The Evaluation of a SoC Processor as an On-Board Computer for a Low Earth Orbit Satellite

by

Jacques Jordaan

Thesis presented at the University of Stellenbosch in partial fulfilment of the requirements for the degree of

Master of Science in Electrical & Electronic Engineering

Department of Electrical & Electronic Engineering
University of Stellenbosch
Private Bag X1, 7602 Matieland, South Africa

Study leader: Mnr. H. Berner

April 2005

Declaration

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

J. Jordaan

Abstract

The use of commercial-off-the-shelf components in low earth orbit (LEO) satellite systems has become a very popular design trend. Not only are many of these components sufficiently radiation tolerant, but are also less expensive than their space qualified counterparts. Commercial processors are already used in the on-board computer (OBC) of satellites like SUNSAT 2004, CanX and SNAP-1. With the increasing demand for smaller, less expensive satellites and a shorter time-to-market window, the feasibility of implementing a satellite's OBC as a system-on-a-chip (SoC) is now considered.

This thesis describes a single-chip implementation, excluding memory, of a LEO microsatellite's OBC on the commercial grade Altera Excalibur embedded processor. A typical satellite OBC system was developed on the Excalibur device to test the functionality and performance of the device as a single-chip OBC.

Opsomming

Die gebruik van kommersiële komponente in lae aardwentelbaan satelliet stelsels het 'n baie gewilde ontwerps neiging geword. Meeste van hierdie komersiële komponente is nie net voldoende bestand teen die radiasie in die ruimte nie, maar boonop goedkoper as soortgelyke komponente wat spesifiek vir de ruimte vervaardig is. Kommersiële verwerkers word alreeds gebruik in die aanboord rekenaar (AR) van satelliete soos SUNSAT 2004, CanX and SNAP-1. Met die aahoudende aanvraag vir kleiner, goedkoper en 'n korter ontwikkelings tydperk, word die implementering van 'n satelliet se AR as 'n stelsel-op-'n-skyfie nou oorweeg.

Hierdie tesis beskryf 'n enkel-skyfie implementasie, geheue uitgesluit, van 'n lae aardwentelbaan mikrosatelliet se AR op die kommersiële Altera Excalibur geïntregreerde verwerker. 'n Tipiese AR stelsel was ontwikkel op die Excalibur verwerker om die funksionaliteit en werkverrigting van die toestel as 'n enkel-skyfie AR te toets.

Acknowledgements

I would like to thank the following people for their contributions:

- Heiko Berner, my supervisor, for his advice
- Francois Retief for all his advice expecially with eCos and the GNU C-Tools
- My family for their support and motivation

Contents

Declaration ii

Abstract iii

Opsomming iv

Acknowledgements v

Contents vi

List of Figures xii

List of Tables xiv

List of Abbreviations and Acronyms xv

1 Introduction 1

1.1 Satellite On-Board Computer Background 1

1.2 Modern Small Satellite Design 2

1.3 An OBC as a SoC 3

1.4 Document Outline 4

2 Suitability of the Excalibur Processor 5

2.1 Processor Requirements 5

2.2 Excalibur Models 6

2.3 Feasibility of the EPXA1 Device as OBC 7

2.4 Conclusion 8

3 The EPXA1 Embedded Processor PLD **9**

3.1 The EPXA1 Device 9

3.1.1 The ARM922T Processor 9

3.1.2 Bus Architecture 10

3.1.3 On-Chip Memories 11

3.1.4 External Memory Interfaces 11

3.1.5 Embedded Peripherals 12

3.1.6 Programmable Logic Architecture 13

3.1.7 Embedded Peripherals Memory Map 14

3.2 The Excalibur EPXA1 Development Kit 14

3.2.1 The Development Board 14

3.2.2 The Development Software 16

4 Software Environment **17**

4.1 The ARM Architecture 17

4.1.1 The ARM Instruction Set 18

4.2 Programming Languages 21

4.2.1 The C Programming Language 21

4.2.2 Compiler Tools 22

4.2.3 VHDL Language 22

4.2.4 Altera's Quartus II 23

4.3 Operating System Selection 23

4.3.1 Nucleus RTOS 23

4.3.2 RTEMS 23

4.3.3 Linux 24

4.3.4 eCos 24

4.4 Conclusion 25

5 Development Environment **26**

5.1 Introduction to eCos 26

5.1.1 Hardware Abstraction Layer 27

5.1.2 eCos Kernel 28

5.1.3 Device Drivers 30

5.2 Cygwin Tools 30

5.3 eCos Configuration Tool 30

5.4 Booting the EPXA1 Device 31

5.4.1 Boot from Flash Mode 31

5.4.2 Boot Code 31

5.4.3 Linker Script 32

5.4.4 The RedBoot ROM Monitor 32

5.4.5 RedBoot Startup Procedure 32

5.5 Configuring the PLD 35

6 EPXA1 OBC Design **37**

6.1 EPXA1 OBC Design Overview 38

6.2 Processor Setup 39

6.3 Memory Interfaces 40

6.3.1 SDRAM Controller 40

6.3.2 Expansion Bus Interface (EBI) 40

6.4 AMBA AHB Bus System 42

6.4.1 Overview 42

6.4.2 AMBA AHB Operation 42

6.4.3 The AHB Bridges 44

6.4.4 AHB Bridge Operation 45

6.5 Interrupt Controller 46
6.5.1 Interrupt Controller Interface 47
6.5.2 Operating Modes 47
6.5.3 Interrupt Mode Implementation 48
6.6 Watchdog Timer 50
6.7 Stripe UART 50
6.8 Reset Module 51
6.9 General Purpose Timers 52
6.10 Ethernet Controller 53
6.11 AHB Bus Interface 54
6.12 EDAC Unit 55
6.12.1 Choosing an EDAC Data Coding Scheme 56
6.12.2 The Quasi-Cyclic (16,8) Code 56
6.12.3 Physical EDAC Implementation 57
6.13 PLD UART 58
6.14 I^{2}C Controller 59
6.14.1 I^{2}C Overview 59
6.14.2 I^{2}C Implementation 59

7 Tests and Measurements 60

7.1 Power Consumption 60
7.1.1 PLD Power Consumption 60
7.1.2 Embedded Stripe Power Consumption 60
7.1.3 Total Power Consumption 63
7.1.4 Low-Power Mode 64
7.2 Floating Point Test 64
7.2.1 Adjusting the PLL1 Output Clock 65

7.2.2 Controlling the Use of the Cache Memory 66

7.2.3 Floating Point Test Results 66

7.3 Interrupt Controller 67

7.4 Watchdog Timer 67

7.5 General Purpose Timers 67

7.6 Ethernet 68

7.7 PLD UART 68

7.8 EDAC Performance 68

7.8.1 EDAC Accuracy 69

7.8.2 EDAC Memory Cycle Times 70

7.9 The I^2C Bus 71

8 Conclusions and Recommendations **73**

8.1 Conclusions 73

8.1.1 Reliability 73

8.1.2 Power Consumption 73

8.1.3 Performance 74

8.1.4 Architecture 74

8.1.5 Development Tools 75

8.1.6 EPXA1 as Single-Chip OBC 75

8.2 Recommendations 75

8.2.1 Radiation Tests 75

8.2.2 Peripheral Interfaces 76

8.2.3 EDAC Unit 76

8.2.4 Development Board 76

A PLD Modules **77**

A.1 Block Diagram 78

A.2 PLD Master Interface 79

A.3 PLD Slave Interface 80

A.4 PLD UART 81

B PLD Interrupt Controller **82**

B.1 Block Diagram 83

B.2 Installed ISRs and DSRs 84

B.3 Installed VSR 87

C I^{2}C Measurements **89**

D CD-ROM Data **90**

Bibliography **91**

List of Figures

1.1 Satellite's Subsystems 1

1.2 The use of COTS technologies to optimize spacecraft mass/cost 3

2.1 The Excalibur Architecture [1] 7

3.1 The Excalibur Device System Architecture [1] 10

3.2 Functional Block Diagram of ARM922T [2] 11

3.3 The Altera Excalibur EPXA1 Development Board 15

4.1 ARM Register Organization 19

4.2 Program Status Register Format 20

5.1 eCos System Architecture 27

5.2 eCos Configuration Tool 31

5.3 EPXA1 Memory Map Layout 33

5.4 Flow Diagram for Configuring the FPGA Contents 36

6.1 General Satellite OBC Model 37

6.2 EPXA1 OBC Model 39

6.3 EPXA1 SDRAM Controller Connection to SDRAM 40

6.4 Expansion Bus Interface Block Diagram [1] 41

6.5 AHB transaction waveform [1] 43

6.6 Functional Block Diagram of AHB Bridge [1] 44

6.7 The EPXA1 Interrupt Controller in the Stripe [1] 46

6.8 EPXA1 eCos Interrupt Handling Flowchart 49

6.9 EPXA1 Reset Sequence 52

6.10 AHB Bridges 54

6.11 SEUs in the S80/T OBC program memory at 1330-km altitude [3] 55

6.12 Encoder Schematic 56

6.13 Decoder Schematic 57

7.1 Current Consumed by the Embedded Stripe versus AHB1 Clock Frequency [4] 61

7.2 Current Consumed Versus Clock Frequency Due to the Embedded Stripe Bridge Interface [4] 62

7.3 Current Consumed Versus Clock Frequency Due to the DPSRAM Interface [4] 62

7.4 Flowchart for Reprogramming PLL1 65

7.5 RedBoot with network package 68

7.6 Memory Write Cycle 69

7.7 I^{2}C Write Transaction in Standard Mode 71

7.8 I^{2}C Read Transaction in Standard Mode 72

A.1 OBC Modules Implemented in the PLD 78

A.2 ASM Chart of Master Interface 79

A.3 ASM Chart of Slave Interface 80

A.4 ASM Chart of PLD UART 81

B.1 6-Bit Priority Interrupt Mode Block Diagram 83

C.1 I^{2}C Write Transaction in Fast Mode 89

C.2 I^{2}C Read Transaction in Fast Mode 89

List of Tables

2.1 Excalibur Characteristics 8

3.1 Memory Map Peripherals and Elements 14

4.1 ARM v4 Processor Modes 18

4.2 Exception Processing Modes 21

5.1 SBI File Format 35

6.1 DPSRAM Interface Modes and Widths 59

7.1 IGRF Test Results (100 loops) 66

7.2 Memory Cycle Times 70

List of Abbreviations and Acronyms

ADCS	- Attitude Determination and Control System
ADS	- ARM Development Suite
AHB	- Advanced High-performance Bus
ALU	- Arithmetic and Logic Unit
AMBA	- Advanced Microcontroller Bus Architecture
APEX	- Advanced Programmable Embedded Matrix
ARM	- Advanced RISC Machines
BGA	- Ball-Grid Array
CAM	- Content Addressable Memory
CAN	- Controller Area Network
COTS	- Commercial-Off-The-Shelf
CPU	- Central Processing Unit
CD-ROM	- Compact Disc ROM
DAC	- Digital-to-Analog Converter
DDR	- Double Data Rate
DHCP	- Dynamic Host Configuration Protocol
DIP	- Dual In-line Package
DPSRAM	- Dual-Port SRAM
DSR	- Deferred Service Routine
EBI	- Expansion Bus Interface
eCos	- Embedded Configurable OS
EDAC	- Error Detection and Correction
ESA	- European Space Agency
ESB	- Embedded System Block
FIFO	- First-In-First-Out
FIQ	- Fast Interrupt Request
FPGA	- Field Programmable Gate Array
FPU	- Floating Point Unit
GCC	- GNU Compiler
GDB	- GNU Debugger

HAL	- Hardware Abstraction Layer
HDL	- Hardware Description Language
IC	- Integrated Circuit
IEEE	- Institute of Electrical and Electronics Engineers
IGRF	- International Geomagnetic Reference Field
IP	- Intellectual Property
IRQ	- Interrupt Request
ISR	- Interrupt Service Routine
I^{2}C	- Interconnected Integrated Circuit
I/O	- Input/Output
JTAG	- Joint Test Action Group
Kbyte	- Kilobytes (1024 bytes)
LED	- Light Emitting Diode
LEO	- Low Earth Orbit
LUT	- Look-Up Table
LVDS	- Low Voltage Differential Signalling
MEMS	- Micro-Electro-Mechanical Systems
MIPS	- Million Instructions Per Second
MMU	- Memory Management Unit
MHz	- Megahertz
Mbps	- Megabits per second
MLQ	- Multi-Level Queue
OBC	- On-Board Computer
OBDH	- On-Board Data Handling
OS	- Operating System
PLD	- Programmable Logic Device
PLL	- Phase Lock Loop
RAM	- Random Access Memory
RedBoot	- Red Hat Embedded Debug and Bootstrap
RISC	- Reduced Instruction Set Computer
ROM	- Read Only Memory
RTC	- Real-Time Clock
RTEMS	- Real-Time Executive for Multiprocessor Systems
RTOS	- Real-Time Operating System
SAA	- South Atlantic Anomaly
SDR	- Single Data Rate
SDRAM	- Synchronous Dynamic Random Access Memory
SEU	- Single Event Upset
SoC	- System-on-Chip
SOPC	- System on a Programmable Chip

SRAM	- Static Random Access Memory
SSTL	- Surrey Satellite Technology Limited
TCP/IP	- Transmission Control Protocol/Internet Protocol
TFTP	- Trivial File Transfer Protocol
UART	- Universal Asynchronous Receiver Transmitter
VHDL	- Very high-speed integrated circuit HDL
VSR	- Vector Service Routine

Chapter 1

Introduction

1.1 Satellite On-Board Computer Background

A satellite consists of various subsystems. Each of these subsystems performs different tasks on-board the satellite. Figure 1.1 illustrates a typical satellite system with some of its subsystems.

One of these subsystems is the on-board data handling (OBDH) system. The OBDH is the key to the satellite's sophisticated capabilities. At the heart of the OBDH system

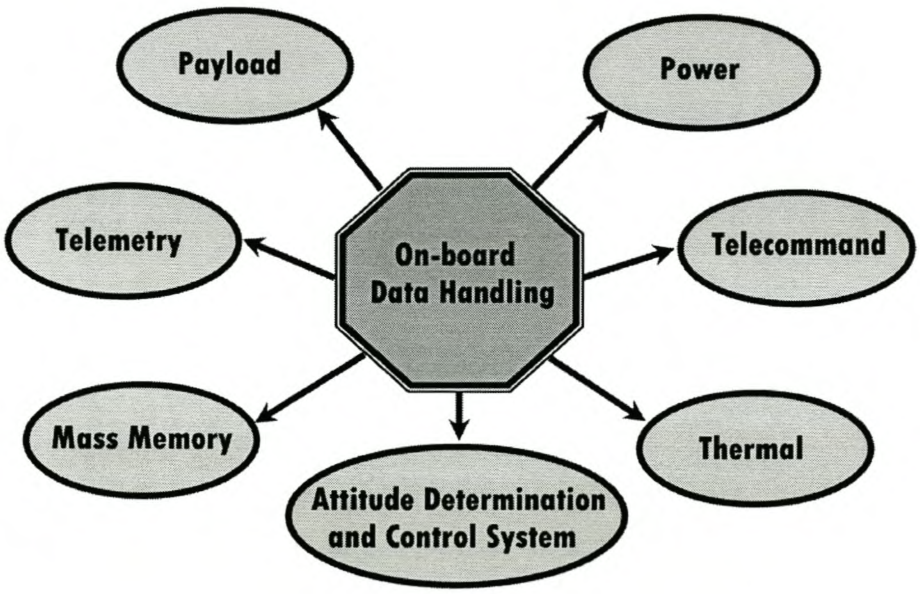


Figure 1.1: Satellite's Subsystems

is an on-board computer (OBC). Depending on the satellite's requirements, the OBC will typically run a real-time multitasking operating system that performs various tasks. These tasks can include payload operations, attitude control, orbit maintenance and basic housekeeping functions. There is usually a secondary OBC to help out with computing-intensive tasks and it can also acts as a complete backup if the primary OBC fails. All the major subsystems and payloads can have their own built-in microcontrollers that provide the low-level data processing and interfaces for that particular subsystem with the OBC.

System startup software (boot code) usually resides permanently in nonvolatile memory and is executed by the OBC at startup. All the primary software is also stored in nonvolatile memory and loaded into random-access memory (RAM) after booting, and executed in RAM. This software can be modified or upgraded at any time via the control ground station. The control ground station normally formulates all the telecommand instructions into a diary format for the satellite's OBC to execute at some point in the future. Data from the subsystems and payloads are also gathered by the OBC to be transmitted to the control ground station. The OBC can also issue its own commands in response to a subsystem's inputs. In short, a satellite is operated via the primary computer running a real-time operating system.

1.2 Modern Small Satellite Design

With the increasing development and miniaturization of advanced microprocessors and commercially-off-the-shelf (COTS) technologies, satellites are able to be made smaller and more cost-effective. This has enabled more countries to start with their own space research programs. The "smaller, faster and cheaper" design trends have been more successful in small companies and research groups than in the large aerospace companies [3]. Recent technological advances have led to integration of complex electronics systems onto a single chip, the so-called System-on-a-Chip (SoC). SoC technology is where not only the central processor is on the chip, but the memory and peripheral electronics as well. SoC technology has the following advantages:

- Reduction of system size
- Flexibility of the system
- Reduction of development period by using existing intellectual property (IP) models

This technology and the advent of Micro-Electro-Mechanical Systems (MEMS) can lead to even smaller, more intelligent satellites that can be mass-produced reasonable cheaply

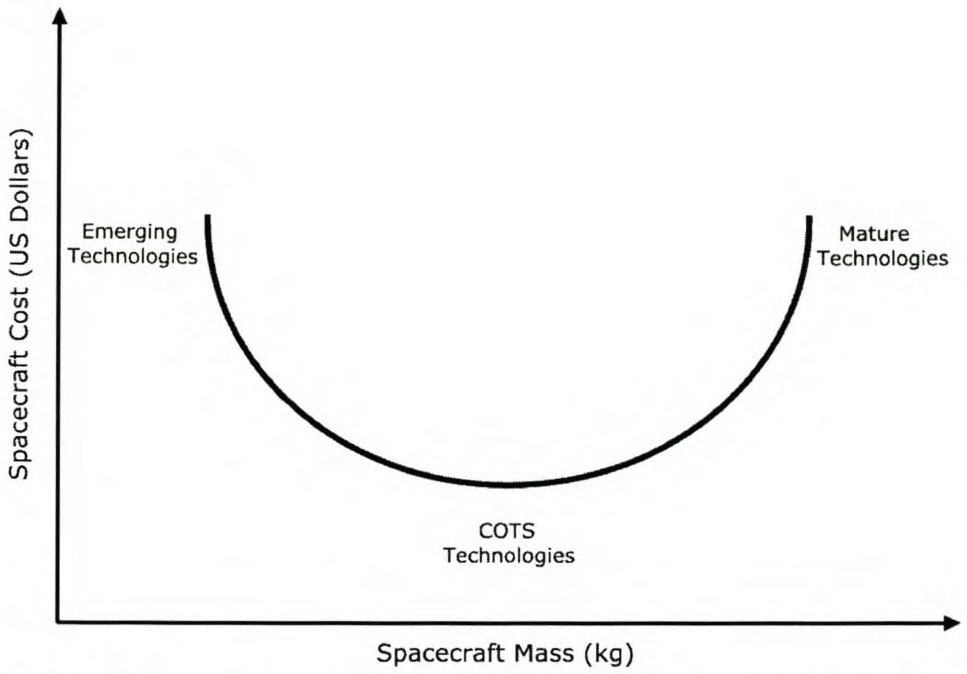


Figure 1.2: The use of COTS technologies to optimize spacecraft mass/cost

in the near future. The graph in Figure 1.2 illustrates how the use of COTS technologies optimizes spacecraft mass/cost.

1.3 An OBC as a SoC

The idea of implementing a satellite's OBC on a single chip is the result of the enduring trend to produce smaller satellites. Surrey Space Centre has already implemented this idea by scaling down an existing OBC to a SoC [5]. Their entire OBC, excluding memory, was implemented on a single high-density programmable logic array chip using soft IP cores. The results showed that it is possible to implement the functionality of a small satellite's OBC on a single programmable logic device. In 2001 the Altera Corporation introduced their Excalibur embedded processor. The Excalibur device combines programmable logic, memory and a processor core to allow the integration of an entire system on a single programmable chip. This document evaluates the use of Altera's Excalibur embedded processor for an OBC.

1.4 Document Outline

- Chapter 1: **Introduction**
  This chapter provides background information on the thesis subject.

- Chapter 2: **Suitability of the Excalibur Processor**
  The OBC processor requirements are covered and a suitable Excalibur embedded processor is chosen.

- Chapter 3: **The EPXA1 Embedded Processor PLD**
  Details of EPXA1 embedded processor PLD are discussed.

- Chapter 4: **Software Environment**
  A suitable software environment and operating system are chosen that will be used to program the EPXA1 device.

- Chapter 5: **Development Environment**
  A brief overview of the selected operating system and development tools are given in this chapter.

- Chapter 6: **EPXA1 OBC Design**
  This chapter covers the hardware and software design of the EPXA1 OBC.

- Chapter 7: **Tests and Measurements**
  In this chapter, the various OBC tests and results are discussed.

- Chapter 8: **Conclusions and Recommendations**
  This concluding chapter summarizes the advantages and disadvantages of an EPXA1 OBC. Some suggestions are also given on how to improve the EPXA1 OBC and where further testing is required.

Chapter 2

Suitability of the Excalibur Processor

The main concern when using COTS products is that they are not originally made for use in the harsh space environment. To get an idea whether a certain COTS processor can be used in space and as a satellite's OBC, it is necessary to draw a set of general requirements for the processor. In the following section a list of requirements are given that helps with the selection of a processor for a typical satellite's OBC. These requirements are then applied to the Excalibur family of embedded processors to determine the feasibility of the processor as a satellite's OBC.

2.1 Processor Requirements

The following list gives the requirements for a typical processor to be used as a satellite's OBC followed by a brief explanation. It must be stated that these requirements are not specific to any satellite and may differ according to each satellite's own requirements.

- **Operate in space**
  First and probably most important is that the processor must be able to function correctly in the space environment. It must be able to withstand the radiation effects that the space environment inflicts. The three main effects of radiation is single-event upsets (SEU), total dose damage and latch ups. To minimize the effects of radiation, the processor should have a large manufacturing process and the use of cache memory disabled. The processor must also be able to handle the thermal extremes experienced in space. For a LEO satellite the temperature ranges are typically between -15 °C and +45 °C. Another good indication is to determine if the processor has a space history and what the results were. For a more in depth look at aspects affecting processors in the space environment, consult [3] and [6].

- **Low power consumption, low power mode is necessary**
  Power is a limited resource on a satellite and it must be used very sparingly. Therefore, the processor must use as little power as possible and ideally contains a low power mode.

- **Processor speed**
  The processor must be fast enough to finish all its tasks on time.

- **Support for several interfaces**
  The processor must be able to interface with different types of memory and I/O devices.

- **General I/O ports**
  A sufficient number of general I/O ports are needed to support the various interfaces and other software implementations.

- **Additional built in functions**
  Any built in functions like timers or interrupt controllers that is useful for the satellite, will be very beneficial.

- **User friendly programming interface**
  The processor must be easy programmable or re-programmable.

2.2 Excalibur Models

The Excalibur device consists of the ARM922T processor core, embedded peripherals and a programmable logic device (PLD). There are currently three Excalibur models available from the Altera Corporation [7]. They are the EPXA1, EPXA4 and EPXA10. Figure 2.1 shows the Excalibur architecture of the three models. The three models contain the same ARM processor (ARM922T) core and peripheral devices. There are, however, differences in the on-chip memory, external memory interfaces, user I/O pins and PLD size. The EPXA1 device features proved more than enough for an implementation of a satellite's OBC. On this basis and the fact that the EPXA1 is the lowest cost member of the Excalibur family, it was chosen for this study. When this study was started, Altera provided an EPXA1 development kit. The development kit consisted of a development board with the EPXA1 device, development software and the necessary cables. It was therefore not necessary to design a development board to evaluate the EPXA1 device. A full description of the EPXA1 device and development kit is given in the next chapter.

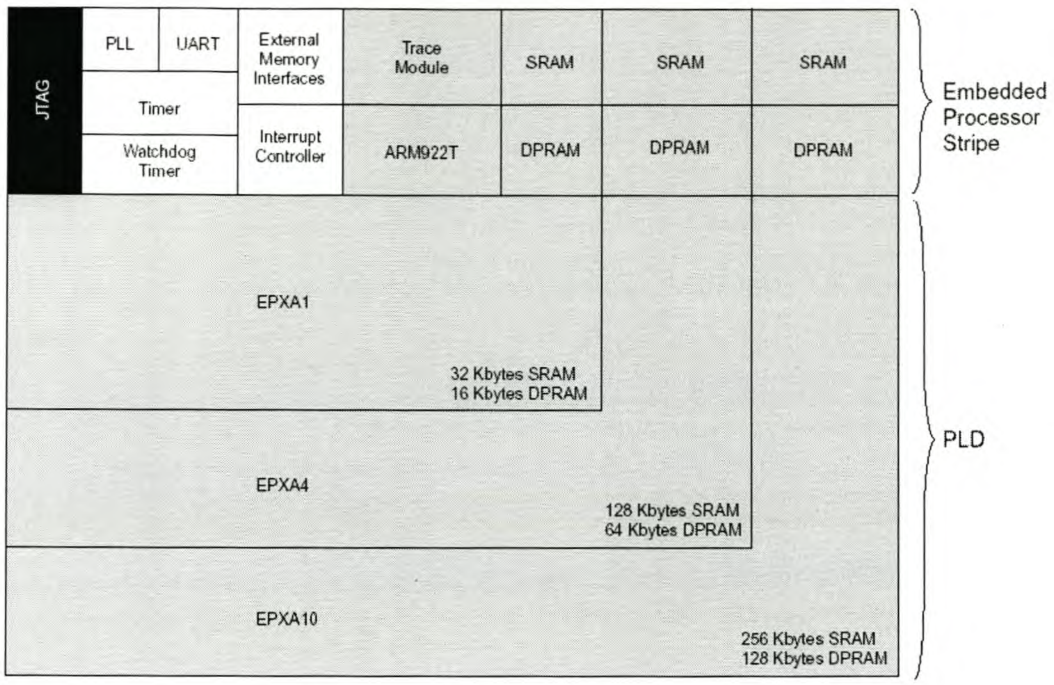


Figure 2.1: The Excalibur Architecture [1]

2.3 Feasibility of the EPXA1 Device as OBC

To evaluate the EPXA1 device according to the requirements stated in section 2.1, it is only necessary to look at certain aspects of the device. These aspects are summarized in Table 2.1.

The most concerning aspect that arises from Table 2.1 is that the EPXA1 device may not be sufficiently tolerant against the radiation present in the space environment. This is due to the very small process size of 0.18 μm. With such a small manufacturing process, the silicon density is high and the device will be very susceptible to the radiation. A feasible process size is at least 0.5 μm [8], but in previous studies [9], a process size of 0.35 μm was also acceptable. The PLD architecture uses SRAM configuration elements. SRAM type PLDs do not have strong tolerance against SEU caused by space radiation.

The commercial temperature range is very limited, but could be overcome with a good thermal design. The processor's cache must be disabled to minimize the effects of space radiation. Unfortunately this will cause a decrease in performance. The power consumption of the EPXA1 device is very hard to predict. The total power consumption is the sum of the stripe's (processor core and hard peripherals of EPXA1) power and PLD's power. Typical power requirements is a consumption of less than 1 W [8] for the processor. This could be compared to the power consumption of 160 mW for the ARM922T core, which

Table 2.1
EXCALIBUR CHARACTERISTICS

Characteristic	**Unit**	**Value**
Process size	μm	0.18
Supply voltage	V	1.8
Operating temperature	°C	0 - 85
Space history		No
Power consumption	W	$0.6948 + P_{PLD}$
Low power mode		Yes (only EPXA1)
Processor frequency (max)	MHz	200
MIPS[1]		210
Available User I/O		186

is well below this typical value.

Some more attractive features are the large number of user I/O pins available, the low power mode and maximum performance of 210 MIPS. The EPXA1 device also contains a few built in peripheral modules like the interrupt controller, UART and memory controllers, while the programmable logic allows the implementation of additional soft-core peripherals.

2.4 Conclusion

The EPXA1 device complies with all the general requirements of a satellite's OBC listed, except being radiation tolerant. This is mainly due to a too small manufacturing process size and the use of SRAM configuration elements.

The next chapter will discuss the details of the EPXA1 device and development board that was used in this study.

[1]Millions Of Instructions Per Second, measured with Dhrystone 2.1

Chapter 3

The EPXA1 Embedded Processor PLD

The EPXA1 device was chosen from the Excalibur family of embedded processors to implement an OBC as a SoC. In the following section an overview of the EPXA1 device will be given followed by details of the EPXA1 development board used to evaluate the EPXA1 device.

3.1 The EPXA1 Device

Altera's Excalibur EPXA1 (EPXA1F484) device is the lowest cost member in the Excalibur family. The device can be divided into two parts, the stripe and the PLD. The stripe contains the ARM922T processor, dual-port and single-port SRAM memories, peripherals and debug modules. The PLD section is similar to Altera's APEX20KE FPGA. Figure 3.1 shows the system architecture of the embedded stripe and the interfaces to the PLD portion of the device. The following sections will give a brief overview of each of the modules in the stripe.

3.1.1 The ARM922T Processor

The ARM922T processor is a member of the ARM9TDMI family of general-purpose RISC (Reduced Instruction Set Computer) microprocessors that includes the ARM9TDMI core, cache and a memory management unit (MMU). Figure 3.2 shows the functional block diagram of the processor. The ARM9TDMI processor core uses the ARM v4T instruction set. It supports the 32-bit ARM or 16-bit Thumb modes. The 32-bit architecture optimizes the data transfer efficiency and therefore improves the performance of the processor. The processor also supports big and little Endian modes and has a task identifier register specifically designed for real-time operating system support. It uses a five stage pipeline

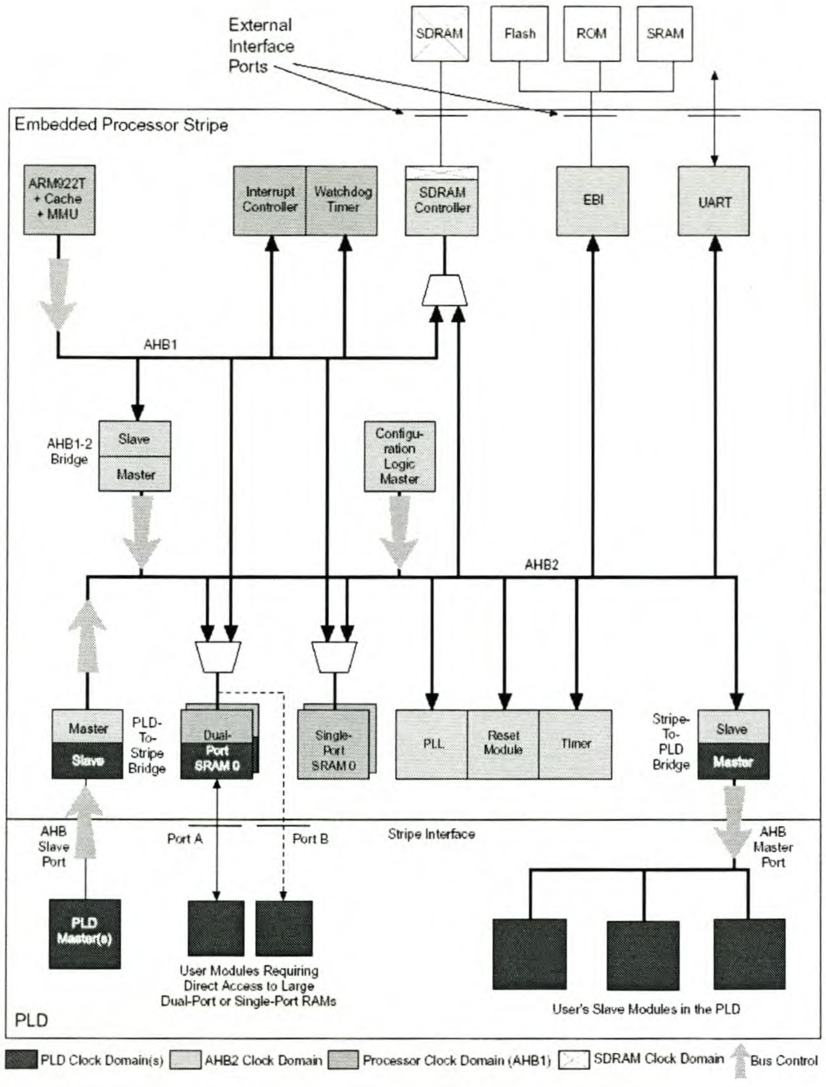


Figure 3.1: The Excalibur Device System Architecture [1]

(fetch, decode, execute, memory and write) and a Harvard architecture (separate instruction and data paths) to achieve a performance ratio of 1.05 MIPS per MHz. The processor contains 8 Kbytes of instruction and 8 Kbytes of data caches along with their associated MMUs. The processor also has a write buffer and AMBA bus interface. The cache can be disabled.

3.1.2 Bus Architecture

The Excalibur device implements the AMBA based advanced high-performance bus (AHB) architecture. AMBA is a high-performance bus standard designed by ARM. The device uses two AHBs, AHB1 and AHB2. Each bus has 32-bit address, read and write data buses.

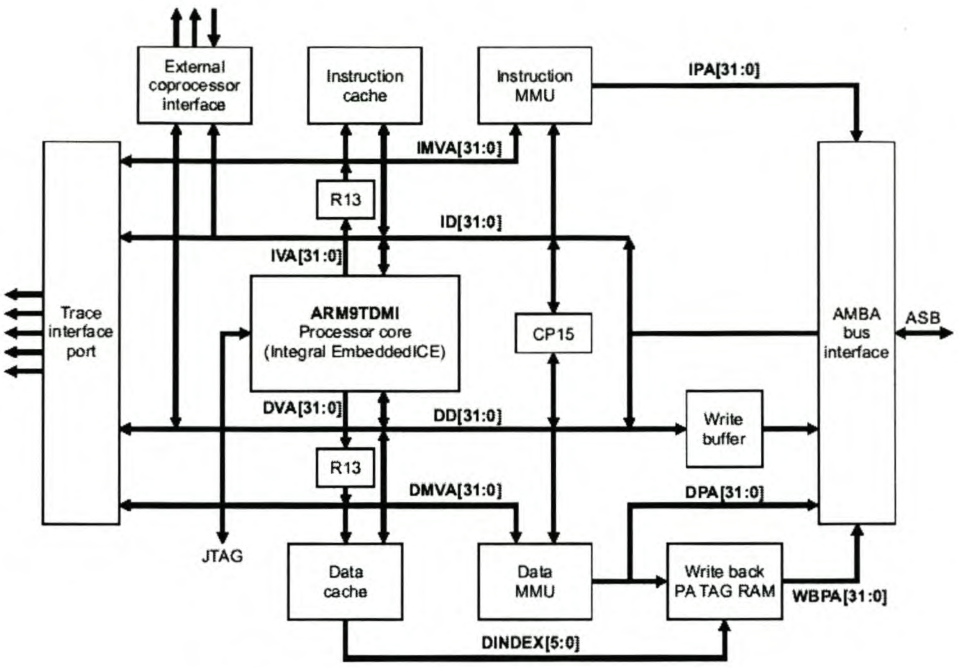


Figure 3.2: Functional Block Diagram of ARM922T [2]

3.1.3 On-Chip Memories

Two blocks of 16 Kbytes SRAM, operating at the AHB1 clock speed, are situated in the stripe. The memory is accessible by both AHB1 and AHB2 and the dual block architecture allows AHB1 and AHB2 to access one exclusive block at the same time.

Apart from the SRAM, there is also one block of 16 Kbytes dual-port SRAM (DPSRAM). The DPSRAM is not only accessible from the stripe (AHB1 and AHB2), but also from the PLD. This feature can serve as an application interface for sharing data between the processor and PLD.

3.1.4 External Memory Interfaces

An SDRAM controller and Expansion Bus Interface (EBI) serve as interface to external memory connected to the Excalibur device.

The SDRAM controller uses a 16-bit wide data bus and supports two blocks of up to 256 Mbytes SDRAM each. It supports single data rate (SDR) or double data rate (DDR) SDRAMs at up to 133 MHz or 266 MHz respectively.

The Expansion Bus Interface (EBI) is a 16-bit, bi-directional external memory interface. It provides an interface between AHB2 and external SRAM, flash or memory mapped devices. The EBI can accommodate four blocks of up to 32 Mbytes of external memory or memory-mapped peripherals.

3.1.5 Embedded Peripherals

The Excalibur contains various peripherals in the stripe. Each peripheral is briefly described in this section.

3.1.5.1 Interrupt Controller

The Interrupt controller interfaces with the processor's two interrupt inputs: interrupt request (IRQ) and fast interrupt request (FIQ). The interrupt controller allows up to 10 interrupts from peripherals within the stripe, one external interrupt from a pin and 6 interrupts from the PLD stripe interface. The six interrupt lines from the PLD can either operate as 6 individual IRQs or as an interrupt bus (up to 64 IRQs with a separate interrupt controller in PLD). The interrupt controller contains 24 configuration and status registers that are only accessible via the embedded processor. Each of the 17 interrupt sources has a corresponding priority register which contains its priority value and specifies whether it is FIQ or IRQ.

3.1.5.2 UART

The universal asynchronous receiver transmitter (UART) provides a basic low-speed (up to 230K Baud) interface between the device and other UART devices. It also has modem communication support.

3.1.5.3 Watchdog Timer

The watchdog timer is a one-shot interval timer used to protect a system against software bugs or other hardware failures.

3.1.5.4 Configuration Logic

The configuration logic module in the stripe is responsible for setting up the system for the embedded processor to boot. It is also used for transferring configuration data to the PLD array.

3.1.5.5 Phase Lock Loops (PLLs)

A reference input clock feeds two PLLs, PLL1 and PLL2, that provide the required internal clocks for the Excalibur device. PLL1 provides the embedded processor clock, CLK_AHB1, and the peripheral bus clock, CLK_AHB2. The CLK_AHB1 and CLK_AHB2 frequencies are one half and one fourth, respectively, of the PLL1 frequency. CLK_AHB1 can operate at a maximum of 200 MHz. PLL2 provides the clocks for the SDRAM controller.

3.1.5.6 Reset Module

When a reset occurs, it is the responsibility of the reset module to determine the cause of the reset. With this knowledge, it will reset the appropriate logic blocks.

3.1.5.7 General Purpose Timer

The general-purpose timer is a dual-channel, 32-bit timer with 32-bit pre-scaler. It can operate in three different modes: as a free-running (heartbeat) timer, a software controlled interval timer with interrupt-on-limit, or a one-shot interrupt after programmable delay.

3.1.6 Programmable Logic Architecture

The PLD section of the EPXA1 device is similar to that of Altera's APEX20KE programmable logic device. It has the following features:

- 4 160 logic elements
- 53 246 memory bits
- Up to 2 PLLs
- True Low Voltage Differential Signaling (LVDS) circuitry
- Support for the use of Altera's megafunctions[1]
- Embedded system blocks (ESBs) for memory support (CAM[2], FIFO, RAM, ROM and Dual-port RAM)

[1]Megafunctions are ready-made, parameterized, pre-tested blocks of intellectual property that are optimized to make efficient use of the architecture of the targeted programmable device.

[2]Content Addressable Memory (CAM) is a memory technology that accelerates search applications like databases, lists or patterns.

Table 3.1
MEMORY MAP PERIPHERALS AND ELEMENTS

Memory Map Element	**Range**
Registers	16 Kbytes
Internal SRAM0, SRAM1 (total)	32 Kbytes
Internal DPSRAM0 (total)	16 Kbytes
EBI0, EBI1, EBI2, EBI3 (each)	16 Kbytes to 32 Mbytes
SDRAM0, SDRAM1 (each)	16 Kbytes to 256 Mbytes
PLD0, PLD1, PLD2, PLD3 (each)	16 Kbytes to 2 Gbytes

- Support for various I/O standards

For more information on these features, consult [10].

3.1.7 Embedded Peripherals Memory Map

The memory-mapped slave peripherals of the EPXA1 device along with their sizes are listed in Table 3.1. The base address, size and the type of access permitted for each peripheral can be configured by its Range Definition Register. Only the base address for the Register's region is fixed at 7FFFC000H with a size of 16 Kbytes.

3.2 The Excalibur EPXA1 Development Kit

The EPXA1 Development Kit consists of a development board that contains the EPXA1 device, programming software and cables for interfacing with the development board from the outside world. The development board costs R4 414.83 and can be used over and over for other future projects.

3.2.1 The Development Board

The Excalibur Development Board is ideal to test and evaluate all the features of the EPXA1 device. A photograph of the development board is shown in Figure 3.3.

The EPXA1 development board has the following features:

- Powerful development board for embedded processor FPGA designs
    - EPXA1F484 device

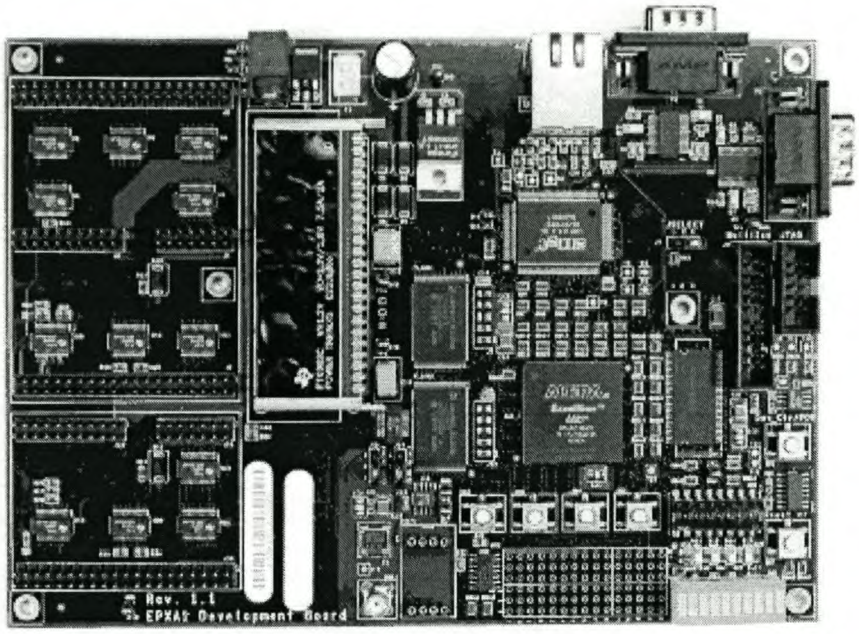


Figure 3.3: The Altera Excalibur EPXA1 Development Board

- Industry-standard interconnections
  - 10/100 Mbps Ethernet, two RS232 ports
- Memory subsystem
  - 8 Mbytes of flash memory, 32 Mbytes of single data rate SDRAM
- Multiple clocks for communication system design
- Multiple ports for configuration and debugging
  - IEEE Std. 1149.1 Joint Test Action Group (JTAG)
  - Configuration of EPXA1 device using flash memory with ByteBlaster II cable
  - Multi-ICE header for debugging
- Expansion headers for greater flexibility and capacity
  - 5-V standard expansion header, 5-V long expansion card header
- Additional user-interface features
  - One user-definable 8-bit dual in-line package (DIP) switch block
  - Four user-definable push-button switches, plus reset switch
  - Ten user-definable LEDs, plus function specific LEDs
- Test points provided for to facilitate system development

3.2.2 The Development Software

The development kit includes the Quartus II design software, the SOPC Builder system development tool, and GNUPro Toolkit developer tools.

Chapter 4

Software Environment

To evaluate and test each part of the EPXA1 device, the necessary software had to be developed. Embedded software is closely developed according to the specific processor's architecture. In this case it is the ARM architecture. A short description of the ARM architecture will be given for background. The next step will be to choose a suitable programming environment. Finally, an operating system (OS) was chosen to execute the development software.

4.1 The ARM Architecture

This section gives a short overview of the ARM v4T architecture.

The ARM architecture incorporates these typical RISC architecture features [2]:

- Large uniform register file
- A load/store architecture, where data-processing operates only on register contents, never directly on the memory
- Simple addressing modes, load/store addresses determined from register contents and instruction fields
- Uniform, fixed-length instruction fields to simplify instruction decode

In addition, the ARM architecture provides:

- Control over the Arithmetic Logic Unit (ALU) and shifter in all the data-processing instructions

Table 4.1
ARM v4 Processor Modes

Processor Modes	Description
User (usr)	Normal ARM Program execution state
FIQ (fiq)	Support data transfer or channel process
IRQ (irq)	Used for general-purpose interrupt handling
Supervisor (svc)	Protected mode for the operating system
Abort (abt)	Implements virtual memory and memory protection
Undefined (und)	Support software emulation of hardware coprocessors
System (sys)	Privileged user mode for the operating system

- Optimized program loops by auto-increment and auto-decrement addressing modes
- Load and store multiple instructions to maximize data throughput
- Conditional execution on all instructions to maximize execution throughput

4.1.1 The ARM Instruction Set

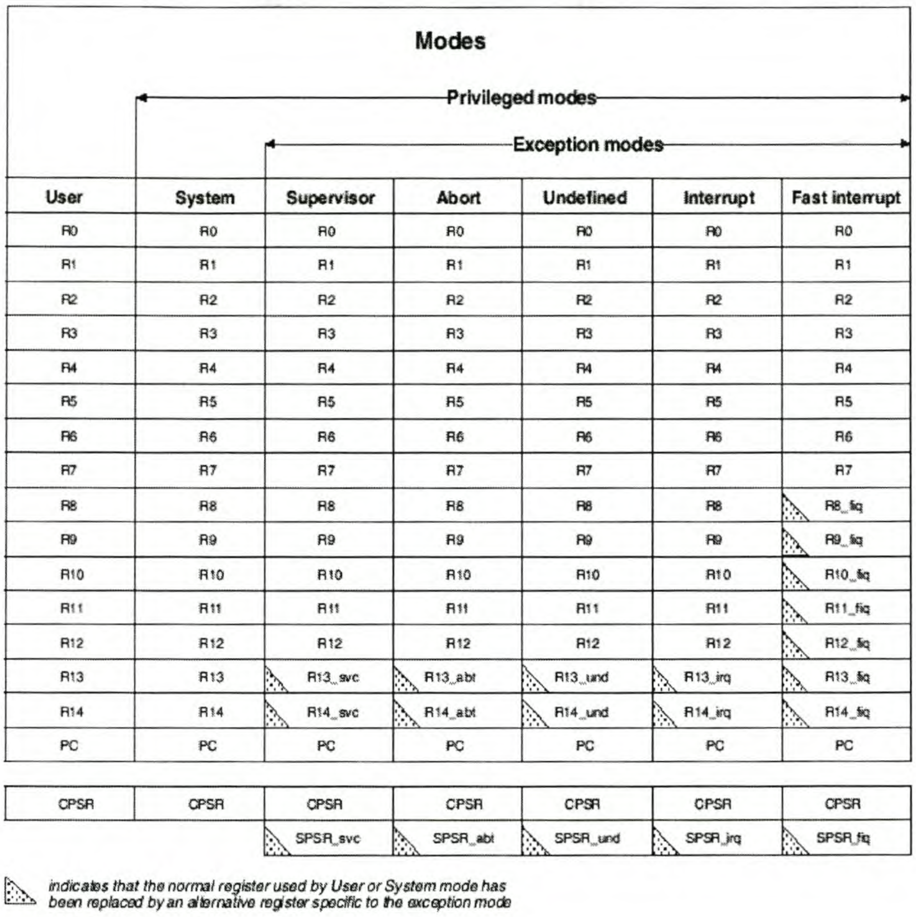
The ARM v4T architecture incorporates both a full 32-bit ARM instruction set and the 16-bit Thumb instruction set. A full description of the two instruction sets is given in [2]. The following sections are only applicable on the ARM instruction set.

4.1.1.1 Data Types

The ARM v4 architecture supports byte (8-bit), halfword (16-bit) and word (32-bit) data types.

4.1.1.2 Processor Modes

The ARM architecture supports the seven processor modes shown in Table 4.1. Application programs are usually executed in User mode. The only way a program running in User mode can access system protected resources or change the mode, is by causing an exception to occur. All the modes except User mode are known as privileged modes. They have full access to system resources and can change between modes freely. The modes FIQ, IRQ, Supervisor, Abort and Undefined are also known as exception modes because they are entered when specific exceptions occur. Each one has some additional registers to avoid corrupting the User mode state when the exception occurs.



Modes

Privileged modes

Exception modes

User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	▨ R8_fiq
R9	R9	R9	R9	R9	R9	▨ R9_fiq
R10	R10	R10	R10	R10	R10	▨ R10_fiq
R11	R11	R11	R11	R11	R11	▨ R11_fiq
R12	R12	R12	R12	R12	R12	▨ R12_fiq
R13	R13	▨ R13_svc	▨ R13_abt	▨ R13_und	▨ R13_irq	▨ R13_fiq
R14	R14	▨ R14_svc	▨ R14_abt	▨ R14_und	▨ R14_irq	▨ R14_fiq
PC	PC	PC	PC	PC	PC	PC

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		▨ SPSR_svc	▨ SPSR_abt	▨ SPSR_und	▨ SPSR_irq	▨ SPSR_fiq

▨ *indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode*

Figure 4.1: ARM Register Organization

The remaining mode, System mode, is a privileged mode and has the same registers available as User mode, but is not subject to the same User mode restrictions. It is intended to be used by operating system tasks.

4.1.1.3 Registers

The ARM has a total of 37 registers of which 31 are general-purpose registers and 6 are status registers. These registers are all 32 bits wide.

Each processing mode uses a different register bank as shown in Figure 4.1. At any point, 15 general-purpose registers, one or two status registers and the program counter are visible. The general-purpose registers (R0 - R15) can be divided into three groups, namely the unbanked registers (R0 - R7), the banked registers (R8 - R14) and the program counter, Register 15.

The unbanked registers refer to the same physical registers for all the processing modes. The use of these registers is solely for general purpose.

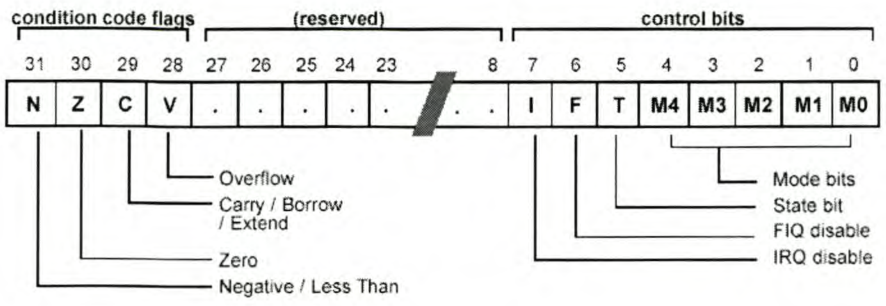


Figure 4.2: Program Status Register Format

The banked registers differ from the unbanked registers in the fact that the physical registers referred to by each of the banked registers depend on the current processor mode. Registers R8 - R12 are general-purpose registers which have two banked registers each. One is used in all processing modes other than FIQ mode, and the other is used in FIQ mode. Registers R13 and R14 have six banked physical registers each. Register R13 is normally known as the Stack Pointer (SP). Each exception mode has its own SP and Link Register (Register R14). The Link Register (LR) is used to hold a subroutine or exception return address. The subroutine or exception return is performed by copying the contents of the LR back to the program counter.

Register R15, as mentioned, holds the program counter.

The current program status register (CPSR) is accessible in all of the processor modes. It contains the status flags of the most recently performed ALU operation, interrupt disable bits, current processor mode, and other status and control information. Each exception mode has a saved program status register (SPSR) that is used to preserve the value of the CPSR when the exception occurs. The format of the CPSR and SPSR is shown in Figure 4.2. The N, Z, C, and V bits are the condition code flags. These may be changed as a result of arithmetic and logical operations, and may be tested to determine whether an instruction should be executed. The bottom eight bits (I, F, T and M[4:0]) are known as the control bits. These will change when an exception arises. If the processor is operating in privileged mode, they can also be manipulated by software. The T bit reflects the operating state, ARM or THUMB. The I and F bits are the interrupt disable bits. When set, these disable the IRQ an FIQ interrupts respectively. The five mode bits (M[4:0]) determine the processor's operating mode.

Table 4.2
EXCEPTION PROCESSING MODES

Exception Type	**Mode**	**Address**
Reset	Supervisor	0x0000 0000
Undefined Instructions	Undefined	0x0000 0004
Software Interrupt (SWI)	Supervisor	0x0000 0008
Prefetch Abort	Abort	0x0000 000C
Data Abort	Abort	0x0000 0010
IRQ (normal interrupt)	IRQ	0x0000 0018
FIQ (fast interrupt)	FIQ	0x0000 001C

4.1.1.4 Exceptions

Exceptions occur whenever the normal flow of a program has to be halted temporarily, for example to service an interrupt from a peripheral. The current processor state must be preserved, so that the original program can resume when the handler routine has finished, before the exception is handled. The ARM supports seven different exceptions that are shown in Table 4.2 along with the processor mode in which the exception is processed.

4.2 Programming Languages

This section and the next (section 4.3) are very dependent on each other. Firstly, in order to write application software a suitable programming language and compiler tools must be selected. This programming language must however be supported by the selected OS. Most of the OS's, that are discussed in the next section, are written in the C programming language and makes this the obvious choice.

4.2.1 The C Programming Language

The C programming language was developed in the early 1970s by researchers at the AT&T Bell Laboratories in New Jersey. It was originally developed with system programming in mind (UNIX), but is now used for the development of numerical and engineering software as well. It contains features of both high- and low-level languages.

4.2.2 Compiler Tools

4.2.2.1 ARM Developer's Suite

As part of the EPXA1 Development Kit, Altera supplied an ARM Development Suite (ADS) Lite especially for the Excalibur devices. This suite includes an assembler, a C compiler, a debugger and a graphical integrated development environment. Unfortunately, it has a 45-Day Evaluation period and is very expensive. For this reasons it was not used in this study.

4.2.2.2 GNUPro Toolkit

The GNUPro Toolkit from Red Hat is open source development software that includes a compiler, a debugger, binary utilities, and other tools compatible with a wide range of processors including the ARM. It was supplied with the EPXA1 Development Kit and is just as easy to get from the Internet.

The GNU compiler collection (GCC) is a complete set of tools for compiling programs written in C, C++ or Objective C. Compilation usually involves up to four stages (preprocessing, compiling, assembly and linking).

The GNU Debugger (GDB) is a debugging program that allows one to simulate a program or execute it directly on the target processor. GDB is able to upload and run a program, set breakpoints and view variables or memory on the processor.

The GNU Binary Utilities (binutils) have many functions. Two of the more important functions are the "objcopy" and "objdump" functions. The "objcopy" function can convert the executable output to a S-record file to be downloaded and executed by a ROM Monitor. The "objdump" function lets one display information about one or more object files.

The GNUPro Toolkit can run on both Windows (under Cygwin) and Linux operating systems. The GNUPro Toolkit was selected as the compiler tools for this study because the chosen OS (see section 4.3) needs the GCC compiler. For more information on the GNUPro Toolkit, consult [11].

4.2.3 VHDL Language

All the logic implemented in the PLD of the Excalibur device has to be written in a hardware description language (HDL). Two HDLs are in common usage today: the very high-speed integrated circuit hardware description language (VHDL) and Verilog. VHDL was chosen as the HDL because of prior knowledge of the language.

4.2.4 Altera's Quartus II

Altera supplied the Quartus II development suite along with the EPXA1 Development Kit. This suite was used to develop, compile and upload most of the PLD code.

4.3 Operating System Selection

As mentioned earlier (section 1.1), a satellite's OBC typically runs a multi-tasking real-time operating system. It has a very important job that includes scheduling events, keeping track of collected data and communicating to all the other subsystems to name but a few. It is therefore necessary to know that an OS can adequately run on the EPXA1 device. Again, the OS of a satellite is highly depended on its requirements. The next section describes some of the operating systems considered for this study.

4.3.1 Nucleus RTOS

The Nucleus real-time kernel is manufactured by a company called Mentor Graphics. It supports various embedded processors including the ARM922T. The kernel provides an extensive set of services to manage tasks, inter-task communication and synchronization, events, memory, timers, hardware interrupts and software signals. It is also portable across many C/C++ compilers. The software is sold in source code format and without royalty fees. Unfortunately the licenses for the Nucleus software are very expensive and was not considered for this study. For more information on the Nucleus RTOS, consult [12].

4.3.2 RTEMS

Real-Time Executive for Multiprocessor Systems (RTEMS) is an open source real-time operating system. Included with RTEMS is the GNU development environment suitable for cross development from Linux or Windows hosts to the target platform. Information about this RTOS for an ARM processor proved to be very sparse and was therefore avoided [13].

4.3.3 Linux

Linux is a free UNIX-type operating system originally created by Linus Torvalds with the assistance of developers around the world. Developed under the GNU General Public License, the source code for Linux is freely available to everyone. Off-the-shelf Linux presents three critical challenges for use in applications with real-time performance requirements [14]:

- lengthy blocking times
- non-preemptive Linux kernel events
- exhaustive, fairness-based scheduling

However, nowadays there are many ways to add real-time capabilities to Linux based systems. One of the real-time Linux distributions that Altera uses in its reference designs is from MontaVista Software [15]. MontaVista Linux Professional Edition fully supports the Excalibur EPXA1 board, but it is commercial software. A less expensive solution was needed. The author did try to port a Linux kernel to the EPXA1 board, but without any success. A lot of time was spent building a cross tool chain for an ARM-Linux target. Success came after weeks of work. Another, bigger problem arised. There was no solution to get the Linux-based ARM boot loader (ARMBoot) running on the EPXA1 board without the Quartus software mentioned earlier. During this period, development was done in the Linux environment and there was no access to a licensed version of the Quartus software for Linux Workstations. Quartus could be run in the Windows environment, but then ARMBoot didn't find the required Quartus files it needed.

After reaching a dead end, an alternate solution was needed.

4.3.4 eCos

The Embedded Configurable Operating System (eCos, [16]) proved to be that solution. eCos is an open-source, royalty-free RTOS free of charge. It consists of a highly modular real-time kernel that was developed by Red Hat especially for embedded systems. eCos is used as the OS for Canada's Smallest Satellite, the CanX-1, as well as Denmark's DTUSat. Both were launched on 1 July 2003.

eCos has a configuration tool that allows the user to add or remove certain components to customize it for a specific application. This ensures that the minimum amount of memory is used. A boot loader, RedBoot, is also supplied as an eCos-based application.

eCos supports a wide variety of embedded processors including the ARM. There is also an existing eCos port to the EPXA1 board available which means that there is no need to write any additional code for the hardware. eCos uses the GNU cross-development tools as described in the previous section and can run in a Linux or Windows environment. For support there is a very active mailing list and online documentation [17].

4.4 Conclusion

Software designs were written in the C programming language and hardware designs in VHDL. The eCos RTOS was selected due to its existing port to the EPXA1 board and excellent support. It was run in a Windows-based environment. The GNU cross-development tools (running in the CYGWIN environment) were used for software compiling and the Quartus II v4 development suite for the hardware designs.

Chapter 5

Development Environment

The eCos RTOS was chosen as the most suitable operating system for the EPXA1 device for this study. This chapter gives a brief overview of the eCos RTOS along with the other development tools that were used to configure and test the EPXA1 device.

5.1 Introduction to eCos

ECos is an open-source real-time operating system specifically designed for embedded systems. The highly configurable nature of eCos allows it to be used on various platforms and applications with real-time requirements. Some of the eCos features are as follows:

- Royalty-free
- Open Source
- Source-configurable kernel
- ISO C and math library
- Open source TCP/IP stack
- GNU Debugger (GDB) support

eCos has a configurable component architecture consisting of a variety of software components. These different software components are added or removed from the system in order to create an embedded system that matches the requirements of the design application. Figure 5.1 shows a typical embedded software system built with eCos. The end product of such a component configuration is a library that can be linked with the application code. The Hardware Abstraction Layer, kernel and device drivers are some of the key eCos components and a short description of each are given in the following sections.

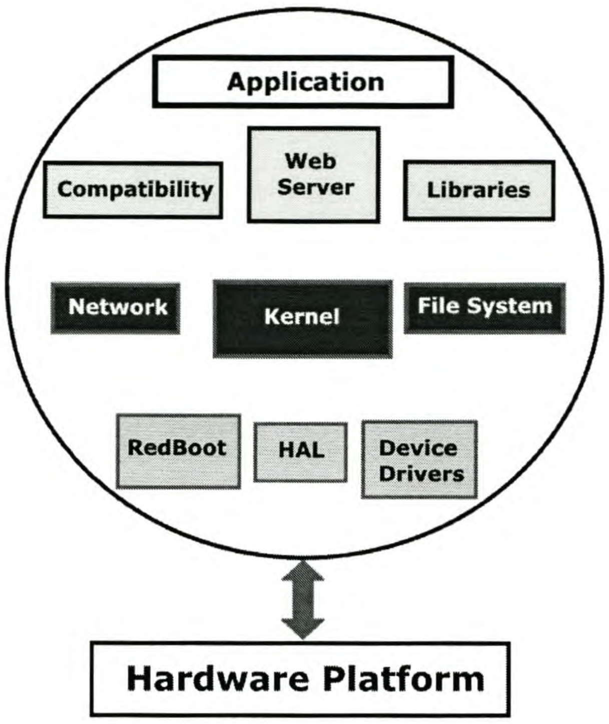


Figure 5.1: eCos System Architecture

5.1.1 Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) provides a software interface that gives general access to the system hardware. The HAL consists of three levels: architecture, variant and platform.

- **Architecture**
  The architectural layer defines the processor family. It contains the code required for CPU startup, interrupt delivery, context switching, and other functions specific to the instruction set architecture of the processor family. The EPXA1 board contains an ARM processor which indicates that the HAL architecture will be that of the ARM family.

- **Variant**
  This level describes the specific processor within the processor family. This will be the ARM9 variant of the ARM family.

- **Platform**
  The platform will point to the specific piece of hardware that contains the processor described by the previous two levels. It typically includes code for the platform to startup, chip configurations and interrupt controllers. In this case the platform will be the EPXA1 device.

The HAL provides many ways for interfacing with the kernel, other eCos devices or user applications. The main HAL interfaces are as follow:

- **Base Type Definitions**
  These are definitions that characterize the properties of the base architecture that are used to compile the portable parts of the kernel, i.e. type definitions, endianness, and labeling.
- **Interrupt Handling**
  These interfaces include definitions of exception and interrupt numbers, interrupt enabling and masking, and real time clock operations.
- **Input/Output Support**
  It provides access to device control registers by means of register read and write macros.
- **Diagnostic Support**
  Provides low-level diagnostic IO support via the UART or other serial IO devices.

5.1.2 eCos Kernel

The kernel provides the core functionality needed for developing multi-threaded applications. Standard functional components like interrupt and exception handling, scheduling, threads and synchronization are all configurable under the eCos system to meet design specifications.

5.1.2.1 Interrupt and Exception Handling

The kernel provides support for installing interrupt handlers and for controlling interrupts when they occur. The support is mainly used by the eCos device drivers and by application code that requires direct interaction with the hardware. The functionality provided by the kernel is mostly separated from the details of the underlying hardware which simplifies the application development. eCos provides several supporting functions for installing, masking, uninstalling, enabling and disabling interrupt handling.

5.1.2.2 The Scheduler

The purpose of the scheduler in a multi-threaded system is to determine which thread should currently be running. The eCos kernel can be configured with one of two schedulers, a bitmap or a multi-level queue (MLQ) scheduler. The bitmap scheduler allows only one thread per priority level as well as pre-emption between the different priority levels. The MLQ scheduler, on the other hand, allows multiple threads to run at the same priority level and also supports time slicing and pre-emption between different priority levels. Both these schedulers use priority levels to determine which thread should be running. There is typically 32 priority levels with 0 being the highest and 31 the lowest priority.

5.1.2.3 Synchronization Mechanisms

The kernel provides different mechanisms for threads to communicate with each other and to share the system's resources. The synchronization primitives provided by the eCos kernel are mutexes, semaphores, condition variables, event flags and mail boxes.

- **Mutexes**
  A mutex (mutual exclusive object) allows the sharing of system resources between threads in a safe manner. A thread locks a mutex, manipulates the shared resource and then unlocks the mutex again. One problem arises when a high priority thread is waiting on a mutex that is currently locked by a low priority thread. This is better known as the priority inversion problem and eCos provides two solutions to this problem: the priority ceiling protocol and the priority inheritance protocol. The priority ceiling protocol raises the priority of all the threads that needs the mutex to a preconfigured value. However, this solution has many disadvantages in a real time environment. A better solution is the priority inheritance protocol. This protocol raises the priority of the thread that currently owns the mutex to the same level as the highest level thread waiting for the mutex.

- **Semaphores**
  A semaphore is used to indicate whether a resource is locked or available. eCos uses a counting semaphore. Counting semaphore objects contain a value that is incremented when a thread posts to a semaphore and decremented when a thread completes a wait for the semaphore.

- **Condition Variables**
  Condition variables are used in conjunction with mutexes to implement long-term waits for some condition to become true. A thread can typically wait for data that is not yet available.

- **Event Flags**
  Flags can be used to signal that a certain event has occurred. This is achieved by associating bits in a 32-bit word with different events.

- **Mail Boxes**
  Mail boxes are also used to indicate that a particular event has occurred and allows for one item of data to be exchanged per event. This item of data is usually a pointer to some other section of data that can be exchanged between the two threads.

5.1.3 Device Drivers

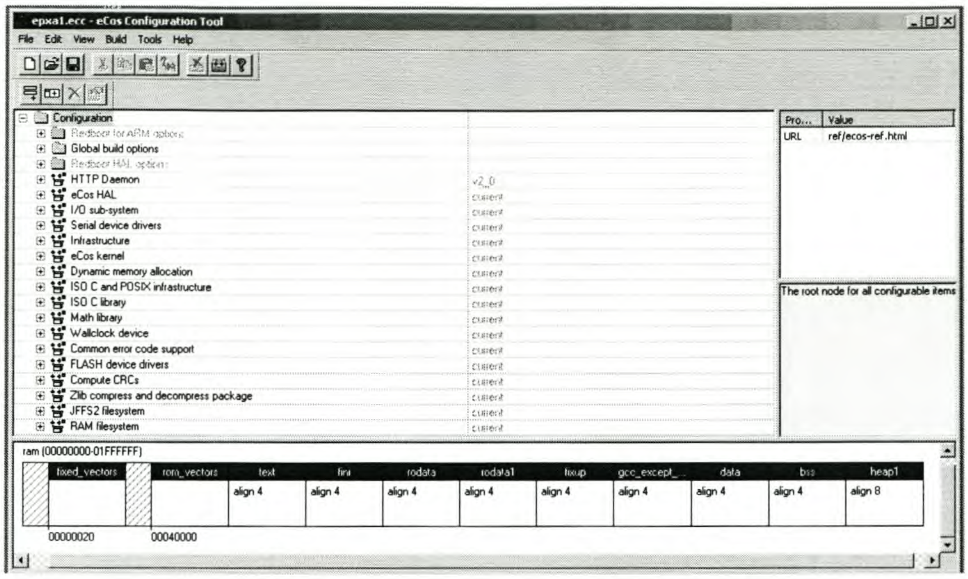
Device drivers control the various hardware components in a system. A device driver must be written in such a way that allows the I/O Sub-System package to present a standard interface to the higher-level software modules. The I/O Sub-System package provides an intermediate layer between the hardware device and eCos application.

5.2 Cygwin Tools

Many useful software tools for embedded systems are developed to run in a UNIX or Linux environment. Cygwin acts as an intermediate layer for these software tools to a Windows environment. The Cygwin tools are ports of the popular GNU development tools and utilities for Windows systems. They function through the use of the Cygwin library which provides the UNIX system calls and environment that these programs require. The Cygwin tools were mainly used to build the eCos cross compiler.

5.3 eCos Configuration Tool

The eCos Configuration Tool is used to construct eCos at source level, prior to compilation or assembly, and provides a configuration file and a set of files used to build user applications. The sources and other files used for building a specific configuration are provided in a component repository, which is loaded when the eCos Configuration Tool is invoked. The eCos Configuration Tool can be used in a command line interface or graphical interface under the Windows operating system. The graphical interface, as shown in Figure 5.2, was used in this study. The eCos Configuration Tool (version 1.3.net) also provides a runtime memory layout tool for configuring the memory layout of the application together with eCos. The eCos Configuration Tool provides templates for various development boards. When targeting eCos for the EPXA1 development board, the EPXA1 template is chosen from the list of templates. The Configuration Tool also has a package management tool for adding or removing packages from the template to customize eCos according to the design requirements.


epxa1.ecc - eCos Configuration Tool
File Edit View Build Tools Help

Configuration	
Redboot for ARM options	
Global build options	
Redboot HAL options	
HTTP Daemon	v2_0
eCos HAL	current
I/O sub-system	current
Serial device drivers	current
Infrastructure	current
eCos kernel	current
Dynamic memory allocation	current
ISO C and POSIX infrastructure	current
ISO C library	current
Math library	current
Wallclock device	current
Common error code support	current
FLASH device drivers	current
Compute CRCs	current
Zlib compress and decompress package	current
JFFS2 filesystem	current
RAM filesystem	current

Pro...	Value
URL	ref/ecos-ref.html

The root node for all configurable items

ram (00000000-01FFFFFF)

fixed_vectors	rom_vectors	text	fini	rodata	rodata1	fixup	gcc_except_...	data	bss	heap1
		align 4	align 4	align 4	align 4	align 4	align 4	align 4	align 4	align 8
00000020	00040000									

Figure 5.2: eCos Configuration Tool

5.4 Booting the EPXA1 Device

There are two methods of configuring the EPXA1 device and making code available at the boot address. These methods are to boot from the flash memory or to boot from an external source. The external pins BOOT_FLASH, MSEL1 and MSEL2 are used to specify the boot method. However, on the EPXA1 development board, these pins are hardwired, so the device boots from a 16-bit flash memory. This means that the device will always boot from the flash memory.

5.4.1 Boot from Flash Mode

In this mode the processor accesses the boot code from the 16-bit flash memory connected to EBI0. The bottom 32 Kbytes of EBI0 are mapped at address 0H. The registers are mapped to their default addresses with the base being 7FFFC000H. It is then up to the boot code to do the rest of the startup procedures (see section 5.4.5).

5.4.2 Boot Code

The EPXA1 device starts its "life" (out of reset) by fetching its first instruction at address 0H. At this address should be the first instruction of the boot code. The boot code is responsible for initializing the EPXA1 device. There was an option of choosing suitable boot code from two available sources. Altera provided boot code for the EPXA10 device that can be easily modified for the EPXA1 device. The other source, eCos or RedBoot, also contains boot code specifically for the EPXA1 device. The latter was chosen due to the fact that the code was specific for the EPXA1 device and the chosen RTOS is eCos. With this option the RedBoot ROM Monitor can be used for debugging purposes.

5.4.3 Linker Script

A linker script is used to link an eCos application. This script typically defines the memory areas, addresses and sizes, into which the code and data are to be put, and allocates the various sections generated by the compiler to these areas. The linker script file (*target.ld*) is created by the eCos Configuration Tool out of a base linker script file and *.ldi* file that was generated by the memory layout tool.

5.4.4 The RedBoot ROM Monitor

The RedBoot (Red Hat Embedded Debug and Bootstrap) ROM Monitor provides a debug and bootstrap environment for an embedded system. RedBoot is a standalone program that can be used with any RTOS, but was exclusively used with eCos applications in this study. RedBoot is configured the same way as an eCos application and therefore can support all the features of the EPXA1 development board.

5.4.5 RedBoot Startup Procedure

The "RedBoot" startup mode is selected when configuring RedBoot to run on the EPXA1 device. In this startup mode RedBoot is stored in the flash memory while a small amount of SDRAM is used for runtime usage. RedBoot uses the eCos HAL for its foundation. This means that at startup the HAL initialization sequence is followed. An extract from the initialization routines for the EPXA1 HAL, performed by the boot code of the EPXA1 device, are explained in the following steps:

- **Hardware Powerup or Reset**
  At hardware powerup or reset, the CPU jumps to the reset vector at address 0H. The assembler boot code resides here.
- **Sets up Exception Vectors**
  The exception vector table (see Table 4.2), with their respective branch instructions to their exception handlers, are defined. When the booting is complete this

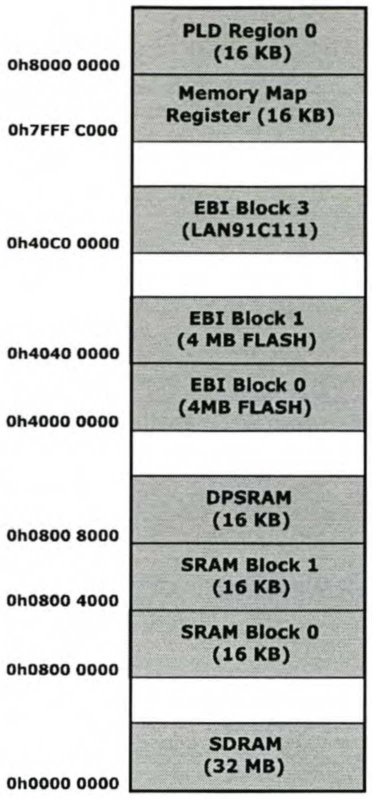


Figure 5.3: EPXA1 Memory Map Layout

table will reside at address 0H, the fixed interrupt vectors taken by the ARM922T microprocessor.

- **Configure and Enable Embedded Stripe PLLs**
  The two PLLs, PLL1 and PLL2, are configured and enabled to the desired frequency. PLL1 is used to synthesize the embedded processor clock and the AHB system clocks. PLL2 is used to synthesize the SDRAM controller clock. Until the PLLs are configured and enabled, the system is clocked directly by the input reference clock (CLK_REF = 25 MHz).

- **Setup Memory Map**
  Before the peripherals and memory can be used, it must be mapped to a base location in the memory space, assigned a size and enabled. This is accomplished by writing the appropriate values to the memory mapped registers. Figure 5.3 shows the memory map layout for the EPXA1 development board as setup by the boot code.

- **Enable Instruction Cache**
  The ARM922T's memory management unit (MMU) controls instruction- and data cache operations. The MMU is implemented as a coprocessor (coprocessor #15).

The instruction cache is enabled after memory mapping to prevent invalid addresses in the cache. As mentioned (section 2.3), the use of cache memory is not advisable in a space environment. However, in the boot process the instruction cache was needed to initialize the SDRAM Controller (see Configure SDRAM Controller and Initialize SDRAM).

- **Configure SDRAM Controller and Initialize SDRAM**
  The SDRAM controller is configured for single data rate (SDR) SDRAM. Specifically, the controller is setup to interface with the Micron MT48LC16M16A2 SDRAM on the development board. First, a wait state assures that PLL2 has been locked for 100μs. At this point the registers SDRAM_TIMING1, SDRAM_TIMING2, SDRAM_CONFIG, SDRAM_REFRESH, SDRAM_ADDR, and SDRAM_MODE0 are loaded. These registers configure how the SDRAM controller will interface with the SDRAM device.

  After the SDRAM controller has been configured, the SDRAM device attached externally to the controller must be initialized. This initialization process must complete within one SDRAM refresh cycle ($\pm$7.81μs). For this reason the initialization code must run as fast as possible. To accomplish this, the SDRAM initialization code is locked into the instruction cache and then executed.

- **Copy Boot Code from Flash to SDRAM**
  Copy the *.rom_vectors*, *.text*, *.fini*, *.rodata*, *.rodata1*, *.fixup*, *.gcc_except_table*, and *.data* sections from the flash to the SDRAM memory. Continue executing the boot code from the SDRAM.

- **Initialize Memory Management Unit**
  Initialize the MMU, which handles the translation of logical addresses to physical addresses.

- **Initialize Stack**
  The stack is set up so that C function calls can be made from within the boot code (*vectors.s*).

- **Clear BBS**
  Clear the BSS section which contains noninitialized local and global variables.

- **Invoke Constructors**
  All global C++ constructors are called from the routine *cyg_hal_invoke_constructors*. The linker handles the generation of the list of global constructors.

- **Start Kernel**
  The routine *cyg_start* is called to start the kernel initialization.

Table 5.1
SBI FILE FORMAT

Offset	Size (bytes)	Data
0H	4	Signature "SBI\0"
4H	4	IDCODE for target system.
8H	4	Offset to configuration data (*coffset*).
CH	4	Size of configuration data in bytes (*csize*).
coffset	*csize*	FPGA configuration data.

5.5 Configuring the PLD

The PLD can be configured or reconfigured at any time. The PLD was configured via the JTAG interface using the Quartus II software or under processor control. In both methods the Quartus II software is used to generate the PLD configuration data. The data is stored in the slave-port binary file (*.sbi*) format showed in table 5.1. Configuring the PLD via the JTAG interface requires the Byteblaster II cable. The PLD is then easily configured by using the built-in programmer of the Quartus II software.

Configuring the PLD under processor control can be done with an eCos procedure. To make the PLD configuration data file, which is in the *.sbi* file format, more compatible with an eCos procedure, it was necessary to convert the *.sbi* file to a C-type array of characters. The configuration logic module in the stripe is responsible for transferring the configuration data to the PLD array. It has a simple interface to application code. The flowchart in Figure 5.4 shows the steps used to configure or reconfigure the PLD logic by an eCos procedure. For more information on configuring the PLD, consult [18]. The eCos procedure for configuring the PLD is included on the attached CD-ROM.

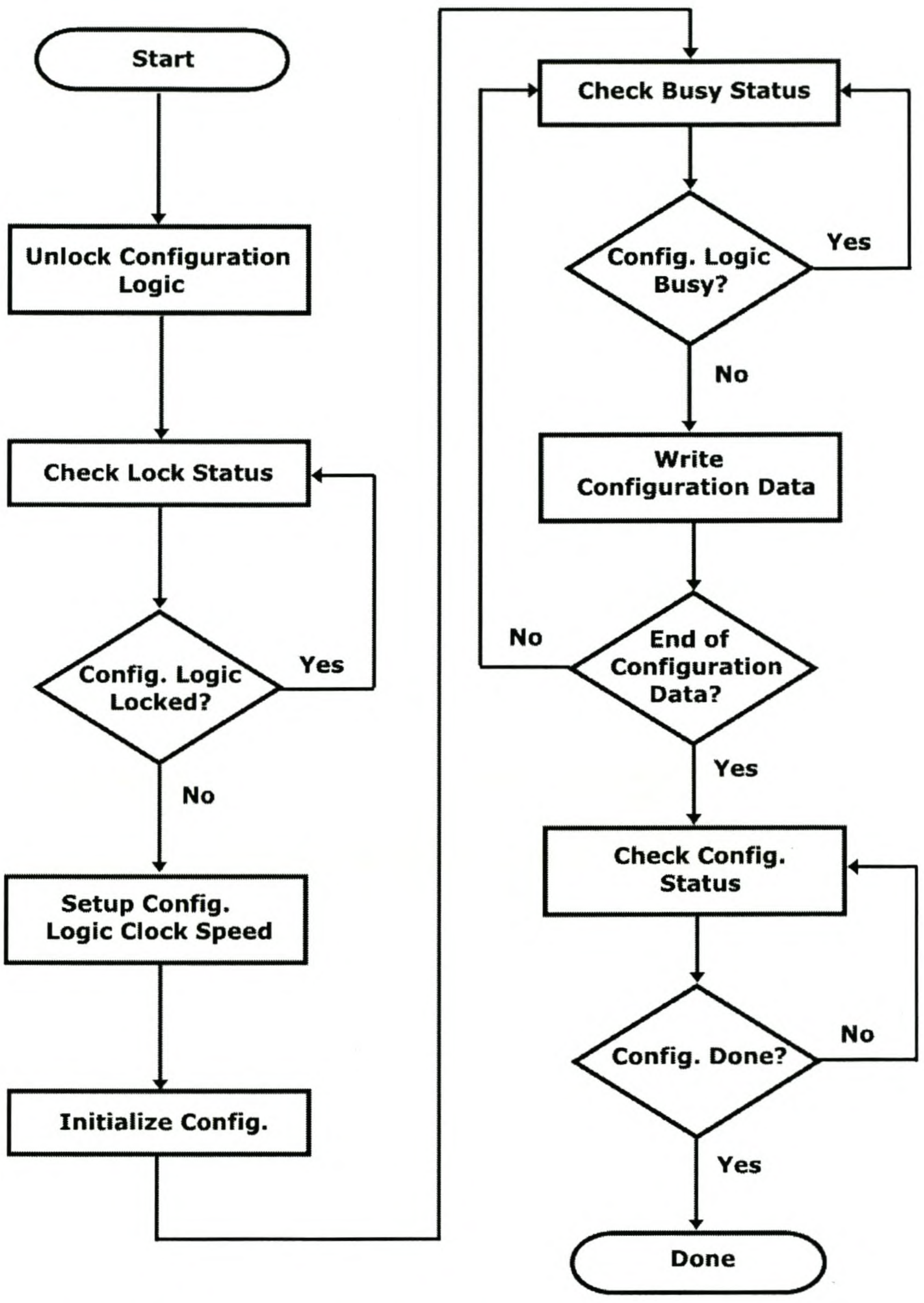


Figure 5.4: Flow Diagram for Configuring the FPGA Contents

Chapter 6

EPXA1 OBC Design

This chapter describes the implementation of an OBC on the EPXA1 device. In most cases the functional specifications for a satellite's OBC are determined by its system requirements. The EPXA1 OBC, however, was not intended for a specific satellite project, so the implementation was based on a general OBC model shown in Figure 6.1.

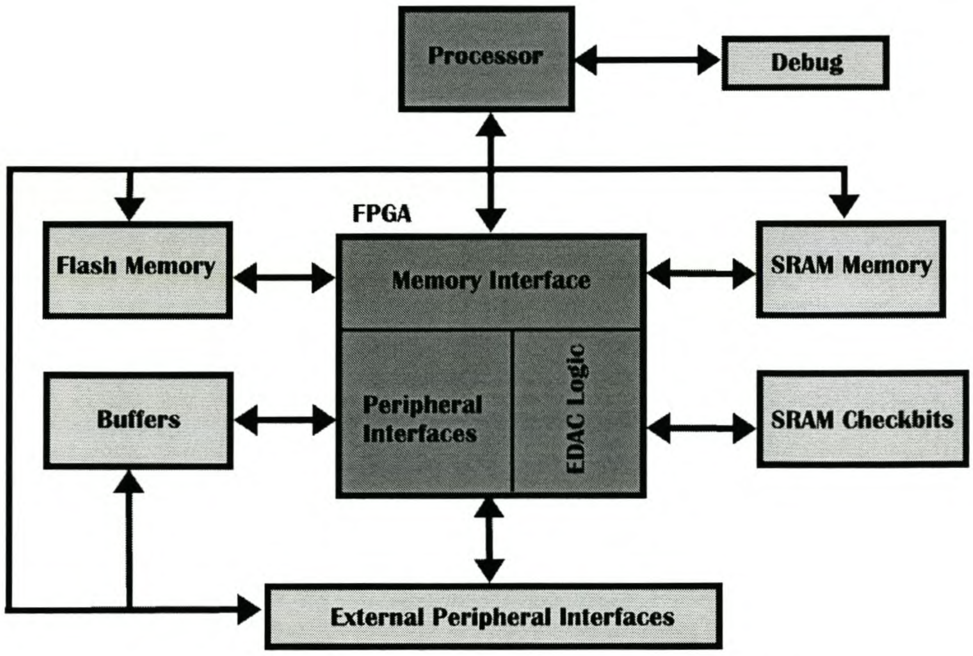


Figure 6.1: General Satellite OBC Model

6.1 EPXA1 OBC Design Overview

The main objective is to implement a similar type of OBC as in Figure 6.1 on the EPXA1 device. The OBC must preferably have a wide variety of interfaces to support different memory types and to communicate with the other satellite subsystems. To protect RAM devices against SEU induced errors, an error detection and correction (EDAC) unit must be considered. The entire OBC system consists of the following:

- Processor
  - ARM922T
- Memory Interfaces
  - SDRAM Controller
  - Expansion Bus Interface for flash, ROM or SRAM devices
- Bus System
  - AMBA AHB Busses
- On-chip Peripherals
  - Interrupt Controller
  - Watchdog Timer
  - UART
  - Reset Module
  - Timers
- PLD Implemented Modules
  - AHB Bus Interface
  - EDAC Unit
  - UART
  - I^{2}C Controller
  - PLD Interrupt Controller

Figure 6.2 illustrates the EPXA1 OBC model. The various OBC interfaces, peripherals and additional implementations will be discussed in the following sections. The complete Quartus block diagram of the implemented PLD modules is shown in Appendix A. All of the designs are included on the CD-ROM attached to the thesis.

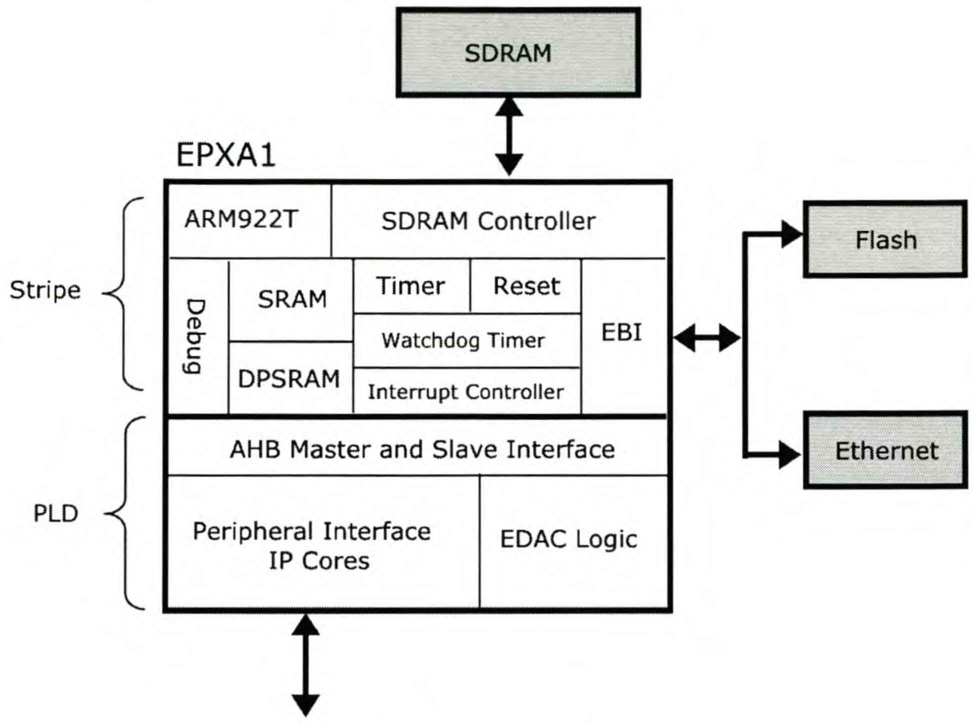


Figure 6.2: EPXA1 OBC Model

6.2 Processor Setup

One of the requirements for the OBC is that the cache of the processor must be disabled because it is prone to the radiation effects of the space environment. By default, the data and instruction caches are disabled at startup. During the startup process the data and instruction caches are enabled (see section 5.4.5). However, it can be disabled at any time by using the eCos macros defined to control the cache. Disabling the cache will decrease the power consumption of the processor, but at the same time decrease its performance. Unfortunately, no information on the performance of the ARM processor with the cache disabled could be found and therefore had to be tested (see section 7.2).

The ARM922T processor's two clock inputs, BCLK and FCLK, are directly connected to the AHB1 clock. This means that the processor's clock frequency is the same as the AHB1 clock. This clock frequency can be adjusted by setting the PLL1 to twice the desired processor speed. At startup the processor's clock frequency is set to 150 MHz by the boot loader.

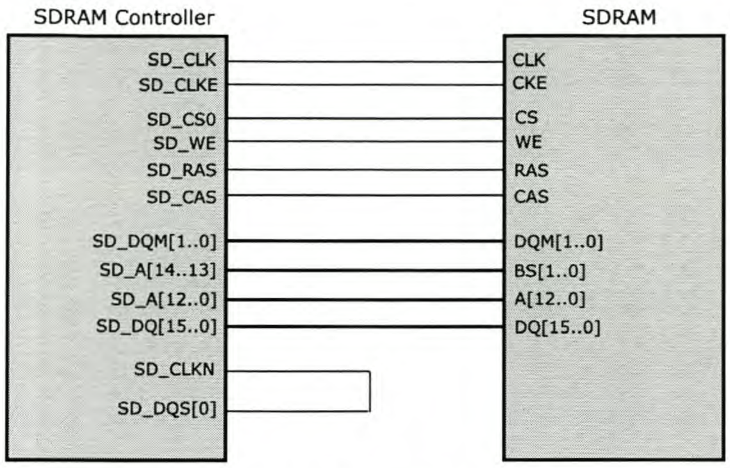


Figure 6.3: EPXA1 SDRAM Controller Connection to SDRAM

6.3 Memory Interfaces

6.3.1 SDRAM Controller

The SDRAM controller provides an interface between the external SDRAM memory and the internal busses of the EPXA1 device. This means that both AHB1 and AHB2 busses can access the SDRAM via the SDRAM controller. The SDRAM controller supports both 16-bit Single Data Rate (SDR) SDRAM and 16-bit Double Data Rate (DDR) SDRAM. The EPXA1 development board contains one 16-bit SDR SDRAM device (Micron MT48LC16M16A2) connected to the EPXA1 SDRAM controller. Figure 6.3 shows the connection between the SDRAM controller and the single 16-bit SDR SDRAM device. The SDRAM controller interfaces with the external SDRAM memory through data, address and control lines. SDRAM memory is organized into banks, rows and columns. The EPXA1 SDRAM controller requires that any SDRAM device that connects to it, has two bank-select lines. The bank-select lines are connected to the most significant address bits (SD_A[14..13]). Commands are sent to the SDRAM from the SDRAM controller through the control lines WE, RAS and CAS. For further information about the data, address and control lines, consult [1].

6.3.2 Expansion Bus Interface (EBI)

The EBI is a very flexible memory interface. It acts as an interface between external devices, such as flash memories or memory-mapped devices, and the AHB2 bus. It runs synchronously to AHB2 and supports all transaction types associated by the AMBA bus architecture. Figure 6.4 shows the EBI block diagram.

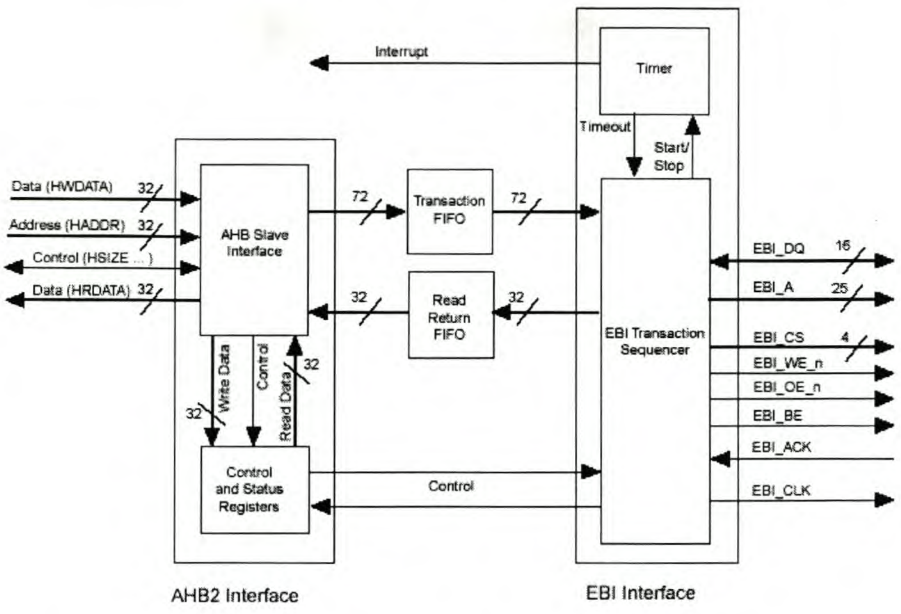


Figure 6.4: Expansion Bus Interface Block Diagram [1]

The EBI consists of two interfaces, the AHB2 interface and the EBI interface. These two interfaces communicate through the use of a transaction FIFO buffer and read return FIFO buffer. The AHB slave interface receives transactions from AHB2 bus masters and posts it to the transaction FIFO buffer. The EBI interface then receives the transactions from the buffer and decodes them. The EBI interface then drives the data, address and control signals to the specific EBI block device. If the transaction requires read data, the EBI interface sends the read data back through the read return FIFO buffer.

- **AHB2 Interface**
  The AHB2 interface consists of the AHB slave interface and the control and status registers. The AHB slave interface decodes the received transactions and returns the response from the targeted slave connected to an EBI block. The control and status registers are used to monitor and set up the interface according to the device attached to the corresponding EBI block.

- **EBI Interface**
  The EBI interface consists of a timer and EBI transaction sequencer. The timer operates as a binary counter that is used to time asynchronous memory access. The EBI transaction sequencer controls all the external signals to the device attached to a specific EBI block.

Each EBI block is configured at startup by the boot code according to the device connected to it. On the EPXA1 development board two Intel 28F320C3 flash memory devices are connected to block 0 (CS0) and block 1 (CS1) respectively and an Ethernet MAC/PHY device to block 3 (CS3). Block 2 is not used. For more information on the EBI consult [1].

6.4 AMBA AHB Bus System

6.4.1 Overview

The Excalibur device contains two AHB busses and three AHB bridges. These busses and bridges are used as communication medium throughout the device. Each bus has one or more bus masters that initiate a data transfer on the bus.

The processor is the sole bus master on the AHB1 bus. Slaves such as the interrupt controller and on-chip SRAM are local to AHB1, which allows the processor fast access to these devices. AHB1 can operate at up to a speed of 200 MHz. Any transaction that is destined for a peripheral outside the AHB1 domain is routed to the AHB1-2 Bridge. The AHB1-2 Bridge is a slave on AHB1 and is the interface that provides the processor access to AHB2.

On AHB2 there are three bus masters: the AHB1-2 Bridge, Configuration Logic module and PLD-to-stripe Bridge. The AHB1-2 Bridge receives transactions from AHB1 that are destined for a peripheral on AHB2 or the PLD. The Configuration Logic module provides configuration information to the PLD and stripe memory elements. The PLD-to-stripe Bridge allows masters implemented in the PLD access to slaves in the stripe.

6.4.2 AMBA AHB Operation

All transfers on the AHB bus are started when a bus master asserts a request signal to the arbiter. The arbiter determines if the bus master is granted the use of the bus. If the bus is granted, the bus master starts the transfer by driving the address and control signals. These signals provide information about the transfer. The address signal corresponds to the address of the slave. The control signals give information on the type of transfer that is going to take place. A basic transfer consists of two sections:

- an address and control phase, which lasts one clock cycle
- a data phase, which may last several clock cycles

Figure 6.5 shows a typical AHB transaction waveform. The master drives the address

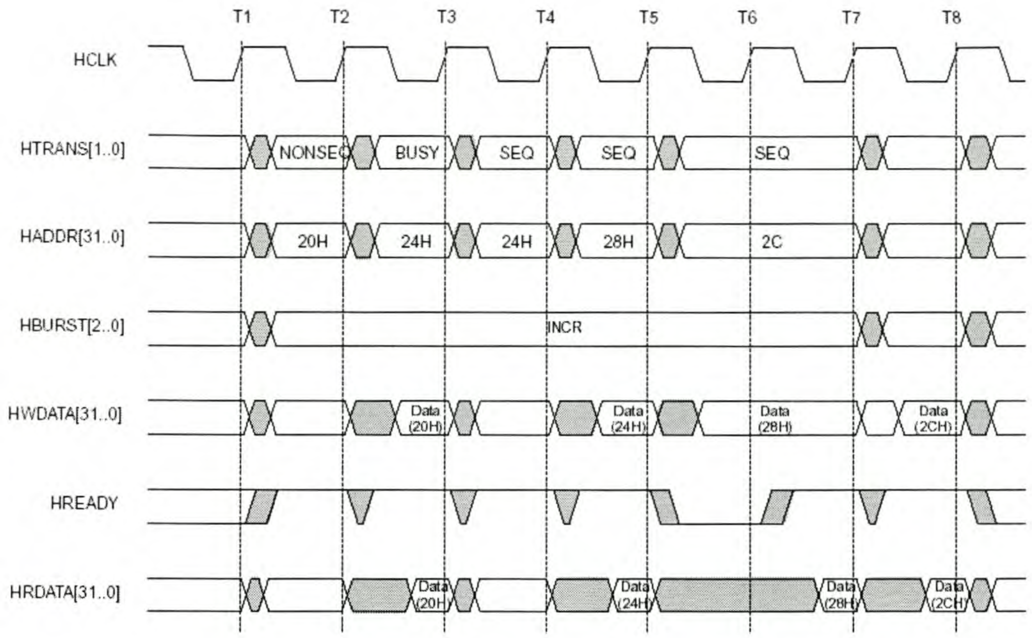


Figure 6.5: AHB transaction waveform [1]

and control signals onto the bus after the rising edge of HCLK. The slave samples the address and control information on the next rising edge of the clock.

If the slave is not ready to receive data, it can pull the HREADY signal to a low state. This causes wait states to be inserted into the transfer and thus allows extra time for the slave. During a transfer the slave replies with a response. The response signal, HRESP, can be OKAY, ERROR, RETRY or SPLIT. This response signal is sampled by the bus master on the third rising edge of the clock.

There are four different transfer types indicated by the HTRANS signal: IDLE, BUSY, NONSEQ (nonsequential) and SEQ (sequential).

Burst information is provided using the HBURST[2..0] signal. The possibilities are:

- incremental bursts of 4, 8, 16 and unspecified length
- wrapping bursts of length 4, 8 and 16
- single transfer

Each transfer will also contain a number of control signals that provide additional information about the transfer. The control signals are sampled at the same time as the address bus and have to remain constant during a burst of transfers. The control signals include the transfer direction, transfer size and a protection control.

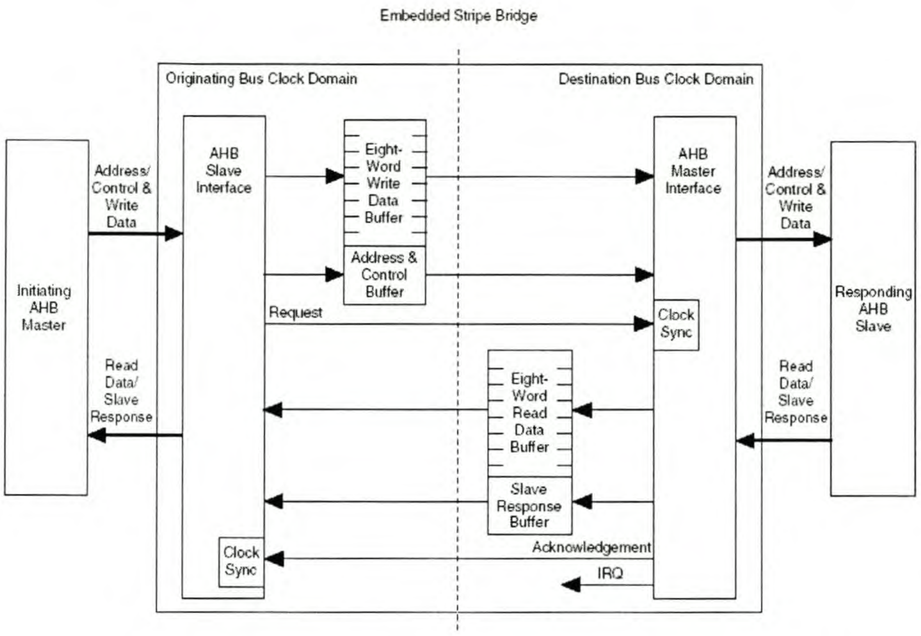

Embedded Stripe Bridge
Originating Bus Clock Domain
Destination Bus Clock Domain
Initiating AHB Master
Address/ Control & Write Data
Read Data/ Slave Response
AHB Slave Interface
Eight-Word Write Data Buffer
Address & Control Buffer
Request
Eight-Word Read Data Buffer
Slave Response Buffer
Clock Sync
Acknowledgement
IRQ
AHB Master Interface
Clock Sync
Address/ Control & Write Data
Read Data/ Slave Response
Responding AHB Slave

Figure 6.6: Functional Block Diagram of AHB Bridge [1]

The data busses HWDATA[31..0] and HRDATA[31..0] are the write and read data to and from the slave respectively.

For more detail about the AMBA AHB bus architecture consult [19].

6.4.3 The AHB Bridges

There are three AHB bridges in the embedded stripe: AHB1-2, PLD-to-stripe and stripe-to-PLD. Two of the bridges, PLD-to-stripe and stripe-to-PLD, provide the interface between the PLD and stripe. PLD masters can only access peripherals in the stripe via the stripe slave port on the PLD-to-stripe bridge while the stripe-to-PLD bridge allows bus masters in the stripe access to slaves implemented in the PLD. Figure 6.6 illustrates the functional block diagram of an AHB bridge. There are four main operating modules in each bridge: AHB slave interface, AHB master interface, write buffer and read buffer. The AHB slave interface receives transactions from a master and sends it to the write buffer. It also receives response information from slaves and sends it back to the initiating master. The AHB master interface synchronizes transactions from the write buffer and regenerates the address and control information for the transaction. It also passes the slave responses to the read buffer. The read and write buffers hold information about the transaction as well as eight words of buffered data.

6.4.4 AHB Bridge Operation

There are two types of write and read transactions for a bridge: posted and non-posted. The type of transaction can be selected by setting the appropriate bit (NP or NW) in the bridge's control register.

- **Non-posted write transaction**
  A master initiates a transaction by transferring the transaction to the bridge's slave interface. The slave interface now tells the initiating master to wait by inserting wait states. While the initiating master waits, the slave interface sends the transaction to the write buffer and informs the master interface that there is a transaction waiting. When the master interface receives the transaction, it requests the appropriate destination bus and synchronizes the transaction to the destination bus's clock domain. Once the master interface is granted the bus, it sends the transaction to the destination slave. The destination slave sends a response back to the master interface which in turn sends it to the read buffer with an acknowledgement that it is ready to process another transaction. The slave interface now reads the response from the read buffer and synchronizes it. It removes the wait stages and sends the response back to the initiating master. The initiating master now knows that the data has reached the destination slave and can issue another transaction. Non-posted writes are relatively slow because of all the delays.

- **Posted-write transactions**
  To minimize the delays of non-posted write transactions, the bridge also supports posted-write transactions. Posted-write transactions allow the initiating master to burst write data to the bridge and then continue to process other transactions before the transaction posted to the bridge reaches its destination.

  The initiating master sends address and control information for the burst to the bridge's slave interface. The slave interface passes it on to the write buffer and informs the master interface. The slave interface accepts data for the burst until the write buffer is full, and then inserts wait states. The slave interface sends a response to the initiating master once the last data for the burst has been received. The initiating master can now start the next transaction. The master interface reads the address and control information, synchronizes the destination bus clock domain and then waits for the bus access. When the bus is granted, the master interface regenerates the address and control information for the transaction and reads the response from the destination slave. If the response is that of an error, the bridge generates an interrupt while the bridge status register preserves information about the transaction that caused the interrupt.

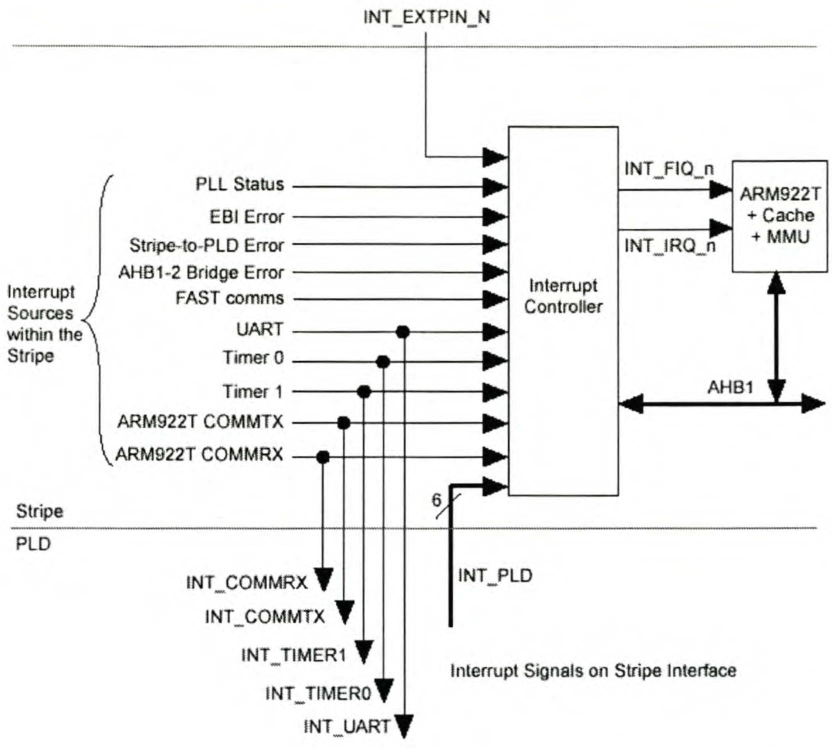


Figure 6.7: The EPXA1 Interrupt Controller in the Stripe [1]

- **Read transactions**

  For both types of read (posted and non-posted), the initiating master requesting the data, must wait for a response from the slave. This makes it similar to the non-posted write transaction waiting for a response from the slave. Non pre-fetched reads can have a significant delay because of arbitration and synchronization. For this reason the bridge also supports pre-fetched read transactions. A pre-fetched read transaction fills the read data buffer in an unspecified length burst. Read pre-fetching therefore increases the performance.

6.5 Interrupt Controller

The interrupt controller is situated in the stripe of the EPXA1 device and can be accessed and modified by a set of registers. Figure 6.7 shows the layout of the interrupt controller. The interrupt controller generates two interrupt signals, INT_FIQ_n and INT_IRQ_n, to the embedded processor from the 17 interrupt sources. There are 10 fixed interrupt sources from the stripe modules, one external source and six sources from the PLD.

6.5.1 Interrupt Controller Interface

The interrupt controller has 24 configuration and status registers that are only accessible from AHB1 by the embedded processor. For each interrupt source there is a priority register which contains the source's priority and whether it generates a normal interrupt (IRQ) or fast interrupt (FIQ). The priority value is used by the interrupt controller to determine which source should be serviced first when several interrupts are active.

6.5.2 Operating Modes

The interrupt controller interprets the 6 interrupt sources from the PLD in one of three modes. The mode is controlled by the INT_MODE register. The three modes are briefly explained.

6.5.2.1 Six Individual Interrupts

The six interrupt sources (INT_PLD[5..0]) are interpreted as six individual interrupts from the PLD. Each of them has its own priority register as well as mask and status bits in the mask and status registers. This is the default mode at system reset.

6.5.2.2 Six-Bit Priority Value

In six-bit priority mode, INT_PLD[5..0] represents a 6-bit priority value. The priority value 0H means that there is no interrupt from the PLD while a non-zero priority (1H to 3FH), contains the priority of the requesting interrupt. The priority registers along with the mask and status bits of the individual PLD interrupt sources have no effect in this mode. Instead, the PLD interrupt priority register, INT_PLD_PRIORITY, holds the requesting PLD interrupt priority value. This mode requires the implementation of an additional interrupt controller in the PLD to facilitate the up to 63 individual interrupts.

6.5.2.3 Five-Bit Priority Value Plus Individual Interrupt

In this mode PLD[5..1] represents the five most significant bits of a 6-bit priority value with the least significant bit always zero. This means that the priority value ranges from 2H to 3EH. The priority value is handled in the same manner as in the six-bit priority mode. However, PLD[0] behaves as an individual interrupt and is treated in the same way as in the six-individual interrupt mode.

6.5.3 Interrupt Mode Implementation

To implement the three operating modes of the interrupt controller, a clear understanding of the eCos interrupt handling methods are necessary.

6.5.3.1 eCos Interrupt Handling

As mentioned in section 4.1.1.4, the ARM processor delivers all exceptions to a set of hardware defined vectors. These vector locations are four bytes apart and start at zero (see Table 4.2). There is only room for one instruction at these vector locations. This means that it must immediately branch to an alternate handling code higher up in memory. These higher locations, from 0x20 up, form the Vector Service Routine (VSR) table. From this table the correct handler, simply known as the VSR, is called. In other words, the actual handling code is reached from the single instruction at the exception entry point via a location 32 bytes higher in memory. Since each VSR is entered in a different processing mode, there has to be a different VSR for each exception that knows how to save the processor's state correctly. For external IRQ sources, the IRQ VSR calls on the default IRQ handler. The IRQ handler determines the vector number of the interrupt from the INT_REQUEST_STATUS register. The vector number corresponds to the bit number of the interrupt in the INT_REQUEST_STATUS register (only when in six individual interrupt mode). From this vector number the attached ISR is called. When installed, the ISR calls the DSR where most of the interrupt handling takes place. This reduces the amount of time and processing spent in the ISR (known as interrupt latency) where the interrupts are disabled. Most of the processing is done in the DSR where the interrupts are enabled. This allows higher priority interrupts to occur while a lower priority interrupt is being served. The flowchart in Figure 6.8 illustrates the eCos interrupt handling of a single hardware interrupt.

After examining the eCos Interrupt Handling method, it was discovered that the three interrupt operating modes were not fully supported by the default IRQ handler. It only supported the six individual interrupt mode. The existing interrupt vector table only contained the vector numbers of 17 interrupts, including the 6 from the PLD. Both the VSR and vector table had to be changed to support all the interrupt modes. None of the PLD interrupts had existing ISRs or DSRs, so this also had to be installed.

6.5.3.2 Six Individual Interrupts Mode

This is the default interrupt handling mode, but can also be selected by writing the value 3H to the INT_MODE register. The six interrupts from the PLD are interpreted as six

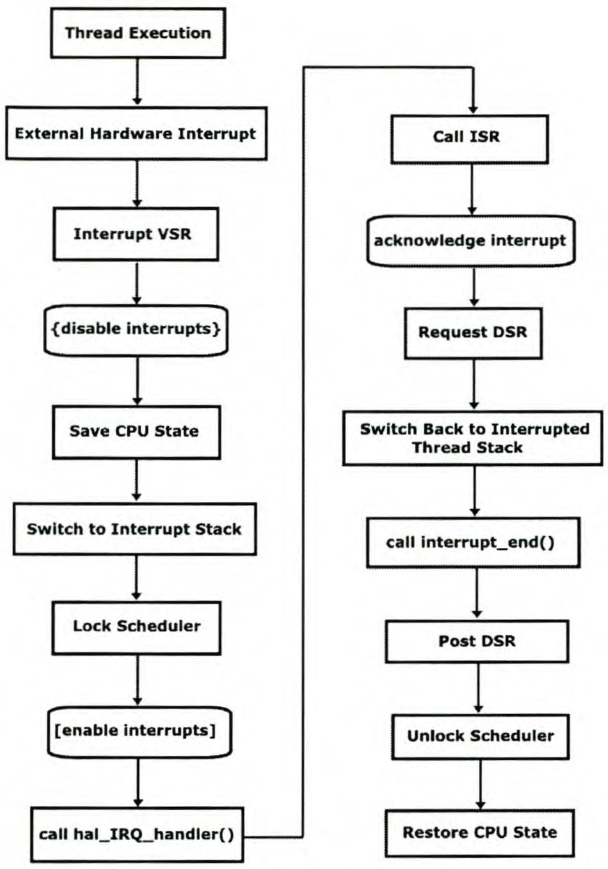


Figure 6.8: EPXA1 eCos Interrupt Handling Flowchart

individual interrupts. Their vector numbers were already present in the vector table. All that had to be added were the ISRs and DSRs for the PLD interrupts. The Quartus block diagram in Figure A.1 includes the implementation of this interrupt mode.

6.5.3.3 Six-Bit Priority Interrupt Mode

The mode is selected by writing 0H to the INT_MODE register and requires the implementation of an additional interrupt controller in the PLD. The six bit priority value (INT_PLD[5..0]) means that there can be up to 63 interrupts from the PLD. A priority encoder was implemented to generate the six-bit priority value from the external interrupt inputs. The priority value from the encoder (INT_PLD[5..0]) is compared with other requesting interrupt priorities by the stripe interrupt controller to produce an IRQ or FIQ request. After this comparison, the interrupt identity register, INT_ID, contains the priority of the highest interrupt priority that is currently pending. The default IRQ handler is called to respond to the IRQ request. The handler then compares the values of the INT_ID and INT_PLD_PRIORITY registers. If the values are the same, it means that the pending interrupt is from the PLD. A function then determines the vector number of the PLD interrupt from its priority value. With the vector number known, the appropriate ISR and DSR can be invoked. If the INT_ID and INT_PLD_PRIORITY values are different, the interrupt was caused by a stripe module and can be handled the same as in the six individual interrupt mode. The Quartus block diagram of the PLD interrupt controller and the eCos source code of the VSR, ISR and DSR are given in Appendix B.

6.5.3.4 Five-bit Priority Plus Individual Interrupt

Writing the value 1H to the INT_MODE register, selects this interrupt mode. The mode is basically a combination of the two modes described above and its implementation is not explained in this text.

6.6 Watchdog Timer

A watchdog timer plays an important role in a satellite's OBC. Due to the radiation in the space environment, program variables in the SRAM can be corrupted (SEU) and cause the software to enter an abnormal state and subsequently crash. When this happens, the watchdog timer resets the system to a known state.

The watchdog timer of the EPXA1 device resides in the stripe and is connected to the AHB1 bus. The watchdog timer can be triggered by the following events:

- counter overflow in hardware trigger mode
- counter overflow in software trigger mode
- an unexpected value written to the reload register

When the watchdog triggers, it asserts the trigger to the reset module. The reset module then resets all the modules in the EPXA1 device (except the trace logic) and provides the reset signal to boot the system from the flash memory connected to the EBI. The three triggers described above operate independently and are described in detail in [1].

6.7 Stripe UART

The stripe contains a UART module that includes modem communications support. This feature typically allows the UART to be used with the modems on a satellite. The UART can be used to communicate with the other subsystems on a satellite as well as debugging the OBC software.

The stripe UART can be used by the embedded processor or masters in the PLD. An additional UART can also be implemented in the PLD logic to be used by other PLD modules or the embedded processor. The EPXA1 development board supports both these UARTs by connecting each to its own transceiver (MAXIM MAX3241E) and DB9 male RS-232 connecter.

The stripe UART was extensively used throughout this study for debugging purposes.

6.8 Reset Module

The EPXA1 device can be reset from various sources. The Reset module in the stripe determines what caused the reset and resets the appropriate modules in the EPXA1 device. If a reset occurs, the PLLs are put in bypass mode, which means that all the clock outputs are generated directly from the input reference clock (CLK_REF). The following are all the different types of reset that can occur and what they comprise of:

- Power-on Reset
  - Resets embedded processor trace port, reset status register (RESET_SR) and embedded JTAG controller
- Warm Reset
  - Resets embedded circuitry, stripe registers and clears PLD contents
  - Asserts EBI external reset signal and nCONFIG to request reconfiguration of PLD
  - Processor is held in reset
- JTAG Reset
  - Resets JTAG controller in PLD
  - Resets JTAG controller in processor core if configured

The reset sequence along with the different reset sources are shown in Figure 6.9. When the reset source is the PLD power-on reset signal or external power-on reset pin (nPOR), the external reset pin (nRESET), internal power-on and warm reset signals are asserted. The reset counter starts counting when the power-on signal is negated. The reset counter

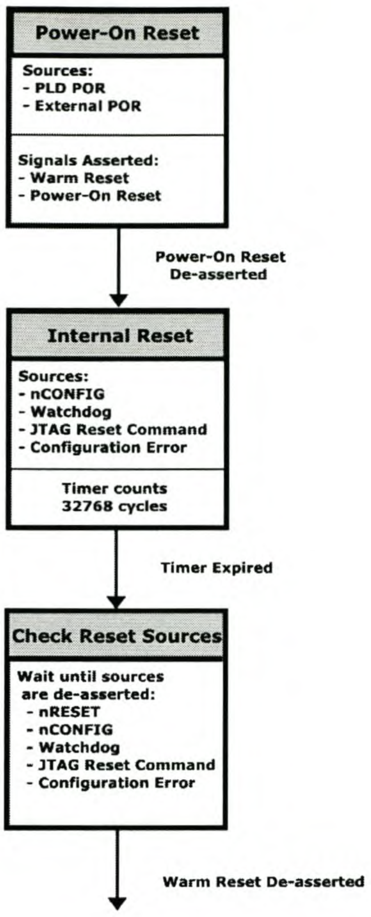


Figure 6.9: EPXA1 Reset Sequence

controls the duration of the reset event and asserts a warm reset. When the counter reaches 32768 clock cycles at the external clock frequency, nRESET is negated, but the warm reset remains asserted until all the reset sources are negated. For more information on the reset module, consult [1].

6.9 General Purpose Timers

The EPXA1 device has a dual-channel timer in the stripe. It can operate in three modes: free-running heartbeat mode, one-shot delay mode and in software interval timer mode. Each timer is configured by its own configuration registers. In all three modes the timer is reset to 0 when the start bit (S-bit in the timer control register) changes from 0 to 1. The period of the timer is a function of the limit value, set in the limit register, and the pre-scaler ratio, set in the pre-scaler ratio register. A short description of each mode is given.

- **Free-Running Heartbeat Mode**
  The timer increments, after reset, until it reaches the limit value in timer limit register. It then resets to 0 and begins incrementing again. An interrupt, if enabled, is requested at the end of each cycle. The timer keeps running while the start bit is set.

- **One-Shot Delay Mode**
  After the timer has been reset, it increments until it reaches the limit value in the limit register. When the limit is reached, the timer stops, the start bit is cleared, and an interrupt is requested (if enabled).

- **Software Interval Timer Mode**
  In this mode the timer keeps incrementing after reset. When it reaches the limit value, an interrupt (if enabled), is requested. However, the timer keeps incrementing until it reaches the value FFFFFFFFH at which point it wraps around to 0 and continues incrementing. The timer is frozen when the start bit S is cleared.

The default eCos kernel uses the first timer (Timer 0) for a software implementation of a Real-Time Clock (RTC) and the second timer (Timer 1) for a delay function. The RTC is very useful in a satellite's OBC for scheduling events.

The configuration values for the RTC can be easily adjusted by the eCos configuration tool. The default configuration for the RTC sets Timer 0 to run in free-running heartbeat mode with a period of 10 ms. The delay function uses Timer 1 in one-shot delay mode. The user can simply invoke the function with the desired delay in microseconds as parameter, from which the limit value is calculated.

These eCos implementation of Timer 0 and Timer 1 can however be disabled in the source code. For more information on the timers consult [1].

6.10 Ethernet Controller

The EPXA1 development board contains an Ethernet controller, the SMSC LAN91C111, connected to the EPXA1's EBI interface (CS3). The SMSC LAN91C111 is a single chip Ethernet controller that provides a dual speed 10/100Mbps connection. The eCos EPXA1 port also provides the device driver for the Ethernet controller.

An Ethernet connection can be very useful during the development phase for uploading big data or application files to memory. The RedBoot monitor can also be configured to

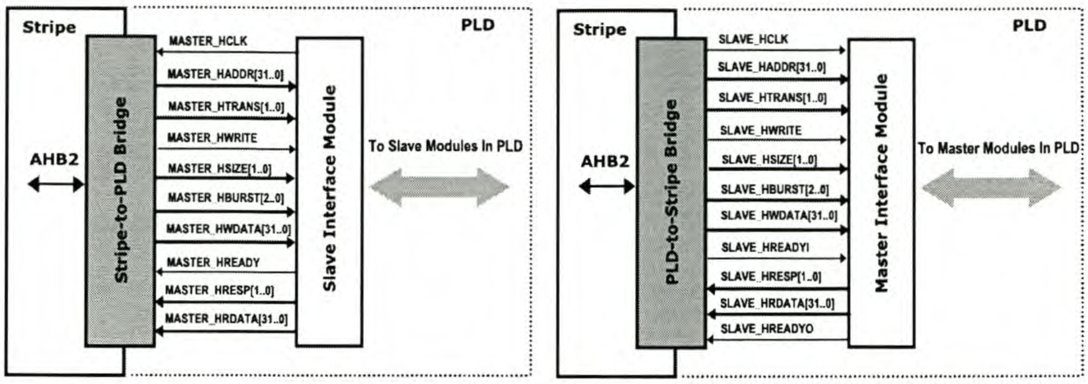


Figure 6.10: AHB Bridges

use either the Serial Port (UART) or Ethernet Channel for communication with the GDB debugger.

6.11 AHB Bus Interface

Two AHB bridges are used to interface the embedded processor stripe with the PLD inside the EPXA1 device. The PLD-to-stripe bridge is used by masters inside the PLD to access slaves inside the stripe. The PLD master accesses the stripe slaves via the slave port of the bridge. The stripe-to-PLD bridge is used by masters in the stripe to access slave modules in the PLD. The PLD slaves are accessed via the master port of the stripe-to-PLD Bridge. The PLD-to-stripe and stripe-to-PLD bridge signals, according to the AMBA AHB specification [19], is shown in Figure 6.10 along with the PLD implementation.

A slave interface module was designed in the PLD that interprets the AMBA AHB master signals from the stripe. Each slave module in the PLD has its own address by which a master can access it. The slave interface module receives the transaction from the master and then sends it to the addressed slave in the PLD. Any read data is collected from the slave and sent back to the master via the bridge.

Similarly, for the masters in the PLD, a master interface module was designed that generates the required AMBA AHB master signals to access slaves in the stripe.

Both the master and slave interface modules are included in the Quartus block diagram in Figure A.1. The master and slave interface ASM charts are given in Figure A.2 and Figure A.3 respectively. The VHDL source code of the two modules is included on the attached CD-ROM.

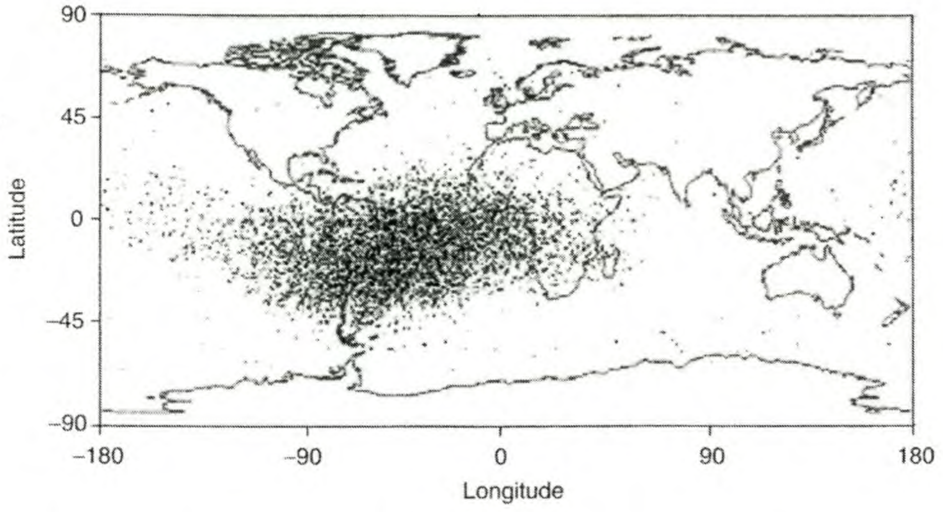


Figure 6.11: SEUs in the S80/T OBC program memory at 1330-km altitude [3]

6.12 EDAC Unit

The program memory is critical in any satellite system. It contains the program code that controls the satellite and therefore needs to be highly reliable, stable and error tolerant. In the case where the program code resides in RAM memory devices, special precautions have to be made to protect this memory against SEU induced errors. For LEO satellites, passing through the South Atlantic Anomaly (SAA) proton belt can cause SEU errors in RAM devices. Figure 6.11 proves this occurrence of SEUs as the satellite passes through the SAA proton belt.

These SEUs can however be detected and corrected by an EDAC unit. The EDAC unit can typically be implemented in three different ways:

- software based
- hardware based (Commercial EDAC Integrated Circuit)
- hardware based (PLD Logic)

To implement a software-based EDAC requires a lot of extra processing which is a very limited resource on a satellite. The system software also has to be designed to support an EDAC of this nature. The use of a commercially EDAC circuit was ruled out due to the existing EPXA1 development board architecture. It will be hard to incorporate an EDAC Integrated Circuit (IC) with the existing components on the board. It will also contradict the idea of an OBC on a single chip.

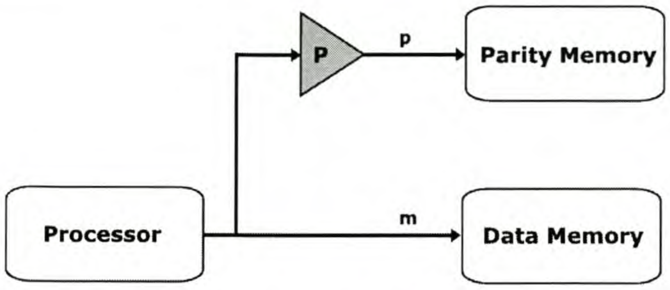


Figure 6.12: Encoder Schematic

An PLD based EDAC unit proved to be the best solution. It could be implemented in the PLD section of the EPXA1 device.

6.12.1 Choosing an EDAC Data Coding Scheme

A wide variety of data coding schemes are currently available to base an EDAC unit on, with the Hamming code being the most well known. The Hamming code allows the correction of one error bit per stored data word. Although Hamming code based designs are simple and generally considered to be acceptable for the use in an OBC [6], recent studies done by Surrey Satellite Technology Limited (SSTL) showed that, although rare, double bit errors can occur in modern high-density memory (4 Mbit die) [5]. This means that there is a definite risk of two error bits occurring within one byte of stored data, either from the impact of a particularly energetic SEU, or from a second SEU creating a second error before the computer had time to correct the first error [20]. With this in mind, a 2-bit correcting code was needed to base the EDAC system on.

For a coding scheme to correct up to two bit errors, the minimum distance between two pairs of code words must be five ($d_{min} = 5$). A coding scheme that meets this requirement was developed at the University of Surrey and is based on quasi-cyclic codes.

6.12.2 The Quasi-Cyclic (16,8) Code

The theory behind the quasi-cyclic code is explained in detail in [20]. The basic encoding and decoding is explained in the following sections.

- **Encoding**
  To encode an 8-bit data vector $\boldsymbol{m}$, the encoder generates an 8-bit parity vector $\boldsymbol{p}$ from the data vector using a parity matrix $\boldsymbol{P}$. The 16-bit code word, $[\boldsymbol{p}\ \boldsymbol{m}]$, is stored in RAM. Figure 6.12 shows the Encoder Schematic.

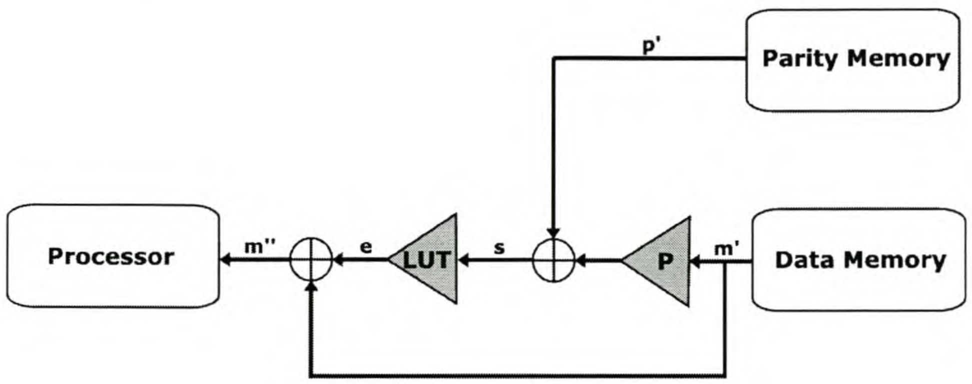


Figure 6.13: Decoder Schematic

- **Decoding**
  For decoding, the entire 16-bit code word $[\boldsymbol{p}'\ \boldsymbol{m}']$ is read. A syndrome vector $\boldsymbol{s}$ is then derived by re-encoding $\boldsymbol{m}'$ and exclusive-OR'ing it with $\boldsymbol{p}'$. Comparing this syndrome vector $\boldsymbol{s}$ with the entries in a look-up table (LUT) will yield an error vector $\boldsymbol{e}$. If there are no errors, both vectors $\boldsymbol{s}$ and $\boldsymbol{e}$ will be zero. If an error did occur and it is correctable, the corrected data vector $\boldsymbol{m}''$ is derived by exclusive-OR the error vector $\boldsymbol{e}$ with the original data vector $\boldsymbol{m}'$. Figure 6.13 shows the complete decoder schematic.

6.12.3 Physical EDAC Implementation

Implementing an EDAC unit in the PLD of the EPXA1 device is possible due to the fact that the PLD can access all of the external memory connected to the EBI and SDRAM controller through the AHB bus. However, this is not an ideal setup. No flow-through EDAC, that resides between the processor and memory, can be implemented with the EPXA1 architecture. The advantage of such a flow-through EDAC structure is the simple control logic.

6.12.3.1 EDAC VHDL Design

The (16,8) Quasi-Cyclic coding scheme can be easily implemented in VHDL. The implementation logic is based on an open source VHDL module provided by the European Space Agency (ESA).

- **Write Cycle**
  During a write cycle the processor sends the memory address and data along with a write flag to the PLD instead of writing the data directly to memory. The logic in

the PLD then generates the parity bits from the received data and stores the coded word at the received memory address.

- **Read Cycle**
  During the read cycle the processor sends the memory address of the requested data to the PLD along with the read flag set. The PLD then request the data from the memory mapped at the received address. The coded word is read and the parity is generated from the stored data received. The generated parity is compared against the stored parity. If the resultant syndrome is not equal to zero, then the LUT is used to match the syndrome to an error location pattern identifying the error bit(s). The data is then corrected and stored again before passing it to the processor.

The EDAC implemented can be used on any memory attached to the EBI or SDRAM controller. Ideally, the EDAC should be tested with external SRAM connected to the EBI. Unfortunately, there is no external SRAM connected to the EPXA1's EBI on the development board. However, testing the EDAC itself is not dependant on the type of memory it is used on. Therefore, the external SDRAM connected to the EPXA1 device could be used to test the EDAC unit. The complete EDAC VHDL source code is included on the attached CD-ROM.

6.13 PLD UART

A basic UART was implemented in the PLD logic. The UART has no modem control signals and can only transmit data. The UART can be used by the processor or by other PLD implemented modules. The 16 Kbytes of on-chip DPSRAM was used as a transmit buffer for the UART. Both the processor and PLD can directly access the DPSRAM without the use of the AHB bridges.

The PLD interface to the DPSRAM is very flexible in the sense that it can be configured to operate in different interfacing modes. Table 6.1 shows the different PLD interfacing modes to the single block of DPSRAM. The embedded processor bus (AHB1) always accesses the DPSRAM as 32-bit data because it is a 32-bit wide bus. For this reason the 32-bit PLD interfacing mode was used in the design. The PLD UART module can be seen on the Quartus block diagram in Figure A.1 while its ASM chart is shown in Figure A.4.

Table 6.1
DPSRAM Interface Modes and Widths

Interface Width	Interface Mode		
	One AHB-PLD	Two AHB-PLD	One PLD-PLD
32-bit	1 x 4 K x 32	-	-
16-bit	1 x 8 K x 16	2 x 4 K x 16	8 K x 16
8-bit	1 x 16 K x 8	2 x 8 K x 8	-

6.14 I^{2}C Controller

The Interconnected Integrated Circuit (I^{2}C) serial bus developed by Philips has the features of low current consumption, high noise immunity and a wide operating temperature range. These features make it ideal for the use in a satellite system. This protocol is easy to implement and is supported by many devices today. I^{2}C was used as the internal bus system on the AAU (CubeSat concept) satellite developed by the University of Aalborg. It is also used on board the University of Stellenbosch's SUNSAT 2004 satellite.

6.14.1 I^{2}C Overview

The I^{2}C protocol specifies a simple bi-directional 2-wire, serial data (SDA) and serial clock (SCL) bus for IC communication. Each device connected to the bus is recognized by its unique address and can operate in either master or slave mode. Currently, there are three modes of operation: standard mode (up to 100 kbits/s), fast mode (up to 400 kbits/s) and high-speed mode (up to 3.4 Mbits/s). Originally the address space consisted of 7-bits, but was increased to 10-bits to allow more devices on the bus (up to 1024 devices). However, both these address formats can be used on the same bus. For more information on the current I^{2}C serial bus specification (version 2.1), consult [21].

6.14.2 I^{2}C Implementation

An I^{2}C driver was implemented in the PLD logic of the EPXA1 device. Two I/O pins (AB9 and T10) were used to connect to the I^{2}C bus. The driver supported the standard and fast I^{2}C speed modes, but only in master mode. This means that the EPXA1 cannot be addressed by the other peripherals on the bus. The processor can also access the I^{2}C bus through a PLD based register. The I^{2}C driver is shown on the Quartus block diagram in Figure A.1 and its VHDL source code is included on the attached CD-ROM.

Chapter 7

Tests and Measurements

Many test programs were written to test the implementations and performance of the EPXA1 device. The results of these tests are discussed in the following sections.

7.1 Power Consumption

It is very difficult to measure the total power consumption of the EPXA1 device due to its FineLine ball-grid array (BGA) package and the fact that there is no provision made for this kind of measurement on the development board. There are, however, theoretical methods of calculating the power consumption. The total power consumption of the EPXA1 device can be divided between the power consumed by the PLD and embedded stripe.

7.1.1 PLD Power Consumption

The power consumption of the PLD is very dependant on the amount of implementation logic that is used. Without being able to measure the power consumption of the PLD, the only other known way was to use a power calculator that Altera recommends [4]. Unfortunately, the power calculator did not support the PLD of the EPXA1 device.

7.1.2 Embedded Stripe Power Consumption

The power consumption of the embedded stripe can be roughly calculated by using the power consumption graphs given in [4]. These graphs were drawn under the following test environment:

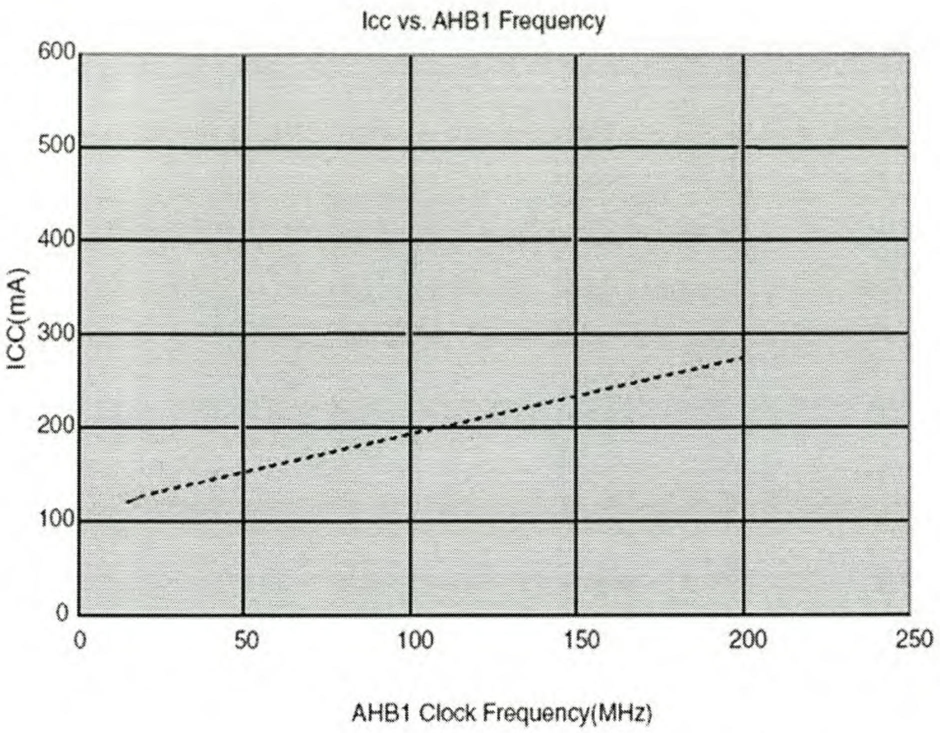


Figure 7.1: Current Consumed by the Embedded Stripe versus AHB1 Clock Frequency [4]

- low temperature, high V$_{cc}$ and varying clock frequency
- embedded processor makes continual memory accesses
- timer, UART and EBI are enabled
- PLLs are locked
- PLD DPSRAM interface makes a memory access on every clock edge
- PLD master writes to PLD slave via AHB2 bus continually

Figure 7.1 shows the current consumed by the embedded stripe when the PLD interfaces are disabled. Figure 7.2 and Figure 7.3 shows the current consumed by the embedded stripe when the PLD interfaces are enabled.

From the three graphs the total current consumption of the embedded stripe can be calculated:

$$I_{cc(total)} = I_{cc(figure.7.1)} + I_{cc(figure.7.2)} + I_{cc(figure.7.3)} \tag{7.1.1}$$

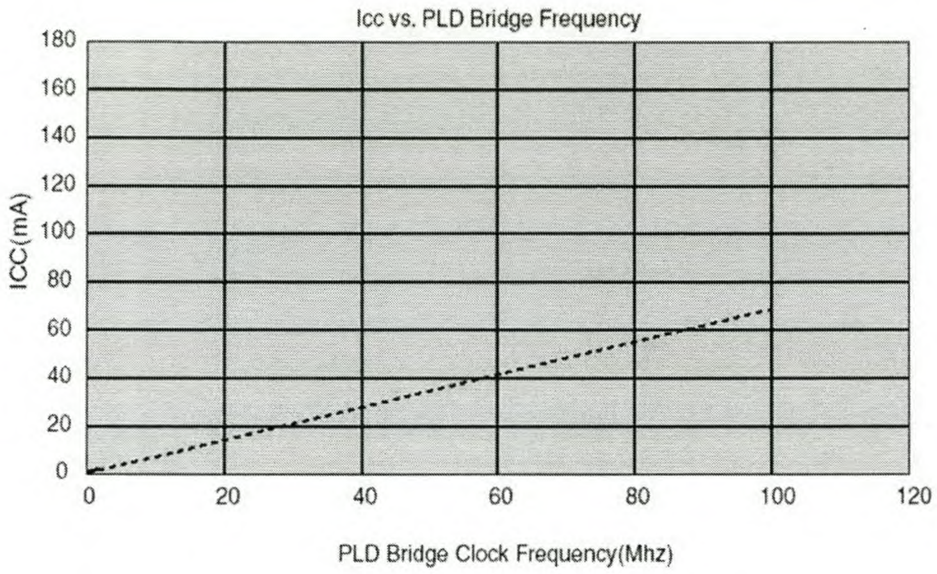


Figure 7.2: Current Consumed Versus Clock Frequency Due to the Embedded Stripe Bridge Interface [4]

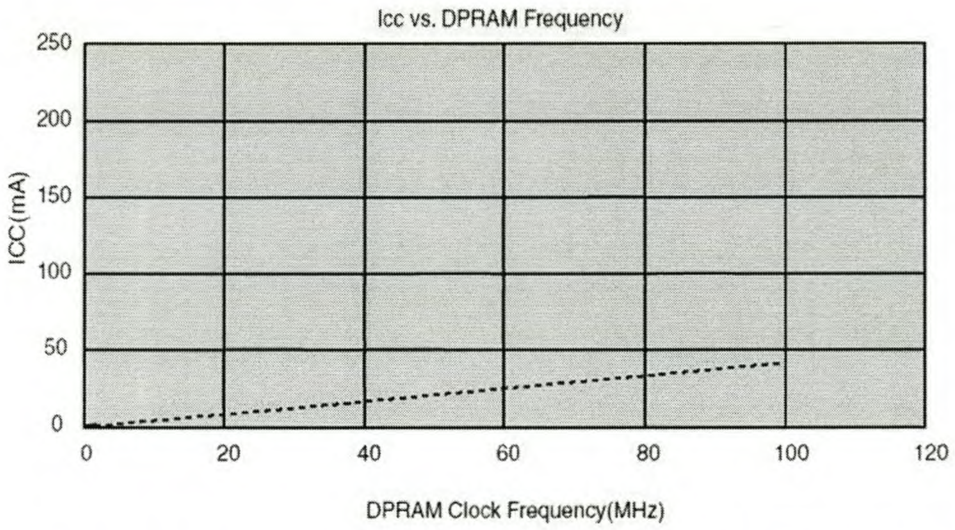


Figure 7.3: Current Consumed Versus Clock Frequency Due to the DPSRAM Interface [4]

Therefore, from (7.1.1), the maximum current consumption of the embedded stripe is:

$$\begin{aligned} I_{cc(max)} &= 275mA + 70mA + 43mA && (7.1.2)\\ &= 388mA && (7.1.3) \end{aligned}$$

With a supply voltage of 1.8V, the maximum power consumption of the embedded stripe can be calculated from Ohm's law:

$$\begin{aligned} P &= VI && (7.1.4)\\ &= 1.8V \times 388mA && (7.1.5)\\ &= 0.6948W && (7.1.6) \end{aligned}$$

7.1.3 Total Power Consumption

The total power consumption of the EPXA1 device is the sum of the PLD power consumption and embedded stripe power consumption.

With the PLD power consumption not known, the total power consumption could not be calculated. According to [22], the maximum current consumption of the EPXA1 device core is 1.1 A. Using equation 7.1.4, gives a maximum power consumption of 1.98 W for a 1.8 V supply voltage. This maximum power estimation is roughly the same as the 2 W estimated by [23].

The total power consumption of the development board was measured under the following circumstances:

- processor running at 150 MHz and PLLs enabled
- continual memory read and write cycle with EDAC enabled
- continual memory read and write cycle with EDAC diasabled
- continual PLD UART write transaction using the DPSRAM as interface between the processor and PLD
- continual I^2C write and read transaction
- continual stripe UART write transaction
- user LEDs used continually
- Ethernet controller disabled

The development board drew 250 mA in this operating state which gave a power consumption of 3 W. In an idle state, the board drew 220 mA (2.64 W).

7.1.4 Low-Power Mode

The EPXA1 device supports a low-power mode. In low-power mode the PLLs can be turned off. This results in a current consumption of 5 mA by the disabled PLL. Low-power mode can be used to place the embedded processor in a suspended mode. To place the EPXA1 in suspended mode, the LP-bit in the PLL control register is set. A PLD pin must then be used to signal to external logic to slow down or stop the clock generator. The suspended mode was not tested because the reference clock of the EPXA1 is fixed and therefore could not be slowed down or stopped.

7.2 Floating Point Test

The ARM922T processor core provides a coprocessor interface that allows the attachment of a floating point unit on the same chip, but unfortunately the EPXA1 device does not have such a floating point unit. The presence of a floating point unit in the OBC is crucial when the OBC has to do floating point calculations, i.e. handling tasks for the Attitude Determination and Control System (ADCS). Two methods of implementing a floating point unit (FPU) on the EPXA1 device were identified:

- PLD based FPU
- Software based FPU

A PLD based FPU is typically in the form of various look-up tables, while a software based FPU is a combination of a math library and built-in compiler functions. eCos already contains a math library that supports various standard mathematical functions that made this the obvious choice.

To test the FPU, an eCos application was written that evaluates a 10th order international geomagnetic reference field (IGRF) model designed by Professor W.H. Steyn (4/5/98). The IGRF model is used in satellite attitude determination. The model was evaluated at different processor frequencies, and with the cache enabled and disabled. The processor frequency can be changed by adjusting the PLL1 output clock frequency (see section 6.2). The source code of this test is included on the attached CD-ROM.

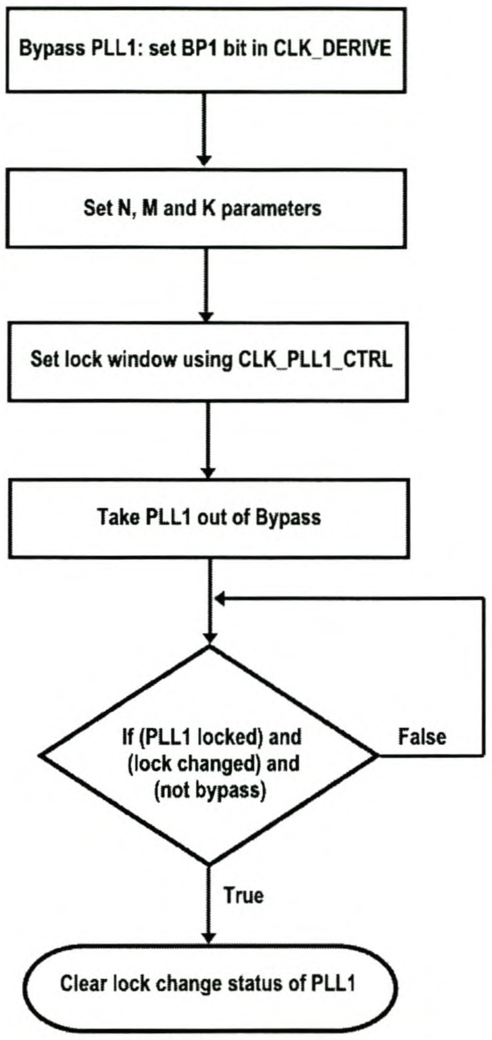

Bypass PLL1: set BP1 bit in CLK_DERIVE
Set N, M and K parameters
Set lock window using CLK_PLL1_CTRL
Take PLL1 out of Bypass
If (PLL1 locked) and (lock changed) and (not bypass)
False
True
Clear lock change status of PLL1

Figure 7.4: Flowchart for Reprogramming PLL1

7.2.1 Adjusting the PLL1 Output Clock

Each PLL is controlled by a set of registers. By adjusting the CLK_PLL1_NCNT, CLK_PLL1_MCNT, CLK_PLL1_KCNT registers of PLL1, the desired output clock frequency can be obtained. This three registers represent the N, M and K values that sets the multiplication and division factors for PLL1. The values of N, M and K are obtained by using an encoding scheme [24]. The flowchart in Figure 7.4 describes the sequence that was used to reprogram PLL1. The desired processor frequency can be obtained by setting the PLL1 output clock frequency to twice the processor frequency.

Table 7.1
IGRF TEST RESULTS (100 LOOPS)

Processor Frequency (MHz)	Cache Enabled		Cache Disabled	
	EPXA1	S3C4510	EPXA1	S3C4510
25	5.636s	6.655s	97.169s	17.294s
50	2.830s	3.328s	61.943	8.647s
100	1.426s	-	44.530s	-
150	0.958s	-	37.987s	-
200	0.724s	-	35.147s	-

7.2.2 Controlling the Use of the Cache Memory

eCos provides macros for enabling and disabling the data and instruction cache of the ARM922T processor. The following macros were used to control the data and instruction cache:

- HAL_DCACHE_IS_ENABLE(dcache) , HAL_ICACHE_IS_ENABLED(icache)

  - returns the status of the data/instruction cache in dcache/icache

- HAL_DCACHE_ENABLE(), HAL_ICACHE_ENABLE()

  - enables the data and instruction cache respectively

- HAL_DCACHE_DISABLE(), HAL_ICACHE_DISABLE()

  - disable the data and instruction cache respectively

7.2.3 Floating Point Test Results

The time it takes to calculate the 10th order model at various frequencies was recorded with the EPXA1 timer. The same test, also using the eCos math library, was conducted by Mnr. F. Retief on the Samsung S3C4510 (ARM7TDMI core) processor. The results of the two tests are given in Table 7.1.

From the results it is clear that disabling the cache of the EPXA1 and S3C4510 processors decreases the performance. Comparing the two processors it also shows that the EPXA1's performance decreases substantially more than that of the S3C4510 processor when the cache is disabled.

7.3 Interrupt Controller

The three interrupt modes as described in section 6.5 were implemented and successfully tested. In all three modes the interrupts from the PLD were generated by the user switches SW2 to SW5 on the EPXA1 development board. A PLD interrupt controller had to be implemented for the priority interrupt modes. DSRs were used for all the interrupts that outputs a text message to the terminal window describing the interrupt. The Quartus block diagram of the PLD interrupt controller and the eCos source code of the VSR, ISR and DSR are given in Appendix B.

7.4 Watchdog Timer

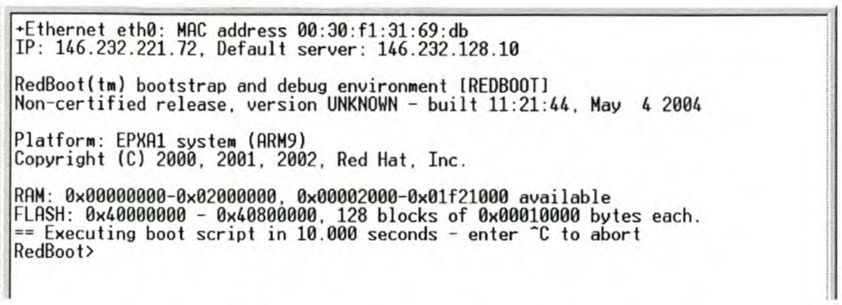
The watchdog timer can be triggered by the hardware or software.

Asserting a low value on the DEBUG_EN pin enables the hardware watchdog. The hardware watchdog then triggers when COUNT in the WDOG_COUNT register overflows. Unfortunately, the DEBUG_EN pin on the EPXA1 development board is connected to 3.3V through a 4,7kΩ register. The hardware watchdog could therefore not be tested.

The software trigger is enabled by writing a non-zero TRIGGER value in the WDOG_CR register. When COUNT equals the TRIGGER value, the watchdog is triggered. The watchdog is also triggered if an invalid value is written to the WDOG_RELOAD register. There are only two valid values: A5A5A5A5H or 5A5A5A5AH. These last two methods of triggering the watchdog was successfully tested with eCos applications. The source code of these tests is provided on the attached CD-ROM.

7.5 General Purpose Timers

The three timer modes as described in section 6.9 (free-running heartbeat, one-shot delay and software interval timer) were successfully configured and tested with an eCos application. The source code is available on the attached CD-ROM. Timer 1 was used in all three tests which means that the eCos delay function had to be disabled in the eCos source code.



```
+Ethernet eth0: MAC address 00:30:f1:31:69:db
IP: 146.232.221.72, Default server: 146.232.128.10

RedBoot(tm) bootstrap and debug environment [REDBOOT]
Non-certified release, version UNKNOWN - built 11:21:44, May  4 2004

Platform: EPXA1 system (ARM9)
Copyright (C) 2000, 2001, 2002, Red Hat, Inc.

RAM: 0x00000000-0x02000000, 0x00002000-0x01f21000 available
FLASH: 0x40000000 - 0x40800000, 128 blocks of 0x00010000 bytes each.
== Executing boot script in 10.000 seconds - enter ^C to abort
RedBoot>
```

Figure 7.5: RedBoot with network package

7.6 Ethernet

The LAN91C111 Ethernet driver is used by the eCos Networking package for the EPXA1 development board. To test this driver, RedBoot was compiled with the eCos Network package included. The network package was configured to use the FreeBSD networking stack. The FreeBSD implementation supports various networking protocols like IPv6, ARP, UDP, TCP, DHCP and TFTP. When the DHCP protocol is enabled, only the valid MAC address of the system is required to get the appropriate IP address from the DHCP server. Figure 7.5 shows the RedBoot startup with the network package included.

RedBoot provides some commands to use in a network environment, like sending ping packets and IP address queries. The network connection was mostly used with RedBoot to upload bigger *.srec* files to the board via TFTP. The RedBoot configuration, which includes the network package, is provided on the attached CD-ROM.

7.7 PLD UART

The PLD UART, described in section 6.13, was implemented successfully. An eCos application writes the data it wants to send to the DPSRAM. The PLD UART reads this data and sends it to the MAX3241E transceiver at a buadrate of 57 600. The eCos application source code is included on the CD-ROM while the ASM chart of the PLD UART is shown in Figure A.4.

7.8 EDAC Performance

The PLD implemented EDAC unit, as described in section 6.12, was tested on the external SDRAM connected to the EPXA1 device. Figure 7.6 illustrates the difference between

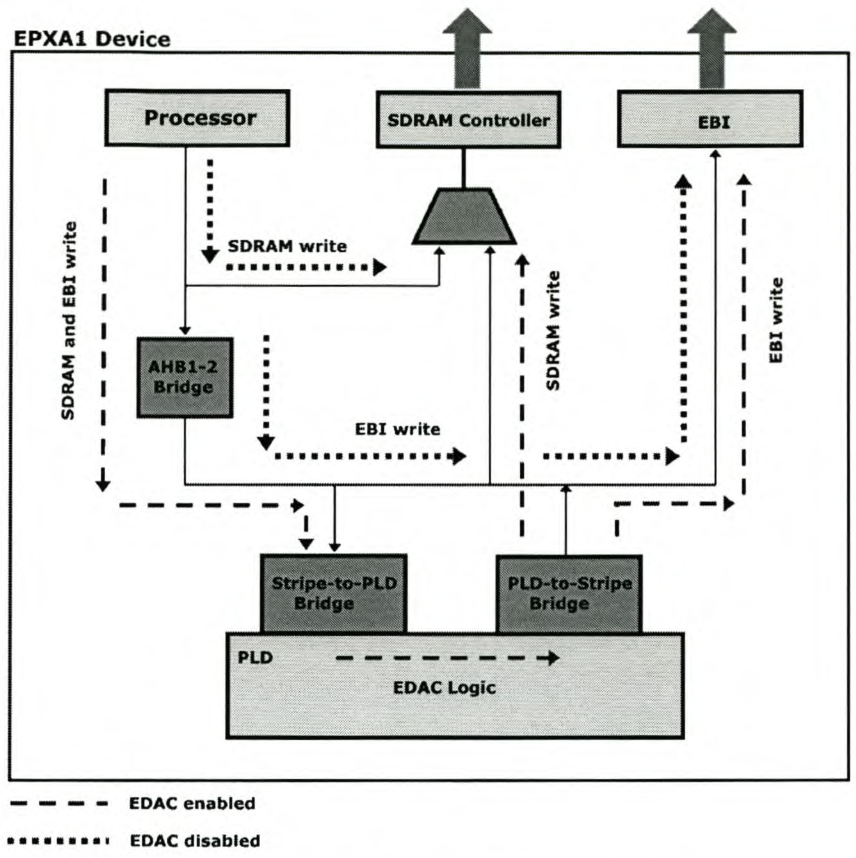


Figure 7.6: Memory Write Cycle

a memory write cycle when the EDAC is enabled and when it is disabled. From this illustration it is clear that the EDAC will add some delays in memory accesses.

7.8.1 EDAC Accuracy

First of all, the EDAC unit was tested for general accuracy. The test will indicate that the EDAC can detect and correct up to two bit errors in an 8-bit data word. The test was implemented with an eCos application (source code available on CD-ROM) that executed the following sequence of instructions:

1. The application writes data to a specific SDRAM address with the EDAC enabled. The data is stored in memory along with the parity bits.
2. The stored data is read back from the same SDRAM address with the EDAC disabled. Two data or parity bits are changed to simulate bit-flip errors.
3. This incorrect data and parity are written back to memory at the same address, still with the EDAC disabled.

Table 7.2
MEMORY CYCLE TIMES

Memory Cycle	Processor (μs)	EDAC (μs)
Write Cycle	~598	~619.23
Read Cycle	~6.41	~22.40
Extended Read Cycle	-	~43.00

4. The EDAC is enabled and the same, incorrect data is requested from memory. If the correct data is received, it means that the EDAC detected and corrected the bit-flip errors.

5. Another memory read to the same address, with the EDAC disabled, should also yield the correct data and parity. This indicates that the EDAC, upon reading the incorrect data or parity, corrected it and wrote the data back to memory (extended read cycle).

The test proved that the EDAC can detect up to two bit errors in an 8-bit word and correct them. There was however some complications. When the processor accesses the memory data with the EDAC disabled just after an EDAC transaction, it often reads back the "old" data. The reason for this is that when the EDAC has finished encoding the data, the data still needs to be sent to the SDRAM controller and eventually to SDRAM. This process is relatively slow in comparison with the processor's access time to the SDRAM without the EDAC. The processor accesses the SDRAM controller at the speed of the AHB1 bus while the EDAC in the PLD must access the SDRAM controller via the PLD-to-stripe bridge and AHB2 bus that runs at half the speed of AHB1. This again shows the time delays the EDAC unit imposes on the memory access. This problem will, however, not occur when the memory is exclusively accessed with the EDAC enabled.

7.8.2 EDAC Memory Cycle Times

The memory access of a system has a very significant effect on its overall performance. For this reason it was necessary to measure the time added by the EDAC unit to a memory cycle. The time added by the EDAC unit for a write, read and extended read cycle were measured with the EPXA1 timer. The results are shown in Table 7.2. These measurements are only rough because the times will vary depending on the amount of transactions that the AHB busses have to handle at the same time.

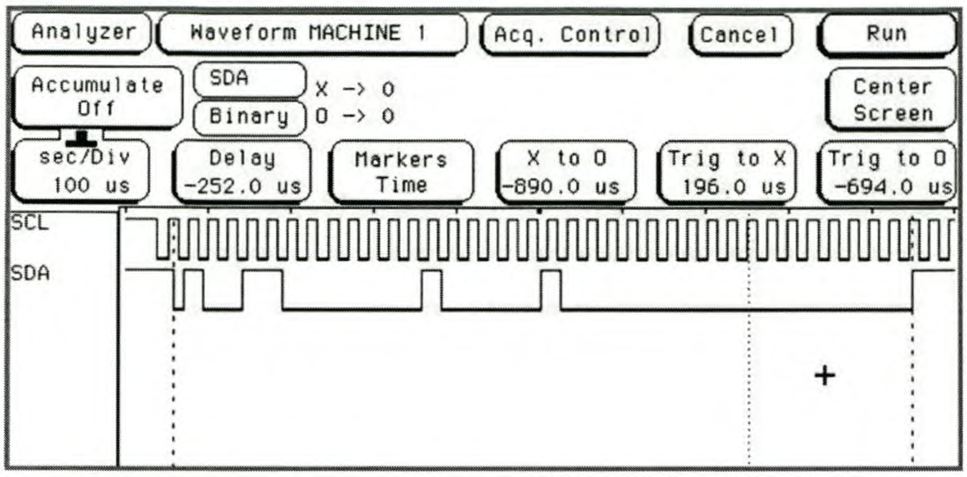


Figure 7.7: I^{2}C Write Transaction in Standard Mode

7.9 The I^{2}C Bus

A slave device had to be connected to the I^{2}C bus to test the I^{2}C driver (see section 6.14). A digital-to-analog (DAC) converter with an I^{2}C interface was chosen as slave device because the DAC's output voltage could be set via the I^{2}C bus and then measured and compared with the programmed value.

A Texas Instruments DAC8571 was chosen to operate as a slave device on the I^{2}C bus for testing purposes because of previous experience with the device. The DAC8571 is a small low-power, 16-bit voltage output DAC. The device requires very little support circuitry which makes it easy to implement.

The slave device was configured and tested successfully in both standard and fast modes, via an eCos application, using the I^{2}C driver in the PLD. The results of a write and read transaction, as measured by a logic analyzer, are shown in Figure 7.7 and Figure 7.8 for the standard mode. The logic analyzer measurements for the fast mode are given in Appendix C.

For the write transaction in Figure 7.7 there are four bytes sent by the master to the slave. They are descibed as follows:

- Address Byte
  - The unique address of the slave, $(10011000)_2$, where the least significant bit indicating that it is a write transaction.
- Control Byte

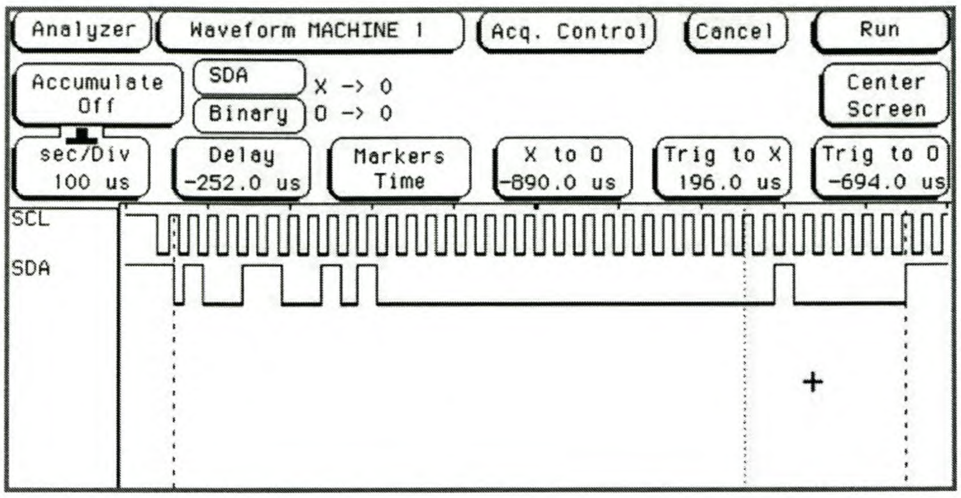


Figure 7.8: I^{2}C Read Transaction in Standard Mode

- The byte tells the slave what to do with the data once it receives it. In this case the value $(00010000)_2$ tells the DAC to write the data to its temporary register and load the DAC with the data.

- Data Bytes
  - Following the address and control bytes are the most significant and least significant data bytes.

For the read transaction the master sends the slave address with the least significant bit indicating a read transaction. The slave then sends the most significant byte, least significant byte and control byte, in that order.

Chapter 8

Conclusions and Recommendations

8.1 Conclusions

Altera's Excalibur EPXA1 embedded processor was evaluated for the implementation of a satellite's OBC. The EPXA1 device contains an ARM922T processor core, embedded peripherals and programmable logic which, in theory, allows the implementation of an entire OBC on a single chip. This possibility was investigated and the following conclusions were made.

8.1.1 Reliability

Two major reliability concerns were identified for using the EPXA1 device in a space environment. These are the small process size (0.18 μm) of the device and the SRAM configuration elements of the PLD. Both these elements make the EPXA1 device very susceptible to SEUs.

Unfortunately, no radiation test results are available for the EPXA1 device to give a more accurate result. There is also no known history where the EPXA1 device was used in a space application.

8.1.2 Power Consumption

The total power consumption of the EPXA1 could not be measured because the development board does not make provision for such a measurement. Instead, the power consumption was calculated. The total power consumption is dependant on the stripe and the PLD logic. The stripe's maximum power consumption was calculated as 694.8 mW.

The PLD's maximum power consumption could not be calculated. However, comparing [22] and [23], the maximum power consumption of the EPXA1 device was estimated as 2 W. The maximum power consumption of the EPXA1's ARM922T core, with the cache enabled, is 160 mW. The power consumption of the development board was measured as 3 W under a certain operating mode.

Although a power consumption of 2 W is more than double the power consumption of previous processors evaluated ([8],[9]), it should be noted that this value represents the entire OBC's power consumption (excluding external memory) and not just the processor.

8.1.3 Performance

The ARM922T processor core of the EPXA1 device has a processing performance of 210 MIPS at 200MHz. This is less than the 360 MIPS of the Hitachi SH7750 processor used on the SUNSAT 2004 satellite.

An IGRF test was conducted on the EPXA1 device to test its performance decrease when the data and instruction cache memories are disabled. The results were compared to that of a Samsung S3C4510 processor with an ARM7TDMI core. Both processors showed a decrease in performance when the cache memories are disabled. The EPXA1's performance was better than that of the S3C4510, but only when the cache memories were enabled. With the cache memories disabled, the EPXA1's performance decreased substantially more than that of the S3C4510 processor and was up to 7 times slower.

8.1.4 Architecture

The ARM922T is a 32-bit RISC processor core with MMU and data- and instruction caches. It contains a 32-bit internal AMBA AHB bus system. The memory interfaces, SDRAM controller and EBI, supports external memory devices with a data width of up to 16-bits. This data width will limit the data transfer rates and performance of the device comparing to wider data widths like 32-bits and 64-bits.

The EPXA1 device contains very useful on-chip peripherals like the memory and interrupt controllers, timers and debug support. The PLD allows the implementation of additional soft-core peripherals which might be needed in a specific OBC. The 186 general purpose I/O pins and up to 63 interrupt lines from the PLD, provides support for a wide variety of external devices attached to the EPXA1 device. However, there are no dedicated general purpose I/O pins for the processor. This means that if a software driver is used for an external device, it will have to communicate via the PLD. This will add some time delays to the interface and complexity to the design.

8.1.5 Development Tools

The Altera Quartus II development tools provide complete support for configuring and programming the EPXA1 device. The eCos RTOS was chosen as most suitable operating system to be used on the EPXA1 device. eCos already has a complete port to the EPXA1 device and with its real-time characteristics and high level of configurability makes it an ideal operating system for a satellite's OBC.

8.1.6 EPXA1 as Single-Chip OBC

It is unlikely that the EPXA1 will ever be used as a single-chip OBC. The radiation concerns discussed along with the poor performance when the cache memories are disabled are the main downfalls of the device. The EPXA1 device is also very expensive compared to similar processors. The total cost of the EPXA1 device is more than double than the processor used on the SUNSAT 2004 satellite.

8.2 Recommendations

From the experiences gathered during this study, a few aspects were identified that still need some research or could be improved. These aspects are highlighted in the following sections.

8.2.1 Radiation Tests

The reliability of the EPXA1 device is suspect due to its small processing size and SRAM configuration elements. These factors make it susceptible to SEUs induced by space radiation. However, these are only predictions based on previous findings ([6], [9]).

Radiation tests should be conducted that simulate the space environment that the OBC will experience. This will give a far more accurate result of the EPXA1's reliability.

8.2.2 Peripheral Interfaces

The implementation of an LVDS (Low Voltage Differential Signaling) communication channel and CAN (Controller Area Network) bus should be considered.

LVDS provides a means of sending data along a twisted pair cable at high speed, with low power and with excellent EMC performance [9]. These features make LVDS ideal for satellite on-board data handling applications. The Quartus II software provides LVDS transmitting and receiving soft-cores for the EPXA1 device which eliminate the use of an external serializer and deserializer. The external LVDS circuitry should, however, be designed according to the LVDS design guidelines given in [25].

The CAN bus provides very strong error management, fault isolation and fault tolerance. The network is fully deterministic, supports priority and can operate at bit speeds of 1.25 Mbps. These features make it feasible to use in a satellite system [5]. There are various commercial CAN IP cores available and the EPXA1 eCos port supports the implementation of a CAN controller in the PLD logic.

8.2.3 EDAC Unit

The EDAC unit implemented in the PLD is far from ideal and there is a lot of room for improvement. The EDAC can be implemented with a flow through architecture by not using the existing memory interfaces, but rather a soft-core memory controller in the PLD logic.

Optimizing the EDAC unit should also be considered. This will increase the memory access times and therefore the OBC's overall performance.

8.2.4 Development Board

A development board specific for the evaluation of the EPXA1 device as an OBC, should be designed. The development board used in this study has a lot of limitations.

Provision should be made for the accurate measurement of the EPXA1 power consumption because power is one of the critical elements in a satellite's design. Provision should also be made for communication interfaces that require specific circuit designs, like LVDS.

Appendix A

PLD Modules

The OBC modules implemented in the PLD are as follows:

- AHB Bus Interface
- EDAC Unit
- UART
- I^{2}C Controller
- PLD Interrupt Controller

The Quartus Block Diagram file is shown in Figure A.1, followed by some of the module's algorithmic state machine (ASM) chart. The 6 individual interrupt mode is used in this design and therefore requires no state machine.

A.1 Block Diagram

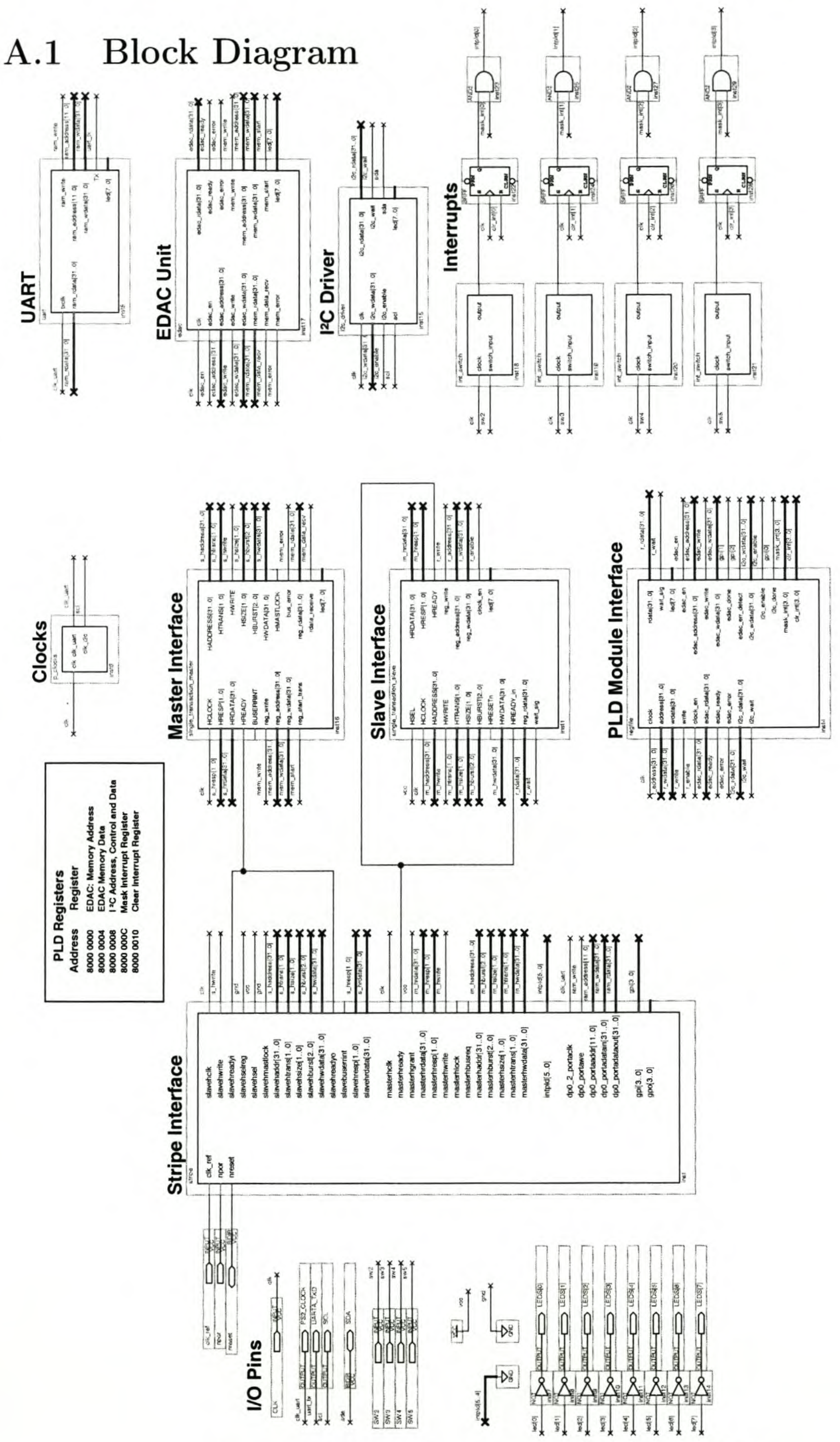

UART
EDAC Unit
I²C Driver
Interrupts
Clocks
PLD Registers
Address Register
8000 0000 EDAC: Memory Address
8000 0004 EDAC Memory Data
8000 0008 I ²C Address, Control and Data
8000 000C Mask Interrupt Register
8000 0010 Clear Interrupt Register
Master Interface
Slave Interface
PLD Module Interface
Stripe Interface
I/O Pins

Figure A.1: OBC Modules Implemented in the PLD

A.2 PLD Master Interface

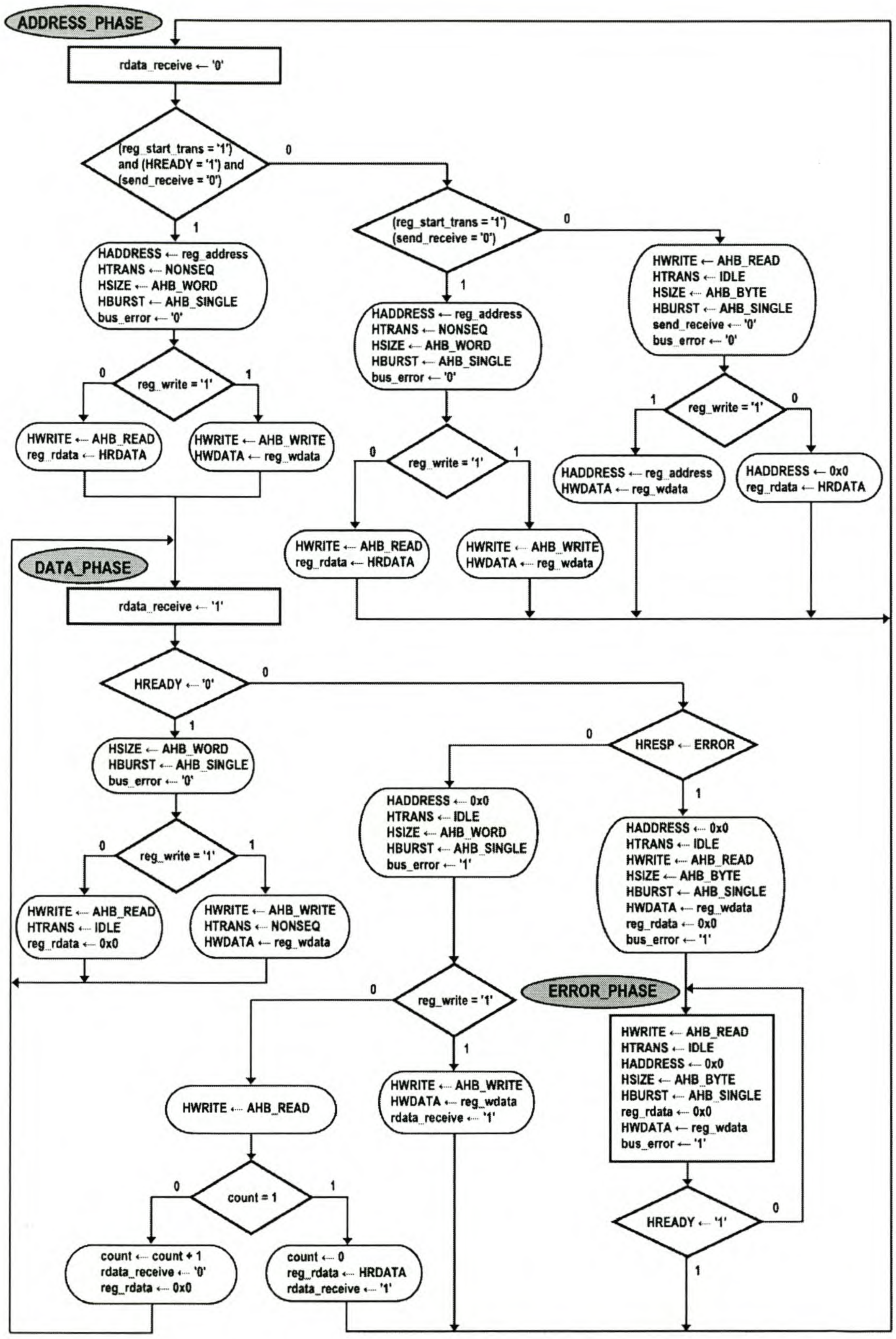


Figure A.2: ASM Chart of Master Interface

A.3 PLD Slave Interface

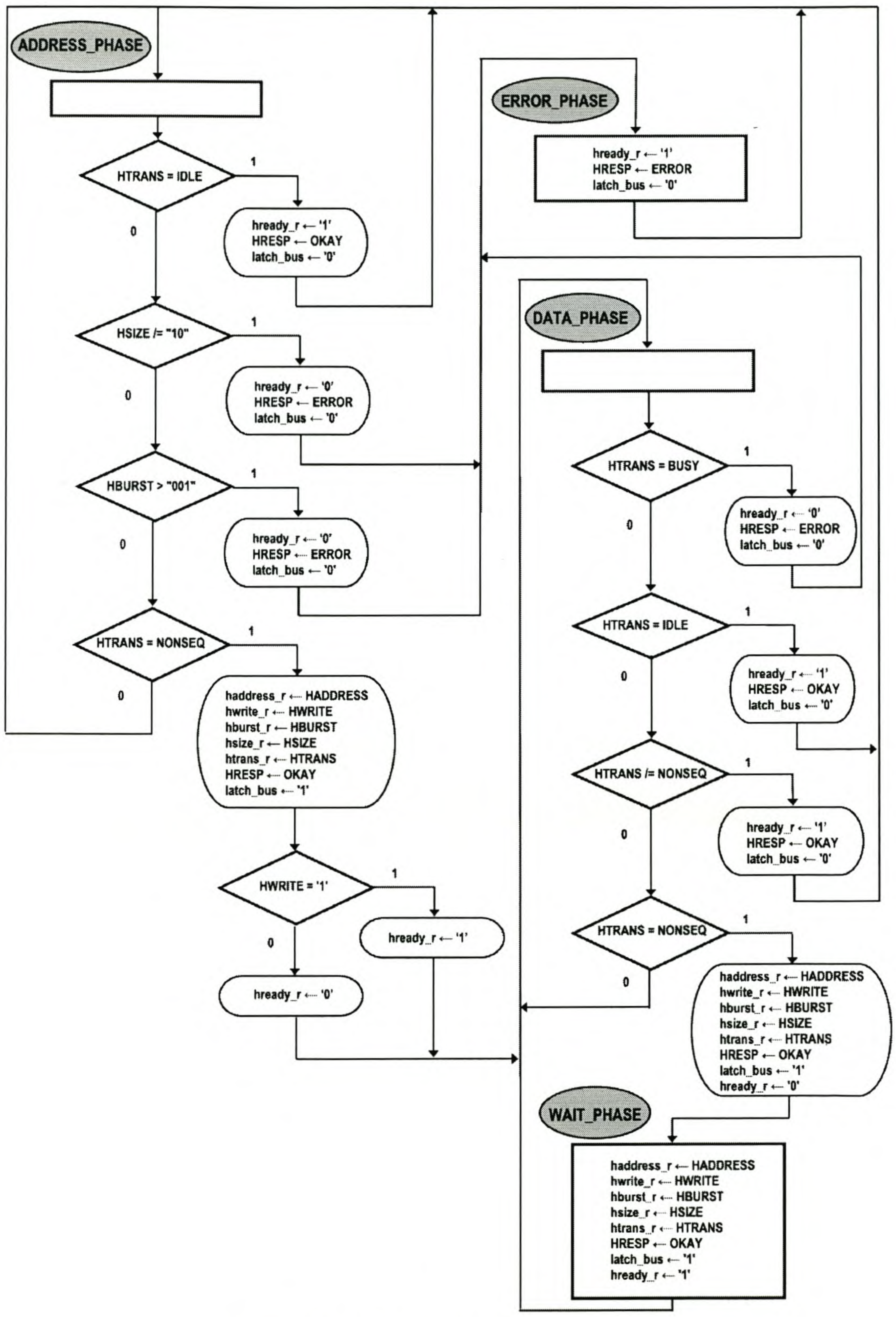


Figure A.3: ASM Chart of Slave Interface

A.4 PLD UART

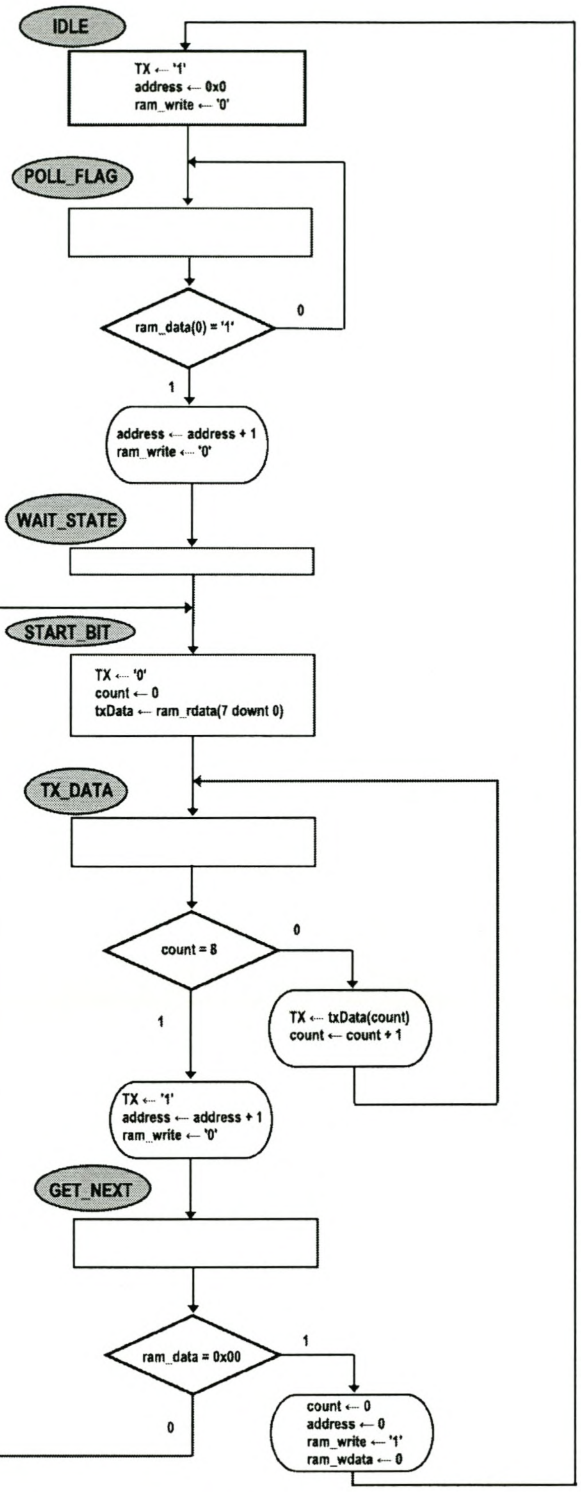


Figure A.4: ASM Chart of PLD UART

Appendix B

PLD Interrupt Controller

Extracts from the 6-bit priority interrupt mode's VSR, ISR and DSR source code are given in the following sections. Figure B.1 shows the Quartus block diagram for the 6-bit priority mode. All three interrupt modes were implemented and the design files are include on the attached CD-ROM.

B.1 Block Diagram

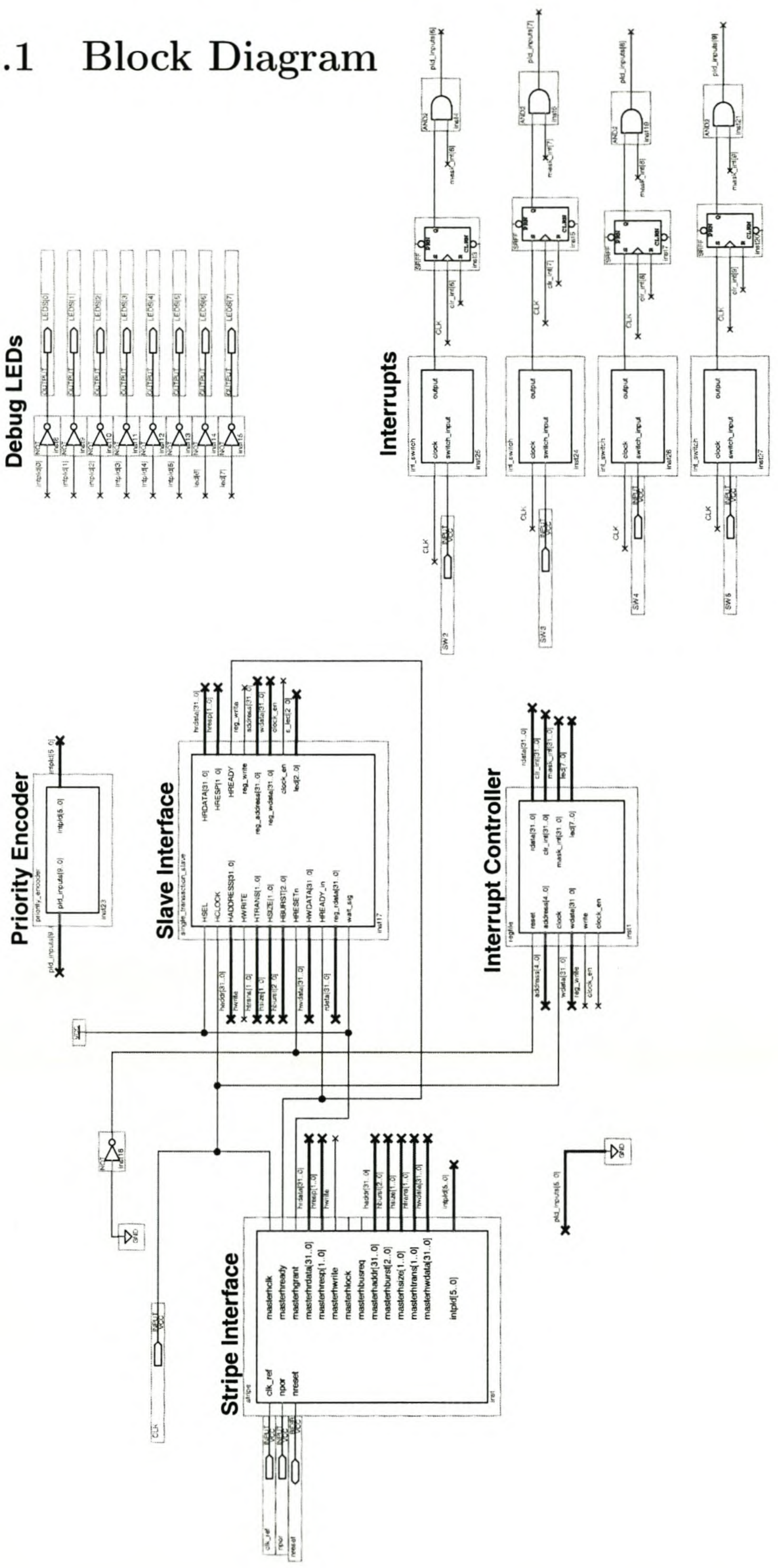


Figure B.1: 6-Bit Priority Interrupt Mode Block Diagram

B.2 Installed ISRs and DSRs

```
/* J.Jordaan 2004 */
/* this program test the 6 bit priority interrupt controller */

#include <cyg/infra/diag.h>
#include <cyg/hal/pld_config.h>
#include <cyg/hal/hal_io.h>
#include <cyg/hal/epxa1.h>
#include <cyg/kernel/kapi.h>

static cyg_interrupt pld_int6;
static cyg_handle_t pld_int6_handle;

static cyg_interrupt pld_int7;
static cyg_handle_t pld_int7_handle;

static cyg_interrupt pld_int8;
static cyg_handle_t pld_int8_handle;

static cyg_interrupt pld_int9;
static cyg_handle_t pld_int9_handle;

//define the new pld interrupt vector numbers
#define CYGNUM_HAL_INTERRUPT_PLD_PRIO_10      27 //Int9
#define CYGNUM_HAL_INTERRUPT_PLD_PRIO_09      26 //Int8
#define CYGNUM_HAL_INTERRUPT_PLD_PRIO_08      25 //Int7
#define CYGNUM_HAL_INTERRUPT_PLD_PRIO_07      24 //Int6


static cyg_sem_t data_ready;

/*------------------------ISR for PLD interrupt 6------------------------*/
//Interrupt from pld with int_pld[5..0] = 000111
cyg_uint32 pld_int6_isr(cyg_vector_t vector, cyg_addrword_t data)
{
    //Disable the pld interrupt through PLD mask register
    //No need to unmask it in processor's int_mask register => not supported
    HAL_WRITE_UINT32(0x80000004, 0x03BF);

    //Acknowledge the interrupt by clearing the int_clear register in PLD
    HAL_WRITE_UINT32(0x80000000, 0x0040);

    //Turn over to DSR
    return ( CYG_ISR_HANDLED | CYG_ISR_CALL_DSR );
}

/*------------------------DSR for PLD interrupt 6------------------------*/
void pld_int6_dsr (cyg_vector_t vector, cyg_ucount32 count,
                             cyg_addrword_t data)
{
    cyg_semaphore_post(&data_ready);

    //Enable the interrupt again
    HAL_WRITE_UINT32(0x80000004, 0x03FF);  //Enable int6 again

    //Output to terminal which switch caused the interrupt
    diag_printf("Interrupt␣Switch␣2!\n");
}

/*-----------------------------ISR for PLD interrupt 7-----------------------------*/
//int_pld[5..0] = 000010
cyg_uint32 pld_int7_isr(cyg_vector_t vector, cyg_addrword_t data)
{
    HAL_WRITE_UINT32(0x80000004, 0x037F);  //disable Int 7
    HAL_WRITE_UINT32(0x80000000, 0x0080);  //clear Int7 in PLD

    return ( CYG_ISR_HANDLED | CYG_ISR_CALL_DSR );

}

/*-----------------------------DSR for PLD interrupt 7-----------------------------*/
void pld_int7_dsr (cyg_vector_t vector, cyg_ucount32 count,
                           cyg_addrword_t data)
{
    cyg_semaphore_post(&data_ready);
    HAL_WRITE_UINT32(0x80000004, 0x03FF);  //Enable int7 again

    diag_printf("Interrupt␣Switch␣3!\n");
}

/*-----------------------------ISR for PLD interrupt 8-----------------------------*/
cyg_uint32 pld_int8_isr(cyg_vector_t vector, cyg_addrword_t data)
{

    HAL_WRITE_UINT32(0x80000004, 0x2FF);  //disable Int 8
    HAL_WRITE_UINT32(0x80000000, 0x00100);  //clear Int8 in PLD

    return ( CYG_ISR_HANDLED | CYG_ISR_CALL_DSR );

}

/*-----------------------------DSR for PLD interrupt 8-----------------------------*/
void pld_int8_dsr (cyg_vector_t vector, cyg_ucount32 count,
                           cyg_addrword_t data)
{
    cyg_semaphore_post(&data_ready);
    HAL_WRITE_UINT32(0x80000004, 0x03FF);  //Enable int8 again

    diag_printf("Interrupt␣Switch␣4!\n");
}

/*-----------------------------ISR for PLD interrupt 9-----------------------------*/
cyg_uint32 pld_int9_isr(cyg_vector_t vector, cyg_addrword_t data)
{

    HAL_WRITE_UINT32(0x80000004, 0x01FF);  //disable Int 9
    HAL_WRITE_UINT32(0x80000000, 0x0200);  //clear Int9 in PLD

    return ( CYG_ISR_HANDLED | CYG_ISR_CALL_DSR );

}

/*-----------------------------DSR for PLD interrupt 9-----------------------------*/
void pld_int9_dsr (cyg_vector_t vector, cyg_ucount32 count,
                           cyg_addrword_t data)
```

```
{
    cyg_semaphore_post(&data_ready);
    HAL_WRITE_UINT32(0x80000004, 0x03FF);  //Enable int9 again

    diag_printf("Interrupt␣Switch␣5!\n");
}

/*-----------------------------Install ISRs and DSRs-----------------------*/
void cyg_user_start(void)
{
    //define the interrupt vectors and their priorities
    //note: these vector numbers had to be added to the existing ones
    //in hal_intr.h
    cyg_vector_t pld_int6_vector = CYGNUM_HAL_INTERRUPT_PLD_PRIO_07;
    cyg_priority_t pld_int6_priority = 7;

    cyg_vector_t pld_int7_vector = CYGNUM_HAL_INTERRUPT_PLD_PRIO_08;
    cyg_priority_t pld_int7_priority = 8;

    cyg_vector_t pld_int8_vector = CYGNUM_HAL_INTERRUPT_PLD_PRIO_09;
    cyg_priority_t pld_int8_priority = 9;

    cyg_vector_t pld_int9_vector = CYGNUM_HAL_INTERRUPT_PLD_PRIO_10;
    cyg_priority_t pld_int9_priority = 10;

    cyg_semaphore_init( &data_ready, 0);

    //create interrupt 6 of PLD
    cyg_interrupt_create(pld_int6_vector, pld_int6_priority,0,&pld_int6_isr,
                         &pld_int6_dsr,&pld_int6_handle, &pld_int6);
    cyg_interrupt_attach(pld_int6_handle);

    //create interrupt 7
    cyg_interrupt_create(pld_int7_vector, pld_int7_priority,0,&pld_int7_isr,
                         &pld_int7_dsr,&pld_int7_handle, &pld_int7);
    cyg_interrupt_attach(pld_int7_handle);

    //create interrupt 8
    cyg_interrupt_create(pld_int8_vector, pld_int8_priority,0,&pld_int8_isr,
                         &pld_int8_dsr,&pld_int8_handle, &pld_int8);
    cyg_interrupt_attach(pld_int8_handle);

    //create interrupt 9
    cyg_interrupt_create(pld_int9_vector, pld_int9_priority,0,&pld_int9_isr,
                         &pld_int9_dsr,&pld_int9_handle, &pld_int9);
    cyg_interrupt_attach(pld_int9_handle);
}
```

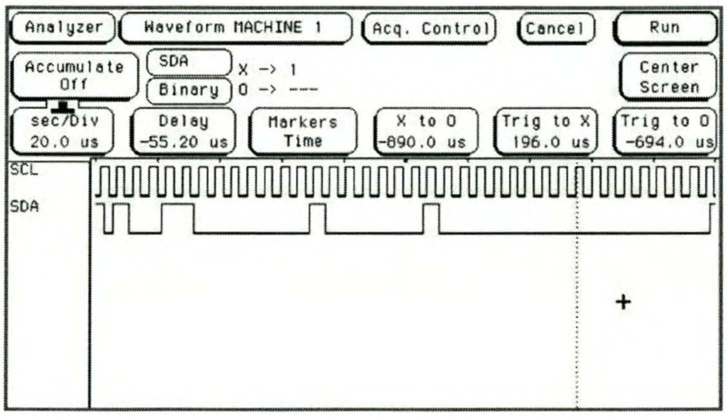
B.3 Installed VSR

```
/* J. Jordaan 2004
 Installed IRQ handler with the 3 interrupt mode support.
 Modified in epxa1_misc.c */

// This procedure returns the interrupt vector number
int hal_IRQ_handler(void)
{
    int vec;
    cyg_uint32 isr ,id ,mode, pld_pri;
     //read interrupt mode
    HAL_READ_UINT32(EPXA1_INT_INT_MODE, mode);
    if (mode == 0x0)  {                    //6-bit mode
        HAL_READ_UINT32(EPXA1_INT_ID, id);
        HAL_READ_UINT32(EPXA1_INT_PLD_PRIORITY, pld_pri);
        if (id == pld_pri) {  //pld interrupt
           // for test use priorities = vector => priorities between 1 and 5
            vec = (int)pld_pri;
           //lowest pld priority > vector 16 to prevent using an existing vector
           vec = vec + 17;           // pld vectors start at 18 for priority 1
           return vec;               // return vector number
        } else { //normal interrupts - don't check priority ,
                 //check int request status register
           HAL_READ_UINT32(EPXA1_INT_REQUEST_STATUS, isr);
           for (vec = CYGNUM_HAL_INTERRUPT_PLD_0;
                     vec <= CYGNUM_HAL_INTERRUPT_FAST_COMMS; vec++) {
                     if (isr & (1<<vec)) {
                     return vec;
                     }
           }
        }

    }
    else if (mode == 0x01) {     //five - bit priority
        HAL_READ_UINT32(EPXA1_INT_ID, id);
        if (id == 0x10) {       //Value corresponds to the id attached to the ISR
                vec = 33;
                return vec;
        }
        if (id == 0x12) {
                vec = 35;
                return vec;
        }
          if (id == 0x14) {
                vec = 37;
                return vec;
        }
        // normal interrupts
        HAL_READ_UINT32(EPXA1_INT_REQUEST_STATUS, isr);
        for (vec = CYGNUM_HAL_INTERRUPT_PLD_0;
                     vec <= CYGNUM_HAL_INTERRUPT_FAST_COMMS; vec++) {
                     if (isr & (1<<vec)) {
                     return vec;
                     }
        }
    }
    else {     // 6 individual interrupt mode
            HAL_READ_UINT32(EPXA1_INT_REQUEST_STATUS, isr);
            for (vec = CYGNUM_HAL_INTERRUPT_PLD_0;
                    vec <= CYGNUM_HAL_INTERRUPT_FAST_COMMS; vec++) {
                    if (isr & (1<<vec)) {
                    return vec;
                    }
            }
    }
    return CYGNUM_HAL_INTERRUPT_NONE;    //No corresponding interrupt source
}
```

Appendix C

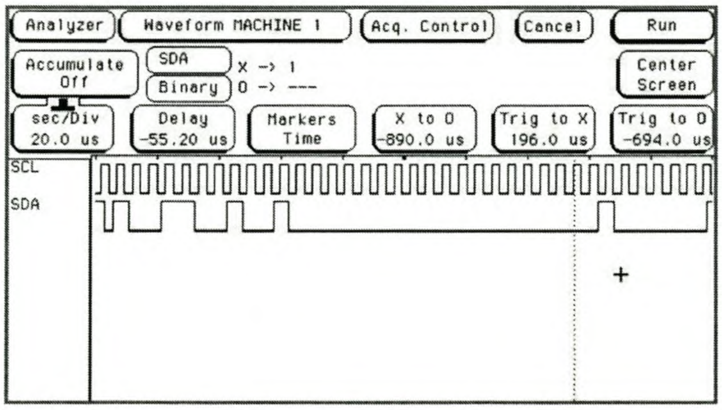
I^{2}C Measurements

Figure C.1 and Figure C.2 show the I^{2}C write and read transactions in fast mode measured by the logic analyzer.



Accumulate Off
SDA Binary
X -> 1
0 -> ---
sec/Div 20.0 us
Delay -55.20 us
Markers Time
X to O -890.0 us
Trig to X 196.0 us
Trig to O -694.0 us

Figure C.1: I^{2}C Write Transaction in Fast Mode



Accumulate Off
SDA Binary
X -> 1
0 -> ---
sec/Div 20.0 us
Delay -55.20 us
Markers Time
X to O -890.0 us
Trig to X 196.0 us
Trig to O -694.0 us

Figure C.2: I^{2}C Read Transaction in Fast Mode

Bibliography

[1] MNL-EPXA10HRM-3.1: *Excalibur Devices: Hardware Reference Manual Version 3.1.* Altera, November 2002.

[2] ARM DDI 0100E: *ARM Architecture Reference Manual.* ARM Limited, 2000.

[3] Peter Fortescue, J.S. and Swinerd, G.: *Spacecraft Systems Engineering.* 3rd edn. Morgan Kaufmann Publishers, San Fransico, 2000.

[4] Altera: Excalibur embedded processor pld stripe power consumption. [Online] Available: `http://www.altera.com/literature/wp/wp_epxan_power_consumption.pdf`, [2004, November 17], June 2002.

[5] Hans Tiggeler, Tanya Vladimirova, D.Z. and Gaisler, J.: Experiences designing a system-on-a-chip for small satellite data processing and control. [Online] Available at: `http://www.ee.surrey.ac.uk/Personal/T.Vladimirova/Publications/MAPLD00%_P20.pdf`, [2004,November 1], 2000.

[6] Grobler, H.: *Aspects Affecting the Design of a Low Earth Orbit Satellite OBC.* Master's thesis, University of Stellenbosch, 1996.

[7] Altera: [Online] Available at: `http://www.altera.com`, [2004, November 1], 2004.

[8] Barnard, A.: *Feasibility of using an ARM processor in a micro satellite on-board computer.* Master's thesis, University of Stellenbosch, December 2001.

[9] Dreijer, G.: *The evaluation of an ARM-based on-board computer for a low earth orbit satellite.* Master's thesis, University of Stellenbosch, September 2002.

[10] DS-APEX20K-4.3: *APEX 20K Programmable Logic Device Family.* Altera, February 2002.

[11] *GNUPro Toolkit User's Guide for Altera for ARM and ARM/Thumb Development.* Red Hat, 2002.

[12] Nucleus Embedded Software Product Overview: [Online] Available at: `http://www.AcceleratedTechnology.com`, [2004, November 1], 2004.

[13] RTEMS ARM Application Supplement: [Online] Available at: `http://www.oarcorp.com`, [2004, November 1], January 2003.

[14] Linux: Real-Time and Performance: [Online] Available at: http://www.linuxdevices.com/products/PD8842791300.html, [2004, November 1], 2003.

[15] MontaVista Linux Professional Edition: [Online] Available at: http://www.mvista.com/, [2004, November 1], 2003.

[16] Massa, A.J.: *Embedded Software Development With eCos.* Prentice Hall, 2003.

[17] eCos Mailing List and Documentation: [Online] Available at: http://sources.redhat.com/ecos/, [2004, November 1], 2004.

[18] Altera: An 298: Reconfiguring excalibur devices under processor control. [Online] Available: http://www.altera.com/literature/an/an298.pdf, [2004, November 17], October 2002.

[19] *AMBA Specification, Rev 2.0.* ARM, 1999.

[20] Hodgart, M. and Tiggeler, H.: A (16,8) error correcting code (t=2) for critical memory applications. [Online] Available: http://www.estec.esa.nl/wsmwww/core/ipdoc/EDAC8Cyclic.pdf, [2004, November 17], 2000.

[21] *The I^2C-Bus Specification, Version 2.1.* Philips Semiconductors, January 2000.

[22] MNL-EPXA1DEVBD-1.0: *EPXA1 Development Board Version 1.0.* Altera, August 2002.

[23] Cravotta, R.: Edn magazine 32-bit processor's comparison. [Online] Available: http://www.edn.com/contents/images/245647t32bit.pdf, [2004, November 17], October 2002.

[24] Altera: Excalibur solutions - using the excalibur stripe plls. [Online] Available: http://www.altera.com/literature/an/an177.pdf, [2004, November 17], July 2002.

[25] Altera: An 120: Using lvds in apex20ke devices. [Online] Available: http://www.altera.com/literature/an/an120.pdf, [2004, November 17], May 2002.

Appendix D

CD-ROM Data

The attached CD-ROM contains the following data:

- VHDL source code of the PLD designs
- eCos program source code in C along with the eCos configuration
- Reference papers and datasheets