RAM**

In order to output only the data that contains information on the spike event, a RAM memory is needed. In fact, the idea is to wait for a spike event to occur, save the sample, and as soon as the event ends, output these values. The main problem with this scheme is that the information about the timing of the events is difficult to reconstruct. Imagine that when the output data relative to two different spikes is received, the amount of time between them is unknown. Thus, a way to reconstruct the correct timing is to count the incoming samples. In that way, when a spike event occurs, it is possible to save the counter number of its start and stop timing. The samples are acquired in groups of 256 samples. After the last sample is acquired, the output can be either the spike data or either the average signal. The flowchart that summarizes the behaviour of this algorithm is depicted in Figure 4.5.

The assurance that while outputting a set of spike data the memory will not be overwritten by another spike event is required. This may happen frequently since the spikes are usually generated in trains. This means that as soon as a
spike has been completely saved into the RAM, it has to be read before the next sample arrives. This is made possible by using a RAM with two different clocks for read and write operations. The memory can be written with a clock frequency equal to 24 kHz and can be read with the clock frequency of the physical interface (see Chapter 7) used to transmit the data to the computer. The output is then valid only at the end of the acquisition of 256 samples and contains only the useful data. The timing reconstruction is then made by the external receiver which will interpret correctly the stream of data. An example of the signal timing is described in Figure 4.6.

![Flow chart of the algorithm using the RAM](image)

Figure 4.5: Flow chart of the algorithm using the RAM

The VHDL code is reported in Appendix B and the resource used on the FPGA are resumed in Table 4.1.

### 4.2 Algorithm for a channel

Considering the complete system, one has to consider that on a single input pin, samples coming from 25 channels are present, as shown in Figure 2.1. The input signal is first transferred to the FIR filter before being analyzed by the algorithm. The data being carried is supposed to be organized in one byte per sample coming
one after another. The aim is to output only the data concerning the spike event and discard the rest.

In order to extend the basic algorithm presented in Section 4.1 to work with 25 channels, the previous sample of each channel needs to be stored. Thus, evaluation of the delta while receiving a new sample is possible. This requires at least a 25 bytes memory. For our purpose, a simple RAM of 32 bytes \(2^5 = 32\) is sufficient. Moreover, for the calculation of the average of the signal, all samples should be summed. This total is then divided by the number of samples. However, this is complicated because the sum can easily grow indefinitely to an enormous number, which can not be stored. In addition, when dividing the sum by the number of samples, a rational number is obtained, leading to a difficult handling of this data. It would be simpler to store an initial value, evaluate the sum of the delta on a finite number \(N\) of samples, and then add this value to the initial sample. The
The sum of the delta is represented by the following formula:

\[ \Delta_{\text{sum}}(N) = \sum_{i=0}^{N-1} s(i+1) - s(i) \]

The idea is to use two RAM memories on the FPGA, one to store the previous value of each channel and the other to store the delta sum. For the sake of simplicity, from now on those memories will be referred to as \textit{RAM previous} for the first one and as \textit{RAM sum} for the second one. Thus, the single channel algorithm discussed in Section 4.1 can be implemented on this data stream. The two RAMs are driven by a finite state machine as sketched in Figure 4.7. The flow chart of this algorithm is described in Figure 4.8.

The state \textit{WAIT} is used to initialize the signals waiting for the rising edge of the strobe signal. The counters for the stage are reset and the \textit{RAM avg} is reset and initialized.

The state called \textit{DELTA} reads from the \textit{RAM previous} the value of the previous sample and evaluates the delta for the current sample. If the current delta passes the threshold value, the current sample is output.

Finally the \textit{OUT} state writes in the \textit{RAM previous} the current sample to be used the next time a sample from the same channel is received.

![Figure 4.7: Multichannel algorithm block scheme](image)

However, this algorithm is replicating the one seen in Section 4.1 for the 25 electrodes and does not perform a real data compression. A solution would be to modify the code in order to include the dual clock RAM presented in Section 4.1.1, but this will result in inferring a 256-Byte RAM for each electrode on the channel.
4.3 Results

The algorithm achieving the best performance is the one described in section Section 4.1.1 because it has a good compression ratio (up to 94%) and the resources used on the FPGA are acceptable. The only issue is to design a module which is able to correctly read the output from the algorithm block and send it to the interface module. However, this can be implemented by using a FIFO and a logic circuit.
Chapter 5
Filtering

As previously mentioned in Chapter 2, the extracellular voltages which are measured have a frequency range of interest of about 100 Hz to 10 kHz [10]. Moreover, these voltages have very low amplitudes, from approximately 10 µV to 1 mV, with noise which can reach 10 µVrms [14]. Thus, the signal over noise ratio of the measured signal can be very low. Furthermore, additional noise is injected by the amplification stage, the A/D converters, and the MEA packaging. Therefore, before processing the data compression algorithm presented in Chapter 4, the input signals of the FPGA are first low-pass filtered in order to cancel their high frequency noise.

A finite impulse response (FIR) filter has been chosen in order to treat the incoming signals. The specification for the stopband attenuation is given by $1/2N$ dB, $N$ being the number of bits of the data.

5.1 Filter theory

In their simplest form, a M-order FIR filter generates its output $y(n)$ by averaging the last M inputs of sample data, $x(n)$. Each sample stored is called a tap. In order to provide optimal filter characteristics [15], each tap is multiplied by a coefficient or weighting value $h(i)$. Mathematically, the expression of the output is the following:

$$y(n) = \sum_{i=0}^{M} h(i)x(n - i)$$

This formula describes the filter behaviour in mathematical terms and Fig-
Figure 5.1 illustrates a typical FIR filter structure (fully parallel).

The implementation above requires M registers, M+1 multipliers and M adders to work and gives the result after one clock cycle. It is also possible to share the adders and multipliers in time in order to reduce the number of components needed at the price of a lower speed. For example, Figure 5.2 describes an implementation with only one adder and one multiplier, which requires M+1 clock cycles to output the result.

5.2 Filter implementation in MATLAB

Fdatool by MATLAB™ is an easy tool for the design of FIR (finite input response) and IIR (infinite input response) filters. It works by filling in a graphical interface with the specifications and the transfer function type. In our case, a filter order of 30 achieves an acceptable filter transfer function. The “least squares” FIR filter type has been chosen because it does not have an high ripple in the passband, while maintaining the filter order quite low. In Table 5.1 are reported the filter specifications and in Figure 5.3 the Bode plot of its transfer function. The fdatool returns, among a various set of parameters, the list of the FIR filter coefficients as well as its complete transfer function.
Table 5.1: FIR filter specifications

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Values</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sampling frequency</td>
<td>24</td>
<td>KHz</td>
</tr>
<tr>
<td>Order</td>
<td>30</td>
<td>-</td>
</tr>
<tr>
<td>First Stopband Frequency</td>
<td>50</td>
<td>KHz</td>
</tr>
<tr>
<td>First Passband Frequency</td>
<td>0.1</td>
<td>KHz</td>
</tr>
<tr>
<td>Second Passband Frequency</td>
<td>10</td>
<td>KHz</td>
</tr>
<tr>
<td>Second Stopband Frequency</td>
<td>10.5</td>
<td>KHz</td>
</tr>
<tr>
<td>First Stopband Weight</td>
<td>48</td>
<td>dB</td>
</tr>
<tr>
<td>Passband Weight</td>
<td>1</td>
<td>dB</td>
</tr>
<tr>
<td>Second Stopband Weight</td>
<td>48</td>
<td>dB</td>
</tr>
</tbody>
</table>

5.2.1 MATLAB FIR filter implementation

The filter coefficients calculated by MATLAB are 16-bit numbers in fixed point format. However, a reduction of the less significant bits can be performed by means of quantization. Therefore, the memory space required to store the coefficients in the FPGA is lowered, and the operational blocks of the filter (adders and multipliers) can be simpler and smaller. However, one must control that the signal over noise ratio introduced by the quantization is still acceptable for the system. Thus several simulations, as described in Table 5.2, were run on MATLAB in order to find the optimal number of bits for the coefficients (12) respecting the
specifications of the stopband attenuation.

<table>
<thead>
<tr>
<th>Coefficients quantization [bit]</th>
<th>SNR [dB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>-8,7</td>
</tr>
<tr>
<td>5</td>
<td>1,42</td>
</tr>
<tr>
<td>8</td>
<td>22,55</td>
</tr>
<tr>
<td>10</td>
<td>32,5</td>
</tr>
<tr>
<td>11</td>
<td>33,37</td>
</tr>
<tr>
<td>12</td>
<td>49,34</td>
</tr>
<tr>
<td>15</td>
<td>56,7</td>
</tr>
</tbody>
</table>

Table 5.2: SNR due to quantization

5.3 Filter implementation in the FPGA

Once the filter has been designed, it needs to be implemented on the FPGA. It’s possible to instantiate the components either in VHDL or either by using the™IP core generator on the Xilinx ISE tool. The second option has been chosen in this case because it automates the optimization of the FPGA resources and save time. In order to pass to the core generator all the needed parameters, MATLAB allows to store the FIR filter characteristics on a file with the .coe extension which can be directly read by the ISE tool.

For the FIR filter datasheet, please refer to [16].

Table 5.3 reports the utilization of the FPGA resources.

<table>
<thead>
<tr>
<th>Logic utilization</th>
<th>used</th>
<th>available</th>
<th>utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of slice registers</td>
<td>376</td>
<td>69120</td>
<td>0%</td>
</tr>
<tr>
<td>Number of slice LUTs</td>
<td>296</td>
<td>69120</td>
<td>0%</td>
</tr>
<tr>
<td>Number of fully used LUT-FF pairs</td>
<td>164</td>
<td>508</td>
<td>32%</td>
</tr>
<tr>
<td>Number of bonded IOBs</td>
<td>17</td>
<td>640</td>
<td>2%</td>
</tr>
<tr>
<td>Number of BUFG / BUFGCTRLs</td>
<td>1</td>
<td>32</td>
<td>3%</td>
</tr>
<tr>
<td>Number of DSP48Es</td>
<td>9</td>
<td>64</td>
<td>14%</td>
</tr>
</tbody>
</table>

Table 5.3: Device utilization summary
Chapter 6

A/D Converter

The signal coming from the MEA is analog and has to be digitalized to be processed in the FPGA. The chosen component is the ADCS7478 from National Semiconductor. Its pins are described in Figure 6.1.

![Figure 6.1: Pinout of the ADCS7478](image)

The frequency of the output signal SDATA is defined by the input clock signal SCLK. The maximum working frequency is 20 MHz, much more than the 48 kHz needed for our application. Moreover, this device is compatible with the SPI™ interface, so it can be used to drive other ADCS7478. The A/D converter is controlled by the input signal CS (chip select). For further development of the system, it will be useful to drive the A/D converters of each channel. In the meanwhile, this signal is just used to initiate the conversion and the serial data transfer.

The timing diagram of the serial output signal SDATA is described in Figure 6.2. When \( \overline{CS} \) goes low, the subsequent rising and falling edges of SCLK will be labelled with reference to the falling edge of \( \overline{CS} \). At this point, when \( \overline{CS} \) is switched to zero, the signal SDATA leaves the TRI-STATE and the converter
moves from track mode to hold mode. The input signal is sampled and held for
conversion at the falling edge of $\overline{CS}$. The converter moves from hold mode to
track mode on the 13th rising edge of SCLK. The SDATA signal will be placed
back into TRI-STATE after the 16th falling edge of the clock signal, or at the ris-
ing edge of $\overline{CS}$, whichever occurs first. After a conversion is completed, the quiet
time $t_{quiet}$ must be satisfied before before the chip select signal $\overline{CS}$ can be brought
low. Therefore, in order to read a complete sample from the ADCS7478, 16 clock
cycles are required. This time is called conversion time and is indicated by $t_{conv}$.
The output is an 8-bit word in straight binary format. This word is serial with
four zeroes at the beginning and four zeroes at the end of the 16 bits, as described
in Figure 6.2.

![Figure 6.2: Timing diagram of the serial output](image)

A serial to parallel converter is thus needed before the data can be processed
by the FIR filter. This component has been implemented in VHDL. It outputs
the word containing the sample at the rising edge of the “data valid” signal and
discards the unuseful spacing bits. The VHDL code is reported in Appendix C.
This block has been successfully tested with a testbench (see C.1) that behaves
exactly as the output of the A/D converter.

The ADC has been mounted on a PCB board, as described in Figure 6.3,
in order to simulate the behavior of one channel. The initial signal is generated
by a MATLAB function and then loaded in a signal generator, as described in
Chapter 3. This signal is connected on the PCB to the input of the SMA connector,
which that corresponds to the input signal $V_{in}$ of the ADC. By choosing a supply voltage $V_{DD}$ of 3 V, the clock signal is a square wave having a low level of 0 V, a high level of 1.3 V, an offset of 1.3 V, and a duty cycle of 50%. Due to the fact that the conversion time $t_{conv}$ is 14 times the clock period, the clock frequency is 384 kHz in order to have a sampling frequency $f_c$ of 24 kHz. Finally, the $CS$ signal has to be driven by a square wave that stays at low level for 16 clock periods and at high level for 100 ns (the minimum time to respect is $t_{quiet} = 50$ ns).

![Figure 6.3: PCB board with the ADC](image)

Figure 6.3: PCB board with the ADC
Chapter 7

Interface towards the computer

This chapter presents the operation of the Ethernet interface which allows the communication with the PC, as showed in Figure 2.4. The protocol to be used to communicate at MAC level is explained in Section 7.1 and in Section 7.2 is reported the implementation of the client side on the FPGA. Finally, the transmission of a frame through the interface is presented in Section 7.3.

7.1 Medium access control (MAC)

The Medium Access Control (MAC) protocol is used to provide the data link layer of the Ethernet LAN system. The MAC protocol encapsulates a SDU (payload data) by adding a 14-byte header (Protocol Control Information) before the data, and appending a 4-byte (32-bit) Cyclic Redundancy Check (CRC) after the data. The entire frame, as depicted in Figure 7.1 is preceded by a small idle period (the minimum inter-frame gap, 9.6 $\mu$s) and an 8-byte preamble which includes the start of the frame delimiter.

```
14 Bytes 46 - 1500 Bytes 4 Bytes
```

Figure 7.1: MAC frame

- **Preamble**
  
  Before the transmission starts, an idle time is required in order to allow
a small time interval for the receiver electronics in each of the nodes to settle after completion of the previous frame. A node starts transmission by sending an 8-byte (64-bit) preamble sequence. This consists of 62 alternating 1’s and 0’s followed by the pattern 11. Strictly speaking the last byte which finishes with the ’11’ is known as the ”Start of Frame Delimiter”.

The purpose of the preamble is to allow time for the receiver in each node to achieve lock of the receiver Digital Phase Lock Loop, which is used to synchronise the receive data clock to the transmit data clock. At the point when the first bit of the preamble is received, each receiver may be in an arbitrary state (i.e. have an arbitrary phase for its local clock). During the course of the preamble the receiver acquires the correct phase. However, during this process, it may miss (or gain) a number of bits. A special pattern (11), is therefore used to mark the last two bits of the preamble. When this is received, the Ethernet receive interface starts collecting the bits into bytes for the processing by the MAC layer. It also confirms the polarity of the transition by representing a ’1’ bit to the receiver (as a check in case this has been inverted).

- **Header**

The header, as described in Figure 7.1, consists of three parts:

1. A 6-byte destination address, which specifies either a single recipient node (unicast mode), a group of recipient nodes (multicast mode), or the set of all recipient nodes (broadcast mode).

2. A 6-byte source address, which is set to the sender’s globally unique node address. This may be used by the network layer protocol to identify the sender, but usually other mechanisms are used (e.g. arp). Its main function is to allow address learning which may be used to configure the filter tables in a bridge.

3. A 2-byte type field, which provides a Service Access Point (SAP) to identify the type of protocol being carried (e.g. the values 0x0800 are used to identify the IP network protocol, other values are used to indicate other network layer protocols). In the case of IEEE 802.3 LLC,
this may also be used to indicate the length of the data part. The type field is also used to indicate when a Tag field is added to a frame.

- **CRC**
  The final field in an Ethernet MAC frame is called a Cyclic Redundancy Check (sometimes also known as a Frame Check Sequence). A 32-bit CRC provides error detection in the case where line errors (or transmission collisions in Ethernet) result in corruption of the MAC frame. Any frame with an invalid CRC is discarded by the MAC receiver without further processing. The MAC protocol does not provide any indication that a frame has been discarded due to an invalid CRC.
  The link layer CRC therefore protects the frame from corruption while being transmitted over the physical medium (the cable). A new CRC is added if the packet is forwarded by the router to another Ethernet link. While the packet is being processed by the router, the packet data is not protected by the CRC. Router processing errors must be detected by the network or the transport-layer checksums.

- **Inter Frame Gap**
  After transmission of each frame, the transmitter must wait for a period of 9.6 $\mu$s (at 10 Mbps) to allow the signal to propagate through the receiver electronics and reach destination. This period of time is known as the Inter-Frame Gap (IFG). While every transmitter must wait for this time between sending frames, receivers do not necessarily see a "silent" period of 9.6 microseconds.

- **Byte order and illegal frames**
  It is important to realise that nearly all serial communications systems transmit the least significant bit of each byte first at the physical layer. Ethernet supports broadcast, unicast, and multicast addresses.
  Furthermore, any frame which is received and which is less than 64 bytes is illegal, and is called a "runt". In most cases, such frames arise from a collision, and while they indicate an illegal reception, they may be observed on correctly functioning networks. Any frame which is received and which
is greater than the maximum frame size, is called a "giant". In theory, the
jabber control circuit in the transceiver should prevent any node from gen-
erating such a frame, but certain failures in the physical layer may also give
rise to over-sized Ethernet frames. Any frame which does not contain an
integral number of received bytes is also illegal. Moreover, a receiver has no
way of knowing which bits are legal, and how to compute the CRC-32 of the
frame. In all three cases, the frame is discarded by the receiver.

7.2 Using Xilinx® MAC wrapper

The interface towards the computer uses the protocol previously described in Sec-
tion 7.1. Moreover, it also needs a physical interface called PHY as an abbreviation
for the physical layer of the OSI model in between the PC and the FPGA. A PHY
connects a link layer device (often called a MAC) to a physical medium such as
an optical fibre or copper cable. The board ML505 hosts a Marvell Alaska PHY
device (88E1111) that satisfy this need. It operates at 10/100/1000 Mb/s and
supports MII, GMII, RGMII, and SGMII interface modes with the FPGA. The
PHY is connected to a Halo HFJ11-1G01E RJ-45 connector with built-in mag-
netics allowing the connection using a crossover RJ-45 cable. The configuration
of this device has to be done by the FPGA, as well as the wrapping of the data
to be sent. This is made possible by using a MAC wrapper module which is
generated by the ™Xilinx CORE Generator software. This component is called
Virtex®-5 FPGA Embedded Tri-Mode Ethernet MAC Wrapper and automates
the generation of HDL wrapper files for the Embedded Tri-Mode Ethernet MAC
in Virtex-LXT FPGAs.

The interface mode selected for this project is the MII (Media Independent
Interface) because it is simple to configure an has an acceptable speed of 100
Mb/s. This setting has to be entered in the Ethernet MAC wrapper screens as
described in the getting started guide [17] on page 23. In this screens only EMAC0
should be activated and the MAC address of the FPGA client should be set to
AA-BB-CC-DD-EE-FF. This address is ordered for the least significant byte in
the register to have the first byte transmitted or received. As an example, the
previous MAC address is entered as FF-EE-DD-CC-BB-AA.

When the core is generated, it comes with an example design providing VHDL instances that can be customized [18]. This design links the local link level to an address swap module that simply takes the input frame from the receiver FIFO and switches the source and destination addresses prior to sending this modified frame into the transmitter FIFO. This is done just for simulation purposes, in order to exercise the two FIFO and to show the functionality of the EMAC block. In Figure 7.2 is sketched the block diagram of this design without including the address swap module. In fact, in this project, only some data have to be transmitted from the FPGA to the PHY. Therefore the address swapping module is useless. The two FIFO provided in the design can be used to receive and transmit data in Local Link format. Focusing on the transmit FIFO, the communication with the external is dictated by the signals listed in Table 7.1.

The transmit FIFO accepts frames in Local Link format and stores them in block RAM for transmission through the EMAC module. As soon as a full frame is written into the transmit FIFO, it presents the data to the Ethernet MAC transmitter client interface. The whole frame is transmitted prior to receiving the acknowledge signal from the Ethernet MAC. The data flow transferred on the
<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Clock domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tx_ll_clock</td>
<td>Input</td>
<td>N/A</td>
<td>Read clock for local link interface</td>
</tr>
<tr>
<td>tx_ll_reset</td>
<td>Input</td>
<td>tx_ll_clock</td>
<td>Synchronous reset</td>
</tr>
<tr>
<td>tx_ll_data_in[7:0]</td>
<td>Input</td>
<td>tx_ll_clock</td>
<td>Write data to be sent to the transmitter</td>
</tr>
<tr>
<td>tx_ll_sof_in_n</td>
<td>Input</td>
<td>tx_ll_clock</td>
<td>Start of frame indicator</td>
</tr>
<tr>
<td>tx_ll_eof_in_n</td>
<td>Input</td>
<td>tx_ll_clock</td>
<td>End of frame indicator</td>
</tr>
<tr>
<td>tx_ll_src_rdy_in_n</td>
<td>Output</td>
<td>tx_ll_clock</td>
<td>Source ready indicator</td>
</tr>
<tr>
<td>tx_ll_dst_rdy_in_n</td>
<td>Output</td>
<td>tx_ll_clock</td>
<td>Destination ready indicator</td>
</tr>
</tbody>
</table>

Table 7.1: Transmit FIFO Local Link interface signals

Local Link interface from source to destination is governed by the four active low signals: `tx_ll_sof_in_n, tx_ll_eof_in_n, tx_ll_src_rdy_in_n` and `tx_ll_dst_rdy_in_n`. Only when both `tx_ll_src_rdy_in_n` and `tx_ll_dst_rdy_in_n` are asserted simultaneously the data is allowed to flow from source to destination as showed in Figure 7.3, where an 8-Byte frame is transferred.

![Figure 7.3: Frame transfer through the Local Link interface](image)

7.3 Transmission of a frame through the interface

The example design provided by Xilinx® has to be modified in order to allow transferring a frame from the FPGA to an external client. First of all, the address swapping module is useless in this project and has to be deleted from the blocks. The idea is to have some data stored in a memory, and then transmit it through the EMAC module to the PHY. Therefore, the data has to be input in the transmitting FIFO in the local link interface as explained before. The frame to be sent has to
be in the correct format, as explained in Section 7.1 and the FIFO signal have to be driven correctly. All of this is implemented in the VHDL code reported in Appendix D and in the top level block is reported in Figure 7.4. The most important signals are listed in Table 7.2. The ones that do not appear in this table are outputs for the EMAC statistics.

This design adds to the blocks in Figure 7.2 a memory and a finite state machine. The former stores the frame to be sent and the latter sends it by attaching the preamble and the start of the frame delimiter. To be compliant with the signal utilization showed in Figure 7.3, the finite state machines drives them accordingly. For the moment, the receive FIFO can be neglected since it is not needed for receiving data from the PC. The $tx_{ll}src_{rdy}_{n.0.i}$ signal is connected to a gpio button on the ML505 board and will trigger the frame transfer.

![Figure 7.4: MAC wrapper design top level block](image)

Section D.2 reports the user constraint file that contains the informations on the FPGA pins connections to the corrects pins of the PHY on the ML5050 board. This file also contains informations about the timing and the setup times to be applied on some pins.

The simulations of the post-routed design showed that the frame is succesfully transferred from the source to the destination. However, the next step is to load the design in the FPGA, connect the ML505 board to the PC via a cross-over RJ-45 cable and check that the Ethernet connection has been established. For
Table 7.2: Main signals of MAC wrapper design top level block

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mii_rxd_0[3:0]</td>
<td>Input</td>
<td>Received data (4 bits)</td>
</tr>
<tr>
<td>mii_rx_clk_0</td>
<td>Input</td>
<td>Receive clock for local link interface</td>
</tr>
<tr>
<td>mii_rx_dv_0</td>
<td>Input</td>
<td>Data valid signal for received data</td>
</tr>
<tr>
<td>mii_rx_er_0</td>
<td>Input</td>
<td>Error signal for received data</td>
</tr>
<tr>
<td>mii_txd_0[3:0]</td>
<td>Output</td>
<td>Transmitted data (4 bits)</td>
</tr>
<tr>
<td>mii_tx_clk_0</td>
<td>Input</td>
<td>Receive clock for local link interface</td>
</tr>
<tr>
<td>mii_tx_er_0</td>
<td>Output</td>
<td>Error signal for transmitted data</td>
</tr>
<tr>
<td>mii_tx_en_0</td>
<td>Output</td>
<td>Transmit enable signal</td>
</tr>
<tr>
<td>reset</td>
<td>Input</td>
<td>Synchronous reset</td>
</tr>
<tr>
<td>tx_llsrc_rdy_0_i</td>
<td>Input</td>
<td>Source ready indicator for the transmitter</td>
</tr>
</tbody>
</table>

For this purpose, it is needed to associate an IP address to the MAC address of the EMAC module on the FPGA. This is done by editing the ARP table on the PC. Depending on the operative system, the command is different. However, when using Linux, it is:

```
arp -s 1.2.3.5 AA-BB-CC-DD-EE-FF
```

This command will add an entry in the ARP table that associates the IP address 1.2.3.5 to the MAC address AA-BB-CC-DD-EE-FF. Then the connection can be configured on the PC using this IP address. Once the connection is established, the packets will start to be sent. At this point, a network protocol analyzer is needed in order to get the data at MAC level. This task is done by an opensource software called Wireshark that allows to filter the incoming packets and easily get the data at MAC level.
Chapter 8

Conclusions

8.1 Achievements

The off-chip system has been developed block by block and each of them has been tested separately. Although each block is functional, the connections with each others have still to be implemented. Nevertheless the performances of the FIR filter and the algorithm are satisfactory. The Ethernet interface has required a lot of study and work on the Xilinx MAC wrapper and it is ready to be tested using an external source for the data. Moreover, the artificial spike generation is ready to use as explained in Chapter 6.

8.2 Future improvements

The first important step is to link together all the blocks of the experimental setup showed in Figure 2.4 and verify the correctness of its operation. To this purpose, one must check that the data input coming from the A/D converter can be reconstructed on the PC. This implies the synchronization of all the blocks with each other by using FIFOs.

It will be also interesting to add a noise level estimator module on the FPGA in order to adapt the threshold of the algorithm accordingly, as reported in [19]. In this way, the false events generated by noisy spikes can be suppressed in the output.
Appendix A

Single channel algorithm: VHDL code

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

entity delta_compr is
  Generic ( N_BITS: natural := 8);
  Port ( clk : in STD_LOGIC;
         rst_b : in STD_LOGIC;
         data_in : in STD_LOGIC_VECTOR (N_BITS-1 downto 0);
         data_out : out STD_LOGIC_VECTOR (N_BITS-1 downto 0));
end delta_compr;

architecture Behavioral of delta_compr is
  signal acquire : boolean := FALSE;
  signal cnt : integer := 0;
  signal sum : integer := 0;
begin
  delta_comp : process(clk, rst_b)
  variable delta : signed (N_BITS-1 downto 0) := CONV_SIGNED(0, N_BITS);
  variable prev_sample : signed (N_BITS-1 downto 0) := CONV_SIGNED(0, N_BITS);
  --variable cnt : integer := 0;
  --variable sum : integer := 0;
  variable average : integer := 0;
  constant delta_th : signed := CONV_SIGNED(2, N_BITS);

  begin
    if (clk’event and clk = ’1’) then
      sum <= sum + CONV_INTEGER(signed(data_in));
      if cnt /= 0 then
        average := sum / cnt; -- average over all samples
      end if
  end if
cnt <= cnt + 1;
delta := abs( signed(data_in) - prev_sample ); -- current delta
if rst_b = '0' then
    data_out <= STD_LOGIC_VECTOR( CONV_SIGNED(0, N_BITS) );
    prev_sample := CONV_SIGNED(0, N_BITS);
elsif (delta >= delta_th) or ( abs( CONV_INTEGER(signed(data_in))) >= 60 ) then
    data_out <= STD_LOGIC_VECTOR(prev_sample);
    acquire <= TRUE;
elsif acquire = TRUE then
    data_out <= STD_LOGIC_VECTOR(prev_sample); -- store last sample
    acquire <= FALSE;
else
    data_out <= STD_LOGIC_VECTOR(CONV_SIGNED(average, N_BITS));
end if;
    prev_sample := signed(data_in);
end if;

end process delta_comp;

end Behavioral;
Appendix B

Algorithm using a double clock
RAM: VHDL code

B.1 Delta compression algorithm

-- Company: EPFL
-- Engineer: Carrozzo Michelangelo
-- Create Date: 12:24:15 02/25/2009
-- Design Name:
-- Module Name: delta_compr_R1 - Behavioral
-- Target Devices:
-- Tool versions:
-- Description:
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

-- Uncomment the following library declaration if instantiating any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity delta_compr is
Generic ( DATA_WIDTH : natural := 8;
          ADDR_WIDTH : natural := 8);
Port ( clk_rd : in STD_LOGIC;
       clk_wr : in STD_LOGIC;
       rst_b : in STD_LOGIC;
       data_in : in STD_LOGIC_VECTOR (DATA_WIDTH-1 downto 0);
       data_valid : out STD_LOGIC;
       data_out : out STD_LOGIC_VECTOR (DATA_WIDTH-1 downto 0));
architecture Behavioral of delta_compr is

------------------------------------------------
-- Signal declaration
------------------------------------------------
signal acquire : boolean := FALSE;
signal cnt     : integer := 0;
signal delta_sum : integer := 0;
signal start_read : STD_LOGIC := '0';
signal average : integer := 0;
signal rst_b_mem : STD_LOGIC := '0';
signal wr_nrd   : STD_LOGIC := '0';
signal address_rd : STD_LOGIC_VECTOR (ADDR_WIDTH-1 downto 0) := (others => '0');
signal address_wr : STD_LOGIC_VECTOR (ADDR_WIDTH-1 downto 0) := (others => '0');
signal data_out_ram: STD_LOGIC_VECTOR (DATA_WIDTH-1 downto 0);
signal event_start : STD_LOGIC_VECTOR (ADDR_WIDTH-1 downto 0) := (others => '0');
signal event_stop  : STD_LOGIC_VECTOR (ADDR_WIDTH-1 downto 0) := (others => '0');

begin

ram_prev:entity work.regfile_dual_clk(rtl)
generic map( ADDR_WIDTH => ADDR_WIDTH,
DATA_WIDTH => DATA_WIDTH)
port map ( clk_rd => clk_rd,
clk_wr => clk_wr,
rst_b => rst_b_mem,
wr_nrd  => wr_nrd,
addr_rd  => address_rd,
addr_wr  => address_wr,
din => data_in,
dout => data_out_ram);

delta_comp : process(clk_wr, rst_b)

variable delta   : signed (DATA_WIDTH-1 downto 0)
:= CONV_SIGNED ( 0, DATA_WIDTH);
variable prev_sample : unsigned (DATA_WIDTH-1 downto 0)
:= CONV_UNSIGNED( 0, DATA_WIDTH);
variable cnt_events : integer := 0;
variable delta_sum : integer := 0;
variable sample_cnt : integer := 0;
constant delta_th  : integer := 5;
constant n_delta   := integer := 256;

begin

if rising_edge(clk_wr) then
-- current delta
delta := CONV_SIGNED(
    CONV_INTEGER(unsigned(data_in)) - CONV_INTEGER(prev_sample),
    DATA_WIDTH);
rst_b_mem <= '1';
data_valid <= '0';
start_read <= '0';

if (abs(CONV_INTEGER(delta)) >= delta_th) or
    ( CONV_INTEGER(unsigned(data_in)) >= 180 ) or
    ( CONV_INTEGER(unsigned(data_in)) <= 40) then
    if acquire = FALSE then
        -- It is the start of the event
        -- save the counter
        event_start <= STD_LOGIC_VECTOR( CONV_UNSIGNED(cnt, DATA_WIDTH));
        address_wr <= STD_LOGIC_VECTOR( CONV_UNSIGNED(0,DATA_WIDTH) );
        wr_nrd <= '1';
        acquire <= TRUE;
    else
        acquire <= TRUE;
        wr_nrd <= '1';
        cnt_events := cnt_events + 1;
        address_wr <= STD_LOGIC_VECTOR( CONV_UNSIGNED(UNSIGNED(address_wr)) + 1, DATA_WIDTH) );
    end if;
elsif cnt_events > 10 then
    -- The event is finished
    acquire <= FALSE;
    cnt_events := 0;
    -- save the counter
    event_stop <= STD_LOGIC_VECTOR( CONV_UNSIGNED(cnt, DATA_WIDTH));
else
    acquire <= FALSE;
    wr_nrd <= '0';
    cnt_events := 0;
end if;

-- Counter control

if cnt < n_delta then
    -- continue to count
    cnt <= cnt + 1;
    delta_sum <= delta_sum + CONV_INTEGER(delta);
else
    -- 256 sample acquired
    cnt <= 0;
    cnt_events := 0;
    delta_sum <= 0;
    data_valid <= '1'; -- data is valid
if acquire = TRUE then
  -- I was acquiring a spike
  wr_nrd <= '0'; -- read the memory
  start_read <= '1';
elsif cnt_events > 10 then
  -- I received a spike and not a noisy interference
  wr_nrd <= '0'; -- read the memory
  start_read <= '1';
else
  -- No events received
  rst_b_mem <= '0';
  start_read <= '0';
  --data_out <= STD_LOGIC_VECTOR(CONV_SIGNED(delta_sum, DATA_WIDTH));
end if;
end if;

prev_sample := unsigned(data_in);
end if;

end process delta_comp;

-----------------------------------------------
-- Output
-----------------------------------------------
read : process(clk_rd, wr_nrd, start_read)
begin

if rising_edge(clk_rd) and cnt = 0 then
  if wr_nrd = '0' and start_read = '1' and
     CONV_INTEGER(UNSIGNED(address_rd)) < 256 then
    -- Read the memory incrementing the address
    address_rd <= STD_LOGIC_VECTOR( CONV_UNSIGNED( CONV_INTEGER(UNSIGNED(address_rd)) + 1, DATA_WIDTH) );
    data_out <= data_out_ram;
  else
    --data_out <= STD_LOGIC_VECTOR(CONV_UNSIGNED(135, DATA_WIDTH));
    data_out <= STD_LOGIC_VECTOR(CONV_UNSIGNED(delta_sum, DATA_WIDTH));
    --address_rd <= (others => '0');
    --data_out <= (others => '0');
  end if;
elsif cnt = 1 then
  address_rd <= (others => '0');
end if;

end process read;

end Behavioral;
B.2 Dual clock RAM block

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity regfile_dual_clk is
  generic ( ADDR_WIDTH : natural := 8; -- # address bits
             DATA_WIDTH : natural := 8); -- # bits per register
  port ( clk_rd, clk_wr, rst_b, wr_nrd: in std_logic;
             -- write ('1') or read ('0')
             addr_rd : in std_logic_vector(ADDR_WIDTH-1 downto 0);
             -- read address
             addr_wr : in std_logic_vector(ADDR_WIDTH-1 downto 0);
             -- write address
             din : in std_logic_vector(DATA_WIDTH-1 downto 0);
             -- data in
             dout : out std_logic_vector(DATA_WIDTH-1 downto 0));
             -- data out
end entity regfile_dual_clk;

architecture rtl of regfile_dual_clk is

  type ram_single is array (2**ADDR_WIDTH-1 downto 0) of
    std_logic_vector(DATA_WIDTH-1 downto 0);
  signal regfile_reg : ram_single;
  --signal regfile_next : ram_single;

begin
  -- memory element (register)
  reg_read: process (clk_rd, rst_b)
  begin

                -------------------------------
    if rising_edge(clk_rd) then
      if wr_nrd = '0' then
        dout <= regfile_reg(to_integer(unsigned(addr_rd)));
      end if;
    end if;

                -------------------------------
  end process;
end architecture;
end process reg_read;

reg_write: process (clk_wr, rst_b)
begin

-----------------------------
if rst_b = '0' then
  regfile_reg <= (others => (others => '0'));
elsif rising_edge(clk_wr) then
  if wr_nrd = '1' then
    regfile_reg(to_integer(unsigned(addr_wr))) <= din;
  end if;
end if;
-----------------------------
end process reg_write;

--dout <= regfile_reg(to_integer(unsigned(addr_rd)));

dout <= regfile_reg(to_integer(unsigned(addr_rd)));
end architecture rtl;
Appendix C

Serial to parallel interface: VHDL code

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ser_to_par is
  generic ( DATA_WIDTH : natural := 8); -- # data bits
  port ( clk : in std_logic;
          rst : in std_logic; -- read CS_b of the ADC
          din : in std_logic; -- data in
          data_valid : out std_logic;
          dout : out std_logic_vector(DATA_WIDTH-1 downto 0));
end entity ser_to_par;

architecture rtl of ser_to_par is

signal par : std_logic_vector(DATA_WIDTH-1 downto 0) := (others => '0');
signal cnt : integer := 0;

begin

  -- memory element (register)
  conv: process (clk, rst)
  begin

    if clk’event and clk = '1' then
      if rst = '1' then
        par <= (others => '0');
        data_valid <= '0';
        cnt <= 0;
      elsif cnt > 19 then
        data_valid <= '0';
        cnt <= 0;
      elsif cnt > 11 then
        data_valid <= '1';
      end if;
    else
      par <= din;
      data_valid <= '1';
      cnt <= cnt + 1;
    end if;
  end process conv;

end process conv;

end architecture rtl;
C.1 Testbench that simulates the ADC output

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library STD;
use STD.TEXTIO.ALL;

entity ser_to_par_tb is
end entity ser_to_par_tb;

architecture bench of ser_to_par_tb is

constant DATA_WIDTH : natural := 8;
-- # data bits
constant CLK_PER : time := 2.6042 us;
-- fclk = 384 kHz
signal clk : std_logic := '0';
signal rst : std_logic := '0';
-- read CS_b of the ADC
signal data_in : std_logic;
-- data in
signal data_valid : std_logic;
signal data_out : std_logic_vector(DATA_WIDTH-1 downto 0);
-- data out
signal verif : std_logic_vector(DATA_WIDTH-1 downto 0)
  := (others => '0');
signal cnt : integer := 16;
```

```vhdl
```
file stimulus: TEXT open READ_MODE is
"//studentspc7/MyDocs/EL/carrozzo/My Documents/
Scolarship/VHDL/RAM/ADC_out.txt";

begin

UUT : entity work.ser_to_par(rtl)
generic map( DATA_WIDTH => DATA_WIDTH)
port map ( clk => clk,
          rst => rst,
          din => data_in,
          data_valid => data_valid,
          dout => data_out);

clk <= not clk after CLK_PER/2;

--------------------------------------------------stimulus
stimul : process
variable in_line : line;
variable var : integer := 0;
--variable cnt : integer := 16;
variable vect: std_logic_vector(DATA_WIDTH-1 downto 0)
:= (others => '0');

begin

rst <= '0';
data_in <= '0';
wait for 3*CLK_PER/2;
rst <= '1';
wait for CLK_PER;
rst <= '0';
--data_in <= (others => '0');
--wait until clk = '1' and clk' event;
data_in <= '0';
wait for 4*CLK_PER; -- 4 leading zeroes

while ( not endfile(stimulus) ) loop

if cnt = 16 then
  cnt <= 1;
  readline(stimulus, in_line);
  -- Must read value into a variable and not a signal
  read(in_line, var);
  vect := STD_LOGIC_VECTOR(CONV_SIGNED(var, DATA_WIDTH));
  -- Transfer read value into a signal
  data_in <= vect(7);
  --cnt   <= cnt +1;

  wait for 6*CLK_PER; -- 6 leading zeroes

  if cnt <= 8 then
    cnt <= 8;
    readline(stimulus, in_line);
    -- Must read value into a variable and not a signal
    read(in_line, var);
    vect := STD_LOGIC_VECTOR(CONV_SIGNED(var, DATA_WIDTH));
    -- Transfer read value into a signal
    data_in <= vect(7);
    --cnt   <= cnt +1;
    wait for 6*CLK_PER; -- 6 leading zeroes

  end if;

end if;

wait for 6*CLK_PER; -- 6 leading zeroes

end loop;

rst <= '0';
data_in <= '0';
wait for 3*CLK_PER/2;
rst <= '1';
wait for CLK_PER;
rst <= '0';
--data_in <= (others => '0');
--wait until clk = '1' and clk' event;
data_in <= '0';
wait for 4*CLK_PER; -- 4 leading zeroes

end process;

end;

46
elsif cnt < 8 then
    data_in <= vect(7-cnt);
    cnt <= cnt +1;
    verif <= vect;
else
    data_in <= '0';
    cnt <= cnt +1;
end if;
wait until clk = '1' and clk' event;
end loop;
end process stimul;

verification : process (data_valid)
begin
    if data_valid = '1' and data_valid' event then
        assert data_out = verif
        report "ERROR"
        severity error;
    end if;
end process verification;
end architecture bench;
Appendix D

MAC wrapper interface: VHDL code

D.1 Top level block

-------------------------------------------------------------------------
-- Title : Virtex-5 Ethernet MAC Example Design Wrapper
-- Project : Virtex-5 Ethernet MAC Wrappers
-- File : MAC_wrapper_example_design.vhd
--
--This is the VHDL design for the Virtex-5 Embedded Ethernet MAC.
--
-- | DESIGN WRAPPER |
-- | ----------------- |
-- | LOCAL LINK WRAPPER |
-- | ____________________ |
-- | BLOCK LEVEL WRAPPER |
-- | --------------------- |
-- | ETHERNET MAC |
-- | WrAPPER |
-- | ------------ |
-- | FSM | LOCAL | I/F |
-- | TX | LINK | PHY |
-- | FIFO | | I/F |
-- | Rx | | |
-- | client | PHY | |
-- | I/F | |
-- |<-- |<-- |
-- |--------------------- |

48
library unisim;
use unisim.vcomponents.all;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-- The entity declaration for the example design.

entity MAC_wrapper_design is
  port(
    -- Client Receiver Interface - EMAC0
    EMAC0CLIENTRXDVLD : out std_logic;
    EMAC0CLIENTRXFRAMEDROP : out std_logic;
    EMAC0CLIENTRXSTATS : out std_logic_vector(6 downto 0);
    EMAC0CLIENTRXSTATSVLD : out std_logic;
    EMAC0CLIENTRXSTATSBYTEVLD : out std_logic;

    -- Client Transmitter Interface - EMAC0
    -- CLIENTEMAC0TXIFGDELAY : in std_logic_vector(7 downto 0);
    EMAC0CLIENTTXSTATS : out std_logic;
    EMAC0CLIENTTXSTATSVLD : out std_logic;
    EMAC0CLIENTTXSTATSBYTEVLD : out std_logic;

    -- MAC Control Interface - EMAC0
    -- CLIENTEMAC0PAUSEREQ : in std_logic;
    -- CLIENTEMAC0PAUSEVAL : in std_logic_vector(15 downto 0);

    -- Clock Signals - EMAC0

    -- MII Interface - EMAC0
    -- MII_COL_0 : in std_logic;
    -- MII_CRS_0 : in std_logic;
    MII_TXD_0 : out std_logic_vector(3 downto 0);
    MII_TX_EN_0 : out std_logic;
    MII_TX_ER_0 : out std_logic;
    MII_TX_CLK_0 : in std_logic;
    MII_RXD_0 : in std_logic_vector(3 downto 0);
    MII_RX_DATAVALID_0 : in std_logic;
    MII_RX_ER_0 : in std_logic;
    MII_RX_CLK_0 : in std_logic;

    -- Source ready
    TX_LL_SRC_RDY_N_0_I : in std_logic;
  );
end MAC_wrapper_design;
Asynchronous Reset

RESET : in std_logic;

end MAC_wrapper_design;

architecture TOP_LEVEL of MAC_wrapper_design is

--------------------------------------------------------------
-- Component Declarations for lower hierarchial level entities
--------------------------------------------------------------
-- Component Declaration for the TEMAC wrapper with
-- Local Link FIFO.
-- Component Declaration for the TEMAC wrapper with
component MAC_wrapper_locallink is
  port(
    -- EMAC0 Clocking
    -- TX Client Clock output from EMAC0
    TX_CLIENT_CLK_OUT_0 : out std_logic;
    -- RX Client Clock output from EMAC0
    RX_CLIENT CLK_OUT_0 : out std_logic;
    -- TX PHY Clock output from EMAC0
    TX_PHY_CLK_OUT_0 : out std_logic;
    -- EMAC0 TX Client Clock input from BUFG
    TX_CLIENT CLK_0 : in std_logic;
    -- EMAC0 RX Client Clock input from BUFG
    RX_CLIENT_CLK_0 : in std_logic;
    -- EMAC0 TX PHY Clock input from BUFG
    TX_PHY_CLK_0 : in std_logic;
    -- Local link Receiver Interface - EMAC0
    RX_LL_CLOCK_0 : in std_logic;
    RX_LL_RESET_0 : in std_logic;
    RX_LL_DATA_0 : out std_logic_vector(7 downto 0);
    RX_LL_SOF_N_0 : out std_logic;
    RX_LL_EOF_N_0 : out std_logic;
    RX_LL_SRC_RDY_N_0 : out std_logic;
    RX_LL_DST_RDY_N_0 : in std_logic;
    RX_LL_FIFO_STATUS_0 : out std_logic_vector(3 downto 0);
    -- Local link Transmitter Interface - EMAC0
    TX_LL_CLOCK_0 : in std_logic;
    TX_LL_RESET_0 : in std_logic;
    TX_LL_DATA_0 : in std_logic_vector(7 downto 0);
    TX_LL_SOF_N_0 : in std_logic;
    TX_LL_EOF_N_0 : in std_logic;
    TX_LL_SRC_RDY_N_0 : in std_logic;
    TX_LL_DST_RDY_N_0 : out std_logic;
    -- Client Receiver Interface - EMAC0
    EMACOCLIENTRXDVLD : out std_logic;
  );

end MAC_wrapper_locallink;

50
EMAC0CLIENTRXFRAMEDROP : out std_logic;
EMAC0CLIENTRXSTATS : out std_logic_vector(6 downto 0);
EMAC0CLIENTRXSTATSVLD : out std_logic;
EMAC0CLIENTRXSTATSBYTEVLD : out std_logic;

-- Client Transmitter Interface - EMAC0
CLIENTEMACOTXIFGDELAY : in std_logic_vector(7 downto 0);
EMAC0CLIENTTTXSTATS : out std_logic;
EMAC0CLIENTTTXSTATSVLD : out std_logic;
EMAC0CLIENTTTXSTATSBYTEVLD : out std_logic;

-- MAC Control Interface - EMAC0
CLIENTEMACOPAUSEREQ : in std_logic;
CLIENTEMACOPAUSEVAL : in std_logic_vector(15 downto 0);

-- Clock Signals - EMAC0

-- MII Interface - EMAC0
MII_COL_0 : in std_logic;
MII_CRS_0 : in std_logic;
MII_TXD_0 : out std_logic_vector(3 downto 0);
MII_TX_EN_0 : out std_logic;
MII_TX_ER_0 : out std_logic;
MII_TX_CLK_0 : in std_logic;
MII_RXD_0 : in std_logic_vector(3 downto 0);
MII_RX_DV_0 : in std_logic;
MII_RX_E_R : in std_logic;
MII_RX_CLK_0 : in std_logic;

-- Asynchronous Reset
RESET : in std_logic
);
end component;

-- Signal Declarations

-- Global asynchronous reset
signal reset_i : std_logic;

-- signal current_col : natural := 0; -- Column counter within frame
-- client interface clocking signals - EMAC0
signal ll_clk_0_i : std_logic;

-- address swap transmitter connections - EMAC0
signal tx_ll_data_0_i : std_logic_vector(7 downto 0);
signal tx_ll_sof_n_0_i : std_logic;
signal tx_ll_eof_n_0_i : std_logic;
-- signal tx_ll_src_rdy_n_0_i : std_logic;
signal tx_ll_dst_rdy_n_0_i : std_logic;

-- address swap receiver connections - EMAC0
signal rx_ll_data_0_i : std_logic_vector(7 downto 0);
signal rx_ll_sof_n_0_i : std_logic;
signal rx_ll_eof_n_0_i : std_logic;
signal rx_ll_src_rdy_n_0_i : std_logic;
signal rx_ll_dst_rdy_n_0_i : std_logic;

-- create a synchronous reset in the transmitter clock domain
signal ll_pre_reset_0_i : std_logic_vector(5 downto 0);
signal ll_reset_0_i : std_logic;

attribute async_reg : string;
attribute async_reg of ll_pre_reset_0_i : signal is "true";

-- EMAC0 Clocking signals

-- MII input clocks from PHY
signal tx_phy_clk_0 : std_logic;
signal rx_clk_0_i : std_logic;

-- Client clocks at 1.25/12.5MHz
signal tx_client_clk_0 : std_logic;
signal rx_client_clk_0 : std_logic;
signal tx_client_clk_0_o : std_logic;
signal rx_client_clk_0_o : std_logic;

-------------------------------------------------------------------
-- types to support frame data
-------------------------------------------------------------------

-- Tx Data and Data_valid record
type data_typ is record
  data : bit_vector(7 downto 0); -- data
  valid : bit; -- data_valid
  error : bit; -- data_error
end record;
type frame_of_data_typ is array (natural range <>) of data_typ;

-- Tx Data, Data_valid and underrun record
type frame_typ is record
  columns : frame_of_data_typ(0 to 65);-- data field
  bad_frame : boolean; -- does this frame contain an error?
end record;
type frame_typ_ary is array (natural range <>) of frame_typ;

-------------------------------------------------------------------
-- Stimulus - Frame data
-------------------------------------------------------------------

52
-- The following constant holds the stimulus for the test. It is
-- an ordered array of frames, with frame 0 the first to be injected
-- into the core receiver PHY interface by the testbench.

constant frame_data : frame_typ_ary := (  
  --------------  
  -- Frame 0  
  --------------  
  0 => (  
      columns => (  
        0 => ( DATA => X"AA", VALID => '1', ERROR => '0'),  
        1 => ( DATA => X"BB", VALID => '1', ERROR => '0'),  
        2 => ( DATA => X"CC", VALID => '1', ERROR => '0'),  
        3 => ( DATA => X"DD", VALID => '1', ERROR => '0'),  
        4 => ( DATA => X"EE", VALID => '1', ERROR => '0'),  
        5 => ( DATA => X"FF", VALID => '1', ERROR => '0'),  
      )),  
  -- Source Address (5A)  
  6 => ( DATA => X"5A", VALID => '1', ERROR => '0'),  
  7 => ( DATA => X"02", VALID => '1', ERROR => '0'),  
  8 => ( DATA => X"03", VALID => '1', ERROR => '0'),  
  9 => ( DATA => X"04", VALID => '1', ERROR => '0'),  
 10 => ( DATA => X"05", VALID => '1', ERROR => '0'),  
 11 => ( DATA => X"06", VALID => '1', ERROR => '0'),  
 12 => ( DATA => X"00", VALID => '1', ERROR => '0'),  
  -- Length/Type = Length = 46  
  13 => ( DATA => X"2E", VALID => '1', ERROR => '0'),  
  14 => ( DATA => X"01", VALID => '1', ERROR => '0'),  
  15 => ( DATA => X"02", VALID => '1', ERROR => '0'),  
  16 => ( DATA => X"03", VALID => '1', ERROR => '0'),  
  17 => ( DATA => X"04", VALID => '1', ERROR => '0'),  
  18 => ( DATA => X"05", VALID => '1', ERROR => '0'),  
  19 => ( DATA => X"06", VALID => '1', ERROR => '0'),  
  20 => ( DATA => X"07", VALID => '1', ERROR => '0'),  
  21 => ( DATA => X"08", VALID => '1', ERROR => '0'),  
  22 => ( DATA => X"09", VALID => '1', ERROR => '0'),  
  23 => ( DATA => X"0A", VALID => '1', ERROR => '0'),  
  24 => ( DATA => X"0B", VALID => '1', ERROR => '0'),  
  25 => ( DATA => X"0C", VALID => '1', ERROR => '0'),  
  26 => ( DATA => X"0D", VALID => '1', ERROR => '0'),  
  27 => ( DATA => X"0E", VALID => '1', ERROR => '0'),  
  28 => ( DATA => X"0F", VALID => '1', ERROR => '0'),  
  29 => ( DATA => X"10", VALID => '1', ERROR => '0'),  
  30 => ( DATA => X"11", VALID => '1', ERROR => '0'),  
  31 => ( DATA => X"12", VALID => '1', ERROR => '0'),  
  32 => ( DATA => X"13", VALID => '1', ERROR => '0'),  
  33 => ( DATA => X"14", VALID => '1', ERROR => '0'),  
  34 => ( DATA => X"15", VALID => '1', ERROR => '0'),  
  35 => ( DATA => X"16", VALID => '1', ERROR => '0'),  
  36 => ( DATA => X"17", VALID => '1', ERROR => '0'),  
  37 => ( DATA => X"18", VALID => '1', ERROR => '0'),  
)
38 => ( DATA => X"19", VALID => '1', ERROR => '0'),
39 => ( DATA => X"1A", VALID => '1', ERROR => '0'),
40 => ( DATA => X"1B", VALID => '1', ERROR => '0'),
41 => ( DATA => X"1C", VALID => '1', ERROR => '0'),
42 => ( DATA => X"1D", VALID => '1', ERROR => '0'),
43 => ( DATA => X"1E", VALID => '1', ERROR => '0'),
44 => ( DATA => X"1F", VALID => '1', ERROR => '0'),
45 => ( DATA => X"20", VALID => '1', ERROR => '0'),
46 => ( DATA => X"21", VALID => '1', ERROR => '0'),
47 => ( DATA => X"22", VALID => '1', ERROR => '0'),
48 => ( DATA => X"23", VALID => '1', ERROR => '0'),
49 => ( DATA => X"24", VALID => '1', ERROR => '0'),
50 => ( DATA => X"25", VALID => '1', ERROR => '0'),
51 => ( DATA => X"26", VALID => '1', ERROR => '0'),
52 => ( DATA => X"27", VALID => '1', ERROR => '0'),
53 => ( DATA => X"28", VALID => '1', ERROR => '0'),
54 => ( DATA => X"29", VALID => '1', ERROR => '0'),
55 => ( DATA => X"2A", VALID => '1', ERROR => '0'),
56 => ( DATA => X"2B", VALID => '1', ERROR => '0'),
57 => ( DATA => X"2C", VALID => '1', ERROR => '0'),
58 => ( DATA => X"2D", VALID => '1', ERROR => '0'),

-- 46th Byte of Data
59 => ( DATA => X"2E", VALID => '0', ERROR => '0'),
others => ( DATA => X"00", VALID => '0', ERROR => '0')),

-- No error in this frame
bad_frame => false));

-- ------------------------------------------------- ---------------------
-- -- CRC engine
-- ----------------------------------------------------------------------
-- function calc_crc (data : in std_logic_vector;
-- fcs : in std_logic_vector)
-- return std_logic_vector is
-- 
-- variable crc : std_logic_vector(31 downto 0);
-- variable crc_feedback : std_logic;
-- begin
-- 
-- crc := not fcs;
-- 
-- for I in 0 to 7 loop
-- crc_feedback := xor data(I);
-- 
-- crc(4 downto 0) := crc(5 downto 1);
-- crc(5) := crc(6) xor crc_feedback;
-- crc(7 downto 6) := crc(8 downto 7);
-- crc(8) := crc(9) xor crc_feedback;
-- crc(9) := crc(10) xor crc_feedback;

54
-- crc(14 downto 10) := crc(15 downto 11);
-- crc(15) := crc(16) xor crc_feedback;
-- crc(18 downto 16) := crc(19 downto 17);
-- crc(19) := crc(20) xor crc_feedback;
-- crc(20) := crc(21) xorcrc_feedback;
-- crc(21) := crc(22) xor crc_feedback;
-- crc(22) := crc(23);
-- crc(23) := crc(24) xor crc_feedback;
-- crc(24) := crc(25) xor crc_feedback;
-- crc(25) := crc(26);
-- crc(26) := crc(27) xor crc_feedback;
-- crc(27) := crc(28) xor crc_feedback;
-- crc(28) := crc(29);
-- crc(29) := crc(30) xor crc_feedback;
-- crc(30) := crc(31) xor crc_feedback;
-- crc(31) := crc_feedback;
-- end loop;
--
-- return the CRC result
-- return not crc;
--
-- end calc_crc;

--============================================= signal tx_state : std_logic_vector( 1 downto 0 ); signal cnt : std_logic_vector( 5 downto 0 );
--=============================================----------------------------------------------------
-- Main Body of Code
----------------------------------------------------

begin

begin

-- Reset Input Buffer
reset_ibuf : IBUF port map (I => RESET, O => reset_i);

-- EMAC0 Clocking
-- Put the PHY clocks from the EMAC through BUFGs. Used to clock the PHY side of the EMAC wrappers.
bufg_phy_tx_0 : BUFG port map (I => MII_TX_CLK_0, O => tx_phy_clk_0);
bufg_phy_rx_0 : BUFG port map (I => MII_RX_CLK_0, O => rx_clk_0_i);

-- Put the client clocks from the EMAC through BUFGs. Used to clock the client side of the EMAC wrappers.
bufg_client_tx_0 :
BUFG port map (I => tx_client_clk_0_o, O => tx_client_clk_0);
bufg_client_rx_0 :
BUFG port map (I => rx_client_clk_0_o, O => rx_client_clk_0);

ll_clk_0_i <= tx_client_clk_0;

-----------------------------------------------------
-- Instantiate the EMAC Wrapper with LL FIFO
-- (MAC_wrapper_locallink.v)
-----------------------------------------------------

v5_emac_ll : MAC_wrapper_locallink
port map (  
  -- EMACO Clocking
  -- TX Client Clock output from EMACO
  TX_CLIENT_CLK_OUT_0 => tx_client_clk_0_o,  
  -- RX Client Clock output from EMACO
  RX_CLIENT_CLK_OUT_0 => rx_client_clk_0_o,  
  -- TX PHY Clock output from EMACO
  TX_PHY_CLK_OUT_0 => open,  
  -- EMACO TX Client Clock input from BUFG
  TX_CLIENT_CLK_0 => tx_client_clk_0,  
  -- EMACO RX Client Clock input from BUFG
  RX_CLIENT_CLK_0 => rx_client_clk_0,  
  -- EMACO TX PHY Clock input from BUFG
  TX_PHY_CLK_0 => tx_phy_clk_0,  
  -- Local link Receiver Interface - EMACO
  RX_LL_CLOCK_0 => ll_clk_0_i,  
  RX_LL_RESET_0 => ll_reset_0_i,  
  RX_LL_DATA_0 => rx_ll_data_0_i,  
  RX_LL_SOF_N_0 => rx_ll_sof_n_0_i,  
  RX_LL_EOF_N_0 => rx_ll_eof_n_0_i,  
  RX_LL_SRC_RDY_N_0 => rx_ll_src_rdy_n_0_i,  
  RX_LL_DST_RDY_N_0 => rx_ll_dst_rdy_n_0_i,  
  RX_LL_FIFO_STATUS_0 => open,  
  -- Unused Receiver signals - EMACO
  EMACOCLIENTRXDVLD => EMACOCLIENTRXDVLD,  
  EMACOCLIENTRXFRAMEDROP => EMACOCLIENTRXFRAMEDROP,  
  EMACOCLIENTRXSTATS => EMACOCLIENTRXSTATS,  
  EMACOCLIENTRXSTATSVLD => EMACOCLIENTRXSTATSVLD,  
  EMACOCLIENTRXSTATSBYTEVLD => EMACOCLIENTRXSTATSBYTEVLD,  
  -- Local link Transmitter Interface - EMACO
  TX_LL_CLOCK_0 => ll_clk_0_i,  
  TX_LL_RESET_0 => ll_reset_0_i,  
  TX_LL_DATA_0 => tx_ll_data_0_i,  
  TX_LL_SOF_N_0 => tx_ll_sof_n_0_i,  
  TX_LL_EOF_N_0 => tx_ll_eof_n_0_i,  
  TX_LL_SRC_RDY_N_0 => tx_ll_src_rdy_n_0_i,  
  TX_LL_DST_RDY_N_0 => tx_ll_dst_rdy_n_0_i,
-- Unused Transmitter signals - EMAC0
CLIENTEMACOTXIFGDELAY => (others => '0'),
EMACOCLIENTTXSTATS => EMACOCLIENTTXSTATS,
EMACOCLIENTTXSTATSVLD => EMACOCLIENTTXSTATSVLD,
EMACOCLIENTTXSTATSBYTEVLD => EMACOCLIENTTXSTATSBYTEVLD,

-- MAC Control Interface - EMAC0
CLIENTEMACOPAUSEREQ => '0',
CLIENTEMACOPAUSEVAL => (others => '0'),

-- Clock Signals - EMAC0
-- MII Interface - EMAC0
MII_COL_0 => '0',
MII_CRS_0 => '0',
MII_TXD_0 => MII_TXD_0,
MII_TX_EN_0 => MII_TX_EN_0,
MII_TX_ER_0 => MII_TX_ER_0,
MII_TX_CLK_0 => tx_phy_clk_0,
MII_RXD_0 => MII_RXD_0,
MII_RX_DV_0 => MII_RX_DV_0,
MII_RX_ER_0 => MII_RX_ER_0,
MII_RX_CLK_0 => rx_clk_0_i,

-- Asynchronous Reset
RESET => reset_i
);

-- Create synchronous reset in the transmitter clock domain.
gen_ll_reset_emac0 : process (ll_clk_0_i, reset_i)
begin
if reset_i = '1' then
  ll_pre_reset_0_i <= (others => '1');
  ll_reset_0_i <= '1';
elsif ll_clk_0_i'event and ll_clk_0_i = '1' then
  ll_pre_reset_0_i(0) <= '0';
  ll_pre_reset_0_i(5 downto 1) <= ll_pre_reset_0_i(4 downto 0);
  ll_reset_0_i <= ll_pre_reset_0_i(5);
end if;
end process gen_ll_reset_emac0;

-------------------------------------------------------------------------------
-- Simulus process
--------------
-- Send four frames through the MAC and Design Example
-- frame 0 = minimum length frame

57
-- p_stimulus : process (mii_tx_clk_0)
-- variable cnt : integer := 0;
-- variable current_col : natural := 0; -- Column counter within frame
-- constant preamble : std_logic_vector(7 downto 0) := "01010101";
---- variable idle_cnt : integer := 0;
---- variable send_low : bit := '1';
---- variable fcs : std_logic_vector(31 downto 0);
-- begin
--
-- Send frame at 100Mb/s speed
--

TX_FRAME_GEN : process(MII_TX_CLK_0)
constant preamble : std_logic_vector(7 downto 0) := "01010101";
begin
if rising_edge(MII_TX_CLK_0) then
if (reset_i = '1') then
    tx_state <= (others => '0');
    TX_LL_SOF_N_0_I <= '1';
    TX_LL_EOF_N_0_I <= '1';
    cnt <= (others => '0');
else
    case tx_state is
--
when "00" =>
    TX_LL_EOF_N_0_I <= '1';
    if (TX_LL_SRC_RDY_N_0_I = '0') then
        tx_ll_data_0_i <= preamble;
        TX_LL_SOF_N_0_I <= '0';
        cnt <= cnt + "000001";
        tx_state <= "01";
    end if;
--
when "01" =>
    TX_LL_SOF_N_0_I <= '1';
    if (cnt = "000111") then
        tx_state <= "10";
        tx_ll_data_0_i <= "11010101";
    end if;
    cnt <= cnt + "000001";
--
when "10" =>
    tx_ll_data_0_i <=
to_stdlogicvector(
        frame_data(0).columns(58

58
conv_integer(cnt)-8 ).data(7 downto 0));
if ( frame_data(0).columns(
    conv_integer(cnt)-8 ).valid = '1' ) then
    TX_LL_EOF_N_0_I <= '0';
    tx_state <= "00";
    cnt <= ( others => '0' );
else
    cnt <= cnt + "000001";
end if;
end when others =>
null;
end if;
end case;
end if;
end process;
end TOP_LEVEL;

D.2 User constraint file

CONFIG PART = 5vlx50tff1136-1;

#########################################
# BLOCK Level constraints
#########################################

# EMAC0 Clocking
# EMAC0 TX Client Clock input from BUFG
NET "TX_CLIENT_CLK_0" TNM_NET = "clk_client_tx0";
TIMEGRP "MAC_wrapper_client_clk_tx0" = "clk_client_tx0";
TIMESPEC "TS_MAC_wrapper_client_clk_tx0" =
PERIOD "MAC_wrapper_client_clk_tx0" 7700 ps HIGH 50 %;
# EMAC0 RX Client Clock input from BUFG
NET "RX_CLIENT_CLK_0" TNM_NET = "clk_client_rx0";
TIMEGRP "MAC_wrapper_client_clk_rx0" = "clk_client_rx0";
TIMESPEC "TS_MAC_wrapper_client_clk_rx0" =
PERIOD "MAC_wrapper_client_clk_rx0" 7500 ps HIGH 50 %;
# EMAC0 TX PHY Clock input from BUFG
NET "TX_PHY_CLK_0" TNM_NET = "clk_phy_tx0";
TIMEGRP "MAC_wrapper_phy_clk_tx0" = "clk_phy_tx0";
TIMESPEC "TS_MAC_wrapper_phy_clk_tx0" =
PERIOD "MAC_wrapper_phy_clk_tx0" 7700 ps HIGH 50 %;
NET "MII_RX_CLK_0" TNM_NET = "phy_clk_rx0";
TIMEGRP "MAC_wrapper_clk_phy_rx0" = "phy_clk_rx0";
TIMESPEC "TS_MAC_wrapper_clk_phy_rx0" = PERIOD "MAC_wrapper_clk_phy_rx0" 7500 ps HIGH 50 %;

# GMII Receiver Constraints: place flip-flops in IOB
INST "*mii0?RXD_TO_MAC*" IOB = true;
INST "*mii0?RX_DV_TO_MAC" IOB = true;
INST "*mii0?RX_ER_TO_MAC" IOB = true;
INST "*mii0?MII_TXD_?" IOB = true;
INST "*mii0?MII_TX_EN" IOB = true;
INST "*mii0?MII_TX_ER" IOB = true;

##################################
# LocalLink Level constraints
##################################

# EMAC0 LocalLink client FIFO constraints.
INST "*client_side_FIFO_emac0?tx_fifo_i?rd_tran_frame_tog"
TNM = "tx_fifo_rd_to_wr_0";
INST "*client_side_FIFO_emac0?tx_fifo_i?rd_retran_frame_tog"
TNM = "tx_fifo_rd_to_wr_0";
INST "*client_side_FIFO_emac0?tx_fifo_i?rd_col_window_pipe_1"
TNM = "tx_fifo_rd_to_wr_0";
INST "*client_side_FIFO_emac0?tx_fifo_i?rd_addr_txfer*"
TNM = "tx_fifo_rd_to_wr_0";
INST "*client_side_FIFO_emac0?tx_fifo_i?rd_txfer_tog"
TNM = "tx_fifo_rd_to_wr_0";
INST "*client_side_FIFO_emac0?tx_fifo_i?frame_in_fifo"
TNM = "tx_fifo_wr_to_rd_0";

TIMESPEC "TS_tx_fifo_rd_to_wr_0" = FROM "tx_fifo_rd_to_wr_0"
TO "MAC_wrapper_client_clk_tx0" 8000 ps DATAPATHONLY;
TIMESPEC "TS_tx_fifo_wr_to_rd_0" = FROM "tx_fifo_wr_to_rd_0"
TO "MAC_wrapper_client_clk_tx0" 8000 ps DATAPATHONLY;

# Reduce clock period to allow 3 ns for metastability settling time
INST "*client_side_FIFO_emac0?tx_fifo_i?wr_tran_frame_tog"
TNM = "tx_metastable_0";
INST "*client_side_FIFO_emac0?tx_fifo_i?wr_rd_addr*"
TNM = "tx_metastable_0";
INST "*client_side_FIFO_emac0?tx_fifo_i?wr_txfer_tog"
TNM = "tx_metastable_0";
INST "*client_side_FIFO_emac0?tx_fifo_i?frame_in_fifo"
TNM = "tx_metastable_0";

60
TIMESPEC "ts_tx_meta_protect_0" = FROM "tx_metastable_0" 5 ns DATAPATHONLY;

TIMESPEC "TS_tx_fifo_addr_0" = FROM "tx_addr_rd_0" TO "tx_addr_wr_0" 10ns;

## RX Client FIFO

# Group the clock crossing signals into timing groups
INST "*client_side_FIFO_emac0?rx_fifo_i?wr_store_frame_tog"
TNM = "rx_fifo_wr_to_rd_0";
INST "*client_side_FIFO_emac0?rx_fifo_i?rd_addr_gray"
TNM = "rx_fifo_rd_to_wr_0";
TIMESPEC "TS_rx_fifo_wr_to_rd_0" = FROM "rx_fifo_wr_to_rd_0"
TO "MAC_wrapper_client_clk_tx0" 8000 ps DATAPATHONLY;
TIMESPEC "TS_rx_fifo_rd_to_wr_0" = FROM "rx_fifo_rd_to_wr_0"
TO "MAC_wrapper_client_clk_rx0" 8000 ps DATAPATHONLY;

# Reduce clock period to allow for metastability settling time
INST "*client_side_FIFO_emac0?rx_fifo_i?wr_rd_addr_gray_sync"
TNM = "rx_metastable_0";
INST "*client_side_FIFO_emac0?rx_fifo_i?rd_store_frame_tog"
TNM = "rx_metastable_0";
TIMESPEC "ts_rx_meta_protect_0" = FROM "rx_metastable_0" 5 ns;

# MII Logic Standard Constraints
INST "mii_txd_0<0>" IOSTANDARD = LVTTL;
INST "mii_txd_0<1>" IOSTANDARD = LVTTL;
INST "mii_txd_0<2>" IOSTANDARD = LVTTL;
INST "mii_txd_0<3>" IOSTANDARD = LVTTL;
INST "mii_tx_en_0" IOSTANDARD = LVTTL;
INST "mii_tx_er_0" IOSTANDARD = LVTTL;
INST "mii_rxd_0<0>" IOSTANDARD = LVTTL;
# MII Logic Placement

INST "mii_rx_clk_0" LOC = "H17";
INST "mii_tx_clk_0" LOC = "K17";

INST "mii_txd_0<0>" LOC = "AF11";
INST "mii_txd_0<1>" LOC = "AE11";
INST "mii_txd_0<2>" LOC = "AH9";
INST "mii_txd_0<3>" LOC = "AH10";
INST "mii_tx_en_0" LOC = "AJ10";
INST "mii_tx_er_0" LOC = "AJ9";

INST "mii_rxd_0<0>" LOC = "A33";
INST "mii_rxd_0<1>" LOC = "B33";
INST "mii_rxd_0<2>" LOC = "C33";
INST "mii_rxd_0<3>" LOC = "C32";
INST "mii_rx_dv_0" LOC = "E32";
INST "mii_rx_er_0" LOC = "E33";

INST "mii_rxd_0<1>" IOSTANDARD = LVTTL;
INST "mii_rxd_0<2>" IOSTANDARD = LVTTL;
INST "mii_rxd_0<3>" IOSTANDARD = LVTTL;
INST "mii_rx_dv_0" IOSTANDARD = LVTTL;
INST "mii_rx_er_0" IOSTANDARD = LVTTL;

INST "mii_txd_0<0>" IOSTANDARD = LVTTL;
INST "mii_txd_0<1>" IOSTANDARD = LVTTL;
INST "mii_txd_0<2>" IOSTANDARD = LVTTL;
INST "mii_txd_0<3>" IOSTANDARD = LVTTL;

INST "mii_rx_clk_0" IOSTANDARD = LVTTL;
INST "mii_rx_clk_0" IOSTANDARD = LVTTL;
INST "mii_rx_clk_0" IOSTANDARD = LVTTL;
INST "mii_rx_clk_0" IOSTANDARD = LVTTL;
Bibliography


