**HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY AND EDUCATION**

**FACULTY FOR HIGH-QUALITY TRAINING**

🙞🙡🕮🙣🙜



**GRADUATION PROJECT**

**IMPLEMENTATION OF SOBEL FILTER**

**BASED ON THE ZYNQ-7000 SOC DEVELOPMENT BOARD**

**NGUYEN HOANG NHAT UYEN**

**Student ID: 18142241**

**Major: ELECTRONICS AND TELECOMMUNICATIONS**

**ENGINEERING TECHNOLOGY**

**Advisor: TRUONG QUANG PHUC, MSc.**

Ho Chi Minh City, July 2022

|  |  |
| --- | --- |
|  | THE SOCIALIST REPUBLIC OF VIETNAM  **Independence – Freedom – Happiness**  ----\*\*\*----  *Ho Chi Minh City, mm dd, 2022* |

# **GRADUATION PROJECT ASSIGNMENT**

Student name: Nguyen Hoang Nhat Uyen Student ID: 18142241

Major: Electronics and Telecommunications Engineering Technology

Class: 18161CLCVT2A

Advisor: Truong Quang Phuc, MSc.Phone number: 0917731988

Date of assignment: 07/04/2022 Date of submission: dd/mm/2022

1. Project title: Implementation of Sobel Filter based on the ZYNQ-7000 SoC Development Board.

2. Initial materials provided by the advisor: Sobel Edge-based Image Template Matching in FPGA, ZYNQ All Programmable SoC Sobel Filter Implementation Using the Vivado HLS Tool.

3. Contents of the project:

- Research for image processing techniques, particularly edge detection filters such as Prewitt, and Sobel,…

- Execute Sobel filter simulation and package IP core Sobel in Vivado HLS.

- Create Sobel filter block design in Vivado.

- Research how to connect and implement the Sobel filter on ZYNQ.

- Valuate result indicators.

- Write the final report.

4. Final product: a Sobel Filter model on the ZYNQ platform.

**CHAIR OF THE PROGRAM ADVISOR**

|  |  |
| --- | --- |
| Shape  Description automatically generated with medium confidence | THE SOCIALIST REPUBLIC OF VIETNAM  **Independence – Freedom – Happiness**  ----\*\*\*----  *Ho Chi Minh City, mm dd, 2022* |

# **ADVISOR’S EVALUATION SHEET**

Student name: Nguyen Hoang Nhat Uyen Student ID: 18142241

Major: Electronics and Telecommunications Engineering Technology

Class: 18161CLCVT2A

Project title: Implementation of Sobel Filter based on the ZYNQ-7000 SoC Development Board.

Advisor: Truong Quang Phuc, MSc.

**EVALUATION**

1. Contents of the project:

2. Strengths:

3. Weaknesses:

4. Approval for oral defense?

5. Overall evaluation:

6. Mark:.........................(*in words:*.......................................................................................)

|  |  |  |
| --- | --- | --- |
|  | *Ho Chi Minh City, mm dd 2022*  **ADVISOR** |  |

|  |  |
| --- | --- |
| Shape  Description automatically generated with medium confidence | THE SOCIALIST REPUBLIC OF VIETNAM  **Independence – Freedom – Happiness**  ----\*\*\*----  *Ho Chi Minh City, June 27, 2022* |

# **PRE-DEFENSE EVALUATION SHEET**

Student name: Nguyen Hoang Nhat Uyen Student ID: 18142241

Major: Electronics and Telecommunications Engineering Technology

Class: 18161CLCVT2A

Project title: Implementation of Sobel Filter based on the ZYNQ-7000 SoC Development Board.

Name of Reviewer:

**EVALUATION**

1. Contents and workloads of the project:

2. Strengths:

3. Weaknesses:

4. Approval for oral defense?

5. Overall evaluation:

6. Mark:........................(*in words:* .............................................................................)

|  |  |
| --- | --- |
|  | *Ho Chi Minh City, mm dd 2022*  **REVIEWER** |

|  |  |
| --- | --- |
| Shape  Description automatically generated with medium confidence | THE SOCIALIST REPUBLIC OF VIETNAM  **Independence – Freedom – Happiness**  ----\*\*\*----  *Ho Chi Minh City, June 27, 2022* |

# **EVALUATION SHEET OF DEFENSE COMMITTEE MEMBER**

Student name: Nguyen Hoang Nhat Uyen Student ID: 18142241

Major: Electronics and Telecommunications Engineering Technology

Class: 18161CLCVT2A

Project title: Implementation of Sobel Filter based on the ZYNQ-7000 SoC Development Board.

Name of Defense Committee Member:

**EVALUATION**

1. Contents and workloads of the project:

2. Strengths:

3. Weaknesses:

4. Approval for oral defense?

5. Overall evaluation:

6. Mark:........................(*in words:* .............................................................................)

|  |  |
| --- | --- |
|  | *Ho Chi Minh City, mm dd, 2022*  **COMMITTEE MEMBER** |

# **ACKNOWLEDGEMENTS**

I would like to express my appreciation to my advisor Truong Quang Phuc for his invaluable patience. Throughout this project, he always helps and spends time giving me useful guidance. I believe that I could not have undertaken everything without my advisor.

I am also deeply indebted to my teachers Nguyen Ngo Lam and Tran Tung Giang for their advice and encourage me to carry out this project.

Moreover, many thanks to my friends, who are willing to support me anytime.

**CHAIR OF THE PROGRAM**

# **Table of Contents**

[**GRADUATION PROJECT ASSIGNMENT** i](#_Toc109159613)

[**ADVISOR’S EVALUATION SHEET** ii](#_Toc109159614)

[**PRE-DEFENSE EVALUATION SHEET** iii](#_Toc109159615)

[**EVALUATION SHEET OF DEFENSE COMMITTEE MEMBER** iv](#_Toc109159616)

[**ACKNOWLEDGEMENTS** v](#_Toc109159617)

[**Table of Contents** vi](#_Toc109159618)

[**List of Acronyms** ix](#_Toc109159619)

[**List of Tables** xi](#_Toc109159620)

[**List of Figures** xii](#_Toc109159621)

[**Abstract** xiv](#_Toc109159622)

[**Chapter 1: INTRODUCTION** 1](#_Toc109159623)

[**1.1 Overview** 1](#_Toc109159624)

[**1.2 Related work** 2](#_Toc109159625)

[**1.3 Objective** 2](#_Toc109159626)

[**1.4 Work content** 3](#_Toc109159627)

[**1.5 Outline** 3](#_Toc109159628)

[**Chapter 2 LITERATURE REVIEW** 5](#_Toc109159629)

[**2.1 An overview of image** 5](#_Toc109159630)

[**2.1.1 Image definition** 5](#_Toc109159631)

[**2.1.2 Color space** 5](#_Toc109159632)

[**2.2 Image processing** 6](#_Toc109159633)

[**2.2.1 Introduction** 6](#_Toc109159634)

[**2.2.2 Conversion of image format** 8](#_Toc109159635)

[**2.2.3 Convolution operation** 9](#_Toc109159636)

[**2.2.4 Binarization** 10](#_Toc109159637)

[**2.2.5 Edge detection algorithm** 11](#_Toc109159638)

[**2.2.5.1 Background** 11](#_Toc109159639)

[**2.2.5.2 Sobel filter** 12](#_Toc109159640)

[**2.2.6 Comparison parameter** 13](#_Toc109159641)

[**2.2.6.1 Mean Square Error** 13](#_Toc109159642)

[**2.2.6.2 Peak Signal to Noise Ratio** 13](#_Toc109159643)

[**2.3 Developing toolchain** 14](#_Toc109159644)

[**2.3.1 FPGA introduction** 14](#_Toc109159645)

[**2.3.2 IP Core** 15](#_Toc109159646)

[**2.3.3 Vivado and High-Level Synthesis** 16](#_Toc109159647)

[2.3.3.1 An introduction to High-Level Synthesis 16](#_Toc109159648)

[2.3.3.2 Vivado application tool 16](#_Toc109159649)

[2.3.3.3 Software Development Kit 17](#_Toc109159650)

[2.3.3.4 Xilinx HLS Video Library 17](#_Toc109159651)

[2.3.3.5 Open source library for image processing 17](#_Toc109159652)

[**2.3.4 Dataflow technique** 18](#_Toc109159653)

[**2.3.5 Python tool** 18](#_Toc109159654)

[**2.4 ZYNQ-7000 platform** 19](#_Toc109159655)

[**2.4.1 Introduction to ZYNQ board** 19](#_Toc109159656)

[**2.4.2 Advanced Extensible Interface protocol** 21](#_Toc109159657)

[**Chapter 3 SYSTEM DESIGN** 25](#_Toc109159658)

[**3.1 Implementation of Sobel Filter based on software application** 25](#_Toc109159659)

[**3.2 Implementation of Sobel Filter based on co-design platform** 25](#_Toc109159660)

[**3.2.1 System requirement specification** 25](#_Toc109159661)

[**3.2.2 System design process** 25](#_Toc109159662)

[3.2.2.1 Flowchart of the edge detection algorithm 27](#_Toc109159663)

[3.2.2.2 Overview of PL and PS parts co-operation 27](#_Toc109159664)

[3.2.2.3 Flowchart over the PL part of IP core Sobel 29](#_Toc109159665)

[3.2.2.4 Flowchart of the IP core Sobel in Test Bench 29](#_Toc109159666)

[**3.2.3 The PL part configuration** 30](#_Toc109159667)

[3.2.3.1 Pre-processing block design 31](#_Toc109159668)

[3.2.3.2 Central processing block design 40](#_Toc109159669)

[**3.3 Execution of Sobel system** 41](#_Toc109159670)

[**Chapter 4 RESULT** 43](#_Toc109159671)

[**4.1 Block diagram of Sobel system** 43](#_Toc109159672)

[**4.2 Measurements of Sobel system based on HLS implementation** 43](#_Toc109159673)

[**4.2.1 Power consumption** 43](#_Toc109159674)

[**4.2.2 Throughput value and Hardware utilization** 44](#_Toc109159675)

[**4.3 Output result and comparison** 46](#_Toc109159676)

[**Chapter 5 CONCLUSION AND FUTURE WORK** 48](#_Toc109159677)

[**5.1 Conclusion** 48](#_Toc109159678)

[**5.2 Future work** 48](#_Toc109159679)

[**REFERENCES** 49](#_Toc109159680)

[**APPENDICES** 52](#_Toc109159681)

# **List of Acronyms**

|  |  |
| --- | --- |
| **AI** | Artificial Intelligence |
| **AMBA** | Advanced Microcontroller Bus Architecture |
| **EOL** | End Of Line |
| **SOF** | Start Of Frame |
| **MMCM** | Mixed-mode Clock Manager |
| **CLB** | Configurable Logic Blocks |
| **PSNR** | Peak Signal to Noise Ratio |
| **MSE** | Mean Square Error |
| **RTL** | Register-transfer level |
| **HLS** | High-Level Synthesis |
| **ADC** | Analog-to-Digital Converter |
| **BRAM** | Block Random Access Memory |
| **I2C** | Inter-Integrated Circuit |
| **SD** | Secure Digital |
| **PAL** | Phase Alternating Line |
| **DMA** | Direct Memory Access |
| **HDMI** | High-Definition Multimedia Interface |
| **VGA** | Video Graphics Array |
| **SoC** | System On Chip |
| **AXI** | Advanced Extensible Interface |
| **PS** | Processing System |
| **PL** | Programmable Logic |
| **OpenCV** | Open Computer Vision |
| **PLL** | Phase-Locked Loop |
| **DVI** | Digital Visual Interface |
| **IDLE** | Integrated Development and Learning Environment |

# **List of Tables**

**Tables in chapter 3**

[**Table 3. 1** Port descriptions of IP DVI to RGB 33](#_Toc109159682)

[**Table 3. 2** Port descriptions of IP Clocking Wizard 34](#_Toc109159683)

[**Table 3. 3** Port descriptions of IP Constant 35](#_Toc109159684)

[**Table 3. 4** Port descriptions of IP Video In to AXI4-Stream 37](#_Toc109159685)

[**Table 3. 5** Port descriptions of IP Video Timing Controller 39](#_Toc109159686)

[**Table 3. 6** Port descriptions of IP ZYNQ Processing 41](#_Toc109159687)

**Tables in chapter 4**

[**Table 4. 1** Result parameters comparison 51](#_Toc77446903)

# **List of Figures**

**Figures in chapter 2**

[**Figure 2. 1** Basis steps in an image processing process 7](#_Toc108723967)

[**Figure 2. 2** (a) 4-neighborhood, (b) 8-neighborhood, and (c) cross-neighborhood 8](#_Toc108723968)

[**Figure 2. 3** (a) color image and (b) grayscale image 9](#_Toc108723969)

[**Figure 2. 4** Thresholding 10](#_Toc108723970)

[**Figure 2. 5** (a) grayscale image and (b) binary image 11](#_Toc108723971)

[**Figure 2. 6** (a) Ideal boundary, (b) Ramp boundary, and (c) Real boundary 12](#_Toc108723972)

[**Figure 2. 7** (a) without dataflow pipelining and (b) with dataflow pipelining 18](#_Toc108723973)

[**Figure 2. 8** Overview of ZYNQ AP SoC Architecture 19](#_Toc108723974)

[**Figure 2. 9** APU’s block diagram 20](#_Toc108723975)

[**Figure 2. 10** CLB’s structure 21](#_Toc108723976)

[**Figure 2. 11** Read data period of AXI4 protocol 22](#_Toc108723977)

[**Figure 2. 12** Write data period of AXI4 protocol 23](#_Toc108723978)

[**Figure 2. 13** AXI4-Stream data transferring 23](#_Toc108723979)

[**Figure 2. 14** Pulse chart of AXI4-Stream data transmission period 24](#_Toc108723980)

**Figures in chapter 3**

[**Figure 3. 1** Flowchart over the Sobel Filter on OpenCV - Python 25](#_Toc109159688)

[**Figure 3. 2** Design process using Vivado tools 26](#_Toc109159689)

[**Figure 3. 3** Flowchart over the entire edge detection system 27](#_Toc109159690)

[**Figure 3. 4** Flowchart illustration for the overview of the PL and PS parts co-operation 28](#_Toc109159691)

[**Figure 3. 5** Hardware illustration for the overview of the PL and PS parts cooperation 28](#_Toc109159692)

[**Figure 3. 6** Flowchart over the PL part of IP core Sobel 29](#_Toc109159693)

[**Figure 3. 7** Flowchart of the IP core Sobel in HLS Test Bench 30](#_Toc109159694)

[**Figure 3. 8** Edge detection system’s block diagram 30](#_Toc109159695)

[**Figure 3. 9** DVI to RGB converter block diagram 31](#_Toc109159696)

[**Figure 3. 10** IP DVI to RGB top-level diagram 32](#_Toc109159697)

[**Figure 3. 11** Port connections of IP DVI to RGB in Vivado 33](#_Toc109159698)

[**Figure 3. 12** IP Clocking Wizard Block Diagram 33](#_Toc109159699)

[**Figure 3. 13** Port connections of IP Clocking Wizard in Vivado 34](#_Toc109159700)

[**Figure 3. 14** IP Constant Block Diagram 34](#_Toc109159701)

[**Figure 3. 15** Port connections of IP Constant in Vivado 35](#_Toc109159702)

[**Figure 3. 16** IP Video In to AXI4-Stream with Video Timing Controller Block Diagram 36](#_Toc109159703)

[**Figure 3. 17** IP Video In to AXI4-Stream top-level diagram 36](#_Toc109159704)

[**Figure 3. 18** Port connections of IP Video In to AXI4-Stream 38](#_Toc109159705)

[**Figure 3. 19** Port connections of IP Video Timing Controller 39](#_Toc109159706)

[**Figure 3. 20** Input block design in Vivado 39](#_Toc109159707)

[**Figure 3. 21** Port connections of IP core Sobel in Vivado 40](#_Toc109159708)

[**Figure 3. 22** Port connections of IP ZYNQ in Vivado 41](#_Toc109159709)

[**Figure 3. 23** ZYBO clock diagram 42](#_Toc109159710)

[**Figure 3. 24** I/O Ports of Sobel system 42](#_Toc109159711)

**Figures in chapter 4**

[**Figure 4. 1** Entire block diagram of Sobel system 48](#_Toc77446887)

[**Figure 4. 2** (a) Power summary (b) Power On-chip 49](#_Toc77446888)

[**Figure 4. 3** Measurements of timing and latency in the synthesis report 49](#_Toc77446889)

[**Figure 4. 4** The estimated hardware utilization of IP core Sobel 50](#_Toc77446890)

[**Figure 4. 5** The RTL exportation of IP core Sobel 50](#_Toc77446891)

[**Figure 4. 6** The ordering of performing functions by HLS Performance 50](#_Toc77446892)

[**Figure 4. 7** Result of edge detection algorithm on ZynQ-7000 platform 51](#_Toc77446893)

[**Figure 4. 8** Result of edge detection algorithm on OpenCV – Python 51](#_Toc77446894)

# **Abstract**

Images are a vital part of the computer vision field due to the huge amount of valid data from them. As the most valuable piece of data in an image, the edges are not only specific to the overall object’s shape but also directly impact the point of view of a human or computer with that image.

This thesis introduced an image processing algorithm, specifically an edge detection operation called Sobel Filter. In order to give a coherent conclusion on why we should implement an image processing application on a co-design platform, the algorithm was applied on both software (OpenCV - Python) and hardware (ZYNQ Board), along with the support toolchain released by Xilinx – Vivado, and Vivado High-Level Synthesis (HLS). All result parameters show that the Sobel Filter executed on OpenCV - Python works effectively, but the Mean Square Error (MSE) of 15234.9 and Peak Signal to Noise Ratio (PSNR) of 6.3 are not satisfactory. Meanwhile, the Sobel Filter based on the ZYNQ co-design board proved that it provided a good output. The result indicators are estimated by Vivado Design Evaluation tools with an MSE of 2628.9, a PSNR of 13.93, a throughput of 78 frames/s, power consumption of 0.025 J/frame, and low hardware utilization.

In conclusion, this thesis shows that the Sobel Filter is suggested to be implemented on a co-design platform to provide an image processing system with high performance and low power usage. Nevertheless, the entire system still has several shortcomings and needs to be improved, a more effective algorithm and a deeper understanding of hardware are considered in this case.

# **Chapter 1: INTRODUCTION**

# **1.1 Overview**

Image processing is a concept that refers to the process of performing a specific operation on an image, to obtain an input-based features new dataset output. The major purpose of image processing is to capture characteristics of the image such as edge detection, and image segmentation, or improve image quality by enhancing brightness or increasing resolution. Presently, image processing is a technology that is developing rapidly, and thanks to it, we can get the desired image data to serve a specific purpose in life. This thesis proposed a method of image processing called edge detection. In an image, the change or continuity in the amplitude shows us information about the subject of the image, local discontinuity in an image from one brightness to another is named an edge, while global discontinuity is called a boundary [1]. Separating these boundaries from the background minimizes the useless background of the image, thereby, reducing the amount of data to be stored and increasing the performance for the next image processing steps.

The emergence of computer vision has facilitated the development of image processing. Applications of image processing are increasingly popular in areas such as telecommunications, automatic control, intelligent transportation, and biomedical engineering,...So far, many developers have only chosen software to implement image processing algorithms, that is the implementation of Gaussian Filter, Fourier transform, edge detection, and Wavelet equations through open libraries such as OpenCV, Scikit-image, Pillow, Numpy, Mahotas. Performing on software like this mainly helps us verify and simulate whether the algorithm in use is working efficiently or not. However, it has been proven that the implementation of software is not an effective approach for real-time image processing applications. The reason is that despite its flexibility, software coding is still faced with many problems due to sequential execution and serial operational memory; these factors take a long time to read/write image frames [2].

For that reason, the implementation of image processing algorithms is more effective with hardware, because it helps to parallel subprograms in the main program. Some hardware accelerators, for example, the field-programmable gate array (FPGAs) are a critical factor in reducing computational time and increasing processing speed. Which hardware platform is selected depends on the algorithm in use, there are numerous tools to implement code for sequential algorithms introduced by Xilinx such as Integrated Synthesis Environment (ISE), PlanAhead, and Software Development Kit (SDK),...In this thesis, I used the Vivado Design Suite which was released for 7-series FPGA families, consisting of Virtex-7, Kintex, Artix, ZYNQ, and HLS for C-based development optimized codes. The combination of coding on software and executing on hardware called FPGA co-design brings a powerful image processing system.

# **1.2 Related work**

This thesis aims to persuade people that the performance of image processing algorithms on co-design is more efficient than just executing on software. This persuasion has even been proved in previous image processing projects, particularly here we mention edge detection. Firstly in [3], the authors presented a License Plate Automatic Recognition model that uses a template matching algorithm, the Canny Filter is utilized for edge detection purposes, which helps to reduce the amount of data. The entire system was implemented on MATLAB and delivers an accuracy of 84.28% when tested with a total of 70 images. Moreover, the system can stably work in low resolution and contrast images or murky, sunny or rainy environments. Next, the authors of the project [4] proposed an automatic insect detection system aimed at helping farmers increase their yields. Because insects come in many different shapes such as circles, ovals, or rectangles,...so the authors used Sobel Filter to segment their images, this helps to identify insects' geometric shapes better. This system is implemented on MATLAB 2015b and delivers an unexpected result - successful in all cases.

In contrast, both two projects above are difficult to optimize the algorithm performance because of the sequential execution of software, this causes high latency and unstable accuracy for the system since it has to do many heavy tasks. Therefore, other image processing projects are implemented on hardware to observe the difference. In 2007, Abbasi presented a model for the Sobel filter on the FPGA platform [5]. Unfortunately, this project proved to be ineffective due to its low complexity and high execution time. After some time, Halder and his colleagues proposed a similar model but with a smaller architecture and faster execution time [6], but it still has shortcomings by over-storing pixels and wasting system resources.

As claimed by the advantages and disadvantages of the above mentioned, it has been found that the software platform can help build and improve image processing algorithms with hardware description language, specifically Verilog, VHDL, or C/C++ in HLS. The hardware platform can help execute the algorithms as optimally as possible, delivering high performance, low power consumption, and saving system resources. Due to that, new image processing projects recently have been considered for implementation on a platform that integrates both software and hardware, which delivers a powerful co-design system.

# **1.3 Objective**

The goal of this thesis is to implement an edge detection system based on Sobel Filter on a software platform and then implement it on a co-design platform. After that, take an observation of the results to conclude that an image processing algorithm should be implemented on a co-design. On the other hand, make an improvement proposal according to shortcomings.

# **1.4 Work content**

From the content of the objective, we can make a list of all the things to do below.

* Study on the image, image processing, Sobel algorithm, an overview of FPGA, HLS, IP, and support toolchains such as Vivado, Vivado HLS, and ZYNQ Board.
* Implement Sobel Filter on OpenCV - Python, summarize all the results: output image, values of MSE, and PSNR.
* Create IP core of Sobel Filter in C++ language using Vivado HLS and OpenCV libraries, simulate the test program to validate the Sobel operation.
* Generate block design of Sobel Filter on Vivado and program this system on ZYNQ Board. Summarize all the results: output image, values of Mean Square Error, Peak Signal to Noise Ratio, throughput, and power consumption.
* Evaluate and compare the result indicators on software and co-design platforms.
* Conclude and propose future work.

# **1.5 Outline**

This thesis includes 5 chapters.

***Chapter 1* INTRODUCTION**

This first chapter mentions an overview of the entire project, the related work, the target, tasks, and work content.

***Chapter 2* LITERATURE REVIEW**

Chapter 2 includes the foundational knowledge about image definition, image processing algorithms, FPGA and SoC ZYNQ-7000 platform, standard AXI protocols, and support toolchains.

***Chapter 3* SYSTEM DESIGN**

This chapter presents all steps for building systems, creating the IP core Sobel, and designing the block diagram as mentioned by manuals of Xilinx and Digilent on both software and hardware.

***Chapter 4* RESULT**

The output results on both software and hardware are shown in this chapter, also the resource indicators used in the project such as the number of logic gates, FIFO, LUT, BRAM,…When comparing the processing speed and PSNR between software and hardware.

***Chapter 5* CONCLUSION AND FUTURE WORK**

Eventually, this chapter presents the overall opinions about this project and proposes future work.

# **Chapter 2 LITERATURE REVIEW**

# **2.1 An overview of image**

## **2.1.1 Image definition**

An image is considered to be a visual representation of the external form of people, animals, or anything around us. In information technology, an image is described as a picture that has been stored in electronic form. Since the image appeared, it has become an essential part of life because it not only describes how we observe everything in the world but is also an effective way for communication and data storage. Image is identified by two definitions - image type and image format. There are several different types of images such as photographs, illustrations, drawings, graphics, and some standard formats of images such as JPEG, GIF, PNG, SVG, or TIFF.

Here is the concept of a pixel, this is defined as an element at the (*x, y*) coordinate of a digital image and has a specified grayscale (or color) value. Therefore, a digital image is considered to be a collection of a matrix containing many pixels. We had already known about digital image – a form to present images in most electronic devices such as phones, cameras, and computers,…Academically, a digital image is represented by a two-dimensional function *z* = f (*x, y*). Where *x* and *y* are spatial coordinates (plane), *z* represents the amplitude of f (*x, y*), as known as the intensity or gray level of the image at the point that contains (*x, y*) [7].

## **2.1.2 Color space**

The color space is a concept used as a mathematical model. It aims to describe actual colors when represented numerically. The commonly used color spaces in digital image processing are RGB and YUV. In addition, there are other color spaces such as CIE LAB, HSV, and CMYK [7].

**RGB color space:** RGB is a color space used in computers and many other digital devices. This color space is on the basis of three colors, denoted respectively by R (red), G (green), and B (blue), which is the foundation for representing all colors. In RGB color space, one pixel contains three parameters R-G-B. RGB values are in the 24 bpp (bits per pixel) model, specifically recorded as a pair of three parameters (r,g,b) whose value ranges from 0 to 255. With the 24-bit model, the maximum number of colors would be 256 x 256 x 256 = 16.777.216 colors.

**YUV color space:** This color space is specified by the YUV color model, which is represented by one luminance and two chrominance values. The YUV space is used in the PAL standard television broadcasting system; this is a common standard in most countries that helps us to create a compatible color range with the color range in human perception rather than the RGB (the type used in computer graphical devices). In YUV, the Y value is on behalf of the luminance component, the two U and V represent the color components. The weighted values of R, G, and B are the origin of Y, U, and V. Specifically, the three parameters R, G, and B are summed by a factor to produce a single signal Y to represent the overall luminance at that considered pixel. After then, U is calculated by removing Y from the blue (B) of RGB and multiplying it by a pre-existing scaling factor. Finally, the V signal is calculated by removing Y from the red (R) of RGB and also multiplied by other pre-existing scaling factors. The following formulas are used to determine Y, U, and V from R, G, and B [8]:

|  |  |
| --- | --- |
|  | (2.1) |
|  | (2.2) |
|  | (2.3) |

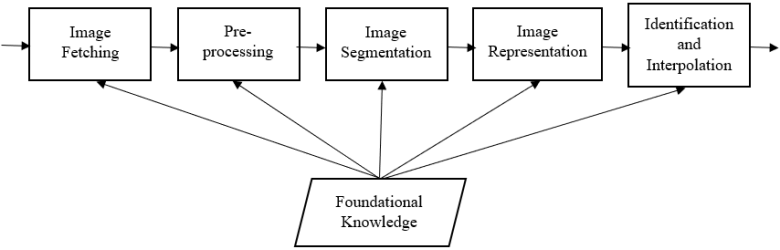
As stated by the value that is used to represent pixels, we commonly distinguish three main types of images.

* **Color image:** each pixel has three values (r,g,b), which include red, green, and blue. Each of these three colors can be in a variety of brightness levels, which has a value that ranges from 0 to 255.
* **Gray image:** each pixel in this matrix has a chroma that varies from white to black and ranges from 0 to 255.
* **Binary image:** this kind of image contains only two levels of black or white light, so, the pixel value is only can be 0 or 1. Despite that, when processing an image on the computer a grayscale image to represent the binary image is used, and now that two value is 0 or 255.

# **2.2 Image processing**

## **2.2.1 Introduction**

As we have discussed, image processing is considered a significant part of computer vision, which performs a mathematical operation on the input image and provides the desired output related to the system's demands. Figure 2.1 below shows the basic steps of image processing [9].



**Figure 2. 1** Basis steps in an image processing process

**Step 1:** *Image Fetching*. Images are collected by color (or black and white) cameras that integrate an image sensor. The output image in this step is analog.

**Step 2:** *Pre-processing*. After step 1, the output image may have noise or low contrast due to the quality of the sensor and environmental conditions. We need this step to improve the image quality. The main functions of pre-processing are to filter out the noise and increase or decrease the contrast.

**Step 3:** *Image segmentation*. This step helps separate input images into different regions for image representation, analysis, and recognition.

**Step 4:** *Image representation*. All the regions after the segmentation step are sequences of pixels, so we need this step because we are interested in the internal characteristics of the image, such as the curve and shapes. In brief, the image representation process means transforming available image data into an appropriate and necessary format for processing by computer.

**Step 5:** *Identification and interpolation*. The following process of this step does object classification based on the descriptive details for that object by comparing the image with the pre-stored template.

Also in the above steps, we can see that the processes listed in the image processing steps must be performed under supervision as well as goes concerning the foundational knowledge of image processing.

To perform an algorithm on an image, we have to execute the operation not only on one pixel but also on the pixels around it. Therefore, there is a concept of the pixels located in the “neighborhood” area of a specified pixel. Figure 2.2 shows us three basic types of neighboring pixels: 4-neighborhood, 8-neighborhood, and cross-neighborhood [1].

|  |  |  |
| --- | --- | --- |
|  |  |  |
| (a) | (b) | (c) |

**Figure 2. 2** (a) 4-neighborhood, (b) 8-neighborhood, and (c) cross-neighborhood

## **2.2.2 Conversion of image format**

Among the transformation methods in the image processing field, the most important one is the transformation between three types of images: color, binary and multi-grayscale images. The purpose of this method is to have the desired image format that is compatible with the specific application. In this thesis, I use the grayscale image (or binary image) as input for the system to reduce the amount of data to be processed.

Here is the concept of monochromatic images, this sounds like a new concept, but we can know it under the name grayscale image. If the image is represented by 8 gray-level values, a pixel has the value belonging to [0-7], but if the image is represented by 256 gray-level values, then a pixel has the value in [0-255]. There are a few common methods to convert RGB images to grayscale images such as *mean* and *weight*.

*The mean method* takes the mean value of R, G, and B as the grayscale value [8].

|  |  |
| --- | --- |
|  | (2.4) |

This method is simple to implement, but it does not work as well as expected. The reason is human eyeballs have different arousal to three main colors red, blue, and green (RGB) so these three colors need to be distributed with different weights, which results in the appearance of the weight method.

*The weight method* uses gamma correction, more specifically, the distribution of red, blue, and green colors according to their wavelengths. There are different conversion standards but the ITU-R BT.601 standard is chosen, which has the following weights:

|  |  |
| --- | --- |
|  | (2.5) |

This weight distribution is characterized by the sensitivity to the color of the cone-cells in human eyes, which is most sensitive to green, then red, and the least sensitive is blue. Therefore, at one RGB pixel, we have this formula to calculate the light intensity of a gray image [10]:

|  |  |
| --- | --- |
|  | (2.6) |

We can also see that these coefficients satisfy with 0.299+0.587+0.114=1. Figure 2.3 is an example of RGB to grayscale image conversion.

|  |  |
| --- | --- |
|  |  |
| (a) | (b) |

**Figure 2. 3** (a) color image and (b) grayscale image

# **2.2.3 Convolution operation**

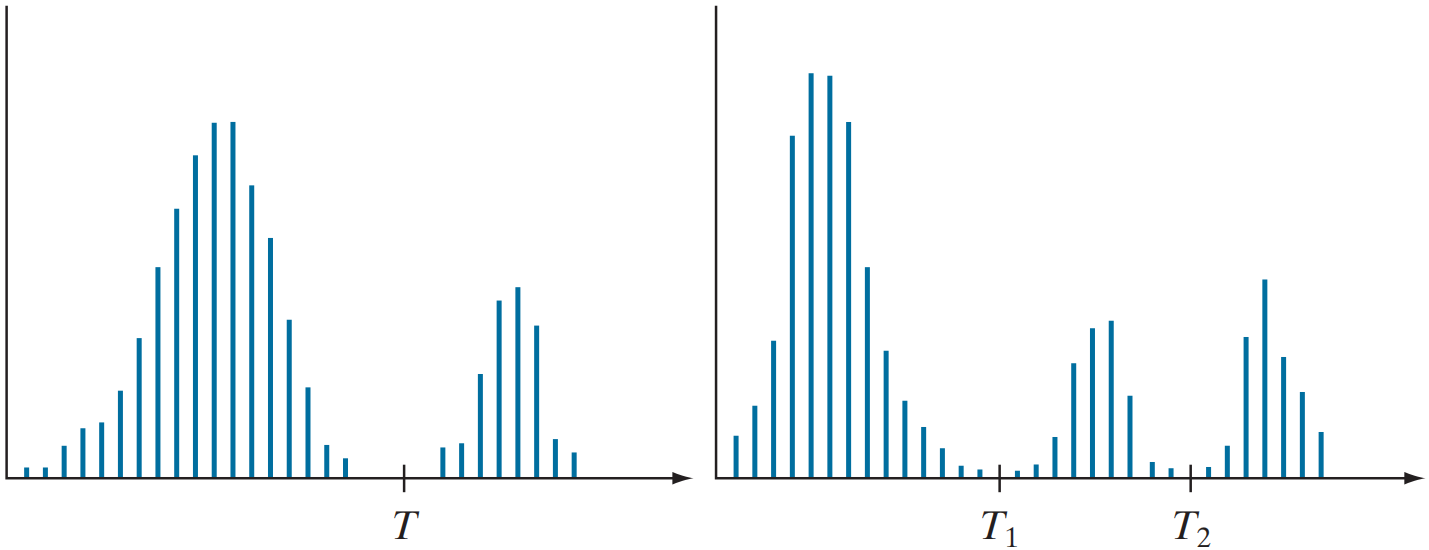
To perform a filter on the image, we need to implement a convolution between the original image matrix and the filter matrix. This useful assistance makes convolution become an important technique in image processing and often be used in image applications such as calculating image derivatives, image smoothing, and edge detection. In mathematics, convolution is a linear operation that results in a function by calculating between two existing functions *f* and *g*. For example, the convolution between the filter matrix and the original image matrix results in a denoised (blurred) image. Suppose that we have an image function f(*x,y*) and a filter k(*x,y*), so the formula for its convolution is given by [10]:

|  |  |
| --- | --- |
|  | (2.6) |

In convolution, the indispensable component is the filter (the kernel matrix). In a kernel matrix, the anchor point is responsible for determining the corresponding matrix area to perform the convolution, the anchor point is usually chosen as the center point of the kernel matrix. Each element on the kernel matrix has a value that is understood as an association coefficient with each pixel’s gray level value in turn (we mention the image area that corresponded to the kernel). The convolution operation takes place by shifting the kernel matrix, so that the matrix passes through all pixels in the original image, starting from the upper left corner of that image. Then the operation places the corresponding anchor point at the considered pixel. Each time the kernel matrix moves, each time it performs a calculation for that current pixel and extracts a new result to replace the old value.

# **2.2.4 Binarization**

Suppose that we have an image that consists of bright-spot objects on a dark-spot background, and these gray levels are represented on an intensity histogram of an f(x, y) function as shown in Figure 2.4, to extract objects from the background, we need to choose a threshold with the value of T. It means for each T we have a different binary image.



**Figure 2. 4** Thresholding

Now, any (*x, y*) point in the image which has a value f (*x, y*) > T is marked as a bright-spot object, otherwise, they are dark-spot backgrounds. In other words, a segmented image g (*x, y*) is given by [7]:

|  |  |
| --- | --- |
|  | (2.7) |

In summary, applying this thresholding method results in a binary image from a grayscale image. Figure 2.5 is an example.

|  |  |
| --- | --- |
|  |  |
| (a) | (b) |

**Figure 2. 5** (a) grayscale image and (b) binary image

# **2.2.5 Edge detection algorithm**

## **2.2.5.1 Background**

A pixel is considered an edge point when there appears a sudden change in gray level or color. In the case of binary images, a point that has value belongs to black is called an edge if its neighborhood has at least one while-point. A boundary is a set of consecutive edge points that form a contour. Boundaries are a typical characteristic when analyzing and identifying an image. Figure 2.6 shows us three main types of boundary [1]:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| (a) | (b) | (c) |

**Figure 2. 6** (a) Ideal boundary, (b) Ramp boundary, and (c) Real boundary

In image processing, a boundary is considered to be a typical characteristic for analysis and recognition. The purpose of using boundaries is to separate distinct gray areas. In other cases, image regions are used to find out the separated line. To talk about the importance of boundaries, we consider this example: when an artist draws portraits of people, they only need to outline the face instead of every detail as in reality.

Therefore, detecting or extracting the edge of an image is a method of finding the appearance of the edge. The goal of edge detection is to filter out the main information of the object’s shape in an image, thereby reducing the number of redundant pixels in memory. In brief, edge detection has become an important algorithm in image processing, there are various edge detection operations such as Roberts, Prewitt, Sobel, and Canny. All of these techniques have the same goal, but they work differently. In this thesis, I use the Sobel Filter, which is not too complicated to be executed but eliminates the noise well.

## **2.2.5.2 Sobel filter**

The Sobel operator performs a 2-D spatial gradient measurement on an image and helps to emphasize high spatial frequency regions – corresponding to the edge of the image. Sobel is usually used to find the approximate absolute magnitude of the gradient at each point in the input grayscale image. This operator uses a 3x3 kernel matrix and convolutes it with the original image to compute approximate values of derivate – this value represents the degree of horizontal and vertical grayscale change. For example, if we assume that A is the source image, and are two matrixes that respectively contain approximated horizontal and vertical derivate values at each pixel, it is expressed as below [11]:

|  |  |
| --- | --- |
|  | (2.8) |
|  | (2.9) |

After calculating the approximate derivate at each point in the source image, we use these obtained gradient values to calculate the total gradient magnitude of the corresponding pixel by this equation:

|  |  |
| --- | --- |
|  | (2.10) |

We can also calculate the direction of that gradient through its deflection angle:

|  |  |
| --- | --- |
|  | (2.11) |

Finally, we need to reduce the complexity of the result by using the approximate absolute value of that pixel’s gradient:

|  |  |
| --- | --- |
|  | (2.12) |

# **2.2.6 Comparison parameter**

To demonstrate that an edge detection algorithm works well, there are two parameters to evaluate the performance.

## **2.2.6.1 Mean Square Error**

The first parameter is the Mean Square Error (MSE). MSE is calculated by the average of the square of the error between the original image and the processed image. It means the lower the MSE value is, the more effective the algorithm is. The equation below shows how MSE is determined [12].

|  |  |
| --- | --- |
|  | (2.13) |

Where and respectively are the number of rows and columns of the matrix of the input image, and are the actual pixel values of original and processed image.

## **2.2.6.2 Peak Signal to Noise Ratio**

The second parameter is the Peak Signal to Noise Ratio (PSNR). PNSR is measured by the decimal scale of the ratio between the maximum power of the image and the Mean Square Error. It means the higher the PSNR is, the more quality the output image is. The equation below shows how PSNR is determined [12].

|  |  |
| --- | --- |
|  | (2.14) |

Where MAX is dedicated to the maximum pixel value. For example, if the size of the image is 8-bit, then MAX is 255.

# **2.3 Developing toolchain**

An overview of FPGA is aimed to be given in this section, together with the components of an image processing system - the IP cores. We also consider giving an introduction to the Vivado HLS support tool for creating soft IP cores, and some important HLS pragmas which help to optimize the algorithm. Last but not least, the Python tool is introduced, which is known as the software environment to implement Sobel Filter.

## **2.3.1 FPGA introduction**

Field-Programmable Gate Array (FPGA) is a term for microchips that use an array of programmable logic blocks. The history of the formation and development of FPGA is over 35 years since it has been first launched by Ross Freeman, who was also the founder of Xilinx in 1984 [13]. FPGA has the capability to hold from a hundred thousand to several billion logic gates and this helps to maximize the programming functionality. Furthermore, FPGA is also as known as a chip that allows users to reconfigure them due to any specific functions. We have to admit that as a sub-sector of Application Specific Integrated Circuit (ASIC) technology, FPGA is inferior in optimizing applicable designs and is limited in performing complex tasks. Specifically, FPGA needs a hardware area about 30 times more than ASIC, 3 times slower in execution time, and 10 times more in power consumption [14]. Despite those drawbacks, FPGA is superior because it saves on manufactured costs and is able to be reconfigured when used, which means even after the silicon fabrication, we can still apply any functionality on the FPGA chip. This makes sense that FPGA is suitable for more research and development purposes than ASIC. Last but not least, the FPGA is capable of executing all operations in parallel, that is exactly what we desire because this thesis aims to demonstrate the strong power of a co-design system. In most cases, FPGA chips are configured by Hardware Description Language (HDL) such as Very High Description Language (VHDL) or Verilog, but thanks to the HLS tool, we can create soft IP cores in C/C++ or System C. The next sections discuss more topics about HLS and IP cores.

To discuss ZYNQ Board in section 2.4, we mention some of the fundamental components of FPGA, which are the main factors used in the Xilinx 7 series FPGAs.

* **DSP48E1:** The first component is Digital Signal Processors (DSPs) slices, its blocks are called DSP48E1 and are designed to support the system doing some arithmetic operations like multiplication, addition, counter, or even logic operations such as NOT, AND, and OR. Besides, we can cascade multiple many DSP48E1 slices when implementing complex functions. The performance of DSP slices is an important bottleneck in real-time systems [15].
* **BRAM:** Secondly, the Block Random Access Memory (BRAMs) is significantly used when the FPGA design has a large amount of data. Its width and depth are customizable as 4/8/16/32 kb (kilobits) so that we can apply BRAM in lots of different kinds of applications. In fact, the more expensive and bigger the area the FPGA is, the more available the BRAM is on it [2].
* **LUT:** There is no doubt that there are a huge amount of data needed to be processed in an FPGA design, so FPGA must have any component to store and define all the value of inputs and output. Hence, instead of using numerous different logic gates to create the logic table, we can use just only a Look-up Table (LUTs), which is seen to be the same with a small RAM [16].
* **Flip-flops:** As we have known before, a flip-flop is a binary register to hold the logical state between clock cycles, it can only have two states - 0 or 1. In FPGA, the flip-flip is used for synchronous or asynchronous storage data. Each slice of the Xilinx 7 series has 8 flip-flops, 4 for synchronous and 4 for asynchronous. When the 4 synchronous is configured, the 4 others are unusable and vice versa [17].

## **2.3.2 IP Core**

As a rule, the more modern an FPGA application is, the more complex the system design is. Fortunately, it still has many standard blocks on the whole circuit board. We can realize that there is an opportunity to reduce the development time when implementing an FPGA system by using the Intellectual Property (IP) core [2]. The IP core is a block that contains a layout design to perform almost fundamental operations, which is reusable so that any vendor can use it as a building block in their chip design. As we can see, there are lots of IP cores built depending on standardized protocol components such as Ethernet, SPI, DRAM controller, PLLs, and DDR core.

IP cores are separated into three categories: hard, firm, and soft. Hard IP cores are not able to be customized for different technologies since it is designed as a completed silicon layout, but it provides strong power and fast processing time. Like the hard IP cores, the firm IP cores are similar to a final layout with permanent blocks of logic data. Conveniently, it still is configurable to several applications. The last but most flexible of the three kinds, soft IP cores are created in the Hardware Description Language and exist as a gate-level netlist. The strongest point of soft IP core is that it can be configured by the user for mapping to any FPGA design [18]. Using IP cores is convenient, but the limitation is that purchasing many IP cores increases the cost.

## **2.3.3 Vivado and High-Level Synthesis**

### 2.3.3.1 An introduction to High-Level Synthesis

Although we can literally use a hard IP core, or already know how to reconfigure a firm IP core and even create a soft IP core by using HDL. But not everyone has a piece of deep knowledge of this specialized computer language. Thus, specifically for amateurs users, the emergence of the Vivado HLS tool is the leverage for the needs construction of IP cores. In Vivado HLS, an IP core is synthesized from the C/C++ programming language to the Register-Transfer Level (RTL), so it can be implemented on an FPGA fabric, which makes the approach of FPGA projects no longer too complicated for beginners and reduces the amount of code that needs to be written [19]. Furthermore, Vivado HLS provides HLS pragmas, which are the parameters that belong to four main functions: hardware function configuration, loop optimizations, hardware interface configuration like arguments/global variables, and the last is hardware memory implementation configuration. It should be noted that the position of pragmas causes an error if it is put in an incorrect place, the correct position is before the loop block, at the beginning of the function definition block, or before the global/local variable declaration. The advantage of using HLS pragmas in the top-function code includes reducing latency, improving throughput performance, and reducing area and device resource utilization [20].

The advantage of using HLS pragmas in the top-function code includes reducing latency, improving throughput performance, and reducing area and device resource utilization. In addition, Vivado HLS supplies the HLS Test Bench, a tool capable of validating the hardware implementation by simulating the inputs and displaying the corresponding outputs [19]. Particularly, in this thesis, we use the test bench tool of HLS to verify the performance of IP core Sobel to ensure that it is working properly and ready to be executed on hardware. On the other hand, we often forget about the HLS tool's shortcomings because it brings quite a lot of benefits. Vivado HLS, for example, does not support the read/write of large amounts of data through the physical Input/Output (I/O) pins, due to the limited number of on-chip memories. Instead of that, HLS does this work by transferring data among the IP cores, and if additional external memory space is needed, HLS uses the malloc function in the C/C++ programming language library.

### 2.3.3.2 Vivado application tool

Vivado was released in April 2012 under the full name Vivado Design Suite. The main function of this tool is synthesis and analysis for HDL designs. As previously implied, Vivado contains a feature name HLS and due to that the IP core no longer needs to manually create RTL. After IP creation, the IP is synthesized into RTL and packaged in an IP repository, Vivado can read the IP’s information from this repository and extract it for use. The next step is to connect all the related IP cores to generate a block diagram, which contains the main architecture of the system. At last, Vivado supports a tool to validate the block diagram, then synthesize, implement and generate the bitstream of the project, this bitstream file mainly configures the Processing System (PS) part on an FPGA co-design system. More details of PS definitions can be found in section 2.4.1.

### 2.3.3.3 Software Development Kit

Once generated, the bitstream file from Vivado is exported to Vivado Software Development Kit (SDK) environment to configure the software platform and application targeted for the PS part on the ZYNQ processor. SDK works with the hardware designs we generated with Vivado and uses an open-source standard named Eclipse. In SDK, the features come with a feature-rich C/C++ code editor, project management, an application build configuration, and error navigation.

### 2.3.3.4 Xilinx HLS Video Library

Vivado supports a library of code for image processing algorithms called Vivado HLS Video library. This library contains definitions or algorithmic functions for images and has a few highlights in general. Firstly, the images are stored as streams of pixels instead of an array of pixels by using the datatype named hls::stream<>, which helps to prevent random access and prohibits modifying any areas of the image if there are unmodified pixels around. The drawback of this rule is that the operation has to duplicate the image to process more than one function. Secondly, the HLS Video library supports avoiding float and double point operations, because of the high computational cost when using it. The support datatypes consist of ap\_fixed<> and ap\_ufixed<>. Although saving cost, we have to face the bit-inaccurate of the output. Lastly, the library contains necessary data structures such as hls::Scalar, hls::Mat, hls::LineBuffer, and hls::Window for transferring data as AXI-Streams. It should be mentioned that we use the AXI4-Stream I/O functions for the interface to the ports of IP cores and OpenCV interface functions for the interface to the HLS Test Bench. Even though the HLS Video library has the above highlights, this library has only 44 functions for video processing, which means we have to implement any algorithms ourselves if it is not within the HLS library.

### 2.3.3.5 Open source library for image processing

In computer vision, there are many different image processing operations, so we need a library to store all of them for reuse. Because of that reason, OpenCV is created as an open-source library aimed to implement many computer vision functions targeting desktop processors and GPUs [21]. We can use OpenCV independently or combine it with other specific libraries to build an application. In section 2.3.3.4, we mentioned Vivado consists of the Xilinx HLS Video library, but its capability is not enough for the validate purposes. Therefore, Xilinx’s designers have integrated the OpenCV library into the Vivado HLS tool, which supports gaining the best optimization for a real-time video processing simulation. This thesis used the OpenCV library to write a function of Sobel Filter, it acts as the edge detection system on the software platform. Besides, we utilize the OpenCV library in HLS Test Bench for testing the hardware-implemented application.

## **2.3.4 Dataflow technique**

Dataflow architecture is a computer architecture but on the other side to the traditional Von Neumann or control flow architectures [23]. The differences are Dataflow architecture does not have a programming counter and is asynchronous in conceptuality. It means the ability to execute and the executed instructions of Dataflow are defined only on the available input arguments for commands so that the order of command execution is unpredictable, i.e. the behavior of Dataflow is undefined.

Optimizing Dataflow allows the execution to have overlapped tasks to increase the overall throughput of our design and reduce latency. In Figure 2.10, we assume that figure (a) does not have a data stream path. Hence, there are 8 clock cycles before new input can be processed by func\_A and 8 cycles before the output is written by func\_C. Then move to figure (b) with the implementation of the data stream path, func\_A can start to process new input after every 3 clock cycles (decrease the II value) and it requires only 5 cycles to extract the final output value [22].

|  |  |
| --- | --- |
|  | |
|  |  |
| (a) | (b) |

**Figure 2. 7** (a) without dataflow pipelining and (b) with dataflow pipelining

## **2.3.5 Python tool**

High-level data structures, interpreted, and object-oriented are the words to describe Python. Programmers usually choose this programming language due to its available extensive standard library and ease to debug. As we know, the Python tool is the code editor for users, all you need when using this tool is proficiency in Python. The Python tool includes many data-science packages such as jupyter, matplotlib, pandas, numpy, spicy, and even OpenCV. For that reason, the OpenCV-Python environment is decided as the software application to execute the Sobel Filter, which is to compare output results with co-design implementation. It also has to interpret that all code of Python is created and edited by a Python installation named Integrated Development and Learning Environment (IDLE).

# **2.4 ZYNQ-7000 platform**

## **2.4.1 Introduction to ZYNQ board**

Along with the development of technology, many companies now can pack all the components of a system into a single chip. A new concept called SoC (System on Chip) is emerged, which is the result of integrating a complicated design on just a single chip and is a well-known trend in electronics. An SoC silicon chip usually consists of hardware processor units, programmable logic, I/O interfaces, and dedicated features. Before the invention of ZYNQ, almost processors are dependent on FPGA (Field Programmable Gate Array) hardware, which caused complicated communication between PL and PS. Due to that, ZYNQ architecture is the newest generation of the Xilinx SoC product line, it is the co-design by combining a dual-core ARM Cortex-A9 processor and Artix-7 FPGA Series logic unit. In ZYNQ, communication between discrete components is through standard AXI communication, which supports achieving high bandwidth and low latency. ZYNQ AP SoC is divided into two subcomponents: PS (Processing System) and PL (Programmable Logic). Figure 2.11 illustrates the overview of ZYNQ AP SoC architecture, with the PS part on the green background and the PL part on the yellow. We have to figure out that the PCle Gen2 controller and Multi-Gigabit data transmitter is not available on ZYNQ-7010 [24].

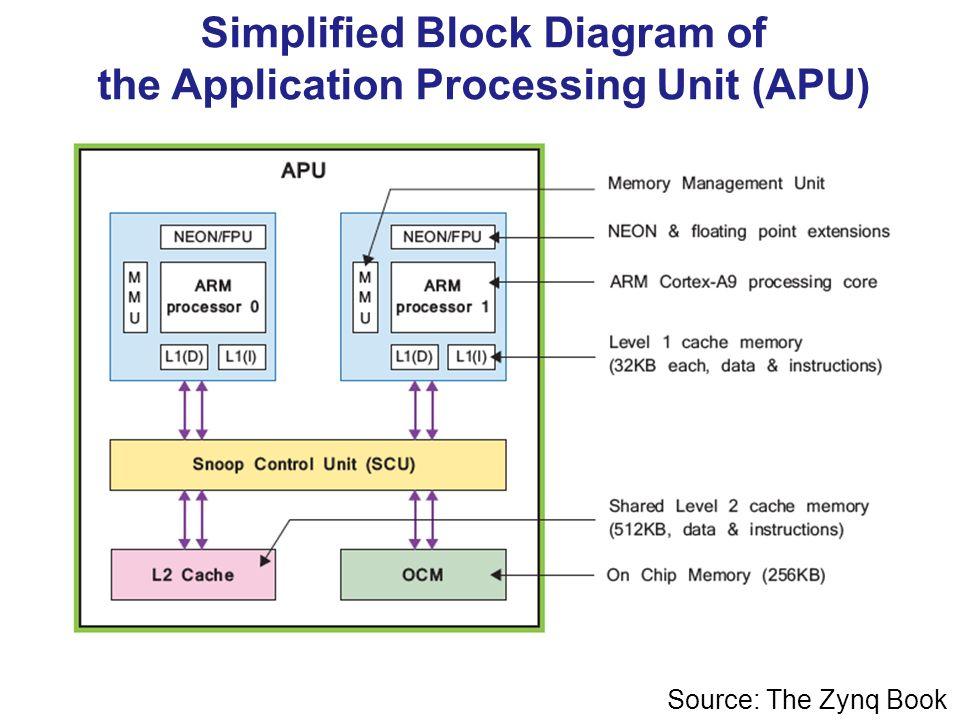
Diagram

Description automatically generated

**Figure 2. 8** Overview of ZYNQ AP SoC Architecture

The PL can be considered almost identical to Xilinx Artix-7 FPGA Series. However, it literally contains more ports and buses that support the tight connection between PL and PS. In fact, PL can not approach the same hardware configuration as a typical 7-Series FPGA, it must be configured directly by the processor or configured through the JTAG port. The PS includes many components such as APU (Application Processing Unit – contains dual-core Cortex-A9 processor), AMBA (Advanced Microcontroller Bus Architecture) Interconnect, DDR3 memory controller, and peripheral controllers consisting of inputs and outputs that are combined with 54 multiplexed I/O pins (as known as MIO). If that peripheral controller does not have inputs and outputs connected with MIO, it could alternatively connect to I/O interfaces via the PL by an extended MIO interface (Extended-MIO or EMIO). On the whole, the peripheral controllers connect to the processor as slave mode through the AMBA interconnection, it contains control registers of read/write functions – these registers are all addressable in the memory space of the processor. The PL is also connected to the interconnect interface as slave mode, so it is possible to use multiple cores in one FPGA structure – each of which has addressable control registers. Not only that, the implemented cores that are used in PL can trigger interrupted mode for the processor and do DMA access to DDR3 memory [24].

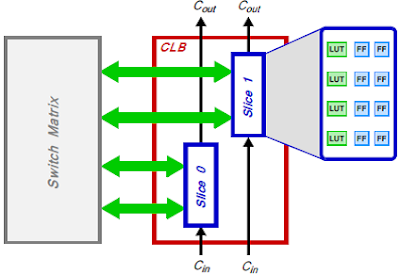
       To achieve deep research of ZYNQ's PL, there is a component that needs to be discussed – APU. The APU includes a dual-core ARM Cortex-A9 processor, each core is equipped with NEON multimedia co-processing units, FPU (Floating Point Unit), MMU (Memory Management Unit), and L1 level cache memory for both data and commands. APU also includes an L2 level cache memory and an on-chip memory (OCM) used for both two cores of the processor. All components are shown in Figure 2.12 below [24].



**Figure 2. 9** APU’s block diagram

The communication between PS and peripheral devices can be done directly through 54 pins MIO block. If we use EMIO, the connection now is done via PL block. I/O interfaces include two SPI (Serial Peripheral Interface), which provide a 4-wire serial communication method for both transmission and reception. Moreover, it has two I2C (Inter-Integrated Circuit) which provide a 2-wire serial communication method, and two CAN (Controller Area Network) – a standard serial communication (commonly used in the automotive industry). In addition, two UART (Universal Asynchronous Receiver Transmitter) – a low-level serial communication interface (commonly used to connect terminals to a computer for debugging purposes) and four 32-bit parallel-interface GPIO (General Purpose Input/Output) are also parts of I/O interfaces. Finally, it has two SD interfaces use to pair with SDCard, two USBs (Universal Serial Bus) that are compatible with 2.0 standard USB, and two Ethernet interfaces that support 10Mbps, 100Mbps, and even 1Gbps connecting speed [24].

One more component in PL that needs to be mentioned is the programmable CLB (Configuration Logic Block). Each CLB contains two slices, each of those is a set of hardware resources for implementing the sequential and combinational logic circuits. In ZYNQ architecture, each slice consists of four LUTs, eight Flip-Flops, and other logical resources. Moreover, each CLB is routed with a switch matrix that provides a flexible routing functionality for creating connections between elements in that CLB or elements from the CLB block to other resources in PL. ZYNQ also provides two functional-hardware blocks, which include Block RAMs (BRAMs) and DSP48E1. Figure 2.13 shows the structure of CLB [24].



**Figure 2. 10** CLB’s structure

## **2.4.2 Advanced Extensible Interface protocol**

To build a complex system by connecting PS with dedicated IP cores on PL, we need a communication interface between them. The main factor of this communication is AXI. AXI is a part of AMBA – a controlled bus subdivision of ARM. Over the last years, ARM has introduced various versions of the AMBA protocol, which describe the connection between processing core and peripheral devices. We know that the first version of AXI was released in commercial in 2003 and belonged to the 3.0 standard AMBA. Then, 4.0 standard AMBA was announced in 2010, which included the second version of AXI – AXI4. There are three types of AXI4 respectively AXI4 (AXI4 Memory-mapped), which allows enabling high-performance memory-mapped requests, a 4-Lite, which allows enabling simple and low-bandwidth memory-mapped, and an AXI4-Stream, which permits data transmission at high speed [24].

The AXI protocol, or more especially the communication between an AXI master and an AXI slave represents the exchange of information between IP cores. Inside that, the memory-mapped connection between the master and the slave uses a structure called Interconnect block. For example, IP Xilinx AXI Interconnect contains communication standards between AXI master and slave and uses to make communicated routing of them [25]. Both AXI4 and AXI4-Lite protocols have 5 channels: AR (Read Address channel), R (Read Data channel), AW (Write Address channel), W (Write Data channel), and B (Write Response channel). Data can simultaneously move in both two directions between master and slave, and data size may be different. While AXI4-Lite allows only one data to be exchanged per time, with AXI4 the limit of one execution can be up to 256 data units. Figure 2.14 shows us one period of the AXI4 read execution, which uses the AR channel and R channel.

**Diagram, table

Description automatically generated**

**Figure 2. 11** Read data period of AXI4 protocol

Figure 2.15 shows us one period of the AXI4 write execution, which uses the AW channel, W channel, and B channel.

Diagram, table

Description automatically generated

**Figure 2. 12** Write data period of AXI4 protocol

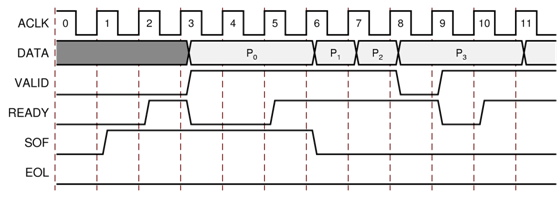
In AXI4 and AXI4-Lite protocol, to transmit a read/write signal by the AXI master, it has to send a read or write address along with data to the AXI slave. In contrast, in the AXI4-Stream interface, there is no need to send any read/write address from the AXI master, and data transmission just only goes in one direction. In short, we can understand that AXI4-Stream is a protocol designed to transmit data streams in one direction [25]. Figure 2.16 shows us the way how data be transferred from master to slave in the AXI4-Stream interface.

Diagram

Description automatically generated

**Figure 2. 13** AXI4-Stream data transferring

In AXI4-Stream, TDATA width (in bits) – or as known as the transferred data width is transmitted per one clock cycle. Transmission is started after the send signal called TVALID was transferred from sender to receiver and the sender had already got the response via TREADY signal from the receiver. TUSER signal has a mission of notifying the first byte of data frame has been transmitted and is called SOF (Start of Frame). TLAST signal notifies that the last byte of the data stream has been transmitted, also known as EOL (End of Line). AXI4-Stream also has optional features such as sending information about data location by TKEEP and TSTRB signals, which helps us to concatenate information between data and its location on TDATA streams; or stream routing by TID and TDIST signals, which roughly corresponds to stream identifier and stream destination identifier [25]. Figure 2.17 shows us the data transmission period of AXI4-Stream with the pulse status chart of ACLK, TDATA, TVALID, TREADY, SOF, and EOL signals.



**Figure 2. 14** Pulse chart of AXI4-Stream data transmission period

# **Chapter 3 SYSTEM DESIGN**

# **3.1 Implementation of Sobel Filter based on software application**

In this section, we executed the edge detection algorithm on the Python software application. Specifically, the code for the algorithm is referenced from the OpenCV library and is edited on Python IDLE. As mentioned in chapter 2, the edge separation algorithm consists of steps: read data from the input image, convert color-to-gray image, blurring, perform Sobel Filter, thresholding, calculate output results, and finally export output image. Expectedly, the OpenCV library provides the ***cv2.cvtColor*** function for converting color image to gray image, function ***cv2.Sobel*** to perform the Sobel Filter, operations from the “math.h” library to calculate the necessary values such as output amplitude, MSE, PSNR, and two functions ***cv2.imread***, ***cv2.imshow*** to import and export data. The flowchart of this implementation is shown in Figure 3.1 and the results obtained from it are presented in chapter 4.

Diagram

Description automatically generated

**Figure 3. 1** Flowchart over the Sobel Filter on OpenCV - Python

# **3.2 Implementation of Sobel Filter based on co-design platform**

## **3.2.1 System requirement specification**

In this section, the Sobel algorithm is implemented on a co-design platform. Therefore, the requirement for this system is to detect the image's edge better than the implementation of the software application, i.e. get the clearest edge of the image. Besides, the system must satisfy the factors: low MSE, high PSNR, low power consumption, and gain the maximum saving of hardware resources. In addition, this system also needs to be able to meet applications at a mid-range level and have the potential for improvement.

## **3.2.2 System design process**

Vivado tools are used to implement the Sobel algorithm, which includes Vivado, HLS, and SDK. As claimed by information in chapter 2, the IP core Sobel is generated by HLS since it is a tool that helps synthesize any project from the C/C++ algorithm to RTL. Vivado is responsible for generating the hardware block diagram, which plays a role as the PL of the entire co-design. Finally, SDK is launched for the software platform configuration. All steps are shown in Figure 3.2.

Diagram

Description automatically generated

**Figure 3. 2** Design process using Vivado tools

We can outline the step-by-step things to do.

***Step 1*** – Open Vivado HLS, create a new project, then select the hardware platform and clock speed. After that, write the code of the edge detection algorithm and its test program to simulate. Once the validation is successful, synthesize and pack the IP core Sobel into an IP repository which can be read by Vivado.

***Steps 2 and 3*** – In step 2, open Vivado, create a new project, then select the hardware platform, and add all necessary IP cores to the “IP Catalog” of Vivado. In step 3, use the IP repository to pick up IPs and connect them. Notice that we can connect ourselves or use the automatic connection tool.

***Step 4*** – Add the associated “Constraints” file (.XDC) which contains hardware connection declarations and clock initialization definitions. A design can include multiple Constraint files.

***Step 5*** – After adding Constraints to the design, validate the block connection, synthesize and implement the system.

***Step 6*** – Create a Bitstream (.bit) file to configure the PL part. This file contains binary strings that describe the entire design, including clock value, I/O connections,…

***Steps 7,***  ***8, and 9*** – Export the hardware design to the SDK tool and create the Board Support Package (BSP). Then write code and compile it to generate the binary file (.elf) for configuring the PS part.

***Step 10*** – Load and execute the program on the ZYNQ board.

It has to be clarified that this program aims to fetch the input data by an High-Definition Multimedia Interface (HDMI) port and extract it via a display monitor by a Video Graphics Array (VGA) port. Both two ports are available on the hardware, but to use them, their related blocks have to be designed. Despite that, the entire system is still not able to display the output.

#### 3.2.2.1 Flowchart of the edge detection algorithm

Figure 3.3 illustrates a flowchart over the edge detection algorithm. It only utilizes two open-source libraries in the PS - either the OpenCV or the HLS Video library. Firstly, the input image is converted from AXI to matrix format. Secondly, it continues to be converted from RGB to Grayscale image. After that, the Sobel Filter is performed, this works on the PS part of the co-design system. Immediately after the image's boundaries have been extracted, it is converted back again to RGB and AXI format to prepare for the output display.

Diagram

Description automatically generated

**Figure 3. 3** Flowchart over the entire edge detection system

#### 3.2.2.2 Overview of PL and PS parts co-operation

Since it was mentioned that the edge detection algorithm is performed in both PS and PL parts, Figure 3.4 shows a detailed overview of how the PL and the PS parts work respectively. In this flowchart, the blue blocks symbolize how the image data is being transferred between types of hardware such as ZYNQ board, DDR memory card, and HDMI ports. The gray blocks show how the algorithm performed in the PS part. Meanwhile, the orange blocks show how the algorithm performed in the PL part.

Diagram

Description automatically generated

**Figure 3. 4** Flowchart illustration for the overview of the PL and PS parts co-operation

In Figure 3.5, the same color coding is used on the hardware illustration for the overview of the PL and PS parts cooperation.

Diagram

Description automatically generated

**Figure 3. 5** Hardware illustration for the overview of the PL and PS parts cooperation

#### 3.2.2.3 Flowchart over the PL part of IP core Sobel

Figure 3.6 thoroughly illustrates the PL part of the IP core Sobel. Firstly, the input data of the image to the IP core Sobel is converted to a matrix by using the hls::AXIvideo2Mat instruction. The reason is despite transferring as AXI format between different cores, the data can not be computed with the convolution operation if it is not displayed under a matrix format. Secondly, the input RGB image is transformed to the Grayscale model, in the HLS library this function is encoded as the hls::CvtColor<HLS\_RGB2GRAY> instruction. After the color conversion, as stated in section 2.3.3.4 that the HLS Video library stores image as streams of pixels, so it can not be able to access while all instruments have not done yet. Hence, the third step is image duplication to ensure that the input gray-scale image is stored to the step of MSE, PSNR calculation, the result of duplication was called ***golden\_image***. The fifth and sixth steps are responsible for the pre-processing before extracting output, which directly converts the image back to the RGB-color model. The processed image was lastly transformed to the AXI format.

Diagram

Description automatically generated

**Figure 3. 6** Flowchart over the PL part of IP core Sobel

#### 3.2.2.4 Flowchart of the IP core Sobel in Test Bench

When executing an algorithm, it is crucial to utilize the Test Bench tool. Because of the ability to simulate the hardware implementation, the Sobel test program is helpful in the IP's functions validating purpose. Once the IP works effectively, it could be embedded in the actual hardware. A flowchart over the IP core Sobel in the HLS Test Bench can be seen in Figure 3.7. An additional library like OpenCV has to be installed to use any functions that are not within the HLS Video library. The Sobel algorithm is referenced by the Sobel module of the top-function code in the previous section. But because the input image is already read as matrix format by the cv::Mat instruction, the data have to be converted again to the AXI stream since the input of the Sobel module is as that datatype. This instruction is performed by the cvMat2AXIvideo, and it is lastly being converted to matrix to be displayed by the AXIvideo2cvMat. Since the simulation acts the same as implementation on hardware, the MSE and PSNR are also determined using some instructions from the **“*math.h”*** library.

Diagram

Description automatically generated

**Figure 3. 7** Flowchart of the IP core Sobel in HLS Test Bench

## **3.2.3 The PL part configuration**

The PL part includes the following blocks shown in Figure 3.8.

Diagram

Description automatically generated

**Figure 3. 8** Edge detection system’s block diagram

In short, there are the main functions of two blocks above.

* ***Pre-processing block*** – this portion receives and decodes the input stream into the 24-bit RGB data type that is fetched through the HDMI port, then convert it to AXI4-Stream data protocol to help transfer it among the system.
* ***Central processing block*** – this portion executes the edge detection algorithm, specifically the Sobel filter to help extract all boundaries of the input image. Then storing the output image in the DDR memory.

#### 3.2.3.1 Pre-processing block design

Design the pre-processing block needs the following IPs: IP DVI to RGB, IP Constant, IP Clocking Wizard, IP Video In to AXI4-Stream, and IP Video Timing Controller.

The first IP is IP DVI to RGB. On PL, we connect this IP’s input directly to the HDMI port to decode the input stream into 24-bit RGB data, where each data channel carries an 8-bit R, G, or B bus.

Figure 3.9 shows us the block diagram of this IP. The data channels allow a considerable skew between themselves and work dependently so that we do not care about the deflected phase with the pixel clock or serial clock.

The clock channel has one 10-bit character transmitted per one data channel period. In which, a 10-bit character is divided into 8-bit useful data and 2-bit control data. Therefore, the output of this IP allows obtaining a control signal during its rest period (empty time) and pixel data during its active time [26].

Diagram, schematic

Description automatically generated

**Figure 3. 9** DVI to RGB converter block diagram

This IP works based on a 200 MHz clock pulse which is given by its input bus, and we also need to generate a clock pulse at the TMDS\_Clk\_p port, its value is dependent on the image resolution. Since this thesis uses a resolution of 1280x720 pixels, the corresponding frequency is 108 MHz (calculated according to the manufacturer’s recommendation), from that the specified pulse period of 9.259 ns is determined. This definition is initialized by a declaration in the hardware Constraints file [26].



|  |
| --- |
|  |
|  |  |

In this block design, the interface of IP DVI to RGB looks like Figure 3.10 below, which has all signals listed and described in Table 3.1. In which, notice that asserts means setting the signal to active mode and de-assert means setting the signal to inactive mode.

Diagram

Description automatically generated

**Figure 3. 10** IP DVI to RGB top-level diagram

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Signal Name** | **Interface** | **Signal Type** | **Init State** | **Description** |
| RefClk | - | I | N/A | 200MHz reference clock. |
| aRst(n) | - | I | N/A | Asynchronous reset, assert if RefClk is not within specification. |
| pRst(n) | - | I | N/A | Active-high reset synchronously with PixelClk. |
| Clk\_p/Clk\_n | TMDS | I | N/A | DVI Clock Channel. |
| Datap[2:0]/Datan[2:0] | TMDS | I | N/A | DVI Data Channel 0. |
| SCL | IIC | I | N/A | Display Data Channel clock signal. |
| SDA | IIC | IO | HiZ | Display Data Channel data signal. |
| PixelClk | RGB | O | N/A | Pixel clock recovered from TMDS clock channel. |
| pVDE | RGB | O | 0 | Video data valid:  1 = active.  0 = blanking period. |
| pHSync | RGB | O | 0 | Horizontal synchronization video timing signal. |
| pVSync | RGB | O | 0 | Vertical synchronization video timing signal. |
| pData (23:0) | RGB | O | zeros | Video pixel data packed as RGB. |
| pLocked | - | O | 0 | Active-high signal for locked status off PixelClk and SerialClk.  De-assert: asynchronously.  Assert: synchronously to PixelClk.  Zero: PixelClk is a lot or not within specification. |
| SerialClk | - | O | N/A | Fast clock, toggling at 5 times the freq of PixelClk. |

**Table 3. 1** Port descriptions of IP DVI to RGB

Figure 3.11 describes the port connections of IP DVI to RGB. In which, “TMDS” is connected to “HDMI port” on ZynQ-7000 hardware, then we automatically have the “DDC” (Display Data Channel) external port. “RefClk” is connected to the “output clock” of the IP Clocking Wizard to receive a 200 MHz pulse. And the output 24-bit RGB of the input data is transferred to IP Video In to AXI4-Stream by connecting “RGB” to “vid\_io\_in”.Graphical user interface

Description automatically generated with medium confidence

**Figure 3. 11** Port connections of IP DVI to RGB in Vivado

The second IP is**IP Clocking Wizard**, which helps to supply a 200 MHz clock pulse for the input clock of IP DVI to RGB. It means this IP Clocking Wizard is capable of creating a clocking circuit for the desired clock frequency. Figure 3.12 illustrates the block diagram of this IP [27].

Diagram

Description automatically generated

**Figure 3. 12** IP Clocking Wizard Block Diagram

Some of the necessary ports of this IP are listed in Table 3.2 [27].

|  |  |  |
| --- | --- | --- |
| **Port** | **I/O** | **Description** |
| clk\_in1 | I | Single-ended primary input clock port. It is available when a single-ended primary clock source is selected. |
| clk\_out1 | O | Output clock of the clocking network. clk\_out1 is not optional. |
| reset | I | Reset (active-High): when asserted, asynchronously clears the internal state of the primitive, and causes the primitive to re-initiate the clocking sequence when released. |
| locked | O | Locked: when asserted, indicates that the output clocks are stable and usable by downstream circuitry. |

**Table 3. 2** Port descriptions of IP Clocking Wizard

Figure 3.13 describes the port connections of IP Clocking Wizard. In that, “clk\_in1” is connected to the “system clock port” of the ZYNQ board which has a 125 MHz frequency, then it supports the creation of a 200 MHz frequency at “clk\_out1” and connects to the “RefClk” port of IP DVI to RGB.

Diagram

Description automatically generated

**Figure 3. 13** Port connections of IP Clocking Wizard in Vivado

Furthermore, in Figure 3.13, we can see that the reset port is connected to a signal called “dout\_constant” which has a value of 0. The reason is that the Clocking Wizard has an active-high asynchronous reset signal, so we have to keep it low.

Therefore, the third IP is IP Constant, which is used to assign a constant value on the desired bus. This IP has two parameters, the first is bus width and the second one is the value (can be in decimal, binary, octal, and hexadecimal format). Figure 3.14 illustrates the block diagram of the IP Constant [28].

A picture containing diagram

Description automatically generated

**Figure 3. 14** IP Constant Block Diagram

Port descriptions and port connections of this IP are described in Table 3.3 and Figure 3.15 as shown below [28].

|  |  |
| --- | --- |
| **Port Name** | **Description** |
| dout[Const Width – 1 : 0] | The output of the Constant block that drives the constant value. |

**Table 3. 3** Port descriptions of IP Constant

Diagram

Description automatically generated

**Figure 3. 15** Port connections of IP Constant in Vivado

The fourth IP is ***IP Video In to AXI4-Stream***. It has the purpose of converting input video streams (including parallel video data, video syncs, blanks, and data valid) to an AXI4-Stream video protocol, which acts as a bridge to connect between the input data and all IPCore in the system [29].

Figure 3.16 shows us the block diagram of this IP. As we had discussed in chapter 2, the AXI4-Stream protocol consists of ACLK, TDATA, TVALID, TREADY, SOF, and EOL signals. So now we get the point why it needs SOF and EOL, these two signals are necessary to identify data locations on the AXI4-Stream bus since there are no sync or blank signals existing in the transmit process.

According to the block diagram, we see that IP Video In to AXI4-Stream needs to be worked along with IP Video Timing Controller to detect the information of the incoming video. We discuss this IP later for more details in the next part.

Diagram, schematic

Description automatically generated

**Figure 3. 16** IP Video In to AXI4-Stream with Video Timing Controller Block Diagram

In this system, IP Video In to AXI4-Stream interface looks like Figure 3.17 below, which has all signals listed and described in Table 3.4 [29].

Table

Description automatically generated

**Figure 3. 17** IP Video In to AXI4-Stream top-level diagram

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal Name** | **Direction** | **Width** | **Description** |
| vid\_io\_in | I | Except vid\_data is 8-256, all that remains are “1” | Receives input video data. |
| vid\_io\_in\_ce | I | 1 | Native video clock enable. |
| aclk | I | 1 | AXI4-Stream ACLK. |
| aclken | I | 1 | AXI4-Stream ACLKEN. Active High. |
| aresetn | I | 1 | AXI4-Stream ARESETN. Active Low. Synchronous to ACLK. |
| axis\_enable | I | 1 | 1 = enable writing data into FIFO.  0 = disable writing data into FIFO. |
| video\_out | O | All is “1” | The AXI4-Stream output data to transfer among the system. |
| vtiming\_out | O | All is “1” | Video timing signals for sync and blank video.  Includes vtd\_active\_video flag:  1 = active video.  0 = blanked video. |
| fid | I | 1 | Field-ID for AXI4-Stream bus. Changes per time SOF signal appears.  0 = even field.  1 = odd field. |
| overflow | O | 1 | Flag indicating that FIFO has overflowed. |
| underflow | O | 1 | Flag indicating that FIFO has under-flowed. |

**Table 3. 4** Port descriptions of IP Video In to AXI4-Stream

Figure 3.18 describes the port connections of IP Video In to AXI4-Stream in this system. In which, *“vid\_io\_in”* is connected to *“RGB”* of IP DVI to RGB, as we had recently discussed. The *“video\_out”* supplies an output of AXI4-Stream data format which is transferred from the 24-bit RGB format received by *“RGB”*. And *“vtiming\_out”* is connected to *“vtiming\_in”* of the IP Video Timing Controller.

Graphical user interface, text, application

Description automatically generated

**Figure 3. 18** Port connections of IP Video In to AXI4-Stream

The fifth IP is **IP Video Timing Controller**, in an image processing system, it has to control the timing signals of input video for synchronization. Therefore, the main function of this IP is detecting and generating timing signals (including horizontal and vertical pulses or polarity,…). IP Video Timing Controller usually goes along with IP Video In to AXI4-Stream or IP AXI4-Stream to Video Out to detect the format and timing of input data, generate outgoing video timing, and keep the same format between input and output data.

Table 3.5 shows the port descriptions of this IP [30].

|  |  |  |  |
| --- | --- | --- | --- |
| **Port Name** | | **Direction** | **Description** |
| vtiming\_in | active\_video\_in | I | Input Active Video: used to set the *detector active\_size* register. |
| hblank\_in | I | Input Horizontal Blank: used to set the *detector hsize* register. |
| hsync\_in | I | Input Horizontal Synchronization: used to set the *detector hsync* register. |
| vblank\_in | I | Input Vertical Blank: used to set the *detector\_vsize* and *f0*\_*vblank\_h* register. |
| vsync\_in | I | Input Vertical Synchronization: used to set the *detector f0*\_*vsync\_v* and the *f0\_vsync*\_*h register*. |
| clk | | I | Video Core Clock. |
| clken | | I | Video Core active-High Clock Enable. |
| det\_clken | | I | Video Timing Detection Core active-High Clock Enable. |
| gen\_clken | | I | Video Timing Generator Core active-High Clock Enable. |
| resetn | | I | Video Core active-Low Synchronous Reset. |
| vtiming\_out | active\_video\_out | O | Output Active Video: generated active video signal. |
| hblank\_out | O | Output Horizontal Blank: generated horizontal blank signal. |
| hsync\_out | O | Output Horizontal Synchronization: generated horizontal synchronization signal. |
| vblank\_out | O | Output Vertical Blank: generated vertical blank signal. |
| vsync\_out | O | Output Vertical Synchronization: generated vertical synchronization signal. |
| fsync\_out[0:0] | | O | Frame Synchronization Output: each frame synchronization bit toggles for only one clock cycle during each frame. |

**Table 3. 5** Port descriptions of IP Video Timing Controller

Figure 3.19 describes the port connections of the IP Video Timing Controller. As we can see, “vtiming\_in” connects to “vtiming\_out” of IP Video In to AXI4-Stream, the “vtiming\_out” matches with “vtiming\_in” and “gen\_clken” goes with “vtg\_ce” port of IP AXI4-Stream to Video Out since sometimes we need to halt the timing generator for synchronization purposes.

Graphical user interface, text, application

Description automatically generated

**Figure 3. 19** Port connections of IP Video Timing Controller

Finally, the following ports “PixelClk”, “aclk”, “clk” are connected for the synchronization clock. Figure 3.20 illustrates the finalized pre-processing block design.

Graphical user interface, application

Description automatically generated

**Figure 3. 20** Input block design in Vivado

#### 3.2.3.2 Central processing block design

Generating a central processing block is exactly building the IP core Sobel. And configure its target hardware to the ZYNQ board.

As we discussed in chapter 2 – the part about AXI4 protocol, the data through IP core Sobel are in AXI4-Stream format, which has 9 signals to carry the meaningful data and notify data status called respectively TVALID, TREADY, TDATA, TKEEP, TSTRB, TUSER, TLAST, TID, TDEST. These all signals are defined as input stream with the image before detecting the edge and as output stream with the resulting image. Since this IP uses a Dataflow structure, we set up its I/O interface as AXI4-Stream.

Graphical user interface, text, application

Description automatically generated

Moreover, IP core Sobel also needs a Constraints file that contains its binding parameter, it declared by the following command. With this binding parameter, this IP works under a 108 MHz pulse clock that is associated with a clock period of 9.259 ns. Notice that the Constraints file of IP core Sobel is a higher priority than the Constraints file at the top-level design of the whole system.



Figure 3.21 shows the port connections of the IP core Sobel. In which, “stream\_in” is connected to “video\_out” of IP Video In to AXI4-Stream, “stream\_out” is connected to “video\_in” of IP AXI4-Stream to Video Out, “ap\_clk” to “aclk” of IP Video Timing Controller. We also need to put “ap\_start” and “ap\_rst\_n” to always value “1” by connecting it to output “hdmi\_hpd” of the second IP Constant.

Graphical user interface, application

Description automatically generated

**Figure 3. 21** Port connections of IP core Sobel in Vivado

The last important part of the central processing block is IP ZYNQ. It acts as a logical connection between the PL and the PS parts of the entire system. Table 3.6 shows the port descriptions of IP ZYNQ, it has to be noted that this thesis just needs to use three main ports. Figure 3.22 shows the port connections of IP ZYNQ, with *“m\_axi\_gp0\_aclk”* connecting with all the asynchronous clock signals of other IP cores. *“DDR”* and *“FIXED\_IO”* are made as external ports [31].

|  |  |  |
| --- | --- | --- |
| Port Name | Direction | Description |
| m\_axi\_gp0\_aclk | I | Global clock signal. All signals are sampled on the rising edge of the global clock. |
| DDR | I/O | DDR external port includes all DDR I/O signals which help communication between ZYNQ and DDR memory card. |
| FIXED\_IO | I/O | Including all the fixed I/O signals such as MIO, FCLK, etc. |

**Table 3. 6** Port descriptions of IP ZYNQ Processing

Graphical user interface, text, application

Description automatically generated

**Figure 3. 22** Port connections of IP ZYNQ in Vivado

# **3.3 Execution of Sobel system**

We need to assign the external ports of the system to package pins on the ZYNQ board for implementation.

Firstly, ZYNQ Board (ZYBO) supplies an external reference clock, which value is 125 MHz and is directly connected to pin L16 on PL. This clock allows PL to be independently used with PS and if the system does not require a processor, this is useful for these simple applications. Figure 3.23 outlines all the clocks supported by ZYBO, including the 125 MHz system clock [24].

Diagram

Description automatically generated

**Figure 3. 23** ZYBO clock diagram

Therefore, Figure 3.24 below outlines all the pins that we need to connect to this edge detection system.

Graphical user interface, application, table, Excel

Description automatically generated

**Figure 3. 24** I/O Ports of Sobel system

After packaging I/O pins, we have to run synthesis, run implementation and generate bitstream to create needed execution files. Then connect the hardware board to an input image source (laptop, camera,…) and output monitor. The final bitstream file is embedded in the hardware target, all steps of this process are done by using the Hardware Manager tool of Vivado.

# **Chapter 4 RESULT**

According to the combination of three parts created in chapter 3, the block diagram of the edge detection system is given in this chapter. However, because the project is complex, it has some issues and the entire design was not able to be finished in hardware; it can not display the output image. Nevertheless, the investigated result of PL design, the algorithm, HLS synthesis and validation as well as OpenCV- Python is shown to provide proof of the importance when executing an image processing on a co-design platform.

# **4.1 Block diagram of Sobel system**

Figure 4.1 illustrates an overview of the Sobel system's block design created in Vivado. It includes most premade IP blocks that were introduced in chapter 3.

Diagram, schematic

Description automatically generated

**Figure 4. 1** Entire block diagram of Sobel system

# **4.2 Measurements of Sobel system based on HLS implementation**

## **4.2.1 Power consumption**

The power consumption of the Sobel system can be seen in Figure 4.2. It is summarized by Vivado Design Evaluation tools. Dynamic power represents the required power when this application is running. It is calculated by the average of switching activity over some time. In Dynamic power, Processing System 7 consumes the most power, while BRAM and DSP consume the least power. Besides, the Static power represents the minimum power that must be provided while the system operates.

|  |  |
| --- | --- |
| Graphical user interface, application  Description automatically generated | Graphical user interface, table  Description automatically generated |
| (a) | (b) |

**Figure 4. 2** (a) Power summary (b) Power On-chip

## **4.2.2 Throughput value and Hardware utilization**

In Vivado HLS, the Sobel system uses a clock period of 13.5 ns. Figure 4.3 shows a summary table with the estimated clock period of 12.49 ns. It seems the timing is not really met the requirement with a margin of 1.01 ns (the required smallest margin is 1.69 ns), fortunately, the actual timing shown in Figure 4.5 is met. The second table mentioned the latency and interval, we can see that the maximum throughput latency is 928503 clock cycles and it can start to process new input data after 928498 clock cycles. This results in a throughput of ~78 frames/s (1/(928503 cycles/frame \* 13.5 ns/cycle)).

Graphical user interface, text, application

Description automatically generated

**Figure 4. 3** Measurements of timing and latency in the synthesis report

In Figure 4.4, it can be found that the hardware utilization is meet the needed usage volume of the IP core Sobel, because the usage of BRAM, DSP48E, FF, and LUT is very low within the available. For more detail, the below table shows how the hardware resources were divided into different functions in HLS. The Sobel function obviously used the most resources.

Table

Description automatically generated

**Figure 4. 4** The estimated hardware utilization of IP core Sobel

The report obtained from the RTL exportation gives the actual results of IP core Sobel implementation. A larger timing margin was achieved of 3.53 ns, instead of 1.47 ns in the synthesis estimation. In addition, the utilization of BRAM, FF, and LUT is significantly lower than what was estimated. We can see that in Figure 4.5 below.

Table

Description automatically generated with low confidence

**Figure 4. 5** The RTL exportation of IP core Sobel

Figure 4.6 illustrates the ordering of performing functions, which can be observed by using the analysis tool in HLS Performance.

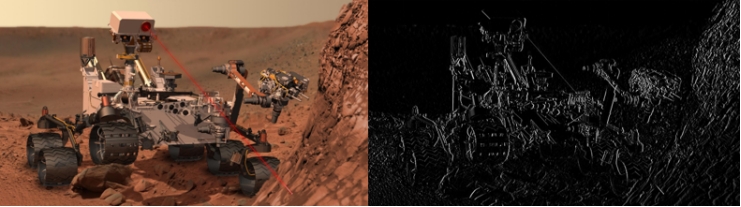
Table

Description automatically generated

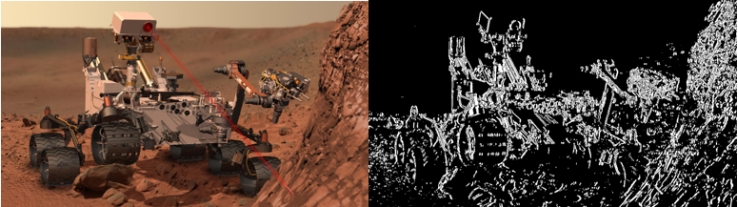
**Figure 4. 6** The ordering of performing functions by HLS Performance

# **4.3 Output result and comparison**

The input and output image of the Sobel Filter implemented on the ZYNQ co-design platform is shown in Figure 4.7. Notice that this is the result from the HLS Test Bench. Besides, Figure 4.8 shows the output when implementing the Sobel Filter on OpenCV – Python.



**Figure 4. 7** Result of edge detection algorithm on ZynQ-7000 platform



**Figure 4. 8** Result of edge detection algorithm on OpenCV – Python

The result parameters that need to be compared will be shown in Table 4.1 below.

|  |  |  |
| --- | --- | --- |
| Platform | MSE | PSNR |
| OpenCV – Python | 15234.953 | 6.30239 |
| ZynQ-7000 | 2628.879 | 13.93310 |

**Table 4. 1** Result parameters comparison

As we can see, on the ZYNQ-7000 platform, the boundaries of the image are fetched more carefully and smoother than the OpenCV – Python, which saves a lot of capacity in the memory. It interprets how the co-design has lower power consumption. Additionally, the value of MSE is less than and PSNR is greater than OpenCV – Python, respectively 5.8 and 2.2 times. It means the output image quality on the ZYNQ-7000 platform is better because we know that the lower the MSE and higher the PSNR are, the better the quality of the processed image is.

# **Chapter 5 CONCLUSION AND FUTURE WORK**

# **5.1 Conclusion**

In general, this thesis is regarded as providing quite coherent proof for the reason why implementing an image processing algorithm based on a co-design platform is better than a software platform. According to all obtained result indicators of MSE and PSNR in the previous chapter, we can see that the quality of the output image on the Python software application is worse than on the ZYNQ co-design. It is found that the implementation on co-design is memory saving, assume that this system is used as a subset of another larger system, there are several steps after detecting edge so the optimization in data storing is very important. On the other side, the energy consumption of 1.936 W and the throughput of 78 frames/s provide a power consumption of 0.024 J/frame. This is a positive result and has the potential to gain maximum savings of hardware resources if some effective solutions are added to the design system. Due to the complexity of the algorithm development, the adjustment of the target period clock is a considerable proposal for future work, as mentioned in section 5.2.

However, it needs to take a step back, observe, and conclude the weaknesses of this thesis. Firstly, the entire system has never been successfully implemented on the ZYNQ PS. The Sobel design in Vivado and SDK occurs some issues with the VGA configuration and the result is can not be displayed on the monitor. This makes it impossible to check out the output image and the execution time. Secondly, the Sobel design in Vivado is not really optimized since it has the Worst Pulse Width Slack of 0.185 ns (WPWS - a parameter that indicates the minimum width of highest and lowest pulses of a clock), fortunately, it seems to be within the required limitation. The third drawback is the Sobel algorithm is not considered an innovation-driven operation for image processing, because there are other better algorithms such as Canny and LoG. But the complexity of them is a hard challenge to the implementation.

# **5.2 Future work**

# **REFERENCES**

|  |  |
| --- | --- |
| [1] | Nguyễn Thanh Hải, Giáo trình xử lý ảnh, Đại học Quốc Gia Thành phố Hồ Chí Minh, 2003. |
| [2] | Soma Prathap; Jatoth Ravi, "Hardware Implementation Issues on Image Processing Algorithms," *National Institute of Technology Warangal,* 2018. |
| [3] | Sagharichi Ha, Pooya; Shakeri, Mojtaba, "License Plate Automatic Recognition Based on Edge Detection," *Faculty of Computer and IT Engineering,* 2016. |
| [4] | Thenmozhi, K; Reddy U, Srinivasulu, "Image Processing Techniques for Insect Shape Detection in Field Crops," *International Conference on Inventive Computing and Informatics,* 2017. |
| [5] | Abbasi, Tanvir, "A Proposed FPGA Based Architecture for Sobel Edge Detection Operator," 2007. |
| [6] | Halder, Santanu; Hasnat, Abul; Khatun, Amina; Bhattacharjee, Debotosh; Nasipuri, Mita, "A Fast FPGA Based Architecture for Skin Region Detection," *International Journal of Innovative Technology and Exploring Engineering,* 2013. |
| [7] | Rafael C Gonzalez; Richard E Woods, Digital Image Processing, New York: Pearson, 2018. |
| [8] | Rang M. H. Nguyen; Michael S.Brown, "Why You Should Forget Luminance Conversion and Do Something Better," *Computer Vision Foundation, IEEE Xplore,* 2017. |
| [9] | Nguyễn Quang Hoan, Giáo trình xử lý ảnh, Học viện Công nghệ Bưu chính Viễn thông, 2006. |
| [10] | Sung Kim; Riley Casper, "Applications of Convolution in Image Processing with MATLAB," *University of Washington,* 2013. |
| [11] | Ramesh Jain; Rangachar Kasturi; Brian G. Schunck, "Edge Detection," in *MACHINE VISION*, McGraw-Hill, 1995, pp. 140-185. |
| [12] | Ansari, Mohd; Kurchaniya, Diksha; Dixit, Manish, "A Comprehensive Analysis of Image Edge Detection Techniques," *International Journal of Multimedia and Ubiquitous Engineering,* 2017. |
| [13] | Oskar Mencer; Dennis Allison; Elad Blatt; Mark Cummings; Michael J. Flynn; Jerry Harris; Carl Hewitt; Quinn Jacobson; Maysam Lavasani; Mohsen Moazami; Hal Murray; Masoud Nikravesh; Andreas Nowatzyk; Mark Shand; Shahram Shirazi, "The History, Status, and Future of FPGAs: Hitting a nerve with field-programmable gate arrays," 2020. |
| [14] | Ian Kuon; Russell Tessier; Jonathan Rose, "FPGA Architecture: Survey and Challenges," *Foundations and Trends in Electronic Design Automation,* 2008. |
| [15] | Xilinx, UltraScale Architecture DSP Slice, www.xilinx.com, 2021. |
| [16] | Rajewski, Justin, "How does an FPGA work?," Embedded Micro, 2015. [Online]. Available: https://learn.sparkfun.com/tutorials/how-does-an-fpga-work/all. [Accessed 6 July 2022]. |
| [17] | Eastland, Nate, "FPGA - Configurable Logic Block," Digilent Inc. Blog, 2015. [Online]. Available: https://digilent.com/blog/fpga-configurable-logic-block/. [Accessed 6 July 2022]. |
| [18] | M. Rouse, "IP core (Intellectual Property core)," TechTarget Contributor, March 2011. [Online]. Available: https://www.techtarget.com/whatis/definition/IP-core-intellectual-property-core. [Accessed 6 July 2022]. |
| [19] | K. Karras; J. Hrica, "Designing protocol processing systems with vivado high level synthesis," 2014. [Online]. Available: https://docs.xilinx.com/v/u/en-US/xapp1209-designing-protocol-processing-systems-hls. [Accessed 8 July 2022]. |
| [20] | Xilinx Inc., "Vivado HLS optimization methodology guide," 2017. [Online]. Available: https://docs.xilinx.com/v/u/2017.4-English/ug1270-vivado-hls-opt-methodology-guide. [Accessed 8 July 2022]. |
| [21] | Xilinx Inc., "Accelerating OpenCV applications with ZYNQ-7000 all programmable SoC," 2015. [Online]. Available: https://docs.xilinx.com/v/u/en-US/xapp1167. [Accessed 8 July 2022]. |
| [22] | Arthur H. Veen, "Dataflow Machine Architecture," *Center for Mathematics and Computer Science,* 1986. |
| [23] | Xilinx Inc., "Vivado Design Suite User Guide: High-Level Synthesis," 2017. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug902-vivado-high-level-synthesis. [Accessed 8 July 2022]. |
| [24] | Digilent, ZYBO FPGA Board Reference Manual, 2017. |
| [25] | Xilinx Inc., "AXI Reference Guide," 2012. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug761\_axi\_reference\_guide. [Accessed 8 July 2022]. |
| [26] | Digilent, "DVI to RGB (Sink) 2.0 IP Core User Guide," 9 October 2019. [Online]. Available: www.digilentinc.com. [Accessed 12 July 2022]. |
| [27] | Xilinx Inc., "Clocking Wizard v6.0 LogiCORE IP Product Guide," 20 April 2022. [Online]. Available: www.xilinx.com. [Accessed 12 July 2022]. |
| [28] | Xilinx Inc., "LogiCORE IP Constant (v1.1)," 9 April 2018. [Online]. Available: www.xilinx.com. [Accessed 12 July 2022]. |
| [29] | Xilinx, "Video In to AXI4-Stream v4.0 LogiCore IP Product Guide," 18 November 2015. [Online]. Available: www.xilinx.com. [Accessed 23 June 2022]. |
| [30] | Xilinx, "Video Timing Controller v6.2 LogiCORE IP Product Guide," 26 February 2021. [Online]. Available: www.xilinx.com. [Accessed 23 June 2022]. |
| [31] | Xilinx Inc., "Processing System 7 v5.5 Product Guide," 10 May 2017. [Online]. Available: www.xilinx.com. [Accessed 12 July 2022]. |
| [32] | Xilinx Inc., "AXI4-Stream to Video Out v4.0 LogiCORE IP Product Guide," 18 November 2015. [Online]. Available: www.xilinx.com. [Accessed 12 July 2022]. |
| [33] | Digilent, "RGB to VGA 1.0 IP Core User Guide," 23 April 2015. [Online]. Available: www.digilentinc.com. [Accessed 12 July 2022]. |

# **APPENDICES**