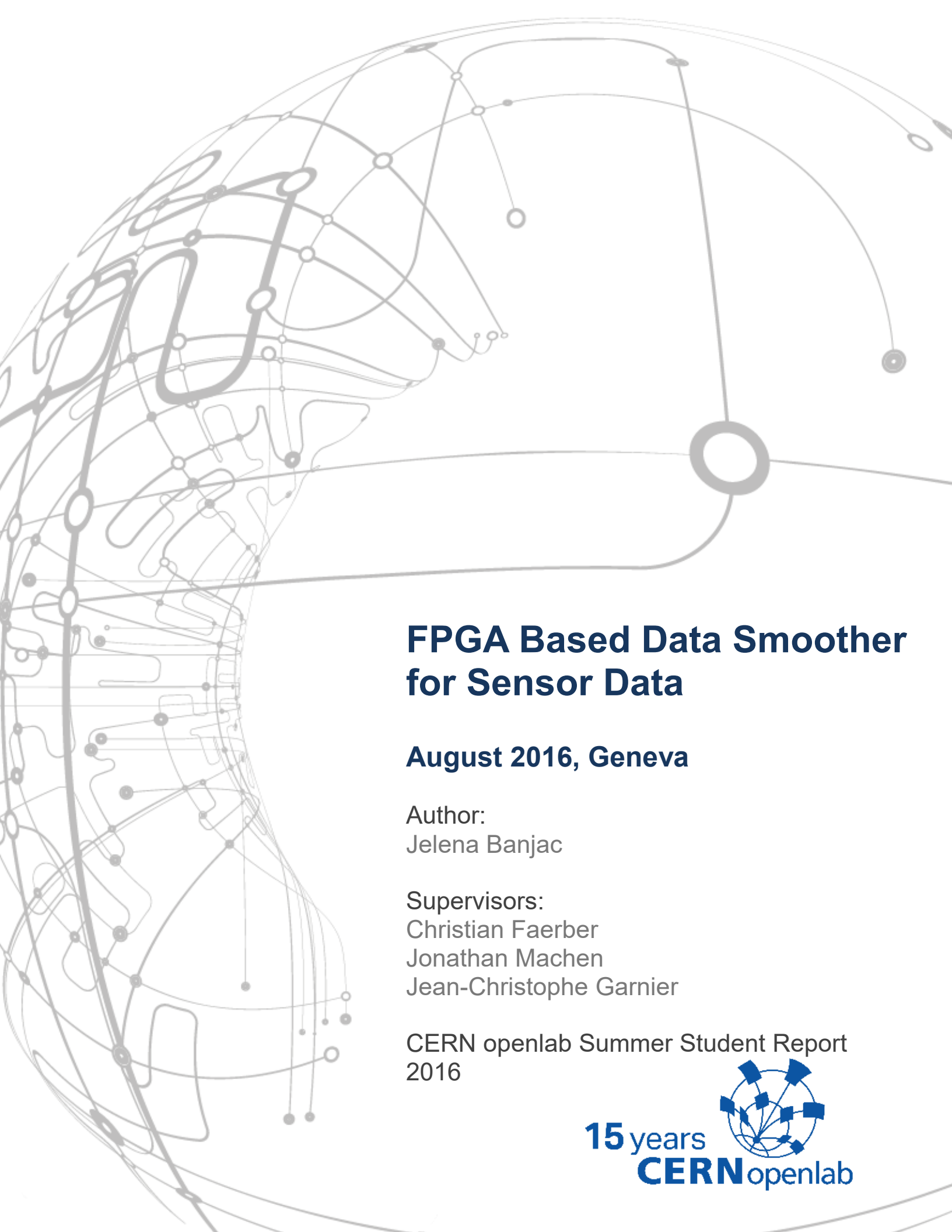


FPGA Based Data Smoother for Sensor Data

August 2016, Geneva

Author:
Jelena Banjac

Supervisors:
Christian Faerber
Jonathan Machen
Jean-Christophe Garnier

CERN openlab Summer Student Report 2016

15 years CERN openlab

Project Specification

In this project it should be tested to use a sensor smoothing algorithm on an FPGA to directly reduce the noise on raw sensor data in general. Noise is a large issue for all high energy physics detectors and it is quite common to use some kind of pre-processing like clustering to reduce the needed bandwidth and reduce the later needed processing complexity due to combinatorics. For sensor data from vibrations monitoring systems the reduction of noise is the main issue.

Different smoothing algorithms will be investigated and afterwards implemented on a modern Cyclon V SOC FPGA. The data source will be a vibration sensor foreseen for monitoring the hard drives of the LHCb computing farm.

To realize the system the processing of fix-point calculations in HDL will be studied and simulated in ModelSim. Furthermore, a test bench for the system has to be written, to test the different smoothing algorithms. Afterwards the performance of the designs will be tested with real sensor data.

Abstract

The primary output of any experiment in which significant information is to be extracted is information which measures the phenomenon under observation. Indistinguishable from this information are random errors which, regardless of their source, are usually described as noise. Of importance to the experimenter is the removal of as much of this noise as possible without, at the same time, overly degrading the underlying information.

In this experimental work, the information from vibration sensor is obtained in the form of four-column table of numbers. This paper is concerned with computational method for the removal of the random noise from such information as well as implementation of this method in HDLs (Hardware Description Languages) and run the algorithm on an FPGA (field programmable gate array).

In this project, random noise from vibration sensor's data is removed using the smoothing algorithm, which is called moving average. This algorithm is implemented inside an average block using VHDL and Verilog languages. The average block is using binary fixed point math library (fixed point addition and fixed point division) as well as Finite State Machine (FSM) and this is ran through the pipeline on an FPGA board.

Finally, we show the outcome of smoothing data from the vibration sensor and what was the influence of the implemented smoothing method using graphs, histograms and simulation results.

Keywords: FPGA, smoothing data, VHDL, Verilog, average block, fixed point addition, fixed point division, sensor data.

Table of Contents

1 Introduction

The field programmable gate arrays (FPGAs) are largely used in many different areas in High Energy Physics (HEP) at CERN. This contribution aims at testing and reviewing the usage of FPGAs in smoothing vibration sensor data in order to reduce the noise from the dataset.

Chapter 1 introduces closely the project challenge, explaining why the project was proposed and what the main goal is.

Chapter 2 describes the development environment which enables any reader to use or to further improve the code.

Chapter 3 tells more about the vibration sensor itself and describes the type of data that is gathered from the vibration sensor.

Chapter 4 demonstrates the data smoothing algorithm used in this project – moving average algorithm. It also analyses some of the main concerns, fixed point division and addition, that need to be implemented in hardware.

Chapter 5 presents the implementation of smoothing algorithm in HDL and describes the algorithms and libraries used.

Chapter 6 evaluates the results from simulations and test benches and analyses overall performance.

Chapter 7 shows project conclusions, project's current status, advantages and disadvantages of implemented algorithm and recommendations for future work.

1.1 Previous Projects

FPGA's are used in a wide field, ranging from fast proto-typing, to accelerating software. I have been working on at the beginning of my summer student internship on acceleration of the execution time of Java programs on FPGAs. Although little used in High Energy Physics analysis code, Java is an ubiquitous programming language, used in countless domains, including the LHC controls. Because Java bytecode runs on a virtual machine, attempts have been made to accelerate Java programs by "hardening" the virtual machine. However these have been marred somewhat by the lack of flexibility of ASIC solutions [1]. Intel's new Xeon/FPGA hybrid platform offers a completely new perspective for such attempts. The main goal was to use Java processor on an FPGA as Java Virtual Machine (JVM) to run Java code on an FPGA platform.

Another attempt was to use the Aparapi API (a parallel API). Aparapi allows Java developers to take advantage of the compute power of GPU devices. It does this by converting Java bytecode to OpenCL at runtime and executing on the GPU. The main goal of the project was to get Java code running on an FPGA. In order to work with Aparapi API, development environment was set up thanks to the great documentation of `aparapi-ucores`[1]. A simple Java program for calculating numeric squares on a Nallatech PCIe card was implemented successfully. However, the real application that was expected to be tested, were not easy enough to implement in a short time.

[1] https://gitlab.com/mora/aparapi-ucores

1.2 Project Challenge

The main project is about how FPGAs can be used to smooth data that we are getting from a vibration sensor. The use of smoothing algorithms will be discussed as well as the algorithm that is used in this project. In addition, the implementation in VHDL and Verilog will also be explained in detail.

The output data from the sensor are used as input data for the smoothing algorithm that is implemented on an FPGA board. It is expected that we get smoother data with reduced noise (see Figure 1).

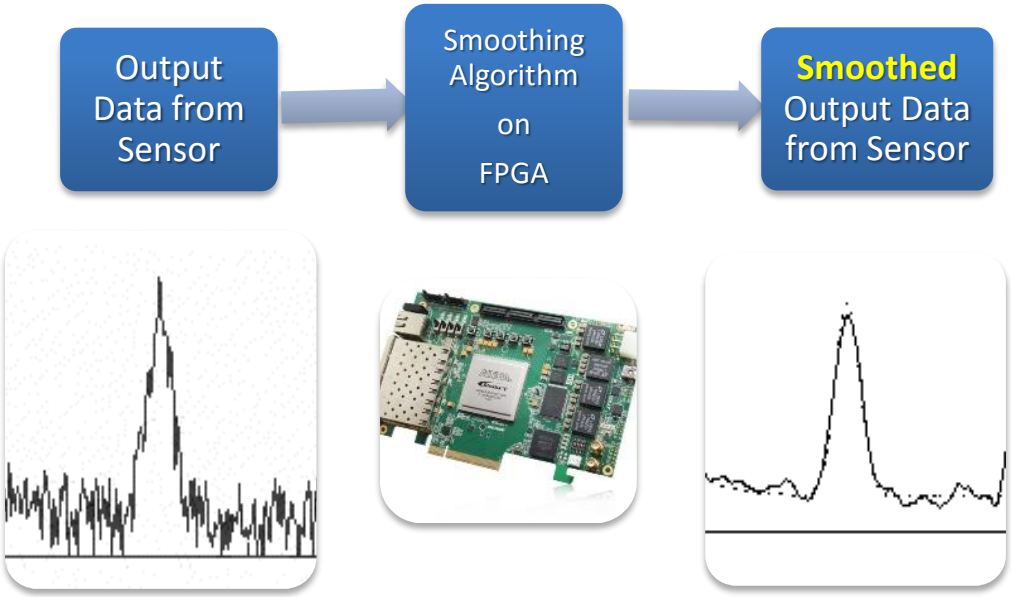


Figure 1. Visualized project challenge

2 Development Environment

The temporary development name of this project is `AverageBlock` and is hosted on gitlab[2].

Average block is developed on a Linux environment and it was tested on CERN CentOS 7[3] but should also run on other distributions.

The folder structure of the projects' root directory can be seen in Figure 2.

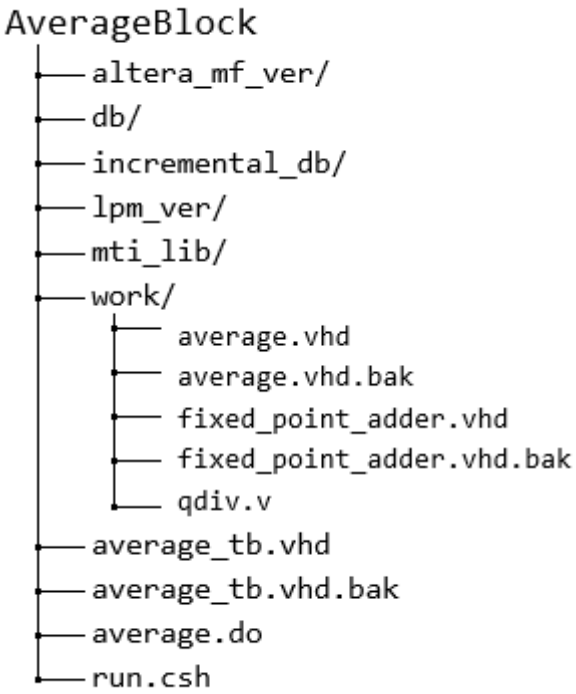


Figure 2. Project folder structure

The `work` directory contains all the files related to the implementation of project written in VHDL and Verilog. Average block is used for averaging data gathered from the vibration sensor (see 5.1) and is implemented in the `average.vhd` block. The fixed point adder (see 5.2) as well as the fixed point divider (see 5.3) are implemented in `fixed_point_adder.vhd` and `qdiv.v` respectively.

The testbench for verifying functionality of the block is created inside `average_tb.vhd`. The ModelSim (see 2.2) Script File `average.do` is used for storing waveform display settings, as well as for other scripting within ModelSim.

File `run.csh` is used to set up the necessary environment variables.

[2] https://gitlab.com/jelena-b94/AverageBlock
[3] https://linux.web.cern.ch/linux/centos7/

2.1 Altera's Quartus II

Altera Quartus II[4] is a design software produced by Altera. Quartus II enables us to compile these designs that are implemented in VHDL and Verilog for hardware description.

Figure 3. Altera Quartus II's logo [3]

Quartus II has incremental compilation [2] feature support. In order to successfully compile this project, it is required that all these compilation stages are compiled successfully.

Before simulating this project design, we need to compile the source files and testbench. For hierarchical designs, compile the lower level design blocks before the higher level design blocks. To compile this average block project, we define parameters in ModelSim `*.do` file:

```
$ vcom work/fixed_point_adder.vhd

$ vlog work/qdiv.q

$ vcom work/average.vhd

$ vcom average_tb.vhd
```

After successful compilation, ModelSim is used for simulation, verification and debugging this project. Average block files are loaded for the simulation inside the `*.do` file as well:

```
$ vsim -novopt average_tb
```

2.2 Mentor Graphics ModelSim

ModelSim[5] provides a comprehensive multi-language HDL simulation and debugging environment for simulation of hardware description languages such as VHDL and Verilog.

Figure 4. ModelSim's logo [4]

In order to open some selected windows, and follow the behaviour of the signals in wave window that we are interested in and for viewing and setting some test patterns (for example, duration of simulation, wave zoom, etc.) in ModelSim, we add additional parameters within the ModelSim do file, `average.do`:

```
$ view wave,

$ add wave -noupdate data_valid

$ run 1000 ns
```

[4] https://www.altera.com/downloads/download-center.html
[5] https://www.altera.com/products/design-software/model---simulation/modelsim-altera-software.html

2.3 R Studio

RStudio[6] is a free and open-source integrated development environment (IDE) for R, a programming language for statistical computing and graphics.

Figure 5. R Studio's logo [5]

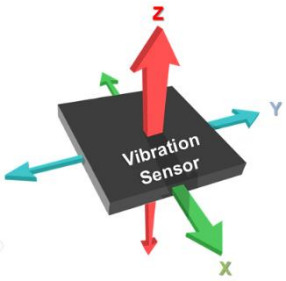
In this project, RStudio is used for visualization of the output data from the vibration sensor as well as the data we get after smoothing it using the FPGA.

Output data from the sensor is written into a simple text file, `noSmooth.txt` in first case (without smoothing block) and `withSmooth.txt` after smoothing the data with the average block.

All the following visualizations of the data in this report are created using Plotly[7]. Plotly for R is an interactive, browser-based charting library built on the open source JavaScript graphing library, plotly.js. It works entirely locally, through the HTML widgets framework.

[6] https://www.rstudio.com/
[7] https://plot.ly/r/

3 Sensor Data



The vibration sensor provides the data used in our tests for all 3 space dimensions (X, Y, Z), which are stored in 16 bit wide locations as fixed point numbers.

The sensor type that is used in this project is ADXL345 (see Figure 6). The ADXL345 is a small, thin, low power, 3-axis accelerometer with high resolution (13-bit) measurement at up to ±16g. Digital output data is formatted as 16-bit twos complement and is accessible through either a SPI (3- or 4-wire) or I^2C digital interface [6]. The I^2C is used in this system.

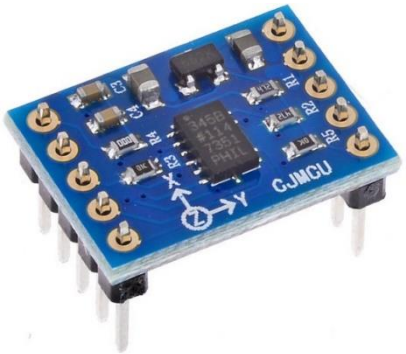


Figure 6. ADXL345 vibration sensor [7]

We needed to implement smoothing because of the electrical noise we have inside the data set that comes from vibration sensor.

Figure 7 shows the role of the vibration sensor in the system. For more details about the average block part see chapter 5.1.

On ARM a Linux distribution is running which writes the data on a SD card and copy the file into mounted network file system.

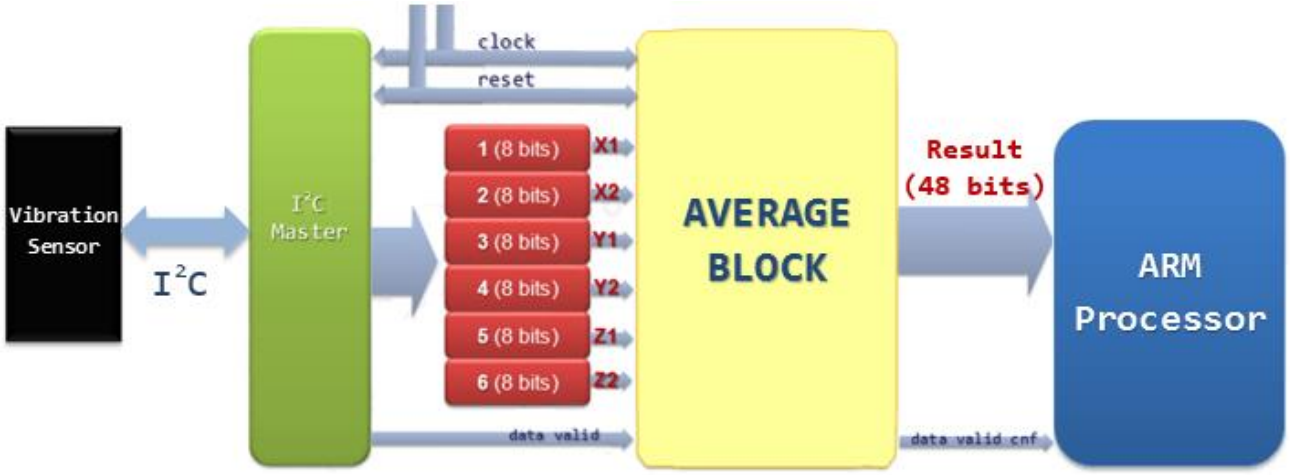


Figure 7. The vibration sensor in general picture

4 Data Smoothing

Data smoothing is the use of an algorithm to reduce noise from a data set, allowing (not always) important patterns to stand out. This can be done in a variety of different ways, including random, random walk, moving average, simple exponential, linear exponential and seasonal exponential smoothing. It is also very important not to overdo data smoothing, at some point it is also bad.

In the following chapter the moving average algorithm which was implemented during this project will be explained.

4.1 Moving Average

Moving average is a calculation to analyse data points by creating series of averages of different subsets of the full data set.

Given a series of numbers and a fixed subset size, the first element of the moving average is obtained by taking the average of the initial fixed subset of the number series. Then the subset is modified by "shifting forward". That is, excluding the first number of the series and including the next number following the original subset in the series. This creates a new subset of numbers, which is averaged. This process is repeated over the entire data series. The plot line connecting all the (fixed) averages is the moving average.

There are several variations[8] of moving average algorithm. In this project, simple form of moving average is implemented. Simple moving average is the unweighted mean of the previous N data.

For better understanding, in the following examples will be explained how the moving algorithm works. In this examples, delays which can appear in calculating sum and quotient is not taken into consideration. Therefore, examples are ideal use cases.

For example, let's say that we have some input data values every clock cycle (see Figure 8). If we calculate average of the first three input data points, we get one output data point that represents average of those first three points (first red value on Figure 9). Afterwards, next three input data points give the next output data point (second red value on Figure 9). This creates a new subset of numbers, which is averaged. This process is repeated over the entire data series. After smoothing, output data can look as shown on the Figure 9.

In addition, there is another example of using input data to calculate smoothed output data. If we use the same subset of input data from Figure 8, we can calculate average value in every clock cycle using the last three input data points (see Figure 10).

The difference between these two examples is that the first one has a reduced bandwidth. In this case, we get the average result every third clock cycle, which is not the case with the second example. In the second example, we get output data every clock cycle. Therefore, second example is more optimized version of the moving average. This we can easily see on the graphs, because first example doesn't have peak anymore. However, second example has peak that is smoother in comparison to data input's peak. In code, the first example is implemented with Finite State Machine (FSM - see 5.4) and the second example is without the FSM.

[8] Variations include: simple, and cumulative, or weighted forms.

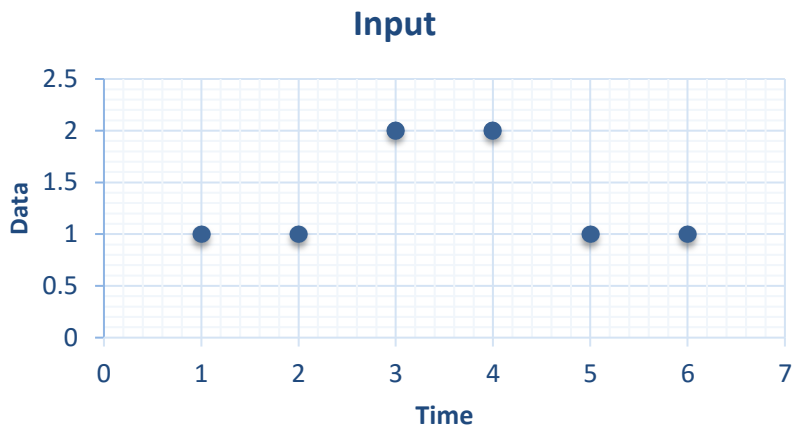


Figure 8. Input data for moving average algorithm

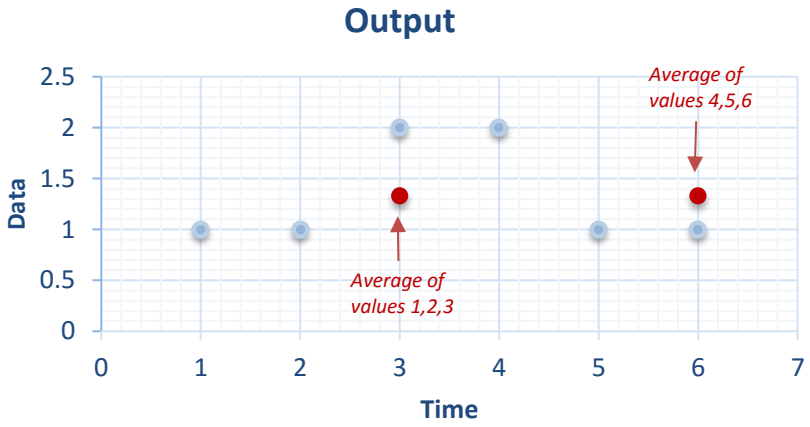


Figure 9. Averaging algorithm with reduced bandwidth

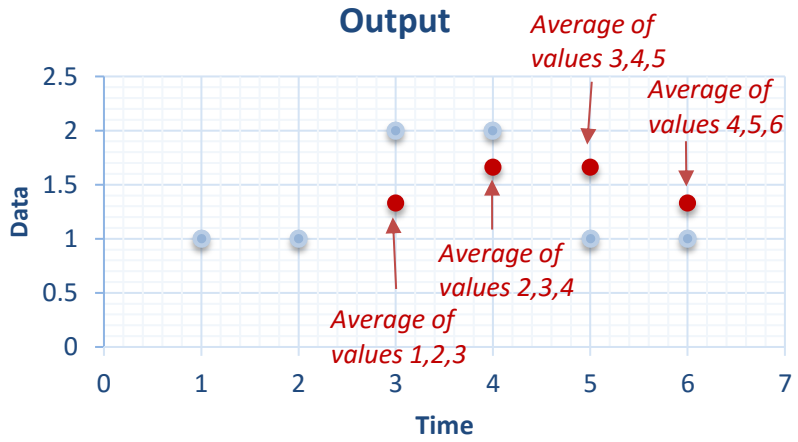


Figure 10. Averaging algorithm with same bandwidth

4.2 Fixed-Point Arithmetic

The fixed-point number representation is a real data type[9] for a number that has a fixed number of digits after and before the radix point[10].

Certain types of embedded systems require the handling of real numbers (or at least what appear to be real numbers). Real numbers can have fractional parts-in other words, something after the decimal point-in contrast to integers which are always whole numbers [9].

Binary fixed point types have a scaling factor that is a power of two (see Figure 11). Summarizing all the products of bit number (0 or 1) with its position weight (2^4, 2^3 …), we can get the decimal value of the binary number.

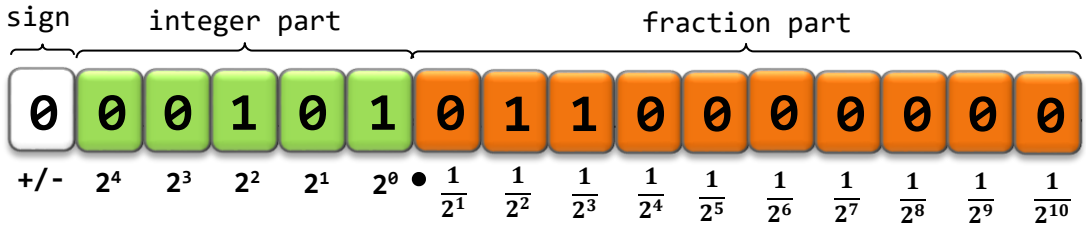


Figure 11. Example of 16-bits fixed point number

There are various notations used to represent word length and radix point in a binary fixed point number. In this project, ***Q****m.f* form is used (*f* represents the number of fractional bits and *m* represents the number of magnitude of integer bits). Since the entire word is a 2's complement integer, a sign bit is implied. For example, ***Q****5.10* describes a number with 5 integer bits and 10 fractional bits stored as a 16-bit 2's complement integer (see Figure 11).

Because fixed point operations can produce results that have more bits that the operands, there is a possibility for information loss. More about this problem is discussed in chapter about fixed point addition (see 4.3).

4.3 Fixed-Point Addition

Adding fixed-point binary numbers is easy. We just perform a normal add (with a caveat) as if it were all an integer number (see Figure 12). If one of the numbers is negative, we take the 2's complement (invert and add 1) to get the negative representation.

$$\begin{array}{lll} & 0001010110000000_2 & (5.375)_{10} \\ + & 0001010110000000_2 & (5.375)_{10} \\ \hline & 0010101100000000_2 & (10.750)_{10} \end{array}$$

Figure 12. Simple example of 16-bits fixed point addition

However, we also need to pay attention to the potential information loss and overflow. For instance, the result of fixed point addition could potentially have more bits than the number of bits in the two

[9] Data type used in a computer program to represent an approximation of a real number. Because the real numbers are not countable, computers cannot represent them exactly using a finite amount of information.
[10] Symbol used in numerical representations to separate the integer part (to the left of the radix point) of a number from its fractional part (to the right of the radix point) [7].

operands and overflow can basically be into a sign bit. In order to keep addition precise and avoid potential overflow in this project, the number of bits of input data (X, Y, Z coordinates) is increased from 16 to 20 bits each.

For more information on how fixed point addition is implemented in VHDL and encountered challenges, see chapter 5.2.

4.4 Fixed-Point Division

Fixed point division is much more complicated than fixed point addition. From the perspective of computer architecture, usually it takes a lot more clock cycles[11] than performing an addition in an FPGA.

Division of binary numbers is similar to the division of decimal numbers. Figure 13 shows the division of the two integer numbers. On the left side is an example of division of two binary numbers and on the right side is an example of division of decimal values of the same numbers.

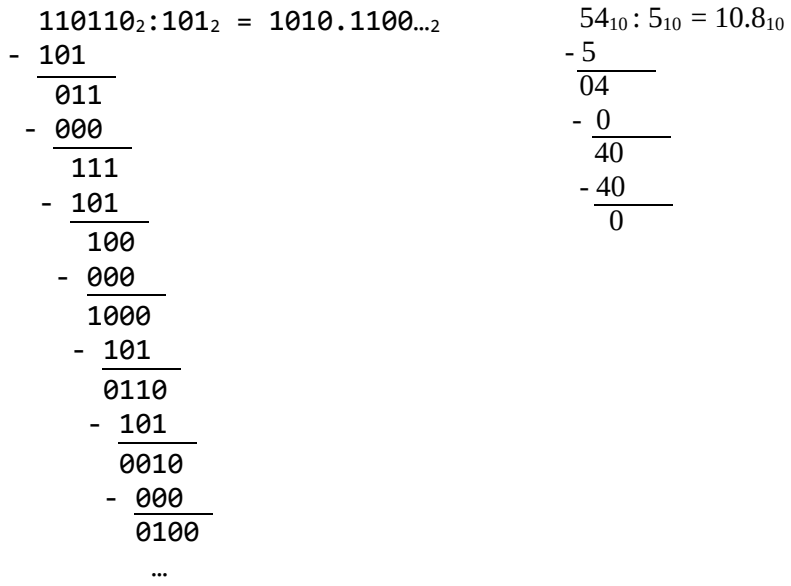


Figure 13. Simple example of division

However, implementation of division of binary numbers in hardware is much more complex and algorithms used for division can vary. In this project we use division algorithm which implementation is well explained in a material about Fixed Point Division[12]. In the following text, implementation is explained using some of the examples from mentioned material.

[11] The time between two adjacent pulses of the oscillator that sets the tempo of the computer processor.
[12] Binary Division – Fixed Point Division by Tom Burke:
https://www.youtube.com/watch?v=TEnaPMYiuR8

The shorter version of division algorithm that shows the most important steps during the process of division is given in the form of pseudocode on the Figure 14. Constant number N is the number of bits of the register and Q is the number of bits for the fractional part.

At the beginning we first need to initialize our counter, divisor and dividend (see the initialization part on the Figure 14). In this algorithm we are ignoring the sign bit of the quotient (`N-1` means that we are throwing away the sign bit) because it is the EXOR (exclusive OR operation) of the two sign bits of divisor and dividend. After the initialization, we are iterating through counter values. In one iteration we are calculating the value of the quotient (for the current iteration) and values of dividend and divisor (for the next iteration). This process repeats until the counter value gets to the zero.

```
// Initialization:
COUNTER = N+Q-1
DIVISOR = Divisor without a sign bit is placed in the highest N-1 bits
          of 2*(N-1)+Q wide register
DIVIDEND = Dividend without a sign bit is placed in the highest N-1 bits
           of N+Q-1 wide register
QUOTIENT = Quotient has all 0s at the begining and is placed in N+Q
           wide register

// I-th iteration:
if DIVISOR_i > DIVIDEND_i then
    QUOTIENT_i = QUOTIENT_previous
    DIVIDEND_next = DIVIDEND_i
else
    QUOTIENT_i[COUNTER_i] = 1
    DIVIDEND_next = DIVIDEND_i - DIVISOR_i
end if
COUNTER_next = COUNTER_i - 1     // Decrease counter
DIVISOR_next = DIVISOR_i >> 1    // Shift divisor one bit to the right
```

Figure 14. Pseudocode for binary division

For better understanding how this algorithm works, we will implement it on the previous example from the Figure 13. Due to the fact that we already know the result of division, at the end we can compare these two results.

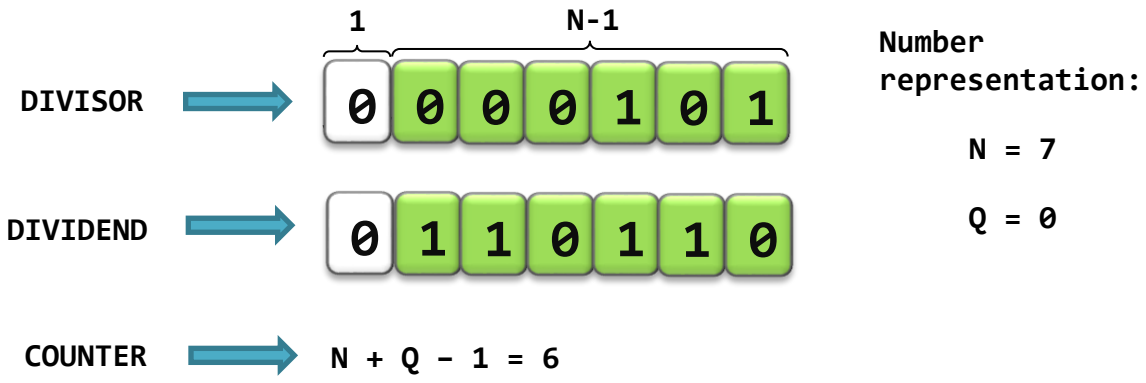


Figure 15. Number representation of divisor and dividend

First, we need registers that are big as these numbers (see Figure 15).

The counter is 6 (N+Q-1=6), the dividend is placed inside the 6-bits wide register (N+Q-1=6) and the divisor is placed in 12-bit wide register (2*(N-1)+Q=12). After this initialization we start to iterate using the binary division algorithm. The result of the iterations as well as result are shown in Table 1.

COUNT	DIVISOR	DIVIDENT	QUOTION
6	000101000000	110110	0000000
5	000010100000	110110	0000000
4	000001010000	110110	0000000
3	000000101000	110110	0001000
2	000000010100	001110	0001000
1	000000001010	001110	0001010
0	000000000101	000100	0001010

Table 1 Example of division algorithm implemented in integer binary division

The quotient is also using the integer representation of the number. So if the result has fractional bits, we miss them because the quotient is truncated (the correct result is 1010.1100… and we got 1010). We also need to pay attention on that the quotient is never bigger than the dividend. If that is the case, then we have an overflow. Overflow bits of quotient are marked with yellow colour. If one of the yellow bits is greater than 0, then we have an overflow.

The next example shows fixed point binary division. We will divide two fixed point numbers ($1.0011_2/0.01_2=0100.110_2$). It starts the same as the integer division from previous example (see Figure 16).

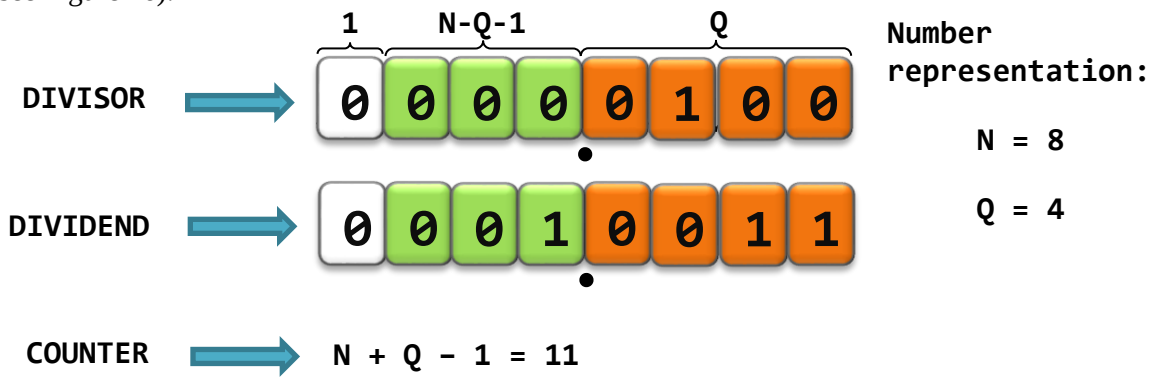


Figure 16 Number representation of divisor and dividend

The counter is 6 (N+Q-1=11), the dividend is placed inside the 6-bits wide register (N+Q-1=11) and the divisor is placed in 12-bit wide register (2*(N-1)+Q=18). After this initialization we start to iterate using the binary division algorithm. The result of the iterations as well as result are shown in Table 2.

COUNT	DIVISOR	DIVIDENT	QUOTION
11	00001000000000000	00100110000	000000000000
10	00000100000000000	00100110000	000000000000
9	00000010000000000	00100110000	000000000000
8	00000001000000000	00100110000	000000000000
7	00000000100000000	00100110000	000000000000
6	00000000010000000	00100110000	000001000000
5	00000000001000000	00000110000	000001000000
4	00000000000100000	00000110000	000001000000
3	00000000000010000	00000110000	000001001000
2	00000000000001000	00000010000	000001001100
1	00000000000000100	00000000000	000001001100
0	00000000000000010	00000000000	000001001100

Table 2 Example of division algorithm implemented in fixed-point binary division (in FPGA)

The quotient is also using the fixed point representation of the number. In this example, we don't lose fractional bits. We also have yellow marked bits that provide us with overflow information.

We can see that this algorithm for binary division is deterministic in time, which means that it always uses the same amount of clock cycles in hardware implementation. For this example to work, we need `4*N-4+3*Q` bits in our registers. Because we have `N+Q-1` shifts and subtracts and one initialization process, it will take `N+Q` clock cycles for this division block to finish.

See chapter 5.3 to see a better overview of the algorithm used in this project.

5 Implementation

5.1 Average Block

The average block represents the main block of the project (see Figure 17). The smoothing algorithm is implemented inside this block. Input data are (X, Y, Z) fixed point numbers that are 16 bits wide. This data is distributed inside 6 registers that are 8 bits long. Therefore, input data is 48 bits long. Average block uses two smaller blocks, for calculating the input sum – block for fixed point adder and for performing division – block for fixed point divider.

The block contains three 48 bit wide registers (R1, R2 and R3) that are used for storing input data of the block. The Finite State Machine (explained in 5.4) is used for controlling the flow of the calculations and writing/reading data.

As a result we get an averaged value of every coordinate. Throughout the whole project, there is a data valid signal pipelined through the design, which is very important. First, for the Finite State Machine conditions for changing state, and second, for setting data output valid signal in time.

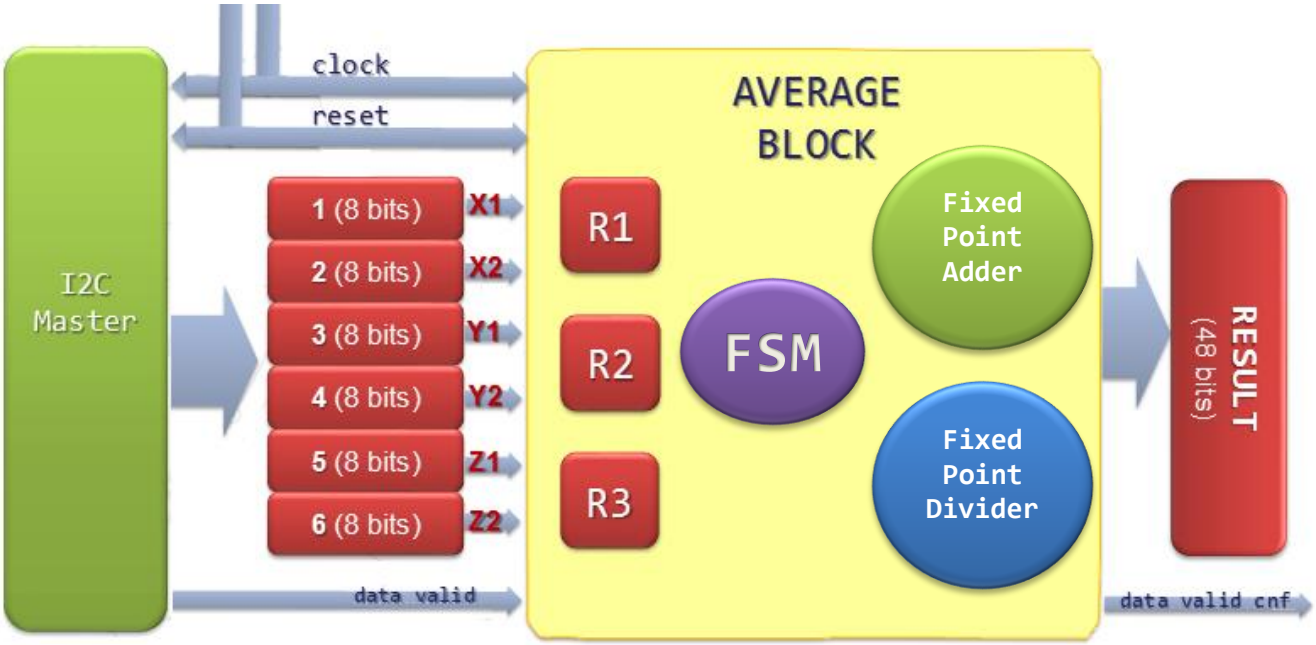


Figure 17. General picture of the project

5.2 Fixed-Point Adder

When adding two numbers an additional bit is required for the result. When adding more than two numbers all of the same word length (WL) width, the number of bits required for the result is ***WL + log_2 (N)***, where N is the number of elements being summed [10].

In this project the sum of three data input values (values from R1, R2 and R3 registers) is calculated. However, this fixed point adder block does not implement the calculation of the sum of all values in the registers. The input for the fixed point adder block has two input values (A and B) and the output of the algorithm is their sum (see Figure 18). How to get the sum of all this register values as soon as possible, is explained about in chapter 5.5 about pipelining.

The Verilog fixed point addition module is transferred to VHDL fixed point adder block and it is based on a Verilog implementation of fixed point math library from OpenCores[13].

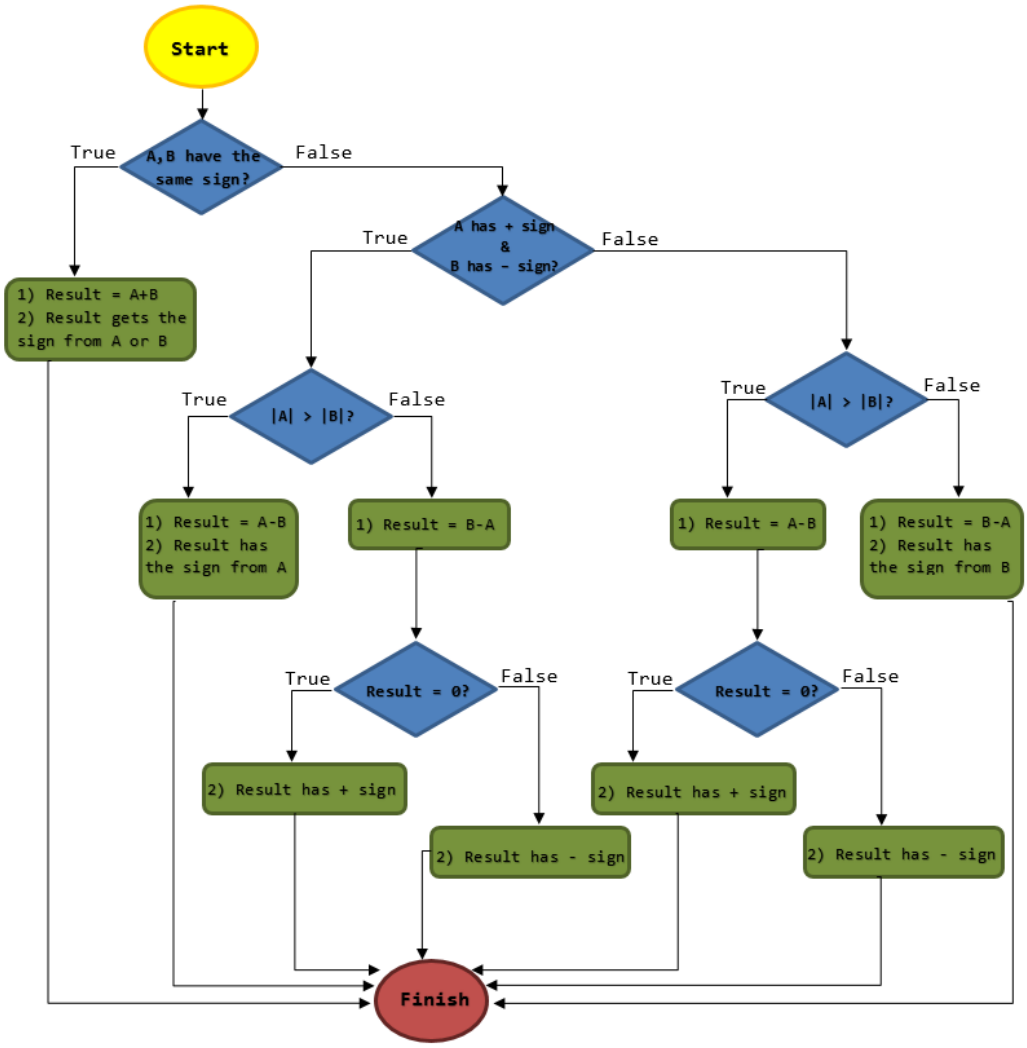


Figure 18. Algorithm for fixed point addition

[13] http://opencores.org/project,verilog_fixed_point_math_library

5.3 Fixed-Point Divider

To calculate the average of input values, besides the fixed point adder block we also have to have a fixed point division block. In this project, the division module from OpenCores[14] is used and this module divides two numbers using the right-shift and subtract algorithm (see Figure 19). Unlike the fixed point adder block, this module requires an input clock. The algorithm that is shown on the picture represents one clock cycle (starts every positive clock edge).

Some changes had to be made due to rounding error in the quotient. Rounding to nearest smaller value is corrected and the algorithm is successfully implemented in the project.

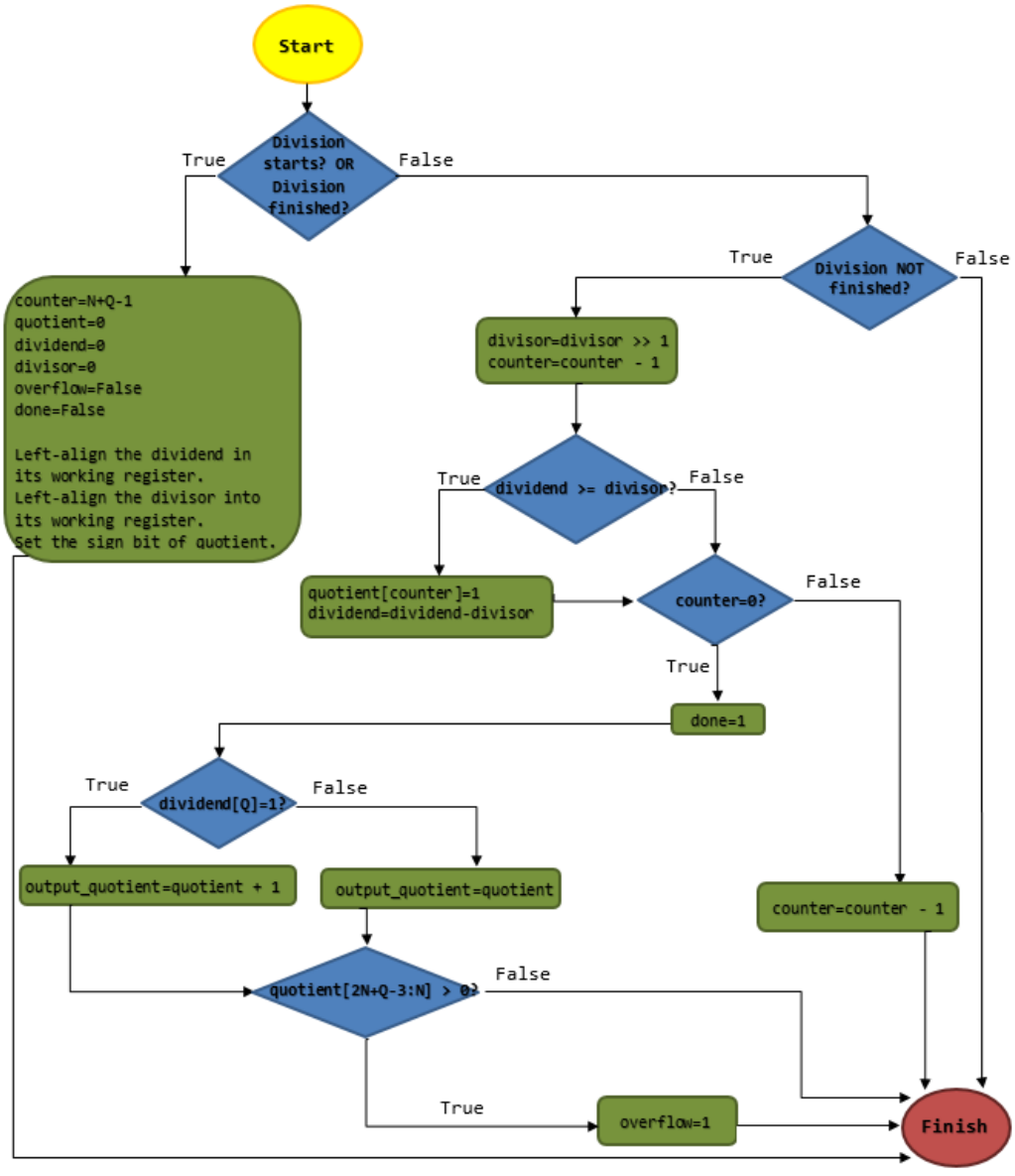


Figure 19. Algorithm for fixed point division

[14] http://opencores.org/project,verilog_fixed_point_math_library

5.4 Finite State Machine (FSM)

Finite State Machine (FSM) is a model of computation used to design both computer programs and sequential logic circuits. It is conceived as an abstract machine that can be in one of a finite number of *states*. The machine is in only one state at a time and the state it is in at any given time is called the *current state*. It can change from one state to another when initiated by a triggering *event* or *condition*; this is called a *transition*. A particular FSM is defined by a list of its states, and the triggering condition for each transition.

In this project, the FSM has five states. In first three transitions (see Figure 20), input data is written inside the R1, R2 and R3 registers. In S3 state, pipelining is started by triggering data valid signal[15] for starting the calculculation (`dv1='1'`). Calculation starts with fixed point addition and afterwards it continues with fixed point division. In the last state, S4, we are waiting a finite number of clock cycles for fixed point division block to finish its calculation. After the last state, we send calculated value as well as data valid signal to the output and then repeating the same process again (starting from state S0).

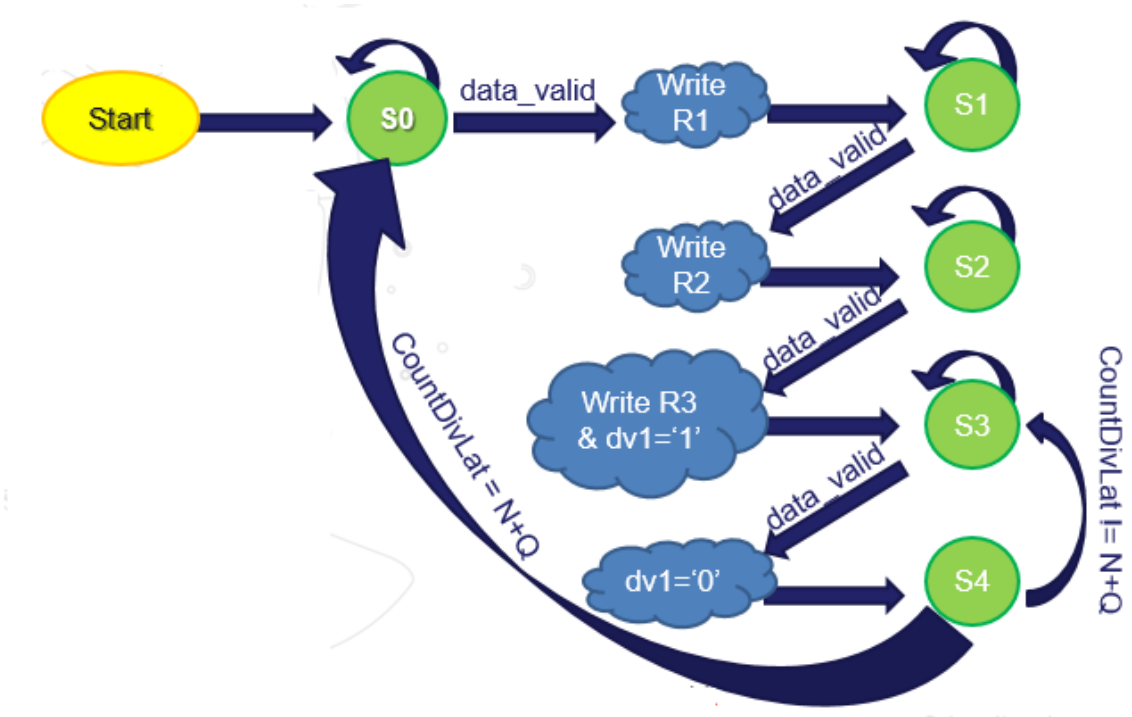


Figure 20. Final state machine in Average Block

[15] In order to know when our output data is valid we use data valid signal. The average block can give us the wrong output before the right one is out, so data valid signal is crucial to correctness of our averaged results.

5.5 Pipelining

Pipeline is a set of data processing elements connected in series, where the output of one element is the input of the next one. The elements of a pipeline are often executed in parallel or in time-sliced fashion.

Pipeline in this project is implemented using several processes (see Figure 21). The first two processes are dealing with fixed point adder. The fixed point adder was rewrote from Verilog to VHDL. Fixed point addition block adds two values from registers. After the calculation, it produces the result of the addition. That result is now input value for fixed point adder block inside the new process.

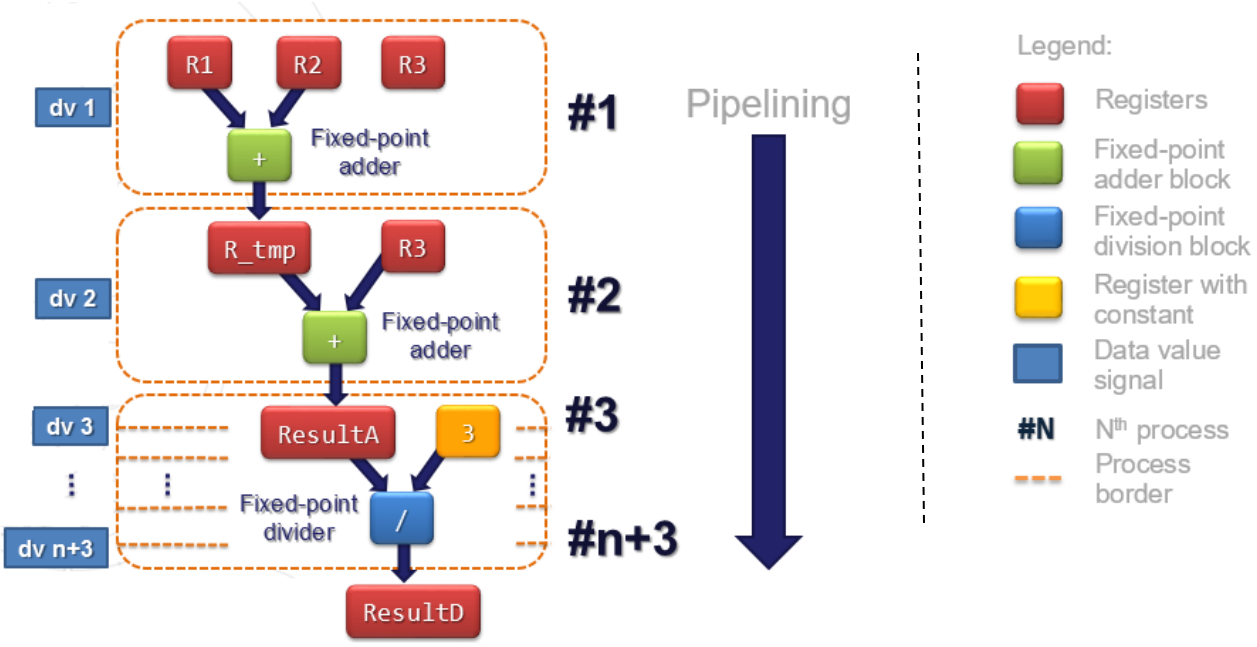


Figure 21. Pipelining implemented in Average Block

Afterwards, N number of processes are used for the latency of the fixed point division. The input values for the fixed point division block are the result from the fixed point adder from the last process and the number of values we summarized (in this case it is 3 values - 3 registers).

The data valid signal is also very important. We propagate the value of the data valid signal (true or false) through whole pipeline process. Therefore, at the end we are sure that output data from fixed point division block is already finished.

The division algorithm used requires several clock cycles, which is called its latency. The division will be finished always after the same period of time, no matter what the input numbers are. At that time we need to take care of data valid signal, so its value doesn't get lost in the process. The solution for that is to implement shifted register of a certain size which helps to propagate and value of valid data throughout division process.

6 Results

Before testing the functionality of the average block on the board, it is necessary to write a test bench and run the simulation. Afterwards, average block can be programmed on the SoCrates II[16] board with a Cyclone V SoC FPGA.

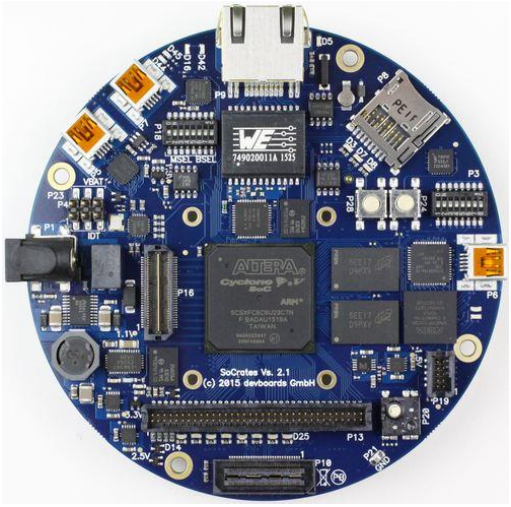


Figure 22. SoCrates II [11]

Figure 23. Cyclone V SoC FPGA [12]

6.1 Test bench and Simulation

With VHDL, it is possible to model the hardware and system design with a test bench, by applying stimulus to the design and to analyse the results. VHDL can be used as a stimulus definition language as well as a hardware description language.

Figure 24 represents a screenshot of successful simulation. Behind this simulation is a test bench that simulates input data for the average block (X, Y, Z coordinates). In this test bench, input data X and Y are zero. However, input data Z increases by one every clock cycle. There is also a signal which notifies us when registers are busy/ready for new input data (`available`).

[16] http://www.devboards.de/en/home/boards/product-details/article/socrates-ii/

On the other hand, in output signals we have result from fixed point adder block (`data_out_addition`) as well as its data valid signal (`data_out_cnf_addition`). Lastly, we have result from fixed point division block (`data_out`) and its data valid signal (`data_out_cnf`).

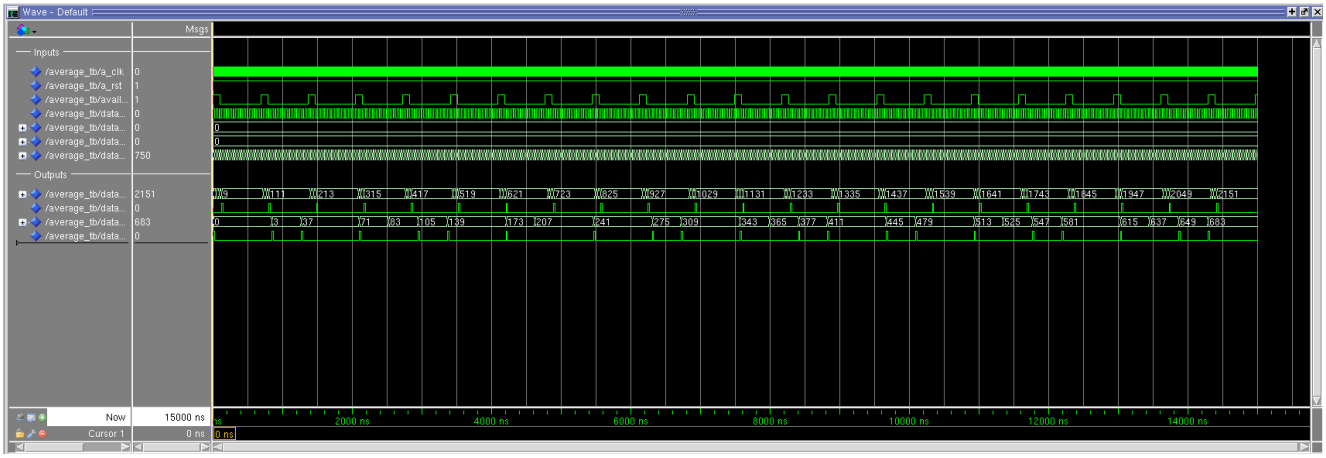


Figure 24. ModelSim simulation of Average Block (full)

For further explanation see Figure 25. For example, first three data inputs are numbers 1, 3 and 5. Their data valid signals are high. However, to write this data inside average block registers, busy signal also needs to be high. That means that the average block is not busy at the moment. First, average block calculates the sum of these input values in the next two clock cycles. Result is 9 and data valid signal for adder is also high. Afterwards, division takes much more clock cycles. In this case it takes around 36 clock cycles. The result of division is 3 and that is the correct answer. Now, it is the same with other data input values.

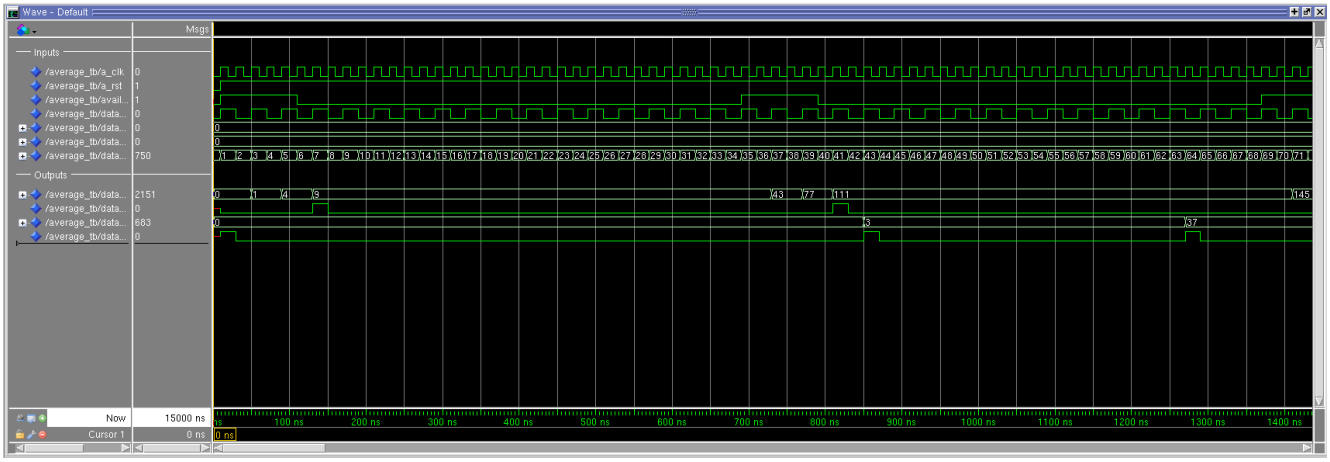


Figure 25. ModelSim simulation of Average Block (zoomed)

6.2 Performance Analysis

The simulation shows the results and the performance which we expected. Afterwards the board was programmed and we can see whether this block works with real data input from the sensor.

Real data from the sensor are written into a text file during a period of time. The text file consists of four columns. The first one represents time and the other three columns are the X, Y and Z coordinates.

In order to improve the visibility of the result of the average block on real sensor data, data plots are created before and after smoothing data.

First plot (see Figure 26) represents visualised Z data values (y axes) during the finite period of time (x axes). This plot presents data without implemented smoothing algorithm.

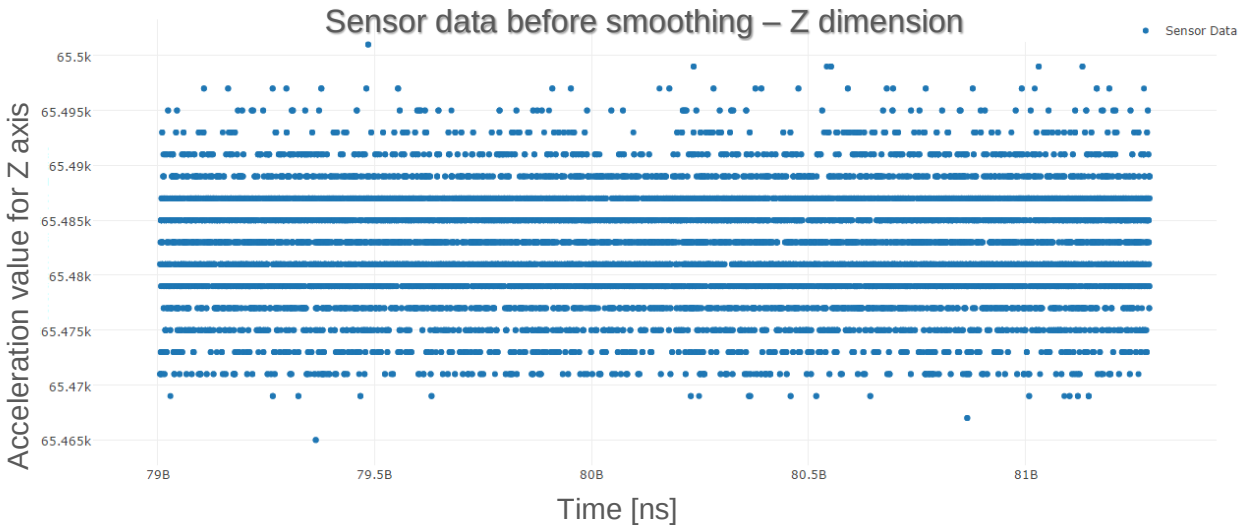


Figure 26. Z axes data from sensor that is not smoothed

The second plot (see Figure 27) also represents visualised Z data values (y axes) during the finite period of time (x axes). However, this plot presents data with implemented smoothing algorithm.

Figure 27. Z axes data from sensor with smoothing

When comparing these two plots of acceleration in Z direction over time, second data plot is seemingly smoothed.

Better way to see the difference between real and smoothed data from the sensor is a histogram. On the Figure 28 we can see what values are most common for the Z coordinate when the data are not smoothed.

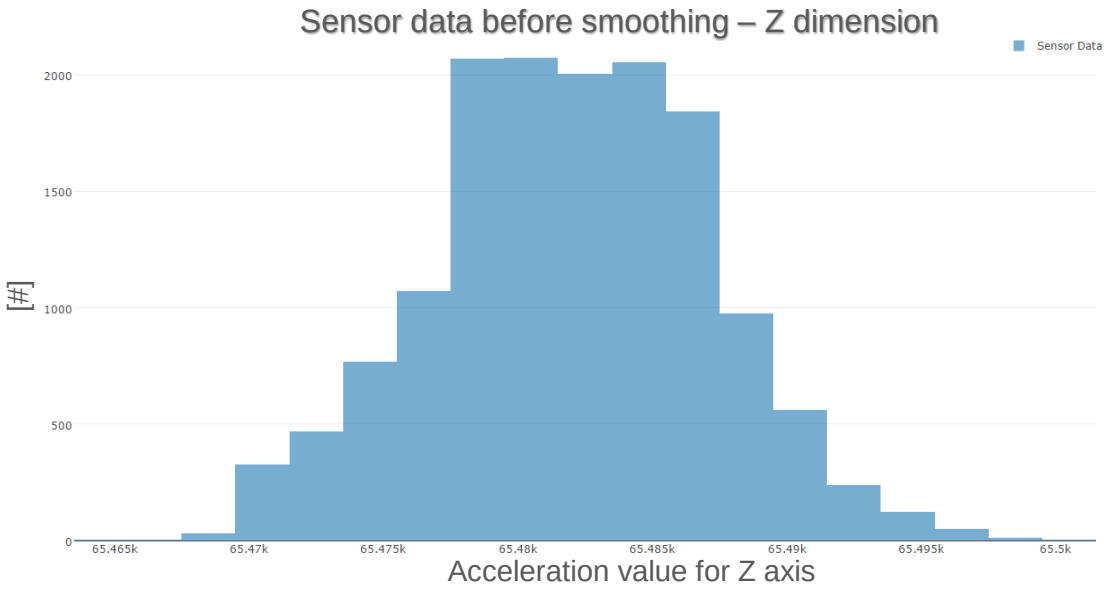


Figure 28. Histogram of Z axes data from sensor that is not smoothed

On the Figure 29 we can see what the most common values for Z coordinate are when data is smoothed. It is expected for both histogram shapes to have the peak on a similar place. We can see from the visualizations that condition is fulfilled. The small deviation is a sensor position effect.

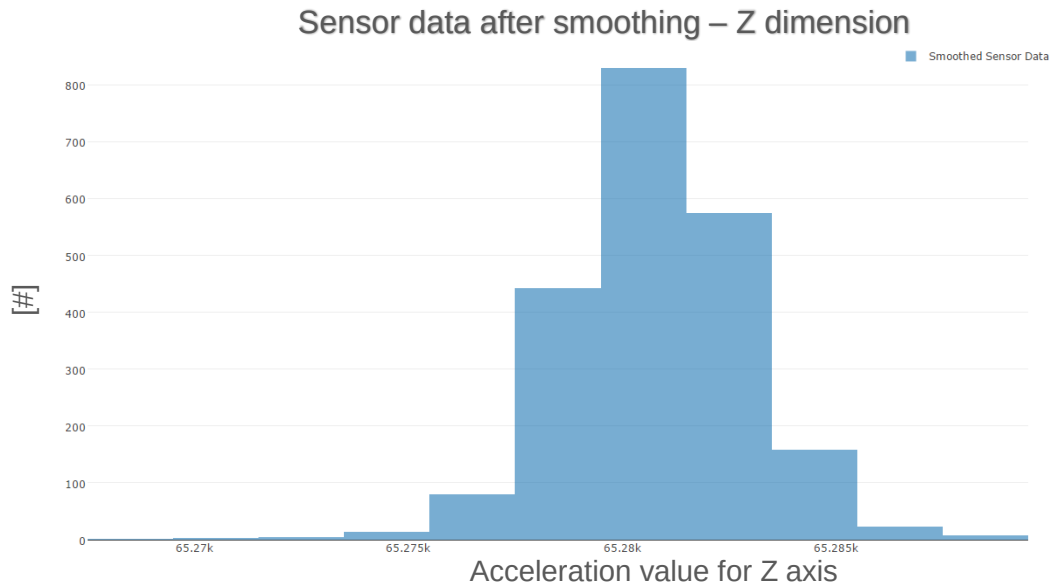


Figure 29. Histogram of Z axes data from sensor that is smoothed

7 Conclusion

The use of smoothing algorithm that is implemented and tested on the sensor data is considered successful. Results before and after the smoothing are compared and the differences are visible.

To achieve the best outcome, several different approaches were made during the implementation on an FPGA. In the following text, these approaches are mentioned. Possible improvements can be done thanks to future proposals.

7.1 Progress Overview and Project Status

One of the first steps was to implement the block with only one 48-bit data input and two fixed point addition blocks. In this case 48-bit input data represented a single bus.

Due to the fact that we don't have all the data in the pipeline, the used FSM needed to be extended with `RESET` state. After that we were sure that only the pipeline defines our data.

Writing test benches is not easy because we need to think about every change of the signals and to take every change into account. After successful simulation, the block for fixed point division is integrated into the system. The next task was to make three 16-bit buses (one for X, second for Y and third for Z coordinate values of the vibration sensor).

After adding three 16-bit data inputs, three pipelines are added - one for X, one for Y and one for Z data input. Now we have three fixed point dividers (for each X, Y and Z), three first stage fixed point addition blocks and three second stage fixed point addition blocks.

The shift register is added because fixed point division takes some clock cycles to complete the calculation. The shift register is the best solution because fixed point addition always takes the same amount of time to complete. Shift register is used to propagate the data valid signal through pipeline.

There was a problem with averaging result. Sometimes result was decreased by one and it seemed like a rounding error. The problem was solved by improving the `qdiv.v` file from fixed point math library. The next problem was the size of registers where input data is placed. The size of these registers is increased from 16-bit to 20-bit because of the problem with overflow in fixed point adder and VHDL coding as well.

In parallel, the average block without FSM was implemented in order to decrease the time of calculation of the average value. The average block without FSM represents a more optimized version of calculating the average value (see Figure 10) and is implemented inside one process. The goal was to implement the streaming of data where the solution is completely pipelined without state machine.

The average block is successfully used for smoothing data from the vibration sensor. Project is done.

7.2 Advantages and Disadvantages

ADVANTAGE:

- Data smoothing makes patterns more visible,
- Removes noise from a data set, allowing important patterns to stand out.

DISADVANTAGE:

- Data smoothing can overlook key information or make important facts less visible.

7.3 Future Work

Ideas for future work enables any reader to further improve the code. Some of the ideas are listed below:

- *Make divider generic*. Make a possibility for user to choose the generic integer (fixed-point) number. At the moment, divider is constant number. If we use generic divider, we can generally produce these registers, process states. However, it can get complicated.
- *Implement different smoothening algorithm.* Possible smoothing algorithms to implement: squares, simple exponential, linear exponential and seasonal exponential smoothing
- *Additional reduction of data rate by amount of averaged points.* Number of registers for writing input data of the block is 3. The next goal is to increase this number to 5.

8 References

[1] An Expanding Role for FPGAs in CERN's Future : http://www.nextplatform.com/2016/01/05/an-expanding-role-for-fpgas-in-cerns-future/ (entered 26.08.2016)

[2] Incremental compilation in Quartus II : http://quartushelp.altera.com/14.0/mergedProjects/comp/increment/comp_view_qid.htm (entered 26.08.2016)

[3] Altera Quartus II's logo : http://www.doulos.com/images/logos/QuartusII.jpg (entered 26.08.2016)

[4] ModelSim's logo : http://media.digikey.com/photos/Altera%20Photos/modelsim.jpg (entered 26.08.2016)

[5] R Studio's logo : https://www.rstudio.com/wp-content/uploads/2014/03/blue-250.png (entered 26.08.2016)

[6] ADXL Product Details : http://www.analog.com/en/products/mems/accelerometers/adxl345.html#product-overview (entered 27.08.2016)

[7] ADXL345 sensor image : http://www.wholeforshop.com/images/Consumer/58631_16914.jpg (entered 26.08.2016)

[8] Van Verth, James M.; Bishop, Lars M. (2008), Essential Mathematics for Games and Interactive Applications: A Programmer's Guide (2nd ed.), CRC Press, p. 7, ISBN 9780123742971. https://books.google.rs/books?id=zkEY9RIm4WkC&pg=PA7&redir_esc=y#v=onepage&q&f=false

[9] A Calculated Look at Fixed-Point Arithmetic by Robert Gordon : https://web.archive.org/web/20020611080806/http://www.embedded.com/98/9804fe2.htm (entered 27.08.2016)

[10] A Fixed-Point introduction by Example https://www.dsprelated.com/showarticle/139.php (entered 27.08.2016)

[11] SoCrates II logo of the board : https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=205&No=816 (entered 26.08.2016)

[12] Cyclone V SoC FPGA http://www.devboards.de/en/home/boards/product-details/article/socrates-ii/ (entered 26.08.2016)

9 Acknowledgments

I wish to express my sincere thanks to my supervisors Dr. Christian Faerber, Jonathan Machen and Jean-Christophe Garnier. The door to Christian's office was always open whenever I ran into a trouble spot or had a question about the FPGA board, project or writing. Sincere thanks to Jonathan for his motivation and support with the project matter. Without supervisor's passionate participation and input, the project could not have been successfully finished.

I take this opportunity to express gratitude to all of the LHCb team members and LHCb Secretariat for their help and support. I also thank my parents for the unceasing encouragement, support and attention.

I would like to express my gratitude towards CERN OpenLab team and fellow OpenLab summer students for their support.

I place on record, my sincere thank you to Dr. Miroslav Popović and Dr. Aleksandar Kovačević, Professors at Faculty of Technical Sciences, University of Novi Sad in Serbia, for the sincere and valuable encouragement extended to me.