

# An Investigation Into Multi-Spectral Tracking

CHRISTIAAN WOOD



THESIS PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF MASTER OF SCIENCE IN ELECTRONIC  
ENGINEERING AT THE UNIVERSITY OF STELLENBOSCH

SUPERVISOR: MR JOHANN TREURNICHT

APRIL 2005

# Declaration

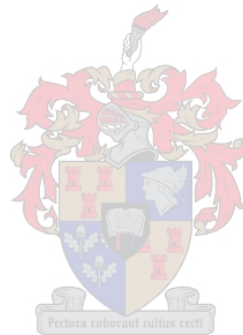
I, the undersigned, hereby declare that the work contained in this thesis is my own original work unless otherwise stated, and has not previously, in its entirety or in part, been submitted at any university for a degree.

.....

Signature

.....

Date

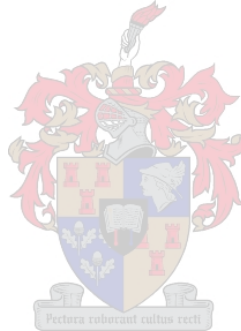


# Abstract

The purpose of this study was to investigate multi-spectral tracking. Various algorithms were investigated and developed to enhance the contrast between target and non-target classes. Different tracking algorithms were implemented on the resulting grayscale input.

A physical tracking system consisting of a video input processor and DSP was designed and built to implement algorithms and investigate the viability of realtime multi-spectral tracking.

It is illustrated that conventional intensity tracking clouds the available information and that by studying various spectral inputs information is extracted more efficiently from the available data.

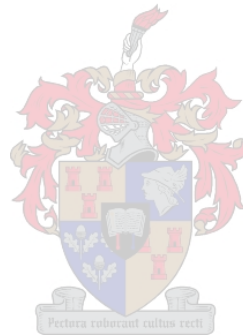


# Samevatting

Die doel van hierdie projek was om multispektrale volging te ondersoek. Verskeie algoritmes om teken teenoor agtergrond kontras te verhoog is ondersoek en ontwikkel. Verskillende volgings algoritmes is op die resulterende grys intree toegepas.

'n Fisiese volgingsstelsel bestaande uit 'n intree video verwerker en DSP is ontwikkel om praktiese intydse multispektrale volging te ondersoek.

Daar word geïllustreer dat konvensionele grysvlak volging die intree informasie verduister en dat deur verskeie spektraalgebiede afsonderlik te beskou informasie meer effektief uit die beskikbare data gehaal word.

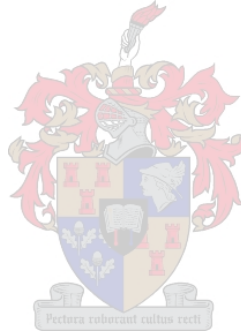


# Acknowledgements

Any ambitious project is a labour of love and would not be possible without the support of many others.

I am eternally indebted to my colleagues and friends at the ESL for their unfaltering support and contribution to a great work atmosphere. Especially Nicol Carstens and Corné van Daalen whose academic and personal input were paramount to the completion of this project.

My supervisor, Johann Treurnicht, whose guidance kept me focussed, support kept me motivated and whose ideas always opened new doors when I saw none.



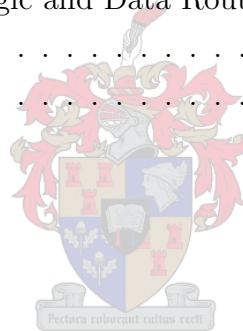
Christiaan Wood  
November 2004

# Contents

<b>Acronyms and Abbreviations</b>	<b>xiii</b>
<b>1 Introduction and Overview</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Background . . . . .	2
1.3 Literature Synopsis . . . . .	2
1.4 Objectives of this Study . . . . .	3
1.5 Dissertation Structure . . . . .	3
<b>2 Design Goals, Overview and Background</b>	<b>5</b>
2.1 Experiment Design . . . . .	6
2.2 A Brief Background on Video . . . . .	7
2.2.1 Analog Television Standards . . . . .	7
2.2.2 Digital Video . . . . .	10
<b>3 Video Processor</b>	<b>13</b>
3.1 Dataflow Considerations . . . . .	13
3.2 Video Processor Hardware and Component Choice . . . . .	16
3.2.1 The Imager . . . . .	16
3.2.2 Analog Video Decoder . . . . .	19
3.2.3 Digital Data Router . . . . .	20
3.2.4 Control Logic . . . . .	22
3.2.5 RAM . . . . .	23
3.2.6 Output Video Logic . . . . .	24
3.2.7 Output Analog Video Encoder . . . . .	25
3.2.8 External Interfaces . . . . .	25
3.2.9 Power Supply . . . . .	27
3.2.10 PCB Design . . . . .	28
<b>4 Video Processor Software Design</b>	<b>29</b>
4.1 Internal ROM . . . . .	30

4.2	I <sup>2</sup> C Protocol . . . . .	31
4.3	I <sup>2</sup> C Configuration . . . . .	33
4.4	Signal Routers . . . . .	40
4.4.1	CPLD U12 Data Router . . . . .	41
4.4.2	CPLD U13 Data Router . . . . .	42
4.4.3	UART . . . . .	42
4.4.4	CPLD U8 Data Router . . . . .	43
4.5	FPGA U9 Control Logic . . . . .	44
<b>5</b>	<b>DSP</b>	<b>49</b>
5.1	The TI TMS320C6211 DSP . . . . .	49
5.2	External Memory Interface . . . . .	50
5.3	DSP Software . . . . .	53
5.3.1	R/W RAM . . . . .	55
5.3.2	Recognizing Input Signal Rising Edges . . . . .	56
5.3.3	Generating the Tracking Overlay . . . . .	57
5.3.4	User Centroid Input . . . . .	58
<b>6</b>	<b>Colour Transforms</b>	<b>59</b>
6.1	Background . . . . .	59
6.1.1	Different Colour Spaces/Models . . . . .	60
6.1.2	Visualizing Colour Spaces . . . . .	63
6.2	General Colourspace Transforms . . . . .	66
6.3	Intelligent Transforms . . . . .	66
6.4	Clustering . . . . .	78
6.5	Automatic Target Detection . . . . .	82
6.6	Conclusions . . . . .	86
<b>7</b>	<b>Tracking Theory and Algorithms</b>	<b>87</b>
7.1	Centroid Tracking . . . . .	87
7.2	Correlation Tracking . . . . .	91
7.3	Estimator . . . . .	94
7.4	Simulation . . . . .	96
<b>8</b>	<b>Hardware Integration</b>	<b>97</b>
<b>9</b>	<b>Tests and Results</b>	<b>103</b>
9.1	Characterizing the System . . . . .	103
9.2	Laboratory Tests . . . . .	104
9.3	Field Tests . . . . .	108

<b>10 Conclusions and Recommendations</b>	<b>113</b>
10.1 Conclusions . . . . .	113
10.2 Recommendations . . . . .	114
<b>A Schematics and PCB Layouts</b>	<b>118</b>
<b>B Extraneous Data</b>	<b>131</b>
B.1 SAA7111A Video Decoder Control Registers . . . . .	131
B.2 SAA7127H Video Encoder Control Registers . . . . .	133
B.3 VHDL ROM Memory Initialization File . . . . .	136
B.4 DSP Asynchronous Read/Write Timing Diagrams . . . . .	145
B.5 VHDL Simulations - Waveform Files . . . . .	147
<b>C Code Listing</b>	<b>159</b>
C.1 FPGA VHDL . . . . .	159
C.2 CPLD Data Router VHDL . . . . .	186
C.3 Video Output Glue Logic and Data Router VHDL . . . . .	194
C.4 DSP Code . . . . .	196
C.5 MATLAB Code . . . . .	213





# List of Figures

2.1	Blockdiagram of a Generic Tracking System . . . . .	5
2.2	Single Analog Video Line . . . . .	8
2.3	Full Analog Video Frame . . . . .	8
2.4	Single Analog Colour Video Line . . . . .	10
3.1	Video Processor General Data Flow . . . . .	14
3.2	Data Throughput on Odd Frames . . . . .	15
3.3	Data Throughput on Even Frames . . . . .	15
3.4	RAM Pingpong Buffer . . . . .	15
3.5	Video Processor Block Diagram . . . . .	17
3.6	E-526SP Camera and Tripod . . . . .	19
3.7	SAA7111 Block Diagram . . . . .	21
3.8	The video processor RAM . . . . .	24
3.9	Cascading JTAG Devices . . . . .	26
3.10	The Video Processor . . . . .	28
4.1	Video Processor Data Paths . . . . .	30
4.2	I <sup>2</sup> C bus setup . . . . .	31
4.3	I <sup>2</sup> C bus START and STOP conditions . . . . .	32
4.4	I <sup>2</sup> C bus Data Transfer . . . . .	33
4.5	I <sup>2</sup> C IC Hardware . . . . .	33
4.6	I <sup>2</sup> C Software Flow Diagram . . . . .	34
4.7	Clock/data timing for RGB (8, 8 and 8) output format. . . . .	37
4.8	Vertical Timing Diagram - First Field . . . . .	39
4.9	Vertical Timing Diagram - Second Field . . . . .	39
4.10	YUV 4:2:2 Data Format . . . . .	43
4.11	Software Flow Diagram of the Video Processor . . . . .	46
5.1	Texas Instruments TMS320C6211 Blockdiagram . . . . .	50
5.2	Texas Instruments TMS320C6211 DSK . . . . .	51
5.3	TMS320C6211 External Memory Interface . . . . .	51
5.4	Flow Diagram of DSP Software . . . . .	54

5.5	Tracking Overlay . . . . .	57
6.1	Spectra of RGB signals . . . . .	61
6.2	The U-V Colour Plane, $Y = 0.5$ . . . . .	62
6.3	The HSV colour space as cone . . . . .	63
6.4	The RGB colour space, Brightness = 25% . . . . .	64
6.5	The RGB colour space, Brightness = 50% . . . . .	64
6.6	The RGB colour space, Brightness = 75% . . . . .	64
6.7	RGB Image Displayed as Normal . . . . .	65
6.8	RGB Image displayed in RGB space . . . . .	65
6.9	Smarties - Grayscale Image . . . . .	68
6.10	Smarties - Red Plane . . . . .	68
6.11	Smarties - Green Plane . . . . .	68
6.12	Smarties - Blue Plane . . . . .	68
6.13	Smarties - RGB Image . . . . .	69
6.14	Smarties - Hue Plane . . . . .	69
6.15	Smarties - Saturation Plane . . . . .	69
6.16	Smarties - Value Plane . . . . .	69
6.17	Selecting Target Object . . . . .	72
6.18	Target, Background and Difference Vectors . . . . .	73
6.19	Image Projected Onto Difference Vector and Scaled . . . . .	73
6.20	Image Projected Onto Difference Vector, Negative Values Discarded . . . . .	74
6.21	Normalized Target and Background Vectors with the Resulting Difference Vector . . . . .	75
6.22	Image Projected Onto Normalized Difference Vector and Scaled . . . . .	76
6.23	Image Projected Onto Normalized Difference Vector, Negative Values Dis- carded . . . . .	76
6.24	Scaled and Truncated Output Image after Saturation Normalization . . . . .	77
6.25	Input RGB Image, Specifying Target . . . . .	79
6.26	Output Image . . . . .	79
6.27	Output Image - Normalizing Colour Vectors . . . . .	79
6.28	RGB Image, Normalized, In RGB Space . . . . .	80
6.29	Input Image - Single Object Tinted Blue . . . . .	81
6.30	Pixels Belonging to Blue Cluster . . . . .	81
6.31	Input Image in RGB space (Normalized Brightness) . . . . .	81
6.32	Target Cluster . . . . .	81

6.33	Background Cluster . . . . .	81
6.34	Input RGB Image . . . . .	83
6.35	Input RGB Image in RGB Space . . . . .	83
6.36	Pixels Belonging to Cluster 1 . . . . .	83
6.37	Cluster 1 in RGB Space . . . . .	83
6.38	Pixels Belonging to Cluster 3 . . . . .	83
6.39	Cluster 3 in RGB Space . . . . .	83
6.40	Automatic Detection of Red Car . . . . .	85
6.41	Automatic Detection of Blue Leaf . . . . .	85
6.42	Automatic Detection of Red Leaf . . . . .	85
6.43	Automatic Detection of Red Smartie . . . . .	85
6.44	Automatic Detection of Blue Smartie . . . . .	85
7.1	Target Image of Arbitrary Shape Plotted as a 3D Function . . . . .	88
7.2	Target Image of Arbitrary Shape . . . . .	88
7.3	Binarized Target Image . . . . .	88
7.4	Centroid Calculated with Mean Values . . . . .	89
7.5	Centroid Calculated with Median Values . . . . .	89
7.6	Centroid Calculated with Weighted Values . . . . .	89
7.7	Centroid Calculated with the Brightest Point . . . . .	89
7.8	Centroid Tracking Flow Diagram . . . . .	90
7.9	Cross Correlation Algorithm . . . . .	91
7.10	Cross Correlation Algorithm for Images of Equal Size . . . . .	92
7.11	Correlation Tracking Flow Diagram . . . . .	93
7.12	Filter Outputs with Ramp Input - 1 . . . . .	95
7.13	Filter Outputs with Ramp Input - 2 . . . . .	95
7.14	Filter Outputs with Noisy Ramp Input . . . . .	95
7.15	Composite .avi Screenshot from MATLAB . . . . .	96
8.1	First Image Read from RAM . . . . .	98
8.2	Second Image Read from RAM . . . . .	98
8.3	RGB Test Image . . . . .	99
8.4	RGB Test Image Read From RAM . . . . .	99
8.5	Voltage Between Power Ground and DSP Digital Ground . . . . .	100
8.6	Ghost Images due to Crosstalk . . . . .	100
8.7	The Complete Tracking System . . . . .	102
9.1	Time Delays in Tracking System . . . . .	104
9.2	Tracking Coloured Wires . . . . .	105
9.3	Tracking a Soldering Iron . . . . .	106

9.4	Tracking Soldering Wire . . . . .	107
9.5	Tracking Humans in Cluttered Environments - 1 . . . . .	109
9.6	Tracking Humans in Cluttered Environments - 2 . . . . .	110
9.7	Tracking Humans in Cluttered Environments - 3 . . . . .	111
9.8	Tracking a Moving Vehicle - 1 . . . . .	112
A.1	Video Processor Schematics - Power Regulators . . . . .	119
A.2	Video Processor Schematics - Video Input . . . . .	120
A.3	Video Processor Schematics - Video Output . . . . .	121
A.4	Video Processor Schematics - 1K50 Control Logic . . . . .	122
A.5	Video Processor Schematics - CPLD Data Routers . . . . .	123
A.6	Video Processor Schematics - External Interfaces . . . . .	124
A.7	Video Processor Schematics - RAM Bank 1 . . . . .	125
A.8	Video Processor Schematics - RAM Bank 2 . . . . .	126
A.9	Video Processor Schematics - DSP Connector . . . . .	127
A.10	Video Processor PCB - Top Layer . . . . .	128
A.11	Video Processor PCB - Bottom Layer . . . . .	129
A.12	DSP Connector PCB . . . . .	130
B.1	Asynchronous Read Timing . . . . .	145
B.2	Asynchronous Write Timing . . . . .	146
B.3	Control Logic Signals - Camera Data Latch - First Pixel . . . . .	148
B.4	Control Logic Signals - Camera Data Latch - Multiple Pixels of the same Frame . . . . .	149
B.5	Control Logic Signals - Camera Data Latch - Multiple Pixels of different Frames . . . . .	150
B.6	Data Router - All Outputs in HI Z . . . . .	151
B.7	Data Router - Latch Input Data from Camera to RAM 1 . . . . .	152
B.8	Data Router - Latch Input Data from Camera to RAM 2 . . . . .	153
B.9	Data Router - RAM to DSP . . . . .	154
B.10	Data Router - DSP to RAM . . . . .	155
B.11	Data Router - Constant (Test) Value to RAM . . . . .	156
B.12	Data Router - Colour 1, RAM 1 Data Serial Transmission (to PC) . . . . .	157
B.13	Data Router - Constant (Test) Data Serial Transmission (to PC) . . . . .	158

# List of Tables

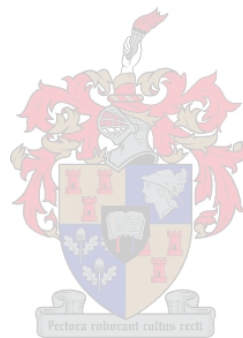
3.1	Analog Video Decoders . . . . .	19
3.2	RAM Comparison . . . . .	23
3.3	JTAG Pin Descriptions . . . . .	26
4.1	SAA7111A For RGB Output Control Registers . . . . .	36
4.2	SAA7111A For Grayscale Output Control Register . . . . .	36
4.3	Output Format of the VPO bus . . . . .	38
4.4	SAA7127H Control Register . . . . .	40
4.5	Data Routing Control Signals . . . . .	41
4.6	Output Data Routing Control Signals . . . . .	42
4.7	Video Processor Mode Control Signals . . . . .	45
5.1	EMIF Signal Descriptions . . . . .	52
B.1	SAA7111A For RGB Output Control Registers . . . . .	132
B.2	SAA7127H Control Registers . . . . .	135



# Acronyms and Abbreviations

<b>A/D</b>	Analog to Digital Converter
<b>AC</b>	Alternating Current
<b>CCIR</b>	International Radio Consultative Committee
<b>CREF</b>	Clock Reference
<b>CRT</b>	Cathode Ray Tube
<b>CPLD</b>	Complex Programmable Logic Device
<b>CVBS</b>	Color, Video, Blank and Sync
<b>D/A</b>	Digital to Analog Converter
<b>DC</b>	Direct Current
<b>DSK</b>	Digital Signal Processor Starter Kit
<b>DSP</b>	Digital Signal Processor
<b>EMIF</b>	External Memory Interface
<b>ESL</b>	Electronic Systems Laboratory
<b>FIR</b>	Finite Impulse Response
<b>FPGA</b>	Field Programmable Gate Array
<b>fps</b>	Frames per second
<b>HREF</b>	Horizontal Frame Sync
<b>HSV</b>	Hue Saturation Value
<b>I/O</b>	Input / Output
<b>I<sup>2</sup>C</b>	Inter-IC Bus
<b>IC</b>	Integrated Circuit
<b>IEEE</b>	Institute of Electrical and Electronic Engineers
<b>JTAG</b>	Joint Test Action Group
<b>MATLAB</b>	Mathematics Laboratory
<b>MIPS</b>	Million Instructions per Second
<b>NTSC</b>	National Television System(s) Committee
<b>PAL</b>	Phase Alternating Line
<b>PC</b>	Personal Computer
<b>PCB</b>	Printed Circuit Board
<b>QPSK</b>	Quadrature Phase Shift Keying
<b>RAM</b>	Random Access Memory
<b>RGB</b>	Red, green and blue
<b>ROM</b>	Read Only Memory

<b>TV</b>	television
<b>UART</b>	Universal Asynchronous Receiver Transmitter
<b>US</b>	University of Stellenbosch
<b>USB</b>	Universal Serial Bus
<b>VPO</b>	Video Port Out
<b>VREF</b>	Vertical Frame Sync
<b>VHDL</b>	Very High Speed Integrated Circuit Hardware Description Language





# Chapter 1

## Introduction and Overview

### 1.1 Motivation

Tracking targets in low contrast and highly cluttered environments is an age old problem. Consider the scenario where an observer is following an aircraft with the naked eye. Against the uniform blue backdrop presented by the sky this is a trivial problem, but as soon as the background changes to something non-uniform following the aircraft becomes exceedingly difficult. Automatic trackers have similar problems and are even disturbed by random reflections from the targets they are tracking.

In the past, conventional optical tracking methods consisted of algorithms applied to an intensity image, a grayscale image representing an average intensity value of the broad input spectrum. This approach is suitable in most high contrast tracking applications (for example, targets moving with respect to a uniform or low intensity background), but for a cluttered low target-background contrast scenario this approach is problematic and requires complex tracking algorithms.

Due to the increasing availability of multi-spectral sensors (sensors providing simultaneous grayscale video feeds representing the average intensity of various narrow spectral regions) it has become viable to investigate the possibility of adapting existing algorithms or develop new tracking algorithms based on the multi-spectral inputs. It is immediately apparent that a colour image conveys more information than a grayscale, but synthesizing this to provide more accurate and robust tracking is the challenge faced in this thesis.

## 1.2 Background

The Electronic Systems Laboratory (ESL) was formed in 1991 as a part of the Electronic Engineering Department of the University of Stellenbosch (US). The ESL's most ambitious project came to fruition in 1999 with the launch of SUNSAT, South Africa's first locally developed satellite. The project was so successful that many subsystems were sold to international customers. As a result, a spin-off company, Sunspace Information Systems was created, which is now solely dedicated to the commercial development of satellites and satellite subsystems.

This initial success has fuelled further development of specific satellite subsystems at the ESL. An academic relationship with an European University hinted towards the availability of a multi-spectral sensor unit which prompted an investigation into the multi-spectral tracking problem.

## 1.3 Literature Synopsis

Much information is available on the conventional grayscale tracking problem. Books by Bar-Shalom ([5], [6]) were found to be particularly informative. Various articles [13], [16], ([17] and [21]) examine more specific tracking problems and algorithms. Such articles were usually on tracking in a very specific environment and were not always useful in their entirety.

In the early 1990's Swain and Ballard [29] showed that the intersection of colour histograms was an effective means of recognizing coloured objects. This technique was sensitive to the colour and intensity of the ambient light source and reflection. In their research on skin detection, Schiele and Waibul [25] have since demonstrated that the effect of ambient lighting could be reduced by normalizing the colour vector through dividing out the luminance component. Finlayson [12] has written some important texts on using colour for image indexing, highlighting some of the pitfalls of the approach.

Precious little information is available on the combination of these two fields - the multi-dimensional tracking problem. The vast majority of literature in this field focusses on facial feature tracking, most notably texts by Schiele and Waibul [25] and Crowley and Berard [8].

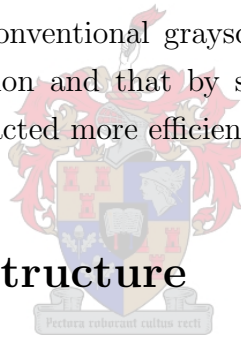
## 1.4 Objectives of this Study

The objectives of this study are twofold, firstly, to develop a low cost multi spectral sensor unit and a DSP-based tracking system. Hardware emphasis will be placed on developing and demonstrating the multi-spectral tracking concept and development focussed on conceptual demonstration as opposed to commercial use by an end user.

Secondly, multi-spectral sensor data will be used to provide a different perspective on the tracking problem. Alternative ways of increasing target-background contrast will be investigated, so as to facilitate more accurate and robust tracking. Emphasis will be placed on information extraction using colour transforms and tracking using slightly modified standard algorithms as opposed to developing new tracking algorithms. This will be used to illustrate the plausibility of multi-dimensional data extraction with the tracking problem. By designing a stand-alone tracking system, all practical considerations will be investigated and the plausibility of realtime multi-spectral tracking carefully considered.

It will be illustrated that conventional grayscale intensity tracking clouds (smears) the available spectral information and that by studying various different spectral images/inputs information is extracted more efficiently from the available data.

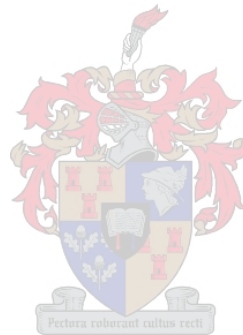
## 1.5 Dissertation Structure



The thesis is structured as follows:

- Chapter 1 motivates the thesis, gives some background and an overview of the scope of the work.
- Chapter 2 discusses the goals and objectives of the work and offers a brief overview of video signals.
- Chapter 3 covers the hardware design of the video processor, and Chapter 4 the software design.
- Chapter 5 discusses the DSP used in the project and presents a basic software framework from where more algorithm specific software can be discussed.
- Colour transforms are discussed in detail in Chapter 6.
- Chapter 7 covers tracking theory and algorithms.

- The integration of all subsystems is discussed in Chapter 8.
- Chapter 9 presents the results.
- Chapter 10 concludes the thesis with a discussion of the most important aspects of the project and what was learnt. A few recommendations are also given as to the problems experienced and scope for further works.

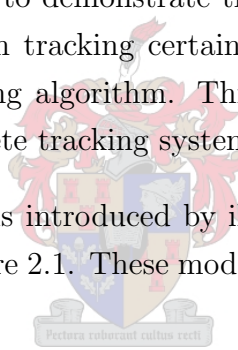


## Chapter 2

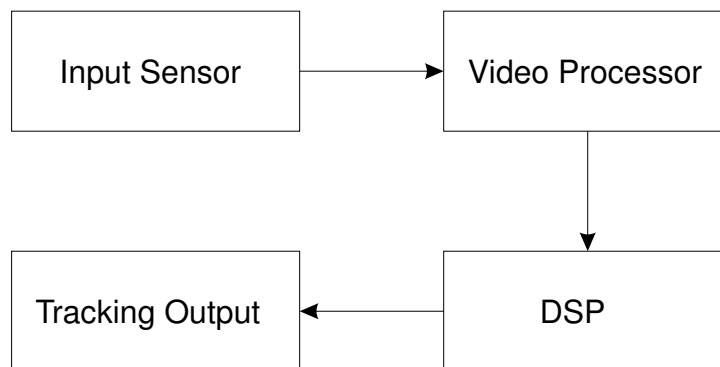
# Design Goals, Overview and Background

The broad goal of this thesis is to demonstrate that by considering multi-spectral information a substantial increase in tracking certainty could be achieved without complex changes to the standard tracking algorithm. This is to be demonstrated as practically viable by developing the complete tracking system.

The discussion on tracking is introduced by illustrating a blockdiagram of a generic practical tracker, shown in Figure 2.1. These modules all appear in some form in different trackers.



Firstly the tracking scenario is acquired by some type of input sensor. The type of input sensor varies (it can be an infra red camera or even radar) and in this case it will be a multi-spectral imager - i.e., a camera providing grayscale video feeds from various spectral regions. This video signal then needs to be processed and stored in memory. This module is commonly known as a framegrabber or capture board. Most conventional



**Figure 2.1:** *Blockdiagram of a Generic Tracking System*

framegrabbers are designed for use in a PC, with internal framegrabbers storing video data on the PC hard disk or external framegrabbers outputting a digital video stream. The framegrabber designed in this thesis differs from conventional framegrabbers in that input video data are written to a large RAM bank, which can be accessed by any external device as normal RAM. The framegrabber module designed for this thesis is subsequently referred to as the *video processor*. There are different approaches in acquiring video data for tracking, some requiring large amounts of memory and some not (this is discussed in further detail in Section 3.1). By configuring the video processor with enough memory to store an entire frame, much flexibility is gained in implementing algorithms, as all the data is available for processing.

Captured video data is then read by a DSP which implements all tracking algorithms and outputs tracking data. Tracking data would usually consist of the location of the object and some form of tracking certainty. The object location can be used to point a pointing device towards the target, or, as in the case of this thesis, to superimpose a rectangular tracking overlay over the input video stream, indicating the position of the target.

## 2.1 Experiment Design

A demonstration design affords one the luxury of carefully tailoring the experiment. This gives leeway in the design of the system as not all scenarios need to be considered and certain experimental restrictions can be set.

The most important experimental restrictions were:

- Limit the bandwidth and range of target movement. This was dependant on the target distance from the camera, but would amount to the target not moving out of the region of interest from one frame to the next. This would typically be 100 pixels per frame.
- Maintain the target vs background size to approximately 10%. This allows the use of a tracking window of constant size, simplifying tracking.
- Restrict the target to any one dominating colour.
- Video resolution was set to 640x480 pixels. This was essentially an arbitrary choice, but large enough to test algorithms and display with good enough quality, and small enough so as not to require excessive amounts of RAM and processing power.

## 2.2 A Brief Background on Video

Any design or discussion on video processor hardware requires a significant background knowledge of both digital and analog video. As the input sensor as well as output display used in this thesis were analog a few words on analog video and television signals are necessary. Inter-IC video data transmission in the video processor were in various digital formats, hence these protocols are briefly discussed.

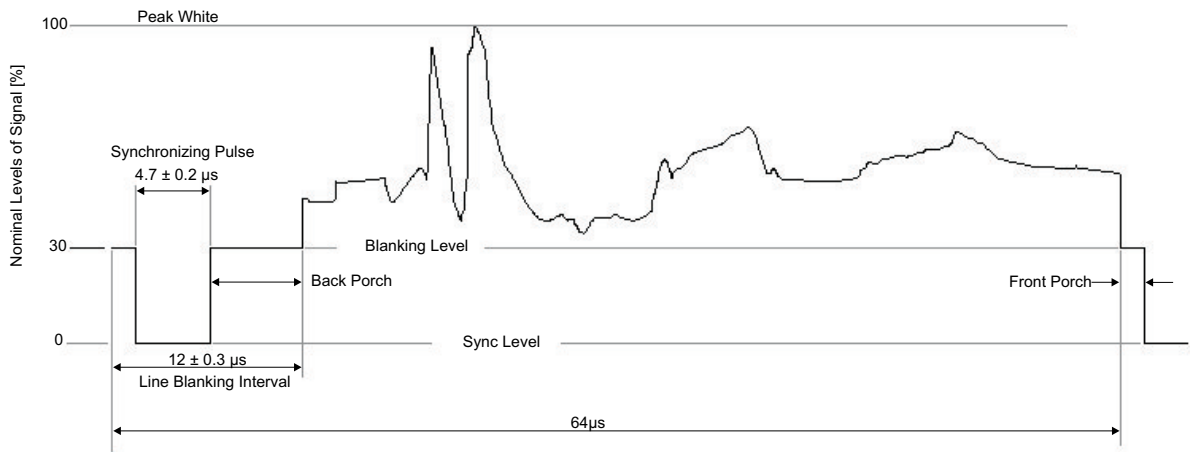
### 2.2.1 Analog Television Standards

Understanding how analog video signals are displayed on a television screen is paramount to the understanding of the signal format itself and this will be the starting point.

A black and white television picture is built up of a single white spot rapidly scanning (both horizontally and (relatively slowly) vertically) the faceplate of a cathode ray tube (CRT). The brightness of the spot is varied to produce the picture. A synchronization signal to control the position of the spot on the screen is combined with the luminance (brightness) information to form the 'composite' TV signal. The synchronization and luminance information is kept separate by assigning them unique signal ranges. Typically the range of a monochrome (black and white) video signal is  $1V_{p-p}$  (although DC levels vary - usually the signals are AC coupled). The luminance information occupies a region of  $0.7V$  and the sync  $0.3V$ . The brightest luminance level (white) corresponds to  $1V$ , while the lowest luminance level (black) corresponds to  $0.3V$ . Black levels are the same as the blanking levels in a CCIR compatible system (Subsection 2.2.2) - blanking occurs at times where the electron gun to the CRT needs to be off (where a retrace needs not be visible on the screen).

This composite TV waveform contains pulses at sub-black level that allow synchronization in both the horizontal and vertical deflection circuits. Both need to be triggered at the right time to ensure the spot is at the correct position on the CRT. Therefore both timebases need to be synchronized and triggered independently off the same set of sync pulses. The method used to allow the receiver to distinguish between these two pulses is to use different widths for the horizontal and vertical sync pulses.

A single video line is shown in Figure 2.2 (redrawn from [19]). The horizontal sync (otherwise known as the 'line' sync) pulse separates each video line and ensures horizontal synchronization. For black and white TV, the *front* and *back porches* are well defined black level spaces before and after a valid video line: the front porch is 0.02 times the distance

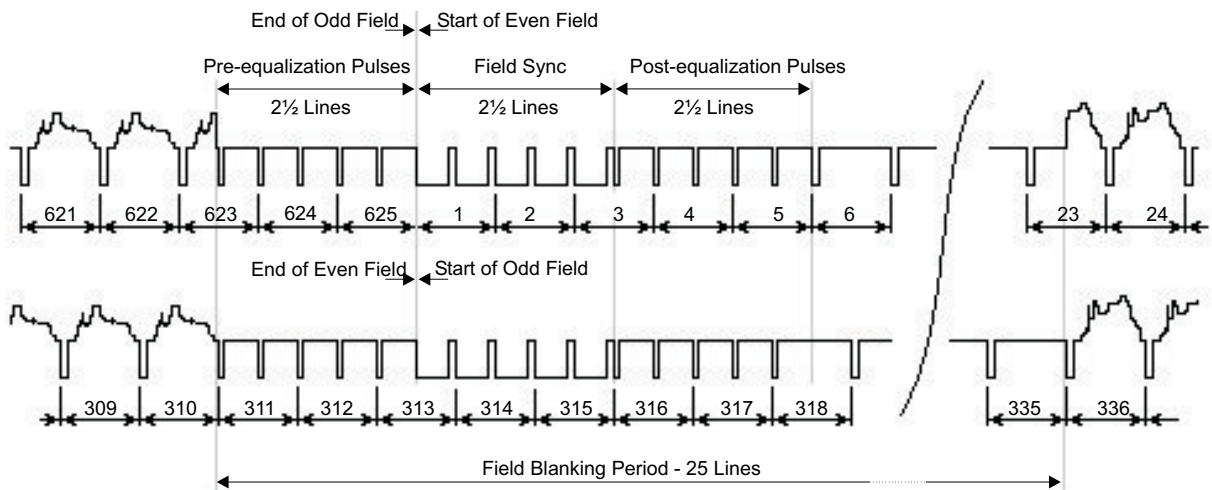


**Figure 2.2:** *Single Analog Video Line*

between pulses, and the back porch is 0.06 times the distance between pulses.

The transmission of a full video frame as well as the vertical synchronization pulses (otherwise known as 'frame' syncs) is shown in Figure 2.3 (redrawn from [19]).

Taking the frame rate as 25 frames per second with 625 lines per frame (European PAL (Phase Alternating Line) standard) it can be seen that the time taken for a single line is  $64\mu\text{s}$  (Figure 2.2).



**Figure 2.3:** *Full Analog Video Frame*

The broad pulses (or serrations) are used to trigger the vertical sync generator. In order to keep the line sync generator running, falling edges (line syncs) still appear every  $64\mu\text{s}$  during the frame sync period. The equalization pulses stem from the analog circuitry initially used in analog receivers. One way to differentiate between line and frame syncs



was, having AC coupled the sync pulses, to integrate the signal (after video information had been stripped) causing a downward level drift during the frame sync period and a negative level triggering the vertical sync generator. The equalization pulses allow the integrator to settle before and after the frame sync pulses.

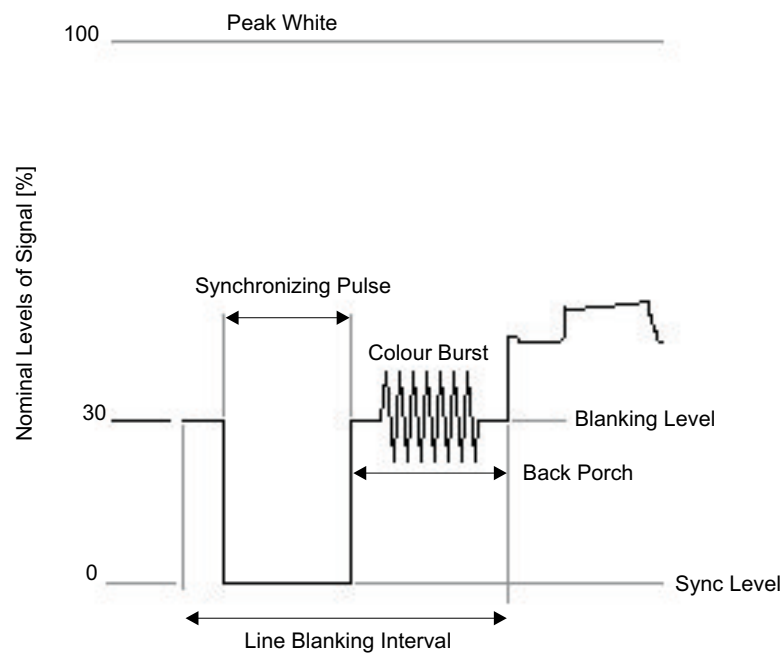
Another concept that needs to be discussed is 'interlacing'. This was introduced in the early days of mechanical video, but persists today as a bandwidth conserving measure. For European (and South African) television, a picture refresh rate of 25 frames per second was chosen<sup>1</sup>. However, showing pictures at this rate would produce an obtrusive flickering effect. This is because the eye is sensitive to changes in brightness at frequencies of up to about 40Hz. In television, increasing the refresh rate by a factor of two is achieved by interlacing, that is, splitting each picture into two fields. Each field is a single vertical frame scan. Each field consists of half the total picture frame lines and is scanned from top to bottom of the display at twice the picture rate. The second field completes the picture by scanning its lines in between those of the first field. Thus a complete picture is produced at 25 frames per second, but the refresh rate is twice that (50Hz) producing a smooth image transition and video.

Colour television could be implemented by transmitting red, green and blue signals and using the electron guns to display these three colours. This was not possible, as the required bandwidth would be higher than the available bandwidth. Also, the analog protocol needed to be improved to allow the transmission of colour, but with the backward compatibility to allow this colour signal to be sent to a black and white TV and be displayed correctly. The colour transmission protocol was carefully structured to preserve all original monochrome information and only add colour. To achieve this, a colour synchronization signal (colour burst) is inserted into the back porch of the vertical blanking signal, Figure 2.4.

The colour data (U and V components) are added to the luminance signals as a QPSK modulated signal. The colour burst acts as reference for the QPSK demodulator to phase lock onto before each line. This colour burst sets the colour phase for the active line[14], but is ignored by a black and white receiver. This composite video signal is known as the

---

<sup>1</sup>The choice of 25 frames/s (as opposed to 24 frames/s for early cinema) also stems from analog TV. In the early days of television, the mains (at 50Hz) would break through quite easily from an imperfectly filtered power supply. This would manifest itself by modulating the brightness or position of the picture. If the difference in mains and picture frequency were 1Hz (25Hz - 24Hz) this modulation would show up as an anomaly tracing the screen once per second and would be very obtrusive. By using a picture frequency of 25 frames/s (with an effective scan frequency of 50Hz) the mains modulation would be stationary on the screen and far less visible.



**Figure 2.4:** *Single Analog Colour Video Line*

CVBS (Color, Video, Blank and Sync) video signal .

## 2.2.2 Digital Video

This analog video signal is the input to the video processor, which converts the signal to a digital format.

Most digital video signals are said to be 'CCIR' compatible. The International Radio Consultative Committee (CCIR), currently known as ITU Radiocommunication Sector (ITU-R) publishes standards for encoding interlaced analog video signals to digital form. The two common standards are CCIR 601 and CCIR 656. They are complementary standards, with CCIR 601 mainly dealing with sampling concepts, while CCIR 656 concentrates on the parallel and serial distribution and details the start/end of active video (EAV/SAV) timing reference signal [10].

Timing references are much easier in digital video, each line is initiated with a timing reference signal and information byte (SAV) and ended with another timing reference signal and information byte (EAV). The EAV/SAV bytes contain information pertaining to the field and position of the video line.

Most digital image sources generate a RGB (red, green and blue) signal, with three independent colour channels. Due to bandwidth constraints, digital video is transmitted in a different format, known as YUV. In the YUV colour space, Y is the brightness (luminance) component of the signal and U and V the colour (chrominance) components. Converting between the RGB and YUV colour spaces is done using Equations 2.1 and 2.2.

$$Y = 0.299R + 0.587G + 0.114B \quad U = (B-Y)*0.565 \quad V = (R-Y)*0.713 \quad (2.1)$$

with reciprocal versions:

$$R = Y + 1.403V \quad G = Y - 0.344U - 0.714V \quad B = Y + 1.770U \quad (2.2)$$

It can be seen that U is the blue colour difference signal and V the red colour difference signal. U and V are frequently referred to as  $C_b$  and  $C_r$  respectively as they are simply scaled versions of each other. The advantages of the YUV (or  $C_b$ -Y- $C_r$ ) signal is that (as with the colour analog TV signal) it remains compatible with black and white television (only consider the Y signal) and that it can easily be manipulated to discard some information to reduce bandwidth. As the human eye has a relatively low colour resolution, high resolution colour images are obtained by combining a high resolution grayscale image with a low resolution colour image. This process is very difficult in the RGB colour space, but trivial in YUV.

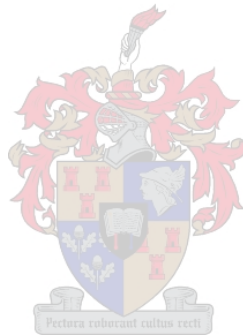
YUV images can be sampled in various ways, such as (in decreasing order of quality):

- YUV 4:4:4
- YUV 4:2:2
- YUV 4:2:0

The numbers reflect the relative importance given to each channel. The first gives the relative amount of Y information on each line, the second the relative amount of U and V information on every even numbered line and the third the same for odd lines. For example, YUV 4:2:2 (the most common format), has half the chrominance information than luminance. The data sequence would be the following: Y1 U1 Y2 V2 Y3 U3 Y4 V4. For every pixel the luminance and one chrominance value is transmitted, cycling U and V. During display, the missing values are approximated by interpolation from the values that are present.

The sampling frequency of digital video was set as 13.5MHz with 720 active pixels per line. This was because of the subcarrier frequency and aspect ratio of the image, however a detailed explanation is beyond the scope of this discussion and can be found in [2].

This concludes the discourse on different video signal formats and now the discussion proceeds to the design of the video processor and imager.



# Chapter 3

## Video Processor

Any complex design is a recursive process with component choice inextricable from software design<sup>1</sup>. A complete understanding of the video processor is needed to sufficiently justify most component choices. Conversely, a good understanding of the hardware is necessary before detailed software design can be attempted. This is a catch twenty two situation, so we will first consider various dataflow diagrams to gain understanding of the broader functions of the video processor, then consider component choices by the most important design constraints, without occluding the chapter with too much software detail.

The video processor can be seen as the data acquisition unit of the tracking system. A DSP was used to implement all the algorithms on the video data acquired by the video processor. So, in essence, the video processor is required to acquire video data and maintain data throughput, but not implement tracking or colour transformation algorithms.

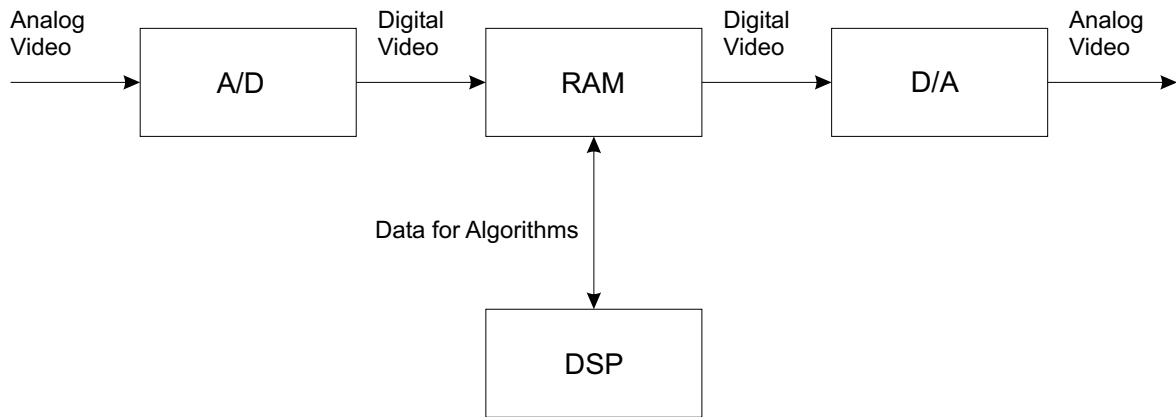
### 3.1 Dataflow Considerations

The video processor is the data acquisition and logic core of the tracking system. It has many functions:

- Analog to digital conversion of input video
- Storing a frame of video to RAM

---

<sup>1</sup>Most software was completed before hardware components were chosen, so as to ensure the chosen components would have sufficient resources



**Figure 3.1:** *Video Processor General Data Flow*

- Making this data available for DSP processing
- Digital to analog conversion of output video
- All glue- and timing logic

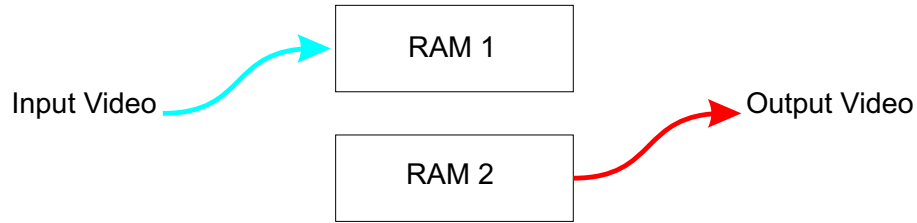
As well as performing all of the above functions, the video processor needs to maintain dataflow independently of algorithm processing. Two distinctly different system configurations are possible with different dataflow formats:

- A pipeline, fall through, memoryless system. The input frame is broken into smaller sections (to reduce processing time) and algorithms implemented realtime on these regions of interest. This approach has the advantage that hardware design is simple. The disadvantages of such an approach is that it limits algorithm development to those that can be executed in realtime on minimal data.
- A memory buffered system. In this configuration, the input data are written to a memory buffer and algorithms implemented on the data in memory. The disadvantage of such an approach is that the hardware design becomes much more complex.

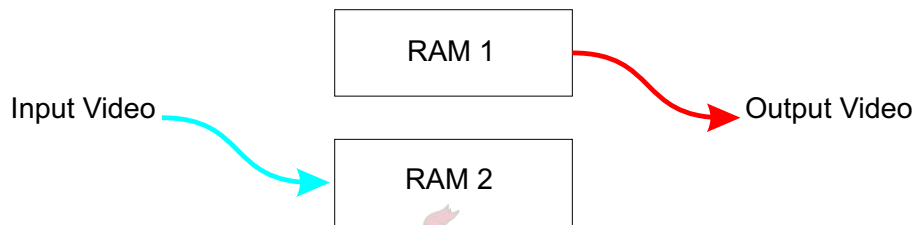
It was decided to design a video processor based on the memory buffered configuration as this gave the freedom to test and implement algorithms on an entire frame without processing time limitations. Processing could still occur in realtime, with a single frame delay to the output, or as slow as was desired. A blockdiagram illustrating the main data flow in the video processor is thus shown in Figure 3.1.

From Figure 3.1 it can be seen that input data are written to the RAM, data is read and written to and from the DSP, and output data are then read from the RAM to be

displayed. It is immediately apparent that the memory buffer is a potential bottleneck. To facilitate efficient dataflow in memory, a dual memory buffer, or ping pong buffer was implemented. Video throughput is illustrated in Figures 3.2 and 3.3.

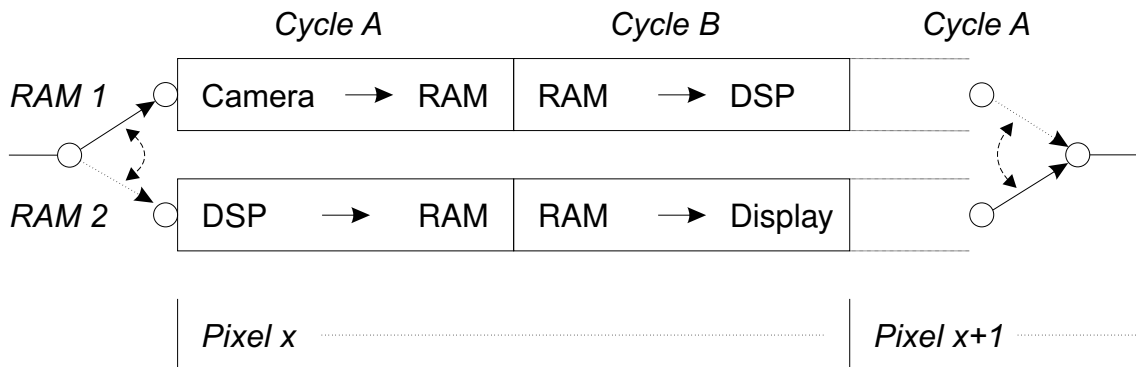


**Figure 3.2:** *Data Throughput on Odd Frames*



**Figure 3.3:** *Data Throughput on Even Frames*

On odd frames, all input video data are written to RAM 1 and read from RAM 2. For even frames, this process is reversed - hence the term pingpong buffer. Note that no communication with the DSP has yet been implemented. This is achieved by creating a dual cycle pingpong buffer, shown in Figure 3.4.



**Figure 3.4:** *RAM Pingpong Buffer*

Each pixel write is broken into two cycles, A and B. For odd frames, data from the camera are written to RAM 1 on cycle A. Simultaneously, the DSP writes tracking information (from the previous frame) to RAM 2. On cycle B, data from RAM 1 are written to the DSP while data from RAM 2 are read to be displayed. This process is repeated for every pixel (of the frame) and the RAM banks switched after every frame.

The dataflow in the video processor is considered in more detail in Figure 3.5. Blue lines indicate analog data, green lines digital data and red lines control signals. To achieve multiple datapaths interfacing with multiple devices, a data router was used to connect the appropriate devices at the appropriate times.

Input data from the camera was converted to a digital data stream by the video decoders and sent to the data router. From here, data are written to RAM as well as the DSP. The DSP implements all algorithms and outputs data (via the data router) to the RAM. The DSP would typically superimpose tracking information over the image. Data are then read from the RAM, arranged in the appropriate format and encoded back to an analog stream to be displayed on a conventional television. For debugging and reading data directly from the RAM, an UART was included for communication with a PC.

## 3.2 Video Processor Hardware and Component Choice

### 3.2.1 The Imager

At the outset of the project, the possibility of using an available multi-spectral sensor existed, but it soon became apparent that this would not be an option. The entire imaging system would need to be designed from scratch. The imager design then became an optimization process characteristic of the entire project - design a subsystem capable of accurate technology demonstration, whilst keeping costs as low as possible and not getting entangled in the inherent complexities of each subsystem.

The basic requirement was a multi-spectral (limited to four spectral regions for demonstration, with theory expanding to hyper-spectral applications) imager. A first iteration was four monochrome cameras, arranged in an array, each with the appropriate filter, thus confining the response of each to a specific spectral region. The drawbacks of such a multi-camera approach were numerous:

- The cameras needed to be perfectly aligned and did not have the same origin.
- The cost of multiple sensors and filters were too high.
- The four camera outputs needed to be synchronized and synthesized into a single, multi-dimensional input signal. This created extra logic and timing complexity.

It was initially decided to use two cameras, one outputting a RGB signal and the other grayscale. The grayscale camera was to be covered by an infrared filter, so as



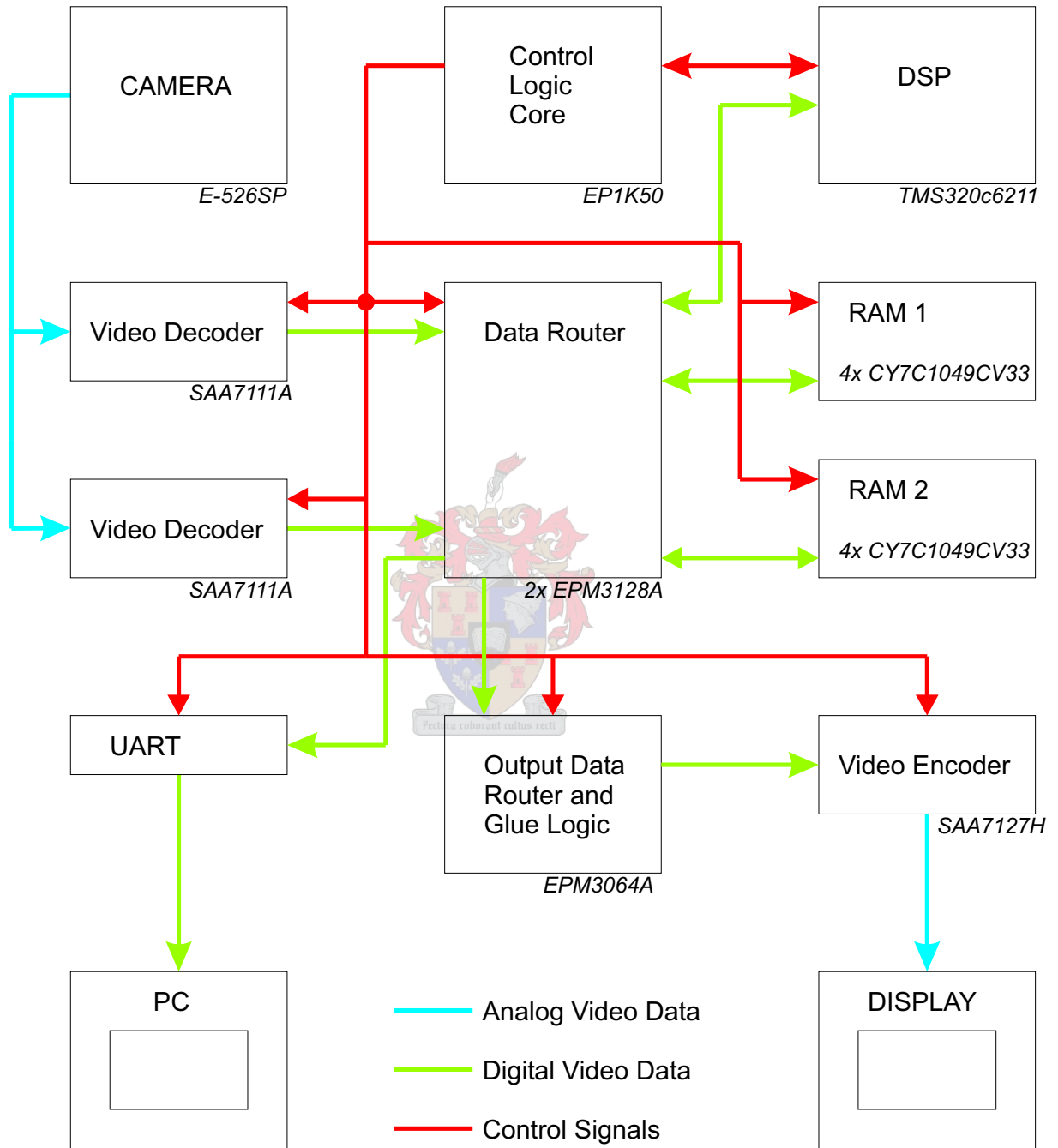


Figure 3.5: Video Processor Block Diagram

to provide a fourth spectrum. The complexities of adding a second camera to gain an additional spectral input outweighed the advantages of having the extra spectral region to study. Therefore, it was decided to use a single colour camera and use the inherent red, green and blue spectral separation as the three spectral regions of choice. All theory could be successfully applied and *visualized* in three spectra. Extraneous filters were also unnecessary.

Many diverse digital mini- and board cameras are available. The inherent problem in using a digital board camera is the output data format. Almost all digital board cameras output data in a CCIR compatible  $C_b$ -Y- $C_r$  4:2:2 or similar format. This means that chrominance values are subsampled with respect to the luminance values. This is the standard for normal video, as the human eye is more sensitive to luminance and only needs ancillary chrominance information to recreate an adequate image. For this project's application it was unacceptable as this implied that two of the input spectra (in YUV) had half the resolution of the third. From Equation 2.2, this also implies that the RGB input resolution would be half the resolution of the grayscale / luminance input signal.

Some standard colour webcams (Creative Webcam 3, Creative Webcam 5, Logitech webcam) were considered as they were freely available at low cost. Problems with webcams though were that their images are generally noisy, their integration times slow and output formats fixed. An Omnivision Evaluation USB2.0 Camera was also considered but rejected because of its high cost. USB cameras are useful if algorithms were implemented solely on a PC, as they eliminate the need for external hardware.

Therefore an analog camera, the Eagle Technology E-526SP, was chosen as it was readily available at similar cost to the digital board cameras. Having an analog input also held the added advantage that any device capable of an analog video signal could be used as an input. The advantage of this approach is that data could be gathered using any video camera, and experiments repeated. Also, using an external video A/D in the video processor any desired digital format could be produced.

The camera was mounted on a bracket attached to a standard tripod, providing 3 degrees of movement, adequate stability and accurate adjustment of camera angle (Figure 3.6).

The E-526SP camera provides an interlaced 25 frames per second and a resolution of 330 TV lines with a S/N ratio of more than 48dB [9]. The camera uses a 1/4 inch colour RJ2421 Sharp CCD with approximately 320000 pixels, 512(H)x582(V) [27]. It was fitted with a 16mm lens providing a viewing angle of 13°.



**Figure 3.6:** *E-526SP Camera and Tripod*

### 3.2.2 Analog Video Decoder

The demonstration application will use an analog PAL format as input and output and therefore requires a video A/D and D/A. Many single IC video input processors are available to convert analog video signals to a digital stream. Three decoders were considered, as listed in Table 3.1

Manufacturer	Part Number	Description
Phillips	SAA7111	Enhanced Video Input Processor
Phillips	SAA7113	9-Bit Video Input Processor
Analog Devices	ADV 7183	Advanced Video Decoder with 10-Bit ADC

**Table 3.1:** *Analog Video Decoders*

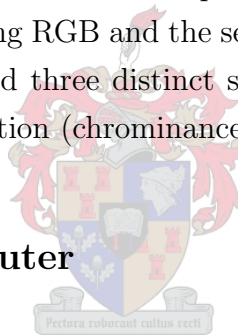
The Phillips SAA7113 is a high end video converter, automatically detecting input field frequency and video standard (PAL or NTSC). It has an 8 bit output bus providing various output data formats. However all formats were YUV 4:2:2 or similar which meant that the chrominance signals had lower bit depths than luminance. The desired output was RGB (or any three linearly independent spectra) which rendered the SAA7113's output format useless for the project's application.

The Analog Devices ADV 7183 is a versatile 10bit video decoder, also capable of many different input standards and with a 16bit output data bus.

The only available video decoder capable of outputting 3 spectra of equal bit depth data was the Phillips SAA7111A video decoder, the blockdiagram shown in Figure 3.7 (reproduced from [23]). Output RGB data are multiplexed on a 16bit output data bus, accompanied by various dedicated output timing pins. The SAA7111A has four analog inputs, various luminance, chrominance and other analog video processing controls. All these control registers are accessed via an I<sup>2</sup>C bus interface. The SAA7111A required minimal external components and was used as per the datasheet. The SAA7111A also has a clock output pin, providing an output clock at 27MHz, synchronized with the video data. This clock was used as the board master clock, to ensure synchronicity between all components.

Initially, a multi-sensor approach was considered and two SAA7111A IC's were used, one for each camera. One SAA7111A would extract RGB data from the first camera, and the second SAA7111A would extract luminance from the second camera covered by an infra-red filter. Due to the complications of synchronizing analog cameras, it was decided only to use a single camera. The output of this camera was then tied to both video decoders, the first providing RGB and the second only luminance and chrominance. Although this still only provided three distinct spectra, it provided the grayscale value (luminance) and colour information (chrominance) at no extra processing cost.

### 3.2.3 Digital Data Router



Dataflow through the video processor is an intricate matter where timing is critical and data flow direction and routing variable. As can be seen in Figure 3.5 different data were required at various points in the block diagram at different times.

Lattice Semiconductor manufacture a range of dedicated signal routers, the `ispGDX` family of IC's. These IC's are available with up to 160 I/O's, are fast, with input to output signal delays and clock to output delays of 5ns. The architecture of these devices consist of a series of programmable I/O cells, interconnected by a global routing pool. The disadvantage of the `ispGDX` family is that the global routing pool only allows for unidirectional data transfer.

Another option was to use multiple 32bit databus transceivers, such as the TI Widebus family of transceivers. This procedure requires many discrete IC's, as a different transceiver is needed for every datapath. When a certain datapath was active all other transceivers would be in HI Z.

The other alternative to the `ispGDX` family or discrete transceivers for signal routing

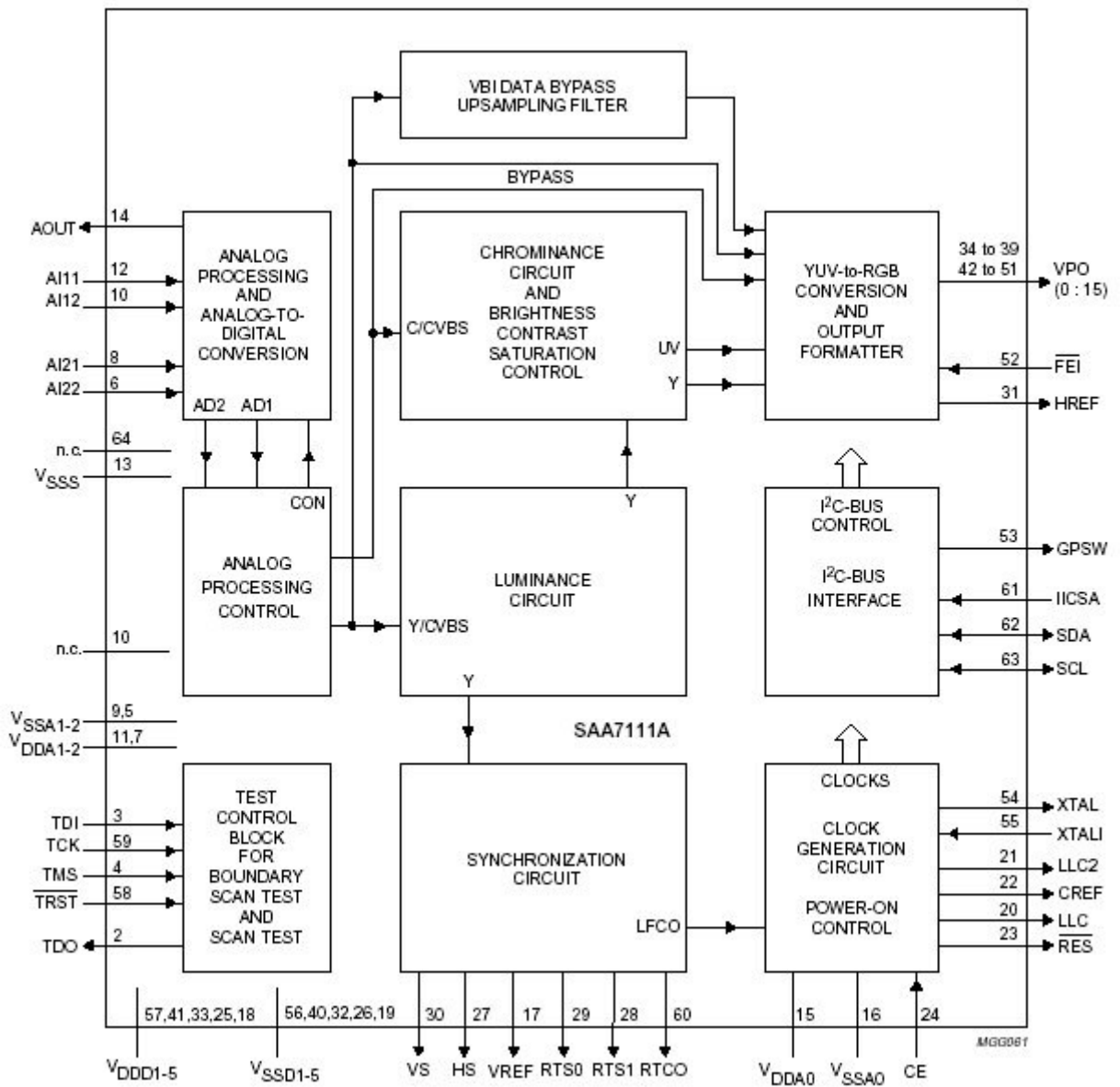


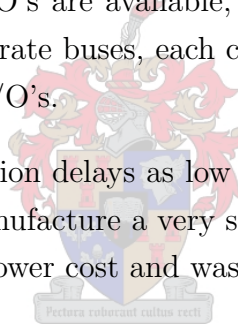
Figure 3.7: SAA7111 Block Diagram

was to use a Complex Programmable Logic Device, or CPLD. CPLD's are fast, non-volatile, easily available and very versatile. The CPLD approach was chosen as CPLD's are capable of implementing logic and handling bi-directional data which the Lattice signal routers and transceivers were not.

Although data timing is considered in more detail in Chapter 4, timing needs to be considered as a requirement for the choice of data routers. Digital video data streams from the video decoder at 13.5MHz. This translates to a period of 74ns in which all data writing/routing needs to occur before the next data becomes available. From Figure 3.4 it can be seen that during this period two different data transmissions occur, which leaves about 37ns for each transmission. This means the input to output delays of all devices in the data path must be less than 37ns.

Also, the number of I/O's needed had to be taken into account. Considering a 32bit data bus (4 x 8bit for each colour), 160 I/O pins are needed for data, plus I/O pins to transmit control information. This is more than most dedicated signal routers or CPLD's offer. Devices with so many I/O's are available, but at high cost. The solution was to split the data bus into two separate buses, each containing 2 colours / 16bits. Now each signal router only required 80 I/O's.

With 96 I/O's and propagation delays as low as 5ns, the Altera EPM3128A CPLD's were an ideal choice. Xilinx manufacture a very similar CPLD, the XC95144XL, but the Altera CPLD was available at lower cost and was thus the component of choice.



### 3.2.4 Control Logic

The video processor requires an enormous amount of timing and control logic to interface all different signals correctly. To this end, a logic device with many I/O's and logic cells would be required. As an Altera ACEX EP1K50QC208-2 FPGA was available at no cost, this was the component implemented in the design. The ACEX 1K family consist of 4 devices, of which the EP1K50 is the second largest. All devices in the ACEX 1K family are pin compatible, which provides versatility and comfort in design - if the device is not suitable, it is simply replaced with the one that is. The EP1K50QC208-2 has a 5k internal RAM, 147 I/O pins and a maximum clock rate of 250MHz and clock to output delay of 4ns.

As this FPGA has a volatile configuration memory, an external ROM was needed to configure it upon startup. Again, the Altera EPC2LC20 was available at no cost and the component used. The EPC2LC20 is a 1695680x1 bit serial configuration device. As

the EP1K50 has a 785000x1 bit data space, one EPC2 device was sufficient. The EPC2 connects to the EP1K50 through a four wire interface as shown in Figure A.4.

The OE, nCS, and DCLK pins supply the control signals for the address counter and the output tri-state buffer. The EPC2 configuration device sends a serial bitstream of configuration data via its DATA pin to the DATA0 input pin on the EP1K50.

The EPC2 device is programmed through its JTAG pins. No extra circuitry is needed and the EP1K50 is automatically configured after being powered up.

### 3.2.5 RAM

The RAM is an integral part of the video processor. This is where video data from the video decoder are stored, read and written from the DSP and read to the output video glue logic and encoder. RAM size is dependant on video resolution. At 640x480 resolution, there are 307200 pixels per frame per colour. This translates to 2457600 bits per frame per colour (at 8 bits per pixel) which requires a storage space of 300kB per frame per colour. Considering four colours, in excess of 1MB of RAM per frame is needed. Due to the use of the pingpong memory buffer, another 1MB for bank two is needed. This is a very large amount of RAM and is impractical to purchase in a single IC. The solution was again to create a large memory bank using smaller RAM IC's. The simplest approach was to use a RAM per colour (per bank), i.e. 8 x 300kB RAM's. 300kB is an idealized size, more practical would be 8 x 512kB RAM.

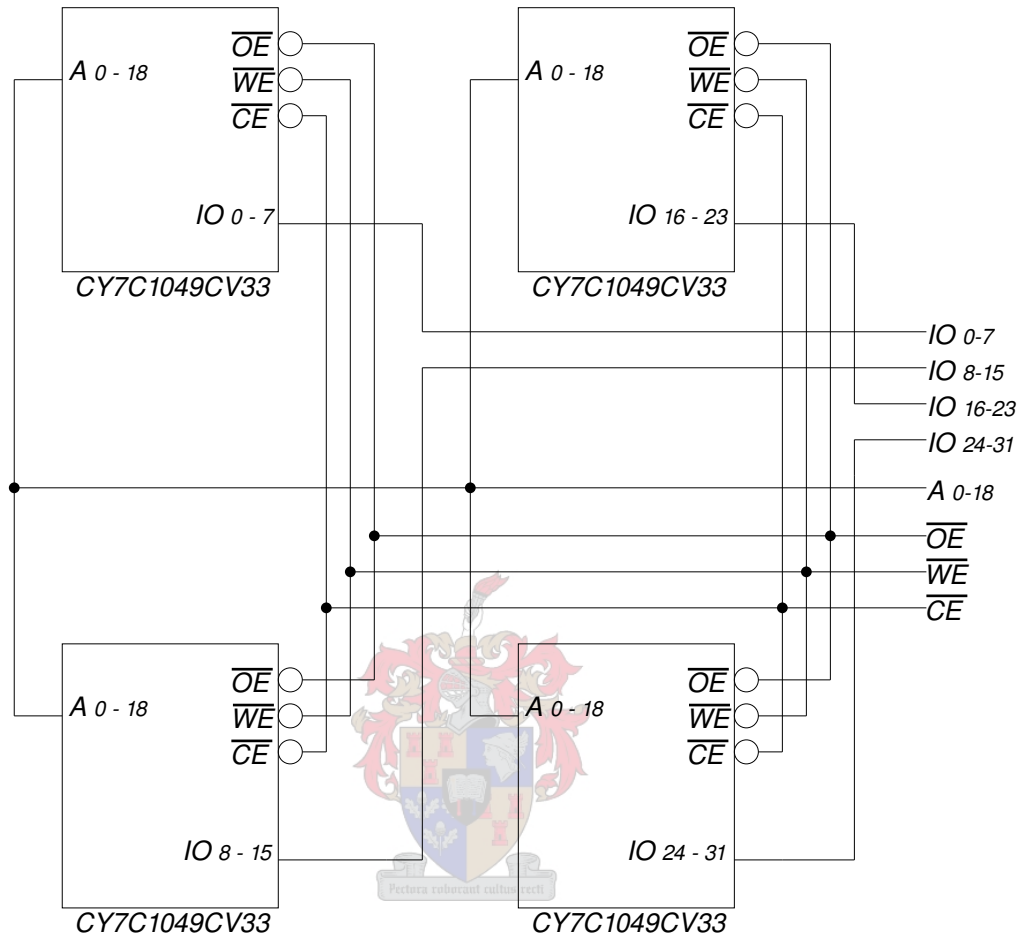
Three general families of RAM were considered, Static RAM (SRAM), Dynamic RAM (DRAM) and Synchronous DRAM (SDRAM) (Table 3.2).

RAM Type	Advantages	Disadvantages
SRAM	Fast, low power consumption	Expensive
DRAM	Cheap, small	Complex Interface, Data needs to be refreshed continually
SDRAM	Faster than conventional DRAM	Complex Interface

**Table 3.2:** *RAM Comparison*

The ideal RAM was SRAM, but would be very expensive as large amounts of RAM were needed. Fortunately a sample of 10 Cypress CYC1049CV33 RAM's were received at no cost. These are 512K x 8 Static RAM's with access times of less than 10ns and were implemented in the design.

Figure 3.8 illustrates how a large RAM was created using the smaller 512kB RAM's. All the control lines ( $\overline{CE}$ ,  $\overline{WE}$ ,  $\overline{OE}$ ) as well as address lines ( $A_{0-18}$ ) were tied together<sup>2</sup>. All the I/O lines were added in parallel, creating the desired 32bit data bus.



**Figure 3.8:** *The video processor RAM*

The RAM block in Figure 3.8 was duplicated, to create the pingpong memory structure.

### 3.2.6 Output Video Logic

Data read from RAM need to be formatted correctly and accompanied by the appropriate timing signals so as to be encoded into an analog video stream. To this end a small CPLD was needed to format the data and generate the video timing signals. As very few I/Os

---

<sup>2</sup>This was a necessity due to the limited amount of control lines available on the EP1K50 FPGA. A more versatile approach would have been to have separate control and data lines, so as to address each colour separately.



were needed (8bits per RAM bank, 8bits to the output video encoder and some control lines) the Altera EPM3064A was chosen. This is the same family as the EPM3128A, used for the input video signal routing and control logic. The EPM3064A has 34 I/O's with a 4ns propagation delay.

### 3.2.7 Output Analog Video Encoder

To display the tracking results on an analog television the digital video stream needed to be converted back to an analog TV signal. Two different video encoders were considered, the Analog Devices ADV 7170 and the Phillips SAA7127H.

The ICs are very similar, both input ports accept CCIR 656  $C_b$ -Y- $C_r$  data streams with 720 active pixels per line in 4:2:2 multiplexed formats, both can operate as master or slave (i.e., respond to embedded timing signals in the data, or respond to timing signals generated by an external device) and both provide simultaneous CVBS, red, green and blue video output. However, neither accepted raw RGB data as input, therefore, to display output video in colour both devices would require the output video logic device to convert the digital video stream to an  $C_b$ -Y- $C_r$  data stream. Both devices also require 720 active pixels per line - as the video resolution was 640x480 this implied that a blank space would occur to the right of the video on the screen. Ideally a device configurable to 640 active pixels per line would be used.

As the devices were so similar, the Phillips SAA7127H was chosen, because it was available and had similar control registers to the SAA7111A (for input video decoding) thus simplifying the control register setup.

### 3.2.8 External Interfaces

Various interfaces to external devices (other than the DSP) exist on the video processor, and are discussed in this section.

#### JTAG

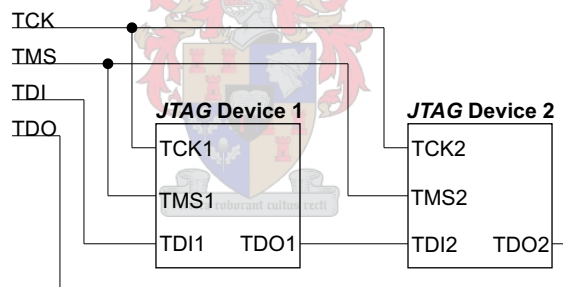
In the 1980s, the Joint Test Action Group (JTAG) developed a protocol for boundary-scan testing that was later standardized the IEEE Std. 1149.1 specification [4]. This boundary-scan test architecture offers the capability of efficient test and in-system programming of components on a PCB with tight lead spacing.

A device operating in JTAG mode uses four required pins, TDI, TDO, TMS, TCK and one optional pin, TRST. These are summarized in Table 3.3.

Pin	Description	Function
TDI	Test data input	Serial input pin for instructions and test and programming data
TDO	Test data output	Serial output pin for instructions and test and programming data
TMS	Test mode select	Input control pin for JTAG Test Access Port controller state machine
TCK	Test clock input	Clock input to BST circuitry
TRST	Test reset input	Asynchronous-boundary scan reset

**Table 3.3:** *JTAG Pin Descriptions*

Different JTAG devices are cascaded on the JTAG bus as shown in Figure 3.9. The TMS and TCK signals are all connected in parallel, whereas the TDO of the first device connects to the TDI of the second, and so forth. TRST was not used in the design.



**Figure 3.9:** *Cascading JTAG Devices*

Seven different devices with boundary-scan capabilities appear on the video processor, the Phillips SAA7111A (x2), Phillips SAA7127H, Altera EPM3128A (x2), Altera EPM3064A and Altera EPC2LC20. As JTAG communication software was dependant on the manufacturer, all Phillips devices were cascaded on a completely separate JTAG bus to the Altera devices.

The JTAG protocol requires pullup and pulldown resistors on the output JTAG ports. Some confusion exists as to the exact number and placing of these resistors as different datasheets suggest different pullup resistor configurations. The Altera and Phillips JTAG ports are shown in Figure A.6.

## Interface to DSP

The TMS320C6211 DSK (DSP Starter Kit) uses a SAMTEC 40x2 pin 0.050" x 0.050" header for interfacing to a daughterboard [30]. The video processor was designed with a 40x2 external port as closely approximating the DSK header as possible and is shown in Figure A.6. The two pinouts were not identical however, and a small connector PCB with the appropriate mating connectors was designed to connect the DSK with the video processor. The schematic is shown in Figure A.9 and the PCB in Figure A.12.

## UART

For serial communication to a PC a UART was implemented in the data routing CPLD's. This is discussed in Section 4.4.3. A Maxim MAX3232 3.3V RS-232 transceiver was used to convert logic levels to RS-232 levels. Data was read into MATLAB, processed and displayed, as shown in Appendix C.5.

## External Port

A 4 pin dip switch, as well as a momentary tactile switch, was used for user input to the video processor. These input devices were connected to the EP1K50 FPGA and their function programmed in software. The external switch was capacitively debounced and the dip switches pulled high when off. The tactile switch was used to initiate a serial RAM dump to PC, and the dip switch selected the RAM bank and colour. Also, on every PLD, a few extra pins were routed to an external header. This was used in debugging the video processor.

### 3.2.9 Power Supply

Various supply voltages were required in the video processor, 3.3V for the video decoders and encoders, 3.3V and 2.5V for most logic IC's and 5V for the external interfaces and DSP. Total current consumption was calculated to be under 1A, therefore the LM7805 linear 5V power regulator was chosen to provide the 5V reference. The LM7805 is simple, readily available and cheap. The 3.3V and 2.5V supplies were obtained by using low dropout voltage regulators (the Texas Instruments REG104-33 and REG104-25 respectively). As the voltage drop between the 5V and 3.3V/2.5V supplies is low, there would be no significant power loss. A more efficient way to generate a 5V supply from a variable input voltage would have been to use a switching supply, such as the single IC Linear

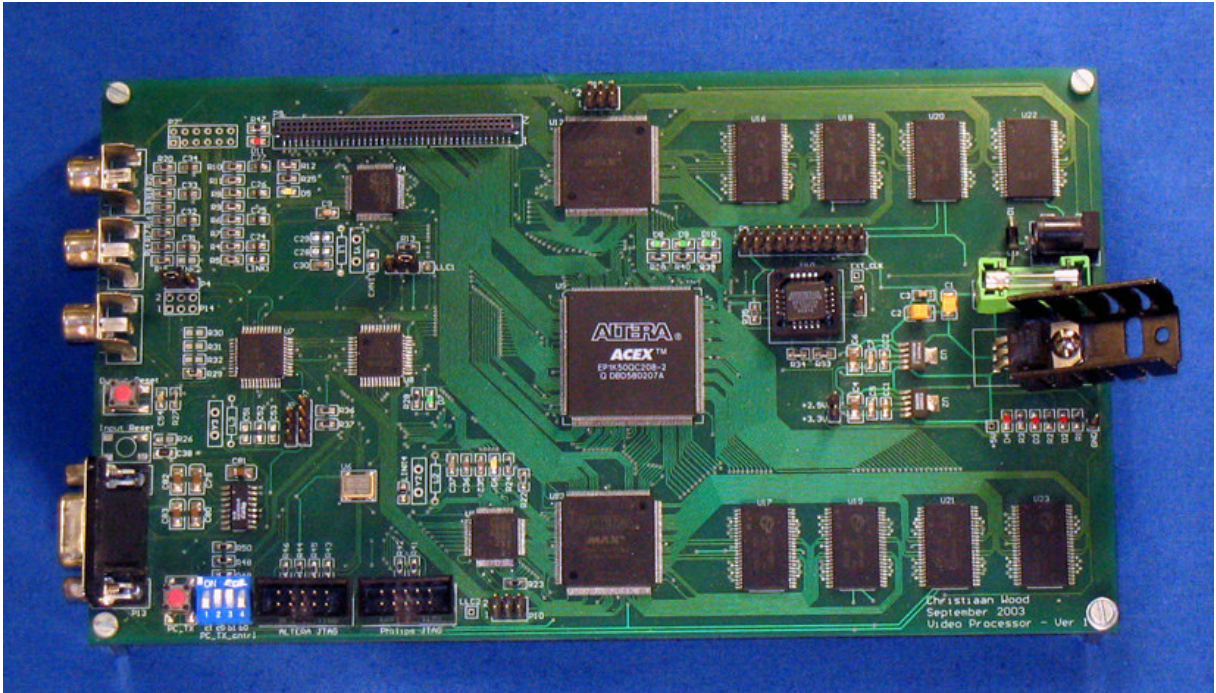
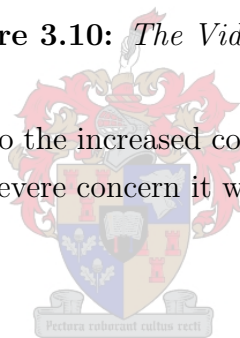


Figure 3.10: *The Video Processor*

Technologies LM2594HV. Due to the increased complexity of this approach, and also the fact that power loss was not a severe concern it was decided to use a linear supply.

### 3.2.10 PCB Design



Discussing the PCB design in detail would be superfluous, but some comments are in order. To keep costs at a minimum it was decided to implement the video processor on a two-layer PCB. As the highest frequency on the board would be 27MHz, this was plausible, but datapaths needed to be kept as short as possible. From Figure 3.10 and also Figure 4.1, the analog video inputs are the top two RCA plugs and output the bottom RCA plug. Input video would be converted to four colours of digital data. From here two colours travelled to the top datarouter and two to the bottom datarouter. The large IC in the centre is the EP1K50 control FPGA and the RAMs are the four ICs on the top and bottom right respectively. Analog signals were kept as short as possible and also kept to the left of the video processor board in Figure 3.10, digital signals kept to the right. The analog and digital grounds were kept separate, connected in at one point with a zero ohm link.

# Chapter 4

## Video Processor Software Design

In this chapter the operation and software of the video processor is discussed in detail. Various devices require some form of software:

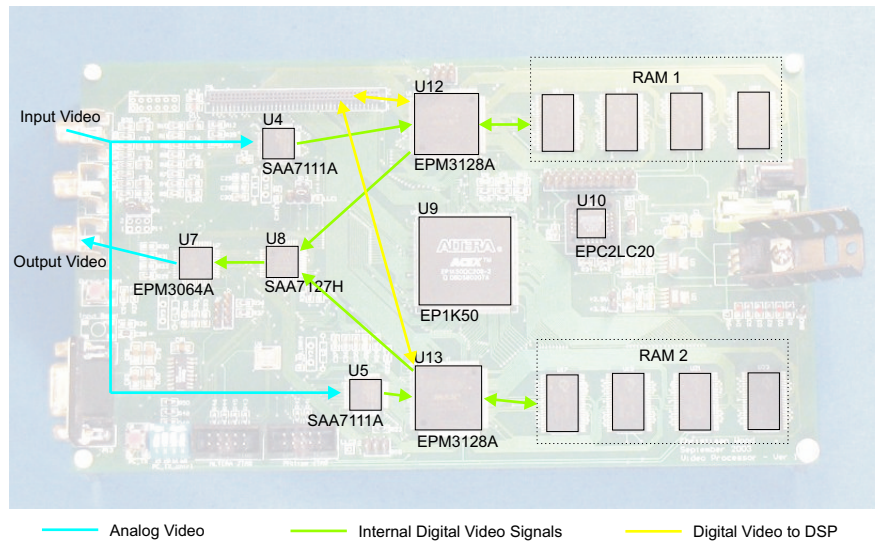
- the SAA7111A and SAA7127H (video encoders and decoders) control registers needed to be initialized through I<sup>2</sup>C.
- the EPM3128A signal routers needed to be programmed
- the EPM3064A output signal router and glue logic needed to be configured
- the EP1K50 FPGA control logic IC needed to be programmed to oversee all the above functions and ensure correct timing.

These devices are illustrated in Figure 4.1, along with the different video signals.

This chapter initiates the software discussion with the video decoders (U4 and U5) and encoder (U7) setup. This was done through an I<sup>2</sup>C kernel in the FPGA (U9) reading initialization data from an internal ROM. Firstly the implementation of the ROM in the FPGA is discussed, then some background the I<sup>2</sup>C protocol is given and thereafter the I<sup>2</sup>C kernel discussed.

After the video decoder setup, digital video would be present on the video processor and the different datapaths and datarouters (U8, U12 and U13) for the digital video data will be discussed.

At this stage, all datapaths would be functional and discussion will proceed to the main video processor control and timing IC software (U9).



**Figure 4.1:** *Video Processor Data Paths*

## 4.1 Internal ROM

As opposed to explicitly defining various initialization constants in the FPGA, it was decided to define an area in the FPGA memory as ROM and read the I<sup>2</sup>C setup data from there. The EP1K50 FPGA possesses an internal RAM of 5k, which is ample for the required ROM.

The ROM was implemented in VHDL using the standard Quartus Library of Parameterized Modules (LPM) ROM megafunction, `lpm_rom`. A ROM component was defined as follows:

```

----- ROM component definition -----
component lpm_rom
  generic (
    lpm_width      : positive;
    lpm_widthhad   : positive;
    lpm_file        : string;

    LPM_ADDRESS_CONTROL: STRING := "UNREGISTERED";
    LPM_OUTDATA: STRING := "UNREGISTERED");

  port (
    address : in std_logic_vector(lpm_widthhad-1 downto 0);
    q       : out std_logic_vector(lpm_width-1 downto 0));
end component;
-----

```

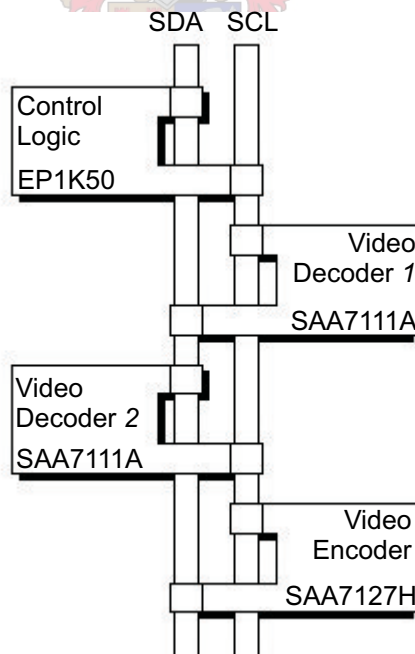
Data to be written to the ROM are stored in a memory initialization file (MIF) (see B.3). 288 Bytes of I<sup>2</sup>C configuration data is stored in this file, requiring a ROM of 512 bytes and a 9bit wide address bus.

Thus, the ROM address width was set to 9bits, data width to 8bits, the input .txt filename specified as `my_i2c.mif` and all ports mapped to VHDL signals.

## 4.2 I<sup>2</sup>C Protocol

All the video encoders and decoders in the video processor required to be initialized through setting various control registers. A standard serial protocol, I<sup>2</sup>C, was required to transmit these control bytes and a brief background on this protocol is offered, before discussion of the design of the I<sup>2</sup>C kernel commences.

The I<sup>2</sup>C (an acronym for Inter-IC bus) bus is a two wire inter IC communication bus conceived in the early 1980's by Phillips as an alternative to the then standard byte-wide communication buses. The two wire protocol greatly simplifies board outlay and provides a standard for all manufacturers.



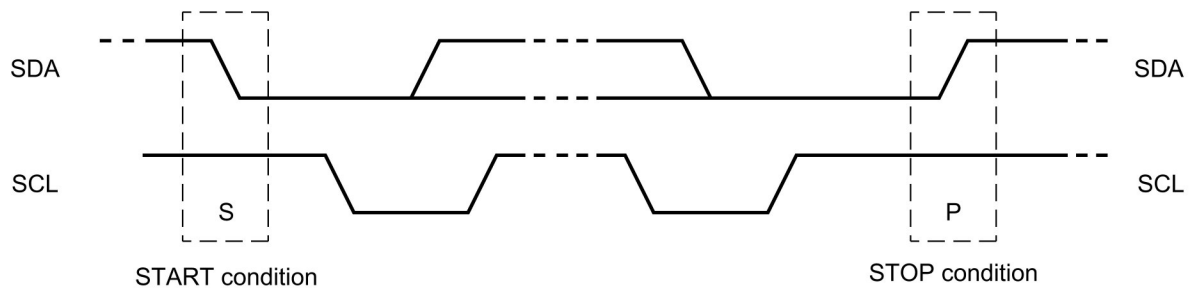
**Figure 4.2:** I<sup>2</sup>C bus setup

The bus physically consists of two active, bidirectional lines, the SDA (Serial Data Line) and SCL (Serial Clock Line) and also a ground connection. Every device on the bus has its own unique address, and can act as a receiver or transmitter, depending on its

functionality. The IC that issues commands is known as the bus master and the other ICs as the slave devices. There may be more than one bus master, but in our case both the video encoders and decoders only receive information, therefore only a single bus master is required. As no discrete I<sup>2</sup>C device was used, the I<sup>2</sup>C kernel was implemented in VHDL in the EP1k50 FPGA as the designated bus master.

Figure 4.2 illustrates the I<sup>2</sup>C bus on the video processor. The EP1K50 FPGA acts as the bus master issuing all I<sup>2</sup>C commands and the video decoders and encoders act as slaves, only receiving commands.

An I<sup>2</sup>C bus communication is initiated by the bus master issuing a START condition, shown in Figure 4.3.



**Figure 4.3:** *I<sup>2</sup>C bus START and STOP conditions*

The master then transmits the address of the device it wants to address and each IC on the bus compares this address with its own. If it does not match, the IC simply waits for the bus to be released by the STOP condition (Figure 4.3[24]). However, if the address does match the IC responds by pulling the SDA line low on the master ACKNOWLEDGE signal. Now the master can start transmitting data on the bus, Figure 4.4 [24]. The bus is released when the master device issues the STOP condition, and all devices then wait for the next START.

As mentioned earlier, the I<sup>2</sup>C bus consists of two active lines and a ground. Internally, the IC's SDA and SCL pins look like Figure 4.5.

The bus interface is built around an input buffer and a open drain or open collector architecture. Both lines are connected to the positive supply by pull up resistors<sup>1</sup> leaving them at a logic HIGH when the bus is free. This was implemented in VHDL using a tri-state buffer, pulling the line low to assert a 0, or setting it to HI Z.

The I<sup>2</sup>C bus can run at various clock rates, 100 kbit/s in the Standard-mode, up to

<sup>1</sup>Taken as 4k7, from the I<sup>2</sup>C Bus Specification [24]



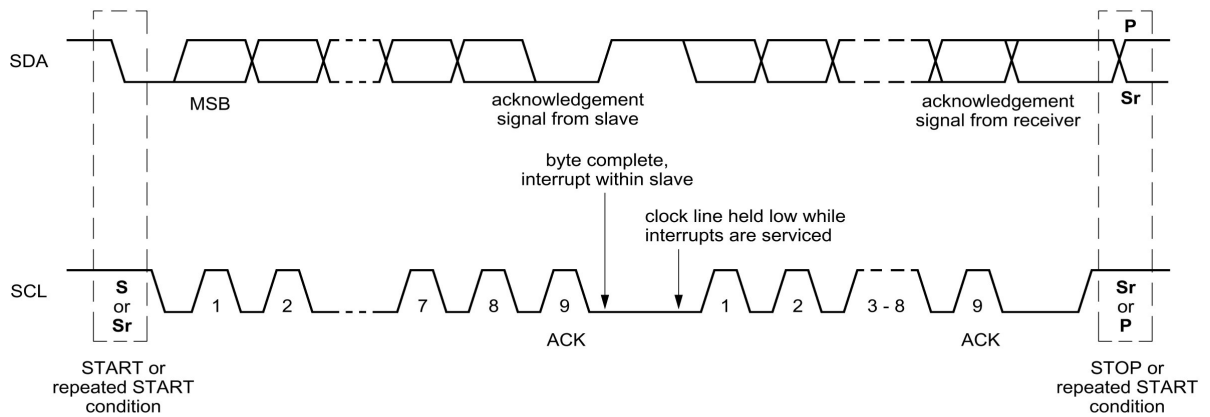


Figure 4.4:  $I^2C$  bus Data Transfer

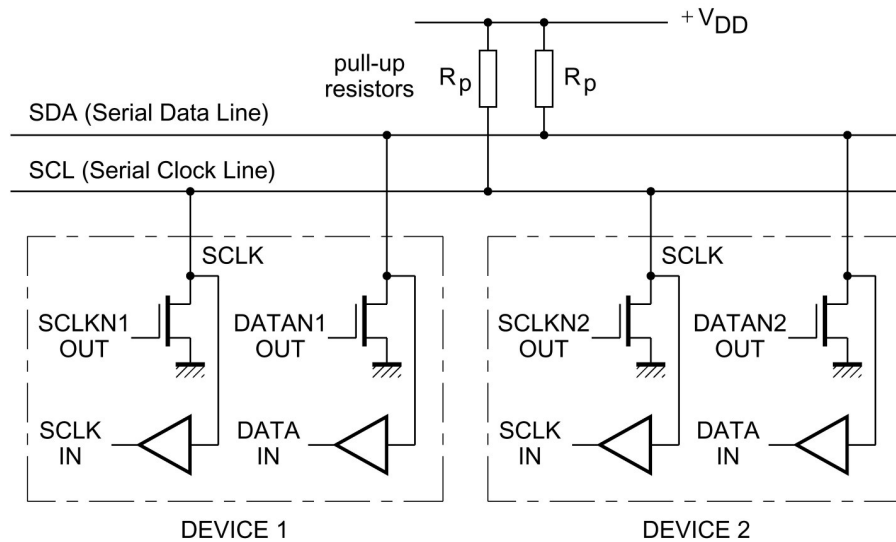


Figure 4.5:  $I^2C$  IC Hardware

400 kbit/s in the Fast-mode, or up to 3.4 Mbit/s in the High-speed mode. As  $I^2C$  was only used for IC setup speed was not critical and the SCL clock rate was set at 100kbit/s.

### 4.3 $I^2C$ Configuration

Figure 4.6 shows the flow diagram of the VHDL process initializing the video decoders and encoders. All data to be written to these IC's were stored in an internal ROM (Section 4.1) programmed in the FPGA.

The VHDL  $I^2C$  kernel waits for the  $\overline{RES}$  line to go HI (indicating that the internal reset has been completed) before initiating  $I^2C$  transactions. Data transfers are initiated

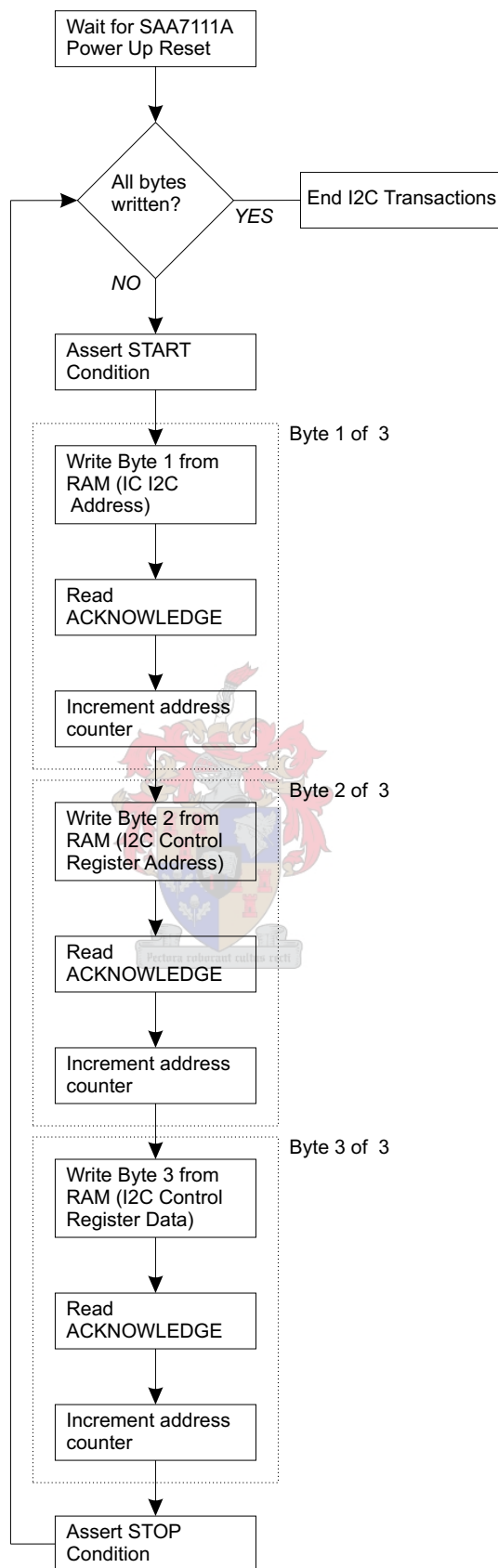


Figure 4.6: I<sup>2</sup>C Software Flow Diagram


by the START condition whereafter bytes are written in groups of three before the next STOP. The three bytes are (in order):

- Slave I<sup>2</sup>C Address
- Data Byte 1 - Control Register Subaddress
- Data Byte 2 - Control Register Value

Data were setup in the ROM in the above fashion, so sequential reads would write data in the correct order. Here is an excerpt from the ROM file:

```
-----
-- sub address 0BH      Luminance contrast
27      :      01001010;
28      :      00001011;
29      :      01000111;

-- sub address 0CH      Chroma saturation
30      :      01001010;
31      :      00001100;
32      :      01000000;
-----
```



The first byte (of the three per group) written (number 27 in this excerpt, stored in the 27th location in ROM) is "01001010". This is 4AH and is the I<sup>2</sup>C slave address of the SAA7111A for writing data [23]. The slave address can be toggled between 4AH and 48H by setting the I<sup>2</sup>C-bus slave address select pin (IICSA) to either HI or LOW. The two SAA7111A video decoders were setup with different slave addresses, so as to enable them to be configured individually. The second byte written (number 28), "00001011", is the SAA7111A control register subaddress (in this case, to control luminance contrast) and "01000111" (number 29) the value written to this register. The FPGA sequentially reads through the entire ROM and in doing so initially configures the first video decoder, then the output video encoder and finally the second video decoder.

All control registers need to be given a value, even if it was identical to the initial value. Table 4.1 shows the control register setup for the video decoder outputting the RGB signal. All registers kept at their default value were omitted from Table 4.1 and were mostly concerned with analog video control. These values are listed in Table B.1.

Subaddress (HEX)	Function	Value	Description
08H	Sync Control	10001000	Sets field to 50Hz, 625 lines
10H	Format/Delay Control	01000000	Sets output to RGB
11H	Output Control 1	10001100	Activates LED, sets VREF pin to vertical frame sync
12H	Output Control 2	00001001	Sets output to 3x 8bit RGB

**Table 4.1:** *SAA7111A For RGB Output Control Registers*

The second video decoder was setup to provide a grayscale output from the same input. Table 4.2 shows the control register setup for the second SAA7111A (only showing values differing from the first table).

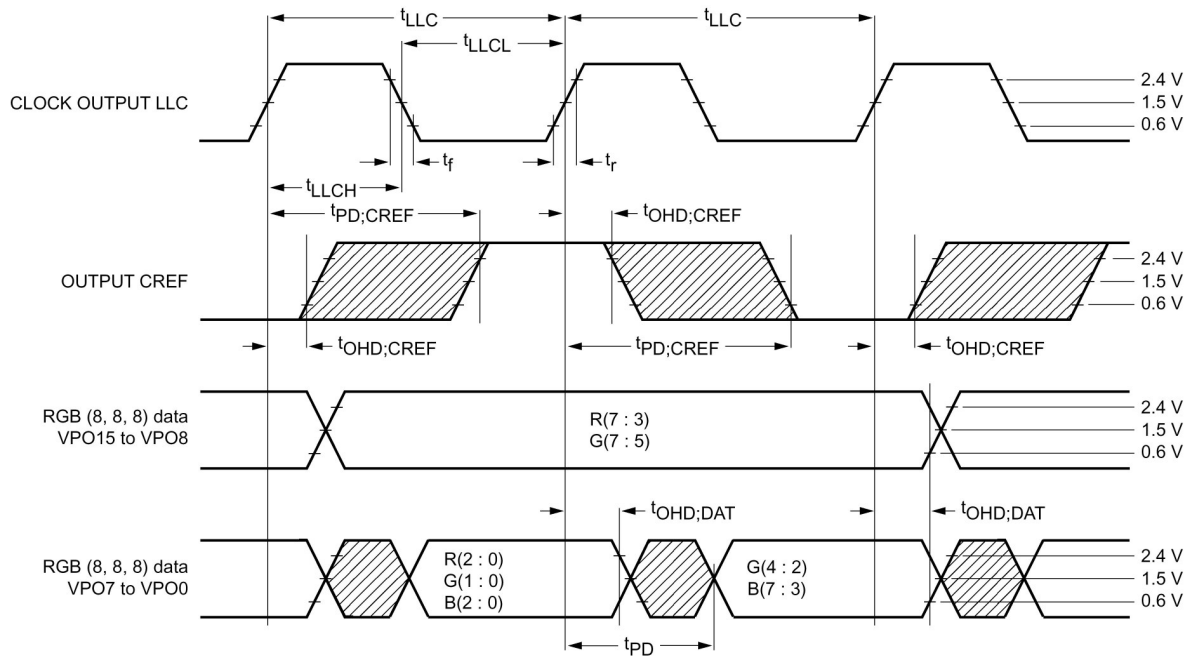
Subaddress (HEX)	Function	Value	Description
10H	Format/Delay Control	01000000	Sets output to YUV 4:2:2 16bits

**Table 4.2:** *SAA7111A For Grayscale Output Control Register*

The output of the second video decoder was set to provide a YUV 4:2:2 signal of 16bits. This implies an 8bit luminance signal, and multiplexed 8bit subsampled red and blue chrominance signals. As the intensity (luminance) was all that was needed from this video decoder, the colour subsampling was irrelevant. This IC provided a grayscale output of the same resolution corresponding with the RGB output of the first video decoder.

The SAA7111A video decoders now offered a digital video stream in the format desired, characterized in Figures 4.7, 4.8 and 4.9 (reproduced from [23]). The inherent timing signals in the analog CVBS video stream have been replaced by two discrete synchronization signals, VREF (vertical frame sync) and HREF (horizontal line sync). Figure 4.7 illustrates the clock and data timing on the video decoder output bus (VPO bus, or Video Port Out bus). 24bit RGB video data are multiplexed on the 16bit VPO bus in the manner described in Table 4.3. The CREF output is a clock reference output at half the main clock frequency and in phase with the output data. Valid data are read from the VPO bus on every CREF edge. For every line the first valid data are available in the first CREF rising edge after HREF goes high. Figures 4.8 and 4.9 illustrates the vertical timing, and the status of the VREF pin with the respect to HREF. The exact position of the VREF pulse is programmable in the SAA7111A control registers 10H and 13H, but the VREF signal is only important in that it specifies the begin/end of a frame - the HREF signal is used for exact data timing. Another important signal is RTS0, the

odd/even field identification bit. When RTS0 is HI the current data belongs to the first field and when RTS0 is LOW the data corresponds to the second field of the interlaced frame.

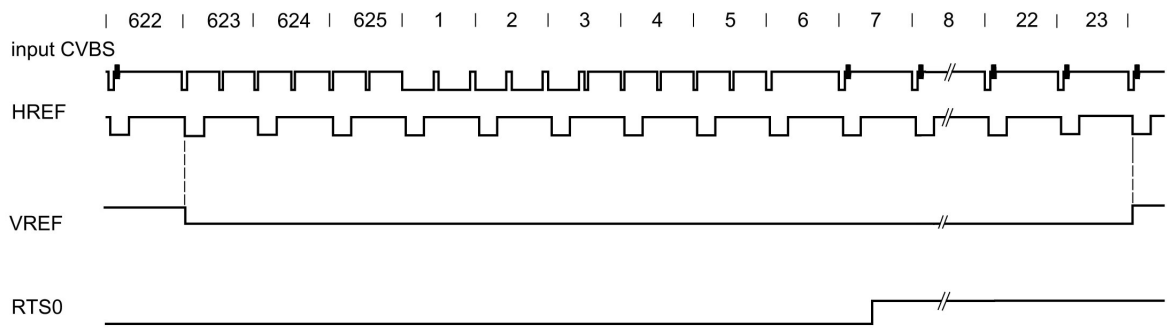


**Figure 4.7:** Clock/data timing for RGB (8, 8 and 8) output format.

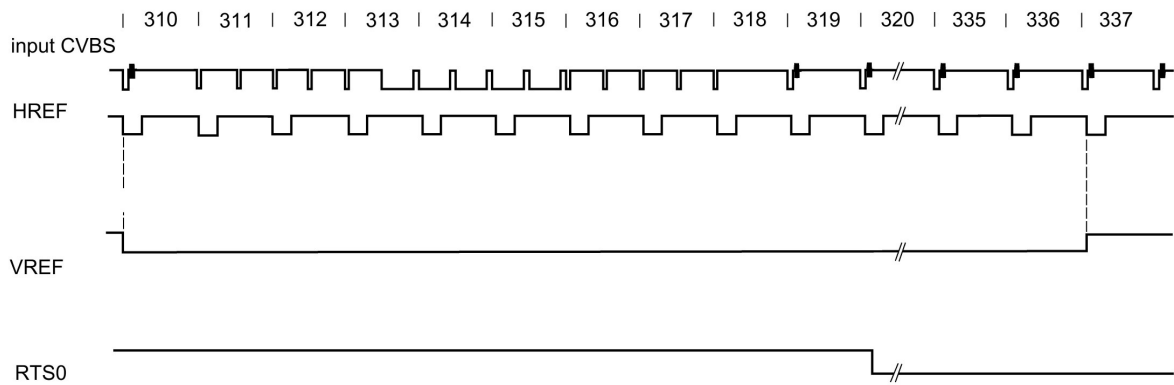


VPO	RGB 24bit byte 1	RGB 24bit byte 2
VPO15	R7	R7
VPO14	R6	R6
VPO13	R5	R5
VPO12	R4	R4
VPO11	R3	R3
VPO10	G7	G7
VPO9	G6	G6
VPO8	G5	G5
VPO7	R2	G4
VPO6	R1	G3
VPO5	R0	G2
VPO4	G1	B7
VPO3	G0	B6
VPO2	B2	B5
VPO1	B1	B4
VPO0	B0	B3

**Table 4.3:** *Output Format of the VPO bus*



**Figure 4.8:** *Vertical Timing Diagram - First Field*



**Figure 4.9:** *Vertical Timing Diagram - Second Field*



The control register setup for the SAA7127H output video encoder is shown in Table 4.4 (again only the values differing from the default.) The full table is shown in Table B.2. Output data format is discussed in more detail in Section 4.4.4.

Subaddress (HEX)	Function	Value	Description
3AH	Input port control 1	00000011	Horizontal and Vertical Triggers taken from RCV1 and RCV2 pins

**Table 4.4:** *SAA7127H Control Register*

With this understanding of the digital video signals present in the video processor, how these are organized by the data routers can now be discussed.

## 4.4 Signal Routers

Three different signal routers exist in the video processor:

- The EPM3128A (U12) data router which controls the dataflow of the red and green video streams.
- The EPM3128A (U13) data router which controls the dataflow of the blue and gray video streams.
- The EPM3064A (U8) data router which controls the dataflow of the output grayscale video.

Data routers U12 and U13 are very similar, differing only in the way the input data are latched (because they extract different colours from the multiplexed video decoder VPO bus) and in that the DSP only writes on the gray channel - therefore U13 DSP communication is bi-directional and U12 not.

The main control logic EP1K50 FPGA communicates via a 6bit control bus with the routing IC's U12 and U13. The four LSBs are communal, but the two MSBs assign an individual data router. Therefore each data router responds to a 5bit control signal, the MSB of which is unique to each. The complete control protocol is listed in Table 4.5. We will refer back to this table in the sections to come.



Control Signal	Data Flow
00000	Outputs in HI Z
00001	Camera to RAM
00010	Camera to RAM
01100	Latch 1st byte from Camera, RAM 2 to Display
01101	Latch 1st byte from Camera, RAM 1 to Display
00100	RAM 2 to DSP, RAM 1 to Display
00111	RAM 1 to DSP, RAM 2 to Display
00110	DSP to Colour 2 Space in RAM 1, RAM 1 to Display
00101	DSP to Colour 2 Space in RAM 2, RAM 2 to Display
01110	Constant (Test Value) to RAM
10000	Colour 1 data, RAM 1 to PC
10001	Colour 2 data, RAM 1 to PC
10010	Colour 1 data, RAM 2 to PC
10011	Colour 2 data, RAM 2 to PC
1x1xx	Constant (Test Value) to PC
11xxx	Constant (Test Value) to PC

**Table 4.5:** *Data Routing Control Signals*

The EP1K50 FPGA uses another control bus to communicate with the output data routing IC, U8, as this IC is completely independent to U12 and U13. A 5bit control bus is used to send commands to U8, and the protocol is listed in Table 4.6.

#### 4.4.1 CPLD U12 Data Router

The functions of U12 and U13 are twofold, to latch and demultiplex video data, and to route this data to the appropriate IC or RAM depending on a control input. As can be seen in Table 4.3, red, green and blue data are all available on the VPO bus. The byte on the middle column of Table 4.3 appears on the first rising edge of CREF (Figure 4.7) and is latched with the 01100 or 01101 command. On the next clock and CREF falling edge the second byte is latched from the VPO bus with the 00001 or 00010 command and both bytes written to the appropriate RAM. This is why two different commands are necessary for each data read from the camera - it needs to be specified which RAM is currently used for new camera data. The data router uses the two LSBs of the control signal to determine which RAM to access - "01" for RAM 1 and "10" for RAM 2. Data are routed from RAM to the DSP (i.e. DSP reads from RAM) using the 00100 and 00111

Control Signal	Output Data	Notes
00000	ZZZZZZZZ	Output in Hi Z
00001	10000000	80H, also for Cb = Cr = 128 = 80H
00010	00010000	10H
00011	00000000	00H
00100	11111111	FFH
00101	Read from RAM 1	Y, RAM 1
00110	Read from RAM 2	Y, RAM 2
11XXX	1, Control Signal(2 downto 0), 0000	read EAV / SAV data from low 3 bits
01000	11101011	Test Constant
01001	01110101	Test Constant

**Table 4.6:** *Output Data Routing Control Signals*

commands - 00100 connects RAM 2 to the DSP data bus and 00111 connects RAM 1 to the DSP data bus.

Timing diagrams illustrating these different control modes are included in Appendix B. These timing diagrams were essential in characterizing the inherent delays in the CPLD's which needed to be taken into account in designing the main timing controller.

## 4.4.2 CPLD U13 Data Router

As mentioned before, U13 requires the extra capability of a bi-directional DSP data bus, therefore two more commands were added to the command protocol. Data are routed from the DSP to RAM (i.e. DSP writes to RAM) by the commands 00110 and 00101.

## 4.4.3 UART

To facilitate error checking and check data in RAM, a software UART was implemented in each data router. An external serial clock at 115200 BAUD (the fastest serial bit rate that could be read by the PC) was supplied by the FPGA (see Section 4.5). The FPGA triggers a serial transmission with the appropriate control byte (which also specifies which RAM is read) as well as supplying the RAM address. The data router UART merely reads data from RAM into a shift register, appends a start and stop bit and shifts it out serially.

This was simulated and is shown in Appendices B.12 and B.13.

#### 4.4.4 CPLD U8 Data Router

The output data router, U8, concerns itself with routing data from either RAM 1 or RAM 2 and then arranging these data into the appropriate format for the SAA7127H video encoder to encode the data stream back to analog video. Routing the data from RAM is a simple matter - given the control signal 00101, U8 latches data from RAM 1 and given control signal 00110 U8 latches data from RAM 2.

As can be seen in Table 4.6, much of the output control is devoted to writing different constants to the output video encoder. This is because of the YUV 4:2:2 input format required by the SAA7127H, shown in Figure 4.10.

BLANKING PERIOD			TIMING REFERENCE CODE				720 PIXELS YUV 4 : 2 : 2 DATA										TIMING REFERENCE CODE			BLANKING PERIOD		
...	80	10	FF	00	00	SAV	C <sub>B</sub> 0	Y0	C <sub>R</sub> 0	Y1	C <sub>B</sub> 2	Y2	...	C <sub>R</sub> 718	Y719	FF	00	00	EAV	80	10	...

**Figure 4.10:** *YUV 4:2:2 Data Format*

80H and 10H need to be written alternately during the blanking period. A digital video line is initiated with a timing reference code, FFH, 00H, 00H and then the start of active video range (SAV) byte. The SAV byte contains information on the field, active video as well as if it is the beginning or end of the line.

The data then follows in the multiplexed YUV 4:2:2 format, firstly the blue chrominance value of pixel 0 ( $C_b$  is merely a scaled value of U), then the luminance of pixel 0, the red chrominance ( $C_r$  is again merely a scaled value of V) value of pixel 0, the luminance value of pixel 1 and then chrominance of pixel 2. The chrominance is subsampled, for every two pixels there is only chrominance information about one. The line is terminated by another timing reference code, the end of active video range (EAV) byte.

The constant bytes defined in Table 4.6 are used to generate the appropriate YUV 4:2:2 video stream. One problem remained, the SAA7127H video encoder required video in a YUV format and the video data in the video processor were RGB. This required a simple colourspace transformation, using the formulae 2.1 and 2.2 [33].

This transformation, however simple, produces processing software overhead that is unnecessary. As grayscale video (from the second video decoder) was already present in memory, this data were used as Y (luminance) information. Chrominance information

could also theoretically be extracted using equation 2.1 but this would also use valuable processing time. As a result it was decided to display a grayscale video output, by setting the  $C_b$  and  $C_r$  values illustrated in Figure 4.10 to 0.

It was initially thought that this approach (generating a CCIR 656 digital video stream from RAM data) would be the simplest way to recreate the analog video. The SAA7127H also has a slave mode, where all timing information needs to be supplied by the user and only video data supplied. This proved to be the simpler approach, as the input data stream was synchronized with the output data stream (by U9, Section 4.5). By connecting the horizontal frame sync, vertical frame sync and RTS0 signal from the input video decoder to the output video encoder this synchronization is maintained and only active video data needed to be sent to the output video encoder for data to be displayed correctly.

## 4.5 FPGA U9 Control Logic

The EP1K50 FPGA, U9, is the main logic and timing control centre of the video processor and subsequently has myriad functions, which are now discussed. Five different processes (see Appendix C.1) run in parallel,

- p0: Generates a serial clock at 115200 BAUD
- p1: Timing loop to flash LED (for debugging)
- p2: Main state machine for data timing and routing
- p3: Generates the I<sup>2</sup>C clock at 100kHz
- p4: I<sup>2</sup>C core - for I2C initializing of video decoders/encoders

Process p4 is the I<sup>2</sup>C core and was discussed in section 4.2.

Processes p0, p1 and p3 are generic clock dividers and are illustrated in Appendix C.1.

This leaves process p2, the main process and state machine synchronizing all data transfers. Process p2 is completely autonomous in that all timing signals are generated internally, but its mode of operation is set by the DSP. This is explained in Table 4.7.

A short aside is necessary here. As the digital video frequency is 13.5MHz, from Figure 3.4 it can be seen that data need to be read/written at at least double this frequency

Control Signal	Description
000	Cam to RAM1, RAM2 to Display
001	RAM1 to DSP, RAM2 to Display
010	DSP to RAM1, RAM2 to Display
100	Cam to RAM2, RAM1 to Display
101	RAM2 to DSP, RAM2 to Display
110	DSP to RAM2, RAM2 to Display

**Table 4.7:** *Video Processor Mode Control Signals*

(27Mhz, 37ns). The video processor was designed with this in mind, thus the camera, RAMs and display were all capable of data transmission at this rate. Unfortunately the external data bus (running at 150Mz) of the TMS320C6211 DSP could not write data to its internal RAM at this rate (more detail is given in Chapter 5)(An erroneous assumption that an external bus speed of 150MHz would be sufficient for data communication at 27MHz led to this error). This posed a serious problem, as no algorithms could be implemented if no data could be written to the DSP. This problem was circumvented by initially reading an entire frame to RAM. During the time the next frame was to be read from the camera, data was now written to the DSP, thereafter allowing the DSP to write to the RAM. Only after this are RAM banks switched and camera data written to a different RAM. This reduces the frame rate by a factor of 3 - to compensate, the bandwidth of laboratory tests were sufficiently reduced so as for it not to be a problem.

A 3bit control bus sets the mode of the video processor. The MSB of the control byte (the 'pingpong' bit) defines which RAM was to be written to and which RAM to be read from. The two LSBs have three states, 00 for normal video transfer (video is written to RAM and read to display, but no data transfers to or from the DSP occur), 01 for data to be read by the DSP and 10 for data to be written to RAM by the DSP (note that for both the last two modes data are still sent to display from the opposite RAM bank). The rows in Table 4.7 are in chronological order - first data from the camera are written to the RAM. Upon the next frame becoming available from the camera data are written to the DSP and as the third frame is sent from the camera data are written back to RAM by the DSP. During all these three cycles the same frame is read from the opposite RAM bank so as to ensure smoothness of video.

With this in mind, the software flow diagram of the video processor, shown in Figure 4.11, can be discussed. Process p2 begins by initializing all variables, signals and outputs to their appropriate values. As no data can be read until the video decoders are initialized, process p4 sets a flag when the I<sup>2</sup>C setup has been completed. Hereafter a process of

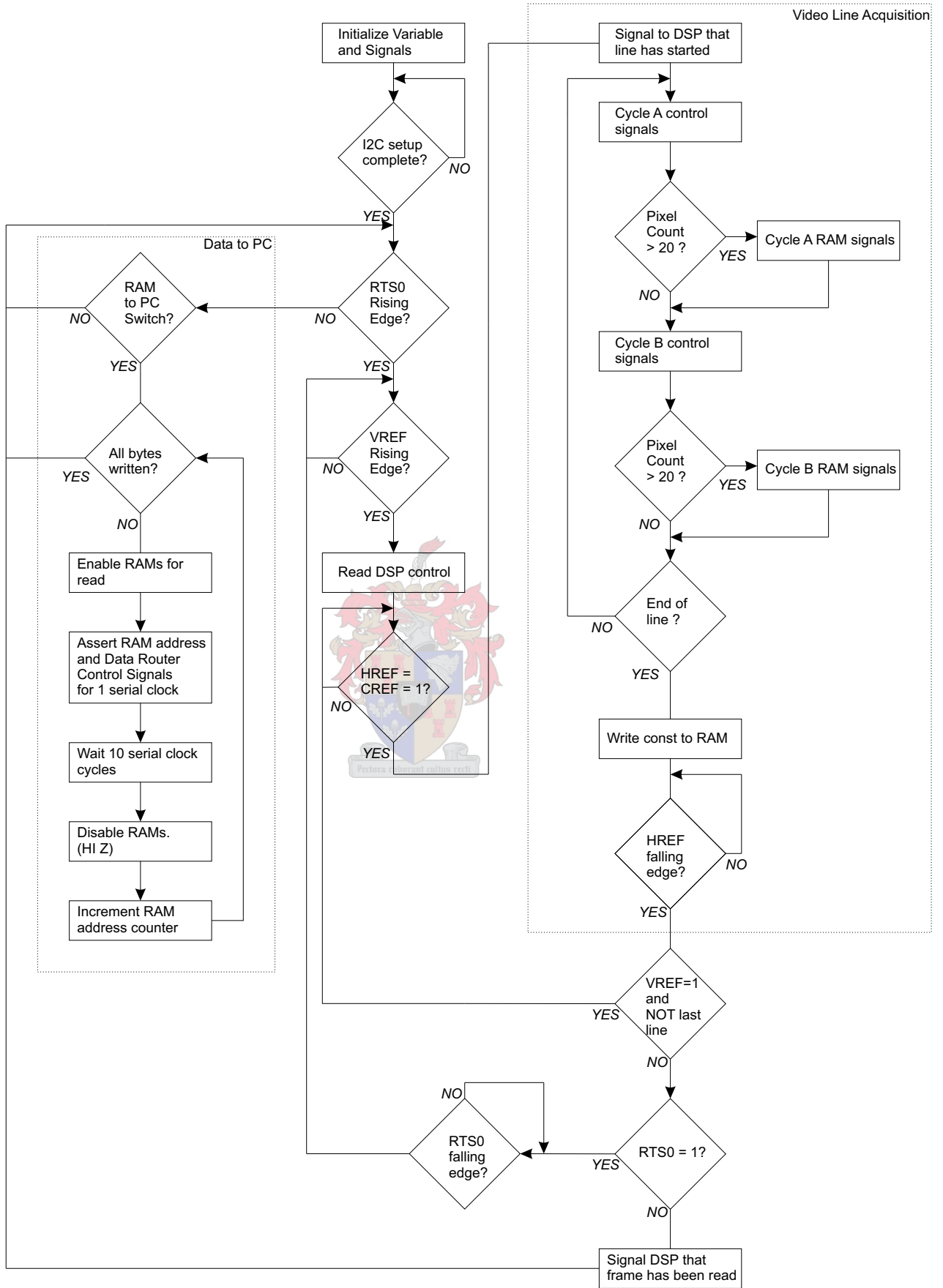


Figure 4.11: Software Flow Diagram of the Video Processor

aligning the video processor to the inherent timing of the video stream begins. The three timing signals from the SAA7111A used were RTS0 (indicating even or odd field), VREF and HREF <sup>2</sup>. Starting at the lowest frequency signal (RTS0), p2 moves inward and references VREF and then HREF <sup>3</sup>. This is achieved by continually polling the input signal and triggering a change of state upon a rising/falling edge. Firstly, RTS0 is polled for a rising edge. If no edge is detected, the external switch triggering a RAM dump to PC is polled. If unsuccessful, p2 returns to wait for a RTS0 rising edge. However, if a memory dump has been requested the RAM dump routine is initiated. Firstly the RAMs are enabled to read, thereafter the RAM addresses and data router control signals are asserted for a single serial clock cycle. As all data transmission is done by the data router, the FPGA merely waits a further 9 serial clocks until the full byte has been transmitted. The RAMs are then disabled (set to HI Z) and the address counter incremented until all data bytes have been written. Hereafter RTS0 is polled again for a rising edge.

A RTS0 rising edge indicates the start of the first field of the interlaced pattern. To align p2 with the vertical frame sync, the process waits for a VREF rising edge. When this occurs, the video processor polls the DSP control bus to determine which mode it should operate in. Now the video processor is ready to read data and waiting for a HREF rising edge. From Figure 4.7 it can be seen that HREF goes HI simultaneously with the first data becoming available from the SAA7111A and should be read on the first CREF pulse, which occurs a clock later. As there is a clock delay in the state transition within p2 and another clock delay for the data routers to respond to the command to latch camera data, reading the first pixel was impossible. The solution was to wait for HREF = CREF = 1 (as opposed to waiting for a rising edge on HREF) and ignoring the first pixel of every line. By ignoring the first pixel, the data router latches could be setup two clocks in advance, ready for the second pixel. As soon as the data latches were setup, the state machine was synchronized and no further pixels lost. This is illustrated in Figure B.3. Hereafter a signal is sent to the DSP informing it that the data acquisition process has begun - this is necessary, as the DSP can only access the RAMs during the period which a frame is streaming from the camera. Then the two-cycle acquisition loop is set in motion. Firstly the FPGA outputs the cycle A control signals (dependant on Table

---

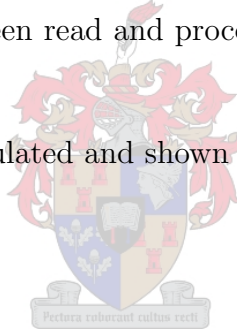
<sup>2</sup>The timing signals were taken from the video decoder decoding the RGB signals. This choice was arbitrary, as both encoders have an identical analog input and would thus output identical timing references.

<sup>3</sup>By using this approach it is ensured that the state machine is referencing the correct place in the video stream. For example, triggering a data read on a HREF signal may be invalid, as it could occur during a blanking interval.

4.7 and listed in Table 4.5) and then (if pixel no  $> 20$ <sup>4</sup>) cycle A RAM signals. The first 20 pixels of each line are still latched in the data routers and presented to RAM, but not written as the RAMs are in HI Z. On the next clock, cycle B control signals are asserted as well as (if pixel no  $> 20$ ) cycle B RAM signals. Hereafter the RAM address counters are incremented. This acquisition process continues until 639 pixels of the line have been written to RAM. The 640th pixel of every line written to RAM is a constant (this was done to arrange data serially written to the PC).

After a line has been read from the camera the state machine begins the process of extracting itself from the timing signals in reverse order. Firstly, p2 waits for the HREF falling edge to signify the end of the line, whereafter the value of VREF is read. If  $VREF = 1$ , data are still from the same field, and p2 waits for  $HREF = CREF = 1$  to read the next line. If  $VREF = 0$  or if 480 lines have been read, RTS0 is polled. If  $RTS0 = 1$  the camera is still transmitting data from the same field and p2 waits for the RTS0 falling edge to indicate the start of the next field (of the interlaced pattern) whereafter it returns to waiting for the VREF rising edge. If however  $RTS0 = 0$ , an entire frame of two fields has been read and process p2 returns to the beginning to wait for the next RTS0 rising edge.

All these scenarios were simulated and shown in Appendix B.



---

<sup>4</sup>The first 20 pixels of camera data were ignored, as they contained a dead pixel area, caused by camera hardware. See Figure 8.1. As there were 720 active pixels per line, this left 700 - more than enough for the desired 640x480 resolution.



# Chapter 5

## DSP

At this point the video processor is functional, handling data flow and writing video data to RAM. However, no processing has been done and no algorithms implemented. This is done in a digital signal processor (DSP).

### 5.1 The TI TMS320C6211 DSP

The Texas Instruments (TI) TMS320C6211 DSP starter kit (or DSK) was available at the outset of the project, and this was the DSP used. The TMS320C6000 high performance DSPs are cost efficient, with low power dissipation and are optimized for broadband networks and digitized imaging applications. The platform includes the code compatible C62x and C64x fixed point DSPs and C67x floating point DSPs.

The TMS320C6211 DSP composes one of the fixed-point DSP families in the TMS320C6000 DSP platform. The TMS320C6211 device is based on the high-performance, advanced *VelociTI<sup>TM</sup>* very-long-instruction-word (VLIW) architecture developed by Texas Instruments (TI), making this DSP an excellent choice for multichannel and multifunction applications [32].

The TMS320C6211 block diagram is shown in Figure 5.1 [31].

The TMS320C6211 is a 32bit fixed point DSP, with a clock rate of 150MHz and instruction time of 6.7ns offering performance of up to 1200 million instructions per second (MIPS). The C6211 uses a two-level cache-based architecture and has a powerful and diverse set of peripherals. The Level 1 program cache (L1P) is a 32Kbit direct mapped cache and the Level 1 data cache (L1D) is a 32Kbit 2-way set-associative cache. The Level 2 memory/cache (L2) consists of a 512Kbit memory space that is shared between

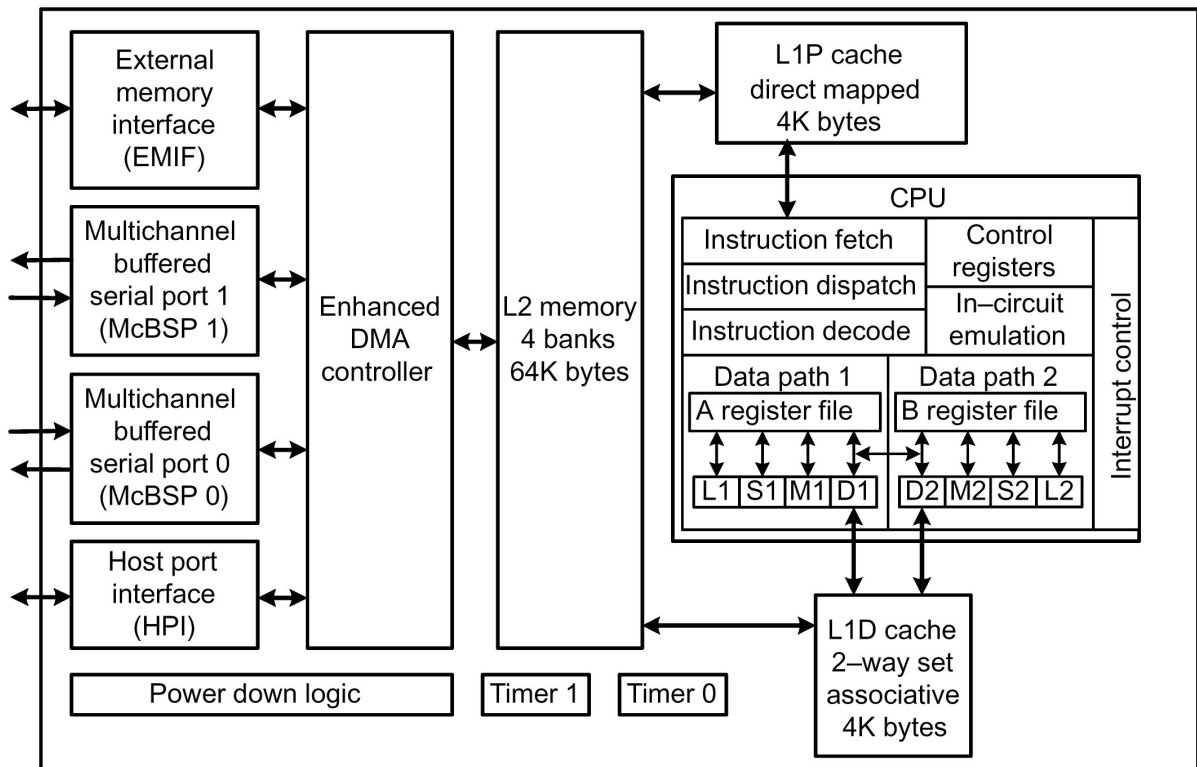


Figure 5.1: Texas Instruments TMS320C6211 Blockdiagram

program and data space. L2 memory can be configured as mapped memory, cache, or combinations of the two [32]. DSP peripherals include two multichannel buffered serial ports (McBSPc), two general purpose 32bit timers, a 16bit host port interface (HPI) and a 32bit external memory interface (EMIF) providing a glueless interface to various types of external memory.

The TMS320C6211 DSK is shown in Figure 5.2.

## 5.2 External Memory Interface

The TMS320C6211 DSP's external memory interface (EMIF) is a versatile interface, capable of a glueless interface with almost any form of memory [31]. Figure 5.3 illustrates the EMIF and all associated signals. These are described in Table 5.1.

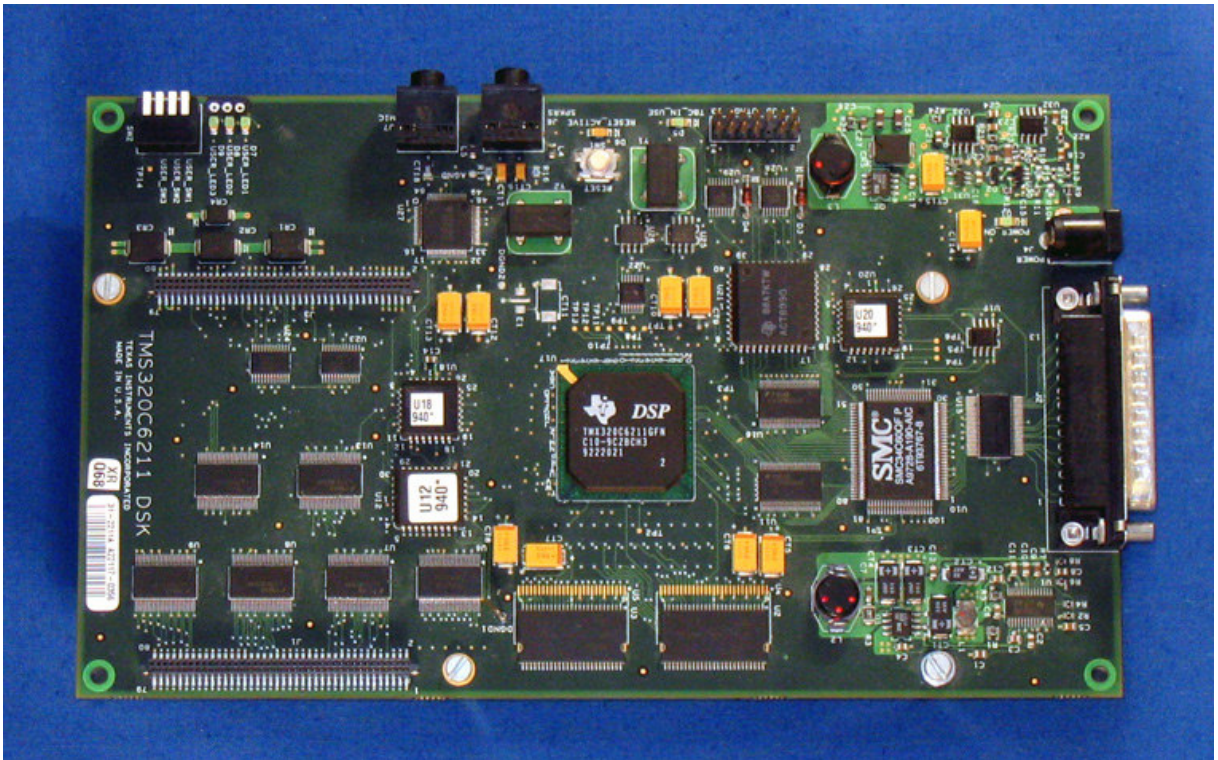


Figure 5.2: *Texas Instruments TMS320C6211 DSK*

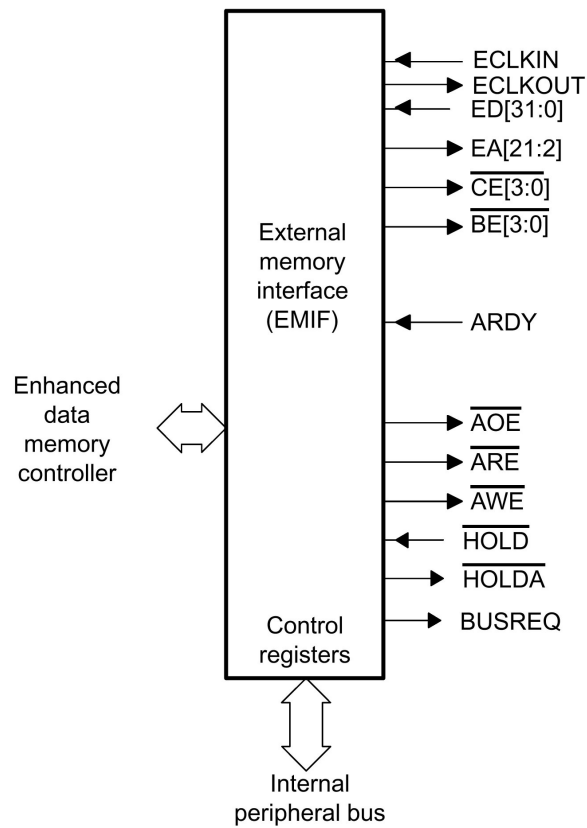


Figure 5.3: *TMS320C6211 External Memory Interface*

PIN	I/O/Z	Description
BUSREQ	O	Active high bus request signal
ECLKOUT	O	EMIF clock output. All EMIF I/O are clocked relative to ECLKOUT.
ECLKIN	I	EMIF clock input. Must be provided by system.
ED[31:0]	I/O/Z	Data I/O. 32-bit data input/output from external memories and peripherals
EA[21:2]	O/Z	External address output. Drives bits 21 to 2 of the byte address.
$\overline{\text{CE0}}$	O/Z	Active low chip select for memory space CE0
$\overline{\text{CE1}}$	O/Z	Active low chip select for memory space CE1
$\overline{\text{CE2}}$	O/Z	Active low chip select for memory space CE2
$\overline{\text{CE3}}$	O/Z	Active low chip select for memory space CE3
$\overline{\text{BE}}[3:0]$	O/Z	Active low byte enables. Individual bytes and halfwords can be selected for both read and write cycles. Decoded from two LSBs of the byte address.
$\overline{\text{RDY}}$	I	Ready. Active low asynchronous ready input used to insert wait states for slow memories and peripherals.
$\overline{\text{AOE}}$	O/Z	Active low output enable for asynchronous memory interface
$\overline{\text{AWE}}$	O/Z	Active low write strobe for asynchronous memory interface
$\overline{\text{ARE}}$	O/Z	Active low read strobe for asynchronous memory interface
$\overline{\text{HOLD}}$	I	Active low external bus hold (3-state) request
$\overline{\text{HOLDA}}$	O	Active low external bus hold acknowledge

**Table 5.1:** *EMIF Signal Descriptions*

The video processor RAM was setup to appear as asynchronous SRAM to the DSP. This was the simplest method, as the DSP could then read/write at its own pace. Control of the EMIF and the memory interfaces it supports was maintained by various memory-mapped registers - the relevant two being the EMIF global control register and the EMIF CE0, CE1, CE2 and CE3 space control registers. The EMIF global control registers set EMIF parameters such as EMIF signal polarities and enables. As all EMIF signals interface to the video processor via the FPGA, these values are arbitrary as the FPGA merely needed to be configured accordingly. Also, as the FPGA possesses some intelligence, not all the signals in Table 5.1 were utilized. The EMIF signals used were ECLKIN, ED[31:0], EA[21:2],  $\overline{\text{CE}}$  and  $\overline{\text{AWE}}$ .

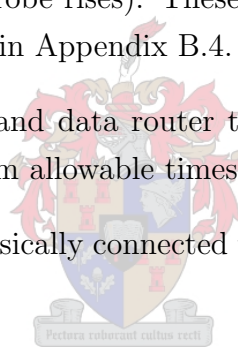
ECLKIN is the EMIF clock input and is provided by the DSK at 150MHz. ED[31:0],

the 32bit data bus, interfaces with the data routers; ED[31:16] with U13 and ED[15:0] with U12. EA[21:2] (the address bus) is routed directly to the FPGA, U9, as is  $\overline{\text{CE2}}$  and  $\overline{\text{AWE}}$ .  $\overline{\text{AOE}}$  and  $\overline{\text{ARE}}$  are generated by the FPGA dependent on the mode of the video processor.

The TMS320C6211 DSP has four separate memory mapped memory spaces assigning the same external port. By configuring these memory spaces differently, various external devices could be connected to the DSP simultaneously. The CE2 space was chosen, as the CE2 enable pin was the only enable pin on the same DSK header as the address and data lines. The CE2 space control registers control the specific operation of the CE2 memory space. The CE2 memory space was configured as a 32bit asynchronous memory. Various timing options were also configurable. They are read/write setup (the setup width - the number of ECLKIN cycles that the address and  $\overline{\text{CE2}}$  are held before the read/write strobe falls), read/write strobe (strobe width - the width of the read/write strobe in ECLKIN cycles) and read/write hold (hold width - the number of ECLKIN cycles that the address is held before the read/write strobe rises). These read/write timing diagrams as well as these timing options are shown in Appendix B.4.

From the RAM datasheets and data router timing diagrams (Appendix B.5), these values were set to their minimum allowable times to ensure maximum EMIF speed.

The video processor was physically connected to the EMIF by a small connector PCB (A.9) discussed in Section 3.2.8.



## 5.3 DSP Software

Although the DSP is used to implement tracking algorithms (which are only discussed in Chapter 7) the non-algorithm specific software will be discussed here, so as to provide a framework within which to discuss algorithm implementation.

The DSP software flowdiagram is shown in Figure 5.4.

Firstly all variables are defined and initialized (as in any C program) and then EMIF setup (as discussed in Section 5.2). Hereafter the DSP waits for a RTS0 rising edge (the start of the frame) to synchronize itself with the video stream as well as the video processor. The DSP control bus commands are then cycled in order - 000 (read frame 1 to RAM 1), 001 (read frame 1 to DSP), 010 (write tracking overlay to RAM 1), 100 (read frame 1 to RAM 2), 101 (read frame 1 to DSP), 110 (write tracking overlay to RAM 2) every time the DSP Control Cycle block is entered. After the appropriate command is

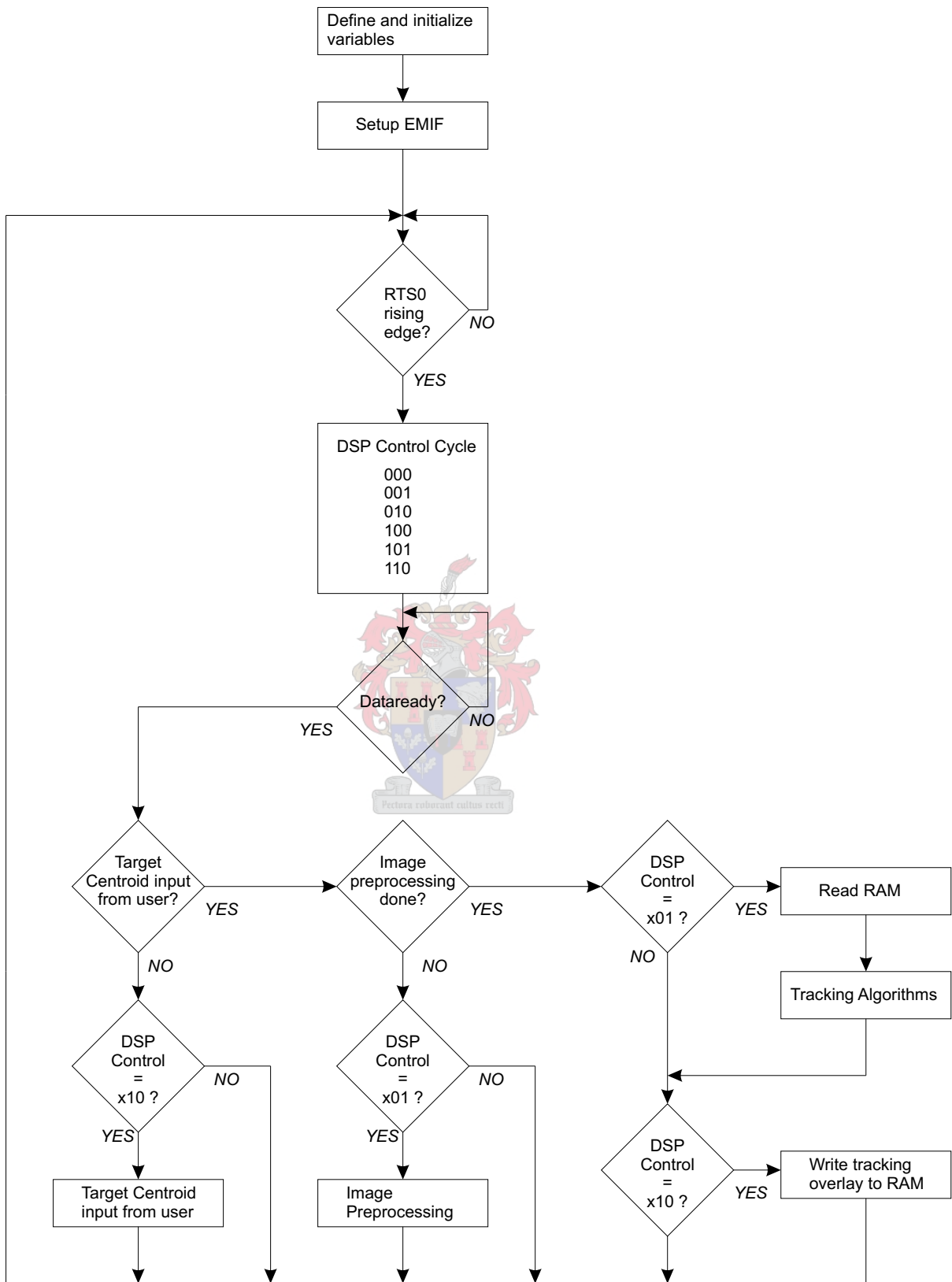


Figure 5.4: Flow Diagram of DSP Software

asserted on the control bus, the DSP waits for the DataReady signal (asserted by the video processor) to acknowledge that the command has been read and the video acquisition and display process has started. Hereafter the user enters an initial location of the target to provide a starting point for the tracking algorithms. The tracking algorithms require some processing to be done on the image before the tracking loop starts - this is done after the user has selected the initial target position. Hereafter the tracking loop is activated, data is read from the RAM and algorithms implemented during the next camera frame. During the third frame the tracking overlay is written back to the RAM, RAM blocks switched and the process repeated.

Some of these flow diagram blocks are not trivial and need to be discussed in further detail.

### 5.3.1 R/W RAM

Due to the EMIF, reading from and writing to the RAM is greatly simplified. The external RAM essentially becomes the memory mapped CE2 region and is written to and read from as one would any register. All timing is taken care of by the DSP EMIF. Data are read as follows:

```
data = (*((volatile unsigned int *) (0xA000000 + Mem_Offset)));
```

and written:

```
((*(volatile unsigned int *) (0xA000000 + mem_offset))) = data;
```

Where `data` is the value to be read/written, `0xA000000` the address of the CE2 memory space and `mem_offset` the external (video processor) RAM address.

Storing a two dimensional image in memory necessitates the image being stored in a long one dimensional array. As the images are 640x480 pixels in size, a single frame becomes an array of 1x307200 pixels and referencing specific pixels no longer as trivial as giving the `x` and `y` location. The memory address of a specific pixel is given by equation 5.1.

$$\text{adr} = x + y \times 640 \tag{5.1}$$

and the  $(x; y)$  coordinates of a memory address given by the inverse equation 5.2

$$x = \text{adr MOD } 640 \qquad y = \text{adr DIV } 640 \qquad (5.2)$$

where *adr* is the memory address of a specific pixel, MOD and DIV are standard C commands, with MOD returning the remainder after division and DIV returning a whole number as division result.


### 5.3.2 Recognizing Input Signal Rising Edges

Two different input signal rising edges need to be detected by the DSP, RTS0 and DataReady. The process is identical for both and only the RTS0 transition will be discussed. The procedure detecting rising edges is:

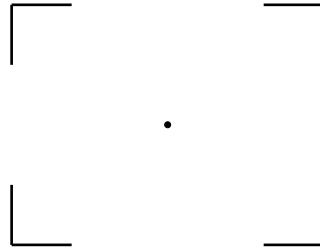
```

/*-----*/
void rising_RTS0() {
    /* declare vars */
    unsigned int edge, edge_rem;
    /* initialise vars */
    edge          = 1;
    edge_rem      = 0;
    /* Detecting falling edge and debouncing noisy input signal - RTS0 */
    while( ((GET_BIT(IO_PORT, 27) == 0) | edge))
    {
        if (GET_BIT(IO_PORT,27) == 0) {
            edge_rem = 1;
        }
        if (edge_rem) {
            if (GET_BIT(IO_PORT,27) == 0) {
                edge = 0;
            }
            else {
                edge_rem = 0;
            }
        }
    }
}
/* reset vars */

```







**Figure 5.5:** *Tracking Overlay*

```

    edge = 1;
    edge_rem = 0;
}
/*-----*/

```

Normally an edge would be detected by comparing the current value of a signal to its previous value - if they differ an edge has occurred. This approach detected all the edges, but gave spurious results and randomly indicated other edges. This was most probably due to noise on the input signal causing momentary spikes to be read as edges. The solution was to check for edges as before, but confirm the transition by reading the signal a third time. By checking that the third value was the same as the second, but different to the first an edge would be indicated. This provided the desired results, as only true edges were triggered as edges.



### 5.3.3 Generating the Tracking Overlay

The output of any tracking algorithm is a new centroid, or location of the target. This is indicated visually by overlaying a tracking block over the image at the location of the tracking centroid. The overlay is a partial rectangle, using only the corners and a hot pixel in the centre, shown in Figure 5.5.

Two arrays of memory addresses were created to generate the overlay pattern, one starting at zero and incrementing in 1's (for the horizontal lines) and the other starting at zero and incrementing in factors of 640 (for the vertical lines). This way, counting through these arrays would automatically write data in the correct shape. These lines were duplicated and offset in such a way that the overlay procedure only took the centroid location as input, counted through the arrays and the overlay would be written in the correct position.

### 5.3.4 User Centroid Input

To initiate tracking, the user needed to give the algorithm an initial value, or location, to start. This was done by allowing the user to manually move the tracking overlay pattern to a desired location and then activate the tracker. The TMS320C6211 DSK provides a memory mapped 4bit dip switch on the external interface and this was used for user input. `USER_SW1` was used to set the initial horizontal position. If the switch was pressed, the tracking overlay slowly looped in the horizontal dimension until the switch was lifted. `USER_SW2` does the same in the vertical dimension. When switch `USER_SW3` was pressed the overlay location was stored, passed to the algorithms and tracking started.



# Chapter 6

## Colour Transforms

Two very different methodologies exist to implement multi-input tracking:

- Implement tracking algorithms on individual inputs separately, and then combine the results.
- Transform the inputs to a single input, then implement algorithms on this input.

Implementing algorithms on individual inputs is not generally a good solution. Some inputs may contain very little if any useful information and tracking of these signals would not be worthwhile. For this reason, methods of transforming the multi-dimensional input to a single dimensional signal better suited to tracking were investigated and devised.

In this project, the multi-dimensional input consisted of different colour inputs, each covering a different spectrum. This multi-spectral input consisted of three spectra, and colour transforms would involve mapping these three spectra to a single, grayscale input.

### 6.1 Background

The three different signals comprising the multi-spectral input signal can be seen as the three components of an arbitrary colour space, where a colour is represented as a combination of these three colour components. In this project, the multi-spectral input could be seen as components of the RGB space, where the three spectra represented red, green and blue colour respectively.

Before a final, single dimensional output was generated, the input signal needed to be

transformed to different colour spaces (or colour models) to facilitate the desired information extraction.

Three different colour models were used in colour transforms:

- RGB
- YUV
- HSV

In this section the motivation behind these colour models, their advantages, and ways of visualizing them to facilitate transforms are all briefly discussed.

### 6.1.1 Different Colour Spaces/Models

The RGB colour space was briefly mentioned in Section 2.2.2 and will be discussed in further detail, as input data from the camera was in this format. The RGB colour space is an additive model based on three primary colours. Primary colours are related to biological concepts and are based on the physiological response of the human eye to light stimuli. The human eye contains photoreceptor cells which are sensitive to three different wavelengths - 564nm, 534nm, and 420nm. Therefore, to generate a optimal colour range, three primary colours need to be used<sup>1</sup>. The spectra of the individual RGB signals are shown in Figure 6.1.

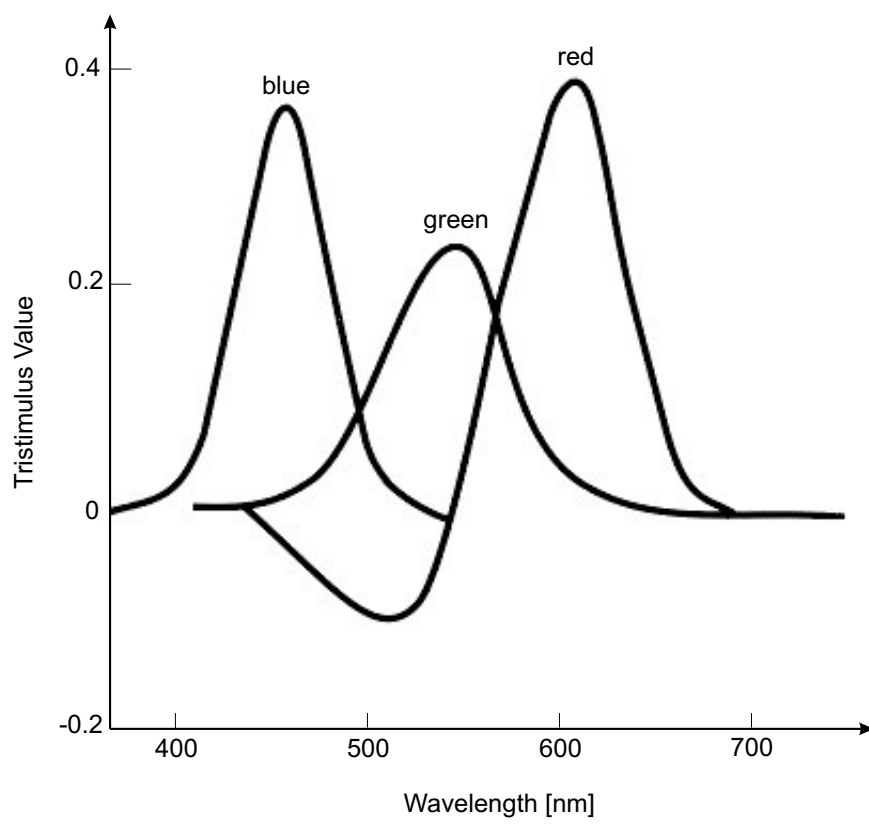
Although the peak responsivities of the cones do not occur at the red, green and blue wavelengths, these three colours are described as primary because they can be used relatively independently to stimulate the three kinds of cones [3]. Also, note that the negative values indicate that some colours cannot be produced exactly by adding primaries.

An advantage of the RGB colour space is that colours can be visualized easily - any colour is merely a weighted combination of the primaries.

The YUV colour space is primarily used in the transmission of television and was also discussed in Section 2.2.2. The YUV space is related to the RGB space by equations 2.1 and 2.2. The advantages of the YUV colour space is that luminance is directly available and information can easily be discarded to reduce the required bandwidth. The YUV

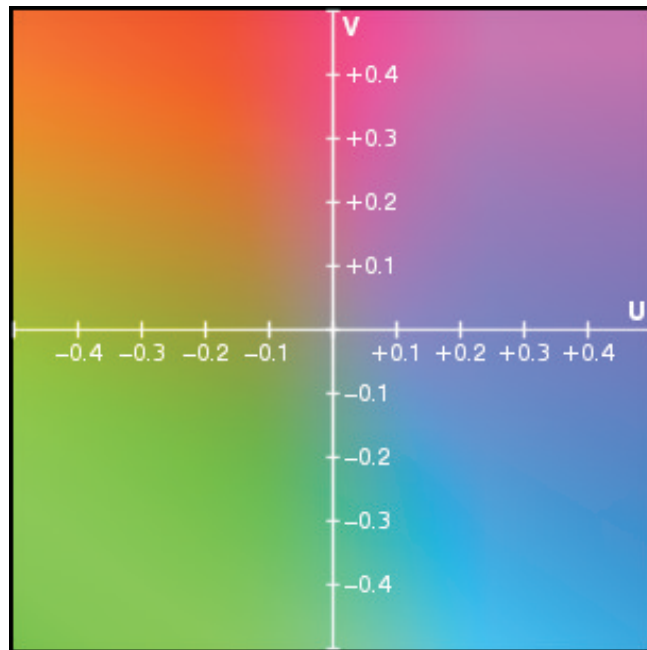
---

<sup>1</sup>The amount of primary colours depend on the different receptors in the eye - for a bird, with four colour receptors, four primaries need to be used.



**Figure 6.1:** *Spectra of RGB signals*

space is more difficult to visualize, but by setting  $Y = 0.5$  a slice through the YUV space is shown in Figure 6.2.



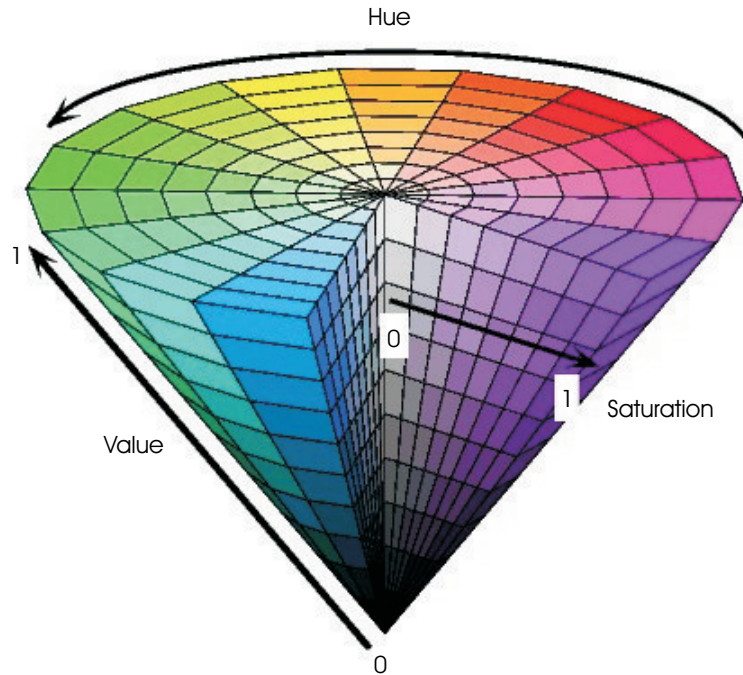
**Figure 6.2:** *The U-V Colour Plane,  $Y = 0.5$*

Another colour space used in colour transforms is the HSV space (*Hue Saturation Value* space). Its constituent components are:

- Hue. The colour (as in red, green or blue etc), usually in a range from 0-360°. (or normalized to 0-100%)
- Saturation. The 'vibrancy' or 'purity' of the colour, usually in a range from 0 to 100%. The lower the saturation of a colour, the greyer and more faded it will appear.
- Value. The brightness of the colour.

The HSV colour space has the advantage over other colour models that it is analogous to the way humans perceive colour. What colour is it? How vibrant is it? Is it light or dark? A visualization of the HSV space is shown in Figure 6.3.

Here, the hue is depicted as a three dimensional conical formation of the colour wheel, the saturation by the distance from the centre of a circular cross-section of the cone and the value by the distance from the bottom point. Although the HSV space does not technically support a one-to-one mapping with physical units, it is useful in that it offers a perspective on viewing colour and colour extraction not offered by other colour models.[1].



**Figure 6.3:** *The HSV colour space as cone*

### 6.1.2 Visualizing Colour Spaces

A useful aspect of 3 dimensional (3D) colour spaces is that they can be represented in a 3D space or a 3D plot. As the input to the video processor was three dimensional, this was particularly useful as the various inputs could be represented on a axis on the three dimensional space.

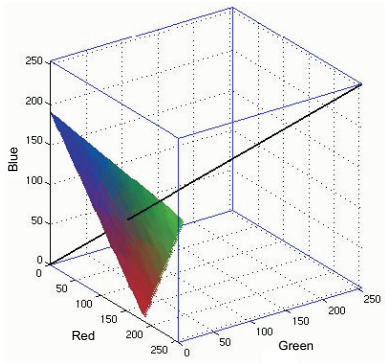
Input image data consisted of 24bit RGB data (8 bits per pixel) (known as 'unsigned integer, 8bit', or 'uint8'), specified using integers between 0 and 255. However, colour transforms require more bit depth accuracy and input data were cast to a double (*MATLAB*) and 32bit word (DSP) and denoted by a value between 0 and 1.

Figures 6.4, 6.5 and 6.6 illustrate three RGB axes, the neutral (gray) axis (equal sum of RGB) as well as orthogonal cross sections through the gray axis at different brightness values. The neutral axis is formally defined as the vector

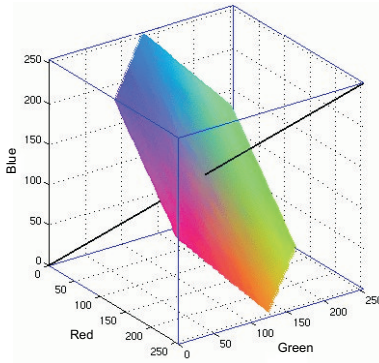
$$\text{neutral} = [1; 1; 1] \quad (6.1)$$

with brightness given by

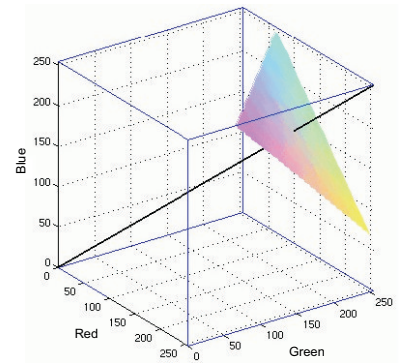
$$\text{brightness} = \frac{R+G+B}{3} \quad (6.2)$$



**Figure 6.4:** *The RGB colour space, Brightness = 25%*



**Figure 6.5:** *The RGB colour space, Brightness = 50%*

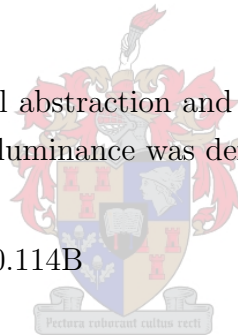


**Figure 6.6:** *The RGB colour space, Brightness = 75%*

Planes of constant brightness are perpendicular to the normal axis, therefore a RGB to grayscale transformation is merely an orthogonal projection of all points to the normal axis.

Brightness is a mathematical abstraction and does not conform well with the human perception of brightness, hence luminance was defined as

$$Y = 0.299R + 0.587G + 0.114B \quad (6.3)$$



which corresponds well to human vision, as the weightings 0.299, 0.587 and 0.114 closely match the eye's sensitivity to red, green and blue colours respectively.

It is also useful to visualize an image as a cloud of pixels in the RGB space. To understand colour space transforms it is essential to know how individual pixels are mapped from one 3D space to another. Figure 6.7 shows an image of a box of Smarties (multi-coloured chocolate sweets), chosen for their bright and distinct colour.

This image is displayed in the RGB space in Figure 6.8. All pixels have lost their  $(x; y)$  location and are displayed merely as a colour.

Distinct clouds of pixels are visible in Figure 6.8. All pixels of similar colour ( $(R; G; B)$  value) and brightness ( $\frac{R+G+B}{3}$ ) occupy a similar space in the RGB space. This is the essence of devising intelligent transforms to enhance target-background contrast (Section 6.3). Many of the colour clouds are elongated along the normal axis, due to differing brightness and glare in the original image. Colour clouds representative of every smartie in the original image can be distinguished.





Figure 6.7: *RGB Image Displayed as Normal*

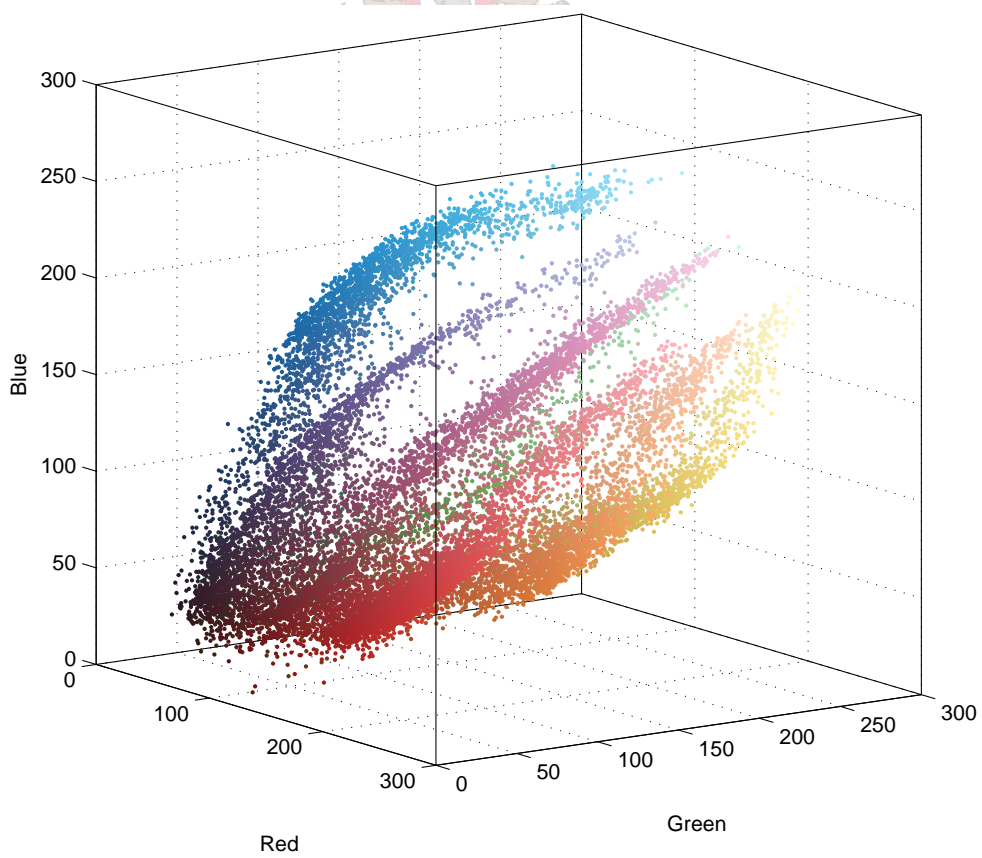


Figure 6.8: *RGB Image displayed in RGB space*

## 6.2 General Colourspace Transforms

Fixed colour transforms are not dependant on image information and usually consist of various image enhancement techniques and fixed colour mappings. In the context of this thesis colour transforms are a preprocess preceding tracking.

In general, the colour of a pixel is defined as

$$[C_0; C_1; C_2]$$

with colourspace transformation a function of the form

$$[C_0; C_1; C_2] \longrightarrow [C'_0; C'_1; C'_2]$$

As the inputs are RGB,

$$[C_0; C_1; C_2] = [R; G; B]$$

all the transforms will be in the form

$$[R; G; B] \longrightarrow [C'_0; C'_1; C'_2]$$

Remembering though that this application concerns itself with mapping a multidimensional input to a single dimensional one, the equation will be simplified to

$$[R; G; B] \longrightarrow [A]$$

where  $A$  the desired output image.

Many fixed colourspace transformations exist, such as RGB to normalized RGB, YUV, HSV and many others. In general, fixed colourspace transformations are assumed to increase separability between target and non-target classes. This was found not be the case [20], and the focus shifted to devise intelligent colourspace transforms.

## 6.3 Intelligent Transforms

Intelligent transforms are defined in the context of this thesis as a 3-1 mapping dependant on the input image and geared towards maximizing target and non-target (background) class separability.

This is done by defining certain aspects of the image as target (target detection), and others as background. Target detection can be done manually (with the user defining the target, known as target specification) or automatically (with an algorithm defining target

parameters). Automatic target detection is a complex and diverse field and beyond the scope of this study (colourspace transforms do however offer an interesting perspective on automatic target detection and will be discussed briefly in Section 6.5). Targets were specified by allowing the user to move a pre-defined rectangular overlay (Figure 5.5) over the input image/video, and indicating when the overlay was over the target of choice. By the very nature of this process some non-target pixels would be defined as target, but by arranging the experiment that the tracking overlay size was as close to the target size as possible, incorrectly marked target pixels were minimized. For preprocessing and colour transforms all pixels within the tracking rectangle were considered as target, and all outside as non-target/background. Target pixels are defined as

$$[C_{t0}; C_{t1}; C_{t2}]$$

and background pixels as

$$[C_{b0}; C_{b1}; C_{b2}]$$

Also, for simplicity targets were assumed to be of a single dominant colour. This is not a prerequisite for multi-spectral tracking, but greatly simplifies transforms. Methods to handle multicoloured targets are briefly discussed in Section 6.4.

The goal of the colour mapping and transforms was to take a RGB input image and transform it to a grayscale image with a higher contrast between the target and background. As first iteration, consider Figures 6.9 through 6.12.

The simplest 3-1 colour mapping would be to isolate the colour with the most desired information. For example, considering the red image, the contrast between all red/yellow Smarties and the non red/yellow Smarties would be increased (with reference to the grayscale image). To isolate blue Smarties, the blue image would be chosen. This is a haphazard approach, as it depends on how well the colour of the chosen object corresponds to the predefined RGB colours (Figure 6.1).

Therefore, colour needs to be characterized specifically. To this end, the input RGB image was transformed to HSV, and the different colour planes shown in Figures 6.13 through 6.16.

Figure 6.14 <sup>2</sup> (Hue) clearly shows the colour separation between objects of different

---

<sup>2</sup>The blocky-ness of the Hue plane is due to the definition of HSV. As hue is defined as an angle, a



**Figure 6.9:** *Smarties - Grayscale Image*



**Figure 6.10:** *Smarties - Red Plane*



**Figure 6.11:** *Smarties - Green Plane*



**Figure 6.12:** *Smarties - Blue Plane*



**Figure 6.13:** *Smarties - RGB Image*



**Figure 6.14:** *Smarties - Hue Plane*



**Figure 6.15:** *Smarties - Saturation Plane*



**Figure 6.16:** *Smarties - Value Plane*

colour. Figures 6.15 and 6.16 illustrate the color purity and brightness respectively. It can be seen that objects can be distinguished by their colour, but are still not uniform in their brightness and color intensity. This posed problems for colour transformation and will be discussed further in the section.

Obtaining the average colour of a target would be achieved by taking the mean value of all target pixel vectors. Such an average colour vector is defined as

$$[\bar{C}_{t0}; \bar{C}_{t1}; \bar{C}_{t2}] = \frac{1}{N} [\sum_{n=1}^N C_{t0n}; \sum_{n=1}^N C_{t1n}; \sum_{n=1}^N C_{t2n}] \quad (6.4)$$

This is analogous to determining the centre of a pixel cloud from Figure 6.8. In a similar fashion, the dominating background colour is calculated as

$$[\bar{C}_{b0}; \bar{C}_{b1}; \bar{C}_{b2}] \quad (6.5)$$

The direction of maximum contrast between the target and background are given by the difference vector

$$[d_0; d_1; d_2] = [\bar{C}_{t0}; \bar{C}_{t1}; \bar{C}_{t2}] - [\bar{C}_{b0}; \bar{C}_{b1}; \bar{C}_{b2}] \quad (6.6)$$

Projecting pixels orthogonally onto the difference vector produces a grayscale image along this axis of maximum contrast. This is a pixel-level dot product process and is illustrated by equation 6.7,

$$[A'] = [d_0; d_1; d_2] \bullet [C_0; C_1; C_2] \quad (6.7)$$

or

$$[A'] = |d||C| \times \cos \theta \quad (6.8)$$

where  $\theta$  the angle between the vectors  $|d|$  and  $|C|$ .

An indication of the confidence of the transform can be obtained from the difference vector - the larger the difference vector, the longer the region projected onto and thus the

---

point will exist with a 360 degree discontinuity between two points of very similar hue, causing the sharp edges in the Hue plane.

higher the contrast of the output image. Conversely, a short difference vector implies a low target vs non-target contrast in the output image.

From Equation 6.8, it can be seen that the difference vector,  $\bar{d}$ , can have negative components and is also not of unit size, therefore the output of equation 6.7 will not fall in the normal range valid for image pixel representation.

$$-2 \leq |d| \leq 2$$

therefore, as  $0 \leq |C| \leq 1$  and  $-1 \leq \cos \theta \leq 1$ ,

$$-2 \leq |A'| \leq 2$$

Values of  $|A'|$  less than 0 and greater than 1 are obviously invalid. There are two ways to overcome this problem

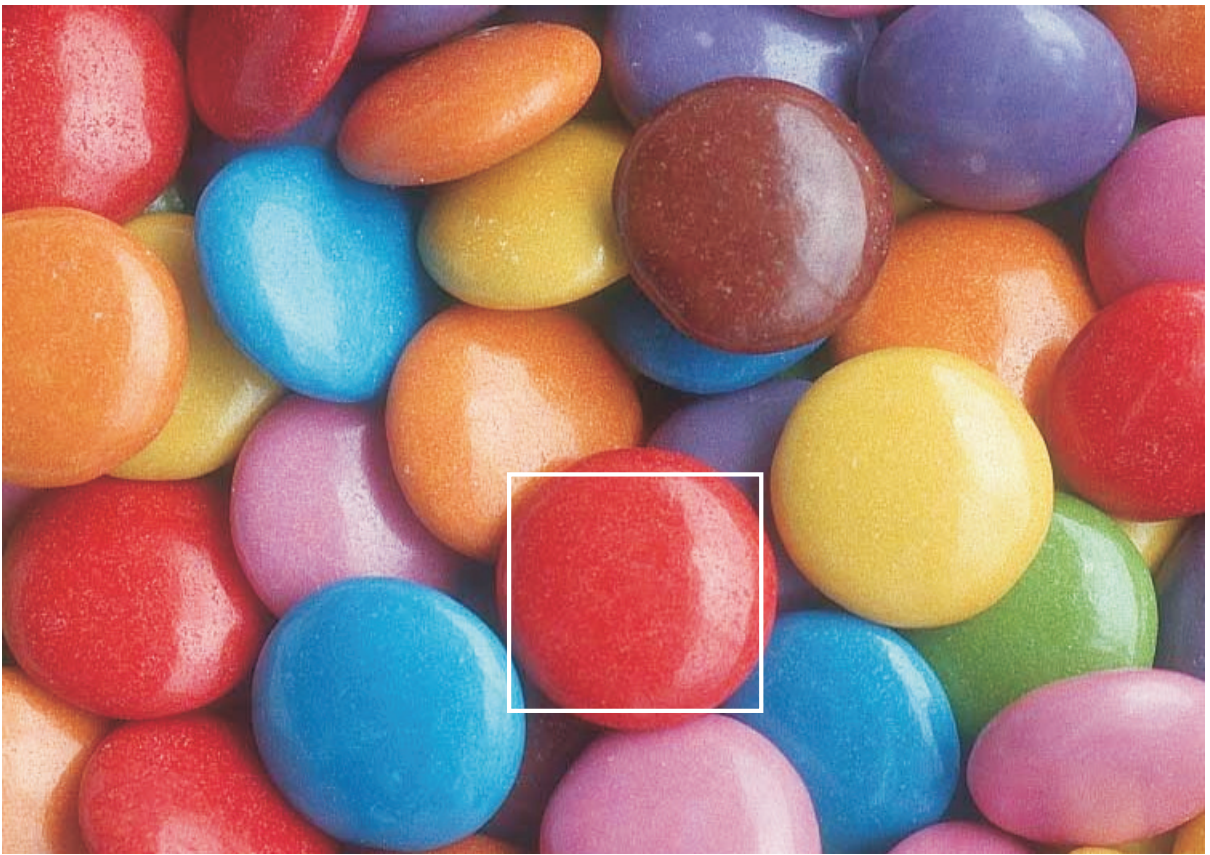
- Scale  $|A'|$  to values  $0 \leq |A'| \leq 1$
- Where  $|A'| < 0$ , set  $|A'| \rightarrow 0$ , and scale positive values of  $|A'|$  to  $0 \leq |A'| \leq 1$ . This 'truncated' image is referred to as  $A$ .

This process is illustrated in Figures 6.17 through 6.20

From Figure 6.17 it is apparent that some non-target pixels are included in the target definition, and also some other target pixels are designated as background. This is not a problem, as the majority of the pixels are labelled correctly. Figure 6.18 shows the two colour vectors as well as the difference vector. Figure 6.19 shows the output image with all values scaled between 0 and 1, Figure 6.20 is the output obtained with negative values discarded. Discarding negative  $A'$  values makes sense, as from equation 6.8 it is apparent that vectors with an angle difference of larger than  $90^\circ$  contain more information about the non-target pixels than the target pixels and can thus be set to 0.

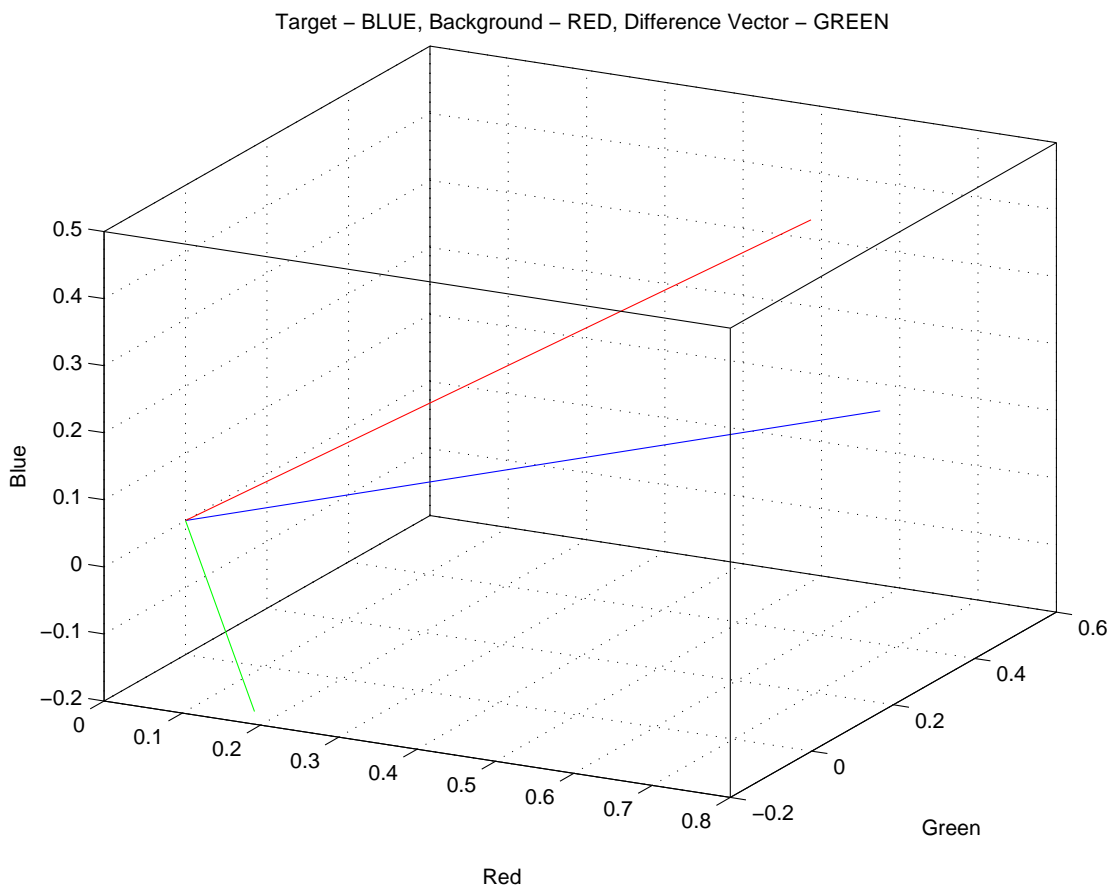
From Figures 6.19 and 6.20 two important observations can be made

- objects of similar colour to the target are still visible
- the brightness and glare have a big effect on isolating target pixels.



**Figure 6.17:** *Selecting Target Object*

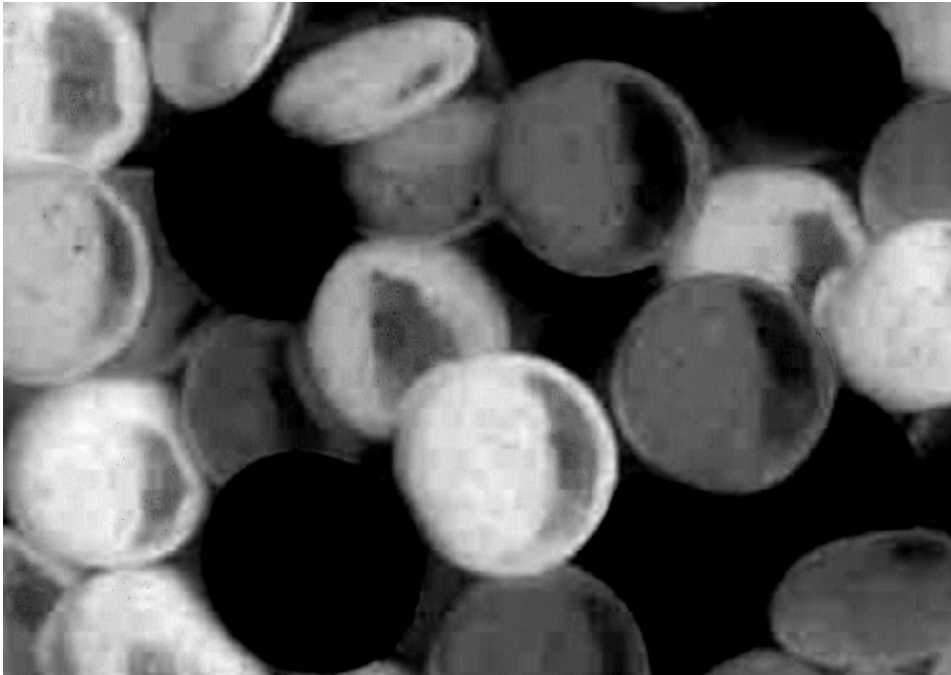




**Figure 6.18:** *Target, Background and Difference Vectors*



**Figure 6.19:** *Image Projected Onto Difference Vector and Scaled*



**Figure 6.20:** *Image Projected Onto Difference Vector, Negative Values Discarded*

What was also apparent in comparing different images is that the colour of the light source also had a big effect on the perceived colour of the target. For example, a target viewed under a red sky at dusk appears redder than it should.

Various authors have illustrated that images can be normalized to remove dependency on individual viewing conditions ([11], [18], [28]). Two different normalizations exist:

- Pixel based normalization. This consists of scaling every image pixel  $(r; g; b)$  to sum to one by  $(\frac{r}{r+g+b}; \frac{g}{r+g+b}; \frac{b}{r+g+b})$ , removing ambiguity due to illumination differences and pose.
- Channel based normalization. This is the scaling of every colour channel (all the values in the R,G and B planes) to sum to one,  $(\frac{R}{\sum_{i=1}^N R_i}; \frac{G}{\sum_{i=1}^N G_i}; \frac{B}{\sum_{i=1}^N B_i})$ . Dependency on the hue of the ambient light is so removed.

Neither approach removes dependency on both illumination hue and brightness, and the normalization best suited to the application should be selected. As this project's application consists of extracting a target from a background in the same field of view, the effect of the colour of the light source is felt by both target and background and is largely cancelled out - it's effect becomes negligible. Image normalization in the context of this thesis was largely pixel based, to remove the effect of the intensity of the illumination.

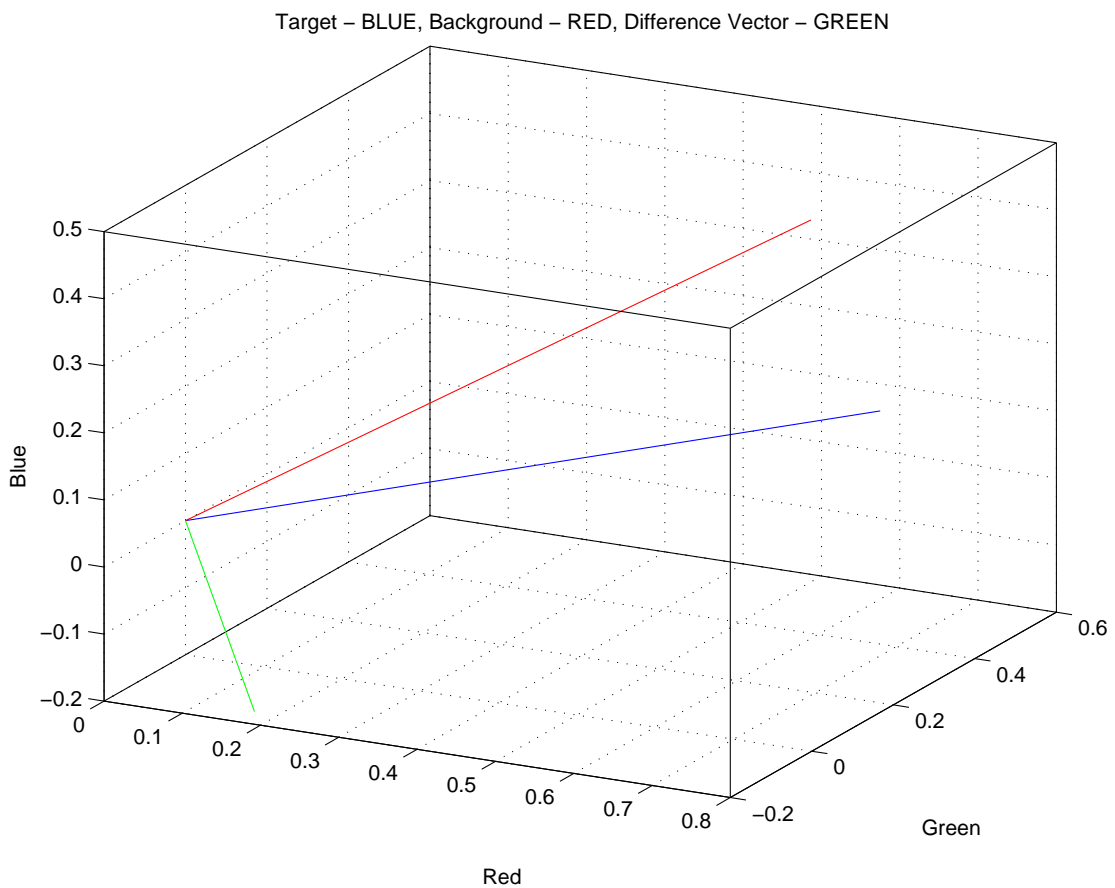
Colour vectors were calculated by equation 6.4 and 6.5 by summing the value of each pixel colour in the region of interest. Pixel based normalization would then involve normalizing each pixel, then calculating the colour vector. This process is distributive, and identical to normalizing the colour vector calculated using the non-normalized pixels. We thus normalize our colour vectors as:

$$C_N = [C_{N0}; C_{N1}; C_{N2}] = \left[ \frac{R}{|C|}; \frac{G}{|C|}; \frac{B}{|C|} \right] \quad (6.9)$$

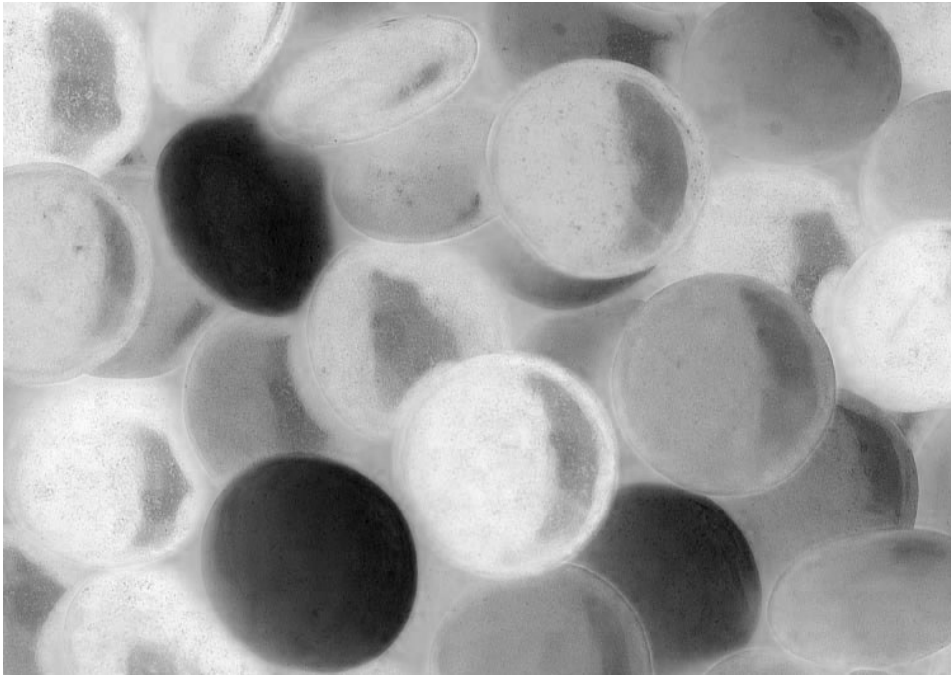
and calculate the new difference vector as:

$$\bar{d}_N = \bar{C}_{Nt} - \bar{C}_{Nb} \quad (6.10)$$

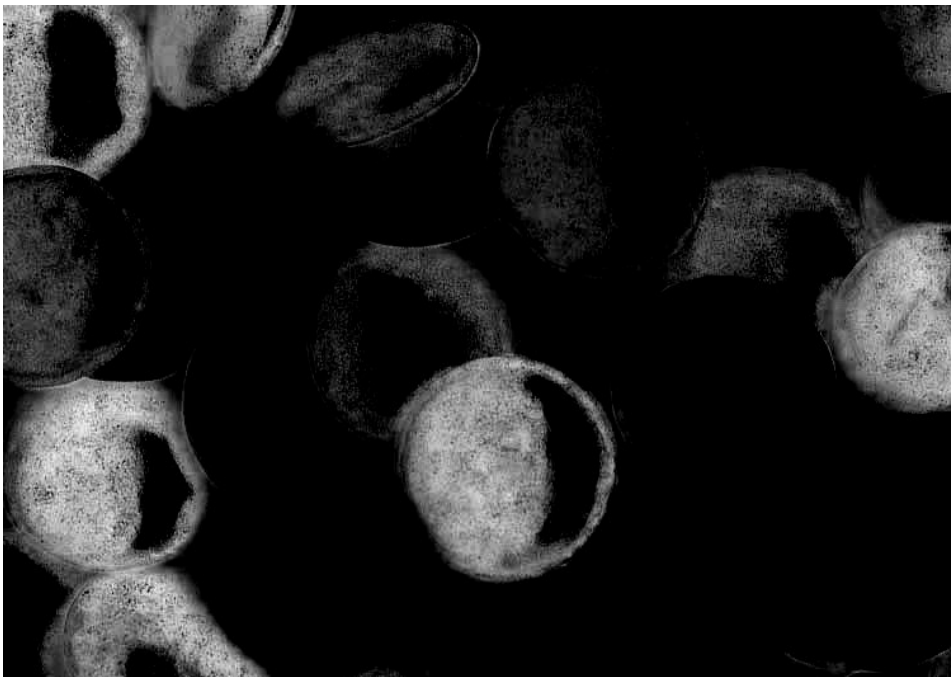
This normalization was implemented and the results shown in Figures 6.21 through 6.23.



**Figure 6.21:** Normalized Target and Background Vectors with the Resulting Difference Vector



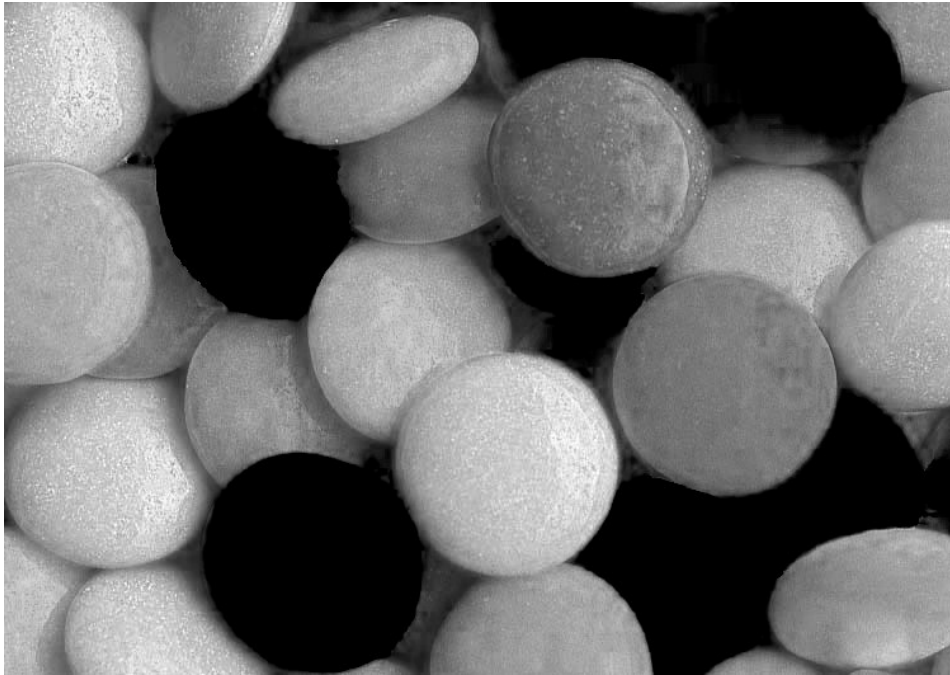
**Figure 6.22:** *Image Projected Onto Normalized Difference Vector and Scaled*



**Figure 6.23:** *Image Projected Onto Normalized Difference Vector, Negative Values Discarded*

By comparing Figures 6.20 and 6.23 it can clearly be seen that the pixel based normalization reduces the effect of the brightness of the image and the target vs background contrast is enhanced.

However, the dark patch due to the glare and reflection in the target is still visible. This is due to reflection desaturating colour (Figure 6.15). Applying pixel based normalization in the saturation plane resulted in the transformed image in Figure 6.24.



**Figure 6.24:** *Scaled and Truncated Output Image after Saturation Normalization*

It can be seen that the effect of glare has been significantly reduced, but target vs non-target contrast also reduced. As glare was usually only a problem in close proximity laboratory tests, glare reduction was not pursued and normalization limited to pixel based colour vector normalization.

We conclude this section with a final example, specifying the yellow kite as target (Figure 6.25), then the results of the colour transform algorithm (Figure 6.26) and normalizing the target and background vectors (Figure 6.27).

While results using the difference vector approach were good, a method of increasing target vs non-target contrast involving pixel clustering was also investigated.

## 6.4 Clustering

Any target appears as a cloud of pixels in the RGB space. If a method of isolating this cloud of pixels could be devised, a very powerful image transform could be developed. This method would consist of two steps, determining pixel clouds and then maximizing their separability. The process of determining pixel clouds is known as clustering or vector quantization (VQ).

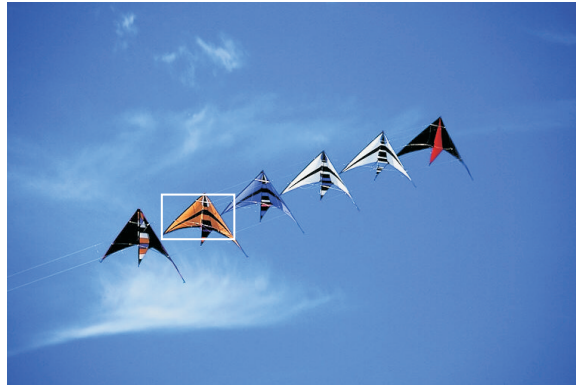
The objective of VQ is to represent a data set (in this case the RGB input image) by a set of  $K$  vectors. These vectors, or cluster centroids, should lie in areas of the feature space with a high data density. Practically, an input image would be transformed to a group of  $K$  vectors, each representing a significant colour. Thus, for our Smarties input image (subsamped and replotted in Figure 6.28, normalizing the brightness and discarding very dim pixels) there will be seven clusters, each representative of a different kind of smartie, and therefore seven cluster centroids, each at the mean value of the cluster colour.

Clustering also provides a powerful method of characterizing multi-coloured targets - each target will consist of a set of cluster centroids. Targets would then be characterized by their colour as well as the the relative position of target cluster centroids (colour patterns).

A drawback of this approach is the  $K$  is assumed to be known and its choice somewhat of a black art. Various clustering algorithms exist and a standard K-means algorithm was implemented, using the Euclidian Metric to determine cluster centroids (Section C.5).

For the first experiment an image was created using Adobe Photoshop by extracting a single object from an image and adjusting its hue to change its colour and increase separation from the background. An image of a group of leaves was taken and one leaf tinted blue. The results of VQ are shown in Figures 6.29 through 6.33.

It can be seen that the target and background clusters were accurately determined and isolated. Due to the blue tint, target separation was already high, and further clustering experiments were done on input images with no distinct class separation. The next



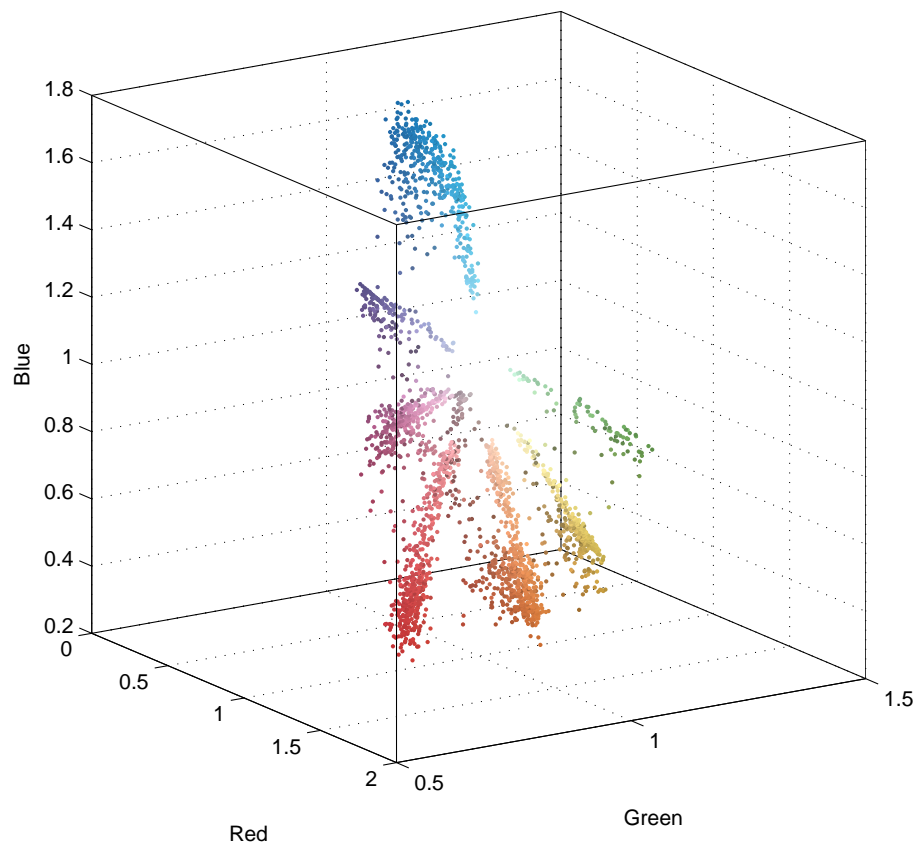
**Figure 6.25:** *Input RGB Image, Specifying Target*



**Figure 6.26:** *Output Image*

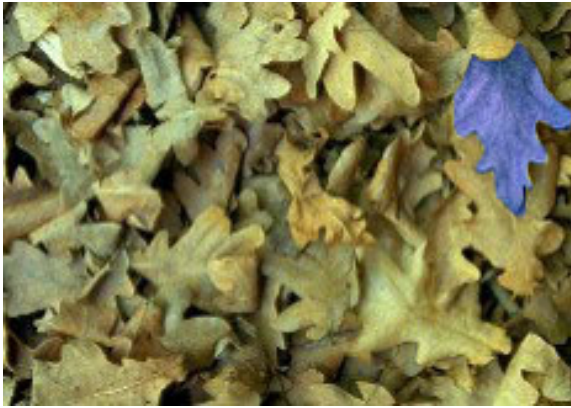


**Figure 6.27:** *Output Image - Normalizing Colour Vectors*



**Figure 6.28:** *RGB Image, Normalized, In RGB Space*

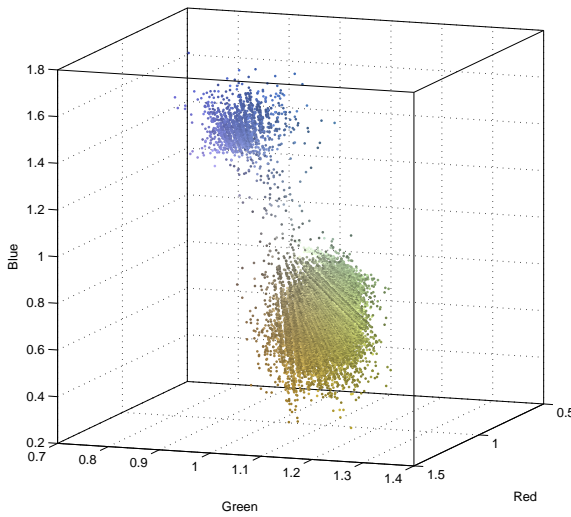




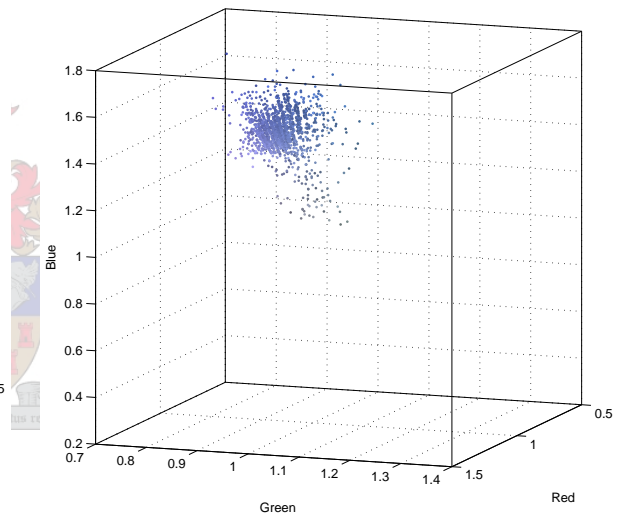
**Figure 6.29:** *Input Image - Single Object Tinted Blue*



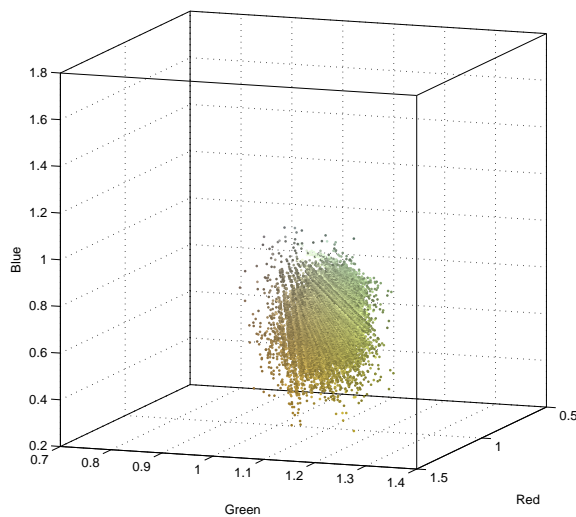
**Figure 6.30:** *Pixels Belonging to Blue Cluster*



**Figure 6.31:** *Input Image in RGB space (Normalized Brightness)*



**Figure 6.32:** *Target Cluster*



**Figure 6.33:** *Background Cluster*

experiment returned to the original Smarties image and attempted to isolate each colour smartie. Before attempting clustering, very bright and very dark pixels were discarded, as these contained less colour information than the pixels of average intensity.

Figures 6.34 to 6.39 illustrate the results of this VQ experiment (only showing two clusters). From the figures it can be seen that VQ results are adequate, but do not show the marked improvement over the difference vector approach that was hoped. The reasons are threefold:

- Class separation is inadequate. Some of what is perceived as different colours are often the same hue, but varying brightness (like yellow and orange) and different object clouds smear into each other. Also, object classes converge at very low and high levels of brightness.
- VQ cannot identify nonlinear and multimodal data relationships. From Figure 6.8 it is apparent that object clusters can have arbitrary shapes, may be separate and may even wrap around each other. VQ cannot characterize these relationships.
- The K means algorithm is optimal in datasets where data density is high in the  $(x; y)$  directions, whereas in this case data are arranged in a polar fashion.

Also, VQ is not solely a preprocess. VQ would need to be performed on every input frame and would be computationally expensive.

For these reasons clustering was not chosen as method of colour transformation. Non-linear extensions to linear discriminant analysis (LDA), also known as the Karhunen-Loeve transform (KLT), show much promise in maximizing class separability [26] and would benefit from further study.

## 6.5 Automatic Target Detection

A target can be viewed as a group of pixels of uniform colour, brightness and location. This is the essential difference between a target and a background, which may have uniform colour and brightness, but not location. A background is after all spread across the entire image. Different algorithms were devised for automatic target detection, and rest on two main approaches:

- Difference Vector. The background vector is calculated as the mean colour direction of the entire image. A window is then moved over the image calculating the potential



Figure 6.34: *Input RGB Image*

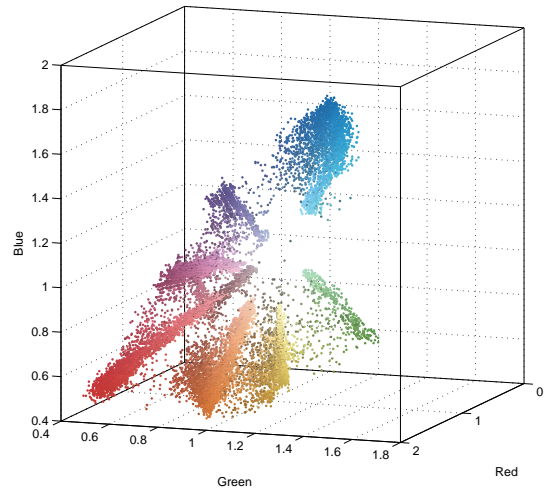


Figure 6.35: *Input RGB Image in RGB Space*



Figure 6.36: *Pixels Belonging to Cluster 1*

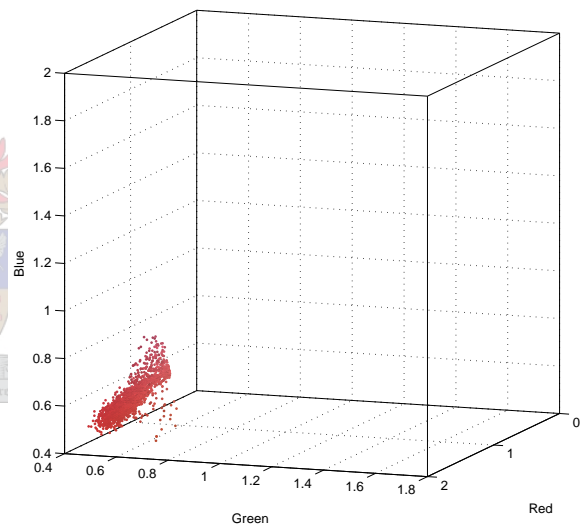


Figure 6.37: *Cluster 1 in RGB Space*

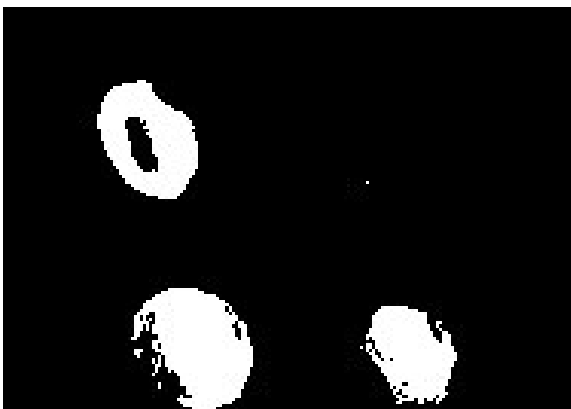


Figure 6.38: *Pixels Belonging to Cluster 3*

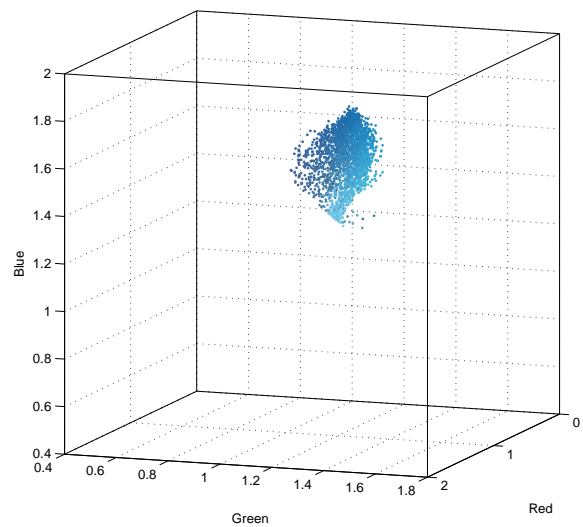


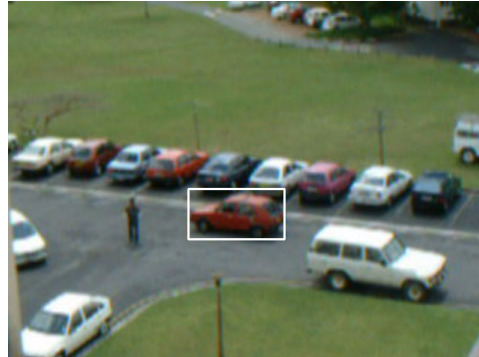
Figure 6.39: *Cluster 3 in RGB Space*

'target' vector and difference vector at each point. The point corresponding to the largest difference vector is the location of an object with the largest colour difference to the background. The advantage of this approach is that the desired direction of the difference vector can be specified, so as to locate targets of specific colours.

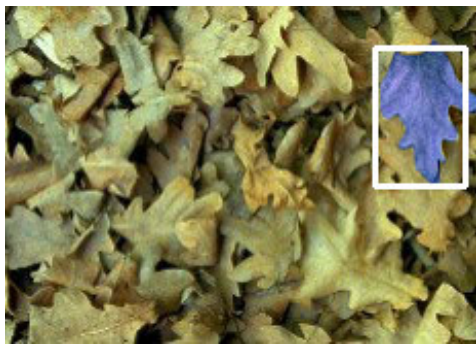
- Five dimensional clusters. All objects of similar brightness and colour appear as clusters in the RGB space. By expanding the 3D RGB space to a 5D RGBxy space by including the location of the pixels all objects of similar brightness, colour *and location* appear as clusters in this space. The background will appear as a cluster in the RGB space, but will be smeared across the RGBxy space. The target will appear as a cluster in both spaces, which will identify it as a potential target. Accurately identifying clusters was problematic, and this approach was not successful.

The difference vector approach was tested (C.5) and the results shown in Figures 6.40 through 6.44. In each case only the approximate size of the target was specified with the algorithm outputting the target location and overlaying the tracking rectangle. In Figure 6.43 the algorithm was biased to extract red objects and in Figure 6.44 blue objects.





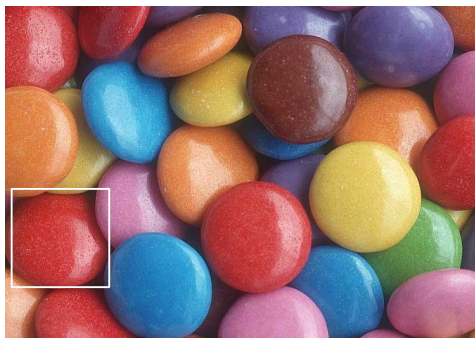
**Figure 6.40:** *Automatic Detection of Red Car*



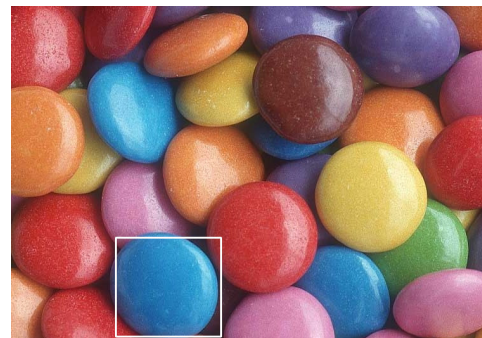
**Figure 6.41:** *Automatic Detection of Blue Leaf*



**Figure 6.42:** *Automatic Detection of Red Leaf*



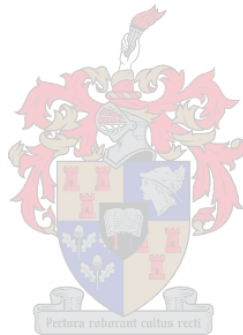
**Figure 6.43:** *Automatic Detection of Red Smartie*



**Figure 6.44:** *Automatic Detection of Blue Smartie*

## 6.6 Conclusions

After much experimentation, it was decided that the colour transform algorithm to implement would be the difference vector approach, utilizing the inherent thresholding of negative values. Also, from the experiment in Figure 6.24, it was seen that saturation normalization decreases the transform dependency on glare, but adversely affects the target vs non-target contrast as it decreases separation in the RGB space. By normalizing the target and background vectors, the spatial separation of pixels in the RGB space is maintained and transform dependency on illumination reduced.

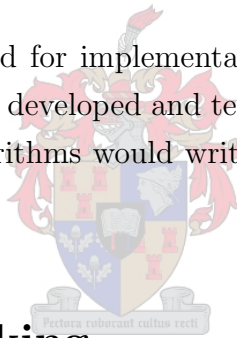


# Chapter 7

## Tracking Theory and Algorithms

Colour transformation outputs a grayscale image with enhanced target vs non-target contrast. This implies that standard tracking algorithms could be implemented using the grayscale images/video as input.

Two algorithms were selected for implementation, centroid tracking and correlation tracking. These algorithms were developed and tested in MATLAB, by using a .avi video file as input. The tracking algorithms would write a tracking overlay onto the target to indicate its position.

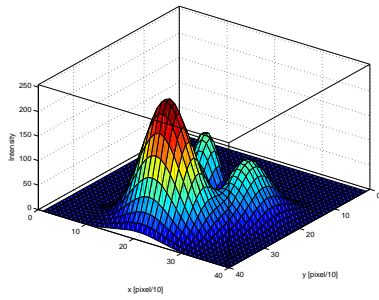


### 7.1 Centroid Tracking

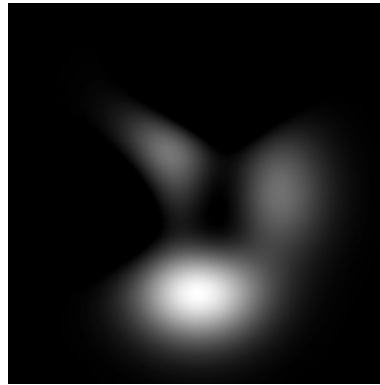
Centroid tracking / hottest spot tracking consists of tracking the brightest object in the area under surveillance. This method is ideally suited to multi-spectral tracking, as colour transform algorithms by their very structure output an image where the target is as bright as possible. Determining the centroid, or centre of the brightest object in the field of view is not trivial. This is illustrated by the arbitrary 'target' image in Figures 7.1 and 7.2. The target is an arbitrary shape with three peaks, one distinctly higher than the other. This raises the obvious question - where is the centre of this target? This becomes more of a philosophical question, as the answer really depends on the specific application. For example, the centre of a U-shaped object would not even be part of the object, and launching a projectile at the 'centre' of the target would miss the target. Figure 7.3, the target image thresholded, illustrates this problem - there are now three potential 'targets'.

Four different methods to determine the target centroid were investigated:

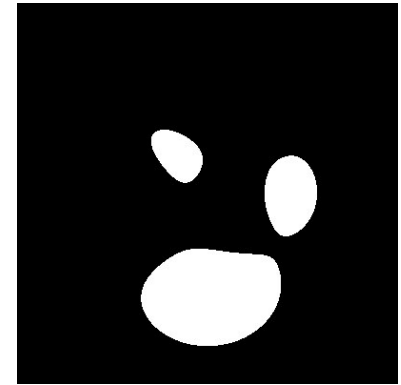
- The mean value of the coordinates of all pixels above a certain threshold. This



**Figure 7.1:** *Target Image of Arbitrary Shape Plotted as a 3D Function*



**Figure 7.2:** *Target Image of Arbitrary Shape*



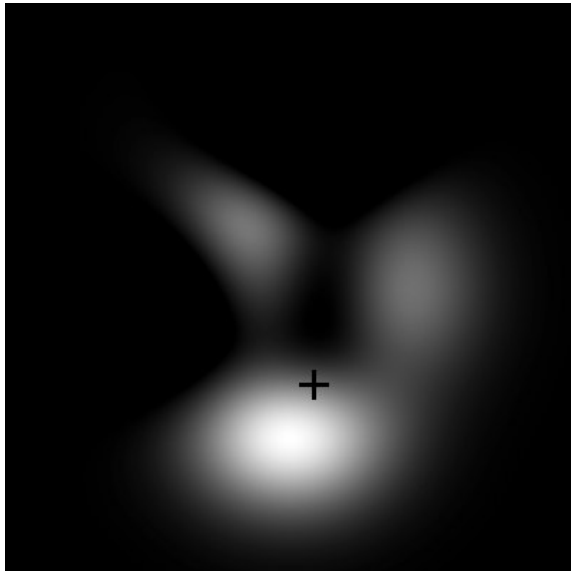
**Figure 7.3:** *Binarized Target Image*

simply amounts to taking the  $x$  and  $y$  average of all pixel coordinates in Figure 7.3 and is analogous to determining the centre of gravity of a disk-shaped object similar to Figure 7.3. This centroid is shown in Figure 7.4.

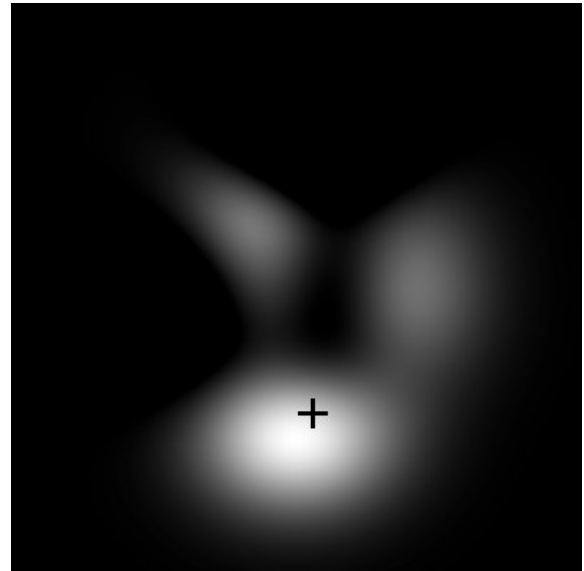
- The median value of the coordinates of all pixels above a certain threshold. This involves selecting the middle of the  $x$  and  $y$  direction datasets of pixel coordinates. This centroid is shown in Figure 7.5.
- Weighing each pixel coordinate by its value (intensity), and then determining the mean of the weighed coordinate datasets. This would weigh brighter pixels, which would normally appear towards the centre of a target, more heavily than background pixels. The centroid calculated in this way is shown in Figure 7.6.
- Taking the centroid as the brightest point. The brightest point is shown in Figure 7.7.

It can be seen that all centroids apart from the brightest spot centroid were slightly off centre with respect to the brighter target spot. The brightest spot algorithm is unstable, and can oscillate between different bright spots on the target and was thus unreliable. The other three centroids were pretty similar, the best being the median centroid, as the median essentially weighs the datadensity - there being more pixels above a certain threshold towards the centre of the target than toward the edges. Both median and weighted centroids are computationally expensive, and in this application unnecessary. Due to the colour transform, all pixels above the threshold *will* be part of the target, and no complex weighing or density measure of these coordinates were necessary. Also, targets were assumed to be unimodal, therefore any perceived separation would be due to the transforms and reflection - points inbetween these 'modes' would still be part of the

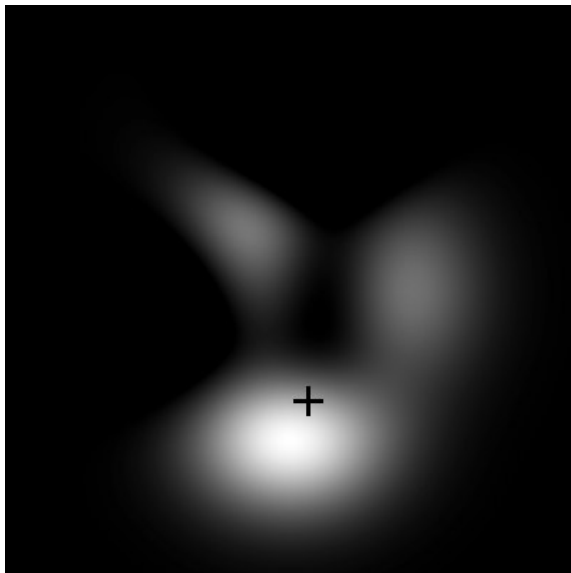




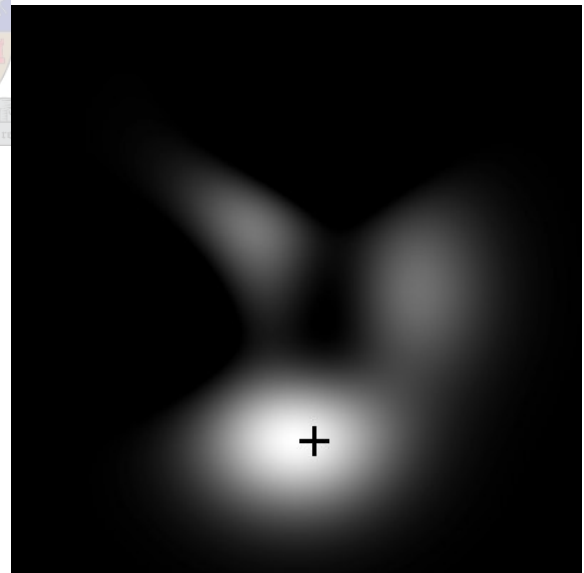
**Figure 7.4:** *Centroid Calculated with Mean Values*



**Figure 7.5:** *Centroid Calculated with Median Values*



**Figure 7.6:** *Centroid Calculated with Weighted Values*

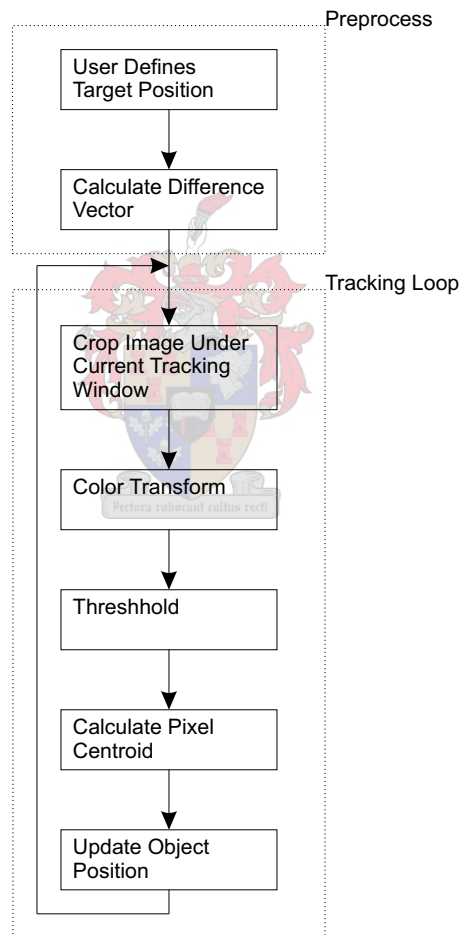


**Figure 7.7:** *Centroid Calculated with the Brightest Point*

target. Therefore the centroid was calculated by taking the mean value of the coordinates of all pixels above a certain threshold.

Selecting the threshold was not as crucial as in many other tracking applications, due to the colour transform outputting a normalized image. This implied that there would *always* be pixels above any threshold. The higher the threshold, the more the calculated centroid would approach the brightest spot centroid. A threshold was calculated with an algorithm proposed by Otsu [22], to maximize separability between the resulting classes of the binarized image. No significant difference between this threshold and any other was found, and the threshold was set to half the maximum image intensity.

The complete tracking process is illustrated in Figure 7.8.



**Figure 7.8:** *Centroid Tracking Flow Diagram*

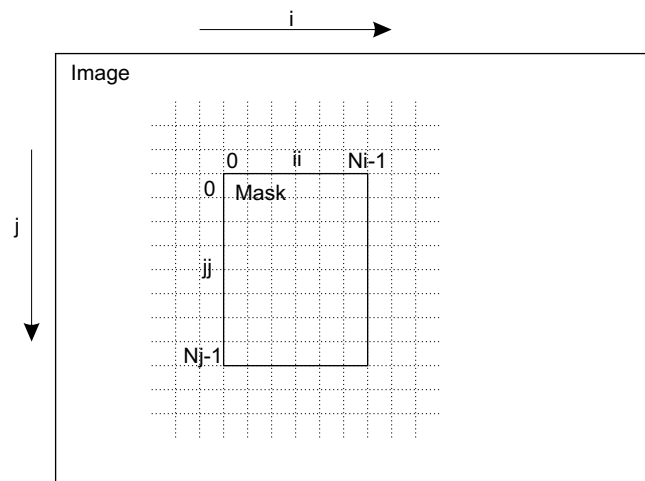
The target is specified by the user using a rectangular overlay and then the difference vector calculated. A tracking window slightly larger than the tracking overlay is then defined. This is to limit the search for the next brightest spot to the vicinity of the old spot. This allows other bright objects to be in the field of view, ignored by the tracking algorithm, but limits the bandwidth of the target movement. The colour transform is

then applied to the area of the video frame under the tracking window, producing the desired transformed output. By limiting the object search to the tracking window only a portion of the image needed to be transformed, significantly speeding up tracking algorithm performance. The smaller image is then thresholded and the mean position of all pixels above the threshold determined, giving the centroid of the object being tracked. The object position is then updated, and the tracking rectangle overlaid on the output image.

This approach was relatively problem free. Due to the image being scaled in the colour transform there were always pixels above the threshold level and the image was never lost. Problems occurred when very similar objects to the one being tracked crossed the tracking window. Solving this problem required that multiple objects be tracked, which was beyond the scope of this study. Also, due to the centroid being given by the mean value of all pixels above a threshold, target rotation would shift the perceived centroid. Tracking susceptibility to spurious objects and target rotation was decreased by including a position and velocity estimator in the algorithm, discussed in Section 7.3.

## 7.2 Correlation Tracking

2D Cross correlation is a powerful method of pattern identification. The standard approach to identify a pattern in a image uses cross correlation with a suitable mask (or correlation template). Where the mask and the image are similar the cross correlation will be high. The mask needs to be an image of similar functional appearance to the pattern being sought. The process is illustrated in Figure 7.9.



**Figure 7.9:** *Cross Correlation Algorithm*

With the (un-normalized) correlation coefficient at position  $(i, j)$  on the image given

by

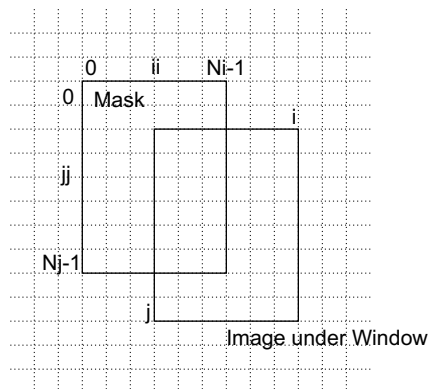
$$r[i][j] = \sum_{jj < Nj/2} \sum_{ii < Ni/2} (\text{mask}[ii + Ni/2][jj + Nj/2] - \overline{\text{mask}}) \times (\text{image}[i + ii][j + jj] - \overline{\text{image}})$$

where  $\overline{\text{mask}}$  is the mean of the mask's pixels and  $\overline{\text{image}}$  the mean of the image pixels covered by the mask [7].

For correlation tracking, the mask will be the target inside the tracking rectangle, as defined by the user. Correlation tracking can be problematic for two main reasons:

- The process can be time consuming as the 2D cross correlation function needs to be computed for every point in the image.
- The correlation template needs to be scaled/rotated at every iteration.

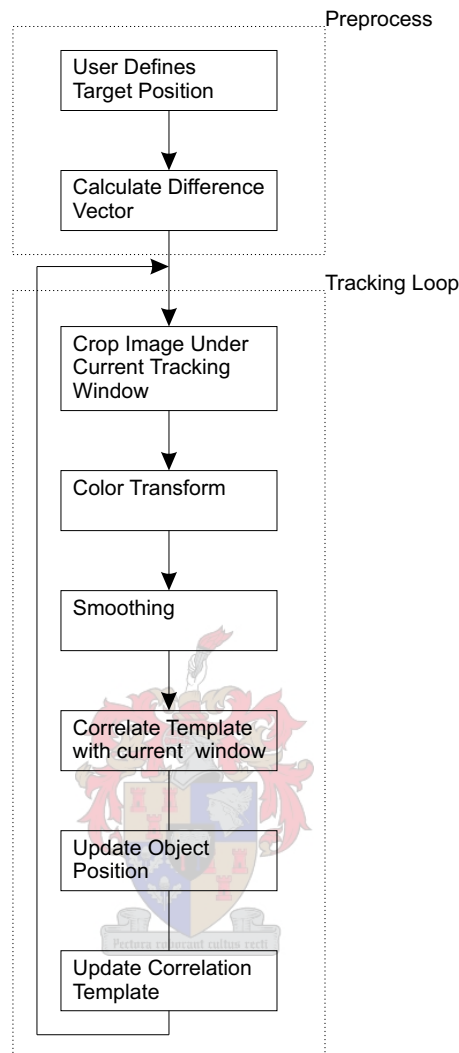
These problems were overcome by only performing cross correlation between the correlation mask and the image under the tracking window. As standard cross correlation algorithms require a mask to be smaller than the input image, the cross correlation algorithm had to be rewritten for it to dynamically alter the correlation boundaries as two images of equal size were correlated whilst only partially overlapping (Figure 7.10).



**Figure 7.10:** *Cross Correlation Algorithm for Images of Equal Size*

After each correlation iteration, the correlation mask was updated by centering the tracking window over the calculated object location and taking the image under the window as the correlation mask.

The basic structure of the correlation tracker is similar to that of the centroid tracker and is shown in Figure 7.11.



**Figure 7.11:** *Correlation Tracking Flow Diagram*

Where centroid tracking involves thresholding and binarization, correlation tracking requires much smoother images, producing a smooth surface of correlation values. Where centroid tracking binarizes the images through thresholding, correlation tracking smooths the image through a low-pass filter. The preprocess is identical to centroid tracking. At the initial position, the image in the tracking overlay is defined as the first correlation template. On the next frame, the image in the tracking window is transformed to grayscale and smoothed. This image is correlated with the correlation template and the location of the next target centroid determined as the position of the highest correlation coefficient. The correlation template is then updated by selecting an image of the size of the tracking overlay centered at the new position. A target overlay is then written to the output image to display target location and the process repeated.

Updating the correlation mask on every sample sometimes caused output instability in a manner analogous to having a control loop inside another control loop. The correlation mask update can be seen as one loop, and the output update as the other. By only updating the correlation mask every five samples, these two loops were separated and no further instability experienced.

Correlation tracking is powerful and accurate, but due to the large amounts of data to be processed it is very slow.

### 7.3 Estimator

The outputs from the tracking algorithms were noisy, and would drift about the target. This was especially apparent in the centroid tracking algorithm. Due to the object rotating, or variations in illumination and reflection, the output of the colour transform algorithms would vary, resulting in inaccuracies in the perceived centroid of the target.

By incorporating a simple estimator<sup>1</sup> in the tracking algorithm output some statistical inference is included to reduce this noise. Estimation with tracking application is a diverse field, and is commonly used in trajectory determination. In this case however, the suppression of higher frequency noise was the main focus, and simple first and second order low pass filters were sufficient.

As first iteration, a first order low-pass filter was implemented as

$$\hat{p}_k = \frac{(p_k + p_{k-1})}{2} \tag{7.1}$$

with  $p$  the  $(x; y)$  coordinates of the target position and  $\hat{p}$  the estimated position. Equation 7.1 amounts to taking the average of the current measured position and the previous position. This was satisfactory for stationary and slow moving targets, and reduced noise substantially. At the cost of this stability however, a constant velocity tracking error was introduced.

To overcome the constant velocity error, velocity was included in the position estima-

---

<sup>1</sup>The use of the term 'estimator' is debatable, as it is implemented as a simple, FIR filter. However, the constant velocity assumption in equation 7.2 implies a simple model of the target movement, hence the term 'estimator' is used.

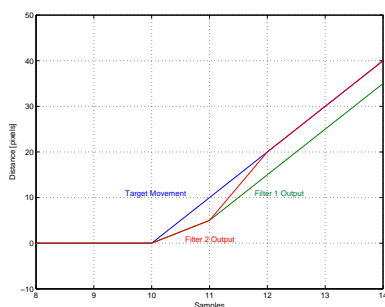
tion, expanding equation 7.1 to a second order filter, or

$$\hat{p}_k = \frac{(p_k + p_{k-1})}{2} + \frac{(p_{k-1} - p_{k-2})}{2} \quad (7.2)$$

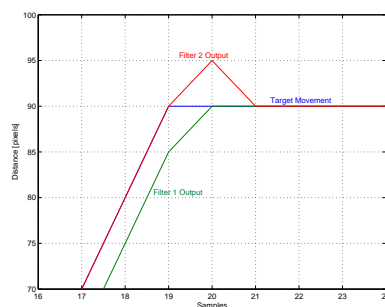
Equation 7.2 amounts to an equal weighting of equation 7.1 (position by averaging current and previous position)(the first term) and the change in position estimate (by taking the velocity at the previous sample)(second term). The factor two in each term determines the weighting of the position and velocity terms. Division by two is simple in hardware, as it amounts to a single right bit shift, and was implemented as is, even though different weightings of the terms would produce different responses.

The averaging term in equation 7.2 substantially reduced steady state errors, and the inclusion of the velocity estimate improved the velocity tracking error. Small errors still occurred upon target acceleration due to the constant velocity assumption inherent in equation 7.2, but were negligible.

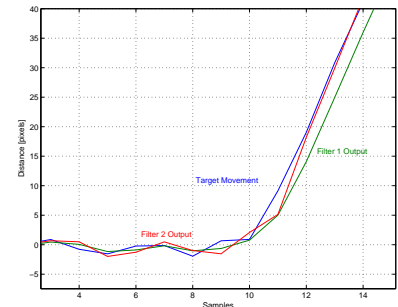
These two filters were simulated in MATLAB and results shown in Figures 7.12 to 7.14. A ramp input of 10 pixels per sample, starting at sample 10, was given. Figure 7.12 shows the start of the ramp and Figure 7.13 the end. It can be seen that filter 1 (equation 7.1) lags the input by half the target rate due to averaging. Filter 2 (equation 7.2) exhibits a similar lag upon target acceleration, but a zero tracking error at constant target velocity. Figure 7.13 illustrates the end of the ramp, with the characteristic overshoot from filter 2 settling two samples after deceleration. Figure 7.14 illustrates these two filters' response to a noisy signal. As can be expected, filter 1 attenuates the higher frequency movement, but still produces a constant velocity error where filter 2 does not.



**Figure 7.12:** *Filter Outputs with Ramp Input - 1*



**Figure 7.13:** *Filter Outputs with Ramp Input - 2*

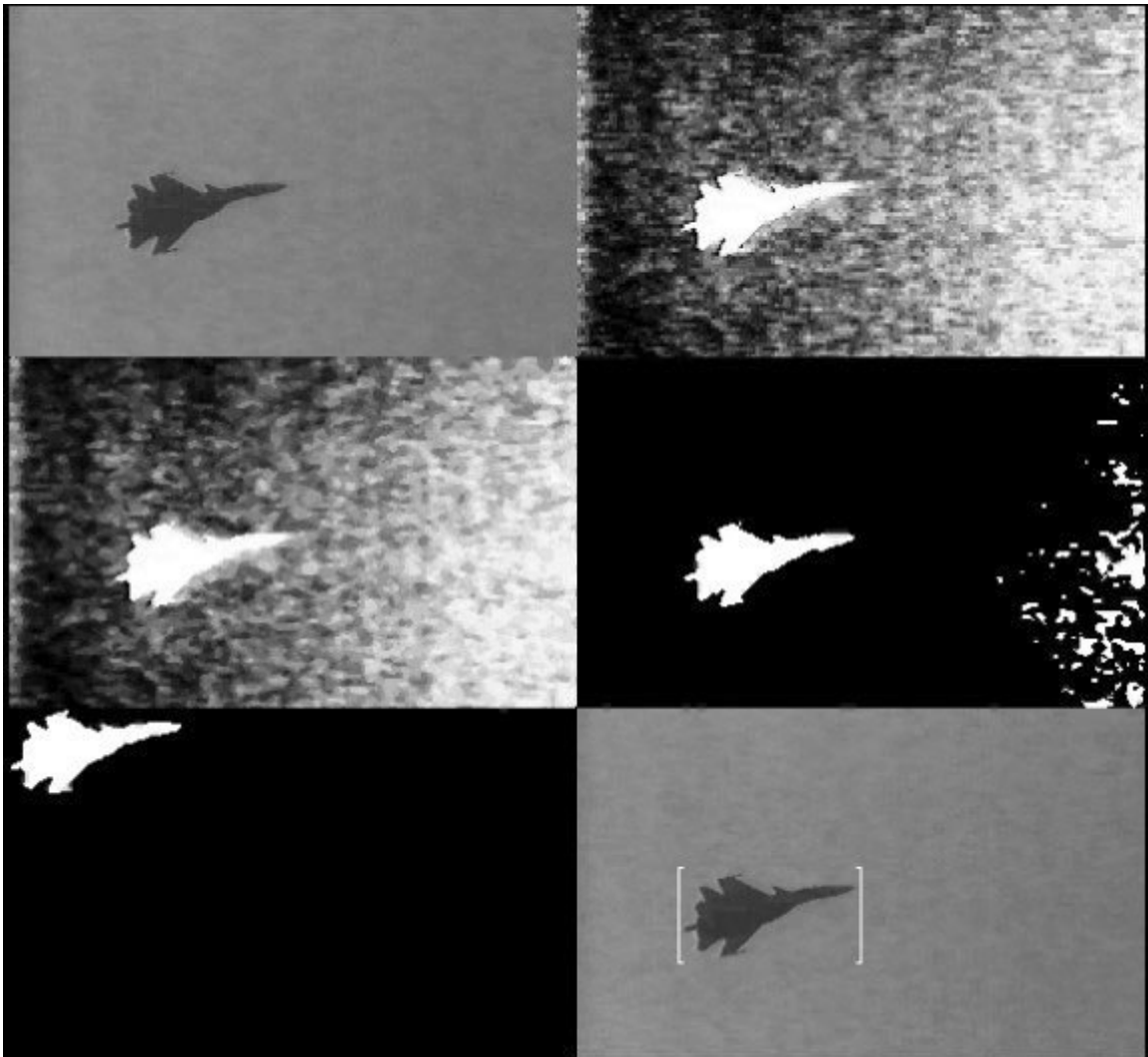


**Figure 7.14:** *Filter Outputs with Noisy Ramp Input*

## 7.4 Simulation

Both centroid and correlation tracking were simulated in MATLAB. Video files were imported in .avi format and the algorithms tested. This provided an idea of what type of performance to expect from the DSP and also highlighted possible pitfalls and problems that could occur.

Figure 7.15 is a screenshot of the .avi output of a MATLAB simulation. From left to right, top to bottom, the input image, input image inverted<sup>2</sup> (for object to be brighter than background), smoothed image, image thresholded, correlation mask, and the algorithm output showing the tracking overlay over the target.



**Figure 7.15:** *Composite .avi Screenshot from MATLAB*

---

<sup>2</sup>The inverted image appears smoother than it should, due to jpg compression.



# Chapter 8

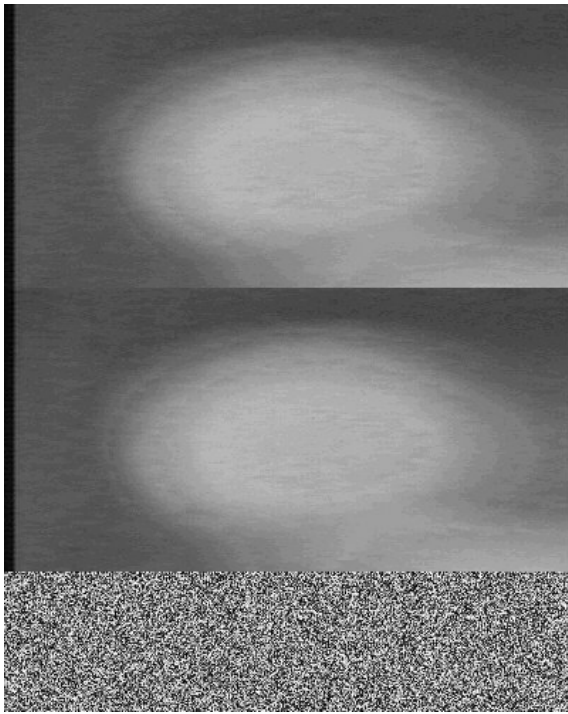
## Hardware Integration

This chapter discusses the practical integration of all the components in the tracking system (video processor, DSP and PC), and highlights the problems that occurred.

The initial step was to ensure all programmable logic devices were operational. This was done by writing some simple code to flash each PLDs general purpose LED. Secondly, the software to write data from RAM to the PC was implemented, providing a way to verify data in the RAM banks. Hereafter, the data routers were programmed and each datapath tested individually. Data were input by hardwiring the DSP port (on the video processor) to fixed values.

Before testing data from the video decoders, the datapath (and byte latching process) from the video encoders were tested, also by asserting fixed values on the bus and reading them to RAM. To initialize the video encoders and decoders an I<sup>2</sup>C kernel was written for the controller FPGA. Each I<sup>2</sup>C device possessed a bit in the I<sup>2</sup>C control register directly wired to an output pin. These output pins were connected to LEDs, and I<sup>2</sup>C communication tested by initially clearing this bit, writing I<sup>2</sup>C setup data and then setting the bit again. For each device, this would activate the LED and then deactivate it after I<sup>2</sup>C initialization, confirming the process. Hereafter a single line was written to RAM, the data dumped to PC. The data appeared to resemble a single video line and an entire frame was written to RAM. The data were dumped to PC and the image displayed in MATLAB (Figure 8.1). The data appeared to be valid, as the picture:

- was coherent and smooth, not random.
- was interlaced.
- had a constant dark region (on the left).



**Figure 8.1:** *First Image Read from RAM*

- contained an area of random pixels, where the input image did not fill the available RAM.

The image was recaptured (after focussing the camera!), the interlaced lines restored and the unused area in RAM discarded and replotted in Figure 8.2.

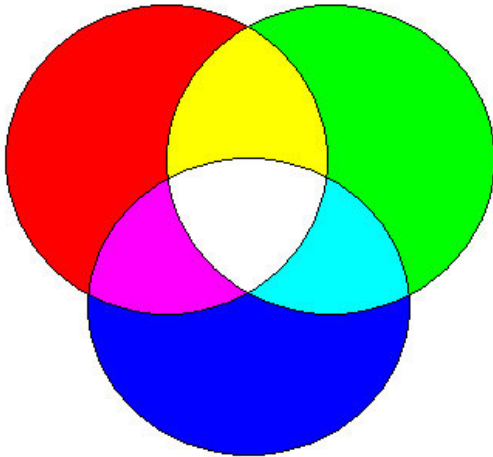
To confirm that the different RGB colours were written to RAM correctly, a simple RGB image (Figure 8.3) was captured, each colour channel read to the PC individually and the RGB image reconstructed (Figure 8.4), so confirming that the frames in RAM were in fact it's constituent colours.

This confirmed the input data path to the video processor. The output data path to the DSP was tested by writing an image to PC (via the serial port), then writing a specific bit (of the image) to the DSP bus and comparing it to the value on the PC. Again, the output video encoders could not be tested statically, but by synchronizing the output data flow with the input (from the video decoder) and tying their reference signals together, synchronicity was ensured and video output correctly.

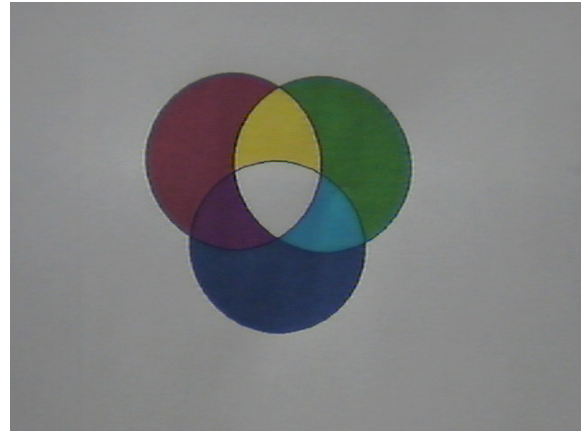
Communication between the DSP and the video processor proved much more problematic. If the DSP and video connector were connected, the video processor would freeze randomly and need to be reset. To identify the problem an external bus consisting of 5 LEDs was connected to the external port on the FPGA, with the LEDs counting in se-



**Figure 8.2:** *Second Image Read from RAM*



**Figure 8.3:** *RGB Test Image*



**Figure 8.4:** *RGB Test Image Read From RAM*

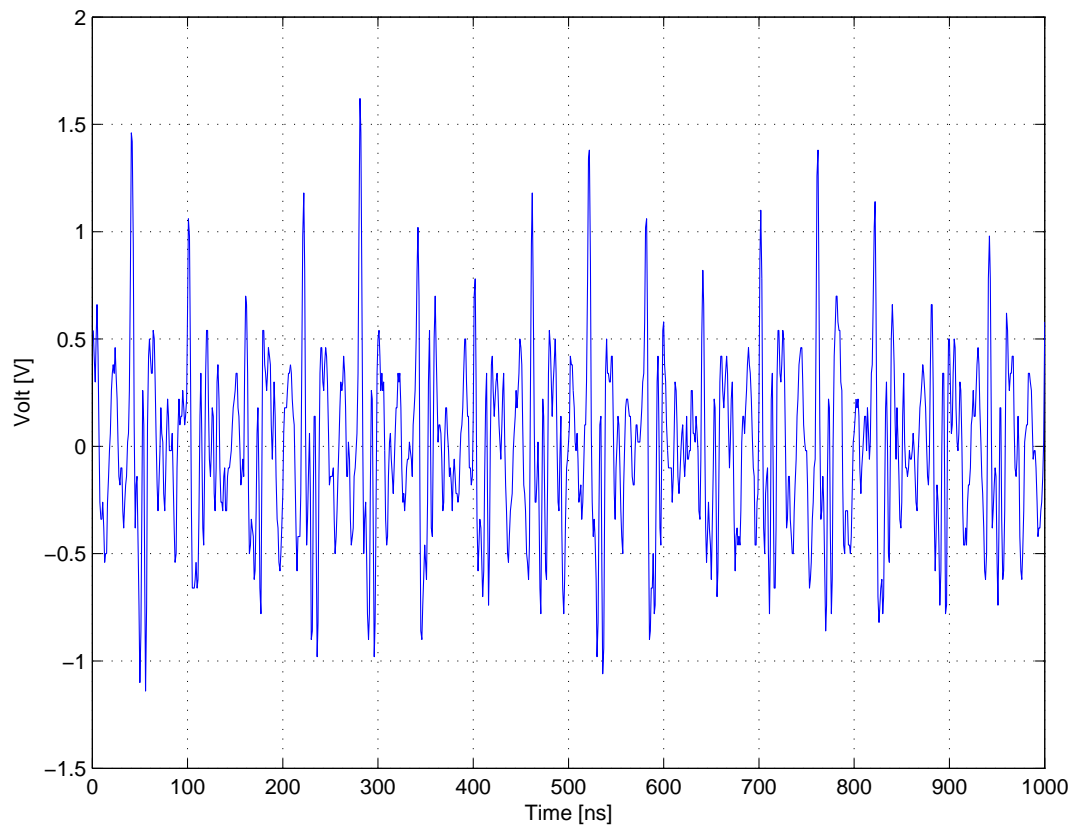
quence as the FPGA state machine stepped through it's states. It was found that the FPGA stalled in random states, sometimes even in LED count values not even occurring in the states.

The focus was shifted to measuring the hardware in an attempt to locate the problem. Figure 8.5 shows one of many measurements taken, the voltage difference between the video processor power ground and the DSP digital ground.

The graph illustrates *very* high voltage peaks at the clock frequency. This voltage oscillation between two ground pins is a phenomena known as *ground bounce* and is caused by the non-zero resistance and inductance of the device, capacitance of the load and the edge rate coupled with the large amount of pins switching simultaneously [15]. This caused the FPGA state machine to enter illegal states from which it could not recover as well as read high and low glitches on inputs that are not actually changing. This also resulted in crosstalk between tracks, manifesting as ghost images on the display (Figure 8.6).

These problems were solved in three ways:

- The pins were set to a slow slew rate in the Quartus Compiler.
- The DSP ground and video processor ground were connected with a large conductor of very low inductance and resistance.
- The VHDL state machine encoding was set to the 'minimal bits' option. All undefined states were explicitly defined and caused the state machine to restart from



**Figure 8.5:** *Voltage Between Power Ground and DSP Digital Ground*



**Figure 8.6:** *Ghost Images due to Crosstalk*

the first state.

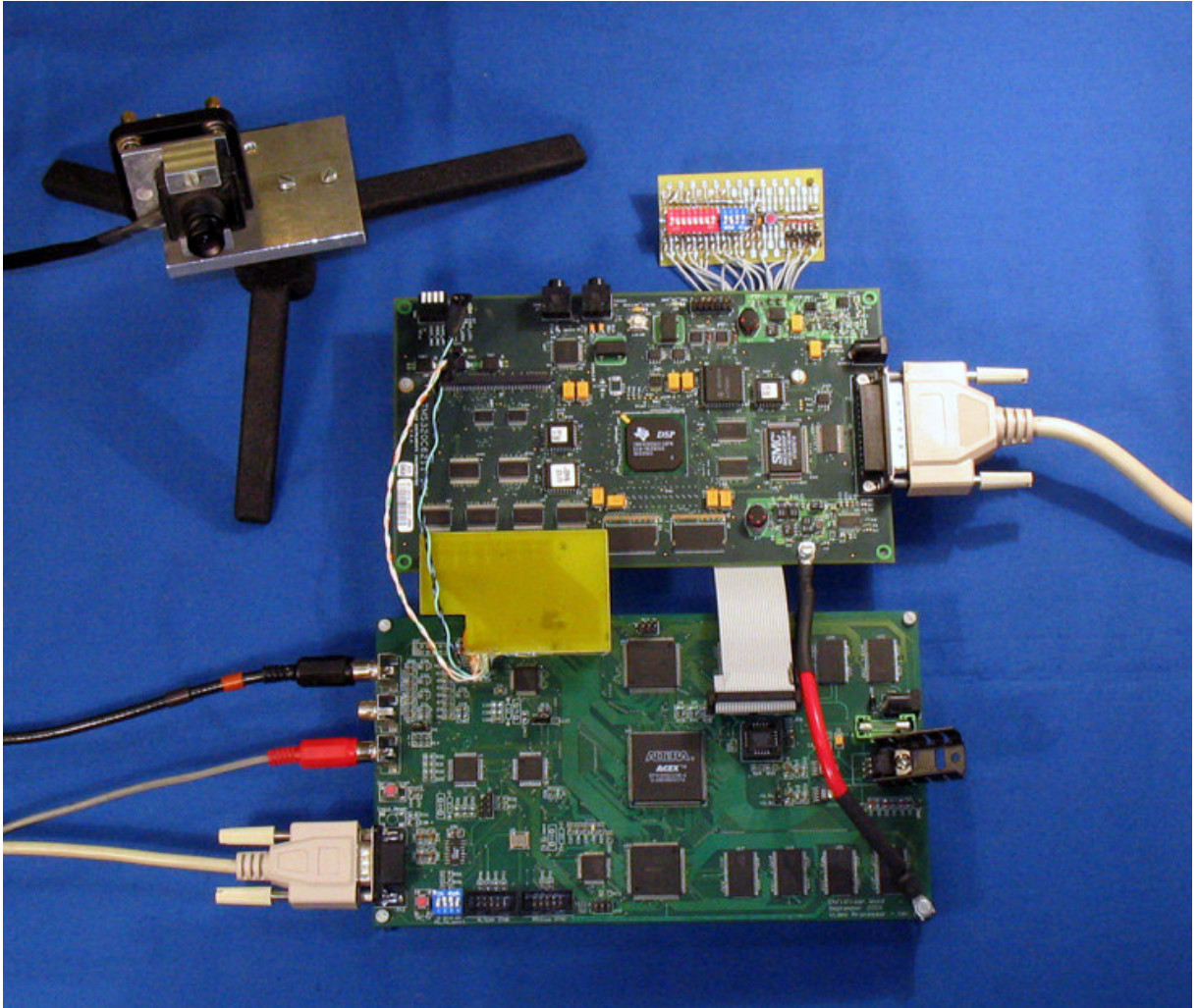
This solved all problems and the video processor functioned perfectly without stalling or generating ghost images.

Data reads to the DSP were tested by asserting a fixed value on the DSP's EMIF and checking the value of the EMIF register using the PC interface, Code Composer Studio. DSP data writes to the EMIF were checked by writing data to RAM, then dumping the RAM to the PC and confirming data entries. It was at this point that it was realized that the DSP EMIF could not read data at the desired frequency and the design methodology changed. Initially it was experimented with reading minimal data to the DSP and serially writing only the centroid position back to the video processor, with the tracking overlay created in the FPGA. This worked (although some bit errors occurred in the serial transmission of the centroid location), but severely limited the tracking algorithm performance as very little data could be transmitted.

The design was then changed to reduce the frame rate and read/write data to/from the DSP instead of reading the next frame.

Tracking algorithms were tested by initially tracking a LED placed on black cardboard - a simple point source on a dark background. As soon as the point source could be followed by the DSP, the framework for implementing other algorithms was established and colour transforms and more complex tracking algorithms were implemented.

The complete tracking system - camera, video processor and DSP - is shown in Figure 8.7.



**Figure 8.7:** *The Complete Tracking System*

# Chapter 9

## Tests and Results

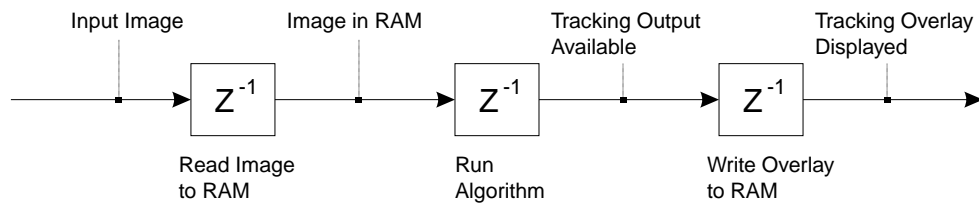
This chapter is the culmination of all the previous chapters, and discusses the testing of the complete system in hardware and the results obtained.

### 9.1 Characterizing the System

The output of any tracking system - the position of the target on the input image - does not correspond exactly with the position of the real target due to the finite processing time required. For any practical application, the delays inherent in the structure of the system need to be characterized.

An analog camera, outputting TV lines, may even produce an image where certain parts of the image are delayed with respect to the other. In the camera used this was not the case, as the camera employed a colour CCD to capture data, where frames can be viewed as discreet snapshots at the frame rate, 25 fps. The processing time of the video decoder (SAA7111A) (the time between input CVBS and VPO output) is negligible, and typically  $6.6\mu\text{s}$ . The output video encoder processing time is of the same order, typically  $2\mu\text{s}$ , and also negligible. What is not negligible though, is the time taken to implement the tracking algorithms and display the tracking overlay. Taking the frame rate as the sample time, this is illustrated in Figure 9.1.

It can be seen that the target location is available two instances (two time delays, or two frames) after the input is read, and the location overlaid another time delay later. If the target location were outputted to a pointing device, this implies that the tracker would lag the target by two sample periods. In a practical system a controller would be design to correct the position error.



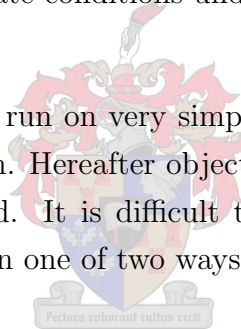
**Figure 9.1:** *Time Delays in Tracking System*

## 9.2 Laboratory Tests

As no physical pointing device was employed, no real position or velocity error could be measured - this would be dependant on the tracking control system.

The errors of the system are largely due to noise on the input signal and algorithms that can not really be quantitatively measured. To judge the tracking performance, many experiments were done to evaluate conditions under which the algorithms would fail and when they work well.

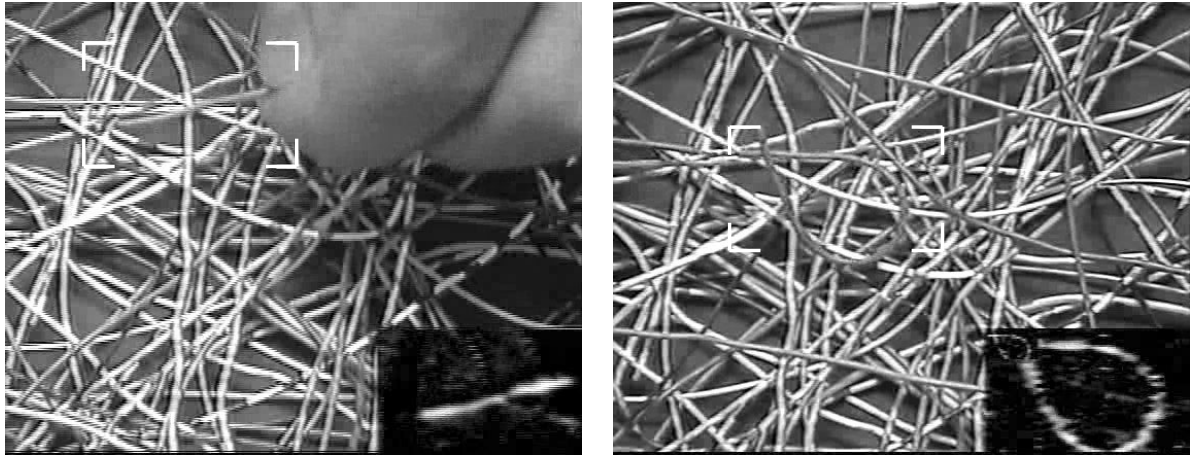
Initial laboratory tests were run on very simple, bright objects, to confirm the basic operation of the tracking system. Hereafter objects with much less contrast with respect to the background were tracked. It is difficult to convey the results of a video based system on paper. This is done in one of two ways:



- Taking snapshots of key moments in a track.
- Taking evenly spaced snapshots of a track to approximate a tracking sequence.

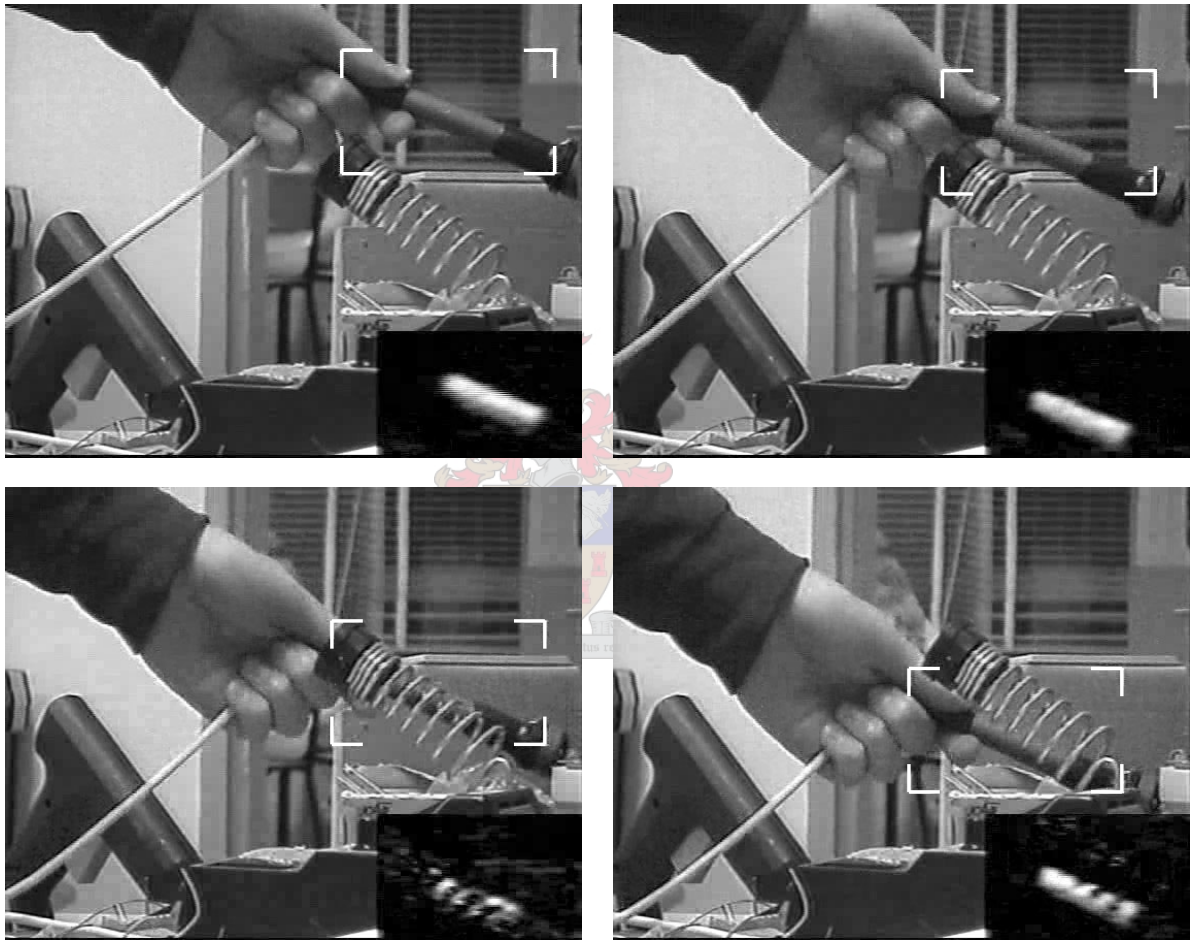
The first laboratory test involved following a blue wire moving against a backdrop of a cluttered desk of differently coloured wires. This is a scenario where conventional grayscale tracking would fail with simple tracking algorithms. Figure 9.2 illustrates two experiments. The output of the colour transform is inserted in the bottom right corner of the video. This gave an indication of tracking confidence. It can be seen that the algorithm has completely extracted the blue wire from the background and tracked its centroid. The result of a correlation tracking experiment is shown in the right frame of Figure 9.2, with the tracking output again in the bottom corner. On the top right corner a small, subsampled image of the correlation mask is also visible. Correlation tracking was successful, but very slow. The  $Z^{-1}$  time delay in Figure 9.1 slowed down to a  $Z^{-3}$  in implementing the two dimensional cross-correlation. Due to the colour transforms, objects were always brighter than their background and correlation tracking proved superfluous.





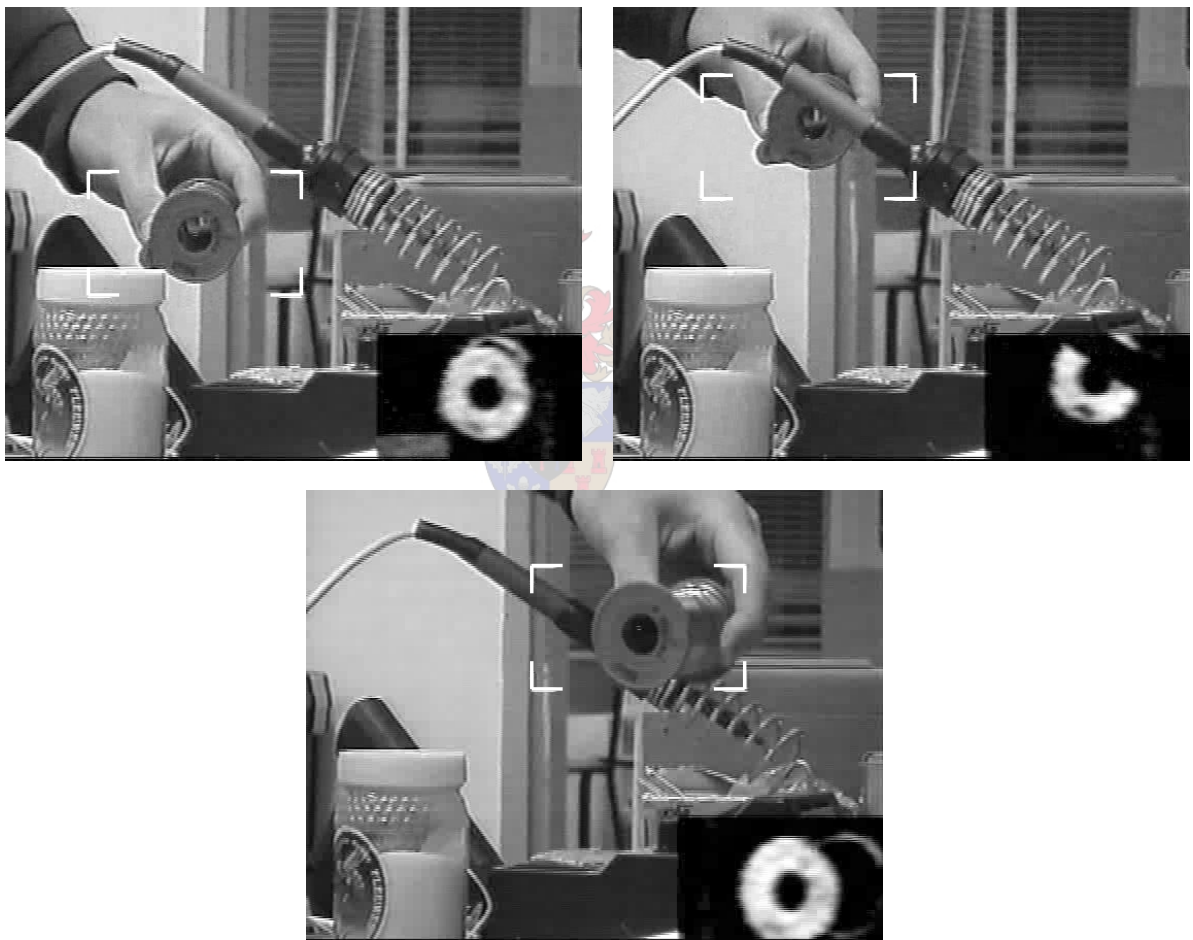
**Figure 9.2:** *Tracking Coloured Wires*

The second experiment involved tracking a soldering iron. This was a natural choice, as the tracking system had been pointing towards the workspace during integration and the soldering station was a common reference point. Figure 9.3 (from left to right, top to bottom) illustrates key moments in the track. The tracking overlay was positioned over the stationary soldering iron, and the colour transform initiated. What is evident from this experiment is the ability to track even if the target has been partially occluded, as can be seen when the soldering iron is moved *behind* the soldering station. This is due to the algorithm focussing on colour, as opposed to shape, and still locking on the partially visible target. Also, due to the position estimation the target track is 'sticky' and would continue to follow the estimated movement even when the target became less visible. This is obviously only valid for short periods of time, but after the target had been identified, the algorithm would require a larger input to move the track to a new target as opposed to just sticking to the original target movement. Position estimation produces a slight lag in the track, visible in the second frame as the target begins accelerating.



**Figure 9.3:** *Tracking a Soldering Iron*

A third simple experiment was to track the soldering wire next to the soldering iron (Figure 9.4). Again, the colour transform and tracking algorithms were initiated by the user after having positioned the overlay over the target of choice - in this case the roll of wire. This again illustrates the ability to track a partially occluded object, visible in the second frame, where the roll of wire is now clearly seen by the algorithm - whereas the soldering iron (bright in the previous track) is now completely ignored. The ability to track a rotating object is also illustrated. Note the consistency in the brightness of the tracking image during rotation and differing reflection from the input image due to the normalization of the colour vectors.



**Figure 9.4:** *Tracking Soldering Wire*

### 9.3 Field Tests

Testing the tracking system outside the laboratory environment was done by recording many tracking scenarios on a video camera and inputting this to the tracking system. This provided the opportunity to repeat the experiments and also record the tracking output on a video recorder - both which would prove problematic if the whole tracking system was moved to the tracking scenario.

To test the tracking of targets against a very cluttered backdrop, video sequences of people walking in a shopping centre were recorded. Potential targets were translating, dynamically changing size and also periodically occluded. Three tracks are shown in Figures 9.5 through 9.7. In each case the input video was paused, the target selected by the user, tracking initiated and then the input video played.

It can be seen that tracking of these targets was very successful. In each of the three cases the targets were tracked accurately for the duration of the input video. It was seen that the colour characterization of the colour transform algorithm is very specific. Spurious targets were completely ignored unless the colours were *very* similar. The track would drift however when targets became occluded for more than two input video frames. This is due to the input image no longer being representative of the target, and the centroid of the available target pixels no longer the centroid of the real target. As soon as the full target became visible, the centre of the track would be restored. Also, even if the target were to disappear for a frame, the algorithm was 'sticky' enough to remain tracking.

Consider the image in the tracking window in Figure 9.5. This illustrates the validity of the unimodal target assumption. In many of the windows, the target was partially occluded and distinctly separate parts visible. By assuming that these 'parts' all belonged to the same target and taking the average location of all pixels above the threshold, the centre of the target is still tracked. The averaging operation inherent in calculating the mean value of the pixels operates as a low pass filter and smooths the perceived centroid trajectory.

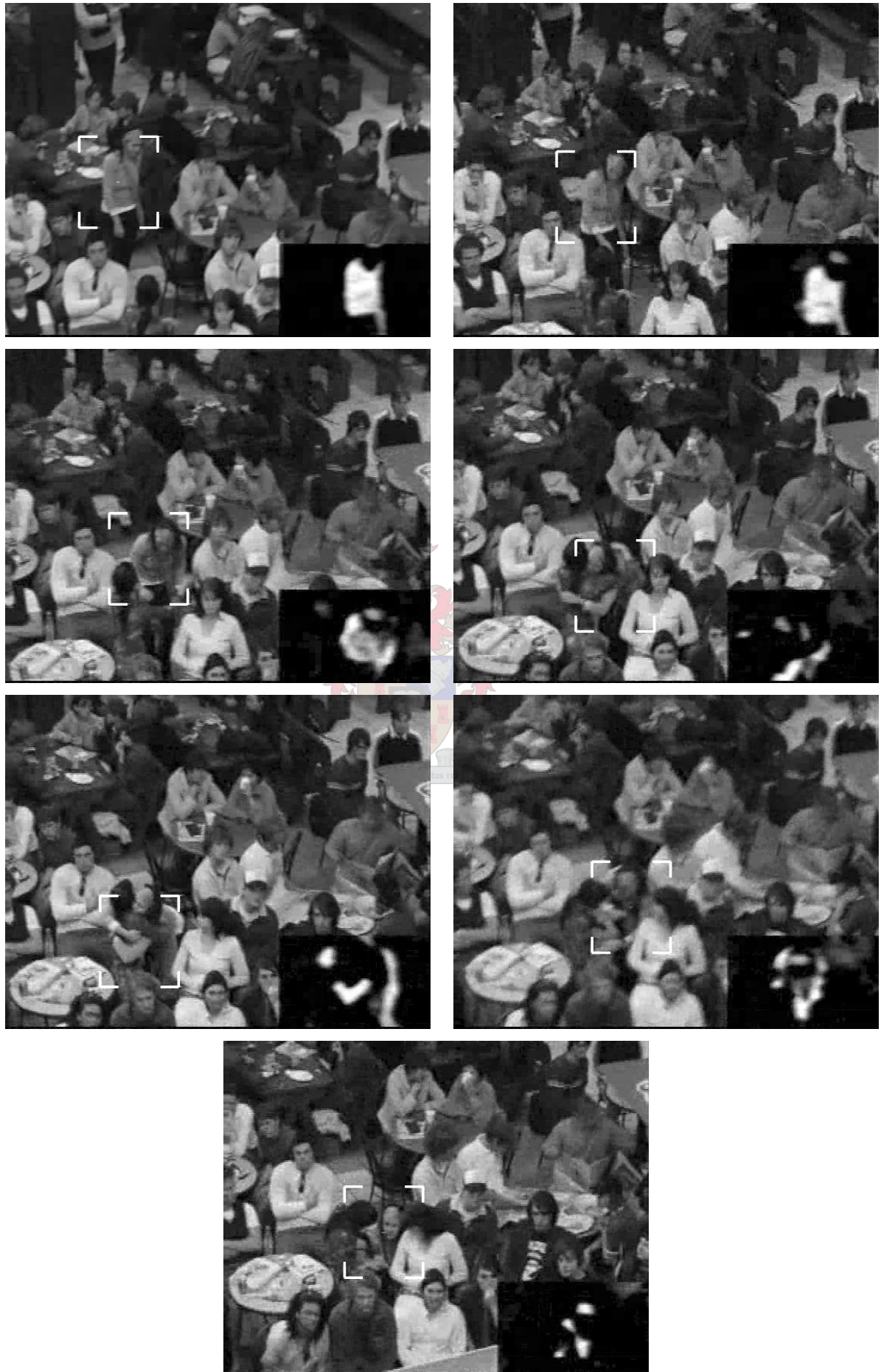


Figure 9.5: *Tracking Humans in Cluttered Environments - 1*



**Figure 9.6:** *Tracking Humans in Cluttered Environments - 2*



Figure 9.7: *Tracking Humans in Cluttered Environments - 3*

Some experiments were also done in tracking moving vehicles. This was problematic, as from any practical vantage point, moving vehicles would expand and/or shrink significantly during the duration of the experiment, as is shown in Figure 9.8. This made the choice of the tracking window problematic.

- If the tracking window were too big, too many background pixels would be included in the target definition. This implies that the target colour vector would not be defined optimally and the difference vector would be very small.
- If the tracking window were too small only a portion of the target would be tracked and the tracker would drift within the boundaries of the actual target. This can be seen in the last frame in Figure 9.8.

Nevertheless, tracking was still successful, as can be seen in Figure 9.8, where a vehicle is confidently tracked whilst changing lanes and moving between other vehicles.



**Figure 9.8:** *Tracking a Moving Vehicle - 1*



# Chapter 10

## Conclusions and Recommendations

### 10.1 Conclusions

This thesis was an investigation into multi-spectral tracking - with emphasis placed on enhancing target vs non-target contrast. It is concluded that:

- Colour transforms can successfully be used in tracking by enhancing the contrast between a target and its background.
- Perceived colour is a function of surface reflectance, viewing geometry and colour of ambient light. Different normalizations exist to remove these dependencies, but must be treated with caution. No single normalization removes all these dependencies, and the optimal one needs to be used. As the most dominating variable in extracting a target from the background is the illumination level, pixel based brightness normalization is the best approach.
- For optimal target vs non-target contrast the target needs to be specified very accurately. Any inclusion of non-target pixels in the target specification will reduce the size of the difference vector, and thus target vs background contrast.
- Colour transforms using the difference vector are efficient, as after initial preprocessing the colour transform is a simple weighted combination of the inputs.

Colour transforms are however not the final answer in tracking an object. These transforms can be used to increase the separability of a target and its background, but are only one part of the tracking system and should be viewed as such. Centroid tracking is ideally suited to multi-spectral tracking, as the colour transformations output an image where the target will always be brighter than its background.

Personally, I learnt much about system integration and managing complex devices interacting with each other. In designing the video processor, much was learnt regarding managing the dataflow in a complex device. I feel this project has equipped me in a large variety of fields, from PCB design, sensor and system design and integration and the efficient realtime implementation of signal processing algorithms.

## 10.2 Recommendations

The scope of this project was very broad and much interesting study still remains, resulting in the following recommendations:

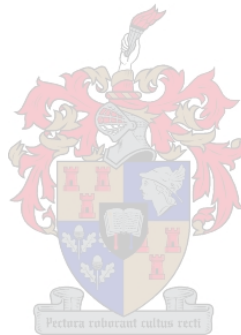
- Colour transforms were not optimal. This was due to inaccurate target specification and limiting the target to a single colour. By specifying targets more accurately and representing targets (and backgrounds) by more than one colour vector more optimal transforms can be devised.
- By representing targets by more than one colour vector *and* separating a target into different subtargets, much more robust tracking can be achieved. This would be done by tracking portions of the target individually as well as referencing their relative positions. This is analogous to tracking a group of objects in formation - if tracking confidence in one object is lost, its relative position with the other would secure tracking.
- Different clustering algorithms should be investigated to separate target and non-target classes. Nonlinear extensions to clustering theory could be promising and warrant further investigation.

# Bibliography

- [1] “Relating HSV to RGB.” [www.planetmirror.com/pub/gtg/node52.html](http://www.planetmirror.com/pub/gtg/node52.html). 2000.
- [2] “Digital Broadcasting Australia Online Forum.”  
<http://www.dtvforum.info/lofiversion/index.php/t530.html>.  
September 2003.
- [3] “RGB Colour Model.” <http://en.wikipedia.org/wiki/RGB>. November 2004.
- [4] ALTERA. *IEEE 1149.1 (JTAG) Boundary-Scan Testing in Altera Devices*,  
September 2000.
- [5] BAR-SHALOM, Y., *Estimation and Tracking: Principles, Techniques, and Software*. John Wiley & Sons Inc, June 2001.
- [6] BAR-SHALOM, Y., *Estimation With Applications to Tracking and Navigation*.  
John Wiley & Sons Inc, June 2001.
- [7] BOURKE, P., “Cross Correlation.”  
<http://astronomy.swin.edu.au/pbourke/analysis/correlate/>. August 1996.
- [8] CROWLEY, J. and BERARD, F., “Multi-Modal Tracking of Faces for Video Communications.” *Conference on Computer Vision and Pattern Recognition (CVPR '97), Puerto Rico*, June 1997, p. p640.
- [9] EAGLE TECHNOLOGY. *E-526SP Camera Operating Manual*.
- [10] EUROPEAN BROADCASTING UNION. *EBU Iinterface for 625Line Digital Video Signals at the 4:2:2 Level of CCIR Recommendation 601*, January 1992.
- [11] FINLAYSON, G., CHATTERJEE, S., and FUNT, B., “Color angular indexing.” *The Fourth European Conference on Computer Vision*, 1996, pp. 16–27.
- [12] FINLAYSON, G., SCHIELE, B., and CROWLEY, J., “Using Colour for Image Indexing.” 1998.

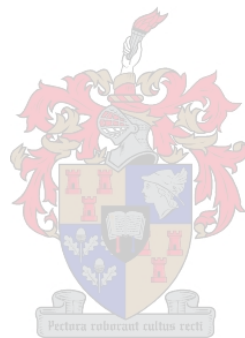
- [13] GENTILE, C., CAMPS, O., and SZNAIER, M., "Segmentation for Robust Tracking in the Presence of Severe Occlusion." *IEEE Transactions on Image Processing*, February 2004, Vol. 13, No. 2, pp. 166–178.
- [14] KUHN, K., "Conventional Analog Television - An Introduction." <http://www.ee.washington.edu/conselec/CE/kuhn/ntsc/95x4.htm>.
- [15] LATTICE. *Ground Bounce*, October 2001.
- [16] LEE, D. and EDDINS, S., "Tracking Objects: Acquiring and Analyzing Image Sequences in MATLAB." *MATLAB - News and Notes*, November 2003, pp. 1–2.
- [17] LEICHTER, I., LINDENBAUM, M., and RIVLIN, E., "A Probabilistic Framework for Combining Tracking Algorithms." tech. rep., Israel Institute of Technology - Department of Computer Science, March 2004.
- [18] LIN, S. and LEE, S., "Using chromaticity distributions and eigenspace analysis for pose-, illumination- and specularly-invariant recognition of 3d object.." *CVPR97*, 1997, pp. 426–431.
- [19] MAYES, L., "TV Synchronization." <http://graffiti.virgin.net/ljmayeres.mal/var/tvsync.htm>. August 2003.
- [20] MIN, C., KYONG, I., and LEONID, V., "Does Colorspace Transformation Make Any Difference on Skin Detection?." tech. rep., Sixth IEEE Workshop on Applications of Computer Vision, December 2002.
- [21] MOKHTARIAN, F. and ABBASI, S., "Matching Shapes With Self-Intersections: Application to Leaf Classification." *IEEE Transactions on Image Processing*, May 2004, Vol. 13, No. 5, pp. 653–661.
- [22] OTSU, N., "A threshold selection method from gray-level histograms.." *IEEE Transactions. Systems, Man, and Cybernetics*, 1979, Vol. 9, No. 1, No. 1, pp. 62–66.
- [23] PHILLIPS. *SAA7111A Enhanced Video Input Processor Datasheet*, May 1998.
- [24] PHILLIPS. *The I2C Bus Specification*, January 2000.
- [25] SCHIELE, B. and WAIBUL, A., "Gaze Tracking Based On Face Colour." *IWAGFR '95- International Workshop on Face and Gesture Recognition, Zurich*, July 1995.
- [26] SCHWARDT, L., "Manipulating Feature Space." tech. rep., University of Stellenbosch, February 2003.
- [27] SHARP. *Sharp RJ2421 Datasheet*.

- [28] SWAIN, M., *Color Indexing*. PhD thesis, University of Rochester, Department of Computer Science, 1990.
- [29] SWAIN, M. and BALLARD, D., “Color Indexing.” *International Journal of Computer Vision*, 1991.
- [30] TEXAS INSTRUMENTS. *TMS320C6000 EVM Daughterboard Interface*, December 1998.
- [31] TEXAS INSTRUMENTS. *TMS320C6000 Peripherals Reference Guide*, April 1999.
- [32] TI, “TMS320C6211 Product Description.”  
<http://focus.ti.com/docs/prod/folders/print/tms320c6211b.html>. 2004.
- [33] WILSON, D., “RGB/YUV Pixel Conversion.”  
<http://www.fourcc.org/fccyvrgb.php>. June 2004.



# Appendix A

## Schematics and PCB Layouts



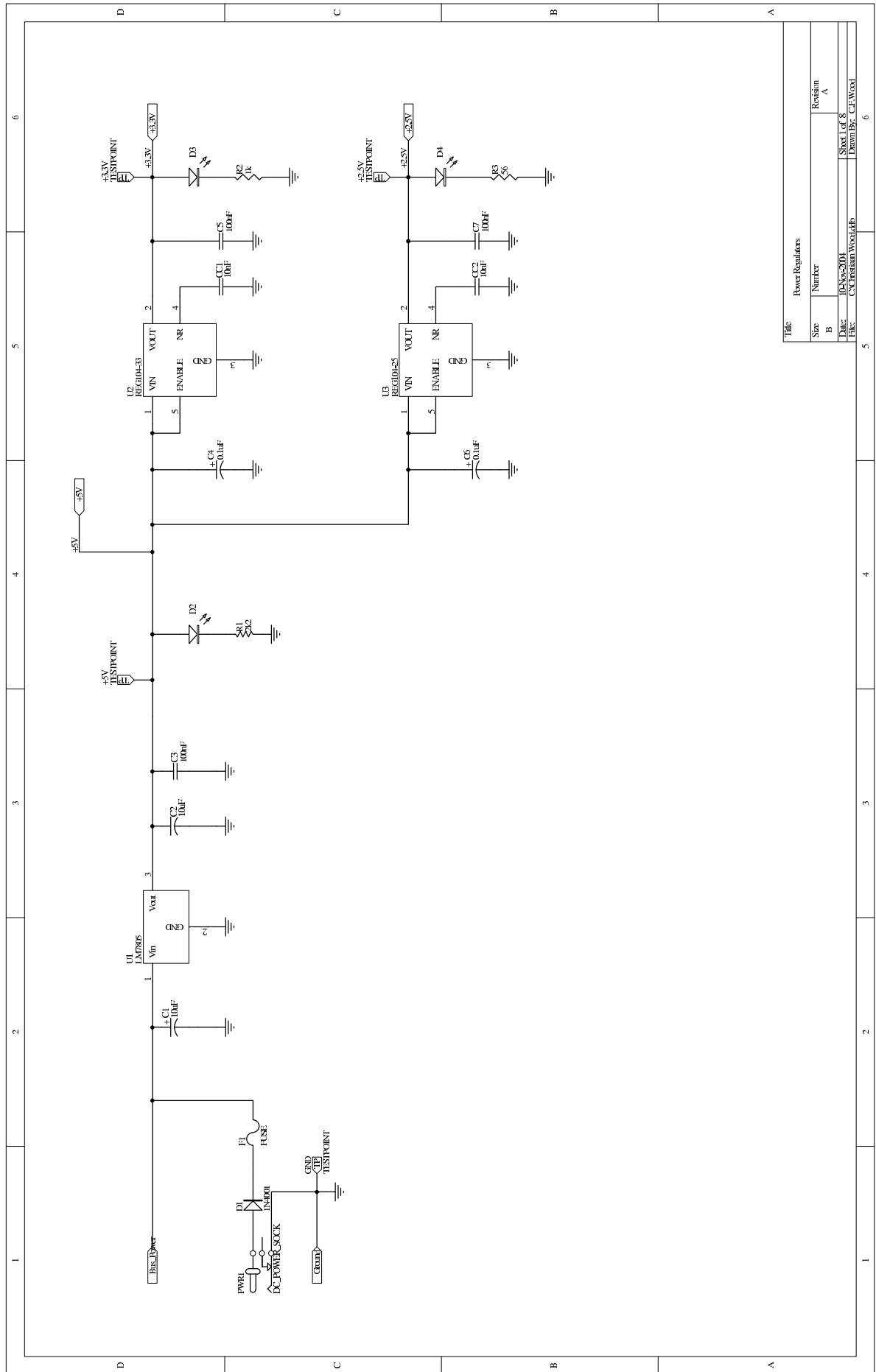


Figure A.1: Video Processor Schematics - Power Regulators

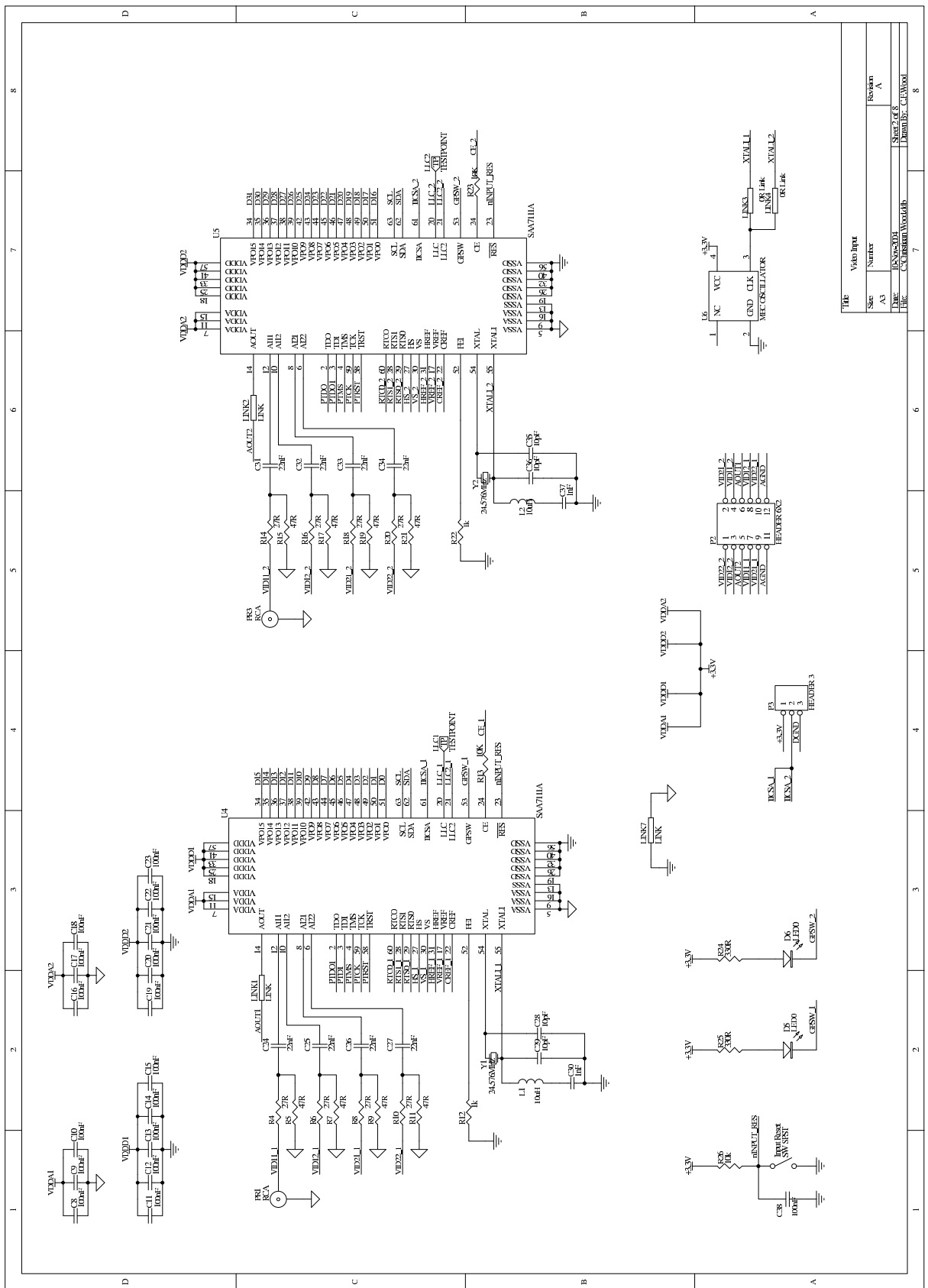


Figure A.2: Video Processor Schematics - Video Input





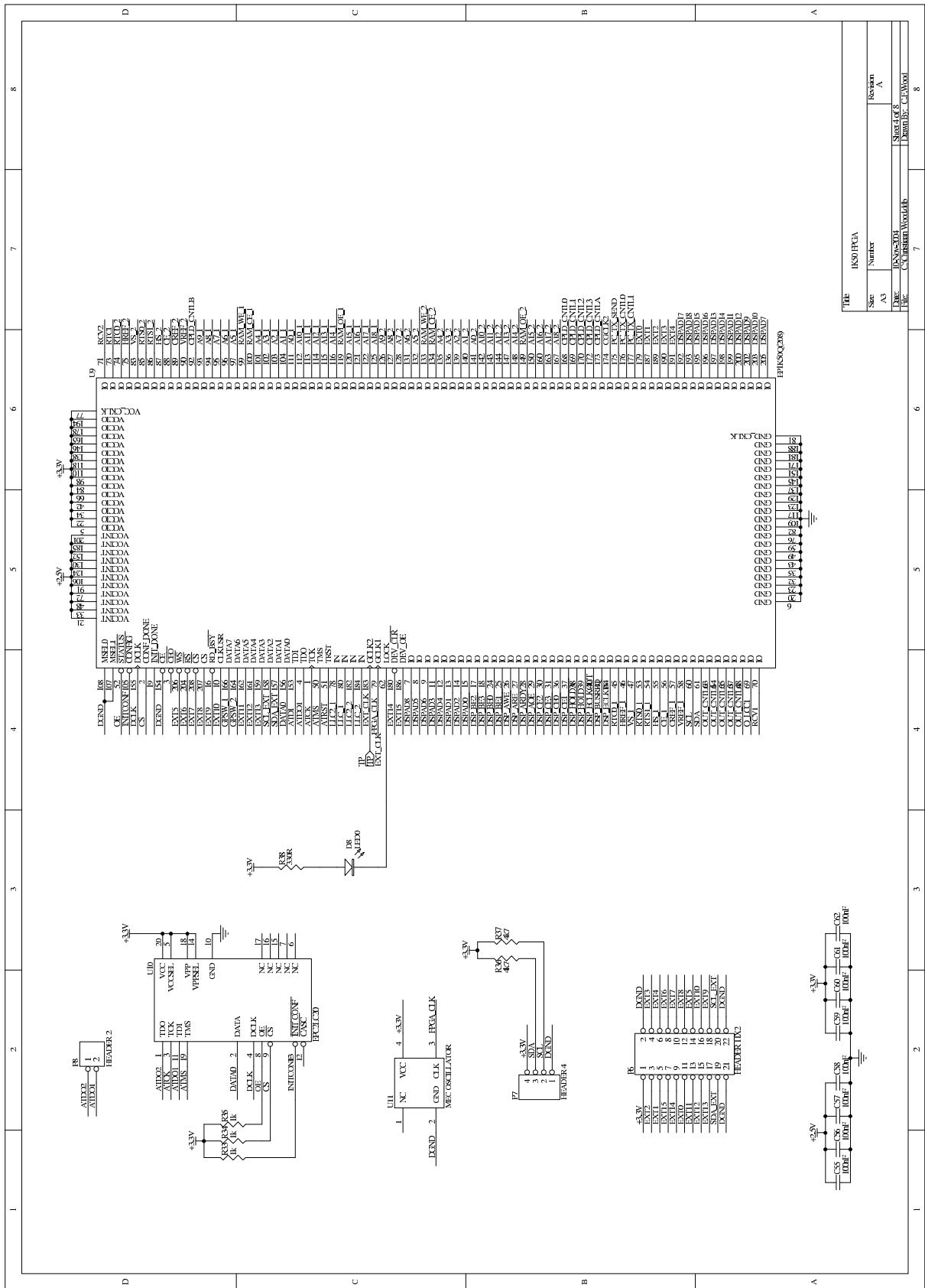


Figure A.4: Video Processor Schematics - 1K50 Control Logic

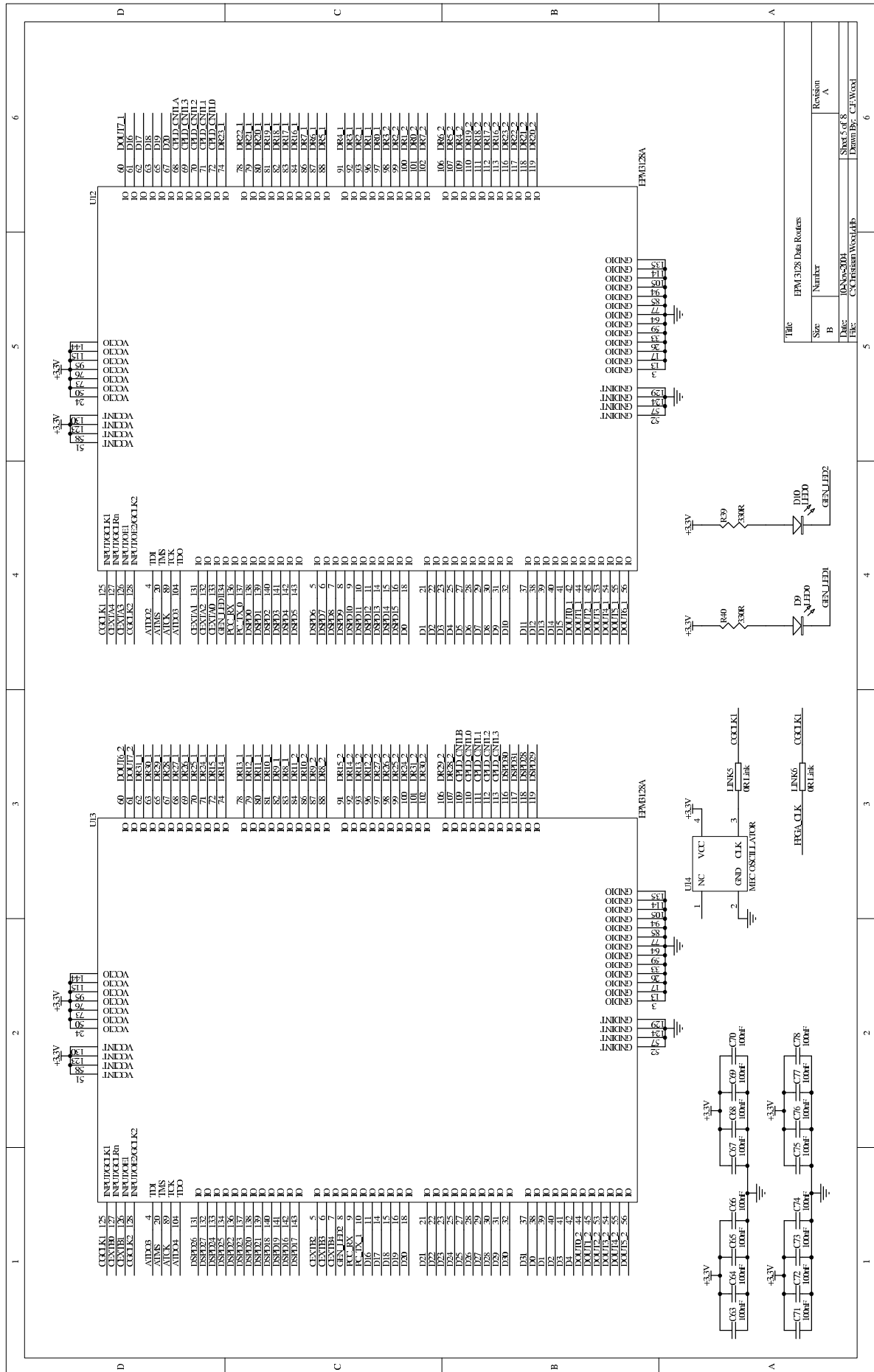


Figure A.5: Video Processor Schematics - CPLD Data Routers

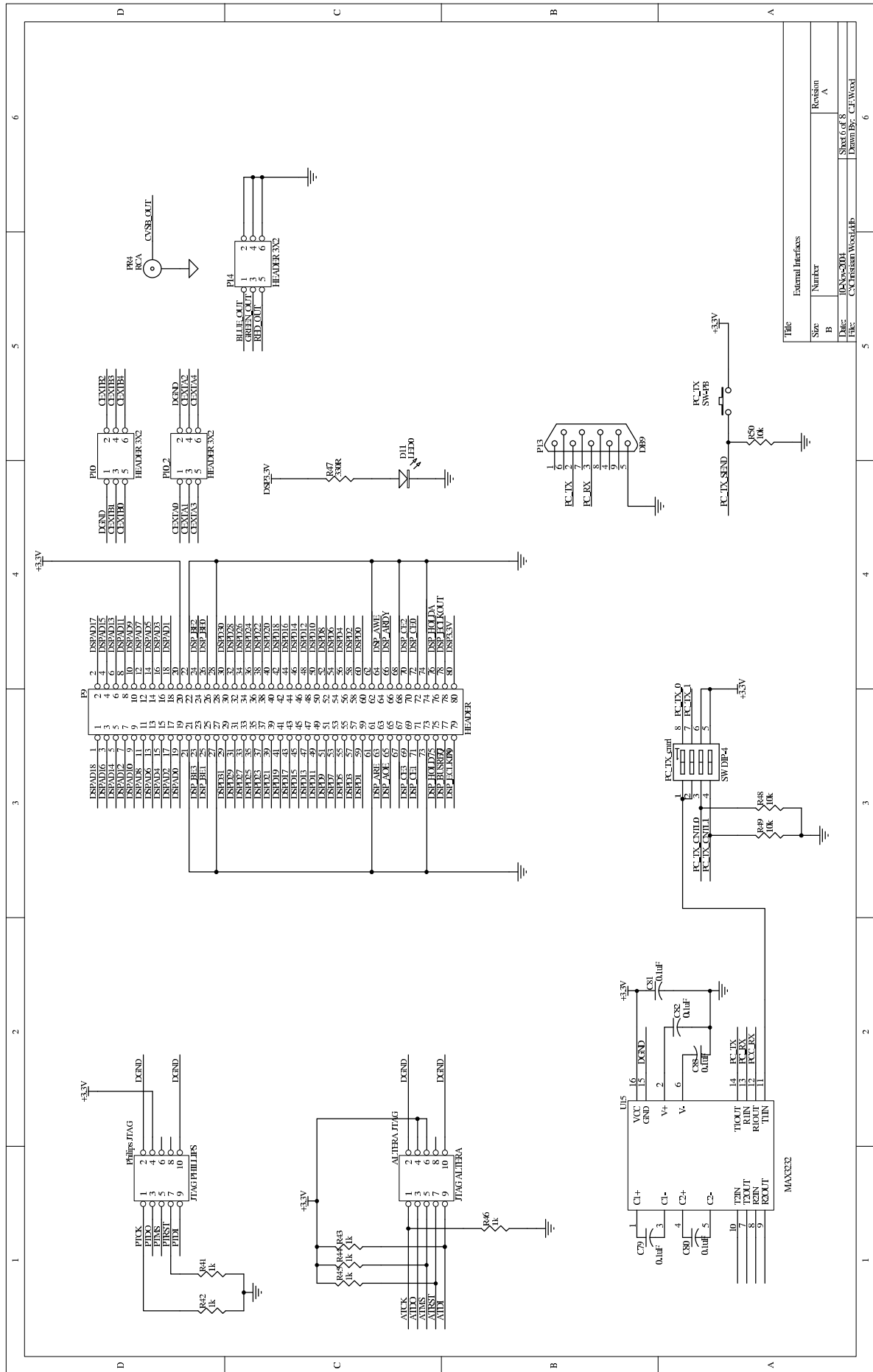
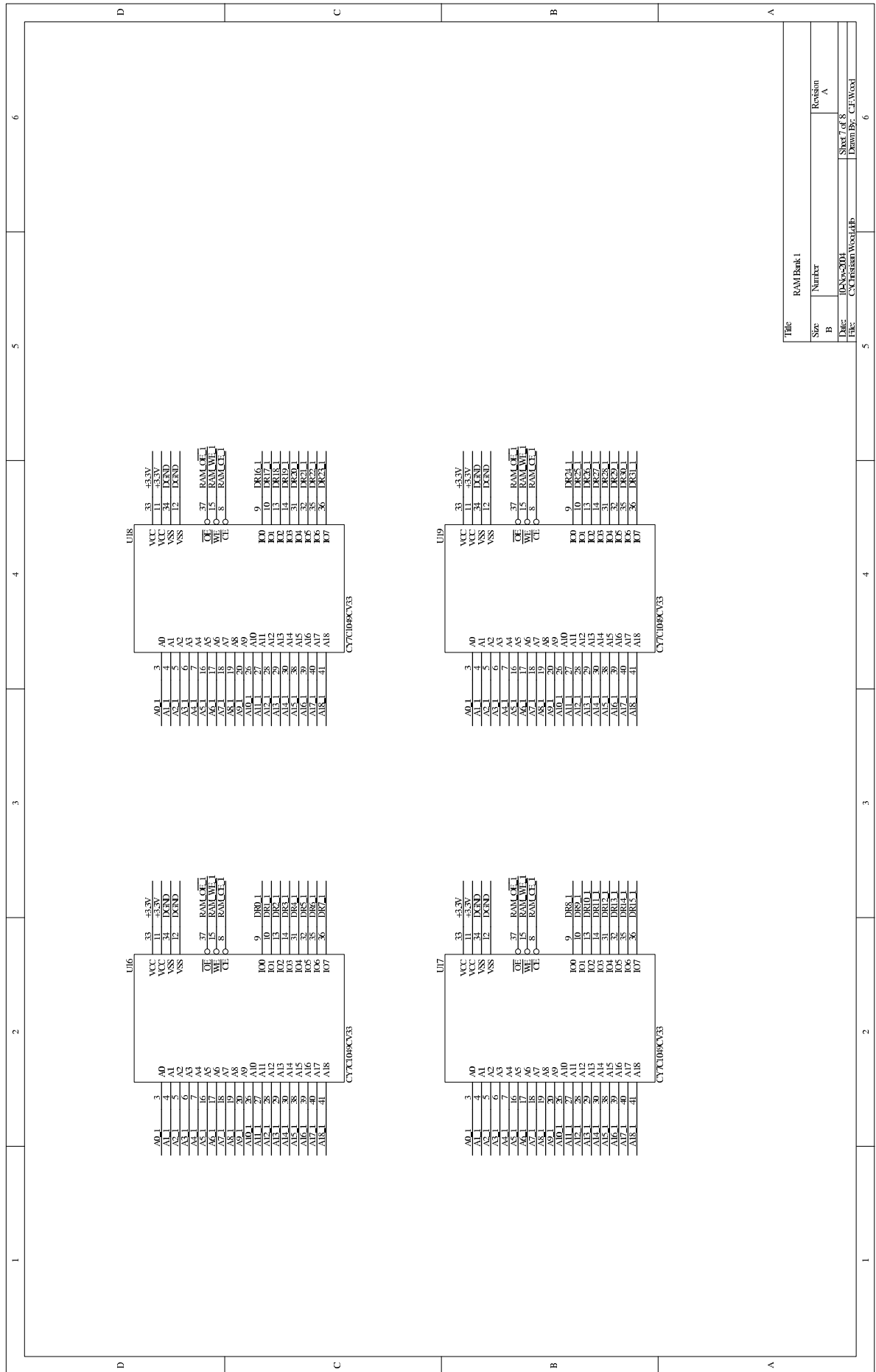


Figure A.6: Video Processor Schematics - External Interfaces



Title		Revision	
RAM Bank 1		A	
Size	Number		
B			
Date:	DESIGNED BY	Sheet 7 of 8	
File:	C:\Christian\Woolf\B	Drawn By: C.F.Woolf	

Figure A.7: Video Processor Schematics - RAM Bank 1

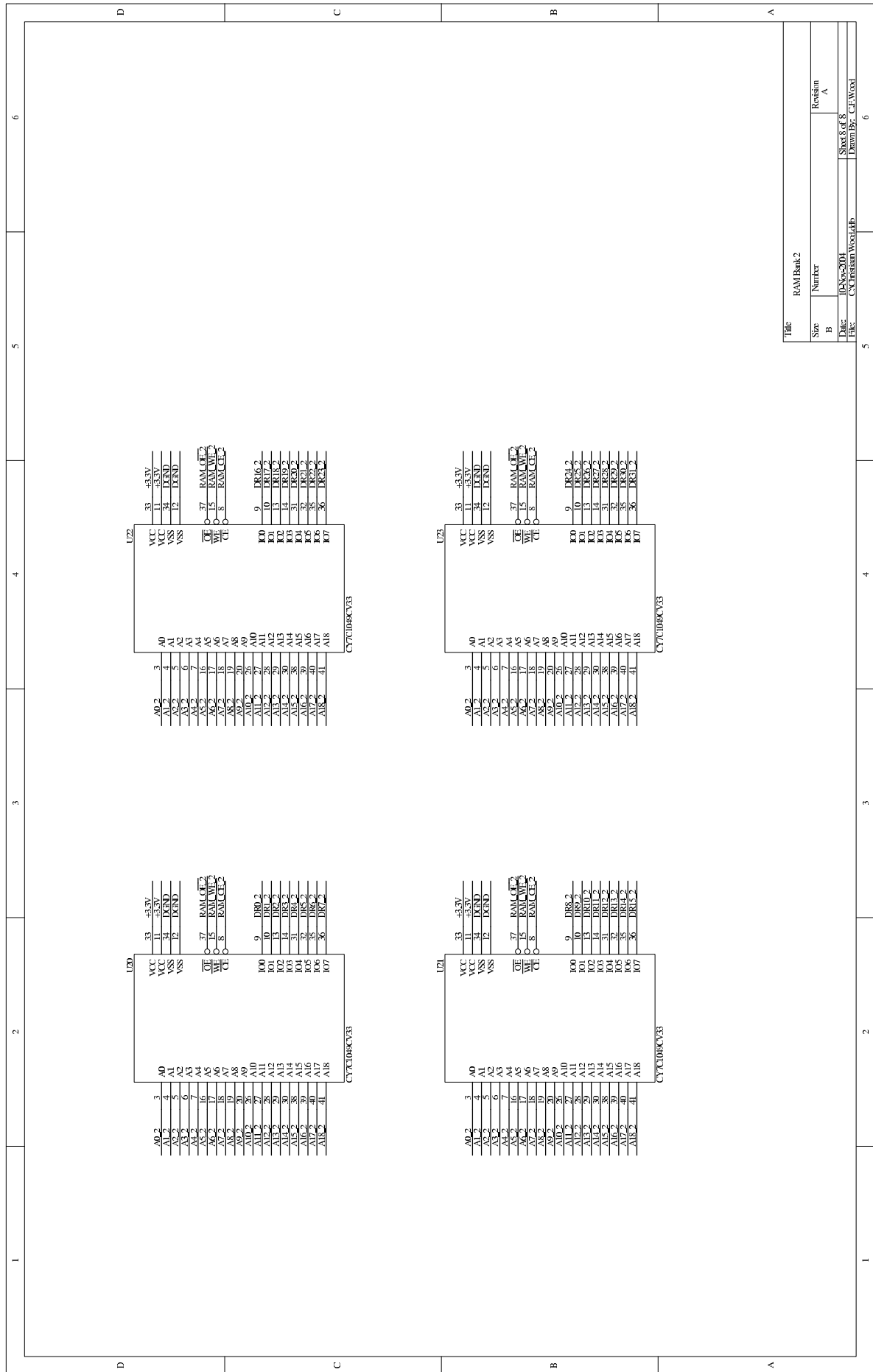


Figure A.8: Video Processor Schematics - RAM Bank 2

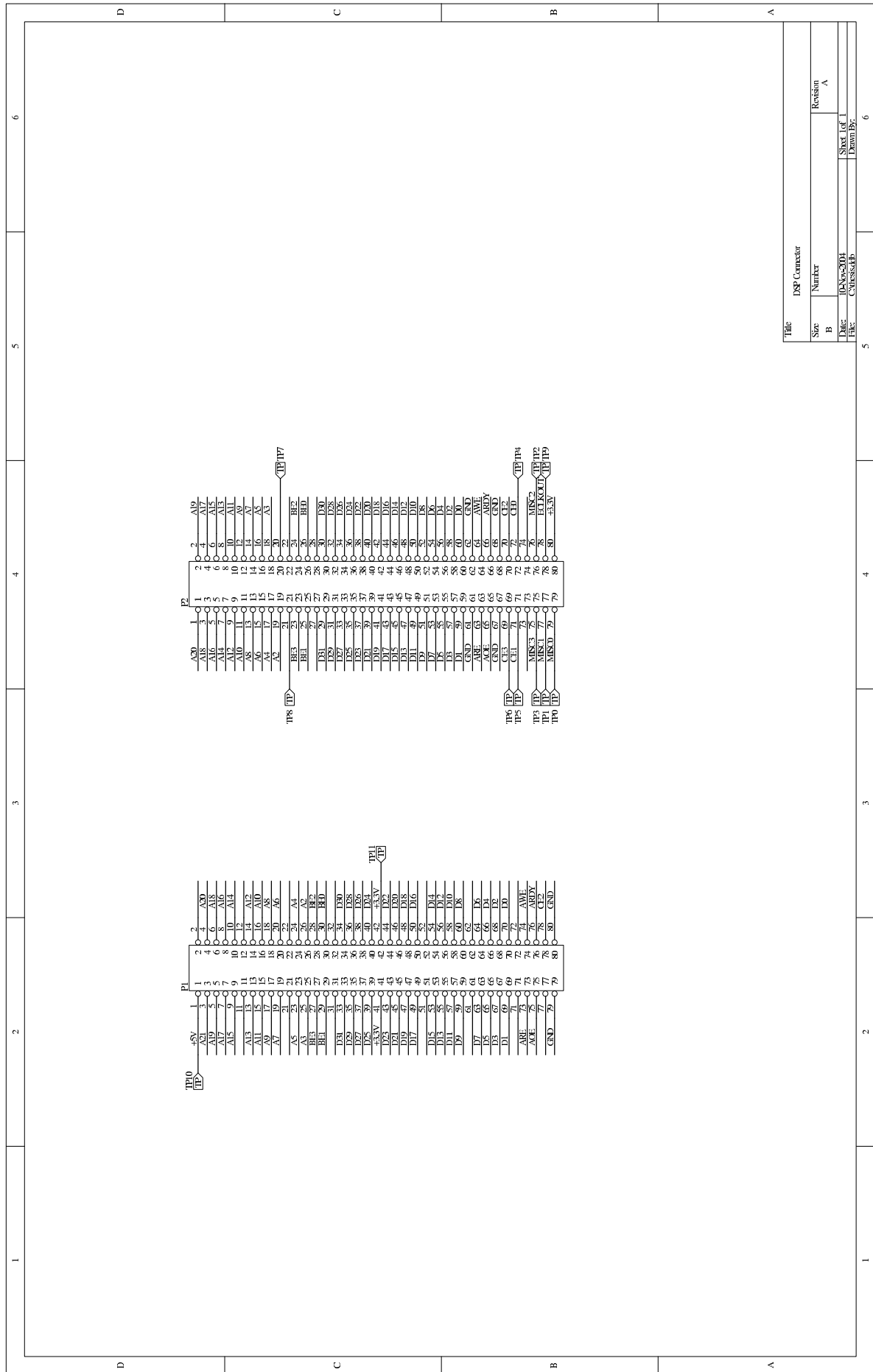


Figure A.9: Video Processor Schematics - DSP Connector

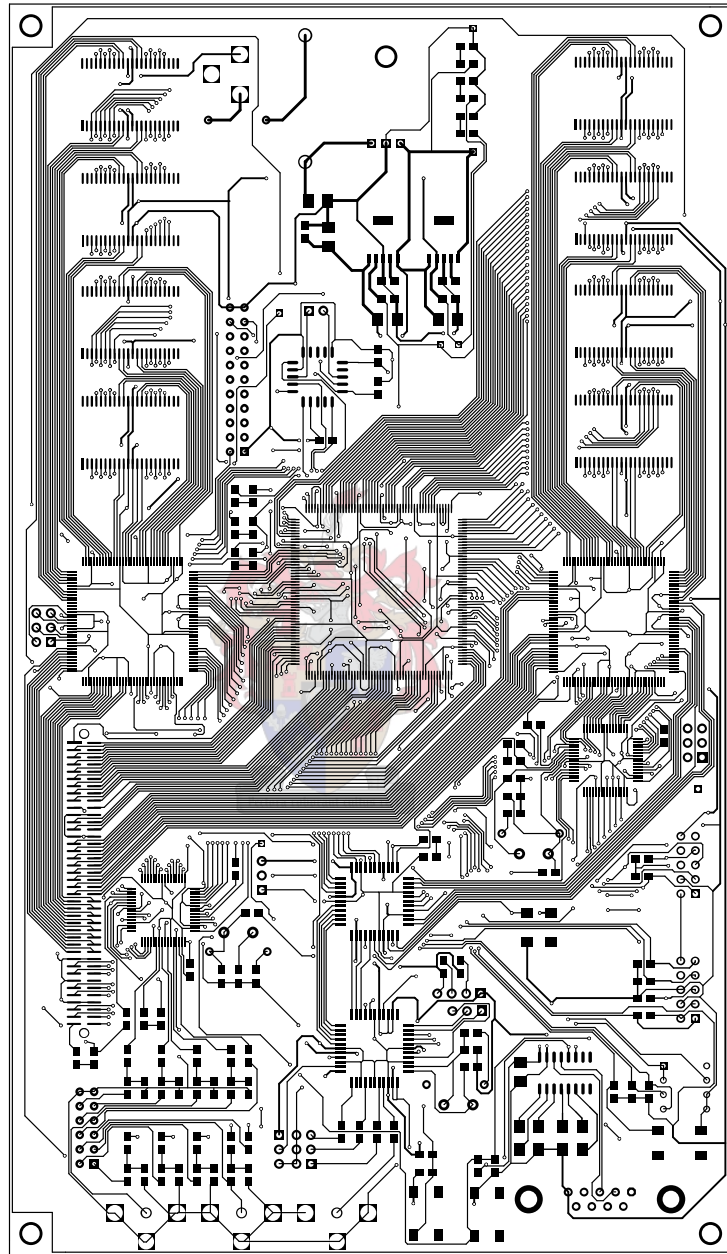
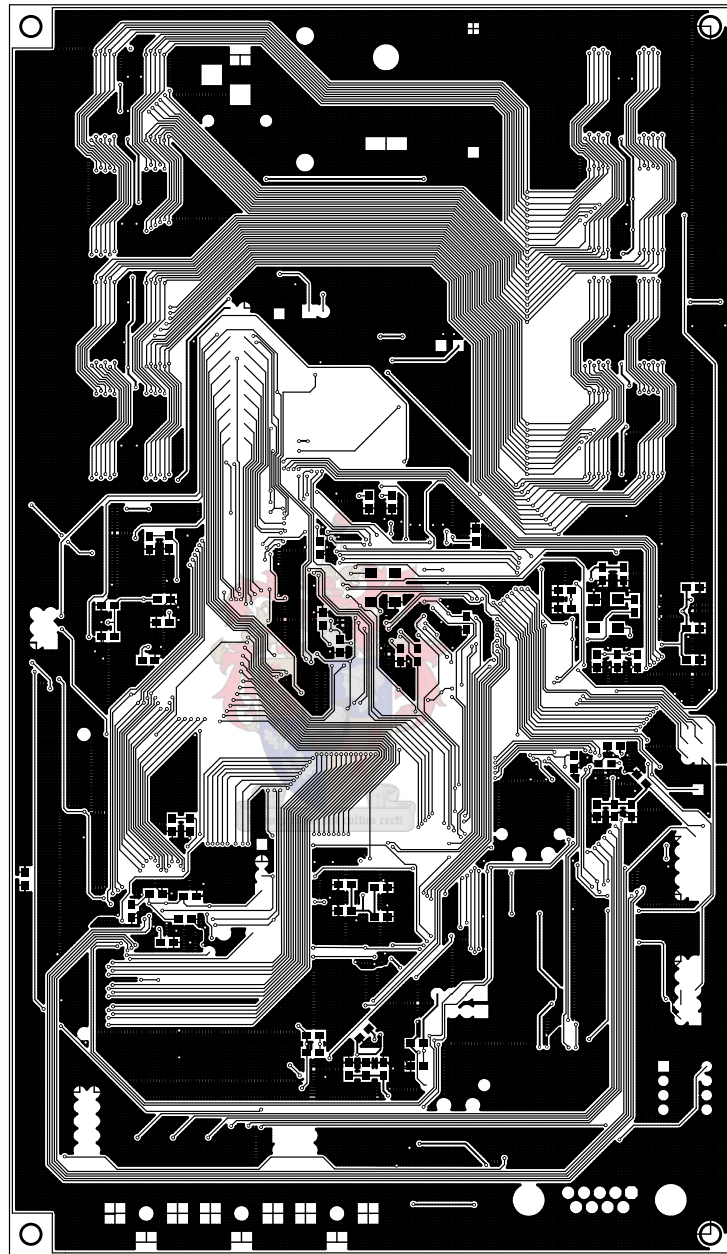
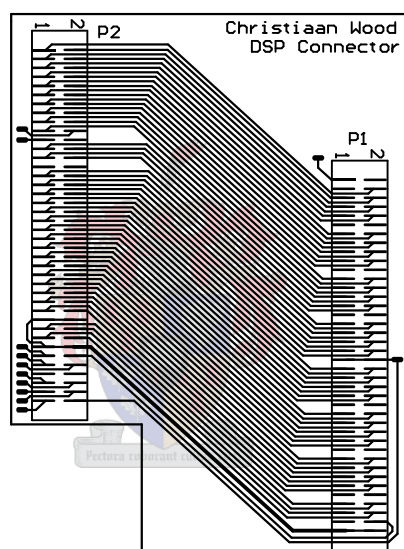


Figure A.10: *Video Processor PCB - Top Layer*





**Figure A.11:** *Video Processor PCB - Bottom Layer*

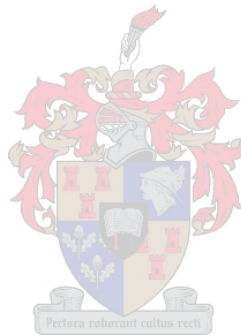


**Figure A.12:** *DSP Connector PCB*

# Appendix B

## Extraneous Data

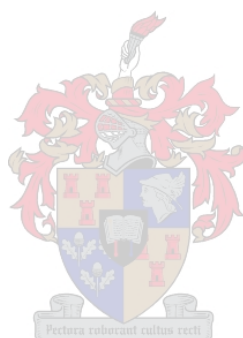
### B.1 SAA7111A Video Decoder Control Registers



Subaddress (HEX)	Function	Value
02H	Analog Input Control 1	11000000
03H	Analog Input Control 2	00100011
04H	Analog Input Control 3	00000000
05H	Analog Input Control 4	00000000
06H	Horizontal Sync Start	11101011
07H	Horizontal Sync Stop	11100000
08H	Sync Control	10001000
09H	Luminance Control	00000001
0AH	Luminance Brightness	10000000
0BH	Luminance Contrast	01000111
0CH	Chrominance Saturation	01000000
0DH	Chroma Hue Control	00000000
0EH	Chrominance Control	00000001
10H	Format/Delay Control	01000000
11H	Output Control 1	10001100
12H	Output Control 2	00001001
13H	Output Control 3	00000000
15H	VBI Data Stream Start	00000000
16H	VBI Data Stream Stop	00000000
17H	MSBs for VBI Control	00000000

**Table B.1:** *SAA7111A For RGB Output Control Registers*

## B.2 SAA7127H Video Encoder Control Registers



Subaddress (HEX)	Function	Value
02H	Analog Input Control 1	11000000
26H	Wide screen signal	00000000
27H	Wide screen signal	00000000
28H	Real-time control, burst start	00100001
29H	Sync reset enable, burst end	00011101
2AH	Copy generation 0	00000000
2BH	Copy generation 1	00000000
2CH	CG enable, copy generation 2	00000000
2DH	Output port control	00001111
38H	Gain luminance for RGB	00011010
39H	Gain colour difference for RGB	00011010
3AH	Input port control 1	00000011
54H	VPS enable, input control 2	00000000
55H	VPS byte 5	00000000
56H	VPS byte 11	00000000
57H	VPS byte 12	00000000
58H	VPS byte 13	00000000
59H	VPS byte 14	00000000
5AH	Chrominance phase	01101011
5BH	Gain U	00000000
5CH	Gain V	00000000
5DH	Gain U MSB, real-time control, black level	00111010
5EH	Gain V MSB, real-time control, blanking level	00110101
5FH	CCR, blanking level VBI	00110101
61H	Standard control	00000110
62H	RTC enable, burst amplitude	00101101

Subaddress (HEX)	Function	Value
63H	Subcarrier 0	11001011
64H	Subcarrier 1	10001010
65H	Subcarrier 2	00001001
66H	Subcarrier 3	00101010
67H	Line 21 odd 0	00000000
68H	Line 21 odd 1	00000000
69H	Line 21 even 0	00000000
6AH	Line 21 even 1	00000000
6BH	RCV port control	00100000
6CH	Trigger control	00111001
6DH	Trigger control	00000000
6EH	Multi control	00000000
6FH	Closed caption, teletext enable	00000000
70H	RCV2 output start	00000000
71H	RCV2 output end	00000000
72H	MSBs RCV2 output	00000000
73H	TTX request H start	01000010
74H	TTX request H delay, length	00000010
75H	CSYNC advance, Vsync shift	00000000
76H	TTX odd request vertical start	00000101
77H	TTX odd request vertical end	00010110
78H	TTX even request vertical start	00000100
79H	TTX even request vertical end	00010110
7AH	First active line	00000000
7BH	Last active line	00101100
7CH	TTX mode, MSB vertical	01000000
7EH	Disable TTX line	00000000
7FH	Disable TTX line	00000000

Table B.2: SAA7127H Control Registers

## B.3 VHDL ROM Memory Initialization File

```

-- my_i2c.mif
-- Christiaan Wood
-- lpm_rom Memory Initialization File

DEPTH = 512;           % Memory depth and width are required  %
WIDTH = 8;            % Enter a decimal number                %
ADDRESS_RADIX = DEC;  % Address and value radixes are optional%
DATA_RADIX = BIN;     % Enter BIN, DEC, HEX, or OCT; unless      %
                    % otherwise specified, radixes = HEX      %

content
begin
-----
-- I2C Setup Data for SAA7111A U5
-- sub address 02H Analog input contr 1
0  : 01001010;
1  : 00000010;
2  : 11000000;
-- sub address 03H Analog input contr 2
3  : 01001010;
4  : 00000011;
5  : 00100011;
-- sub address 04H Analog input contr 3
6  : 01001010;
7  : 00000100;
8  : 00000000;
-- sub address 05H Analog input contr 4
9  : 01001010;
10 : 00000101;
11 : 00000000;
-- sub address 06H Horizontal sync start
12 : 01001010;
13 : 00000110;
14 : 11101011;
-- sub address 07H Horizontal sync stop
15 : 01001010;
16 : 00000111;
17 : 11100000;
-- sub address 08H Sync control
18 : 01001010;
19 : 00001000;
20 : 10001000;
-- sub address 09H Luminance control
21 : 01001010;

```





```
22 : 00001001;
23 : 00000001;
-- sub address 0AH Luminance brightness
24 : 01001010;
25 : 00001010;
26 : 10000000;
-- sub address 0BH Luminance contrast
27 : 01001010;
28 : 00001011;
29 : 01000111;
-- sub address 0CH Chroma saturation
30 : 01001010;
31 : 00001100;
32 : 01000000;
-- sub address 0DH Chroma Hue control
33 : 01001010;
34 : 00001101;
35 : 00000000;
-- sub address 0EH Chroma control
36 : 01001010;
37 : 00001110;
38 : 00000001;
-- sub address 10H Format/delay control
39 : 01001010;
40 : 00010000;
41 : 01000000;
-- sub address 11H Output control 1
42 : 01001010;
43 : 00010001;
44 : 10001100;
-- sub address 12H Output control 2
45 : 01001010;
46 : 00010010;
47 : 00001001;
-- sub address 13H Output control 3
48 : 01001010;
49 : 00010011;
50 : 00000000;
-- sub address 15H V_GATE1_START
51 : 01001010;
52 : 00010101;
53 : 00000000;
-- sub address 16H V_GATE1_STOP
54 : 01001010;
55 : 00010110;
56 : 00000000;
```



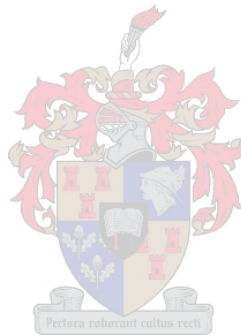
```

-- sub address 17H V_GATE1_MSB
57 : 01001010;
58 : 00010111;
59 : 00000000;
-- sub address 11H Output control 1
60 : 01001010;
61 : 00010001;
62 : 00001100;
-----
-- I2C Setup Data for SAA7127H U7
-- sub address 26H Wide screen signal
63 : 10001000;
64 : 00100110;
65 : 00000000;
-- sub address 27H Wide screen signal
66 : 10001000;
67 : 00100111;
68 : 00000000;
-- sub address 28H Real-time control, burst start
69 : 10001000;
70 : 00101000;
71 : 00100001;
-- sub address 29H Sync reset enable, burst end
72 : 10001000;
73 : 00101001;
74 : 00011101;
-- sub address 2AH Copy generation 0
75 : 10001000;
76 : 00101010;
77 : 00000000;
-- sub address 2BH Copy generation 1
78 : 10001000;
79 : 00101011;
80 : 00000000;
-- sub address 2CH CG enable, copy generation 2
81 : 10001000;
82 : 00101100;
83 : 00000000;
-- sub address 2DH Output port control
84 : 10001000;
85 : 00101101;
86 : 00001111;
-- sub address 38H Gain luminance for RGB
87 : 10001000;
88 : 00111000;
89 : 00011010;

```

```
-- sub address 39H Gain colour difference for RGB
90 : 10001000;
91 : 00111001;
92 : 00011010;
-- sub address 3AH Input port control 1
93 : 10001000;
94 : 00111010;
95 : 00000011;
-- sub address 54H VPS enable, input control 2
96 : 10001000;
97 : 01010100;
98 : 00000001;
-- sub address 55H VPS byte 5
99 : 10001000;
100 : 01010101;
101 : 00000000;
-- sub address 56H VPS byte 11
102 : 10001000;
103 : 01010110;
104 : 00000000;

-- sub address 57H VPS byte 12
105 : 10001000;
106 : 01010111;
107 : 00000000;
-- sub address 58H VPS byte 13
108 : 10001000;
109 : 01011000;
110 : 00000000;
-- sub address 59H VPS byte 14
111 : 10001000;
112 : 01011001;
113 : 00000000;
-- sub address 5AH Chrominance phase
114 : 10001000;
115 : 01011010;
116 : 01101011;
-- sub address 5BH Gain U
117 : 10001000;
118 : 01011011;
119 : 00000000;
-- sub address 5CH Gain V
120 : 10001000;
121 : 01011100;
122 : 00000000;
-- sub address 5DH Gain U MSB, real-time control, black level
```



```
123 : 10001000;
124 : 01011101;
125 : 00111010;
-- sub address 5EH Gain V MSB, real-time control, blanking level
126 : 10001000;
127 : 01011110;
128 : 00110101;
-- sub address 5FH CCR, blanking level VBI
129 : 10001000;
130 : 01011111;
131 : 00110101;
-- sub address 61H Standard control
132 : 10001000;
133 : 01100001;
134 : 00000110;
-- sub address 62H RTC enable, burst amplitude
135 : 10001000;
136 : 01100010;
137 : 00101101;
-- sub address 63H Subcarrier 0
138 : 10001000;
139 : 01100011;
140 : 11001011;
-- sub address 64H Subcarrier 1
141 : 10001000;
142 : 01100100;
143 : 10001010;
-- sub address 65H Subcarrier 2
144 : 10001000;
145 : 01100101;
146 : 00001001;
-- sub address 66H Subcarrier 3
147 : 10001000;
148 : 01100110;
149 : 00101010;
-- sub address 67H Line 21 odd 0
150 : 10001000;
151 : 01100111;
152 : 00000000;
-- sub address 68H Line 21 odd 1
153 : 10001000;
154 : 01101000;
155 : 00000000;
-- sub address 69H Line 21 even 0
156 : 10001000;
157 : 01101001;
```



```
158 : 00000000;
-- sub address 6AH Line 21 even 1
159 : 10001000;
160 : 01101010;
161 : 00000000;
-- sub address 6BH RCV port control
162 : 10001000;
163 : 01101011;
164 : 00100000;
-- sub address 6CH Trigger control
165 : 10001000;
166 : 01101100;
167 : 00111001;
-- sub address 6DH Trigger control
168 : 10001000;
169 : 01101101;
170 : 00000000;
-- sub address 6EH Multi control
171 : 10001000;
172 : 01101110;
173 : 00000000;
-- sub address 6FH Closed caption, teletext enable
174 : 10001000;
175 : 01101111;
176 : 00000000;
-- sub address 70H RCV2 output start
177 : 10001000;
178 : 01110000;
179 : 00000000;
-- sub address 71H RCV2 output end
180 : 10001000;
181 : 01110001;
182 : 00000000;
-- sub address 72H MSBs RCV2 output
183 : 10001000;
184 : 01110010;
185 : 00000000;
-- sub address 73H TTX request H start
186 : 10001000;
187 : 01110011;
188 : 01000010;
-- sub address 74H TTX request H delay, length
189 : 10001000;
190 : 01110100;
191 : 00000010;
-- sub address 75H CSYNC advance, Vsync shift
```



```

192 : 10001000;
193 : 01110101;
194 : 00000000;
-- sub address 76H TTX odd request vertical start
195 : 10001000;
196 : 01110110;
197 : 00000101;
-- sub address 77H TTX odd request vertical end
198 : 10001000;
199 : 01110111;
200 : 00010110;
-- sub address 78H TTX even request vertical start
201 : 10001000;
202 : 01111000;
203 : 00000100;
-- sub address 79H TTX even request vertical end
204 : 10001000;
205 : 01111001;
206 : 00010110;
-- sub address 7AH First active line
207 : 10001000;
208 : 01111010;
209 : 00000000;
-- sub address 7BH Last active line
210 : 10001000;
211 : 01111011;
212 : 00101100;
-- sub address 7CH TTX mode, MSB vertical
213 : 10001000;
214 : 01111100;
215 : 01000000;
-- sub address 7EH Disable TTX line
216 : 10001000;
217 : 01111110;
218 : 00000000;
-- sub address 7FH Disable TTX line
219 : 10001000;
220 : 01111111;
221 : 00000000;
-- sub address 11H
222 : 11111000;
223 : 00010001;
224 : 10001100;
-----
-- I2C Setup Data for SAA7111A U4
-- sub address 02H Analog input contr 1

```

```
225 : 01001000;
226 : 00000010;
227 : 11000000;
-- sub address 03H Analog input contr 2
228 : 01001000;
229 : 00000011;
230 : 00100011;
-- sub address 04H Analog input contr 3
231 : 01001000;
232 : 00000100;
233 : 00000000;
-- sub address 05H Analog input contr 4
234 : 01001000;
235 : 00000101;
236 : 00000000;
-- sub address 06H Horizontal sync start
237 : 01001000;
238 : 00000110;
239 : 11101011;
-- sub address 07H Horizontal sync stop
240 : 01001000;
241 : 00000111;
242 : 11100000;
-- sub address 08H Sync control
243 : 01001000;
244 : 00001000;
245 : 10001000;
-- sub address 09H Luminance control
246 : 01001000;
247 : 00001001;
248 : 00000001;
-- sub address 0AH Luminance brightness
249 : 01001000;
250 : 00001010;
251 : 10000000;
-- sub address 0BH Luminance contrast
252 : 01001000;
253 : 00001011;
254 : 01000111;
-- sub address 0CH Chroma saturation
255 : 01001000;
256 : 00001100;
257 : 01000000;
-- sub address 0DH Chroma Hue control
258 : 01001000;
259 : 00001101;
```



```
260 : 00000000;
-- sub address 0EH Chroma control
261 : 01001000;
262 : 00001110;
263 : 00000001;
-- sub address 10H Format/delay control
264 : 01001000;
265 : 00010000;
266 : 00000000;
-- sub address 11H Output control 1
267 : 01001000;
268 : 00010001;
269 : 10001100;
-- sub address 12H Output control 2
270 : 01001000;
271 : 00010010;
272 : 00001001;
-- sub address 13H Output control 3
273 : 01001000;
274 : 00010011;
275 : 00000000;
-- sub address 15H V_GATE1_START
276 : 01001000;
277 : 00010101;
278 : 00000000;
-- sub address 16H V_GATE1_STOP
279 : 01001000;
280 : 00010110;
281 : 00000000;
-- sub address 17H V_GATE1_MSB
282 : 01001000;
283 : 00010111;
284 : 00000000;
-- sub address 11H Output control 1
285 : 01001000;
286 : 00010001;
287 : 00001100;
end;
```





## B.4 DSP Asynchronous Read/Write Timing Diagrams

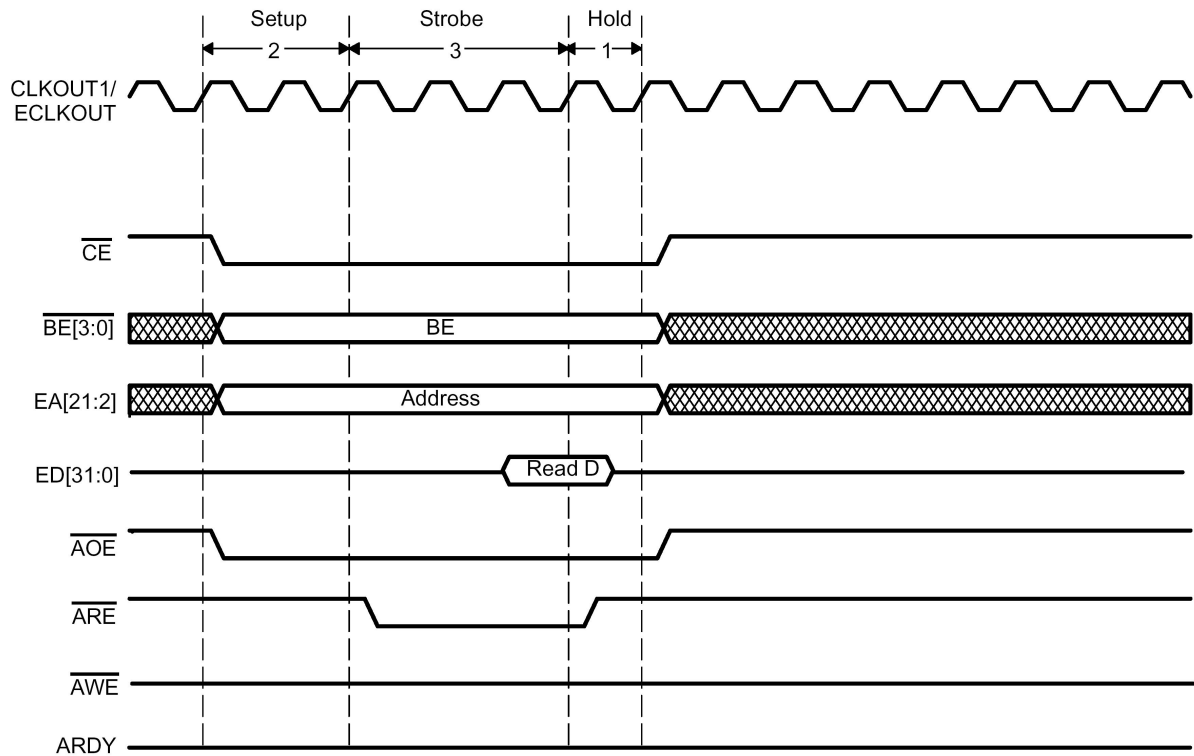


Figure B.1: Asynchronous Read Timing



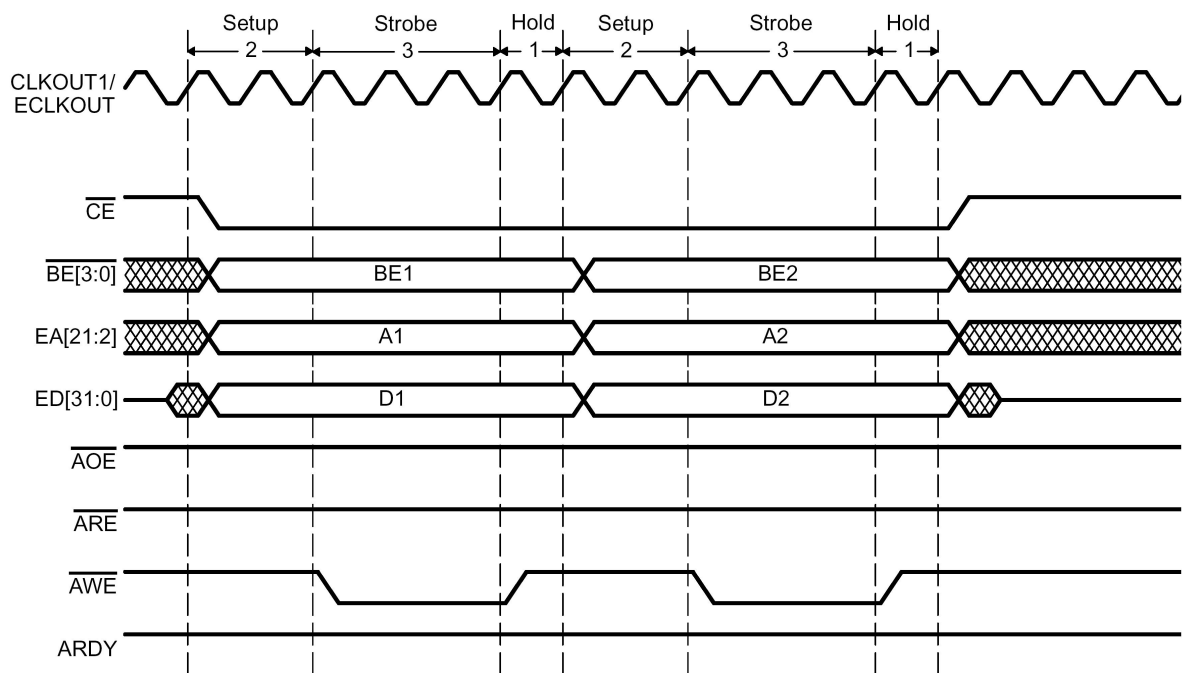
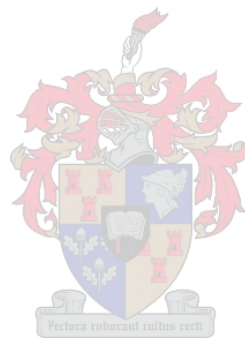


Figure B.2: *Asynchronous Write Timing*



## B.5 VHDL Simulations - Waveform Files



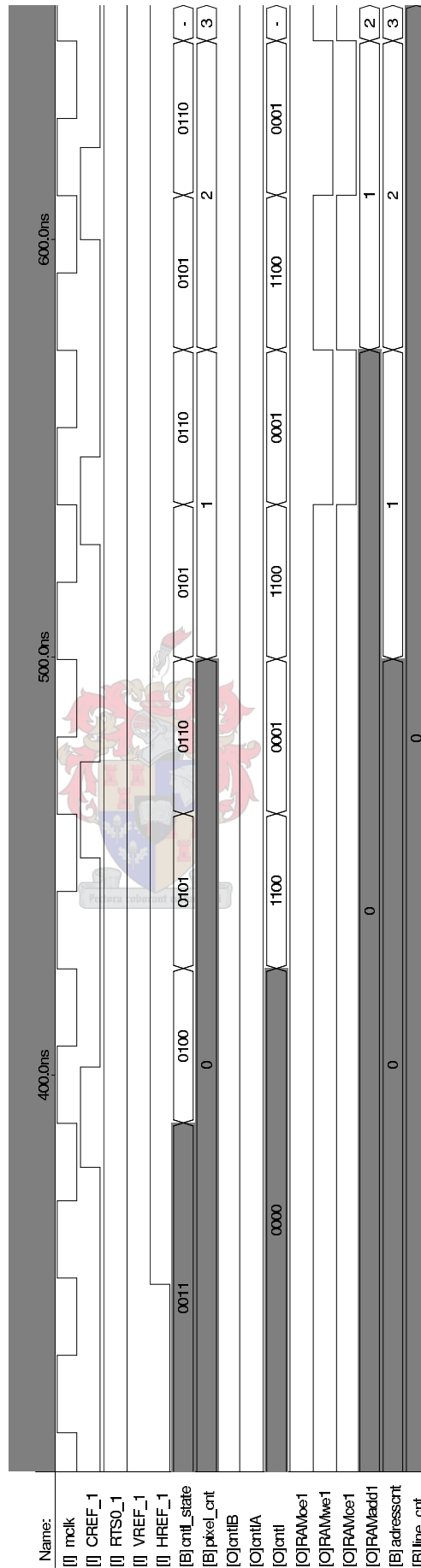


Figure B.3: Control Logic Signals - Camera Data Latch - First Pixel

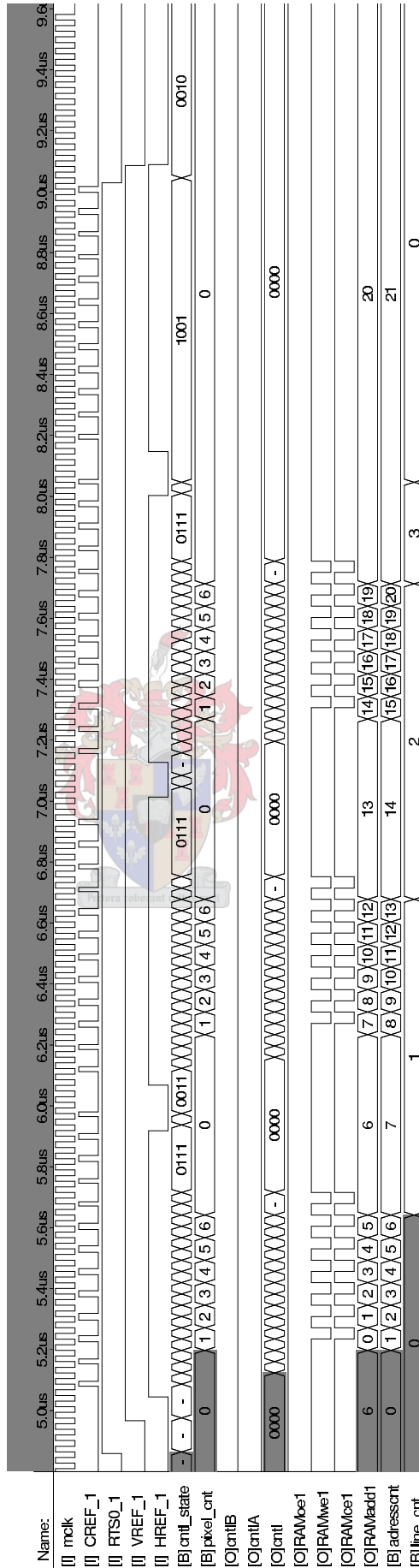


Figure B.4: Control Logic Signals - Camera Data Latch - Multiple Pixels of the same Frame

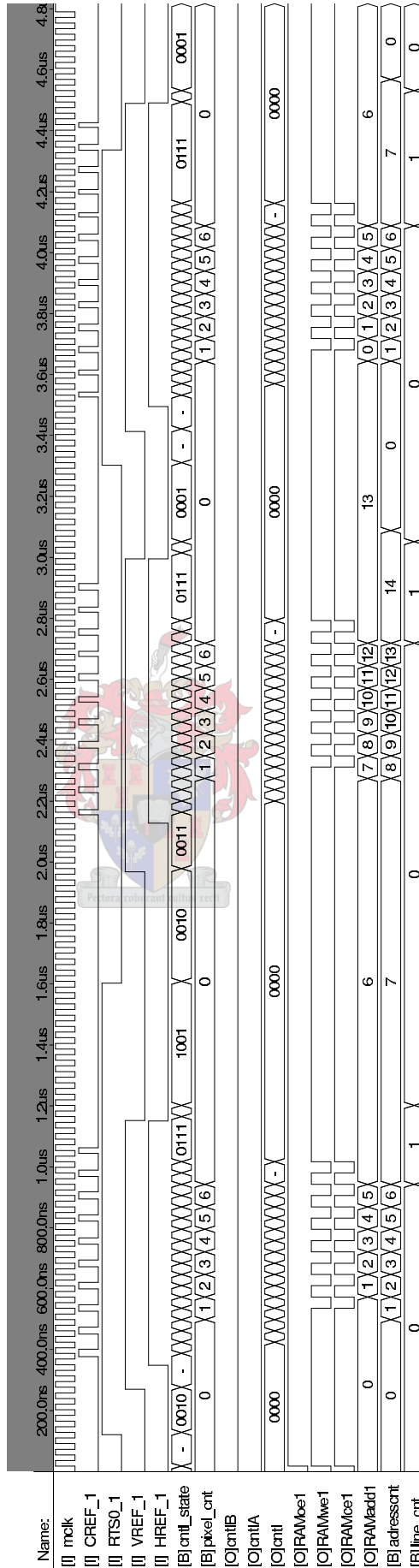


Figure B.5: Control Logic Signals - Camera Data Latch - Multiple Pixels of different Frames

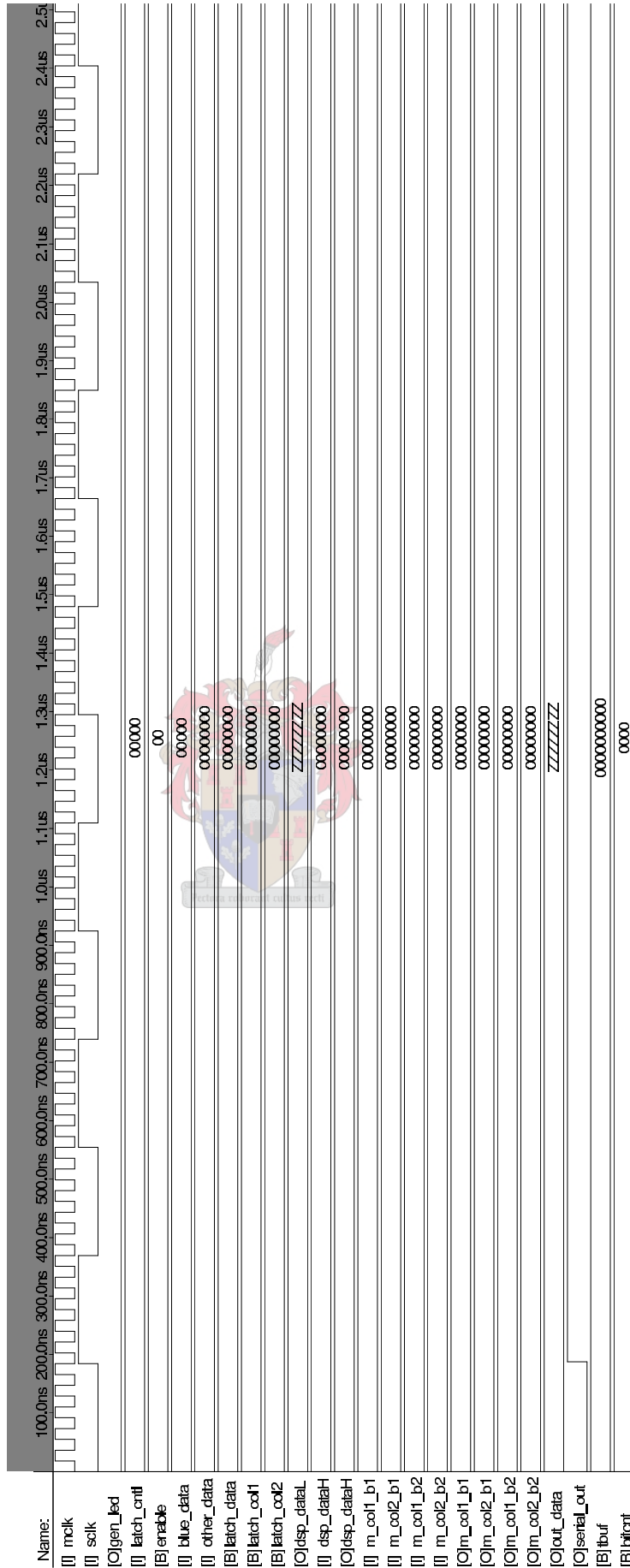


Figure B.6: Data Router - All Outputs in HI Z

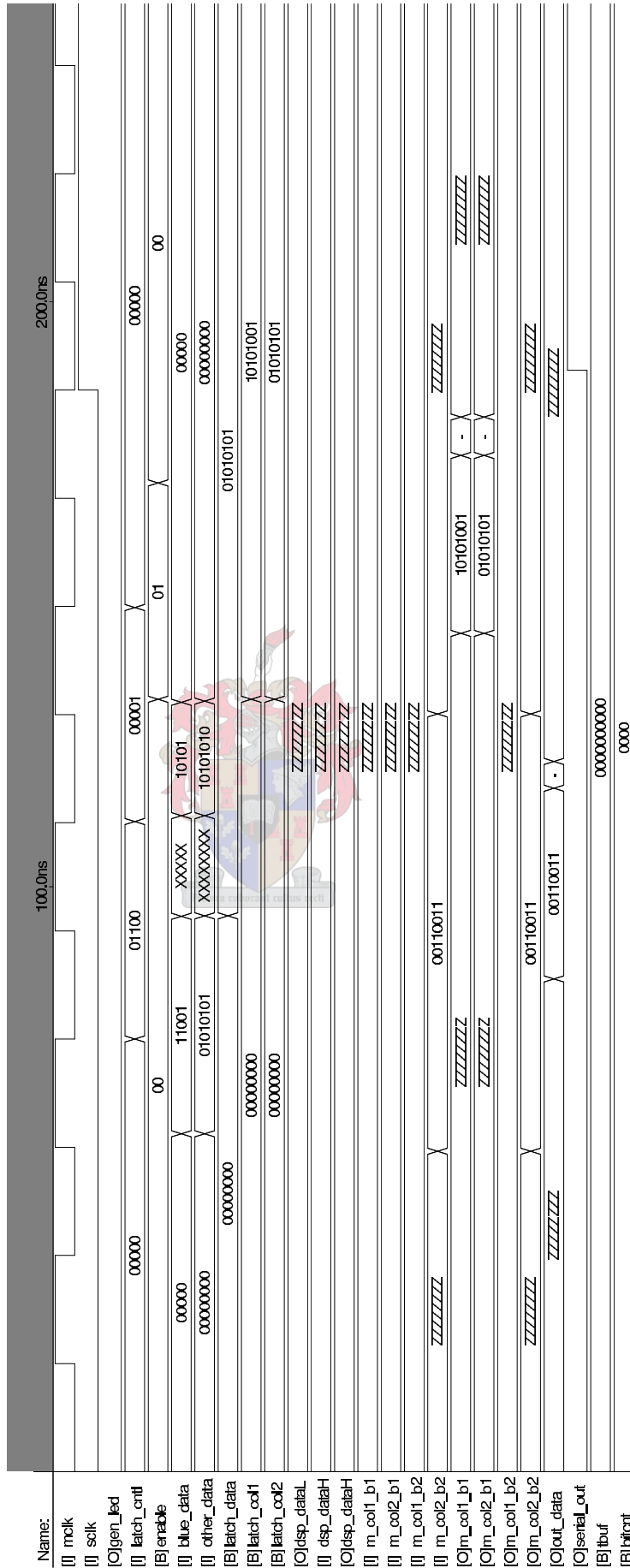


Figure B.7: Data Router - Latch Input Data from Camera to RAM 1



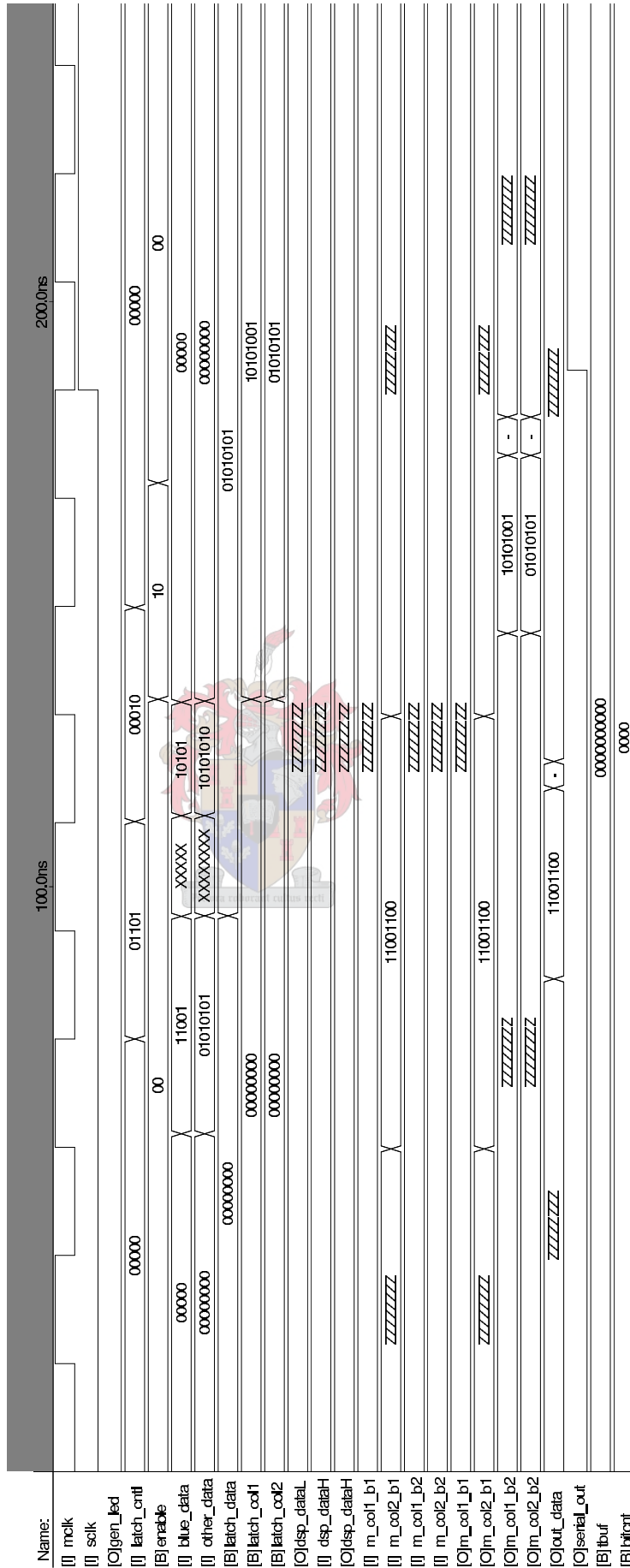


Figure B.8: Data Router - Latch Input Data from Camera to RAM 2

Name:		100_0ms	
[I] mclk			
[I] sclk			
[O]gen_led			
[I] latch_cmt	00100	00111	00000
[B] enable		00	
[I] blue_data		ZZZZZ	
[I] other_data		ZZZZZ	
[B] latch_data		00000000	
[B] latch_cool1		00000000	
[B] latch_cool2		00000000	
[O]dsp_dataL	ZZZZZZ	00110011	01010101
[I] dsp_dataH		ZZZZZZ	ZZZZZZ
[O]dsp_dataH	ZZZZZZ	ZZZZZZ	ZZZZZZ
[I] m_cool1_b1	01010101	01010101	ZZZZZZ
[I] m_cool2_b1	10101010	10101010	ZZZZZZ
[I] m_cool1_b2	00110011	00110011	ZZZZZZ
[I] m_cool2_b2	11001100	11001100	ZZZZZZ
[O]jm_cool1_b1	01010101	01010101	ZZZZZZ
[O]jm_cool2_b1	10101010	10101010	ZZZZZZ
[O]jm_cool2_b2	00110011	00110011	ZZZZZZ
[O]out_data	ZZZZZZ	11001100	ZZZZZZ
[O]out_data	ZZZZZZ	ZZZZZZ	ZZZZZZ
[O]seital_out		00000000	
[B] tbuf		0000	
[B] bitcnt			

Figure B.9: Data Router - RAM to DSP

Name:		100,0ns	
[I] mclk			
[I] sclk			
[O]gen_led			
[I] latch_cmt	00110	00000	00101
[B] enable		00	00000
[I] blue_data		ZZZZZ	
[I] other_data		ZZZZZ	
[B] latch_data		00000000	
[B] latch_cool1		00000000	
[B] latch_cool2		00000000	
[O]dsp_dataL		ZZZZZZ	
[I] dsp_dataH	00001111	ZZZZZZ	11110000
[O]dsp_dataH	00001111	ZZZZZZ	11110000
[I] m_cool1_b1	01010101		ZZZZZZ
[I] m_cool2_b1	10101010		ZZZZZZ
[I] m_cool1_b2			00110011
[I] m_cool2_b2			11001100
[O]m_cool1_b1	01010101		ZZZZZZ
[O]m_cool2_b1	10101010		ZZZZZZ
[O]m_cool1_b2		ZZZZZZ	11110000
[O]m_cool2_b2		ZZZZZZ	00110011
[O]out_data	00001111		11001100
[O]serial_out	10101010		11001100
[B]tbuf		0000000000	
[B]bitcnt		0000	

Figure B.10: Data Router - DSP to RAM

Name:	Value
[I] mclk	
[I] sclk	
[O]gen_led	
[I] latch_cmt	00000
[B] enable	00
[I] blue_data	ZZZZZ
[I] other_data	ZZZZZ
[B] latch_data	00000000
[B] latch_cool1	00000000
[B] latch_cool2	00000000
[O]dsp_dataL	ZZZZZZ
[I] dsp_dataH	ZZZZZZ
[O]dsp_dataH	ZZZZZZ
[I] m_cool1_b1	ZZZZZZ
[I] m_cool2_b1	ZZZZZZ
[I] m_cool1_b2	ZZZZZZ
[I] m_cool2_b2	ZZZZZZ
[O]m_cool1_b1	ZZZZZZ
[O]m_cool2_b1	ZZZZZZ
[O]m_cool1_b2	ZZZZZZ
[O]m_cool2_b2	ZZZZZZ
[O]out_data	ZZZZZZ
[O]serial_out	ZZZZZZ
[B]tbuf	00000000
[B]bitcnt	0000

Figure B.11: Data Router - Constant (Test) Value to RAM

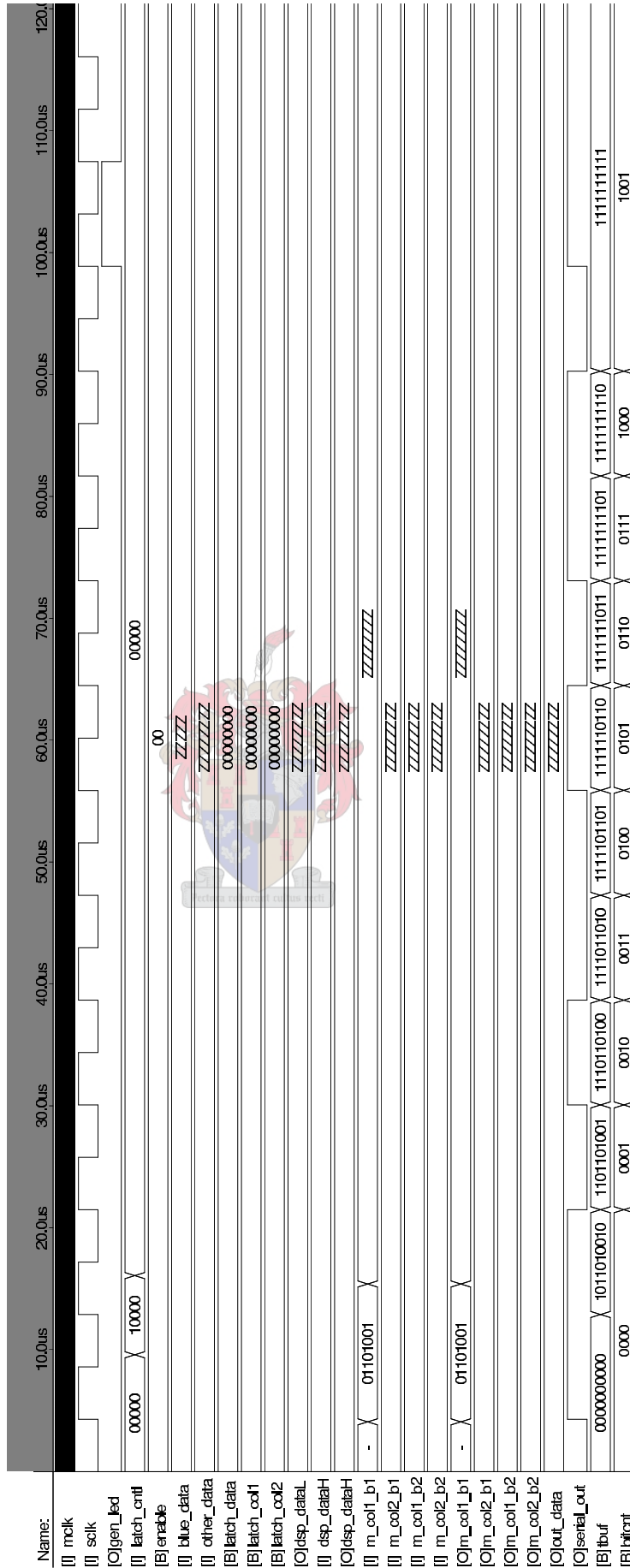


Figure B.12: Data Router - Colour 1, RAM 1 Data Serial Transmission (to PC)

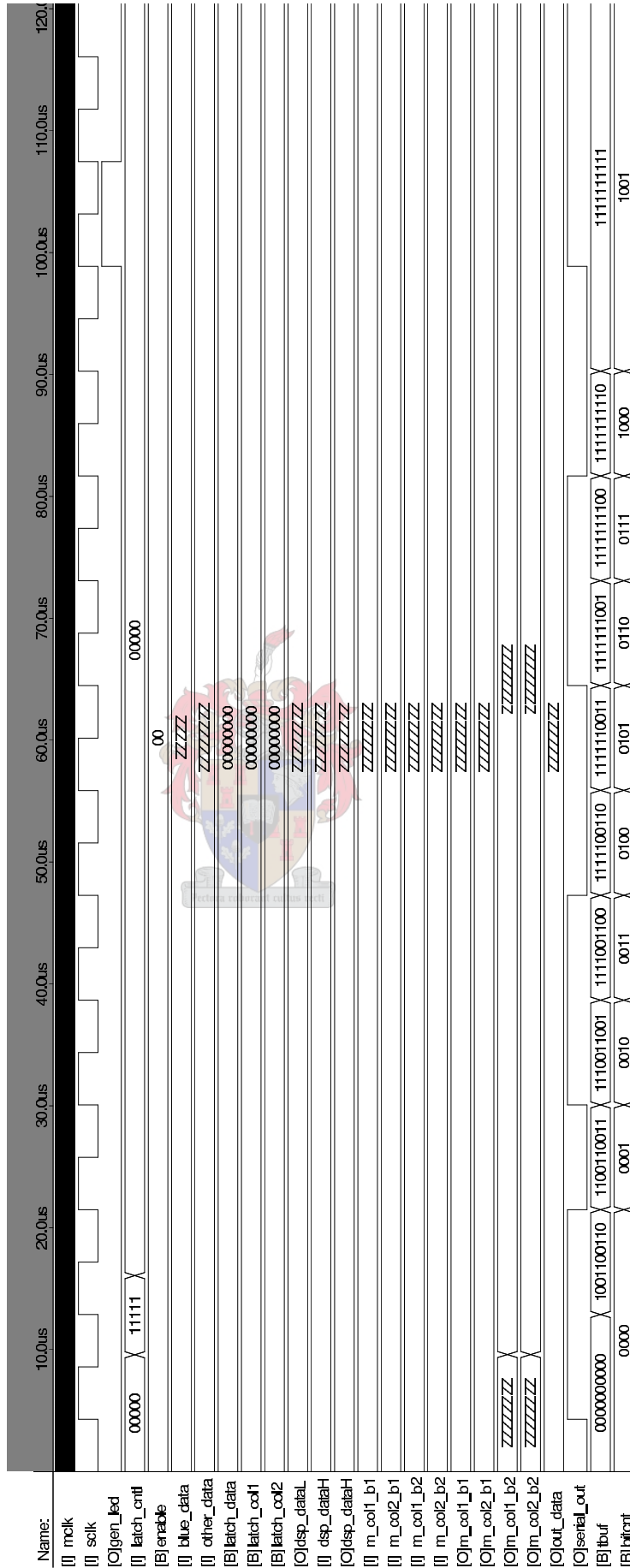


Figure B.13: Data Router - Constant (Test) Data Serial Transmission (to PC)

# Appendix C

## Code Listing

### C.1 FPGA VHDL

```
-- Christiaan Wood
-- Main Program controlling all timing and logic functions

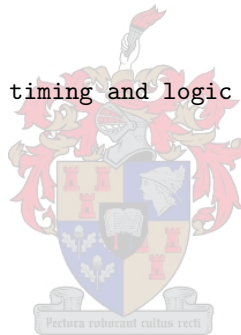
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity u9 is
  port(
    -- clocks
    mclk      : in std_logic;
    ext_clk   : in std_logic;
    sclk      : out std_logic;

    -- serial comms
    TX_sw     : in std_logic;
    TXcntl    : in std_logic_vector(1 downto 0);

    -- external port
    ext_data  : in std_logic_vector(7 downto 0);
    ext_cntl  : in std_logic_vector(3 downto 0);
    ext_sw    : in std_logic;
    ext_leds  : out std_logic_vector(4 downto 0);
    gen_led   : out std_logic;

    -- CPLD control
```



```
cnt1A      : out std_logic;
cnt1B      : out std_logic;
cnt1       : out std_logic_vector(3 downto 0);
outcnt1    : out std_logic_vector(4 downto 0);

-- camera1 A/D, pin names taken directly from datasheets
RTC0_1     : in std_logic;
HREF_1     : in std_logic;
VS_1       : in std_logic;
RTS0_1     : in std_logic;
RTS1_1     : in std_logic;
HS_1       : in std_logic;
CE_1       : out std_logic;
CREF_1     : in std_logic;
VREF_1     : in std_logic;
LLC_1      : in std_logic;
LLC2_1     : in std_logic;
GPSW_1     : in std_logic;

-- camera2 A/D, pin names taken directly from datasheets
RTC0_2     : in std_logic;
HREF_2     : in std_logic;
VS_2       : in std_logic;
RTS0_2     : in std_logic;
RTS1_2     : in std_logic;
HS_2       : in std_logic;
CE_2       : out std_logic;
CREF_2     : in std_logic;
VREF_2     : in std_logic;
LLC_2      : in std_logic;
LLC2_2     : in std_logic;
GPSW_2     : in std_logic;

-- output D/A
OLCC1      : out std_logic;
RTCI       : out std_logic;
RCV_1      : out std_logic;
RCV_2      : out std_logic;

-- I2C comms
SCL        : inout std_logic;
SDA        : inout std_logic;

-- RAM cnt1
RAMadd1    : out std_logic_vector(18 downto 0);
RAMadd2    : out std_logic_vector(18 downto 0);
```



```

RAMwe1      :   out std_logic;
RAMwe2      :   out std_logic;
RAMce1      :   out std_logic;
RAMce2      :   out std_logic;
RAMoe1      :   out std_logic;
RAMoe2      :   out std_logic;

-- DSP signals
dsp_ADD     :   in  std_logic_vector(18 downto 0);
dsp_BE      :   in  std_logic_vector(3  downto 0);
dsp_CEO     :   out std_logic;
dsp_CE1     :   in  std_logic;
dsp_CE2     :   in  std_logic;           -- change
dsp_CE3     :   out std_logic;
dsp_ARE     :   in  std_logic;
dsp_AWE     :   in  std_logic;
dsp_AOE     :   in  std_logic;
dsp_ARDY    :   out std_logic;
dsp_HOLD    :   out std_logic;
dsp_HOLDA   :   in  std_logic;
dsp_BUSREQ  :   out std_logic;
dsp_ECLKOUT :   out std_logic;
dsp_serial  :   in  std_logic);

end entity u9;

architecture logic of u9 is

----- ROM component definition -----
component lpm_rom
  generic (
    lpm_width      : positive;
    lpm_widthhad   : positive;
    lpm_file        : string;

    LPM_ADDRESS_CONTROL: STRING := "UNREGISTERED";
    LPM_OUTDATA: STRING := "UNREGISTERED");

  port (
    address : in std_logic_vector(lpm_widthhad-1 downto 0);
    q       : out std_logic_vector(lpm_width-1 downto 0));

end component;

-----

-- p0, Create Serial Clock at 115200 BAUD
signal clk_buf1      :   std_logic;
constant div         :   integer range 0 to 118 := 117;

```

```

-- p1, Flash LED's
signal cnt      : integer range 0 to 30000000;
signal led      : std_logic;

-- p2, Main state machine
type cntl_state_type is (cntl_init, cntl_wait_4_RTS0_up, cntl_wait_4_VREF_up, ...
... cntl_wait_4_HREF_up, cntl_wait_4_firstpixel, cntl_line_capture1, ...
... cntl_line_capture2, cntl_wait_4_HREF_down, cntl_wait_4_VREF_down, ...
... cntl_wait_4_RTS0_down, dump_address_read, dump_addresshigh_read, ...
... test_address_write, test_data_in, cntl_const2RAM_1, cntl_const2RAM_2, ...
... TXserial_test1, TXserial_test2, RAM4DSP, ill1, ill2, ill3, ill4, ill5, ...
... ill6, ill7, ill8, ill9, ill10,ill11, ill12);
signal cntl_state : cntl_state_type;
signal cntl_var   : std_logic_vector(3 downto 0);
signal adresscnt : integer range 0 to 524287;
signal deb_sclk  : std_logic;
signal line_cnt  : integer range 0 to 700;
signal pixel_cnt : integer range 0 to 720;
signal pingpong  : std_logic;           -- 0 when write to RAM1, 1 to RAM2
signal lRTS0     : std_logic;
signal lVREF     : std_logic;
signal lHREF     : std_logic;

-- vir 640x480, to read first 240 lines
constant max_line : integer range 0 to 700 := 240;

-- vir 640x480, +20 to ignore first 20
constant max_pixel : integer range 0 to 720 := 639+20;
signal dataready   : std_logic;

-- target id variables
signal block_height : integer range 0 to 100 := 30;
signal block_width  : integer range 0 to 100 := 80;
signal target_loc_vert : integer range 0 to 240 := 120;
signal target_loc_hor  : integer range 0 to 720 := 440;
signal DSPtarget_loc_vert : integer range 0 to 240 := 120;
signal DSPtarget_loc_hor  : integer range 0 to 720 := 440;

-- test TX
signal TXbitcnt : integer range 0 to 32 := 0;
signal tbuf     : std_logic_vector(9 downto 0);
signal modeflag : std_logic_vector(1 downto 0);
signal RAMadd1f : std_logic_vector(18 downto 0);
signal RAMadd2f : std_logic_vector(18 downto 0);
signal RAMwelf  : std_logic;
signal RAMwe2f  : std_logic;

```

```

signal RAMce1f      :   std_logic;
signal RAMce2f      :   std_logic;
signal RAMoe1f      :   std_logic;
signal RAMoe2f      :   std_logic;

-- p3, Generate I2C clock
signal clk_buf2     :   std_logic;
constant div2       :   integer range 0 to 270 := 135;

-- p4, I2C core - setup SAA7111 & SAA7127
type statetype is (nop, init, idle, start_a, start_b, start_c, start_d, ...
... stop_a, stop_b, stop_c, write_a, write_b, write_c, write_d, ...
... wait_for_ack_a, wait_for_ack_b, wait_for_ack_c, wait_for_ack_d, ...
... wait_for_ack_e, transmit);
signal state        :   statetype;
signal i2c_tbuf     :   std_logic_vector(8 downto 0);
signal i2c_bitcnt   :   integer range 0 to 9;

signal SDA_out_flag, SCL_out_flag :   std_logic;
signal SDA_in_flag  :   std_logic;
signal SDA_low_flag :   std_logic;
signal onoff        :   std_logic;
signal start_cnt    :   integer range 0 to 3;
signal byte         :   integer range 0 to 10;
signal deb3         :   std_logic;
signal i2c_finish   :   std_logic;

constant TX_data_0 :   std_logic_vector(7 downto 0) := "00010001";
constant TX_data_1 :   std_logic_vector(7 downto 0) := "00011100";
constant slave_address_write :   std_logic_vector(7 downto 0) := "01001000";

-- RAM signals (lpm_ROM)
signal address      :   std_logic_vector(8 downto 0);
signal q            :   std_logic_vector(7 downto 0);
signal RAM_add_cnt :   integer range 0 to 300;

----- BEGIN -----

begin

-- RAM definitions, 8bit address, 8bit data, filename. g0: lpm_rom
generic map (lpm_widthad => 9,
            lpm_width => 8,
            lpm_file => "my_i2c.mif")

```

```

port map (address => address, q => q);

-- Create Serial Clock at 115200 BAUD
p0: process (mclk) is
variable cnt1 : integer range 0 to 511;
begin
  if rising_edge(mclk) then
    if cnt1 = div then
      clk_buf1 <= not clk_buf1;
      cnt1 := 0;
    else
      cnt1 := cnt1 + 1;
    end if;

  end if;
end process p0;

-- Flash LED's - for debugging
p1: process (mclk) is
begin
  if rising_edge(mclk) then
    if cnt = 10000000 then
      cnt <= 0;
      led <= not(led);
    else
      cnt <= cnt+1;
    end if;
  end if;
end process p1;

-- Process for all signals to CPLDs
-- Do Memory Dump
p2: process (mclk, clk_buf1, CREF_1, VREF_1, HREF_1, RTS0_1,
LLC2_1, ext_sw) is

variable ad_high_cnt : integer range 0 to 3000; -- Address valid time
variable data_high_cnt : integer range 0 to 3000; -- Data valid time
variable deb1, deb2, deb3 : std_logic; -- Variables to debounce switches
variable col_flipflop : std_logic;
-- variable modeflag : std_logic_vector(1 downto 0);
variable modecounter : integer range 0 to 3;

begin

  if (ext_sw = '0') then -- external reset

```

```

    ext_leds(0) <= '0';          -- reset ERROR led
    cntl_state <= cntl_init;
elseif rising_edge(mclk) then

    case cntl_state is

    -- initialise
    when cntl_init =>
        -- All RAM signals in high Z.
        -- disable outputs to prevent both CPLD and RAM to try and output data
            -- (note 17, RAM datasheet)
--
        ext_leds(0) <= '0';
        ext_leds(1) <= '0';
        ext_leds(2) <= '0';
        ext_leds(3) <= '0';
        ext_leds(4) <= '0';

        -- set all RAM controls to '1'
        RAMoe1f <= '1';
        RAMce1f <= '1';
        RAMwe1f <= '1';
        RAMoe2f <= '1';
        RAMwe2f <= '1';
        RAMce2f <= '1';

        -- no control signals to CPLD data routers
        cntlA <= '0';
        cntlB <= '0';
        cntl <= "0000";

        -- various variables initialised
--
        RCV_1 <= '0'; --          RCV_2 <= '0';
        pingpong <= '1';
        line_cnt <= 0;
        pixel_cnt <= 0;
        lRTSO <= RTS0_1;
        deb_sclk <= '0';
        dataready <= '0';
        modeflag <= "00";
        modecounter := 0;

---
        gen_led <= '1';

        -- wait for I2C setup to complete before reading data from camera
        if i2c_finish = '1' then
            cntl_state <= cntl_wait_4_RTS0_up;

```

```
end if;
```

-----

```
-- wait for RTS0 up (this indicates the first half (interlaced)
  -- of the first frame of a line)
-- also wait for possible mem dump command
when cntl_wait_4_RTS0_up =>
  deb_sclk <= clk_buf1;
  adresscnt <= 0;
  lRTS0 <= RTS0_1;

  -- THIS IS WHERE WE DECIDE EXACTLY WHATS HAPPENS IN THE CYCLE
  -- poll modeflag
    -- modeflag, (00=normal), (01=dsp wants to read),
    -- (10=dsp wants to write)
  -- DSP sets flag, says it wants to read
  -- must write b4 every ping pong?
  -- pingpong <= some dsp signal?? then dsp determines which bank??
if (RTS0_1 = '1' and lRTS0 = '0') then      -- RTS0 rising edge
---   pingpong <= dsp_HOLD_A;
---   modeflag(0) <= dsp_CE1;
---   modeflag(1) <= dsp_serial;
  cntl_state <= cntl_wait_4_VREF_up;
end if;

-- Dump contents of active RAM to PC
if TX_sw = '0' then
  deb1 := '1';                                -- debounce TX_sw
end if;
if (TX_sw = '1') and (deb1 = '1') then
  deb1 := '0';
  -- What to do when we press the button??
  -- to test serial TX
    -- cntl_state <= TXserial_test1;
  -- to dump memory content to PC
    cntl_state <= dump_address_read;
  -- to freeze capture, to allow DSP to read memory as RAM
    -- cntl_state <= RAM4DSP;
end if;
```

-----

```
-- wait for VREF up (this indicates the start of a frame (one VREF for each half))
when cntl_wait_4_VREF_up =>
  -- assign non-normal r/w enable's and addresses
```

```

LVREF <= VREF_1;
if VREF_1 = '1' and LVREF = '0' then

    -- Read flags from DSP
    if RTS0_1 = '1' then          -- only read once first half of interlaced frame
        pingpong <= dsp_HOLD_A;
        modeflag(0) <= dsp_CE1;
        modeflag(1) <= dsp_serial;
    end if;

    cntl_state <= cntl_wait_4_HREF_up;
end if;
-----

-- wait for HREF up (this indicates the start of a line)
when cntl_wait_4_HREF_up =>
    -- change - wait for HREF = CREF = 1.
    -- then u know first pixel has started. ignore first, then capture. ;)

    if (HREF_1 = '1') and (CREF_1 = '1') then
        cntl_state <= cntl_wait_4_firstpixel;
    end if;
-----

-- Ignores first available pixel, sets up datarouters and latches for second pixel
when cntl_wait_4_firstpixel =>

    -- must do here, otherwise cntl sigs not hi
    if RTS0_1 = '1' then
        dsp_HOLD <= '1';          -- signal to DSP that RAM's ready
        dataready <= '1';
    end if;

    -- Write to RAM1, Read from RAM2
    if pingpong = '0' then        -- write to RAM1
        if modeflag = "00" then
            cntl <= "1100";        -- LATCH
        elsif modeflag = "01" then
            cntl <= "0111";        -- RAM1 to DSP
        --
        RAMadd1f <= dsp_ADD;
        elsif modeflag = "10" then
            cntl <= "0101";        -- DSP to RAM1
        --
        RAMwe1f <= dsp_AWE;
        --
        RAMadd1f <= dsp_ADD;
    end if;

```

```

-- Write to RAM2, Read from RAM1
else
    if modeflag = "00" then
        cnt1 <= "1101";
    elsif modeflag = "01" then
        cnt1 <= "0100";
--
        RAMadd2f <= dsp_ADD;
    elsif modeflag = "10" then
        cnt1 <= "0110";
--
        RAMwe2f <= dsp_AWE;
--
        RAMadd2f <= dsp_ADD;
    end if;
end if;

cnt1_state <= cnt1_line_capture1;

-----

-- capture line, part 1
when cnt1_line_capture1 =>
--
    dsp_HOLD <= '1'; -- signal to DSP that RAM's ready
    cnt1A <= '0';
    cnt1B <= '0';

-- Write to RAM1, Read from RAM2
if pingpong = '0' then
    if modeflag = "00" then
        cnt1 <= "0001";
    elsif modeflag = "01" then
        cnt1 <= "0111";
--
        RAMadd1f <= dsp_ADD;
    elsif modeflag = "10" then
        cnt1 <= "0101";
--
        RAMwe1f <= dsp_AWE;
--
        RAMadd1f <= dsp_ADD;
    end if;

else
-- write to RAM2
-- Write to RAM2, Read from RAM1
    if modeflag = "00" then
        cnt1 <= "0010";
    elsif modeflag = "01" then
        cnt1 <= "0100";
--
        RAMadd2f <= dsp_ADD;
    elsif modeflag = "10" then
        cnt1 <= "0110";
--
        RAMwe2f <= dsp_AWE;

```



```

--          RAMadd2f <= dsp_ADD;
          end if;
        end if;

        if pixel_cnt > 20 then          -- ignore first 20 pixels (bad camera optics)
          outcnt1 <= "00001";

-- Write to RAM1, Read from RAM2
        if pingpong = '0' then
          if modeflag = "00" then      -- normal operation
            RAMce1f <= '0';
            RAMwe1f <= '0';

            RAMwe2f <= '1';
            RAMce2f <= '0';
            RAMoe2f <= '0';

          elsif modeflag = "01" then   -- read from RAM2 as normal, RAM1 to DSP
            RAMwe2f <= '1';
            RAMce2f <= '0';
            RAMoe2f <= '0';

          elsif modeflag = "10" then   -- read from RAM2 as normal, DSP to RAM1
            RAMwe2f <= '1';
            RAMce2f <= '0';
            RAMoe2f <= '0';
          end if;

        else
-- Write to RAM2, Read from RAM1
          if modeflag = "00" then      -- normal operation
            RAMce2f <= '0';
            RAMwe2f <= '0';

            RAMwe1f <= '1';
            RAMce1f <= '0';
            RAMoe1f <= '0';

          elsif modeflag = "01" then   -- read from RAM1 as normal, RAM2 to DSP
            RAMwe1f <= '1';
            RAMce1f <= '0';
            RAMoe1f <= '0';

          elsif modeflag = "10" then   -- read from RAM1 as normal, DSP to RAM2
            RAMwe1f <= '1';
            RAMce1f <= '0';

```

```

        RAMoe1f <= '0';
    end if;
end if;
end if;
cntl_state <= cntl_line_capture2;

```

```

-----
-- capture line, part2
when cntl_line_capture2 =>
    -- Write to RAM1, Read from RAM2
    if pingpong = '0' then
        if modeflag = "00" then          -- normal operation
            RAMce1f <= '1';
            RAMwe1f <= '1';

            RAMwe2f <= '1';
            RAMce2f <= '1';
            RAMoe2f <= '1';

        elsif modeflag = "01" then      -- read from RAM2 as normal, RAM1 to DSP
            RAMwe2f <= '1';
            RAMce2f <= '1';
            RAMoe2f <= '1';

        elsif modeflag = "10" then      -- read from RAM2 as normal, DSP to RAM1
            RAMwe2f <= '1';
            RAMce2f <= '1';
            RAMoe2f <= '1';
        end if;

    else
        -- Write to RAM2, Read from RAM1
        if modeflag = "00" then          -- normal operation
            RAMce2f <= '1';
            RAMwe2f <= '1';

            RAMwe1f <= '1';
            RAMce1f <= '1';
            RAMoe1f <= '1';

        elsif modeflag = "01" then      -- read from RAM1 as normal, RAM2 to DSP
            RAMwe1f <= '1';
            RAMce1f <= '1';
            RAMoe1f <= '1';

        elsif modeflag = "10" then      -- read from RAM1 as normal, DSP to RAM2

```

```

        RAMwe1f <= '1';
        RAMce1f <= '1';
        RAMoe1f <= '1';
    end if;
end if;

-- ignore first 20 pixels (this 'if' (>=) will run b4 top one (>))
if pixel_cnt >= 20 then
    adresscnt <= adresscnt + 1;
    outcntl <= "001" & ext_cntl(1) & not(ext_cntl(1));
end if;

-- Give DSP flag when ready
-- dsp_HOLD_A <= '1';

if pingpong = '0' then
    if modeflag = "00" then
        RAMadd1f <= conv_std_logic_vector(adresscnt,19);
        RAMadd2f <= conv_std_logic_vector(adresscnt,19);

    elsif modeflag = "01" then
--        RAMadd1f <= dsp_ADD;
        RAMadd2f <= conv_std_logic_vector(adresscnt,19);

    elsif modeflag = "10" then
--        RAMadd1f <= dsp_ADD;
        RAMadd2f <= conv_std_logic_vector(adresscnt,19);
    end if;

else
    if modeflag = "00" then
        RAMadd1f <= conv_std_logic_vector(adresscnt,19);
        RAMadd2f <= conv_std_logic_vector(adresscnt,19);

    elsif modeflag = "01" then
--        RAMadd1f <= conv_std_logic_vector(adresscnt,19);
        RAMadd2f <= dsp_ADD;

    elsif modeflag = "10" then
--        RAMadd1f <= conv_std_logic_vector(adresscnt,19);
        RAMadd2f <= dsp_ADD;
    end if;
end if;

```

```

if pixel_cnt = max_pixel then          -- last pixel to capture
ext_leds(3) <= '1';
    pixel_cnt <= 0;                    -- reset pixel counter
    line_cnt <= line_cnt + 1;          -- increment line counter

-- reset control signals
-- dsp_HOLDA <= '0';
cntlA <= '0';
cntlB <= '0';
-- cntl <= "0000";
outcntl <= "00000";
lhREF <= href_1;
cntl_state <= cntl_const2RAM_1;
else
    pixel_cnt <= pixel_cnt + 1;        -- increment pixel counter

-- Write to RAM1, Read from RAM2
if pingpong = '0' then                -- write to RAM1
    if modeflag = "00" then
        cntl <= "1100";                -- LATCH
    elsif modeflag = "01" then
        cntl <= "0111";                -- RAM1 to DSP
    elsif modeflag = "10" then
        cntl <= "0101";                -- DSP to RAM1
    end if;

-- Write to RAM2, Read from RAM1
else                                    -- write to RAM2
    if modeflag = "00" then
        cntl <= "1101";                -- LATCH
    elsif modeflag = "01" then
        cntl <= "0100";                -- RAM2 to DSP
    elsif modeflag = "10" then
        cntl <= "0110";                -- DSP to RAM2
    end if;
end if;

    cntl_state <= cntl_line_capture1;
end if;

-----

-- write a constant, 55H, to RAM after each line. for debugging
when cntl_const2RAM_1 =>

    cntlA <= '0';
    cntlB <= '0';

```

```

-- Write to RAM1, Read from RAM2
if modeflag = "00" then
  if pingpong = '0' then
    RAMce1f <= '0';
    RAMwe1f <= '0';
  else
    -- Write to RAM2, Read from RAM1
    RAMce2f <= '0';
    RAMwe2f <= '0';
  end if;
end if;

cntl_state <= cntl_const2RAM_2;

-----

when cntl_const2RAM_2 =>

  if modeflag = "00" then
    cntl <= "0000";
  end if;
-- RCV_2 <= '0'; -- manually set HREF
--
-- finished read/write for RAM's
if modeflag = "00" then
  if pingpong = '0' then
    RAMce1f <= '1';
    RAMwe1f <= '1';
  else
    -- Write to RAM2, Read from RAM1
    RAMce2f <= '1';
    RAMwe2f <= '1';
  end if;
end if;

cntl_state <= cntl_wait_4_HREF_down ;

-----

-- wait for HREF down (this indicates the end of a line)
-- can probably skip this (we count pixels, then know when line is finished),
-- but this way, we know we only check for VREF down after HREF is down
-- (thus we are sure we measure right)
when cntl_wait_4_HREF_down =>
  -- options: - finished with line (not frame) - jump back to wait for HREF up
  --           - finished with first half of frame - continue, after RTS0 down,
  --           - jump to wait for VREF up
  --           - finished with second half of frame - continue to wait for RTS0 up.

```

```

LHREF <= HREF_1;
if HREF_1 = '0' and LHREF = '1' then          -- HREF falling edge
    cntl_state <= cntl_wait_4_VREF_down;
end if;

-----

when cntl_wait_4_VREF_down =>
    -- stil busy with frame, ie read next line
    -- read the maximum decided line (per frame), wait for new frame
    if (VREF_1 = '1') and (line_cnt /= max_line) then
        cntl_state <= cntl_wait_4_HREF_up;

    -- finished first half - go to wait for wait for RTS0 down
    elsif RTS0_1 = '1' then
        line_cnt <= 0;
        lRTSO <= RTS0_1;
--      RCV_1 <= '0';          -- manually set VREF
        cntl_state <= cntl_wait_4_RTS0_down;

    else
    -- finished 2cnd half - go to wait for wait for RTS0 up.
        line_cnt <= 0;
--      RCV_1 <= '0';          -- manually set VREF
--      cntl_state <= RAM4DSP;    -- MAAK MY BORD RAM VIR DSP

        dsp_HOLD <= '0';          -- can wait longer?? signal to DSP
--      that RAM's no longer available

        dataready <= '0';

        cntl <= "0000";
        modeflag <= "11";
        RAMoe1f <= '1';
        RAMce1f <= '1';
        RAMwe1f <= '1';
        RAMoe2f <= '1';
        RAMwe2f <= '1';
        RAMce2f <= '1';

        cntl_state <= cntl_wait_4_RTS0_up;  -- wat dit was
    end if;

-----

when cntl_wait_4_RTS0_down =>

    lRTSO <= RTS0_1;
    if RTS0_1 = '0' and lRTSO = '1' then          -- RTS0 falling edge

```

```

        cntl_state <= cntl_wait_4_VREF_up;
    end if;

-----

-- Dump contents of active RAM to PC
when dump_address_read =>
    -- Sync process with serial clock
    deb_sclk <= clk_buf1;
    modeflag <= "11";

    if deb_sclk = '0' and clk_buf1 = '1' then
        -- assert addresses
        RAMadd1f <= conv_std_logic_vector(adresscnt,19);
        RAMadd2f <= conv_std_logic_vector(adresscnt,19);
        adresscnt <= adresscnt + 1;

        -- enable RAM's for read
        RAMwe1f <= '1';
        RAMce1f <= '0';
        RAMoe1f <= '0';
        RAMwe2f <= '1';
        RAMce2f <= '0';
        RAMoe2f <= '0';

        ---
        gen_led <= '0';
        -- gen_led on while mem dump

        --
        if adresscnt = 524287 then
        -- full RAM content
        --
        if adresscnt = 200 then
        -- only dump small portion of RAM
        --
        if adresscnt = 307200 then
        -- 640x480 pic size
            adresscnt <= 0;
            RAMwe1f <= '1';
            RAMce1f <= '1';
            RAMoe1f <= '1';
            RAMwe2f <= '1';
            RAMce2f <= '1';
            RAMoe2f <= '1';
            cntl_state <= cntl_wait_4_RTS0_up;
        else
            cntl_state <= dump_addresshigh_read;
        end if;
    end if;
end if;
-- sclk

-----

-- loops untill byte is written
when dump_addresshigh_read =>
    -- Sync process with serial clock

```

```

deb_sclk <= clk_buf1;
if deb_sclk = '0' and clk_buf1 = '1' then

    if ad_high_cnt < 1 then          -- assert address for 1 serial clk
        cnt1B <= '1';
        cnt1A <= '1';
        cnt1 <= "00" & TXcnt1;
    else
        cnt1B <= '0';
        cnt1A <= '0';
        cnt1 <= "0000";
    end if;

    -- just waits the amount of time it takes to transmit byte
    -- wait 10 serial clocks to transmit all data
    if ad_high_cnt = 10 then
        ad_high_cnt := 0;
        cnt1_state <= dump_address_read;
    else
        ad_high_cnt := ad_high_cnt + 1;
    end if;
end if;

```

---

```

-- Test write to RAM
when test_address_write =>
    modeflag <= "11";

    -- Sync process with serial clock
    deb_sclk <= clk_buf1;
    if deb_sclk = '0' and clk_buf1 = '1' then
        -- read external data pins for address
        RAMadd1f <= "00000000000" & ext_data;
        RAMwe1f <= '0';
        RAMce1f <= '0';
        RAMoe1f <= '0';
        RAMadd2f <= "00000000000" & ext_data;
        RAMwe2f <= '0';
        RAMce2f <= '0';
        RAMoe2f <= '0';

        cnt1A <= '0';
        cnt1B <= '0';
        cnt1 <= "10" & ext_cnt1(1 downto 0);    -- which bank to write to
        cnt1_state <= test_data_in;
    end if;

```



```

-----
-- read test data in, and write to RAM at location specified
when test_data_in =>
    modeflag <= "11";

    deb_sclk <= clk_buf1;
    if deb_sclk = '0' and clk_buf1 = '1' then

        -- assert data for two clocks, then disable RAM's
        if data_high_cnt = 2 then
            RAMwe1f <= '1';
            RAMce1f <= '1';
            RAMoe1f <= '1';
            RAMwe2f <= '1';
            RAMce2f <= '1';
            RAMoe2f <= '1';
        end if;

        -- after 2 more clocks, leave process
        if data_high_cnt = 4 then
            data_high_cnt := 0;
            cntl_state <= cntl_wait_4_RTS0_up; -- <= the correct one
            -- do mem dump immediately, to confirm data was written
            -- (b4, camdata overwrites data)
--            cntl_state <= dump_address_read;

        else
            data_high_cnt := data_high_cnt + 1;
        end if;
    end if;
-----

when TXserial_test1 =>
    -- Sync process with serial clock
--    deb_sclk <= clk_buf1;
--    if deb_sclk = '0' and clk_buf1 = '1' then
--
--        dsp_CE1 <= '1';
--        tbuf(0) <= '0';
--        tbuf(8 downto 1) <= '0' & ext_data(7 downto 1);
--        tbuf(9) <= '1';
--        TXbitcnt <= 0;
--        cntl_state <= TXserial_test2;
--    end if;

when TXserial_test2 =>

```

```

-- Sync process with serial clock
-- deb_sclk <= clk_buf1;
-- if deb_sclk = '0' and clk_buf1 = '1' then
--     dsp_CE1 <= tbuf(0);
--     tbuf(8 downto 0) <= tbuf(9 downto 1);
--     tbuf(9) <= '1';
--     if TXbitcnt = 9 then
--         cntl_state <= cntl_wait_4_RTS0_up;
--     else
--         TXbitcnt <= TXbitcnt + 1;
--     end if;
-- end if;

```

---

```

when RAM4DSP =>

```

```

    modeflag <= "11";

```

```

-- freeze capturing

```

```

-- enable RAM's for read

```

```

-- dsp_HOLD_A <= '1';

```

```

-- DSP signals

```

```

-- dsp_ARE <= 'Z';

```

```

-- dsp_AWE <= 'Z';

```

```

-- dsp_AOE <= 'Z';

```

```

RAMwe1f <= '1';

```

```

RAMce1f <= '0';

```

```

RAMoe1f <= '0';

```

```

RAMwe2f <= '1';

```

```

RAMce2f <= '0';

```

```

RAMoe2f <= '0';

```

```

RAMadd1f <= dsp_ADD;

```

```

RAMadd2f <= dsp_ADD;

```

```

cntl <= "0011";

```

```

cntl_state <= cntl_wait_4_RTS0_up;

```

---

```

-- illegal states causes jumo to init,

```

```

-- to cover for every possible state value

```

```

when ill1 =>

```

```

    ext_leds(2) <= '1';

```

```

    cntl_state <= cntl_init;

```

```

when ill2 =>

```

```

        ext_leds(2) <= '1';
        cntl_state <= cntl_init;
when ill3 =>
        ext_leds(2) <= '1';
        cntl_state <= cntl_init;
when ill4 =>
        ext_leds(2) <= '1';
        cntl_state <= cntl_init;
when ill5 =>
        ext_leds(2) <= '1';
        cntl_state <= cntl_init;
when ill6 =>
        ext_leds(2) <= '1';
        cntl_state <= cntl_init;
when ill7 =>
        ext_leds(2) <= '1';
        cntl_state <= cntl_init;
when ill8 =>
        ext_leds(2) <= '1';
        cntl_state <= cntl_init;
when ill9 =>
        ext_leds(2) <= '1';
        cntl_state <= cntl_init;
when ill10 =>
        ext_leds(2) <= '1';
        cntl_state <= cntl_init;
when ill11 =>
        ext_leds(2) <= '1';
        cntl_state <= cntl_init;
when ill12 =>
        ext_leds(2) <= '1';
        cntl_state <= cntl_init;
-----

        -- initialise, also, to ensure no illegal states occur
when others =>
        cntl_state <= cntl_init;
-----

        end case;
    end if;
end process p2;
-----

-- Generate I2C clock p3: process (mclk) is
variable cnt2    :    integer range 0 to 511;
begin
    if rising_edge(mclk) then

```

```

        if cnt2 = div2 then
            clk_buf2 <= not clk_buf2;
            cnt2 := 0;
        else
            cnt2 := cnt2 + 1;
        end if;
    end if;
end process p3;

-- I2C core - setup SAA7111 p4: process (clk_buf2) is
begin
    if rising_edge(clk_buf2) then
        case state is

            when init =>
                -- initialise variables
                ext_leds(0) <= '1';
                i2c_finish <= '0';
                byte <= 0;
                start_cnt <= 0;
                SDA_out_flag <= '1';
                SCL_out_flag <= '1';
                SDA_low_flag <= '1';
                RAM_add_cnt <= 0;

                if ext_data(0) = '1' then -- wait for SAA7111 power-up reset to finish
                    state <= idle;
                    state <= nop;
                end if;

                -- to manually override SAA7111 reset indicator
                when nop =>
                    i2c_finish <= '1';
                    state <= idle;

            when idle =>

                if RAM_add_cnt = 288 then -- lines in RAM
                    ext_leds(0) <= '0';
                    i2c_finish <= '1'; -- set flag to indicate i2c setup complete
                end if;

                if (byte = 0) and (RAM_add_cnt < 288) then -- last entry + 1 = 225
                    address <= conv_std_logic_vector(RAM_add_cnt,9);
                    RAM_add_cnt <= RAM_add_cnt + 1;
                end if;
            end case;
        end if;
    end process;
end;

```

```

        byte <= 1;
        state <= start_a;
    end if;

    if byte = 1 then
        i2c_tbuf(8) <= '0';
        i2c_tbuf(7 downto 0) <= q;
        byte <= 2;
        i2c_bitcnt <= 0;
        address <= conv_std_logic_vector(RAM_add_cnt,9);
        RAM_add_cnt <= RAM_add_cnt + 1;
        state <= transmit;
    end if;

    if byte = 2 then
        i2c_tbuf(8) <= '0';
        i2c_tbuf(7 downto 0) <= q;
        byte <= 3;
        i2c_bitcnt <= 0;
        address <= conv_std_logic_vector(RAM_add_cnt,9);
        RAM_add_cnt <= RAM_add_cnt + 1;
        state <= transmit;
    end if;

    if byte = 3 then
        i2c_tbuf(8) <= '0';
        i2c_tbuf(7 downto 0) <= q;
        byte <= 4;
        i2c_bitcnt <= 0;
        state <= transmit;
    end if;

    if byte = 4 then
        state <= stop_a;
        byte <= 0;
    end if;

-- START condition
when start_a =>
    SDA_out_flag <= '1';
--
    SCL_out_flag <= '1';
    state <= start_b;
when start_b =>
    SDA_out_flag <= '1';
    SCL_out_flag <= '1';

```

```

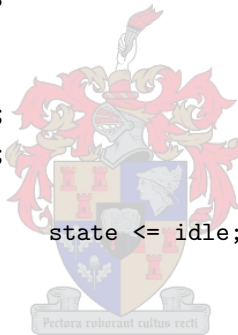
        state <= start_c;
when start_c =>
    SDA_out_flag <= '0';
    SCL_out_flag <= '1';
    state <= start_d;
when start_d =>
    SDA_out_flag <= '0';
    SCL_out_flag <= '0';
    state <= idle;
--      state <= transmit;

-- STOP condition
when stop_a =>
    SDA_out_flag <= '0';
    SCL_out_flag <= '0';
    state <= stop_b;
when stop_b =>
    SDA_out_flag <= '0';
    SCL_out_flag <= '1';
    state <= stop_c;
when stop_c =>
    SDA_out_flag <= '1';
    SCL_out_flag <= '1';
    state <= idle;
--      when stop_d => --      state <= idle;

-- WRITE condition
when write_a =>
    SDA_out_flag <= i2c_tbuf(8);
--      SDA_out <= i2c_tbuf(8);
    state <= write_b;
when write_b =>
    SCL_out_flag <= '1';
    state <= write_c;
when write_c =>
    SCL_out_flag <= '0';
    -- keep SCL high
    state <= write_d;
when write_d =>
--      SCL_out_flag <= '0';
    SDA_out_flag <= '1';
    state <= transmit;

-- wait for ACKNOWLEDGE
when wait_for_ack_a =>
    SDA_out_flag <= '1';

```



```

        state <= wait_for_ack_b;
    when wait_for_ack_b =>
        SCL_out_flag <= '1';
        state <= wait_for_ack_c;
    when wait_for_ack_c =>
--      if TX_sw = '1' then
--      if SDA_in_flag = '1' then
        SDA_low_flag <= SDA;
--      SDA_in_flag <= SDA;
        SCL_out_flag <= '0';
        state <= wait_for_ack_d;
    when wait_for_ack_d =>
--      SDA_out_flag <= '0';
        state <= wait_for_ack_e;
    when wait_for_ack_e =>
--      if SDA_in_flag = '0' then
        SDA_in_flag <= '1';
--      SDA_out_flag <= '0';
        SDA_low_flag <= '1';
        state <= idle;
--      end if;

-- setup transmit buffer
when transmit =>
    i2c_tbuf(8 downto 1) <= i2c_tbuf(7 downto 0);
    i2c_tbuf(0) <= '1';
    if i2c_bitcnt = 8 then
        state <= wait_for_ack_a;
    else
        i2c_bitcnt <= i2c_bitcnt + 1;
        state <= write_a;
    end if;
end case;
end if;
end process p4;

```

---

```

-- Various Assignments
CE_1 <= '1';          -- chip enable for SAA7111 (U4)
CE_2 <= '1';          -- chip enable for SAA7111 (U5)

with (pingpong & modeflag) select
    RAMadd1 <= RAMadd1f when "000" | "100" | "101" | "110",
              dsp_ADD when "001" | "010",
              RAMadd1f when others;

```

```

with (pingpong & modeflag) select
    RAMwe1 <=  RAMwe1f when "000" | "100" | "101" | "110",
               '1' when "001",
               dsp_AWE when "010",
               RAMwe1f when others;

with (pingpong & modeflag) select
    RAMce1 <=  RAMce1f when "000" | "100" | "101" | "110",
               '0' when "001" | "010",
               RAMce1f when others;

with (pingpong & modeflag) select
    RAMoe1 <=  RAMoe1f when "000" | "100" | "101" | "110",
               '0' when "001",
               '1' when "010",
               RAMoe1f when others;

with (pingpong & modeflag) select
    RAMadd2 <= RAMadd2f when "100" | "000" | "001" | "010",
               dsp_ADD when "101" | "110",
               RAMadd2f when others;

with (pingpong & modeflag) select
    RAMwe2 <=  RAMwe2f when "100" | "000" | "001" | "010",
               '1' when "101",
               dsp_AWE when "110",
               RAMwe2f when others;

with (pingpong & modeflag) select
    RAMce2 <=  RAMce2f when "100" | "000" | "001" | "010",
               '0' when "101" | "110",
               RAMce2f when others;

with (pingpong & modeflag) select
    RAMoe2 <=  RAMoe2f when "100" | "000" | "001" | "010",
               '0' when "101",
               '1' when "110",
               RAMoe2f when others;

dsp_BUSREQ <= RTS0_1;
dsp_ECLKOUT <= HREF_1; -- oranje draad
-- dsp_ADD <= "ZZZZZZZZZZZZZZZZZZZZ";
-- dsp_BE <= "ZZZZ";
dsp_CEO <= 'Z';      -- use this bit to send test serial data from FPGA
-- dsp_CE1 <= '0';   -- ground pin next to CE3, use this
                    -- bit to send test serial data from FPGA

```



```
-- dsp_CE2 <= 'Z';
-- dsp_CE3 <= 'Z';           -- use this bit to send data to display
-- dsp_ARE <= 'Z';
-- dsp_AWE <= 'Z';
-- dsp_AOE <= 'Z';
  dsp_ARDY <= '1';
  gen_led <= pingpong;       -- LED shows which pingpong we are in
  sclk <= clk_buf1;         -- write serial clock
  OLCC1 <= mclk;           -- SAA7127 clock
  RTCI <= 'Z';
  RCV_1 <= VREF_1;         -- VS
  RCV_2 <= HREF_1;         -- HS

  -- setup tri-state buffer for I2C logic
  SCL <= '0' when (SCL_out_flag = '0')
  SDA <= '0' when (SDA_out_flag = '0') else 'Z';

end architecture logic;
```



## C.2 CPLD Data Router VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity u12 is
  port(
    mclk      : in std_logic;
    latch_cntl : in std_logic_vector(4 downto 0);
    sclk      : in std_logic;

    serial_out : out std_logic;
    serial_in  : in std_logic;
    gen_led    : out std_logic;
    cam_data   : in std_logic_vector(12 downto 0);
    blue_data  : in std_logic_vector(2 downto 0);
    other_data : in std_logic_vector(4 downto 0);
    dsp_data   : out std_logic_vector(15 downto 0);

    out_data   : out std_logic_vector(7 downto 0);
    ram_col1_b1 : inout std_logic_vector(7 downto 0);
    ram_col1_b2 : inout std_logic_vector(7 downto 0);
    ram_col2_b1 : inout std_logic_vector(7 downto 0);
    ram_col2_b2 : inout std_logic_vector(7 downto 0));

end u12;

architecture logic of u12 is

  type statetype is (idle, transmit);
  signal state      : statetype;
  signal deb        : std_logic;

  signal latch_data4 : std_logic_vector(4 downto 0);
  signal latch_col1  : std_logic_vector(7 downto 0);
  signal latch_col2  : std_logic_vector(7 downto 0);
  signal enable      : std_logic_vector(1 downto 0);

begin

p0: process (sclk) is

  variable TX_data      : std_logic_vector(7 downto 0);
```

```
variable tbuf          :   std_logic_vector(9 downto 0);
variable bitcnt        :   integer range 0 to 9;

begin
  if rising_edge(sclk) then
    case state is
    when idle =>
      serial_out <= '1';
      gen_led <= '0';

      if latch_cntl(4) = '0' then
        deb <= '1';
      end if;

      if (latch_cntl(4) = '1') and deb = '1' then
        deb <= '0';

        if latch_cntl = "10000" then
          TX_data := ram_col1_b1;
        elsif latch_cntl = "10001" then
          TX_data := ram_col2_b1;
        elsif latch_cntl = "10010" then
          TX_data := ram_col1_b2;
        elsif latch_cntl = "10011" then
          TX_data := ram_col2_b2;
        else
          TX_data := "00110011";
        end if;

        tbuf(0) := '0';
        tbuf(8 downto 1) := TX_data(7 downto 0);
        tbuf(9) := '1';
        bitcnt := 0;
        state <= transmit;
      end if;

    when transmit =>
      serial_out <= tbuf(0);
      tbuf(8 downto 0) := tbuf(9 downto 1);
      tbuf(9) := '1';
      if bitcnt = 9 then
        state <= idle;
        gen_led <= '1';
      else
        bitcnt := bitcnt + 1;
      end if;
    end case;
  end if;
end;
```

```

    end case;
    end if;
end process p0;

```

```

p2: process (mclk) is
begin
    if rising_edge(mclk) then

        case latch_cntl is
            when "01100" | "01101" =>
                latch_data4 <= cam_data(4 downto 0);

            when "00001" | "00010" =>
                latch_col1 <= (cam_data(12 downto 8) & latch_data4(4 downto 2));
                latch_col2 <= (cam_data(7 downto 2) & latch_data4(1 downto 0));
                enable <= latch_cntl(1 downto 0);

            when others =>
                enable <= "00";
        end case;
    end if;
end process p2;

```

----- Signal Routing -----

```

dsp_data <= (ram_col2_b1 & ram_col1_b1) when latch_cntl = "00111" else
            (ram_col2_b2 & ram_col1_b2) when latch_cntl = "00100" else ...
            ... "ZZZZZZZZZZZZZZZZ";

out_data <= ram_col1_b1 when (latch_cntl = "01101") or (latch_cntl = "00100") or ...
            ... (latch_cntl = "00110") else
            ram_col1_b2 when (latch_cntl = "01100") or (latch_cntl = "00111") or ...
            ... (latch_cntl = "00101") else "ZZZZZZZZ" ;

ram_col1_b1 <= latch_col1 when enable = "01" else
            "01010101" when latch_cntl = "01110" else "ZZZZZZZZ";

ram_col2_b1 <= latch_col2 when enable = "01" else
            "01010101" when latch_cntl = "01110" else "ZZZZZZZZ";

ram_col1_b2 <= latch_col1 when enable = "10" else
            "01010101" when latch_cntl = "01110" else "ZZZZZZZZ";

ram_col2_b2 <= latch_col2 when enable = "10" else
            "01010101" when latch_cntl = "01110" else "ZZZZZZZZ";

```

---

```
end logic;
```



```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity u13 is
  port(
    mclk      : in std_logic;
    latch_cntl : in std_logic_vector(4 downto 0);
    sclk      : in std_logic;

    serial_out : out std_logic;
    serial_in  : in std_logic;
    gen_led    : out std_logic;
    blue_data  : in std_logic_vector(4 downto 0);
    other_data : in std_logic_vector(15 downto 0);
    dsp_dataL  : out std_logic_vector(7 downto 0);      -- blue
    dsp_dataH  : inout std_logic_vector(7 downto 0);   -- gray

    out_data   : out std_logic_vector(7 downto 0);
    ram_col1_b1 : inout std_logic_vector(7 downto 0);
    ram_col1_b2 : inout std_logic_vector(7 downto 0);
    ram_col2_b1 : inout std_logic_vector(7 downto 0);
    ram_col2_b2 : inout std_logic_vector(7 downto 0));

end u13;

architecture logic of u13 is

  type statetype is (idle, transmit);
  signal state      : statetype;
  signal deb        : std_logic;

  signal latch_data4 : std_logic_vector(4 downto 0);
  signal latch_data  : std_logic_vector(7 downto 0);

  signal latch_col2 : std_logic_vector(7 downto 0);
  signal latch_col1 : std_logic_vector(7 downto 0);
  signal enable     : std_logic_vector(1 downto 0);

begin

p0: process (sclk) is

  variable TX_data      : std_logic_vector(7 downto 0);
  variable tbuf         : std_logic_vector(9 downto 0);

```

```
variable bitcnt          :   integer range 0 to 9;

begin
  if rising_edge(sclk) then
    case state is
    when idle =>
      serial_out <= '1';
      gen_led <= '0';

      if latch_cntl(4) = '0' then
        deb <= '1';
      end if;

      if (latch_cntl(4) = '1') and deb = '1' then
        deb <= '0';

        if latch_cntl = "10000" then
          TX_data := ram_col1_b1;
        elsif latch_cntl = "10001" then
          TX_data := ram_col2_b1;
        elsif latch_cntl = "10010" then
          TX_data := ram_col1_b2;
        elsif latch_cntl = "10011" then
          TX_data := ram_col2_b2;
        else
          TX_data := "00110011";
        end if;

        tbuf(0) := '0';
        tbuf(8 downto 1) := TX_data(7 downto 0);
        tbuf(9) := '1';
        bitcnt := 0;
        state <= transmit;
      end if;

    when transmit =>
      serial_out <= tbuf(0);
      tbuf(8 downto 0) := tbuf(9 downto 1);
      tbuf(9) := '1';
      if bitcnt = 9 then
        state <= idle;
        gen_led <= '1';
      else
        bitcnt := bitcnt + 1;
      end if;
    end case;
  end case;
```

```

    end if;
end process p0;

```

```

p2: process (mclk) is
begin
    if rising_edge(mclk) then

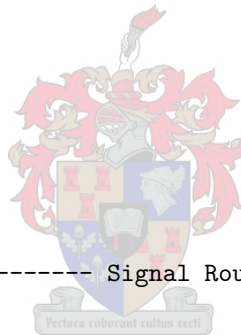
        case latch_cntl is
            when "01100" | "01101" =>
                latch_data4 <= blue_data(4 downto 0);
                latch_data <= other_data(15 downto 8);

            when "00001" | "00010" =>
                latch_col1 <= blue_data(4 downto 0) & latch_data4(2 downto 0);
                latch_col2 <= latch_data;

                enable <= latch_cntl(1 downto 0);

            when others =>
                enable <= "00";
        end case;
    end if;
end process p2;

```



----- Signal Routing -----

```

dsp_dataH <= ram_col2_b1 when latch_cntl = "00111" else
    ram_col2_b2 when latch_cntl = "00100" else "ZZZZZZZ";

dsp_dataL <= ram_col1_b1 when latch_cntl = "00111" else
    ram_col1_b2 when latch_cntl = "00100" else "ZZZZZZZ";

out_data <= ram_col2_b1 when (latch_cntl = "01101") or (latch_cntl = "00100") or ...
    ... (latch_cntl = "00110") else
    ram_col2_b2 when (latch_cntl = "01100") or (latch_cntl = "00111") or ...
    ... (latch_cntl = "00101") else "ZZZZZZZ";

ram_col1_b1 <= --dsp_data(15 downto 8) when latch_cntl = "01001" else
    latch_col1 when enable = "01" else
    "01010101" when latch_cntl = "01110" else "ZZZZZZZ";

ram_col2_b1 <= dsp_dataH when latch_cntl = "00101" else
    latch_col2 when enable = "01" else

```



```
        "01010101" when latch_cnt1 = "01110" else "ZZZZZZZZ";

ram_col1_b2 <= --dsp_data(15 downto 8) when latch_cnt1 = "01011" else
              latch_col1 when enable = "10" else
              "01010101" when latch_cnt1 = "01110" else "ZZZZZZZZ";

ram_col2_b2 <= dsp_dataH when latch_cnt1 = "00110" else
              latch_col2 when enable = "10" else
              "01010101" when latch_cnt1 = "01110" else "ZZZZZZZZ";

-----
end logic;
```



## C.3 Video Output Glue Logic and Data Router

### VHDL

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity u8 is
  port(  mclk      :   in std_logic;
        outcnt1   :   in std_logic_vector(4 downto 0);
        dataA     :   in std_logic_vector(7 downto 0);
        dataB     :   in std_logic_vector(7 downto 0);
        data_out  :   out std_logic_vector(7 downto 0));
end u8;


architecture logic of u8 is
  signal cnt : integer range 0 to 30000000;
  signal led : std_logic;
begin
-----
p0: process (mclk) is
  begin
    if rising_edge(mclk) then
      if cnt = 3000000 then
        cnt <= 0;
        led <= not(led);
      else
        cnt <= cnt+1;
      end if;
    end if;
  end process p0;
-----
p1: process (mclk) is
  begin
    if rising_edge(mclk) then
      case outcnt1 is
        -- out data options:  80H, 10H, 00, FF, EAV, SAV, Y, Cb, Cr
        -- EAV / SAV options:
        -- 1XXX0000 -> 1FVH0000

        when "00000" =>
          -- do nothing
          data_out <= "ZZZZZZZZ";

```

```
when "00001" =>
    data_out <= "10000000";    -- 80H, also for Cb = Cr = 128 = 80H
when "00010" =>
    data_out <= "00010000";    -- 10H
when "00011" =>
    data_out <= "00000000";    -- 00H
when "00100" =>
    data_out <= "11111111";    -- FFH
when "00101" =>
    data_out <= dataA;         -- Y, RAM A
when "00110" =>
    data_out <= dataB;         -- Y, RAM B
when "11000" | "11001" | "11010" | "11011" | "11100" | "11101" | ...
    ... "11110" | "11111" =>
    data_out <= "1" & outcntl(2 downto 0) & "0000";
when "01000" =>
    data_out <= "11101011";
when "01001" =>
    data_out <= "01110101";
when others =>
    -- do nothing
    data_out <= "ZZZZZZZZ"; -- just to be safe

end case;
end if;
end process p1;
end logic;
```



---

## C.4 DSP Code

```

#include "emif.h"
/* #include <c6x.h> */
#include "c6211dsk.h"
/*#include "intr.h"*/

/*-----*/
/* Function Prototypes */
/* */
void rising_RTS0(void);
void rising_dataready(void);
void drawblock(unsigned int x);
void drawtrack(unsigned int x);
void preprocess(unsigned int x);
unsigned int user_centroid(void);
unsigned int read_RAM(unsigned int x, unsigned short iter);
/* */
/*-----*/
/* Global Variables */
/* */
far unsigned int subm[60][200]; /* was [60][200] */
far unsigned int cor_mat[15][50];

double rot_vector[4] = {0, 0, 0, 0}; /* red, green, blue, gray */
double temp_vector[4]; /* red, green, blue, gray */
int hor_v, vert_v, rowk, rowk_1, colk, colk_1;

/* Default Block Size */
unsigned short blockwidth = 200;
unsigned short blockheight = 60;
/* */
/*-----*/

void main()
{
unsigned int centroid;
unsigned int pingpong, modeflag0, modeflag1, statecnt;
unsigned int setup_flag, scan_flag, user_hor, user_vert;
unsigned short iteration = 0;
/*-----*/
/* EMIF SETUP */
/* */

/* Get default values for all EMIF registers */

```



```

/* L2CFG = 0x1840000, c6211dsk.h */
/*REG_WRITE(L2CFG, 0x80000000);*/
REG_WRITE(L2CFG, 0x80000000);
/*-----*/

/*-----*/
/* */
/*          STAT/ */
/*          CNTL 1 0 */
/*          | | */
/* IO_PORT = 0x ZZZX XXXX 0000 0000 0000 0000 0000 */
/*          210  321 */
/*          BRD  LED */
/*          /DIP_SWITCH */
/* */
/*-----*/

/*-----*/
/* Read DIP switches */
/* ( user_dip_settings & 0x1000000 ) >> 24 ) dipswitch 1 */
/* ( user_dip_settings & 0x2000000 ) >> 25 ) dipswitch 2 */
/* ( user_dip_settings & 0x4000000 ) >> 26 ) dipswitch 3 */
/* */
/*-----*/

/*-----*/
/* Interrupt clear and initialisation */
/* */
/*          CSR =  0x100;          /* disable all interrupts */
/* */
/*-----*/

/*-----*/
/* Initialise Variables */

pingpong   = 0x0;
modeflag0  = 0x0;
modeflag1  = 0x0;
statecnt   = 0;

setup_flag = 0x1;
scan_flag  = 0x0;
user_hor   = 230;
user_vert  = 62;
hor_v      = 0;
vert_v     = 0;

```

```

centroid = 86210;

/*-----*/
while(1)
{
    /* Detecting falling edge and debouncing noisy input signal - RTS0 */
    rising_RTS0();

    switch (statecnt)
    {
    case 0:
        modeflag0 = 0x0;
        modeflag1 = 0x0;
        pingpong = pingpong ^ 0x1;
        statecnt = statecnt + 1;
        break;

    case 1:
        modeflag0 = 0x1;
        modeflag1 = 0x0;
        statecnt = statecnt + 1;
        break;

    case 2:
        modeflag0 = 0x0;
        modeflag1 = 0x1;
        statecnt = 0;
        break;
    }

    /* Write STATE to Framegrabber */
    REG_WRITE(IO_PORT, ((pingpong << 24) | (modeflag1 << 25) | ...
    ... (modeflag0 << 26) | (1 << 27) | (0 << 28)) & 0x1f000000);

    /* Manually Set States */
    /* temp_io = REG_READ(IO_PORT) & 0x07000000;
    modeflag1 = (temp_io >> 25) & 0x1;
    modeflag0 = (temp_io >> 26) & 0x1;
    pingpong = (temp_io >> 24) & 0x1;
    REG_WRITE(IO_PORT, ((pingpong << 24) | (modeflag1 << 25) | ...
    ... (modeflag0 << 26) | (1 << 27) | (0 << 28)) & 0x1f000000);
    */

    /* Wait for Framegrabber to signal that data is ready */

```

```

rising_dataready();

/* Setup - User Centres Block Around Target */
/* Initial Position: Row - 62-206, Col - 230-550 */

if ((modeflag1 == 1) & (modeflag0 == 0) & (setup_flag == 1)) {
    if (GET_BIT(IO_PORT,24) == 0) {
        if (user_hor < 550) {
            user_hor = user_hor + 10;
        }
        else {
            user_hor = 230;
        }
    }

    if (GET_BIT(IO_PORT,25) == 0) {
        if (user_vert < 206) {
            user_vert = user_vert + 4;
        }
        else {
            user_vert = 62;
        }
    }

    if (GET_BIT(IO_PORT,26) == 0) {
        setup_flag = 0;
        scan_flag = 1;
    }

    centroid = user_hor + user_vert*640;
    rowk = centroid/640;
    colk = centroid%640;
    rowk_1 = rowk;
    colk_1 = colk;
    drawblock(centroid);
}

/* Preprocess - Define Colour Transfrom Vectors */
if ((modeflag0 == 1) & (modeflag1 == 0) & (setup_flag == 0) & (scan_flag == 1)) {
    preprocess(centroid);
    scan_flag = 0;
}

/* Write to RAM */
if ((modeflag1 == 1) & (modeflag0 == 0) & (setup_flag == 0) & (scan_flag == 0))
{

```



```

REG_WRITE((0xA0000000 + 4*centroid),0xffffffff);
REG_WRITE((0xA0000000 + 4*(centroid+153600)),0xffffffff);

drawblock(centroid);
if (GET_BIT(IO_PORT,26) == 0) {drawtrack(134945);}
/*drawtrack(centroid);*/
}

/* Read from RAM */
/* & 0xff000000 >> 24 - GRAY */
/* & 0x00ff0000 >> 16 - BLUE */
/* & 0x0000ff00 >> 8 - GREEN */
/* & 0x000000ff - RED */

/* Read from RAM */
if ((modeflag0 == 1) & (modeflag1 == 0) & (setup_flag == 0) & (scan_flag == 0)) {
    REG_WRITE(IO_PORT, ((pingpong << 24) | (modeflag1 << 25) | (modeflag0 << 26) ...
    ... | (1 << 27) | (1 << 28)) & 0x1f000000);
    centroid = read_RAM(centroid, iteration);
    if (iteration == 0) { iteration = 1;}
    if (iteration == 1) { iteration = 2;}
}

REG_WRITE(IO_PORT, ((pingpong << 24) | (modeflag1 << 25) | (modeflag0 << 26) | ...
... (0 << 27) | (0 << 28)) & 0x0f000000);

} /* End While */
} /* End MAIN */

/*-----*/
void rising_RTS0()
{
    /* declare vars */
    unsigned int edge, edge_rem;

    /* initialise vars */
    edge = 1;
    edge_rem = 0;

    /* Detecting falling edge and debouncing noisy input signal - RTS0 */
    while( ((GET_BIT(IO_PORT, 27) == 0) | edge))
    {
        if (GET_BIT(IO_PORT,27) == 0) {
            edge_rem = 1;
        }
        if (edge_rem) {

```

```

        if (GET_BIT(IO_PORT,27) == 0) {
            edge = 0;
        }
        else {
            edge_rem = 0;
        }
    }
}
/* reset vars */
edge = 1;
edge_rem = 0;
}
/*-----*/
void rising_dataready()
{
    /* declare vars */
    unsigned int edge, edge_rem;

    /* initialise vars */
    edge      = 1;
    edge_rem  = 0;

    /* Detecting falling edge and debouncing noisy input signal - RTS0 */
    while( ((GET_BIT(IO_PORT, 28) == 0) | edge))
    {
        if (GET_BIT(IO_PORT,28) == 0) {
            edge_rem = 1;
        }
        if (edge_rem) {
            if (GET_BIT(IO_PORT,28) == 0) {
                edge = 0;
            }
            else {
                edge_rem = 0;
            }
        }
    }
    /* reset vars */
    edge = 1;
    edge_rem = 0;
}
/*-----*/

/*-----*/
void drawblock(unsigned int x)
{

```

```

/* Declare Vars */
const unsigned short horline[21] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...
... 11, 12, 13, 14, 15, 16, 17, 18, 19, 20};
const unsigned short vertline[8] = {640, 1280, 1920, 2560, 3200, 3840, ...
... 4480, 5120};

unsigned short count;
unsigned int interlace;

/* Second half of interlace */
count = 0;
interlace = 153600;

/* Draw Center Dot */
REG_WRITE((0xA0000000 + 4*x),0xffffffff);
REG_WRITE((0xA0000000 + 4*(x+interlace)),0xffffffff);

/* justify block so x is in center */
x = x - (blockwidth/2) - (blockheight/2)*640;

/* Draw Horizontal Lines */
while (count < 21) {
    REG_WRITE((0xA0000000 + 4*(horline[count] + (x))),0xffffffff);
    REG_WRITE((0xA0000000 + 4*(horline[count] + (x+interlace))),0xffffffff);

    REG_WRITE((0xA0000000 + 4*(horline[count] + ...
... (x+blockheight*640))),0xffffffff);
    REG_WRITE((0xA0000000 + 4*(horline[count] + ...
... x+interlace+blockheight*640)),0xffffffff);

    REG_WRITE((0xA0000000 + 4*(-horline[count] + (x+blockwidth))),0xffffffff);
    REG_WRITE((0xA0000000 + 4*(-horline[count] + ...
... (x+blockwidth+interlace))),0xffffffff);

    REG_WRITE((0xA0000000 + 4*(-horline[count] + ...
... (x+blockwidth+blockheight*640))),0xffffffff);
    REG_WRITE((0xA0000000 + 4*(-horline[count] + ...
... (x+blockwidth+(blockheight*640)+interlace))),0xffffffff);
    count++;
}

/* Draw Vertical Lines */
count = 0;
while (count < 8) {
    REG_WRITE((0xA0000000 + 4*(vertline[count] + (x))),0xffffffff);
    REG_WRITE((0xA0000000 + 4*(vertline[count] + (x+interlace))),0xffffffff);

```

```

    REG_WRITE((0xA0000000 + 4*(-vertline[count] + ...
    ... (x+blockheight*640))),0xffffffff);
    REG_WRITE((0xA0000000 + 4*(-vertline[count] + ...
    ... x+interlace+blockheight*640)),0xffffffff);

    REG_WRITE((0xA0000000 + 4*(+vertline[count] + (x+blockwidth))),0xffffffff);
    REG_WRITE((0xA0000000 + 4*(+vertline[count] + ...
    ... (x+blockwidth+interlace))),0xffffffff);

    REG_WRITE((0xA0000000 + 4*(-vertline[count] + ...
    ... (x+blockwidth+blockheight*640))),0xffffffff);
    REG_WRITE((0xA0000000 + 4*(-vertline[count] + ...
    ... (x+blockwidth+(blockheight*640)+interlace))),0xffffffff);
    count++;
}

}

/*-----*/
unsigned int read_RAM(unsigned int x, unsigned short iter)
{
    /* declare vars */
    unsigned int row, col, threshold, threshcnt, rowavg, colavg, absrow, ...
    ... abscol, control_row, control_col;
    double absrow_new, abscol_new;
    unsigned int maxi;
    int temp_data, max_pixel;
    /*unsigned int cor_mat[60][200];*/

    unsigned int row_prev, col_prev;
    /*double K = 0.5;*/

    int max_delay_v, max_delay_h, delay_v, delay_h, iv, ih, jv, jh, max_v, ...
    ... max_h, nh, nv;
    int sxy, max_sxy;
    short temp_row, temp_col, mask_update;

    nh = 50;
    nv = 15;
    max_delay_h = 50-1;
    max_delay_v = 15-1;

    /* initialise vars */
    row_prev = x/640;
    col_prev = x%640;
    row = 0;
    col = 0;

```

```

maxi = 0;
threshold = 10;
threshcnt = 0;
rowavg = 0;
colavg = 0;
absrow = 0;
abscol = 0;
max_pixel = 0;

/* Read image from RAM */
row = 0;
while (row < 60) {
    col = 0;
    while (col < 200) {
        subm[row][col] = (REG_READ(0xA0000000 + 4*((row+((x/640) - ...
        ... blockheight/2))*640 + col+((x%640) - blockwidth/2))) & 0xffffffff);
        col = col + 1;
    }
    row = row + 1;
}

/*
    col = 0;
    while (col < 200) {
        subm[row][col] = (REG_READ(0xA0000000 + 4*((row+((x/640) - 30))*640 + ...
        ... 153600 + col+((x%640) - 100))) & 0xff000000) >> 24;
        col = col + 1;
    }
    row = row + 1;
*/
}

/* Do Colour Transformation */
for (row = 0; row < 60; row++) {
    for (col = 0; col < 200; col++) {
        temp_data = (int)(rot_vector[0]*(double)(subm[row][col] & 0xff) + ...
        ... rot_vector[1]*(double)((subm[row][col] & 0xff00) >> 8) + ...
        ... rot_vector[2]*(double)((subm[row][col] & 0xff0000) >> 16));

        if (temp_data > 0) {
            subm[row][col] = temp_data;
        }
        else {
            subm[row][col] = 0;
        }
        if (temp_data > max_pixel) { max_pixel = temp_data; }
    }
}

```

```

    /*temp_vector[3] = max_pixel;*/

/*-----*/
/* NOW - HOTTEST SPOT TRACKING OR CORRELATION */

/* Threshold / Normalize */
/* Threshold - for Hot Spot tracking */
/* Normalize - for display and Correlation Tracking */
threshold = 2*max_pixel/3;
for (row = 0; row < 60; row++) {
    for (col = 0; col < 200; col++) {
        if (((subm[row][col]) & 0xff) > threshold) {
            /*subm[row][col] = 0xff;*/
            subm[row][col] = 254*subm[row][col]/max_pixel;
            threshcnt = threshcnt + 1;
            rowavg = rowavg + row;
            colavg = colavg + col;
        }
        else {
            /*subm[row][col] = 0x00;*/
            subm[row][col] = 254*subm[row][col]/max_pixel;
        }
    }
}

/* Get average value of points > threshold */
row = rowavg/threshcnt;
col = colavg/threshcnt;
absrow = row+(x/640)-blockheight/2;
abscol = col+(x%/640)-blockwidth/2;

/* Low Pass Filter */
/* absrow_new = ((1-K)*row_prev + K*absrow);
abscol_new = ((1-K)*col_prev + K*abscol);
*/
/* absrow_new = (row_prev/2 + absrow/2);
abscol_new = (col_prev/2 + abscol/2);
*/
absrow_new = absrow;
abscol_new = abscol;

/*-----*/
/* Correlation Tracking */

/* First iteration - no correlation template, skip corr, return input centroid */
if (iter == 0) {

```

```

    absrow_new = row_prev;
    abscol_new = col_prev;
    max_v = 0;
    max_h = 0;
    mask_update = 0;
}
/* Have Corr mask, so do corr */
else {

/* Calculate the correlation series */
max_sxy = -100;
/*temp_vector[2] = 0;*/
for (delay_h = -max_delay_h; delay_h < max_delay_h ;delay_h++) {
    for (delay_v = -max_delay_v; delay_v < max_delay_v ;delay_v++) {

        sxy = 0;

        for (iv = 0; iv < nv; iv++) {
            jv = iv + delay_v;

            if (jv < 0 || jv >= nv)
                {}
            else {

                for (ih = 0; ih < nh; ih++) {
                    jh = ih + delay_h;

                    if (jh < 0 || jh >= nh)
                        {}
                    else {
                        sxy = sxy + (cor_mat[iv][ih])*(subm[4*jv][4*jh]);
                        if (sxy > max_sxy) {
                            max_sxy = sxy;
                            max_v = delay_v;
                            max_h = delay_h;
                        }
                    }
                }
            }
        }
    }
}

absrow_new = (4*max_v + 30)+(x/640)-blockheight/2;
abscol_new = (4*max_h + 100)+(x%640)-blockwidth/2;

```

```

} /* ELSE */

/* temp_vector[0] = row;
   temp_vector[1] = col;
   temp_vector[2] = max_v*4;
   temp_vector[3] = max_h*4;
*/

/* Get new Correlation Mask */

if (mask_update == 10) {
    mask_update = 0;
}
else {
    mask_update++;
    row = 0;
    while (row < 15) {
        col = 0;
        while (col < 50) {
            temp_row = 4*row + 4*max_v;
            temp_col = 4*col + 4*max_h;
/*            temp_row = 4*row;
            temp_col = 4*col;*/
            if ((temp_row >= 0) & (temp_row < 60) & (temp_col >= 0) & ...
                ... (temp_col < 200)) {
                cor_mat[row][col] = subm[temp_row][temp_col];
            }
            else {
                cor_mat[row][col] = 0x00;
            }
            col = col + 1;
        }
        row = row + 1;
    }
} /* ELSE */

/* Display Correlation Mask */
/* row = 0;
   while (row < 15) {
       col = 0;
       while (col < 50) {
           subm[row][col] = subm[4*row][4*col];
           col = col + 1;
       }
       row = row + 1;
   }
}

```



```

*/
/*-----*/

/* control_row = (absrow + (pos_vector[1]/640))/2 + ...
   ... ((pos_vector[1]/640)-(pos_vector[0]/640));
   control_col = (abscol + (pos_vector[1]%640))/2 + ...
   ... ((pos_vector[1]%640)-(pos_vector[0]%640));
*/
/* control_row = (absrow + (pos_vector[1]/640))/2;
   control_col = (abscol + (pos_vector[1]%640))/2;
*/

control_row = (absrow + rowk)/2 + (rowk - rowk_1)/2;
control_col = (abscol + colk)/2 + (colk - colk_1)/2;
/*control_row = absrow;
control_col = abscol;*/

rowk_1 = rowk;
colk_1 = colk;
rowk = absrow;
colk = abscol;

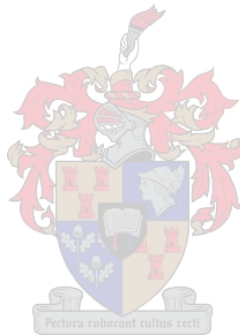
/* absrow_new = control_row;
   abscol_new = control_col;
*/
temp_vector[0] = absrow;
temp_vector[1] = abscol;
temp_vector[2] = control_row;
temp_vector[3] = control_col;

/* Keep Centroid in Frame Limits */
if (absrow > 206) { absrow = 206; }
if (absrow < 62 ) { absrow = 62; }
if (abscol > 550) { abscol = 550; }
if (abscol < 230 ) { abscol = 230; }

maxi = (int)absrow_new*640 + (int)abscol_new;
return maxi;
}
/*-----*/
void drawtrack(unsigned int x)
{
    unsigned int row, col, rowoffset, coloffset;

    rowoffset = x/640 - blockheight/2;
    coloffset = x%640 - blockwidth/2;

```



```

for (row = 0; row < 60; row++) {
    for (col = 0; col < 200; col++) {
        REG_WRITE((0xA0000000 + 4*((row+rowoffset)*640 + ...
        ... (col+coloffset))), (subm[row][col] << 24);
        REG_WRITE((0xA0000000 + 4*((row+rowoffset)*640 + ...
        ... (col+coloffset) + 153600)), (subm[row][col] << 24);
    }
}
}
/*-----*/
void preprocess(unsigned int x)
{
    /* Declare Vars */
    unsigned int temp_input, pixelcnt, tcnt, bcnt;
    unsigned int tred, tblue, tgreen, tgray;
    unsigned int bred, bblue, bgreen, bgray;

    double norm_tv[3], norm_bv[3], n_tv, n_tv1, n_bv, n_bv1, n_rt, n_rt1;
    double target_vector[4] = {0, 0, 0, 0}; /* red, green, blue, gray */
    double background_vector[4] = {0, 0, 0, 0}; /* red, green, blue, gray */

    unsigned short i;
    /*unsigned short row, col; */

    /* Init Vars */
    temp_input = 0;

    tred = 0;
    tgreen = 0;
    tblue = 0;
    tgray = 0;
    tcnt = 0;

    bred = 0;
    bgreen = 0;
    bblue = 0;
    bgray = 0;
    bcnt = 0;

    n_tv = 0;
    n_bv = 0;
    n_tv1 = 0;
    n_bv1 = 0;
    i = 0;

    /* Read Entire Frame, Get Spectral data of target vs background */

```

```

/* pixelcnt = 0;
   while (pixelcnt < 307200) {
       temp_input = REG_READ(0xA0000000 + 4*pixelcnt);

       if (((pixelcnt/640) > (x/640 - blockheight/2)) & ...
           ... ((pixelcnt/640) < (x/640 + blockheight/2)) & ...
           ... ((pixelcnt%640) > (x%640 - blockwidth/2)) & ...
           ... ((pixelcnt%640) < (x%640 + blockwidth/2))) {
           tred = tred + ((temp_input) & 0xff);
           tgreen = tgreen + ((temp_input & 0xff00) >> 8);
           tblue = tblue + ((temp_input & 0xff0000) >> 16);
           tgray = tgray + ((temp_input & 0xff000000) >> 24);
           tcnt = tcnt + 1;
       }
       else {
           bred = bred + ((temp_input) & 0xff);
           bgreen = bgreen + ((temp_input & 0xff00) >> 8);
           bblue = bblue + ((temp_input & 0xff0000) >> 16);
           bgray = bgray + ((temp_input & 0xff000000) >> 24);
           bcnt = bcnt + 1;
       }
       pixelcnt++;
   }

   /* Get Avg Target Colour Values */
   target_vector[0] = (double)tred/tcnt;
   target_vector[1] = (double)tgreen/tcnt;
   target_vector[2] = (double)tblue/tcnt;
   target_vector[3] = (double)tgray/tcnt;

   /* Get Avg Background Colour Values */
   background_vector[0] = (double)bred/bcnt;
   background_vector[1] = (double)bgreen/bcnt;
   background_vector[2] = (double)bbblue/bcnt;
   background_vector[3] = (double)bgray/bcnt;

   /* Normalize Target and Background Colour Vectors */
   n_tv = target_vector[0]*target_vector[0] + ...
   ... target_vector[1]*target_vector[1] + target_vector[2]*target_vector[2];
   n_bv = background_vector[0]*background_vector[0] + ...
   ... background_vector[1]*background_vector[1] + ...
   ... background_vector[2]*background_vector[2];

   n_tv1 = 100.0;
   for (i = 0; i < 10; i++) {
       n_tv1 = n_tv1/2 + n_tv/(2*n_tv1);

```

```

    }

    n_bv1 = 100.0;
    for (i = 0; i < 10; i++) {
        n_bv1 = n_bv1/2 + n_bv/(2*n_bv1);
    }

    norm_tv[0] = target_vector[0]/n_tv1;
    norm_tv[1] = target_vector[1]/n_tv1;
    norm_tv[2] = target_vector[2]/n_tv1;

    norm_bv[0] = background_vector[0]/n_bv1;
    norm_bv[1] = background_vector[1]/n_bv1;
    norm_bv[2] = background_vector[2]/n_bv1;

    /* Get Colour Rotation Vector */
/*   rot_vector[0] = norm_tv[0] - norm_bv[0];
    rot_vector[1] = norm_tv[1] - norm_bv[1];
    rot_vector[2] = norm_tv[2] - norm_bv[2];

    /* Normalize Rotation Vector */
/*   n_rt = rot_vector[0]*rot_vector[0] + rot_vector[1]*rot_vector[1] + ...
    ... rot_vector[2]*rot_vector[2];

    n_rt1 = 1.0;
    for (i = 0; i < 10; i++) {
        n_rt1 = n_rt1/2 + n_rt/(2*n_rt1);
    }

    rot_vector[0] = rot_vector[0]/n_rt1;
    rot_vector[1] = rot_vector[1]/n_rt1;
    rot_vector[2] = rot_vector[2]/n_rt1;

    temp_vector[0] = rot_vector[0];
    temp_vector[1] = rot_vector[1];
    temp_vector[2] = rot_vector[2];
*/
    rot_vector[0] = -0.8236;
    rot_vector[1] = 0.4106;
    rot_vector[2] = 0.3911;
}

/*-----*/

```

## C.5 MATLAB Code

```
% ----- %
% binread.m
% Christiaan Wood
% read binary data from .txt file and display

%clear all;
figure;
width = 640;
height = 480; % must be even
inputdata = 'ramb3.txt';

% read data into variable
picdata = uint8(textread(inputdata,'%n','delimiter',' '));

% arrange array into a 640x480 matrix
picdata2 = uint8([]); for i = 0:height-1,
    picdata2(i+1,:) = picdata((i*width)+1:(i*width)+width)';
end

% rearrange interlaced image to display correctly
picdata3 = [];
for i = 1:height/2,
    picdata3 = [picdata3; picdata2(i,:); picdata2(i+height/2,:)];
end

% display image
imshow(picdata3);

% ----- %
```

```

% ----- %
% colourspace2.m
% Christiaan Wood
% plots input image in RGB space

close all;
clear all;

% read image
img = imread('smallersmarties2.jpg');
HSV = rgb2hsv(img);
[im,jm,zm] = size(img);
blankimg = uint8(zeros(im,jm));

figure;
hold on;
remRGB2 = [];

% plot individual pixels in RGB space in their correct colour
for i = 1:5:im,
    for j = 1:5:jm,
        if ((RECT(2) < i)&(i < RECT(2)+RECT(4))) & ...
            ... ((RECT(1) < j)&(j < RECT(1)+RECT(3))),
            normRGB = double([img(i,j,1) img(i,j,2) img(i,j,3)]');
            remnorm = (normRGB(1) + normRGB(2) + normRGB(3))/3;
            normRGB = normRGB/remnorm;
            if ((remnorm > 100)),
                remRGB2 = [remRGB2; normRGB(1) normRGB(2) normRGB(3) ...
                    ... i j [squeeze(double(img(i,j,:))/255)'] ];
                plot3(normRGB(1),normRGB(2),normRGB(3),'.', ...
                    ... 'color',[squeeze(double(img(i,j,:))/255)']);
            end
        else
            plot3(img(i,j,1),img(i,j,2),img(i,j,3),'o','color', ...
                ... [squeeze(double(img(i,j,:))/255)']);
        end
    end
end

% Display properly
set(gca,'XDir','reverse');
set(gca,'YDir','reverse');
grid on;
axis square;
xlabel('Red');
ylabel('Green');

```

```

xlabel('Blue');

% Kmeans clustering
numc = 7; % number of clusters (for smarties)

[idx C] = kmeans(remRGB2(:,1:3),numc); % determine clusters

% plot images with only pixels belonging to same cluster
for idxindex = 1:numc,
    blankimg = uint8(zeros(im,jm));
    for i = 1:length(idx),
        if idx(i) == idxindex,
            blankimg(remX(i),remY(i)) = uint8(255*(remRGB2(i,1) + ...
                ... remRGB2(i,2) + remRGB2(i,3))/3);
        end
    end
    figure;
    imshow(blankimg);
end

% Plot clusters in RGB space
for idxindex = 1:numc,
    figure;
    grid on;
    for i = 1:length(idx),
        if idx(i) == idxindex,
            hold on;
            plot3(remRGB2(i,1),remRGB2(i,2),remRGB2(i,3),'.', ...
                ... 'color',remRGB2(i,6:8));
            end
        end
    hold off;
    set(gca,'XDir','reverse');
    set(gca,'YDir','reverse');
    grid on;
    axis square;
    xlabel('Red');
    ylabel('Green');
    zlabel('Blue');
end

% Polar conversion plots
center = [1;1;1];
%center = [mean(remRGB2(:,1) mean(remRGB2(:,2) mean(remRGB2(:,3)
dist = [(remRGB2(:,1) - center(1)) (remRGB2(:,2) -center(2)) ...

```



```
... (remRGB2(:,3) - center(3))];

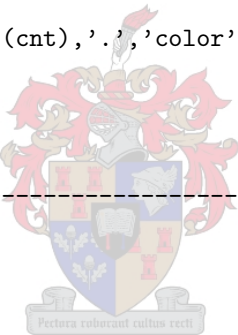
sizedist = [];
for cnt = 1:length(idx),
    sizedist = [sizedist norm(dist(cnt,1:3))];
end

refvector = dist(1,:);
sizeref = norm(refvector);

vectangle = [];
for cnt = 1:length(idx),
    temp = (dot(refvector,dist(cnt,:)))/(sizeref*sizedist(cnt));
    vectangle = [vectangle (temp)];
end

figure;
for cnt = 1:length(idx),
    hold on;
    plot(vectangle(cnt),sizedist(cnt),'.','color',remRGB2(cnt,6:8));
end

% ----- %
```

A watermark of a university crest is centered on the page. The crest features a shield with various symbols, topped by a crown and a crest with a bird. Below the shield is a motto scroll with the Latin text "Pectora roburant cultus recti".



```

% ----- %
% tg.m
% Christiaan Wood
% target background intensity experiments

close all;
clear all;

i = imread('feather3.jpg');

% Do HSV conversion
HSV = rgb2hsv(i);
% Do grayscale conversion
igray = double(rgb2gray(i));

% experiments in saturation and value equalization
itest = HSV(:,:,2).*igray;
sat = HSV(:,:,2)./(max(max(HSV(:,:,2))));
[h,w] = size(i(:,:,1));
%figure; imshow(i);
equalsat = 0.75*ones(h,w);
HSVequalsat = HSV;
%HSVequalsat(:,:,2) = equalsat;
HSVequalsat(:,:,3) = equalsat;

i2 = hsv2rgb(HSVequalsat);
%i = i2;

% Get grayscale intensity images representing each colour
ir = (i(:,:,1)); ig = (i(:,:,2)); ib = (i(:,:,3));
%figure; imshow(ir);
%figure; imshow(ig);
%figure; imshow(ib);

% Normalise each colour individually (NOT USED)
norm_ir = double(ir)/(max(max(double(ir))));
%figure; imshow(norm_ir);

norm_ig = double(ig)/(max(max(double(ig))));
%figure; imshow(norm_ig);

norm_ib = double(ib)/(max(max(double(ib))));
%figure; imshow(norm_ib);

% Combine normalised colours to create pseudo colour image

```



```

norm_i(:,:,1) = norm_ir;
norm_i(:,:,2) = norm_ig;
norm_i(:,:,3) = norm_ib;
%figure; imshow(norm_i);

% User selects target
figure;
[target, rect2] = imcrop(i);
[h2,w2] = size(target(:,:,1));
rect2 = round(rect2);
figure; imshow(target);
% gives target colour vector between 0,1
target_vector = (squeeze(mean(mean(target))))/255;
% normalises colour vector
norm_target_vector = target_vector/(norm(target_vector));

% User selects background - can do automatically, just use whole image.
% (or whole image, but not target area)
figure;
[background, rect3] = imcrop(i);
[h3,w3] = size(background(:,:,1));
rect3 = round(rect3);
figure; imshow(background);
% gives background colour vector between 0,1
background_vector = (squeeze(mean(mean(background))))/255;
% normalises colour vector
norm_background_vector = background_vector/(norm(background_vector));

% should the target and background vectors be normalised????
% normalise them, and we optimise for colour (not intensity)
%rot_vector = target_vector - background_vector;
rot_vector = norm_target_vector - norm_background_vector;

% Plot original image
close all; figure; imshow(i); figure;

% Plot colour and rotation vectors
plot3([0 norm_target_vector(1)], [0 norm_target_vector(2)], ...
... [0 norm_target_vector(3)], 'b');
grid on;
hold on;
plot3([0 norm_background_vector(1)], [0 norm_background_vector(2)], ...
... [0 norm_background_vector(3)], 'r');

```

```

plot3([0 rot_vector(1)], [0 rot_vector(2)], [0 rot_vector(3)], 'g');

title('Target Colour - BLUE, Background Colour - RED, Rotation Vector - GREEN');
xlabel('Red');
ylabel('Green');
zlabel('Blue');

% Create Image to use for tracking
tracking_image = double(i(:,:,1))*rot_vector(1) + ...
... double(i(:,:,2))*rot_vector(2) + double(i(:,:,3))*rot_vector(3);

% Contrast = difference between highest and lowest value
contrast = (max(max(tracking_image)) - min(min(tracking_image)));
% Adjust image to make all values positive, and optimal contrast
opt_tracking_image = (tracking_image + abs(min(min(tracking_image))))/contrast;

% Adjust image to optimise positive value's contrast (throw away neg values)
pos_tracking_image = tracking_image/(max(max(tracking_image)));

% normalised rot vector tracking image
norm_rot_vector = rot_vector/(norm(rot_vector));
norm_tracking_image = double(i(:,:,1))*norm_rot_vector(1) + ...
... double(i(:,:,2))*norm_rot_vector(2) + double(i(:,:,3))*norm_rot_vector(3);

%figure; imshow(uint8(norm_tracking_image));
figure; imshow(opt_tracking_image);
figure; imshow(pos_tracking_image);
figure; imshow(norm_tracking_image);

% Determine Tracking Confidence
confidence = 100*(norm(rot_vector',3)/1.4422);
xlbl = num2str(confidence);
xlbl = ['Confidence = ',xlbl];
xlabel(xlbl);

% ----- %

```

```

% ----- %
% autotarget.m
% Christiaan Wood
% experiments in automatic target detection

close all;
clear all;
i = imread('smallsmarties.jpg');

% pixel resolution in which to search for target
increment = 10;
[h,w] = size(i(:,:,1));

% User defines target size - only rect2 size is used, NOT position
figure; [target, rect2] = imcrop(i);
[h2,w2] = size(target(:,:,1));
rect2 = round(rect2);

% Define whole image as background
background = i;
[h3,w3] = size(background(:,:,1));

% gives background colour vector between 0,1
background_vector = (squeeze(mean(mean(background))))/255;
% normalises colour vector
norm_background_vector = background_vector/(norm(background_vector));

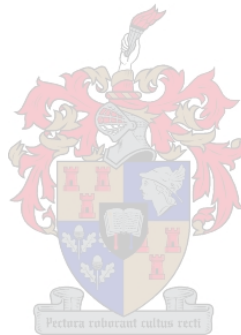
target_mat = zeros(h,w);
for cnt_i = 1:increment:h-h2,
    for cnt_j = 1:increment:w-w2,
        test_img = imcrop(i,[cnt_j cnt_i rect2(3) rect2(4)]);
        test_vector = (squeeze(mean(mean(test_img))))/255;
        norm_test_vector = test_vector/(norm(test_vector));
        % if (max(test_vector) == test_vector(3)), % define desired colour of target
            target_mat(cnt_i,cnt_j) = norm(norm_test_vector - norm_background_vector);
        % end
    end
end

% find location of largest difference vector
[i_loc, j_loc] = find(target_mat == max(max(target_mat)));

% Draw track overlay
i(i_loc:i_loc+h2,j_loc,:) = 255;
i(i_loc:i_loc+h2,j_loc+w2,:) = 255;

```

```
i(i_loc,j_loc:j_loc+w2,:) = 255;  
i(i_loc+h2,j_loc:j_loc+w2,:) = 255;  
  
% Display probable target  
figure; imshow(i);  
  
% ----- %
```



```

% ----- %
% viddetectRGB.m
% Christiaan Wood
% same as viddetect.m, but input file MUST be RGB, and viddetectRGB.m
%   tracks all three colours
% testing video tracking
% if .avi grayscale - comment out 't1 = rgb2gray(t1)' and 't2 = rgb2gray(t2)'
%                   - comment in 'Mf(firstframe) = im2frame(uint8(fmask),MAP)'
%                   in both places
% also, check uint8/uint16 when making movie (this has been automated)

close all;
clear all;
warning off MATLAB:mir_warning_variable_used_as_function;

activecolor = 1;           % 1 - 3; Red = 1, Green = 2, Blue = 3
firstframe = 1;           % frame to start tracking on
MAP = gray(256);          % define colormap to convert RGB to grayscale
center = [];

filein = 'fighterplane.avi'; % filename of .avi file to read in
aviobj = aviread(filein);    % read .avi file
fileinfo = aviinfo(filein); % get .avi file info

%aviobj = M;                % optional: to read in MATLAB movie

% ----- %
% START repeating algorithm on rest of movie, using previous template for
% correlation with current frame
%close all;
for cnt = firstframe:fileinfo.NumFrames, % loop from second frame to end
%for cnt = 1:30

% display frame currently being processed
disp(strcat(['Current frame: ' num2str(cnt)]));

% read frame and do pre-processing
t = aviobj(cnt).cdata;      % read frame
if strcmp(class(t),'uint16') % check if image is uint16
    t = uint8(imdivide(t,257)); % make image uint8
end
s = t;                      % save original image frame
if cnt == firstframe,
    [fcrop,RECT] = imcrop(s); % define target to track interactively
    sf = size(fcrop);

```

```

    close;
end

t = t(:,:,activecolor);           % t1 = image for tracking algorithm
%t = rgb2gray(t);                 % convert RGB to grayscale (if we track gray image)
tcomp = imcomplement(t);          % necessary, depending on if image is
                                  % lighter or darker than background

histq = histeq(tcomp);            % do histogram equalization
L = medfilt2(histq,[3 3]);         % filter image
%L = tcomp;                       % bypass filter/histeq
%figure; imshow(L); title('Filtered Image'); % show filtered image

% do edge detection
%bwlarge = edge(L(:,:,activecolor),'canny',thresh);
% [bw,thresh] = edge(L,'canny',thresh,sigma);

% do image thresholding
% determine threshold
if cnt == firstframe,             % or, define a array of frames where we
                                  % calculate threshold
    %sel = imcrop(L,RECT);
    %level = graythresh(sel);      % define new gray2bw threshold on selection
    level = graythresh(L);        % define new gray2bw threshold on entire frame
    level = 0.95;                 % manually define threshold
end

% apply thresholding
bwlarge = im2bw(L,level);
%figure; imshow(bwlarge); title('BW Image');

% ----- %
if cnt ~= firstframe,

% DO CORRELATION
% show image we correlate with
%figure; imshow(template); title('Corr template');

% do cross correlation of template (from prev image) and current frame
%xcorrel = normxcorr2(template,bwlarge);

% do cross correlation of template (from prev image) and current frame
xcorrel = normxcorr2(template,L);

% save correlation template for display later
templatesave = template; sts = size(templatesave); sx =

```

```

size(xcorrel);

% throwaway non-sensical edges of correlation image
xcorrel([1:floor(st(1)/2) sx(1)-floor(st(1)/2)+1],:) = 0;
xcorrel(:, [1:floor(st(2)/2) sx(2)-floor(st(2)/2)+1],:) = 0;

%figure; mesh(xcorrel);
%figure; imshow(xcorrel);

% update center array
center = [center ; findcenter(xcorrel) - ceil(size(template)/2)];

% define new area on image we are interested in
margin = 5;           % how much bigger than template area we select everytime
temp1 = center(cnt-firstframe,:)-floor(st./2)-margin;
temp2 = st+2*margin;           % rearrange arrays
RECT = [temp1(2) temp1(1) temp2(2) temp2(1)]; % define new RECT

end

% ----- %
% determine tracking template
bw = imcrop(bwlarge,RECT); % crop new template area
%bwc = imcrop(L,RECT);
bwsave = bw;
%figure; imshow(bw); % show cropped area

% try to discard small objects
%[labeled,numObjects] = bwlabel(bw,4);% Label components
%data = regionprops(labeled,'basic');

%areaarray = [];
%for tel = 1:numObjects,
%  areaarray = [areaarray data(tel).Area];
%end
%largest = find(areaarray == max(areaarray));
%[I,J] = find(labeled == largest);
%bwlabeled = zeros(size(bw));
%for tel = 1:length(I),
%  bwlabeled(I(tel),J(tel)) = 1;
%end
%bw = bwlabeled;

% automatically crop selection to minimum size (throw away black edges)
i = (max(bw,[],2));
j = (max(bw,[],1))';

```



```

i = [min(find(i==1)) max(find(i==1))];
j = [min(find(j==1)) max(find(j==1))];
si = i(2)-i(1)+1;
sj = j(2)-j(1)+1;
template = zeros(si+2,sj+2);           % define size of correlation template
%figure; imshow(template);           % show template

st = size(template);
%st = sf;
% make template
template(2:si+1,2:sj+1) = bw(i(1):i(2),j(1):j(2));
%figure; imshow(template);           % show template

% ----- %

% generate mask
% make mask odd and pad
sm = st;                               % initially make mask and template equal size
for i = 1:2,
    if mod(st(i),2) == 0,
        sm(i) = sm(i)+ 1 + 4;
    else
        sm(i) = sm(i) + 4;
    end
end
mask = im2bw(zeros(sm));               % create mask of zeros of right size
%figure; imshow(mask);                % show blank mask shape

% create mask pattern
mask(1,1:3) = 1;
mask(1,sm(2)-2:sm(2)) = 1;
mask(sm(1),1:3) = 1;
mask(sm(1),sm(2)-2:sm(2)) = 1;
mask(:,1) = 1;
mask(:,sm(2)) = 1;
%figure; imshow(mask);                % show mask pattern

% overlay mask on target
smask = s; if cnt ~= firstframe,
    % overlay mask on original image
    smask(:,:,activecolor) = immask(smask(:,:,activecolor), ...
    ... mask,center(cnt-firstframe,:),255);
    % figure; imshow(smask(:,:,activecolor));
    % title('Tracking');               % show mask overlaid on image
end

```

```

% create image to display realtime
disptemplate = zeros(size(bwlarge));

if cnt ~= firstframe,
    % pad with 0's, for display
    disptemplate(1:sts(1),1:sts(2)) = templatesave;
end

trackcomposite = [t histq;L 255*bwlarge; 255*disptemplate
smask(:,:,activecolor)]; imshow(trackcomposite);

if cnt == firstframe,
    figure; imshow(bwsave);
    figure; imshow(bw);
    figure; imshow(template);
    figure;
end

% create MATLAB movie
composite = [uint8(255*ind2rgb(smash(:,:,1),MAP)) ...
... uint8(255*ind2rgb(smash(:,:,2),MAP)); ...
... uint8(255*ind2rgb(smash(:,:,3),MAP)) s];
%figure; imshow(composite);
Mf(cnt) = im2frame(composite); % if fmask is RGB
Mcomp(cnt) = im2frame(trackcomposite,MAP);
end
% end at last frame

%Mfcrop = Mf(1:fileinfo.NumFrames); % throwaway first frame of MATLAB movie ??
filename = 'trackrun13'; disp(strcat(['Generating .avi file '
filename '.avi']));
movie2avi(Mf,filename,'compression','Cinepak'); % create .avi file

disp(strcat(['Generating .avi file ' strcat(filename,'composite') '.avi']));
movie2avi(Mcomp,strcat(filename,'composite'),'compression','Cinepak');

% ----- %

```