Lehrstuhl für Integrierte Systeme
Fakultät für Elektrotechnik und Informationstechnik
Technische Universität München

Internship in Digital systems and FPGA engineering

Implementation of an FIR Filter on FPGA for Laser Line Extraction from Pixel Data

Mohammed Adib Obaid

Abstract

Current 3D laser line scanners have precision in the range of a micrometer. These scanners work on the principle of laser triangulation and use a camera chip in the receive path. The captured pixel data is then processed on an FPGA to generate 3D profile data. In order to do this, the laser line, as seen by the camera, must be extracted from the pixel data. For this purpose, several methods have been proposed. One of these methods employs an FIR filter to calculate the derivative of the incoming pixel stream orthogonally to the laser line direction. Afterwards, the zero crossing of this derivative is detected. The position of the zero crossing marks the position of the laser line in the camera image. From this position, the distance of the laser scanner to the scanned object can be derived.

This project was implemented using VHDL for the digital modules. In addition to that, Python tools were developed to assist in system modeling, simulation and test vector generation. The report highlights the proposed solution, an overview of the scripts and tools developed to assist in the system architecture design, a comprehensive look at each module, and finally a look at how the system can be improved.

Contents

1 Problem definition and the Proposed solution

1.1 3D Laser Line Scanners

3D laser line scanners work on the principle of laser triangulation and use a camera chip in the receive path. The setup shown in Figure 1.1 consists of a camera sensor with an angle α from the vertical line, and a laser line source directed on top of the object. When the laser is cast onto the object, the laser line is captured by the camera sensor.

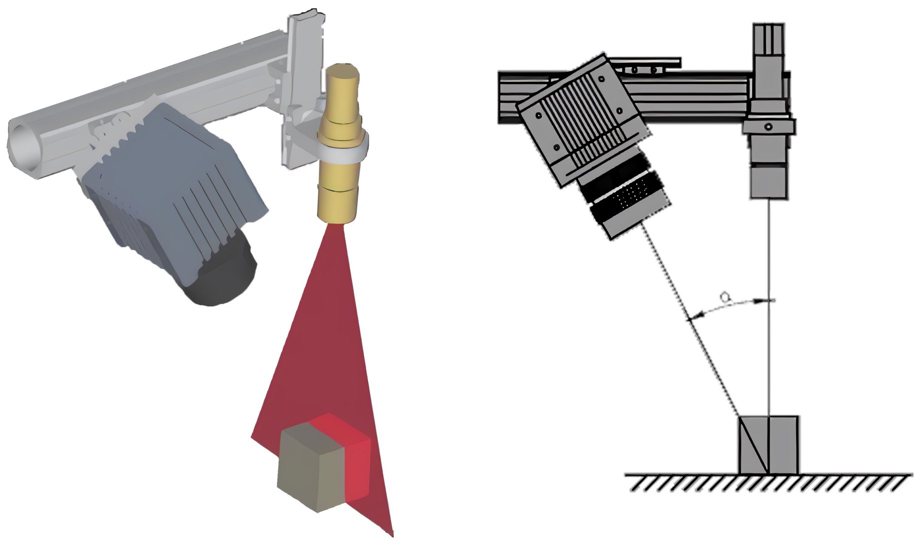


Figure 1.1: 3D laser line scanning setup [1]

The captured pixel data is then processed on an FPGA to generate 3D profile data. In order to do this, the laser line, as seen by the camera, must be extracted from the pixel data. For this purpose, several methods have been proposed to extract the line data from the captured image. One of these methods employs an FIR filter to calculate the derivative of the incoming pixel stream orthogonally to the laser line direction. Afterwards, the zero crossing of this derivative is detected. The position of the zero crossing marks the position of the laser line in the camera image. From this position, the distance of the laser scanner to the scanned object can be derived.

1.2 The proposed solution

Figure 1.2 shows pixel data of a laser line cast onto a spherical shaped object. As shown in the image, the reflected laser line from the scanned object is shown as a white line in

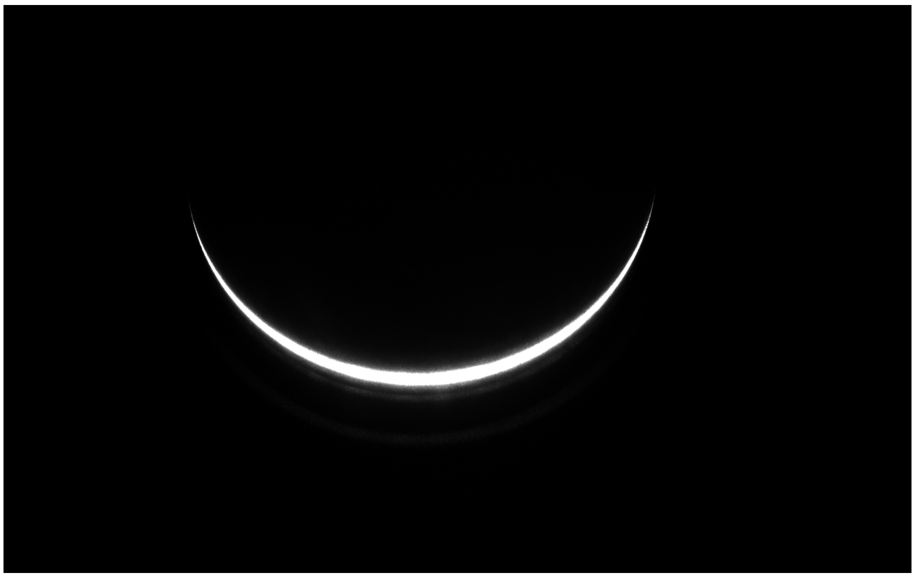


Figure 1.2: Pixel data of a laser line projected onto a spherical object

the pixel data.

The goal in this project is to extract the white line from the pixel data, and the proposed method is to process each vertical pixel line individually as shown in Figure 1.3. The extracted data is orthogonal to the laser line. The column data forms a gaussian distribution curve where the point of maximum intensity should point to the middle of the white laser line.

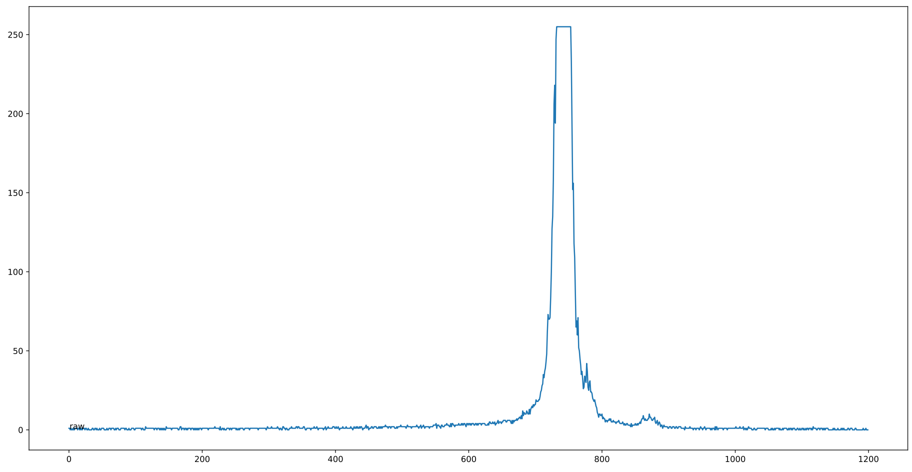


Figure 1.3: Pixel column distribution

There are several methods of exctracting the point of maximum intensity from a gaussian distribution curve. Figure 1.4 shows an overview of three maximum point finding methods [1]. The first method is looking for the point of maximum intensity of the incoming data. Second method is to accept any point with an intensity higher than a preset threshold. Afterwars, the center position between these points is selected [2]. Third method is to find the center of gravity of the area under the Gaussian curve [2]. In this project, a fourth method is explored, where the derivative of the gaussian curve formed by the pixel intensity is taken. A zero-crossing in the first derivative occurs at the same position of the extremum of the incoming pixel data.

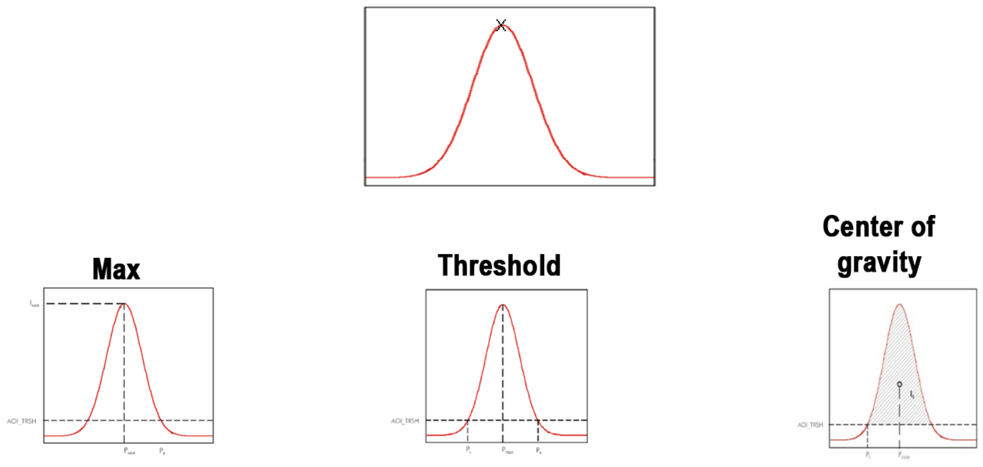


Figure 1.4: Extraction methods [1]

Figure 1.5 shows how the gaussian distribution curve would look like after taking its first derivative. The resulting differentiated dataset crosses from a positive value to a negative value at the point of maximum intensity.

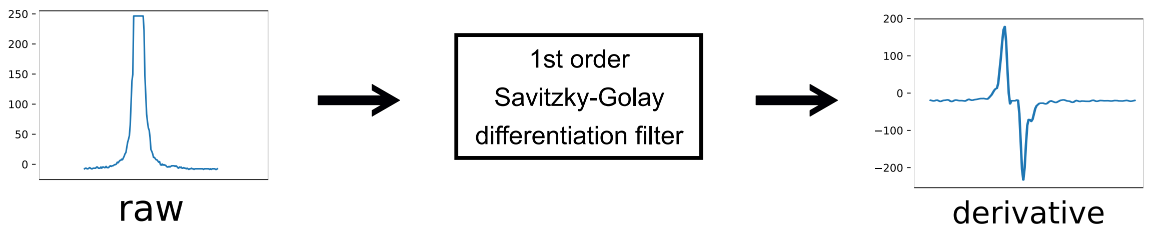


Figure 1.5: Image data to differentiated data

The processing system is composed of two main parts: The filters and the zero-crossing detector. The filtering part of the system has two jobs: it needs to smooth out the data and to differentiate the gaussian curve. While the zero-crossing part of the system will detect a positive to negative going zero-crossing position, this would mark the position of the laser line in the received pixel column.

1.2.1 FIR filters

Several filter setups has been experimented with during the initial system development phase. The FIR filter setup that was implemented in VHDL in this project consists of two FIR filters, a smoothing filter and a differentiation filter. Both filters are 5 taps Savitzky–Golay filters.

The filter sizes were chosen to limit the resource usage in the final design, as increasing the number of taps didn't provide enough extra smoothing to justify the extra FPGA area needed to implement it as shown in Figure 1.6

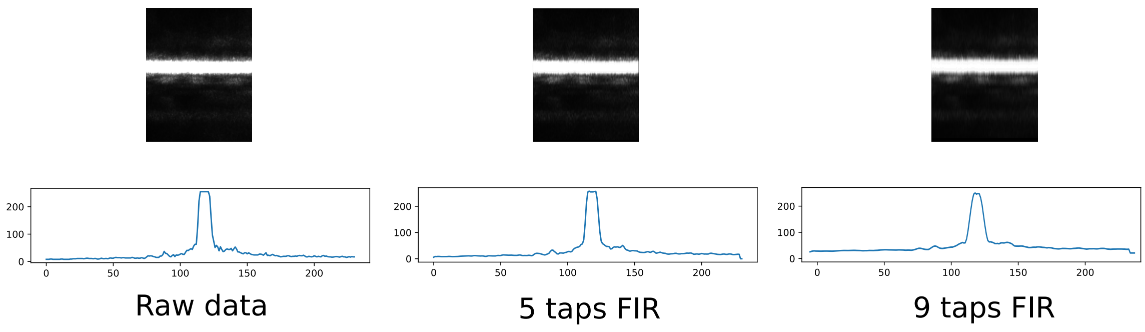


Figure 1.6: Smoothing filter tap comparison

The differentiation filter is a 5-tap first order Savitzky–Golay filter. Figure 1.7 shows how the data looks like after passing both filters, as shown in the third sub-plot there is a zero-crossing at the point of maximum intensity shown in the first and second subplots.

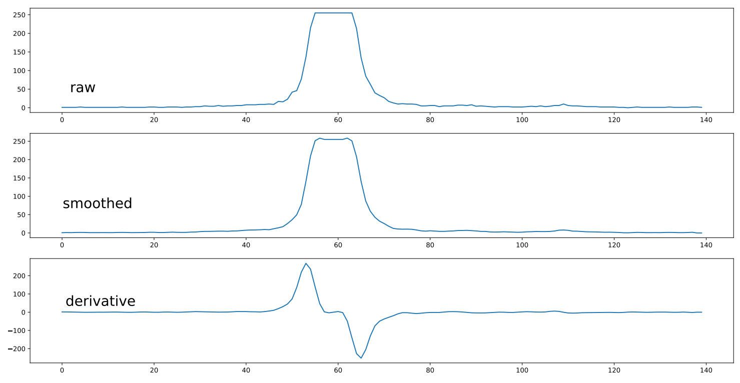


Figure 1.7: (a) Raw pixel data, (b) Smoothing filter output, (c) Differentiation filter output

Since pixel data is 8-bits wide and the current system does operations on signed 16-bit integers then there was no need for normalization for filter results. The filter coefficients are as shown in (1.1):

$$\begin{aligned} smoothingfilter &\Leftarrow \{-3, 12, 17, 12, -3\} \\ differentiationfilter &\Leftarrow \{2, 1, 0, -1, -2\} \end{aligned} \tag{1.1}$$

1.2.2 Zero crossing

There are several methods to find the position at which a positive-to-negative going zero-crossing has happened. One method is to search for the pattern $\{positive, 0, negative\}$, but the issue with this approach is that oftentimes the pixel data platues at the max value for several pixels before going down again which results in an extended amount of zeros at the zero-crossing as shown in Figure 1.8

A solution to this problem would be supporting varrying numbers of zeros in the middle of the matching pattern. This solution is not robust for an FPGA application as it requires going through the same set of data multiple times.

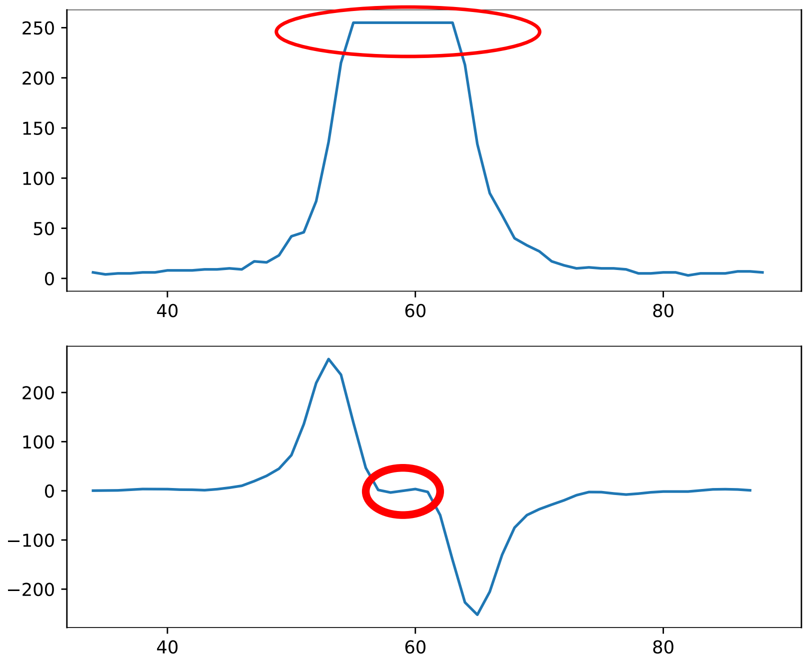


Figure 1.8: Several zero values at the zero-crossing position

Another method to detect zero-crossings is to find the maximum value and the minimum value, ensure that the max value's index is less than the min value's index, then find the index in the middle. Generally this method provides accurate results for images that aren't very noisy.

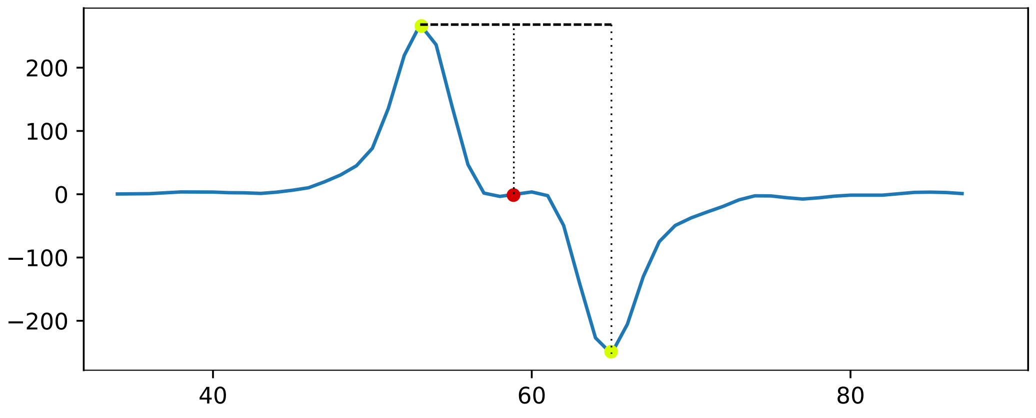


Figure 1.9: Zero-crossing at midpoint of two peaks

Figure 1.9 shows how the zero-crossing can be located using the midpoint between the two peaks. Eq.(1.2) shows the equation used to find the zero-crossing index. Max_i and Min_i are the indicies of the maximum and minimum values in the dataset respectively.

$$Zero_i = Max_i + \frac{Min_i - Max_i}{2} \tag{1.2}$$

2 Repository structure

2.1 Repository folder structure

The repository is divided into folders that correspond to the function of the files within it. Figure 2.1 shows how the repository is structured.

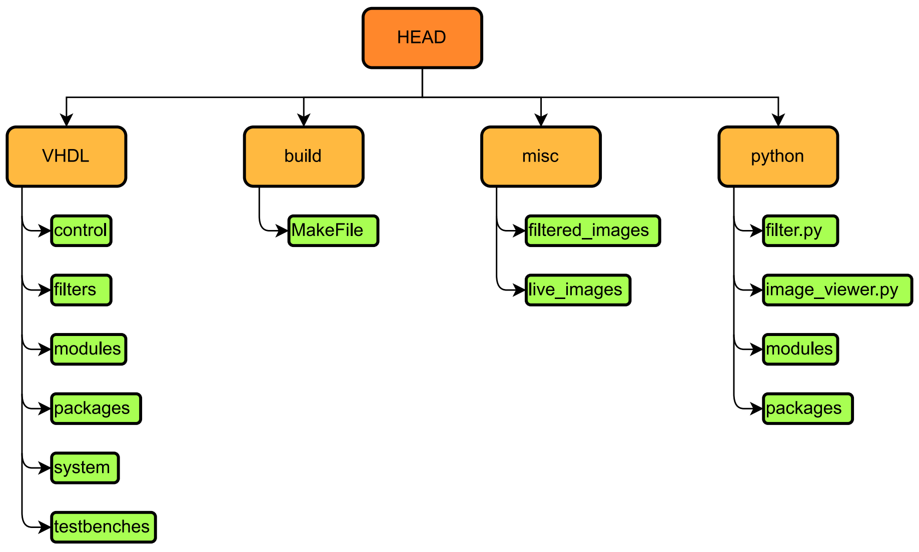


Figure 2.1: Repository structure

2.2 Python tools

2.2.1 Filter class

For this project, a filter class was written in Python to simulate the behaviour of a hardware filter. The filter class also includes methods to process the incoming data such as convolution.

Listing 2.1: FIR filter class methods

```python
class FIRFilter:
    def convolve(self, data_stream)
    def detect_zero_crossings(self, data)
    def cull_data(self, data, threshold_factor)
    def _find_min_max(self, sequence)
```

2.2.2 GUI viewer

The image viewer is a helpful tool to visualize the processed data from the camera sensor. Figure 2.2 is a screenshot of the image viewer and its structure.

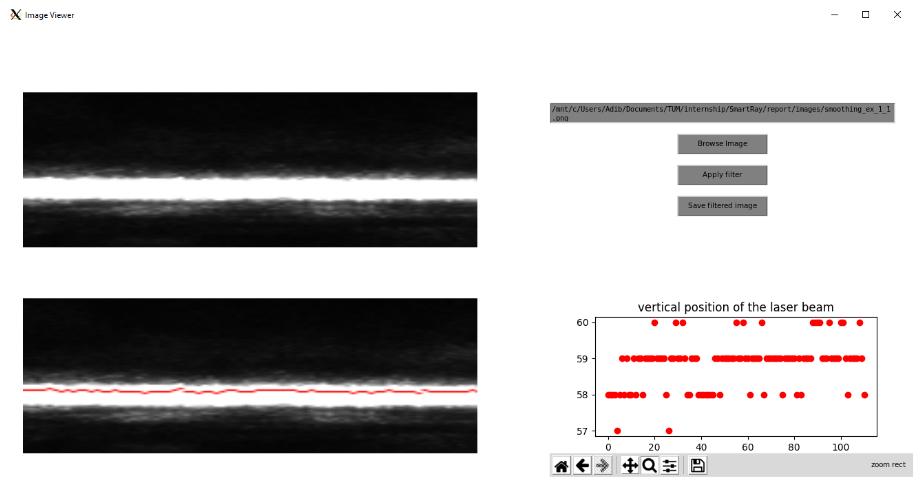


Image Viewer
/mnt/c/Users/Adib/Documents/TUM/internship/SmartRay/report/images/smoothing_ex_1_1.png
Browse Image
Apply filter
Save filtered image
vertical position of the laser beam
60
59
58
57
0
20
40
60
80
100
zoom rect

Figure 2.2: Image viewer tool

The viewer is built using Python and TKinter with TkAgg as a rendering backend. The program is built using a mixture of OOP and functional programming paradigms. This is because of how TKinter widgets interact with each other.

Image viewer accepts only grayscale images as shown in Figure 2.2. In addition, the Image Viewer has slow processing time for large images because the processing is done sequentially, as concurrency isn't really optimal in Python since it doesn't provide significant speedup.

2.2.3 VHDL test vector generation scripts

Two scripts were written to assist in executing VHDL testbenches. `test_vector_file_gen.py` & `test_vector_file_data_viewer.py` are the scripts in the repository. `test_vector_file_gen` reads image data from the `misc` folder and exports a `.txt` file with the correct fomatting for the `CTRL` and `DATA` signals. Those two signals are the inputs of the system, `CTRL` is a 2-bit control signal (see Table 2.1) and `DATA` is an 8-bit pixel value.

2.3 VHDL code and hardware implementation

VHDL modules within the repository are divided into their functional behaviour. Development of the VHDL system modules followed a structural design pattern, thus all of the basic digital components has been written in VHDL to ensure consistency in behaviour and avoid synthesizer ambiguity. In this section, only the major system modules will be explained. The rest are shown in Appendix A.

2.3.1 FIR filter

FIR filters are generally easy to implement in VHDL as they are only composed of multipliers, adders and basic registers. On the other hand, the decision of which architecture to implement comes with tradeoffs that depend on the general goal of the filter design. Figure 2.3 shows 4 possible filter architectures Direct FIR, Transposed FIR, pipelined Transposed FIR and Folded Direct FIR filters [3].

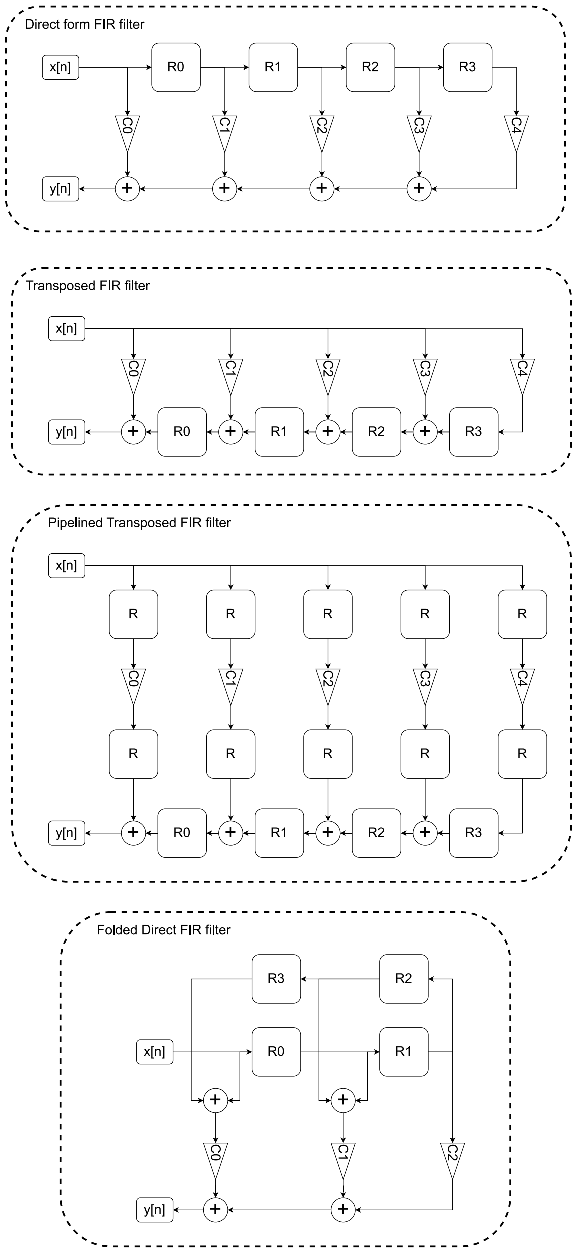


Figure 2.3: (a) Direct FIR, (b) Transposed FIR, (c) Pipelined Transposed FIR, (d) Folded Direct FIR

Direct FIR filter is the basic version of FIR filters, it is what would be basically seen in a DSP hard IP core. There is a major design shortcoming which is a very long combinational critical path from the multipliers to the output through adders as shown in Figure 2.4

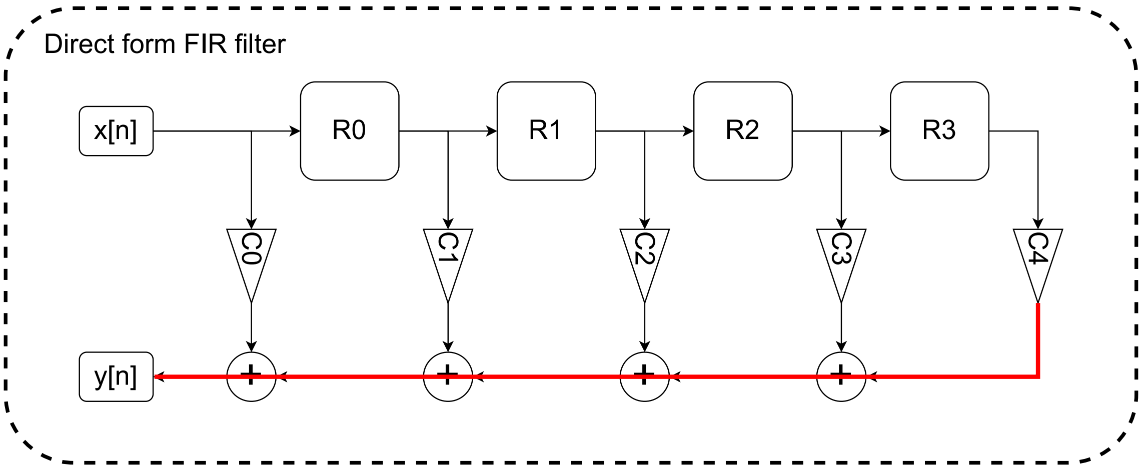


Figure 2.4: Critical path in Direct FIR architecture

The critical path delay as shown in (2.1) highlights the problem with the Direct FIR design. Where N is the size of the filter.

$$T_{crit} = T_{mul} + (N - 1) * T_{adder} \tag{2.1}$$

On the other hand, the Transposed FIR filter design breaks down the critical path as shown in Figure 2.5. The critical path delay is $T_{crit} = T_{mul} + T_{adder}$

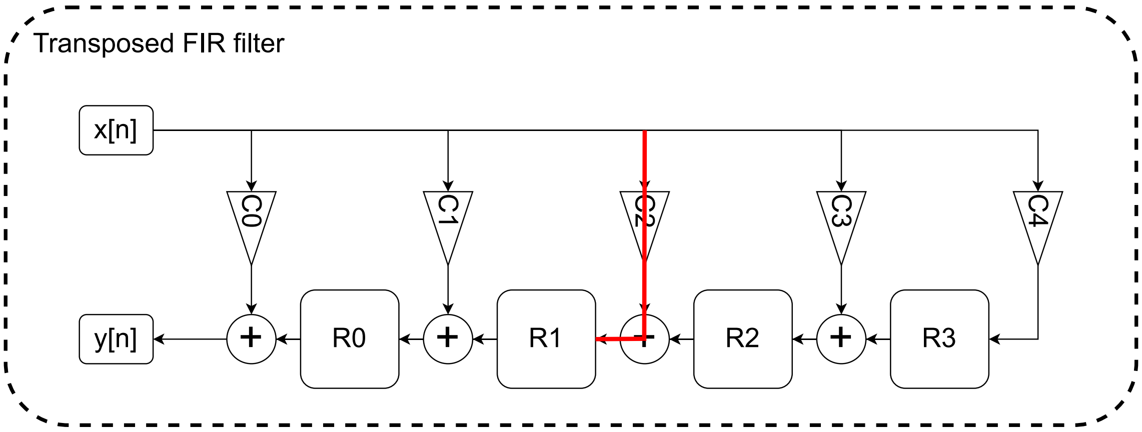
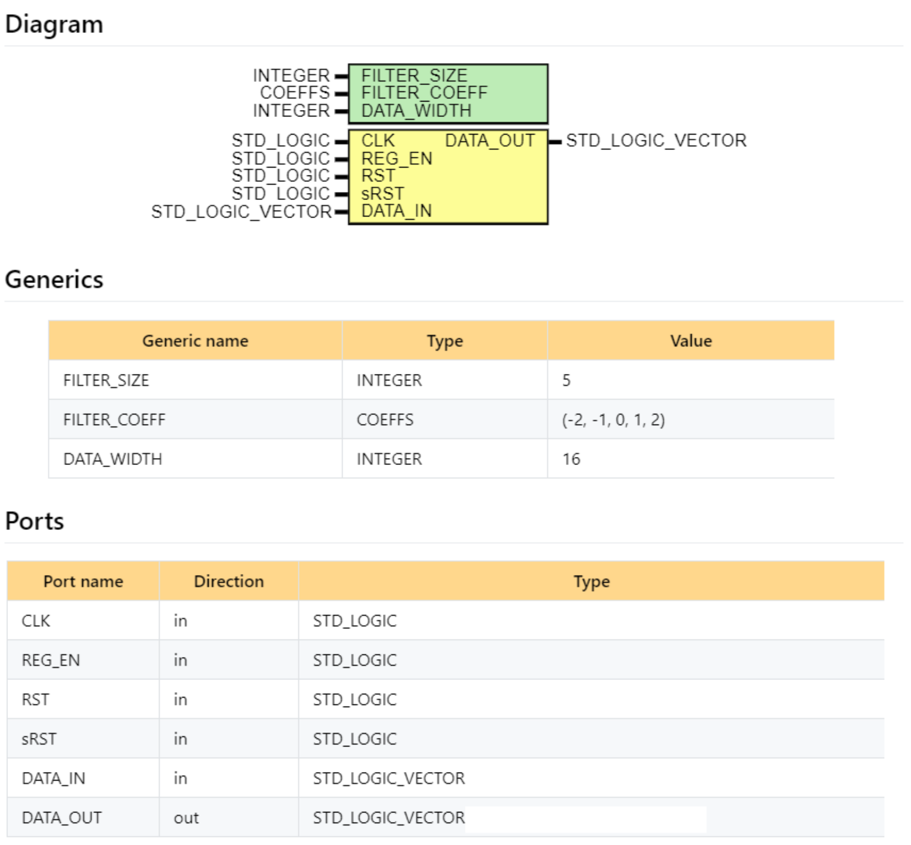


Figure 2.5: Critical path in Transposed FIR architecture

The pipelined transposed FIR architecture breaks down the critical path further into being T_{mul} but introduces $2 * N$ more registers, where N is the filter size. Also, a bigger fanout is introduced to the $X[n]$ signal as well. The introduced trade-off in the design area is a decision that has to be made. During this project, the transposed FIR filter architecture was the one implemented.

The VHDL implementation was done using generate statements which take into account the filter generic. Figure 2.6 shows extensive details about the module implementation in VHDL. The data was generated using TerosHDL in vscode.

As shown in Figure 2.6, the filter coefficients, filter size and the input/output data width are generics which can be specified at module instantiation.



Diagram

Generics

Generic name	Type	Value
FILTER_SIZE	INTEGER	5
FILTER_COEFF	COEFFS	(-2, -1, 0, 1, 2)
DATA_WIDTH	INTEGER	16

Ports

Port name	Direction	Type
CLK	in	STD_LOGIC
REG_EN	in	STD_LOGIC
RST	in	STD_LOGIC
sRST	in	STD_LOGIC
DATA_IN	in	STD_LOGIC_VECTOR
DATA_OUT	out	STD_LOGIC_VECTOR

Figure 2.6: FIR filter HDL specs

2.3.2 Zero crossing module

The zero-crossing algorithm is fairly simple and straight-forward to implement, since the implementation of the algorithm depends on the module finding the maximum and minimum values in the pixel column. Therefore, a peak-detector module was written in VHDL. The peak-detector clocks out the index of the peak value whether it was a maximum or a minimum peak.

After the indicies of both peaks has been acquired, another module ensures that the index of the maximum value is less than the index of the minimum value, because we are looking for a positive to negative going zero crossing. The module ensures that by clocking out both indicies if the values are correct, otherwise it clocks out zeros.

Finally, the zero crossing index is calculated according to (1.2). The zero index is then clocked out of the module. Figure 2.7 shows the structure of the zero-crossing module. The abundant registers are there for delay matching.

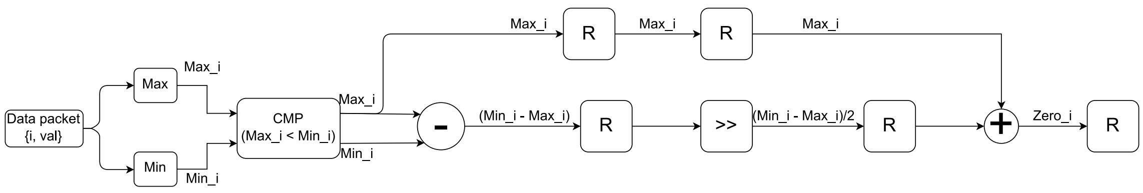
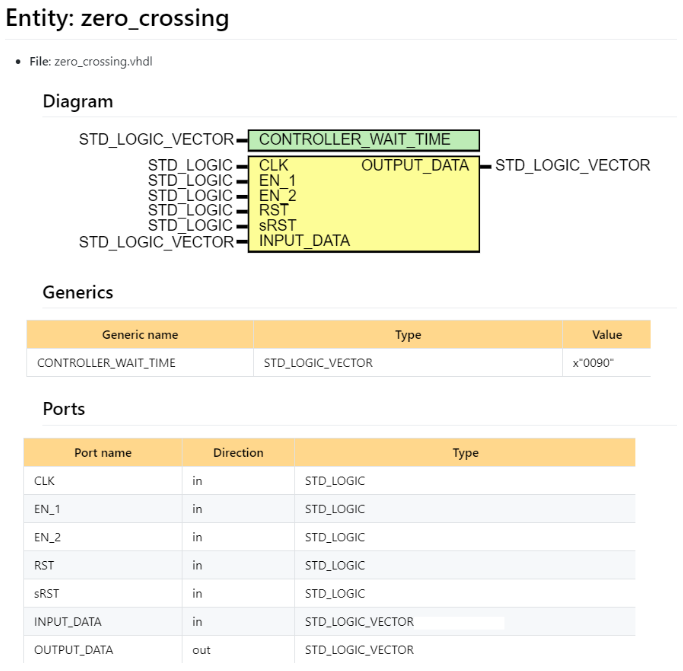


Figure 2.7: Zero-crossing structural view

Figure 2.8 shows the specifications of the zero crossing module.



Entity: zero_crossing

- File: zero_crossing.vhdl

Diagram

Generics

Generic name	Type	Value
CONTROLLER_WAIT_TIME	STD_LOGIC_VECTOR	x"0090"

Ports

Port name	Direction	Type
CLK	in	STD_LOGIC
EN_1	in	STD_LOGIC
EN_2	in	STD_LOGIC
RST	in	STD_LOGIC
sRST	in	STD_LOGIC
INPUT_DATA	in	STD_LOGIC_VECTOR
OUTPUT_DATA	out	STD_LOGIC_VECTOR

Figure 2.8: Zero-crossing HDL specs

2.3.3 Input controller

The last part of the 3 major system modules is the input controller. The input controller takes the 8-bit pixel data and converts it to 16 bit. It also takes two control bits which specify the type of the data it is receiving. Table 2.1 shows what each `CTRL` value corresponds to in terms of package type.

CTRL	TYPE
0b00	IDLE
0b01	START
0b10	DATA
0b11	STOP

Table 2.1: Input controller control bits

The input controller is basically a state machine with the enable and reset signals of the system modules as its outputs. Figure 2.9 shows the specifications of the input controller. The module also has generics that control the timings of the output pins, the enable outputs go logic ’0’ when the module is flushed out of useful data.

The state machine depends on an internal counter that counts the clock cycles. Figure 2.10 shows the state diagram of the input controller’s state machine.

During the `STOPPING` state, there are several conditions to check against the internal counter to disable the modules sequentially depending on the time it takes to flush those modules out.

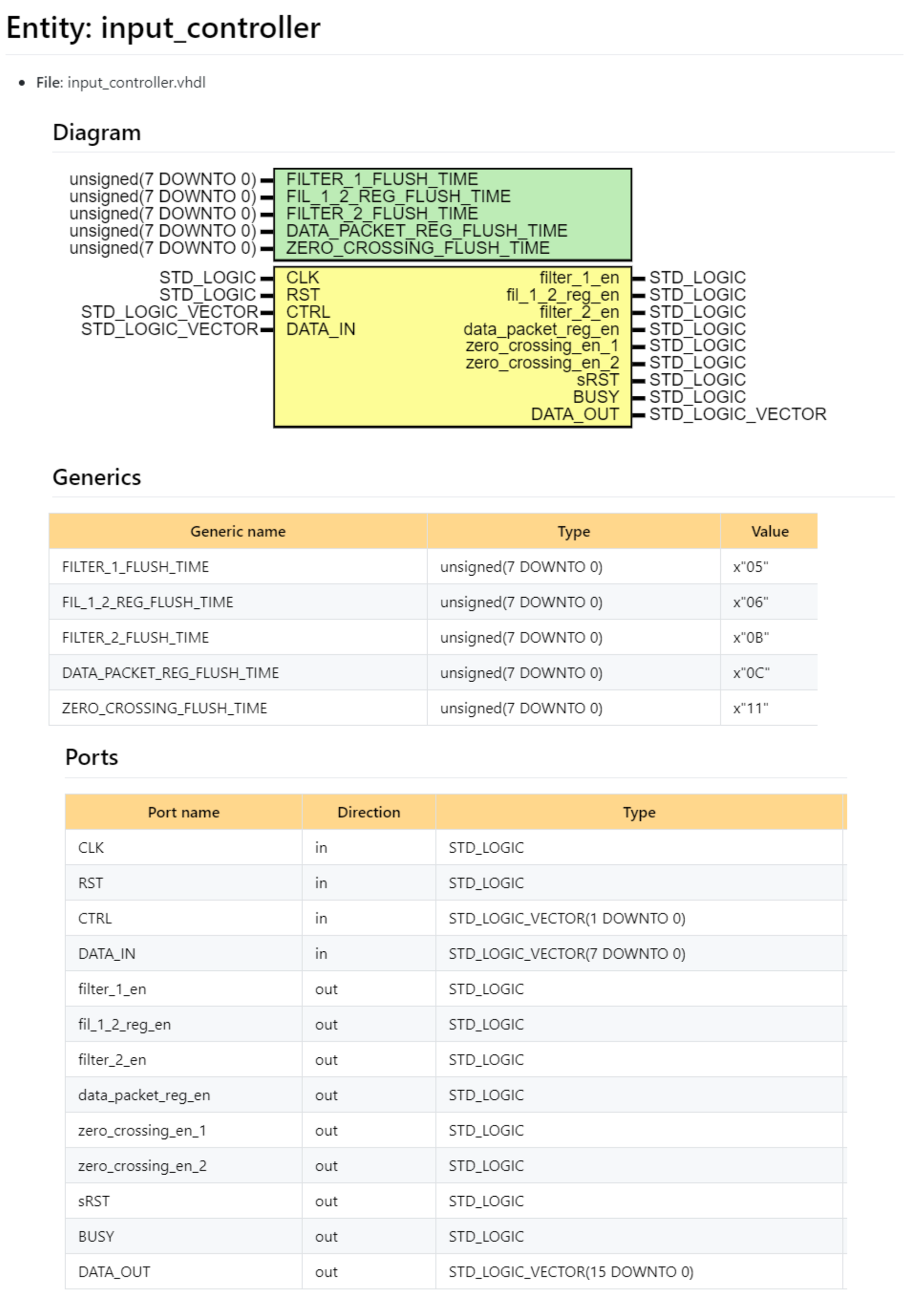
2.3.4 TOP module

The TOP module is just a structural combination of all the aforementioned sub-system modules. Figure 2.11 shows the block diagram of the system. The index counter module is basically a counter that starts counting after a specified delay. This module is used to combine the delayed output data from the filters with their indicies so they can be used in the zero crossing module.

2.4 Build system

The VHDL code analysis, elaboration and simulation are all done using GHDL. The main appeal of the tool is the ability of working with VHDL modules from the terminal. GHDL allows the user to generate a wave `.vcd` file, the wave file can be viewed using GTK-Wave.

To automate the process of code analysis, elaboration, and simulation Make was used. The MakeFile is located in the build folder in the repository where all the compiled binaries are located as well. The MakeFile also provides some other functionality to the user such as viewing the wavefiles in GTKWave.



Entity: input_controller

- **File:** input_controller.vhdl

Diagram

Generics

Generic name	Type	Value
FILTER_1_FLUSH_TIME	unsigned(7 DOWNTO 0)	x"05"
FIL_1_2_REG_FLUSH_TIME	unsigned(7 DOWNTO 0)	x"06"
FILTER_2_FLUSH_TIME	unsigned(7 DOWNTO 0)	x"0B"
DATA_PACKET_REG_FLUSH_TIME	unsigned(7 DOWNTO 0)	x"0C"
ZERO_CROSSING_FLUSH_TIME	unsigned(7 DOWNTO 0)	x"11"

Ports

Port name	Direction	Type
CLK	in	STD_LOGIC
RST	in	STD_LOGIC
CTRL	in	STD_LOGIC_VECTOR(1 DOWNTO 0)
DATA_IN	in	STD_LOGIC_VECTOR(7 DOWNTO 0)
filter_1_en	out	STD_LOGIC
fil_1_2_reg_en	out	STD_LOGIC
filter_2_en	out	STD_LOGIC
data_packet_reg_en	out	STD_LOGIC
zero_crossing_en_1	out	STD_LOGIC
zero_crossing_en_2	out	STD_LOGIC
sRST	out	STD_LOGIC
BUSY	out	STD_LOGIC
DATA_OUT	out	STD_LOGIC_VECTOR(15 DOWNTO 0)

Figure 2.9: Input controller HDL specs

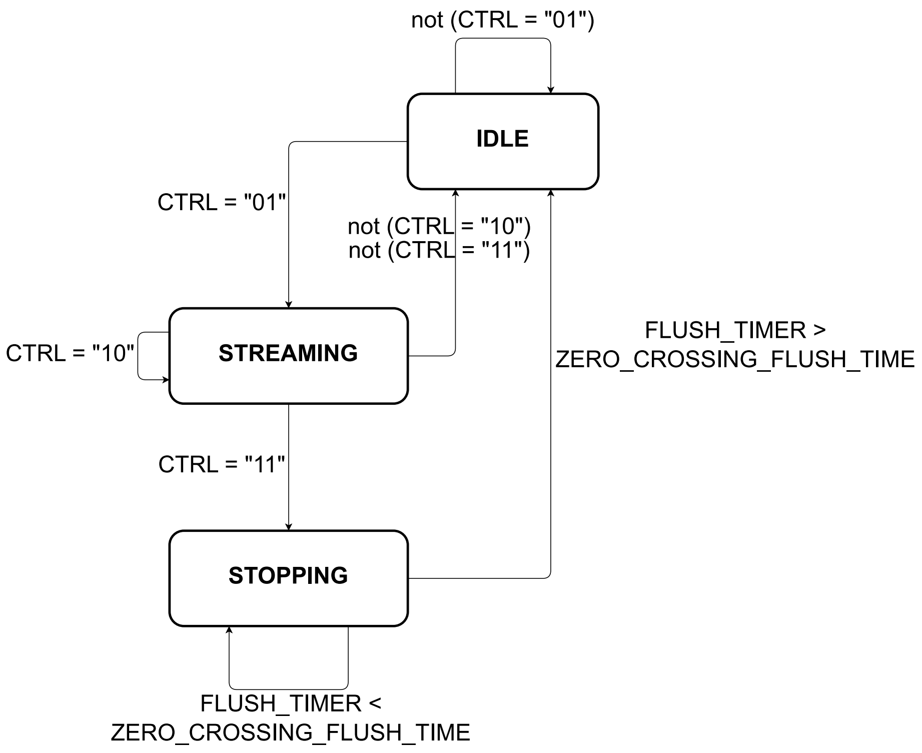


Figure 2.10: Input controller state machine

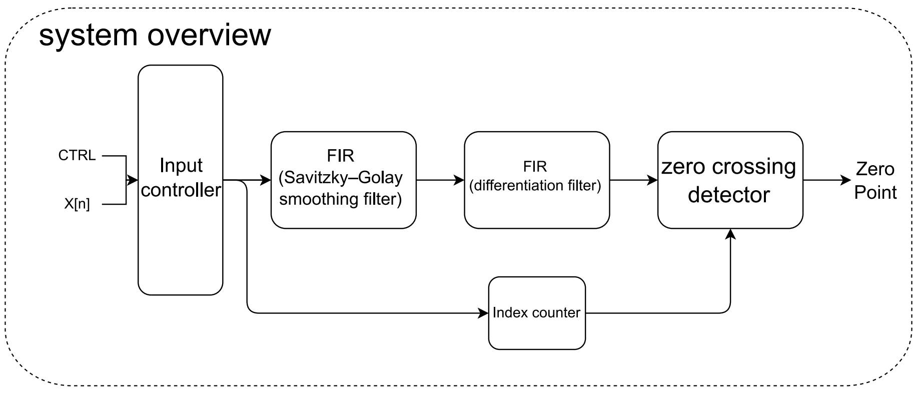


Figure 2.11: Block diagram of TOP

3 Conclusion

3.1 Results discussion

The system processes data from the image sensor one pixel column at a time. In each processed pixel column the zero-crossing module will find an index where the zero-crossing happens. This index is the location of the point of maximum intensity on the laser line. Figure 3.1 shows the result of passing an image through the system. The red line highlights the points of maximum intensity detected by the system.

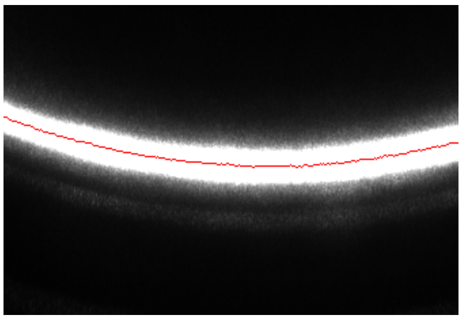


Figure 3.1: Example processing result

The system produces accurate results when the images are not noisy nor highly exposed. The fix for noise was addressed by the use of a smoothing filter. But the high exposure produces high intensity patches in the image as shown in Figure 3.2. The patches skew the result of the zero-crossing detector as well. This is due to the fact that the patches are of saturated intensity which makes it hard for the smoothing filter to get rid off.

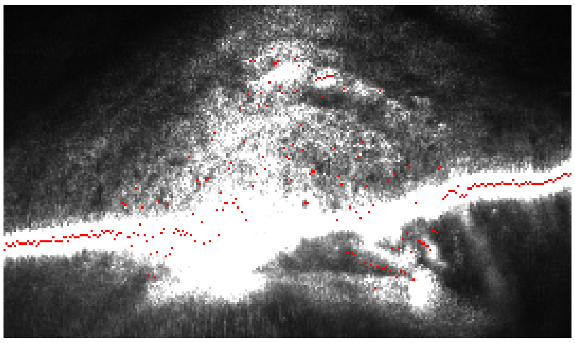


Figure 3.2: Skewed results due to over-saturated areas

Figure 3.3 shows how the pixel column looks like when the image is exposed for too long. It also shows the detected index is shifted due to the high distortion in the image. A solution to this problem would be to adjust the exposure time according to the material being scanned.

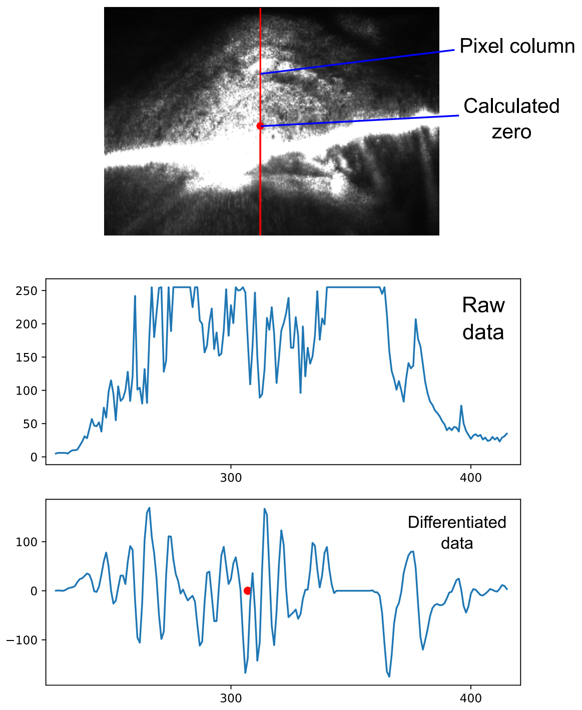


Figure 3.3: Highly exposed image next to graph of raw and differentiated data

3.2 Next steps

3.2.1 System optimization

The design has been implemented in VHDL and its functional behavior has been verified. Afterwards, the design was synthesized for an Intel Cyclone V and achieved a maximum frequency of 228 MHz. However, further speed improvements can be done to increase the clock frequency further. Such as the use of approximated multipliers.

3.2.2 Data threshold

The differentiation filter and the zero-crossing algorithm work better when the noise is eliminated completely from the input pixel data. A method which was tested in the Python script was to cull the data which is below a certain threshold, 80% of the maximum value is a dataset provided very good results. The issue with this method is that the maximum value is not known beforehand which means that the system needs to read all the values first then calculate 80% of that value then cull the data points below that value. This introduces large latencies to the system. So, a set threshold or a dynamic threshold could be used on a frame-to-next-frame basis.

List of Figures

Bibliography

[1] D. A. Klipfel, “Improving the 3d scan precision of laser triangulation,” Automation Technology GmbH, 2016, accessed on: Dec., 16, 2021. [Online]. Available: https://www.visiononline.org/userAssets/aiaUploads/file/25-Improvingthe3DScanPrecisionofLaserTriangulation-Dr_AthinodorosKlipfel.pdf

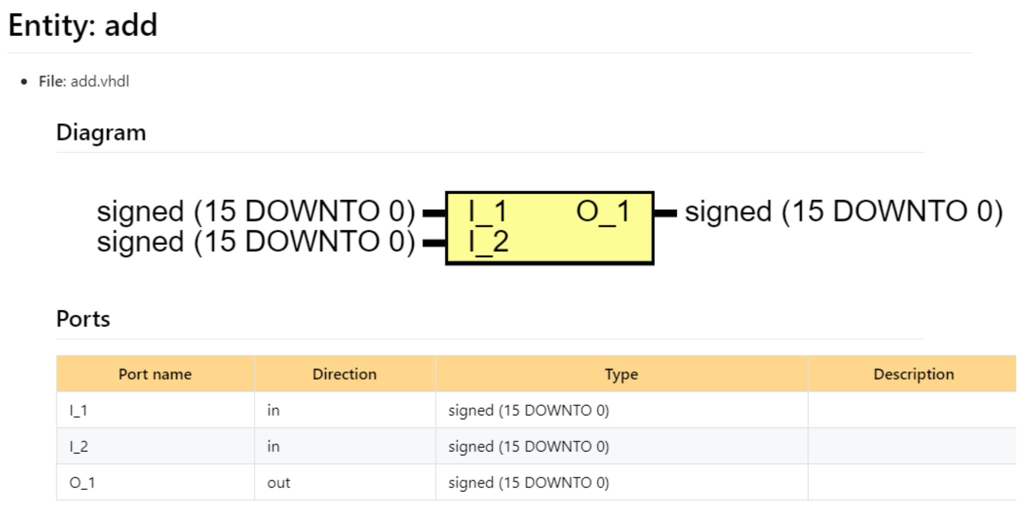
[2] P. Fasogbon, L. Duvieubourg, and L. Macaire, “Fast laser stripe extraction for 3d metallic object measurement,” in *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*, 2016, pp. 923–927.

[3] A. Akif, “Fir filter features on fpga,” B.S. Thesis, Department of Electrical Engineering, Linköping University, Linköping, 2018, accessed on: Dec., 16, 2021. [Online]. Available: https://liu.diva-portal.org/smash/get/diva2:1256720/FULLTEXT01.pdf

Appendix A

1 VHDL module specifications

1.1 Adder



Entity: add

- File: add.vhdl

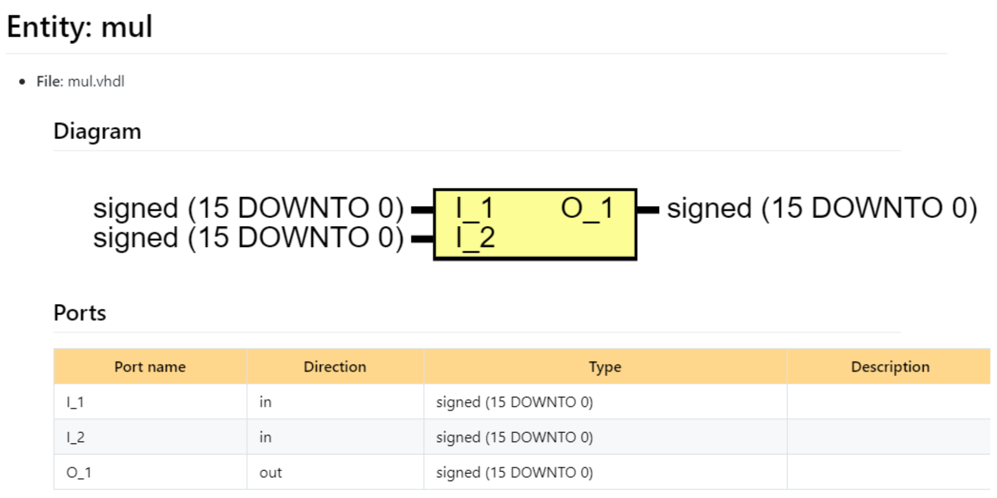
Diagram

Ports

Port name	Direction	Type	Description
I_1	in	signed (15 DOWNTO 0)	
I_2	in	signed (15 DOWNTO 0)	
O_1	out	signed (15 DOWNTO 0)	

Figure 1: Adder specifications

1.2 Multiplier



Entity: mul

- File: mul.vhdl

Diagram

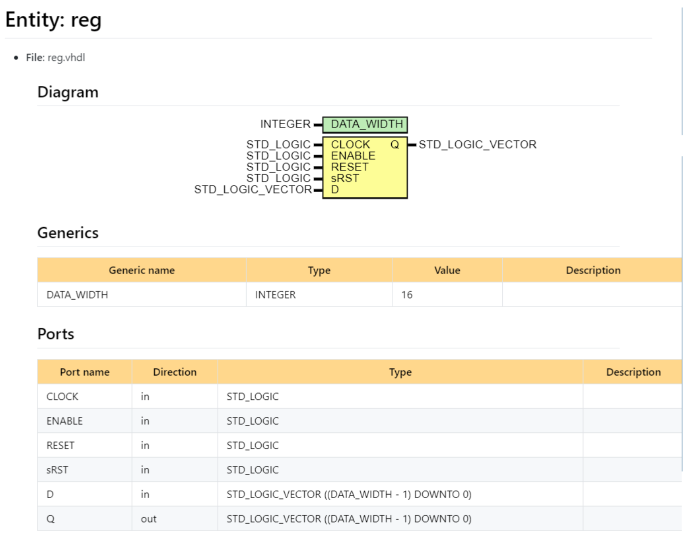
Ports

Port name	Direction	Type	Description
I_1	in	signed (15 DOWNTO 0)	
I_2	in	signed (15 DOWNTO 0)	
O_1	out	signed (15 DOWNTO 0)	

Figure 2: Mul specifications

1.3 D-FlipFlop

The D-FlipFlop supports both synchronous and asynchronous resets.



Entity: reg

- **File:** reg.vhdl

Diagram

Generics

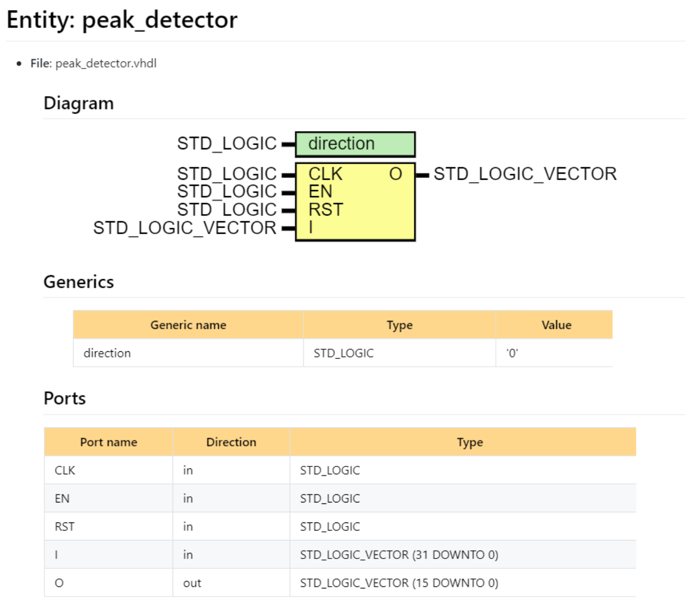
Generic name	Type	Value	Description
DATA_WIDTH	INTEGER	16	

Ports

Port name	Direction	Type	Description
CLOCK	in	STD_LOGIC	
ENABLE	in	STD_LOGIC	
RESET	in	STD_LOGIC	
sRST	in	STD_LOGIC	
D	in	STD_LOGIC_VECTOR ((DATA_WIDTH - 1) DOWNTO 0)	
Q	out	STD_LOGIC_VECTOR ((DATA_WIDTH - 1) DOWNTO 0)	

Figure 3: FlipFlop specifications

1.4 Peak Detector



Entity: peak_detector

- **File:** peak_detector.vhdl

Diagram

Generics

Generic name	Type	Value
direction	STD_LOGIC	'0'

Ports

Port name	Direction	Type
CLK	in	STD_LOGIC
EN	in	STD_LOGIC
RST	in	STD_LOGIC
I	in	STD_LOGIC_VECTOR (31 DOWNTO 0)
O	out	STD_LOGIC_VECTOR (15 DOWNTO 0)

Figure 4: Peak Detector specifications