



UNIVERSITY OF  
LIVERPOOL

UNIVERSITY OF LIVERPOOL

# Real Time Image Processing on FPGAs

A THESIS SUBMITTED TO THE UNIVERSITY OF LIVERPOOL FOR THE DEGREE OF  
DOCTOR OF PHILOSOPHY IN THE FACULTY OF SCIENCE AND ENGINEERING

Department of Electrical Engineering and Electronics

Shaonan Zhang

January 2018

### **Abstract**

In recent years, due to improvements in semiconductor technology, FPGA devices and embedded systems have both been gaining popularity in numerous areas, from vehicle-mounted systems to the latest iPhones. Recently, as Intel (Altera) and Xilinx both released their new generations of ARM A9 processor integrated FPGAs, they have become very popular platforms which combine the hardware features of an FPGA and an embedded systems software's flexibility. This makes it suitable platforms to apply complex algorithms for real time processing of video images.

Feature tracking is a popular topic in image processing and usually includes one or more pre-processing methods such as corner detection, colour segmentation, etc. that could be undertaken on the FPGA with little latency. After the pre-processing, complex post-processing algorithms running on the ARM processors, that use the results from the pre-processing, can be implemented in the embedded systems.

The research described in this thesis investigated the use of low cost FPGASoC devices for real time image processing by developing a real-time image processing system with several methods for implementing the pre-processing algorithms within the FPGA. The thesis also provides the details of an embedded Linux based FPGASoC design and introduces the OpenCV library and demonstrates the use of OpenCV co-processing with the FPGA. The tested system used a low cost FPGASoC board, the DE1-SOC, which is manufactured by Terasic Inc. As a platform which contains a Cyclone V FPGA designed by Intel with a dual-core ARM A9 processor, the application developed is based on a customized OpenCV programme running on the ARM processors and

## **Abstract**

---

concurrently receives the pre-processing result processed by the FPGA. With the FPGA acceleration, the developed system outperforms a software-only system by reducing the total processing time by 48.2%, 49.5% and 56.1% at resolutions of 640x480, 800x600 and 1024x768 separately.

This reduction in processing time allows an improvement in the performance of systems using the results from the real-time image processing system.

### **Acknowledgements**

I would like to express my special thanks for my supervisor, Professor Jeremy S. Smith for his guidance and support for my research and my thesis writing. Meanwhile, I would like to thank my colleagues and their help. I also wish to thank my parents for their supports on my PhD study. Then I would like to thank Zhe Yang, who gave me a lot of help during the first two years of my research. Additionally, I would like to thank the members of staff of the Department of Electrical Engineering and Electronics, especially Professor A. Marshall for use of the Departmental Facilities and Equipment. Finally, thanks to all my other friends for their encouragement and help.

## ***Glossary***

ALiS—Alternate Lighting of Surfaces

ALU—Arithmetic Logic Unit

ASIC—Application Specific Integrated Circuit

CLB—Configurable Logic Block

CPU—Central Processing Unit

CRT—The Cathode Ray Tube

DSP—Digital Signal Processing

EDS—Embedded Development Suite

FPC—Flexible PIC Concentrators

FPGA—Field-Programmable Gate Array

FPGASoC—Field-Programmable Gate Array System on a Chip

GPU—Graphic Processing Unit

HDL—Hardware Description Language

HDMI—High-Definition Multimedia Interface

HLS – High Level Synthesis

HPS—Hardware Processing System

LCD—Liquid-crystal display

IP—Intellectual Property

## **Glossary**

---

OpenCV—Open Source Computer Vision Library

OpenCL— Open Computing Language

PAL—Phase Alternation Line

PIC—Physical Interface Cards

PC—Personal Computer

RTL—Register Transfer Logic

SD—Secure Digital

SOPC—System On a Programmable-Chip

USB—Universal Serial Bus

UVC—USB Video device Class

U-Boot—the Universal Boot Loader

VIP—Video Image Processing

# Contents

ABSTRACT .....	I
ACKNOWLEDGEMENTS .....	III
GLOSSARY .....	IV
CONTENTS.....	VI
TABLE OF FIGURES.....	X
TABLE OF TABLES .....	XIII
CHAPTER 1 INTRODUCTION .....	I
1.1  MOTIVATION.....	I
1.2  EXISTING TECHNOLOGIES AND SYSTEM PERFORMANCE .....	3
1.2.1  CPU & GPU .....	3
1.2.2  FPGA vs CPU vs GPU .....	4
1.2.3  SOPC.....	8
1.2.4  EMBEDDED SYSTEMS & FPGASoC.....	10
1.3  THESIS CONTRIBUTION .....	11
1.4  THESIS OUTLINE .....	13
CHAPTER 2 BACKGROUND .....	15
2.1  INTRODUCTION.....	15
2.2  MACHINE VISION & IMAGE PROCESSING.....	15
2.3  FIELD PROGRAMMABLE GATE ARRAY (FPGA) .....	17
2.3.1  ON CHIP PROCESSORS .....	20
2.3.2  ON CHIP BUSES .....	21
2.4  FPGA PROGRAMMING.....	24
2.4.1  HDL .....	25
2.4.2  HIGH-LEVEL SYNTHESIS.....	29
2.4.3  OPENCL .....	30
2.5  VIDEO IMAGE FORMAT .....	31
2.5.1  RGB .....	31

## Contents

---

2.5.2 GRAY SCALE.....	32
2.5.3 YUV(YCrCb).....	33
2.5.4 CHROMA SUBSAMPLING .....	34
2.5.5 INTERLACED VIDEO.....	34
2.6 SUMMARY.....	35
CHAPTER 3 DESIGN TOOLS AND HARDWARE ENVIRONMENT .....	37
3.1 INTRODUCTION.....	37
3.2 DESIGN TOOLS .....	37
3.2.1 QUARTUS II .....	37
3.2.2 INTEL PLATFORM DESIGNER.....	38
3.2.3 MODELSIM .....	47
3.2.5 DSP BUILDER.....	48
3.2.4 INTEL SoC EMBEDDED DESIGN SUITE (EDS) .....	49
3.2.5 TIMING ANALYZER .....	50
3.3 HARDWARE ENVIRONMENT.....	51
3.3.1 DEVELOPMENT BOARD.....	51
3.3.2 VIDEO INPUT DEVICES .....	54
3.3.3 VIDEO DISPLAY DEVICE.....	57
3.4 SUMMARY.....	57
CHAPTER.4 REAL TIME IMAGE PROCESSING ON FPGAs WITH THE HDL APPROACH .....	59
4.1 INTRODUCTION.....	59
4.2 THE CANNY EDGE DETECTION ALGORITHM.....	60
4.2.1 GAUSSIAN FILTERING .....	60
4.2.2 SOBEL EDGE DETECTOR .....	62
4.3 ALGORITHMS IMPLEMENTATION.....	67
4.3.1 GRAYSCALE TRANSFORMATION.....	67
4.3.2 GAUSSIAN FILTERING .....	69
4.3.3 SOBEL EDGE DETECTOR & NON-MAXIMUM SUPPRESSION.....	71
4.3.4 THRESHOLDING WITH HYSTERESIS.....	74

## Contents

---

4.4	SYSTEM ARCHITECTURE .....	76
4.5	RESULTS & DISCUSSION .....	79
4.6	SUMMARY .....	84
	CHAPTER.5 REAL TIME IMAGE PROCESSING ON FPGAS WITH THE HLS APPROACH .....	85
5.1	INTRODUCTION.....	85
5.2	HLS COMPLIER.....	85
5.3	HARRIS CORNER DETECTION.....	86
5.4	ALGORITHM IMPLEMENTATION.....	92
5.5	RESULTS AND DISCUSSION.....	94
5.5.1	CANNY EDGE DETECTOR .....	94
5.5.2	HARRIS CORNER DETECTOR .....	98
5.6	SUMMARY.....	100
	CHAPTER.6 A CO-PROCESSING FPGASoC SYSTEM .....	101
6.1	INTRODUCTION.....	101
6.2	THE FPGASoC SYSTEM DESIGN.....	101
6.3	COMPILE THE LINUX KERNEL .....	106
6.4	OPENCV .....	108
6.5	SYSTEM IMPLEMENTATION .....	110
6.6	RESULTS AND DISCUSSION.....	111
6.7	SUMMARY.....	115
	CHAPTER.7 CONCLUSIONS AND FUTURE WORK.....	116
7.1	CONCLUSIONS.....	116
7.2	FUTURE WORK.....	119
7.2.1	ALGORITHMS & SYSTEM IMPLEMENTATION .....	119
7.2.2	DEVICES & OTHER APPROACH.....	119
	REFERENCES .....	<b>ERROR! BOOKMARK NOT DEFINED.</b>
	APPENDIX A CODES OF CANNY EDGE DETECTION WITH HDL APPROACH .....	128
	APPENDIX B CODES OF CANNY EDGE DETECTION WITH HLS APPROACH .....	145
	APPENDIX C CODES OF HARRIS CORNER DETECTION WITH HDL APPROACH .....	153
	Shaonan Zhang	viii

## Contents

---

APPENDIX D CODES OF HARRIS CORNER DETECTION WITH HLS APPROACH .....	169
APPENDIX E CUSTOMIZED “GOOD FEATURE TO TRACK” OPENCV CODE.....	178

## ***Table of Figures***

FIGURE 1-1 A COMPARISON OF THE STRUCTURES OF CPU AND GPU .....	5
FIGURE 2-1 A REAL TIME IMAGE PROCESSING SYSTEM.....	17
FIGURE 2-2 AN ADAPTIVE LOGIC MODULE (ALM) BLOCK DIAGRAM .....	18
FIGURE 2-3 BASIC STRUCTURE OF FPGA .....	19
FIGURE 2-4 A TYPICAL AVALON-ST SIGNALS BETWEEN SOURCE AND SINK .....	22
FIGURE 2-5 ALTERA/INTEL SOC FPGA DEVICE BLOCK DIAGRAM .....	24
FIGURE 2-6 DESIGN FLOW FOR FPGA DESIGNS .....	26
FIGURE 2-7 A 4X4 BAYER SUBSET .....	32
FIGURE 3-1 QUARTUS II GUI (VER 17.0).....	38
FIGURE 3-2 QSYS GUI (VER 17.0) .....	39
FIGURE 3-3 EXAMPLE OF A 24-BIT VIDEO STREAM WITH AVALON-ST FORMAT SIGNAL TIMING .....	41
FIGURE 3-4 SYMBOL TRANSMISSION ORDER .....	43
FIGURE 3-5 HORIZONTALLY SUBSAMPLED Y'CbCr .....	43
FIGURE 3-6 FRAME BUFFER BLOCK DIAGRAM .....	47
FIGURE 3-7 ALTERA EDS COMMAND SHELL.....	49
FIGURE 3-8 TIMING REPORT OF TIMING ANALYZER .....	50
FIGURE 3-9 ARCHITECTURE OF DE1-SOC DEVICE .....	52
FIGURE 3-10 DE1-SOC FPGA BOARD TOP VIEW .....	54
FIGURE 3-11 AN NTSC CAMERA.....	55
FIGURE 3-12 TRDB-D5M CAMERA DAUGHTER CARD .....	56
FIGURE 3-13 LOGITECH C270 WEBCAM .....	57
FIGURE 4-1 RESULT ON DIFFERENT $\sigma$ VALUES AFTER GAUSSIAN FILTERING.....	61
FIGURE 4-2 5X5 GAUSSIAN FILTER KERNEL WITH $\sigma = 1.4$ .....	62
FIGURE 4-3 EXAMPLE OF SOBEL RESULT .....	63

## Table of Figures

---

FIGURE 4-4 FOUR POSSIBLE DIRECTIONS OF THE EDGES .....	64
FIGURE 4-5 PIXELS NEED TO BE COMPARED WITH.....	65
FIGURE 4-6 STAGES OF A CANNY EDGE DETECTION PROCESS.....	66
FIGURE 4-7 EXAMPLE OF CANNY RESULT.....	66
FIGURE 4-8 DATA FLOWCHART WITH 3 LINE-BUFFERS.....	69
FIGURE 4-9 THE KERNEL OF THE ORIGINAL IMAGE.....	70
FIGURE 4-10 RAPID METHOD WITH CANNY EDGE DETECTION.....	75
FIGURE 4-11 ACCURATE METHOD WITH CANNY EDGE DETECTION .....	75
FIGURE 4-12 ARCHITECTURE OF THE SYSTEM WITH TRDB-D5M.....	77
FIGURE 4-13 THE VIDEO FORMAT CONVERSION IN THE SYSTEM.....	77
FIGURE 4-14 THE QSYS (PLATFORM DESIGNER) DESIGN OF THE SYSTEM .....	78
FIGURE 4-15 INTERCONNECTION OF THE ACCURATE METHOD OF CANNY EDGE DETECTION IP.....	78
FIGURE 4-16 ARCHITECTURE OF THE SYSTEM WITH AN NTSC CAMERA.....	79
FIGURE 4-17 AN ORIGINAL RGB FRAME OF THE VIDEO STREAM.....	80
FIGURE 4-18 DESIRED CED RESULT ACHIEVED USING MATLAB.....	81
FIGURE 4-19 RESULT OF THE ACCURATE METHOD OF CED.....	81
FIGURE 4-20 RESULT OF RAPID METHOD OF CED .....	82
FIGURE 5-1 HLS DESIGN PROCESS .....	86
FIGURE 5-2 THE BASIC PRINCIPLE OF THE HARRIS CORNER DETECTION .....	87
FIGURE 5-3 BLACK AND WHITE CHESSBOARD IMAGE.....	90
FIGURE 5-4 CHESSBOARD IMAGE FILTERED WITH THE HARRIS CORNER DETECTOR.....	91
FIGURE 5-5 STAGES OF A HARRIS CORNER DETECTION PROCESS.....	91
FIGURE 5-6 EXAMPLE CODE OF A SHIFT REGISTER WITH HLS COMPILER.....	92
FIGURE 5-7 EXAMPLE CODE OF APPLICATION DEFINITION WITH THE AVALON-ST INTERFACE.....	93
FIGURE 5-8 EXAMPLE CODE OF HANDLING PACKETS SIGNALS WITH HSL COMPILER.....	93
FIGURE 5-9 CANNY EDGE DETECTION ALGORITHM IMPLEMENTATION WITH HLS APPROACH.....	93

## Table of Figures

---

FIGURE 5-10 HARRIS CORNER DETECTION ALGORITHM IMPLEMENTATION USING THE HLS APPROACH .....	94
FIGURE 5-11 ORIGINAL RGB DATA .....	96
FIGURE 5-12 CANNY EDGE DETECTOR WITH HLS APPROACH.....	97
FIGURE 5-13 CANNY EDGE DETECTOR WITH HDL APPROACH .....	97
FIGURE 6-1 A HIGH-LEVEL VIEW OF THE DEVELOPMENT FLOW .....	102
FIGURE 6-2 THE SYSTEM BOOT FLOW.....	103
FIGURE 6-3 GENERATION OF THE PRELOADER .....	103
FIGURE 6-4 THE DEVICE TREE GENERATION FLOW .....	104
FIGURE 6-5 SD CARD LAYOUT .....	105
FIGURE 6-6 SOFTWARE SCREENSHOT OF EMBEDDED LINUX COMMAND LINE.....	106
FIGURE 6-7 STEPS FOR BUILDING THE LINUX KERNEL .....	107
FIGURE 6-8 CONFIGURATION MENU OF LINUX KERNEL.....	108
FIGURE 6-9 64-BIT WORD STRUCTURE OF THE SYSTEM .....	110
FIGURE 6-10 ARCHITECTURE OF THE SYSTEM.....	111
FIGURE 6-11 CANNY EDGE DETECTION RESULT OF OPENCV.....	112
FIGURE 6-12 CANNY EDGE DETECTION RESULT OF HDL METHOD.....	112
FIGURE 6-13 A FRAME RESULT OF FEATURE TRACKING APPLICATION USING OPENCV .....	113
FIGURE 6-14 A FRAME RESULT OF FEATURE TRACKING APPLICATION USING OPENCV ACCELERATED BY FPGA .....	114

## ***Table of Tables***

TABLE 1-1 COMPARISON OF POWER CONSUMPTION FOR CANNY EDGE DETECTOR IMPLEMENTATIONS .....	8
TABLE 3-1 PACKET TYPES OF VIP PACKET .....	41
TABLE 3-2 HARDWARE OF BOTH FPGA AND HPS SYSTEM .....	53
TABLE 3-3 SPECIFICATION OF LOGITECH C270 WEBCAM .....	56
TABLE 4-1 THE APPROXIMATE CONDITIONS OF THE FOUR DIRECTIONS .....	73
TABLE 4-2 ANGLE ERROR WITH APPROXIMATE VALUE FOR THE RAPID METHOD .....	73
TABLE 4-3 THE APPROXIMATE CONDITIONS FOR THE 4 DIRECTIONS .....	74
TABLE 4-4 ANGLE ERROR WITH THE APPROXIMATE VALUE USING THE ACCURATE METHOD.....	74
TABLE 4-5 UTILIZATION SUMMARY WITH EACH METHOD IN DIFFERENT RESOLUTION.....	83
TABLE 5-1 COMPARISON BETWEEN THE HLS AND THE HDL APPROACH WITH CANNY EDGE DETECTOR.....	96
TABLE 5-2 COMPARISON BETWEEN THE HLS AND THE HDL APPROACH WITH HARRIS CORNER DETECTOR	99
TABLE 6-1 DESCRIPTIONS OF THE DIFFERENT BOOT STAGES .....	103
TABLE 6-2 INFORMATION STORED ON THE SD CARD .....	105
TABLE 6-3 FEATURE TRACKING ALGORITHM COMPARISON.....	115

## ***Chapter 1 Introduction***

### ***1.1 Motivation***

Image processing and computer vision has become a popular research area in both electrical engineering and computer science in recent decades. Although it has been widely applied in many fields, industry has shown more interest in real time image processing in recent years. Using classical serial computer architectures, it often takes significant time to process the data. To achieve real-time performance with low latency, distributed systems such as multi-core CPUs, GPUs, FPGAs or ASICs are used to improve the performance.

FPGAs can achieve real-time performance in many applications that general serial processors cannot without sacrificing resolution [1]. FPGAs usually have lower costs of system fabrication and maintenance compared with CPUs and GPUs. An FPGA's reconfiguration characteristics provides more flexibility than ASICs which cannot be reprogrammed after they have been fabricated.

In 2010, Altera, now owned by Intel, introduced their 28nm technology FPGASoC. In this generation, Altera integrated a Hardware Processor Systems (HPS) which includes a dual core ARM Coretex-A9 processor, into their low-cost product range the Cyclone V FPGA [2]. This 925 MHz ARM hard processor core is a good replacement for the synthesised soft NIOS II processor core on the Cyclone IV with a maximum frequency of 190MHz [2]. The FPGASoC combines the benefits of FPGA hardware for fast data processing and relative high-performance embedded software flexibility. By integrating both systems into a

single component, it reduces the hardware complexity, physical size, power consumption, and cost of the final system.

Many complex image processing applications, for instance object tracking and face detection, are not only based on time consuming convolution algorithms like edge detection and corner detection, but also require the original raw data to make high level decisions. Thus, the FPGA provide a good solution for parallel processing which can keep the original image data whilst undertaking low level convolution algorithms, for example edge detection, with low latency. Therefore, with the combination of an FPGA and an ARM processor, Intel's FPGASoC, has the potential to implement complex image processing applications, in real-time, on a low-cost board making it suitable for embedded image processing tasks.

This research investigated the use of low cost FPGASoC devices for real time image processing by developing a real-time image processing system with several methods for implementing pre-processing algorithms within the FPGA. The thesis also provides the details of an embedded Linux based FPGASoC design and introduces the OpenCV library and demos the use of OpenCV co-processing with the FPGA. The results demonstrate the improved performance obtained by using the combined FPGA – ARM combination compared with just using the ARM processor.

## ***1.2 Existing Technologies and System Performance***

### ***1.2.1 CPU & GPU***

Various types of architecture are used for real-time image processing platforms. One of the popular architectures uses a PC as the host system. This approach has been a very popular architecture for real time image processing for several decades. Every functional hardware module can be implemented separately via an expansion interface, for example a graphic card or USB camera.

As this technology has been developed over decades, there are many applications which focus on the PC. For example, Kang and Doraiswami developed a system which used an USB interface board and a webcam, allowing the PC to capture video for endoscopic applications [3]. The system implemented several image processing modules including contrast enhancement, Canny Edge Detection and the Hough Transform on the PC using MATLAB for real time image processing.

Many computer-based vision systems now use the software approach, using a generic CPU and GPU to perform all the image processing tasks. With the rapid increase of the processing capabilities of GPUs, the software approach now offers greater processing capability to handle more complex image processing tasks in real-time.

There is another example that Balfour et al. presented a vision-based closed loop control system for welding applications where a PC associated with a video capture card and a graphic card was used to perform real-time video acquisition, image analysis, display and process control [4]. However, these are not ideal for compact embedded vision systems as it requires a host computer

and a graphic card and therefore has a high-power consumption associated with these systems.

### ***1.2.2 FPGA vs CPU vs GPU***

The advantages and disadvantages of FPGA based image processing systems against CPUs & GPUs is now discussed. FPGAs are usually considered as a low-level and parallel device which provides flexibility so that it can be programmed to implement any logical circuit. Thus, it is also sometimes used to prototype ASIC designs.

However, FPGA design is usually considered too difficult or niche to use when compared with GPUs and CPUs, even for the senior designers whose proficiency is in software programming languages, as FPGA systems need to be developed using hardware languages [5]. The most common hardware languages for FPGAs are Verilog [6] and VHDL [7], which are called hardware description languages (HDLs). The main difference between these languages and traditional software languages is that HDLs describe hardware i.e. registers and Boolean logic functions, whereas software languages such as C describes the sequential instructions without knowing about the precise hardware implementation details. As a result, researchers and application scientists tend to choose software design because of the relative ease of development and the sheer number of abstractions and classifications available which significantly increase productivity.

Due to their earlier development, both CPUs and GPUs shared a great number of established libraries which have plenty of resources for designers to productively implement various tasks. FPGAs only provide a limited number of "IP" libraries and component features for users' designs.

However, these issues are just the first question to consider when designers choose whether to use FPGAs, GPUs or CPUs to build a system. The following is a comparison of implementations between FPGAs, GPUs and CPUs.

### 1.2.2.1 Structure

A significant element which impacts the choice of FPGAs, CPUs and GPUs is the structure. A suitable architecture for the targeted application can significantly improve its performance and reduce the system cost. For image processing, which can be considered as 2D signal processing, a high memory bandwidth for the memory intensive processing operations is required [5]. Figure 1-1 shows the architecture of a CPU and a GPU. Both CPU and GPU have the control units, ALUs (Arithmetic Logic Unit), DRAM (Dynamic Random-Access Memory) and Cache. But the difference between CPU and GPU is a large proportion of GPU chip is the ALUs

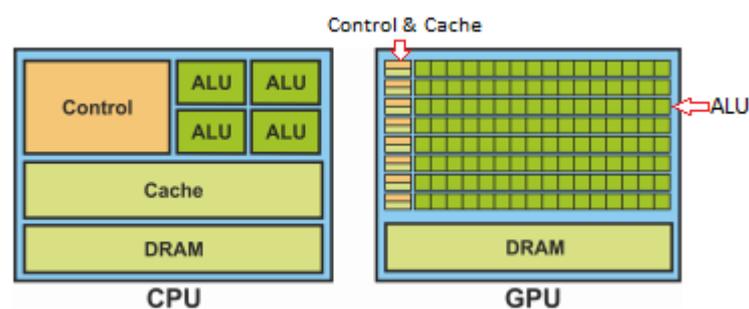


Figure 1-1 A Comparison of the Structures of CPU and GPU [8]

FPGAs show good performance on parallelism. Since it can implement a non-von-Neumann massive-parallel architecture [8], its calculated results can be fed directly to the next processing stage without temporary storage in the main memory. Hence, the requirement for memory bandwidth is much lower than

when implemented with a GPU or CPU. Therefore, it is thought to have better performance in image processing applications as its algorithms can exploit a great degree of parallelism [8].

The differences between CPUs and GPUs are caused by their different design goals as they are targeted at two different scenarios. CPUs need to be very versatile to deal with a variety of different data types. This makes the CPU's internal structure extremely complicated. Whilst what GPUs face is a large-scale data which is highly unified and mutually independent and normally operate in a pure computing environment with limited external interrupts. Comparing with the processing cores of both, the FPGAs programmable block is much simpler. Which hardware to choose depends on the complexity of the task required. For example, if a real time image processing system requires high resolution and complex calculations, GPU is more suitable for the work. But if the system only requires convolution based low level algorithms, FPGA become more suitable.

### **1.2.2.2 Performance**

There are various elements which should be considered when measuring how a system performs. As far as real-time image processing is concerned, factors like speed and energy efficiency are significant.

As mentioned by Asano, et al. [8], how fast a CPU, GPU or FPGA can achieve varies for each application. Generally speaking, FPGAs demonstrate extremely high parallelism at a lower clock rate; in comparison, the GPU is endowed with a high clock speed, also with relatively high parallelism, but is limited by poor memory management. CPUs, on the contrary, show relatively high clock speeds, facilitate memory management but their parallelism is limited. For an image processing system which contains numerous inherent parallelisms, the

advantage of a CPU is not obvious. This leaves a comparison between GPUs and FPGA. For example, GPUs have better implementation in normalized cross-correlation [9] and two-dimensional filters [8]; while FPGAs were identified as having outstanding performance in matched filter computations [10], K-Means clustering tasks and stereo vision [8].

To assess energy efficiency, Fowers et al. implemented a sliding window operation, sum of absolute differences and corrector and determined that, “the FPGA used orders of magnitude less energy than other devices in many situations” [11]. Table 1-1 shows a comparison of power consumption and energy usage of FPGA, CPU and GPU implementing a Canny Edge Detector at different resolutions. The researchers use a CPU Intel Core2 Duo E6600, 2.4 GHz; a GPU GeForce GTX 580, 1.54 GHz and a 28 nm Arria V 5AGXFB3 FPGA device for comparison [12]. The power consumption of FPGA remains almost the same at 1.5 Watts as the effect of the change of system is negligible. In the meantime, the power consumption of CPU and GPU stays around 150 Watts and 240 Watts. Thus, the energy consumptions of CPU are over one thousand times higher than the FPGA. With the advantages in architecture, the energy efficiency of the GPU is better than the CPU, but still it is incomparable with the FPGA. FPGA still leads with a factor of hundreds. However, with higher resolutions, the energy consumption of the FPGA is increasing fast which is 98.2 Millijoules at 3936x3936 which is 14.7 times higher than the performance at 1024x1024. But for the GPU, it increases by a factor of 2.5, the difference of power consumption between FPGA and GPU is reduced gradually.

Table 1-1 Comparison of power consumption for Canny Edge Detector Implementations

[12]

Resolution	CPU		GPU		FPGA	
	Power(W)	Energy(J)	Power(W)	Energy(J)	Power(W)	Energy(mJ)
512x512	141	2.8	231	0.5	1.5	1.7
1024x1024	147	8.8	244	6.08	1.5	6.7
3936x3936	153	213.1	251	15.0	1.5	98.2

### 1.2.3 SOPC

Several years ago, with the advances of semiconductor technology, it became possible to integrate the entire embedded/computer system including processors, memory and other system units into a single programmable chip - FPGA. This technology is called "System-on-a-Programmable-Chip" (SOPC) [13]. As a new approach, it provided another architecture capable of implementing a standalone embedded vision system.

Endowed with both a reconfigurable and compact nature, SOPCs provide high flexibility and performance with low risks; as its design can easily be adapted into various types of FPGAs and therefore present a wide range of performance. Concerns about problems like changing the architecture of the system are unnecessary as all the components of the SOPC could be separately implemented as individual non-device-specific soft IP cores which included the processors. Moreover, the utilization of soft processor core, for instance Nios II from Altera and Microblaze [14] from Xilinx, has accessed the system architecture's configurability to make a trade-off between performance and area via adjusting the architecture [15]. Furthermore, SOPCs have other

advantages like low cost and short development time compared with the ASIC approach [16]. For example, Hau et al. designed a rapid prototyping of an automated video surveillance system [17]. The system was designed with the Altera video processing and embedded design framework (VIP, UIP) and implemented on an Altera DE2-70 board which contained an Altera Cyclone IV FPGA. By using high performance and parallel hardware accelerators, the system demonstrated a real-time object detection algorithm. In addition, by the implementation of a software processor (NIOS II), user controls and application flexibility were achieved by applying high level programming language (C++). Wu et al. designed a real-time image processing system [18]. The system was integrated into a low-cost programmable chip and the system performance was maximized by using the cache and streaming transfer within the system. The effective bus-mastering scheme was also demonstrated. The system was implemented with an Altera Apex 20K programmable board, a SDRAM, a CameraLink CMOS camera with a custom designed camera interface card for video capture and a VGA monitor for video display. Both the systems described above are designed with hardware/software co-design method and implemented with a NIOS II CPU. Both systems were implemented with a SOPC approach and used the Avalon bus interface [19] to reduce the complexity of development and enhance the performance of the systems.

One of the key challenges that both systems had to overcome was the bandwidth of the external memory that stored the image. This memory, acting as a frame buffer would have to allow the image capture unit to write the image to the memory, the display unit to read data from the memory and the CPU to read and write data from the frame buffer. Also, if this memory is also used by the CPU for program and variable storage these accesses would also be sharing the available bandwidth.

### ***1.2.4 Embedded Systems & FPGASoC***

Another type of architecture is embedded systems which consist of one or more microprocessors/microcontrollers to control the whole system and perform the image acquisition and processing tasks. These microprocessors could be general purpose microprocessors or DSP microprocessors [20]. In recent years, the ARM processor has become the most popular microprocessor in embedded systems. With the availability of the ARM processors for System on a Chip (SOC) applications, it can build the whole system in a single device. For software development, it can use either Linux or Android as the Operating System platform where several open source libraries such as OpenCV and Python libraries can be installed to simplify the application development. However, as the more significant complexity of hardware architecture of microprocessors, it is result in a relatively less significant in speed [21].

In 2014, Guennouni et al have developed an application for multiple objects detection based on OpenCV libraries [22]. The system is based on cascade object detection algorithm for multiple object detection. Then Varfolomieiev et al have developed an improved algorithm of median flow for visual object tracking and its implementation on an ARM platform [23]. The algorithm has been implemented using the OpenCV library and tested on BeagleBoard-xM based on ARM processor. The algorithm uses the "good feature to track" algorithm and an edge detection algorithm. However, those pre-processing algorithms involved in these designs are more suitable for FPGA implementation instead of ARM processor.

Recently, as mentioned, a newer generation of FPGASoC has been introduced. These provide an upgraded version of the SOPC. They inherit the benefits of

SOPC and provide higher CPU performance because the CPU is hardwired rather than implemented using the FPGA logic blocks. Russell and Fishchaber designed a sign recognition system based on OpenCV using a Zynq SOC board [24]. The system uses the Xilinx Zynq-7020 chip to acquire 1920x1080 video with the VITA-2000 sensor attached to Flexible Physical Interface Cards (PIC) Concentrators (FPC) slot. The system was designed within six weeks and could process a frame in 5 seconds.

Typically, these image processing applications apply some low-level image processing tasks to pre-process the image. For example, the image may be Gaussian filtered to reduce noise in the image and then edge detection algorithms applied to highlight the edges of features in the image. Then higher-level algorithms use the information from the low-level information to make decisions about the object contents. For example, in vehicle traffic sign analysis, the edges will be analysed to determine the shapes of the signs, i.e. whether they are circular, square, rectangular etc. to aid in their classification.

### ***1.3 Thesis Contribution***

This research investigated the use of a low cost FPGASoC device for real time image processing by developing a real-time image processing system with several approaches for the pre-processing algorithms, using the FPGA, to reduce the processing time. Additionally, it synchronizes the original data in parallel with the pre-processed data in memory for further processing, i.e. the pre-processed image is stored as a 64-bit word with 8 bits each for the RGB values and 32-bit for the pre-processing results. Simultaneously, it provides the

infra-structure for implementing complex image processing applications on the integrated ARM system supported by the OpenCV library. The FPGA design was developed in Quartus II using the Video Image Processing (VIP) IP which provides several sub-systems such as frame buffer, clocked video in & out in Platform Designer (formally Qsys), which is Intel's (formally Altera) tool for developing SOPC systems. Therefore, the programmable hardware design needed to the algorithm development to be compatible with Intel's VIP based IP format so that it could be compatible with the Intel VIP subsystems.

This research applied two approaches, a Hardware Description Languages (HDL) approach and High-Level Synthesis (HLS) approach, to develop the algorithm IP for the FPGA. Firstly, the research shows the two methods to implement the Canny Edge Detection algorithm with the HDL approach. With both methods, it would show the correlations of resource usage and latency with accuracy and resolution.

Secondly, the HLS approach is researched by developing the Canny Edge Detector algorithm and Harris Corner Detection algorithm. With this approach, it would improve the productivity and reduce the difficulty of FPGA programming.

The next, the high-level processing details of an embedded Linux based FPGASoC design and the associated libraries are presented. Then an IP is developed using the HDL method which contains the original RGB data, Harris Corner Detection result, Canny Edge Detection result and Grayscale result all synchronized together, pixel by pixel, as a 64-bit word for each pixel. Concurrently, this design is based on the customized OpenCV application for post-processing implementations.

To summarise the novelty in this research, it is the development of an embedded FPGASoC image processing architecture where convolution-based image pre-processing takes place, in real-time, in the FPGA fabric allowing the ARM SoC processor to concentrate on the post processing algorithm thus reducing the time between the image is captured and the result presented. Such an approach will open up the market for low-cost real-time image processing applications as the system capital and running costs are significantly lower than using a PC based system. The comparison between the HDL and HLS approaches allows recommendation on which to select when developing an embedded image processing system.

## ***1.4 Thesis Outline***

This thesis is structured as follows.

Chapter 2 describes the background knowledge of the systems referred to in this thesis and the development environment.

Chapter 3 provides details of the design tools and hardware devices used in this research. It introduces the design tools and includes details of the Intel Video and Image Processing Suite (VIP).

Chapter 4 presents the FPGA image pre-processing design with the HDL approach and the HLS approach. It introduces the Canny Edge Detection (CED) algorithm and then describes the implementation of an accurate version and a simplified version of the CED algorithm using the hardware description language (HDL).

Chapter 5 presents the FPGA image pre-processing design with the HLS approach. It presents details of the implementation of the CED algorithm using the HLS approach. It also introduces the Harris Corner Detection (HCD) algorithm. Then it compares the results of both the CED and HCD implementation with the HDL approach and presents a discussion of the results. It also presents the system architecture in detail and discusses the results obtained.

Chapter 6 introduces an embedded Linux based FPGASoC design. It provides details of the FPGASoC design and introduces the OpenCV library. It also presents details of implementation of HCD and CED together in one IP. After that, it demonstrates the use of the FPGA to accelerate the OpenCV design in the FPGASoC design.

Finally, Chapter 7 presents the conclusions and potential for future research.

## ***Chapter 2 Background***

### ***2.1 Introduction***

This chapter presents the background knowledge of the systems referred to in this thesis and the development environment. The second section of this chapter briefly introduces the field of computer vision and image processing and describes the background to this technology. The third section presents the characteristics of FPGAs, including their history of development, internal structures, core technology and representative products. The next section focuses on the comparison between FPGAs and architectural approaches. The following section discusses the design flow for FPGA designs and provides information on the existing FPGA design methods. Finally, the video image format adopted is described.

### ***2.2 Machine Vision & Image Processing***

With the development of modern electronics, it is possible to make the computer 'see' things. This can be achieved by imitating the eye's function via image perception and content interpretation, this is called computer vision. The principle of computer vision is analysing images and then extracting relevant information from the scene [25], which could be a two-dimensional scene or a three-dimensional scene.

Nowadays, computer vision extends far beyond basic image processing. It also integrates communication and graphical techniques, processor and computer aided design as well as information handling and control. Using these sub-

systems help to build a computer vision application, for instance motion detection, image restoration, recognition and scene reconstruction etc. People usually set it goals of identifying various types of objects which could occurred in the scene.

Machine vision (MV), is frequently defined as the application of computer vision for manufacturing and industry [26]. For instance, MV shows good performance in automatic inspection and analysis tasks. It has become a significant technology in industry as it allows the replacement of humans in the visual inspection process with the associated cost reductions and increases in accuracy. Also, machine vision systems allow inspection tasks to be completed in hazardous work environments that are not suitable for manual work. Machine vision encompasses mechanical engineering, control, electric lighting, optical imaging, sensors, analogue and digital video technology, and industrial automation, etc. Besides, as a category of computer vision which mainly depends on machine-based image processing, it also has requirements of computer networks and digital input/output equipment when it controls other manufacturing devices.

With that in mind, image processing is the process of extracting information from the scene, converting an image signal into a digital signal and then electronically processing it. It is also considered as a collation of intensity data with its spatial arrangement [27]. It is usually a video frame or an image as the input for an image processing application. A real-time image processing system is required to implement specific image processing algorithms to verify the real-time performance before applying it into a computer vision system [18]. The specific task of a real-time system is processing the data required in the given interval, then analysing its performance to estimate the ability of it to process the data (image) in real time. A real-time system generally has three basic

functions, video acquisition, processing and display or control system as shown as Figure 2-1. At the first step, the image data is acquired from a digital/analogue camera or other video input devices. Then the processing system can be subdividing into a pre-processing system and a post-processing system. The pre-processing system includes the video format transformation which will be discussed in Section 2.5, pre-processing algorithms such as edge detection or image segmentation based on FPGA in this research with HDL or HLS programming method will be discussed in Section 2.3 and Section 2.4, respectively. For post-processing system, it will be discussed in details in Chapter 6.

Lastly there could be a video output which presents the processed results on a visual display or it could directly generate a control signal to other systems.

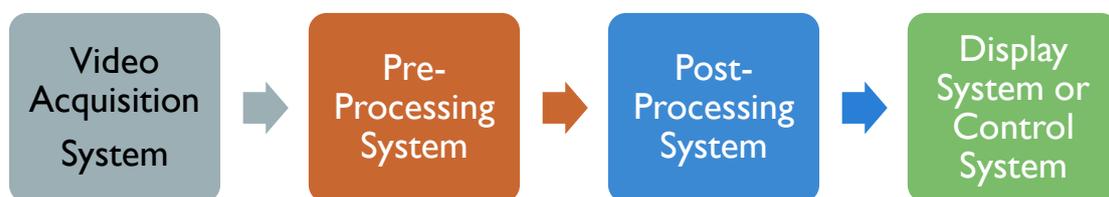


Figure 2-1 A real time image processing system

### ***2.3 Field Programmable Gate Array (FPGA)***

A formidable processing system is extremely important for building a satisfactory real-time image processing system. As mentioned previously, several architectures could be chosen as the processing hardware. The FPGA was firstly introduced in the 1980s by Xilinx and was named by Actel in 1988.

During more than 30 years of development, the FPGA has increased more than 10 000 times in logic capacity and 100 times in speed [28]. As its name suggests, FPGAs are designed as reconfigurable semiconductor devices. They are built with a matrix of logic blocks or adaptive logic modules (ALMs) of various types, such as multiplier blocks, memory and general logic; the ALMs are surrounded and connected by the programmable routing fabric, which is illustrated in Figure 2-3; then the array is surrounded by I/O blocks that interface the device to the outside world. Each ALM is made up of both registers and lookup tables (LUTs) [29] shown as Figure 2-2.

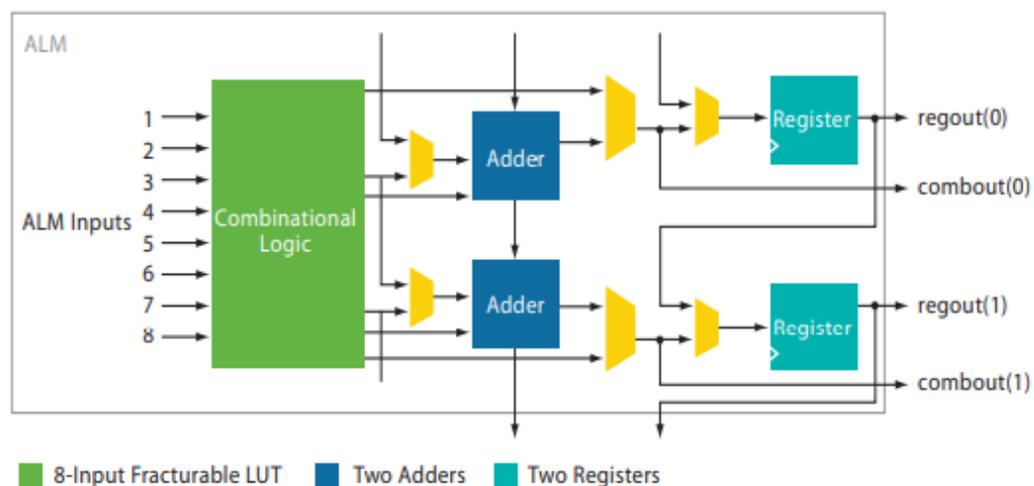


Figure 2-2 An Adaptive Logic Module (ALM) Block Diagram [29]

When the FPGA is configured for the specific digital circuits, each item of the CLBs would be assigned a simple independent logic function. The CLBs utilize the LUTs to implement the Boolean logic function and then it is connected by the routing fabric to make the structure of the whole digital circuit. The I/O blocks consequently link the logic matrix to the outside. With an increase of the order of magnitudes of logic functions, the FPGA device, in theory, is able to be programmed into any kind of logic circuits. However, it will never be as fast

as dedicated hardwired circuit, or as energy efficient, when using the same semi-conductor technology.

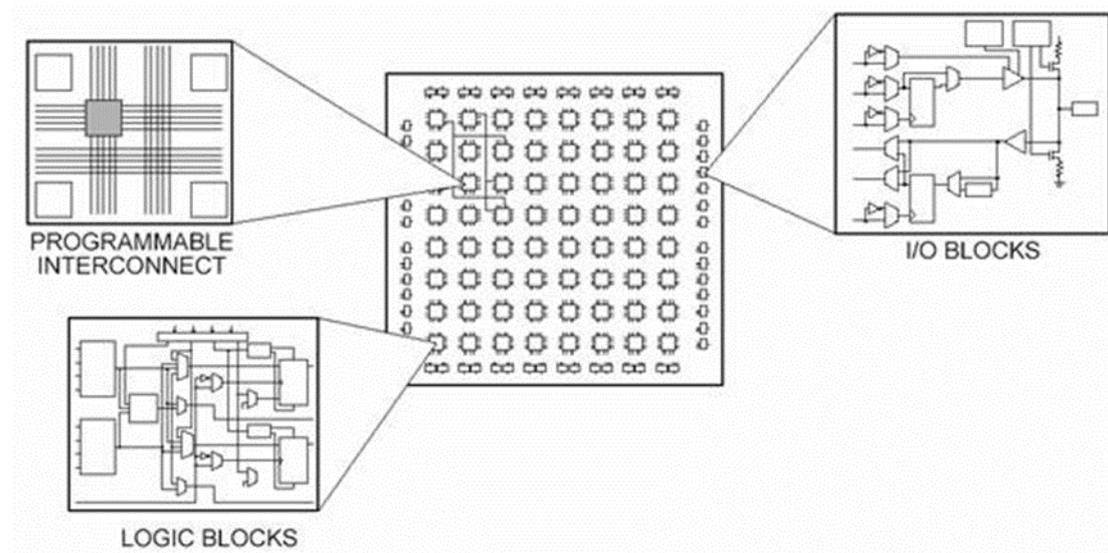


Figure 2-3 Basic Structure of FPGA [30]

Currently, there is a new trend in the FPGA field; that is to further develop the approach of coarse-grained architecture, synthesizing traditional FPGAs' interconnects and logic blocks together with embedded microprocessors and their related peripherals. This is how the System on a Programmable Chip (SOPC) [13] was developed. This kind of architecture utilizes the soft processor cores developed by the FPGA logic vendor, for instance the MicroBlaze (Xilinx) and Nios II (Intel/Altera).

The more recently introduced alternative approach based on hard processors could be found in the Xilinx Zynq-7000 All Programmable SoC [31] and Altera Cyclone V FPGA SOC [32] (shown in Figure 3-9), both integrate FPGA hardware with a dual-core ARM-A9 processor. Another example is the Microsemi SmartFusion which integrates the ARM Cortex-M3 hard processor core as well as analogue peripherals, such as multi-channel DACs and ADC with the flash-based FPGA fabric [33]. In the next section, these processors will be described.

### ***2.3.1 On chip processors***

As previously mentioned, there are two types of processors provided by FPGA manufacturers: hard processor and soft processor. A hard processor means an integrated processor like an ARM processor in hardware within the IC, whilst a soft processor is implemented within the FPGA fabric. Hard processors generally offer better performance (speed), lower power consumption and higher density. However, a soft processor system can have highly configurable features by using a specialized instruction set. Also, a programmable quantity of processors can be instantiated, as needed, with each tuned to the required area and performance specifications.

There are several processors commonly used in SOPC designs. Firstly, the soft processors, such as the Nios and Nios II developed by Intel/Altera, Microblaze developed by Xilinx and OpenRISC developed by OpenCores. Nios II [34] is a 32-bit soft-processor architecture designed specifically for the Intel/Altera family of FPGAs. Nios II incorporates many enhancements over the original Nios architecture, making it more suitable for a wide range of embedded computing applications, from DSP systems to control applications. Similar to the original Nios, Nios II has a RISC soft-core architecture which is implemented entirely in the programmable logic and memory blocks of the Intel/Altera FPGAs. The soft-core nature of the Nios II processor lets the system designer specify and generate a custom Nios II core, tailored for specific application requirements. System designers can extend the Nios II's basic functionality by adding a predefined memory management unit or defining custom instructions and custom peripherals [34].

As a competitor product, the MicroBlaze [35] is a soft microprocessor core designed by Xilinx for Xilinx FPGAs. It is implemented entirely in the general-purpose memory and logic fabric of Xilinx FPGAs. In terms of its instruction set

architecture, MicroBlaze is similar to the RISC-based DLX architecture which is described in the popular computer architecture book by Patterson and Hennessy [36]. With few exceptions, the MicroBlaze can issue a new instruction every cycle, maintaining single-cycle throughput under most circumstances [35].

On the other hand, hard processors are usually developed by specialized semiconductor companies. The PLD vendors purchases their IP licenses, fabricating and optimizing the processors into their specific FPGA family such as Altera's Cyclone V and Xilinx's Xilinx Zynq-7000 which include a 1.0 GHz and 800 MHz dual-core ARM Cortex-A9 respectively.

The Cortex-A9 [37] is a 32-bit hard processor with the ARMv7-A architecture which made it high-performance and low-power. Compared with soft-core processors, the hardware devices inside the chip which improve its performance includes, but is not limited to, an AMBA Level 2 Cache Controller, a global timer, the Floating-Point Unit and Direct Memory Access (DMA) [37]. When integrated inside an FPGA, the Cortex-A9 also provides the Advanced Extensible Interface (AXI) interconnects which directly connects to the FPGA allowing data exchange between the ARM processors and the FPGA logic. More details of on chip buses is presented in the next section.

### ***2.3.2 On chip buses***

Each of the processors use their own, different, on-chip bus. General speaking, a bus, in this context, is a public communication trunk that transfers information between the various functional components of a computer. The on-chip bus is the most common method to connect IP cores in an SoC, through which the

data communication could be transferred between the IP cores. Two standards of on-chip buses, Avalon Interface [19] and AMBA [38] are mentioned here. The former, the Avalon Interface bus, was launched by Altera to allow all peripherals to interface with its own IP and softcore processors. There are several types of Avalon Interfaces, for example the Avalon Streaming Interface (Avalon-ST) and the Avalon Memory Mapped Interface (Avalon-MM), the Avalon Conduit Interface and the Avalon Tri-State Conduit Interface (Avalon-TC) [19].

Avalon-ST interfaces support data paths requiring the following features: Low latency, high throughput point-to-point data transfer, Multiple channel support with flexible packet interleaving, error, and start and end of packet support for data bursts and automatic interface adaptation [19].

Figure 2-4 shows the signals that are typically included in an Avalon-ST interface. As this figure indicates, a typical Avalon-ST source interface drives the valid, data, error, and channel signals to the sink. The ready signal from the sink to source indicates when the sink is able to receive data.

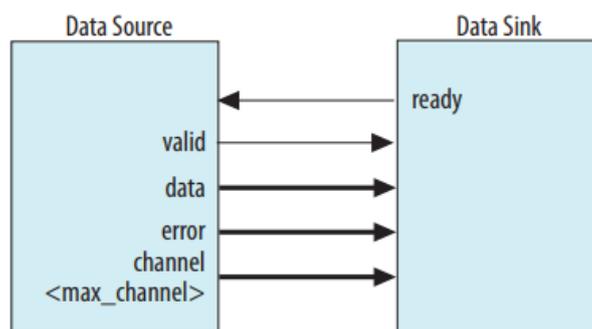


Figure 2-4 A typical Avalon-ST signals between Source and Sink [39]

Avalon Memory-Mapped (Avalon-MM) interfaces can be used to implement read and write interfaces for master and slave components. The components include

microprocessors, memories and DMAs which typically use memory-mapped interfaces to perform read or write operations [19]. Avalon-MM interfaces range from simple to complex. SRAM interfaces that have fixed-cycle read and write transfers have simple Avalon-MM interfaces, but pipelined interfaces capable of burst transfers are more complex [19]. Many components may have both Avalon-ST and Avalon-MM interfaces where the ST interfaces are used in a data-flow system to receive and transmit streaming data whilst the MM Interface is used to write configuration information to the IP Component and monitor its status.

The AMBA bus is another on-chip bus developed by ARM. It has four levels of hierarchy: Advanced high-performance bus (AHB), advanced system bus (ASB), advanced peripheral bus (APB) and most recently the advanced extensible interface (AXI). Generally, the high-performance system bus (AHB or ASB) is mainly used to meet high bandwidth requirements between high-speed devices such as the CPU, DMA and memory. While, most of the low-speed external devices of system are connected by the low bandwidth bus APB; then the system bus and the peripherals bus are connected by a bridge (AHB/ASB-APB-Bridge). The AXI3 or AXI 1.0 interface is used on ARM Cortex-A processors including the Cortex-A9 which also provide the interconnections between the FPGA and the HPS in the Cyclone V SOC [40] as shown in Figure 2-5.

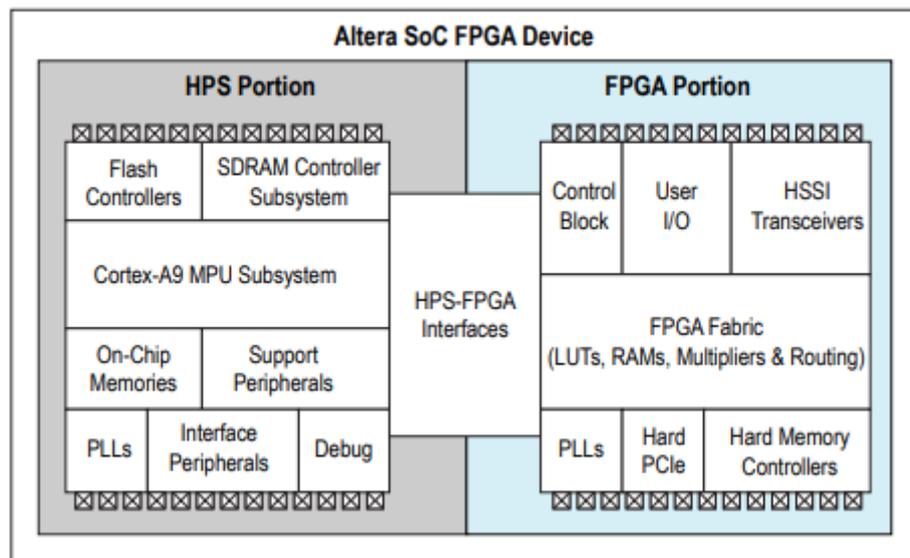


Figure 2-5 Altera/Intel SoC FPGA Device Block Diagram [40]

The design of the internal architecture only begins when the bus system is selected. Designers consequently work on the data and control paths, the hardware/software co-design, processor optimisation and both on-chip and off-chip components definition etc. these could be designed using different tools.

## 2.4 FPGA Programming

FPGA configuration/programming is also not like a general CPU program which is based on instructions that are decoded by hardware; it contains a complete configuration of the FPGA hardware. This limits the methods that can be used for FPGA programming. In this section, the languages used for FPGA programming and the design flow are discussed.

### **2.4.1 HDL**

Hardware description languages (HDL) are languages used to describe the electronic system hardware behaviour, structure and data flow. It is a method popularly applied in the design of ASICs and FPGAs. There has been over 40-years of history of evolution since HDLs were developed and they have been widely used in all of the design stages of electronic systems, such as simulation, verification, modelling and synthesis, etc. Hundreds of hardware description languages appeared by the 1980s, which played a significant role in promoting design automation. However, these languages generally varied from each other as they were targeted at a specific design area. As a result, the numerous languages made it difficult for the user to choose an appropriate system. Therefore, an urgent need emerged for a standard hardware description language to be generally accepted. To meet this requirement two languages VHDL and Verilog emerged. So far, they both are regarded as two of the most common languages for ASIC and FPGAs designers.

HDLs function in every stage of the design flow, ensuring the correct implementation of the system design. Figure 2-6 shows the design flow for FPGA designs. An abstract requirement would eventually be transformed into the specific configuration bitstream for the FPGA after certain programming procedures.

The design flow begins at the architecture design. Before that, some preparatory work should be carried out, such as the specification of the project, system design, FPGA chip selection and lists of IPs which could be used.

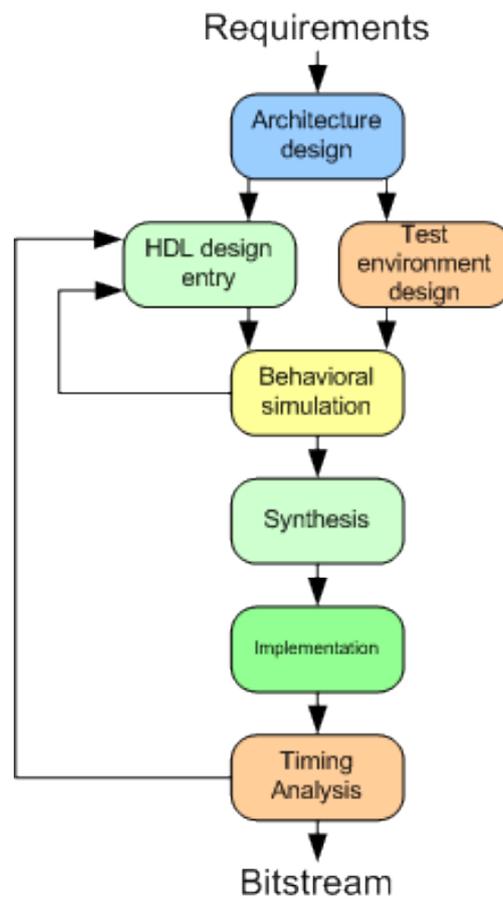


Figure 2-6 Design flow for FPGA designs [41]

Generally, the top-down design method is used to divide the system into several basic units. Then, each basic unit is divided into the lower-level basic units, this operation would go on until the Electronic Design Automation (EDA) element library can be used directly.

The procedure of the HDL design entry and the test environment design are carried out simultaneously. The former is the process of presenting the designed system or circuit in the form required by the HDL system and then inputting it to the EDA tool. The method most widely used, in practice, is the HDL language text input, which can be divided into ordinary HDL and behaviour HDL. Ordinary HDL (i.e. structural HDL and Register Transfer Logic (RTL) HDL) is mainly used in simple small designs. While in medium and large projects,

behavioural HDL is implemented using the mainstream languages of either Verilog HDL or VHDL or a combination of both.

The design will then undergo behavioural simulation. Its function is to verify the logic function of the designed system before synthesis for the target hardware. At this stage, there is no propagation delay information in the simulation just a logic functional simulation. Before the behavioural simulation, the waveform files and test vectors should be established via a Waveform Editor or HDL. The simulation results will generate the report file and then output the signal waveform, from which the changes of each node can be observed. If an error is found, the designer will go back to the last design step for modification. The common simulation tools are ModelSim from Model Tech, VCS from Synopsys and NC-Verilog as well as NC-VHDL from Cadence.

Synthesis is then undertaken. The objective of synthesis is to translate the description on a higher level of abstraction into a lower level description. It optimizes the logical connections generated by the description and timing requirements and flattens the hierarchical design for implementation in the FPGA.

Synthesis refers to compiling the design into a logical netlist which consist of the basic logic units such as AND gates, OR gates, NOT gates, D type flip-flops rather than a true gate level circuit. As the synthesis needs to make use of the FPGA manufacturer's layout and wiring function it is generated according to the standard gate structure net table which is generated after the synthesis. To be able to convert into a standard portable structure network table, the writing of the HDL description must conform to the style required by a specific package.

The stages of synthesis are as follows. It configures the integrated generated logical network table to a specific FPGA chip. During this procedure, the layout

and routing is the most important process as it configures the hardware primitives and the underlying units in the logical netlist to the inherent hardware structure of the chip. This operation often needs to make choices between the best performance (speed) and the best area usage.

At present, the structure of a FPGA is very complex, especially for timing constraints. The timing-driven engine needs to be used for layout and routing. After finishing the routing, the software tool will automatically generate a report to provide information about the use of each part of the design. Moreover, as only FPGA manufacturers truly understand the chip structure, the tools for layout and routing must be provided by the FPGA manufacturers.

The final process for the design is timing analysis, referring to the annotation of the delay information of the layout and routing to the design network table to detect any timing violations. Based on the result, the designer could identify the highest working frequency of the design in that chip, check whether the timing constraints are satisfied and analyse the clock quality. The system must be modified and returned to the HDL design entry stage if the constraints are not being satisfied.

In summary, as mentioned previously, HDL languages are not easy to learn. There is an obvious difference between them and other programming languages. It requires designers to learn and master this language. It also requires a large effort for the design process rather than just coding. Therefore, a suitable tool which could link the high-level programming language with HDL seems to be necessary.

### ***2.4.2 High-Level Synthesis***

The recent development of the High-Level Synthesis (HLS) allows an alternative method to implement the same design goals, without using HDL languages. HLS is a process that translates a high-level program, normally written in C/C++, at a behavioural or algorithmic level into a low-level automatic digital description, for instance, code in HDL. The popular HLS tools include Xilinx's Vivado HLS and Intel's HLS Compiler, both can automatically translate C/C++ into Verilog or VHDL. What's more, the HLS aims to allow the programmer to generate the design at a higher level which relieves them from the pressure of learning a new HDL. Therefore, it brings noticeable benefits when utilised, such as promoting the design speed and shortening the development time. It is also causing less descriptive mistakes which makes the programming and any modifications easier.

However, the compiler does not always make the best choice when it interprets the code. An incorrect and ineffective interpretation would consequently impact the performance of system. So certain compiler directives, such as buses input/output and loop pipelining /unrolling, needs to be added in the basic code to ensure proper implementation by the compiler.

While, effective programming requires far more than simply implementing the HLS. The programmer should have an in-depth understanding of an HDL to master the HLS. On the other hand, a HLS could be a great assistant if the HDL is well understood, even though people commonly thought the best way to program efficiently is to use the low-level language directly. Additionally, designers need to be aware that there is a noticeable distinction between simulation results and hardware results. The HLS tools don't guarantee the cycle accuracy. To deal with this, the programmer should have further HDL language knowledge; and the extra debugging skills.

### ***2.4.3 OpenCL***

As well as the choice between HDL and HLS when programming an FPGA, people could also implement their design using OpenCL. OpenCL is a framework of program development for parallel programming on heterogeneous platforms. This environment was originally developed for GPUs but now it has developed into a synthesize platform which includes GPUs, CPUs, FPGAs and other processors. It consists of a language (based on C99 and C++11) for writing kernels (functions running on OpenCL devices) and a set of application programming interfaces (APIs) to control and define the platform. OpenCL provides standard parallel computing based on task and data segmentation.

When implementing OpenCL on FPGAs, it uses the structure of FPGAs for a small application optimized processor core. This approach asks programmers to define the parallelism clearly and explicitly instead of the automatic parallelization in HLS. Consequently, programmers tend to reduce the massive parallelism on FPGA to the GPU's level; this operation eases the GPU programmer's access to FPGAs but causes a large performance cost. A comparison test has been undertaken between OpenCL and other three HLS languages includes Bluespec System Verilog, LegUp and Chisel about the performance of each architecture on FPGAs [40]. It showed that OpenCL and other advanced frameworks which applied a GPU-programmer-friendly architecture had a poor and unstable performance, while the approaches with low-level architecture were implemented quickly and efficiently on an FPGA.

## **2.5 Video image Format**

In image processing on FPGAs the real-time image data is collected from the scene by an input device and is then processed by the FPGA on the target board and finally output as a digital signal stream showing the results on a display device. During the entire period, the image format would experience several translations in order to meet the different format requirements in each step of the processing. Therefore, it is necessary to know how to make transformations between the two different image formats. This part introduces the most common formats used for image representation.

### **2.5.1 RGB**

The RGB colour model is an additive colour model in which red, green and blue light are added together in various ways to reproduce a broad array of colours. The name of the model comes from the initials of the three additive primary colours, red, green and blue.

Typical RGB input devices are colour TV and video cameras, image scanners, video games, and digital cameras. Typical RGB output devices are TV sets of various technologies (CRT, LCD, plasma, OLED, Quantum-Dots etc.), computer and mobile phone displays, video projectors, multicolour LED displays and large screens such as Jumbotron.

#### **2.5.1.1 Bayer RGB**

The Bayer filter is a colour format popularly applied in digital cameras, camcorders, and scanners for creating colour images. A three-chip colour digital

camera requires three monochrome sensors and associated R, G & B filters to obtain the colour image R, G, B component but the cost is high. Whilst a single CCD can obtain the colour image by covering a CCD surface with a red, green and blue mosaic filter and then implement the output signal through specific processing algorithms to synthesise an RGB value for each pixel. This design concept is called the Bayer pattern. A filter pattern is made of 50% green and red and blue account for 25% respectively.

As for sensor, the image structure of raw output is Bayer RGB. The process that transforms the Bayer data to RGB is called De-mosaicing. The process keeps the values of the red plane and blue plane and take the average of 2 green values from the 4x4 subset (Figure 2-7).

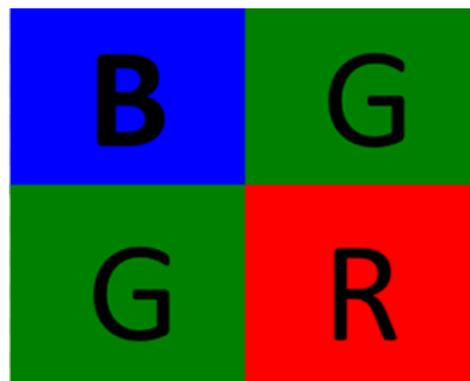


Figure 2-7 A 4x4 Bayer subset

### ***2.5.2 Gray Scale***

A grayscale image, which means the images are simply presented by different gray shades, is generally extracted and used because each pixel just indicates the amount of light received at that pixel, i.e. a Gray Scale image just carries intensity information.

The conversion between RGB and Gray Scale is given as equation (2.1):

$$Y' = 0.299R + 0.587G + 0.114B \quad (2.1)$$

### **2.5.3 YUV(YCrCb)**

YUV, is also known as Y'CrCb, Y Pb/Cb Pr/Cr, Y'CBCR or YCBCR. YUV is a popular colour coding method adopted by the European television systems. It can be categorized as YUV (PAL) and Y'CbCr (YUV compression and offset version). Generally, YUV (PAL) is used for colour TV sets while the Y'CbCr is widely used in computer systems, therefore the YUV discussed in this thesis refers to Y'CbCr.

In Y'CbCr the Y (Y') stands for the brightness (Luminance or Luma). The U, Y (or Cr and Cb) indicate the chroma components including the red-difference and blue-difference. There is a difference between Y and Y', as the former is luminance, which is the non-linear encoding of light based on the Gamma-corrected RGB primaries.

There is a mathematical coordinate conversion formula which associates the Y'CbCr colour spaces with that of RGB and vice versa.

The conversion formula between YUV and RGB is showed as equation (2.2) (2.3) follows (RGB values range from 0 ~ 255):

$$Y' = 0.299R + 0.587G + 0.114B$$

$$U = -0.147R - 0.289G + 0.436B \quad (2.2)$$

$$V = 0.615R - 0.515G - 0.100B$$

Thus,

$$R = Y + 1.14V$$

$$G = Y - 0.39U - 0.58V \quad (2.3)$$

$$B = Y + 2.03U$$

### ***2.5.4 Chroma Subsampling***

Digital signals are usually compressed to reduce file size and save the bandwidth of transmission. As the human visual system is much more sensitive to variations in brightness than colour [43], a video system can be optimized by devoting more bandwidth to the luma component (usually denoted  $Y'$ ), than to the colour difference components  $C_b$  and  $C_r$ .

In compressed images, for example, the 4:2:2  $Y'CbCr$  scheme requires 2/3rds the bandwidth of (4:4:4) RGB. This reduction results in almost no visual difference as perceived by the viewer.

### ***2.5.5 Interlaced video***

Interlaced video is a technique for doubling the perceived frame rate of a video display without consuming extra bandwidth. The interlaced signal contains two fields of a video frame captured sequentially. This enhances motion perception to the viewer and reduces flicker by taking advantage of the phi phenomenon.

The phi phenomenon is the optical illusion of perceiving a series of images, when viewed in rapid succession, as continuous motion.

This effectively doubles the time resolution (also called temporal resolution) as compared to non-interlaced footage (for frame rates equal to field rates). Interlaced signals require a display that is natively capable of showing the individual fields in a sequential order. CRT displays and Alternate lighting of surfaces (ALiS) plasma displays are made for displaying interlaced signals.

Interlaced scan refers to one of two common methods for "painting" a video image on an electronic display screen (the other being progressive scan) by scanning or displaying each line or row of pixels. This technique uses two fields to create a frame. One field contains all the odd-numbered lines in the image; the other contains all the even-numbered lines.

## **2.6 Summary**

This chapter gives an introduction to FPGAs and its related technology as well as devices. The image processing system is developed from the field of machine vision as well as image processing. Then the FPGA technology was introduced, and it was explained how they could be used to form SOPCs.

An FPGA is made of a matrix of CLBs with divergent functions and tasks. There are two types of processors for FPGAs, the hard processor and soft processor. Generally, each of the chip processors is supported by different formats of on-chip-buses provided by their companies. Based on their unique structures, FPGAs show distinct performance increases compared with GPUs and CPUs due to its high parallelism at low clock rates and good computing ability with high energy efficiency. A FPGA application is usually designed with the HDL, which differs from other programming languages such as C++. To ease the

programming, transformation frameworks like HLS and OpenCL are applied which can take high level code and produce HDL code. The image format is another issue should be considered during the data processing, for example how to make the conversation between the original RGB, YUV, Gray Scale and Interlaced video.

## ***Chapter 3 Design tools and Hardware Environment***

### ***3.1 Introduction***

This chapter presents the design tools and hardware that has been used in this work as the technical background was given in Chapter 2. Section 3.2 introduces the toolkit of Quartus II, whose different components have different contributions in building the system architecture, including both the program and the system design part. The system hardware environment is then introduced in details and the specific devices used in this research listed.

### ***3.2 Design Tools***

There are several common system-design service providers in the field. Quartus II developed by Intel was selected as it not only supports various FPGA/FPGASoC devices, but also provides a series of EDA tools which ensure good linkage between each part designed in this work. Perhaps the main reason for choosing Altera devices rather than Xilinx is that Altera FPGA devices have been used for teaching and research for many years at the University of Liverpool., Quartus II provides several toolkits and IPs which are targeted at Intel FPGAs.

#### ***3.2.1 Quartus II***

Quartus II is a development tool developed by Intel, formally Altera. It is used for analysis and synthesis of HDL designs, compiling the hardware design,

configuring the target device with the programmer and performing timing analysis. Quartus includes an implementation of VHDL and Verilog for hardware descriptions, visual editing of logic circuits, and vector waveform simulation [44]. All the processes shown in Figure 2-6 can be undertaken by Quartus II or with its associated tools. Figure 3-1 shows the GUI of the Quartus II software.

The Quartus II software contains several tools: Platform Designer (previously SOPC/Qsys Builder), SoC Embedded Design Suite (SoCEDs), DSP Builder etc.

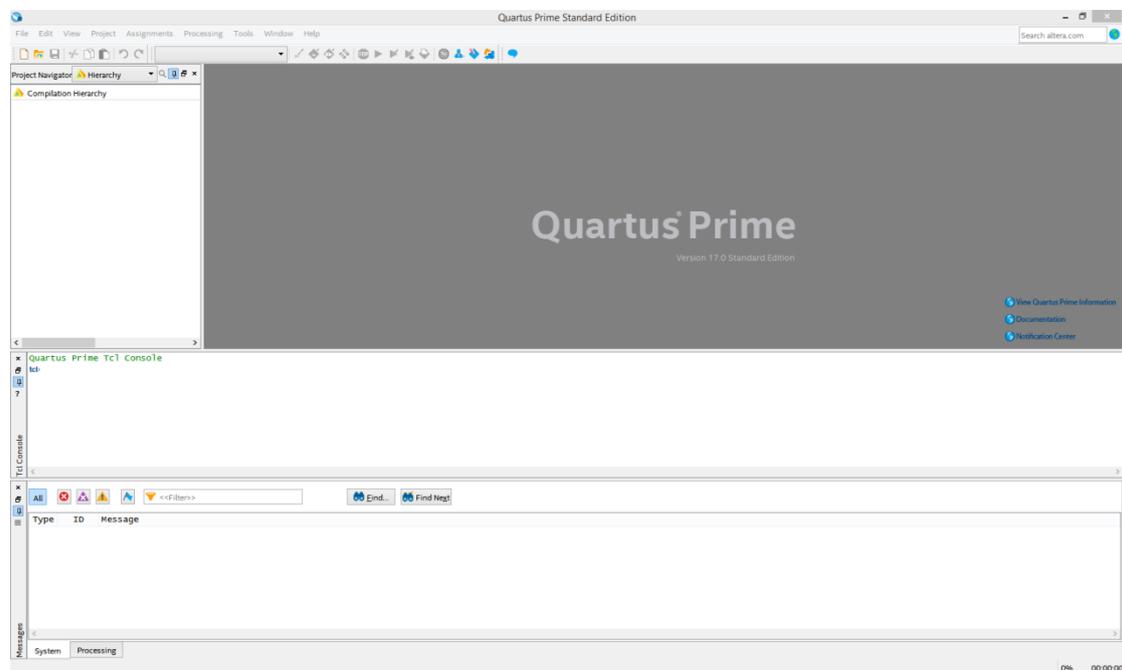


Figure 3-1 Quartus II GUI (ver 17.0)

### ***3.2.2 Intel Platform Designer***

Intel Platform Designer is a system integration tool which is a part of the Quartus II design software. It saves time in the FPGA design process by automatically generating interconnect logic to connect IP (intellectual property) functions and subsystems. It allows the various functions of the IP modules

available in the system library to be integrated into a system in much less time than being programmed directly by the designer [44], for example, components from the Video and Image Processing (VIP) Suite IP modules can be quickly instantiated and connected.

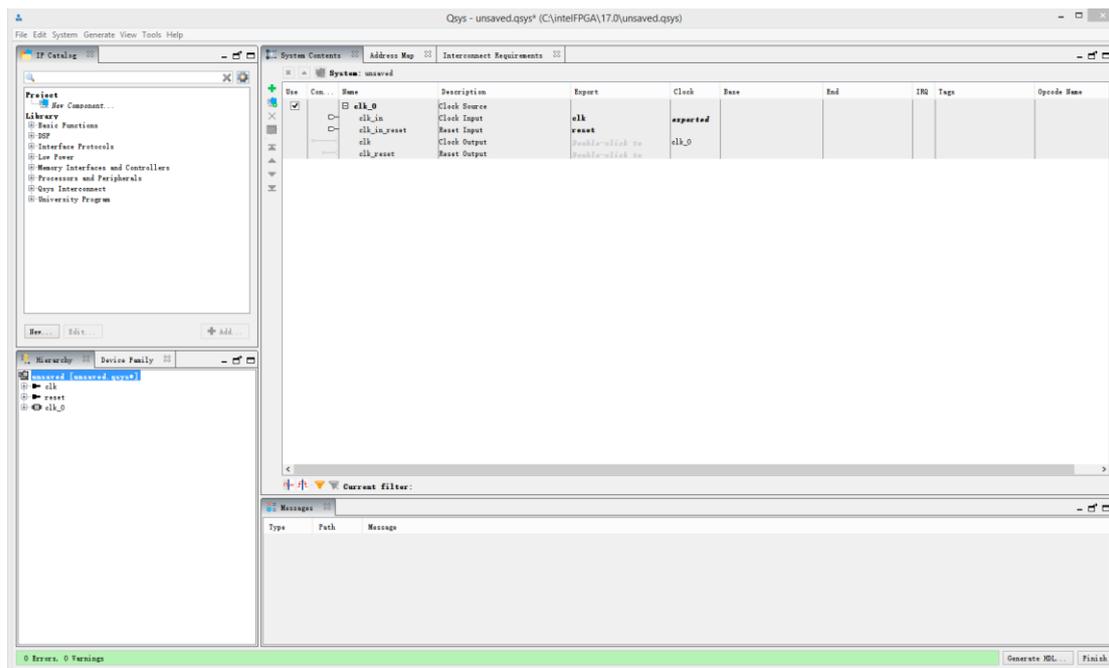


Figure 3-2 Qsys GUI (ver 17.0)

### 3.2.2.1 Altera VIP suite

The Video and Image Processing (VIP) Suite produced by Intel is a design tool and IP libraries for developing Video applications [39]. The video system is assembled by the Platform Designer tool of Quartus with the support of IP modules. The VIP IP modules offer standard interfaces to support control input, data input/output and external memory access. These interfaces facilitate the development of video systems through implementing the functions in Platform designer. VIP modules used in this research include the Clocked Video Input /

Output, Frame Buffers, etc. To understand the operation of the VIP modules several features of VIP suite are explained in the following sections.

***a) Avalon-ST Video Packet***

Although the VIP suite uses the Avalon-ST buses for transferring video streams, it also provides information on the video stream before each video data packet.

The packets of the Avalon-ST protocol are split into symbols. Each of the symbol represents a single data unit. For all packet types on an Avalon-ST interface, the number of symbols is sent in parallel and the bit width of all symbols is fixed [39]. The symbol bit width and number of symbols sent in parallel defines the structure of the packets.

The VIP suite defines the following three types of packet (shown in Table 3-1): Video data packets containing only uncompressed video data; Control data packets containing the control data which configure the cores for incoming video data packets; Ancillary (non-video) data packets containing ancillary packets from the vertical blanking period of a video frame

Another seven packet types are reserved for user applications, and five packet types are reserved for future use by Intel.

The packet type is defined by a 4-bit packet type identifier. This type identifier is the first value of any packet. It is the symbol in the least significant bits of the interface. Functions do not use any symbols in parallel with the type identifier.

Table 3-1 Packet types of VIP Packet [29]

Type Identifier	
<b>0</b>	Video data packet
<b>1-8</b>	User packet types
<b>9-12</b>	Reserved for future Altera use
<b>13</b>	Ancillary data packet
<b>14</b>	Reserved for future Altera use
<b>15</b>	Control data packet

Figure 3-3 is an example of transferring 12 packets with a 24-bit parallel video stream using the Avalon-ST interface.

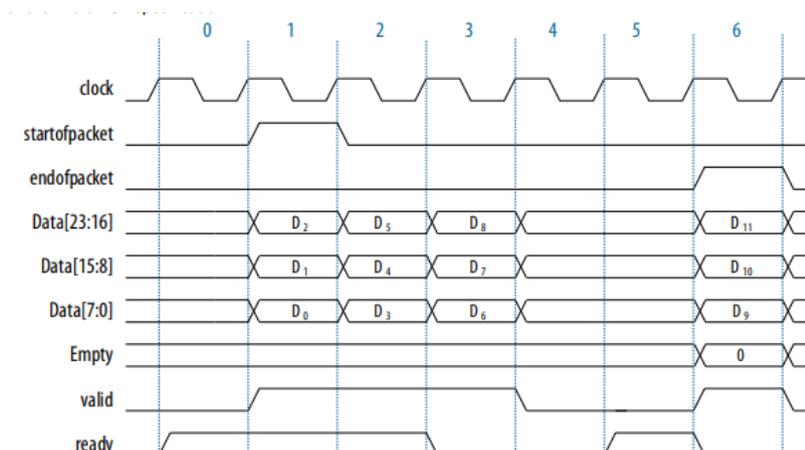


Figure 3-3 Example of a 24-bit Video stream with Avalon-ST format signal timing [39]

The data is transferring only when the valid signal is high and in the period of the startofpacket and endofpacket. As the ready latency is "1" in this example, when the ready signal falls in cycle 3, it means that it will not be ready to receive data in cycles 4 and 5. With the return of the ready signal in cycle 5, it will be ready to receive data in cycle 6.

For the symbols of D0, D1... shown in Table 3-1, it could be colour plane data from an Avalon-ST Video image packet or data from a control packet or a user packet [39]. The type of packet is determined by the lowest 4 bits of the first symbol transmitted as mentioned above.

### ***b) Colour Pattern***

The organization of the colour plane samples within a video data packet is referred to as the colour pattern.

This parameter also defines the bit width of the symbols for all packet types on an Avalon-ST interface. An Avalon-ST interface must be at least four bits wide to fully support the Avalon-ST Video protocol.

A colour pattern is represented as a matrix which defines a repeating pattern of colour plane samples that make up a pixel (or multiple pixels). The height of the matrix indicates the number of colour plane samples transmitted in parallel, the width determines how many cycles of data are transmitted before the pattern repeats.

Each colour plane sample in the colour pattern maps to an Avalon-ST symbol. The mapping is such that colour plane samples on the left of the colour pattern matrix are the symbols transmitted first. Colour plane samples on the top are assigned to the symbols occupying the most significant bits of the Avalon-ST data signal.

A colour pattern can represent more than one pixel. This is the case when consecutive pixels contain samples from different colour planes. There must always be at least one common colour plane among all pixels in the same colour pattern. Colour patterns representing more than one pixel are identifiable by a repeated colour plane name. The number of times a colour plane name is

repeated is the number of pixels represented [39]. Figure 3-5 below shows two pixels of horizontally subsampled Y' CbCr (4:2:2) where Cb and Cr alternate between consecutive pixels.

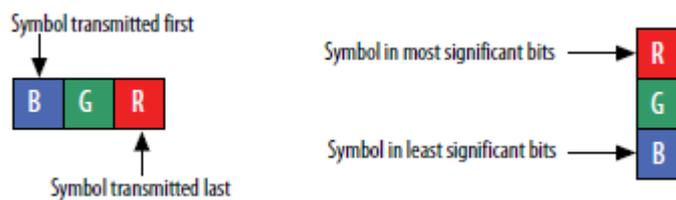


Figure 3-4 Symbol Transmission Order [39]



Figure 3-5 Horizontally Subsampled Y'CbCr [39]

### ***c) IPs cores Used in the system***

As previously mentioned, the Intel VIP suite includes several IP cores for real time image processing. To simplify the design process, a few IP cores from the VIP suite has been used in the system design. This section gives an introduction to the IP cores that have been used in the system.

#### *Clocked Video Input/output II*

Both Clocked Video Input and Output IP cores are used on in the system. At the start of the data flow process, the Clocked Video Input II (CVI II) receives the data from outside the FPGA either from a video decoder or from interface electronics for a CCD or CMOS camera. The CVI II IP converts the raw clocked video data into the Avalon-ST Video format, followed by the data packets. For

the output, after image processing, the Clocked Video Output II (CVO II) IP core processes the Avalon-ST Video format so that it can generate a video output for driving a DAC or generating appropriate digital signals. Both cores are compatible with video image standards (BT.656 [45], BT1120 [46], DVI etc.). These standards were widely used in High-Definition Multimedia Interface (HDMI), Serial Digital Interface (SDI) and DisplayPort.

The CVI II's is capable of spanning multiple clock domains as it needs to convert between the input pixel frequency which is a function of the image acquisition system (camera) and the clock frequency of the FPGA. The horizontal and vertical blanking signals are identified and stripped by the CVI II to leave just the active picture data which is forwarded to the next IP block.

The CVO II IP has corresponding capabilities to support various video formats at different operating frequencies. Furthermore, through applying the active picture packets and Avalon-ST Video control, it inserts horizontal and vertical blanking and generates the correct synchronization timing to convert the Avalon-ST video back to a clocked video output.

#### *Scaler II and Clipper II*

Both the Scaler II and the Clipper II IPs are relatively simple cores among the IP cores used in this research. The Scaler II IP core adjusts the resolution of the video streams, supporting the scaling algorithms of near-point, bilinear, bicubic and polyphase interpolation to either increase or decrease the image resolution. It can receive video data in either a 4:2:2 or 4:4:4 format and samples and utilizes the control packets to alter the input resolution. Moreover, both the output resolution and filter coefficients can be changed whilst the IP core is running by configuring the IP cores using a processor to access the IP

registers via the Avalon-MM slave interface. This allows the resolution to be changed, in real-time, under software control. In this project, the Scaler II IP is used to change the video input data into several different resolutions for testing the custom IP developed.

The Clipper II IP core provides a mechanism to reduce the video area, and change its aspect ratio, by removing the edges of the active video area. The specific active area can be adjusted by giving the offsets from a point or every border. The Clipper II IP core reads the Avalon-ST Video control packets to deal with any changes of input resolutions. It is used for clipping the frame to a regular size in this design.

#### *Colour Space Converter II*

The Colour Space Converter II IP core makes a video transformation between different colour spaces. It enables programmers to specify colours via utilizing three coordinate values. The IP core can be configured at run time using the Avalon-MM slave interface [39].

There are four noticeable features on this IP core. Firstly, it allows conversion from one space to another to be efficient and flexible. Then, it provides a few pre-sets conversions between standard colour spaces, such as CbCrY': SDTV to Computer B'G'R' and UVY' to Computer B'G'R'. Consequently, the entry of custom coefficients is permitted to be translated between any two of three-valued colour spaces video streams. Finally, it could support up to 4 pixels in every transmission.

#### *Deinterlacer II*

The Deinterlacer II IP core, also called the 4K HDR passthrough, provides four kinds of deinterlacing algorithms, consisting of Vertical Interpolation ("Bob"), Field weaving ("Weave"), Motion Adaptive and Motion Adaptive High Quality (Sobel edge interpolation). It is used to convert the Interlaced format into a normal progressive format [39].

Interlaced video is a standard popularly adopted in televisions standards like NTSC and PAL to give a perceived higher frame rate whilst reducing the transmission bandwidth. However, most modern LED and LCD displays adopt the progressive video format. Also, any subsequent spatial image processing for example edge detection, normally works on the full frame (both fields) rather than just a single field. Thus, the conversion is necessary for interlaced video input. This IP core also supports the pass-through of progressive video up to a resolution of 4K.

#### *Frame Buffer II*

The Frame Buffer II IP core buffers video frames consisting of interlaced or non-interlaced video fields into external Random-Access Memory. This is one of the most complex IP cores, it allows up to four pixels for each transmission and supports a configurable inter buffer offset to achieve the best interlacing of the memory's banks for maximum efficiency [39]. Modes of write-only and read-only are also supported.

The Frame Buffer II IP supports both double and triple buffering to implement various functions. Frame repeating or dropping are required in triple buffering when the input frame rate is not the same as the display frame rate. However, the frame rate for double buffering is the same between the input and output.

The Frame Buffer II IP core includes two main blocks, the writer to save input pixels to memory and the reader to retrieve video frames from the same position and then generate the output. Their operation is illustrated in Figure 3-6.

In addition, the Frame Buffer II IP core can be configured to be a Frame Writer only or a Frame Reader only. This requires it to be controlled by the processor continuously. This function is used to drop redundant frames when the processor can't process the frames in time in a FPGASoC co-processing system. It also repeats the previous frame when a new frame has not been delivered by the processing system.

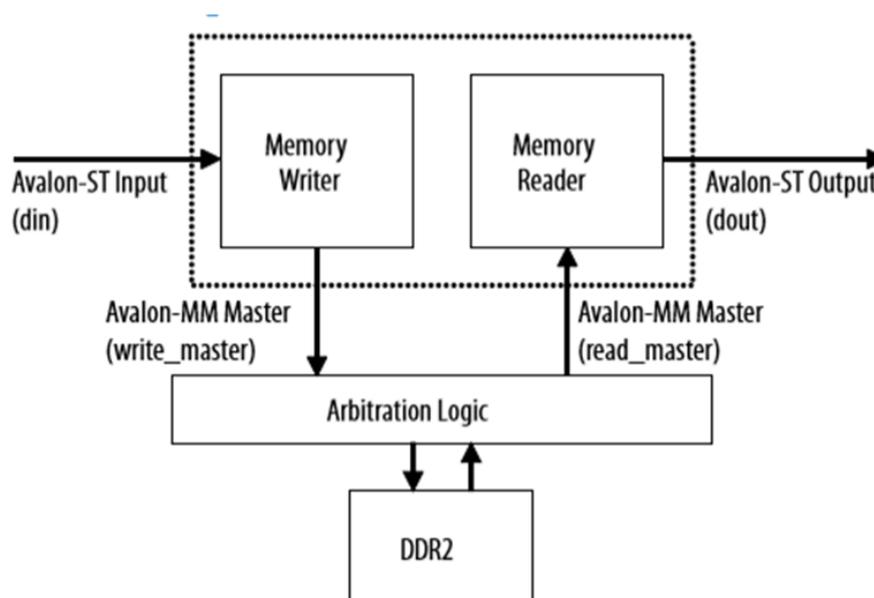


Figure 3-6 Frame Buffer Block Diagram [39]

### 3.2.3 ModelSim

The ModelSim-Intel FPGA Edition software is a version of the ModelSim software targeted for Intel FPGAs devices. The software supports Intel gate-

level libraries and includes behavioral simulation, HDL test benches, and Tcl scripting. It is a multi-language HDL simulation environment by Mentor Graphics, for simulation of hardware description languages such as VHDL, Verilog and SystemC, and includes a built-in C debugger. Simulation is performed using the graphical user interface (GUI), or automatically using scripts.

### ***3.2.5 DSP builder***

DSP Builder is a digital signal processing (DSP) design tool that allows push-button HDL generation of DSP algorithms directly from the MathWorks Simulink environment on Intel FPGAs. DSP Builder for Intel FPGAs tool generates high quality, synthesizable VHDL/ Verilog code from MATLAB functions and Simulink models. The generated RTL code can be used for Intel FPGA programming. DSP Builder for Intel FPGAs is widely used in radar designs, wireless and wireline communication designs, medical imaging, and motor control applications. This tool allows developers to design algorithms, set the desired data rate, clock frequency, and offers accurate bit and cycle simulation, synthesize fixed- and floating-point optimized HDL, auto-verify in ModelSim-Intel FPGA software, and auto-verify/co-simulate on hardware. DSP Builder for Intel FPGAs adds additional library blocks alongside existing Simulink libraries with the DSP Builder for Intel FPGAs (Advanced Blockset) and DSP Builder for Intel FPGAs (Standard Blockset) [47].

### 3.2.4 Intel SoC Embedded Design Suite (EDS)

The SoC EDS is generally used to develop the embedded software on Intel FPGA SoC devices. It is a comprehensive tool suite which embodies run-time software, utility programs, development tools and application examples used for application software development and initialising the firmware. It has the function of both application software development and expedite firmware. Currently, the EDS includes DS-5, the ARM Development Studio, which allows sophisticated debugging of both the ARM cores and the FPGA logic [48]. The EDS tool also includes both C and C++ toolchains for ARM Linux development. Within the command shell of EDS (a version of Cygwin [49]), it is possible to cross compile C/C++ programs that run on the SoC processors.

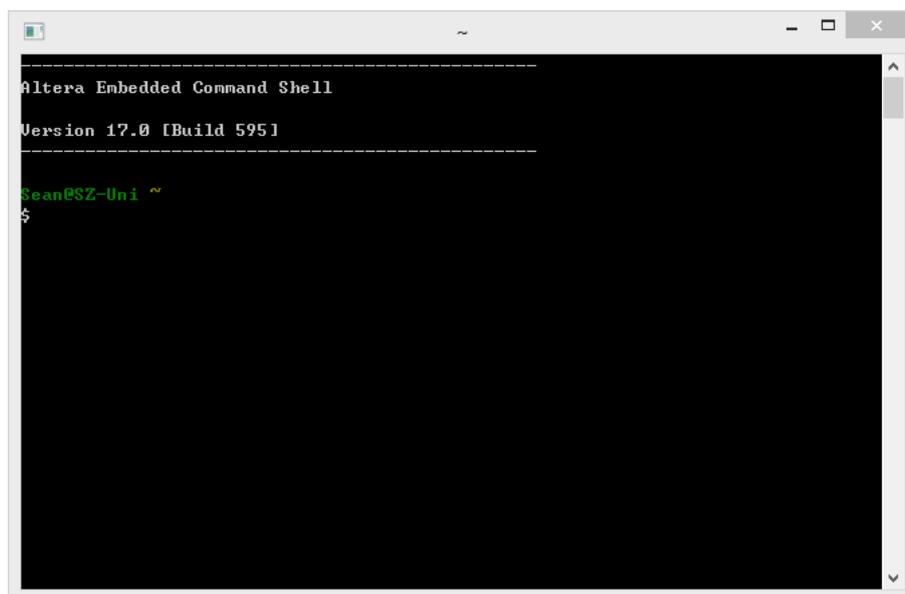


Figure 3-7 Altera EDS command Shell

### 3.2.5 Timing Analyzer

The Timing Analyzer validates the timing performance of all logic in design using industry-standard constraint, analysis, and reporting methodology [50]. The Intel Quartus Prime software generates timing analysis data by default during design compilation [50]. The timing analysis process involves running the compiler, specifying timing constraints, and viewing timing analysis reports like Figure 3-8.

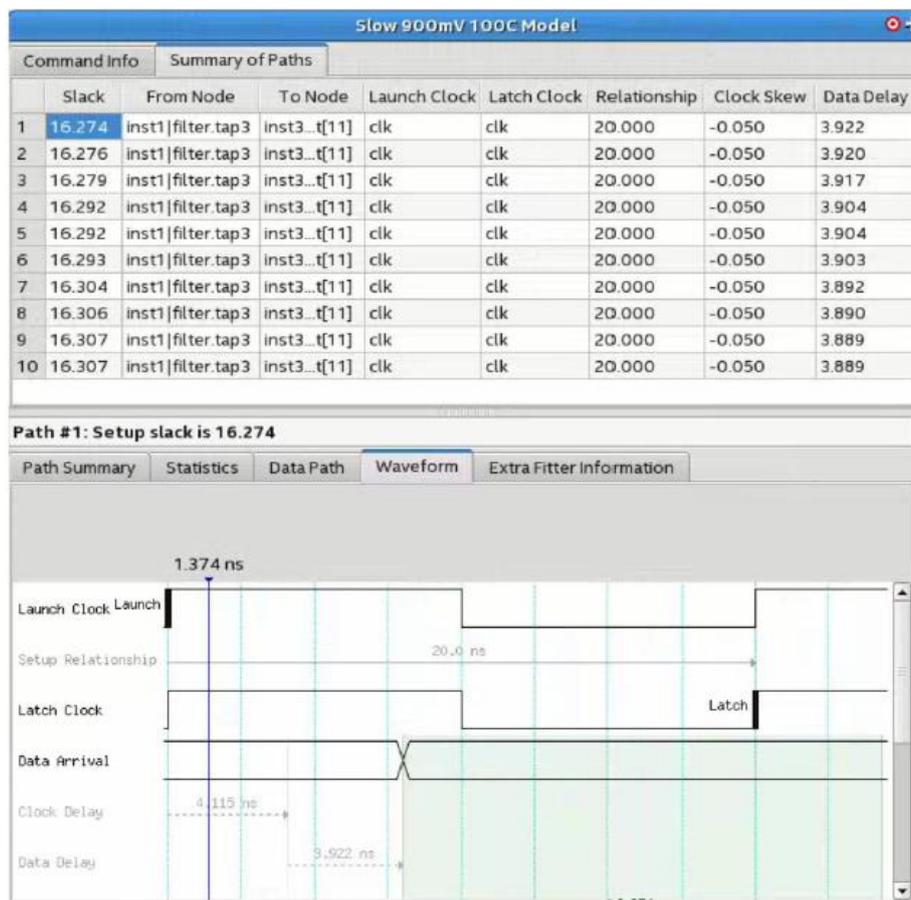


Figure 3-8 Timing report of Timing Analyzer [50]

### ***3.3 Hardware Environment***

Although the FPGA is considered as the core part in this real-time system, other hardware devices are also important for implementation of image processing. This section gives a brief introduction about the hardware environment used in this project. As previous discussed, the whole system includes 3 sub-systems: the video acquisition system, video processing system and video display/control system. Specific to hardware devices, it generally involves three types of devices; the video input device; the development board (DE1-SOC); and the video display device. Sections of the video acquisition system and the video display system are integrated on the development board.

#### ***3.3.1 Development board***

As mentioned previously, the DE1-SOC Development Board with an integrated Cyclone V SE FPGA was selected as the development environment for this project, mainly on the grounds of cost and availability. This FPGA chip consists of 110k programmable logic elements and a Dual-core ARM Cortex-A9 processor.

Therefore, there are two sets of systems combined on the board, the FPGA and the Hard Processor System (HPS). Each of these systems is connected to a different set of devices (as shown in Figure 3-9). The kernel part of both systems (the Cyclone V FPGA and ARM Cortex-A9 processor) are integrated into one single chip which share interconnection buses between the FPGA and the ARM processor.

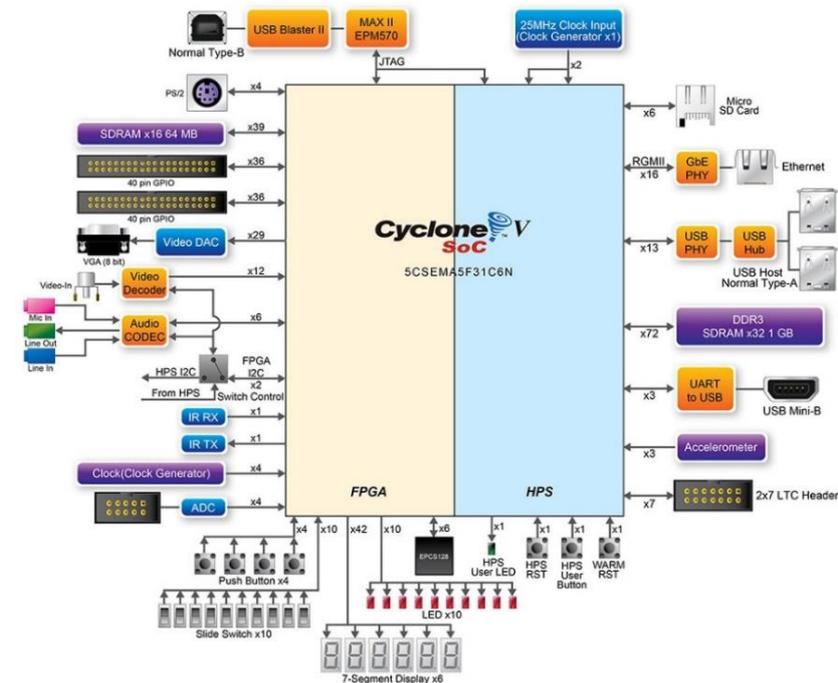


Figure 3-9 Architecture of DE1-SOC device [51]

The details of both systems are listed in Table 3-2. As listed in the table, the FPGA-part system is integrated with a 64MB SDRAM chip with a 16-bit bus width working at 120MHz. This can provide a memory bandwidth up to  $120\text{MHz} \times 16 \text{ bits} = 228\text{MB/s}$  theoretically which is not sufficient for buffering a 24-bit RGB video stream with  $1920 \times 1080 @ 60\text{fps}$  which requires at least  $1920 \times 1080 \times 24\text{bits} \times 60\text{fps} = 355.8\text{MB/s}$  bandwidth. But it is sufficient for a  $1024 \times 768 @ 30\text{fps}$  video stream which required about  $1024 \times 768 \times 24\text{bits} \times 30\text{fps} = 67.7\text{MB/s}$ .

Table 3-2 Hardware of both FPGA and HPS system [51]

FPGA	HPS
Altera Cyclone® V SE device	800MHz Dual-core ARM Cortex-A9 MPCore processor
Altera Serial Configuration device	1GB DDR3 SDRAM (32-bit data bus)
USB Blaster II (on board) for programming; JTAG Mode	1 Gigabit Ethernet PHY with RJ45 connector
64MB SDRAM (16-bit data bus)	2-port USB Host, Normal Type-A USB connector
4 Push-buttons	Micro SD card socket
10 Slide switches	Accelerometer (I2C interface + interrupt)
10 Red user LEDs	UART to USB, USB Mini-B connector
Six 7-segment displays	Warm reset button and cold reset button
Four 50MHz clock sources from clock generator	One user button and one user LED
24-bit CD-quality audio CODEC with line-in, line-out, and microphone-in jacks	LTC 2x7 expansion header
VGA DAC (8-bit high-speed triple DACs) with VGA-out connector	
TV Decoder (NTSC/PAL/SECAM) and TV-in connector	
PS/2 mouse/keyboard connector	
IR receiver and IR emitter	
Two 40-pin Expansion Header with diode protection	
A/D Converter, 4-pin SPI interface with FPGA	

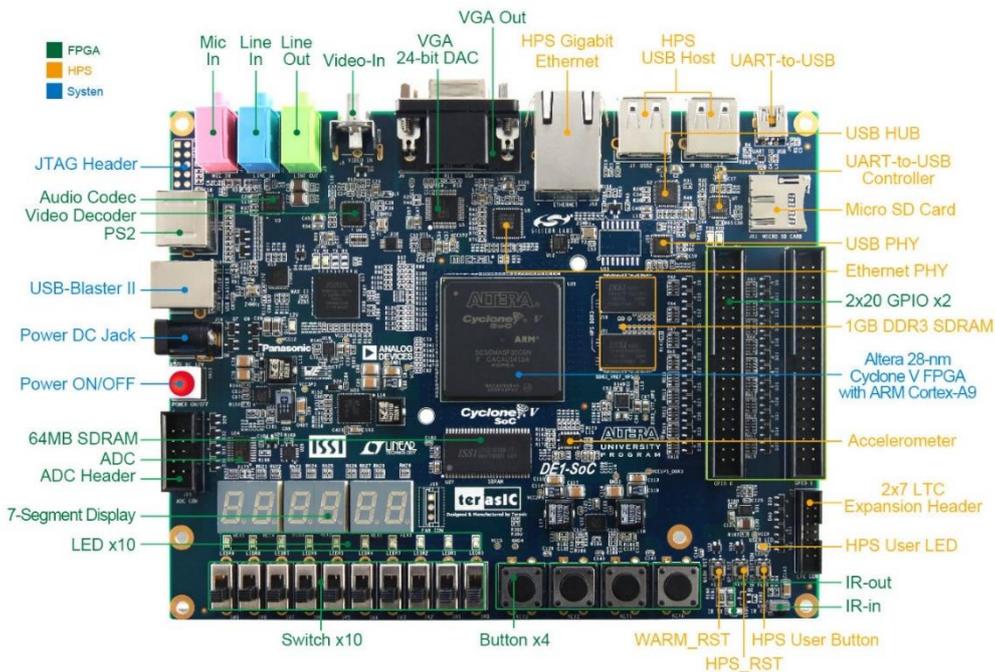


Figure 3-10 DE1-SOC FPGA board Top View [51]

### 3.3.2 Video input devices

In the video processing system, digital or analogue cameras are used as the video input device to acquire images and video into the system. Therefore, several cameras have been considered as video input devices which include an NTSC analogue camera, a TRDB-D5M Camera and Logitech C270 USB webcam. Each camera has different interface ports and work at different resolutions, which means each of them have their own advantages.

### 3.3.2.1 NTSC Analog camera

As shown in Table 3-2, the DE1-SOC board provides a TV decoder chip which supports NTSC/PAL/SECAM format and a TV-in connector. Therefore, an NTSC camera (Figure 3-11) was initially considered as the video input device.



Figure 3-11 An NTSC Camera

However, with the NTSC format video as input, it is required to deinterlace the video stream to normal RGB video stream for image processing. The deinterlacer would use a significant amount of resources on FPGA and generate a large latency to the system. In the meantime, after deinterlacing the video stream have a resolution of 720x480 which is much lower than the D5M digital camera.

### 3.3.2.2 Terasic TRDB-D5M Digital Camera

As the development board has two 40-pin expansion headers, a TRDB-D5M Camera [52] daughter card (Figure 3-12) could be chosen. This digital camera offers various resolutions up to 2592x1944@15fps or 640x480@70fps both with an RGB Bayer pattern. Moreover, it provides external digital controls for the frame rate and frame size which provides more complexity [52]. It requires a

colour format transformation which transforms the video stream from the RGB Bayer pattern to RGB for further processing.

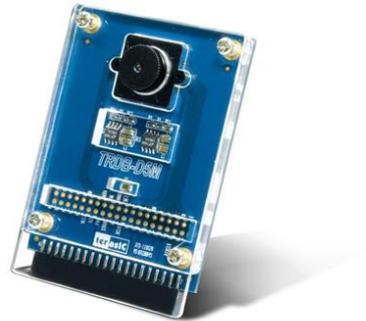


Figure 3-12 TRDB-D5M Camera Daughter Card [52]

### 3.3.2.3 Logitech C270 USB webcam

Another potential video input device is a USB webcam as the board provides two USB 2.0 ports. The Logitech C270 webcam (Figure 3-13), one of the most common USB webcams on the market was picked as input device. Its specification is listed in the Table 3-3 below.

Table 3-3 Specification of Logitech C270 webcam [50]

<b>Camera Specifications:</b>	
<b>USB Type</b>	High Speed USB 2.0
<b>Video Capture (4:3 SD)</b>	320x240, 640x480, 800x600
<b>Frame Rate (max)</b>	30fps @ 640x480
<b>UVC-compatible</b>	Yes
<b>Supported output format</b>	24 bits sRGB/YUV



Figure 3-13 Logitech C270 webcam [53]

However, the USB ports are not directly accessible from the FPGA as they are interfaced directly to the HPS system. Therefore, it can only work in the OS on the HPS with both the USB driver and camera driver correctly configured in the operating system of the ARM processor.

### ***3.3.3 Video display device***

The last part of the hardware environment is a video display device.

A video display device is an output device to present the visual results from the system. There are several formats of interfaces on market which includes VGA, DisplayPort, HDMI etc. As the DE1-SOC board contains a VGA interface port, a VGA supported monitor has been picked up as the video display device. The selected monitor works at 1440x900@60Hz which is sufficient for this research.

## ***3.4 Summary***

This chapter presented the design tools and hardware environment been used in this research. At first, as a development toolkit, Quartus II launched by Intel was selected to build the system's hardware architecture. Along with the HDL

design procedure mentioned in last chapter, the designer initially programmed the core IP using Intel Platform Designer, then linked it up with the established IP modules available in the system library via the Altera VIP suite, which could save the time in the process of design. After the original logic was established, Modelsim was used for behavioral simulation. The waveform files would be generated, and the signal waveform would be output to help verify the logic functions. Finally, the timing analysis would be carried on with the Timing Analyzer. Moreover, another tool from Quartus II, the EDS was used to develop the embedded software on the FPGA SoC devices. This part of design will be further discussed in Chapter 5.

This chapter also introduced the hardware environment. A complete real-time processing system consisted of the video input devices, the development board and the video display device. Considering the cost, the availability, the DE1-SOC Development Board with an integrated Cyclone V SE FPGA which contained a Dual-core ARM Cortex-A9 processor and 110K programmable logic elements was used. It is a combination of the FPGA and the Hard Processor System. Concerning the input devices, there were three selections, the NTSC analog camera, the TRDB-D5M Camera and the Logitech C270 USB webcam. Each of them has unique advantages. As for the output device, the monitor works at 1440x900@60Hz which was sufficient.

## ***Chapter 4 Real time image processing on FPGAs with the HDL Approach***

### ***4.1 Introduction***

Based on the background discussed earlier, a real-time image processing system has been developed and evaluated on the DE1-SOC development board. In this chapter, detailed descriptions of the system developed are presented with an emphasis on an FPGA configured using the Hardware Description Language (HDL) approach.

This chapter focuses on researching the relationship between hardware resource usage and latency with resolution and accuracy in implementing the hardware algorithms using the HDL approach. Canny Edge Detection is used as the demonstration for the hardware algorithm implementation. There are two methods implemented for the demonstration, one is focused on low-latency and low resource usage, whilst the other focuses on accuracy. The advantages of both methods are given in this chapter for guidance.

In section 4.2, an overview of the Canny Edge Detection algorithm is presented. Section 4.3 shows the implementation of the system algorithm with both methods. Section 4.4 introduces the design of the system architecture. Section 4.5 presents results and a discussion of the system. In the final section, a summary of this chapter is presented.

## **4.2 The Canny Edge Detection Algorithm**

As a popular algorithm used in the pre-processing stages of machine vision applications, an edge detection algorithm is a perfect solution for implementation on an FPGA platform as it mainly involves convolution methods which are a mathematical way of combining two functions to form a third function. There are several kinds of edge detection methods which could be chosen, including the Sobel detector, Gauss-Laplace detector, Canny detector, Kirsch detector, Robert detector and Prewitt detector. They present different performances based on the complexity of the edge computation and the ability of edge extraction algorithm when the image suffers from heavy noise contamination. However, most of them do not offer the noise reduction solution with the restriction of simple gradient computation.

The Canny Edge Detector [54] is regarded as one of the most reliable algorithms because it shows good performance and can achieve a low error-rate and improves the identified edges' localization. A typical Canny algorithm is comprised of the following steps.

### **4.2.1 Gaussian Filtering**

The first step is to filter out any noise in the original image before trying to locate and detect any edges. As Gaussian filtering can be implemented with a convolution method, it is appropriate to use it in the Canny algorithm. A 2-dimensional Gaussian function is described in equation (4.1):

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\left(\frac{x^2+y^2}{2\sigma^2}\right)} \quad (4.1)$$

where  $x$  represents the distance from the origin in the horizontal axis,  $y$  is the distance from the origin in the vertical axis, and  $\sigma$  is the standard deviation.

To implement the Gaussian function using a convolution method, it is necessary to calculate the convolution mask for the Gaussian function. A convolution (kernel) mask is usually much smaller than the actual image. As a result, the mask is slid over the image and applied at each location in the image. The larger the kernel size is, the lower the detector's sensitivity to noise. The larger the value of  $\sigma$ , the result becomes more smoothing with less noise. However, with more smoothing of the image, the less edges will be detected by the detector. Figure 4-1 shows the result of a Gaussian filter with various of values of  $\sigma$ .



**Figure 4-1 Result on different  $\sigma$  values after Gaussian filtering**

It is common practice to use a 5x5 kernel with  $\sigma=1.4$  which is calculated as shown in Figure 4-2.

$$\frac{1}{159}$$

2	4	5	4	2
4	9	12	9	4
5	12	15	12	5
4	9	12	9	4
2	4	5	4	2

Figure 4-2 5x5 Gaussian Filter Kernel with  $\sigma = 1.4$

### 4.2.2 Sobel edge detector

After reducing the noise and smoothing the image, the next step is to find the intensity gradient of the image by performing standard Sobel edge detection. The Sobel operator uses a pair of 3x3 convolution masks, one estimating the gradient in the vertical direction (x) and the other estimating the gradient in the horizontal (y) which are shown in equation (4.2).

$$G'_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad G'_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad (4.2)$$

Then, it can calculate the approximate absolute gradient magnitude (edge strength) at each pixel as shown by equation (4.3):

$$|G'| = \sqrt{G'_x{}^2 + G'_y{}^2} \quad (4.3)$$

Simultaneously, the gradient angle of each pixel can be calculated as shown in equation (4.4):

$$\theta = \arctan\left(\frac{G'_y}{G'_x}\right) \quad (4.4)$$

Figure 4-3 shows the Sobel result after Gaussian filtering of the original image.



Figure 4-3 Example of Sobel Result

### ***3) Non-maximum suppression***

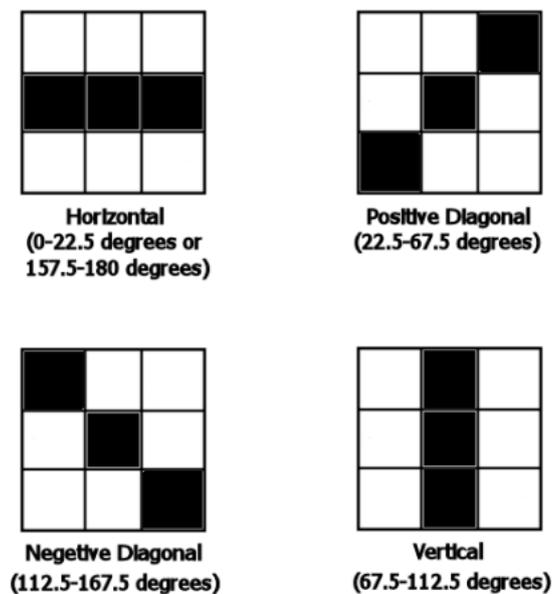
After the magnitude and gradient angles of each pixel are calculated, the result may contain thick edges which contains spurious results on the edges. The use

of non-maximum suppression for each pixel in the previous result image is to sharpen the edges.

Finding the gradient direction for each pixel, for a 3x3 image result, has 4 range of directions which is decided by the gradient angle of each pixel:

- If the gradient angle is 0-22.5 degrees or 157.5-180 degrees, the direction of the pixel is horizontal.
- If the gradient angle is from 22.5 degrees to 67.5 degrees, the direction of the pixel is positive diagonal.
- If the gradient angle is from 112.5 degrees to 157.5 degrees, the direction of the pixel is negative diagonal.
- If the gradient angle is from 67.5 degrees to 112.5 degrees, the direction of the pixel is vertical.

As shown in Figure 4-4, each dark area represents an edge pixel.



**Figure 4-4 Four possible directions of the edges**

Once the orientation of each pixel has been calculated, the second step is to compare the magnitude value of each pixel with the two pixels next to it but in different directions. If the magnitude of the current pixel is the largest compared to the other 2 pixels this pixel will be preserved as a thin edge. Otherwise, the value will be suppressed. Figure 4-5 shows the pixels (marked in red) that need to be compared for each condition.

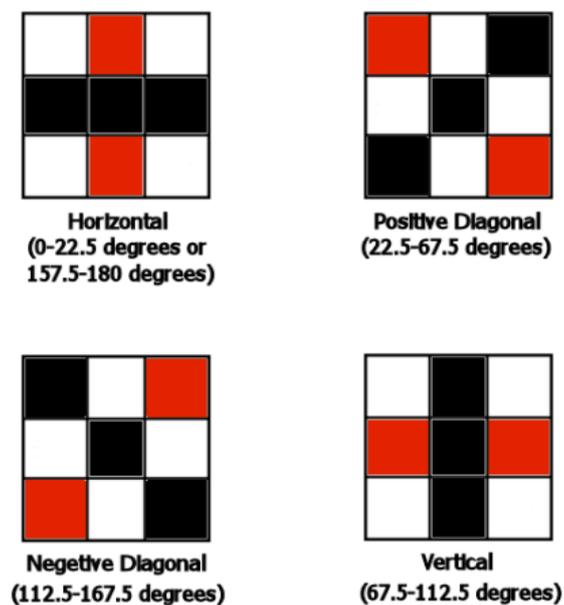


Figure 4-5 Pixels need to be compared with

#### ***4) Thresholding with Hysteresis***

After application of non-maximum suppression, the remaining edge pixels provide a more accurate representation of the real edges in an image. However, some edge pixels remain which are caused by noise or colour variations. To account for these spurious values, it is essential to filter out edge pixels with a low gradient value and preserve edge pixels with a high gradient value. This is

accomplished by selecting high and low threshold values. If an edge pixel's gradient value is higher than the high threshold value, it is marked as a strong edge pixel. If an edge pixel's value is smaller than the low threshold value, it will be suppressed. Canny [36] recommends that the ratio of the high to low limit be in the range two or three to one, based on predicted signal-to-noise ratios. Pixels which are between the two thresholds are accepted if they are connected to a strong edge pixel.

The full process of a Canny Edge Detection method is shown in Figure 4-6.



**Figure 4-6 Stages of a Canny Edge Detection process**

The desired result of the full Canny Edge detection processing is shown in Figure 4-7



**Figure 4-7 Example of Canny Result**

### **4.3 Algorithms Implementation**

The proposed Canny edge detection implementation is targeting for a FPGA based real time vision system, where the whole algorithm is coded using the Verilog hardware description language firstly. Consequently, the edge detector's performance must achieve a real-time response with low latency, while at the same time use the limited logic resources available so that the system can fit in the FPGA. Therefore, it is required to discuss the methods used to implement the algorithm.

#### **4.3.1 Grayscale Transformation**

As mentioned previously, to implement the Canny Edge Detector, firstly it needs to transform the original RGB image to a Grayscale image. A Grayscale image is an image where each pixel of the image contains only intensity information. For an RGB pixel the Grayscale value of the pixel is computed as equation (4.5):

$$Y' = 0.299R + 0.587G + 0.114B \quad (4.5)$$

As previously mentioned above, both latency and resource usage are of concern in the real-time system, therefore it is required to simplifying the arithmetic used in the system. Especially for multiplications and divisions as they may take several clock cycles for each calculation. If multiple multiplications and divisions are implemented in one algorithm block, it will increase the complexity of synchronizing the data.

Therefore, rather than using floating point arithmetic, multiply and divide can be implemented using shift operations. Shifting to the left is equal to multiply by 2, whilst shifting to right is equal to divide by 2. It is possible to simplify the original equation to an approximate result which is given in equation (4.6):

$$\begin{aligned}
 Y' &= 0.299R + 0.587G + 0.114B \approx 0.297R + 0.586G + 0.113B & (4.6) \\
 &\approx \frac{76 \times R + 150 \times G + 29 \times B}{256} \\
 &= \frac{((64 \times R + 8 \times R + 2 \times R) + (128 \times G + 16 \times G + 4 \times G + 2 \times G) + (32 \times B - 4 \times B + B))}{2^8} \\
 &= \frac{(2^6 \times R + 2^3 \times R + 2 \times R) + (2^7 \times G + 2^4 \times G + 2^2 \times G + 2 \times G) + (2^5 \times B - 2^2 \times B + B)}{2^8}
 \end{aligned}$$

As all the calculations have been implemented using a shift approach, an approximate RGB to Grayscale transformation can be implemented without using any multiplications or divisions. Furthermore, if a more accurate result was required it could be achieved by increasing the magnitude of the divisor. For example, as shown in equation (4.7):

$$\begin{aligned}
 Y' &= 0.299R + 0.587G + 0.114B & (4.7) \\
 &\approx \frac{306 \times R + 601 \times G + 117 \times B}{1024}
 \end{aligned}$$

However, as the computation becomes better in accuracy, the system becomes more complex which results in more resources required and a longer latency.

### 4.3.2 Gaussian Filtering

After the Grayscale value has been calculated, the first step of the Canny Edge Detector can be implemented, the Gaussian filter. As mentioned previously, the Gaussian filter can be implemented using a convolution method. In order to implement a convolution algorithm, it is required to use a method called window filtering.

For example, for a 3x3 kernel convolution algorithm, it has a kernel as equation (4.8).

$$X = \begin{bmatrix} X_1 & X_2 & X_3 \\ X_4 & X_5 & X_6 \\ X_7 & X_8 & X_9 \end{bmatrix} \quad (4.8)$$

To implement it on each pixel of a video stream, 3-line buffers are required to buffer 3 lines at a time, and output 3 lines at same time as shown in Figure 4-8.

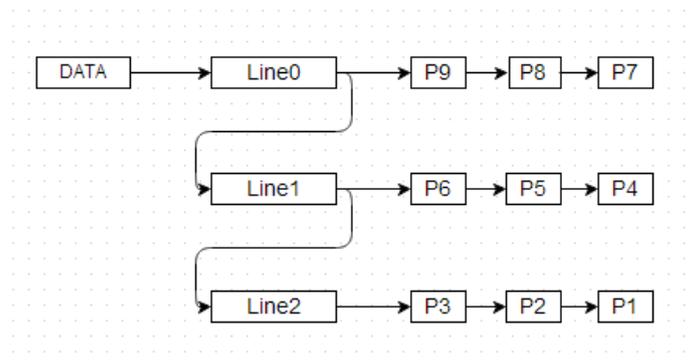


Figure 4-8 Data flowchart with 3 line-buffers

Then each pixel with its surrounding 8 pixels could form a 3x3 original image kernel as shown in Figure 4-9.

$P_1$	$P_2$	$P_3$
$P_4$	$P_5$	$P_6$
$P_7$	$P_8$	$P_9$

**Figure 4-9 the kernel of the original image**

Using this kernel, the result of each pixel is given by equation (4.9).

$$P' = X * P = X_1 \times P_1 + X_2 \times P_2 + X_3 \times P_3 + X_4 \times P_4 + X_5 \times P_5 + X_6 \times P_6 + X_7 \times P_7 + X_7 \times P_7 + X_7 \times P_7 \quad (4.9)$$

Continuously, as each new pixel is received at the output, the system will implement the desired algorithm on each pixel.

In this case, as discussed previously, a 5x5 Gaussian filter is applied. Therefore, it is required to buffer 5 lines of the video stream.

Each pixel is calculated as shown in equation (4.10).

$$P' = \frac{1}{159} \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} * P \quad (4.10)$$

To reduce the cost and latency of the system, an approximate result with fixed point mathematics could be implemented as equation (4.11)

$$P' \approx \frac{1}{128} \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} * P \quad (4.11)$$

### 4.3.3 Sobel Edge Detector & Non-maximum Suppression

As previously discussed, the third step of the algorithm is to calculate the gradient and direction of each pixel. With the methods discussed previously, the gradients in both the vertical and horizontal and horizontal directions can be calculated. However, to calculate the absolute gradient of each pixel, it is usually to use an approximate equation (4.12) to simplify the calculation:

$$|G'| = \sqrt{G'_x{}^2 + G'_y{}^2} \approx |G'_x| + |G'_y| \quad (4.12)$$

Afterwards, as the directions of each pixel is in one of four given conditions, a rapid method has been used to reduce the latency and resources used by avoiding the arctan calculation and division. As the four conditions of magnitude angles are:

$$(0^\circ, 22.5^\circ] \text{ or } (157.5^\circ, 180^\circ]$$

$$(22.5^\circ, 67.5^\circ] \quad (4.13)$$

$$(67.5^\circ, 112.5^\circ]$$

$$(112.5^\circ, 157.5^\circ]$$

It can be simplified as four conditions for  $\tan \theta$ :

$$(-0.414, 0.414]$$

$$(0.414, 2.414] \tag{4.14}$$

$$(-0.414, -2.414]$$

$$(-\infty, -2.414) \text{ or } (2.414, +\infty)$$

As  $\tan \theta = \frac{G'_y}{G'_x}$ , it can be simplified with an approximate result to avoid division and decimals comparison as:

$$\frac{G'_y}{G'_x} > 0.414$$

$$\frac{G'_y}{G'_x} > 0.414 \approx 0.5 \tag{4.15}$$

$$2G'_y > G'_x$$

In summary, the four conditions are approximated as shown in Table 4-1:

**Table 4-1 The approximate conditions of the four directions**

Angles	Tangent Value	Approximate Rapid Method Conditions
$(0^\circ, 22.5^\circ]$ or $(157.5^\circ, 180^\circ]$	$(-0.414, 0.414]$	$0 <  2G'_y  \leq  G'_x $
$(22.5^\circ, 67.5^\circ]$	$(0.414, 2.414]$	$ G'_x  <  2G'_y  \leq  5G'_x $ and $\frac{G'_y}{G'_x} > 0$
$(67.5^\circ, 112.5^\circ]$	$(-0.414, -2.414]$	$ G'_x  <  2G'_y  \leq  5G'_x $ and $\frac{G'_y}{G'_x} < 0$
$(112.5^\circ, 157.5^\circ]$	$(-\infty, -2.414)$ or $(2.414, +\infty)$	$ 2G'_y  >  5G'_x $

As both the absolute values can easily be achieved in Verilog and using this method can reduce the latency and the resources used in the FPGA. However, as this method uses an approximate result, some information is lost during processing. Table 4-2 shows the angle errors in the calculation.

**Table 4-2 Angle Error with approximate value for the rapid method**

Expected Angle	Expected Value	Rapid Approximate Value	Rapid Approximate Angle	Angle Lost
22.5°	tan22.5°	0.5	tan26.5°	4°
67.5°	tan67.5°	2.5	tan68.2°	0.7°
112.5°	tan112.5°	-2.5	tan111.8°	-0.7°
157.5°	tan157.5°	-0.5	tan153.5°	-4°

However, this method could be easily improved by increasing the precision. Thus, the 4 conditions are approximated and given in Table 4-3. Meanwhile, with the improvement of accuracy, the angle error can be reduced as shown in

Table 4-4.

**Table 4-3 The approximate conditions for the 4 directions**

Angles	Approximate Accurate Method Conditions
$(0^\circ, 22.5^\circ]$ or $(157.5^\circ, 180^\circ]$	$0 <  100G'_y  \leq  41G'_x $
$(22.5^\circ, 67.5^\circ]$	$ 41G'_x  <  100G'_y  \leq  241G'_x $ and $\frac{G'_y}{G'_x} > 0$
$(-0.414, -2.414]$	$ 41G'_x  <  100G'_y  \leq  241G'_x $ and $\frac{G'_y}{G'_x} < 0$
$(-\infty, -2.414)$ or $(2.414, +\infty)$	$ 100G'_y  >  241G'_x $

**Table 4-4 Angle Error with the approximate value using the accurate method**

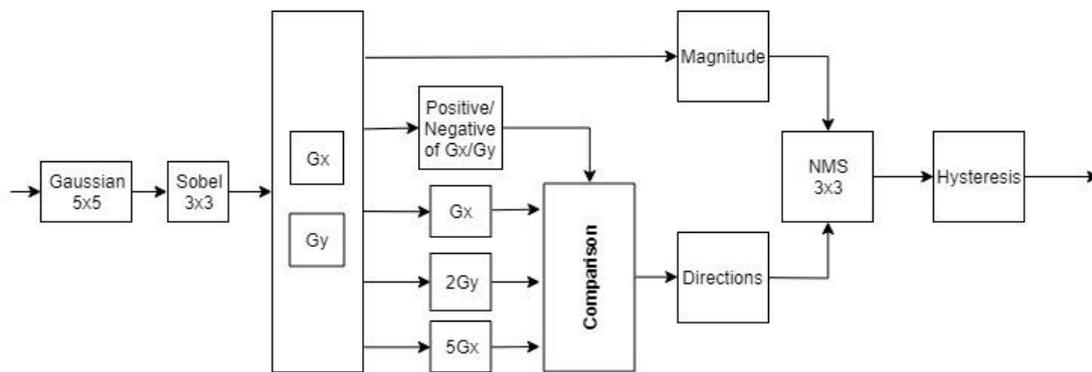
Expected Value	Accurate Approximate Value	Accurate Approximate Angle	Angle Lost
Tan $22.5^\circ$	0.414	Tan $22.490^\circ$	$0.01^\circ$
Tan $67.5^\circ$	2.414	Tan $67.498^\circ$	$0.002^\circ$

### ***4.3.4 Thresholding with Hysteresis***

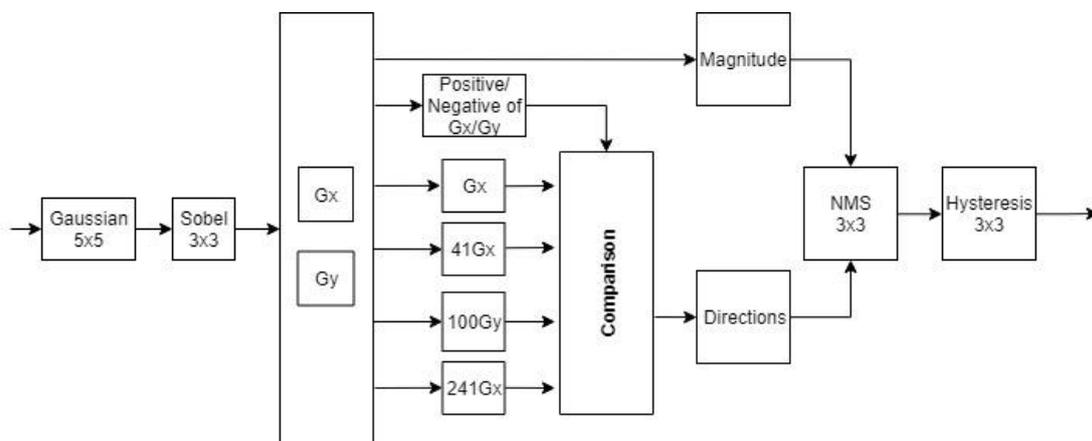
As previous discussed, two thresholds are required to threshold the result. To reduce the resources used and the latency, as with the rapid method, two fixed thresholds are used to filter the result. Then every thin edge pixel will be directly considered as an edge pixel.

With the accurate method, the threshold is determined by the largest value of the magnitude from the previous result. The high threshold is set as a 1/6 of the largest magnitude empirically. The ratio of the low threshold versus the high threshold is 1:2 as Canny suggested [45]. If any pixel is between the two thresholds, it will be checked to see if it is surrounded by any thin or thick edge pixels. If it is, it will be marked as an edge pixel. If it is not, it will be suppressed.

Figure 4-10 and Figure 4-11 show the full algorithm architectures of the two methods.



**Figure 4-10 Rapid method with Canny Edge Detection**



**Figure 4-11 Accurate Method with Canny Edge Detection**

## ***4.4 System Architecture***

When the algorithm has been implemented in the system, there are still a few IPs from the VIP suite required for the full system implementation. For these tests the D5M camera was used so the video input received the raw video streaming from the D5M camera, then the Bayer Resampler resampled the video format from the Bayer pattern to the RGB pattern. This produces a 24-bits RGB pattern video stream suitable for the designed Canny Edge Detection IP.

A frame buffer is then necessary for buffering the video to change the frame rate to that of the output device. Then Video Output IP is required to transform the video stream from Avalon Streaming to the VGA format signals. The frame buffer buffered the video stream into DDR3 memory which is on the HPS side of the FPGA rather than the FPGA side. As the HPS IP provides an fpga2sdram pipeline, the Frame Buffer IP could directly read and write to the DDR3 memory connected to the HPS.

Simultaneously, the designed CED IP keeps the original data for further processing, it combines the 24-bits original RGB data and the processed 8-bits edge data to a 32-bits Y-RGB video stream. The RGB data will be delayed in order to synchronize the processed edge data from the CED IP.

The architecture of the system with the D5M camera is shown in Figure 4-12. The red arrows in the figure shows the interconnections used to buffering the video stream. The video format conversion of the whole system is shown in Figure 4-13. Figure 4-14 shows the design in the Platform Designer system whilst Figure 4-15 shows the interconnections of designed CED IP with the accurate method.

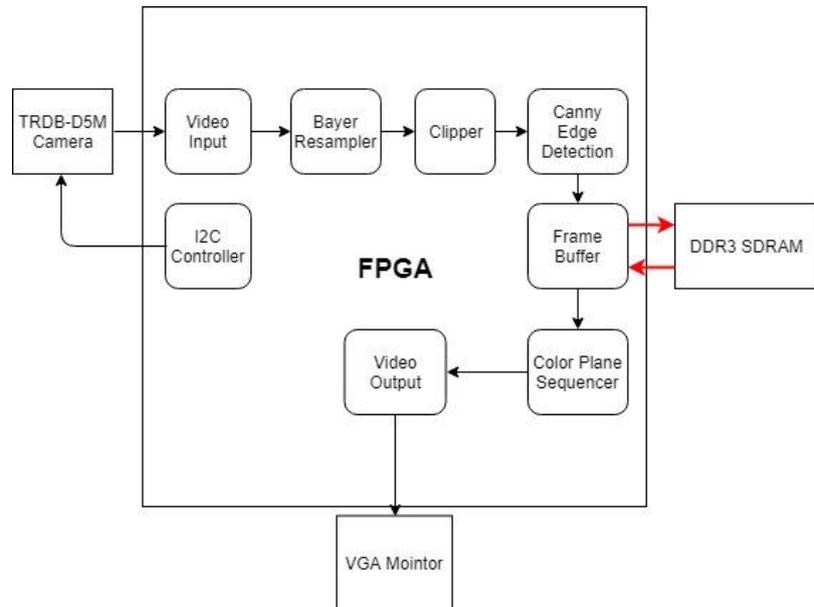


Figure 4-12 Architecture of the System with TRDB-D5M

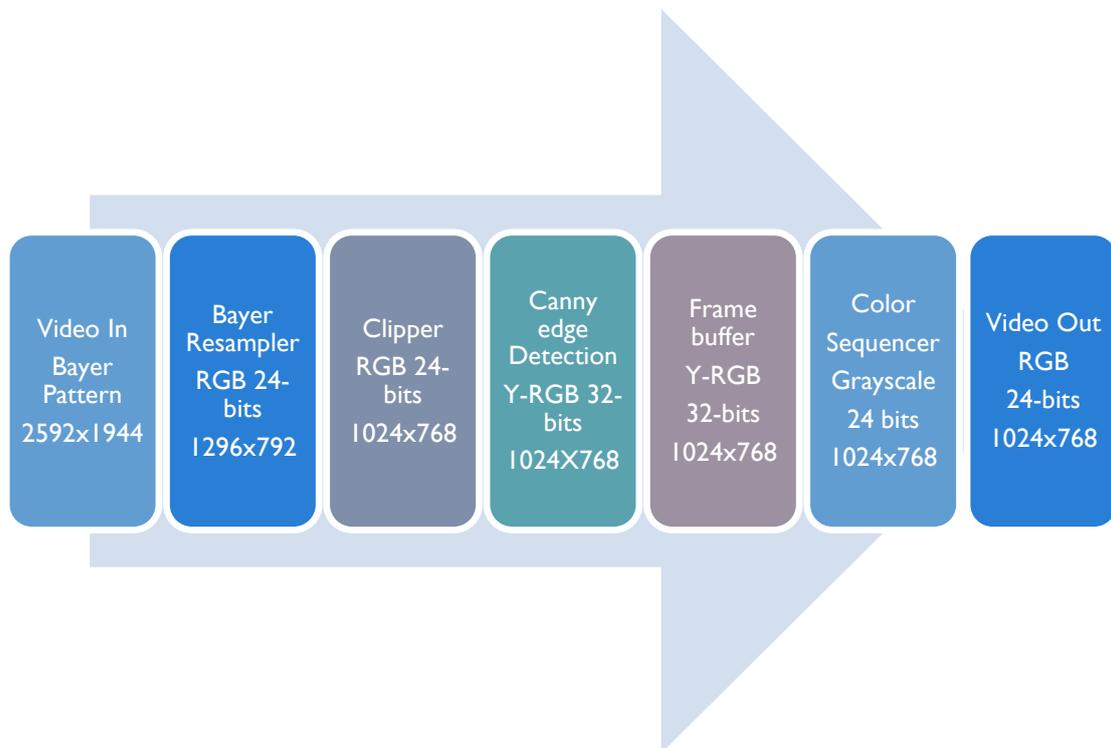


Figure 4-13 The Video Format conversion in the system

Use	C...	Name	Description	Export	Clock	Base	End	IRQ	Tags
<input checked="" type="checkbox"/>		hps_0	Arria V/Cyclone V Hard Processor System		multiple	# 0x0000_0000	0xffff_ffff		
<input checked="" type="checkbox"/>		master_secure	JTAG to Avalon Master Bridge		DE1_CLKs_...	# 0x0001_0000	0x0001_0007		
<input checked="" type="checkbox"/>		sysid_qsys	System ID Peripheral		DE1_CLKs_...	# 0x0001_0040	0x0001_004f		
<input checked="" type="checkbox"/>		led_pio	PIO (Parallel I/O)		DE1_CLKs_...	# 0x0001_0080	0x0001_008f		
<input checked="" type="checkbox"/>		dipsw_pio	PIO (Parallel I/O)		DE1_CLKs_...	# 0x0001_00c0	0x0001_00cf		
<input checked="" type="checkbox"/>		button_pio	PIO (Parallel I/O)		DE1_CLKs_...	# 0x0002_0000	0x0002_0007		
<input checked="" type="checkbox"/>		jtag_uart	JTAG UART		DE1_CLKs_...	# 0x0003_0000	0x0003_0007		
<input checked="" type="checkbox"/>		master_non_sec	JTAG to Avalon Master Bridge		DE1_CLKs_...	# 0x0003_0000	0x0003_0007		
<input checked="" type="checkbox"/>		intr_capturer_0	Interrupt Capture Module		DE1_CLKs_...	# 0x0003_0000	0x0003_0007		
<input checked="" type="checkbox"/>		clk_0	Clock Source		exported	clk_0			
<input checked="" type="checkbox"/>		DE1_CLKs	System and SDRAM Clocks for DE-seri...		exported	clk_0			
<input checked="" type="checkbox"/>		video_pil_0	Video Clocks for DE-series Boards		DE1_CLKs_...	# 0x0004_0000	0x0004_000f		
<input checked="" type="checkbox"/>		audio_and_video_config_0	Audio and Video Config		DE1_CLKs_...	# 0x0004_0000	0x0004_000f		
<input checked="" type="checkbox"/>		video_decoder_0	Video-In Decoder		DE1_CLKs_...	# 0x0004_0000	0x0004_000f		
<input checked="" type="checkbox"/>		video_bayer_resampler_0	Bayer Pattern Resampler		DE1_CLKs_...	# 0x0004_0000	0x0004_000f		
<input checked="" type="checkbox"/>		video_bridge_raw_to_vip_bridge_0	VIP Bridge: RAW to VIP		DE1_CLKs_...	# 0x0004_0000	0x0004_000f		
<input checked="" type="checkbox"/>		alt_vip_cl_ch_1	Clipper II (4K Ready)		DE1_CLKs_...	# 0x0004_0000	0x0004_000f		
<input checked="" type="checkbox"/>		video_bridge_vip_to_raw_bridge_0	VIP Bridge: VIP to RAW		DE1_CLKs_...	# 0x0004_0000	0x0004_000f		
<input checked="" type="checkbox"/>		canny_1024x768_0	canny_1024x768		exported	clk_0			
<input checked="" type="checkbox"/>		video_bridge_raw_to_vip_bridge_1	VIP Bridge: RAW to VIP		DE1_CLKs_...	# 0x0004_0000	0x0004_000f		
<input checked="" type="checkbox"/>		alt_vip_cl_vfb_3	Frame Buffer II (4K Ready)		DE1_CLKs_...	# 0x0004_0000	0x0004_000f		
<input checked="" type="checkbox"/>		alt_vip_cl_cps_0	Color Plane Sequencer II		DE1_CLKs_...	# 0x0004_0000	0x0004_000f		
<input checked="" type="checkbox"/>		alt_vip_cl_scl_0	Scaler II (4K Ready)		DE1_CLKs_...	# 0x0004_0000	0x0004_000f		
<input checked="" type="checkbox"/>		alt_vip_itc_0	Clocked Video Output		DE1_CLKs_...	# 0x0004_0000	0x0004_000f		
<input checked="" type="checkbox"/>		SDRAM	SDRAM Controller		DE1_CLKs_...	# 0x0004_0000	0x0004_000f		

Figure 4-14 The Qsys (Platform Designer) design of the system

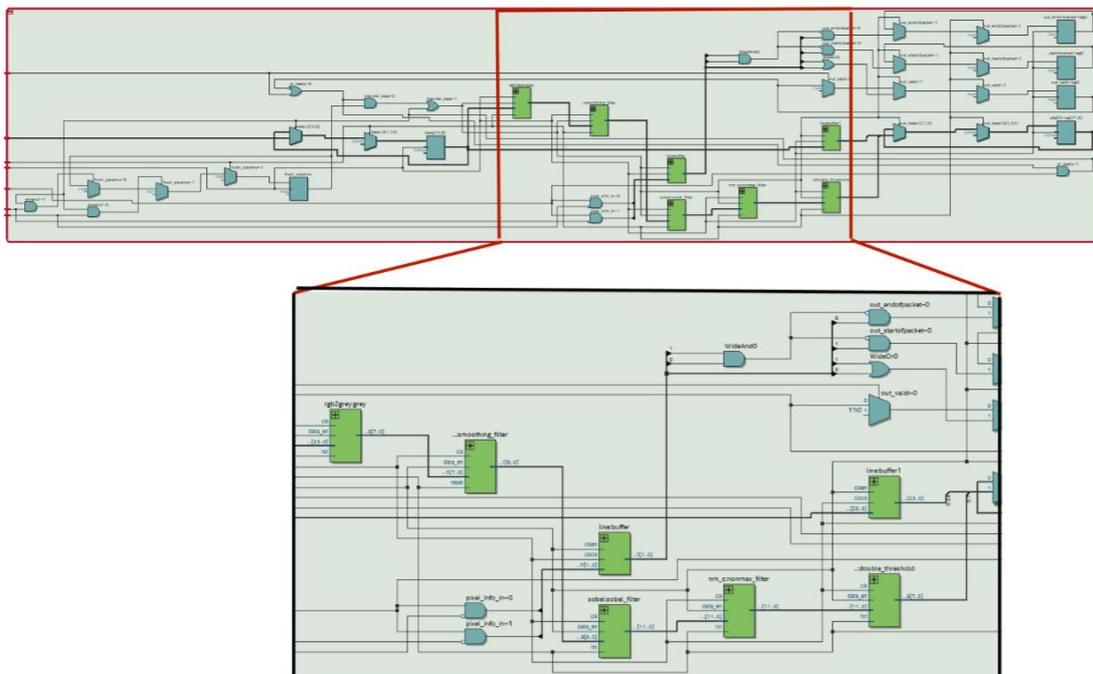
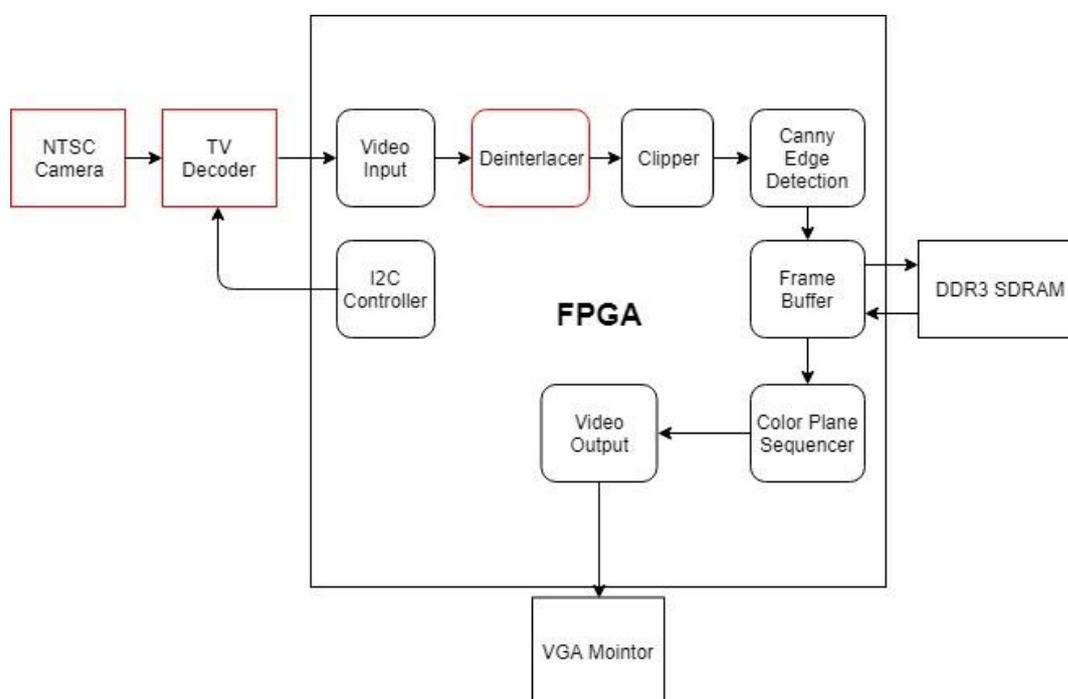


Figure 4-15 Interconnection of the accurate method of Canny Edge Detection IP

If the NTSC camera is used as the input source, a deinterlacer will be required in the system as shown by the red block in Figure 4-16. The system also using a TV decoder in the design which is not integrated in the FPGA but provided on the DE1-SOC board. This TV Decoder chip (Analog Device ADV7180) [42] needs to be configured using the I2C controller in the FPGA and it is directly

connected between the video-in port and the FPGA. The TV decoder converts the video from an analogue signal to a digital signal, it will be received by the FPGA in an interlaced format. With a few format conversions, it will be transformed into a 24-bits RGB video stream at 640x480 resolution with 30fps for performing the Canny Edge Detection. However, as its resolution is quite low and the deinterlacer requires large areas and generates a significant delay in the system, the NTSC camera was replaced by the TRDB-D5M as this allowed more control over the image resolutions.



**Figure 4-16 Architecture of the System with an NTSC Camera**

## **4.5 Results & Discussion**

As previous mentioned, the system buffering the video is the DDR3 memory which connects to the HPS in the FPGA. The HPS is able to access the video which is buffered in the memory for analysis using the ARM processor. A frame

of the original RGB data with the resolution of 1024x768 is extracted from the video stream and shown in Figure 4-17. Figure 4-18 shows the baseline result achieved using MATLAB on a PC for comparison. This enables a comparison between the hardware CED methods and “ground truth”. The edge pixels that are counted using MATLAB is 24,352.

Figure 4-19 shows the result of the accurate method obtained from the video stream. Compared with the MATLAB result, it has some edge loss (circled in red) from the ground truth result. Counted using MATLAB, the result contains 19,526 edges information which has 19.9% information loss of the edge pixels. There are probably two reasons for this problem, the first is the system is a real-time system, some of the lost edge pixels’ magnitude are just around the low threshold. Therefore, it may not be appearing in this frame but will appear in the next frame. The second reason is that, despite this method being with high accuracy, it still has not has the same precision as the MATLAB program which is using double precision floating point numbers. The approximate calculation will be less accurate.

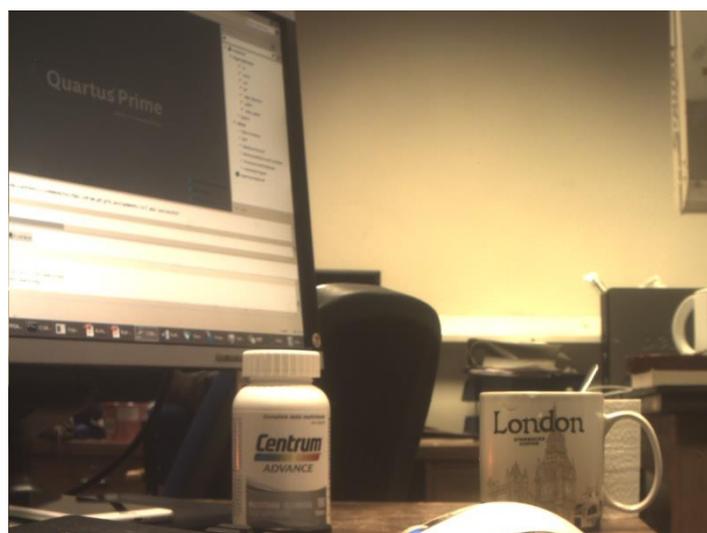


Figure 4-17 An Original RGB Frame of the video stream



Figure 4-18 Desired CED Result achieved using MATLAB



Figure 4-19 Result of the accurate method of CED

Figure 4-20 shows the result from the rapid method. As can be observed, there is greater information loss in the result (shown in the red circles). It contains 14,025 edges counted with MATLAB which is a 40.5% information loss of the edge pixels. However, most of the information lost is weak edges, it still able to recognize the objects. In general, it still performs the CED algorithm well.



Figure 4-20 Result of rapid method of CED

With this situation, the rapid method is more useful if a system is running at very high resolution or a system requires ultra-low latency. The accurate method is more suitable for a system requiring more edge information.

Table 4-5 shows a comparison of the system resources and latency with the two methods at different resolutions. The latency is measured by comparing the processed data with the original data. It shows that the rapid method's speed is faster than the accurate method by about 1,000 cycles and with 30% less use of ALMs (Adaptive Logic Modules) and 17.3% less cost in memory bits at 1024x768 resolution. However, at this resolution, the rapid method suffers about 40% information loss compared with the more accurate method's 20%. But with either method, the memory used on this FPGA is only about 6% or less. With a smaller resolution, the memory used and latency both have been reduced slightly. But the ALMs almost remain the same. Also compared with the resolution of 640x480, it can be concluded that the ALMs used in the system is proportional to accuracy. Meanwhile, the memory used in the system is

proportional to the resolution. The latency is proportional to both resolution and accuracy.

With this situation, the rapid method is more useful if a system is running at very high resolution or a system requires ultra-low latency. The accurate method is more suitable for a system requiring more edge information.

**Table 4-5 Utilization summary with each method in different resolution**

<b>Canny Edge Detection</b>						
<b>System Frequency: 100MHz</b>						
<b>Resolution</b>	1024x768		800x600		640x480	
<b>Pixels/ frame</b>	786432		480000		307200	
<b>Method</b>	Rapid	Accurate	Rapid	Accurate	Rapid	Accurate
<b>ALMs</b>	572.4	688	521.6	685	504.2	683.1
<b>Memory Used(kB)</b>	25.1 (5%)	30.3 (6%)	19.6 (4%)	23.6 (5%)	14.4 (3%)	19.8 (4%)
<b>M10K Used</b>	35	38	22	29	17	25
<b>DSP</b>	0	0	0	0	0	0
<b>Latency (clock cycles)</b>	4117	5145	3221	4025	2581	3225

## ***4.6 Summary***

In summary, this chapter presented the resource usage and performance for different resolutions and accuracy of the Canny Edge Detection algorithm implementation on the FPGA using a Hardware Description Language.

Two implementations of CED has been presented, one is focused on low-cost and low latency, and the other is on accuracy. For the rapid implementation, it has 30% less ALMs usage and 20% less in latency than the accurate implementation. However, it suffers 40.5% information compared with the accurate version's 19.9%. With this situation, it can be concluded that the ALMs used in the system is proportional to accuracy. Meanwhile, the memory used in the system is proportional to the resolution. The latency is proportional to both resolution and accuracy. With this situation, the rapid method is more useful if a system is running at very high resolution or a system requires ultra-low latency. Finally, the accurate method is more suitable for a system which requires more edge information.

## ***Chapter 5 Real time image processing on FPGAs with the HLS Approach***

### ***5.1 Introduction***

In the last chapter, a real-time image processing system was developed and evaluated on the DE1-SOC development board. A Canny Edge Detection algorithm was implemented in the system that was developed using the HDL approach. In general, an image processing IP like CED usually take several months to develop. But as Intel introduce the High-Level Synthesis tool which could be used for FPGA programming with the C/C++ Language, this whole development process could be shorten into several weeks.

This Chapter presents the implementation and results of the CED algorithm and Harris Corner Detection to compare with the HDL approach. It shows the effectiveness and existing issues of using the HLS approach.

Section 5.2 gives an overview of the HLS Compiler. Section 5.3 gives the introduction to the Harris Corner Detection algorithm. Section 5.4 describes the implementation of both algorithms. Section 5.5 presents the results comparison between the HDL approach and HLS approach. In the last section, a summary of this chapter is presented.

### ***5.2 HLS Compiler***

The Intel HLS Compiler is a high-level synthesis (HLS) tool that takes in untimed C++ as input and generates production-quality RTL that is optimized for Intel FPGAs [55]. With this feature, this tool could accelerate the design and verification time over RTL for FPGA hardware design. According to Intel,

applications developed in C++ are typically faster than HDL and requires 80% fewer lines of code. The Intel HLS Compiler generates reusable, high-quality code that meets performance requirements and is within 10%-15% of the area of hand-coded RTL [55].

As mentioned previously, the most common way to build complex system designs in Intel Platform Designer is to use IP cores. The HLS Compiler provides the ability to create IP cores directly from HLS projects. Once the core has been synthesized, the HLS Compiler will generate an IP core packet which could be imported by Platform Designer directly. Figure 5-1 shows the whole design process of HLS design.

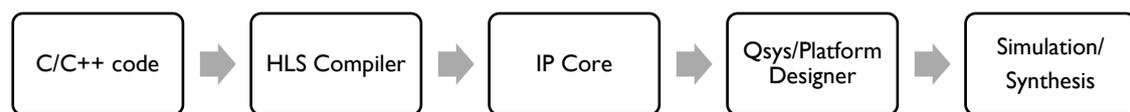


Figure 5-1 HLS design process

This requires the HLS code to include directives that indicate what type of bus interfaces should be present and which variables should be made available on them.

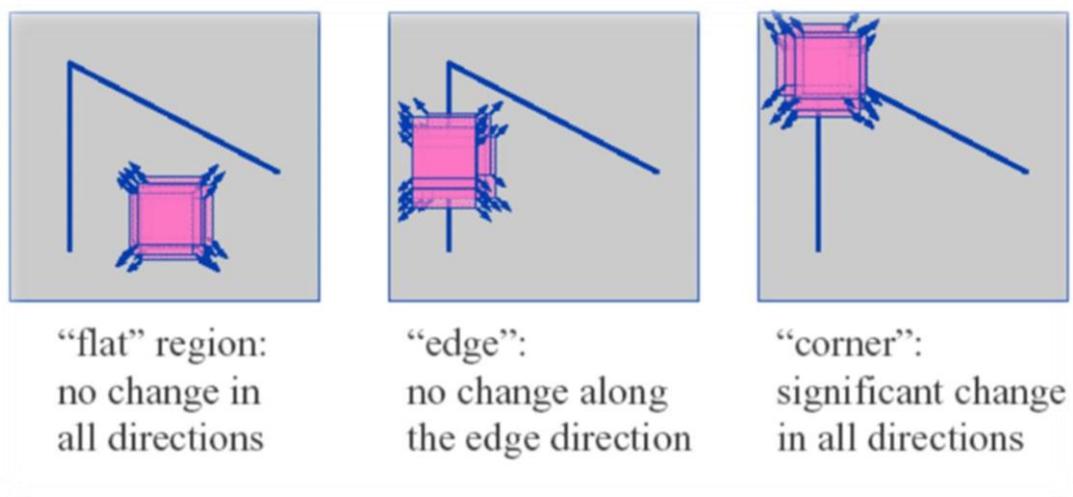
### 5.3 Harris Corner Detection

Apart from the Canny Edge Detector introduced in the last chapter, another algorithm called the Harris Corner Detector is used to evaluate the capability of the HLS approach.

The Harris Corner Detector was proposed by C. Harris and M. J. Stephens is an algorithm targeted at feature extraction [56]. Inspired by the autocorrelation

function in signal processing, this algorithm established the autocorrelation matrix of the pixel in the image whose eigenvalue could be denoted as the first-order curvature of the autocorrelation function. A point which has high curvature values in both the X and Y directions would be considered as a corner. The principle of the Harris detector is derived from people’s perceptual judgment of the diagonal point; that is, the image has a significant change on the grayscale in all directions.

Through calculating the changing value of grayscale in the moving window, the difference among the flat region, edge and corner could be identified. As shown on Figure 5-2, when the window moves in a flat region, the grayscale value would not have obvious change; when it moves at the edge, the value does not change much along the edge direction but change a lot along the direction perpendicular to the edge; when the window moves in the corner, the grayscale value changes a lot in any direction. The corner is what is needed to be identified.



**Figure 5-2 The basic principle of the Harris Corner Detection**

The corner could be defined as a junction of two edges where the brightness changes [57]. It is a significant concept in computer vision as it could reflect

the most important information within an image even though it only takes a small percentage, given its features of invariant through translation, rotation or illumination. Therefore, it is commonly used in computer vision to compress the amount of data for post processing.

Based upon the classic Moravec's corner detector, the Harris Corner Detector made an improvement in distinguishing edges and corners more accurately [58]. On the one hand, it replaces the binary window function with a Gaussian function, which gives pixels closer to the centre point a greater weight to reduce the effects of noise. On the other hand, the Harris detector approximated any direction of the pixel's movement through Taylor expansion, rather than only considering every 45 degrees in the Moravec detector.

To calculate the changing values in the grayscale, let's assume a grayscale 2-dimensional image given by  $I$  and a local window  $(x, y)$  shifted  $(\Delta x, \Delta y)$ . Therefore, an autocorrelation function denoted by  $f$  could be given as (5.1):

$$f(x, y) = \sum_{(x_k, y_k) \in W} (I(x_k, y_k) - I(x_k + \Delta x, y_k + \Delta y))^2 \quad (5.1)$$

Where  $f(x, y)$  presents the sum of the squares of the difference between the original window and moves towards the  $(\Delta x, \Delta y)$  direction. The larger value of  $f(x, y)$  means the larger possibilities that the window  $(x, y)$  locates at the corner or the edge. Next, the  $I(x_k + \Delta x, y_k + \Delta y)$  could be approximated by a Taylor expansion (5.2):

$$I(x_k + \Delta x, y_k + \Delta y) \approx I(x, y) + I_x(x, y)\Delta x + I_y(x, y)\Delta y \quad (5.2)$$

To simplify it (5.3),

$$f(x, y) = \sum_{(x,y) \in W} (I_x(x, y)\Delta x + I_y(x, y)\Delta y)^2 \quad (5.3)$$

Then it could be written in matrix form(5.4)

$$f(x, y) \approx (\Delta x \quad \Delta y)M \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} \quad (5.4)$$

Where  $M$  represents the structure tensor as given in (5.5):

$$\begin{aligned} M &= \sum_{(x,y) \in W} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \\ &= \begin{bmatrix} \sum_{(x,y) \in W} I_x^2 & \sum_{(x,y) \in W} I_x I_y \\ \sum_{(x,y) \in W} I_x I_y & \sum_{(x,y) \in W} I_y^2 \end{bmatrix} \end{aligned} \quad (5.5)$$

Decomposing  $M$  into a combination of eigenvalues and eigenvectors (5.6):

$$M = ABA \quad (5.6)$$

Where  $A$  is composed of feature vectors;  $B$  is a 2\*2 diagonal matrix whose diagonal is the characteristic value  $\lambda_1 \lambda_2$ . It is known that the displacement vector is multiplied by  $A$  to get a direction vector, when this direction vector is multiplied by  $B$ , the following results could occur:

It would be identified as flat region when  $\lambda_1 \lambda_2$  are both very small;

It would be identified as edge when  $\lambda_1 \gg \lambda_2$  or  $\lambda_1 \ll \lambda_2$ ;

It would be identified as corner when  $\lambda_1 \lambda_2$  are similar and are both very large;

Then a response function could be given to determine the corner (5.7):

$$\lambda_{min} \approx \frac{\lambda_1 \lambda_2}{(\lambda_1 + \lambda_2)} = \frac{det(M)}{trace(M)} \quad (5.7)$$

This formula shows that if the small eigenvalues  $\lambda_1 \lambda_2$  are large, then both the two eigenvalues  $\lambda_1 \lambda_2$  are large, so the window could be identified as a corner.

Therefore, Harris used a bit of heuristic thinking to define  $R$  as (5.8):

$$R = det(M) - k(trace(M))^2 = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2 \quad (5.8)$$

Where  $K$  is an empirically constants and usually valued between 0.04 and 0.06. Then, the corner could be identified through judging the value of  $R$ .

Figure 5-3 and Figure 5-4 below shows the result of a chessboard image after filtering with the Harris Corner Detector.

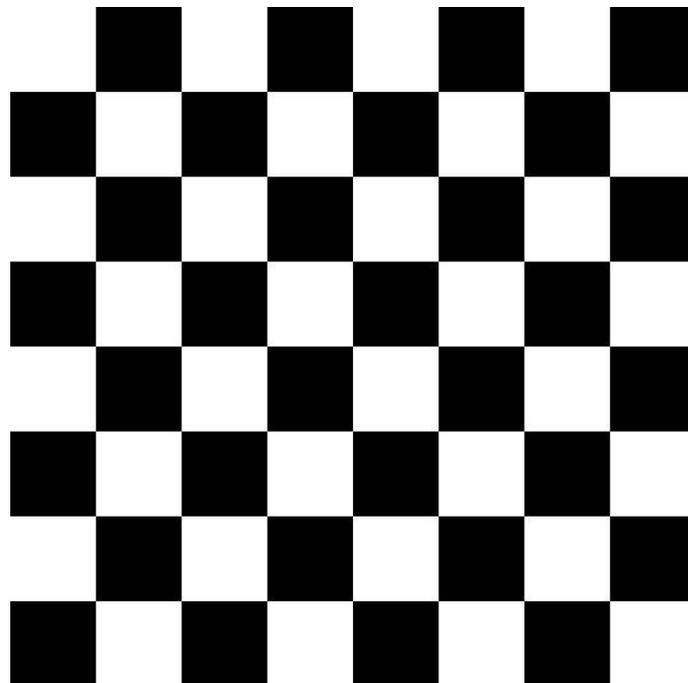


Figure 5-3 Black and White Chessboard image

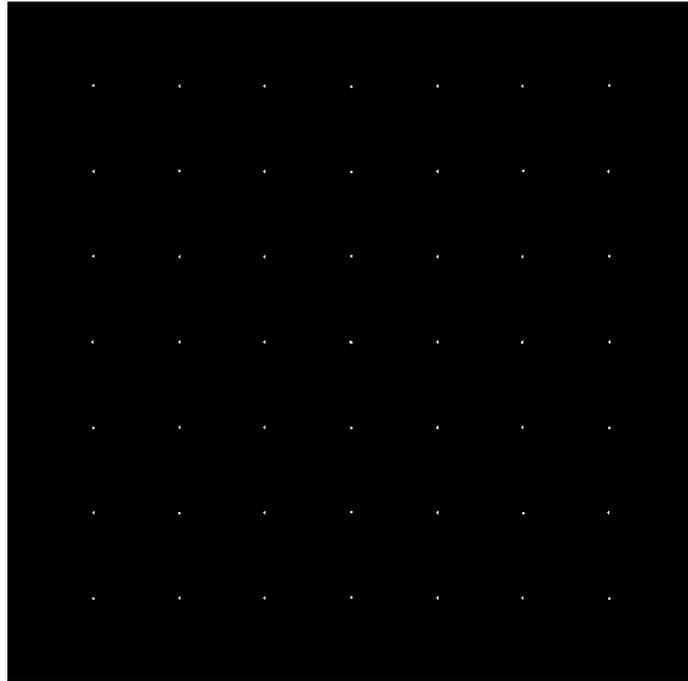


Figure 5-4 Chessboard image Filtered with the Harris Corner Detector

In summary, the full process of a Harris Corner Detection method is shown in Figure 5-5.



Figure 5-5 Stages of a Harris Corner Detection process

## ***5.4 Algorithm Implementation***

The aim of this section is to develop a real time image processing IP on an FPGA using the HLS approach. The Canny Edge Detector and Harris Corner Detector are implemented by the HLS Compiler coded in C++, however there are still a few differences from a general C++ program.

Usually, a normal C++ programme will handle the existing data or values passed by pointers. However, a real-time image processing IP core usually runs continuously. As shown in Chapter 4, a CED algorithm is required to store several lines for convolutional processing. Using the HLS Compiler, it is required to use the “for” loop to store the incoming data. Simultaneously, to get several data for each kernel computation, it is necessary to use the #pragma unroll method to “unroll” the loop. Figure 5-6 shows an ‘N’ depth shift register coded with the HLS Compiler.

```
1. #pragma unroll
2. for (int i = N - 1; i > 0; --i) {
3.     buffer[i]=buffer[i-1];
4. }
5. buffer[0] = data_in;
```

**Figure 5-6 Example code of a shift register with HLS Compiler**

Meanwhile, as a FPGA design, it is required to handle signals which is not very common in C/C++ programming. At the moment, the HLS Compiler only provides support for both Avalon-ST and Avalon MM Master interfaces. In this project, it is required to handle packets including the Avalon-ST interface. Thus, the application is required to be defined as shown in Figure 5-7 where unsigned char means the width of the interface is 8 bits.

```
1. void exmaple(ihc::stream_in<unsigned char, ihc::usesPackets<true> >& data_in, ihc::stream_out<unsigned char, ihc::usesPackets<true> >& data_out )
```

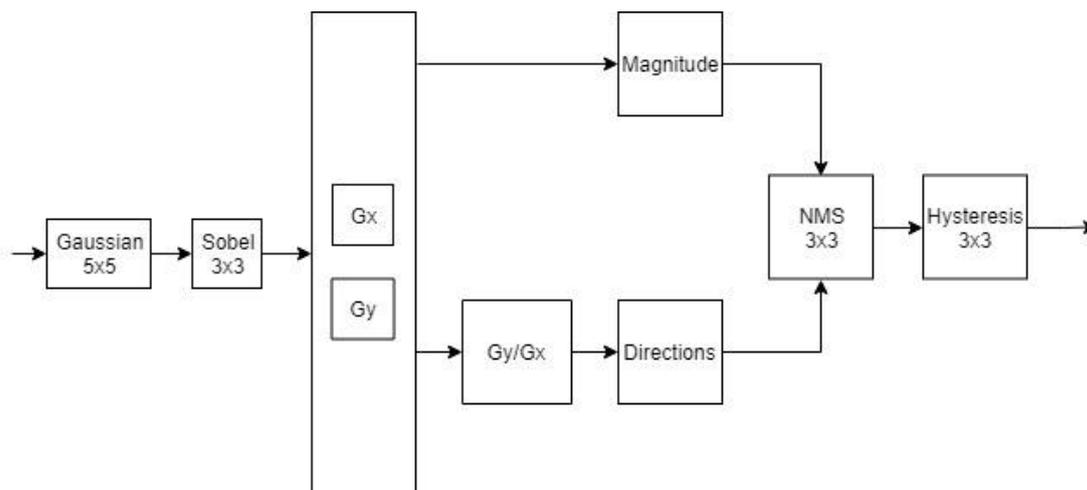
**Figure 5-7 Example code of application definition with the Avalon-ST interface**

In the meantime, as the packet signals include startofpacket and endofpacket. The startofpacket and endofpacket signals can be processed as shown in Figure 5-8:

```
1. while (!end_of_packet) {
2.     // read in data
3.     data = data_in.read(start_of_packet, end_of_packet);
4.
5.     //...
6.     // write out data
7.     data_out.write(data, start_of_packet, end_of_packet);
8. }
```

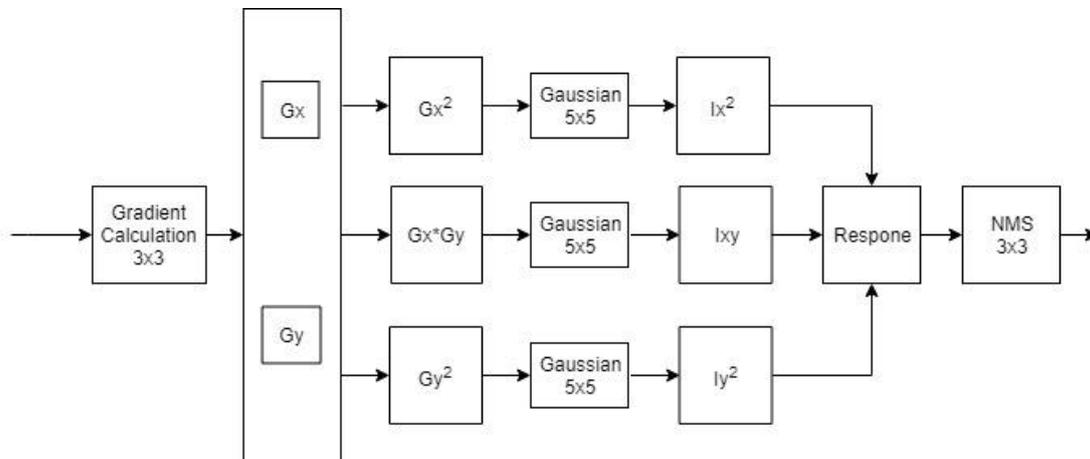
**Figure 5-8 Example code of handling packets signals with HSL Compiler**

With the preceding information, the Canny Edge Detector algorithm is implemented with the same accuracy as the HDL accuracy method in the last chapter which is shown as Figure 5-9.



**Figure 5-9 Canny Edge Detection algorithm implementation with HLS approach**

Then the Harris Corner Detector algorithm is implemented as shown in Figure 5-10.



**Figure 5-10 Harris Corner Detection algorithm implementation using the HLS approach**

## **5.5 Results and Discussion**

### **5.5.1 Canny Edge Detector**

The original design with the HDL approach was developed with two stages. The first stage includes writing the high-level MATLAB model, evaluates the balance of the resource usage, algorithm accuracy and system latency and writes Verilog code. The first stage takes about 6 months for the Canny Edge Detector algorithm as it takes significant effort to get familiar with the HDL approach for algorithm implementation.

The second stage takes about 2 months for testing the implementation and evaluating the results. Therefore, it takes 8 months in total to implement the Canny Edge Detector algorithm using the HDL approach.

For the HLS approach, it takes about 4 weeks to convert the original high-level model into HLS Compiler friendly code as the first stage. However, the high-level MATLAB model takes about 2 weeks to write. Therefore, the total time for the implementation of Canny Edge Detector algorithms takes about 1.5 months.

And for the second stage, the testing takes longer than expected. Although the HLS Compiler offers the reports for analysing the design, it still did not detail enough to show the resource usage and latency for a complex algorithm. As the HLS Compiler generates the code into a design that contains both the Verilog and VHDL languages, it is not possible to test with the ModelSim tool for behavioural testing. Meanwhile, by becoming familiar with HLS Compiler, it is able to reduce the latency and resources usage of the design by avoiding floating point calculations and most of the divisions. In total, the second stage takes almost 2.5 months using the HLS approach.

Table 5-1 shows a comparison of the resources used and latency between the two approaches at different resolutions. It shows that the HLS approach is slower than the HDL approach with about twice the latency and requiring 84% more memory bits at 1024x768 resolution. For the ALMs, it required about 5 times more than the HDL method.

Like the HDL approach, the system buffers the video in the DDR3 memory which is on the HPS side. It can then access the video, which is buffered in the memory for result analysis, using the ARM processor. A frame of original RGB data is extracted from the video stream is shown in Figure 5-11. Figure 5-12 shows the result achieved using the HLS approach from the video stream. Figure 5-13 shows the result of the HDL approach from the last chapter for comparison. However, there are no significant differences between the two results.

Table 5-1 Comparison between the HLS and the HDL approach with Canny Edge Detector

<b>Canny Edge Detection</b>				
<b>System Frequency: 100MHz</b>				
<b>Resolution</b>	1024x768		800x600	
<b>Method</b>	HDL	HLS	HDL	HLS
<b>ALMs</b>	688	2659.9	685	2651.3
<b>Memory Used(KB)</b>	30.3 (6%)	51.3 (10%)	23.6 (5%)	42.3 (9%)
<b>M10K Used</b>	38	59	27	48
<b>DSP</b>	0	0	0	0
<b>Latency (clock cycles)</b>	5145	9406	4025	7023

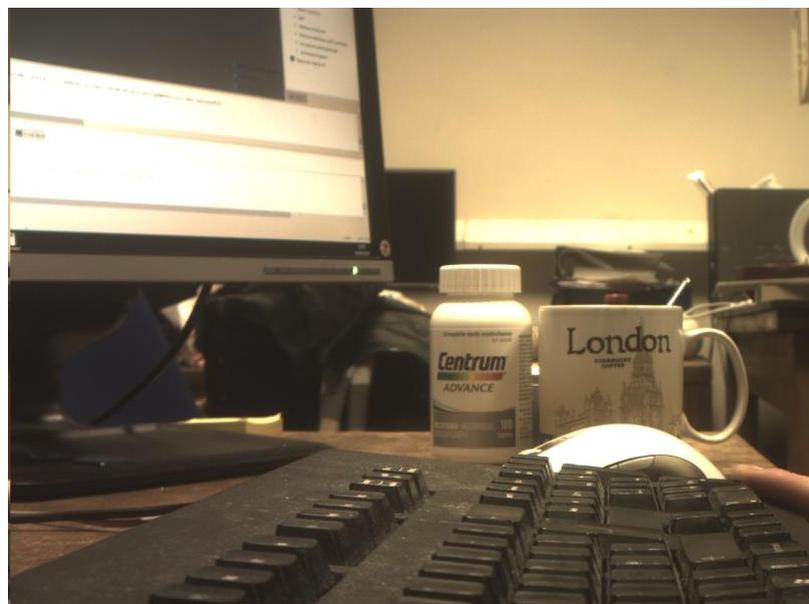


Figure 5-11 Original RGB data



Figure 5-12 Canny Edge detector with HLS approach



Figure 5-13 Canny Edge Detector with HDL approach

### ***5.5.2 Harris Corner Detector***

The first stage of Harris Corner Detector algorithm implementation with HDL approach takes about 1 month in designing a high-level MATLAB model. Then it takes another month to write the Verilog code. It takes 2 months in total for the first stage which is much faster than the Canny Edge Detector implementation, as the experience has been achieved during the previous development.

The second half takes 2 months again for testing the implementation and evaluating the result. Therefore, it takes 4 months in total to implement the Harris Corner Detector algorithm using the HDL approach.

For the HLS approach, it takes about 3 weeks to convert the original high-level model into HLS Compiler friendly code as the first stage. As the high-level MATLAB model takes about 1 month to write, the total time for the implementation of Harris Corner Detector algorithms is about 1.5 months. For the second stage, the testing takes 2 months for the Harris Corner Detector.

Table 5-2 shows a comparison of the resources used and latency between the two approaches at different resolutions. It shows that the HLS approach is much slower than the HDL approach with almost triple the latency and requiring 186% more memory bits at 1024x768 resolution. For the ALMs, it required about 2.5 times more than the HDL method.

It should be mentioned that although the HLS approach has much larger resource requirements and latency than the HDL method, its major benefit is the significantly reduced design time. Even for a relative novice HLS designer, this IP module could be finished in a few weeks rather than the HDL approach which may take several months. Moreover, with more familiarity with the HLS

Compiler and FPGA design, the design time and system cost could be further reduced.

Table 5-2 Comparison between the HLS and the HDL approach with Harris Corner Detector

<b>Harris Corner Detection</b>				
<b>System Frequency: 100MHz</b>				
<b>Resolution</b>	1024x768		800x600	
<b>Method</b>	HDL	HLS	HDL	HLS
<b>ALMs</b>	2796.8	6914.5	2868.3	6910.3
<b>Memory Used (KB)</b>	67.1 (12%)	192.3 (34%)	52.4 (5%)	180.2 (30%)
<b>M10K Used</b>	66	167	65	159
<b>DSP</b>	12	15	12	15
<b>Latency (clock cycles)</b>	4123	12067	3227	8542

As a recently publicised tool for Intel FPGAs, the HLS tool still does not provide enough support on programming. This includes less support on existing libraries, limited documentation on the HLS Compiler, less design examples and a few bugs. Hopefully this will be solved with future releases of the HLS system with further optimization of HLS Compiler. In summary the HLS approach is close to replacing the traditional HDL programming approach.

## ***5.6 Summary***

This chapter makes a detailed comparison of performance of two approaches on system design, which are the HDL and the HLS, through presenting the implementation of Canny Edge Detection algorithm and Harris Corner Detection algorithm. Being different from the former algorithms, Harris Corner Detection is used for point feature extraction. It largely shrinks the amount of post processing data as it only focuses on the corners in the image, which could contain significant information. Then, two approaches are implemented on two algorithms respectively to test the performance of both the HDL and HSL approaches.

Consequently, the HSL approaches generally shows a much shorter development period than HDL, with 8 months for HDL and 2.5 months for HSL, in total, needed to implement the Canny Edge Detector, and 4 months for HDL and 2 months for HSL, in total, needed to implement the Harris Corner Detector. However, the HSL approach also has higher requirement on a device's memory and shows higher latency than the HDL. According to the results, the HSL has twice the latency to the HDL, followed by 84% more memory bits required and 5 times more ALMs than the latter when implementing the Canny Edge Detector. As far as the Harris Corner Detector is concerned, the HSL even has triple the latency to the HDL, followed by 186% more memory bits required and 1.5 times more the ALMs than the HDL. Thus, HSL and HDL shows different advantages in FPGA programming. As the HDL approach is suitable for a system requiring more accuracy or less latency and the HLS approach is suitable for a system requiring less development time.

## ***Chapter 6 A Co-processing FPGASoC system***

### ***6.1 Introduction***

In the previous two chapters, the HDL approach and the HLS approach have been researched by implementing a Canny Edge Detector and a Harris Corner Detector as pre-processing algorithms. In this Chapter, the FPGASoC system which contains both the FPGA and HPS system are used to explore the potential for pre and post co-processing. A customized OpenCV programme is used as the post processing algorithm in the system and accelerated by the FPGA using the pre-processing algorithms. Meanwhile, with the objective of implementing post processing algorithms, it is necessary to build the software environment for OpenCV on the FPGASoC system.

In this chapter Section 6.2 presents an overview of the FPGASoC designs. Section 6.3 provides an introduction to OpenCV [59]. Section 6.4 describes the architecture of the system. Section 6.5 presents the results of the system and discusses them. In the last section, a summary of this chapter is presented.

### ***6.2 The FPGASoC System Design***

Compared with just FPGA designs, FPGASoC designs also include the HPS system. The HPS, which uses an ARM processor, can implement software programs in the system to perform actions including controlling the behaviour of the IP cores on the FPGA side, data processing or accessing devices on the HPS. The HPS can operate in two modes: bare-metal or OS mode. The bare-metal mode can run with the ARM DS-5 tool which provides the ability for

debugging of the code. However, if the application involves device API or other 3<sup>rd</sup> part libraries, it is required to run on a configured operation system for example embedded Linux. The embedded Linux system integration includes several parts; Preloader, Device Tree, U-Boot, kernel and a Root Filesystem, and also any hardware FPGA design and custom software applications, which make up an FPGASoC design as shown in Figure 6-1. The Preloader and Device Tree are built using Quartus II and the U-Boot, Kernel and Root Filesystem Linux kernel are built using the Yocto Project [60]. Then an IDE for example ARM DS-5 is used to load and debug the custom applications. The configured system can be configured on an SD card.

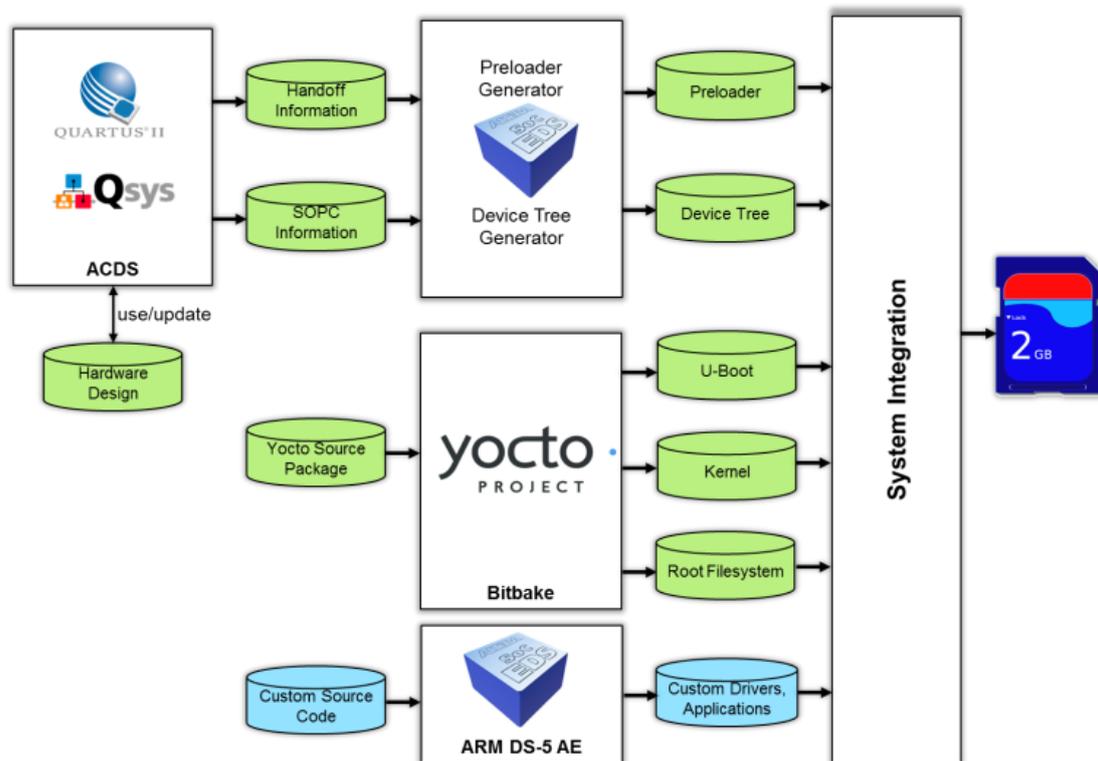


Figure 6-1 a high-level view of the development flow [61]

When booting from an SD card, there are several stages to initialise the system as shown in Figure 6-2. Table 6-1 presents a short description of the different boot stages:

Table 6-1 Descriptions of the different boot stages [61]

Stage	Description
BootROM	Performs minimal configuration and loads the Preloader into 64KB on chip RAM
Preloader	Configures clocking, IOCSR, pinmuxing, SDRAM and loads U-boot into SDRAM
U-boot	Configures FPGA, loads the Linux kernel
Linux	Runs the end application

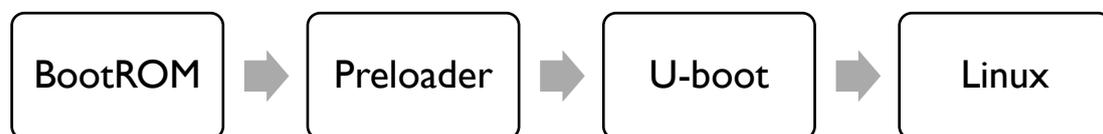


Figure 6-2 the system boot flow

The Preloader configures the HPS component based on the information from the handoff folder generated by Quartus, it initializes the SDRAM and then loads the next stage of the boot process into the SDRAM and passes control to it.

Figure 6-3 shows the flow for generating a preloader image.

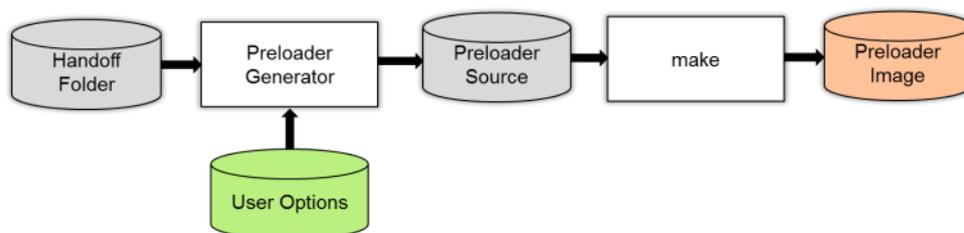
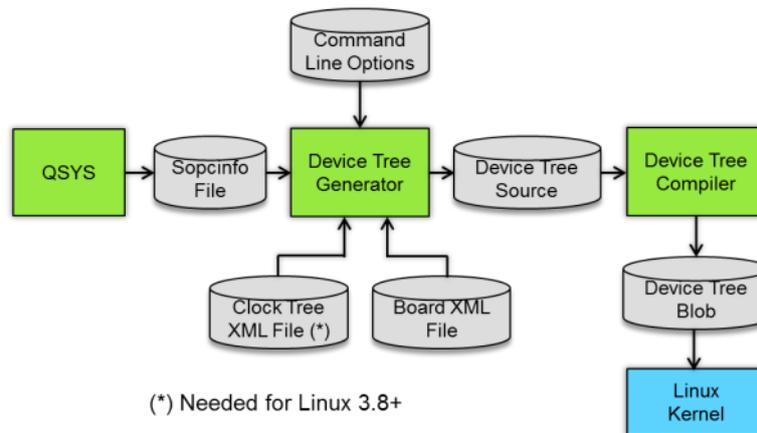


Figure 6-3 Generation of the preloader [55]

A Device Tree is a data structure that describes the underlying hardware to the operating system – primarily Linux. By passing this data structure to the OS kernel, a single OS binary may be able to support many variations of hardware. This flexibility is particularly important when the hardware includes an FPGA.

The Device Tree Generator tool is part of Intel SoC EDS and is used to create device trees for SoC systems that contain FPGA designs created using System Builder. The generated Device Tree describes the HPS peripherals, selected FPGA Soft IP and peripherals that are board-dependent. Figure 6-4 shows the processes for device tree generation.



**Figure 6-4 The Device Tree generation flow [61]**

The final part is the Linux kernel execution. When the Linux kernel boots up, it starts off by performing low-level architecture specific initialization sequences (setting up the processors registers, memory management unit, interrupt controller, etc.). It also loads up a serial driver to output debug messages to show the information in the boot flow via a serial terminal.

After that, it starts initializing all the kernel subsystems and drivers that were compiled into the kernel. Lastly, it attempts to mount the Root Filesystem which contains the shell custom programs.

It is not necessary to rebuild U-Boot, the Linux kernel and the Root Filesystem for all changes to the FPGA section, it is only necessary when a driver needs to be updated or inserted or when an update is required to the Linux Kernel.

Once all the stages have been completed, it is necessary to programme or “flash” the image onto an SD card. Figure 6-5 shows the layout of the SD card required for correctly booting the system:

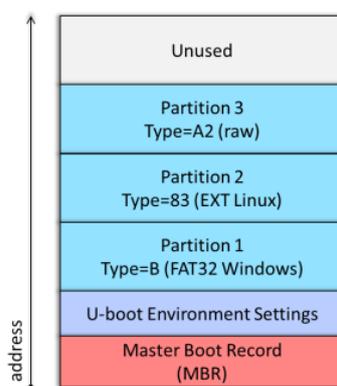


Figure 6-5 SD card layout [61]

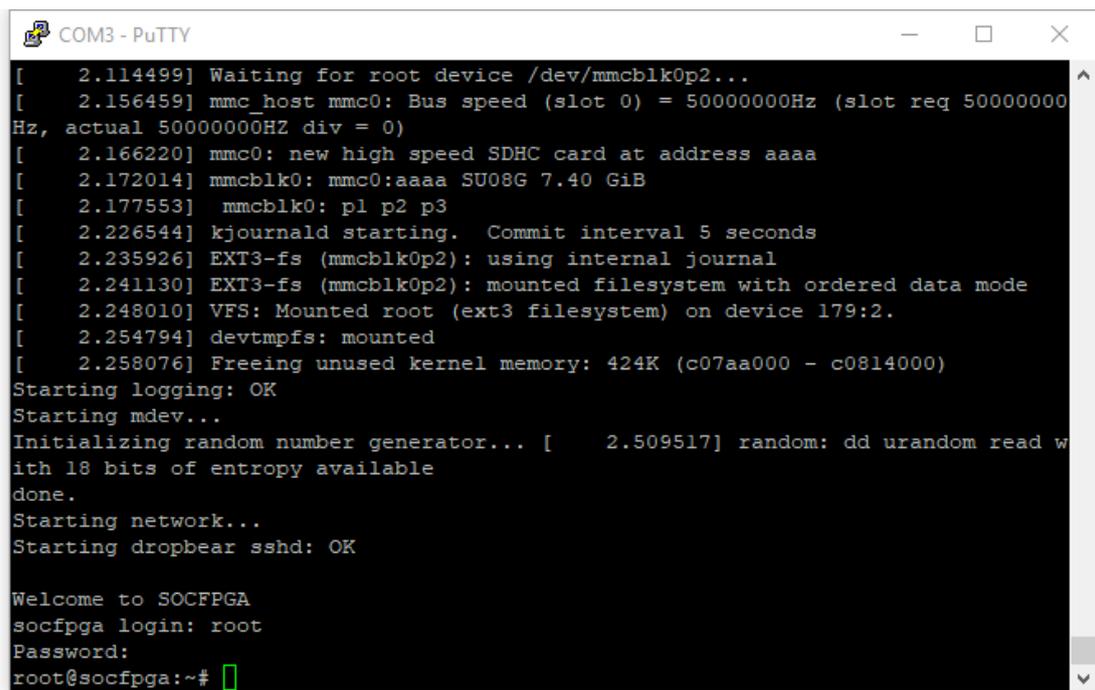
Table 6-2 summarizes the information that is stored on the SD card and its location:

Table 6-2 information stored on the SD card [61]

Location	File	Description
<b>Partition 3</b>	n/a	Preloader image
<b>Partition 3</b>	n/a	U-boot image
<b>Partition 1</b>	soc_system.rbf	FPGA configuration file
<b>Partition 1</b>	socfpga.dtb	Device Tree Blob file
<b>Partition 1</b>	u-boot.scr	U-boot script for configuring FPGA
<b>Partition 2</b>	various	Linux root filesystem
<b>Partition 1</b>	zImage	Compressed Linux kernel image file

When the SD card is built/updated and installed on the board, as the processor powers up, the HPS would load Linux into SDRAM automatically. With a PC

running serial console terminal software (like Putty), it is possible to control the boot process using the host computers USB port, via a virtual serial port to the board. Figure 6-6 shows the data from the virtual serial port on a host PC, in this case a Windows PC. If the user application has been loaded on the SD card, it could be loaded and executed using the serial terminal, alternatively it could be added to the boot script.

A screenshot of a PuTTY terminal window titled "COM3 - PuTTY". The window displays a series of Linux boot logs. The logs include: "[ 2.114499] Waiting for root device /dev/mmcblk0p2...", "[ 2.156459] mmc\_host mmc0: Bus speed (slot 0) = 500000000Hz (slot req 500000000Hz, actual 500000000HZ div = 0)", "[ 2.166220] mmc0: new high speed SDHC card at address aaaa", "[ 2.172014] mmcblk0: mmc0:aaaa SU08G 7.40 GiB", "[ 2.177553] mmcblk0: p1 p2 p3", "[ 2.226544] kjournald starting. Commit interval 5 seconds", "[ 2.235926] EXT3-fs (mmcblk0p2): using internal journal", "[ 2.241130] EXT3-fs (mmcblk0p2): mounted filesystem with ordered data mode", "[ 2.248010] VFS: Mounted root (ext3 filesystem) on device 179:2.", "[ 2.254794] devtmpfs: mounted", "[ 2.258076] Freeing unused kernel memory: 424K (c07aa000 - c0814000)", "Starting logging: OK", "Starting mdev...", "Initializing random number generator... [ 2.509517] random: dd urandom read with 18 bits of entropy available done.", "Starting network...", "Starting dropbear sshd: OK", "Welcome to SOCFPGA", "socfpga login: root", "Password:", "root@socfpga:~#". The terminal shows a green cursor at the end of the last line.

```
[ 2.114499] Waiting for root device /dev/mmcblk0p2...
[ 2.156459] mmc_host mmc0: Bus speed (slot 0) = 500000000Hz (slot req 500000000
Hz, actual 500000000HZ div = 0)
[ 2.166220] mmc0: new high speed SDHC card at address aaaa
[ 2.172014] mmcblk0: mmc0:aaaa SU08G 7.40 GiB
[ 2.177553] mmcblk0: p1 p2 p3
[ 2.226544] kjournald starting. Commit interval 5 seconds
[ 2.235926] EXT3-fs (mmcblk0p2): using internal journal
[ 2.241130] EXT3-fs (mmcblk0p2): mounted filesystem with ordered data mode
[ 2.248010] VFS: Mounted root (ext3 filesystem) on device 179:2.
[ 2.254794] devtmpfs: mounted
[ 2.258076] Freeing unused kernel memory: 424K (c07aa000 - c0814000)
Starting logging: OK
Starting mdev...
Initializing random number generator... [ 2.509517] random: dd urandom read w
ith 18 bits of entropy available
done.
Starting network...
Starting dropbear sshd: OK

Welcome to SOCFPGA
socfpga login: root
Password:
root@socfpga:~#
```

Figure 6-6 Software Screenshot of embedded Linux command line

### 6.3 Compile the Linux kernel

As previous explained, if a new driver needs to be installed or updated, it is necessary to rebuild the Linux kernel. There are several steps in building the Linux kernel as shown in the Figure 6-7.

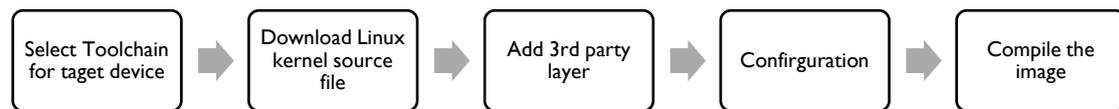


Figure 6-7 Steps for building the Linux kernel

Before starting to build the Linux kernel, some preparation work is needed. It is best to use a Linux based computer for the kernel compilation, in this research a Ubuntu 14.04 OS was used as the host machine, there are a few libraries and packages that need to be installed which include: *sed*, *wget*, *cvs*, *subversion*, *git-core*, *coreutils*, *unzip*, *texi2html*, *texinfo*, *libsdl1.2-dev*, *docbook-utils*, *gawk*, *python-pysqlite2*, *diffstat*, *help2man*, *make gcc*, *build-essential*, *g++*, *desktop-file-utils*, *chrpath*, *libgl1-mesa-dev*, *libglu1-mesa-dev*, *mercurial*, *autoconf*, *automake*, *groff*, *libtool* and *xterm* [61].

Then, with the necessary programmes and libraries installed, the toolchain for the target device, in this case a Cyclone V FPGA-SoC, needs to be selected. The toolchain is a set of programming tools that are used to perform the software development task or to create a software product, which is usually another computer program or a set of related programs. Typically, a development toolchain includes a compiler and linker (which transform the source code into an executable program), libraries (which provide interfaces to the operating system), and a debugger (which is used to test and debug the programs created). In this research, the toolchain that are provided with the Intel EDS tool were used for building the kernel.

The next step is downloading the source file of Linux kernel. The Linux kernel source is provided by Linux Kernel Organization. Companies like Intel adds some libraries into the source file to make it suitable for their own devices like the Cyclone V SoC.

The last step, before building the kernel image, is to configure the kernel. The kernel source file usually provides a graphic menu as shown in Figure 6-8. With this menu, it is straight forward to select the drivers needed in the system. For example, if a webcam like a C270 is going to be used in the system, then several drivers need to be added in the kernel image including: Media USB Adapters driver, USB Video Class driver and UVC input events device support.

When the build is finished, the SD card is updated with the new kernel image. Also, U-Boot and the device tree may need to be updated for the installed driver.

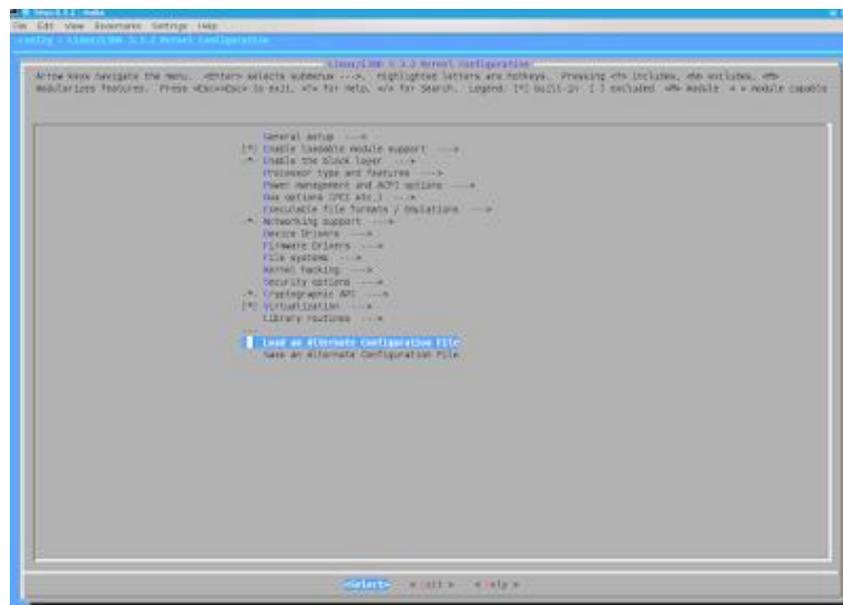


Figure 6-8 Configuration menu of Linux Kernel

## 6.4 OpenCV

With the embedded Linux running on HPS system, it is productive if the high-level image processing application can make use of existing image processing

libraries. OpenCV, whilst originally target for Intel CPUs, is one of the most widely used image processing libraries and was used in this research.

OpenCV is a cross-platform programming function library. It implements a number of computer vision algorithms, ranging from the basic filtering to advanced object detection. Based on the BSD license and programmed through C++ and C, OpenCV plays a major role of standardising the APIs for real-time computer vision as it provides interface bindings for languages like Python, Ruby, MATLAB and Java and can be used on various operation systems including Linux, Windows, Android and Mac OS.

Because of the abundant algorithms and functions which can be found in OpenCV and as it is open-source, OpenCV is widely used to make machine portable algorithms. It also accelerates the speed of system development.

OpenCV was initially established by Intel and is now maintained by Itseez [59]. It usually has a significant update each year and the latest version published on the 23rd December 2017 is OpenCV 3.4. Its application areas are very wide, including the Human–computer interaction (HCI), motion tracking, augmented reality, 2D and 3D feature toolkits, Structure from Motion (SFM) and recognition of objects, face, and actions.

In this research, the OpenCV library is required to be compiled before it is installed into the system. It requires the same version of toolchain which is used for building for the Linux Kernel. The user application is also required to be built using the same toolchain.

## 6.5 System Implementation

Apart from the HPS system, it still needs an FPGA design to capture the video stream into the HPS memory and read out the result to display on a monitor. The system is similar to the architecture described in the previous chapters but replaces the frame buffer with a separate frame writer and a separate frame reader. Then both IPs are configured to use the fpgas2sdram bus to access the DDR3 memory directly. In parallel, the ARM processor on the HPS can access the DDR3 memory to read the image data for processing. In addition, the frame writer and frame reader are required to be controlled by the ARM processor to control the write and read actions to adapt to the processing time of the HPS system. The redundant frames are dropped by the frame writer and missing frames are filled by repeating the previous frame by the frame reader. At the same time, the system has been integrated with the Canny Edge and Harris Corner Detection algorithms. As the fpgas2sdram only supports several modes i.e. 16-bit, 32-bit, 64-bit, 128-bit and 256-bit transfer mode, the 64-bit mode was used in this design to keep the result of Canny Edge and Harris Corner Detection algorithms. Both pre-processed data streams have been merged into 64-bits word and combined with the grayscale result and the original RGB data of each pixel. As shown in Figure 6-9, for each 64-bit pixel, it contains the 24-bit original RGB data, 16-bit Harris Corner intensity result, 8-bit Canny Edge intensity result, 8-bit Grayscale result and 8-bit reserved data. The whole architecture of the design is shown as Figure 6-10.

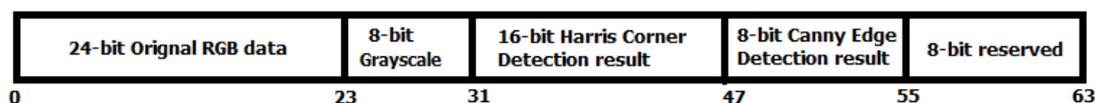


Figure 6-9 64-bit word structure of the system

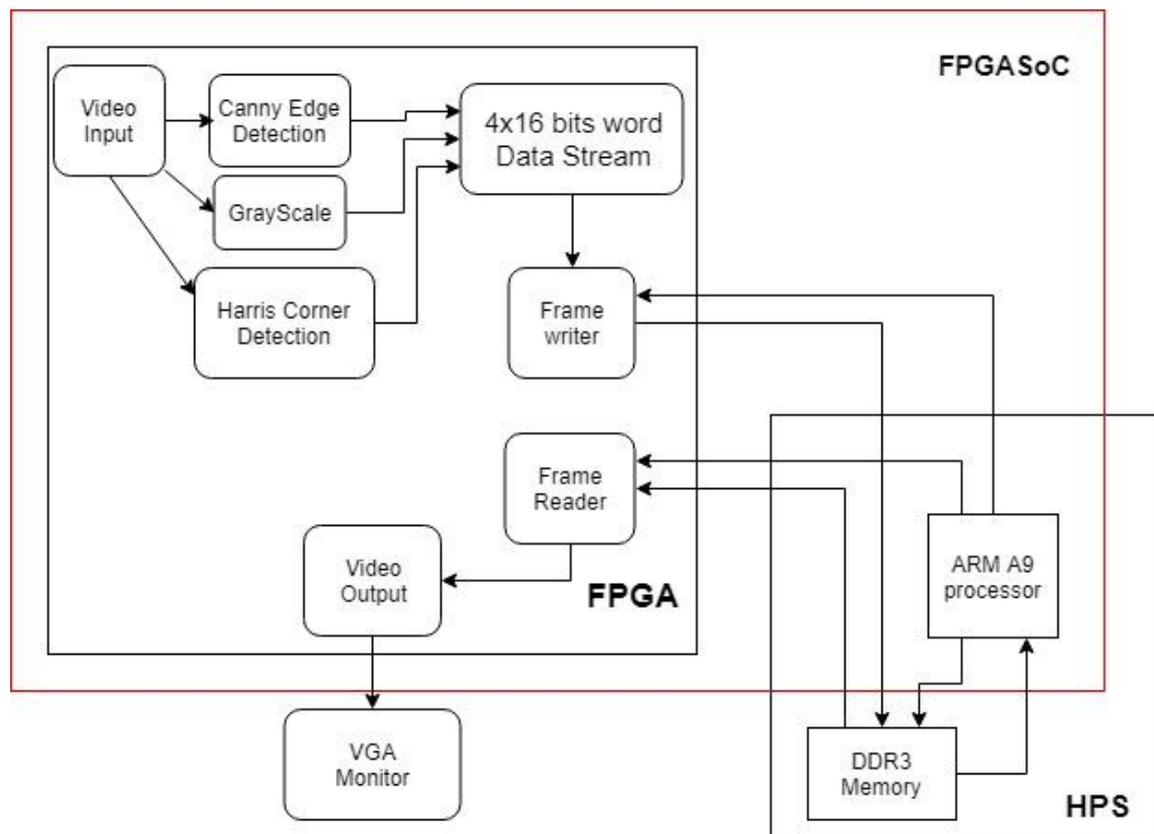


Figure 6-10 Architecture of the System

## 6.6 Results and Discussion

With the aim of demonstrating the performance improvement of the system, a CED algorithm is applied to the original 24-bit RGB data using an OpenCV application with an image resolution of 1024x768. The result compared with the HDL method are shown in Figure 6-11 and Figure 6-12. The performance of the OpenCV implementation is a little bit better as the algorithm is using the square root to calculate magnitude instead of using the absolute value. This increases the precision of the system but will also increase the processing time. The processing time of the OpenCV Canny algorithm takes about 86ms to process a frame. Therefore, the frame rate is already down to 11 fps with only an implementation of CED and no further high-level image processing. Whilst,

with the HDL method, it takes only adds a latency of 5,145 clock cycles i.e. a 0.05ms delay at 100 MHz without reducing the frame rate. As a result, it can reduce by 86ms the processing time for a CED based algorithm like image segmentation or moving object tracking.

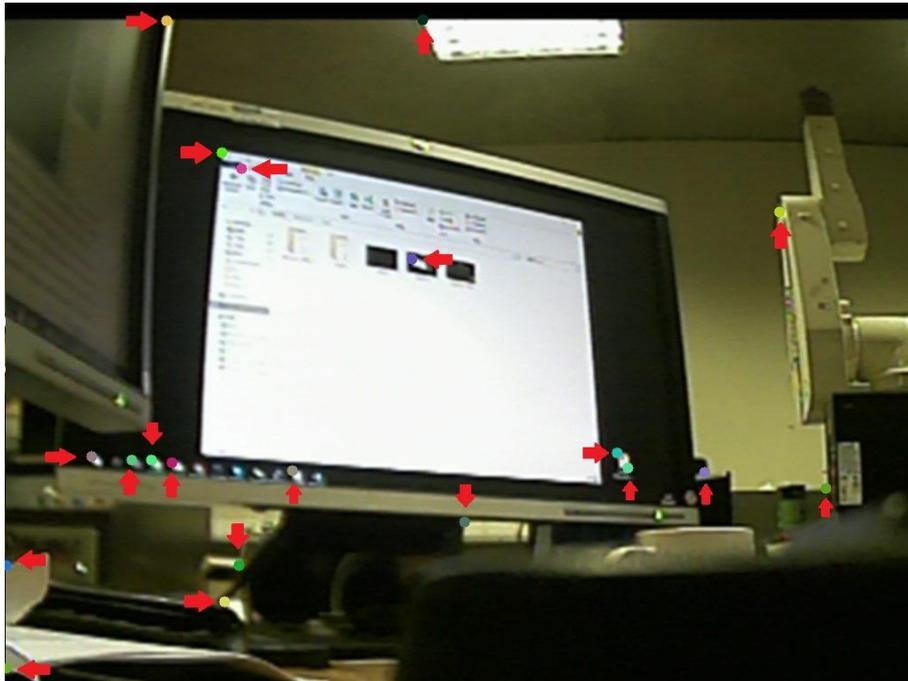


Figure 6-11 Canny Edge Detection Result of OpenCV



Figure 6-12 Canny Edge Detection Result of HDL method

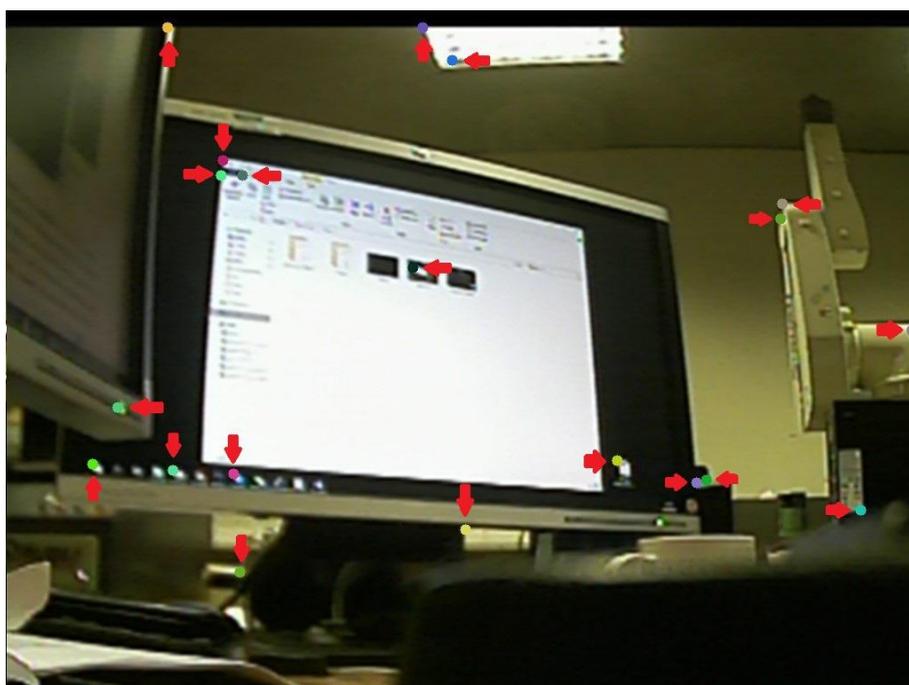
Subsequently, another OpenCV application has been applied to test the system: the feature tracking algorithm with 20 points (shown in Figure 6-14) requires 0.3691s to process a frame at the resolution of 1024x768. The feature points are marked with red arrows in the figure.



**Figure 6-13 A frame result of feature tracking application using OpenCV (Feature points are marked in with red arrows)**

Meanwhile, as the feature tracking application is based on the Harris Corner Detection algorithm which would also be implemented on the FPGA. Performing the Harris Corner Detection algorithm in hardware would significantly increase the system performance. With the objective of combining the FPGA design with the OpenCV application, the original application was modified to receive the data from the FPGA and extract the information from the memory at the same time. However, as the data extracted from the memory, pixel by pixel, is a 64-bit combined word as previous mentioned, it is required to extract the different results separately to complete the whole data extraction process for post-

processing. Then the result of feature tracking application using OpenCV accelerated by the FPGA is shown as Figure 6-14 below. The feature points are marked with red arrows. Compared with the original OpenCV result, it has some different feature points. This is because some of the feature points have a similar Harris Corner result as they may show up dynamically in the result. In the meantime, the strongest feature points remain stable.



**Figure 6-14 A frame result of feature tracking application using OpenCV accelerated by FPGA (Feature points are marked in with red arrow)**

Table 6-3 shows a comparison of the total processing time of the two methods at different resolutions. With the acceleration of FPGA, the total processing time has been reduced by 48.2%, 49.5% and 56.1% at the resolutions of 640x480, 800x600 and 1024x768 respectively. It can extrapolate that with the higher resolution, the results become more effective. However, it requires more time for data extraction as it is proportional to the resolution. In this case, the FPGA has improved the performance of the application sufficiently.

Table 6-3 Feature Tracking Algorithm Comparison

Feature Tracking Algorithm (OpenCV)		
Resolution	Total Processing Time/Frame Rate	
	ARM only	Accelerated with FPGA (Data Extraction Time)
640x480	0.1268s/7.89fps	0.0656s (0.0216s)/15.24fps
800x600	0.2165s/4.62fps	0.1092s (0.0352s)/9.16fps
1024x768	0.3691s/2.71fps	0.1624s (0.0559s)/6.16fps

However, the capability of the integrated ARM processor is not powerful enough for true real-time performance. The system is still running at a low frame rate even with the acceleration of the FPGA. With next generation of the devices, the performance of the system may be improved.

## 6.7 Summary

To summarize, this chapter presented the FPGASoC design that combines both the FPGA and HPS systems. The FPGA part has integrated both the Canny Edge and Harris Corner Detection algorithms. The HPS part is based on a customized Linux with the OpenCV framework. With the comparison of only the Canny Edge Detection algorithm, the FPGA dominated the performance. With the feature tracking algorithm, the FPGA accelerated the total processing time by reducing it to 48.2%, 49.5% and 56.1% at the resolution of 640x480, 800x600 and 1024x768. Thus, with the acceleration of FPGA on pre-processing algorithms, the performance of a high-level algorithm can be improved.

## ***Chapter 7 Conclusions and Future work***

### ***7.1 Conclusions***

Image processing applications typically require some pre-processing algorithms on the raw image data followed by post-processing algorithms on both the pre-processed results, and the original data, to extract useful information from the image.

The pre-processing algorithm is generally applied to the full image and can be time consuming unless it is undertaken in dedicated hardware such as a GPU or FPGA. The cost and power requirements of GPU based systems make them unsuitable for low cost embedded applications.

This research investigated the use of a low cost FPGASoC device for real time image processing by developing a real-time image processing system with several approaches for the pre-processing algorithms, using the FPGA, to reduce the processing time. Additionally, it synchronizes the original data in parallel with the pre-processed data in memory for further processing, i.e. the pre-processed image is stored as a 64-bit word with 8 bits each for the RGB values and 32-bit for the pre-processing results. Simultaneously, it provides the infra-structure for implementing complex image processing applications on the integrated ARM system with support from the OpenCV library. The FPGA design was developed in Quartus II using the Video Image Processing (VIP) IP which provides several sub-systems such as frame buffer, clocked video in & out in Platform Designer (formally Qsys), which is Intel's (formally Altera) tool for developing SOPC systems. Therefore, the programmable hardware design needed to develop the algorithm to be compatible with Intel's VIP based IP format so that it could be compatible with the Intel VIP subsystems.

Firstly, the research shows the two implementations of Canny Edge Detection algorithm with the HDL approach, one is focused on low-cost and low latency, and the other is on higher accuracy. For the rapid implementation, it uses 30% less ALMs and the latency is 20% less than the accurate implementation. However, it suffers from 40.5% information loss compared with the accurate version's 19.9%. With these results, it shows that as expected the ALMs used in the system is proportional to accuracy. Meanwhile, the memory used in the system is proportional to the resolution. Whilst the latency is proportional to both resolution and accuracy.

Secondly, the HLS approach is researched by making a detailed comparison of the performance of two approaches on system design, which are the HDL and the HLS, through presenting the implementation of the Canny Edge Detection algorithm and the Harris Corner Detection algorithm. Then, the two approaches were implemented on two algorithms respectively to test the performance of both the HDL and HSL approaches. Consequently, the HSL approaches generally requires a much shorter development period than HDL, with 8 months for HDL and 2.5 months for HSL in total needed to implement the Canny Edge Detector and 4 months for HDL and 2 months for HSL in total needed to implement the Harris Corner Detector. However, the HSL approach also has higher requirement on a device's memory and a higher latency than the HDL implementation. According to the results, the HSL has twice the latency of the HDL implementation, followed by 84% more memory bits required and 5 times more ALMs when implementing the Canny Edge Detector. For the Harris Corner Detector, the HSL has triple the latency compared to the HDL implementation, followed by 186% more memory bits required and 1.5 times more ALMs than the HDL implementation. Thus, HSL and HDL shows different advantages in FPGA programming as HDL approach shows more accuracy and HSL approach has more efficiency.

The next, an IP is developed with HDL method which contains the original RGB data, Harris Corner Detection result, Canny Edge Detection result and Grayscale result all synchronized together pixel by pixel. Concurrently, this design is based on the customized OpenCV application for post-processing implementations. Later with the feature tracking algorithm, the FPGA accelerates the total processing time by reducing it to 48.2%, 49.5% and 56.1% at the resolution of 640x480, 800x600 and 1024x768 differently. With the higher resolution, the result become more effective. Thus, with the acceleration of FPGA on pre-processing algorithms, the performance of a high-level algorithm can be improved.

To summarise the novelty in this research is the development of an embedded FPGASoC image processing architecture where image pre-processing takes place in real-time in the FPGA fabric allowing the ARM SoC processor to concentrate on the post processing algorithm thus reducing the time between the image is captured and the result presented. Such an approach will open up the market for low-cost real-time image processing applications as the system capital and running costs are significantly lower than using a PC based system. The comparison between the HDL and HLS approaches allows recommendation on which to select when developing an embedded image processing system. As the HDL approach is suitable for a system requiring more accuracy or less latency and the HLS approach is suitable for a system requires less development time.

## **7.2 Future work**

During the research on this project, there have been a number of ideas for future work.

### **7.2.1 Algorithms & System implementation**

The implementation of the algorithm in this system could be improved by several methods, the first is implementation of another edge tracking method which may improve the result of broken edges. However, it may cost more time in processing the video stream. Another idea is to replace the Sobel detector with other detectors like Scharr, as it may have better result in some situations. Also, an Eight-directional Canny could be implemented to improve the results. In the meantime, more work would then be needed on customized OpenCV applications for testing on the system.

Then, it should be possible to implement the other more lower level algorithms in parallel in the FPGA, and a multiplexor can be used under software control, to select which results gets written into the HPS system. This would provide dynamic flexibility when selecting the lower level algorithm to use whose results are then used by the high-level image processing algorithm.

### **7.2.2 Devices & Other approach**

In this research, the whole system is developed on the Cyclone V SoC device. But during the development, it appears that the integrated ARM processor has not the performance for complex algorithms. In the meantime, the FPGA on chip is quite enough for pre-processing algorithms. It is necessary for further development with another FPGASoC device which contains a more powerful

ARM processor on chip. With a FPGASoC device which lays emphasis on ARM processor instead of FPGA, it may be more suitable for embedded vision system.

Additionally, OpenCL based on the FPGASoC could be evaluated. The system will be mainly based on HPS and utilize FPGA to optimize the computation. It will provide another approach for low-cost real-time image processing designs.

### ***References***

- [1] C. T. Johnston, K. T. Gribbon and D. G. Bailey, "Implementing image processing algorithms on FPGAs," in *In Proceedings of the Eleventh Electronics New Zealand Conference*, New Zealand, November 2004, ENZCon'04, pp. 118-123..
- [2] Intel Corporation, Intel Corporation, [Online]. Available: <http://www.altera.com>.
- [3] J. Kang and R. Doraiswami, "Real-time image processing system for endoscopic applications," in *Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canadian Conference*, May 2003.
- [4] C. Balfour, J. S. Smith and S. Amin-Nejad, "Feature correlation for weld imageprocessing applications," *International Journal of Production Research*, pp. Volume 42, pp 975-995, March 2004.
- [5] S. A. Clukey, "Architecture for Real-Time, Low-SWaP Embedded Vision Using FPGAs," Master's Thesis, University of Tennessee, 2016.
- [6] S. Palnitkar, *Verilog HDL: a guide to digital design and synthesis*, Prentice Hall Professional, 2003.
- [7] P. J. Ashenden, *The designer's guide to VHDL. Vol. 3*. Morgan Kaufmann, Morgan Kaufmann, 2010.
- [8] S. Asano, T. Maruyama and Y. Yamaguchi, "Performance comparison of FPGA, GPU and CPU in image processing," in *2009 International Conference on Field*, 2009.

## References

---

- [9] E. Fykse, "Performance Comparison of GPU , DSP and FPGA implementations of image processing and computer vision algorithms in embedded systems," Norwegian University of Science and Technology, 2013.
- [10] Z. K. Baker, M. B. Gokhale and J. L. Tripp, "Matched Filter Computation on FPGA, Cell and GPU," *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, p. pp. 207–218, 2007.
- [11] J. Fowers, G. Brown, P. Cooke and G. Stitt, "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, 2012.
- [12] P. R. Possa, S. A. Mahmoudi, N. Harb, C. Valderrama and P. Manneback, "A Multi-Resolution FPGA-Based Architecture for A Multi-Resolution FPGA-Based Architecture for," *IEEE TRANSACTIONS ON COMPUTERS VOL. 63, no. 10* , pp. pp.2376-2388, 2014.
- [13] Intel Corporation, "SOPC Builder User Guide ,," December 2010.
- [14] Xilinx, Inc., [Online]. Available: [www.xilinx.com](http://www.xilinx.com).
- [15] J. Rose, "Hard vs. soft: the central question of pre-fabricated silicon," in *In Multiple-Valued Logic, 2004. Proceedings. 34th International Symposium on*, 19-22 May 2004.
- [16] P. Mead, "Systems on Programmable Chips -Will SOPC Eclipse SoC? ,," in *Designing Systems on Silicon IEE Cambridge Seminar*, December 2001.

## References

---

- [17] H. T. Ngo, R. W. Ives, R. N. Rakvic and R. P. Broussard, "Real-time video surveillance on an embedded, programmable platform," *Microprocessors and Microsystems*, Vols. 37(6-7), pp. pp.562-571, 2013.
- [18] F. Wu, J. S. Smith, A. J. Tickle and Q. Huang, "System of a programmable-chip-based image-processing system," *Journal of Electronic Imaging*, 22(2), 023026, 2013.
- [19] Intel Corporation, "Avalon Interface Specifications," 2017.
- [20] M. B. Sandler, L. Hayat, L. Costa and A. Naqvi, "A comparative evaluation of DSPs, microprocessors and the transputer for image processing," in *Acoustics, Speech, and Signal Processing 1989 International Conference*, 23-26 May 1989.
- [21] E. Jamro and K. Wiatr, "Implementation of convolution operation on general purpose processors," in *Euromicro Conference, Proceedings 27th*, 2001.
- [22] S. Guennouni, A. Ahaitouf and A. Mansouri, "Multiple object detection using OpenCV on an embedded platform," in *Information Science and Technology (CIST)*, 2014.
- [23] A. Varfolomieiev and O. Lysenko, "An improved algorithm of median flow for visual object tracking and its implementation on ARM platform.," *Journal of Real-Time Image Processing*, vol. 3, no. 11, pp. pp.527-534., 2016.
- [24] R. Matthew and S. Fischaber, "OpenCV based road sign recognition on Zynq," in *Industrial Informatics (INDIN), 2013 11th IEEE International Conference In*

## References

---

- Industrial Informatics (INDIN), 2013 11th IEEE International Conference*, July, 2013.
- [25] A. Rosenfeld, "Computer vision: basic principles," *Proceedings of the IEEE*, pp. pp.863-868., 1988.
- [26] C. Steger, M. Ulrich and C. Wiedemann, *Machine Vision Algorithms and Applications (2nd ed.)*, Weinheim: Wiley-VCH, 2018.
- [27] S. P. CHAMBERS, "TIPS: a transputer based real-time vision system," Liverpool University, 1990.
- [28] S. Trimberger, "Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology," *Proceedings of the IEEE, 103(3)*, pp. pp.318-331., 2015.
- [29] Intel Corporation, "FPGA Architecture," Intel Corporation, 2006.
- [30] National Instruments, "FPGA Fundamentals," 03 May 2012. [Online]. Available: <http://www.ni.com/white-paper/6983/en/>.
- [31] XILINX INC., "Zynq-7000 All Programmable SoC Data Sheet," 2017.
- [32] Intel Croporation, "Cyclone V Device Overview," 2016.
- [33] Microsemi Corporation, "SmartFusion Customizable System-on-Chip," 2015.
- [34] Intel Croporation, "Nios II Classic Processor Reference Guide," 2016.
- [35] XILINX INC., "MicroBlaze Processor Reference Guide," 2009.

## References

---

- [36] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design, the hardware/software interface*, Morgan Kaufmann Publishers, 1998.
- [37] ARM Limited, "Cortex-A9 Technical Reference Manual," ARM Limited, 2010.
- [38] ARM Limited, "AMBA AXI and ACE Protocol Specification," ARM Limited, 2017.
- [39] Intel Corporation., "Video and Image Processing Suite User Guide," Intel Corporation., 2017.
- [40] Intel Croporation, "Cyclone V Hard Processor System Technical Reference Manual," Intel Croporation, 2018.
- [41] ring0, "FPGA design flow overview," Universal Tech Media Corporation, 2009. [Online]. Available: <http://www.fpgacentral.com/docs/fpga-tutorial/fpga-design-flow-overview>.
- [42] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal and M. Luján, "An empirical evaluation of High-Level Synthesis languages and tools for database acceleration," in *24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014.
- [43] M. Livingstone, *Vision and Art: The Biology of Seeing*, New York: Harry N. Abrams, 2002.
- [44] Intel Croporation, "Quartus II Handbook," Intel Croporation, 2017.
- [45] International Telecommunication Union , "BT.656 : Interface for digital component video signals in 525-line and 625-line television systems operating at the 4:2:2

## References

---

- level of Recommendation ITU-R BT.601," International Telecommunication Union , 12 2007. [Online]. Available: <http://www.itu.int/rec/R-REC-BT.656/en>.
- [46] International Telecommunication Union, "BT.1120 : Digital interfaces for studio signals with 1 920 × 1 080 image formats," International Telecommunication Union, 12 2017. [Online]. Available: <https://www.itu.int/rec/R-REC-BT.1120-8-201201-I/en>.
- [47] Intel Corporation, "DSP Builder for Intel FPGAs," Intel Corporation, 2017.
- [48] Intel Corporation., "Intel SoC FPGA Embedded Development Suite User Guide," Intel Corporation., 2017.
- [49] C. authors, "Cygwin User's Guide," 2017.
- [50] Intel Corporation, "Timing Analyzer Quick-Start Tutorial," Intel Corporation, 2017.
- [51] Terasic Technology Inc., "DE1-SOC User Manual," Terasic Technology Inc., 2016.
- [52] Terasic Technologies Inc., "TRDB\_D5M UserGuide," Terasic Technologies Inc., 2014.
- [53] Logitech., "HD Webcam C270 SPECIFICATIONS," Logitech., [Online]. Available: [http://support.logitech.com/en\\_us/product/hd-webcam-c270/specs](http://support.logitech.com/en_us/product/hd-webcam-c270/specs).
- [54] J. Canny, "A computation approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. vol. 8, no. no 6, pp. pp. 769-798, November 1986.

## References

---

- [55] Intel Corporation, "Product Brief Intel HLS Compiler," 2017.
- [56] C. Harris and M. Stephens, "A Combined Corner and Edge Detector," in *Alvey vision conference*, 1988.
- [57] K. G. Derpanis, "The Harris Corner Detector.," York University, 2004.
- [58] N. Dey, P. Nandi, N. Barman, D. Das and S. Chakraborty, "A Comparative Study between Moravec and Harris Corner Detection of Noisy Images Using Adaptive Wavelet Thresholding Technique," *International Journal of Engineering Research and Applications*, vol. Vol. 2, no. Issue 1, pp. pp.599-606, 2012.
- [59] OpenCV team, "About OpenCV," OpenCV team, [Online]. Available: <https://opencv.org/about.html>.
- [60] Yocto Project, "Yocto Project," [Online]. Available: <https://www.yoctoproject.org/>.
- [61] RocketBoards.org, "Embedded Linux Beginners Guide," 2017. [Online]. Available: <https://rocketboards.org/foswiki/Documentation/EmbeddedLinuxBeginnerSGuide>.

## ***Appendix A Codes of Canny Edge Detection with HDL approach***

### **RGB to Gray Scale transformation module**

```
1. module rgb2grey
2. (   input      clk,
3.     input      rst,
4.
5.     input      data_en,
6.     input      [23:0] in_data,
7.
8.
9.     output [7:0] out_data
10.
11.     );
12.
13.
14. wire [7:0] R;
15. wire [7:0] G;
16. wire [7:0] B;
17.
18.
19. reg [15:0] R_r;
20. reg [15:0] G_r;
21. reg [15:0] B_r;
22.
23. assign B = in_data[7:0];
24. assign G = in_data[15:8];
25. assign R = in_data[23:16];
26.
27. // calculate results
28. reg [31:0] grey;
29. reg [7:0] grey_result;
30. always @(posedge clk)
31. begin
32.     if (rst == 1'b1)
33.         begin
34.
35.             R_r    <= 16'b0;
```

```

36.          G_r    <=  16'b0;
37.          B_r    <=  16'b0;
38.
39.          grey    <=  32'b0;
40.          grey_result <=  8'b0;
41.      end
42.      else if (data_en)
43.      begin
44.
45.          R_r    <=  {2'b0,R,6'b0}  +{5'b0,R,3'b0}  +{7'b0,R,1'b0};
46.
47.          G_r    <=  {1'b0,G,7'b0}  +{4'b0,G,4'b0}  +{6'b0,G,2'b0}  +{7
          'b0,G,1'b0};
48.          B_r    <=  {3'b0,B,5'b0}  +{8'b0,B}      -{6'b0,B,2'b0};
49.          grey    <=  R_r +  G_r +  B_r;
50.
51.          grey_result <=  grey[15:8];
52.      end
53. end
54. assign out_data=grey_result;
55. endmodule

```

### Gaussian filter Module

```

1. module gaussian_filter (
2.     clk,
3.     reset,
4.
5.     data_in,
6.     data_en,
7.
8.     // Outputs
9.     data_out
10. );
11.
12.
13. parameter WIDTH = 1024; // Image width in pixels
14.
15.
16. // Inputs

```

```
17. input          clk;
18. input          reset;
19.
20. input          [ 7: 0] data_in;
21. input          data_en;
22.
23.
24. // Outputs
25. output         [ 8: 0] data_out;
26.
27.
28. // Internal Wires
29. wire           [ 7: 0] iline2;
30. wire           [ 7: 0] iline3;
31. wire           [ 7: 0] iline4;
32. wire           [ 7: 0] iline5;
33.
34. // Internal Registers
35. reg            [ 7: 0] oline_1[ 4: 0];
36. reg            [ 7: 0] oline_2[ 4: 0];
37. reg            [ 7: 0] oline_3[ 4: 0];
38. reg            [ 7: 0] oline_4[ 4: 0];
39. reg            [ 7: 0] oline_5[ 4: 0];
40.
41.
42. reg            [15: 0] level_1[6: 0];
43.
44. reg            [15: 0] level_2[ 4: 0];
45. reg            [15: 0] level_3;
46. reg            [15: 0] level_4;
47.
48.
49. reg            [8:0] final_result;
50.
51. // Integers
52. integer        i;
53.
54.
55. // Gaussian Smoothing Filter
56. //
57. //           [ 2  4  5  4  2 ]
58. //           [ 4  9 12  9  4 ]
59. // 1 / 159   [ 5 12 15 12  5 ]
```

```
60. //          [ 4 9 12 9 4 ]
61. //          [ 2 4 5 4 2 ]
62. //
63. // mask X
64.
65. always @(posedge clk)
66. begin
67.     if (reset == 1'b1)
68.     begin
69.         for (i = 4; i >= 0; i = i-1)
70.         begin
71.             oline_1[i] <= 8'h00;
72.             oline_2[i] <= 8'h00;
73.             oline_3[i] <= 8'h00;
74.             oline_4[i] <= 8'h00;
75.             oline_5[i] <= 8'h00;
76.             level_1[i] <= 12'h000;
77.         end
78.
79.
80.     end
81.     else if (data_en )
82.     begin
83.         for (i = 4; i > 0; i = i-1)
84.         begin
85.             oline_1[i] <= oline_1[i-1];
86.             oline_2[i] <= oline_2[i-1];
87.             oline_3[i] <= oline_3[i-1];
88.             oline_4[i] <= oline_4[i-1];
89.             oline_5[i] <= oline_5[i-1];
90.         end
91.         oline_1[0] <= data_in;
92.         oline_2[0] <= iline2;
93.         oline_3[0] <= iline3;
94.         oline_4[0] <= iline4;
95.         oline_5[0] <= iline5;
96.
97.         level_1[0] <= {7'b0,oline_1[0], 1'b0} + {7'b0,oline_1[4], 1'b
0} + {7'b0,oline_5[0], 1'b0} + {7'b0,oline_5[4], 1'b0};
98.         //times 4
99.         level_1[1] <= {6'b0,oline_1[1], 2'b0} + {6'b0,oline_1[3], 2'b
0} + {6'b0,oline_2[0], 2'b0} + {6'b0,oline_2[4], 2'b0};
```

```

100.         level_1[2] <= {6'b0,oline_4[0], 2'b0} + {6'b0,oline_4[4], 2'
           b0} + {6'b0,oline_5[1], 2'b0} + {6'b0,oline_5[3], 2'b0};
101.
102.         //5
103.         level_1[3] <= {8'b0,oline_1[2]} + {8'b0,oline_5[2]}+ {8'b0,
           oline_3[0]} + {8'b0,oline_3[4]};
104.         //9
105.         level_1[4] <= {8'b0,oline_2[1]} + {8'b0,oline_2[3]}+ {8'b0,
           oline_4[1]} + {8'b0,oline_4[3]};
106.         //12
107.         level_1[5] <= {8'b0,oline_2[2]} + {8'b0,oline_4[2]} + {8'b0,
           oline_3[1]} + {8'b0,oline_3[3]};
108.         //15
109.         level_1[6] <= {4'b0,oline_3[2], 4'b0} - oline_3[2];
110.
111.         level_2[0] <= level_1[0]+ level_1[6];
112.
113.         level_2[1] <= level_1[1]+ level_1[2];
114.
115.         // Multiplied by 5
116.         level_2[2] <= {level_1[3], 2'b0} + level_1[3];
117.         // Multiplied by 9
118.         level_2[3] <= {level_1[4], 3'b0} + level_1[4];
119.         // Multiplied by 12
120.         level_2[4] <= {level_1[5], 3'b0} + {level_1[5], 2'b0};
121.         level_3 <= level_2[0] + level_2[1]+ level_2[2]+ level_2[3]+ lev
           el_2[4];
122.         // level_4 <= level_3/115;
123.
124.         final_result <= level_3/159;
125.     end
126. end
127.
128. assign data_out = final_result;
129. line line_buffer1 (
130.     .clock      (clk),
131.     .clken      (data_en),
132.     .shiftin     (data_in),
133.     .shiftout    (iline2),
134.     .taps        ()
135. );
136. defparam
137.     line_buffer1.WD = 8,

```

```
138.     line_buffer1.SIZE  = WIDTH;
139.
140. line line_buffer2 (
141.     .clock      (clk),
142.     .clken      (data_en),
143.     .shiftin    (iline2),
144.     .shiftout   (iline3),
145.     .taps       ()
146. );
147. defparam
148.     line_buffer2.WD      = 8,
149.     line_buffer2.SIZE   = WIDTH;
150.
151. line line_buffer3 (
152.
153.     .clock      (clk),
154.     .clken      (data_en),
155.     .shiftin    (iline3),
156.     .shiftout   (iline4),
157.     .taps       ()
158. );
159. defparam
160.     line_buffer3.WD      = 8,
161.     line_buffer3.SIZE   = WIDTH;
162.
163. line line_buffer4 (
164.
165.     .clock      (clk),
166.     .clken      (data_en),
167.     .shiftin    (iline4),
168.     .shiftout   (iline5),
169.     .taps       ()
170. );
171. defparam
172.     line_buffer4.WD      = 8,
173.     line_buffer4.SIZE   = WIDTH;
174.
175. endmodule
```

### Sobel filter module

```
1. module sobel
```

```
2.
3.
4.
5.    ( input    clk,
6.      input    rst,
7.
8.
9.    input      [8:0] in_data,
10.   input    data_en,
11.
12.   output [11:0] out_data
13.
14.   );
15.
16. parameter WIDTH = 1024;
17. wire [8:0] iline1;
18. wire [8:0] iline2;
19. reg  [8:0] oline0[2:0];
20. reg  [8:0] oline1[2:0];
21. reg  [8:0] oline2[2:0];
22. reg  [11: 0]   gx;
23. reg  [11: 0]   gy;
24. reg          [11: 0] gx_level_1[ 2: 0];
25. reg          [11: 0] gy_level_1[ 2: 0];
26. reg  [11: 0]   gx_magnitude;
27. reg  [11: 0]   gy_magnitude;
28. reg  [11: 0]   gx_m;
29. reg  [11: 0]   gy_m;
30. reg  [11: 0]   g_magnitude;
31. reg  [11: 0]   g_m;
32. reg  [15:0]   gy_100;
33. reg  [15:0]   gx_41;
34. reg  [15:0]   gx_241;
35. reg          neg,neg1;
36. reg  [3:0]    direction;
37. reg  [11: 0]  fresult;
38. integer      i;
39.
40.
41. always @(posedge clk or posedge rst)
42. begin
43.   if (rst)
44.   begin
```

```
45.
46.     for (i = 2; i >= 0; i = i-1)
47.     begin
48.         oline0[i] <= 9'b0;
49.         oline1[i] <= 9'b0;
50.         oline2[i] <= 9'b0;
51.         gx_level_1[i] <= 12'h0;
52.         gy_level_1[i] <= 12'h0;
53.
54.     end
55.
56.     gx          <= 12'h000;
57.     gy          <= 12'h000;
58.     fresult     <= 12'h000;
59.     gx_magnitude <= 12'h000;
60.     gy_magnitude <= 12'h000;
61.     g_magnitude <= 12'h000;
62.     gx_m        <= 12'h000;
63.     gy_m        <= 12'h000;
64.     g_m         <= 12'h000;
65.     gy_100     <= 16'h000;
66.     gx_41      <= 16'h000;
67.     gx_241     <= 16'h000;
68.     neg        <= 1'b0;
69.     direction  <= 4'h0;
70. end
71. else if (data_en)
72. begin
73.     oline0[2] <= oline0[1];
74.     oline1[2] <= oline1[1];
75.     oline2[2] <= oline2[1];
76.     oline0[1] <= oline0[0];
77.     oline1[1] <= oline1[0];
78.     oline2[1] <= oline2[0];
79.     oline0[0] <= in_data;
80.     oline1[0] <= iline1;
81.     oline2[0] <= iline2;
82.     gx_level_1[0] <= oline0[0] + {oline1[0],1'b0}+ oline2[0];
83.
84.     gx_level_1[1] <= oline0[2] + {oline1[2],1'b0} + oline2[2];
85.     gx <= gx_level_1[0] - gx_level_1[1];
86.     // Calculate Gy
87.     gy_level_1[0] <= oline0[0] + {oline0[1],1'b0}+oline0[2];
```

```

88.
89.
90.     gy_level_1[1]  <=  oline2[0] + {oline2[1],1'b0}+oline2[2];
91.     gy <= gy_level_1[0] - gy_level_1[1] ;
92.     // Calculate the magnitude G
93.     gx_magnitude  <=      (gx[11]) ? (~gx) + 12'h001 : gx;
94.     gy_magnitude  <=      (gy[11]) ? (~gy) + 12'h001 : gy;
95.     neg           <=      gx[11]^gy[11];
96.     gy_100        <=  {gy_magnitude,6'b0}    +{gy_magnitude,5'b0}    +{g
    y_magnitude,2'b0};
97.     gx_41         <=  {gx_magnitude,5'b0}    +{gx_magnitude,3'b0}    +gx
    _magnitude;
98.     gx_241        <=  {gx_magnitude,8'b0}    -
    {gx_magnitude,4'b0}    +gx_magnitude;
99.     g_magnitude  <=  gx_magnitude    +    gy_magnitude;
100.    neg1          <=  neg;
101.
102.    //100gy<41gx
103.    if(gy_100<=gx_41)
104.    begin
105.        direction <=  4'b0001;
106.        g_m       <=  g_magnitude;
107.    end
108.    else if((gy_100>gx_41)&&(gy_100<gx_241)&&(neg1==1'b0))
109.    begin
110.        direction <=  4'b0010;
111.        g_m       <=  g_magnitude;
112.    end
113.    else if((gy_100>gx_41)&&(gy_100<gx_241)&&(neg1==1'b1))
114.    begin
115.        direction <=  4'b0100;
116.        g_m       <=  g_magnitude;
117.    end
118.    else
119.    begin
120.        direction <=  4'b1000;
121.        g_m       <=  g_magnitude;
122.    end
123.    // Calculate the final result
124.
125.    fresult[11:8]    <=  direction;
126.    fresult[7:0]    <=  (g_m[11:10] == 2'b0) ? g_m[9:2] : 8'hFF
;

```

```
127.
128.     end
129. end
130.
131.
132. assign out_data = fresult;
133.
134. line u0 (
135.     .clken(data_en),
136.     .clock(clk),
137.     .shiftin(in_data),
138.     .shiftout(iline1)
139. );
140. defparam
141.     u0.WD      = 9,
142.     u0.SIZE = WIDTH;
143. line u1 (
144.     .clken(data_en),
145.     .clock(clk),
146.     .shiftin(iline1),
147.     .shiftout(iline2)
148. );
149. defparam
150.     u1.WD      = 9,
151.     u1.SIZE = WIDTH;
152. endmodule
```

### Non-Maximum Suppression Module

```
1. module nm_s (
2.     // Inputs
3.     input    clk,
4.     input    rst,
5.
6.
7.     input  [11:0] data_in,
8.     input  data_en,
9.
10.    output [11:0] data_out
11. );
12.
13.
```

```
14. parameter WIDTH = 1024; // Image width in pixels
15. wire [11:0] iline1;
16. wire [11:0] iline2;
17.
18.
19. reg [11:0] oline0[2:0];
20. reg [11:0] oline1[2:0];
21. reg [11:0] oline2[2:0];
22.
23. reg [11:0] r_line0[2:0];
24. reg [11:0] r_line1[2:0];
25. reg [11:0] r_line2[2:0];
26.
27. reg [ 7: 0] sobel_result;
28. reg [ 3: 0] direction;
29. reg [11: 0] fresult;
30. integer i;
31.
32. always @(posedge clk)
33. begin
34.     if (rst)
35.     begin
36.
37.         for (i = 2; i >= 0; i = i-1)
38.         begin
39.             oline0[i] <= 12'h0;
40.             oline1[i] <= 12'h0;
41.             oline2[i] <= 12'h0;
42.         end
43.
44.     end
45.     else if (data_en)
46.     begin
47.         oline0[2] <= oline0[1];
48.         oline1[2] <= oline1[1];
49.         oline2[2] <= oline2[1];
50.
51.         oline0[1] <= oline0[0];
52.         oline1[1] <= oline1[0];
53.         oline2[1] <= oline2[0];
54.
55.         oline0[0] <= data_in;
56.         oline1[0] <= iline1;
```

```
57.     oline2[0]  <= iline2;
58.     case (oline1[1][11:8])
59.         4'b0001 :
60.             begin
61.                 if((oline1[1][7:0]>oline1[2][7:0])&&(oline1[1][7:0]>olin
        e1[0][7:0]))
62.                     begin
63.                         direction      <=  4'b0001;
64.                         sobel_result   <=  oline1[1][7:0];
65.                     end
66.                 else
67.                     begin
68.                         direction      <=  4'b0;
69.                         sobel_result   <=  0;
70.                     end
71.             end
72.         4'b0010 :
73.             begin
74.                 if((oline1[1][7:0]>oline2[2][7:0])&&(oline1[1][7:0]>olin
        e0[0][7:0]))
75.                     begin
76.                         direction      <=  4'b0010;
77.                         sobel_result   <=  oline1[1][7:0];
78.                     end
79.                 else
80.                     begin
81.                         direction      <=  4'b0;
82.                         sobel_result   <=  0;
83.                     end
84.             end
85.         4'b0100 :
86.             begin
87.                 if((oline1[1][7:0]>oline2[0][7:0])&&(oline1[1][7:0]>olin
        e0[2][7:0]))
88.                     begin
89.                         direction      <=  4'b0100;
90.                         sobel_result   <=  oline1[1][7:0];
91.                     end
92.                 else
93.                     begin
94.                         direction      <=  4'b0;
95.                         sobel_result   <=  0;
96.                     end
```

```
97.         end
98.         4'b1000 :
99.         begin
100.            if((oline1[1][7:0]>oline0[1][7:0])&&(oline1[1][7:0]>oline2[1][7:0]))
101.                begin
102.                    direction    <=  4'b1000;
103.                    sobel_result <=  oline1[1][7:0];
104.                end
105.            else
106.                begin
107.                    direction    <=  4'b0;
108.                    sobel_result <=  0;
109.                end
110.            end
111.            default :
112.            begin
113.                direction    <=  4'b0;
114.                sobel_result <=  0;
115.            end
116.        endcase
117.        fresult[11:8] <=  direction;
118.        fresult[7:0]  <=  sobel_result;
119.    end
120. end
121.
122. assign data_out = fresult;
123. line_buffer_1 (
124.     .clock      (clk),
125.     .clken      (data_en),
126.     .shiftin     (data_in),
127.     .shiftout    (iline1),
128.     .taps        ()
129. );
130. defparam
131.     buffer_1.WD    = 12,
132.     buffer_1.SIZE = WIDTH;
133.
134. line_buffer_2 (
135.     .clock      (clk),
136.     .clken      (data_en),
137.     .shiftin     (iline1),
138.     .shiftout    (iline2),
```

```
139.     .taps          ()
140. );
141. defparam
142.     buffer_2.WD      = 12,
143.     buffer_2.SIZE   = WIDTH;
144.
145. endmodule
```

### Double Thresholding with Hysteresis Module

```
1. module double_threshold_filtering (
2.
3.     input    clk,
4.     input    rst,
5.
6.
7.     input  [11:0] in_data,
8.     input  data_en,
9.
10.    output [7:0] out_data
11. );
12.
13. parameter WIDTH = 1024; // Image width in pixels
14.
15. wire [11:0] iline1;
16. wire [11:0] iline2;
17.
18. reg  [11:0] oline0[2:0];
19. reg  [11:0] oline1[2:0];
20. reg  [11:0] oline2[2:0];
21.
22. reg  [7:0]  T_in;
23. reg  [7:0]  H_T;
24. reg  [7:0]  L_T;
25.
26.
27.
28. reg  [ 7: 0] result;
29.
30.
31. integer          i;
32. always @(posedge clk)
```

```
33. begin
34.   if (rst )
35.     begin
36.       //nm_result <= 8'h00;
37.       H_T      <= 8'h00;
38.       L_T      <= 8'h00;
39.       T_in     <= 8'h00;
40.       result   <= 8'h00;
41.       for (i = 2; i >= 0; i = i-1)
42.         begin
43.           oline0[i] <= 12'b0;
44.           oline1[i] <= 12'b0;
45.           oline2[i] <= 12'b0;
46.         end
47.
48.     end
49.   else if (data_en)
50.     begin
51.
52.       oline0[2] <= oline0[1];
53.       oline1[2] <= oline1[1];
54.       oline2[2] <= oline2[1];
55.
56.
57.       oline0[1] <= oline0[0];
58.       oline1[1] <= oline1[0];
59.       oline2[1] <= oline2[0];
60.
61.       oline0[0] <= in_data;
62.       oline1[0] <= iline1;
63.       oline2[0] <= iline2;
64.
65.       //nm_result <= in_data;
66.       T_in     <= (in_data[7:0]>T_in) ? in_data[7:0] : T_in;
67.
68.       H_T      <= T_in/6;
69.
70.       L_T      <= {1'b0,H_T[7:1]};
71.
72.       if(oline1[1][7:0]<L_T)
73.         begin
74.           result <=8'h00;
75.         end
```

```
76.     else if (oline1[1][7:0]>H_T)
77.         begin
78.             result <=8'hFF;
79.         end
80.     else
81.         begin
82.
83.
84.             if( (oline0[1][11:8]!=0)||(oline2[1][11:8]!=0)||
85.                 (oline0[2][11:8]!=0)||(oline2[0][11:8]!=0)||
86.                 (oline0[0][11:8]!=0)||(oline2[2][11:8]!=0)||
87.                 (oline1[0][11:8]!=0)||(oline1[2][11:8]!=0))
88.                 begin
89.                     result <= 8'hFF;
90.                 end
91.             else
92.                 begin
93.                     result <= 8'h00;
94.                 end
95.             end
96.
97.         end
98.     end
99.
100. assign out_data = result;
101.
102. line u0 (
103.     .clken(data_en),
104.     .clock(clk),
105.     .shiftin(in_data),
106.     .shiftout(iline1)
107. );
108. defparam
109.     u0.WD = 12,
110.     u0.SIZE = WIDTH;
111.
112.
113. line u1 (
114.     .clken(data_en),
115.     .clock(clk),
116.     .shiftin(iline1),
117.     .shiftout(iline2)
118. );
```

```
119. defparam
120.     u1.WD      = 12,
121.     u1.SIZE = WIDTH;
122.
123. endmodule
```

## Appendix B Codes of Canny Edge Detection with HLS approach

```
1. #include "HLS/hls.h"
2. #include "HLS/math.h"
3.
4. #include "HLS/ac_int.h"
5.
6.
7. #define N 1024
8. #define M 3
9. #define RGB_D N*7-11
10. //typedef ac_int<8, false> index_t;
11.
12.
13. struct int_v24 {
14. unsigned char data[3];
15. };
16. struct int_v32 {
17. unsigned char data[4];
18. };
19.
20. struct direction_sobel {
21. unsigned char data;
22. unsigned char direction;
23. };
24.
25.
26. // Default stream behavior
27. hls_avalon_streaming_component hls_always_run_component
28. component void filters(
29.     ihc::stream_in<int_v24, ihc::bitsPerSymbol<8>,ihc::usesPackets
    <true> >& a,
30.     ihc::stream_out<int_v32, ihc::bitsPerSymbol<8>, ihc::usesPackets
    <true> >& b) {
31.
32.
33. bool start_of_packet = false;
34. bool end_of_packet = false;
35.
36.
```

```
37. int_v24 a1;
38. int_v32 b1;
39. int_v24 rgb_buffer[RGB_D];
40. int buffer[3];
41. int grey;
42.
43.
44. int g_levelx0,g_levelx1,g_levelx2,g_levelx3,g_levelx4,g_levelx5,g_levelx6;
45. int g_level21, g_level22, g_level23, g_level24, g_level25, g_level26,g_level
    3,g_level4;
46. int s_levelx0,s_levelx1,s_levelx2;
47. int s_levely0,s_levely1;
48. int s_level2x,s_level2y;
49. int s_level3x,s_level3y;
50. int s_level3,s_level4,s_level5;
51. unsigned char Threshold,H_T,L_T,final_result;
52.
53. float s_grad;
54.
55. unsigned short int line0[N], line1[N], line2[N], line3[N],line4[N];
56. short int gaussian_line0[N], gaussian_line1[N],gaussian_line2[N];
57.
58. direction_sobel sobel_result, non_max_result;
59.
60. direction_sobel sobel_line0[N], sobel_line1[N],sobel_line2[N];
61. direction_sobel non_max_line0[N],non_max_line1[N],non_max_line2[N];
62.
63.
64. while(!end_of_packet) {
65.     // Blocking read from the input stream
66.     a1 = a.read(start_of_packet, end_of_packet);
67.     #pragma unroll
68.     for(int i=0;i < 3; i++)
69.     {
70.         buffer[i]=a1.data[i];
71.     }
72.     //grey result
73.
74.     grey=(buffer[0]*76+buffer[1]*150+buffer[2]*29)/256;
75.     // 5 buffered lines
76.     #pragma unroll
77.     for (int i = N - 1; i > 0; --i) {
78.         line0[i] = line0[i-1];
```

```

79.         line1[i] = line1[i-1];
80.         line2[i] = line2[i-1];
81.         line3[i] = line3[i-1];
82.         line4[i] = line4[i-1];
83.     }
84.     line0[0] = grey;
85.     line1[0] = line0[N-1];
86.     line2[0] = line1[N-1];
87.     line3[0] = line2[N-1];
88.     line4[0] = line3[N-1];
89.     g_level21= (line0[N-1]*2)      + (line0[N-
2]*4)  + (line0[N-3]*5)      + (line0[N-4]*4)      + (line0[N-
5]*2);
90.     g_level22= (line1[N-1]*4)      + (line1[N-
2]*9)  + (line1[N-3]*12)     + (line1[N-4]*9)      + (line1[N-
5]*4);
91.     g_level23= (line2[N-1]*5)      + (line2[N-
2]*12) + (line2[N-3]*15)     + (line2[N-4]*12)     + (line2[N-
5]*5);
92.     g_level24= (line3[N-1]*4)      + (line3[N-
2]*9)  + (line3[N-3]*12)     + (line3[N-4]*9)      + (line3[N-
5]*4);
93.     g_level25= (line4[N-1]*2)      + (line4[N-
2]*4)  + (line4[N-3]*5)      + (line4[N-4]*4)      + (line4[N-
5]*2);
94.
95.
96.
97.     g_level3=      g_level21+g_level22+g_level23 +g_level24+g_level
25;
98.     //g_level4=      g_level3/115;
99.     g_level4=      g_level3/159;
100.
101.
102.     #pragma unroll
103.     for (int i = N - 1; i > 0; --i) {
104.         gaussian_line0[i] = gaussian_line0[i-1];
105.         gaussian_line1[i] = gaussian_line1[i-1];
106.         gaussian_line2[i] = gaussian_line2[i-1];
107.     }
108.     gaussian_line0[0] = g_level4;
109.     gaussian_line1[0] = gaussian_line0[N-1];
110.     gaussian_line2[0] = gaussian_line1[N-1];

```

```
111.         //Sobel filter
112.         s_levelx0=gaussian_line0[N-1] +(gaussian_line1[N-
113.         1]*2) +gaussian_line2[N-1];
114.         s_levelx1=gaussian_line0[N-3] +(gaussian_line1[N-
115.         3]*2) +gaussian_line2[N-3];
116.         s_levely0=gaussian_line0[N-1]+(gaussian_line0[N-
117.         2]*2)+gaussian_line0[N-3];
118.         s_levely1=gaussian_line2[N-1]+(gaussian_line2[N-
119.         2]*2)+gaussian_line2[N-3];
120.         s_level2x=s_levelx0-s_levelx1;
121.         s_level2y=s_levely0-s_levely1;
122.         s_level3=abs(s_level2x) + abs(s_level2y);
123.         //calculate the direction
124.         s_level4=s_level3>>2;
125.         s_level5=(s_level4<255)?s_level4:255;
126.         s_grad=(float)s_level2y/(float)s_level2x;
127.         if((s_grad>=-0.415)&&(s_grad<=0.415)){
128.             //direction 1: -22.5 degrees to 22.5 degrees
129.             sobel_result.direction=1;
130.             sobel_result.data=s_level5;
131.         }
132.         else if ((s_grad>0.415)&&(s_grad<=2.414)){
133.             //direction 2: 22.5 degrees to 67.5 degrees
134.             sobel_result.direction=2;
135.             sobel_result.data=s_level5;
136.         }
137.         else if ((s_grad<-0.415)&&(s_grad>=-2.414)){
138.             //direction 3: -22.5 degrees to -67.5 degrees
139.             sobel_result.direction=3;
140.             sobel_result.data=s_level5;
```

```
150.     }
151.
152.     else{
153.
154.         //direction 4: 67.5 to 90 degrees & -67.5 to -90 degrees
155.         sobel_result.direction=4;
156.         sobel_result.data=s_level5;
157.     }
158.
159.
160.     //Buffer another 3 lines
161.     #pragma unroll
162.     for (int i = N - 1; i > 0; --i) {
163.         sobel_line0[i] = sobel_line0[i-1];
164.         sobel_line1[i] = sobel_line1[i-1];
165.         sobel_line2[i] = sobel_line2[i-1];
166.     }
167.     sobel_line0[0] = sobel_result;
168.     sobel_line1[0] = sobel_line0[N-1];
169.     sobel_line2[0] = sobel_line1[N-1];
170.
171.
172.     if(sobel_line1[N-2].direction==1){
173.
174.         if((sobel_line1[N-2].data>sobel_line1[N-
175.             1].data)&&(sobel_line1[N-2].data>sobel_line1[N-3].data)){
176.
177.             non_max_result.direction=1;
178.             non_max_result.data=sobel_line1[N-
179.                 2].data;
180.         }
181.         else {
182.             non_max_result.direction=0;
183.             non_max_result.data=0;
184.         }
185.     }
186.     else if(sobel_line1[N-2].direction==2){
187.         if((sobel_line1[N-2].data>sobel_line0[N-
188.             1].data)&&(sobel_line1[N-2].data>sobel_line2[N-3].data)){
189.
190.             non_max_result.direction=1;
191.             non_max_result.data=sobel_line1[N-
192.                 2].data;
```

```
189.         }
190.         else {
191.             non_max_result.direction=0;
192.             non_max_result.data=0;
193.         }
194.     }
195.     else if(sobel_line1[N-2].direction==3){
196.         if((sobel_line1[N-2].data>sobel_line2[N-
197.             1].data)&&(sobel_line1[N-2].data>sobel_line0[N-3].data)){
198.             non_max_result.direction=1;
199.             non_max_result.data=sobel_line1[N-
200.                 2].data;
201.         }
202.         else {
203.             non_max_result.direction=0;
204.             non_max_result.data=0;
205.         }
206.     }
207.     else if(sobel_line1[N-2].direction==4){
208.         if((sobel_line1[N-2].data>sobel_line0[N-
209.             2].data)&&(sobel_line1[N-2].data>sobel_line2[N-2].data)){
210.             non_max_result.direction=1;
211.             non_max_result.data=sobel_line1[N-
212.                 2].data;
213.         }
214.         else {
215.             non_max_result.direction=0;
216.             non_max_result.data=0;
217.         }
218.     }
219.     else{
220.         non_max_result.direction=0;
221.         non_max_result.data=0;
222.     }
223.     #pragma unroll
224.     for (int i = N - 1; i > 0; --i) {
225.         non_max_line0[i] = non_max_line0[i-1];
226.         non_max_line1[i] = non_max_line1[i-1];
227.         non_max_line2[i] = non_max_line2[i-1];
```

```

228.         }
229.         non_max_line0[0] = non_max_result;
230.         non_max_line1[0] = non_max_line0[N-1];
231.         non_max_line2[0] = non_max_line1[N-1];
232.
233.
234.         if(non_max_line2[N-1].data>Threshold){
235.             Threshold=non_max_line2[N-1].data;
236.         }
237.
238.         H_T=Threshold/6;
239.         L_T=H_T/2;
240.
241.
242.         if (non_max_line1[N-2].data<L_T){
243.             final_result=0;
244.         }
245.         else if(non_max_line1[N-2].data>H_T){
246.             final_result=255;
247.         }
248.         else
249.         {
250.
251.             if((non_max_line1[N-2].direction==1)&&( non_max_line0[N-
252.                 1].direction==1)|| (non_max_line0[N-3].direction==1)||
253.                 (non_max_line0[N-
254.                 2].direction==1)|| (non_max_line1[N-1].direction==1)||
255.                 (non_max_line1[N-
256.                 3].direction==1)|| (non_max_line2[N-1].direction==1)||
257.                 (non_max_line2[N-
258.                 2].direction==1)|| (non_max_line2[N-3].direction==1))){
259.                 final_result=255;
260.             }
261.
262.         }
263.         else{
264.             final_result=0;
265.         }
266.     }
267. }
268. #pragma unroll
269. for (int i = RGB_D - 1; i > 0; --i) {
270.     rgb_buffer[i] = rgb_buffer[i-1];
271. }

```

```
267.         rgb_buffer[0] = a1;
268.
269.
270.         b1.data[0]=final_result;
271.
272.         b1.data[1]=rgb_buffer[RGB_D - 1].data[0];
273.         b1.data[2]=rgb_buffer[RGB_D - 1].data[1];
274.         b1.data[3]=rgb_buffer[RGB_D - 1].data[2];
275.         b.write(b1, start_of_packet, end_of_packet);
276.
277.     }
278. }
```

## ***Appendix C Codes of Harris Corner Detection with HDL approach***

### **Sobel Filter Module**

```
1. module sobel
2.
3.
4.
5.     ( input      clk,
6.       input      rst,
7.
8.
9.       input      [8:0] in_data,
10.      input      data_en,
11.
12.      output [23:0] gx2_result,
13.      output [23:0] gy2_result,
14.      output [23:0] gxy_result
15.
16.    );
17.
18. parameter WIDTH = 1024;
19.
20.
21. wire [8:0] iline1;
22. wire [8:0] iline2;
23.
24. wire [23: 0] gx2;
25. wire [23: 0] gy2;
26. wire [23: 0] gxy;
27.
28. reg [8:0] oline0[2:0];
29. reg [8:0] oline1[2:0];
30. reg [8:0] oline2[2:0];
31.
32.
33. reg [11: 0] gx;
34. reg [11: 0] gy;
```

```
35.
36.
37.
38.
39. reg          [11: 0] gx_level_1[ 2: 0];
40.
41.
42. reg          [11: 0] gy_level_1[ 2: 0];
43.
44.
45.
46.
47. reg [23: 0] gxy_r;
48. reg [23: 0] gx2_r;
49. reg [23: 0] gy2_r;
50.
51. integer      i;
52.
53. always @(posedge clk or posedge rst)
54. begin
55.     if (rst)
56.     begin
57.
58.         for (i = 2; i >= 0; i = i-1)
59.         begin
60.             oline0[i] <= 9'b0;
61.             oline1[i] <= 9'b0;
62.             oline2[i] <= 9'b0;
63.             gx_level_1[i] <= 12'h0;
64.             gy_level_1[i] <= 12'h0;
65.
66.         end
67.
68.         gx          <= 12'h000;
69.         gy          <= 12'h000;
70.         gx2_r       <= 24'h000;
71.         gy2_r       <= 24'h000;
72.         gxy_r       <= 24'h000;
73.
74.
75.     end
```

```
76.     else if (data_en)
77.     begin
78.
79.         oline0[2] <= oline0[1];
80.         oline1[2] <= oline1[1];
81.         oline2[2] <= oline2[1];
82.
83.
84.         oline0[1] <= oline0[0];
85.         oline1[1] <= oline1[0];
86.         oline2[1] <= oline2[0];
87.
88.         oline0[0]  <= in_data;
89.         oline1[0]  <= iline1;
90.         oline2[0]  <= iline2;
91.
92.         gx_level_1[0]  <=  oline0[0] + {oline1[0],1'b0}+ oline2[0];
93.
94.         gx_level_1[1]  <=  oline0[2] + {oline1[2],1'b0} + oline2[2];
95.
96.         gx <= gx_level_1[0] - gx_level_1[1];
97.
98.         // Calculate Gy
99.
100.        gy_level_1[0]  <=  oline0[0] + {oline0[1],1'b0}+oline0[2];
101.        gy_level_1[1]  <=  oline2[0] + {oline2[1],1'b0}+oline2[2];
102.        gy <= gy_level_1[0] - gy_level_1[1] ;
103.
104.        gx2_r <= gx2;
105.
106.        gy2_r <= gy2;
107.
108.        gxy_r <= gxy;
109.
110.    end
111. end
112.
113.
114.
115. assign  gxy_result=gxy_r;
116. assign  gx2_result=gx2_r;
```

```
117. assign gy2_result=gy2_r;
118. line u0 (
119.   .clken(data_en),
120.   .clock(clk),
121.   .shiftin(in_data),
122.   .shiftout(iline1)
123. );
124. defparam
125.   u0.WD      = 9,
126.   u0.SIZE = WIDTH;
127.
128.
129. line u1 (
130.   .clken(data_en),
131.   .clock(clk),
132.   .shiftin(iline1),
133.   .shiftout(iline2)
134. );
135. defparam
136.   u1.WD      = 9,
137.   u1.SIZE = WIDTH;
138.
139. mult u2 (
140.   .clock(clk),
141.   .clken(data_en),
142.   .dataa(gx),
143.   .datab(gx),
144.   .result(gx2)
145. );
146. mult u3 (
147.   .clock(clk),
148.   .clken(data_en),
149.   .dataa(gy),
150.   .datab(gy),
151.   .result(gy2)
152. );
153. mult u4 (
154.   .clock(clk),
155.   .clken(data_en),
156.   .dataa(gx),
157.   .datab(gy),
```

```
158.     .result(gxy)
159. );
160. endmodule
```

### Gaussian Filter Module

```
1. module gaussian_filter (
2.     // Inputs
3.     clk,
4.     reset,
5.
6.     data_in,
7.     data_en,
8.
9.     // Outputs
10.    data_out
11. );
12.
13.
14. parameter WIDTH = 1024; // Image width in pixels
15.
16.
17. // Inputs
18. input                clk;
19. input                reset;
20.
21. input                [ 23: 0] data_in;
22. input                data_en;
23.
24.
25. // Outputs
26. output               [ 31: 0] data_out;
27.
28.
29. // Internal Wires
30. wire                [ 31: 0] iline2;
31. wire                [ 31: 0] iline3;
32. wire                [ 31: 0] iline4;
33. wire                [ 31: 0] iline5;
34. wire                [ 31: 0] data_in_reg;
35. wire                [ 9: 0]  de;
```

```
36. wire          [ 31: 0]   level_4;
37. // Internal Registers
38. reg          [ 31: 0]   oline_1[ 4: 0];
39. reg          [ 31: 0]   oline_2[ 4: 0];
40. reg          [ 31: 0]   oline_3[ 4: 0];
41. reg          [ 31: 0]   oline_4[ 4: 0];
42. reg          [ 31: 0]   oline_5[ 4: 0];
43.
44.
45. reg          [ 31: 0]   level_1[6: 0];
46.
47. reg          [ 31: 0]   level_2[ 4: 0];
48. reg          [ 31: 0]   level_3;
49.
50.
51.
52. reg          [ 31: 0] final_result;
53.
54. // Integers
55. integer          i;
56.
57.
58. // Gaussian Smoothing Filter
59. //
60. //          [ 2 4 5 4 2 ]
61. //          [ 4 9 12 9 4 ]
62. // 1 / 159 [ 5 12 15 12 5 ]
63. //          [ 4 9 12 9 4 ]
64. //          [ 2 4 5 4 2 ]
65. //
66. // mask X
67.
68. always @(posedge clk)
69. begin
70.     if (reset == 1'b1)
71.     begin
72.         for (i = 4; i >= 0; i = i-1)
73.         begin
74.             oline_1[i] <= 32'h00;
75.             oline_2[i] <= 32'h00;
76.             oline_3[i] <= 32'h00;
```

```
77.         oline_4[i] <= 32'h00;
78.         oline_5[i] <= 32'h00;
79.         level_1[i] <= 32'h000;
80.     end
81.
82.
83.     end
84.     else if (data_en )
85.     begin
86.         for (i = 4; i > 0; i = i-1)
87.         begin
88.             oline_1[i] <= oline_1[i-1];
89.             oline_2[i] <= oline_2[i-1];
90.             oline_3[i] <= oline_3[i-1];
91.             oline_4[i] <= oline_4[i-1];
92.             oline_5[i] <= oline_5[i-1];
93.         end
94.         oline_1[0] <= data_in_reg;
95.         oline_2[0] <= iline2;
96.         oline_3[0] <= iline3;
97.         oline_4[0] <= iline4;
98.         oline_5[0] <= iline5;
99.
100.        level_1[0] <= {oline_1[0], 1'b0} + {oline_1[4], 1'b0} + {oline_5[0], 1'b0} + {oline_5[4], 1'b0};
101.            //times 4
102.        level_1[1] <= {oline_1[1], 2'b0} + {oline_1[3], 2'b0} + {oline_2[0], 2'b0} + {oline_2[4], 2'b0};
103.        level_1[2] <= {oline_4[0], 2'b0} + {oline_4[4], 2'b0} + {oline_5[1], 2'b0} + {oline_5[3], 2'b0};
104.
105.            //5
106.        level_1[3] <= oline_1[2] + oline_5[2] + oline_3[0] + oline_3[4];
107.            //9
108.        level_1[4] <= oline_2[1] + oline_2[3] + oline_4[1] + oline_4[3];
109.            //12
110.        level_1[5] <= oline_2[2] + oline_4[2] + oline_3[1] + oline_3[3];
111.            //15
```

```
112.         level_1[6] <= {oline_3[2], 4'b0} - oline_3[2];
113.
114.
115.
116.
117.
118.         level_2[0] <= level_1[0]+ level_1[6];
119.
120.         level_2[1] <= level_1[1]+ level_1[2];
121.
122.         // Multiplied by 5
123.         level_2[2] <= {level_1[3], 2'b0} + level_1[3];
124.         // Multiplied by 9
125.         level_2[3] <= {level_1[4], 3'b0} + level_1[4];
126.         // Multiplied by 12
127.         level_2[4] <= {level_1[5], 3'b0} + {level_1[5], 2'b0};
128.
129.
130.
131.         level_3 <= level_2[0] + level_2[1]+ level_2[2]+ level_2[3]+ level_2[4];
132.
133.
134.         //level_4 <= level_3>>7;
135.
136.         final_result <= level_4;//(level_3[31]==1'b0)?({7'b0,level_3[31:7]}):({7'b1,level_3[31:7]});
137.     end
138. end
139.
140. assign data_in_reg = (data_in[23]==1'b0)? ({8'b0,data_in}):({8'b1,data_in});
141. assign data_out = final_result;
142. assign de = 159;
143.
144.
145.
146.
147. line line_buffer1 (
148.     .clock (clk),
149.     .clken (data_en),
```

```
150.     .shiftin      (data_in_reg),
151.     .shiftout    (iline2),
152.     .taps        ()
153. );
154. defparam
155.     line_buffer1.WD = 32,
156.     line_buffer1.SIZE = WIDTH;
157.
158. line line_buffer2 (
159.     .clock      (clk),
160.     .clken      (data_en),
161.     .shiftin    (iline2),
162.     .shiftout   (iline3),
163.     .taps      ()
164. );
165. defparam
166.     line_buffer2.WD = 32,
167.     line_buffer2.SIZE = WIDTH;
168.
169. line line_buffer3 (
170.
171.     .clock      (clk),
172.     .clken      (data_en),
173.     .shiftin    (iline3),
174.     .shiftout   (iline4),
175.     .taps      ()
176. );
177. defparam
178.     line_buffer3.WD = 32,
179.     line_buffer3.SIZE = WIDTH;
180.
181. line line_buffer4 (
182.
183.     .clock      (clk),
184.     .clken      (data_en),
185.     .shiftin    (iline4),
186.     .shiftout   (iline5),
187.     .taps      ()
188. );
189. defparam
190.     line_buffer4.WD = 32,
```

```
191.     line_buffer4.SIZE  = WIDTH;
192.
193. div divde(
194.     .clock(clk),
195.     .clken(data_en),
196.     .denom(de),
197.     .numer(level_3),
198.     .quotient(level_4),
199.     );
200.
201. endmodule
```

### Harris Corner Detection Module

```
1. module harris_m (
2.     // Inputs
3.     input    clk,
4.     input    rst,
5.
6.
7.     input  [31:0] gx2_in,
8.     input  [31:0] gy2_in,
9.     input  [31:0] gxy_in,
10.    input  data_en,
11.
12.    output [63:0] data_out
13. );
14.
15.
16. parameter WIDTH = 1024; // Image width in pixels
17.
18. wire    [63: 0]    gxy2;
19. wire    [63: 0]    gx2y2;
20. wire    [63: 0]    gxy22;
21. reg     [63: 0]    gxy2_reg;
22. reg     [63: 0]    gx2y2_reg;
23. reg     [63: 0]    gxy22_reg;
24. reg     [ 31: 0]    gx2_reg;
25. reg     [ 31: 0]    gy2_reg;
26. reg     [ 31: 0]    gxy_reg;
27. reg     [ 31: 0]    gxpy_reg;
```

```
28.
29. reg      [ 63: 0]   R_111;
30. reg      [ 63: 0]   R_112;
31. reg      [ 63: 0]   R_12;
32.
33. reg      [ 63: 0]   fresult;
34. // State Machine Registers
35.
36. // Integers
37.
38.
39. always @(posedge clk)
40. begin
41.     if (rst)
42.     begin
43.         gx2_reg    <=  32'h0;
44.         gy2_reg    <=  32'h0;
45.         gxy_reg    <=  32'h0;
46.         gxpy_reg   <=  32'h0;
47.         gxy2_reg   <=  64'h0;
48.         gx2y2_reg  <=  64'h0;
49.         gxy22_reg  <=  64'h0;
50.
51.
52.     end
53.     else if (data_en)
54.     begin
55.
56.         gx2_reg    <=  gx2_in;
57.         gy2_reg    <=  gy2_in;
58.         gxy_reg    <=  gxy_in;
59.         gxpy_reg   <=  gx2_in+gy2_in;
60.
61.
62.         gxy2_reg   <=  gxy2;
63.         gx2y2_reg  <=  gx2y2;
64.         gxy22_reg  <=  gxy22;
65.
66.         R_111     <=  gx2y2_reg  -  gxy2_reg;
67.         R_112     <=  {5'b0,gxy22_reg[63:5]};
68.
```

```
69.    R_12    <=    R_111    -    R_112;
70.
71.    fresult <=    (R_12[63]==1'b0)?({1'b0,R_12[62:0]}):(64'h0);
72.
73.    end
74. end
75.
76. /*****
    **
77. *          Combinational Logic
    *
78. *****/
79.
80. assign data_out = fresult;
81.
82. /*****
    **
83. *          Internal Modules
    *
84. *****/
85.
86.
87. mult1 u2 (
88.     .clock(clk),
89.     .clken(data_en),
90.     .dataa(gx2_reg),
91.     .datab(gy2_reg),
92.     .result(gx2y2)
93. );
94. mult1 u3 (
95.     .clock(clk),
96.     .clken(data_en),
97.     .dataa(gxy_reg),
98.     .datab(gxy_reg),
99.     .result(gxy2)
100. );
101. mult1 u4 (
102.     .clock(clk),
103.     .clken(data_en),
```

```
104.     .dataa(gxpy_reg),
105.     .datab(gxpy_reg),
106.     .result(gxy22)
107. );
108.
109. endmodule
```

### Double threshold filtering Module

```
1. module double_threshold_filtering (
2.
3.     input    clk,
4.     input    rst,
5.
6.
7.     input  [63:0] in_data,
8.     input  [63:0] threshold,
9.     input  data_en,
10.
11.    output [7:0] out_data
12. );
13.
14. parameter WIDTH = 1024; // Image width in pixels
15.
16. wire [63:0] iline1;
17. wire [63:0] iline2;
18.
19. reg  [63:0] oline0[2:0];
20. reg  [63:0] oline1[2:0];
21. reg  [63:0] oline2[2:0];
22.
23. reg          [ 7: 0] result;
24.
25.
26. integer          i;
27.
28.
29.
30.
31. always @(posedge clk)
32. begin
```

```
33.   if (rst )
34.   begin
35.       result      <=  8'h00;
36.       for (i = 2; i >= 0; i = i-1)
37.       begin
38.           oline0[i] <= 64'h0;
39.           oline1[i] <= 64'h0;
40.           oline2[i] <= 64'h0;
41.       end
42.
43.   end
44.   else if (data_en)
45.   begin
46.
47.       oline0[2] <= oline0[1];
48.       oline1[2] <= oline1[1];
49.       oline2[2] <= oline2[1];
50.
51.
52.       oline0[1] <= oline0[0];
53.       oline1[1] <= oline1[0];
54.       oline2[1] <= oline2[0];
55.
56.       oline0[0]  <= in_data;
57.       oline1[0]  <= iline1;
58.       oline2[0]  <= iline2;
59.
60.
61.
62.
63.
64.       if ((oline1[1]>threshold)&&
65.           //(oline1[1][63:0]>oline0[1][63:0])&&
66.           //(oline1[1][63:0]>oline2[1][63:0])&&
67.
68.           (oline1[1]>oline0[2])&&
69.           (oline1[1]>oline2[0])&&
70.           (oline1[1]>oline0[0])&&
71.           (oline1[1]>oline2[2]))//&&
72.
73.           //(oline1[1][63:0]>oline1[0][63:0])&&
```

```
74.         //(oline1[1][63:0]>oline1[2][63:0])
75.     )
76.     result <= (oline1[1][30:18]>8'hFF)?(8'hFF):(oline1[1][25:18]);
77.     //result <= (oline1[1][63:20]>16'hFFFF)?(16'hFFFF):(oline1[1][35
       :20]);
78.     else
79.     result <= 8'h00;
80.
81.
82.     end
83. end
84.
85.
86.
87.
88.
89. assign out_data = result;
90.
91. //assign T_in_wire=T_in;
92.
93. line u0 (
94.     .clken(data_en),
95.     .clock(clk),
96.     .shiftin(in_data),
97.     .shiftout(iline1)
98. );
99. defparam
100.     u0.WD = 64,
101.     u0.SIZE = WIDTH;
102.
103.
104. line u1 (
105.     .clken(data_en),
106.     .clock(clk),
107.     .shiftin(iline1),
108.     .shiftout(iline2)
109. );
110. defparam
111.     u1.WD = 64,
112.     u1.SIZE = WIDTH;
113.
```

```
114. endmodule
```

## ***Appendix D Codes of Harris Corner Detection with HLS approach***

```
1. #include "HLS/hls.h"
2. #include "HLS/math.h"
3.
4. #include "HLS/ac_int.h"
5.
6.
7. #define N 1024
8. #define M 3
9. #define RGB_D N*7-11
10.
11.
12.
13. struct int_v24 {
14. unsigned char data[3];
15. };
16. struct int_v32 {
17. unsigned char data[4];
18. };
19.
20. struct direction_sobel {
21. unsigned char data;
22. unsigned char direction;
23. };
24.
25. struct matrix {
26. long I_x2;
27. long I_y2;
28. long I_xy;
29. };
30.
31. struct matrix_long {
32. long long I_x2;
33. long long I_y2;
34. long long I_xy;
35. };
36. struct gaussian_level1 {
```

```
37.
38. long g_levelx0;
39. long g_levelx1;
40. long g_levelx2;
41. long g_levelx3;
42. long g_levelx4;
43. long g_levelx5;
44. long g_levelx6;
45.
46. };
47. struct gaussian_level2 {
48.
49. long g_level20;
50. long g_level21;
51. long g_level22;
52. long g_level23;
53. long g_level24;
54.
55. };
56.
57.
58.
59. // Default stream behavior
60. hls_avalon_streaming_component hls_always_run_component
61. component void harris(
62.     ihc::stream_in<int_v24, ihc::bitsPerSymbol<8>,ihc::usesPackets
        <true> >& a,
63.     ihc::stream_out<int_v24, ihc::bitsPerSymbol<8>,ihc::usesPacket
        s<true> >& b
64.     // ,ihc::stream_out<int, ihc::usesPackets<true> >& c
65.     // ,ihc::stream_out<int, ihc::usesPackets<true> >& d
66.     ) {
67.
68.
69. bool start_of_packet = false;
70. bool end_of_packet = false;
71.
72.
73. int_v24 a1;
74. int_v24 b1;
75. //int b1,c1,d1;
```

```
76. //int_v24 rgb_buffer[RGB_D];
77. int buffer[3];
78. int grey;
79.
80. int s_levelx0,s_levelx1,s_levelx2;
81. int s_levely0,s_levely1;
82. int s_x,s_y;
83.
84. long long R,threshold,th,RX;
85.
86. matrix M_I,M_G;
87. matrix_long M_R,M_S,M_F;
88. gaussian_level1 g_Ix2_level1,g_Iy2_level1,g_Ixy_level1;
89. gaussian_level2 g_Ix2_level2,g_Iy2_level2,g_Ixy_level2;
90.
91. long long x2,y2,xy,x2_y2;
92.
93. long long result_x2y2,result_xy2,result_xy22;
94.
95. long long R_line0[N],R_line1[N],R_line2[N];
96.
97. matrix line0[N], line1[N], line2[N], line3[N],line4[N];
98.
99. unsigned short int grey_line0[N],grey_line1[N],grey_line2[N];
100.
101. unsigned short int final_result;
102.
103.
104. while(!end_of_packet) {
105.     // Blocking read from the input stream
106.     a1 = a.read(start_of_packet, end_of_packet);
107.     #pragma unroll
108.     for(int i=0;i < 3; i++)
109.     {
110.         buffer[i]=a1.data[i];
111.     }
112.     //grey result
113.
114.     grey=(buffer[0]*76+buffer[1]*150+buffer[2]*29)>>8;
115.
116.
```

```
117.         #pragma unroll
118.         for (int i = N - 1; i > 0; --i) {
119.             grey_line0[i] = grey_line0[i-1];
120.             grey_line1[i] = grey_line1[i-1];
121.             grey_line2[i] = grey_line2[i-1];
122.         }
123.         grey_line0[0] = grey;
124.         grey_line1[0] = grey_line0[N-1];
125.         grey_line2[0] = grey_line1[N-1];
126.
127.
128.
129.
130.
131.
132.         //Sobel filter
133.         s_levelx0=grey_line0[N-1] +(grey_line1[N-1]*2) +grey_line2[N-
134.         1];
135.         s_levelx1=grey_line0[N-3] +(grey_line1[N-3]*2) +grey_line2[N-
136.         3];
137.         s_levely0=grey_line0[N-1]+(grey_line0[N-2]*2)+grey_line0[N-
138.         3];
139.         s_levely1=grey_line2[N-1]+(grey_line2[N-2]*2)+grey_line2[N-
140.         3];
141.
142.         s_x=s_levelx1-s_levelx0;
143.
144.         s_y=s_levely0-s_levely1;
145.
146.         M_I.I_x2=s_x*s_x;
147.         M_I.I_y2=s_y*s_y;
148.         M_I.I_xy=s_x*s_y;
149.
150.         // 5 buffered lines
151.         #pragma unroll
152.         for (int i = N - 1; i > 0; --i) {
153.             line0[i] = line0[i-1];
154.             line1[i] = line1[i-1];
155.             line2[i] = line2[i-1];
```

```

154.         line3[i] = line3[i-1];
155.         line4[i] = line4[i-1];
156.     }
157.     line0[0] = M_I;
158.     line1[0] = line0[N-1];
159.     line2[0] = line1[N-1];
160.     line3[0] = line2[N-1];
161.     line4[0] = line3[N-1];
162.
163.
164.
165.
166.
167.     //Gaussian filter
168.     //x2
169.     g_Ix2_level1.g_levelx0= (line0[N-1].I_x2 + line4[N-
170.     1].I_x2 + line0[N-5].I_x2 + line4[N-5].I_x2)<<1;
171.     g_Iy2_level1.g_levelx0= (line0[N-1].I_y2 + line4[N-
172.     1].I_y2 + line0[N-5].I_y2 + line4[N-5].I_y2)<<1;
173.     g_Ixy_level1.g_levelx0= (line0[N-1].I_xy + line4[N-
174.     1].I_xy + line0[N-5].I_xy + line4[N-5].I_xy)<<1;
175.
176.     //x4
177.     g_Ix2_level1.g_levelx1= (line1[N-1].I_x2 + line3[N-
178.     1].I_x2 + line0[N-2].I_x2 + line4[N-2].I_x2)<<2;
179.     g_Ix2_level1.g_levelx2= (line0[N-4].I_x2 + line4[N-
180.     4].I_x2 + line1[N-5].I_x2 + line3[N-5].I_x2)<<2;
181.     g_Iy2_level1.g_levelx1= (line1[N-1].I_y2 + line3[N-
182.     1].I_y2 + line0[N-2].I_y2 + line4[N-2].I_y2)<<2;
183.     g_Iy2_level1.g_levelx2= (line0[N-4].I_y2 + line4[N-
184.     4].I_y2 + line1[N-5].I_y2 + line3[N-5].I_y2)<<2;
185.     g_Ixy_level1.g_levelx1= (line1[N-1].I_xy + line3[N-
186.     1].I_xy + line0[N-2].I_xy + line4[N-2].I_xy)<<2;
187.     g_Ixy_level1.g_levelx2= (line0[N-4].I_xy + line4[N-
188.     4].I_xy + line1[N-5].I_xy + line3[N-5].I_xy)<<2;
189.
190.     //x5
191.     g_Ix2_level1.g_levelx3= line2[N-1].I_x2 + line0[N-
192.     3].I_x2 + line4[N-3].I_x2 + line2[N-5].I_x2;
193.     g_Iy2_level1.g_levelx3= line2[N-1].I_y2 + line0[N-
194.     3].I_y2 + line4[N-3].I_y2 + line2[N-5].I_y2;

```

```

183.         g_Ixy_level1.g_levelx3= line2[N-1].I_xy + line0[N-
           3].I_xy + line4[N-3].I_xy + line2[N-5].I_xy;
184.         //x9
185.         g_Ix2_level1.g_levelx4= line1[N-2].I_x2 + line3[N-
           2].I_x2 + line1[N-4].I_x2 + line3[N-4].I_x2;
186.         g_Iy2_level1.g_levelx4= line1[N-2].I_y2 + line3[N-
           2].I_y2 + line1[N-4].I_y2 + line3[N-4].I_y2;
187.         g_Ixy_level1.g_levelx4= line1[N-2].I_xy + line3[N-
           2].I_xy + line1[N-4].I_xy + line3[N-4].I_xy;
188.
189.         //x12
190.         g_Ix2_level1.g_levelx5= line2[N-2].I_x2 + line1[N-
           3].I_x2 + line3[N-3].I_x2 + line2[N-4].I_x2;
191.         g_Iy2_level1.g_levelx5= line2[N-2].I_y2 + line1[N-
           3].I_y2 + line3[N-3].I_y2 + line2[N-4].I_y2;
192.         g_Ixy_level1.g_levelx5= line2[N-2].I_xy + line1[N-
           3].I_xy + line3[N-3].I_xy + line2[N-4].I_xy;
193.
194.         //x15
195.
196.         g_Ix2_level1.g_levelx6= (line2[N-3].I_x2<<4)-line2[N-3].I_x2;
197.         g_Iy2_level1.g_levelx6= (line2[N-3].I_y2<<4)-line2[N-3].I_y2;
198.         g_Ixy_level1.g_levelx6= (line2[N-3].I_xy<<4)-line2[N-3].I_xy;
199.
200.
201.
202.
203.         g_Ix2_level2.g_level20= g_Ix2_level1.g_levelx0+g_Ix2_level1.g_l
           evelx6;
204.         g_Iy2_level2.g_level20= g_Iy2_level1.g_levelx0+g_Iy2_level1.g_l
           evelx6;
205.         g_Ixy_level2.g_level20= g_Ixy_level1.g_levelx0+g_Ixy_level1.g_l
           evelx6;
206.
207.         g_Ix2_level2.g_level21= g_Ix2_level1.g_levelx1 + g_Ix2_level1.g
           _levelx2;
208.         g_Iy2_level2.g_level21= g_Iy2_level1.g_levelx1 + g_Iy2_level1.g
           _levelx2;
209.         g_Ixy_level2.g_level21= g_Ixy_level1.g_levelx1 + g_Ixy_level1.g
           _levelx2;
210.

```

```
211.          g_Ix2_level2.g_level22= (g_Ix2_level1.g_levelx3<<2) +g_Ix2_level1.g_levelx3;
212.          g_Iy2_level2.g_level22= (g_Iy2_level1.g_levelx3<<2) +g_Iy2_level1.g_levelx3;
213.          g_Ixy_level2.g_level22= (g_Ixy_level1.g_levelx3<<2) +g_Ixy_level1.g_levelx3;
214.
215.          g_Ix2_level2.g_level23= (g_Ix2_level1.g_levelx4<<3) +g_Ix2_level1.g_levelx4;
216.          g_Iy2_level2.g_level23= (g_Iy2_level1.g_levelx4<<3) +g_Iy2_level1.g_levelx4;
217.          g_Ixy_level2.g_level23= (g_Ixy_level1.g_levelx4<<3) +g_Ixy_level1.g_levelx4;
218.
219.          g_Ix2_level2.g_level24= (g_Ix2_level1.g_levelx5<<3) +(g_Ix2_level1.g_levelx5<<2);
220.          g_Iy2_level2.g_level24= (g_Iy2_level1.g_levelx5<<3) +(g_Iy2_level1.g_levelx5<<2);
221.          g_Ixy_level2.g_level24= (g_Ixy_level1.g_levelx5<<3) +(g_Ixy_level1.g_levelx5<<2);
222.
223.
224.          M_G.I_x2 =      (g_Ix2_level2.g_level20+g_Ix2_level2.g_level21+
          g_Ix2_level2.g_level22 +g_Ix2_level2.g_level23+g_Ix2_level2.g_level24)/159;
225.
226.          M_G.I_y2 =      (g_Iy2_level2.g_level20+g_Iy2_level2.g_level21+
          g_Iy2_level2.g_level22 +g_Iy2_level2.g_level23+g_Iy2_level2.g_level24)/159;
227.
228.          M_G.I_xy =      (g_Ixy_level2.g_level20+g_Ixy_level2.g_level21+
          g_Ixy_level2.g_level22 +g_Ixy_level2.g_level23+g_Ixy_level2.g_level24)/159;
229.
230.          x2=M_G.I_x2;
231.          y2=M_G.I_y2;
232.          xy=M_G.I_xy;
233.          x2_y2=M_G.I_x2+M_G.I_y2;
234.
235.
236.          result_x2y2 =   x2*y2;
```

```
237.         result_xy2 = xy*xy;
238.         result_xy22 = x2_y2*x2_y2;
239.
240.         R=(result_x2y2-result_xy2)-(result_xy22>>5);
241.
242.         RX=(R>0)?R:0;
243.
244.
245.         if(threshold<R)
246.         {
247.             threshold = R;
248.             //th=threshold>>3;
249.         }
250.         else{
251.             threshold=threshold;
252.         }
253.         th=threshold>>4;
254.         //th=0;//2200000000;
255.
256.
257.         #pragma unroll
258.         for (int i = N - 1; i > 0; --i) {
259.             R_line0[i] = R_line0[i-1];
260.             R_line1[i] = R_line1[i-1];
261.             R_line2[i] = R_line2[i-1];
262.         }
263.         R_line0[0] = RX;
264.         R_line1[0] = R_line0[N-1];
265.         R_line2[0] = R_line1[N-1];
266.
267.         if(
268.             (R_line1[N-2]>th) &&
269.             (R_line1[N-2] > R_line0[N-3]) &&
270.             (R_line1[N-2] > R_line0[N-1]) &&
271.             (R_line1[N-2] > R_line2[N-1]) &&
272.             (R_line1[N-2] > R_line2[N-3])
273.
274.         ){
275.
276.             final_result=255;
277.         }
```

```
278.         else
279.         {
280.             final_result=0;}
281.
282.
283.
284.
285.             b1.data[0]=final_result;
286.             b1.data[1]=final_result;
287.             b1.data[2]=final_result;
288.             b.write(b1, start_of_packet, end_of_packet);
289.
290.
291.         }
292.     }
293.
294.
295. int main (void) {
296.
297.     bool pass = true;
298.
299.
300.
301.     if (pass) {
302.         printf("PASSED\n");
303.     } else {
304.         printf("FAILED\n");
305.     }
306.
307.     return 0;
308. }
```

## Appendix E Customized “Good feature to track”

### OpenCV Code

```
1. #define soc_cv_av
2. #include <iostream>
3. #include <stdio.h>
4. #include <unistd.h>
5. #include <fcntl.h>
6. #include <sys/mman.h>
7. #include <sys/types.h>
8. #include <inttypes.h>
9. #include <memory.h>
10. #include <pthread.h>
11. #include <stdlib.h>
12. #include <time.h>
13. #include <sys/time.h>
14. #include "/home/shaonan/intelFPGA/17.0/embedded/ip/altera/hps/altera_hps/hwlib/include/hwlib.h"
15. #include "/home/shaonan/intelFPGA/17.0/embedded/ip/altera/hps/altera_hps/hwlib/include/soc_cv_av/socal/socal.h"
16. #include "/home/shaonan/intelFPGA/17.0/embedded/ip/altera/hps/altera_hps/hwlib/include/soc_cv_av/socal/hps.h"
17. #include "/home/shaonan/intelFPGA/17.0/embedded/ip/altera/hps/altera_hps/hwlib/include/soc_cv_av/socal/alt_gpio.h"
18. #include "hps_0.h"
19.
20. #include "math.h"
21.
22. #include "opencv2/imgproc.hpp"
23. #include "opencv2/imgcodecs.hpp"
24. #include "opencv2/objdetect.hpp"
25.
26. #define HW_REGS_BASE ( ALT_STM_OFST )
27. #define HW_REGS_SPAN ( 0x04000000 )
28. #define HW_REGS_MASK ( HW_REGS_SPAN - 1 )
29.
30.
31.
32. #define ALT_AXI_FPGASLVS_OFST (0xC0000000) // axi_master
```

```
33. #define HW_FPGA_AXI_SPAN (0x4000000) // Bridge span
34. #define HW_FPGA_AXI_MASK ( HW_FPGA_AXI_SPAN - 1 )
35. //1024x768
36.
37. #define Buffer0 ( 0x04000000 )
38. #define Buffer1 ( 0x04600000 )
39. #define Buffer2 ( 0x04C00000 )
40.
41. #define mapped_Buffer0 ( 0x06000000 )
42. #define mapped_Buffer1 ( 0x06600000 )
43. #define mapped_Buffer2 ( 0x06C00000 )
44.
45. //1024x768
46.
47. #define Reader1_Buffer0 ( 0x12000000 )
48. #define Reader1_Buffer1 ( 0x12240000 )
49. #define Reader1_Buffer2 ( 0x12480000 )
50.
51.
52. #define Buffer_r0 ( 0x10000000 )
53. #define Buffer_r1 ( 0x10240000 )
54. #define Buffer_r2 ( 0x10480000 )
55.
56. //640
57. /*
58. #define Buffer0 ( 0x04000000 )
59. #define Buffer1 ( 0x04258000 )
60. #define Buffer2 ( 0x044B0000 )
61.
62. #define mapped_Buffer0 ( 0x06000000 )
63. #define mapped_Buffer1 ( 0x06258000 )
64. #define mapped_Buffer2 ( 0x064B0000 )
65.
66. //640
67. #define Reader1_Buffer0 ( 0x12000000 )
68. #define Reader1_Buffer1 ( 0x120E1000 )
69. #define Reader1_Buffer2 ( 0x121C2000 )
70.
71.
72. #define Buffer_r0 ( 0x10000000 )
73. #define Buffer_r1 ( 0x100E1000 )
```

```
74. #define Buffer_r2 ( 0x101C2000 )
75. */
76. /*
77. //800
78. #define Buffer0 ( 0x04000000 )
79. #define Buffer1 ( 0x043A9800 )
80. #define Buffer2 ( 0x04753000 )
81.
82. #define mapped_Buffer0 ( 0x06000000 )
83. #define mapped_Buffer1 ( 0x063A9800 )
84. #define mapped_Buffer2 ( 0x06753000 )
85.
86.
87. //800
88. #define Reader1_Buffer0 ( 0x12000000 )
89. #define Reader1_Buffer1 ( 0x1215F900 )
90. #define Reader1_Buffer2 ( 0x122BF200 )
91.
92.
93. #define Buffer_r0 ( 0x10000000 )
94. #define Buffer_r1 ( 0x1015F900 )
95. #define Buffer_r2 ( 0x102BF200 )
96. */
97. //1024x768
98.
99. #define WIDTH      (1024)
100. #define HEIGHT     (768)
101. //640
102. //#define WIDTH     (640)
103. //#define HEIGHT    (480)
104.
105. //800
106. //#define WIDTH     (800)
107. //#define HEIGHT    (600)
108. #define resolution ( WIDTH*HEIGHT )
109.
110.
111. #define buffer_size0 ( resolution*3*8 )
112. #define buffer_size ( resolution*3*3 )
113.
114. #define frame_size ( resolution*3 )
```

```
115. #define page_size ( resolution*8 )
116.
117. #define processing_area ( resolution )
118. using namespace cv;
119. using namespace std;
120.
121.
122. static volatile unsigned int *frame_reader_addr=NULL;
123. static volatile unsigned int *frame_writer_addr=NULL;
124.
125. static volatile unsigned int *frame_reader_addr_base=NULL;
126. static volatile unsigned int *frame_writer_addr_base=NULL;
127.
128.
129.
130.
131.     Mat img(HEIGHT,WIDTH,CV_16UC1);
132.
133.     Mat eig(HEIGHT,WIDTH,CV_32FC1);
134.
135.     Mat frame(HEIGHT,WIDTH,CV_8UC3);
136.     Mat frame_gray(HEIGHT,WIDTH,CV_8UC1);
137.     Mat dst(HEIGHT,WIDTH,CV_8UC3);
138.     Mat edge1,dst_scaled;
139.     RNG rng(12345);
140.
141.
142.
143.     int blockSize = 2;
144.     int apertureSize = 3;
145.     double k = 0.04;
146. int thresh = 200;
147. int max_thresh = 255;
148.
149. struct greaterThanPtr :
150.     public std::binary_function<const float *, const float *, bool>
151. {
152.     bool operator () (const float * a, const float * b) const
153.     // Ensure a fully deterministic result of the sort
154.     { return (*a > *b) ? true : (*a < *b) ? false : (a > b); }
155. };
```

```
156.
157. ///////////////////////////////////////////////////////////////////
158. // VIP Frame Buffer(writer/reader): configure
159.
160. void    frame_reader_conf(){
161.     frame_reader_addr[0]=0x01;
162.
163.     frame_reader_addr[5]=0x800300; //1024
164.     //frame_reader_addr[5]=0x5001E0; //640
165.     //frame_reader_addr[5]=0x640258; //800
166.     frame_reader_addr[6]=Buffer_r0;
167. }
168.
169. void    frame_writer_conf(){
170.
171.
172.     frame_writer_addr[0]=0x01;
173.     frame_writer_addr[8]=0x00010001;
174.
175. }
176.
177.
178.
179.
180.
181.
182.
183.
184. void goodfeature( OutputArray _corners,
185.                  int maxCorners, double minDistance,
186.                  InputArray _mask
187.                  )
188. {
189.
190.
191.     Mat tmp;
192.
193.
194.     if (frame_gray.empty()||eig.empty())
195.     {
196.         _corners.release();
```

```
197.     return;
198. }
199.
200.
201. dilate( eig, tmp, Mat());
202.
203. Size imgsize = frame_gray.size();
204. std::vector<const float*> tmpCorners;
205.
206. // collect list of pointers to features - put them into temporary image
207. Mat mask = _mask.getMat();
208. for( int y = 1; y < imgsize.height - 1; y++ )
209. {
210.     const float* eig_data = (const float*)eig.ptr(y);
211.     const float* tmp_data = (const float*)tmp.ptr(y);
212.     const uchar* mask_data = mask.data ? mask.ptr(y) : 0;
213.
214.     for( int x = 1; x < imgsize.width - 1; x++ )
215.     {
216.         float val = eig_data[x];
217.         if( val != 0 && val == tmp_data[x] && (!mask_data || mask_data[
x]) )
218.             tmpCorners.push_back(eig_data + x);
219.     }
220. }
221.
222. std::vector<Point2f> corners;
223. size_t i, j, total = tmpCorners.size(), ncorners = 0;
224.
225. cout<<"** Total of corners detected: "<<total<<endl;
226.
227. if (total == 0)
228. {
229.     _corners.release();
230.     return;
231. }
232.
233. std::sort( tmpCorners.begin(), tmpCorners.end(), greaterThanPtr() );
234.
235. if (minDistance >= 1)
```

```
236.     {
237.         // Partition the image into larger grids
238.         int w = frame_gray.cols;
239.         int h = frame_gray.rows;
240.
241.         const int cell_size = cvRound(minDistance);
242.         const int grid_width = (w + cell_size - 1) / cell_size;
243.         const int grid_height = (h + cell_size - 1) / cell_size;
244.
245.         std::vector<std::vector<Point2f> > grid(grid_width*grid_height);
246.
247.         minDistance *= minDistance;
248.
249.         for( i = 0; i < total; i++ )
250.         {
251.             int ofs = (int)((const uchar*)tmpCorners[i] - eig.ptr());
252.             int y = (int)(ofs / eig.step);
253.             int x = (int)((ofs - y*eig.step)/sizeof(float));
254.
255.             bool good = true;
256.
257.             int x_cell = x / cell_size;
258.             int y_cell = y / cell_size;
259.
260.             int x1 = x_cell - 1;
261.             int y1 = y_cell - 1;
262.             int x2 = x_cell + 1;
263.             int y2 = y_cell + 1;
264.
265.             // boundary check
266.             x1 = std::max(0, x1);
267.             y1 = std::max(0, y1);
268.             x2 = std::min(grid_width-1, x2);
269.             y2 = std::min(grid_height-1, y2);
270.
271.             for( int yy = y1; yy <= y2; yy++ )
272.             {
273.                 for( int xx = x1; xx <= x2; xx++ )
274.                 {
275.                     std::vector <Point2f> &m = grid[yy*grid_width + xx];
276.
```

```
277.         if( m.size() )
278.         {
279.             for(j = 0; j < m.size(); j++)
280.             {
281.                 float dx = x - m[j].x;
282.                 float dy = y - m[j].y;
283.
284.                 if( dx*dx + dy*dy < minDistance )
285.                 {
286.                     good = false;
287.                     goto break_out;
288.                 }
289.             }
290.         }
291.     }
292. }
293.
294.     break_out:
295.
296.     if (good)
297.     {
298.         grid[y_cell*grid_width + x_cell].push_back(Point2f((float)x
299. , (float)y));
300.
301.         corners.push_back(Point2f((float)x, (float)y));
302.         ++ncorners;
303.
304.         if( maxCorners > 0 && (int)ncorners == maxCorners )
305.             break;
306.     }
307. }
308. else
309. {
310.     for( i = 0; i < total; i++ )
311.     {
312.         int ofs = (int)((const uchar*)tmpCorners[i] - eig.ptr());
313.         int y = (int)(ofs / eig.step);
314.         int x = (int)((ofs - y*eig.step)/sizeof(float));
315.
316.         corners.push_back(Point2f((float)x, (float)y));
```

```
317.         ++ncorners;
318.         if( maxCorners > 0 && (int)ncorners == maxCorners )
319.             break;
320.     }
321. }
322.
323.     Mat(corners).convertTo(_corners, _corners.fixedType() ? _corners.type()
    : CV_32F);
324. }
325.
326.
327.
328. void goodFeaturesToTrack_Demo( )
329. {
330.     int maxCorners = 20;
331.     //if( maxCorners < 1 ) { maxCorners = 1; }
332.
333.     /// Parameters for Shi-Tomasi algorithm
334.     vector<Point2f> corners;
335.     double minDistance = 10;
336.
337.
338.     /// Copy the source image
339.
340.
341.     /// Apply corner detection
342.     goodfeature( corners,
343.                 maxCorners,
344.                 minDistance,
345.                 Mat()
346.                 );
347.
348.
349.     /// Draw corners detected
350.     cout<<"** Number of corners detected: "<<corners.size()<<endl;
351.     int r = 6;
352.     for( size_t i = 0; i < corners.size(); i++ )
353.         { circle( dst, corners[i], r, Scalar(rng.uniform(0,255), rng.uniform(0
    ,255), rng.uniform(0,255)), -1, 8, 0 ); }
354.
355.     //
```

```
356.
357.  /// Set the need parameters to find the refined corners
358.  Size winSize = Size( 5, 5 );
359.  Size zeroZone = Size( -1, -1 );
360.  TermCriteria criteria = TermCriteria( TermCriteria::EPS + TermCriteria::C
    OUNT, 40, 0.001 );
361.
362.  /// Calculate the refined corner locations
363.  cornerSubPix( frame_gray, corners, winSize, zeroZone, criteria );
364.
365.  /// Write them down
366.  for( size_t i = 0; i < corners.size(); i++ )
367.  { cout<<" -
    - Refined Corner ["<<i<<" ("<<corners[i].x<<","<<corners[i].y<<)"<<endl;
    }
368. }
369.
370.
371.
372. static long get_tick_count(void)
373. {
374.     struct timespec now;
375.     clock_gettime(CLOCK_MONOTONIC, &now);
376.     //return now.tv_sec*1000000 + now.tv_nsec/1000;
377.     return now.tv_sec*1000000 + now.tv_nsec/1000;
378. }
379.
380.
381. void processing(uint8_t *ptr,uint8_t *ptr1,uint8_t *ptr2,uint16_t *ptr3)
382. {
383.     uint8_t *data0,*data1,*data2;
384.     uint16_t *ptrx;
385.     uint16_t *data3;
386.     int i,j=0,k=0,l=0,m=0;
387.     data0=ptr;
388.     data1=ptr1;
389.     data2=ptr2;
390.     data3=ptr3;
391.     ptrx=(uint16_t *)ptr;
392.     for(i=0;i<=page_size;i+=8){
393.         data1[j]=data0[i];
```

```
394.     data1[j+1]=data0[i+1];
395.     data1[j+2]=data0[i+2];
396.
397.     data2[k]=data0[i+3];
398.
399.     data3[k]=ptrx[l+2];
400.     j+=3;
401.     k++;
402.     l+=4;
403. }
404.
405.
406. }
407.
408.
409.
410. void* read_data(){
411.
412.     //int edgeThresh = 20;
413.     //int edgeThreshSchar=1;
414.
415.     uint8_t *data0,*data1,*data2;
416.     uint16_t *data3;
417.     //unsigned char *data_base=NULL;
418.
419.     uint32_t time_start,time_elapsed;
420.     data0=(uint8_t *)malloc(page_size);
421.     data1=(uint8_t *)malloc(frame_size);
422.     data2=(uint8_t *)malloc(resolution);
423.     data3=(uint16_t *)malloc(resolution);
424.     //data_base=data0;
425.     time_start = get_tick_count();
426.     while(1)
427.     {
428.
429.         while (frame_writer_addr[5]<0x80000000){};
430.
431.
432.         printf("frame_writer_addr[6]=0x%X\n",frame_writer_addr[6]);
433.
434.
```

```
435.     switch(frame_writer_addr[6]){
436.         case Buffer0:
437.             memcpy((void*)data0,(void*)mapped_Buffer0,page_size);
438.             break;
439.         case Buffer1:
440.             memcpy((void*)data0,(void*)mapped_Buffer1,page_size);
441.             break;
442.         case Buffer2:
443.             memcpy((void*)data0,(void*)mapped_Buffer2,page_size);
444.             break;
445.         default:
446.             break;
447.     }
448.     frame_writer_addr[8]=0x00010001;
449.
450.     processing(data0,data1,data2,data3);
451.
452.
453.     /*1024
454.         Mat frame(768,1024,CV_8UC3,data1);
455.         Mat frame_gray(768,1024,CV_8UC1,data2);
456.         Mat img(768,1024,CV_16UC1,data3);
457.     */
458.     Mat frame(HEIGHT,WIDTH,CV_8UC3,data1);
459.
460.
461.     //Mat frame_gray(HEIGHT,WIDTH,CV_8UC1,data2);
462.     //Mat img(HEIGHT,WIDTH,CV_16UC1,data3);
463.
464.     //img.convertTo(eig, CV_32FC1,1,0);
465.     cvtColor( frame, frame_gray, COLOR_RGB2GRAY );
466.     frame.copyTo(dst);
467.     goodFeaturesToTrack_Demo();
468.     //imwrite("opencv.jpg",dst);
469.
470.
471.
472.     while(frame_reader_addr[7]<0x04000000){};
473.
474.
475.
```

```
476.     memcpy((void *)Reader1_Buffer0 , dst.ptr(), frame_size);
477.     //memcpy((void *)Reader1_Buffer0 , (void *)data1, frame_size);
478.     frame_reader_addr[5]=0x800300; //1024
479.     //frame_reader_addr[5]=0x5001E0; //640
480.     //frame_reader_addr[5]=0x640258; //800
481.
482.     frame_reader_addr[6]=Buffer_r0 ;
483.
484.     time_elapsed = get_tick_count() - time_start;
485.     if (time_elapsed)
486.     printf("time_elapsed whole processing  =%.5f s\r\n", (float)time
    e_elapsed/1000000);
487.
488.     time_start = get_tick_count();
489.
490.     //printf("frame writer counter=%d\n",frame_writer_addr[3]);
491.     //if (frame_writer_addr[3]<30){
492.     //imwrite( "opencv.jpg", dst );
493.     //imwrite( "rgb.jpg")
494.     //}
495.
496. }
497. free(data0);
498.
499. }
500.
501.
502.
503.
504. int main(int argc,char ** argv)
505. {
506.
507.
508.
509.
510.
511.     void *lw_axi_virtual_base=NULL;
512.     void *virtual_base=NULL;
513.     int fd;
514.
515.
```

```
516.
517.     if( ( fd = open( "/dev/mem", ( O_RDWR | O_SYNC ) ) ) == -1 )
518.         {
519.             printf( "ERROR: could not open \"/dev/mem\"...\n" );
520.             return( 1 );
521.         }
522.
523.
524.
525.
526.     printf("Memory mapped succeeded\n");
527.
528.     lw_axi_virtual_base = mmap( NULL, HW_REGS_SPAN, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, HW_REGS_BASE );
529.     printf("lw_virtual_base=0x%X\n",lw_axi_virtual_base);
530.     if( lw_axi_virtual_base == MAP_FAILED ) {
531.         printf( "ERROR: mmap() failed...\n" );
532.         close( fd );
533.         return( 1 );
534.     }
535.
536.
537.     //IP control
538.     frame_writer_addr= ( unsigned int *)((uint8_t *)lw_axi_virtual_base + (
        ( ALT_LWFGASLVS_OFST + ALT_VIP_CL_VFB_3_BASE ) & ( HW_REGS_MASK ) ));
539.     frame_reader_addr= ( unsigned int *)((uint8_t *)lw_axi_virtual_base +
        ( ( ALT_LWFGASLVS_OFST + ALT_VIP_CL_VFB_0_BASE ) & ( HW_REGS_MASK ) ));
540.
541.
542.
543.     frame_writer_conf();
544.     printf("writer configure succeeded\n");
545.
546.     frame_reader_conf();
547.
548.     printf("reader configure succeeded\n");
549.
550.
551.     frame_writer_addr_base=( unsigned int *) mmap( (void *)mapped_Buffer0 ,
        buffer_size0, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, Buffer0 );
552.     if( frame_writer_addr_base == MAP_FAILED ) {
```

```
553.     printf( "ERROR: mmap() failed...\n" );
554.     close( fd );
555.     return( 1 );
556. }
557.
558.     frame_reader_addr_base= ( unsigned int *) mmap( (void *)Reader1_Buffer0
, buffer_size, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, Buffer_r0 );
559.     if( frame_reader_addr_base == MAP_FAILED ) {
560.         printf( "ERROR: mmap() failed...\n" );
561.         close( fd );
562.         return( 1 );
563.     }
564.
565.
566.     printf("frame_reader_addr_base=0x%X\n",frame_reader_addr_base);
567.     printf("frame_writer_addr_base=0x%X\n",frame_writer_addr_base);
568.
569.
570.
571.     read_data();
572.
573.
574.
575.     close( fd );
576.
577.
578.
579.     return 0;
580. }
```