Signal Averaging for Digitizer ADQ214
Abstract

Signal averaging is a signal processing technique applied in the time domain, intended to increase the strength of a signal relative to noise that is obscuring it. From a very long sequence of data, a number of smaller data sequences called records are collected. The form of averaging performed in this thesis was not among samples within a record, but among samples from different records. For example, let's say a sample \( x(n, k) \) which is a sample \( n \) from record \( k \), where \( 1 \leq n \leq N \) and \( N \) is the record size, and \( 1 \leq k \leq K \), where \( K \) is the total number of records it would perform the averaging.

Input signals for multi-record is periodic, typically repeated pulses. These records are stored in the memory of the Signal Processing (SP) Devices Digitizer ADQ214. Averaging is being implemented in two ways: software implementation and hardware implementation. In a software implementation the stored records are read out from a Digitizer to PC over a USB interface and averaging is performed in a PC with Matlab.

Averaging in a PC takes a significant amount of time because of reading out data through USB interface. The amount of records and number of samples per record play an important role in transferring a record from the Digitizer on board DDR memory to the PC through a USB interface. A large number of records and long record length increases the time to perform averaging. This limitation is removed by implementing averaging in hardware.

Verilog, a hardware description language is being used for designing the averaging unit in one of the Virtex5 FPGAs available on the Digitizer ADQ214. Performing averaging in hardware takes much less time than averaging in software. In a hardware implementation it is required to transfer data, which is the result in this case, only once from the Digitizer board to the PC regardless of the number of records under consideration.
Acknowledgements

The work presented in this thesis could not have been possible without the guidance and support of my supervisor Per Löwenborg at SP (Signal Processing) Devices. He has been an invaluable source of support and guidance throughout this thesis work. I am very grateful also to Martin Olsson, Anders Kagerin, Jan-Erik Eklund, Stefan Ahlqvist and Anders Forslund who have helped me with all those difficulties I encountered while performing this work at SP Devices. I would also like to thank Svante Signell for being my supervisor at the Royal Institute of Technology.
Contents

List of Figures ............................................................................................................................................. 6
List of Tables .................................................................................................................................................. 7

Introduction .................................................................................................................................................. 9
  1.1 Purpose .................................................................................................................................................. 9
  1.2 Background ......................................................................................................................................... 9
  1.3 Operating modes ................................................................................................................................... 10
    1.3.1 Trigger mode ............................................................................................................................... 10
    1.3.2 Multi-record mode ...................................................................................................................... 11
    1.3.3 Delay mode ..................................................................................................................................... 11

Averaging .................................................................................................................................................... 17
  2.1 Matlab implementation ....................................................................................................................... 17
    2.1.1 Overview ....................................................................................................................................... 17
    2.1.2 Functionality ............................................................................................................................... 17
    2.1.3 Input parameters for “interface_ADQ.m” .................................................................................. 18
    2.1.4 Output parameters for “interface_ADQ.m” ............................................................................. 19
    2.1.5 Results .......................................................................................................................................... 20
  2.2 HW implementation .............................................................................................................................. 25
    2.2.1 ADQ214 Architecture ................................................................................................................ 25
    2.2.2 Overview of proposed system ..................................................................................................... 26
    2.2.3 Architecture ................................................................................................................................... 27
    2.2.4 Memory units ............................................................................................................................... 27
    2.2.5 Simple averaging unit .................................................................................................................. 29
    2.2.6 Simple averaging block .............................................................................................................. 30
    2.2.7 Simple averaging block for dual channel ................................................................................... 34
    2.2.8 Triggers and data acquisition ....................................................................................................... 35
    2.2.9 Pre-trigger buffers ....................................................................................................................... 35
    2.2.10 SPI registers ............................................................................................................................... 38
    2.2.11 State machine .................................................................................................................................. 39
    2.2.12 Pipelining ....................................................................................................................................... 42
    2.2.13 Top level architecture ............................................................................................................... 46
    2.2.14 Averaging block with other components ................................................................................... 47
    2.2.15 C/C++ programming for Streaming Mode for Averaging ....................................................... 51
    2.2.16 Result ............................................................................................................................................ 55

Conclusion ..................................................................................................................................................... 59
References ..................................................................................................................................................... 61
List of Figures

Figure 1    External trigger states.......................................................................................... 10
Figure 2    Delay mode and level trigger..................................................................................11
Figure 3    Simple averaging of four records............................................................................12
Figure 4    Averaging with and without interpolation...............................................................15
Figure 5    Matlab and ADQ-API interface with ADQ214........................................................16
Figure 6    Averaging with NofRecords = 50............................................................................22
Figure 7    Averaging with NofRecords = 1000.......................................................................22
Figure 8    Averaging with NofRecords = 10,000................................................................. 23
Figure 9    Comparison between the typical input signal and the averaged signals...............24
Figure 10   Comparison between averaging with and without interpolation feature............24
Figure 11   Resolution comparison.........................................................................................25
Figure 12   Block diagram of the ADQ214..............................................................................26
Figure 13   Block diagram of a simple dual-port RAM.............................................................27
Figure 14   Read first mode......................................................................................................28
Figure 15   Block diagram of a simple averaging unit..............................................................29
Figure 16   Block diagram of a simple averaging block...........................................................31
Figure 17   Block diagram of skip first sample block..............................................................32
Figure 18   Sample skipping method.......................................................................................33
Figure 19   Block diagram of a dual channel simple averaging block.....................................34
Figure 20   Block diagram of a pre-trigger buffer................................................................. 36
Figure 21   Block diagram for a single channel pre-trigger buffer..........................................36
Figure 22   Block diagram for a pre-trigger buffer for dual channel.......................................37
Figure 23   Signal flow diagram of state machine................................................................. 39
Figure 24   Block diagram of state machine.......................................................................... 41
Figure 25   Timing issue for the pre-trigger buffer signals.................................................... 42
Figure 26   Pipelining for hold off mode...................................................................................43
Figure 27   Pipelining for pre-trigger mode.............................................................................44
Figure 28   Signals for read mode...........................................................................................45
Figure 29   Top level architecture of the complete averaging block......................................46
Figure 30   Level trigger block...............................................................................................47
Figure 31   Generation of Allow_level_trigger_reset_i in hardware.......................................48
Figure 32   Connection of Averaging block with other components......................................49
Figure 33   Connection diagram of ADQ214 with PC..............................................................50
Figure 34   Output from hardware implementation with 10,000 records............................ 55
Figure 35   Pre-trigger mode with pre-trigger samples = 10 and level trigger level = 100.....56
Figure 36   Holdoff mode with holdoff delay samples = 10 and level trigger level = 100......57
List of Tables

Table 1   SPI registers’ connections with configuration signals…………………………………  38
Table 2   Signal descriptions for signals in C/C++ program…………………………………….. 51
Table 3   Partial SPI registers ......................................................................................... 52
Table 4   Signals’ description connected to SPI registers……………………………………….. 53
Chapter 1

Introduction

This report discusses the thesis work for designing and implementing averaging both in software and hardware for SP (Signal Processing) Devices Digitizers. The primary focus is on the ADQ214 digitizer [1] from SP Devices.

1.1 Purpose

The purpose of this thesis work is to perform signal averaging both in software and hardware. For the software part a function in Matlab shall be written to perform the averaging using “multi-record mode” available for ADQ214. The results can be shown by plotting in Matlab. For the hardware part the architecture for the averaging circuit shall be designed in Verilog-HDL (Hardware Description Language) and verified through simulation. Then the design shall be implemented by programming one of the on board FGPAs available on an ADQ214. Final testing for the implemented hardware shall be performed to verify the proper functionalities of the design.

The averaging circuit shall be controlled by user defined inputs such as the number of records, number of samples per record, trigger types, delay types etc.

The hardware architecture for the averaging circuit has to be designed in such a way that it is compatible with the existing architecture of the digitizer ADQ214. This thesis work will start from designing, testing and verifying a simple averaging circuit to a complete product for averaging on ADQ214.

1.2 Background

SP Devices is in need of an averaging module to increase the strength of the signal of their product ADQ214. There are different averaging algorithms to choose for example: 'Moving average algorithm', or 'Savitzky-Golay algorithm' where an averaging is computed on a single data set i.e. a record. If the signal is periodic the averaging can be performed on the samples from several records.

In this thesis averaging will be performed of samples among different records and the input signal will be a periodic signal. In other words it will be record averaging for periodic signal. In order to avoid the loss of data precision no division will be performed in the hardware implementation. The hardware will implement recursive accumulation of samples among several records and the signal scaling by division will be performed in PC.
1.3 Operating modes

There are a number of modes and parameters which play important roles in averaging.

1.3.1 Trigger mode

There are three different types of triggers for ADQ214 [1] which can be used. These are (1) software trigger, (2) external trigger, and (3) level trigger. For averaging purposes ADQ214 can be triggered either by level trigger or external trigger. For level trigger mode the trigger type is set to 3 and a value for the level number at which the level trigger should be activated is assigned. For external trigger mode the trigger type is set to 2. So setting trigger type = 1 will activate software trigger mode, trigger type = 2 will activate external trigger mode and trigger type = 3 will activate level trigger mode. Level trigger mode and external trigger mode are shown in figure 2. Both external and level trigger mode work with pre-trigger delay and trigger holdoff mechanisms which are described in section 1.3.3. ADQ214 provides an extra feature for sampling an external trigger. An external trigger is sampled at four times higher frequency than the data sampling frequency [1]. So each trigger may have one of four states or phases [0 1 2 3]. This state or phase information is known as the trigger accuracy information because it provides more accurate position or phase information of the trigger. This state information can be used for averaging with higher resolution. Figure 1 shows the external trigger states. For software trigger mode the trigger type is set to 1 which is not used in this work.

![Figure 1: External trigger states.](image-url)
1.3.2 Multi-record mode

In this mode samples can be recorded in the memory of the ADQ214 every time the trigger is activated. The length of samples per record and the numbers of records are user defined with a constraint for example $N_{\text{of Samples Per Record}} \times N_{\text{of Records}} \leq 64\text{Msamples}$ \[1\].

1.3.3 Delay mode

Delay mode can be set to either pre-trigger samples delay or trigger hold off. Figure 2(a) and 2(b) illustrate these delay modes. The number of delayed samples can be set to a value with a constraint. For example for pre-trigger samples delay $N_{\text{of Delay Samples}} \leq N_{\text{of Samples Per Record}}$ and for trigger hold off $N_{\text{of Delay Samples}} \leq 2^{31}$ \[1\]. The trigger can be either external trigger or level trigger. Figure 2(c) shows level trigger mechanism. A level trigger is generated when the input signal passes a certain level of amplitude.

Figure 2: (a) Delay mode with delay holdoff = 3. (b) Delay mode with pre-trigger delay = 10. (c) Level trigger with trigger level = 6000.
1.3.4. Averaging mode

The averaging can be performed in two different modes: simple averaging and averaging with higher resolution. Averaging with higher resolution is achieved by interpolating the groups sorted by the trigger state numbers \([0 \ 1 \ 2 \ 3]\). In this mode the trigger must be in external trigger mode.

Simple averaging mode (without interpolating feature) is illustrated in figure 3. The averaging is performed among the corresponding samples of all records. Four records have been chosen for simplicity. The records are repetitive but they may arrive with a little phase difference. When simple averaging is performed, samples from different records are averaged but the phase differences those records might have are not taken into consideration. However this issue is taken into consideration during the averaging with interpolation feature.

For averaging with higher resolution feature it could be possible to get an averaging with up to four times higher resolution. In this case the records are sorted according to the states of the trigger. There are four different states of a trigger. Every time the trigger is activated, one of the state numbers \([0 \ 1 \ 2 \ 3]\) is valid which is recorded at the same time the record is recorded. The records are sorted out into different groups according to the state numbers \([0 \ 1 \ 2 \ 3]\). The averaging is performed within the corresponding samples of each group and then interpolated. Even though there are four states \([0 \ 1 \ 2 \ 3]\) of a trigger it can happen one, two or even three of these states are missing after the acquisition is complete. It depends on the number of records. If the number of records is small it can happen that some of the states of the trigger are missing. So when we have all of the four states available i.e.
none of the states is missing, we will have four groups and after interpolating those four groups we will get an output with four times higher resolution. If we have three states, we will have three groups and after interpolating those three groups we will get an output with three times higher resolution and so on.

**Example: #1**

NofRecords= 10  
Records r1, r7, r9 has state number 3  
Records r2, r5 has state number 2  
Records r3, r4, r6 has state number 0  
Records r8, r10 has state number 1

No state is missing. So it’s possible to achieve an interpolation by four. Averaging from each of these four groups are performed and interpolated according to the following order.

\[
\text{Interpolated}_\text{Av} =  
\begin{bmatrix}
\text{upsample(Av\_State0, 4)} & 0 & 0 & 0 \\
0 & \text{upsample(Av\_State1, 4)} & 0 & 0 \\
0 & 0 & \text{upsample(Av\_State2, 4)} & 0 \\
0 & 0 & 0 & \text{upsample(Av\_State3, 4)}
\end{bmatrix}
\]

Where  
Av\_State0 = the results from averaging samples among records r3, r4 and r6.  
Av\_State1 = the results from averaging samples among records r8 and r10.  
Av\_State2 = the results from averaging samples among records r2 and r5.  
Av\_State3 = the results from averaging samples among records r1, r7 and r9.

For simple averaging, the averaging is performed among the samples of all of the records. The number of sample positions in Interpolated\_Av in this case is four times the number of sample positions in the simple averaging result. If the number of samples in each record is 100, Interpolated\_Av will contain 400 sample positions in this case whereas the result from simple averaging will contain 100 sample positions. So depending on the number of the trigger states \([0 1 2 3]\) available while averaging, the number of interpolation will vary which in turn will vary the number of sample positions in Interpolated\_Av. Interpolated\_Av can have sample positions max. four times the number of samples per record.

**Example: #2**

NofRecords= 10  
Records r1, r7, r9 has state number 3  
Records r2, r5, r8 has state number 2  
Records r3, r4, r6, r10 has state number 0

Here state 1 is missing, so we will get an interpolation by three. If the number of samples per record is 100, the number of sample positions for Interpolated\_Av will be 300 in this case.
Figure 4 shows a comparison between simple averaging of 16 records with and without interpolating feature. Figure 4(a) shows the averaging without interpolating feature where as figure 4(b) shows the averaging with interpolating feature. In figure 4(a) for each sampling event we have one averaged sample. In figure 4(b) all the records are first sorted out in to groups according to the trigger state numbers. As described earlier, for each record a trigger state number is recorded. This is an example where all of the four trigger states are available. Among 16 records, trigger state = 0 has four records, trigger state = 1 has four records, trigger state = 2 has three records and trigger state = 3 has five records. Then averaging is performed within the corresponding samples of each group. Later the averaged values are interpolated. As we have the entire four trigger states available in this case we will achieve four times higher resolution of the output curve than we get in figure 4(a). For each averaged value in figure 4(a) we have four averaged values in figure 4(b). So the number of sample positions of averaged samples in figure 4(b) is four times higher than the number of sample positions of averaged samples in figure 4(a). This will lead to smoother and more precise curve for the averaging with interpolating feature which will be seen in section 2.1.5.
Figure 4: Simple averaging of 16 records. a) without interpolating feature. b) with interpolating feature.
1.3 ADQ-API and Matlab interface

ADQ-Application Programming Interface (API) [2] provides a simple and powerful programming interface to ADQ214. The programming interface written in C/C++ handles all communication with the connected ADQ214 device with just a few highly abstracted functions.

For the Matlab interface there is a file mex_ADQ.dll which serves as an interface between the low level device drivers ADQ-API and Matlab [2]. From Matlab when a script is run to perform any operation such as to get data from for example ADQ214, a function named “interface_ADQ214” is called with appropriate parameters. The function “interface_ADQ214” implements a set of functions and the selection of a particular function among the set of functions depends on the function’s arguments. The function “interface_ADQ214” communicates with ADQ-API with the help of mex_ADQ. Finally ADQ-API communicates with the device ADQ214. Matlab and ADQ-API interface with ADQ214 is illustrated in figure 5.

Figure 5: Block diagram of Matlab and ADQ-API interface with ADQ214.
Chapter 2

Averaging

A signal can be corrupted by thermal noise and quantization noise. These effects can be reduced by signal averaging. Averaging is performed among the samples from a set of records. Multi-record mode is useful in this case. Multi-record mode allows collecting and storing a finite numbers of records to perform the averaging. Focus is on ADQ214 which is a dual channel digitizer with 14-bit data in each channel. ADQ214 has an USB/PXI interface to connect to a PC or another device.

2.1 Matlab implementation

2.1.1 Overview

A function has been written in Matlab to perform the signal averaging. This function can be called from a Matlab script or from a function. According to the input parameters the function is set to a specific mode for performing the averaging. The function has the capability to check if the input parameters are correct for a particular mode. It uses the available function “interface_AQD214” to acquire data from ADQ214 for performing signal averaging.

2.1.2 Functionality

This function can collect a finite number of records and perform simple averaging and averaging with higher resolution by using a set of the functions from the available function “interface_ADQ”. This function is capable of calculating the average of a finite number of rounds. A round is defined as one iteration corresponding to averaging over all records. The number of rounds and records, record size, delay type and trigger type are user defined. It performs the averaging among a specified number of records and stores the results. In each round it calculates a new averaged wave from the specified number of records and then this is averaged with the stored averaged wave from the previous round and finally the new averaged wave is stored. By using one of the output parameters Status_ChX it’s possible to know how much higher resolution one can achieve from that event. The details of the input output parameters of “interface_ADQ.m” are described in the sections 2.1.3 and 2.1.4.
2.1.3 Input parameters for “interface_ADQ.m”

**Ch_No:** Selects the input Channel(s).
- 0 = No Channel
- 1 = Channel A
- 2 = Channel B
- 3 = Channel A and B

**Trigger_Type:** Selects the type of the trigger.
- 2 = Extern
- 3 = Level

**LevelTrigg_Level:** Defines the level at which the level trigger should be activated. Valid range is -8192 to +8191 (14 bit signed number).

**Delay_Type:** Defines the type of delay.
- 0 = Hold off
- 1 = Pre-trigger

**NofDelaySamples:** Defines the number of delay samples.
- For pre-trigger : NofDelaySamples <= NofSamplesPerRecord
- For hold off : NofDelaySamples <= 2^31

**NofSamplesPerRecord:** Defines the record size.

**NofRecords:** Defines the number of records.

**NofRounds:** Defines the number of rounds the averaging should be performed. For each round a number of records is collected, averaged and the result is stored. If NofRounds > 1, averaging is performed again between the stored result and the new result.

**Interpolation:** Defines whether the averaging is with higher resolution. A value 0 means no interpolation and 1 means interpolation.
2.1.4 Output parameters for “interface_ADQ.m”

**Average_Int_ChA:** Contains Interpolated data. If a particular state is missing the corresponding data value is zero here. Size of Average_Int_ChA = 4*NofSamplesPerRecord.

**Average_Int_ChB:** Same as for Chanel A above.

**Average_Actual_Int_ChA:** This matrix is calculated from Average_Int_ChA. If any of the states (trigger states 0 1 2 3) is missing, we skip the corresponding value (which is zero) from the matrix Average_Int_ChA and put the rest of the values to Average_Actual_Int_ChA.

Average_Actual_Int_ChA may contain 1, 2, 3 or 4 times interpolated values. Size of Average_Actual_Int_ChA = (1 or 2 or 3 or 4)*NofSamplesPerRecord.

**Average_Actual_Int_ChB:** Same as for Chanel A above.

**Status_ChA:** Contains the information whether all the four states has been hit or not. Size=4. [0 1 0 0] indicates that only state 1 has been hit. [1 1 0 1] indicates state 0, 1, 3 have been hit and state 2 has been missing and so on. Using the information from this matrix[x x x x], the Average_Actual_Int_ChA matrix is calculated.

**Status_ChB:** Same as for Chanel A above.

**Average_Reg_ChA:** Contains the regular average or simple average calculated by averaging the corresponding samples over all records.

**Average_Reg_ChB:** Same as for Chanel A above.

**Last_Batch_ChA:** Contains the last record for Chanel A.

**Last_Batch_ChB:** Contains the last record for Chanel B.

**Trig_State:** Contains the trigger states for all of the records.
2.1.5 Results

Here the input signal is a sinusoidal with noise. After performing signal averaging a more accurate signal is achieved. The more the number of records the better the averaged signal becomes. In figure 6-8 we see that as the number of records increases the resulting averaged signal becomes more and more close to the original sinusoidal signal. The green lines represent the input signals, the blue line represents the result from the simple averaging without interpolating feature and the red line represents the result with interpolating feature i.e. averaging with higher resolution.

In figure 9 it is seen how much better the averaged signal can be achieved after averaging 10,000 records. Figure 8 and 9 shows the same result but in figure 9 only one record is plotted to make it more clear. In figure 10 it is seen that a better and smoother signal is achieved from the averaging with higher resolution feature than the one without higher resolution feature. In this case the resulting signal from averaging with interpolating feature has four times higher resolution than the resulting signal without interpolating feature.

It is not always that the result from interpolated averaging will be better than the result from simple averaging i.e. averaging without interpolation. If we check the figure 6 will see that the result from simple averaging i.e. averaging without interpolation is better than the result from interpolated averaging. This is because the number of records is quite small here which is. Each sample in the resultant curve of the simple averaging is a result of averaging 50 samples from 50 different records. But for interpolated averaging it is not the case. 50 records have been sorted into different groups (max. 4 groups) depending on the trigger status. So each group will obviously get less than 50 records as 50 records have been divided among them and thus the averaged result in each group will not be as good as the simple averaging in this case. When the number of records is very small some groups might get a lot more records than others and thus not creating a better result after interpolating those groups.

If we check the figure 11 where the number of records is 10,000 we see the result from interpolated averaging is much smoother and more accurate than the result from without interpolation. The result from interpolated averaging has four times higher resolution than the result from without interpolation. For each sample position in simple averaging four sample positions are achieved after interpolation and thus more accurate information of the nature of the curve.

There is a delay in the averaged signal which is due to the filtering performed. It is a FIR filter.

\[
y(n) = \frac{1}{N} \sum_{k=0}^{N-1} x(n - k)
\]

\[
Y(z) = \frac{1}{N} \left(1 + z^{-1} + \ldots + z^{-N+1}\right)X(z)
\]

\[
= \frac{1}{N(z^{-N})} X(z)
\]

\[
|Y(q)| = e^{-j2\pi N q} \left| \frac{\sin(\pi N q)}{N\sin(\pi q)} \right| |X(q)|
\]

Sample delay = \(\frac{N}{2}\) samples
This delay will disappear in HW implementation as the filtering performed there is recursive filler.

\[
y(N) = \frac{1}{N} \sum_{k=0}^{N-1} x(n - k) \\
= \frac{1}{N} \cdot x(N - 1) + \frac{N - 1}{N} \cdot \frac{1}{N - 1} \cdot \sum_{k=0}^{N-2} x(n - k) \\
y(N) = \frac{1}{N} \cdot x(N - 1) + \frac{N - 1}{N} \cdot y(N - 1) \\
N y(N) = x(N - 1) + (N - 1) \cdot y(N - 1)
\]
Figure 6: Averaging with NofRecords=50.

Figure 7: Averaging with NofRecords =1000.
Figure 8: Averaging with NofRecords=10,000.

Figure 9: Comparison between the typical input signal and the averaged signals.
Figure 10: Comparison between Simple averaging with and without Interpolating feature.

Figure 11: Averaging with interpolation has four times higher resolution than simple averaging.
2.2 HW implementation

2.2.1 ADQ214 Architecture

ADQ214 is a dual channel digitizer. Two input channels ChA and ChB are connected to two ADCs. The outputs from the ADCs are connected to the first FPGA which is known as Algorithm FPGA. Most of the signal processing happens in this FPGA. Algorithm FPGA is connected to the second FPGA which is known as Communication FPGA [5]. This FPGA handles the communication tasks with the external devices. ADQ214 can be connected by USB or PXI interface to the external devices or PC [1]. The following figure shows a basic block diagram of the ADQ214.

![Figure 12: Simple block diagram of the ADQ214.](image)

A new block for averaging will be implemented on the Algorithm FPGA as shown in the figure above. Two ADCs are running at a clock speed of 400MHz [1]. Two FPGAs are running at a clock speed of 200MHz [5]. So for each clock cycle inside the Algorithm FPGA, it receives two samples from each ADC for each channel. In other words Algorithm FPGA receives two samples for each channel from ADC at every clock cycle.
2.2.2 Overview of proposed system

The hardware implementation for signal averaging is done using the hardware description language Verilog. The two FPGAs available on the ADQ214 are two Xilinx Virtex5 FPGAs. As mentioned earlier that the new averaging block will be implemented the Algorithm FPGA. Figure 13 shows a basic block diagram of the averaging block. This unit has seven inputs and three outputs.

Data_ChA_In and Data_ChB_In are two 32-bit data inputs from channel A and B. Each of these 32-bit data inputs contains two 16-bit samples. Data_Valid_In defines whether the incoming data is valid or not. UserRegister_In is 32-bit wide and has a depth = 4. UserRegister_In receives the signals coming from user registers. The information for example: the number of records, record length, delay type and the number of delays, trigger type, read/write commands are sent to the averaging unit through UserRegister_In. Trigger and Triggerz are two trigger inputs to the averaging unit coming from ADC. Data_ChA_Out and Data_ChB_Out are two 32-bit data outputs for channel A and B and these outputs are the results after accumulating the samples. Thus each 32-bit output is the result of accumulation of all of corresponding samples over all records and is equivalent to only one sample.
whereas each 32-bit input contains two 16-bit samples. Data_Valid_Out defines whether the output data are valid or not.

2.2.3 Architecture

A bottom-up approach is used to design the hardware architecture for the averaging unit. The design starts with a simple block and then move towards the final unit. The design uses a number of basic units such as adders, multiplexers, registers, comparators etc to create basic building blocks. For memory a RAM and a FIFO are used. The descriptions of different units and blocks are given in the following sections.

2.2.4 Memory units

Xilinx Block Memory Generator is being used to generate the memory for the design. The Xilinx LogiCORE™ IP Block Memory Generator core is an advanced memory constructor that generates area and performance-optimized memories using embedded block RAM resources in Xilinx FPGAs [8]. A 32-bit wide dual port RAM is generated.

![Figure 13: Block diagram of a simple dual-port RAM [8].](image-url)
The Simple Dual-port RAM provides two ports, A and B, as illustrated in Figure 13. Write access to the memory is allowed via port A, and read access is allowed via port B. There are different operating modes for this memory. The operating mode for each port determines the relationship between the write and read interfaces for that port. Port A and port B can be configured independently with any one of three write modes: Write First Mode, Read First Mode, or No Change Mode. Read First Mode has been used in this case. In Read First mode, data previously stored at the write address appears on the data output, while the input data is being stored in memory. This read-before-write behavior is illustrated in Figure 14.

Figure 14: Read first mode [8].
2.2.5 Simple averaging unit

This unit is the basic unit for averaging. The inputs, the outputs and the connections among different components inside this unit are shown in figure 15.

![Block diagram of a simple averaging unit](image)

Figure 15: Block diagram of a simple averaging unit.

All the inputs to the simple averaging unit are latched by registers. Each sample is 16 bits wide which is the Data_In signal. Data_In is sign extended and is fed as a 32-bit input to one of the inputs of the 32-bit adder. The adder adds this Data_In with 0 or with the output from the RAM depending on the signal First_Batch. A value 1 in First_Batch indicates the first batch and in this case all the samples are added with 0 and saved into the memory. A value 0 in First_Batch results in the addition of the current sample to the result from the previous accumulation in other words the output from the RAM. By increasing the data width of the accumulator we can perform a number of accumulations without any data overflow. Here a 32-bit wide RAM is chosen for this purpose. Each row of the RAM is 32 bits wide and stores a sample. Each incoming sample is actually 16-bit wide.
and so the accumulation unit has 16 extra bits. Using these extra 16 bits it is possible to perform at least \(2^{16} = 65535\) accumulation without any data overflow. As it takes two clock cycles for the RAM to read out memory content after writing the address to Address_Read port of the RAM, Data_Valid_Out and Data_Out are synchronized accordingly as shown in the figure 14. An extra register Tex has been inserted in the path between the data output of the RAM and data input to the RAM to meet the timing requirement. This results in inserting one additional register for Data_In, Data_Valid_In, Addr_Sample, and First_Batch as shown in Figure 15.

2.2.6 Simple averaging block

The sampling clock is running at 400 MHz in ADC. But the clock inside the FPGA is running at 200 MHz. So for each clock cycle inside the FPGA we receive two 16-bit samples from the ADC unit. In order to handle two 16-bit samples at each clock cycle inside the FPGA we need two simple averaging units. Using two such simple averaging units a new unit named simple averaging block is created as illustrated in figure 16. A 32-bit data from ADC is split into two 16-bit data and sent to these two simple averaging units. The lower 16 bits of 32-bit ADC data are sent to Data_In_1 for instance#1 of the simple averaging unit. The higher 16 bits of 32-bit ADC data are sent to Data_In_2 for instance#2 of the simple averaging unit. Here Data_In_1 will thus receive the first sample and Data_In_2 will receive the second sample.

If we count the samples from 0 the instance#1 will receive even numbered samples for example sample no. 0, 2, 4, 6 and so on. On the other hand instance#2 will receive odd numbered samples for example sample no. 1, 3, 5, 7 and so on. Signal Control_Output selects the outputs from instance#1 and instance#2 and sends to Data_Out and Data_Valid_Out. Control_Output changes its value in every clock cycle. If the value of the Control_Output is 0 at the current clock cycle, the value will be 1 in the next clock cycle. A value 0 will let the outputs from the instance#1 to pass. A value 1 will let the outputs from instance#2 to pass. Thus we get the first sample from the instance#1 and then second sample from the instance#2, then the third sample from the instance#1 and the fourth sample from instance#2 and so on.

When we start an acquisition we get two samples at each clock cycle from ADC. A special care has to be taken before using the first two samples from ADC. There are two trigger signals which tell us which of these two samples the first sample is. For example if we receive sample0 and sample1 from ADC and from the trigger signals we know that sample0 is the first sample then we don’t need to do anything. We can simply use these samples to our averaging unit. We send sample0 to instance#1 and sample1 to instance#2. In the next clock cycle we send sample2 to instance#1 and sample3 to instance#2 and so on. But when we know that sample1 is the first sample then we need to skip the sample0. Thus we send sample1 to instance#1 and sample2 to instance#2. In the next clock cycle we send sample3 to instance#1 and sample4 to instance#2 and so on. In order to handle this sample skip issue we have a block named skip first sample as shown in the figure 16 and 17.
Simple averaging block is a complete logic block for each channel. Two such blocks will be instantiated for data channel A and data channel B as ADQ214 is a dual channel digitizer. This will be described later in section 2.2.7.

In skip first sample block the signal Skip_First_Sample determines whether we need to skip the first sample or not. A value 1 indicates to skip the first sample and 0 indicates no skipping. To skip the first sample we delay the signals Addr_Sample, First_Batch, Data_Valid for the instance#2 and at the same time we switch the data before we send to the simple averaging units (instance#1 and
instance#2). This because we always want to send the first valid sample to instance#1 and the second to the instance#2 and so on.

Figure 17: Block diagram of skip first sample block.

When we have to skip the first sample sample0, we switch the data so that instance#1 receives the sample1 (first valid sample in this case) and instance#2 receives sample0 (which has to be skipped). Skipping is done by delaying the Data_Valid, Addr_Sample, First_Batch signals to instance#2 which means during this clock cycle even instance#2 receives sample0 due to the fact that Data_Valid is 0 it does not process this data.
Figure 17 illustrates how the sample skipping actually works. At each clock cycle we receive two samples from ADC. We assume that the number of samples to be processed inside the averaging unit is 6. In figure 18(a), the trigger indicates that the first sample is sample0. So there is no need to skip any sample. Valid samples are Sample0, Sample1, Sample2, Sample3, Sample4 and Sample5. Sample0, Sample2, Sample4 will be sent to the instance#1 of simple averaging unit and Sample1, Sample3, Sample5 will be sent to the instance#2 of simple averaging unit. In figure 18(b), the trigger indicates that the first sample is Sample1 and so we have to skip the sample Sample0. Valid samples are Sample1, Sample2, Sample3, Sample4, Sample5 and Sample6. We always send the first sample to the instance#1 of simple averaging unit and the second sample to the instance#2 of simple averaging unit. So in this case we have to switch the samples so that Sample1, Sample3, Sample5 are sent to the instance#1 of simple averaging unit and Sample2, Sample4, Sample6 are sent to the instance#2 of simple averaging unit. Signals Addr_Sample, First_Batch, Data_Valid to the instance#2 of simple averaging unit are delayed by one clock cycle. So the instance#2 of simple averaging unit doesn’t process Sample0 but Sample6 instead which is what we want here.
2.2.7 Simple averaging block for dual channel

Simple averaging block as shown in figure 16 is a complete block for processing data from a channel. As ADQ214 is a dual channel digitizer we will need two such simple averaging blocks. The following figure shows a basic block diagram for a dual channel averaging block. The control signals are same for both ChA and ChB. Only data is different for ChA and ChB. The outputs are latched to get better timing.

Figure 19: Block diagram of a dual channel simple averaging block.
2.2.8 Triggers and data acquisition

We start data acquisition when a trigger is detected and we start sending data to our dual channel averaging block. Every time the trigger is detected a record is collected or processed. For ADQ214 a trigger can be software trigger, level trigger or external trigger [1]. For averaging purposes we will use only level trigger and external trigger. The Matlab implementation part has been done using level trigger and external trigger. In that case we performed averaging in PC by collecting data from ADQ214 through USB interface. If we look at the block diagram of the current AQ214 in figure 12 we will see that the Algorithm FPGA does not have the information of external trigger. The external trigger port is connected to the Communication FPGA. When we perform averaging in PC we have the facility to get the information of trigger for each record from the Communication FPGA. But as we know that we are going to implement the dual channel averaging block inside the Algorithm FPGA we will not have the external trigger to use at this moment. So we will use only level trigger for data acquisition for averaging inside the Algorithm FPGA. The next generation ADQ214 board will have the external trigger port connected to the Algorithm FPGA. The same design for dual channel averaging will be used but instead of only level trigger we will be able to feed either level trigger or external trigger as a trigger input for data acquisition.

Regardless of what type of trigger we use for data acquisition we always see the data from the time a trigger is detected. So in this case we don’t have any clue about the nature of data before the trigger event. In some applications it might be necessary to see the data just before the trigger event occurs and for that we will have to save the data samples in advance before a trigger event occurs. We will use buffers to save those pre-trigger samples. The next section describes this feature in details.

2.2.9 Pre-trigger buffers

In order to have available all of the pre-trigger samples we have to save those samples in some memory or buffers. We will use Xilinx LogiCORE IP FIFO Generator to generate FIFO for saving our pre-trigger samples. FIFO generator supports depths up to 4,194,304 words and data widths from 1 to 1024 bits. In our design we will generate a FIFO which has 16-bit data width and a depth of 1024 words. As we receive two samples from ADC for each channel we will need two of such FIFOs for saving data samples for each channel. For two channels we will need a total of four such FIFOs. With two such FIFOs for each channel we will be able to save 1024*2 = 2048 pre-samples for each channel. Figure 20 shows a basic block diagram of such FIFO. We will use the name pre-trigger buffer from now on to mention this FIFO for saving pre-trigger samples.

Even if we have the choice to save 1024 pre-samples in each pre-trigger buffer and a total of 2048 pre-samples for each channel we may want to save fewer amounts of pre-samples. We define that amount by setting a value to Prog_Full_Thresh_PreTrigg. When the buffer is filled with the number of pre-samples equal to the value of Prog_Full_Thresh_PreTrigg the signal Prog_Full_Out will go high indicating that all pre-samples have been collected. Signal Wr_En_PreTrigg enables the buffer to write data, Rd_En_PreTrigg enables to read out data from the buffer, Reset_PreTrigg resets the content of the buffers and initializes the pointers. Signal Empty_Out and Full_Out will not be used for averaging purposes because one will never reach a situation when one would have to check theses two signals and these signals indicate that the buffer is empty and full respectively.
As we get two samples from ADC for each channel at each clock cycle inside the FPGA we need two pre-trigger buffers as shown in the figure 21. As we see in this figure the only necessary outputs from the pre-trigger buffers are Data_Out and Prog_Full_Out. So the other outputs from the FIFOs are ignored.
Two FIFOs receive two samples at each clock cycle. All the control signals are same for both of the FIFOs. The only difference is data signals. As each of these can store 1024 pre-samples we will be able to store 2048 pre-samples. As we have two channels we will need two of such pre-trigger buffers. Figure 22 shows a basic block diagram of pre-trigger buffers which can save two pre-samples from each channel, i.e. a total of four pre-samples for two channels at each cycle.

The following block is a complete pre-trigger buffer for dual channels needed for averaging purpose. Data_Out directly goes to the averaging unit. Prog_Full_Out goes to the state machine and the state machine generates the signals Reset_PreTrigg, Wr_En_PreTrigg, Rd_En_PreTrigg and Prog_Full_Thresh_PreTrigg for the pre-trigger buffer for dual channels.

![Figure 22: Block diagram of pre-trigger buffer for dual channels.](image-url)
2.2.10 SPI registers

We need to configure and send different commands to different units for getting the desired operations. We send the commands from API through some registers in FPGA known as SPI registers. We send the values from SPI registers to the user registers UserRegister in averaging blocks and to other processing units for example level trigger block. A number of new registers have been added to the previous SPI registers. The detail of the SPI registers used for averaging purpose is shown in the following table.

Table 1: SPI registers' connections with configuration signals.

<table>
<thead>
<tr>
<th>SPI Address : {15'h32,33} == decimal {50 51}</th>
<th>UserRegister_in[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserRegister_in[0][0]</td>
<td>Pre-sampels / #Holdoff</td>
</tr>
<tr>
<td>UserRegister_in[0][1]</td>
<td>ARMED / #Reset</td>
</tr>
<tr>
<td>UserRegister_in[0][2]</td>
<td>Read / #Write</td>
</tr>
<tr>
<td>UserRegister_in[0][3]</td>
<td>unused</td>
</tr>
<tr>
<td>UserRegister_in[0][4]</td>
<td>unused</td>
</tr>
<tr>
<td>UserRegister_in[0][5]</td>
<td>leveltrig_cha_en</td>
</tr>
<tr>
<td>UserRegister_in[0][6]</td>
<td>leveltrig_chb_en</td>
</tr>
<tr>
<td>UserRegister_in[0][7]</td>
<td>leveltrig_posedge</td>
</tr>
<tr>
<td>UserRegister_in[0][8]</td>
<td>leveltrig_data_valid</td>
</tr>
<tr>
<td>UserRegister_in[0][9]</td>
<td>unused</td>
</tr>
<tr>
<td>UserRegister_in[0][31]</td>
<td>unused</td>
</tr>
<tr>
<td>SPI Address : {15'h34,35} == decimal {52,53}</td>
<td>#of Holdoff delays or #of Pre-sampels</td>
</tr>
<tr>
<td>UserRegister_in[1]</td>
<td></td>
</tr>
<tr>
<td>SPI Address : {15'h36,37} == decimal {54,55}</td>
<td>#of Samples</td>
</tr>
<tr>
<td>UserRegister_in[2]</td>
<td></td>
</tr>
<tr>
<td>SPI Address : {15'h38,39} == decimal {56,57}</td>
<td>#of Batches</td>
</tr>
<tr>
<td>UserRegister_in[3]</td>
<td></td>
</tr>
<tr>
<td>SPI Address : {15'h3A,3B} == decimal {58,59}</td>
<td>Contains the value for TrigMask1_i</td>
</tr>
<tr>
<td>SPI Address : {15'h3C,3D} == decimal {60,61}</td>
<td>Contains the value for TrigLevel1_i</td>
</tr>
<tr>
<td>SPI Address : {15'h3E,3F} == decimal {62,63}</td>
<td>Contains the value for TrigPreLevel1_i</td>
</tr>
<tr>
<td>SPI Address : {15'h40,41} == decimal {64,65}</td>
<td>Contains the value for TrigCompareMask1_i</td>
</tr>
<tr>
<td>SPI Address : {15'h42,43} == decimal {66,67}</td>
<td>Contains the value for TrigMask2_i</td>
</tr>
<tr>
<td>SPI Address : {15'h44,45} == decimal {68,69}</td>
<td>Contains the value for TrigLevel2_i</td>
</tr>
<tr>
<td>SPI Address : {15'h46,47} == decimal {70,71}</td>
<td>Contains the value for TrigPreLevel2_i</td>
</tr>
<tr>
<td>SPI Address : {15'h48,49} == decimal {72,73}</td>
<td>Contains the value for TrigCompareMask2_i</td>
</tr>
<tr>
<td>SPI Address : {15'h4A,4B} == decimal {74,75}</td>
<td>Unused</td>
</tr>
</tbody>
</table>

38
2.2.11 State machine

The state machine generates the appropriate control signals and addresses which will be sent to the simple averaging block for desired operations. Figure 23 illustrates the basic flow diagram of the state machine.

![State machine diagram]

There are nine different states to generate all of the control signals as well as addresses. These states are: Wait_trig, Hold_Off, Collect, Collect_last, Fillup_presamples, Wait_pretrigger, Collect_pretrigger, Read_one and Read_two. The state machine starts with checking armed/#reset signal. From API we send a number of information items such as the number of records, the number of samples per record, whether it is a read/write operation, whether state machine should be armed/rest etc. to the averaging unit and some other units. We send this information to the desired destinations through SPI registers available on the FPGA as described in section 2.2.10. The registers
which receive this information from SPI registers are here called UserRegister_in. There are four UserRegister_in and each of them is 32-bit wide. UserRegister_in[0][1] in the state machine determines whether it is reset or armed. If the value is 0 it indicates reset otherwise it is armed. When it is in reset mode it sets a number of signals to zero. When it is in armed mode it will go to the state Wait_trig. First it will check whether it is going to be a read or write operation.

UserRegister_in[0][2] determines whether it is a read or write command. A value 1 indicates a read operation and it will go to the state Read_one activating and/or deactivating some signals. State Read_one sends the signals to read out the even numbered samples stored in the instance#1 of the simple averaging block. State Read_two sends the signals to read out the odd numbered samples stored in the instance#2 of the simple averaging block. Each sample coming out of the simple averaging unit is 32-bit wide. When the read operation is done it stays in the state Read_one without sending any read signals and waits for an armed/reset signal for a new operation.

A value 0 in UserRegister_in[0][2] indicates a write operation. When it is a write operation it first checks if it is in holdoff/pre-trigger mode. A value 0 in UserRegister_in[0][0] determines holdoff mode. A value 1 in UserRegister_in[0][0] determines pre-trigger mode. When it is in holdoff mode it first checks if there is any trigger event and if the number of collected batches are less than or equal to the total number of batches i.e. batch <= NofBatches. If both conditions are true it goes to Hold_off state otherwise it stays in the same state Wait_trig waiting for a new trigger. Both conditions: trigger event and batch <= NofBatches has to be satisfied to switch to Hold_off state.

State Hold_off counts the number of delay cycles i.e. the amount of time it will be in hold mode before it starts collecting samples and after that amount of delay cycles it goes to the state Collect. In state Collect it generates the appropriate signals to collect the samples and accumulate those samples in to the memory of simple averaging block. Both states Hold_off and Collect generate signals and move forward if and only if the data valid is 1. If data valid is 0 it stays in the same state whether it is a Hold_off or Collect state without generating any further signals and waits for the data valid to be 1. State Collect_last is a state where the First_batch signal is updated after the state Collect has collected a batch. After updating the value of First_batch the next state becomes the state Wait_trig and moves forward depending on the status of the signals of the state machine. A simple input output diagram for the state machine is shown in Figure 24.

When it is in pre-trigger mode the state machine goes to state_fillup_presamples in the next clock cycle. Before it goes to fillup_presamples it resets the pre-trigger buffers and asserts other signals for example the number of pre-samples to be collected, number of records, and number of samples per record. As shown in figure 20, signal Prog_full_thresh_pretrigg determines the amount of pre-trigger samples to be collected and so this signal sets the pre-trigger buffer size in the pre-trigger buffers. When the buffers are full with this amount of pre-trigger samples it sets the prog_full_out_fifo to 1 letting us know that the desired number of pre-samples have been collected. Signal prog_full_out_fifo is 0 until pre-trigger buffers are filled with all pre-trigger samples.

There is a pipelining issue here when reading prog_full_out_fifo and making a particular decision for that. It takes 2 cycles before the state machine sees the actual value of prog_full_out_fifo i.e. when prog_full_out_fifo goes from 0 to 1, the state machine sees after two clock cycle. To handle this issue we need some sort of mechanism which is described in section 2.2.12. State fillup_presamples fills up the pre-trigger buffer with the desired amount of pre-samples. When all pre-samples are collected it checks if there is any trigger event or not. If a trigger event is detected in this cycle it will go to the collect_pretrigg_mode and start collecting samples from pre-trigger
buffers and sending them to the simple averaging unit. If a trigger event is not detected when it detects that all pre-samples have been collected, it will go to wait_presamples_trig state and wait for a trigger event. While waiting we read out a sample from pre-trigger buffer and throw it away as it is not a valid data. Data from pre-buffer is marked as valid when we detect a trigger event. Until then we write new data into the buffers, read out old data and throw it away. When we detect a trigger event inside the state wait_presamples_trig we go to the state collect_pretrigg_mode and start sending data to simple averaging unit. When collecting a complete record is done the state is switched to the state wait_presamps_trig to be ready for next record as shown in figure 23.

![State Machine Diagram]

Figure 24: Block diagram of state machine.

The state machine generates control signals for three different operating modes: hold off, pre-trigger and read. The signal Collect indicates that now the state is in Collect state. When Collect is 1 Data_valid_out for the hold off mode is assigned to data_valid signal of the averaging block. When Collect is 0, a value 0 is assigned to data_valid signal of the averaging unit. A multiplexer is used to handle this which will be shown in figure 26. We also use multiplexer in between the state machine and the simple averaging blocks to send the appropriate signals to the simple averaging block which will be shown in figure 29.
2.2.12 Pipelining

Now we have all of the necessary modules for averaging and these are state machine, pre-trigger buffers and simple averaging block. We need to connect them appropriately. Special care has to be taken when connecting state machine with other blocks for example pre-trigger buffers because both of these modules are sending signals to each other. Thus we need to synchronize signals between them properly. Both for pre-trigger and hold off mode will have different stages of pipelining for control signals and data signals.

When the operation is in pre-trigger mode, the state machine will receive prog_full_out_fifo from pre-trigger buffers which lets the state machine know that all pre-samples have been collected in the pre-trigger buffers. The amount of pre-samples is fixed by the signal prog_full_thresh_pretrigg which is sent to the pre-trigger buffers by the state machine.

Let us consider the above figure for one pre-trigger buffer. Here the number of pre-samples is 6. When we enter to the state fill_up_presamples, write enable is set to 1 forever i.e. a new sample will be written in the pre-trigger buffer at each clock cycle. When all of the 6 samples have been scheduled to be written into the pre-trigger buffer, at cycle 7 the state machine needs to check if there is any trigger event or not. If there is a trigger event at cycle 7, the state machine will send a signal to pre-trigger buffer to read out the first sample, send appropriate control signals to the simple averaging block to receive the output data from the pre-trigger buffer and go to state collect_pre-trigger for sending the rest of the samples to the simple averaging block. If no trigger event is detected the state machine will send a signal to the pre-trigger buffer to read out a sample but it will not send any control signal to the simple averaging block to receive this sample and then it will go to the state wait_pre-trigger. So it is very important for the state machine to know that all of the pre-samples have been collected at cycle 7 by checking the signal prog_full_out_fifo. But the fact is that the state machine never sees this value at cycle 7. For example when we schedule to write the last pre-sample into the pre-trigger buffer at cycle 6, the pre-trigger buffer will receive this signal at cycle 7 and then the write operation will be done at cycle 8. While writing a sample at cycle 8, pre-trigger buffer detects that the number of samples it has written is equal to the thresh hold value which is in this case 6 and so it will assert the signal prog_full_out_fifo indicating that all pre-samples have been collected. Thus the state machine will see this value at cycle 9. But the state machine must see this value at cycle 7. If we assign the thresh hold value to “prog_full_thresh_pretrigg -1” the state machine will see the correct value of prog_full_out_fifo at cycle 8. But if we assign the thresh hold

![Figure 25: Timing issue for the pre-trigger buffer signals.](image-url)
value to “prog_full_thresh_pretrigg -2” instead the state machine will see the correct value of
prog_full_out_fifo at cycle 7 which is required. So when we want to have 6 pre-samples we assign 4
to prog_full_thresh_pretrigg. It is to be noted that even though we set the threshold value to 4, by the
time we come to cycle 7 and detects the prog_full_out_fifo is 1 the 6 pre-samples have already been
scheduled to be written into the pre-trigger buffer by this time. So the basic if we want to have X
number of pre-samples we assign X-2 to prog_full_thresh_pretrigg. We do this inside the state
machine before we send the threshold value to prog_full_thresh_pretrigg.

During the write operation either of the two modes: pre-trigger mode or hold off mode is activated at
a time. For both modes the state machine generates the same signals for averaging block but in
different names. This is because the pipelining stages for signals varies for these two modes in other
words signals go through different level of pipelining stages before they are feed to the averaging
block and the level of pipelining stages for both modes are not same. We will simply use a
multiplexer to choose signals from either of these two modes when we are writing into the averaging
block.

During the hold off mode we have pipelined everything in such a way that we could start data
acquisition inside the averaging block with 0 - (2^31-1) hold off delays. By 0 we mean no delay at
all. The figure above illustrates the pipelining stages of signals for the hold off mode. Trigger inputs
(trigg and triggz) and user register inputs (UserRegister) remain the same always.
During the pre-trigger mode all control signals from state machine are latched before sending to the averaging block. This is because it takes one clock cycle for pre-trigger buffers to read out one sample. Figure 27 illustrates the pipelining stages of signals for the pre-trigger mode.

Figure 27: Pipelining for pre-trigger mode.
During the Read mode the control signals for read operation goes directly to the averaging unit. Figure 28 illustrates the signals for read mode. The data_valid signal from state machine to the averaging block is always zero.

Figure 28: Signals for read mode.
2.2.13 Top level architecture

The top level architecture of the averaging is designed by connecting the state machine with the averaging block. As the state machine generates signals differently in three different modes (hold off, pre-trigger and read) we need to multiplex those signals appropriately before sending to the averaging unit.

![Diagram of Top level architecture of the complete averaging block.](image)

Figure 29: Top level architecture of the complete averaging block.

Figure 29 above illustrates the connections of the signals from different modes. Two multiplexers are used here. One of the multiplexers selects all of the signals to the averaging block from either pre-trigger mode or hold off mode. The other multiplexer selects Address_sample from either read or write operation and send it to the averaging block. Hold off mode and pre-trigger mode belong to write operation.
2.2.14 Averaging block with other components

Now we need to connect the top level averaging block with other existing components inside the algorithm FPGA. As mentioned earlier, the current ADQ214 board doesn’t have the external trigger port connected to the algorithm FPGA. So we will have access only to level trigger inside the algorithm FPGA. We will use the available level trigger block to generate the triggers and then send those trigger signals to the averaging block. Figure 30 illustrates the block diagram of the Level Trigger Block available today.

Level trigger block needs to be configured first. A number of signals are used for configuring it and these signals come from API. The signal Allow_level_trigger_reset_i needs to be toggled every time we want the level trigger block to be ready to generate the trigger vector Level_trigger_vector_o. Sending signals from API is a very slow process. So for toggling the signal Allow_level_trigger_reset_i from API is a bottleneck for generating the trigger signals (trigger vector). Averaging block has an operating speed up to 100MHz that mean the averaging block can accept triggers and new record at a speed of 100MHz [5]. But toggling the signal Allow_level_trigger_reset_i from API and generating triggers will decrease the performance drastically. So we have decided to generate the toggling signal Allow_level_trigger_reset_i in hardware instead. A state machine is designed to generate the Allow_level_trigger_reset_i. Figure 31 illustrates this new state machine for Allow_level_trigger_reset_i. There are two states State_0 and State_1. The reset signal rst_i comes from the top level of algorithm FPGA which is known as adq_alg_top. The same reset signal goes to the level trigger block. For simplicity only the relevant signals are shown in the figure. Level trigger block has a lot more in signals but it receives only one signal Allow_level_trigger_reset_i from the new state machine.
When the state machine switches from reset to armed mode, it switches to Send_0 state and sends ‘0’ to the Allow_level_trigger_reset_i. In the next clock cycle the state switches to State_1 and sends ‘1’ to the Allow_level_trigger_reset_i. When level trigger block receives ‘0’ followed by ‘1’ it becomes ready to generate a new trigger vector. The state machine waits in the state State_1 until it detects a trigger on the trigger vector. As soon as it detects a trigger, it sends ‘0’ to the Allow_level_trigger_reset_i and switches to the state State_0. In the next clock cycle it sends ‘1’ to the Allow_level_trigger_reset_i and switches to State_1 and wait for a trigger at the trigger output LevelTrigg_vector_piped_o of the level trigger block and this process goes on and on.

The level trigger vector needs to be aligned with the Datahr_a_piped_level_o, Datahr_b_piped_level_o, and Data_valid_piped_level_o. We do it by latching the level trigger vector two times and we get the LevelTrigg_vector_piped_o as shown in the figure 31. By generating this Allow_level_trigger_reset_i in hardware we let the level trigger block to generate trigger vector at a very high speed compared to sending Allow_level_trigger_reset_i signal from API. The maximum PRF for the level trigger block is 25MHz which means that the time difference between two successive trigger pulses generated by the level trigger block is 8 clock cycles where the clock is running at 200MHz. By allowing the Allow_level_trigger_reset_i to be generated in hardware we can achieve this maximum trigger pulse generation (at 25MHz) from the level trigger block.

Averaging unit has a PRF up to 100MHz i.e. the set up time for the averaging unit from the time it has received the last sample from the current record to the first sample of the next record is 2 clock cycles, where the clock is running at 200MHz. So this new implementation for configuring the level trigger block will improve the performance of the averaging unit a lot.
All of the configuration signals from API are latched in the algorithm registers known as SPI registers inside the Algo_FPGA. From SPI registers the configuration signals are sent to different modules such as level trigger block, averaging etc. Figure 32 illustrates a connection diagram for API, SPI registers, Level trigger block and averaging block.

![Figure 32: Connection of Averaging block with other components.](image)

The Algo_FPGA is connected with the Comm_FPGA. Comm_FPGA handles all the communication of the ADQ214 board with the external world for example PC with the help of USB/PXI interface. API commands are sent through USB interface.

The output of the averaging block is bypassed through the Comm_FPGA and then sent to the API. There are different modes of how data can be collected from Comm_FPGA to PC. When we do the averaging, the data is collected in streaming mode and in this case we don’t use the DDR memory available connected with Comm_FPGA. Figure 33 illustrates the connection diagram of ADQ214 with the PC.
Figure 33: Connection diagram of ADQ214 with PC.
2.2.15 C/C++ programming for Streaming Mode for Averaging

In order to achieve the desired output from the averaging block it’s important that we configure the averaging block and other components properly. We configure different components of the ADQ214 from API which is an environment written in C/C++ language.

There are different modes available in the API for data transfer from ADQ214 board to PC. In case of averaging streaming mode is used where on board DDR memory is not used. The old existing function adq214_streaming has been modified to suit with the new hardware “Averaging” inside the ADQ214. A few small sub functions have been created in C/C++ for generating configuration signals for level trigger block and averaging block. Table 2 illustrates these signals.

Table 2: Signal descriptions for signals in C/C++ program [2].

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Descriptions of the signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>TrigLevel</td>
<td>Defines the level at which the level trigger block should generate a trigger pulse. Range: -8192 &lt;= TrigLevel &lt;= 8192</td>
</tr>
<tr>
<td>stream_ch</td>
<td>Defines which channel the streaming is active for. Valid values: ADQ214_STREAM_ENABLED_A, ADQ214_STREAM_ENABLED_B, ADQ214_STREAM_ENABLED_BOTH</td>
</tr>
<tr>
<td>pll_divider</td>
<td>Defines the divider value in the pll. Range: 2 &lt;= divider &lt;= 20. f_adc = 800MHz/divider</td>
</tr>
<tr>
<td>n_holdoff_delays</td>
<td>Defines the number of delay hold off or pre-trigger samples. Holdoff mode: A value n will cause 0 or n<em>2 holdoff delays. For example if n_holdoff_delays = 0 will cause no hold off delay at all. n_holdoff_delays = 1 will case 2 samples of delay. n_holdoff_delays = 2 will case 4 samples of delay and so on. Pre-trigger mode: Incase of pre-trigger mode the minumum number of pre-trigger samples has to be assigned to 3 in API. A value n will cause n</em>2 pre-trigger samples. For example if n_holdoff_delays = 3 will cause 6 pre-trigger samples. n_holdoff_delays = 4 will cause 8 pre-trigger samples. n_holdoff_delays = 5 will cause 10 pre-trigger samples and so on.</td>
</tr>
</tbody>
</table>
**n_records**

Defines the number of records we want to accumulate. Minimum value is 1. A value n will cause n numbers of records to be collected for averaging.

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Descriptions of the signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>n_samples_collect</td>
<td>Defines the number the samples per record. Has to be always even number. A value n will cause n number of samples per record. Minimum value = 2.</td>
</tr>
</tbody>
</table>

- n_samples_collect = 2 will define 2 samples per record.
- n_samples_collect = 4 will define 4 samples per record.
- n_samples_collect = 6 will define 6 samples per record and so on.

We write the configuration signals to the address of the SPI registers. Full detail of the SPI registers is shown in Table 1. The following table 3 is a part from the table 1.

**Table 3: Partial SPI registers.**

<table>
<thead>
<tr>
<th>SPI Address : {15’h36,37} == decimal {54,55}</th>
<th>#of Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPI Address : {15’h38,39} == decimal {56,57}</td>
<td>#of Batches/Records</td>
</tr>
</tbody>
</table>

- UserRegister_in[2]
- UserRegister_in[3]

So we will write the value of n_samples_collect to the addresses {54,55} of the SPI registers.

Sample code:

```
CHECKADQ(ADQ214_WriteAlgoRegister(adq_cu,1,54, NofSamples_lsb));
CHECKADQ(ADQ214_WriteAlgoRegister(adq_cu,1,55, NofSamples_msb));
```

Each SPI register is 16-bit wide. So we have to split the decimal value to lsb and msb and then write those to the appropriate addresses. Address 54 holds the lsb part (NofSamples_lsb) of the value of n_samples_collect and address 55 holds the msb part (NofSamples_msb) of the value of the n_samples_collect.

Sample code for splitting a 32-bit value to two 16-bit values:

```
// NofSamples is 32-bit. NofSamples_msb and NofSamples_lsb are 16-bit unsigned
NofSamples_msb = (NofSamples >> 16);
NofSamples_lsb = (NofSamples & 0xFFFF);
```

We do the same for other configuration signals.
For the other interested configuration signals we write values directly to the appropriate address of the SPI registers.

**Table 4: Signals’ description connected to SPI registers.**

<table>
<thead>
<tr>
<th>SPI Address : (15’h32,33) == decimal {50 51}</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UserRegister_in[0]</strong></td>
<td></td>
</tr>
</tbody>
</table>
| UserRegister_in[0][0]                        | 1 = Pre-Trigger  
0 = Holdoff   |
| UserRegister_in[0][1]                        | 1 = ARMED  
0 = Reset   |
| UserRegister_in[0][2]                        | 1 = Read  
0 = Write   |
| UserRegister_in[0][3]                        | Unused  
(Perhaps can be used to define Extern trigger/ Level Trigger for next generation ADQ214 with extern trigger port with Algo_FPGA) |
| UserRegister_in[0][4]                        | Unused   |
| UserRegister_in[0][5]                        | 1 = selects channel A for level trigger generation  
0 = doesn’t select channel A |
| UserRegister_in[0][6]                        | 1 = selects channel B for level trigger generation  
0 = doesn’t select channel B |
| UserRegister_in[0][7]                        | 1 = checks positive edge for level trigger generation  
0 = checks negative edge for level trigger generation |
<p>| UserRegister_in[0][8]                        | 1 = sets the data as valid. This is kept always 1. |</p>
<table>
<thead>
<tr>
<th>UserRegister_in[0][9]</th>
<th>Unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserRegister_in[0][10]</td>
<td>Unused</td>
</tr>
<tr>
<td>UserRegister_in[0][11]</td>
<td>Unused</td>
</tr>
<tr>
<td>UserRegister_in[0][31]</td>
<td>Unused</td>
</tr>
</tbody>
</table>
SPI registers with address {50, 51} will contain the configuration signals for example to arm or reset the averaging block, the mode Pre-trigger or delay hold off, Read or Write etc. UserRegister_in[0] is a 32-bit wide register which is connected to SPI register with address {50, 51}.

So if we want to write new samples into the averaging we will send a value ‘0’ to UserRegister_in[0][2] from API or a ‘1’ if we want to read out the accumulated value from the averaging block.

A value at ‘0’ to UserRegister_in[0][0] will set the averaging mode in hold off mode and ‘1’ will set the averaging mode in pre-trigger mode.

A value at ‘0’ to UserRegister_in[0][1] will reset the averaging block and ‘1’ will set the averaging block in armed mode.

A value at ‘0’ to UserRegister_in[0][5] will disable the channel A and a ‘1’ will enable.
A value at ‘0’ to UserRegister_in[0][6] will disable the channel B and a ‘1’ will enable.
A value at ‘0’ to UserRegister_in[0][7] will set level trigger block to generate trigger pulse when the incoming signal passes level trigger level and goes towards the negative. For ‘1’ trigger pulse will be generated when the incoming signal passes the level but in positive direction.

A value at ‘0’ to UserRegister_in[0][8] will indicate invalid data to the averaging block. This is kept always to ‘1’.

Sample code to reset the averaging unit:

```c
CHECKADQ(ADQ214_WriteAlgoRegister(adq_cu,1,50,0));
CHECKADQ(ADQ214_WriteAlgoRegister(adq_cu,1,51,0));
```

Sample code to configure for read mode:

```c
CHECKADQ(ADQ214_WriteAlgoRegister(adq_cu,1,50,6));
CHECKADQ(ADQ214_WriteAlgoRegister(adq_cu,1,51,6));
```
2.2.16 Result

The result achieved from HW implementation is satisfactory. The averaging speed is much higher for HW implementation in FPGA than software implementation in Matlab. Figure 34 shows the result from HW implementation where the number of records is 10,000.

As we see that the resulting signal i.e. the blue curve is more accurate and with less noise which is seen in the original signal i.e. the red curve.
The HW implementation has been done in such a way one can even see the nature of averaged curve before the trigger point in pre-trigger mode. Figure 35 illustrates this.

Figure 35: Pre-trigger mode with pre-trigger samples = 10 and level trigger level = 100.

By careful observation it is seen that 10 pre-samples have been included in the resultant curve. The level trigger block will generate a trigger pulse when the input signal level crosses or is equal to 100 and triggers the averaging unit to start data acquisition. By using pre-trigger mode and setting pre-trigger samples = 10 the above result is achieved. The first 10 samples in the curve are the averaged samples of input signals which are the samples under the level trigger level = 100 in this case. The rest of the samples are the averaged samples of input signal samples which reside at the level equal and/or above the level trigger level = 100 in this case.
It is also possible to skip an amount of samples after a trigger has occurred for data acquisition. This is achieved from the holdoff mode. Figure 36 illustrates the result from this mode. Figure 35 can be used as a reference to understand figure 36.

Figure 36: Holdoff mode with holdoff delay samples = 10 and level trigger level = 100.

Here 10 samples have been skipped from the time the signal level is equal or greater than the level trigger level = 100 in this case. So the first sample seen above is actually the 11th sample as the first 10 samples have been skipped. In this mode it is possible to achieve averaging with 0 sample skipping to 65kilo samples skipping.

The PRF (Pulse Repeat Frequency) for the averaging unit is 100 MHz which means that the averaging unit will be ready to start a new acquisition two clock cycles after the last sample from the previous acquisition has been received. If the averaging unit is trying to trigger within this two clock
cycles, the averaging unit will simply ignore the trigger input within these two clock cycles and get ready for a trigger impulse after that. It takes two clock cycles for the state machine to be ready for a new trigger i.e. a new acquisition.
Chapter 3
Conclusion

The goal of this thesis work is to implement signal averaging both in software and hardware for SP Devices digitizer ADQ214 and the goal has been achieved successfully.

A function for signal averaging has been implemented in Matlab which we refer to as the software implementation. The software implementation also utilizes the special feature of the trigger information and performs averaging with interpolation as well which means it is possible to achieve two types of averaging: one without interpolation and one with interpolation. Averaging can be achieved in a number of modes for example pre-trigger mode, hold off mode, with level trigger or external trigger. For averaging with interpolation we must choose external trigger as trigger input because the special feature of the trigger information is achieved when the trigger input is external. Averaging with interpolation gives more precise and accurate information of the signal than averaging without interpolation.

A hardware accelerator for averaging has been designed in Verilog and implemented on one of the onboard Virtex5 FPGAs available on the SP Devices digitizer ADQ214 and this is referred to as the hardware implementation. Both pre-trigger and holdoff modes have been implemented in hardware.

The hardware implementation increases the performance a lot in particular the speed if it is compared to the software implementation. All the samples are stored and accumulated in the hardware at a speed of 200 MHz. It is possible to store and accumulate 65536 records without any overflow as the averaging unit has to wait only two clock cycles in between two successive acquisitions.

In case of software implementation the amount of samples for each record stored on the onboard DDR memory are transferred to the PC using USB interface and stored in memory of the PC. The communication speed here is the bottleneck especially when it is necessary to average a larger amount of records. For averaging for example 65000 records it is required to transfer 65000 records to the PC using USB interface and this takes a significant amount of time. The amount of samples per record also plays an important role when it about to transfer a record from the on board DDR memory to the PC thorough USB interface.

In case of hardware implementation it is required only one time to transfer the result from the board to the PC when a read operation is sent regardless of the number of records is under consideration.

As the current digitizer ADQ214 doesn’t have external trigger port connected to the algorithm FPGA, only trigger pulse from level trigger block is used to trigger the averaging block. The averaging unit works at a speed of 200 MHz with a set up time of 2 clock cycles. It is necessary that the trigger pulse generator also has a very low setup time i.e. the time between two successive pulses. A configuration unit for level trigger block has been implemented in hardware to maximize the performance of the averaging block in other words to minimize the setup time of the level trigger block. Implementation of this configuration unit was not directly a part of the goal of this thesis but in order to maximize the overall performance the necessity of implementation of this unit has aroused. After minimizing the setup time for level trigger block a PRF for level trigger block is achieved as much as 25 MHz. Even though the averaging unit has a PRF = 100 MHz the overall PRF
of the system will be 25 MHz because of the level trigger block. The averaging unit will have to wait for a trigger pulse from the level trigger block. In the next generation ADQ214 board will have a direct connection of external trigger pin with the algorithm FPGA and thus it will be possible to achieve a higher PRF of the whole system because the setup time for external trigger block is much less than that of level trigger block.

The averaged signal contains more accurate information of the original signal as the number of records is increased in averaging. The more the number of records the more the result is close to the original signal. Here original signal means the input signal without noise and distortion. The current design is capable of handling 65536 records with 2024 samples per record. The number of samples per record can be increased up to 9 mega samples which is the maximum limit supported by Xilinx LogiCORE IP Block Memory Generator. The only thing that needs to be done is to increase the depth of the RAM for the averaging unit. The current signal’s width is enough for addressing such amount of samples per record. While increasing the depth of the RAM it is necessary to consider the free space available on the FPGA as well. The number of pre-samples can be chosen up to 2024 samples now but it can be increase just by increasing the size of the pre-trigger buffer up to 4,194,304 samples but again maximum size will also depend on the available free space available on the FPGA as well.

As this thesis has been carried out at SP Devices and with their product ADQ214, the difficulties encountered were to get to know the product well, the existing systems available on ADQ214, the ways of configuring the ADQ214, finding proper information from different documents for example different users’ guides etc. Sometimes all necessary information was not available in those users’ guide but to the persons who actually implemented those parts of the system. Learning and writing Verilog code as well as understanding Verilog code written by others was a challenge as all FPGA code available for ADQ214 is in Verilog. But with help from other engineers at SP Devices it has been possible to carry out the thesis and reach the goal of this thesis work.
References


