An SOPC Based Image Processing System

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

By

Fan Wu

Department of Electrical Engineering and Electronics

September 2007

Abstract

Recent advances in semiconductor technology have made it possible to integrate an entire system including processors, memory and other system units into a single programmable chip - FPGA, these configurations are called "System-on-a-Programmable-Chip" (SOPC). SOPCs have the advantage that they can be designed quicker than existing technologies and are cheap to produce for low volume (<10,000) applications. Also, SOPCs are of great benefit as they offer compact and flexible system designs due to their reconfigurable nature and high integration of features. One processor intensive application, which is ideal for SOPC technology, is that of image processing where there is a repeated application of operations on the 2D data. This research investigated the use of SOPC technology for image processing by developing a modular system capable of real-time video acquisition, processing and display.

This system is comprised of a CameraLink CMOS camera with a custom designed camera interface card for video acquisition, a VGA mode CRT monitor with a Lancelot VGA card for video display, an industrial SDRAM device for video data buffering, and an Altera Apex 20K FPGA for evaluating the SOPC design. Four custom designed IP components have been developed and integrated with other Altera provided standard IP components to drive all off-chip peripherals and perform the required video functions such as processing the images. These custom designed IPs are the video capture controller, video display controller, video memory controller and Cache. A Nios processor was chosen to perform the actual image processing, and the whole system was developed on the Altera Nios development board. In order to solve the complex on-chip data communication, while not degrading the transferring speed of large-amounts of video data, an effective solution called Simultaneously Multi-Mastering Avalon Streaming Transfer with Peripheral-Controlled Waitrequest was raised. Rather than using the software approach to initialise DMA-like transfers, this solution takes advantage of the FPGA hardware resource to perform bus arbitration and hence increases the system efficiency.

The system produced is an alternative to conventional desktop-based, i.e. a vision-based closed loop process control system for welding, or microprocessor-based vision systems.

Acknowledgement

I would like to express my gratitude to my supervisor Professor Jeremy S. Smith for his guidance and help for my research work. I would like to thank my parents for their support. I would like to thank my university colleagues James Buckle, Mr. Gordon Cook and Andrew Tickle for their help especially in improving my English. I would like to thank Kenjin Wong, Yanwei Shou and Lihua Yang for lending me some experimental equipment. I would like to thank all of my friends both in Liverpool and China for their encouragement. Finally I would like to thank my wife Liang Wen for her understanding and support during my PhD studies.

Sincerely yours,

Fan Wu, 2007

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Contents

ABSTRACT I

ACKNOWLEDGEMENT II

DECLARATION III

CONTENTS IV

LIST OF FIGURES X

LIST OF TABLES XIV

CHAPTER 1. INTRODUCTION 1

1.1 Motivation and objectives 1

1.2 Overview 3

CHAPTER 2. INTRODUCTION TO IMAGE PROCESSING, COMPUTER VISION & OVERVIEW OF AN PC BASED VISION SYSTEM FOR WELDING 5

2.1 Computer vision & image processing 5

2.2 Vision-based closed loop process control system for welding 7

2.2.1 General description 7

2.2.2 Image capture and processing 8

2.2.3 Weld image-processing software development 9

2.2.3.1. Feature correlation design 9

2.2.3.2. Front view image processing program (FVIPP) 12

2.2.3.3. Side view image processing program (SVIPP) 15

CHAPTER 3. AN INTRODUCTION TO SYSTEM-ON-A-PROGRAMMABLE-CHIP (SOPC) TECHNOLOGY 16

3.1 SOPC history 16

3.1.1 SOC design 16

3.1.2 ASIC/SOC versus FPGA/SOPC 17

3.1.2.1. Integration 17

3.1.2.2. Flexibility 18
3.1.2.3. Performance 18
3.1.2.4. Power dissipation 19
3.1.2.5. Design flow 19
3.1.2.6. Time-to-market 19
3.1.2.7. Costs 19
3.1.2.8. Conclusion 20

3.2 SOPC architecture 21

3.3 SOPC design flow 22
3.3.1 Design specification 23
3.3.2 Hardware (HW) / Software (SW) partition 23
3.3.3 Design entry 24
3.3.4 Simulations 25
3.3.5 Synthesis 25
3.3.6 Place & route 25
3.3.7 Download & verify in circuit 25
3.3.8 Software development 26
3.3.9 Software compilation 26

3.4 Real time image processing system based on SOPC 26
3.4.1 On-chip vs. off-chip 27
3.4.2 Hard logic fabric vs soft logic fabric 28
3.4.3 Hardware vs software 29

3.5 Common SOPC Processor overview 30
3.5.1 Soft processor 30
3.5.1.1. Nios 30
3.5.1.2. Microblaze 31
3.5.1.3. OpenRISC 31
3.5.1.4. Leon 31
3.5.2 Hard processor 32
3.5.2.1. Excalibur 32
3.5.2.2. Virtex™-II Pro 32
3.5.2.3. PSoC 33
3.5.2.4. FPSLIC 33

3.6 Introduction to the Nios processor system and development tools 34
3.6.1 Nios system architecture 34
3.6.2 Avalon bus module 35
3.6.2.1. Avalon master/slave peripherals 35

3.6.2.2. Avalon master/slave ports 36
3.6.3 Avalon bus transfers 38
3.6.3.1. Avalon slave transfers 39
3.6.3.2. Avalon master transfers 42
3.6.4 Development tools overview 43
3.6.4.1. Quartus II software 43
3.6.4.2. SOPC Builder 44
3.6.4.3. ModelSim 46
3.6.4.4. Programming tool 46
3.6.4.5. Software download tool 46
3.6.5 Proposed image processing system with the Nios processor system architecture 47

CHAPTER 4. THE NIOS INTEGRATED REAL-TIME IMAGE PROCESSING SYSTEM - HARDWARE 48

4.1 Overview of the system hardware architecture 48

4.2 Video display device & interface 50
4.2.1 CRT monitor 50
4.2.2 Lancelot VGA board 50
4.2.3 VGA connector 51

4.3 Video memory device & interface 51

4.4 Video capture device & interface 52
4.4.1 CameraLink camera 52
4.4.2 Camera interface card (custom designed) 53

4.5 Altera's Apex device 56

4.6 Nios development board 57

4.7 Summary 58

CHAPTER 5. THE NIOS INTEGRATED REAL-TIME IMAGE PROCESSING SYSTEM – SOFT SYSTEM CORE 61

5.1 Overview of the system core architecture 61

5.2 Video memory controller 64
5.2.1 Main features 64
5.2.2 Description of the video memory controller 64
5.2.2.1. Avalon interface 68

5.2.2.2. SDRAM controller 71
5.2.2.3. SDRAM data path 79
5.2.3 Summary 80

5.3 Video display controller 81
5.3.1 Main features 82
5.3.2 Video graphic arrays (VGA) 82
5.3.3 Description of the video display controller 82
5.3.3.1. Avalon interface 85
5.3.3.2. VGA driver 87
5.3.4 Summary 93

5.4 Video capture controller 94
5.4.1 Main features 94
5.4.2 CameraLink 94
5.4.3 Description of the video capture controller 96
5.4.3.1. Avalon interface 98
5.4.3.2. Video receiver 101
5.4.4 Camera serial controller (UART) 103
5.4.5 Summary 104

5.5 Cache 105
5.5.1 Main features 105
5.5.2 Overview of the Cache 105
5.5.3 Description of the Cache 107
5.5.4 Summary 111

5.6 System clock generator 114
5.6.1 Data clock 115
5.6.2 System clock 116
5.6.3 Video display clock 116
5.6.4 Video capture clock 117
5.6.5 Discussions 117

5.7 Explanation of some design issues 118
5.7.1 Multiple-bank operation 118
5.7.1.1. Triple-bank operation 118
5.7.1.2. Quad-bank operation 119
5.7.2 Double line buffer in the video display & capture controller 120
5.7.3 Synchronisation of multiple clock domains 121
5.7.4 Other issues 122

CHAPTER 6. SIMULTANEOUS MULTI-MASTERING AVALON STREAMING TRANSFER WITH PERIPHERAL-CONTROLLED WAITREQUEST 123

6.1 Synchronisation of DMA controllers solution 123

6.2 Avalon bus arbitration 126

6.3 Implementation of simultaneous multi-mastering streaming Avalon transfer with peripheral-controlled waitrequest 130

6.4 Another solution 135

CHAPTER 7. SYSTEM CORE GENERATION, SYNTHESIS & IMPLEMENTATION 136

7.1 System core generation 136
7.1.1 PTF files 136
7.1.2 System generation 137

7.2 System core synthesis 140
7.2.1 System core synthesis 141
7.2.2 Synthesis results & discussions 143

CHAPTER 8. SYSTEM TESTS, IMAGE PROCESSING ALGORITHM IMPLEMENTATION & PERFORMANCE ANALYSIS 146

8.1 System tests 146
8.1.1 Simulations 146
8.1.1.1. Video memory controller simulation results & discussions 147
8.1.1.2. Video display controller simulation results and discussions 153
8.1.1.3. Cache simulation results and discussions 153
8.1.1.4. Video capture controller simulation results and discussions 154
8.1.1.5. Full system simulations results and discussions 162
8.1.2 Hardware verifications 164
8.1.2.1. Video memory test 164
8.1.2.2. Video display test 167
8.1.2.3. Cache test 170
8.1.2.4. SMMAST-PCW test 173
8.1.2.5. Video capture test 174
8.1.2.6. Full system test 176

8.2 Implementation of various image processing algorithms and real-time performance analysis 179
8.2.1 General software development 179
8.2.1.1. Setting up video display & capture 179

8.2.1.2. Implementing multiple bank operation & interrupt services 179
8.2.1.3. Memory read/write operations 181
8.2.1.4. Timer function 181
8.2.2 Discussion of software coding style – optimisation issue 181
8.2.3 System performance analysis 184
8.2.4 Implementation of five image processing algorithms and performance analysis 184
8.2.4.1. Inversion 184
8.2.4.2. Sobel edge detector 186
8.2.4.3. Gaussian blur filter (a low pass filter) 190
8.2.4.4. Sharpness filter – (a high pass filter) 193
8.2.4.5. Feature correlation 195

8.3 Summary 197

CHAPTER 9. CONCLUSIONS AND FUTURE WORK 198

9.1 Conclusions 198

9.2 Future work 200
9.2.1 Optimisation/improvements 200
9.2.1.1. Later generation FPGA 201
9.2.1.2. FPGA Hardware processing 201
9.2.2 Application 201

REFERENCES 203

APPENDIXES 212

Appendix A Schematics 212

Appendix B Camera interface card PCB details 215

Appendix C Pin assignments for video components 216

Appendix D Truth table for the operation commands of SDRAM 219

Appendix E Register maps 220

Appendix F PTF Files – An example of Cache 224

List of Figures

Figure 2-1 Common steps in real-time image processing system 6
Figure 2-2 Closed loop weld process control (From [3]) 7
Figure 2-3 Real time image capture and analysis (From [3]) 8
Figure 2-4 Image edge feature correlation (From [3]) 10
Figure 2-5 Front view measured Items 12
Figure 2-6 Front view image processing program 13
Figure 2-7 Configuration mode 14
Figure 2-8 Calibration mode 14
Figure 2-9 Side view image processing program 15
Figure 3-1 Apex 20K device block diagram (From [27]) 18
Figure 3-2 NRE cost of developing ASICs 20
Figure 3-3 Overall SOPC design flow 23
Figure 3-4 System module integrated with user logic into an Altera PLD (From [35]) 34
Figure 3-5 Avalon bus module block diagram – an example system (From [35]) 35
Figure 3-6 Fundamental slave read transfer (From [35]) 39
Figure 3-7 Fundamental slave write transfer (From [35]) 40
Figure 3-8 Slave read transfer with peripheral-controlled waitrequest (From [35]) 40
Figure 3-9 Slave write transfer with peripheral-controlled waitrequest (From [35]) 41
Figure 3-10 Streaming slave read transfer (From [35]) 42
Figure 3-11 Streaming slave write transfer (From [35]) 42
Figure 3-12 Fundamental master read transfer (From [35]) 43
Figure 3-13 Fundamental master write transfer (From [35]) 43
Figure 3-14 Streaming master R/W transfer (From [35]) 43
Figure 3-15 SOPC Builder system contents page (From [55]) 45
Figure 3-16 SOPC Builder (From [42]) 46
Figure 4-1 Block diagram of the hardware architecture of SIPS 48
Figure 4-2 The Nios integrated real-time image processing system 49
Figure 4-3 Lancelot VGA board 50
Figure 4-4 VGA connector 51
Figure 4-5 Toshiba SDRAM SODIMM THLY 6480H1FG-80 51
Figure 4-6 COHU 7800 series 1280x1024 CMOS progressive scan camera 53
Figure 4-7 Camera interface card 53

Figure 4-8 Block diagram of the camera interface card 54
Figure 4-9 MDR connector (from [63]) 55
Figure 4-10 Nios development board (From: [58]) 58
Figure 5-1 Top level block diagram of SIPS core 63
Figure 5-2 Block diagram of the video memory controller 65
Figure 5-3 ASM chart of the Avalon streaming slave transfer 70
Figure 5-4 Block diagram of the SDRAM controller 71
Figure 5-5 Mode Register Set cycle (From [62]) 73
Figure 5-6 Page Mode Read/Write (Burst Length = 8, CAS Latency = 3) (From [62]) 76
Figure 5-7 Timing chart for Burst Stop cycle (From [62]) 77
Figure 5-8 Auto Refresh cycle (From [62]) 78
Figure 5-9 Block diagram of the video display controller 83
Figure 5-10 ASM chart of the Avalon streaming master read transfer 88
Figure 5-11 Block diagram of the VGA driver 89
Figure 5-12 640×480 VGA horizontal timing (From [73]) 90
Figure 5-13 640×480 VGA vertical timing (From [73]) 91
Figure 5-14 Channel Link operation (From [60]) 95
Figure 5-15 Block diagram of the video capture controller 96
Figure 5-16 ASM chart of the Avalon streaming master write transfer 100
Figure 5-17 Block diagram of the video receiver 102
Figure 5-18 Camera timing (From [63]) 103
Figure 5-19 Cache mapping to the main memory 106
Figure 5-20 Cache structure (From [83]) 107
Figure 5-21 Block diagram of the Cache 108
Figure 5-22 Schematic drawing of the Cache structure 113
Figure 5-23 Cache read handle ASM chart 114
Figure 5-24 Dedicated global clock pin connections to PLL & dedicated clock lines for EP20K30E, EP20K60E, EP20K100E, EP20K160E & EP20K200E devices (from [84]) 115
Figure 5-25 Clock circuitry (From [58]) 117
Figure 5-26 Triple-bank operation 119
Figure 5-27 Quad-bank operation 119
Figure 5-28 Metastable output propagating invalid data throughout the design (from [86]) 121

Figure 5-29 Two flip-flop synchroniser (from [86]) ... 122
Figure 6-1 Triple-ported video memory slave ... 123
Figure 6-2 Nios DMA peripheral with master & slave ports (From [88]) ... 124
Figure 6-3 I/O to memory DMA controller – Excalibur (From [73]) ... 124
Figure 6-4 Interface synchronisation (From [74]) ... 125
Figure 6-5 Centralised, parallel arbitration bus architecture ... 127
Figure 6-6 Simultaneous multi-mastering Avalon bus arbitration ... 128
Figure 6-7 Successive fundamental read transfers to a common slave (From [90]) ... 128
Figure 6-8 An arbitration view during conflict between two streaming masters ... 130
Figure 6-9 Master peripheral and slave peripheral with streaming control feature ... 131
Figure 6-10 Block diagram of simultaneous multi-mastering image processing system ... 132
Figure 6-11 ASM chart of streaming control master ... 132
Figure 6-12 An example timing diagram of simultaneous multi-mastering Avalon streaming transfer with peripheral-controlled waitrequest ... 133
Figure 6-13 Multi-ported memory slave solution ... 135
Figure 7-1 Custom defined library components ... 138
Figure 7-2 Top-level system view in SOPC Builder ... 139
Figure 7-3 System generation results ... 140
Figure 7-4 Top system module view ... 141
Figure 7-5 Block diagram of the SIPS core ... 142
Figure 8-1 SIPS simulation scheme ... 147
Figure 8-2 Video memory controller simulation result – burst write (BL = 1) ... 149
Figure 8-3 Video memory controller simulation result – burst write (BL = 80) ... 150
Figure 8-4 Video memory controller simulation result – burst read (BL =1) ... 151
Figure 8-5 Video memory controller simulation result – burst read (BL = 80) ... 152
Figure 8-6 Video display controller simulation result 1 ... 155
Figure 8-7 Video display controller simulation result 2 ... 156
Figure 8-8 Cache simulation result – Cache write ... 157
Figure 8-9 Cache simulation result – Cache miss (BL=1) ... 158
Figure 8-10 Cache simulation result – Cache hit ... 159
Figure 8-11 video capture controller simulation result 1 ... 160
Figure 8-12 video capture controller simulation result 2 ... 161
Figure 8-13 Full system simulation result ... 163

Figure 8-14 Video memory test scheme 165
Figure 8-15 Video display test scheme 167
Figure 8-16 Cache test scheme 171
Figure 8-17 Cache hit/miss operation waveform in SignalTap II 172
Figure 8-18 SMMAST-PCW test scheme 173
Figure 8-19 simultaneously multi-mastering operation waveform in SignalTap II 174
Figure 8-20 Video capture test scheme 174
Figure 8-21 system view of the video capture test without a camera 175
Figure 8-22 video capture test result 176
Figure 8-23 Test image – camera interface card (original) 177
Figure 8-24 Test image – mugs and jug (original) 177
Figure 8-25 An example of the testing background 178
Figure 8-26 Inversion image – camera interface card 185
Figure 8-27 Inversion image – mugs and jug 185
Figure 8-28 Sobel edge detector image – camera interface card (threshold=60) 187
Figure 8-29 Sobel edge detector image – mugs and jug (threshold=60) 187
Figure 8-30 Sobel convolution kernels 187
Figure 8-31 Gaussian blur filter image – camera interface card (mask size 7x7) 190
Figure 8-32 Gaussian blur filter image – mugs and jug (mask size 7x7) 190
Figure 8-33 Gaussian filters coefficients (from [104]) 191
Figure 8-34 Sharpness filter image –camera interface card 193
Figure 8-35 Sharpness filter image –camera interface card 193
Figure 8-36 Feature correlation image (measured on pool edges & wire edges) 195

List of Tables

Table 2-1 Measurements of the front view image processing program 12
Table 3-1 World SOC Market, 2003, 2004, 2009 and AAGR ('04-'09) 17
Table 3-2 Avalon slave port signals 37
Table 3-3 Avalon master port signals 38
Table 4-1 Device table 49
Table 4-2 VGA connector pin assignment 51
Table 4-3 COHU 7800 series camera specifications 52
Table 4-4 COHU 7800 CameraLink cable pin assignments 55
Table 4-5 Apex 20K200E device features (From [27]) 57
Table 5-1 Video memory controller top level signals 66
Table 5-2 Internal signals of the video memory controller 68
Table 5-3 SDRAM address mapping 69
Table 5-4 Mode register set command 72
Table 5-5 Streaming read command 74
Table 5-6 Streaming write command 75
Table 5-7 Auto refresh command 77
Table 5-8 Description of the word mask 80
Table 5-9 Video display controller top level signals 83
Table 5-10 Internal signals of the video display controller 85
Table 5-11 Pixel assignment in 8-bit monochrome mode for video display 86
Table 5-12 Pixel assignment in 24-bit RGB mode for video display 86
Table 5-13 640×480 VGA horizontal timing 90
Table 5-14 640×480 VGA vertical timing 91
Table 5-15 Alternating operations of the double line buffer 92
Table 5-16 CameraLink signals 95
Table 5-17 Video capture controller top level signals 96
Table 5-18 Internal signals of the video capture controller 98
Table 5-19 Pixel assignment in 8-bit monochrome mode for video capture 99
Table 5-20 Pixel assignment in 24-bit RGB mode for video capture 99
Table 5-21 Cache top level signals 108
Table 7-1 Synthesis result in 8-bit mode 143
Table 7-2 Estimated performance on various FPGAs 145

Table 8-1 Full simulation data input sets 162
Table 8-2 Estimated full simulation data output sets 164
Table 8-3 Video memory test scheme 165
Table 8-4 Walking 1's test 166
Table 8-5 Increment test 167
Table 8-6 Video display test scheme 168
Table 8-7 Video display margin test in 24-bit mode 169
Table 8-8 Video display margin test in 8-bit mode 169
Table 8-9 Cache test scheme 171
Table 8-10 Video capture test scheme 175
Table 8-11 Inversion processing power analysis – without optimisation 186
Table 8-12 Inversion processing power analysis – with optimisation 186
Table 8-13 Pixel alignments in 8-bit grey level mode 188
Table 8-14 Sobel edge detector processing power analysis 189
Table 8-15 Gaussian blur filter processing power analysis 192
Table 8-16 Sharpness filter convolution kernel 194
Table 8-17 Sharpness filter processing power analysis 194
Table 8-18 Feature correlation processing power analysis 197
Table 9-1 SIPS performance summary 199

Chapter 1. Introduction

1.1 Motivation and objectives

Real time vision systems have been widely used in security, quality control and automatic handling etc. Conventional vision systems can be classified into two different types of architecture [1].

One is a board type device for a host computer (i.e. personal computer PC). Every function module can be implemented as a separate board with a computer expansion interface, for example a video acquisition card and graphic card. J.Kang and R. Doraiswami developed such a system with an add-on universal serial bus (USB) interface board to allow the PC to capture video from an external web-cam for endoscopic applications [2]. C. Balford, J. S. Smith and S. Amin-Nejad present a vision-based closed loop control system for weld application where a PC associated with a video capture card and a graphic card was used to perform real-time video acquisition, image analysis and display [3]. Although some systems use hardware accelerated approaches, for example, by incorporating an array of processing elements (or computing elements) built from Application Specific Integrated Circuits (ASICs) or Field Programmable Gate Arrays (FPGAs) into a video acceleration board with the ability of providing parallel processing to handle some or all of the complex image processing tasks, such as systems using a Splash 2 board [4], other applications for discrete-time cellular neural network (DTCNN) use a highly parallel integrated circuits (HiPIC) add-on board [5]. Generally most computer based vision systems use the software approach – by using the generic Central Processing Unit (CPU) to perform all image processing tasks for cost reasons. With the rapid increase of the speed and functionality of the generic processors, like Intel processors, the software approach now offers more processing power to handle more complex image processing tasks in real-time. These PC based vision systems can offer distinct performance and expandable functionality, however, they are not ideal for compact vision systems because a host computer is required to connect and control all peripherals.

Another type of vision systems are embedded systems which consist of one or more microprocessors to control the whole system and perform image processing tasks.

These microprocessors could be general purpose microprocessors (e.g. Motorola 68030) or digital signal processing (DSP) microprocessors (e.g. TI 320 series) [6]. This type of architecture allows the whole system to be built into a single standalone device. For example, a vision-based target tracking system was developed for an embedded application of surveillance with a drone by using two-processor including a microprocessor for camera control and a DSP processor for implementing the block matching algorithm [7]. Another example is a mobile mini robot with embedded CMOS vision system designed for an indoor soccer game scenario, image processing algorithms including feature extraction and object classification are performed by a PIC microcontroller [8]. However, the significant increase of hardware complexity results in a much less significant speed-up for microprocessors. Therefore in the future, further development of the complex superscalar processors is unlikely [9]. Besides the performance issue, the lack of flexibility and optimisation, and complexities of firmware coding on some particular microprocessors [10] are other issues that system developers may encounter. In fact the obsolescence of microprocessors has been a major concern for many companies with regards of the difficulties of further development and upgrades [11].

Recent advances in semiconductor technology have made it possible to integrate the entire embedded/computer system including processors, memory and other system units into a single programmable chip - FPGA, and this technology is called "System-on-a-Programmable-Chip" (SOPC) [12]. This solution therefore offers an alternative architecture to implement the standalone vision system. In fact, as the rapid increase of FPGA density (about 10 times in two years [9]) has already made it expand quicker than microprocessors, taking into account the silicon area and throughput, the FPGAs significantly outperform microprocessor [13], and in the future the performance gap will further increase.

Due to the reconfigurable ability and compact nature, SOPC offers high performance and flexibility with low risks. SOPC designs can be very easy migrated into different kinds of FPGAs to provide various performances without worrying problems such as the changes of the system architecture, because all components can be implemented as a separate non-device specific soft Intellectual Properties (IP) core including the processors. This is a feature that conventional microprocessor systems can't easily

fulfill. By using the soft processor core such as Nios processor from Altera and Microblaze from Xilinx [14], the system architecture is configurable allowing a trade-off between performance and area by changing the architecture [15]. Designs can be customised and further optimised to suit different system platforms by changing Cache size and type, optimising processor instructions and so on. Furthermore, the SOPCs have other advantages like low cost and short development time [16].

One processor intensive application which is ideal for SOPC technology is that of image processing where there is a repeated application of operations on the two-dimensional (2D) data. Therefore, the objectives of this research were to investigate the use of SOPC technology in building a real time image processing system with the capability of performing video acquisition, display and processing. To achieve this requires solutions to problems that would occur under this new system architecture like the multi-mastering burst transfers. This system is named "an SOPC based image processing system" (SIPS) as it aims at implementing general image processing, whilst further development into a complete computer vision system, for various applications, can be achieved by integrating more system controllers.

This research was motivated from an existing PC based vision system for welding application [3]. One of the improvements for this system was to provide a complete computer vision-based sensor capable of delivering the required measurements on demand without the use of a host computer, so that it allows easier integration of the vision sensing system with commercial welding process controllers.

1.2 Overview

This thesis is organised as nine chapters and a brief introduction for each chapter is now given as follows:

Chapter Two firstly gives some explanations of fundamental concepts regarding image processing and computer vision. Then an overview of the conventional PC based vision system used for welding applications is presented. The software implementation detail of the specific image processing algorithm called 'feature correlation' is presented.

Chapter Three gives a general introduction to the SOPC technology in terms of its development history, architecture and design flow. Concerns are given specifically for applying SOPC technology in constructing an image processing system. Following that, a brief overview of commonly used processors in SOPC design is given. Finally this chapter gives the background information of the system architecture, bus protocol and development tools that SIPS used.

Chapter Four starts to describe SIPS from the hardware side. All hardware components including the custom fabricated units are introduced, and the reasons for their choices are also presented.

Chapter Five continues to describe SIPS but mainly focus on explaining the SOPC design. All video IP architectures are described in detail; some typical design issues regarding the implementation details are also discussed.

Chapter Six explains a novel solution to a multi-mastering problem. A classic approach is given before explaining the solution used in SIPS. Finally an alternative method is also introduced.

Chapter Seven describes how the whole system is generated, synthesised and implemented on the actual hardware. Generation and synthesis results are given as well as discussions.

Chapter Eight describes system test details and discusses the results. The system tests include system level simulation and hardware verification. This chapter also presents the software implementation details. Several image processing algorithms are given as examples and a performance analysis is presented for each of them to help better understanding the performance of SIPS.

Chapter Nine presents a conclusion to this system and indicates the potential areas for future work.

Chapter 2. Introduction to Image Processing, Computer Vision & Overview of an PC Based Vision System for Welding

This chapter consists of two sections. The first section explains some fundamental disciplines involved in developing the SIPS. The second section presents a PC based vision system for a welding application. The feature correlation algorithm applied in processing the weld images is described.

2.1 Computer vision & image processing

Vision allows humans to perceive and understand the world surrounding them. With the development of modern electronics, the effect of human vision can be imitated by electronically perceiving images and understanding their contents with modern computers and this, as a science, is called computer vision. It is concerned with extracting information about a scene by analysing images of that scene [17]. This scene could be two-dimensional or three-dimensional (3D) depending on the practical applications. As a technological discipline, computer vision seeks to apply the theories and models of computer vision to the construction of computer vision systems. Examples of applications of computer vision systems include systems for controlling processes (e.g. an industrial robot or an autonomous vehicle), detecting events (e.g. for visual surveillance), organising information (e.g. for indexing databases of images and image sequences), modeling objects or environments (e.g. industrial inspection, medical image analysis or topographical modeling) and interaction (e.g. as the input to a device for computer-human interaction) etc.

Computer vision is now recognised as an interdisciplinary research field that is chiefly dependant on image processing but also spans processor design, graphical and communication techniques, control, information handling and computer aided design processes. The typical tasks of computer vision systems are recognition, motion detection, scene reconstruction and image restoration etc. It is very often the case that the general goal of a computer vision system is to recognise objects of various types that may be presented in the scene.

Machine vision (MV) is the application of computer vision to industry and

manufacturing. Whereas computer vision is mainly focused on machine-based image processing, machine vision often requires digital input/output devices and computer networks to control other manufacturing equipment such as robotic arms. Machine vision is a subfield of engineering that encompasses computer science, optics, mechanical engineering, and industrial automation.

Image processing is the collation of spatially arranged intensity data, forming an image, which is processed to extract information about the scene [18]. The input of image processing is an image such as a frame of video, while the output could be an image or a set of features of the image. Image processing is rather independent of an application domain. However, it plays an essential role in computer vision systems such as a vision based robot control system because a robust image processing algorithm is required. Because of this, a real-time image processing system is often required to be set up to verify specific image processing algorithms and estimate its performance in a real-time manner before being applied into a complete (computer) vision system to perform particular vision task. The real time goal is to process all the required data in a given time interval before the next image is ready for processing. To estimate the performance of a real-time image processing system it is required to analyse how much data it can handle in real time. Such a system generally provides three main functions which are video acquisition, processing and output (see Figure 2-1), and the video data transfer in this system is one way. The video data can be acquired by an analogue/digital camera or a video recording device. As described in Chapter 1, the image processing can be performed by using general processors such as CPU, general purpose microprocessor, DSP processor, synthesised processor running on a FPGA, or processing elements built into FPGAs or ASICs. The video output generally refers to video display on monitors which gives a visual indication of the image processing results. Furthermore, a data storage function is normally required to buffer the data output from each level before being sent to the next due to the existence of speed differences between each function module.

Figure 2-1 Common steps in real-time image processing system

The next section gives an example of how image processing is integrated into a vision-based system for automated welding.

2.2 Vision-based closed loop process control system for welding

The vision-based closed loop process control system was designed to improve the weld quality. In this system, image analysis and processing are applied to the images captured by the camera for supplying real-time measurements on the weld, in order to provide additional information to the process operator [3]. The description of this system, in the next several sections, focuses on the image acquisition and processing parts as the author has been involved in developing the image processing software, and more importantly its application to the research described in this thesis. More details regarding the post processing and how the whole system functions can be obtained in [3].

2.2.1 General description

Figure 2-2 illustrates a simplified diagram of the closed loop weld process control system.

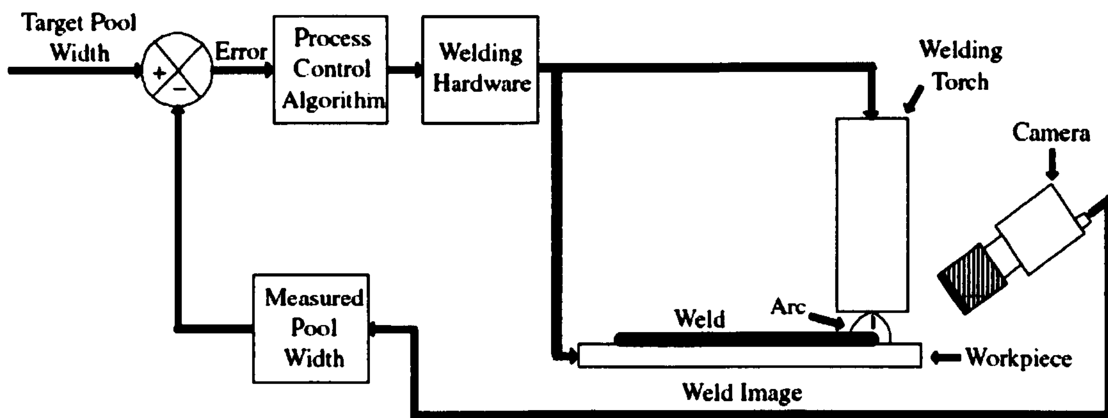


Figure 2-2 Closed loop weld process control (From [3])

The real-time molten pool width is one of the measurements extracted by this vision based system. As seen from the figure, this system aims to extract the real-time measured pool width from analysing the live welding images captured from the camera and compares the measured result with the target pool width supplied into the image processing software. If a difference exists then an error is generated, the process control algorithm generates adjustments based on this error to modify the behaviour of the welding process such as changing the current via the welding hardware interface. This

process is then repeated when the next measurement is available from the image-processing software thereby controlling or regulating the welding characteristics.

In this vision-based system, the image processing has been specified for welding applications and integrated with the welding control software. Apart from the process control, one of the key features of this system is to process the weld images captured from the camera in real time. This process is undertaken by running image processing software programs in a standard PC.

2.2.2 Image capture and processing

The basic layout of the weld image capture and analysis system is shown in Figure 2-3.

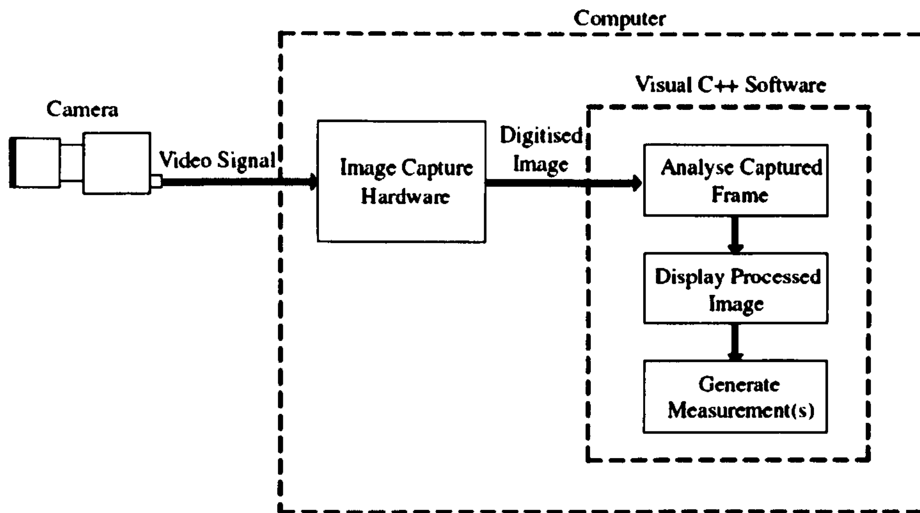


Figure 2-3 Real time image capture and analysis (From [3])

The analogue composite video signal is captured from two commercial CCD cameras and then fed into a Pentium class PC running Microsoft Windows 2000 for analysing and processing the weld images via the image capture hardware interfaces. These interfaces include a Picolo [19] capture board for the front view image processing program (FVIPP) [20] and a 'WinTV' [21] card for the side view image processing program (SVIPP). The FVIPP and SVIPP were written in Microsoft Visual C++. They supply separate measurements and are used together to provide a comprehensive analysis on the weld images from different views. The FVIPP used the Picolo device driver to access the digitised images or video from the Picolo board while the SVIPP used the standard 'Video for Windows' (VFW) interface [22]. In order to provide a visual indication of the operation of the image processing software, the modified video with the key features highlighted are displayed on a standard PC monitor along with

the relevant measured results. Following this, the final phase of the image analysis process is to provide the relevant image measurements to the welding control system software VEE, which has been designed to run concurrently with the image processing programs in the same computer. They run as different threads and communicate with each other through pipes [23]. Such a measurement will allow a controller, as and when required, to modify the welding process parameters in order to regulate or maintain a desired set of weld characteristics.

The image processing system is capable of working in real-time at the frame rate of the video capture device, provided that the associated processing algorithm can be executed during the time between the capture of successive pictures or 'frames'. This time is typically 40ms for the standard 25Hz interlaced CCD camera system that has been used.

A Control Area Network (CAN) [24] interface has been developed and integrated into the VEE software. This allows alteration of the welding process parameters at speeds of up to 25Hz via a distributed network of embedded controllers.

2.2.3 Weld image-processing software development

The general requirement of the image processing algorithm is to identify target edges such as the left and right edges of the molten weld pool or the weld wire, so that a corresponding width measurement can be achieved and made use in real-time with the feedback control system. A feature correlation algorithm was developed to implement these edge detections.

2.2.3.1. Feature correlation design

The basic idea of this algorithm is firstly to extract some sample image features such as the weld pool edge features (L_{fa} and R_{fa} in Figure 2-4) in the calibration stage.

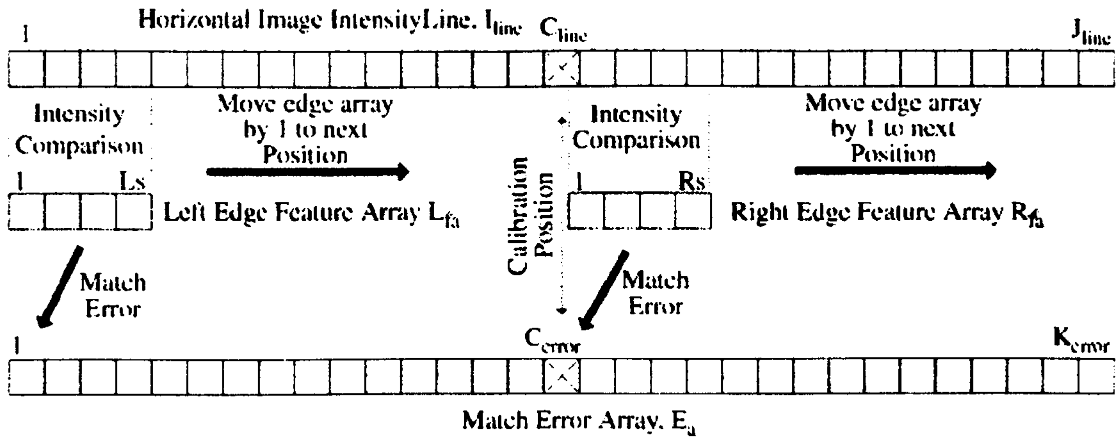


Figure 2-4 Image edge feature correlation (From [3])

Then copy a horizontal (I_{line}) or vertical line at the specified calibration position from a live captured frame, use these typical edge feature intensities to compare with specific positions at that line along the array of weld image pixel intensities, and then use the comparison results to generate a match error array (E_a). The equation of calculating the match error array for the left edge feature is given in Equation 2-1.

$$E_a[i] = \sum_{x=1}^{Ls} \left| L_{fa}[x] - I_{line}[x + (i-1)] \right| \text{ for } 1 \le i \le (C_{line} - L_s) \quad \text{(Equation 2-1)}$$

Where E_a is the error array at position or index i, C_{line} is the captured line calibration position and L_s is the left feature array size. The right feature error array can be calculated by a similar manner.

Once the error array is calculated, by searching where the minimum error is in the error array the position of highest association with the edge features can be obtained as shown in Equation 2-2.

$$L_{match_min} \longleftarrow \min\{E_a[i] | 1 \le i \le (C_{error} - L_s)\} \quad \text{(Equation 2-2)}$$

Where L_{match_min} is the minimum left feature correlation error.

The position of the minimum left feature match error i_{min_l} is then recorded as shown in Equation 2-3.

$$L_{match_min} = E_a[i_{min_l}] \quad \text{(Equation 2-3)}$$

The location of the best match for the centre of the left edge feature match L_{match_centre} is given in Equation 2-4

$$L_{match_centre} = E_a[i_{min_l} + (L_s/2)] \quad \text{(Equation 2-4)}$$

Following the same method the best match for the centre of the right edge feature R_{match_centre} can be located by using Equation 2-5.

$$R_{match_centre} = E_a[i_{\min_r} + (R_s/2)] \quad \text{(Equation 2-5)}$$

Finally the weld pool width PW_{pix} in pixels can be achieved by using Equation 2-6.

$$PW_{pix} = i_{\min_r} - i_{\min_l} \quad \text{(Equation 2-6)}$$

This value will then be translated into a measurement in mm for use with the weld software by taking into account the resolution of the image capture system and the field of view of the weld imaging optics.

This algorithm was originally developed by C. Balfour, J. S. Smith and S. Amin-Nejad. The calculations listed above are taken from [3]. By applying this algorithm into different welding situations, a sequence of measurements and the welding status can be obtained. The next two sections describe the two programs with the feature correlation algorithm applied to achieve the measurements.

2.2.3.2. Front view image processing program (FVIPP)

The main objectives of the front view image processing algorithm is to drive the Picolo board to capture live video into PC memory, and use the feature correlation algorithm to identify the left and right molten weld pool edge, weld wire edges and wire end edges, so that a measurement of the weld pool width, wire width and wire end width can be made in real-time for use with the feedback control system (see Figure 2-5). Furthermore, determination of the welding status such as blobbing or stubbing is also one of the requirements. The final task this program required to perform is to create a pipe so that all of these measurements can be transmitted to the weld control software and commands from the control software can be sent to the FVIPP like starting and stopping the image processing program.

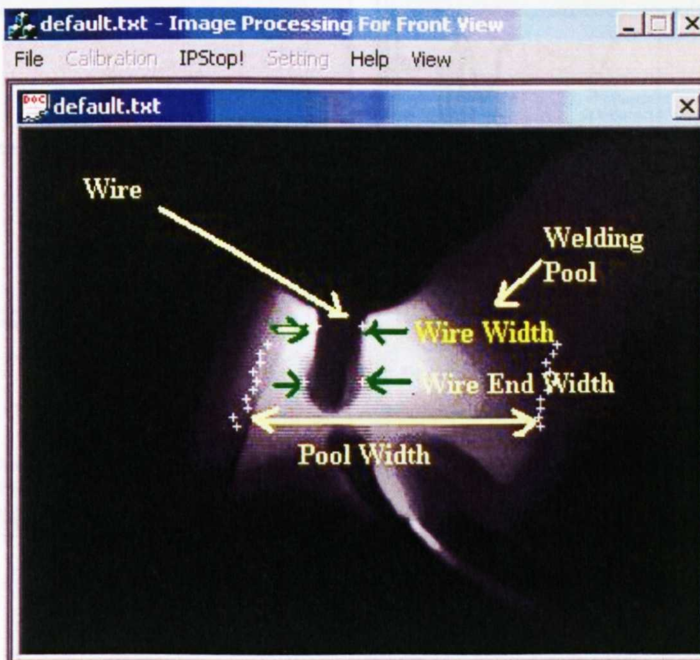


Figure 2-5 Front view measured Items

Table 2-1 lists the measurements that the FVIPP calculates.

Table 2-1 Measurements of the front view image processing program

Measured item	Description
Pool Width	The measured width of the welding pool (in pixels)
Wire Width	The measured width of the wire (in pixels)
Wire End Width	The measured width of the end of the wire (in pixels)
Blobbing Flag	Blobbing flag, indicates whether the end of the wire is blobbing
Stubbing Flag	Stubbing flag, indicates whether the wire is stubbing

Figure 2-6 shows two running views of the FVIPP with the measured results displayed. Blobbing and stubbing are recorded in these two figures.

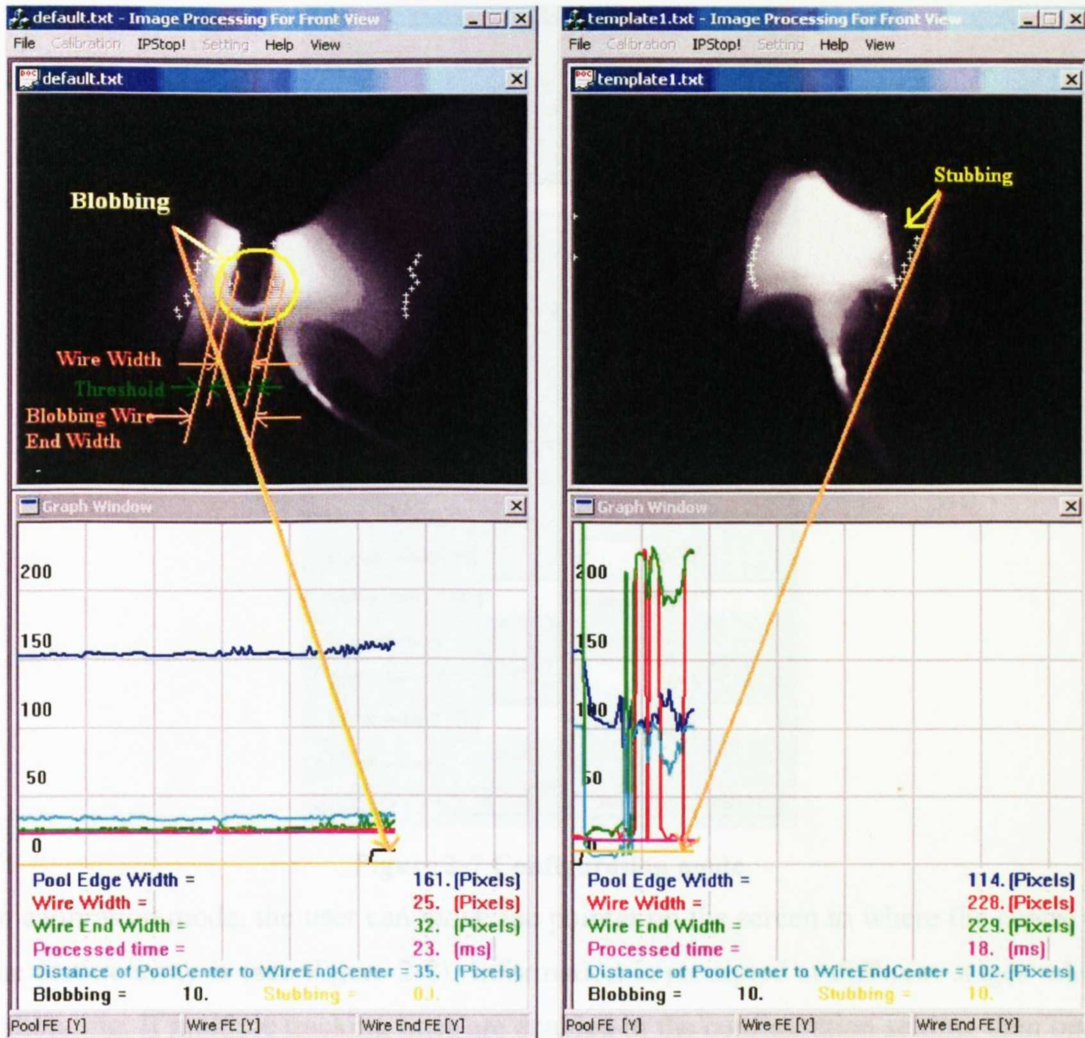


Figure 2-6 Front view image processing program

The rules used to define blobbing and stubbing are explained as follows.

Blobbing rule: within a specified time period (m readings), if there are more than n samples (n readings) whilst the difference between the wire end width and wire width is greater than the pre-defined threshold, then it is a blobbing. The left picture of Figure 2-6 shows an occurrence of blobbing.

Stubbing rule: within a specified time period (m readings), if there are more than n samples (n readings) whilst the amplitude of the wire center is greater than the pre-defined threshold, then it is a stubbing. The right picture of Figure 2-6 shows an occurrence of stubbing.

All settings for determining the blobbing and stubbing as well as the other parameters for use with this program can be specified in the configuration stage (see Figure 2-7).

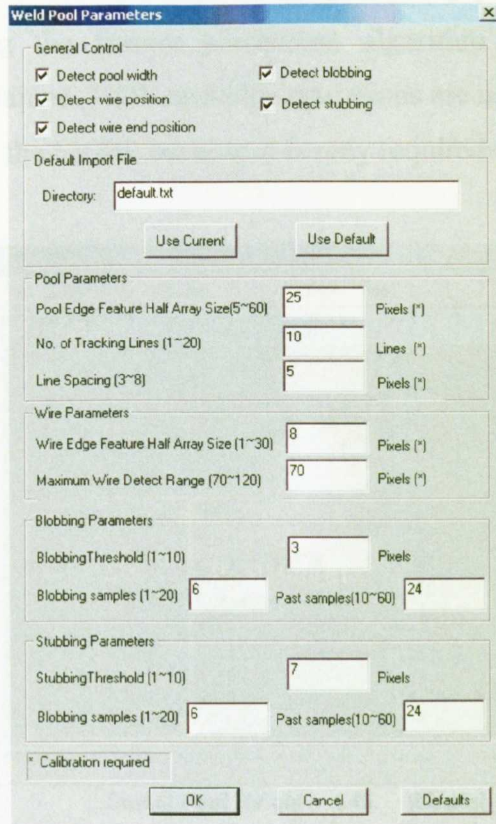


Figure 2-7 Configuration mode

In calibration mode, the user can move the pointer on the screen to where the centre of the feature array is (see Figure 2-8). Information is prompted at different stages while calibrating. If multiple tracking lines are enabled in the configuration setting, then once a feature array is positioned, the other arrays for the same feature can be automatically located on the other lines in parallel by using the Sobel operator.

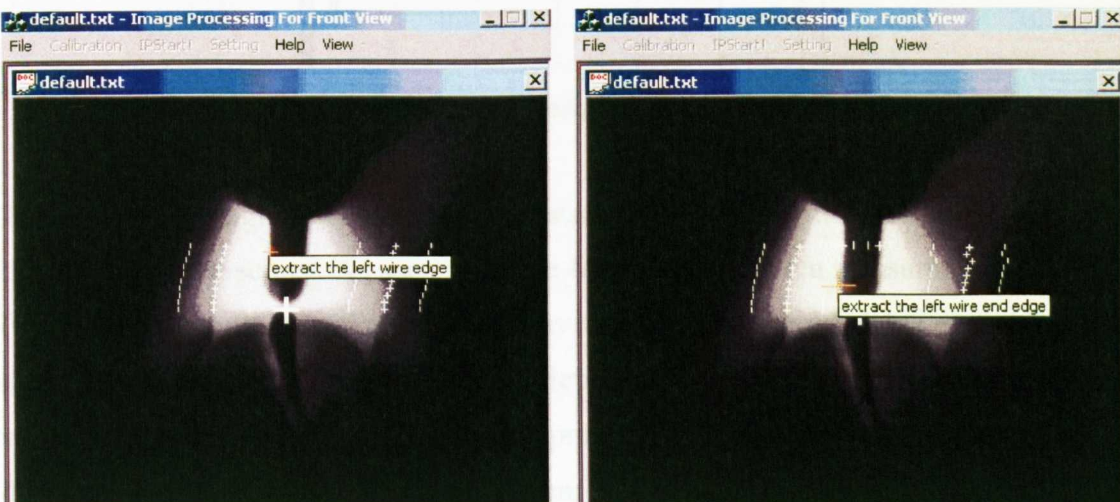


Figure 2-8 Calibration mode

2.2.3.3. Side view image processing program (SVIPP)

The main requirement for the side view image processing program is to identify the pool edge by applying the feature correlation algorithm so that a pool height measurement can be obtained. VFW and pipe operations are used in this program. This program is simpler than the FVIPP because it is only required to identify one pool edge (see Figure 2-9).

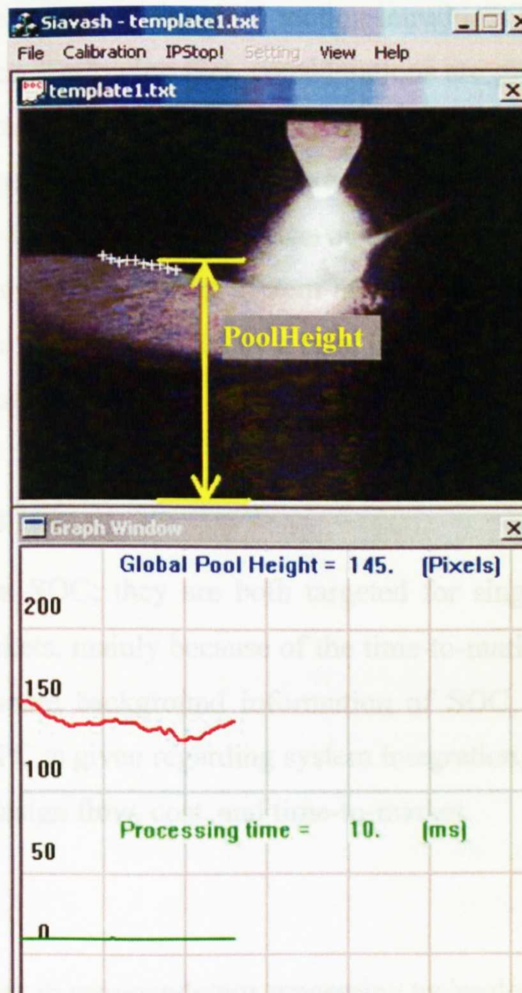


Figure 2-9 Side view image processing program

The weld pool height is defined as the distance between the measured pool edge and the bottom of the screen. As seen from the figure, there are ten measured pool heights (ten crosses), the final pool height is the average of these pool heights except the two maximum and minimum ones. This side view image processing program runs together with the front view image processing program to generate relative results from different views for the welding system to further improve its performance.

Chapter 3. An Introduction to System-On-a-Programmable-Chip (SOPC) Technology

This chapter consists of six sections. The first section introduces the predecessor of SOPC - System-on-Chip (SOC), a comparison is given between SOPC and SOC design in various aspects including performance, cost and time-to-market. The second section discusses the SOPC architecture. The third section introduces the common design flow of SOPC-based systems. Following that, considerations are given for implementing an image processing system based on a SOPC solution. The fifth section introduces several commercial/open source processor cores which are commonly used in SOPC designs. Finally, some background information of the Nios processor system, which the SIPS was based on, is given including system architecture, bus module, bus transfers and development tools. This is followed by a brief introduction to SIPS which is further expanded in a later chapter.

3.1 SOPC history

SOPC is derived from SOC; they are both targeted for single chip applications but aimed at different markets, mainly because of the time-to-market and cost factors. This section firstly gives some background information of SOC, and then a comparison between SOC and SOPC is given regarding system integration, flexibility, performance, power consumption, design flow, cost, and time-to-market.

3.1.1 SOC design

With rapid development in semiconductor processing technologies, the density of gates on the die increased in line with what Moore's law predicted [25]. This helped in the realisation of more complicated designs on the same IC. Over the last few years, with the advent of leading edge technology applications like high-definition television (HDTV) and 3rd generation mobile devices, an increasingly evident need has been that of incorporating the traditional microprocessor, memories and peripherals - or in other words the whole system - on a single silicon. This is what has marked the beginning of the SOC era.

According to a news report, "System-on-a-Chip: Technology, Markets", the worldwide

SOC market was currently estimated at nearly $14.4 billion in 2004. It is expected to grow at an AAGR (average annual growth rate) of 24.6%, this market will reach $43.2 billion by 2009. Unit growth will average 18.4% on average, per year, to reach 2.2 billion in 2009, and average unit prices will increase from a current level of $15.2 to $19.6 by the end of the forecast period [26]. Table 3-1 summaries the information about the growth in the SOC industry from 2003 to 2009 (data source from [26]).

Table 3-1 World SOC Market, 2003, 2004, 2009 and AAGR ('04-'09)

Category	2003	2004	2009	AAGR% 2004-2009
SOC Revenues ($ Millions)	10,363	14,395	43,200	24.6
SOC Units (Units in Millions)	709	945	2,200	18.4
SOC Average Selling Price ($)	14.6	15.2	19.6	5.2

Generally, SOC-based systems are implemented on ASICs which refer to non field programmable devices (e.g. standard cell or sea of gates) to make a distinction from SOPC-based systems.

3.1.2 ASIC/SOC versus FPGA/SOPC

SOPC/FPGA shares many of the characters of SOC/ASIC; however, due to the nature of the silicon chip, they have a few differences in various aspects which generally affect the decision of selecting which one is to be used to implement the actual applications. This section therefore gives a comparison overview between SOPC/FPGA and SOC/ASIC, and explains why SOPC was chosen to implement the image processing system.

3.1.2.1. Integration

Both SOC and SOPC designs are targeted for a single chip application. However because an ASIC is designed for a specific application so that its density and pin-outs can be custom specified to allow higher integration, the size and shape of the chip can be fabricated into the specified requirement to allow it to be easier fit into the end product than with on FPGA. Furthermore, based on the chip capacity, an ASIC can be mixed with specific analogue components which hence makes the chip more powerful and the end product more compact. For example more and more multimedia devices

require slim and light designs such as cell phones and other hand held devices.

3.1.2.2. Flexibility

Due to the reconfigurability of FPGA, SOPC based systems offer high flexibility. The hardware can be reconfigured to implement different tasks for different systems, for example they can be easily upgraded for further development. The system architecture can be optimised and changed, the functions of hardware and software can be easily migrated with each other to meet different system requirements. This makes SOPC designs competitive and allows them to be evolved into the next generation products. An ASIC offers no flexibility as it is designed for a specific application. Any tiny change or improvement made on the SOC design would result in a huge effort to re-design and re-layout the chip. So it is very unlikely to be able to re-use the technology in next generation products [16].

3.1.2.3. Performance

ASIC generally offers higher performance than FPGA because routings and layout can be optimised before fabrication and each block can be optimised for the desired function. On an FPGA the interconnect structure is fixed (see Figure 3-1). Also, FPGAs often require pipelining (pipelining here means inserting registers to re-clock the data between logic layers) or synchronous interface when designing for speed, which eventually leads to increasing latencies and slows down the overall speed, especially when the data is bi-directional with lots of handshaking. For example at 200MHz clock 3-4 layers of logic is expected to get through in FPGA, while in an ASIC 25 layers can be easily achieved. Furthermore ASIC offers better electrical performance as the hardware can be 'fine-tuned'.

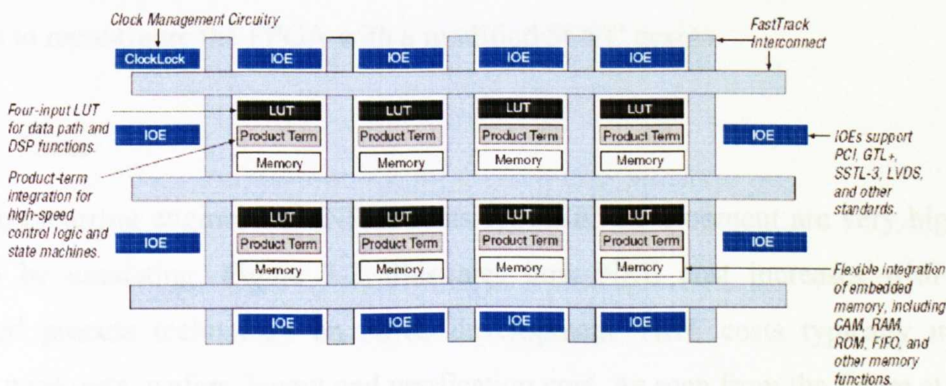


Figure 3-1 Apex 20K device block diagram (From [27])

3.1.2.4. Power dissipation

FPGAs generally consume more power than ASICs. Typically because firstly, the FPGAs contain longer routing tracks with significant parasitic capacitance, and the switching activity on these routing tracks causes significant power dissipation. Secondly FPGAs have fixed clock routing structures; all registers within a clock domain are connected to a clock network even if they are not used. When the clock is toggling, power is wasted in clock segments which connect unused registers. Finally the fixed structure of FPGAs means that not all the transistors in Look-Up Tables (LUTs) are actually used, but the unused ones still draw leakage current [28].

3.1.2.5. Design flow

FPGA/SOPC design flow is simpler than that of ASIC/SOC because the SOPC design flow eliminates the complex and time-consuming floorplanning, place and route, timing analysis, and mask/re-spin (due to verification errors) stages of the project since the design logic is already synthesised to be placed onto an already verified, characterised FPGA device.

3.1.2.6. Time-to-market

The SOC design cycle is significantly longer than SOPC. Tasks like formal verification, test development, layout and other manufacturing like fabrication would significantly increase the ASIC time-to-market period, while designing SOPCs in FPGAs much of the routing, placement and timing etc are handled by software. Because FPGA devices are purchased free from manufacturing defects, formal verification is not required which could save weeks of time. Furthermore, design re-spin and modifications would cause ASICs in delayed time-to-market, whilst it would probably just take a few minutes to reconfigure the FPGA with a modified SOPC design.

3.1.2.7. Costs

The non-recurring engineering (NRE) costs for ASIC development are very high and tend to be escalating. Figure 3-2 illustrates how NRE cost increases with more advanced process technology on ASIC development. NRE costs typically include design, mask-sets, wafers, layout and verification cost. As seen from the figure at 90nm

it needs $30 million dollars to develop an ASIC, and that's only if it can be done correctly at the first attempt. So developing an ASIC is both risky and expensive. New business models say that only the highest volume consumer products can justify SOC [16]. SOPC development doesn't have a fixed NRE cost and generally it's much lower than that of ASICs because no verification and fabrication is required. Furthermore, no NRE cost is associated with design re-spins and modification for SOPC design, but this could lead to a significant NRE increase for ASICs. For instance it would cost approximately $850,000 for a mask manufactured using 130 nm technology [29].

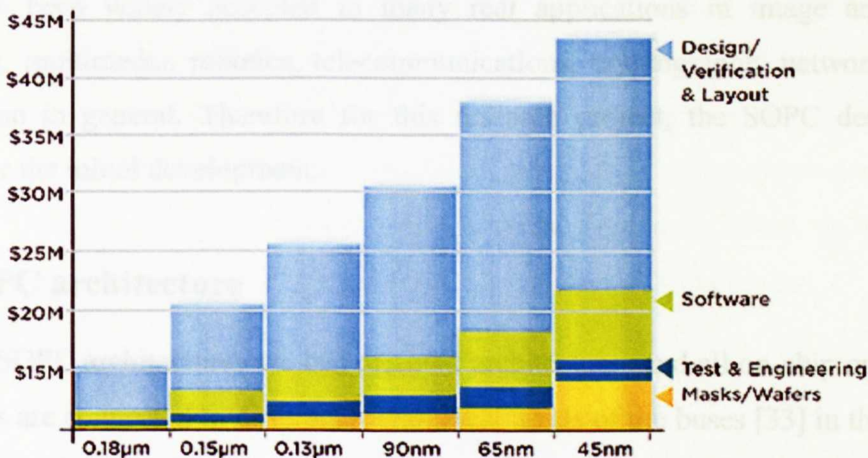


Figure 3-2 NRE cost of developing ASICs
(Source: International Business Strategies, Inc)

However, although the NRE costs for ASIC are high, the unit cost is much lower than FPGAs, as ASICs are targeted for very high volume market. Therefore the total cost for ASIC is only more effective than FPGAs for the very high consumer market, while for smaller designs and/or lower production volumes, the FPGA is a more ideal choice than ASIC.

3.1.2.8. Conclusion

Although SOC based systems offer high performance, better integration and lower power consumption than SOPC based systems, it is risky and the NRE is the major threshold to prevent the vendors going for SOC solution, only if they understand the risk and can predict, long in advance, how many devices their market will require [15]. If they choose too many, they pays the steep cost of inventory. If they choose too few then they may miss important market opportunities or lose the ability to acquire market share. When an FPGA is used, the FPGA vendor shoulders the inventory risk, which is

shared across a much larger number of customers.

Moreover, SOPC-based systems offer excellent development platform for research and teaching activities like [30], [31] and [32] as they can be re-used for various student projects and re-configured as many times as required to verify a theory or prove some concepts etc. Also, because of the high risk of ASIC design, more and more vendors tend to use FPGA based emulation platforms to verify the SOC design before going for fabrication. In industry the SOPC has been a good alternative to SOC. That is why SOPC has been widely accepted in many real applications in image and signal processing, multimedia, robotics, telecommunications, cryptography, networking and computation in general. Therefore for this research project, the SOPC design was adapted for the initial development.

3.2 SOPC architecture

Common SOPC architectures are bus oriented architectures and all on-chip or off-chip peripherals are connected to different hierarchical levels of the buses [33] in the system depending on their bandwidth and speed requirements. In modern system level design [34], the use with configurable and re-usable IPs becomes more and more important in terms of cost and time-to-market. Therefore PLD vendors usually develop their own bus systems or offer licenses to the bus systems developed by third-party semiconductor manufacturers to allow all bus compatible IPs including their standard IPs, third-party IPs and other custom IPs to be optimised and easily integrated into their own PLD devices. Therefore SOPC architectures depend on the PLD devices and processors which are chosen. Four kinds of standard on-chip buses are commonly available in SOPC design, which are the Avalon bus [35], Advanced Microcontroller Bus Architecture (AMBA) bus [36], CoreConnect bus [37] and Wishbone bus [38]. The Avalon bus was developed by Altera to allow all peripherals to interface with its own softcore processors. The AMBA bus developed by ARM is adopted in Altera's Excalibur PLD because an ARM embedded microprocessor system is pre-fabricated in this PLD. AMBA has four levels of hierarchy: Advanced high-performance bus (AHB), advanced system bus (ASB), advanced peripheral bus (APB) and the most recent ones advanced extensible interface (AXI). CoreConnect was originally developed by IBM to be used in systems embedded with the PowerPC processors. It is also licensed to

Xilinx's high-end PLD devices. It has three levels of hierarchy: Processor Local Bus (PLB), On-chip Peripheral Bus (OPB), and Device Control Register (DCR) to allow the processors to efficiently communicate with different speed peripherals. Wishbone is known for its flexible interconnection capability and free open source. Therefore it is widely adopted into open source IPs. In order to allow communication between different hierarchical levels on the same bus system or even different bus systems special IPs are developed to cope with this problem which are often called bridges, for instance, AHB-APB, AHB-Avalon, PLB-AHB.

Once the bus system is adopted, designers can pay more attention to the inner architecture, which is concerned with the hardware/software partitioning, processor optimisation, data and control paths, definition of on-chip and off-chip components etc. Section 3.4 describes some necessary considerations should be given to design an SOPC based system by presenting an example of image processing system.

3.3 SOPC design flow

Common SOPC design flow was developed based on traditional FPGA design flow which includes register transfer level (RTL) coding, simulation and synthesis, with the addition of software development flow and hardware (HW)/software (SW) partitioning. Figure 3-3 illustrates the block diagram of a typical SOPC design flow. In practical, commercial programmable logic device (PLD) suppliers like Altera and Xilinx provide sophisticated computer-aided design (CAD) tools dedicated to their PLDs to help design and analyse SOPC designs with increasing complexity, and these sometimes save a lot of work in a specific design step. So the actual implementation of the design flow might be slightly different depending on which PLD vendor is chosen.

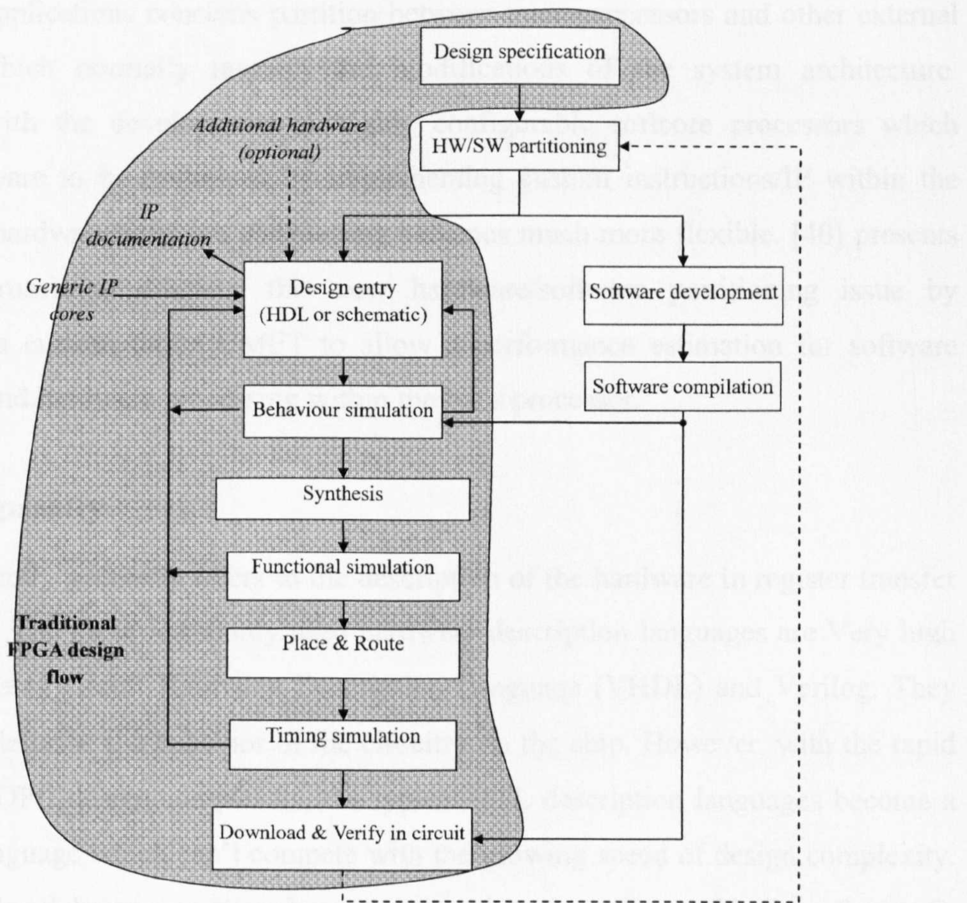


Figure 3-3 Overall SOPC design flow

3.3.1 Design specification

The design specification normally includes system requirements, selection of the FPGA family and device to be evaluated on, number of input/output (I/O) pins and the required standards, clocks and their frequency requirements, memory requirements, test methodology etc.

3.3.2 Hardware (HW) / Software (SW) partition

Hardware/software partitioning is concerned with deciding which functions should be implemented in hardware and which ones in software. Generally the hardware approach provides better performance but costs more, while software is cheaper, and more flexible, but the speed performance is not as good as hardware. When hardware and software trade-off becomes an important consideration in terms of costs, time-to-market and performance, a hardware/software Codesign [39] approach is generally used to achieve the optimal balance. Traditional hardware/software partitioning in

embedded applications concerns partition between microprocessors and other external hardware which normally requires the modifications of the system architecture. However, with the development of highly configurable softcore processors which allows software to be optimised by implementing custom instructions/IP within the processors, hardware/software partitioning becomes much more flexible. [40] presents a new approach to discover this new hardware/software partitioning issue by developing a custom tool COMET to allow a performance estimation for software processing and hardware processing within the Nios processor.

3.3.3 Design entry

The design entry generally refers to the description of the hardware in register transfer level (RTL). The most commonly used hardware description languages are Very high speed integrated circuit Hardware Description Language (VHDL) and Verilog. They are used to describe the behavior of the circuitry on the chip. However, with the rapid growth of SOPC design complexity, the typical RTL description languages become a low level language which can't compete with the growing speed of design complexity. Other high level languages have been developed such as SystemVerilog, SystemC, HardwareC, SpecC, JVHDL and SuperVerilog [41]. They aim at automating the creation of a HDL language description from a program written in a higher level software-like language. Then the designer can concentrate more on the system modeling design. Also, some PLD suppliers often offer the electronic design automation (EDA) software tools normally with a friendly Graphical User Interface (GUI) to ease the system level integration. Altera's SOPC Builder [42] is one of the examples.

This design entry step also includes specification of the methodology, hierarchical design partitioning etc. Following the trend of system level design, it is very important to use a hierarchical design methodology and design/select modular components for the SOPC system. A good hierarchical design can also simplify the overall verification and reduce the design time because individual IP blocks can be separately designed and verified.

3.3.4 Simulations

Simulations are usually to verify the function of the RTL design. Normally it includes behaviour simulation, functional simulation and timing simulation. Behaviour simulation is used to verify the logic function of a module in behaviour level while the other two are gate-level simulation that actually simulate the module at gate level with the option of adding the standard delay format (SDF) after synthesis and place & route. The simulation process often requires the designer to modify the design if there is a problem with the simulation results. Most PLD vendors provide gate level simulation tools which are integrated into their development software.

3.3.5 Synthesis

Synthesising is the process of converting a design representation from RTL code to the gate level [29]. This process is generally automated by EDA tools. A netlist file is produced after synthesis. This netlist is normally in Electronic Design Interchange Format (EDIF).

3.3.6 Place & route

Place and route actually lays out the design on the FPGA device based on the netlist file. It requires the assignments of the system I/Os to the actual pin outs of the device. Obviously this is a manufacturer-dependent process because different FPGA vendors use different structures. This is also performed by EDA tools. A static timing analysis is generally performed after place & route.

3.3.7 Download & verify in circuit

This final step is to download and test the system design in the real device. The most popular download scheme is via Joint test action group (JTAG) interface. Most PLD vendors provide on chip debugging tools to verify the system in circuit both in hardware and software aspects such as Altera's SignalTap [43] and Xilinx's ChipScope Pro. The overall design flow can be restarted from HW/SW partitioning process if there is a need for optimisation or from the design entry process if another IP is required to be developed and integrated into the end application.

3.3.8 Software development

The software is developed for the soft core processor. C, C++ and assembly are the common embedded languages used for software development in SOPC designs.

3.3.9 Software compilation

Normally the processor supplier provides a full development kit including software compilation tool to support SOPC development. This also includes the compilation for simulation. The compilation result is normally a binary file containing the program data and instruction information which can be written into the on-chip memories directly or off-chip memories like SRAM or FLASH via a serial communication interface.

3.4 Real time image processing system based on SOPC

Based on the functionality of a real time image processing system described in Chapter 2, the image processing system based on SOPC can be roughly broken down into four parts which are video acquisition block, video processing block, video buffer and video output block in the behaviour level. In practice, these function blocks can be implemented as separate IPs wrapped with a dedicated bus interface and integrated into a standard on-chip bus system. However, the flexibility of an SOPC design would result in more considerations given in designing the details of the system architecture. Firstly, although SOPC defines an entire system that can be integrated into a single silicon chip, how much portion of this system should be integrated into this chip and how much should or must be left off-chip? Secondly, since dedicated circuit structures which built in hard fabric can be implemented on FPGAs and they offer higher performance but less flexibility than the equivalent structures built in soft fabric, concerns about whether to use hard fabric or soft fabric to implement specific functions must be taken to the design in order to achieve the best optimal balance between performance, cost and system flexibility. Finally it's the traditional hardware/software partitioning issue, which concerns how much processing and control should be implemented by the software and how much should be done by the hardware. All of these considerations must be well balanced on cost, performance and the types of FPGA chosen etc.

3.4.1 On-chip vs. off-chip

Ideally the more of the system which is integrated into the programmable chip, the more flexibility, lower power consumption, higher integration and mostly better performance the system can offer. However, this is often restricted by the time-to-market period, cost, and the available feature/resource of the selected programmable device. Several examples are given to explain what the designer should be concerned about when making on-chip / off-chip decisions for an image processing system.

For example if a USB camera is chosen as the video source, then a USB host controller is needed to control this camera and receive data stream via the USB bus. So should this host controller be integrated with the system core on-chip or just simply use an external USB host controller chip to do the job? Certainly if there is an available developed IP for USB host controller (i.e. from public source) then the on-chip solution is the best option in terms of performance and the simplification of the off-chip board design. Otherwise, considerations must be given whether to spend years to develop such a complex IP core or pay a few thousand pounds to buy one in the market, or to implement this function just simply by using an external chip although it is still costly. By using the off-chip solution, the overall performance degrades and the extra software development for controlling this USB chip and exacting the image data from the serial data streams must also be taken into account as USB is not solely designed for vision tasks.

The second example is for the video memory buffer. On-chip memory is fast compared to off-chip memory. However in reality a real-time image processing system generally requires buffering a large amount of data compared to the available on-chip memory resource from most of the FPGAs, especially when the video frame has a high resolution and deep pixel depth. Certainly to buy a very good FPGA with sufficient memory to buffer all required video frames is the best option. But when the selected device doesn't have the required resource then off-chip memory has to be used instead. However, if a small buffer is required in the design then on-chip memory is the best solution. Also if off-chip memory is used, then a memory controller must be developed to drive this off-chip memory.

Some special FPGAs have integrated analog circuits such as digital-analog converter (DAC) and analog-digital converter (ADC). If the selected FPGA has this feature and DAC or ADC function is required, for example, a DAC is commonly used to convert the digital video output to analog signals for display on analog monitors, then use of the DAC function on chip can simplify the off-chip board design and hence reduce the overall cost. Otherwise, this function has to be implemented in off-chip.

For the actual image processing, on-chip processing is generally the best way to provide high efficient processing power because it can maintain a short delay, low latency and wide throughput. However, off-chip co-processors might be an option to share the processing load with the on-chip processing engine if the selected device doesn't support multiprocessors mode.

There is no golden rule for making on-chip/off-chip decision for SOPC design; a good solution must be made by considering all aspects including cost, time-to-market, performance, selection of the programmable device and the actual system requirements.

3.4.2 Hard logic fabric vs soft logic fabric

The soft logic fabric of an FPGA consists of an array of combinational logic elements while hard logic fabric is a circuit structure that allows the implementation of a logic function that could also be implemented in the soft logic fabric [15]. Obviously this hard logic fabric on FPGA is similar to the cell or gate on ASIC. Like ASIC, the benefit of using hard logic fabric is the structure is smaller, faster and consumes less power than the equivalent structure built out of the programmable soft fabric. The most commonly used functions built in hard logic fabric are dedicated flip-flops, embedded memory blocks, multipliers and even processors (see section 3.5 for information of most commonly used hard processors). Furthermore, recent advanced FPGAs such as Stratix [44] from Altera provide dedicated DSP blocks to perform matrix operations, floating point operation etc. Therefore for image processing, if a DSP function is required, then the design should use the DSP blocks if they are available on the selected device. Also if memory is used, for example for the small video buffers, then the design should use the embedded memory blocks instead of LUTs.

Certainly all of the function of an image processing system can be turned into hard logic fabric, but this obviously just makes an ASIC instead of FPGA. This is not realistic. The only way to determine if a hard structure is truly useful in the FPGA context for specific function part of the image processing system is to empirically measure the net benefit of the structure across a set of benchmark applications that are representative of the target market [15].

3.4.3 Hardware vs software

The range of the hardware here includes the soft fabric and hard fabric on the FPGA while the software means the operation performed by the processor. This is actually a typical hardware/software partitioning problem. The advantage of using hardware control and processing is that it delivers fast speed; however, it takes more overall hardware resource, offers less flexibility and possibly degrades the fmax. In contrast, software control and processing provides high flexibility and more resource saving but at lower speed.

Two distinct cases can be considered when designing an image processing system. The first one is, high data thought-put and complex data paths are the challenges in performing real-time image processing, in order to increase the data transfer efficiency, burst transfer (called direct memory access) should be employed throughout the system wherever there is a requirement for high bandwidth transfer instead of using the software approach to move data from one target to another by the processor. The second consideration is the actual image processing. If specific processing algorithms are required, then hardware processing can be a better option like adapting the DSP blocks, optimising the CPU instructions and even implementing custom instructions. Generally to develop an image processing like SIPS, in the earliest stage, single processor is implemented in order to ensure the video data consistency throughout the system and provide the flexibility to perform various processing, and then more optimisation can be done specifically within the processor or by implementing low-level processing tasks on the on-chip hardware for specific applications. Certainly if hardware resource is available, implementation of multiple processors on the same chip is also a good choice to increase the processing rate.

3.5 Common SOPC Processor overview

There are several commercial/open source processors which are designed for use in PLD devices. PLD vendors generally provide two types of processors either built in hard fabric or soft fabric. As described in section 3.4.2, hard processor offers better performance and higher density. However, the highly configurable feature of soft processor can make up this difference by using a specialised instruction set. Also, a programmable quantity of processors can be instantiated as needed, each tuned to required area and performance specifications [15]. This section gives an overview of some commonly used processors in SOPC designs.

3.5.1 Soft processor

There are several soft processors available. They are Nios [45] and Nios II developed by Altera, Microblaze [46] and Picoblaze developed by Xilinx, Mico8 and Mico32 developed by Lattice, OpenRISC [47] developed by OpenCores, Leon [48] developed by Gaisler Research, OpenSPARC developed by Sun Microsystems. The latter three are open source and under the GNU general public license. This section gives an overview of some of these soft processors.

3.5.1.1. Nios

The Nios CPU is a configurable, 16- or 32-bit general-purpose RISC processor with a single issued, 5-stage pipelined Harvard architecture and a compiler-friendly instruction set. The Nios CPU implementation includes up to 512 internal general-purpose registers. The compiler uses the internal registers to accelerate subroutine calls and local variable access. The Nios instruction set includes Load and Store instructions that the compiler uses to accelerate data access and local-variable (stack) access. Users can incorporate custom logic directly into the Nios arithmetic logic unit (ALU) to modify or extend the Nios instruction set. For example accelerate software algorithms by reducing the number of operations for "inner loop" tasks to a single cycle, implement instructions in a single-cycle or multi-cycle. The automatically-generated software development kit (SDK) includes macros for accessing custom instruction hardware for C and assembly-language programs. The Nios processor outputs the instructions and data via the Avalon bus master ports (see section 3.6).

3.5.1.2. Microblaze

MicroBlaze 7.0 features a configurable 3-stage or 5-stage pipeline (trading off size for clock rate) Harvard architecture, a general purpose register window with thirty-two 32-bit registers, 32-bit instruction word with three operands and two addressing modes, with an instruction completing in each cycle. Cache structure and size, peripherals, and interfaces can be customised to the application. In addition, hardware support for certain operations, such as multiplication, division, and floating-point arithmetic, can be added or removed. For performance critical portions of software programs, custom coprocessors can be added to MicroBlaze and connected via a dedicated FIFO-style coprocessor interface called Fast Simplex Link (FSL). Also different combinations of buses such as OPB, local memory bus (LMB) and Fast Simplex Link (FSL) can be implemented to allow flexible system design.

3.5.1.3. OpenRISC

The OpenRISC 1200 is a 32-bit scalar RISC with Harvard architecture, 5 stage integer pipeline, virtual memory support (MMU) and basic DSP capabilities. It features single-cycle instruction execution on most instructions and a thirty-two register general-purpose register window. Additional units such as a floating-point unit can be added as standard units. Eight custom units can be added and controlled through special-purpose registers or customer instructions. Default caches include a one-way direct-mapped 8kB data cache and a one-way direct-mapped 8kB instruction cache, each with 16-byte line size. Both caches are physically tagged. OpenRISC has interfaces including power management interface, development interface, interrupt interface, and instruction and data WISHBONE host interfaces.

3.5.1.4. Leon

The LEON3 is a 32-bit processor based on the SPARC V8 architecture. It implements a 7-stage pipelined, separate instruction and data cache Harvard architecture. It supports 2x32 register window size. A unique debug interface allows non-intrusive hardware debugging and provides access to all registers and memory. Other floating-point units/custom units can be connected to the processor via a general interface to allow parallel or sequential execution with the integer unit (IU). Built in AHB masters allow the instruction and data cache to interface to the AMBA-AHB bus.

3.5.2 Hard processor

Hard processors are usually developed by specialised semiconductor companies. PLD vendors just simply purchase licenses to them and fabricate and optimise the processors into their specific FPGA family. Examples are Altera's Excalibur devices [49] and Xilinx's Virtex II Pro FPGAs [50]. Some PLD vendors produce their own processors and fabricate them with the FPGA to offer a variation of SOPC design. They are like a configurable microcontroller. Typical examples are Programmable-System-on-a-Chip (PSoC) [51] from Cypress and FPSLIC (Field Programmable System Level Integrated Circuits) [52] from Atmel.

3.5.2.1. Excalibur

The Excalibur devices integrate an industry-standard ARM922T™ 32-bit RISC processor core operating at up to 200MHz. This processor has a Harvard architecture with separate instruction and data memory. It features an ARM v4T instruction set with 32 bit load and store. The instruction set supports 16- and 8-bit memories, 5 stage pipeline, and task identifier register for real time operating system (RTOS) support. It has 32x8 bit hardware multiplier but no hardware divider and floating point unit (FPU). The microprocessor system has a built-in SRAM and SDRAM controller. Also it has embedded programmable on-chip peripherals such as universal asynchronous receiver/transmitter (UART), flexible interrupt controller and general-purpose timer etc. The ARM microprocessor subsystem is implemented as an "embedded stripe" next to the "FPGA stripe" in the whole device and AHB-Avalon bridge is used to link the communication.

3.5.2.2. Virtex™-II Pro

The Virtex™-II Pro platform FPGAs provide up to two PowerPC™ 405 developed by IBM. This PowerPC processor features PowerPC User Instruction Set Architecture (UISA), 5-stage data path pipeline with single-cycle execution of most instructions including loads and stores, 32 x 32-bit general purpose register window, 32 bit Harvard architecture, hardware multiplier and divider but no FPU. Integration of the PowerPC core into the Virtex™-II Pro device is accomplished by taking advantage of the IPImmersion architecture, which allows hard IP cores to be diffused at any coordinate within the Platform FPGA fabric, while maintaining unprecedented connectivity with

the surrounding Configurable Logic Block (CLB) array. IBM CoreConnect bus architecture is used in both the processor system and the FPGA system.

3.5.2.3. PSoC

The PSoC is a family of mixed-signal arrays featuring a microprocessor developed by Cypress. This microprocessor is called M8C which features an 8-bit Harvard architecture. PSoC has configurable logic blocks which are designed to implement the analog-digital integrated peripheral interfaces, which include analog-to-digital converters, digital-to-analog converters, timers, counters, and universal asynchronous receivers–transmitters (UARTs) etc.

3.5.2.4. FPSLIC

The FPSLIC family of devices combines an 8-bit Harvard architecture AVR microcontroller with a 32kB SRAM, a 40k gate equivalent FPGA device and fixed peripherals including UART and timer/counter on a single chip. FPGA custom peripherals are supported by using the AVR peripheral control.

3.6 Introduction to the Nios processor system and development tools

Altera is an industry pioneer for providing SOPC solutions. Not only does it supply FPGA devices with various performance levels, but provides IP blocks including the soft processor and a serious of extensive EDA tools to support the development of SOPC designs. In the University of Liverpool, Altera's PLDs have been already adapted into teaching and researching activities for many years. SIPS was therefore originally developed based on the Nios processor architecture which is targeted to the FPGAs provided by Altera.

3.6.1 Nios system architecture

As a commercial processor supplier, Altera provides a specific system architecture for SOPC designers to connect all function blocks and the other user logic with the Nios processor, via the dedicated system bus protocol – Avalon bus module. Figure 3-4 illustrates a block diagram of the Nios processor system architecture, where system module refers to the portion of the design where involves with interconnections with the Avalon bus module. The reason for emphasising the system module here is because this portion can be automatically integrated and generated by the Altera's integration tool – SOPC Builder, which is described in section 3.6.4.

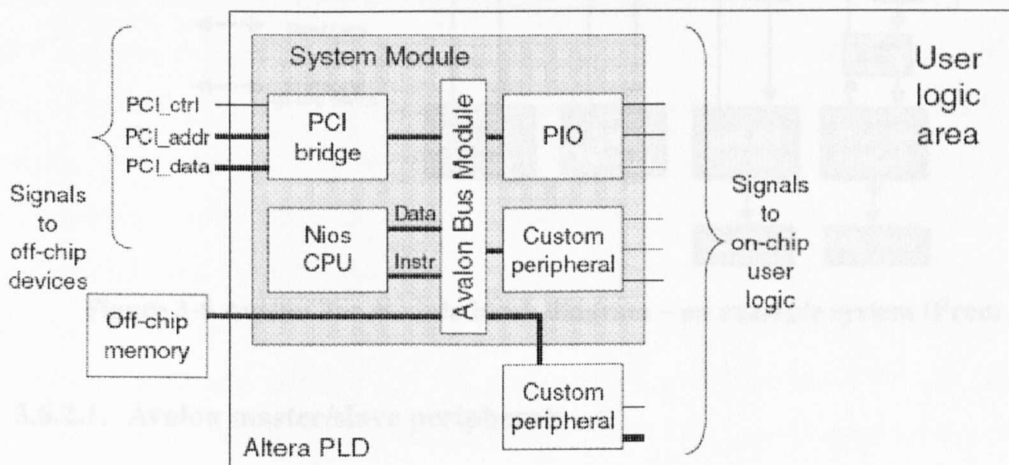


Figure 3-4 System module integrated with user logic into an Altera PLD (From [35])

In this system architecture, all modules connected to the Avalon bus are treated as a separate function entity and named peripherals (see section 3.6.2.1). They could be vendor provided IP blocks such as the Nios processor, parallel input/output (PIO) [53] or custom defined modules. These custom defined modules could be parameterised and

documented as an IP block for further usage. For integration purpose, all modules need to interface to the Avalon bus module have an Altera pre-defined bus interface, which could be either a set of master ports or slave ports (see section 3.6.2.2). The next section describes this bus interface in details.

3.6.2 Avalon bus module

The Avalon bus module is the backbone of the Nios processor system. It is the abstraction of the communication paths for all the sub modules contained in the system module. Not only does it manage the interconnection, but also provides data path multiplexing, address decoding, wait-state generation, arbitration and streaming read/write capacities for all component peripherals connected to it. Figure 3-5 shows an example system block diagram.

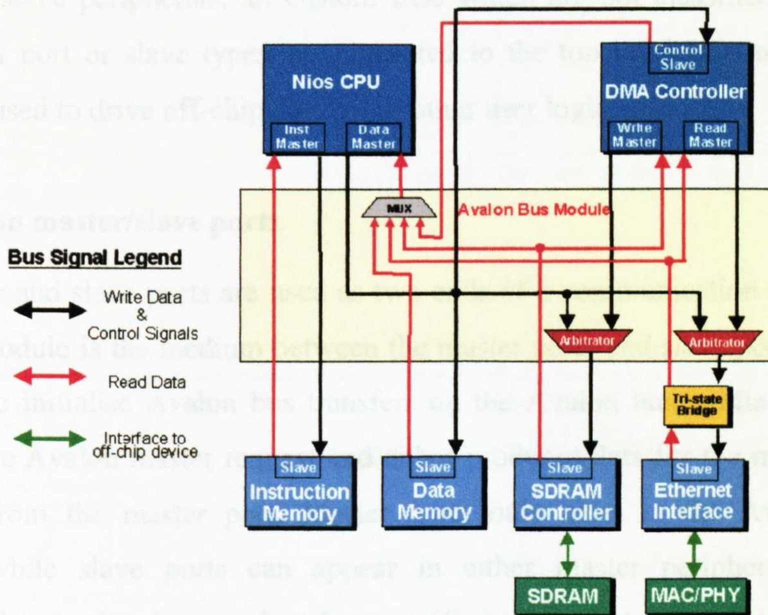


Figure 3-5 Avalon bus module block diagram – an example system (From [35])

3.6.2.1. Avalon master/slave peripherals

An Avalon peripheral is a modular component which could be either on-chip or off-chip that performs some system-level task, and communicates with the other peripherals in the processor system through the Avalon bus. Based on the control function Avalon peripherals can be classified into either a master peripheral or a slave peripheral. The main characteristic of master peripherals differ from slave peripherals is that the master peripheral has the ability to initialise Avalon bus transfers to the slave

peripheral. A master peripheral contains at least one Avalon master port. However, it can have Avalon slave ports to accept requests from the other master peripherals generally for configuration purposes. Avalon slave peripherals only respond to the bus transfer requests from the master peripherals.

Avalon bus peripherals can be understood as specialised IP blocks designed for use in the Nios processor system, as long as they perform sufficient system level tasks, and are fully synthesisable, verified and predicted to be reusable. Any user logic with an interface to the Avalon bus module can also be treated as a peripheral. In this thesis an Avalon master peripheral is sometimes abbreviated as 'master' and Avalon slave peripheral is abbreviated as 'slave'.

In the master/slave peripherals, all custom I/Os which are not classified as standard Avalon master port or slave types are promoted to the top level I/Os of the system module to be used to drive off-chip devices or other user logic.

3.6.2.2. Avalon master/slave ports

Avalon master and slave ports are used as two ends of a communication channel. The Avalon bus module is the medium between the master ports and slave ports. A master port is used to initialise Avalon bus transfers on the Avalon bus, while a slave port responds to the Avalon master request and either produces data for the master port or accept data from the master port. Master ports only exist in the Avalon master peripherals, while slave ports can appear in either master peripherals or slave peripherals. The Avalon bus module has specified a number of standard signals to define this master/slave port. Table 3-2 and Table 3-3 list the most commonly used Avalon bus signals defined for the master/slave port interface. Not all signals are required for all master/slave ports, the selection of these signals are based on the types of transfer that the master/slave ports require. For example, *dataavailable*, *readyfordata* and *endofpacket* are only used for streaming slave. The design of the Avalon interface in master/slave peripherals should conform to the Avalon bus interface specification [35] to ensure the Avalon bus module functions correctly.

Table 3-2 Avalon slave port signals

Signal type	Width	Dir	Req	Description
clk	1	in	no	All bus transactions are synchronous to this clock. Only asynchronous slave ports can omit clk.
reset	1	in	no	Global reset signal. Implementation is peripheral specific.
chipselect/ chipselect_n	1	in	yes	Chip select signal to the slave. The slave port should ignore all other Avalon signal inputs unless chipselect is asserted.
address	1-32	in	no	Address lines from the Avalon bus module.
read/ read_n	1	out	no	Read request signal to slave.
readdata	1-32	in	no	Data lines to the Avalon bus module for read transfers.
write/write_n	1	in	no	Write request signal to slave.
writedata	1-32	in	no	Data lines from the Avalon bus module for write transfers.
waitrequest	1	out	no	Used to stall the Avalon bus module when the slave port is not able to respond immediately.
readyfordata	1	out	no	Signal for streaming transfers. Indicates that the streaming slave can receive data.
dataavailable	1	out	no	Signal for streaming transfers. Indicates that the streaming slave has data available.
endofpacket	1	out	no	Signal for streaming transfers. May be used to indicate an “end of packet” condition to the master port. Implementation is peripheral-specific
irq	1	out	no	Interrupt request. Slave asserts irq when it needs to be serviced by a master.

Table 3-3 Avalon master port signals

Signal type	Width	Dir	Req	Description
clk	1	in	yes	All bus transactions are synchronous to clk.
reset	1	in	no	Global reset signal. Implementation is peripheral specific.
address	1-32	out	yes	Address lines from the Avalon bus module. All Avalon masters are required to drive a byte address on their address output port.
read/ read_n	1	out	no	Read request signal from the master port.
readdata	8, 16, 32	in	no	Data lines from the Avalon bus module for read transfers.
write/write_n	1	out	no	Write request signal from the master port.
writedata	8, 16, 32	out	no	Data lines to the Avalon bus module for write transfers.
waitrequest	1	in	yes	Forces the master port to wait until the Avalon bus module is ready to proceed with the transfer.
endofpacket	1	in	no	Signal for streaming transfers. May be used to indicate an end of packet condition from the slave to the master port. Implementation is peripheral specific.

3.6.3 Avalon bus transfers

The Avalon bus module provides various types of bus transfers to satisfy different master/slave port requirements. For example, the streaming transfer accommodates high bandwidth peripherals that a block of data is often required to send/receive, while fundamental transfer only transfers a single unit of data within one clock cycle, and because of its simplicity it generally requires less control and hence saves more bus resource. The Avalon master port is not directly connected to the slave port due to the existence of the Avalon bus module, signals output from a master port on the Avalon bus might be different from the corresponding signals that are input to the slave port on the target peripheral. Therefore, a separate discussion for the Avalon bus transfers on the slave port interface and on the master port interface is presented in this section.

Only descriptions of the bus transfers involved in the development of SIPS are given. Information of the other types of bus transfers such as read transfers with latency, transfers with fixed wait-states, transfers with setup and hold time, and tri-state transfers can be obtained from the Avalon bus interface specification.

3.6.3.1. Avalon slave transfers

This section describes the transfers between the Avalon bus module and the slave peripherals which respond to the transfer request. Description and timing of fundamental slave transfer, slave transfer with peripheral-controlled waitrequest and slave streaming transfer are presented.

Fundamental slave transfer

The fundamental slave read transfer is the basis for all Avalon slave read transfers. All other slave read transfer modes use a sub set of the fundamental signals, and implement a variation of the fundamental slave transfer. The fundamental slave transfer is initiated by the Avalon bus module, and transfers one unit of data, the full width of the peripheral's data port, to the Avalon bus module in a write transfer or from the Avalon bus module in a read transfer. All of the data are sampled at rising edge of the clock.

<u>Fundamental Avalon slave read transfer:</u>

Figure 3-6 illustrates a timing diagram of the fundamental Avalon slave read transfer. All control signals including *address*, *byteenable_n*, *read_n* and *chipselect* are asserted from the Avalon bus module before time C. The Avalon slave peripheral starts decoding the *address*. At time D, *readdata* becomes valid. At the rising edge of the next clock cycles time E, *readdata* is read by the Avalon bus model, the read transfer terminates. The next clock cycle could be the start of another slave transfer.

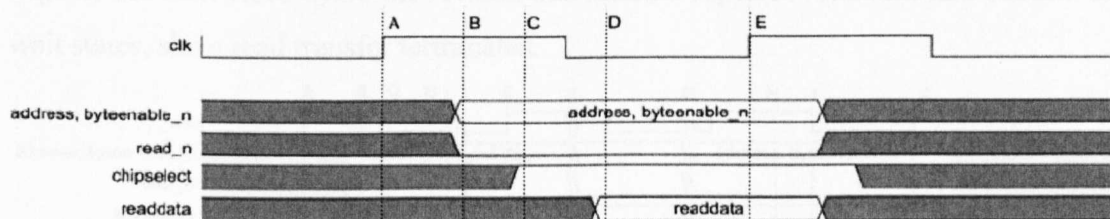


Figure 3-6 Fundamental slave read transfer (From [35])

Fundamental Avalon slave write transfer:

Figure 3-7 illustrates a timing diagram of the fundamental Avalon slave write transfer. All control signals along with the *writedata* are asserted to the slave port before time C. At the rising edge of the next clock cycle time D, the Avalon slave peripheral captures the *writedata* and the write transfer terminates.

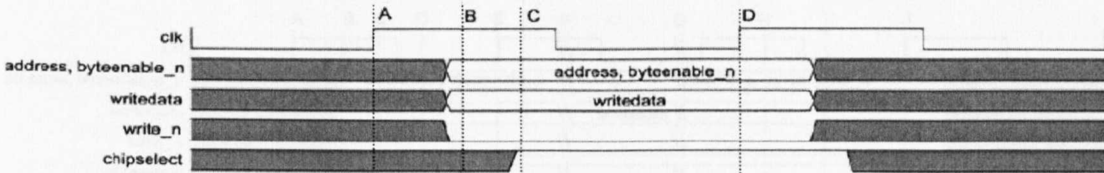


Figure 3-7 Fundamental slave write transfer (From [35])

Avalon slave transfer with peripheral-controlled waitrequest

Slave transfers with peripheral-controlled waitrequest are used for slave peripherals which need a number of clock cycles to respond the requests, and this number is not fixed and depends on the current state of the slave peripherals. For example, auto-refresh is needed to be performed on a Synchronous Dynamic Random Access Memory (SDRAM) device within a limited time scale, so any master peripheral which sends requests to the SDRAM device while the auto-refresh is in progress must be held until the device is free. And this hold period is dependent on the state of the peripheral. In such a case slave transfers with peripheral-controlled waitrequest are required.

Avalon slave read transfer with peripheral-controlled waitrequest:

Figure 3-8 illustrates a timing diagram of the slave read transfer with peripheral-controlled waitrequest. *Waitrequest* asserted at time D by the slave peripheral after the read request inilialised. So in the next clock cycle the read request is held up. At time H the slave produces valid *readdata* so *waitrequest* is deasserted at time I. At the rising edge of the next clock cycle the Avalon bus module captures *readdata* and cancels the wait states, slave read transfer terminates.

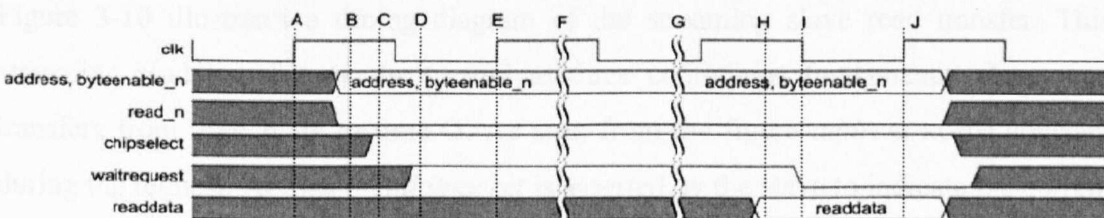


Figure 3-8 Slave read transfer with peripheral-controlled waitrequest (From [35])

Avalon slave write transfer with peripheral-controlled waitrequest:

Figure 3-9 illustrates a timing diagram of the slave write transfer with peripheral-controlled waitrequest. Similar to the read transfer, the slave peripheral is not ready to accept the *writedata* until time J, so *waitrequest* is asserted at time D after the write transfer inilialised, until time I.

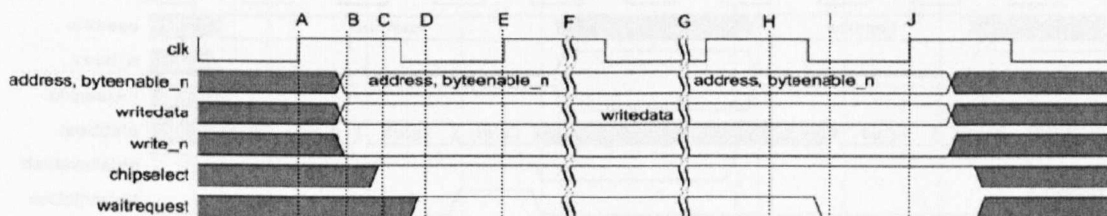


Figure 3-9 Slave write transfer with peripheral-controlled waitrequest (From [35])

Streaming slave transfer

Streaming transfers between a streaming master port and a streaming slave port enable a large block of data to be successively transferred. Simple flow control signals are used in the streaming slave, such that whenever the slave has new data or can accept new data, the Avalon bus module just transfers the data. Unlike the typical DMA controllers, the Avalon bus module eliminates the need for the master to continuously check the status registers in the slave peripheral to determine whether the slave can send or receive data, and hence the data transfer efficiency is increased. Furthermore, the *address* bus doesn't need to be incremental like in typical DMA transfers; the implementation of this *address* bus is dependent on how the slave decodes it. For example, the SDRAM page read/write only requires the first column address so in Avalon streaming transfer the *address* can stay constant during the whole period of the streaming transfer. In this thesis, streaming transfer is sometimes called a DMA transfer or burst transfer.

Streaming slave read transfer:

Figure 3-10 illustrates a timing diagram of the streaming slave read transfer. This streaming read transfer can be treated as three continuous fundamental slave read transfers from time A up to time G. As seen from the figure *address* keeps constant during the request. At time F *endofpacket* is asserted by the slave to indicate the current packet is ending. In the same cycle at time G, *dataavailable* is deactivated, the slave peripheral cannot present valid data and the Avalon bus module suspends the read

request. At time I the slave asserts valid *dataavailable* again; the steaming read request is retrieved immediately at time J. New data is presented in the next clock cycle and the Avalon bus module starts capturing the remaining data until time N, when the streaming slave read transfer terminates.

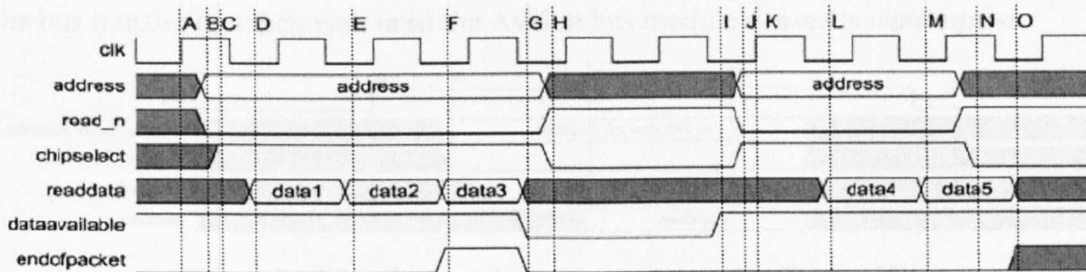


Figure 3-10 Streaming slave read transfer (From [35])

Streaming slave write transfer:

Figure 3-11 illustrates a timing diagram of the streaming slave write transfer. It is very similar to the streaming read transfer, except that *readyfordata* is used to control the data flow.

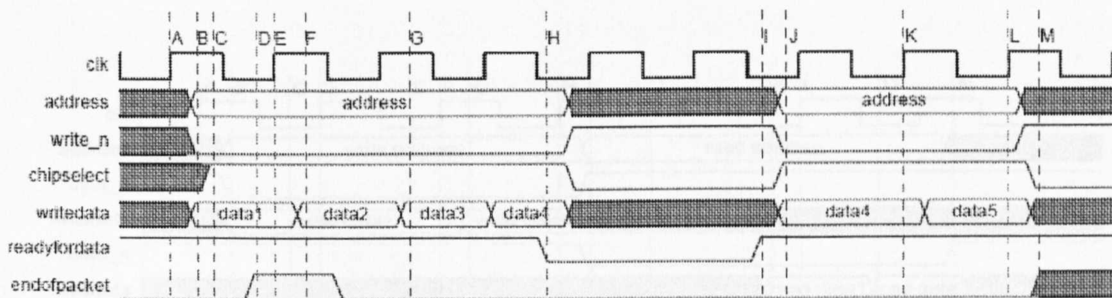


Figure 3-11 Streaming slave write transfer (From [35])

3.6.3.2. Avalon master transfers

Avalon master transfers generally refer to the transfers between the master peripherals and the Avalon bus module which initialises the transfers. The timing of the Avalon master transfers is similar to that of the Avalon slave transfers. Figure 3-12 and Figure 3-13 illustrate a timing diagram of fundamental Avalon master read transfer and write transfer. A timing diagram of Avalon streaming R/W transfer is shown in Figure 3-14.

The major difference between the master and slave transfer is the data flows are in the reversed direction on each of them. Furthermore, although there are several flow control signals generated either by the Avalon slave peripheral including *dataavailable*,

readyfordata and the peripheral-controlled *waitrequest* or by the Avalon arbitrator (see Chapter 6), the Avalon bus module simply interprets them all as wait state request and only presents *waitrequest* to the Avalon master peripherals. So a golden rule for designing master peripherals to handle master transfers is: Assert all signals to initiate the bus transfer and then wait until the Avalon bus module deasserts *waitrequest*.

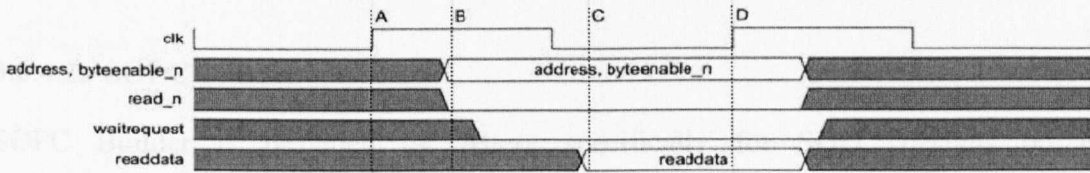


Figure 3-12 Fundamental master read transfer (From [35])

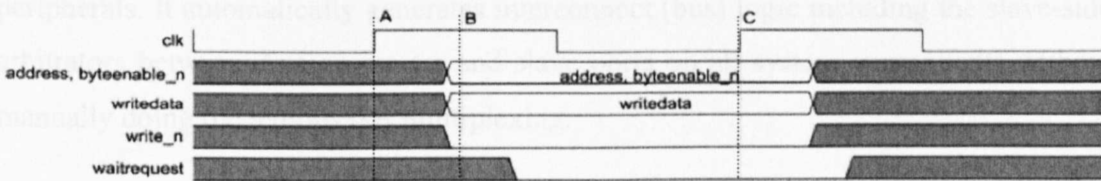


Figure 3-13 Fundamental master write transfer (From [35])

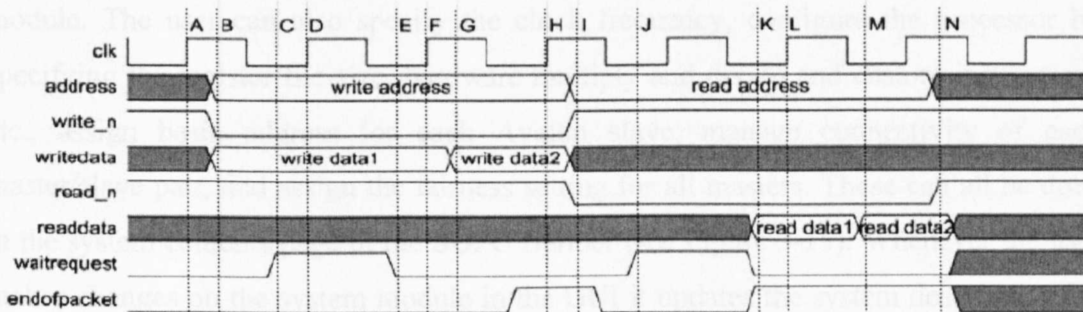


Figure 3-14 Streaming master R/W transfer (From [35])

3.6.4 Development tools overview

Four software design tools are provided by Altera to support the development of SOPC designs; they are Quartus II [54], SOPC Builder, programmer and software compiler. Along with using the simulation tool - ModelSim, the complete design flow described in section 3.3 can be undertaken.

3.6.4.1. Quartus II software

Quartus II provides a complete, multiplatform design environment that easily adapts to

specific design needs. It is a comprehensive environment for SOPC design. The Quartus II software includes solutions for all phases of FPGA and CPLD design including design coding, system synthesis, timing analysis, and gate-level simulations. It also provides several tools to help debugging the system design such as RTL viewer, Technology map viewer and SignalTap etc.

3.6.4.2. SOPC Builder

SOPC Builder is provided by Altera specifically for SOPC designs on its programmable logic devices. It is an integration tool for composing bus-based systems out of library components such as CPUs, memory interfaces, and custom-defined peripherals. It automatically generates interconnect (bus) logic including the slave-side arbitrators between Avalon master and slave ports on all system components without manually doing the tedious bus multiplexing.

The SOPC Builder has a GUI where the user can see all available IP components including the user-defined ones and a list of arranged components of the current system module. The user can also specify the clock frequency, configure the processor by specifying the register file size, hardware multiply and divide and custom instructions etc., assign basic address for each Avalon slave, manage connectivity of each master/slave pair, and assign the fairness setting for all masters. These can all be done in the system contents page of the SOPC Builder (see Figure 3-15). Whenever the user makes changes on the system module in the GUI it updates the system description file (PTF file) which stores the actual contents in the current system module at the same time. For example it stores the information of all IP components the top-level system module contains, and descriptions of the IO ports on each IP and the global connectivity. The description of the slave-side arbitrators is also contained in this file if there are connections from multiple masters to the same slave. This system PTF file is not a VHDL file but a record file of the current system that 'guides' the system generator program to generate the actual top-level VHDL file of the SOPC design.

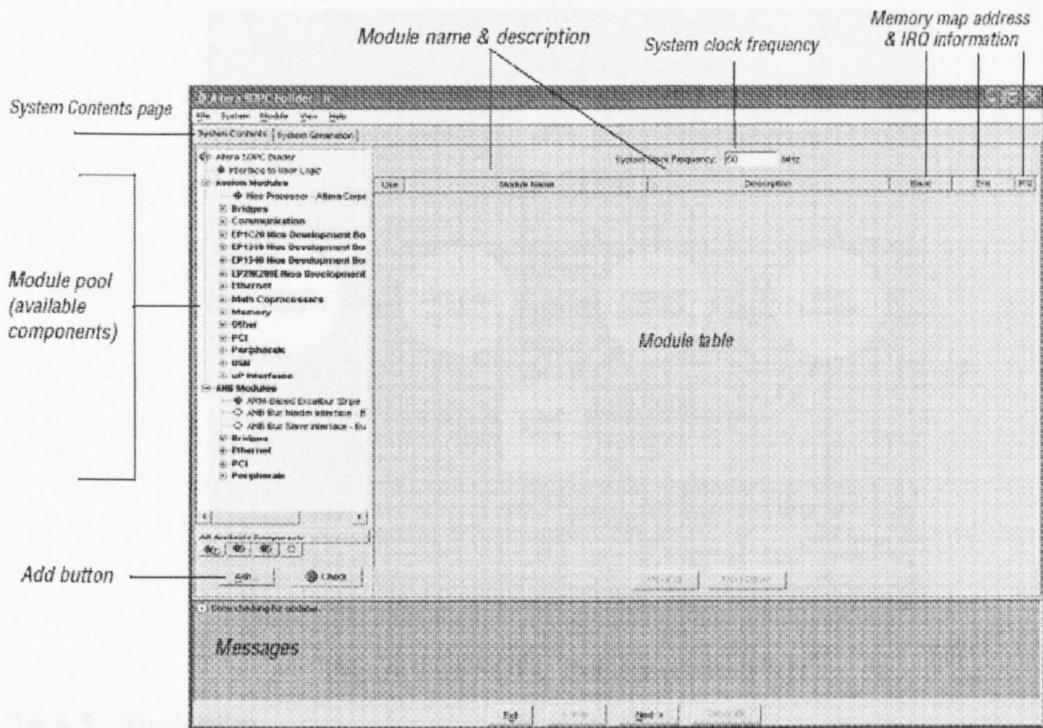


Figure 3-15 SOPC Builder system contents page (From [55])

Once the user finishes editing the system module either though the GUI or by directly modifying the system PTF file, the system generator program of the SOPC Builder will generate a top-level VHDL file describing the top module of the SOPC design associated with a software development kit (SDK) if the Nios processor is in the design. Figure 3-16 illustrates how the SOPC Builder works and what the output is in each step.

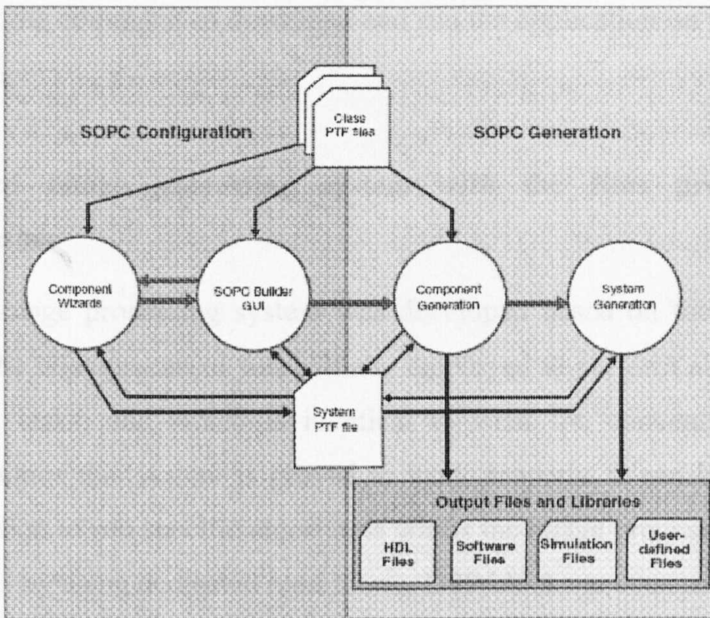


Figure 3-16 SOPC Builder (From [42])

3.6.4.3. ModelSim

As Quartus II doesn't support behaviour simulation, generally a third party simulation tool is used instead, a typical one is ModelSim. The ModelSim software is a dual-language simulator which supports designs containing Verilog HDL, VHDL, or both.

3.6.4.4. Programming tool

Once the system synthesis completes, the Quartus IIgenerates several programming files with various formats. They can be used to configure the device by different means via the programmer. The most common ways are via Joint Test Action Group (JTAG) or active serial mode [55]. A USB-Blaster download cable [56] is normally used to download the system design to the device for configuration.

3.6.4.5. Software download tool

The user can compile the software code in the Nios SDK Shell [57], which is a UNIX-like command shell that allows the user to build software, download software to the Nios development board [58], and run utilities and various test programs on the board. Once the software is compiled, the resultant binary code is stored in S-record format so that this complied code can be downloaded into Flash memory via the serial communication interface. Nios-build is the command to compile a source software file,

and Nios-run is the command to download and run the application on the development board [59].

3.6.5 Proposed image processing system with the Nios processor system architecture

The proposed image processing system was developed based on the Nios processor system where the Nios processor was initially employed to perform all system control and processing tasks, and which is identical to what the conventional embedded processors do. Once this system is proved to work properly, it can be applied into a specific application to run specific algorithms whilst the image processing performance can be improved by using dedicated hard fabric, DSP blocks or even more processors if resource is available. For the benefits of future development, this image processing system is able to work both in 8-bit grey level mode and 24-bit RGB mode, as these two modes are widely used in industrial vision based systems.

The performance goal of SIPS is to maximise the data bandwidth and transfer rate, and optimise the system architecture to allow the Nios processor to fetch data more efficiently. For the reasons of cost and better integration for further development, the main video function IPs were all developed from the bottom.

In the next chapters, detailed descriptions are given for SIPS in terms of the hardware and the soft system core design.

Chapter 4. The Nios Integrated Real-time Image Processing System - Hardware

Based on SOPC technology, a Nios integrated real-time image processing system was developed and evaluated on the Nios development kit. In the next two chapters, a detailed description of this system is presented with emphasis on the hardware & soft system core.

This chapter focuses on describing the hardware of this system. Section 4.1 gives an overview of the system hardware architecture; a description of each hardware component is presented in section 4.2, 4.3, 4.4, and 4.5 respectively. Section 4.6 introduces the hardware platform, where the whole system was developed. In the last section a summary of this chapter with an explanation for the reason for choosing the dedicated hardware, is presented.

4.1 Overview of the system hardware architecture

Following the guidelines described in Chapter 3 for implementing a real-time image processing system based on SOPC, the hardware architecture of SIPS was designed as shown in Figure 4-1 .

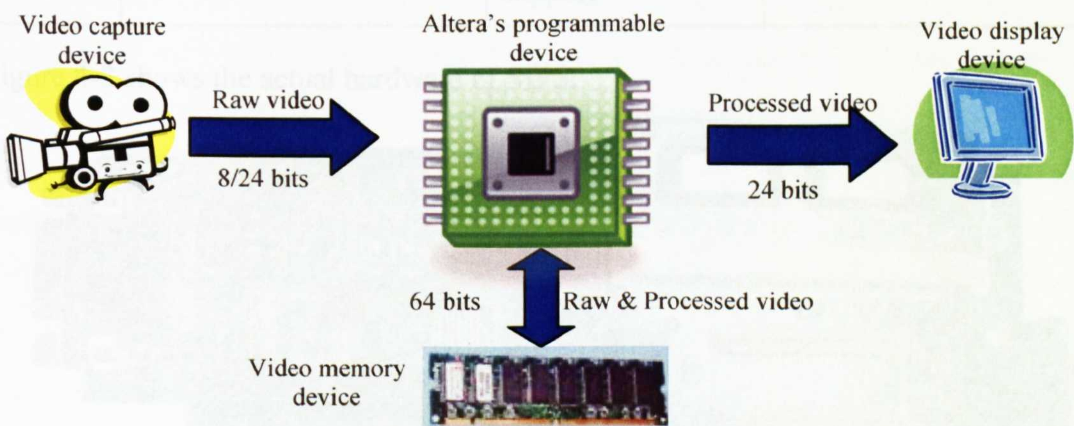


Figure 4-1 Block diagram of the hardware architecture of SIPS

Table 4-1 lists the device chosen for each function module and the reasons for its choice.

Table 4-1 Device table

Module	Device	Addition hardware	Reasons to choose
Video capture	CameraLink [60] CMOS camera, 8/24-bits	A custom designed camera interface card	High data rate, digitised interface, easy to implement, able to modify the camera operations and support windowing
Video display	640x480 Cathode ray tube (CRT) analog monitor (Video graphic arrays VGA format)	A VGA board contains a DAC	Cheap, easy to find and implement
Video memory	Synchronous dynamic random access memory (SDRAM), 144 pin	A simple SDRAM socket	Cheap, high data rate & wide data width
System control & video processing	Altera's Apex 20K programmable device [27]	Off-chip static random access memories (SRAMs), flash etc. for memory support	Cheap, support SOPC integration, meet the initial design requirements of SIPS, had immediate access

Figure 4-2 shows the actual hardware of SIPS.

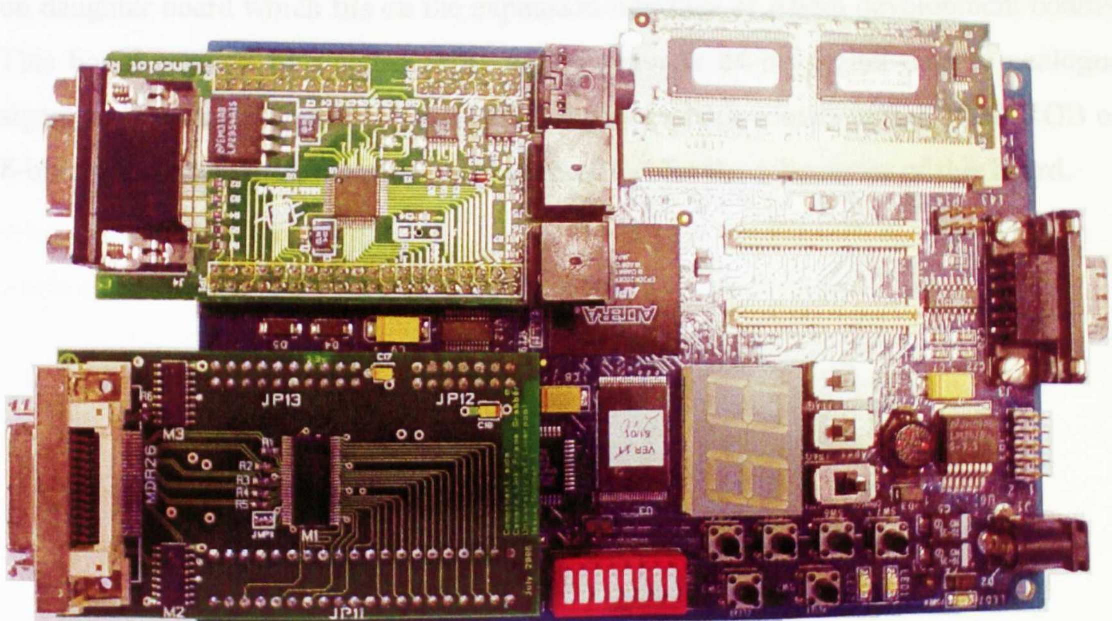


Figure 4-2 The Nios integrated real-time image processing system

4.2 Video display device & interface

The video display device is a typical CRT monitor with a resolution of 640x480. It supports the VGA mode. Video data & timing signals are sent to the display from the programmable device via a dedicated hardware interface – Lancelot VGA board [61]. Descriptions of this Lancelot VGA board with a VGA connector are presented in this section.

4.2.1 CRT monitor

The principle of how a CRT works is when the electrons emitted from an electron gun (a source of electrons) strikes the phosphorescent screen, light is emitted. A common CRT is colour (typically using three electron guns to produce red, green, and blue (RGB) images that, when combined, render a multicolour image). They come in a variety of display modes, including Colour Graphics Adapter (CGA), Video Graphics Array (VGA), Extended Graphics Array (XGA), and the high-definition Super Video Graphics Array (SVGA). For this system VGA mode was used as most CRTs support this mode.

4.2.2 Lancelot VGA board

The hardware interface which interfaces the programmable device is a dedicated video board called the Lancelot VGA board (Figure 4-3). The Lancelot VGA board is an add-on daughter board which fits on the expansion interface of Altera development boards. This board consists of a video DAC which converts 24-bit digital data to analogue signal so that the CRT monitor can display images which contain either 24-bit RGB or 8-bit monochrome data, per pixel. See Appendix A for the schematics of this board.

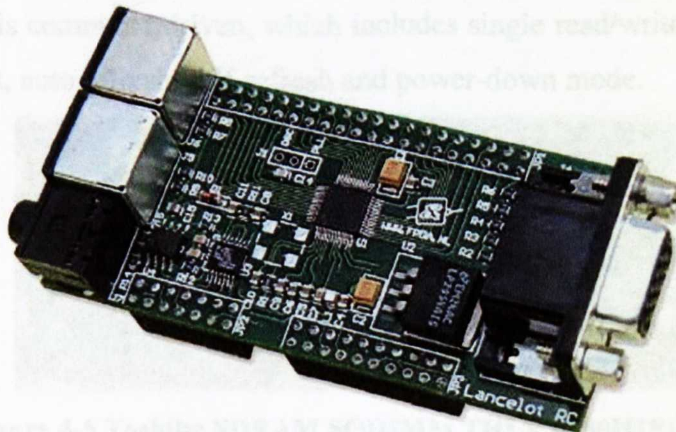


Figure 4-3 Lancelot VGA board

4.2.3 VGA connector

A VGA connector, as it is commonly known, is a three-row 15 pin connector. VGA connectors and their associated cabling are used solely to carry analog component RGBHV (red - green - blue - horizontal sync - vertical sync) video signals. Figure 4-4 & Table 4-2 illustrates the pin assignments on this VGA connector.

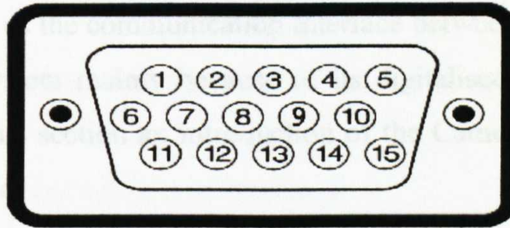


Figure 4-4 VGA connector

Table 4-2 VGA connector pin assignment

Pin number	Description	Pin number	Description	Pin number	Description
1	Video-Red	2	Video-Green	3	Video-Blue
4	No connect	5	GND	6	GND
7	GND	8	GND	9	No connect
10	GND	11	No connect	12	DDC data
13	H-sync	14	V-sync	15	DDC clock

Note: only the highlighted pins are driven by valid signals in this system.

4.3 Video memory device & interface

The main video data storage in this system is an industry standard single data rate synchronous dynamic random access memory (SDR SDRAM) model TOSHIBA THLY 6480H1FG-80 [62] (Figure 4-5). It is an 8,388,608-word by 64-bit SDRAM consisting of four TC59SM716FT/FTL DRAMs on a printed circuit board. This memory device is command driven, which includes single read/write, page read/write, mode register set, auto refresh, self-refresh and power-down mode.

Figure 4-5 Toshiba SDRAM SODIMM THLY 6480H1FG-80

To connect this memory device to the FPGA, a 144-pin small outline dual in-line

memory module (SODIMM) socket, compatible with standard SDRAM modules, was used on board. It is a 144-pin module so it supports 64-bit data transfer. See Appendix C for detailed pin assignments.

4.4 Video capture device & interface

CameraLink was chosen as the communication interface between the video camera and the image processing system mainly because of its digitalised interface and parallel video data transfer. In this section an introduction of the CameraLink camera and the interface card is presented.

4.4.1 CameraLink camera

The camera used for testing this system was a COHU 7800 series 1280x1024 CMOS progressive scan camera [63] (Figure 4-6). It supports an 8-bit monochrome video format. However, because this system was designed and can be configured to work with both 8-bit monochrome and 24-bit RGB video, the color version camera is also supported without changing any hardware. Table 4-3 lists the main features of this camera provide.

Table 4-3 COHU 7800 series camera specifications

Pixels	Up to 1280 x 1024
Pixel clock	5MHz, 10MHz, 20MHz, 40MHz
Frame rate	30 fps at full frame resolution; >30 fps with smaller regions of interest
Video output	8-bit (CameraLink format) data; horizontal (line) drive, vertical (frame) drive & pixel clock
Microprocessor control	Via video connector. All adjustments are software controlled through non-volatile registers; programmable logic, firmware can be upgraded via internal port, preset configurations supported.
Gain and offset control	Gain: 9 values in video amplifier Offset: 47 values for video black level reference setting

This camera has a built in microprocessor which not only provides digitised video data & enables output, but also supports a serial communication protocol to allow the camera to be modified externally, i.e. reducing the capture window size for higher frame rate.

Figure 4-6 COHU 7800 series 1280x1024 CMOS progressive scan camera

4.4.2 Camera interface card (custom designed)

Signals from the CameraLink camera are a number of pairs of Low-voltage differential signalling (LVDS) data which contain the video data and timing signals. The Camera interface card, which was designed with Cadstar by the author and manufactured by PCB Train [64], mainly consists of the LVDS receivers and transmitters to convert the LVDS data streams into parallel MultiVolt I/O data which the FPGA can accept. It also converts the control and configuration data driven from the FPGA into pairs of LVDS signals to either trigger the camera or configure the working mode of the camera. Figure 4-7 shows the component side of this interface card.

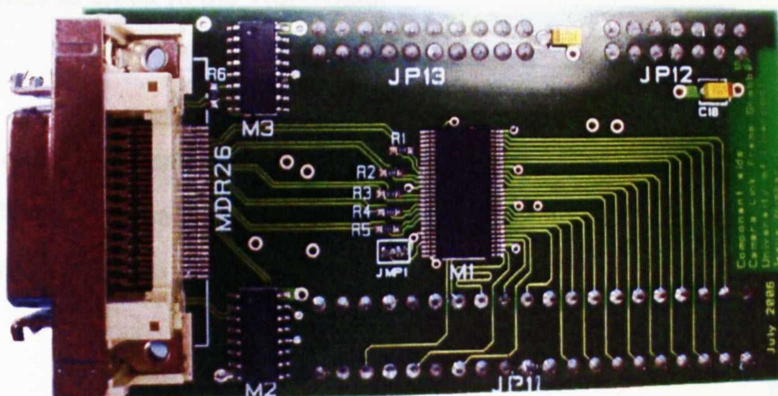


Figure 4-7 Camera interface card

The main components on this interface card are a 3M™ Mini D Ribbon (MDR) connector [65], a 28-bit LVDS receiver modelled DS90CR286 [66], an LVDS quad CMOS differential line receiver modeled DS90C032 [67], an LVDS quad CMOS differential line driver modeled DS90C031 [68] and some decoupling capacitors and resistors. Appendix A shows the schematics of this interface card. Two PCB diagrams of it are shown in Appendix B and the pin assignments shown in Appendix C. There are 3 headers on the board to allow this interface card to plug in the Nios development board. Figure 4-8 illustrates a block diagram of the schematic of this interface card.

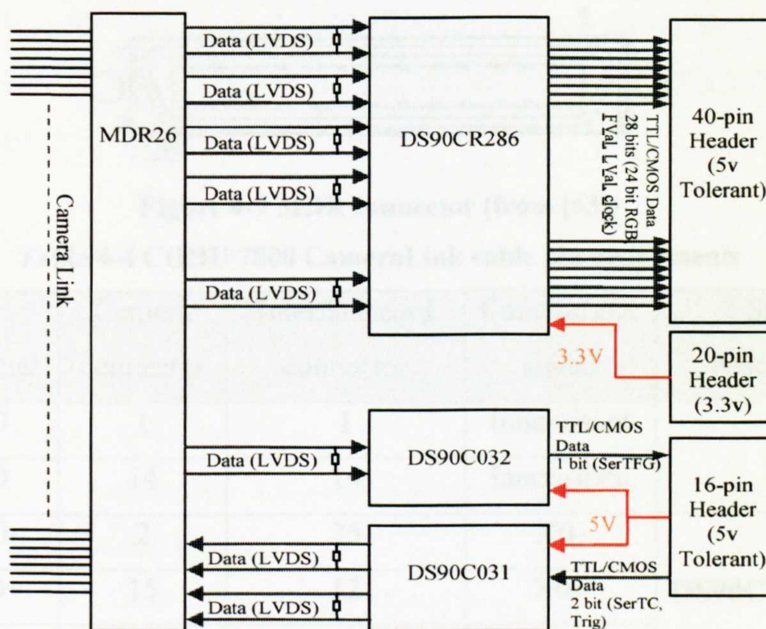


Figure 4-8 Block diagram of the camera interface card

DS90CR286

This 28 bit LVDS receiver converts four pairs of LVDS signals into 28 bit 3.3 volt input data to the FPGA which includes 24-bit RGB, horizontal line drive signal *LVAL* and vertical frame drive signal *FVAL*. It has an onchip PLL to multiply the LVDS clock by 7 times to clock in the video data.

DS90C031

This LVDS quad differential line driver converts two 5-volt output signals from the FPGA, which are external camera trigger signal *Trig* and camera serial control signal *SerTC* from the FPGA, to 2 pairs of LVDS signals and transmits them to the camera.

DS90C032

This LVDS quad differential line receiver receives and converts one pair of LVDS signals from the camera, which is the camera status information, to a 5-volt input signal *SerTFG* to the FPGA.

MDR connector

This MDR Connector, which exists on both ends of the CameraLink cable, meets the stringent demands of reliable high-speed differential signaling applications [63]. Figure 4-9 and Table 4-4 describes the camera signals and how the interface card connector should match that.

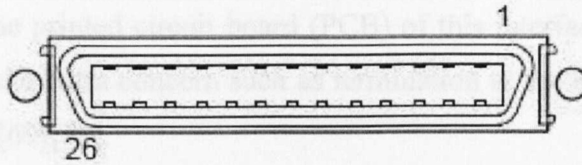


Figure 4-9 MDR connector (from [63])

Table 4-4 COHU 7800 CameraLink cable pin assignments

COHU camera signal	Camera connector	Interface card connector	CameraLink signal	Signal descriptions
DGNND	1	1	Inner shied	--
DGNND	14	14	Inner shied	
TXOUT0-	2	25	X0-	
TXOUT0+	15	12	X0+	**DS90CR286**
TXOUT1-	3	24	X1-	24 bits video data
TXOUT1+	16	11	X1+	and 4 video
TXOUT2-	4	23	X2-	enables include
TXOUT2+	17	10	X2+	*FVAL, LVAL*
TXOUT3-	6	21	X3-	
TXOUT3+	19	8	X3+	
TXCLK-	5	22	Xclk-	**DS90CR286**
TXCLK+	18	9	Xclk+	Video pixel clock
RX+	7	20	SerTC+	**DS90C031**
RX-	20	7	SerTC-	*SerTC*
TX-	8	19	SerTFG-	**DS90C032**
TX+	21	6	SerTFG+	*SerTFG*
-	9	18	CC1-	
-	22	5	CC1+	--
-	10	17	CC2-	
-	23	4	CC2+	
TRIG-	11	16	CC3-	**DS90C031**
TRIG+	24	3	CC3+	*Trig*
NC	12	15	CC4-	
NC	25	2	CC4+	
GDNDD	13	13	Inner shied	--
GDNDD	26	26	Inner shied	

When designing the printed circuit board (PCB) of this interface card, there are a few things needed to take extra concern such as termination at the RX LVDS, inputs traces and power supply [69][70].

4.5 Altera's Apex device

The Apex 20K programmable logic device family is industry's first PLD incorporating SOPC integration. Reference [27] identifies the main features of the Apex 20K programmable logic device family as,

- MultiCore™ architecture integrating look-up table (LUT) logic, product-term logic, and embedded memory
- LUT logic used for register-intensive functions
- Embedded system block (ESB) used to implement memory functions, including first-in first-out (FIFO) buffers, dual-port RAM, and content-addressable memory (CAM)
- ESB implementation of product-term logic used for combinatorial-intensive functions
- High density: 30,000 to 1.5 million typical gates, up to 51,840 logic elements (LEs) Up to 442,368 RAM bits that can be used without reducing available logic, and up to 3,456 product-term-based macrocells
- Flexible clock management circuitry with up to four phase-locked loops (PLLs)

By integrating the SOPC design on this programmable device, this device can control all off-chip peripherals and fulfill various image processing algorithms.

The one chosen to evaluate the SOPC design was Apex EP20K200E484-2x with the typical contents shown in Table 4-5.

Table 4-5 Apex 20K200E device features (From [27])

Maximum system gates	526,000
Typical gates	211,000
LEs	8,320
ESBs	52
Maximum RAM bits	106,496
Maximum macrocells	832
Maximum user I/O pins	382
PLLs	2

4.6 Nios development board

The Nios development board provides a hardware platform to immediately start developing systems based on Altera Apex devices (see Figure 4-10). Reference [58] lists the main features of the Nios development board as following,

- 1 Mbyte (512 K x 16-bit) of flash memory pre-configured with the 32-bit Nios reference design and software
- 256 Kbytes of SRAM (in two 64 K x 16-bit chips)
- On-board logic for configuring Apex device from flash memory
- 3.3-V expansion/prototype headers (access to 40 user I/Os)
- 5-V-tolerant expansion/prototype headers (the Lancelot VGA board and the CameraLink interface card are fit in here)
- Small outline DIMM (SODIMM) socket, compatible with standard SDRAM modules (it's where the video memory device located)
- Two IEEE-1386 Peripheral Component Interconnect (PCI) [71] mezzanine connectors
- One RS-232 serial connector
- One user-definable 8-bit dual in-line package (DIP) switch block
- Four user-definable push-button switches
- Dual 7-segment light-emitting diode (LED) display
- Two user-controllable LEDs
- JTAG connector for ByteBlaster™ II and MasterBlaster™ download cables
- Oscillator and zero-skew clock distribution circuitry

- Power-on reset circuitry
- Power-supply circuitry (Input: 9-V unregulated, center-negative)

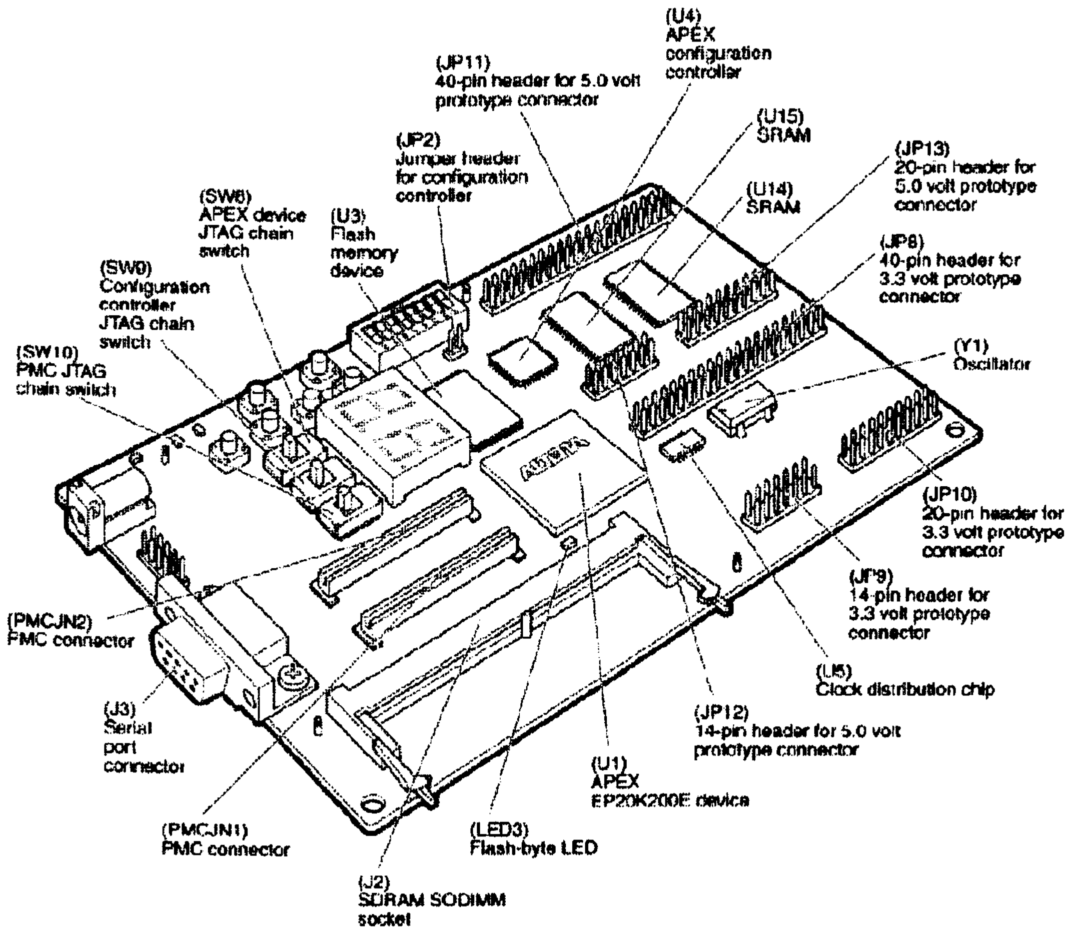


Figure 4-10 Nios development board (From: [58])

4.7 Summary

This chapter has given an overview of the major hardware components used in SIPS including the programmable device and other off-chip peripherals. The video data storage device is an off-chip 144-pin SDRAM. The video display device is a 640x480 VGA mode CRT monitor whilst an additional VGA display interface card (Lancelot board) was used to convert the digital video data into analogue signals so that they can be displayed on the analog CRT monitor. The video capture device is a CMOS camera with CameraLink interface; an off-chip on-board camera interface card was custom designed and fabricated to convert the LVDS video data streams into parallel MultiVolt I/O data which the FPGA accepts. An Altera's Apex EP20K200E484-2x FPGA was chosen to implement the main SOPC core design. The whole system was developed

based on the Nios development platform which contains the dedicated programmable device, expansion connectors to allow the video interface boards easily plugged in, and some other hardware components such as flash memory, SRAMs, oscillator and reset circuitry, which were useful to set up the system and verify the design conveniently.

When making decisions for choosing the video components, the main considerations were the cost, the difficulty of how they can be implemented, and most importantly whether they satisfy the system requirements such as speed, capacity etc. A good example of cost and popularity is the VGA mode CRT monitors. They are cheap and easy to find. Although the display device was a CRT monitor, Liquid crystal display (LCD) is also supported as long as it has the VGA mode, but certainly the price of LCD is higher than CRT. The video memory storage device - SDRAM, although it needs longer access time than SRAM or onchip memory, it is cheaper and has more capacities than the other choices. SRAM is fast and there are two available on the development board, but the total memory of them is 256 Kbytes, which is not enough for the SIPS which requires at least 640x480x8x3 ≈ 0.879 Mbytes for 8-bit monochrome mode and 640x480x24x3 ≈ 2.636 Mbytes for 24-bit RGB mode (see section 5.7.1). Furthermore they have partly been used for the CPU program and data memory. The on-chip memory is even less (13 Kbytes). Flash memory device could be an option but due to its complexity of erasing and programming operations it's not ideal for real time processing.

The VGA interface card – Lancelot board, although it wasn't made for commercial usage, it has been widely adapted in many imaging system developments such as [72], [73] and [74]. The main reasons of choosing it for evaluation were it supports VGA mode and is compatible with the hardware development platform.

The hardest part to decide was the camera interface. There are many vision communication interface standard which can be chosen such as CameraLink, USB, Institute of Electrical and Electronics Engineers (IEEE) 1394 (Firewire) [75] etc. USB is an ideal choice because of its broad usage in commercial market and low price, and actually there is a USB1.1 host slave IP core [76] which is available from OpenCore.Org. However, the maximum bandwidth of USB1.1 [77] is insufficient to support smooth video transfer in this system. Supposed it needs to capture ten frames

per second, and the pixel format is in 8-bit monochrome. Theoretically it requires 10x8x640x480 = 23.437 Mb/s which is twice over the capacity of the USB1.1 transfer rate which is 12 Mb/s. Although USB 2.0 [78] and Firewire are fast and provide high bandwidth, there is no exiting IP components which can be used for them while it would be time-consuming to develop. However, CameraLink, which provides digitised interface, high transfer rate (can be up to 2.38 Gbits/s [60]), and most importantly it's specialised for vision systems, so it is easy to be built into this image processing system. Furthermore, the serial control interface allows the camera to be configured and controlled, i.e. by reducing the capture window size to obtain a higher frame rate, which is not the function that general USB and Firewire cameras can offer. The only thing needed to do was to construct a camera interface card to allow the camera to convert the LVDS streams to MultiVolt I/Os so that the FPGA accepts and vice versa. This camera interface card fits on the expansion interface of the Nios development board.

There are various FPGAs which could be chosen to evaluate the SIPS core. The reason for choosing the Apex 20K was it had immediate availability to use as well as the Nios development platform when this research commenced. As shown in the experimental results the number of I/Os and gates requirements are all met although the on-chip memory is not enough to support the system in 24-bit RGB mode. However, the flexibility of this system can always make it easy to be implemented in a better FPGA. Chapter 7 will be giving some performance analysis on several FPGAs which could probably be used in the future for implementation of the system in 24-bit RGB mode and optimisation.

Although the SIPS core is able to be generated into an 8-bit monochrome or 24-bit RGB system (see Chapter 5), neither of these configurations require changes on the hardware because it has been designed to accommodate the maximum capacity.

Chapter 5. The Nios Integrated Real-time Image Processing System – Soft System Core

5.1 Overview of the system core architecture

In Chapter 4, a detailed hardware description of this image processing system was presented. As described, the system core of this image processing system was synthesised and evaluated on the Altera's programmable device Apex 20K200E. This chapter therefore focuses on describing this soft system core in details. Based on the system level design methodology and the Nios processor system architecture described in Chapter 3, all off-chip peripheral controllers were implemented as separate IP modules and wrapped with the Avalon bus interface so that they can be integrated together automatically with the Avalon bus module by the SOPC Builder. All of the main IPs used in SIPS are listed and described below.

- Nios processor
- Flash controller
- SRAM controller
- Timer [79]
- UART controller [80]
- Video memory controller
- Video capture controller
- Video display controller
- Cache

The Nios processor, Flash controller, SRAM controller, Timer and UART controller are provided by Altera as standard IP components, while the other IP components including the **video memory controller**, **video capture controller**, **video display controller** and **Cache** are designed by the author.

The **Nios processor**, as described in Chapter 3, is the main computing engine for image processing. It also provides the overall controls for the other system peripherals.

The **video memory controller** provides interface controls for the memory device

SDRAM to buffer all incoming video streams and also stores the processed video data ready to be displayed.

The **video capture controller** decodes the video enable signals driven from the CameraLink camera via the interface card and uploads the raw video streams to the memory device.

The **video display controller** drives the Lancelot VGA board with relevant VGA timing signals and the video data to get the video displayed on the CRT monitor.

The **Cache** acts like a 'bridge' between the Nios processor and the main memory for more efficient data transaction.

The **Flash controller** drives the off-chip flash device which can be used as general-purpose readable memory and non-volatile storage, or to hold the Apex device configuration file which is used by the configuration controller to load the Apex device at power-up.

The **SRAM controller** provides control to the dual off-chip SRAM chips, which can be used by the Nios processor for storing the program data and instruction in zero-wait-state.

The **Timer** module is a 32-bit interval timer which is useful to measure the specific software timing for testing purpose.

The **UART** is used to provide controls for on-board serial connectors typically used for host communication with a desktop workstation. This IP module is also used to control the CameraLink serial communication.

The video display and capture controller can be configured individually to allow the system to work in either 8-bit monochrome mode or 24-bit RGB mode. All of these custom IPs can be programmed individually to meet different system requirements such as changing the resolution, modifying the burst transfer length etc. They are completely reusable and can be fitted in other types of programmable devices.

Figure 5-1 illustrates the architecture of this system core and the communication flows with the external hardware.

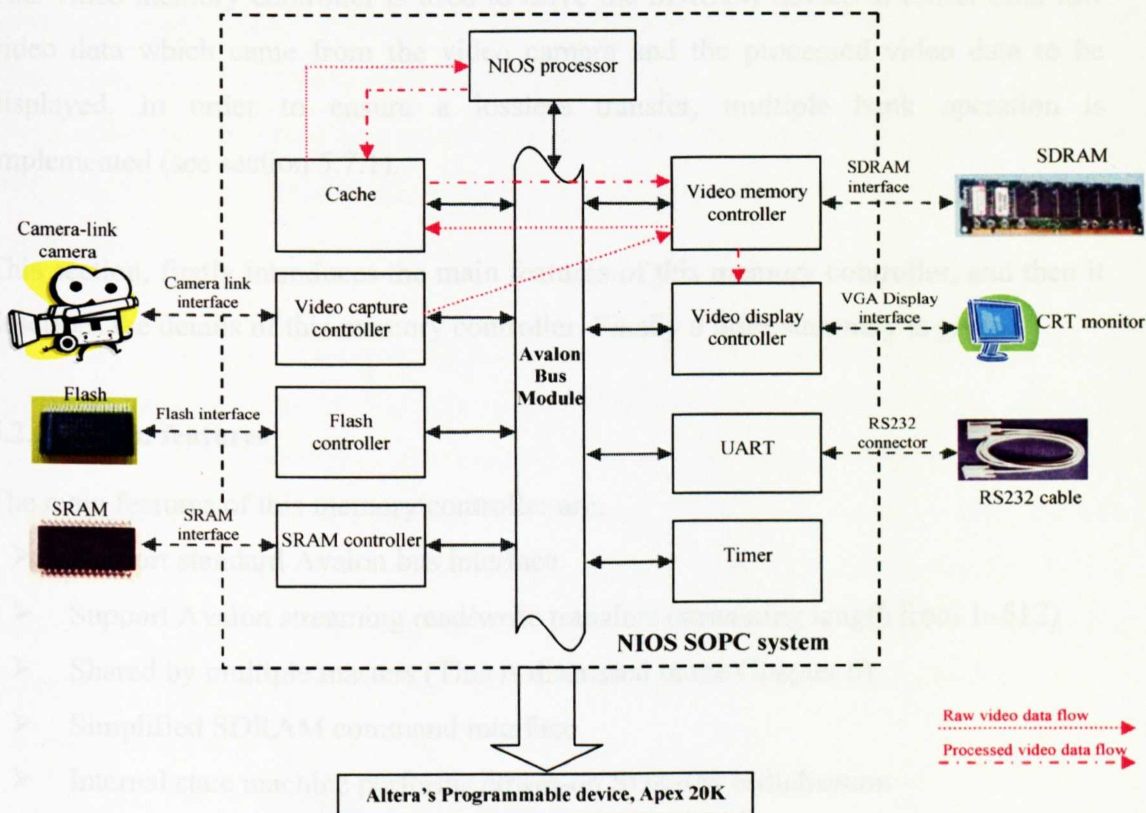


Figure 5-1 Top level block diagram of SIPS core

In the next few sections, detailed descriptions are given to the video memory controller, display controller, capture controller and Cache. Following that descriptions of all clocks used in this system are given. Finally it presents some typical design issues including the multiple bank operations, the use of double line buffer mode in both the video display and capture controller, and the synchronisation of multiple clock domains.

5.2 Video memory controller

This video memory controller is used to drive the SDRAM device to buffer both raw video data which came from the video camera and the processed video data to be displayed. In order to ensure a lossless transfer, multiple bank operation is implemented (see section 5.7.1).

This section, firstly introduces the main features of this memory controller, and then it describes the details of this memory controller. Finally a brief summary is given.

5.2.1 Main features

The main features of this memory controller are,

- Support standard Avalon bus interface
- Support Avalon streaming read/write transfers (streaming length from 1~512)
- Shared by multiple masters (This is discussed in the Chapter 6)
- Simplified SDRAM command interface
- Internal state machine performs power-on SDRAM initialisation
- Full page operation mode only, burst read burst write
- 8 bytes (64 bits) data width, contain two pixels in 24-bit RGB mode or eight pixels in 8-bit monochrome mode
- One video line mapped into one SDRAM row
- Programmable CAS latency of 2 or 3 clock cycles
- Perform auto refresh internally, 4096 Refresh cycles / 64ms
- Support over 100MHz working frequency

5.2.2 Description of the video memory controller

This video memory controller was designed to program the memory device to perform page read, page write, mode register set and internal auto-refresh operations. The SDRAM is controlled by bus commands [81]. There are several command functions such as Bank Activate command, Bank Precharge command, Precharge All command, Write/Read command, Burst Stop command, Mode Register Set command [62]. However, it would slow down the system operations if all of these commands were inilialised by a separate Avalon transfer. Therefore this video memory controller has simplified this and only supports **Avalon streaming read**, **Avalon streaming write** &

Avalon mode register set operations. These operations are fulfilled on the SDRAM device by implementing a sequence of SDRAM commands. For example, when an Avalon streaming read request is initialised, the memory controller firstly sends out a Bank Activate command to open a specified row in a specified bank, and then a Page Read command is issued to continuously read data from the specific row, finally this Full Page Read is terminated by a Precharge command and the whole operation completes.

Figure 5-2 shows the top-level block diagram of this video memory controller. There are three sub-modules of this soft core which are the Avalon interface, SDRAM controller and SDRAM data path module. The Avalon interface module handles the data transfers with the Avalon bus and decodes the address, to drive the SDRAM controller to generate proper control signals and data to the SDRAM device for reading and writing. The SDRAM controller generates the actual SDRAM commands to the memory device. The SDRAM data path block handles the video data multiplexing.

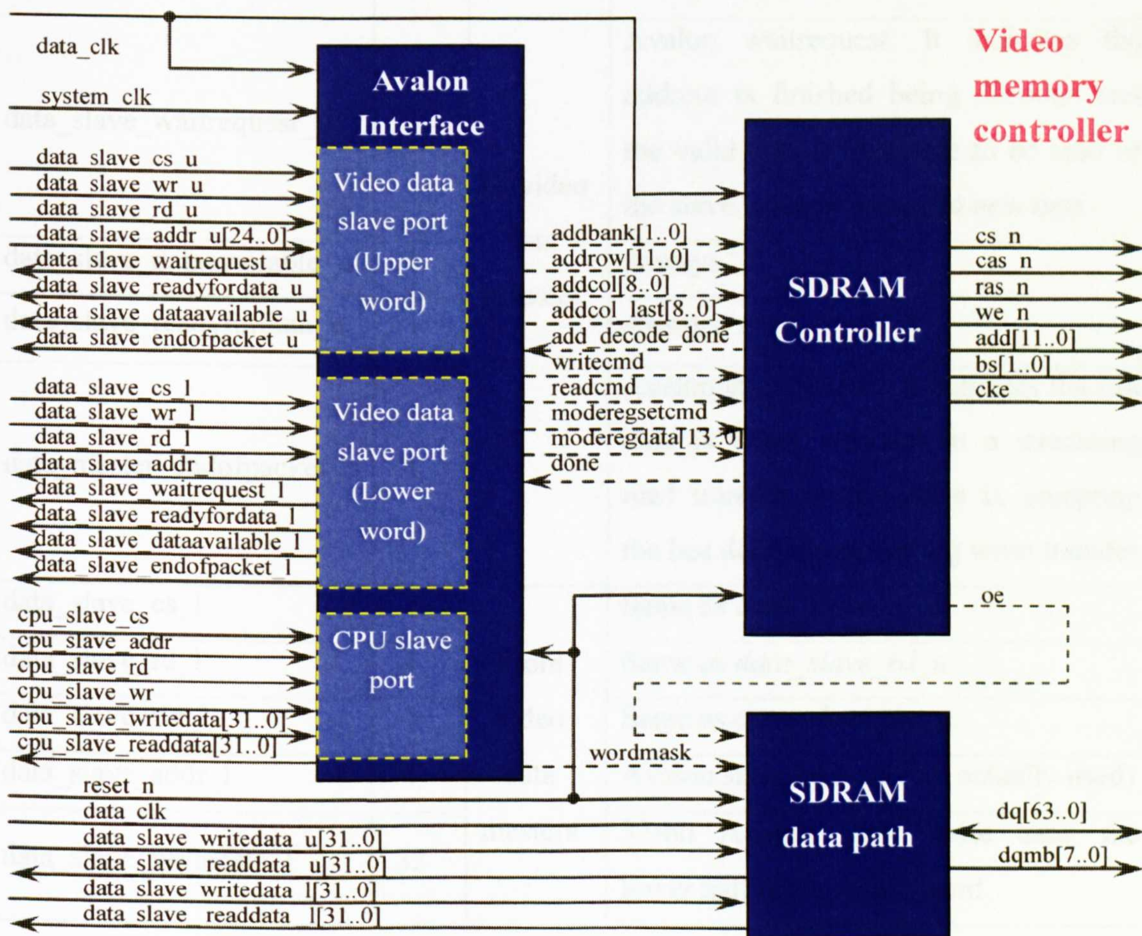


Figure 5-2 Block diagram of the video memory controller

Table 5-1 lists the top level I/Os of the memory controller.

Table 5-1 Video memory controller top level signals

Signal	Width	Direction	Description
reset_n	1	Global	Global reset, active low
data_clk	1	From clock generator	Video data transfer clock
system_clk	1		System clock, drives Avalon transfers inilialised by the Nios processor
data_slave_cs_u	1	From video data masters	Avalon slave chip enable, active high
data_slave_rd_u	1		Avalon master read enable, active high
data_slave_wr_u	1		Avalon master write enable, active high
data_slave_addr_u	25		Avalon slave address
data_slave_writedata_u	32		32-bit Avalon slave write data, the upper half of the video word
data_slave_readdata_u	32	To video data masters	32-bit Avalon slave read data, the upper half of the video word
data_slave_waitrequest_u	1		Avalon waitrequest. It indicates the address is finished being decoded and the valid data is available to be read or the slave is ready to accept new data
data_slave_dataavailable_u	1		Always '1'
data_slave_readyfordata_u	1		Always '1'
data_master_endofpacket_u	1		Avalon endofpacket. It indicates the last data is being returned in a streaming read transfer or the slave is accepting the last data in a streaming write transfer
data_slave_cs_l	1	From video data masters	Same as *data_slave_cs_u*
data_slave_rd_l	1		Same as *data_slave_rd_u*
data_slave_wr_l	1		Same as *data_slave_wr_u*
data_slave_addr_l	1		Avalon slave address (not actually used)
data_slave_writedata_l	32		32-bit Avalon slave write data, the lower half of the video word

data_slave_readdata_l	32	To video data masters	32-bit Avalon slave read data, the lower half of the video word
data_slave_waitrequest_l	1		Same as *data_slave_waitrequest_u*
data_slave_dataavailable_l	1		Always '1'
data_slave_readyfordata_l	1		Always '1'
data_master_endofpacket_l	1		Same as *data_master_endofpacket_u*
cpu_slave_cs	1	From Nios processor	Avalon slave chip select, active high
cpu_slave_rd			Avalon slave read enable, active high
cpu_slave_wr	1		Avalon slave write enable, active high
cpu_save_addr	1		Avalon slave address
cpu_slave_writedata	32		Avalon slave write data, contains the SDRAM mode register data
cpu_slave_readdata	32	To Nios processor	Avalon slave read data, contains the SDRAM mode register data
cs_n	1	SDRAM I/Os	Chip select, active low
cas_n	1		Column address strobe, active low
ras_n	1		Row address strobe, active low
we_n	1		Write enable, active low
add	12		SDRAM address input
bs	2		SDRAM bank select
cke	1		Clock enable, controls the clock activation and deactivation, active high
dq	64		Multiplexed pins for data output and input
dqmb	8		Output Disable/Write Mask, In write cycle, sampling *dqmb* high will block the write operation with zero latency, active high

Table 5-2 lists the interconnection signals between those three sub modules.

Table 5-2 Internal signals of the video memory controller

Signal	Width	Description
addbank	2	SDRAM bank address decoded from *data_slave_addr_u*
addrow	12	SDRAM row address decoded from *data_slave_addr_u*
addcol	9	SDRAM column address decoded from *data_slave_addr_u*
addcol_last	9	SDRAM end column address
add_decode_done	1	Address decoding stage done, data cycle starts from the next clock cycle
writecmd	1	Avalon streaming write request
readcmd	1	Avalon streaming read request
moderegsetcmd	1	Mode register set request
moderegdata	14	Mode register data to be set
done	1	Indicates the last valid data is returning or last data is being written into the SDRAM, Avalon requests complete
wordmask	2	Video word mask
oe	1	Output enable, high during SDRAM write, low during read

5.2.2.1. Avalon interface

The Avalon interface module provides I/O controls over the Avalon bus interface signals of the video memory controller to transmit and receive video data to/from the Avalon streaming masters (video capture masters, video display masters, Cache masters), and also to receive mode register data from the Nios processor.

The Avalon interface module contains three Avalon slave ports. Two of them are Avalon streaming slaves and the third is a simple Avalon slave.

CPU slave port – simple Avalon slave port

When there is an Avalon write request initialised from the Nios processor (a fundamental Avalon write) to this slave signal *moderegsetcmd* is asserted to indicate the mode register needed to be reset. The mode register data is contained in the signal *cpu_slave_writedata*. The actual Mode register command is implemented in the SDRAM controller module.

Video data slave ports – streaming Avalon slave ports

These two Avalon slave ports are streaming Avalon slave ports which are mastered by multiple Avalon streaming master ports to transfer video data through the SDRAM device to the rest of this system. The streaming transfer length can be from 1 to the maximum column size of the SDRAM device which is 512.

The data width of the SDRAM device is 64; however the maximum data width of the Avalon bus port is 32. Therefore two Avalon master-slave pairs are ideally required to transfer a 64-bit word in a single data clock cycle to maximise the system efficiency. Each of them handles 32 bits.

In order to save system memory space, the SDRAM address is only carried in one master-slave pair. In this system the one which handles the upper video word was chosen to do this. So only bus *data_slave_addr_u* needs to be decoded in the memory controller. This address contains not only the SDRAM bank (*addbank*), row (*addrow*) and the start column address (*addcol*) where the burst transfer starts, but the end column address (*addcol_last*) where the streaming transfer should terminate. So by comparing the start column address and the end column address the burst length can be obtained. Table 5-3 shows details of the address bus *data_slave_addr_u*.

Table 5-3 SDRAM address mapping

Address bit / operation	24	23	22	21...12	11	10	9......3	2	1	0
Write	*addbank*		*addrow*			*addcol/addcol_last*			*wordmask*	
Read	*addbank*		*addrow*			*addcol/addcol_last*			Don't care	

Note: *addcol* and *addcol_last* are decoded in two separate states in the address decoding stage

Whenever there is an Avalon R/W request initialised by an Avalon streaming master, the memory controller immediately decodes the bank, row and start column addresses, and presents streaming read command *readcmd* or streaming write command *writecmd* to the SDRAM controller. It also asserts the waitrequest *data_slave_waitrequest_x* (x=u and l) signal to stall the master until the address is decoded. Once the address is

decoded by checking if the signal *add_decode_done* is asserted, it deasserts the waitrequest signal and waits for the end of the transfer by checking the signal *done*.

Figure 5-3 shows an ASM chart of how the Avalon interface module handles Avalon streaming transfers involved with the two video data slaves.

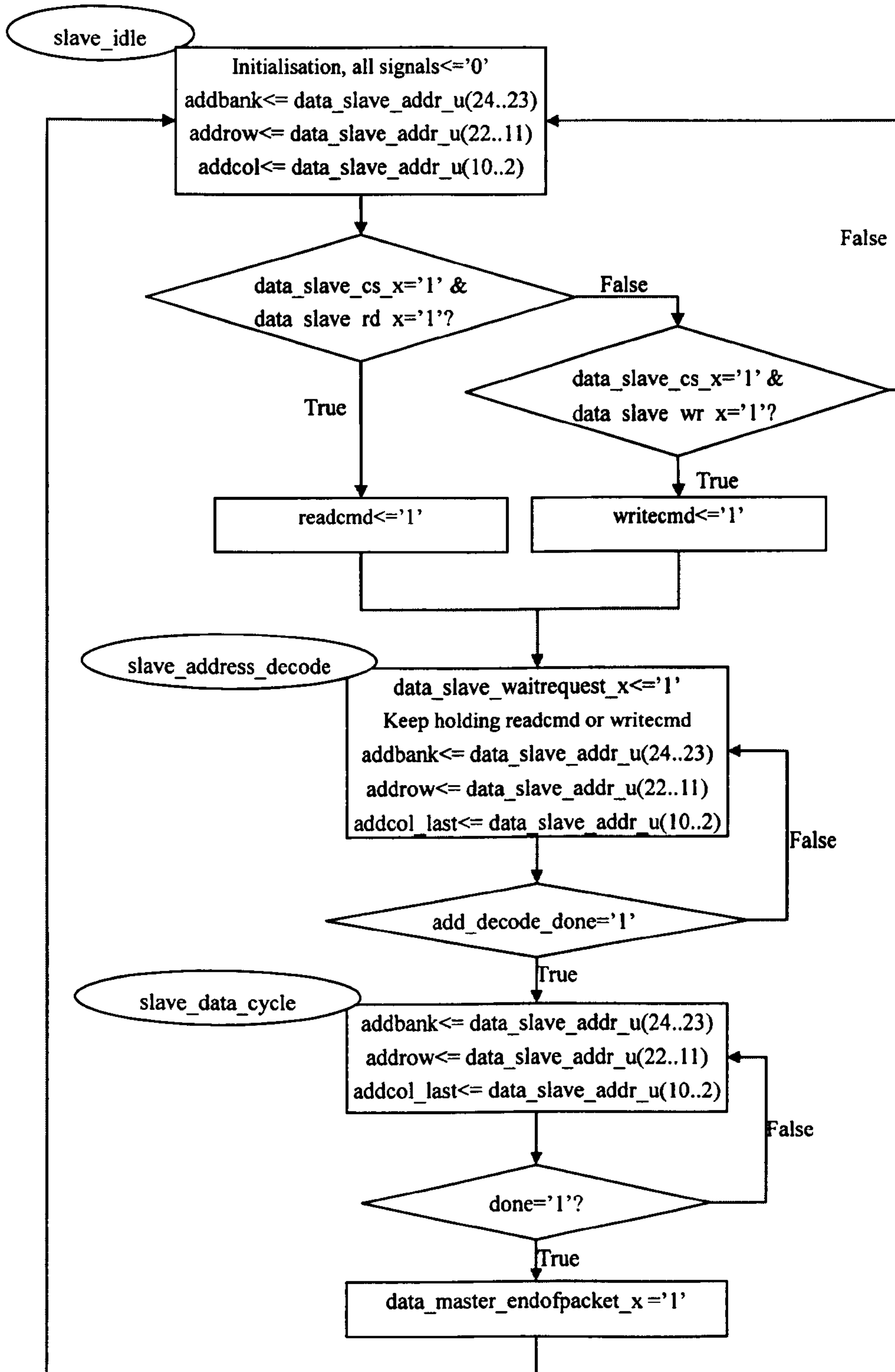


Figure 5-3 ASM chart of the Avalon streaming slave transfer

5.2.2.2. SDRAM controller

The SDRAM controller responds to the three Avalon requests which are the mode register set, Avalon streaming read and Avalon streaming write request from the Avalon interface module and implements a sequence of SDRAM operations to complete these requests. It also implements a power-up sequence to initialise the SDRAM device. Figure 5-4 shows the interconnection of the main components in this module.

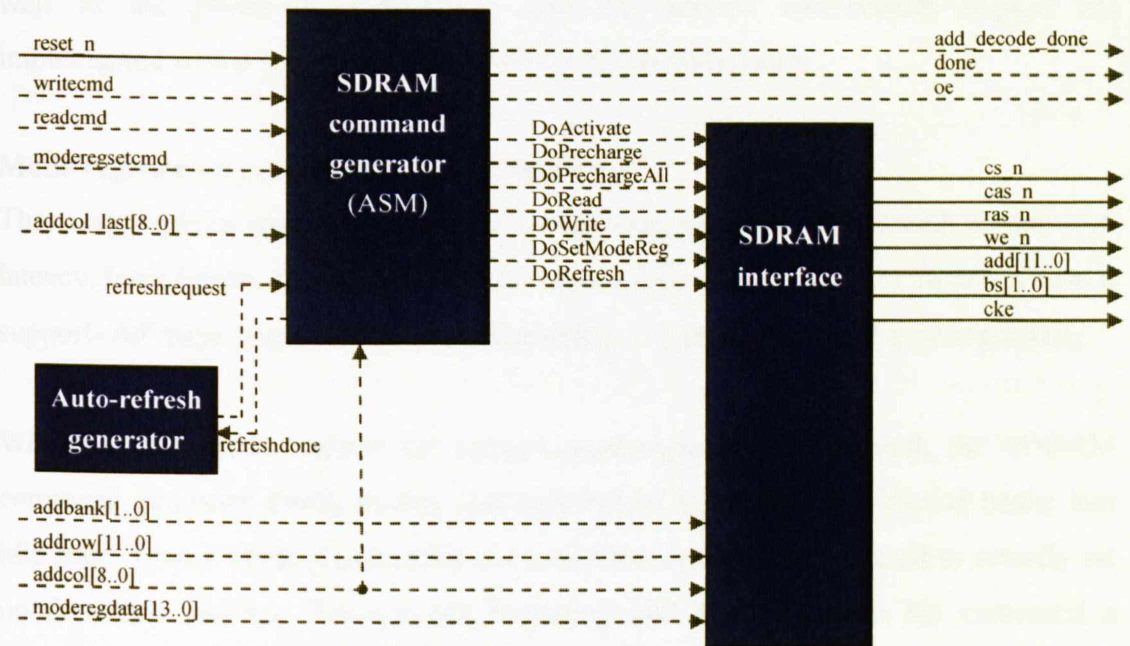


Figure 5-4 Block diagram of the SDRAM controller

There are three main blocks in the SDRAM controller and they are the SDRAM command generator, SDRAM interface and auto-refresh generator. The SDRAM command generator basically generates several SDRAM command signals to fulfil the three Avalon requests as well as the power-up initialisation and the internal auto-refresh function. These SDRAM command signals are *DoActivate*, *DoPrecharge*, *DoPrechargeAll*, *DoRead*, *DoWrite*, *DoSetModeReg* and *DoRefresh*. This command generator also calculates the burst length by comparing *addcol* and *addcol_last* and generates the signal *add_decode_done* and *done* to the Avalon interface module.

The SDRAM interface block interprets the command signals from the command generator and drives the SDRAM control interface signals *cs_n*, *ras_n*, *cas_n* and *we_n* with different combinations of values to implement these command requests. For example, when *DoAcivate* is asserted, control signal *cs_n*, *ras_n* are driven low and the

others are held high to activate an SDRAM row addressed at *add[11..0]* in bank *bs[1..0]*. Appendix D shows a truth table for all SDRAM operation commands.
The auto-refresh generator is a counter which basically generates 4096 auto-refresh requests in every 64ms. The actual Auto-Refresh command is asserted by the SDRAM command generator.

The next few paragraphs mainly focus on explaining how the three Avalon requests as well as the power-up initialisation and the internal auto-refresh request are implemented with a sequence of SDRAM operation commands.

Mode register set command – ‘moderegsetcmd’

The mode register stores the operation information of the SDRAM such as the CAS latency, burst length, burst type, and write burst mode (see Figure 5-5). In this system it supports full page burst length, sequential addressing mode and burst read burst write.

Whenever the mode register set request *moderegsetcmd* is asserted, the SDRAM command generator firstly asserts *DoPrechargeAll* to switch all SDRAM banks into idle state. Then a Mode Register Set command *DoSetModeReg* is issued to actually set up the mode register. Between the Precharge and Mode Register Set command a minimum time gap t_{RP} (Precharge to Active command period) must be met. Also after the Mode Register Set command issued a minimum time period of t_{RSC} must be elapsed before performing the next command. These timing are implemented in the state machine as wait states. Figure 5-5 illustrates the Mode Register Set timing cycle.

Table 5-4 summarises the SDRAM operations which must be performed to complete this Mode Register Set cycle with the corresponding SDRAM control signal status.

Table 5-4 Mode register set command

Command	SDRAM command	*bs01*	*A10*	*A0-A9 A11*	*cs_n*	*ras_n*	*ras_n*	*we_n*	Descriptions
Mode register set	*DoPrecharge*	X	H	X	L	L	H	L	Precharge all banks
	DoSetModeReg	V	V	V	L	L	L	L	Set mode register

Note: V = valid, H = high level, L = low level, X = don’t care

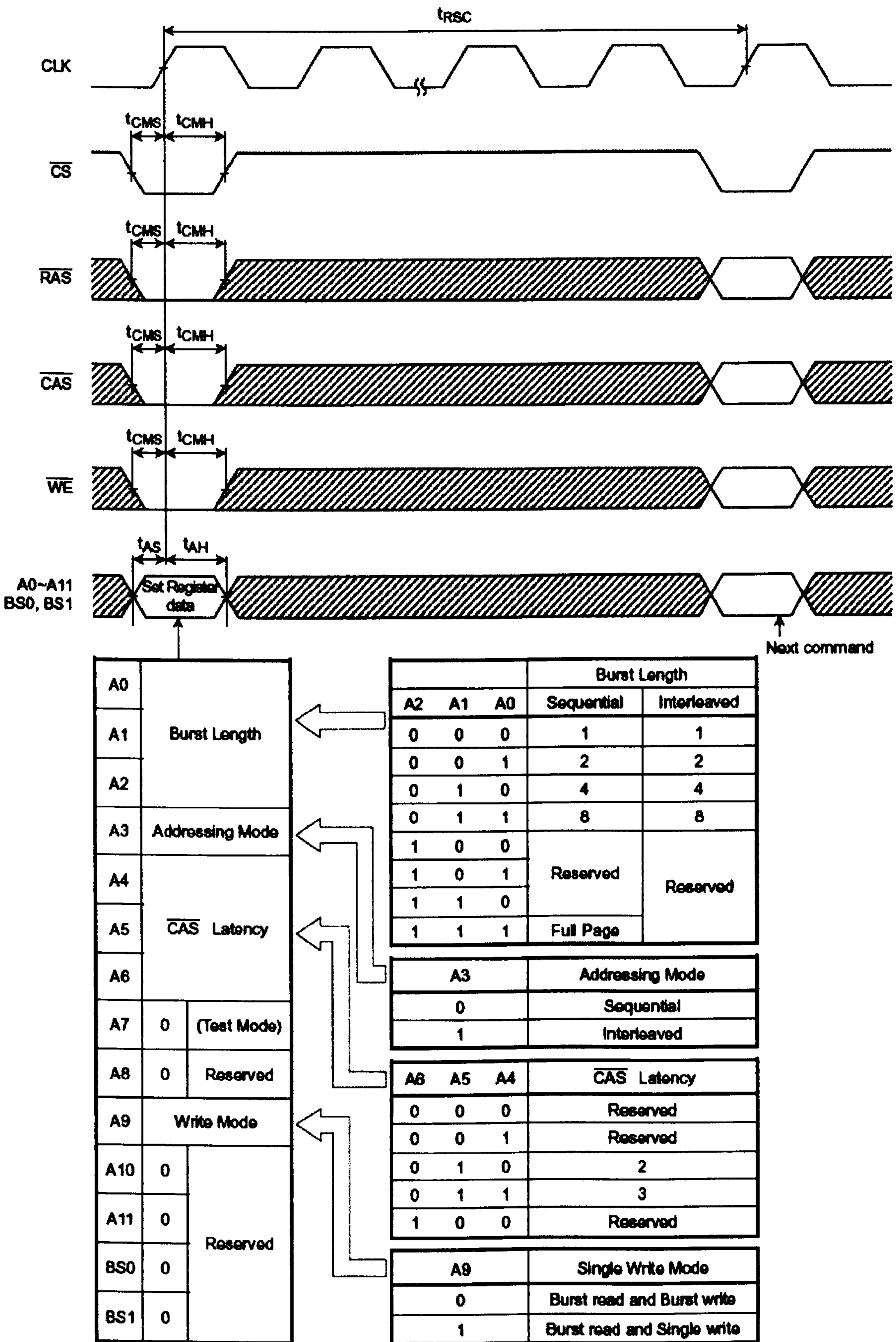


Figure 5-5 Mode Register Set cycle (From [62])

Streaming read command

To implement a streaming read transfer the SDRAM must be firstly performed a Bank Activate command *DoActivate* to put the idle bank into the active state, like opening a bank and row. Following that an SDRAM Read command *DoRead* is issued after *tRCD* (Active to Read/Write command delay time) from the Bank Activate command, the data is read out sequentially, synchronised to the positive edges of the data clock *data_clk*. The initial read data becomes available after CAS latency from the issuing of the Read command. As the SDRAM is operating at full page mode, Precharge command is used to terminate the burst read operation at CAS latency before the last data returned. The column address of the last data is specified at address bus *addcol_last*.

Upon returning of the last data the signal *done* is asserted to inform the Avalon interface that the streaming read request is completed. Figure 5-6 shows an example timing diagram of the burst read operation and Figure 5-7 illustrates the Burst Stop cycle by a timing chart.

Table 5-5 summarises the SDRAM operations which are performed to complete this streaming read command.

Table 5-5 Streaming read command

Command	SDRAM command	*bs0 1*	*A10*	*A0-A9 A11*	*cs_n*	*ras_n*	*ras_n*	*we_n*	Descriptions
Streaming read	*DoActivate*	V	V	V	L	L	H	H	Activate the selected bank
	DoRead	V	L	V	L	H	L	H	Burst read
	DoPrecharge	X	H	X	L	L	H	L	Burst read stop operation

Streaming write command

To implement a streaming write transfer the SDRAM must be firstly performed a Bank Activate command *DoActivate* to put the idle bank into the active state. An SDRAM Write command *DoWrite* is then issued after *tRCD* from the Bank Activate command, the input data is latched sequentially, synchronised with the positive edges of the data clock *data_clk*. The first write data should become valid on the *dq* bus when the

SDRAM write command actually occurs on the SDRAM interface. A Bank Precharge command is issued to terminate the burst write operation at one clock cycle after the last data is written.

Upon the completion of writing the last data signal *done* is asserted to inform the Avalon interface that the streaming write request is completed. Figure 5-6 shows an example timing diagram of the burst write operation and Figure 5-7 illustrate the Burst Stop cycle by a timing chart.

Table 5-5 summarises the SDRAM operations which are performed to complete this streaming write command.

Table 5-6 Streaming write command

Command	SDRAM command	*bs01*	*A10*	*A0-A9 A11*	*cs_n*	*ras_n*	*ras_n*	*we_n*	Descriptions
Streaming Write	*DoActivate*	V	V	V	L	L	H	H	Activate the selected bank
	DoWrite	V	L	V	L	H	L	L	Burst write
	DoPrecharge	X	H	X	L	L	H	L	Burst write stop operation

Auto-refresh command (internally generated)

The internal auto refresh command *refreshrequest* driven by the auto-refresh generator instructs the SDRAM controller to issue the Auto Refresh command to the SDRAM device. If other SDRAM commands are in progress, the auto-refresh request will be suspended until the other commands finish. However, if the refresh request and the other commands are issued at the same time, then auto-refresh has the higher priority and the other commands must wait until the auto-refresh command completes. By repeating the Auto Refresh cycle, each row is refreshed automatically provided by an internal refresh counter. The maximum refresh period of that device t_{REF} is 64ms and there are 4096 rows in each bank, so a burst of 4096 auto refresh cycles must be completed within 64ms. The memory device must have an Auto Refresh command issued at least every $64ms/4096 = 15.625\mu s$. In this system, as the video data was

designed to be clocked by a 100MHz clock, the SDRAM controller should generate an Auto Refresh command no more than every $15.625\,\mu s / 0.01\,\mu s = 1562$ data clock cycles.

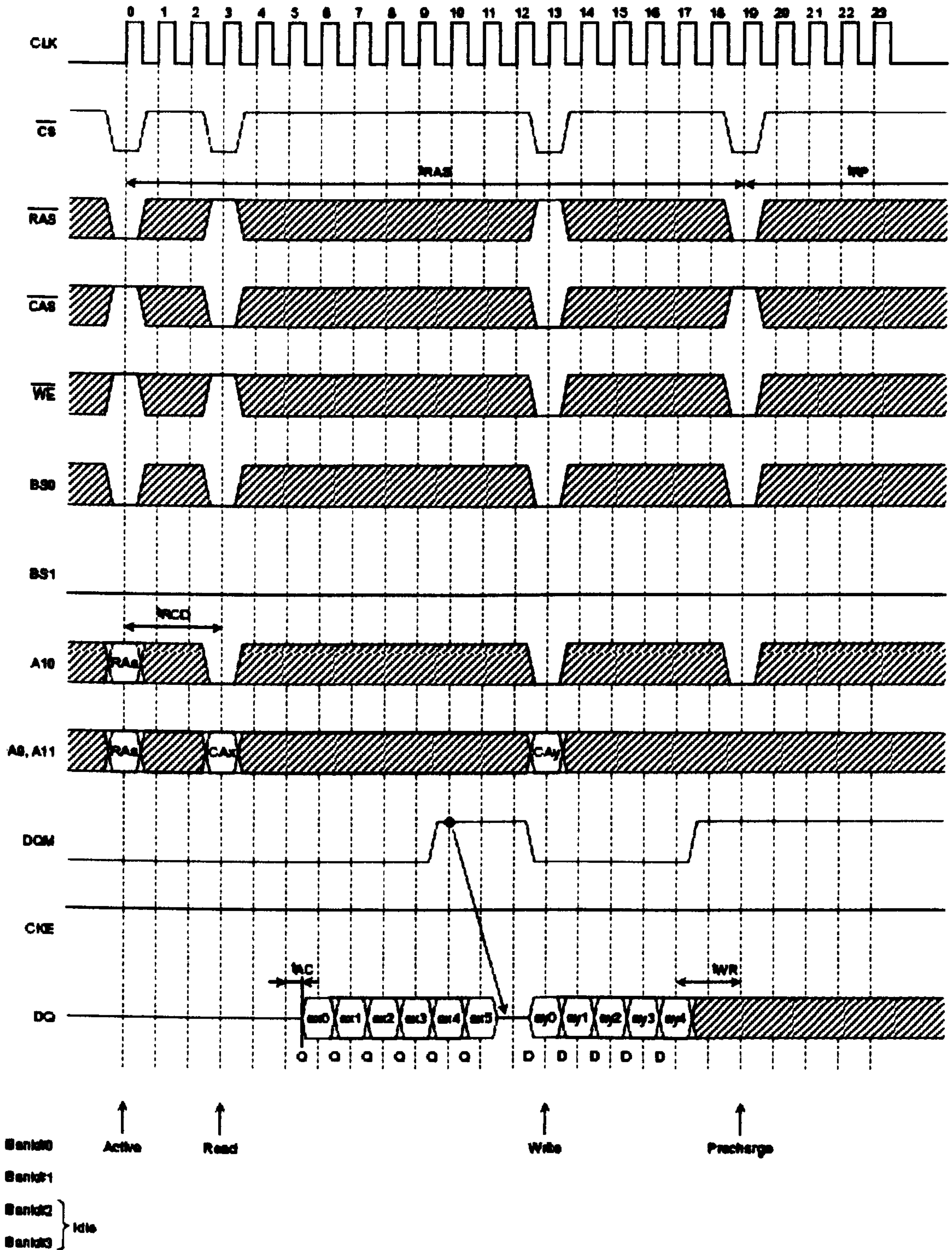


Figure 5-6 Page Mode Read/Write (Burst Length = 8, CAS Latency = 3) (From [62])

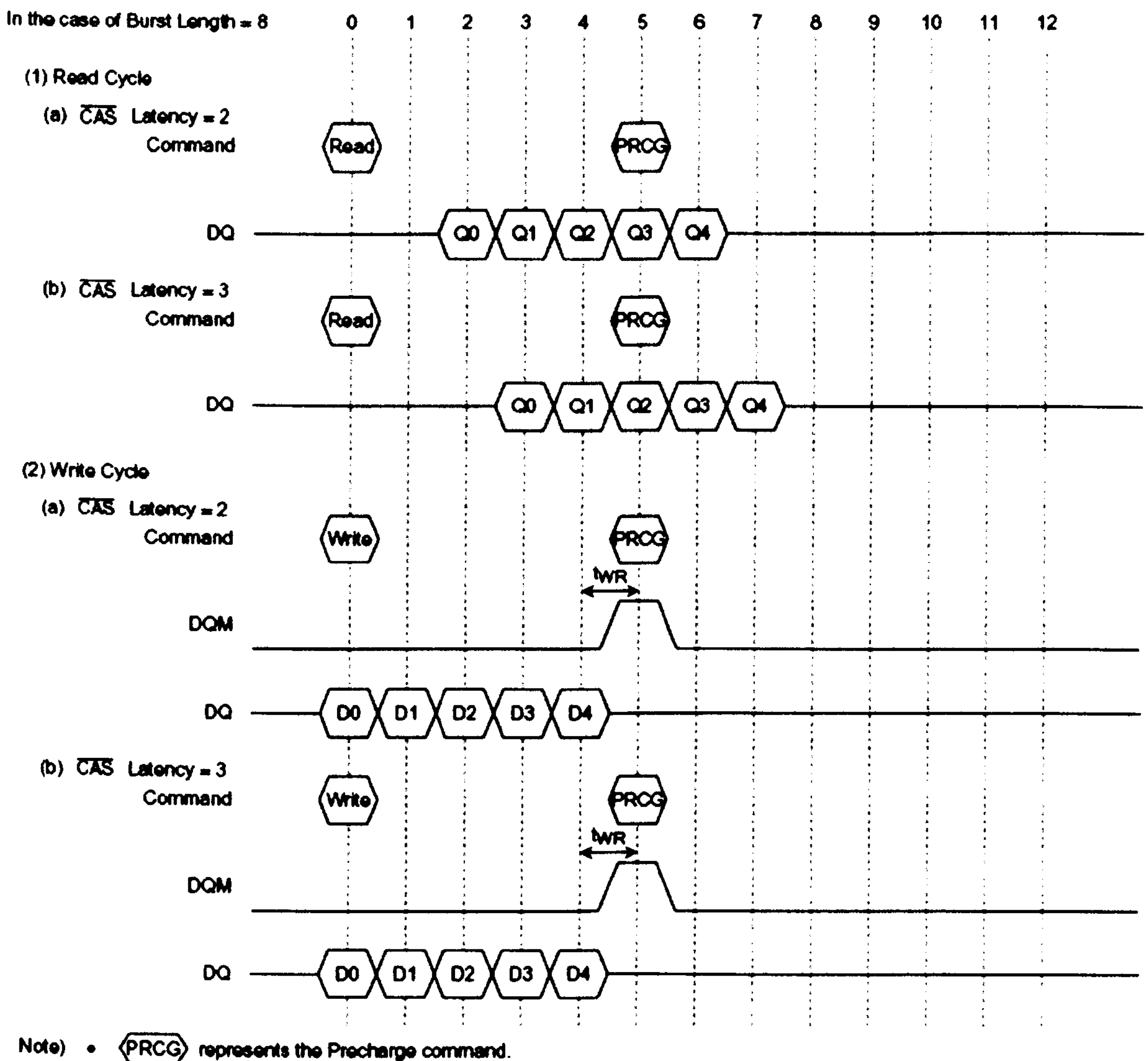


Figure 5-7 Timing chart for Burst Stop cycle (From [62])

The Auto Refresh command must be performed when all banks are in idle state therefore a Precharge All command must be implemented t_{RP} before issuing the Auto Refresh command. After the Auto Refresh command issued a minimum period of t_{RC} (Ref/Active to Ref/Active Command Period) must be elapsed before activating the next command. Figure 5-8 illustrates an Auto-Refresh cycle. Table 5-7 summarises the SDRAM operations which are performed to complete the auto-refresh operation.

Table 5-7 Auto refresh command

Command	SDRAM command	*bs0,1*	*A10*	*A0-9 A11*	*cs_n*	*ras_n*	*ras_n*	*we_n*	Descriptions
Auto refresh	*DoPrechargeAll*	X	H	X	L	L	H	L	Precharge all banks
	DoRefresh	X	X	X	L	L	L	H	Auto Refresh

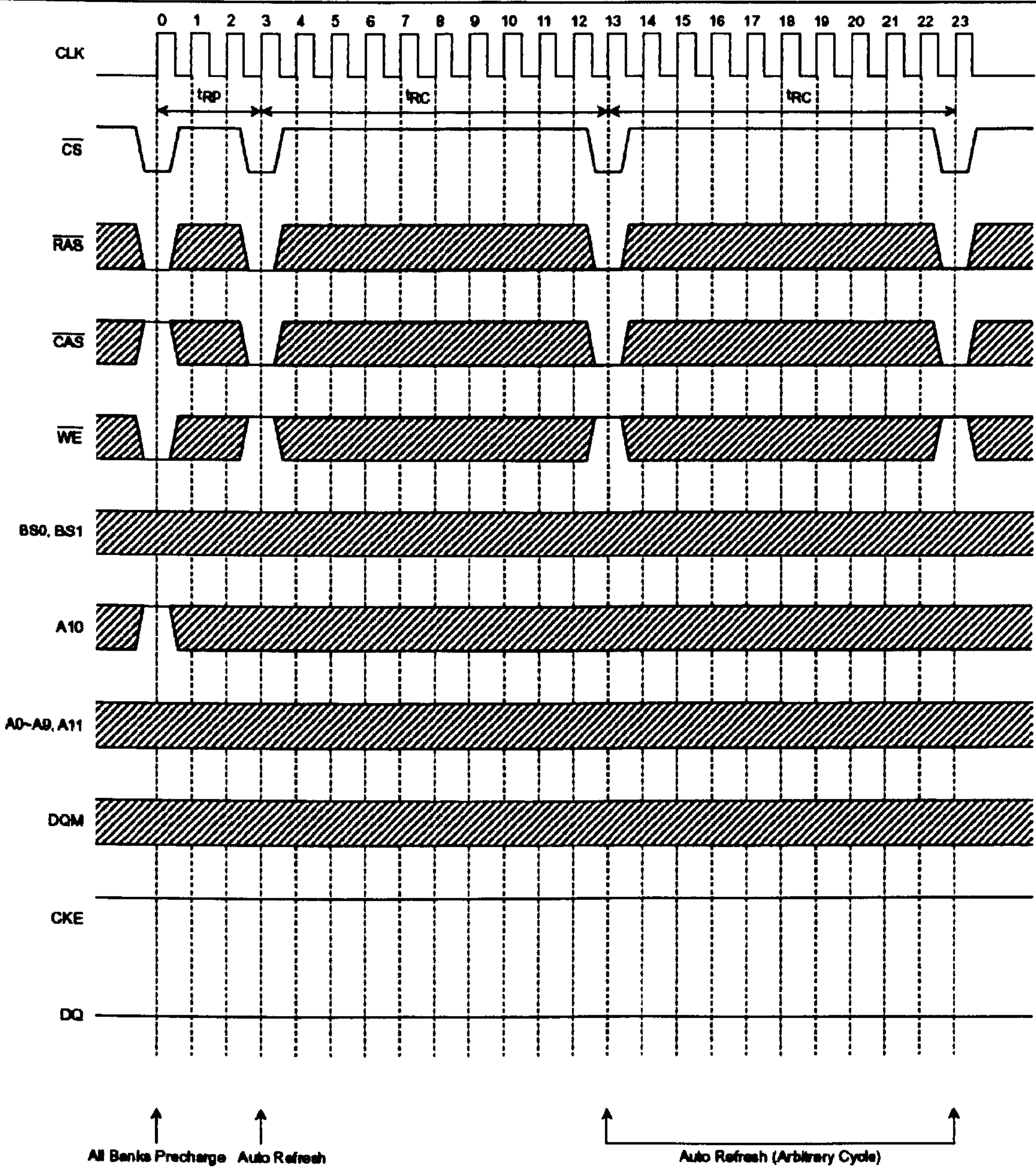


Figure 5-8 Auto Refresh cycle (From [62])

Power-up Sequence

According to the SDRAM specification, power-up must be performed in the following sequence.

1) Power must be applied to VCC and VCCQ (simultaneously) while all input signals are held in the "NOP" state; this is the default state in the SDRAM interface module. The clock signal must be started at the same time. This has already been satisfied by the board design.
2) After power-up a delay of at least 200µs is required. It is required that *dqmb* and *cke* signals must be held "High" (VCC levels) to ensure that the *dq* output is in a

High-impedance state.

3) All banks must be precharged.
4) The Mode Register Set command must be asserted to initialise the mode register.
5) A minimum of eight Auto-Refresh dummy cycles is required to stabilise the internal circuitry of the device.

To implement this sequence, the SDRAM command generator simply generates a *DoPrechargeAll* command after 200µs when *reset_n* activates. Then it issues a *DoSetModeReg* command to initialise the mode register with the default value (burst write burst read, CAS latency of 2, sequential addressing mode, full page burst length). Finally eight *DoRefresh* will be issued before entering the normal state.

5.2.2.3. SDRAM data path

The SDRAM data path module controls the multiplexing of the Avalon data and the SDRAM data *dq*. During an Avalon streaming read cycle, *data_slave_readdata_u* = *dq[63..32]*, while *data_slave_readdata_l* = *dq[31..0]*. During an Avalon streaming write cycle, *dq[63..32]* = *data_slave_writedata_u*, while *dq[31..0]* = *data_slave_writedata_l*.

As the SDRAM data pins are bidirectional, in the memory controller they are treated as a tri-state bus. During the Avalon streaming read transfers bus *dq* is placed in a high impedance 'Z'.

The data path module also generates the SDRAM word mask bus *dqmb*, which is driven from the two lowest significant bits (LSBs) of the Avalon address bus, to mask the specific video half-word (32 bits) within the 64-bit data (see Table 5-8) in write cycles.

Table 5-8 Description of the word mask

Wordmask	*dqmb[7..0]*	Description
"00"	"00000000"	Full word/64 bit mode (*data_slave_writedata_u* and *data_slave_writedata_l* are both written into SDRAM)
"01"	"00001111"	Half word mode (only *data_slave_writedata_u* is written into SDRAM)
"10"	"11110000"	Half word mode (only *data_slave_writedata_l* is written into SDRAM)
"11"	"11111111"	No data is written into SDRAM

5.2.3 Summary

The video memory controller was designed to drive the SDRAM device to buffer all video data used in this system. It has three sub-modules which are the Avalon interface, SDRAM controller and SDRAM data path module. The memory controller provides a standard Avalon bus interface to allow video masters to write to or read from the memory. The Avalon interface module is the one which controls this bus interface. It decodes the address information, and drives the SDRAM controller to perform Page Read or Page Write on the SDRAM device. The SDRAM controller, which actually drives the control signals to the SDRAM device, also generates the Auto-Refresh command and performs power-up initialisation on the device. The SDRAM data path module handles data transaction between the Avalon data ports and the SDRAM data pins. This memory controller also provides an Avalon slave port interface to allow the processor to modify the mode register content by writing to the CPU slave port in the Avalon interface module. The actual Mode Register Set command is implemented by the SDRAM controller.

Although the use of the pipelining can deliver the highest performance of using the SDRAM device [82], for example if two requests addressed on the same row in the same bank are issued, the memory controller could pipeline these two requests without issuing another Bank Activation command for the second request and a Precharge command is not necessary to be issued for the first request or even the second one if the third request is still addressed on the same bank and row. However, due to the existence of multiple masters in this system and the arbitration scheme used by the

Avalon bus module (see Chapter 6), this pipelining feature is difficult to implement in this system. Moreover, because the memory is operated in multiple bank mode (see section 5.7.1) which means all video masters always operate on different banks and rows, implementing the pipelining wouldn't really increase the performance. Therefore, in SIPS, a Bank Activation command is always asserted before performing a Page Read or Write on the SDRAM device.

In order to maximise the system performance, some special design methods were applied in the memory controller. Firstly, as the data bus width of the SDRAM device is 64 which is twice the full width of an Avalon bus port, it would be inefficient for a master to access the same SDRAM address twice to request a 64-bit word data. However, by implementing two Avalon master/slave pairs between the memory controller and the video masters, a 64-bit data word can be transferred in a single clock cycle.

In this system, a large block of video data is frequently required to be transferred, for example, the video display controller always requests a video line's data with a length of 80 words in 8-bit monochrome mode or 320 words in 24-bit RGB mode with video horizontal resolution of 640 (see section 5.3), the video capture controller always writes a video line's data into the memory (see section 5.4). By implementing Avalon streaming transfers between the memory controller and video masters the data transfer efficiency in this system can therefore be maximised. In order to accommodate these streaming transfers the SDRAM device is configured to work in page mode only. However various burst length from 1 to 512 can be achieved by terminating the burst at an appropriate time.

5.3 Video display controller

The video display controller provides the function of requesting and receiving video data from the memory device, and driving the Lancelot VGA board with these data as well as the relevant VGA timing signals to get the video displayed on the CRT monitor. Furthermore it provides an I/O channel to allow the Nios processor to configure and control this video display controller.

In this section, it firstly introduces the main features of this display controller. Secondly a brief overview of the VGA display mode is presented. Following that it describes this display controller core in details. Finally a brief summary is given.

5.3.1 Main features

The typical features of this display controller are:

- Always output 24-bit video data
- Support 2 types of pixel format - 8-bit monochrome or 24-bit RGB
- 1 video word = 64 bits = 2 pixels in RGB or 8 pixels in monochrome mode
- Double line buffer mode to guarantee sufficient bandwidth
- Display images with variable size of up to 640x480
- Frame rate = 60 f/s
- Direct access to the memory device to request video data
- Support standard Avalon bus interface
- Variable Avalon streaming transfer length based on the horizontal resolution
- 25MHz pixel clock
- Interrupt supported

5.3.2 Video graphic arrays (VGA)

VGA is a display standard for the PC. VGA uses an analogue monitor such as CRT. It was first marketed by IBM in 1987. VGA generally refers to a resolution with 640x480. It may also refer to the 15-pin VGA socket on a PC in general. The original VGA only displays 256 colours by using a colour palette, however, in this system it can display up to 16 million colours in 24-bit RGB mode or 256 grey levels in monochrome mode, depends on how this display controller generated (set in the generic map).

5.3.3 Description of the video display controller

In implementation, this video display controller consists of two sub-modules. They are the Avalon interface module and VGA driver module. Figure 5-9 shows the connection diagram of this display controller core. The Avalon interface module controls all the Avalon bus interface signals for requesting and receiving video data from the memory device. The VGA driver mainly generates the VGA timing signals to drive the VGA

display along with the video data read from the system memory.

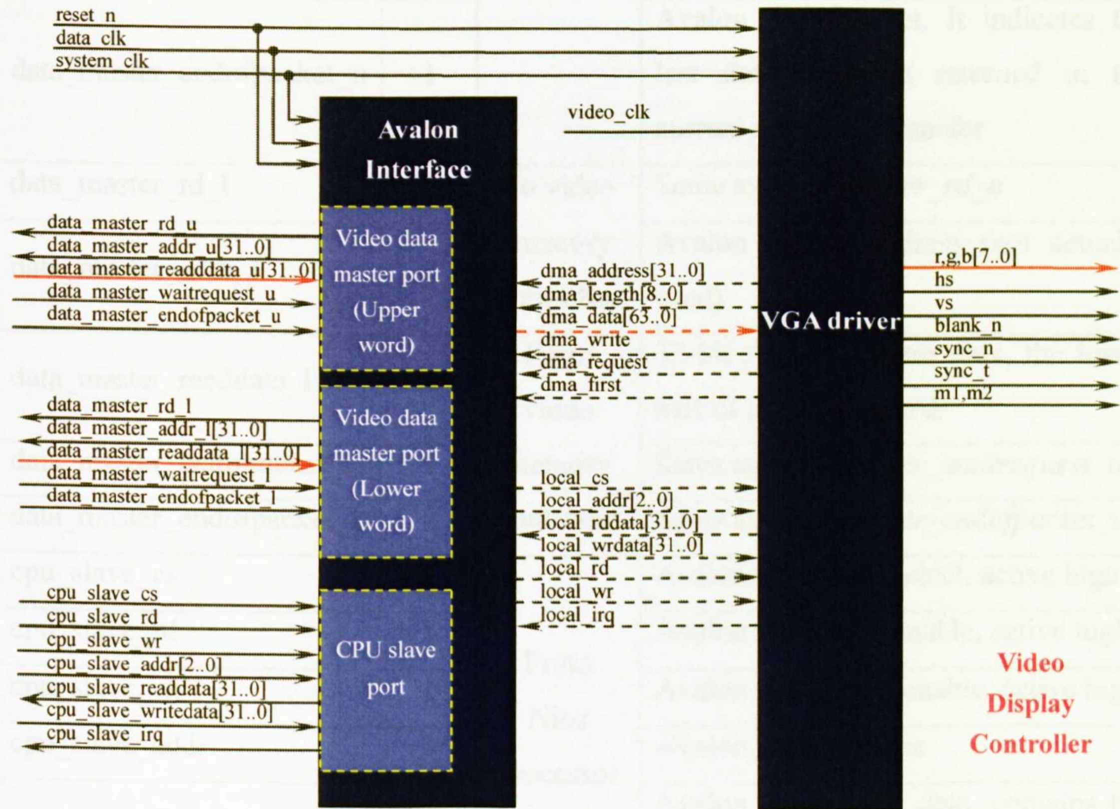


Figure 5-9 Block diagram of the video display controller

Table 5-9 lists the top level I/Os of the display controller.

Table 5-9 Video display controller top level signals

Signal	Width	Direction	Description
reset_n	1	Global	Global reset, active low
data_clk	1	From clock generator	video data transfer clock
system_clk	1		System clock, drives Avalon transfer inilialised by the Nios processor
video_clk	1		VGA pixel clock
data_master_rd_u	1	To video memory controller	Avalon master read enable, active high
data_master_addr_u	32		Avalon master address
data_master_readdata_u	32	From video memory controller	32-bit Avalon master data, the upper half of the video word
data_master_waitrequest_u	1		Avalon waitrequest. It indicates the address is finished being decoded and

			the valid data is available to be read
data_master_endofpacket_u	1		Avalon endofpacket. It indicates the last data is being returned in the current streaming transfer
data_master_rd_l	1	To video memory controller	Same as *data_master_rd_u*
data_master_addr_l	1		Avalon master address (not actually used)
data_master_readdata_l	32	From video memory controller	32-bit Avalon master data, the lower half of the video word
data_master_waitrequest_l	1		Same as *data_master_waitrequest_u*
data_master_endofpacket_l	1		Same as *data_master_endofpacket_u*
cpu_slave_cs	1	From Nios processor	Avalon slave chip select, active high
cpu_slave_rd	1		Avalon slave read enable, active high,
cpu_slave_wr	1		Avalon slave write enable, active high,
cpu_slave_addr	3		Avalon slave address
cpu_slave_writedata	32		Avalon slave write data, contains the configuration data
cpu_slave_readdata	32	To Nios processor	Avalon slave read data, contains the status registers
cpu_slave_irq	1		Avalon interrupt, active high, indicates a video frame has displayed out, cleared by a CPU write
hs	1	To VGA	Horizontal synchronisation signal
vs	1		Vertical synchronisation signal
r	8	To video DAC on the Lancelot VGA board	Video blue
g	8		Video red
b	8		Video green
blank_n	1		Control signals
sync_n	1		
sync_t	1		
m1	1		Mode select signals
m2	1		

Table 5-10 lists the interconnection signals between the Avalon interface and the VGA driver module.

Table 5-10 Internal signals of the video display controller

Signal	Width	Description
dma_address	32	DMA start address. It is where a new video frame starts for display in the memory device. Specified by the software
dma_length	9	DMA transfer length of a video line. For example it is 80 in 8-bit monochrome and 320 in 24-bit RGB mode with line resolution of 640
dma_data	64	One video word = 2 pixels in 24-bit RGB or 8 pixels in 8-bit monochrome
dma_write	1	Write enable to the line buffers
dma_request	1	DMA transfer request
dma_first	1	Indicates the first line DMA transfer request
local_cs	1	*cpu_slave_cs*
local_addr	3	*cpu_save_addr*
local_rddata	32	*cpu_slave_readdata*
local_wrdata	32	*cpu_slave_writedata*
local_rd	1	*cpu_slave_rd*
local_wr	1	*cpu_slave_wr*
local_irq	1	*cpu_slave_irq*

5.3.3.1. Avalon interface

The Avalon interface module provides I/O controls to the Avalon bus for the video display controller to request and receive video data from the memory device through the memory controller, and also to communicate with the Nios processor for receiving configuration data and sending status information.

There are three Avalon bus ports in the Avalon interface module including one simple Avalon slave port and two Avalon streaming read master ports.

CPU slave port – simple Avalon slave port

The simple Avalon slave port is mastered by the Nios processor for configuration purpose such as enabling display, configuring image resolution, setting up DAC mode, specifying the DMA start address (the address in the SDRAM where the image starts) and burst length. Furthermore, current system status and configuration information also can be sent out so that the user can keep track of them. These Avalon signals are passed to the VGA driver sub-module for decoding.

Video data master ports – streaming Avalon master ports

The two Avalon masters are video data streaming read masters which master the two streaming slave ports in the memory controller. Each of them handles 32 bit data. They are used for requesting and receiving video streams from the main memory. Therefore in every master read unit cycle the video display controller can receive 64-bit data which contains 2 video pixels in 24-bit RGB or 8 pixels in 8-bit monochrome mode. Table 5-11 and Table 5-12 show the pixel assignments in the 64-bit *dma_data* in 8-bit monochrome mode and 24-bit RGB mode respectively.

Table 5-11 Pixel assignment in 8-bit monochrome mode for video display

dma_data(63..32) (*data_master_readdata_u*)				*dma_data(31..0)* (*data_master_readdata_l*)			
63..56	55..48	47..40	39..32	31..24	23..16	15..8	7..0
Pixel 8n+8	Pixel 8n+7	Pixel 8n+6	Pixel 8n+5	Pixel 8n+4	Pixel 8n+3	Pixel 8n+2	Pixel 8n+1

Note: n=0~video line width / 8

Table 5-12 Pixel assignment in 24-bit RGB mode for video display

dma_data(63..32) (*data_master_readdata_u*)		*dma_data(31..0)* (*data_master_readdata_l*)	
63..56	55.. 32	31..24	23..0
X	Pixel 2n+2	X	Pixel 2n+1

Note: n=0~video line width / 2 X=don't care

Each streaming request returns a video line data, the length of this video line is specified in *dma_length* derived from the video horizontal resolution. For example, if the horizontal resolution is 640, then the DMA length is 80 in 8-bit monochrome mode or 320 in 24-bit RGB mode.

Whenever there is a *dma_request* from the VGA driver, the Avalon interface module sends out a read request to the Avalon bus along with the relevant address information. Once this request is accepted and decoded by the memory controller (by monitoring signal *data_master_waitrequest_u* or *data_master_waitrequest_l*) then a block of valid video data will be returned and stored in one of the two line buffers, the signal *dma_write* is held high during the period of valid data returning. This data will be sent out to display in the next active video line scan period (during *hs* is high). Signal *data_master_endofpacket_u* or *data_master_endofpacket_l* is used to tell the master port that the last data is returning.

The start address of a requested video frame is specified in signal *dma_address* which is decoded from the CPU configuration data. When requesting the first video line the Avalon master address *data_master_addr_u* is *dma_address*, then it will be increased by one memory row space in every read request until the whole frame has been transferred. Upon this point the Avalon master address is reset to *dma_address* which can be the same as the previous one or different depends on how the software handles it. Signal *dma_first* indicates whether the new request is for the first frame line.

Figure 5-10 shows an ASM chart of the video data master read transfers.

5.3.3.2. VGA driver

The main function of this VGA driver is to generate the necessary timing signals along with the video data to drive the video for display. Figure 5-11 shows the composition of this VGA driver.

As discussed in Chapter 4, to display an image, CRT monitors typically require 5 signals: R, G, B, hsync, and vsync. The R, G, and B signals represent the intensity of the red, green and blue signal elements that compose a pixel. The *hs* signal provides the monitor with a horizontal synchronisation signal; the *vs* signal provides a vertical synchronisation signal to the monitor [73]. The VGA driver in this design also generates an additional synchronisation signals which indicates when the pixels are active. This signal is *blank_n* signal. If the *blank_n* is asserted the output of the video DAC is forced to zero.

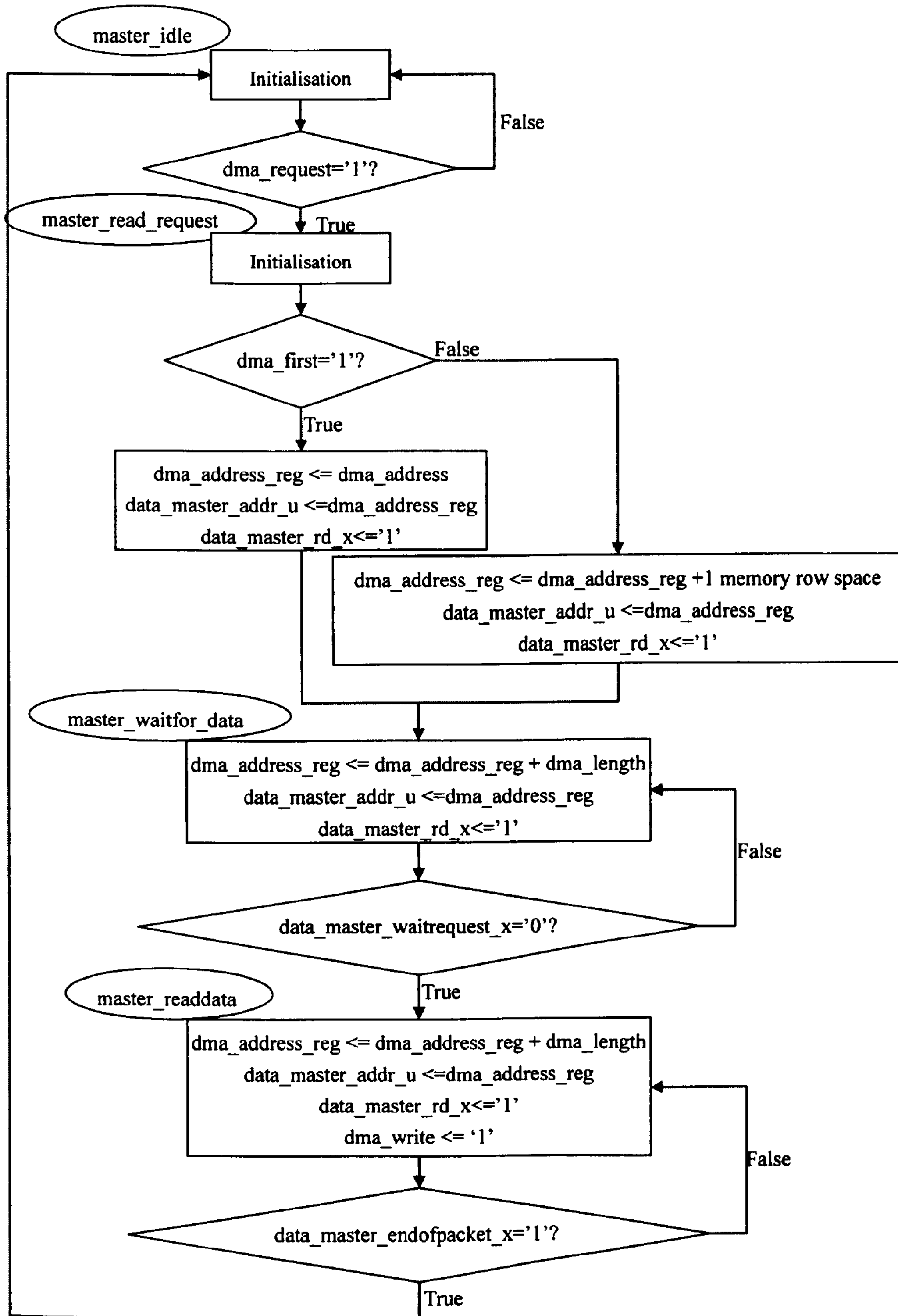


Figure 5-10 ASM chart of the Avalon streaming master read transfer

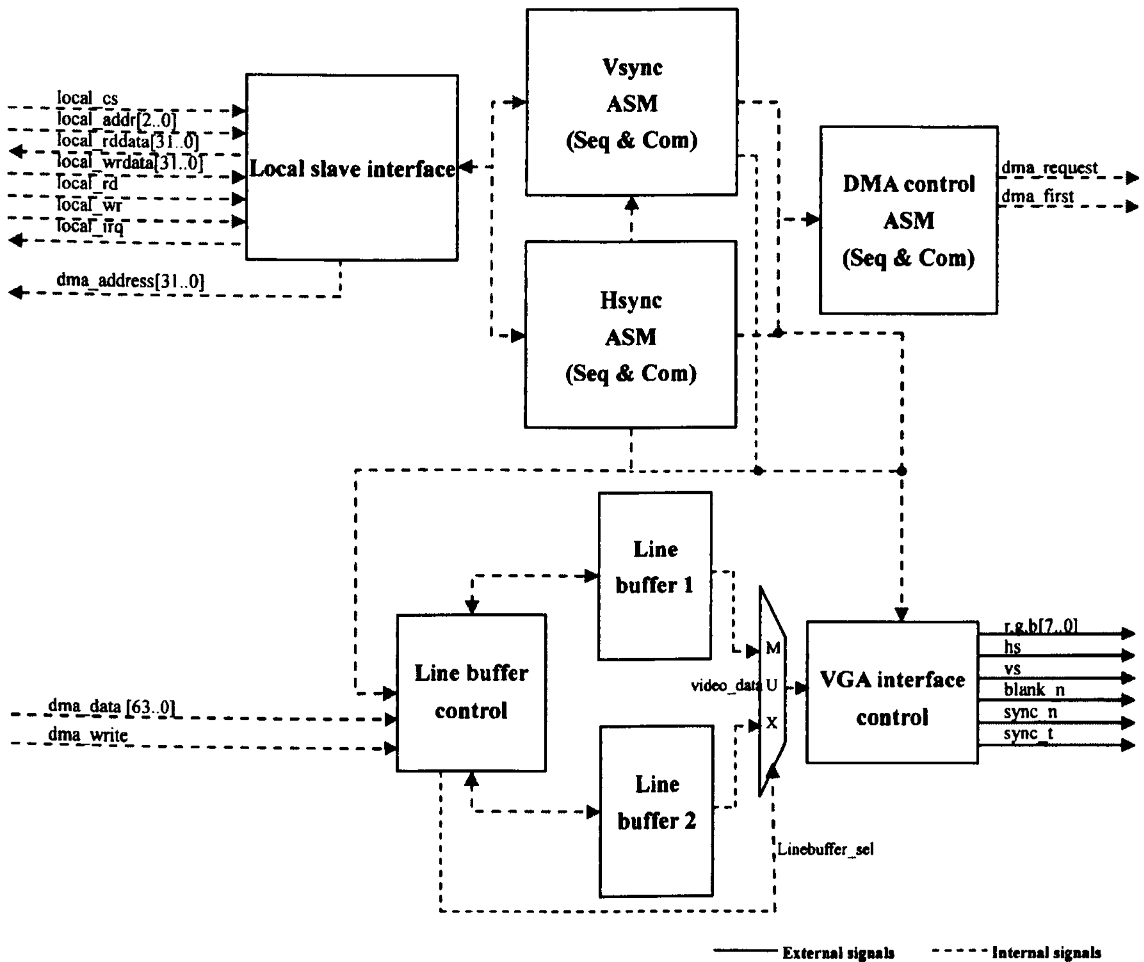


Figure 5-11 Block diagram of the VGA driver

Local slave interface

The Nios processor provides several registers to configure and control this video display controller. These registers include a resolution register, DMA address register, video control register and status register. Refer to Appendix E for the register map. This local slave interface generates these registers by decoding the local address. Interrupt signal *local_irq* is asserted when the bottom line of an image has been sent out to the display. This interrupt is useful because the Nios processor knows when a video frame has finished being display and it can set the next display bank address (see multiple bank operation in section 5.7.1) during the video blanking period. This interrupt must be cleared by writing to the DMA address register before carrying on displaying the next image to prevent stalling the system.

Hsync & Vsync ASM

In the VGA driver, those horizontal and vertical timing signals are driven by two state machines respectively. The two state machines are triggered when the start bit in the control register is set. The typical resolution and frame rate used in this system is 640x480 at 60 fps. Figure 5-12 and Figure 5-13 show the vertical and horizontal timing diagram at these typical settings, while Table 5-13 and Table 5-14 list the timing details for each of them.

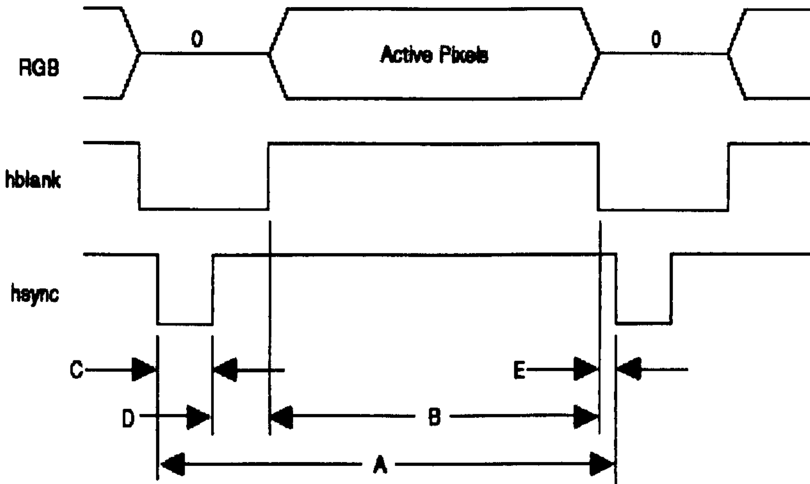


Figure 5-12 640×480 VGA horizontal timing (From [73])

Table 5-13 640×480 VGA horizontal timing

640×480 VGA Horizontal Timing		
	Description	Time (us)
A	Line scan period	31.77
B	Active video period	25.60
C	Sync period	3.77
D	Back porch	1.89
E	Front porch	0.51

Since the active video period at 640x480 is 25.60μs, it is not difficult to obtain the frequency of the pixel clock which is $\frac{640}{25.60\mu s} = 25MHz$.

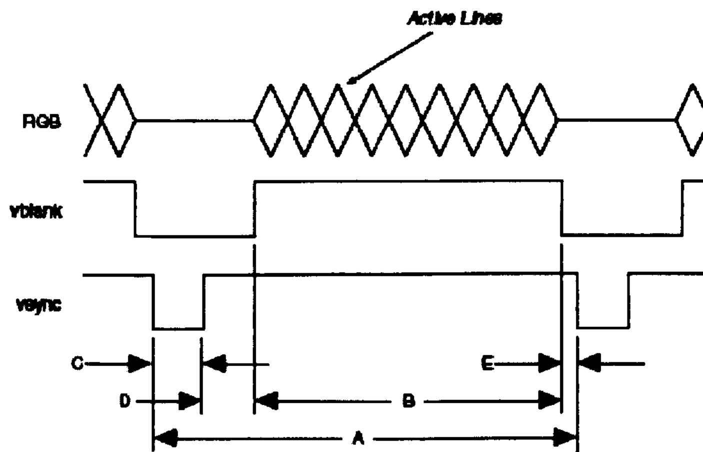


Figure 5-13 640×480 VGA vertical timing (From [73])

Table 5-14 640×480 VGA vertical timing

640×480 VGA vertical timing		
	Description	Time (ms)
A	Frame period	16.68
B	Active video period	15.25
C	Sync period	0.064
D	Back porch	1.02
E	Front porch	0.343

DMA control ASM

This state machine generates the signals *dma_request* & *dma_first* based on the vertical & horizontal timing. Whenever a video line has finished displaying during the active video period it should immediately initialise *dma_request* to request a new video line. Signal *dma_first* indicates the current image has reached the bottom so the next *dma_requst* should start from the initial *dma_address*.

Line buffers 1 & 2

These two line buffers are actually two identical FIFOs mapped on the same memory address space. The size of this line buffer can be conditionally generated as 128x64 or 512x64 based on the pixel format. If the pixel format is 24-bit RGB, then the size of the line buffer is 512x64, 2 pixels per line. If the pixel format is 8-bit grey-level, then the size of the line buffer is 128x64, 8 pixels per row. This configuration satisfies the maximum horizontal width which is 640 in this system. The write port of these buffers

is driven by data clock while the read port is driven by the video clock. Whenever signal *dma_write* is enabled, one of these line buffers begins to be written with the new video data (*dma_data*) until it's filled up. While the other one sends out the stored image data to the VGA driver for display during the active pixel period. These two buffers perform read and write alternatively. By using this two-buffer mode, a sufficient bandwidth can be guaranteed. In order to increase the system performance, the I/O interfaces of these two buffers are all synchronous.

Line buffer control

This line buffer control block provides the mechanism to switch the operation mode of these two line buffers in terms of the video timing & Avalon bus status. The switch happens when the one being filled up with new video data finishes filling and all stored data are sent out from the one being read. Table 5-15 gives a view of how these two line buffers alternate the operations based on the video line number.

Table 5-15 Alternating operations of the double line buffer

Measured Item		Line Number								
W	Line buffer 1	2		4		6	....	480		
	Line buffer 2	1	3		5		....			
R	Line buffer 1			2		4	....	478		480
	Line buffer 2		1		3		....		479	
Linebuffer_Sel		0	1	0	1	0	....		1	0

For example, when the video display is enabled, the 1st video line is written into line buffer 2. Just before the 1st active line period the 2nd video line is written into line buffer 1 while VGA driver starts displaying the 1st video line from line buffer 2. When the 1st video line finishes displaying, operations swap, the 3rd video line is written into line buffer 2 again and during the active video period VGA driver displays the 2nd video line from line buffer 1.

VGA interface control

This block synchronises the video data read from the line buffers with the timing signals *hs*, *vs* and *blank_n* from the timing state machines and drives them to the output.

In 24-bit RGB mode, the 24-bit pixel contained in *video_data* is mapped into VGA data outputs *r*, *g* and *b* directly. While in 8-bit monochrome mode, each 8-bit pixel contained in *video_data* is fed into *r*, *g* and *b* respectively so that a black/white image can be displayed.

5.3.4 Summary

The video display controller provides an I/O interface to the Avalon bus module and the display interface board. It consists of two sub-modules which are the Avalon interface module and the VGA driver module. The Avalon interface module controls the Avalon streaming transfers for reading video data from the memory device. Two Avalon streaming read master ports are used to master the two streaming slave ports in the memory controller. Video data read from the memory device with the length of one video line is fed into the line buffers in the VGA driver module. During the active video period this video data is sent out to the display. The VGA driver generates the relevant timing signals.

The display controller provides a configuration slave port for the Nios processor to control the display such as changing resolution, enabling display and changing DMA address. When a frame is displayed an interrupt is asserted.

This display controller can be conditionally generated into either 8-bit monochrome mode or 24-bit RGB mode. In 8-bit monochrome mode, the size of the line buffers is 128x64, and each buffer location stores 8 video pixels. The burst length of a streaming read is eight times less than the total number of pixels in one video line. While in 24-bit RGB mode, the size of the line buffers is 512x64, each buffer location stores 2 video pixels. The burst length is two times less than the total number of pixels in one video line. This configuration makes the system more flexible, economic and efficient. In 8-bit monochrome mode, in order to display black/white images the 8-bit data is mapped into the output *r*, *g* and *b* respectively.

5.4 Video capture controller

The video capture controller provides the function of buffering video data from the external camera via the camera interface card and transferring those data into the memory device via the Avalon bus. Also, it provides an I/O channel to allow the Nios processor to configure and control this video controller.

In this section, it firstly presents the main features of this video capture controller, and then a brief introduction of the CameraLink standard is given. Thirdly a detailed description of the controller core is given followed by a brief introduction to the serial control interface to the camera. Finally a brief summary of this capture controller core is presented.

5.4.1 Main features

The main features of the video capture controller are as follows:

- Support both 8-bit monochrome and 24-bit RGB video input
- Provide a serial control interface to configure the camera
- Support input video line resolution with size of up to 1024
- Double line buffer mode to guarantee sufficient bandwidth
- Direct access to the memory device to transmit video data
- Variable Avalon streaming transfer length based on the horizontal resolution
- 1 video word = 64 bits = 2 pixels in 24-bit RGB or 8 pixels in 8-bit monochrome mode
- Pixel clock supplied by the camera with various programmable frequencies
- Interrupt supported

5.4.2 CameraLink

CameraLink is a communication interface for vision applications. The interface extends the base technology of Channel Link to provide a specification more useful for vision applications [60]. It is designed by National Semiconductor which is based on LVDS for the physical layer. Channel Link consists of a driver and receiver pair. The driver accepts 28 single-ended data signals and a single-ended clock. The data is serialized 7:1, and the four data streams and a dedicated clock are driven over five LVDS pairs. The receiver accepts the four LVDS data streams and LVDS clock, and then drives the

28 bits and a clock to the board. Figure 5-14 illustrates the Channel Link operation.

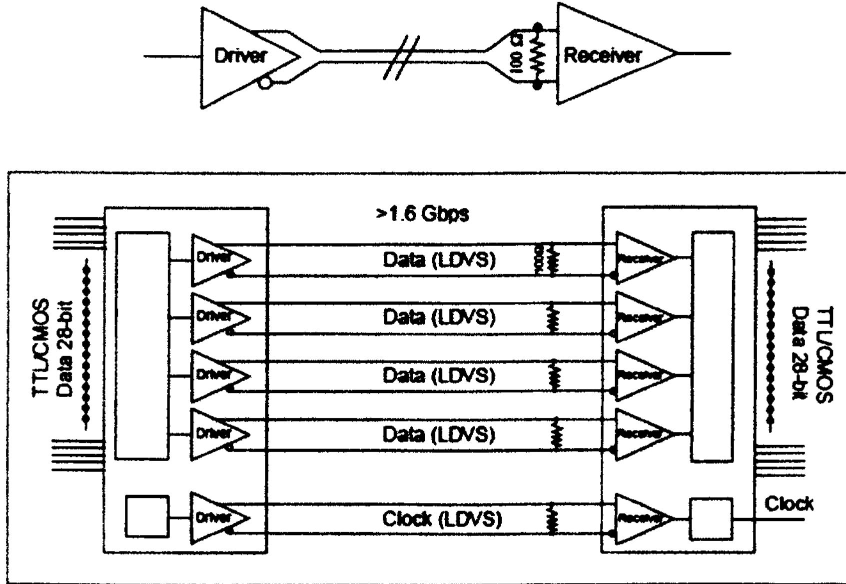


Figure 5-14 Channel Link operation (From [60])

According to the CameraLink specification, the typical signals transferred through a CameraLink cable are camera control signals, serial communication signals, a video clock and 28-bit data which includes 24-bit video data and four enables. Table 5-16 lists all the CameraLink signals.

Table 5-16 CameraLink signals

Signal	Descriptions	
CLOCK		Pixel clock
DATA	Video data	24-bit video data
FVAL		Frame Valid (FVAL) is defined high for valid lines
LVAL		Line Valid (LVAL) is defined high for valid pixels
DVAL		Data Valid (DVAL) is defined high when data is valid
Spare		A spare has been defined for future use
CC1	Camera control signals	Camera control 1
CC2		Camera control 2
CC3		Camera control 3
CC4		Camera control 4
SerTFG	Serial communication	Differential pair with serial communications to the interface card
SerTC		Differential pair with serial communications to the camera

5.4.3 Description of the video capture controller

The video capture controller was implemented with two sub-modules. They are the Avalon interface module and the video receiver module. Figure 5-15 shows the block diagram of this capture controller.

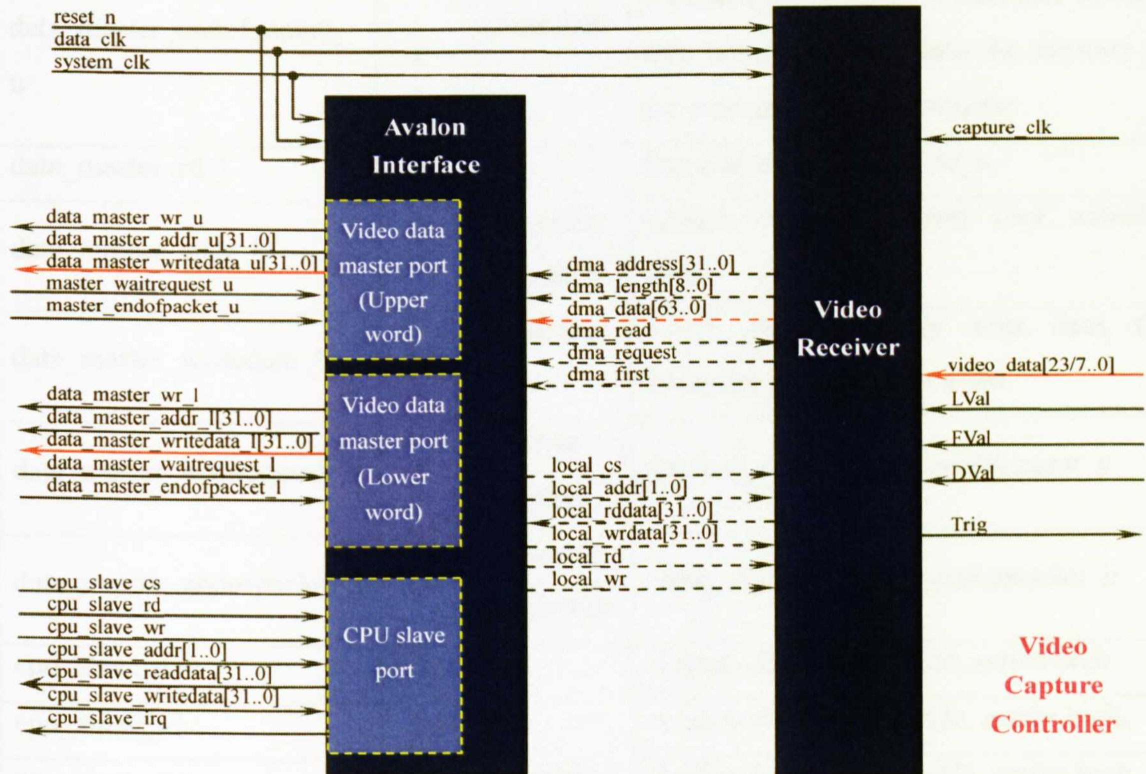


Figure 5-15 Block diagram of the video capture controller

Table 5-17 lists the top level I/Os of the video capture controller.

Table 5-17 Video capture controller top level signals

Signal	Width	Direct	Description
reset_n	1	Global	Global reset, active low
data_clk	1	From clock generator	video data transfer clock
system_clk	1		System clock, drives the Avalon transfer inilialised by the Nios processor
data_master_wr_u	1	To video memory controller	Avalon master write enable, active high
data_master_addr_u	32		Avalon master address
data_master_writedata_u	32		32-bit Avalon master write data, the upper half of the video word

data_master_waitrequest_u	1	From video memory controller	Avalon waitrequest. It indicates the address is finished being decoded and the memory device is ready to receive new data
data_master_endofpacket_u	1		Avalon endofpacket. It indicates the last data is being written into the memory in the current streaming transfer
data_master_rd_l	1	To video memory controller	Same as *data_master_rd_u*
data_master_addr_l	1		Avalon master address (not actually used)
data_master_writedata_l	32		32-bit Avalon master write data, the lower half of the video word
data_master_waitrequest_l	1	From video memory controller	Same as *data_master_waitrequest_u*
data_master_endofpacket_l	1		Same as *data_master_endofpacket_u*
cpu_slave_cs	1	From Nios processor	Avalon slave chip select, active high
cpu_slave_rd	1		Avalon slave read enable, active high,
cpu_slave_wr	1		Avalon slave write enable, active high,
cpu_slave_addr	2		Avalon slave address
cpu_slave_writedata	32		Avalon slave write data, contains the configuration data
cpu_slave_readdata	32	To Nios processor	Avalon slave read data, contains the status registers
cpu_slave_irq	1		Avalon interrupt, active high, indicates a full video frame has been captured to the memory, cleared by a CPU write
capture_clk	1	Camera Link signals	*Xclk*, Pixel clock
video_data	8/24		Video data
FVal	1		*FVAL*, frame valid, active high
LVal	1		*LVAL*, line valid, active high
DVal	1		*DVAL*, data valid, active high
Trig	1		*CC3*, external camera trigger, active high

Table 5-18 lists the interconnection signals between the Avalon interface and the video receiver.

Table 5-18 Internal signals of the video capture controller

Signal	Width	Description
dma_address	32	DMA start address. It is where a new video frame starts to be captured into in the memory device. Specified by the software
dma_length	9	DMA transfer length of a video line. For example it is 80 in monochrome or 320 in RGB with line resolution of 640
dma_data	64	One video word = 2 pixels in 24-RGB or 8 pixels in 8-bit monochrome
dma_read	1	Read enable to the line buffers
dma_request	1	DMA transfer request
dma_first	1	Indicates first DMA transfer request
local_cs	1	*cpu_slave_cs*
local_addr	2	*cpu_save_addr*
local_rddata	32	*cpu_slave_readdata*
local_wrdata	32	*cpu_slave_writedata*
local_rd	1	*cpu_slave_rd*
local_wr	1	*cpu_slave_wr*
local_irq	1	*cpu_slave_irq*

5.4.3.1. Avalon interface

The Avalon interface module provides controls over the Avalon bus interface signals for the video capture controller to transmit video data to the memory device via the memory controller, and also to communicate with the Nios processor for receiving configuration data and sending status information.

There are three Avalon bus ports in the Avalon interface module including one simple Avalon slave port and two Avalon streaming write master ports.

CPU slave port – simple Avalon slave port

This simple Avalon slave port is mastered by the Nios processor for configuration

purpose such as starting video capture, specifying the DMA start address and clearing the interrupt. Furthermore, current camera status such as the resolution and video timing can be sent back so that the user can keep track of them. These Avalon signals are passed to the video receiver module for decoding.

Video data master ports – streaming Avalon master ports

The two Avalon masters are video data streaming masters which master the two streaming slaves in the memory controller. They are used for sending the captured video data to the main memory. Each of them handles 32 bit data. Therefore in every master write unit cycle the video capture controller can transmit 64 bit data which contains 2 video pixels in 24-bit RGB or 8 in 8-bit monochrome mode. Table 5-19 and Table 5-20 show the pixel assignments in this 64-bit *dma_data* in 8-bit monochrome mode and 24-bit RGB mode respectively.

Table 5-19 Pixel assignment in 8-bit monochrome mode for video capture

dma_data(63..32) (data_master_writedata_u)				*dma_data(31..0) (data_master_writedata_l)*			
63..56	55..48	47..40	39..32	31..24	23..16	15..8	7..0
Pixel 8n+8	Pixel 8n+7	Pixel 8n+6	Pixel 8n+5	Pixel 8n+4	Pixel 8n+3	Pixel 8n+2	Pixel 8n+1

Note: n=0~video line width / 8

Table 5-20 Pixel assignment in 24-bit RGB mode for video capture

dma_data(63..32) (data_master_writedata_u)		*dma_data(31..0) (data_master_writedata_l)*	
63..56	55.. 32	31..24	23..0
X	Pixel 2n+2	X	Pixel 2n+1

Note: n=0~video line width / 2 X=don't care

Whenever there is a *dma_request* asserted by the video receiver, the Avalon interface block sends out an Avalon streaming write request to the memory controller slave ports along with the relevant address information. Once this request is accepted and decoded by the memory controller (by monitoring signal *data_master_waitrequest_x*) then a block of video data *dma_data* read from the video receiver will be transferred out to the Avalon bus and stored in the main memory. Signal *dma_read* is held high during the period of active data transferring. This signal is used to enable the video receiver to

read from the temporary line buffers. The whole streaming write transfer completes upon assertion of the signal *data_master_endofpacket_x*, which means the memory slave is receiving the last write data and will be closed in the next clock cycle. Figure 5-16 shows an ASM chart of an Avalon streaming write transfer.

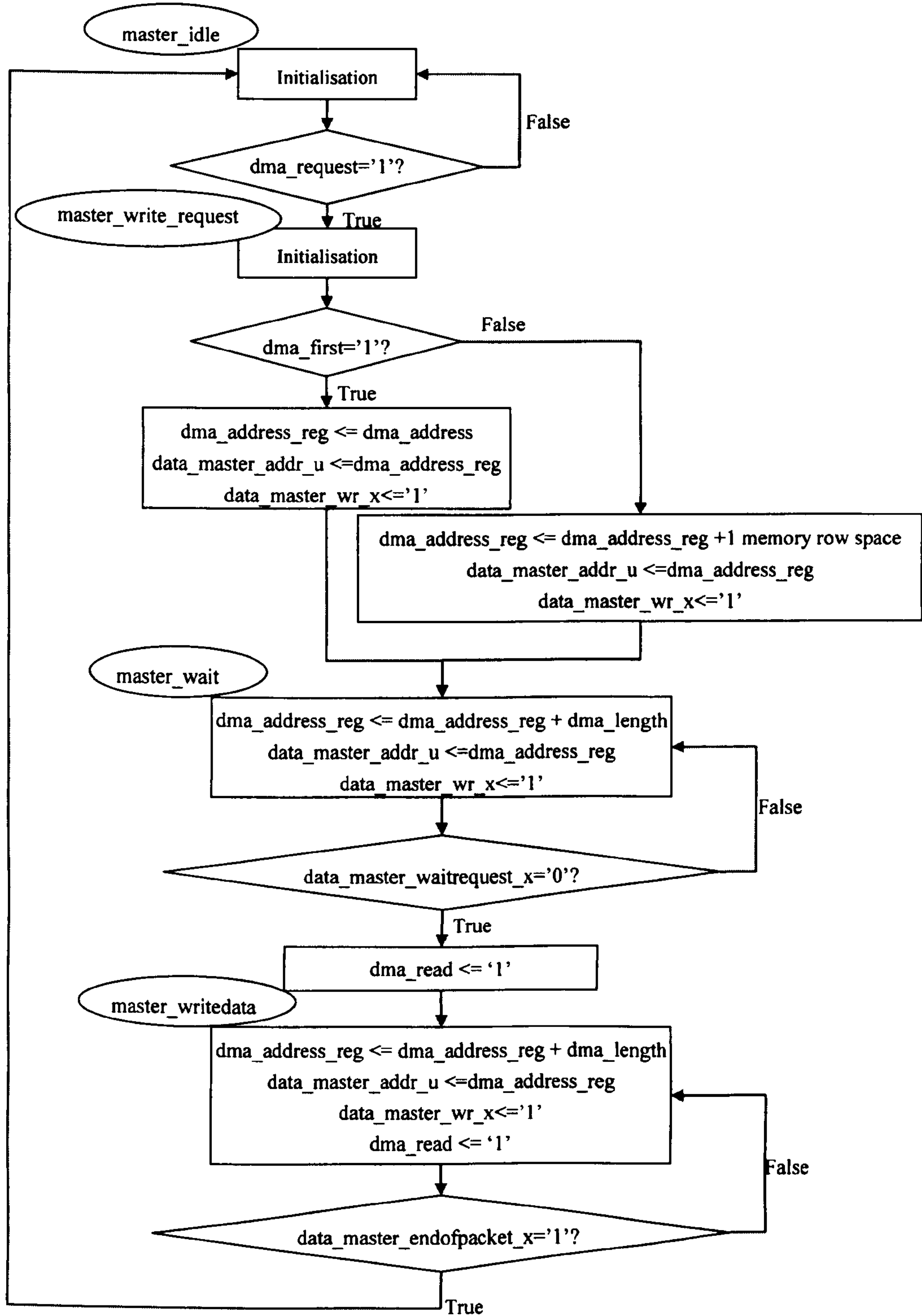


Figure 5-16 ASM chart of the Avalon streaming master write transfer

This streaming write transfer always sends out data of a complete video line. The length of it is determined by how long the input timing signal *LVal* is asserted. For example, if the pixel clock is 20MHz while *LVal* is asserted for 32 μs, then a horizontal resolution can be calculated as 640. The streaming transfer length is therefore 80 in 8-bit monochrome mode or 320 in 24-bit RGB mode.

The start address where a video frame starts being written into in the memory is specified in *dma_address*. When sending the first video line out the Avalon interface block passes the *dma_address* to *data_master_addr_u*, and then it increases this data master address by one memory row space in every write transfer until the bottom line is sent. Signal *dma_first* indicates whether a new write request is for the first frame line or not.

5.4.3.2. Video receiver

The video receiver mainly buffers the incoming video data into two temporary line buffers when timing control signal frame valid *FVal*, line valid *DVal* and data valid *DVal* are in active state, and sends out the buffered data to the Avalon interface module. Figure 5-17 shows a block diagram of this video receiver.

As seen in the figure two state machines are used to decode these timing signals. Besides buffering the video data, the video receiver also decodes the CPU address and uses the decoded registers to control and configure this video receiver. The next few paragraphs focus on describing each function block of the video receiver in details.

Local slave interface

Two registers have been provided to control and configure the video capture controller. They are the control register and DMA address register. The control register, which includes a start bit , a trigger bit and an interrupt enable bit. The start bit is used to trigger the two state machines to start checking the input timing signals and buffering the video data. The trigger bit is used to drive the output signal *Trig* (Camera control 3, *CC3*) to trigger the camera when it is in software trigger mode.

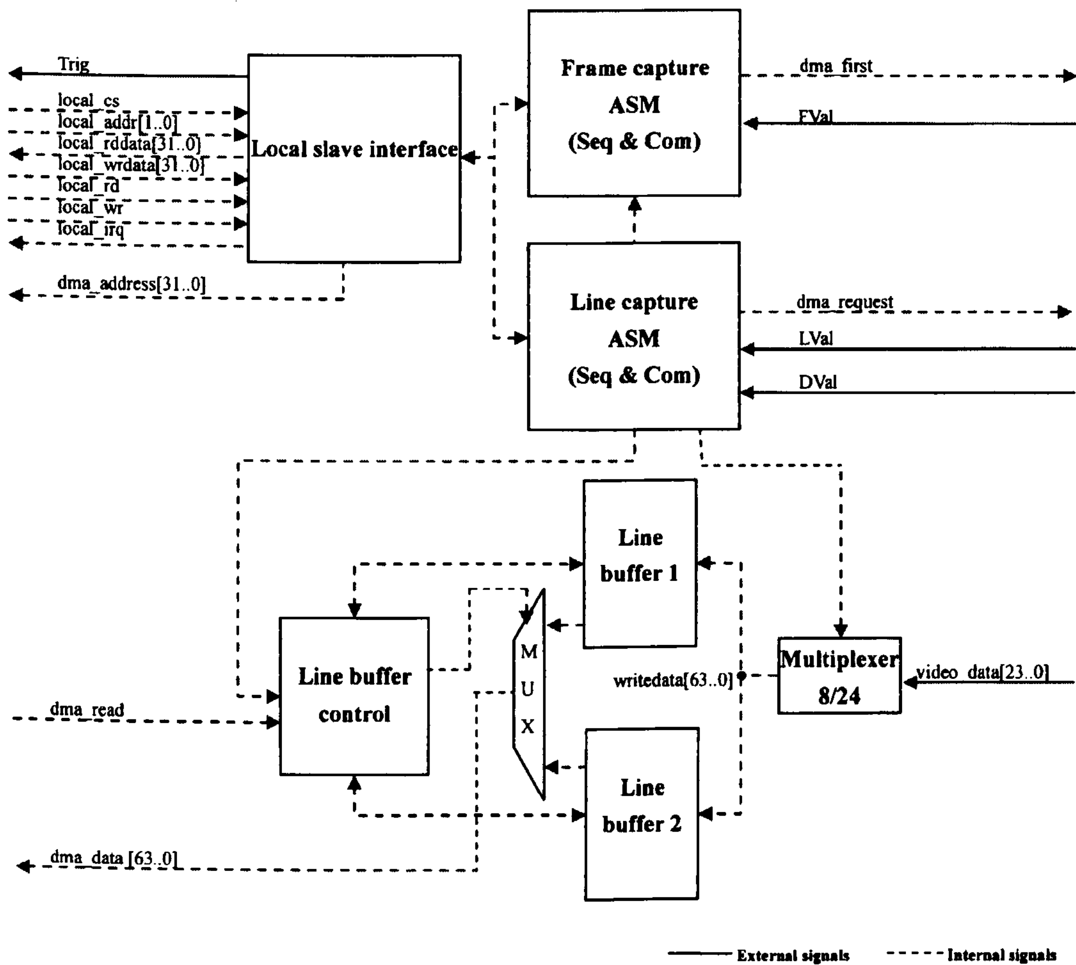


Figure 5-17 Block diagram of the video receiver

When a video frame has been captured and stored in the main memory an interrupt is generated. If an interrupt is enabled then it must be cleared before carrying on capturing the next frame to prevent stalling the whole system. This interrupt is cleared upon writing to the DMA address register. By writing different bank start address into this DMA address register during the video blanking period a real-time processing can be achieved.

Frame capture & Line capture ASM

These two ASMs are driven by input *FVal*, *LVal* and *DVal* respectively. Figure 5-18 shows a timing diagram of both *FVal* (frame enable) and *LVal* (line enable). *DVal* is not actually used in the current system as the camera only provides two control signals.

When they are both valid the line capture ASM sends a write enable signal to the line

buffer control block to enable one of the line buffers to start recording the incoming video data. When *LVal* deactivates the line capture ASM sends a buffer swap request to the line buffer control block as well as a DMA request to the Avalon interface module.

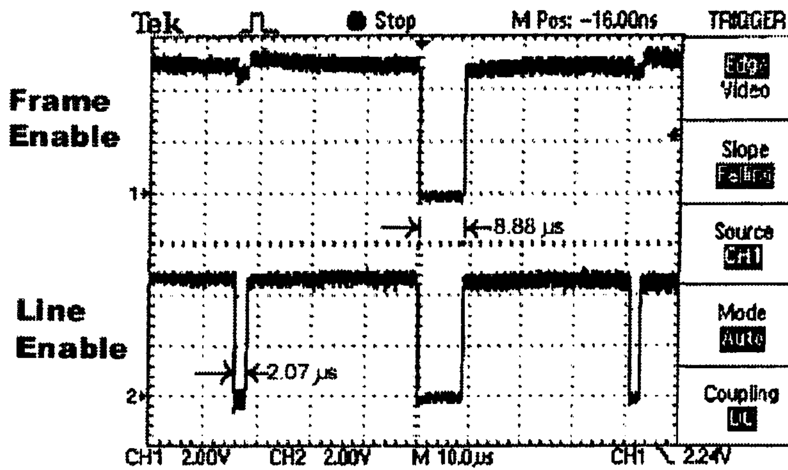


Figure 5-18 Camera timing (From [63])

Line buffer 1 & 2

These two line buffers are identical dual-port ram with a size of 512x64 or 128x64 depending on the video mode. Each memory location stores either 8 pixels in 8-bit monochrome mode or 2 pixels in 24-bit RGB mode. The maximum number of pixels each of them can store is 1024. They perform a read and a write in an alternating order. The line buffer control block controls this sequence. The write port of these two buffers is driven by *capture_clock* while the read port is driven by the *data_clock*. The write and read address are counted in the line buffer control block. In order to increase the system performance, the I/O interfaces of these two buffers are all synchronous.

Line buffer control

This line buffer control block provides a mechanism to switch the operating mode of the two line buffers in terms of the video timing. This switch happens during the inactive period of *LVal*. The Avalon interface block always operates on the buffer which is not being written with new video data. By doing this, it can ensure the video data sent to the memory is always the video line just captured.

5.4.4 Camera serial controller (UART)

In a standard CameraLink communication protocol, the camera is able to be controlled

and configured though a serial communication protocol. The serial protocol used in SIPS is full duplex, asynchronous, 8 data bits, 1 start bit, 1 stop bit and no parity. Baud rate is fixed at 9600. Signal *SetTC* is used to carry the serialised command message generated by the processor to modify the operation of the camera [63]. When the camera receives a status request message it returns the appropriate values in a response message in a serial format via signal *SetTFG*. A UART, which is a type of "asynchronous receiver/transmitter", was used in this system to handle communication between the processor and the serial protocol. It is an Altera's provided IP component so it can be integrated into the SOPC design. More information of how to use and configure this UART controller can be obtained in [80].

5.4.5 Summary

The video capture controller designed in this system is based on the CameraLink specification. It has two main modules which are the Avalon interface module and video receiver module. CameraLink signals frame valid *FVal*, line valid *LVal*, data valid *DVal*, 8/24-bit *video_data* and 1 bit *capture_clock* are all fed into the video receiver module for processing. By decoding the 3-bit enable signals the video capture controller is able to capture all valid video data losslessly and save them all to the memory device. Captured video data are firstly stored in one of the two temporary line buffers, when this buffer is filled up by one video line, all data stored in it will be sent to the main memory by Avalon streaming write transfers. The Avalon interface module controls these kinds of transfers. These two line buffers are performed read and write alternatively.

In order to allow the processor to control the capture controller, an additional Avalon bus slave is provided. The processor can start video capturing, specify DMA start address and enable interrupt by writing to the capture controller. It can also be informed of an interrupt when a whole frame is captured and saved into the memory.

This capture controller can be conditionally generated into either 8-bit monochrome mode or 24-bit RGB mode. In 8-bit monochrome mode, the size of the line buffers is 128x64, and each buffer location stores 8 video pixels. The burst length of a streaming write is eight times less than the total number of pixels in one video line. While in 24-

bit RGB mode, the size of the line buffers is 512x64, each buffer location stores 2 video pixels. The burst length is two times less than the total number of pixels in one video line.

In order to control and configure the CameraLink camera via a serial control protocol, an Altera's IP component – UART, was used. By using this UART the processor is able to transmit configuration messages to the camera and receive the status information from the camera.

5.5 Cache

In SIPS, the Cache is a communication medium to allow the Nios processor to communicate with the main memory indirectly. It also stores frequently used data for the processor to rapidly access. In this section, it firstly introduces the main features of this custom IP component, and then it gives an overview of how the Cache works. Thirdly it describes the implementation details of the Cache. Finally a summary is given.

5.5.1 Main features

The typical features of the Cache are:

- A direct mapped Cache
- Write-though scheme implemented
- Cache depth is the column size of the SDRAM device which is 512
- Each Cache memory location stores 64-bit valid video data
- Support 32-bit Nios processor
- Support standard Avalon bus interface
- Configurable Avalon streaming read transfer length of up to the maximum Cache depth

5.5.2 Overview of the Cache

The main video data storage device used in this system is a dynamic RAM which requires a longer access time (for example, it is required to activate a bank and row before doing R/W on a specific column) than other types of memory devices such as SRAM. If there is some data that the processor uses frequently then it would be

inefficient for the CPU to access the main memory for the same data each time. Moreover, as the video memory is shared by multiple masters, frequently accessing the memory slave for the same data would result in creating more stalls for the other masters, and consequently the whole system speed would be degraded. However, by placing a Cache between the processor and the main memory device, this problem can be solved because the Cache is a temporary storage area where frequently accessed data can be stored for rapid access [83].

Figure 5-19 shows how the Cache is mapped into the main memory. It is a Direct Mapped Cache which is also referred to as 1-Way set associative cache. The size of the Cache designed in this system is equal to the number of columns in one row of the main memory. Therefore a block of data in the main memory is always loaded into the same cache line in the cache. For example, column 0 of any row in any bank in memory must be stored in row 0 of the Cache memory. If column 0 of row 0 is stored within the cache and column 0 of row 1 is requested, then column 0 of row 0 will be replaced with column 0 of row 1. Direct Mapped cache only requires that the current requested address be compared with only one cache address.

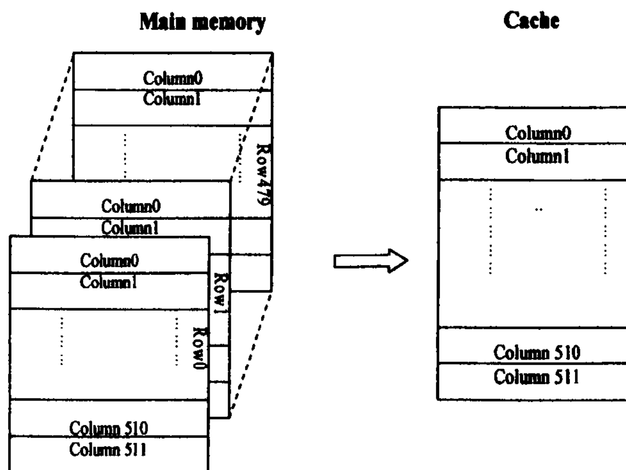


Figure 5-19 Cache mapping to the main memory

Data stored in the Cache contains three fields (see Figure 5-20). The first field is a single bit which is a flag indicates if the data field has actual data or not. It would be useful when the Cache is just started up and nothing stored in it. The second field, which is called Tag, stores the SDRAM row and bank address information. So by addressing the entry of the Cache and reading the tag the corresponding main memory address can be identified. If the tag is equal to the bank and row address specified in

the CPU address then it is a Cache hit, the information stored in the last field, which is the actual data will be returned to the CPU without accessing the main memory during a CPU read. Otherwise, it is a Cache miss; the Cache master(s) will send a read request to the main memory to request the data. The returned data will also be saved in the Cache memory associated with the address information.

A Cache write is much simpler than Cache read. If a CPU write occurs, it updates the corresponding field in the Cache with the write data and also writes that data to the main memory directly.

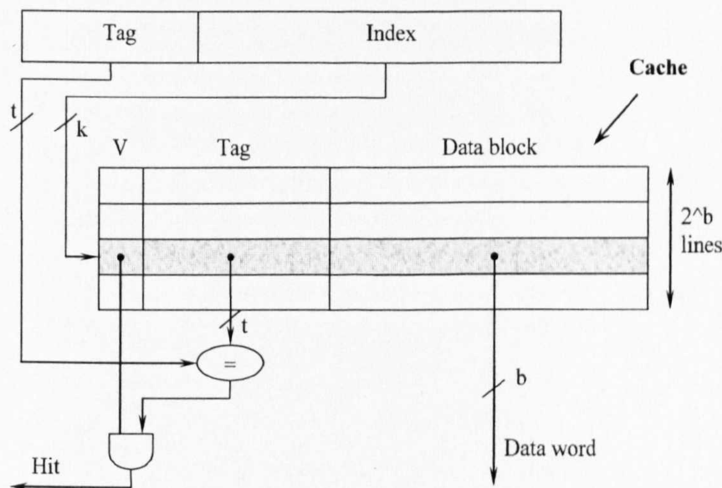


Figure 5-20 Cache structure (From [83])

5.5.3 Description of the Cache

The Cache communicates with the processor and the memory controller, so it has three Avalon ports which include two Avalon streaming master ports and one simple Avalon slave port (see Figure 5-21). The Avalon slave port is mastered by the Nios processor, which allows the processor to read data either from the Cache memory (if it is a Cache hit) or from the main memory (a Cache miss), and write data to the main memory. The other two Avalon streaming master ports have the same usage as the ones in the video display controller and the capture controller. The only difference is the streaming master ports in the video display controller are read-only masters; the ones in the capture controller are write-only masters while the Cache's are read & write masters.

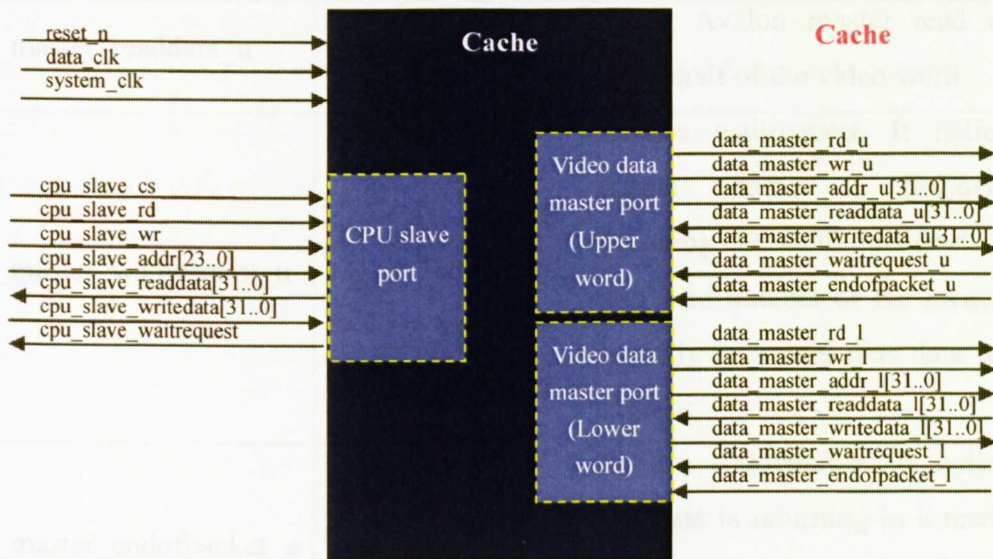


Figure 5-21 Block diagram of the Cache

Table 5-21 lists the top level I/Os of the Cache.

Table 5-21 Cache top level signals

Signal	Width	Direction	Description
reset_n	1	Global	Global reset, active low
data_clk	1	From clock generator	video data transfer clock
system_clk	1		System clock, drives Avalon transfer initialised by the Nios processor
cpu_slave_cs	1	From Nios processor	Avalon slave chip select, active high
cpu_slave_rd	1		Avalon slave read enable, active high,
cpu_slave_wr	1		Avalon slave write enable, active high,
cpu_slave_addr	24		Avalon slave address
cpu_slave_writedata	32		Avalon slave write data (video data)
cpu_slave_readdata	32	To Nios processor	Avalon slave read data (video data)
cpu_slave_waitrequest	1		Avalon slave controlled waitrequest
data_master_rd_u	1	To video memory controller	Avalon master read enable, active high
data_master_wr_u			Avalon master write enable, active high
data_master_addr_u	32		Avalon master address
data_master_writedata_u	32		32-bit Avalon master write data, the upper half of the video word

data_master_readdata_u	32	From video memory controller	32-bit Avalon master read data, the upper half of the video word
data_master_waitrequest_u	1		Avalon waitrequest. It indicates the master address is finished being decoding and valid data starts returning in a read transfer or the memory slave is ready to accept new data in a write transfer
data_master_endofpacket_u	1		Avalon endofpacket. It indicates the last data is returning in a read transfer or the slave is accepting the last write data in a write transfer
data_master_rd_l	1	To video memory controller	Same as *data_master_rd_u*
data_master_wr_l	1		Same as *data_master_wr_u*
data_master_addr_l	1		Avalon master address (not used)
data_master_writedata_l	32		32-bit Avalon master write data, the lower half of the video word
data_master_readdata_l	32	From video memory controller	32-bit Avalon master read data, the lower half of the video word
data_master_waitrequest_l	1		Same as *data_master_waitrequest_u*
data_master_endofpacket_l	1		Same as *data_master_endofpacket_u*

The width of the CPU address bus *cpu_slave_addr* is 24, which is one bit wider than the SDRAM address. It is because the data width of the Nios processor is 32. So the lowest significant bit (LSB) of the CPU address is used to select the upper 32-bit or the lower 32 bit word. For example, CPU address of 0x0000002 means it is operating on the lower 32-bit word at SDRAM address 0x0000002 while CPU address of 0x0000003 indicates it is accessing the upper 32-bit word at the same SDRAM address.

Figure 5-22 shows the internal structure of the Cache.

The core of the Cache is two identical dual-port rams with size of 512x48. They are the actual Cache memory which stores the information of the V flag, Tag and data. They

share the same Cache entry. Each of them handles 32-bit video data so that the CPU can access the lower or upper 32-bit data separately in one 64-bit Cache location. The LSB of the CPU address is used to select which ram to be operated. Both of these rams have synchronous port interface to increase the system performance. The write port of these two rams is driven by the data clock while the read port is driven by the system clock. Both the CPU write data and data master read data can be written into the Cache rams via the write port. The read port always responds to the CPU read command.

There are four function blocks to handle all Cache operations as seen in Figure 5-22. The next few paragraphs describe how Cache operations including Cache write, Cache miss and Cache hit are handled by these four function blocks.

Cache write

When CPU initialises a write request, the CPU slave write-control block immediately holds up the processor by asserting *cpu_write_waitrequest*. It also triggers the data master write-control process to start writing into the main memory and the Cache memory by asserting *writerequest* as well as the other control signals. This data master write-control process is similar as the one described in the video capture controller. However, the Cache can only do single write with 32-bit data each time. The CPU doesn't need to wait until the Avalon write transfer to the memory finishes. Once the data masters are granted access to the memory it acknowledges the CPU slave write-control block by asserting *writegranted*, then CPU is released and Cache write completes. The reason of waiting for access granting is if an ongoing write to the memory slave is being held for long enough due to conflicts in accessing the memory by multiple masters (see Chapter 6) so that CPU can initialise another write request, then the second CPU write would be likely ignored because the data master write-control process is still processing the first request.

Cache read

When CPU initialises a read request, the CPU slave read-control block immediately holds up the processor by asserting *cpu_read_waitrequest*. It then waits for the read data from one of the two Cache rams become valid. When valid data returned it checks the V flag and compares the Tag field with the corresponding bits in the CPU address line. If it's a Cache hit, then the CPU slave read-control releases the processor in the

next system clock cycle, the Cache read with hits then completes. Otherwise, it asserts *readrequest* to trigger the data master read-control process to start reading data from the main memory. The data master read-control process is similar to the one described in the video display controller. Once the Avalon streaming read request is fulfilled and all requested data are written into the Cache rams, the CPU slave read-control block releases the CPU and the Cache read with miss completes. Figure 5-23 illustrates how the Cache read is handled in terms of hit and miss by using an ASM chart.

5.5.4 Summary

The Cache designed in this system is a direct mapped Cache. Each Cache line can be mapped into specific column in any row of any bank in the main memory. The entry of a Cache line corresponds to the column address to the main memory. When the CPU initialises a Cache read, it always checks the specific Cache line whether it contains the valid data that CPU requires by checking the Tag and V flag. If it is a Cache hit then the data is returned to the CPU immediately and the CPU read request terminates, otherwise it is a Cache miss then an Avalon streaming read request is sent to the main memory, the CPU is stalled until the required data is returned from the main memory. This data is also saved into the Cache. Cache write uses 'write-through' scheme. When CPU writes to the Cache it also writes to the main memory. However, the CPU doesn't wait for the completion of the write transfer to the main memory unless the Cache write transfer is suspended due to conflicts in accessing the memory device (see Chapter 6). By doing this it can save approximately half of time required for a CPU write.

The Cache communicates with the data master of the Nios processor, however, because the CPU data master is not a latency-aware master [45], the Cache CPU slave couldn't implement Avalon transfer with latencies [35] which can possibly increase the CPU efficiency of reading data from the memory and give the CPU more time to do other operations while waiting for valid data returned.

In order to maximise data usage efficiency for some specific image processing algorithms which requires continually reading a block of data from a single video line, the length of a Cache read with misses is configurable, which means the Cache can be filled up by up to a memory row's data in a single streaming read transfer. Therefore if

CPU needs to read data from the same row continually and without switching to the other rows before finishing the current one, then every Cache read would be a hit except from the first one. This feature is useful for algorithms such as image inversion and correlation which requires continuously processing data from the same image line before going for the next one. The last memory address is reserved for the CPU to configure the burst length. This address is 0xFFFFFF

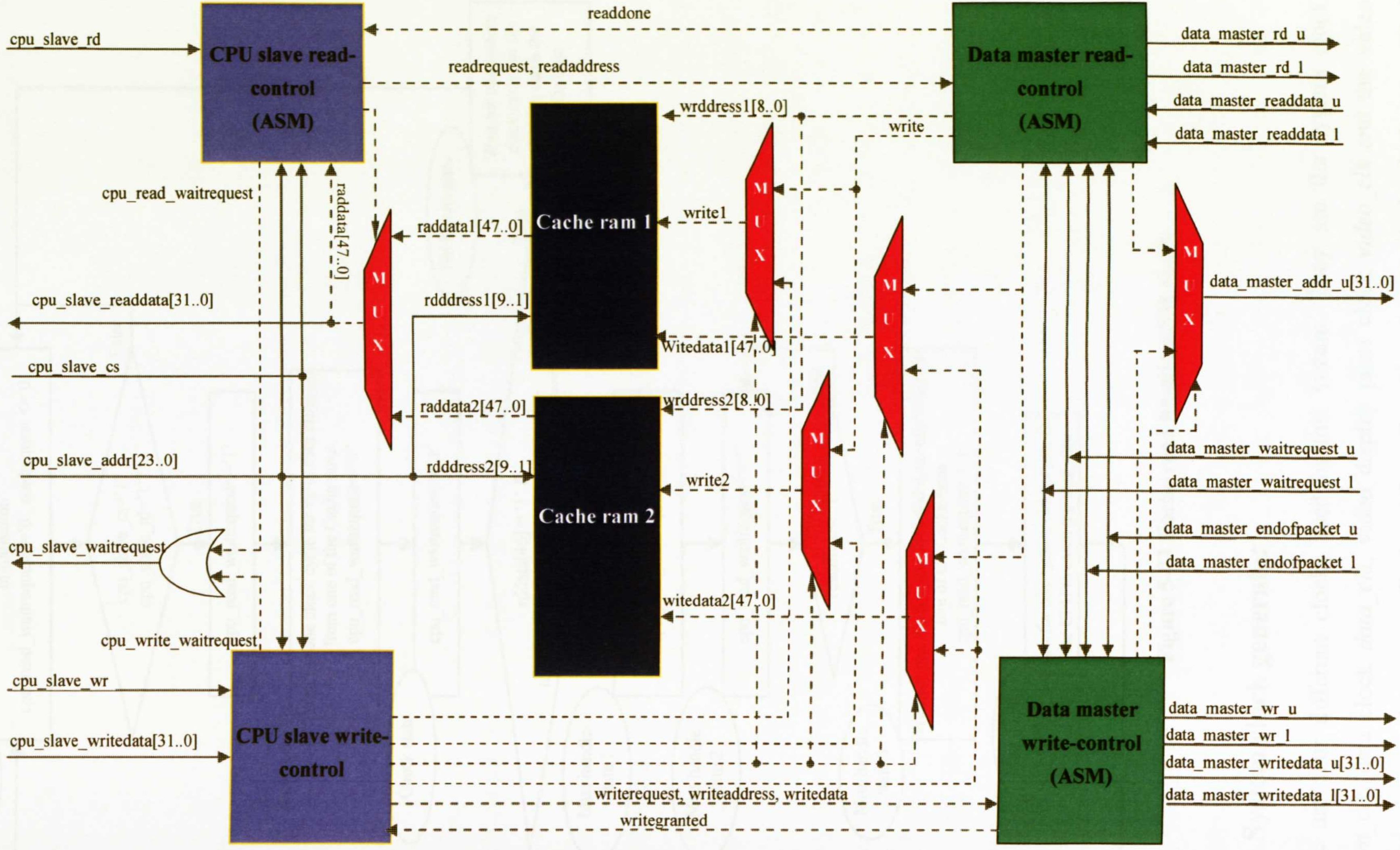


Figure 5-22 Schematic drawing of the Cache structure

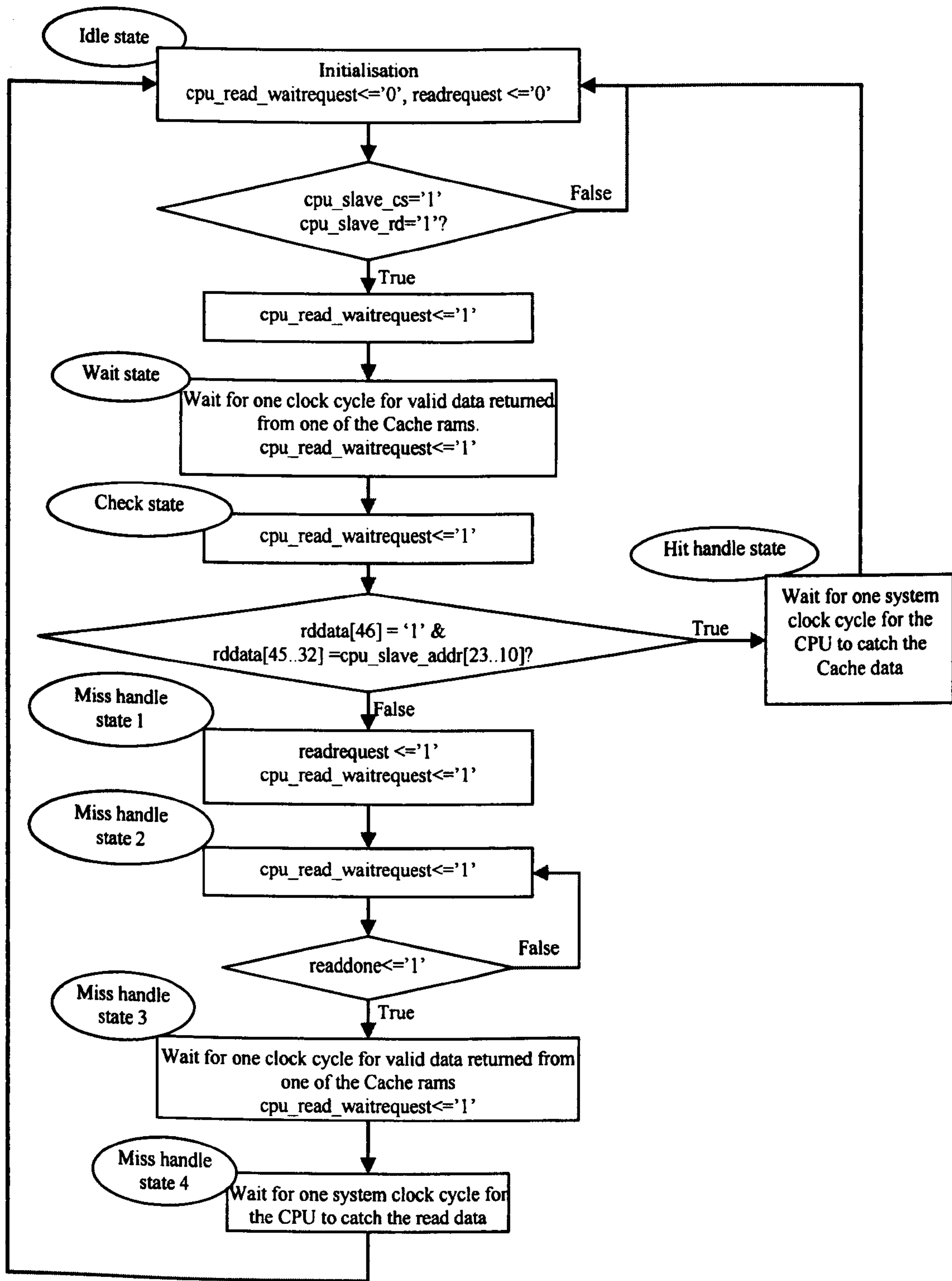


Figure 5-23 Cache read handle ASM chart

5.6 System clock generator

There are four different clocks used in this system. They are the system clock *system_clk*, data clock *data_clk*, video display pixel clock *video_clk* and the video capture pixel clock *capture_clk*. Apart from the video capture clock which is supplied

externally, the other three clocks are generated from phase-locked loops (PLLs) [84]. Figure 5-24 shows a connection diagram of the PLLs with global clock pins in the given FPGA device.

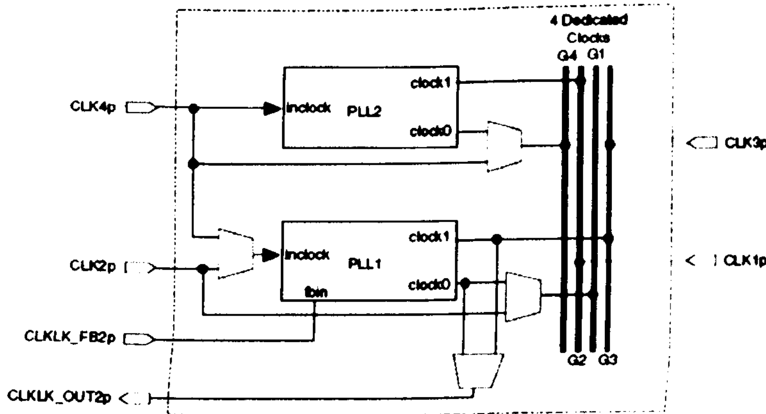


Figure 5-24 Dedicated global clock pin connections to PLL & dedicated clock lines for EP20K30E, EP20K60E, EP20K100E, EP20K160E & EP20K200E devices (from [84])

5.6.1 Data clock

All data transfers involved with accessing the main memory are driven by this clock. For example all Avalon streaming R/W transfers in this system, the whole memory controller, the write port of the line buffers in the display controller and the read port of the line buffers in the capture controller. This clock is twice faster than the system clock in order to increase the data transfer speed. In an Avalon master or slave it is possible to use two different clocks as long as the corresponding master and slave ports are both clocked by the same speed and no pre-defined Avalon fixed wait-state involved. This data clock is also output to drive the SDRAM device. This clock has a frequency of **100MHz**. The reason for choosing this frequency is because most of the SDRAM devices support this high frequency and this is the frequency the PLL can generate and the memory controller can run at.

The data clock is generated from a PLL driven by an on-board oscillator with a frequency of 33.333MHz. Because the Nios development board already ties the clock pin on the SODIMM socket to pin CLKLK_OUT2p on the FPGA device which is the clock output from this PLL, this data clock must be assigned to that dedicated pin CLKLK_OUT2p. To generate a 100MHz it just needs to multiply the input clock by a factor of three.

5.6.2 System clock

This clock is used to drive the Nios processor, the Avalon CPU slave ports in all video IP components and all Altera provided peripherals. The read port of the Cache memory is also driven by this clock. This system clock has a frequency of **50MHz** which is twice slower than the data clock. The reason of choosing this frequency is because this is closed to the marginal frequency that the Nios processor can run safely at according to the reference design.

As there are a lot of logic paths involved with the data clock and the system clock, the system clock must be phase aligned with the data clock to avoid potential timing failures and increase the system performance. For example, in the Cache the CPU R/W requests can drive the streaming Avalon transfers which operated at 100MHz. In order to make this happen the system clock must be generated from the same PLL because all PLL output clocks have the same phase delay against the input clock and most importantly, the system clock and the data clock have an integer multiple relation. To generate a 50MHz clock it just needs to multiply the input clock with a factor of three and then divide the result by two.

5.6.3 Video display clock

This is the pixel clock to drive the video display. The read port of the line buffers in the video display controller is also driven by this clock. According to the VGA timing information given in Table 5-13, the frequency of the pixel clock should be $640 \div 25.17us \approx 25MHz$ if the image is horizontally sized at 640.

Ideally this clock should be generated from the same PLL with the data clock and system clock. However, as seen from Figure 5-24 there are only two clock outputs available from each PLL in the given device. Furthermore the same clock source can not drive more than one PLL on this particular device. It has no other ways to get the video display clock phase aligned with the other two. Since so, the video display clock can be generated by a counter which divides either the system clock or the data clock down to 25MHz. However this could result in long clock skew and high fan-outs for the clock that the counter divides, consequently this would affect the system speed. Another option is to use the second PLL (there are two available on this device) to

generate the video display clock. The source clock is in fact the output data clock, as on the development board the source clock of that PLL has been hard wired to the CLKLK_OUT2p pin as shown in Figure 5-25. So the display clock can be simply achieved by dividing the input clock by four.

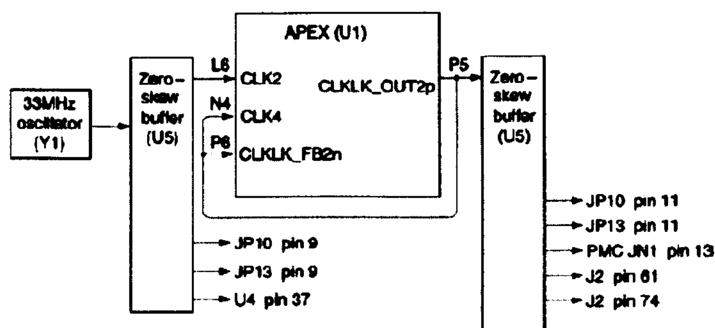


Figure 5-25 Clock circuitry (From [58])

5.6.4 Video capture clock

This clock is supplied by the CameraLink camera and fed into the video capture controller. This clock can be programmed to be 40MHz, 20MHz, 10MHz and 5MHz with default value of 40MHz based on the required exposure time. Which one therefore should be chosen? Ideally the video capture should be synchronised with the display and have the same frame rate to ensure there is no frame missing for display and the same frame isn't displayed multiple times. The clock which results in the closest frame rate of the video display should be chosen. As shown in Figure 5-18 the capture line blanking is 2.07 us and the vertical blanking is 8.88 us. When the pixel clock is 40MHz the frame rate is $1/(480 \times (640 * 25\,ns + 2.07\,us) + 8.88\,us) \approx 115.17\,f/s$. By using the same calculation, it can get the frame rate is 61.11 f/s at 20MHz, 31.52 at 10MHz, 16.01 at 5MHz. 61.11 is the closest number to the video display. Therefore **20MHz** was chosen as the capture pixel clock and will be used for analysing the system efficiency in Chapter 8.

5.6.5 Discussions

The data and system clock frequencies given in this section are just design requirements. The actual clock speed could be different depending on the whole system performance after synthesis. Synthesis results are presented in Chapter 7.

The video display clock is generated from the second PLL, but can the system clock be generated by that PLL instead while using the first PLL to generate the display clock, or generating both the system clock and display clock from the same PLL. The answer is not ideal because there are more paths involved with the system clock and the data clock especially in the Cache than the data clock and display clock. Moreover, the system clock tends to have more chances to be in metastability [85] state than the video display clock because it has a higher frequency. If the system clock and the data clock were not phase aligned, it had more chances to cause synchronisation fails.

5.7 Explanation of some design issues

This section explains some typical design issues for overall system implementation and optimisation which may or may not have appeared in the previous sections.

5.7.1 Multiple-bank operation

Because the video capture, video display and the Nios processor are operated at different clock frequency, it must ensure the display controller doesn't display unprocessed data, or the video capture controller doesn't capture raw video data into where are being processed or displayed. One of the solutions is to use multiple-bank operation. The idea of this is to split the memory device into multiple banks (or multiple different locations in a particular area in the memory) and the size of each bank is one video frame. Each bank performs one task at a time. For example, it can have a capture bank for storing the raw data, and a display bank for storing the processed data. Operations on each bank changed only if specific condition satisfied, i.e., the capture bank is full so this bank will be used for display next. For the benefit of real-time processing, only triple-bank and quad-bank operation [18] are presented here.

5.7.1.1. Triple-bank operation

If the processing is to be superimposed on captured image, for example the feature correlation, (see section 8.2) then the triple-bank operation is needed. In this mode, the memory is split into three banks, one bank stores the captured image data, one bank is used to process the captured data, and one bank is used to display the processed data. These three video banks are placed in rotation therefore.

Figure 5-26 demonstrates how this triple bank mode works. This bank sequence moves forward only when the captured bank is filled up, and the CPU finishes processing on the process bank. So for example, when the display bank is done however the capture bank is still in process then the next display frame would be still the old one. The video capture and display controller generate separate interrupt to the processor when they finish operating on the specific bank.

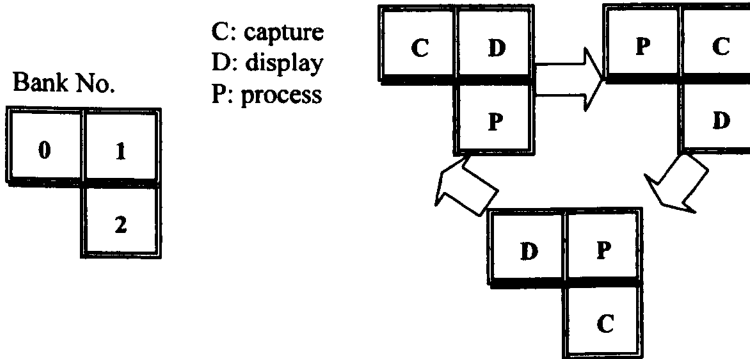


Figure 5-26 Triple-bank operation

Under this mode, the total memory bits taken up in the SDRAM is $640 \times 480 \times 8 \times 3 \approx 7.03 Mbits$ in 8-bit monochrome mode or $640 \times 480 \times 24 \times 3 \approx 21 Mbits$ in 24-bit RGB mode.

5.7.1.2. Quad-bank operation

If the displayed image contains little or no raw image, for example displaying real-time control graphics, then the quad bank mode of operation is utilised. Figure 5-27 illustrates how quad-bank operation is implemented.

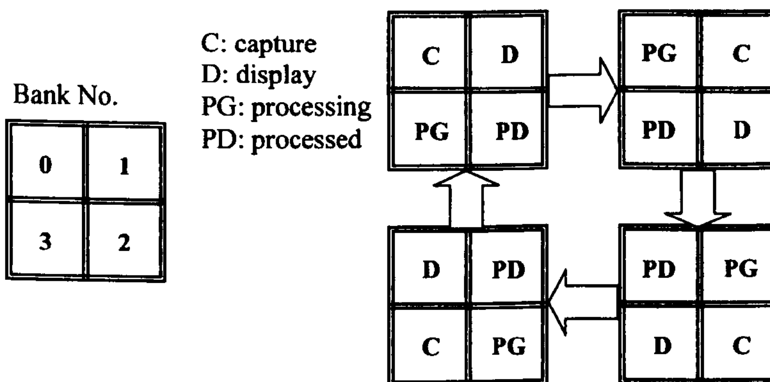


Figure 5-27 Quad-bank operation

Under this mode, the memory is split into four banks. They are the capture bank, display bank, processing bank and processed bank. The processed bank stores the destination image generated from the source image in the processing bank. This

operation can be exploited for a pure graphical display removing the previous data and updating for each image such as digital filter implementation (see section 8.2). This mode would take up the memory $640 \times 480 \times 8 \times 4 \equiv 9.375 Mbits$ in 8-bit monochrome mode or $640 \times 480 \times 24 \times 4 \approx 28.125 Mbits$ in 24-bit RGB mode.

5.7.2 Double line buffer in the video display & capture controller

The purpose of using line buffers is to synchronies the data paths between the memory and the video display or capture interface and increase the data transfer efficiency because the video pixel clock frequency is slower than the data transfer frequency. For the video display, it must ensure the line buffer is filled up with new data before being sent to display. While for video capture, a video line must be finished capturing into the memory before the next one is ready.

In VGA mode, the gap between every active video line period is 31.77-25.17=6.6μs. With one line buffer, a video line must be transferred fully into the line buffer during this gap. Ignoring the overhead time, an Avalon streaming read transfer takes 3.2μs in 24-bit RGB mode and 0.8μs in 8-bit grey-level mode, if the data clock is 100MHz. It looks it is OK as it doesn't exceed the video gap. However, this memory is shared by multiple masters which include the Cache and the video capture controller. Obviously the worse case for the video display is in 24-bt mode, those masters try to access the memory at the same time and the display controller gains the least priority and the Cache requests a transaction of a whole video line. Then plus the pending time, the total time that the display controller needs to wait until it receives the whole requested video line would be 3.2 × 3=9.6μs. It has greatly exceeded the time gap 6.6 us. Furthermore, the data clock might not be able to be satisfied at 100MHz, if that happens, the transferring time would become even longer. It's therefore concluded one line-buffer mode is unsafe. What about double line buffer? Even when the worse case occurs, there is still 31.77μs sufficient time for it to complete the whole transaction.

As for video capture, Figure 5-18 has already indicated the time gap between every active video line is 2.07μs, and this even couldn't satisfy the timing requirement of transferring one video line in 24-bit mode. So double line buffer for the video capture is required.

5.7.3 Synchronisation of multiple clock domains

Section 5.6 has described that the video display clock is not phase aligned with the other two clocks due to the lack of clock outputs from one PLL. Also, because the capture clock is supplied from the external camera, it is not phase aligned with the system and data clock either. In this multiple clock domain system, synchronisation failure can occur when a control signal generated in one clock domain is sampled too close to the rising edge from another clock domain. This sampled output would be in metability state which can cause illegal signal values to be prorogated throughout the rest of the system as shown in Figure 5-28.

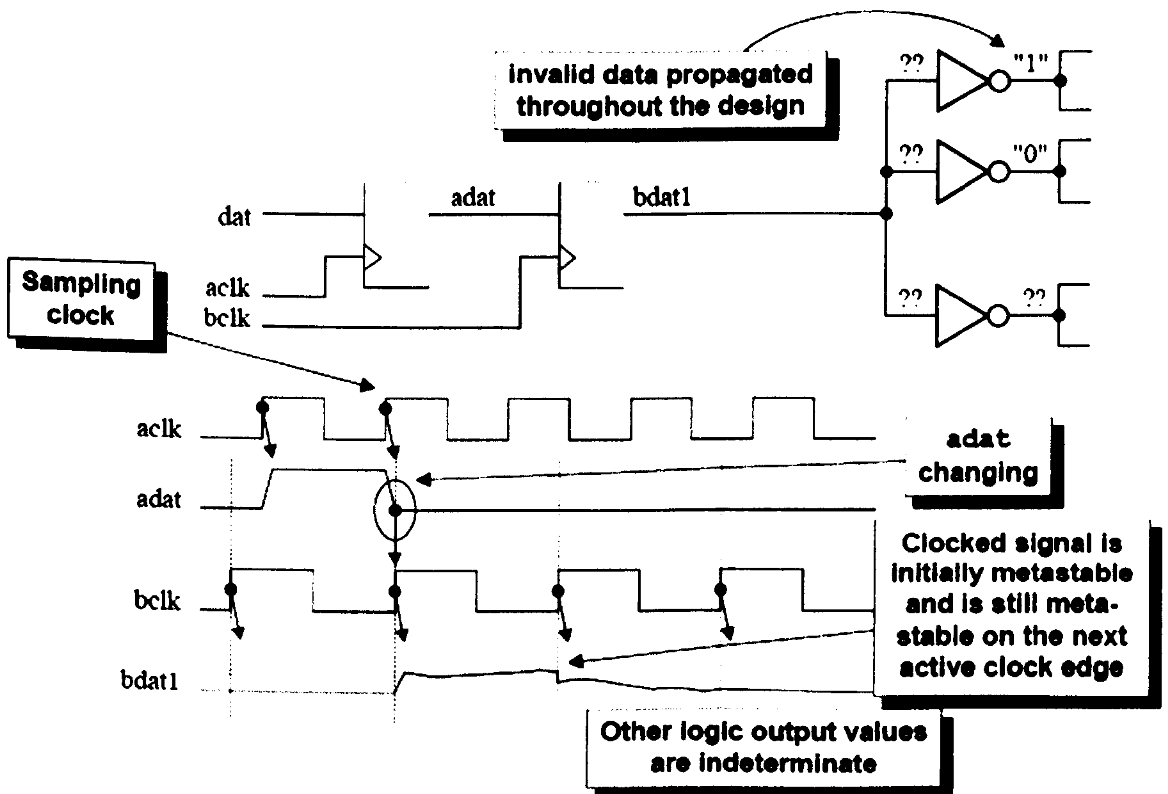


Figure 5-28 Metastable output propagating invalid data throughout the design (from [86])

The most common solution is to use two-flip-flop synchroniser as shown in Figure 5-29. As seen from the figure the output from the first flip-flop is in metastability state, however, after a full clock cycle to permit any metastability on the metastable output signal to decay, when it comes to the second flip-flop it becomes stable and valid signal is synchronised into the new clock domain. It is theoretically possible for the first flip-flop output to still be sufficiently metastable by the time the signal is clocked into the second stage to cause the second flip-flop output to also go metastable. However, for

most synchronisation applications, the two flip-flop synchroniser is sufficient to remove all likely metastability [86].

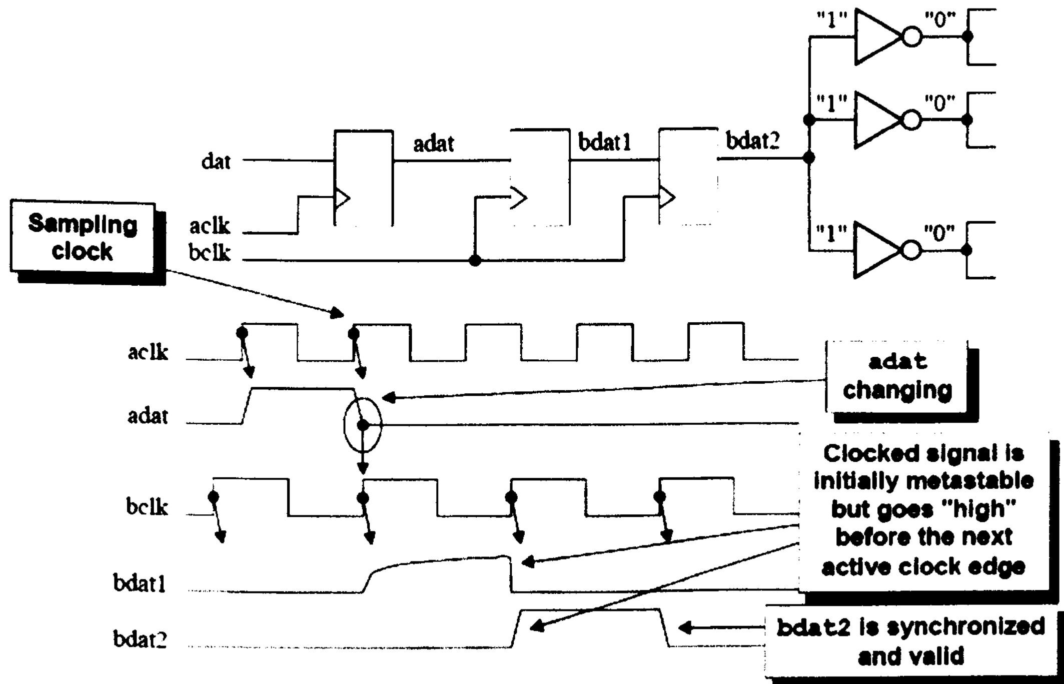


Figure 5-29 Two flip-flop synchroniser (from [86])

5.7.4 Other issues

In the current design of SIPS, it utilises many pipelining (as mentioned in section 3.1.2 pipelining means inserting registers to re-clock the data between logic layers) on the data paths including the SRAM interface and Avalon interfaces to optimise the system performance (f_{max}). Also, all embedded memory blocks have synchronous read/write ports. However, all of these operations increase the latencies and hence decreases the processing efficiency. This could be subjected to be optimised after synthesis if the specific FPGA device can meet the timing requirements.

Chapter 6. Simultaneous Multi-mastering Avalon Streaming Transfer with Peripheral-Controlled Waitrequest

In the Nios embedded image processing system described in Chapter 4 and Chapter 5, the capture device, display device and Cache can directly access the memory slave as a streaming transfer via a dedicated I/O bus. In this bus protocol the video memory is always acting as a slave peripheral and shared by these three masters, appearing like a "triple-ported" slave (Figure 6-1). These configurations maximise the efficiency of the data transfer compared to polling and I/O interrupts [83] [87] commonly used for lower bandwidth devices. Consequently the system performance is increased by giving more time to the processor for processing the video data.

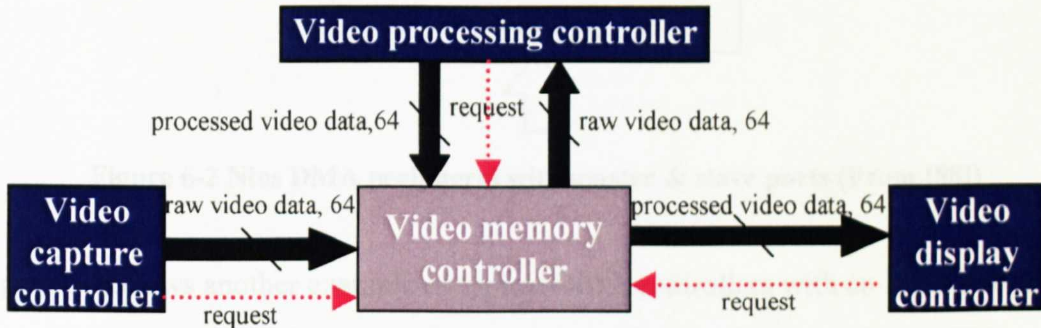


Figure 6-1 Triple-ported video memory slave

However, a problem is raised if more than one master on the bus (known as multi-mastering) requests streaming transfer from/to the memory slave at the same time, which one should be granted access to the memory first and how to force the others to wait? This could be understood as a bus arbitration problem which is deciding which bus master gets to use the bus next [83]. But before explaining this bus arbitration issue, this chapter will review another solution of using conventional DMA controllers.

6.1 Synchronisation of DMA controllers solution

A typical DMA controller contains not only data master(s) to direct the reads or writes between itself and memory [83], but a configuration port to allow the processor to setup the DMA controller such as specifying the memory address that is the source or destination of the data to be transferred, the number of bytes to transfer and enabling the DMA controller etc.

Figure 6-2 shows an example of typical DMA controllers. It is the Nios DMA module, which is an Altera SOPC Builder library component included in the Nios development kit [88]. This DMA controller has two Avalon master ports—a master read port and a master write port—and one Avalon slave port for controlling the DMA.

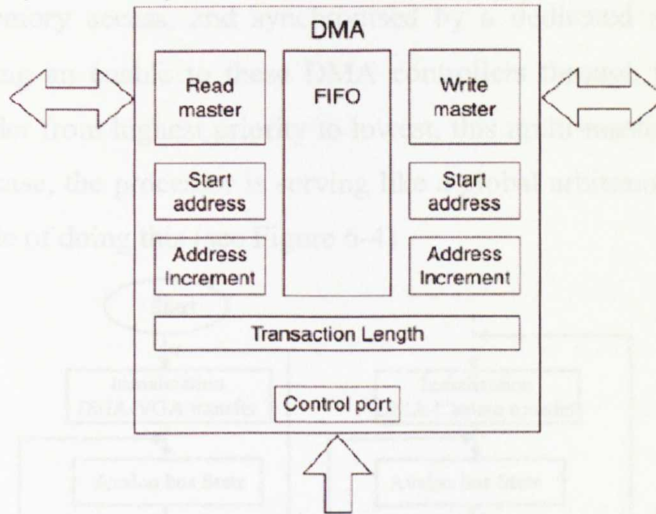


Figure 6-2 Nios DMA peripheral with master & slave ports (From [88])

Figure 6-3 shows another example of typical DMA controllers with an AHB interface.

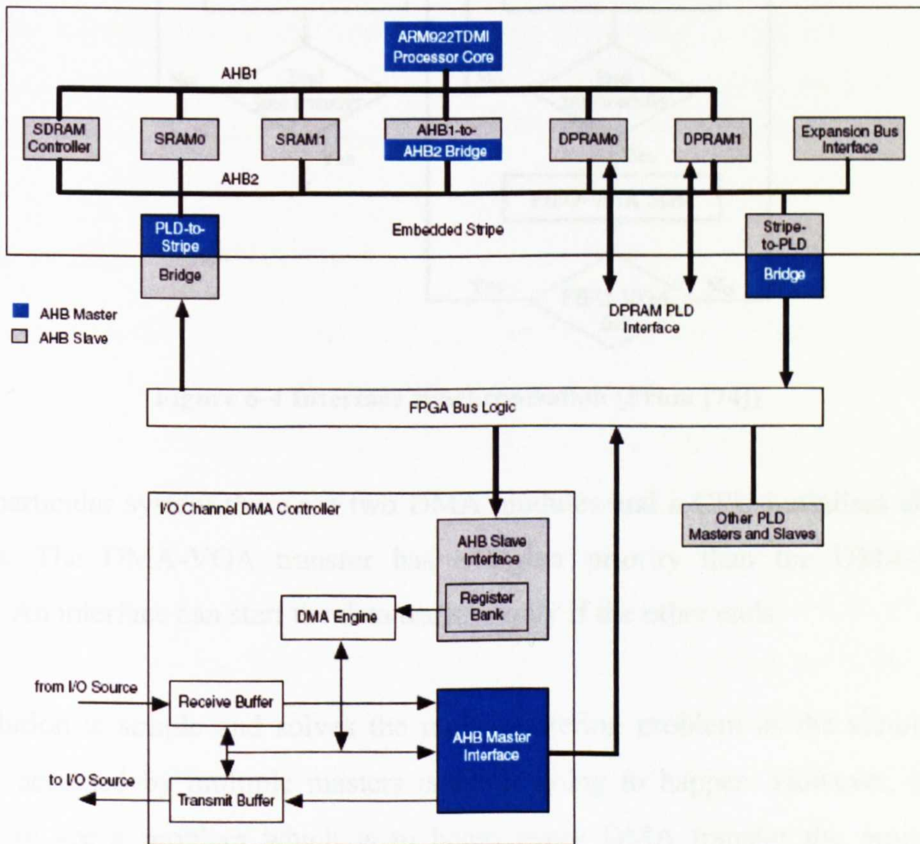


Figure 6-3 I/O to memory DMA controller – Excalibur (From [73])

This DMA controller also contains an AHB slave for configuration purposes and an AHB master to receive/transmit DMA data.

Obviously if these typical DMA controllers are applied into each peripheral that requires direct memory access, and synchronised by a dedicated master such as a processor by writing an enable to these DMA controllers through the configuration port, in a given order from highest priority to lowest, this multi-mastering problem can be solved. In this case, the processor is serving like a global arbitrator. Reference [74] gives us an example of doing this (see Figure 6-4).

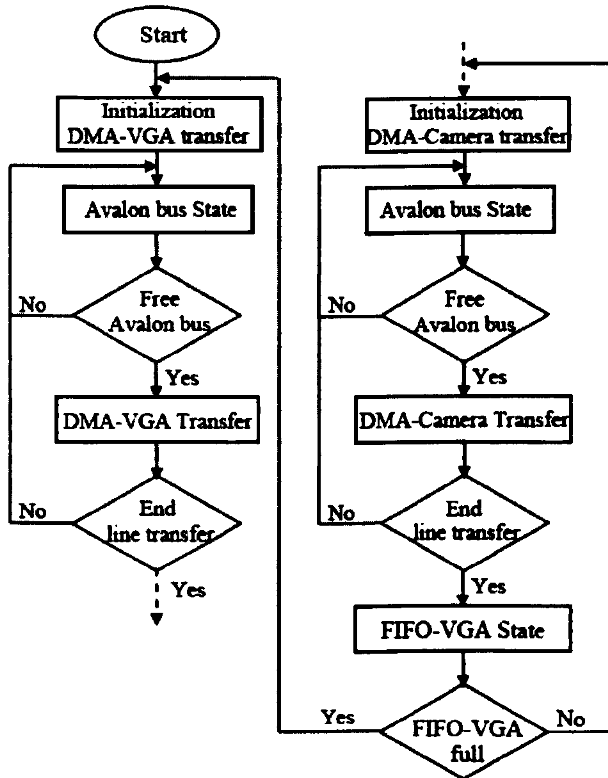


Figure 6-4 Interface synchronisation (From [74])

In this particular system there are two DMA modules and a CPU initialises all DMA transfers. The DMA-VGA transfer has a higher priority than the DMA-Camera transfer. An interface can start the data transfer only if the other ends.

This solution is simple and solves the multi-mastering problem as the simultaneous memory accessed by multiple masters is never going to happen. However, it is not difficult to see a problem which is to begin every DMA transfer the processor is required to write to enable the transfer and before initialising the next DMA transfer

the processor needs some mechanism to check if the current one is completed, this kind of mechanism can be done by polling the status register or handling interrupts generated by the DMA controller [83][88]. Although all this work wouldn't affect the actual DMA transfer by introducing overhead time, it burdens the processor. For example, if it takes 3 system clock cycles to write to enable the DMA transfer, and 10 cycles for interrupt handling, then in total it would take the processor 13 system clock cycles in every DMA transfer. This would become even worse by using the polling approach.

Moreover, in a more general case, to determine the priority for each DMA controller of I/O device, the software needs to understand specific timing issues which tend to be more hardware. For example, if DMA controller 1 is given a higher priority than DMA controller 2, but actually requests from DMA controller 2 happen more frequent than DMA controller 1 while DMA controller 1 takes longer transferring time than DMA controller 2, then this would always result in DMA controller 2 being held up for too long and missing data.

So is there any other way to increase the system performance by reliving the processor burden and also be able to solve the simultaneous multi-mastering problem? **Yes, let the Avalon bus module do the arbitration.**

6.2 Avalon bus arbitration

As mentioned before this multi-mastering can be understood as a bus arbitration problem. In the traditional centralised/parallel arbitration scheme [83] a centralised arbitrator is required between bus masters and slaves to determine which master can access the slave(s) (Figure 6-5). Each bus master can independently request control of the bus from the arbitrator but only one master can gain access to the slave(s) at a time. If multiple masters attempt to access the bus, the arbitrator allocates bus resources to a single master based on a fixed set of arbitration rules. For example, in the priority arbitration scheme the master with the highest priority receives control of the bus first. This centralised arbitration scheme is commonly used in PCs and traditional processor system bus architecture such as the PCI bus, AMBA – AHB and the shared-bus type of the Wishbone bus interconnection [89].

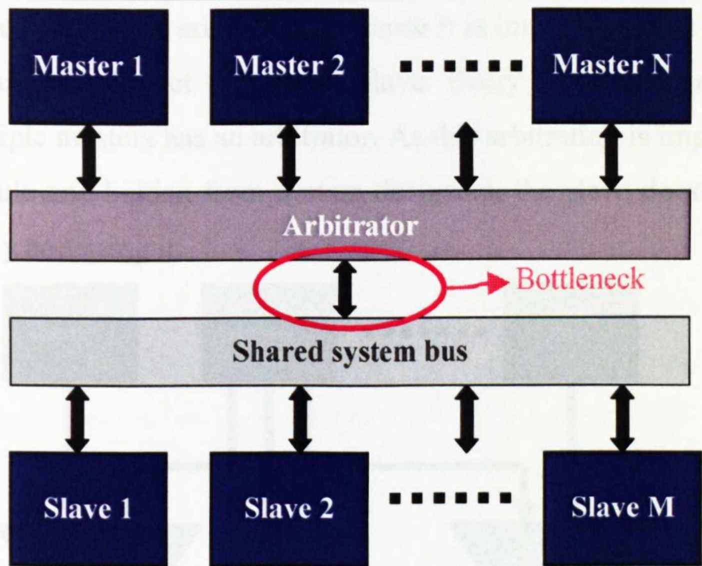


Figure 6-5 Centralised, parallel arbitration bus architecture

Although the arbitrator may become the bandwidth bottleneck as only one master can access the system bus at a time, this centralized bus arbitration scheme works well for a traditional microprocessor system and PC because the masters and slaves are physically separate devices located on a printed circuit board or across backplanes. As the board resources and the number of available I/O pins are limited so that designers must use a common set of bus lines.

However, when a system is able to be integrated into a single chip by using the SOPC/SOC technology, the available resource becomes much more and the interconnection of all on-chip peripherals can be placed and routed in this single chip, it becomes possible to eliminate this bandwidth bottleneck by using new bus architectures with more bus usage to increase the overall bandwidth. The simultaneous multi-master Avalon bus architecture [90] is one of them.

Unlike the traditional bus architecture, the simultaneous multi-master Avalon bus doesn't have a shared bus and is a point-to-point implementation [91], [92] classifies it as circuit-switched star topology. Each master and slave pair has a dedicated connection between them. Because master and slave peripherals are connected with dedicated paths, multiple masters can be active at the same time and can simultaneously transfer data to their slaves as long as they are not sharing the same slave. If more than one master requires access to a single slave, then an arbitrator will be placed on the path between the masters and the slave (see Figure 6-6). This

arbitration is called slave-side arbitration, because it is implemented at the point where two (or more) masters connect to a single slave. Every slave peripheral that can be accessed by multiple masters has an arbitrator. As this arbitration is implemented in the Avalon bus module and hidden from system designers, the slave doesn't know which master is currently accessing it.

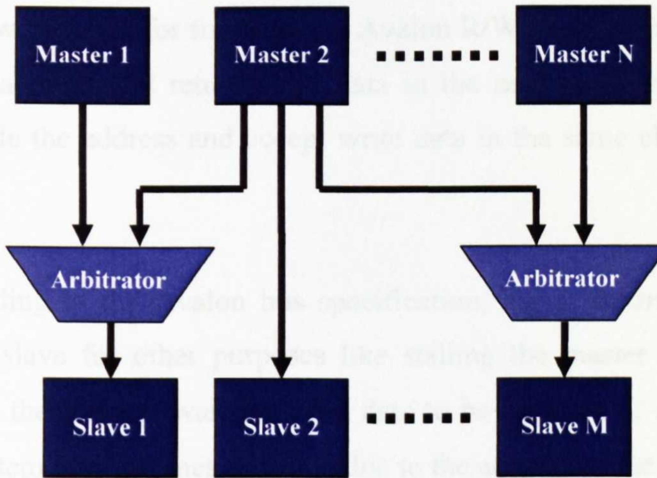


Figure 6-6 Simultaneous multi-mastering Avalon bus arbitration

The arbitration scheme of the Avalon bus is fairness-based arbitration scheme, sometimes referred to as a round-robin or weighted round-robin scheme. For any given connection between a master and slave, designers can select how much access each master has to a given slave. By setting the correct number the master has the highest fairness setting is more likely to gain access to the slave. The fairness setting is configurable though the SOPC builder. Figure 6-7 shows the basic operation of how two masters access a slave at the same time.

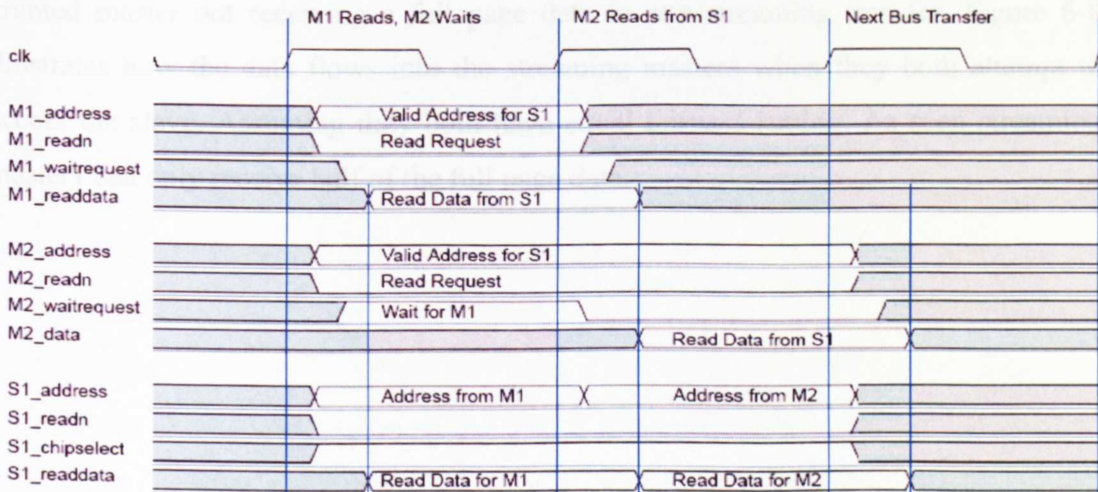


Figure 6-7 Successive fundamental read transfers to a common slave (From [90])

Master M1 and M2 tried to access the slave S1 at the same time, and M1 won the arbitration and M2 was forced to wait for one clock cycle. As seen the signal 'waitrequest' is the wait control signal which keeps the other masters, not granted access to the slave, in the wait state.

This arbitration works well for fundamental Avalon R/W transfers, as long as the slave can decode the address and return valid data in the next clock cycle during a read transfer or decode the address and accept write data in the same clock cycle during a write transfer.

However, according to the Avalon bus specification, signal *waitrequest* can also be asserted by the slave for other purposes like stalling the master which has already gained access to the slave to wait for valid data to be returned or sending data to the slave. In this system this becomes essential due to the activity of the SDRAM device. If this is used in a multi-mastering system then obviously this would 'confuse' all masters because in the multi-mastering arbitration scheme, signal *waitrequest* is generated by the arbitrator logic and active only for the masters which are not granted access to the slave instead of the granted one. This problem exists in this image processing system as the specific SDRAM device does need a few clock cycles to activate the bank, row and columns plus the CAS latency and the insertion of the auto-refresh operation, the memory controller need to assert *waitrequest* to tell the master port to wait for start of the actual data transfer. Furthermore, as the SDRAM is operating at full page mode, multiple streaming transfers requested on this SDRAM slave could result in the first granted master not receiving a full page data in one streaming transfer. Figure 6-8 illustrates how the data flows into the streaming masters when they both attempt to access the slave. Assuming they both have equal fairness setting. As seen streaming master1 can only receive half of the full page data.

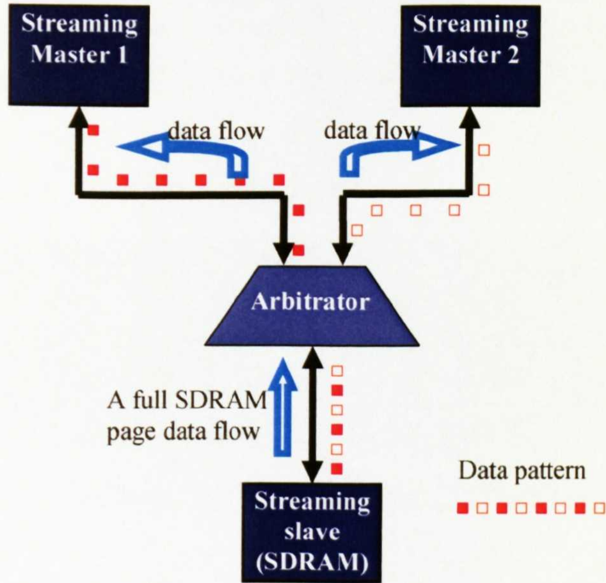


Figure 6-8 An arbitration view during conflict between two streaming masters

Therefore a solution must be developed to allow the streaming transfer with peripheral-controlled waitrequest to work properly in this multi-mastering system, and ensure a single streaming transfer can always transfer a full memory page losslessly. The next section will discuss a solution to this simultaneous multi-mastering Avalon streaming transfer with peripheral-controlled waitrequest (**SMMAST-PCW**) problem.

6.3 Implementation of simultaneous multi-mastering streaming Avalon transfer with peripheral-controlled waitrequest

Experimental work has been undertaken to investigate the possibility of implementing the SMMAST-PCW between multiple streaming Avalon master ports and a single streaming Avalon slave port. However, it can be concluded that operations would fail if implementing this kind of transfer in such a multi-mastering system due to the design of the Avalon bus architecture. The main bottleneck is that the peripheral-controlled waitrequest interferes with the Avalon arbitrator generated waitrequest.

A feasible solution would become available if the Avalon arbitrator never generates a waitrequest to the streaming master though it is still there, and uses something else to respond to the waitrequest generated by the arbitrator. In the actual implementation, a simple Avalon master port (non-streaming) is placed in every master peripheral, and a

simple Avalon slave port (non-streaming, no peripheral-controlled waitrequest) is placed in the common slave peripheral. The one in the master peripheral is called the "streaming control master port" and the one in the slave peripheral is called the "streaming control slave port" (see Figure 6-9).

Figure 6-9 Master peripheral and slave peripheral with streaming control feature

The streaming data master ports work exactly the same as the normal streaming Avalon master such as the master ports in the video capture controller and video display controller. However, the streaming master wouldn't initialise any streaming transfer until the streaming control master wins the arbitration and the returned data from the streaming control slave indicates that the streaming slave is not busy. So conflicts between the streaming data masters and slaves will never happen and the master will never be confused by the waitrequest which could be generated from the Avalon arbitrator or peripherals. Second, when the granted streaming data master sends request to the streaming data slave, the slave immediately sets the busy bit and the streaming control slave sends out this bit as master read data so the steaming control master knows the status of the slave and can continue holding the other pending masters until the current streaming transfer finishes. So a non-abortion streaming transfer can be guaranteed. By using this solution, sufficient bandwidth and lossless streaming transfer can be achieved. Figure 6-10 is the interconnection diagram of all video masters, the memory slave and Avalon arbitrators in this SOPC image processing system with the streaming control master/slave applied.

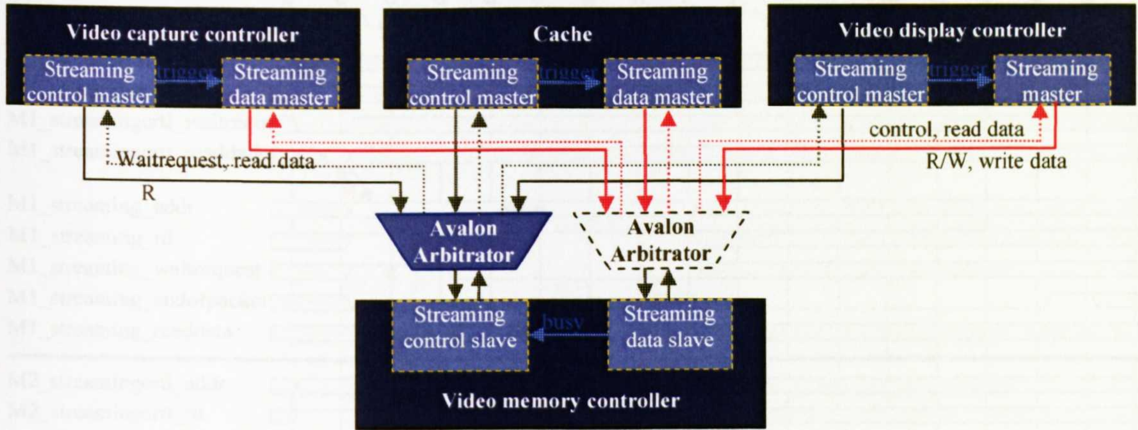


Figure 6-10 Block diagram of simultaneous multi-mastering image processing system

Figure 6-11 illustrates how a streaming control master should respond to the Avalon arbitration and the streaming control slave.

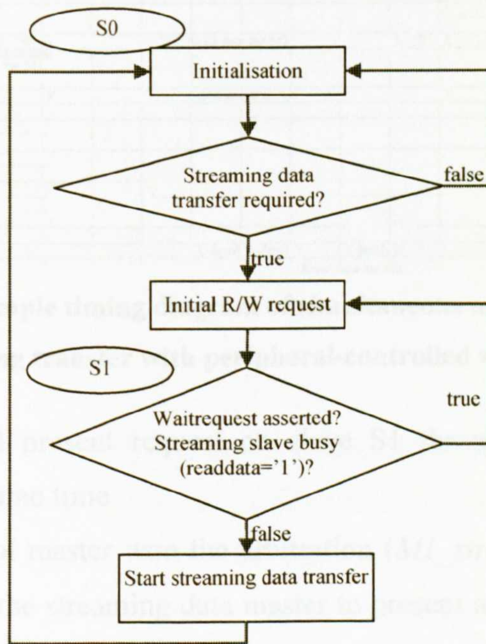


Figure 6-11 ASM chart of streaming control master

Figure 6-12 shows an example timing diagram of simultaneous multi-mastering Avalon streaming transfer with peripheral-controlled waitrequest.

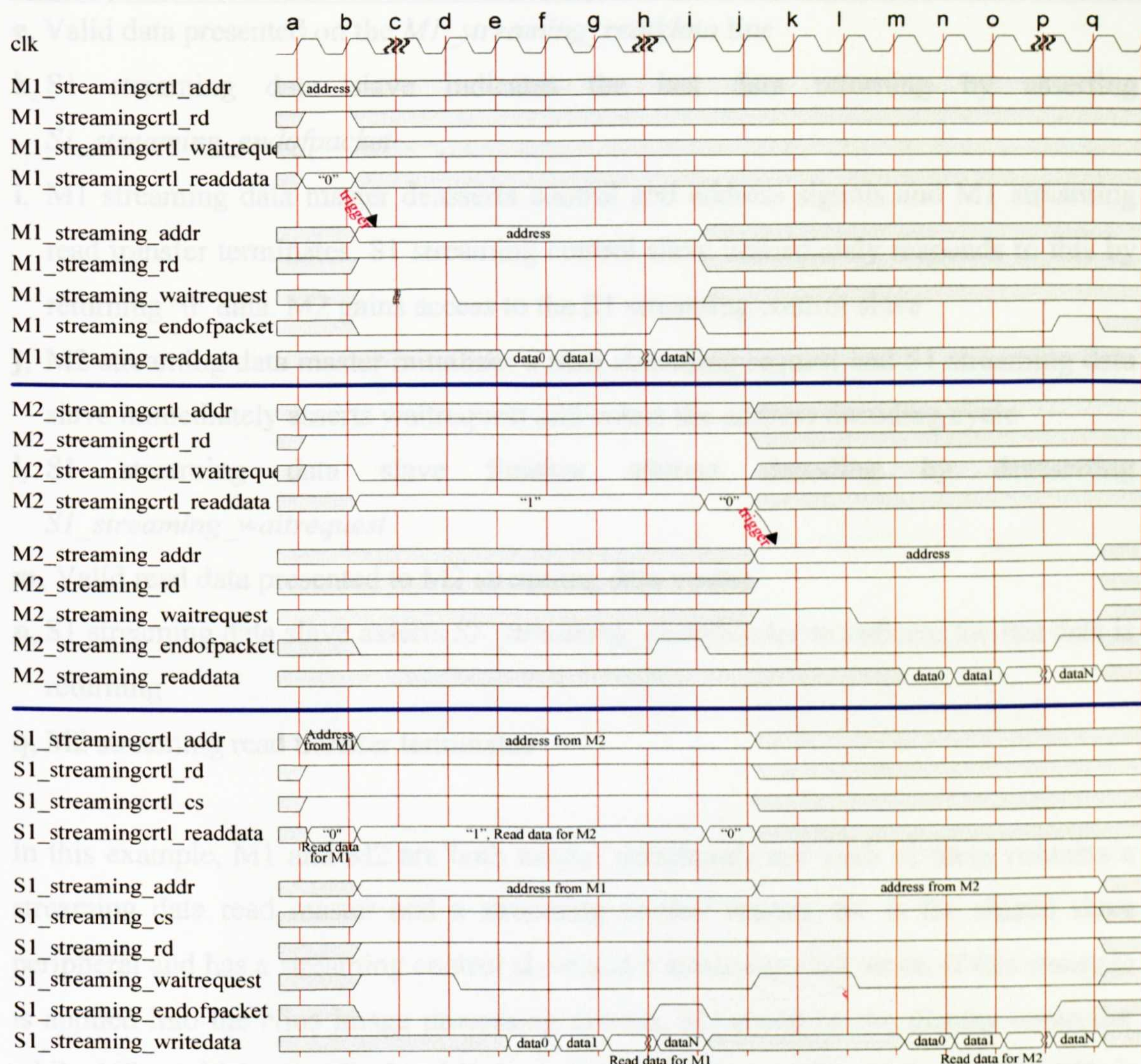


Figure 6-12 An example timing diagram of simultaneous multi-mastering Avalon streaming transfer with peripheral-controlled waitrequest

a, Master M1 and M2 present request on slave S1 through the streaming control master/slave at the same time

b, M1 streaming control master won the arbitration (*M1_streamingcrtl_waitrequest* is low), and initialises the streaming data master to present a streaming read request to the streaming data slave in S1. S1 streaming data slave responds to this request immediately and the streaming control slave returns '1' on *S1_streamingctrl_readdata* to indicate the slave is busy

c, M2 is being held and M1 streaming data master is waiting for S1 to decode the address. *S1_streaming_waitrequest* is asserted by S1 streaming slave and passed to *M1_streaming_waitreuest*

d, S1 streaming data slave indicates the completion of address decoding by deasserting the *S1_streaming_waitrequest*

e, Valid data presented on the *M1_streaming_readdata* line

h, S1 streaming data slave indicates the last data returning by asserting *S1_streaming_endofpacket*

i, M1 streaming data master deasserts control and address signals and M1 streaming read transfer terminates. S1 streaming control slave immediately responds to this by returning '0' data. M2 gains access to the S1 streaming control slave

j, M2 streaming data master initialises a read streaming request and S1 streaming data slave immediately asserts waitrequest and enters the address decoding cycle

l, S1 streaming data slave finishes address decoding by deasserting *S1_streaming_waitrequest*

m, Valid read data presented to M2 streaming data master

p, S1 streaming data slave asserts *S1_streaming_endofpacket* to indicate the last data is returning

q, M2 streaming read transfer terminates

In this example, M1 and M2 are both master peripherals and each of them contains a streaming data read master and a streaming control master. S1 is the shared slave peripheral and has a streaming control slave and a streaming data slave. If this example is applied into the Nios image processing system, M1 could be the display controller while M2 could be the Cache. M1 is getting data from video bank 1 while M2 is reading data from video bank 2.

As seen from the timing diagram, although this solution would introduce one extra data clock cycle overhead time to every streaming transfer, the processor isn't involved in doing any arbitration, and it doesn't need to check the status of every streaming transfer. All arbitration is done by the hardware. The processor will be told a bank has been finished displaying or capturing a frame by interrupts. By using this simultaneous multi-mastering streaming Avalon transfer with peripheral-controlled waitrequest data transferring efficiency can be further increased.

Furthermore, this solution maintains the slave module generic. Whenever a new master requires streaming access to the shared slave, what it needs to do is to implement the streaming master ports by following the SMMAST-PCW solution described above. Additional changes on the slave are not required.

6.4 Another solution

Section 6.2 has described the main bottleneck of implementing the SMMAST-PCW is that the peripheral-controlled waitrequest interferes with the Avalon arbitrator generated waitrequest. So if the Avalon arbitrator doesn't exist, can this problem be solved? Yes, it can.

The Avalon multi-mastering bus architecture allows point-to-point interconnection. Therefore by creating more steaming slave ports in the memory controller and each one responds to a streaming master port as shown in Figure 6-13, the multi-mastering problem can be solved. In this solution, the memory controller DOES appear as a triple-ported slave.

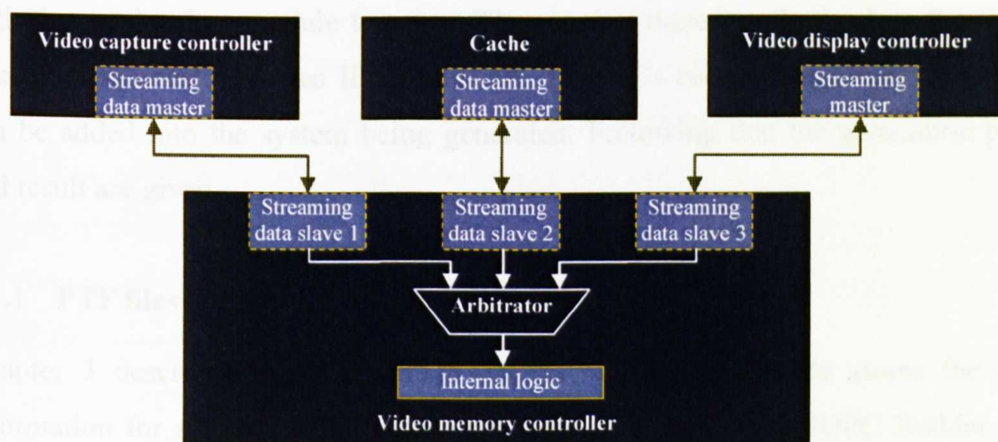


Figure 6-13 Multi-ported memory slave solution

This solution solves the problem because it removes the Avalon arbitrator, and the memory slave can generate separate waitrequest to hold up its corresponding master without worrying about interfering with the others. Obviously the main difficulty of this solution is to design the custom arbitrator.

The advantage of this solution is the arbitrator is custom defined, so non-overhead time can be achieved to maximise the data transfer rate. Also, pipelining (see section 5.2.3) could be possibly implemented. However, whenever there is a new master needs to connect to the slave, the memory controller is needed to be modified or re-generated, which makes the slave module none general. Furthermore, this custom-defined arbitrator could be as complicated as the Avalon arbitrator, for example if the same arbitration scheme is used, then this would be time consuming to implement. Therefore, this solution is not fulfilled in the current SIPS design.

Chapter 7. System Core Generation, Synthesis & Implementation

The SIPS system core was written in a high-level hardware description language - VHDL. Following describing all main IP modules of the system core, this chapter mainly discusses how this VHDL core was generated and synthesised. Section 7.1 presents how to generate the system core by using SOPC builder and PTF files [93]. Section 7.2 presents a detailed discussion of the synthesis results.

7.1 System core generation

As described in Chapter 3, SOPC Builder automatically generates a synthesisable Nios processor system by integrating all standard/custom IPs out of the component library, with the Avalon bus module together. This section therefore firstly describes how to document the custom video IPs into SOPC Builder's component library so that they can be added into the system being generated. Following that the generation process and result are given.

7.1.1 PTF files

Chapter 3 describes that the system description file (PTF) file stores the design information for systems being edited and generated within the SOPC Builder. Apart from the System PTF file, there is another type of PTF files which is the Class.ptf files. It describes the SOPC Builder library components such as I/O signals, how they match the Avalon bus interface, and simulation settings. Each library component displayed on the left panel in Figure 3-15 has a unique class.ptf file. When SOPC Builder starts up it searches for components by "looking" in all directories on a configurable search path for files named class.ptf. When it discovers a file named class.ptf, it will read the file to see if it contains a valid and correct PTF file description of a library component. There is a one-to-one correspondence between components displayed in the GUI's library-list and discovered class.ptf files. Practically a system PTF file is the collection of class.ptf files of all IP components that the system contains (see Figure 3-16) with extra description of the global connectivity of all sub-modules.

A typical class.ptf PTF file contains 7 fields. "CLASS SECTION NAME" field contains the formal name of the library component. "ASSOCIATED FILES" field

describes the add_program, edit_program and generator_program files. These files are used to direct the SOPC Builder on how to add the information of the class.ptf file into the system PTF file, and how a library component is edited and generated into the system top-level file. If the default generator is chosen as the generator-program, then there will be an extra field called "DEFAULT GENERATOR" which contains information on the files to run, top-module name etc. The "MODULE DEFAULTS" field describes all information of the IP component such as all Avalon masters/slaves it contains, the Avalon module type, properties and port wiring. The "USER INTERFACE" field contains the information that describes terms for use by the SOPC Builder GUI (e.g., tips, module pool organisation, etc.). More information of the PTF syntax can be obtained in [93].

In the Nios embedded image processing system, it contains not only the library components that Altera provides such as the flash controller and the SRAM controller, but four user-defined modules. In order to integrate them into the whole system module a separate class.ptf PTF file must be produced for each of them. Figure 7-1 illustrates the custom-defined library IP components displayed on the system content page of the SOPC Builder.

An example of the class.ptf PTF file written for the Cache is included in Appendix F.

7.1.2 System generation

When all IP components including the custom-defined ones are ready to be used, the top-level system module can be built up by adding the required IPs and connecting all of them in the SOPC Builder. Figure 7-2 illustrates the top-level view of the image processing system core which includes information of all IP components and interconnections, fairness settings for each master, basic address assignments for all Avalon slaves, unique interrupt number for each master which has an interrupt output, and the clock frequency.

As mentioned in Chapter 6 the fairness setting of the arbitration scheme is configurable. User can assign a number from 1 to 99 for each master/slave pair. In this system, all video master peripheral has the same share on the memory slave which is 1

(Figure 7-2). That means every master will be granted access to the slave on average 33.33% of the time. Zero means no connection between the specified master and slave path.

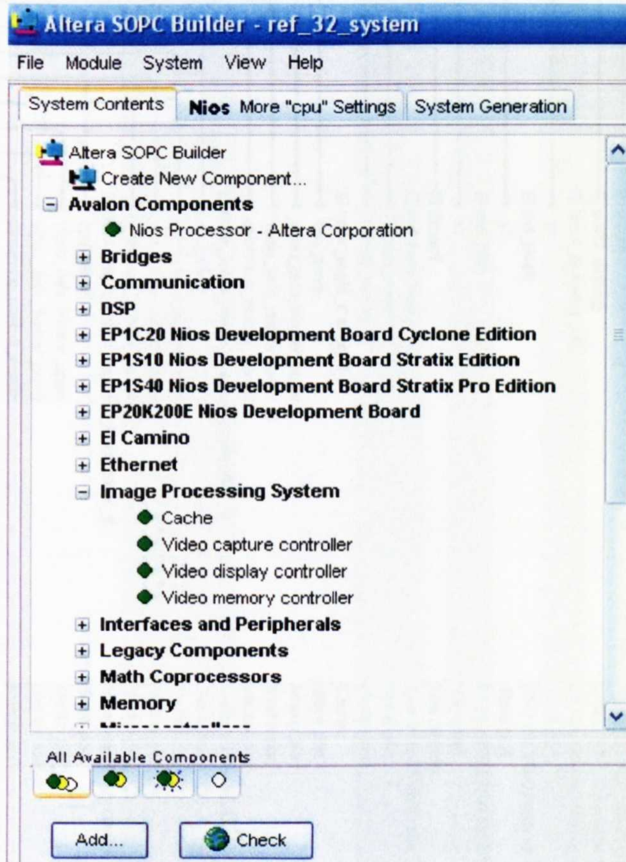


Figure 7-1 Custom defined library components

Once the system module has finished assembly, the system generation procedure starts when the user clicks **Generate** as the final SOPC Builder GUI action. During the Top-Module Generation phase, SOPC Builder writes the definition of the system's top-module into the system HDL file. The top module definition includes proper declaration of all the system's I/O ports, instances of every module in the system, instances of all the arbitration modules that contain the bus logic, and interconnections between all the modules. It also generates the software-support (SDK) directories and some simulation files. Figure 7-3 shows part of the generation results in SOPC Builder.

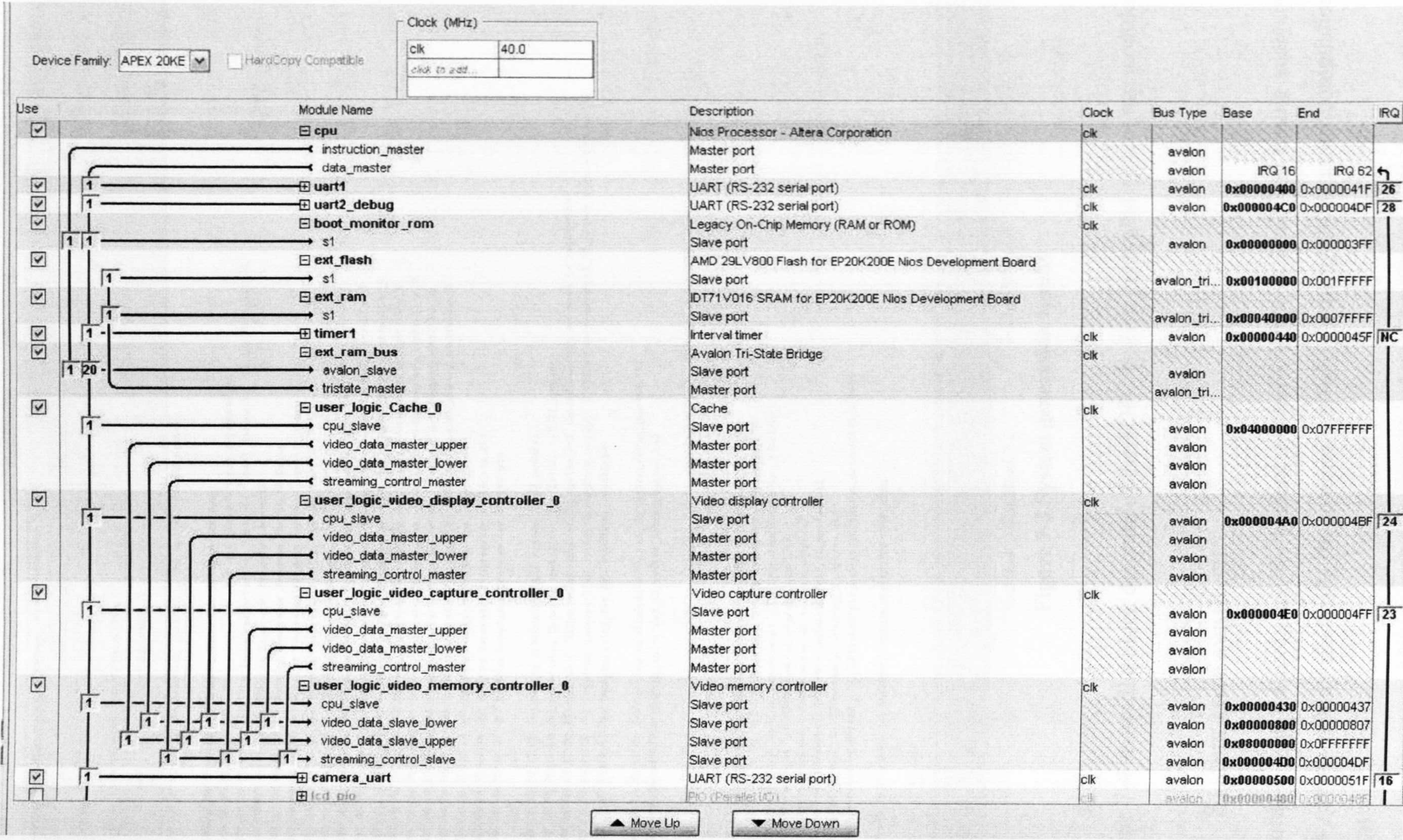


Figure 7-2 Top-level system view in SOPC Builder

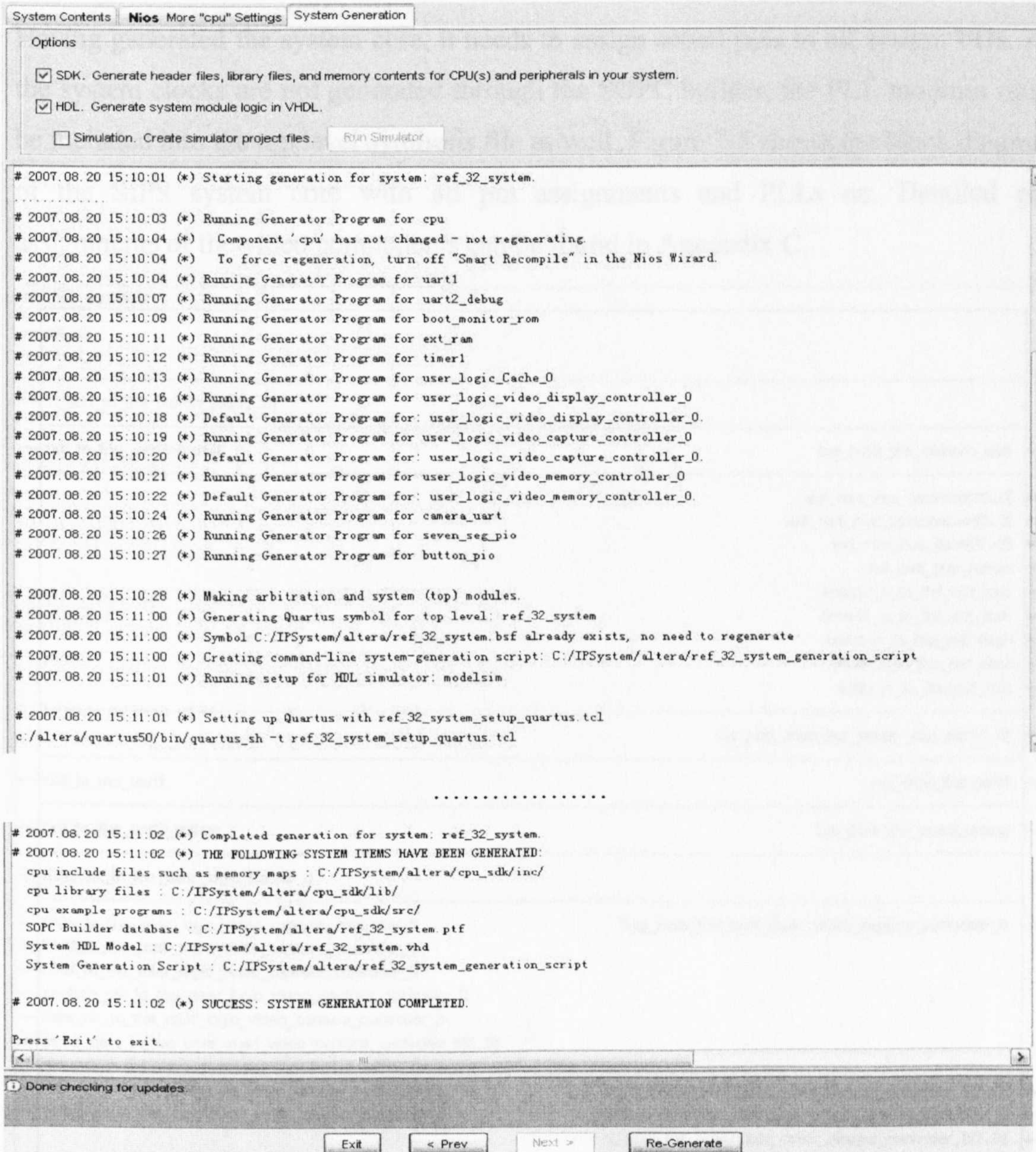


Figure 7-3 System generation results

Figure 7-4 shows the block diagram of the SOPC builder generated system top module.

7.2 System core synthesis

Quartus II was the tool used for system synthesis, place & route and timing analysis. This section mainly discusses the synthesis results given by the Quartus compilation report.

7.2.1 System core synthesis

Having generated the system core, it needs to assign actual pins to all system I/Os. As the system clocks are not generated through the SOPC builder, the PLL modules must be included into the top level synthesis file as well. Figure 7-5 shows the block diagram of the SIPS system core with all pin assignments and PLLs on. Detailed pin assignments of the video components can be found in Appendix C.

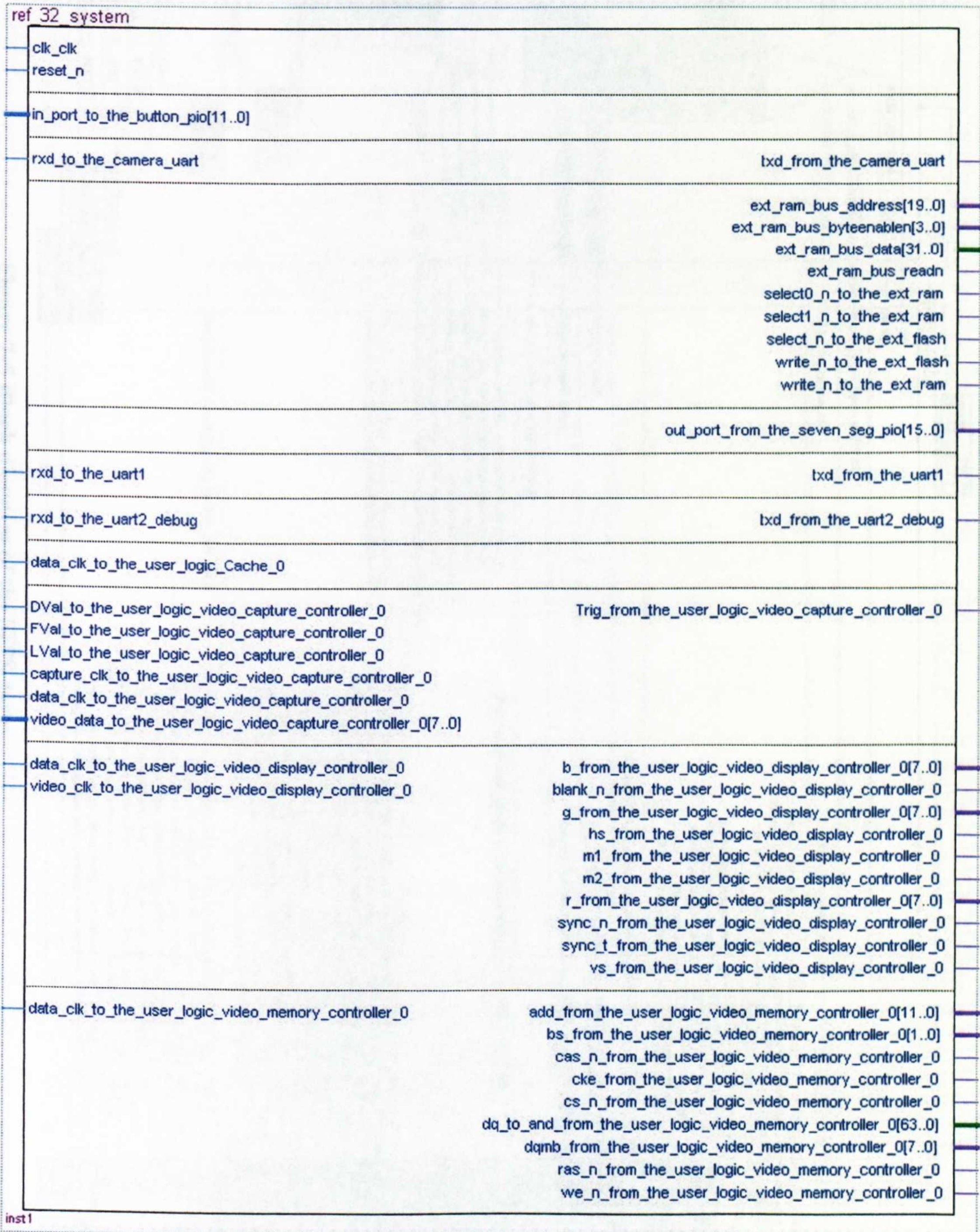


Figure 7-4 Top system module view

Figure 7-5 Block diagram of the SIPS core

7.2.2 Synthesis results & discussions

The synthesis result of the Nios (version 3.2) integrated image processing system core in 8-bit mode on the Apex device EP20K200EFC484-2X is shown in Table 7-1.

Table 7-1 Synthesis result in 8-bit mode

LEs	5776 / 8320 (69%)
Memory bits/ESBs	105472 / 106496 (99%)
Embedded System Blocks (ESBs)	51 /52 (98%)
Total pins	238 / 376 (63%)
PLLs	2/2 (100%)
Data clock f_{max}	88.28MHz
System clock f_{max}	41.35MHz

Note: The synthesis results might be different when using different version of Quartus II software and Nios CPU.

This synthesis result was given by the Quartus II 5.0 compilation report. This result shows the image processing system core has used most of the resource of this FPGA device, especially the memory blocks, and this is good. However, the required clock frequencies could not be satisfied which are 100MHz data clock and 50MHz system clock. As seen from the table the system speed is degraded. A few methods have been tried out to optimise the speed such as using LogicLock [54] design methodology and configuring the Quartus synthesis tool to compile the system with optimised speed. But it still couldn't meet the design requirements. One of the reasons is the existence of multiple masters sharing the same slave that requires the Avalon arbitrator to decode more address and other control information, and all these paths are combinational logic. Also, more CPU peripherals require more resource to decode the CPU address and control signals and hence result in more cell delays. Furthermore, the use of waitrequest in all video components could significantly increase the delay. Finally as the design gets larger, the area it covers on the device also gets broader, and this would result in the interconnection delay in certain paths gets longer and eventually affect the system performance. Certainly the performance of the chosen device should also be considered as one of the main constraints.

Reference [94] and [95] give some examples of speed performance on similar devices

by implementing the Nios embedded system with some other peripherals on. [94] indicates the maximum frequency of the reference design with 32-bit Nios processor 1.1 on device EP20K200EFC484-1X is 43.26MHz (-1X has a higher speed performance than -2X). [95] gives a comparison between two different systems (they both have the same Nios processor with version 3.0) on device Apex20KE1000-2X. The fmax is 58MHz when 3804 logic elements (LEs) are used however the speed is degraded to 40MHz when the total LEs are 4663. From these two examples it can see that the speed performance of SIPS on the selected device is reasonable because it has a larger number of gates than the second example and used a device with lower speed performance than the first example. By implementing SIPS on different device various performance results can be achieved.

Table 7-2 lists some synthesis results of SIPS on several devices to give a view of how system performance varies on different FPGAs. This performance estimation was only performed after synthesis and based on the same SIPS design without any optimisation given for the specific device, the actual place & route wasn't taken into account as pin assignments were unknown. All synthesis was performed by Quartus II 5.0 with Nios 3.2 utilised.

In order to run SIPS in 24-bit RGB mode a minimum of 22.8kB embedded memory is required, however the Apex 20K series device only offers 15kB memory. Therefore SIPS in 24-bit mode wasn't evaluated.

Since the synthesis result couldn't meet the requirements, it had to redefine the data and system clock frequency. In order to maintain the integer multiple relation between the system clock and the data clock, and not exceed the allowed fmax, **40MHz** for the system clock and **80MHz** for the data clock were chosen. These frequencies will be used to calculate the image processing performance in Chapter 8.

Due to the degrading of the data clock speed, the auto-refresh counter has to be changed as well. To perform 4096 times of auto-refresh within 64ms the SDRAM controller must generate an auto refresh command no more than every $15.625\mu s / 0.0125\mu. = 1250$ data clock cycles.

Table 7-2 Estimated performance on various FPGAs

<table>
<tr><th>Device family</th><th>Device</th><th>Video mode</th><th>System clock f_{max}</th><th>Data clock f_{max}</th><th>LEs</th><th>Memory bits</th><th>PLL</th></tr>
<tr><td rowspan="2">Apex 20KE</td><td rowspan="2">EP20K200EFC484-1X</td><td>8</td><td>51.4MHz</td><td>99.67MHz</td><td>5776 (69%)</td><td>105472 (99%)</td><td>2 (100%)</td></tr>
<tr><td>24</td><td colspan="5">N/A</td></tr>
<tr><td rowspan="2">Apex II</td><td rowspan="2">EP2A15F672C7</td><td>8</td><td>42MHz</td><td>83.02MHz</td><td>5808 (69%)</td><td>105472 (99%)</td><td>2 (100%)</td></tr>
<tr><td>24</td><td colspan="5">N/A</td></tr>
<tr><td rowspan="2">Stratix</td><td rowspan="2">EP1S10 F780 C5</td><td>8</td><td>75.29MHz</td><td>165.70MHz</td><td>5294 (50%)</td><td>105472 (11%)</td><td>2 (33 %)</td></tr>
<tr><td>24</td><td>79.80MHz</td><td>167.20MHz</td><td>5239 (49%)</td><td>187392 (20%)</td><td>2 (33 %)</td></tr>
<tr><td rowspan="2">Stratix II</td><td rowspan="2">EP2S15F484C5</td><td>8</td><td>97.91MHz</td><td>165.98MHz</td><td>4503 (36%)</td><td>105472 (25%)</td><td>2 (33 %)</td></tr>
<tr><td>24</td><td>102.13MHz</td><td>163.61MHz</td><td>4473(35%)</td><td>187392 (44%)</td><td>2 (33 %)</td></tr>
<tr><td rowspan="2">Cyclone</td><td rowspan="2">EP1C20F400C7</td><td>8</td><td>70.27MHz</td><td>121.88MHz</td><td>4908 (24%)</td><td>105472 (35%)</td><td>2 (100%)</td></tr>
<tr><td>24</td><td>73.14MHz</td><td>117.04MHz</td><td>4871 (24%)</td><td>187392 (63%)</td><td>2 (100%)</td></tr>
<tr><td rowspan="2">Cyclone II</td><td rowspan="2">EP2C35F484C7</td><td>8</td><td>78.18MHz</td><td>127.81MHz</td><td>5018 (15%)</td><td>105472 (21%)</td><td>2 (50%)</td></tr>
<tr><td>24</td><td>73.01MHz</td><td>122.01MHz</td><td>4966 (14%)</td><td>187392 (38%)</td><td>2 (50%)</td></tr>
</table>

Chapter 8. System Tests, Image Processing Algorithm Implementation & Performance Analysis

In order to ensure the system is working properly, simulations and hardware tests for individual IP component associated with the hardware interface and the whole system are essential. This chapter therefore firstly focuses on discussing the test scheme that has been undertaken for the main IP components and the whole system. It then describes the software development which includes the general issues and the implementation of various image processing algorithms. A performance analysis is given for each of these image processing tests. Finally a summary is given for this chapter.

8.1 System tests

Various tests have been undertaken for each video IP component with its dedicated hardware interface. The units under test include the video memory, Cache, video display, video capture and the whole system. In this section, it describes how all these tests were undertaken in both simulation and hardware verification and the test results will be discussed. Furthermore, a special test given for the SMMAST-PCW will also be presented.

8.1.1 Simulations

The simulation scheme and results presented in this section are for the SOPC top module, which includes all video IP components and the Nios processor, Avalon bus module and other Altera provided IPs such as the SRAM, Flash, and Timer. The SOPC Builder generates all simulation instances for simulations. The actual simulations were done by using the simulation tool - ModelSim. Details of setting up simulation for Nios processor design can be obtained in [96].

Figure 8-1 shows the block diagram of the simulation scheme of this SOPC top module. As seen in the figure, the whole SIPS core is the unit under test. There are three extra blocks for driving the whole simulation. They are the external SRAM test interface, video memory interface test controller and the video capture interface test controller.

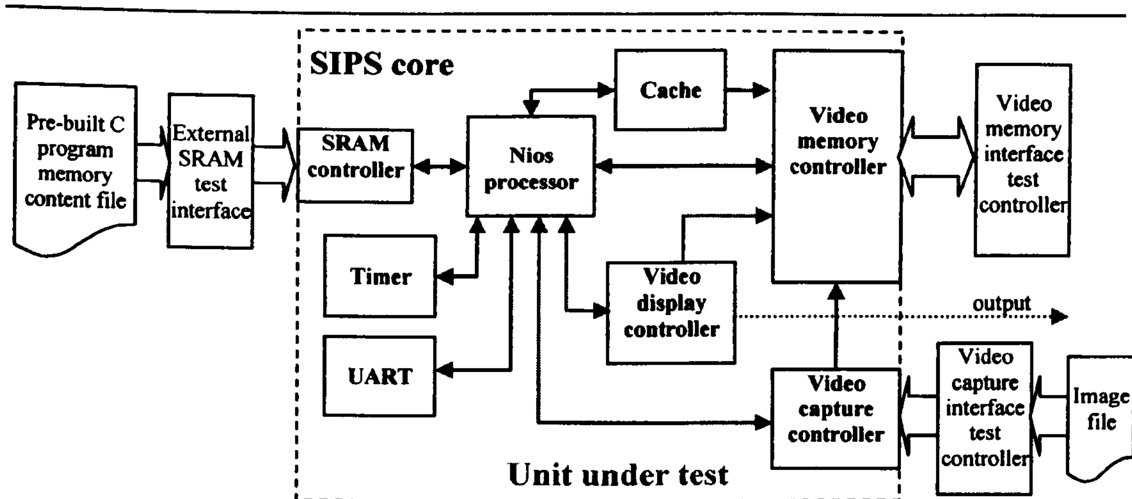


Figure 8-1 SIPS simulation scheme

In SOPC Builder, the user can specify which CPU program is used for simulation. This program could be a C program which the SOPC Builder can build and store as the instruction set in the dedicated memory device where the CPU fetches the instruction. In SIPS, SRAMs are used to store the CPU data and program. Therefore, in simulation, the content of the pre-built C program memory file has to be read into the SRAM controller through the auto-generated SRAM test interface to drive the processor. In order to simulate the function of the SDRAM device, a video memory interface test controller was created in the test bench to respond to the SDRAM command operations. There is a memory module inside it to store the data being written. So this interface block works 'virtually' as the SDRAM device. The video capture interface test controller was used to generate the video timing control signals and video data read from an image data file to drive the video capturing in SIPS. The next few sections will discuss some typical simulation results of each main IP component. (Note: only 8-bit video mode simulation is presented in this section).

8.1.1.1. Video memory controller simulation results & discussions

Figure 8-2 shows the simulation result of an Avalon streaming write to the memory controller with bust length (BL) of one. In this simulation, data 0x0000000B is being written into the upper 32 bit word at SDRAM address 0x005 (0x16 right shift 2 bits). From this figure, it can be seen that an Avalon streaming write with length 1 takes **5** data clock cycles to complete which includes 1 cycle for issuing the streaming control slave (time *a* to *b*) and 4 cycles to do the entire write cycle (*b* to *d*). The actual SDRAM write cycle completes with 3 cycles delay (*d* to *f+1*). There are small spikes on the

signal *data_slave_waitrequest_u* and *data_slave_waitrequest_l*, however, that wouldn't cause problems to the actual system as this system is a synchronous system and these spikes are very short pulses. In fact these spikes only exist in the behaviour simulation.

Figure 8-3 shows the simulation result of an Avalon streaming write to the memory controller with bust length of 80. It can be seen, from this figure, that an Avalon streaming write with length 80 takes **84** data clock cycles to complete (time *a* to *f*) while the actual SDRAM burst write completes with 3 cycles latency (*f* to *g+1*). Time *d* to *g* is the write sequence of 80-word data being written to memory.

Figure 8-4 shows the simulation result of an Avalon streaming read from the memory controller, while the burst length is 1. In this simulation, 64-bit data 0x0000000500000004 at SDRAM address 0x02 (0x08 right shift 2 bits) is required to be read. From this figure, it can be seen that an Avalon streaming read with length 1 takes **10** data clock cycles to complete which includes 1 cycle for the streaming control slave to perform arbitration (time *a* to *b*), 6 cycles for issuing the Bank Activate and Read command (time *b* to *d*), 2 cycles CAS latency (time *d* to *f*), 1 data cycle (time *f* to *g*) and another extra cycle for valid data returned to the master (time *g* to *h*).

Figure 8-5 shows the simulation result of an Avalon streaming read from the memory controller with burst length of 80. From this figure, it can be seen that an Avalon streaming read with length 1 takes **89** data clock cycles to complete which includes 1 cycle for the streaming control slave to perform arbitration (time *a* to *b*), 6 cycles for issuing the Bank Activate and Read command (time *b* to *d*), 2 cycles CAS latency (time *d* to *e*), 80 data cycles (time *e* to *g*) and another extra cycle for valid data returned to the master (time *g* to *h*).

From all of the simulation results given for the memory controller, it can be concluded that an Avalon streaming write transfer with burst length n ($1 \le n \le$ maximum memory column size) takes ***n+4*** data clock cycles to complete. An Avalon streaming read transfer with burst length n ($1 \le n \le$ maximum memory column size) takes ***n+9*** data clock cycles to complete. *tRCD* (Active to Read/Write Command Delay Time) of 2 data clock cycles and CAS latency of 2 data clock cycles are applied into these calculations.

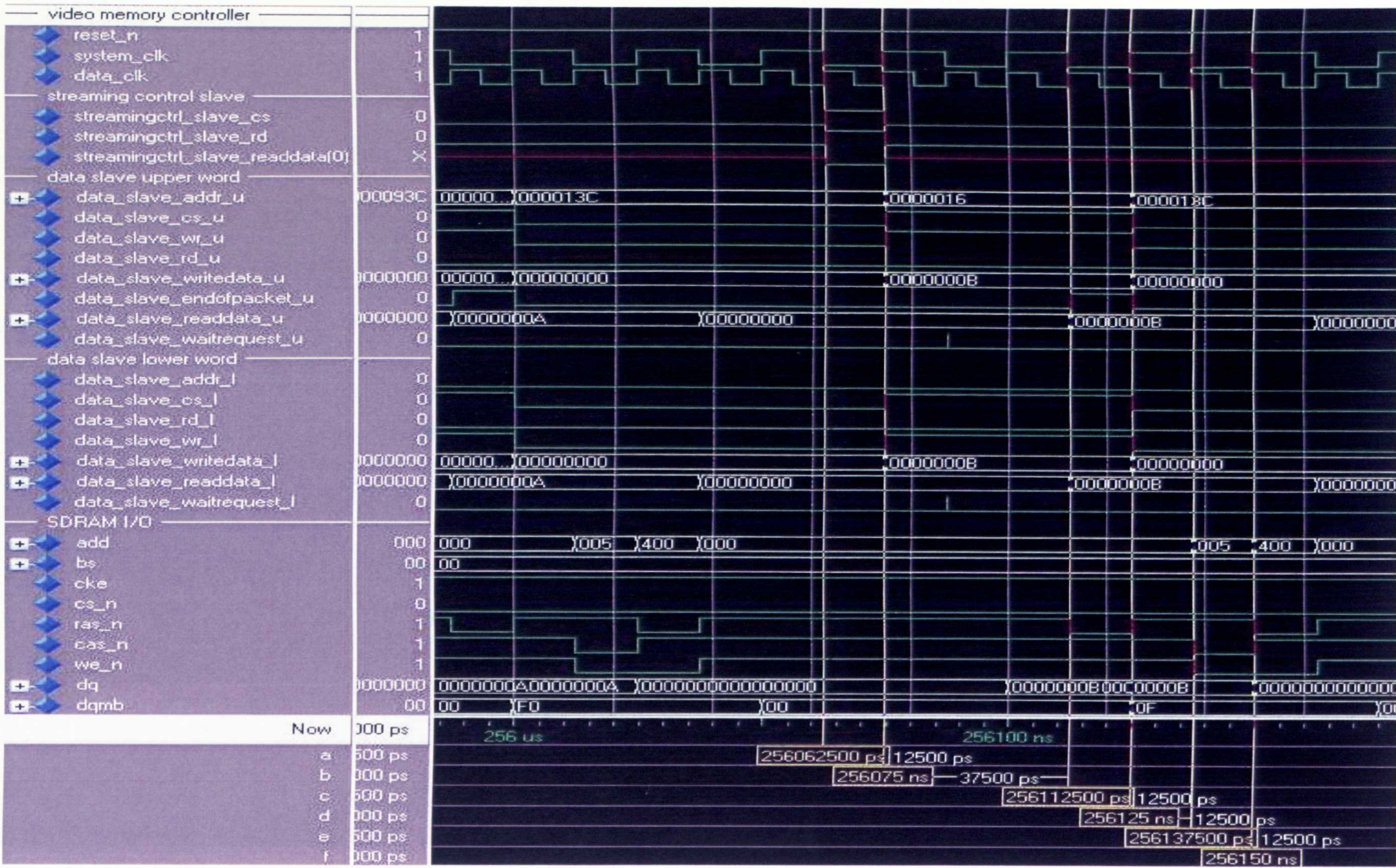


Figure 8-2 Video memory controller simulation result – burst write (BL = 1)

Figure 8-3 Video memory controller simulation result – burst write (BL = 80)

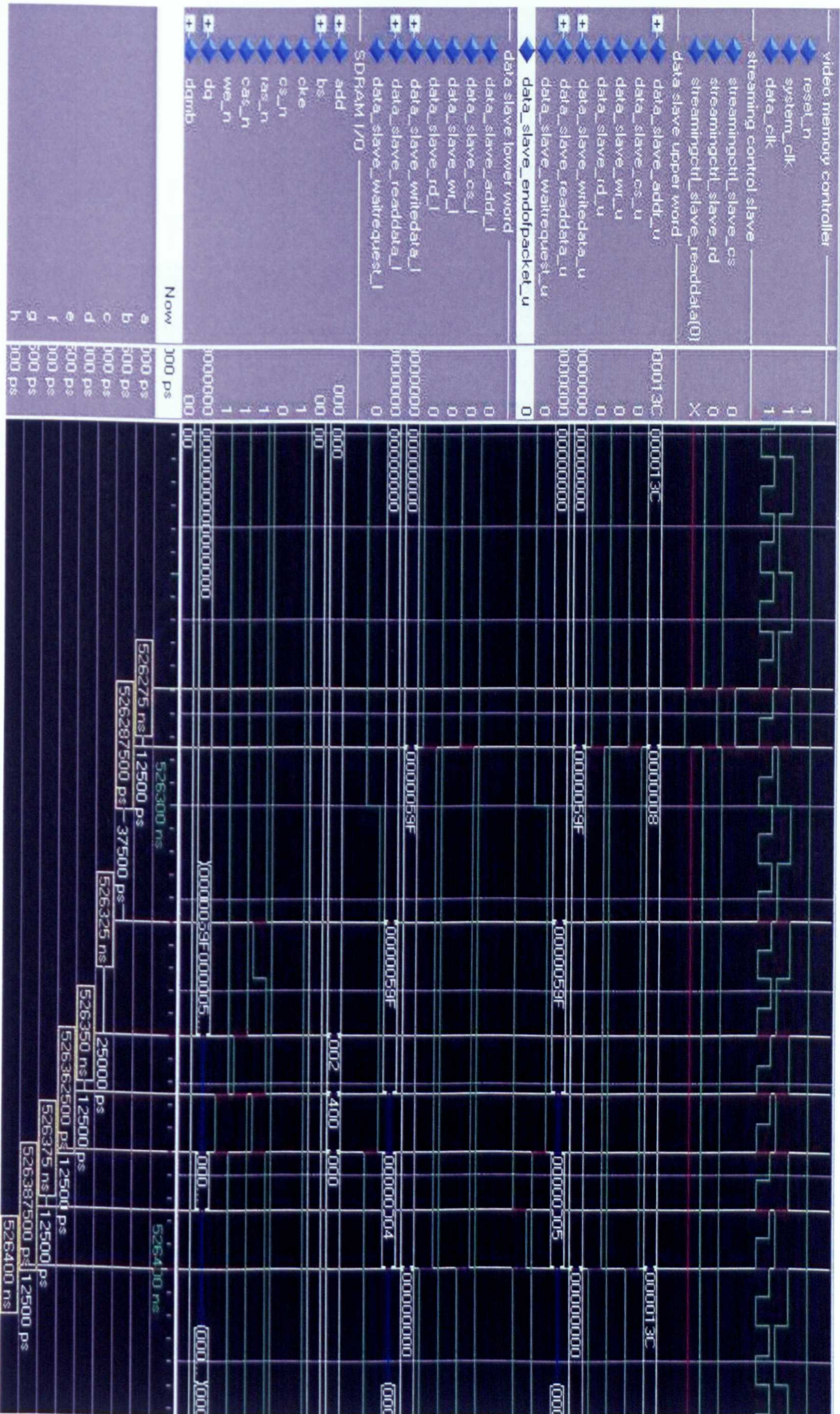


Figure 8-4 Video memory controller simulation result – burst read (BL =1)

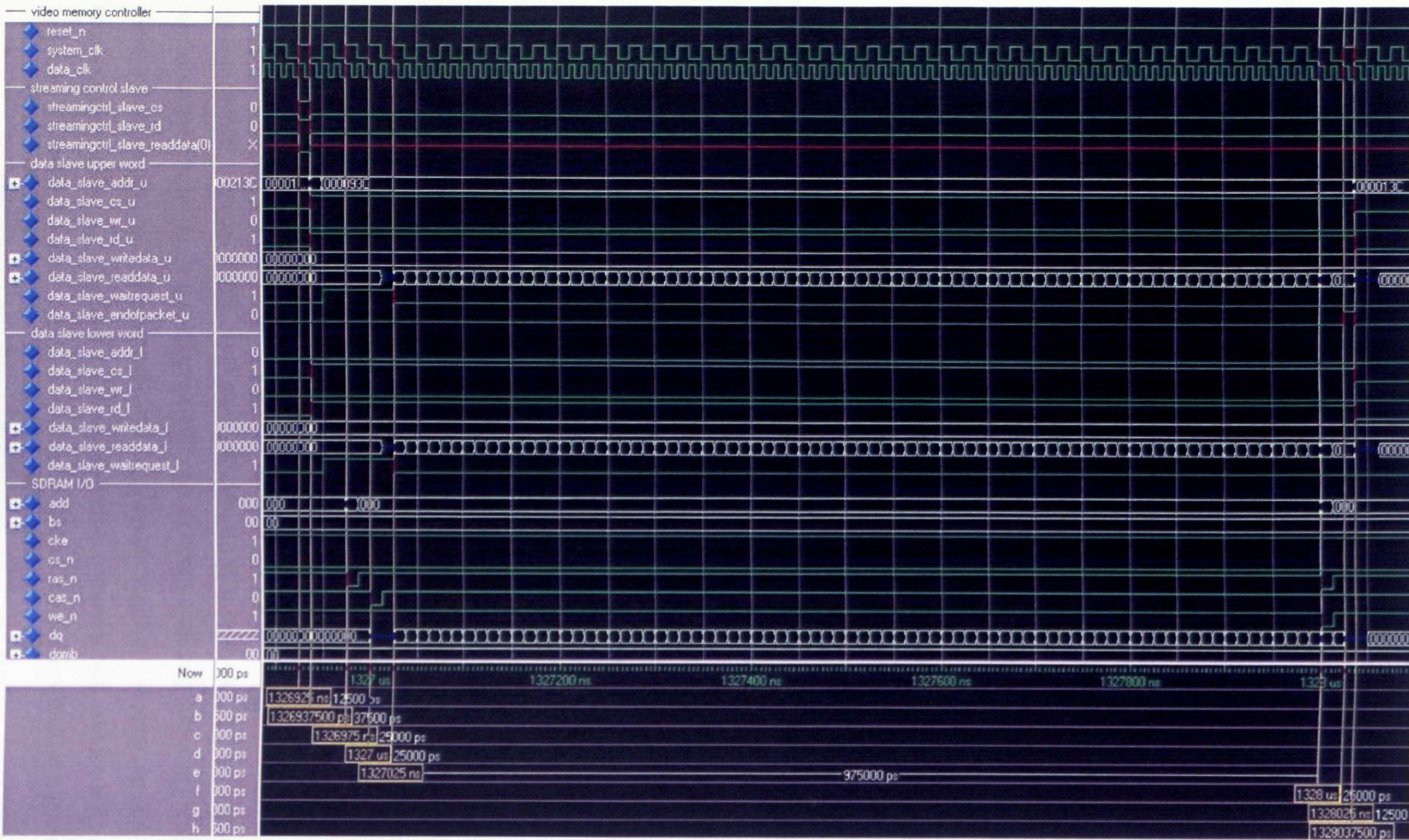


Figure 8-5 Video memory controller simulation result – burst read (BL = 80)

8.1.1.2. Video display controller simulation results and discussions

Figure 8-6 gives a view of the display timing of an entire image with size of 640x480. This figure shows that the video display starts as soon as the CPU sends configuration data into the display controller via the CPU slave port just prior to time a (see the small image). There are five time spans in this figure which represent various stages of the vertical timing. Time *a* to *b* is the vertical sync pulse width which is approximately 0.064ms. Time *b* to *c* represents the vertical back porch width which is approximately 1.03ms. The active frame period is approximately 15.37ms from time *c* to *d*. Time *d* to *e* is the vertical front porch width which is approximately 0.35ms. The total frame period is therefore approximately 16.8ms which gives a frame rate for that size of image about 60 f/s. At the end of the active vertical sync (*vs*) an interrupt is asserted (at time *e*) on the CPU slave port to inform the CPU that a frame has been displayed.

Figure 8-7 gives a zoomed view of the horizontal video timing. As shown in the figure, time *b* to *c* is the horizontal sync pulse width which is 3.84μs. Time *c* to *d* represents the horizontal sync back porch width which is 2.12μs. The active video period is from time *d* to *e* which is 25.6μs for sending 640 pixels. The last time span is the horizontal sync front porch width which is 0.48μs. The total time of displaying one horizontal line is therefore 32.04μs. There are 480 lines in this simulation so the active frame period of 480 lines should be 32.04x480≈15.37ms which matches the time value shown in Figure 8-6. At time *a* when *blank_n* is deactivated, an Avalon streaming read request to the memory controller is asserted for reading the next line data into one of the line buffers. Whenever the bus is free (no conflict in accessing the memory) this read request is handled and valid data is returned in a burst (see the small image).

8.1.1.3. Cache simulation results and discussions

Figure 8-8 shows the operation of a Cache write. In this example the CPU is writing data 0x000001E0 into the lower 32-bit word at SDRAM address 0xD8. At time *a* a write request from the CPU initialised. After 1 data clock cycle, at time *b*, this write-though request to the memory commences, and the whole write transfer completes at time *e*. At time *d* when the CPU slave detects that the write request has been granted to the memory it deasserts waitrequest and the CPU write finishes at time *d*. The total CPU write takes up **2** system clock cycles.

Figure 8-9 shows an example of Cache misses. In this example, the CPU is reading data from the lower 32-bit word at SDRAM address 0x04 (BL=1). The CPU read request is issued at time *a*. At time *b*, after the Cache has calculated it is a miss, a read request to the memory is initialised, and this request completes at time *c*. The whole CPU read finishes at time *d*. The returned data is 0x00000408. It should be noted that the upper 32-bit word is also read out from the memory at the same time so the next CPU read at this address for acquiring the upper 32-bit word would be a Cache hit. The entire Cache miss read takes **12** system clock cycles to complete when BL is 1.

Figure 8-10 illustrates a Cache hit example. The CPU is reading the upper 32-bit word from the same SDRAM address as in the last example. As mentioned before, this would be a Cache hit. It can be seen from the figure that a Cache hit takes **4** system clock cycles to complete.

From all of the Cache simulation results, a summary can be made which is a Cache write takes **2** system clock cycles, a Cache miss takes **12** cycles while a Cache hit needs **4** cycles to complete when BL is 1.

8.1.1.4. Video capture controller simulation results and discussions

Figure 8-11 illustrates the simulation result of the video capture controller with capturing one frame sized at 640x480. The total active frame period is approximately 16.35ms. An interrupt is generated immediately after *fval* becomes inactive.

Figure 8-12 gives a zoomed view of one video line capturing. The video line activates at time *a*, and lasts till time *b*. As the capture clock in this simulation is 20MHz, the total active line period is 32μs. During the inactive line period (time *b* to *c*) an Avalon streaming write request is issued to transfer the video line that has just been captured to the memory. The total data transfer time is 105μs (see small image) which takes 84 data clock cycles at a frequency of 80MHz.

The video input timing used for simulating the video capture controller was based on the information given by the camera manual. The actual timing was measured in the hardware test, which will be presented in the next section.

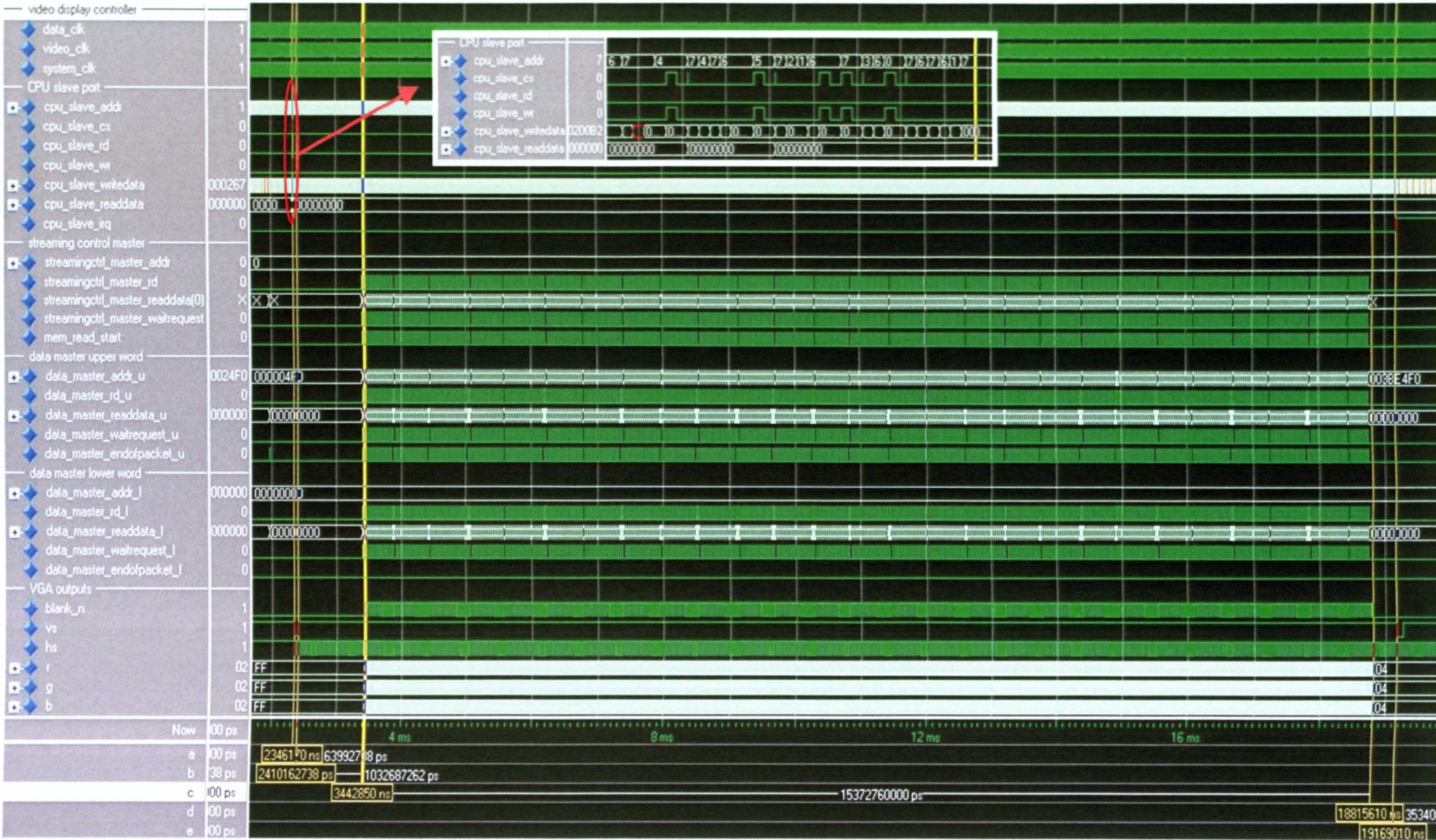


Figure 8-6 Video display controller simulation result 1

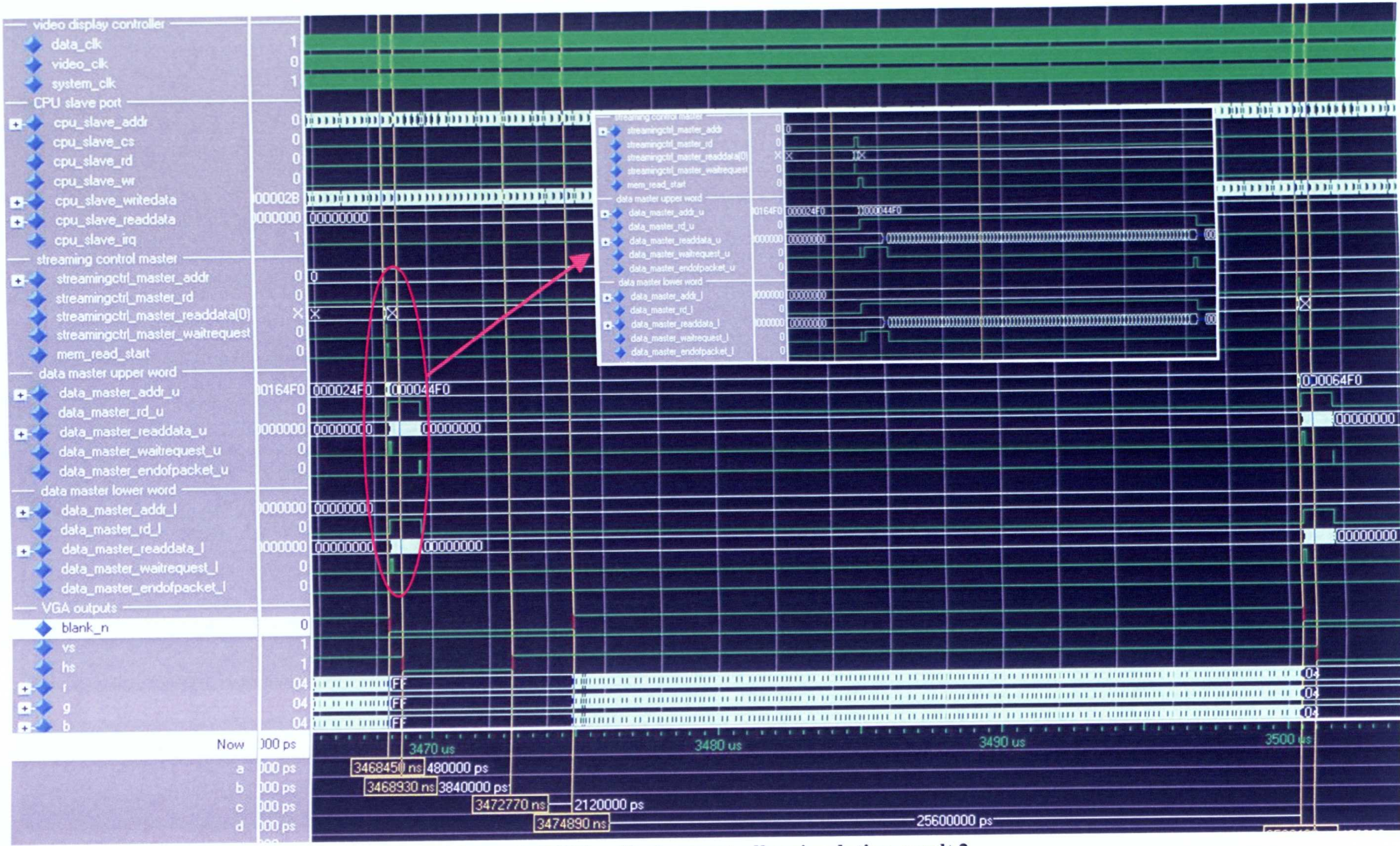


Figure 8-7 Video display controller simulation result 2

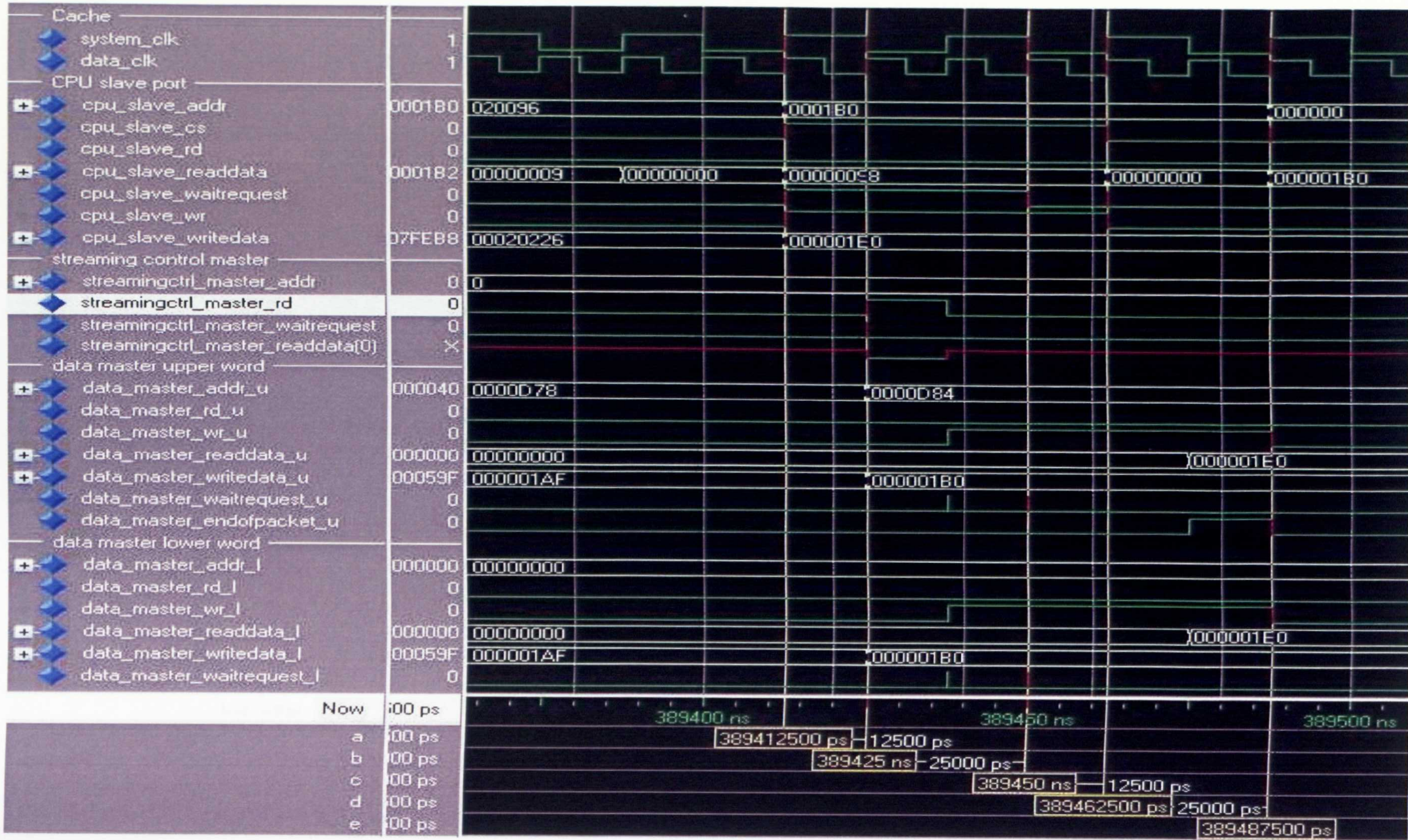


Figure 8-8 Cache simulation result – Cache write

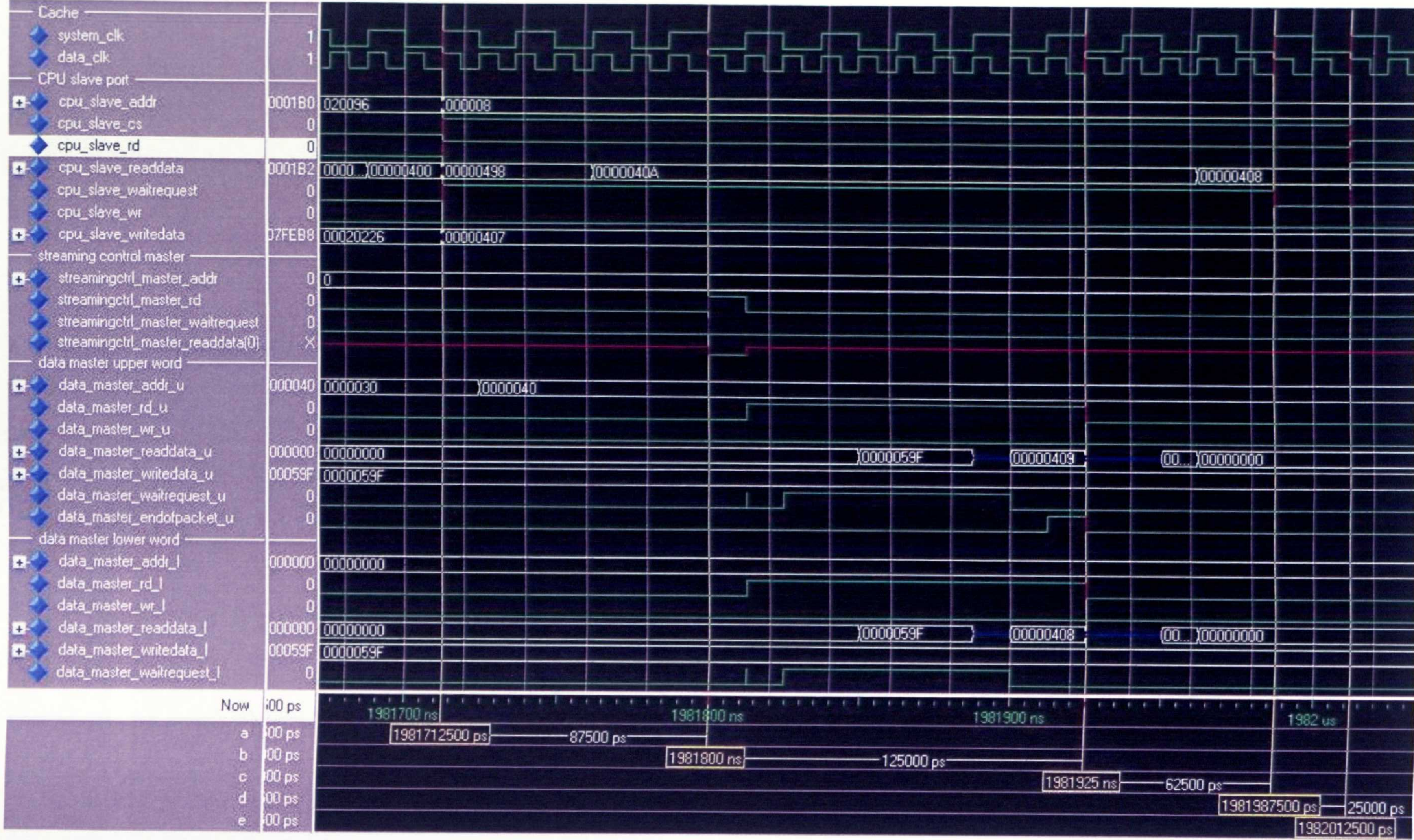


Figure 8-9 Cache simulation result – Cache miss (BL=1)

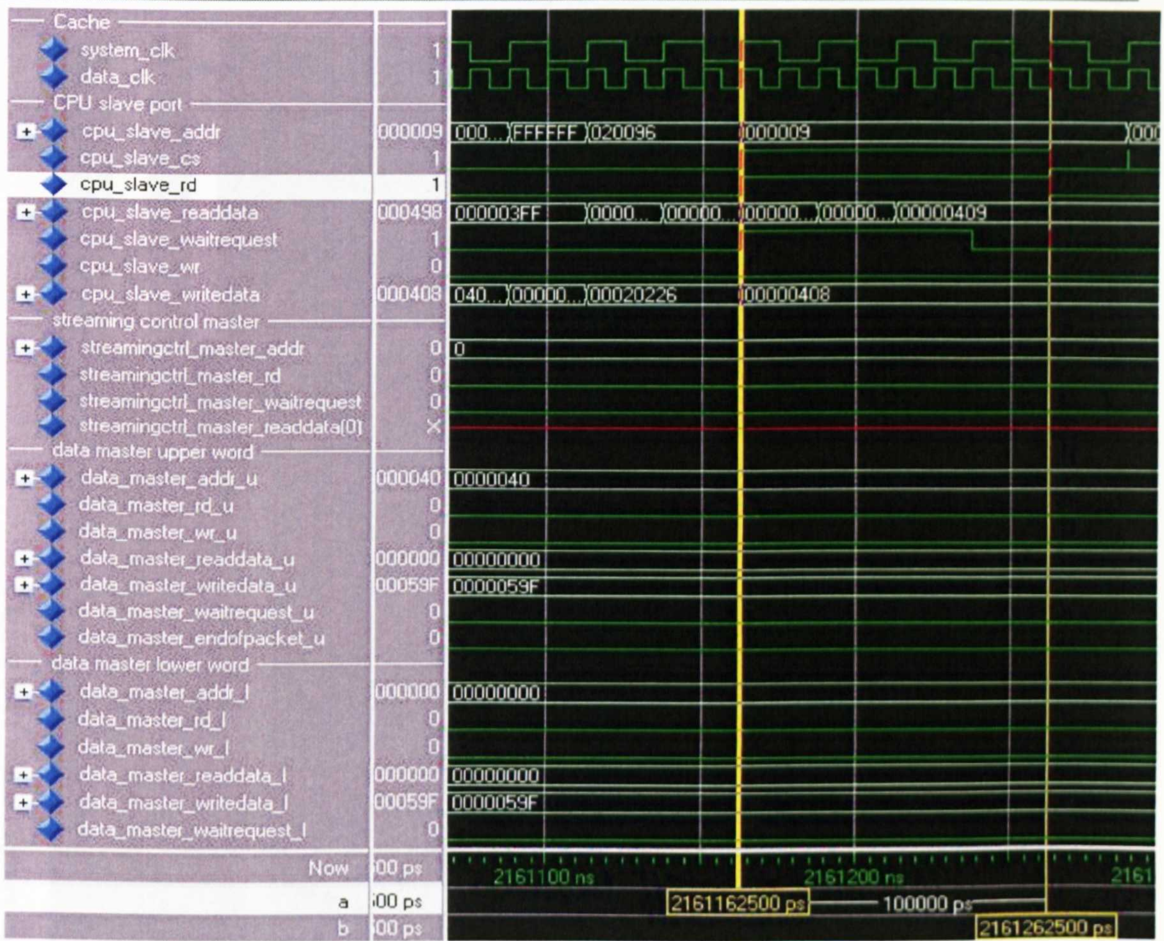


Figure 8-10 Cache simulation result – Cache hit

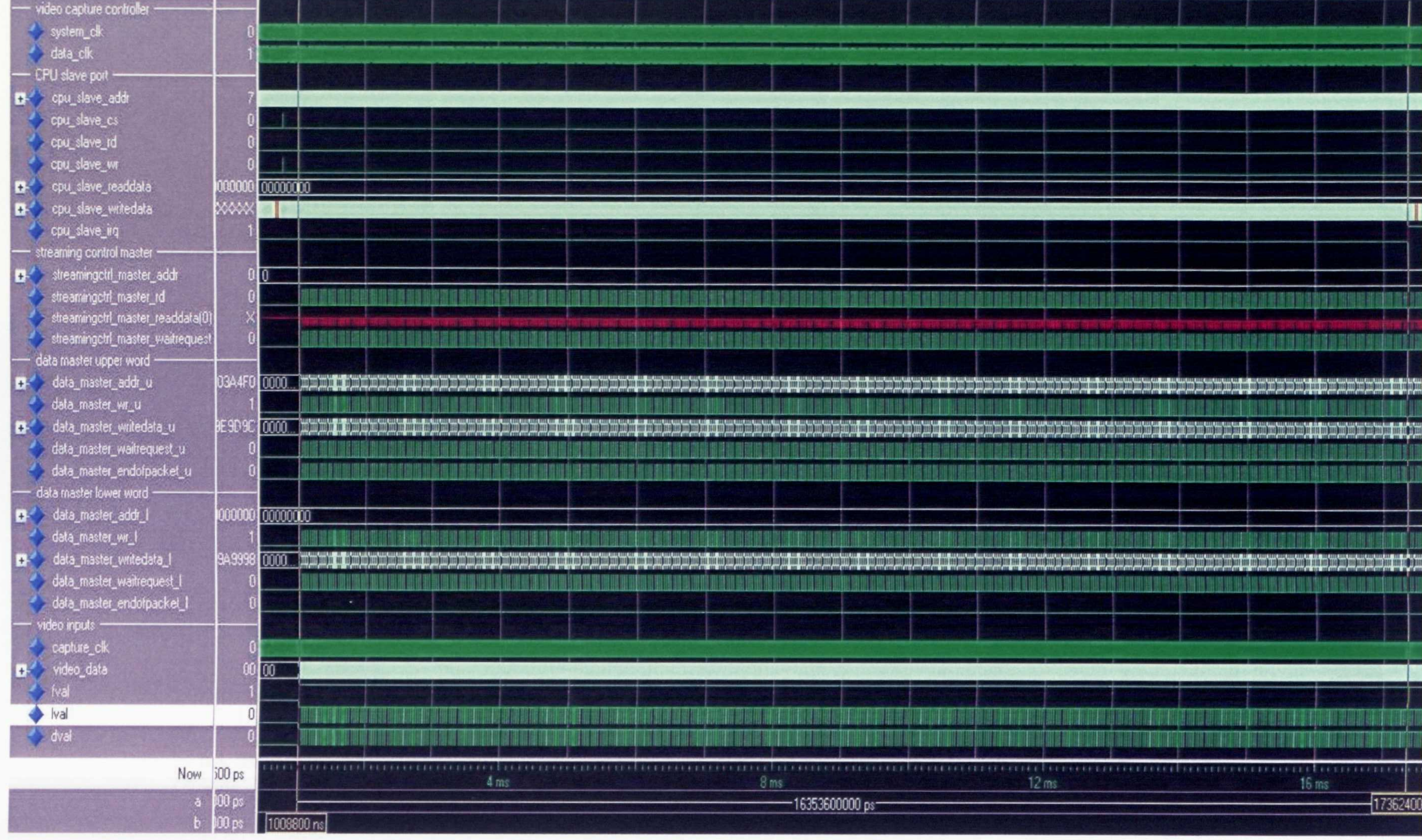


Figure 8-11 video capture controller simulation result 1

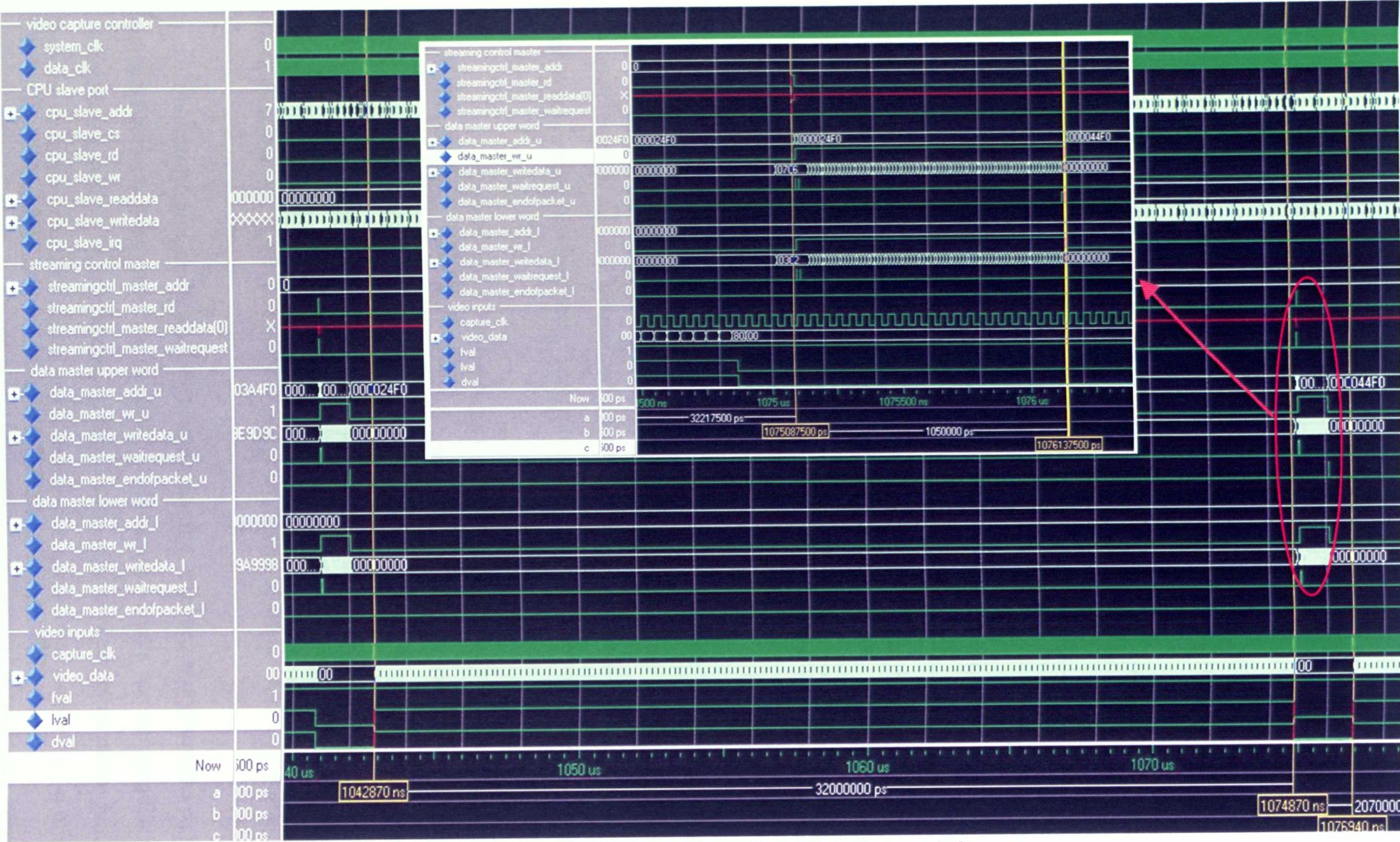


Figure 8-12 video capture controller simulation result 2

8.1.1.5. Full system simulations results and discussions

Figure 8-13 shows a full system simulation result. As this system is aimed for doing general image processing, in order to examine the accuracy of it, a Sobel edge detector was applied into this simulation example. Details of the Sobel edge detector algorithm will be explained in section 8.2. This example utilised the Quad-bank operation.

In phase 1, the video starts to be captured into bank 0, as there is no valid data in the other banks, the display controller sends out data with values of zero. In phase 2, the operations in all banks are changed. Bank 0 starts to be processed and the processed data is put into bank 3. Video is continually being captured into bank 1. In phase 3, the video display moves to bank 3. This bank contains valid data which has just been processed. So in this phase, valid data is started to be sent to the display.

Table 8-1 lists some pixel data used for simulation. These data are fed into the capture controller. So by comparing the processed data output on the display with the calculated results a judgement of the accuracy of image calculations and correctness of data transfer can be obtained.

Table 8-1 Full simulation data input sets

Column number(C) / Line number(L)	1	2	3	4	5	6	7	…	640
1	0x00	0x00	0x1C	0x38	0x54	0x70	0x8C	…	…
2	0x00	0x00	0x45	0x8A	0xCF	0x14	0x59	…	…
3	0x00	0x80	0xEE	0x5C	0xCA	0x38	0xA6	…	…
4	0x00	0x80	0x18	0xB0	0x48	0xE0	0x78		
…	…	…	…	…	…	…		…	…

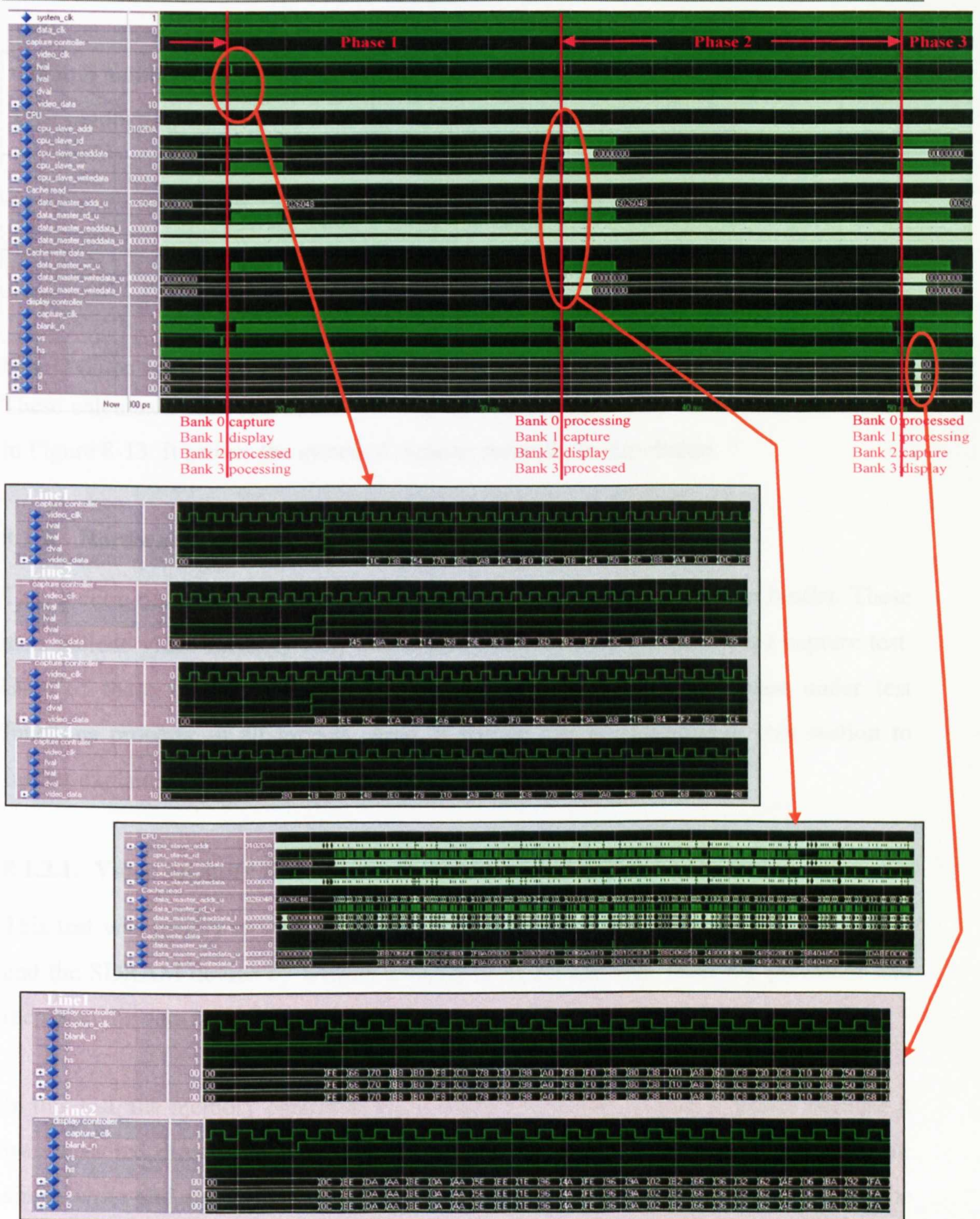


Figure 8-13 Full system simulation result

Table 8-2 lists the calculated results of the input data shown in Table 8-1 based on the Sobel algorithm. For example,

$$L2C2 = |L1C3 + 2 \times L2C3 + L3C3 - L1C1 - 2 \times L2C1 - L3C1| + |L1C1 + 2 \times L1C2 + L1C3 - L3C1 - 2 \times L3C2 - L3C3|$$

The output data simply takes the lowest 2 bytes of the calculated results. This is different from the actual implementation of the Sobel edge detector.

Table 8-2 Estimated full simulation data output sets

Column number(C) / Line number(L)	1	2	3	4	5	6	7	…	640
1	D/C								
2	D/C	0x66	0x70	0xB8	0xB0	0xF8	0xC0	…	D/C
3	D/C	0x0C	0xBE	0xDA	0xAA	0xBE	0x0A		D/C
…	D/C	…	…	…	…	…	…	…	D/C

Note: D/C-Don't care

These calculated data sets match the display outputs shown on the bottom waveforms in Figure 8-13. It proves the system functions correctly in simulation.

8.1.2 Hardware verifications

This section discusses the hardware tests undertaken for all main video blocks. These tests include video memory test, Cache test, video display test and video capture test. Each of these tests might have several sub-tests to ensure the device under test functions properly in all aspects. Also, a special test is presented in this section to further explain the solution to the multi-mastering issue.

8.1.2.1. Video memory test

This test was mainly to verify the working condition of the video memory controller and the SDRAM device by writing some data to the memory from the processor and then read the data back for checking.

In this test, the memory controller has a slightly different Avalon interface because it was mastered directly by the Nios processor data master (no Cache between them); single write and single read can only be performed on the memory controller by the Nios processor each time (no streaming transfer). There is only one 32 bit video data port so a multiplexer was placed on the data path. In order to guarantee the synchronisation of the data transfer, the data clock of the memory controller was driven by the same clock as for the processor – system clock. There is no streaming control slave port. Signal *data_slave_waitrequest_x* was activated immediately when an Avalon transfer request is issued instead of having a clock cycle delay.

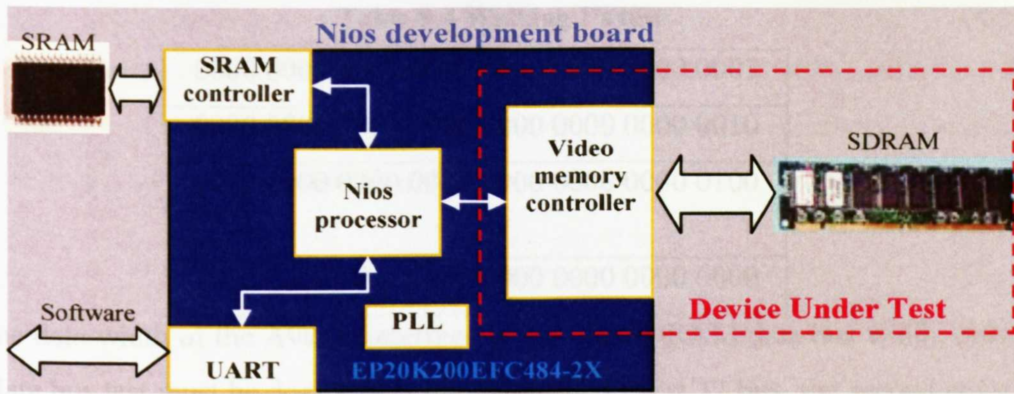


Figure 8-14 Video memory test scheme

Figure 8-14 shows the block diagram of the video memory test. Table 8-3 lists all device components under test and all the other supported components.

Table 8-3 Video memory test scheme

Device under test	Video memory controller & SDRAM
Support components	Nios processor, SRAM controller, UART, PLL
Test platform	Nios development board

Test Strategy

The test strategy has three individual tests: a data bus test, an address bus test, and a device test. They were aiming at ensuring the SDRAM device is working properly, no electrical wiring problem or missing chips and catastrophic failures, and the memory controller is able to handle Avalon transfers and send out proper control signals to drive the memory device as in simulations.

Data bus test

This test was to confirm that any value placed on the data bus by the processor is correctly received by the memory device at the other end. By using a way called "walking 1's test" [97] each bit on the data bus can be tested independently. To perform the walking 1's test, simply write the first data value in Table 8-4, verify it by reading it back, write the second value, and verify. When reach the end of the table, the test is completed.

Table 8-4 Walking 1's test

0000 0000 0000 0000 0000 0000 0000 0001
0000 0000 0000 0000 0000 0000 0000 0010
0000 0000 0000 0000 0000 0000 0000 0100 ⋮
1000 0000 0000 0000 0000 0000 0000 0000

As the data width of the Avalon interface is half the SDRAM data bus width, only 32, the data bus test must be done twice, one is for the lowest 32 bits, the second is for the highest 32 bits.

Address test

The purpose of this test is to confirm that each of the address pins can be set to 0 and 1 without affecting any of the others. The smallest set of addresses that will cover all possible combinations is the set of "power-of-two" addresses. These addresses are analogous to the set of data values used in the walking 1's test. The corresponding memory locations are 0x00001, 0x00002, 0x00004, 0x00008, 0x00010, 0x00020 etc, until 0x400000 (23 address lines). In addition, address 0x00000 must also be tested.

To confirm that no two memory locations overlap, firstly some initial data value at each power-of-two offset within the device is written. Then a new value, usually an inverted copy of the initial value, is written to the first test offset. Finally this test is verified by reading back from the memory device to check if the initial data value is still stored at every other power-of-two offset.

Device test

Once confirm the address and data bus wiring are working, it is necessary to test the integrity of the memory device itself.

For a complete device test, every memory location must be visited twice (write and verify). First pass is to write any value into every memory location, and verify it by reading it back, and the second pass is to write its inverted value into the same location and verify it. Since there is a possibility of missing memory chips, it is best to select a set of data that changes with (but is not equivalent to) the address. A simple example is an "increment test" [97].

The data values for the increment test are shown in the first two columns of Table 8-5. The third column shows the inverted data values used during the second pass of this test. There are many other possible choices of data, but the incrementing data pattern is adequate and easy to compute.

Table 8-5 Increment test

Memory offset	Binary value	Inverted value
0x0000	0000 ……. 0001	1111 ……. 1110
0x0001	0000 ……. 0010	1111 ……. 1101
. ⋮ ⋮	. ⋮ ⋮	. ⋮ ⋮
0x03F	1111 ……. 1111	0000 ……. 0000
0x400	0000 ……. 0001	1111 ……. 1110
. ⋮ ⋮	. ⋮ ⋮	. . ⋮ ⋮

Simple test programs have been written to undertake these three tests. The test result indicated that the SDRAM device is able to be driven by the memory controller and in good working condition.

8.1.2.2. Video display test

The idea of this video display test is to check if the video display controller is able to drive the Lancelot board to display images on CRT monitor correctly with user-defined information such as the resolution setting. As the memory controller was involved in this test, streaming transfer on it can also be tested (although it's still driven by the system clock). Figure 8-15 shows the block diagram of the video display test scheme. Table 8-6 lists all device components under test and all other supported components.

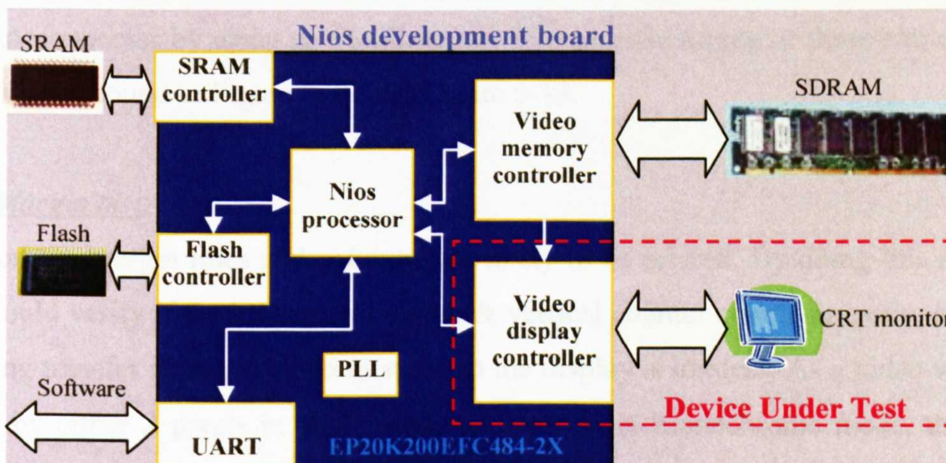


Figure 8-15 Video display test scheme

Table 8-6 Video display test scheme

Device under test	Video display controller & VGA display device
Support components	Nios processor, video memory controller, SRAM controller, UART, Flash controller, PLL
Test platform	Nios development board

Start-up procedure:

Before explaining the test scheme, a description of how to start displaying video properly is given below.

1, set up the image resolution by writing data into the resolution register;

2, set up the DMA start address by writing data into the DMA register;

3, set the video DAC in RGB mode by writing '1' into the Set DAC mode bit in the control register;

4, enable the video display by writing '1' into the Start Video bit in the control register.

If the display interrupt is enabled a subroutine must be setup to handle this interrupt.

Test Strategy

A few experiments have been undertaken to test this display block to make sure it can display an entire image properly with no missing pixel, no misalignment, no colour and positioning problem.

Test 1, VGA timing signal test

In this test, it mainly focuses on checking the video timing output signals *hs* and *vs* on the VGA connector by using an Oscilloscope. The specific timing of these two control signals can be found in Figure 5-12 and Figure 5-13.

Test2, Margin test

Pixels on the margin lines and columns are likely to be missed. By doing this margin test it could verify if the horizontal counter & vertical counter counts properly, and the streaming transfer from the video memory to the display is lossless. As a video word is formed by either 2 pixels in RGB mode or 8 pixels in monochrome mode, this test could also tell whether the pixel alignments are correct or not.

The video pixels stored in the memory device were directly written from the Nios

processor in this test. Table 8-7 and Table 8-8 show the test results that should be expected on this margin test.

Table 8-7 Video display margin test in 24-bit mode

	1	2	3	4 ………………	637	638	639	640
1	Red (0xFF0000)							
2	Green (0x00FF00)							
3	Blue (0x0000FF)							
4 … 477	Red	Green	Blue	Black		Blue	Green	Red
478	Blue							
479	Green							
480	Red							

Table 8-8 Video display margin test in 8-bit mode

	1	2	3	4	5	6	7	8	………	633	634	635	636	637	638	639	640
1	White 0xFF																
2	Black 0x00																
3	Black 0x00																
4	Black 0x00																
5	White 0xFF																
6	Black 0x00																
7	White 0xFF																
8	Black 0x00																
9 ⋮ 472	W	B	B	B	W	B	B	B	Grey 0x0F	B	W	B	W	B	B	B	W
473	Black																
474	White																
475	Black																
476	White																
477	Black																
478	Black																
479	Black																
480	White																

This test was examined by observing the image on the CRT monitor screen, especially the image margins, if it matches the image data written into the memory. Images with smaller resolution were also tested by following this method.

Test3, Static image display test

By utilising the flash memory on the Nios development board, it was possible to download a well drawn image into the SDRAM device via the flash memory, and test the video display block by sending this static image to the screen. The Nios SDK has provided a utility - "nios-run' to write a file in Motorola S-record format [98] into the flash memory. However to convert a 640x480 image in 24-bit RGB or 8-bit monochrome into the Motorola S-Record format, it need a few more steps.

a) Crop a 24-bit or 8-bit bitmap (BMP) file into size 640x480
b) Export this file into a hex file by using tools like "010 edit" [99]. Only the data field in the BMP file is needed to be converted
c) Use a custom PERL program to take out the line feed and new line characters from the hex file
d) Use bin2srec utility [100] to convert this hex file into a Motorola S-record file which is executable by the Nios system
   Usage: bin2srec <options> INFILE > OUTFILE
e) run the "nios-run" script to write the flash file into the flash memory
   Usage: nr –x [File Name] [Flash based address]

After uploading the image into the flash memory, then it's simple to just read the flash memory content and write it though to the SDRAM memory, then follows the video display start-up procedure to get the image displayed.

Test results were examined by comparing the displayed image sent out from the SIPS with the original image in observation.

8.1.2.3. Cache test

The purpose of the Cache test is to check if the Cache is functioning properly on both hit and miss scenarios working along with the actual memory device. In this test, the memory controller is driven by the data clock. Figure 8-16 shows the block diagram of the Cache test scheme. Table 8-9 lists all device components under test and all other supported components.

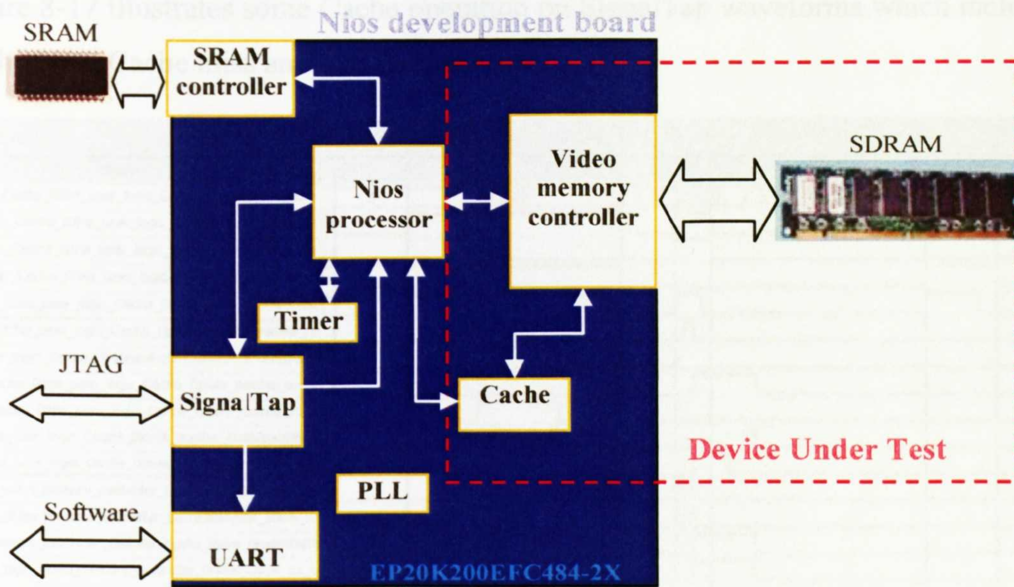


Figure 8-16 Cache test scheme

Table 8-9 Cache test scheme

Device under test	The Cache, multi-mastering issue
Support components	Nios processor, video memory controller, SRAM controller, UART, Nios Timer, PLL, SignalTap II
Test platform	Nios development board

Test Strategy

The Cache test is similar to the device test for the memory controller described in section 8.1.2.1. However it's mainly to check the integrity of the communication channel between the main memory and the Cache rather than checking the SDRAM. Moreover a simple performance comparison between Cache miss and Cache hit was also investigated. Apart from the normal data checking, some performance analyses were also undertaken by using SignalTap II.

SignalTap II is a system-level debugging tool that captures and displays real-time signals in a SOPC design. By using a SignalTap II Embedded Logic Analyzer (ELA) in systems generated by SOPC Builder, designers can observe the behaviour of hardware (such as peripheral registers, memory buses, and other on-chip components) in response to software execution.

Figure 8-17 illustrates some Cache operation on SignalTap waveforms which include a Cache hit, a Cache miss and two Cache writes.

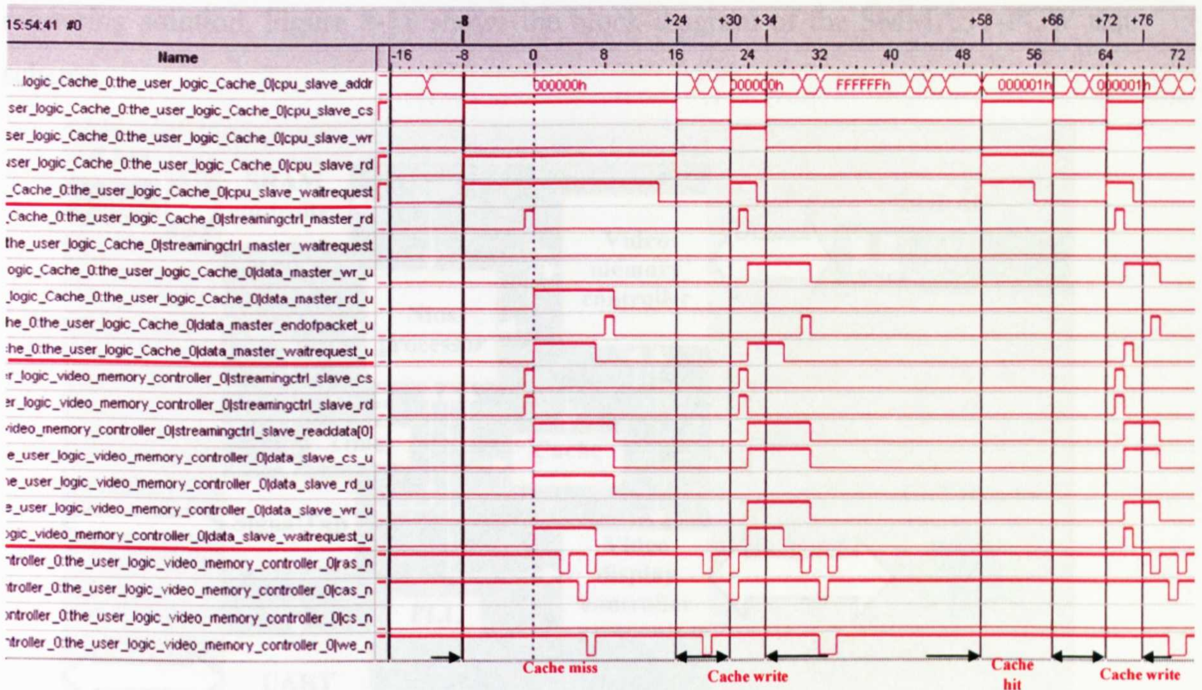


Figure 8-17 Cache hit/miss operation waveform in SignalTap II

In this test, the data clock frequency is 80MHz, the system clock frequency is 40MHz. Active to Read/Write Command delay time *tRCD* = 2 data clock cycles, CAS latency *tCL* = 2 data clock cycles.

From the waveform it can tell a Cache miss takes 24 data clock cycles to complete which is **12** system clock cycles while a Cache hit uses 8 data clock cycles which is **4** system clock cycles, if there is no other conflict in accessing the memory. A Cache write always takes 4 data clock cycles which **2** system clock cycles to complete if there is no other conflict in accessing the memory. These results match the simulation results. They will be used to estimate the processing performance on the SIPS in the next section.

A Nios Timer was also used to estimate the timing of both Cache hit and Cache write.

Test results were examined by running a software program on the SIPS to automatically verify the read data against the write data.

8.1.2.4. SMMAST-PCW test

By combining the Cache and video display in the system it is possible to test the multi-mastering solution. Figure 8-18 shows the block diagram of the SMMAST-PCW test scheme.

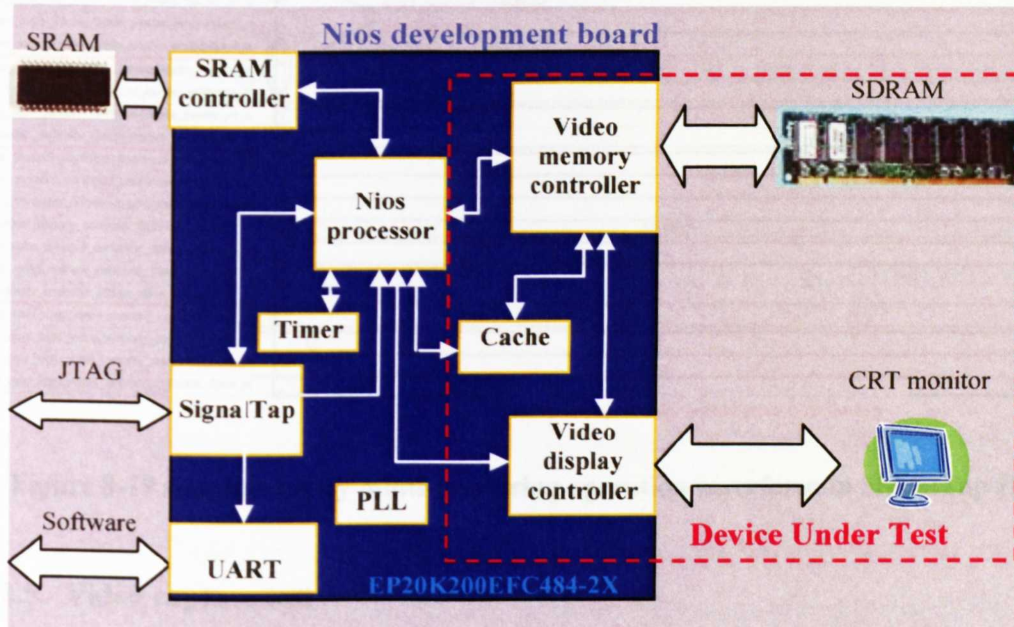


Figure 8-18 SMMAST-PCW test scheme

This idea of testing this multi-mastering transfer is to check if the memory can be operated by both the Cache and the video display controller at the same time with continuously reading and writing from/to the memory while the video is displaying. The simplest way to do that is to continuously invert an image on the display. The initial image is stored in the memory. By reading the data from the memory via the Cache and writing the inverted results back to the memory it can be observed that the original and the inverted image keeps altering on the screen. The images used for inversion was the ones used for the video display margin tests.

This test was verified by checking the following two scenarios.

1) The inversion stops occurring on the screen: this could happen if conflicts occur but can't be solved properly, then one of the masters would be stalled forever.
2) By slowing down the inversion, it can check how each pixel is inverted on the screen. If there are any pixels aren't inverted but the others are, then it means there is missing transfer between the Cache and the memory.

Figure 8-19 shows how a conflict is solved when the Cache and the video display

controller present read request to the memory at the same time. In this example, the video display controller won the arbitration and gained access to the memory slave first.

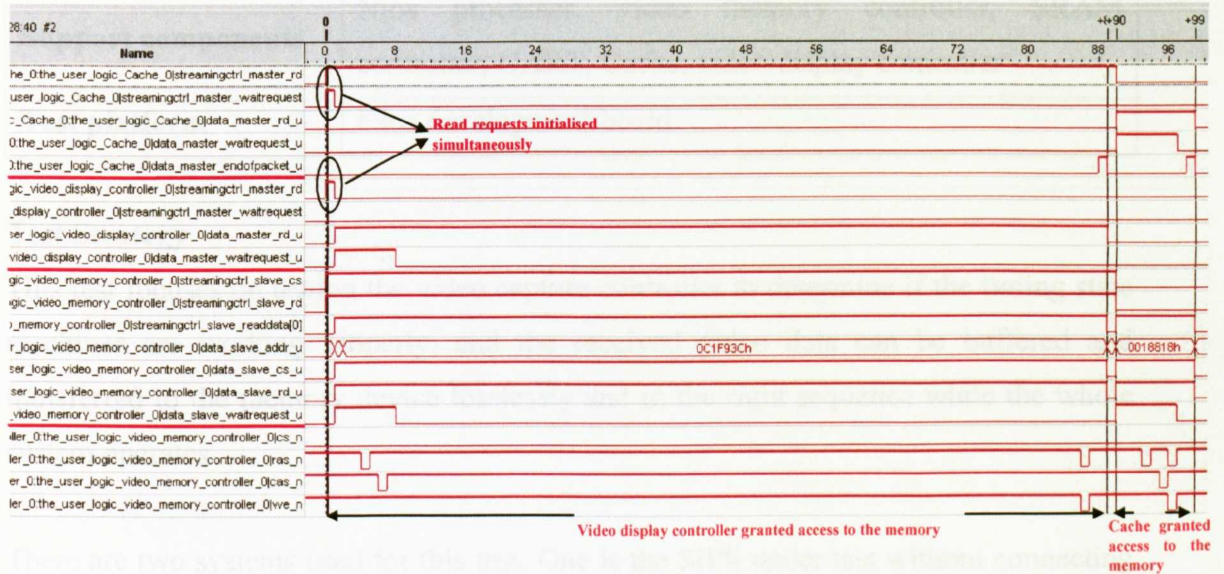


Figure 8-19 simultaneously multi-mastering operation waveform in SignalTap II

8.1.2.5. Video capture test

This test was mainly to test the video capture controller, if it is compatible with the rest of the system while multiple video masters exist. Another Nios system was used to generate the video timing and data signals as the camera interface card was still under manufacture when tests were performed. Figure 8-20 shows the block diagram of the video capture test scheme. Table 8-10 lists all device components under test and all other supported components.

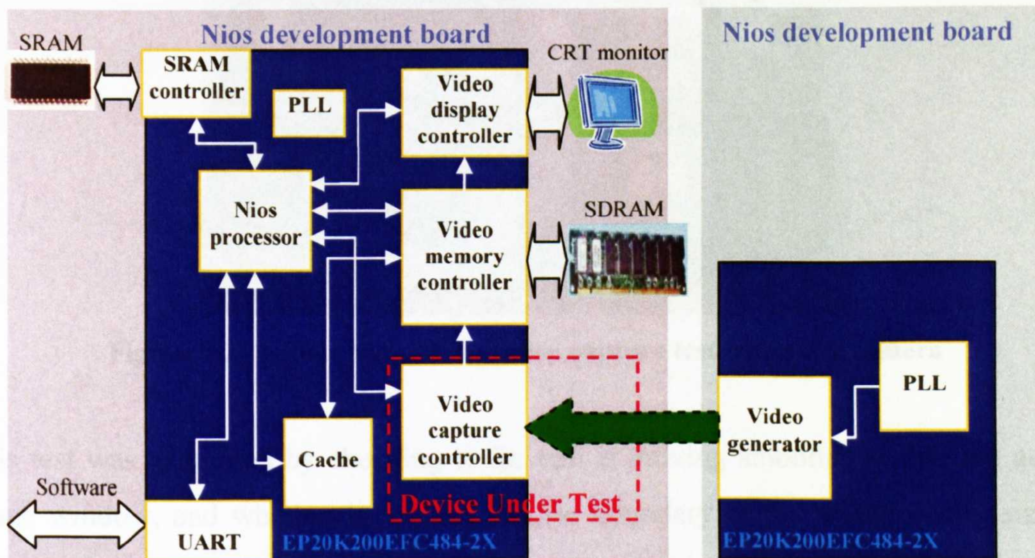


Figure 8-20 Video capture test scheme

Table 8-10 Video capture test scheme

Device under test	Video capture controller
Support components	Nios processor, video memory controller, SRAM controller, UART, Cache, video display controller
Test platform	Nios development board

Test Strategy

This test focuses on testing the video capture controller to determine if the timing state machines are working properly, and the received video data can be buffered and transferred to the memory device losslessly and in the right sequence while the whole system operates.

There are two systems used for this test. One is the SIPS under test without connecting to the external video camera, and the second one is simulating a video camera and generates video timing control and data signal. The signals that the auxiliary system produces are *LVal*, *FVal*, and 8-bit *data* representing the monochrome video pixel.

These two systems are connected via a ribbon cable (as shown in Figure 8-21). The video that the second board (left) generates is a white bouncing ball on a black background (Figure 8-22).

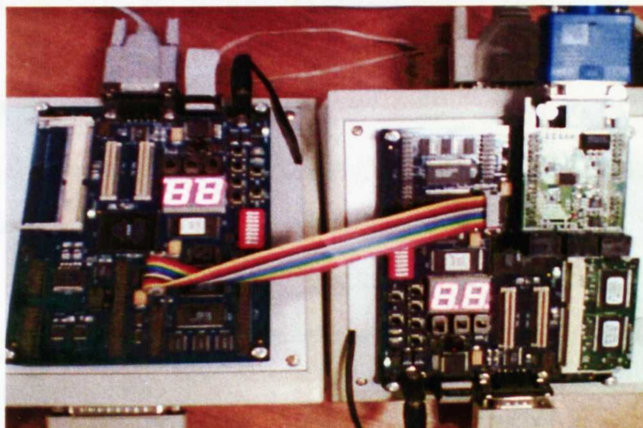


Figure 8-21 system view of the video capture test without a camera

This test was examined by checking if the ball is moving smoothly within the active frame window, and whether it can reaches the boundary of the window with size of 640x480. This boundary could be set by writing white color (0xFF) into the top, bottom lines and left, right columns.

Figure 8-22 video capture test result

8.1.2.6. Full system test

This test was to test the full system together with the CameraLink camera and the camera interface card plugged in.

Start-up procedure:

Before using the camera, it is necessary to configure the camera to the right mode so that the SIPS can work properly. What needed to be set typically is the resolution, exposure time, gain, offset and trigger control. Details of the message format and control commands can be found in the camera manual [63].

By using the UART software routine nr_uart_txchar and nr_uart_rxchar, it is able to send or receive a character to/from the camera via the custom defined camera UART each time. For example:

nr_uart_txchar ('c', na_camera_uart);

It is used to send a character 'c' to the serial device via the camera UART. Details of the software can be obtained in [80].

The interrupt and capture controller setup procedure are the same as the video display controller's. Figure 8-23 and Figure 8-24 show two sample images taken by the CameraLink camera and displayed on a CRT screen. No processing is applied into these images. Capture and display resolution are both 640x480.

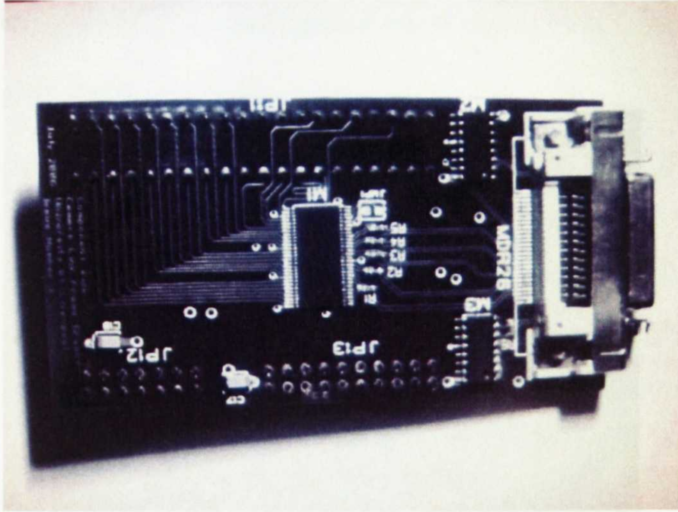


Figure 8-23 Test image – camera interface card (original)

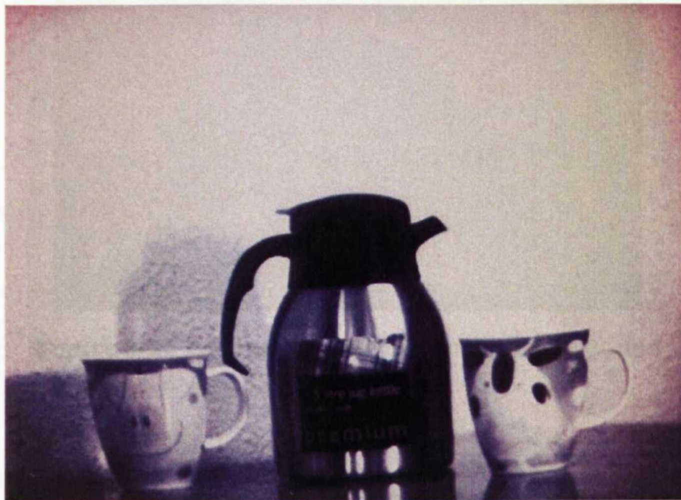


Figure 8-24 Test image – mugs and jug (original)

This CameraLink camera is sensitive to light so a proper lighting is ideally required to be set up to obtain high quality images. Figure 8-25 shows an example of an ideal testing environment.

Although there is not sufficient memory to generate SIPS in the 24-bit mode, some tests have been undertaken to test the SIPS in 24-bit mode by reducing the capture controller memory down to 8-bit mode while keeping the display memory 24-bit mode. When transferring the 8-bit captured video data pixel to the memory it always transfers it as a 24-bit data by mapping the 8-bit monochrome into r, g and b colour field. When the Nios processor starts processing images, it treats the data as 24-bit RGB and processes r, g and b separately although they are actually the same. However, this test is not presented in this thesis. The testing results presented in this thesis are all based

on 8-bit grey scale.

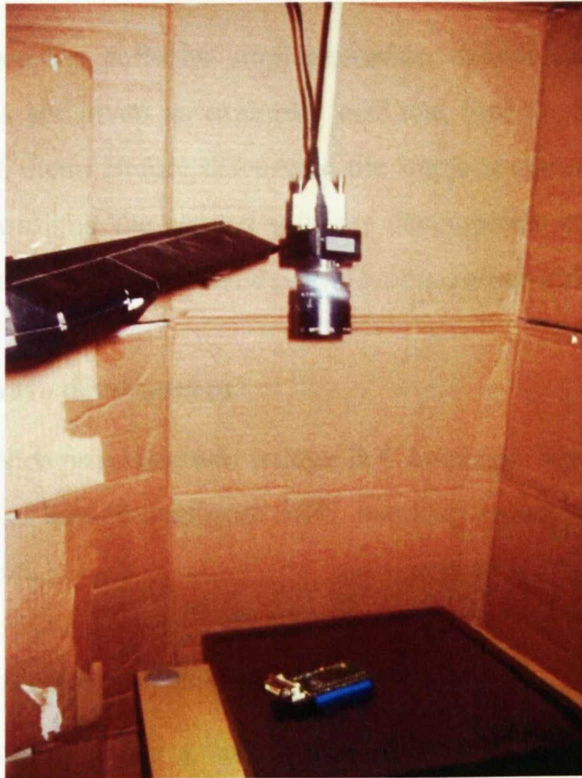


Figure 8-25 An example of the testing background

8.2 Implementation of various image processing algorithms and real-time performance analysis

This section describes the software implementation details on SIPS. Five image processing algorithms are given as examples and real time performance analysis is presented for each of them. Before discussing the implementation details of specific image processing algorithms, the general software development description for setting up SIPS and testing routines as well as the optimisation coding style are given.

8.2.1 General software development

The software for the Nios processor was written in C language. This section gives some C program examples of how to configure SIPS, handling interrupts and implementing multiple-bank operation.

8.2.1.1. Setting up video display & capture

The video display controller has a base address of 0x4a0. To configure the video display it needs to set:

```
unsigned int *VGA=(unsigned int*)( 0x4a0 );
VGA [4] = 0x28001e0; //resolution register 640x480
VGA [7] = GlobaProcessingBankNo<<22; //DMA address register
VGA[0] = 0x8; //Set up RGB mode
VGA[0] = 0x4; //enable display
```

The video capture controller's base address is 0x4e0. To configure the video capture what can be set is:

```
unsigned int * CAMERA =(unsigned int*)( 0x4e0 );
CAMERA[7]=GlobalCaptureBankNo<<22; // DMA address register
CAMERA[0]=0x4; //enable capture
```

8.2.1.2. Implementing multiple bank operation & interrupt services

To implement multiple bank operation, both the display and capture interrupt must be first enabled. The nr_installuserisr function is used to install a user interrupt service routine(ISR) for a specific interrupt number [101]. In SIPS, the interrupt number for display is 24 while capture is 23.

```
nr_installuserisr(24,VGAISR, context);
```

```
nr_installuserisr(23,CAMERAISR, context);
```

In the given example, VGAISR is the interrupt handler for the video display; CAMERAISR is the interrupt service routine for the video capture.

```
void VGAISR(int context)        {
        unsigned int *VGA=(unsigned int*)( 0x4a0 );
        if (GlobaProcessedBankNo == 0)
                GlobaDisplayBankNo =3;
        else
                GlobaDisplayBankNo=GlobaProcessedBankNo -1;
        GlobaDisplayIrq = 1;
        VGA[7] = GlobaDisplayBankNo<<22; //change DMA address register & clear irq
}

void CAMERAISR(int context)        {
        unsigned int *CAMERA = (unsigned int*)( 0x4e0 );
        if (GlobaProcessingBankNo == 3)
                GlobalCaptureBankNo =0;
        else
                GlobalCaptureBankNo=GlobaProcessingBankNo+1;
        GlobalCameraIrq = 1;
        CAMERA[7]=GlobalCaptureBankNo<<22; //change DMA address register & clear irq
}
```

In the main function, an example of quad-bank operation implementation is given as follows.

```
if (GlobalCameraIrq ==   1)        {
        //Image processing algorithms start
        //.................
        //Image processing algorithms end
        if (GlobaDisplayIrq == 1) {
                GlobalProcessedBankNo = GlobalProcessingBankNo;
                if (GlobalProcessingBankNo == 3)
                        GlobalProcessingBankNo = 0;
                else
                        GlobalProcessingBankNo++;
                GlobaDisplayIrq = 0;
                GlobalCameraIrq =0;
        }
}
```

8.2.1.3. Memory read/write operations

The base address of the Cache CPU slave is 0x4000000. To access a memory location it needs to do:

```
unsigned int *Data = (unsigned int*)( 0x4000000 );
Data[j*1024+i]=0xffffff;
```

In this example, *Data[j*1024+i]* is for the data at column *i* in row *j* of the SDRAM. Number 1024 means there are 1024 32-bit words contained in each SDRAM row. As in SIPS each memory row stores one video line. *j* actually means the video line number. In 24-bit mode, *i* means the *i'th* pixel in the current video line (pixel stars at 0). For example, to write data 0xfefefe into the 7th pixel in row 10 what needed to be written is *Data[10*1024+7]*=0xfefefe. However, in 8-bit video mode, *Data[j*1024+i]* includes the *(i*4)'th, (i*4+1)'th, (i*4+2)'th* and *(i*4+3)'th* pixel in row *j*. Therefore for example, to write data 0x3e into pixel 7 in row 10 it needs to write *Data[10*1024+1]= 0x3e<<24*. To read the 7th pixel in row 10 it needs to put *Data[10*1024+1]>>24 & 0xff*.

8.2.1.4. Timer function

In order to analysis the performance of specific image processing algorithm running on SIPS, a Timer was used to measure the processing time. An example is given below.

```
dwStartTick = GetTickCount();
        image processing programs........
lTicksUsed = GetTickCount();
printf("%d\n",dwStartTick - lTicksUsed - timer_overhead);
```

Two timer check statements are placed at the entry and the end of the program. Subroutine 'GetTickCount()' is used to obtain the current counter number of the timer. The Timer is driven by the system clock and every timer tick is one system clock period. So by multiplying the number of ticks that the timer has elapsed with the system clock period it is able to tell how much time it takes to process an image frame.

8.2.2 Discussion of software coding style – optimisation issue

Experimental work has been undertaken to find out how the software coding style would affect the image processing efficiency of SIPS. As the Nios processor is the one

which does the calculations, the SIPS efficiency really depends on the way the processor processes data. For example, data type of 'unsigned int' is more efficient than 'int'. So if the data being processed doesn't care of signed operation, then use 'unsigned int' as much as possible.

As the CPU data are stored in SRAMs which has registered input and output in order to satisfy the system performance requirement, by decreasing the number of times of accessing that memory could consequently increase the system efficiently. Take the following program as example; it has two 'for' loops (which is very common to be used in processing a block of image data). The total number of times that the inner 'for' loop is executed is 480x160. Figure 8-17 shows how many clock cycles it takes to change the condition of these two loop statements (from +34 to +58).

```
for (j=0;j<480;j=j+1)      {
        for (i=0;i<160;i=i+1)      {
                Data [j*1024+i] = ~ Data [j*1024+i];
        }
}
```

However, by decreasing the execution times of the 'for' loops it could increase the processing efficiency. For example, the following program executes the inner 'for' loop eight times less than the previous example. Table 8-11 and Table 8-12 show the processing power comparison on these two examples.

```
for (j=0;j<480;j=j+1)      {
        for (i=0;i<160;i=i+8)      {
                Data[j*1024+i]     = ~ Data [j*1024+i];
                Data [j*1024+i+1] = ~ Data [j*1024+i+1];
                Data [j*1024+i+2] = ~ Data [j*1024+i+2];
                Data [j*1024+i+3] = ~ Data [j*1024+i+3];
                Data [j*1024+i+4] = ~ Data [j*1024+i+4];
                Data [j*1024+i+5] = ~ Data [j*1024+i+5];
                Data [j*1024+i+6] = ~ Data [j*1024+i+6];
                Data [j*1024+i+7] = ~ Data [j*1024+i+7];
        }
}
```

The only problem of doing this is it would increase the CPU instruction memory size. However, the CPU instruction master is a latency-aware master which hence has a

higher efficiency.

Apart from the Nios processor, how efficient the Cache is used also affects the overall processing efficiency. Take the following program line for example.

```
Data[j*1024+i]=Data[j*1024+i]>>8 + Data[(j+1)*1024+i+1]+ 2*(Data[j*1024+i+1]>>24);
```

In this example, the last read data *Data[j*1024+i+1]* could have been a hit however it is a miss now because *Data[(j+1)*1024+i+1]* overwrote that Cache line. In order to avoid this happening, it'd better put all processed data from the same memory row together.

```
Data[j*1024+i]= Data[j*1024+i]>>8+ 2*( Data[(j)*1024+i+1]>>24) + Data[(j+1)*1024+i+1];
```

By doing this the second read *Data[(j)*1024+i+1]* would be a hit.

If a data is used several times in a single calculation while that Cache line could possibly be overwritten. For example,

```
Data[j*1024+i]=abs(Data[j*1024+i] + Data[(j+1)*1024+i])+ abs(Data[j*1024+i]- Data[(j+1)*1024+i]);
```

In this case, *Data[j*1024+i]* was read twice, and both of them were a Cache miss. However, it can be improved by saving those data into temporary variables before calculating. For example,

```
Temp1= Data[j*1024+i];
Temp2= Data[(j+1)*1024+i]
Data[j*1024+i]=abs(Temp1+Temp2)+abs(Temp1-Temp2);
```

By doing this, *Data[j*1024+i]* was only read once, and so was *Data[(j+1)*1024+i]*. The efficiency increased because to access the CPU data memory is faster than accessing the SDRAM.

The final thing needed to be emphasised is the division operation. It takes up the Nios processor lots of clock cycles to implement this if the Dividend is not the power of 2. Furthermore, the calculating time of this kind of division varies depending on the actual image content. So be careful of using division which is not the power of 2. However, this issue could be improved by implementing custom instructions for the Nios processor [102].

8.2.3 System performance analysis

This section gives an overview of the overall system performance for processing images. Section 5.6 describes if the pixel clock from the camera is 20MHz, then the frame rate is 61f/s. Suppose SIPS is taking video with size 640x480 in and displaying the same size video. The system clock is 40MHz and data clock is 80MHz. Within one second, the capture block performs 480x61 streaming writes to the memory while the display block performs 480x60 streaming reads. In 8-bit grey level mode, the total time that the Avalon bus occupied for the video capture and display is $[480\times 60\times (640\div 8+9)+480\times 61\times (640\div 8+4)]\times 12.5ns = 62.784ms$. So the rest of time remains for the Nios processor to perform real-time processing. The percentage of this time is approximately 93.712%. In 24-bit RGB mode, the real-time processing efficiency is 76.297%. However, this calculation only gives an overview of how much time is given for SIPS to process real-time data. As for how much data the SIPS can process it must be analysed with specific processing algorithms.

8.2.4 Implementation of five image processing algorithms and performance analysis

This section describes how the image processing algorithms are implemented on SIPS. Five examples are given and they are the inversion, Sobel edge detector, Gaussian blur filter, Sharpness filter and feature correlation.

All test results presented in this section are based on images size of **640x480** in **8-bit grey level mode**. General test conditions were: Camera pixel clock is **20MHz** with frame rate is 61 f/s. **Zero gain and offset** is applied to the camera. The video display pixel clock is **25MHz** with frame rate of 60f/s. System clock is **40MHz** while the data clock is **80MHz**. All of the following tests were undertaken based on the two test images shown in Figure 8-23 and Figure 8-24. So there would be two test result images for each image processing algorithm. All tests for the same test image were undertaken with same lighting, same camera to object distance and same camera focus.

8.2.4.1. Inversion

This algorithm simply toggles every bit of all active pixels. So for example if the original pixel is black, then the inverted result is white. Figure 8-26 and Figure 8-27

show two test result images with inversion applied. Only the right half is inverted in these two images.

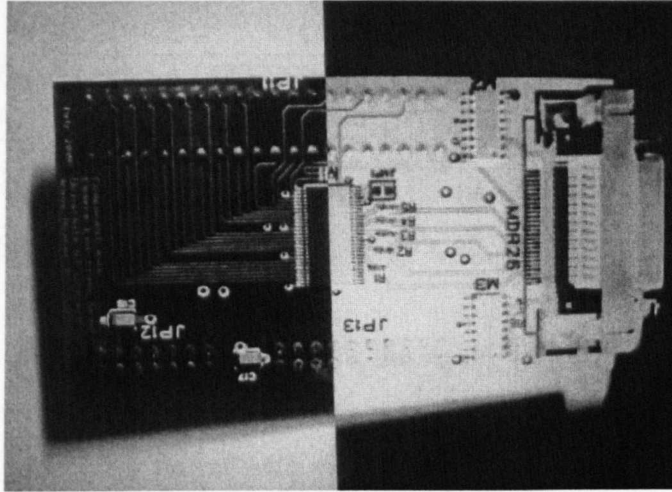


Figure 8-26 Inversion image – camera interface card

Figure 8-27 Inversion image – mugs and jug

An example of software program for doing this inversion is given as following.

```
for (j=top_boundary;j<bottom_boundary;j=j+1)        {
        for (i=left_boundary;i<right_boundary;i=i+1) { // 4 pixels per 32 bit word
                ProcessedData[j*1024+i] = ~ProcessingData[j*1024+i];
        }
}
```

Table 8-11 shows a performance analysis for this program. This table tells that the processing frame rate of video with size 640x480 is 19.6f/s.

Table 8-11 Inversion processing power analysis – without optimisation

Processing window size	Number of Timer ticks (average)	Processing time (ms)	Processing power (f/s)
80x60	32222	0.8055	1241
160x120	129064	3.226	309
320x240	511039	12.775	78
640x480	2036506	50.912	19.6

However, if this program is optimised like the example given in section 8.2.2, the processing power can be increased as shown in Table 8-12. This table shows with optimisation, the processing power with video size of 640x480 is increased to 29.5f/s, which is almost 50% more than without optimisation.

Table 8-12 Inversion processing power analysis – with optimisation

Processing window size	Number of Timer ticks (average)	Processing time (ms)	Processing power (f/s)
80x60	26978	0.674	1482
160x120	87167	2.179	459
320x240	343257	8.581	116
640x480	1355111	33.877	29.5

8.2.4.2. Sobel edge detector

The Sobel operator performs a two-dimension spatial gradient measurement on an image and so emphasises regions of high spatial gradient that correspond to edges. Typically it is used to find the approximate absolute gradient magnitude at each point in an input greyscale image [103]. Figure 8-28 and Figure 8-29 show two test result images with the Sobel edge detector applied.

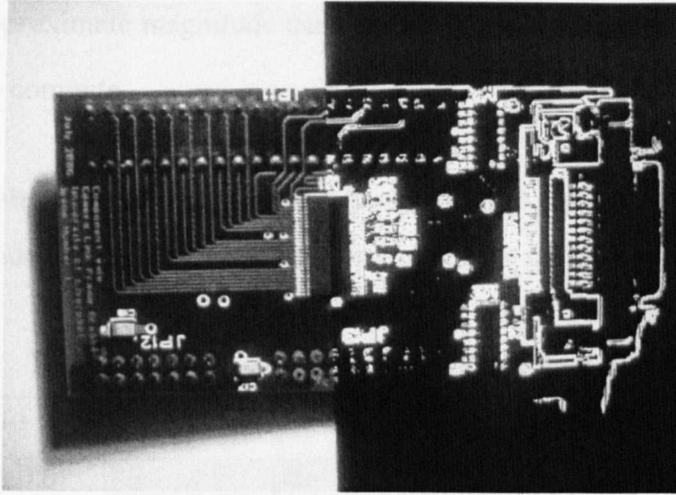


Figure 8-28 Sobel edge detector image – camera interface card (threshold=60)

Figure 8-29 Sobel edge detector image – mugs and jug (threshold=60)

The Sobel detector consists of a pair of 3×3 convolution kernels as shown in Figure 8-30. One estimates the gradient in the x-direction (columns) and the other estimates the gradient in the y-direction (rows).

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

Gx

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

Gy

Figure 8-30 Sobel convolution kernels

The kernels can be applied separately to the input image, to produce separate measurements of the gradient component in each orientation (call these *Gx* and *Gy*). The absolute magnitude of the gradient at each point can be calculated by:

$$|G| = \sqrt{Gx^2 + Gy^2}$$

Typically, an approximate magnitude can be obtained by using: $|G| = |Gx| + |Gy|$, which is much faster to compute.

In the actual implementation on SIPS, it is faster to compute four pixels at a time in 8-bit grey level mode. Table 8-13 gives an overview of the pixel alignments in 8-bit grey level mode.

Table 8-13 Pixel alignments in 8-bit grey level mode

column / row	i-1				i				i+1			
j-1	P[j-1,i-1] >>0	P[j-1,i-1] >>8	P[j-1,i-1] >>16	P[j-1,i-1] >>24	P[j-1,i] >>0	P[j-1,i] >>8	P[j-1,i] >>16	P[j-1,i] >>24	P[j-1,i+1] >>0	P[j-1,i+1] >>8	P[j-1,i+1] >>16	P[j-1,i+1] >>24
j	P[j,i-1] >>0	P[j,i-1] >>8	P[j,i-1] >>16	P[j,i-1] >>24	P[j,i] >>0	P[j,i] >>8	P[j,i] >>16	P[j,i] >>24	P[j,i+1] >>0	P[j,i+1] >>8	P[j,i+1] >>16	P[j,i+1] >>24
j+1	P[j+1,i-1] >>0	P[j+1,i-1] >>8	P[j+1,i-1] >>16	P[j+1,i-1] >>24	P[j+1,i] >>0	P[j+1,i] >>8	P[j+1,i] >>16	P[j+1,i] >>24	P[j+1,i+1] >>0	P[j+1,i+1] >>8	P[j+1,i+1] >>16	P[j+1,i+1] >>24

It can be seen from the above table to convolute four pixels contained in data *P[j,i]* with 3x3 convolution kernels it requires the other adjacent fourteen pixels which are stored in the data at column *i-1*, *i* and *i+1* in line *j-1*, *j* and *j+1* separately. As discussed in section 8.2.2, by using temporary variables the efficiency can be increased. An example of doing this with Sobel operation applied is shown below.

```
for (j=top_boundary;j<bottom_boundary;j=j+1)        {
        for (i=left_boundary;i<right_boundary;i=i+1) { // 4 pixels per 32 bit word
                data[0] = ProcessingData[( j - 1 )*1024 + i - 1];
                data[1] = ProcessingData[( j - 1 )*1024 + i    ];
                data[2] = ProcessingData[( j - 1 )*1024 + i + 1];
                data[3] = ProcessingData[( j     )*1024 + i - 1 ];
                data[4] = ProcessingData[( j     )*1024 + i     ];
                data[5] = ProcessingData[( j     )*1024 + i + 1];
                data[6] = ProcessingData[( j + 1 )*1024 + i - 1];
                data[7] = ProcessingData[( j + 1 )*1024 + i    ];
                data[8] = ProcessingData[( j + 1 )*1024 + i + 1];
            sumr[0] = abs(abs (
                ( ( ( data[1] >> 8 ) & 0xff ) - ( ( data[0] >> 24 ) & 0xff ) ) +
```

```
            2*( ( ( data[4] >> 8 ) & 0xff ) - ( ( data[3] >> 24 ) & 0xff ) ) +
            ( ( ( data[7] >> 8 ) & 0xff ) - ( ( data[6] >> 24 ) & 0xff ) ) ) +
            abs (
            ( ( (data[0] >> 24) & 0xff ) + 2 * ( ( data[1] >> 0) & 0xff ) + ( ( data[1] >> 8) & 0xff ) ) -
            ( ( (data[6] >> 24) & 0xff ) + 2 * ( ( data[7] >> 0) & 0xff ) + ( ( data[7] >> 8) & 0xff ) )
            ));
        sumr[1] =.......
        sumr[2] =.......
        sumr[3] =.......
    ProcessedData[1024*(j)+i] = ((( 60 <  sumr[0]/4)? 255 : 0) & 0xff) | ((( 60 < sumr[1]/4)? 255 :
    0) & 0xff)<<8 | (( 60 <  sumr[2]/4)? 255 : 0) & 0xff)<<16  | ((( 60 < sumr[3]/4)? 255 : 0) &
    0xff)<<24;
    }
}
```

In the above example, only the calculation for the first pixel in data *P[j,i]* is given. The others just simply follow the same calculation rules.

In the Sobel operation, the output values can easily overflow the maximum allowed pixel value. In order to avoid this, the input can be divided by a normalising factor to ensure the output values stay in the permitted range, in 8-bit grey level mode, this range is 0~255. Furthermore, by applying post thresholding it could further highlight the edges. A normalising factor of 4 and a threshold of 60 were applied into the test result images Figure 8-28 and Figure 8-29.

Table 8-14 shows a performance analysis for the sample program given above for the Sobel edge detector.

Table 8-14 Sobel edge detector processing power analysis

Processing window size	Number of Timer ticks (average)	Processing time (ms)	Processing power (f/s)
80x60	515730	12.893	77.559
160x120	2063505	51.587	19.384
320x240	8253571	206.339	4.846
640x480	33011145	825.278	1.211

As seen from the table the processing power of the Sobel operation is much slower than the inversion because it involves with two 3x3 convolutions.

8.2.4.3. Gaussian blur filter (a low pass filter)

The Gaussian blur filter, as its name indicates, is used to 'blur' images and removes detail and noise. It is in fact a low pass filter because high spatial frequency components from an image are removed. Figure 8-31 and Figure 8-32 show two test result images with the Gaussian blur filter applied. Only the right half of these two images was processed.

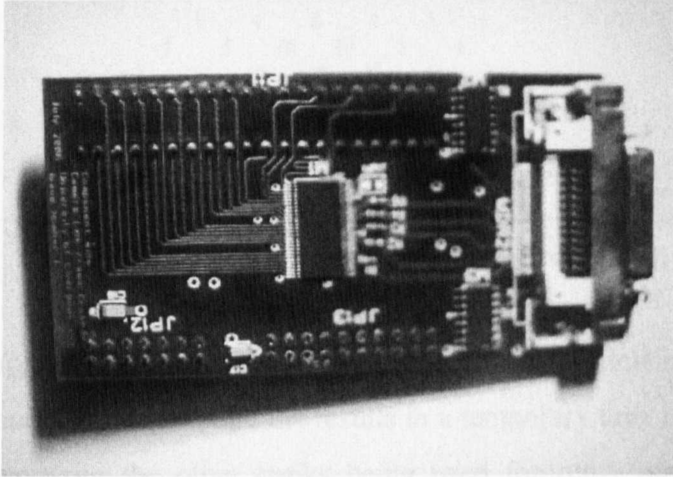


Figure 8-31 Gaussian blur filter image – camera interface card (mask size 7x7)

Figure 8-32 Gaussian blur filter image – mugs and jug (mask size 7x7)

Practically the Gaussian filter operates a two-dimension convolution kernel to represent the shape of a Gaussian distribution which has the form of $G(x,y)=\frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}$.

Because the Gaussian is isotropic which is circularly symmetric, the convolution kernel can be separated into horizontal and vertical component. Thus the two-dimension convolution can be performed by first convolving with a one-dimension Gaussian in

the horizontal direction, and then convolving with another one-dimension Gaussian in the vertical direction. Figure 8-33 lists some possible convolution kernel coefficients for one-dimension Gaussian in horizontal direction. . The vertical component is exactly the same but is orientated vertically.

Index N	Coefficients	Sum of coefficients = 2^N
0	1	1
1	1 1	2
2	1 2 1	4
3	1 3 3 1	8
4	1 4 6 4 1	16
5	1 5 10 10 5 1	32
6	1 6 15 20 15 6 1	64
7	1 7 21 35 35 21 7 1	128
8	1 8 28 56 70 56 35 8 1	256
9	1 9 36 84 126 126 84 36 9 1	512
10	1 10 45 120 210 252 210 120 45 10 1	1024
11	1 11 55 165 330 462 462 330 165 55 11 1	2048

Figure 8-33 Gaussian filters coefficients (from [104])

In the actual implementation on SIPS, it would be more efficient to calculate the horizontal components first and store the results in a temporary area in the SDRAM (be careful not to overwrite the other banks being used for other purposes), and then calculate the vertical components by using the data which have been stored in the temporary area in the SDRAM. Finally the results are saved into the processed bank in the SDRAM where it will be displayed in the next round. An example of software of implementing this is given below. In this example, a 7x7 convolution kernel was used.

```
// This loop is to calculate the horizontal components
for (j=top_boundary;j<bottom_boundary;j=j+1)      {
        for (i=left_boundary;i<right_boundary;i=i+1) { // 4 pixels per 32 bit word
          data[0] = ProcessingData[( j    )*1024 + i - 1 ];
          data[1] = ProcessingData[( j    )*1024 + i    ];
          data[2] = ProcessingData[( j    )*1024 + i + 1 ];
          sumr[0] = ( ( data[0] >> 8  & 0xff ) + 6  * ( data[0] >> 16 & 0xff ) + 15 * ( data[0] >> 24 &
                  0xff ) + 20 * ( data[1] >> 0  & 0xff ) + 15 * ( data[1] >> 8  & 0xff ) + 6  * ( data[1] >>
                  16 & 0xff ) + ( data[1] >> 24 & 0xff )) / 64;
          sumr[1] = ......
          sumr[2] = ......
          sumr[3] = ......
          Temp[1024*(j)+i] = (sumr[0] & 0xff) | (sumr[1] & 0xff)<<8 | (sumr[2] & 0xff)<<16 |
          (sumr[3] & 0xff)<<24;
        }
}
```

```
// This loop is to calculate the vertical components
for (j=top_boundary;j<bottom_boundary;j=j+1)        {
        for (i=left_boundary;i<right_boundary;i=i+1) { // 4 pixels per 32 bit word
          data[0] = Temp[( j - 3 )*1024 + i ];
          data[1] = Temp[( j - 2 )*1024 + i ];
          data[2] = Temp[( j - 1 )*1024 + i ];
          data[3] = Temp[( j    )*1024 + i ];
          data[4] = Temp[( j + 1 )*1024 + i ];
          data[5] = Temp[( j + 2 )*1024 + i ];
          data[6] = Temp[( j + 3 )*1024 + i ];
          sumr[0] = ( ( data[0] >> 0  & 0xff ) + 6 * ( data[1] >> 0  & 0xff ) + 15 * ( data[2] >> 0  &
                 0xff ) + 20 * ( data[3] >> 0  & 0xff ) + 15 * ( data[4] >> 0  & 0xff ) + 6  * ( data[5]
                 >> 0  & 0xff ) + ( data[6] >> 0  & 0xff ) ) / 64;
          sumr[1] = ......
          sumr[2] = ......
          sumr[3] = ......
          ProcessedData[1024*(j)+i] = (sumr[0] & 0xff) | (sumr[1] & 0xff)<<8 | (sumr[2] & 0xff)<<16
                 | (sumr[3] & 0xff)<<24;
        }
}
```

Again in this example four pixels are calculated together within one loop cycle. Details of how the actual pixel number matches the SDRAM data can be found in Table 8-13.

Table 8-15 shows a performance analysis for the sample program given above for the Gaussian filter with a 7x7 convolution kernel applied.

Table 8-15 Gaussian blur filter processing power analysis

Processing window size	Number of Timer ticks (average)	Processing time (ms)	Processing power (f/s)
80x60	864104	21.602	46.290
160x120	3439819	85.995	11.628
320x240	13688085	342.202	2.922
640x480	54599030	1364.975	0.732

As seen from the table the processing power of this Gaussian filter is slow, even less than one frame/second when operated in full frame size. It is due to its large convolution mask. Practically the larger the convolution mask is, the smoother effect it can get, and consequently the more processing time it takes.

8.2.4.4. Sharpness filter – (a high pass filter)

A sharpness filter is used to sharpening an image by making the detailed parts such as edges and lines more precise. As the detailed parts of an image correspond to the high spatial frequency components, the sharpness filter is actually a high pass filter. Figure 8-34 and Figure 8-35 show the sharpening effect on the two sample images. Only the right half part of the two images was processed with the sharpness filter.

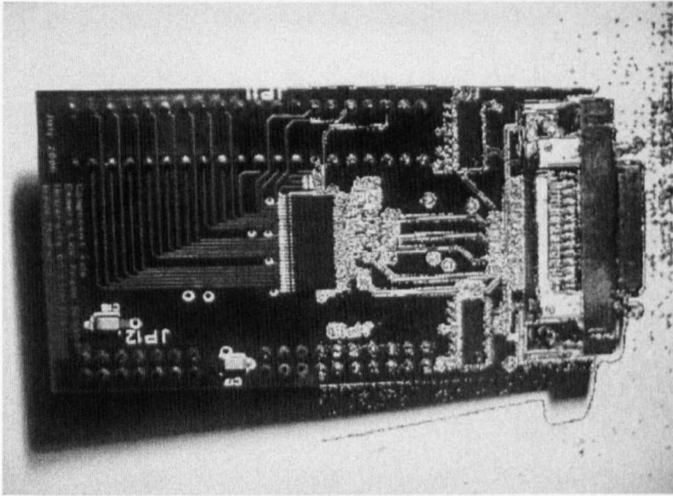


Figure 8-34 Sharpness filter image –camera interface card

Figure 8-35 Sharpness filter image –camera interface card

In the actual implementation, the sharpness filter is slightly simpler than the Sobel and Gaussian operator because it only contains one convolution mask and its normalising factor is 1. The convolution kernel of it is shown in Table 8-16.

Table 8-16 Sharpness filter convolution kernel

0	-1	0
-1	5	-1
0	-1	0

An example of software of implementing the sharpness filter is given below.

```
for (j=top_boundary;j<bottom_boundary;j=j+1)        {
        for (i=left_boundary;i<right_boundary;i=i+1) { // 4 pixels per 32 bit word
           data[1] = ProcessingData[( j - 1 )*1024 + i     ];
           data[3] = ProcessingData[( j      )*1024 + i - 1 ];
           data[4] = ProcessingData[( j      )*1024 + i     ];
           data[5] = ProcessingData[( j      )*1024 + i + 1 ];
           data[7] = ProcessingData[( j + 1 )*1024 + i     ];
           sumr[0] = ( 5 * ( data[4] >> 0  & 0xff ) - ( data[1] >> 0  & 0xff ) - ( data[3] >> 24 & 0xff ) -
                        ( data[4] >> 8  & 0xff ) - ( data[7] >> 0  & 0xff ) );
           sumr[1] = ( 5 * ( data[4] >> 8  & 0xff ) - ( data[1] >> 8 & 0xff ) - ( data[4] >> 0  & 0xff ) -
                        ( data[4] >> 16  & 0xff ) - ( data[7] >> 8  & 0xff ) );
           sumr[2] = ( 5 * ( data[4] >> 16  & 0xff ) - ( data[1] >> 16 & 0xff ) - ( data[4] >> 8  & 0xff ) -
                        ( data[4] >> 24  & 0xff ) - ( data[7] >> 16  & 0xff ) );
           sumr[3] = ( 5 * ( data[4] >> 24  & 0xff ) - ( data[1] >> 24  & 0xff ) - ( data[4] >> 16  & 0xff )
                        - ( data[5] >> 0   & 0xff ) - ( data[7] >> 24  & 0xff ) );
           ProcessedData[1024*(j)+i] = (sumr[0] & 0xff) | (sumr[1] & 0xff)<<8 | (sumr[2] & 0xff)<<16
                        | (sumr[3] & 0xff)<<24;
        }
}
```

Table 8-17 shows a performance analysis for the sample program given above for the Sharpness filter.

Table 8-17 Sharpness filter processing power analysis

Processing window size	Number of Timer ticks (average)	Processing time (ms)	Processing power (f/s)
80x60	258117	6.452	154.968
160x120	1045220	26.130	38.269
320x240	4184450	104.611	9.559
640x480	16741720	418.543	2.389

As seen from the table, the processing time of the sharpness filter is approximately twice faster than the Sobel detector. It is because they both use standard convolution method however the Sobel detector has approximately twice the data to calculate than the Sharpness filter.

8.2.4.5. Feature correlation

The feature correlation algorithm implementation on SIPS is exactly the same as what was done on the PC-based IPS. However, because there is no full application on SIPS, the extraction of the feature arrays on the calibration stage was very simple by just running another program to print out the data in the feature arrays out, and then manually copy those data into the feature arrays used by the actual image processing program. Figure 8-36 shows a welding image after being processed by the feature correlation algorithm.

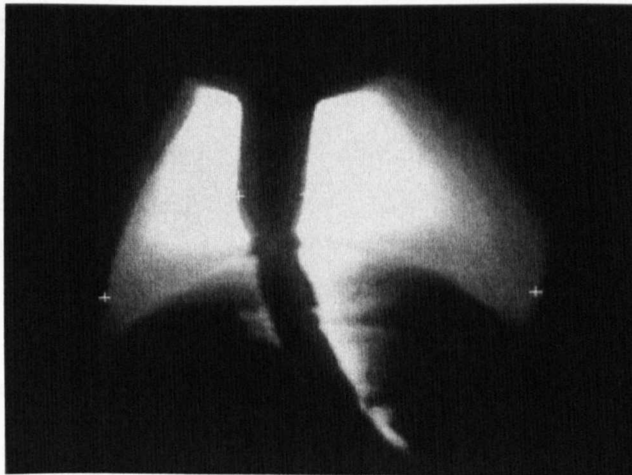


Figure 8-36 Feature correlation image (measured on pool edges & wire edges)

This example was trying to find the welding pool edges and the welding wire edges. Each of them uses two feature arrays including the left and the right edge. The welding pool feature array size is 60 pixels while the length of image intensity line where the program looks for errors is 600 pixels from 40 to 640. The welding wire feature array size is 20 and the length of image intensity line is 360 pixels from 40 to 400. In total there are two intensity lines were used in this example. Furthermore, as the processed image is almost the same as the raw image except for the crosses, Triple bank operation is used for this algorithm. The following program gives an example of implementing feature correlation on SIPS. This example only shows how to find the welding wire right edge match errors and draw the cross on the right edge of the wire.

```
// calculate the error array
for (index = WireCalibrationPosition_x; index <= WirelineRightBoundary - WireFeatureArraySize;index++) {
    errarray[index- WireCalibrationPosition_x]=0;
    for (i=0;i<=WireFeatureArraySize;i=i+4) {
        errarray[index- WireCalibrationPosition_x] = errarray[index - WireCalibrationPosition_x] +
            abs( ( 0xff & WireRightFeatureArray[i] ) - ( 0xff & ProcessingData
            [(WireCalibrationPosition_y) * 1024 + (index + i) / 4] ) ) + abs( ( 0xff &
            WireRightFeatureArray[ I + 1] ) - ( 0xff & ProcessingData[(WireCalibrationPosition_y) *
            1024 + (index + i) / 4] >> 8 ) ) + abs( ( 0xff & WireRightFeatureArray[ i + 2] ) - ( 0xff &
            ProcessingData[(WireCalibrationPosition_y) * 1024 + (index + i) / 4] >> 16 ) ) +
            abs( ( 0xff & WireRightFeatureArray[i+3] ) - ( 0xff &
            ProcessingData[(WireCalibrationPosition_y) * 1024 + (index + i) / 4] >> 24 ) );
    }
}
rightcrosspos = 0;errarray_min = errarray[0];
// find where the minimum error is
for (index = 0;index <= WirelineRightBoundary - WireCalibrationPosition_x - WireFeatureArraySize;
index++) {
    if (errarray_min > errarray[index]) {
        errarray_min = errarray[index];
        rightcrosspos = index;
    }
}
rightcrosspos = rightcrosspos + WireCalibrationPosition_x+WireFeatureArraySize/2;
// draw the cross in the horizontal direction
for (i=(rightcrosspos)/4-1;i<(rightcrosspos)/4+2;i=i+1) {
    ProcessedData[WireCalibrationPosition_y*1024+i] = 0xffffffff;
}
//draw the cross in the vertical direction
for (j=WireCalibrationPosition_y-5;j<WireCalibrationPosition_y+5;j=j+1){
    ProcessedData[j*1024+rightcrosspos /4] = ProcessedData [j*1024+rightcrosspos /4] | ( 0xff <<
    (8*(rightcrosspos%4)));
}
```

As the feature correlation algorithm has many parameters which could affect the processing such as the size of the feature arrays and the length of the image intensity lines, detailed processing power analysis is not given for this algorithm. Table 8-18 only lists the processing power for the example given above.

Table 8-18 Feature correlation processing power analysis

Processing details	Number of Timer ticks (average)	Processing time (ms)	Processing power (f/s)
Two sets of feature arrays with size of 20 and 60 pixels, two image intensity lines with size of 600 and 360 pixels	860527	21.513	46.48

8.3 Summary

This chapter firstly presented all tests which have been undertaken for SIPS both in simulation and hardware verification to prove the system works properly. Then a general software development guideline was given to correctly setup SIPS and test functions, followed by a discussion of coding style which leads to the optimisation issue. Finally it described how five image processing algorithms were implemented on SIPS. Test results, example program and performance analysis were given for each of the algorithm implementation.

As the 24-bit RGB SIPS could not be evaluated on the current development platform, only the software development for the 8-bit grey-level mode presented in this thesis. However, sufficient tests have been done to ensure the system can operate properly in 24-bit mode. Compared to the coding efficiency in 8-bit mode, the 24-bit RGB mode would be more efficient because a 32-bit data contains exactly one image pixel.

Chapter 9. Conclusions and Future Work

9.1 Conclusions

Vision based systems have been used in a wide range of applications. More and more applicants demand such systems with a compact size so that they can be placed close to applicants whilst offer high performance and the potential to be upgraded or optimised in future. Furthermore, commercial investors also urge to develop such a system quicker and more cost effective. Conversional desktop or embedded microprocessors based vision systems become difficult to meet these demands as they are either ponderous, bulky, timing consuming to develop, lack flexibility or are difficult to upgrade and optimise. This research, therefore investigated an alternative solution by utilising the modern technology – System-on-a-Programmable-Chip to implement this kind of systems. SIPS, which is the successfully developed single-chip vision based system with the abilities of performing stand-alone real-time video acquisition, processing and display, has demonstrated some distinct properties as follows:

- Reconfigurability - SIPS can be reconfigured to meet different system requirements. For example, the entire system can be re-configured to work either in 8-bit monochrome mode or 24-bit RGB mode. Furthermore, the reconfigurability of SIPS makes it suitable for future development either in optimisation or for specific applications with various performance requirements.

- Programmability - SIPS can be programmed to implement various image processing algorithms. Different video timing can also be achieved by programming the timing registers to allow the system to interface different types of off-chip peripherals and work at different modes.

- Flexibility - There is no restricted requirement for which type of programmable device, processor core or some off-chip peripherals such as the display and memory device that must be chosen to implement the system. System migration to other types of FPGAs is easy because the system core was designed to be re-usable and re-configurable.

- Processing speed – Table 9-1 shows the performance information of some image processing algorithms running on SIPS. As seen from the table various processing speed up to 90 million pixels per second can be achieved on SIPS.

Table 9-1 SIPS performance summary

Operation	Performance
Inversion	90 Megapixels /s
Sobel edge detection (3x3)	3.7 Megapixels /s
Gaussian filter (7x7)	3.7 Megapixels /s
Sharpness filter (3x3)	2.2 Megapixels / s

- Single-chip integration - One of the major advantages of this image processing system compared with the existing PC-based vision system is its compactness; just imagine a tiny size chip can do the equivalent work of a PC associated with a graphic card and a video capture card. Furthermore, due to the high integration of the system on the chip, the end application board design is simpler so that it can be smaller and placed closer to the applicant.

- Power dissipation - SIPS consumes low power. Normally off-chip devices consume most of the power such as a cooling fan, huge hard drive and other PCI cards etc. However the high integration of SIPS results in lower load of off-chip peripherals so the power dissipation is low. Probably a battery supply would be used for the end application.

- Cost - The price of FPGA tends to be getting lower. The FPGA that SIPS was evaluated on is almost the lowest level and that means SIPS can be implemented on almost any other FPGAs, which provides the designers a wider selection in terms of cost and performance. Further cost can be saved by using cheap video memory and display device.

The SIPS system core consists of a Nios soft processor core, a video capture controller core, a video display controller core, a memory controller core, a Cache core and other vendor provided IP blocks for extra support. These IP components were integrated with the dedicated bus system – Avalon bus, and evaluated on an Altera's Apex

20K200E484-2x FPGA to perform general processing and control all off-chip peripherals including a VGA mode CRT monitor, a CameraLink CMOS camera, an SDRAM memory device, two SRAM chips and a Flash memory. This whole system was developed based on the Nios development kit.

In order to increase the data processing efficiency and maximise the system performance, a few techniques have been applied into the system design such as using multiple master/slave pairs to increase the data bandwidth, implementing Avalon streaming transfers in all video data transfer paths to increase the data transfer rate and clocking the streaming data transfer with a fast clock. Furthermore, in order to solve the conflict accessing to the common memory slave problem an effective solution of simultaneous multi-mastering Avalon streaming transfer with peripheral-controlled waitrequest was raised. This solution also maximises the image processing power by eliminating the need for the CPU to arbitrate and synchronise all accesses to the shared memory.

All of this work on SIPS has proved the challenges for this research. The successful integration of the full image processing system on a programmable chip and the effective bus mastering scheme has demonstrated the novelty of this research. The development of the main SIPS core from the initial VHDL coding, the construction of the custom PCB board from the original schematics and the successful implementation of the system on the low-level FPGA have approved very rewarding. With the fast evolvement of modern technologies, there are always concerns to design a system by using very good equitments, the techniques used in the SIPS design are scalable and can be applied into developing such a system with later generation FPGAs or faster cameras and so on to achieve higher speed and greater processing power.

9.2 Future work

This image processing system has a lot of potentials to be optimised / improved and developed into a full application for vision-based system.

9.2.1 Optimisation/improvements

There are two major directions of optimisation/improvements regarding what can be

optimised / improved with a later generation FPGA and what can be optimised for a specific application.

9.2.1.1. Later generation FPGA

Further work can be given for SIPS by implementing it on a later generation FPGA to find out how much the performance improvement can be and to test some functions which the original device couldn't support. Some suggested optimisation/improvements are given as follows.

a) evaluate SIPS in 24-bit mode
b) implement a CPU cache
c) generate all clocks from the same PLL to reduce the latencies caused by re-synchronising different clock domains
d) test the system operated at the original designed frequencies
e) implement a more advanced Cache which supports more Cache lines
f) reduce the number of pipelining and use asynchronous memory interface

9.2.1.2. FPGA Hardware processing

The initial SIPS was developed without any specific application but to prove a concept of using SOPC solution to implement a vision based system, thus software approach was used to perform general processing. However, for specific applications, special investigation can be given to find out the best balance between hardware processing and software processing for specific algorithms. Generally hardware processing is used in low-level matrix-based image processing such as filtering while software processing is used in high-level processing such as feature measurement. Hardware processing can be achieved by implementing custom instructions for the soft processor core, or dedicated DSP blocks, or a processor array which consists of a number of processing elements and supports parallel processing by implementing instructions like single instruction multiple data (SIMD). Furthermore, if resource is available, it is possible to investigate the integration of multiple soft processor cores on the same chip.

9.2.2 Application

Further work can be undertaken on SIPS by developing it into a specific application for industrial interest such as the vision-based closed loop process control system for

welding. For example, by integrating some network controllers such as a synthesised CAN core [105] or an Ethernet controller core SIPS will be allowed to transmit the real-time measurements like what the conventional PC based system did to a remote computer. Moreover, a higher level software program can also be developed to allow the system to be easily controlled, for instance the calibration stage before running the feature correlation algorithm. Finally, prototyping the application board might be needed for the end product.

References

[1]. Muramatsu, S, Otsuka, Y, Takenaga, H, Kobayashi, Y, Furusawa, I, Monji, T, "Image processing device for automotive vision systems", Intelligent Vehicle Symposium, 2002 IEEE, 17-21 June 2002, Volume 1, pp 121- 126

[2]. J. Kang, R. Doraiswami, "Real-time Image Processing System for Endoscopic Applications", Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canada, -7 May 2003, Volume 3, pp 1469 - 1472

[3]. C. Balfour, J. S. Smith and S. Amin-Nejad, "Feature correlation for weld image-processing applications", International Journal of Production Research, March 2004, Volume 42, pp 975-995

[4]. Arnold, J.M, Buell, D.A, Hoang, D.T, Pryor, D.V, Shirazi, N, Thistle, M.R, "The Splash 2 processor and applications", Computer Design: VLSI in Computers and Processors, 1993. ICCD '93. Proceedings 1993 IEEE International Conference, 3-6 Oct 1993, pp 482-485

[5]. Ikenaga, T., Ogura, T., "A DTCNN universal machine based on highly parallel 2-D cellular automata CAM^2", Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on, May 1998, Volume 45 Issue 5, pp 538-546

[6]. Sandler, M.B, Hayat, L, Costa, L, Naqvi, A, "A comparative evaluation of DSPs, microprocessors and the transputer for image processing", Acoustics, Speech, and Signal Processing, 1989. ICASSP-89., 1989 International Conference on, 23-26 May 1989, Volume 3 pp 1532-1535

[7]. Raphaël Canals, Anthony Roussel, Jean-Luc Famechon, and Sylvie Treuillet, "A Biprocessor-Oriented Vision-Based Target Tracking System", Industrial Electronics, IEEE Transaction on, April 2002, Volume. 49, Issue 2, pp 500-506

[8]. Palacin, J., Sanuy, A., Clua, X., Chapinal, G., Bota, S., Moreno, M.,Herms, A., "Autonomous mobile mini-robot with embedded CMOS vision system", IECON 02 [Industrial Electronics Society, IEEE 2002 28th Annual Conference of the], 5-8 Nov. 2002, Volume 3, pp 2439 - 2444

[9]. Jamro, E, Wiatr, K, Inst. of Electron., AGH Tech. Univ. of Cracow, "Implementation of convolution operation on general purpose processors", Euromicro Conference, 2001. Proceedings 27th, 2001, pp 410-417

[10]. Anthony Rowe, Charles Rosenberg, Illah Nourbakhsh, "A Second Generation Low Cost Embedded Color Vision System", Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), June 2005, Volume 3 pp 136-136

[11]. Karen Parnell, Roger Bryner, "Comparing and Contrasting FPGA and Microprocessor System Design and Development", WP213 (v1.1), Xilinx, Inc., 21st July 2004, http://www.xilinx.com

[12]. Altera Corporation, http://www.altera.com

[13]. André DeHon, "Comparing Computing Machines", 3rd November 1998, Configurable Computing: Technology and Applications, Proc. SPIE 3526

[14]. Xilinx, Inc., www.xilinx.com

[15]. Rose, J, "Hard vs. soft: the central question of pre-fabricated silicon", Multiple-Valued Logic, 2004. Proceedings. 34th International Symposium on, 19-22 May 2004, pp 2-5

[16]. Pat Mead , "Systems on Programmable Chips –Will SOPC Eclipse SoC?", Designing Systems on Silicon IEE Cambridge Seminar, December 2001,

[17]. Azriel Rosenfeld, "Computer Vision: Basic Principles", Proceedings of the IEEE, August 1988, Volume. 76, Issue 8, pp 863-868

[18]. CHAMBERS, Simon Paul, "TIPS: a transputer based real-time vision system", PhD. Thesis, Liverpool University 1990

[19]. Euresys s.a., http://www.euresys.com

[20]. Fan Wu, "Front view image processing program manual", University of Liverpool, April 2004

[21]. Hauppauge Computer Works, Inc., http://www.hauppauge.com

[22]. Beck Zaratian, "Microsoft Visual C++ 6.0 Programmer's Guide", September 4, 1998, Microsoft Press

[23]. Ben Ezzell with Jim Blaney, "Windows 95 Developer's Handbook", February 1997, Sybex Inc.

[24]. Bosch, "CAN specification" Version 2.0, Robert Bosch GmbH, 1991

[25]. Udaya Kamath, Rajita Kaundin, "System-on-Chip Designs Strategy for Success", White paper, Wipro Technologies, June 2001,

[26]. Ravi Krishnan, BCC, "System-on-Chip: Technology and Markets", November 2004, Electronics.ca Publications

[27]. Altera Corporation, "APEX 20K Programmable Logic Device Family" data sheet, Version 5.1, March 2004, http://www.altera.com

[28]. Greg Martin, "Platform ASICs vs. FPGAs", RapidChip Technical Marketing, LSI Logic Corp. September 1, 2005, Online resource http://www.soccentral.com/

[29]. Altera Corporation, "ASIC to FPGA Design Methodology & Guidelines", application note 311, Version 1.0, July 2003, http://www.altera.com

[30]. Abner Barros, Péricles Lima, Juliana Xavier, Manoel E. Lima, "Teaching SoC Design in a Project-Oriented Course based on Robotics", Proceedings of the 2005 IEEE International Conference on Microelectronic Systems Education (MSE'05), 12-14 June 2005, pp 25-26

[31]. Tyson S. Hall and James O. Hamblen, "System-on-a-Programmable-Chip Development Platforms in the Classroom", IEEE Transactions on education, November 2004, Volume. 47, Issue 4, pp. 502-507

[32]. James O. Hamblen, "Using an FPGA-based SOC Approach for Senior Design Projects", Proceedings of the 2003 IEEE International Conference on Microelectronic Systems Education (MSE'03), 1-2 June 2003, pp. 18-19

[33]. Kyeong Keol Ryu, Eung Shin, Mooney, V.J, "A Comparison of Five Different Multiprocessor SoC Bus Architectures", Digital Systems, Design, 2001. Proceedings. Euromicro Symposium on, 2001, pp 202-209

[34]. Keutzer, K., Newton, A.R., Rabaey, J.M., Sangiovanni-Vincentelli, A., "System-Level Design: Orthogonalization of Concerns and Platform-Based

Design", Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions, December 2000, Volume 19, Issue 12, pp 1523-1543

[35]. Altera Corporation, "Avalon Interface Specification Reference Manual", 2004, http://www.altera.com

[36]. ARM Limited, "AMBA specification" Rev 2.0, 13th May 1999, http://www.arm.com

[37]. IBM Corporation, http://www.ibm.com/

[38]. Opencores.org, "Wishbone System-on-chip (SoC) Interconnection Architecture for Portable IP cores Specification", revision B.3, September 2002, http://www.opencores.org

[39]. Wayne Wolf, "A Decade of Hardware/ Software Codesign", IEEE Computer, Apr. 2003, Volume. 36, Issue 4, pp. 38-43

[40]. Finc, M, Zemva, "A. Rapid HW/SW co-design of softcore processor systems", Eurocon 2003. Computer as a Tool. The IEEE Region 8, 22-24 September, 2003, Volume 1, pp 104-108

[41]. Habbi, A. Tahar, S., "A survey on system-on-a-chip design languages", Proceedings of The 3rd IEEE International Workshop on System-on-Chip, July 2003, pp. 212-215

[42]. Altera Corporation, "SOPC Builder Data Sheet", Version 2.0, January 2003, http://www.altera.com

[43]. Altera Corporation, "Using SignalTap II Embedded Logic Analyzers in SOPC Builder Systems", Application note 323, Version 1.0, September 2003, http://www.altera.com

[44]. Altera Corporation, "Stratix Device Family Data Sheet", v3.2, July 2005, http://www.altera.com

[45]. Altera Corporation, "Nios 3.0 CPU" Data sheet, Version 2.2, October 2004, http://www.altera.com

[46]. Xilinx, Inc., "MicroBlaze Processor Reference Guide", UG081 (v7.0), September 2006, http://www.xilinx.com

[47]. Damjan Lampret, "OpenRISC 1200 IP Core Specification", Rev. 0.7, Sep 6, 2001, http://www.opencores.org

[48]. Gaisler Research AB., "SPARC V8 32-bit Processor LEON3 / LEON3-FT CompanionCore Data Sheet", Version 1.0.2, October 2006

[49]. Altera Corporation, "Excalibur Device Overview", Version. 2.0, May 2002 http://www.altera.com

[50]. Xilinx, Inc., "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: complete data sheet", DS083 (v4.6) March 5, 2007, http://www.xilinx.com

[51]. Cypress Semiconductor, http://www.cypress.com

[52]. Atmel Corporation, http://www.atmel.com

[53]. Altera Corporation, "Nios PIO" Data sheet, Version 3.1, January 2003, http://www.altera.com

[54]. Altera Corporation, "Quartus II Handbook", May 2005, http://www.altera.com

[55]. Altera Corporation, "Nios Hardware Development Tutorial", Version 1.2, January 2004, http://www.altera.com

[56]. Altera Corporation, "USB-Blaster Download Cable User Guide", Version 2.3, May 2007, http://www.altera.com

[57]. Altera Corporation, "Nios Software Development Tutorial", Version 1.3, July 2003, http://www.altera.com

[58]. Altera Corporation, "Nios Embedded Processor Development Board" Data sheet, Version 2.2, July 2003, http://www.altera.com

[59]. Altera Corporation, "Nios Embedded Processor Software Development Reference Manual", Version 3.2, March 2003, http://www.altera.com

[60]. Automated Imaging Association, "CameraLink Specifications of the CameraLink Interface Standard for Digital Cameras and Interface cards" Version 1.1, January 2004

[61]. Marco Groeneveld , "Lancelot VGA video controller for the Altera Excalibur processors", 2003, http://www.fpga.nl

[62]. Toshiba, TC59SM716/08/04AFT/AFTL-70,-75,-80 specification, 11th June 2001

[63]. Cohu Inc., Electronics Division, "7800 Series 1280x1024 CMOS progressive scan camera technical reference manual", 2002

[64]. PCB Train, http://www.pcbtrain.co.uk

[65]. 3M company, "3M™ Mini D Ribbon (MDR) Connectors.050'' Surface Mount Right Angle Receptacle - Shielded 102 Series", 16th August 2005

[66]. National Semiconductor, "DS90CR285/DS90CR286 +3.3V Rising Edge Data Strobe LVDS 28-Bit Channel Link-66MHz", July 2004

[67]. National Semiconductor, "DS90C032 LVDS Quad CMOS Differential Line Receiver", September 2003

[68]. National Semiconductor, "DS90C031 LVDS Quad CMOS Differential Line Driver", June 1998

[69]. National Semiconductor, "National Semiconductor Channel Link Design Guide", May 2005.

[70]. John Goldie, National Semiconductor, "Channel-Link PCB and Interconnect Design-In Guidelines", Application Note 1108, August 1998

[71]. The PCI Special Interest Group, "PCI Local Bus Specification", product version, Revision 2.1, June 1995

[72]. Chi-Leung San, Chiu-Sing Choy, Pak-Kee Chan, Cheong-Fat Chan, Kong-Pang Pun, "Realization of Card-Centric Framework: A Card-Centric Computer", IEEE International Symposium on Circuits and Systems (ISCAS), 2005, Volume 5, pp 4999-5002

[73]. Altera Corporation, "Using Excalibur DMA controllers for video imaging", application note 287, http://www.altera.com

[74]. A. Ben Atitallah, P. Kadionik, F. Ghozzi, P. Nouel, N. Masmoudi, Ph. Marchegay, "Hardware Platform Design for Real-Time Video Applications", Microelctronitronics, 2004. ICM 2004 Proceedings. The 16th International Conference on, 6-8 Dec. 2004, pp 722-725

[75]. Burke Henehan (TI) and Michael Johas-Teener (formerly Zayante and Apple, now Broadcom) Michael Scholles (Fraunhofer IPMS), Dave Thompson (Agere

Systems), "1394 Standards and Specifications Summary", 1394 Trade Association, April 2006

[76]. Steve Fielding, "USBHostSlave IP Core Specification", Rev 1.1, October 13 2006, OpenCores. Oranisation

[77]. Compaq, Intel, Microsoft, NEC, "Universal Serial Bus Specification", Revision 1.1, September 23, 1998

[78]. Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips, "Universal Serial Bus Specification", Revision 2.0, April 27, 2000

[79]. Altera Corporation, "Nios Timer" Data sheet, Version 3.2, July 2003, http://www.altera.com

[80]. Altera Corporation, "Nios UART" Data sheet, Version 3.0, January 2003, http://www.altera.com

[81]. Altera Corporation, "SDR SDRAM Controller White Paper", ver1.1, August 2002, http://www.altera.com

[82]. Lattice Semiconductor Corporation, "SDR SDRAM Controller Reference Design RD1010", July 2005, www.latticesemi.com

[83]. David A. Patterson and John L. Hennessy, "Computer Organization & Design, the hardware/software interface", 1998, Morgan Kaufmann Publishers

[84]. Altera Corporation, "Using the ClockLock & ClockBoost PLL Features in APEX devices", application note 115, Version 2.6, November 2003, http://www.altera.com

[85]. William J. Dally and John W. Poulton, "Digital Systems Engineering", June 1998, Cambridge University Press

[86]. Clifford E. Cummings, Sunburst Design, Inc., "Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs", 20th June 2005

[87]. Hang Yang, Hongyi Chen, Guoqiang Bai, "An Improved DMA Controller for High Speed Data Transfer in MPU Based SOC", Solid-state and Integrated Circuits Technology 2004. Porceedings. 7th International Conference on, October 2004, Volume 2, pp 1372-1375

[88]. Altera Corporation, "Nios DMA" data sheet Version 1.2, July 2003, http://www.altera.com

[89]. Sanghun Lee, Chanho Lee, Hyuk-Jae Lee, "A new multi-channel on-chip-bus architecture for system-on-chips", SOC Conference, 2004. Proceedings. IEEE International, 12-15 Sept. 2004, pp 305-308

[90]. Altera Corporation, "Simultaneously Multi-Mastering with the Nios Processor", 2002, http://www.altera.com,

[91]. Austin Hung, "Cache Coherency for Symmetric Multiprocessor Systems on Programmable Chips", A master thesis, University of Waterloo, 2004

[92]. Markus Winter and Gerhard Fettweis, "Interconnection Architecture For System-on-Chip Design Providing Little Overhead, Low Latency and High Throughput", 9th EUTOMICRO Conference on Digital System Design, Aug. 2006

[93]. Altera Corporation, "SOPC Builder PTF File Reference Manual", Version 1.2, December 2003, http://www.altera.com

[94]. Altera Corporation, "Estimating Nios Resource Usage & Performance in Altera Devices, Application note 178", Version 1.0, September 2001, http://www.altera.com

[95]. Patrick Pelgrims , Dries Driessens, Tom Tierens, "Overview Excalibur, Leon, Microblaze, Nios, Openrisc, VIRTEX II PRO", Version 1.1, 2003, http://opencontent.org/openpub/

[96]. Altera Corporation, "Simulating Nios Embedded Processor Designs", application note 189, Version 2.1, February 2003, http://www.altera.com

[97]. Barr, Michael. "Software-Based Memory Testing" Embedded Systems Programming, July 2000, pp. 28-40.

[98]. Motorola Inc, "M68EVB912B32 Evaluation Board User's Manual", February 1997

[99]. Sweetscapre Software, "010 EDITOR Reference Manual", http://www.sweetscape.com

[100]. Ant Goffart, http://www.s-record.com/

[101]. Altera Corporation, “Implementing Interrupt Service Routines in Nios Systems”, application note 284, Version 1.0, January 2003, http://www.altera.com

[102]. Altera Corporation, “Custom Instructions for the Nios Embedded Processor”, Application note 188, Version 1.2, September 2002, http://www.altera.com

[103]. Robert Fisher, Simon Perkins, Ashley Walker and Erik Wolfart, “HYPERMEDIA IMAGE PROCESSING REFERENCE (HIPR2)”, 2003, School of Informatics, University of Edinburgh, http://homepages.inf.ed.ac.uk/rbf/HIPR2

[104]. Waltz, F, John W. V. Miller, “An efficient algorithm for Gaussian blur using finite-state machines”, SPIE Conference on Machine Vision Systems for Inspection and Metrology, 1998

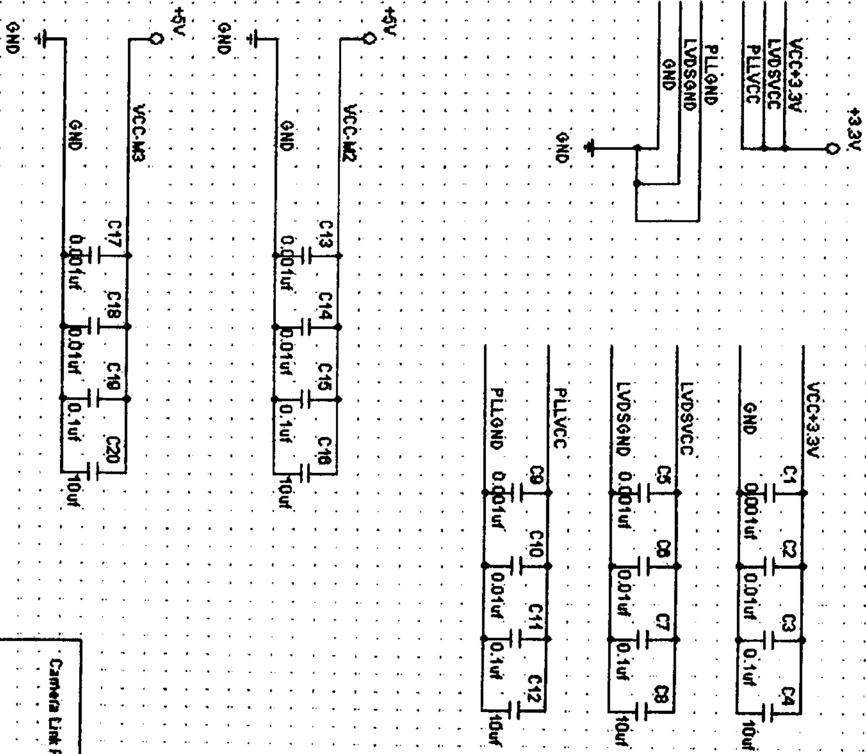
[105]. Fan Wu, “A Synthesised Controller Area Network (CAN) Core”, M.Sc dissertation, September 2003, University of Liverpool

Appendixes

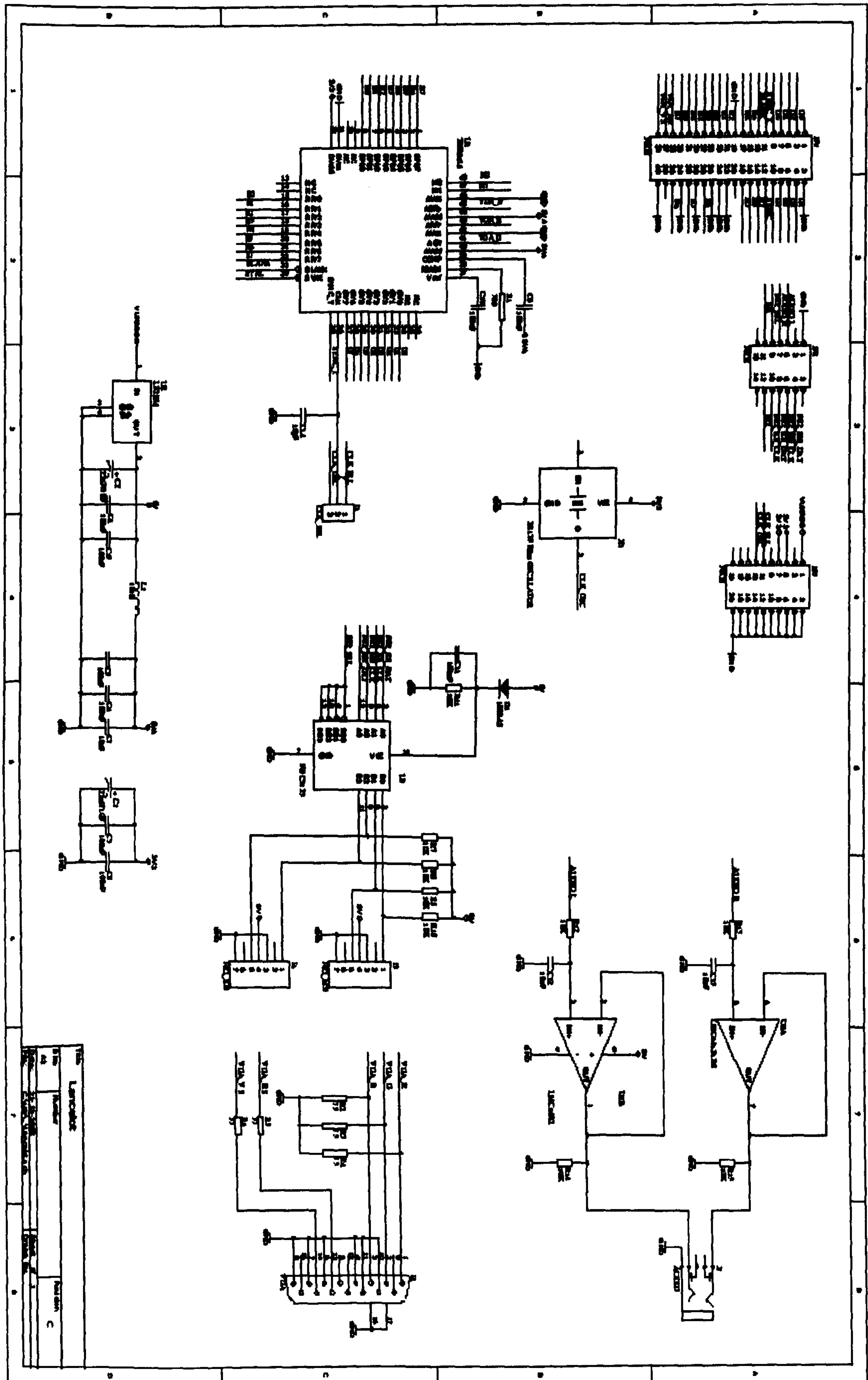
Appendix A Schematics

1. Schematic of the Camera Interface Card 1

2. Schematic of the Camera Interface Card 2

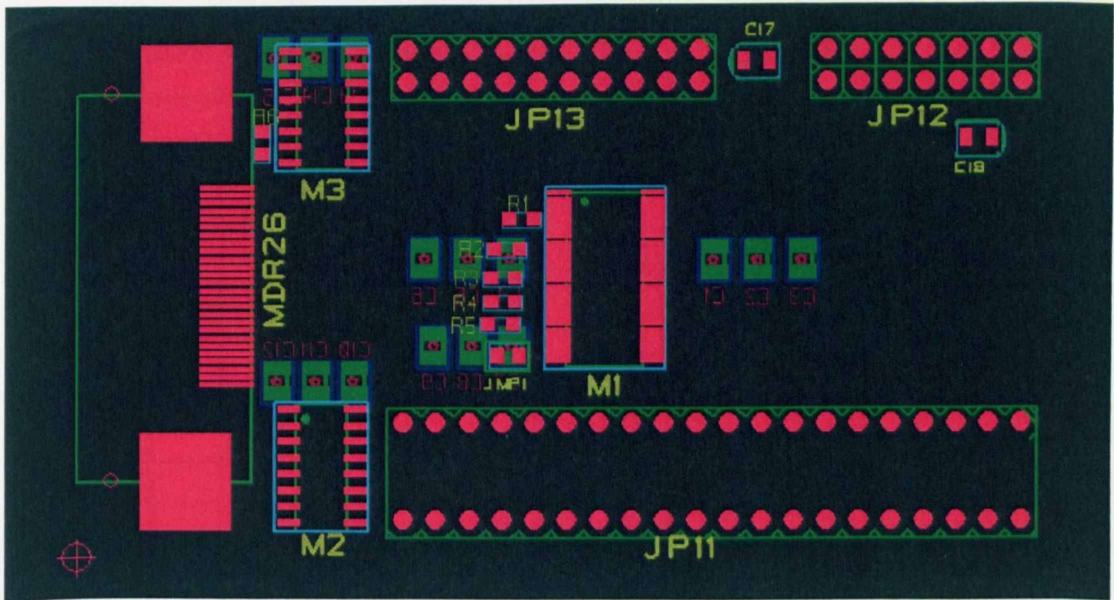


3. Schematic of the Lancelot VGA board

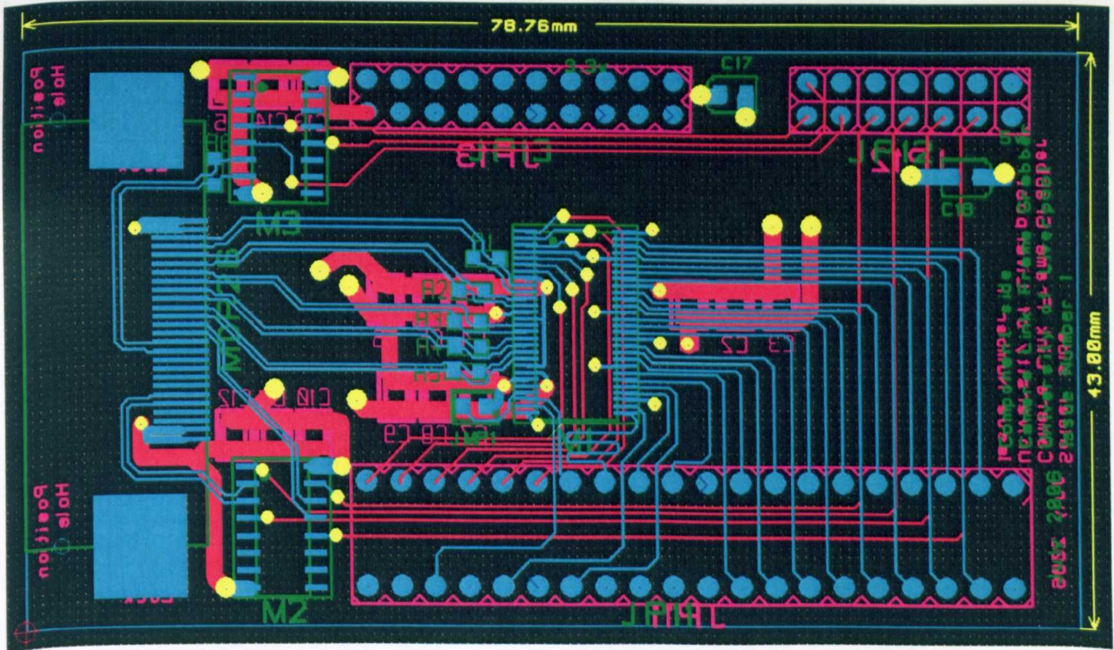


Appendix B Camera interface card PCB details

1. Component side of the CameraLink Interface card PCB



2. CameraLink Interface card PCB with four layers



Appendix C Pin assignments for video components

1. SDRAM device I/O pin mappings

Signal name	Apex 20K		Signal name	Apex 20K
bs[0]	Y14		dq[26]	V2
bs[1]	U13		dq[27]	Y22
ras_n	E16		dq[28]	AA20
we_n	C15		dq[29]	AA19
cs_n	C16		dq[30]	V21
cke	T13		dq[31]	V22
cas_n	R12		dq[32]	AB5
clk0	P5		dq[33]	AA5
add[0]	U16		dq[34]	AA4
add[1]	V16		dq[35]	AB4
add[2]	U15		dq[36]	AB3
add[3]	W16		dq[37]	AB19
add[4]	V15		dq[38]	AB20
add[5]	Y16		dq[39]	AA17
add[6]	W15		dq[40]	AA18
add[7]	T14		dq[41]	AB21
add[8]	Y15		dq[42]	AA8
add[9]	R13		dq[43]	AB7
add[10]	U14		dq[44]	AA7
add[11]	V14		dq[45]	AB6
dq[0]	V19		dq[46]	AA6
dq[1]	U20		dq[47]	AB17
dq[2]	W4		dq[48]	AA14
dq[3]	V4		dq[49]	AA15
dq[4]	W3		dq[50]	AB18
dq[5]	Y3		dq[51]	AA16
dq[6]	V3		dq[52]	AA9
dq[7]	Y19		dq[53]	AB8
dq[8]	R17		dq[54]	AA10
dq[9]	Y20		dq[55]	AA11
dq[10]	T17		dq[56]	AA12
dq[11]	P16		dq[57]	AB15
dq[12]	AA3		dq[58]	AB16
dq[13]	W2		dq[59]	AA13
dq[14]	Y2		dq[60]	Y5
dq[15]	Y4		dq[61]	Y6
dq[16]	W5		dq[62]	T6
dq[17]	W21		dq[63]	P7
dq[18]	W22		dqmb[0]	Y12
dq[19]	Y21		dqmb[1]	T12

dq[20]	W19		dqmb[2]	Y11
dq[21]	V20		dqmb[3]	E17
dq[22]	W1		dqmb[4]	D17
dq[23]	AB2		dqmb[5]	C17
dq[24]	V1		dqmb[6]	H16
dq[25]	Y1		dqmb[7]	F16

2. Lancelot VGA board I/O pin mappings

Signal name	Apex 20K	Lancelot		Signal name	Apex 20K	Lancelot
r[0]	R5	JP1-25		b[0]	N6	JP1-27
r[1]	K1	JP1-23		b[1]	L20	JP1-28
r[2]	P20	JP1-21		b[2]	J18	JP1-29
r[3]	K1	JP1-18		b[3]	M17	JP1-32
r[4]	P21	JP1-17		b[4]	K18	JP1-31
r[5]	N2	JP1-16		b[5]	J3	JP1-33
r[6]	L7	JP1-15		b[6]	R4	JP1-36
r[7]	N5	JP1-14		b[7]	K5	JP1-35
g[0]	N20	JP1-3		hs	J7	JP1-37
g[1]	K20	JP1-4		vs	J5	JP1-39
g[2]	P4	JP1-5		blank_n	V12	JP1-13
g[3]	K4	JP1-6		sync_n	R22	JP1-12
g[4]	V11	JP1-7		sync_t	N22	JP1-11
g[5]	K22	JP1-8		m1	J1	JP2-11
g[6]	K19	JP1-9		mm2	J12	JP2-12
g[7]	P22	JP1-10				

3. Camera interface card I/O pin mappings

Signal name	Apex 20K	Camera interface card
capture_clk	R19	J11-RXCLKOUT
LVal	U3	J11-RXOUT24
FVal	T1	J11-RXOUT25
DVal	R2	J11-RXOUT26
Trig	P17	J12-Trig
video_data[0]	R1	J11-RXOUT0
video_data[1]	K2	J11-RXOUT1
video_data[2]	P18	J11-RXOUT2
video_data[3]	N3	J11-RXOUT3
video_data[4]	M15	J11-RXOUT4
video_data[5]	P1	J11-RXOUT6
video_data[6]	P2	J11-RXOUT27
video_data[7]	P3	J11-RXOUT5
video_data[8]	R3	J11-RXOUT7
video_data[9]	T21	J11-RXOUT8

video_data[10]	N17	J11-RXOUT9
video_data[11]	T22	J11-RXOUT12
video_data[12]	K1	J11-RXOUT13
video_data[13]	M2	J11-RXOUT14
video_data[14]	T20	J11-RXOUT10
video_data[15]	L14	J11-RXOUT11
video_data[16]	U5	J11-RXOUT15
video_data[17]	R20	J11-RXOUT18
video_data[18]	N19	J11-RXOUT19
video_data[19]	P19	J11-RXOUT20
video_data[20]	L15	J11-RXOUT21
video_data[21]	N16	J11-RXOUT22
video_data[22]	N1	J11-RXOUT16
video_data[23]	M3	J11-RXOUT17
Vcc	W20	J12-EN*031
Vcc	N15	J12-EN031
Vcc	U19	J12-EN032
Vcc	U4	J12-EN*032
txd_camera_uart	U2	J12-SerTC
rxd_camera_uart	T3	J12-SerTFG

Appendix D Truth table for the operation commands of SDRAM

Command	Device State	CKE_{n-1}	CKE_n	DQM [5]	BS0, BS1	A10	A11, A9~A0	$\overline{CS}$	$\overline{RAS}$	$\overline{CAS}$	$\overline{WE}$
Bank Activate	Idle [3]	H	X	X	V	V	V	L	L	H	H
Bank Precharge	Any	H	X	X	V	L	X	L	L	H	L
Precharge All	Any	H	X	X	X	H	X	L	L	H	L
Write	Active [3]	H	X	X	V	L	V	L	H	L	L
Write with Auto Precharge	Active [3]	H	X	X	V	H	V	L	H	L	L
Read	Active [3]	H	X	X	V	L	V	L	H	L	H
Read with Auto Precharge	Active [3]	H	X	X	V	H	V	L	H	L	H
Mode Register Set	Idle	H	X	X	V	V	V	L	L	L	L
No-Operation	Any	H	X	X	X	X	X	L	H	H	H
Burst stop	Active [4]	H	X	X	X	X	X	L	H	H	L
Device Deselect	Any	H	X	X	X	X	X	H	X	X	X
Auto-Refresh	Idle	H	H	X	X	X	X	L	L	L	H
Self-Refresh Entry	Idle	H	L	X	X	X	X	L	L	L	H
Self-Refresh Exit	Idle (Self Refresh)	L	H	X	X	X	X	H	X	X	X
								L	H	H	X
Clock Suspend Mode Entry	Active	H	L	X	X	X	X	X	X	X	X
Power Down Mode Entry	Idle/Active [6]	H	L	X	X	X	X	H	X	X	X
								L	H	H	X
Clock Suspend Mode Exit	Active	L	H	X	X	X	X	X	X	X	X
Power Down Mode Exit	Any (Power Down)	L	H	X	X	X	X	H	X	X	X
								L	H	H	X
Data Write/Output Enable	Active	H	X	L	X	X	X	X	X	X	X
Data Write/Output Disable	Active	H	X	H	X	X	X	X	X	X	X

Note 1. V = Valid, X = Don't Care, L = Low level, H = High level
2. CKE_n signal is input level when commands are issued.
CKE_{n-1} signal is input level one clock cycle before the commands are issued.
3. These are state designated by the BS0, BS1 signals.
4. Device state is Full Page Burst operation.
5. LDQM, UDQM (TC59SM716AFT/AFTL)
6. Power Down Mode can not entry in the burst cycle.
When this command assert in the burst cycle, device state is clock suspend mode.

Appendix E Register maps

1. Global register map

Base address	End address	Description
0x00000000	0x00000000	Boot monitor rom
0x00000400	0x0000041F	UART (For communication between the host terminal and SIPS)
0x00000420	0x0000042F	Seven segment PIO (used to control the seven segment LEDs on board)
0x00000430	0x00000437	Video memory controller
0x00000440	0x0000045F	Timer
		………………………………..
0x00000470	0x0000047F	Buffer PIO (to controller the onboard buttons)
		………………………………..
0x000004A0	0x000004BF	Video display controller
0X000004C0	0x000004DF	UART debug
0x000004E0	0x000004FF	Video capture controller
		………………………………..
0x00000500	0x0000051F	Camera UART
		………………………………..
0x00040000	0x0007FFFF	External SRAM
0x00100000	0x001FFFFF	External Flash
		………………………………..
0x04000000	0x7FFFFFFF	Cache
		………………………………..

2. Video memory controller register map

Offset	Register	Mode
0	Mode register	W
1	--	--

Mode register

Bit	Name	Description
31-3	--	Reserved
2-0	CAS latency	Defines the CAS latency in the SDRAM mode register

3. Video display controller register map

Offset	Register	Mode
0	Control register	W
1	Status register	R
2	--	--
3	--	--
4	Resolution register	W/R
5	--	--
6	--	--
7	DMA address register	W/R

Control register

Bit	Name	Description
31-4	-	Reserved
3	Set DAC mode	A logic '1' sets the video DAC in RGB mode
2	Start video	Writing '1' to this bit starts the internal state machine
1	Interrupt Enable	Writing '1' to enable the interrupt
0	Reset	The video controller is automatic reset during power-up

Status register

Bit	Name	Description
31-10	--	Reserved
9	New frame	Logic '1' indicates the next video line is the first line of a new frame
8-4	--	Reserved
3	Blank VS	This bit indicates the vertical blanking status

2	Blank HS	This bit indicates the horizontal blanking status
1	VS	The status of the internal *vs* signal
0	HS	The status of the internal *hs* signal

Resolution register

Bit	Description
31-26	Reserved
25-16	Horizontal resolution
15-0	Vertical resolution

DMA address register

Bit	Description
31-0	DMA start address

4. Video capture controller register map

Offset	Register	Mode
0	Control register	W
0	Status register	R
1	Resolution register	R
2	--	--
3	--	--
4	--	--
5	--	--
6	--	--
7	DMA address register	W/R

Control register

Bit	Name	Description
31-3	--	Reserved
2	Start video	Writing '1' to this bit starts the internal state machine
1	Trigger	Write '1' to generate a *Trig* pulse
0	Enable interrupt	Writing '1' to enable interrupt

Status register

Bit	Name	Description
31-2	--	Reserved
1	Frame valid	The status of the internal *FVal* signal
0	Line valid	The status of the internal *LVal* signal

Resolution register

Bit	Description
31-26	Reserved
25-16	Horizontal resolution
15-0	Vertical resolution

DMA address register

Bit	Description
31-0	DMA start address

Appendix F PTF Files – An example of Cache

```
CLASS user_logic_Cache
{
   ASSOCIATED_FILES
   {
      Add_Program = "";
      Edit_Program = "";
      Generator_Program = "mk_user_logic_Cache.pl";
   }
   MODULE_DEFAULTS
   {
      class = "user_logic_Cache";
      class_version = "2.0";
      SYSTEM_BUILDER_INFO
      {
         Instantiate_In_System_Module = "1";
         Is_Enabled = "1";
         Date_Modified = "--unknown--";
      }
      WIZARD_SCRIPT_ARGUMENTS
      {
      }
      SLAVE cpu_slave
      {
         SYSTEM_BUILDER_INFO
         {
            Bus_Type = "avalon";
            Address_Alignment = "native";
            Address_Width = "24";
            Data_Width = "32";
            Has_IRQ = "0";
            Has_Base_Address = "1";
            Read_Wait_States = "peripheral_controlled";
            Write_Wait_States = "peripheral_controlled";
            Setup_Time = "0";
            Hold_Time = "0";
            Is_Memory_Device = "0";
            Uses_Tri_State_Data_Bus = "0";
            Is_Enabled = "1";
         }
         PORT_WIRING
         {

            PORT system_clk
            {
               width = "1";
               direction = "input";
               type = "export";
            }
            PORT reset_n
            {
               width = "1";
               direction = "input";
               type = "reset_n";
            }
            PORT cpu_slave_addr
            {
               width = "24";
               direction = "input";
               type = "address";
            }
            PORT cpu_slave_cs
            {
               width = "1";
               direction = "input";
               type = "chipselect";
            }
```

```
        PORT cpu_slave_rd
        {
           width = "1";
           direction = "input";
           type = "read";
        }
        PORT cpu_slave_wr
        {
           width = "1";
           direction = "input";
           type = "write";
        }
        PORT cpu_slave_writedata
        {
           width = "32";
           direction = "input";
           type = "writedata";
        }
        PORT cpu_slave_readdata
        {
           width = "32";
           direction = "output";
           type = "readdata";
        }
        PORT cpu_slave_waitrequest
        {
           width = "1";
           direction = "output";
           type = "waitrequest";
        }

         }
}
         MASTER video_data_master_upper
         {
                SYSTEM_BUILDER_INFO
                {
                       Bus_Type = "avalon";
                       Address_Width = "27";
                       Max_Address_Width = "32";
                       Data_Width = "32";
                       Is_Enabled = "1";
                       Do_Stream_Reads = "1";
                       Do_Stream_Writes = "1";

                }
                PORT_WIRING
                {
                       PORT data_clk
                       {
                              width = "1";
                              direction = "input";
                              type = "export";
                       }
                       PORT data_master_addr_u
                       {
                              width = "27";
                              direction = "output";
                              type = "address";
                       }
                       PORT data_master_writedata_u
                       {
                              width = "32";
                              direction = "output";
                              type = "writedata";
                       }
                       PORT data_master_readdata_u
                       {
```

```
                        width = "32";
                        direction = "input";
                        type = "readdata";
                }
                PORT data_master_rd_u
                {
                        width = "1";
                        direction = "output";
                        type = "read";
                }
                PORT data_master_wr_u
                {
                        width = "1";
                        direction = "output";
                        type = "write";
                }
                PORT data_master_waitrequest_u
                {
                        width = "1";
                        direction = "input";
                        type = "waitrequest";
                }
                PORT data_master_endofpacket_u
                {
                        width = "1";
                        direction = "input";
                        type = "endofpacket";
                }
        }
}
    MASTER streaming_control_master
    {
        SYSTEM_BUILDER_INFO
        {
                Bus_Type = "avalon";
                Address_Width = "2";
                Max_Address_Width = "32";
                Data_Width = "32";
                Is_Enabled = "1";
                Do_Stream_Reads = "0";
        }
        PORT_WIRING
        {
                PORT streamingctrl_master_addr
                {
                        direction = "output";
                        width = "2";
                        type = "address";
                }
                PORT streamingctrl_master_readdata
                {
                        direction = "input";
                        width = "32";
                        type = "readdata";
                }
                PORT streamingctrl_master_rd
                {
                        direction = "output";
                        width = "1";
                        type = "read";
                }
                PORT streamingctrl_master_waitrequest
                {
                        direction = "input";
                        width = "1";
                        type = "waitrequest";
                }
        }
```

```
        }

        MASTER video_data_master_lower
        {
                SYSTEM_BUILDER_INFO
                {
                        Bus_Type = "avalon";
                        Address_Width = "32";
                        Max_Address_Width = "32";
                        Data_Width = "32";
                        Is_Enabled = "1";
                        Do_Stream_Reads = "1";
                        Do_Stream_Writes = "1";

                }
                PORT_WIRING
                {
                        PORT data_master_addr_l
                {
                                width = "32";
                                direction = "output";
                                type = "address";
                }
                        PORT data_master_writedata_l
                {
                                width = "32";
                                direction = "output";
                                type = "writedata";
                }
                        PORT data_master_readdata_l
                {
                                width = "32";
                                direction = "input";
                                type = "readdata";
                }
                PORT data_master_rd_l
                {
                                width = "1";
                                direction = "output";
                                type = "read";
                }
                PORT data_master_wr_l
                {
                                width = "1";
                                direction = "output";
                                type = "write";
                }
                PORT data_master_waitrequest_l
                {
                                width = "1";
                                direction = "input";
                                type = "waitrequest";
                }
                PORT data_master_endofpacket_l
                {
                                width = "1";
                                direction = "input";
                                type = "endofpacket";
                }
                }
    }
    SIMULATION
    {
       DISPLAY
       {
          SIGNAL a
          {
```

```
   name = "data_clk";
}
SIGNAL b
{
   name = "system_clk";
}
SIGNAL c
{
   name = "reset_n";
}
SIGNAL d
{
   name = "cpu_slave_addr";
   radix = "hexadecimal";
}
SIGNAL e
{
   name = "cpu_slave_cs";
}
SIGNAL f
{
   name = "cpu_slave_rd";
}
SIGNAL g
{
   name = "cpu_slave_wr";
}
SIGNAL h
{
   name = "cpu_slave_writedata";
   radix = "hexadecimal";
}
SIGNAL i
{
   name = "cpu_slave_readdata";
   radix = "hexadecimal";
}
SIGNAL j
{
   name = "cpu_slave_waitrequest";
}
SIGNAL k
{
   name = "data_master_addr_u";
   radix = "hexadecimal";
}
SIGNAL l
{
   name = "data_master_rd_u";
}
SIGNAL m
{
   name = "data_master_wr_u";
}
SIGNAL n
{
   name = "data_master_waitrequest_u";
}
SIGNAL o
{
   name = "data_master_endofpacket_u";
}
SIGNAL p
{
   name = "data_master_writedata_u";
   radix = "hexadecimal";
}
SIGNAL q
```

```
          {
             name = "data_master_readdata_u";
             radix = "hexadecimal";
          }
          SIGNAL r
          {
             name = "streamingctrl_master_addr";
             radix = "hexadecimal";
          }
          SIGNAL s
          {
             name = "streamingctrl_master_rd";
          }
          SIGNAL t
          {
             name = "streamingctrl_master_readdata";
             radix = "hexadecimal";
          }
       }
    }
 }
 USER_INTERFACE
 {
    USER_LABELS
    {
       name = "Cache";
       technology = "Image Processing System";
    }
 }
 DEFAULT_GENERATOR
 {
    top_module_name = "Cache";
    black_box = "0";
    vhdl_synthesis_files = "Cache.vhd,cache_slave.vhd,cache_dpram_512x48.vhd,
cache_master.vhd";
    verilog_synthesis_files = "";
    black_box_files = "";
 }
}
```