Hardware Acceleration of Image Processing Algorithms on Low-Power Embedded Platforms

By

Francesco Buttafuoco

Master in Embedded Systems, Computer Engineering
Polytechnic University of Turin

Defence Committee:
Prof Andrea Calimera, Polytechnic University of Turin, Italy
Prof Iris Bahar, School of Engineering at Brown, USA

December 2017

Abstract

This paper describes the implementation of two main algorithms based on hardware-software co-design. The aim of the two algorithms is to identify some features from particular images. The first algorithm is used for Iris Recognition. The algorithm was developed by [1] for recognizing person by their iris patterns. We propose hardware accelerators for the most timing consuming part of the algorithm. Next, we identify overall eight parameters at both the algorithmic and hardware levels to trade-off accuracy with runtime. To identify the optimal values for these parameters, we identify different novel design space exploration techniques including Gradient Based Local Search. The second algorithm is called Gradient-Based Cross-Spectral Stereo Matching [3] and it is used to identify visual stereo matching when weather conditions offer zero visibility. The aim of this part was to implement the hardware architecture on an UltraScale FPGA.

Acknowledgements

Firstly, I would like express my gratitude to my supervisor, Prof. Andrea Calimera, who offered me this amazing experience at the Brown School of Engineering. I would like to thank the other supervisor, Prof. Iris Bahar, whose motivation and encouragement have been essential during these months; she has always been present, supporting me to overcome all the difficulties and giving me precious suggestions. I would like to thank the other guys I have worked with: Soheil Hashemi, Tann Hokchhay, Prof. Sherief Reda and Christopher Picardo. I need to thank my parents too, supporting me in many ways. A final thanks to my girlfriend, who help and support me in this journey.

Table of Contents

Page

List of Tables **vii**

List of Figures **ix**

1 Introduction **1**

I Iris Recognition **3**

2 Algorithm review **4**

2.1 Briefly description 4

2.2 Iris Segmentation 6

2.3 Focus Assessment 8

3 System Architecture Overview and SW/HW Partitioning **10**

3.1 System Architecture Overview 10

3.2 SW/HW Partitioning 13

4 Scale Analysis **16**

5 Hardware accelerator for the segmentation part **18**

5.1 Sofware-C code side 18

5.2 Hardware-Verilog side 25

6 Hardware accelerator for the focus assessment **29**

7 Methodology for Approximate Computing based on Iris Scanning **33**

7.1 Introduction 33

7.2 Methodology 36

7.3 Results 38

II GBCSSM-Pinggera **41**

8 Algorithm review **42**

8.1 Introduction 42

8.2 The Gradient Based Cross-Spectral Stereo Matching Algorithm 43

9 Architecture **45**

9.1 System Architecture Setup 45

9.2 System Generator Implementation 48

9.2.1 Convolution 49

9.2.2 Sum of Squares 52

9.2.3 Square Root 52

9.2.4 L2 Unit Normalization 54

9.2.5 Division (Final HOG Matrix) 55

9.3 Hardware Implementation using Vivado 55

9.4 Conclusion 60

Bibliography **61**

List of Tables

Table **Page**

3.1 Available resources on Spartan-6 XC6SLX45T-3FGG484 FPGA. 11

7.1 The list of parameters evaluated in the design space exploration. Values in brackets show the possible values. 35

7.2 The formulation used to model the runtime behavior as a function of the design space parameters. Note that as we do not modify any parameters in the encoding components, we consider its runtime as a constant. 35

7.3 The components chosen for hardware acceleration, the corresponding speedups, and the hardware utilization of each accelerator. 38

7.4 Parameter provided by the gradient descent method 39

7.5 The hardware characteristics of the end-to-end system. 39

9.1 Available resources in Virtex UltraScale XCVU095-2FFVA2104E FPGA. 46

LIST OF FIGURES

FIGURE		Page
2.1	The components of an iris recognition system.	4
2.2	Example of an iris pattern.	5
2.3	Flowchart of the iris segmentation stage.	6
2.4	Integrodifferential operator.	7
2.5	The (8 x 8) convolution kernel for fast focus assessment.	9
3.1	Picture of a Spartan-6 SP605 development board.	11
3.2	Overview architecture for Iris Code implementation.	12
3.3	Percentage of runtime used by various algorithms in the iris scanning pipeline.	14
4.1	The distribution of the candidates and mean Error Pixel with respect to the scales factor.	17
4.2	Iris segmentation using different scale factors.	17
5.1	Contour integral.	23
5.2	Fields of the register 7.	23
5.3	Overview of the integrodifferential operator.	25
5.4	1st Layer State Machine for Integrodifferential Operator.	26
5.5	2nd Layer State Machine for Integrodifferential Operator.	27
5.6	Fields of register in fixed precision.	27
5.7	The schematics of the convolution circuit.	28
6.1	Architecture of the Convolution Accelerator.	30
6.2	State Machine for Focus Assessment Module.	32

7.1 Architecture of the Convolution Accelerator. 38
7.2 The camera and FPGA board Setup. 40

8.1 Cost Matrix and Disparity Map generation. 43
8.2 Cost Matrix and Disparity Map generation. 44

9.1 Picture of a Xilinx Virtex UltraScale FPGA VCU108 development board. 45
9.2 Vivado IDE on the left, SDK on the right. 46
9.3 Architecture for Testing. 47
9.4 Architecture of the design-in-board. 48
9.5 Flow of testing. 48
9.6 GBCSSM Design implemented in System Generator. 49
9.7 First Step of the pipeline. 50
9.8 Convolution Module. 51
9.9 2D Convolution. 51
9.10 Sum of Squares Module. 52
9.11 Square Root Module. 54
9.12 Square Root Module. 54
9.13 Overview architecture for GBCSS implementation. 56
9.14 Flow of the GBCSS implementation. 57
9.15 Top Conv Module. 58
9.16 Square Root Module. 59
9.17 PRE WRITE Module. 60

CHAPTER 1

INTRODUCTION

For real time large-scale application, image processing algorithms have to be implemented in embedded systems because powerful systems, that run at GHz, cannot be used due to the high price components, high power consumption, scalability, ecc. For this reason, architectures based on hardware−software co-design combine the advantages of both hardware and software solutions. Such systems contain an embedded microprocessor and several dedicated hardware units connected via a communication bus. The advantage of using hardware accelerators is to provide high speed and low power consumption. The purpose of this paper is to describe the implementation of an iris recognition and GB-CSSM algorithms based on a hardware−software co-design system, suitable for integration in a FPGA. In both projects, I worked with Xilinx software and Xilinx FPGA. This paper is organised in 2 main parts. The part I reviews the basic principles of the iris recognition algorithm, the scale analysis on the segmentation part, the HW accelerator of the segmentation part, the HW accelerator of the focus assessment part and a novel design space exploration technique used in the paper [2]. The part II reviews the basic idea about the GB-CSSM algorithm and the hardware architecture.

Part I

Iris Recognition

Chapter 2

Algorithm Review

2.1 Briefly description

The iris recognition is a kind of the biometrics technologies based on the physiological characteristics of human body and, compared with the feature recognition based on the fingerprint, palm-print, face, etc, the iris has some advantages such as uniqueness, stability, high recognition rate. Iris patterns have now been tested in many field and laboratory trials, producing no false matches in several million comparison tests. The iris pattern is unique to each person and to each eye, and is essentially stable during an entire lifespan.

A typical iris recognition system uses a pipeline of four components that takes input images from a camera and produce as output the 2048 bit encoding of the iris. Fig 2.1 shows the flowchart used for our experiments. At the front-end, a camera with an infrared sensitive sensor captures

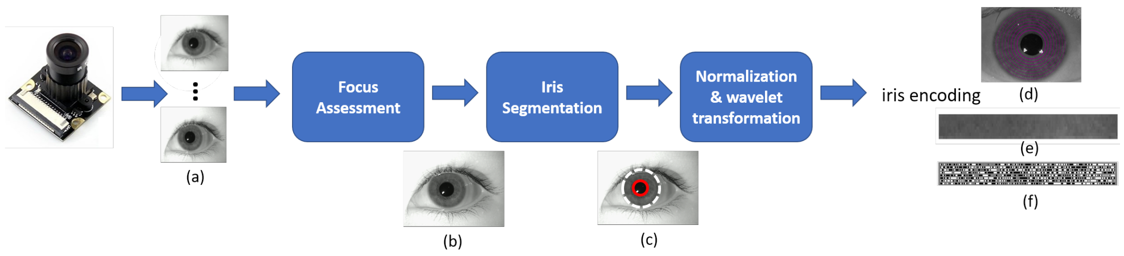


FIGURE 2.1. The components of an iris recognition system.

multiple frames of an iris illuminated with infrared LEDs as given in Fig 2.1.a. The ideal image is taken at a distance of about 35 cm by ensuring a minimum of 70 pixels in iris radius, letting to capture the rich details of iris patterns. The focus assessment stage captures the sharpest frame from the previous sequence of frames as illustrated in 2.1.b. The degree of focus for each frame is computed convolving a 8x8 kernel with each frame. The segmentation stage extract the the center points and the radius for the iris and the pupil. In this work, the integrodifferential algorithm is utilized to compute the center points. This stage is shown in 2.1.c. The output of the segmentation algorithm is then fed to the normalization algorithm where the iris pixels are subsampled and organized in a Cartesian coordination system. This is achieved by simply spacing the angular resolution and the radial resolution equally, based on the segmentation results. Figure 2.1.d and 2.1.e show the subsampling process and the resulting 2-D output of the normalization respectively. Finally, the normalized pixels are encoded into 2048 bits using a 2-D Gabor demodulation as shown in Figure 2.1.f. These 2048 bits are the iris signature. The encode sequence is illustrated for one iris by the bit stream shown graphically in Figure 2.2. Since the

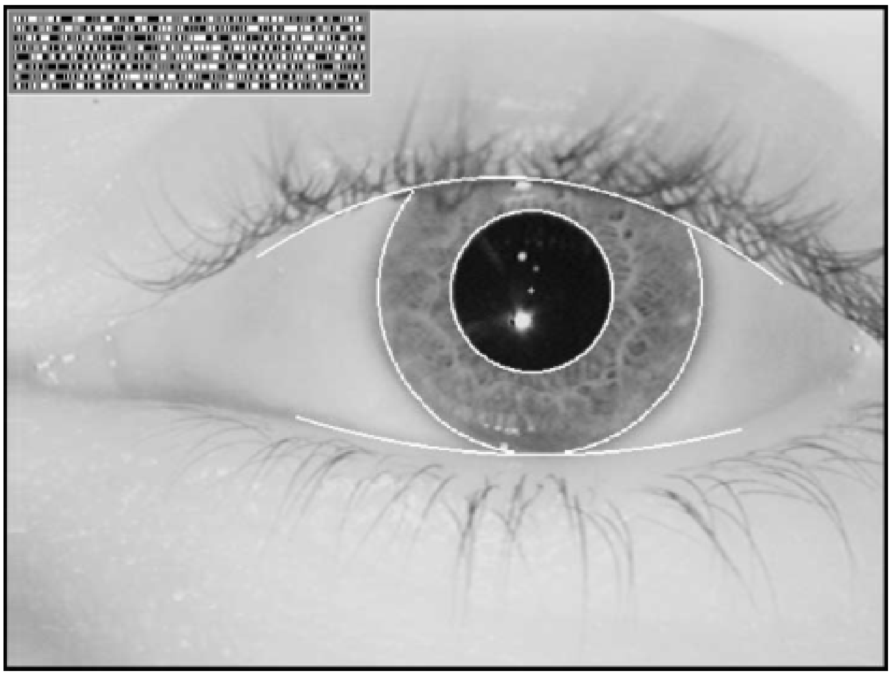


FIGURE 2.2. Example of an iris pattern.

goal is to minimize the response time to the user, the runtime of the system is the prime objective for iris scanning systems. After profiling the entire flow, the most time consuming stages are Iris Segmentation and Focus Assessment. Before going deeply in the hardware implementation, it is better focusing on the algorithms behind these stages.

2.2 Iris Segmentation

The Segmentation is the process of locating the iris in the image. The flowchart shown in Figure 2.3 shows the steps required for the segmentation assessment. The first step is scale the image.

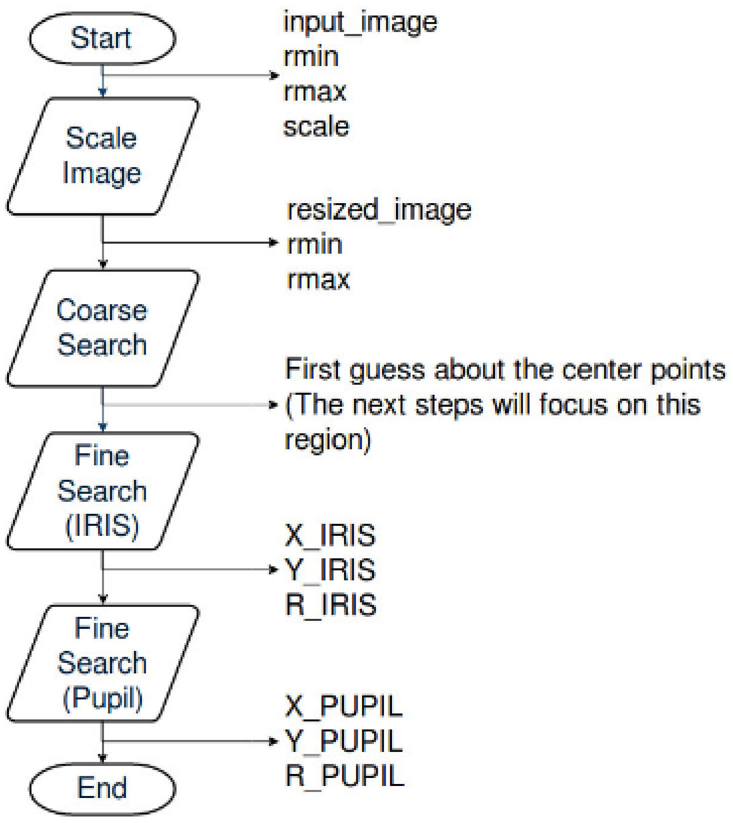


FIGURE 2.3. Flowchart of the iris segmentation stage.

We use 480x640 as the original size input frames, however, as the segmentation is the most computationally demanding part of the algorithm, to speedup the runtime, a resized image is used. As will be described more fully in the Chapter 4, after some explorations, we use a 120x160 (scale factor of 4) resized image as our input. The algorithm chosen to compute the rescaling is the Nearest-neighbor interpolation, that consist in sub-sampling the original image

with a offset of 4 pixels. Comparing the different resizing algorithms typically used in rescaling processes, the Nearest-neighbor interpolation is the simplest and fastest solution. The r_{min} and r_{max}, shown in the flowchart, are the minimum and maximum possible radius used for iris search. We set these parameters to be equal to 60 and 180 respectively, because are closely related to the dimension of the eye captured by the camera. The rescaling stage returns the resized image, the resized minimum and the maximum radius as well. Another parameter that is used in the integrodifferential algorithm is the N_{points}, a value that devises for how many points the contour integral is to be computed. This parameter is scaled too. The initial value is 600, and after the scaling is equal to 150.
Next, the integrodifferential algorithm is utilized to compute the center points. The coarse search and fine search for the iris and pupil use the same operator, with some variations. The formula of the integrodifferential operator is shown in Figure 2.5. The operator searches over the image for

$$\max_{(r,x_0,y_0)} \left| G_\sigma(r) * \frac{\partial}{\partial r} \oint_{r,x_0,y_0} \frac{I(x,y)}{2\pi r} ds \right|$$

FIGURE 2.4. Integrodifferential operator.

the maximum value computed convolving a $G_\sigma(r)$ Gaussian function of scale σ with a contour integrator along a circular arc ds of radius r and center x_0, y_0. The symbol $*$ denotes convolution and $G_\sigma(r)$ is a smoothing function. The complete operator behaves as a circular edge detector, searching iteratively for the maximal contour integral derivative.
The first step is the coarse search. It is required to identify the possible area to look for the iris center point. In this stage, instead of apply the integrodifferential operator for all the points, some filtering techniques are applied. Firstly, the candidates are searched in a restricted region delimited by the minimum radius along all the directions. The possible center point is expected to have a minimum radius r_{min}, so points with a position that have a distance from the borders lower than r_{min} are discarded. Among all the possible candidates, only the local minima are chosen. A 3x3 window is used for this purpose: if the candidate, located in the middle of this window, has the lower value, is selected. Further, for each pixel out of the boundary off the image,

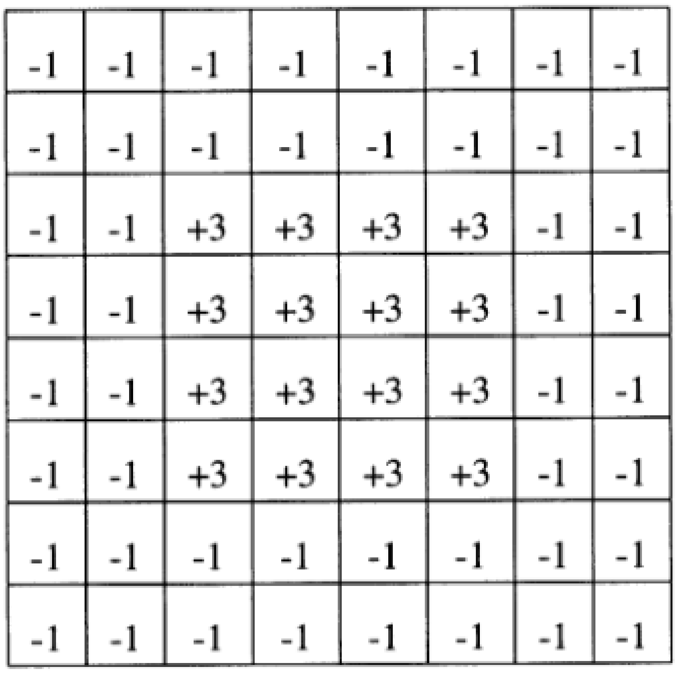
the pixel is assumed to be candidate if the intensity of the pixel is darker than a predefined threshold and it is a local minima. The center point is expected to be inside the pupil, the darkest region of the image. The value of the threshold is set to be 50. Only for these candidates is computed the integrodifferential operator. In this stage the $G_\sigma(r)$ smoothing function uses a $\sigma = \infty$, that transforms the gaussian curve in a costant function equal to 1. The convolution is computed using 7 discrete elements of $G_\sigma(r)$.
The center point returned by the coarse search is used as input for the next stage. The fine search for the iris applies the integrodifferential operator in a window 15x15 centred in the previous center point. The maximum contour integral identifies the center point and radius of the iris. These value are passed to the next stage, the fine search for the pupil. It again applies the integrodifferential operator in a window 15x15 centred in the iris coordination. Finally, this stage returns the center point of the pupil. In fine iris and pupil search, a $\sigma = 0.5$ is used, smoothing the derivative. This approach reduces the counter integral because an high value for a particular radius, due to an interference in the image (for example reflection of the infrared LED), is smoothed to the nearest low counter integrals.

2.3 Focus Assessment

In our implementation, the focus assessment is performed in real time on 10 frame per second by measuring the power in middle and upper frequency bands of the 2-D Fourier spectrum. In the image domain, the defocus is normally represented as convolution of an in-focus image by the 2-D function modelled as a Gaussian. The kernel suggested by the paper [1] is shown in the Figure **??** In frequency domain, the convolution is represented as a 2-D Fourier transformation. The convolution with this kernel amplifies the higher frequencies. Thus, an effective way to estimate the focus of an image is to measure the total power of the result of the convolution. The frame with higher energy is passed to the next stage. The formulation of this step can be expressed as

$$max_i(frame_i * kernel)$$

An optimization adopted to improve the runtime was to perform the convolution not on the entire frame, but only in a particular region of interest (ROI). After some exploration, by choosing a



-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	+3	+3	+3	+3	-1	-1
-1	-1	+3	+3	+3	+3	-1	-1
-1	-1	+3	+3	+3	+3	-1	-1
-1	-1	+3	+3	+3	+3	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1

FIGURE 2.5. The (8 x 8) convolution kernel for fast focus assessment.

ROI of 50%, the runtine decreases by one half without any impact on the final result.

Chapter 3

System Architecture Overview and SW/HW Partitioning

This project has been realized in collaboration with the Videology Imaging Solutions company. The latter asks to realize the implementation of iris detection and encoding on the Spartan-6 SP605 development board requiring a runtime of few seconds. In addiction, to validate the performance and to compare against industry standards, the MMU open source dataset and live feeds captured from the camera system are used.

3.1 System Architecture Overview

The board that we used was a Spartan-6 SP605 development board, as shown in Figure 3.1. The SP605 provides many board features, the main features used in this project are:

- Spartan-6 XC6SLX45T-3FGG484 FPGA
- 128 MB DDR3 Component Memory
- Linear BPI Flash
- USB JTAG
- Clock generator at fixed 200Mhz

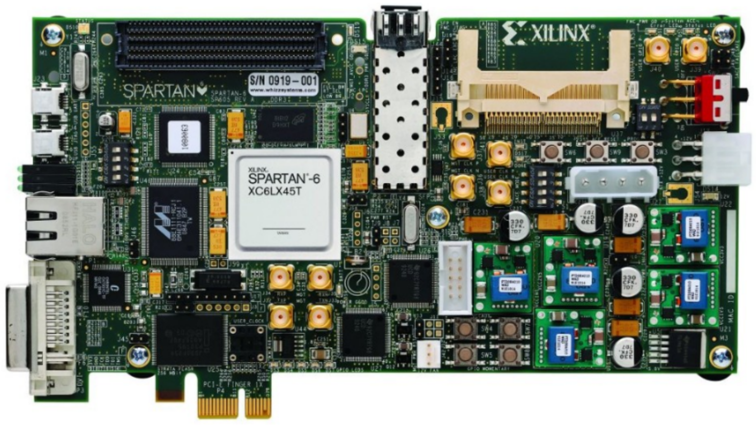


FIGURE 3.1. Picture of a Spartan-6 SP605 development board.

- USB UART
- User I/O

The main resources of the FPGA are in the Table 3.1. For further information about the fpga itself [6] and [5]. The entire project was entire deployed on the FPGA without any external

Table 3.1: Available resources on Spartan-6 XC6SLX45T-3FGG484 FPGA.

Device	Logic Cells	Flip-Flops	Max Distributed RAM Kb	Blocks RAM of 18 Kb
XC6SLX45T	43661	54576	401	116

processor unit or resource. To capture the pictures, we used a infrared camera 5MP USB3 camera 24B5.0XUSB3, provided by Videlogy. The company provided a camera interface module already adjusted for iris capture which plugs into the SP605 development board and outputs raw YUV video. The overview of the architecture is shown in Figure 3.2. The design uses a microblaze softcore processor as the main controller and for software-side computations. The processor runs at 100 MHz with floating point unit enabled.

Three main buses are used AXI, AXI-lite, and Xilinx LMB (local memory bus). The AXI bus runs at 100 MHz. allows high throughput communications necessary for accessing the onboard DDR3 memory, which is the only slave of this bus. The four masters for the bus are the microblaze and the camera interface running at 100 MHz and the convolution accelerator and line integral accelerator running at 30 MHz. The AXI-lite bus operates at 50 MHz and allows for lower speed, single transfer communications. This bus allows the microblaze to send control signals to the

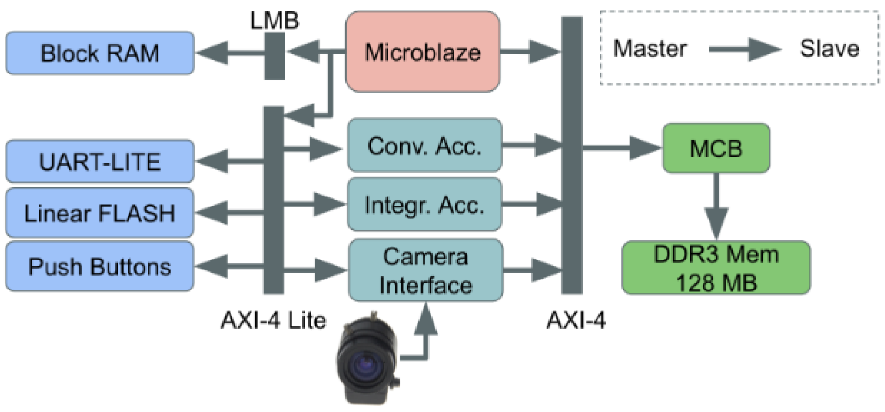


FIGURE 3.2. Overview architecture for Iris Code implementation.

accelerators as well as reading from the IO module such as the push buttons. The LMB is used for transactions between microblaze and the on-chip block RAM, which is responsible for instruction and data caching.

The linear flash stores the FPGA configuration bitstream with a bootloader and the iris code software. At boot time, after the FPGA is configured, the bootloader copies the software to the DDR3 memory and starts the execution.

Next, a description of all main component present in the architecture. The main processor is a Microblaze. MicroBlaze is the Xilinx FPGA-based, 32-bit RISC Harvard architecture soft processor. It supports advanced architecture options such as AXI interface, Memory Management Unit (MMU), instruction and data-side cache, configurable pipeline depth and Floating-Point unit (FPU). The soft processor core is included with the Xilinx software tools. The UART-LITE module is used to interact with the external word. It is used to send informations and status of the board to the user and to return the encode of the iris as well. The Push Button is used to kick start the process of frame capturing and encode pattern generation. The MCB (memory controller block) is the interface used by the Microblaze and the accelerators to access to the DDR3. It is connected to the AXI protocol that support the BURST transfer mode, used to the frame capturing. The MCB-Based Camera Interface Module performs asynchronous data transfer from a FIFO buffer to the DDR3 memory on the Xilinx SP605 board. The FIFO unit used in the development of this module is the RAM FIFO clock domain converter. The unit has a slave

and a master port connected to AXI-4 Lite and AXI-4 interconnect respectively. The slave port receives control signals from the Microblaze module such as destination address and number of frames to be transferred. The master port sends pixel data as well as control signals to the MCB. The slave port of the camera interface module follows AXI-4 Lite protocol, which supports only single transfer of 32-bit data. The master port follows AXI-4 Burst protocol. The burst length is configured to be 128 words where each word is 32 bits. The Convolution and Integrointegral Accelerator perform the focus assessment and the integrointegral operator respectively. Both modules will be described deeply in the next chapters.
Finally, the entire project was built with Xilinx's EDK 14.6 (Embedded Development Kit), a development package provided by Xilinx that collects different tools useful for the development. The project manager consists of two separate environments: ISE, XPS and SDK. Xilinx ISE (Integrated Synthesis Environment) is a software tool for synthesis and analysis of HDL designs. It was used to build and test all the accelerators present in the architecture. XPS (Xilinx Platform Studio) to configure and build the hardware specification of their embedded system (processor core, memory-controller, I/O peripherals, etc.) The XPS converts the designer's platform specification into a synthesizable RTL description, and writes a set of scripts to automate the implementation of the embedded system (from RTL to the bitstream-file.) For the MicroBlaze core, the EDK normally generates an encrypted (non human-readable) netlist. The SDK (Software Development Kit) is the Integrated Design Environment for creating embedded applications built on Eclipse. Our code was written in C and compiled using SDK.

3.2 SW/HW Partitioning

As an initial step, we profile the pipeline, running entirely in software, to measure the runtime of its different algorithmic components. To minimize the runtime, we then synthesize the most computational intensive modules into hardware accelerators until we reach the resource limit of our programmable logic device.
The pie chart in Figure 3.3 summarizes the runtime profiling results when running the flow in software. As shown in the graph, the overwhelming majority of the runtime is spent in the focus

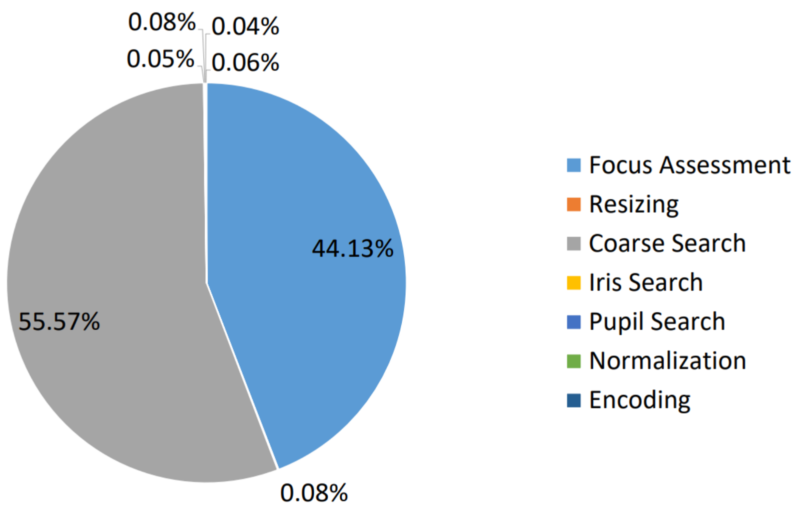


FIGURE 3.3. Percentage of runtime used by various algorithms in the iris scanning pipeline.

assessment and the segmentation components, while the encoding component consumes less than 1% of the total runtime. Therefore, we choose to maps these two components into custom hardware accelerators.

To measure the runtime for each step, an AXI timer is adopted. Next, is shown the C code used to use the module. The timer is set up using macros defined in the library *xtmrctr.h*, allowing to reset, start and stop it. The timer stores the result in 2 register of 32 bits. For this reason, the number of clock cycles measured is printed out concatenating the 2 registers in hexadecimal. The run time in second is computed dividing the clock cycles by the clock frequency of the timer, in our case is et to 50Mhz.

```
XTmrCtr  timer1;
Xint32 t1_L, t1_H, t2_L, t2_H;
...
void tic(){
    XTmrCtr_SetResetValue(&timer1,0,0);
    XTmrCtr_SetResetValue(&timer1,1,1);
    XTmrCtr_SetOptions(&timer1, 0, XTC_CASCADE_MODE_OPTION);
    XTmrCtr_Start(&timer1,0);
    t1_L = XTmrCtr_GetValue(&timer1,0);
    t1_H = XTmrCtr_GetValue(&timer1,1);
}
void toc(){
    t2_L = XTmrCtr_GetValue(&timer1,0);
    t2_H = XTmrCtr_GetValue(&timer1,1);
    xil_printf("%x%x,",t2_H-t1_H,t2_L-t1_L);
    XTmrCtr_SetResetValue(&timer1,0,0);
    XTmrCtr_SetResetValue(&timer1,1,1);

}
...
tic();
focus_analysis();
toc();
tic();
image_rescaling();
toc();
tic();
segmentation();
toc();
tic();
normaliseiris();
toc();
tic();
encode();
toc();
...
```

Chapter 4

Scale Analysis

To reduce the runtime of the iris segmentation, we tried to reduce the dimension the size of the input image. The segmentation part doesn't require an high resolution to identify the center point. Moreover, the run time is affected by the number of the candidates, indeed the coarse search computes the integrodifferencial operator in all the candidates that pass the constraints defined in the Section 2.2. On the other hand, in the fine search the number of candidates is always fixed because the search window is equal to 6x6. The original size of the frame is 640x480 pixels, consequently providing a huge number of possible candidates. To determine the scale factor that identifies the best trade-off between runtime and error, the algorithm was executed multiple times using different scale factors. The scale factors used are shown below

$$scale\_factor = [0.1 \quad 0.2 \quad 0.25 \quad 0.3 \quad 0.4 \quad 0.5 \quad 1]$$

The experiment collects for each factor the number of candidates in which the integrodifferencial operator is executed and the mean error of pixels with respect to the scale equal to 1. The results are shown in Figure 4.1(a) and Figure 4.1(b). The two graph-bar show an exponential relationship among the two metrics. Looking at the 4.1(b) it possible identify a huge gap between scale 0.2 and 0.25. The mean error introduced by the scale 0.25 is 30 times lower than 0.2 scale and only 6 times higher than scale 1. Moreover, the scale 0.25 reduces the number of candidates to 51,

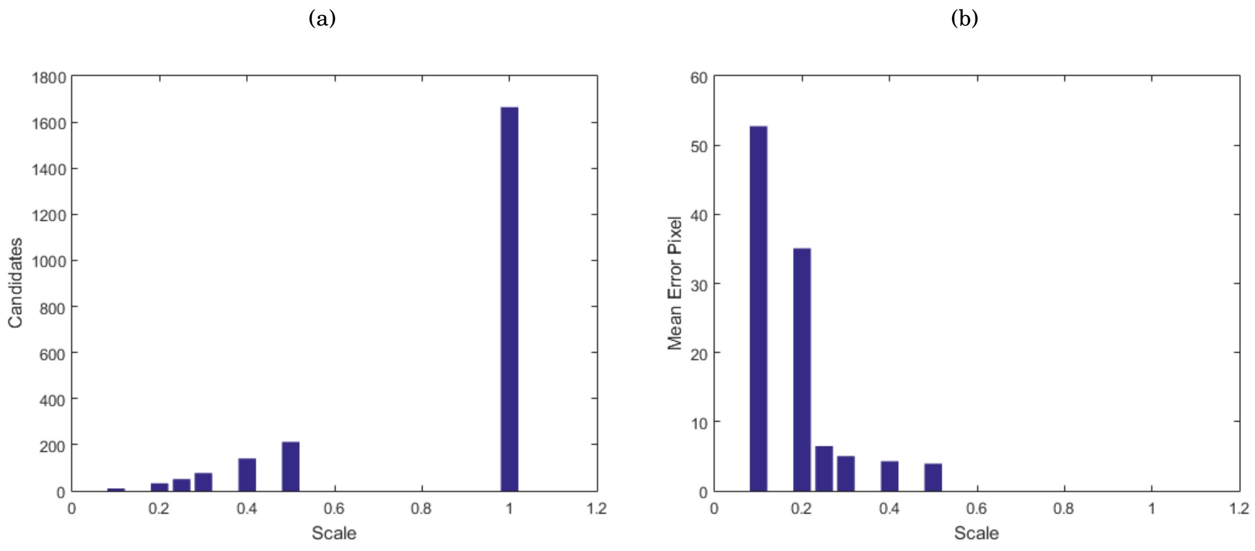


FIGURE 4.1. The distribution of the candidates and mean Error Pixel with respect to the scales factor.

reducing by 32 times the candidates identified using scale equal to 1. This information allows us to choose as ideal scale factor 0.25 to be applied to the input image used by the iris segmentation. The Figure 4.2 shows the iris segmentation in two images using different factors. The runtime shown is provided by MATLAB.

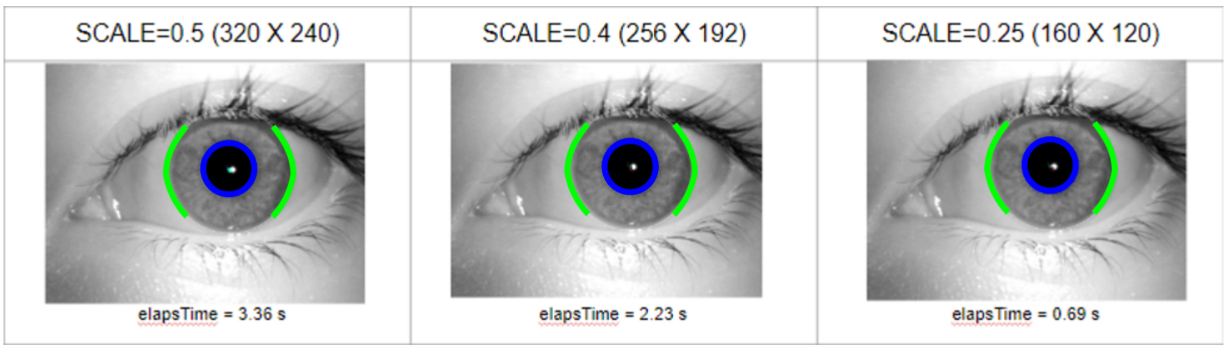


FIGURE 4.2. Iris segmentation using different scale factors.

CHAPTER 5

HARDWARE ACCELERATOR FOR THE SEGMENTATION PART

Although the optimization of the scale factor described in the Section 4 provide a 32X speed-up, the run time of the entire system is still higher than the required one. The main run-time improvement is provided by the HW-SW co-design's introduction. The first part of the algorithm implemented in hardware is the integrodifferential operator.

5.1 Sofware-C code side

The macros, defined to read and write the registers in the peripheral, are used to access the peripheral by the C program. The algorithm's implementation is defined in the library *segment.c*.The wrapper function is called *segmentation()*. The body is partitioned in:

- coarse search
- search maximum
- iris search
- search maximum
- pupil search

- search maximum

As explained in Section 2.2, the coarse search is the first step to identify the points in which there is an high probability to find the iris's center point. The search has place in a sub-portion of the image, focusing only in the part of an image with an offset greater or equal to the minimum radius of the iris in each direction. For each pixel with intensity lower than a threshold value, the integrodifferential computation can start. The threshold value was empirically defined after some exploration in order to apply a first coarse filter, so that the operator is computed for only the point that are more likely the center point. The threshold was fixed to 50. Just as reminder, the center point is expected to be in the pupil, the dark region of the image. The black is encoded in RGB as 0 and white as 255. For this reason are selected only the pixel with an intensity lower than 50. Before running the accelerator on that pixel, the latter has to be the minimum local in a window 3x3 centred on it. This avoid to call the accelerator for each near point that satisfy the threshold constraint. If the pixel satisfies all the previous constraints, the integrodifferential value can be computed. The results of the the accelerator are stored in two matrices, the *maxblurSCH* that stores the integrodifferential value, and the *maxradSCH* that stores the associated radius. When all the values are computed and stored in the matrices, the search of the maximum in the *maxblurSCH* matrix is performed. That one provides the point where the iris's center point is located. Next, the integrodifferential operator is performed in a window 6x6 centred in the center point computed previously. Again, the results are stored in the previous two matrices and finally the maximum is identified. The coordination of the maximum identifies the center point of the iris, and the value in the *maxradSCH* matrix at the same location is the radius. Finally, the integrodifferential operator is performed once again in a window 6x6 centred in the iris center point. Unlike the previous cases, in which the integrodifferential operator worked in a range from pre-defined minimum and maximum radius (related to the physically dimension of the capture eye), the range is computed dynamically, assigning the 10% of the iris's radius to the minimum radius and the 80% to the maximum radius. The final maximum research identifies the pupil's center point and the radius as well. Next the *segmentation()* implementation

```
void segmentation(){
        ///////////////////////// coarse search ////////////////////////
        memset(maxradSCH, 0, sizeof(maxradSCH));
        memset(maxblurSCH, 0, sizeof(maxblurSCH));
        for (center_x = rmin_t; center_x < ROWS-rmin_t; center_x++)
         for (center_y = rmin_t; center_y < COLS-rmin_t; center_y++)
          if (image[center_x][center_y]<th)
           if((image[center_x][center_y]<=image[center_x-1][center_y-1])&&
                (image[center_x][center_y]<=image[center_x-1][center_y])&&
                (image[center_x][center_y]<=image[center_x-1][center_y+1])&&
                (image[center_x][center_y]<=image[center_x][center_y-1])&&
                (image[center_x][center_y]<=image[center_x][center_y])&&
                (image[center_x][center_y]<=image[center_x][center_y+1])&&
                (image[center_x][center_y]<=image[center_x+1][center_y-1])&&
                (image[center_x][center_y]<=image[center_x+1][center_y])&&
                (image[center_x][center_y]<=image[center_x+1][center_y+1]))
                 partiald_acc(
                 center_x,
                 center_y,
                 0, //sigma
                 &maxradSCH[center_x][center_y],
                 &maxblurSCH[center_x][center_y],
                 Iris,
                 rmin_t,
                 rmax_t);

        //search maximum
        max_matrix(maxblurSCH, &x_max, &y_max);
        ///////////////////////////////////////////////////////////////////////////

        //////////////////////// iris search ////////////////////////
        memset(maxradSCH, 0, sizeof(maxradSCH));
        memset(maxblurSCH, 0, sizeof(maxblurSCH));
        for (center_x = x_max-5; center_x <= x_max+5; center_x++)
         for (center_y = y_max-5; center_y <= y_max+5; center_y++)
          partiald_acc(
           center_x,
           center_y,
      1, //sigma
      &maxradSCH[center_x][center_y],
      &maxblurSCH[center_x][center_y],
      Iris,
      rmin_t,
      rmax_t);

   //search maximum
   max_matrix(maxblurSCH, &CIX, &CIY);
   RI = maxradSCH[CIX][CIY];
   ////////////////////////////////////////////////////////////////////////

   //////////////////////// pupil search ////////////////////////
   memset(maxradSCH, 0, sizeof(maxradSCH));
   memset(maxblurSCH, 0, sizeof(maxblurSCH));
   for (center_x = CIX-5; center_x <= CIX+5; center_x++)
    for (center_y = CIY-5; center_y <= CIY+5; center_y++)
     partiald_acc(
      center_x,
      center_y,
      1, //sigma
      &maxradSCH[center_x][center_y],
      &maxblurSCH[center_x][center_y],
      Pupil,
      (int)round(0.1*RI),
      (int)round(0.8*RI));

   //search maximum
   max_matrix(maxblurSCH, &CPX, &CPY);
   RP = maxradSCH[CPX][CPY];
}
```

Every time the integrodifferential operator has to be performed in a pixel, the *partiald_acc* is called. The function wraps all the macros to set-up the peripheral. It receives as parameters:

- coordinate x pixel

- coordinate y pixel
- sigma
- pointer to the array that handle the radius output
- pointer to the array that handle the integrodifferential output
- type of search
- minimum radius
- maximum radius

The coordinates identify the point in which the operation is computed. The sigma is a parameter related to the convolution performed in the integrodifferential operator. It sets the accelerator to compute the convolution with a constant function or a Gaussian function. The two pointers are used to store the values returned by the accelerator. The type of search sets the computation using the iris or the pupil contour integral. The minimum and maximum radius define the range of the partial derivative. The first part of the function configures the peripheral. To set-up the accelerator, the function uses some macros that access to internal registers. Each register has its own purpose. The first step is to reset the accelerator. XPS assigns for each module in the design a base address that allows to control and monitor each module. The accelerator has a pre-built software reset, a feature that allows to reset the module writing in a specific register in the space address of the accelerator. The macro *AC_mReset* receives as input the base address of the accelerator and sets to 1 the register located at the offset 0x00000030. Then, the base address of the input image is written in the register 8. Accordantly to the type of search, two different base address are written in the register. The iris and pupil search don't perform the same circular integral. This choose is driven by some optimization due to the physiology of the edge between sclera and iris, and between iris and pupil. If we look closely a typical eye image, it's possible to see that the border of the iris is more pronounced in the lateral contour of the iris. The upper and bottom parts are more subjected to noise due to the eyelids. For this reason only a partial contour integral is computed. On the other hand, a fully contour integral is computed for the pupil. The Figure 5.1 shows in green the iris's contour integral, in blue the pupil's contour integral. To compute the contour integral, the accelerator requires polar coordination to locate

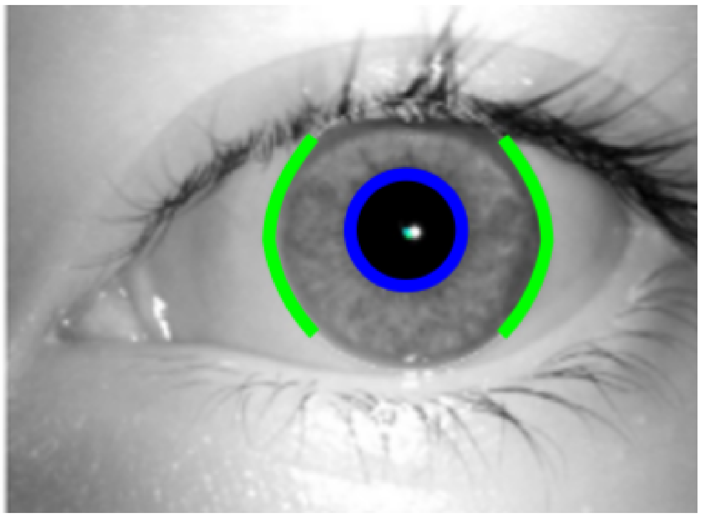


FIGURE 5.1. In green the iris's contour integral, in blue the pupil's contour integral.

the different points in the border. To improve the performance of the module, the coordination are not every time computed by the module itself, but are pre-computed through MATLAB and stored in an header file. In this way, the accelerator has to simply access to the arrays to access to any point on the contour. Two array are used to store polar coordination: one for the cosine and one for sine. Each of them provides the corresponding Cartesian coordinate x and y to access to each pixel. Going back to the code, the type input parameter specifies which polar coordination bases address have to be passed to the accelerator. Finally, the coordination x and y of the selected point, the r_{min}, r_{max}, sigma parameters are written in dedicated registers. At this point the set-up stage ends and the accelerator is started writing 1 in the register 0. After the start, the function waits until the module finishes to compute the results, checking if the register 7 is equal to 4. The figure shows the fields of the register 7. When the module terminates the computation,

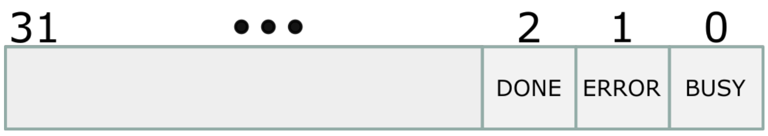


FIGURE 5.2. Fields of the register 7.

the integrodifferential and radius results are read from registers 3 and 4 and stored in the corresponding pointers. Next the *partiald_acc* function

```
void partiald_acc(int center_x, int center_y, int sigma, int* rad_max,
                                int* conv_max, int type, int rmin, int rmax){
        AC_mReset(XPAR_AC_0_BASEADDR);
        AC_mWriteReg (XPAR_AC_0_BASEADDR, AC_SLV_REG8_OFFSET, image);
        if(type==Pupil){
                AC_mWriteReg (XPAR_AC_0_BASEADDR, AC_SLV_REG9_OFFSET, sin_coord_pupil);
                AC_mWriteReg (XPAR_AC_0_BASEADDR, AC_SLV_REG10_OFFSET, cos_coord_pupil);
        }
        else if(type==Iris){
                AC_mWriteReg (XPAR_AC_0_BASEADDR, AC_SLV_REG9_OFFSET, sin_coord_iris);
                AC_mWriteReg (XPAR_AC_0_BASEADDR, AC_SLV_REG10_OFFSET, cos_coord_iris);
        }
        AC_mWriteReg (XPAR_AC_0_BASEADDR, AC_SLV_REG1_OFFSET, center_x);
        AC_mWriteReg (XPAR_AC_0_BASEADDR, AC_SLV_REG2_OFFSET, center_y);
        AC_mWriteReg (XPAR_AC_0_BASEADDR, AC_SLV_REG5_OFFSET, rmin);

        if(rmin>=rmax){
                *rad_max=rmin;
                *conv_max=0;
        }
        else
        {
                AC_mWriteReg (XPAR_AC_0_BASEADDR, AC_SLV_REG6_OFFSET, rmax);
                AC_mWriteReg (XPAR_AC_0_BASEADDR, AC_SLV_REG14_OFFSET, (Xuint32)(sigma));
                AC_mWriteReg (XPAR_AC_0_BASEADDR, AC_SLV_REG11_OFFSET, type);
                AC_mWriteReg (XPAR_AC_0_BASEADDR, AC_SLV_REG0_OFFSET, (Xuint32)(1));
                while(!(AC_mReadReg (XPAR_AC_0_BASEADDR, AC_SLV_REG7_OFFSET) == 4));
                *rad_max=AC_mReadReg (XPAR_AC_0_BASEADDR, AC_SLV_REG3_OFFSET);
                *conv_max=AC_mReadReg (XPAR_AC_0_BASEADDR, AC_SLV_REG4_OFFSET);
        }
}
```

5.2 Hardware-Verilog side

The peripheral is made by three main blocks. The slave interface, the master interface and the main core, that implements the computation. The figure 5.3 shows the main overview of the peripheral. The slave and master interfaces are generic template, provided by Xilinx, handling

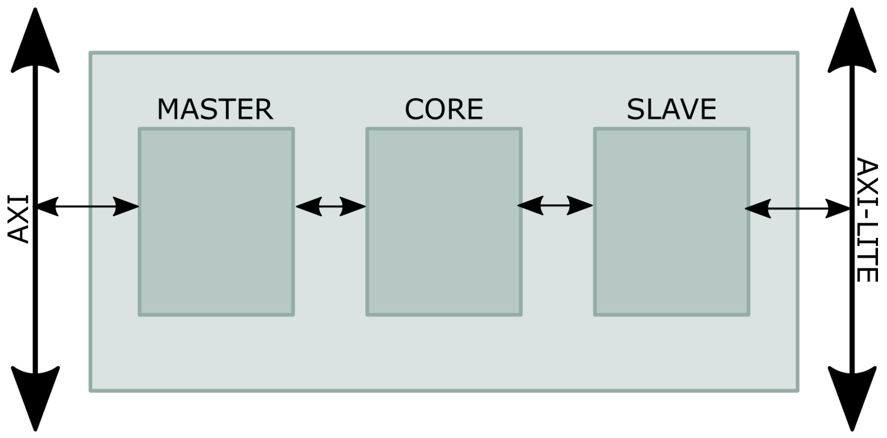


FIGURE 5.3. Overview of the integrodifferential operator.

the protocol for write and read operations. A detail explanation of the two interfaces is out of this project because it will require too much time. In this paper, I'm going to describe only the remarkable changes on them. The slave interface allows the processor to send and receive data to/from the module. The AXI-LITE protocol is used for this interconnection. The communication is achieved by reading and writing registers in the slave interface. The control signals are connected between the main core and the slave interface in order to set-up and start the computation.
The master interface implements the read from the space address of the DRAM in which is defined the input image and polar coordination. The communication is performed through the AXI protocol, that achieves higher performance with respect to AXI-LITE.
The hardware accelerator is comprised of two levels of state machines. The first layer generates the line integral of the curve for a given center coordinates and radius, while the second layer generates the values from r_{min}-r_{max} and passes those to the first layer state machine. The second layer also stores the integral values, generates the differentials and calculates the 1D convolution as required by Dougman et. al. [1]. Figures 5.4 and 5.5 below show the structure of the state machines implemented in the hardware accelerator. The first layer receives from the 2nd layer

the center point's coordination x and y and the radius as well. To computes the coordination in the line integral, the FSM uses three steps in which requires the polar coordination x and y and finally grabs the pixel from the image at that coordination. The FSM repeats the request of the coordination to the memory for all the N points. All the values received during the state *wait_v* are accumulated and finally sent back to the layer 2. The 2nd layer starts and waits the the 1st layer's results and send them to the convolution unit. It also saves, in a dedicated register, the maximum value computed by the convolution unit. When the accelerator ends the entire process, it sends the END signal, the maximum convolved value and the radius related (connected to the registers 7, 4 and 3 respectively) to the slave interface. Figure 5.7 shows the hardware

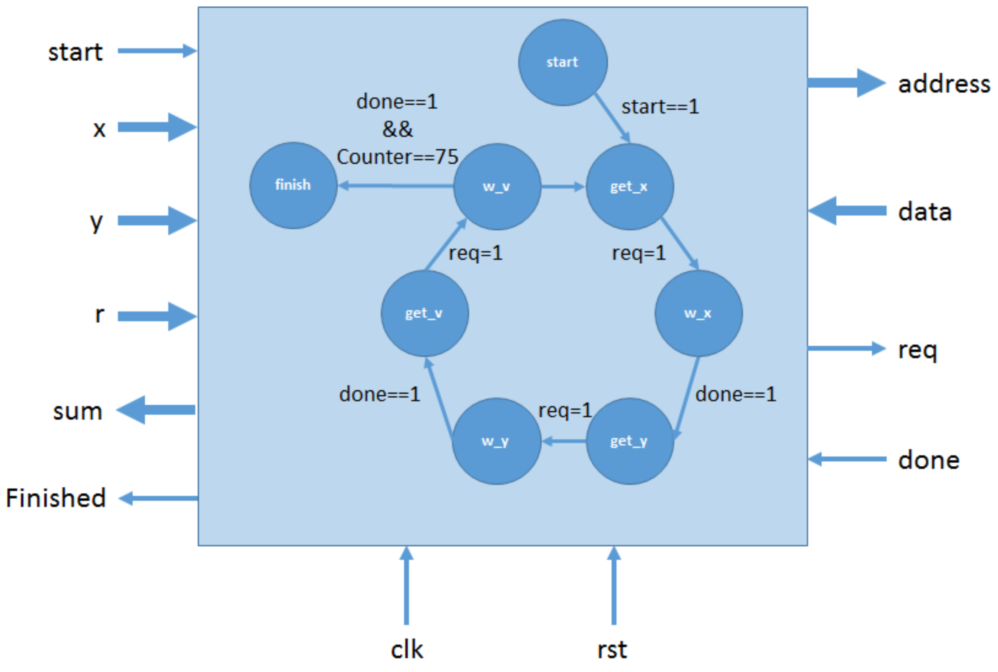


FIGURE 5.4. 1st Layer State Machine for Integrodifferential Operator.

structure for the convolution implementation. As shown in the figure, the result from the lower level state machine is pushed through a shift register while any given time, the subtraction, the multiplication, and the accumulation are carried out for the current state of the shift register. For storing the value of the filters on the other hand, a register is utilized. The hardware ad-hoc convolution unit ha the big advantage to be less time consuming with respect to the software approach. This improvement is provided by giving up the use of the floating point precision by

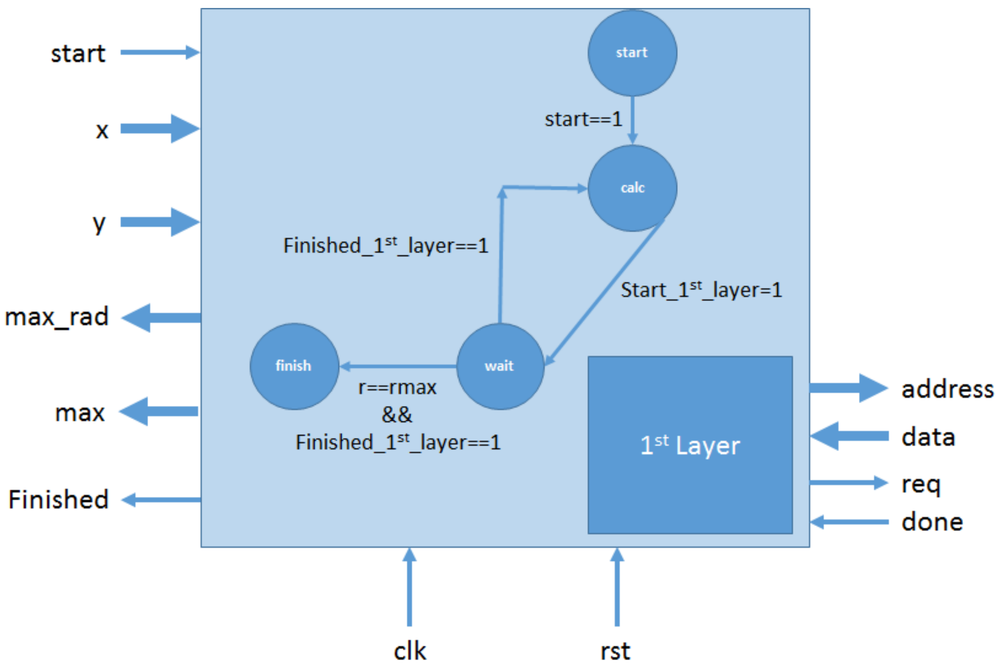


FIGURE 5.5. 2nd Layer State Machine for Integrodifferential Operator.

using the processor. In order to maintain a decimal precision in the result, a fixed point precision is adopted. To maintain a good trade-off between integer range and decimal precision, a deep exploration has been carried out. The figure shows the final arrangement, in which a thirty-two bits register is split in two fields, three bits are dedicated to the integer part and the remaining twenty-nine bits to the decimal part.

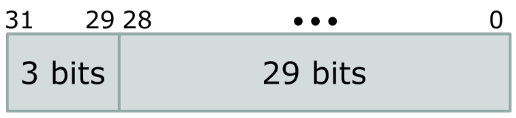


FIGURE 5.6. Fields of register in fixed precision.

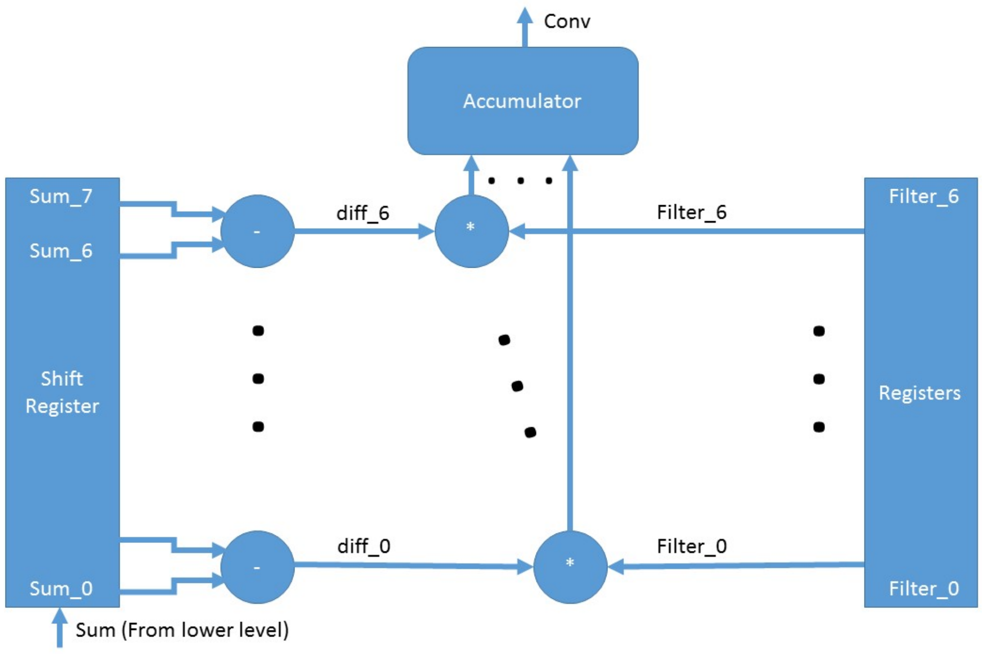


FIGURE 5.7. The schematics of the convolution circuit.

CHAPTER 6

HARDWARE ACCELERATOR FOR THE FOCUS ASSESSMENT

As shown in the Figure 3.3, the second algorithm more time consuming is the focus assessment. The focus assessment identifies the best focused frame in a steam captured at frame-rate equal to 10. The fastest solution adopted in this case, to identify the more focused frame, is to compute the convolution with a pre-defined kernel as suggested by Dougman et. al. [**?**]. To further run-time's reduction, the convolution is computed in a restricted portion of the frame, called ROI (Region Of Interest). After some exploration, a ROI of 50% is chosen guaranteeing the same results, cutting of one half the runtime. To reduce drastically the run-time, a co-design SW/HW is adopted. The unit has a slave and a master port connected to AXI-4 Lite and AXI-4 interconnect respectively. The slave port receives the destination address from the Microblaze module. The master port is devoted to the communication with the DDR3. The energy computed by the accelerator is 64 bits width, so it is stored in two slave registers. The MCB retrieves the energy reading those two registers through the slave port.

The function that controls the focus assessment accelerator is called *focus_assessment_by_accelerator*. It updates the base address so that the module starts from the first position in the ROI. Then it resets the module through the software reset feature provided by the accelerator. It writes the base address in the register 8 and starts the module setting to 1 the register 0. Then, it ways that the module ends checking the register 7. Finally stores the energy computed in the two variables devoted to keep the highest and lowest part of the 64 bits result. Next, the function

```
void focus_assessment_by_accelerator(int* image, Xuint32 *sumH, Xuint32 *sumL){
        Xuint32 address;
        address=(Xuint32)image;
        address+=( (ROWS)*COLS_ORIG + COLS )*4;

        CONV_mReset(XPAR_CONV_0_BASEADDR);
        CONV_mWriteReg (XPAR_CONV_0_BASEADDR, CONV_SLV_REG8_OFFSET, address);
        CONV_mWriteReg (XPAR_CONV_0_BASEADDR, CONV_SLV_REG0_OFFSET, (Xuint32)(1));
        while(!(CONV_mReadReg (XPAR_CONV_0_BASEADDR, CONV_SLV_REG7_OFFSET) == 4));
        *sumH=CONV_mReadReg (XPAR_CONV_0_BASEADDR, CONV_SLV_REG1_OFFSET);
        *sumL=CONV_mReadReg (XPAR_CONV_0_BASEADDR, CONV_SLV_REG2_OFFSET);
}
```

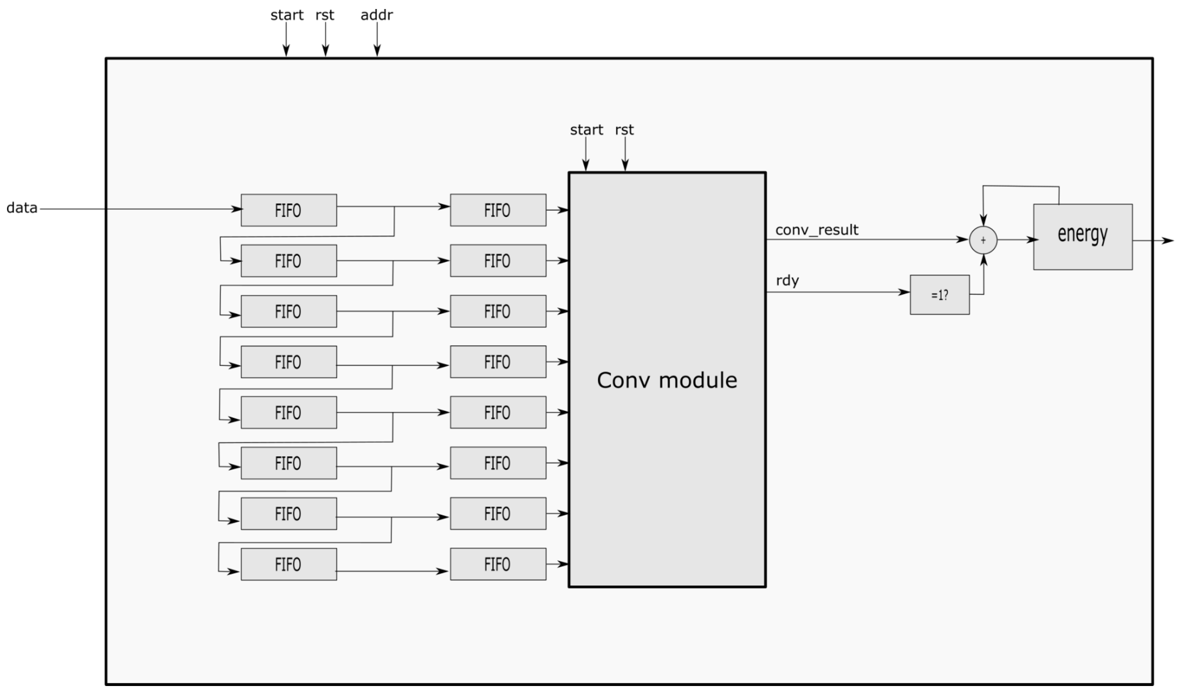


FIGURE 6.1. Architecture of the Convolution Accelerator.

The accelerator is operating at 30 MHz. The energy computation takes 0.08s for each frame. The Figure 6.1 shows the architecture of the module. It receives the data from the master port and fed the data in a linear buffer to maintain the consistence between the rows of the frame. The column of FIFOs that receives the data from the master port is updated every time the DDR3 provides the data. The Convolution module works at each clock cycle. For this reason, it requires an interface that exploits the second column of FIFOs that fed the module every clock cycle.

Figure 6.2 below shows the structure of the state machines implemented in the hardware accelerator. The first FSM manages the read request and the FIFO read and write. Going deeply, the two states *get_x* and *wait_x* are devoted to retrieve the data from the DRAM3. Meanwhile the values are fed in the linear FIFOs. As soon as all the FIFOs contain entire rows, the start signal for the convolution module is asserted. The last *finish* states are used to conclude the calculation of the convolution. The second FSM manages the convolution module. To compute the convolution in pipeline, the first process computes the convolution between the columns 1st and 8th, the second process between the 2nd and 9th, and so on. So that after 8 clock cycles the first result is ready and can start to compute the convolution between the columns 9th and 17th. A multiplexer provides the right result at each clock cycle according to the ready signal asserted by each process.

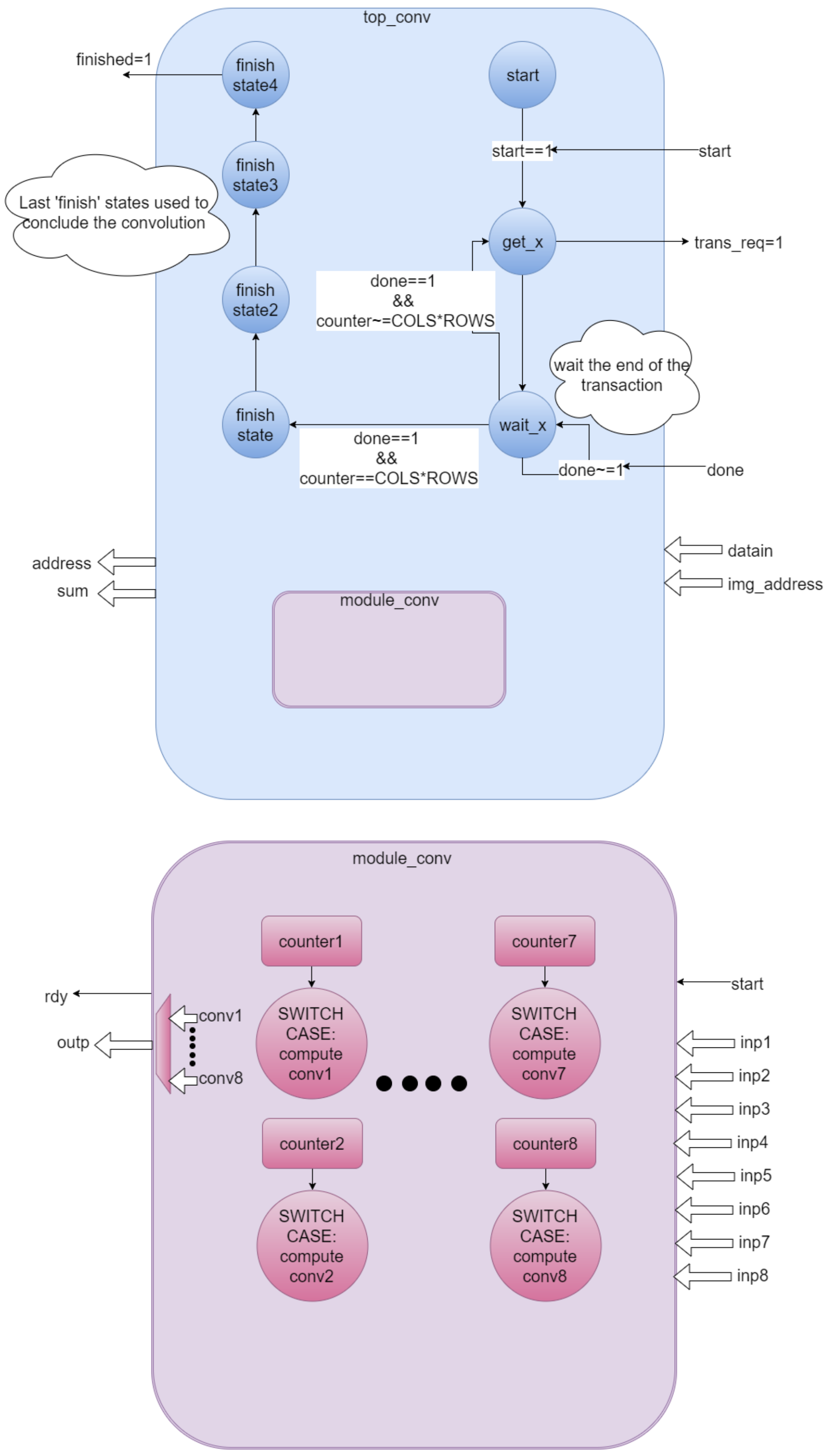


FIGURE 6.2. State Machine for Focus Assessment Module.

CHAPTER 7

Methodology for Approximate Computing based on Iris Scanning

7.1 Introduction

The realization of the co-design SW/HW implementation allows us to provide an architecture developed on the Spartan6 that satisfies successfully the constraints required by the company. Next, we decide to explore the error resilience of the design and identify some methodologies to minimize the response time. In this chapter I'm going to expose only my contributions on the paper, skipping the part in which I was less focused.

Approximate computing promotes the introduction of small degrees of inaccuracy into computing systems to achieve disproportionately significant reduction in computing resources such as power, design area, runtime or energy keeping the correctness of the output. We identify overall eight parameters at both the algorithmic and hardware levels to trade-off accuracy with runtime. The work integrates the use of approximate computing within the SW/HW flow, while performing a novel design space exploration of the SW/HW design parameters to identify their optimal settings. While many advances have been made in the approximate computing paradigm, most of the work evaluates the quality-energy trade-offs of a single module or algorithm in isolation. The goal is to minimize the response time to the user from the time of image capture to the encoding of the iris, under the constraints that (1) the encoding is accurate by industry standards, e.g. ISO/IEC 19794-6 consider an iris encoding which is represented 2048 bits as high quality even if the quality drops by 25% compared to the ideal case, because there is 1 in 13 billion chance to have a Hamming distance less than 25% between the irises of two different individuals, and (2) the resultant SoC can fit in the given resources of the logic device. Next, are listed the three main blocks that make up the design and the parameter identified in each of them.

- Focus assessment: As the energy of each frame is computed as a convolution of a kernel filter with the image, one obvious accuracy trade-off is the *kernel size* of the filter. Furthermore, instead of computing the focus assessment on entire frames, we can only compute the energy for a subset of the image; i.e., a region of interest (*ROI*), to further reduce the runtime.
- Iris segmentation: We identify six more accuracy knobs in iris segmentation stage. Here, *Npoints* in the number of points used to compute the differential in each circle; *Scale* is the resizing factor used to reduce the segmentation image resolution; *Thresh* represents the threshold beyond which a point is considered to be dark enough to be a candidate; *Rmin* and *Rmax* define the range of radiuses for which the integration is performed; and *Search Window Size* gives the size of the window around which the local iris and pupil searches are performed.
- Encoding: Lastly, in the encoding step, as the parameters providing accuracy vs. runtime trade-offs also affect the signature specification, in order to stay consistent, we refrain from changing any parameters in this step.

Table 7.1 summarizes the parameters evaluated in the design space exploration as well as their possible value sets. The proposed parameters result in a design space of $648,270$ design corners. Clearly a brute force exploration of the design space in not possible and a design space methodology is required for effective exploration. As evaluating all of the corners on hardware is not an option, we simulate and formulate the accuracy and the runtime re- spectively. Since the accuracy performance of one component in the pipeline greatly affects the other components and the final results, we have to estimate the accuracy of

Table 7.1: The list of parameters evaluated in the design space exploration. Values in brackets show the possible values.

Pipeline Accelerator	Parameters [List of values]
Focus Assessment	Kernel Size [8,6,4]
	RoI [1,0.78,0.50,0.33,0.20]
Iris Segmentation	Npoints [600,400,200,150,100,75,50]
	Scale [1,0.85,0.75,0.50,0.25,0.20]
	Thresh [102,90,77,64,51,35,26]
	Rmin [45,55,65,75,85,95,100]
	Rmax [180,170,160,150,140,130,120]
	Search Window Size [11×11, 7×7, 3×3]

a set of parameters using the entire flow through simulation. Thus, to compute the accuracy, we run a SW/HW co-simulation of the entire flow. To speed up the computationally demanding co-simulation, we use Verilator [**?**] to compile the Verilog-based hardware accelerators into C-based simulators, and then use gcc to compile all the components in software. Verilator is the fast free Verilog HDL simulator. It compiles synthesizable Verilog into C++ code and uses a test-bench in C++. Verilator is about 100 times faster than interpreted ISIM, the simulator provided by Xilinx.

Unlike accuracy, the runtime of the pipeline flow can be readily decomposed. To that end, we mathematically formulate the runtime based on the input parameter set. A summary of the runtime models is shown in Table 7.2. With the runtime formulated, we profiled some training sets of parameters to quantify the coefficients. We verified the formulation on another set of parameters demonstrating a modelling errors of less than 5%. Note that this runtime merely guides the design space exploration and will not translate into inaccuracies. The script devoted

Table 7.2: The formulation used to model the runtime behavior as a function of the design space parameters. Note that as we do not modify any parameters in the encoding components, we consider its runtime as a constant.

Pipeline Component	Runtime Model
Focus Assessment	$\propto RoI^2.KernelSize^2$
Iris Segmentation	$= R_{Coarse} + R_{Iris} + R_{Pupil}$
–Coarse Search	$\propto Scale^3.Thresh.Npoints.(Rmax - Rmin)$
–Iris Local Search	$\propto WindowSize^2.Npoints.(Rmax - Rmin)$
–Pupil Local Search	$\propto WindowSize^2.Npoints.r_{Iris}$
Total	$= R_{FocusAssessment} + R_{Resize} + R_{Segmentation} + R_{Encoding}$

to compute the runtime is written in python. After analysing the equations to determine the run time for each step, we identified four parameters to use to compute quickly the run-times. Next, the python script used.

```
...
RMAX_RMIN=(RMAX-RMIN)/(RMAX_ORG-RMIN_ORG)


A=161.974
B=20.740
C=11.562
D=2.990


R1=A*(pow(SCALE,3))*(THRESH)*(NPOINTS)*(RMAX_RMIN)
R2=B*(NPOINTS)*(RMAX_RMIN)*(pow(SEARCH_WINDOWS,2))
R3=C*(NPOINTS)*(pow(SEARCH_WINDOWS,2))
runtime=R1+R2+R3+D
...
```

7.2 Methodology

With an accuracy co-simulation flow and runtime models in place, the methodologies are exploited to navigate the exponential design space of the parameter set. One of them, used to identify the best approximate parameters, is the gradient descent model. The gradient descent is a very handy method for optimization. While gradient descent is often not the most efficient method, it is an absolutely essential tool to prototype optimization algorithms and for preliminary testing of models. Once the model formulation is stable, one might wants to invest more in considering better optimization methods.

For the gradient descent approach, we iteratively change each quality knob individually by one in order to generate a new design point. The design point with the highest gradient, where the gradient is defined by

$$\max_k \frac{\partial runtime(k)}{\partial quality(k)} \quad where \quad k \in \{KernelSize, ROI, Npoints, ...\}$$

Algorithm 1: Gradient Descent Method

```
// Params: List of tunable parameters with their possible values.
// Approx_Params: The set of resulting parameters.
Input   :Params
Output:Approx_Params
Initialize(Gradient Descent method);
while $acc\_curr < acc\_tresh$ do
    for $time\_step = 1$ to Number of Parameters do
        $ACC$ = getAccuracy();
        $RUN$ = getSimulatedRuntime();
        $GRAD_{time\_step}$ = getGradient($ACC$, $RUN$);
    end
    $BEST$ = getMax($GRAD$);
    if $max\_err <$THRESHOLD and $runtime < best\_rutime$ then
        $best\_RUN = RUN$;
        $best\_ACC = ACC$;
        $Approx\_Params = params$;
    end
end
return $Approx\_Params$
```

is chosen as the parent for the next iteration. The algorithm 1 describes the parameter sampling and gradient descent method. Firstly, the parameters are initialized. Then, for each time step, a new knob is chosen and computed the gradient. There are eight time steps, each one corresponding to a different knob. Once the parameters are chosen, we compute the runtime using the model and error rates through simulation. According to the highest gradient, the knob selected in that time step is selected and used for the next iteration. The algorithm is repeated until the error introduced is lower than the maximum threshold error.In order to assess accuracy, we cross validate the signature of each image in the dataset, using the approximated parameters, against all of the signatures of the same eye in the repository when computed with full quality. To ensure 100% accuracy in the design, if at any point the maximum hamming distance error of any two images from the same eye goes above 0.35%, the design is discarded.

7.3 Results

The first analysis is to evaluate the benefits achieved from mapping part of the computing pipeline into custom hardware accelerators. The results of the comparison between the runtime and hardware utilization of implementations full SW and HW/SW using the highest value parameters are shown in the Table table:runtime. Next, the gradient descent method was used

Table 7.3: The components chosen for hardware acceleration, the corresponding speedups, and the hardware utilization of each accelerator.

Pipeline Component	Speedup	HW utilization (%)
Focus Assessment	1234×	25.71
Iris Segmentation	6.8×	13.24
System		15.42

to discover sets of parameters that achieve better performance. Figure 7.1 shows the design points evaluate using the gradient method methodology. The method requires going through 100

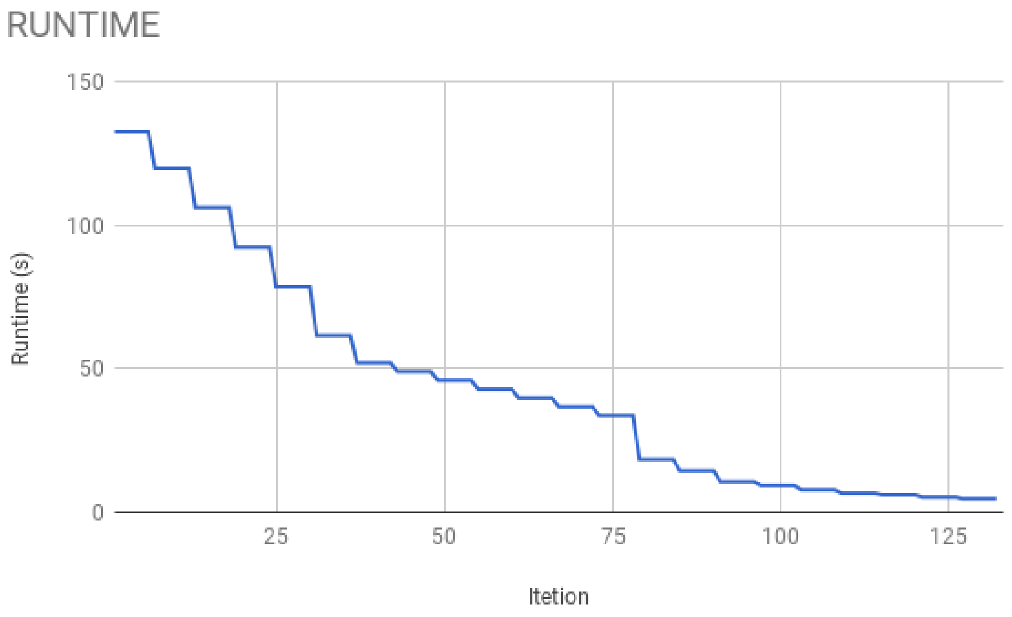


FIGURE 7.1. Architecture of the Convolution Accelerator.

exploration points before finding the best set of parameters that satisfies the error rate constraint. The run time achieved is 6.42 seconds, obtaining a 4921X speed-up in focus assessment module and 217X in the segmentation iris. The parameters obtained are shown in the Table 7.4. Finally,

Table 7.4: Parameter provided by the gradient descent method

NPOINTS	SCALE	ROWS	COLS	THRESH	RMIN	RMAX	S_W	K_S	ROI
100	0.5	240	320	26	45	120	5	6	0.5

we highlight the immense benefits of exploring the hardware/software co-design domain in conjugation with the approximate parameter exploration. Table 7.5 summarizes the results. As shown in the table, significant benefits are achieved in terms of both runtime, and the total energy. Compared to a pure SW implementation, the approximate SW/HW pipeline is able to achieve a speed-up of 378× while meeting the industry standard accuracy requirements. For the

Table 7.5: The hardware characteristics of the end-to-end system.

Design	Runtime (s)	HW (%) Utilization	Energy (kJ)	Memory (MB)
Pure SW	2419 .6	15.42	47.9	0.69
HW/SW	198.3	54.37	3.94	2.2
Approx. HW/SW	6.4	46.42	0.12	0.89

experiments we use a Spartan6 Xilinx development board interfaced to a 5 MP Videology camera with infrared optical filter and infrared LEDs for illumination. We also use an Agilent 34410A multimeter to monitor the current and measure power consumption accordingly. Figure 7.2 shows our camera and FPGA setup.

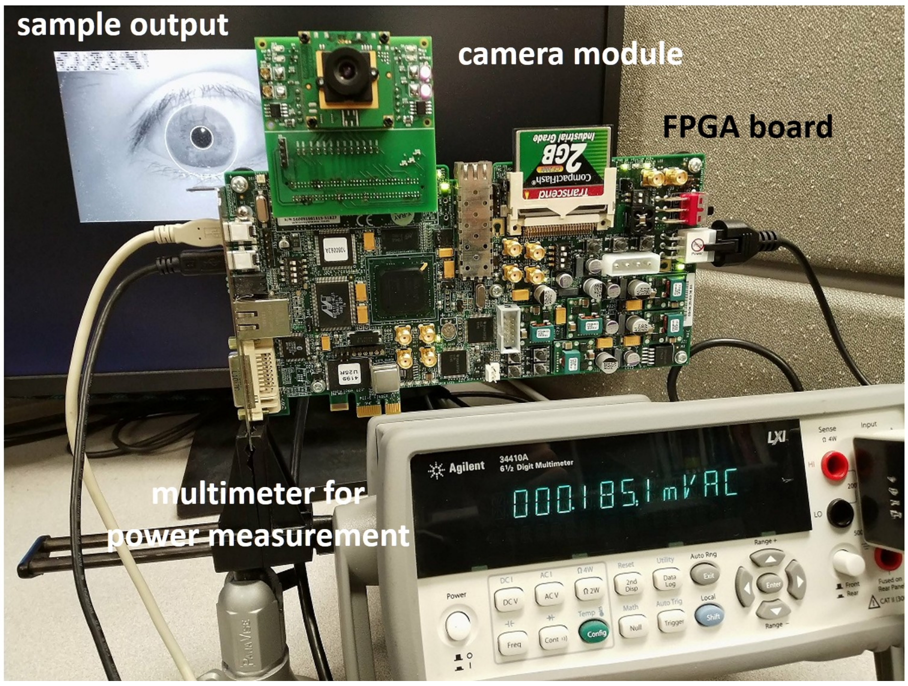


Figure 7.2: The camera and FPGA board Setup.

Part II

GBCSSM-Pinggera

Chapter 8

Algorithm review

In this is second part I'm going to describe another project that I have started working after three months from my arrive. The scope of this project was to develop an hardware architecture of the Gradient-Based Cross-Spectral Stereo Matching Algorithm published in [3]. The scope of this work was to start a first implementation to be published in a next paper showing the advantage of using a custom hardware design that can be mount in system that require high performance al low power consumes.

8.1 Introduction

The Gradient-Based Cross-Spectral Stereo Matching technique is used to generate disparity maps that allow to identify main features given 2 similar images. The goal of the algorithm is to recognize a particular context providing two images capturing the same scenario but in different conditions. Typical application are aircraft search rescue and all other application in which is required recognize the context from captured images. The visual stereo matching provides low level of detail comparing images in weather condition really different. Pixel-based stereo matching of visual input images is a well known process that offers good matching results but when attempting to perform pixel-based comparisons among visual and full-infrared images, the results are poor at best due to the variable nature of pixel intensities. To exploit a visual and full-infrared stereo matching, a gradient-based cross-spectral stereo matching (GB-CSSM) algorithm introduced by Pinggera provides reliable results.

8.2 The Gradient Based Cross-Spectral Stereo Matching Algorithm

The Pinggera's algorithm converts the input images in HOG (Histogram of Oriented Gradients), that extracts the relevant information, and then, compared through a Distance Operator, produces a result that keep only the significant features. The Figure 8.1 shows the main overview of the algorithm. The two HOG fed the Distance Operator that extracts the Cost Matrix from which we obtain a disparity map. Finally is applied a Optimization Step (SGM) that highlights the features of the Cost Matrix. The Figure 8.2 shows all the components required to generate a

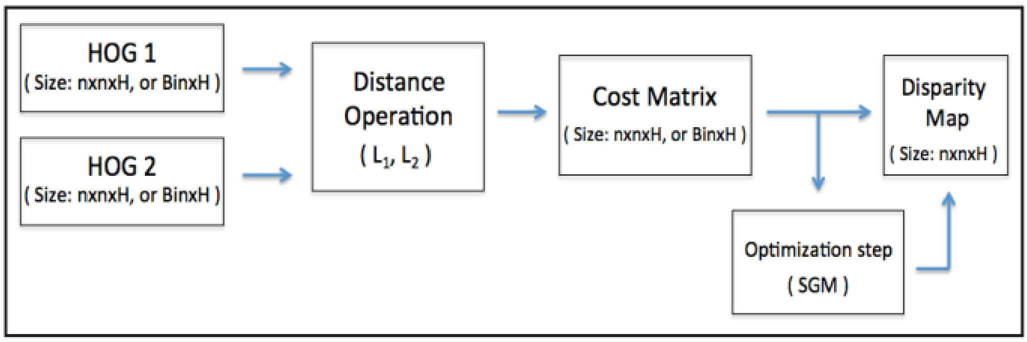


Figure 8.1: Cost Matrix and Disparity Map generation.

HOG descriptor per input image. A magnitude gradient is the basic element in every HOG. Mathematically, it refers to the result obtained by the square root of the sum of squared gradient components, where the number of gradient components varies depending on the number of coordinate axes under evaluation. Each of the gradient components is obtained convolving every pixel in the input image I with kernels h. The final HOG is obtained dividing the magnitude matrix by the norm of entire matrix. All the steps the algorithm are related to some input paramenters that directly affect the output disparity maps produced. These parameteters are reletated to different parts of the flow, such as the type of kernel used, the number of coordinations,

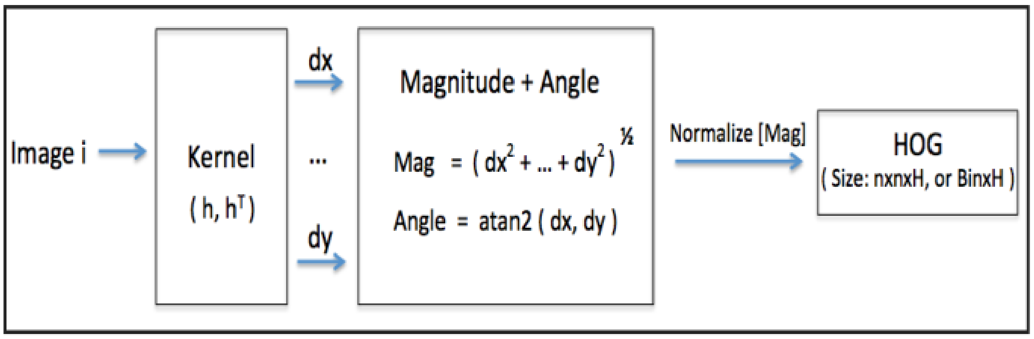


Figure 8.2: Cost Matrix and Disparity Map generation.

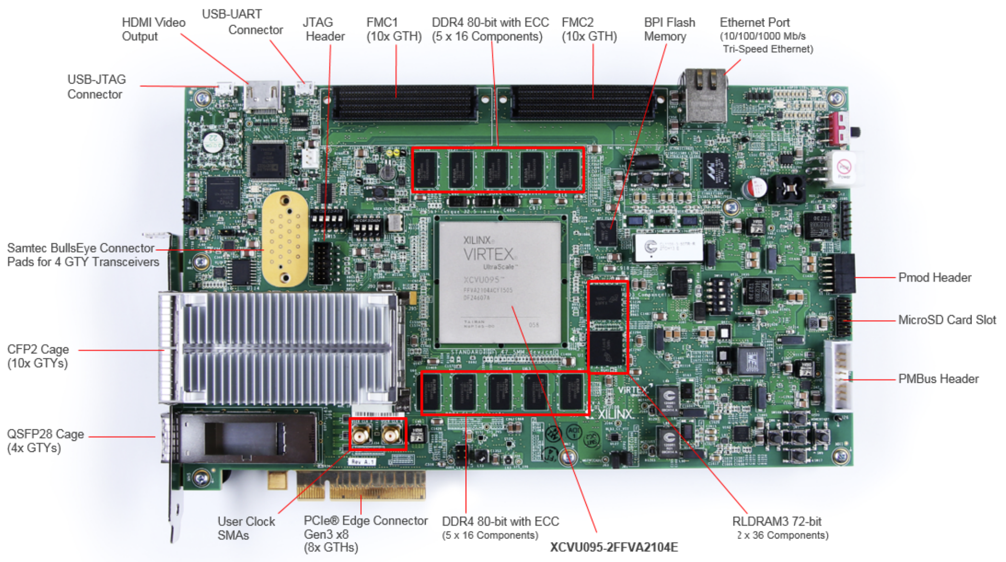
the kind of magnetude and ect. The main goal of the paper [3] was to identify a set of input parameters able to produce the more meaningful disparity maps. The task of my work was to use one set of input parameters that produces the most meaningful disparity map and build an hardware implementation embedding those in the architecture.

CHAPTER 9

ARCHITECTURE

9.1 System Architecture Setup

The board used in this project was Xilinx Virtex UltraScale FPGA VCU108 shown in Figure 9.1. The main features of this development board are listed below.



HDMI Video Output
USB-UART Connector
JTAG Header
FMC1 (10x GTH)
DDR4 80-bit with ECC (5 x 16 Components)
FMC2 (10x GTH)
BPI Flash Memory
Ethernet Port (10/100/1000 Mb/s Tri-Speed Ethernet)
USB-JTAG Connector
Samtec BullsEye Connector Pads for 4 GTY Transceivers
CFP2 Cage (10x GTYs)
QSFP28 Cage (4x GTYs)
Pmod Header
MicroSD Card Slot
PMBus Header
User Clock SMAs
PCIe® Edge Connector Gen3 x8 (8x GTHs)
DDR4 80-bit with ECC (5 x 16 Components)
XCVU095-2FFVA2104E
RLDRAM3 72-bit 2 x 36 Components)

FIGURE 9.1. Picture of a Xilinx Virtex UltraScale FPGA VCU108 development board.

- Virtex UltraScale XCVU095-2FFVA2104E FPGA
- Zynq XC7Z010 based system controller
- Two 2.5 GB DDR4 component memory interfaces
- 1 GB BPI flash memory
- USB JTAG interface with micro-B USB connector
- USB-to-UART bridge with micro-B USB connector

The main resources of the FPGA are in the Table 9.1. For further information about the fpga itself [7] and [4]. The project was built using Vivado 2017.2, a bunch of tools that

Table 9.1: Available resources in Virtex UltraScale XCVU095-2FFVA2104E FPGA.

Device	Logic Cells (K)	Flip-Flops	Max Distributed RAM (Kb)	Blocks RAM of 36 Kb
XCVU095	1,176	1,075,200	4,800	3,456

provides specific flows for programming. The Vivado IDE uses the IP integrator with graphic connectivity screens to specify the device, select peripherals, and configure hardware settings. Xilinx provides integration between a hardware design and the software development with an integrated flow called Software Development Kit (SDK). Figure 9.2 shows the main tools with GUI used in this project. The system is tested in two steps. The entire work-directory is

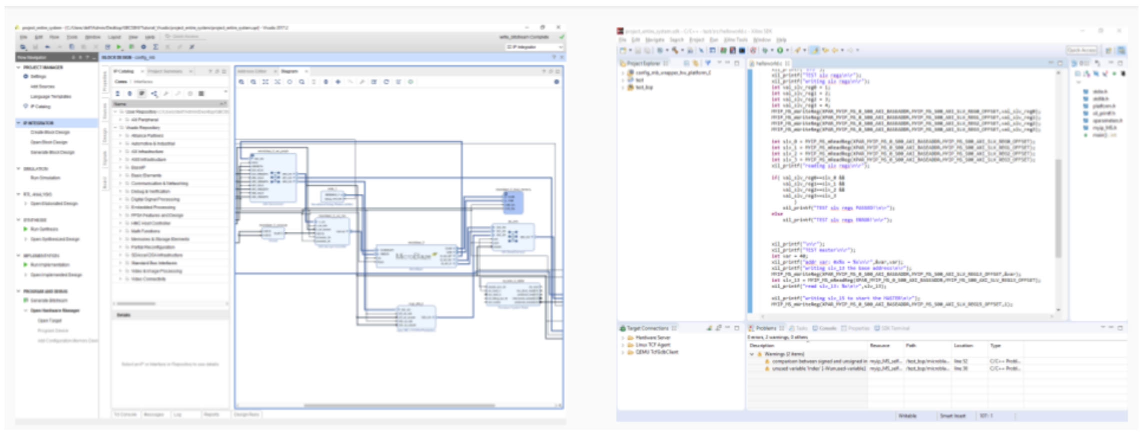


FIGURE 9.2. Vivado IDE on the left, SDK on the right.

split in two folders, one that contains the entire design, the second only the custom peripheral.

The latter project is used to develop and test only the peripheral. To test it, the project uses the AXI Verification IP (VIP) core, that has been developed by Xilinx to support the simulation of customer designed AXI-based IP. It is a handy tools to simulate modules connected to AXI bus. The custom peripheral has two ports, the slave one connected to AXI-LITE, the master one to AXI. The Figure 9.3 shows the architecture used to test the custom peripheral. The custom

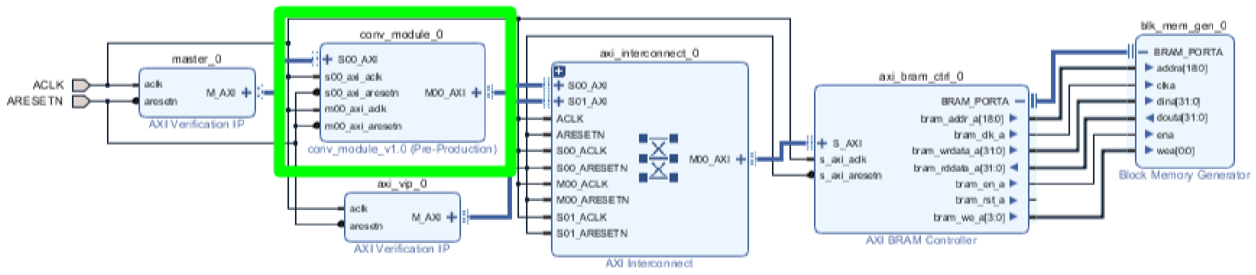


FIGURE 9.3. Architecture for Testing.

peripheral is highlighted with a green box. Its master port is connected to the block memory generator that simulates the DRAM. The AXI_master simulates the processor interface. The test-bench initializes all these blocks and sets the commands that the AXI_master will execute. About the entire design's project, it contains all the modules that are synthesized in the board. This project cannot be simulated due to the component in board constraints. Moreover, every time the bit-stream has to be created, the process requires more or less one hour, that doesn't allow a feasible test procedure. The Figure 9.4 shows the main architecture's overview. The design uses a Microblaze softcore processor as the main controller. It runs at 100MHz. The clock is generated by the System Reset Module, that manages the reset signals for all the modules and creates the clocks used in the design. In fact, the system receives a clock at 300Mhz and from it are generated two different clocks: one that drives the Memory controller Interface at 307MHz, the other one connected to the other components in the design at 100Mhz. The Memory Controller Interface translates all the requests read/write from the AXI-protocol to DDRAM transfers. The custom peripheral that implements the GBCSSM algorithm is connected to the Microblaze through the AXI-Lite interconnect module and to the MCI through a AXI interconnect module. The UART

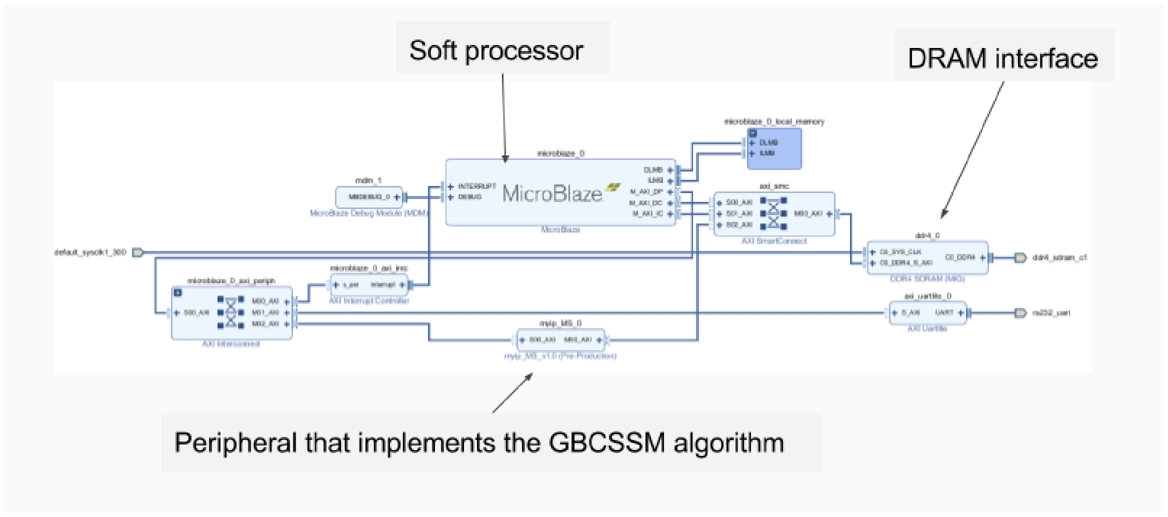
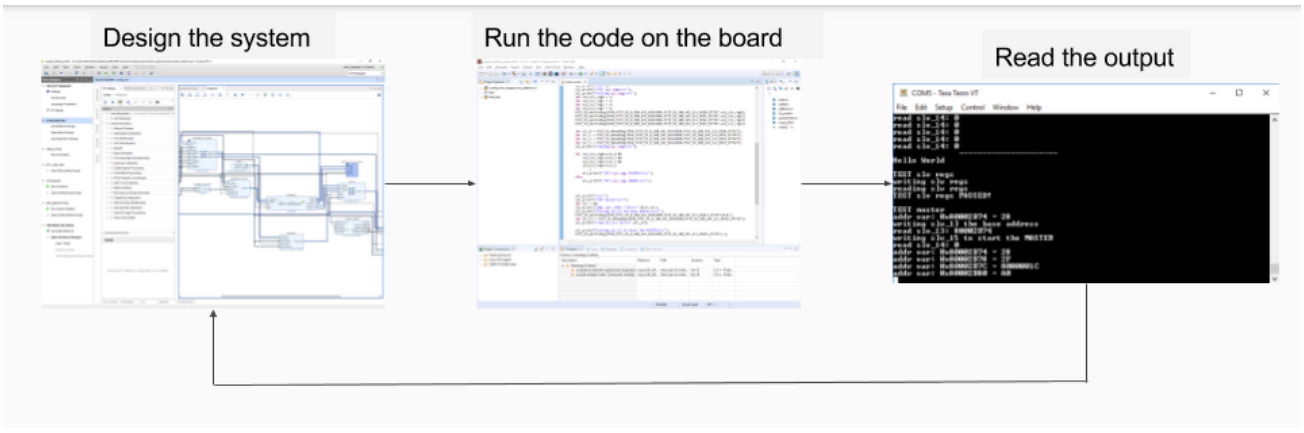


FIGURE 9.4. Architecture of the design-in-board.

interface uses a bit-rate of 234000 and it is used to return in a console the final Disparity Map. The Figure 9.4 shows the flow of testing.



```
read slv_14: 0
read slv_14: 0
read slv_14: 0
read slv_14: 0
read slv_14: 0
          ------------------------
Hello World

TEST slv regs
writing slv regs
reading slv regs
TEST slv regs PASSED!

TEST master
addr var: 0x00002974 - 28
writing slv_13 the base address
read slv_13: 00002974
writing slv_15 to start the MASTER
read slv_14: 0
addr var: 0x00002974 - 28
addr var: 0x00002978 - 2F
addr var: 0x0000297C - 0000001C
addr var: 0x00002980 - 00
```

FIGURE 9.5. Flow of testing.

9.2 System Generator Implementation

The first hardware implementation of the algorithm was developed using System Generator Vivado. System Generator is a DSP design tool from Xilinx that enables the use of the MathWorks model-based Simulink design environment for FPGA design. Designs are captured in the DSP friendly Simulink modeling environment using a Xilinx specific blockset. These blocks include the common DSP building blocks such as adders, multipliers and registers. This tool simplified the design's implementation because it provides Matlab's tools that speed-up the time devoted in the realization of the design. In this way, the efforts were spent on the implementation of the modules in Verilog, meanwhile the collateral modules and interconnections were built in Simulink. The Figure 9.6 shows the design implemented in System Generator. The operations

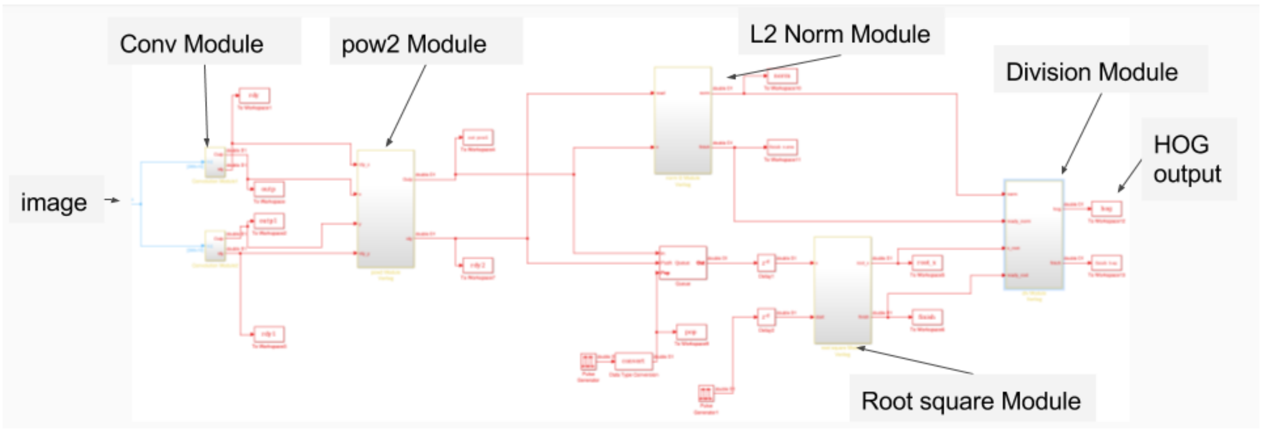


FIGURE 9.6. GBCSSM Design implemented in System Generator.

performed by the pipeline are:

- Convolution in Two Coordinate Axes
- Magnitude Matrix
  - Sum of squared convolution results $x^2 + y^2$
  - square root
- L2 Unit Norm
- HOG matrix computation

9.2.1 Convolution

The image is firstly converted to a grey scale image and then fed two convolution modules with kernel h and h^T respectively. The size of the sample image is 288x340 but, for simulation time constraints, only the first 10 rows are used. The simulation time, for the entire image, is about 16 minutes. For this reason a limited number of rows are used. The results are compared with a Matlab script at each step of the pipeline to be sure to compute a consistent result with the original algorithm. The Figure 9.7 shows this first step. The Input From Image and Colour Space Conversion are modules provided by Simulink. The Convolution Module is aa wrappers that contain the interface to fed correctly the data for the convolution unit. The Figure 9.8 shows the

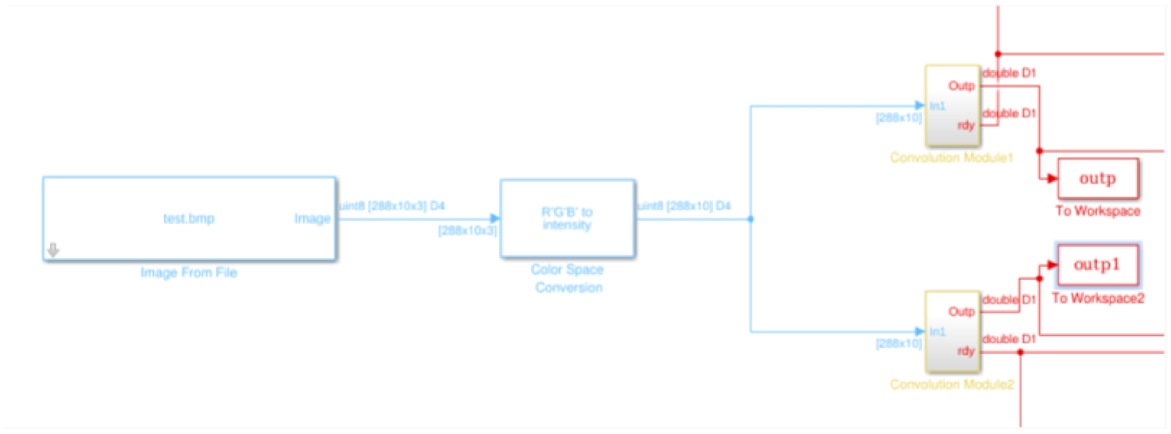


FIGURE 9.7. First Step of the pipeline.

internal implementation of the Convolution Module. The image received as a matrix is passed to Padding Module that adds a padding of zeros around the image. This step is required to replicate the result obtained by the convolution function used in the original algorithm. Next, the image is flatted to replicate the stream receive by the hardware implementation. In fact, the image is stored as a long array in the memory and the module receives only the base address, so that it has to manage the index processing. To compute the convolution, a kernel of 3x3 is used. To keep the module pipelined, a linear buffer is used. In each buffer is stored one entire row, made up of 288 pixels. As soon as all the FIFOs are full, the start signal is asserted. In this case, the signal is generated through the Step signal Simulink's module. The values from the FIFOs fed the Convolution Unit, that contains the implementation in Verilog of the convolution unit. The unit receives also the nine kernel's parameters as Constant Simulink's components. The unit has 3 main processes and each of them, every 3 clock cycles, computes the result for three corresponding set of input values. The Figure 9.9 shows how the convolution is performed. The three rows are fed

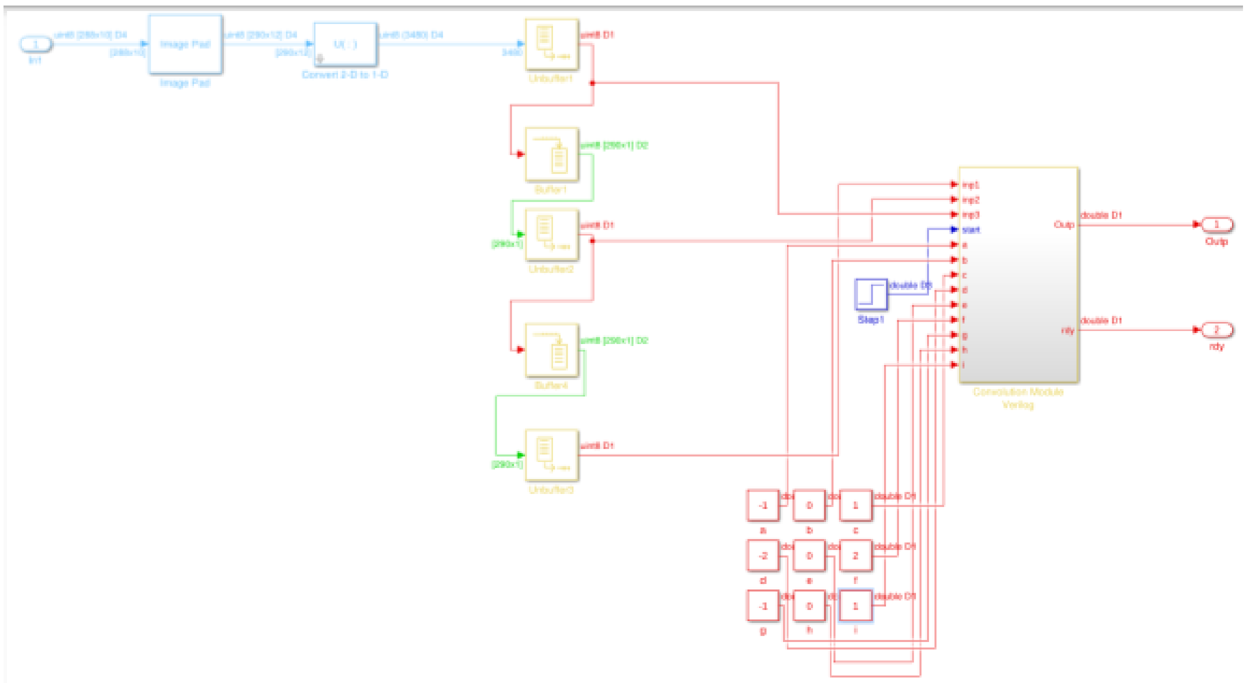


FIGURE 9.8. Convolution Module.

in parallel to the module, multiplied by the corresponding kernel's parameters and accumulated. After three clock cycles, all the nine multiplications are performed and accumulated, providing the result and asserting the ready signal. Due to the pipeline architecture, each process computes the convolution using input values shifted by one position, providing a new result every clock cycle.

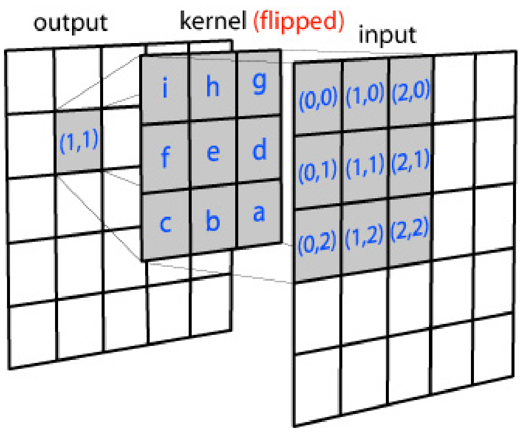


FIGURE 9.9. 2D Convolution.

9.2.2 Sum of Squares

The next step is the sum of the results computed by the two Convolution Modules. The Figure 9.10 shows the Sum of Square Module. It receives the input values and the relative ready signal. The module is a combination process that perform the following operation:

$$x^2 + y^2$$

The module computes the result if both ready signals are asserted. The module has two output signals: the result and the ready signal.

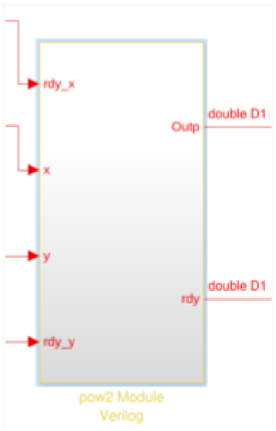


FIGURE 9.10. Sum of Squares Module.

9.2.3 Square Root

The Square Root Module computes the Square Root for each values provided by the Sum of Squares module. The Figure shows the architecture of this step. The Verilog implementation of the Square Root requires 65 clock cycles to compute the result. It is an successive approximation process that identifies the square root which its square is closest to the input. The main section of the Verilog code that performs the root square is shown below.

```
...
reg [31:0] root_x = 32'b0000000000000000_0000000000000000;  // binary point at bit 16
reg [31:0] trial_root = 32'b1000000000000000_0000000000000000;  // binary at at bit 16
assign x_test = {2'd0, x, 30'd0 };
assign x_less_than_trial_root_squared = (x_test < trial_root_squared ) ? 1'b1 : 1'b0;
```

```
...
always @ (posedge clk)
begin
    if (!ce || !start)
        begin
        trial_root <= 32'b1000000000000000_0000000000000000;
        finish <= 0;
        end
    else
    begin
        if(count==6'b111111)
            begin
            finish <= 1;
            root_x <= trial_root;
            trial_root <= 32'b1000000000000000_0000000000000000;
            end
        else
            begin
            finish <= 0;
            if (x_less_than_trial_root_squared)
               trial_root<=trial_root^(32'b11000000000000000000000000000000>>count);
            else
               trial_root<=trial_root^(32'b01000000000000000000000000000000>>count);
            end
    end

end
...
```

The register *trial_root* is loaded with value 32'b1000000000000000_0000000000000000 on the rising edge that makes count = 6'b111111. Durring the time count = 6'b111111 it is determined whether or not bit 31 (the MSB) of *root-x* should be 1 or 0. The register *trial_root* is updated on the next rising edge of clk and at that time has the correct bit 31 (MSB) and bit 30 is set to one in anticipation of the test for bit test on the next rising clock edge. On each subsequent clock edge one more bit is determined.

The input values is encoded using 32 bits. The result is a 32 bits value with 16 bits for the fractional part. Due to this delay, a queue is used to keep all the values provided by the Sum of Squares Module. A pulse Generator asserts the queue's pop signal every 65 clock cycles.

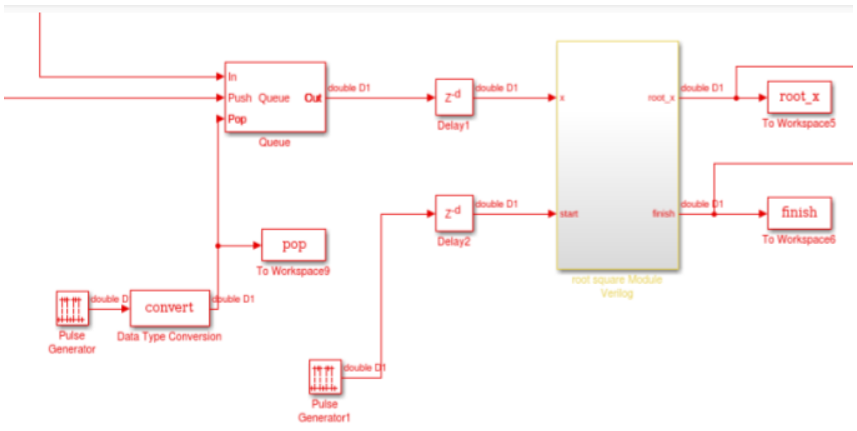


FIGURE 9.11. Square Root Module.

9.2.4 L2 Unit Normalization

The L2 Unit Norm performs the L2 Normalization of all the Sum of Squares results. The Figure 9.12 shows the module present in the design. The module performs the following operation

$$\sqrt{\sum_i x_i}$$

Actually, the equation shown above is different from the typical L2 Norm equation because, in this case, the x is not squared. In this architecture the square x is extracted and computed separately because it is shared among multiple modules. This choice achieve higher speed/lower area occupied having modules less complex. The module is internally split in two process. The

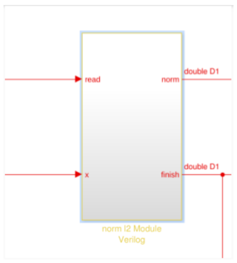


FIGURE 9.12. Square Root Module.

first one is devoted to accumulate the input values, the second one computes the square root of it. The latter uses the same successive approximation process described in the Section 9.2.3, but the result is stored in 64 bits. In this case the fractional part used is 32 bits. The module requires $rows\_im * cols\_im + 128$ clock cycles to finish.

9.2.5 Division (Final HOG Matrix)

The final step computes the HOG Matrix performing a division between root square value (16.16) and the norm L2 value (32.32). The result preserves the 30 bits precision for the fractional part. To compute a division on 32 bits, from the norm value, expressed in 64 bits, are extracted the first 16 bits for the integer part and the first 16 bits for the fractional part, obtaining a 16.16 format. This reduction in size was possible by some exploration of the integer and fractional ranges. To obtain a result with 30 bits for fractional part, the root square value is left-shifted by 30 positions, then divided by the norm L2 value. The shift step is required because in the division, by definition, the result's format will contain the difference of the bits in the operands' format. Next, the steps to obtained the desired resolution are shown.

$$(16.16 << 30)/16.16 \Rightarrow 18.46/16.16 \Rightarrow 2.30$$

The final resolution satisfies all the possible values in the HOG matrix because the values are always between 0 and 2.

9.3 Hardware Implementation using Vivado

The architecture implemented using System-Generator allows to test all the modules in Verilog and to replicate the original algorithm without focusing on the interfaces. The advantage of this step was to focus on the functionalities of the module and don't take in account problems related to memory allocation, bus transfer, clock constraints, etc. After testing the correctness of the architecture, the next step was to implement a custom peripheral and create a design that runs in the FPGA. The architecture is made up by 3 main cores: a soft processor Microblaze that controls the entire system, a Memory Controller Unit that performs transfer transactions with the external DDRAM and a custom peripheral that generates the HOG matrix. The last

version developed had the entire flow to create the HOG matrix. This part was the most complex to develop, because the GBCSS algorithm requires two HOG matrices and a Distance operator, easier to develop with respect to the HOG pipeline.
The Figure 9.13 shows the architecture implemented. Three main bus are used: AXI-4, AXI-4 Lite and LMB (local memory bus). The LMB bus is used for transactions between Microblaze and the

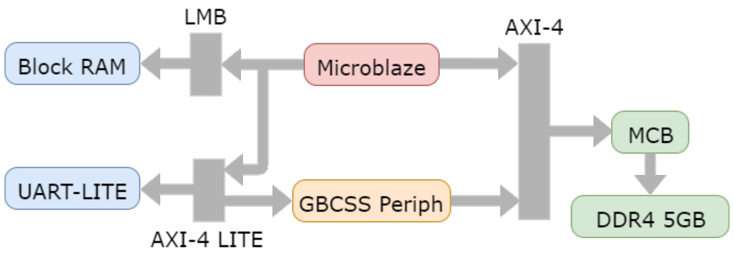


FIGURE 9.13. Overview architecture for GBCSS implementation.

on-chip block RAM, responsible for data and instruction caching. The AXI and AXI-Lite runs at 100MHz. The AXI-Lite bus is used for single transfer communications, used for communications with the peripherals. The AXI bus allows high throughput communications necessary for accessing the DDR4. It provides the Transfer Burst mode, that allows to transfer multiple data in a single transaction. To achieve high performance, the protocol defines independent transaction channels for address and data in write and read modes. Read and write transactions have their own address channel. The appropriate address channel carries all of the required address and control information for a transaction. The data channel carries both the read/write data and the read response information.
The overview of the architecture of the custom peripheral is shown in Figure 9.14. The peripheral has the slave and master interfaces devoted to the communication with the Microblaze and MCB respectively. The architecture is close to that one developed in System Generator environment. It reuses most of the module developed in the previous section, adapted to this design.
The peripheral receives as input, through the slave interface, the base address of the image, the base address of the output HOG array and the start signal. The base address of the two arrays are assigned during the execution of the C code by the Microblaze. It coordinates the initialization of the variables, sets-up the peripheral and starts it. After starting the peripheral, the C-code

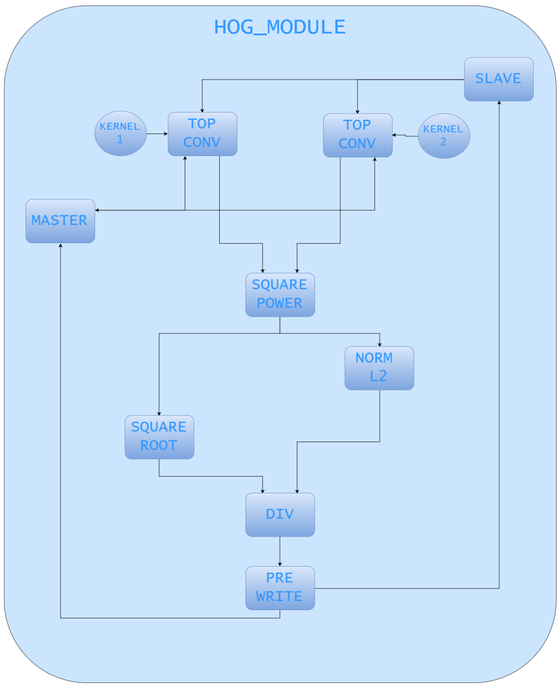


FIGURE 9.14. Flow of the GBCSS implementation.

pulls the end-signal that the peripheral sends as soon as his task is completed.

The first module in the flow is the *Top Conv*. The Figure 9.15 shows its internal architecture and FSM. The Module is devoted to request the data from the Memory and computes the convolution. The data is stored in Linear Buffers of 3 FIFOs to keep consistency among rows of the image and to prevent data loss. Each row is made by 288 values of 32 bits each. For this reasons, FIFOs of 512 length are used. To receive the data, the Burst read transfer is exploited. Every time the FIFOs are empty and the last row is not reached, the module sends a read request to the master interface. The master, as soon receive the start signal asserted, starts a read burst transfer of 512 values, the minimum number of values to read one row from the memory. Going deeply, the master actually requires 16 values of 32 bits at the time, updating the base address every time. To transfer 512 values, the master repeat this transfer 32 times. Because the DDRAM is an asynchronous resource, it requires some clock cycles before providing the data that fed the linear buffer. On the other hand, the Unit that performs the convolution is pipelined and, when started, requires new values at each clock cycle. To solve this time-related problem, 3 buffers are introduced to collect the data and when each of them contains one entire row, the convolution process can start. This solution is adopted to reduce the complexity of this module

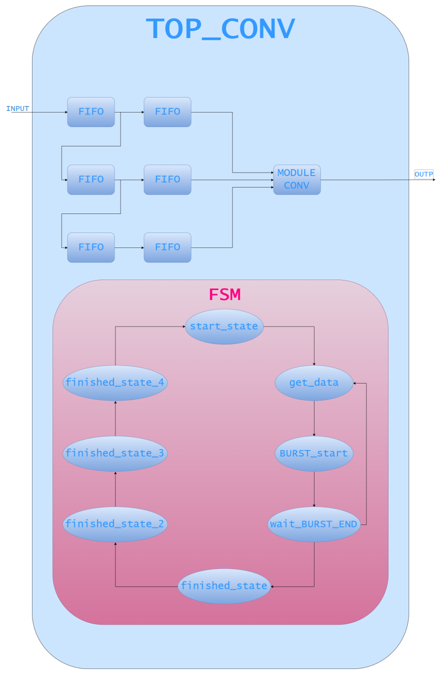


FIGURE 9.15. Top Conv Module.

because the idea was to build a flow that works, and then optimize it. The FSM is illustrated in the Figure 9.15. It handles the read requests through the three states *get_data*, *BURST_START* and *wait_BURST_end*. It repeats these steps until all the rows are loaded. Meanwhile, the control signals for the FIFOs and Convolution Unit are controlled. Finally, the *finally_stage* steps are used to conclude the last convolutions without requesting new data to the master interface. The Convolution Unit is the same one used in the System Generator design. The kernel parameters of the two *Top Conv* Modules are hard-encoded in the design. The output values, expressed in 32 bits, and relative ready signals are sent to the *SQUARE POWER* Module that implements the sum of the input squared. This is a combinational circuit that maintains the result in 32 bits. This result is sent to the *SQUARE ROOT* and the *NORM L2* modules. The former is shown in Figure 9.16. The main core of this module is the Square Root Unit, already explained in the previous section. It requires, for each input, 65 clock cycles to provide the result. It is fed by a Buffer that contains all input values. This solution is adopted because the module starts to compute the square root for each value in the buffer, when the norm value is computed. The norm L2 computation requires less time than computing the square root for each value. The result is in the 16.16 fixed point form.

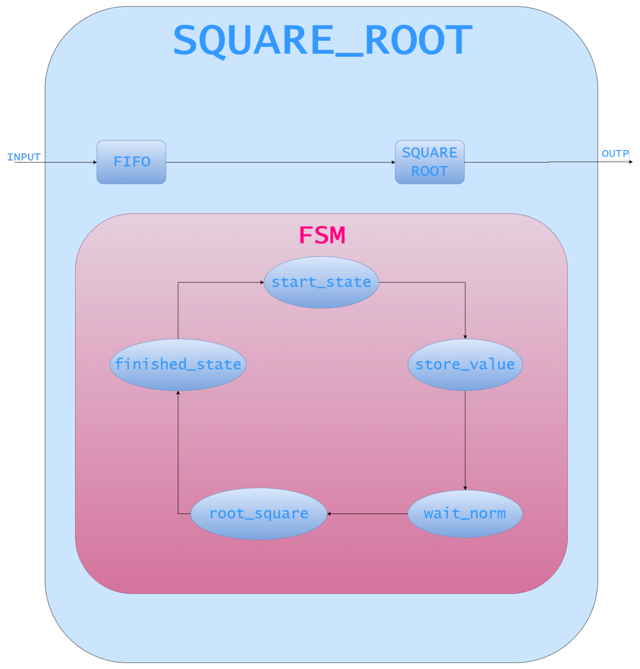


FIGURE 9.16. Square Root Module.

The next two modules are the same described to the previous section, so I will only cite them. The *NORM L2* Module sums all the results from the *SQUARE POWER* module and then computes the L2 norm. The result is in the 32.32 fixed point form. The *DIVISION* module computes the division between the *SQUARE ROOT* output and *NORM L2* output. The final result is expressed in 2.30 fixed point form.

The *PRE WRITE* Module is the last component devoted to store the result in the DDRAM. The Figure 9.17 shows its architecture. Every value of the final HOG is computed each 65 clock cycles. To store the data in the memory through a Burst transfer, the Master needs 512 values consecutively. The PRE WRITE module collects all the results in a buffer long 512 values and, as soon the buffer is full, the master can perform the BURST write transfer. The FSM handles the requests of write transfers though the states *get_values*, that wait the buffer full, and *wait_end_trans* that wait the master end the write transaction.

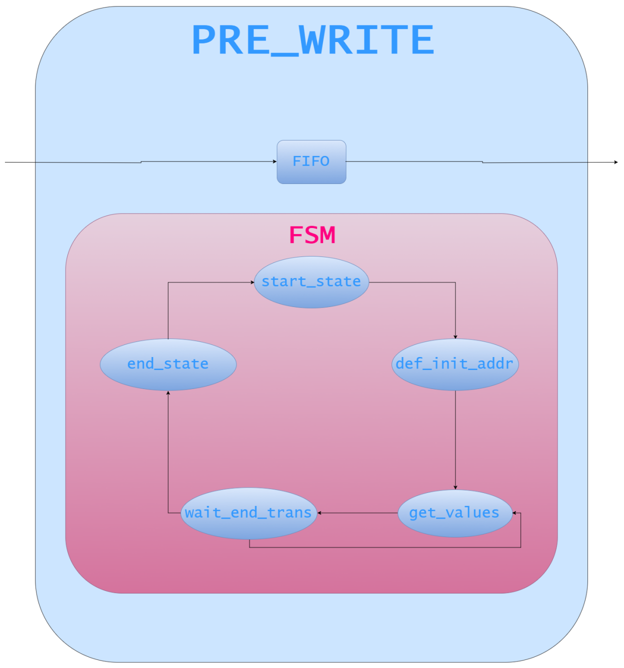


FIGURE 9.17. PRE WRITE Module.

9.4 Conclusion

The design was tested and produces the same HOG matrix of the original algorithm in MATLAB using same input image and input parameters. The next step will introduce two *HOG_MODULE*s fed with visual and IR images and compare the generated HOG matrices using a distance operator, that I couldn't developed due to time limit. The are some improvement that can be applied to the developed design to reduce the run-time:

- Improve memory utilization by storing 4 pixels values/word instead of 1 pixel value/word. This should improve the convolution module by 4X.
- Use two Memory Interfaces to access in parallel to the two banks of DDR4.

Moreover, there are some possible trade-off between precision and runtime that can be exploited:

- To obtain a HOG with the same size of the input images, it is required add a padding of 0 around the input image. Removing this step will reduce the run-time.
- Reducing the fixed point precision will increase the runtime. It will affect the norm l2 and square root modules.

Bibliography

[1] J. Daugman, *How iris recognition works*, 1, 14 (2014), pp. 1–10.

[2] S. Hashemi, T. Hokchhay, F. Buttafuoco, and S. Reda, *Approximate computing for biometric security systems: A case study on iris scanning*, (2017), pp. 1–5.

[3] C. B. Picardo and J. G. R. Delva, *Comprehensive comparison of gradient-based cross-spectral stereo matching generated disparity maps*, (2016), pp. 1–5.

[4] Xilinx, *Ultrascale fpga product tables and product selection guide*.

[5] ——, *Xilinx ds160 spartan6 family overview*.

[6] ——, *Xilinx ug526 sp605 hardware user guide*.

[7] ——, *Xilinx virtex ultrascale fpga vcu108 user guide*.