Image Processing and Computer Vision:

A Comparison between CPU and FPGA

By

Hassan Mohammed Warrag Abdelrahman

INDEX NO.124042

Supervisor

Dr. Abdelrahim Elobeid Ahmed

A REPORT SUBMITTED TO

UNIVERSITY OF KHARTOUM

In partial fulfillment for the degree of

B.Sc. (HONS) Electrical and Electronic Engineering

(ELECTRONIC SYSTEMS SOFTWARE ENGINEERING)

Faculty of Engineering

Department of Electrical and Electronic Engineering

October 2017

DECLARATION OF ORIGINALITY

I declare this report entitled “**Image Processing and Computer Vision: A Comparison between CPU and FPGA**” is my own work except as cited in references. The report has not been accepted for any degree and it is not being submitted currently in candidature for any degree or other reward.

Signature:______________________________

Name:_________________________________

Date:__________________________________

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my supervisor Dr. Abdelrahim Elobeid Ahmed, for the guidance and support he provided throughout the project and for his time, patience, and for providing me with the opportunity to work with him. It has been a great privilege and honor.

I would like to extend my thanks to the Eng. Montasir Abdeljlel Salih and Eng. Imad Abd-Alhafeez and more generally to Telecommunication research center for providing us with computing resources and support required to accomplish this project.

DEDICATION

This thesis is dedicated to my mother Amal, my father Mohammed, my brothers Ahmed, Abdelrahman, Abubaker, Omer and my lovely sister Islam. I will remain unable to pay you back no matter what I did.

ABSTRACT

Computer vision and image processing are a fundamental issue in computer science. Computer vision is a growing field with many applications in real world. Some of those applications require real time processing. This can be achieved by either improving the core algorithm or by improving the hardware that the algorithm will run on. Regarding the hardware, it is hard to choose between the flexibility provided by general purpose architectures (such as CPUs) and the efficiency provided by purpose built architectures (such as FPGAs).

This thesis aims to find a suitable tool for designing image processing and computer vision algorithms in an FPGA and to compare between CPU and FPGA in terms of computational time, resources needed and accuracy(Peak signal to noise ratio).

Objectives of this thesis were achieved and a detailed comparison between FPGA and CPU was made. During the research, SIFT algorithm was implemented using MATLAB Simulink and then it was implemented in an FPGA Xilinx Virtex 6.

Results using multiple image processing and computer vision algorithms were obtained and it was noticed that in most of the cases, FPGA gave an outstanding performance compared to CPU under specific configurations.

A discussion of the reasons behind those results was made and some of the improvements to increase FPGA's performance in image processing and computer vision applications and future work were suggested.

المستخلص

مجال رؤية الكمبيوتر ومجال معالجة الصور هي مجالات أساسية في علوم الكمبيوتر.مجال رؤية الكمبيوتر هو مجال متزايد مع العديد من التطبيقات في العالم الحقيقي. بعض هذه التطبيقات تتطلب معالجة في الوقت الحقيقي. ويمكن تحقيق ذلك من خلال تحسين الخوارزمية الأساسية أو من خلال تحسين الأجهزة التي سيتم تشغيل الخوارزمية عليها. وفيما يتعلق بالأجهزة، فإنه من الصعب الاختيار بين المرونة التي توفرها معماريات الأغراض العامة (مثل وحدات المعالجة المركزية) والكفاءة التي توفرها الأبنية المبنية لأغراض محددة (مثل مصفوفة البوابات المنطقية القابلة للبرمجة).

تهدف هذه الأطروحة إلى إيجاد أداة مناسبة لتصميم خوارزميات معالجة الصور والخوارزميات الحاسوبية في مصفوفة البوابات المنطقية القابلة للبرمجة ومقارنتها بين وحدة المعالجة المركزية و مصفوفة البوابات المنطقية القابلة للبرمجة من حيث الوقت الحسابي والموارد المطلوبة والدقة (نسبة الإشارة إلى الضوضاء).

وقد تحققت أهداف هذه الرسالة وأجريت مقارنة مفصلة بين مصفوفة البوابات المنطقية القابلة للبرمجة ووحدة المعالجة المركزية. خلال البحث، تم تنفيذ خوارزمية (تحويل الميزات الغير متأثر بالحجم – سيفت) باستخدام ماتلاب سيمولينك ومن ثم تم تنفيذه في مصفوفة البوابات المنطقية القابلة للبرمجة من نوع Xilinx Virtex 6.

تم الحصول على نتائج باستخدام العديد من معالجة الصور وخوارزميات رؤية الكمبيوتر وكان من الملاحظ أنه في معظم الحالات، أعطت مصفوفة البوابات المنطقية القابلة للبرمجة أداء متميزا مقارنة بوحدة المعالجة المركزية وفق تكوينات محددة.

وأجري نقاش للأسباب الكامنة وراء هذه النتائج، واقترحت بعض التحسينات لزيادة أداء مصفوفة البوابات المنطقية القابلة للبرمجة في معالجة الصور وتطبيقات رؤية الكمبيوتر والعمل في المستقبل .

Table of Contents

Table of Figures

Table of tables

List of Abbreviations

FPGA	Field Programmable Gate Array
CPU	Central Processing Unit
SIFT	Scale Invariant Feature Transform
OCR	Optical Character Recognition
PLA	Programmable Logic Array
PLD	Programmable Logic Device
CPLA	Complex Programmable Logic Array
LUT	Lookup Table
ROM	Read Only Memory
RAM	Random Access Memory
SoC	System on Chip
MUX	Multiplexer
DSP	Digital Signal Processor/Processing
ASIC	Application-Specific Integrated Circuit
HDL	Hardware Description Language
VHDL	Very High Scale Integrated Circuit Hardware Description Language
FIL	FPGA in the Loop
DoG	Difference of Gaussian

Chapter 1 : Introduction

1.1 Overview

Computer vision is extracting information from images using artificial systems. It is being used in many applications like 3D construction, medical imaging, object detection and pattern recognition. In some applications it is critical to process the data in real time. Due to the limitations in memory and computation time, many researches were developed and being developed every day to overcome those limitations. Some of those researches focus on improving the software part (core algorithms) and other researches focus on improving the hardware that the algorithm will run on.

FPGA is a programmable integrated circuit with a parallel architecture that was designed as a stand-alone system to improve performance and reduce power consumption. Another advantage of the FPGA is its ability to be upgraded easily unlike ASIC systems.

1.2 Problem Statement

When designing a system that includes computer vision processes at any of its stages, it is hard to choose between the flexibility of General purpose hardware architectures (such as a CPU) or the efficiency of purpose built architectures (such as an FPGA).

1.3 Project Objectives

This project aims to:

1. To compare between languages used to program FPGAs and find a suitable solution for image processing and computer vision applications.
2. To compare between CPU and FPGA in the following areas:
    - Processing time.
    - Hardware resources needed.
    - Accuracy (peak signal to noise ratio).

1.4 Thesis Layout

The rest of this thesis is organized as follows:

- **Chapter 2 (Literature Review):** a brief description of image processing, computer vision and FPGA history and their applications
- **Chapter 3 (Methodology and Implementation):** provides details about the project necessary to replicate the implementation of (2D filter, Median filter, Edge detection, Color correction, Gamma correction and SIFT) in Simulink and FPGA using FPGA-in-the-loop approach and a brief description of how to acquire results.
- **Chapter 4 (Results Discussion and Analysis):** provides research results and analysis and discussion of the acquired results with respect to performance in time, hardware resources needed and peak signal to noise ratio).
- **Chapter 5 (Conclusion and Future Work):** Contains comments about the project with focus on how much of the objectives were achieved, in addition to possible future work.

Chapter 2 : Literature Review

2.1 Images

2.1.1 Definitions and terminologies

An image is an approximation of real world scenes. It can be two dimensional such as a photograph or a screen play, or three dimensional such as a hologram.

An image is constructed from smaller units called Pixels. A digital image is usually represented in matrix form of pixels each containing a value.[1]

2.1.2 Types of Images

There are multiple types of images (shown in **Figure 2.1**) such as:[1]

- **Binary Image:** in this type the pixel value is either 0 or 1 (black and white respectively).Thus one bit is required to represent each pixel.
- **Gray image:** in this type pixel value is usually in the range from 0 to 255 each representing a shade of gray. Thus each pixel requires 8 bits keeping in mind that the range of grey levels is not constant and it can be from 0 to 1 in some applications.
- **Colored image (RGB):** here each pixel has a particular color represented by three components each describes the amount of red, green and blue values in that color. Thus each component values are in the range of 0 to 255 so 8 bits are required for each one. So each pixel requires 24 bits.

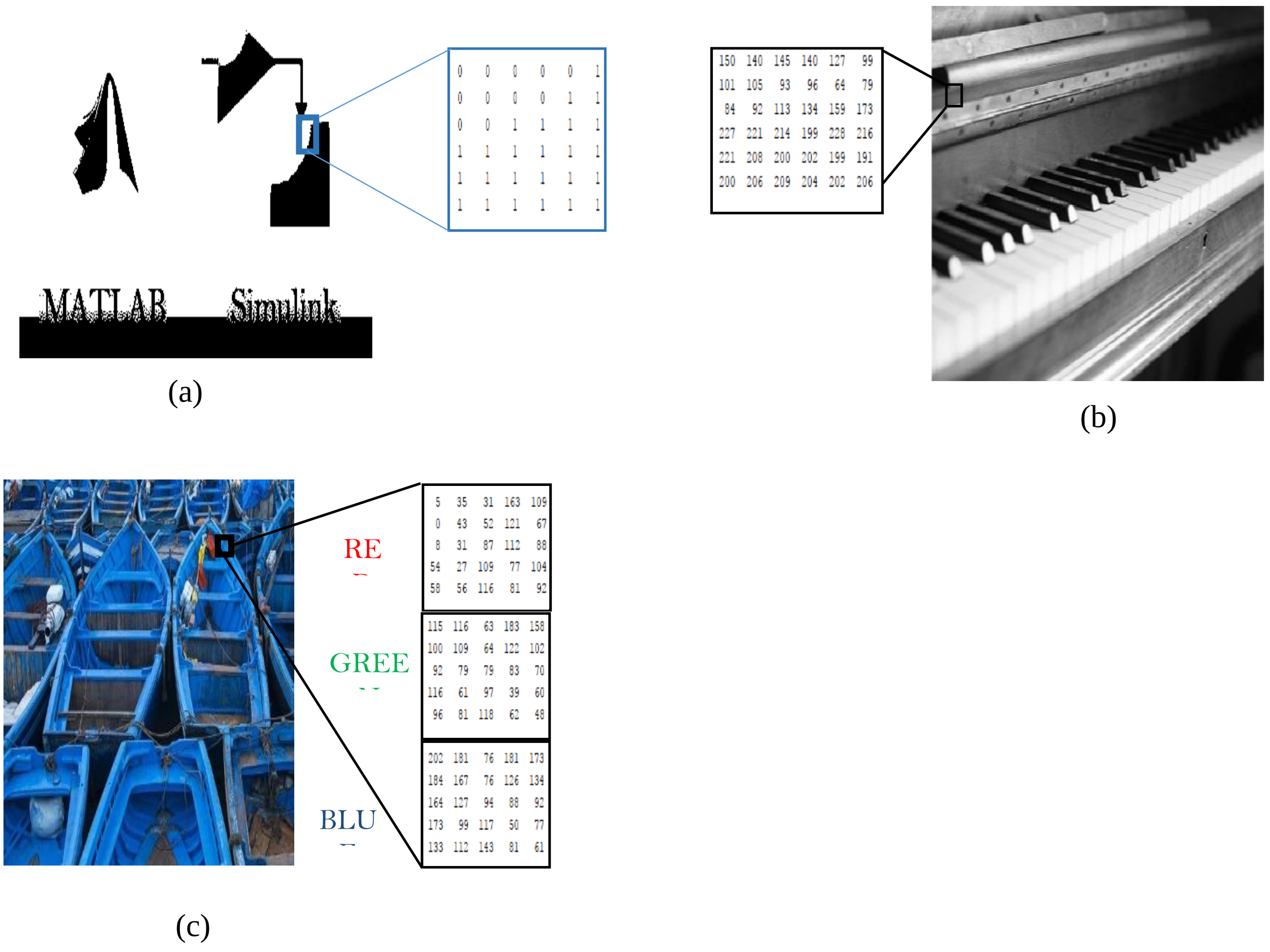


Figure 2.1: *(a) Binary image. (b) Grey image. (c) Colored image.*

2.2 Image Processing

2.2.1 Definitions and terminologies

Image processing is the set of techniques used to convert an image (or a stream of images or a video) into digital form and perform some signal processing and mathematical operations on it to get an enhanced version of the image or extract some useful characteristics.[2]

2.2.2 Historical background

Many techniques of digital image processing were developed in early 1960's and it was called digital picture processing then. The computational power was relatively high keeping in mind the computational power capabilities available at that time. By the 1970's, cheaper computers were developed and dedicated hardware became available which allowed images to be processed in real time for some applications. With the fast improvement in the next decades, digital image processing has become the most common form of image processing. However digital image processing for medical applications started a little bit earlier, in the 1990's.

In 2002 gradient domain image processing was introduced as a new way to process images in which the difference between pixels are manipulated rather than the value of the pixel itself.

2.2.3 Applications of image processing

Image processing is generally used to:

- **Visualization** to get hidden info.
- **Image sharpening and restoration** to get an enhanced version of the image.
- **Image recognition** to distinguish objects in that image.[3]

2.3 Computer Vision

2.3.1 Definitions and terminologies

Computer vision is extracting information from images using artificial systems. It's being used nowadays in many real world applications such as: optical character recognition (OCR), 3D construction, medical imaging, object detection and pattern recognition. Computer vision algorithms assumes that some image processing techniques were used as a preprocessing step to enhance the image and/or to remove noise and unwanted details.[3]

2.3.2 Historical background

Late in 1960's computer vision began as a step to mimic human visual system to improve robots development. The main difference between computer vision and digital image processing at that time was that computer vision aim was to extract three-dimensional structure from images to achieve full scene understanding.

Many of the algorithms exist today were developed based on studies in 1970's.From that time until 1990's most of the studies were based on more rigorous mathematical analysis and quantitative aspects of computer vision. By the 1990's statistical techniques were used in practice to recognize faces in images and by the end of this decade a significant change came about with the increased interaction between the fields of computer vision and computer graphics. This included image based rendering and other applications.

2.3.3 Computer vision challenges

The main challenges in computer vision field are the **finite memory** and the **computational time**. A good algorithm is the one that is reasonably accurate, fast, and within the memory limits.

Multiple approaches were developed to overcome those problems by enhancing either the algorithm itself or by enhancing the hardware that the algorithm will run on.[3]

2.4 The FPGA

2.4.1 Introduction

In order to understand the architecture of Filed Programmable Gates Array (FPGA) and why it’s different from a typical General purpose architectures, first we have to get a better insight of what is hardware abstraction.

Generally, there are four main hardware abstraction levels, which are:

- **System level:** for describing high level main control and data path.
- **Register transport level:** for describing function units.
- **Gate level.**
- **Transistor level.**

In designing hardware systems, there are two main topologies based on the input type, which are:

- **Software inputs topology:** such as microprocessors and microcontrollers.
- **Hardware inputs topology:** such as programing logic devices technologies (FPGA, PLA, PLD, and CPLA).

Most of the latter type devices mentioned above are Field programmable devices.[2]

2.4.2 Field programmable devices

Field Programmable Devices types:

- **Read Only Memories.**
- **Programmable Logic Arrays.**

- **Complex Programmable Logic Devices.**
- **Field Programmable Gate Arrays (FPGA).**

Some of the new hardware trends are: configurable processors, standard bus structure, software programming and software utilities. But the discussion of these trends is out of our scope.

2.4.3 FPGA architecture

The FPGA is one of the field programmable devices. It has been designed using the hardware inputs design topology. Its structure components[2] (shown in **Figure 2.2**) are:

- **Lookup table (LUT)**: it's a small memory block with ***n*** inputs and one output and it stores up to $\boldsymbol{2^n}$ entries. Its output can be programmed to implement any function of inputs.
- **Storage Elements** such registers.
- **Memories** such as ROM and RAM.
- **Programmable Switches** to control the interconnection between logic blocks.

Homogeneous and Heterogeneous FPGAs

In summary, fine-grained homogeneous FPGAs consist of a sea of identical logic blocks. However a heterogeneous mixture of both fine-grained logic blocks and larger coarse-grained blocks targeted for accelerating commonly used operations is used in the current FPGAs instead of the fine-grained homogeneous structure.

If we were to compare the two structures we will find that heterogeneous systems are faster because the coarse-grained blocks does not have the same overhead that comes with fine-grained flexibility. However heterogeneous devices are harder to program and much of the functionality may be underused if the mix of the specialized blocks is not right.

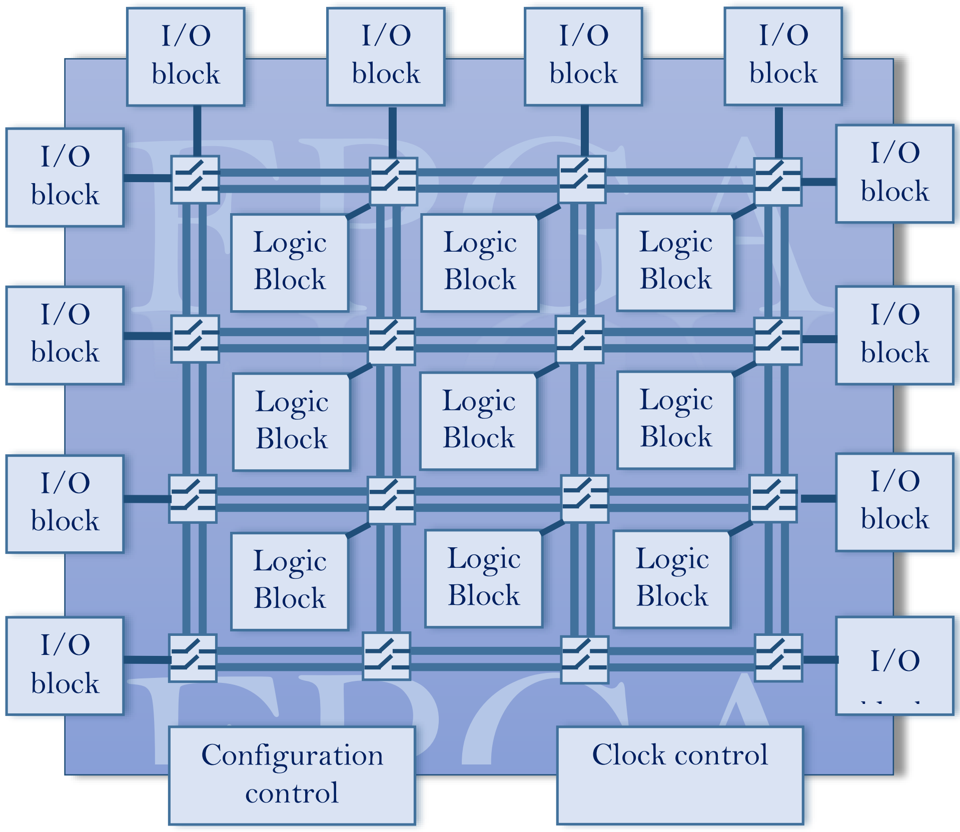


Figure 2.2: *FPGA architecture.*

The flexibility in enabling different logic blocks (shown in **Figure 2.3**) to be connected to implement the desired functionality is achieved by the **programmable interconnects** (The **programmable switches**). The connection of signals between the FPGA and external devices is achieved by the **input and output blocks**.

Since FPGAs are usually designed as synchronous devices, they require a clock signal. Although multiple clocks may be used in the design, it is important to note that registers, synchronous memories and I/O blocks (shown in **Figure 2.4**) can only be controlled by one clock. The clock domain consists of all the circuitry that is controlled by a particular clock.

The configuration data is contained within the FPGA's static memory (SRAM) cells. However, flash-programmed devices are sometimes used by some manufacturers. The configuration bits control all of the switches, structures and options within the FPGA.

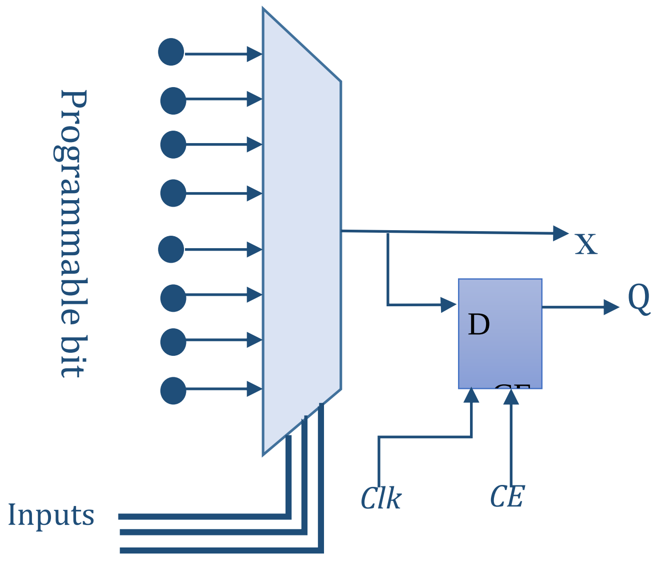


Figure 2.3: *Logic Block architecture.*

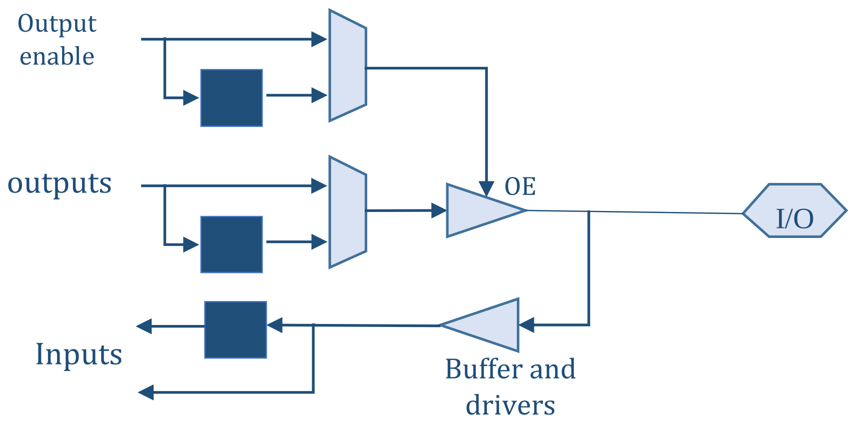


Figure 2.4: *I/O Block architecture.*

Since power consumption is one of the main reasons to use an FPGA, we can work to reduce it by considering each term of the equation:

$$P = NCV_{dd}^2 f$$

Where P is the power consumption, NC is the load capacity, V_{dd} is the power supply voltage, f is the frequency.

As follows:

- By limiting the number of outputs that change with each clock cycle.

- By minimizing the fan-out from each gate and minimizing the use of long wires (to keep the capacitance lower).
- By reducing power supply voltage.
- By reducing the clock frequency.

2.4.4 FPGA manufacturers

Two of the main FPGA manufacturers in terms of market share are **Xilinx** and **Altera**:

Xilinx was one of the first developers of field programmable gate array technology. It has multiple families but the two main families are the **Spartan** and **Virtex** series. Spartan devices are designed mainly for low cost meanwhile Virtex series are designed mainly for high performance. In the newer generations, the 7 series devices, the Spartan family is replaced by **Artix** and **Kintex** families. And within each generation, a range of device sizes is available.[2]

Altera provides three families of FPGAs: the low cost **Cyclone** series, the mid-range **Arria** series and high performance **Stratix** series. With the **Stratix II** Altera introduced a new adaptive logic module (Hutton et al. 2004) meanwhile the original Stratix used the same logic architecture as the Cyclone.

None of the current families incorporate a processor core within the logic. In terms of embedded processors, Altera has focused its efforts on its soft-core NIOS processor.

Xilinx all programmable System on chip (SoCs) portfolio integrates the hardware programmability of an FPGA with the software programmability of a processor, providing new levels of system performance, flexibility and scalability. The portfolio gives the design overall system benefits of power reduction and lower cost with fast time to market.

The flexible programmable logic provides you with optimization and differentiation, allowing you to add the peripherals and accelerators you need to adapt to a broad base of applications.

2.4.5 Embedded FPGA

Embedded FPGAs are used for the following reasons:

- **Flexibility**: support changing requirements, multiple chip variants and multiple un-ratified standards.

- **Lower R&D costs:** reduces total research and development cost and time to bring multiple SoCs to market.
- **Increase Performance:** Integration of FPGA into SoC increase performance by eliminating chip-to-chip delays.
- **Lower power consumption:** The integration of hardware accelerators using programmable logic results in reducing the capacity load of the main CPU.
- **Fine-grained architecture:** using Fine-grained architecture gives us the option to optimize hardware resources in lookup tables (we can either use independent LUT3s or one LUT4) or in multiplexers (using 4 of the 2:1 MUX or one 8:1 MUX) or in storage elements (optional Flip Flops).
- **Multiple inputs and outputs:** enable independent logic to share the same logic cell: this leads to much higher logic cell utilization in real life designs.

2.4.6 FPGA challenges

Earlier, when FPGAs were firstly used, they had limitations in memory and the number of gates needed for large data processing such as digital signal processing (DSP). But nowadays FPGAs contain DSP, ALU and also RAMs and ROMs as memories.[4][2]

So the current challenges of FPGAs are:

- **Memory:** for large applications we might need to use external memory.
- **Interfacing:** we need to design a communication protocol for any communication. This limits the resources that can be used by the main system. So the hardware design requirement and choosing a specific FPGA is of great importance and quite a challenge. Another approach is use an external communication IC but this will add more cost for the hardware.
- **Complexity:** because of the limitation in memory and the need to use an external IC for communications at the interface side and the main system, the complexity of the overall system increases. It is needed to perform simulation and hardware verification of the main functions as a first stage and then we can design the whole system and verify it.

2.4.7 FPGA applications

Some of FPGA applications include:[4]

- **Alternative of ASICs and custom silicon:** FPGAs are increasingly being used to implement a variety of systems that were only realized using ASIC and custom silicon in the past.
- **Digital signal processing:** high speed DSP was traditionally implemented using specifically tailored microprocessors called **digital signal processors.** But FPGAs now contain embedded multipliers and adders and a reasonable amount of RAMs which can be used to perform DSP.
- **Embedded microcontroller:** small control functions were traditionally handled by special purpose embedded processor called microcontroller. But nowadays FPGAs have more than enough capabilities to implement those control functions combined with the selection of custom I/O function.
- **Physical layer communications:** FPGAs have been used to implement glue logic that works as an interface between physical layer communication chips and high level networking protocol layers.
- **Neural network:** the capacity and performance of the current FPGAs made them a much more realistic alternative to implement neural network applications.

2.5 FPGA Programming Tools and Languages

Over the last 20 years, researches moved to synthesis of hardware from high level languages (C and matlab for example), the reasons behind that are[2]:

- It enable existing software to be easily ported or recompiled to a hardware implementation.
- Many of the algorithms are tested first using a high level language (like C and Matlab for example) before designing them, so it’s easier to allow those high level languages to convert them.
- Making hardware design looks like software design simplifies troubleshooting.

2.5.1 Types of languages:

Generally the languages used to program the FPGA can be divided to[2]:

1. **Hardware Description languages (HDL):** allow the complete development and modelling of systems containing FPGAs and other programmable hardware (e.g. VHDL and Verilog).

2. **Software based languages:** have a single source that can be compiled to both hardware and software. This results in speeding up the development process considerably. Generally software based languages can be categorized as follows:
   - **Structural approach:** high level synthesis based on structure hardware language (e.g. JHDL and Quartz).
   - **Augmented Languages:** mostly based on C language and can be used for modelling complete system (e.g Handel-C,Mitrion-C and System-C). It represents the hardware datatypes. the algorithm can be designed at many levels such as:
     - Architecture level: as RAM, ROM, Register…etc.
     - Arithmetic level: as addition, summation…etc.
     - Gate level: as AND, OR, MOSFET...etc.
   - **Native compilation techniques:** used to transfer from native languages to FPGA (e.g. MATLAB, Catapault C, ImpulseC and Transmogrifier C).
   - **Visual languages:** a graphical dataflow-based programming environment either from either :
     - Behavioral (such as Visual VHDL).
     - Dataflow (such as Khoros and Simulink).
     - Hybrid (such as **VERTIPH** –the **V**isual **E**nvironment for **R**eal **T**ime **I**mage **P**rocessing in **H**ardware).

2.5.2 MathWork [5]

MATLAB and Simulink for model-based design provide signal, image and video processing engineers with a development platform that spans design, modeling, simulation, code generation, and implementation.

When using MATLAB and Simulink to target FPGAs or ASICs (Application Specific Integrated Circuits), we can first design and run a simulation of the system with MATLAB, Simulink and stateflow and then generate bit-true, cycle-accurate, synthesizable Verilog and VHDL code using **HDL coder** which is provided by MATLAB.

2.5.2.1 HDL coder

HDL coder is a MATLAB toolbox product. It generates portable, synthesizable Verilog and VHDL codes from MATLAB codes and Simulink Models. Then the generated code can be used for FPGA programming or ASIC prototyping and design. HDL coder also provides an advisor to configure automation for programming Xilinx and Altera FPGAs and it supports optimizing the HDL code and Verifying the product by generating testbench for simulation and using FPGA-in-the-loop. Functions and workflow of HDL coder are shown in **Figure 2.5 .**

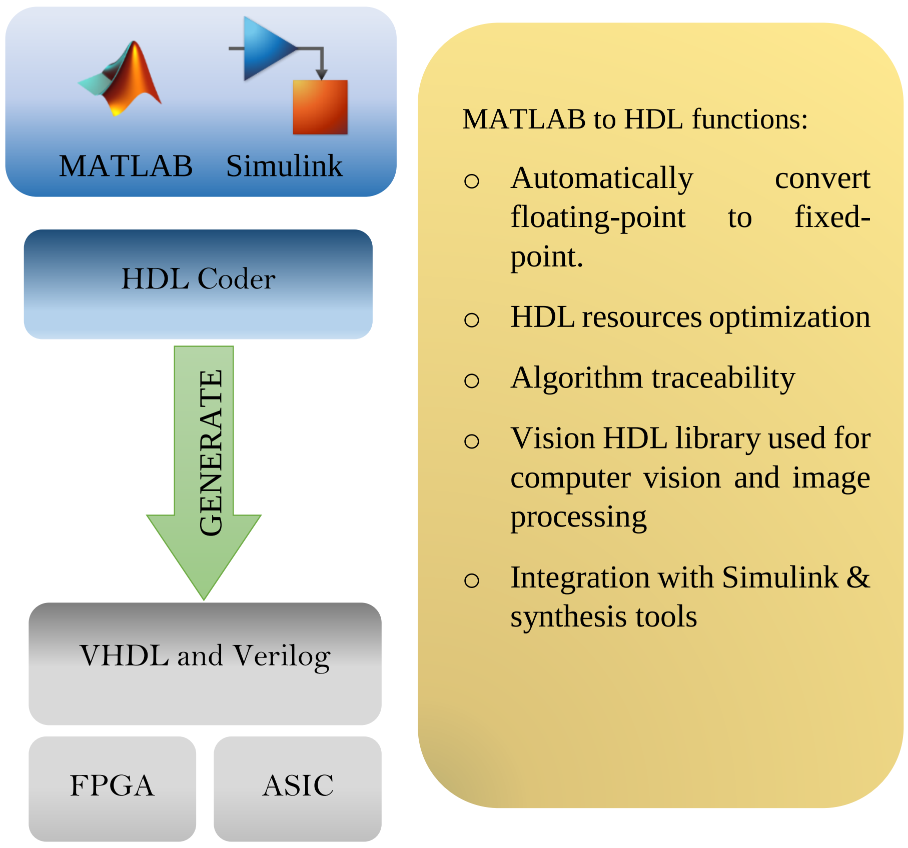

MATLAB
Simulink
HDL Coder
GENERATE
VHDL and Verilog
FPGA
ASIC
MATLAB to HDL functions:
Automatically convert floating-point to fixed-point.
HDL resources optimization
Algorithm traceability
Vision HDL library used for computer vision and image processing
Integration with Simulink & synthesis tools

Figure 2.5: *MATLAB HDL coder functions and workflow.*

2.5.2.2 FPGA-in-the-loop:

FPGA in the loop (FIL) simulation allows us to use Simulink or MATLAB software to test the design in real hardware for any existing HDL code. **Figure 2.6** shows the workflow of the FIL.

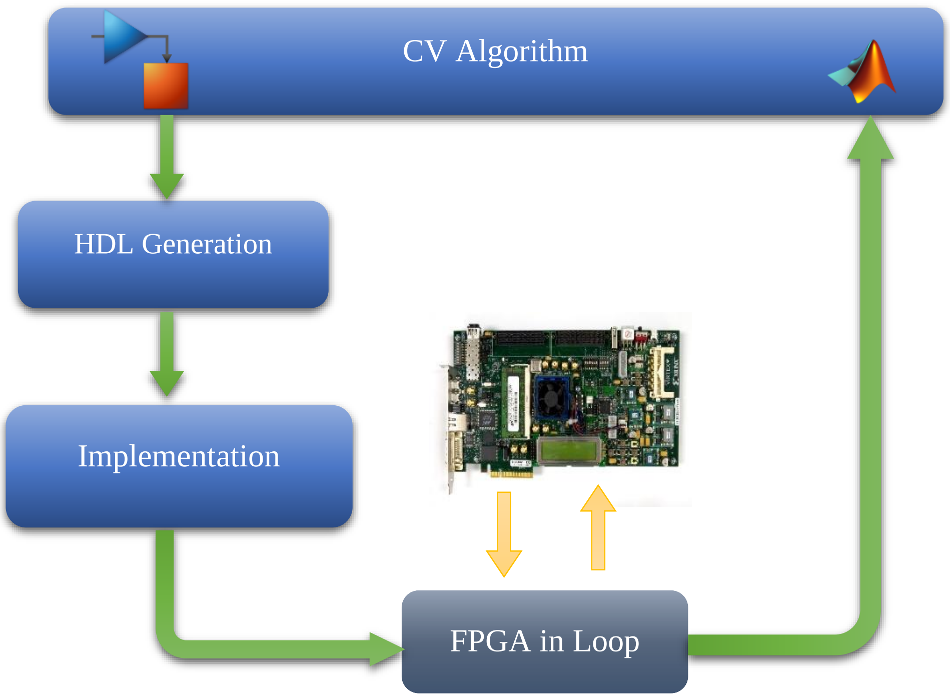


Figure 2.6: *FPGA in loop workflow.*

2.5.2.3 Vision HDL Toolbox

Vision HDL toolbox is a MATLAB toolbox product used for modelling and implementation of image processing for hardware target. It is built to provide capability for concurrency to use image processing algorithms that have high inherent parallelism. It provides:

- Pixel-based functions and blocks.
- Conversion between frame and pixel.
- Standard and custom frame sizes and frame rates.
- Optimization of HDL code generation.

When vision HDL toolbox is used for prototyping video and image processing algorithms on FPGA/SoC it provides:

- Efficient and readable HDL code: when it's used with **HDL Coder**.
- FPGA-in-the-loop testing and acceleration: when it's used with **HDL verifier**.

2.5.2.4 Xilinx system generator

When we are targeting Xilinx FPGAs, we can use Xilinx library for bit-true and cycle-true blocks to build a model in Simulink. Xilinx System generator (which is a plugin to

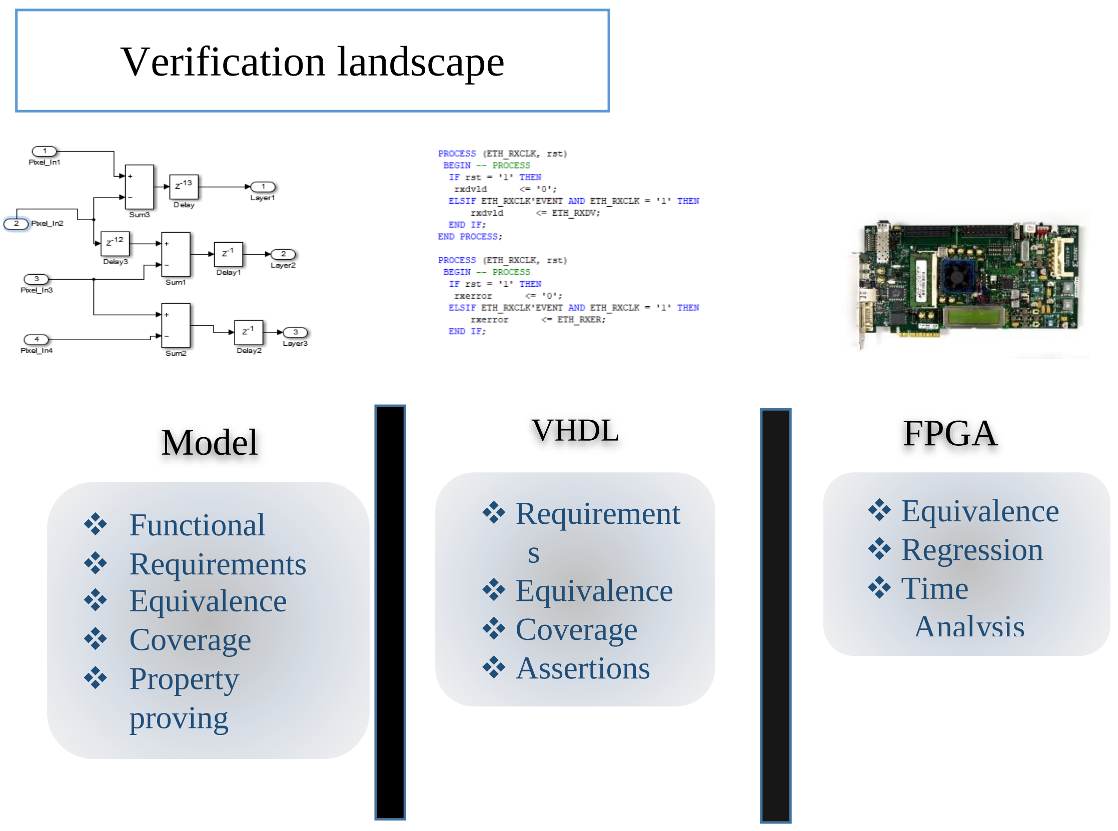
Simulink code generation software) is used to automatically generate synthesizable HDL code mapped to pre-optimized Xilinx algorithms.

2.5.2.5 Verification landscape:

Generally, there are three levels of verification:

- Simulink model simulation.
- Xilinx ISE (checks if the RTL code is synthesizable and performs clock analysis and generates an output file-.bit file- which is compatible with the target FPGA).
- Implementing project in hardware in loop for analysis to compare the real results with requirements.

The three levels are shown is **Figure 2.7** .



```
PROCESS (ETH_RXCLK, rst)
 BEGIN -- PROCESS
  IF rst = '1' THEN
   rxdvld      <= '0';
  ELSIF ETH_RXCLK'EVENT AND ETH_RXCLK = '1' THEN
      rxdvld      <= ETH_RXDV;
  END IF;
END PROCESS;

PROCESS (ETH_RXCLK, rst)
 BEGIN -- PROCESS
  IF rst = '1' THEN
   rxerror      <= '0';
  ELSIF ETH_RXCLK'EVENT AND ETH_RXCLK = '1' THEN
      rxerror      <= ETH_RXER;
  END IF;
```

Figure 2.7: *Verification landscape.*

2.6 FPGAs and Computer Vision

2.6.1 Live Video Edge Detection in Simulink[6]

Since there is no easy way to program an FPGA, Chad Eckles and Dr. Ramakrishna Mukkamala developed a simple live video edge detection using Simulink and demonstrated how edge detection works on live video . The purpose of their paper was to familiarize people on Simulink with an emphasis on explaining how to implement basic image and video processing techniques.

Then after acquiring this basic knowledge, people can use Xilinx blocks inside Simulink to convert this algorithm to an algorithm that works on Xilinx FPGAs.

2.6.2 FPGA Based Implementation of Image Edge Detection using Canny Edge Detection Algorithm[7]

In their paper, K.Naresh , M.Mahender presented a methodology for implementing real time DSP applications on field programmable gate arrays(FPGAs). They applied Edge detection on FPGA using Xilinx system generator. Sobel filter was used as an edge detector. First, the design was implemented targeting the low cost available Spartan3A DSP 3400 device, then a Vertix5 device. The Edge Detection method has been verified successfully with no visually perceptual errors in the resulted images.

2.6.3 A Comparison of Affine Region Detectors[8]

A paper with the above name was first published in 2006. The authors of this paper compare the performance on a set of test images under varying imaging conditions. Six types of detectors were included: detectors based on affine normalization around Harris and Hessian points, a detector of maximally stable external regions (MSER), and edge-based region detector, a detector based on intensity extrema, and a detector of salient regions. Performance was measured against changes in viewpoint, scale, illumination, defocus and image compression.

The comparison has shown that the performance of all detectors declines slowly with similar rates, as the change of viewpoint increases. There was not a single detector which outperformed the other detectors for all the scene types and all types of transformation. Though in many cases the highest score was obtained by the MSER detector, followed by Hessian-Affine. It was found that MSER performed well on images containing homogeneous regions with distinctive boundaries. Hessian-Affine and Harris-Affine provided more regions than the other detectors which is useful in matching scenes with occlusion and clutter.

It was found that the detectors were complementary, i.e. they extract regions with different properties and the overlap of these regions is small if not empty. So several detectors should be used simultaneously to obtain better performance. The output of different detectors can be combined by concatenating the respective matches. This increases the number of matches and therefore the robustness to occlusion, at the expense of processing time

2.6.4 Distinctive Image Features from Scale-Invariant Keypoints[9]

A method for extracting distinctive invariant features from images was presented by David G. Lowe in his paper. This method can be used to perform matching between different views of an object or scene. Features are invariant to image scale and rotation, and are shown to provide robust matching across substantial range of affine distortion, change in 3D viewpoint, addition of noise, and change in illumination. The paper also described an approach to use these features for object recognition by matching individual features to a database of features from known objects.

Scale Invariant Feature Transform (SIFT) keypoints described here are useful due to their distinctiveness which is achieved by assembling a high-dimensional vector representing the image gradients within local regions of the image.

SIFT algorithm and its implementation on MATLAB – Simulink will be discussed in details in Chapter 3.

Chapter 3 : Methodology and Implementation

3.1 Comparison between HDL and software-based languages

1. HDL languages are better at describing the structure of a hardware design.
2. HDL languages provide more flexible options for optimizing the design in terms of speed and resources needed. Because low level design has a strong focus on the actual hardware rather than the algorithm.
3. In HDL languages, the designer must control everything in quite fine detail. But including both the data and control flows in a complex algorithm is rather a difficult task.

In our project, we decided to use HDL coder instead of System generator for the following reasons:

- HDL coder supports converting MATLAB code to HDL.
- HDL coder supports converting Floating to Fixed-Point.
- HDL coder allows Simulink blocks to work with different targets.
- HDL coder provides automatic test generation.
- HDL coder allows verification of FIL using PC resources.
- HDL coder works with vision HDL library.

3.2 Algorithms:

In this part we will discuss the algorithms that were applied to the FPGA and in each one we will discuss briefly how it works, its Simulink model and the configurations for the blocks used in the model.

3.2.1 2-D filter

3.2.1.1 Introduction

General 2-D filters are used for various reasons depending on the value of the matrix used for the filter. For a causal discrete-time FIR filter of order N of 1 dimension, each value of the output is the weighted sum of the most recent input values. General equation for 2-D filters is:[1]

$$y[n] = b_0 x[n] + \ b_1 x[n-1] + b_2 x[n-2] \dots + b_N x[n-N]$$

Depending on the 2-D filter kernel, the filter could be either high pass filter of low pass filter. Equations for the two filters are given below:

$$HighPass = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

$$LowPass = \begin{bmatrix} 0.0625 & 0.125 & 0.0625 \\ 0.125 & 0.25 & 0.125 \\ 0.0625 & 0.125 & 0.0625 \end{bmatrix}$$

High pass filter is usually used to attenuate fine details to focus only on the edges or more generally the high transitions.

Low pass filter is usually used to smooth the image by taking a weighted average of the pixel and its neighborhood which helps in reducing noise and undesired edges and details.

Here we used the low pass filter given above and we also used a filter called Gaussian filter which is given below:[10]

$$Gaussian = \begin{bmatrix} 0.0113 & 0.0838 & 0.0113 \\ 0.0838 & 0.6193 & 0.0838 \\ 0.0113 & 0.0838 & 0.0113 \end{bmatrix}$$

3.2.1.2 Simulink implementation

First, you have to make sure that the dimensions of your image is the same as the configuration in the frame to pixels and pixels to frame blocks, not to miss any pixels from the original image. This configuration will be the same for the rest of the models.

The model is shown in **Figure 3.1** and the subsystem is shown in **Figure 3.2** .

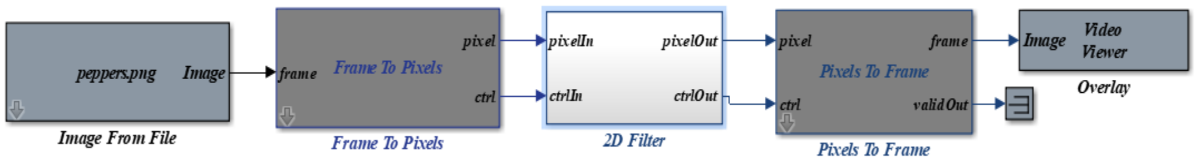


Figure 3.1: *2-D filter Simulink model.*

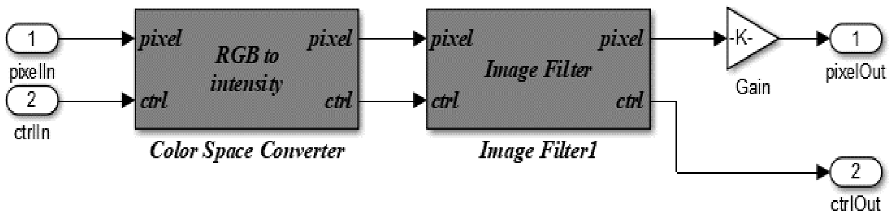


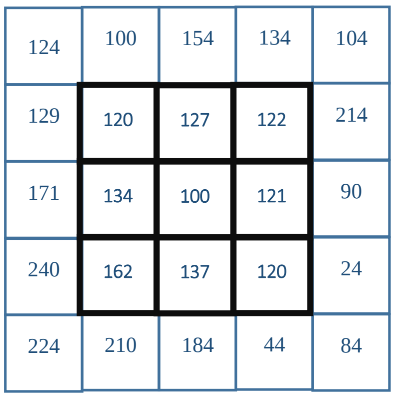
Figure 3.2: *2-D filter subsystem.*

For the 2D filter subsystem, choose the option of RGB to intensity in the block **Color Space Converter**. For the block **Image filter** you can enter your filter matrix. Here you can add the matrices mentioned before.

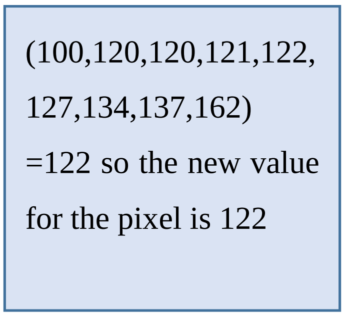
3.2.2 Median

3.2.2.1 Introduction

Generally, the median filter is used to remove noise (which is usually black or white dots) by taking into account a group of pixels around each pixel(neighborhood around the center pixel) and finding the median of those pixel then replacing the value of the center pixel by the new value of the median. An example is shown in **Figure 3.3**.[1]



124	100	154	134	104
129	120	127	122	214
171	134	100	121	90
240	162	137	120	24
224	210	184	44	84



(100,120,120,121,122, 127,134,137,162) =122 so the new value for the pixel is 122

Figure 3.3: *Example of median filter.*

After finding the median of the first group of pixel the window is moved to the next group and so on until we have changed all of the pixels in a given image.

Median filter gives the best results when the noise in the image is salt-pepper noise (impulse noise). But it has an undesirable effect of removing some of the edges.

3.2.2.2 Simulink implementation

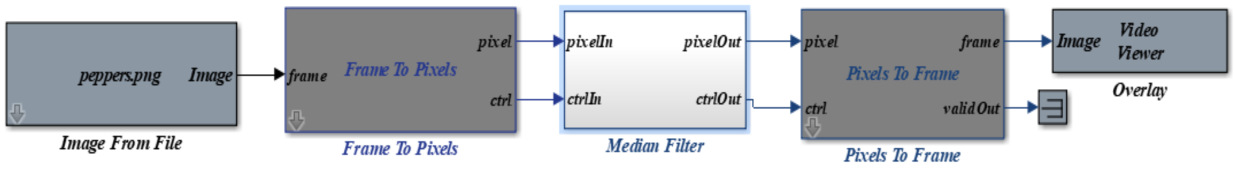


Figure 3.4: *Median filter Simulink model.*

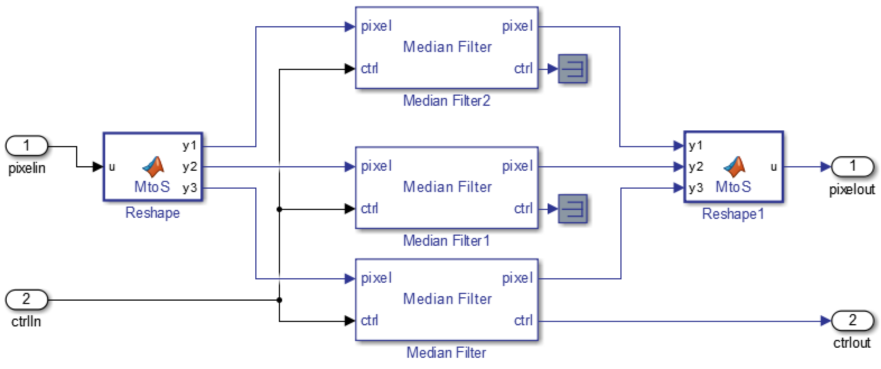


Figure 3.5: *Median filter subsystem.*

3.2.3 Edge detection

3.2.3.1 Introduction

Here we will discuss one of the filters used for Edge detection which is Sobel filter. This filter performs a 2-D spatial gradient measurement on a given image. Usually an edge is known by high transition between two surfaces which is represented by high frequency for that part of the image. For this reason Sobel filter emphasizes regions of high frequencies that often corresponds to edges. [10]

Sobel filter is performed by applying two kernels for the window. The size of the window we are working on is 3 × 3 and the two kernels are G_x for x-direction and G_y for y-direction . They are defined as below:

$$G_x = \begin{bmatrix} 0.125 & 0 & -0.125 \\ 0.125 & 0 & -0.125 \\ 0.125 & 0 & -0.125 \end{bmatrix}$$

$$G_y = \begin{bmatrix} 0.125 & 0.125 & 0.125 \\ 0 & 0 & 0 \\ -0.125 & -0.125 & -0.125 \end{bmatrix}$$

The final result is calculated as below:

$$result = \sqrt{{G_x}^2 + {G_y}^2}$$

A good approximation of this final result is the absolute sum of the two:

$$result = |G_x| + |G_y|$$

3.2.3.2 Simulink implementation

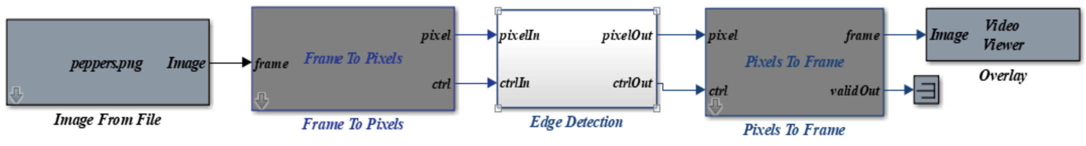


Figure 3.6: *Edge detection Simulink model.*

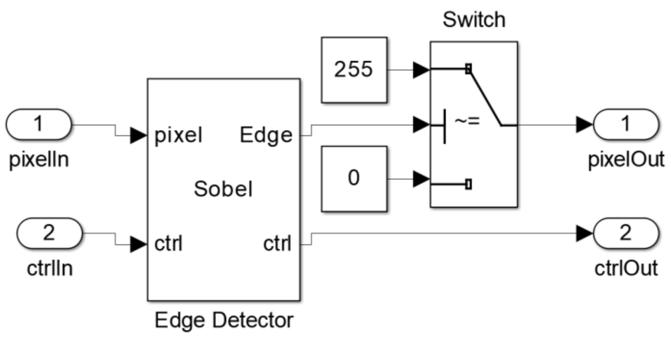


Figure 3.7: *Edge detection subsystem.*

3.2.4 Color correction

3.2.4.1 Introduction

In most of image processing applications, color correction is used to alter the overall color of light. In an RGB color space the image is saved in $m \times n \times 3$ array and each pixel can be represented as 3D vector$[R,G,B]^T$. The color correction matrix M is applied to input image in order to obtain the color corrected image as follows:[10]

$$\begin{bmatrix} R_{out} \\ G_{out} \\ B_{out} \end{bmatrix} = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} \times \begin{bmatrix} R_{in} \\ R_{in} \\ R_{in} \end{bmatrix}$$

Or in equations format as follows:

$$R_{out} = M_{11} \times R_{in} + M_{12} \times G_{in} + M_{13} \times B_{in}$$

$$G_{out} = M_{21} \times R_{in} + M_{22} \times G_{in} + M_{23} \times B_{in}$$

$$B_{out} = M_{31} \times R_{in} + M_{32} \times G_{in} + M_{33} \times B_{in}$$

In our model we choose the color correction matrix M as follows:

$$\begin{bmatrix} 0.79 & 0.11 & 0.1 \\ 0.05 & 0.82 & 0.13 \\ 0.15 & 0.08 & 0.77 \end{bmatrix}$$

Then by applying the above equations for each pixel we get the new values of red, green, and blue for the pixel.

3.2.4.2 Simulink implementation

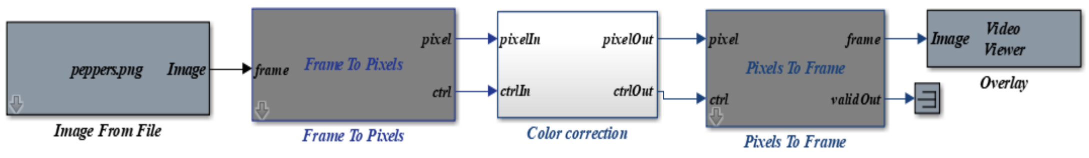


Figure 3.8: *Color correction Simulink model.*

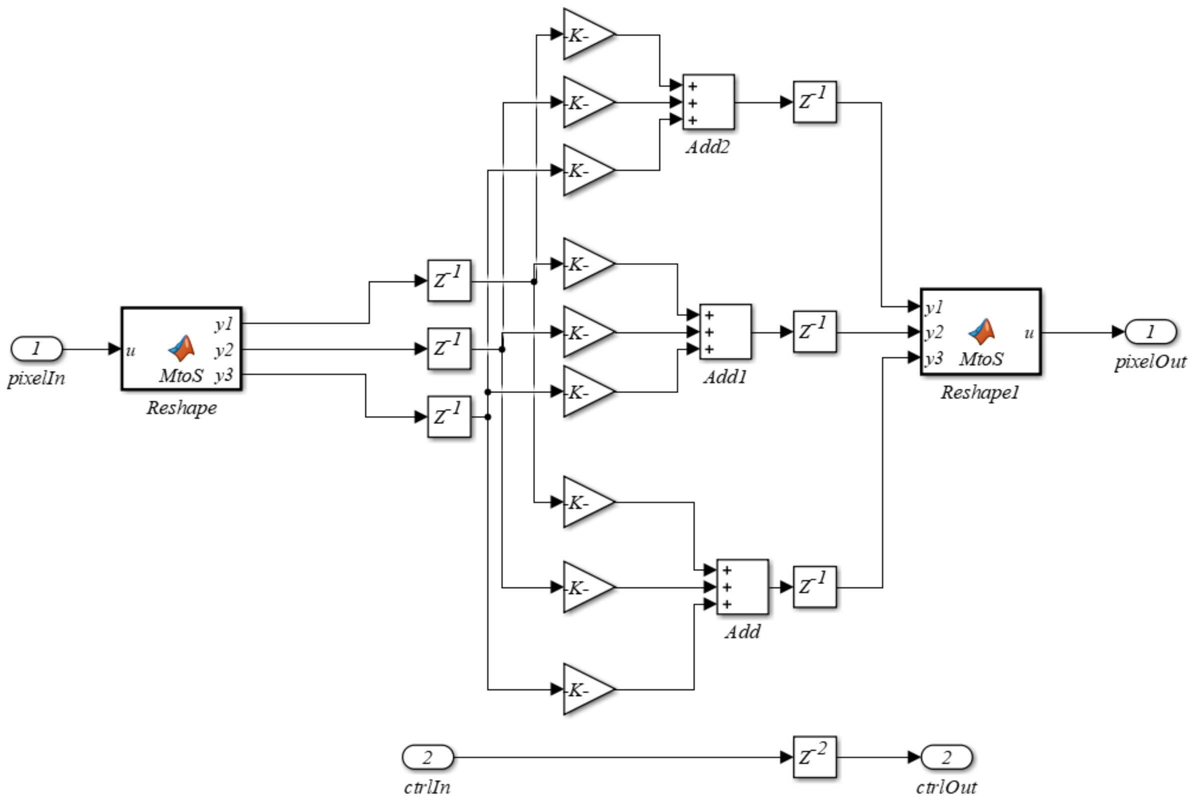


Figure 3.9: *Color correction subsystem.*

3.2.5 Gamma correction

3.2.5.1 Introduction

Gamma correction operation is used to adjust pixel luminance in a given image. Gamma correction controls the overall brightness of the image, when it’s not applied the image seems to be too dark.[1][10]

It’s a nonlinear operation and could be defined by the following power low expression:

$$V_{out} = K \times V_{in}^{r}$$

Where K is a constant (equals 1 in most cases), V_{out} and V_{in} are output and input pixels respectively. They are non-negative real values represents the output and input respectively. Often they are normalized to be in range of [0, 1]. For 8-bit grayscale images it's often in the range [0,255]. *r* is the gamma value, when this value *r*<1 it's called encoding gamma (which lighten the image) and when *r*>1 it's called decoding gamma(which darkens the image).

3.2.5.2 Simulink implementation

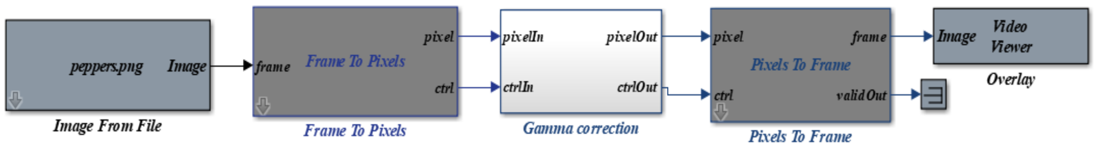


Figure 3.10: *Gamma correction Simulink model.*

Inside the **Lookup table** block, enter the following equation in the Table data field:

*unit8((([0:255]/255).^(2))*255)*

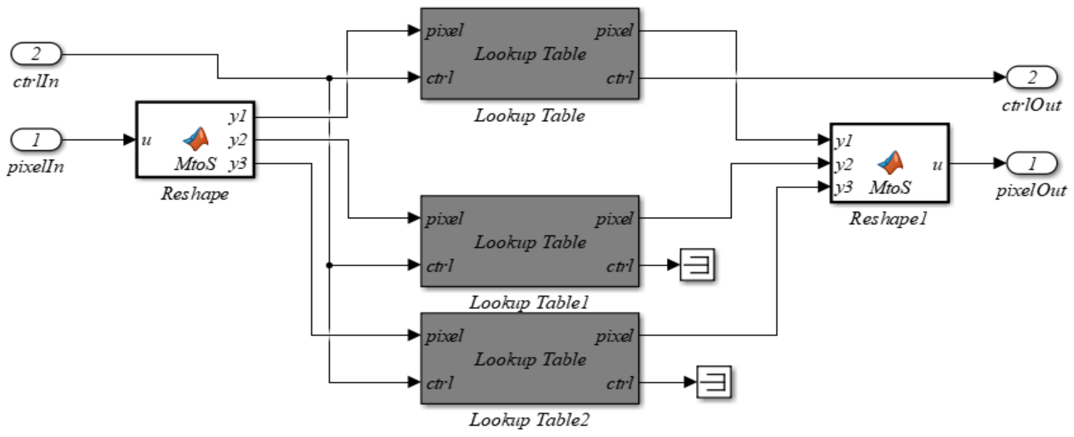


Figure 3.11: *Gamma correction subsystem.*

3.2.6 SIFT

3.2.6.1 Introduction

Scale Invariant Features Transform (SIFT) is an algorithm used to locate the keypoints features of an image and create a descriptor for each keypoint. These descriptors can be used to detect an object or a pattern in an image. Object recognition is used in many applications such as: Manufacturing Industry, face recognition for security purposes, vehicle tracking or any real time embedded system that requires object identification from an image or a stream of images. Object recognition is one of the main problems that Computer Vision tries to solve. Object recognition techniques try to identify an object in an image or a stream of images. The main purpose of these techniques is to find a way of describing the desired object, and use this description to identify the same object in another image with reliable accuracy and satisfactory performance. This represents a huge challenge, because the accuracy depends generally on the number of keypoints used to describe an object. And this will result in decreasing the performance due to the need of more computations. Keeping in mind that the same process above will be used to find the keypoints in the target image to identify its keypoints and match their descriptors.

The classical approach for object recognition was to explore all locations in the target image in order to locate the desired object. Many techniques were developed to solve object recognition problem by extracting features rather than the computationally-expensive classical approach. Some techniques used grouping of edges or line segments, these techniques worked well for some classes of object, but their results were not always reliable. Other techniques used Harris corner detector to find interest points and create a descriptor, these techniques gave good results in terms of speed and dealing with cluttered images, but if the scale has been changed they fail.

SIFT was introduced by David G. Lowe in 1999 as an object recognition algorithm to identify locations in an image that are invariant with respect to scaling, rotation, image translation and minimally affected by noise and distortions.

SIFT keypoints are the maxima or minima of scale space after applying Difference of Gaussian (DoG) and these keypoints are assigned to canonical orientation so that the descriptor is invariant to image orientation. The descriptor is created by including any pixel in the neighbour of the key location by a certain distance and measure their orientation relative to the key location orientation.[3]

SIFT can be implemented as follows:[11]

1. **Extrema Detection of a Scale-Space:**
   SIFT uses DoG as an approximation to Laplacian of Gaussian, because the Laplacian of Gaussian is computationally expensive compared to DoG. This step (shown in **Figure 3.15)** is used to make it possible to detect the scaled

keypoint using the same window. Then the extrema is obtained from the resulting image(shown in **Figure 3.16** and **Figure 3.17)**. Each pixel is compared to its 8 direct neighbouring pixels (in the same scale) and with the 18 pixels of the previous and the next scales. Thus, in total each pixel is compared with its 26 neighbouring pixels.

2. **Keypoint Localization :**

   DoG has high response for edges, so edges should be removed. For this, SIFT used Hessian matrix to compute the principal curvature and if the ratio is greater than a threshold the edge is removed (shown in **Figure 3.18** and **Figure 3.19**). This threshold is given as 10 in the paper.

$$\boldsymbol{H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}}$$

$$\boldsymbol{Tr(H) = D_{xx} + D_{yy}}$$

$$\boldsymbol{Det(H) = D_{xx}D_{yy} - (D_{xy})^2}$$

$$\boldsymbol{R = \frac{Tr(H)^2}{Det(H)}}$$

Where ***H*** is the Hessian matrix of the image, ***Tr(H)*** is the trace of the matrix ***H***, ***DET(H)*** is the determinant of ***H***, ***R*** *is the ratio.*

3. **Orientation Assignment:**

   Depending on the image scale a neighbourhood of pixels is chosen and the orientation of each pixel relative to the keypoint pixel is calculated and used to create an orientation histogram with a pre-specified number of bins.

$$\boldsymbol{m(x,y) = \sqrt{(A(x+1,y) - A(x-1,y))^2 + (A(x,y+1) - A(x,y-1))^2}}$$

$$\boldsymbol{\theta(x,y) = tan^{-1}(\frac{A(x,y+1) - A(x,y-1)}{A(x+1,y) - A(x-1,y)})}$$

   The highest peak in the histogram is taken along with any value greater than 0.8 of this peak value. The result is used to create keypoint of the same location and scale but with different directions to increase the matching stability.

4. **The Descriptor:**

   A descriptor is created using the orientation bin histogram represented as a vector, forming keypoint descriptor along with some measurements to achieve robustness.

Once the descriptor is created it can be used to identify an object in another image by comparing this descriptor with target image's descriptor. Usually the matching process take into account the nearest neighbour, but sometimes the second-closest match is closer than the nearest match due to noise. Hence, a ratio between the closest and the second-closest distance is taken and if it's greater than 0.8 they are rejected. Overall steps for SIFT is shown in **Figure 3.12** .

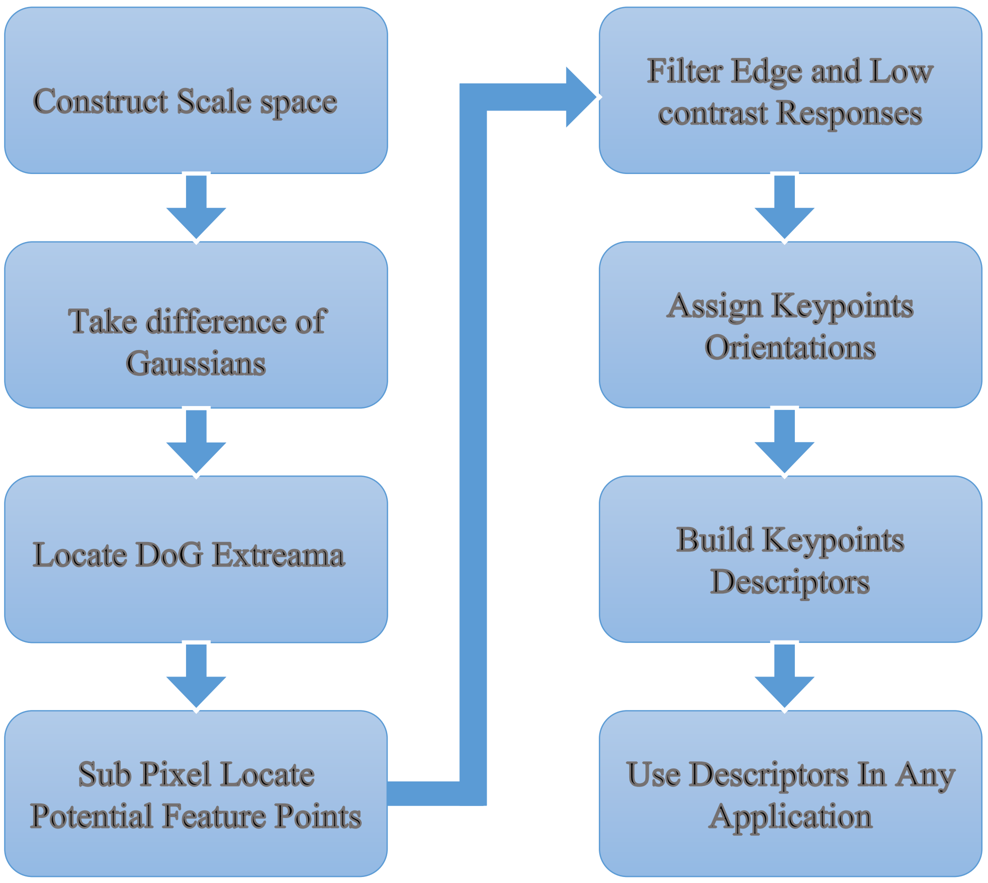

Construct Scale space
Take difference of Gaussians
Locate DoG Extreama
Sub Pixel Locate Potential Feature Points
Filter Edge and Low contrast Responses
Assign Keypoints Orientations
Build Keypoints Descriptors
Use Descriptors In Any Application

Figure 3.12: *SIFT steps flowchart.*

3.2.6.2 Simulink implementation

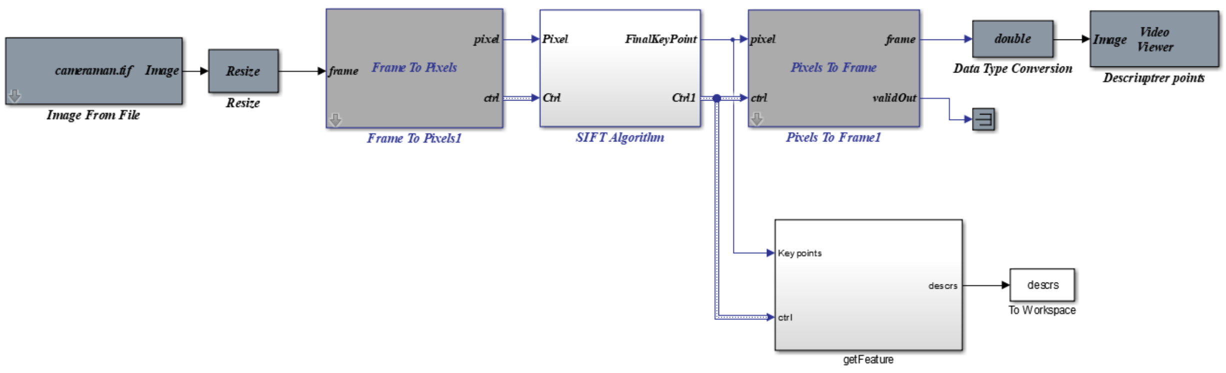


Figure 3.13: *SIFT Simulink model.*

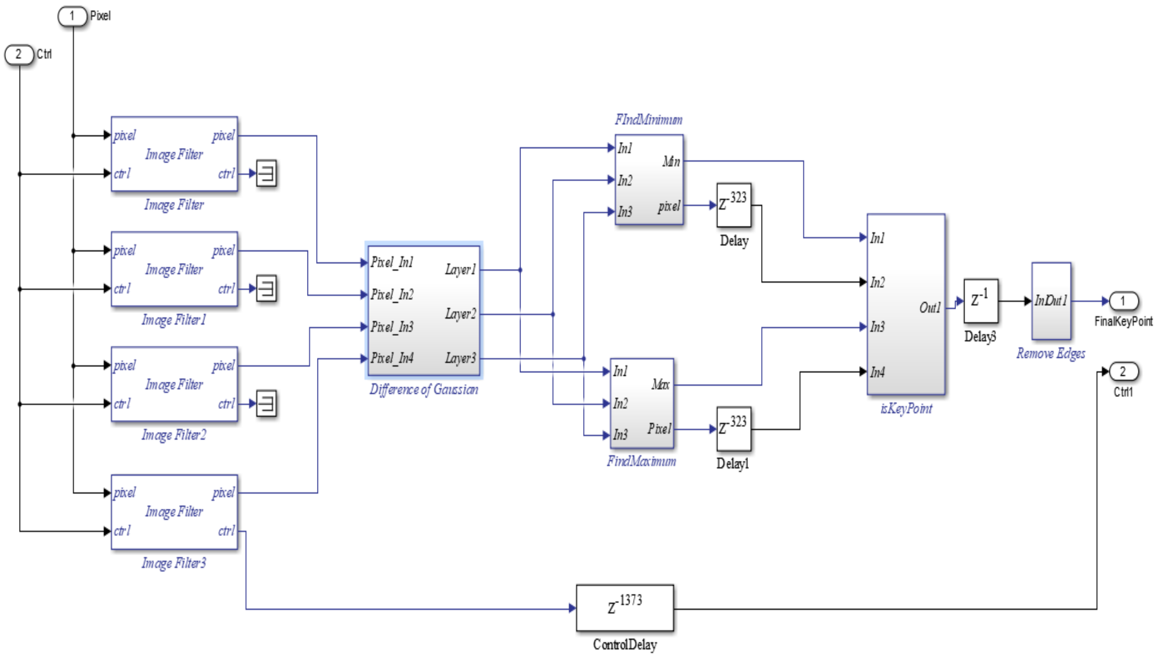


Figure 3.14: *SIFT algorithm subsystem.*

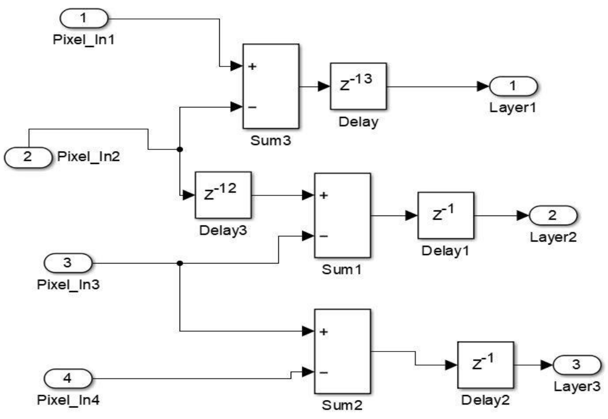


Figure 3.15: *Difference of Gaussian subsystem.*

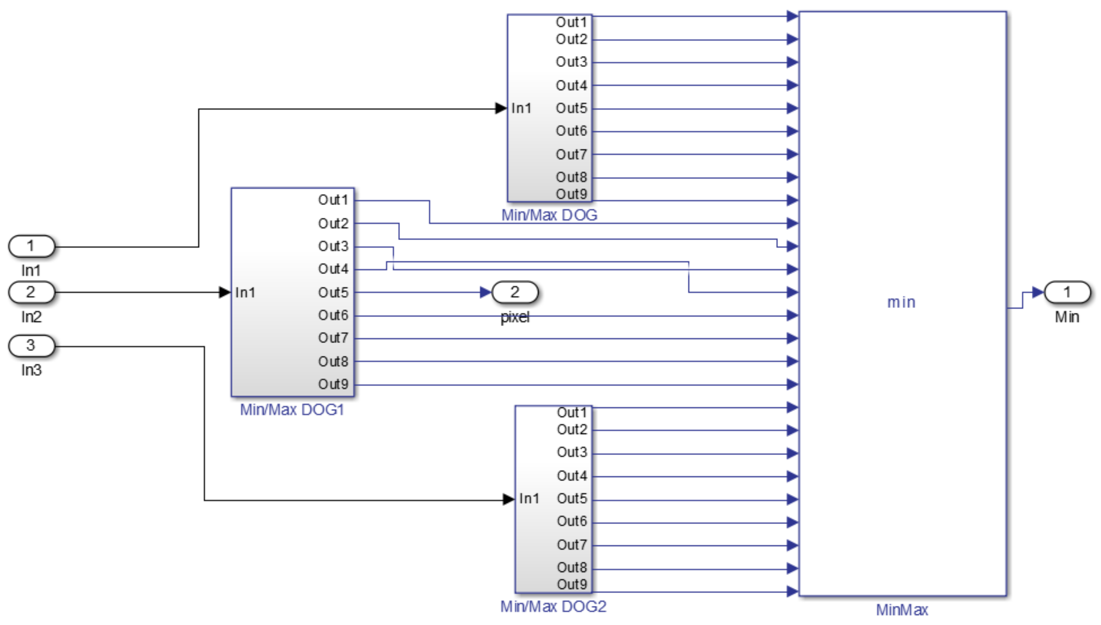


Figure 3.16: *FindMinimum subsystem.*

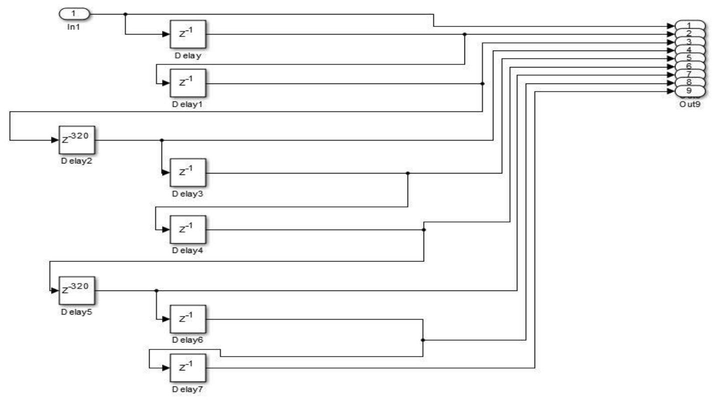


Figure 3.17: *Min/Max DOG subsystem.*

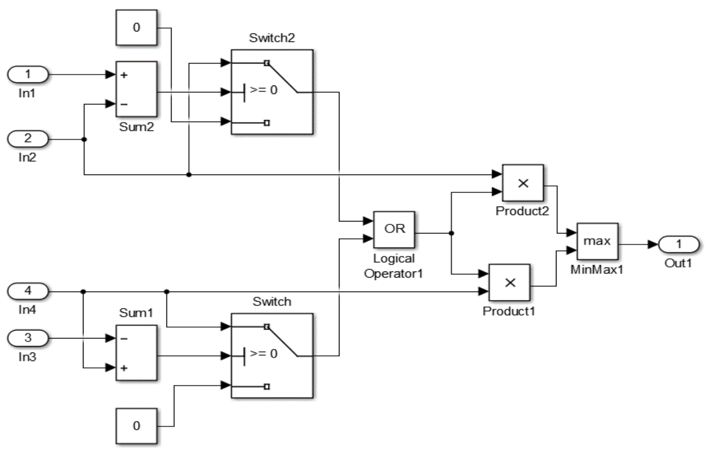


Figure 3.18: *isKeyPoint subsystem.*

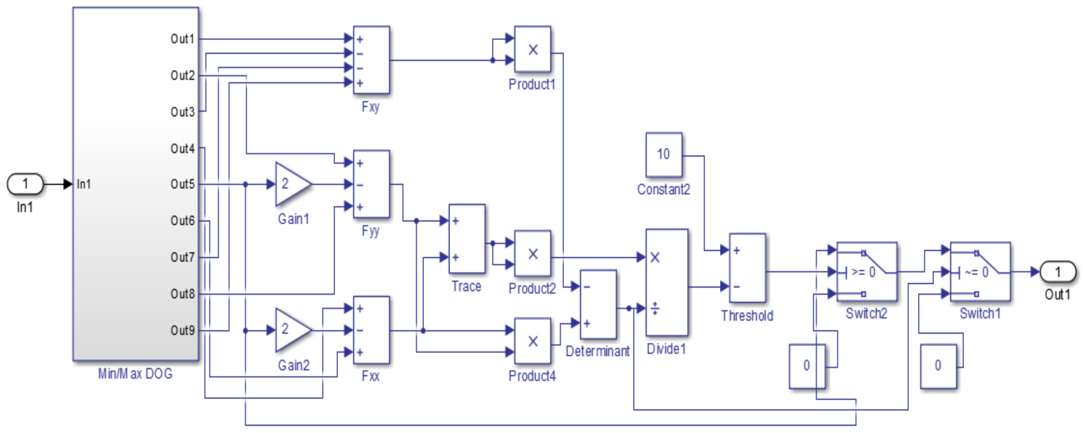


Figure 3.19: *Remove Edges subsystem.*

3.3 FPGA Implementation

3.3.1 Simulink configuration

Simulink configuration mostly focuses on two main configurations: Simulation time and datatype.

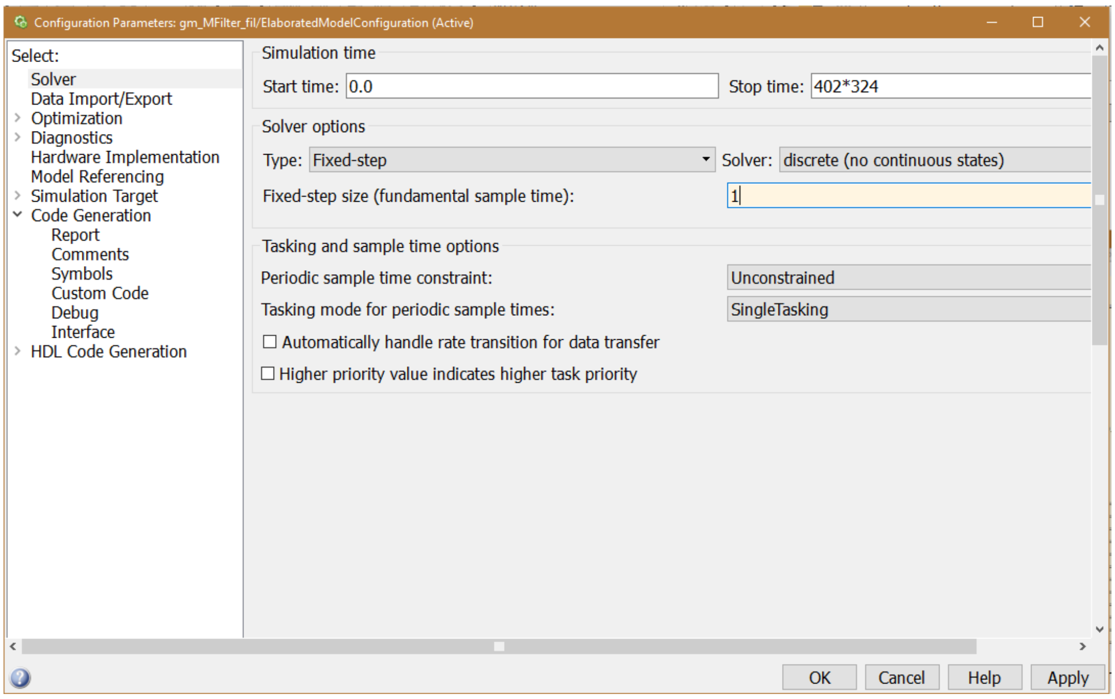
3.3.1.1 Simulation time

In order to synchronize the sample time with the FPGA clock period (to give FPGA function call 1 step time) we have to do the following:

- Sample time must be integer.
- Discrete (no continuous states) solver.

- Stop time is the number of pixels in image including all porch regions.
- Change automatic fixed step size (fundamental sample time) to manual calculation.

You can find the configuration window below in **Simulation>model configuration parameters>Solver**


Configuration Parameters: gm_MFilter_fil/ElaboratedModelConfiguration (Active)

Select:
- Solver
- Data Import/Export
- Optimization
- Diagnostics
- Hardware Implementation
- Model Referencing
- Simulation Target
- Code Generation
  - Report
  - Comments
  - Symbols
  - Custom Code
  - Debug
  - Interface
- HDL Code Generation

Simulation time

Start time: 0.0 Stop time: 402*324

Solver options

Type: Fixed-step Solver: discrete (no continuous states)

Fixed-step size (fundamental sample time): 1

Tasking and sample time options

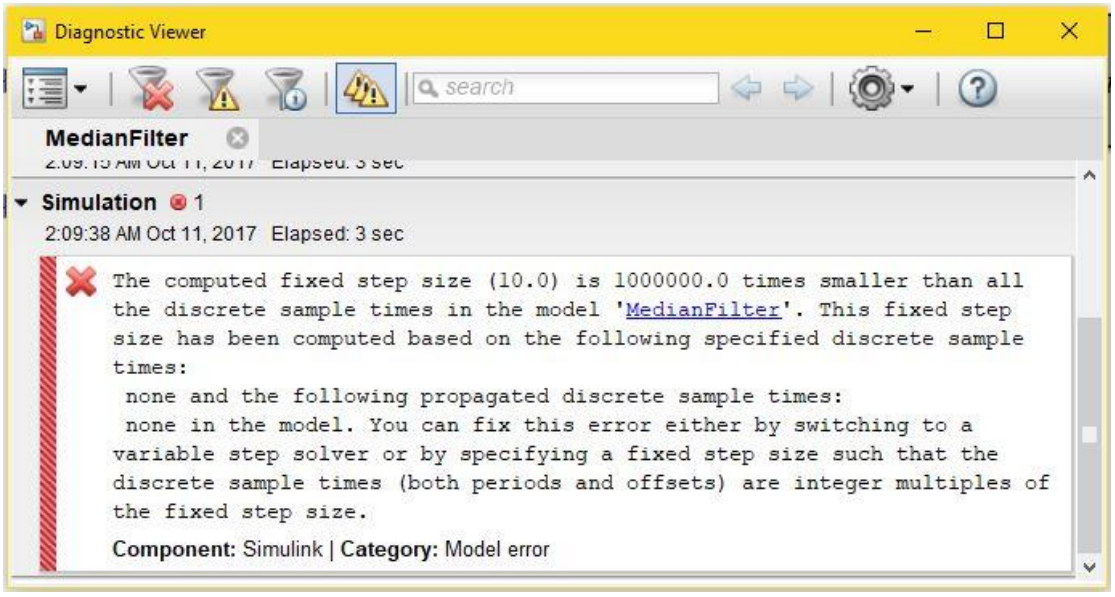
Periodic sample time constraint: Unconstrained

Tasking mode for periodic sample times: SingleTasking

☐ Automatically handle rate transition for data transfer

☐ Higher priority value indicates higher task priority

Figure 3.20: *Configuration window for simulation time.*


Diagnostic Viewer

MedianFilter

Simulation 1

2:09:38 AM Oct 11, 2017 Elapsed: 3 sec

```
The computed fixed step size (10.0) is 1000000.0 times smaller than all
the discrete sample times in the model 'MedianFilter'. This fixed step
size has been computed based on the following specified discrete sample
times:
 none and the following propagated discrete sample times:
 none in the model. You can fix this error either by switching to a
variable step solver or by specifying a fixed step size such that the
discrete sample times (both periods and offsets) are integer multiples of
the fixed step size.
```

Component: Simulink | **Category:** Model error

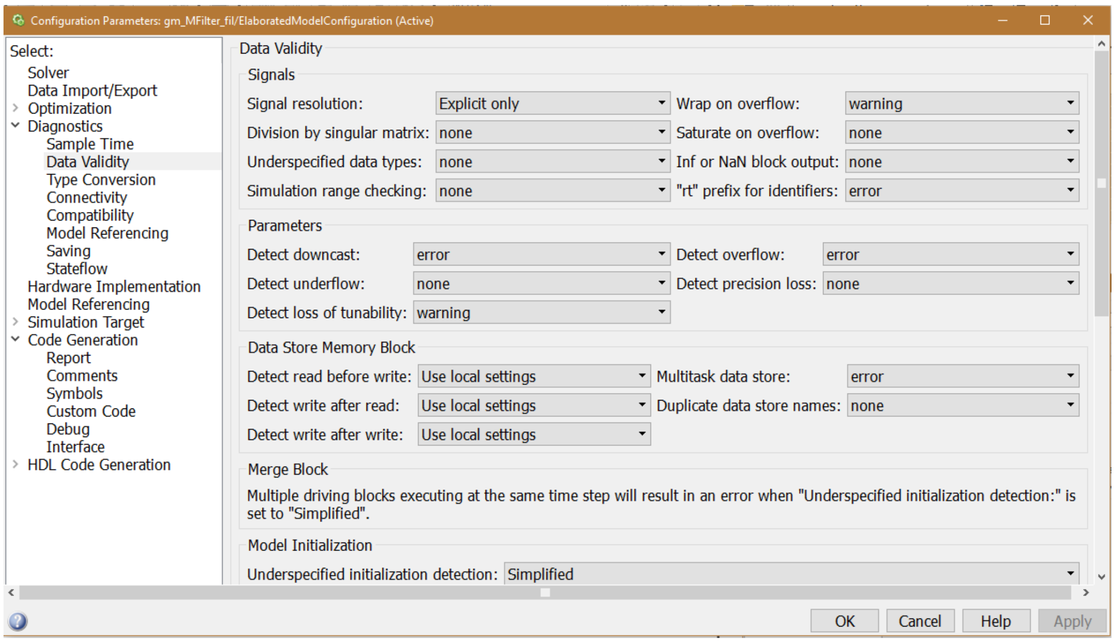
Figure 3.21: *Error that might occur if sample time configuration is not done correctly.*

3.3.1.2 Signals datatype (input and output datatype of any block in the model)

We use 16-bits fixed point datatype with 8-bits for fraction (because the largest value for each pixel is 255). These configurations must be done:

- Error in overflow and downcast.
- Use local flow for read before write and write and write after read.
- Error in multitask data store.

You can find the window below in **Simulation>model configuration parameters>Diagnostic>Data Validity**



Configuration Parameters: gm_MFilter_fil/ElaboratedModelConfiguration (Active)

Data Validity

Signals

Signal resolution:	Explicit only	Wrap on overflow:	warning
Division by singular matrix:	none	Saturate on overflow:	none
Underspecified data types:	none	Inf or NaN block output:	none
Simulation range checking:	none	"rt" prefix for identifiers:	error

Parameters

Detect downcast:	error	Detect overflow:	error
Detect underflow:	none	Detect precision loss:	none
Detect loss of tunability:	warning		

Data Store Memory Block

Detect read before write:	Use local settings	Multitask data store:	error
Detect write after read:	Use local settings	Duplicate data store names:	none
Detect write after write:	Use local settings		

Merge Block

Multiple driving blocks executing at the same time step will result in an error when "Underspecified initialization detection:" is set to "Simplified".

Model Initialization

Underspecified initialization detection: Simplified

Figure 3.22: *configuration window for signals datatype.*

3.3.2 HDL workflow advisor

The HDL Workflow Advisor is a tool that supports a number of tasks covering the stages of the FPGA design process. Some tasks perform model validation or checking; others run the HDL code generator or third-party tools. Each folder at the top level of the HDL Workflow Advisor contains a group of related tasks that you can select and run:

- **Set Target**: The tasks in this category enable you to select the desired target device and map its I/O interface to the inputs and outputs of your model.
- **Prepare Model For HDL Code Generation**: The tasks in this category check your model for HDL code generation compatibility. The tasks also report on model settings, blocks, or other conditions (such as algebraic loops) that would impede code generation, and provide advice on how to fix such problems.
- **HDL Code Generation**: This category supports all HDL-related options of the Configuration Parameters dialog, including setting HDL code and test bench generation parameters, and generating code, test bench, or a co-simulation model.
- **FPGA Synthesis and Analysis**: The tasks in this category support:
    - Synthesis and timing analysis through integration with third-party synthesis tools
    - Back annotation of the model with critical path and other information obtained during synthesis
- **FPGA-in-the-Loop Implementation**: This category implements the phases of FIL, including providing block generation, synthesis, logical mapping, PAR (place-and-route), programming file generation, and a communications channel. These capabilities are specifically designed for a particular board and tailored to your RTL code. An HDL Verifier™ license is required for FIL.
- **Download to Target**: The tasks in this category depend on the selected target device and might include:
    - Generation of a target-specific FPGA programming file.
    - Programming the target device.
    - Generation of a model that contains a Simulink Real-Time interface subsystem.

3.3.3 FIL

Use FIL in the Workflow Advisor, HDL Coder uses the loaded design to create the HDL code. This HDL code is then augmented by customized code for FIL communication with your design and assembled into an FPGA project. The applicable downstream tools are used to process that project to create a programming file that is automatically downloaded to the FPGA device on a development board for verification.

HDL Verifier supports the use of a FIL block in a model reference block and a System object™ in conjunction with a MATLAB program.**Table 3-1** shows the supported FPGA devices for FIL simulation.

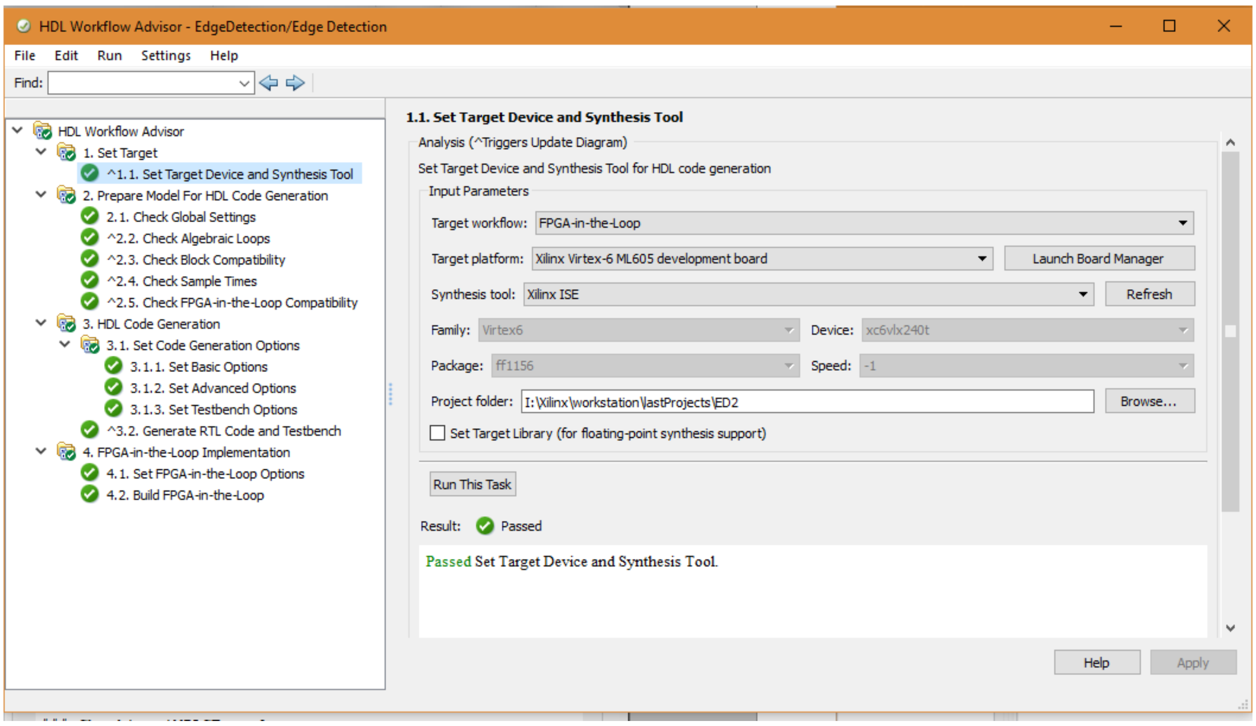
Table 3-1 *Supported FPGA Devices for FIL Simulation*

Vendor	Supported Devices	Required Hardware	Required Software
Altera	Any Altera FPGA board within the supported FPGA family, for example: Cyclone III, IV, V, and V SoC Arria II and Arria V Stratix IV and V Additional boards within the supported device families can be custom added with the FPGA Board Manager .	Altera FPGA board USB Blaster I or USB Blaster II download cable	Windows: Quartus II 12.1 or higher version, Quartus II executable directory must be on system path Linux: Quartus II 13.0sp1 with a patch, or Quartus II 13.1. Quartus II library directory must be on LD_LIBRARY_PATH *before* starting MATLAB, only 64-bit Quartus are supported Installation of USB Blaster I or II driver
Xilinx	The FPGA board must be using an FPGA device in the supported Xilinx FPGA family: Virtex 7, Kintex 7 , Artix® 7 and Zynq 7000. All Xilinx 7-series FPGA boards that HDL Verifier supports directly (as of this release) can perform simulation through an on-board Digilent JTAG cable. Additional boards using the supported FPGA devices can be added with the FPGA Board Manager.	The FPGA board must be using a Digilent download cable. If your board has a standard Xilinx 14 pin JTAG connector, you can obtain the HS2 cable from Digilent.	For Windows® operating systems: Xilinx Vivado 2014.2. Vivado executable directory must be on system path For Linux operating systems: Xilinx Vivado 2014.2 and Digilent Adept2

3.3.3.1 Configuration

Steps to implement FIL:

1. **Set target as FIL:**


HDL Workflow Advisor - EdgeDetection/Edge Detection

File Edit Run Settings Help

Find:

- HDL Workflow Advisor
  - 1. Set Target
    - ^1.1. Set Target Device and Synthesis Tool
  - 2. Prepare Model For HDL Code Generation
    - 2.1. Check Global Settings
    - ^2.2. Check Algebraic Loops
    - ^2.3. Check Block Compatibility
    - ^2.4. Check Sample Times
    - ^2.5. Check FPGA-in-the-Loop Compatibility
  - 3. HDL Code Generation
    - 3.1. Set Code Generation Options
      - 3.1.1. Set Basic Options
      - 3.1.2. Set Advanced Options
      - 3.1.3. Set Testbench Options
    - ^3.2. Generate RTL Code and Testbench
  - 4. FPGA-in-the-Loop Implementation
    - 4.1. Set FPGA-in-the-Loop Options
    - 4.2. Build FPGA-in-the-Loop

1.1. Set Target Device and Synthesis Tool

Analysis (^Triggers Update Diagram)

Set Target Device and Synthesis Tool for HDL code generation

Input Parameters

Target workflow: FPGA-in-the-Loop

Target platform: Xilinx Virtex-6 ML605 development board — Launch Board Manager

Synthesis tool: Xilinx ISE — Refresh

Family: Virtex6 Device: xc6vlx240t

Package: ff1156 Speed: -1

Project folder: I:\Xilinx\workstation\lastProjects\ED2 — Browse...

☐ Set Target Library (for floating-point synthesis support)

Run This Task

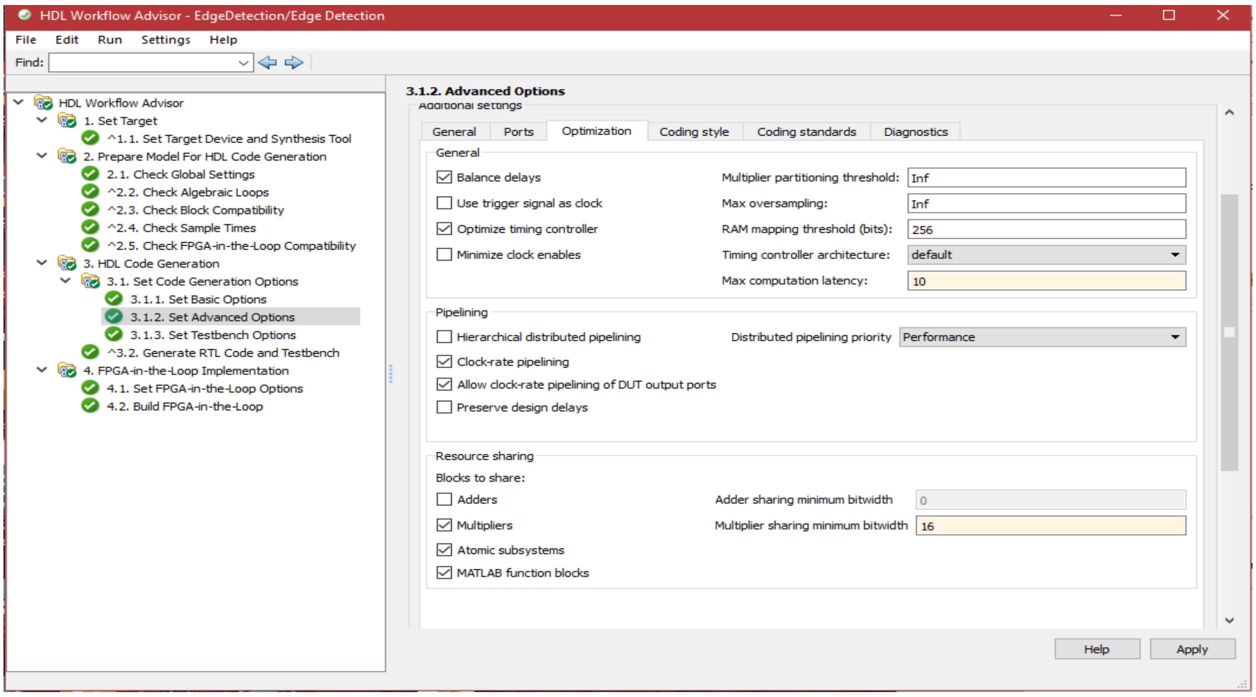
Result: ✓ Passed

Passed Set Target Device and Synthesis Tool.

Help Apply

Figure 3.23: *Step 1 for implementing FIL.*

2. **Optimize : in set advance options configure pipelining and number of resources.**



HDL Workflow Advisor - EdgeDetection/Edge Detection

File Edit Run Settings Help

Find:

- HDL Workflow Advisor
  - 1. Set Target
    - ^1.1. Set Target Device and Synthesis Tool
  - 2. Prepare Model For HDL Code Generation
    - 2.1. Check Global Settings
    - ^2.2. Check Algebraic Loops
    - ^2.3. Check Block Compatibility
    - ^2.4. Check Sample Times
    - ^2.5. Check FPGA-in-the-Loop Compatibility
  - 3. HDL Code Generation
    - 3.1. Set Code Generation Options
      - 3.1.1. Set Basic Options
      - 3.1.2. Set Advanced Options
      - 3.1.3. Set Testbench Options
    - ^3.2. Generate RTL Code and Testbench
  - 4. FPGA-in-the-Loop Implementation
    - 4.1. Set FPGA-in-the-Loop Options
    - 4.2. Build FPGA-in-the-Loop

3.1.2. Advanced Options

Additional settings

General | Ports | Optimization | Coding style | Coding standards | Diagnostics

General

- ☑ Balance delays
- ☐ Use trigger signal as clock
- ☑ Optimize timing controller
- ☐ Minimize clock enables

Multiplier partitioning threshold:	Inf
Max oversampling:	Inf
RAM mapping threshold (bits):	256
Timing controller architecture:	default
Max computation latency:	10

Pipelining

- ☐ Hierarchical distributed pipelining
- ☑ Clock-rate pipelining
- ☑ Allow clock-rate pipelining of DUT output ports
- ☐ Preserve design delays

Distributed pipelining priority: Performance

Resource sharing

Blocks to share:

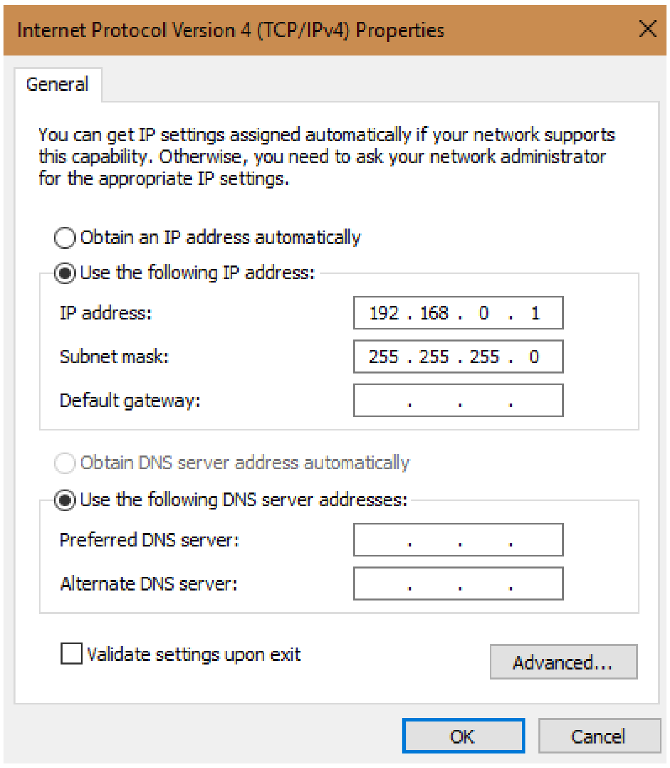
- ☐ Adders
- ☑ Multipliers
- ☑ Atomic subsystems
- ☑ MATLAB function blocks

Adder sharing minimum bitwidth	0
Multiplier sharing minimum bitwidth	16

Help Apply

Figure 3.24: *Step 2 for implementing FIL.*

3. Setup Local network and ARP table:



Internet Protocol Version 4 (TCP/IPv4) Properties

General

You can get IP settings assigned automatically if your network supports this capability. Otherwise, you need to ask your network administrator for the appropriate IP settings.

○ Obtain an IP address automatically

● Use the following IP address:

IP address:	192 . 168 . 0 . 1
Subnet mask:	255 . 255 . 255 . 0
Default gateway:	. . .

○ Obtain DNS server address automatically

● Use the following DNS server addresses:

Preferred DNS server:	. . .
Alternate DNS server:	. . .

☐ Validate settings upon exit

Advanced...

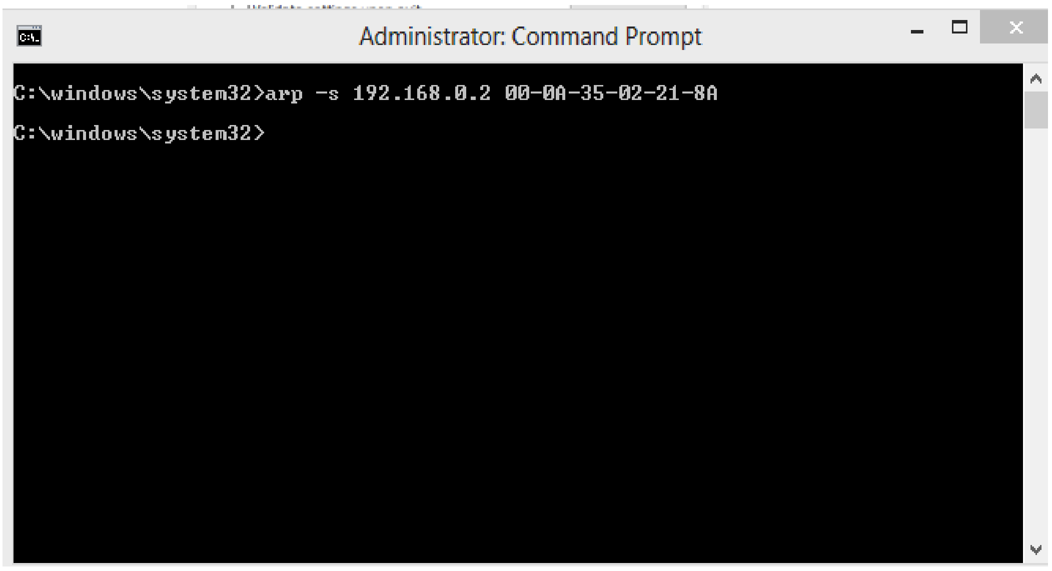
OK Cancel

Figure 3.25: *Step 3 for implementing FIL.*

To add FPGA Mac and IP pair to ARP table:

In Windows: With system administrator privileges, execute the following in a command shell:

arp -s 192.168.0.2 00-0A-35-02-21-8A



Administrator: Command Prompt

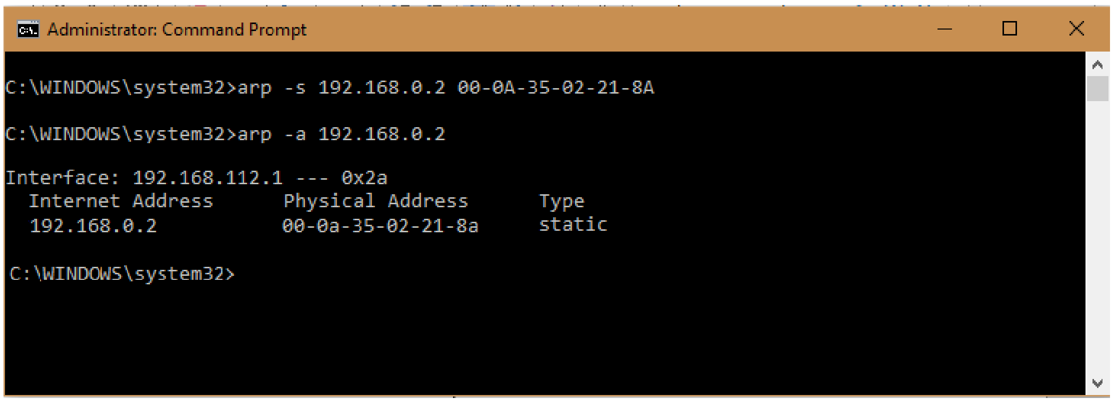
```
C:\windows\system32>arp -s 192.168.0.2 00-0A-35-02-21-8A

C:\windows\system32>
```

Figure 3.26: *adding FPGA MAC and IP pair to ARP table.*

To confirm that the operation was successful perform the command:

arp -a 192.168.0.2

you should see the IP and MAC pair you entered previously as in the figure below:

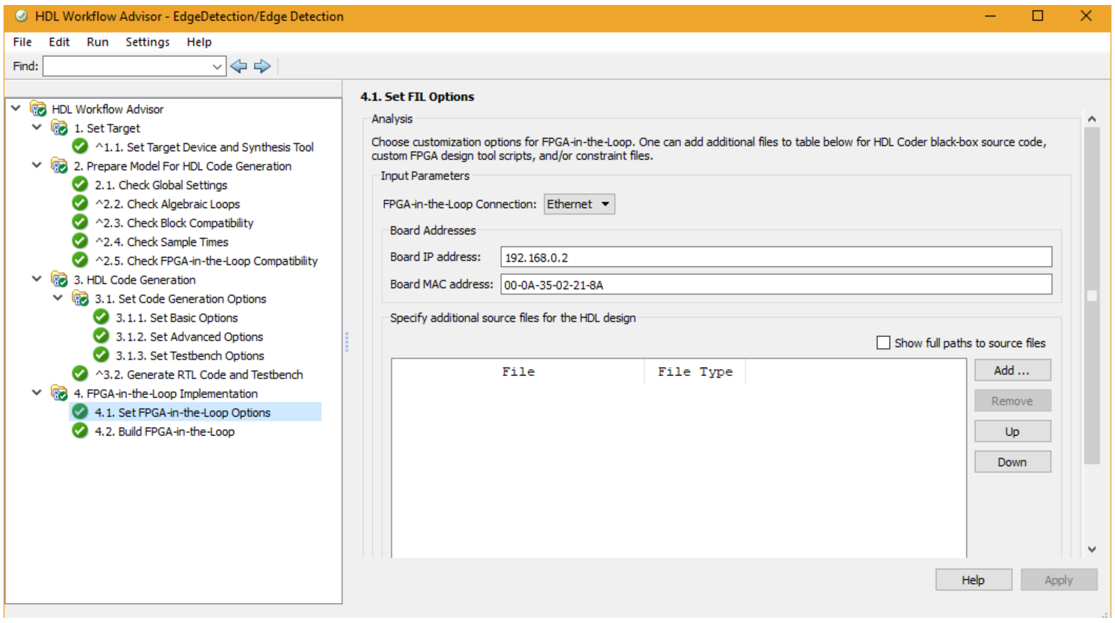


Administrator: Command Prompt

```
C:\WINDOWS\system32>arp -s 192.168.0.2 00-0A-35-02-21-8A

C:\WINDOWS\system32>arp -a 192.168.0.2

Interface: 192.168.112.1 --- 0x2a
  Internet Address      Physical Address      Type
  192.168.0.2           00-0a-35-02-21-8a     static

C:\WINDOWS\system32>
```

Figure 3.27: *checking the ARP table for FPGA entry.*

4. Configure MAC address and IP address in FPGA :


HDL Workflow Advisor - EdgeDetection/Edge Detection

File Edit Run Settings Help

Find:

- HDL Workflow Advisor
  - 1. Set Target
    - ^1.1. Set Target Device and Synthesis Tool
  - 2. Prepare Model For HDL Code Generation
    - 2.1. Check Global Settings
    - ^2.2. Check Algebraic Loops
    - ^2.3. Check Block Compatibility
    - ^2.4. Check Sample Times
    - ^2.5. Check FPGA-in-the-Loop Compatibility
  - 3. HDL Code Generation
    - 3.1. Set Code Generation Options
      - 3.1.1. Set Basic Options
      - 3.1.2. Set Advanced Options
      - 3.1.3. Set Testbench Options
    - ^3.2. Generate RTL Code and Testbench
  - 4. FPGA-in-the-Loop Implementation
    - 4.1. Set FPGA-in-the-Loop Options
    - 4.2. Build FPGA-in-the-Loop

4.1. Set FIL Options

Analysis

Choose customization options for FPGA-in-the-Loop. One can add additional files to table below for HDL Coder black-box source code, custom FPGA design tool scripts, and/or constraint files.

Input Parameters

FPGA-in-the-Loop Connection: Ethernet

Board Addresses

Board IP address: 192.168.0.2

Board MAC address: 00-0A-35-02-21-8A

Specify additional source files for the HDL design

☐ Show full paths to source files

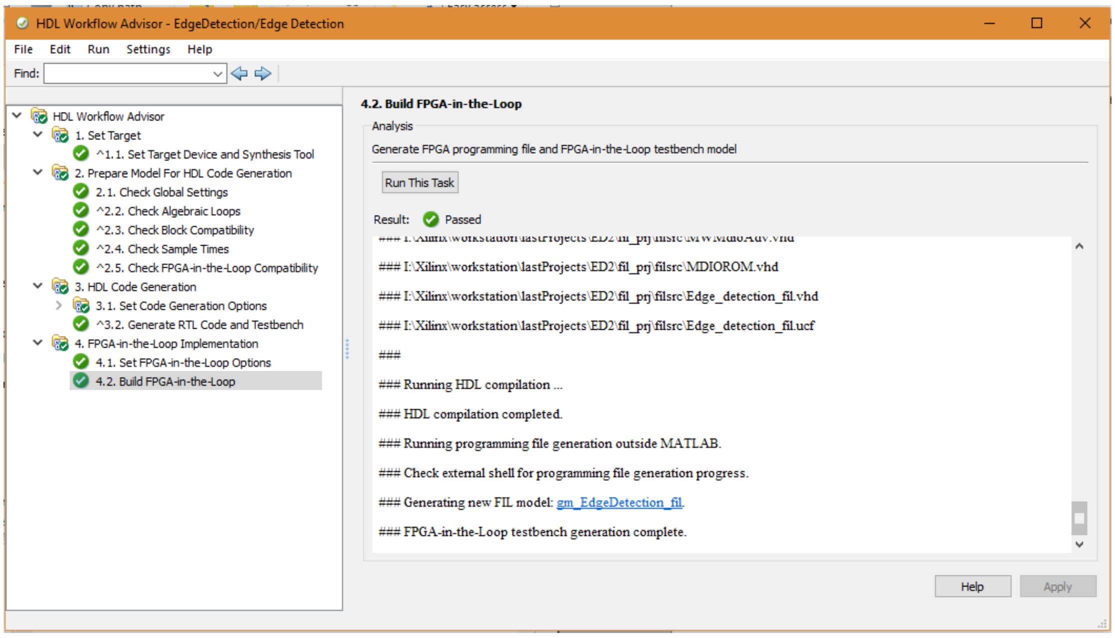
File	File Type

Add ... Remove Up Down

Help Apply

Figure 3.28: *Step 4 for implementing FIL.*

5. Build FPGA block and RUN HDL compilation:


HDL Workflow Advisor - EdgeDetection/Edge Detection

File Edit Run Settings Help

Find:

- HDL Workflow Advisor
  - 1. Set Target
    - ^1.1. Set Target Device and Synthesis Tool
  - 2. Prepare Model For HDL Code Generation
    - 2.1. Check Global Settings
    - ^2.2. Check Algebraic Loops
    - ^2.3. Check Block Compatibility
    - ^2.4. Check Sample Times
    - ^2.5. Check FPGA-in-the-Loop Compatibility
  - 3. HDL Code Generation
    - 3.1. Set Code Generation Options
    - ^3.2. Generate RTL Code and Testbench
  - 4. FPGA-in-the-Loop Implementation
    - 4.1. Set FPGA-in-the-Loop Options
    - 4.2. Build FPGA-in-the-Loop

4.2. Build FPGA-in-the-Loop

Analysis

Generate FPGA programming file and FPGA-in-the-Loop testbench model

Run This Task

Result: ✓ Passed

```
### I:\Xilinx\workstation\lastProjects\ED2\fil_prj\filsrc\MWMdioAdv.vhd
### I:\Xilinx\workstation\lastProjects\ED2\fil_prj\filsrc\MDIOROM.vhd
### I:\Xilinx\workstation\lastProjects\ED2\fil_prj\filsrc\Edge_detection_fil.vhd
### I:\Xilinx\workstation\lastProjects\ED2\fil_prj\filsrc\Edge_detection_fil.ucf
###
### Running HDL compilation ...
### HDL compilation completed.
### Running programming file generation outside MATLAB.
### Check external shell for programming file generation progress.
### Generating new FIL model: gm_EdgeDetection_fil.
### FPGA-in-the-Loop testbench generation complete.
```

Help Apply

Figure 3.29: *Step 5 for implementing FIL.*

3.3.3.2 Implementation in MATLAB

After completing the above configurations and steps we will end up with an output model like the model below:

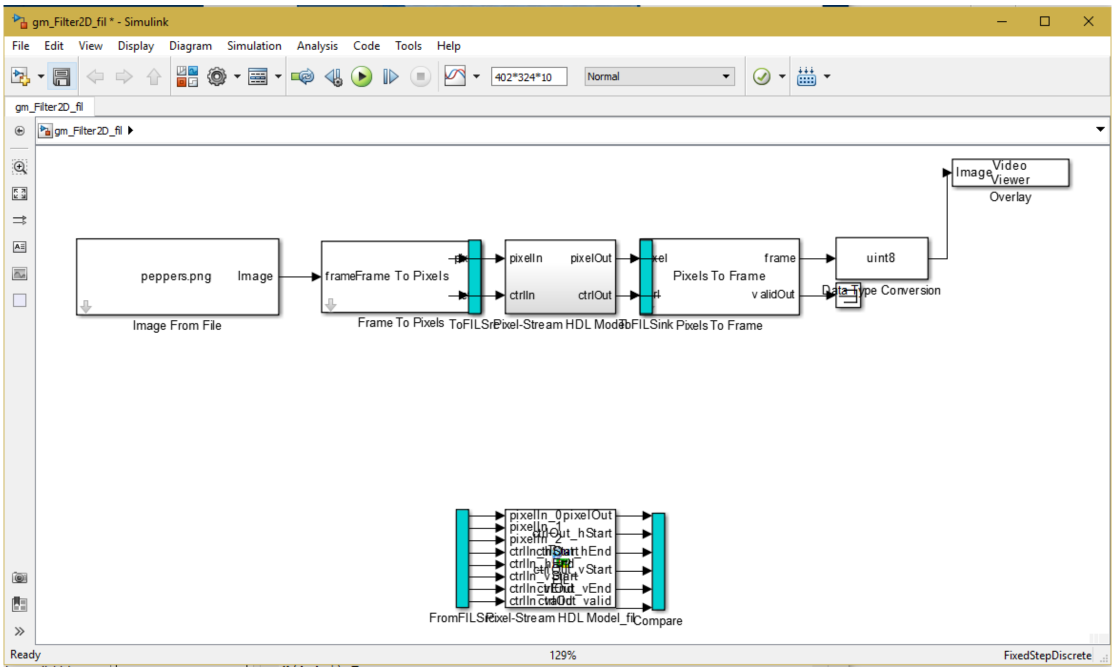


Figure 3.30: *FIL output model.*

You can modify the auto-generated model to use the FIL frame to pixels and FIL pixels to frame blocks to improve the bandwidth of the communication with the FPGA board by sending one frame at a time.

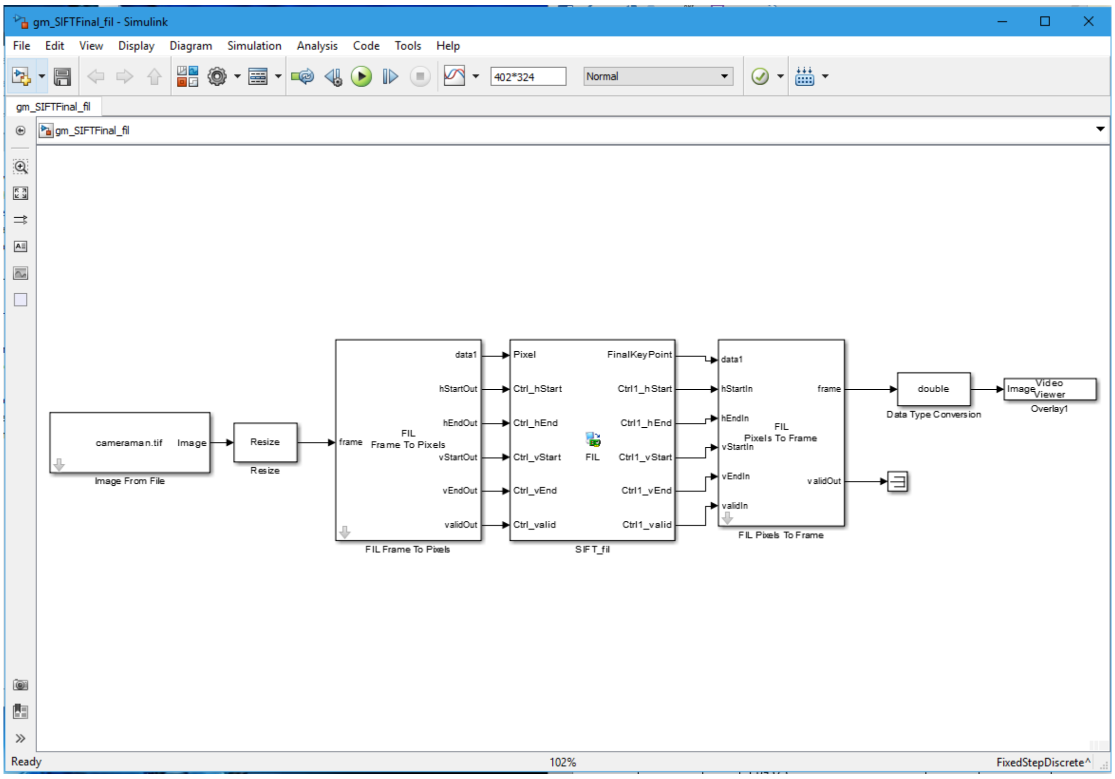


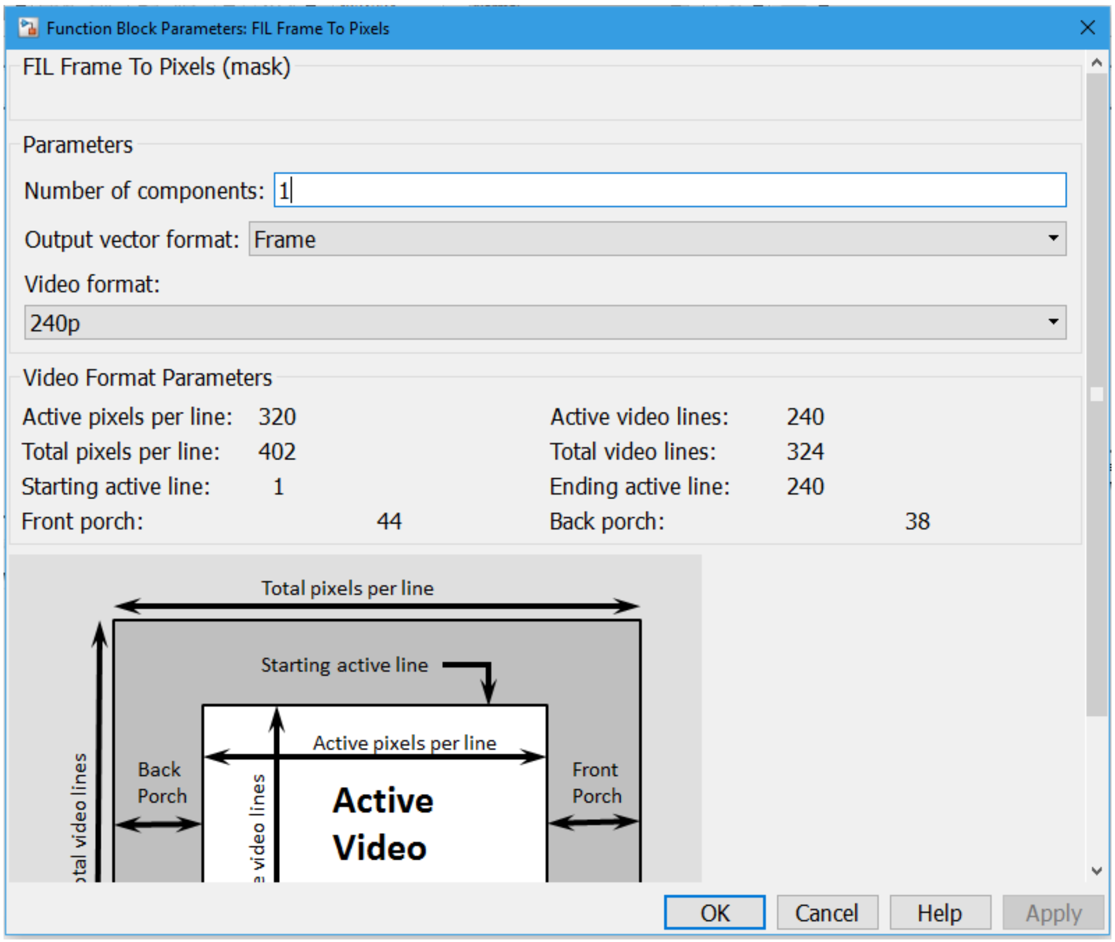
Figure 3.31: *FIL output block after putting FIL frame to pixels and pixels to frame blocks.*

In FIL frame to pixels, you can configure the width of the output vector to be a single pixel, a line, or an entire frame. The block returns control signals in vectors of the same width as the pixel data. This optimization makes more efficient use of the communication link between the FPGA board and the Simulink simulation when using FPGA-in-the-loop.

Options for the FIL blocks mentioned above are:

- **Pixel:** output scalar values for pixel and control signals.
- **Line:** output vectors contain total pixels per line values.
- **Frame:** output vectors contain total pixels per line× total video lines values.

Larger values result in faster communication between FPGA boards and Simulink. Choose the largest option that is supported by the I/O and memory resources on your board.


Function Block Parameters: FIL Frame To Pixels

FIL Frame To Pixels (mask)

Parameters

Number of components: 1

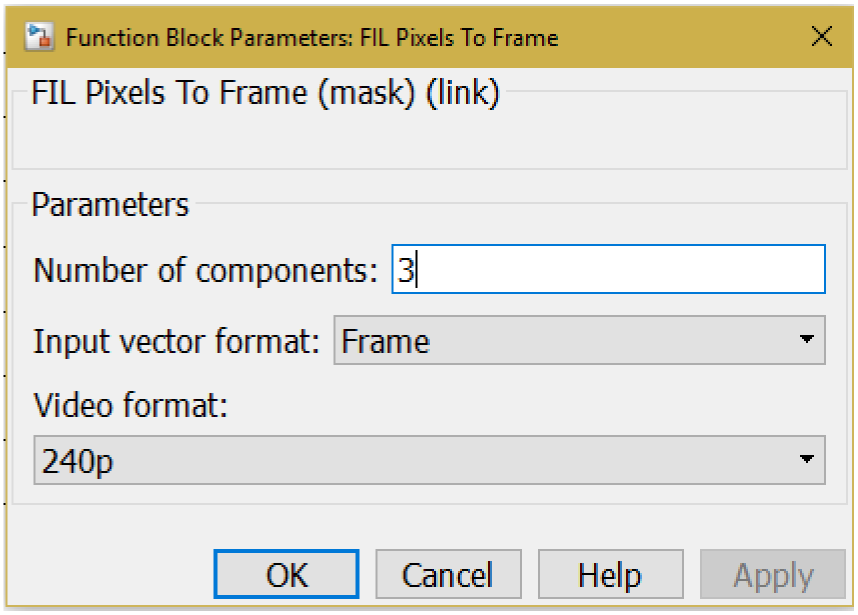
Output vector format: Frame

Video format: 240p

Video Format Parameters

Active pixels per line:	320	Active video lines:	240
Total pixels per line:	402	Total video lines:	324
Starting active line:	1	Ending active line:	240
Front porch:	44	Back porch:	38

Figure 3.32: *FIL frame to pixels block configuration.*


Function Block Parameters: FIL Pixels To Frame

FIL Pixels To Frame (mask) (link)

Parameters

Number of components: 3

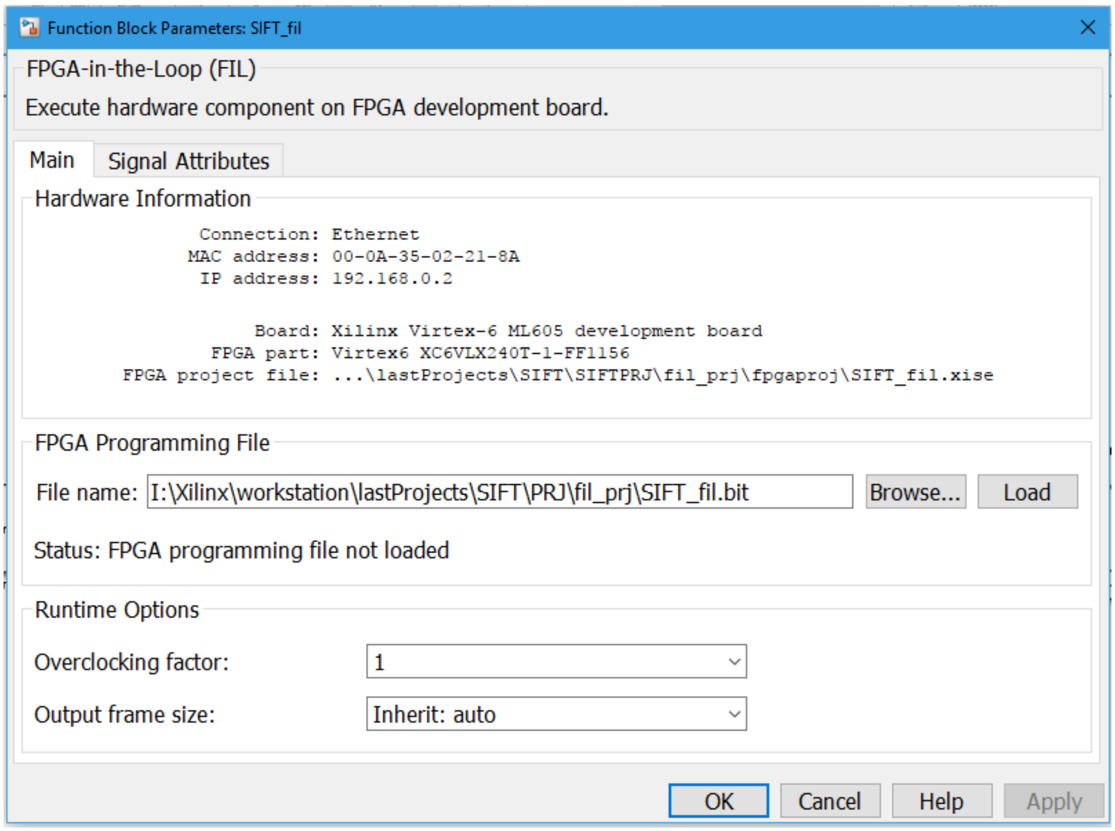
Input vector format: Frame

Video format: 240p

Figure 3.33: *FIL pixels to frame block configuration.*

Notice that in FIL frame to pixels and pixels to frame there is a parameter called "**Number of components**", this designate how many components will be there for the output. It should be 1 for binary and gray images and it should be 3 if the output is an RGB colored image.

After configuring FIL frame to pixels and pixels to frame blocks, we should load the .bit file we generated to the FPGA. By clicking on the generated block we see the following window for loading the file and setting the overclocking and output frame size:



FPGA-in-the-Loop (FIL)

Execute hardware component on FPGA development board.

Hardware Information

```
        Connection: Ethernet
       MAC address: 00-0A-35-02-21-8A
        IP address: 192.168.0.2

             Board: Xilinx Virtex-6 ML605 development board
         FPGA part: Virtex6 XC6VLX240T-1-FF1156
 FPGA project file: ...\lastProjects\SIFT\SIFTPRJ\fil_prj\fpgaproj\SIFT_fil.xise
```

FPGA Programming File

File name: I:\Xilinx\workstation\lastProjects\SIFT\PRJ\fil_prj\SIFT_fil.bit

Status: FPGA programming file not loaded

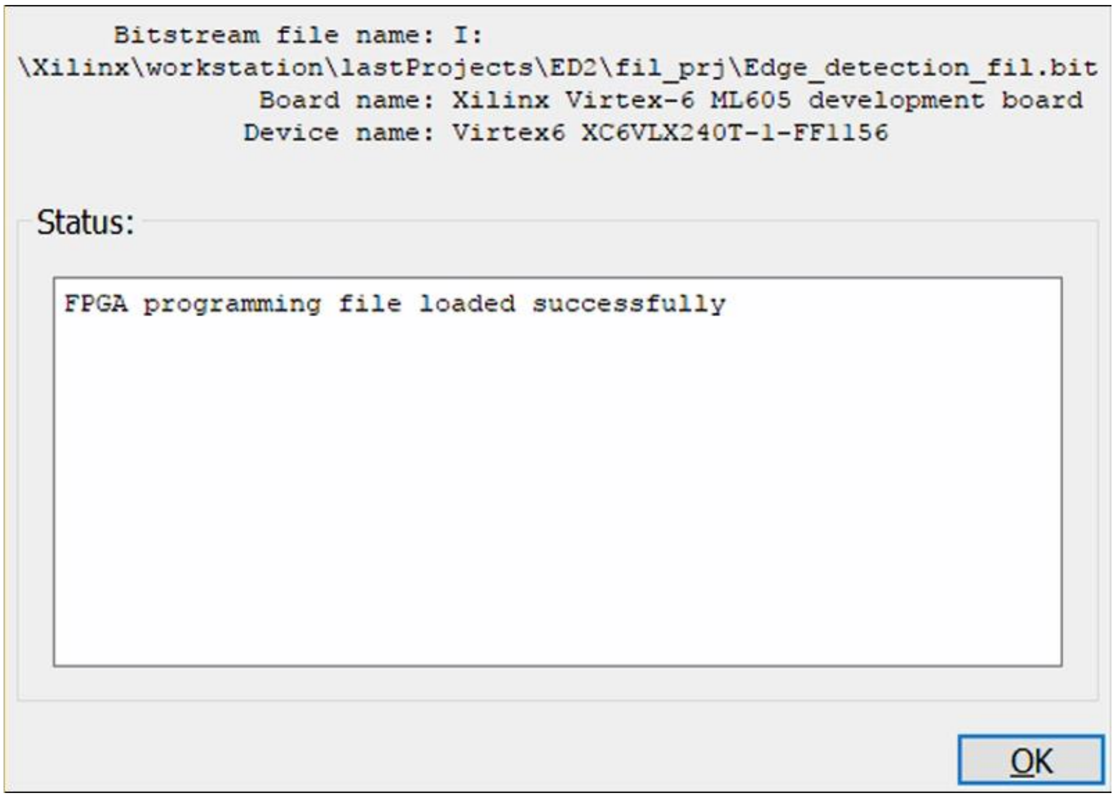
Runtime Options

Overclocking factor: 1

Output frame size: Inherit: auto

Figure 3.34: *configuration window to load the .bit file to the FPGA.*

After pressing the load button, a window should pop up with the progress of loading, at the end of the loading we should see the following window:


```
      Bitstream file name: I:
\Xilinx\workstation\lastProjects\ED2\fil_prj\Edge_detection_fil.bit
               Board name: Xilinx Virtex-6 ML605 development board
              Device name: Virtex6 XC6VLX240T-1-FF1156
```

Status:

```
FPGA programming file loaded successfully
```

Figure 3.35: *window for successful loading of the .bit file to the FPGA.*

3.4 Acquiring Results

3.4.1 Time analysis options in MATLAB[12]

FIL block have internal configurations to measure execution time of FPGA in MATLAB without the need for external hardware or the need to monitor MATLAB processes in CPU.

Under runtime options, set the overclocking factor to 1 to specify that each input value is sampled once by the FPGA's internal board clock before the input changes. The number of samples by one call (FPGA work) depends on the image size.

When output frame size is set as inherited, the output frame size will be as the input frame size. In case of series of pixels in the input, the output will be one pixel and the function calls will depend on the size of the image.

3.4.1.1 TIC-TOC

Tic starts a stopwatch timer to measure performance. The function records the internal time at the execution of tic command.

Toc reads the elapsed time from the stopwatch timer started by the tic function. The function reads the internal time at the execution of the toc command and displays the elapsed time since the most recent call to the tic function that had no output, in seconds.

The clock function is based on the system time, which can be adjusted periodically by the operating system, and thus might not be reliable in time comparison operations.

3.4.1.2 CPUTIME

cputime returns the total CPU time (in seconds) used by your MATLAB application from the time it was started. This number can overflow the internal representation and wrap around.

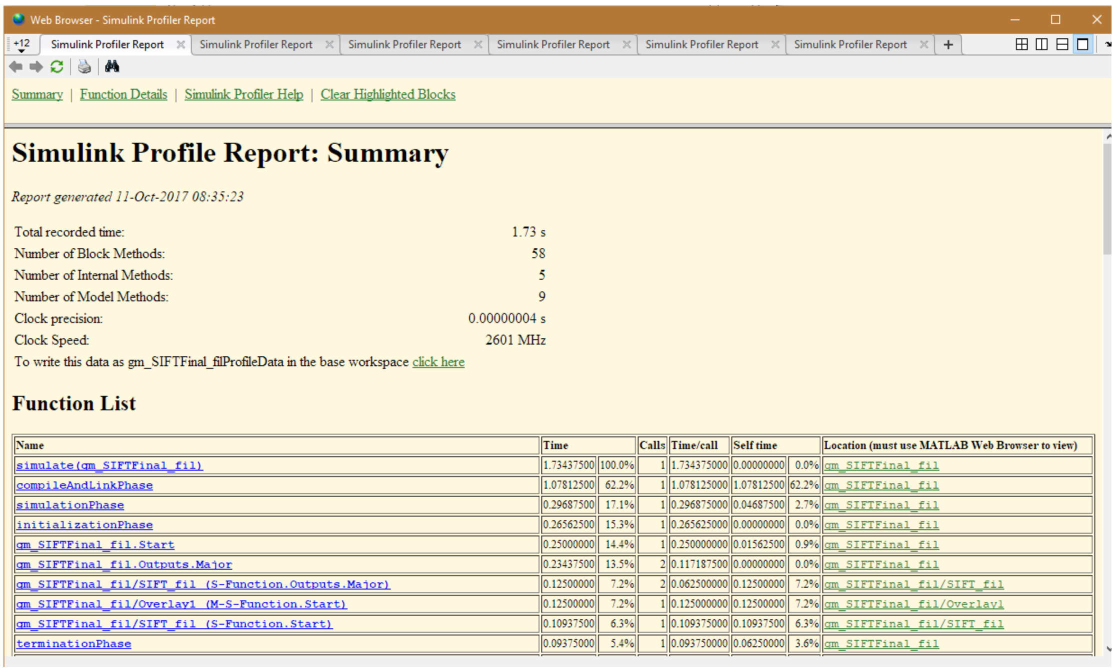
3.4.1.3 Comparing cputime and tic-toc :

Although you can measure performance using the cputime, tic and toc functions are better for this purpose. Generally for CPU-intensive calculations run on Microsoft Windows machines, the elapsed time using the two techniques are close in value, ignoring any first time costs. There are cases, however, that show a significant difference between these methods. For example, in case of a Pentium 4 with hyperthreading running windows, there can be a significant difference between the values returned by cputime versus tic and toc. Tic-toc method provides more reliable results than cputime.

3.4.2 Profile advisor

In Simulink Profile report includes:

- **Total recorded time:** Total time for simulating the model.
- **Number of Model Methods:** Number of methods called by the model.
- **Clock precision:** Precision of the profiler's time measurement.
- **Function List:**
    - Time: Time spent in this function, including all child functions called.
    - Calls: Number of times this function was called.
    - Time/call: Time spent per call.
    - Self-time: Total time spent in this function, not including any calls to child functions.
    - Location: Link to the location of the block in your model.



Summary | Function Details | Simulink Profiler Help | Clear Highlighted Blocks

Simulink Profile Report: Summary

Report generated 11-Oct-2017 08:35:23

Total recorded time:	1.73 s
Number of Block Methods:	58
Number of Internal Methods:	5
Number of Model Methods:	9
Clock precision:	0.00000004 s
Clock Speed:	2601 MHz

To write this data as gm_SIFTFinal_filProfileData in the base workspace click here

Function List

Name	Time		Calls	Time/call	Self time		Location (must use MATLAB Web Browser to view)
simulate(gm_SIFTFinal_fil)	1.73437500	100.0%	1	1.734375000	0.00000000	0.0%	gm_SIFTFinal_fil
compileAndLinkPhase	1.07812500	62.2%	1	1.078125000	1.07812500	62.2%	gm_SIFTFinal_fil
simulationPhase	0.29687500	17.1%	1	0.296875000	0.04687500	2.7%	gm_SIFTFinal_fil
initializationPhase	0.26562500	15.3%	1	0.265625000	0.00000000	0.0%	gm_SIFTFinal_fil
gm_SIFTFinal_fil.Start	0.25000000	14.4%	1	0.250000000	0.01562500	0.9%	gm_SIFTFinal_fil
gm_SIFTFinal_fil.Outputs.Major	0.23437500	13.5%	2	0.117187500	0.00000000	0.0%	gm_SIFTFinal_fil
gm_SIFTFinal_fil/SIFT_fil (S-Function.Outputs.Major)	0.12500000	7.2%	2	0.062500000	0.12500000	7.2%	gm_SIFTFinal_fil/SIFT_fil
gm_SIFTFinal_fil/Overlay1 (M-S-Function.Start)	0.12500000	7.2%	1	0.125000000	0.12500000	7.2%	gm_SIFTFinal_fil/Overlay1
gm_SIFTFinal_fil/SIFT_fil (S-Function.Start)	0.10937500	6.3%	1	0.109375000	0.10937500	6.3%	gm_SIFTFinal_fil/SIFT_fil
terminationPhase	0.09375000	5.4%	1	0.093750000	0.06250000	3.6%	gm_SIFTFinal_fil

Figure 3.36: *Example of profile report.*

Chapter 4 : Results Discussion and Analysis

4.1 Results

4.1.1 Languages used for FPGA

It was found that MATLAB is suitable tool for implementing image processing and computer vision algorithms in FPGA.

4.1.2 Performance

FPGA working time and the number of calls from the CPU were obtained to measure the performance. Also, the time of the simulation phase and the total time were obtained. The difference between total time and simulation time is the initialization and compilation time. Results were obtained in two modes with respect to pixels transmitting:

- **Pixel by pixel:**
  In this mode one pixel is sent at a time. Multiple images with different sizes were used. In each experiment, a different image was used. Results are shown in **Table 4-1** .
  The total time includes the initialization time and compilation time plus the simulation time. The initialization and compilation times are small, hence, the difference between the total time and simulation time is small.
  Pixels transmitting time illustrates the total time of sending and receiving the pixels by MATLAB. The number of FPGA calls is relative to the image size.
- **A whole frame:**
  In this mode, the whole frame is transmitted to the FPGA and then the CPU calls the FPGA to perform its processing. Each algorithm was tested using one image. The results are shown in **Table 4-2** .

4.1.3 Hardware resources used

In our research, we used ***CPU*** Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz and FPGA Xilinx Virtex 6 (XC6VLX240T). This type of FPGA has DSP(64-bit adder and 25×18 multiplier) and two sizes of RAM (18 KB and 38 KB) and has an LUT with 6 inputs. In **Table 4-3** we find the resources needed by each algorithm.

In **Table 4-3 connections** means constraint connections which tells us how many connections there are between end points.

Critical path is the path that spends the takes the maximum time. **Levels of logic** means the number of LUTs between two end points in the critical path.

Table 4-1: *Results when using Pixel by pixel mode.*

ALGORITHM	IMAGES	FPGA TIME		FPGA CALLS	WRITE IN/OUT IN MATLAB	SIMULATION PHASE	TOTAL TIME
Edge Detection	1	5.21875	21.3%	130249	13.703125	22.875	24.546875
	2	5.359375	22.4%	130249	13.140625	22.34375	23.937500
	3	5.484375	17.5%	130249	20.5625	29.50	31.281250
	4	5.26562	16.2%	130249	21.1400	30.781	32.531250
	5	8.390625	15.9%	188720	34.859375	50.59375	52.703125
	6	12.71875	17.2%	290957	48.109375	71.84375	73.812500
SIFT	_	4.578125	14.6%	130249	19.796875	29.5625	31.328125
Gamma Correction	_	5.640625	16.9%	130249	22.5625	31.75	33.390625
Degamma Correction	_	5.5625	16.4%	130249	22.21875	32.09375	33.859375
Color Correction	1	17.96875	17.0%	420001	73.96875	103.40625	105.515625
	2	17.984375	17.1%	420001	74.0625	102.6875	105.28125
2D Filter	1	5.500000	20.1%	130249	14.671875	24.78125	27.37500
	2	5.265625	9.5%	130249	21.5625	33.09375	55.37500
	3	5.750000	15.8%	130249	23.734375	32.84375	36.28125
	4	8.656250	19.3%	130249	31.0625	42.4375	44.953125
	5	4.843750	15.4%	130249	19.171875	28.90625	31.484375

Median Filter	1	1.265625	4.1%	26239	4.765625	9.90625	30.640625
	2	5.890625	16.8%	130249	22.9375	32.78125	35.031250

Table 4-2 : *Results when using whole frame mode.*

ALGORITHM	IMAGES	FPGA TIME		FPGA CALLS	IN CPU
Edge Detection	1080p	0.1093	8.4%	1	1.29625
Gamma Correction	1080p	0.0781	15%	1	0.515625
SIFT	240p	0.1246	1.53%	1	8.1603
2D Filter	480p	0.1406	16.9%	1	0.8275
Median Filter	1080p	1.083	11.4%	1	9.453125

Table 4-3: *Resources needed by each algorithm.*

ALGORITHM	LUTS	LUT-FF PAIRS	REGIS-TERS	L/O PINS	RAM/FIFO		CLOCKS BUFFERS	DSP DSP48E1s	Connections	Maximum frequency	Levels of logic
Color Correction	3409	1419	2112	30	3		4 BUFG	6	17293	100.756M Hz	7
					2 RAMB36E1	*1 RAMB18E1*					
Gamma And Degamma Correction	3003	3167	2034	30	3		4 BUFG	0	16539	108.225M Hz	7
					2 RAMB36E1	*1 RAMB18E1*					
Edge Detection	3607	3972	2808	30	6		4 BUFG	5	19387	89.969M Hz	14
					2 RAMB36E1	*4 RAMB18E1*					
2D Filter	3685	2099	3026	30	7		4 BUFG	3	19796	87.758M Hz	13
					2 RAMB36E1	*5 RAMB18E1*					
Median Filter	4804	5372	3773	30	15		4 BUFG	0	26712	73.276M Hz	7
					2 RAMB36E1	*13 RAMB18E1*					
SIFT	19037	22548	15496	30	83		4 BUFG	5	88859	6.695MHz	300
					2 RAMB36E1	*81 RAMB18E1*					

4.1.4 Peak signal to noise ratio (PSNR)

The PSNR block computes the peak signal to noise ratio in decibels between two images. This ratio is often used as a quality measurement between the original and the resulting image. The higher the PSNR the better the quality of the resulting image.

The Mean Square Error (MSE) and the PSNR are the two error measurements that were used in our research to compare image transformation quality. MSE represents the cumulative squared error between the transformed and the original images. Whereas PSNR represents a measure of the peak error. The lesser the value of MSE the lesser the error.

Below are the equations for MSE and PSNR respectively:

$$MSE = \frac{\sum_{M,N}(I_1(m,n) - I_2(m,n))^2}{M * N}$$

Where ***M and N*** are the dimensions of the image.

$$PSNR = 10\log_{10}(\frac{R^2}{MSE})$$

Where ***R*** is the maximum value for a pixel.

The general relation between PSNR and MSE is given below:

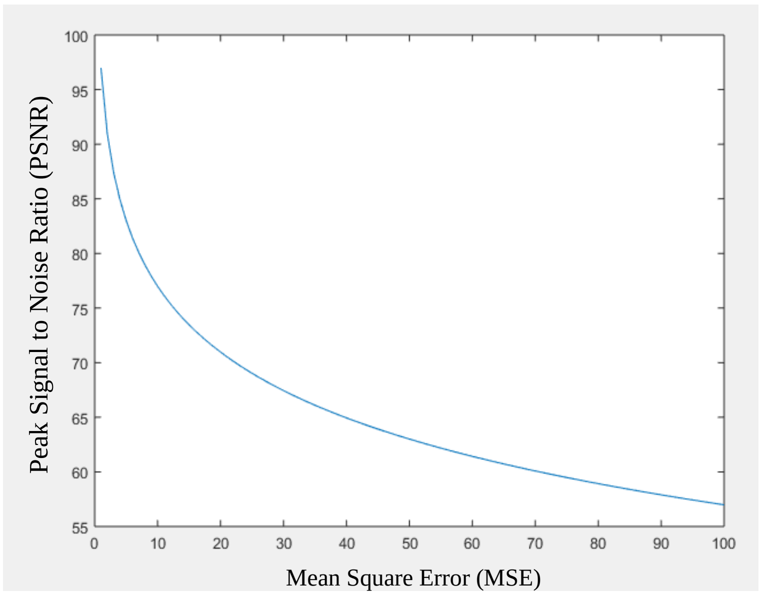


Figure 4.1: *relationship between PSNR and MSE.*

In our experiments, PSNR for the previous results for each algorithm was calculated using two approaches:

- **Using Simulink vision HDL blocks:**

  All algorithms gave PSNR = ∞ . Thus no noise was detected.

- **Using Simulink computer vision blocks:**

  For each algorithm, PSNR was obtained using several images. The results are shown in **Table 4-4**.

Table 4-4: *PSNR values for some of the algorithms before and after removing some rows and columns.*

ALGORITHM	PSNR	PSNR after removing some rows or columns
Edge Detection	35-40	Not noise PSNR=infinity
Median Filter	28-30	Not noise PSNR=infinity
Gamma Correction	41-48	44-62
2D filter	38-43	49-80

4.2 Analysis and Discussion

4.2.1 Languages used for FPGA

Due to the reasons mentioned in chapter 3, MATLAB is a suitable tool for implementing image processing and computer vision in FPGAs

4.2.2 Performance

When using the first mode (Pixel by pixel), its seen that the FPGA stays idle for a while waiting for the next call. This is why the overall processing time is increased. Also it was seen that pixel transmitting took about 2 to 4 times the FPGA time, which means that most of the process time was spent on the transmission of pixels between MATLAB and FPGA. Using pixel by pixel mode has a negative effect on the usage of the FPGA.

When using the second mode (whole frame), its noticed that the FPGA calls were reduced to only one call. This is because the FPGA is only called when the frame is fully transmitted. In general, FPGA time is about 10% of the CPU time.

Thus, using the whole frame mode maximizes the usage of the FPGA.

From **Table 4.3** we find that SIFT algorithm's levels of logic are 300 with ratio between logic and route of 37.1% logic and the rest 62.9% route. The accepted logic to route ratio is less than 40% logic. Thus our algorithm is in the accepted range.

It was found that the speedup (inverse of the ratio between FPGA time to CPU time) of SIFT is greater than the other algorithms. This is because the levels of logic for critical path and the number of paths are larger than the other algorithms.

Median filter was used in an experiment to find a relationship between CPU time and FPGA time. Multiple image resolutions were tested and the graph of the relationship found is in the figure below.

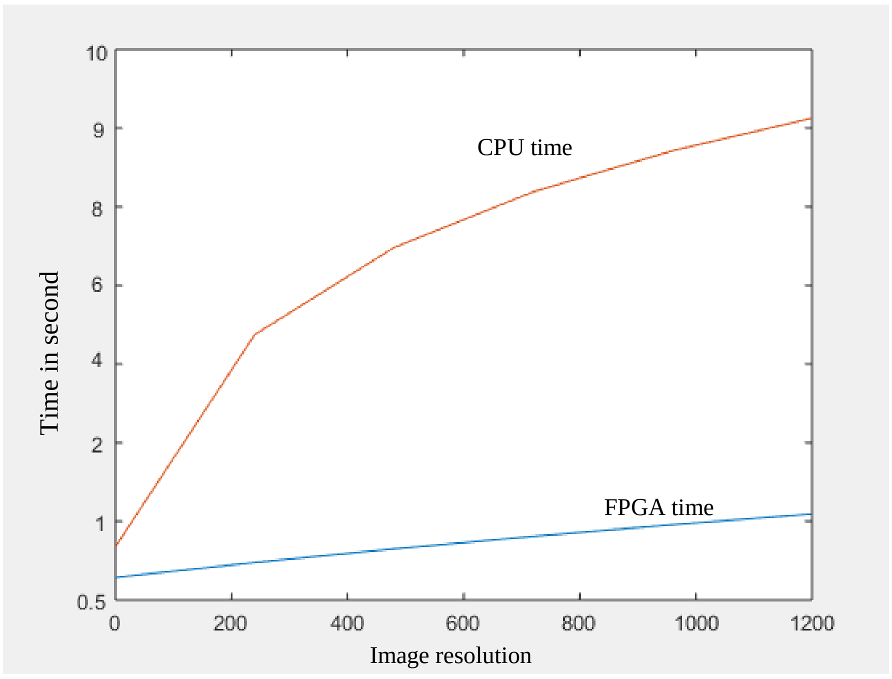


Figure 4.2: *relation between image resolution and time in seconds for median filter in FPGA and CPU.*

4.2.3 PSNR

In median filter and edge detection, the noise was noticed to be in the border of the pixels. Thus by removing the first and the last rows and columns, the PSNR calculated was infinity (∞) and no noise was detected.

In Gamma correction, it was found that PSNR had a lower value. This is due to the use of lookup table in the implementation, thus the mapping resulted only in integer values. Thus the error caused by truncating the decimal points decreased the value of the PSNR.

In 2D filter, PSNR obtained was low. This was expected because integer multiplication and division were used. The floating point is rounded to the nearest integer value. Which means truncating the decimal points.

If we want to improve PSNR to become greater than 90, one approach is to use fixed-point type in the output blocks instead of integer type.

Chapter 5 : Conclusion and Future Work

5.1 Conclusion

In order to identify the state of completion for the project, individual objectives and the progress done in each one is show below:

- **To compare between languages used to program FPGAs and find a suitable solution for image processing and computer vision applications:** it had been found that MATLAB is a suitable tool for image processing and computer vision applications in FPGAs.
- **To compare between CPU and FPGA in the following areas (Processing time, Hardware resources needed, Accuracy "peak signal to noise ratio"):** it was found that it is better to use whole frame transmission instead of pixel by pixel transmission. When using FPGA the speed was found to be a factor of 9× greater than the CPU speed in standard models of image processing. Regarding to PSNR it was found to be low for some algorithms but it is generally in the accepted range.

In addition to the initial objectives of the project, SIFT algorithm was implemented using Simulink, then it was implemented in FPGA.

5.2 Future work

- To implement the algorithms mentioned in chapter 3 in multiple FPGA types and compare the results acquired.
- To use SIFT descriptor developed in chapter 3 in some of computer vision application by matching the features from each descriptor.
- To improve the speed of SIFT algorithm by reducing hardware resources needed. This can be achieved by optimizing the VHDL using different tool.
- To compare between MATLAB and the newly trending language (OpenCL) and their implementations of image processing and computer vision in FPGA.

References

[1] R. C. Gonzalez and R. E.Woods, *Digital image processing*, 3rd Editio. prentice Hall, 2008.

[2] D. G. Bailey, *Design for Embedded Image Processing on FPGAs*. 2011.

[3] R. Szeliski, “Computer Vision : Algorithms and Applications,” *Computer (Long. Beach. Calif).*, vol. 5, p. 832, 2010.

[4] C. Maxfield, *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows*, vol. 1. 2004.

[5] K. Kintali and Yongfeng Gu, “Model-Based Design with Simulink , HDL Coder , and Xilinx System Generator for DSP,” *MathWorks, Inc*, pp. 1–15, 2012.

[6] C. Eckles, “Live Video Edge Detection in Simulink,” 2010.

[7] K. N. Aresh and M. M. Ahender, “FPGA Based Implementation of Image Edge Detection using Canny Edge Detection Algorithm,” *Int. J. Sci. Engeneering Techology Res.*, vol. 3, no. 29, pp. 5840–5844, 2014.

[8] K. Mikolajczyk, T. Tuytelaars, C. Schmid, A. Zisserman, J. Matas, F. Schaffalitzky, T. Kadir, and L. Van Gool, “A comparison of affine region detectors,” *Int. J. Comput. Vis.*, vol. 65, no. 1–2, pp. 43–72, 2005.

[9] S. Keypoints and D. G. Lowe, “Distinctive Image Features from,” *Int. J. Comput. Vis.*, vol. 60, no. 2, pp. 91–110, 2004.

[10] J. Zhao, “Video / Image Processing on FPGA,” 2015.

[11] C. Liu, J. Yuen, and A. Torralba, “Sift flow: Dense correspondence across scenes and its applications,” *Dense Image Corresp. Comput. Vis.*, pp. 15–49, 2015.

[12] Bill McKeeman, “MATLAB Performance Measurement,” *MathWorks, Inc*, 2016.

Appendix A: Samples of Output Results

A.1 Edge detection

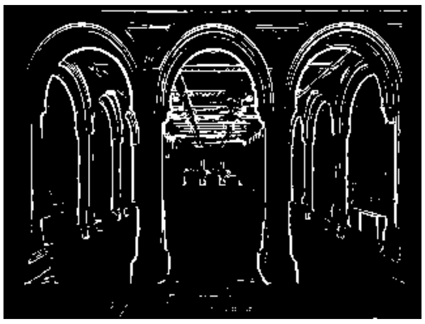


Figure A.1: Output image

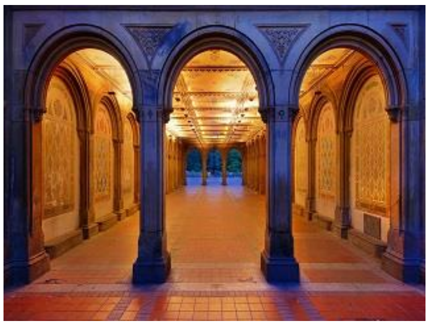


Figure A.2: Original image

A.2 Median filter

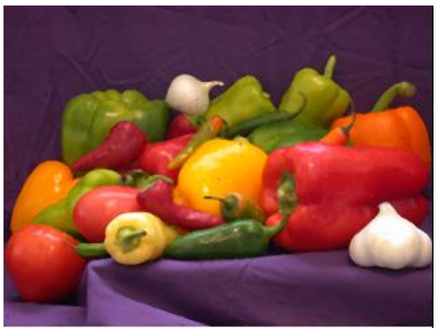


Figure A.3: Output image

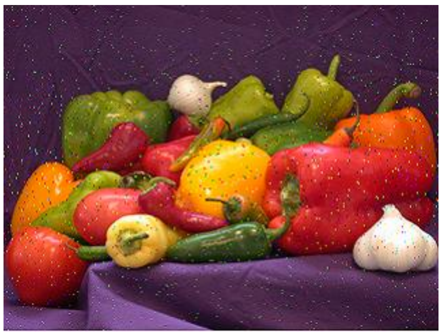


Figure A.4: Original image

A.3 Gamma correction

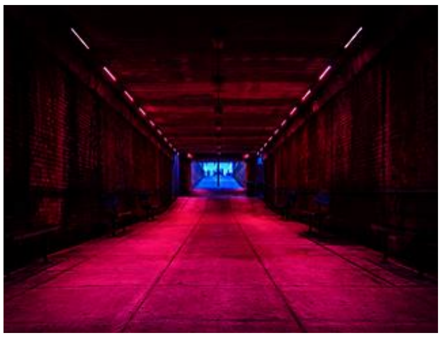


Figure A.5: Output image Degamma

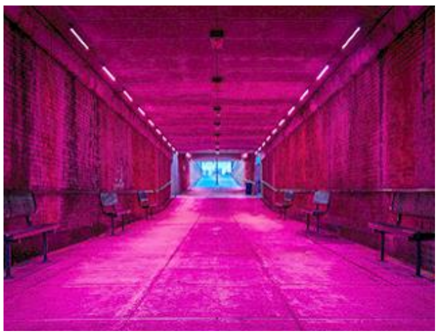


Figure A.6: Output image gamma

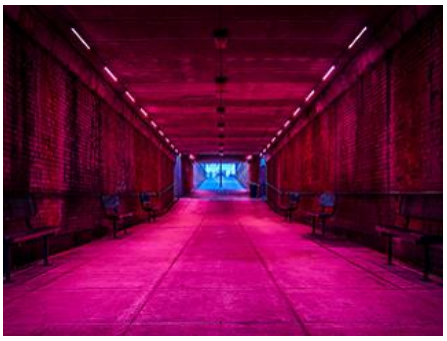


Figure A.7: Original image

A.4 Color correction

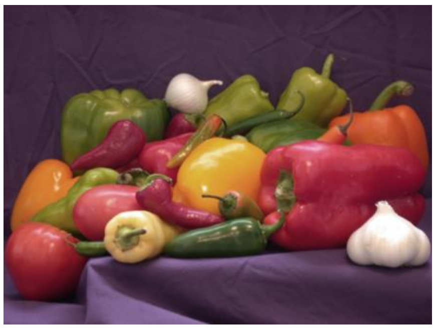


Figure A.8: Output image

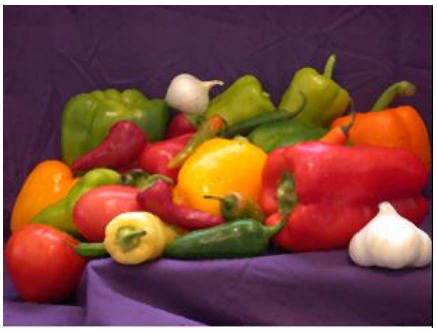


Figure A.9: Original image

A.5 2-D filter

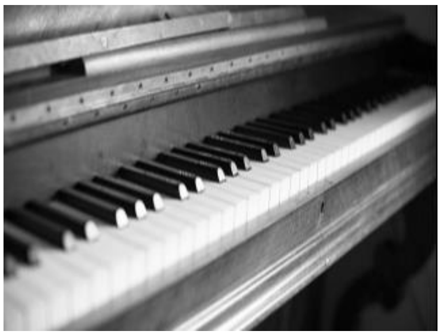


Figure A.10: Output image

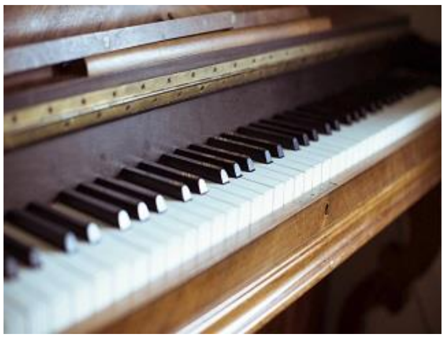


Figure A.11: Original image

Appendix B: SIFT getFeature Model:

Inside getFeature

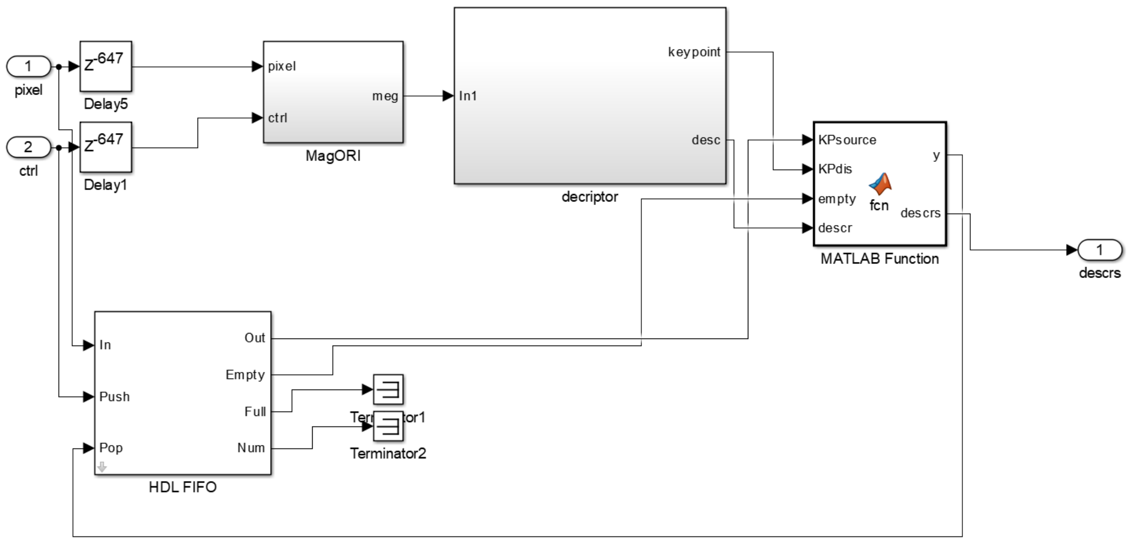


Figure B.1: *getFeature model*

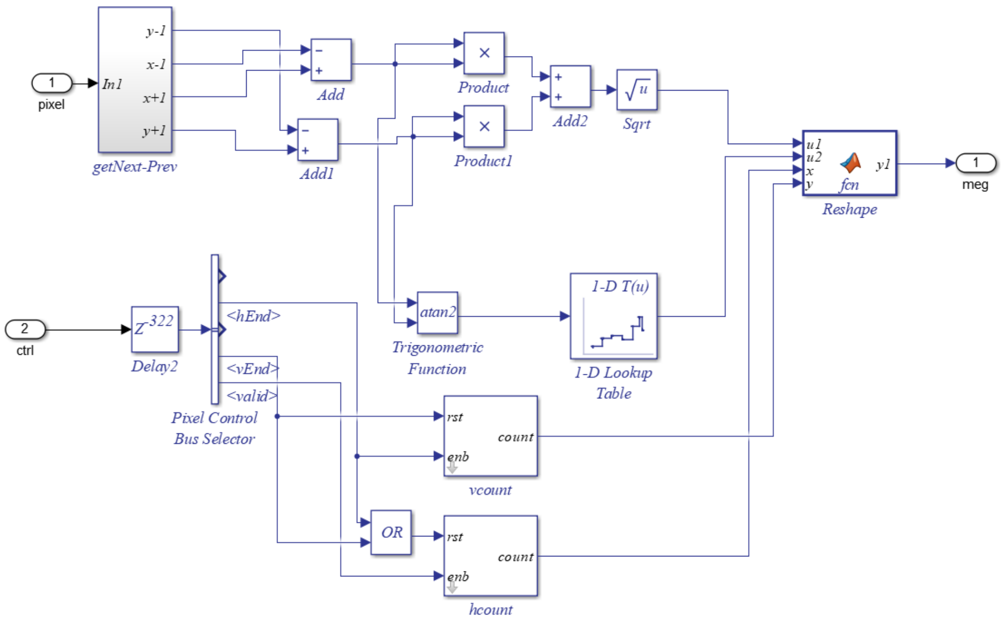


Figure B.2: *MagOri model calculate magnitude and orientation*

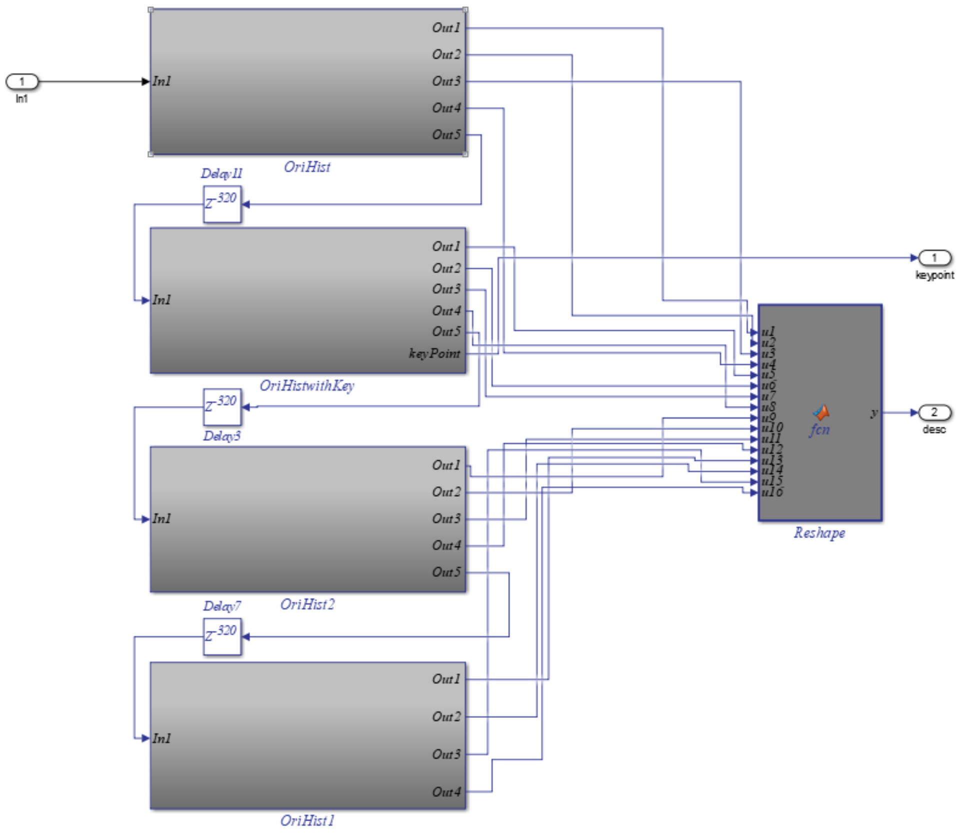


Figure B.3: *descriptor*

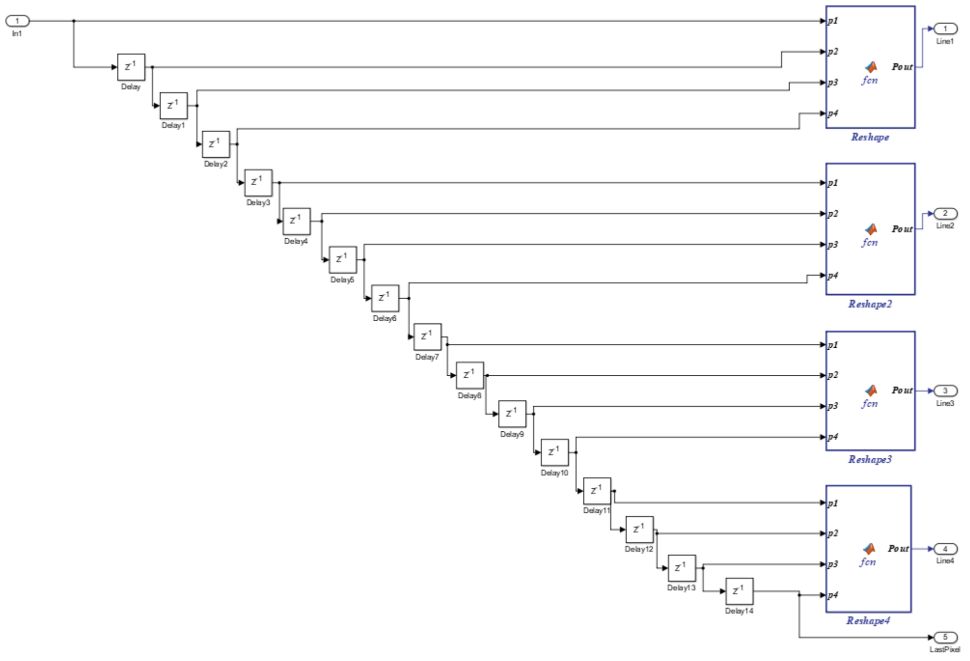


Figure B.4: *OriHist subsystem (orientation histogram)*