The University of Western Australia
Faculty of Engineering, Computing and Mathematics
School of Electrical, Electronic and Computer Engineering
Centre for Intelligent Information Processing Systems

FPGA Based Embedded Vision Systems

Final Year Project

Lixin Chin (10206267)

Supervisors: A/Prof Thomas Bräunl
A/Prof Anthony Zaknich

Submitted 27th October 2006

Lixin Chin
54 Davis Road
ATTADALE WA 6156

27th October 2006

The Dean
Faculty of Engineering, Computing and Mathematics
The University of Western Australia
35 Stirling Highway
CRAWLEY WA 6009

Dear Sir,

I submit to you this dissertation entitled "FPGA Based Embedded Vision Systems" in partial fulfilment of the requirement of the award of Bachelor of Engineering.

Yours faithfully,

Lixin Chin.

Abstract

Embedded micro-controller systems are becoming increasingly popular in image processing applications. Imaging algorithms can consume large amounts of the processing time on a CPU, which also needs to handle other tasks such as I/O. A significant amount of research has been performed in recent years into the acceleration of image processing algorithms using reconfigurable hardware logic devices such as FPGAs (Field Programmable Gate Arrays). This project combines the two, presenting an embedded controller with an on-board FPGA for real-time image processing.

In addition, this project investigates the implementation of several imaging algorithms in hardware logic. FPGA implementations of algorithms for performing colour space conversion, image thresholding and object location are presented and analysed.

Finally, this project outlines the design and implementation of a new hardware divisor for performing 8-bit division. The error probability function of this division algorithm is fully characterised and contrasted against existing hardware division algorithms.

Acknowledgements

Many thanks to my supervisors, A/Prof Thomas Bräunl and A/Prof Anthony Zaknich for all their help, and for allowing me the opportunity to work on this project.

Thanks also to Ivan Neubronner for his tremendous assistance with the hardware side of this project, especially with the PCB layout.

A special thanks to Dr Farid Boussaid for helping me with my RSI.

Thanks to my family for their support, and for putting up with me this last year.

Finally, thanks to my project partners, Bernard Blackham and David English, and to everyone in the Robotics Labs. Thanks for everything, its been a great year — the burning submarine was especially memorable.

Thank you everyone.

Abbreviations

ADC/DAC	Analog-to-Digital / Digital-to-Analog Converter
ARM	Acorn RISC Machine
ASIC	Application Specific Integrated Circuit
CPLD	Complex Programmable Logic Device
DMA	Direct Memory Access
DSP	Digital Signal Processor / Processing
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input Output
HDL	Hardware Description Language
HSI	Hue, Saturation, Intensity
HSL	Hue, Saturation, Lumience
HSV	Hue, Saturation, Value
JPEG	Joint Photographic Experts Group
LUT	Lookup Table
MMIO	Memory Mapped Input Output
RISC	Reduced Instruction Set Computer
RGB	Red, Green, Blue
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This paper is one of three [1, 2] describing a new embedded controller for performing real-time image processing using a Field Programmable Gate Array (FPGA). Embedded systems are increasingly used for applications demanding computationally expensive image processing. Despite the increasing power of embedded controllers, imaging algorithms can consume a large amount of a CPU's processing time. At the same time, much research has been undertaken into accelerating graphics and imaging using reconfigurable hardware logic devices such as FPGAs [3]. This project combines the two to create an embedded controller with an integrated FPGA for real-time image processing.

The current generation of robots used at the Mobile Robot Lab[1] at the University of Western Australia are controlled using the EyeBot controller EyeCon [4]. The EyeBot platform has been through several revisions, the current being the EyeBot M5. The EyeBot has proven to be both flexible and powerful, driving not only simple wheel and track driven robots, but also omni-directional robots, balancing and walking robots, and autonomous planes. Additional work is currently in progress on an AUV (autonomous underwater vehicle) and a semi-autonomous wheelchair.

Many of these applications rely heavily on processing images from an on-board camera. A disadvantage of the current EyeCon hardware is that it lacks a dedicated

[1]The Mobile Robot Lab in the Centre for Intelligent Information Processing Systems, the School of Electrical, Electronics and Computer Engineering, Faculty of Engineering, Computing and Mathematics.

processor for images, hence all imaging algorithms have to be performed by the CPU. Since the CPU also needs to handle all I/O operations as well as executing user applications, this places a significant burden on the processor. In order to solve these problems, a new hardware platform needed to be built to replace the existing EyeBot M5. This new EyeBot M6 is fully described in this thesis, (Blackham, 2006) [1] and (English, 2006) [2].

1.1 Project Scope

The aims of this project were:

1. To design and build an embedded micro-controller platform capable of performing real-time image processing using an on-board FPGA.

2. To investigate and implement various image processing algorithms for inclusion in the FPGA.

3. To produce a hardware/software platform capable of replacing the current EyeBot M5 controller. The new platform needed to be powerful and extensible enough for users to be able to design their own mobile robotics applications.

1.2 Major Contributions

The major contributions of this project were:

1. The design and implementation of VHDL modules for performing colour object location on the University of Western Australia's newly designed EyeBot M6 platform. The implemented modules include modules for performing colour space conversion, and subsequent processing modules for image thresholding and object location.

2. In order to perform the colour space conversion in hardware, this project also included the design and implementation of a hardware division unit optimised for fast, highly accurate, 8-bit division.

3. A full analysis of the new hardware divisor architecture, as well as an in-depth comparison against existing hardware divisors.

4. Contributions towards the design of the EyeBot M6 hardware, as well as software testing on the platform.

1.3 Thesis Outline

This thesis is divided into the following chapters

Embedded Platforms: A background description of embedded and programmable logic systems. This chapter presents an overview of current micro-controller platforms, and a review of FPGA accelerated image processing systems.

Hardware Design: An overview of the hardware of the newly developed EyeBot M6 platform.

Computer Vision: An overview of the theory behind the location of coloured objects from images as well as the methods of representing coloured images, and the functions for converting between one representation and another.

Fixed Point Arithmetic: This chapter presents a means of representing and operating on fractional numbers using only integer storage and integer operators.

Division Algorithms: Presents an architecture for performing 8-bit division, along with a full analysis and comparisons to existing division algorithms.

FPGA Implementations and System Performance: An overview of the FPGA implementations of the object location and colour space conversion algorithms. This chapter also presents an evaluation of the performance of the implemented modules, along with the FPGA resources consumed.

Conclusion: A summary of the work accomplished during this project, as well as suggestions for future work.

Chapter 2

Embedded Platforms

The steady progression of Moore's Law, combined with the improvements in manufacturing techniques have enabled increasing sophistication in embedded systems controllers. The increase in processing power has reached the point where current embedded CPUs are comparable in performance to the desktop CPUs of a decade ago. With better manufacturing techniques, these new embedded CPUs are also cheaper and more power efficient than the previous generation of processors. This increased capability has enabled the application of embedded systems to tasks which in the past would have been performed by much larger systems. In particular, there is increasing interest in the use of small, portable, imaging devices. In the automotive industry for example, there is interest in using embedded image processing devices to analyse driving conditions and detect objects on the road [5]. Despite the power of embedded processors, the volume of information contained in image data can swamp a CPU executing complicated algorithms.

In recent years there has been a significant amount of research into the use of FPGAs to accelerate computing tasks. FPGAs (Field Programmable Gate Arrays) are semiconductor devices containing programmable logic and programmable interconnects. A FPGA is essentially a hardware processing unit that can be reconfigured at runtime [6]. FPGAs evolved out of the older CPLD (Complex Programmable Logic Device) chips. Compared to CPLDs, FPGAs typically contain a much higher number of logic cells. Additionally the architecture of FPGAs includes several higher

level embedded function blocks, such as multipliers and block RAMs. This allows FPGAs to implement much more complicated functions than the older CPLDs.

The speed of a FPGA is generally slower than that of an equivalent ASIC (Application Specific Integrated Circuit) chip, however an ASIC's functionality and architecture are fixed on manufacture, whereas a FPGA can be reconfigured as necessary. This leads to substantially lower development and manufacturing costs, and also allows the final system a greater degree of flexibility.

On a FPGA, algorithms are constructed from blocks of hardware logic, instead of instructions interpreted and executed by a processor. In addition, the architecture of FPGAs allows for the simultaneous, parallel execution of multiple tasks. All these factors mean that certain algorithms can be executed much, much faster on a FPGA than they could on a CPU [3, 7].

2.1 Hardware Description Languages

To configure a FPGA, users first provide a description of the desired functional modules in the form of either a schematic, or hardware description language (HDL). This description is then synthesised to produce a binary file (usually using software provided by the FPGA manufacturer) used to configure the FPGA device.

The advantage of using a hardware description language is that it allows the user to both describe and verify the functioning of a system before it is implemented on hardware. HDLs also allow for the succinct description of concurrent systems, with multiple subcomponents all operating at the same time. This is in contrast to standard programming languages, which are designed to be executed sequentially by a CPU. Using a HDL also allows for a more flexible and powerful expression of system behaviour than simply connecting components together using a schematic.

Common HDLs used in FPGA design are VHDL [8] (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language) and Verilog [9]. VHDL developed from the Ada programming language, and has a relatively verbose syntax. In

addition, VHDL is both strongly typed and case insensitive [8]. By contrast, Verilog evolved out of the C programming language, and as such is a much more terse language than VHDL. Verilog is also more weakly typed than VHDL, and is case sensitive [9]. The two languages are highly similar in functionality, and both are widely supported by software synthesis tools [10].

This project has chosen to use VHDL for describing and synthesising the FPGA modules. The stronger typing in VHDL means certain errors will be caught during synthesis which might otherwise be missed in Verilog.

2.2 Micro-controller Platforms

In recent years, the availability of powerful, low cost, micro-controllers and cameras has led to the development of several micro-controller platforms for mobile vision applications. This project investigated several of these platforms to determine their suitability as a base on which to build the next generation EyeBot controller.

The MDP Balloon Board is one such platform recently developed by the Cambridge-MIT Institute as part of the Multidisciplinary Design Project [11]. The current Balloon Board (version 2.05 at the time of writing) is based around a 206Mhz StrongARM CPU. Version 3 of the MDP board has been in development since 2004, and is currently nearing production. The MDP board v3 is based around the Intel XScale CPU and comes in two versions, one with a FPGA, and the other with a CPLD. The Balloon Board v3 is technically impressive, combining a fast processor (a 630MHz Intel PXA270, ARM9 architecture CPU), large amounts of RAM (128MB) and a 400K gate FPGA. The board supports a number of peripherals, including Bluetooth, serial, USB host and slave, and several GPIOs. It is also very small and light, slightly larger than a credit-card and about 30g in weight [11]. Despite the MDP board's notable specifications, its lack of availability made it unsuitable for use in this project.

Another similar platform is the Qwerk Robot Controller, developed by the Carnegie Mellon Robotics Institute [12]. Unfortunately the available documentation is much

less comprehensive than that available for the Balloon Board. The Qwerk is known to contain a 200Mhz ARM9 processor with a hardware floating point unit, a Xilinx Spartan-3E FPGA, 32MB RAM and 8MB Flash memory. Peripheral support includes Ethernet, USB Host and Wireless LAN, WebCam video input support, motor and servo controllers, and GPIOs. The Qwerk board is notable for two reasons. Firstly it is one of the very few micro-controller platforms available with a hardware floating point unit. Secondly is its advertised support for "sensorless feedback," measuring the back-emf (voltage) of a DC motor to estimate the current speed of the motor — the Qwerk seems to be the only platform with this feature [12]. Much like the Balloon board, the Qwerk has only been in production since August 2006, and thus was unavailable for consideration at the commencement of this project.

The CMUCam Vision System is another micro-controller system also developed by Carnegie Mellon [13, 14]. It is a very small, focused, device, consisting only of a camera, micro-processor, and serial interface. The second revision of the system added a frame-buffer chip, allowing the device to store an entire camera frame. The processor is an 8-bit Ubicom running at 75MHz, and the camera is an OmniVision OV6620. The entire device is very small, 45mm $\times$ 57mm $\times$ 50mm in size [13]. The CMUCam is a very specialised system, more a "smart camera" than a general purpose micro-controller platform. The system is notable for the amount of functionality that has been packed into such a small device, but its lack of peripheral support and processing power also makes it unsuitable for this project's needs.

In the end the lack of a suitable, available, micro-controller platform at the commencement of this project meant that a new platform had to be developed. This platform, the EyeBot M6 is detailed in Chapter 3.

2.3 Image Processing Systems

FPGAs hold several advantages over CPUs when it comes to image processing. While they often run at much lower clock speeds, the parallel nature of hardware logic allows FPGAs to execute certain algorithms much faster than a regular CPU.

Several researchers have reported speedup factors from 20 to as much as 100 from FPGAs as compared to standard processors [3, 15].

Zemcik [16] outlines the basic hardware architecture often used by researchers — a simple hardware board incorporating a FPGA, processor and RAM all linked by a central bus. The FPGA performs time critical computation tasks, while the processor performs non-critical but algorithmically complex tasks. Zemcik uses a DSP (Digital Signal Processor) instead of a CPU, but the basic architecture is the same as those used in the micro-controller platforms in Section 2.2. The paper also outlines FPGA architectures for performing volume rendering and raytracing. Zemcik demonstrates the use of FPGAs for performing output processing (graphics rendering), but the same hardware could easily be used to perform input processing (computer vision) instead. This illustrates the flexibility available from FPGA based systems.

Borgatti [17] proposes a similar architecture for using a FPGA as a co-processor to accelerate DSP applications. Unlike Zemcik, Borgatti proposed an integrated device incoporating both FPGA and processor on a single chip. This is the same concept shown in the Xilinx Virtex series FPGAs which combine embedded processor cores and FPGA logic blocks into a single chip [18]. This demonstrates the increasingly prevalent desire to integrate the capabilities of FPGAs with general purpose computation units.

Other researchers have focused their attention on the different algorithms which may be accelerated by FPGAs. Krips [19] outlines an FPGA implementation of a neural network based system for real-time hand tracking. A neural network consists of multiple neurons connected together, where the output of each neuron is the sum of the inputs to the neuron multiplied by an associated weighting. The neuron's output is then processed by a (in general) non-linear "activation function" before becoming the input to the next layer of neurons. The function and performance of a neural network can be adjusted by tuning the input weightings of each neuron through some sort of training process. A neural network is an example of a spatial computing alogrithm — many calculations needing to be performed in parallel. This

is contrasted with time sequential algorithms where tasks are executed in series. The parallelism of hardware logic means FPGAs are well suited for spatial computing, whereas CPUs are oriented towards time sequential computing.

Torres-Huitzil [20] outlines a related system, an image processing architecture based on a neurophysiological model for motion perception. Biologically inspired vision models provide good accuracy, but perform poorly on regular CPUs since they are oriented towards spatial computing instead of time sequential computing. The FPGA implementation was found to be approximately 100 times faster than a Pentium IV desktop CPU performing the same function, however it still was not fast enough for real-time applications. The researchers suspect that this could be rectified by using a faster or larger FPGA [20]. Unfortunately the algorithm used by the researchers is unsuitable for implementation in this project, since it consumed the majority of the resources available on a FPGA much larger than the one used in this project.

A much simpler vision system was proposed by García-Campos [21]. The paper outlines a FPGA based system for colour image tracking. The system first converted the input RGB image data into HSI colour space. The converted image was then thresholded to produce a bitmap which was fed into a row/column accumulation module to locate the coloured object in the source image. Notably, the system did not perform the colour conversion directly. Instead the 8-bits per channel RGB data was sampled by the cameras, then decimated down to 5-bits per channel. The resulting 15-bit combined channel data was then used as an index into a precomputed lookup table to perform the HSI colour space conversion. Given that the output HSI data contained 8-bits per channel, the lookup table would have required $3 \times 2^{15} = 3 \times 32$ kB of space. This approach was probably chosen due to the difficulty of performing the RGB to HSI conversion directly, as it requires several division operations. Chapter 4 examines this problem in more detail. The system proposed by García-Campos was conceptually very simple, but nevertheless produced quite good results [21]. A modified version of this system has been chosen for implementation in this project.

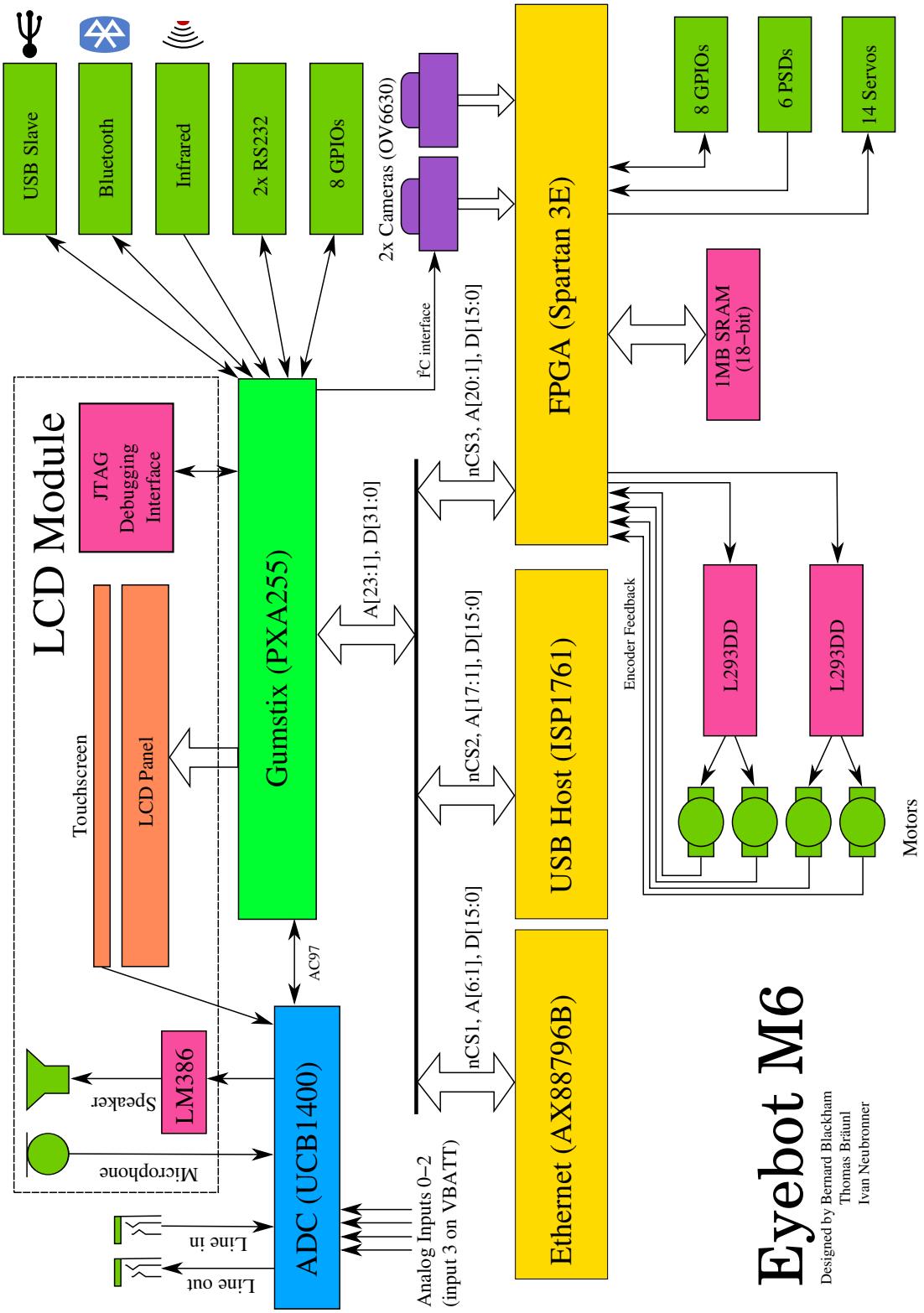


Figure 3.1: Hardware block diagram of the new EyeBot M6 platform

Chapter 3

Hardware Design

The hardware side of the final project design is fully described in (Blackham, 2006) [1], but is given a brief overview here. The block diagram of the new hardware platform is shown in Figure 3.1. This architecture is similar to that implemented by other researchers [16], with the FPGA acting as a co-processor to the CPU. Both have their own RAM, allowing them to perform calculations independent of each other.

The Gumstix Connex 400m-bt [22] platform, shown in Figure 3.2, was chosen to form the core of the new EyeBot M6 platform. The Gumstix features a 400MHz Intel XScale PXA255 (ARM9 architecture) CPU, bluetooth, USB slave and an LCD controller. It also features 64MB of RAM and 16MB of Flash memory storage. The platform is extensible via expansion boards plugged into the two Hirose connectors on the Gumstix. This platform was chosen because it provides the desired features of speed, low power consumption and I/O support, at a reasonable price. The Gumstix also comes with an embedded Linux operating system, which is advantageous, since it means that the system can be programmed using freely available

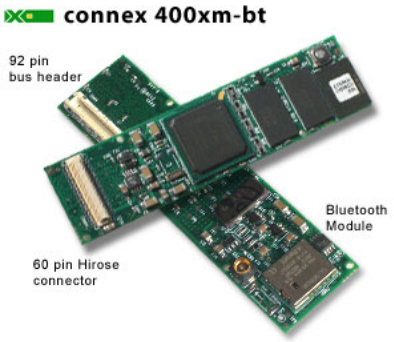


Figure 3.2: Image of the Gumstix embedded platform [22]

development tools.

Using this platform as a starting point, this project then designed and built an expansion board attached to the Hirose connectors. This board contains connectors for all of the robot's I/O devices — servo and motor controllers, encoders, position sensing devices (PSDs), and general purpose I/O pins (GPIO). It also contains two camera interfaces, USB host, ethernet, an ADC/DAC (the AC'97 shown in Figure 3.1), and a FPGA. The main board also contains an additional connector for an expansion board. This second expansion board mounts the LCD, touchscreen and speaker. Having a second board in this fashion means it is possible to replace the LCD or speaker without disturbing the other components on the main board.

The FPGA chosen for this project was the Xilinx Spartan3-500E [18]. This is the largest FPGA available in a non-ball-grid-array (BGA) configuration [1]. The choice was made to avoid BGA components due to the cost and complexity of manufacturing and soldering PCBs with BGA chips. Additionally, the Spartan3-500E is readily available at a low price, and the Xilinx FPGA development tools are freely available at their website. This is important, since it means that programming for the EyeBot M6 platform does not require expensive development environments.

Software optimisations have reduced the FPGA configuration time down to $\approx$ 100ms [1]. This allows for the possibility of implementing multi-core FPGA algorithms. If a particular algorithm is too large to fit on a single FPGA core, the possibility exists to implement the algorithm in multiple stages using multiple FPGA cores, and dynamically switch between them on the fly.

As compared to Cambridge's Balloon Board v3 [11], the EyeBot M6 possesses less CPU power and RAM, but has a larger FPGA. The new EyeBot also has a faster CPU than the Qwerk board [12], though it lacks the Qwerk's floating point unit. And while notably larger than the CMUCam system, the EyeBot M6 has considerably more processing power and I/O support. In addition, while all of the other micro-controller platforms include support for cameras, none of them support stereo cameras. This allows the EyeBot M6 to be used for stereo vision applications, which none of the other platforms can achieve [2].

Chapter 4

Computer Vision

The primary function of vision is to extract enough information from an image, or series of images, to provide guidance to a host system [23]. This applies not only to organic vision systems, but also to artificial vision systems. Much research has been undertaken in the study of techniques and algorithms for extracting useful data from pictures. Since the beginning of computer vision in the late 1950s/early 1960s several methods have emerged for obtaining pertinent information from images and exposing it to the host system in an understandable format.

In more recent years, the development of reconfigurable hardware logic devices (CPLDs and FPGAs) have prompted research into implementing and accelerating image processing algorithms in hardware logic. Most image processing algorithms are both data-parallel and computation-intensive, making them well suited for implementation on FPGAs. Research has shown that the use of FPGAs in computer vision systems can lead to sizeable performance benefits [15, 24]. A number of these algorithms have been implemented in the FPGA of the new EyeBot M6.

4.1 Binary Images

The analysis of binary images is one of the simplest ways to extract meaningful data from pictures. It is particularly useful when trying to determine the location or

orientation of an object within an image. This method of object location has shown itself to be amenable to FPGA acceleration, although the implementation on the EyeBot M6 differs in several ways from that proposed by previous researchers [21].

A binary image is first constructed from the original picture by marking all the pixels which correspond to the object of interest.

$$p(x,y) = \begin{cases} 1 & \text{if } (x,y) \in \text{object}, \\ 0 & \text{if } (x,y) \notin \text{object}. \end{cases} \tag{4.1}$$

4.1.1 Object Location

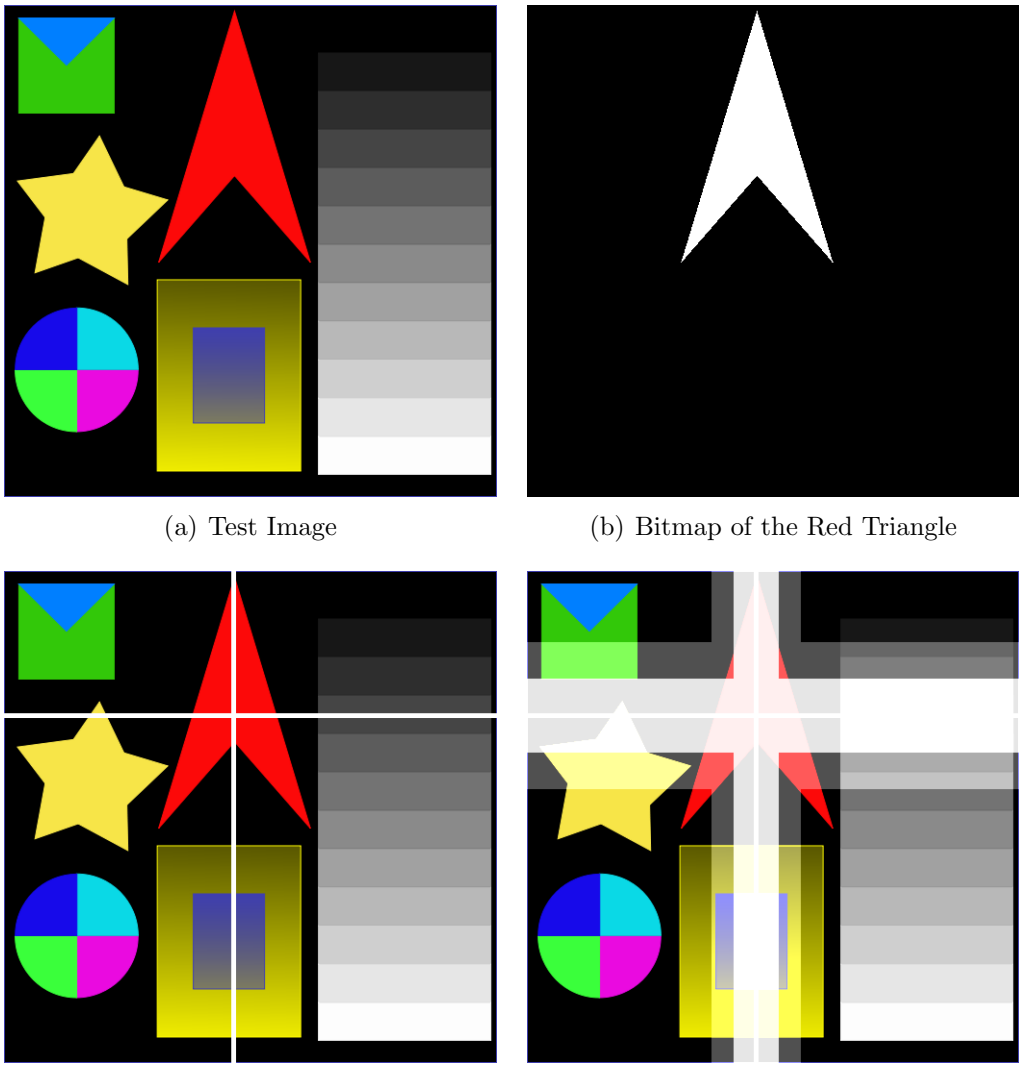
Once the bitmap has been constructed, it is a simple matter to calculate the centre of mass or centroid of the object [25]. This gives the relative position of the object with reference to an origin, usually defined as the top left corner of the picture. It may also be useful to calculate the two dimensional standard deviation, as this gives a measure of the width or spread of the object.

$$\bar{x} = \frac{\sum_{x=0}^{x_{max}} x\text{hist}_{row}(x)}{\sum_{x=0}^{x_{max}} \text{hist}_{row}(x)} \qquad \bar{y} = \frac{\sum_{y=0}^{y_{max}} y\text{hist}_{col}(y)}{\sum_{y=0}^{y_{max}} \text{hist}_{col}(y)} \tag{4.2}$$

$$\sigma_x = \sqrt{\frac{\sum_{x=0}^{x_{max}} (x-\bar{x})^2\text{hist}_{row}(x)}{\sum_{x=0}^{x_{max}} \text{hist}_{row}(x)}} \qquad \sigma_y = \sqrt{\frac{\sum_{y=0}^{y_{max}} (y-\bar{y})^2\text{hist}_{col}(y)}{\sum_{y=0}^{y_{max}} \text{hist}_{col}(y)}} \tag{4.3}$$

In Equations (4.2) and (4.3), the terms $\text{hist}_{row}(x)$ and $\text{hist}_{col}(y)$ refer to the row and column counts of the pixels in the binary image. Essentially these are the row and column histograms of the number of pixels in each row and column which belong to the object of interest.

$$\text{hist}_{row}(x) = \sum_{y=1}^{y_{max}} p(x,y) \qquad \text{hist}_{col}(y) = \sum_{x=1}^{x_{max}} p(x,y) \tag{4.4}$$



(a) Test Image (b) Bitmap of the Red Triangle

(c) Arithmetic Mean of the Red Triangle (d) Standard Deviation of the Red Triangle

Figure 4.1: Example of the Object Location Algorithm

Due to the computations required in calculating the arithmetic mean, it may be preferable to sacrifice accuracy for processing speed. For the purposes of object location, it is often sufficient to simply find the (x, y) point corresponding to the largest number of matching pixels in the row and column histograms.

$$x_{\text{mode}} = x|_{\text{hist}_{row}(x)=\max(\text{hist}_{row}(x))} \qquad y_{\text{mode}} = y|_{\text{hist}_{col}(y)=\max(\text{hist}_{col}(y))} \tag{4.5}$$

This simplifies the object location algorithm to a series of simple iterate-and-compare operations across the row and column histograms.

Figure 4.1 demonstrates the use of the object location algorithm to obtain the location of the red triangle in the test pattern.

4.1.2 Object Orientation

Another useful calculation which can be applied to binary images is to locate the axis of minimum inertia. This will give a measure of the orientation of the object in the image. Or phrased another way, it gives a measure of the amount by which an image must be rotated to orient the object to a known axis of reference.

Let θ be the angle as measured anticlockwise from the horizontal axis to the axis of minimum inertia. Let $\tilde{x} = x - \bar{x}$ and $\tilde{y} = y - \bar{y}$, shifting the coordinate system to centre around the centroid of the object of interest. The second moments of the image are then given by [26]

$$a = \sum_{\tilde{x}} \sum_{\tilde{y}} \tilde{x}^2 p(\tilde{x}, \tilde{y}) = \sum_{\tilde{x}} \tilde{x}^2 \sum_{\tilde{y}} p(\tilde{x}, \tilde{y}) = \sum_{\tilde{x}} \tilde{x}^2 \text{hist}_{row}(\tilde{x}) \tag{4.6}$$

$$b = 2 \sum_{\tilde{x}} \sum_{\tilde{y}} \tilde{x}\tilde{y} p(\tilde{x}, \tilde{y}) \tag{4.7}$$

$$c = \sum_{\tilde{x}} \sum_{\tilde{y}} \tilde{y}^2 p(\tilde{x}, \tilde{y}) = \sum_{\tilde{y}} \tilde{y}^2 \sum_{\tilde{x}} p(\tilde{x}, \tilde{y}) = \sum_{\tilde{y}} \tilde{y}^2 \text{hist}_{col}(\tilde{y}) \tag{4.8}$$

The angle of the axis of minimum inertia can then be expressed as functions of the second moments.

$$\sin(2\theta) = \frac{b}{\sqrt{b^2 + (a-c)^2}} \tag{4.9}$$

$$\cos(2\theta) = \frac{a-c}{\sqrt{b^2 + (a-c)^2}} \tag{4.10}$$

$$\tan(2\theta) = \frac{\sin(2\theta)}{\cos(2\theta)} = \frac{b}{a-c} \tag{4.11}$$

4.1.3 FPGA vs. CPU

This method of object location has the advantage of being simple, and thus easy to implement and test. More importantly, its simplicity and repetitive nature make it well suited for acceleration via a FPGA.

Consider the calculation of the row and column histograms. On a CPU these are calculated sequentially, along with any other tasks the CPU needs to perform. This means that the frame rate of the algorithm depends not only on the complexity of the algorithm and the speed of the processor, but also on the processor load. A heavily loaded processor will produce a lesser frame rate. By contrast, a FPGA is able to parallelize its operations. The construction of the row/column histograms is thus independent of any other algorithms executed by the FPGA. Not only that, but since all algorithms on the FPGA are constructed from hardware logic blocks, the FPGA is able to accomplish more per clock cycle (with regards to this algorithm) than a CPU. The effects of this are shown in Chapter 7.

In essence the FPGA is able to fully process a single pixel in the source image every clock cycle, whereas the CPU takes multiple clock cycles to accomplish the same task. In addition, it is possible to instantiate multiple object locators on the FPGA, allowing for the search for multiple objects simultaneously with little to no impact on the overall performance of the system.

4.1.4 Thresholding

The algorithms described previously all operate on an image which has already been pre-processed. Namely, they require an image which has already been reduced to a bitmap indicating pixels corresponding to the object of interest. One of the most common ways of constructing an image bitmap from a source picture is to threshold the picture. A pixel is considered *active* if its intensity falls within a certain range, and *inactive* if it falls outside this range. In the case of coloured images, a pixel can be compared to a different threshold range on each of the different colour channels. A colour pixel is considered active if it meets the threshold criteria on all of the

different colour channels. This method is a simple and efficient method of extracting objects from images [21], however its accuracy depends on the ability to distinguish the object of interest from the background. On images where the target object is similar in colour to the background it becomes quite difficult to accurately extract the object from the rest of the image.

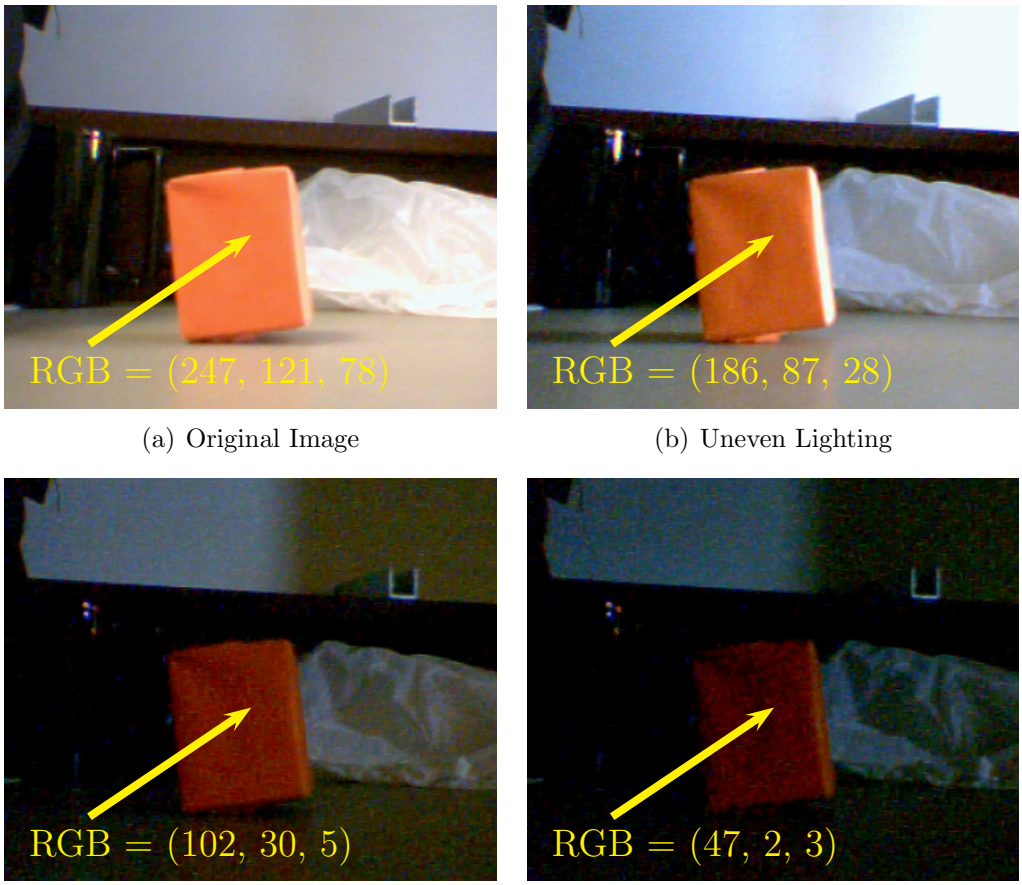
4.2 Colour Spaces

At this point, it is important to consider how coloured images are represented and stored. Coloured images are represented using a colour space, or colour model, which is defined as a "specification of a 3D colour coordinate system and a visible subset in the coordinate system within which all colours in a particular color gamut lie" [27]. Each colour space is optimised for a different application, and the same image may be represented using multiple colour spaces.

4.2.1 RGB

RGB (Red, Green, Blue) is an additive model in which red, green and blue light is combined in various amounts to create other colours [28]. One of the most common applications of RGB is the display of coloured images on CRT or LCD screens. In addition, most digital cameras, including the ones used on the EyeBot M6 [29], output data in RGB format. This representation is thus important for the input and output of coloured images.

There are, however, several problems with using the RGB colour space for image processing. The main problem lies in the colour space's close coupling between colour and brightness. Changing the values of any one RGB channel not only changes the colour of a pixel, but also its brightness. This means that the same coloured object will have vastly different RGB channel values in different lighting conditions. This is illustrated in Figure 4.2, where the same orange box is show under various lighting conditions. As can be seen, the RGB channel values of the box differ greatly as the lighting environment changes.



(a) Original Image

(b) Uneven Lighting

(c) Dark Lighting

(d) Very Dark Lighting

Figure 4.2: Examples of varying lighting conditions

When using RGB colour space for image processing, it is usual to first normalise the channel values in some fashion to decouple the image brightness from the colour information [30, 31]. An alternative technique is to convert the RGB data into a different colour space, then use the second colour space for image processing.

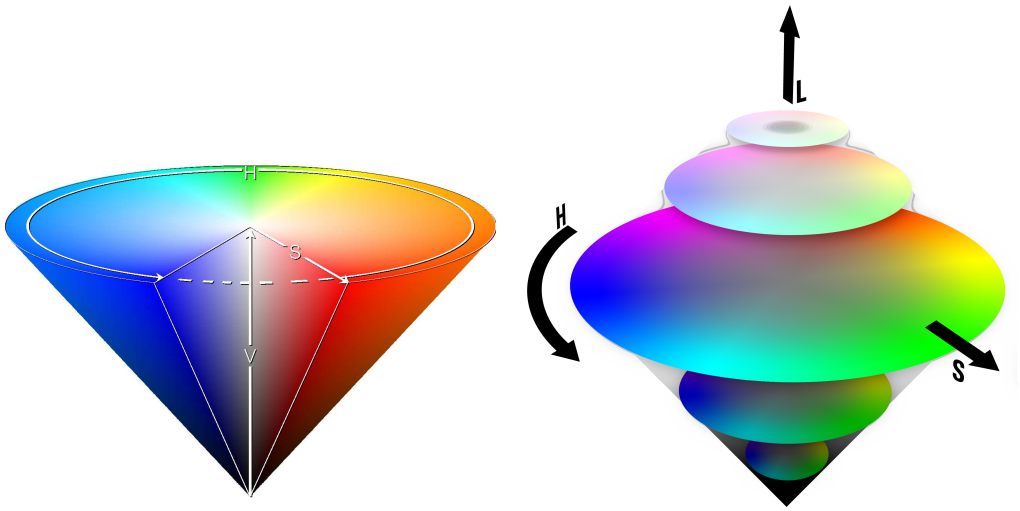
4.2.2 HSL/HSV

The HSL (Hue, Saturation, Lumience) and HSV or HSI (Hue, Saturation, Value or Intensity) colour spaces solve many of the problems associated with RGB. Both models define a coloured pixel in terms of three components [27]. Hue is the 'type' of colour (green, yellow, etc), saturation is the 'vibrancy' of the colour, and the value or luminance determines how bright a colour appears.

This is important for two reasons. Firstly, unlike RGB, HSV and HSL separate colour from brightness. Thus it is possible to locate, say, an orange box, independent of local lighting conditions. Secondly the important colour information is located in a single channel (Hue). Both these factors make HSV/HSL very good colour models for performing image processing [32]. Although the two colour spaces are very similar, HSL produces a more 'natural' looking grayscale channel (Lumience) than HSV (Value) [27].

Under HSL, the saturation component ranges from full colour to an equivalent grey value, whereas in HSV, it ranges from full colour to white. In HSL, the luminance component ranges from black, through the chosen colour, to white, whereas in HSV, the value component ranges only halfway, from black to the chosen colour. The hue component is the same for both colour spaces. This is illustrated in Figure 4.3, which shows the visualisation of both colour spaces.

The conversion from RGB to HSL colour space is non-linear, given mathematically by the following equations. Assuming the (Red, Green, Blue) channel values have been normalised to lie between 0.0 and 1.0, let MAX equal the maximum of the (Red, Green, Blue) values and MIN equal the minimum of these values.



(a) HSV Colour Space (b) HSL Colour Space

Figure 4.3: Visualisation of the HSV and HSL colour spaces [33, 34]

$$\text{Hue} = \begin{cases} \text{undefined} & \text{if MAX} = \text{MIN} \\ 60° \times \frac{\text{Green - Blue}}{\text{MAX - MIN}} + 0° & \text{if MAX} = \text{Red and Green} \geq \text{Blue} \\ 60° \times \frac{\text{Green - Blue}}{\text{MAX - MIN}} + 360° & \text{if MAX} = \text{Red and Green} < \text{Blue} \\ 60° \times \frac{\text{Blue - Red}}{\text{MAX - MIN}} + 120° & \text{if MAX} = \text{Green} \\ 60° \times \frac{\text{Red - Green}}{\text{MAX - MIN}} + 240° & \text{if MAX} = \text{Blue} \end{cases} \tag{4.12}$$

$$\text{Saturation} = \begin{cases} 0 & \text{if MAX} = \text{MIN} \\ \frac{\text{MAX - MIN}}{\text{MAX + MIN}} = \frac{\text{MAX - MIN}}{2\times\text{Lumience}} & \text{if } 0 < \text{Lumience} \leq \frac{1}{2} \\ \frac{\text{MAX - MIN}}{2-(\text{MAX + MIN})} = \frac{\text{MAX - MIN}}{2-2\text{Lumience}} & \text{if Lumience} > \frac{1}{2} \end{cases} \tag{4.13}$$

$$\text{Lumience} = \frac{1}{2}(\text{MAX} + \text{MIN}) \tag{4.14}$$

This gives $0° \leq \text{Hue} \leq 360°$, $0 \leq \text{Saturation} \leq 1$, and $0 \leq \text{Lumience} \leq 1$. Note that the condition of MAX = MIN signifies that the pixel corresponds to a grayscale value, and hence the Hue channel has no meaning. This corresponds to points lying along the central axis of the HSL colour cone.

If, as is more often the case, the (Red, Green, Blue) channel values are expressed as 8-bit quantities, these equations become

$$\text{Hue} = \begin{cases} 255 & \text{if MAX} = \text{MIN} \\ 42 \times \frac{\text{Green - Blue}}{\text{MAX - MIN}} + 42 & \text{if MAX} = \text{Red} \\ 42 \times \frac{\text{Blue - Red}}{\text{MAX - MIN}} + 126 & \text{if MAX} = \text{Green} \\ 42 \times \frac{\text{Red - Green}}{\text{MAX - MIN}} + 210 & \text{if MAX} = \text{Blue} \end{cases} \tag{4.15}$$

$$\text{Saturation} = \begin{cases} 0 & \text{if MAX} = \text{MIN} \\ 127 \times \frac{\text{MAX - MIN}}{\text{Lumience}} & \text{if } 0 < \text{Lumience} \leq 127 \\ 127 \times \frac{\text{MAX - MIN}}{255 - \text{Lumience}} & \text{if Lumience} > 127 \end{cases} \tag{4.16}$$

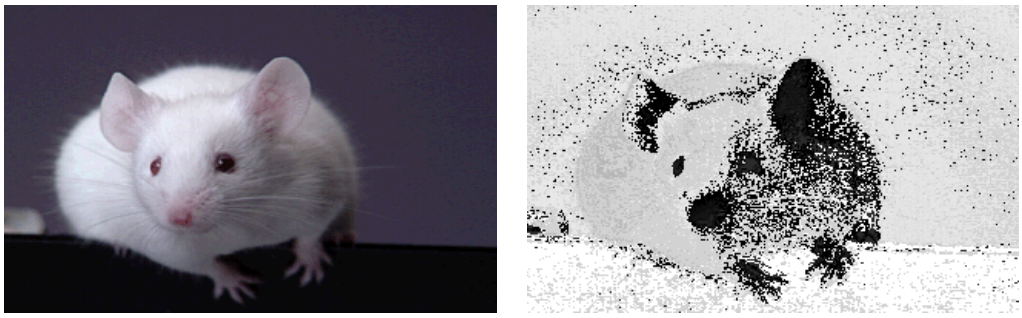
$$\text{Lumience} = \frac{1}{2}(\text{MAX} + \text{MIN}) \tag{4.17}$$

This gives $0 \leq \text{Hue} \leq 252$, $0 \leq \text{Saturation} \leq 255$, and $0 \leq \text{Lumience} \leq 255$, with Hue = 255 signifying a grayscale pixel. In addition to being scaled, the Hue values have been rotated by 60° in order to reduce the number of case statements.

Figure 4.4 illustrates the decomposition of a RGB image into separate Hue, Saturation and Lumience channels. It can be seen that areas of similar colour on the original image correspond to similar Hue values, vibrant areas of the image correspond to larger Saturation values, and the Lumience channel is essentially a grayscale version of the original image.

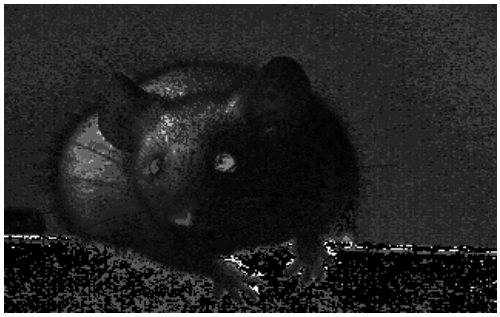
Figure 4.5 shows the same images from Figure 4.2, but the pixels are now analysed using the HSL colour space. As can be seen from the images, the Hue component changes only slightly under varying lighting conditions. The white lines indicate the results of running the object location algorithm over the images. It can be seen that using the Hue channel allows the algorithm to locate the orange box (target Hue value = 54 ± a tolerance of 5) under all but the darkest lighting conditions.

The principle difficulty with implementing the RGB to HSL conversion on the FPGA

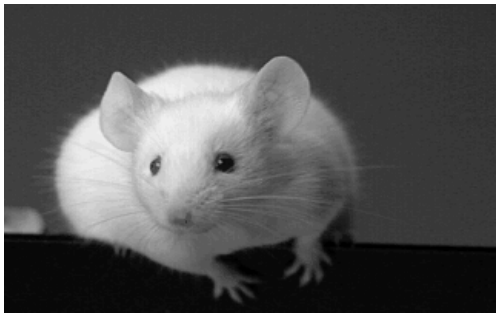


(a) Original Image

(b) Hue



(c) Saturation



(d) Lumience

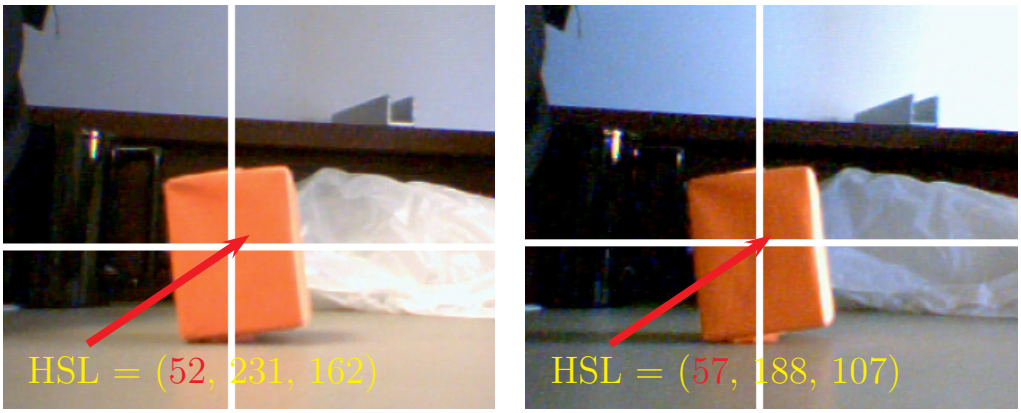
Figure 4.4: Example of HSL Channel Decomposition

lies in the division operation. Xilinx's Spartan-3E FPGAs (indeed most FPGAs) lack inbuilt hardware division units. This presents a problem, since the calculation of all three HSL channels from RGB data involve some form of division. This problem will be explored further in Chapter 6, and a solution outlined.

4.2.3 YCbCr

An alternative colour space is YCbCr, which represents an image using one luminance (brightness, Y) channel, and two chromiance (colour, Cb and Cr) channels [35]. This colour space is used to encode images for television, as well as for JPEG and MPEG encoding.

Multiple, highly similar yet subtly different, standards exist for YCbCr. Even the name of the colour space changes from standard to standard. YCbCr is sometimes referred to by YUV, or YPbPr, which are also slightly different standards from the 'main' YCbCr standard.



(a) Original Image

(b) Uneven Lighting

(c) Dark Lighting

(d) Very Dark Lighting

Figure 4.5: Examples of varying lighting conditions using HSL Colour Space

The RGB to YCbCr colour space conversion specified by the JPEG standard is given by the following formulae [36].

$$\mathrm{Y} = 0.299 \times \mathrm{R} + 0.587 \times \mathrm{G} + 0.114 \times \mathrm{B} \tag{4.18}$$

$$\mathrm{Cb} = 128 - 0.168736 \times \mathrm{R} - 0.331264 \times \mathrm{G} + 0.5 \times \mathrm{B} \tag{4.19}$$

$$\mathrm{Cr} = 128 + 0.5 \times \mathrm{R} - 0.418688 \times \mathrm{G} - 0.081312 \times \mathrm{B} \tag{4.20}$$

Where (R, G, B) and (Y, Cb, Cr) are 8-bit quantities. This can alternatively be expressed in matrix form as

$$\begin{bmatrix} \mathrm{Y} \\ \mathrm{Cb} \\ \mathrm{Cr} \end{bmatrix} = \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} + \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{bmatrix} \begin{bmatrix} \mathrm{R} \\ \mathrm{G} \\ \mathrm{B} \end{bmatrix} \tag{4.21}$$

Figure 4.6 illustrates the decomposition of a RGB image into separate Y, Cb and Cr channels. The Y (Luma) channel is similar to the Lumience channel in HSL/HSV in that it is essentially a grayscale version of the image. The Cb and Cr (Chromiance) channels encode the colour components of the original image. The subdued appearance of the chromiance channels illustrates that the human eye is more sensitive to changes in lumiance than changes in chromiance [35]. This allows for a decimation in the chromiance channels with little perceptual loss of information, hence the use of YCbCr colour encoding in lossy image formats such as JPEG.

Apart from its use in image formats, YCbCr is also important in image processing. The YCbCr colour space has been found to be very effective for skin detection [31]. Chai and Bouzerdoum have claimed that the YCbCr colour space provides a good coverage of human skin tones, and that pixels associated with skin tones have similar Cb and Cr values [37]. This can seen in Figure 4.6 where skin regions in the original image show little variation in intensity in the decomposed Cb and Cr images.

Implementing the RGB to YCbCr conversion on the FPGA is a matter of repetitive multiplications against the input RGB data, involving constant coefficients. Unfor-

(a) Original Image

(b) Y (Luma)

(c) Cb (Chroma-Blue)

(d) Cr (Chroma-Red)

Figure 4.6: Example of YCbCr Channel Decomposition

0	-1	0
-1	4	-1
0	-1	0

(a) Laplace operator

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

(b) Horizontal Sobel operator

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

(c) Vertical Sobel operator

Figure 4.7: 3×3 Convolution kernels

tunately, the coefficients of the conversion matrix are all fractional quantities and the FPGA has no standard way of dealing with floating point numbers. The solution is to perform the operations using fixed point arithmetic, which will be further elaborated on in Chapter 5.

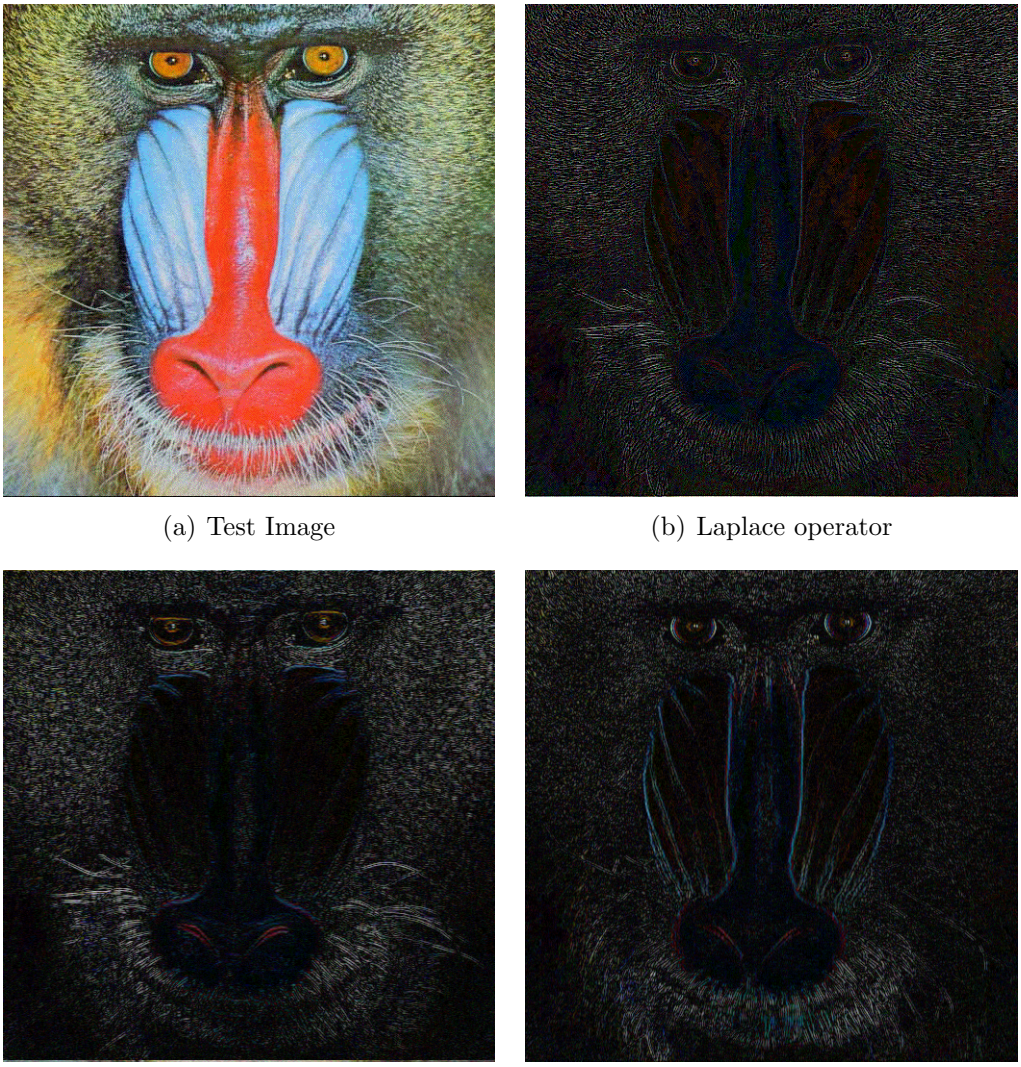
4.3 Edge and Corner Detection

Edges in an image represent the boundaries between objects and the background, while corners represent the intersection of two or more edges. In image processing, edge points are the points in the image where the difference in intensities of neighbouring pixels is at a maximum. Similarly, corner points are points in the image for which the curvature of edge lines is maximised.

Two of the most common edge detection algorithms are the Laplace and Sobel operators, both of which can be defined in terms of 3×3 convolution kernels, as shown in Figure 4.7. The results of applying these operators to an image is shown in Figure 4.8.

4.3.1 Convolution

A large number of image processing algorithms can be represented as filters in the frequency domain. Blurring, for example, can be represented as a low-pass filter, and sharpening by a high-pass filter. One method of applying these filters is to first convert the image into its frequency domain representation using a Fourier



(a) Test Image

(b) Laplace operator

(c) Horizontal Sobel operator

(d) Vertical Sobel operator

Figure 4.8: Examples of the Laplace and Sobel edge detection operators

Transform, then apply the desired filters using multiplication.

The disadvantage of this approach is that implementations of the Fourier Transform can occupy large amounts of FPGA real estate. The Radix-2 Fast Fourier Transform (FFT) algorithm is noted as consuming "45% of the slices and 30% of the block RAM available on the Virtex-2000E FPGA chip" [38]. This presents a problem, since the Virtex-2000E chip possesses over 4 times as many logic cells, and 3 times as much block RAM, as the Spartan3-500E chip used in this project [39, 40].

An alternative to computing the Fourier Transform is to convolve the image instead. The convolution theorem states that multiplication in the frequency domain is equivalent to convolution in the time domain. For small sized images, convolution is often faster and easier than the Fourier Transform [38]. The two-dimensional discrete convolution is performed by applying a mask or kernel to the image using a series of multiplications and additions. For example, a 3×3 kernel (k_{ij}) applied to the pixel at position i_{22}

$$\begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & \mathbf{k_{22}} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix} \cdot \begin{bmatrix} i_{11} & i_{12} & i_{13} \\ i_{21} & \mathbf{i_{22}} & i_{23} \\ i_{31} & i_{32} & i_{33} \end{bmatrix} = \begin{array}{c} k_{11}i_{11} + k_{12}i_{12} + k_{13}i_{13} \\ +k_{21}i_{21} + \mathbf{k_{22}i_{22}} + k_{23}i_{23} \\ +k_{31}i_{31} + k_{32}i_{32} + k_{33}i_{33} \end{array} \tag{4.22}$$

The kernel is then shifted and applied to the next pixel, and so on until the entire image has been convolved. At a worst case, a 3×3 kernel requires nine multiplications for almost every pixel in the image. This is the kind of repetitive task that can occupy large amounts of CPU time, yet be very amenable to hardware acceleration via a FPGA. Further details on FPGA acceleration of the convolution operation on the EyeBot M6 can be found in (English, 2006) [2].

4.3.2 SUSAN

The SUSAN algorithm is outlined in a paper by Smith and Brady [41] as a method of performing both edge and corner detection simultaneously. The algorithm does not rely on convolutions, although it can be calculated using masks of varying sizes.

This is typically a circular mask of 37 pixels, but the algorithm can also be applied using a 3×3 pixel mask. The mask is first placed at each point in the image, and for each point the intensity of every pixel within the mask is compared to the intensity of the centre point [41]. If r_0 is the position of the centre point, r is the position of any other pixel within the mask, $I(r)$ is the intensity of pixel r, t is some threshold value, then the output of the comparison, c, is given by:

$$c(r, r_0) = \begin{cases} 1 & \text{if } |I(r) - I(r_0)| \leq t, \\ 0 & \text{if } |I(r) - I(r_0)| > t, \end{cases} \tag{4.23}$$

The number of pixels within the mask with similar intensity to the central pixel can be obtained by summing c.

$$n(r_0) = \sum_r c(r, r_0) \tag{4.24}$$

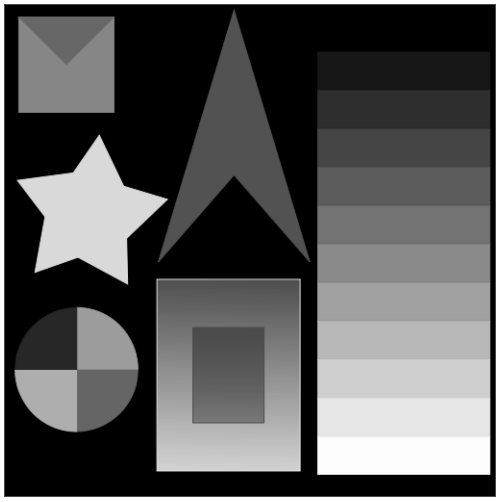
This number is referred to as the USAN (Univalue Segment Assimilating Nucleus) area of point r_0 [41].

The USAN area is then compared to a constant geometric threshold, g, to obtain an edge or corner response. A threshold of $g = 3n_{max}/4$ is used for edge detection, and $g = n_{max}/2$ is used for corner detection.

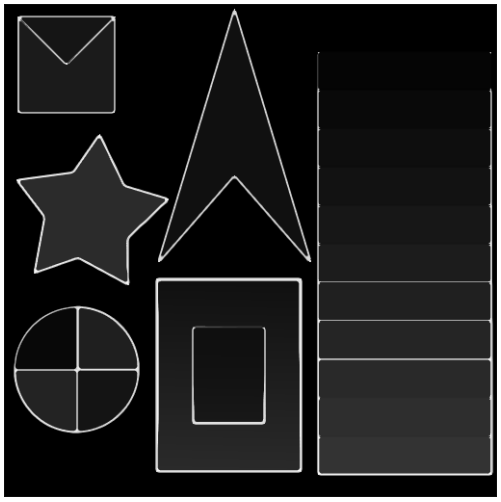
$$R(r_0) = \begin{cases} g - n(r_0) & \text{if } n(r_0) < g, \\ 0 & \text{otherwise.} \end{cases} \tag{4.25}$$

This is sufficient to extract the edges from the image. Eliminating the false positives from the initial corner response requires some further processing. Figure 4.9 shows the results of executing the SUSAN algorithm over a test pattern.

The SUSAN algorithm has been shown to be well suited to FPGA acceleration. Frame rates of over 100 frames per second have been reported when the algorithm was implemented on a Xilinx XVC50-6 Virtex FPGA [42].



(a) Test Image



(b) SUSAN Edges

(c) SUSAN Corners

Figure 4.9: Example of the SUSAN algorithm

Chapter 5

Fixed Point Arithmetic

Fixed point notation is a method of representing fractional numbers using only integer storage. This representation is important since the FPGA has no inbuilt support for floating point numbers. Using floating point numbers would entail the construction of hardware floating point arithmetic units on the FPGA, then the replacement of all arithmetic operations in the VHDL modules to use the new floating point units. This would consume large amounts of FPGA resources, and increase the time required to perform any calculation. By using fixed point representation instead, it is possible to perform all the standard arithmetic operations (addition, subtraction, multiplication and division) on fractional numbers by using the FPGA's inbuilt integer arithmetic units.

Essentially a fixed point number is simply an integer with an invisible radix (decimal or binary) point [43]. Consider the number 153.974. It can be represented by an integer 153974, but remembering that the decimal point lies before the 3^{rd} digit. The placement of the radix point is completely arbitrary, and depends on the numbers being represented. Since the radix point is not stored as part of the number representation, it must be tracked separately by the programmer.

There are several notations in use to represent the bitwidth and location of the radix point in fixed point numbers. The most common notation is Q-format, where the number after the Q represents the number of fractional bits. For example Q17 represents a number with 17 fractional bits. A less ambiguous form of the Q notation

is Q$x.y$, when 'x' is the number of bits to the left of the radix point (representing the integer part of the number) and 'y' is the number of bits to the right of the radix point (the fractional part of the number). So Q2.16 would represent a number with 2 integer bits and 16 fractional bits stored in an 18-bit word.

5.1 Fixed Point Format

The number of bits allocated to the integer and fractional components of a fixed point variable depend greatly on the range and precision demanded of the variables in a particular algorithm [44, 45]. For an unsigned variable α, the number of integer bits (QI) must fulfil the expression

$$0 \leq \alpha \leq 2^{QI} \tag{5.1a}$$

$$QI = \lceil \log_2(\lfloor \alpha_{\max} \rfloor) \rceil \tag{5.1b}$$

If the algorithm requires the use of signed (two's complement) variables, then this equation must be modified to include a single bit used to represent the sign of the variable. The relationship for signed variables ($\pm\alpha$) is defined by the expression

$$-2^{QI-1} \leq \alpha \leq 2^{QI-1} \tag{5.2a}$$

$$QI = \lceil \log_2(\lfloor \max(|\alpha_{\max}|, |\alpha_{\min}|) \rfloor) \rceil + 1 \tag{5.2b}$$

In this case QI also includes the sign bit.

While the size of the integer component determines the range of a fixed point variable, the size of the fractional component determines the variable's resolution. For a given resolution ϵ the number of fractional bits (QF) required is given by:

$$\epsilon = \frac{1}{2^{QF}} \tag{5.3a}$$

$$QF = \left\lceil \log_2 \left(\frac{1}{\epsilon} \right) \right\rceil \tag{5.3b}$$

The integer and fractional bits are combined to form a single word ($WL = QI+QF$) representing the fixed point variable. This gives the full range of the variable as

$$0.0 \leq \alpha \leq 2^{QI} - 2^{-QF} \tag{5.4}$$

for an unsigned variable and

$$-2^{QI-1} \leq \alpha \leq 2^{QI-1} - 2^{-QF} \tag{5.5}$$

for a signed two's complement variable. It can be seen that there will be a trade-off between range and resolution in order to fit the fixed point number into a given word size.

5.2 Converting Floating Point to Fixed Point

Once an appropriate fixed point format has been selected, the fixed point approximation for the floating point variable can be calculated. This is a simple matter of scaling the floating point number by 2 raised to the power of the number of fractional bits.

$$\text{Fixed Point} = \lfloor \text{Floating Point} \times 2^{QF} \rfloor \tag{5.6}$$

Since the fixed point representation is restricted to integer storage, the fractional part of the scaled floating point number must be discarded. The most common method is to simply discard the leftover fraction by flooring the scaled floating point number, however greater accuracy can be obtained by rounding the number instead.

$$\text{Fixed Point} = \text{Round}(\text{Floating Point} \times 2^{QF}) \tag{5.7}$$

5.3 Elementary Arithmetic Operations

Arithmetic with fixed point numbers is much the same as arithmetic with integers, and can be performed using the same hardware integer arithmetic units. The two differences which need to be considered are alignment of the radix point during addition and subtraction, and tracking the movement of the radix point during multiplication and division.

In order to add or subtract two fixed point numbers and obtain a meaningful result, the radix point of both numbers must first be aligned. One of the operands must be left or right shifted, which will discard the information stored in the upper (in the case of a left shift) or lower (in the case of a right shift) bits. When left shifting, the lower bits should be filled with zeros, and when right shifting the upper bits should be filled with the sign bit (or zero in the case of an unsigned number). In addition, when shifting signed numbers, care must be taken that the sign bit does not change.

Bit shifting unsigned fixed point numbers

$$\text{Left shift by 2}$$
$$|x_1|x_0|y_0|y_1|y_2|y_3|y_4|y_5| \Longrightarrow |y_0|y_1|y_2|y_3|y_4|y_5|0|0| \tag{5.8}$$

$$\text{Right shift by 2}$$
$$|x_1|x_0|y_0|y_1|y_2|y_3|y_4|y_5| \Longrightarrow |0|0|x_1|x_0|y_0|y_1|y_2|y_3| \tag{5.9}$$

Bit shifting signed, two's complement, fixed point numbers

$$\text{Left shift by 2}$$
$$|s|x_0|y_0|y_1|y_2|y_3|y_4|y_5| \Longrightarrow |s|y_1|y_2|y_3|y_4|y_5|0|0| \tag{5.10}$$

$$\text{Right shift by 2}$$
$$|s|x_0|y_0|y_1|y_2|y_3|y_4|y_5| \Longrightarrow |s|s|s|x_0|y_0|y_1|y_2|y_3| \tag{5.11}$$

Once the two numbers have been radix aligned, the standard integer addition and subtraction operations may be performed to produce a result of the same word size

and fixed point alignment.

Multiplication and division are similar, but do not require that their operands be radix aligned first. Instead the radix point moves during the operation, producing a result that is of a different alignment to the initial operands.

The size of a product is equal to the sum of the size of the original multiplicands. This applies to the word size, the number of bits in the integer component, and the number of bits in the fractional component. In Q-format notation

$$QM_1.F_1 \times QM_2.F_2 = Q(M_1 + M_2).(F_1 + F_2) \tag{5.12}$$

Since it is often necessary to perform further operations using the original word size of the multiplicands, the result of the multiplication will need to be truncated. It is customary to keep the result in the same format as the original multiplicands, hence preserving the bits clustered close to the radix point, which is often where the most useful information lies.

Of the four basic arithmetic operations, division is the most involved. Integer division units produce both a result (quotient) and a remainder. It is self evident that if both the dividend and the divisor are integers, then the quotient and the remainder will both be integers.

$$\frac{\text{Dividend}}{\text{Divisor}} = \text{Quotient} + \frac{\text{Remainder}}{\text{Divisor}} \tag{5.13}$$

If the divisor is an integer, but the dividend is a fixed point number, then both the quotient and the remainder will possess a fractional component equal in size to the fractional component of the dividend.

$$\frac{\text{Dividend} \times 2^{QF}}{\text{Divisor}} = \text{Quotient} \times 2^{QF} + \frac{\text{Remainder} \times 2^{QF}}{\text{Divisor}} \tag{5.14}$$

If the divisor is a fixed point number with QF fractional bits, then this is equivalent to right shifting the dividend by QF (dividing by 2^{QF}). This will reduce the size of the fractional component of both the result and the remainder, and may even cause

them to have effectively a negative number of fractional bits.

$$\frac{\text{Dividend}}{\text{Divisor} \times 2^{QF}} = \text{Quotient} \times 2^{-QF} + \frac{\text{Remainder} \times 2^{-QF}}{\text{Divisor}} \tag{5.15}$$

Since the quotient and remainder are effectively fixed width windows into the true value of the quotient and remainder, bitshifting the dividend and divisor in this fashion will change the accuracy of the result of the division. In particular, increasing the fractional component of the dividend will increase the resolution of the quotient, whereas increasing the fractional component of the divisor will have the opposite result.

In order to maximise the accuracy of the hardware division units, this project has designed the divisors to divide a 16-bit dividend by an 8-bit divisor. Although both dividend and divisor are really 8-bit quantities, this means that the dividend can be bitshifted upwards by 8-bits, thus gaining an extra 8-bits of accuracy in the quotient. Since this project is primarily concerned with using division to compute the RGB to HSL colour space conversion algorithm, the focus lies on the quotient, and not the remainder.

Chapter 6

Division Algorithms

As mentioned previously, the main difficulty in implementing the RGB to HSL conversion on the FPGA is the lack of hardware divisors. The calculation of all three HSL channels from the RGB source involve some form of division. The division by two in the Lumience channel can be replace by a simple bit-shifting operation, but the divisions in the Hue and Saturation channels require a proper divisor. FPGAs do not come with embedded hardware for performing division, so a divisor needs to be constructed from the FPGA's logic blocks.

Since floating-point arithmetic requires too many FPGA resources, the division operation has to be approximated using fixed point arithmetic and integer hardware division units. The RGB to HSL conversion operates on 8-bit quantities, so it is desirable to use a hardware division unit that is optimised for processing byte sized numbers. This presents a slight problem, since much of the current research into hardware arithmetic units focuses either on numbers with large word sizes [46] or floating point numbers [47]. In addition it is desirable to have a divisor with a high degree of accuracy, fully pipelined implementation, and a short pipeline depth.

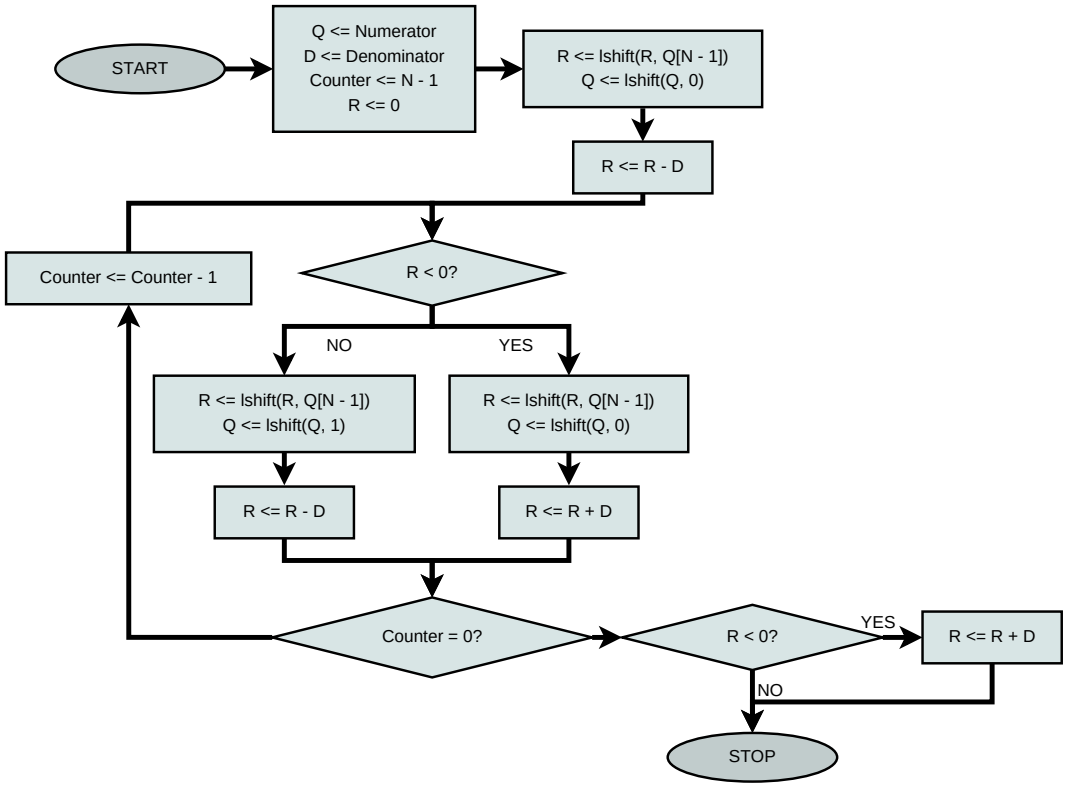


Figure 6.1: Dataflow diagram for the Radix-2 Non-Restoring Divisor [43]

6.1 Radix-2 Non-Restoring Divisor

The standard algorithm for performing integer division is the Radix-2 Non-Restoring Divisor. This algorithm is shown in diagrammatic form in Figure 6.1. The algorithm divides an N bit numerator by an M bit denominator to produce an N bit quotient and an M bit remainder.

$$\text{Quotient}_{\ N\text{-bit}} = \left\lfloor \frac{\text{Numerator}_{\ N\text{-bit}}}{\text{Denominator}_{\ M\text{-bit}}} \right\rfloor \tag{6.1a}$$

$$\text{Remainder}_{\ M\text{-bits}} = \text{Numerator}_{\ N\text{-bit}} \bmod \text{Denominator}_{\ M\text{-bit}} \tag{6.1b}$$

Expressed in another way,

$$\frac{\text{Numerator}_{\ N\text{-bit}}}{\text{Denominator}_{\ M\text{-bit}}} = \text{Quotient}_{\ N\text{-bit}} + \frac{\text{Remainder}_{\ M\text{-bits}}}{\text{Denominator}_{\ M\text{-bit}}} \tag{6.2}$$

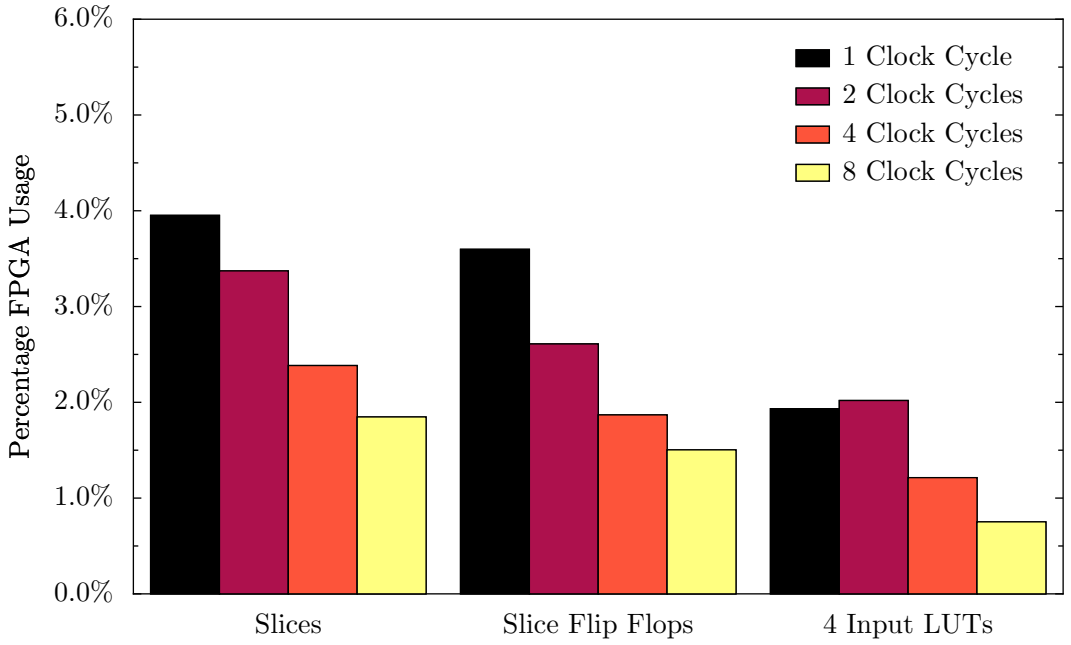


Figure 6.2: Spartan3-500E Resource Usage for the Non-Restoring Divisor

This algorithm is widely used on FPGAs, and implementations are readily available [48]. Despite this, there are several problems with the algorithm. The first problem is with the central loop, as shown in Figure 6.1. The algorithm requires one execution of the loop for every bit in the quotient. This means the algorithm is well suited to long pipelines, since every pass through the loop can be pipelined to a separate clock cycle. Unfortunately this same loop structure also detrimentally affects the performance of short pipelined implementations. The main purpose of pipelining is to increase the clock speed of hardware units at the expense of requiring more clock cycles to produce a result. The FPGA on the EyeBot M6 runs at a clock speed of only 50MHz. Increasing the pipeline depth for the sake of clock speed has no effect once the clock speed of the FPGA module exceeds the threshold clock speed of 50Mhz. At this point a longer pipeline merely delays the time between receiving the inputs and producing an output.

Figure 6.1 shows the percentage resource usage of the Spartan3-500E FPGA when instantiating the non-restoring divisors provided by Xilinx CORE Generator. Since the divisor is not fully pipelined, it not only requires multiple clock cycles to produce each result, but cannot process any more inputs until the first has finished. As can be

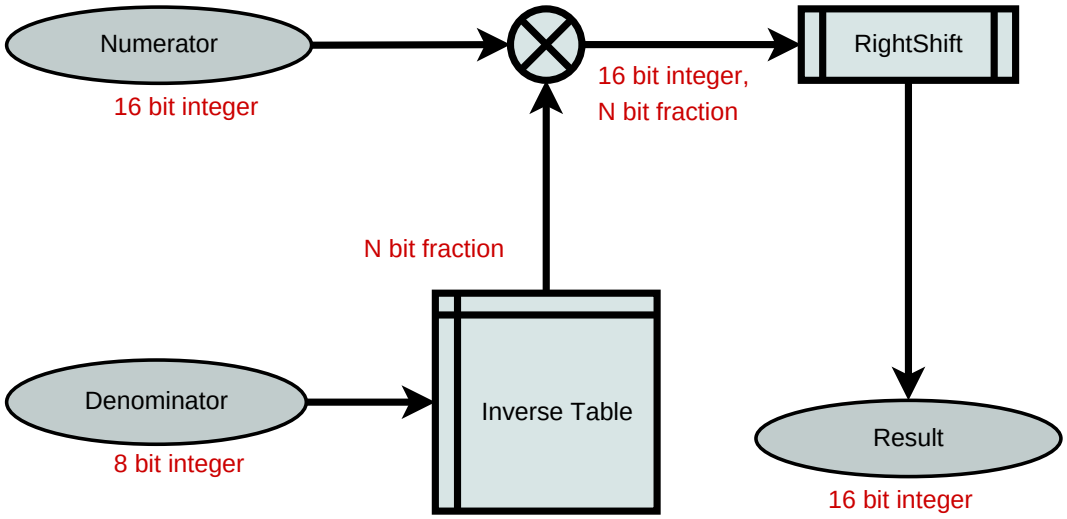


Figure 6.3: Dataflow diagram for the basic Lookup Table Divisor

seen, the amount of FPGA resources required is inversely proportional to the amount of time required for the divisor to produce a result. Implementations with a short pipeline require a noticeably greater amount of FPGA space than implementations with longer pipelines. Instead, what is desired is a divisor that is able to produce an output every clock cycle, yet only requires a few clock cycles to produce each result, at the same time requiring few FPGA resources.

The second problem with the Non-Restoring Divisor is that the result it produces is truncated integer division. This means that when calculating Num / Denom it actually produces the result $\lfloor$ Num / Denom $\rfloor_{N\text{-bits}}$, that is, it calculates the floor of the division to N-bits. While this works well in most cases, it is suboptimal in the case where integer division is used to approximate floating-point division. In this case the result of the division should be as close to the 'true' result of Num / Denom as possible.

6.2 Lookup Table Divisor

In order to solve the problems of both pipeline depth and accuracy, a different divisor was designed for this project. The dataflow diagram for this algorithm, the Lookup Table Divisor, is shown in Figure 6.3. It relies on the fact that division is equivalent

to multiplication by the inverse of the denominator. By using a precomputed lookup table of inverses, the divisor in essence uses a hardware multiplier (of which the FPGA has several) to implement a hardware divisor (of which the FPGA has none).

The entries in the inverse table are computed according to the equation

$$\text{Inverse}\left[\text{Denominator}\right] = \text{Round}\left(\frac{1}{\text{Denominator}} \times 2^N\right) \tag{6.3}$$

The division is then performed by multiplying the numerator by the inverse of the denominator, then right-shifting to align the result to the same radix point as the original numerator.

$$\text{Result} = \left\lfloor \frac{\text{Numerator} \times \text{Inverse}[\text{Denominator}]}{2^N} \right\rfloor \tag{6.4a}$$

$$= \left\lfloor \frac{\text{Numerator} \times \text{Round}\left(\frac{1}{\text{Denominator}} \times 2^N\right)}{2^N} \right\rfloor \tag{6.4b}$$

Through careful calculation of the values in the lookup table, this divisor can produce results closer to the true value than those produced by the Non-Restoring Divisor.

In Figure 6.3 and Equations (6.3) and (6.4) the bitwidth of the inverse table is left unspecified. The choice of bitwidth depends on several factors, including the availability of FPGA resources, and the maximum bitwidth of the FPGA's hardware multipliers. The Xilinx Spartan-3E hardware multipliers have a maximum individual bitwidth of 17 for unsigned numbers or 18 for signed numbers — this includes the sign bit. In addition, the choice of bitwidth is complicated by the fact that division by certain numbers requires less bits than division by other numbers.

Consider the case of a division by 4. To minimise the size of the inverse requires finding the minimum number of bits (minimise N) required to express $1/4 \times 2^N$ as an integer quantity. It is easily be seen that $1/4 = 2^{-2}$, and hence the inverse can

be expressed as

$$\begin{aligned} \text{Inverse}[4] &= \text{Round}\left(\frac{1}{4} \times 2^N\right) \\ &= \text{Round}\left(2^{-2} \times 2^2\right) \\ &= 1|_{2-bits} \end{aligned} \tag{6.5}$$

In other words, only a single bit is required to store the inverse of 4, since all the work is done by the later bitshift right by 2 bits (equivalent to a division by 4). It is intuitively obvious that a division by any power of 2 can be expressed by a multiplication by 1, then a subsequent bitshift by N bits, where $N = \log_2(\text{Denominator})$.

Following on from this point, it is theorised that there is a specific bitwidth for every possible denominator for which the accuracy is maximised. From Equation (6.4), the result of dividing n by d using an N-bit inverse is

$$\text{Result}(n, d, N) = \left\lfloor \frac{n \times \text{Round}\left[\frac{1}{d} \times 2^N\right]}{2^N} \right\rfloor \tag{6.6}$$

Thus the magnitude of the error between the result output by the divisor and the true value of dividing n by d is

$$\text{Error}(n, d, N) = \left|\text{Result}(n, d, N) - \frac{n}{d}\right| \tag{6.7}$$

Thus for any particular combination of denominator and N, the total error value is obtained by summing Equation (6.2) for all possible values of the numerator.

$$\text{Total Error}(d, N) = \sum_{n=1}^{n_{max}} \text{Error}(n, d, N) \tag{6.8a}$$

$$= \sum_{n=1}^{n_{max}} \left|\text{Result}(n, d, N) - \frac{n}{d}\right| \tag{6.8b}$$

$$= \sum_{n=1}^{n_{max}} \left| \left\lfloor \frac{n \times \text{Round}\left[\frac{1}{d} \times 2^N\right]}{2^N} \right\rfloor - \frac{n}{d} \right| \tag{6.8c}$$

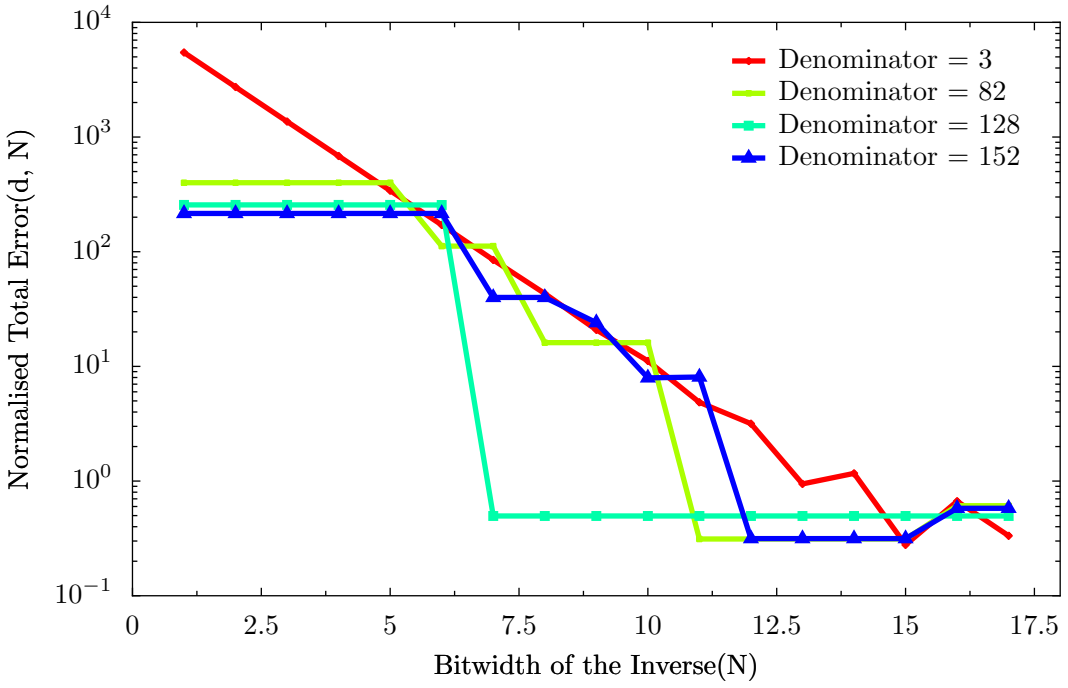


Figure 6.4: Normalised Total Error of the Lookup Divisor for different denominators and varying bitwidth of the inverse

$$\therefore \text{Normalised Total Error}(d, N) = \frac{1}{n_{max}} \sum_{n=1}^{n_{max}} \left| \left\lfloor \frac{n \times \text{Round}\left[\frac{1}{d} \times 2^N\right]}{2^N} \right\rfloor - \frac{n}{d} \right| \quad (6.8d)$$

Figure 6.4 describes the change in the Normalised Total Error for different denominators as the bitwidth used to store the inverse changes. As can be seen from the graph, the total error is very high for low bitwidths, but decreases as the bitwidth increases, as expected. Also apparent from the graph is that for any particular denominator, there is a particular bitwidth beyond which the accuracy of the division does not improve. Indeed, as can be seen, increasing the bitwidth beyond this optimum point may even *decrease* the accuracy of the division.

By carefully varying the bitwidth of the inverse table for each denominator, it is thus possible to improve the accuracy of the divisor over using a fixed bitwidth for the entries in the inverse lookup table.

Using a fixed inverse bitwidth of 17 (the largest bitwidth of unsigned integers that can fit in a single Spartan-3E hardware multiplier), produces a normalised total

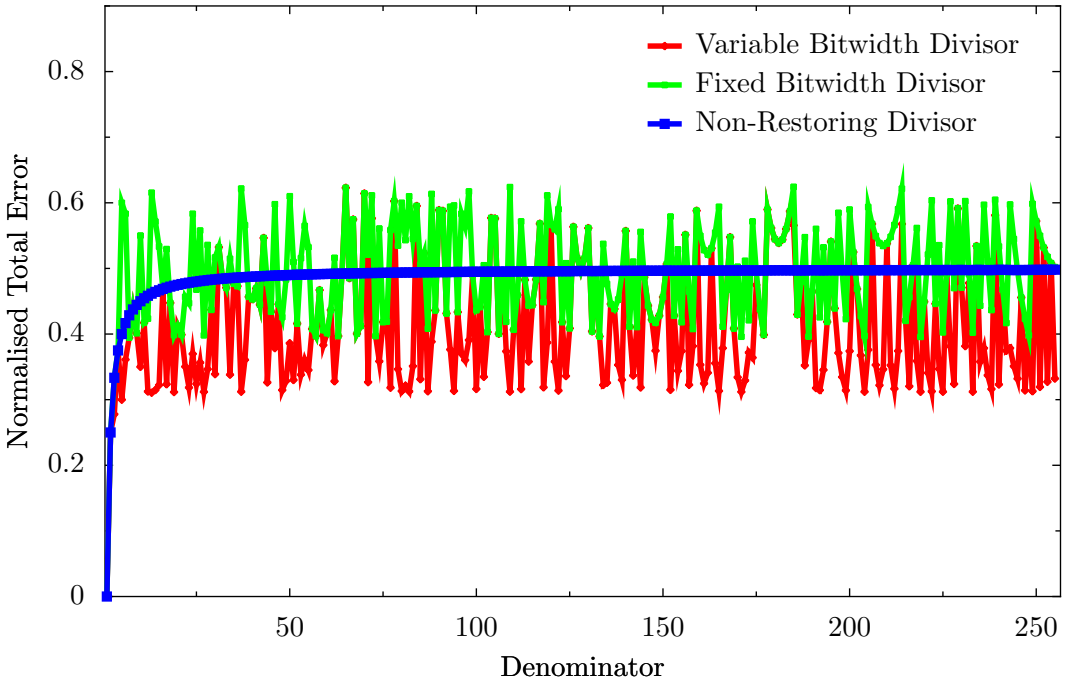


Figure 6.5: Normalised Total Error of the Lookup and Non-Restoring Divisors

error of

$$\text{Normalised Total Error}(d, \text{Fixed Width}) = \frac{1}{n_{max}} \sum_{n=1}^{n_{max}} \left| \left\lfloor \frac{n \times \text{Round}\left[\frac{1}{d} \times 2^{17}\right]}{2^{17}} \right\rfloor - \frac{n}{d} \right| \tag{6.9}$$

And from Equation (6.1a) the normalised total error of the Non-Restoring Divisor is given by

$$\text{Normalised Total Error}(d, \text{Non-Restoring}) = \frac{1}{n_{max}} \sum_{n=1}^{n_{max}} \left| \left\lfloor \frac{n}{d} \right\rfloor - \frac{n}{d} \right| \tag{6.10}$$

These three normalised total error values are plotted in Figure 6.5. As can be seen, the error of the Non-Restoring divisor is very constant for all values of the denominator. This is because the Non-Restoring divisor always rounds the result down to the nearest integer, discarding the fractional component entirely. Considering the fact that the result of most divisions will have a fractional component, then on average half of these divisions will have a fractional component of ≤ 0.5, and the other half will have a fractional component of > 0.5. Thus the average fractional

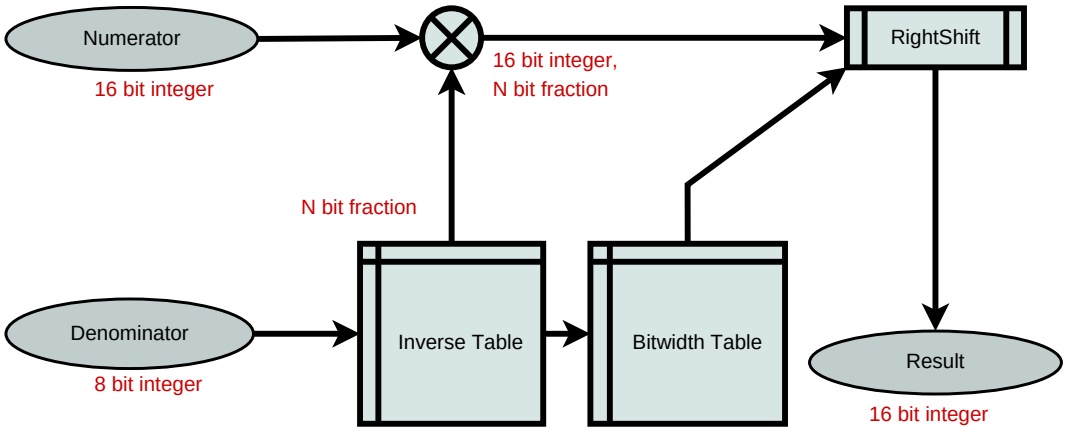


Figure 6.6: Dataflow diagram for the Lookup Table Divisor with Variable Bitwidth inverse

component of the division results will be ≈ 0.5, which as shown in the graph, is the normalised error value of the Non-Restoring Divisor.

By contrast, the error value of the Lookup Table Divisor fluctuates greatly depending on the denominator. There are two sources of error in the Lookup Divisor algorithm. The first is due to the choice of value entered into the inverse table. Since the inverse table can only store integers, parts of the fractional component of the bitshifted inverse $(1/d \times 2^N)$ are lost. Rounding the inverse preserve greater accuracy over truncating it, but the inverses of many numbers are not rational in radix-2 representation.

The other source of error in the Lookup Divisor algorithm comes from truncating the result. Unfortunately, because of the variability in the stored inverse, the error result from truncation is not readily obvious.

Nevertheless, it can be seen that the Variable Bitwidth version of the Lookup Divisor algorithm gives superior results (less average error) than the Fixed Bitwidth version. Furthermore, although the average error of the Fixed Bitwidth version fluctuates around the error of the Non-Restoring Divisor, the average error of the Variable Bitwidth version is usually much better than the Non-Restoring Divisor.

Implementing the Variable Bitwidth version of the Lookup Divisor algorithm requires an extra lookup table to store the bitwidth of the inverse of each denominator.

Lookup Divisor Version	Slices	18x18 Multipliers
Variable Width Inverse	216	1
Fixed Width Inverse	103	1

Table 6.1: FPGA resource usage of the Variable Bitwidth Lookup Divisor vs. the Fixed Bitwidth Lookup Divisor

Since this table only needs to store the size of the inverse, not the inverse itself, each entry only occupies 5-bits (the number of bits needed to store the maximum size of the inverse which is 17). This version of the Lookup Divisor is shown in Figure 6.6. Unfortunately, adding the Bitwidth table and its associated extra logic noticeably increases the amount of FPGA resources used by the divisor, as seen in Table 6.1.

6.3 Further Optimisations

From Figure 6.5, it can be seen that the Lookup Table Divisor gives a more accurate result to the Non-Restoring Divisor *on average*. However it can also be seen that there are cases in which the Lookup Divisor algorithm performs *worse* than the Non-Restoring Divisor.

From Equation (6.4), it can be seen the result of the Lookup Divisor algorithm is given via the truncation of a larger bitwidth quantity. The accuracy of the result can be improved by changing this truncation into a rounding operation instead. Unlike truncation, the rounding operation requires that more bits be computed than are to be output in the result [49]. This isn't a problem for the Lookup Divisor algorithm, since the bitwidth of the inverse, N, is always greater or equal to one, hence there is always an extra bit on which to base the rounding operation. Furthermore, while rounding on a CPU requires extra computation, it can be performed quickly and simply on a FPGA.

This transforms the result of the Lookup Table Divisor from Equation (6.4) to

$$\text{Inverse}\left[\text{Denominator}\right] = \text{Round}\left(\frac{1}{\text{Denominator}} \times 2^N\right) \tag{6.11a}$$

$$\text{Result} = \text{Round}\left(\frac{\text{Numerator} \times \text{Inverse}[\text{Denominator}]}{2^N}\right) \tag{6.11b}$$

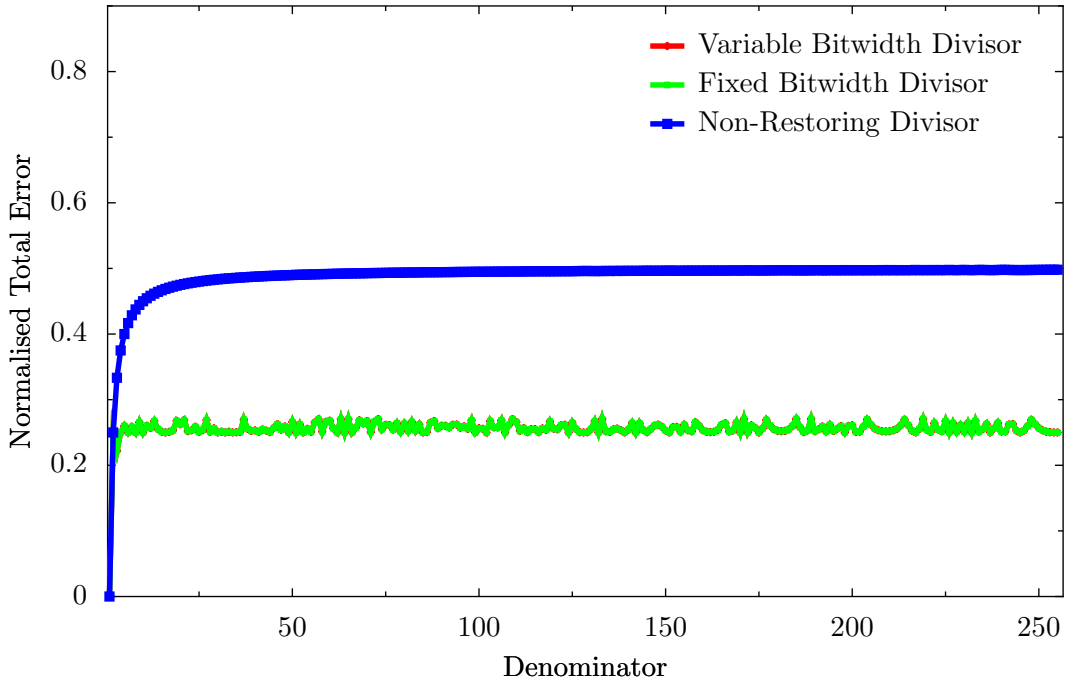


Figure 6.7: Normalised Total Error of the Lookup Divisor (with Rounding) and the Non-Restoring Divisor

$$= \text{Round}\left(\frac{\text{Numerator} \times \text{Round}\left(\frac{1}{\text{Denominator}} \times 2^N\right)}{2^N}\right) \tag{6.11c}$$

While this may seem a small optimisation, the effect on the accuracy of the Lookup Table Divisor is significant, as shown in Figure 6.7. Two facts are readily apparent from the graph. Firstly, the introduction of rounding has greatly improved the accuracy of the Lookup Divisor. In fact, the error of the divisor is now half that of the error of the Non-Restoring Divisor.

This is not entirely surprising. The Non-Restoring Divisor discards the fractional component of the result entirely. By contrast the Rounding Lookup Divisor uses the value of the most significant bit of the fractional component ($2^{-1} = 0.5$) to perform the rounding operation. In essence the Non-Restoring Divisor starts discarding information beginning from the most significant bit of the fraction ($2^{-1} = 0.5$) whereas the Lookup Divisor only starts to discard information beginning form the

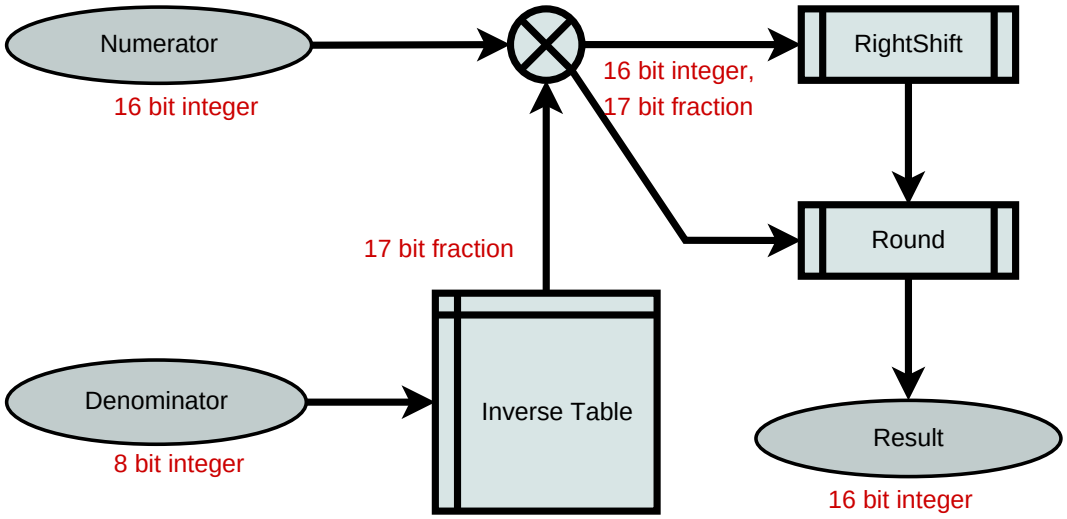


Figure 6.8: Dataflow diagram for the Lookup Table Divisor with Rounding

second most significant bit of the fraction ($2^{-2} = 0.25$). As can be seen from the graph, these are approximately the magnitude of the average error in both divisors.

The second item of note on the graph is rounding the result effectively cancelled any benefit gained by selectively varying the bitwidth of the inverse. This actually a beneficial result. Since the fixed-width divisor performs as well as the variable-width divisor (when the result is rounded), it is possible to avoid the extra complexity required for implementing the variable-width divisor.

The final version of the Lookup Table Divisor is shown in Figure 6.8. By using fixed widths quantities for the inverse table it avoids the need for a separate bitwidth table, and thus saves space on the FPGA. By rounding the output the divisor produces results of greater accuracy than the standard Non-Restoring Divisor.

Figure 6.9 shows the amount of FPGA resources consumed by the final version of the Lookup Table Divisor as compared to the resources consumed by different, openly available, implementations of the Radix-2 Non-Restoring Divisor [48]. All the division units operate on unsigned integers, and divide a 16-bit number by an 8-bit number, since this is what is required by the architecture of the RGB to HSL converter. It can be seen that the resource requirements of the Lookup Divisor compare favourably with the requirements of the Non-Restoring Divisors. The main

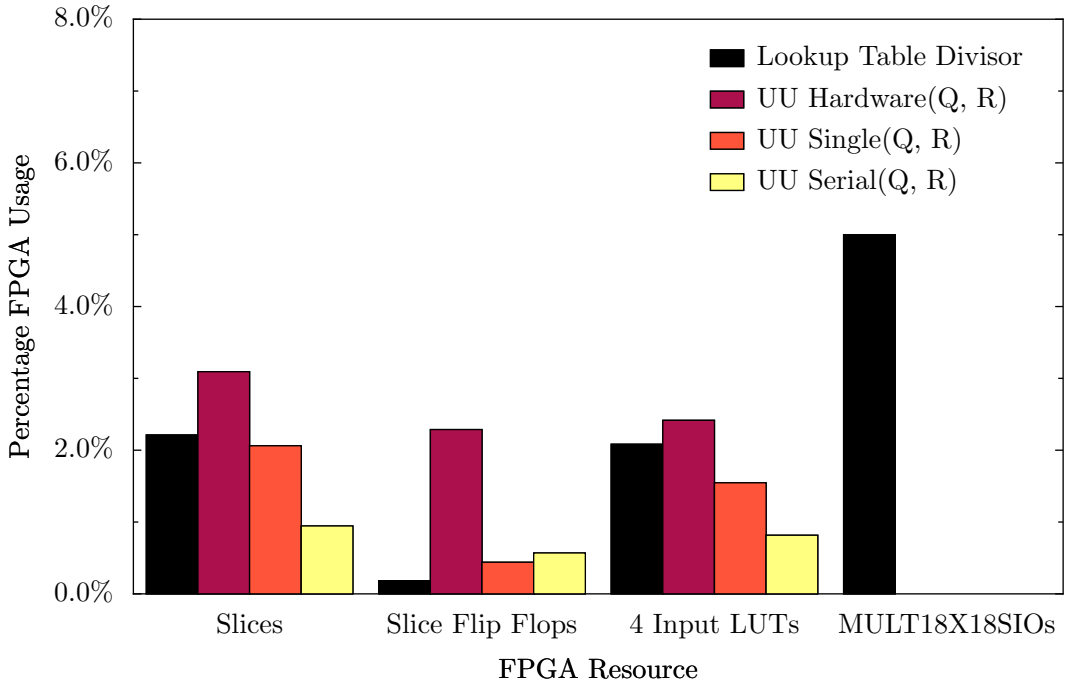


Figure 6.9: FPGA Usage of the Lookup Table Divisor compared with different implementations of the Radix-2 Non-Restoring Divisor

difference being that the Lookup Divisor requires the use of one of the FPGA's hardware multipliers, whereas the Non-Restoring Divisors can be instantiated using regular logic cells.

The Lookup Table Divisor shown in the graph is fully pipelined, returning an output every clock cycle, and only takes 2 clock cycles to process each set of inputs. The UU Hardware divisor is a fully pipelined Radix-2 Non-Restoring Divisor that takes 8 clock cycles to process each individual set of inputs. Both these divisors operate at greater than the 50MHz limit required of the FPGA. By comparison, the UU Single divisor use the Non-Restoring algorithm but calculates the result in only a single clock cycle. However it runs at less than 22MHz, limiting its use. Finally, the UU Serial divisor is much like the UU Hardware divisor in that it takes 8 cycles to fully process its inputs. Unlike the previous divisors however, the UU Serial divisor is not fully pipelined, and cannot accept any inputs until it has finished processing the current result. These results are summarised in Figure 6.10.

It can be seen that the Lookup Table Divisor compares favourably with the Non-

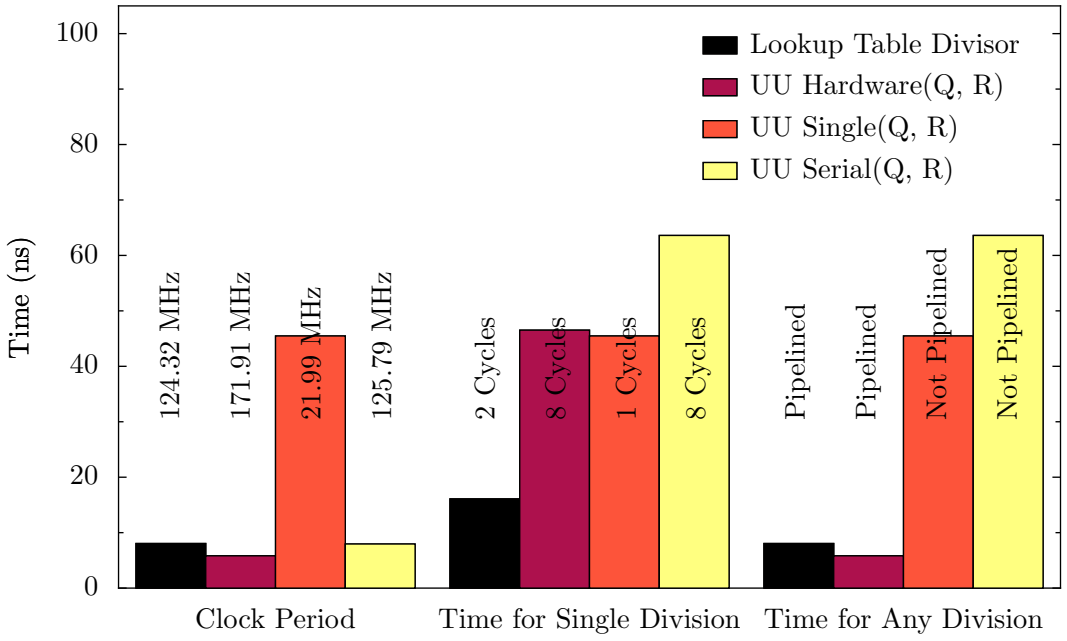


Figure 6.10: Performance of the Lookup Table Divisor compared with different implementations of the Radix-2 Non-Restoring Divisor

Restoring Divisor in its optimised niche of byte-sized division. Not only does the Lookup Divisor return a more accurate fixed-point result, but it also has a comparable FPGA resource utilisation, as well as a shorter pipeline depth and comparable clock speed.

6.4 Error Distribution

The previous section provided calculations for the mean error produced by both the Lookup Divisor and the Non-Restoring Divisor. However, in order to fully characterise the systems, more information is required.

Let the inputs to the division algorithms be two random variables, n and d. Assume that both are discrete random variables uniformly distributed between $n \in [0, 65535]$ and $d \in [1, 255]$, that is n is a 16-bit integer and d is an 8-bit integer. It is not necessary to consider to case of $d = 0$, since division by zero is an undefined operation. Therefore the number of possible combinations of n and d is given by the expression

$$\text{div}_{\text{combs}} = (n_{max} - n_{min} + 1) \times (d_{max} - d_{min} + 1) \tag{6.12a}$$

$$\text{div}_{\text{combs}} = (2^{16} - 1 - 0 + 1) \times (2^{8} - 1 - 1 + 1) \tag{6.12b}$$

$$\text{div}_{\text{combs}} = 2^{24} - 2^{16} \tag{6.12c}$$

Furthermore, let $\text{Error}(n, d)$ be the magnitude of the error resulting from dividing n by d using one of the previously outlined algorithms. Since n and d are uniformly distributed random variables, the probability associated with any particular combination of n and d is $1/\text{div}_{\text{combs}}$. The mean error expected from a division operation is thus given by [50]

$$\bar{e} = \frac{\sum_{n_{min}}^{n_{max}} \sum_{d_{min}}^{d_{max}} \text{Error}(n, d)}{\text{div}_{\text{combs}}} invalid \tag{6.13}$$

Similarly, the standard deviation can be obtained from

$$\sigma_e = \sqrt{\frac{\sum_{n_{min}}^{n_{max}} \sum_{d_{min}}^{d_{max}} (\text{Error}(n, d) - \bar{e})^2}{\text{div}_{\text{combs}}}} \tag{6.14}$$

The output result of the Non-Restoring Divisor is given by Equation (6.1a), hence it has a mean error value of

$$\bar{e}_{nrd} = \frac{\sum_{n_{min}}^{n_{max}} \sum_{d_{min}}^{d_{max}} \left| \left\lfloor \frac{n}{d} \right\rfloor - \frac{n}{d} \right|}{\text{div}_{\text{combs}}} \tag{6.15a}$$

$$\bar{e}_{nrd} = 0.487844 \tag{6.15b}$$

This is the same as the mean error obtained from Figure 6.7, as expected. In addition, the standard deviation of the error of the Non-Restoring Divisor is given by

$$\sigma_{e-nrd} = \sqrt{\frac{\sum_{n_{min}}^{n_{max}} \sum_{d_{min}}^{d_{max}} \left(\left| \left\lfloor \frac{n}{d} \right\rfloor - \frac{n}{d} \right| - \bar{e}_{nrd} \right)^2}{\text{div}_{\text{combs}}}} \tag{6.16a}$$

$$\sigma_{e-nrd} = 0.290281 \tag{6.16b}$$

Which indicates that the error of the Non-Restoring Divisor is widely spaced. While centred around $\bar{e}_{nrd} = 0.487844$, there will still be a high probability of obtaining a much higher error value.

Performing the same analysis on the Rounding, Fixed-Width (17-bits) Lookup Table Divisor produces a mean of

$$\bar{e}_{ltd} = \frac{\sum_{n_{min}}^{n_{max}} \sum_{d_{min}}^{d_{max}} \left| \text{Round}\left(\frac{n \times \text{Round}\left(\frac{1}{d} \times 2^{17}\right)}{2^{17}} \right) - \frac{n}{d} \right|}{\text{div}_{\text{combs}}} \tag{6.17}$$

$$\bar{e}_{ltd} = 0.255862 \tag{6.18}$$

And a standard deviation of

$$\sigma_{e-ltd} = \sqrt{\frac{\sum_{n_{min}}^{n_{max}} \sum_{d_{min}}^{d_{max}} \left(\left| \text{Round}\left(\frac{n \times \text{Round}\left(\frac{1}{d} \times 2^{17}\right)}{2^{17}} \right) - \frac{n}{d} \right| - \bar{e}_{ltd} \right)^2}{\text{div}_{\text{combs}}}} \tag{6.19}$$

$$\sigma_e = 0.157503 \tag{6.20}$$

As can be seen, both the mean and the standard deviation of the Lookup Divisor algorithm are approximately half those of the Non-Restoring Divisor. This is not surprising, since the rounding operator in the Lookup Divisor allows the divisor to capture more information in the result than the truncation operator of the Non-Restoring Divisor.

Unfortunately, these same rounding and truncations operations make it difficult to obtain an analytic expression for the probability density functions of the divisor errors. However it is possible to construct a program to obtain a parametric approximation to the probability error density functions. Since both the numerator and the denominator are assumed to be uniformly distributed random variables, it is sufficient to iterate across every possible combination of numerator and denominator and record the error value. The resulting error values can then be discretized and counted to produce a histogram — a probability mass function of the divisor

error. This is shown plotted in Figure 6.11.

As can be seen, the error values of the Non-Restoring Divisor are fairly uniformly distributed across range, whereas the error of the Lookup Divisor is clumped towards the lower end of the error range. The error distribution for the Non-Restoring Divisor is mostly symmetric as expected — there is as much a chance of a high error as there is of a lower error value. The asymmetry of the Lookup Divisor is likely due to the two rounding operations in the algorithm.

In addition, the probability mass functions show several spikes, notably at $e = 0, 0.25, 0.5$ and 0.75. This is likely a result of having discrete random variables as inputs to the division algorithms instead of continuous random variables.

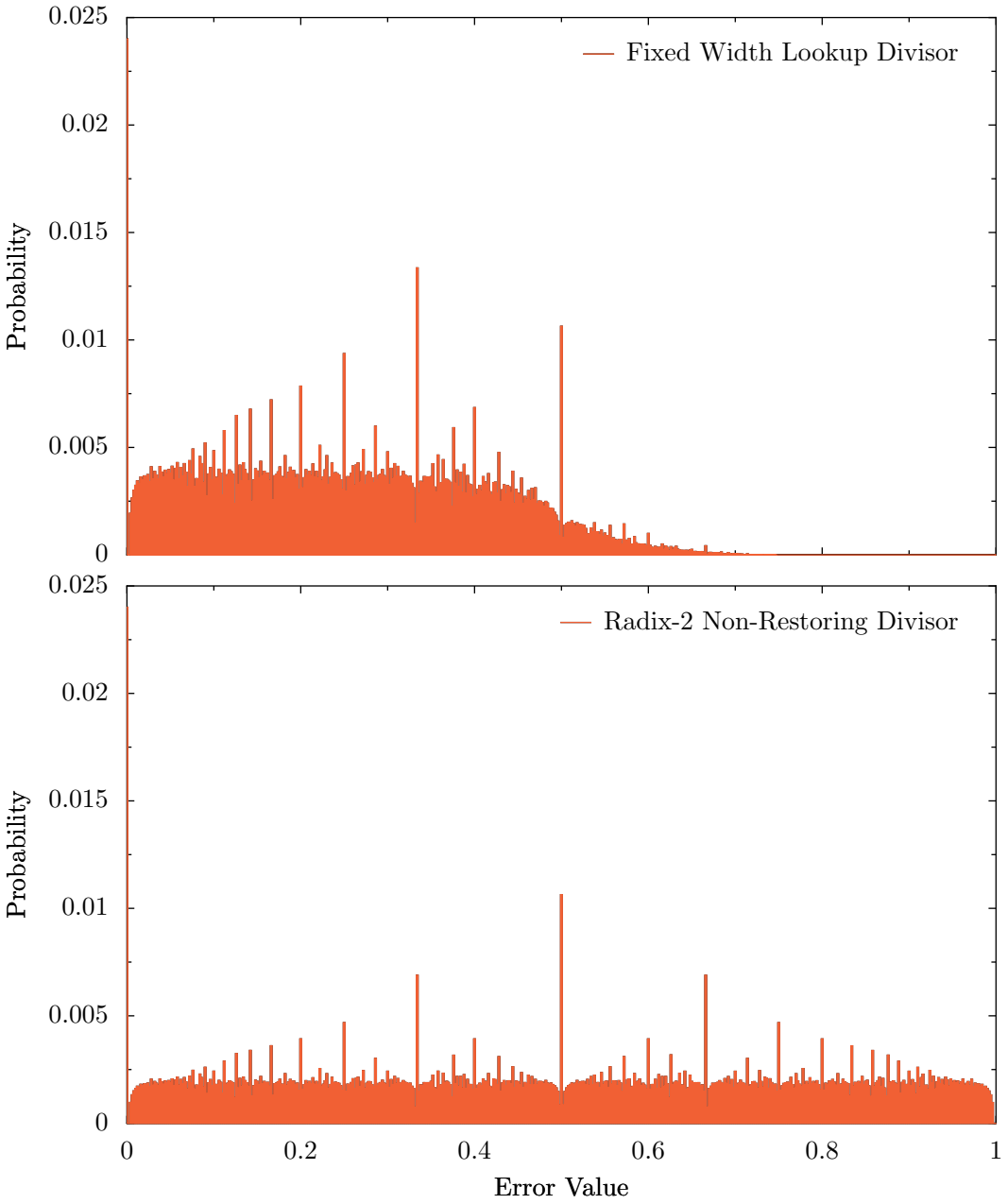


Figure 6.11: Probability distribution of the error values from the divisors

Chapter 7

FPGA Implementations and System Performance

7.1 FPGA Modules

The various image processing algorithms have been implemented as separate VHDL modules, and the inputs of the modules have been designed to be as generous as possible in order to maximise code reuse. When linked together, the modules form a system for performing detection and location of coloured objects using the FPGA.

7.1.1 EyeLink Flow Control Protocol

The EyeLink Flow Control Protocol was designed as a method to throttle the flow rate of data between different modules. In every transaction of signal data, one module will be sending data, and the other module will be receiving data. Between

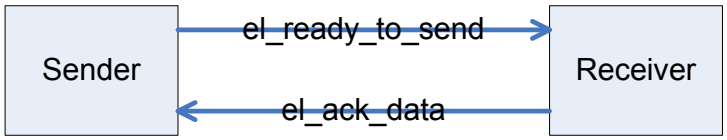


Figure 7.1: Signals of the EyeLink Flow Control Protocol

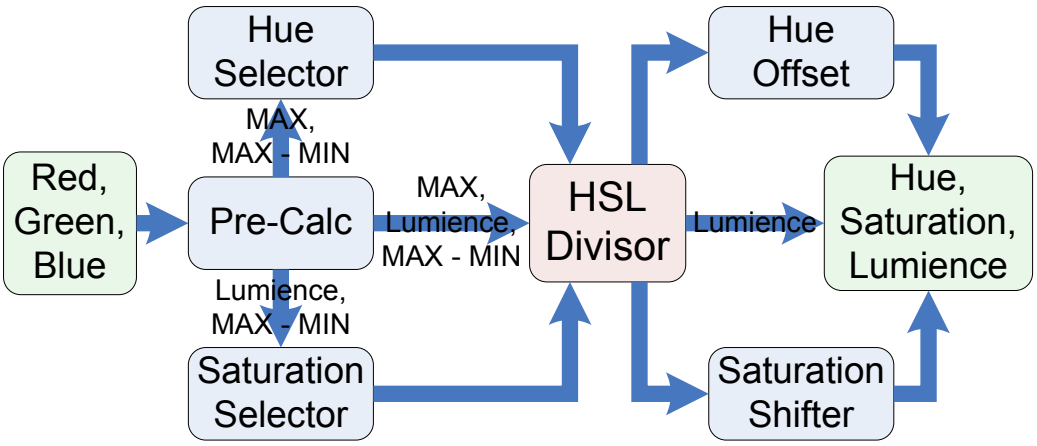


Figure 7.2: Block diagram of the RGB to HSL colour space converter

the sender and the receiver, the protocol defines two signal lines `el_ready_to_send` and `el_ack_data`. This is illustrated in Figure 7.1.

When the sender has valid data to give to the receiver, it asserts `el_ready_to_send` high. When the receiver is ready and able to receive data, it asserts `el_ack_data`. When both signals are high, the sender puts its data onto the signal line, and the receiver reads the same data from the line. This protocol ensures that the sender does not flood the receiver with more data than it can handle. It also ensures that the receiver only processes valid data from the sender. In essence it allows two modules with different data rates to operate correctly when linked together.

7.1.2 RGB to HSL Converter

This module implements the RGB to HSL colour space conversion outlined in Chapter 4, Equations (4.15), (4.16) and (4.17). The module is divided up into several distinct components as shown in the block diagram in Figure 7.2.

The Pre-Calc component reads in the (Red, Green, Blue) channel data and outputs the Lumience, a code corresponding to the maximum RGB colour channel, and the value of the maximum RGB channel minus the minimum RGB channel (delta).

The Saturation Selector component prepares the saturation parameters for the divisor. The numerator of the saturation is the delta value from the Pre-Calc component,

Algorithm 7.1 RGB to HSL: Pre-Calc

Require: Red, Green, Blue $\in [0, 255]$
Ensure: delta, Lumience $\in [0, 255]$
  max_val $\Leftarrow$ max(Red, Green, Blue)
  min_val $\Leftarrow$ min(Red, Green, Blue)
  delta $\Leftarrow$ max_val $-$ min_val
  if max_val = Red **then**
    max_channel $\Leftarrow$ 'R'
  else if max_val = Green **then**
    max_channel $\Leftarrow$ 'G'
  else if max_val = Blue **then**
    max_channel $\Leftarrow$ 'B'
  end if
  Lumience $\Leftarrow$ max_val/2 + min_val/2
  /* division by 2 $\equiv$ right shift by 1, dividing before adding ensures all calculations fit within 8-bits */

bitshifted into the upper 8-bits of a 16-bit number to obtain greater precision from the division. The denominator is the Lumience value or 255 minus the Lumience value, depending on the size of the Lumience.

Algorithm 7.2 RGB to HSL: Saturation Selector

Require: delta, Lumience $\in [0, 255]$
Ensure: saturation_numerator$[7:0] = 0$, saturation_denominator $\in [0, 255]$
  saturation_numerator $\Leftarrow$ delta shift left by 8
  if Lumience ≤ 127 **then**
    saturation_denominator $\Leftarrow$ Lumience
  else
    saturation_denominator $\Leftarrow 255 -$ Lumience
  end if

Similarly, the Hue Selector prepares the hue parameters for the divisor. The parameters vary greatly depending on which of the original RGB channels was brightest. Additional logic is added in order to keep all values as unsigned integers instead of switching back and forth between signed and unsigned quantities.

The next component is the HSL Divisor, which calculates the division portion of the RGB to HSL conversion algorithm for both the Hue and Saturation channels (hue_quotient and sat_quotient, both Q8.8 fixed point numbers). The divisor is a version of the Lookup Table Divisor module modified to carry additional state

Algorithm 7.3 RGB to HSL: Hue Selector

Require: delta $\in [0, 255]$
Ensure: hue_numerator$[7:0] = 0$, hue_denominator $\in [0, 255]$
  hue_denominator $\Leftarrow$ delta
  if max_channel = 'R' and Green $>$ Blue **then**
    hue_numerator $\Leftarrow$ (Green $-$ Blue) shift left by 8
    hue_subtract $\Leftarrow$ false
  else if max_channel = 'R' and Green $\leq$ Blue **then**
    hue_numerator $\Leftarrow$ (Blue $-$ Green) shift left by 8
    hue_subtract $\Leftarrow$ true
  else if max_channel = 'G' and Blue $>$ Red **then**
    hue_numerator $\Leftarrow$ (Blue $-$ Red) shift left by 8
    hue_subtract $\Leftarrow$ false
  else if max_channel = 'G' and Blue $\leq$ Red **then**
    hue_numerator $\Leftarrow$ (Red $-$ Blue) shift left by 8
    hue_subtract $\Leftarrow$ true
  else if max_channel = 'B' and Red $>$ Green **then**
    hue_numerator $\Leftarrow$ (Red $-$ Green) shift left by 8
    hue_subtract $\Leftarrow$ false
  else if max_channel = 'B' and Red $\leq$ Green **then**
    hue_numerator $\Leftarrow$ (Green $-$ Red) shift left by 8
    hue_subtract $\Leftarrow$ true
  end if

information through the divisor's pipeline.

Finally the Hue Offset and Saturation Shifter components takes the output from the divisor, discards the upper and lower bits, and rounds the result to produce the final HSL image data.

Algorithm 7.4 RGB to HSL: Saturation Shifer

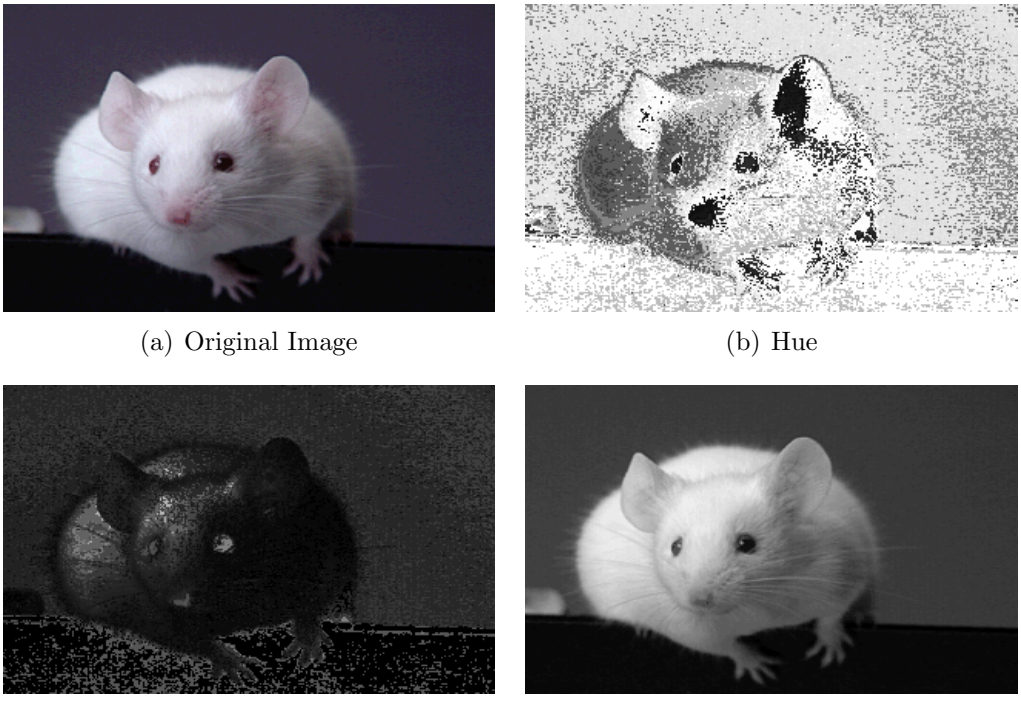
Require: sat_quotient in Q8.8 fixed point format
Ensure: saturation $\in [0, 255]$
  if delta $= 0$ **then**
    saturation $\Leftarrow 0$ /* Grayscale pixel */
  else
    sat $\Leftarrow$ sat_quotient$[8:1]$ + sat_quotient$[0]$
  end if
  /* The results of the saturation division are always fractional. Taking sat_quotient$[8:1]$ instead of sat_quotient$[7:0]$ is the same as shifting right by 1, which performs the division by 2 necessary for the saturation channel. Adding the last bit rounds the final result instead of truncating. */

This module is similar to that constructed by previous researchers for performing colour object detection using FPGAs. The implementation proposed by García-Campos [21] downsampled the three 8-bit RGB channels into three 5-bit channels, which were then used as as an index into a table of Hue, Saturation and Intensity values. Their implementation was simpler than the one constructed for this project, but also less accurate, since the resolution of the input RGB data was reduced from 256 levels per channel to 32 levels per channel.

Figure 7.3 shows the results of running an image through the hardware RGB to HSL converter on the FPGA. The decomposed images are very similar to the reference decomposition in Figure 4.4. The saturation and lumience channels are almost identical to the reference decomposition, but the hue channel shows some visible differences. The FPGA implementation manages to correctly identify the red portions of the image (the eyes, nose, hands and ears of the mouse), but differs noticeably from the reference implementation on the white fur of the mouse. Part of this difference is due to the fact that the Hue channel is cyclic. The Hue channel 'wraps around,' so that low Hue numbers (dark areas on the image) are actually 'close' to large Hue numbers (bright areas on the image).

Algorithm 7.5 RGB to HSL: Hue Offset

Require: hue_quotient in Q8.8 fixed point format
Ensure: hue $\in [0, 255]$
  if delta = 0 **then**
    hue $\Leftarrow$ 255 /* Grayscale pixel */
  else
    hue $\Leftarrow$ 42$\times$ hue_quotient
    /* 42 is Q6.0 format, and Q8.8 $\times$ Q6.0 = Q14.8 */
    if max_channel = 'R' **then**
      hue_offset $\Leftarrow$ 42
    else if max_channel = 'G' **then**
      hue_offset $\Leftarrow$ 126
    else if max_channel = 'B' **then**
      hue_offset $\Leftarrow$ 210
    end if
    if hue_subtract = true **then**
      hue $\Leftarrow$ hue_offset $-$ (hue[15 : 8] + hue[7])
    else
      hue $\Leftarrow$ hue_offset + hue[15 : 8] + hue[7]
    end if
    /* discard the fractional bits and the upper (empty) integer bits, and round the result for greater accuracy */
  end if



(a) Original Image (b) Hue

(c) Saturation (d) Lumience

Figure 7.3: HSL Channel Decomposition from the FPGA module

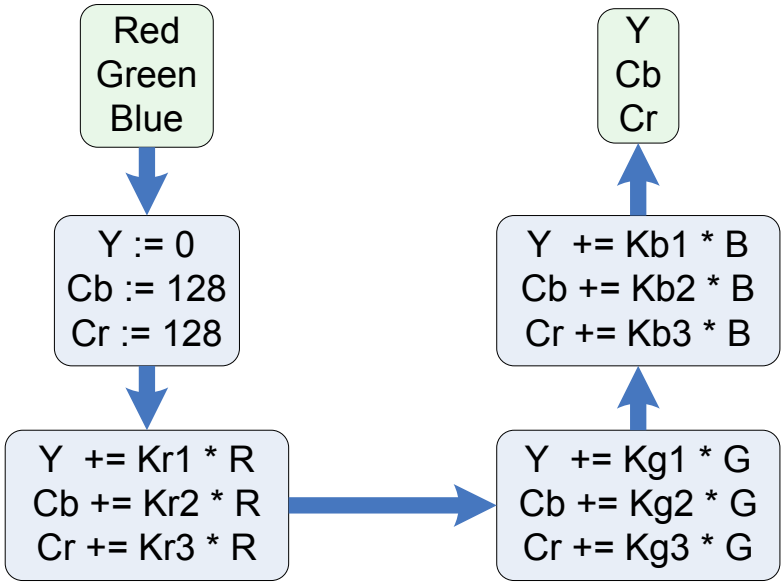


Figure 7.4: Block diagram of the RGB to YCbCr colour space converter

7.1.3 RGB to YCbCr Converter

The RGB to YCbCr colour converter operates as a three stage pipeline implementing the matrix multiplication required to perform the colour space conversion shown in Chapter 4, Equation (4.21). The incoming RGB data is loaded into registers in the first stage. In subsequent states the registered data is then processed using multiply-accumulate blocks to produce the YCbCr data, which is output in the final stage. Figure 7.4 shows the block diagram for this module.

Since the coefficients of the RGB to YCbCr matrix are all non-integer quantities, the FPGA implementation represents them using fixed point arithmetic in a similar fashion to the Lookup Table Divisor. Figure 7.5 shows the results of running an image through the hardware RGB to HSL converter on the FPGA.

7.1.4 Colour Thresholder

The colour thresholder module is configured by setting the limits of the three image channels. Subsequently, three-channel pixel data is fed into the unit, which outputs a '1' if the pixel fits within the set limits, or a '0' otherwise. Asserting the `reset`

Algorithm 7.6 RGB to YCbCr Converter

Require: Red, Green, Blue $\in [0, 255]$
Ensure: Y, Cb, Cr $\in [0, 255]$

$Y \Leftarrow 0$
$Cb, Cr \Leftarrow 16777216$ /* 128 in Q8.17 */

$Y \Leftarrow Y + 39191 \times Red$ /* 0.299 in Q8.17 */
$Cb \Leftarrow Cb - 22117 \times Red$ /* −0.168736 in Q8.17 */
$Cr \Leftarrow Cr + 65531 \times Red$ /* 0.5 in Q8.17 */

$Y \Leftarrow Y + 76939 \times Green$ /* 0.587 in Q8.17 */
$Cb \Leftarrow Cb - 43419 \times Green$ /* −0.331264 in Q8.17 */
$Cr \Leftarrow Cr - 54878 \times Green$ /* −0.148688 in Q8.17 */

$Y \Leftarrow Y + 14942 \times Blue$ /* 0.114 in Q8.17 */
$Cb \Leftarrow Cb + 65536 \times Blue$ /* 0.5 in Q8.17 */
$Cr \Leftarrow Cr - 10658 \times Blue$ /* −0.081312 in Q8.17 */

/* Right shift and round to obtain the output channel values in Q8.0 format. */
$Y \Leftarrow Y[24:17] + Y[16]$
$Cb \Leftarrow Cb[24:17] + Cb[16]$
$Cr \Leftarrow Cr[24:17] + Cr[16]$

(a) Original Image

(b) Y (Luma)

(c) Cb (Chroma-Blue)

(d) Cr (Chroma-Red)

Figure 7.5: YCbCr Channel Decomposition from the FPGA module

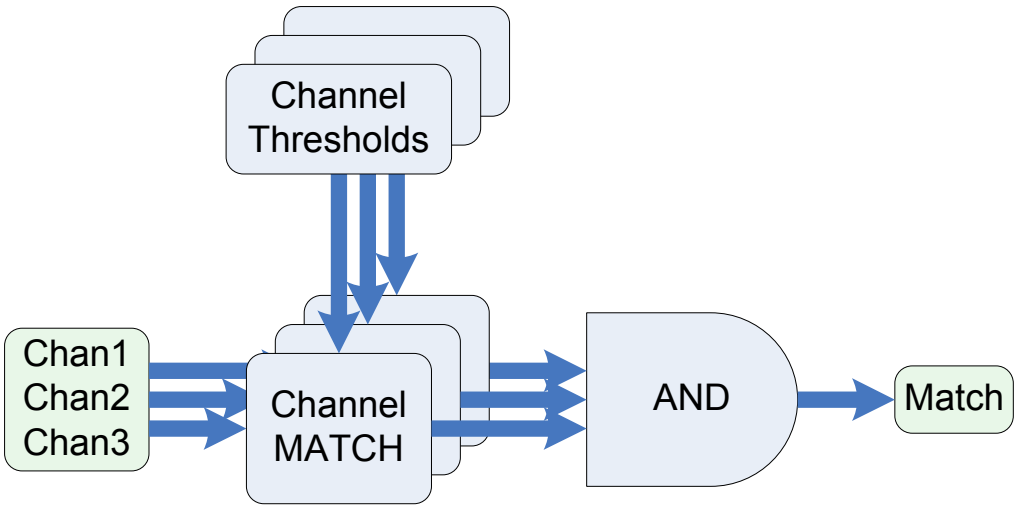


Figure 7.6: Block diagram of the Colour Thresholder module

signal resets the internal state of the module to the default state. In the default state, the module will match on every pixel (the limits for all the channels are set to $[0, 255]$). A block diagram of the module is shown in Figure 7.6.

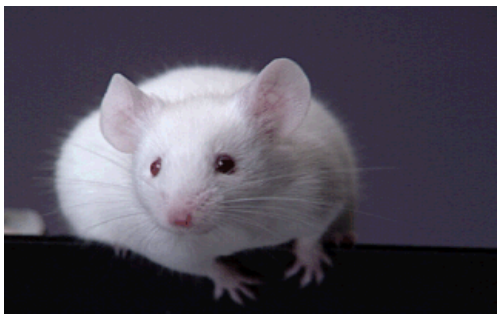
Algorithm 7.7 Colour Thresholder

Require: Chan_1, Chan_2, Chan_3, Max_Val, Min_Val $\in [0, 255]$, Set_Chan $\in [0, 3]$, Reset $\in [0, 1]$
Ensure: Match $\in$ [false, true]
  if Reset $= 1$ **then**
    /* Resets the internal state of the module to default values */
    $\forall x \in [1, 3]$, $\min_x \Leftarrow 0$, $\max_x \Leftarrow 255$
  else if Set_Chan $\neq 0$ **then**
    $\min_{\text{Set\_Chan}} \Leftarrow$ Min_Val
    $\max_{\text{Set\_Chan}} \Leftarrow$ Max_Val
  else if Set_Chan $= 0$ **then**
    if $\forall x \in [1, 3]$, $\min_x \leq \text{Chan}_x \leq \max_x$ **then**
      Match $\Leftarrow 1$
    else
      Match $\Leftarrow 0$
    end if
  end if

Figure 7.7 shows the results of feeding the original image data through the colour conversion modules. The converted images are then piped through the colour threshold module to produce the final result. The threshold values were chosen to select

(a) Original image — Lenna

(b) HSL thresholded: $40 \leq Hue \leq 50$



(c) Original image — Mouse

(d) YCbCr thresholded: $77 \leq Cb \leq 127 \wedge 133 \leq Cr \leq 173$

Figure 7.7: Output from the FPGA colour conversion and threshold modules

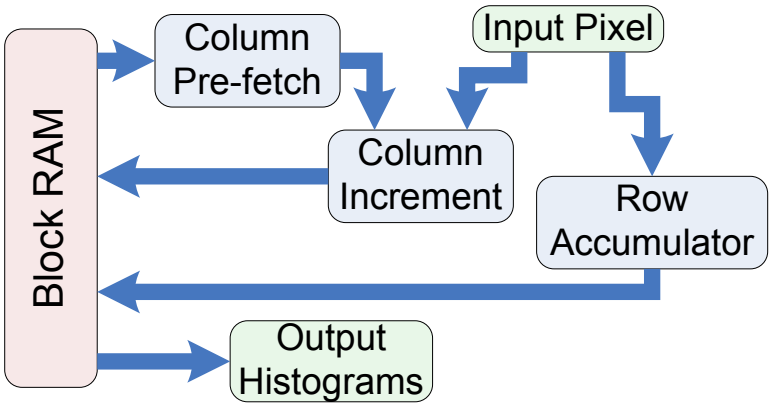


Figure 7.8: Block diagram of the Mass Counter module

the portions of both images with a strong red content. This works well on the mouse image, which is mostly white and blue. It is relatively easy to extract the red eyes and nose of the mouse from the rest of the image. By contrast, the background of the Lenna image has very similar colour tones to the foreground, making it difficult to distinguish between the two using colour thresholding alone.

7.1.5 Object Locator

The object locator module takes in a stream of bits representing an image mask, where a '1' designates pixels belonging to the object of interest, and a '0' designates other pixels. From this bitstream, row and column counts are made of the pixels and the result stored in one of the FPGA's block RAMs, where they may be streamed out to the CPU, or to another module for further processing. The module is able to process an input pixel every clock cycle, but requires an additional clock cycle at the end of every row in order to write out the row count to RAM. The block diagram of this module is shown in Figure 7.8. Unlike the colour converters, the data output from this module is in 10-bit chunks, not 8-bit. 10-bits is necessary in order to store the row or column count from a 512×512 or 640×480 resolution image.

The complexity of this algorithm stems from the need to synchronise accesses to the module's internal block RAM. The block RAM is dual-ported, meaning it can be both read from and written to at the same time. However only a single read and

Module	Slices	18x18 Multipliers	BlockRAMs
Available	**4656**	**20**	**20**
RGB to HSL Converter	170	3	2
RGB to YCbCr Converter	121	8	0
Colour Thresholder	54	0	0
Object Locator	134	0	1

Table 7.1: Resource usage of the implemented modules on a Xilinx Spartan3-500E FPGA

a single write can be performed each clock cycle. Moreover, while a write can be completed in a single clock cycle, reads take two cycles. On the first clock edge, the read address is given to the RAM, however the read data will not be available until the next clock edge. This means that data needs to be requested from the RAM a clock cycle before it is needed.

This module differs from the others in that the results are not directly streamed out as they are processed. Instead an entire image frame needs to be processed, and the results accumulated in the module's block RAM before being usable. In addition, this module does not directly calculate the centroid of the object, only the row and column histograms from Equation (4.4). Further calculations, including finding the centre of mass and the axis of minimum inertia, are left for the CPU.

7.2 System Performance

Table 7.1 shows the amount of FPGA resources required to implement each of the VHDL modules. As can be seen, the primary resource used by the modules is not logic slices, but instead block RAMs and multipliers.

All of the implemented FPGA modules are fully pipelined, returning an output every clock cycle. On the 50Mhz FPGA, this means they can each process approximately 190 medium resolution (512×512) frames per second. In addition, the parallel nature of hardware logic blocks means that the FPGA can perform all these operations more or less simultaneously. By contrast, a standard CPU needs to perform these calculations sequentially, along with any other tasks the processor needs to complete.

Algorithm 7.8 Object Locator

```
Require: Pixel_In ∈ [0, 1]
  if row_cool_down = true then
    /* The end of the current row has been reached. Writing out the row data to
    internal storage takes priority over all other tasks. */
    write_address ⇐ number_of_columns + current_row
    write_data ⇐ row_accumulator
    read_address ⇐ 0 /* Prepare for the 1st column */
    current_row ⇐ current_row + 1
    row_cool_down ⇐ false
    row_accumulator ⇐ 0
  else if Reset = 1 then
    /* Resets the internal state of the module to default values */
    current_row, current_column ⇐ 0
    number_of_columns ⇐ 352 /* CIF resolution — camera resolution */
    row_accumulator ⇐ 0
    row_cool_down ⇐ false
  else if Load_Enable = 1 then
    /* Count the value of the current pixel. */
    row_accumulator ⇐ row_accumulator + Pixel_In
    if current_row = 0 then
      /* First row, no previous column data to increment */
      column_count ⇐ Pixel_In
    else
      /* Otherwise increment the column data prefetched from RAM */
      column_count ⇐ read_data + Pixel_In
    end if
    /* Write out the current column data to RAM */
    write_address ⇐ current_column
    write_data ⇐ column_count
    if current_column = number_of_columns − 1 then
      /* Reached the end of the current row */
      current_column ⇐ 0
      row_cool_down ⇐ true
    else
      current_column ⇐ current_column + 1
      /* Prefetch the next column's data */
      read_address ⇐ current_column + 1
    end if
  end if
  /* Write/Read data to/from the block RAM on the clock. */
  loop /* On the Clock Edge */
    block_ram[write_address] ⇐ write_data
    /* read_data will be available for the next clock cycle */
    read_data ⇐ block_ram[read_data]
  end loop
```

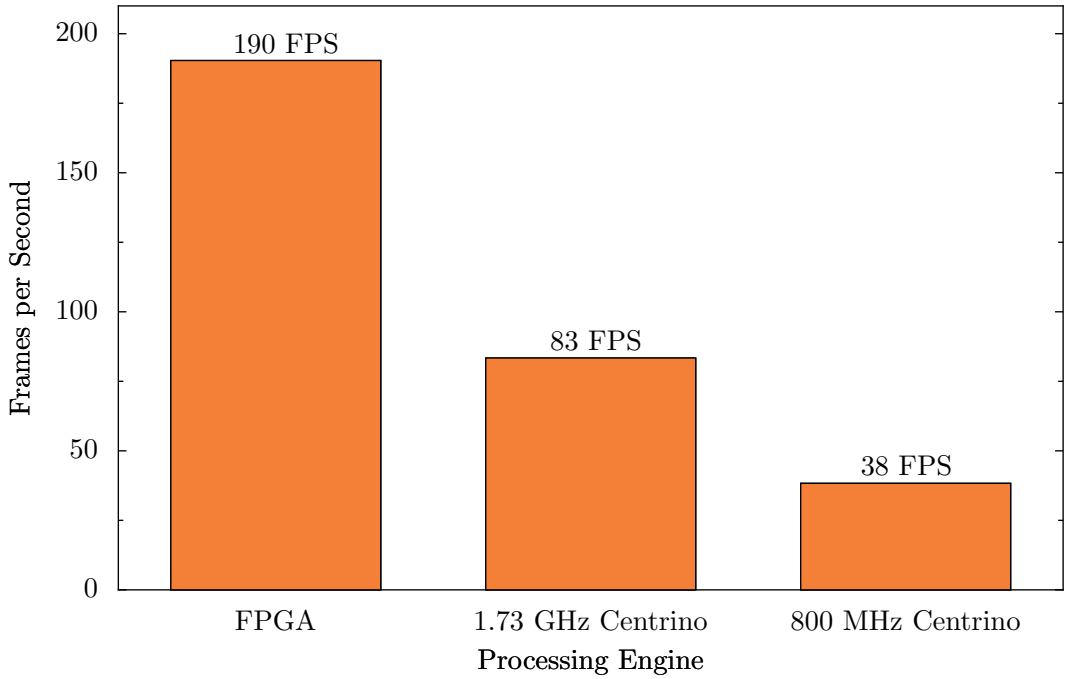


Figure 7.9: Frame rate of the FPGA vs a Laptop CPU [512×512 Resolution Images]

The effects of this are shown in Figure 7.9. Despite possessing a significant advantage in clock speed, the CPU still fails to match the frame rate of the FPGA.

The numbers in Figure 7.9 correspond to the maximum frame rate attainable from the FPGA modules. The actual frame rate will be constrained by the transfer rate of data into the FPGA (likely from the cameras) and out of the FPGA (to the CPU or memory). The frame rate of the CPU will similarly be constrained by I/O.

On the testing framework used in this project, the image processing modules are connected to the system bus through I/O buffers. The CPU writes data to the FPGA, with the write address determining which module receives the data. The processed data is then read back from the FPGA by the CPU, again with the read address determining which module the data is read from.

Using MMIO (Memory Mapped I/O) and 512×512 resolution images, the colour space converter modules manage a frame rate of about 2 frames a second, the colour thresholder module 3 frames a second, and the object locator 15 frames a second. These results do not seem to match the theoretical performance of the FPGA, given that the modules are designed to run at the same speed, and given the maximum

frame rate calculated in Figure 7.9. These inconsistencies can be explained by examining the data rate of the system bus.

Under the test configuration, the CPU has to both write and read data to and from the FPGA. This means the performance of the FPGA modules is twice constrained by the system bus. The bus runs at a maximum speed of 22.4MHz, and using MMIO, the CPU-to-FPGA data rate has been found to be about 3.6MB/s [1].

A full colour medium resolution image occupies 512×512 pixels, times 3 bytes per pixel (3 colour channels), which is equal to 768kB per frame. On the test system, each frame processed by the colour space converters thus requires the CPU to write 768kB to the FPGA and then read an additional 768kB back. Both transfers require a significant amount of time on the system bus, leading to the low frame rates. The output data from the colour thresholder module has only a single bit for each input pixel (so $512 \times 512/8 = 32$kB per frame) hence the data transfer from the FPGA to the CPU takes much less time, leading to a higher frame rate. The object locator module has even less I/O data — only a single bit per input pixel, and only outputs 512 rows plus 512 columns times 10 bits per row and column counter, which equals 320kB per processed frame. Hence this module achieves the highest frame rate using the testing framework.

This situation can be improved by using DMA (Direct Memory Access) for transferring data between the CPU and the FPGA. MMIO involves transferring a single 16-bit word at a time across the system bus. While easy to program, this method of I/O does not take full advantage of the bandwidth available on the system. By contrast, DMA requires making a request to the operating system to transfer a large block of data at once, which while more involved, is more efficient [1].

Using MMIO, it takes 55.546 milliseconds to write a single 512×512 bitmap to the object locator module, and 0.85ms to read back the resulting histogram data. This corresponds to a frame rate of approximately 17.7 frames per second. Using DMA for writes instead of MMIO gives a write time of 44.039ms, corresponding to a frame rate of about 22.28 frames per second.

Despite possessing an 8-fold advantage in clock speed, the ARM9 CPU on the Eye-

Bot M6 only managed a processing time of 38.17 milliseconds per frame ($\approx$ 26.2 frames per second). The faster frame rate of the CPU is likely due to processor's data cache, which is large enough to store an entire input frame to the object locator algorithm. This allows the CPU to minimise accesses to the system bus, improving its performance.

If the input data can be taken directly from the camera(s) instead of the CPU, then the CPU-to-FPGA write time becomes irrelevant. In this case the frame rate of the algorithms would depend only on the camera-to-FPGA transfer rate, and the FPGA-to-CPU transfer rate. Since the cameras are connected to the FPGA through a different bus to the CPU, it is anticipated that the frame rate will be much higher with this configuration. Unfortunately the VHDL code necessary to communicate with the cameras was in an unstable state at the time of writing. Without the cameras, the data for the image processing modules needed to be transfered from the CPU, leading to long write times and slow frame rates.

Chapter 8

Conclusion

In conclusion, this project has constructed a new hardware platform, the EyeBot M6, capable of replacing the existing EyeBot controllers. This platform features a modern CPU, a FPGA, stereo cameras, and support for a wide variety of high speed I/O devices. In addition, this project has constructed VHDL code for several image processing algorithms for execution on the FPGA. These include modules for performing colour space conversion, image thresholding, and object location. All of these modules have been tested both in simulation and on the FPGA itself.

Unfortunately the VHDL code needed to retrieve data from the cameras was not in a usable state at the time of writing, thus the image processing modules needed to be tested using data streamed from the CPU. This led to low frame rates due to the relatively slow transfer speeds of the system bus and the large quantities of image data that needed to be transfered back and forth between the CPU and FPGA. Once the cameras are fully operational, it is anticipated that the frame rates of the image processing units will increase significantly.

In addition, this project investigated an architecture for performing fast, accurate, 8-bit division in hardware. This Lookup Table Divisor has been integrated into the RGB to HSL colour space converter. An analysis of the error distribution of this algorithm has been performed, and the Lookup Divisor has been shown to be more accurate than the standard Non-Restoring Divisor.

8.1 Future Work

The code to interface the image processing modules to the CPU is currently quite temperamental. Different versions of this code exist to connect different modules to the system bus, however no code exists for connecting all the modules to the bus at the same time. More debugging and cleanup of this code is necessary.

In addition, the RGB to HSL conversion module currently seems to be suboptimised. In order to achieve a 50MHz clock speed, the Xilinx synthesiser software requires the 'Register Balancing' option to be enabled. By introducing more pipeline stages, it should be possible to reach a 50MHz clock speed without requiring this option.

The RGB to YCBCR converter module is fully pipelined and produces an output every clock cycle, as well as being able to accept an input every clock cycle. While this is good from a throughput and performance perspective, it does mean the module consumes 8 out of the 20 available hardware multipliers. For future work it would be desirable to create a version that only works on a single input at a time (a blocking pipeline). This should allow the converter to operate using only 3 (or even less) multipliers, at the expense of throughput — a reasonable trade-off in some circumstances.

While hardware logic may be faster, many algorithms are more simply expressed in terms of CPU code. However, there exist several small processor cores which have been designed for embedding in FPGAs. One example of this is the Xilinx PicoBlaze 8-bit micro-controller which has been specifically designed for Xilinx Spartan and Virtex FPGAs [18]. Including one of these cores into the FPGA would increase the flexibility of the FPGA, and could help ease the implementation of certain algorithms.

Finally, the camera code needs to be polished and connected to the rest of the image processing modules in order for the system to be able to perform real-time image processing.

References

[1] B. Blackham, "Development of a Hardware Platform For Real-Time Image Processing, The." Final Year Project Thesis, October 2006. School of Electrical, Electronic and Computer Engineering, The University of Western Australia.

[2] D. English, "FPGA Based Embedded Stereo Vision Processing Platform." Final Year Project Thesis, October 2006. School of Electrical, Electronic and Computer Engineering, The University of Western Australia.

[3] Z. Guo, W. Najjar, F. Vahid, and K. Vissers, "A Quantitative Analysis of the Speedup Factors of FPGAs Over Processors," in *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th International Symposium On Field Programmable Gate Arrays*, (New York, NY, USA), pp. 162–170, ACM Press, 2004.

[4] T. Bräunl, *Embedded Robotics: Mobile Robot Design and Applications With Embedded Systems*. New York: Springer-Verlag Berlin Heidelberg, 2003.

[5] D. M. Gavrila and V. Philomin, "Real-Time Object Detection For "smart" Vehicles," in *Proceedings of the Seventh IEEE International Conference On Computer Vision, 1999., The*, vol. 1, pp. 87–93, September 1999.

[6] J. Rose, A. El Gamal, and A. Sangiovanni Vincentelli, "Architecture of Field Programmable Gate Arrays," in *Proceedings of the IEEE*, vol. 81, pp. 1013–1029, July 1993.

[7] J. Villarreal, D. Suresh, G. Stitt, and W. Najjar, "Improving Software Performance With Configurable Logic," in *Design Automation For Embedded Systems*, vol. 7, pp. 325–339, Springer Netherlands, November 2002.

[8] IEEE Computer Society, 3 Park Avenue, New York, NY, USA, *1076™IEEE Standard VHDL Language Reference Manual*, IEEE Std 1076™-2002 ed., May 2002.

[9] IEEE Computer Society, 3 Park Avenue, New York, NY, USA, *IEEE Standard Verilog®Hardware Description Language*, IEEE Std 1364-2001 ed., September 2001.

[10] D. J. Smith, "VHDL and Verilog Compared and Contrasted-Plus Modeled Example Written in VHDL, Verilog and C," in *Design Automation Conference Proceedings 1996, 33rd*, pp. 771–776, June 1996.

[11] Cambridge-MIT, "MDP Micro-Controller Board." `http://www-mdp.eng.cam.ac.uk/about/mdp_balloon_spec.pdf`, August 2006. [Online].

[12] Carnegie Mellon Robotics Institute, "Qwerk Robot Controller." `http://www.terk.ri.cmu.edu/downloads/qwerk.pdf`, August 2006. [Online].

[13] A. Rowe, C. Rosenberg, and I. Nourbakhsh, "A Low Cost Embedded Color Vision System," in *IEEE/RSJ International Conference On Intelligent Robotics and Systems*, 2002.

[14] A. Rowe, C. Rosenberg, and I. Nourbakhsh, "A Second Generation Low Cost Embedded Color Vision System," in *IEEE Conference On Computer Vision and Pattern Recognition*, 2005.

[15] P. McCurry, F. Morgan, and L. Kilmartin, "Xilinx FPGA Implementation of an Image Classifier For Object Detection Applications," in *Image Processing, 2001. Proceedings. 2001 International Conference On*, vol. 3, pp. 346–349, 7-10 October 2001.

[16] P. Zemcik, "Hardware Acceleration of Graphics and Imaging Algorithms Using FPGAs," in *SCCG '02: Proceedings of the 18th Spring Conference On Computer Graphics*, (New York, NY, USA), pp. 25–32, ACM Press, 2002.

[17] M. Borgatti, L. Calì, G. De Sandre, B. Forêt, D. Iezzi, F. Lertora, G. Muzzi, M. Pasotti, M. Poles, and P. L. Rolandi, "A Reconfigurable Signal Processing IC With Embedded FPGA and Multi-Port Flash Memory," in *DAC '03: Proceedings of the 40th Conference On Design Automation*, (New York, NY, USA), pp. 691–695, ACM Press, June 2003.

[18] "Xilinx: The Programmable Logic Company." `http://www.xilinx.com`, April 2006. [Online].

[19] M. Krips, T. Lammert, and A. Kummert, "FPGA Implementation of a Neural Network For a Real-Time Hand Tracking System," in *Proceedings of the First*

IEEE International Workshop On Electronic Design, Test and Applications, pp. 313–317, January 2002.

[20] C. Torres-Huitzil and B. Girau, "FPGA Implementation of an Excitatory and Inhibitory Connectionist Model For Motion Perception," in *Proceedings of the 2005 IEEE International Conference On Field-Programmable Technology*, pp. 259–266, December 2005.

[21] R. García-Campos, J. Batlle, and R. Bischoff, "Architecture of an Object-Based Tracking System Using Colour Segmentation," in *IWISP'96: 3rd International Workshop On Image and Signal Processing* (B. G. Mertzios and P. Liatsis, eds.), (Manchester, UK), pp. 299–302, November 1996.

[22] "Gumstix — All Things Small." `http://www.gumstix.com`, April 2006. [Online].

[23] T. P. Pridmore and W. M. M. Hales, "Understanding Images: an Approach to the University Teaching of Computer Vision," in *Engineering Science and Education Journal*, vol. 4, pp. 161–166, August 1995.

[24] D. Benitez, "Performance of Remote FPGA-Based Coprocessors For Image-Processing Applications," in *Proceedings. Euromicro Symposium On Digital System Design, 2002.*, pp. 268–275, 2002.

[25] L. D. F. Costa and R. M. Cesar, Jr., *Shape Analysis and Classification: Theory and Practice*. Image Processing Series, 2000 N.W. Corporate Blvd., Boca Raton, Florida, U.S.A.: CRC Press LLC, 2001.

[26] R. Owens, "Discrete Binary Images." `http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/OWENS/LECT2/node3.html`, October 1997. [Online].

[27] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*. Systems Programming Series, Addison-Wesley, C edition, 2nd ed., April 2005.

[28] D. Forsyth and J. Ponce, *Computer Vision: A Modern Approach*. Upper Saddle River, N.J., U.S.A.: Prentice Hall, 2003.

[29] OmniVision, *Advanced Information: OV6630 Single-Chip CMOS CIF Color Digital Camera*, 4 March 2000.

[30] R.-L. Hsu, M. Abdel Mottaleb, and A. K. Jain, "Face Detection in Color Images," in *Pattern Analysis and Machine Intelligence, IEEE Transactions On*, vol. 24, pp. 696–706, May 2002.

[31] V. Vezhnevets, V. Sazonov, and A. Andreeva, "A Survey On Pixel Based Skin Colour Detection Techniques," in *Proceedings of Graphicon*, pp. 85–92, 2003.

[32] B. D. Zarit, B. J. Super, and F. K. H. Quek, "Comparison of Five Color Models in Skin Pixel Classification," in *Proceedings. International Workshop On Recognition, Analysis, and Tracking of Faces and Gestures in Real-Time Systems, 1999*, pp. 58–63, 1999.

[33] Wikipedia, "HSV Color Cone." `http://en.wikipedia.org/wiki/Image:HSV_cone.png`, April 2006. [Online].

[34] Wikipedia, "HSL Color Cones." `http://en.wikipedia.org/wiki/Image:Color_cones.png`, April 2006. [Online].

[35] D. S. Taubman and M. W. Marcellin, *JPEG2000 Image Compression Fundamentals, Standards and Practice*. Kluwer International Series in Engineering and Computer Science, The, Kluwer Academic Publishers, 2002.

[36] Joint Photographic Experts Group, "JPEG File Interchange Format." `http://www.jpeg.org/public/jfif.pdf`, September 1992. v1.02, [Online].

[37] D. Chai and A. Bouzerdoum, "A Bayesian Approach to Skin Color Classification in YCbCr Color Space," in *Tencon 2000*, vol. 2, pp. 421–424, 2000.

[38] I. S. Uzun, A. Amira, and A. Bouridane, "FPGA Implementations of Fast Fourier Transforms For Real-Time Signal and Image Processing," in *Vision, Image and Signal Processing, IEE Proceedings*, vol. 152, pp. 283–296, 3 June 2005.

[39] Xilinx, "Virtex E/EM Product Tables." `http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex_e_e m/resources/virtex_e_em_table.htm`, April 2006. [Online].

[40] Xilinx, "Spartan-3E FPGA Family: Complete Data Sheet." `http://direct.xilinx.com/bvdocs/publications/ds312.pdf`, 21 March 2005. [Online].

[41] S. M. Smith and J. M. Brady, "SUSAN — A New Approach to Low Level Image Processing," *International Journal of Computer Vision*, vol. 23, pp. 45–78, May 1997.

[42] C. Torres-Huitzil and M. Arias Estrada, "An FPGA Architecture For High Speed Edge and Corner Detection," in *CAMP 2000: Fifth IEEE International Workshop On Computer Architectures For Machine Perception*, pp. 112–116, 2000.

[43] R. Townsend, *Digital Computer Structure and Design*. Butterworth and Co. Ltd, 2nd ed., 1982.

[44] H. Keding, M. Willems, M. Coors, and H. Meyr, "FRIDGE: a Fixed-Point Design and Simulation Environment," in *DATE '98: Proceedings of the Conference On Design, Automation and Test in Europe*, (Washington, DC, USA), pp. 429–435, IEEE Computer Society, 1998.

[45] E. L. Oberstar, "Fixed Point Representation And Fractional Math." `http://www.superkits.net/whitepapers/Fixed Point Representation & Fractional Math.pdf`, 2005. [Online].

[46] R. Trummer, P. Zinterhof, and R. Trobec, "A High-Performance Data-Dependent Hardware Divider," in *Parallel Numerics '05*, (Portoroz, Slovenia), pp. 193–205, International Workshop On Parallel Numerics 2005, April 2005.

[47] B. R. Lee and N. Burgess, "Improved Small Multiplier Based Multiplication, Squaring and Division," in *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium On*, pp. 91–97, April 2003.

[48] Opencores, "Hardware Division Units." `http://www.opencores.org/projects.cgi/web/divider/overview`, September 2006. [Online].

[49] J. B. Gosling, *Design of Arithmetic Units For Digital Computers.* Macmillan Computer Science Series, London, UK: The Macmillan Press Ltd, 1980.

[50] A. Leon-Garcia, *Probability and Random Processes For Electrical Engineering.* Addison-Wesley, May 1994.