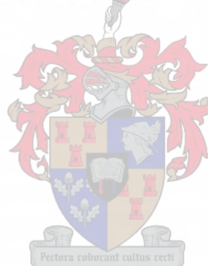
Development of a high speed PCI data capture card for the SUNSAT I ground station.

WJ van der Westhuizen



MARCH 2002

THESIS PRESENTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE MSc ENG (ELECTRONIC) DEGREE AT
THE UNIVERSITY OF STELLENBOSCH

SUPERVISOR: PROF. PJ BAKKES

Stellenbosch University http://scholar.sun.ac.za

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my original work, and has never been submitted, in part or in its entirety, at any University, for any requirements towards the achievement of any degree.

WJ van der Westhuizen

Stellenbosch University http://scholar.sun.ac.za

Abstract

The primary payload of the University of Stellenbosch's micro-satellite, SUNSAT I, was a high quality imager, capable of taking stereoscopic images of the surface of the earth. Although the orbit of SUNSAT I will have the satellite pass over the whole earth, contact via the SUNSAT I ground station at the University will only be made for 3% of each day. To be able to photograph any part of the earth onboard memory was provided on the satellite to store the image until it can be downloaded to the ground station. A high speed downlink was also added to the satellite to be able to download a complete image from the onboard memory in one pass and also to take realtime pictures as the satellite passes over the ground station. At that stage there was no way to capture the data at the ground station. A high speed digital data capture mechanism was needed. The thesis discusses the development of a high speed capture card.

Due to the high speeds needed it provided an excellent opportunity to develop the card using the PCI bus, the first design to do so at the University, as an interface with a computer. A prototype card was developed first to do proof of concept. It is shown that the prototype card fulfilled the functional requirements and it was also used to capture the first image from the complete satellite during the final tests before launch.

Finally a production card for use in the ground station was designed and assembled.

Stellenbosch University http://scholar.sun.ac.za

Opsomming

Die hoof loonvrag van die Universtieit van Stellenbosch se eerste mikrosatelliet, SUNSAT I, is ‘n hoë kwaliteit kamera wat stereoskopiese fotos van die oppervlak van die aarde kan neem. Alhoewel die satelliet oor die hele oppervlakte van die aarde gaan beweeg, is dit vir slegs 3% van die dag sigbaar vanaf die grondstasie by die Universiteit van Stellenbosch. Om ‘n foto van enige plek op aarde te kan neem is daar aanboord geheue op die satelliet geplaas om die foto te stoor totdat dit by die grondsatsie afgelaai kan word. ‘n Hoë spoed skakel is ook daargestel om die data in die geheue van die satelliet in een verbyvlug te kan aflaai. Dit stel die satelliet ook in staat om intydse fotos gedurende ‘n verbyvlug van die satelliet te kan neem. Op daardie stadium het daar nog nie ‘n manier bestaan om die hoë spoed data vas te lê nie. ‘n Meganisme om die hoë spoed digitale data te vang was nodig. Hierdie tesis bespreek die ontwikkeling van so ‘n data vang kaart.

Weens die hoë snelhede wat benodig word, het die PCI bus die ideale oplossing gebied om die data teen ‘n hoë spoed op ‘n persoonlike rekenaar te stoor. Dit was ook die eerste ontwerp aan die Universiteit wat van die PCI bus gebruik gemaak het. In die eerste instansie is ‘n prototipe ontwikkel om te bewys dat dit wel moontlik is om die data teen die benodigde tempo te kan stoor. Daar word gewys dat die prototipe aan die behoefte voldoen en dit is ook gebruik om die eerste beeld vanaf die volledige satelliet te vang voordat dit gelanseer is.

In die tweede instansie word ‘n produksiekaart ontwikkel en aanmekaar gesit.

Stellenbosch University http://scholar.sun.ac.za

To my wife and parents

Stellenbosch University http://scholar.sun.ac.za

Contents

Acknowledgements v

List of abbreviations and acronyms vi

List of figures viii

List of tables xi

1 Introduction 1

2 Design of the prototype 5

2.1 Specifications 5

2.2 Functional description 5

2.2.1 PCI interface 6

2.2.1.1 The AMCC S5933 Matchmaker PCI controller 6

2.2.1.2 Signal descriptions 8

2.2.1.3 Operation registers 10

2.2.2 FPGA interface 15

3 Implementation of the prototype design 16

3.1 Hardware implementation 16

3.1.1 Data interface 17

3.1.1.1 External inputs 18

3.1.1.2 Signals 19

3.1.1.3 VHDL implementation 19

3.1.1.4 Simulations 23

3.1.2 S5933 interface 23

3.1.2.1 Mailbox implementation 23

3.1.2.2 FIFO implementation 38

3.2 Software 66

Stellenbosch University http://scholar.sun.ac.za

3.2.1 Flat Real Mode 66
3.2.1.1 What is Flat Real Mode? 66
3.2.1.2 Implementation 68
3.2.1.3 Drivers 71
3.2.1.4 Processor mode 72
3.2.1.5 A20 line 74
3.2.1.6 Enabling Flat Real Mode 74
3.2.2 Allocate memory 80
3.2.2.1 Testing allocated memory 80
3.2.3 Capture data 81
3.2.4 Check PCI device 82
3.2.4.1 PCI BIOS installation check 85
3.2.4.2 Find PCI device 86
3.2.4.3 Configuration read 87
3.2.4.4 Error code routines 90
3.2.5 Calculation of addresses 91
3.2.6 Capturing of the data 93
3.2.6.1 Mailbox implementation 94
3.2.6.2 FIFO implemtation 96
3.2.7 Store data on hard disk 99
3.2.8 Deallocating the memory 100
3.3 Evaluation of the prototype 100
4 Design of the final product 104
4.1 Overview 104
4.2 Specification 104
4.3 Functional description 105
4.3.1 PCI interface 107
4.3.2 Controller interface 107

Stellenbosch University http://scholar.sun.ac.za

4.3.3 Clock 108
4.3.4 Memory 109
4.3.5 Data interface 109
4.3.6 External interface 109
5 Implementation of the final product design 112
5.1 Overview 112
5.2 Memory 113
5.2.1 Differences between buffered and unbuffered DIMMs 116
5.2.1.1 Unbuffered DIMMs 116
5.2.1.2 Buffered DIMMs 119
5.2.2 The serial presence detect (SPD) function 122
5.2.3 DIMM speed classification 126
5.2.4 Choice of DIMM to be used on the capture card 126
5.3 Controller 128
5.4 Clock 130
5.4.1 ICD2053 registers 131
5.4.1.1 Control register 131
5.4.1.2 Program register 132
5.5 Data interface 135
5.6 PCI interface 135
6 Hardware implementation of final product design 136
6.1 PCB layout 136
6.2 Population of the card 140
7 Conclusion 141
A Mailbox implementation 143
A1 VHDL 143
A2 Pinout 150
B FIFO implementation 152

Stellenbosch University http://scholar.sun.ac.za

B1 VHDL 152

B2 Pinout 166

C Software 169

D Serial presence detect (SPD) 192

E DIMM 195

E1 Pinout 195

E2 Functional description 196

Bibliography 199

Stellenbosch University http://scholar.sun.ac.za

Acknowledgements

- I would not have been able to finish this project without the help of God the Almighty. He gave me the inspiration and perseverance to finish it.
- Also a special thanks to my parents and my wife for their support and understanding.
- Prof. Bakkes thank you for your continued pressure to ensure that this document was finally finished.
- Thank you to Hans Grobler and Nicky Steenkamp for the information and suggestions towards the software routines.
- Finally thank you to the whole SUNSAT I launch team, especially to Prof JJ du Plessis, Sias Mostert, Johan Arendse, Johan Grobbelaar and Dirk van der Merwe. It was great working with you guys on such a groundbreaking project as South Africa's first orbiting satellite.

Stellenbosch University http://scholar.sun.ac.za

List of abbreviations and acronyms

APRS	Automatic Position Reporting System
ASIC	Application Specific Integrated Circuit
BGA	Ball-Grid Array
BIOS	Basic Input Output System
CPU	Central Processing Unit
DIMM	Dual In-Line Memory Module
DMA	Direct Memory Access
DOS	Disk Operating System
DWORD	Double WORD
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
GB	Gigabyte
GPS	Global Positioning System
H	Hexadecimal
HMA	High Memory Area
I/O	Input Output
INT	Interrupt
JPL	Jet Propulsion Laboratory
kB	Kilobit
km	Kilometre
LSB	Least significant Bit
m	Metre
Mb	Megabyte
MB/s	Megabit per second

Stellenbosch University http://scholar.sun.ac.za

MHz	Megahertz
NASA	National Aeronautics and Space Administration
ns	Nanosecond
PC	Personal Computer
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PM	Protected Mode
QPSK	Quaternary Phase-Shift Keying
RAM	Random Access memory
RM	Real Mode
SDRAM	Synchronous Dynamic Random Access Memory
SIMM	Single In-Line Memory Module
SUNSAT	Stellenbosch University Satellite
TSR	Terminate and Stay Resident
UTC	Universal Time Constant
V	Volt
VHDL	Very High Speed Integrated Circuit Hardware Description Language
WOD	Whole Orbit Data
XMS	Extended Memory

Stellenbosch University http://scholar.sun.ac.za

List of figures

1.1 Delta II 1

1.2 Image downloaded from SUNSAT I 3

2.1 Prototype high level block diagram 5

2.2 S5933 block diagram 7

2.3 S5933 signal pins 8

2.4 Add-on register read 13

2.5 Add-on register write 14

2.6 Asynchronous FIFO RDFIFO# access 15

3.1 FPGA interface 16

3.2 High level flow diagram: Data interface 18

3.3 Data interface flow diagram 20

3.4 Serial to parallel converter 21

3.5 Shift register simulation 23

3.6 Data_Valid simulation 23

3.7 Mailbox implementaion hogh level flow diagram 24

3.8 AMBEF read state diagram 26

3.9 AOMB write state diagram 28

3.10 Data_IO process flow diagram 35

3.11 Mailbox implementation read and write cycle 36

3.12 Wait until data is read from AOMB 37

3.13 Data is lost and Data_Error is asserted 37

3.14 FIFO implementation high level flow diagram 39

3.15 Synchronise a signal ta a clock 43

Stellenbosch University http://scholar.sun.ac.za

3.16 AIMB read state diagram 45
3.17 FIFO state machine state diagram 49
3.18 FIFO implementation AOMB write state diagram 53
3.19 Set_DQ flow diagram 57
3.20 FIFO state machine 63
3.21 FIFO_Counter 64
3.22 AOMB state machine 64
3.23 AIMB state machine 65
3.24 Enabling of the FIFO implementation interface 65
3.25 Data path 66
3.26 The stack 69
3.27 Main program flow diagram 70
3.28 The CR0 register 72
3.29 Segment descriptor 75
3.30 *Capture* high level flow diagram 82
3.31 PCI device checks 83
3.32 Configuration space header 88
3.33 Mailbox implementation software flow diagram 94
3.34 FIFO implementation software flow diagram 97
3.35 *MemToHD* routine 99
3.36 Taking a photo with SUNSAT 102
3.37 Prototype ground station setup 102
3.38 Prototype data capture card 103

4.1 Data capture card high level block diagram 105
4.2 Final design basic flow diagram 106
4.3 Controller interface 107
4.4 Controller flow diagram 108

Stellenbosch University http://scholar.sun.ac.za

5.1 Type and voltage keying 114
5.2 Unbuffered DIMM block diagram 117
5.3 Buffered vs unbuffered operation 119
5.4 Buffered DIMM block diagram 120
5.5 SPD wiring option 123
5.6 SPD data valid 125
5.7 SPD start and stop 125
5.8 SPD acknowledge 125
5.9 ICD2053B control register 131
5.10 ICD2053B control register definition 132
5.11 ICD2053B program register 132
5.12 CDC516 function table 134

6.1 Final data capture card diagram 137

Stellenbosch University http://scholar.sun.ac.za

List of tables

2.1 S5933 signal description 8

2.2 PCI bus operation registers 10

2.3 Add-on interface registers 11

3.1 External inputs 19

3.2 Signal definition 19

3.3 Mailbox implementation S5933 interface signals 24

3.4 Mailbox implementation signals 25

3.5 Output of Data_Valid_Process process 32

3.6 FIFO implementation S5933 interface signals 39

3.7 FIFO implementation signals 40

3.8 AIMB_Signals process outputs 47

3.9 FIFO_Signals process outputs 50

3.10 AOMB_State_Machine process outputs 54

3.11 RW_DQ process outputs 56

3.12 Set_Z process outputs 58

3.13 Set_wrfifo_and_rdfifio process outputs 59

3.14 Set_Selct process outputs 60

3.15 Set_BE process outputs 60

3.16 Set_ADR process outputs 61

3.17 Set_WR process outputs 61

3.18 Set_RD process outputs 62

3.19 PCI BIOS function calls 83

3.20 Return codes 84

3.21 PCI operation registers 92

Stellenbosch University http://scholar.sun.ac.za

4.1 SUNSAT external signals 110

4.2 Decoded SUNSAT signals 110

4.3 Lantaba external signals 110

4.4 R+ and R- decoded 111

5.1 Simplified pin function description 114

5.2 Unbuffered DIMM input capacitance 118

5.3 Buffered DIMM input capacitance 121

5.4 SPD interface signals 122

5.5 Available command truth table 127

5.6 ICD2053B pin description 130

6.1 Additional signals 138

A.1 Mailbox implementation pinout 150

B.1 FIFO implementation pinout 166

D.1 SPD EEPROM data format 192

E.1 DIMM pinout 195

E.2 Functional description 196

Stellenbosch University http://scholar.sun.ac.za

Chapter 1

Introduction

In 1992 the Department of Electrical and Electronic Engineering at the University of Stellenbosch embarked on a project of building their own micro satellite. The satellite was aptly named SUNSAT, Stellenbosch UNiversity SATellite.

The primary payload of SUNSAT I is a high resolution imager, capable of taking stereoscopic three colour images with a spatial resolution of 15m per pixel. The three colour bands fall in the green, near-infrared and infrared sections of the spectrum respectively, and can produce useful information relating to vegetation cover on the earth's surface. The satellite also contains various secondary payloads and experiments. These include a high accuracy GPS receiver, a scientific magnetometer, star camera and a number of school experiments.

Figure 1.1 Delta II

Stellenbosch University http://scholar.sun.ac.za

SUNSAT I was launched by NASA as a secondary payload on a Delta 2 launch vehicle on 23 February 1999. The primary mission payload was an experimental scientific satellite called ARGOS. Another secondary payload, which was a micro-satellite from Denmark entitled Ørstedt, was launched with SUNSAT, and together they were placed in a polar, low earth orbit with a perigee of 520km and an apogee of 850km. Figure 1.1 shows the Delta II launch vehicle ready for launch.

The last communication with SUNSAT I took place on 19 January 2001. The cause of the communications failure was that an irreversible, probably physical, failure had occurred on the satellite. Nevertheless, the SUNSAT project exceeded all its original goals during its lifespan of nearly two years.

Statistics from SUNSAT I's operational life:

- 696 days between launch at 10:29:55 UTC on 23 February 1999 and the last contact at 15:22:37 UTC on 19 January 2001, giving 10027 orbits, or nearly 500 million kilometres
- 51 high-resolution images captured all over the globe, in 3 spectral bands and 15 m pixel sizes on ground
- 937 command diaries uploaded in operating SUNSAT
- 241700 telecommands executed successfully
- 161144 kB of whole orbit data (WOD) downloaded
- 94868 kB of GPS-data downloaded in support of JPL
- 7052 kB of data for the star camera experiment
- 3144 kB of APRS digilogs, renewing interest in APRS via satellite
- 1656 kB of Magnetometer data
- 888 kB of international school experimental data
- Several hours of PAL videotape data of Southern Africa

Figure 1.2 shows one of the images captured from SUNSAT I.

Stellenbosch University http://scholar.sun.ac.za

Figure 1.2 Image downloaded from SUNSAT I

One of the ways that data can be downloaded from the satellite was via a high speed S-band downlink. This high speed path makes it possible to download a complete image from the satellite's onboard memory, on the RAM tray, in a single pass of the satellite. It also provides the capability to take a realtime image as the satellite passes over the ground station.

It is now possible to download the data at a very high tempo, but there was no way to store the data for easy access. There was a need for an interface to capture the incoming data and store it on a computer's hard disk. This provided the ideal opportunity to develop a high speed PCI interface to capture the data.

The development of the PCI data capture card formed part of the SUNSAT I ground

Stellenbosch University http://scholar.sun.ac.za

station project. Because of the tight time schedule and limited budget, the development of the card had to follow a complete development cycle. The first step was to develop a prototype to prove that the concept can be implemented. The prototype was designed, implemented and tested. Only then was a final production card designed. The designs of the final card had to be approved by a team of project engineers in a design meeting before the PCB layout was done. After the layout was done several checks were done on the PCB layout before it finally manufactured.

This document will follow this path that was followed to finally produce the populated data capture card. Another purpose of this document is to serve as a manual to the person using the data capture card for further development.

Stellenbosch University http://scholar.sun.ac.za

Chapter 2

Design of the prototype

2.1 Specifications

The data that needs to be stored is in the form of a serial data stream with a strobe line. The strobe signal is active low. At that stage the SUNSAT I orbit required a data storage rate of 40Mb/s. The prototype has to be capable of handling an input frequency of up to 40 MHz.

2.2 Functional description

Since the PCI bus has a 32-bit wide data bus, the serial data is first converted into 32-bit wide data packets which are then saved in the computers memory via the PCI bus. By converting the incoming data stream in this way the frequency at which the data needs to be stored is 32 times slower. For the serial data stream a bit had to be saved every 25ns while for the 32-bit data stream 800ns is available to store the data before the next 32-bits has to be saved. As shown in figure 2.1, the FPGA

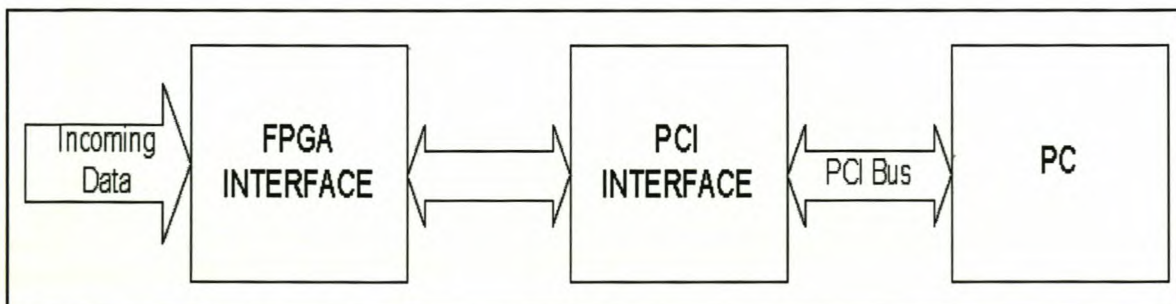


Figure 2.1 Prototype high level block diagram

Stellenbosch University http://scholar.sun.ac.za

interface is used to convert the incoming data into 32-bit packets and to control the PCI interface to send the data to the computer. The PCI interface handles the complexities of the PCI bus and allows the FPGA interface and the computer to communicate with each other.

2.2.1 PCI interface

In choosing a PCI chipset, the first choice was between a FPGA and an ASIC. At that stage the only ASIC available in South Africa was the AMCC S5933 PCI controller. Both ALTERA and XILINX provided their PCI cores for free, as long as it is being used for academic purposes only. The PCI cores are already implemented PCI controllers. These cores are very expensive, because it can be completely customised to the specific application. The user's own application can also be incorporated in the same FPGA, making the end product smaller and more cost effective. One of the planned applications for the final data capture card is to be used with other commercial applications of SUNSAT technology, which would mean that the PCI core had to be bought from the FPGA vendor. At that stage the prototype was urgently needed to test the satellite. This meant that a fast answer had to be found. The solution came in form of the AMCC S5933 PCI Matchmaker developer's kit. The kit provided everything needed to immediately commence the design FPGA interface.

2.2.1.1 The AMCC S5933 Matchmaker PCI controller

The S5933 is a powerful and flexible PCI controller supporting several levels of interface sophistication. At the lowest level, it can serve simply as a PCI bus target with modest transfer requirements. For high-performance applications, the S5933 can become a Bus master to attain the PCI bus peak transfer capability of 132 MB/s. The S5933 allows the designer to focus on the actual application, not debugging the PCI interface.

Stellenbosch University http://scholar.sun.ac.za

The block diagram in figure 2.2 shows the major functional elements within the S5933. The S5933 provides three physical bus interfaces: the PCI bus, the add-on bus and an optional external non-volatile memory. Data movement can occur between the PCI bus and the add-on bus or the PCI bus and the non-volatile memory. Transfers between PCI and add-on buses can take place through the mailbox registers or the FIFOs, or can make use of the pass-thru data path. FIFO transfers on the PCI bus can be performed either through software control or through hardware, using the S5933 as bus master.

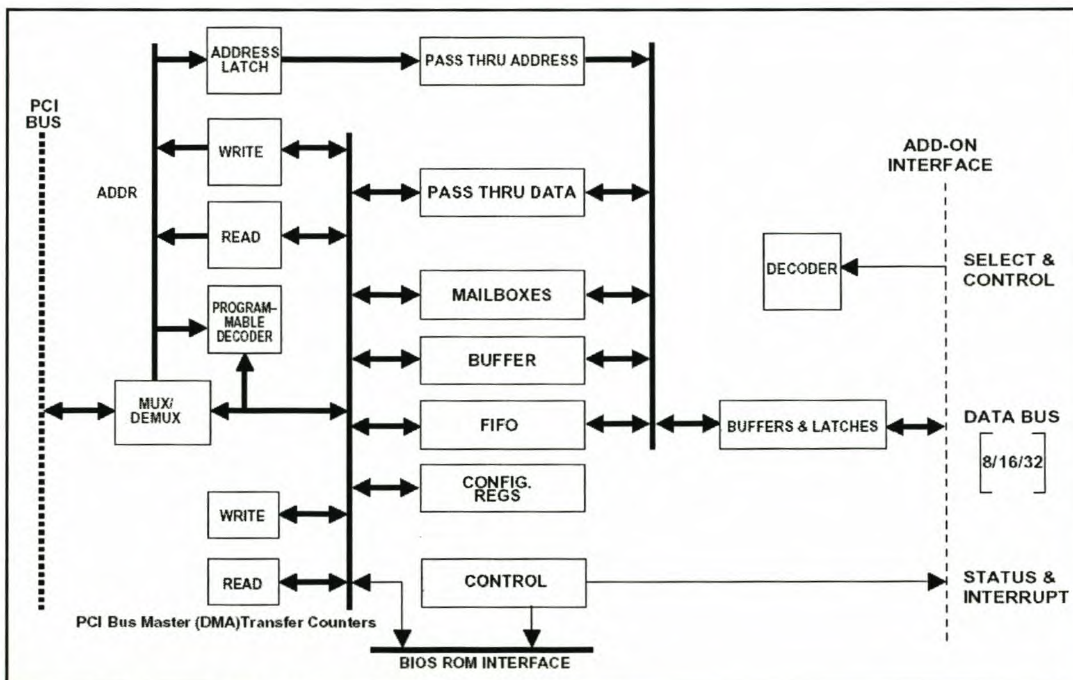


Figure 2.2 S5933 block diagram

The S5933 provides a simple, general-purpose interface to the add-on bus. The add-on data path is a 32-bit bus. Data transfers to and from the S5933 internal registers are accomplished through a chip select decode in conjunction with either a read or write strobe.

Stellenbosch University http://scholar.sun.ac.za

2.2.1.2 Signal descriptions

Figure 2.3 shows the complete list of the S5933 signal pins.

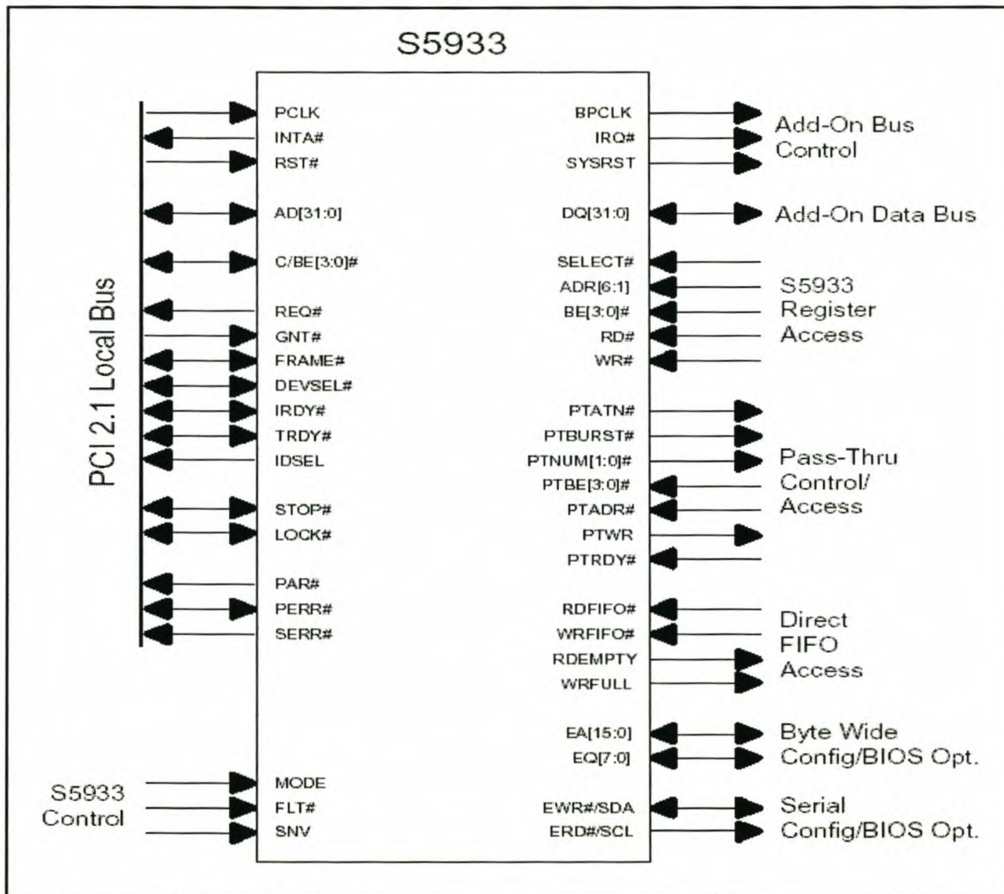


Figure 2.3 S5933 signal pins

Table 2.1 provides a description of the signals used by the prototype. For a complete list refer to the S5933 datasheet provided on the attached CD, under */DATASHEETS/AMCC*.

Table 2.1 S5933 signal description

Signal	Description
CLK	Clock. The rising edge of this signals the reference upon which all other signals are based. The maximum frequency is 33 MHz and the minimum is 0 Hz

Stellenbosch University http://scholar.sun.ac.za

DQ[31:00]	Datapath DQ0 - DQ31. These pins represent the datapath for the add-on peripheral's data bus.
ADR[6:2]	Add-on Adresses. These signals are the address lines to select one of the 16 DWORD registers within the controller. ADR[6:2] — Register Name 0 0 0 0 0 — Add-on Incoming Mailbox Reg. 1 0 0 0 0 1 — Add-on Incoming Mailbox Reg. 2 0 0 0 1 0 — Add-on Incoming Mailbox Reg. 3 0 0 0 1 1 — Add-on Incoming Mailbox Reg. 4 0 0 1 0 0 — Add-on Outgoing Mailbox Reg. 1 0 0 1 0 1 — Add-on Outgoing Mailbox Reg. 2 0 0 1 1 0 — Add-on Outgoing Mailbox Reg. 3 0 0 1 1 1 — Add-on Outgoing Mailbox Reg. 4 0 1 0 0 0 — Add-on FIFO Port 0 1 1 0 1 — Add-on Mailbox Empty/Full Status 0 1 1 1 1 — Add-on General Control/Status Register
BE[3:0]#	Byte Enable. These pins provide for individual byte control during register read or write operations.
SELECT#	Select for the add-on interface.
WR#	Write strobe.
RD#	Read strobe.
WRFIFO#	Write FIFO.
RDFIFO#	Read FIFO.
WRFULL	Write FIFO full. This pin indicates if the add-on-to-PCI FIFO is able to accept more data.
BPCLK	Buffered PCI clock.

Stellenbosch University http://scholar.sun.ac.za

2.2.1.3 Operation registers

a PCI bus

The PCI bus operation registers are mapped as 16 consecutive DWORD registers located at the address space specified by the Base Address Register 0 of the PCI bus. These locations are the primary method of communication between the PCI and add-on buses. Data can be exchanged either through the mailboxes, transferred through the FIFO in blocks under program control or under Bus master control. Table 2.2 lists the PCI bus operation registers.

Table 2.2 PCI bus operation registers

Abbreviation	Register Name
OMB1	Outgoing Mailbox Register 1
OMB2	Outgoing Mailbox Register 2
OMB3	Outgoing Mailbox Register 3
OMB4	Outgoing Mailbox Register 4
IMB1	Incoming Mailbox Register 1
IMB2	Incoming Mailbox Register 2
IMB3	Incoming Mailbox Register 3
IMB4	Incoming Mailbox Register 4
FIFO	FIFO register port (bidirectional)
MWAR	Master Write Address Register
MWTC	Master Write Transfer Count Register
MRAR	Master Read Address Register
MRTC	Master Read Transfer Count Register
MBEF	Mailbox Empty/Full Status
INTCSR	Interrupt Control/Status Register
MCSR	Bus Master Control/Status Register

Stellenbosch University http://scholar.sun.ac.za

For a detailed description of the PCI Bus Operation Register set refer to the S5933 datasheet provided on the attached CD, under */DATASHEETS/AMCC*.

b Add-on bus

The add-on bus interface provides access to 18 DWORDs (72 bytes) of data, control and status information. This register group represents the primary method for communication between the add-on and PCI buses. The flexibility of this arrangement allows a number of user-defined software protocols to be built. The register structure is very similar to that of the PCI operation register set. The major difference between the PCI bus and the add-on bus register sets are the absence of bus master controls registers on the add-on side and the addition of two pass-through registers. Table 2.3 lists the add-on interface registers.

Table 2.3 Add-on interface registers

Abbreviation	Register Name
AIMB1	Add-on Incoming Mailbox Register 1
AIMB2	Add-on Incoming Mailbox Register 2
AIMB3	Add-on Incoming Mailbox Register 3
AIMB4	Add-on Incoming Mailbox Register 4
AOMB1	Add-on Outgoing Mailbox Register 1
AOMB2	Add-on Outgoing Mailbox Register 2
AOMB3	Add-on Outgoing Mailbox Register 3
AOMB4	Add-on Outgoing Mailbox Register 4
AFIFO	Add-on FIFO Register Port
MWAR	Bus Master Write Address Register
APTA	Add-on Pass-Through Address Register
APTD	Add-on Pass-Through Data Register
MRAR	Bus Master Read Address Register

Stellenbosch University http://scholar.sun.ac.za

AMBEF	Add-on Mailbox Empty/Full Status Register
AINT	Add-on Interrupt Control Register
AGCSTS	Add-on General Control and Status Register
MWTC	Bus Master Write Transfer Count Register
MRTC	Bus Master Read Transfer Count Register

For a detailed description of the Add-On Bus Operation Register set refer to the S5933 datasheet provided on the attached CD, under */DATASHEETS/AMCC*.

c Mailbox

The S5933 has eight 32-bit mailbox registers. The mailboxes are useful for passing command and status information between the add-on and PCI bus. The PCI interface has four incoming mailboxes (add-on to PCI) and four outgoing mailboxes (PCI to add-on). The add-on interface has four incoming mailboxes (PCI to add-on) and four outgoing mailboxes (add-on to PCI). The PCI incoming and add-on outgoing mailboxes are the same, internally. The add-on incoming and PCI outgoing mailboxes are also the same, internally.

The mailbox status may be monitored in two ways. The PCI and add-on interfaces each have a mailbox status register to indicate the empty/full status of bytes within the mailboxes. As the outgoing mailbox of the one interface is the incoming mailbox of the other interface, the incoming mailbox status bits on one interface are identical to the corresponding outgoing mailbox status bits on the other interface. A write to an outgoing mailbox sets the status bits for that mailbox. Reading the incoming mailbox clears all corresponding status bits for that mailbox. The mailboxes may also be configured to generate interrupts to the PCI and/or add-on interface.

The following list shows the relationship between the mailbox registers on the PCI and add-on interfaces.

Stellenbosch University http://scholar.sun.ac.za

PCI Interface		**Add-on Interface**
OMB1	=	AIMB1
OMB2	=	AIMB2
OMB3	=	AIMB3
OMB4	=	AIMB4
IMB1	=	AOMB1
IMB2	=	AOMB2
IMB3	=	AOMB3
IMB4	=	AOMB4
MBEF	=	AMBEF

Figure 2.4 shows the timing diagram for a standard add-on register read, which is used to read any of the add-on mailbox registers.

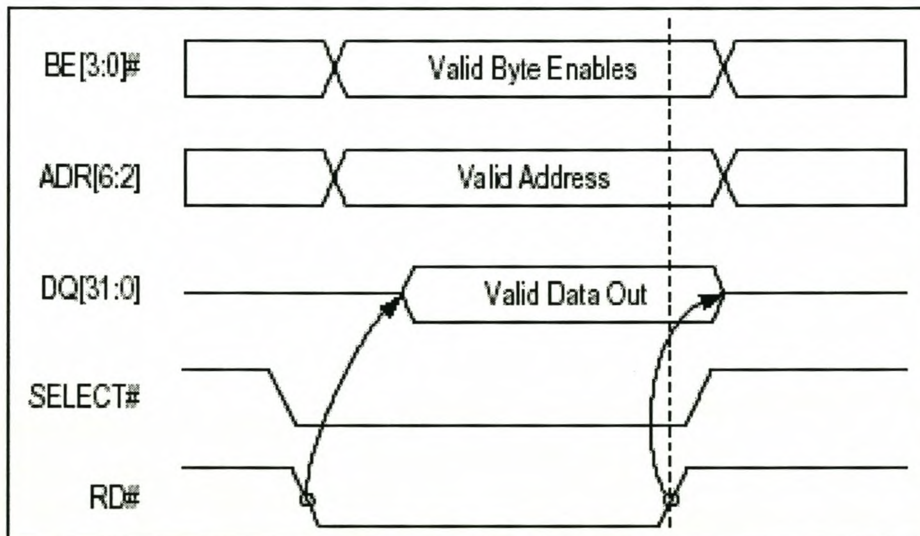


Figure 2.4 Add-on register read

Figure 2.5 shows the timing diagram for a standard add-on register write, which is used to write to any of the add-on mailbox registers.

Stellenbosch University http://scholar.sun.ac.za

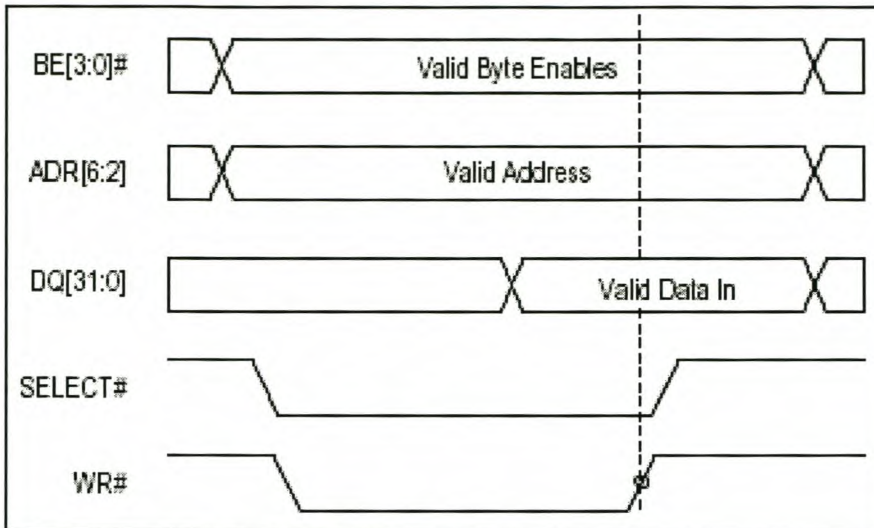


Figure 2.5 Add-on register write

d FIFO

The S5933 FIFO interface allows a high degree of functionality and flexibility. Different FIFO management schemes, endian conversion schemes, and advance conditions allow for a wide variety of add-on interfaces. Applications may implement the FIFO as either a PCI target or program it to enable the S5933 to be a PCI bus master.

The S5933 has two internal FIFOs. One FIFO is for PCI bus to add-on bus, the other is for add-on bus to PCI bus transfers. Each of these has eight 32-bit registers. The FIFOs are both addressed through a single PCI/Add-on Operation Register offset, but which internal FIFO is accessed is determined by whether the access is a read or write.

The add-on interface implements FIFO status pins to indicate the full and empty conditions of the PCI to add-on and add-on to PCI FIFOs. The RDEMPTY and WRFULL status outputs are always available to the add-on. Their functions are listed below.

Stellenbosch University http://scholar.sun.ac.za

Signal	**Function**
RDEMPTY	Indicates empty condition of the PCI to add-on FIFO
WRFULL	Indicates full condition of the add-on to PCI FIFO

The add-on interface also implements FIFO control pins to manipulate the S5933's FIFOs. The RDFIFO# and WRFIFO# inputs are always available. These pins allow direct access to the FIFO without generating a standard add-on register access using RD#, WR#, SELECT#, address pins and the byte enables.

The FIFO may be accessed through the Add-on FIFO Port Register (AFIFO) read or write. Depending on the device configuration, this register can be accessed either synchronous to BPCLK or asynchronous to BPCLK. The default configuration for the S5933 is that the AFIFO is accessed asynchronous to BPCLK.

Figure 2.6 shows an asynchronous FIFO register RDFIFO# access.

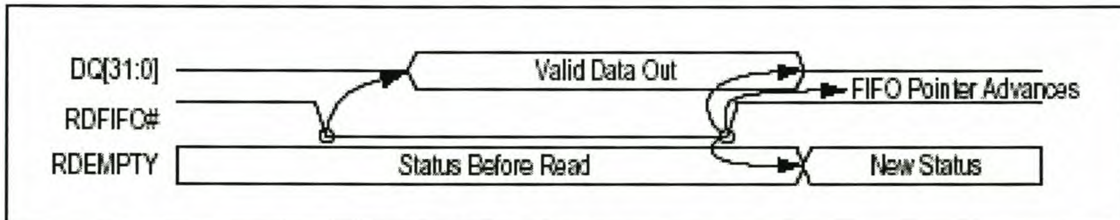


Figure 2.6 Asynchronous FIFO RDFIFO# access

2.2.2 FPGA interface

Due to the time frame in which the prototype was needed the FPGA tutor board designed by Markus Leugner [LEU96] was used for the FPGA interface. This was the most cost effective and least time consuming way of obtaining a FPGA that was capable of meeting the logic density and clock frequencies that were required. The tutor has an ALTERA EPF8636A FPGA and all the I/O pins are available for use. By using the tutor it was possible to immediately start implementing the interface and not spend time debugging the FPGA circuit.

Stellenbosch University http://scholar.sun.ac.za

Chapter 3

Implementation of the prototype design

3.1 Hardware implementation

The S5933 provides several ways in which the add-on application can communicate with the PCI bus. Since this was the first application to use the S5933 to interface with the PCI bus a simple approach was used. The first implementation of the prototype was done by using the S5933 mailbox registers. After the successful implementation of the mailbox interface a more complicated FIFO register interface was used. Although the FIFO may be either a PCI target or a PCI master, a target implementation was done.

The FPGA interface design consists of two parts, as shown in figure 3.1.

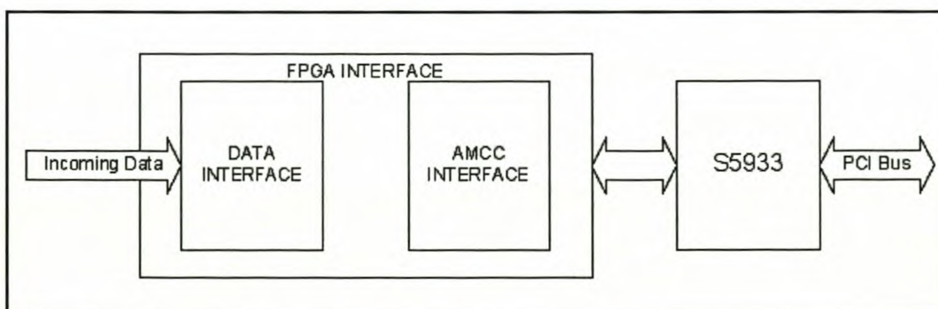


Figure 3.1 FPGA interface

The data interface handles the incoming serial data stream and then passes the processed data on to the AMCC interface, which passes it on to the PCI bus through

Stellenbosch University http://scholar.sun.ac.za

the S5933. The data interface is the same for both the mailbox and the FIFO implementation.

The implementation of the data interface and then the AMCC interface is discussed in the following sections. The mailbox implementation describes the use of the S5933 mailbox registers and the FIFO implementation describes how the mailbox registers are used in conjunction with the FIFO registers to transfer the data to and from the PCI bus.

All FPGA implementations were done in VHDL using the ALTERA development environment Max+plus II.

3.1.1 Data interface

Figure 3.2 shows the high level flow diagram of the data interface. As a bit is received, it is shifted into a shift register. Once 32 bits have been received the data in the shift register is stored in a temporary register. Then a flag is set to indicate that the data is ready for further processing.

The following sections lists the external input signal and the internal signals used to realize the complete interface. Then the implementation in VHDL is discussed.

Stellenbosch University http://scholar.sun.ac.za

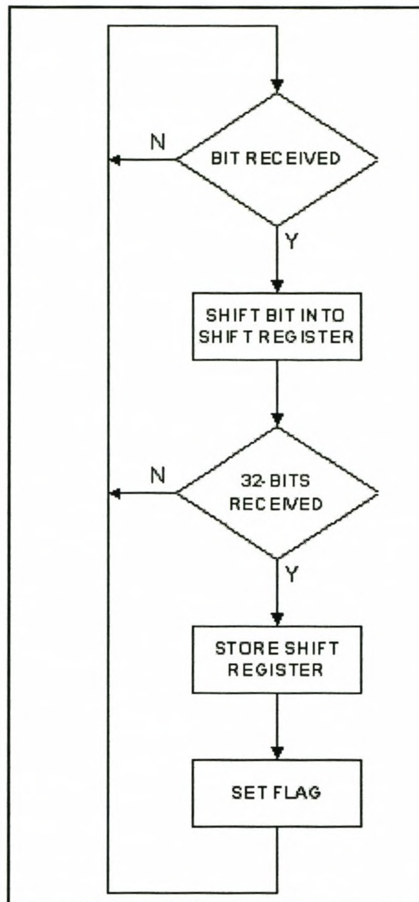


Figure 3.2 High level flow diagram: Data interface

3.1.1.1 External inputs

The incoming data is in the form of a serial data stream and an associated strobe line the external signal definitions are listed in table 3.1.

Stellenbosch University http://scholar.sun.ac.za

Table 3.1 External inputs

Name	**Input/ Output**	**Type**	**Description**
Data_In	in	std_logic	Incoming serial data stream
Data_Strb	in	std_logic	Incoming data strobe signal

3.1.1.2 Signals

Table 3.2 gives a list of all the signals used to implement the data interface as well as a description.

Table 3.2 Signal definition

Name	**Type**	**Initial value**	**Description**
Data_Counter	Integer range 0 to 31	n/a	Used to count the number of bits received.
Shift_Reg	Std_logic_vector (31 downto 0)	n/a	Used to implement a shift register for the incoming data
Data	Std_logic_vector (31 downto 0)	n/a	Used to implement a register to hold the data after 32-bits was received.
Data_Valid	Std_logic	0	Set when 32-bits was received.

3.1.1.3 VHDL implementation

Figure 3.3 shows the flow diagram of the data interface.

The incoming data is latched into a 32-bit long shift register. Once 32 bits have been received the output of the shift register is latched into an internal register and

Stellenbosch University http://scholar.sun.ac.za

Data_Valid is set to indicate that 32 bits were received.

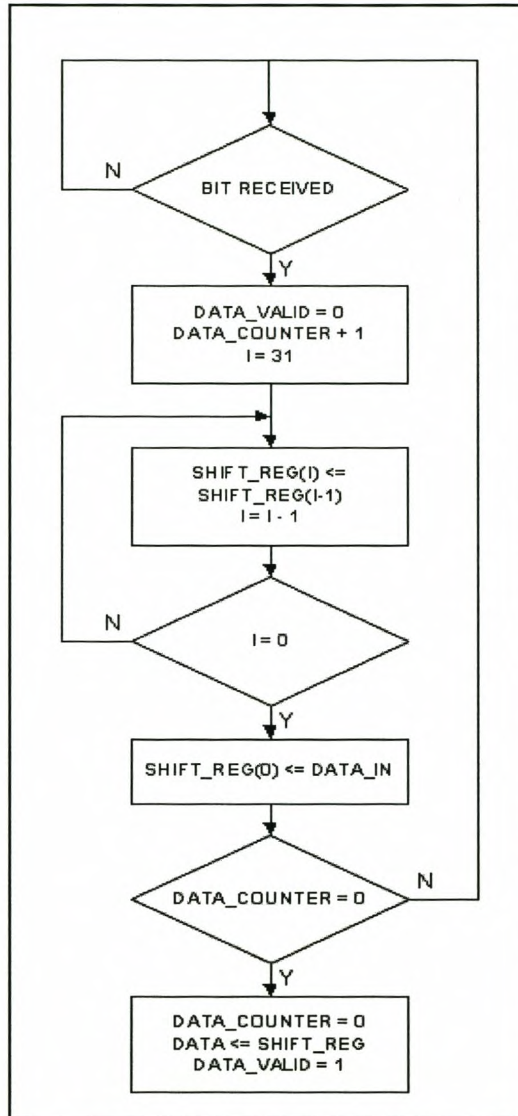


Figure 3.3 Data interface flow diagram

Data_Strobe_Counter is the process that counts the incoming bits. As soon as a bit arrives the counter is incremented. Since Data_Counter is defined between 0 and 31, a cyclic counter is implemented.

Stellenbosch University http://scholar.sun.ac.za


```
Data_Strobe_Counter:  process

begin
  wait until Data_Strb'event and Data_Strb = '1';
  Data_Counter <= Data_Counter +1;
end process Data_Strobe_Counter;
```

The serial-in, parallel-out shift register is implemented in the Shift_Register process. Figure 3.4 shows the logic implementation of the VHDL code.

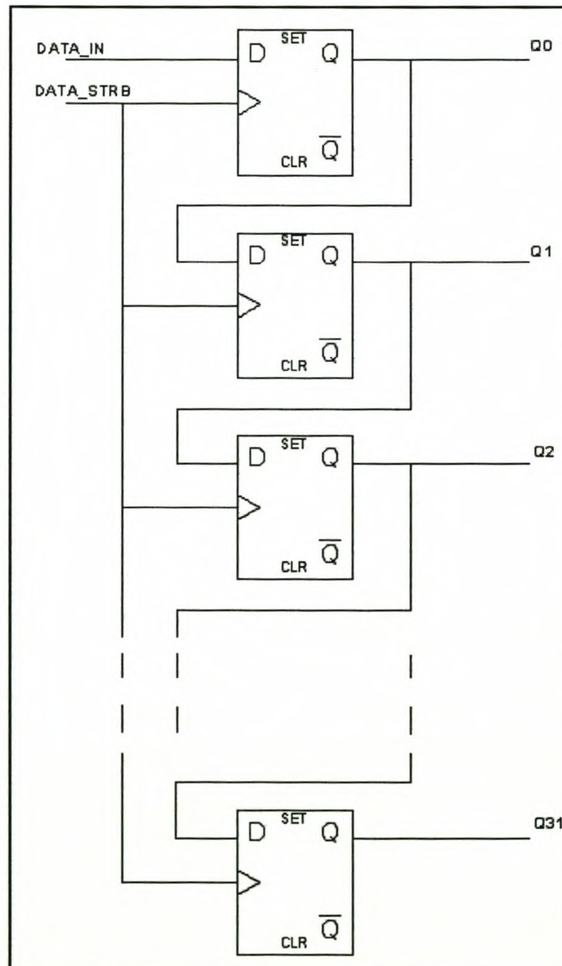


Figure 3.4 Serial to parallel converter

Each time a new bit arrives the bits in the register are shifted to the next and the new

Stellenbosch University http://scholar.sun.ac.za

bit is added in position 0.

```
Shift_Register: process

variable i : integer range 0 to 31;

begin
  wait until Data_Strb'event and Data_Strb = '1';
  for i in 31 downto 1 loop
    Shift_Reg(i) <= Shift_Reg(i-1);
  end loop;
  Shift_Reg(0) <= Data_In;
end process Shift_Register;
```

When 32 bits have been received, it is latched into a 32-bit register. Data_Output implements the register to which the received 32 bits are moved. Data_Valid is also set in the Data_Output process.

```
Data_Output: process

begin
  wait until Data_Strb'event and Data_Strb = '1';
  if Data_Counter = 0 then
    Data_Valid <= '1';
    Data <= Shift_Reg;
  else
    Data_Valid <= '0';
  end if;
end process Data_Output;
```

Stellenbosch University http://scholar.sun.ac.za

3.1.1.4 Simulations

Figure 3.5 shows the working of the shift register. As a bit is received, it is latched in.

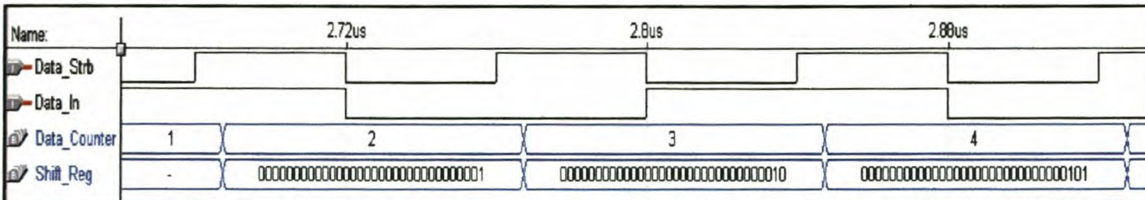


Figure 3.5 Shift register simulation

Figure 3.6 shows that after 32 bits have been received, it is latched into a register and Data_Valid is set.

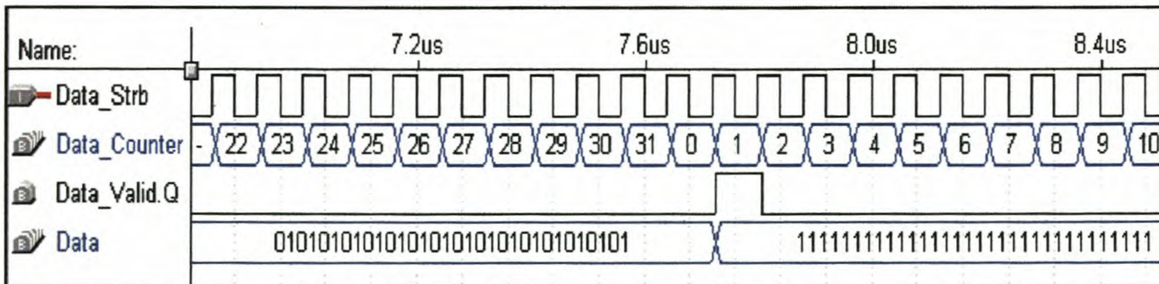


Figure 3.6 Data_Valid simulation

3.1.2 S5933 interface

3.1.2.1 Mailbox implementation

Figure 3.7 shows the high level flow diagram for the S5933 Mailbox interface. Once the flag that indicates that the data interface has received 32 bits is set, the mailbox interface checks to see if the previous data send to the PCI bus was read. If not a signal is set to indicate that 32 bits of data was lost. On the other hand if the previous data was read by the PCI bus then the new data is written to the outgoing mailbox register of the S5933.

Stellenbosch University http://scholar.sun.ac.za

The following sections lists the external input signal and the internal signals used to realize the complete interface. Then the implementation in VHDL is discussed.

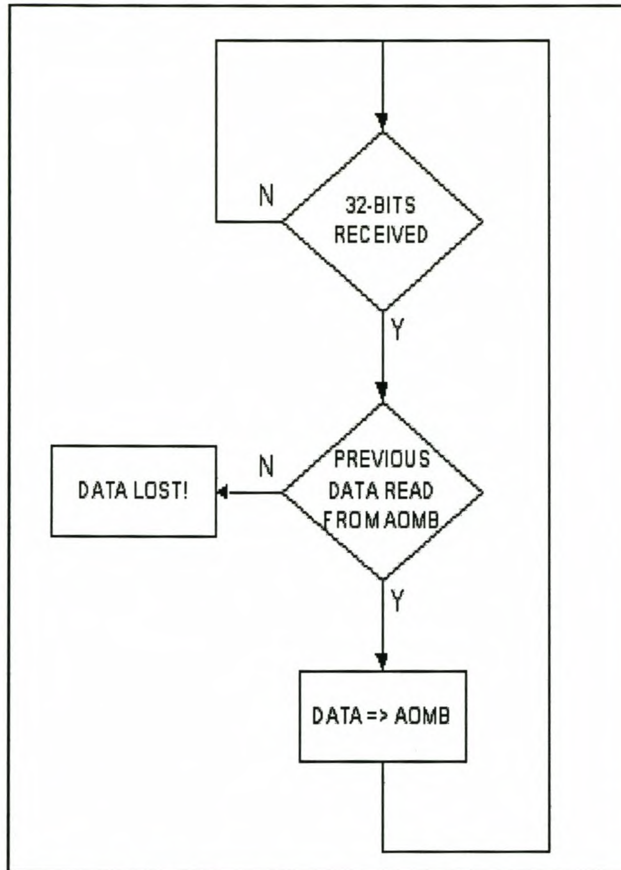


Figure 3.7 Mailbox implementation high level flow diagram

a External inputs

Table 3.3 lists the signals used to interface with the S5933.

Table 3.3 Mailbox implementation S5933 interface signals

Name	Input/ Output	Type	Description
CLK	in	std_logic	Buffered PCI clock.

Stellenbosch University http://scholar.sun.ac.za

DQ	inout	std_logic_vector(31 downto 0)	S5933 data path.
ADR	out	std_logic_vector(6 downto 2)	S5933 address lines.
BE	out	std_logic_vector(3 downto 0)	S5933 byte enable lines.
Selct	out	std_logic	S5933 select line.
RD	out	std_logic	S5933 read strobe.
WR	out	std_logic	S5933 write strobe.
DataError	out	std_logic	Indicates that AOMB 1 has been overwritten

b Signals

Table 3.4 gives a list of all the signals used to implement the mailbox interface as well as a description.

Table 3.4 Mailbox implementation signals

Name	**Type**	**Initial value**	**Description**
Data_Valid_Flag	std_logic	1	Indicates that incoming data is ready to be written to the mailbox register.
PCI_Counter	integer range 0 to 2	n/a	PCI clock counter.

Stellenbosch University http://scholar.sun.ac.za

Data_Read	std_logic_vector(31 downto 0)	n/a	Used to implement a register to store the data read from the S5933 interface.
Read_Flag	std_logic	0	Set if data must be read from DQ
Write_Flag	std_logic	0	Set if data must be written to DQ
MBox_E_F	std_logic	0	Used to indicate when data was read from the mailbox register.
Data_Read_Flag	std_logic	0	Set when data was read.
Fin_Read	std_logic		Set when finished reading AMBEF register.
Reset	std_logic	0	Used to initialize Data_Valid_Flag

c VHDL implementation

The S5933's AMBEF register indicates if the data in the AOMB register was read by the PCI bus. Bits 16 to 19 indicates if the data was read from AOMB 1. Figure 3.8 shows the state diagram of the state machine that reads the AMBEF register.

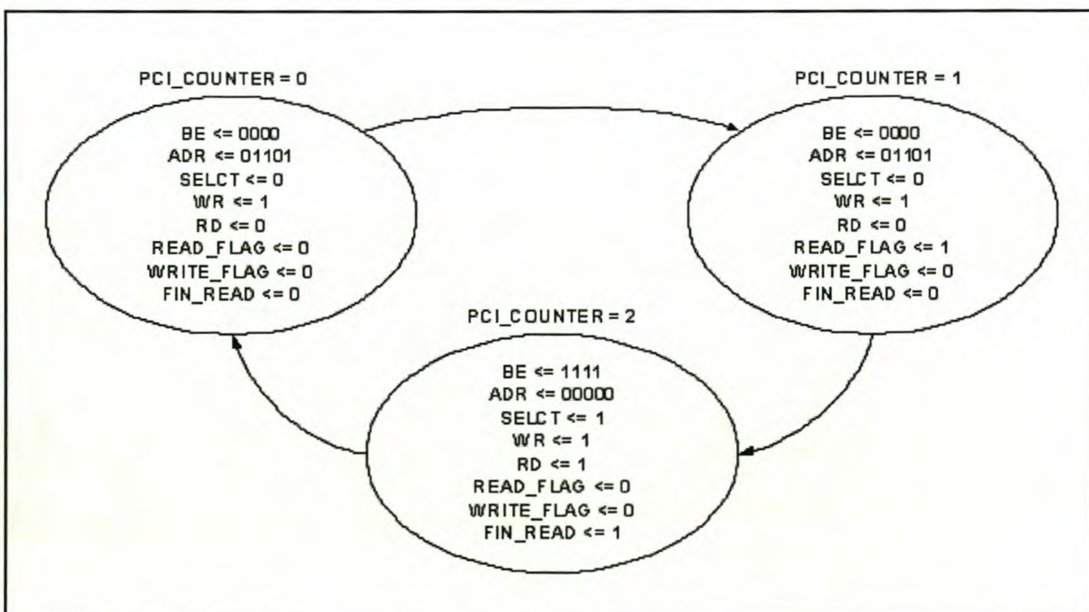


Figure 3.8 AMBEF read state diagram

Stellenbosch University http://scholar.sun.ac.za

This state machine is implemented by the following:

```
case PCI_Counter is
   when 0 =>
     Be <= "0000";
     ADR <= "01101";
     Selct <= '0';
     WR <= '1';
     RD <= '0';
     Read_Flag <= '0';
     Write_Flag <= '0';
     Fin_Read <= '0';
   when 1 =>
     Read_Flag <= '1';
     Write_Flag <= '0';
     Fin_Read <= '0';
   when 2 =>
     Selct <= '1';
     RD <= '1';
     WR <= '1';
     Be <= "1111";
     ADR <= "00000";
     Read_Flag <= '0';
     Write_Flag <= '0';
     Fin_Read <= '1';
   end case;
```

The add-on register read cycle can be completed in three clock cycles, as shown in figure 3.8. The flow of the state machine is controlled by the counter implemented in the Clock_Counter process:

Stellenbosch University http://scholar.sun.ac.za


```
Clock_Counter: process

begin
  wait until clk'event and clk = '1';
  if PCI_Counter = 2 then
    PCI_Counter <= 0;
  else
    PCI_Counter <= PCI_Counter + 1;
  end if;
end process Clock_Counter;
```

PCI_Counter is defined as an integer between 0 and 2, which means that 2 bits will be used to implement PCI_Counter. That is why the IF-statement is necessary to implement the 0 to 2 cyclic counter.

Data must also be written to the AOMB. The write cycle can also be completed in three clock cycles, as shown in figure 3.9. Figure 3.9 shows the state diagram of the state machine that is used to write data to AOMB 1.

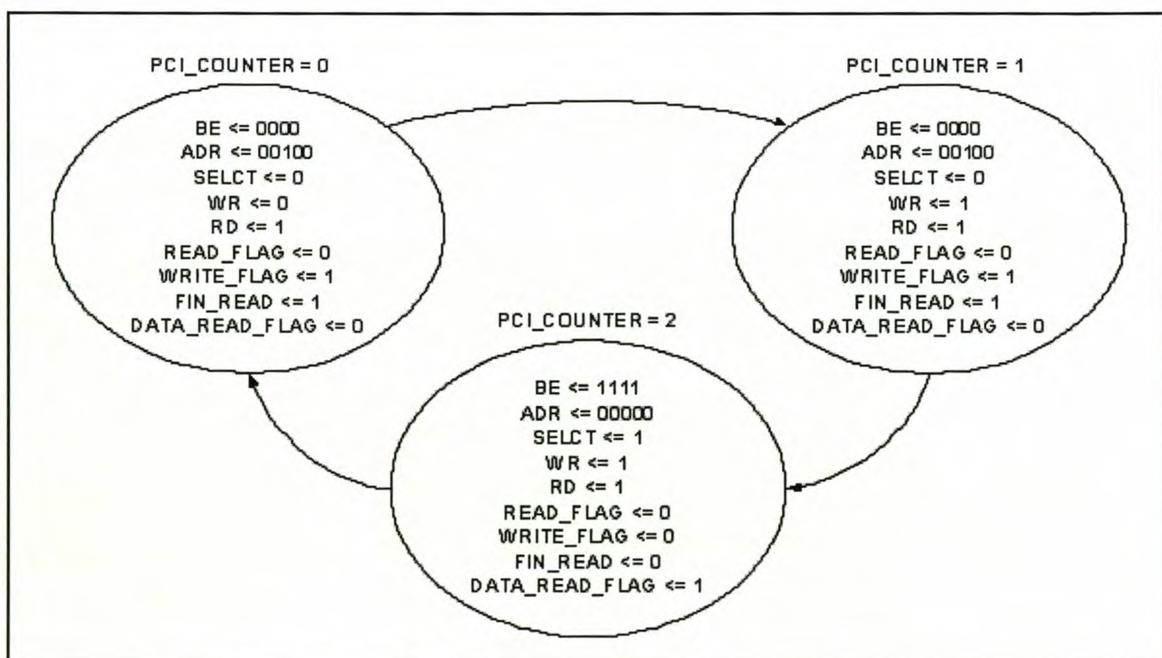


Figure 3.9 AOMB write state diagram

Stellenbosch University http://scholar.sun.ac.za

This state machine is implemented by the following:

```
case PCI_Counter is
  when 0 =>
    Data_Read_Flag <= '0';
    Be <= "0000";
    ADR <= "00100";
    Selct <= '0';
    WR <= '0';
    RD <= '1';
    Read_Flag <= '0';
    Write_Flag <= '1';
    Fin_Read <= '1';
  when 1 =>
    Data_Read_Flag <= '0';
    WR <= '1';
    Selct <= '0';
    Read_Flag <= '0';
    Write_Flag <= '1';
    Fin_Read <= '1';
  when 2 =>
    Selct <= '1';
    WR <= '1';
    RD <= '1';
    Be <= "1111";
    ADR <= "00000";
    Data_Read_Flag <= '1';
    Read_Flag <= '0';
    Write_Flag <= '0';
    Fin_Read <= '0';
  end case;
```

Stellenbosch University http://scholar.sun.ac.za

Data is read and written to DQ by setting the Read_Flag and Write_Flag signals. The Set_DQ process either read from or write to DQ according the these two signals.

```
Set_DQ: process

begin
  if Read_Flag = '1' then
    DQ <=  (others =>'Z');
    Data_Read <= DQ;
  elsif Write_Flag = '1' then
    DQ <= data;
  else
    DQ <= (others =>'Z');
  end if;
end process Set_DQ;
```

When Read_Flag is asserted, DQ is set to tri-state and then the data from the AMBEF register is read to Data_Read. When Write_Flag is asserted, DQ is set to data. If none of these signals are asserted, DQ is tri-stated.

After the AMBEF has been read, Fin_Read is asserted. The MBEF process checks if the data that was written to AOMB 1 has been read by the PCI bus.

```
MBEF: process

begin
  wait until Fin_Read'event and Fin_Read = '1';
  if (Data_Read(19) = '1') or (Data_Read(18) = '1') or
     (Data_Read(17) = '1') or (Data_Read(16) = '1')   then
    MBox_E_F <= '0';
  else
```

Stellenbosch University http://scholar.sun.ac.za


```
    MBox_E_F <= '1';
  end if;
end process MBEF;
```

If the data has not been read, any of Data_Read(19:16) set, then Mbox_E_F is not set. If the data has been read, thus the mailbox is empty, Mbox_E_F is set.

Data_Valid_Flag is one of the signals that determines when data can be written to the AOMB 1 register. The Data_Valid_Process process generates the Data_Valid signal.

```
Data_Valid_Process: process

begin
  if Reset = '1' then
    if Data_Valid = '1' then
      Data_Valid_Flag <= '1';
    elsif Data_Read_Flag = '1' then
      Data_Valid_Flag <= '0';
    else
      Data_Valid_Flag <= Data_Valid_Flag;
    end if;
  else
    Data_Valid_Flag <= '0';
  end if;
end process Data_Valid_Process;
```

The initial value for the Reset signal is zero. It is asserted as soon as the first bit is received by the data interface. The Rset process generates the Reset signal.

Stellenbosch University http://scholar.sun.ac.za


```
Rset: process

begin
  wait until Data_Strb'event and Data_Strb = '1';
  Reset <= '1';
end process Rset;
```

Table 3.5 is the output table for the Data_Valid_Process process.

Table 3.5 Output of the Data_Valid_Process process

Reset	Data_Valid	Data_Read_Flag	Data_Valid_Flag
0	x	x	0
1	0	0	No Change
1	1	0	1
1	0	1	0
1	1	1	Invalid

When Reset is asserted and both Data_Valid and Data_Read_Flag are not asserted then Data_Valid_Flag remains unchanged.

The Data_Overflow process generates the DataError signal.

```
Data_Overflow: process

begin
  wait until Data_Valid'event and Data_Valid = '1';
  if Data_Valid_Flag = '1' then
    DataError <= '1';
  else
    DataError <= '0';
```

Stellenbosch University http://scholar.sun.ac.za


```
  end if;
end process Data_Overflow;
```

The data received by the data interface must be read from the AOMB 1 register before the next 32 bits arrive. If this does not happen then those 32 bits are lost. Thus if Data_Valid is asserted while Data_Valid_Flag is still asserted, then the 32 bits will be lost and DataError is asserted.

The Data_IO process determines when data is read and written to the add-on interface registers of the S5933.

```
Data_IO: process

begin
  wait until clk'event and clk = '1';
  if MBox_E_F = '0' or Fin_Read = '0' then
    Data_Read_Flag <= '0';

     Read AMBEF state machine

  elsif MBox_E_F = '1' and Fin_Read = '1' and Data_Valid_Flag = '1' then

    Write AOMB 1 state machine

  else
    Data_Read_Flag <= '0';
    Selct <= '1';
    WR <= '1';
    RD <= '1';
    Be <= "1111";
    ADR <= "00000";
```

Stellenbosch University http://scholar.sun.ac.za


```
      Read_Flag <= '0';
      Write_Flag <= '0';
      Fin_Read <= '0';
    end if;
  end process Data_IO;
```

Figure 3.10 shows the flow diagram of this process.

If the data have not been read from AOMB 1, indicated by Mbox_E_F = 0, or data was written to the AOMB 1, indicated by Fin_Read = 0 then the Data_Read_Flag is cleared and the AMBEF is read. Also if the previous data were read from AOMB 1, Mbox_E_F = 1, and if AMBEF was read, Fin_Read = 1, and 32 bits are available, Data_Fin_Flag = 1, then the data is written to AOMB 1.

A complete code listing is provided in appendix A1 and the pinout of the Mailbox interface is provided in appendix A2.

Stellenbosch University http://scholar.sun.ac.za

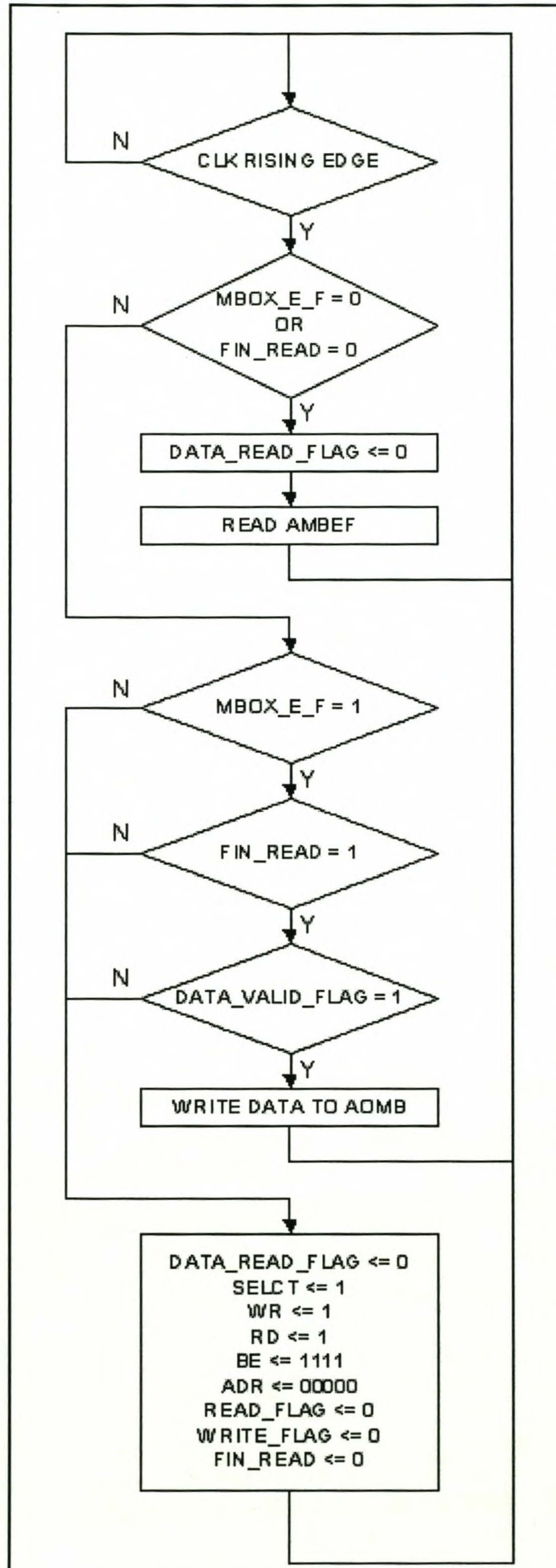


Figure 3.10 Data_IO process flow diagram

Stellenbosch University http://scholar.sun.ac.za

d Simulations

Max+plus II reported the maximum frequencies for Data_Strb and Clk as follow: Clk: 38.61MHz and Data_Strb: 59.52MHz.

Figure 3.11 shows the simulation of the read cycle of the AMBEF register and the write cycle to the AOMB register.

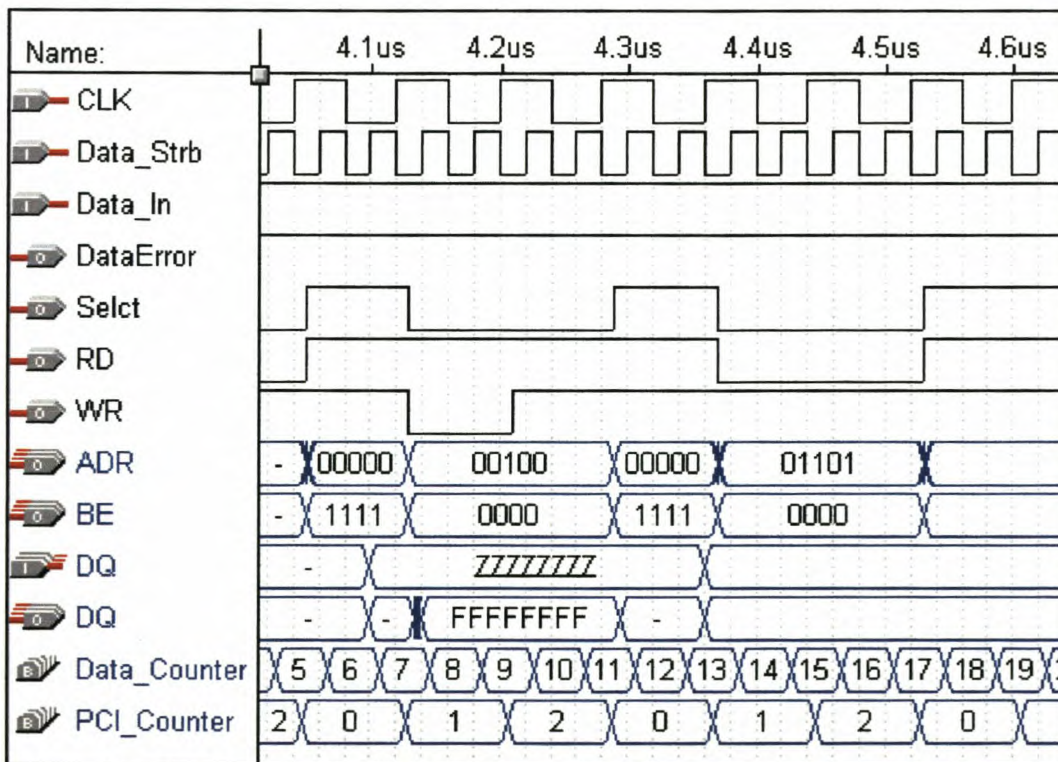


Figure 3.11 Mailbox implementation read and write cycle

For the write cycle Selct and WR are driven low and for the read cycle Selct and RD are driven low.

Figure 3.12 shows that the implementation waits until the data currently in the AOMB is read (bits 16 to 19 of the AMBEF register must be zero) before the new data is written to the register.

Stellenbosch University http://scholar.sun.ac.za

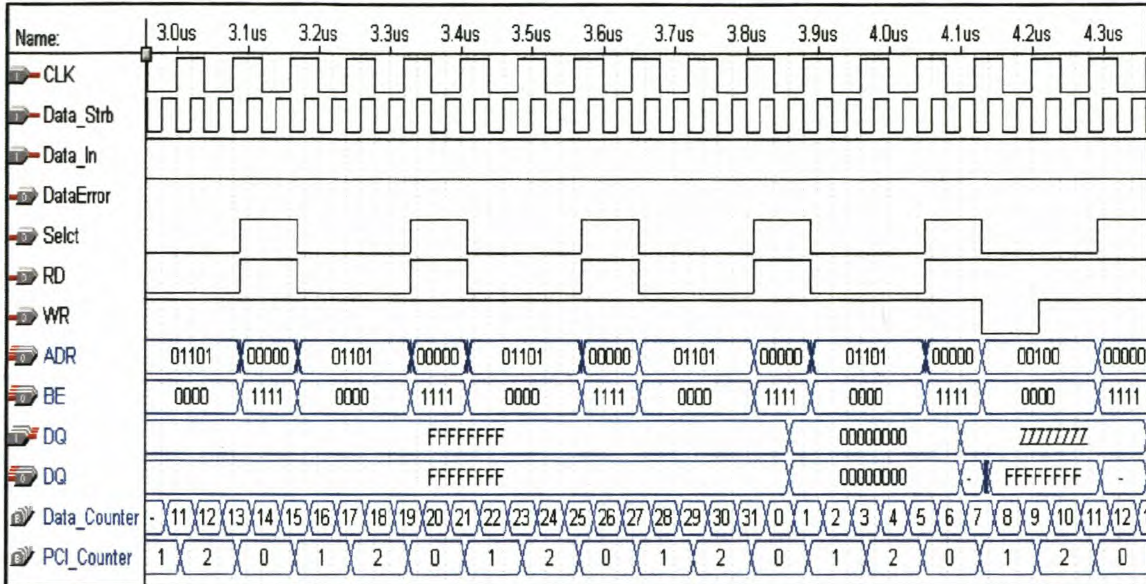


Figure 3.12 Wait until data is read from AOMB

If the data is not read from AOMB before the new data is available then the new data is lost and Data_Error is asserted, as shown in figure 3.13.

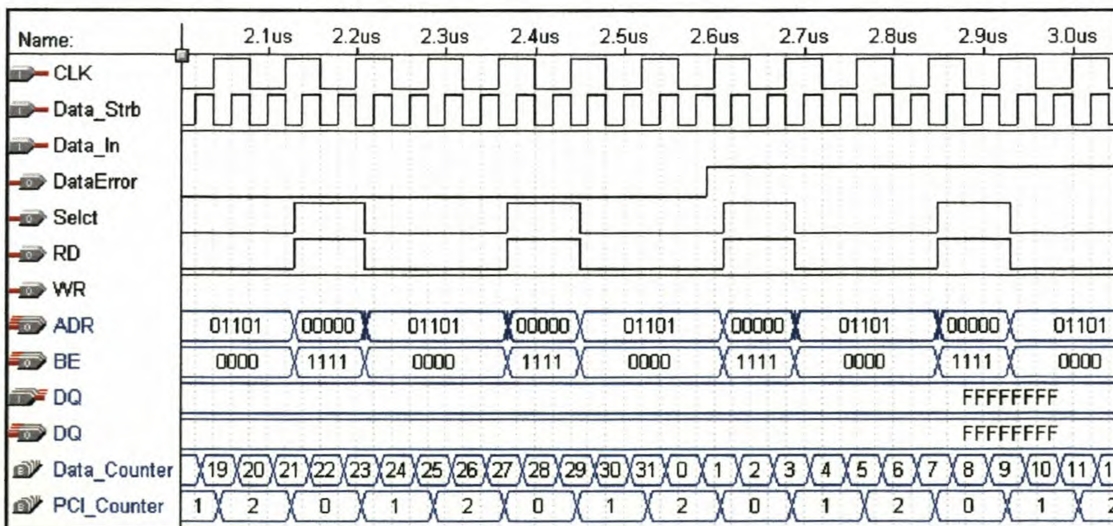


Figure 3.13 Data is lost and Data_Error is asserted

Stellenbosch University http://scholar.sun.ac.za

3.1.2.2 FIFO implementation

For the FIFO implementation the mailbox registers are used in conjunction with the FIFO register. With the mailbox implementation only one 32-bit register is available to transfer the captured data to the PCI bus. The FIFO provides eight 32-bit registers, which can be used as a buffer. If the PC was unable to read the data from the register before the next one arrives the data is not lost. This added advantage also comes with a cost, the logic needed to implement the interface is more complex.

For the FIFO implementation the mailbox registers are used in conjunction with the FIFO register. As with the mailbox implementation the software that runs on the PC polls the MBEF register to determine if data is available to be stored. With the mailbox implementation the register had to be polled after each time that data were read from the IMB 1 of the PCI bus interface. With the FIFO implementation the MBEF still needs to be polled, but only after four 32-bit reads have been performed. This reduces the amount of time needed before the next data can be read.

Figure 3.14 shows the high level flow diagram for the FIFO implementation.

The AIMB 1 is used as a way to switch the interface on and off. After the data have been received the AIMB 1 is checked. If the interface has been turned on then the data is stored in the FIFO. This is done four times. Thereafter a write is done to the AOMB 1 register. The data written to AOMB 1 are never read by the PC it is only done to set the appropriate bits in the MBEF, so that the software will start reading the data from the FIFO.

The following sections lists the external input signal and the internal signals used to realise the complete interface. Then the implementation in VHDL is discussed.

Stellenbosch University http://scholar.sun.ac.za

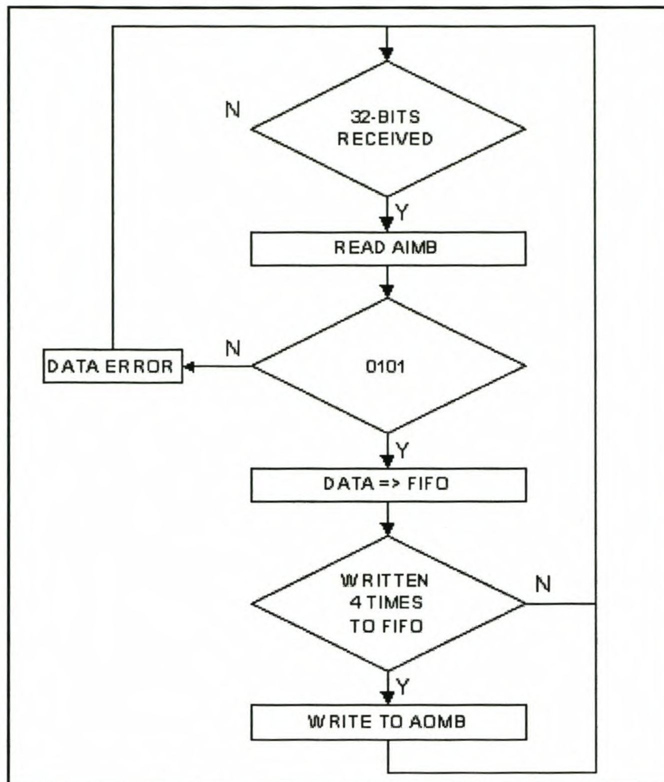


Figure 3.14 FIFO implementation high level flow diagram

a External inputs

Table 3.6 lists the signals used to interface with the S5933.

Table 3.6 FIFO implementation S5933 interface signals

Name	Input/ Output	Type	Description
CLK	in	std_logic	Buffered PCI clock.
DQ	inout	std_logic_vector(31 downto 0)	S5933 data path.

Stellenbosch University http://scholar.sun.ac.za

ADR	out	std_logic_vector(6 downto 2)	S5933 address lines.
BE	out	std_logic_vector(3 downto 0)	S5933 byte enable lines.
Selct	out	std_logic	S5933 select line.
RD	out	std_logic	S5933 read strobe.
WR	out	std_logic	S5933 write strobe.
Status	out	std_logic	Indicates if data is being captured
WRFULL	in	std_logic	Indicates whether the add-on -to-PCI bus FIFO is able to accept more data.
RDFIFO	out	std_logic	S5933 FIFO read
WRFIFO	out	std_logic	S5933 FIFO write

b Signals

Table 3.7 gives a list of all the signals used to implement the mailbox interface as well as the description.

Table 3.7 FIFO implementation signals

Name	**Type**	**Initial value**	**Description**
FIFO_Counter	integer range 0 to 3	n/a	Increments each time data is written to the FIFO register.
Data_Valid_Sync_Reset	std_logic	n/a	Used as a clear signal.
Data_Valid_Async_Out	std_logic	n/a	Used to generate the sync signal.

Stellenbosch University http://scholar.sun.ac.za

Data_Valid_Sync	std_logic	n/a	Sync version of Data_Valid.
Data_Read	std_logic_vector(31 downto 0)	n/a	Used to store data read from the AMBEF.
Read_Flag	std_logic	0	Set if data is read from DQ.
Write_Flag	std_logic	0	Set if data is written to DQ.
DQ_FIFO	std_logic	n/a	Set when a FIFO write transactions is performed.
DQ_AMBEF	std_logic	n/a	Set when a mailbox status register read transaction is performed.
DQ_AOMB	std_logic	n/a	Set when a mailbox write transaction is performed.
Write_FIFO	std_logic	n/a	Set to write data to FIFO.
Read_FIFO	std_logic	n/a	Set to read data from FIFO.
FIFO_Ready	std_logic	n/a	Set when 4 * 32-bit data have been received.
FIFO_Ready_Sync	std_logic	n/a	FIFO_Ready synced to Clk.
FIFO_Ready_Sync _Reset	std_logic	n/a	Used as a clear signal.
FIFO_Ready_ Async_Out	std_logic	n/a	Used to generate the sync signal.
Reset	std_logic	n/a	Sets state machines to a default known state.
Not_FIFO	std_logic	n/a	Set when finished accessing the FIFO.

Stellenbosch University http://scholar.sun.ac.za

Data_Fin_Read	std_logic	n/a	Set when finished checking AIMB.
FIFOState_Type	FIFO_DefaultState, FIFO_s1, FIFO_s2, FIFO_s3, FIFO_s4	n/a	Defines the possible states of the state machine.
FIFOState	FIFOState_Type	n/a	The current state of the state machine.
AIMBState_Type	AIMB_DefaultState, AIMB_s1, AIMB_s2	n/a	Defines the possible states of the state machine.
AIMBState	AIMBState_Type	n/a	The current state of the state machine.
AOMBState_Type	AOMB_DefaultState, AOMB_s1, AOMB_s2	n/a	Defines the possible states of the state machine.
AOMBState	AOMBState_Type	n/a	The current state of the state machine.
AIMB_Selct	std_logic	n/a	Used to assert Select.
AIMB_RD	std_logic	n/a	Used to assert RD.
AIMB_WR	std_logic	n/a	Used to assert WR.
AIMB_Be	std_logic	n/a	Used to set Be.
AIMB_ADR	std_logic	n/a	Used to set Adr.
AOMB_Selct	std_logic	n/a	Used to assert Select.
AOMB_RD	std_logic	n/a	Used to assert RD.
AOMB_WR	std_logic	n/a	Used to assert WR.
AOMB_Be	std_logic	n/a	Used to set Be.
AOMB_ADR	std_logic	n/a	Used to set Adr.
Make_Z	std_logic	n/a	Tri-states DQ.

Stellenbosch University http://scholar.sun.ac.za

c VHDL implementation

Data_Valid is set by the data interface as soon as 32 bits have been received. The data interface's timing is provided by the Data_Clk and the FIFO interface's by Clk, which is the PCI clock. In order for the FIFO interface to be able to use the Data_Valid signal reliably, it is necessary for the Data_Valid signal to be synchronised to the PCI clock.

Figure 3.15 shows the logic implementation that synchronises an incoming signal Async_In with a clock signal Clk.

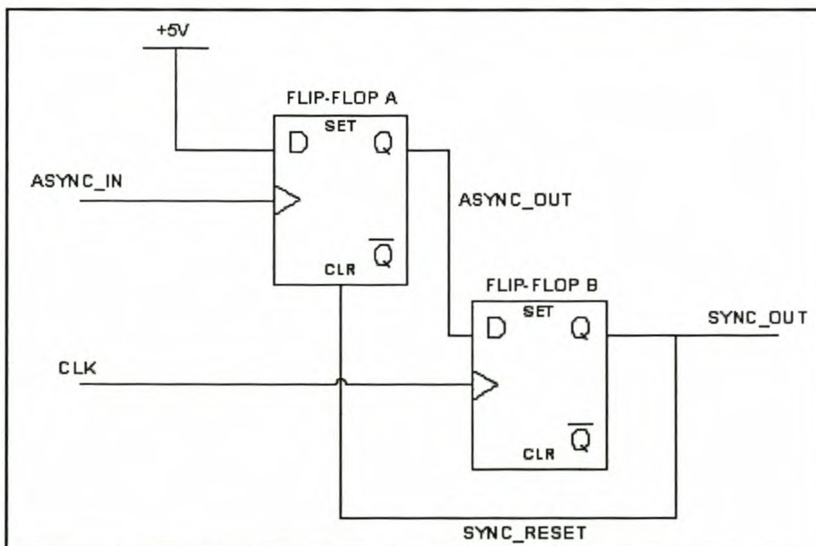


Figure 3.15 Synchronise a signal to a clock

As shown in the timing diagram in figure 3.15, as soon as Async_In is asserted, the output of the positive-edge-triggered D flip-flop A, Async_Out, is driven high as well. Asyn_Out will stay in this state until the clear signal is asserted. The output of flip-flop B, Sync_Out will only be asserted on a positive-edge of clk if Async_Out is asserted. Sync_Out is also directly connected to the clear signal of flip-flop A. When Sync_Out is asserted flip-flop A is cleared.

The VHDL implementation for the logic in figure 3.15 is as follow:

Stellenbosch University http://scholar.sun.ac.za


```
Flip_Flop_A: process(Sync_Reset, Async_In)

begin
  if Sync_Reset = '0' then
    Async_Out <= '0';
  elsif Async_In'event and Async_In = '1' then
    Async_Out <= '1';
  end if;
end process Flip_Flop_A;

Flip_Flop_B: process

begin
  wait until clk'event and clk = '1';
  if Async_Out = '1' then
    Sync_Reset <= '1';
    Sync_Out <= '1';
  else
    Sync_Reset <= '0';
    Sync_Out <= '0';
  end if;
end process Flip_Flop_B;
```

Flip-flop A is implemented in the Flip_Flop_A process, while flip-flop B in the Flip_Flop_B process. A change in the signal in brackets after the declaration of the process will trigger that process. The Flip_Flop_A process is triggered by a change in Sync_Reset and Async_In.

This method of synchronisation is used to synchronise Data_Valid with Clk. The synchronised version of Data_Valid is Data_Valid_Sync.

Stellenbosch University http://scholar.sun.ac.za

The next step in the process is to check the AIMB 1 to find out if the command has been given to continue capturing or wait. Figure 3.16 shows the state diagram of the AIMB state machine that reads the AIMB 1.

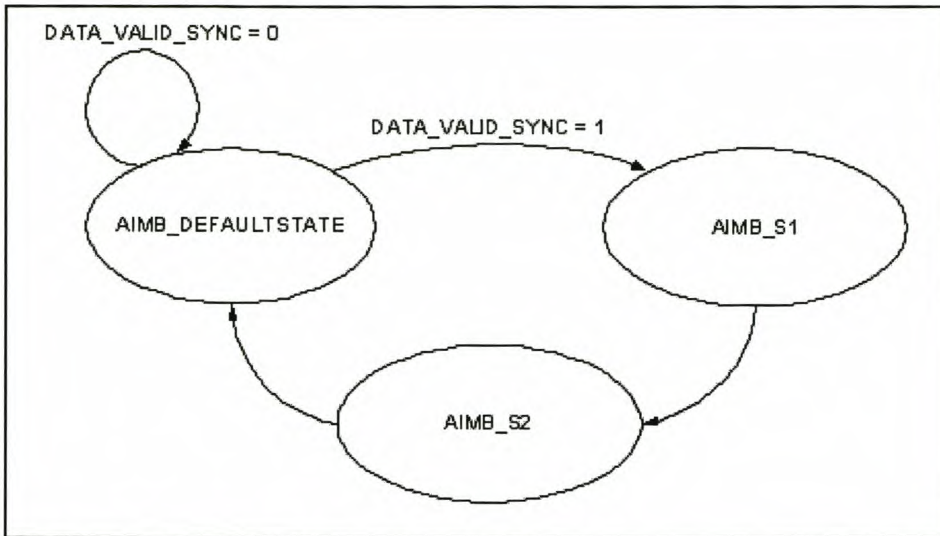


Figure 3.16 AIMB read state diagram

This state diagram is implemented in the AIMB_State_Machine process. AIMBState is declared of type AIMBState_Type. This means that the state machine can have the following states: AIMB_DefaultState, AIMB_s1 and AIMB_s2.

It is necessary to ensure that all state machines are initialised in a known state. The reset signal is used in all the state machines to ensure that they will initialise in the default state. The Res process is used to generate the Reset signal.

```
res: process

begin
  wait until Data_Strb'event and Data_Strb = '1';
  reset <= '1';
end process res;
```

As soon as the first data bit is received the Reset signal is asserted.

Stellenbosch University http://scholar.sun.ac.za

The AIMB_State_Machine process uses the Reset signal to set the state machine to the AIMB_DefaultState at initialisation. As soon as Data_Valid_Sync is asserted the state changes to AIMB_s1, then AIMB_s2 and then back to AIMB_DefaultState. The three states are necessary to complete a read cycle to the S5933's add-on interface register.

The VHDL implementation for the AIMB_State_Machine process is as follow:

```
AIMB_State_Machine: process(Clk)

begin
 if reset = '0' then
  AIMBState <= AIMB_DefaultState;
 elsif (Clk'event and Clk = '1') then
  case AIMBState is
        when AIMB_DefaultState =>
         if Data_Valid_Sync = '1' then
          AIMBState <= AIMB_s1;
         else
          AIMBState <= AIMB_DefaultState;
         end if;
        when AIMB_s1 =>
         AIMBState <= AIMB_s2;
        when AIMB_s2 =>
         AIMBState <= AIMB_DefaultState;
  end case;
 end if;
end process AIMB_State_Machine;
```

The AIMB_Signals process sets the signals used to generate the external outputs. Table 3.8 show the output table for each state.

Stellenbosch University http://scholar.sun.ac.za

Table 3.8 AIMB_Signals process outputs

	AIMB_DefaultState	AIMB_s1	AIMB_s2
AIMB_Selct	1	0	0
AIMB_WR	1	1	1
AIMB_RD	1	0	0
AIMB_Be	1	0	0
AIMB_ADR	0	1	1
DQ_AIMB	0	0	1
Data_Fin_Read	0	0	1

The VHDL implementation of table 3.8:

```
AIMB_Signals: process

begin
 wait until Clk'event and Clk = '1';
 case AIMBState is
  when AIMB_DefaultState =>
   AIMB_Selct <= '1';
   AIMB_WR <= '1';
   AIMB_RD <= '1';
   AIMB_BE <= '1';
   AIMB_ADR <= '0';
   DQ_AIMB <= '0';
   Data_Fin_Read <= '0';
  when AIMB_s1 =>
   AIMB_Selct <= '0';
   AIMB_WR <= '1';
   AIMB_RD <= '0';
```

Stellenbosch University http://scholar.sun.ac.za


```
        AIMB_BE <= '0';
        AIMB_ADR <= '1';
        DQ_AIMB <= '0';
        Data_Fin_Read <= '0';
      when AIMB_s2 =>
        AIMB_Selct <= '0';
        AIMB_WR <= '1';
        AIMB_RD <= '0';
        AIMB_BE <= '0';
        AIMB_ADR <= '1';
        DQ_AIMB <= '1';
        Data_Fin_Read <= '1';
      end case;
    end process AIMB_Signals;
```

The AIMB 1 is used as a control register for the FIFO implementation. It provides a way for the software to enable and disable the capturing of data. The FIFO_State_Machine process controls the state machine that writes data to the FIFO. Figure 3.17 shows the state diagram for the FIFO state machine.

The state machine has the following states: FIFO_DefaultState, FIFO_s1, FIFO_s2, FIFO_s3, FIFO_s4. Reset is used to initialise the state machine to the default state, FIFO_DefaultState. The state changes to FIFO_s1 as soon as Data_Fin_Read is asserted. The content that was read from AIMB 1 determines the next state. If bits 0 to 3 are equal to '0101' then the next state will be FIFO_s2, if not the state is returned to FIFO_DefaultState. FIFO_s2, FIFO_s3 and FIFO_s4 are used to write data to the FIFO register. Once the asynchronous write cycle has completed the state is set back to FIFO_DefaultState.

Stellenbosch University http://scholar.sun.ac.za

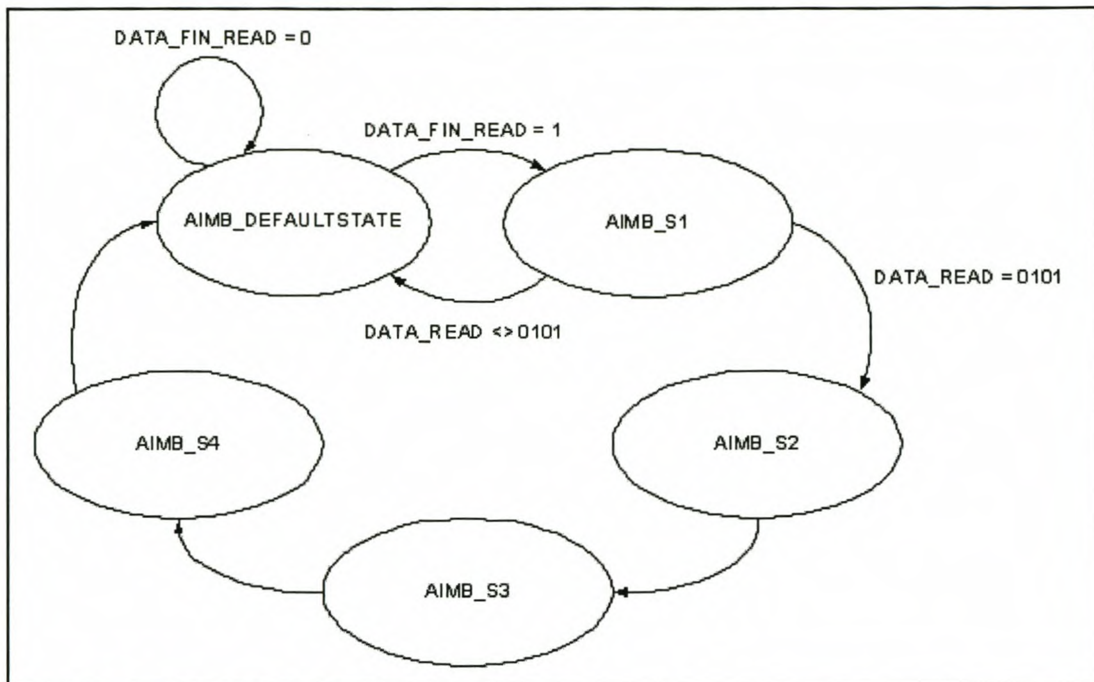


Figure 3.17 FIFO state machine state diagram

The VHDL implementation for FIFO_State_Machine is as follow:

```
FIFO_State_Machine: process(Clk)

begin
 if reset = '0' then
  FIFOState <= FIFO_DefaultState;
 elsif (Clk'event and Clk = '1') then
  case FIFOState is
        when FIFO_DefaultState =>
         if Data_Fin_Read = '1' then
          FIFOState <= FIFO_s1;
         else
          FIFOState <= FIFO_DefaultState;
         end if;
        when FIFO_s1 =>
         if (Data_Read(0) = '1') and (Data_Read(1) = '0') and
```

Stellenbosch University http://scholar.sun.ac.za


```
          (Data_Read(2) = '1') and (Data_Read(3) = '0') then
          FIFOState <= FIFO_s2;
          Status <='0';
        else
          FIFOState <= FIFO_DefaultState;
          Status <='1';
        end if;
      when FIFO_s2 =>
        FIFOState <= FIFO_s3;
      when FIFO_s3 =>
        FIFOState <= FIFO_s4;
      when FIFO_s4 =>
        FIFOState <= FIFO_DefaultState;
    end case;
  end if;
end process FIFO_State_Machine;
```

The FIFO_Signals process sets the signals used to generate the external outputs. Table 3.9 show the output table for each state.

Table 3.9 FIFO_Signals process outputs

	FIFO_DefaultState	FIFO_s1	FIFO_s2	FIFO_s3	FIFO_s4
DQ_FIFO	0	0	1	1	1
Read_FIFO	0	0	0	0	0
Write_FIFO	0	0	0	1	0
Not_FIFO	1	1	1	1	0

The VHDL implementation of table 3.9:

```
FIFO_Signals: process
```

Stellenbosch University http://scholar.sun.ac.za


```
begin
  wait until Clk'event and Clk = '1';
  case FIFOState is
    when FIFO_DefaultState =>
      DQ_FIFO <= '0';
      Read_FIFO <= '0';
      Write_FIFO <= '0';
      Not_FIFO <= '1';
    when FIFO_s1 =>
      DQ_FIFO <= '0';
      Read_FIFO <= '0';
      Write_FIFO <= '0';
      Not_FIFO <= '1';
    when FIFO_s2 =>
      DQ_FIFO <= '1';
      Read_FIFO <= '0';
      Write_FIFO <= '0';
      Not_FIFO <= '1';
    when FIFO_s3 =>
      DQ_FIFO <= '1';
      Read_FIFO <= '0';
      Write_FIFO <= '1';
      Not_FIFO <= '1';
    when FIFO_s4 =>
      DQ_FIFO <= '1';
      Read_FIFO <= '0';
      Write_FIFO <= '0';
      Not_FIFO <= '0';
  end case;
end process FIFO_Signals;
```

Stellenbosch University http://scholar.sun.ac.za

When the data was written to the FIFO register, Not_FIFO is set to '0'. Not_FIFO needs to be counted to determine when 4 FIFO writes have occurred. The Not_FIFO_Counter process counts the number of writes:

```
Not_FIFO_Counter: process

begin
  wait until Not_FIFO'event and Not_FIFO = '0';
  FIFO_Counter <= FIFO_Counter + 1;
end process Not_FIFO_Counter;
```

FIFO_Counter is defined with a range between 0 and 3. The Set_FIFO_Ready process sets the FIFO_Ready signal to indicate that 4 writes have occurred.

```
Set_FIFO_Ready: process

begin
  if FIFO_Counter = 0 then
    FIFO_Ready <= '1';
  else
    FIFO_Ready <= '0';
  end if;
end process Set_FIFO_Ready;
```

Since the accesses to the FIFO register is asynchronous FIFO_Ready needs to be Synchronised to the PCI bus clock, Clk. The same method as to synchronise Data_Valid is used to synchronise FIFO_Ready with Clk. The synchronised version of FIFO_Ready is FIFO_Ready_Sync.

To indicate to the software that the data are ready to be read, a write to the AOMB 1 must be done. The software monitors the MBEF register on the PCI interface for

Stellenbosch University http://scholar.sun.ac.za

changes made to the AOMB 1 on the add-on interface. The data written to the AOMB 1 is not important only that data has been placed in that register. Figure 3.18 shows the state diagram of the state machine that writes data to the AOMB 1 register.

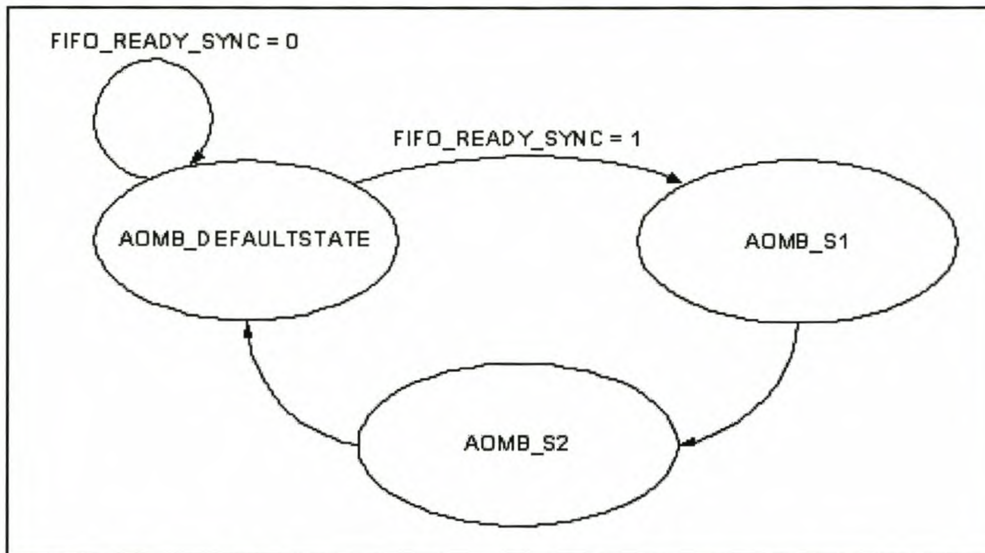


Figure 3.18 FIFO implementation AOMB write state diagram

As soon as FIFO_Ready_Sync is set then a write to the AOMB 1 register is done. The state machine is implemented in the AOMB_State_Machine process and has the following states: AOMB_DefaultState, AOMB_s1 and AOMB_s2. Reset is used to set the state to AOMB_DefaultState when initialised. When FIFO_Ready_Sync is asserted the state changes to AOMB_s1 and then AOMB_s2, which controls the write to the AOMB 1 register. When the write cycle has completed the state returns to AOMB_DefaultState.

The VHDL implementation of the state diagram in figure 3.18 is as follow:

```
AOMB_State_Machine: process(Clk)

begin
  if reset = '0' then
```

Stellenbosch University http://scholar.sun.ac.za


```
      AOMBState <= AOMB_DefaultState;
    elsif (Clk'event and Clk = '1') then
      case AOMBState is
            when AOMB_DefaultState =>
              if FIFO_Ready_Sync = '1' then
                AOMBState <= AOMB_s1;
              else
                AOMBState <= AOMB_DefaultState;
              end if;
            when AOMB_s1 =>
              AOMBState <= AOMB_s2;
            when AOMB_s2 =>
              AOMBState <= AOMB_DefaultState;
      end case;
    end if;
  end process AOMB_State_Machine;
```

The FIFO_Signals process sets the signals used to generate the external outputs. Table 3.10 show the output table for each state.

Table 3.10 AOMB_State_Machine process outputs

	AOMB_DefaultState	**AOMB_s1**	**AOMB_s2**
AOMB_Selct	1	0	0
AOMB_WR	1	0	1
AOMB_RD	1	1	1
AOMB_BE	1	0	0
AOMB_ADR	0	1	1
DQ_AOMB	0	1	1

The VHDL implementation of table 3.10:

Stellenbosch University http://scholar.sun.ac.za


```
AOMB_Signals: process

begin
 wait until Clk'event and Clk = '1';
 case AOMBState is
  when AOMB_DefaultState =>
   AOMB_Selct <= '1';
   AOMB_WR <= '1';
   AOMB_RD <= '1';
   AOMB_BE <= '1';
   AOMB_ADR <= '0';
   DQ_AOMB <= '0';
  when AOMB_s1 =>
   AOMB_Selct <= '0';
   AOMB_WR <= '0';
   AOMB_RD <= '1';
   AOMB_BE <= '0';
   AOMB_ADR <= '1';
   DQ_AOMB <= '1';
  when AOMB_s2 =>
   AOMB_Selct <= '0';
   AOMB_WR <= '1';
   AOMB_RD <= '1';
   AOMB_BE <= '0';
   AOMB_ADR <= '1';
   DQ_AOMB <= '1';
  end case;
end process AOMB_Signals;
```

The following process uses the signals set by the state machines to set the external lines to read and write data the S5933.

Stellenbosch University http://scholar.sun.ac.za

DQ_FIFO, DQ_AIMB and DQ_AOMB determine if data is read or written to DQ. The RW_DQ process uses these signals to set Read_Flag and Write_Flag. These two signals are then used by the Set_DQ process to either read the contents of DQ or write data to DQ.

Table 3.11 show the output table of the RW_DQ process.

Table 3.11 RW_DQ process outputs

	Read_Flag	**Write_Flag**
DQ_FIFO = '1'	0	1
DQ_AIMB = '1'	1	0
DQ_AOMB = '1'	0	1
All others	0	0

The VHDL implementation of table 3.11:

```
RW_DQ: process

begin
 if DQ_FIFO = '1' then
-- Set when a FIFO write transactions is performed
  Read_Flag <= '0';
  Write_Flag <= '1';
 elsif DQ_AIMB = '1' then
-- Set when a mailbox read transaction is performed
  Read_Flag <= '1';
  Write_Flag <= '0';
 elsif DQ_AOMB = '1' then
-- Set when a mailbox write transaction is performed
  Read_Flag <= '0';
```

Stellenbosch University http://scholar.sun.ac.za


```
    Write_Flag <= '1';
  else
    Read_Flag <= '0';
    Write_Flag <= '0';
  end if;
end process RW_DQ;
```

Figure 3.19 show the flow diagram for the Set_DQ process.

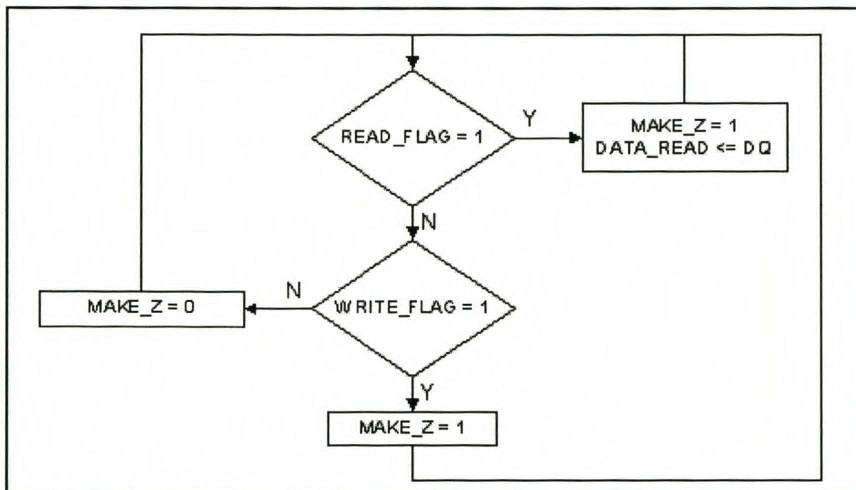


Figure 3.19 Set_DQ flow diagram

The VHDL implementation of figure 3.19:

```
Set_DQ: process

begin
  if Read_Flag = '1' then
    Make_Z <= '1';
    Data_Read <= DQ;
  elsif Write_Flag = '1' then
    Make_Z <= '0';
  else
    Make_Z <= '1';
```

Stellenbosch University http://scholar.sun.ac.za


```
  end if;
end process Set_DQ;
```

The Set_DQ process sets Make_Z that is used by the Set_Z process to either tri-state the I/O pins or set DQ equal to Data. In order to read the data from the add-on register DQ must be set to tri-state. Table 3.12 show the output table of the Set_Z process.

Table 3.12 Set_Z process outputs

	DQ
Make_Z = '1'	tri-state
Make_Z = '0'	Data

The VHDL implementation of table 3.12:

```
Set_Z: process (Make_Z, Data)

begin
  if Make_Z = '1' then
    DQ <= (others =>'Z');
  else
    DQ <= Data;
  end if;
end process Set_Z;
```

WRFIFO and RDFIFO are the external outputs used to write to or read from the FIFO register. The Set_wrfifo_and_rdfifio process uses the Write_FIFO and Read_FIFO signals generated by the FIFO state machine to set these outputs. Table 3.13 show the output table for this process.

Stellenbosch University http://scholar.sun.ac.za

Table 3.13 Set_wrfifo_and_rdfifo process outputs

	WRFIFO	**RDFIFO**
Write_FIFO = '1'	0	1
Read_FIFO = '1'	1	0
All others	1	1

The VHDL implementation for table 3.13:

```
Set_wrfifo_and_rdfifo: process

begin
  if Write_FIFO = '1' then
    WRFIFO <= '0';
    RDFIFO <= '1';
  elsif Read_FIFO = '1' then
    WRFIFO <= '1';
    RDFIFO <= '0';
  else
    WRFIFO <= '1';
    RDFIFO <= '1';
  end if;
end process Set_wrfifo_and_rdfifo;
```

To access the other registers of the add-on interface of the S5933 the following external signals must be set: Selct, BE, ADR, WR and RD. These are set by the following processes respectively: Set_Selct, Set_Be, Set_ADR, Set_WR and Set_RD. Tables 3.14, 3.15, 3.16, 3.17 and 3.18 show the output table for each of these process. After each table the VHDL implementation is given.

Stellenbosch University http://scholar.sun.ac.za

Table 3.14 Set_Selct process outputs

	Selct
AIMB_Selct = '0'	0
AOMB_Selct = '0'	0
All others	1

```
Set_Selct: process

begin
  if AIMB_Selct = '0' then
    Selct <= '0';
  elsif AOMB_Selct = '0' then
    Selct <= '0';
  else
    Selct <= '1';
  end if;
end process Set_Selct;
```

Table 3.15 Set_BE process outputs

	BE
AIMB_Be = '0'	0000
AOMB_Be = '0'	0000
All others	1111

```
Set_BE: process

begin
  if  AIMB_Be = '0' then
    BE <= "0000";
```

Stellenbosch University http://scholar.sun.ac.za


```
  elsif AOMB_BE = '0' then
    BE <= "0000";
  else
    BE <= "1111";
  end if;
end process Set_BE;
```

Table 3.16 Set_ADR process outputs

	ADR
AIMB_ADR = '1'	00000 - address for AIMB 1
AOMB_ADR = '1'	00100 - address for AOMB 1
All others	00000

```
Set_ADR: process

begin
  if AIMB_ADR = '1' then
    ADR <= "00000";
  elsif AOMB_ADR = '1' then
    ADR <= "00100";
  else
    ADR <= "00000";
  end if;
end process Set_ADR;
```

Table 3.17 Set_WR process outputs

	WR
AIMB_WR = '0'	0
AOMB_WR = '0'	0

Stellenbosch University http://scholar.sun.ac.za

All others	1

```
Set_WR: process

begin
  if AIMB_WR = '0' then
    WR <= '0';
  elsif AOMB_WR = '0' then
    WR <= '0';
  else
    WR <= '1';
  end if;
end process Set_WR;
```

Table 3.18 Set_RD process outputs

	RD
AIMB_RD = '0'	0
AOMB_RD = '0'	0
All others	1

```
Set_RD: process

begin
  if AIMB_RD = '0' then
    RD <= '0';
  elsif AOMB_RD = '0' then
    RD <= '0';
  else
    RD <= '1';
  end if;
```

Stellenbosch University http://scholar.sun.ac.za

end process Set_RD;

A complete code listing is provided in appendix B1 and the pinout of the FIFO interface is provided in appendix B2.

d Simulations

Max+plus II reported the maximum frequencies for Data_Strb and Clk as follow: Clk: 42.73MHz and Data_Strb: 45.24MHz.

Figure 3.20 shows the working of the FIFO state machine that writes the received data to the FIFO.

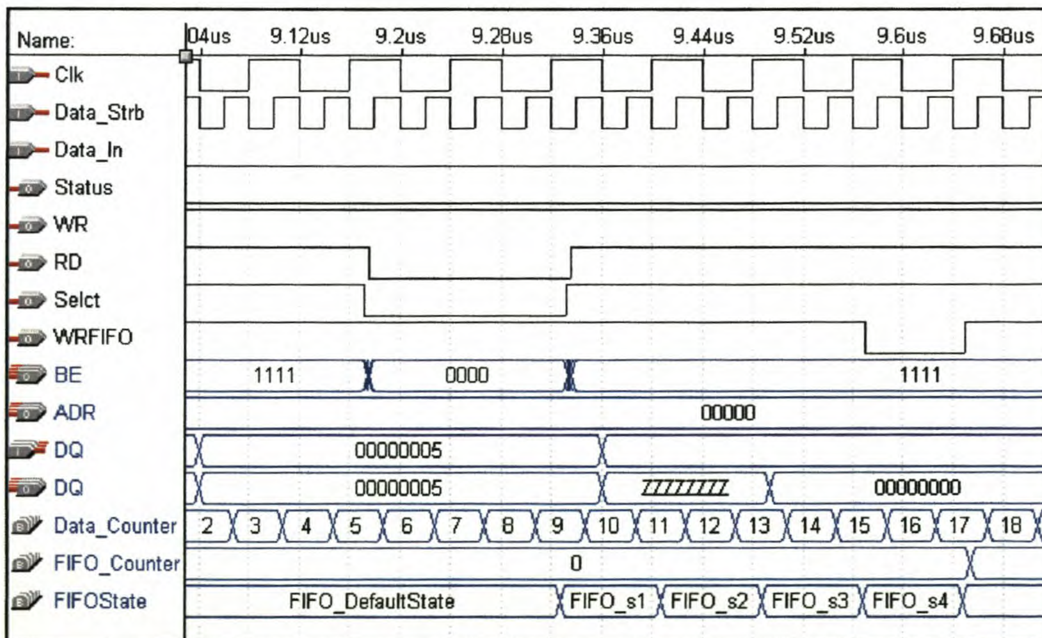


Figure 3.20 FIFO state machine

After the data was read the state changes to FIFO_s1 and if bits 0 to 3 are equal to '1010' then the state machine cycles through the other states to write the data to the FIFO. When the write is complete the state machine returns to its default state.

Figure 3.21 shows that four writes to the FIFO are done before the flag is set to indicate to the software that the data are ready.

Stellenbosch University http://scholar.sun.ac.za

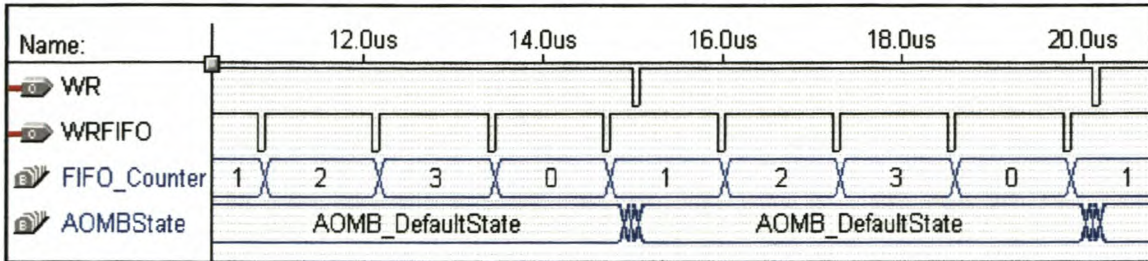


Figure 3.21 FIFO_Counter

The flag is set by performing a write to the AOMB register, which is controlled by the AOMB state machine. Figure 3.22 shows the working of the AOMB state machine. After the write is complete the state machine returns to its default state.

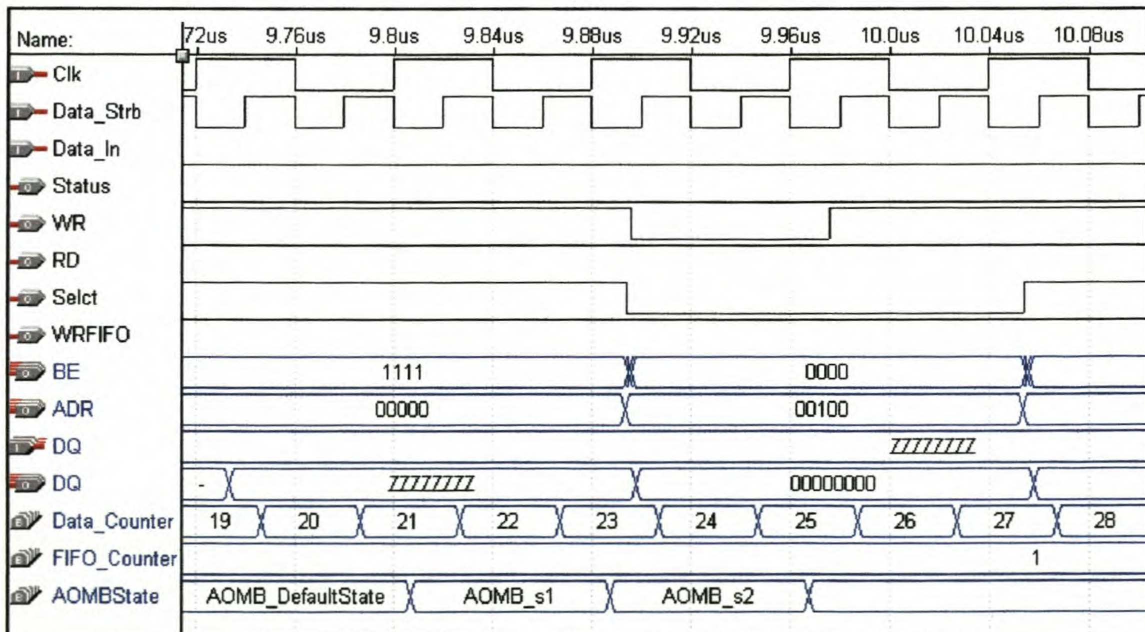


Figure 3.22 AOMB state machine

The AIMB is used to enable and disable the interface. The read from the register is controlled by the AIMB state machine, figure 3.23.

Stellenbosch University http://scholar.sun.ac.za

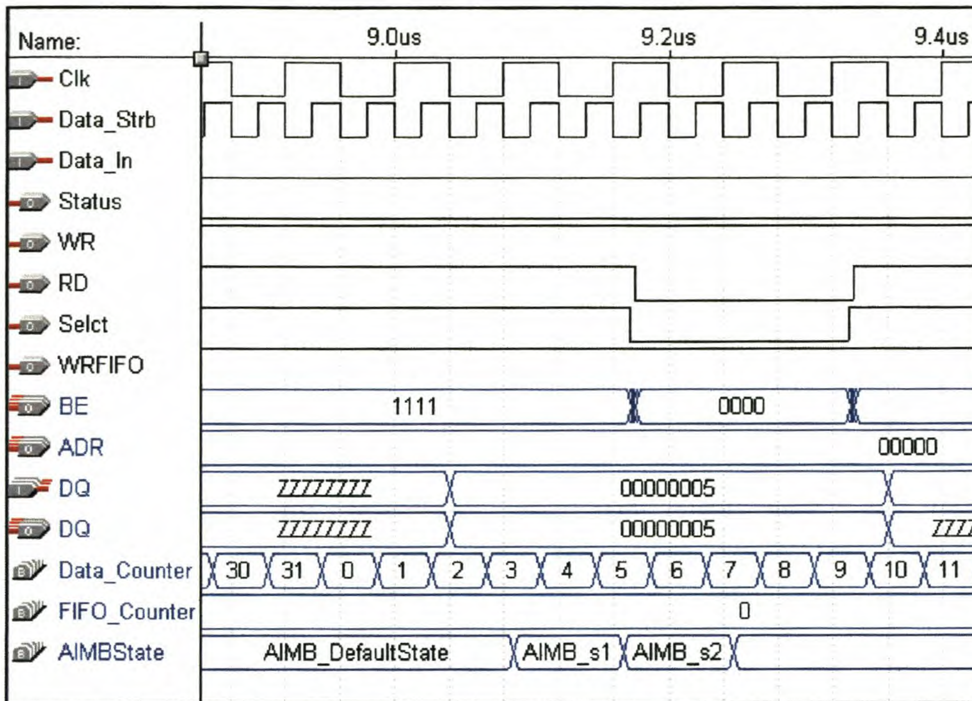


Figure 3.23 AIMB state machine

After the data was read from the AIMB register the state machine returns to its default state. Figure 3.24 show how the interface is enabled.

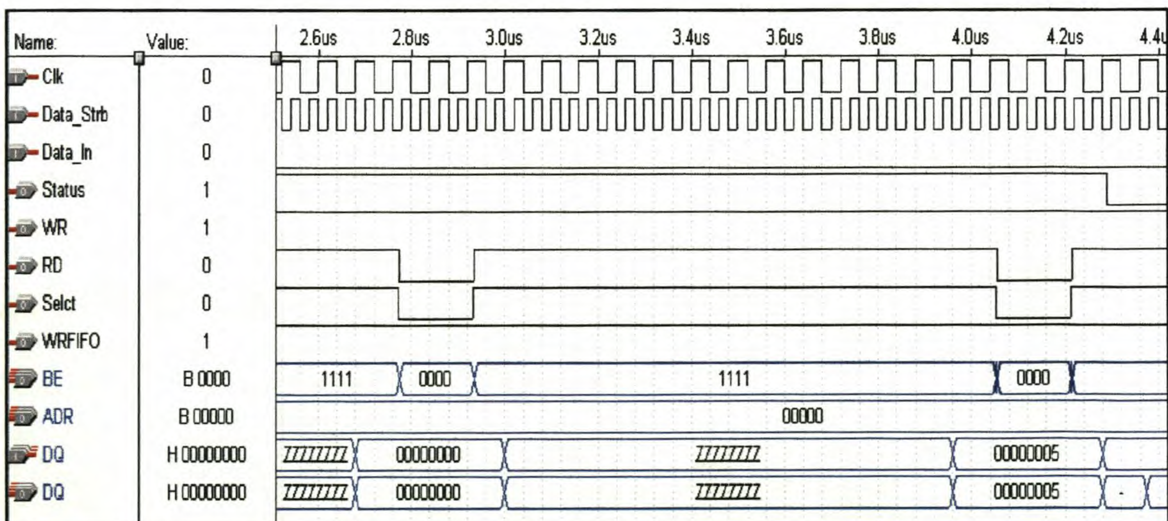


Figure 3.24 Enabling of the FIFO implementation interface

As soon as 05H is read from the AIMB then the interface is enabled.

Stellenbosch University http://scholar.sun.ac.za

3.2 Software

The last step in the prototype development phase was to write the software that reads the data from the PCI card and saves it to the computer's hard disk. During the time when the software was developed hard disks were too slow to write the captured directly to the disk. However an option does exist to write the data at the required speed, and it is called DMA. DMA allows direct data transfers between system memory and I/O boards. However, DMA has two restrictions: no more than 64KB of data can be transferred in one shot, and page boundaries cannot be crossed. Since this was the first attempt to interface with the PCI bus and the lack of experience with DMA access, the choice was made to first read the data into the computer's memory and then after the data has been captured transfer the data to hard disk. Figure 3.25 shows the block diagram of the data path.

Figure 3.25 Data path

Unfortunately due to the amount of memory needed, using the computers memory is not that simple a solution. This is due to the 64KB segment limit in real mode and the complexity of working in protected mode. However, there is a way to circumvent this problem: flat real mode.

3.2.1 Flat Real Mode

3.2.1.1 What is Flat Real Mode?

The 386+ processor has a few different processor modes,
most important are:

Stellenbosch University http://scholar.sun.ac.za

- Real Mode
- Protected Mode
- V86 Mode

The advantage of Protected Mode (PM) above Real Mode (RM) is that in PM a flat 4GB of memory is available. So there is no need for segmenting anymore, another advantage is that it is possible to protect code and data in memory, by making it inaccessible for other programs. The first advantage is the reason why lots of demonstration programs are coded in PM. The problem with PM is that it is not easily possible to use DOS or the BIOS.

But by fooling the processor a variant of RM can be set up, being the Flat Real Mode (FRM). In fact the processor is still in RM, so the BIOS/DOS can be used, but all segments have a size of 4GB instead of 64KB. Using the 386+ 32-bit registers all the memory can be easily accessed. These two reasons make flat real mode the perfect option for applications that need to access a large amount of memory and interface with hardware using the BIOS.

The 64K segment limits with RM are set in the segment descriptors. By changing these descriptors one can set the limits to much more, up to 4GB. However, in RM it is not possible to set the segment descriptors, this can only be done in PM. So the segment limits can only be changed by switching to PM. Going to PM for this purpose is really simple, it is not necessary to bother about interrupts etc. because they are not needed for setting the limits and can therefor be switched off. They must just be switched back on once finished in PM.

When in FRM, full access is available for up to 4GB of memory, but most of it probably isn't present in the computer, other parts can be occupied by other programs. To know which memory can be used an XMS driver must be used, one is included with DOS, HIMEM.SYS. It is used to allocate, deallocate and lock XMS memory.

Stellenbosch University http://scholar.sun.ac.za

However, there's still one thing, which has to be made sure before using extended memory, the address line 20 (A20) must be enabled. Fortunately HIMEM.SYS includes a function for it, but as always, there are also some difficulties. The problem is that the A20-line has to be enabled all the time. When extended memory is used and DOS is loaded into HMA, the A20-line is turned off every time a new program is executed. If the A20 isn't enabled addresses will be clipped at 1MB.

When finished it is not necessary to return from FRM to RM, for normal RM programs there aren't any difference. The only thing that has to be done is that the allocated XMS memory must be unallocated before leaving the program.

3.2.1.2 Implementation

Due to the nature of the application, C++ with linked assembler routines were used. With this implementation it is possible to do low-level hardware accesses, fast memory writes and still use the high level C++ routines to commit the data to hard disk. The only problem is transferring data between the assembler routines and the main program. The easiest way is to use global variables which are common between the main program as well as the assembler routines. However, this is not a good programming technique. The other way in which data can be passed to the assembler routines is on the stack. When an assembler procedure is called the value of the variable is pushed onto the stack, next is the variables offset. Then the current segment's address is pushed and only then control is passed to the assembler routine. To retrieve the variable the current BP value must be stored as well. This is done by pushing the BP as well. Figure 3.26 shows the state of the stack after the BP has been stored.

Stellenbosch University http://scholar.sun.ac.za

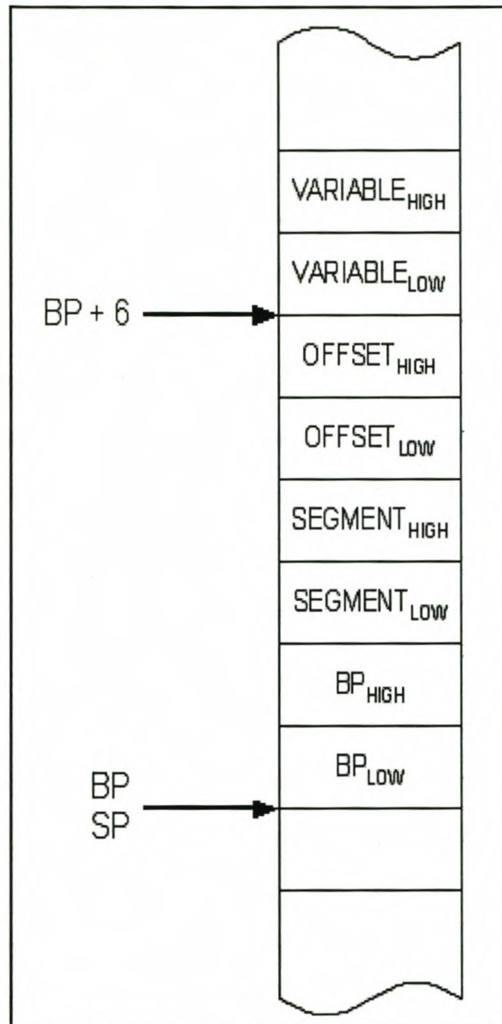


Figure 3.26 The stack

The value of the variable passed to the procedure is now available at the following address: BP+6. Once the assembler routine has finished the original value of BP can be retrieved from the stack. Data from the assembler routine to the main program is passed in the AX register.

Figure 3.27 shows the flow diagram of the program that reads the data from the AMCC's output registers, stores it in the computer's memory and then writes it to the hard disk. The first step is to determine if the correct drivers are loaded and that the processor is in the correct mode. Then the FRM is enabled and the memory is

Stellenbosch University http://scholar.sun.ac.za

allocated. Now the allocated memory is tested to ensure that no errors have occurred and that the capture of the incoming data can continue. Next the incoming data is captured to memory and then stored to hard disk. Finally the allocated memory is unlocked and unallocated. The following sections will explain the function of each of the elements in the diagram.

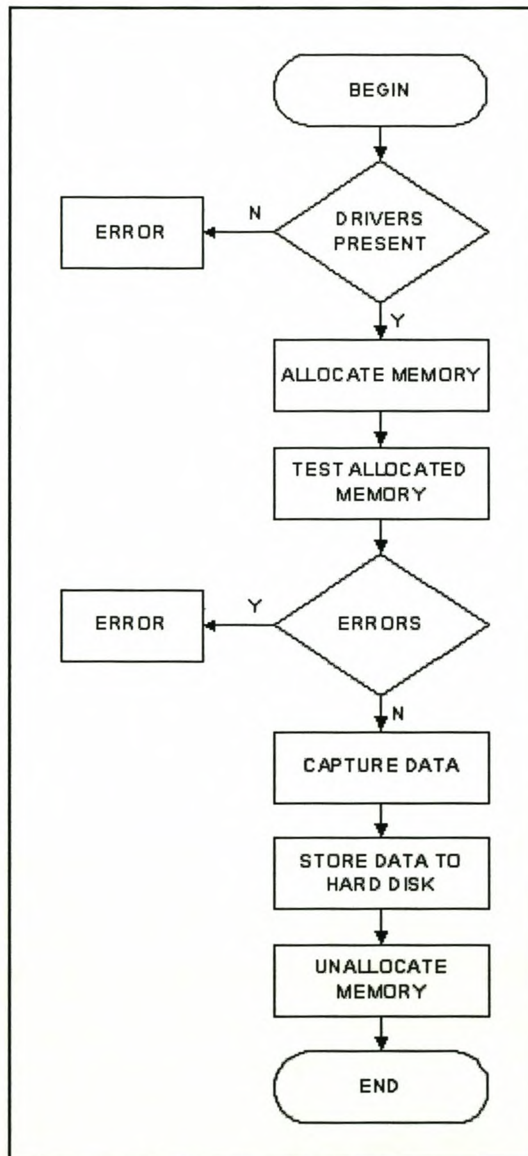


Figure 3.27 Main program flow diagram

Stellenbosch University http://scholar.sun.ac.za

3.2.1.3 Drivers

An XMS driver is needed to access the extended memory. The assembler routine *XMSInit* returns a true if an XMS driver is present, it also determines the version of the driver and obtains the driver address.

The presence of an XMS driver is detected by using INT 2F. INT 2F is a general mechanism for verifying the presence of a TSR and communicating with it. AX determines the function of the interrupt as follow:

AH = identifier of program which is to handle the interrupt

  00h-7Fh reserved for DOS

  B8h-BFh reserved for networks

  C0h-FFh reserved for applications

AL is the function code

The identifier for the XMS driver is 43h. Which means that AH must be 43h.

To determine if the XMS driver is present, AX is set to 4300h. This function will return the following:

AL = 80h : XMS driver installed

AL <> 80h : no driver

The following code checks for the XMS driver and jumps to *XMSFound* if present or to *XMSError* if not.

```
mov  ax,   4300h  ;Verify if there is an XMS driver
int  2Fh
cmp  al,   80h
je   XMSFound
jmp  XMSError
```

The address of the driver is obtained by setting AX = 4310h. This returns the driver entry point: ES:BX. This is then stored in the global variable *XMSDriverAddr*, as shown in the following assembler code.

Stellenbosch University http://scholar.sun.ac.za


```
mov   ax,      4310h  ;Get XMS driver call address
int   2Fh
mov   word ptr _XMSDriverAddr[0],      bx
mov   word ptr _XMSDriverAddr[2],      es
```

If no XMS driver is present the program exit.

3.2.1.4 Processor mode

Next it is necessary to determine the current mode the processor is in. This is done by examining bit 0 of the Control Register 0 (CR0) as done in the assembler routine *InProtected*.

```
@InProtected$qv PROC FAR
        mov   eax,   cr0
        and   al,    1            ;Isolate Protected Mode bit
        ret
@InProtected$qv ENDP
```

CR0 contains system control flags, which control or indicate conditions that apply to the system as a whole, not to an individual task.

Figure 3.28 shows the format of the CR0 register.

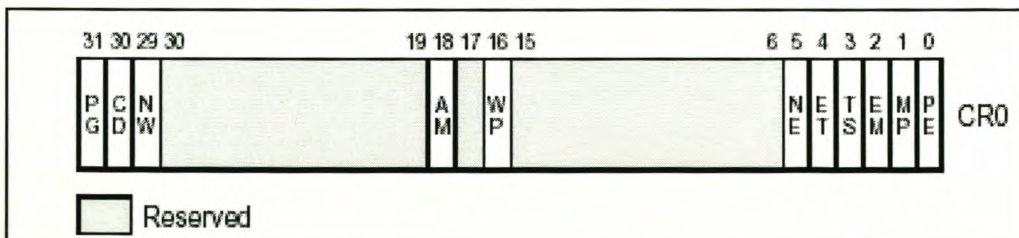


Figure 3.28 The CR0 register

Stellenbosch University http://scholar.sun.ac.za

PE (Protection Enable, bit 0)

Setting PE causes the processor to begin executing in protected mode. Resetting PE returns to real-address mode.

MP (Monitor Coprocessor, bit 1)

MP controls the function of the WAIT instruction, which is used to coordinate a coprocessor.

EM (Emulation, bit 2)

EM indicates whether coprocessor functions are to be emulated.

TS (Task Switched, bit 3)

The processor sets TS with every task switch and tests TS when interpreting coprocessor instructions.

ET (Extension Type, bit 4)

ET indicates the type of coprocessor present in the system (eg 80287 or 80387).

NE (Numeric Error, bit 5)

Enables the native (internal) mechanism for reporting FPU errors when set and enables the PC-style FPU error reporting mechanism when clear.

WP (Write Protect, bit 16)

Inhibits supervisor-level procedures from writing into user-level read-only pages when set.

AM (Alignment Mask, bit 18)

Enables automatic alignment checking when set.

NW (Not Write-through, bit 29)

When the NW and CD flags are clear, write-back (for Pentium processors) or write-through (for Intel486 processors) is enabled for writes that hit the cache and

Stellenbosch University http://scholar.sun.ac.za

invalidation cycles are enabled.

CD (Cache Disable, bit 30)
When the CD and NW flags are clear, caching of memory locations for the whole of physical memory in the processor's internal and external caches are enabled.

PG (Paging, bit 31)
PG indicates whether the processor uses page tables to translate linear addresses into physical addresses.

InProtected returns a true if the processor is in PM.

3.2.1.5 A20 line

Next the A20 line must be enabled. *LocalA20Enable* enables the A20 line.

3.2.1.6 Enabling Flat Real Mode

The last step is to setup the FRM. *EnableFlatReal* loads the ES, FS and GS registers with the 4GB limit selector. This is done by pointing the Global Descriptor Table Register (GDTR) to the GDT declared as follow:

```
GDT DW  0000h,   0000h,   0000h,   0000h
    DW  0FFFFh,  0000h,   9200h,   0FFCFh
```

Where

0000h, 0000h, 0000h, 0000h

declares the "NULL" descriptor.

The format of the segment descriptor

0FFFFh, 0000h, 9200h, 0FFCFh

Stellenbosch University http://scholar.sun.ac.za

is given in figure 3.29.

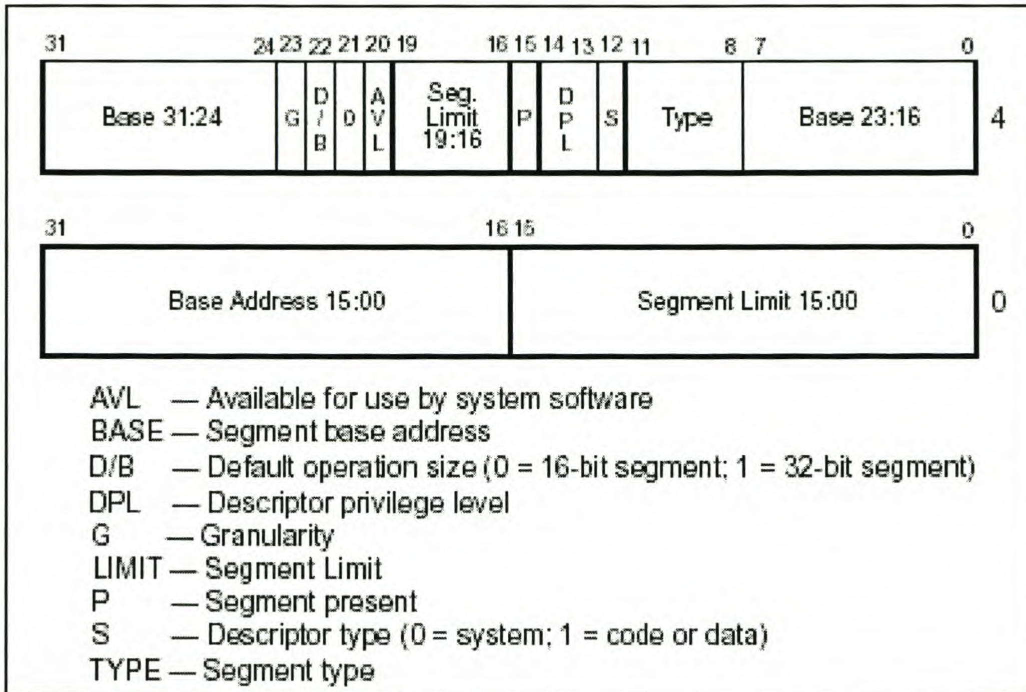


Figure 3.29 Segment descriptor

Segment limit fields

Specifies the size of the segment. The processor puts together the two segment limit fields to form a 20-bit value. The processor interprets the segment limit in one of two ways, depending on the setting of the G (granularity) flag:

- If the granularity flag is clear, the segment size can range from 1 byte to 1 MB, in byte increments.
- If the granularity flag is set, the segment size can range from 4KB to 4GB, in 4KB increments.

Base address fields

Defines the location of byte 0 of the segment within the 4GB linear address space. The processor puts together the three base address fields to form a single 32-bit value. Segment base addresses should be aligned to 16-byte boundaries. Although

Stellenbosch University http://scholar.sun.ac.za

16-byte alignment is not required, this alignment allows programs to maximize performance by aligning code and data on 16-byte boundaries.

Type field

Indicates the segment or gate type and specifies the kinds of access that can be made to the segment and the direction of growth. The interpretation of this field depends on whether the descriptor type flag specifies an application (code or data) descriptor or a system descriptor. The encoding of the type field is different for code, data, and system descriptors.

S (descriptor type) flag

Specifies whether the segment descriptor is for a system segment (S flag is clear) or a code or data segment (S flag is set).

DPL (descriptor privilege level) field

Specifies the privilege level of the segment. The privilege level can range from 0 to 3, with 0 being the most privileged level. The DPL is used to control access to the segment.

P (segment-present) flag

Indicates whether the segment is present in memory (set) or not present (clear). If this flag is clear, the processor generates a segment-not-present exception (#NP) when a segment selector that points to the segment descriptor is loaded into a segment register. Memory management software can use this flag to control which segments are actually loaded into physical memory at a given time. It offers a control in addition to paging for managing virtual memory.

D/B (default operation size/default stack pointer size and/or upper bound) flag

Performs different functions depending on whether the segment descriptor is an executable code segment, an expand-down data segment, or a stack segment. (This flag should always be set to 1 for 32-bit code and data segments and to 0 for 16-bit

Stellenbosch University http://scholar.sun.ac.za

code and data segments.)

- Executable code segment. The flag is called the D flag and it indicates the default length for effective addresses and operands referenced by instructions in the segment. If the flag is set, 32-bit addresses and 32-bit or 8-bit operands are assumed; if it is clear, 16-bit addresses and 16-bit or 8-bit operands are assumed. The instruction prefix 66H can be used to select an operand size other than the default, and the prefix 67H can be used select an address size other than the default.
- Stack segment (data segment pointed to by the SS register). The flag is called the B (big) flag and it specifies the size of the stack pointer used for implicit stack operations (such as pushes, pops, and calls). If the flag is set, a 32-bit stack pointer is used, which is stored in the 32-bit ESP register; if the flag is clear, a 16-bit stack pointer is used, which is stored in the 16-bit SP register. If the stack segment is set up to be an expand-down data segment (described in the next paragraph), the B flag also specifies the upper bound of the stack segment.
- Expand-down data segment. The flag is called the B flag and it specifies the upper bound of the segment. If the flag is set, the upper bound is FFFFFFFFH (4GB); if the flag is clear, the upper bound is FFFFH (64 Kb).

G (granularity) flag

Determines the scaling of the segment limit field. When the granularity flag is clear, the segment limit is interpreted in byte units; when flag is set, the segment limit is interpreted in 4-KB units. (This flag does not affect the granularity of the base address; it is always byte granular.) When the granularity flag is set, the twelve least significant bits of an offset are not tested when checking the offset against the segment limit. For example, when the granularity flag is set, a limit of 0 results in valid offsets from 0 to 4095.

Available and reserved bits

Bit 20 of the second double word of the segment descriptor is available for use by

Stellenbosch University http://scholar.sun.ac.za

system software; bit 21 is reserved and should always be set to 0.

This means that:

Segment Limit 19:00 = FFFFFH

The segment limit is set to 4GB (4KB * FFFFFH = 4GB)

Base Address 31:00 = FF00H 0000H

Type = 0010

Specifies a data segment type with read and write access.

S = 1

Specifies a code or data segment.

DPL = 00

Sets the segment privilege level to the most privileged level.

P = 1

Indicates that the segment is present in memory.

D/B = 1

Indicates 32-bit code or data segments.

G = 1

Indicates that the segment limit should be interpreted in 4-KB units.

To load the GDTR to point to the declared GDT, LGDT is used. LGDT loads the values of the source operand into the GDTR. The source operand specifies a 6-byte memory location that contains the base address (a linear address) and the limit (size of table in bytes) of the GDT. The 48-bit source operand, *LoadGDT*, is declared as follows:

Stellenbosch University http://scholar.sun.ac.za

LoadGDT DW 3 DUP (?)

Then *LoadGDT* is loaded with the size of the GDT.

mov word ptr LoadGDT[0], 16

Next the linear address of the GDT is calculated by first loading the segment address of the GDT.

mov ax, SEG GDT

This is only a 16-bit address and needs to be extended to a 32-bit address.

movzx eax, ax

MOVZX extends the 16-bit value to a 32-bit value by adding a zero-extension.

shl eax, 4

Multiplies the segment address by 16 to calculate the linear address. This is done to calculate the 20-bit memory address.

The offset of the GDT is also loaded and extended to a 32-bit value.

mov bx, OFFSET GDT

movzx ebx, bx

The linear address is then calculated by adding the two values and is stored in *LoadGDT*.

add eax, ebx

mov dword ptr LoadGDT[2], eax

The GDTR is then loaded to point to GDT. The type of *LoadGDT* is overridden and it is referred to a 48-bit register.

lgdt pword ptr LoadGDT

Before switching to RM all maskable hardware interrupts must be disabled

cli

Next the processor is switched to PM by setting the PE flag in CR0.

mov eax, cr0

or eax, 1

mov cr0, eax

After entering PM, the segment registers continue to hold the contents they had in RM. A far jump clears the execution pipe and will reset the CS register.

Next the FS, GS and ES is loaded with the 4GB limit.

Stellenbosch University http://scholar.sun.ac.za


```
mov   fs,   bx
mov   gs,   bx
mov   es,   bx
```

To switch back to RM the PE flag is cleared.

```
and   al,   0FEh
mov   cr0,  eax
```

A far jump is again executed to clear the execution pipe and to load the appropriate base and access rights values in the CS register. The other registers are however still loaded with the 4GB limit. Now up to 4GB can now be addressed in RM.

3.2.2 Allocate memory

XMSAlloc allocates a specified amount of memory and returns a handle to the allocated memory, 0FFFFH is returned if an error occurred while trying to allocate the memory. The amount of memory in bytes is passed to *XMSAlloc* on the stack.

Next *XMSLockMem* is called to lock the allocated memory so that it cannot be moved. *XMSLockMem* also returns the physical address of the memory. The handle to the allocated memory is passed to *XMSLockMem* on the stack. If an error occurs while locking the memory, 0FFFFH is returned as the physical address of the memory. This returned address is stored in the global variable *BufferLinAddr*.

3.2.2.1 Testing allocated memory

TestMode tests the allocated memory. This is done by filling the entire allocated memory and then checking it for errors.

First the start of the allocated memory needs to be retrieved.

```
mov   edi,  [_BufferLinAddr]
```

stores the start of the memory in EDI. The memory will be filled with 32-bit chunks

Stellenbosch University http://scholar.sun.ac.za

of data at a time. To do this the EAX register is loaded with the amount of double words needed to fill the entire allocated memory. This is done by taking the amount of bytes that needs to be filled and dividing by 4 (4 bytes in a double word). For 1MB EAX must be filled with (1024*1024)/4=4000H

```
mov   eax,   4000h
```

Next the memory is filled with the decreasing value of EAX.

```
StoreLoop:
        mov   fs:[edi],   eax   ;Write double word in memory
        add   edi,        4     ;Advance to next location
        dec   eax               ;Repeat for whole memory
        jnz   StoreLoop
```

Then the content of the memory is examined to detect any errors.

```
ReadLoop:
        cmp   fs:[esi],   eax   ;Same as was written?
        jne   ReadError
        add   esi,        4     ;Advance to next location
        dec   eax               ;Repeat for whole memory
        jnz   ReadLoop
        mov   ax,         1
...
        ret
ReadError:
        xor   ax,         ax
...
        ret
```

A true is returned if no errors were detected.

3.2.3 Capture data

Now that the memory is allocated and locked the data can be captured. However before the data can be captured, it must first be determined if a PCI bus is present,

Stellenbosch University http://scholar.sun.ac.za

if the capture card is present and what the addresses of the card are. All this is done by the Capture assembler routine. Using assembler gives a further speed advantage above a high level programming language.

Figure 3.30 shows the high level flow diagram of Capture.

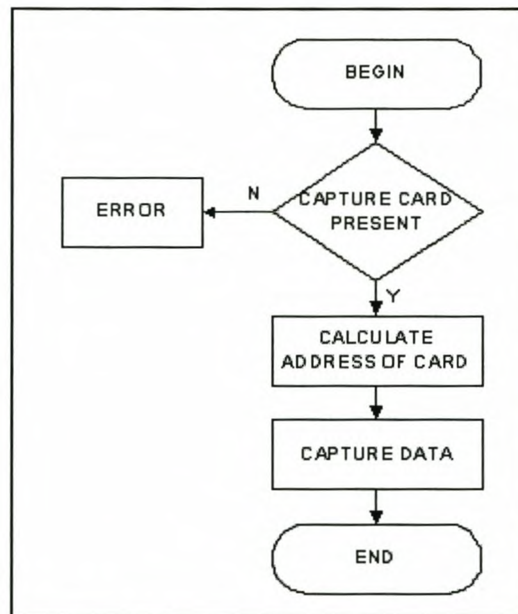


Figure 3.30 *Capture* high level flow diagram

First the checks for the PCI bus and card are done and the base address is obtained. Then the addresses of the registers on the PCI card are calculated and finally the data are captured.

3.2.4 Check PCI device

Figure 3.31 shows the flow diagram of the checks that are done. Information about the PCI devices attached to a PC is obtained by using the function calls of the system's PCI BIOS. These functions are invoked by using INT 1AH in RM.

Stellenbosch University http://scholar.sun.ac.za

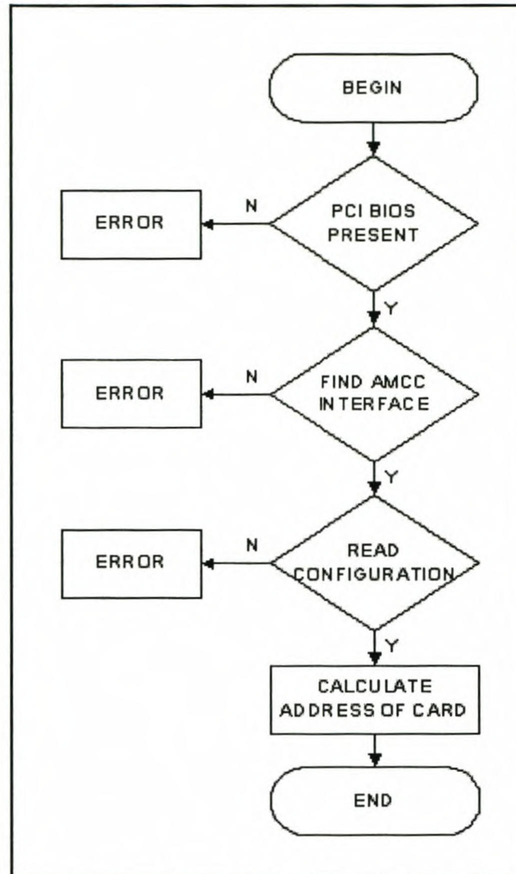


Figure 3.31 PCI device checks

All PCI BIOS function calls have the PCI FUNCTION ID value of B1H loaded in the AH register. The value loaded in the AL register identifies the specific PCI function to be performed. Table 3.19 lists the various functions currently defined for this function call.

Table 3.19 PCI BIOS function calls

Function	AH	AL
Installation Check	B1H	01H
Find PCI Device	B1H	02H
Find PCI Class Code	B1H	03H
Generate Special Cycle	B1H	06H

Stellenbosch University http://scholar.sun.ac.za

Read Configuration Byte	B1H	08H
Read Configuration Word	B1H	09H
Read Configuration DWord	B1H	0AH
Write Configuration Byte	B1H	0BH
Write Configuration Word	B1H	0CH
Write Configuration DWord	B1H	0DH
Get IRQ Routing Information	B1H	0EH
Set PCI IRQ	B1H	0FH

Table 3.20 lists the various return codes.

Table 3.20 Return codes

Return codes	**AH**
Successful	00H
Function Not Supported	81H
Bad Vendor ID	83H
Device Not Found	86H
Bad Register Number	87H
Set Failed	88H
Buffer Too Small	89H
CL	Number of last PCI bus in the system.
CF	Present Status: set = No BIOS present reset = BIOS present

Stellenbosch University http://scholar.sun.ac.za

3.2.4.1 PCI BIOS installation check

The PCI BIOS installation check is done with the Installation Check function call. This function call allows the caller to determine the presence and version of the current PCI BIOS interface.

Inputs:

AH = B1H

AL = 01H

```
stc                     ;Sets CF
mov   ax,   0B101H
int   1Ah
```

CF is set because the function will clear it if the check was successful.

The function returns the following:

AH = Status - SeeTable 3.20

CF = Status - See Table 3.20

EDX = “PCI” = 20494350h (“P”in DL, “C”in DH, etc. There is a “space” character in the high order byte)

EDI = physical address of protected-mode entry point

AL = PCI hardware characteristics

BH = PCI interface level major version

BL = PCI interface level minor version

CL = number of last PCI bus in system

EAX, EBX, ECX, and EDX may be modified

all other flags (except IF) may be modified

For the check to be successful both AH=0 and CF must be clear.

```
jb    Error_1       ;Jumps if CF not clear
cmp   ah,   0       ;Checks if AH = 0
jne   Error_2
```

If an error has occurred a jump is performed to the specific error code routines. The

Stellenbosch University http://scholar.sun.ac.za

error code routines are discussed in 3.2.4.4.

3.2.4.2 Find PCI device

To find the PCI device the Find PCI Device PCI function call is used. This function call allows the caller to determine the location (Bus Number, Device Number and Function Number) for a specific Vendor and Device ID.

Inputs:

AH = B1H

AL = 02H

CX = Device ID (DID)

DX = Vendor ID (VID)

SI = Device Index (0 - N)

For the AMCC card:

VID = 1234H

DID = 5678H

```
stc                        ;Sets the CF
mov   ax,   0B102H
mov   cx,   5678H
mov   dx,   1234H
mov   si,   0
int   1AH
```

SI is set to zero because there is only one device with the specific VID and DID.

The function returns the following:

CF = Status - See Table 3.20

AH = Status - See Table 3.20

BH = Bus Number

BL = Device/Function number (bits 7-3: Device, bits 2-0: Function)

EAX, EBX, ECX, and EDX may be modified all other flags (except IF) may be modified

Stellenbosch University http://scholar.sun.ac.za

For the check to be successful both AH=0 and CF must be clear.

```
jb      Error_3             ;Jumps if CF not clear
cmp     ah,     0           ;Checks if AH = 0
je      Good_1
cmp     ah,     81H         ;Unsupported function
je      Error_4
cmp     ah,     83H         ;Bad vendor ID
je      Error_5
cmp     ah,     86H         ;Device not found
je      Error_6
cmp     ah,     87H         ;Bad PCI register number
je      Error_7
jmp     Error               ;If something went wrong!
```

If an error has occurred a jump is performed to the specific error code routines. The error code routines are discussed in 3.2.4.4.

```
Good_1:
        mov     [BusNum],   bh      ;Stores the Bus Number
        mov     [DevFun],   bl      ;Stores the Dev Func
```

If there were no errors the Bus Number and Device Function are stored.

3.2.4.3 Configuration read

Finally the base address of the PCI card is necessary to calculate the addresses of the registers on the card. The base address is stored in the base address register (BAR) in the configuration register space of every PCI compliant device. The structure of the configuration space is shown in Figure 3.32.

Stellenbosch University http://scholar.sun.ac.za

Byte 3	Byte 2	Byte 1	Byte 0	Address
Device ID		Vendor ID		00h
PCI Status		PCI Command		04h
Class Code			Revision ID	08h
Built-In Self Test	Header Type	Latency Timer	CacheLine Size	0Ch
Base Address Register 0				10h
Base Address Register 1				14h
Base Address Register 2				18h
Base Address Register 3				1Ch
Base Address Register 4				20h
Base Address Register 5				24h
Reserved Space				28h
Reserved Space				2Ch
Expansion ROM Base Address				30h
Reserved Space				34h
Reserved Space				38h
Max. Latency	Min. Grant	Interrupt Pin	Interrupt Line	3Ch

Figure 3.32 Configuration space header

To read the contents of this space the Read Configuration group of function calls are used. The BAR has 32 bits, thus the Read Configuration DWord function call is used.

Inputs:

AH = B1H

AL = 0AH

BH = Bus Number

BL = Device Number

DI = Register Number

As shown in Figure 3.32 the BAR0 is stored in register 10H.

```
stc                                    ;Sets the CF
```

Stellenbosch University http://scholar.sun.ac.za


```
mov   ax,    0B10AH
mov   bh,    [BusNum]
mov   bl,    [DevFun]
mov   di,    10H
int   1AH
```

The function returns the following:

CF = Status - See Table 3.20

ECX = dword read

AH = Status - See Table 3.20

EAX, EBX, ECX, and EDX may be modified

all other flags (except IF) may be modified

For the check to be successful both AH=0 and CF must be clear.

```
jb    Error_8              ;Jumps if CF not clear
cmp   ah,    0             ;Checks if AH = 0
je    Good_2
```

If an error has occurred a jump is performed to the specific error code routines. The error code routines are discussed in 3.2.4.4.

```
cmp   ah,    81H           ;Unsupported function
je    Error_9
cmp   ah,    83H           ;Bad vendor ID
je    Error_10
cmp   ah,    86H           ;Device not found
je    Error_11
cmp   ah,    87H           ;Bad PCI register number
je    Error_12
jmp   Error                ;If something went wrong!
```

If there were no errors the Base Address is stored.

```
Good_2:
```

```
        mov   [BAR0],     ecx            ;Stores the BAR0
```

3.2.4.4 Error code routines

The error codes are generated according to table 3.20.

```
Error:                                   ;Unknown error, something went wrong
        mov   ax,   0FFFFH
        jmp   Fin
Error_1:                                 ;Installation check CF <> 0
        mov   ax,   1
        jmp   Fin
Error_2:                                 ;Installation check AH <> 0
        mov   ax,   2
        jmp   Fin
Error_3:                                 ;Find device CF <> 0
        mov   ax,   3
        jmp   Fin
Error_4:                                 ;Find device: Unsupported function
        mov   ax,   4                    ;81H
        jmp   Fin
Error_5:                                 ;Find device: Bad vendor ID
        mov   ax,   5                    ;83H
        jmp   Fin
Error_6:                                 ;Find device: Device not found
        mov   ax,   6                    86H
        jmp   Fin
Error_7:                                 ;Find device: Device not found
        mov   ax,   7                    ;87H
        jmp   Fin
Error_8:                                 ;Config read CF <> 0
```

```
        mov  ax,  8
        jmp  Fin
Error_9:                       ;Config read: Unsupported function
        mov  ax,  9            ;81H
        jmp  Fin
Error_10:                      ;Config read: Bad vendor ID
        mov  ax,  10           ;83H
        jmp  Fin
Error_11:                      ;Config read: Device not found
        mov  ax,  11           ;86H
        jmp  Fin
Error_12:                      ;Config read: Device not found
        mov  ax,  12           ;87H
        jmp  Fin
```

After the error code has been stored in AX the function terminates and control is returned to the main program.

```
Fin:
        ...
        ret
```

3.2.5 Calculation of addresses

Now that base address is known, the addresses of the specific registers can be calculated. Bit 0 of the BAR0 is used to determine whether the register maps into memory or I/O space. This bit is therefor masked when calculating the addresses of the registers.

```
and  eax,  0FFFFFFFEH
```

Table 3.21 lists the registers and their offsets.

Table 3.21 PCI operation registers

PCI Operation Registers	**Address offset**
Outgoing Mailbox Register 1 (OMB1)	00h
Outgoing Mailbox Register 2 (OMB2)	04h
Outgoing Mailbox Register 3 (OMB3)	08h
Outgoing Mailbox Register 4 (OMB4)	0Ch
Incoming Mailbox Register 1 (IMB1)	10h
Incoming Mailbox Register 2 (IMB2)	14h
Incoming Mailbox Register 3 (IMB3)	18h
Incoming Mailbox Register 4 (IMB4)	1Ch
FIFO Register Port (bidirectional) (FIFO)	20h
Master Write Address Register (MWAR)	24h
Master Write Transfer Count Register (MWTC)	28h
Master Read Address Register (MRAR)	2Ch
Master Read Transfer Count Register (MRTC)	30h
Mailbox Empty/Full Status Register (MBEF)	34h
Interrupt Control/Status Register (INTCSR)	38h
Bus Master Control/Status Register (MCSR)	3Ch

AOMB1 = IMB1 = 10H

```
or      eax,        10H
mov     [AOMB],     eax
```

AMBEF = MBEF = 34H

```
or      eax,        34H
mov     [AMBEF],    eax
```

OMB1 = AIMB1 = 00H

```
or      eax,        00H
```

```
mov   [OMB],      eax
```

FIFO = 20H

```
or    eax,        20H
mov   [FIFO],     eax
```

MCSR = 3CH

```
or    eax,        3CH
mov   [MCSR],     eax
```

After the addresses have been calculated it is then stored in the corresponding variable.

3.2.6 Capturing of the data

Before the data are captured, the allocated memory address is setup, the Direction Flag (DF) is cleared to ensure that SI and DI will increment at the end of any string instructions and interrupts are disabled to ensure that the routine cannot be interrupted.

```
xor   ax,    ax
mov   fs,    ax
mov   edi,   [_BufferLinAddr]     ;Start of memory allocated
cld                               ;Clear direction flag
cli                               ;Disables int
```

Next the data capture board is switched off by writing FFFFFFFFH to OMB.

```
mov   edx,   [OMB]                ;Make sure that board is switched off
mov   eax,   0FFFFFFFFH
out   dx,    eax
```

This done by first setting up the OMB address in EDX and then loading EAX with the desired value. Then an OUT instruction is called to write the value to the specific register.

The setup and capturing of the data for the two implementations of the prototype are discussed in the following sections.

3.2.6.1 Mailbox implementation

Figure 3.33 shows the flow diagram of how the data is captured with the mailbox implementation.

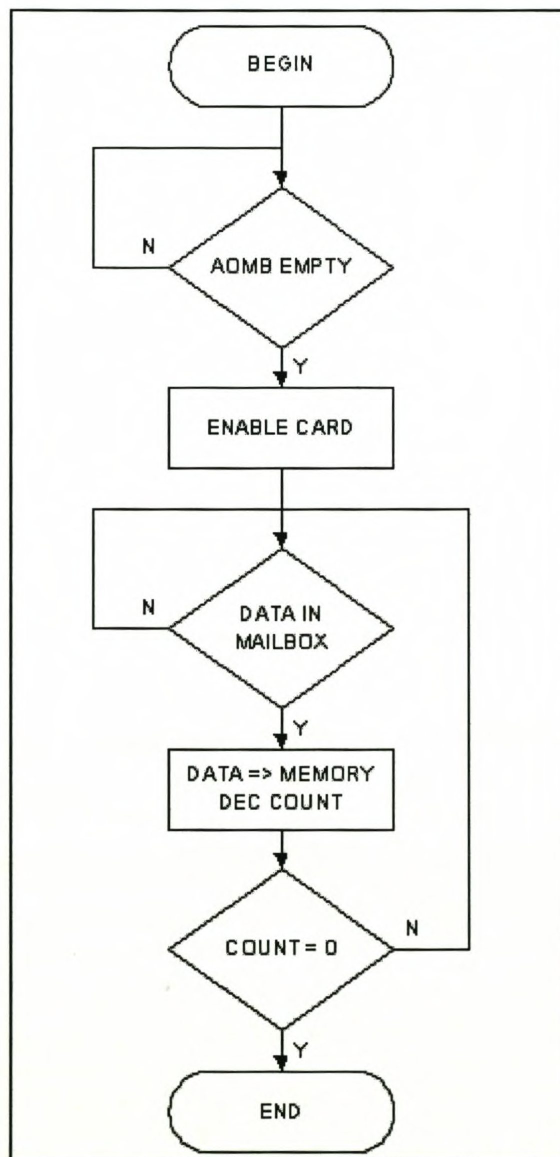


Figure 3.33 Mailbox implementation software flow diagram

First the AMBEF is checked to ensure that the AOMB is empty. If there are data in the AOMB it is read to clear the register.

```
mov   dx,    word ptr [AMBEF]
in    eax,   dx
and   eax,   0F0000H
cmp   eax,   0
jz    loop1                        ;AOMB is empty
mov   dx,    word ptr [AOMB]
in    eax,   dx                    ;clears AOMB
```

Next the capture card is enabled.

```
loop1:
      mov   edx,   [OMB]
      mov   eax,   0FFFFFFF5H
      out   dx,    eax
```

This is done by writing FFFFFFF5H to the OMB.

For testing purposes only 1MB of data is read from the card. The program is hardcoded by loading *count* with the amount of DWORDs needed to capture 1MB.

```
mov   [count],     40001h
```

AMBEF is then checked to see when 32 bits of data have been written to AOMB.

```
loop2:
      mov   dx,          word ptr [AMBEF]
      in    eax,         dx
      and   eax,         0F0000H
      cmp   eax,         0F0000H
      jne   loop2
```

AMBEF will be equal to 0F0000H when 32 bits were written to AOMB. The contents of AOMB is then read into memory.

```
mov  dx,        word ptr [AOMB]
in   eax,       dx
mov  fs:[edi],  eax
add  edi,       4
jmp  loop2
```

STOSD does the same function, but it takes more instruction cycles to execute. Since speed is a critical factor, the above method was used.

```
dec  [count]
jz   klaar
```

If COUNT is zero then 1MB of data has been captured.

3.2.6.2 FIFO implementation

Figure 3.34 shows the flow diagram of how the data are captured with the FIFO implementation.

First the MCSR is checked to ensure that the FIFO is empty. If there are data in the FIFO it is read to clear the register.

```
regread:
        mov  edx,  [MCSR]     ;Reads the status register
        in   eax,  dx
        and  eax,  020H       ;Mask bit 6 to determine if FIFO is
        cmp  eax,  020H       ;empty
        jz   loop1
        mov  edx,  [FIFO]
        in   eax,  dx
        jmp  regread
```

Next the capture card is enabled.

```
loop1:
        mov  edx,  [OMB]
```

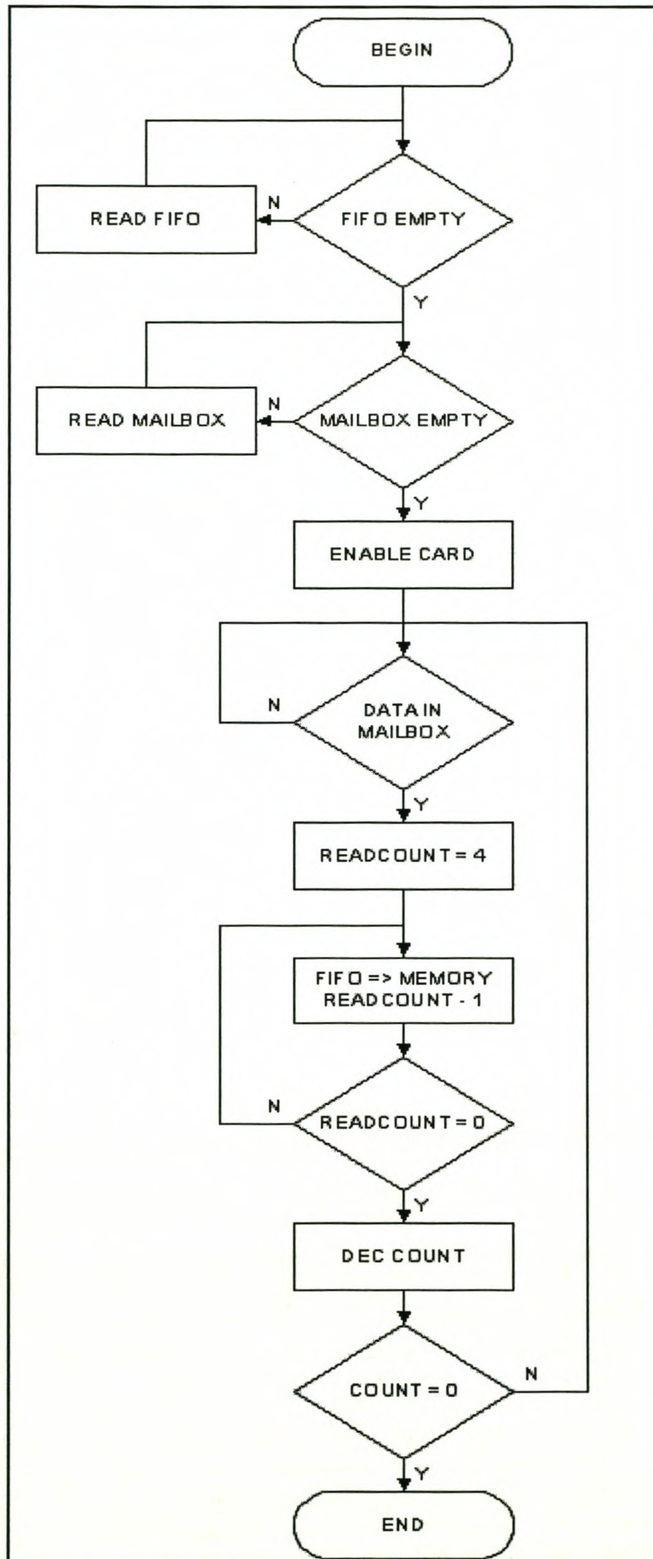


Figure 3.34 FIFO implementation software flow diagram

```
        mov   eax,   0FFFFFFF5H
        out   dx,    eax
```

This is done by writing FFFFFFF5H to the OMB.

Again for testing purposes the program is written to read 1MB of data, COUNT is loaded with the amount of DWords needed to capture 1MB divided by four because AOMB is used as an indicator to show when 4 DWords were written to the FIFO.

```
    mov   [count],      10001h
```

AMBEF is then checked to see if data was written to AOMB.

```
    loop2:
        mov   dx,            word ptr [AMBEF]
        in    eax,           dx
        and   eax,           0F0000H
        cmp   eax,           0F0000H
        jne   loop2
```

AMBEF will be equal to 0F0000H when 32 bits were written to AOMB.

Next the bits in AMBEF is cleared by reading AOMB.

```
        mov   dx,            word ptr [AOMB]
        in    eax,           dx
```

Now the four DWords in the FIFO can be read.

```
        mov   [readcount],   4
        mov   dx,            word ptr [FIFO]
    loop3:
        in    eax,           dx
        mov   fs:[edi],      eax
        add   edi,           4
        dec   [readcount]
```

```
        jz      loop2
        jmp     loop3
```

If COUNT is zero then 1MB of data has been captured.

```
    dec     [count]
    jz      klaar
```

Finally for both implementations the board is switched off by writing FFFFFFFFH to OMB and interrupts are enabled.

```
    klaar:
            mov     edx,    [OMB]
            mov     eax,    0FFFFFFFFH
            out     dx,     eax
            sti                             ;Enables int
            mov     ax,     0
            jmp     Fin
```

3.2.7 Store data on hard disk

Figure 3.35 shows how the data are stored to the computer's hard disk.

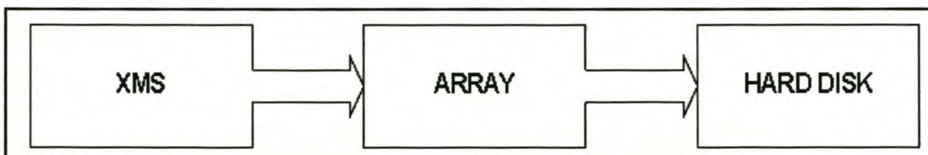


Figure 3.35 *MemToHD* routine

The data is first read from memory into an array. Since the data are captured in a serial stream it is not possible to know where a byte starts, byte synchronization must first be done. The data are therefor first converted to binary and stored as 1s and 0s in the file. This is done to simplify the testing of the card and software. The final software will write the data directly to the file without first converting it.

3.2.8 Deallocating the memory

Before the program exists the allocated memory must first be unlocked and then freed. The A20 address line must also be disabled. This is done by the *XMSUnlockMem*, *XMSFreeMem* and *LocalA20Disable* assembler routines.

For a complete listing of the source code is provided in appendix C.

3.3 Evaluation of the prototype

The final tests were done on the FIFO implementation. The next step was to create a test setup to verify the results from the simulations. Due to the high frequencies of the specification, testing proved a bit tricky. At that stage no logic analyser available was able to sample at a rate high enough to do any measurements. The engineering model RAM tray was used to generate a reliable data stream to test the card and the capture process.

At first an easily recognisable pattern was generated to verify that the FPGA implementation could handle the data tempo. Once this was verified, the capture software was tested to ensure that the whole system would meet the design specification. After a series of debugging runs the VHDL code and the software were optimised to a point where the pattern was captured at a rate of 40MHz.

Next a counter was implemented in the RAM tray. This generated a 32-bit counter to further verify the speed of the card. With a pattern it is still possible that 32-bit pieces of data may be missed and it will not be possible to detect it from the captured data. With the counter this will be easy to detect. The problem with the counter is that there is no byte synchronisation. To compensate a start sequence was defined to be able to detect the start of the counting sequence as well as the start of the first byte. After the data are captured, a software sequence was used to look for the start

sequence and do byte synchronisation. This test was also completed successfully at the required rate of 40MHz.

Finally a live test using the satellite, the high speed downlink and the demodulator was done. As shown in figure 3.36 the satellite was mounted on the 3-axis orientation platform to simulate the orbit of the satellite. A photo was taken and transmitted in real time using the high speed downlink. The signal was then received by a setup, figure 3.37, similar as what would have been used in the final ground station for the satellite. As shown in figure 3.37 it consists of the receiver with the demodulator(A) and the prototype data capture card(B). Figure 3.38 shows the prototype which consists of the PCI development board(B) and the FPGA interface(A) implemented in the FPGA tutor board. The test was successfully complete and the first image using the complete download path on the satellite was captured.

It was thus proven that the PCI bus can be used to capture the data received from the imager on SUNSAT I via the high speed down link.

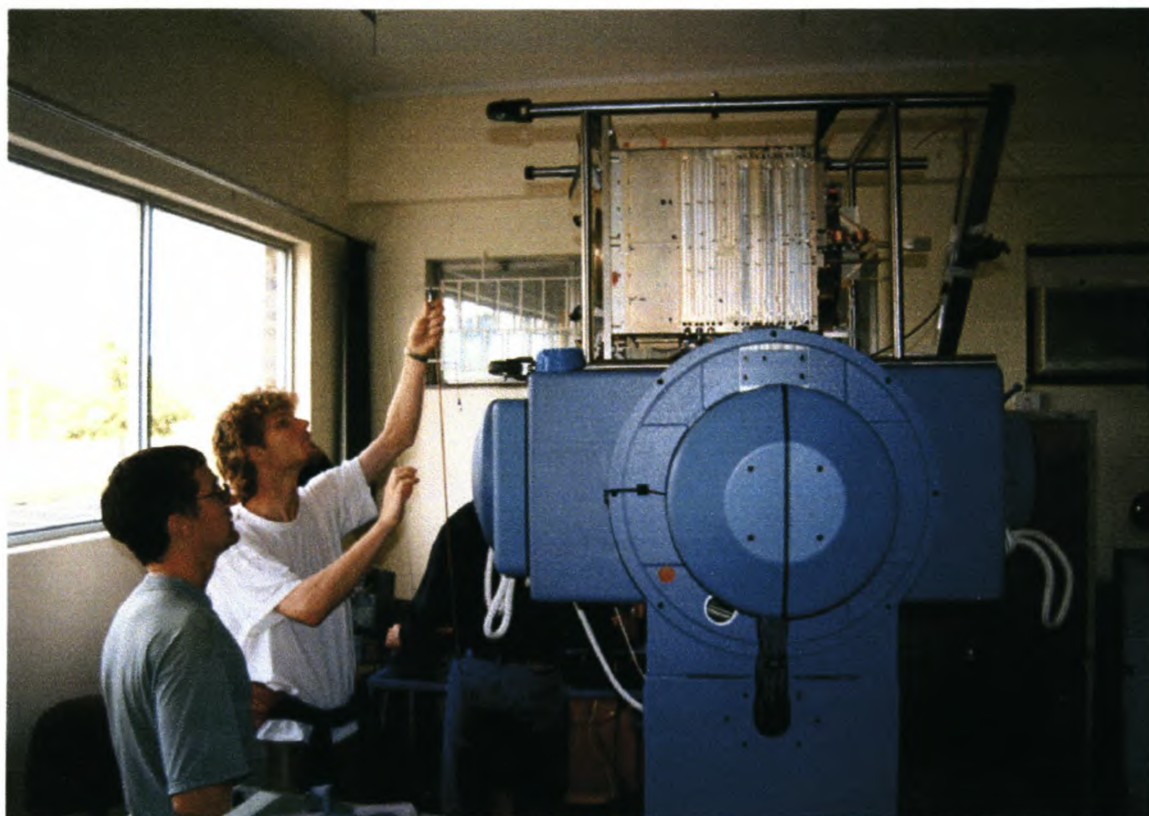


Figure 3.36 Taking a photo with SUNSAT

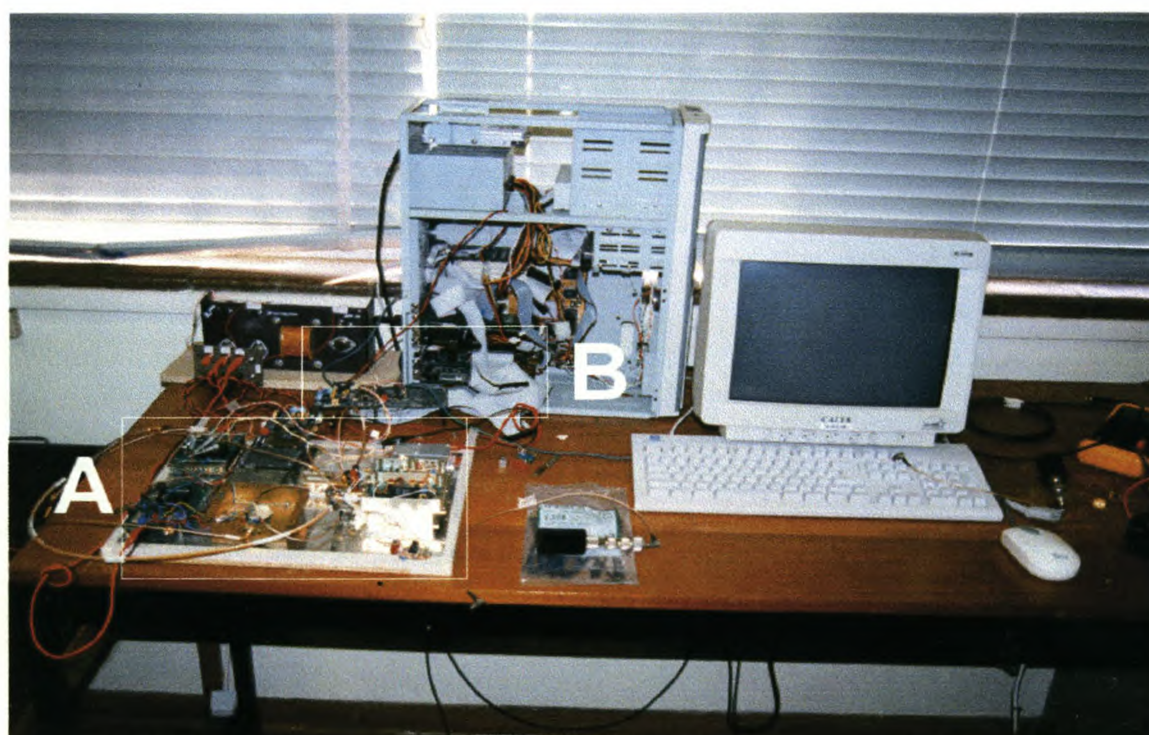


Figure 3.37 Prototype ground station setup

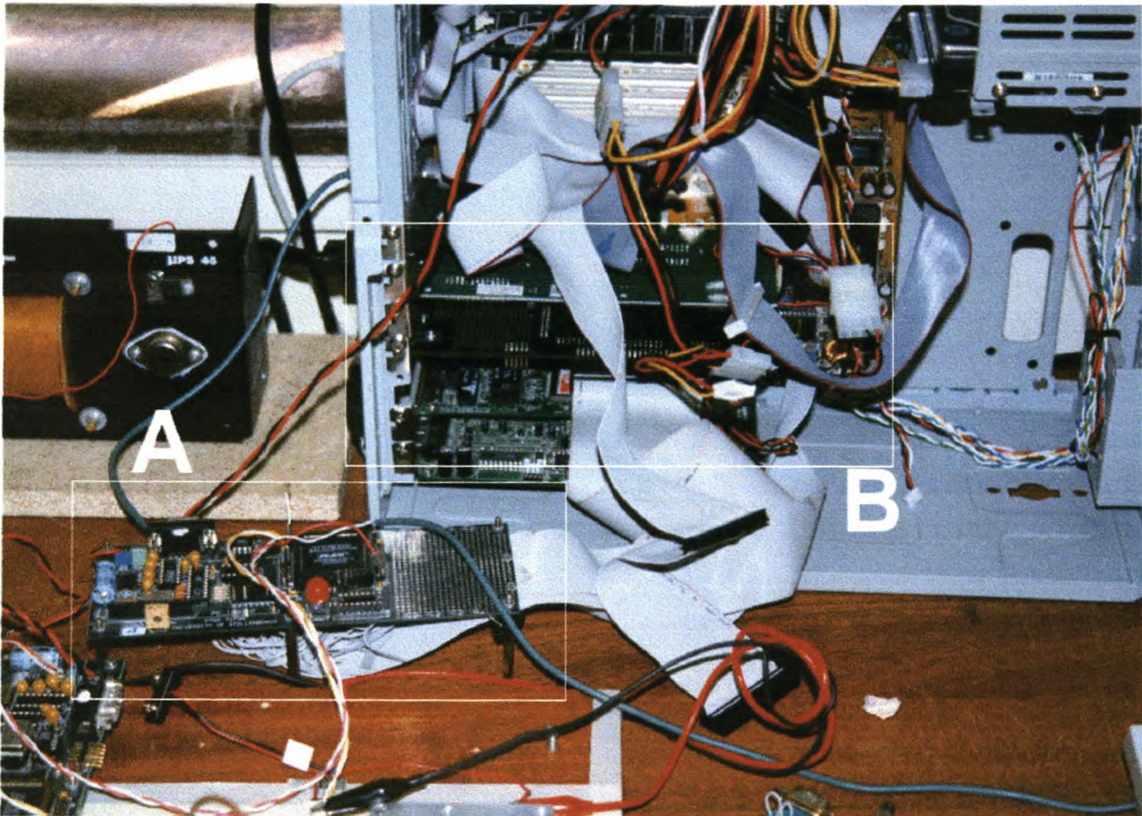


Figure 3.38 Prototype data capture card

Chapter 4

Design of the final product

4.1 Overview

Now that it was proven that the PCI bus can be used with relative ease and the high data transfer rates that can be practically achieved the final card can be designed. This card will then be used to capture the downloaded images from SUNSAT I. There is also a possible commercial application for the card in that the South African Air Force has shown interest in developing a digital camera for reconnaissance. This project is known as the Lantaba project. The data generated by the camera also needs to be stored on a computer at a very high rate. The capture card is thus being developed to satisfy the needs of both applications.

4.2 Specification

The specifications for both SUNSAT and Lantaba are as follow:

- Capture a serial data stream at 60MHz (SUNSAT)
- Capture a parallel (8 bit) data stream at 7.5MHz which is encoded with a National DS92LV1021 (Lantaba)
- The largest image that needs to be stored is approximately 1Gb
- It must program the programable oscillator of the camera (Lantaba)
- Count the amount of lines received from the camera to capture an image (Lantaba)

The implication of these specifications is that the card must have several external interfaces to be able to be used in both applications. It must also be able to capture data at a rate of up to 60 M/Bit per second.

4.3 Functional description

Since this card is already considered for two applications, more will most definitely arise. Keeping this in mind the card must be made as flexible as possible so that it can be used in as many applications as possible. Therefor although the PCI interface is capable of handling very high transfer rates, memory is also provided directly on the capture card. This can then be used to store the captured data and then allow for further processing directly on the card without using the computer's CPU. The layout of the card provides that some of the FPGAs can be configured via the PCI bus. This allows different applications to be downloaded to the FPGAs allowing the reuse of hardware. It allows for the FPGAs to be firstly configured to capture the data and reprogrammed with applications such as compression routines or pattern recognition routines. By using the FPGAs for these type of applications in some cases it is also more efficient to use them than the computer itself. The FPGAs provide a hardware implementation of the routines that is much faster that the equivalent software implementation.

Figure 4.1 shows the high level layout of the final data capture card.

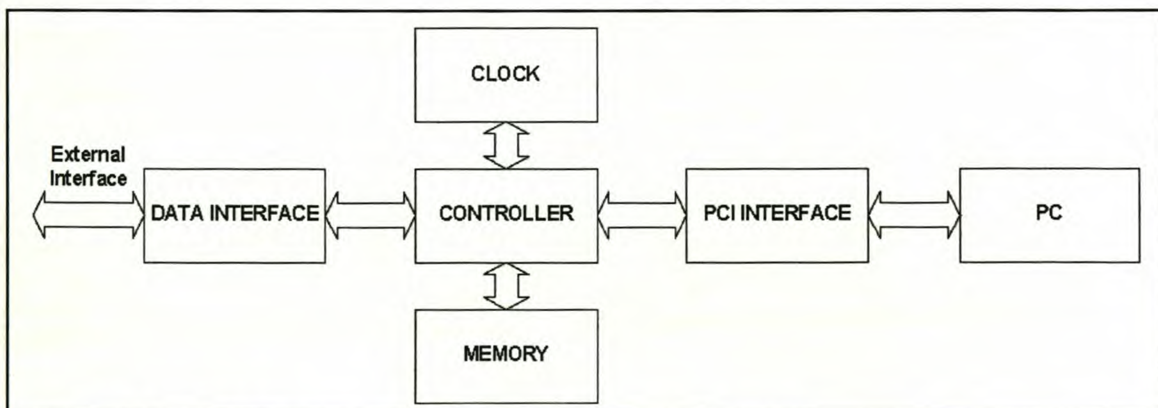


Figure 4.1 Data capture card high level block diagram

The controller is the heart of the card, with the data interface providing the interface to the outside world and the PCI interface to the PCI bus. The clock component controls the clock signals for all the components on the card and the memory component provides storage on the card.

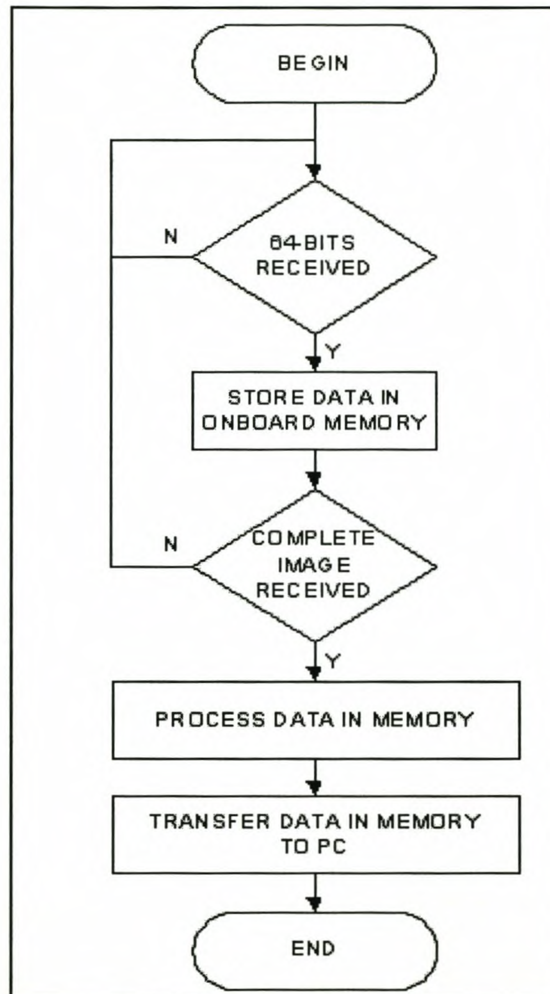


Figure 4.2 Final design basic flow diagram

As shown in figure 4.2 the basic functionality for both the Lantaba and SUNSAT applications are as follow:

- The incoming serial or parallel data is packed in 64-bit wide chunks. This is done by way of a shift register.
- When 64-bits are available it is stored in the onboard memory.

- When a complete image was received the image can either be processed on the card or stored on the computers hard disk via the PCI bus.

4.3.1 PCI interface

It provides a standard interface for data and commands to be transferred between the computer and the controller interface over the PCI bus.

4.3.2 Controller interface

This is the heart of the card. It controls the working of the board. As shown in figure 4.3 the controller consists of two parts: main controller FPGA and program controller FPGA. Each of these parts will be implemented in a separate FPGA.

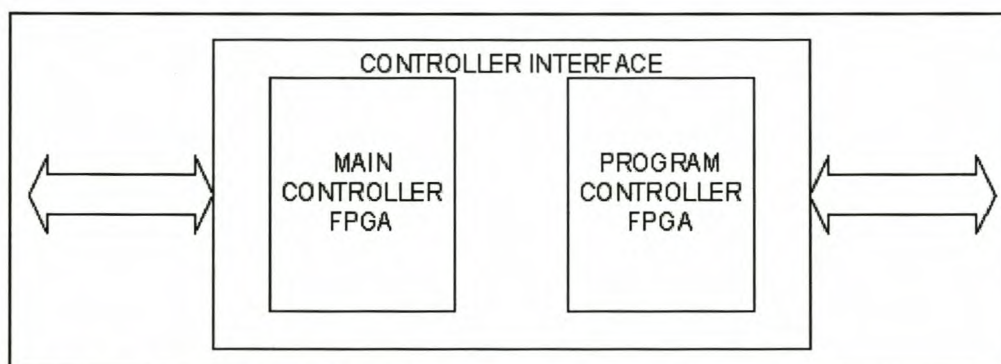


Figure 4.3 Controller interface

The main controller FPGA controls the PCI interface, the memory, the data interface and programs the clock and the program controller FPGA. The program controller FPGA controls the 64-bit data busses among all the components, allows direct interfacing to the PCI bus through the PCI interface and programs the data interface. Figure 4.4 shows the basic flow diagram of the working of the controller. At power-up only the program controller FPGA is configured. The program controller FPGA then waits for software to initialise the programming of the main controller FPGA and then

the data interface. When all the components have been properly configured and initialised the downloaded application can be executed. This application can either capture data to the memory on the card, process data already in memory or read the data from memory to be stored on the computer.

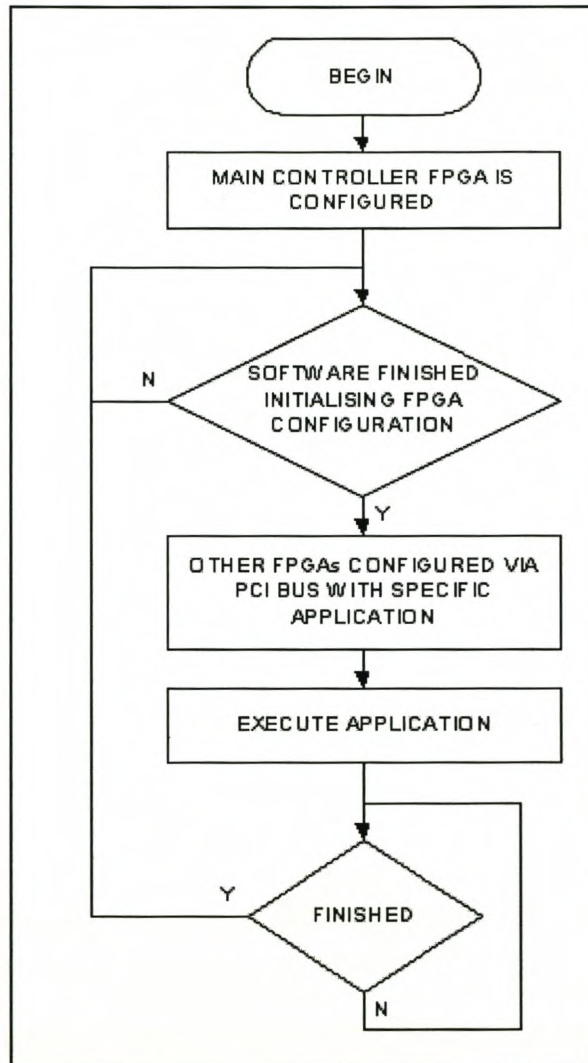


Figure 4.4 Controller flow diagram

4.3.3 Clock

The PCI clock is only distributed to the controller FPGAs, the global clock signal is generated by the clock component. Although the PCI clock can only go as high as

66MHz the specifications for the card require that the global clock must be as high as 100MHz. For further flexibility the global clock is not fixed. It is possible to program the frequency of the clock with software through the main controller FPGA.

4.3.4 Memory

The memory component provides a very fast large capacity memory pool in which incoming data can be stored at a very high tempo and where captured data can be processed.

4.3.5 Data interface

The data interface provides a generic outside interface for the capture card to ensure its flexibility. Further more, it can also be reprogrammed to perform other operations on data stored in the onboard memory. The same piece of hardware can thus be used to perform several functions.

For the SUNSAT I application when the serial data is captured and converted to parallel, it is unknown where the bytes start. After the data has been received, it is possible to do a byte synchronisation on the data so that when the data is downloaded to the computer the bytes are correct.

4.3.6 External interface

Although the external signals needed for both the SUNSAT and the Lantaba applications are fixed, additional signals are needed to ensure the flexibility of the card. Listed below are the external signals and descriptions for the two applications.

Table 4.1 SUNSAT external signals

Signal name	Description
SDAT1A	Input from SN65LVDS31 High-Speed Differential line driver from the SUNSAT ground station.
SDAT1B	
SDAT2A	
SDAT2B	
SDAT3A	
SDAT3B	
SDAT4A	
SDAT4B	

These signals are then decoded by the SN65LVDS32 High-Speed Differential Line Receiver to the following signals.

Table 4.2 Decoded SUNSAT signals

Signal name	Description
SDAT1	Incoming serial data stream
SDAT2	Strobe line associated with the serial data stream
SDAT3	Active low signal that is active when valid data is received by the QPSK demodulator
SDAT4	Spare

Table 4.3 Lantaba external signals

Signal name	Description
DISWITCH	Camera switch line. It is used to switch the camera on and off.
R+	Output form the serializer chip on the camera board
R-	Output form the serializer chip on the camera board

PCG_SCLK	Serial clock input line for programming purposes
PCG_DATA	Serial data input for programming

R+ and R- are decoded with a National DS92LV1021 deserialiser to the following signals.

Table 4.4 R+ and R- decoded

Signal name	**Description**
R[7:0]	Incoming parallel data from the camera
R8	Line sync pulse indicates the end of a CCD line(6042x3 pixels).
R9	Strobe line associated with the incoming data.

The following additional signals are also provided:

SPARE[13:1]
SPARE[28:16]
SPARE[41:29]

Chapter 5

Implementation of the final product design

5.1 Overview

With the design of the final card some interesting challenges were encountered. The main consideration that had to be kept in mind during all the stages of the design was speed. The effect that each design decision had on the performance of the complete design had to be considered. Although speed was the main requirement of the final product the financial and time implications also had to be considered. A set of design reviews were held to ensure that the development cycle of the card will be as short as possible, thus keeping the number of PCB iterations to a minimum. The time frames in which the card was needed for SUNSAT and Lantaba were very short and everything had to be done to eliminate possible problems that could arise.

One of the most challenging processes in the design of the final card was the choice of components. Factors that had to be kept in mind were: speed (timing), supply voltage, signal voltage, the amount of power that the chip can sink as well as deliver, packaging, cost, volume and availability. It happened a few times that a specific component that would have been perfect either has not been released, is not available in South Africa or is only available in bulk. The choice of components for each part of the design directly affects each other. The following sections will describe the component choices that were made.

5.2 Memory

What type of memory to use had the biggest impact on the choice of components for other parts of the design. It influenced the supply voltage as well as the signal levels and driving capabilities of the components connected to the memory. As a total of 1GB of memory was needed the obvious choice for lowest price per megabyte was computer memory. At that stage SIMMs were the most common type of memory, but the new Pentium II processors and motherboards were becoming more affordable and with it DIMMs started to become the new memory standard. Since this was the first PCI card to be developed at the University of Stellenbosch and the life span of the card is to be a few years, the best choice was to use DIMMs.

Using DIMMs had the following implications: DIMMs have a 3.3V supply voltage and signal levels, the inputs are not 5V compatible and DIMMs have a very high input capacitance, which means that chips connected to it must have very high driving capabilities. To realise the 1GB of memory that is needed four 256MB DIMMs split into two banks are used. This means that the components connected to the memory must be able to drive at least two DIMMs at a time.

The specification for DIMMs allows for both 3.3V and 5V supply voltages. 5V DIMMs are obsolete and almost impossible to get hold of. Further more DIMMs are classified as buffered (registered) or unbuffered. Both these specifications are implemented by using the keying methology. Figure 5.1 shows the keying methodology employed on DIMMs. One key defines the device type and the other the voltage. The voltage key provides a positive interlock so that DIMMs can only be plugged into a system with the proper supply voltage, reducing potential damage to the module devices. The appropriate connector must be used or else the system will not work.

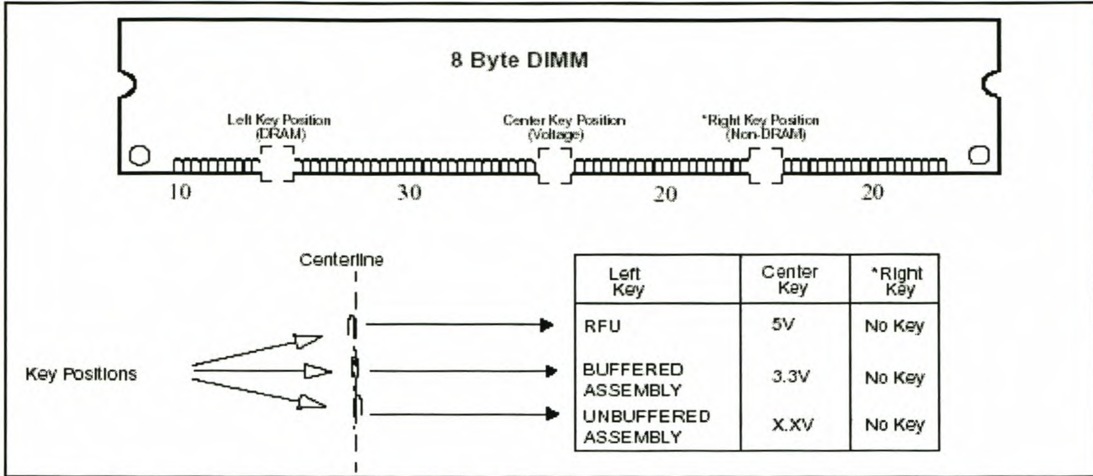


Figure 5.1 Type and voltage keying

RFU - Reserved for Future Use

For the 168-pin DIMM there are three different data bus widths/organizations available:

- 64-bit Non-Parity
- 72-bit Parity
- 72-bit ECC
- 80-bit ECC

The simplified pin function descriptions for a DIMM are listed below in table 5.1.

Table 5.1 Simplified pin functional description

Symbol	**Type**	**Description**
A[n:0]	Input - Synchronous	Some address pin definitions change as a function of array size and # of banks used.
CLK	Input - Clock	Master Clock input.

CKE	Input - Clock Enable	Activates the CLK signal when high and deactivates when low. By deactivating the clock, CKE low initiates the Power Down mode, Self Refresh mode or Suspend Mode.
RAS#	Input - Synchronous	Row address strobe.
CS#	Input - Synchronous	Selects chip when active.
CAS#	Input - Synchronous	Column address strobe.
WE#	Input - Synchronous	Write Enable strobe.
DQM,DQM L/H	Input - Synchronous	DQ Mask. Active high. Controls the data output buffers in read mode. In write mode it masks the data from being written to the memory array.
DQ(x:0)	Input/Output - Synchronous	Data IO pins.
CB(x:0)	Input/Output - Synchronous	Parity/Check BitInput/Output
NC/RFU	No connect/ Reserved for Future Use	This pin should be left No Connect on the device so that the normal functionality of the device is not be affected by the external connection to this pin. This pin could be used in future.
Vcc, Vss	Power pins	Supply Pins for the core
VccQ, VssQ	Power pins	Supply Pins for the output buffers

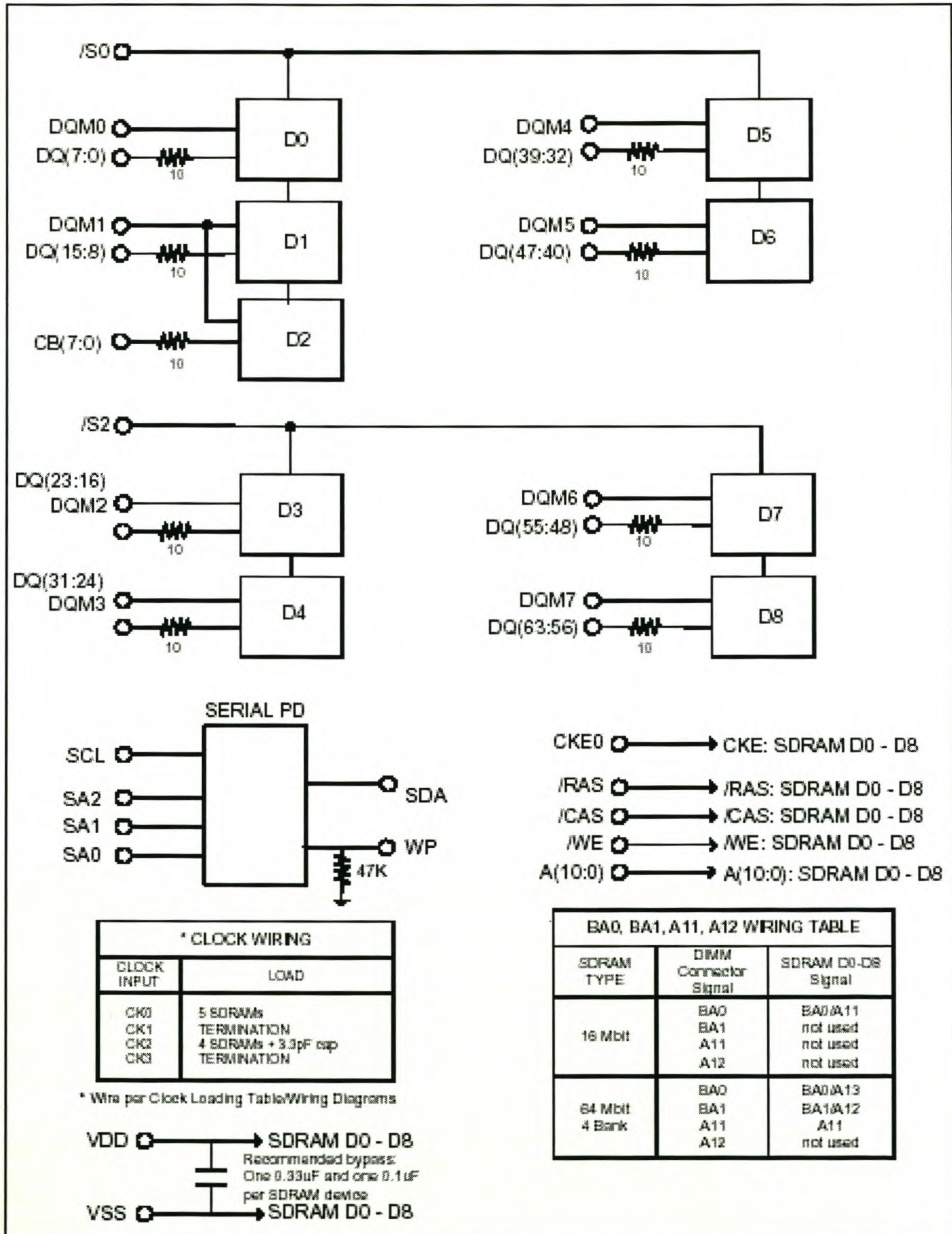
5.2.1 Differences between buffered and unbuffered DIMMs

5.2.1.1 Unbuffered DIMMs

Unbuffered DIMMs allow the system to take full advantage of the SDRAM speed by eliminating the on-card buffers and are intended for systems with one or two DIMMs or systems that do the buffering on the main board.

A common pinout assignment ensures support for multiple types of 168-pin unbuffered DIMMs. The common DIMM pinouts for controls signals: RAS#, CAS#, WE# and CS# greatly simplify the memory subsystem design that can support all the different data bus widths/organizations available. Figure 5.2 shows the block diagram of an unbuffered DIMM.

A careful consideration with unbuffered DIMMs is that all the external lines are directly connected to the SDRAM chips as seen in figure 5.2. Because the SDRAM chips have very capacitive inputs it increases the capacitive load that the components connected to the DIMM must be able to handle. Table 5.2 provides a list of the input capacitances of a few DIMM models.



* CLOCK WIRING	
CLOCK INPUT	LOAD
CK0	5 SDRAMs
CK1	TERMINATION
CK2	4 SDRAMs + 3.3pF cap
CK3	TERMINATION

BA0, BA1, A11, A12 WIRING TABLE		
SDRAM TYPE	DIMM Connector Signal	SDRAM D0-D8 Signal
16 Mbit	BA0	BA0/A11
	BA1	not used
	A11	not used
	A12	not used
64 Mbit 4 Bank	BA0	BA0/A13
	BA1	BA1/A12
	A11	A11
	A12	not used

Figure 5.2 Unbuffered DIMM block digram

Table 5.2 Unbuffered DIMM input capacitance

Signal	**Samsung**	**Fujitsu**	**IBM**	**Mitsubishi**
A0 - A11 BA0 - BA1	40-60	85	95	55
RAS CAS	40-60	85	95	55
CKE0 - CKE1	25-35	45	46	35
CLK0 - CLK1	25-35	25	42	35
CS0 - CS3	15-25	25	28	22
DQM0 - DQM7	15-25	15	18	22
DQ0 - DQ63	5-15	19	16	22

All the values are measured in pF

These ratings are for a single DIMM. Which means that for the data capture card the components must be able to drive two DIMMs at a time, the total capacitance can be as high as 190pF.

According to the specification of the DIMM the maximum rise time of a signal from a logic low to a logic high state is 5ns. Which means that enough current must be supplied to provide a voltage increase of 1.2V 5ns.
With:

$$I = C\frac{dV}{dt}$$

$$= \frac{190p \times 1.2}{5n}$$

$$I = 45.6mA$$

Which means the components must be able to drive almost 100mA.

5.2.1.2 Buffered DIMMs

All control and address signals are synchronized with the positive edge of externally supplied clocks and are registered on-DIMM and hence delayed by one clock cycle in arriving at the SDRAM devices. This pipelining allows the path between the memory controller and the DIMMs to be achieved in two clock cycles rather than one. Use of an onboard register also reduces the capacitive loading of the DIMM on input control and address signals. The SDRAM device data lines (DQ) are connected directly to the DIMM tabs through 10 Ohm series resistors. Figure 5.3 shows the effect of the registered mode on the data outputs for a read operation.

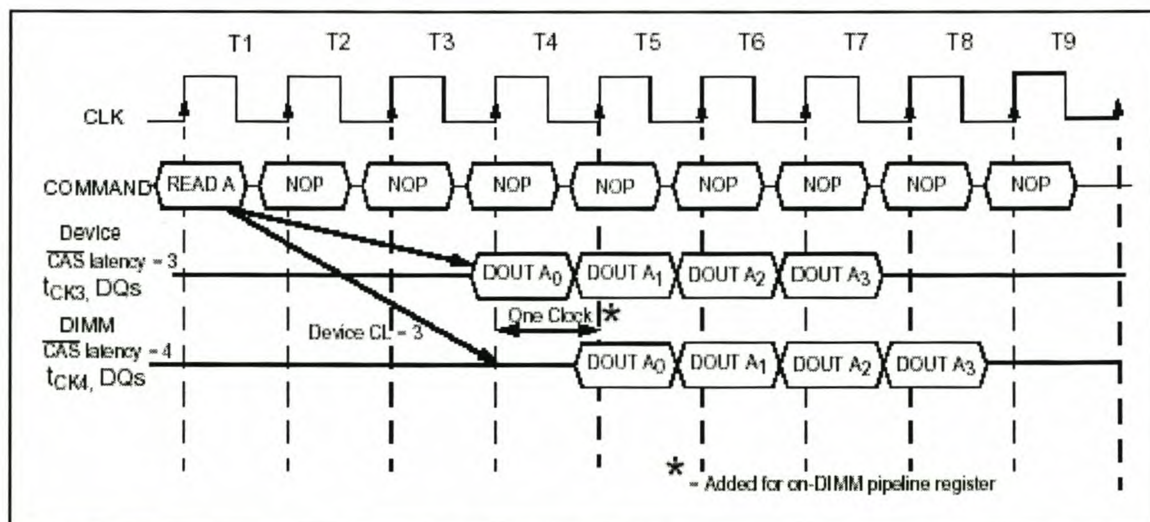


Figure 5.3 Buffered vs. unbuffered operation

Without the register, the data is delayed according to the device CAS latency, in this case three clocks. CAS latency is the number of clock cycles needed for data to be retrieved. With the register, the data is delayed according to the device CAS latency plus an additional clock cycle. This is known as the DIMM CAS latency, and in this example is four clocks. The data path can be thought of as a pipeline in which the register effectively lengthens the pipe by one clock cycle. Figure 5.4 shows the block diagram of a registered DIMM.

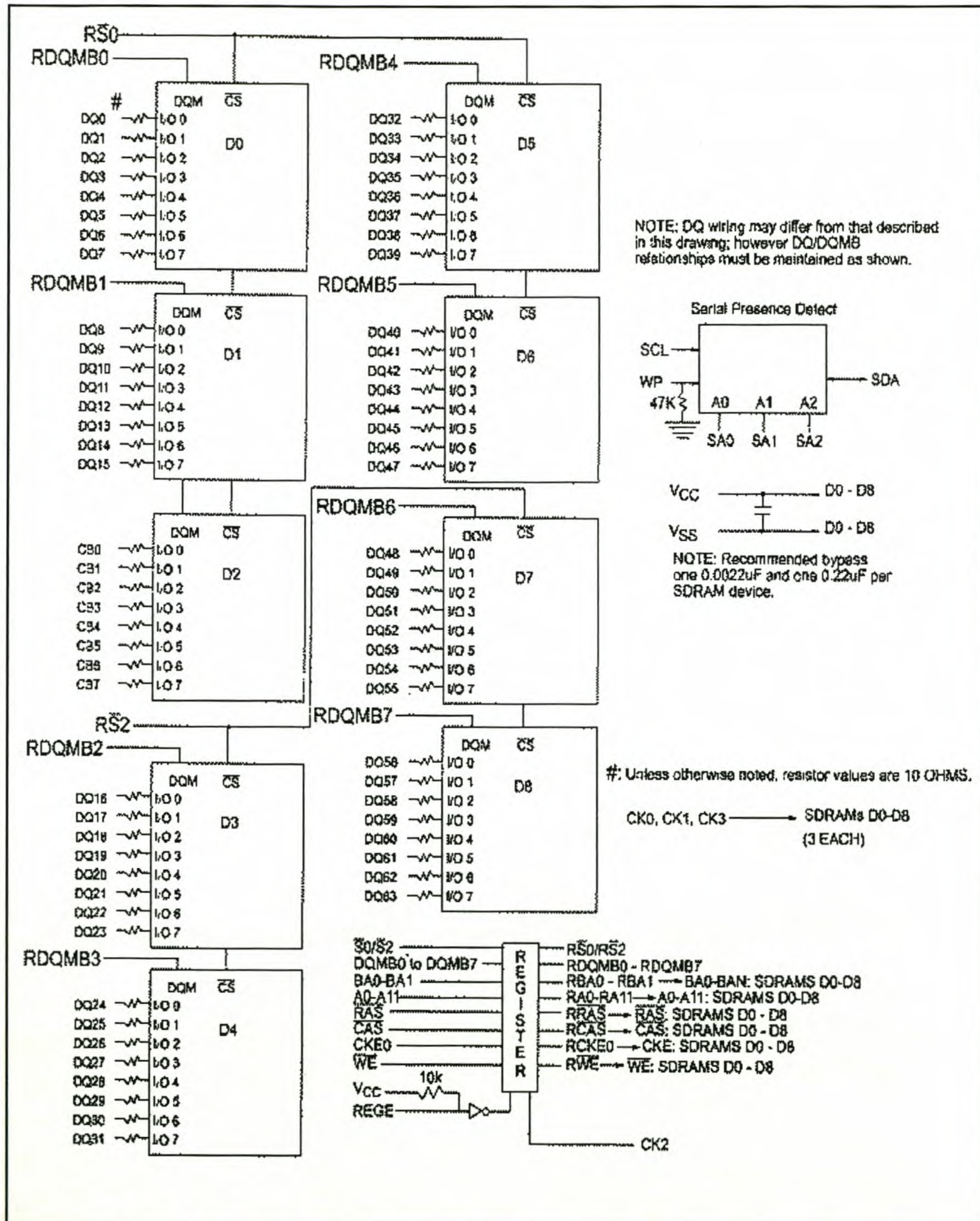


Figure 5.4 Buffered DIMM block diagram

The buffering of the address and control signals dramatically reduces the capacitance of the DIMM's input signals. Table 5.3 shows the input capacitances of a few buffered DIMM modules.

Table 5.3 Buffered DIMM input capacitance

Parameter	**Symbol**	**Micron**	**Mosel Vitelic**
Input Capacitance: A0-A11, BA0, BA1, RAS#, CAS#, WE#	Ci1	8	5
Input Capacitance: S0#-S3#, CKE0, DQMB0#-DQMB7#	Ci2	8	5
Input Capacitance: CK0	Ci3	6	4
Input Capacitance: REGE	Ci4	5	5
Input Capacitance: SCL, SA0-SA2, WP	Ci5	12	5
Input/Output Capacitance: DQ0-DQ63, CB0-CB7, SDA	Cio	16	6.5

All the values are measured in pF

These ratings are for a single DIMM. Which means that for the data capture card the components must be able to drive two DIMMs at a time, the total capacitance will now be a meagre 32pF.

With the specified rise time of 5ns:

$$I = C\frac{dV}{dt}$$
$$= \frac{32p \times 1.2}{5n}$$
$$I = 7.68mA$$

Which means the components connected to the DIMM must only be able to drive

15.36mA. This is considerably less than the almost 100mA needed for the unbuffered DIMMs.

5.2.2 The serial presence detect (SPD) function

DIMM modules include a serial EEPROM which stores information about the module. The system board can gather information from this chip and change the necessary settings in order to be compatible with the DIMM.

Table 5.4 lists the SPD interface signals.

Table 5.4 SPD interface signals

Signal	Description
SA(2:0)	Address inputs
SCL	Clock input
SDA	Serial data input/output

The SPD interface uses the I^2C (Synchronous 2-Wire Bus) protocol. Figure 5.5 shows the common clock / common data wiring option used for the data capture card.

SA(0:2) are wired at each DIMM socket in a binary sequence for a maximum of 8 devices. The SDA and SCL are common across all positions and are required to have pull-up resistors.

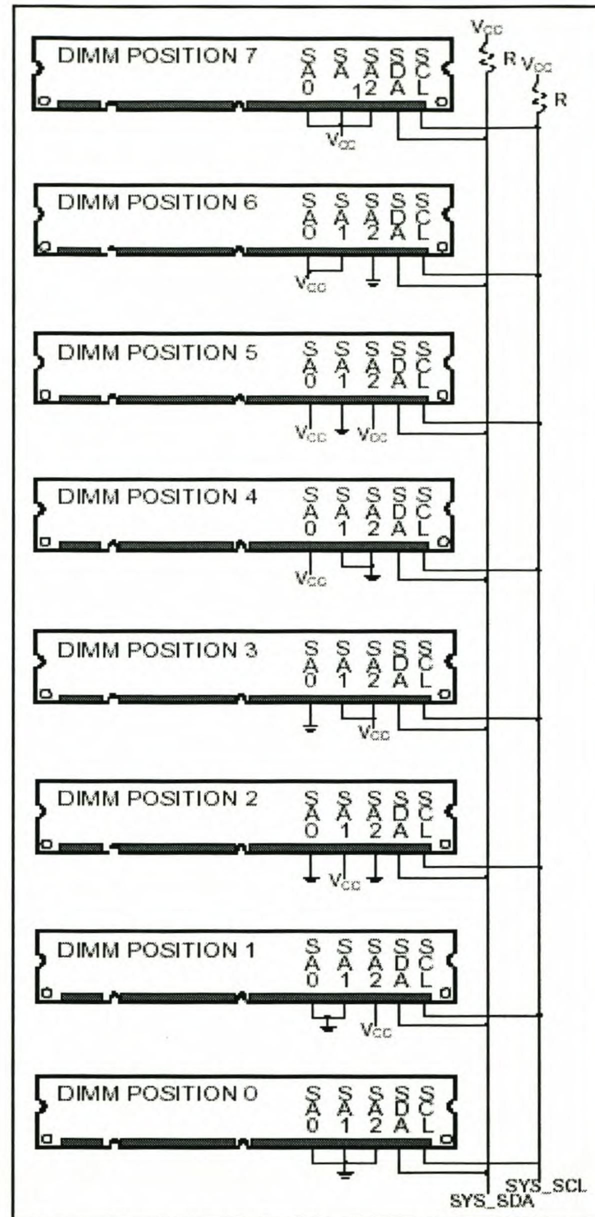


Figure 5.5 SPD wiring option

The format for the SPD EEPROM data is listed in appendix D. The required fields must be supported for the DIMM to be compliant to the standard.

SPD clock and data conventions

Data states on the SDA line can change only during SCL LOW. SDA state changes during SCL HIGH are reserved for indicating start and stop conditions (Figures 5.6 and 5.7).

SPD start condition

All commands are preceded by the start condition, which is a HIGH-to-LOW transition of SDA when SCL is HIGH. The SPD device continuously monitors the SDA and SCL lines for the start condition and will not respond to any command until this condition has been met.

SPD stop condition

All communications are terminated by a stop condition, which is a LOW-to-HIGH transition of SDA when SCL is HIGH. The stop condition is also used to place the SPD device into standby power mode.

SPD acknowledge

Acknowledge is a software convention used to indicate successful data transfers. The transmitting device, either master or slave, will release the bus after transmitting eight bits. During the ninth clock cycle, the receiver will pull the SDA line LOW to acknowledge that it received the eight bits of data (Figure 5.8). The SPD device will always respond with an acknowledge after recognition of a start condition and its slave address. If both the device and a write operation have been selected, the SPD device will respond with an acknowledge after the receipt of each subsequent eight-bit word. In the read mode the SPD device will transmit eight bits of data, release the SDA line and monitor the line for an acknowledge. If an acknowledge is detected and no stop condition is generated by the master, the slave will continue to transmit data. If an acknowledge is not detected, the slave will terminate further data transmissions and await the stop condition to return to standby power mode.

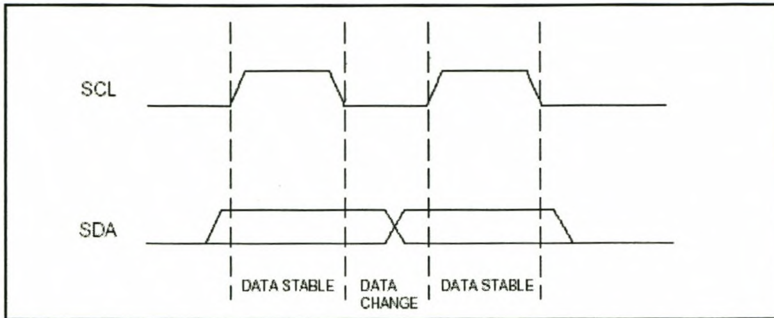


Figure 5.6 SPD data valid

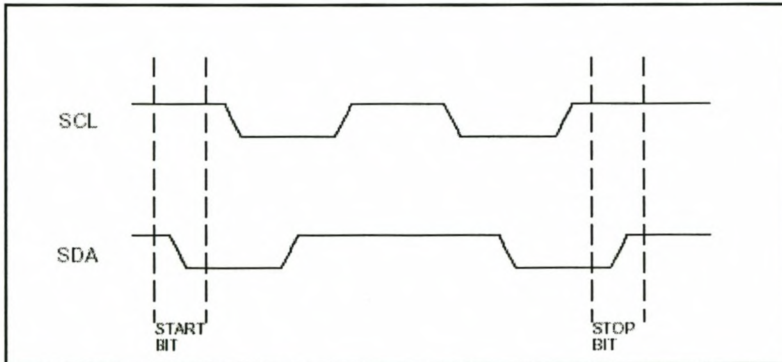


Figure 5.7 SPD start and stop

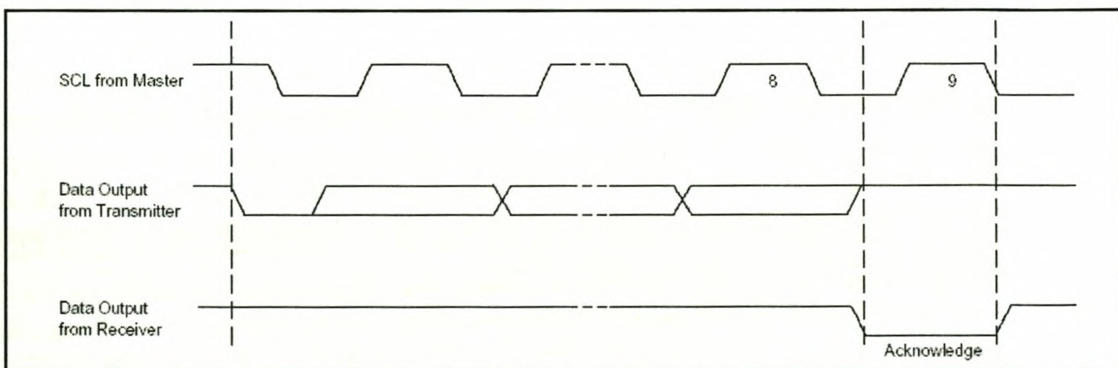


Figure 5.8 SPD acknowledge

5.2.3 DIMM speed classification

Intel created both the specifications for the PC66 and the PC100 SDRAM DIMMs which tightly define their electrical characteristics. The PC100 and PC66 DIMMs look the same and have the same pinouts. Both start out with the same basic concepts and similar physical appearances but both are really quite different from performance aspects. On the module itself, the PC100 specification requires that the length of the traces be kept shorter than the PC66 modules. The shorter the traces the faster the electrical signals can travel on and off the module. The routing of the traces on the PC100, because they need to be kept shorter, has to use a 6 layer PCB as compared to a 4 layer PCB used on the PC66. The PC100 modules require faster speed SDRAM components than the PC66. A PC100 module can run at the same speed as a PC66 module, but a PC66 module will not work at the same speed as that of a PC100 module.

The SPD content of the PC100 SDRAM modules is different from the content of the PC66 SDRAM modules. The maximum rated speed or bus frequency information is contained in the SPD, along with the CAS latency parameters. A system will use this information to determine if it can support the module and how to configure the onboard memory controller.

5.2.4 Choice of DIMM to be used on the capture card

In the first version of the final design unbuffered DIMMs was used. As predicted the high capacitance of the input pins required that additional driver chips be added between the DIMMs and the other components. The problem with these buffers was that they had to be capable of handling very high frequencies, provide enough current to drive the DIMMs and operate at 3.3V levels. The SN74ALB16244, 16-bit buffer/driver with 3-state outputs from Texas Instruments was considered to be used in conjunction with the unbuffered DIMMs. Due to the delays that the buffers induced

as well as availability problems the decision was made to use buffered DIMMs and connect them directly to the other components.

The 256MB registered SDRAM DIMM, MT36LSDT3272, from Micron was chosen. At that stage no local distributer of DIMMs in South Africa had any registered DIMMs. The only other option was to import the DIMM from overseas. Micron marketed through Crucial Technology to educational institutions was at that stage the only viable source. Although the DIMM that was ordered has a Crucial Technology part number, it is equivalent to the Micron one listed above.

The complete pinout for the 168-pin buffered DIMM is provided in appendix E1, as well as a functional description of the pins in appendix E2.

The truth table in Table 5.5 provides a general reference of available commands.

Table 5.5 Available commands truth table

Name (function)	**CS#**	**RAS#**	**CAS#**	**WE#**	**DQMB**	**ADDR**	**DQs**
COMMAND INHIBIT (NOP)	H	X	X	X	X	X	X
NO OPERATION (NOP)	L	H	H	H	X	X	X
ACTIVE (Select bank and activate row)	L	L	H	H	X	Bank/Row	X
READ (Select bank and column, and start READ burst)	L	H	L	H	X	Bank/Col	X
WRITE (Select bank and column, and start WRITE burst)	L	H	L	L	X	Bank/Col	Valid
BURST TERMINATE	L	H	H	L	X	X	Active
PRECHARGE (Deactivate row in	L	L	H	L	X	Code	X

bank or banks)							
AUTO REFRESH or SELF REFRESH (Enter self refresh mode)	L	L	L	H	X	X	X
LOAD MODE REGISTER	L	L	L	L	X	Op-code	X
Write Enable/Output Enable	–	–	–	–	L	–	Active
Write Inhibit/Output High-Z	–	–	–	–	H	–	High-Z

The complete DIMM datasheet is provided on the attached CD, under *\DATASHEETS\DIMM*.

5.3 Controller

Each of the two parts of the controller is realised in a FPGA. Both the main controller and program controller interface with both 5V and 3.3V devices and have a very high I/O count. At the time of the design of the final card the choice of large I/O FPGAs were very limited and these devices were still very expensive. The first choice was the vendor. The final choice was between Actel, Xilinx and Altera. The problem with Actel at that stage was that almost all their FPGAs were only one time programable. Given the fact that this design was the first to utilise the PCI bus, have mixed supply and voltage levels and make use of DIMMs, it will not have been possible to add all the unknown factors into the simulation of the design before programming the FPGAs. The different components of the design had to be tested first before they could be integrated into one final design. The design and simulation tools of Xilinx were very underdeveloped which made simulation process very tedious. On the other hand the Altera tools provided an integrated environment for design, simulation and implementation of VHDL applications. Good design experience was obtained from undergraduate use of the tools as well as projects done for postgraduate subjects. On the hardware side Altera was also on the leading edge with new FPGA technologies. The choice was between one of their BGA (ball-grid array) package

options. The FLEX 10K family offered high performance, over 100MHz, high density, up to 250 000 gates, 3.3V supply voltage and MultiVolt I/O operation, able to drive 5V and 3.3V devices. The EPF10K30ABC356-3 was chosen for the program controller and the EPF10K50VBC356-3 for the main controller. These chips have the same pinout, but the 10K30 has 30 000 gates and the 10K50 50 000.

The 10k30/50 has a maximum output driving capability of 25mA, which means that with:

$$
\begin{aligned}
I &= C\frac{dV}{dt} \\
C &= I\frac{dt}{dV} \\
&= \frac{25m \times 5n}{1.2} \\
C &= 104\,pF
\end{aligned}
$$

The maximum capacitive load that the FPGA can drive is 104pF, which is adequate for the DIMMs being used.

Only the program controller is configured at power-on with an Altera EPC2 configuration EEPROM or can also be configured using the Atera Bit- or ByteBlaster during the debugging phase. The main controller can either be configured by the program controller or with the Bit- or ByteBlaster.

Several LEDs are also connected to the two FPGAs to assist in debugging.

The complete datasheets for the FPGAs and the EPC2 are provided on the attached CD, under *\DATASHEETS\ALTERA*.

5.4 Clock

To help with the development stage of the card a variable clock was needed. This would make it possible to first test the components at low speeds and then operate them at full speed. The Cypress ICD2053B programmable clock generator was selected due to its ease of use and frequency range. The output frequency may be changed on the fly to any desired frequency value between 391 kHz and 100 MHz at TTL levels. Table 5.6 gives the signal description for the ICD2053B.

Table 5.6 ICD2053B pin description

Name	**Pin Number**	**Description**
XTALOUT	1	Reference crystal feedback
SCLK	2	Serial clock input line for programming purposes
GND	3	Ground
DATA	4	Serial data input line for programming purposes
CLKOUT	5	Programmable clock output. This clock output can be three-stated by either pin 7, when it is configured as an Output Enable pin, or by bit 1 of the Control register.
VDD	6	+5 volts
MUXREF/ OE	7	If bit 3 (Pin 7 Usage) in the Control register is set to 1, this input pin controls the multiplexed reference frequency function. If bit 3 (Pin 7 Usage) in the Control Register is set to 0, this input pin controls the three-state output function. On power-up, pin 7 implements the OE function; a HIGH on pin 7 enables CLKOUT.
XTALIN	8	Reference crystal input or external reference input ($f_{(REF)}$)

5.4.1 ICD2053 registers

The ICD2053B contains two registers, the Control and Program registers. These registers are written to using a protocol which uses a Protocol word = 011110 to distinguish Control register data from Program register data. This Protocol word is recognized by the four sequential 1s; therefore, all other data sent must have a 0 bit stuffed in after each sequence of three sequential 1s (whether originally followed by a 1 or a 0). This is called bit-stuffing.

All serial words are shifted in bit by bit starting with the LSB. A low-to-high transition on SCLK is used to shift data. Whenever the Protocol word is detected, the preceding 8 bits are transferred into the Control register. The control command is then immediately executed.

5.4.1.1 Control register

The Control register is used to control the non-frequency setting aspects of the ICD2053B. It is an 8-bit register, which is defined as shown in figure 5.9 and 5.10.

7	6	5	4	3	2	1	0
0 (Reserved)	0 (Reserved)	Duty Cycle Adjust (Set to 1)	0 (Reserved)	Pin 7 Usage	MUXREF Control	OE Control	Enable Program Word

Figure 5.9 ICD2053B control register

Bit	Definition
RESERVED	For future use. Set to 0.
Duty Cycle Adjust	Set to 1 to reduce duty cycle by approximately 0.7 ns. Normally set to 1.
Pin 7 Usage	Definition of whether pin 7 is $\overline{MUXREF}$ or OE input pin 0 = Pin 7 is OE input (default) 1 = Pin 7 is $\overline{MUXREF}$ input
MUXREF Control	Allows internal control of MUXREF. If enabled, this feature automatically multiplexes the reference frequency to the CLKOUT output. This is used to change output glitch-free to new frequencies. 0 = CLKOUT is VCO frequency 1 = CLKOUT is $f_{(REF)}$ (default)
OE Control	Forces the CLKOUT output into a three-state mode 0 = CLKOUT is VCO frequency or $f_{(REF)}$ (default) (depending on current MUXREF state) 1 = CLKOUT is three-stated
Enable Program Word	Enable Program word loading into Program register. When enabled, the Program word may be shifted in. This permits changing the Control register without disturbing Program register data. 0 = Program register is disabled from loading (default) 1 = Program register is enabled to receive data

Figure 5.10 ICD2053B control register definition

5.4.1.2 Program register

The Program register can be loaded with a 22-bit programming word, the fields of which are defined in figure 5.11.

Field	# of Bits	Notes
P Counter value (P')	7	MSB (Most Significant Bits)
Duty Cycle Adjust Up (D)	1	Set to logic 1 to increase duty cycle by approx. 0.7 ns. Normally set to 1.
Mux (M)	3	
Q Counter value (Q')	7	
Index (I)	4	LSB (Least Significant Bits)

Figure 5.11 ICD2053B program register

The output clock frequency, CLKOUT, is determined by the following relation:

$$\mathrm{CLKOUT} = {}^{F_{VCO}}\!/_{M}$$

With:

$$F_{VCO} = \left(2 \times f_{(REF)} \times \frac{P}{Q}\right)$$

With:

$$P' = P - 3$$

$$Q' = Q - 2$$

To assist with these calculations, Cypress provides the BITCALC program. BITCALC automatically generates the appropriate values for the Control as well as the Program register according to the desired function and frequency. A copy of BITCALC as well as the complete datasheet are included on the attached CD, under *\DATASHEETS\CLOCK*.

Although using the ICD2053B provides a variable clock signal for the board, there still two problems: the signal is 5V TTL levels and the chip does not have enough output power to drive the clock inputs of all the chips on the board including the DIMMs.

However Quality Semiconductor has released an application note describing how to convert signals from 5V to 3.3V, using their QuickSwitch bus switch family. The application note as well as the datasheet for the QS3L384 are included on the attached CD, under *\DATASHEETS\CLOCK*.

The output limiting characteristics of the QuickSwitch can be used to make a very efficient 5V TTL to 3V TTL converter. By supplying 4.3V to the VCC pin of a QuickSwitch device, the driven output will be limited to 3.3V maximum, even under light loading. A 4.3V VCC is created by adding two 1N4007 diodes between the 5V supply and the device. The diodes will provide approximately 0.70V drop, supplying the QS3L384 with a VCC of 4.3V.

Now that a 3.3V clock is available, the next step is to provide a way to distribute it to all the components on the board. The Texas Instruments CDC516 3.3V Phase Lock Loop Clock Driver was used to provide enough clock lines with adequate driving capabilities. The CDC516 is specifically designed for use with synchronous DRAMs. The CDC516 is organised in four banks of four outputs. Each bank of outputs can be enabled or disabled separately via the 1G, 2G, 3G, and 4G control inputs. When the G inputs are high, the outputs switch in phase and frequency with CLK; when the G inputs are low, the outputs are disabled to the logic-low state. Figure 5.12 shows the function table of the CDC516.

INPUTS					OUTPUTS				
1G	2G	3G	4G	CLK	1Y (0:3)	2Y (0:3)	3Y (0:3)	4Y (0:3)	FBOUT
X	X	X	X	L	L	L	L	L	L
L	L	L	L	H	L	L	L	L	H
L	L	L	H	H	L	L	L	H	H
L	L	H	L	H	L	L	H	L	H
L	L	H	H	H	L	L	H	H	H
L	H	L	L	H	L	H	L	L	H
L	H	L	H	H	L	H	L	H	H
L	H	H	L	H	L	H	H	L	H
L	H	H	H	H	L	H	H	H	H
H	L	L	L	H	H	L	L	L	H
H	L	L	H	H	H	L	L	H	H
H	L	H	L	H	H	L	H	L	H
H	L	H	H	H	H	L	H	H	H
H	H	L	L	H	H	H	L	L	H
H	H	L	H	H	H	H	L	H	H
H	H	H	L	H	H	H	H	L	H
H	H	H	H	H	H	H	H	H	H

Figure 5.12 CDC516 function table

The datasheet for the CDC516 is included on the attached CD, under *\DATASHEETS\CLOCK*.

5.5 Data interface

As with the controller interface the data interface is realised in a FPGA. The same FPGA, the Altera EPF10K30ABC356-3, as is used for the program controller is used. The choice of this FPGA is based on the same reasons as that of the controller interface.

For the Lantaba interface the National DS92LV1210 deserialiser is used to decode the incoming data from the Lantaba camera. For the SUNSAT interface the Texas Instuments SN65LVDS32 high-speed differential line receiver is used to interface with the differential driver on the ground station interface. These signalling techniques reduce the power consumption and increase the signalling speed.

5.6 PCI interface

The AMCC S5933 PCI bus controller is used as an interface with the PCI bus. This device was chosen because of the hands on experience obtained in the development of the prototype data capture card.

Chapter 6

Hardware implementation of final product design

6.1 PCB layout

The layout of the card is very important. In addition to the EMI problems due to the high clock frequencies, the physical track lengths of the clock signals are very important. It is necessary to keep the track lengths of the clock signals of every DIMM equal to ensure that the specifications are met. The schematic capture and the PCB layout were done by Johan Grobbelaar from SED at the University of Stellenbosch. Figure 6.1 shows the diagram of the card with the interconnecting signals.

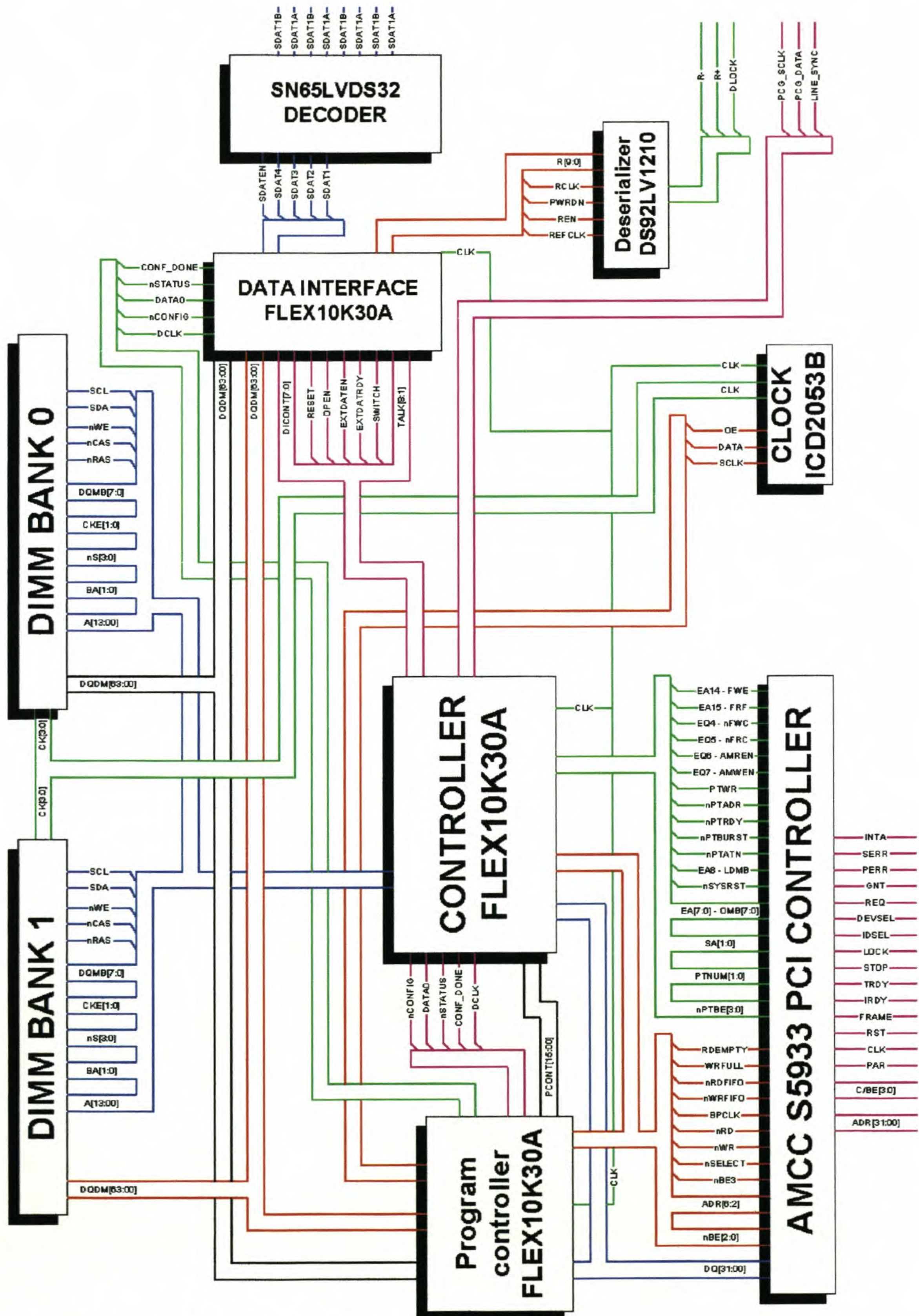


Figure 6.1 Final data capture card diagram

In addition to the control signals needed for the different components table 6.1 lists the additional signals and their descriptions.

Table 6.1 Additional signal descriptions

Signal	**Description**
SWITCH	Switches the camera on and off
EXTDATRDY	Shows when 64-bits of data were received
EXTDATEN	Enable the FPGA for data capturing
OPEN	Output enable
RESET	Resets the FPGA
DICONT[7:0]	Extra control lines
TALK[8:0]	Extra communication lines
PCONT[15:00]	16-bit communications bus

At power up only the program controller FPGA is configured. The other FPGAs will only be configured before the actual data capturing starts. The lines of the unconfigured FPGAs are all in tri-state until the FPGA is configured. It is thus possible for the devices connected to these FPGAs to latch up or go into an undefined state. To ensure that all the components on the board are disabled until accessed by the FPGAs the following lines have resistors connected to them to either pull them up to Vcc or down to ground.

SN65LVDS32 - Deserializer

PWRDWN# and REN are pulled down. Driving these signals low enters the device in power-down mode.

SN65LVDS32 - Line Receive

G# is pulled up. Since G is connected to ground, driving G# high tri-states the output signals.

ICD2053B - Clock generator

MUXREF#/OE is pulled up. The default value of bit 3 of the control register is 0, thus the MUXREF#/OE pin is used as OE. Driving this pin high causes the clock output to be tri-stated.

QS3L384 - Quick Switch

BEA# is pulled up. With BEA# high outputs A[4:0] are disconnected.

CD516 - Clock Driver

G[4:1] are pulled down. Driving G[4:1] low causes the output to stay low, thus disabling the output signals.

DIMM

CKE[1:0] lines are pulled low. Driving the clock enable signal low deactivates the clock signals. Deactivating the clock causes the device to go into power-down mode.

To aid in the debugging of the board some LEDs were added as well as several test points on crucial signal lines.

Part of the PCI specification is that both 5V as well as 3.3V are supposed to be available on the PCI socket. Unfortunately after some measurements were made on several motherboards it was found that very few motherboards available at that stage supplied 3.3V to the PCI bus. The only motherboards that did, were those manufactured by Intel. External power connectors were added to be used if the motherboard did not supply any of the voltages.

The complete schematics and PCB layout of the final card are provided on the attached CD, under *\DATASHEETS\PCB*.

6.2 Population of the card

Most of the components on the board are surface-mount and can be soldered by hand, but it is impossible to solder the BGA components by hand. This meant that the board had to be soldered using the reflow soldering process. With this process soldering paste is applied to the solder pads on the PCB. The components are then carefully placed in the correct position. The whole board is then baked in an oven. This baking process requires that the components must be evenly heated to certain temperatures for certain times to ensure secure connections. These times are determined by the temperature profile of the different components. The temperature profile is how long it takes for the complete chip to reach the required temperature ranges. To determine the temperature profile of the FPGAs a mechanical sample was obtained from ALTERA and this was used to make measurements to determine the temperature profile. Another requirement of this soldering process is that the components are completely moisture free. If any moisture is present, it might cause the component to explode during the baking process. To ensure that the components were moisture free, they were baked at a low temperature for a long time the day before the board was soldered. The temperature profile measurements and soldering were done by Elprom.

Although it is possible to solder boards with surface mount components on both sides with reflow soldering, it was not done with this board because of the cost. When double sided boards are soldered, the components must first be glued to the PCB. This meant that the surface mount components on the solder side of the PCB and the through hole components had still to be done by hand. This was done by Johan Arendse from the ESL laboratory at the University of Stellenbosch.

The board is now complete and ready for the implementation phase.

Chapter 7

Conclusion

As was seen the prototype met the requirements of its design specification completely. With the design and implementation of the prototype a thorough knowledge was obtained of the working of the PCI interface and the coupling to the PCI bus. Thorough knowledge was also obtained in software interfacing to the PCI bus and memory manipulation. Expert knowledge was obtained in the simulation and implementation of designs using VHDL. Simulation is a very powerful tool in the implementation and debugging of a design. Building on this, the final card was designed. The design of the final card provided a series of interesting design challenges. First was component choice. The choices made affected the complete design. The factors of cost, performance and availability had to be considered in all choices. The component choice led to the next obstacle of different signal voltage levels which had to be solved. A study into different memory types provided good knowledge of available memory types and also the implementation and working of DIMMs. In all choices that were made the effect on the performance of the card as a whole had to be kept in mind.

The design meetings that were held definitely reduced the amount of modifications needed to fix design errors and even prevented a catastrophic flaw in the layout. But to err is human and the following errors still slipped through the design checks:
Clock Generation Schematic:
CK0 and CK1, G(4:1) on U13 - CDC516, is pulled up to 3V. It must be pulled down to ground.

CLKCTL4, BEA# on U11 - QS3L384, is pulled down to ground. It must be pulled up to 5V.

The other problem is that the DS92LV1210 deserialiser has the incorrect footprint. This happened because the chip that was ordered as a sample was a different footprint than the one that was received. The PCB layout had to be done before the chips arrived.

With the implementation of the board further problems with the design will unfortunately still be found. Hopefully they will all be on the external layer of the board so that they can be fixed.

Although the high speed data link was one of the few subsystems of SUNSAT I that failed and never worked and thus meant that the card was not needed at that time and that the Lantaba project was dropped due to Government cutbacks. The data capture card was the first design done at the University of Stellenbosch utilising the PCI bus. It also was one of the first digital high speed and mixed supply voltage PCB layouts done at the University. Still today very few PCI designs have been done and there is still a need for a PCI capture card. This design provides an excellent opportunity for an undergraduate student to do his final year design project on or even as a project for a masters student. Although the design was done a while back all the technologies used on the board are still relevant and being used today.

What was also found during the process of the development of the board is that the PCI bus is extremely complex and expensive to incorporate in a design. However there will always be applications where very high speeds are required as well as the functionality that is offered by the PCI bus. For other less complex designs other ways of interfacing with the personal computer are much more cost effective, such as the Universal Serial Bus and the IEEE1394 FireWire.

Appendix A

Mailbox implementation

A1 VHDL

```
--************************************************************************
--***                                                                  ***
--*** MailBox implementation                                           ***
--***                                                                  ***
--*** Ver 1.0                                                          ***
--***                                                                  ***
--*** WJ van der Westhuizen                                            ***
--***                                                                  ***
--************************************************************************

library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.std_logic_arith.all;

entity MailBox is
 Port ( -- AMCC interface
        CLK         : in      std_logic;
        DQ          : inout   std_logic_vector(31 downto 0);
        ADR         : out     std_logic_vector(6 downto 2);
        BE          : out     std_logic_vector(3 downto 0);
        Selct       : out     std_logic;
        RD          : out     std_logic;
        WR          : out     std_logic;
        -- External data interface
        Data_In     : in      std_logic;
        Data_Strb   : in      std_logic;
        DataError   : out     std_logic
     );
end MailBox;

Architecture Architecture_MailBox of MailBox is

-- Data interface signals
signal Data_Counter : integer range 0 to 31;
-- The amount of bits recieved
signal Shift_Reg : std_logic_vector (31 downto 0);
```

```
-- Buffer in which the incoming data is stored
signal Data_Valid    : std_logic := '0';
-- Set when 32 bits was recieved and Shift_Reg moved to data
signal Data_Valid_Flag : std_logic := '1';
-- Used to set DataError and as a condition to read data
-- Reset = 0            => Data_Valid_Flag = 0
-- Data_Valid = 1       => Data_Valid_Flag = 1
-- DataReadFlag = 1           => Data_Valid_Flag = 0
signal Data : std_logic_vector(31 downto 0);
-- Temp. store place for the recieved data
-- AMCC interface signals
signal PCI_Counter  : integer range 0 to 2;
--  PCI clock counter
signal Data_Read : std_logic_vector(31 downto 0);
-- Used to store data read from the AMCC card
signal Read_Flag : std_logic := '0';
-- Set if data must be read from DQ
signal Write_Flag : std_logic := '0';
-- Set if data must be written to DQ
signal MBox_E_F   : std_logic := '0';
-- Used to see when data was read from Mailbox
-- When bits in Data_Read is set MBox_E_F is set
signal Data_Read_Flag     : std_logic := '0';
-- Set when data was read
signal Fin_Read : std_logic := '0';
-- Set when data was read
signal Reset : std_logic := '0';
--  Used to initialize the signals

begin

--*************************************************************
--***                                                      ***
--***  Counts the incoming bits to packets of 32 bits      ***
--***                                                      ***
--*************************************************************

Data_Strobe_Counter:  process

begin
 wait until Data_Strb'event and Data_Strb = '1';
 Data_Counter <= Data_Counter +1;
end process Data_Strobe_Counter;
```

```
--*****************************************************************
--***                                                          ***
--***  Shifts the incoming data into a 32 bit register         ***
--***                                                          ***
--*****************************************************************

Shift_Register: process

variable i : integer range 0 to 31;

begin
  wait until Data_Strb'event and Data_Strb = '1';
  for i in 31 downto 1 loop
    Shift_Reg(i) <= Shift_Reg(i-1);
  end loop;
  Shift_Reg(0) <= Data_In;
end process Shift_Register;

--***********************************************************************
--***                                                               ***
--***  Sets Data_Valid and moves recieved data to Data              ***
--***                                                               ***
--***********************************************************************

Data_Output: process

begin
  wait until Data_Strb'event and Data_Strb = '1';
  if Data_Counter = 0 then
    Data_Valid <= '1';
    Data <= Shift_Reg;
  else
    Data_Valid <= '0';
  end if;
end process Data_Output;

--********************************************************
--***                                                ***
--*** Die deel stuur data na die AMCC kaart          ***
--***                                                ***
--********************************************************

--******************************************************************************
--***                                                                      ***
--***  Checks Data_Valid and Data_Valid Flag                               ***
--***  If previous data was not read by the PC sets DataError ***
```

```
--***                                                                   ***
--***************************************************************************

Data_Overflow: process

begin
 wait until Data_Valid'event and Data_Valid = '1';
 if Data_Valid_Flag = '1' then
  DataError <= '1';
 else
  DataError <= '0';
 end if;
end process Data_Overflow;

--**************************************************************
--***                                                       ***
--***  Generates a Reset signal                             ***
--***                                                       ***
--**************************************************************

Rset: process

begin
 wait until Data_Strb'event and Data_Strb = '1';
 Reset <= '1';
end process Rset;

--**************************************************************
--***                                                       ***
--***  Counts the PCI clock                                 ***
--***                                                       ***
--**************************************************************

Clock_Counter: process

begin
 wait until clk'event and clk = '1';
 if PCI_Counter = 2 then
  PCI_Counter <= 0;
 else
  PCI_Counter <= PCI_Counter + 1;
 end if;
end process Clock_Counter;

--*********************************************************************
--***                                                              ***
--***  Initializes Data_Valid with Reset                           ***
```

```
--***  Sets Data_Valid_Flag if Data_Valid is set           ***
--***  Clears Data_Valid_Flag if Data_Read_Flag is set         ***
--***                                                           ***
--***************************************************************************

Data_Valid_Process: process

begin
 if Reset = '1' then
  if Data_Valid = '1' then
   Data_Valid_Flag <= '1';
  elsif Data_Read_Flag = '1' then
   Data_Valid_Flag <= '0';
  else
   Data_Valid_Flag <= Data_Valid_Flag;
  end if;
 else
  Data_Valid_Flag <= '0';
 end if;
end process Data_Valid_Process;

--*******************************************************************
--***                                                           ***
--***  Read or write to DQ                                      ***
--***                                                           ***
--*******************************************************************

Set_DQ: process

begin
 if Read_Flag = '1' then
  DQ <=  (others =>'Z');
  Data_Read <= DQ;
 elsif Write_Flag = '1' then
  DQ <= data;
 else
  DQ <= (others =>'Z');
 end if;
end process Set_DQ;

--*******************************************************************
--***                                                           ***
--***  When Fin_Read event then MBox_E_F is set                 ***
--***                                                           ***
--*******************************************************************

MBEF: process
```

```
begin
  wait until Fin_Read'event and Fin_Read = '1';
  if (Data_Read(19) = '1') or (Data_Read(18) = '1') or
     (Data_Read(17) = '1') or (Data_Read(16) = '1')   then
    MBox_E_F <= '0';
  else
    MBox_E_F <= '1';
  end if;
end process MBEF;

--************************************************************************
--***                                                                 ***
--***  Checks the AMBEF register and reads data from AIMB                ***
--***                                                                 ***
--************************************************************************

Data_IO: process

begin
  wait until clk'event and clk = '1';
  if MBox_E_F = '0' or Fin_Read = '0' then
    Data_Read_Flag <= '0';
    case PCI_Counter is
    when 0 =>
      Be <= "0000";
      ADR <= "01101";
      Selct <= '0';
      WR <= '1';
      RD <= '0';
      Read_Flag <= '0';
      Write_Flag <= '0';
      Fin_Read <= '0';
    when 1 =>
      Read_Flag <= '1';
      Write_Flag <= '0';
      Fin_Read <= '0';
    when 2 =>
      Selct <= '1';
      RD <= '1';
      WR <= '1';
      Be <= "1111";
      ADR <= "00000";
      Read_Flag <= '0';
      Write_Flag <= '0';
      Fin_Read <= '1';
    end case;
  elsif MBox_E_F = '1' and Fin_Read = '1' and Data_Valid_Flag = '1' then
```

```
    case PCI_Counter is
    when 0 =>
      Data_Read_Flag <= '0';
      Be <= "0000";
      ADR <= "00100";
      Selct <= '0';
      WR <= '0';
      RD <= '1';
      Read_Flag <= '0';
      Write_Flag <= '1';
      Fin_Read <= '1';
    when 1 =>
      Data_Read_Flag <= '0';
      WR <= '1';
      Selct <= '0';
      Read_Flag <= '0';
      Write_Flag <= '1';
      Fin_Read <= '1';
    when 2 =>
      Selct <= '1';
      WR <= '1';
      RD <= '1';
      Be <= "1111";
      ADR <= "00000";
      Data_Read_Flag <= '1';
      Read_Flag <= '0';
      Write_Flag <= '0';
      Fin_Read <= '0';
    end case;
  else
    Data_Read_Flag <= '0';
    Selct <= '1';
    WR <= '1';
    RD <= '1';
    Be <= "1111";
    ADR <= "00000";
    Read_Flag <= '0';
    Write_Flag <= '0';
    Fin_Read <= '0';
  end if;
end process Data_IO;

end Architecture_MailBox;
```

The VHDL file is provided on the attached CD, under *\VHDL\MAILBOX*.

A2 Pinout

Table A.1 Mailbox implementation pinout

Signal	**Pin number**
ADR2	72
ADR3	71
ADR4	70
ADR5	69
ADR6	67
BE0	13
BE1	9
BE2	8
BE3	7
CLK	12
Data_In	31
Data_Strb	54
DataError	1
DQ0	2
DQ1	15
DQ2	16
DQ3	18
DQ4	19
DQ5	21
DQ6	22
DQ7	23
DQ8	24
DQ9	25
DQ10	34

DQ11	35
DQ12	37
DQ13	39
DQ14	40
DQ15	41
DQ16	42
DQ17	43
DQ18	44
DQ19	45
DQ20	46
DQ21	79
DQ22	81
DQ23	82
DQ24	83
DQ25	84
DQ26	64
DQ27	63
DQ28	62
DQ29	61
DQ30	60
DQ31	58
RD	57
Selct	66
WR	56

Appendix B

FIFO implementation

B1 VHDL

```
--*******************************************************************
--***                                                           ***
--***  FIFO implementation                                      ***
--***                                                           ***
--***  Ver 1.0                                                  ***
--***                                                           ***
--***  WJ van der Westhuizen                                    ***
--***                                                           ***
--*******************************************************************

library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.std_logic_arith.all;

entity MBFIFO is
  Port ( Clk          : in    std_logic;
         DQ           : inout std_logic_vector(31 downto 0);
        ADR           : out   std_logic_vector(6 downto 2);
         BE           : out   std_logic_vector(3 downto 0);
         Selct        : out   std_logic;
         RD           : out   std_logic;
         WR           : out   std_logic;
         -- FIFO
         WRFULL              : in    std_logic;
         RDFIFO              : out   std_logic;
         WRFIFO              : out   std_logic;
         -- Incoming data
         Data_In    : in   std_logic;
         Data_Strb  : in    std_logic;
         Status  : out   std_logic
      );
end MBFIFO;
```

```
Architecture Architecture_MBFIFO of MBFIFO is

-- Data Interface
signal Data_Counter : integer range 0 to 31;
-- The amount of bits recieved
signal Data : std_logic_vector(31 downto 0);
-- Temp. store place for the recieved data
signal Shift_Reg : std_logic_vector (31 downto 0);
-- Buffer in which the incoming data is stored
signal Data_Valid : std_logic;
-- Set when 32 bits was recieved
-- FIFO interface
signal FIFO_Counter : integer range 0 to 3;
--  Data_Valid counter
--  Data_Valid Sync
signal Data_Valid_Sync_Reset : std_logic;
--  Used as a clear signal
signal Data_Valid_Async_Out : std_logic;
--  Used to generate the sync signal
signal Data_Valid_Sync : std_logic;
-- Sync version of Data_Valid
signal Data_Read : std_logic_vector(31 downto 0);
-- Used to store data read from the AMCC card
signal Read_Flag : std_logic := '0';
-- Set if data is read from DQ
signal Write_Flag : std_logic := '0';
-- Set if data is written to DQ
--  Signals to set Read_Flag and Write_Flag
signal DQ_FIFO : std_logic;
--  Set when a FIFO write transactions is performed
signal DQ_AIMB : std_logic;
--  Set when a mailbox status register read transaction is performed
signal DQ_AOMB : std_logic;
--  Set when a mailbox write transaction is performed
--  FIFO
signal Write_FIFO : std_logic;
--  Set to write data to FIFO
signal Read_FIFO : std_logic;
--  Set to read data from FIFO
signal FIFO_Ready : std_logic;
--  Set when 4 * 32Bit data have been recieved
signal FIFO_Ready_Sync : std_logic;
--  FIFO_Ready synced to Clk
--  FIFO_Ready Sync
signal FIFO_Ready_Sync_Reset : std_logic;
```

```
-- Used as a clear signal
signal FIFO_Ready_Async_Out : std_logic;
-- Used to generate the sync signal
-- State machines
signal Reset : std_logic;
-- Sets state machine to a default known state
signal Not_Fifo : std_logic;
-- Set when finished accessing the FIFO
signal Data_Fin_Read : std_logic;
-- Set when finished checking AMEF
-- Read/Write state machine
type FIFOState_Type is (FIFO_DefaultState, FIFO_s1, FIFO_s2, FIFO_s3, FIFO_s4);
-- Defines the possible states of the state machine
signal FIFOState : FIFOState_Type;
-- The current state of the state machine
-- AIMB state machine
type AIMBState_Type is (AIMB_DefaultState, AIMB_s1, AIMB_s2);
-- Defines the possible states of the state machine
signal AIMBState : AIMBState_Type;
-- The current state of the state machine
-- AOMB state machine
type AOMBState_Type is (AOMB_DefaultState, AOMB_s1, AOMB_s2);
-- Defines the possible states of the state machine
signal AOMBState : AOMBState_Type;
-- The current state of the state machine
-- Signals to determine the state of Selct, RD, WR, Be and ADR
-- Vir AIMB toegange
signal AIMB_Selct : std_logic;
signal AIMB_RD : std_logic;
signal AIMB_WR : std_logic;
signal AIMB_Be : std_logic;
signal AIMB_ADR : std_logic;
-- Vir AOMB toegange
signal AOMB_Selct : std_logic;
signal AOMB_RD : std_logic;
signal AOMB_WR : std_logic;
signal AOMB_Be : std_logic;
signal AOMB_ADR : std_logic;
signal Make_Z   : std_logic;
```

```
begin

--************************************************************************
--***                                                                  ***
--*** The following part handles the incoming serial data              ***
--***                                                                  ***
--************************************************************************

--****************************************************************
--***                                                          ***
--*** Counts the incoming bits to packets of 32 bits           ***
--***                                                          ***
--*** Pulse Data_Valid when Data_Counter = 0                   ***
--***                                                          ***
--****************************************************************

Data_Clock_Counter: process

begin
  wait until Data_Strb'event and Data_Strb = '1';
  if Data_Counter = 0 then
    Data_Counter <= Data_Counter +1;
         Data_Valid <= '1';
  else
    Data_Counter <= Data_Counter +1;
    Data_Valid <= '0';
  end if;
end process Data_Clock_Counter;

--****************************************************************
--***                                                          ***
--*** Sync Data_Valid with clock (Data_Valid_Sync)             ***
--***                                                          ***
--****************************************************************

DataValid_Async_In : process(Data_Valid_Sync_Reset, Data_Valid)

begin
  if Data_Valid_Sync_Reset = '1' then
    Data_Valid_Async_Out <= '0';
  elsif Data_Valid'event and Data_Valid = '1' then
    Data_Valid_Async_Out <= '1';
  end if;
end process DataValid_Async_In;

DataValid_Sync_Out : process
begin
```

```
 wait until Clk'event and Clk = '1';
 if  Data_Valid_Async_Out = '1' then
       Data_Valid_Sync_Reset <= '1';
  Data_Valid_Sync <= '1';
 else
  Data_Valid_Sync_Reset <= '0';
  Data_Valid_Sync <= '0';
 end if;
end process DataValid_Sync_Out;

--*****************************************************************
--***                                                           ***
--***  Shifts the incoming data into a 32 bit register          ***
--***                                                           ***
--*****************************************************************

Shift_Register: process

variable i : integer range 0 to 31;

begin
 wait until Data_Strb'event and Data_Strb = '1';
 for i in 31 downto 1 loop
  Shift_Reg(i) <= Shift_Reg(i-1);
 end loop;
 Shift_Reg(0) <= Data_In;
end process Shift_Register;

--*****************************************************************
--***                                                           ***
--***  Moves recieved data to Data                              ***
--***                                                           ***
--*****************************************************************

Data_Output: process

begin
 wait until Data_Strb'event and Data_Strb = '1';
 if Data_Counter = 0 then
  Data <= Shift_Reg;
 end if;
end process Data_Output;

--************************************************************************
--***                                                                  ***
--***  The follwing part handles the communications with the           ***
--***  AMCC PCI card                                                   ***
```

```
--***                                                                    ***
--************************************************************************

--*****************************************************************
--***                                                           ***
--***  Counts DataValid                                         ***
--***                                                           ***
--***                                                           ***
--*****************************************************************

Not_FIFO_Counter: process

begin
  wait until Not_Fifo'event and Not_Fifo = '0';
  if FIFO_Counter = 3 then
    FIFO_Counter <= 0;
    FIFO_Ready <= '0';
  elsif FIFO_Counter = 0 then
    FIFO_Counter <= FIFO_Counter + 1;
    FIFO_Ready <= '1';
  else
    FIFO_Counter <= FIFO_Counter + 1;
    FIFO_Ready <= '0';
  end if;
end process Not_FIFO_Counter;

--*****************************************************************
--***                                                           ***
--***  Sync FIFO_Ready with Clk(FIFO_Ready_Sync)                ***
--***                                                           ***
--*****************************************************************

Fifo_Async_In : process(FIFO_Ready_Sync_Reset, FIFO_Ready)

begin
  if FIFO_Ready_Sync_Reset = '1' then
    FIFO_Ready_Async_Out <= '0';
  elsif FIFO_Ready'event and FIFO_Ready = '1' then
    FIFO_Ready_Async_Out <= '1';
  end if;
end process Fifo_Async_In;

Fifo_Sync_Out : process
begin
  wait until Clk'event and Clk = '1';
  if  FIFO_Ready_Async_Out = '1' then
        FIFO_Ready_Sync_Reset <= '1';
```

```
  FIFO_Ready_Sync <= '1';
 else
  FIFO_Ready_Sync_Reset <= '0';
  FIFO_Ready_Sync <= '0';
 end if;
end process Fifo_Sync_Out;

--*****************************************************************
--***                                                           ***
--***  Generates a reset signal                                 ***
--***                                                           ***
--*****************************************************************

res: process

begin
 wait until Data_Strb'event and Data_Strb = '1';
 reset <= '1';
end process res;

--*****************************************************************
--***                                                           ***
--***  FIFO state machine                                       ***
--***                                                           ***
--*****************************************************************

FIFO_State_Machine: process(Clk)

begin
 if reset = '0' then
  FIFOState <= FIFO_DefaultState;
 elsif (Clk'event and Clk = '1') then
  case FIFOState is
      when FIFO_DefaultState =>
       if Data_Fin_Read = '1' then
        FIFOState <= FIFO_s1;
       else
        FIFOState <= FIFO_DefaultState;
       end if;
      when FIFO_s1 =>
       if (Data_Read(0) = '1') and (Data_Read(1) = '0') and
      (Data_Read(2) = '1') and (Data_Read(3) = '0') then
        FIFOState <= FIFO_s2;
        Status <='0';
       else
        FIFOState <= FIFO_DefaultState;
        Status <='1';
```

```
            end if;

          when FIFO_s2 =>
            FIFOState <= FIFO_s3;
          when FIFO_s3 =>
            FIFOState <= FIFO_s4;
          when FIFO_s4 =>
            FIFOState <= FIFO_DefaultState;
        end case;
  end if;

end process FIFO_State_Machine;

FIFO_Signals: process

begin
  wait until Clk'event and Clk = '1';
  case FIFOState is
    when FIFO_DefaultState =>
      -- Puts DQ in tri-state
      DQ_FIFO <= '0';
      -- Prevents any FIFO transactions
      Read_FIFO <= '0';
      Write_FIFO <= '0';
      Not_Fifo <= '1';
    when FIFO_s1 =>
      -- Puts DQ in tri-state
      DQ_FIFO <= '0';
      -- Prevents any FIFO transactions
      Read_FIFO <= '0';
      Write_FIFO <= '0';
      Not_Fifo <= '1';
    when FIFO_s2 =>
      -- Setup to read or write to DQ
      DQ_FIFO <= '1';
      -- Setup to read or write to FIFO
      Read_FIFO <= '0';
      Write_FIFO <= '0';
      Not_Fifo <= '1';
      when FIFO_s3 =>
     -- Setup to read or write to DQ
      DQ_FIFO <= '1';
      -- Setup to read or write to FIFO
      Read_FIFO <= '0';
      Write_FIFO <= '1';
      Not_Fifo <= '1';
  when FIFO_s4 =>
```

```
    -- Setup to read or write to DQ
    DQ_FIFO <= '1';
    -- Setup to read or write to FIFO
    Read_FIFO <= '0';
    Write_FIFO <= '0';
    Not_Fifo <= '0';
  end case;

end process FIFO_Signals;

--*****************************************************************
--***                                                           ***
--***  Read or write to FIFO                                    ***
--***                                                           ***
--*****************************************************************

Set_wrfifo_and_rdfifo: process

begin
  if Write_FIFO = '1' then
   WRFIFO <= '0';
   RDFIFO <= '1';
  elsif Read_FIFO = '1' then
   WRFIFO <= '1';
   RDFIFO <= '0';
  else
   WRFIFO <= '1';
   RDFIFO <= '1';
  end if;
end process Set_wrfifo_and_rdfifo;

--*****************************************************************
--***                                                           ***
--***  Read or write to DQ                                      ***
--***                                                           ***
--*****************************************************************

RW_DQ: process

begin
  if DQ_FIFO = '1' then
--  Set when a FIFO write transactions is performed
   Read_Flag <= '0';
   Write_Flag <= '1';
  elsif DQ_AIMB = '1' then
--  Set when a mailbox status register read transaction is performed
   Read_Flag <= '1';
```

```
    Write_Flag <= '0';
  elsif DQ_AOMB = '1' then
-- Set when a mailbox write transaction is performed
    Read_Flag <= '0';
    Write_Flag <= '1';
  else
    Read_Flag <= '0';
    Write_Flag <= '0';
  end if;

end process RW_DQ;

Set_DQ: process

begin
  if Read_Flag = '1' then
    Make_Z <= '1';
    Data_Read <= DQ;
  elsif Write_Flag = '1' then
    Make_Z <= '0';
  else
    Make_Z <= '1';
  end if;
end process Set_DQ;

Set_Z: process (Make_Z, Data)
begin
  if Make_Z = '1' then
    DQ <= (others =>'Z');
  else
    DQ <= Data;
  end if;
end process Set_Z;

--*****************************************************************
--***                                                           ***
--***  State machine that reads the AIMB1                       ***
--***                                                           ***
--*****************************************************************

AIMB_State_Machine: process(Clk)

begin
  if reset = '0' then
    AIMBState <= AIMB_DefaultState;
  elsif (Clk'event and Clk = '1') then
    case AIMBState is
```

```
          when AIMB_DefaultState =>
            if Data_Valid_Sync = '1' then
              AIMBState <= AIMB_s1;
            else
              AIMBState <= AIMB_DefaultState;
            end if;
          when AIMB_s1 =>
            AIMBState <= AIMB_s2;
          when AIMB_s2 =>
            AIMBState <= AIMB_DefaultState;
    end case;
  end if;
end process AIMB_State_Machine;

AIMB_Signals: process

begin
  wait until Clk'event and Clk = '1';
  case AIMBState is
    when AIMB_DefaultState =>
      AIMB_Selct <= '1';
      AIMB_WR <= '1';
      AIMB_RD <= '1';
      AIMB_BE <= '1';
      AIMB_ADR <= '0';
      DQ_AIMB <= '0';
      Data_Fin_Read <= '0';
    when AIMB_s1 =>
      AIMB_Selct <= '0';
      AIMB_WR <= '1';
      AIMB_RD <= '0';
      AIMB_BE <= '0';
      AIMB_ADR <= '1';
      DQ_AIMB <= '0';
      Data_Fin_Read <= '0';
    when AIMB_s2 =>
      AIMB_Selct <= '0';
      AIMB_WR <= '1';
      AIMB_RD <= '0';
      AIMB_BE <= '0';
      AIMB_ADR <= '1';
      DQ_AIMB <= '1';
      Data_Fin_Read <= '1';
    end case;
end process AIMB_Signals;

--*******************************************************************************
```

```
--***                                                              ***
--***  State machine that writes data to the AOMB to indicate      ***
--***  to the PC that 4 * 32Bit data was written to the FIFO       ***
--***                                                              ***
--*****************************************************************************

AOMB_State_Machine: process(Clk)

begin
  if reset = '0' then
    AOMBState <= AOMB_DefaultState;
  elsif (Clk'event and Clk = '1') then
    case AOMBState is
            when AOMB_DefaultState =>
              if FIFO_Ready_Sync = '1' then
                AOMBState <= AOMB_s1;
              else
                AOMBState <= AOMB_DefaultState;
              end if;
            when AOMB_s1 =>
              AOMBState <= AOMB_s2;
            when AOMB_s2 =>
              AOMBState <= AOMB_DefaultState;
    end case;
  end if;
end process AOMB_State_Machine;

AOMB_Signals: process

begin
  wait until Clk'event and Clk = '1';
  case AOMBState is
    when AOMB_DefaultState =>
      AOMB_Selct <= '1';
      AOMB_WR <= '1';
      AOMB_RD <= '1';
      AOMB_BE <= '1';
      AOMB_ADR <= '0';
      DQ_AOMB <= '0';
    when AOMB_s1 =>
      AOMB_Selct <= '0';
      AOMB_WR <= '0';
      AOMB_RD <= '1';
      AOMB_BE <= '0';
      AOMB_ADR <= '1';
      DQ_AOMB <= '1';
    when AOMB_s2 =>
```

```
    AOMB_Selct <= '0';
    AOMB_WR <= '1';
    AOMB_RD <= '1';
    AOMB_BE <= '0';
    AOMB_ADR <= '1';
    DQ_AOMB <= '1';
  end case;
end process AOMB_Signals;

--*****************************************************************
--***                                                           ***
--***    Sets Selct                                             ***
--***                                                           ***
--*****************************************************************

Set_Selct: process

begin
 if AIMB_Selct = '0' then
  Selct <= '0';
 elsif AOMB_Selct = '0' then
  Selct <= '0';
 else
  Selct <= '1';
 end if;
end process Set_Selct;

--*****************************************************************
--***                                                           ***
--***    Sets WR                                                ***
--***                                                           ***
--*****************************************************************

Set_WR: process

begin
 if AIMB_WR = '0' then
  WR <= '0';
 elsif AOMB_WR = '0' then
  WR <= '0';
 else
  WR <= '1';
 end if;
end process Set_WR;

--*****************************************************************
--***                                                           ***
```

```
--***    Sets RD                                              ***
--***                                                         ***
--*****************************************************************

Set_RD: process

begin
 if AIMB_RD = '0' then
   RD <= '0';
 elsif AOMB_RD = '0' then
   RD <= '0';
 else
   RD <= '1';
 end if;
end process Set_RD;

--*****************************************************************
--***                                                         ***
--***    Sets BE                                              ***
--***                                                         ***
--*****************************************************************

Set_BE: process

begin
 if  AIMB_Be = '0' then
   BE <= "0000";
 elsif AOMB_BE = '0' then
   BE <= "0000";
 else
   BE <= "1111";
 end if;
end process Set_BE;

--*****************************************************************
--***                                                         ***
--***    Sets ADR                                             ***
--***                                                         ***
--*****************************************************************

Set_ADR: process

begin
 if AIMB_ADR = '1' then
   ADR <= "00000";
 elsif AOMB_ADR = '1' then
```

```
    ADR <= "00100";
  else
    ADR <= "00000";
  end if;
end process Set_ADR;

end Architecture_MBFIFO;
```

The VHDL file is provided on the attached CD, under *\VHDL\FIFO*.

B2 Pinout

Table B.1 FIFO implementation pinout

Signal	Pin number
ADR2	72
ADR3	71
ADR4	70
ADR5	69
ADR6	67
BE0	13
BE1	9
BE2	8
BE3	7
CLK	12
Data_In	31
Data_Strb	54
DataError	1
DQ0	2
DQ1	15
DQ2	16
DQ3	18

DQ4	19
DQ5	21
DQ6	22
DQ7	23
DQ8	24
DQ9	25
DQ10	34
DQ11	35
DQ12	37
DQ13	39
DQ14	40
DQ15	41
DQ16	42
DQ17	43
DQ18	44
DQ19	45
DQ20	46
DQ21	79
DQ22	81
DQ23	82
DQ24	83
DQ25	84
DQ26	64
DQ27	63
DQ28	62
DQ29	61
DQ30	60
DQ31	58
RD	57

Selct	66
WR	56

Appendix C

Software

B1 AMCC.CPP

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#include <except.h>
#include <iostream.h>

unsigned long XMSDriverAddr;                    // Address for XMS function calls
unsigned int  XMSHandle;
unsigned long BufferLinAddr;
unsigned long DataCounter;
int CaptureResult;
// Data funcs
extern int Capture();
// Mem funcs
extern int InProtected();
extern int XMSInit();
extern int LocalA20Enable();
extern int LocalA20Disable();
extern int XMSQueryFree();
extern int XMSAlloc(unsigned int Size);
extern int XMSFreeMem(unsigned int Handle);
extern unsigned long XMSLockMem(unsigned int Handle);
extern int XMSUnLockMem(unsigned int Handle);
extern void EnableFlatReal();
extern int TestMode();
extern void MemToHD(unsigned char far *dataarea);
typedef unsigned char dword;                    /* 32-bit */
dword far *testarr;
char DATStr[128];

void ConToBin(unsigned long dat)
{
 stpcpy(DATStr,"");
 unsigned long bit;
 bit=1;
 bit = bit<<31;
```

```
 for (int x=0;x<32;x++) {
 if ((dat&bit)!=0)
{
   strcat(DATStr,"1");
}
 else
 strcat(DATStr,"0");
 bit = bit >> 1;
}
}

main()
{
  DataCounter = 0;
  if (XMSInit())
   {
   printf("XMSInit returned True - XSMDriver OK\n");
   }
  else
   {
   printf("XMSInit returned False - XMSDriver not found\n");
   exit(0);
   }
  cout << "Free memory : " << dec << XMSQueryFree() << endl;
  if (InProtected())
  {
   printf("Processor in protected mode!\n");
   exit(0);
  }
  else
  {
   printf("Processor not in protected mode\n");
  }
  cout << "LocalA20Enable :" << hex << LocalA20Enable() << endl;
  EnableFlatReal();
  printf("Flat Real Mode Enabled!\n");
  XMSHandle = XMSAlloc(1024);                    //1024 for 1 meg
                                                 // 31744 for 31 meg
  cout << "Allocated 31M : Handle = " << dec << XMSHandle << endl;
  cout << "Free memory : " << dec << XMSQueryFree() << endl;
  printf("Locking Memory.\n");
  BufferLinAddr = XMSLockMem(XMSHandle);
  cout << "BufferLinAddr :" << dec << BufferLinAddr << endl;
  printf("Testing the memory...\n");
  if (TestMode())
   printf("All went OK!\n");
```

```
  else
    printf("Error in testing\n");
  printf("Capturing data\n");
  CaptureResult = Capture();
  if (CaptureResult != 0)
  {
    printf("Error capturing data!!\n");
    cout << "CaptureResult :" << hex << CaptureResult << endl;
    exit(0);
  }
// file stuff
  FILE *stream;
  testarr = new dword[4096];
  if (testarr == NULL)
  {
    cout << "ERROR"  << endl;
    exit(0);
  }
  printf("Maak leer op HD oop.\n");
/* open a file for update */
  stream = fopen("DUMMY.FIL", "w+");
  printf("File open:\n");
  for (int j = 0; j < 64; j++)                    // j * 16 384bytes = amount saved
                                                  //1984 for 31 meg
  {                                               //64 for 1 meg
    printf("Int array\n");
    for (int i = 0; i < 4096; i++)
    {
     testarr[i] = 0;
    }
    printf("Array int fin\n");
    printf("$#@@#$ Read data from memory and write to HD $#@@#$\n");
    MemToHD(testarr);
    cout << "Saved :" << dec << DataCounter << " Bytes" << endl;
    for (int i = 0; i < 4096; i++)
    {
/* write some data to the file */
      ConToBin(testarr[i]);
      fprintf(stream, "%s" ,DATStr);              // \n for new line
    } // end for
  }
  fclose(stream);
  delete [] testarr;
  printf("FIN\n");
  cout << "Unlocking Memory : " << XMSUnLockMem(XMSHandle) << endl;
  cout << "Freeing 8M : " << XMSFreeMem(XMSHandle) << endl;
  cout << "Free memory : " << dec << XMSQueryFree() << endl;
```

```
  cout << "LocalA20Disable :" << hex << LocalA20Disable() << endl;
}
```

B2 FLAT.ASM

```
.model LARGE
.386p

.data

EXTRN _XMSDriverAddr : DWORD;
EXTRN _BufferLinAddr : DWORD;
EXTRN _DataCounter      : DWORD;

readcount    dd            0

LoadGDT DW 3 DUP (?)

GDT   DW  0000h, 0000h, 0000h, 0000h
      DW 0FFFFh, 0000h, 9200h, 0FFCFh

.code

;*************************************************************************
;************                                              ************
;************                   XMSInit                    ************
;************                                              ************
;*************************************************************************
PUBLIC @XMSInit$qv
;Test if an XMS driver is present and get the API call address
;This is called from PASCAL as a function that returns a Boolean
@XMSInit$qv PROC FAR
  mov   ax,   4300h                          ;Verify if there is an XMS driver
  int   2Fh
  cmp   al,   80h
  je    XMSFound
  jmp   XMSError
XMSFound:
  mov   ax,   4310h                          ;Get XMS driver call address
  int   2Fh
  mov   word ptr _XMSDriverAddr[0],   bx
  mov   word ptr _XMSDriverAddr[2],   es
  mov   ah,   0                              ;Get XMS version info
  call  dword ptr [_XMSDriverAddr]
  cmp   ah,   02h                            ;Must be major version 2 or above
```

```
  jb   XMSError
  xor  dx,  dx
  mov  ax,   01h                          ;Return true
  ret
XMSError:
  xor  dx,  dx
  xor  ax,   ax                           ;Return False
  ret
@XMSInit$qv ENDP
;****************************************************************************
;************                                              ************
;************            InProtected                       ************
;************                                              ************
;****************************************************************************
PUBLIC @InProtected$qv
;Test if the system is already in protected mode (including V86 mode)
@InProtected$qv PROC FAR
  mov  eax,  cr0
  and  al,  1                             ;Isolate Protected Mode bit
  ret
@InProtected$qv ENDP
;****************************************************************************
;************                                              ************
;************            LocalA20Enable                    ************
;************                                              ************
;****************************************************************************
PUBLIC @LocalA20Enable$qv
;Enable the A20 line.  Needed to be able to address mem above 1M
@LocalA20Enable$qv PROC FAR
  mov  ah,  05h                           ;Enable local A20 line
  call dword ptr [_XMSDriverAddr]
  ret
@LocalA20Enable$qv ENDP
;****************************************************************************
;************                                              ************
;************        LocalA20Disable                       ************
;************                                              ************
;****************************************************************************
PUBLIC @LocalA20Disable$qv
;Disable the A20 line.  Must be done before we leave the program
@LocalA20Disable$qv PROC FAR
  mov  ah,  06h                           ;Disable local A20 line
  call dword ptr [_XMSDriverAddr]
  ret
@LocalA20Disable$qv ENDP
;****************************************************************************
;************                                              ************
```

```
;************          XMSQueryFree                ************
;************                                      ************
;*****************************************************************
PUBLIC @XMSQueryFree$qv
;Queries the amount of free XMS memory
@XMSQueryFree$qv PROC FAR
  mov  ah,   08h
  call dword ptr [_XMSDriverAddr]
  mov  ax,   dx                        ;Move K free into AX for return
  ret
@XMSQueryFree$qv ENDP
;*****************************************************************
;************                                      ************
;************          XMSAlloc                    ************
;************                                      ************
;*****************************************************************
PUBLIC @XMSAlloc$qui
;Allocate XMS memory. The amount (in K) to allocate is passed on the stack
@XMSAlloc$qui PROC FAR
  push bp
  mov  bp,   sp
  mov  dx,   [bp+6]                    ;Get amount to allocate
  mov  ah,   09h
  call dword ptr [_XMSDriverAddr]
  cmp  ax,   0000h
  je   AllocError
  mov  ax,   dx
  pop  bp
  ret
AllocError:
  mov  ax,   0FFFFh                    ;Indicate Error
  pop  bp
  ret
@XMSAlloc$qui ENDP
;*****************************************************************
;************                                      ************
;************          XMSFreeMem                  ************
;************                                      ************
;*****************************************************************
PUBLIC @XMSFreeMem$qui
;Free a chunk of XMS memory
@XMSFreeMem$qui PROC FAR
  push bp
  mov  bp,   sp
  mov  dx,   [bp+6]                    ;Get Handle of memory to free
  mov  ah,   0Ah
  call dword ptr [_XMSDriverAddr]
```

```
  pop   bp
  ret
@XMSFreeMem$qui ENDP
;*************************************************************************
;************                                            ************
;************          XMSLockMem                        ************
;************                                            ************
;*************************************************************************
PUBLIC @XMSLockMem$qui
;Lock a previously allocated chunck of memory.  Stops it from being
;moved around.  It also returns you the physical address of the memory
@XMSLockMem$qui PROC FAR
  push bp
  mov   bp,    sp
  mov   dx,    [bp+6]                    ;Get Handle from stack
  mov   ah,    0Ch                       ;Lock memory
  call  dword ptr [_XMSDriverAddr]
  cmp   ax,    01h
  je    cool
  mov   ax,    0FFFFh
  pop   bp
  ret
cool:
  mov   ax,    bx
  pop   bp
  ret
@XMSLockMem$qui ENDP
;*************************************************************************
;************                                            ************
;************          MSUnLockMem                       ************
;************                                            ************
;*************************************************************************
PUBLIC @XMSUnLockMem$qui
;Unlocks the block of memory previously locked by XMSLockMem
@XMSUnLockMem$qui PROC FAR
  push bp
  mov   bp,    sp
  mov   dx,    [bp+6]                    ;Get handle from stack
  mov   ah,    0Dh
  call  dword ptr [_XMSDriverAddr]
  pop   bp
  ret
@XMSUnLockMem$qui ENDP
;*************************************************************************
;************                                            ************
;************          EnableFlatReal                    ************
;************                                            ************
```

```
;***************************************************************************
PUBLIC @EnableFlatReal$qv
;This procedure sets up the flat real mode. It loads the minimal GDT
;and switches to PM, then it loads the ES, FS and GS registers with
;the 4MB limit selector
@EnableFlatReal$qv PROC FAR
  mov   word ptr LoadGDT[0],   16                  ;GDT has 16 bytes
  mov   ax,    SEG GDT                             ;Get segment address of GDT
  movzx eax,   ax                                  ;Extend to 32 bits
  shl   eax,   4                                   ;*16
  mov   bx,    OFFSET GDT                          ;Get offset of GDT
  movzx ebx,   bx                                  ;Extend to 32 bits
  add   eax,   ebx                                 ;Linear address
  mov   dword ptr LoadGDT[2],   eax                ;Write base address of table into
                                                   ;memory
  lgdt  pword ptr LoadGDT                          ;Load the table
  mov   bx,    08h                                 ;BX points to entry 1
  cli                                              ;No interrupt allowed
  push  ds
  mov   eax,   cr0
  or    eax,   1                                   ;Enable protected mode
  mov   cr0,   eax
  jmp   ClearPipe1                                 ;Clear execution Pipe
ClearPipe1:
  mov   fs,    bx
  mov   gs,    bx
  mov   es,    bx
  and   al,    0FEh                                ;Now switch back to real mode
  mov   cr0,   eax
  jmp   ClearPipe2                                 ;Clear Execution pipe again
ClearPipe2:
  pop   ds
  sti                                              ;Enable interrupts again
  ret
@EnableFlatReal$qv ENDP
;***************************************************************************
;************                                          ************
;************            TestMode                      ************
;************                                          ************
;***************************************************************************
PUBLIC @TestMode$qv
@TestMode$qv PROC FAR
  push  fs
  push  si
  push  di
  xor   ax,    ax
  mov   fs,    ax
```

```
  mov  edi, [_BufferLinAddr]              ;Start of memory allocated
  mov  eax, 40000h                        ;Do for 31Meg div 4 time
                                          ;7C0000h vir 31 meg
StoreLoop: ;40000h vir 1 meg
  mov  fs:[edi],  eax                     ;Write double word in memory
  add  edi,  4                            ;Advance to next location
  dec  eax                                ;Repeat for whole memory
  jnz  StoreLoop
  mov  esi, [_BufferLinAddr]              ;Start of memory allocated
  mov  eax, 40000h                        ;Do for 31Meg div 4 time
                                          ;7C0000h vir 31 meg
ReadLoop:                                 ;40000h vir 1 meg
  cmp  fs:[esi],  eax                     ;Same as was written?
  jne  ReadError
  add  esi,  4                            ;Advance to next location
  dec  eax                                ;Repeat for whole memory
  jnz  ReadLoop
  mov  ax,  1
  pop  di
  pop  si
  pop  fs
  ret
ReadError:
  xor  ax,   ax
  pop  di
  pop  si
  pop  fs
  ret
@TestMode$qv ENDP
;*****************************************************************************
;************                                                    ************
;************             MemToHD                                ************
;************                                                    ************
;*****************************************************************************
PUBLIC @MemToHD$qnuc
@MemToHD$qnuc PROC FAR
  push  bp
  push  si
  push  di
  mov  bp,  sp
  mov  di,  [bp+10]
  mov  ax,  [bp+12]
  mov  es,  ax
  xor  ax,  ax
  mov  fs,  ax
  mov  esi,  [_BufferLinAddr]
  add  esi,  _DataCounter
```

```
  mov  [readcount],  4096
  cld                                          ;Clear direction flag
loop1:
  mov  eax,  fs:[esi]
  stosd                                        ;EAX -> ES:[(E)DI]
  add  esi,  4
  dec  [readcount]
  jz   klaar
  jmp  loop1
klaar:
  mov  eax,  16384
  add  _DataCounter,  eax
  pop  di
  pop  si
  pop  bp
  ret  0
@MemToHD$qnuc ENDP

END
```

B3 CAPTURE.ASM

```
.MODEL  LARGE
.386p

EXTRN _BufferLinAddr : DWORD;

.DATA

BusNum          db          0
DevFun          db          0
BAR0            dd          0
AMBEF           dd          0
AOMB            dd          0
OMB             dd          0
FIFO            dd          0
MCSR            dd          0
Temp            dd          0
count           dd          0
readcount       dd          0
test            DD          0

.CODE

public @Capture$qv
```

```
Public  @SwitchOn$qv
Public  @SwitchOff$qv
;*****************************************************************************
@Capture$qv PROC FAR
        push  bp
        push  fs
        push  di
        push  es
        push  si
        mov   bp,sp
;*****************************************************************************
;       PCI Bios installation check
;       AX = B101H
;       Ret: AH = 0 + CF clear => installed
;*****************************************************************************
  stc                                       ;Sets CF
  mov       ax,    0B101H
  int   1Ah
  jb        Error_1                         ;Jumps if CF not clear
  cmp       ah,    0                        ;Checks if AH = 0
  jne  Error_2
;*****************************************************************************
;       Find PCI device
;       AMCC card:
;       VID:  1234
;       DID:  5678
;       AX =  B102H
;       CX =  DID
;       DX =  VID
;       SI =  Device index (0) - The only one device with 1234 & 5678
;*****************************************************************************
  stc                                       ;Sets the CF
  mov       ax,    0B102H
  mov       cx,    5678H
  mov       dx,    1234H
  mov       si,    0
  int   1AH
  jb        Error_3                         ;Jumps if CF not clear
  cmp       ah,    0                        ;Checks if AH = 0
  je        Good_1
  cmp       ah,    81H                           ;Unsupported function
  je   Error_4
  cmp       ah,    83H                           ;Bad vendor ID
  je   Error_5
  cmp       ah,    86H                           ;Device not found
  je   Error_6
  cmp       ah,    87H                           ;Bad PCI register number
```

```
  je   Error_7
  jmp  Error                                    ;If something went wrong!
Good_1:
  mov       [BusNum],    bh                     ;Stores the Bus Number
  mov       [DevFun],    bl                     ;Stores the Dev Func
;******************************************************************************
;       Read configuration dword
;       AX =  B10AH
;       BH =  bus number
;       BL =  devfunc
;       DI =  register number - 10H BAR0
;******************************************************************************
  stc                                           ;Sets the CF
  mov       ax,    0B10AH
  mov       bh,    [BusNum]
  mov       bl,    [DevFun]
  mov       di,    10H
  int  1AH
  jb        Error_8                             ;Jumps if CF not clear
  cmp       ah,          0                      ;Checks if AH = 0
  je        Good_2
  cmp       ah,    81H                          ;Unsupported function
  je   Error_9
  cmp       ah,    83H                          ;Bad vendor ID
  je   Error_10
  cmp       ah,    86H                          ;Device not found
  je   Error_11
  cmp       ah,    87H                          ;Bad PCI register number
  je   Error_12
  jmp  Error                                    ;If something went wrong!
Good_2:
  mov       [BAR0],      ecx                    ;Stores the BAR0
;******************************************************************************
;       Calculation of the adresses for the
;       AMBEF and AOMB
;******************************************************************************
  mov       eax,   [BAR0]                       ;BAR0 & 0xFFFFFFFE
  and eax,  0FFFFFFFEH
  or        eax,   10H                          ;eax | 0x10
  mov       [AOMB],      eax                    ;Stores the AOMB
  mov       eax,   [BAR0]                       ;BAR0 & 0xFFFFFFFE
  and eax,  0FFFFFFFEH
  or        eax,   34H                          ;eax | 0x34
  mov       [AMBEF],     eax                    ;Stores the AMBEF
  mov       eax,   [BAR0]                       ;BAR0 & 0xFFFFFFF
  and eax,  0FFFFFFFEH
  or        eax,   00H                          ;eax | 0x00
```

```
  mov      [OMB],        eax                       ;Stores the OMB
  mov      eax,   [BAR0]                           ;BAR0 & 0xFFFFFFF
  and eax, 0FFFFFFFEH
  or       eax,   20H                   ;eax | 0x20
  mov      [FIFO],       eax                       ;Stores the FIFO
  mov      eax,   [BAR0]                           ;BAR0 & 0xFFFFFFF
  and eax, 0FFFFFFFEH
  or       eax,   3CH                   ;eax | 0x3C
  mov      [MCSR],       eax                       ;Stores the MCSR
  jmp DataCap
;******************************************************************************
Error:                                  ;Unknown error, something went wrong
       mov   ax,    0FFFFH
       jmp   Fin
;******************************************************************************
Error_1:                                ;Installation check CF <> 0
       mov   ax,  1
       jmp   Fin
;******************************************************************************
Error_2:                                ;Installation check AH <> 0
       mov   ax,  2
       jmp   Fin
;******************************************************************************
Error_3:                                ;Find device CF <> 0
       mov   ax,    3
       jmp   Fin
;******************************************************************************
Error_4:                                ;Find device: Unsupported function
       mov   ax,    4                   ;81H
       jmp   Fin
;******************************************************************************
Error_5:                                ;Find device: Bad vendor ID
       mov   ax,    5                   ;83H
       jmp   Fin
;******************************************************************************
Error_6:                                ;Find device: Device not found
       mov   ax,    6                   ;86H
       jmp   Fin
;******************************************************************************
Error_7:                                ;Find device: Device not found
       mov   ax,    7                   ;87H
       jmp   Fin
;******************************************************************************
Error_8:                                ;Config read CF <> 0
       mov   ax,    8
       jmp   Fin
;******************************************************************************
```

```
Error_9:                                 ;Config read: Unsupported function
        mov   ax,    9                   ;81H
        jmp   Fin
;***************************************************************************
Error_10:                                ;Config read: Bad vendor ID
        mov   ax,    10                  ;83H
        jmp   Fin
;***************************************************************************
Error_11:                                ;Config read: Device not found
        mov   ax,    11                  ;86H
        jmp   Fin
;***************************************************************************
Error_12:                                ;Config read: Device not found
        mov   ax,    12                  ;87H
        jmp   Fin
;***************************************************************************
;                       Fin
;***************************************************************************
Fin:
  pop si
  pop es
  pop di
  pop fs
  pop bp
  ret
;***************************************************************************
;                       Data capture part
;***************************************************************************
DataCap:
  xor  ax,    ax
  mov  fs,    ax
  mov  edi, [_BufferLinAddr]                ;Start of memory allocated
  cld                                       ;Clear direction flag
  cli                                       ;Disables int
  mov         edx,   [OMB]                  ;Make sure that board is switched
                                            off
  mov         eax,   0FFFFFFFFH
  out  dx,    eax
regread:
  mov         edx,   [MCSR]                        ;Reads the status register
  in          eax,   dx
  and  eax,  020H                           ;Mask bit 6 to determine if FIFO is
  cmp         eax,   020H                          ;empty
  jz          loop1
  mov         edx,   [FIFO]
  in          eax,   dx
  jmp  regread
```

```
loop1:
  mov       edx,   [OMB]
  mov       eax,   0FFFFFFF5H
  out  dx,  eax
 mov        [count],      10001h            ;Counter to get (7C0000h * 32)/4 bits
                                    ;1F0001h vir 31 meg
loop2:                              ;10001h vir 1 meg
  mov       dx,    word ptr [AMBEF]
  in        eax,   dx
  and  eax, 0F0000H
  cmp       eax,   0F0000H
  jne  loop2
  dec  [count]
  jz        klaar
  mov  dx,  word ptr [AOMB]
  in        eax,   dx
  mov       [readcount], 4
  mov       dx,    word ptr [FIFO]
loop3:
  in        eax,   dx
  mov       fs:[edi],     eax
  add  edi, 4
;  stosd                            ;EAX -> ES:[(E)DI]
  dec  [readcount]
  jz        loop2
  jmp  loop3
klaar:
  mov       edx,   [OMB]
  mov       eax,   0FFFFFFFFH
  out  dx,  eax
  sti                                           ;Enables int
  mov       ax,    0
  jmp  Fin
@Capture$qv ENDP
;****************************************************************************
@SwitchOn$qv PROC
                   push   bp
                   mov    bp,sp
;****************************************************************************
;       PCI Bios installation check
;       AX = B101H
;       Ret: AH = 0 + CF clear => installed
;****************************************************************************
  stc                                   ;Sets CF
  mov       ax,    0B101H
  int   1Ah
  jb        OnError_1                   ;Jumps if CF not clear
```

```
  cmp         ah,     0                          ;Checks if AH = 0
  jne  OnError_2
;****************************************************************************
;      Find PCI device
;      AMCC card:
;      VID:   1234
;      DID:   5678
;      AX =   B102H
;      CX =   DID
;      DX =   VID
;      SI =   Device index (0) - The only one device with 1234 & 5678
;****************************************************************************
  stc                                            ;Sets the CF
  mov         ax,                 0B102H
  mov         cx,                 5678H
  mov         dx,                 1234H
  mov         si,                 0
  int   1AH
  jb          OnError_3                          ;Jumps if CF not clear
  cmp         ah,                 0              ;Checks if AH = 0
  je          OnGood_1
  cmp         ah,                 81H                   ;Unsupported function
  je   OnError_4
  cmp         ah,                 83H                   ;Bad vendor ID
  je   OnError_5
  cmp         ah,                 86H                   ;Device not found
  je   OnError_6
  cmp         ah,                 87H                   ;Bad PCI register number
  je   OnError_7
  jmp  OnError                                   ;If something went wrong!
OnGood_1:
  mov         [BusNum],    bh                           ;Stores the Bus Number
  mov         [DevFun],    bl                           ;Stores the Dev Func
;****************************************************************************
;      Read configuration dword
;      AX =   B10AH
;      BH =   bus number
;      BL =   devfunc
;      DI =   register number - 10H BAR0
;****************************************************************************
  stc                                            ;Sets the CF
  mov         ax,          0B10AH
  mov         bh,          [BusNum]
  mov         bl,          [DevFun]
  mov         di,          10H
  int   1AH
  jb          OnError_8                          ;Jumps if CF not clear
```

```
  cmp     ah,       0                ;Checks if AH = 0
  je      OnGood_2
  cmp     ah,       81H                     ;Unsupported function
  je  OnError_9
  cmp     ah,       83H                     ;Bad vendor ID
  je  OnError_10
  cmp     ah,       86H                     ;Device not found
  je  OnError_11
  cmp     ah,       87H                     ;Bad PCI register number
  je  OnError_12
  jmp OnError                        ;If something went wrong!
OnGood_2:
  mov     [BAR0],   ecx                     ;Stores the BAR0
;***************************************************************************
;       Calculation of the adresses for the
;       AMBEF and AOMB
;***************************************************************************
  mov     eax,      [BAR0]                  ;BAR0 & 0xFFFFFFFE
  and eax,     0FFFFFFFEH
  or      eax,      10H              ;eax | 0x10
  mov     [AOMB],   eax                     ;Stores tha AOMB
  mov     eax,      [BAR0]                  ;BAR0 & 0xFFFFFFFE
  and eax,     0FFFFFFFEH
  or      eax,      34H              ;eax | 0x34
  mov     [AMBEF],  eax                     ;Stores tha AMBEF
  mov     eax,      [BAR0]                  ;BAR0 & 0xFFFFFFF
  and eax,     0FFFFFFFEH
  or      eax,      00H              ;eax | 0x00
  mov     [OMB],    eax                     ;Stores the OMB
  jmp OnDataCap
;***************************************************************************
OnError:                        ;Unknown error, something went wrong
  mov     [test], 0FFFFFFFFH
  jmp OnFin
;***************************************************************************
OnError_1:                           ;Installation check CF <> 0
  mov     [test], 1
  jmp OnFin
;***************************************************************************
OnError_2:                           ;Installation check AH <> 0
  mov     eax,  2
  mov     [test],       eax
  jmp OnFin
;***************************************************************************
OnError_3:                           ;Find device CF <> 0
  mov     eax,      3
  mov     [test], eax
```

```
 jmp OnFin
;******************************************************************************
OnError_4:                              ;Find device: Unsupported function
 mov      eax,      4   ; 81H
 mov      [test],   eax
 jmp OnFin
;******************************************************************************
OnError_5:                                  ;Find device: Bad vendor ID
 mov      eax,      5                           ;83H
 mov      [test], eax
 jmp OnFin
;******************************************************************************
OnError_6:                                  ;Find device: Device not found
 mov      eax,      6                           ;86H
 mov      [test],   eax
 jmp OnFin
;******************************************************************************
OnError_7:                                  ;Find device: Device not found
 mov      eax,      7                           ;87H
 mov      [test],   eax
 jmp OnFin
;******************************************************************************
OnError_8:                                  ;Config read CF <> 0
 mov      eax,      8
 mov      [test], eax
 jmp OnFin
;******************************************************************************
OnError_9:                              ;Config read: Unsupported function
 mov      eax,      9                       ;81H
 mov      [test],   eax
 jmp OnFin
;******************************************************************************
OnError_10:                             ;Config read: Bad vendor ID
 mov      eax,      10                      ;83H
 mov      [test], eax
 jmp OnFin
;******************************************************************************
OnError_11:                             ;Config read: Device not found
 mov      eax,      11                      ;86H
 mov      [test],   eax
 jmp OnFin
;******************************************************************************
OnError_12:                             ;Config read: Device not found
 mov      eax,      12                      ;87H
 mov      [test],   eax
 jmp OnFin
;******************************************************************************
```

```
OnFin:
  mov       bx,         [bp+6]
  mov  ax,        [bp+8]
  mov  es,        ax
  mov       eax,        [test]
  mov  es:[bx], eax
  pop  bp
  ret
;******************************************************************************
; Data capture part
;******************************************************************************
OnDataCap:
  mov       edx,        [OMB]
  mov       eax,        0FFFFFFF5H
  out  dx,        eax
  mov       [test], 0
  jmp  OnFin
@SwitchOn$qv ENDP
;******************************************************************************
@SwitchOff$qv PROC
  push  bp
  mov   bp,sp
;******************************************************************************
;       PCI Bios installation check
;       AX = B101H
;       Ret: AH = 0 + CF clear => installed
;******************************************************************************
  stc                                         ;Sets CF
  mov       ax,  0B101H
  int  1Ah
  jb        OffError_1                        ;Jumps if CF not clear
  cmp       ah,  0  ;    Checks if AH = 0
  jne  OffError_2
;******************************************************************************
;       Find PCI device
;       AMCC card:
;       VID:  1234
;       DID:  5678
;       AX =  B102H
;       CX =  DID
;       DX =  VID
;       SI =  Device index (0) - The only one device with 1234 & 5678
;******************************************************************************
  stc                                         ;Sets the CF
  mov       ax,  0B102H
  mov       cx,   5678H
  mov       dx,  1234H
```

```
  mov       si,  0
  int   1AH
  jb        OffError_3                          ;Jumps if CF not clear
  cmp       ah,  0                              ;Checks if AH = 0
  je        OffGood_1
  cmp       ah,  81H                                    ;Unsupported function
  je  OffError_4
  cmp       ah,  83H                                    ;Bad vendor ID
  je  OffError_5
  cmp       ah,  86H                                    ;Device not found
  je  OffError_6
  cmp       ah,  87H                                    ;Bad PCI register number
  je  OffError_7
  jmp OffError                                  ;If something went wrong!
OffGood_1:
  mov       [BusNum],   bh                              ;Stores the Bus Number
  mov       [DevFun],   bl                              ;Stores the Dev Func
;*****************************************************************************
;      Read configuration dword
;      AX =  B10AH
;      BH =  bus number
;      BL =  devfunc
;      DI =  register number - 10H BAR0
;*****************************************************************************
  stc                                           ;Sets the CF
  mov       ax,         0B10AH
  mov       bh,         [BusNum]
  mov       bl,         [DevFun]
  mov       di,         10H
  int   1AH
  jb        OffError_8                          ;Jumps if CF not clear
  cmp       ah,         0                       ;Checks if AH = 0
  je        OffGood_2
  cmp       ah,         81H                             ;Unsupported function
  je  OffError_9
  cmp       ah,         83H                             ;Bad vendor ID
  je  OffError_10
  cmp       ah,         86H                             ;Device not found
  je  OffError_11
  cmp       ah,         87H                             ;Bad PCI register number
  je  OffError_12
  jmp OffError                                  ;If something went wrong!
OffGood_2:
  mov       [BAR0],     ecx                             ;Stores the BAR0
;*****************************************************************************
;      Calculation of the adresses for the
;      AMBEF and AOMB
```

```
;*****************************************************************************
  mov        eax,          [BAR0]                    ;BAR0 & 0xFFFFFFFE
  and eax,       0FFFFFFFEH
  or         eax,          10H               ;eax | 0x10
  mov        [AOMB],       eax                       ;Stores tha AOMB
  mov        eax,          [BAR0]                    ;BAR0 & 0xFFFFFFFE
  and eax,       0FFFFFFFEH
  or         eax,          34H               ;eax | 0x34
  mov        [AMBEF],      eax                       ;Stores tha AMBEF
  mov        eax,          [BAR0]                    ;BAR0 & 0xFFFFFFF
  and eax,       0FFFFFFFEH
  or         eax,          00H               ;eax | 0x00
  mov        [OMB],        eax                       ;Stores the OMB
  jmp OffDataCap
;*****************************************************************************
OffError:                                  ;Unknown error, something went wrong
  mov        [test], 0FFFFFFFFH
  jmp OffFin
;*****************************************************************************
OffError_1:                                  ;Installation check CF <> 0
  mov        [test], 1
  jmp OffFin
;*****************************************************************************
OffError_2:                                  ;Installation check AH <> 0
  mov        eax,   2
  mov        [test],       eax
  jmp OffFin
;*****************************************************************************
OffError_3:                                  ;Find device CF <> 0
  mov        eax,          3
  mov        [test], eax
  jmp OffFin
;*****************************************************************************
OffError_4:                                ;Find device: Unsupported function
  mov        eax,          4                 ;81H
  mov        [test],       eax
  jmp OffFin
;*****************************************************************************
OffError_5:                                  ;Find device: Bad vendor ID
  mov        eax,          5                         ;83H
  mov        [test], eax
  jmp OffFin
;*****************************************************************************
OffError_6:                                  ;Find device: Device not found
  mov        eax,          6                         ;86H
  mov        [test],       eax
  jmp OffFin
```

```
;***************************************************************************
OffError_7:                                     ;Find device: Device not found
  mov       eax,        7                             ;87H
  mov       [test],     eax
  jmp  OffFin
;***************************************************************************
OffError_8:                                     ;Config read CF <> 0
  mov       eax,        8
  mov       [test], eax
  jmp  OffFin
;***************************************************************************
OffError_9:                                 ;Config read: Unsupported function
  mov       eax,        9                       ;81H
  mov       [test],     eax
  jmp  OffFin
;***************************************************************************
OffError_10:                                    ;Config read: Bad vendor ID
  mov       eax,        10                            ;83H
  mov       [test], eax
  jmp  OffFin
;***************************************************************************
OffError_11:                                    ;Config read: Device not found
  mov       eax,        11                            ;86H
  mov       [test],     eax
  jmp  OffFin
;***************************************************************************
OffError_12:                                    ;Config read: Device not found
  mov       eax,        12                            ;87H
  mov       [test],     eax
  jmp  OffFin
;***************************************************************************
OffFin:
  mov       bx,         [bp+6]
  mov  ax,        [bp+8]
  mov  es,        ax
  mov       eax,        [test]
  mov  es:[bx], eax
  pop  bp
  ret
;***************************************************************************
;      Data capture part
;***************************************************************************
OffDataCap:
  mov       edx,        [OMB]
  mov       eax,        0FFFFFFFFH
  out  dx,        eax
  mov       [test], 0
```

```
  jmp  OffFin
@SwitchOff$qv ENDP
  END
```

All the files are provided on the attached CD, under *\SOFTWARE*.

Appendix D

Serial presence detect (SPD)

Table D.1 SPD EEPROM data format

Byte Number	Function	Implementation
0	Defines number of bytes written into serial memory at module manufacturer	Required
1	Total number of bytes of SPD memory device	Required
2	Fundamental memory type (FPM, EDO, SDRAM..)	Required
3	Number of row addresses on this assembly (includes Mixed-size Row addresses)	Required
4	Number Column Addresses on this assembly (includes Mixed-size Col addresses)	Required
5	Module Rows on this assembly	Required
6	Data Width of this assembly	Required
7	Module Data Width continuation	Required
8	Voltage interface standard of this assembly	Required
9	SDRAM Cycle time, CL=X (highest CAS latency)	Required
10	SDRAM Access from Clock (highest CAS latency)	Required
11	DIMM Configuration type (non-parity, ECC)	Required
12	Refresh Rate/Type	Required
13	Primary SDRAM Width	Required
14	Error Checking SDRAM width	Required
15	Minimum Clock Delay Back to Back Random Column Address	Required
16	Burst Lengths Supported	Required

17	Number of Banks on Each SDRAM Device	Required
18	CAS# Latencies Supported	Required
19	CS# Latency	Required
20	Write Latency	Required
21	SDRAM Module Attributes	Required
22	SDRAM Device Attributes: General	Required
23	Min SDRAM Cycle time at CL X-1 (2nd highest CAS latency)	Required
24	SDRAM Access from Clock at CL X-1 (2nd highest CAS latency)	Required
25	Min SDRAM Cycle time at CL X-2 (3rd highest CAS latency)	Optional
26	Max SDRAM Access from Clock at CL X-2 (3nd highest CAS latency)	Optional
27	Min Row Precharge Time (Trp)	Required
28	Min Row Active to Row Active (Trrd)	Required
29	Min RAS to CAS Delay (Trcd)	Required
30	Minimum RAS Pulse Width (Tras)	Required
31	Density of each row on module (mixed, non-mixed sizes)	Required
32	Command and Address signal input setup time	Required
33	Command and Address signal input hold time	Required
34	Data signal input setup time	Required
35	Data signal input hold time	Required
36-61	Superset Information (may be used in future)	
62	SPD Data Revision Code	Required
63	Checksum for bytes 0-62	Required
64-71	Manufacturer's JEDEC ID code per JEP-108E	Optional
72	Manufacturing Location	Optional
73-90	Manufacturer's Part Number	Optional

91-92	Revision Code	Optional
93-94	Manufacturing Date	Optional
95-98	Assembly Serial Number	Optional
99-125	Manufacturer Specific Data	Optional
126	Intel specification frequency	Required
127	Intel Specification CAS# Latency support	Required
128+	Unused storage locations	

Appendix E

DIMM

E1 Pinout

Table E.1 DIMM pinout

PIN	SYMBOL	PIN	SYMBOL	PIN	SYMBOL	PIN	SYMBOL
1	VSS	43	VSS	85	VSS	127	VSS
2	DQ0	44	DNU	86	DQ32	128	CKE0
3	DQ1	45	S2#	87	DQ33	129	S3#
4	DQ2	46	DQMB2	88	DQ34	130	DQMB6
5	DQ3	47	DQMB3	89	DQ35	131	DQMB7
6	VDD	48	DNU	90	VDD	132	NC (A13)
7	DQ4	49	VDD	91	DQ36	133	VDD
8	DQ5	50	NC	92	DQ37	134	NC
9	DQ6	51	NC	93	DQ38	135	NC
10	DQ7	52	CB2	94	DQ39	136	CB6
11	DQ8	53	CB3	95	DQ40	137	CB7
12	VSS	54	VSS	96	VSS	138	VSS
13	DQ9	55	DQ16	97	DQ41	139	DQ48
14	DQ10	56	DQ17	98	DQ42	140	DQ49
15	DQ11	57	DQ18	99	DQ43	141	DQ50
16	DQ12	58	DQ19	100	DQ44	142	DQ51
17	DQ13	59	VDD	101	DQ45	143	VDD
18	VDD	60	DQ20	102	VDD	144	DQ52
19	DQ14	61	NC	103	DQ46	145	NC
20	DQ15	62	NC	104	DQ47	146	NC
21	CB0	63	NC (CKE1)	105	CB4	147	REGE
22	CB1	64	VSS	106	CB5	148	VSS
23	VSS	65	DQ21	107	VSS	149	DQ53
24	NC	66	DQ22	108	NC	150	DQ54
25	NC	67	DQ23	109	NC	151	DQ55
26	VDD	68	VSS	110	VDD	152	VSS
27	WE#	69	DQ24	111	CAS#	153	DQ56
28	DQMB0	70	DQ25	112	DQMB4	154	DQ57
29	DQMB1	71	DQ26	113	DQMB5	155	DQ58
30	S0#	72	DQ27	114	S1#	156	DQ59
31	DNU	73	VDD	115	RAS#	157	VDD
32	VSS	74	DQ28	116	VSS	158	DQ60
33	A0	75	DQ29	117	A1	159	DQ61

34	A2	76	DQ30	118	A3	160	DQ62
35	A4	77	DQ31	119	A5	161	DQ63
36	A6	78	VSS	120	A7	162	VSS
37	A8	79	CK2	121	A9	163	CK3
38	A10	80	NC	122	BA0	164	NC
39	BA1	81	WP	123	A11	165	SA0
40	VDD	82	SDA	124	VDD	166	SA1
41	VDD	83	SCL	125	CK1	167	SA2
42	CK0	84	VDD	126	NC (A12)	168	VDD

E2 Functional description

Table E.2 Functional description

Symbol	Type	Description
RAS#, CAS#, WE#	Input	Command Inputs: RAS#, CAS# and WE# (along with S0#-S3#) define the command being entered.
CK0-CK3	Input	Clock: CK0-CK3 are driven by the system clock. All SDRAM input signals are sampled on the positive edge of CK. CK also increments the internal burst counter and controls the output registers.
CKE0	Input	Clock Enable: CKE0 activates (HIGH) and deactivates (LOW) the CK0-CK3 signals. Deactivating the clock provides POWER-DOWN and SELF REFRESH operation (all banks idle) or CLOCK SUSPEND operation (burst access in progress). CKE0 is synchronous except after the device enters power-down and self refresh modes, where CKE0 becomes asynchronous until after exiting the same mode. The input buffers, including CK0-CK3, are disabled during power-down and self refresh modes, providing low standby power.
S0#-S3#	Input	Chip Select: S0#-S3# enable (registered LOW) and disable (registered HIGH) the command decoder. All commands are masked when S0#-S3# are registered HIGH. S0#-S3#are considered part of the command code.

DQMB0-DQMB7	Input	Input/Output Mask: DQMB is an input mask signal for write accesses and an output enable signal for read accesses. Input data is masked when DQMB is sampled HIGH during a WRITE cycle. The output buffers are placed in a High-Z state (two-clock latency) when DQMB is sampled HIGH during a READ cycle.
BA0, BA1	Input	Bank Address: BA0 and BA1 define to which bank the ACTIVE, READ, WRITE or PRECHARGE command is being applied.
A0-A11	Input	Address Inputs: A0-A11 are sampled during the ACTIVE command (row-address A0-A11) and READ/WRITE command (column-address A0-A9, with A10 defining AUTO PRECHARGE) to select one location out of the memory array in the respective bank. A10 is sampled during a PRECHARGE command to determine if both banks are to be precharged (A10 HIGH). The address inputs also provide the op-code during a LOAD MODE REGISTER command.
DQ0-DQ63	Input/ Output	Data I/O: Data bus.
CB0-CB7	Input/ Output	Check Bits.
VDD	Supply	Power Supply: +3.3V ±0.3V.
VSS	Supply	Ground.
WP	Input	Write Protect: Serial presence-detect hardware write protect.
SDA	Input/ Output	Serial Presence-Detect Data: SDA is a bidirectional pin used to transfer addresses and data into and data out of the presence-detect portion of the module.
SCL	Input	Serial Clock for Presence-Detect: SCL is used to synchronize the presence-detect data transfer to and from the module.
SA0-SA2	Input	Presence-Detect Address Inputs: These pins are used to configure the presence-detect device.

REGE	Input	Register Enable.
DNU	–	Do Not Use: These pins are not connected on this module but are assigned pins on the compatible DRAM version.

Bibliography

[CEP98] Altera. *Configuration EPROMs for FLEX Devices Ver. 9*, October 1998.

[10K98] Altera. *FLEX 10KE Embedded Programmable Logic Family Ver. 1*, August 1998.

[AMC96] AMCC. *S5933 PCI Controller Data Book*, Spring 1996.

[CYP95] Cypress. *ICD2053B Programmable Clock Generator*, October 1995.

[SOL96] Edward Solari, George Willse. *PCI Hardware and Software Architecture & Design 3rd Edition*. Annabooks, 1996.

[HIT98] Hitachi. *123MB Unbuffered SDRAM DIMM, 100MHz Memory Bus (HB52E168EN) 16-Mword x 64-bit, 2-Bank Module Rev. 0.1*, 1998.

[DEI94] HM Deitel, PJ Deitel. *C++ How to Program*. Prentice-Hall, Inc., 1994.

[IBM97] IBM. *168 Pin Unbuffered DRAM DIMM Product Overview Rev 1/97*, 1997.

[IBM96] IBM. *168 Pin Unbuffered SDRAM DIMM Characteristics Rev. 8/96*, 1996.

[IBM98] IBM. *168 Pin SDRAM Registered DIMM Functional Description & Timing Diagrams Rev. 1/98*, 1998.

[ISC98] Intel. *Intel 100MHz Pentium(tm) II processor/440BX AGPset Uniprocessor Customer Reference Schematics Rev. 1.0*, 1998.

[IRD98] Intel. *PC SDRAM Registered DIMM Specification Rev. 1.0*, 1998.

[IUD98] Intel. *PC SDRAM Unbuffered DIMM Specification Rev. 1.0*, 1998.

[ISP97] Intel. *PC SDRAM Serial Presence Detect (SPD) Specification Revision 1.2A*, December 1997.

[ISW99] Intel. *Architecture Software Developer's Manual Volumes 1 to 3*, 1999.

[UFF87] John Uffenbeck. *The 8086/8088 Family: Design, Programming, and Interfacing*. Prentice-Hall, Inc., 1987.

[SCA88] Leo J Scanlon. *8086/8088/80286 Assembly Language*. Brady, 1988.

[LEU96] Markus Leugner. *User Manual for the FPGA Tutor Boards*. University of Stellenbosch, 29 July 1996.

[MIC98] Micron. *MT18LSDT3272 Synchronous DRAM Module Rev. 8/98*, 1998.

[HOR94] Paul Horowitz, Winfield Hill. *The Art of Electronics Second Edition*. Cambridge University Press, 1994.

[PCI95] PCI Special Interest Group. *PCI Local Bus Specification Revision 2.1*, 1 June 1995.

[QAP98] Quality Semiconductor, Inc.. *Application Note AN-11A - Bus*

Switches Provide 5V and 3.3V Logic Conversion with Zero Delay, 3 August 1998.

[QQS97] Quality Semiconductor, Inc.. *QuickSwitch Products High-Speed 10-Bit Bus Switch With Flow-Thru Pinout*, 31 July 1997.

[BRW97] Ralf Brown. *Interrupt List Release 53*, 12 January 1997.

[LIP91] Stanley B Lippman. *C++ Primer 2nd Edition*. Addison-Wesley Publishing Company, 1991.

[TDL98] Texas Instruments. *SN65LVDS32, SN65LVDS3486, SN65LVDS9637 High-Speed Differential Line Receivers*, November 1998.

[TCD96] Texas Instruments. *CDC516 3.3V Phase-Lock Loop Clock Driver with 3-State Outputs*, July 1996.

[TCT98] Texas Instruments. *High Speed Clock Distibution Design Techniques for CDC509/516/2509/2510/2516 Application Report: SLMA003*, March 1998.

[TBD97] Texas Instruments. *SN74ALB16244 16-Bit Buffer/Driver with 3-state Outputs*, July 1997.