Introduction to Image Processing Using VHDL

M. Tuna Alatan

Bilkent University, EEE Department

August 2024

Contents

1 Introduction

This report is prepared to summarize a three-week internship that has been conducted at Artron Inc.. After giving a brief motivation about the internship, the first some of the main blocks of VHDL are presented. These sub-modules include block RAM construction, timing signal generation, test pattern generation and getting image data from a text file.

Before diving into the report, it is also worth mentioning that the code discussed in this report with the helper scripts can be accessed through the following link: `https://github.com/tuna-alatan/Intro-Image-Processing-VHDL`.

In the next part of this report, the image processing algorithm implementations are presented, as well as some typical results from these algorithms.

1.1 Motivation

VHDL is a very important tool to convert any signal processing or machine learning algorithm into hardware using embedded programming. In order to obtain a product or a prototype out of an algorithm in an economically feasible manner, these algorithms have to be implemented in hardware by the help of VHDL. Although this kind of approach is very attractive from an engineering and economical point of view, the conversion of the algorithms using VHDL is a quite tedious and time-consuming process.

The motivation for this internship is to get more experience on this issue by working with real-life problems and simulators. Although VHDL is taught in the first-year curriculum of Bilkent EEE, this internship aims to improve and digest some VHDL topics in depth.

2 Block RAM Construction Using VHDL

Block RAM (or BRAM) stands for Block Random Access Memory; it is used for storing data on the FPGA board. The data is stored in an array-shaped data structure where every data has a specific address. Users can access the desired data using the corresponding address.

In VHDL a block RAM can be implemented using two main components: `bram` and `bram_ctrl`.

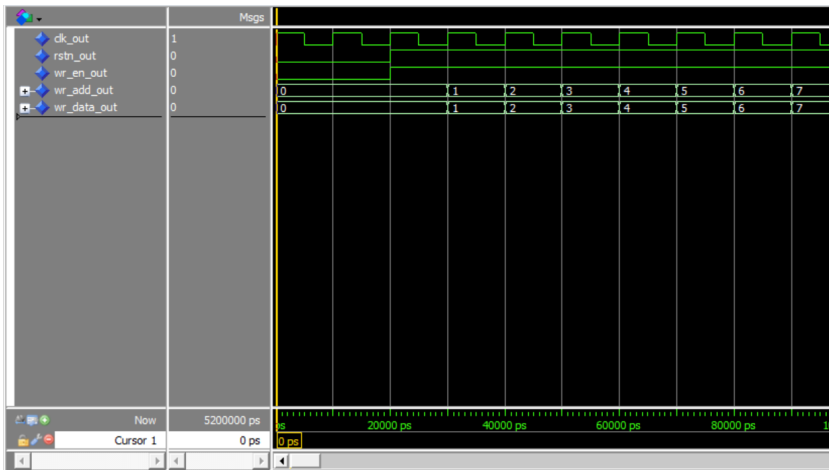
The `bram` component acts as the data structure. An array with the specified data size and count is constructed in this component. Each element of this array is the data and the index of each element is the address of the data. The component has write enable and read enable signals as inputs. If the write enable signal is high, the data (`wr_data`) is written to the write address (`wr_add`). If the read enable signal is high, the data on the read address (`rd_add`) is assigned to the `rd_data` output signal.

The `bram_ctrl` component controls the date flow of the BRAM. Write enable, address, data and read enable, address signals are generated as output in this component and the component takes read data as input.

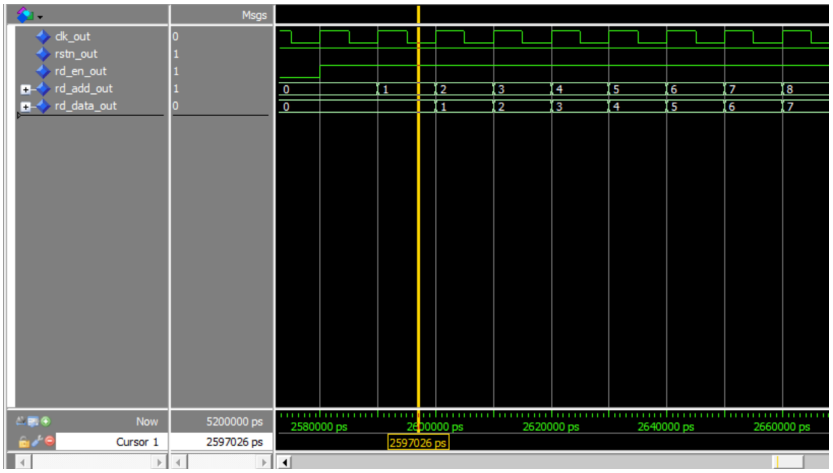
The components are connected directly to each other with `wr_en`, `wr_add`, `wr_data`, `rd_en`, `rd_add`, `rd_data` signals. The output of one is the input of the other.

This module in particular first writes the address number of each address as data and then starts to read every address starting from 0.

Figure 1 is a screen-shot from ModelSim which is the simulation application that is used in this internship, screen-shots from the simulation is used throughout this report to present the function of the modules. For example, as can be observed from Figure 1b when sending the read address the data on the corresponding address is sent on the following cycle, this property of BRAM is crucial for upcoming modules.



(a) The simulation results for the writing operation of the BRAM.



(b) The simulation results for the reading operation of the BRAM.

Figure 1: The simulation results for BRAM module.

In the next section, another main block of VHLD is presented to understand the timing process in such systems.

3 Frame Timing Signal Generation

The available storage on an FPGA board is usually limited and for this reason, a full resolution frame is not typically stored on the device. Each pixel data is flowing into the device and the computations are done simultaneously. In order to understand when each pixel is sent and when a whole frame is sent timing signals are used.

There are two main signals: `d_val` and `f_val`, which are explained below:

`d_val` (data valid) is the signal that represents whether the pixel is active or not. For this project, each pixel will be high for one clock period and low for a fixed amount of time between each pixel. Moreover, between each frame the downtime will be the aforementioned fixed amount of time. Thus, this downtime can be calculated according to the resolution of the image, the clock frequency and the fps (frames per second).

`f_val` (frame valid) just like the data valid signal, it is high for the duration of a frame and low for a fixed amount of time.

In Figure 2 only 2 peaks can be observed in the signal `d_val` which means that only five rows have been generated. The distance between each respective falling and rising edge of the signal `d_val` is a fixed amount of time and this distance is also used after a rising edge of `f_val` and before a falling edge of `f_val`.

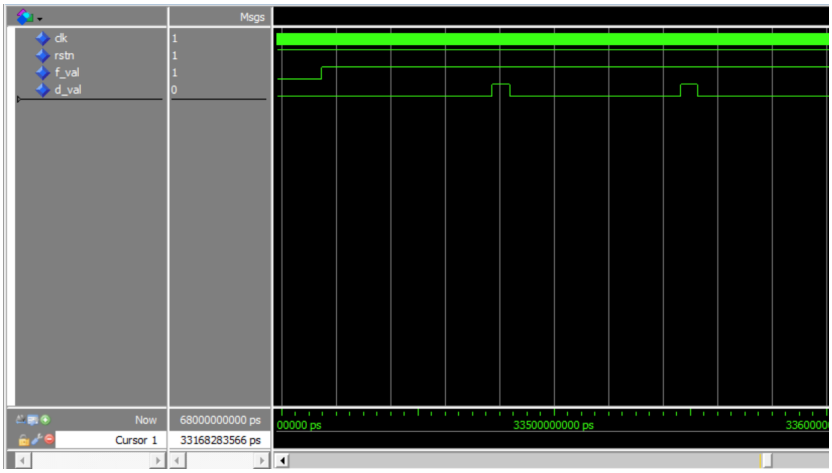


Figure 2: The data and frame valid signals on top of each other.

There are 2 dips in the `f_val` signal in Figure 3. Their downtime length is the fixed amount that is discussed previously. There are 480 peaks of the signal `d_val` between those dips which the height of the image.

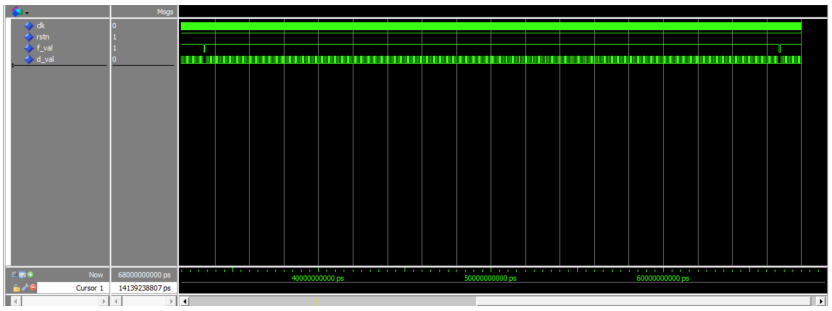
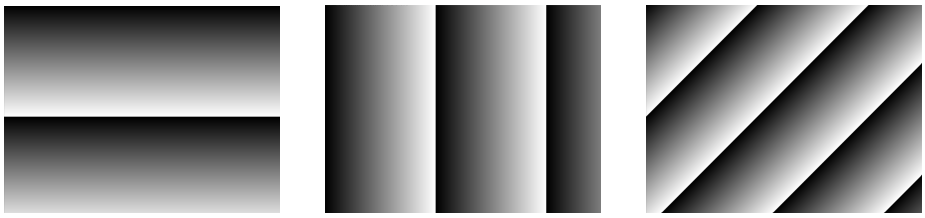


Figure 3: The timing signals of an one whole frame.

4 Test Pattern Generation

After generating the timing signals, the next step is to generate an image. In order to test the image processing operations three main test patterns are generated. Each test pattern has its own module and they are graycale images. There are three main test patterns: horizontal (4a), vertical (4b) and diagonal (4c). Test patterns are generated by assigning pixel values when `d_val` signal is high. The test pattern components have three main outputs: `d_val`, `f_val` and `v_data`. `d_val` and `f_val` are delayed versions of the input timing signals and `v_data` is the value of the pixel when the pixel is active.



(a) (b) (c)

Figure 4: Test patterns.

5 Image From Text File

In some cases, the test patterns are not enough for testing the image processing operations, using random images from the internet as an input image is also crucial for testing the modules. This module cannot be synthesized so it can only work on simulations.

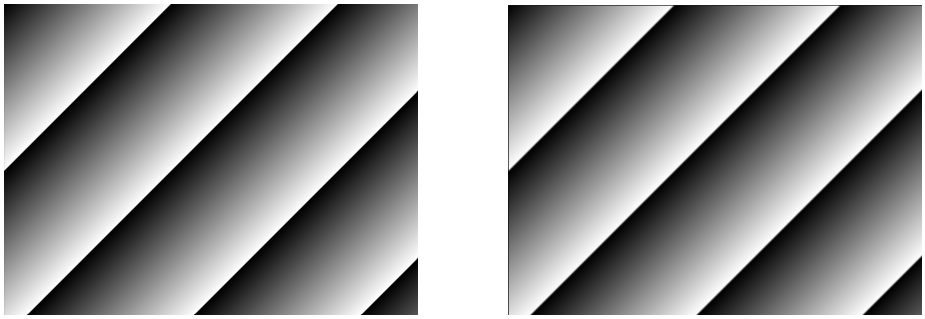
The desired image with a widely used image format (.png, .jpeg, etc.) is first converted to a text file by a python script. The text file is then added to the directory of the simulation, this way the module can use the image. The module uses the `STD.textio` library for handling the file operations.

6 Fundamental Image Processing Operations

After defining the fundamental blocks of VHDL, this section is devoted to presenting image processing algorithms that have been implemented using VHDL.

6.1 Basic Low Pass Filtering

In image processing, filtering is the process of applying a filter to an image in order to change its pixel values according to a specified neighborhood or kernel, usually to improve features, minimize noise, or extract particular details. In particular a low pass filter is used for blurring images, which is the filter that is used in this project. The result of the low pass filter can be better observed in Figure 5. Since the kernel size is small the effect is subtle but it can still be observed.



(a) Before low pass filter. (b) After low pass filter

Figure 5: The effect of the low pass filter.

A kernel is a small matrix that slides over an image, and for each position, it multiplies its values with the corresponding pixel values in the image, then sums them up to produce a new pixel value. For different goals different coefficients are used, in a low pass filter all coefficients are $\frac{1}{n}$ where n is the total number of coefficients.

The main problem of implementing filtering operation in VHDL is the lack of storage. Since the full frame cannot be stored on the device, indexing the neighboring pixels is a challenge. In order to overcome this challenge the desired rows of the image must be stored on a BRAM. For example, this module implements a 3x3 kernel so for each pixel 2 previous rows must be stored. This approach uses less memory but for the downside it is hard-coded which means that to implement a different kernel size the whole code must be changed.

The module aligns the neighbouring 8 pixels with the current pixel and applies the filtering operation to all of the 9 pixels at the same. This operation can be better observed in Figure 6. The red frame in Figure 6 represents the pixels that are used for filtering. The first batch of three signals refers to the

previous row, the middle batch refers to the current row and the last batch refers to the next row.

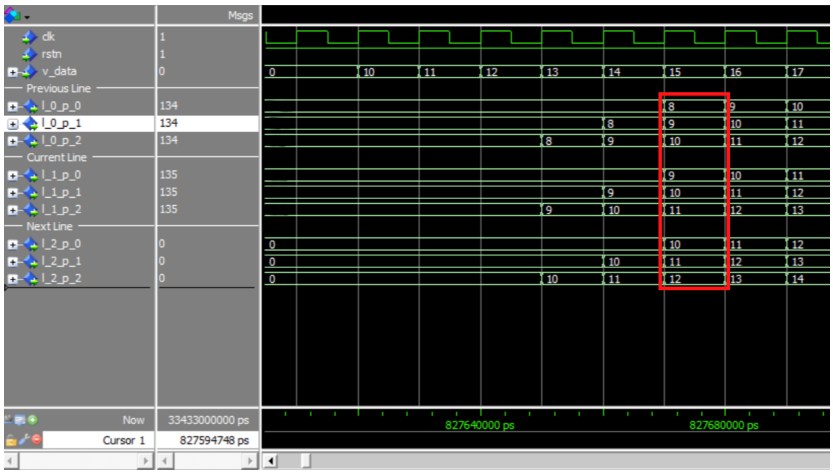


Figure 6: The simulation waveform that shows the alignment of the neighbouring pixels.

6.2 Histogram Calculation

In the context of an image, a histogram is a graphical representation that displays the distribution of the observed pixel intensity values, showing how many pixels in the image correspond to each intensity level.

In VHDL histogram calculation can be implemented using BRAM that has a depth of 8-bits because an image can only have values ranging from 0 to 255. The only problem of using a BRAM is that the fact that a data can only be read in the following cycle. Reading, incrementing and writing the data back to the original address takes 3 cycles long. Thus, if there are 2 or 3 pixels that have the same value and they are successive a problem occurs. In order to overcome this problem this module checks the following 2 pixels and increments the count accordingly. This operation can be observed in Figure 7. The state signal represents the state of the machine and as can be inferred from the figure that if 2 or 3 pixels that have the same value comes successively the machine enters the skip mode and updates the count accordingly.

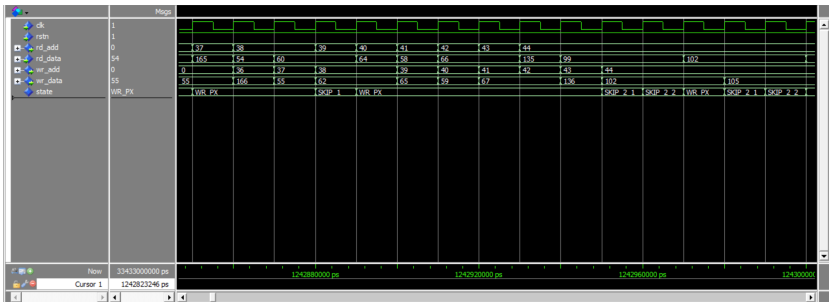


Figure 7: Histogram calculation process in the simulation.

The data inside the BRAM can also be monitored in the simulation (Figure 8). When compared to ground truth values that has been generated by a python script it can be stated that the module works properly. These values represent how many times the corresponding pixel intensity value has occurred in the image. The pixel intensity values are used as address indexes.

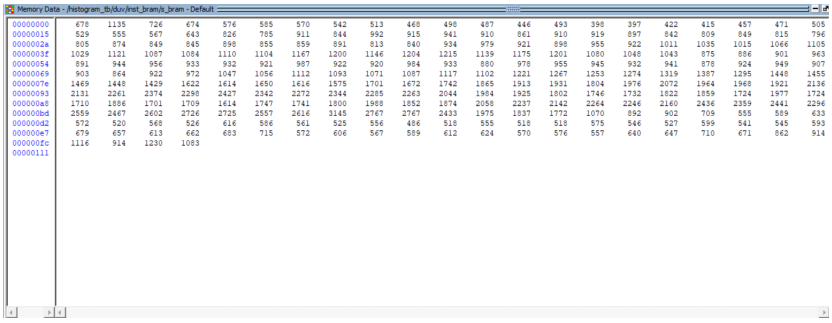


Figure 8: The data inside the BRAM that holds the histogram information.

6.3 Image Down Scaling Using Bi-linear Interpolation

The next image processing algorith to implement is interpolation. Bi-linear interpolation for image scaling calculates the pixel value in a resized image by taking a weighted average of the four nearest pixels in the original image, providing smoother transitions than nearest-neighbor interpolation. However, this module only scales the image on the horizontal axis (Figure 9) because adding the vertical axis would develop the need of an BRAM and that would complicate the algorithm a lot, not to mention my internship slowly started to come to an end. Nevertheless, the indexing algorithm and the arithmetic operations are still the same for both axes and memory management is pretty similar to the other modules that are presented in this report.

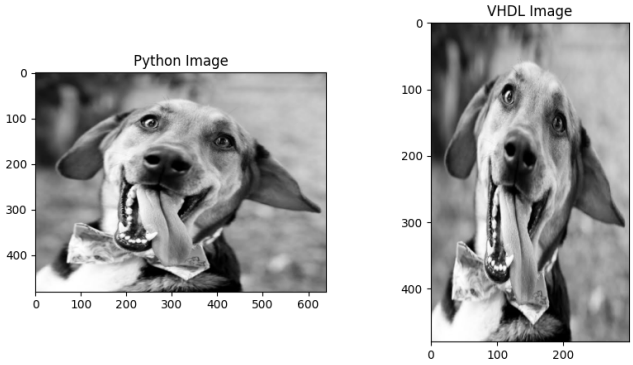


Figure 9: The original image and the resized image side by side.

7 Conclusion

VHDL is a powerful to implement image processing algorithms on a FPGA boards. Compared to high-level languages a VHDL program is much faster and more efficient, thanks to its parallel computing capabilities. Moreover, dealing with data on a low-level develops a better understanding of how computers work and how they interact with external components. Better understanding of computers leads to solutions that are more creative and effective.

However, writing a VHDL program is not a developer friendly experience at all. Functions that are only one line of code in high-level programs are combinations of multiple components that are hundreds of lines of codes. Additionally, VHDL programs are generally hard-coded which means that to change a parameter of an algorithm or a feature of it the whole program must be change, unlike a high-level program where only a variable is enough. For example, to change the kernel size of the filtering operation that is covered in this report the whole component must be rewritten or if the BRAM delays the read data signal for one more clock the histogram calculation module is must be rewritten.

It is also important to mention that the algorithms that are discussed in this report are basic algorithms that are developed for simulation setting, implementing them for a real-world use increases the complexity of the algorithms.

In conclusion, while VHDL offers significant advantages in efficiency, speed, and low-level hardware understanding, its complexity and lack of flexibility make it challenging and less user-friendly for dynamic or real-world applications.

Acknowledgments

I would like to thank Artron Inc. for giving me an internship opportunity and their welcoming manners towards me throughout my internship. Moreover, special thanks to Yunus Emre Doğan for devoting his time and attention during my internship, his experience and knowledge on VHDL programming was a huge part of my learning process.