EFFICIENT IMPLEMENTATION OF SOBEL EDGE DETECTION WITH ZYNQ-7000

by

Mohammad Tasneem Obaid

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science in Engineering

Department of Electrical and Computer Engineering

Fort Wayne, Indiana

May 2020

THE PURDUE UNIVERSITY GRADUATE SCHOOL

STATEMENT OF COMMITTEE APPROVAL

Dr. Todor Cooklev, Chair

Department of Electrical and Computer Engineering

Dr. Chao Chen

Department of Electrical and Computer Engineering

Dr. Yanfei Liu

Department of Electrical and Computer Engineering

Approved by:

Dr. Chao Chen

Head of the Graduate Program

ACKNOWLEDGMENTS

I first thank Dr. Todor Cooklev for providing me with the opportunity to work on this project through his teachings and guidance. I appreciate his patience and flexibility in working with me as I maintained balance in work, courses and this research. Next, I thank my graduate committee; Dr. Chao Chen and Dr. Yanfei Liu for their support and direction in my thesis writing. I thank Department of Electrical and Computer Engineering for entrusting me with this shining opportunity and also providing the required infrastructure within the school for completion of project. I thank my parents for always encouraging me and instilling in me the value of hard work. Most importantly I thank my wife, Wasima, for her encouragement and understanding throughout my graduate studies

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Edge detection is one of the most important application in image processing. Field-Programmable Gate Arrays (FPGAs) have become popular computing platforms for signal and image processing. The Zynq-7000 System on Chip (SOC) is a dual-processor platform with shared memory. The thesis describes a novel and fast implementation of Sobel edge detection using the Zynq-7000 SoC. Our implementation is a combination of software and hardware using the Vivado HLS and Zynq (SoC). As a result our implementation is fast. We make a comparison with other conventional edge detection techniques and show that the speed of operation of this design is much faster.

1. INTRODUCTION

1.1 Overview

For the implementation of image processing algorithms, it is essential for embedded systems to achieve low power and high performance at the same time. The implementation process involves several steps. The initial step of implementing image processing algorithms is software written in a high-level language like C or C++. Validation and development are much easier in high-level languages. The ultimate goal is a Hardware Description Language (HDL). However, producing optimized HDL code from that a high-level software is not easy. For this task there are High-level Synthesis tools that convert the high-level code into the hardware description language automatically.

We describe the design and implementation of an image edge detection hardware accelerator. The target platform is the Zynq-7000 SoC. Here the Sobel image edge detection algorithm is developed in the Vivado HLS tool and then it is exported to Vivado to use it as an Intellectual Property (IP). After that the implementation of the generated IP is synthesized and tested on Zynq7000 Zedboard. Finally an application in the Software Development Kit is created to use this peripheral in order to apply a Sobel filter in an image which is read from a SD card connected to the board.

The thesis is organized as following. Sobel edge detection is introduced in chapter 2. A brief description of Vivado HLS, different optimization techniques and libraries is given in chapter 3. Chapter 4 describes the Zynq-7000 in short. The development of the image processing algorithm in Vivado HLS and the hardware implementation on the Zynq platform, and performance comparison are shown in chapter 5.

1.2 Advantages of the Developed Implementation

Zynq-7000 SoC, on a single chip, is a combination of FPGA fabric in a Programmable Logic domain and dual-core ARM Cortex-A9 CPUs with a rich set of standard I/O peripherals and a multi-ported memory controller in an SoC Processing System domain. Over 2,000 interconnects interface the Processing System to the Programmable Logic. This provides the high-performance, low-latency communication, extension, flexibility, and capability between processing and programmable logic that other systems connecting discrete processor-based devices to FPGAs through printed circuit boards cannot achieve.

We implement the Sobel filter as a HLS Kernel. This is more efficient than a straightforward implementation like implementation using general processors. We create a unique HLS kernel using very limited resources. For example we use only 1173 Flip Flops from 106400 available Flip Flops. This helps to execute the operation very fast. HLS provides a technique for migrating algorithms into the FPGA logic from a processor. Therefore, it helps moving code from the Cortex ARM A9 processor to the FPGA logic.

There are two more reasons for fast operation. We add the right amount of pragmas and techniques in order to achieve better performance both in memory transactions and computations. A pragma is a technique that helps speed up the code. Block RAM is used to store and process the input data and then write it back to the DDR. To transfer data with bursts we use the 'memcpy' command for the transactions.

In the following sections we will introduce image processing, edge detection, Zedboard and related previous work.

1.3 Image Processing

Image processing is a method of conducting certain operations on an image to produce an improved image or retrieve any valuable information from it. It is a form of signal processing in which input is an image and output may be image or features / functions associated with that image. The processing of images is among increasingly rising technologies nowadays [1]. Image processing basically includes the following three steps:

- Importing the image via image acquisition tools.
- Analyzing and manipulating the image.
- Output in which result can be altered image or report that is based on image analysis.

There are two fundamental types of image processing: analog and digital image processing.

1.3.1 Analog Image Processing

Operations like Laplace and Fourier Transforms can be used to transform the image, based on the algorithm's need. There are a number of problems associated with analog image processing, e.g. image accuracy, noise, blurred image, etc. which makes any implementation difficult. In Digital Image Processing, on the other hand, the image is stored in pixels and mathematical operations can be performed on those pixel values. Any noise level, image accuracy, and other issues, can be eliminated by adjusting the pixel values. That's why digital image processing is used to eliminate the disadvantages of analog image processing [2].

1.3.2 Digital Image Processing

Every pixel value has its own meaning, i.e. the color, brightness etc. Data is stored in matrix form. The first step is to convert the image into pixel values and store them in the matrix when the image is acquired from any source. Then the operations on those matrices are performed. But the issue that occurs when getting the image from any source is that it can distort the image, change color, increase or decrease brightness, erase edges. It is generally called noise. Noise happens while the image is being captured, e.g. real time images, digital imaging, scanners. A slight noise adds irrelevant information that can alter the pixel values of the image. So when taking an image from any source, a proper way has to be found. Gaussian Blur is used to remove noise of any kind from the image [3].

1.4 Image Edge Detection

Edge detection is a technique to identify the boundaries of objects inside images. This works by detecting discontinuities in brightness and color. The points at which image brightness changes sharply are typically organized into a set of curved line segments termed edges. Edge detection is used in fields such as image processing, computer vision, and machine vision for image segmentation and data extraction. Now commonly used edge detection techniques will be discussed.

1.4.1 Canny Edge Detection

Canny Edge Detection Technique is very common edge detection technique for edge extraction in image processing. It's used to get the object's boundaries inside the image. This was built by John F. Canny in 1986. It is often used due to low error rate, proper localization and negligible response. Compared to other edge detection algorithms this algorithm is more difficult to implement because of intense computation. The Canny algorithm gives better edge detection, better localization and direct response [4].

1.4.2 Sobel Edge Detection

Sobel Edge Detection is a technique based on two kernels, one kernel detects the horizontal edges and the other kernel recognizes the vertical edges. Each kernel has the effect of calculating the gradient in both a horizontal and a vertical direction. The image is read initially after start i.e. the pixel values are read. The image is then convolved with the filter. After that horizontal and vertical kernels of the operator are convolved with the original image [5].

1.4.3 Prewitt Edge Detection

The Prewitt operator is used in image processing particularly within algorithms for edge detection. Technically, it is a discrete differentiation operator, which computes an approximation of the image intensity function gradient. At any point in the image, either the corresponding gradient vector or the norm of this vector is the product of the Prewitt operator. The Prewitt operator is based on converting the image in horizontal and vertical directions with a low, separable, and integer weighted filter and is thus relatively inexpensive in terms of computations such as Sobel and Kayyali operators [6].

1.5 ZedBoard

ZedBoard is a low-cost, all programmable (DEFINE SOC) SoC development board for the Xilinx Zynq-7000. This board contains everything to build a system based on Linux, Android, Windows or another OS / RTOS. Additionally, the processing device and the programmable logic I/Os are connected to multiple expansion connectors for quick user access. The advantage is the closely coupled ARM processing device Zynq-7000 AP SoCs and a programmable 7-series logic to construct unique and efficient designs with the ZedBoard [7].

The main features [8] of Zedboard are:

- Xilinx Zynq-7000 AP SoC XC7Z020-CLG484
- Dual-core ARM Cortex™-A9
- 512 MB DDR3
- 256 MB Quad-SPI Flash
- On-board USB-JTAG Programming
- 10/100/1000 Ethernet
- USB OTG 2.0 and USB-UART
- Analog Devices ADAU1761 SigmaDSP® Stereo, Low Power, 96 kHz, 24-Bit Audio Codec
- Analog Devices ADV7511 High Performance 225 MHz HDMI Transmitter (1080p HDMI, 8-bit VGA, 128x32 OLED)
- PS & PL I/O expansion

1.6 Related Work

In this section we review research work related to this thesis.

1.6.1 A hybrid approach using Sobel and Canny operator

Classical methods for detecting edges are noise sensitive due to the implementation of various modes of differential operation. For edge detection noise is observed as edge points instead of actual edge with noise interference. Hence good immunity to noise is required. Sobel edge operator's location is correct, but is noise sensitive. The Canny approach quickly detects, and suppresses, the weak side. Thus, using a hybrid approach enhances the outcome by visually improving and completing the details of the image, there are no false edges and it has an ideal effect. Also, the hybrid algorithm uses a median filter to eliminate any noise. This filtering median smoothens the image data and provides better output. The extraction image is thus composed of fairly complete profile and rich detailed information. It effectively increases edge detection precision and gives quite an ideal effect on edge detection [9].

1.6.2 An Improved Sobel Algorithm Based on Median Filter

The Sobel operator has been combined with median filtering, and this technique can effectively eliminate the image's salt and pepper noise [10]. The explanation for this is that median filtering performs fine when filtering out the jump signal. Mutation of continuous signal induces the stochastic signal of salt and pepper noise, so the combination of median filtering and Sobel operator will help isolate the edge of the image from the salt and pepper noise signal. This algorithm consists of the following steps. The correct median filter template is chosen based on the consistency of noise. Process the image with a median filter, and filter out salt and pepper noise. The edge template coefficient was defined by the Sobel operator template in image 1 to the user. After the Sobel edge detect operator is given to convert with the template for every pixel of the image to make convolution with the template and get the point gradient. The gradient amplitude is the output of the point. And finally, we get the edge detected image [10].

1.6.3 Analysis of Image Quality using the Sobel Filter

The image quality can be analyzed using the Sobel filter [11]. The suggested technique is as shown in Figure 1.1. Assessing the effect of satellite image bandwidth using Sobel filter. The data areprocured and preprocessed during the first phase of the experiment. The Sobel filter is implemented with 3x3, 5x5, 7x7, 9x9 and various window sizes respectively. Finally, the appropriate size of the window is chosen based on the statistical analysis is of mean, standard deviation and SNR [11].

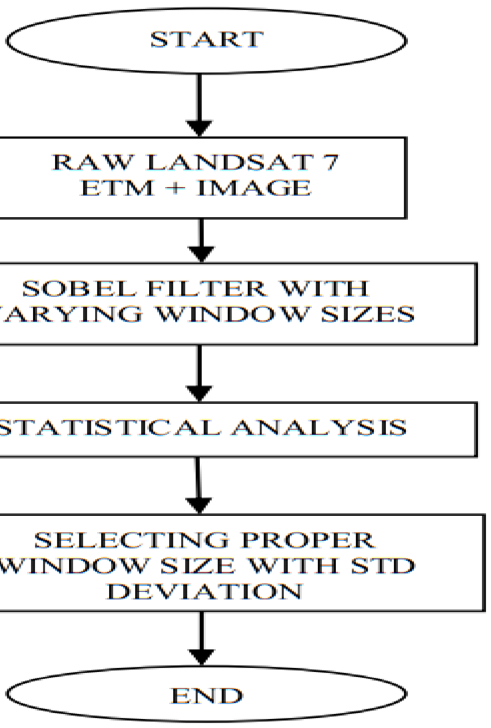

START
RAW LANDSAT 7 ETM + IMAGE
SOBEL FILTER WITH VARYING WINDOW SIZES
STATISTICAL ANALYSIS
SELECTING PROPER WINDOW SIZE WITH STD DEVIATION
END

Figure 1. 1: Methodology to Assess the Impact of Bandwidth using Sobel Filter [11]

1.6.4 Energy Efficient Platform for Sobel Filter Implementation

Using various embedded platforms and parallel computing systems, the Sobel filter algorithm used in image processing has been benchmarked to determine energy consumption and image processing efficiency, making design choices simpler for a software engineer. Test findings show that the Mali T764 GPU on Radxa Rock2 platform appeared to be 6.85x more energy efficient and 3.7x better performing than the Parallella platform while computing Sobel filter with a 1080p resolution picture using 16 core Epiphany co-processor [12].

1.6.5 Gradient Estimation of Distorted Images

The performance of the proposed Generalized Sobel Filter against the implemented Jacobian Gradient Correction (GCJ) process is assessed. Two simple methods in the analysis as reference for comparisons are found: 1) ignoring the distortion (the gradients are measured using the Sobel operator in the distorted space and the effects are not corrected); 2) explicit elimination of the distortion (i.e. rectification) before calculating the gradient using the Sobel operator. Using regular Sobel filters, the reference gradient is determined from the rectilinear reference images. The high-resolution input images are converted to blurred 1024 / 768 pixel standard resolution images. The calculation of the local histograms of gradient orientations is carried out using the following parameters: 1) each image is divided into tiles of 24 to 24 pixels and 2) 18 bins are considered equally spacing the interval [−180 u, 180]. In order to determine the output of the method under consideration with respect to different degrees of distortion, each image in the dataset is processed to add varying percentages of distortion ranging from a minimum of 10% to a maximum of 50% [13].

1.6.6 Optimization of Processor Architecture for Image Edge Detection Filter

The processor architecture has been optimized for the image detection filter [14]. The Sobel instances are linked in a way that exploits the parallelism and I / O capabilities of FPGA in one wide Sobel combinational block. By using the calculation results of previous window operations of the same row and up to two previous rows, the architecture reduces the number of calculations over the whole cycle [14].

1.6.7 Optimized Approach of Sobel Edge Detection Using Xilinx System Generator

For this research a new method is used which combines Sobel X-Y edge detection with Gaussian filter using the histogram stretching method [15]. This research is focused on medical for detecting tumors and fractures in the human body. The authors [15] propose a new optimized edge detection technique, which is a better edge detection technique in terms of improved mean square error (MSE) and image signal to noise ratio (PSNR) using very limited resources used in the FPGA kit development platform SPARTAN 3A DSP 3400A [15].

1.6.8 Edge Detection via IP-Core based Sobel Filter on FPGA

Edge detection is one of the most important image / video-processing applications. The goal of this paper is to extract edges of video streams in real time. The device consists of OV7670 CMOS-Camera module, Digilent Basys3 FPGA plate, and VGA monitor to view the video stream being processed. This analysis consists of three parts: capturing RGB data from the camera module, transforming RGB data into grayscale and then implementing Sobel filter to detect the edges of real-time images. Whole architecture is performed on the Vivado 2018.1 FPGA Architecture Suite using custom IP-Cores encoded with Very High-Speed Integrated Circuit Hardware Description Language (VHDL). From the test, it is shown that the design is realized with high precision and low level of resource utilization for the sake of FPGA's parallel processing capability [16].

1.6.9 Zynq FPGA based System Design for Video Surveillance with Sobel Edge Detection

Advances in the semiconductor domain have allowed various applications to be realized in video surveillance using computer vision and deep learning, video surveillance in industrial automation, defense, ADAS, live traffic analysis, etc. Image comprehension requires high precision input data that is dependent on Image resolution and camera position. Interesting data may be thermal image or live stream coming for various sensors. Composite (CVBS) is a common video interface able to stream up to quality HD (1920x1080). Unlike serial high speed interfaces such as HDMI / MIPI CSI, analog composite video interface is a normal single wire that allows longer distances. Image comprehension includes edge detection and the further processing classification. Sobel filter is one of the most commonly used edge detection filters that can be integrated into live stream. This paper proposes Zynq FPGA-based video surveillance system architecture with Sobel edge detection, where the input Composite video decoded (analog CVBS input to YCbCr digital output), processed in HW, and transmitted to HDMI display, simultaneously stored in SD memory for later processing. The HW architecture is adjustable for resolutions from VGA to Full HD for 60fps and 4K for 24fps. To highlight the usable course, the device is built on Xilinx ZC702 platform and TVP5146 [17].

1.6.10 Sobel Filter Based on GPUs Cards

In a few years the graphics processors or GPUs have become important devices for applications involving massively parallel computation. The applications currently include in multimedia processing, computer science and real-time image processing. They deliver other advantages from an equal Processing capacity, such as treatment acceleration and energy consumption down. In this research, it is demonstrated that the efficacy of the sobel filter approach (extraction of features) by parallelizing the processing applied to various images of different sizes [18].

1.6.11 Parallel implementation of Sobel filter using CUDA

To solve the problem of intensive computation of the image processing applications and to achieve high real-time efficiency, efficient solutions need to be considered. The graphics processing unit (GPU) is an effective and latest tool used to accelerate extensive calculation algorithms to minimize execution time by leveraging the power of parallel programming techniques and attaining the highest efficiency. In this research, it is presented that a parallel GPU implementation of an edge detection algorithm using CUDA (Compute Unifies Architecture) setting, using a Sobel operator. Moreover, by checking the algorithm on a typical central processing unit (CPU) to compare the computational efficiency of such systems, it is evaluated and proved that the high performance of GPU implementation. The experimental findings show that by its higher performance compared to sequential calculation the efficacy of the GPU implementation is [19].

1.6.12 Power Evaluation of Sobel Filter on Xilinx Platform

Power consumption has become a primary factor in design flow in programmable devices. Power consumption is crucial in FPGA designs for powered battery equipment among the key concerns about power consumption, device efficiency, battery life, thermal challenges, or reliability. In this research, it is a review of the FPGA-based architecture for the low cost fall detector Sobel Edge Detection algorithm, and present an accurate evaluation of dynamic power estimation and real-time measurement. This research uses the Root Mean Square Error index to determine the rate of accuracy between estimated power consumption and measured. This application is implemented on the Zynq -7000 AP SoC family, which is low power.

FPGA power consumption depends on the design and is determined by factors such as clock frequency, levels of operation, logic block and interconnection structure, voltage power supply and output loading. XPA may provide an instrument for deriving the strength of FPGA implementations. The frequency of the clock is set at 197.161 MHz and calculated power is the amount of static and dynamic energy. The static power depends on the family unique to FPGA.

Dynamic control depends on the operation of logic, input / output, signaling and clock switching [20].

1.6.13 An Improved Sobel Edge Detection

This research proposes a method combining Sobel edge detection operator and de-noise soft-threshold wavelet to detect edges on images that contain white Gaussian noises. A lot of methods of edge detection are being proposed in the last years. The widely used methods that combine mean de-noising with Sobel operator or median filtering and Sobel operator can't very well eliminate salt and pepper tone. We first use soft-threshold wavelets in this paper to remove noise, then use Sobel edge detection operator to detect edges on the image. This method is used primarily on images which include white Gaussian noises. From the pictures obtained from the experiment we can see very clearly that the approach proposed in this research has a more noticeable impact on edge detection compared to conventional edge detection methods.

Sobel edge operand has the advantage of its smoothing effect on the random noises in the picture. And because it is the differential divided by two rows or two columns, the edge elements on both sides have been improved and make the edge appear dense and light. Sobel operator is operator of a gradient. The first derivative of a digital image is based on a variety of approximation of two-dimensional gradient and produces a peak on the image's first derivative, or a zero-crossing point on the second derivative. Calculate the magnitude and the argument value of the first-order or second-order horizontal and vertical gradients of the image, finally calculate the maximum modulus along the angular direction and get the image point. But when the picture has lots of white Gaussian noises, the peak value of the first derivative is very difficult to achieve, the explanation being that the noise points and the useful signals mix up. This paper therefore incorporates de-noising of Sobel operator and soft-threshold wavelet [21]. The core idea of the algorithm [21] is:

- Do wavelet decomposition to the image matrix, and get noise from the wavelet coefficients.
- Process the HL, LH and HH wavelet coefficients obtained via the decomposition, and hold the low frequency coefficients unchanged.
- To eliminate Gaussian white noise signals, pick the correct threshold.

- Do inverse wavelet transformation to matrix of image and get the matrix of image after de-noising.
- After the Sobel edge detection operator template is given, convolute with this template on every pixel of the image, get the gradient of this point and the amplitude of the gradient is the output of this point. Finally a picture of the edge detection is found.

1.6.14 Reconfigurable Computing Architectures

FPGAs offer a road to the assurance of hardware customization without the immense development and manufacturing costs and lead times of custom Very Large Scale Integration (VLSI). More computational throughput per unit of silicon can be derived from FPGAs than from the processors [22].

1.6.15 Introduction to Reconfigurable Systems

Reconfigurable systems are with soft-definable features. They can be as basic as a single switch that parameterizes a specific feature on a chip, or as complex as a data center that houses computer banks for a cloud computing application where hundreds of virtual machines can be temporarily marshaled to form a dedicated network to solve a computing problem or converted into a cluster of media servers to provide entertainment on demand for a large number of users at the same time [23].

2. SOBEL EDGE DETECTION AND DESIGN FLOW

2.1 Overview

Edge Detection is a mathematical method of trying to find the regions in an image where it has a sharp change in intensity or a sharp change in color. A high value indicates a steep change and a low value indicates a shallow change. There are several commonly used edge detection techniques such as Canny, Prewitt, Fuzzy Logic, Roberts and Sobel [24].

2.2 Brief Description

The Sobel Operator is used for edge detection in image processing and computer vision. It creates an image emphasizing edges. Technically it can be described as a discrete differentiation operator that can compute an approximation of the gradient of the image intensity function. At each point in the image, the result of the Sobel operator is the corresponding gradient vector. The Sobel operator is used in convolving the image with a small, separable, and integer-valued filter in the horizontal and vertical directions and it makes the computations relatively inexpensive. Also the gradient approximation that it produces is relatively crude, in particular for high-frequency variations in the image [25].

2.3 How it works

At first the image is processed in the X and Y direction separately. Then the results are combined together to form a new image which represents the sum of the X and Y edges of the image. While using a Sobel Edge Detector, it is best to convert the image from an RGB scale to a Grayscale image at first. The next step is kernel convolution. A kernel is a 3 x 3 matrix consisting of asymmetrically or symmetrically weighted indexes. Two 3 x 3 kernels are used, one for X direction and another for Y direction. Because of the the kernel matrices shown below the gradient for X-direction has negative numbers on the left hand side and positive numbers on the right hand side. Some of the center pixels are preserved. Similarly, on the bottom the gradient for Y direction has negative numbers, positive numbers on top, and some on the middle row pixels are preserved.

When an image is scanned across the X direction, for example, the following X Direction Kernel is used to scan for large changes in the gradient [26].

X- Direction Kernel

-1	0	1
-2	0	2
-1	0	1

Similarly, when the image is scanned across the Y direction, the following Y Direction Kernel is used to scan for large gradients as well [26].

Y- Direction Kernel

-1	-2	-1
0	0	0
1	2	1

An image matrix can be considered for an example. If kernel convolution is applied, an edge between the column of 50 and 100 values will be found [26].

50	50	100	100
50	50	100	100
50	50	100	100
50	50	100	100

-1	0	1
-2	0	2
-1	0	1

-50 -100 -50 +100 +200 +100
200

Figure 2. 1: Kernel convolution

In the example shown above an X Direction Kernel is used for convolution. The image is processed from left to right and the example above shows the calculation of the (2, 2) point in the image matrix. A value of 200 is calculated and therefore a fairly prominent edge is found at this point. The bigger the value at the end the more noticeable the edge will be. If the edge needs to be exaggerated, then the filter values of -2 and 2 must be changed to higher magnitude, perhaps -5 and 5. This will make the gradient of the edge larger and therefore, more noticeable.

Once the image is processed in the X direction, then the image can be processed in the Y direction. Then to produce final image magnitudes of both the X and Y kernels will be added together showing all edges in the image [26]. This is described in the next section.

2.4 Sobel Algorithm

The Sobel filter consists of two 3 x 3 kernels. One is used for changes in the horizontal direction and another is used for changes in the vertical direction. Both kernels are convolved with the original image to calculate the approximations of the derivatives. Here G_x is the image containing the horizontal derivative approximation and G_y is the image that contains the vertical derivative approximations. So the computations are

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * A$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * A$$

Here A is the original source image. At each pixel in the image the gradient approximations denoted by G_x and G_y are combined together to calculate the gradient magnitude using

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

The gradient’s direction is calculated using

$$\Theta = arctan\left(\frac{G_y}{G_x}\right)$$

A Θ value of 0 will indicate a vertical edge that is darker on the left side.

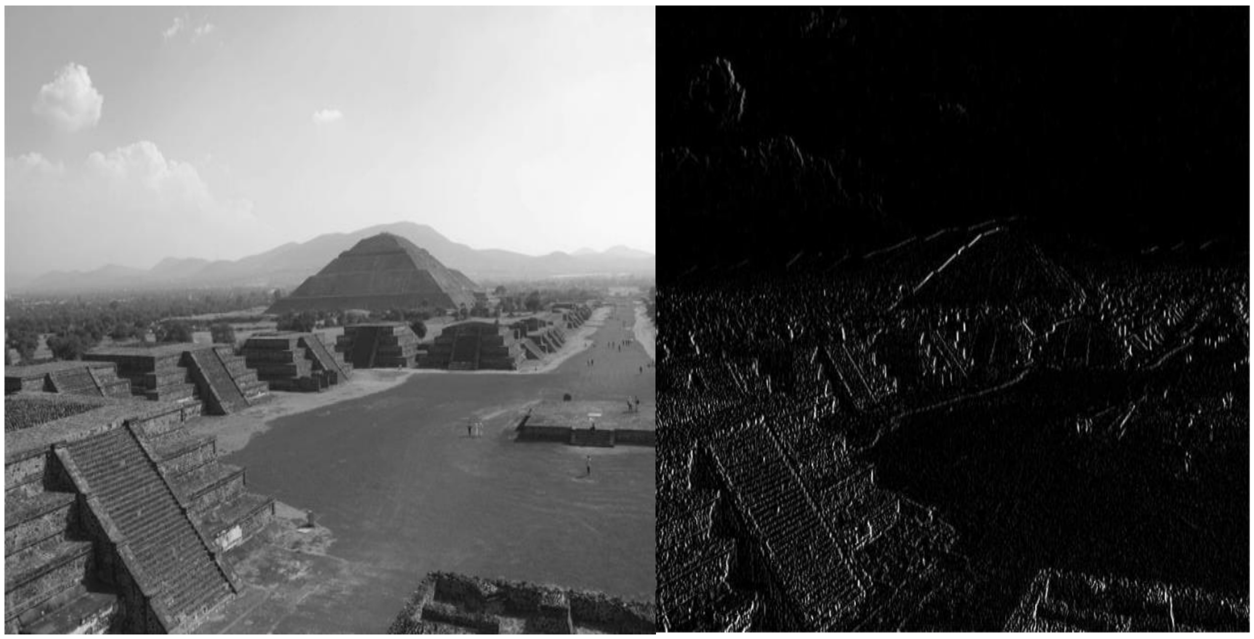


Figure 2. 2: Kernel convolution in horizontal direction

In Figure 2.2 on the left is the original image and on the right is G_x.

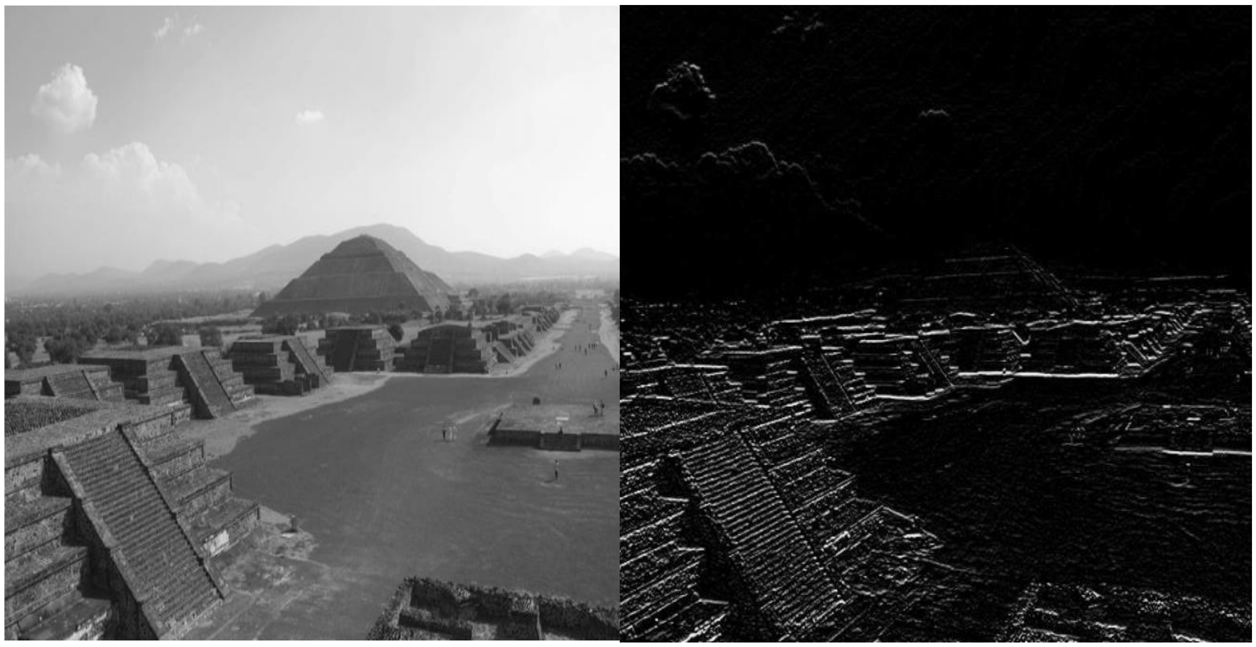


Figure 2. 3: Kernel Convolution in vertical direction

In Figure 2.3 on the left is the original image and on the right is G_y.

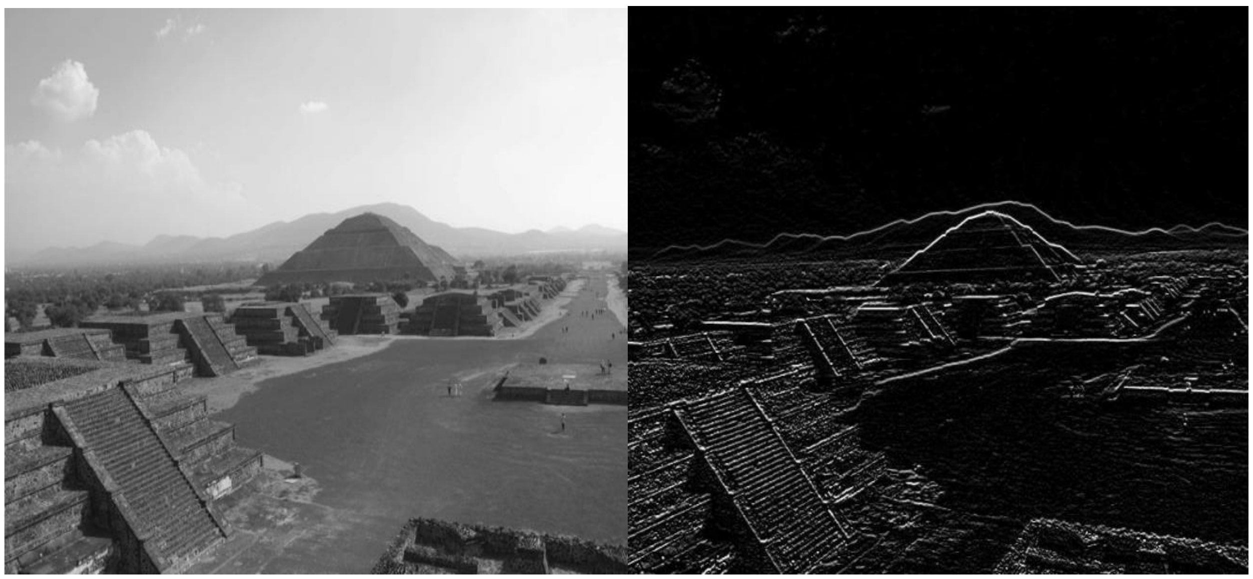


Figure 2. 4: Combined output

The image to the right in figure 2.4 is the result of combining the G_x and G_y derivative approximations calculated from image A on the left [27].

2.5 Design flow

The next task is to implement the Sobel algorithm using the Zynq platform. The entire process consists of the following steps.

● The first two steps of the Sobel algorithm (Converting RGB to Grayscale and Jpg to binary) are implemented in MATLAB. The binary file is required for the operation of programmable logic.

● Reading the input from the memory.

● Processing the image on the Programmable Logic side using an image processing accelerator

● Saving the output data in the memory

● Generate image using data from memory using MATLAB

The high level architecture is shown in Figure 2.5. It contains processing systems and programmable logic units of Zedboard. There are Image processing Intellectual Property (IP) and Direct Memory Access (DMA) controller IP in the Programmable Logic (PL) side.

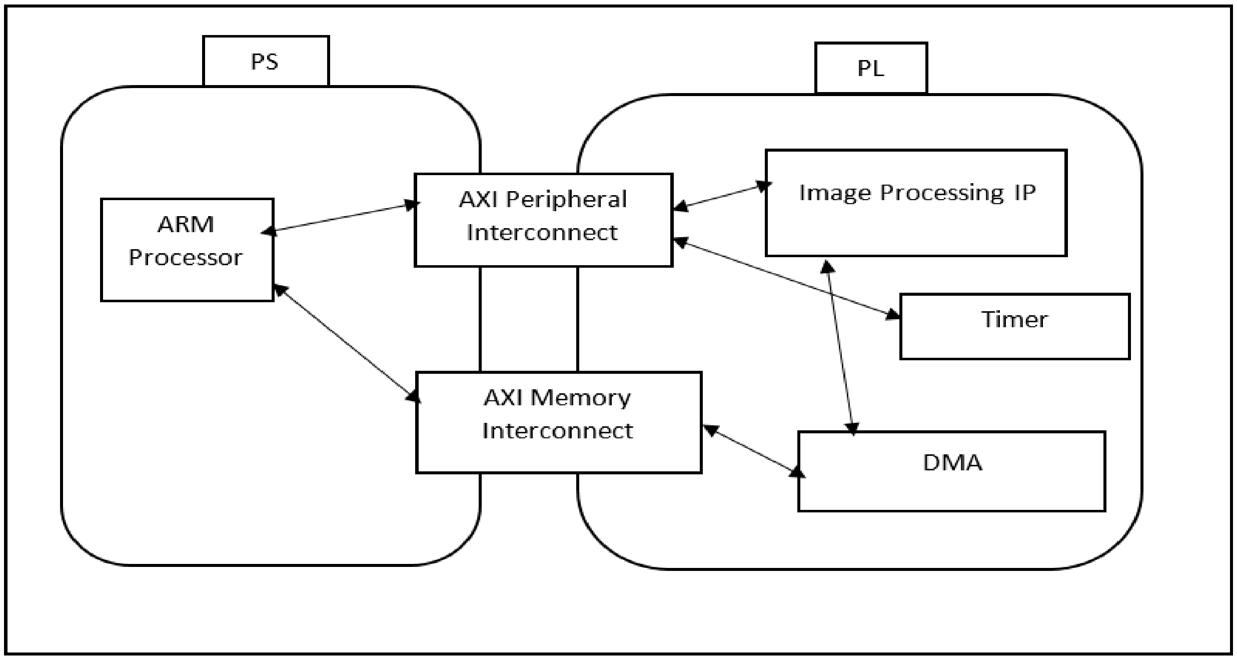


Figure 2. 5: High Level Architecture of the Zedboard

3. VIVADO HIGH LEVEL SYNTHESIS

3.1 Overview

The Xilinx Vivado High-Level Synthesis (HLS) is a very efficient tool to transform a C specification into a register transfer level (RTL) implementation that can be synthesized into a field programmable gate array (FPGA). C specifications can be written in C, C++ and the FPGA provides a massively parallel architecture with specific advantages in performance, cost, and power consumption over other computational environments such as traditional processors [28].

There are several benefits of High Level synthesis.

● Hardware designers are more productive, they can work at a higher level of abstraction while building high performance hardware.

● Algorithms can be developed using C, which consumes less development time.

● Since the verification can be done at C level, the functional correctness of the design can be validated more quickly than other traditional hardware description languages.

● C synthesis process can be controlled through optimization directives so the specific high performance hardware implementations can be created.

● From the C source code multiple implementations can be created using optimization directives.

3.2 Vivado HLS Design flow

Using the Vivado HLS tool a C function is synthesized into an IP block that can be integrated into a hardware system. The rest of the Xilinx design tools is tightly integrated with it and provide comprehensive language support and features for creating the optimal implementation for the C algorithm.

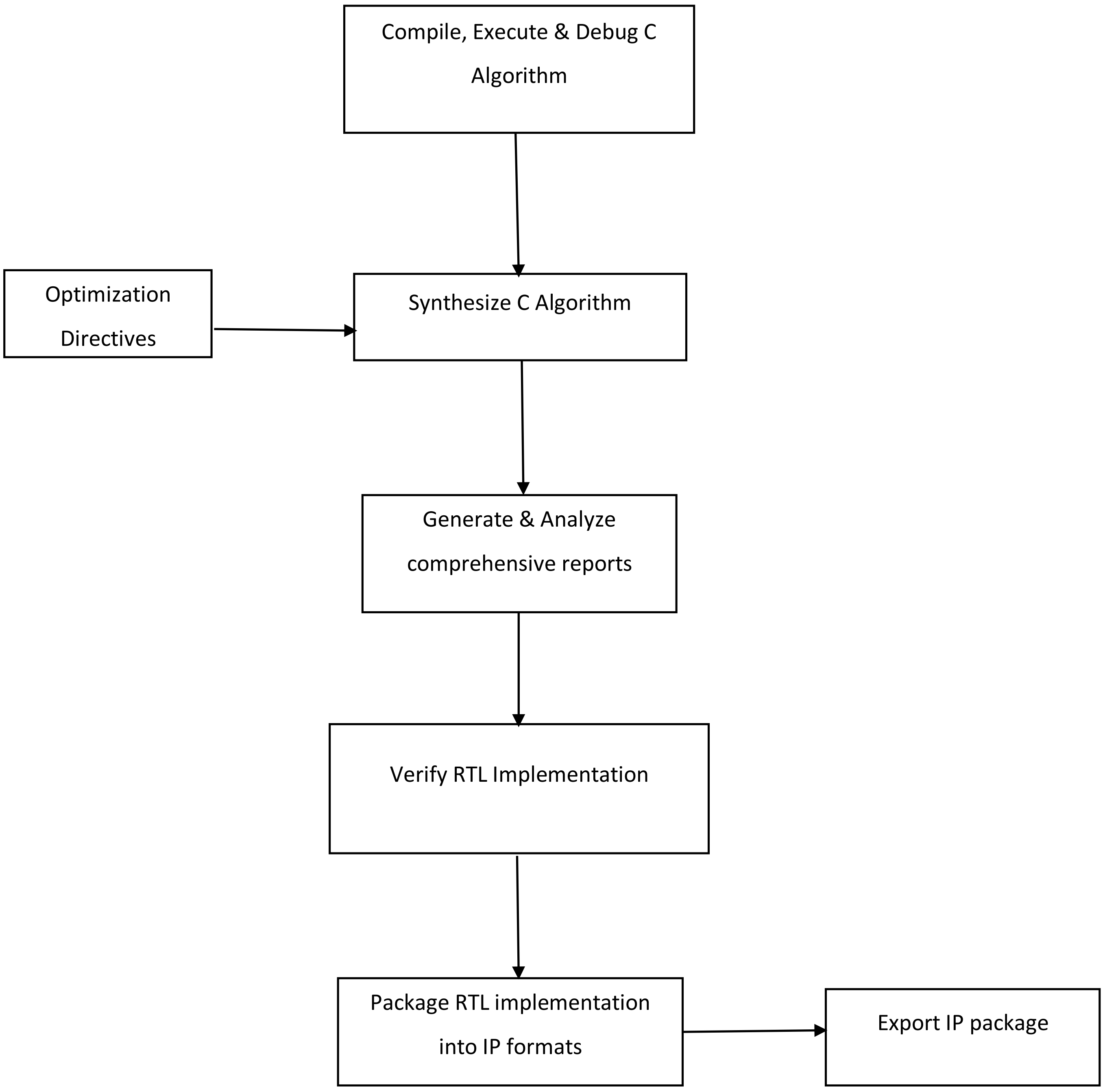

Compile, Execute & Debug C Algorithm
Optimization Directives
Synthesize C Algorithm
Generate & Analyze comprehensive reports
Verify RTL Implementation
Package RTL implementation into IP formats
Export IP package

Figure 3. 1: Vivado HLS Design Flow

The Vivado HLS Design Flow is shown in the Figure 3.1. In the first step the algorithm is required to be compiled and debugged in C. In the next step the C algorithm is synthesized using optimization directives. Here one top-level function is defined that can be synthesized and all other sub functions in its hierarchy are synthesized.

This analysis produces a report which gives the estimation about performance, utilization, and interface. So, the directives can be changed for further optimization of the implementation and different versions of the same algorithm can be made until the desired performance characteristics are obtained.

Different optimization directives include a directive to start executing a task in a pipeline or to set the latency for the functions and loops. Besides, the limits for the number of resources to be used are specified. The decision of input and output port behavior of the algorithm is taken by some of the directives.

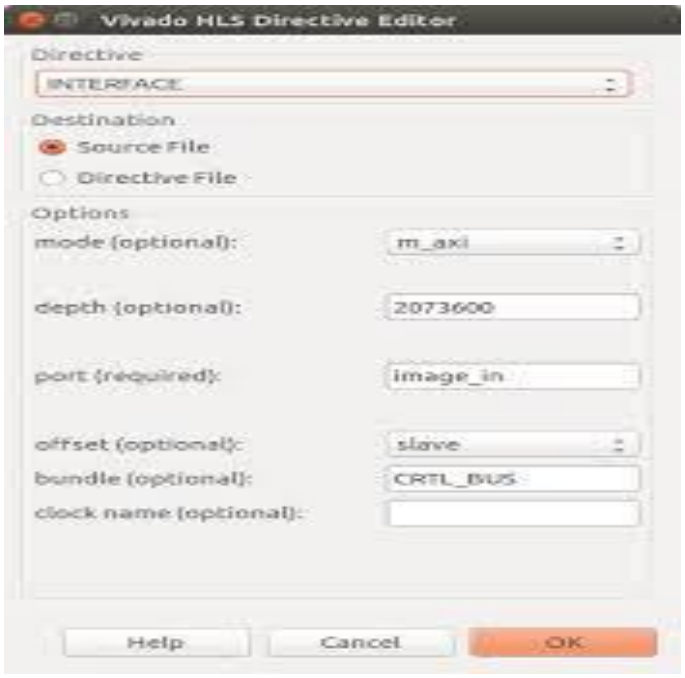
Now, in hardware description language RTL is implemented after the synthesis. For RTL implementation, the output is developed in either VHDL or Verilog. Later, C/RTL simulation verifies the RTL implementation and further results are compared with the C simulation. After that, this RTL implementation result is packed into IP and can be utilized by different tools in the design flow.

Next, we discuss the interface implementation, verification, optimization of the design and HLS library.

3.3 AXI4 Interfaces

AXI (Advanced eXtensible Interface) is part of ARM's AMBA (Advanced Microcontroller Bus Architecture), a family of micro controller buses first introduced in 1996. AXI4 is used for high-performance memory-mapped requirements [29].

AXI4 interfaces can introduce AXI4 stream (axis), AXI-4 master (m_axi) and AXI4-Lite (s_axilite) which is supported by Vivado HLS [28]. Different pragmas are required to be used to declare interface directives. Figure 3.2 shows how the directives can also be declared by Vivado HLS directive editor.


Vivado HLS Directive Editor

Directive: INTERFACE

Destination: [✓ Source File] Directive File

Options

mode (optional): m_axi

depth (optional): 2073600

port (required): image_in

offset (optional): slave

bundle (optional): CRTL_BUS

clock name (optional):

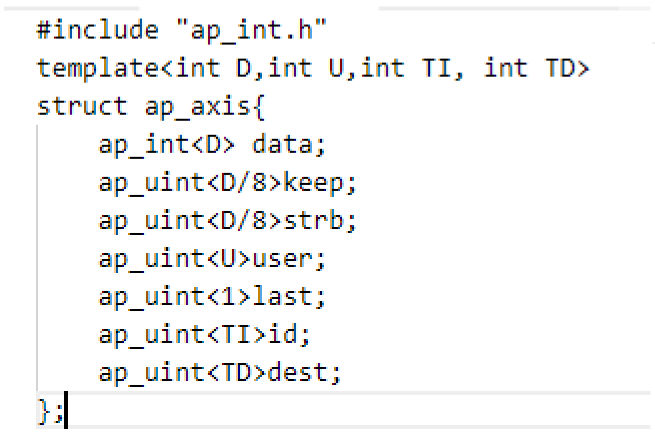
Figure 3. 2: Vivado HLS Directive editor

AXI4-Stream Interface: It can be defined on input or output argument including arrays or pointers. Data is transferred constantly through AXI4 Stream in a sequential manner and sign bit is prolonged repeatedly to the next byte. In addition, data transfer begins from first address and any address management is not necessary. It is really useful for burst data transfer which is good for image processing application. AXI4 stream interface can be classified into two basic types:

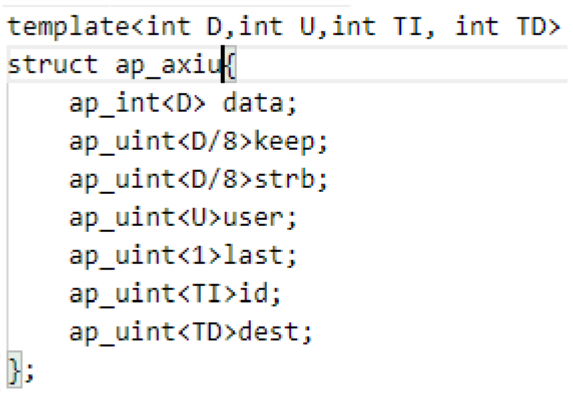
● AXI-4 stream interface without side-channel

● AXI-4 stream interface with side-channels

The side channel in the interface is assigned or controlled by the use of C++ code struct. Structs for the side channel is included in ap_axi_sdata.h header file which is shown below. HLS provides hls::stream class for C++ reference argument to model stream interface.



```
#include "ap_int.h"
template<int D,int U,int TI, int TD>
struct ap_axis{
    ap_int<D> data;
    ap_uint<D/8>keep;
    ap_uint<D/8>strb;
    ap_uint<U>user;
    ap_uint<1>last;
    ap_uint<TI>id;
    ap_uint<TD>dest;
};
```



```
template<int D,int U,int TI, int TD>
struct ap_axiu{
    ap_int<D> data;
    ap_uint<D/8>keep;
    ap_uint<D/8>strb;
    ap_uint<U>user;
    ap_uint<1>last;
    ap_uint<TI>id;
    ap_uint<TD>dest;
};
```

After synthesis the data ports are implemented with all the side channel port, TVALID and TREADY protocol ports. To get access from the Zynq, TVALID and TREADY must be 1 [28]. The example shows 32 bit signed and unsigned AXI4 stream interface declaration and implementation. Here ap_axis refers signed integer and ap_axiu refers unsigned integer. In DMA operations, the side channel TLAST is effective.

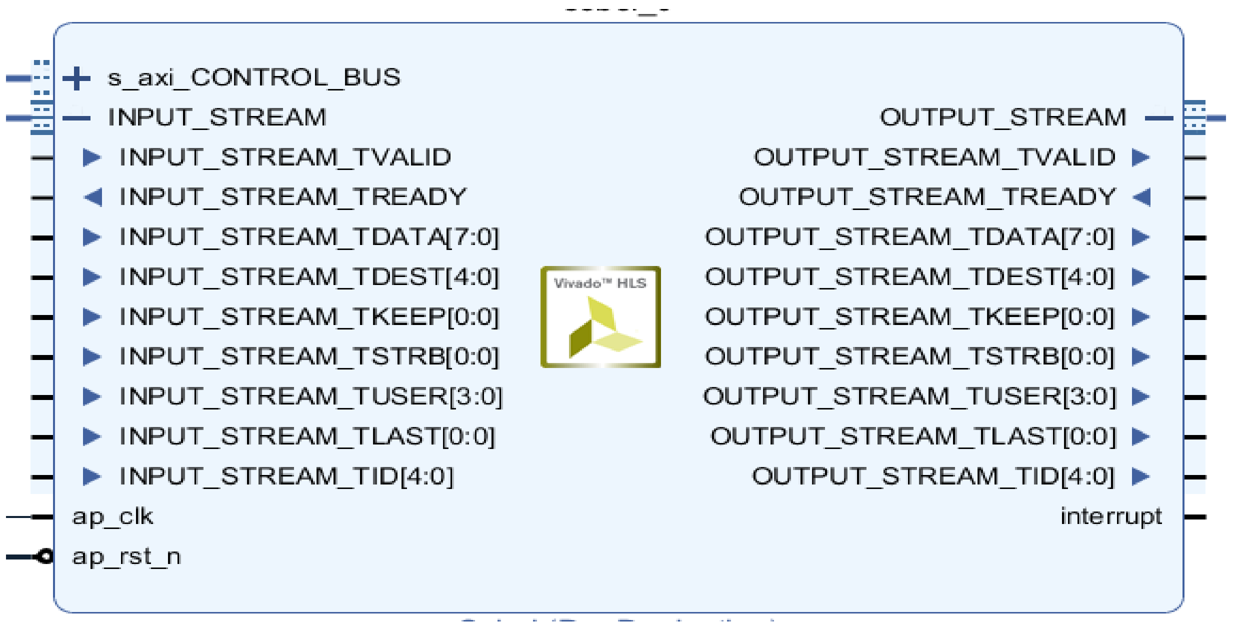


Figure 3. 3: AXI-Stream Interface Implementation

AXI4-Lite Interface: It can be set to any argument, but not to an array. Furthermore, multiple arguments can be clustered together. It is convenient when memory mapped single transfer of data is necessary**.** Burst data transfer is not supported by AXI Lite Interface since it does not support array parameters in the function. All the parameter results in HLS are reachable via axi4 lite interface and they are just some position in the memory. Vivado HLS implementation for multiple arguments is shown in the following example. Here return is used to show the function's return value and bundle is used to group all arguments, which are given with the same name.

```
#pragma HLS INTERFACE s_axilite port=port_name bundle=multiple argument
#pragma HLS INTERFACE s_axilite port=return bundle=multiple argument
```

After synthesis the following ports are generated by implemented AXI4 lite port [28].

- ➢ ap_done: set when function completes complete operation
- ➢ ap_ready: set when function is available to accept further data
- ➢ ap_data: data input or output arguments
- ➢ ap_clk: synchronous clock must be from same master clock
- ➢ ap_addr: specify the address of the interface
- ➢ ap_rst_n: use to reset the interface

Obtained C drivers are helpful to program the interface. Here ap_done and ap_ready ports means when the function accomplishes all of its operation and when the function is ready to accept new data.

AXI4 Master Interface: It specifies on pointers and array. Other arguments can be bundled together with this interface.

3.4 I/O Protocols

Ap_none plus ap_stable is used to indicate that I/O protocol is not required to be connected to the port.

3.5 Wire Handshakes

The command Ap_hs is used to generate a two-way handshake signal. To read or write sequential order this mode can be used with arrays.

3.6 Memory Interface

ap_memory interface is applied by default to implement array argument. They are customized to communicate with the RAMs and ROMs in terms of necessity to access the memory, which can be standard BRAM interface that has data, chip-enable, address, write-enable and address ports. The single port or dual port is defined by the RESOURCE directive. Implementation of ap_memory interface is shown in the following figure 3.4.

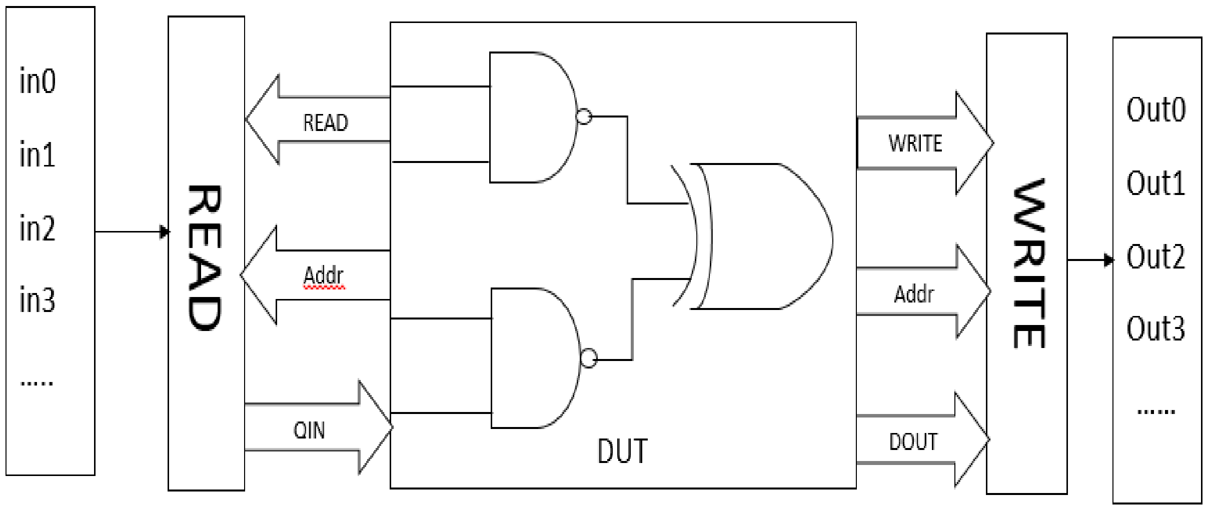


Figure 3. 4: RAM Interface

The ap_memory and bram interface are functionally similar but in ap_memory all interfaces are cited as detached ports. Whilst in bram- interface shows grouped single ports are ready to connect Xilinx BRAM with the single point-to-point connection. Ap_fifo interface is used to when data from the array in a sequential manner are needed to access. The port can be permitted to be connected with the FIFOs by Ap_fifp interface to have empty-full communication in both directions.

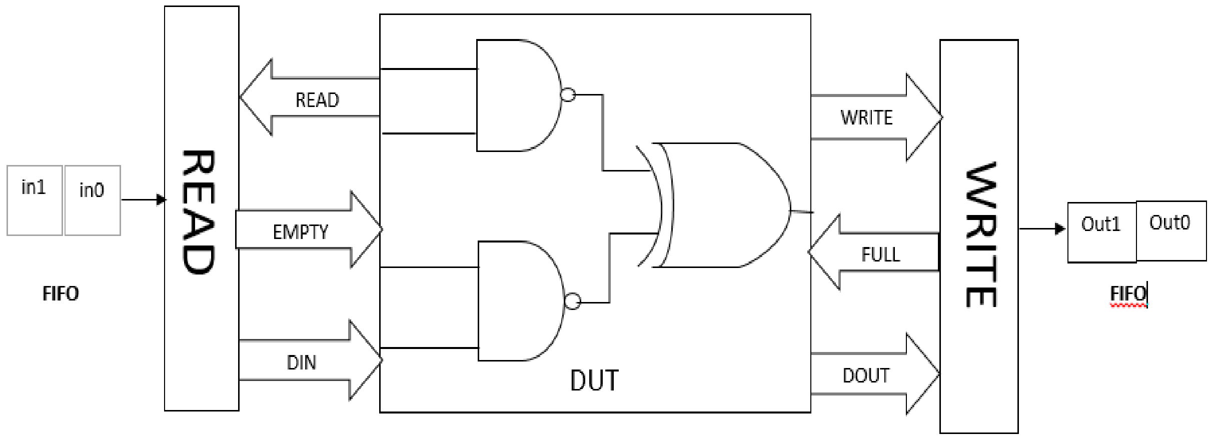


Figure 3. 5: FIFO Interface

3.7 Interface at block level

Ap_start, ap_done, ap_idle and ap_ready signals are applied to the block. The developed block can be controlled without any effect of input-output interface constraints by this signal.

ap_start: start the process on the data

ap_ready: high if implemented block is ready to accept new data

ap_ done: this signal indicates that the design finishes operation on data

ap_idle: implemented block is idle

3.8 Interface (Clock and Reset)

All the designs will be operated by only one and the same clock. There must be a constant uncertainty in the clock period given by the HLS tool. As a result, it is required to deduct clock uncertainty every time from the estimated clock period. The uncertainty for the clock period can be clearly declared but by default it takes 12.5 percent. Ap_rst_n is a port which puts FPGA registers and BRAM in an initial reset condition. There are three reset modes which can be added as an optimization directives:

None: no reset

Control: reset all the control register

State: reset all the control register, besides, it reset memories. Here all global and static variables are set back to their initial values.

All: reset is done for all the register and all the memories.

3.9 Design Optimization

There are optimization pragmas to obtain the important goals and for better performance. Pragma is special purpose directive that is used to turn on and off some features. Those pragmas can be included in the design to force Vivado HLS tool to generate the design as per given specifications. In this portion, we explain some of the optimization techniques.

Throughput Optimization:

Pipelining: As the whole system operates concurrently, the next step can begin its execution before the completion of the previous task. The pipeline directive can be applied to the loops and functions. Figure 3.6 describes how pipeline improves the throughput in the design.

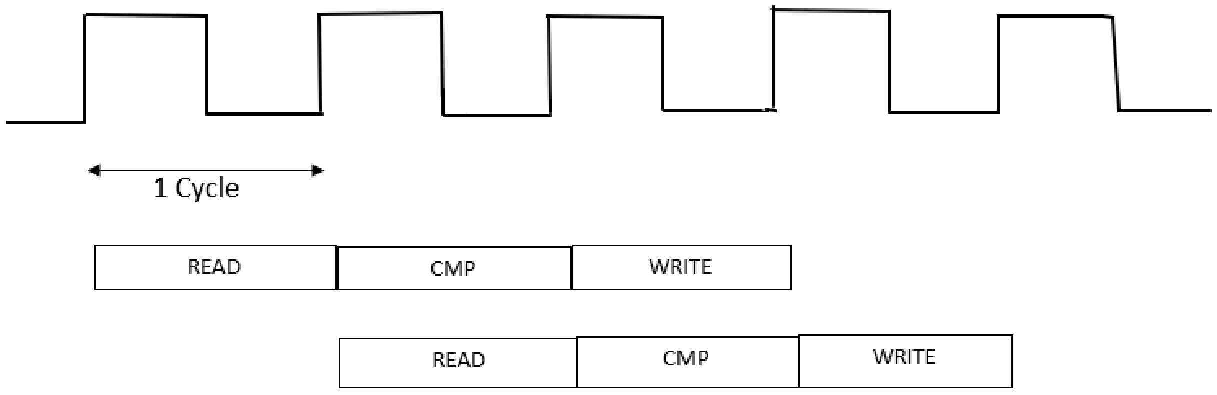


Figure 3. 6: Function without Pipelining

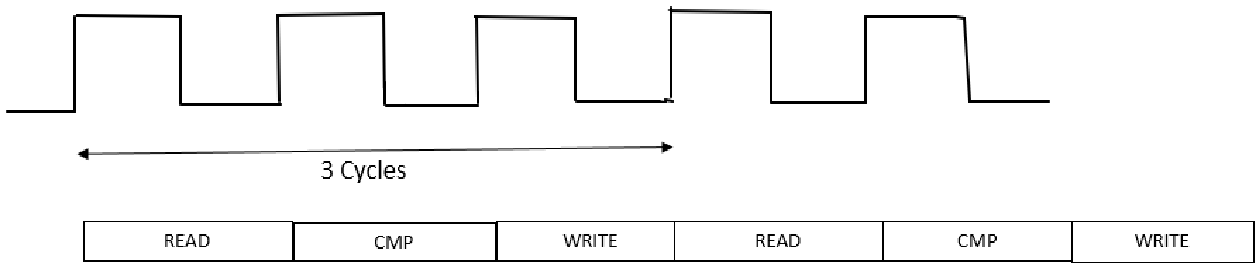


Figure 3. 7: Function with Pipelining

The function takes 3 clock cycles without pipeline till next read and takes 2 clock cycles to develop an output. Now, pipelined function can read every clock cycle with same latency and same resources. Now in order to implement in a simultaneous manner, pipelining in the loop demands the operations within the loop. Pipelining in the loop is described for the given 'for' loop.

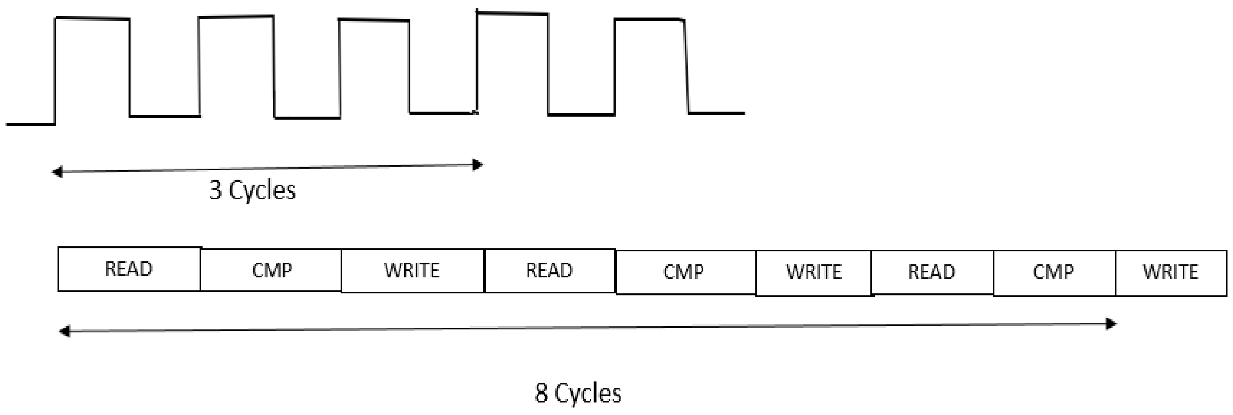


Figure 3. 8: Loop without Pipelining

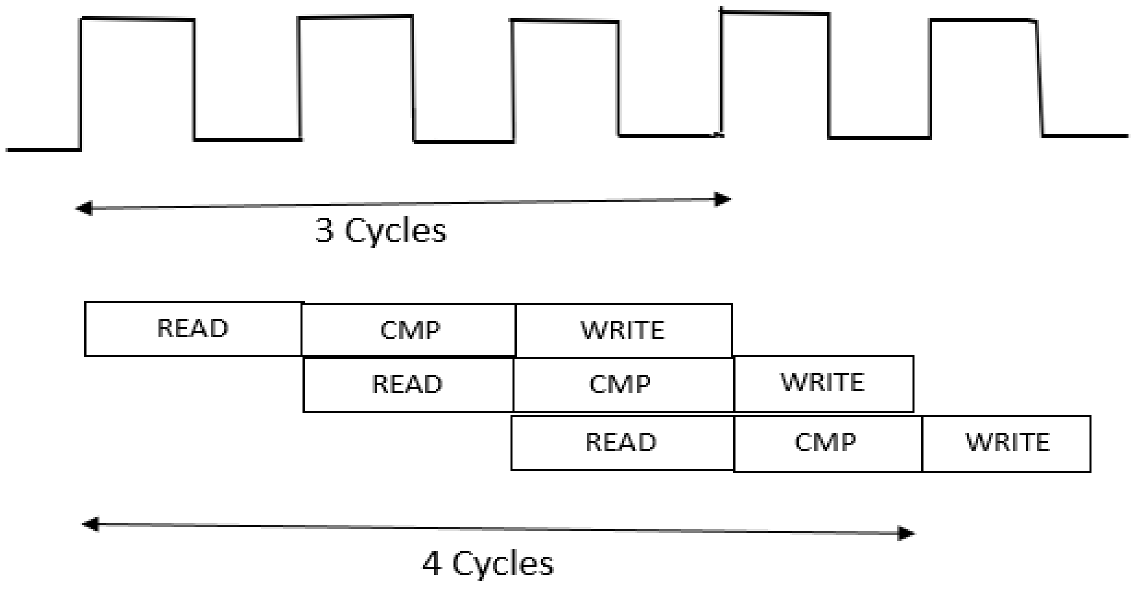


Figure 3. 9: Loop with Pipelining

In Figure 3.9 it is shown that loop with pipeline has latency of 4 clock cycles less than the latency of the loop without pipeline. The major dissimilarities between pipelined functions and pipelined loops is that, in functions a pipeline never finishes, it runs forever, while in loop it operates only for the loop iteration.

Loop Unrolling: UNROLL directive [28] is used to partially or fully unroll the loops. All the loops are rolled by default, which implies that the same hardware systems used for the operations in the loop.

Rolled Loop: It requires different clock cycles to complete each iteration in the loop which needs a multiplier and single port BRAM.

Partially Unrolled Loop: Partially unrolled loop by the factor of 2 consists of 2 multiplier and dual port BRAM that performs read and write in single clock cycle. Latency of the partially unrolled loop is half of rolled loop.

Unrolled Loop: One operation needs only one clock cycle to finish. In addition, it uses very few hardware resources.

UNROLL directive can be supplied to the loops and it can be applied to the functions. All the operation will be in parallel in completely unrolled loop that depends on data dependency.

Latency Optimization:

Latency pragma is used for assuring that implemented design executes all the operations in the functions in given range of clock cycles. It gives particular latency for a single repetition of the loop if the latency pragma is given inside the loop and if the aim is to operate the total latency of the loop then this pragma should be announced outside the loop.

For all the iterations separating latency is declared as:

Loop A: for (i=0;i<n;i++)

#pragma HLS latency max=10

Declaration of latency for all iterations:

#pragma HLS latency max=10

Loop A: for (i=0;i<n;i++)

There is another option to reduce latency by merging sequential loops for further optimization. In additional latency can also be improved by flattening the nested loop. For flattening loop directive should be given to the loop which is innermost of the loop body. It can be defined as:
Set_directive_loop_flatten top/inner

Area Optimization:

It is crucial to use proper precision data types for better area optimization for the variable because use of improper bit-width can result in slower hardware implementation, and in addition increased in latency. Arrays are also implemented as RAM or registers so out of bound element may increase the hardware resources.

For better optimization the INLINE directive can be used to share components by calling another function within the function. Small arrays can be mapped into one large array by using ARRAY_MAP pragma. The number of block RAM required for the hardware implementation can be lowered down by using this technique. To limit the number of operators by forcing the synthesis tool in Vivado HLS to share the operators.

3.10 C Libraries

HLS provides several libraries such as:

● Video

● Math stream

● Precision Date Types math

● Stream

● Linear Algebra

● DSP

In our work we use the math, video, and stream libraries. Next, we introduce these libraries.

hls stream: First, streaming data types do not have any address management. Furthermore, read and write are executed sequentially. To design stream data structure, hls::stream<> class is given as a C++ class. As FIFO interface with the depth of 1 and optimization directive is applied, streaming structure is implemented without any declaration. To adjust its depth value hls_stream.h header file is used to use this stream class. So blocking and non-blocking read and write methods can be obtained. Reading FIFO is allowed by the Non-blocking methods even if it is empty [28].

hls math: This library is very efficient for the synthesis of C and C++ math library functions that include floating point operations. During synthesis Hls_math.h is used. Instead of using the C math library, using vivado hls math library gives accurate C and C/RTL simulation result. Mathematical operations can be performed on the float and double data types, yet, it gives unreliable output but fast hardware (RTL) implementation.

hls video library: In order to include all videos and image functions (hls_video.h), the header file is used. Memory line buffer and window buffer is applied for the implementation of the edge detection algorithm. In additional, to test the algorithm OpenCV functions are used.

Line Buffer:

For the instantiation of line buffers this class is convenient. In this class, all the operations on the line buffer are defined as methods.

● Total number of rows and columns can be defined by the users in the line buffer

● Using methods in this class make debugging and implementation of line buffer easy.

● Data types can be expressed in parameters.

All the methods of line buffer class are explained with example as in below. This figure describes the line buffer along with initial data.

Table 3. 1: Line Buffer with initial position of data

Row	Column0	Column1	Column2	Column3	Column4
Row0	1	2	3	4	5
Row1	6	7	8	9	10
Row2	11	12	13	14	15

Line Buffer data type is practiced as explained in the example to instantiate the line buffer.

hls:: LineBuffer<rows,columns,type >variable;

hls:: LineBuffer<3,4,char >Buffer_LINE

In the line buffer data is coordinated in raster scan method. Every time various column number is used to sum with new data. For entering new data on the top or bottom of the column vertical shift shift_pixels_down [28] is helpful. To enter data insert_top_row [28] is used on the top of that column. The used example is to add 20 on the top of the column 1.

Buffer_LINE.shift_pixel_down(1); [28]// is used for vertical shift of column 1

Buffer_LINE.insert_top_row(20,1); [28]//insert data into column 1 and on the top.

Table 3. 2: Vertical shifting up the data and inserting data at the top

Row	Column0	Column1	Column2	Column3	Column4
Row0	1	20 ←	3	4	5
Row1	6	7	8	9	10
Row2	11	12	13	14	15

The following example displays how to enter the new value at the bottom of the line buffer at a specific column.

Here new data are added at the bottom of the column 0.

Buffer_LINE.shift_pixels_up(0); [28]// shift data up in column 0

Buffer_LINE.insert_bottom_row(20,0) [28]

Table 3. 3: Vertical Shifting down the data and inserting data at the bottom

Row	Column0	Column1	Column2	Column3	Column4
Row0	1	20	3	4	5
Row1	6	7	8	9	10
Row2	20 ←	12	13	14	15

To obtain value from any location of the line buffer method getval (row, column) is needed.

Window Buffer:

Using C++ memory window class two dimensional memory window is operated and announced This class has the similar specifications as Line Buffer class. Here are a few methods from this class is discussed through example.

Memory window is instantiated using hls::window<row,column,type>variable. Example: hls::window<3,4,char>Buffer_Window;//. This will develop 3x3 memory window buffer.

Table 3. 4: Initial Memory Buffer

Memory Window	C0	C1	C2
Row 0	1	2	3
Row 1	4	5	6
Row 2	7	8	9

In this case to shift up and shift down the row data shift_pixels_down() and shift_pixels_up() are used respectively. This table exhibits the memory window results just after the operation.
Buffer_Window.shift_pixels_down() [28]// is used to shift down the row and new data can be added to the first row.
Buffer_Window.shift_pixels_up() [28]// is used to shift up the row and new data can be added to the bottom row.

Table 3. 5: Operation of shifting up

Memory Window	C0	C1	C2
Row 0	1	2	3
Row 1	4	5	6
Row 2	new	new	new

Table 3. 6: Operation of shifting down

Memory Window	C0	C1	C2
Row 0	new	new	new
Row 1	4	5	6
Row 2	7	8	9

Memory window can be moved in left or right using shift_pixel_left() [28] and shift_data_left() [28] methods as above. Using insert_pixel(value,row,column) [28] method value can be inserted at any location of the window. Such as, Buffer_Window(10,2,2) will insert 10 at third row and third column.

Table 3. 7: Inserting new Value

Memory Window	C0	C1	C2
Row 0	1	2	3
Row 1	4	5	6
Row 2	7	8	10

Insert_row(), insert_col(), insert_left_col(), insert_right_col(), insert_top_row() and insert_bottom_row() are used to insert block in the window.

OpenCV Video Library Functions:

In Vivado HLS OpenCV functions are used and they can be implemented and synthesized. OpenCV interface functions are helpful in converting data to and from the OpenCV data types and AXI-4 stream data types. Video data declared as hls:: mat data types can be turned into AXI-4 stream data type using AXI-4 stream function. It is applied as the high-performance interface. Video processing functions can process and edit video images. These functions can be also synthesized.

3.11 Verification of RTL

For the verification of synthesized RTL output C/RTL simulation is used. Verification process is revealed in the figure.

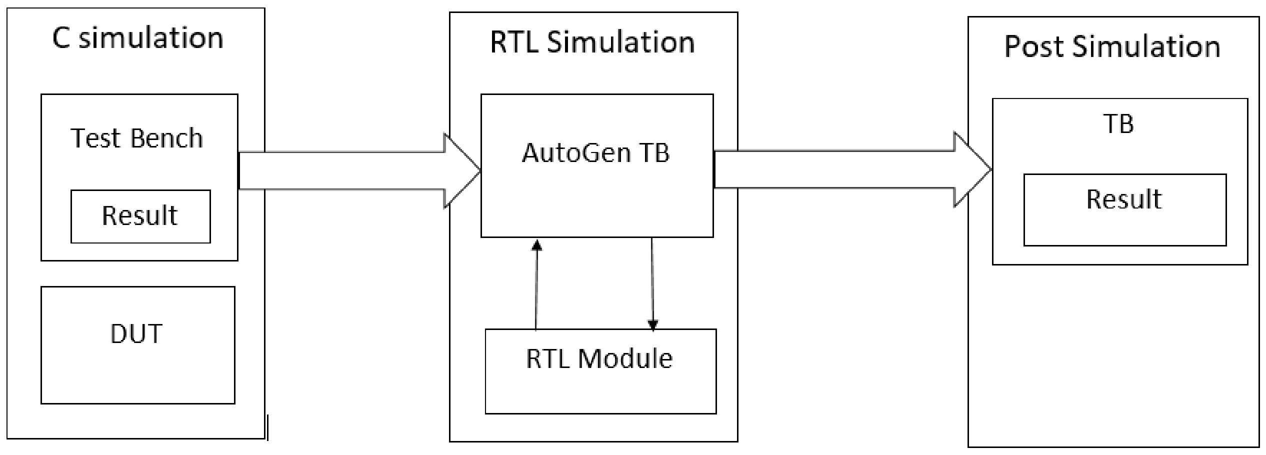


Figure 3. 10: Flow of RTL Verification

In figure 3.10, output from the C simulation is applied as input vectors in RTL simulation. After that input vectors are provided to the implemented RTL module. The output of RTL simulation is verified by the C test bench.

4. SYSTEM ON CHIP

4.1 Zynq7 Processing System

Dual-core ARM Cortex A9 [30] processor and Xilinx programmable logic are the constituents of Zynq7 processing system. It gives the benefit of high performance consuming very low power. While booting, at first the processor system is booted which is followed by the programmable logic booting. It can be configured totally, partially or dynamically. Zynq7 Soc unit is provided with different power domain. All the functional units of the Zynq processing system are explained in the following block diagram.

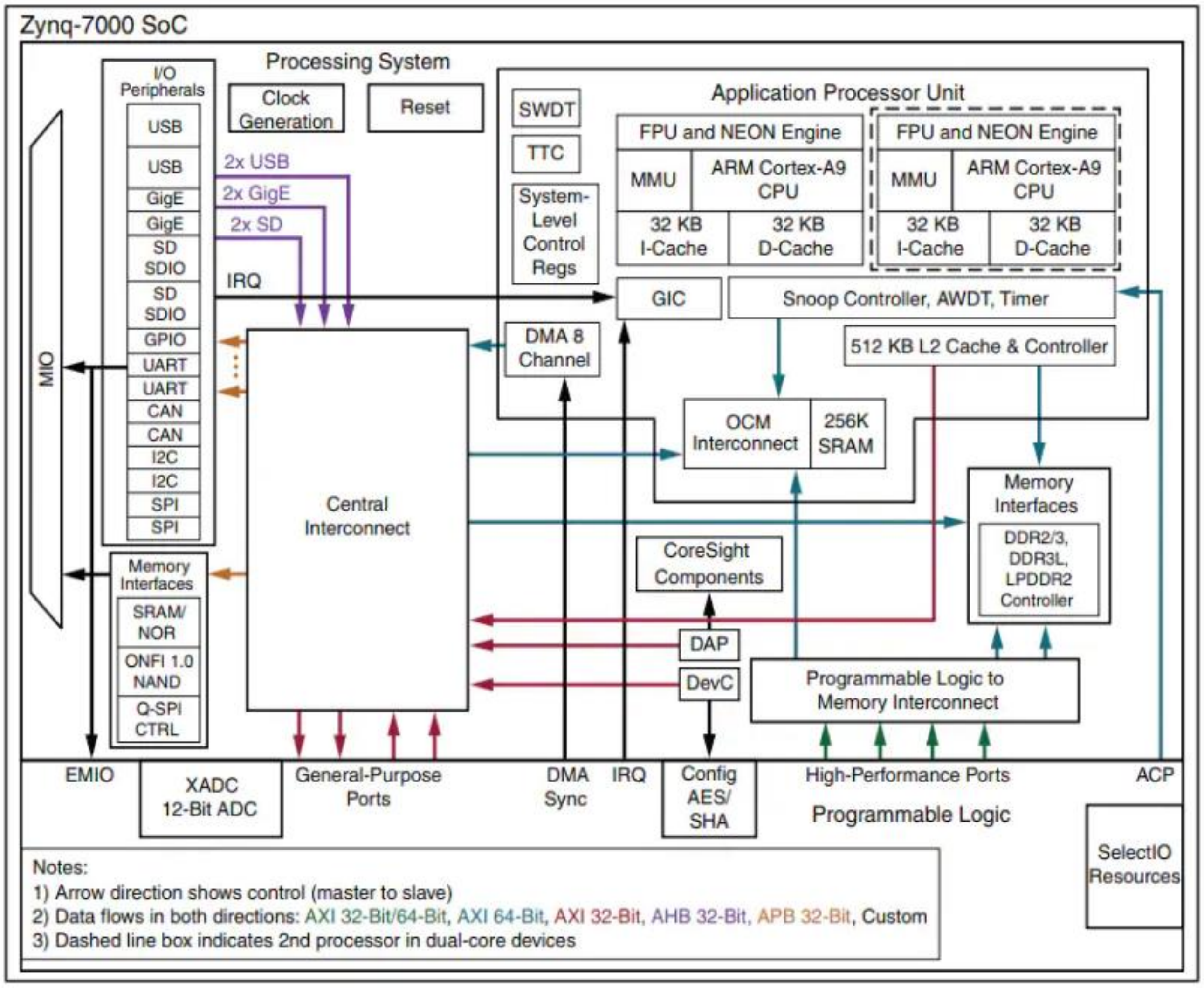


Figure 4. 1: Zynq Overview

PL and PS both have different power management unit which is displayed in the block diagram so the use of either one is possible for the power management.

- PS (Processing System) blocks
    - APU (Application Processor Unit)
    - Memory interface
    - Interconnect
    - IOP (Input/Output peripherals)
- PL (Programmable Logic).

Description of processing system is shown below.

1. Application Processor unit:
    - Two ARM cortex A9 processor 39
    - NEON 128b co-processor
    - Level 2 cache with parity -512 kb
    - Timers and watchdogs

SCU (Snoop Control Unit) is provided in order to maintain level 1 and level 2 coherency. In addition to this ACP (Accelerator Coherency Port) is given to slave PS to master programmable Logic unit. ACP port can also access L2 cache, On-Chip Memory, and 64b AXI slave [30]. 256kb of dual port On-Chip Memory is provided with parity support. For transferring data between any memories in the system, four channel DMA controller is supplied in the PS, and four-channel DMA controller is provided for data transfer to and from memory and PL [30].

2. Memory interface:
    - DDR controller
    - SMC (Static Memory Controller)
    - SPI-Quad Controller
    - Transaction Scheduler and DDR controller

3. Input output peripherals
    - GPIO
    - Two gigabit Ethernet controller

● USB 2.0

● Two SDIO/SD controller

● Master and Slave SPI controllers

● Two I2C controller

Description for Programmable Logic is described as follows:

● CLB-configurable logic blocks: look up table with 6 input, adders,

● Dual port block RAM: 36kb up to 72 bit wide

● DSP48E1 Digital signal Processing: high resolution 48 bit signal processor

● Clock management unit: buffers for high speed and low skew clock

● Configurable input outputs: lowest power and high speed input output

● High performance low power gigabit transceivers: transceivers with the speed up to 6.25 Gb/s.

● XADC- analog to digital converter: up to 17 analog input with on chip temperature and power supply sensor.

● PCI express

Clock and Reset system:

Dedicated 33.3333 MHZ clock is provided for the processor subsystem. Moreover 100 MHZ clock is provided for the PL part. Physically spread out frequency programmable clocks are provided for the PL part. Reset for the PS resets all the debugging sessions and configuration morovere system reset wipes out all the memory contents. System reset is entirely independent of the PL portion.

4.2 Timer

Cortex A9 Processor timer: It consists of the private 32-bit timer and 32-bit watchdog timer, and both processors use the simple 64-bit global timer. Both timers operate on half of the CPU frequency.

● Global Timer:

Global counter gives the frequency which is half of the clock frequency and it has auto increment of 64 bit.

● Private Timer:

Two modes are found for private mode. The first one is single shot and the next one is auto reload mode. Interruption is produced when 32-bit timer value comes to zero. The private timer can be configured for starting value.

● System watchdog timer:

Watchdog timer is used to handle signal catastrophic system failure. Input from PL bus clock or external or internal clock can be taken by watchdog timer. Besides it has 24 bit internal counter. At a reset, it can produce the interrupt or reset the system. 41

● Triple Timer Counters (TTC):

Independent timer counter has 16 bit up-down counter and 16 bit pre-scalar. Internal or external clock can be the input for this timers and all of them have individual interrupts.

4.3 UART

UART is a full duplex and UART controller supports large-scale baud rate and full duplex communication. For transmission and reception of the data, two 64 byte FIFO are used. Serialization and deserialization of the transmitter FIFO and receiver FIFO can be controlled by UART controller. Status register, interrupt status register, and modem status register are appointed to read states of FIFOs, modem signal and other controller function. Mode register and configuration register control the UART functions.

It is shown in the figure, UART controller and APU communicate using APB bus. Data arrived from memory is stored in TxFIFO and received data is stored in RxFIFO. It operates on 600,9600,28800,115200,460800,921600 baud rates and also produces this baud rates using UART reference clock. Data width for transmitter and receiver FIFO is 8 bit. It can run on one of the four mode, Normal mode, local loopback mode, automatic mode and remote loopback mode [30].

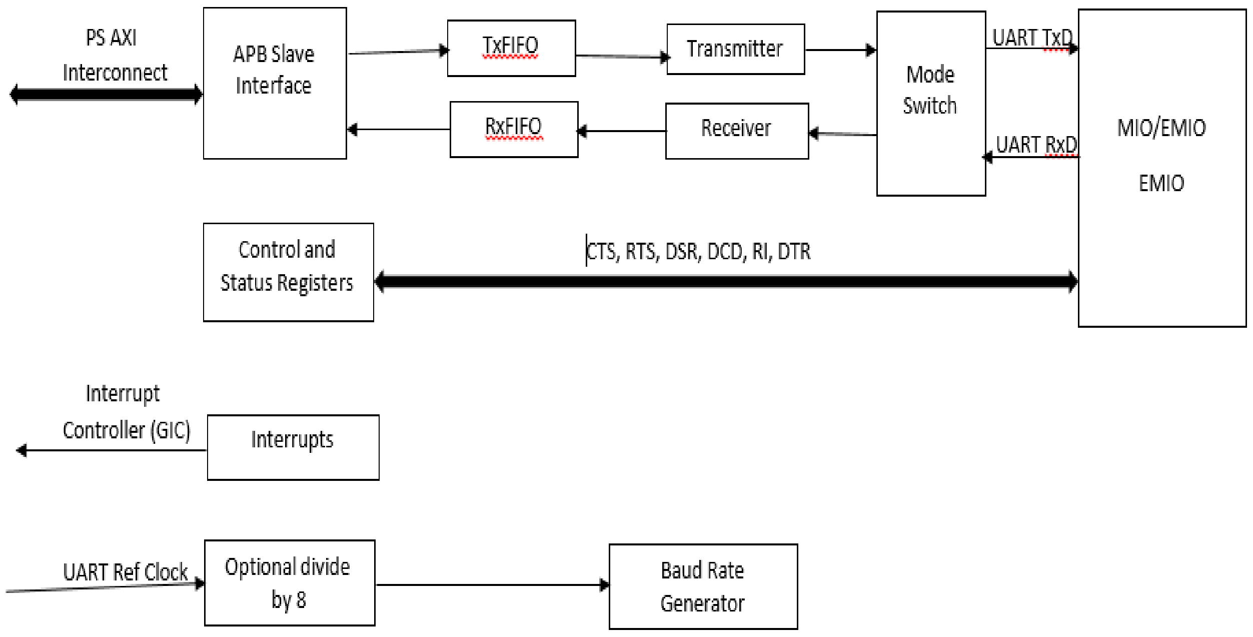


Figure 4. 2: UART Function Diagram

4.4 DMA Controller

DMA controller is used to transfer a large amount of data without any interference of the processor. Data transfer can be between anywhere system memories and PL peripherals [31]. It uses 64 bit AXI master interface with clock_2x frequency. DMA controller includes eight channels, which are all configurable. To push memory request DMA engine is used for reading or writing. All the status and control register are accessible through software. The figure 4.3 presents the block design of the DMA controller.

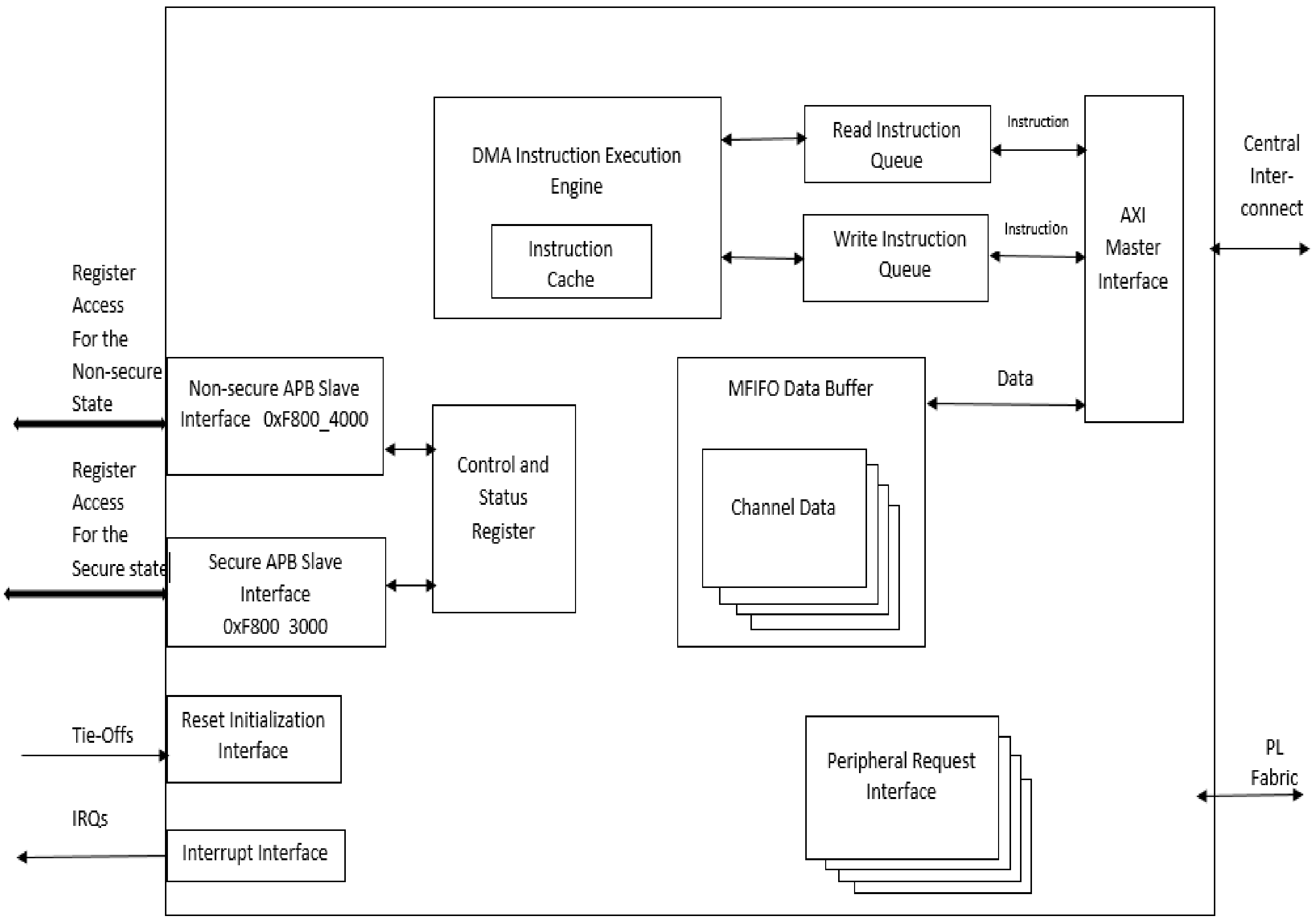

DMA Instruction Execution Engine
Instruction Cache
Read Instruction Queue
Write Instruction Queue
Instruction
Instruction
AXI Master Interface
Central Inter-connect
Register Access For the Non-secure State
Non-secure APB Slave Interface 0xF800_4000
Control and Status Register
MFIFO Data Buffer
Channel Data
Data
Register Access For the Secure state
Secure APB Slave Interface 0xF800_3000
Tie-Offs
Reset Initialization Interface
IRQs
Interrupt Interface
Peripheral Request Interface
PL Fabric

Figure 4. 3: Block Diagram of DMA Controller

DMA transmission execution engine controls DMA transmission along with processing the program code. Instruction is stored by instruction cache temporarily. Read and write instruction is used as a storage buffer for instruction before start any transmission on AXI and multi-channel data FIFO is used as the storage buffer for read and write during DMA transmission. DMA to PL peripheral interface supports asynchronous request from PL peripherals.

4.5 AXI Interface

Advanced Extensible Interface protocol is majorly used in Xilinx IPs. This protocol is a part of ARM advance microcontroller bus architecture (AMBA) [29]. AXI4 [29] is advanced version of AMBA

Overview of AXI: Information between AXI slave and AXI master peripherals is exchanged by AXI interface. A structure which is called AXI interconnect connects memory mapped AXI slave and AXI master blocks. Xilinx AXI interconnect includes five channels.

- Read address channel
- Read data channel
- Write data channel
- Write address channel
- Write response channel

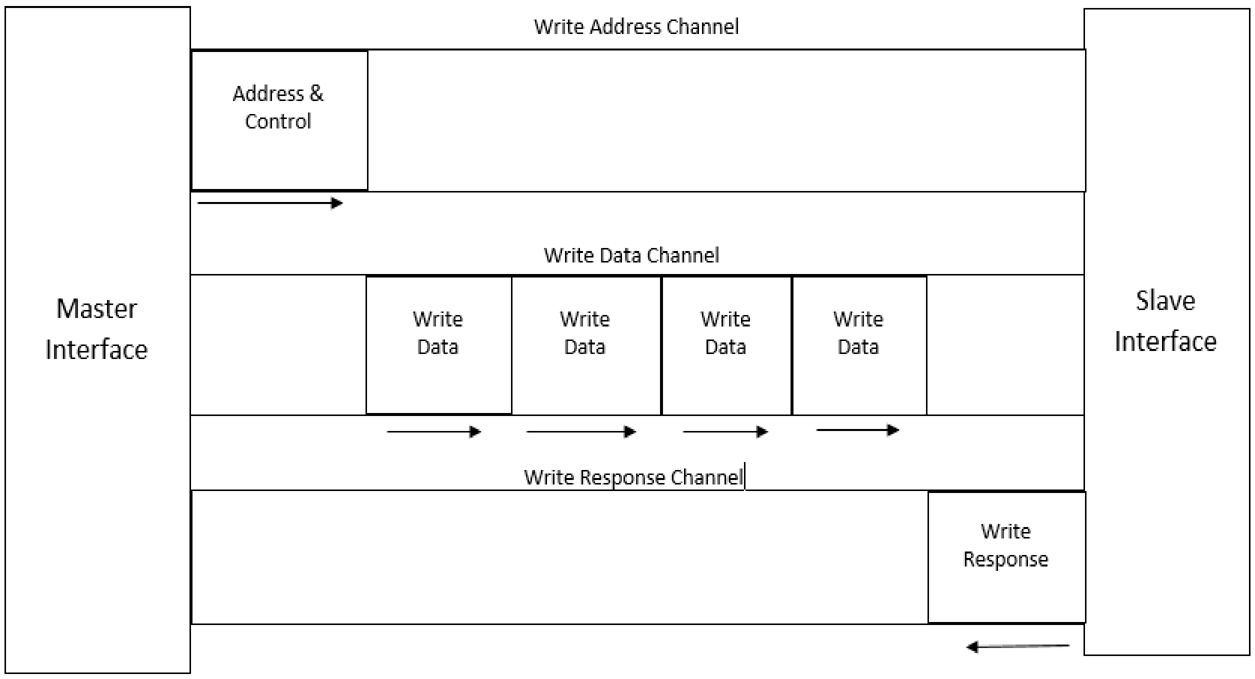


Figure 4. 4: The above figure is the architecture for Write Channel [13]

This interface runs bi-directionally and may have various data width. Yet it approves only 256 data transfer in a burst mode. AXI4 stream solely approves burst 44 data transfer while AXI4 lite can have merely more than one data per transaction. Due to separate dar and address connection, Bi-directional data transfer is obtainable. It supports various pipeline phases to maintain timing closure. Besides, both AXI master and AXI slave has dissimilar clock. There are three types of AXI4 interface which are AXI4, AXI4-lite and AXI4 stream.

4.6 Zed Board Hardware

There is abundance of board peripherals achievable on the Zed board. Some peripherals are available by processing system On the other hand, some of them are accessible purely for programmable logic. Oscillators which are used to generate clocks, UART, DDR peripherals are applied in this project. Developed bit stream is dumb into FPGA applying micro USB cable through JTAG.

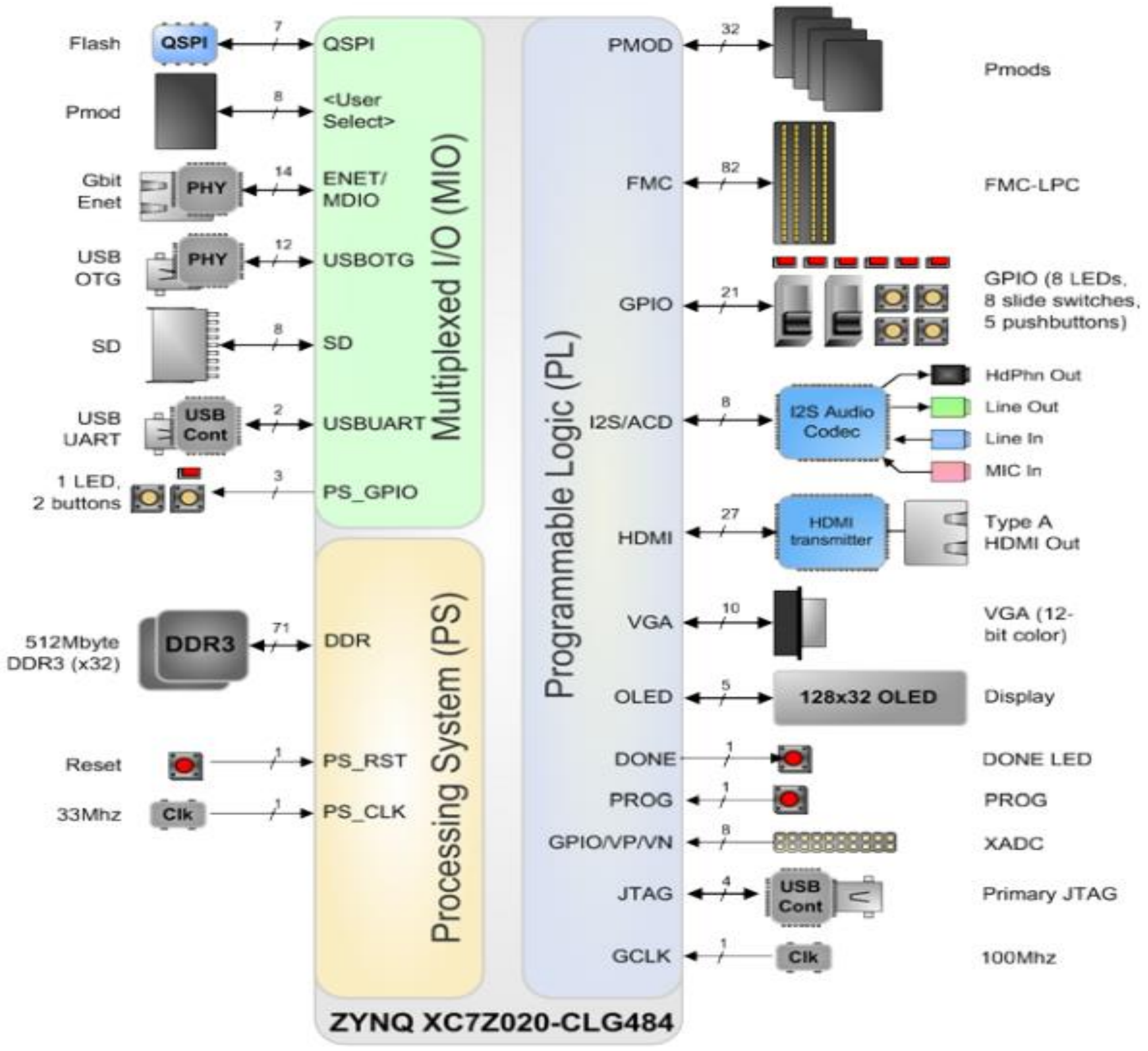


Figure 4. 5: Zed board Block Diagram is shown.

5. SOBEL FILTER IMPLEMENTATION IN ZYNQ-7000 SOC

5.1 Overview

In this chapter we discuss the Sobel filter implementation for both the software and hardware part . The main goal for this part of the project is to generate the noise-free image and feed it into the Sobel filter so that edges can be detected. Also performance of the edge detection using Zynq-7000 SoC will be compared with other platforms.

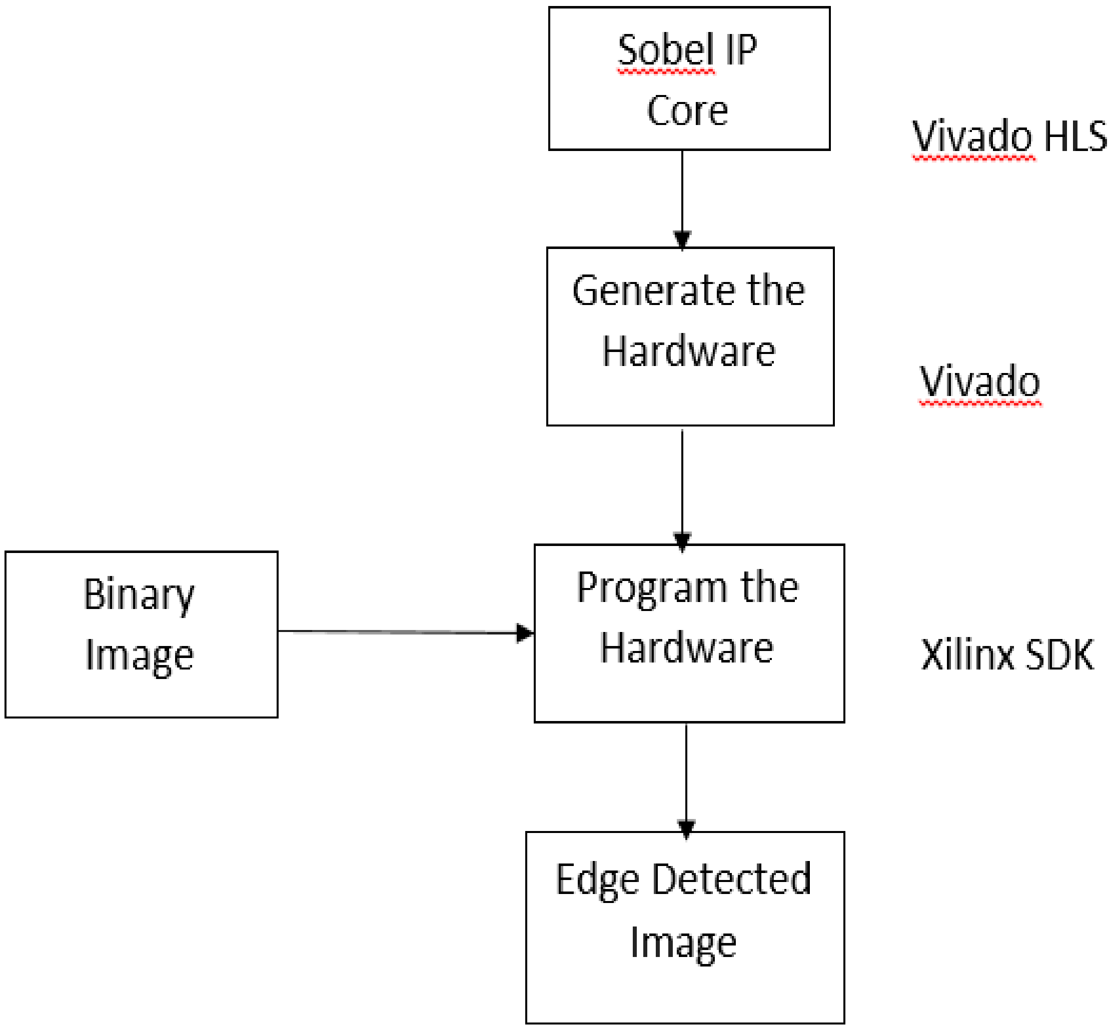
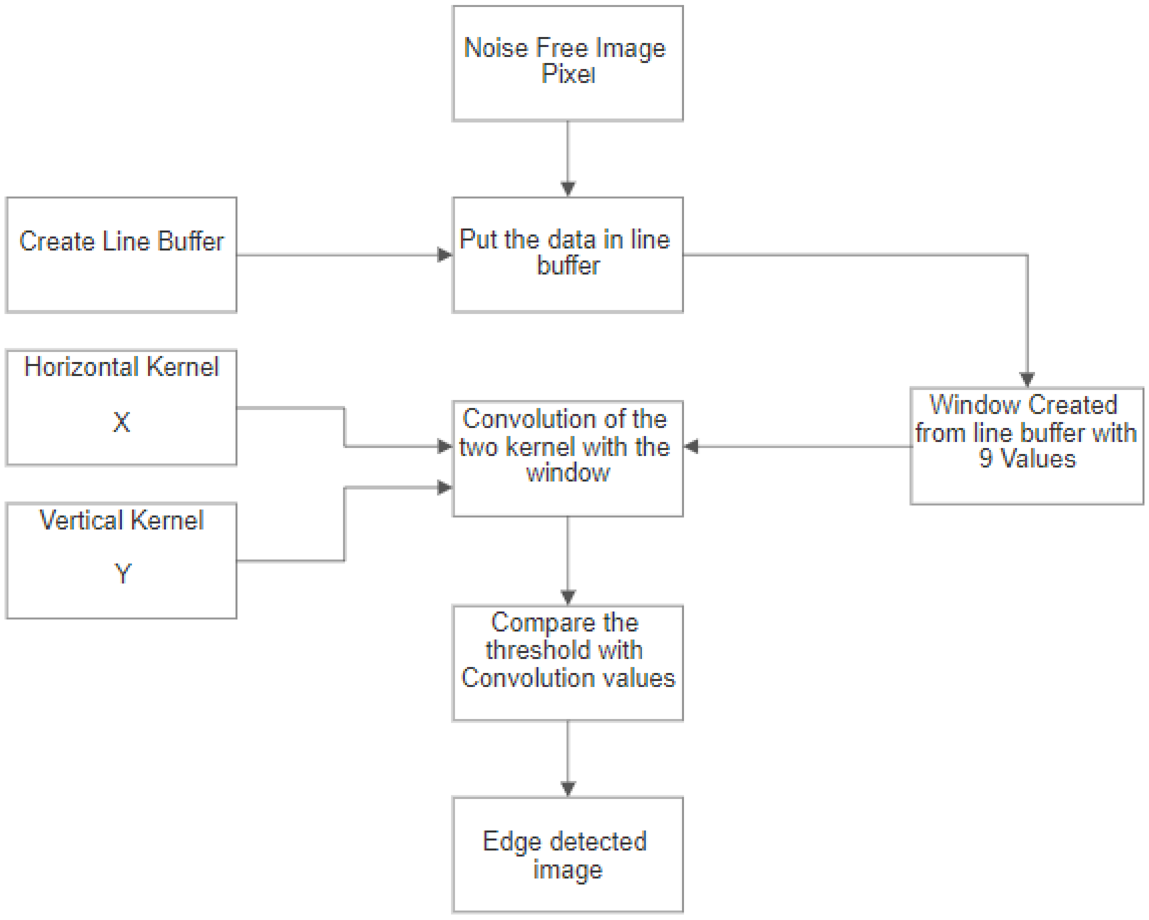


Figure 5. 1: Working Sequence for Sobel Implementation

From Figure 5.1, at first the IP core for Sobel Edge Detector will be implemented. To create the IP core Vivado HLS will be used. After that the task of hardware generation will be done where the newly created IP core will be placed. In the final part, the hardware will be programmed in the Xilinx SDK tool. Binary image will be used as an input for the hardware, which in turn will generate the edge detected image.

5.2 Generation of Sobel IP Core



Noise Free Image Pixel
Create Line Buffer
Put the data in line buffer
Horizontal Kernel X
Vertical Kernel Y
Convolution of the two kernel with the window
Window Created from line buffer with 9 Values
Compare the threshold with Convolution values
Edge detected image

Figure 5. 2: Working sequence of Sobel in Vivado HLS

As already discussed in Chapter 1, there are two kernels which will be used to perform convolution with the binary noise-free image. They are the Horizontal Kernel X and Vertical Kernel Y:

X Kernel (Horizontal):

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Y Kernel (Vertical):

$$\begin{bmatrix} -1 & 2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Noise-free image pixel and also the two kernels from the test bench will be sent to the main core function. In the main core function, the first aim is to create the line buffer which will store the values of the noise-free image pixel. The size of the line buffer is 3*240, 3 rows and 240 columns. The data is not be sent till the line buffer has 240 pixel values in the first row. The convolution cannot work at least until the first line of the line buffer has been filled. That is why it has to wait 241 pixel counts to start work. Once the line buffer fills up the convolution starts operating. Each pixel is convolved with X kernel and Y kernel, and the results are stored in appropriate locations. Then, the results from the two convolutions are added together and the result is transformed into an absolute value, which is subtracted from 255. The main part is to decide the threshold values which will decide the edge of the image. For the places where the image will be made black the threshold value is 110 and for the places where the image will be made white the threshold value is 25. This white color shows the edge of the image. These threshold values has been used as it detects almost all the edges for the image. To get the pixel value from the line buffer specific column and row value will be used to fetch the data and perform the convolution with the two different kernels. Once the values has been compared with the thresholds the data is sent to the output stream. Also when the row value and column value is greater than the kernel size then the pixel has to be shifted by 1.It is done to move the window. As discussed earlier in chapter 3 and also in chapter 4 that the window size should always be 3 because of the middle value can be compared to other values. That's why the data is fetched from the line buffer according to a size of 3*3. When the line buffer on a particular row ends, increment the row size to populate more values into the window and also the column position will be changed to get new values from the line buffer. When all the pixel values has been worked on, exit from the loop and put the remaining result back to the output stream. This is the basic working module of Sobel to detect the edge of the image.

First the design will be simulated and will be checked whether the desired output is coming or not.

5.3 Simulation Result

```
Compiling../../../test_core.cpp in debug mode
Compiling../../../core.cpp in debug mode
Generating csim.exe
Calling Core function
Core function ended
Saving image
@I [SIM-1] CSim done with 0 errors.
```

The simulation was successful as no errors were found.

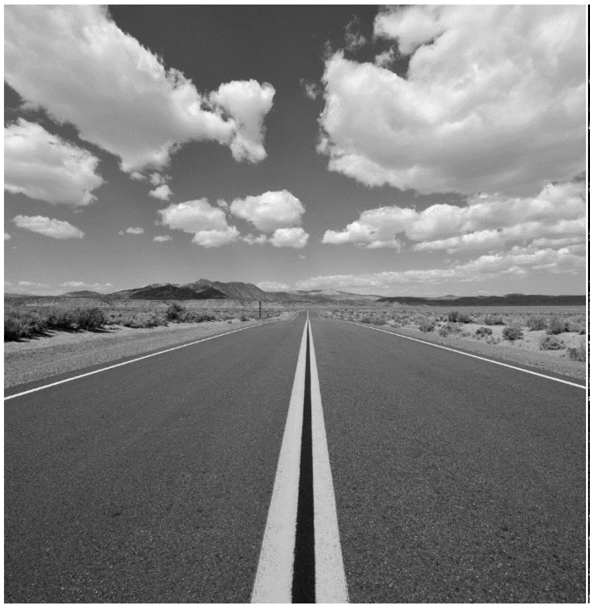


Figure 5. 3: Grayscale Image.

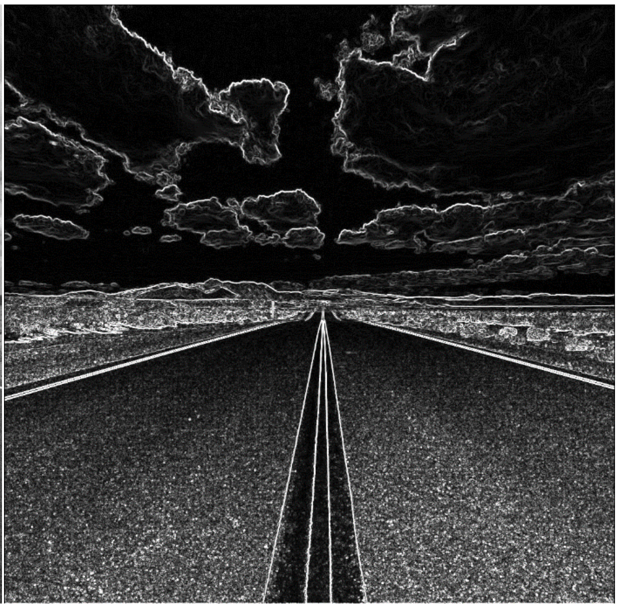


Figure 5. 4: Edge Detected Image

Figure 5.3 shows the grayscale image generated. Figure 5.4 shows the edge detected image of the noise free image which was generated from the Vivado HLS. The white lines shows the edge of the image. The core that has been generated will be used to make it an IP core which in turn will be used in the hardware.

Next step will be to synthesize the code. This will be used to make an IP core. After that the synthesis will be done the RTL will be exported to Vivado.

5.4 Synthesis of the Sobel Filter

Here from Figure 5.4 it can be noticed that it took around 11.36 ns for performing the synthesis. Synthesis is performed so that the RTL code (VHDL/Verilog) can be used in the hardware. Pipelining is used to make it work in less time. Latency is the time difference where the user gives the data as input and get the processed data as the output. Table 5.2 shows the latency which is 691449 for both minimum and maximum values. Also the interval is 691450.

Table 5. 1: Timing for the Sobel Filter

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	11.36	1.25

Table 5. 2: Latency of the Sobel filter

Latency		Interval		
max	min	max	min	Type
691449	691449	691450	691450	none

Table 5. 3: Utilization of the system

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	8	0	512
FIFO	-	-	-	-
Instance	4	10	226	220
Memory	3	-	0	0
Multiplexer	-	-	-	337
Register	-	-	947	-
Total	7	18	1173	1069
Available	280	220	106400	53200
Utilization (%)	2	8	1	2

Table 5.3 shows the DSP utilization is 8%, Flip Flop is 1% Look-Up Table is 2%. This table demonstrates the total utilization of the system.

Table 5. 4: Generated Ports for AXI Lite

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
s_axi_CRTL_BUS_AWVALID	in	1	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_AWREADY	out	1	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_AWADDR	in	5	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_WVALID	in	1	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_WREADY	out	1	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_WDATA	in	32	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_WSTRB	in	4	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_ARVALID	in	1	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_ARREADY	out	1	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_ARADDR	in	5	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_RVALID	out	1	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_RREADY	in	1	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_RDATA	out	32	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_RRESP	out	2	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_BVALID	out	1	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_BREADY	in	1	s_axi	CRTL_BUS	return void
s_axi_CRTL_BUS_BRESP	out	2	s_axi	CRTL_BUS	return void
s_axi_KERNEL_BUS_AWVALID	in	1	s_axi	KERNEL_BUS	array
s_axi_KERNEL_BUS_AWREADY	out	1	s_axi	KERNEL_BUS	array
s_axi_KERNEL_BUS_AWADDR	in	6	s_axi	KERNEL_BUS	array
s_axi_KERNEL_BUS_WVALID	in	1	s_axi	KERNEL_BUS	array
s_axi_KERNEL_BUS_WREADY	out	1	s_axi	KERNEL_BUS	array
s_axi_KERNEL_BUS_WDATA	in	32	s_axi	KERNEL_BUS	array
s_axi_KERNEL_BUS_WSTRB	in	4	s_axi	KERNEL_BUS	array
s_axi_KERNEL_BUS_ARVALID	in	1	s_axi	KERNEL_BUS	array

s_axi_KERNEL_BUS_ARREADY	out	1	s_axi	KERNEL_BUS	array
s_axi_KERNEL_BUS_ARADDR	in	6	s_axi	KERNEL_BUS	array
s_axi_KERNEL_BUS_RVALID	out	1	s_axi	KERNEL_BUS	array
s_axi_KERNEL_BUS_RREADY	in	1	s_axi	KERNEL_BUS	array
s_axi_KERNEL_BUS_RDATA	out	32	s_axi	KERNEL_BUS	array
s_axi_KERNEL_BUS_RRESP	out	2	s_axi	KERNEL_BUS	array
s_axi_KERNEL_BUS_BVALID	out	1	s_axi	KERNEL_BUS	array
s_axi_KERNEL_BUS_BREADY	in	1	s_axi	KERNEL_BUS	array
s_axi_KERNEL_BUS_BRESP	out	2	s_axi	KERNEL_BUS	array

From Table 5.5 it can be observed that AXI stream has been used for both input and output stream. AXI stream has been used as both in the input and output stream the data comes in a burst. Whereas for the two kernels and the return of the control bus AXI lite has been used because the data load is not that big.

Table 5. 5: Generated Ports for AXI Stream

ap_clk	in	1	ap_ctrl_hs	doSobel	return value
ap_rst_n	in	1	ap_ctrl_hs	doSobel	return value
interrupt	out	1	ap_ctrl_hs	doSobel	return value
inStream_TDATA	out	1	axis	inStream_V_data_V	pointer
inStream_TVALID	in	8	axis	inStream_V_data_V	pointer
inStream_TREADY	in	1	axis	inStream_V_dest_V	pointer
inStream_TDEST	out	1	axis	inStream_V_dest_V	pointer
inStream_TKEEP	in	6	axis	inStream_V_keep_V	pointer
inStream_TSTRB	in	1	axis	inStream_V_strb_V	pointer
inStream_TUSER	in	1	axis	inStream_V_user_V	pointer
inStream_TLAST	in	2	axis	inStream_V_last_V	pointer
inStream_TID	in	1	axis	inStream_V_id_V	pointer
outStream_TDATA	out	5	axis	outStream_V_data_V	pointer
outStream_TVALID	out	8	axis	outStream_V_data_V	pointer

outStream_TREADY	in	1	axis	outStream_V_dest_V	pointer
outStream_TDEST	out	1	axis	outStream_V_dest_V	pointer
outStream_TKEEP	out	6	axis	outStream_V_keep_V	pointer
outStream_TSTRB	out	1	axis	outStream_V_strb_V	pointer
outStream_TUSER	out	1	axis	outStream_V_user_V	pointer
outStream_TLAST	out	2	axis	outStream_V_last_V	pointer
outStream_TID	out	1	axis	outStream_V_id_V	pointer

From Table 5.5 it can be observed that the size of the data bus is 8 bits for both input and output stream. It can be increased based on the load of the data.

Table 5. 6: Memory Consumed for Sobel Filter

Memory	Module	BRAM_18K	FF	LUT	Words	Bits	Banks	W*Bits*Banks
lineBuff_value_0_U	deSobel_lineBuff_value_0	1	0	0	240	8	1	1920
lineBuff_value_1_U	deSobel_lineBuff_value_0	1	0	0	240	8	1	1920
lineBuff_value_2_U	deSobel_lineBuff_value_0	1	0	0	240	8	1	1920
Total	3	3	0	0	720	24	3	5760

Table 5.6 shows how much memory has been used for the implementation of the Sobel filter. Also it can be noticed that 4 Bram has been used.

Table 5. 7: Multiplexer used for Sobel Filter

Name	LUT	Input Size	Bits	Total Bits
ap_NS_fsm	6	12	1	12
ap_sig_ioackin_outStream_TREADY	1	2	1	2
col_assign_phi_fu527_p4	32	2	32	64
col_assign_reg_523	32	2	32	64
countWait_1_reg_568	8	2	8	16
countWait_ phi_fu527_p4	17	2	17	34
countWait_reg_557	17	2	17	34

idxRow_phi_fu_538_p4	32	2	32	64
idxRow_reg_534	32	2	32	64
kernel1_address0	8	10	4	40
kernel2_address0	8	10	4	40
lineBuff_value_0_address0	8	3	8	24
lineBuff_value_0_address1	8	3	8	24
lineBuff_value_1_address0	8	4	8	32
lineBuff_value_1_address1	8	3	8	24
lineBuff_value_2_address0	8	4	8	32
lineBuff_value_2_address1	8	3	8	24
out_Stream_TDATA	8	3	8	24
out_Stream_TDEST	6	3	6	18
out_Stream_TID	5	3	5	15
out_Stream_TKEEP	1	3	1	3
out_Stream_TLAST	1	3	1	3
out_Stream_TSTRB	1	3	1	3
out_Stream_TUSER	2	3	2	6
pixConvolved_phi_fu_549_p4	32	2	32	64
pixConvolved_reg_545	32	2	32	64
reg_595	8	2	8	16
Total	337	95	324	810

Table 5.7 describes how many multiplexers has been used. It has 337 Look Up Tables with total bits of 810.

5.5 Co-simulation

After the synthesis has been done co-simulation result will be analyzed. Co-simulation results shows whether the simulation result and the synthesis result are same or not. If it is same then it means that the Sobel filter will work in the hardware. Otherwise the IP core will not work in the hardware. Some changes need to be made to make it work in the hardware.

Table 5. 8: Co-simulation Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	691453	691453	691453	691453	691453	691453

Now the hardware will be made.

5.6 Generating the Hardware

In this step Sobel IP core will be used to make the hardware.

● First the Zynq processor is initialized. It is the main processor which communicates with all the other modules.

● Initialize the DMA (Direct Memory Access). It is the main memory module which will bypass the processor for fetching the data.

● AXI Timer: It will generate the time to show how long the hardware need to give the output. Through this, the time will be tracked.

● Sobel IP Core: This is the Sobel filter. Here the noise free image will be given as an input to this module which will generate the edge detected image.

Figure 4.5 shows the hardware implementation for the Sobel filter. The hardware module will be dumped in the Zynq (SoC). The noise free image pixels will come from DDR and will be sent to the DMA. The DMA will output the pixels to the IP core. The pixels will be processed there and the output will be given to the DMA again. From there it will be sent to the DDR and the result will be saved. This result will be edge detected image pixels. AXI interconnect will take care of the connections. The noise free image pixels will not have to go through the processor but will be directly put into the DMA. After this step the bit stream will be generated for this module which in turn will be exported to the Xilinx SDK.

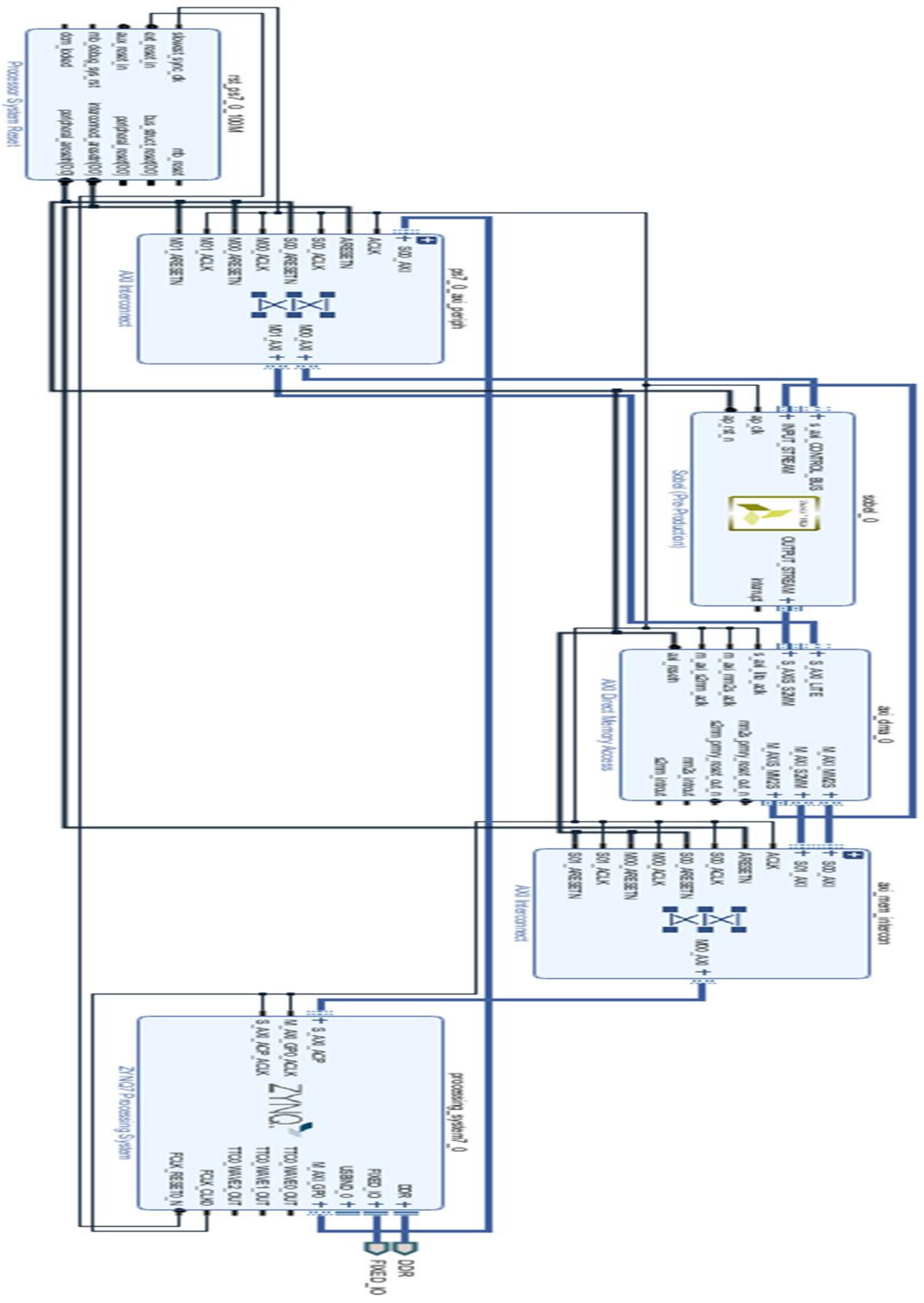


Figure 5. 5: Hardware for Sobel Filter Implementation

5.7 Xilinx SDK for Sobel Filter

In this part the hardware will be tested to see whether it is giving the correct output or not. The testing procedure consists of the following steps:

● Initialize the DMA.

● Initialize the IP core.

● Store the value of the noise free image pixels in the header file. Take the value from the header file and send it to the hardware.

● Initialize the AXI timer.

● Write the two Sobel Kernel values(X-Horizontal side, Y-Vertical side) and start the IP core.

● Flush the cache.

● Specify the transfer function from Device to DMA and also from DMA to Device.

● Invalidate the cache.

● Stop the timer

When we run the hardware it shows us the following output.

```
Starting Hardware......
Stopping the Hardware...
Total Execution time: 0.031581 sec
```

Here, it took around 0.031581 seconds to give the output.

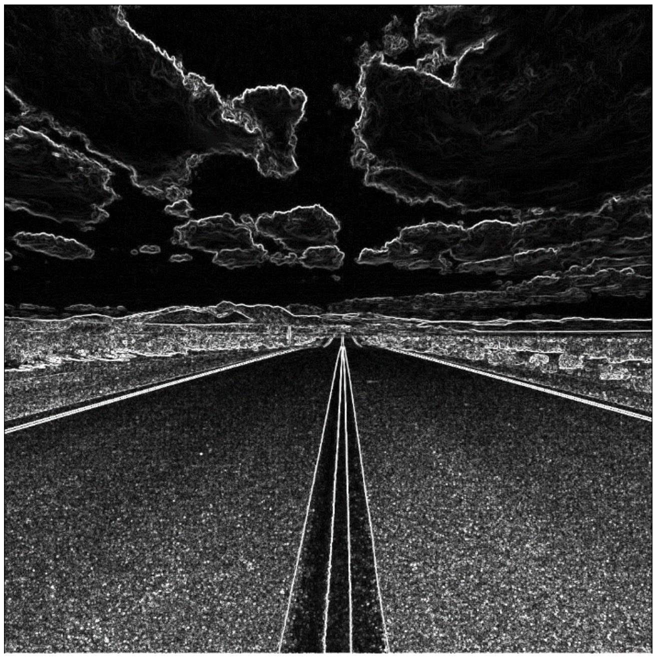
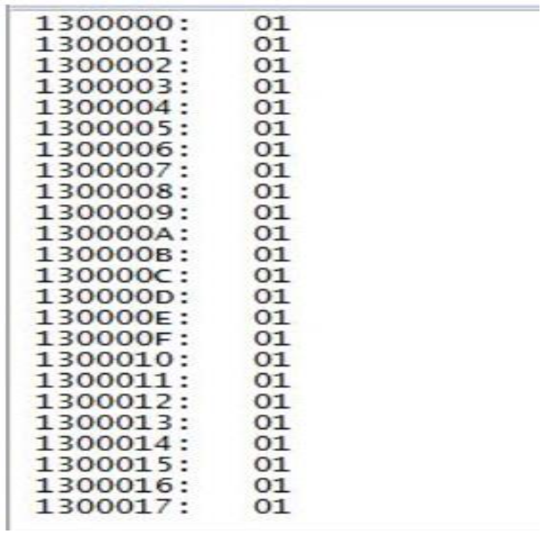


Figure 5. 6: Image Output in Zynq



```
1300000:    01
1300001:    01
1300002:    01
1300003:    01
1300004:    01
1300005:    01
1300006:    01
1300007:    01
1300008:    01
1300009:    01
130000A:    01
130000B:    01
130000C:    01
130000D:    01
130000E:    01
130000F:    01
1300010:    01
1300011:    01
1300012:    01
1300013:    01
1300014:    01
1300015:    01
1300016:    01
1300017:    01
```

Figure 5. 6: Sobel Output Stream

Here from Figure 5.6 it can be observed that the Zynq output is same like the Vivado HLS output. The hardware module is working fine like the software module. It can be verified that the edge has been detected. The white lines are the edge of the image.

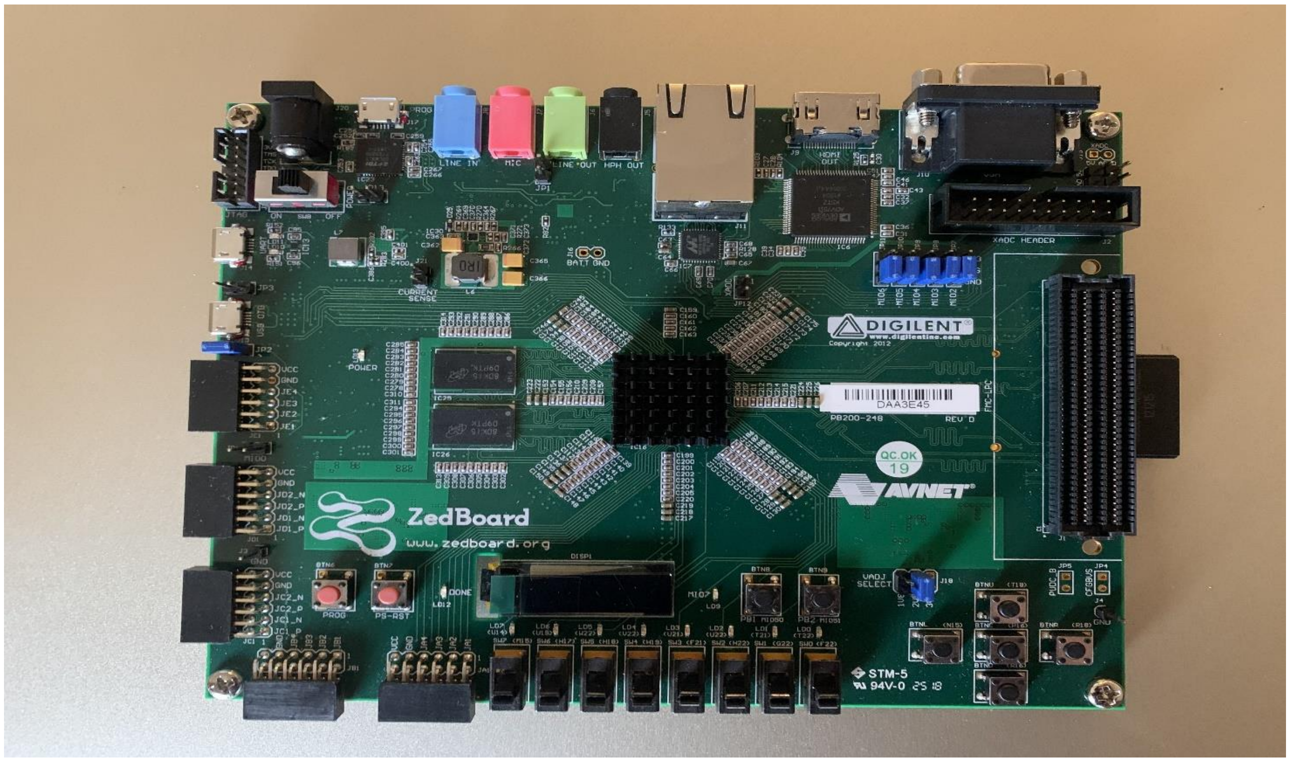
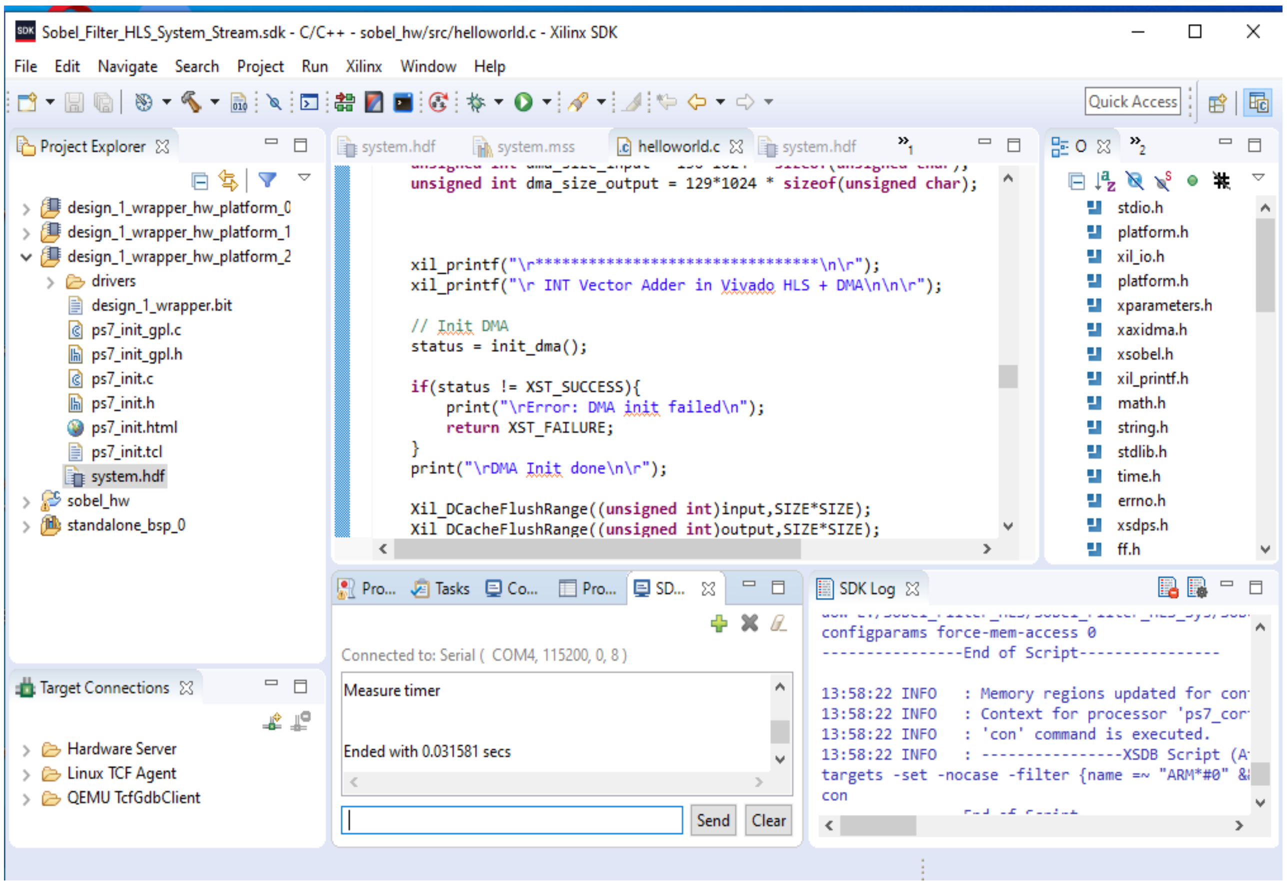


Figure 5. 7: Zedboard



```
unsigned int dma_size_output = 129*1024 * sizeof(unsigned char);



xil_printf("\r********************************\n\r");
xil_printf("\r INT Vector Adder in Vivado HLS + DMA\n\n\r");

// Init DMA
status = init_dma();

if(status != XST_SUCCESS){
    print("\rError: DMA init failed\n");
    return XST_FAILURE;
}
print("\rDMA Init done\n\r");

Xil_DCacheFlushRange((unsigned int)input,SIZE*SIZE);
Xil_DCacheFlushRange((unsigned int)output,SIZE*SIZE);
```

```
Connected to: Serial ( COM4, 115200, 0, 8 )

Measure timer


Ended with 0.031581 secs
```

```
configparams force-mem-access 0
----------------End of Script----------------

13:58:22 INFO   : Memory regions updated for con
13:58:22 INFO   : Context for processor 'ps7_cor
13:58:22 INFO   : 'con' command is executed.
13:58:22 INFO   : ----------------XSDB Script (A
targets -set -nocase -filter {name =~ "ARM*#0" &
con
```

Figure 5. 8: Output screenshot for Xilinx SDK

Figure 5.9 displays the output screen for Xilinx SDK. The output is stored in the form of pixel values which can be seen in Figure 5.7. MATLAB is used to generate the final image from the image pixels generated from the hardware.

5.8 Comparison of Implementation of Sobel filter using different platforms

In this section different platforms to implement the Sobel filter are discussed and compared. As we shall see, they are not as fast as the technique discussed here.

5.8.1 Sobel Edge Detection by Vertex-5 FPGA

The authors [32] discuss Sobel edge detection using Vertex-5 FPGA, where the the image is greyscale, and the number of rows is 480 and the number of columns is 640. For target device Xilinx FPGA device XC5VLX50 of family Vertex-5, XC6SLX25 of family Spartan-6, XC3S400 of family Spartan- 3 is used. Processing on single frame should be completed in a certain desired time to meet a real-time requirement. For typical CMOS digital sensor camera maximum frame rate is 30 frame per second and maximum pixel clock frequency for which pixel data outs are valid is 48 MHz. Processing of a single frame takes 0.07696 s, or 0.06412 s, for pixel clock frequency of 40MHz, or 48MHz respectively, thus meeting the real-time requirement [32]. Nevertheless the execution time is slower than the method described here.

5.8.2 Sobel filter implementation by Aparapi and Java

We compare three implementations of the Sobel edge detection algorithm. The first one is sequential version implemented in Java, the second one is thread version implemented in java using java thread and the third one is GPU version implemented in Java using the aparapi library. The first two versions run on CPU, while the third version runs on GPU. Image is read by an implementation using ImageIO class of java and get array of pixel using its getRGB() method. The same class is used for writing output image. With the various combination (Table 5.9) it can be seen that the speed on GPU using aparapi is 3 to 6 times as compared to CPU sequential implementation with sufficient workload. Speed is 2 times with GPU implementation as compared to threaded implementation if number of thread less than 8. On sufficient workload, time is constant in case of aparapi implementation and scale well. Threaded implementation give up to 2 times speed with respect to GPU implementation using aparapi if number of thread is more than 8. Also the speed is 2 to 10 times with respect to sequential implementation[33].

Table 5. 9: Execution time in milliseconds

Image Resolution	Sequential Implementation	Threaded Implementation				Aparapi Implementation
		2	4	8	16	
255x255	40	43	38	40	47	773
512x512	104	88	79	66	68	778
1024x1024	295	170	149	123	104	794
2048x2048	1060	521	315	207	158	860
3506x3506	3155	1597	937	580	430	1185
4096x4096	4160	1997	1096	856	612	1110

From the above comparison it can be concluded that they are not fast enough as Zynq.

6. CONCLUSION

The objective of this thesis is to develop an efficient implementation of Sobel edge detection. This objective has been met successfully. The modules for the Sobel Filter are implemented in a combination of software and hardware. We have verified that the results are correct. Although the desired outcome was achieved, there are several possible areas of improvement. For example the runtime of the simulation in Vivado HLS for the Sobel Implementation can be decreased. Also more emphasis has to be given to see whether the utilization of the Programming Logic is less for both the IP core. Also though proper edge has detected, it can be more accurate if relevant threshold values can be found for the Sobel Implementation.

REFERENCES

[1] R. N. Beck, "Image processing in the context of imaging science," in *Proceedings., International Conference on Image Processing*, 1995, vol. 2, pp. 304-307 vol.2.

[2] S. Collins and M. Wade, "A critical review of analogue image processing," in *IEE Colloquium on Integrated Imaging Sensors and Processing*, 1994, pp. 1/1-1/6.

[3] L. M. G. Fonseca, L. M. Namikawa, and E. F. Castejon, "Digital Image Processing in Remote Sensing," in *2009 Tutorials of the XXII Brazilian Symposium on Computer Graphics and Image Processing*, 2009, pp. 59-71.

[4] L. Yuan and X. Xu, "Adaptive Image Edge Detection Algorithm Based on Canny Operator," in *2015 4th International Conference on Advanced Information Technology and Sensor Application (AITS)*, 2015, pp. 28-31.

[5] S. Israni and S. Jain, "Edge detection of license plate using Sobel operator," in *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, 2016, pp. 3561-3563.

[6] L. Yang, X. Wu, D. Zhao, H. Li, and J. Zhai, "An improved Prewitt algorithm for edge detection based on noised image," in *2011 4th International Congress on Image and Signal Processing*, 2011, vol. 3, pp. 1197-1200.

[7] A. R. Kapgate and S. S. Agrawal, "Edge based data embedding steganography algorithm using ZedBoard," in *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, 2017, pp. 2203-2207.

[8] Digilent. *ZedBoard Zynq-7000 ARM/FPGA SoC Development Board*. Available: https://store.digilentinc.com/zedboard-zynq-7000-arm-fpga-soc-development-board/

[9] A. Kalra and R. L. Chhokar, "A Hybrid Approach Using Sobel and Canny Operator for Digital Image Edge Detection," in *2016 International Conference on Micro-Electronics and Telecommunication Engineering (ICMETE)*, 2016, pp. 305-310.

[10] M. Chunxi, G. Wenshuo, Y. Lei, and L. Zhonghui, "An improved Sobel algorithm based on median filter," in *2010 2nd International Conference on Mechanical and Electronics Engineering*, 2010, vol. 1, pp. V1-88-V1-92.

[11] K. S. C, G. S. S, K. R. N, and A. L. C, "Analysis of Image Quality using Sobel Filter," in *2019 Third International Conference on Inventive Systems and Control (ICISC)*, 2019, pp. 526-531.

[12] R. J. ius and V. M. ius, "Energy efficient platform for sobel filter implementation in energy and size constrained systems," in *2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, 2015, pp. 1-5.

[13] A. Furnari, G. M. Farinella, A. R. Bruna, and S. Battiato, "Generalized Sobel Filters for gradient estimation of distorted images," in *2015 IEEE International Conference on Image Processing (ICIP)*, 2015, pp. 3250-3254.

[14] Z. E. M. Osman, F. A. Hussin, and N. B. Z. Ali, "Optimization of Processor Architecture for Image Edge Detection Filter," in *2010 12th International Conference on Computer Modelling and Simulation*, 2010, pp. 648-652.

[15] S. K. Mohapatra, B. R. Swain, and S. K. Mahapatra, "Optimized approach of sobel edge detection technique using Xilinx system generator," in *2015 2nd International Conference on Electronics and Communication Systems (ICECS)*, 2015, pp. 338-341.

[16] S. Yaman, B. Karakaya, and Y. Erol, "Real Time Edge Detection via IP-Core based Sobel Filter on FPGA," in *2019 International Conference on Applied Automation and Industrial Diagnostics (ICAAID)*, 2019, vol. 1, pp. 1-4.

[17] S. Eetha, S. Agrawal, and S. Neelam, "Zynq FPGA Based System Design for Video Surveillance with Sobel Edge Detection," in *2018 IEEE International Symposium on Smart Electronic Systems (iSES) (Formerly iNiS)*, 2018, pp. 76-79.

[18] M. Afif, Y. Said, H. Bahri, and M. Atri, "Efficient implementation of sobel filter based on GPUs cards," in *2016 International Image Processing, Applications and Systems (IPAS)*, 2016, pp. 1-4.

[19] H. B. Fredj, M. Ltaif, A. Ammar, and C. Souani, "Parallel implementation of Sobel filter using CUDA," in *2017 International Conference on Control, Automation and Diagnosis (ICCAD)*, 2017, pp. 209-212.

[20] H. N. T. K, C. Belleudy, and T. V. Pham, "Power evaluation of Sobel filter on Xilinx platform," in *2014 IEEE Faible Tension Faible Consommation*, 2014, pp. 1-5.

[21] G. Wenshuo, Z. Xiaoguang, Y. Lei, and L. Huizhong, "An improved Sobel edge detection," in *2010 3rd International Conference on Computer Science and Information Technology*, 2010, vol. 5, pp. 67-71.

[22] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable Computing Architectures," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 332-354, 2015.

[23] J. C. Lyke, C. G. Christodoulou, G. A. Vera, and A. H. Edwards, "An Introduction to Reconfigurable Systems," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 291-317, 2015.

[24] S. Singh and R. Singh, "Comparison of various edge detection techniques," in *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, 2015, pp. 393-396.

[25] Ashish. (2018). *Understanding Edge Detection (Sobel Operator)*. Available: https://medium.com/datadriveninvestor/understanding-edge-detection-sobel-operator-2aada303b900

[26] S. Sodha. (2017). *An Implementation of Sobel Edge Detection*. Available: https://www.projectrhea.org/rhea/index.php/An_Implementation_of_Sobel_Edge_Detection

[27] Edge Detection [Online]. Available: https://www.cs.auckland.ac.nz/compsci373s1c/PatricesLectures/Edge%20detection-Sobel_2up.pdf

[28] Xilinx. Vivado Design Suite User Guide High-Level Synthesis [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf

[29] Xilinx. AXI Reference Guide [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf

[30] Xilinx. Zynq-7000 SoC Technical Reference Manual [Online]. Available: www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[31] Xilinx. AXI DMA v7.1 [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf

[32] G. Chaple and R. D. Daruwala, "Design of Sobel operator based image edge detection algorithm on FPGA," in *2014 International Conference on Communication and Signal Processing*, 2014, pp. 788-792.

[33] K. G. Gupta, N. Agrawal, and S. K. Maity, "Performance analysis between aparapi (a parallel API) and JAVA by implementing sobel edge detection Algorithm," in *2013 National Conference on Parallel Computing Technologies (PARCOMPTECH)*, 2013, pp. 1-5.