EMBEDDED MACHINE VISION

– A PARALLEL ARCHITECTURE APPROACH –

CHAN KIT WAI

NATIONAL UNIVERSITY OF SINGAPORE

2005

EMBEDDED MACHINE VISION

– A PARALLEL ARCHITECTURE APPROACH –

CHAN KIT WAI

(B.Tech.(Hons), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF ENGINEERING

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2005

Acknowledgements

First of all, I would like to thank my project supervisor, Dr Prahlad Vadakkepat for his help and guidance in writing this thesis. For that, he has spent his precious time guiding me for making this thesis readable. I would also like to express my gratitude for his advice and the freedom that he had given, to explore the areas of my interest.

I would also like to thank those who had gave their technical advice and time for answering numerous questions. In particular, Dr Tang Kok Zuea, Boon Kiat and Dr Wang.

Special thanks, goes to my wife for giving her unlimited support in many ways; especially working through late nights for the preparation of this thesis. Her understanding and encouragement are important during this demanding period of my career and studies.

Jason Chan Kit Wai

Nov 2005

Contents

Acknowledgements **ii**

Contents **iii**

Summary **vii**

List of Tables **ix**

List of Figures **x**

List of Abbreviations **xiv**

1 Introduction **1**

1.1 Vision System For Mobile Robots 1

1.2 Different Architectures for Image Processing 4

1.2.1 Microprocessors 5

1.2.2 DSP Processors 6

1.2.3 Application Specific Integrated Circuit 7

1.2.4 Reconfigurable Architecture 7

1.3 Data Processing at Different Level 9

1.4 Motivation and Contribution 11

1.5 Thesis Outline 12

2 System Level Architecture Design 13

2.1 System Components Studies 13

2.1.1 Image Sensors 14

2.1.2 Memories 17

2.1.3 FPGA Development Board 17

2.2 Simulation and Development Tools 18

2.2.1 Programming Tools 19

2.2.2 FPGA Design Flow 19

2.2.3 Verilog vs VHDL 22

2.3 Image Representation 24

3 An Analytic Model for Embedded Machine Vision 28

3.1 Introduction 28

3.2 Analytic Model to Determine Image Buffer Size 29

3.2.1 Concept of Queuing Theory 29

3.2.2 Row buffering 31

3.3 Analytic Model to Determine Computational Speed 34

3.4 Analysis of Image Segmentation Algorithm 36

3.4.1 Computation using microprocessor 37

3.4.2 Computation using custom architecture 39

3.5 Analysis of Image Convolution Algorithm 40

3.6 Summary 42

4 Image Acquisition, Compression, Buffering and Convolution 43

4.1 Image Acquisition 44

4.1.1 Image sensor interface signals 44

4.1.2 Image acquisition: implementation 46

4.2 Image Compression 49

4.2.1 Image compression: concept 49

4.2.2 Image compression: implementation 52

4.3 Image Buffering 56

4.3.1 Image buffering: theory 56

4.3.2 Image buffering: implementation 60

4.4 Convolution Theory 65

5 FPGA Implementation of Parallel Architecture 67

5.1 Edge Detection Theory 68

5.2 Proposed Parallel Architecture for Edge Detection 71

5.3 Thresholding 75

5.4 Edge Detection: Analysis and Results 76

5.4.1 Experiment of edge detection with different scenes 80

5.4.2 Images with resolution 320 x 240 80

5.4.3 Image with resolution of 1280 x 1024 81

5.5 Proposal Parallel Architecture for Low Pass Filter 83
5.5.1 Noise pixels in high resolution image 83
5.5.2 Low Pass Filter 84
5.6 System Resource Utilization 88
5.6.1 On-Chip memory size requirements 88
5.6.2 Logic resources 89
5.6.3 System performance 89
5.7 Summary of Results 90

6 Conclusions and Future Work 92
6.1 Conclusions 92
6.2 Future Work 93

Bibliography 95

Author's Publications 101

Summary

Machine vision is one of the essential sensory functions in mobile robotics. By applying vision processing techniques, certain features can be extracted from a given scene. However, there are certain limitations in implementing an on-board image processor. Limited computational power, low data transfer rate and tight memory budget, place constraints on the performance. As a result, image resolution and frame rate are often compromised.

To implement efficient solutions, algorithms and hardware architectures must be well matched. This can be achieved for algorithms with high degree of regularity that are identified to exploit its parallelism. The operations can be mapped into custom functional units to achieve higher performance compared to the fixed processing units. Such approaches can eliminate the necessity of employing high-end processors.

Reconfigurable architectures pose as a suitable platform for computationally demanding image processing algorithms. Custom logic can be designed to exploit parallelism at different areas and levels of an application.

Suitable image sensor, FPGA IC Chip and the suitable simulation and developmental tools are selected. An analytical mathematical model to estimate the various performance parameters associated real-time image processing is proposed. The model allows system designers to estimate the required memory size and processing frequency of a given microprocessor architecture. In one of the examples, the reduction in the number of instructions per pixel, resulted into processing a

pixel in a single cycle. Next, the image acquisition, compression, buffering and image convolution are studied. Custom architectures are designed with the considerations of optimising the logic and memory resources. The image buffering is modelled as a producer-consumer problem. Techniques are employed to reuse memory locations. Data that reaches the end of its lifetime is automatically removed to free up the memory location for new data.

A parallel architecture is proposed to the perform 2D convolution operation with the aim of processing a pixel within a single clock cycle. The customized architecture allows direct computation instead of conventional load store operations. Specifically, the low pass filter, edge detection and thresholding algorithm are investigated. For edge detection, two separate 2D convolution processes and a thresholding process are computed within a single clock cycle. A study is conducted to evaluate the effects of adding a low pass filter to the design. After which, a threshold operation is performed to extract the desired edge features of an image. Two types of image processing, with and without low pass filter are compared.

To achieve minimal usage of hardware resources, the redundant memory locations, logics and computations are removed. For instance, the multipliers are replaced by an equivalent bit-wise shifter and a 9 pixels convolution is reduced to a 6 pixels convolution.

The synthesis results obtained are very encouraging. The total number of slices occupied by the design is 5% of the total hardware resource available. Lastly, simulation and actual hardware implementation are provided to demonstrate the performance of the embedded machine vision using FPGA.

List of Tables

1.1 Specifications of various commercially available on-board vision processors . . . 3

2.1 Comparison of available on-board vision processor . . . 15

2.2 Development and analysis tools . . . 19

2.3 Comparsion of VHDL and Verilog . . . 23

4.1 Properties of exclusive OR operations . . . 51

List of Figures

1.1 Typical machine vision system 2

1.2 (a) Eyebot (b) CMUCam (c) Khepera Camera Turret [1][13][18] . . 3

1.3 Programmability vs parallelism 5

1.4 Fixed Arithmetic Logic Unit (ALU) vs Custom ALU 8

1.5 Data processing at different level 9

1.6 Stages for image processing 10

2.1 MicroViz setup configuration 14

2.2 OV7620 Image sensor and FPGA 15

2.3 Timing waveform of pixel data bus [36] 16

2.4 MicroViz Prototype board 18

2.5 FPGA design flow 20

2.6 Gate level netlist 21

2.7 Configuration Logic Block [30] 22

2.8 Colour Space 24

2.9 (a)RGB colour image (b)Greyscale image (c)Binary image 25

2.10 RGB colour space [35] 25

2.11 HSI colour space [35] 26

3.1 Queue model of vision system 30

3.2 Burst time and emptying time 31

3.3 Thresholding 37

3.4 Assembly code representation of C program 38

3.5 Convolution algorithm in C 41

4.1 Image acquisition process 44

4.2 CMOS image sensor array 44

4.3 CMOS image sensor architecture [36] 45

4.4 Timing Diagram of the control signals 47

4.5 Image acquisition block 47

4.6 Synthesized circuit of the image acquisition block 48

4.7 Simulation result of the image acquisition block 49

4.8 Pixel amplitude of a single line 50

4.9 Number of bits to represent compressed pixel 50

4.10 Block diagram of Compression and Decompression 52

4.11 Simulation results 53

4.12 Synthesized circuit of XOR compression module 54

4.13 XOR gate 54

4.14 Histogram of image with low frequency content 55

4.15 Histogram of image with high frequency content 55

4.16 Image buffering stage 56

4.17 A 3 x 3 convolution mask on a 5 x 4 image 57

4.18 Producer and consumer of pixels before transformation 58

4.19 Producer and consumer of pixels after transformation 58

4.20 Buffering using FIFO 59

4.21 Reduction of memory space after data reuse 59

4.22 Convolution window using registers 61

4.23 Image buffer module 61

4.24 Synthesize result of Image buffer (Part 1) 63

4.25 Synthesize result of Image buffer (Part 2) 64

4.26 Image convolution stage 65

4.27 Image convolution 66

5.1 Image processing stage 67

5.2 Image intensity level derivatives 68

5.3 Convolution window 69

5.4 Prewitt operator 69

5.5 Sobel operator 70

5.6 Acquiring nine pixels from image buffering module 71

5.7 Architecture of G_x 72

5.8 Architecture of G_y 73

5.9 Architecture for gradient magnitude and thresholding 73

5.10 Simulation of architecture using Visual C/C++ 75

5.11 Thresholding 76

5.12 Sum of $|Gx|$ and $|Gy|$ component 77

5.13 Detecting edges of the green carpet 78

5.14 Detecting edges of a tennis ball and the boundary lines 79

5.15 Edge detection with image resolution of 320 x 240 80

5.16 Magnified image of Figure13 81

5.17 (a) Original image of 1280 x 1024 produces (b) fine edge pixels . . . 82

5.18 (a) Magnified image of Figure 15 and (b) Edge detection of fine lines 82

5.19 Edge detection with different image resolution 83

5.20 Insertion of Low pass filter before edge detection 84

5.21 Convolution coefficients of Low Pass Filter 84

5.22 Architecture of Low Pass Filter 85

5.23 (a) Orignal image (b) Edge detection without Low Pass filter 86

5.24 (a) Original 1280 x 1024 image (b) Resultant Image applied with Low pass filter 86

5.25 (a) Edge detection without Low Pass filter (b) Edge detection with Low Pass filter 87

5.26 (a) Without Low pass filtering (b) With Low pass filtering 87

5.27 Comparison of image buffer size required for different resolution . . 88

5.28 Synthesis report from Xilinx synthesis tool 89

5.29 Computation time with different resolution 90

List of Abbreviations

ALU	*Arithmetic Logic Unit*
ASICs	*Application Specific Integrated − Circuit*
CCD	*Charge Coupled Device*
CLB	*Configuration Logic Blocks*
CMOS	*Complementary Metal − oxide Semiconductor*
CPU	*Central Processing Unit*
DSP	*Digital Signal Processing*
EDA	*Electronic Design Automation*
EDIF	*Electronic Design Interchange Format*
EE	*Electrical Engineering*
FIFO	*First In First Out*
FPGA	*Field Programmable GateArray*
FPS	*Frames per Second*
HDL	*Hardware Description Language*
HREF	*Horizontal Reference*
HSI	*Hue Saturation Intensity*
I2C	*Inter − IC Connection*
IIC	*Inter IC Connect*
IOBs	*Input Output Blocks*
ISE	*Integrated Software Environment*
JTAG	*Joint Test Access Group*
LUT	*Look − upTables*

MIMD	*Multiple Instruction Multiple Data*
MISD	*Multiple Instruction Single Data*
P&R	*Place and Route*
PCLK	*Pixel Clock*
RAM	*Random Access Memory*
RGB	*Red Green Blue*
SIMD	*Single Instruction Multiple Data*
SISD	*Single Instruction Single Data*
UCF	*User Constraints File*
VHDL	*VHSIC HDL*
VHSIC	*Very High Speed Integrated Circuit*
VSYN	*Vertical Synchronization*

Chapter 1

Introduction

Robot vision is one of the most essential development set by the robotic community at large. Research and development in robot vision has grown dramatically over the past decade. The interest and concerns of image processing for mobile robots can be seen from the vast amount of literature on this subject, including major projects spearheaded in the industry and research institutes. In particular much emphasis is placed on localization and navigation abilities of mobile robots [1][2][12][13].

Machine vision is one of the essential sensory functions for mobile robotics. By applying vision processing techniques, certain features can be extracted from a given scene. These are used to describe the environment. Collectively, such description is necessary for localization and navigation. This forms the basic behavior of any mobile robot and pave the way for the development of intelligence robot.

1.1 Vision System For Mobile Robots

A typical machine vision system consists of a Charge Coupled Device (CCD) camera, a frame grabber and a host computer for the execution of the image processing algorithm.

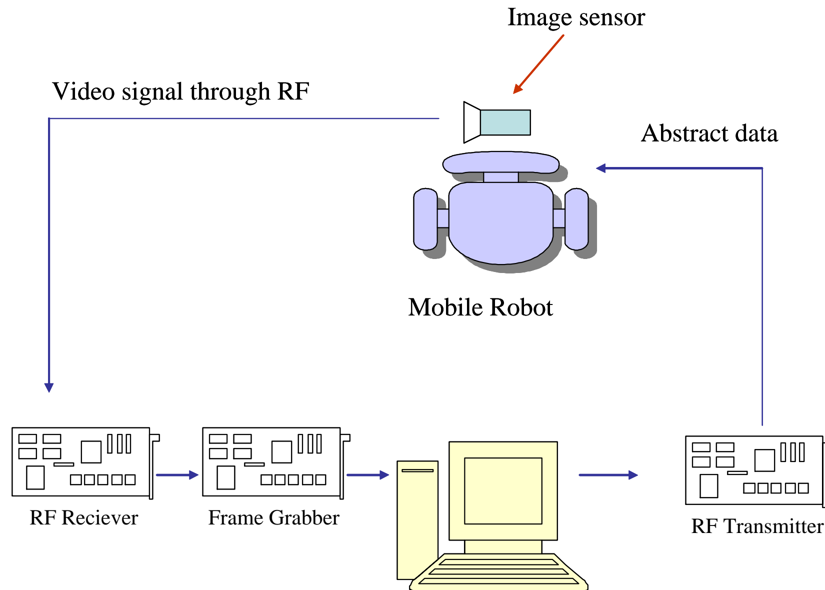


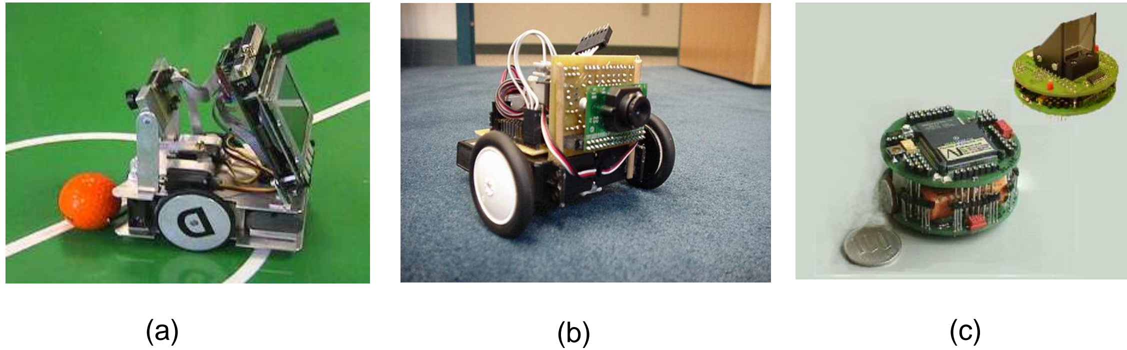
Figure 1.1: Typical machine vision system

A typical image processing system is shown in Figure 1.1. A host computer receives images from a CCD camera, performs image recognition algorithms and transmits control signals to the mobile robot. Such configuration setup is shown in Figure 1.1 is often used in many mobile robotic systems [14][45][16].

A variety of standard image processing tools are supported on a general purpose computer. For instance, some of the commonly used programming libraries and tools are Intel Processing Library, Matlab, Visual C/C++ and Borland C/C++.

However, there are certain limitations that require the processing to be performed on board. The ability to perform on-board processing of real-time images sets many constraints. At many times, limited computational power, low data transfer rate and tight memory budget place constraints on the implementation and performance of the robots. As a result, image resolution and frame rate are often compromised.

A survey is performed to study some of the existing on-board vision systems. The EyeBot, CMUCam1, CMUCam2 and Khepera Camera Turret are reviewed (Figure 1.2). The EyeBot processes an image resolution of 80 x 60 pixels on a



(a) (b) (c)

Figure 1.2: (a) Eyebot (b) CMUCam (c) Khepera Camera Turret [1][13][18]

20MHz processor. The Khepera vision turret is a commercially available vision module exclusively targeted for Khepera miniature mobile robot [13]. It can process a relative high resolution image of up to 160 x 120 pixels. The Camera Turret uses a V6300 digital Complementary Metal-Oxide Semiconductor (CMOS) camera along with a dedicated 32bit Central Processing Unit (CPU) in the turret. Table 1.1 shows the comparison of the various on-board vision processors mentioned.

Table 1.1: Specifications of various commercially available on-board vision processors

Descriptions	CMUCam1	CMUCam2	Eyebot	Khepera vision Turret
CPU	Ubicom sx28	Ubicom Sx52	Motorola 32 bit	Motorola 32bit
CPU speed	75 Mhz	75 Mhz	25 Mhz	-
Max resolution	143 x 80	288 x 160	60 x 80	160 x 120
FPS @ max res	2 to 17 fps	50 fps	10 fps	-
Min resolution	80 x 143	80 x 143	60 x 80	160 x 120
FPS @ min res	30 fps	50 fps	10 fps	-
Memory size	Int. 138 bytes	Ext. FIFO 384K x 8bit	-	SRAM 128K x 8bit
Com port	115200bps	115200bps	-	57600bps

On-board image processor poses certain challenges in the following areas:

Speed: Real-time images are to be computed at high frame rate for closed loop vision control.

Power: The power consumption should be reduced to the minimal for longer battery life. The power consumed by the processor depends on the algorithm, switching frequency (clock frequency) and the switching voltages.

Memory requirements: Vision algorithms often demands more memory compared to other embedded applications. Temporary storages are often used to store image buffers at different stages of the image transformation and analysis. Generally, First-In-First-Out (FIFO) or dual ported Random Access Memory (RAM) are used to buffer the input image for the subsequent processing.

Size constraints: The size of the embedded machine vision should be small enough to fit onto miniature mobile robot.

With the four major constraints specified, it is noted that there is a relationship between all the four constraints. The area of the IC chip is related to the clock speed, amount of memories and logic elements within the die. By lowering the clock speed of the processor, the energy consumption is also reduced accordingly. As a result, this research focuses on the reduction of clock speed and memory requirements for various image processing algorithms.

1.2 Different Architectures for Image Processing

The computational demands associated with high performance image processing has led to several architectures being proposed. Namely, the Microprocessor Architecture, the dedicated Digital Signal Processing (DSP) Processor, Application Specific IC (ASIC) Architecture and the Reconfigurable Architecture. These mentioned architectures are targeted for different types of processing requirements. Figure 1.3 shows the relationship of the different architectures in programmability

vs the data parallelism space [20][19].

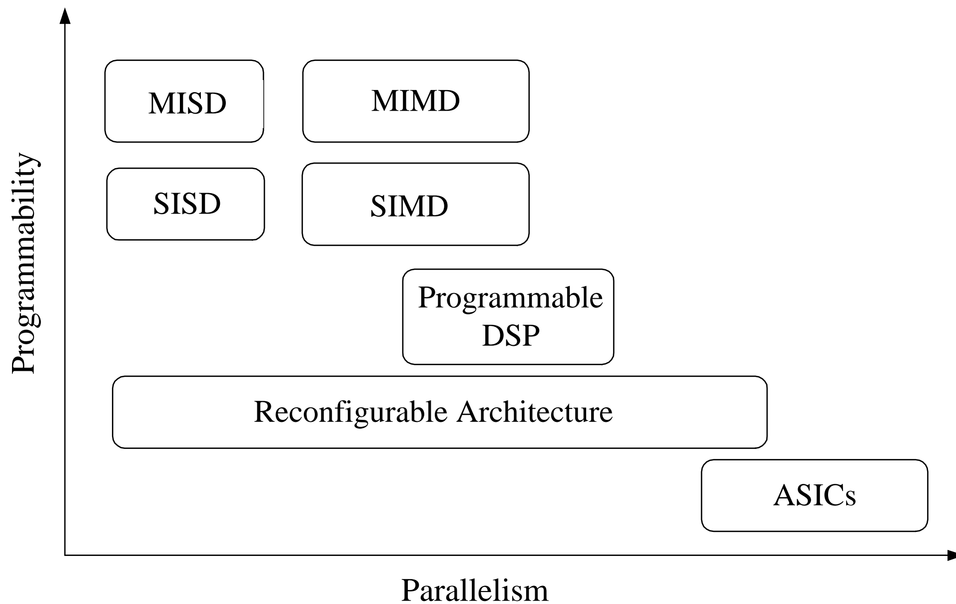


Figure 1.3: Programmability vs parallelism

1.2.1 Microprocessors

The Microprocessor can be further categorised into four different architectures. These have been named in Flynn's classification [20][4] as: Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD) , Multiple Instruction Single Data (MISD) and Multiple Instruction Multiple Data (MIMD). The latter two have generally been used for demanding image processing algorithms.

General purpose computer systems using microprocessor technology are commonly used in the industry. This popular platform provides well established tools and rapid implementation of image processing applications. In addition, the applications are portable to future variants of such system.

The microprocessor is also often used in industry applications. The keys factors for its popularity are: short time to market, low setup cost, backward compatibility, commercially available image processing tools and software modules. In addition,

the doubling of the processor speed in every 18 months gives them the luxury of improving system performance with near zero development cost.

To a large extent, the performance of such systems greatly depend on the computing speed of the processor. This solution does not actually map the software with appropriate hardware functional units to exploit both data and computational parallelism. Rather, it is an interpreter and translator of algorithms being read from memory. The microprocessor architecture requires many load, store and branch operations. These operations are used to perform various data manipulations. Hence, most of the computing time is spent on "overhead" instructions rather than the actual processing of data. As a result, the silicon area to data processing ratio is low. Most of the silicon area is used for communication, control logic, functions and the management of the flow of computing instructions. As such, in microprocessor implementations, most computationally complex applications spend 90% of execution time on 10% of the codes [22]. Therefore, research has been carrying out in parallel processor architecture. It is a well-known fact that parallel processors always perform better than a microprocessor.

1.2.2 DSP Processors

Signal processing applications, by their very definition, process signals which are generated in real time. Traditionally, much signal processing work has operated on one-dimensional signals, such as speech or audio. To obtain real time performance for these applications, processors with architectures and instruction set specially tailored to signal processing began to emerge [5]. Typical features included multiply and accumulate instructions, special control logic and instructions for tight loops, pipelining of arithmetic units and memory accessing, and Harvard architecture (with separate data and program memory spaces). More recent designs (such as some in the Texas range of DSP processors) have featured explicitly for (two-dimensional) image processing, particularly with image compression in mind.

When carefully programmed to exploit the special architectural features, these

processors can yield very impressive performance rates. However, there is a cost. The programming model at the machine level is much more complex than for traditional microprocessors. Highly optimizing compilers are needed if the processor's potential is to be realized with a high level language.

1.2.3 Application Specific Integrated Circuit

Application Specific Integrated Circuit (ASIC) has the highest degree of computational parallelism. This device is usually chosen in cases whereby sequential processors have reached the performance limits. Any further improvement in performance can only be obtained by adding more processors. For this reason, parallel processing techniques have been widely studied for image processing applications [21]. In some cases, techniques have been developed specifically for image processing; in other cases, standard parallel processing techniques have merely been applied.

1.2.4 Reconfigurable Architecture

In the mid-1980s, a new technology for implementing digital logic was introduced: the Field Programmable Gate Array (FPGA). The introduction of FPGA provides the flexibility to configure the hardware. The FPGA consists of hardware logic that are unconnected. It can be programmed to interconnect the various available logic components to implement any desired digital function. For the advantages it offers, the reconfigurable devices open a new area of research in custom and parallel computing [29][6][11].

The rapid progress in microelectronics and FPGA provides an architectures that have higher speed and density. Hence, the FPGA architectures are potential candidates for computational intensive applications. They also provide customization of hardware without the risk and high setup cost involved with ASIC

implementation. The main advantage of FPGA-based processors is that they offer near supercomputer performance at relatively low costs [59]. FPGAs provide the benefits of customized hardware architecture and at the same time allowing for dynamic reprogrammability. It is an important characteristic that meets the changing requirements of the wide range of applications.

Reconfigurable architectures can be designed to achieve different levels of performance for a given application. The custom logic are designed to exploit parallelism at different areas and levels of the application. Of particular importance and interest, is the use of these techniques to produce compact and fast circuit. Such mapping tends to be most successful for implementing algorithms with high degrees of parallelism [10].

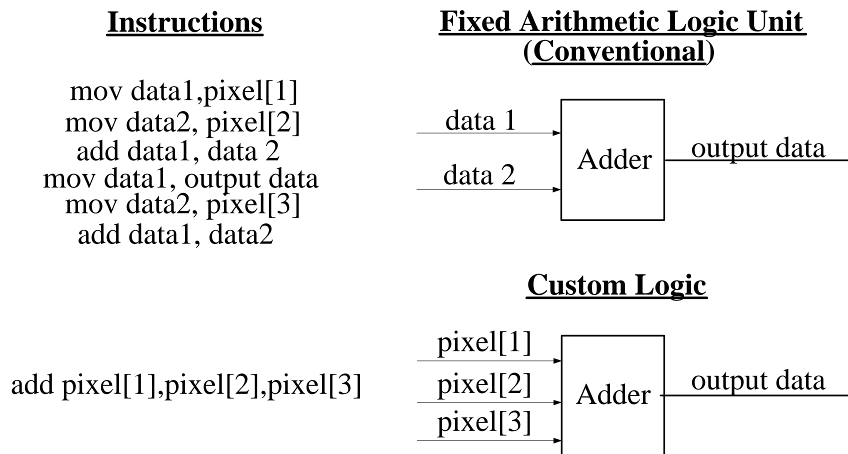


Figure 1.4: Fixed Arithmetic Logic Unit (ALU) vs Custom ALU

To implement efficient solutions, the algorithm and hardware architecture must be well matched to improve overall computational efficiency and concurrency. This can be achieved for algorithms with high degree of regularity that are identified to exploit its parallelism. The operations are mapped into custom functional units to achieve higher performance compared to the fixed processing unit. Figure 1.4 demonstrates the example of computational efficiency of processing three pixels in a single cycle, as compared to multiple cycles for a fixed Arithmetic Logic Unit (ALU).

1.3 Data Processing at Different Level

Image processing consists of several sub-system operations. They are generally categorized into pre-processing, segmentation, feature extraction and classification. The process is sequential with each step gradually transforming the image data to give a higher level of abstract image information.

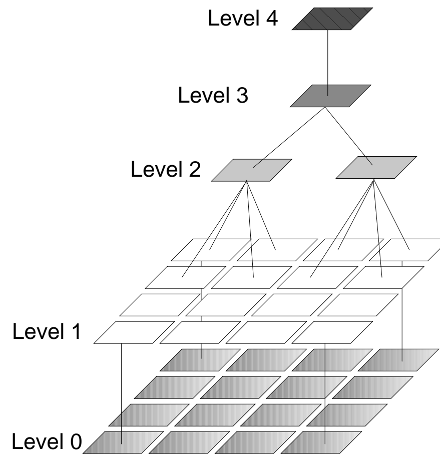


Figure 1.5: Data processing at different level

The amount of data to be processed is modelled using a pyramid architecture as shown in Figure 1.5. The bottom level of the pyramid represents the data volume to be processed and similarly the top level of the pyramid represents abstract information derived from the image. The lowest level comprised of the raw pixels acquired from the source image. Intermediate level 1, 2, 3 are typically pre-processing, segmentation, feature extraction and classification. The final level produces abstract data as a feedback control signal in vision servo.

The vision task at the lowest level is often identified as the process that consumes the most computing resource. The Low level tasks consist of pixel-based transformation such as filtering and edge detection. These tasks are characterized by large amount of data pixels, small neighbourhood operators, and simple structured operations (e.g multiply and add functions) [31]. Computational intensive and yet repetitive algorithms fall in this category at the lowest level of the pyramid; convolution, thresholding and component labelling.

On the other hand, higher level tasks are more dynamic in nature. These tasks are more decision oriented and do not have a repetitive execution of a set of algorithms. The intensive processing of image at each stage, requires efficient architectural support for frequently accessed functions. The first step to exploit parallelism is to identify the sub-system that demands heavy workload. Next, the critical section of the algorithm within the sub-system must be identified as well. With reference to Figure 2.2, performance improvement will be significant when exploiting parallelism in Level zero. The following sections discuss about the hardware architecture design for preprocessing, edge detection and boundary detection tasks. Figure 1.6 shows the different stages of image processing for object recognition.

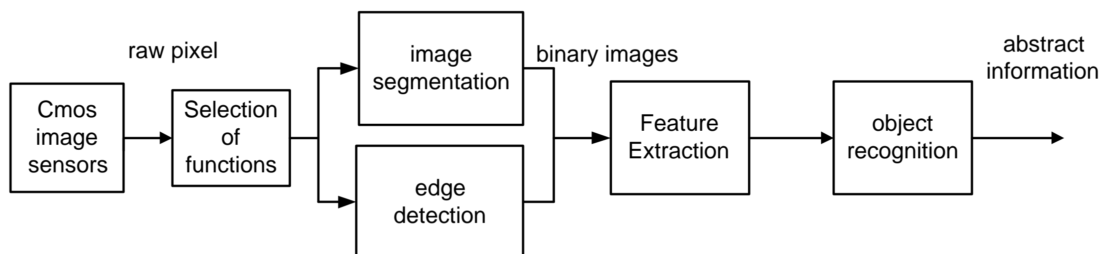


Figure 1.6: Stages for image processing

Researchers have recognized that a new architecture is necessary for real-time image processing. Several optical sensors are developed, to perform on-chip preprocessing task at the pixel level. This dramatically simplify the extraction of the desired information [34][33]. Any image processing task that is performed within the sensor itself reduces the communication and processing workload of the host controller.

On-chip processing has an important role to play in the viability of visual servoing applications. With the increasing accessibility of custom logic design, this makes the development of smart image sensing architectures attractive.

1.4 Motivation and Contribution

Mobile robots with size constraints generally have limitations on the kind of hardware that can be used for the vision system. As a result, most of the vision processing operations have to be performed off the board, i.e. on a host computer. To achieve a self-contained and fully autonomous robot, real-time vision processing is required. At many times, to achieve the desired performance, a high speed processor is required.

Machine-vision applications that demand computational expensive algorithms can be accelerated by custom computation units. With the emergence of reconfigurable devices, many of the on-going research efforts use FPGAs to increase the performance of computationally intensive image processing applications. Such approaches can reduce the necessity of employing high-end processors.

The aim of this research is to investigate the methods of achieving the desired performance, without utilizing high-end microprocessors. Techniques of exploring computationally efficient algorithms and exploring various hardware architectures are studied. Low-level tasks consisting of pixel-based transformations, such as filtering, image segmentation, image convolution and edge detection algorithms are implemented in this work.

With the aim of exploring custom hardware architectures, an analytic mathematical model is derived. The model is used to study the required processing speed of Digital Signal Processor and memory requirements. Additionally, the mathematical model helps to analyse the performance of custom architecture without the need for a simulation model.

Together with the mathematical model and the selected FPGA board, memory chip and CMOS image sensor, the custom architecture is tested in both simulation environment and actual hardware setup.

Using the available FPGA logic resources, the custom architecture is configured to exploit the computation parallelism. The limitations discussed in section 1.1 are

addressed in the proposed design. Real-time VGA images is computed at a very high speed of 30 fps. Furthermore, the memory optimisation technique employed allows all image buffers to fit within the available on-chip memory. Collectively, this work addresses three main constraints of processing real-time images in embedded system. These are computational speed, memory size and physical size constraints.

1.5 Thesis Outline

This thesis is organised as follows, Chapter 2 introduces and evaluates on the various type of image sensor, FPGA and development tools required for the experimental setup. It also includes the introduction to the different types of colour space.

Chapter 3 presents an analytical mathematical model to estimate the various performance parameters associated real-time image processing. The model allows system designers to estimate the required memory size and processing frequency of a given microprocessor architecture. In Chapter 4, the image acquisition, compression, buffering and image convolution are studied. Custom architecture are designed with the considerations of optimising for logic and memory resources.

Chapter 5 is devoted to the FPGA Implementation of Parallel Architecture. Specifically, the low pass filter, edge detection and thresholding algorithm are investigated. The parallel architecture is designed to accomplish high performance image processing task. Methods and techniques are investigated to implement the design with the minimal resources needed.

Finally the thesis is concluded in Chapter 6 with a brief on the major results and observations obtained and an outline of possible directions for future work.

Chapter 2

System Level Architecture Design

The chapter discusses about the various type of image sensor, FPGA and development tools required for the experimental setup. In additional, the different colour space that are suitable for image processing is also included.

2.1 System Components Studies

Selecting the proper hardware components is one of the critical decisions that controls the success or failure of the project. There are many criteria to be considered in the process of selection. The few main considerations are component size, memories size, sensor resolution and frame rate.

There are various types of image sensors available in the market. A comparison of CCD image sensors and CMOS image sensor is conducted. Furthermore, the various types of CMOS sensor are narrow down for selection. In this project, the selection of image sensor is very much focused on the resolution and the interface to the FPGA.

The following sections discuss about the various image sensor, memories and FPGA development board available in the market. Figure 2.1 shows the overview

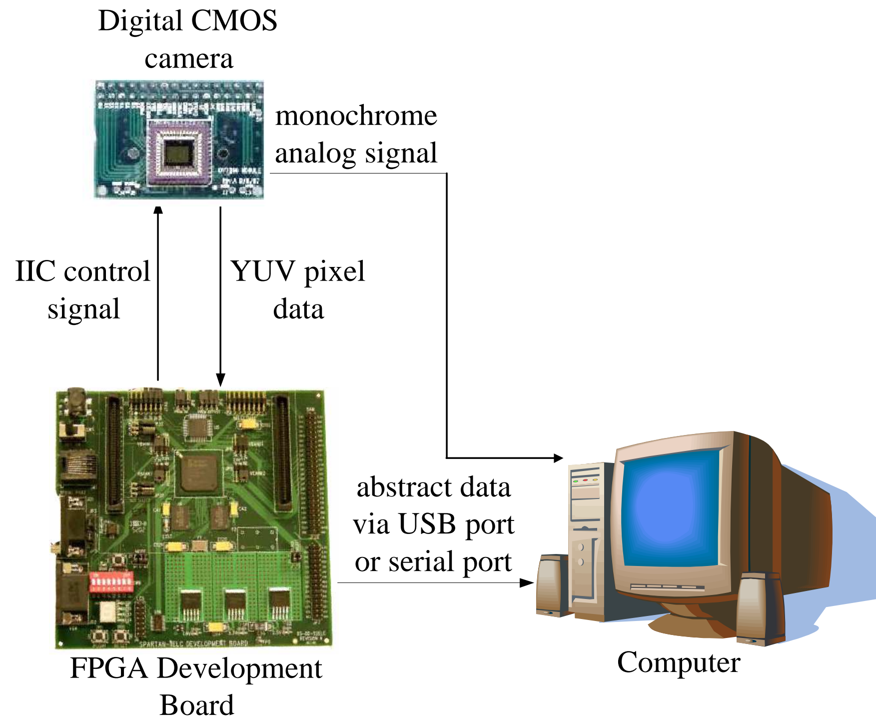


Figure 2.1: MicroViz setup configuration

of the physical interface circuitry between the various components.

2.1.1 Image Sensors

CMOS sensors rose to the top of the hype curve in the 1990s, promising to do away with their predecessors, the CCD sensor. CCDs traditionally use a process that consumes more power as compared to CMOS image sensors. It consumes as much as 100 times more power than an equivalent CMOS sensor [37]. As a result, CMOS sensor with low power dissipation at the chip level, coupled with its small form factor and the ability to deliver high frame rate, emerges as the suitable candidate for many low power mobile applications.

A major advantage of CMOS over CCD camera technology is its ability to integrate additional circuitry on the same die as the sensor itself. This makes it possible to integrate the Analog to Digital Converters (ADCs) and associated pixel grabbing circuitry. Thus a separate frame grabber is not needed [2].

A study is conducted to evaluate the suitability of various image sensors for this purpose. The six different sensors are shown in table 2.1.

Table 2.1: Comparison of available on-board vision processor

Supplier	Model	Resolution	AD	Output format	Frame rate
VLSI Vision	VV6300	160 x 120	8 bit	Bayers RGB	60 fps
Hynix	HV7131GP	652 x 492	10 bit	YCrCb, RGB	30 fps
Pictos	MK00-D190	640 x 480	10 bit	RGB, YCrCB, JPEG	30 fps
Kodak	KAC-0311	640 x 480	10 bit	Bayers RGB	60 fps
OmniVision	OV6620	356 x 292	10 bit	YCrCb, RGB	60 fps
OmniVision	OV7620	664 x 492	10 bit	YCrCb, RGB	60 fps

The OV7620 CMOS image sensor from OmniVision is chosen since is offers the best configurations based on its resolution, frame rate and data format. The OV7620 is able to configure the data output in RGB bayers format or YCrCb format for different types of image processing requirements.

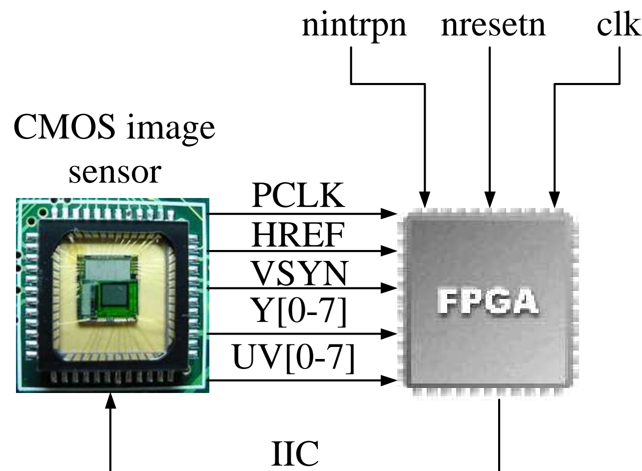


Figure 2.2: OV7620 Image sensor and FPGA

The OV7649 (Figure 2.2) is a 1/3” color camera module with digital output

ports. The digital video port supplies a continuous 8/16 bit-wide image data stream. All camera functions, such as exposure, gamma, gain, white balance, color matrix, windowing, are programmable through the Inter-IC Connection (IIC) interface.

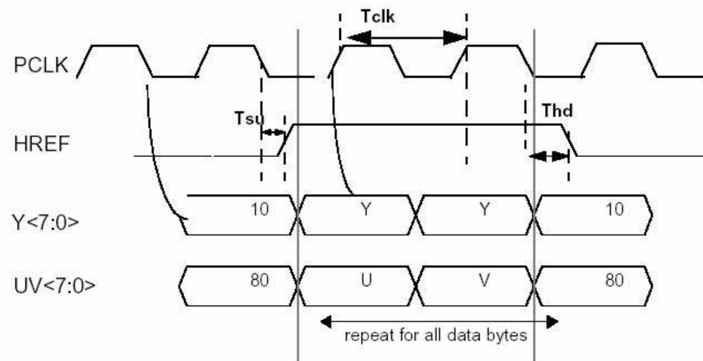


Figure 2.3: Timing waveform of pixel data bus [36]

The OV7620 supports some flexible YCrCb 4:2:2 output format. For instance, for every Pixel Clock (pclk) cycle, the 16 bit pixel data is placed on the Y and UV data bus. Using the YUV 4:2:2 subsampling format, the sequence output is given as

Y (8 bit databus) :	Y0 Y1 Y2 Y3
UV (8 bit databus):	U0 V1 U2 V3

Hence, the respective Y,U and V is mapped to the following four pixels:

Pixel 0	Pixel 1	Pixel 2	Pixel 3
[Y0 U0 V1]	[Y1 U0 V1]	[Y2 U2 V3]	[Y3 U2 V3]

2.1.2 Memories

In any image processing system, buffering the input image signal is necessary. The camera module produces 640 x 480 (VGA) colour pixels at a rate of 30 frame/sec. In order to process an entire image, the entire image is often buffered prior to any processing. For a given data output format of YCrCb 4:2:2, a pixel consists of 16 bits. Hence, from the calculations, the data to be stored is very large.

Number of pixel per frame: 640x480=30,7200 pixels
Size per frame (RAM): 30,7200 x 16 bit= 4.9152 Mbit = 600 KBytes
Buffer Memory for processed image: 614.4 KBytes

A memory storage space of 4.9152M bit is required to store an entire frame. These values exclude data buffers and other overheads. The significant huge amount of memory space seriously poses a problem in the embedded world, where memories are very expensive.

2.1.3 FPGA Development Board

A survey is performed to evaluate the different types of FPGA available in the industry. There are various vendors that manufacture FPGAs. The more prominent ones are Xilinx, Altera, Cypress and Quicklogic.

Xilinx and Altera are the leading manufacturers of FPGAs. They provide extensive support for both industrial and academic developers. As a result, the Spartan-IIE from Xilinx is selected as a suitable platform for this research project. The Spartan-IIE system board connected together with the CMOS sensor board are shown in Figure 2.4.

The Spartan-IIE system board utilizes the 300,000-gate (XC2S300E- 6FG456C) with a 456 fine-pitch ball grid array package. The high gate density and large number of user I/Os allows complete system solutions to be implemented in the

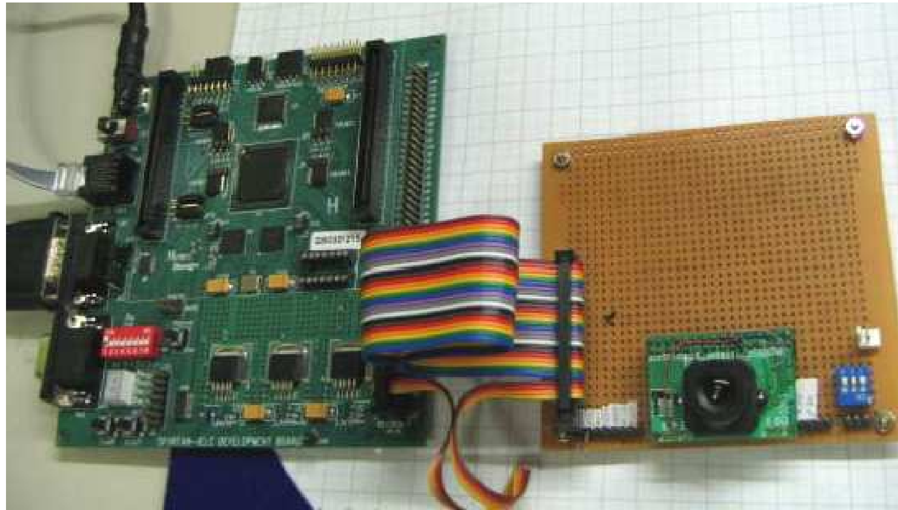


Figure 2.4: MicroViz Prototype board

low-cost Spartan-IIE FPGA. The board also supports the Memec Design P160 expansion module standard, which allows application-specific expansion modules to be easily added.

The Spartan-IIE incorporates several large block RAM memories. These complement the distributed RAM LUTs that provide shallow memory structures implemented in CLBs. Block RAMs are organized in columns. Most Spartan-IIE devices contain two such columns, one along each vertical edge. The XC2S400E has four block RAM columns. The XC2S300E has a total of 16 RAM blocks [30].

2.2 Simulation and Development Tools

The following section discussed about the programming and analysis tools used in this research project. The selection of Hardware Description Language (HDL) and the introduction to FPGA design flow is also covered in this section.

2.2.1 Programming Tools

There are many development and analysis tools required for this research as shown in Table 2.2. Simulations necessary prior to actual implementation. The Visual C/C++ is used as a platform to test and evaluate any new algorithms. After which, the equivalent verilog codes are written according to those verified in C. The verilog codes are simulated using ModelSim, producing simulation waveform of data signals for verification purpose. The Xilinx Integrated Software Environment (ISE) translates verilog codes into hardware logic circuits. This process is often known as synthesis. After the FPGA is programmed, the data signals are verified using the oscilloscope, ANT16 logic analysis and the Chipscope Pro. The schematic design and PCB is designed using Protel 99SE.

Table 2.2: Development and analysis tools

Visual C/C++ 6.0: For simulation of algorithm in C
ModelSim: Simulation package for vhdl and verilog code
Xilinx ISE 6: Design Entry, design synthesis and device programming.
Xilinx EDK 6: Hardware specifications, MicroBlaze microcontroller
Chipscope Pro: Internal register Logic Analyzer
ANT16: External data bus Logic Analyzer
Irfanview: Portable Pixel Map file image viewer
Protel 99 SE: Schematic entry and PCB design

2.2.2 FPGA Design Flow

The FPGA design flow is illustrated in Figure 2.5. An idea or concept is translated into Verilog HDL. This language is often used in the design at an entry stage.

Alternatively, Electronic Design interchange Format (EDIF) or schematic entry is used for design entry. Following that, the user constraints file (ucf) specify the timing and pin location constraints. A logic synthesis tool reads a HDL entry and produces a netlist consisting of a description of basic logic cells and their interconnections (Figure 2.6).

The implementation of a digital logic design with a FPGA involve a design flow similar to ASIC design flow [28].

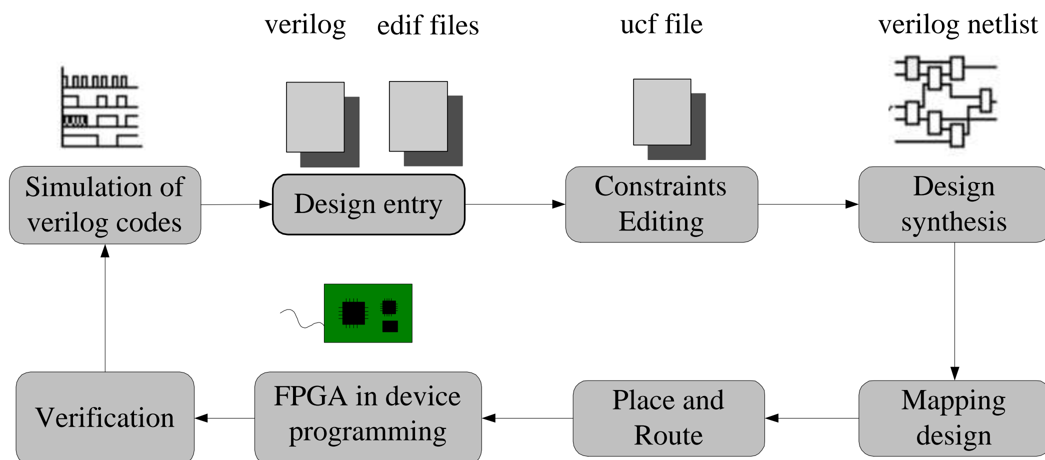


Figure 2.5: FPGA design flow

The mapping function allocates Configuration Logic Blocks (CLB) and Input Output Blocks (IOBs) resource for all basic logic elements in the design. It considers the available resources together with the constraints specified and map the digital logic design into the targeted FPGA chip.

The Place and Route (P&R) process decides the location of the cells in a block and places the connections between the cells and blocks. The generated bit stream file is programmed into the FPGA via a Joint Test Access Group (JTAG) connection. The results are verified using Chipscope Pro, PC-USB Logic Analyser and oscilloscope. This process is often repeated for many iterations to yield satisfactory results.

The Xilinx RAM-based FPGA features a logic block that is based on LUTs. A

```
(cell LUT4 (cellType GENERIC)
        (view view_1 (viewType NETLIST)
           (interface
              (port I0 (direction INPUT))
              (port I1 (direction INPUT))
              (port I2 (direction INPUT))
              (port I3 (direction INPUT))
              (port O (direction OUTPUT))
           )
        )
     )
     (cell MUXCY (cellType GENERIC)
        (view view_1 (viewType NETLIST)
           (interface
              (port DI (direction INPUT))
              (port CI (direction INPUT))
              (port S (direction INPUT))
              (port O (direction OUTPUT))
           )
        )
     )
```

Figure 2.6: Gate level netlist

LUT is a small one bit wide memory array, the address lines for the memory are inputs from the logic block and the one bit output from the memory is the LUT output. A LUT with K inputs would then correspond to a 2^k x 1 bit memory. It can realize any logic functions of its K inputs by programming the logic function's truth table directly into the memory.

Each Spartan-IIE CLB contains four Logic Cells (LCs), organized in two similar slices; a single slice is shown in Figure 2.7. This arrangement allows the CLB to implement a wide range of logic functions. Furthermore, each LUT can provide a 16 x 1 bit synchronous RAM. The two LUTs within a slice can be combined to create a 16 x 2-bit or 32 x 1 bit synchronous RAM, or a 16 x 1 bit dual-port synchronous RAM [30] [29].

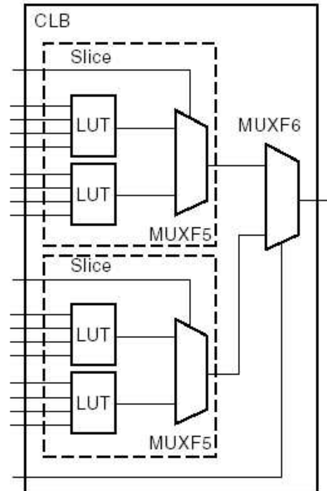


Figure 2.7: Configuration Logic Block [30]

2.2.3 Verilog vs VHDL

Schematic capture and hardware description language are used for design entry. The two industry standard hardware description languages are VHSIC HDL (VHDL) and Verilog.

VHDL was developed by committee intended for documenting digital hardware behaviour. It originated out of the Very High Speed Integrated Circuit (VHSIC) Program as a part of a US Department of Defense Project in 1981. Although it was adopted by many Electronic Design Automation (EDA) companies and carried strong support from the European electronics market, VHDL had significant deficiencies. There was no facility for handling timing information [25].

On the other hand, Verilog HDL came from the commercial world and was developed as part of a complete simulation system. It was also developed to describe digital based hardware systems. The Verilog HDL is used extensively, since launched in 1983 by Gateway. It became the IEEE standard 1364 in December 1995 [26].

A comparison is made between the two HDL lanaguages is shown in Table 2.3. It is also noted that there are increasing number of universities adopting to teach

Table 2.3: Comparsion of VHDL and Verilog

	VHDL	Verilog
Learning curve	A strongly typed language with heavy syntax	Easy to pick up for those with C language background
Design reusability	Procedures and functions may be placed in a package	Define modules to reuse design
Datatypes	Dedicated functions are needed to convert objects from one datatype to another	Easy to use and geared towards modelling hardware structure
HDL modelling capability	Good for modelling large design structure, unable to provide gate level modelling	Developed with gate level modeling in mind
Usage in Digital Design Market world wide	40% (mainly in europe, military and academic institutes)	60% (mainly in US and Asia companies)

this language as part of their advanced Electrical Engineering programs; and to dated more than 75 companies offer Verilog HDL products and services [25].

As a result, Verilog is chosen for this research project. The primary reasons are the ease of usage, which is similar to C and the popularity of its usage in the industry.

2.3 Image Representation

Images are represented in a form of analogue or digital signals. Analogue signals are traditionally used in many types of video equipment, and mainly used for television broadcasting.

However, in recent years, digital image and video are rapidly taking over many applications. The most common digital signals used are RGB and YCbCr. The RGB format is commonly used for display devices such as LCD display panels, while the YCrCb is often used for data transmission and data processing.

A digitised colour image is represented as an array of pixels, where each pixel contains numerical components that define a colour. The images captured from the camera will consist of these array of pixels.

After an image is captured, it is represented in various formats. Typically, the binary, greyscale, Red Green Blue (RGB), Hue Saturation Intensity (HSI) or the YCrCb format are used (Figure 2.8). The binary image format represents the simplest form of an image, with a one bit representing one pixel. Hence, for a given image of 640 x 480, is represented by 38400 bytes. Although a binary image offers a small file size, there is a significant loss in image quality.

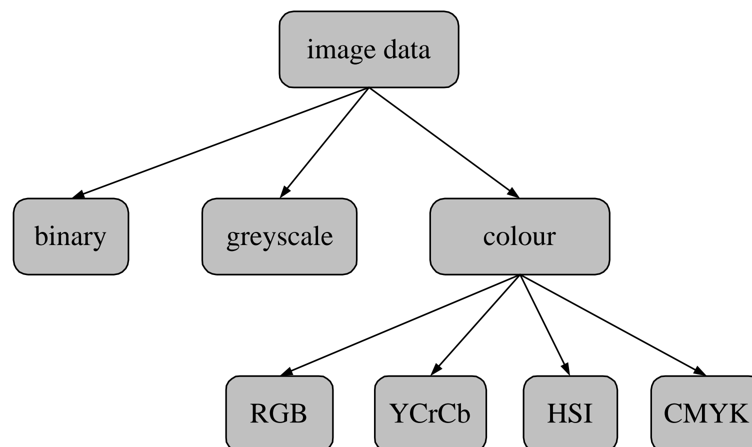


Figure 2.8: Colour Space

In a typical 8 bit grey scale image, there are 256 shades of grey. Each pixel

represents different shade of grey according to its brightness.

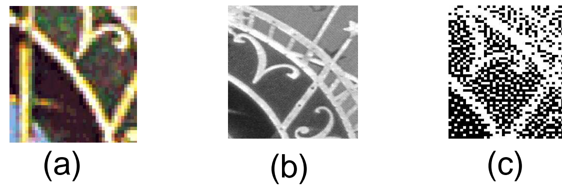


Figure 2.9: (a)RGB colour image (b)Greyscale image (c)Binary image

Colour image can be represented in RGB, YCrCb or HSI colour format. RGB is typically used to display images, YCrCb for transmission of image data and HSI for image processing. Figure 2.9 shows the different data representation of image in binary, greyscale and RGB colour format.

RGB colour space

The RGB space is widely used throughout computer graphics. The individual component are added together to form a desired colour, it is represented using cartesian coordinate system as shown in Figure 2.10.

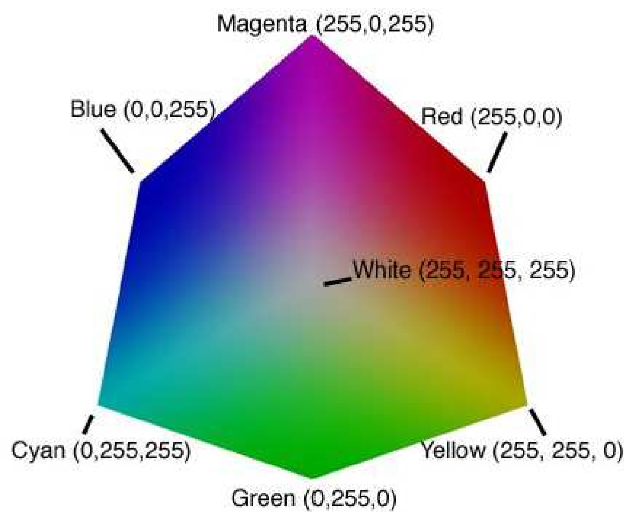


Figure 2.10: RGB colour space [35]

However, RGB colour space is not very efficient when dealing with real world images. Processing an image in the RGB colour space requires the modification of all three colour components [27]. For instance, to increase the intensity or colour

of a given pixel, all three components must be read, calculate the new value and written back into the memory for each pixel. For this reason, other colour space is derived from the basic RGB colour space to ease date processing. The HSI and the YCrCb colour space is discussed in the following section.

HSI colour space

HSI is preferred in some systems as it separates apparent "colour" from "brightness". Colour in HSI space is relatively more robust to illumination, lights and noise as compared to RGB is more sensitive to highlights and shadow. The HSI colour space is shown in Figure 2.11.

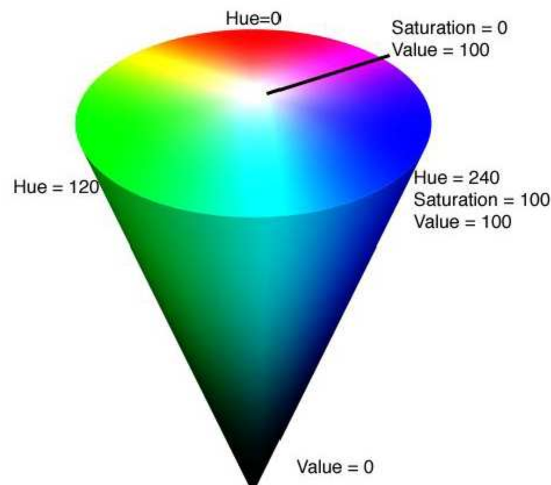


Figure 2.11: HSI colour space [35]

YCrCb colour space

The image data represented in YCrCb color space is sampled in 4:2:2, 4:2:0 or 4:1:1 sampling format. In YCrCb 4:2:2 format, for every four samples of Y component, there are two Cb and Cr. Each sample is typically 8 bits. This positioning of YCrCb colour component sampling offers a reduction of bandwidth for transmission. This is due to the fact that YCrCb 4:2:2, 4:2:0 and 4:1:1 use a lower sampling rate for the chromatic components, hence require less storage space and transmission bandwidth.

The YCrCb colour space is also derived from the RGB colour space. The Y component is the luminance, Cr represents a colour value consisting of the luminance deducted from the color red (R-Y) and Cb represents the color value of the luminance deducted from the color blue (B-Y).

The next chapter presents an analytical mathematical model to estimate the various performance parameters associated real-time image processing. The model allows system designers to estimate the required memory size and processing frequency of a given microprocessor architecture.

Chapter 3

An Analytic Model for Embedded Machine Vision

3.1 Introduction

This chapter focuses on the performance of the embedded machine vision. A model is introduced to estimate the performance and memory resource requirements.

In literature, there are three basic techniques for performance evaluation; namely measurement, simulation and analytic modelling [40] [41].

An analytic model is often used to predict performance. It can evaluate the performance with minimal efforts and costs over a wide range of choices for system parameters and configurations [42]. For various processor architectures, the key resources and workload requirements can be analytically modelled with sufficient realism to provide insight into the bottlenecks and key parameters affecting the system performance [46]. However, such approach is impractical, if the vision system is modelled in great details [40].

An analytic model is derived to provide a mathematical description of the vision system. Such approach is considered being far less time consuming and more

flexible compared to simulation based methods. A model is proposed to analyse the performance of processing real-time images in embedded systems. Last but not least, the limitations of general purpose processors are also discussed.

Section 3.2 presents a model to determine the optimal memory size for buffering real-time images. Section 3.3 presents another model to calculate the processing frequency required to perform certain algorithms. Specifically, image segmentation and convolution algorithms are analysed in Section 3.4 and 3.5 respectively.

3.2 Analytic Model to Determine Image Buffer Size

3.2.1 Concept of Queuing Theory

The image acquisition process is modelled using a producer and consumer process. The CMOS image sensor produces image data and the image acquisition consumes image data. Due to the difference in arrival and consumption rate, the producer and consumer processes are buffered by a message queue. A message queue is a set of memory locations that provides temporary storages for data that are being passed from one process to other. In general, a producer places new message into the queue while the consumer acquires the message by removing it from the queue. For this approach, a First-In First-Out(FIFO) RAM is commonly used.

It many embedded systems, it is required to estimate the maximum number of messages that will queue in a system. Empirical methods to estimate the required capacity are not reliable [44], and these methods are often not able to determine the optimal memory size for different algorithm. Furthermore, empirical methods are often conducted using the actual experimental setups or simulation models. For these reasons, an analytic model is derived from queuing theory to compute the maximum message queue length.

Queueing theory plays a very important role in analytical modelling [42]. The concept is used as the fundamental formulae to calculate the system requirements for real-time operations.

The system is modelled using the queueing analysis as illustrated in Figure 3.1. The definitions used in this model are as follows:

$n_q = no.\ of\ jobs\ in\ queue$

$\lambda_b = arrival\ rate\ of\ pixel\ during\ burst$

$\mu_b = service\ rate\ of\ pixel\ during\ burst$

$t_e = time\ require\ to\ empty\ buffer$

$t_b = burst\ time$

$t_o = time\ of\ occurence$

$t_s = service\ time\ per\ pixel$

$n_{ipp} = no.\ of\ instructions\ per\ pixel$

$n_{cpi} = no.\ of\ clock\ per\ instruction$

$t_{clk} = processor\ clock\ cycle$

$f_{clk} = clock\ frequency$

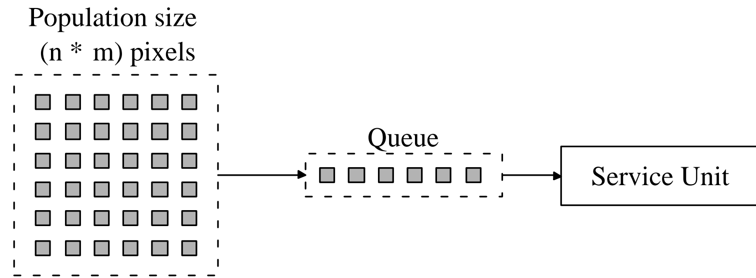


Figure 3.1: Queue model of vision system

The image sensor, or producer is said to have a population size of n*m messages. If the messages arrive at a rate faster then the system can service, a queue is formed. The sudden arrival of messages for a period of time is call a burst. During the burst t_b, a buffer is required to absorb any excess of production of pixels over the consumption.

At the beginning of each burst, the message queue should be empty. The pixels will be placed into the queue synchronous to the clock of the image sensor, which is also refer as the arrival rate λ_b. At the same time, some pixels are consumed by the image acquisition routine, at a consumption rate μ_b. If $\lambda_b \leq \mu_b$, the queue will not grow, else the queue will grow. In the situation where the pixels arrive at a rate faster than it can handle for a long enough period of time, the messages continue to stack up in the queue, to a point overflow occurs.

$$n_q(t) = \lambda_b \; t \; - \; \mu_b \; t \tag{3.1}$$

Equation 3.1 is used to determine the number of messages $n_q(t)$ in the queue at time t, where $\lambda_b \; t$ is the number of messages arrived in the queue at time t and $\mu_b \; t$ is the number of messages consumed from the queue at time t.

3.2.2 Row buffering

A row buffer is proposed in this work to acquire a single row of pixels and process it before acquiring the next row of pixels.

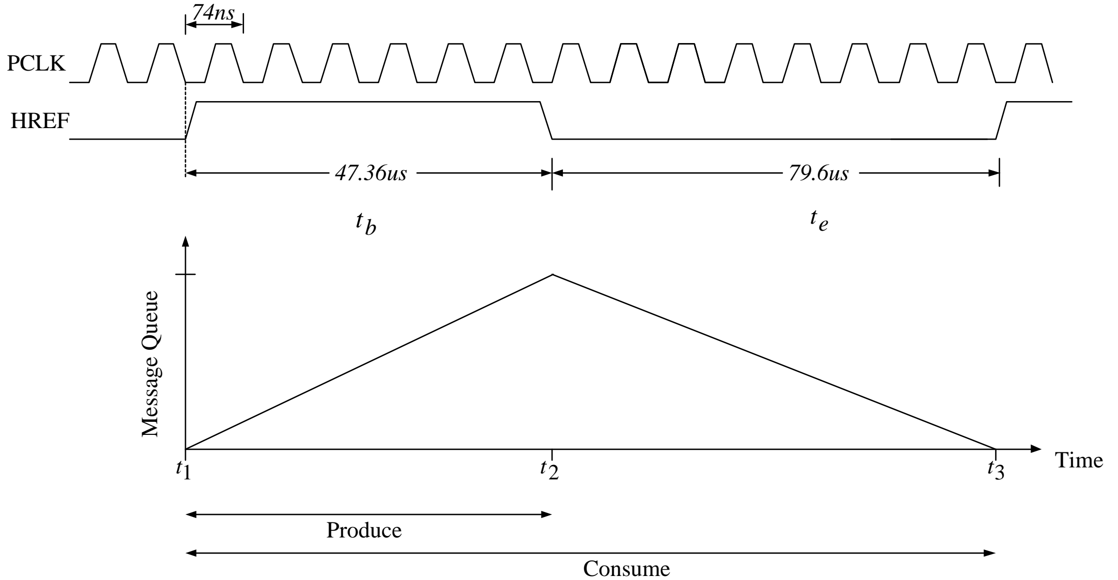


Figure 3.2: Burst time and emptying time

Figure 3.2 illustrates the producer and consumer of pixels in the message queue. It shows that the burst t_b is 47.36 us and the maximum time allowed for the consumer to service all pixels is $t_b + t_e$. The required memory size can be determined by the maximum length of a queue. From Figure 3.2, it is noted that the queue length peaks after the last pixel is place in the queue, at t=t2.

At t_1 (Figure 3.2), the first pixel is placed into the message queue. For every $PCLK$, a pixel arrives at the message queue when HREF is high. At the same time, some of these pixels are consumed in a First-In First-Out order. The message queue continues to grow as long as the arrival rate is greater than consumption rate (service time). At end of each burst, immediately after the last pixel is placed into the message queue at t_2, the message queue stop to grow. The production rate is equals to zero while the consumption rate remains the same. At this point, the total no. of messages remaining in queue is

$$\begin{aligned} n_q(t_b) &= \lambda_b \; t_b \;\; - \;\; \mu_b \; t_b \\ &= (\lambda_b - \mu_b) \; t_b \quad , \end{aligned} \tag{3.2}$$

where

$$\lambda_b = \frac{w_i}{t_b} \; , \tag{3.3}$$

$$\mu_b = \frac{w_i}{t_b + t_e} \; , \tag{3.4}$$

The maximum time given to the consumer to empty the message queue before the arrival of the next pixel is $t_e = t_2 - t_1$. Hence, the worst case consumption rate μ_b is given in (3.4), where w_i is the image width (number of pixels in a row).

If a new burst begins before the queue is totally emptied, the allocated message queue will not have the capacity to store all pixels for the next row. As a result,

buffer overflow will occurs. Hence, it is important to note that the all message should be consumed within the allocated time.

If the $\frac{\lambda_b}{\mu_b}$ ratio is large, the emptying time will be very much longer than the burst time. As a result, the maximum queue length increases accordingly.

With (3.3) and (3.4), $n_q(t_b)$ is simplified to

$$\begin{aligned} n_q &= (\lambda_b - \mu_b)\ t_b \\ &= (\frac{w_i}{t_b} - \frac{w_i}{t_b + t_e})t_b \\ &= \frac{w_i t_b (t_b + t_e - t_b)}{t_b (t_b + t_e)} \quad , \end{aligned}$$

$$n_q = \frac{w_i t_e}{t_b + t_e} \tag{3.5}$$

For instance, if t_e is equal to t_b, the messages are consumed at half the rate of the arrival rate. As a result, the maximum queue length is half of the image width as shown in (3.6).

$$n_q = \frac{w_i\ t_b}{2t_b} = \frac{1}{2}\ w_i \tag{3.6}$$

In this example (Figure 3.2), with $t_b = 47.36\ us$, $t_e = 79.6\ us$ and $w_i = 640$, the required buffer size calculated with (3.5) is found to be 402 pixels. With this model, the buffer size can be adjusted accordingly by reducing the t_e parameter. However, it should be noted that reducing t_e results in the reduction of service time. The next section will present a model to illustrates the effects of t_e on the processor clock frequency.

3.3 Analytic Model to Determine Computational Speed

In many embedded system design, the processor and its clocking frequency are generally determined either by simulation or empirical methods. Many at times, simulation models may not be available. As mentioned earlier, empirical methods are not reliable and do not address the issue of scalability. In this section, a model is derived to estimate the required processing speed.

With the consumption rate μ_b and the no. of message in the queue n_q, the time required to empty the queue is

$$t_e = \frac{n_q}{\mu_b} \tag{3.7}$$

Together with the concept and equations derived in the previous section, the time required to service a pixel can be computed as follows:

$$n_q = (\lambda_b - \mu_b)\ t_b\ , \tag{3.8}$$

$$\lambda_b = \frac{1}{t_{pclk}}\ , \tag{3.9}$$

$$\mu_b = \frac{1}{t_s}\ , \tag{3.10}$$

where t_{pclk} is the inter-arrival time of each pixel, and t_s is the time needed to service one pixel. By substituting (3.8),(3.9),(3.10) into (3.7), t_s is simplified as

$$\begin{aligned} t_e &= t_s[(\lambda_b - \frac{1}{ts})\ t_b] \\ &= t_s t_b \lambda_b - t_b \end{aligned} \tag{3.11}$$

$$t_s = \frac{t_e + t_b}{t_b \lambda_b} \tag{3.12}$$

With reference to Figure 3.2, a pixel arrives at an interval of 74 ns. The burst time t_b is 47.36 us and the emptying time t_e is 79.6 us. Hence, the service time per pixel (3.12) is calculated to be 198.37 ns.

To estimate the processing speed required to compute one pixel, the number of instructions to process a single pixel must be known. An image algorithm may consist of several instructions to compute one pixel. In additional, a single instruction may require of more than one clock cycle to complete.

In general, the target processor is first identified and the instruction set architecture is studied. With the instruction set, the no. of instructions needed to compute a single pixel is estimated. Thus, the service time per pixel is expressed as

$$t_s = n_{ipp}\ n_{cpi}\ t_{clk}\ , \tag{3.13}$$

and together with (3.12) and (3.13), the required processing clock frequency is expressed as

$$\begin{aligned} f_{clk} &= \frac{n_{ipp}\ n_{cpi}}{t_s}\ , \\ &= \frac{t_b\ \lambda_b\ n_{ipp}\ n_{cpi}}{t_e\ +\ t_b}\ , \end{aligned} \tag{3.14}$$

where n_{ipp} is the number of instructions per pixel and n_{cpi} is the number of cycles per instruction.

Lastly, the production rate and consumption rate ratio are found to be

$$\frac{\lambda_b}{\mu_b} = \frac{198.37}{74} = 2.68 \quad .$$

The analytic model discussed in this section provides an estimation of the clock frequency required to perform certain image processing algorithm. With this model, an in-depth analysis of image segmentation and image convolution algorithm are discussed in the following sections. The clock frequency is calculated based on a selected processor architecture and image algorithm is written in C. As such, this will assist in the selection of processor architecture, instruction set and most importantly the optimal processing speed.

3.4 Analysis of Image Segmentation Algorithm

The segmentation process partitions an image into meaningful regions [24]. The nature of this process involves scanning of each pixels. Thresholding is one of the most important approaches in image segmentation. It is a process to convert greyscale images to binary images. Typically, a greyscale image is thresholded to obtain a binarised version of the image that consists of only foreground and background. To obtain a binary image which consists of only foreground and background pixel, a threshold value, T, is used to partition the image into two regions, such that

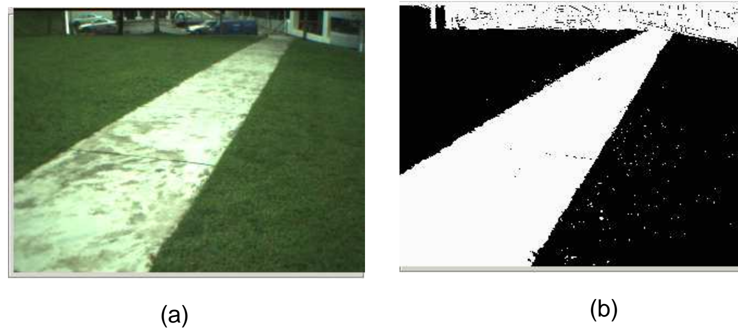
$$If\ f(x,y) \geq T\ then\ g(x,y) = 1$$

$$If\ f(x,y) < T\ then\ g(x,y) = 0$$

where the image $g(x, y)$ denotes the binarised version of image $f(x, y)$.

A simple experimental setup for image segmentation is simulated in Visual C/C++ and MicroChip MPLab. The program reads in the raw image data, converts the Red, Green, Blue (RGB) primaries to greyscale image and performs image

thresholding. Finally, the program writes to an image file for viewing. The result is shown in Figure 3.3.



(a) (b)

Figure 3.3: Thresholding

For instance, a real-time vision system with a resolution of 640 x 480 is required to be processed at a rate of 30 fps (frames per second). The analytical model is used to estimate the processing clock speed needed for handling real-time image thresholding process.

3.4.1 Computation using microprocessor

Image thresholding can be performed using a Digital Signal Processor. Often, such architectures requires several load, store and branch operation to perform data manipulation.

The dsPIC30F6012 DSP controller from Microchip Technology Inc. is used as a target chip in this work. It is a 16 bit Harvard RISC machine designed for embedded system. The processor can reach up to the speed of 30 MIPS. The simplified threshold algorithm written C is translated to assembly language as shown in Figure (3.4). In addition, the number of cycles to complete a specific instruction is also shown.

A further analysis of the program flow reveals that the computational time of each path is different. For instance, if $pixel_in < threshold$, the branch instruction is not taken and the program executes through B1, B2, B3. The total time to

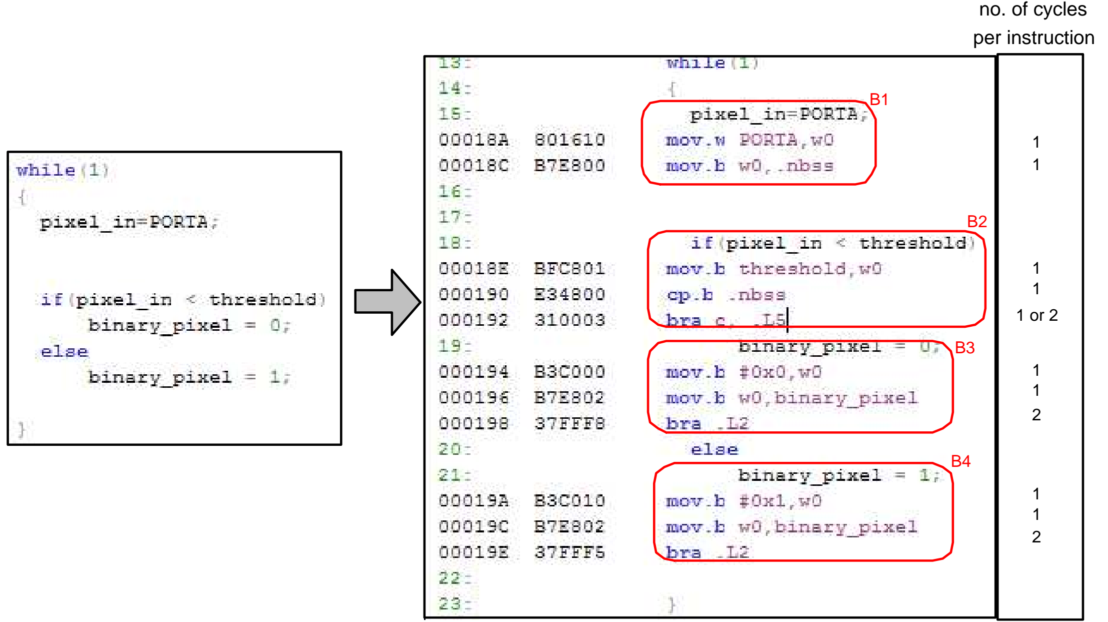


Figure 3.4: Assembly code representation of C program

complete one iteration is 9 clock cycles. Otherwise, the flow of the program is taken in the sequence of B1, B2 and B4. As a result, 10 clock cycles are needed to process a pixel.

By taking the worst case condition, $n_{ipp}\ n_{cpi} = 10$, hence

$$\begin{aligned} f_{clk} &= \frac{t_b\ \lambda_b\ n_{ipp}\ n_{cpi}}{t_e + t_b} \\ &= \frac{47.36\ us(\frac{1}{74\ ns})10}{79.6\ us + 47.36\ us} \\ &= 50.4\ MHz \end{aligned} \tag{3.15}$$

From the calculations, it is noted that a considerably high clocking frequency is needed to perform image thresholding process. Due to the amount of data to be processed in real-time, the processor will not be able to compute all pixels within the allocated time, if $f_{clk} < 50.4\ MHz$.

Hence, from the calculations, a Digital Signal Processor is not suitable for such operations. The image thresholding algorithm does not actually map to appropriate hardware resource to achieve efficiency. Most of computing time is spent on "overheads" instructions rather than for the actual processing of data. It is reported that most computationally complex applications spend 90% of their execution time on only 10% of their code [22].

3.4.2 Computation using custom architecture

Using the model to calculate f_{clk}, the parameters corresponding to the hardware resources are identified to improve computational efficiency. The clock frequency can be reduced by reducing n_{ipp} and n_{cpi}. With the model as a reference, the operations required in image segmentation process are mapped into custom functional units to achieve a lower clocking frequency

In an ideal case, n_{ipp} and n_{cpi} is set to 1. Effectively, from (3.15), the new clock frequency is calculated as

$$\begin{aligned} f_{clk} &= \frac{t_b \ \lambda_b \ n_{ipp} \ n_{cpi}}{t_e + t_b} \\ &= \frac{47.36 \ us(\frac{1}{74 \ ns})}{79.6 \ us + 47.36 \ us} \\ &= 5.04 \ MHz \end{aligned}$$

The possibilities of setting $n_{ipp} \ n_{cpi} = 1$ can be achieved by processing a pixel within a single clock cycle. With a customised architecture, it is possible to complete the same image processing task with a reduction of clock frequency by 90% compared to the implementation in Microchip dsPIC30F6012 processor.

Furthermore, the proposed custom architecture can be further improved by reducing the buffer size. The consumption rate can be customised to match the arrival rate, and at the same time achieving $n_{ipp} \ n_{cpi}$ equal to 1. As a result, all

pixels arrived will be immediately processed or consumed. Hence, the emptying time t_e is zero, and the buffer size required is zero as well.

$$n_q = \frac{w_i \; t_e}{t_b + t_e} = 0 \quad .$$

Consequently, f_{clk} can also be determined from (3.14)

$$\begin{aligned} f_{clk} &= \frac{t_b \; \lambda_b \; n_{ipp} \; n_{cpi}}{t_e + t_b} \\ &= \lambda_b \\ &= p_{clk} \end{aligned}$$

where, the processing clock frequency is equals to the pixel clock frequency. In theory, processing a pixel upon arrival eliminates the need for any buffer memory. However, in practice, all computation must be completed before the arrival of next pixel. Otherwise, the old data will be overwritten by new arrivals.

3.5 Analysis of Image Convolution Algorithm

The last section of this chapter analyses the computation requirement to perform an image convolution algorithm. The convolution algorithm requires a pixel to compute one output pixel, based on 3x3 input convolution window. An experiment is conducted in Visual C/C++ and Microchip MPLab environment. The algorithm is coded in C and compiled into assembly language to obtain the number of operations required to process a single block of C program.

The C program is clustered into blocks to illustrate the number of cycles associated with each block. From Figure 3.5, the main computation takes place in the last block. The image is convolved with a convolution mask. From the assemble code obtained, the block that performs convolution calculations consists of 67 operations. Hence, $n_{ipp} \; n_{cpi}$ is 67 and the f_{clk} can be computed as follows:

no. of cycles per block

```
for(j=0; j<10; j++)                                              12
  for(i=0; i<10; i++)
  {
      image_in[j][i]=pixel_in++;;                                14
  }

for(j=0; j<10; j++)                                              12
  for(i=0; i<10; i++)
  {
      if(i>0 && j>0)                                             4 or 6
      {
      image_out[j][i]= - image_in[j-1][i-1] + image_in[j-1][i+1]
                         -  (2*image_in[j][i-1])  + (2*image_in[j][i+1])   67
                       - image_in[j+1][i-1] + image_in[j+1][i+1];
      }
  }
```

Figure 3.5: Convolution algorithm in C

$$f_{clk} = \frac{t_b \; \lambda_b \; n_{ipp} \; n_{cpi}}{t_e + t_b}$$

$$= \frac{47.36 \; us(\frac{1}{74 \; ns}) \; 67}{79.6 \; us + 47.36 \; us}$$

$$= 337.68 \; MHz$$

With the results calculated, it is not surprising that it requires a processor with such a high frequency to perform the convolution process. However, the clock frequency can be reduced by using the same approach mentioned in Section 3.4.2.

If n_{ipp} n_{cpi} is reduced from 67 to 1, the new clock frequency will be 5.04 MHz. With this approach, the custom architecture is able to perform image convolution at a relatively low clock frequency. The processing frequency of the custom architecture compared to Microchip dsPIC30F6012 can be reduced by 98.5%.

3.6 Summary

The image processing analytic model is presented to estimate the various performance parameters associated with the vision system. The model allows system designers to estimate the required memory size and processing frequency of a given microprocessor architecture. Specifically, the MicroChip DSP processor is used in this work.

Such an approach is considered being far less time-consuming and more flexible compared to simulation based approaches. It helps to compare different processor architectures without actual implementation or simulation. Furthermore, this can be extended to calculate the amount of energy consumed.

The model provides the design space exploration to achieve the desired performance. In one of the examples, the reduction of the number of instruction per pixel yields a lower clock frequency. With such hints, it provides a motivation to process a pixel in a single cycle, which is achievable with custom architecture.

The customized architecture allows direct computation instead of conventional load store operations. Such structure is able to compute a pixel in a single instruction within a single clock cycle. The possibilities of such an custom architecture can be realized in a reconfigurable fabric of FPGA.

This chapter provides a theoretical calculations with assumptions that the custom architecture is able to process a pixel in a single cycle. In Chapter 4 and 5, the detailed implementations of such architectures are discussed.

The next chapter discusses on the image acquisition process with emphasis on the control signals and data format of the pixels produced. A brief introduction to the CMOS image sensor architecture and its interface logic is also covered. In addition, a simple yet effective compression technique is proposed as well.

Chapter 4

Image Acquisition, Compression, Buffering and Convolution

Digital image processing encompasses a sequence of processes that transforms signals from one process to other. In this chapter, the image acquisition process, followed by image compression and storage are discussed.

The basic function of image acquisition is to acquire a digital image from an image sensor. Typically, a CCD camera or a CMOS digital image sensor is used for image capturing. CMOS image sensor is chosen for this work.

An image must be digitised both spatially and in amplitude. Digitisation of spatial coordinates (i, j) is also known as image sampling while amplitude digitisation is known as grey-level quantisation [47]. In CMOS digital image sensor, the sampling and digitisation processes are performed on the chip.

This chapter explores the features of the CMOS image sensor i.e., OV7620, and the methods for fast image compression and storage solutions. The concept of image compression, and its implementation are also included. Section 4.3 explores the various methods for buffering the input image that is used for convolution process while section 4.4 discusses about the basics of image convolution.

4.1 Image Acquisition

4.1.1 Image sensor interface signals

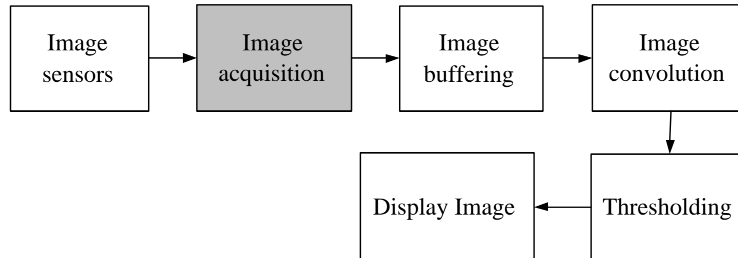


Figure 4.1: Image acquisition process

The first stage of any vision system is the image acquisition stage (Figure 4.1). After the image has been obtained, various methods of processing can be applied to the image to perform the extraction of the desired information. Typically, image acquisition involves both hardware and software aspects. Hence, it is necessary to first understand the interfacing I/O of the CMOS image sensor.

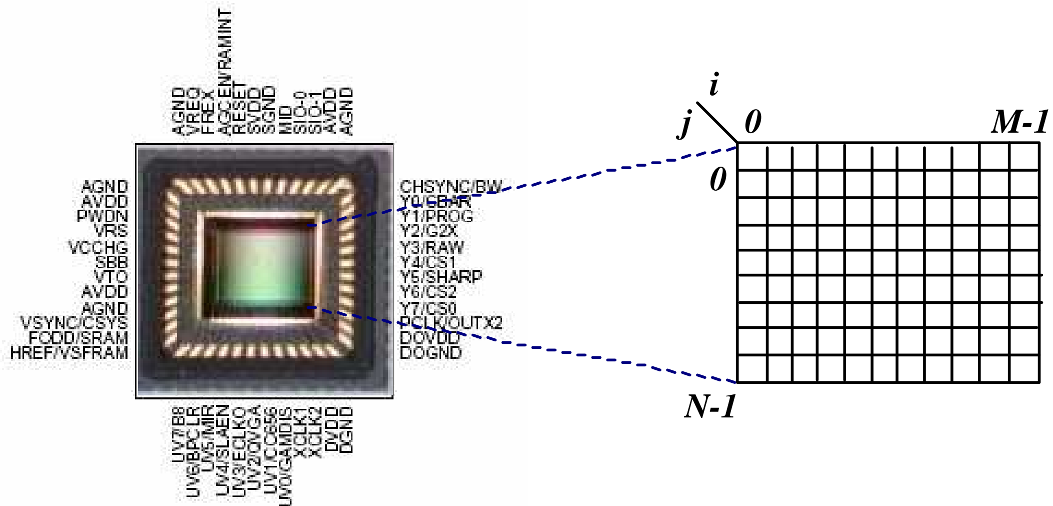


Figure 4.2: CMOS image sensor array

The OV7620 CMOS image sensor from Omnivision is used in this research. It is a highly integrated CMOS with a resolution of 664 x 492 pixels.

When an image is captured by the sensor, it is arranged in the form of an N x M array (Figure 4.2), where each element in the array is a discrete quantity. The output resolution of the image sensor can be configured to QVGA (320 x 240) or VGA (640 x 480) resolution by setting the register in the image sensor chip.

The image captured in the form of pixels is expressed as a two-dimensional light intensity function, $f(i, j)$, where the amplitude of fat coordinate i and j gives the brightness of the particular pixel. For instance, $f(5, 2)$ refers to the brightness level of the pixel in second row, fifth column.

The OV7620 consists of three primary control signals and two output data ports. The three control signals (PCLK, HREF and VSYN), provide the synchronization signals for the output data pixels to be read by the image acquisition device. The two digital data ports, Y $< 7 : 0 >$ and UV $< 7 : 0 >$ together provide data pixels in either YUV and RGB colour space formats. The output data transfer is based on a line by line transfer with synchronous pixel read out scheme.

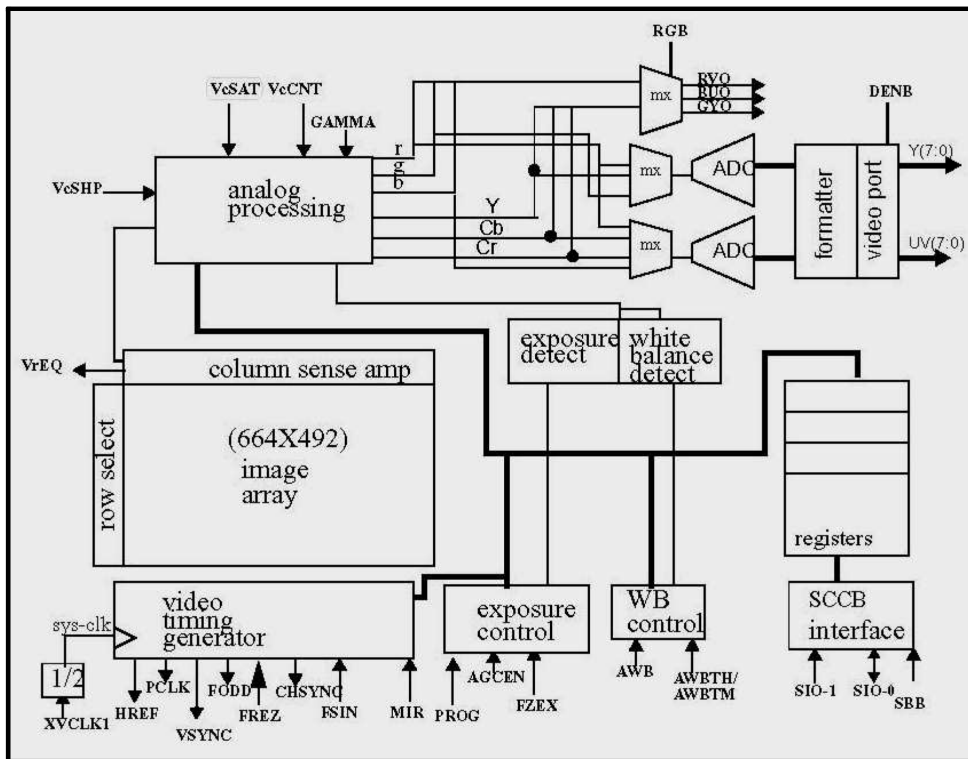


Figure 4.3: CMOS image sensor architecture [36]

The internal architecture of OV7620 is shown in Figure 4.3. The row select determines which row to be sampled and the column sense amplifier produces electric current which corresponds to the illumination of an image. The analog processing unit samples and digitises the analog signals to generate the digital representation signals either in RGB or YUV formats.

The digital output port of the OV7620 offers different type of output sequences. The output sequence can be configured as YUV 4:2:2, YUV 4:1:1, YUV 4:4:4 or RGB in Bayer-filter pattern colour format.

To understand the details of the three controls signals, a PC logic analyser is used.

4.1.2 Image acquisition: implementation

The output controls signals are used to synchronise the output pixel data. The VSYN signal represents the arrival of a new frame. When VSYN and HREF go simultaneously high, it indicates the beginning of an image frame; the first pixel of the first line. The control signals describe in the followings are shown in Figure 4.4.

PCLK is the output pixel clock from the image sensor. A new data is available for every rising edge of the PCLK. The HREF is used to synchronise the rows within an image frame. The HREF goes high at the beginning of each new active row and goes low at the end of each row. All data on the Y and UV output ports are considered valid only when HREF is high. Otherwise, when HREF is low, the data is not within the display window and should not be considered as valid pixels. As a result, the rising edge PCLK together with HREF = 1, indicate that a new pixel is ready to be read.

Before any pixels is being processed, intermediate internal signals are derived for the ease of control and processing. The acquisition module (Figure 4.5) interfaces to the CMOS image digital output port and generates internal signals (PCLK_valid, Y_valid, Row_cnt and Col_cnt), for further processing along the pipeline. The

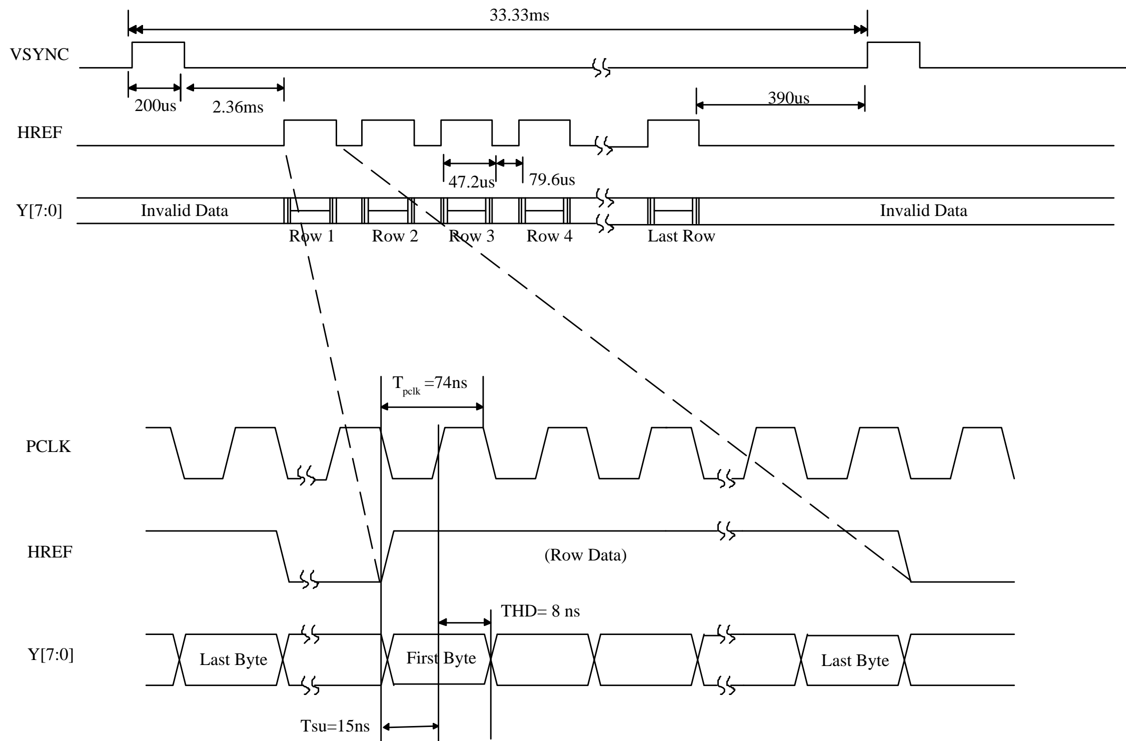


Figure 4.4: Timing Diagram of the control signals

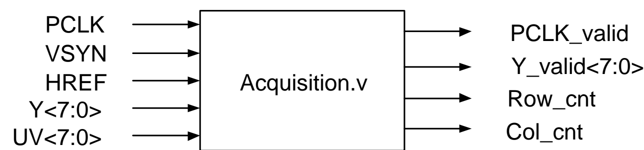


Figure 4.5: Image acquisition block

PCLK_valid is generated based on the input signals, PCLK and HREF. For every rising edge of PCLK_valid, it indicates a new valid pixel on Y_valid bus. A row counter (row_cnt) and a column counter (col_cnt) are generated from PCLK, HREF and VSYN for the purpose of tracking the current active pixel.

The Y_valid only consists of pixels in greyscale format. The colour components are removed in the acquisition module. The acquisition module described in verilog is simulated and synthesized as shown in Figures 4.6 and 4.7 respectively.

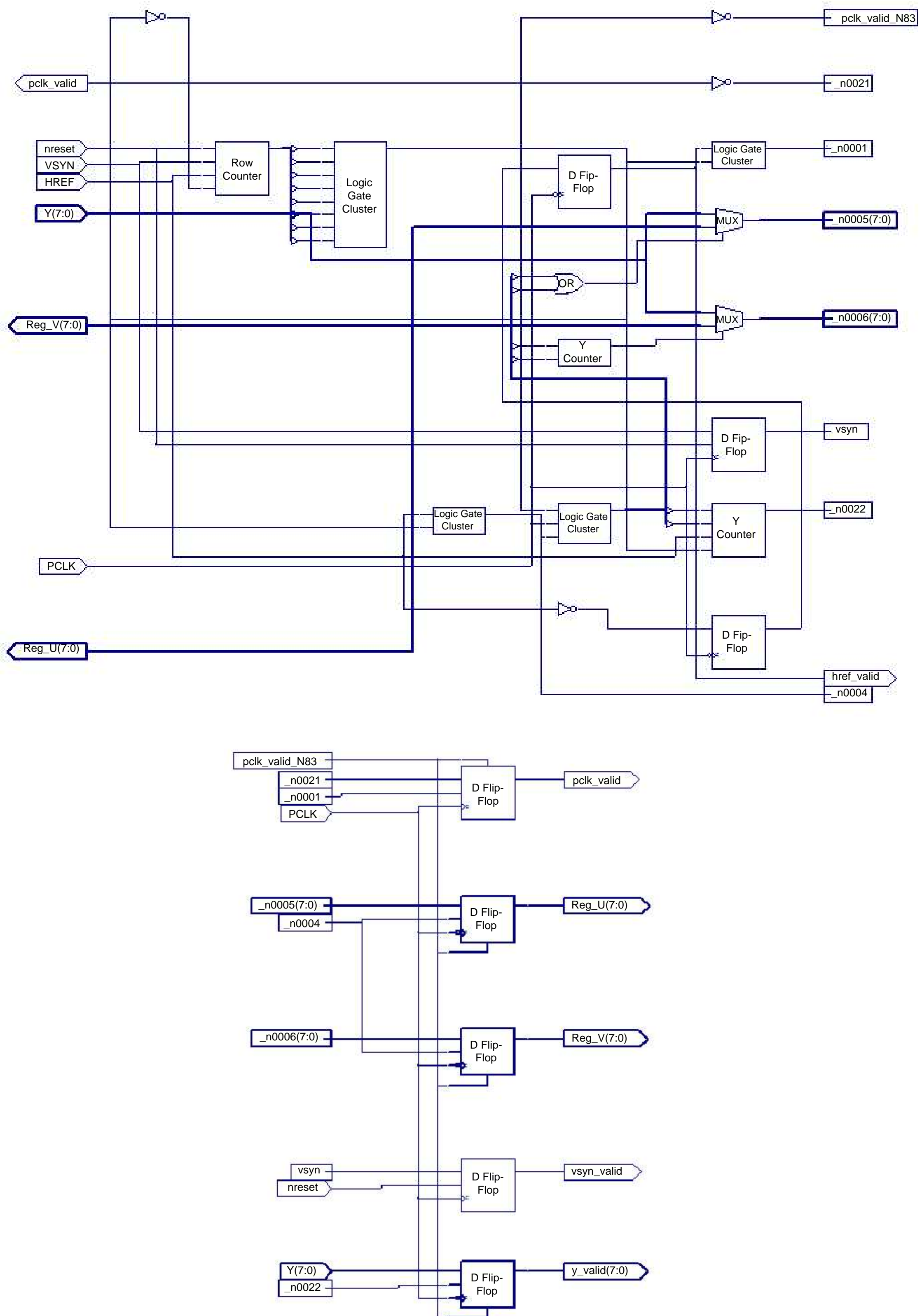


Figure 4.6: Synthesized circuit of the image acquisition block

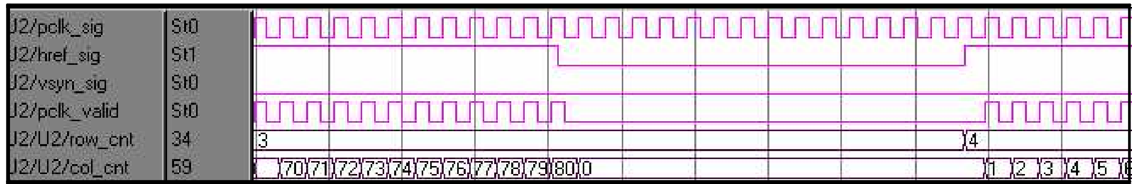


Figure 4.7: Simulation result of the image acquisition block

4.2 Image Compression

4.2.1 Image compression: concept

As mentioned in Chapter 1, memories occupy a large part of the chip area in most embedded multimedia systems.

Image compression addresses the problem of reducing the amount of data required to represent a digital image. Moreover, compressed image also helps to reduce the transmission bandwidth. The compressed image stored in the memory, is later read and decompressed to reconstruct the original image or as an approximation of the original image.

Digital image compression is commonly divided into two basic classes. They are lossy and lossless compression. Lossy compression is often used where the loss of certain information within the image is acceptable. Lossless compression is a technique to compress data where no data are loss after decompression.

All digital image compression techniques are based on the exploitation of information redundancy that exists in most digital images Most compression techniques are based on the removal of such redundant data [48] [47].

A relatively simple solution is to encode the differences between successive samples rather than the samples themselves. Since differences between samples are expected to be smaller than the actual sampled amplitudes and fewer bits are required to represent the differences. In this case, the mathematical representation is expressed in (4.1) .

$$e(n) = s(n) - s(n-1) \ , \tag{4.1}$$

where $s(n)$ is the current sampled sequence and $e(n)$ is the amplitude difference between the current and previous samples.

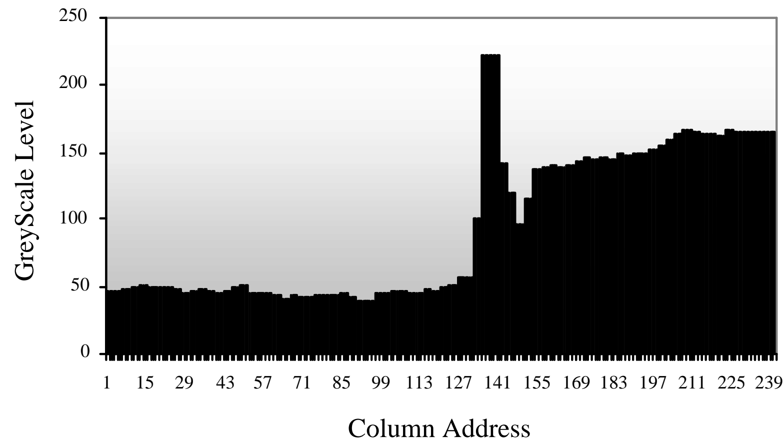


Figure 4.8: Pixel amplitude of a single line

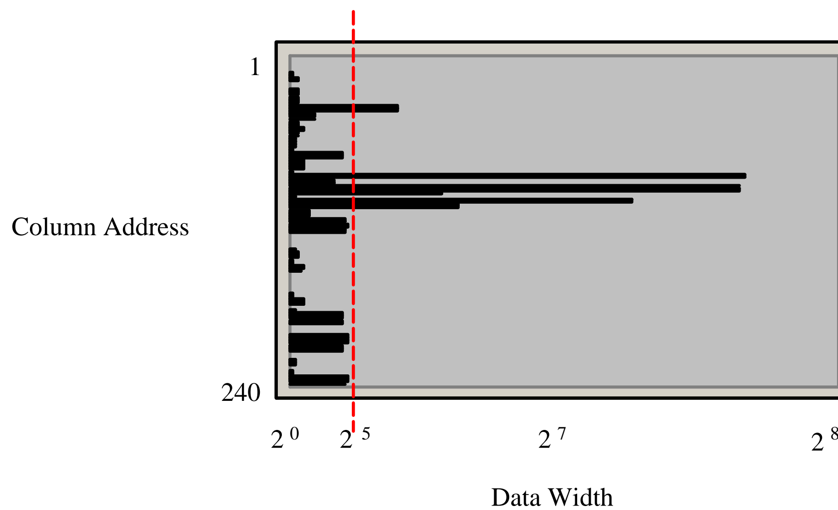


Figure 4.9: Number of bits to represent compressed pixel

To illustrate the spatial correlation of an image, the pixel greyscale level is plotted against the location of the pixels. Figure 4.8 shows that, given a nature

image, it exhibits the properties where the difference in the grayscale level between neighbour pixels is small. As a result, there is a reduction of data width required to represent a pixel. Figure 4.9 shows the number of bits required to store a single line after compression. For this example, 90.02% of the pixels can be represented with a data width of 5 bits instead of 8 bits. This is especially useful for images with low frequency contents. By exploiting this property, $e(n)$ are stored in the memory instead of $s(n)$. To extend this idea to the entire image, only the absolute value of the first pixel is stored in memory i.e., $s(n = 0)$.

Due to the spatial correlation of neighbouring pixels, the average change in amplitude between any neighbouring pixels is relatively small. Consequently, an encoding scheme that exploits the redundancy in the samples, results in a lower bit rate for memory storage.

To compute $e(n)$, a subtraction module can be used at the compression stage, after which the decompression can be achieved by using an adder. The subtraction /addition module can be implemented using the Xilinx Core Generator. It provides a customisable core for the purpose of addition and subtraction in a single module. In additional, this module operates on both signed and unsigned data types.

Alternatively, an approximate of $e(n)$ can be computed using the properties of an exclusive OR operations (Table 4.1).

$x \oplus 0 = x,$
$x \oplus 1 = \sim x,$
$x \oplus x = 0,$
$x \oplus y = y \oplus x,$

Table 4.1: Properties of exclusive OR operations

Hence, $e(n)$ can be approximated as

$$s(n) \oplus s(n-1) = e(n) \ , \tag{4.2}$$

where $e(n)$ is the reduced data type and $s(n)$ is the current sampled pixel. Consequently, the original pixel can be recovered from $e(n)$ as follows

$$
\begin{aligned}
s'(n) &= s(n-1) \oplus e(n) \\
&= s(n-1) \oplus s(n) \oplus s(n-1)) \\
&= s(n) \oplus (s(n-1) \oplus s(n-1)) \\
&= s(n) \oplus 0 \\
&= s(n) \ ,
\end{aligned}
$$

where $s'(n)$ is the pixel recovered.

4.2.2 Image compression: implementation

Figure 4.10 shows the block diagram of the compression and decompression modules. The resultant implementation should be carried out with the considerations of area and speed. The compressed image is stored in memory which is later decompressed by another XOR module.

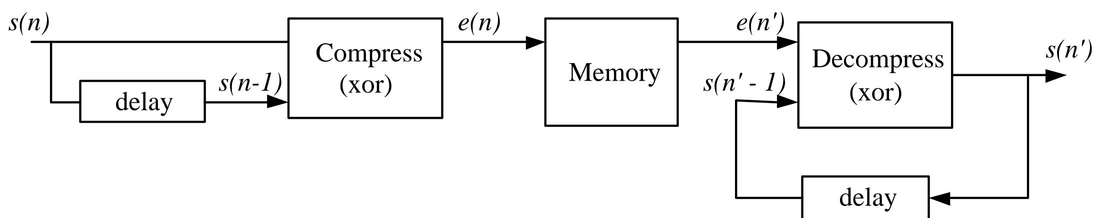


Figure 4.10: Block diagram of Compression and Decompression

Two images with low (Figure 4.14) and high frequency (Figure 4.15) content are analysed. The image with high frequency content shows a better histogram distribution.

Figures 4.14(a) and 4.14(c) show the uncompressed original image and compressed image using the XOR operation respectively. The histograms for both the images are shown in Figures 4.14(b) and 4.14(d) respectively.

From Figure 4.14(d), it is observed that when $e(n) = 0$, it has the highest frequency. In this particular sample, there are 14.39% of the adjacent pixels having the same intensity value.

The XOR compression and decompression are described using Verilog HDL and simulated in ModelSim (Figure 4.11). The synthesized logic gates are shown in Figures 4.12 and 4.13. From the synthesis report, it is noted that the implementation only consumes 9 slices and 16 Flip-Flops, with a timing delay of 6.042ns.

Figure 4.11: Simulation results

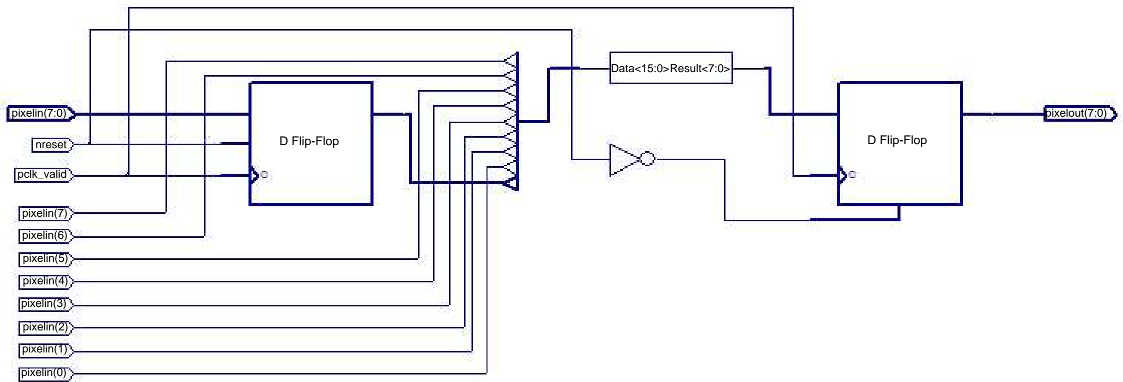


Figure 4.12: Synthesized circuit of XOR compression module

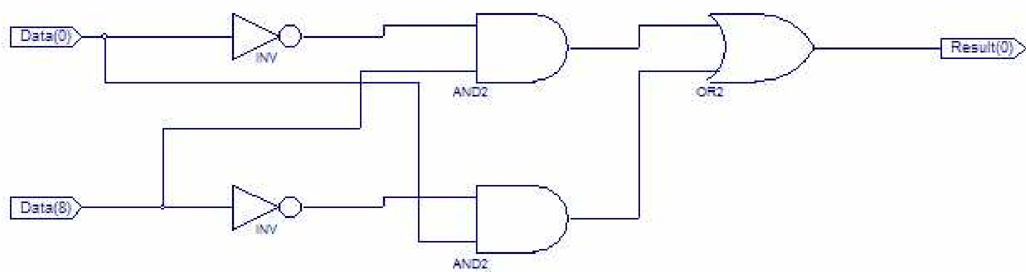


Figure 4.13: XOR gate

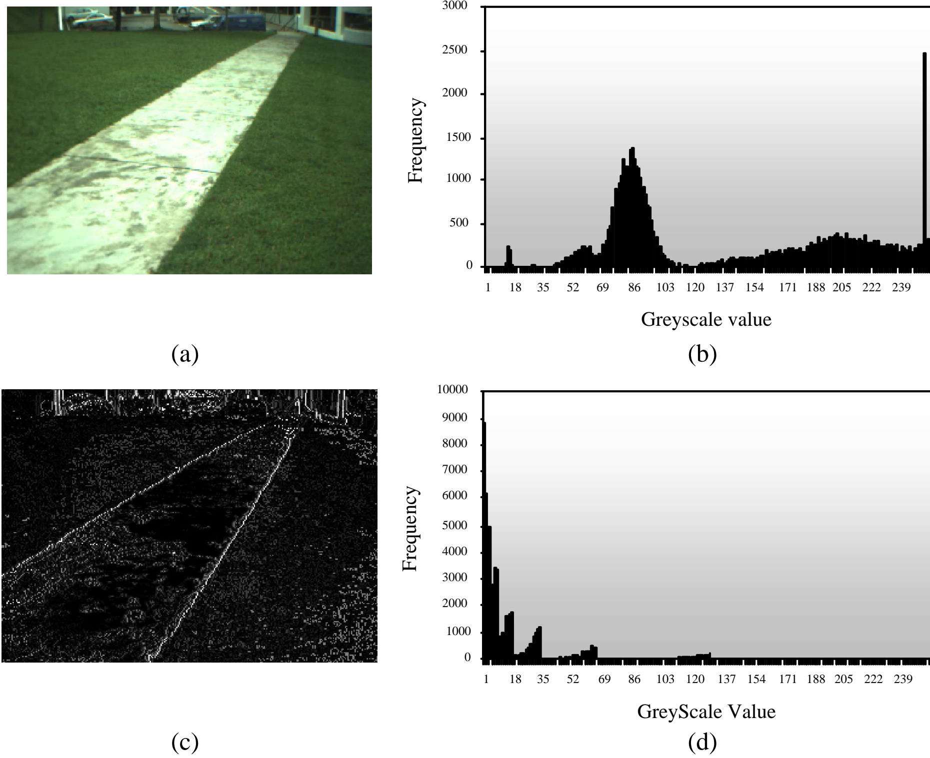


Figure 4.14: Histogram of image with low frequency content

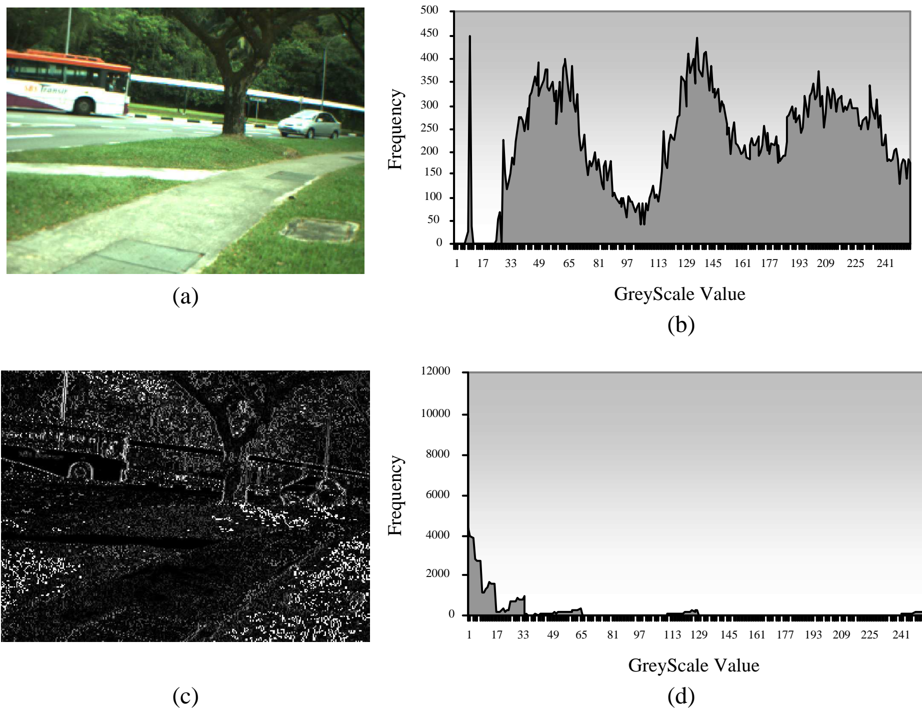


Figure 4.15: Histogram of image with high frequency content

4.3 Image Buffering

Prior to any actual processing being performed, buffering the input image is necessary.

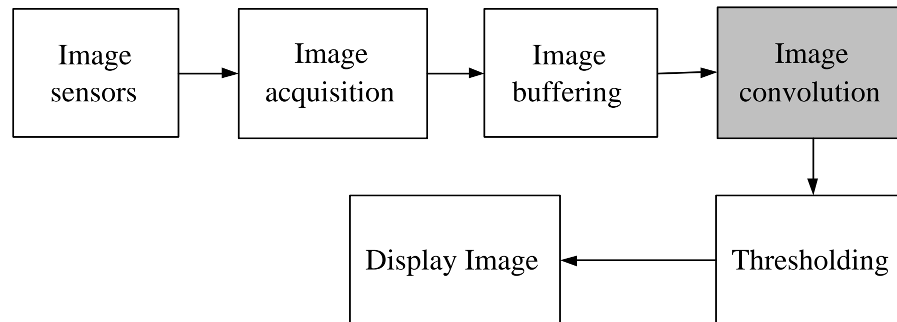


Figure 4.16: Image buffering stage

Image buffering is often used to compensate for a difference in the rate of flow of the image data between processes. Image buffering is used to synchronize the image acquisition module and the computation of image data. With reference to Figure 4.16, buffering is necessary to provide or store neighbourhood pixels for the subsequent image convolution process.

4.3.1 Image buffering: theory

The 3 x 3 image convolution function requires nine pixels to be sampled for the computation of 1 output pixel. The conventional approach is to buffer the entire frame into a SRAM and retrieve nine pixels for each convolution computation. As a result, nine read access are required for the calculation of a single pixel. This section discusses the method of memory reuse. A 3 x 3 convolution window performed on a 5 x 4 image (Figure 4.17) is used as an example.

In order to reduce the frequency of memory access and total memory space required, each memory location is reused. The principle of data reuse is to efficiently utilize the available memory space by reusing its storage as much as possible. By

W11	W12	W13		
W21	W22	W23		
W31	W32	W33		

Figure 4.17: A 3 x 3 convolution mask on a 5 x 4 image

exploring the lifetime of a data variable, each memory location is allocated with an occupancy time related to a particular data variable. When these locations are no longer used by the data variable for any read or write access, these data variables are said to have reached the end of lifetime. Hence, these locations can be reused by new arrived data variable. As a result, the exploration of memory location to be reused, resulting in the reduction of physical memory space is possible. The reduction strategy is based on the concept of data transformation presented in [54] [55].

The buffering of an input image for the purpose of convolution function is used for the study. The lifetime of a data pixel and the data representation of pixels occupancy are illustrated in Figure 4.18.

The lifetime of each pixel is represented as a producer and consumer approach, wherein the producer is associated with the image acquisition processes and the consumer is associated with the convolution processes. Figure 4.18 illustrates the writing of a new pixel is written to the memory during each clock cycle. The first read cycle begins only after the last pixel is written to the memory. After 4 rows of pixels are written, the convolution process reads nine pixels at the same time to perform a 3 x 3 image convolution function. This process is repeated for (N - 2) x (M - 2) times. In this example, the total memory space required is 20 data pixels.

Figure 4.19 illustrates that the consumer is brought closer to the producer. Such technique reduces the lifetime of all data variable by seven clock cycles. Immediately after the 14th pixel is written to the memory, the 1st pixel is consumed or read.

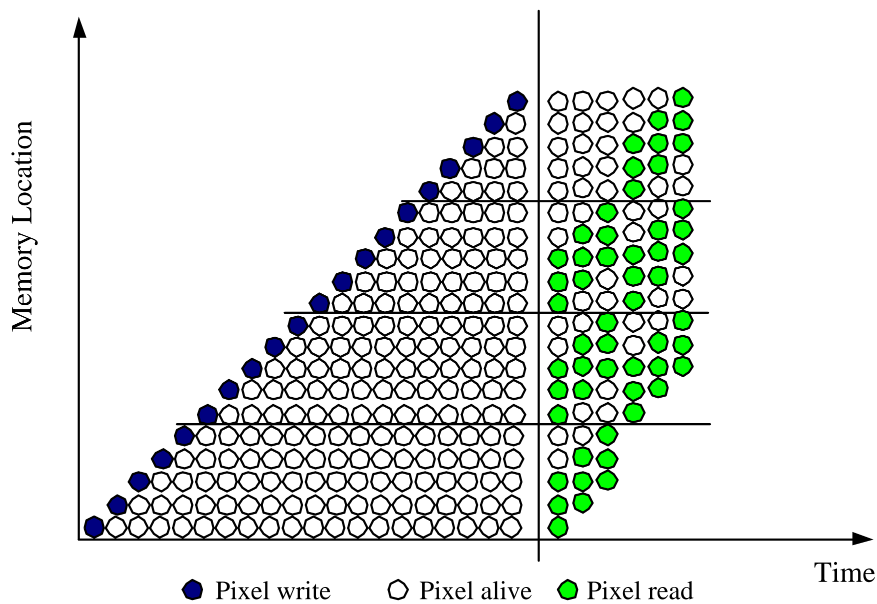


Figure 4.18: Producer and consumer of pixels before transformation

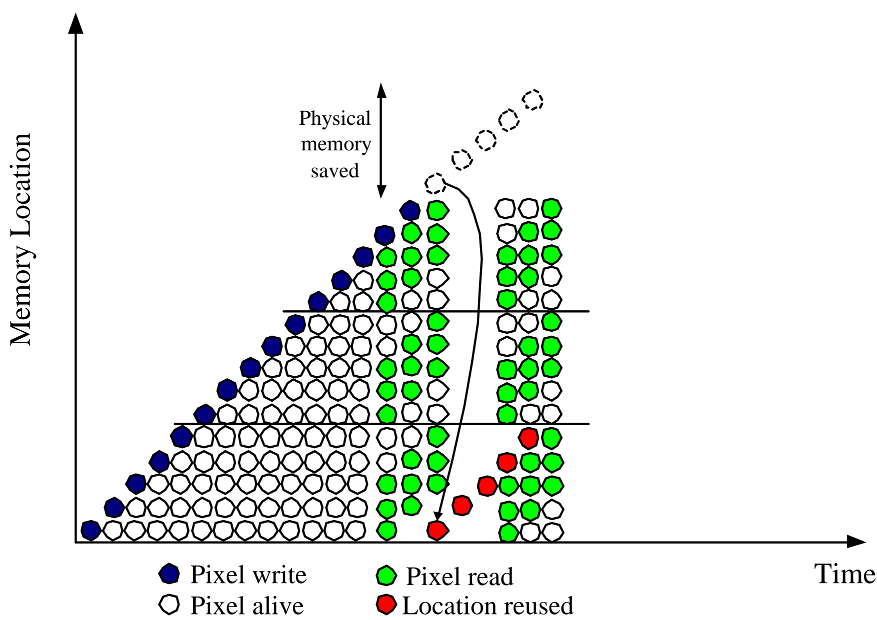


Figure 4.19: Producer and consumer of pixels after transformation

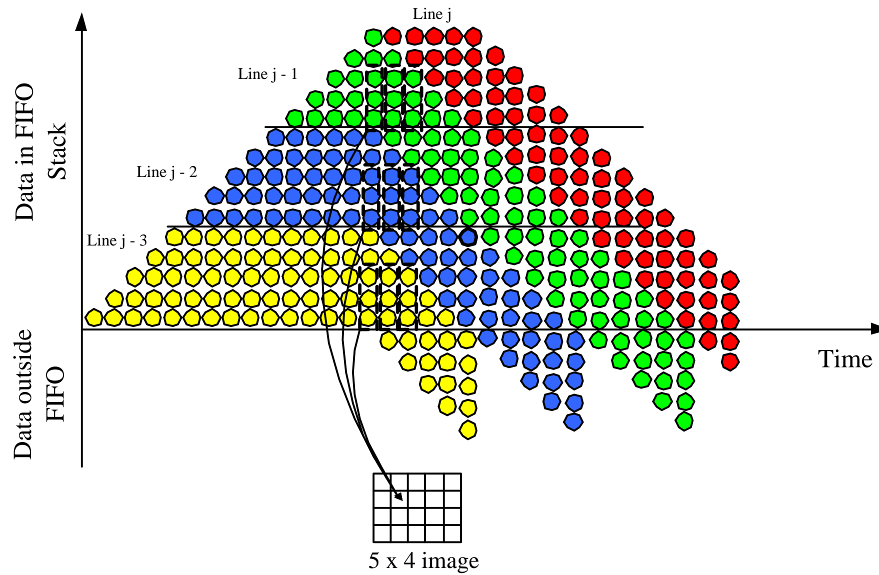


Figure 4.20: Buffering using FIFO

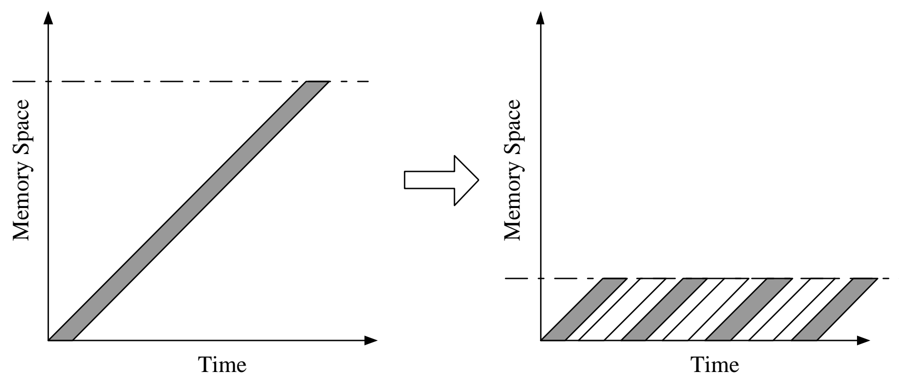


Figure 4.21: Reduction of memory space after data reuse

After all pixels in the first row are read, these memory locations are free to be reused. As a result, the 4th row of pixels has reused the first five data locations. From this example, it shows that a buffer size of only three lines are required to buffer the input image for a 3 x 3 image convolution.

Instead of using Static RAM for implementing the image buffer, FIFO RAM can be used. The FIFO RAM offers several advantages over the Static RAM in certain areas. The simplicity of FIFO RAM allows data to be accessed without the hassle of address decoding. In additional, data placed on top of the stack are retrieved at the bottom of the stack. As a result, data that reaches the end of its

lifetime is automatically removed from the bottom of the stack after a data is being read, freeing memory location for other data variables. For this reason, the reuse of such free memory location achieves overall memory reduction.

The FIFO RAM is allowed to be filled before any data is being read (Figure 4.20). After the FIFO RAM is filled, for each pixel placed on top of the stack, a pixel is read from the bottom of the stack at the same clock cycle. As the FIFO RAM remains full, it performs a line delay function.

Figure 4.21 illustrates the reduction of total memory space required after the reuse of free memory locations.

In theory, the nine pixels forming the 3 x 3 window are sampled at fixed locations as illustrated in Figure 4.20. However, in practice, a typical FIFO RAM can only be read at the end of the stack. Thus, such method is not possible. However, the next section discusses on how this can be made possible.

4.3.2 Image buffering: implementation

A revised solution is proposed to access the nine pixels at the designated memory locations. With reference to Figure 4.22, a single FIFO RAM can be broken up into two separate FIFO RAM together with 9 registers. The 3 registers, W31, W32 and W33 replace the 3 memory spaces from the FIFO1. Similarly, the other 3 registers replace 3 memory spaces in FIFO2. The size of both FIFO1 and FIFO2 are determined by the width of the image as (4.3).

$$S_{fifo1} = S_{fifo2} = N - 3 \; , \tag{4.3}$$

where S_{fifo1} and S_{fifo2} are the sizes of the FIFO RAM required and N is the image width. Such architecture maintains a single logical FIFO RAM for the functionality discussed in the earlier section and provides a method to sample the designated memory locations.

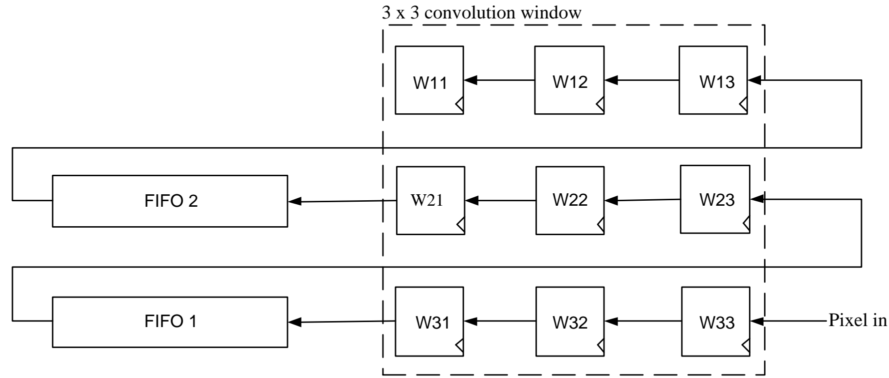


Figure 4.22: Convolution window using registers

With reference to Figures 4.20 and 4.22, the convoluted output pixel is expressed as shown in (4.4).

$$r_{i-1,j-1} = \begin{bmatrix} h_{i-2,j-2} & h_{i-1,j-2} & h_{i,j-2} \\ h_{i-2,j-1} & h_{i-1,j-1} & h_{i,j-1} \\ h_{i-2,j} & h_{i-1,j} & h_{i,j} \end{bmatrix} \otimes \begin{bmatrix} M_1 & M_2 & M_3 \\ M_4 & M_5 & M_6 \\ M_7 & M_8 & M_9 \end{bmatrix} \tag{4.4}$$

Where $h_{i,j}$ is the pixel input, $r_{i-1,j-1}$ is the resultant pixel and M is the convolution mask. It is interesting to note that, for every pixel produced at location (i,j), an output pixel at $(i+1,j+1)$ is produced.

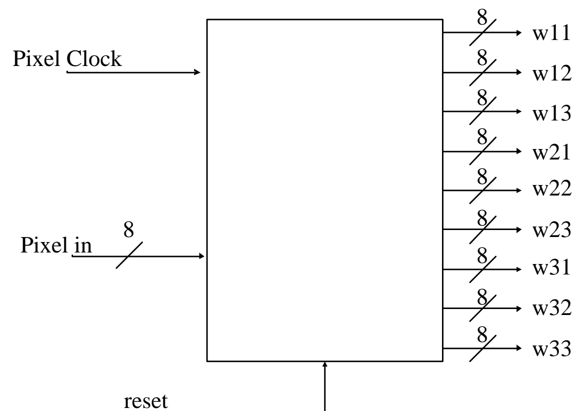


Figure 4.23: Image buffer module

The image buffer is designed and implemented as a module with the various input/output ports and control signal as shown in Figure 4.23. It uses a dedicated

Block RAM available in the Xilinx Spartan 2eS300. The block RAM is accessed through an FIFO module generated by Core Generator, a program distributed by Xilinx Inc. The Core Generator makes use of the block RAM within the FPGA and embedding logic cell to implement an asynchronous FIFO RAM.

The Spartan 2eS300 chip has a total of 8 Kbytes spread across 16 blocks of embedded RAM. Each block of RAM is 512 bytes. The 317 bytes FIFO RAM is implemented using an entire single block RAM of 512 bytes. A counter is used to count the number of memory locations occupied by data. If the counter counts 317 times, it asserts a FIFO full signal indicating that the logical 317 FIFO is full. Upon receiving this signal, the 317 FIFO performs the function of shifting pixels in and out of the FIFO RAM at every clock cycle.

For every clock cycle, a new pixel is shifted into the W33 register. In the same way, the data in W33 register is shifted to the remaining 2 registers and subsequently written into the FIFO1. The shifting technique can be viewed as a common technique used in parallel to serial converter in many serial communication systems.

The image buffer is generated using two FIFO RAM together with some control logic and registers. The design is synthesized, simulated and tested with convolution function in the Chapter 5. Figures 4.24 and 4.25 show the synthesized design of the image buffer. The image buffer consists of nine key registers that are used in the convolution window. Four comparators are used to generate control signals for read and write sequence to the two FIFO RAM.

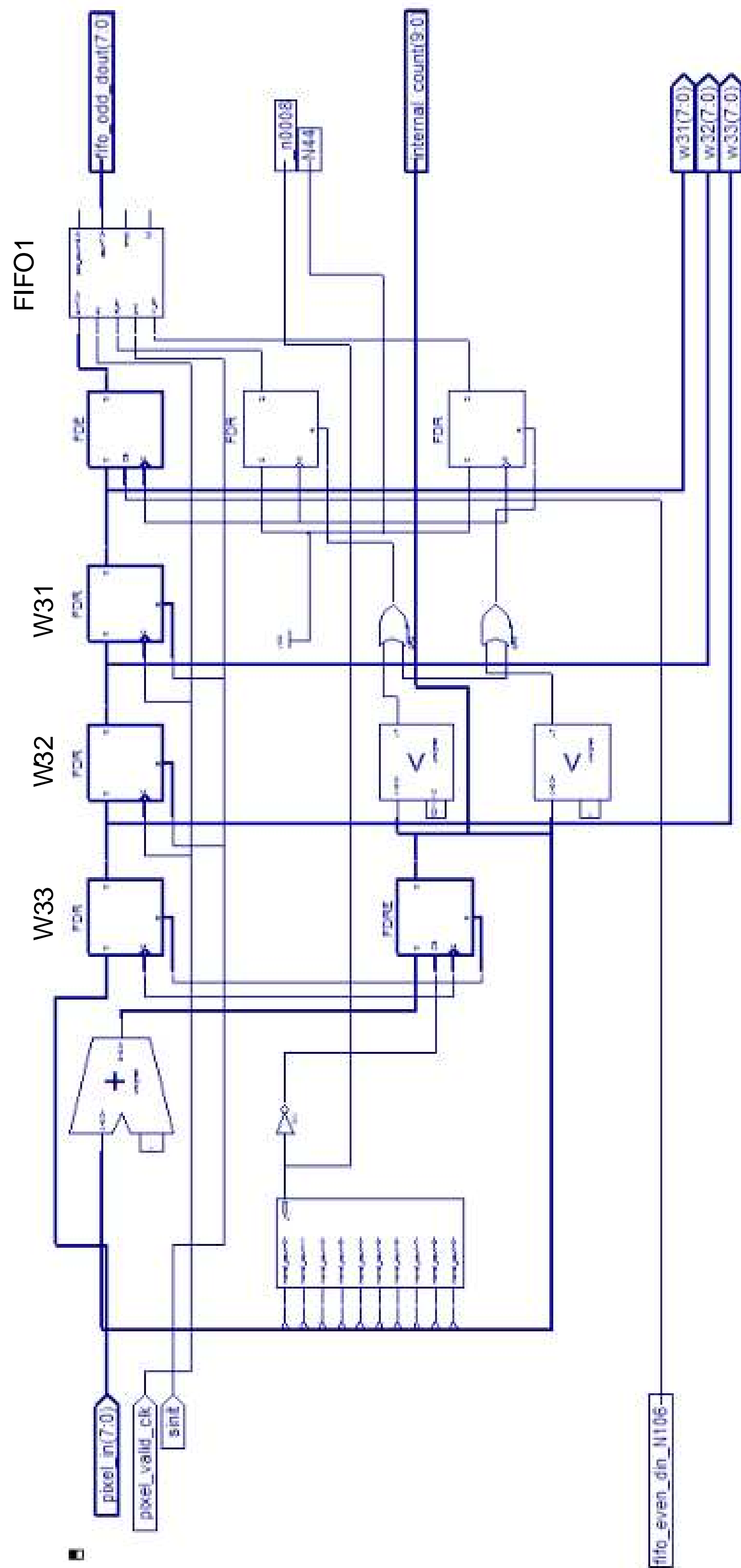


Figure 4.24: Synthesize result of Image buffer (Part 1)

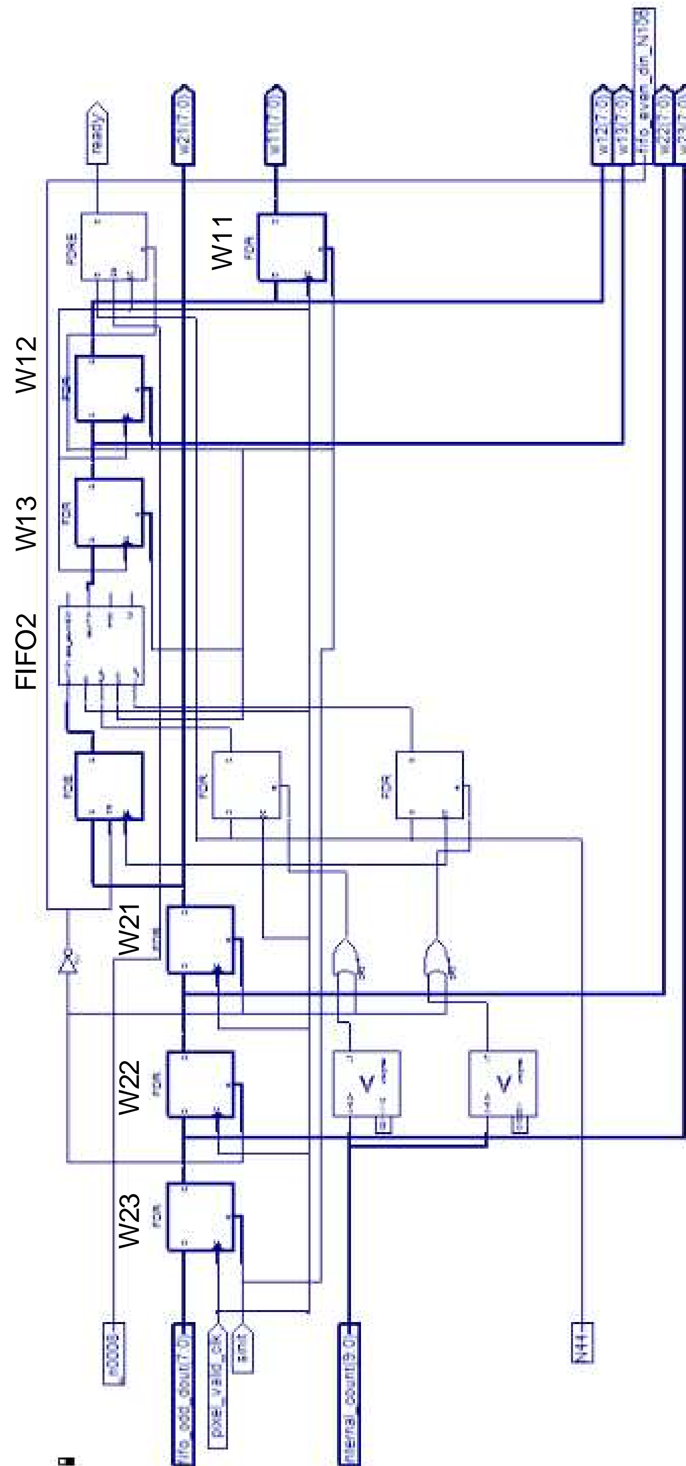


Figure 4.25: Synthesize result of Image buffer (Part 2)

4.4 Convolution Theory

Image convolution is used to enhance certain features, de-enhance the rest of the features, identify edges, smooth out noise or discover previously known shapes in an image [49]. It is one of the most commonly used processes in image processing.

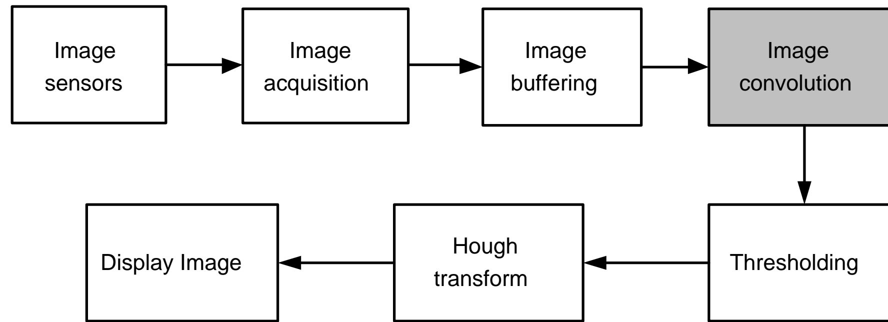


Figure 4.26: Image convolution stage

Convolution process is a computationally demanding process to be carried out in real-time. The operation often compute a set of neighbour pixels with a predefined convolution mask to produce a single pixel.

Convolution in an image can perform the function of image filtering. The convolution operator is often called filter or kernel. The size of typical convolution filters are 3 x 3 and 5 x 5 pixels. Several convolution operators are available in the literature for performing specific functions. A few common convolution operators are "High pass", "Low pass", "Laplacian", "Median" and "Sobel" Roberts. Figures 4.27 illustrates the convolution operation of a 3 x 3 convolution filter with an N x M image.

The output of the convolution function is represented as shown in Equation 4.5.

$$r_{i,j} = \sum_{k=1}^{9} f(k)\ h(k)\ , \tag{4.5}$$

where $r_{i,j}$ is the result of the convolution, $f(k)$ is the convolution mask and $h(k)$ is image window. For example, the resultant pixel is computed as

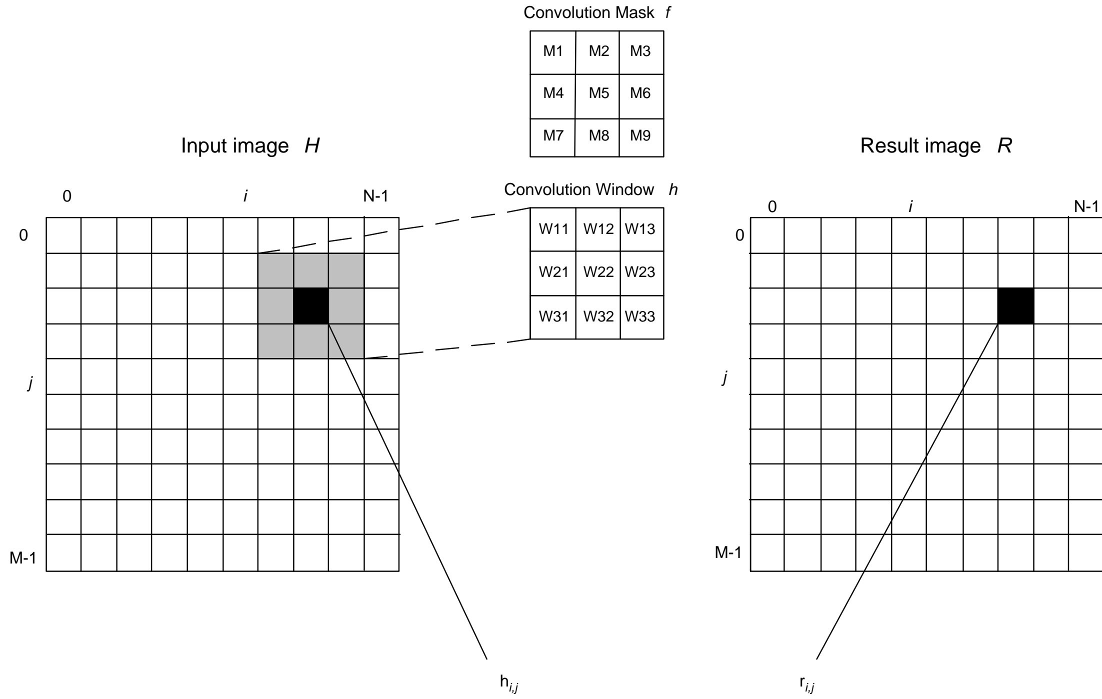


Figure 4.27: Image convolution

$$
\begin{aligned}
r = & \quad W11M1 + W12M2 + W13M3 \\
& + \; W21M4 + W22M5 + W23M6 \; , \\
& + \; W31M7 + W32M8 + W33M9
\end{aligned} \tag{4.6}
$$

Typically, the convolution function is performed using a predefined convolution mask. The convolution window h is superimposed upon the input image H, commencing at the origin. Convolution is performed using Multiply and Accumulate (MAC) operation. Each element in the convolution window is multiplied by the corresponding element in the convolution mask f. The nine results are summed together and the final value is written to a resultant image R, at the position $r_{i,j}$. To compute the next $r_{i,j}$. , the window is shifted to the next pixel. This process is repeated until all pixels are computed to produce an output image.

In the next chapter, a FPGA implementation of Parallel Architecture is presented. The image buffer discussed in this chapter provides an overlaying convolution window for the subsequent convolution function.

Chapter 5

FPGA Implementation of Parallel Architecture

This chapter discusses the concept and implementation of thresholding, Low Pass filter and edge detection algorithm. The design is realised in a parallel architecture with the considerations of computational efficiency. Figure 5.1 shows various stages of processing within the image processing system.

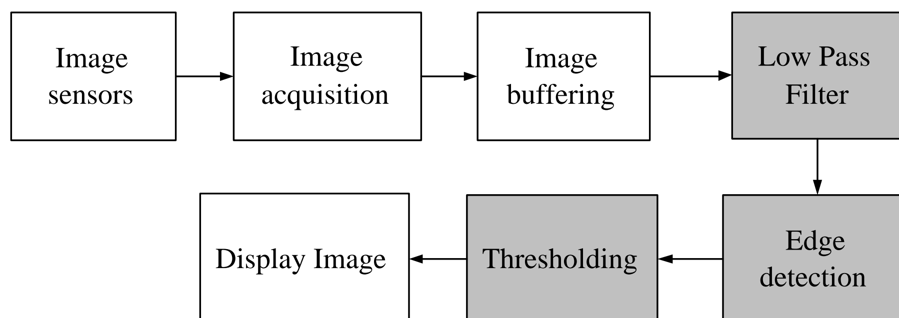


Figure 5.1: Image processing stage

5.1 Edge Detection Theory

Edge detection is by far the most common approach for detecting meaningful discontinuities in grey level. The reason is that isolated points and thin lines are not frequent occurrences in most practical applications [47]. The principle of edge detection is that an edge is defined where there is a steep intensity gradient in the image. An edge can be defined as a boundary between two regions with relatively distinct greyscale properties. Hence, the gradient of the pixel at the edge provides some indication on the presence of an edge pixel.

With this theory, the derivatives of the intensity values across the image are calculated to determine the maximum intensity derivation. The point with the maximum derivation is said to be an obvious edge. Subsequently, all other edges can be determined by a predefined threshold. Thus, the image will consists of only edges and non-edges pixels, represented in a binary image format.

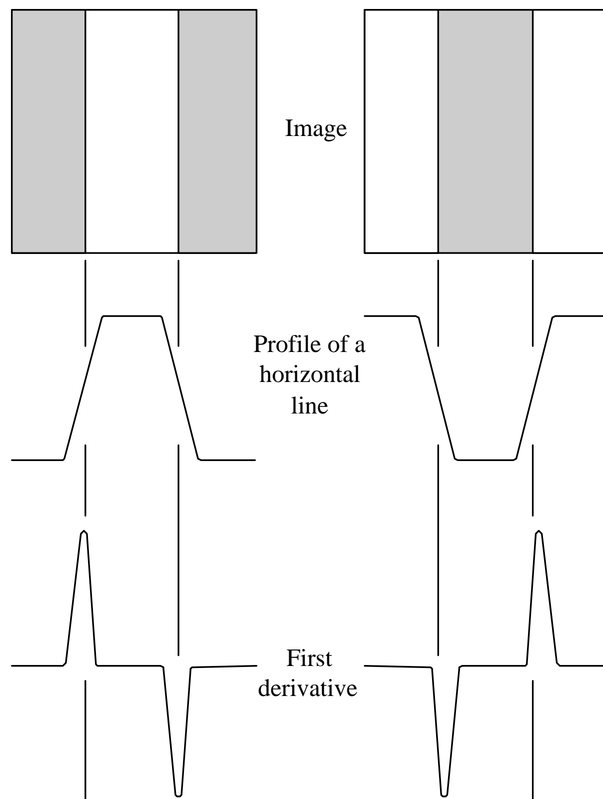


Figure 5.2: Image intensity level derivatives

Figure 5.2 further illustrates the concept of detecting edges through the derivatives of the intensity values across the image. The image shows that there is an abrupt transition of dark grey level to white grey level. However the profile of this image is modelled as a smooth change in the grey level. This is due to the fact that any natural images captured has a gradual change in intensity with respect to its surrounding pixels.

Subsequently, the first derivative can be obtained from the profile of the horizontal line. It is also noted that, the leading edge of a profile transition results in a positive derivative, a trailing edge results in a negative derivative and a constant grey level results in a zero derivative.

The detection of edges or contours from a two dimensional image is performed by convolution and moving window operations, conceptually combined into computations that determine the magnitude of contrast changes [56].

W11	W12	W13
W21	W22	W23
W31	W32	W33

Figure 5.3: Convolution window

$h_y(i,j)=$

-1	-1	-1
0	0	0
1	1	1

(a)

$h_x(i,j)=$

-1	0	1
-1	0	1
-1	0	1

(b)

Figure 5.4: Prewitt operator

Edge detection algorithms uses the theory of convolving a moving window (Figure 5.3) with a set of operator masks. The Prewitt operator (Figure 5.4) and Sobel operator (Figure 5.5) are two commonly known edge detection operators.

$h_y(i,j)=$

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

(a)

$h_x(i,j)=$

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

(b)

Figure 5.5: Sobel operator

The Prewitt operator demonstrates the simple concept of the first derivative. It provides an equal weight to the pixel difference which is horizontally or vertically adjacent to the origin. From Figure 5.4, it can be seen that $h_y(i,j)$ returns the maximum rate of change in y component of the image. Consequently, a zero difference in the y component results in zero gradient.

The Sobel edge operator is recognised as one of the best and yet simple to implement algorithm [24]. The coefficients of the mask are derived to extract features with high edge contrast. The Sobel operator works by applying more weight to the central pixel differences as opposed to just the horizontal or vertical pixels used in the Prewitt's operator. The Sobel operators also have the advantages of providing both differencing and smoothing effect. From Figures 5.4, 5.5(a) and 5.5(b).

$$\begin{aligned}\frac{d_f}{d_x} &= G_x(i,j) \\ &= (-w11 - 2(w12) - w13) + (w31 + 2(w32) + w33) \quad ,\end{aligned} \tag{5.1}$$

and

$$\begin{aligned}\frac{d_f}{d_y} &= G_y(i,j) \\ &= (-w11 - 2(w21) - w31) + (w13 + 2(w23) + w33) \quad ,\end{aligned} \tag{5.2}$$

where the gradient magnitude and orientation are given as

$$\begin{aligned} G_m(i,j) &= \sqrt{G_y(i,j)^2 + G_x(i,j)^2} \\ &\approx \mid G_y(i,j) \mid + \mid G_x(i,j) \mid \quad , \end{aligned} \tag{5.3}$$

$$G_\theta(i,j) = \tan^{-1}(\frac{G_y}{G_x}). \tag{5.4}$$

5.2 Proposed Parallel Architecture for Edge Detection

Two dimensional convolution is often characterized by a large amount of data together with small neighbourhood operators. The convolution process consists of multiply and addition operations. It is also known that custom architectures are more efficiently used than instruction-set architectures when the algorithm processes large amount of data with high degree of regularity [60] [58] [59]. These reasons have led to the proposed architecture to exploit computational parallelism.

The architecture is designed to match the speed of the image sensor's frame rate. For instance, the image sensor maximum frame rate is 30 fps. Hence, the computation of an entire image must be less than or equal to 33.33ms.

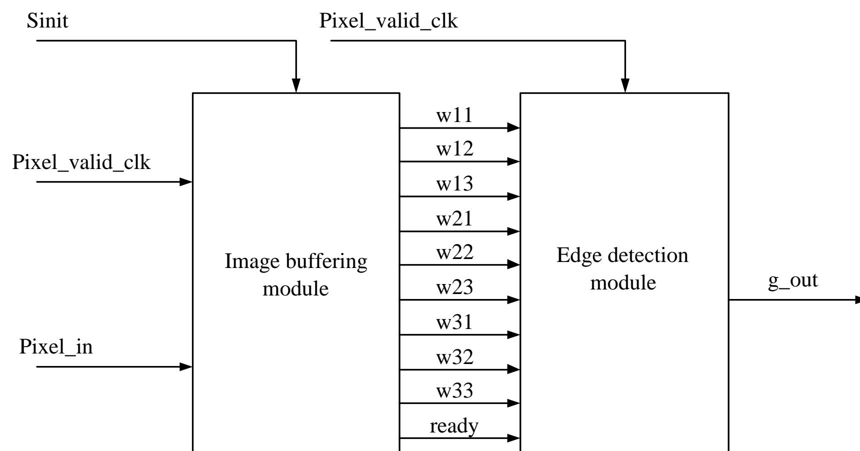


Figure 5.6: Acquiring nine pixels from image buffering module

The image buffer module discussed in the previous chapter provides an overlaying window for the operations of convolution function. The edge detection module receives nine input pixels and performs a parallel computation as shown in Figure 5.7.

Considering the Sobel operation expressed in (5.5), (5.6) and (5.7), each of the G_x and G_y operation consists of 5 signed additions and 2 unsigned multiplications. By grouping similar operations together, 1 multiplication operation is reduced. Hence, a total of 11 additions and 4 multiplications are required to obtain the gradient magnitude of a single pixel.

Based on (5.5), the equivalent is implemented in hardware as shown in Figure 5.7. Similarly, the architecture for G_y and G_out are shown in Figure 5.7 and 5.8 respectively.

The image buffering together with the edge detection module forms the convolution operation that is implemented in hardware architecture. All six pixels are computed in a parallel configuration to produce a single output pixel value, $|G_x|$.

$$\begin{aligned} G_x(i,j) &= (-w11 - 2(w12) - w13) + (w31 + 2(w32) + w33) \\ &= 2(w23 - w21) + (w13 - w31) + (w33 - w11) \quad , \end{aligned} \tag{5.5}$$

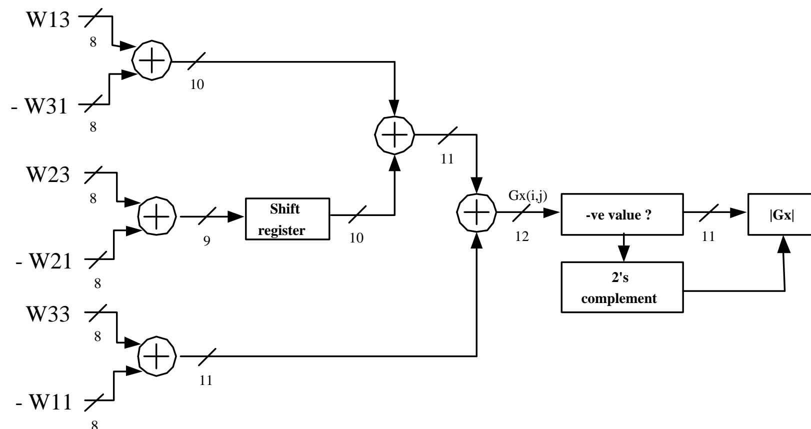


Figure 5.7: Architecture of G_x

$$\begin{aligned} G_y(i,j) &= (-w11 - 2(w21) - w31) + (w13 + w23 + w33) \\ &= 2(w32 - w12) + (w31 - w13) + (w31 - w11) \quad , \end{aligned} \tag{5.6}$$

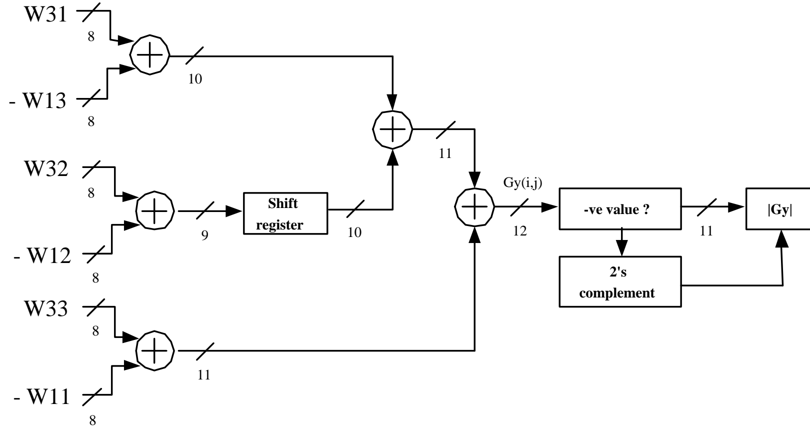


Figure 5.8: Architecture of G_y

$$G_{out} = |(G_y i, j| + |G_x i, j| \tag{5.7}$$

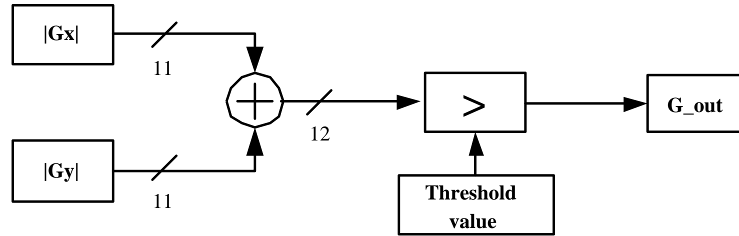


Figure 5.9: Architecture for gradient magnitude and thresholding

The G_x component computation is based on 6 different inputs. There are 9 input pixels from the Image buffering module buffer of which only 6 pixels are used in this module. This is due to the 3 zero coefficients in $h_x(i,j)$.

With reference to Figure 5.7, the (w23 - w21) signed arithmetic operation produces a 9 bit result. This value is subsequently multiplied by 2 to produce a 10 bit pixel values. The multiplication is implemented using a bit-wise shift operation instead of a multiplier. A bit wise shift operation in digital logic implementation

is done simply by rewiring the 9 bit signed value to 10 bit signed value. With this method, there is no logic gate delay. Nevertheless, wiring delay is still accountable. By replacing the multiplier with shift operation, it is able to achieve higher speed and smaller circuit area design compared to the multiplier implementation.

The sum of (2(w23) - w21)+ (w13- w31) together with (w33- w11) produces a 12 bit result of $G_x(i, j)$. Although the (w33- w11) sum operation produce a 9 bit value, it is deliberately assigned to produce an 11 bit result. This is to match the summation operation with another 11 bit operand, which produces a 12 bit pixel value.

The magnitude of G_x is computed based on the sign of G_x. If the MSB (Most Significant Bit) of G_x is negative, it is converted to positive by applying a 2's complement operation which produces $|G_x|$. Since $|G_x|$ is an unsigned register, the sign bit is not needed and the resultant $|G_x|$ is truncated to an 11 bit register.

In the same way, the architecture for $|G_y|$ is identical. Two identical architectures are used for the concurrent computation of $|G_x|$ and $|G_y|$ in a single clock cycle.

The output image is threshold to a predefined value to produce a binary image that consists of only edge data pixels. This edge information is then used for further feature extraction.

The concept and architecture designed are first tested in Visual C/C++ environment before the actual coding of Verilog HDL. Figure 5.10 shows the various image processing algorithms applied to a 320 x 240 colours image. Specifically, thresholding, edge detection, image compression and image decompression are demonstrated.

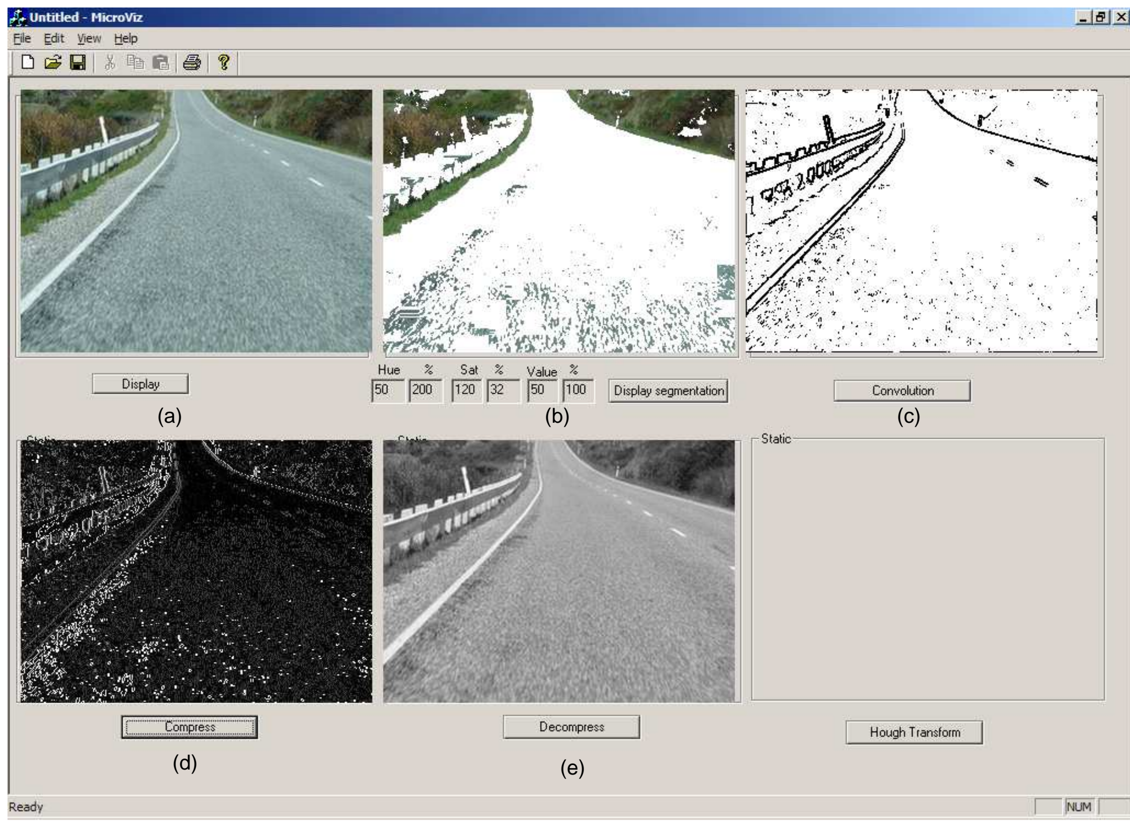


Figure 5.10: Simulation of architecture using Visual C/C++

5.3 Thresholding

Thresholding is one of the most important approaches in image segmentation. It is a process to convert greyscale images to binary images. Typically, a greyscale image is threshold to obtain a binarised version of the image which consists of only foreground and background.

To obtain a binary image which consists of only foreground and background pixel, a threshold value T, is used to partition the image into pixels with just two values, such that

$$If\ f(x,y) \geq T\ then\ g(x,y) = 1$$

$$If\ f(x,y) < T\ then\ g(x,y) = 0$$

where the image $g(x,y)$ denotes the binarised version of image $f(x,y)$.

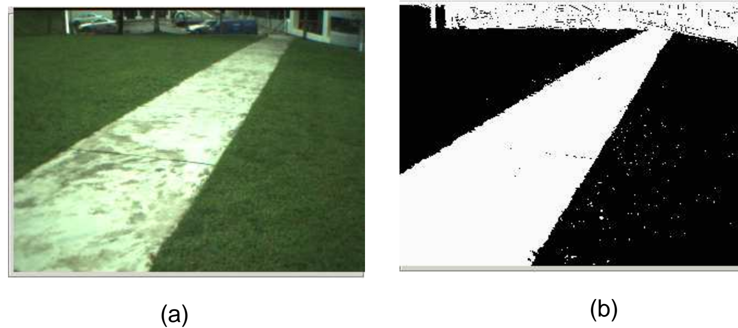


Figure 5.11: Thresholding

Figure 5.11(a) shows a greyscale image and Figure 5.11(b) shows the binary image. The selection of threshold T, is a critical issue which determines the content of the image to be classified as a foreground or background information.

Consequently, the foreground information is normally useful for further analysis or processing. As such, unsuitable threshold values will produce inaccurate results.

5.4 Edge Detection: Analysis and Results

Using the edge detection and thresholding module discussed, various types of images with different resolution are experimented. This section will discuss on the results of edge detection technique employed on different scenes and different image resolutions.

The proposed architecture discussed is simulated and implemented in Xilinx FPGA. The architecture is tested with different scenes of different image resolutions. Specifically, the low resolution QVGA (320 x 240) and the high resolution SXGA (1280 x 1024) are experimented.

An initial study is to experiment with different image scenes. After which, a comparison of edge detection with different image resolution is shown. Finally, the system resource utilisation to process a QVGA and a SXGA is also presented.

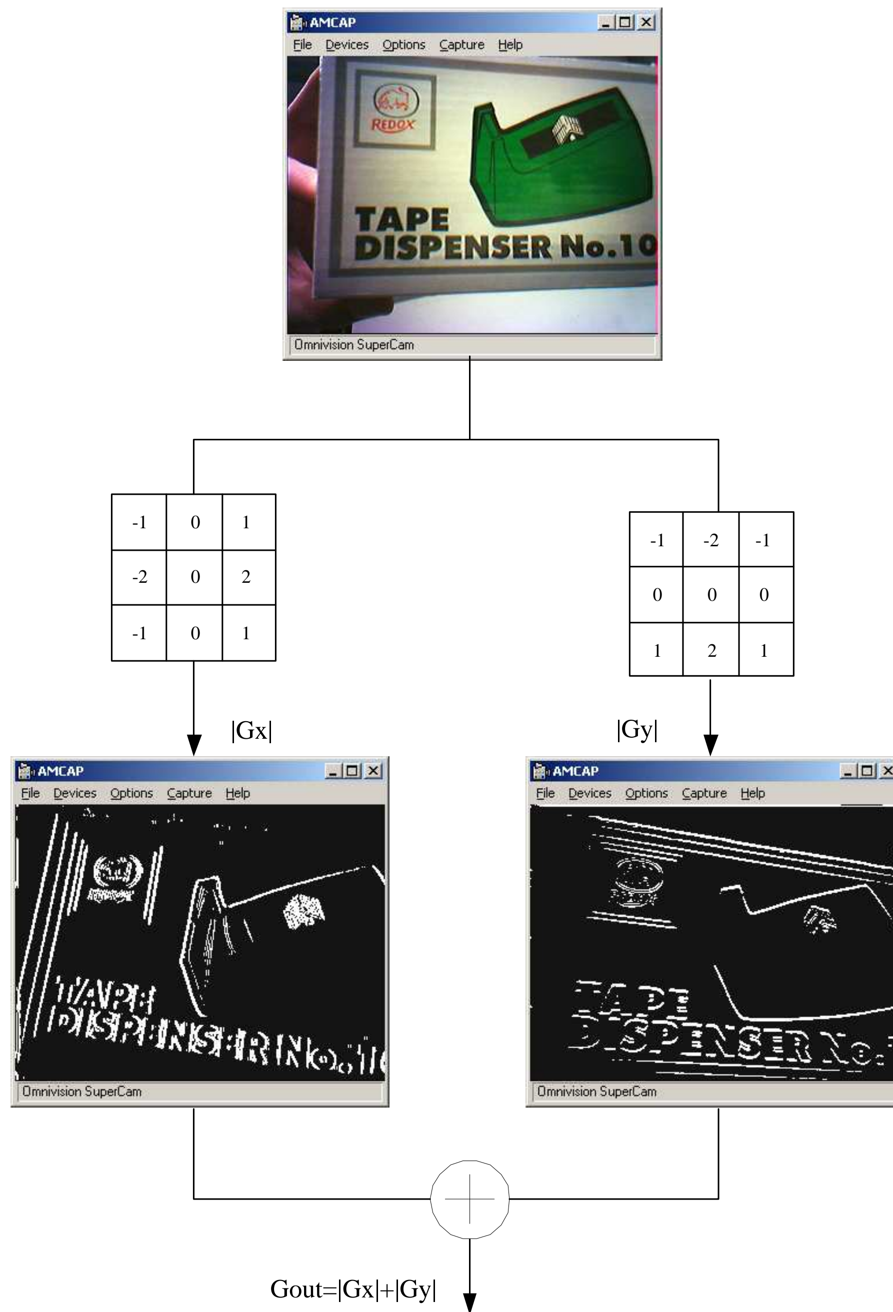
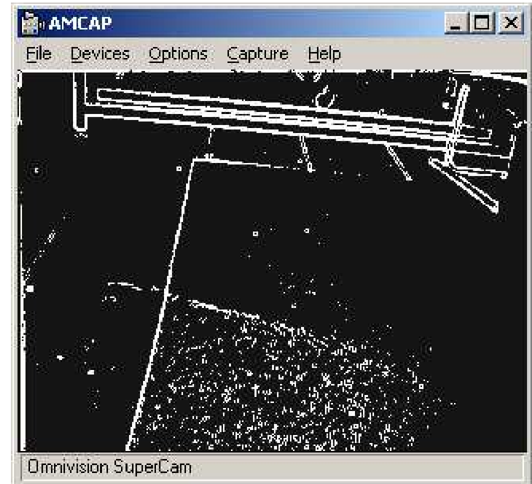
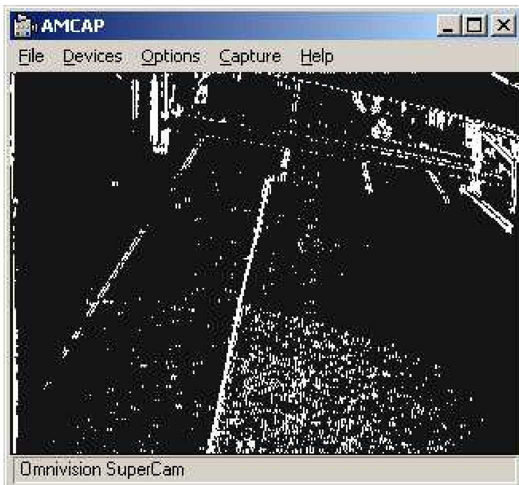


Figure 5.12: Sum of $|Gx|$ and $|Gy|$ component

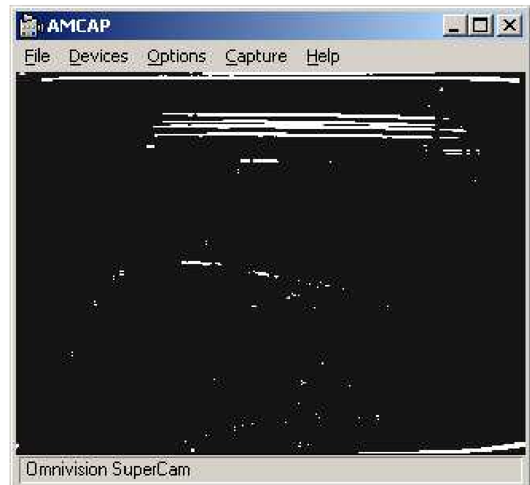
(a) Original image



(b) Edge pixels

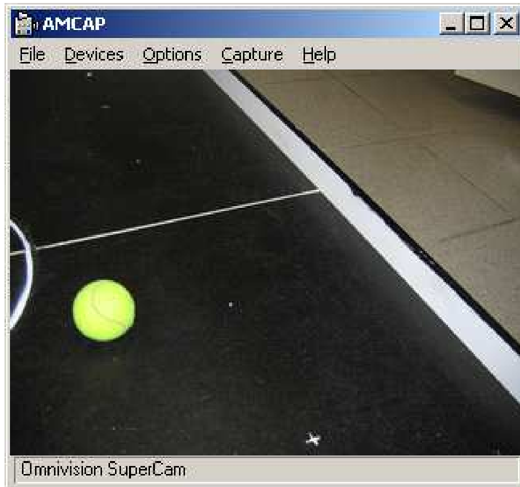


(c) Output of Gx component

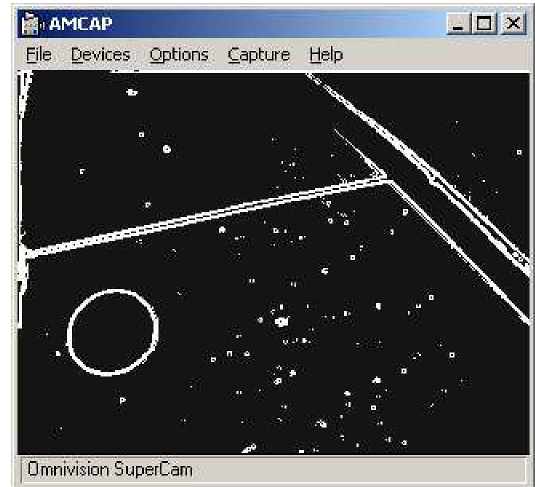


(d) Output of Gy component

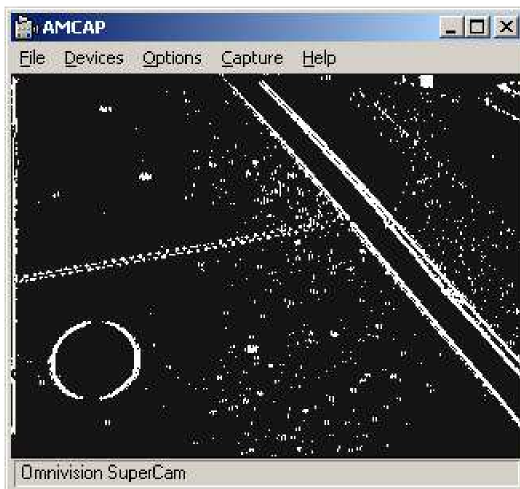
Figure 5.13: Detecting edges of the green carpet



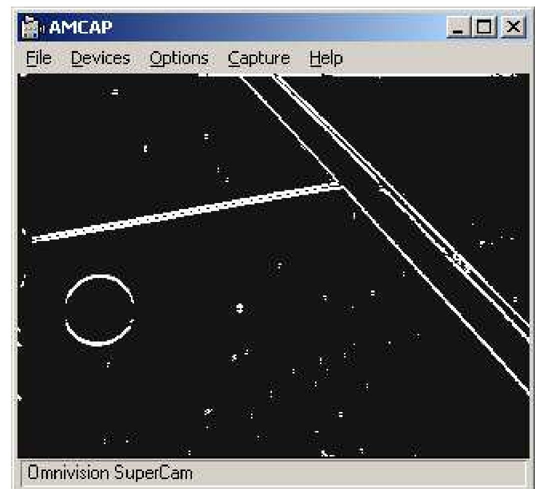
(a) Original image



(b) Edge pixels



(c) Output of Gx component



(d) Output of Gy component

Figure 5.14: Detecting edges of a tennis ball and the boundary lines

5.4.1 Experiment of edge detection with different scenes

The Sobel edge detection module is performed by approximating the magnitude of the two vectors Gx and Gy to be $|G_y(i,j)| + |G_x(i,j)|$. As such, the results of two separate convolution process are added together. The addition of the horizontal and vertical convolution results are shown in Figure 5.12. It can be seen that the $|Gx|$ image responded strongly to vertical lines and $|Gy|$ responded strongly to horizontal lines.

The real-time images are captured at 30 fps using AMCAP program. Figure 5.13 shows that the edges are detected along the green carpet. Figure 5.14 shows a clear outline of a tennis ball in a robot soccer field.

5.4.2 Images with resolution 320 x 240

Figures 5.15(a) and (b) show the original image and the processed image using edge detection operation. It is observed that the edges are well separated from the background with a careful selection of threshold value. An optimal threshold value of 78.42% produces an output image as shown in Figure 5.15. However, it is also observed that some non-edge pixels are also classified as edge pixels. These noise pixels are due to the noise or lighting reflection originates from the image captured. A false edge detected may affect the reliability of image recognition.

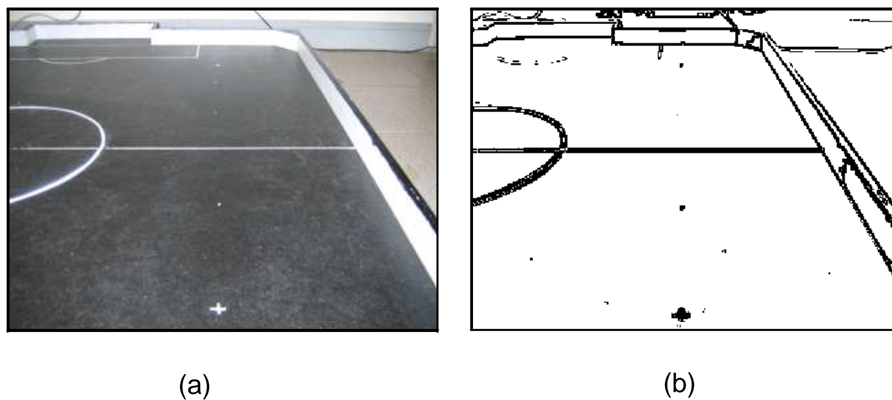


Figure 5.15: Edge detection with image resolution of 320 x 240

Figure 5.16 shows a magnified image of Figure 5.15, with the emphasis on a horizontal white line. It is observed that the changes in intensity level are often seen as a slow change in grey value between connected pixels.

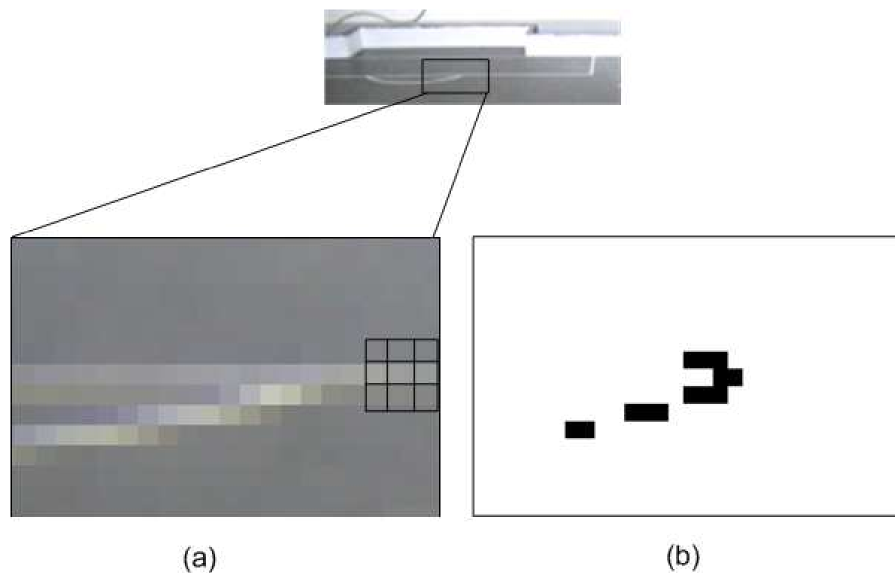


Figure 5.16: Magnified image of Figure13

The output image in Figure 5.16(b) shows that the horizontal line in the original image (Figure 5.16(a)) is not detected as an edge pixel. The difference between grey value of the line and the background is not significant. Hence, it is not detected as an edge. It is noted that the grey level of a particular pixel is related to the image resolution as well as its neighbourhood pixels. As a result the differential value falls below the predefined threshold value. From this experiment, it is concluded that pixels have a gradual change in intensity with respect to its neighbourhood pixels.

5.4.3 Image with resolution of 1280 x 1024

In order to solve the difficulty in detecting the fine line, a high pass convolution filter can be applied to enhance certain features in the image. However, by applying a high pass filter, the noise pixels are amplified as well. This may result in producing many undesired edge pixels.

Another experiment is conducted using an image with a resolution of 1280 x 1024. From Figure 5.17(b) and 5.18(b), it can be seen that the output image produces a sharper edge compared to the image with 320 x 240 resolution.

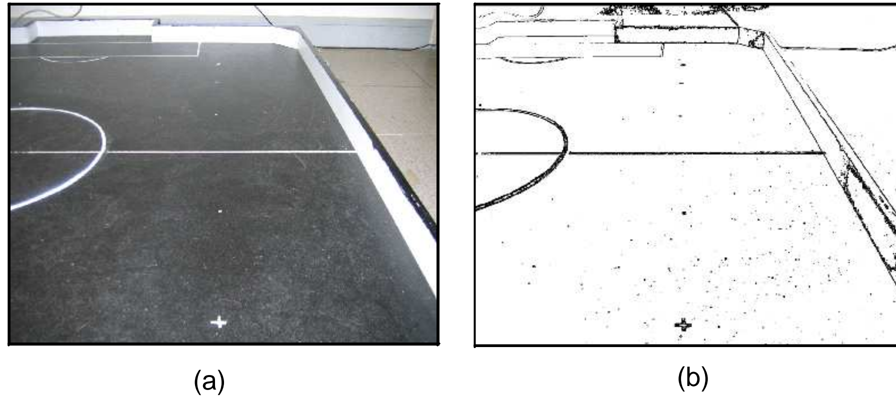


Figure 5.17: (a) Original image of 1280 x 1024 produces (b) fine edge pixels

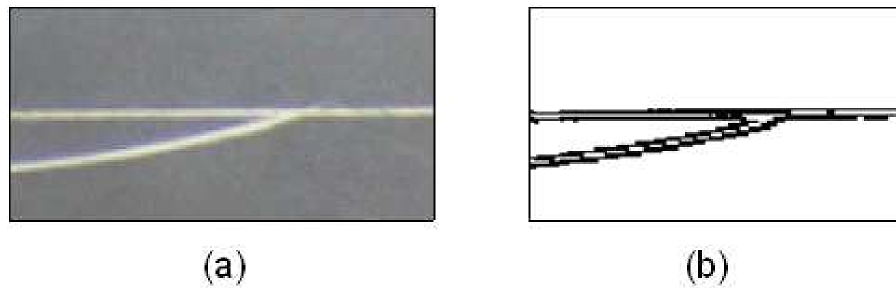


Figure 5.18: (a) Magnified image of Figure 15 and (b) Edge detection of fine lines

Figure 5.18(a) and Figure 5.18(b) are the magnified images of Figure 5.17(a) and Figure 5.17(b) respectively. With the same threshold of 78.4%, a fine resolution of edge pixels is obtained.

5.5 Proposal Parallel Architecture for Low Pass Filter

5.5.1 Noise pixels in high resolution image

From Section 5.4.3, it can be seen that a high resolution image produces an output image with fine edges and is able to detect the fine horizontal line. This section looks into some of the problems encountered when a high image resolution is used.

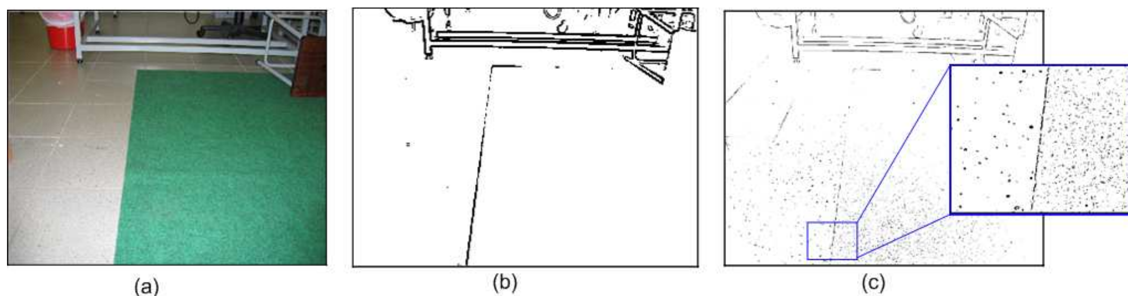


Figure 5.19: Edge detection with different image resolution

An edge detection operation followed by thresholding is applied to two images of different resolutions. A threshold value of 78.43% is applied to both experiments. Figure 5.19(a) shows the original image. Figure 5.19(b) and 5.19(c) show the output images with resolutions 320 x 240 and 1280 x 1024 respectively.

By comparing both output images, it is interesting to note that the low resolution image produces better results. It is observed that the edge is clearly outlined with unwanted background features suppressed.

On the other hand, an image with a higher resolution produces an undesirable result. With a higher resolution, the changes of grey value on the green carpet become significant. With the same convolution operator and a predefined threshold value, the desired edge pattern does not distinguish from the background (Figure 5.19(c)).

5.5.2 Low Pass Filter

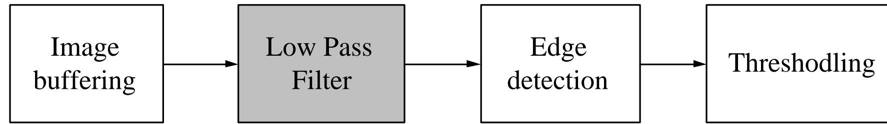


Figure 5.20: Insertion of Low pass filter before edge detection

Image noise usually is seen as random fluctuations in grey-level values superimposed with the ideal grey value. The characteristic of such image usually has a high spatial frequency. In order to remove unwanted noise from an image, while preserving all of the essential edges, a low pass filter is applied to the input image.

Neighborhood averaging is one of the commonly used techniques of applying a low pass filter to smoothen an image. It seeks to remove as much noise as possible while preserving the essential edge information.

Figure 5.20 shows that a low pass filter is applied before the edge detection process. The simplest Low Pass filter arrangement is implemented using a convolution mask in which all coefficients have a value of 1. However in practice, the sum of nine pixels would result in a large value that cannot be represented within the number of grey levels. Hence, the sum is often divided by 9 as shown in Figure 5.21. The mathematical representation of the low pass convolution filter is shown in(5.8).

$$h(i,j) = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Figure 5.21: Convolution coefficients of Low Pass Filter

$$r(i,j) \quad = \quad \frac{1}{9}[f(i,j) \otimes h'(i,j) \tag{5.8}$$

The low pass filter (Figure 5.22) is implemented using a similar architecture discussed in Section 5.2. Instead of using a multiplier, a bit-wise shift register is

used to reduce gate counts. As a result, the division by 9 is replaced by a division of 8, which is simply a shift of 3 bits to the right. Again, such technique achieves a higher speed and smaller circuit compared to a 9 bit multiplier.

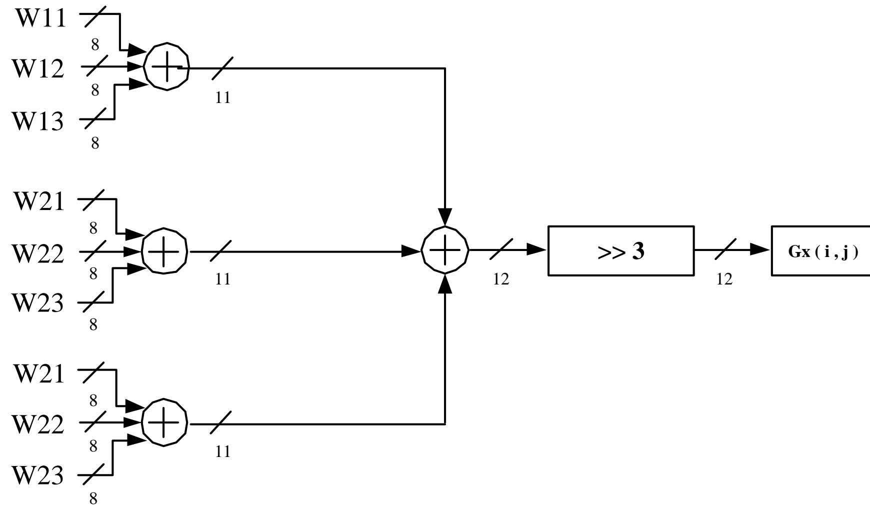


Figure 5.22: Architecture of Low Pass Filter

Figure 5.23 shows the original image and a processed image without low pass filter. Figure 5.24 illustrates the effect of applying a low pass filter to the input image. A comparsion of Edge detection with and without Low Pass filter is shown in Figure 5.25. The unwanted noisy pixels are suppressed after a low pass filter followed by a Sobel edge enhancement operation. Subsequently, all of the three processes are combined, with a low pass filter followed by an edge detection and thresholding. Finally, the image is threshold and the effects are demonstrated in Figure 5.26

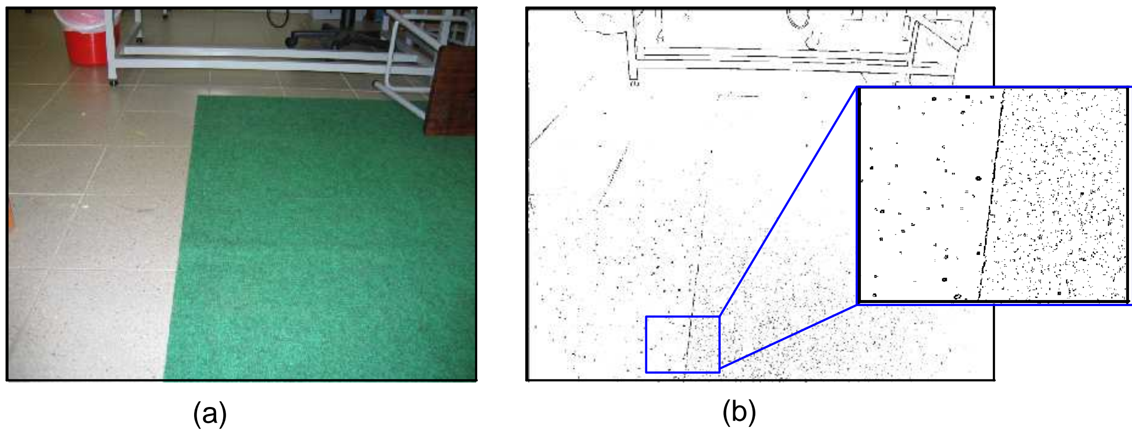


Figure 5.23: (a) Orignal image (b) Edge detection without Low Pass filter

Figure 5.24: (a) Original 1280 x 1024 image (b) Resultant Image applied with Low pass filter

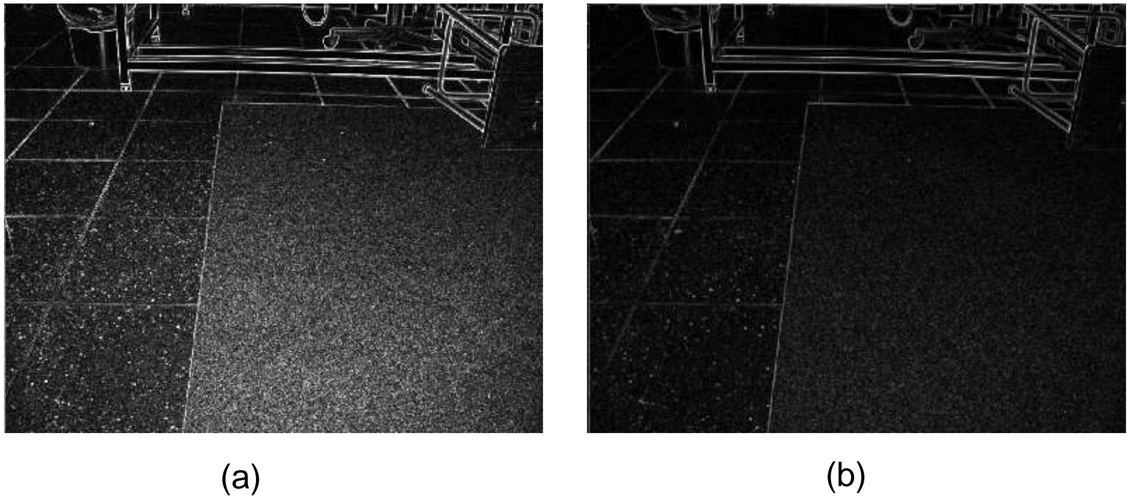


Figure 5.25: (a) Edge detection without Low Pass filter (b) Edge detection with Low Pass filter

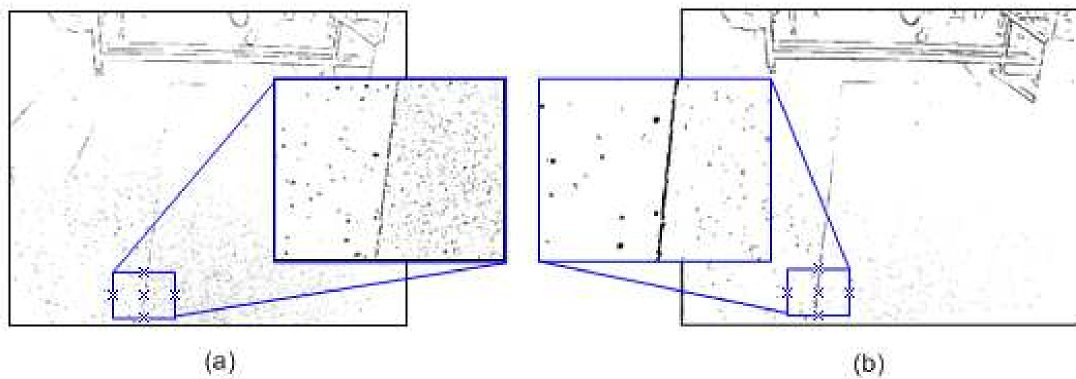


Figure 5.26: (a) Without Low pass filtering (b) With Low pass filtering

5.6 System Resource Utilization

5.6.1 On-Chip memory size requirements

As mentioned in Chapter 4, the memory size of a logical FIFO is given as (w - 3) x 2 Bytes, where w is the image width. The FIFO is generated using the core generator from Xilinx. Each FIFO is generated from the embedded Block RAM in the SpartanIIE chip. The SpartanIIE300 consists a total of 16 blocks RAM and a single block RAM is given as 512 bytes. A total of 8192 bytes of block RAM is available on chip.

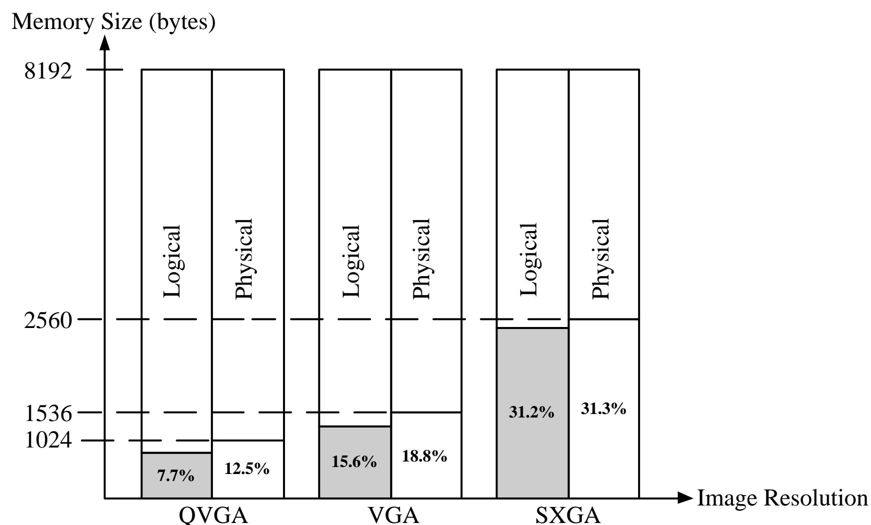
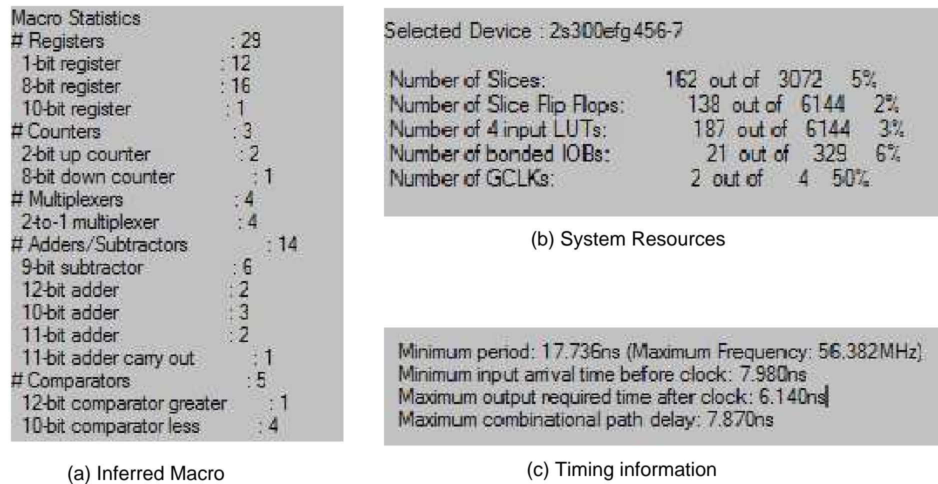


Figure 5.27: Comparison of image buffer size required for different resolution

The logical FIFO memory is defined as the memory size required for buffering while the physical FIFO memory is defined as the actual memory utilised. This is due to the fact that, a single FIFO memory must occupy at least one physical block RAM.

Although increasing the image resolution requires more memory space, the on-chip block RAM is still capable of buffering the SXGA resolution. Figure 5.27 shows the comparison of logical and physical FIFO memory requirements for QVGA, VGA and SXGA resolution.

5.6.2 Logic resources



(a) Inferred Macro

(b) System Resources

(c) Timing information

Figure 5.28: Synthesis report from Xilinx synthesis tool

The Sobel edge detection architecture is designed with the considerations of the gate delay and logic optimization. The entire design is synthesized and all the macros inferred from the verilog codes are shown in Figure 5.28(a). The adders and subtractors are the main functions of the Sobel operations. The registers are used in register transfer level design and the comparators are used for control logic and thresholding purposes.

The inferred hardware logics gates are mapped to a Xilinx Spartan2s300efg456-7 device. From the synthesis report (Figure 5.28(b)), the entire system architecture only occupies 5% of the total slices on chip. The minimum period is given as 17.736 ns.

5.6.3 System performance

As mentioned earlier, the pixels are processed in a parallel approach. In additional, the system computes two sets of 2D convolution within a single clock cycle. The lag time for such computation is 7.870 ns. Since all computations are performed within a single clock cycle, all the processes must be synchronised. As a result, the image

acquisition, image buffering and edge detection process, use a common clock. A clock frequency of 27 Mhz is used in the design. Figure 5.29 shows the computation time required to process images with different frame rates and resolutions.

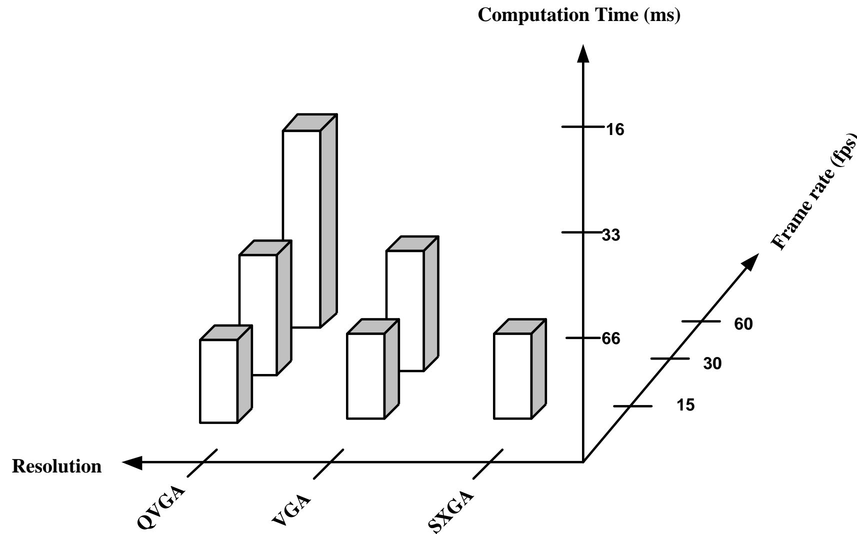


Figure 5.29: Computation time with different resolution

The maximum frame rate that the system can achieve is calculated from the synthesis report. The total gate delay is 17.74 ns. Thus, the system is able to operate at a maximum frequency of 56.38 Mhz. With this information, and assuming that the image sensor transmit a pixel for every clock, the time required to process one frame is (*image_width* * *image_height* * *pixel_clk*). Hence, for a 320 x 240 image, the system can achieve a computational performance of 734 fps. This shows that the system has a great potential of processing real-time video at a very high frame rate.

5.7 Summary of Results

This chapter has presented the design and implementation of the Low Pass filter, the Sobel edge detector and the thresholding operations. The entire system architecture is decomposed into independent modules to ease modular testing. With

each independent module, the algorithms are designed with the consideration to achieve high computational speed with minimum hardware resource required.

After the review of Sobel operator, a parallel architecture is proposed to perform the two 2D convolution operations. The target of completing all operations within a single clock cycle is set. To achieve such demanding performance, it is necessary to exploit the computation parallelism of the Sobel algorithm.

A study is also conducted to evaluate the effects of adding a low pass filter to the design. After which, a threshold operation is performed to extract the desired edge features of an image. In summary, to achieve minimal hardware resources, redundant logics and computations are removed.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This chapter reviews the outcome of this work and, evaluate the results and discussions with respect to the initial objectives. The overall aim is to investigate the methods of achieving the desired performance with the considerations of various constraints stated.

The four major constraints mentioned in Chapter 1 which comprises the demand of computation speed, limitations to memory space, size constraints and lastly energy consumption issues.

With these aims, a review of research work and study of hardware system components is conducted. A few of the existing embedded image processor are compared in Section 1.1. A good portion of work is spent to identify the suitable hardware components used in the experiment. In Section 2.4, the type of FPGA and image sensor are chosen with detailed considerations of resources and system interface. Efforts to implement the entire design on a single chip is realised. This will help to reduce cost and the overall size of the system.

A substantiate amount of time is spent to study various image processing algorithms, the simulation and development tools and the design flow of FPGA

implementation.

Along with the practical considerations, an analytic model is presented in Chapter 4 to estimate the various performance parameters associated with embedded machine vision.

In Chapter 5, the parallel architecture is designed to accomplish high performance image processing task. Methods and techniques are investigated to implement the design with the minimal resources needed. Practical methods such as eliminating the computation of redundance data and replacing multipliers with shift registers are discovered in the course of implementation. These techniques are employed in both Low Pass filtering and edge detection process.

The methods obtained from this work are very encouraging. The custom parallel architecture is able to perform a series of image processing at a very high speed. Real-time image processing at 30 fps with image resolution of 320 x 240 and 640 x 480 is tested in the hardware. In fact, the system shows great potential of processing images even at a higher frame rate. Finally from the evaluation of the work done, the next section raises some directions and considerations for future work.

6.2 Future Work

The future work can be carried at along the following directions. At the moment, the analytical mathematical model helps to estimate the memory buffer required and the processing clock speed for certain image processing algorithm.

As for the parallel architecture, the same concepts and techniques can be applied to other image processing algorithm. However, it is recommended to work on algorithms that are repetitive and characterized by large amount of data to be processed. The pyramid architecture for data processing should be used as a reference. Image algorithm such as image erosion, dilation, opening and closing are

suitable to exploit computational parallelism.

Lastly, it will be an interesting area to study the soft-core architecture of a Microprocessor. With the reconfigurable architecture of FPGA, the soft-core Microprocessor can include custom instruction set for handling demanding operations.

Bibliography

[1] Thomas Braunl, “Improv amd EyeBot Real-time Vision on-board Mobile Robots”, *IEEE, Mechatronics and Machine Vision in Practice*, vol.4, pp. 131-135, 1997.

[2] A. Rowe, C. Rosenberg and I. Nourbakhsh, “A Low Cost Embedded Color Vision System”, *IROS 2002 conference*, 2002.

[3] R. T. Chin and C. R. Dyer, “Model-Based Recognition in Robot Vision”, *ACM COMPUTING SURVEYS*, vol.18, pp.67-108, 1986.

[4] M.J. Flynn, “Very high speed computing systems”, *Proc. IEEE*, vol.54, no.12, 1966.

[5] A. Aliphas and J.D. Feldman, “The versatility of digital signal processing chips”, *IEEE Spectrum*, vol.24, no.6, pp.40-45, 1987.

[6] S. Hauck, “The Roles of FPGA’s in reprogrammable system”, *proceedings of the IEEE*, vol.86, no.4, pp.615-638, 1988.

[7] P. L. Athanas and A. L. Abbott, “Real-time Image Processing on a Custom Computing Platform”, *IEEE Computer*, vol.28, no.2, pp.16-25, 1995.

[8] D. Crookes and K. Benkrid, “An FPGA implementation for image component labeling”, *Reconfigurable Technology: FPGAs for Computing and Applications, Proc. SPIE 3844*, pp.16-23, 1999.

[9] D. Crookes, “Architectures for high performance image processing: The future”, *Journal of Systems Architecture*, vol.45, no.10, pp.739-748, 1999.

[10] R. Woods, D. Trainor and J. P. Heron, "Applying an XC6200 to real-time image processing", *IEEE Design and Test of Computers*, vol.15, pp.30-38, 1998.

[11] D. Bhatia, "Field programmable gate arrays. A cheaper way of customizing product prototypes", *Proc. IEEE*, vol.13, no.1 pp.16-19, 1994.

[12] The Vision and Autonomous Systems Center (http://*vasc.ri.cmu.edu/*), 2005.

[13] The K-team, Khepera vision turret K6300 (http://*www.k-team.com/robots/khepera/k6300.html/*), 2004.

[14] The RoboCup Federation (http://*www.robocup.org*), 2003.

[15] Kitano Symbiotic Systems Project- Open PINO Platform (http://*www.symbio.jst.go.jp/PINO/index.html*), 2003.

[16] ActivRobots (http://*www.activrobots.com/*), 2004.

[17] RC1000PP Product Information Sheet (http://*www.te.rl.ac.uk/europractice/vendors/rc1000.pdf*), 2004.

[18] The CMUcam Vision Sensor (http://*www.cs.cmu.edu/ cmucam/*), 2004.

[19] Viorela Ila, *Reconfigurable Devices Architecture for Robotics Applications*, PhD thesis, University of Girona, 2005.

[20] D. Sima, T. Fountain and P. Kacsuk, *Advanced Computer Architectures: A design Space Approach*, Pearson Education Limited, England, 1997.

[21] A.N. Choudray and J.H. Patel, *Parallel architectures and parallel algorithms for integrated vision systems*, Kluwer Academic Publishers, Dordrecht, 1990.

[22] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A quantitative approach*, Morgan Kaufmann, Calif, 1990.

[23] Stephen Brown and Jonathan Rose, *Architecture of FPGAs and CPLDs: A Tutorial*, Department of Electrical and Computer Engineering, University of Toronto.

[24] G.J. Awcock and R. Thomas, *Applied Image Processing Book*, McGraw-Hill, USA, 1996.

[25] Keith Jack *Verilog HDL vs. VHDL For the First Time User Bill Fuchs*, OVI, 1995.

[26] Douglas J. Smith *VeriBest Incorporated, VHDL and Verilog Compared and Contrasted Plus Modeled Example Written in VHDL, Verilog and C*, 2003.

[27] Keith Jack *Video Demystified, A handbook for the Digital Engineer*, LLH Technology Publishing, Eagle Rock, 1997.

[28] Michael John Sebastian Smith *Application-Specific Integrated Circuits*, Addison-Wesley Professional, England, 1997.

[29] Stephen Brown and Jonathan Rose *Architecture of FPGAs and CPLDs: A Tutorial*, Department of Electrical and Computer Engineering, University of Toronto.

[30] *Spartan-IIE 1.8V FPGA Family: Complete Data Sheet*, Xilinx, July 2003.

[31] N. K. Ratha and A. K. Jain, " Computer Vision Algorithms on Reconfigurable Logic Arrays ", *IEEE Transactions, Parallel and Distributed Systems* , vol.10, pp. 29-43, 1999.

[32] Kwangho Yoon, Chanki Kim, Bumha Lee, and Doyoung Lee, " Single-Chip CMOS Image Sensor for Mobile Applications", *IEEE Journal of Solid-State Circuits*, vol.37, pp. 1839-1845, 2002.

[33] G.J. Awcock, M.T. Rigby“ Single Integrated Imaging Sensors and Processing ”, *IEE Colloquium on*, vol.5, pp. 2-5, 1994.

[34] S. Shigematsu, H. Morimura Y. Tanabe, T. Adachi, and K. Machida “A Single-Chip Fingerprint Sensor and Identifier”, *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, vol.34, pp. 1852-1859, 1999.

[35] Darrin Cardani “Adventures in HSV Space”, *The Advanced Developers Hands On Conference*, 2001.

[36] *Advanced information on OV7620 Data Sheet*, OmniVision, July 2003.

[37] Difference between CCD and CMOS image sensors in a digital camera. (http://*electronics.howstuffworks.com/question362.htm*), 2005.

[38] YUV Colour Space. (http://*softpixel.com/ cwright/programming/colorspace/yuv/*), 2005.

[39] YUV From Wikipedia, the free encyclopedia. (http://*en.wikipedia.org/wiki/YUV*), 2005.

[40] K.Kant *Introduction to Computer System Performance Evaluation*, Mc Graw-Hill, Singapore, 1992.

[41] David J. Lilja *Measuring Computer Performance: A practitioner's guide*, Cambridge University Press, United Kingdom, 2000.

[42] Hisashi Kobayashi *MModeling and Analysis: An introduction to System Performance Evaluation Methodology*, Addison-Wesley, Philippines, 1978.

[43] Raj Jain *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley- Interscience, New York, 1991.

[44] Queueing Theory for Embedded Systems Designers (http://*www.kalinskyassociates.com/Wpaper5.html*), 2005.

[45] Kitano Symbiotic Systems Project- Open PINO Platform (http://*www.symbio.jst.go.jp/PINO/index.html*), 2003.

[46] P.Heidelberger and S.S.Lavenberg, " Computer Performance Evaluation Methodology ", *IEEE Transactions on Computers*, vol.C-33, no.12, pp. 1195-1220, 1984.

[47] Rafeal C. Gonzalez *Digital Image Processing*, LLH Addison Wesley Publishing, New York, 1992.

[48] Ioannis Pitas *Digital Image Processing Algorithms*, Prentice Hall, UK, 1993.

[49] Keith Jack *Introductory computer vision and image processing*, McGraw Hill Book Company, Singapore, 1991.

[50] K. Wiatr and E. Jamro, " Implementation of image data convolutions operations in FPGA reconfigurable structures for real-time vision systems ", *International IEEE Conference on Information Technology: Coding and Computing* , pp. 152-157, 2000.

[51] K. Wiatr and E. Jamro, " Implementation of convolution operation on general purpose processors", *Proceedings of the Euromicro Conf. on Multimedia and Telecommunication , euromicro*, vol. 00, pp. 0410, 2001.

[52] D. Crookes, "Architectures for high performance image processing: The future", *Journal of Systems Architecture*, vol.45, no.10, pp.739-748, 1999.

[53] Domingo Benitez, " Performance of reconfigurable architectures for image-processing applications", *Journal of Systems Architecture*, vol.49, no.4, pp.193-210, 2003.

[54] Eddy De Greef, " Memory size reduction through storage order optimization for embedded parallel multimedia applications",

[55] F.Catthoor, W.Geurts and H.De Man, " Loop transformation methodology for fixed-rate video image and telecom processing applications", *Proc. Int. Conference on Application Specific Array Processors*, , pp.427-438, 1994.

[56] Terry W.Griffin and Nelson L.Passos, An Experiment with hardware implementation of edge enhancement filters ", *The Journal of computing in small colleges*, vol. 17, pp. 24-31, 2002.

[57] K. Wiatr and E. Jamro, " Implementation of image data convolutions operations in FPGA reconfigurable structures for real-time vision systems ", *International IEEE Conference on Information Technology: Coding and Computing* , pp. 152-157, 2000.

[58] K. Wiatr and E. Jamro, " Implementation of convolution operation on general purpose processors", *Proceedings of the Euromicro Conf. on Multimedia and Telecommunication , euromicro*, vol. 00, pp. 0410, 2001.

[59] D. Crookes, "Architectures for high performance image processing: The future", *Journal of Systems Architecture*, vol.45, no.10, pp.739-748, 1999.

[60] Domingo Benitez, " Performance of reconfigurable architectures for image-processing applications", *Journal of Systems Architecture*, vol.49, no.4, pp.193-210, 2003.

Author's Publications

Journal Publications

Chan, Kit Wai, Prahlad Vadakkepat and Xiao Peng, "Hierarchical robot control structure and Newton's divided difference approach to robot path planning", *Journal of Harbin Institute of Technology*, Vol.8(3) , pages 303-308, 2001.

Conference Publications

Chan Kit Wai and Prahlad Vadakkepat, "Real-Time Debugger for Robot Soccer System", *Proc. of 2002 FIRA Robot World Congress*, pages 639 - 642, 2002.

C.C. Ko, Ben M. Chen, C.D. Cheng, X. Xiang, Chan Kit Wai and Y.P. Khanal, P. Vadakkepat, "Development of a Web-based Mobile Robot Control Experiment", *Proc. of 2002 FIRA Robot World Congress*, pages 488 - 493, 2002.

Tey Ghee Kwan, Prahlad Vadakkepat, Chan Kit Wai, Liu Xin, "Mobile Robot Path Planning using Electrostatic Potential Field", *Proc. Of 2003 FIRA World Congress*, 2003

Yeo Hui Mei, Chan kit wai, Prahlad Vadakkepat, "Evaluation of K-Means clustering and thresholding techniques" , *proc. Of 2004 FIRA World Congress*, 2004.

Chan Kit Wai, Prahlad Vadakkepat and Tan Kok Kiong, "An Analytic Model for Embedded Machine Vision: Architecture and Performance Exploration", *Proc. Of ICARA 2004, Palmerston North, New Zealand*, 2004.