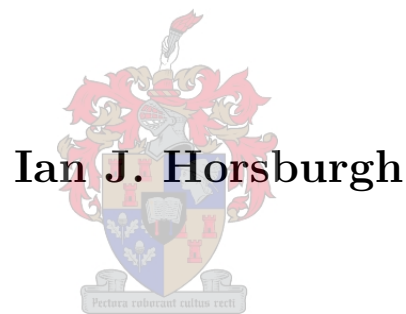


The Development of a Mass Memory Unit for a Micro-Satellite using NAND Flash Memory



Thesis presented in partial fulfilment of the requirements for the degree of Master of
Science Engineering at The University of Stellenbosch

Supervisor: Prof. P.J. Bakkes

Department of Electrical and Electronic Engineering
University of Stellenbosch
April 2005

Declaration

I declare that the contents of this thesis is original and my own work unless otherwise stated and that it has not, to my knowledge, been published in any part or as a whole at any other university in order to obtain a degree.

I.J. Horsburgh

Date

Abstract

This thesis investigates the possible use of NAND flash memory for a mass memory unit on a micro-satellite. The investigation begins with an analysis of NAND flash memory devices including the complexity of the internal circuitry and the occurrence of bad memory sections (bad blocks). Design specifications are produced and various design architectures are discussed and evaluated. Subsequently, a four bus serial access architecture using 16-bit NAND flash devices was chosen to be developed further.

A VHDL design was created in order to realise the intended system functionality. The main functions of the design include a sustained write data rate of 24 MB/s, bad block management, multiple image storing, error checking and correction, defective device handling and reading while writing. The design was simulated extensively using NAND flash simulation models.

Finally, a demonstration test board was designed and produced. This board includes an FPGA and an array of 16 8-bit NAND flash devices. The board was tested successfully and a write data rate of 12 MB/s was achieved along with all the other main functions.

Opsomming

Hierdie tesis ondersoek die moontlike gebruik van NAND flash tegnologie as die geheue eenheid van 'n mikrosatelliet. As 'n beginpunt word NAND flash tegnologie ondersoek in terme van die kompleksiteit van interne stroombane en die voorkoms van defektiewe geheusegmente. Daarna word ontwerp spesifikasies voortgebring en verskillende ontwerp moontlikhede met mekaar vergelyk. Vanuit hierdie oorwegings is daar besluit om die oplossing te implementeer met 'n vier-bus seriële struktuur bestaande uit 16-bis NAND flash toestelle.

Om die ontwerp spesifikasies te realiseer, is 'n VHDL stelsel geskep. Die belangrikste funksies van hierdie stelsel is 'n konstante skryf tempo van 24 MB/s, die bestuur van defektiewe geheusegmente, die stoor van meer as een beeld, foutopsporing en -herstel, optimale werking in die geval van defektiewe geheue toestelle en laastens, die gelyktydige lees en skryf van data. Die stelsel is breedvoerig getoets met NAND flash simulasiemodelle.

Ten slotte is 'n fisiese demonstrasiebord, bestaande uit 'n FPGA en 16 8-bis NAND flash toestelle, ontwerp en gebou. Fisiese metings was 'n sukses. 'n Skryf tempo van 12 MB/s is gehaal, tesame met die korrekte werking van die ander hoof funksies.

Acknowledgements

The author would like to thank the following:

- Professor P.J Bakkes (thesis supervisor) for his enthusiasm and support.
- Nicola Coetzee for love, encouragement and taking time out of her busy schedule to proof read this document.
- His parents for paying for those plane tickets back to Scotland when he needed a break.

Contents

| | Page |
|---|-----------|
| 1 Introduction | 1 |
| 1.1 Aims and Objectives | 1 |
| 1.2 The Rest of the Document | 2 |
| 2 Background | 4 |
| 2.1 Flash Memory | 4 |
| 2.1.1 What is Flash Memory? | 4 |
| 2.1.2 NAND versus NOR Flash | 4 |
| 2.2 NAND Flash Memory | 7 |
| 2.2.1 Memory Organisation and Interface | 7 |
| 2.2.2 Internal Structure | 8 |
| 2.2.3 Operation | 9 |
| 2.2.3.1 Writing/Programming | 9 |
| 2.2.3.2 Reading | 9 |
| 2.2.3.3 Erasing | 10 |
| 2.2.4 Bad Blocks | 11 |
| 2.3 Digital Design | 12 |
| 2.3.1 VHDL | 12 |
| 2.3.2 Synthesis | 12 |
| 2.3.3 Simulation | 13 |
| 2.3.4 FPGA's | 13 |
| 3 Design Architectures | 15 |
| 3.1 Design Specifications | 15 |
| 3.1.1 Data Rate Calculation | 15 |
| 3.1.2 Mass Memory Capacity | 17 |
| 3.1.3 Cost | 17 |
| 3.1.4 Physical Size and Mass | 18 |
| 3.1.5 Power | 18 |
| 3.1.6 Radiation Problems | 18 |

| | | |
|----------|---|-----------|
| 3.1.7 | Complexity | 19 |
| 3.1.8 | Reliability | 19 |
| 3.2 | NAND Flash Issues | 19 |
| 3.2.1 | Data Rate | 19 |
| 3.2.2 | Bad Blocks | 21 |
| 3.2.3 | Failure Modes Mechanisms and Symptoms | 21 |
| 3.2.3.1 | Permanent failures | 21 |
| 3.2.3.2 | Soft Errors | 21 |
| 3.3 | Design Alternatives | 23 |
| 3.3.1 | Page Size - 528 or 2112 Bytes? | 23 |
| 3.3.2 | I/O Lines - 8 or 16 bit? | 23 |
| 3.3.3 | Multiple Buses | 24 |
| 3.3.4 | Serial or Parallel? | 25 |
| 3.3.4.1 | Serial | 25 |
| 3.3.4.2 | Parallel | 26 |
| 3.3.5 | Number of Devices | 26 |
| 3.3.5.1 | Data Rate Constraints | 26 |
| 3.3.5.2 | Area Constraints | 27 |
| 3.3.5.3 | Power Constraints | 27 |
| 3.3.5.4 | Device Input Capacitance | 28 |
| 3.3.6 | Bad Block Table | 28 |
| 3.3.7 | Cache | 29 |
| 3.3.8 | Reference Tables | 30 |
| 3.4 | FPGA Considerations | 30 |
| 4 | System Design Overview | 33 |
| 4.1 | Design Decisions | 33 |
| 4.1.1 | NAND Device Flavour | 33 |
| 4.1.2 | Bus Access | 34 |
| 4.1.3 | Number of Buses | 35 |
| 4.1.4 | Number of Devices | 35 |
| 4.1.5 | Bad Block Table | 36 |
| 4.1.6 | Memory Structure | 37 |
| 4.1.7 | FPGA Selection | 38 |
| 4.1.7.1 | Development Software | 39 |
| 4.2 | VHDL Design Overview | 40 |
| 4.2.1 | Data Router | 40 |
| 4.2.2 | Bus Controller | 42 |
| 4.2.2.1 | Bad Block Table Unit | 43 |

| | | |
|----------|--|-----------|
| 4.2.2.2 | NAND Flash Interface Control | 46 |
| 4.2.2.3 | EDAC | 46 |
| 4.2.3 | Summary | 48 |
| 5 | Detailed System Design | 49 |
| 5.1 | Design Changes | 49 |
| 5.2 | VHDL Design Detail | 50 |
| 5.2.1 | Bus Controller | 50 |
| 5.2.1.1 | Interface Control Unit | 50 |
| 5.2.1.2 | Bad Block Table Unit | 53 |
| 5.2.1.3 | EDAC | 54 |
| 5.2.2 | Data Router | 57 |
| 5.2.2.1 | Bad Block Counter | 58 |
| 5.2.2.2 | Data Rate Calculation | 58 |
| 5.2.2.3 | Bus Select | 58 |
| 5.2.2.4 | Read, Write and Erase Routing Control | 62 |
| 5.2.3 | Device Deactivation | 62 |
| 5.2.4 | Testing and PC Interface | 63 |
| 5.2.5 | Command Structure | 64 |
| 5.3 | Printed Circuit Board Design - Schematics and Layout | 65 |
| 5.3.1 | FPGA and Configuration | 66 |
| 5.3.2 | NAND Flash Devices | 68 |
| 5.3.3 | Power Supply | 68 |
| 5.3.4 | Decoupling | 68 |
| 5.3.5 | LEDs and Test Points | 69 |
| 5.3.6 | Clk and Reset | 69 |
| 5.3.7 | Device Placement and Routing | 69 |
| 5.4 | Read While Write | 71 |
| 5.4.1 | Motivation | 71 |
| 5.4.2 | Design Solution | 71 |
| 6 | Simulations and Results | 73 |
| 6.1 | VHDL Simulations | 73 |
| 6.2 | NAND Flash Simulation Model | 73 |
| 6.2.1 | Basic Operations | 74 |
| 6.2.2 | Bad Block Table | 76 |
| 6.2.3 | Image Write Sequence | 79 |
| 6.2.4 | Image Read Sequence | 81 |
| 6.2.5 | Image Erase Sequence | 83 |

CONTENTS

viii

| | | |
|----------|---|------------|
| 6.2.6 | Device Deactivation | 83 |
| 6.2.7 | Error Detection and Correction | 86 |
| 6.3 | Demonstration Board Testing and Results | 87 |
| 6.3.1 | Factory Marked Bad Blocks | 89 |
| 6.3.2 | PCB Results | 90 |
| 6.3.3 | Power and Area Calculations | 92 |
| 6.3.4 | 16-Bit Device Write Data Rate | 93 |
| 6.3.5 | Read While Write Simulation | 95 |
| 7 | Conclusions And Recommendations | 98 |
| 7.1 | Conclusions | 98 |
| 7.2 | Recommendations | 99 |
| A | Reference Tables | 102 |
| B | Demonstration Board Schematics | 105 |
| C | VHDL Code | 115 |

List of Figures

| Figure | Page |
|---|------|
| 2.1 NAND and NOR Structure [16] | 5 |
| 2.2 NAND Programming and Erasing [15] | 6 |
| 2.3 Samsung K9F1208U0M - Internal Structure [8] | 8 |
| 2.4 NAND - Example Program Cycle [15] | 9 |
| 2.5 NAND - Example Read Cycle [15] | 10 |
| 2.6 NAND - Example Erase Cycle [15] | 10 |
| 2.7 Program, Read and Erase Data Transfers [15] | 11 |
| 3.1 Satellite Subsystem Configuration | 16 |
| 3.2 Programming with Multiple Devices | 20 |
| 3.3 Program and Erase Endurance [15] | 22 |
| 3.4 NAND Flash Serial [11] | 25 |
| 3.5 NAND Flash Parallel [11] | 26 |
| 3.6 NAND Flash Cache Program Operation [12] | 29 |
| 4.1 Proposed Bus Architecture | 36 |
| 4.2 Proposed Memory Structure | 38 |
| 4.3 Main Design Block Diagram | 41 |
| 4.4 Data Router Block Diagram | 42 |
| 4.5 Bus Controller Block Diagram | 43 |
| 4.6 Bad Block Table Format | 44 |
| 4.7 Bad Block Table Block Diagram | 45 |
| 4.8 Interface Control Block Diagram | 47 |
| 4.9 EDAC Block Diagram | 47 |
| 5.1 Bad Block Table - Address Generation Flow Diagram | 55 |
| 5.2 Bad Block Table - Update and Store | 56 |
| 5.3 Data Router - Count Bad Blocks Flow Diagram | 59 |
| 5.4 Data Router - Data Rate Calculation | 60 |
| 5.5 Data Router - Bus Select Flow Diagram | 61 |

LIST OF FIGURES

x

| | | |
|------|---|-----|
| 5.6 | Test Generator and PC Interface | 64 |
| 5.7 | FPGA Configuration | 67 |
| 5.8 | PCB Layers | 68 |
| 5.9 | Device Layout and Routing Top | 70 |
| 5.10 | Device Layout Routing Bottom | 70 |
| 5.11 | Read-While-Write System Structure | 72 |
| 5.12 | Read-While-Write Memory Structure | 72 |
| 6.1 | VHDL Simulation - Write, Read and Erase | 75 |
| 6.2 | VHDL Simulation - Store Command | 77 |
| 6.3 | VHDL Simulation - Load Command | 78 |
| 6.4 | VHDL Simulation - Write Sequence | 80 |
| 6.5 | VHDL Simulation - Read Sequence | 82 |
| 6.6 | VHDL Simulation - Erase Sequence | 84 |
| 6.7 | VHDL Simulation - Write Sequence with Bus 2 Deactivated | 85 |
| 6.8 | VHDL Simulation - Load Bad Block Table ECC Detection | 86 |
| 6.9 | VHDL Simulation - Load Bad Block Table ECC Correction | 86 |
| 6.10 | PCB - FRONT | 88 |
| 6.11 | PCB - BACK | 88 |
| 6.12 | Flow Chart - Retrieving Factory Marked Bad Blocks [8] | 89 |
| 6.13 | Oscilloscope Capture - Part of Erase Sequence | 91 |
| 6.14 | Oscilloscope Capture - Part of Read Sequence | 91 |
| 6.15 | Oscilloscope Capture - Part of Write Sequence | 92 |
| 6.16 | VHDL Simulation - 16-Bit Device Part Image Write Sequence | 93 |
| 6.17 | VHDL Simulation - 16-Bit Device Image Write Sequence | 94 |
| 6.18 | VHDL Simulation - Reading While Writing 1 | 96 |
| 6.19 | VHDL Simulation - Reading While Writing 2 | 97 |
| B.1 | Schematics Page 1 | 106 |
| B.2 | Schematics Page 2 | 107 |
| B.3 | Schematics Page 3 | 108 |
| B.4 | Schematics Page 4 | 109 |
| B.5 | Schematics Page 5 | 110 |
| B.6 | Schematics Page 6 | 111 |
| B.7 | Schematics Page 7 | 112 |
| B.8 | Schematics Page 8 | 113 |
| B.9 | Schematics Page 9 | 114 |

List of Tables

| Table | Page |
|--|------|
| 3.1 Number of FPGA pins needed for multiple buses | 24 |
| 3.2 Comparison of available low-cost FPGAs | 32 |
| 4.1 Cyclone Family of FPGAs | 39 |
| 5.1 Section Sizing with Defective Devices | 63 |
| 5.2 List of Commands | 65 |
| 6.1 Bad Block Addresses for Load and Store Simulations | 76 |
| 6.2 Factory Marked Bad Blocks | 90 |
| 6.3 FPGA Power Consumption | 92 |
| A.1 Maximum Data Rate and Minimum Number of Devices | 103 |
| A.2 Power and Area Calculations | 104 |

List of Abbreviations and Symbols

| | |
|---------------|---|
| ASIC | Application Specific Integrated Circuit |
| CAD | Computer Aided Design |
| ECC | Error Checking and Correction |
| EDA | Electronic Design Automation |
| EDAC | Error Detection and Correction |
| EPROM | Erasable Programmable ROM |
| EEPROM | Electrically Erasable Programmable ROM |
| FIFO | First In First Out Register |
| FPGA | Field Programmable Gate Array |
| HDL | Hardware Descriptive Language |
| IC | Integrated Circuit |
| JTAG | Joint Action Test Group |
| LED | Light Emitting Diode |
| LEO | Low Earth Orbit |
| Mb | Megabit |
| MB | Megabyte |
| MMU | Mass Memory Unit |
| OBC | On-Board Computer |
| PCB | Printed Circuit Board |
| RAM | Random Access Memory |
| ROM | Read Only Memory |
| SRAM | Static RAM |
| SUNSAT | Stellenbosch University Satellite |
| VHDL | VHSIC HDL |
| VHSIC | Very High Speed IC |

Chapter 1

Introduction

The Electrical and Electronic Engineering department at Stellenbosch University created the SUNSAT program in 1991 to help inspire postgraduate students in the department by facilitating their involvement in a unique and challenging real-world application. The SUNSAT program was responsible for developing and launching South Africa's first satellite, the SUNSAT-1 micro-satellite. The Electronic Systems Laboratory (ESL) was established to serve as a facility where SUNSAT-1 could be designed and built, and also for continuing satellite research.

The primary payload of SUNSAT-1 was a high resolution (15m) push broom imager. The images captured were stored on board the satellite and downloaded to the Stellenbosch-based ground station when it was within communication range. A mass-memory unit (MMU) was used to store these images on board the satellite and was implemented using an array of SRAM memory devices. The next generation micro-satellite MMU's must be capable of storing larger amounts of data at higher data rates in order to keep up with the advances in imager technology and the demand for higher resolution images.

The memory medium NAND flash has emerged over the last few years as the market leader in high density non-volatile data storage. The widespread use of flash memory in commercial digital storage applications has driven its development, with the result that 1GB of memory is now available in a single device package. The low cost per bit, low power consumption and small package size make NAND flash an ideal candidate for a MMU on a micro-satellite.

1.1 Aims and Objectives

The main objective of this thesis is to investigate NAND flash memory for a MMU on the next generation micro satellites. This research project will look mainly at the capabilities

of the technology, the different architectures of devices and various design implementations.

The next aim is to design a MMU which implements a chosen design architecture. This board is not intended as a fully functional prototype, but as a demonstration board of NAND flash capabilities. Finally, a reading while writing function for the MMU is to be investigated. The radiation effects on the devices are not covered in this text.

1.2 The Rest of the Document

Chapter 2 briefly discusses flash technology and the latest techniques of digital electronic design. Firstly NAND and NOR flash devices are compared in detail with regards to physical size, cost, density, power and compatibility. The complex internal architecture of NAND flash devices are then discussed, followed by the basic operations of the device and an introduction to defective blocks contained within these devices.

Chapter 3 looks more closely at the design specifications and investigates the various implementations to meet them. The chapter begins by looking at each individual problem associated with designing a MMU on board a micro-satellite and then looks at how an array of NAND flash devices can be implemented to solve these problems. Bad block tables and error detection and correction schemes follow. The chapter finishes by looking at the available low cost FPGA's on the market.

In Chapter 4 the important design decisions are made and a proposed architecture for the final design is given. The VHDL architecture is analysed and the intended functionality of each component is given.

Chapter 5 looks in detail at the VHDL design for the demonstration MMU. The low level functionality of each VHDL component is described and a command structure is introduced. The design of the demonstration board PCB follows and a read-while-write implementation proposal concludes the chapter.

Chapter 6 shows the results and simulations for the system design. The VHDL system design is simulated and verified with NAND flash memory models. Next, the PCB is built and checked before the full system is tested. A power analysis of the board follows and the simulation of the read-while-write implementation concludes the chapter.

Chapter 7 signals the end of the thesis with conclusions that were drawn from the project.

Proposals for future studies and suggestions to improve the design are also given.

Chapter 2

Background

This chapter analyses and compares NAND and NOR flash memory, and examines the reasons why NAND flash in particular was chosen for this project. A look at the structure and organisation of a typical NAND device and its basic operations is followed by an introduction to digital design with VHDL and FPGA's.

2.1 Flash Memory

2.1.1 What is Flash Memory?

Flash memory is a non-volatile memory storage medium, which means that it retains data after power is switched off. It has evolved from erasable programmable read only memory (EPROM) into a type of electronically erasable programmable read only memory (EEPROM). The term 'flash' is derived from the expression that the whole memory could be erased in a 'flash'. Flash memory today is divided into blocks that can be erased individually for better versatility. What separates normal EEPROM's from flash is the fact that normal EEPROM's can erase and reprogram on a byte or word basis, while flash memory can be programmed on a byte or word basis but must be erased a block at a time as mentioned earlier. The major advantage that flash memory has over normal EEPROM's is that flash uses only one transistor to store a bit of data, while in an EEPROM two transistors are required. This means the cost per bit is significantly reduced, which is a much bigger advantage to the memory market than the flexibility of byte programming. The two main types of flash memory are NAND flash and NOR flash, and the differences in their architectures will be looked at next.

2.1.2 NAND versus NOR Flash

The majority of semiconductor memories achieve random access by connecting memory cell transistors to the bit lines in parallel. This is the same for NOR-type flash. If any

memory cell is turned on by the corresponding word line, the bit line goes low. Since this logic function is similar to a NOR gate, this cell arrangement is known as NOR flash memory. The architecture of NAND-type flash is a series connection of memory cells transistors to the bit line. In a NAND flash memory array, the bit line goes low if all the inputs are high. This logic function is similar to that of a NAND logic gate. The contrasting structures of these two types of flash memory is illustrated in Figure 2.1.

| | NAND | NOR |
|---------------|--------------------------|---------------------------|
| Cell Array | | |
| Layout | | |
| Cross-section | | |
| Cell size | $4F^2$ | $10F^2$ |

Figure 2.1: *NAND and NOR Structure [16]*

The biggest advantage that NOR flash has over NAND flash is the faster random access speed due to the memory cells being connected to the bit lines in parallel. This parallel connection is also the cause of its biggest disadvantage: its cell size. The NOR structure provides direct access to individual cells. This simplifies the overall device architecture, but increases cell area because of the need for contacts at each drain and source connection. A NAND bit line is a series connection of transistors, which means each transistor only has to pass a small amount of current. Physically, NAND devices use a smaller transistor because they do not have to “pull-down” a whole bit line of relatively high capacitance. The NAND cell is also more compact, as it does not provide contacts to individual source and drain regions. However, cells in the NAND structure require reading and writing through other cells in the stack; an architecture that results in inherently slow access.

In the case of semiconductor memory, the bit cost is dependent on the memory cell area

per bit, and Figure 2.1 shows the size of a NOR cell is $10F^2$ compared to $4F^2$ for a NAND cell. Cost is also a function of die size, and with NOR flash, a relatively large amount of extra memory cells are fabricated on the die. These are used to repair defects in the memory array in order to produce a device that has faultless memory locations. NAND flash, on the other hand, does not require this large amount of extra memory to repair defective regions on the die because it is not necessary to repair the entire array. NAND flash can instead contain defective areas, comprising no more than 2% of the total memory array (see Section 2.2.4), and therefore the production yield is much higher, again resulting in a significant cost reduction. The fact that NAND cells are considerably smaller than NOR cells also means that greater densities of NAND flash devices are possible on a standard package.

NOR and NAND Flash both use electron tunneling for programming and erasing (see Figure 2.2), providing low power consumption. With NAND flash electron tunneling occurs across the whole channel area of a cell. This means charge trapped per density is lower, resulting in superior endurance in program/erase cycles to NOR flash. A charge pump is required in order to provide the high internal voltages that are required for erase and write operations. Cells in the NAND structure require higher voltages, typically 20V for erasing and writing, compared with the 12V required for the NOR structure.

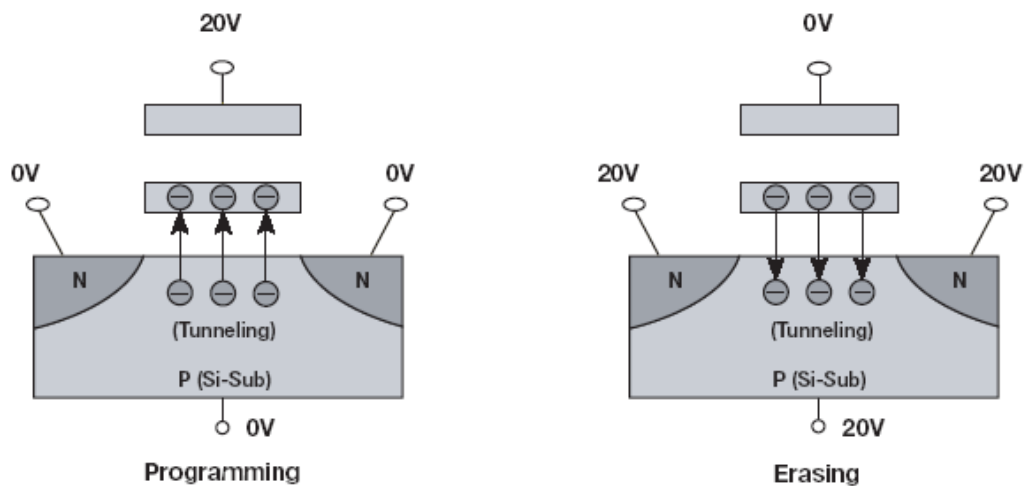


Figure 2.2: *NAND Programming and Erasing [15]*

NOR applications typically use the memory to store and execute Operating System code. As a result, the NOR products evolved with lower density and higher random access performance. On the other hand, NAND is more appropriate for mass storage applications. The time it takes for a NAND flash device to read out the first data byte is significantly greater than that of NOR, but it is faster than the seek time for a hard disk by several orders of magnitude. Therefore it is seen as a possible solid state replacement to hard

disks. The term ‘solid state’ refers to the fact that the device has no movable parts. NOR flash has been around for much longer than NAND flash. As a result, problems have arisen with supplier incompatibility. Many suppliers have carved out niches by modifying the basic NOR architecture, either to suite a unique application or to avoid patents. This has led to an incompatibility in the NOR family of memories, which means that picking the wrong supplier could leave the designer stranded when the device capacity becomes exhausted. This can lead to a total redesign when a different supplier is used.

The demand for higher density storage in the last few years has pushed the evolution of NAND flash memory. Consumer applications such as cellphones, digital cameras, portable music devices, and USB flash memory drives have been the main protagonists in this increase in demand. Learning from the NOR standard’s lessons and having fewer suppliers, the NAND architecture has both scalability and compatibility, making it a more dependable memory to source. The random access times which are the main advantages of NOR flash, are not critical for mass storage, but size, endurance and cost are. Therefore, NAND flash is an obvious choice as a suitable non-volatile memory device for use on a micro-satellite mass memory unit.

2.2 NAND Flash Memory

2.2.1 Memory Organisation and Interface

The NAND flash device’s main memory array is split into blocks and pages. Each block has either 32 or 64 pages, and each page has either 528 or 2112 bytes. A page is split into two sections: a “normal area” consisting of 512 or 2048 bytes and a “spare area” consisting of 16 or 64 bytes. The number of blocks is determined by the size of the device. For example, the 64 MB device has $4096 \text{ blocks} \times 32 \text{ pages} \times 528 \text{ bytes}$.

The standard NAND flash device interface is indirect; it has no dedicated address or data lines, just control lines and either 8 or 16 bidirectional I/O lines. The commands, addresses and data inputs are multiplexed onto the I/O lines. The address is input to the device through a series of cycles, while the address latch enable pin is held high and similarly, the commands are input to the device while the command latch enable pin is held high. The higher density devices simply have more address cycles to access the greater amount of blocks or pages. This scheme reduces pin counts dramatically and allows for system upgrades to future densities by maintaining consistency in the system board design.

2.2.2 Internal Structure

The overall internal architecture of a NAND flash device is extremely complex as can be seen in Figure 2.3, which shows a block diagram of the Samsung K9F1208U0M device. The internal architecture is structured efficiently in an attempt to shield the user from the complexity of the device and provide a transparent, user friendly interface. A description of the NAND flash interface pins can be found in [8].

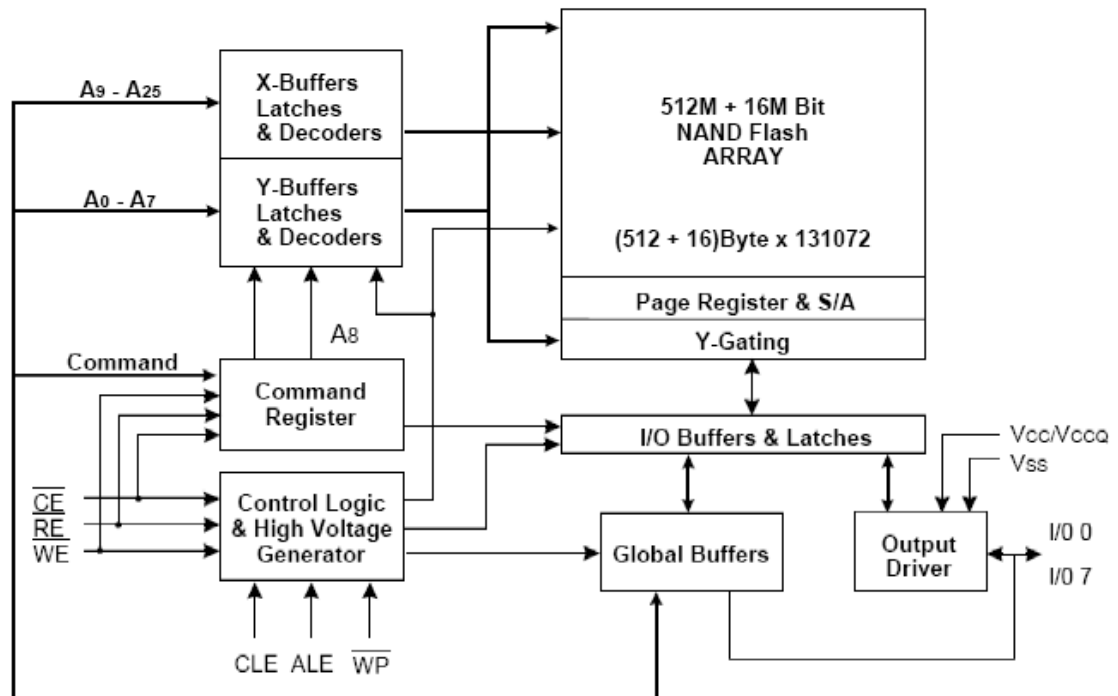


Figure 2.3: *Samsung K9F1208U0M - Internal Structure* [8]

The functions of the main blocks are summarised next.

Output Driver: This block controls the direction of the bidirectional I/O lines.

Control Logic: This block contains the state machines that control the complex sequence of command and address cycles that are needed.

High Voltage Generator: This block contains the Charge Pump which generates the high voltages needed for writing and erasing.

NAND flash Array: This block is the main memory array of NAND flash cells.

X,Y - Buffers,Latches and Decoders: These blocks control the addressing of the main memory array. Addresses are decoded in order to select the correct column (Y) and row (X).

Command Register: This block holds the value of the current command.

Page Register: This block is used to temporarily store a page of data before it is to be written to the memory array, or after it has been read from the memory array.

The Samsung K9F1208U0M main memory array is made up of 16 data cells plus 2 select cells that are serially connected to form a NAND structure. Each of the 16 data cells reside in a different page. A block consists of 2 NAND structured strings, and therefore a total of 135168 NAND cells reside in a block.

2.2.3 Operation

2.2.3.1 Writing/Programming

Programming is performed on a page by page basis which is similar to sector writing on a hard disk. Programming a page of data, as shown in Figure 2.4, requires sending the 80H command, followed by the column, page and block address. This is followed by the page of data to be stored, sent a byte at a time, which fills up the internal page register of the device. Finally, the command 10H is sent to signal the end of the data. There is then a wait period for the data to be transferred from the internal register to the main memory array. This delay, t_{PROG} , is typically $200\mu\text{s}$ to $300\mu\text{s}$ but can be a maximum of $500\mu\text{s}$ to $700\mu\text{s}$.

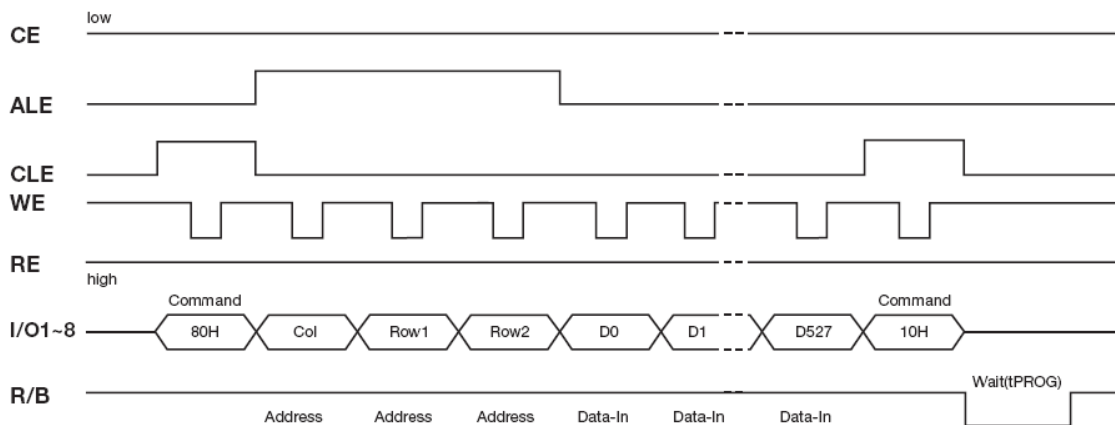


Figure 2.4: NAND - Example Program Cycle [15]

2.2.3.2 Reading

Reading a page of the device follows a similar format and is shown in Figure 2.5. The command 00H is written to the device and is then followed by the address cycles. Next there is a waiting period during which the data is transferred from the main memory

array to the internal page register. Only when this period is over can all the data be read out sequentially with a toggle of the read enable pin, RE . The read wait delay, tR , is a maximum of $12\mu s$ to $25\mu s$.

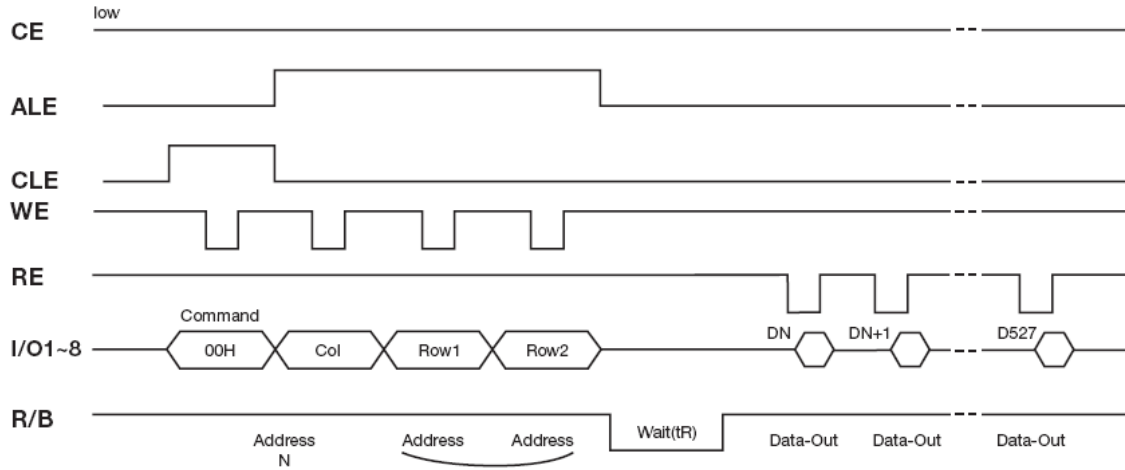


Figure 2.5: NAND - Example Read Cycle [15]

2.2.3.3 Erasing

Erasing is performed on a block by block basis. During this procedure, the device goes into a busy state while the charge pump is charges up and the actual memory cells are being erased. The erase sequence shown in Figure 2.6 is: command 60H - address cycles - command D0H - block erase wait. The wait time during which a block is being erased, $tBERS$, is typically 2 to 3ms.

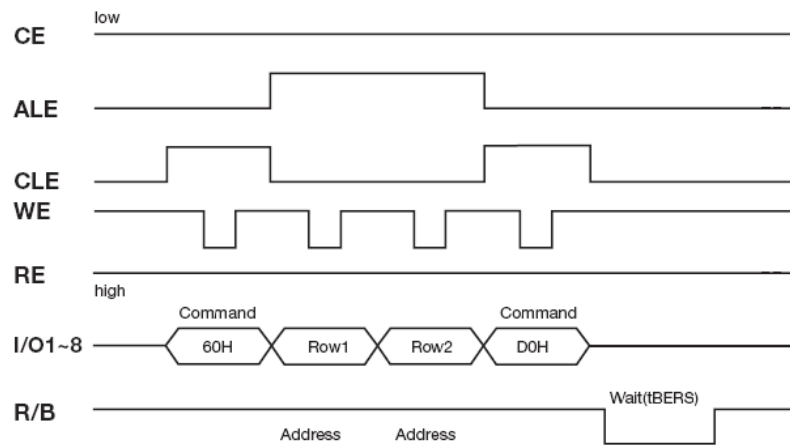


Figure 2.6: NAND - Example Erase Cycle [15]

A fundamental property of NAND flash devices is that a program operation can only

change the stored bit from a logic 1 to a logic 0. The erase operation changes the stored bit from a logic 0 to a logic 1. Therefore before a page can be programmed, the whole block in which the page resides must be erased. Figure 2.7 shows the register transfer operations during a read, write or an erase.

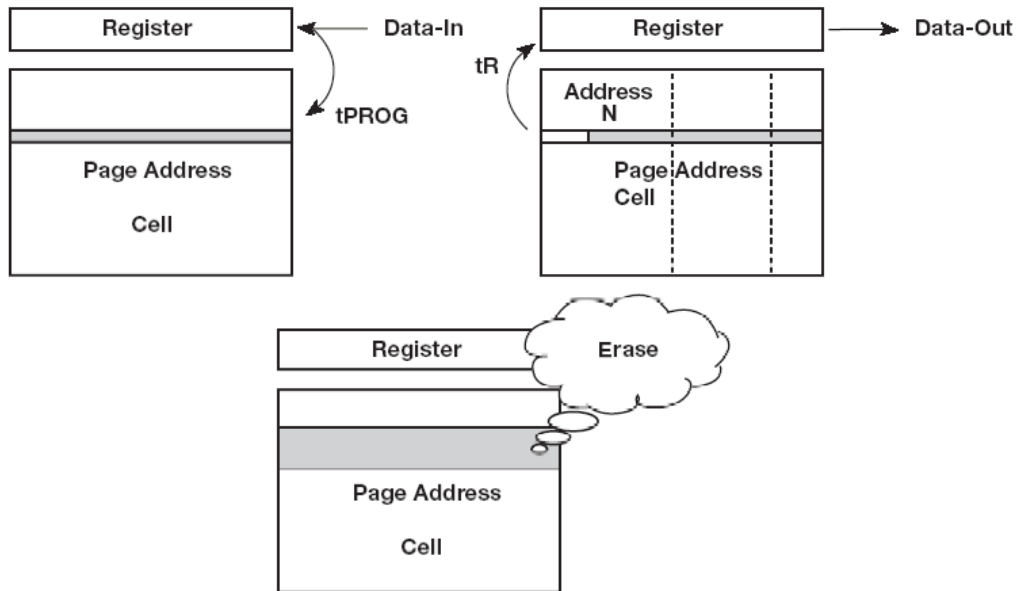


Figure 2.7: Program, Read and Erase Data Transfers [15]

2.2.4 Bad Blocks

The NAND flash device was designed to serve as a low cost solid state mass storage medium. In order to achieve this goal, the standard specification for the NAND flash device allows the existence of bad blocks in a certain percentage. These bad blocks are marked during extensive environmental and function testing at the supplier. The cause of bad blocks could be due to either decoder failure, word line failure or memory cell failure. Once a bad block is located, the supplier recommends that the block no longer be accessed. These blocks are marked by the supplier by storing 00H at byte 517 on the first or second page of a bad block, while a good block contains FFH at the same location.

NAND flash devices also do not possess infinite write/erase capability. Each block can typically be erased or reprogrammed from 100,000 to 1,000,000 times before the end of its life. The primary wear out mechanism is believed to be excess charge trapped in the oxide of a memory cell, and the net effect is that the erase times increase until an internal timer times out. This error is then reported back to the system controller through the reading of the device status register, after which the block is marked as bad. Since each block is an independent unit, it can be erased and reprogrammed without affecting the

lifetime of the other blocks.

A bad block table is therefore necessary in order to keep track of any blocks that are marked bad from the supplier, or that have been worn out over time. This will then enable the system to skip any bad blocks and to avoid any data being lost or corrupted.

2.3 Digital Design

2.3.1 VHDL

Digital electronic circuit design in recent years has become more and more dependant on computer-aided design (CAD) tools. This is primarily due to the increase in complexity, but also because these tools can help the designer design circuits extremely quickly. The name of this process is electronic design automation (EDA). The EDA tools allow two tasks to be performed: synthesis, in other words the translation of a specification into an actual implementation of the design; and simulation, in which the specification or the detailed implementation can be exercised in order to verify correct operation.

Synthesis and simulation EDA tools allow the designer to use two different design methods: schematic capture, where the designer uses a graphical interface to draw the design; or hardware descriptive languages (HDL's), where the designer uses a textual description of the design, much like a software programming language. Two of the most common HDL's that are in use today are: Verilog and VHDL. VHDL was developed by the Department of Defense and the IEEE in the 1980's, and was standardized by the IEEE in 1987 and extended in 1993.

2.3.2 Synthesis

At the time when the development of VHDL was initiated by the Department of Defense, a major concern was to have a standardised and unique method for documentation of complex digital circuits which would also allow the simulation of circuit descriptions. Based on these objectives, VHDL provided semantic elements mainly for simulation purposes, but now due to the standardisation of VHDL, it also serves as a description language of the input data to synthesis tools. The aim of the synthesis process is to generate a gate-level netlist for the target technology. The netlist can be optimised under various constraints, such as minimum area or maximum possible clock frequency. Automatic synthesis normally performs well in synthesising synchronous designs where all registers are clocked by a global clock signal, and asynchronous actions are limited to an external reset signal in order to force the circuit into a well defined state after the power

on. Synthesis tools also provide device mapping or fitting. During the mapping, timing and area information of all usable gates of the target technology must be available. The designer has also to specify optimisation constraints which the synthesis tool tries to fulfil.

2.3.3 Simulation

VHDL is a language for describing digital systems. To verify that a model is correct, a simulator is used to animate the model. Simulation can be used to verify either the behavioural model of the system or the gate-level timing model of the system. The behavioural simulation verifies the functionality of the design irrespective of any device architecture implementation. The gate-level timing simulation looks at the design when it has been compiled and fitted (place-and-root) to the intended device, and therefore propagation delays of signals through the device are also taken into account. The gate-level timing model is basically how one would expect the design to behave inside the specific physical device.

The simulator provides an invaluable verification and debugging tool for any designer. Proper verification of designs at the functional and post place-and-route stages helps to ensure design functionality and resolve any design issues quickly.

2.3.4 FPGA's

Before the advent of programmable logic, custom logic circuits were built at the board level using standard components, or at the gate level in expensive ASIC's. The FPGA is an integrated circuit that contains many (64 to over 10,000) identical logic cells that can be viewed as standard components. Each logic cell can independently take on any one of a limited set of personalities. The individual cells are interconnected by a matrix of wires and programmable switches. A user's design is implemented by specifying the simple logic function for each cell and selectively closing the switches in the interconnect matrix. The array of logic cells and interconnects form the fabric of basic building blocks for logic circuits. Complex designs are created by combining these basic blocks to create the desired circuit.

Designs can be downloaded to FPGA's multiple times with different functionality implemented each time. If a mistake is made in the design, the design can be debugged, re-compiled and downloaded again. There is no PCB, solder or components to change. The designs run much faster than a board with discrete components, since everything runs within the FPGA, on its silicon die. FPGA's lose their functionality when the power is

switched off, just like RAM in a computer that loses its content after shut down. Designs have to be downloaded again when power is switched back on to restore the functionality.

Chapter 3

Design Architectures

The aim of this chapter is to gain an understanding of the main issues at hand when designing a mass memory unit for a micro-satellite. The problems that are associated with using NAND flash devices for a mass memory unit are also addressed. After discussing each issue, ideas on how to resolve the specific problem will be given.

3.1 Design Specifications

This section looks specifically at the design specifications for a mass memory unit on a next generation micro-satellite. These specifications are irrespective of the type of memory device to be used.

3.1.1 Data Rate Calculation

The satellite imager is the main source of the data which is to be stored on the mass memory unit (MMU). The imager produces a constant stream of high speed data which is routed directly to the MMU. This stream must be stored in the MMU without the loss of any data. The MMU is controlled by the on-board computer (OBC) and routes data to the Downlink subsystem when the images are to be sent to the groundstation. Figure 3.1 shows the position of the MMU subsystem on a typical micro-satellite design.

The rate at which the data is transferred from the imager to the MMU depends firstly on the characteristics of the satellite imager sensing equipment, and secondly on the orbit characteristics of the satellite itself. The data rate can be increased by overhead such as error coding or housekeeping but it can also be decreased by compression.

For this thesis a data rate was not defined specifically, but looking at currently developed imager parameters, an estimate for the data rate can be calculated which is in line with present day technology. The following parameters are assumed:

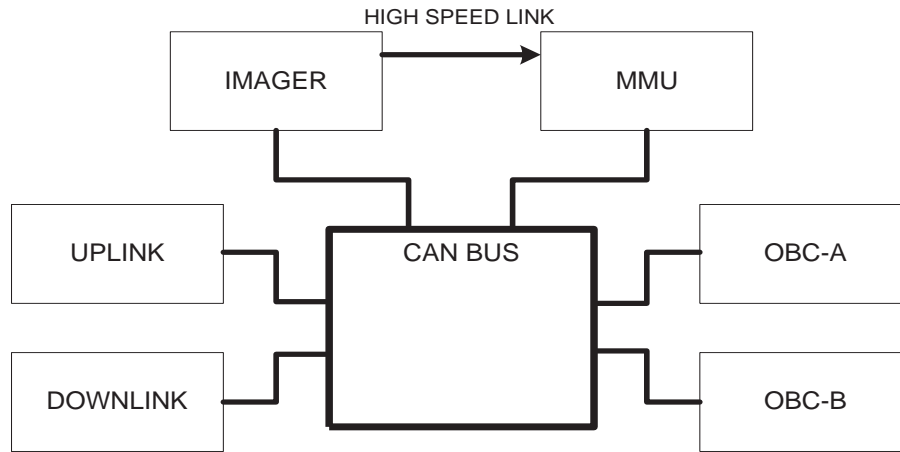


Figure 3.1: *Satellite Subsystem Configuration*

- CCD elements = 12000
- Square pixels with resolution = 5m × 5m
- Square image = 12000 × 5m = 6km
- Number of bits to encode each sample = 8 bits
- Length of Image = 480km

To work out the data rate from these parameters, firstly the ground speed of the satellite must be calculated. For a micro-satellite in a circular orbit at a height above the earth, $h = 700\text{km}$ (typical for a LEO satellite), the period is:

$$P = 2\pi\sqrt{\frac{(R_E + h)^3}{\mu}} = 98.8 \text{ minutes} \quad (3.1)$$

where the radius of the earth, $R_E = 6378 \text{ km}$ and earth's gravitational constant, $\mu = GM = 398600.5 \text{ km}^3/\text{s}^{-2}$.

The ground speed is then simply the circumference of the earth divided by the period:

$$V_g = \frac{2\pi R_E}{P} = 6.76 \text{ km/s} \quad (3.2)$$

If the pixel width is 5m then the sample rate, SR, of one pixel must be:

$$SR_{pixel} = \frac{V_g}{5} = 1352 \text{ samples/s} \quad (3.3)$$

There are 12000 pixels in the CCD array, which means the sample rate for one line is:

$$SR_{line} = SR_{pixel} \times 12000 = 16224000 \text{ samples/s} \quad (3.4)$$

Finally, one can work out the data rate, given the number of bits used to quantify the intensity of each pixel, b , which then gives 2^b amplitude levels. The estimated required data rate is then:

$$\text{Data Rate} = SR_{line} \times 8 = 15.47 \text{ MB/s} \quad (3.5)$$

A data rate of 15.47 MB/s will now be used for the rest of this thesis as a target write specification for the MMU.

3.1.2 Mass Memory Capacity

The capacity of the MMU depends on the size of the images to be stored and also on how many of these images have to be stored at one time. If the images are to be stored without any data compression, then one image can easily exceed 1 GB. As the images can only be downloaded to the ground station when the satellite is in range, the ability to store multiple images is a must.

Using the parameters assumed in Section 3.1.1, the size of a 480 km image can be calculated. The first two parameters to be found are: the size of one line and the total number of lines in the image:

$$\text{One line} = \text{number of pixels} \times b = 12000 \times 8 = 96000 \text{ bits} \quad (3.6)$$

$$\text{Number of lines} = \frac{\text{Image length}}{\text{Pixel Width}} = \frac{480 \times 10^3}{5} = 96000 \text{ lines} \quad (3.7)$$

Now one can determine the size of one image:

$$\text{Size}_{image} = \text{one line} \times \text{number of lines} = 1.073 \text{ GB} \quad (3.8)$$

Seeing that the MMU must be able to store multiple images in order to be efficient, a minimum requirement should be that the MMU have a capacity of 5.365 GB. This would make it possible to store at least five images at any given time.

3.1.3 Cost

As the research conducted by the ESL laboratory in micro-satellite design is funded mainly by Sunspace¹ and sponsorship from commercial companies, the budget therefore available is not endless. Minimising costs is therefore essential for the ongoing research to continue.

Each of the key design decisions including major component selection, development software selection and number of PCB layers, were made with cost minimisation in mind.

¹SunSpace was established in 2000, through the Unistel Group and the Office of Intellectual Property of the University of Stellenbosch.

3.1.4 Physical Size and Mass

The physical size of the MMU is dependent on the required capacity, the control circuitry and any redundancy that may be required. If the satellite is designed in a modular fashion, so that each subsystem fits into a predefined section or tray, then the dimensions of that section will drive the physical size requirements.

One of the main requirements in designing any subsystem on a micro-satellite is to minimise its size and mass wherever possible. The mass of the satellite is one of the major factors involved in the escalating costs when building and launching a spacecraft. As the staggering cost of a launch is directly proportional to the mass of the spacecraft, one can never be too conservative when it comes to the weight of the on-board components.

3.1.5 Power

Micro-satellites rely on rechargeable batteries and solar power to provide power to all the subsystems on board. Solar power is used when the satellite's solar panels are in view of the sun, and the batteries are used when the satellite is in the earth's shadow and is hidden from the sun. It is then obvious that the MMU must be as efficient as possible in order not to drain the power resources excessively. Excessive recharging of the batteries seriously affect their lifetime, and therefore also the lifetime of the satellite.

The power specification for the Sunsat-1 micro-satellite was not to exceed 5W. Due to advances in technology and the increased availability of 3.3 V devices, the specification for this thesis will be to keep the power below 2W.

A NAND flash is a non-volatile type of memory. This means that when the power is switched off the data is retained. This provides a massive saving on power, as power only needs to be supplied when the devices are reading, erasing or writing, and can be switched off when the unit is idle.

3.1.6 Radiation Problems

The problem with designing any subsystem of a satellite with commercial off the shelf components is that they are far more susceptible to space radiation than radiation hardened devices that are manufactured specifically for space missions. All digital devices are affected by radiation and NAND flash devices are no exception.

Radiation tolerance is a huge issue when discussing any components inclusion on a spacecraft. It was decided that this thesis will not concentrate on radiation issues but rather

on the architecture and operation of the MMU. The radiation problems associated with NAND flash memory will therefore be used as a topic for another Masters thesis.

3.1.7 Complexity

One of the design aims that should always be kept in mind is simplicity of maintenance and upgrading for future users, in spite of underlying functional complexity. This can be achieved by splitting a system up in a modular format so that each module has its own functional characteristics that can be tested or simulated on its own. This helps reduce complexity and makes fault finding a less laborious task.

It was decided that VHDL would be used for data control and also to create an interface to the NAND flash devices. Following the discussions in the previous paragraph, it was decided to implement the VHDL design in a modular fashion, so that it can be reused or expanded for further NAND flash research. Fortunately, this will not be a major issue, as VHDL is a language which supports a modular approach extremely well.

3.1.8 Reliability

Designing and launching a micro-satellite of any kind is a high risk business. If an important subsystem fails, there is very little that can be done to fix it up in space. That is why each subsystem must be highly reliable, to the extent that if something does go wrong the system can correct itself or still work with reduced functionality. One method to achieve this is to include redundancy, in the form of extra circuitry or devices in the design.

The MMU on-board a micro-satellite is one of many critically important subsystems. A non-operational MMU basically reduces the micro-satellite to a system which is only able to take images when in range of the ground station, as no data can be stored on-board. The aim is therefore to have a memory system with multiple memory devices which would still be able to function to a certain extent if one or more of the memory devices fail.

3.2 NAND Flash Issues

This section looks at the problems associated with using NAND flash devices for a MMU.

3.2.1 Data Rate

A NAND flash device has a minimum write cycle time of 50 ns [8], which means that it can sustain a maximum data rate of approximately 19 MB/s. It seems at first glance

that designing the MMU is as simple as filling up each chip one by one until all the data is stored, as the required data rate of 15.47 MB/s is easily met. This is however not the case. Only one page of data can be stored on the device before it then goes into a busy state. This can last for up to $700\mu\text{s}$, and only once it returns to a ready state can the device then be accessed again. The data rate would therefore be slowed down to about 1 MB/s if the sequential method is still used. The only way to sustain the high data rate during this busy period is to write to another device until it also goes in a busy state. This process of writing sequentially to multiple devices must therefore continue until the first device has finished transferring data from its internal register to the main memory array.

Figure 3.2 shows an example of a system with four NAND flash devices. Initially setup cycles are written to device 1 and then the data stream starts writing the first 528 bytes to this device. Once the page buffer of device 1 is full the device cannot be accessed again until after time, t_{PROG} , has elapsed. The data stream is then sustained by writing to device 2, 3 and 4 instead. In order for the data stream to flow, the setup cycles for these devices must be executed before the data stream arrives. Once device 1 is ready, the data stream can be routed to this device again. The result is a continuous data stream stored in memory without the loss of any information.

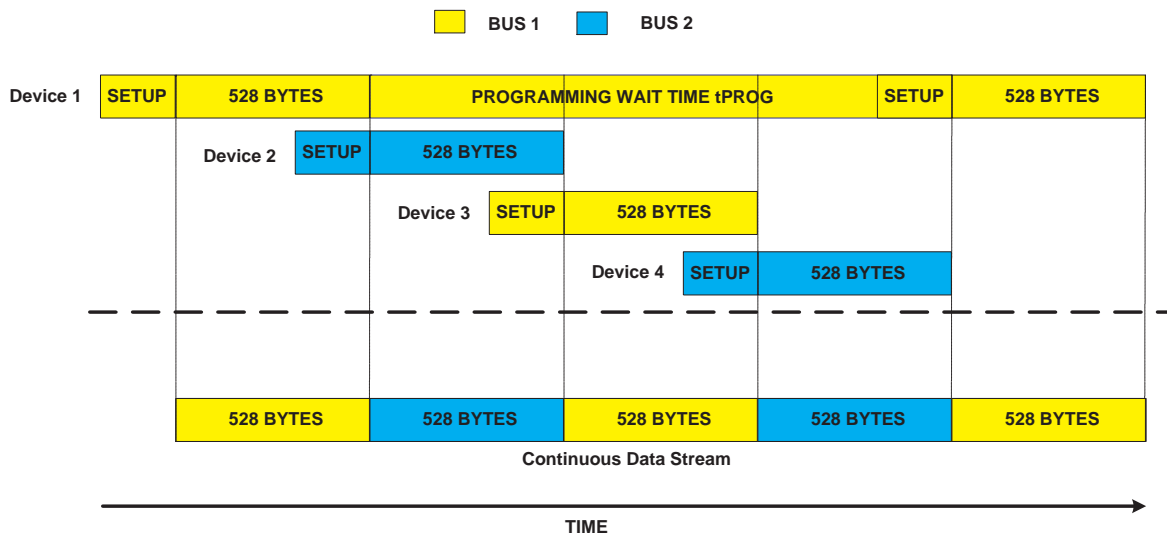


Figure 3.2: *Programming with Multiple Devices*

The minimum number of devices needed in the system is therefore enough devices to ensure the data rate is sustained before the first device accessed is ready to be accessed again. The data rate determines the amount of time it takes to write one page of data to a device. The number of devices needed in the system is therefore not only a function of the capacity, but also of the data rate.

3.2.2 Bad Blocks

As mentioned in Section 2.2.4, bad blocks have to be tracked in a system with NAND flash memory devices. Initial bad blocks found by the supplier are marked when the device is shipped. These blocks have to be read and stored when the device is received by the designer. Bad blocks can also occur during any erase or write cycle and are detected by reading the status register of the device once the erase or write operation has been completed. When a block is found to be bad, the integrity of the data stored is not assured, and the block must be mapped out. This causes the following problem during the write sequence: if a block is found to be bad once the device has finished storing data into that block, there is no time to restore the data elsewhere, because the data continues to stream through at a constant rate. Erasing is different as there are no specific time constraints or data rates to sustain, and therefore bad blocks can easily be mapped out once they are encountered. Reading data from bad blocks is not a problem, but erasing or programming invalid blocks can affect adjacent blocks in some rare cases.

3.2.3 Failure Modes Mechanisms and Symptoms

With NAND flash devices random bit errors can occur during use. The consequence of these errors do not necessarily mean that a block has gone bad, but may be caused by a random bit flip. Generally, a block should only be marked as bad if there is a program or erase failure. The two main failure types can be distinguished as *permanent failures* and *soft failures*.

3.2.3.1 Permanent failures

A permanent error is an error which cannot be corrected or fixed. The most common permanent errors are caused by endurance stress when programming or erasing (see Figure 3.3). This error may be manifested as a cell, page, or block failure which can be detected by a status register read operation after either a program or an erase, and subsequently marked as a bad block.

3.2.3.2 Soft Errors

A soft error is a an error which can be corrected and only occurs randomly. These soft errors manifest themselves as either a bit reversed or a bit reported reversed. This phenomenon is known as “bit-flipping” and is the result of one of the following errors:

- Drifting Effects: A phenomena that slowly changes a cell’s voltage level from its initial value.

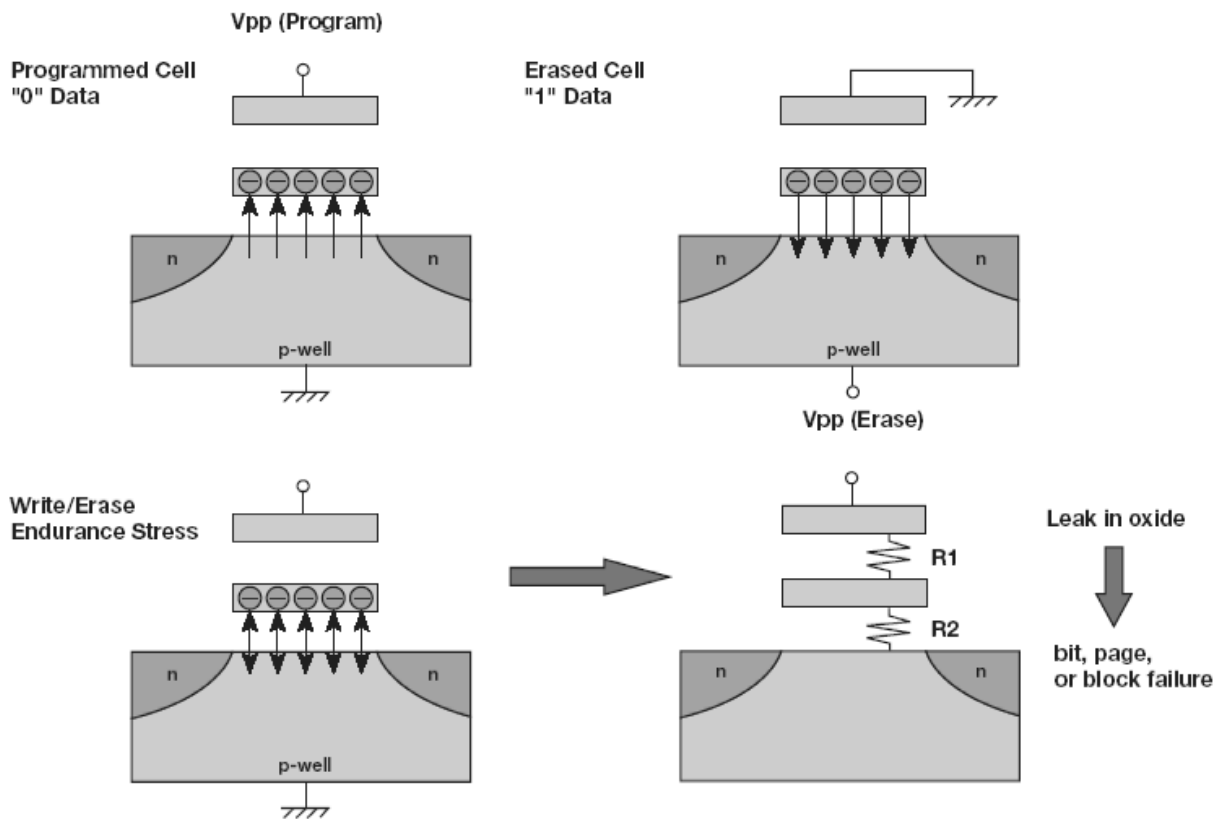


Figure 3.3: *Program and Erase Endurance [15]*

- Program-Disturb Errors: This is sometimes referred to as “over programming” effects. A programming operation on a page induces the flip of a bit on another, unrelated page. Bias voltage conditions in the block during page programming can cause a small amount of current to tunnel into memory cells which cause this error.
- Read-Disturb Errors: This effect causes a page read operation to induce a permanent change of value in one of the bits read.

Since NAND flash devices are susceptible to bit-flipping, an error checking and correcting (ECC) implementation is required to maintain the integrity of the data stored. A number of issues must be discussed before ECC can be included in the system. The first issue is that the ECC code word generated from a block of data has to be stored with the data, thereby adding extra overhead. The second issue is that when the batch of data is read back and another code word generated and compared to the original, the block of data must be temporarily stored before any corrections can take place. This temporary storage may be in the form of a FIFO and will use valuable FPGA resources.

3.3 Design Alternatives

3.3.1 Page Size - 528 or 2112 Bytes?

The currently available NAND flash devices have two different page sizes. The 528 byte page is associated with devices with total size under 1 Gb, while the recently developed devices bigger than 1 Gb (inclusive) are available with a page size of 2112 bytes. The differences between read and write speeds in a 512 Mb device and a 1 Gb device are shown next. See Section 2.2.3 for operation details.

512 Mbit device

$$\text{Read Time} = 5 \text{ cycles} \times 50\text{ns} + 12\mu\text{s} + 528 \text{ cycles} \times 50\text{ns} = 36.55 \mu\text{s} \quad (3.9)$$

$$\text{Read Speed} = \frac{528 \text{ bytes}}{36.55 \mu\text{s}} = 13.78 \text{ MB/s} \quad (3.10)$$

$$\text{Write Time} = 6 \text{ cycles} \times 50\text{ns} + 528 \text{ cycles} \times 50\text{ns} + 500\mu\text{s} = 526.70 \mu\text{s} \quad (3.11)$$

$$\text{Write Speed} = \frac{528 \text{ bytes}}{526.70 \mu\text{s}} = 0.96 \text{ MB/s} \quad (3.12)$$

1 Gbit device

$$\text{Read Time} = 6 \text{ cycles} \times 50\text{ns} + 25\mu\text{s} + 2112 \text{ cycles} \times 50\text{ns} = 130.90 \mu\text{s} \quad (3.13)$$

$$\text{Read Speed} = \frac{2112 \text{ bytes}}{130.90 \mu\text{s}} = 15.39 \text{ MB/s} \quad (3.14)$$

$$\text{Write Time} = 6 \text{ cycles} \times 50\text{ns} + 2112 \text{ cycles} \times 50\text{ns} + 700\mu\text{s} = 805.90 \mu\text{s} \quad (3.15)$$

$$\text{Write Speed} = \frac{2112 \text{ bytes}}{805.90 \mu\text{s}} = 2.50 \text{ MB/s} \quad (3.16)$$

It can be seen from equations 3.12 and 3.16 that the write speed is over 2 times faster on a device with a 2112 byte page than that of a device with a 528 byte page. The read speed is also quicker but not as dramatically as the write speed.

3.3.2 I/O Lines - 8 or 16 bit?

As well as having different page sizes, NAND flash chips also have two options for width of the I/O lines. Devices can either have 8 or 16 bits for these lines. With the 16 bit device, the commands and addresses are multiplexed on the first eight I/O lines and the other eight bits are simply ignored. The obvious advantage with the 16-bit device is that the data rate (ignoring setup cycles) can be doubled immediately, because 16 bits can now be stored every 50 ns instead of just 8 bits. The only downside to the 16-bit device

is that the internal page register is still the same size and writing to the device at double the speed of a 8-bit device will simply fill the internal register up twice as fast. This makes the time until the device can be accessed again longer and may require the system to have more flash devices to help sustain the data rate.

3.3.3 Multiple Buses

As mentioned in Section 2.2.1, the bidirectional I/O lines of the NAND flash device are multiplexed for command, address and data. That means that a device must be set up before data can be written to or read from it. To sustain the high data rate, there must more than one set of control and data lines, to enable the next device to be setup ahead of the last device finishing (see Figure 3.2). For ease of explanation, as set of control and data lines will from now be referred to as a bus.

The number of buses that can be used in the design is basically restricted to the number of available I/O pins on the FPGA used to control the lines, and the amount of available space on the final PCB. The number of FPGA pins needed for 8-bit and 16-bit data I/O lines for a 2, 3, 4 and 5 bus structure is shown in Table 3.1.

| <i>Number of Buses</i> | <i>Number of FPGA Pins</i> | |
|------------------------|----------------------------|----------------------|
| | <i>8 bit device</i> | <i>16 bit device</i> |
| 2 | 30 | 46 |
| 3 | 45 | 92 |
| 4 | 60 | 115 |
| 5 | 75 | 138 |
| 6 | 90 | 161 |

Table 3.1: *Number of FPGA pins needed for multiple buses*

If the design can include more buses than the number required to meet the data rate requirements, these extra buses can give added redundancy and reliability. If one bus is diagnosed as having a serious fault, it can simply be skipped as there will be enough buses still left to maintain the data rate.

The down side to multiple buses is that the routing and the placement of these buses on the physical PCB becomes more complex with every bus that is added. Along with the complexity, the PCB size also increases, as does the cost of producing it. The VHDL design to control these buses will become bigger and lead to more logic elements in the FPGA being used, therefore requiring a larger FPGA.

3.3.4 Serial or Parallel?

3.3.4.1 Serial

Multiple NAND flash devices can be connected in a serial configuration with common control signals and a common wired I/O bus (see Figure 3.4). In this configuration the control signals, CLE, ALE, WE and RE are all shared by the NAND flash devices, and this minimises the pin count in the system's design.

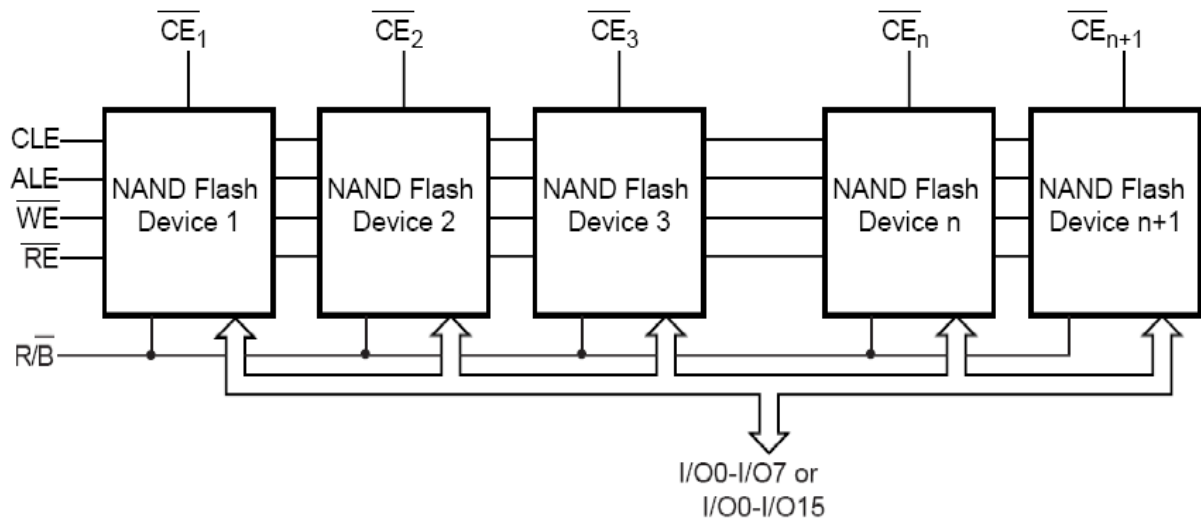


Figure 3.4: *NAND Flash Serial [11]*

The chip enable (\overline{CE}) signals are wired separately for each device. When \overline{CE} is low, the corresponding device is selected. Only one NAND flash device can be selected at any one time and only the selected device can see the control signals and the system bus. The ready/busy ($\overline{R/B}$) output lines of each memory device can be common wired. The use of an open-drain output allows the $\overline{R/B}$ pins from several NAND flash devices to be connected to a single pull-up resistor. A low on this line will indicate that one or more of the devices are busy. There are two possibilities for the wiring of the write protect pin (\overline{WP}), the first is to common wire them all so that all the devices can be protected or unprotected together, and the second is to wire them separately. The second solution for \overline{WP} requires more pins in the system design.

Although the serial configuration offers reduced pin counts, the maximum sustainable data rate is limited to that which one device can sustain, as only one byte can be written at any one time.

3.3.4.2 Parallel

Writing to two or more devices, instead of writing to just one device at a time, can dramatically increase the maximum sustainable data rate of a system. This can be achieved by connecting NAND flash devices in parallel to create a module with a higher bus width. The control signals can then be common wired and the I/O lines connected in parallel to the system bus. With such a configuration, the system bus width must therefore be a multiple of the bus width of a single NAND flash device. The parallel implementation of NAND flash devices is illustrated in Figure 3.5.

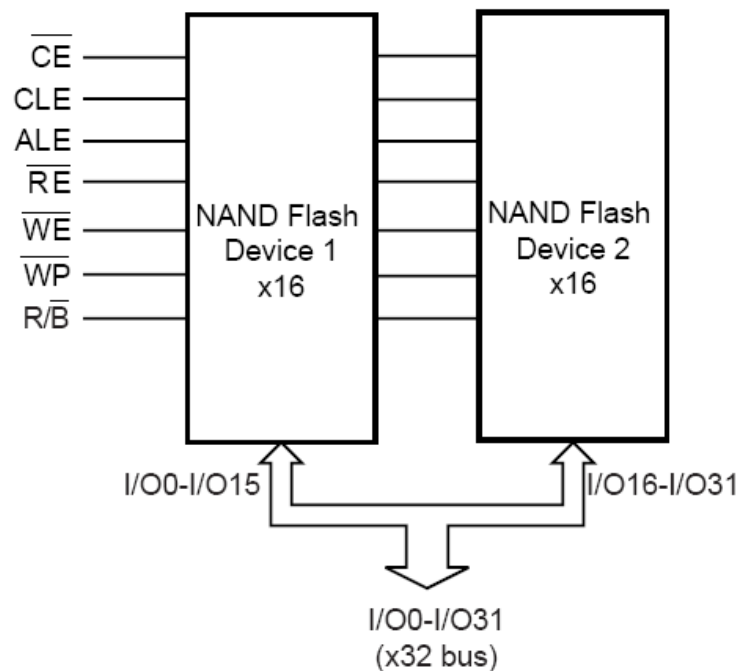


Figure 3.5: *NAND Flash Parallel* [11]

One of the disadvantages of writing to more than one chip at a time is a decrease in reliability. If one bus fails, it can eliminate other buses in parallel with it as well. Also, the blocks with the same address on each device can be treated as a single, larger, block to be programmed, erased, and read together. This would lead to a dramatic increase in the complexity of the bad block management, as it is possible that a block on the first device can turn bad while the corresponding block on the second device is still valid.

3.3.5 Number of Devices

3.3.5.1 Data Rate Constraints

The number of NAND flash devices needed in the mass-memory unit is driven not only by the required capacity, but also by the required data rate and the system architecture

(8/16 bit, serial/parallel). As explain in Section 2.2.3, this is because the system cannot revisit the device which has just been programmed until after the programming wait time, t_{PROG} , has elapsed.

Considering the Samsung K9F1208U0M device and ignoring setup times, writing 528 bytes of data at a 50 ns write cycle time will take the following time:

$$\text{Write Time (without setup cycles)} = 50 \text{ ns} \times 528 \text{ bytes} = 26.4 \mu\text{s} \quad (3.17)$$

The device then goes into a busy state for up to 500 μs , which means that the a certain number of devices must exist to maintain the data rate before the first device can be written to again. The number of devices needed can be calculated by dividing the programming delay, t_{PROG} , by the time it takes to write a page. This gives the number of times that 528 bytes need to be written in order to pass a period of t_{PROG} . The minimum number of devices then needed is just this number plus one for the device which is busy.

$$\text{Minimum Number of NAND Devices} = 1 + \frac{500 \mu\text{s}}{26.4 \mu\text{s}} = 20 \text{ Devices} \quad (3.18)$$

3.3.5.2 Area Constraints

Each NAND Flash device takes up an area of roughly 184.8 mm². The final PCB will also have to include an FPGA, oscillator, voltage regulators and interface ports. The area the devices must be placed will be restricted to the amount of area left after all the other components have been placed. The intension is that the MMU will fit into an Eurocard U6 sized tray of dimensions 233 mm \times 160 mm.

3.3.5.3 Power Constraints

The more devices that are operational at one time, the higher the power that is being consumed. A Samsung K9F1208U0M device draws a maximum of 20mA current during an erase, write or read operation and operates with a 3.3V supply. The maximum power consumed by a device operation is then given by:

$$\text{Power (Samsung K9F1208U0M)} = 20 \text{ mA} \times 3.3 \text{ V} = 66 \text{ mW} \quad (3.19)$$

Following this, power consumed when n devices needed to sustain the data rate are all operational at the same time is :

$$\text{Power (Samsung K9F1208U0M)} = n \times 66 \text{ mW} \quad (3.20)$$

3.3.5.4 Device Input Capacitance

The number of NAND flash devices that are connected to one bus affects the rise and fall times of the driver output, and hence the maximum frequency that the system can achieve. This is due to the input capacitance of each device. The driver sees the sum of all input capacitances connected to the driver as they are connected in parallel. This creates an RC effect on the line. The output impedance of the driver plus the impedance of the PCB trace equals the total impedance, and the sum of all input capacitances plus the capacitance of the PCB trace is the total capacitance. The time constant and rise time (10% - 90%) for the line when there are n devices is calculated as follows:

$$\tau = (R_{trace} + R_{driver_output}) \times (n \times (C_{input_capacitance} + C_{trace})) \quad (3.21)$$

$$T_{rise_time} = 2.2 \times \tau \quad (3.22)$$

It can be seen from equations 3.21 and 3.22 that the signal rise time will increase significantly with every device added to the line.

The data sheet for the Samsung K9F1208U0M device [8] states rise and fall times of 5 ns for a capacitive loading of 100pF. Each K9F1208U0M device has an input capacitance of 10pF therefore if 10 devices are connected to one common driver, the rise and fall times would be equal to 5 ns.

3.3.6 Bad Block Table

A lookup table is needed to keep track of the bad blocks contained on individual NAND flash devices to prevent access to these blocks. Such a table can be stored in a known good block on one of the devices and can then be referenced during a read, write or erase sequence to avoid the use of a bad block.

There are various ways in which to implement this. One table can be used for all the devices or separate tables can be implemented to store individual bad blocks for each bus. Additionally, one can either store the whole address of the block or use a bit to signify the status of every block on the device. The amount of internal FPGA RAM required using the one bit storage method for a Samsung K9F1208U0M flash with n number of devices on the unit is simply:

$$\text{Internal FPGA RAM required} = n \times 4096, \quad (3.23)$$

as there are 4096 blocks on each device.

The amount of FPGA RAM required if only the addresses of the bad blocks are stored, with each block address equal to 12 bits, and m number of locations are set aside to store these addresses is:

$$\text{Internal FPGA RAM required} = m \times n \times 12 \quad (3.24)$$

In order to decided which method to choose for the bad block table implementation, the number of devices, the number of expected bad blocks and the amount of FPGA RAM likely to be available must be known.

3.3.7 Cache

Some NAND flash devices contain an extra internal register called a Cache Register, which can be used to improve the programming throughput. In the standard Page Programming operation, the device has to finish programming the data into the memory array before a new page can be programmed. The advantage of the Cache Programming operation is that it allows new data to be input while previous data which has been transferred to the Page Buffer is programmed into the memory array. The Cache Program sequence can be repeated up to 31 times, allowing all 32 pages in a block to be programmed. One entire Cache Program operation is shown in Figure 3.6.

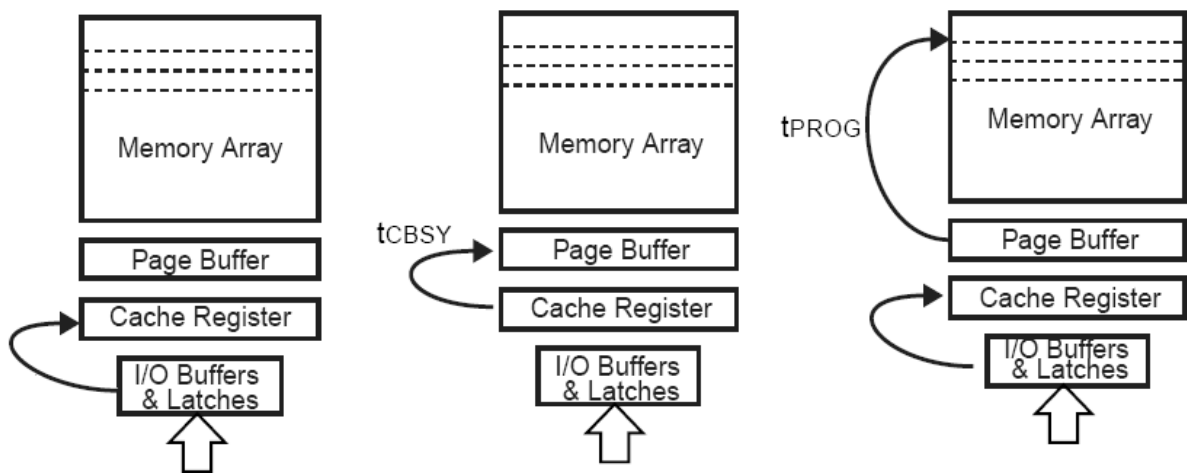


Figure 3.6: NAND Flash Cache Program Operation [12]

The Cache busy time, t_{CBSY} , is the time it takes to transfer the contents of the Cache Register to the Page Buffer and is typically $3\mu s$ when the Page Buffer is empty. The Cache busy time can however take up to $700\mu s$, which is the same time as t_{PROG} , if the Page Buffer is still transferring data into the main memory array.

Although the use of the Cache register may increase the speed of programming the flash device, the cache busy time will increase with every page that is written and will eventually become equal to tPROG. This does not really help for a sustained data rate, as the device revisit time will change depending on the number of pages in a block that have been written and will make the device access algorithm extremely complicated.

3.3.8 Reference Tables

Table A.1 found in appendix A shows the maximum sustainable data rate with a 60 ns write cycle time for all the different design architectures described and also shows the minimum number of NAND flash devices the MMU must include for various data rates. Excluding the maximum achievable data rate, it can be seen from the table that there is no difference in the number of devices needed for a device with either 8-bit or 16-bit I/O lines. The 16-bit devices can sustain a higher data rate but require nearly double the amount of devices to do so. The devices with a 2112 byte page register are far more efficient than their 528 byte equivalents, as the minimum number of 2112 byte devices needed are less than half the required amount for the same bus configuration and I/O lines.

Table A.2 shows the power and area calculations as well as the number of FPGA pins needed to interface to the minimum number of NAND flash devices for the specific data rate. From the table it can be seen that the 2112 byte page register devices offer a massive saving in both power and area. The 8-bit and 16-bit devices consume the same amount of power and board area, but the 16-bit devices require more FPGA pins to interface to the devices.

3.4 FPGA Considerations

To select the appropriate FPGA for a design the following aspects must be considered: internal clock speed, number of logic elements, internal RAM, user I/O pins, vendor tools, package and cost. Each of these issues will be addressed presently.

Internal Clock Speed

Almost all digital designs have some concept of state built into them. The present state of the system can either be updated as soon as the next state changes, in which case the system is said to be *asynchronous*, or the present state can be updated only when a clock signal changes, which is *synchronous* behaviour. For a project like this, where data is being transferred at high data rates and the interface timing to external devices is of critical importance, a synchronous design must be used.

The internal clock speed of an FPGA determines the speed at which flip-flops can be triggered and therefore the tempo of state changes. The speed which is given in the data sheets by FPGA suppliers often refers to the maximum possible internal clock speed, but in reality this speed can vary depending on the types of component used in the designer's code and the design efficiency. Most FPGA vendor compiler tools inform the designer of the maximum possible clock speed after the design has been compiled and fitted to a specific device.

Number of Logic Elements

A logic element, also known as a logic cell or a logic module, is one of the units used to determine the density of an FPGA. The greater the number of logic elements in an FPGA, the larger the design that can be programmed into it. A logic element consists typically of a flip-flop, a four-input look-up table and carry logic. The flip-flop can be configured for D, T, SR or JK operation, or bypassed entirely for pure combinational logic. The look-up table is basically a truth table used to define the function of the element. The carry logic is used if the cell is part of a carry chain where the output of one logic cell is the input to the next.

Internal RAM

FPGA's can also contain a certain amount of internal RAM available for use by the designer. If a design requires a lot of data to be stored temporarily, i.e. a reference table for bad blocks, then a sufficient amount of internal RAM is required.

User I/O Pins

FPGA's have a restricted amount of I/O pins available to the user. The designer must have an idea of how many I/O pins the design needs and select the appropriate FPGA and package.

Vendor Tools Each individual FPGA vendor supplies software tools with which the designer can use to develop. The designer is therefore restricted to using that specific vendor's tools for their design implementation. Access to or familiarisation with certain development tools can sway the designer to a particular FPGA vendor.

Package

The package within which the FPGA is housed is also an important aspect of FPGA selection. The two main types of FPGA package are the ball grid array (BGA) and quad flat pack (QFP). The BGA package's connecting balls are positioned on the bottom of the

device, while the QFP is more conventional, with pins connected to the sides. The BGA package can populate a greater amount of pins, but requires a complicated and expensive method of attaching it to the board. It is also difficult to debug for any connection problems because the majority of the pins are hidden underneath the device. The QFP package can be attached by a skilled technician to the PCB very easily, and every pin is visible for the finding of any short circuits or dry solder joints.

Cost

The final consideration when selecting an FPGA is the cost. The cost of FPGA's vary with the functionality and performance that they provide. The major FPGA suppliers, Xilinx, Altera, Lattice and Actel, all produce low-cost FPGA's with high capacities.

Table 3.2 compares the various low cost FPGA families of the leading manufacturers.

Table 3.2: *Comparison of available low-cost FPGAs*

| Manufacturer | Name | Capacity | RAM (kBytes) | User I/O Lines |
|--------------|-------------------------|--------------------|--------------|----------------|
| Actel | ProASIC ^{PLUS} | 3072-56320 TILES | 27-198 | 158-712 |
| Altera | Cyclone | 2910-20060 LCs | 58-288 | 65-301 |
| Lattice | LatticeEC | 192-5120 PFUs/PFFs | 6-164 | 67-576 |
| Xilinx | Spartan IIE | 1728-15552 LEs | 24-216 | 60-514 |

Chapter 4

System Design Overview

This chapter presents an overview of the design of a demonstration MMU. The first section looks at the decisions supporting the design concept development. The second section structures the design by dividing it into individual components. It looks in detail at the proposed functionality of these components and how they will integrate with each other.

4.1 Design Decisions

A number of key decisions had to be made at this point before the main design could be developed. The main issues were:

- What type of NAND flash device will be used?
- How many NAND flash devices are needed in the system?
- How many buses are required to interface with the NAND flash devices?
- Serial or Parallel access to the NAND devices?
- How will the bad block table be implemented?
- How will the memory be structured?
- What FPGA and VHDL development software will be used?

4.1.1 NAND Device Flavour

NAND flash devices come in many different flavours. The main differences are the packaging, amount of I/O lines and page size. The following decisions were made to choose the most appropriate device for a mass memory application:

Package

The two types of package that NAND flash devices are supplied in are TSOP1 and TBGA. The TSOP1 was selected, because it offers a much simpler method of attaching the device to the board and debugging is much simpler when the pins are in view.

I/O lines

The 16-bit devices were selected because they offer a much higher data rate capability compared to that of an 8-bit device. The 16-bit devices, when writing at slower speeds than their maximum, offer greater safety margins for data setup and hold times. If the 16-bit devices should write at the same data rate as the 8-bit counterpart, the write cycle time would be doubled. This would provide a greater safety margin and the added bonus that the page register would still be filled up at the same rate.

Page Size

The choice of which page size to select is an easy one. The high density NAND flash devices (over 1 Gb) all have a page size of 2112 bytes. By using higher density devices, less devices are needed to meet the required capacity of the MMU. Table A.1 shows that the 2112 bytes per page devices offer a much better option than their 528 counterparts, as they require fewer devices to sustain the data rate. One 2112 byte device can write four times as much data as a 528 byte device before it goes into a busy state.

Capacity

The specification calculated in Section 3.1.2 for the capacity of the MMU was 5.265 GB. By using available capacities, this would require 86×64 MB, 43×128 MB, 22×256 MB, or 11×512 MB devices. As the 512 MB devices offer the smallest amount of area to be covered on the board, they were the device capacity selected for this design.

4.1.2 Bus Access

As explained in Section 3.3.3, it is necessary for the design to include at least two buses in order to maintain the high data rate. Table A.1 in Appendix A shows that a two-bus architecture with 16-bit devices and serial access can sustain a maximum data rate of almost 32 MB/s, which is more than double the proposed specification of 15.47 MB/s (equation 3.5). A two-bus architecture with parallel access can sustain a maximum data rate of nearly 64 MB/s, but this comes at a cost of needing 26 NAND flash devices to do so. Parallel access offers less reliability, because if one bus should fail, the data rate would suffer dramatically. Consider this example: If one has a multiple bus structure and writes

to two buses in parallel, then the event of one bus becoming faulty would cause the loss of the other working bus with which it is paired, as both buses are needed to sustain the data rate. On the other hand, if one bus becomes faulty in a multiple bus serial access structure, the data rate is would not be affected, as long as two buses are still working. The capacity of the memory would then only be reduced by one bus.

Since serial access offers more reliability and can easily sustain the required data rate, it was decided that the MMU would be implemented this way.

4.1.3 Number of Buses

For the MMU to have active redundancy, the number of buses must be more than the minimum required amount. If two buses is the minimum requirement then the maximum limit to the number of buses is the available user I/O pins in the selected FPGA. Using 16-bit devices, of which the pin counts are shown in Table 3.1, it can be seen that a four-bus architecture requires 115 user I/O pins. If the pins required for debugging, such as leds and test points, and external interfacing ports are added to this number then it could easily approach 200 pins. The greater the number of pins, the more area the FPGA will require and the more complex the routing and placing will be for the final PCB. It was therefore decided to limit the number of buses to four, in order for the physical design to be practical.

A four-bus structure with serial access offers reliability and active redundancy, as two buses can be faulty while still sustaining the data rate. Partitioning the physical hardware design into four buses also helps minimise track length and capacitance. The length of individual tracks on a PCB must be minimised in order to avoid transmission line effects. The amount of devices on one bus must also be restricted as the accumulation of input capacitances affect signal transition times.

4.1.4 Number of Devices

Without redundancy, two buses and seven devices are needed to sustain the data rate of 15 MB/s. The number of devices was changed to eight, in order to have four devices on each bus. As discussed in Section 4.1.3, four buses have to be used to implement active redundancy. This would lead to a total of 16×512 MB Samsung K9K4G16U0M devices, equal to a capacity of 8 GB. This configuration is shown in Figure 4.1.

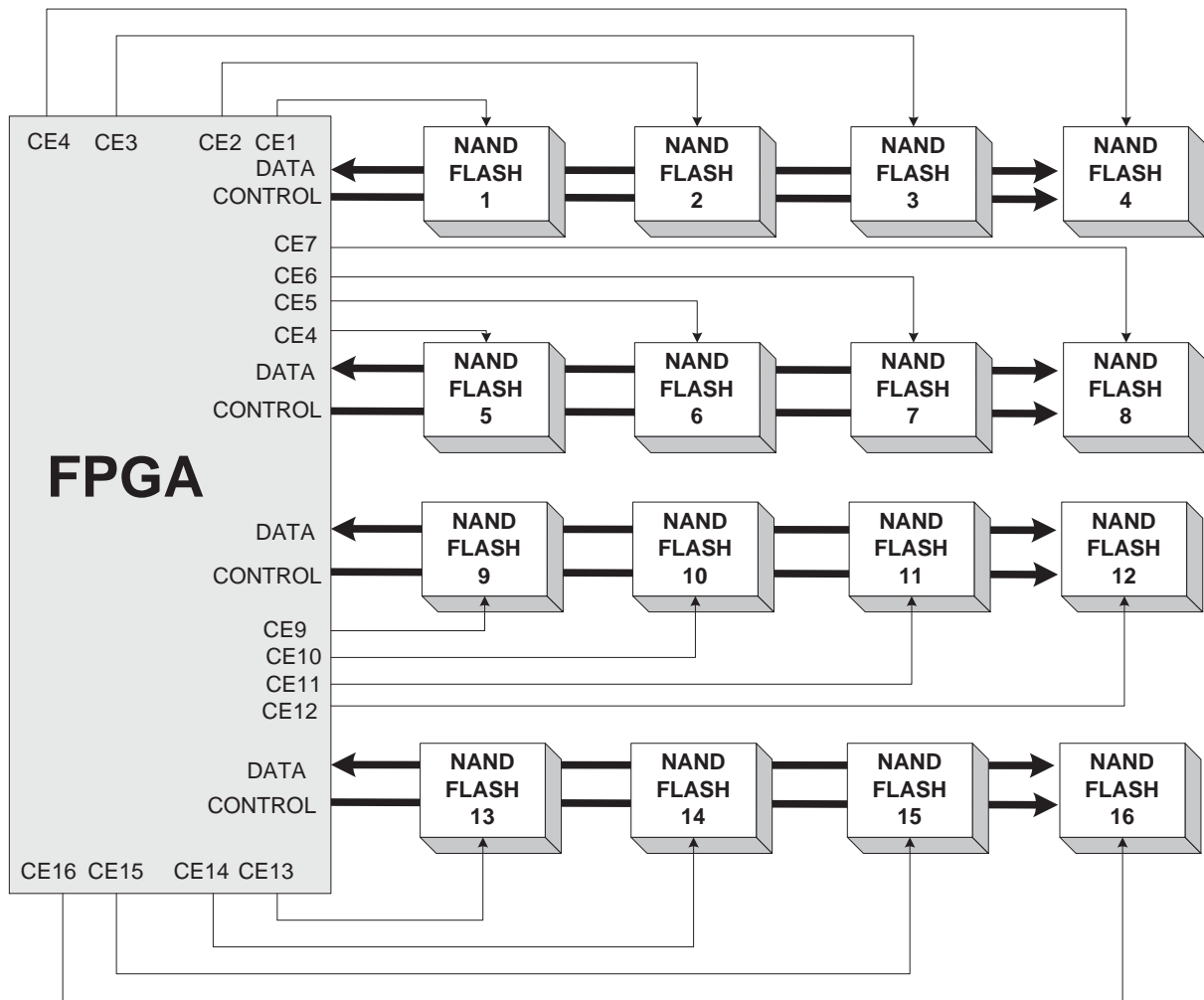


Figure 4.1: *Proposed Bus Architecture*

4.1.5 Bad Block Table

The two available options for the method of storing bad blocks in a table were discussed in Section 3.3.6. The first option was to store a bit for every block in a device, where the value of the bit would determine if that block was good or bad. From equation 3.16 the amount of memory required for 16 devices with 4096 blocks is:

$$\text{Internal FPGA RAM required} = 4096 \times 16 = 64 \text{ kB} \quad (4.1)$$

The second option was to store only addresses of the bad blocks contained within each device. Pointers and offsets can then be used to partition the table into sections for each device. The data sheet for the samsung 512MB device [8], states that the maximum number of bad blocks is 80¹, including bad blocks shipped from the supplier and blocks

¹This number does not take into consideration the radiation effects on NAND flash devices.

which go bad over the lifetime of the device. If 128 locations (16 bits per location - see Section 4.2.2.1) are set aside to store bad block addresses for each device, equation 3.16 states that

$$\text{Internal FPGA RAM required} = 128 \times 16 \times 12 = 48 \text{ kB.} \quad (4.2)$$

Since the required amount of FPGA RAM needed to store the address is significantly lower than the amount needed to describe every bit, it was then decided to progress with the second bad block table design.

The next decision was whether to have one bad block table for all the buses or a separate table for each bus. A separate bad block table for each set of devices on a bus will enable each set of devices to be treated as an independent unit. This means that if the design is expanded or reduced, a unit is simply added or removed. The disadvantage of one bad block table is that only one address can be generated at one time, thereby leading to a slower bad block table generation process. A separate bad block table situated on each bus was therefore picked as the best implementation.

4.1.6 Memory Structure

The memory area consisting of all the NAND flash devices in the system must have a certain structure if multiple images are to be stored. The intention is to store a minimum of five images at any time and make it possible to write, read or erase any one of them.

The intention is that the memory unit will only be used for mass data storage, which means that random access and complex addressing will not be necessary. The memory will be split up into different sections, each the size of an image, and all the addressing will be implemented internally. This will reduce the complexity significantly as complex physical and virtual memory mapping is not required.

It was therefore decided that the whole memory area be split up into five image sections. Each section will have its own unique address and will be erased, read or written individually. With the first block on each device reserved for the bad block table, the size of each section can be calculated as follows:

$$\text{Total Number of Blocks} = 16 \times 4095 = 65520 \text{ Blocks} \quad (4.3)$$

$$\text{Size of Image Section Total} = \frac{65520}{5} = 13104 \text{ Blocks} \quad (4.4)$$

If every image is split evenly across all the 16 devices then the size of an image section on 1 device will be:

$$\text{Size of Image Section Device} = \frac{13104}{16} = 819 \text{ Blocks} \quad (4.5)$$

The image sections will be 13104 blocks in size and each device will contribute 819 blocks to the image section. Figure 4.2 shows the proposed memory structure. The left side shows how the total number of blocks in the memory array will be split up into image sections of 13104 blocks, and the right side shows how each device will be split up into image sections of 819 blocks.

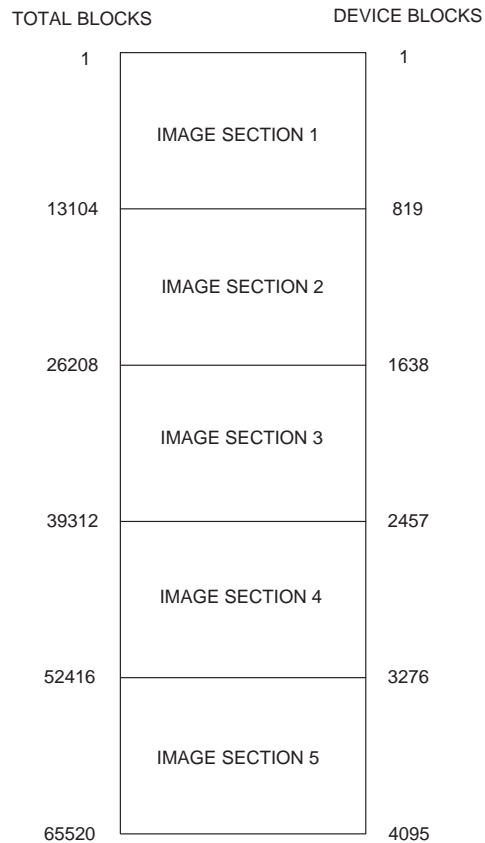


Figure 4.2: *Proposed Memory Structure*

4.1.7 FPGA Selection

The main system design was implemented in an FPGA. The selection of the correct family of FPGA devices is vital, especially in a complex system. One of the important aspects is to ensure that there is another device in the family with a larger capacity which can be used if the design grows to exceed the capacity of the current device.

The listed families of devices in Table 3.2 were considered for this thesis and the Altera Cyclone device family was selected. The reasons for this choice will be discussed next.

Internal RAM - The Cyclone devices offer the highest amount of internal RAM per device. The internal RAM is required for storing bad block addresses and also to implement any necessary FIFO's.

Speed - Initial tests with Cyclone devices indicated that they were capable of the high internal clock speeds necessary for this design.

Availability - As the Cyclone devices have been on the market for a reasonable amount of time compared to the new Actel and Lattice devices so they are readily available.

Development Software - The University of Stellenbosch has full licenses for Altera development software and the author is familiar with these development tools. If another manufacturer had been chosen, other development software would have had to be bought, which would have increased the costs significantly.

Capacity - The Cyclone devices offer a high capacity of logic elements in standard package sizes.

Table 4.1 shows the devices in the Altera Cyclone family which are currently available. The development software recommends the device from the family depending on the size of the VHDL design. The EP1C12 was the chosen device.

Table 4.1: *Cyclone Family of FPGAs*

| Part Number | Logic Elements | RAM (kBytes) | User I/O Pins (max) |
|-------------|----------------|--------------|---------------------|
| EP1C3 | 2910 | 58 | 104 |
| EP1C4 | 4000 | 76 | 301 |
| EP1C6 | 5980 | 90 | 185 |
| EP1C12 | 12060 | 234 | 249 |
| EP1C20 | 20060 | 288 | 301 |

4.1.7.1 Development Software

In the next paragraph, the most important features of the development software used in this design, will be summarised.

The available Altera development software is MaxPlus II and Quartus II. Quartus II offers more advanced features than MaxPlus II but is more resource intensive and therefore slower. As the Cyclone device family is only supported by Quartus II, it was selected as the development software.

Quartus II has its own in-built simulator for verification of the design VHDL functionality. Initial testing with the simulator found that it was slow, lacked features other simulators could offer, and was not very user friendly. Altera-Modelsim is another simulator that

is available with the Quartus II software. Altera-Modelsim is a cut down version of the full version of Modelsim by Model Technology. Given that the features offered by Altera-Modelsim are by far superior to that of the simulator in Quartus II, it was decided that Modelsim would be used for all simulation requirements.

Samsung, the main suppliers of NAND flash memory, have made available precompiled VHDL models for all its NAND flash devices. These models do not include any source code, and can be downloaded on the official Samsung website [6]. This can be used to connect to the design code in order to ensure the correct interface functionality and provide the designer with the ability to test the full design before it is placed in hardware. Since the models provided are precompiled files, the same development software that was used to compile the files is needed to simulate the models. The full version of Modelsim (not Altera-Modelsim) was used to create the precompiled files which meant it was necessary to use the full version of Modelsim (Modelsim SE 5.8c) for all simulations in the design.

4.2 VHDL Design Overview

In this section the design is split up into a number of key components. By doing this, it is possible to reduce the system complexity and keep the system modular. As explained in Sections 4.1.3 and 4.1.4, the system has four sets of control and data lines (buses) that interface with four banks of NAND flash devices. It is therefore necessary for each bus to have an interface driver in the FPGA to control the timing and complex switching of the control lines which interface with the NAND flash devices. The system must also contain a component that sends the required amount of data to each bus at the right time in order to maintain the data rate. The VHDL design can then be split into five main parts - a data interleaver/de-interleaver and four interface controllers. In further discussions, these components will be referred to as the data router and bus controllers, respectively. The block diagram shown in Figure 4.3 shows the proposed system structure.

4.2.1 Data Router

The purpose of the data router is to control the interleaving/de-interleaving of data during write and read sequences, and also to control the whole of the erase sequence. The data router must be able to send commands to the appropriate bus controllers before the data stream is routed. This will enable the execution of necessary setup cycles that NAND flash devices need, before the data arrives for a write sequence or before the data is to read and routed back for a read sequence.

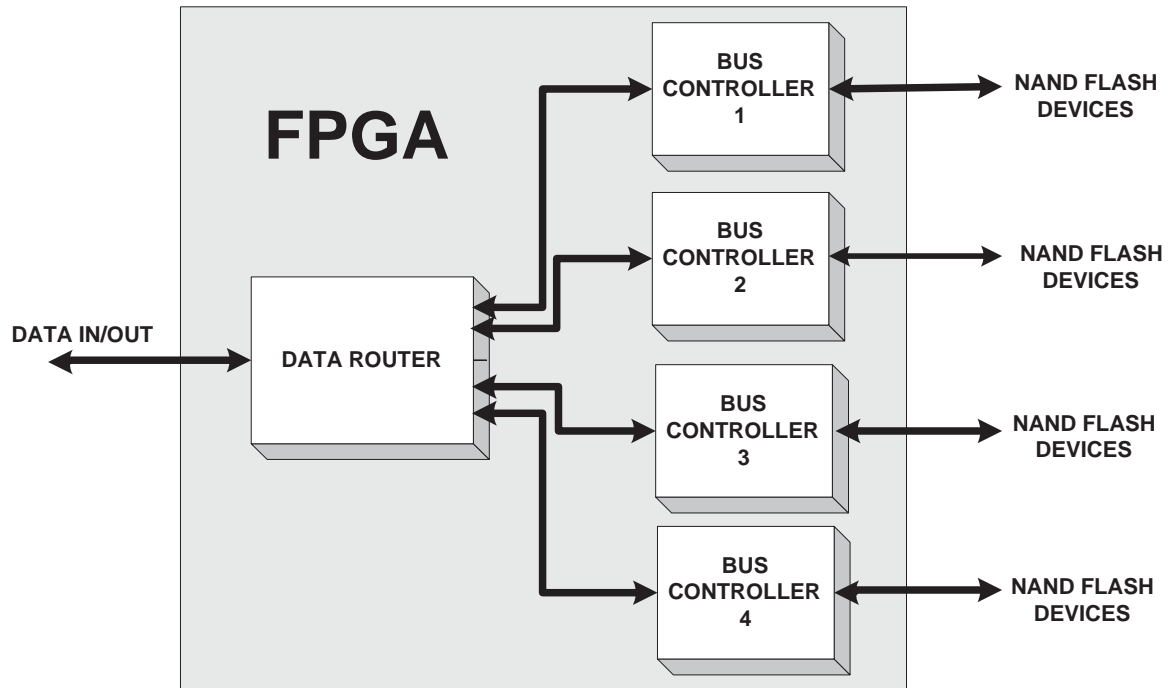


Figure 4.3: *Main Design Block Diagram*

The data router must also have access to the amount of bad blocks on each individual device. The addresses of the bad blocks are not of importance but the amount of bad blocks will aid the data router to be able to perform the following tasks:

- choose the correct bus to route the data to or receive data from so that all the available memory is used. The data rate can still be achieved with only 2 buses, provided the minimum number of devices are available, and that these devices have free blocks to which the data can be written. Keeping a track of available blocks on each bus will therefore enable a greater amount of memory to be accessed.
- calculate if there is enough devices with available blocks to maintain the data rate. There must be a certain amount of devices with blocks available to maintain the data rate. If a write sequence continues without enough devices available, data will be lost while trying to write to a device which is still in a busy state which followed the previous write to that device.
- report the number of bad blocks to the OBC or ground station for analysis.
- switch a bus off if it is not functioning correctly. This can be done simply by making the amount of bad blocks on each device on that bus equal to the total number of blocks on the device. To the data router, it would seem that there are no available blocks on any of the devices on that particular bus, which means that it would bypass it on read, write and erase sequences.

The data router is the main interface between the MMU and the OBC. Therefore, there should be a command handling mechanism and command structure within the VHDL design, as is discussed in detail in the following chapter. Figure 4.4 shows the proposed structure for the data router.

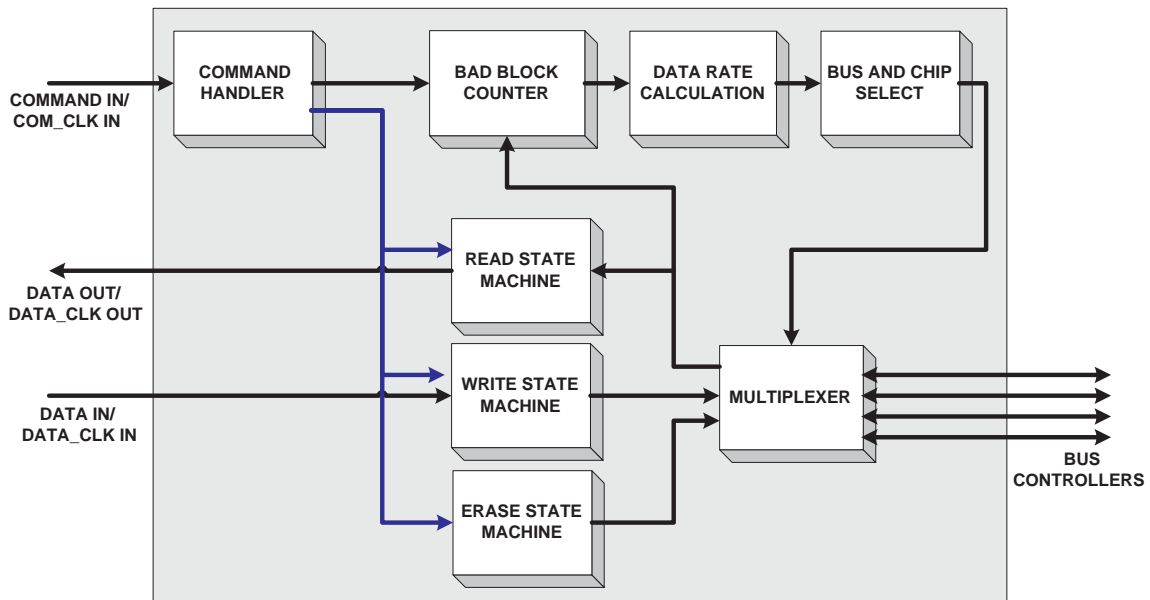


Figure 4.4: *Data Router Block Diagram*

4.2.2 Bus Controller

The bus controller will be responsible for controlling all the interface operations to the NAND flash devices. A separate bad block table will be situated in each bus controller and will be accessed during read, write and erase sequences in order to skip bad block addresses during any of the operations. To increase reliability, it was decided to perform error detection and correction on the data as it enters and leaves the bus controller, and also on the bad block table as it is loaded into internal RAM. The main operations that the bus controller must be able to perform are therefore:

- page program operation to write a page of data to a device.
- read a page of data from a device.
- erase a block from a device.
- reset all the devices on the bus.
- perform error detection and correction on data.

- read the bad block table from a selected device and store it in the internal RAM of the FPGA.
- read the bad block table from internal RAM of the FPGA and store it in the selected device.
- generate addresses while skipping bad blocks.

Accordingly the bus controller can be split into three main components, namely the interface control unit, bad block table unit and error detection and correction unit. Figure 4.5 shows the proposed format for the bus controller.

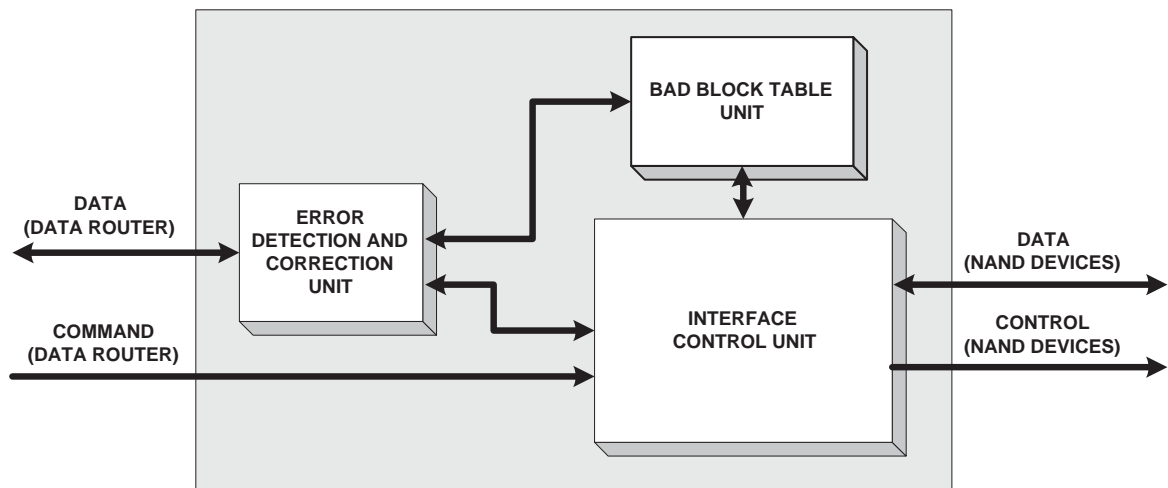


Figure 4.5: *Bus Controller Block Diagram*

In the following paragraphs, each of these components will be discussed in turn.

4.2.2.1 Bad Block Table Unit

The bad block table situated in each bus controller consists of 128 locations for each of the four connected devices' bad block addresses. The bad block addresses have to be accessed for every operation, but only during an erase sequence will the table to updated with any new bad blocks. For incremental addressing, starting at block 1 and working up though the device, it will be necessary for the bad blocks in the table to be ordered from the lowest block address to highest block address. This means that in order to access the next bad block address, one simply has to retrieve the value of the next location from the bad block table.

To store any new bad block addresses during an erase sequence, a section of the FPGA internal RAM must be used to hold this address before it can be placed at the correct

position in the table. It was decided to split the RAM into two sections for each device, one section for the present bad blocks (normal area) and another section for bad blocks that occur during an erase sequence (spare area). The new bad blocks found during an erase have to be placed in the spare area because reordering of the normal area every time a new bad block occurs would dramatically slow down the whole erase sequence. Once the erase sequence is finished, the new bad blocks in the spare area can be added to the present bad blocks in the normal area, followed by a reordering of the normal area to put the addresses in ascending numerical order. This reordering is done during the process where the table is stored back in flash memory.

The normal area of each device will contain 100 locations for present bad blocks, while the spare area will contain 28 locations for any new bad blocks that occur during an erase sequence. Pointers will be used to store the current position in both the normal and spare areas. Offsets will be used to access bad blocks of the different devices contained within the table. The proposed format for the the bad block table is shown in Figure 4.6.

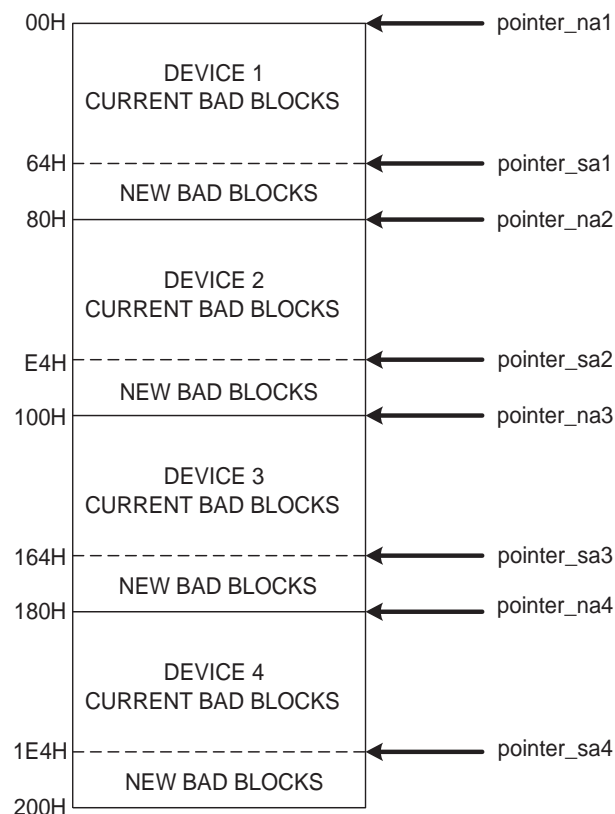


Figure 4.6: *Bad Block Table Format*

A block address is 12 bits wide for a device with 4096 blocks. As 16-bit devices were chosen for this design, 16 bits will be used to store the bad block address. The extra 4

bits will be used for status information about the block. These 4 bits state whether the block is in the normal area or in the spare area of the bad block table. A total of 2048 bytes is then required to store the whole table. This fits comfortably into a single page, with enough space left for error checking and correction (ECC) bytes.

Bad Block Table Access and Control

In addition to the bus controller, the entire bad block table is also to be stored at block 1 on every device on a bus. This is because block 1 of every device is guaranteed to be a valid block [8]. This enables multiple copies of the bad block table to be present, which increases system reliability. The operations that require access to the bad block table are: loading the table from a NAND flash device, storing the table into a NAND flash device, and address generation for bad block skipping. The bad block table unit can be split into four different sections as shown in Figure 4.7.

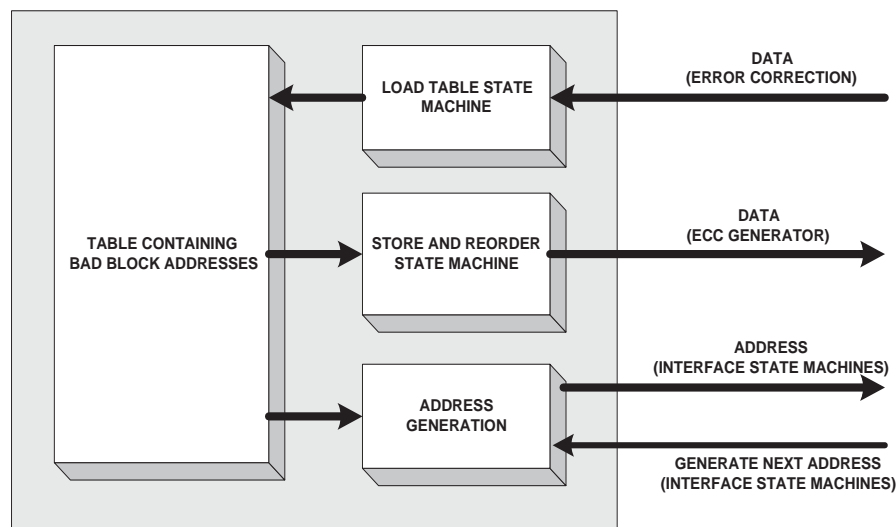


Figure 4.7: *Bad Block Table Block Diagram*

The **load operation** simply retrieves the bad block addresses from the interface control unit and stores them directly in the internal RAM.

The **store operation** is slightly more complex, as this is where the reordering of the bad block addresses is implemented. The store operation must compare the block addresses of the normal area to the block addresses of the spare area of the bad block table before sending the lower addresses to be stored in flash. This will enable the bad block table to be reordered sequentially from lowest block to highest.

The **address generation operation** creates incremental addresses starting at block 1 of device 1. The generated address is compared to the block address read from the bad block table and skipped if it is the same as the bad block address.

The next component in the bus controller structure shown in Figure 4.5, is the interface control unit. Its functionality will be discussed next.

4.2.2.2 NAND Flash Interface Control

The interface control unit is implemented as a number of state machines which setup the correct command and address cycles for interfacing with the NAND flash devices. Due to the fact the write, read, erase and device reset operations all require a different number of command and address cycles, it was decided to create a separate state machine for each operation. Each state machine would then control the exact sequence of commands and addresses needed for that operation.

As the state machines need to have access to the NAND control and data lines, a multiplexer was implemented to connect the appropriate state machine to these lines. A controller was added to receive commands from the data router and to signal the activation of the appropriate state machine for the intended operation. The controller also controls the multiplexer switching.

The I/O lines for data transfer to and from the NAND flash devices are bidirectional. It is therefore necessary to set these lines to a high impedance state when reading data back from the device. To do this a bidirectional buffer is required. The read state machine requires the use of the I/O lines in both directions, firstly, for sending out the read command and the page/block address, and secondly for reading data back. The read state machine has control of the bidirectional buffer as it is the only state machine that requires the high impedance state of the I/O lines. The block diagram of the proposed interface control is shown in Figure 4.8.

The final component in the bus controller unit, the error detection and correction unit, will be discussed next.

4.2.2.3 EDAC

The need for error detection and correction (EDAC) arose from the fact that NAND flash devices are prone to single bit errors during writing and reading (Section 3.2.3). Soft errors occur at a rate of 10^{-10} or about 1 bit per 10 billion bits programmed [15]. Due to

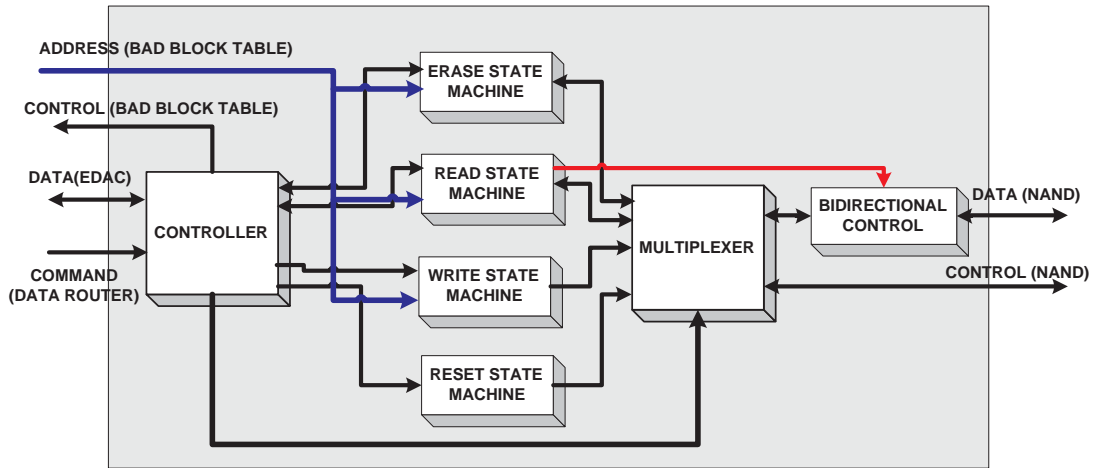


Figure 4.8: Interface Control Block Diagram

this low probability, single bit correcting Hamming code is sufficient for NAND flash.

The recommended Hamming code from the major NAND flash suppliers is a 22-bit code word for every 256 bytes. The code words can be stored in the flash spare area of a page after the data has been stored. For a 2048 byte page, 12 bytes in the spare area are needed. The code words are firstly created on a page during a write sequence, and then stored in flash. The same algorithm is then used to create another code word while the same page is being read from flash. The two code words are then compared and any errors are detected. In the case of a single bit flip, the error can be corrected. This method will not effect the write data rate as the extra bytes can be stored after the page of data is written and the data stream has been routed to another bus. The read time will be however affected slightly because the data has to be stored before the code words can be compared and errors corrected. The EDAC system therefore requires a code word generator, a code word comparison unit, a FIFO (to store the page of data) and a error correction unit. Figure 4.9 shows the proposed EDAC system.

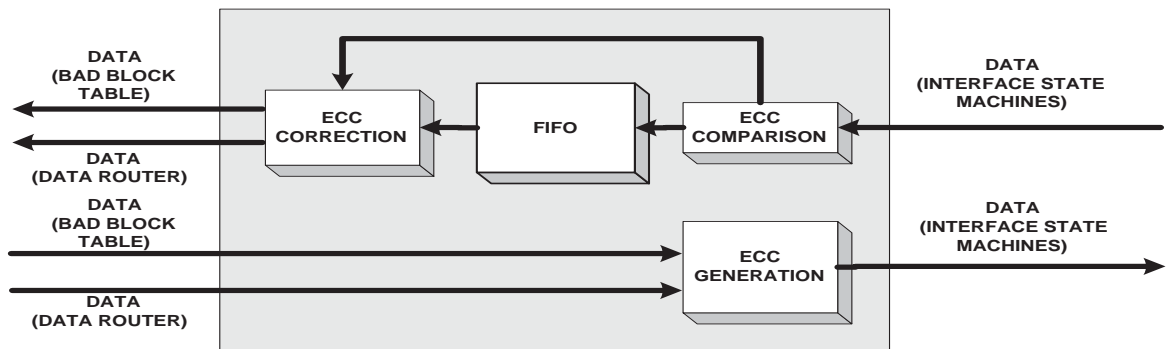


Figure 4.9: EDAC Block Diagram

4.2.3 Summary

As the present chapter has presented an overview of the VHDL design, the next step would be to look at the detailed implementation of the proposed functionality.

Chapter 5

Detailed System Design

In this chapter the VHDL and the mass memory demonstration board designs are given in detail. The VHDL components are described individually and the more complex components are described through the use of flow charts. Following this is the PCB design of the mass memory demonstration board, which ranges from component selection to board layout and routing. The chapter finishes with a discussion of simultaneous read and write functionality which was added to the design at a later stage.

5.1 Design Changes

The design and manufacture of a mass memory board is intended only as a demonstration of how NAND flash devices can be used for a MMU. It was therefore decided that the Samsung K9F1208U0A device would be used, as it would show the same design functionality as the high capacity devices but at a lower cost to the project. The fact that a stock of Samsung K9F1208U0A devices were already available in the laboratory, was another reason to choose it over the higher density devices. The Samsung K9F1208U0A NAND flash device has a capacity of 64 MB, with 8-bit I/O lines. Since an 8-bit device is to be used instead of a 16-bit one, the proposed data rate specification will be halved. Additionally, the capacity specification will be reduced by a factor of 8 because the chosen device size is 64MB compared to 512MB used in the initial estimate. The size of the page register is 528 bytes.

The new specifications for the design are therefore:

- Data Rate - 8MB/s
- Capacity - 1GB

The amount of devices needed to sustain the data rate of 8MB/s using Samsung K9F1208U0A

devices with $t_{\text{PROG}} = 500\mu\text{s}$ and a write cycle time of 120 ns, is as follows:

$$\text{Number of Devices} = 1 + \frac{500\mu\text{s}}{(120\text{ns} \times 518)} = 11 \text{ devices} \quad (5.1)$$

It was decided to split the memory into 5 sections, with each section used to store a separate image. An image will be 13000 blocks in size. The sections are made slightly larger than an image, so that the extra blocks can be used to replace any bad blocks that might occur.

5.2 VHDL Design Detail

A state machine design methodology was used for the VHDL design on the mass memory board, as this approach allows a structured design to be implemented. State machines also allow the VHDL code to be easily expanded, as new functionality can be added by simply adding a new state. Another advantage is that state machines allow debugging to be done efficiently, as a fault can be easily isolated to a single state.

As mentioned in Section 3.1.7, VHDL is a hardware descriptive language that supports a modular approach. Individual components can be designed and simulated before being integrated into the final design. The following sections describe each individual VHDL component in detail.

5.2.1 Bus Controller

As illustrated in Figure 4.5, the bus controller is the high level component name for the connection of the interface control unit, the bad block table unit and the EDAC unit.

5.2.1.1 Interface Control Unit

The four main interface operations are to write a page, read a page, erase a block and reset the device. The operations each require different command and address cycles, therefore a separate state machine was used for each operation in order to reduce complexity.

The maximum write/read cycle time (relaxed value) given in [8] is 60ns. To compensate for propagation delays and delayed rise times and fall times due to capacitive loading, an 80ns cycle time was implemented as the maximum write/read cycle time. A 25MHz clock was then required to drive the interface state machines so that the data hold and setup times were both 40ns. This write cycle time gives a maximum write data rate of 11.92MB/s. Due to ECC the maximum read data rate was restricted to 7.95MB/s, a 120ns cycle time.

Page Program State Machine

The page program state machine writes 518 bytes (512 data bytes + 6 ECC bytes) to flash memory. The input data stream is sent through from the data router. The page address, block address and the selected device are supplied by the address generator in the bad block table unit. Once the address is latched in the write state machine, a signal *addr_get* is sent to the address generator so that the next valid address and device can be found. The state machine does not wait for the ready/busy (*rdy/by*) line of the active flash device to signal the completion of a program operation. Instead it returns to the start state and waits for another command. This is done to allow another device to be written to on the same bus while the previous device is still busy programming.

The page program state machine also writes the bad block table to flash, where it is stored in all the NAND flash devices on the bus at page addresses 0 and 1 of block 0. Two separate commands from the controller can start the page program state machine, one for a normal page write and one for a bad block table write.

When the page program state machine is first activated, the address is latched and the chip select of the NAND flash device to be programmed is pulled low. The 80H command is then written to the device followed by 4 address cycles. The first address cycle selects the column address and the next 3 cycles set the page and block address. The state machine then waits for the data to be routed through from the data router via the ECC generator of the EDAC, before it is written to the device. The 10H command is sent to the flash device once all 518 bytes of data have been transferred to the page register. The state machine then returns to its start state awaiting the next command.

Read State Machine

The main function of the read state machine is to read 518 bytes (512+6 ECC bytes) from flash memory and send them to the data router. The address is supplied by the bad block table unit. Once the address is latched, a signal *addr_gen* is sent to the address generator so that the next valid address can be generated. Another function of the read state machine is to read the bad block table from flash and send it to the internal FPGA RAM bad block table. The final functionality required of the read state machine is to read the status register after an erase operation by the erase state machine. The value of the status register shows whether a block is good or has become bad. Three commands from the controller can be used to start the read state machine: one for a normal page read, one for a bad block table read, and one for a status register read.

When the read state machine is activated, it selects the target flash device by pulling the

chip select of that device low. The command 00H is sent to the flash device to activate a read operation. This is followed by 4 address cycles for the column, page, and block addresses. The device then goes into a busy state as the data is retrieved from internal memory and transferred to the page register. Finally, the read state machine reads out the 518 bytes from the page register and sends it to the error detection unit of the EDAC, after which it is transferred to the data router.

The read state machine also reads the value of the status register after an erase operation. This is achieved by sending the 70H command to the flash device which has just finished an erase operation and reading one byte from the device by toggling the read enable pin. The byte from the status register is then checked to see if a bad block has occurred. If bit 0 is a '1' then the block has failed during the erase operation. The block address is then sent to the bad block table unit to be stored in the spare area of the bad block table.

Erase State Machine

The main function of the erase state machine is to erase a single block of data. Block addresses are supplied by the address generator in the bad block table unit. The erase state machine also erases the bad block table stored in flash. The bad block table is stored in block 0 of all the devices on the bus. Two commands from the controller can be used to start the erase state machine, one to erase a normal block of data and one to erase the bad block table.

When the erase state machine is activated by the controller, the chip select of the target flash device is first pulled low. The command 60H is then written to the flash device and is followed by 3 address cycles. The 3 address cycles are for the block address of the block to be erased. The command D0H then follows to initiate the erase process within the flash device. The erase state machine waits for the *rdy/by* line to go low (busy) and then return to a high (ready) before returning to its start state.

Reset State Machine

The reset state machine resets all the NAND flash devices which are connected to that particular bus. The command FFH is written to all the flash devices, after which the state machine waits for the *rdy/by* line to go low (busy) and return high (ready) before returning to its idle state.

Controller and Multiplexer

The controller for the interface state machines simply receives commands from the data router for the required operation and sends a command to the appropriate state machine. The controller sends a load signal (*load*) to the bad block table unit if the table is to be read from flash, and a store command (*store*) if the table is to be written to flash. For an erase operation the controller sends a command to the erase state machine and waits until it is finished before sending a read status register command to the read state machine. The controller also changes the select lines to the multiplexer in order for the control and data lines to be available to the currently operating state machine.

5.2.1.2 Bad Block Table Unit

The bad block table unit consists of the bad block table, a state machine which generates valid addresses by skipping bad blocks, a state machine which loads the block addresses into the table and a state machine which reads and reorders the bad block addresses from the table. The 16-bit address that is stored in the table is made up of 12 bits for the address of the bad block and 4 bits for status information. The 4 status bits mark the address as either an empty location in the normal area (0111), an empty location in the spare area (1111) or a proper bad block address (0000) which can be located in either area.

Load Table

The load table state machine receives the bad block addresses from the error correction unit of the EDAC after they had been read from flash, and writes them to the bad block table in the internal RAM of the FPGA. The state machine starts at RAM address 0 and writes the bad block addresses to each RAM location sequentially.

The load table state machine also deals with any new bad blocks that have occurred during an erase operation. The address of the bad block, including the device number, is sent from the read state machine in the interface control unit and is then stored in the spare area of the particular device's section of the bad block table.

Address Generation

Valid addresses for the write, read and erase state machines are generated in the address generator state machine. A page, block and device address is generated for the read and write state machines, and a block and device address is generated for the erase state machine. After the previous operation has finished, the page address of each device is incremented by 1 for a write and read operation, and the block address is incremented

by 1 for an erase operation. The start address for a write or read operation on image section 1 is page 0 of block 1 on device 1. Block 0 on device 1 is reserved for the bad block table. The second, third, and fourth addresses are page 0 of block 1 on device 2, 3, and 4 respectively and the fifth address is page 1 of block 1 on device 1.

The verification process of the newly generated address is as follows: The bad block address at the present pointer location is read from the normal area (see Figure 4.6) of the bad block table and is compared with the new address. If the addresses match, a new address is generated (block address incremented by 1), the next bad block address is read from the bad block table (pointer incremented by 1) and the two addresses are again compared. This process continues until a valid address has been generated. If one device has no available blocks left, the next device in line is selected and checked for an available block. When all devices have no available blocks left, the state machine returns to the start state. The flow diagram for the address generation state machine is shown in Figure 5.1.

Table Store and Update

The table store state machine reads the bad block addresses from the bad block table in RAM, reorders them from lowest address to highest address for each device and then sends them to the interface state machines via the error generator of the EDAC unit to be stored into flash. The reordering is achieved by reading the first address in the spare area and the first address in the normal area, comparing the two addresses and sending the lower address to the EDAC unit. The second address in the area from which an address was sent out is then read and compared with the first address of the other area, and again the lowest address is sent out. This continues until all bad block addresses have been sent out for this particular device and empty addresses are then sent to fill up the rest of the 128 locations. 7FFFH and FFFFH are sent as empty addresses for the normal and spare areas respectively. This process is repeated for all four device bad block tables until the full bad block table is stored in flash. The flow diagram for the store state machine is shown in Figure 5.2.

5.2.1.3 EDAC

As discussed in Section 3.2.3, a Hamming algorithm was chosen to implement error correction, due to the low error rate present in the system. The error detection and correction unit can detect and correct one bit out of every 256 bytes [9]. To do this, an 11-bit code word is produced for every 256 bytes, which means that two code words are produced for every page. The following paragraphs describe each component of the EDAC in turn.

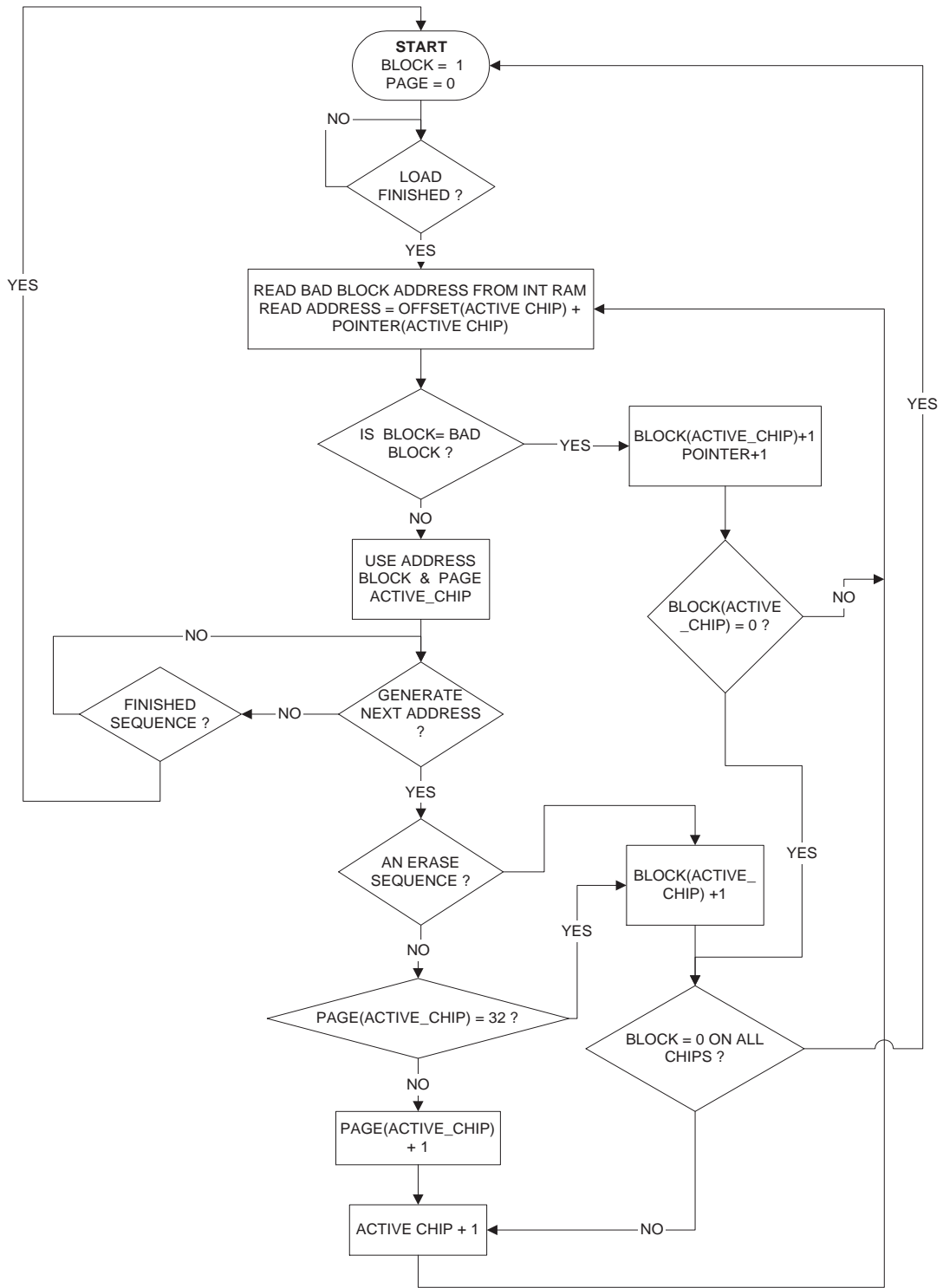


Figure 5.1: Bad Block Table - Address Generation Flow Diagram

Error Code Generation

The error code generator produces the error codes for data, which is received from either the data router or the bad block table unit, and is to be stored in flash. The data is

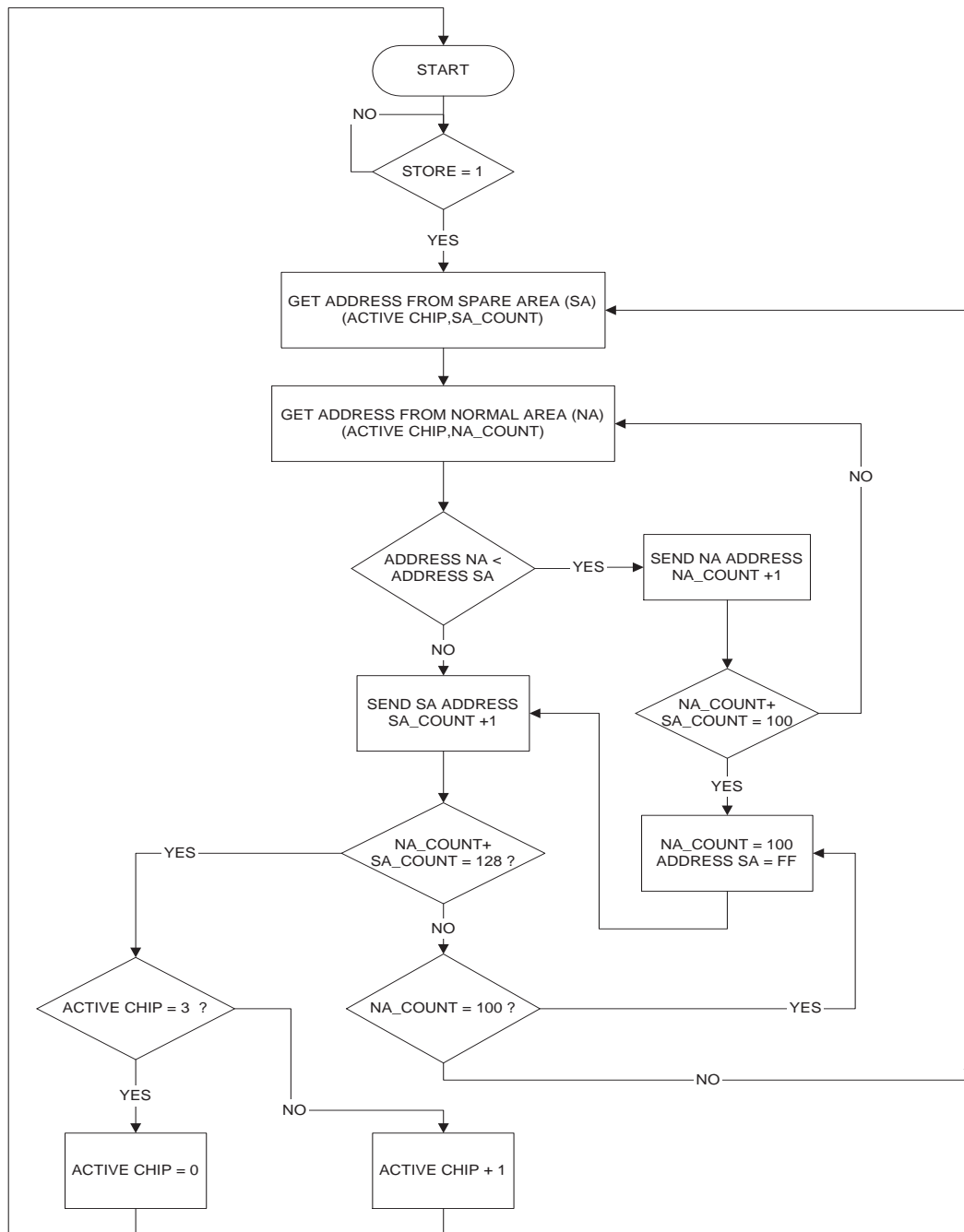


Figure 5.2: *Bad Block Table - Update and Store*

latched and then routed to the interface control unit without affecting the data rate. The two code words generated from one page of data are sent to the interface control unit after the last byte of data for that page has been sent.

Error Detection

After a page of data has been read from flash it is passed to the error detection component. The same Hamming algorithm as used in the error code generator is then applied to the page of data and another two code words are produced. The new code words and the previously generated code words are then compared. The following results are possible after the XOR comparison:

- all bits at '0' - no errors to report.
- 11 bits at '1' - a 1-bit correctable error.
- only 1 bit at a '1' - an error in the code word.
- random data - a non-correctable error.

Next, the error detection component sends out two 11-bit addresses and a status flag to the error correction component. The two 11 bit addresses refer to the byte and bit address of any bit that is flipped, while the status flag is set if an error occurred.

FIFO

The FIFO is a 512 byte first-in-first-out register to temporarily hold a page of data while the error detection unit detects any errors contained in the data. The error correction component accesses the data exiting from the FIFO.

Error Correction

The error correction component corrects any 1-bit errors contained within either of the 256 bytes of data in a page. The error correction component state machine stays idle until the two 11-bit addresses and status flags are received from the error detection component. It's function is then to read the data out of the FIFO a byte at a time and route it either to the data router (normal page read) or to the bad block table (bad block table read). If the status flags indicate any 1-bit errors on either of the 256 bytes then these errors are corrected by using the byte and bit address sent from the error detection component. The flag *read_bytes* controlled by the data router, signals when the data is to be sent through to the data router. The data remains in the FIFO until this flag is set to '1'.

5.2.2 Data Router

The data router component routes data to and from the selected buses and also controls the erase operation. The data router consists of one main state machine for bad block counting and bus selecting, and three smaller state machines for controlling the operation of the write, read and erase operations respectively.

5.2.2.1 Bad Block Counter

The flow diagram for the counting of bad blocks for each device is shown in Figure 5.3. In order to maximise the available space left in the total cumulative flash memory during any operation, the amount of bad blocks on each device must be known by the data router. To do this, the number of available blocks on each device is stored in internal registers of the FPGA. During the initial loading of the bad block table into internal RAM, it is also routed through the data router where the amount of bad blocks on each device are counted. If the last four bits of an address contain all 0's then that address is counted as a bad block, otherwise it is an empty address in the normal area or spare area of the bad block table. After the 28 empty addresses of the spare area are read, with the last four bits all 1's, the data router moves on to the next device.

The registers containing the number of bad blocks on each bus can be accessed by the OBC so that an accurate status of the available memory can be found.

5.2.2.2 Data Rate Calculation

In order to calculate if the data rate is sustainable, the available block registers have to be accessed. If 11 devices have blocks available, then the data rate can still be sustained. However, if fewer than 11 devices have available blocks, the data rate requirements can not be met and the main data router state machine returns to an idle state, signalling the end of memory available in that section. The flow diagram for the data rate calculation is shown in Figure 5.4.

5.2.2.3 Bus Select

The bus select part of the main data router state machine chooses which bus is next in line for the current read, write, or erase operation. At the start of a sequence, a full image read/write/erase, if all buses are functional, the bus select algorithm simply cycles through each bus in turn. At the end of a sequence, when devices on one or more buses have no available blocks, the bus select algorithm selects the next bus with available blocks, provided it is not the bus currently active. If a bus has been deactivated, available blocks for all devices equal zero, which means it will be skipped by the bus select algorithm.

The bus select part of the main data router state machine finds the next bus to be accessed in the sequence when one of the *get_next_bus* flags have been set by either the write, read or erase routing state machine. The appropriate command is then sent to the selected bus controller. This command is sent only a few clock cycles after the data has begun to

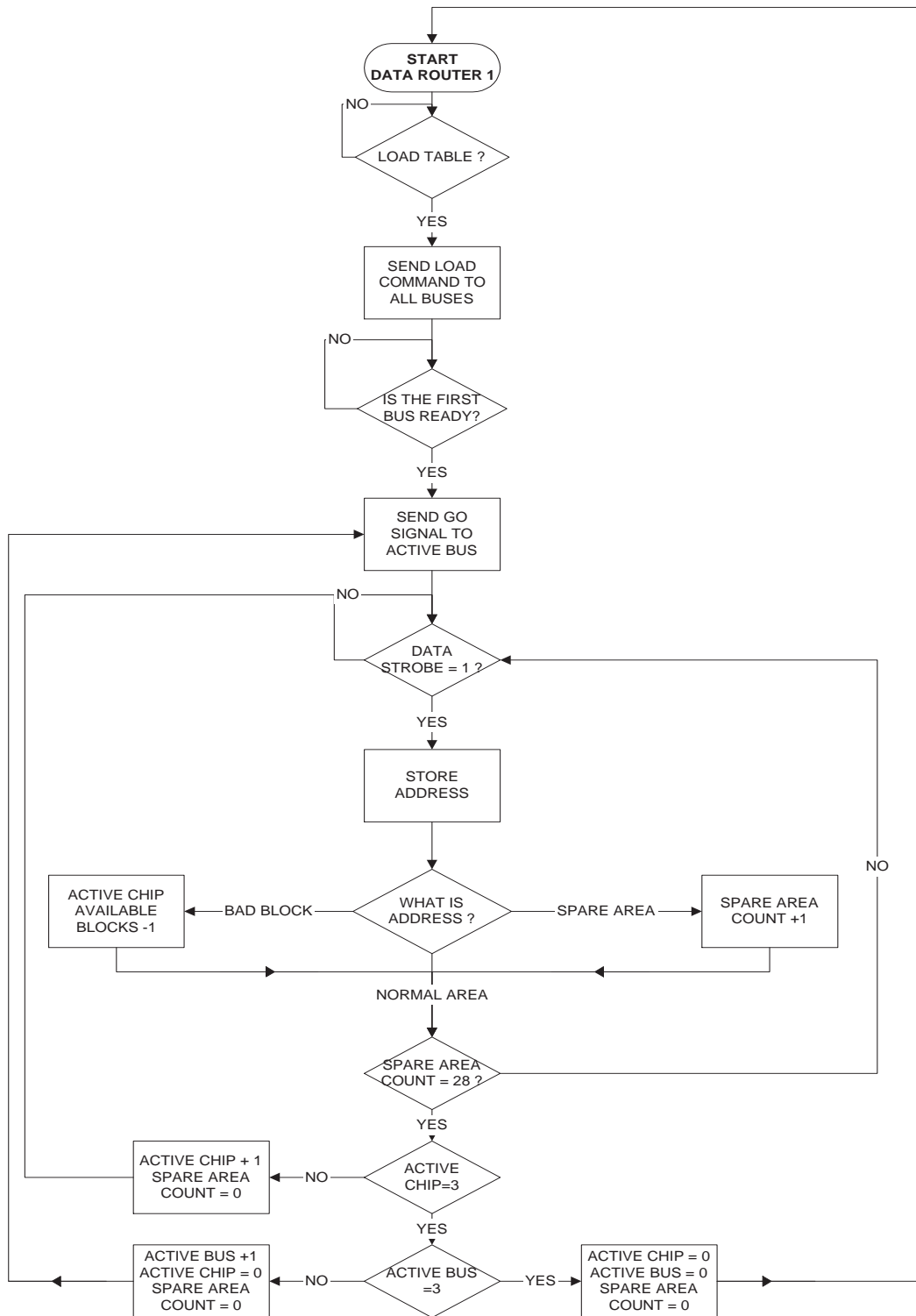


Figure 5.3: Data Router - Count Bad Blocks Flow Diagram

be routed to the previously selected bus, enabling the command and address setup cycles to be completed on the destination flash device long before the data arrives.

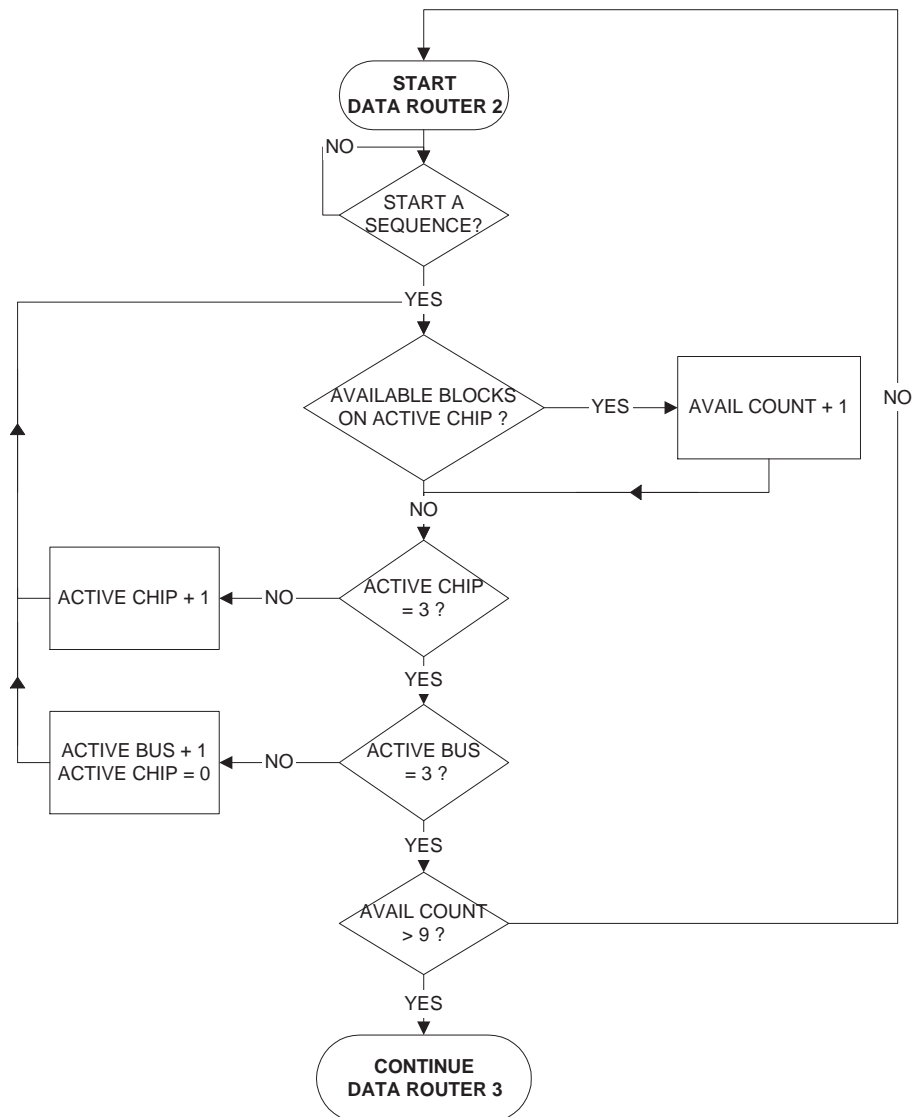


Figure 5.4: *Data Router - Data Rate Calculation*

The bus select algorithm is the same for the image write, read and erase operations, which means that the same blocks are written, read, and erased. After 32 pages have been either routed to or routed from all the devices in a particular sequence for an image read or write operation, the available block registers are decreased by 1 for each device used. If the current operation is an image erase, the number of available blocks is decreased at the end of each sequence, as a whole block is erased at a time. The flow diagrams showing the bus select algorithm are shown in Figure 5.5.

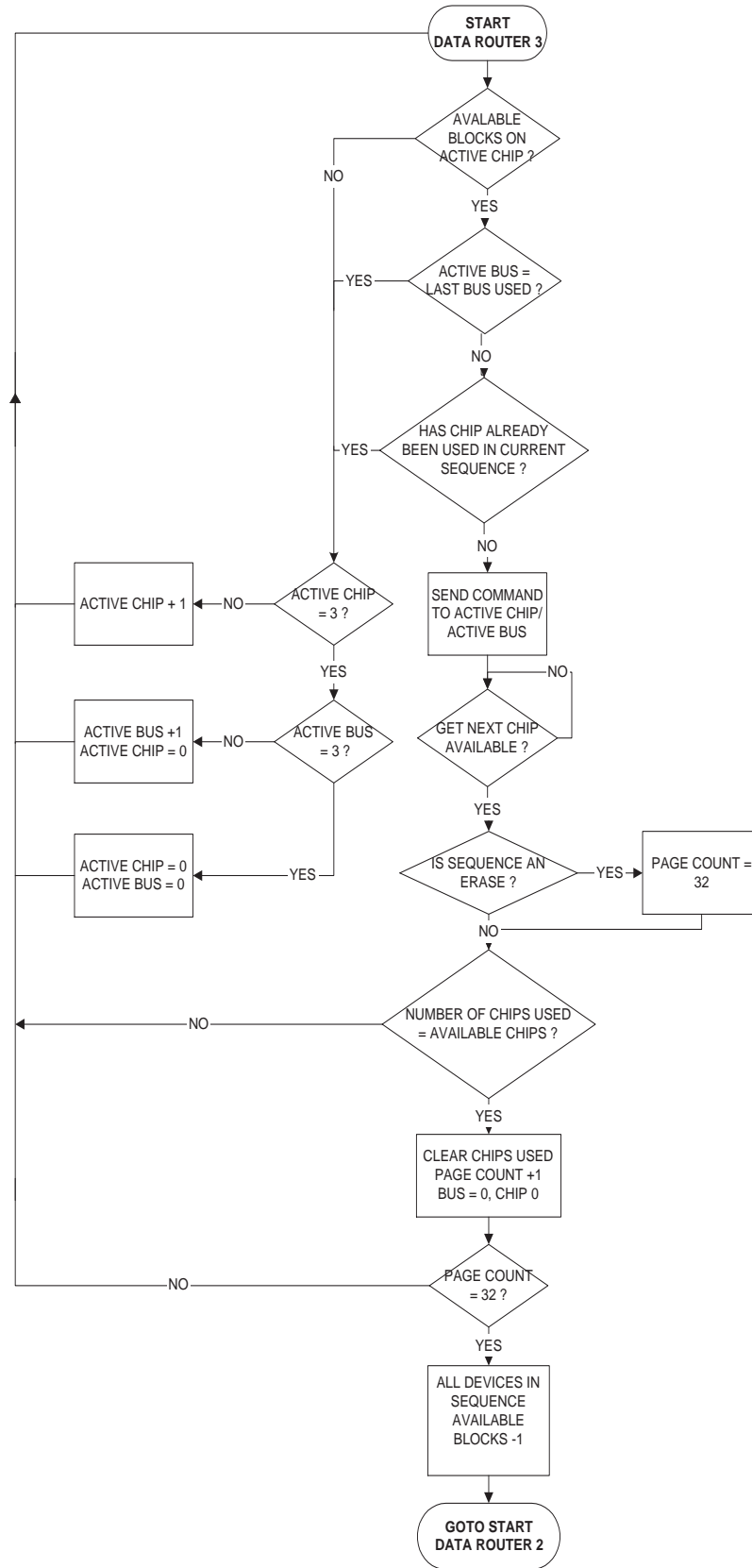


Figure 5.5: Data Router - Bus Select Flow Diagram

5.2.2.4 Read, Write and Erase Routing Control

Write Routing State Machine

The write routing state machine is used to latch the incoming data and then to send it out to the bus selected by the bus select algorithm. When 512 bytes have been sent to one bus, a flag *get_next_bus_wr* is set. This causes the bus select state machine to switch the current bus to the next bus in the sequence and the data to be routed to this bus. The data continues to be routed until the *start_write* flag is set to '0' when the image write operation is complete, or the data rate can no longer be sustained. This flag is controlled by the main data router state machine.

Read Routing State Machine

The read routing state machine routes data from the separate buses into a single output data stream. A flag *read_bytes* is set to '1' to signal to active bus to start sending the data to the data router. When 512 bytes have been read, the *get_next_bus_rd* flag is set so that the bus select part of the main state machine can change to the next bus in the sequence. The *read_start* flag, controlled by the main router state machine, signals when an image read operation is complete and the read state machine can return to an idle state.

Erase Routing State Machine

The erase routing state machine sets the *get_next_bus_er* flag when the next block is to be erased and waits for the read status byte to be returned. Once this byte is returned the next block is signalled to be erased. The *start_erase* flag controller set by the main router state machine, activates and terminates the erase routing state machine.

5.2.3 Device Deactivation

If a device has been found to be faulty by the OBC or any monitoring system, a command can be sent to the data router to prevent that particular device from being accessed. This is achieved by setting the register which contains the amount of blocks available on that device to zero, which will cause the data router to skip that device.

If a device is deactivated, the sectioning of the total flash memory area is changed so that size of an image section stays the same. If only one device is disabled, the section boundaries change so that the memory area contains four whole sections of 13104 blocks and one partial section of 9009 blocks. This means that four whole images can be stored, which is better than five partial images, as would have been the case, had the sectioning not

be resized. This enables the memory unit to still be fully functional at a reduced capacity.

A register *dev_status* is used to store the status of all the devices. This register is set in the data router when the command to deactivate/activate a device is received. The register is accessed by the bus controllers, in order to adjust the addressing for the image section boundaries accordingly. These section boundaries are used by the data router to count the number of bad blocks in the current image section. Therefore, the section adjustments have to take place when the load bad block command is received, before the bad block table is routed through to the data router.

A maximum of five devices can be deactivated before the data rate can no longer be sustained. Table 5.1 shows the number of 13104 image sections and the amount of blocks left in the subsequent partial image section when a number of devices have been deactivated.

Table 5.1: *Section Sizing with Defective Devices*

| <i>Number of Defective Devices</i> | <i>Number of 13104 Sections</i> | <i>Partial Section Size</i> |
|------------------------------------|---------------------------------|-----------------------------|
| 0 | 5 | - |
| 1 | 4 | 9009 |
| 2 | 4 | 4914 |
| 3 | 4 | 819 |
| 4 | 3 | 9928 |
| 5 | 3 | 5733 |

By dynamically adjusting the flash memory sectioning, it is therefore possible to keep the system working as well as it possibly could, in the case of multiple device failures.

5.2.4 Testing and PC Interface

To test the functionality of the system design at the high data transfer rates, A test generator, UART and FIFO were added to the design.

Test Generator

The function of the test generator is to generate test patterns at the required testing data rate. These test vectors are then sent directly to the data router in order to check for correct system functionality, the test patterns were changed throughout the development process

UART and FIFO

A UART of baud rate 57600 was implemented for serial data transfer between the test board and a personal computer (PC). A 4096 byte FIFO was included to store 8 pages of data before it is sent out through the UART to the PC. This enabled the full bad block table to be sent without slowing down the read cycle time. The component configuration for the test generation and PC interface control is shown in Figure 5.6.

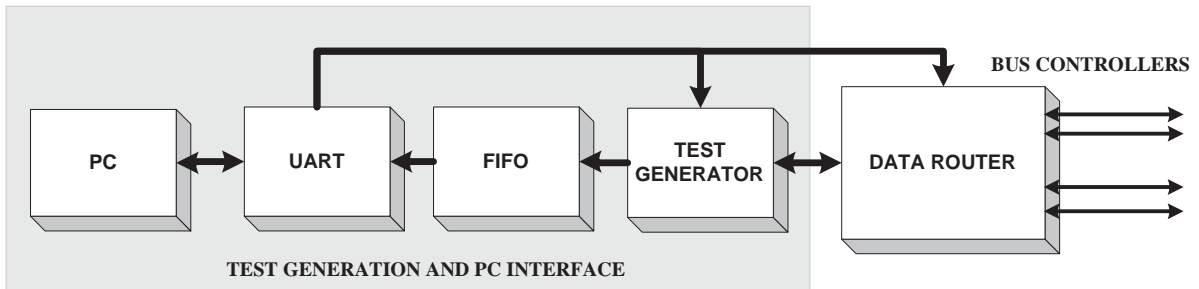


Figure 5.6: *Test Generator and PC Interface*

5.2.5 Command Structure

A command structure was implemented for controlling the demonstration board externally. The process is as follows: The commands are sent from the serial port of the PC through to the UART of the system, after which the command registers of the data router and test generator latch the command from the UART. Each command consists of one byte and are listed next:

Board Wide Reset: This command resets all the state machines included in the design in order to return them to a good known state. Another way to do this, is to use the switch on the test board.

Write Image Operation: This command starts a write image operation, and has to follow a bad block table load operation. A whole section of the memory is written to with data generated by the test generator.

Read Image Operation: This command starts a read image operation, and has to follow a bad block table load operation. A whole section of the memory is read and is sent through to the UART.

Erase Image Operation: This command starts an erase image operation, and has to follow a bad block table load operation. A whole section of the memory is erased.

Bad Block Table Load Section n: This command loads the bad block table into internal RAM and sets the memory section address boundaries to section n.

Device n on/off: This command deactivates/activates device n and the section boundaries are subsequently updated, after the bad block table load operation has taken place.

Bad Blocks bus n: This command sends out the number of bad blocks on each device of the selected bus, via the UART. It has to follow a bad block table load operation.

NAND Flash Reset: This command resets all the NAND flash devices on the board.

Table 5.2 shows a list of all the available command codes.

Table 5.2: *List of Commands*

| Command | Function |
|-----------|---------------------------------|
| 00H | Board Wide Reset |
| 02H | Start a write operation |
| 03H | Start a read operation |
| 04H | Start an erase operation |
| 05H | Store BB table in flash |
| 06H | Erase BB table from flash |
| 09H | Load BB table for section 1 |
| 0AH | Load BB table for section 2 |
| 0BH | Load BB table for section 3 |
| 0CH | Load BB table for section 4 |
| 0DH | Load BB table for section 5 |
| 14H | Read bad block count for bus 0 |
| 15H | Read bad block count for bus 1 |
| 16H | Read bad block count for bus 2 |
| 17H | Read bad block count for bus 3 |
| 20H - 2FH | Deactive/Activate Device 1 - 16 |
| FFH | Device Reset |

5.3 Printed Circuit Board Design - Schematics and Layout

A printed circuit board (PCB) was designed and developed to illustrate the actual physical working of the design, instead of relying only on the VHDL simulations for verifying

correct system functionality. The main PCB components are described in detail in the following sections, while the PCB schematics can be found in Appendix B.

5.3.1 FPGA and Configuration

The design blocks described in the previous sections are all VHDL components implementable on an FPGA. It was decided to use the Cyclone family of FPGA's, specifically the EP1C12Q240C8 device, as this was recommended by the Quartus II development software. This recommendation resulted from the compilation of the full design, which gave a true reflection of the amount of LE's, user I/O pins and internal RAM needed in the FPGA for the design to fit. The size of the selected FPGA is such that future modifications or expansions would be possible. Another advantage is that this device does not require any external components apart from decoupling capacitors for the power supply pins.

The Cyclone FPGA is an SRAM-based device and requires configuration data to be reloaded at each power-up sequence. Two configuration schemes were used in this design, namely active serial mode and JTAG programming mode. Figure 5.7 shows how the two configuration schemes are connected to the FPGA.

For active serial programming, a serial configuration device is required. This enables the configuration data to be stored and downloaded to the FPGA during the power cycle. The EPCS4 configuration device was selected as it has a capacity of 4Mbits and was recommended for use with the selected FPGA. This device is also packaged in a small 8-pin SOIC package, which contributes to area saving on the PCB. There are four signals on the serial configuration device that interface directly with the Cyclone device's control signals. The serial configuration device signals DATA, DCLK, ASDI and nCS interface with DATAO, DCLK, ADSO and nCSO control signals on the Cyclone device, respectively. The configuration device is programmed via the Altera ByteBlaster II download cable, which is controlled by the Quartus II development software.

The serial configuration works as follows: The FPGA acts as the configuration master and provides the clock signal to the serial configuration device. Firstly, the FPGA device enables the serial configuration device by pulling the nCS signal low via the nCSO signal. Subsequently, the FPGA sends instructions and addresses to the serial configuration device via the ASDO signal. The serial configuration device responds to this by sending the configuration data to the FPGA's DATAO pin on the falling edge of DCLK. Finally, the data is latched into the FPGA on the DCLK signal's rising edge.

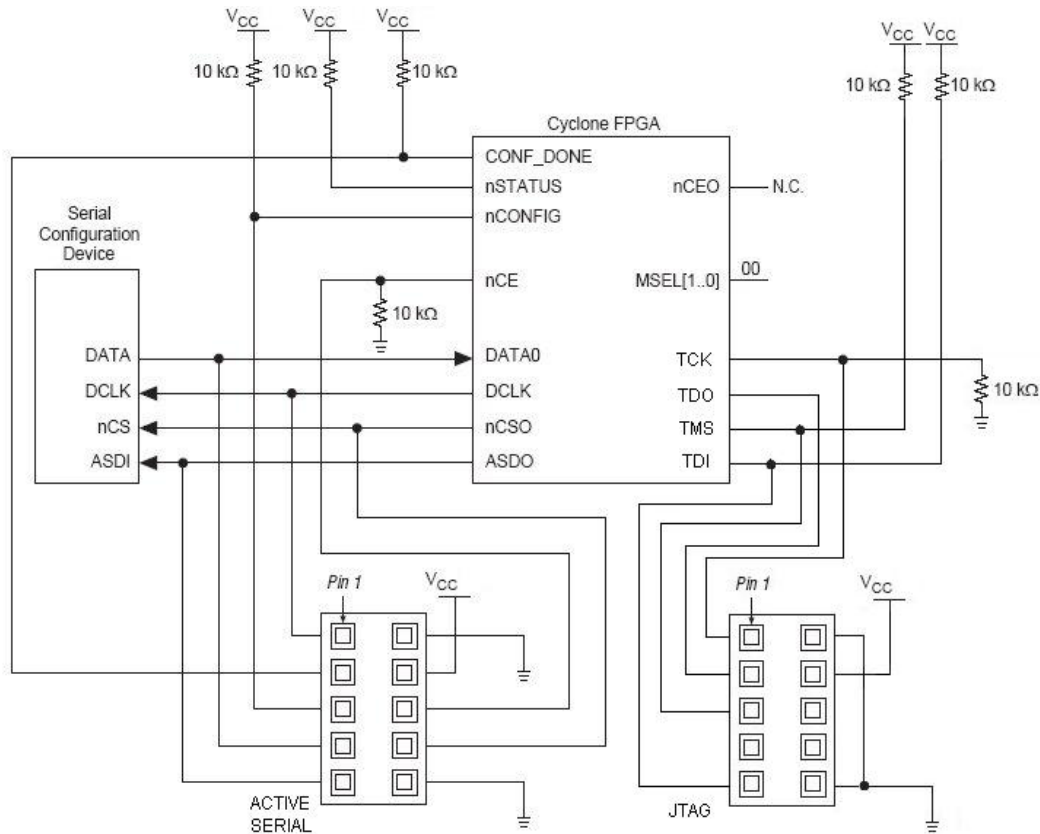


Figure 5.7: *FPGA Configuration*

The other method for configuring the Cyclone FPGA is the JTAG programming mode. The FPGA contains a dedicated set of JTAG control lines. These lines are brought to a 10-pin IDC header with pins compatible with the Altera ByteBlaster II download cable, which is used to download a new configuration directly to the FPGA through the JTAG port. Configuration data loaded by the JTAG port is only available until the power is removed. This means configuration data must then be reloaded at the next power-up.

The I/O lines of the FPGA are used for the following purposes:

NAND flash control and data lines - A set of control and data lines are needed for each bus. Separate chip enable and ready/busy lines for each device on a bus are also required.

Status LED's - Five LED's are to be used to signal the current status of the board.

External Interfacing and debugging - 24 I/O lines are connected to a 16-pin and 8-pin IDC header for external interfacing and for routing any debug signals from the FPGA.

5.3.2 NAND Flash Devices

16 NAND flash devices were required to fit onto the PCB. Four sets of data and control lines from the FPGA were therefore used to split the devices into separate buses, each bus containing four devices. Another requirement is for each device to contain its own chip enable and ready/busy lines, so that each device on a bus can be selected individually and each device's status can be monitored. The ready/busy line is an open-drain driver. It requires a pull-up resistor, R_p , connected to V_{cc} and a decoupling capacitor, C_L , connected to Gnd. A $1k\Omega$ resistor and a $100pF$ capacitor were selected according to the Samsung K9F1208U0A datasheet [8].

5.3.3 Power Supply

The FPGA requires two different supply voltages: a 3.3V supply for the I/O pins, V_{IO} , and a 1.5V supply for the internal logic, V_{INT} . Two linear voltage regulators, LM1086CT-3.3 and 5203A-2 (ON) were used to generate the 3.3V and 1.5V respectively. In order to allow current to only flow in one direction a diode was inserted across the supply pins. Additionally, a green LED was placed on the power supply terminal to indicate that the supply voltage is present. Separate layers were used for the 3.3V supply and ground planes. These are sandwiched between the top and bottom signal layers, while the 1.5V supply to the FPGA is routed on the bottom signal layer. Figure 5.8 shows the PCB layer structure.

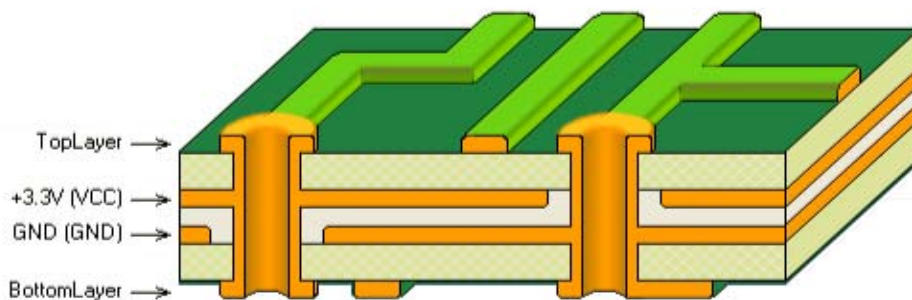


Figure 5.8: *PCB Layers*

5.3.4 Decoupling

To facilitate power supply noise reduction and the fast internal and external transitions required by the FPGA, decoupling capacitors were used. During high speed transitions, decoupling capacitors provide an instant source of current the period before the power supply can respond. $100nF$ decoupling capacitors, as recommended by Altera for Cyclone

devices [2], were therefore placed on all VCC/GND and VINT/GND pairs on the FPGA, and also on VCC/GND pairs for each NAND flash device. Care was taken to place these as close to the pins as possible, to minimize the area of the current loop produced. The larger the area of the current loop the larger the inductance. The high frequency impedance produced by this inductance adds to the other circuit-impedance elements and reduces the effectiveness of the power-supply decoupling.

5.3.5 LEDs and Test Points

Status LED's and multiple test points were added to the PCB design for debugging purposes. Test points were added to the control lines *we*, *ale*, *cle* and *re* for each bus. The status LED's are used for the following purposes:

- provision of a 'heartbeat' to show that the design is downloaded to the FPGA correctly.
- indication if the board is in idle or busy state. If the data router main state machine is in the idle state, the board is ready for the next command, otherwise it is busy with another operation.
- indication whether a read, write, or erase operation is currently taking place.

Resistors were added in series with the LED's for current limiting purposes.

5.3.6 Clk and Reset

The FPGA requires an external clock source to execute the synchronous design. An 80MHz oscillator was chosen as this clock source. To generate the required clock frequencies of 25MHz and 50MHz required for the design, an internal PLL of the Cyclone FPGA was used. A reset switch was also added to enable the internal state machines to be reset either at power-up or in the event of the system falling into an unknown state during development.

5.3.7 Device Placement and Routing

In order to keep the tracks lengths to a minimum it was decided to place eight NAND flash devices on the top and eight NAND flash devices on the bottom of the PCB. As the devices on the bottom are directly underneath the devices on the top, the tracks are routed to the bottom layer through vias. The flash devices were placed towards the edge of the board to allow future expansion of more devices on each bus. Additionally, the choice was made to place the FPGA at a 45 degree angle with the PCB edge. Not only does this

reduce track lengths, it also avoids 90 degree corners, which introduce excess capacitance and can cause a small change in characteristic impedance of the track. Figures 5.9 and 5.10 show the final layout for the top and bottom of the PCB as completed in Protel 99 SE. The FPGA is shown in blue and the 16 NAND flash devices are shown in red.

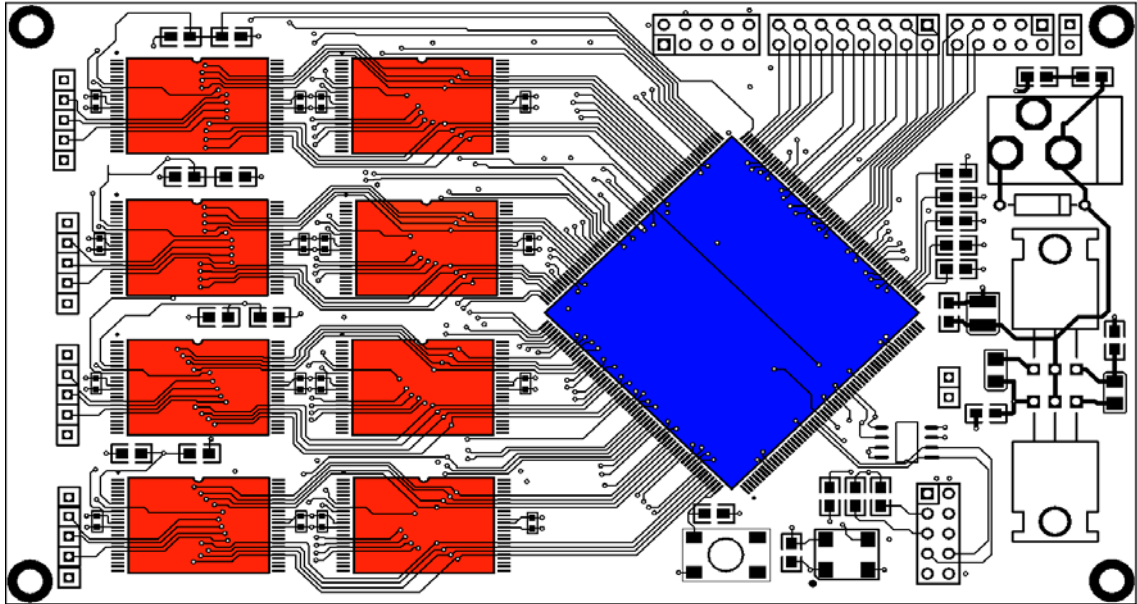


Figure 5.9: *Device Layout and Routing Top*

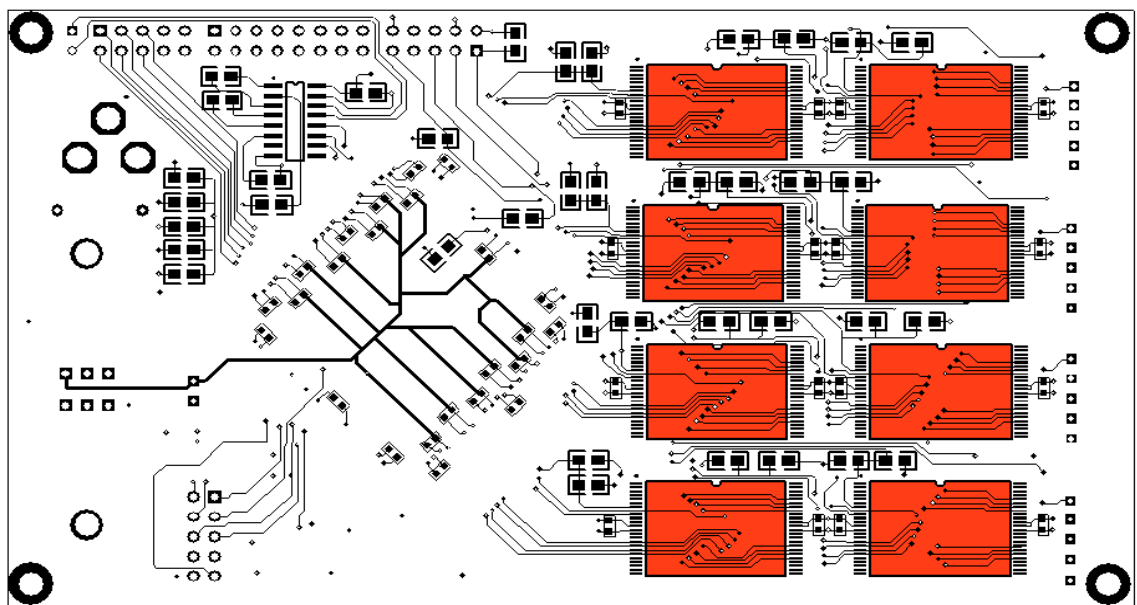


Figure 5.10: *Device Layout Routing Bottom*

5.4 Read While Write

The possibility of the memory system reading and writing simultaneously was the last problem to investigate. This was seen as a separate issue to the main MMU design.

5.4.1 Motivation

The need for the MMU to have this extra functionality arose from the fact that the images could not be downloaded to the ground station while an image of a local area was being taken. This function had also never been achieved in previous Sunsat MMU's and would increase the efficiency of the MMU and the satellite.

5.4.2 Design Solution

It was decided that the current system with the Samsung K9F1208U0A device would make a simultaneous read/write implementation nearly impossible because 14 of the 16 devices could be busy at one time during an image write process. The other alternative was to use a Samsung device with a 2112 Byte page register, the Samsung K9K2G16U0A. This would mean that only six of the devices would be busy at one time during an image write sequence and the other 10 would be idle. Equation 5.2 shows the number of busy devices for a 2112 byte page register storing a 2048 byte page with 24 bytes for ECC.

$$\text{Number of Devices Busy} = 1 + \frac{700\mu}{80n \times 2072} = 6 \text{ devices} \quad (5.2)$$

As only six devices are needed to sustain the write data rate, the system could be split up into two main areas, each controlling two buses with four devices. This would mean that when one area is writing, the other area can support a read operation. The two areas are completely independent of each other and variable read data rates are possible. The proposed modifications to the main MMU design are shown in Figure 5.11.

The modified system contains two data routers. Each data router is controller by either the four most or four least significant bits of the command input lines. A multiplexer was required to connect the output data lines to the data router executing the image read operation, and the input data lines to the data router executing the image write sequence.

The main system VHDL code was also modified to deal with the increased page size. The EDAC unit was the most affected by the changes, as the FIFO had to be increased to 2048 bytes in size, and the ECC generator had to produce 24 ECC bytes for every page instead of six bytes as before.

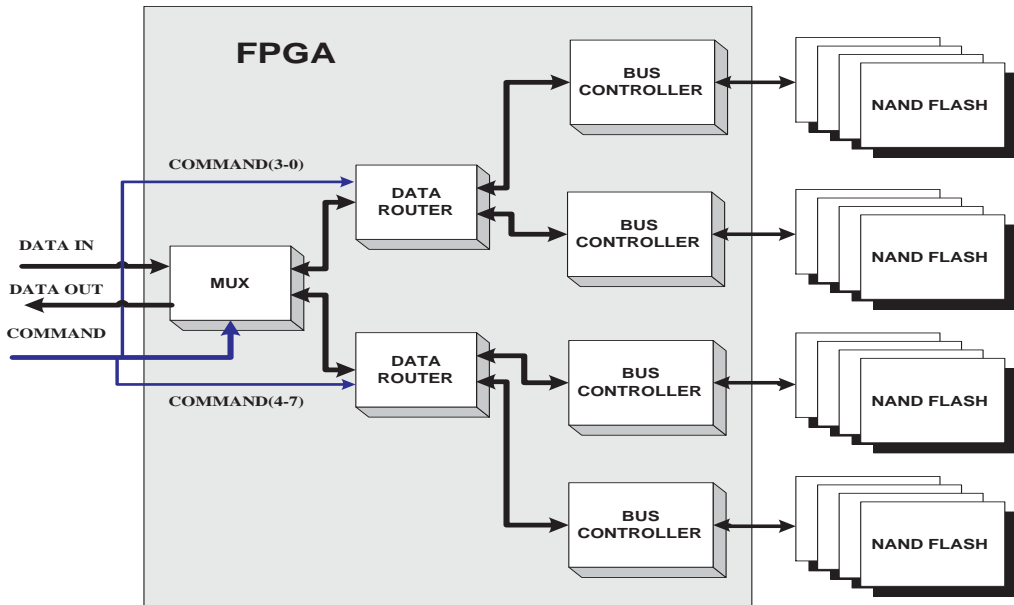


Figure 5.11: *Read-While-Write System Structure*

It was also decided to divide each memory area into two image sections of 16376 and 16384 blocks respectively. As a result, four 4 complete images can be stored on the whole system at any time. The new memory structure is shown in Figure 5.12.

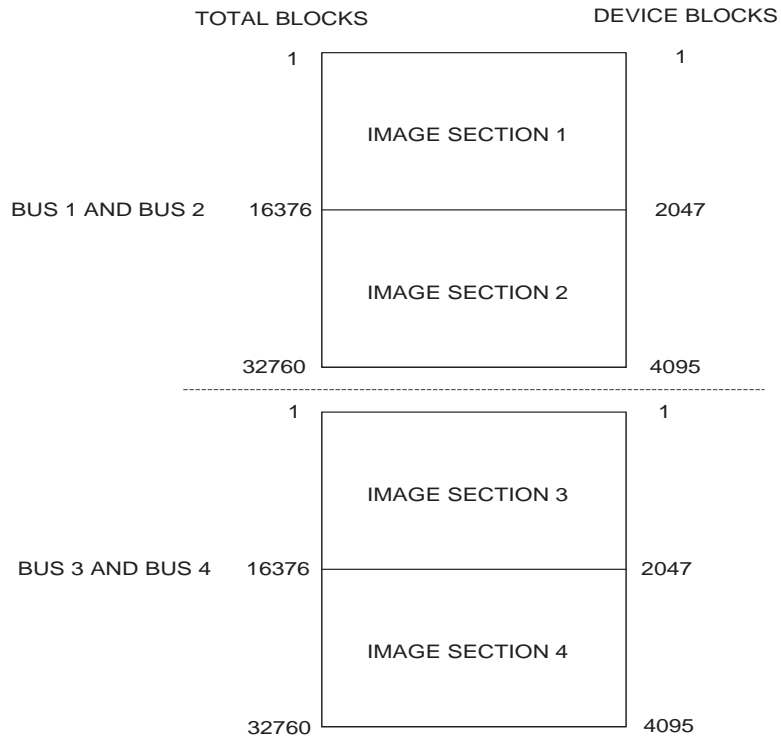


Figure 5.12: *Read-While-Write Memory Structure*

Chapter 6

Simulations and Results

As the previous chapter described the design of the various VHDL components in the system, this chapter starts by focusing on the simulation results of the VHDL components and the fully integrated VHDL system. Finally, the integration of the VHDL design into the demonstration printed circuit board is illustrated by the measured results.

6.1 VHDL Simulations

The correct functionality of the VHDL components can be simulated before they are downloaded into the physical FPGA. This eases system development as signals hidden to the PCB can be monitored within the simulator. Simulation software Modelsim SE 5.8c was used to simulate the design at various stages of development. All the simulations shown in this chapter are gate-level timing simulations. Quartus II produces a VHDL netlist file (VHO) and a standard delay format output file (SDO) for the design and these files are imported into Modelsim to accurately simulate the gate-level timing characteristics of the design.

6.2 NAND Flash Simulation Model

A simulation model for the Samsung K9F1208U0A NAND flash device was downloaded from the Samsung website [6]. This model was used to simulate the exact operation of the NAND flash devices. A VHDL testbench¹ was written to attach the NAND flash models to the rest of the system. The ability to simulate the full design allowed any small timing problems to be resolved quickly and provided an alternative source of timing information aside from the data sheet. As the downloaded model consists of pre-compiled files, access to the source files was not possible. The programming time t_{PROG} , block

¹VHDL program for simulations purposes only. It is used to create test stimulus for the design.

erase time, $tBERS$ and read transfer time tR , were all included in the model as generics, which means the values of these delays could be altered within Modelsim to practical values for simulation purposes.

6.2.1 Basic Operations

The three basic interface operations, write, read and erase, were the first to be simulated and checked for correct functionality. Figure 6.2 shows the write (top), read (middle) and erase (bottom) simulations. For the write and read simulations, page 0 of block 1 on device 0 of bus 0 is programmed and read consecutively. For the erase operation, block 1 on device 0 of bus 0 is erased.

Write : First the ce line of device 0 on bus 0 is pulled low before the commands 00H and 80H are written to the device. The address is written next in four cycles: The first cycle is for the column address, which is always 00H to start at the first byte on a page. The second cycle is for the page (first 5 bits) and the start of the block address (last 3 bits). The third cycle and the first bit of the fourth cycle represent the rest of the block address. Since block 1 is to be programmed, the first bit of the block address is a '1' giving 20H on the second address cycle. The data to be stored, in this case generated by the test generator unit, is then written to the flash device one byte at a time. Once all 512 data bytes and six ECC bytes have been written to the flash device, the 10H command is written to mark the end of the page and initiate the program operation. The $rb\bar{b}$ line of device 0 is pulled low by the device itself to indicate that it is in a busy state. This line then returns high when the program operation is complete.

Read: The ce pin of device 0 on bus 0 is pulled low and the command 00H is written to the flash device. The four address cycles follow, after which the device goes into a busy state shown by $rb\bar{b}$ being pulled low. When $rb\bar{b}$ returns high, the data is read from the flash device one byte at a time by toggling the bus re pin. From the simulation it can be seen that the data read out is the same data that was stored in the previous write operation.

Erase: The ce pin of device 0 on bus 0 is pulled low and the command 60H is written to the flash device. The three address cycles follow for the block address, as the column and page address not needed, after which command D0H is written to the device. The device then goes into a busy state ($rb\bar{b}$ low) and then back to a ready state ($rb\bar{b}$ high) after a period of $tBERS$ (set to 200 ns for this simulation). Once the erase operation is complete a read status register operation commences. This is initiated with a 70H command, after which the data is accessed by toggling the re pin. The status register value seen in the

simulation is C0H which indicates that the block is still good after the erase operation.

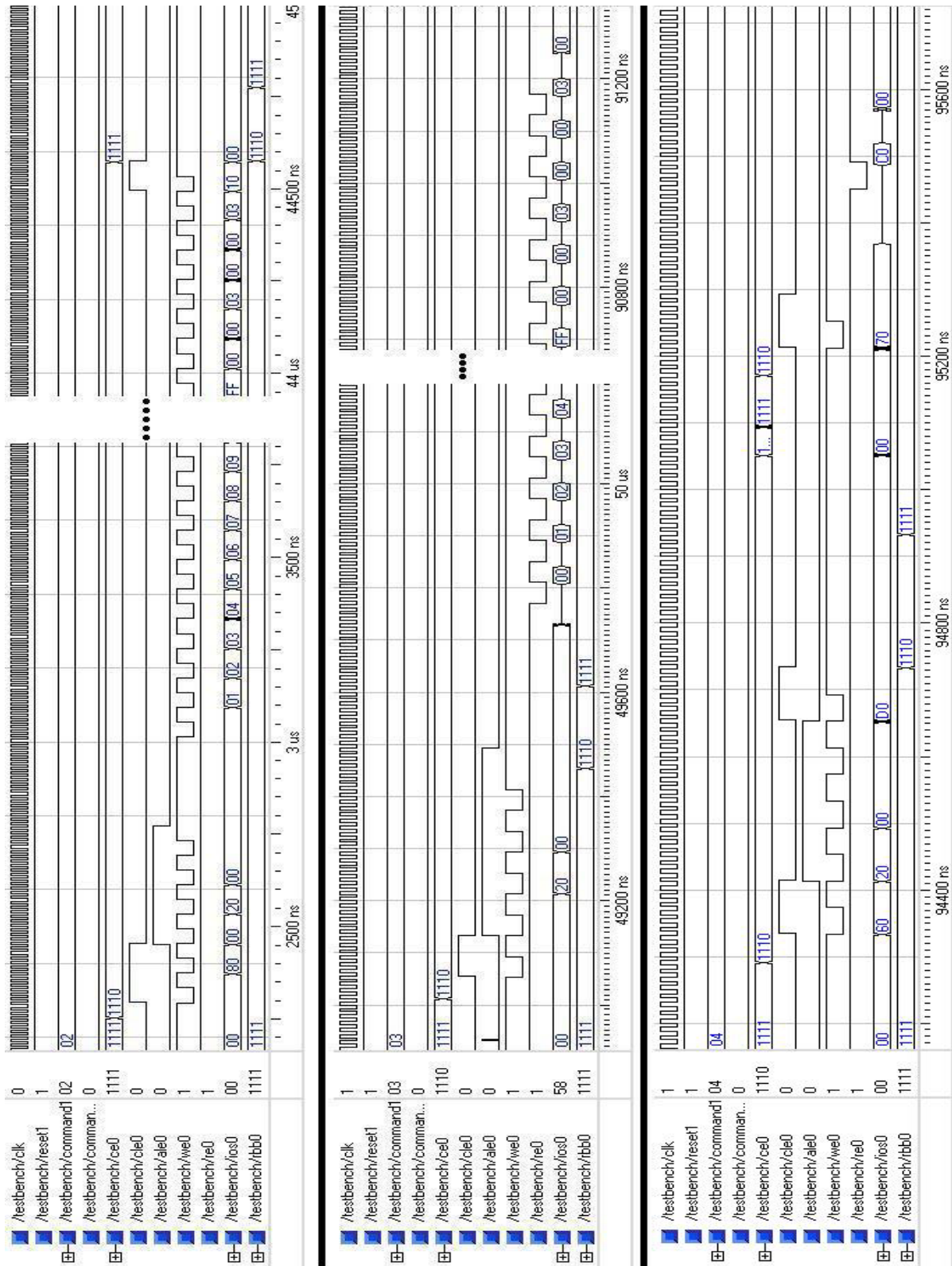


Figure 6.1: VHDL Simulation - Write, Read and Erase

6.2.2 Bad Block Table

Next on the verification agenda was the loading and storing of the bad block table. Figures 6.2 and 6.3 show the store and load simulations respectively. The load operation involves all four buses reading the first two pages from block 0 of device 0 and saving them to the FPGA internal RAM, the store operation writes the bad block table from internal RAM into flash memory. If any new bad blocks have occurred during an erase sequence, the bad block addresses are also reordered at this stage.

The test setup used for the store and load simulations is shown in Table 6.1. The present bad block addresses represent the bad block addresses currently stored in the normal area of the bad block table, and the new bad blocks addresses represent addresses of any new bad blocks which have occurred during an erase sequence and are stored in the spare area of the bad block table.

Table 6.1: *Bad Block Addresses for Load and Store Simulations*

| BUS | PRESENT BAD BLOCKS ADDRESSES | NEW BAD BLOCKS ADDRESSES |
|-----|------------------------------|--------------------------|
| 0 | 02H,05H | 03H |
| 1 | 02H,05H | 01H,06H |
| 2 | 02H,05H | 06H |
| 3 | 02H,05H | 01H,03H |

Store: The store command 05H on the *command* input lines together with the rising edge of the *command_strobe* input start the store sequence on all four buses. From the simulation it can be seen that all bad block addresses on each bus are written to flash in numerical order and have been reordered correctly ². The rest of the data to be stored in flash after all the bad block addresses are written, are blank normal area addresses 7FFFH, or blank spare area addresses FFFFH. Only the normal area is shown in the simulation.

Load: The load simulation shows the bad block addresses being read from flash memory. These bad block addresses are then stored in the internal RAM of the FPGA. The addresses are also routed to the data router in order to make the amount of bad blocks on each device on a bus known to the data router. Additionally, the bad block table is sent out through the UART. It can be seen in the simulation that the addresses read out

²Addresses are 16 bits long, so 02H = 0002H when written to flash.

6.2.3 Image Write Sequence

The simulation shown in Figure 6.4 is an image write sequence. It shows the first 16 pages of test data that were written to the flash memory. The test data generated by the test generator unit, starts at 00H and increments each next byte by one. Since each page is 512 bytes in size, 00H to FFH is stored twice on each page. The last six bytes stored are the three ECC bytes for each half of the page. The blocks of high activity shown on *we* and *ios* of each bus is where the data is actually written to the target flash device.

The write cycle time in the simulation is 80ns. This gives a maximum write data rate of 11.92MB/s, which the system can sustain without dropping or losing any data. The zoomed in section shown at the bottom of Figure 6.4 shows the switch over from one bus to the next. When bus 0 has written the last byte of test data to be placed in that particular page, FFH, the next byte, 00H, is written to bus 1. Bus 0 continues to write the 6 ECC bytes after the last byte of test data is written.

The 16 pages of data stored were written to 16 different devices. This can be seen by looking closely at the *ce* lines of each bus. The setup cycles for each page are executed just after the start of writing of the previous page. This can be seen by looking at the *ale* spikes on the simulation. These spikes indicate that an address is being written to the flash device.

After a device has been selected and a page of data has been written to that device, the device goes into a busy state. The simulation shows that the ready/busy line, *rbb*, is low for 200 μ s (typical value for *tPROG*) before returning high. During this period, five other devices are accessed in the write sequence. If *tPROG* was set to 500 μ s (the maximum value for *tPROG*) then 13 other devices would have been accessed in the write sequence before the initial device had returned to a ready state. This shows that a minimum number of devices must be available to sustain the data rate.

The time it takes to write an image section is calculated below from the following parameters: image section size = 13104 blocks; pages in a block = 32 pages; write cycle time = 80ns; size of a page = 512 bytes; *tPROG* = 500 μ s; therefore,

$$\text{Image Write Time} = 13104 \times 32 \times (80\text{ns} \times 512) + 500\mu\text{s} = 17.2 \text{ s} \quad (6.1)$$

6.2.4 Image Read Sequence

The simulation shown in Figure 6.5 is one of an image read sequence. It shows the first 16 pages being read from the flash memory device array and the complete output stream *data_out0* that is built up from the individual pages. The test data read out is the same test data that was written to the flash in the previous simulation - 00H through to FFH is stored twice on each page. The blocks of high activity in the simulation show a page of data being read from the flash memory, appearing on the *re* and *ios* lines.

The magnified in section at the bottom of the simulation shows the complete output data stream. The byte FFH is the end of one page of data and the byte 00H is the start of the next page. From the smooth transition, it can be seen that there is no lost data or any timing problems at this critical point. The read cycle time out of the flash device is 80ns, while the data is read out of the system at 120ns (as stated in Section 5.2.1.1). This gives a maximum read data rate of 7.95MB/s.

The 16 pages of data were read from 16 different devices. The pages are stored in their respective EDAC FIFO until the data router signals them to be read. In order to always have a page of data in the FIFO, ready to be read, the second page is read a few clock cycles after the first page.

The busy time for a read access tR is set to the maximum value of $12\mu s$ for the simulation.

The total time to read an image section can be calculated from the following parameters: image section size = 13104 blocks; pages in a block = 32 pages; read cycle time = 120ns; size of a page = 512 bytes; $tR = 12\mu s$; therefore,

$$\text{Image Read Time} = 12\mu s + 13104 \times 32 \times (120ns \times 512) = 25.8 s \quad (6.2)$$

6.2.5 Image Erase Sequence

Shown in Figure 6.6 is an image erase simulation. It shows the first 16 blocks in the sequence being erased. The whole sequence erases the total amount of blocks in the image section, i.e. 13104 blocks when all devices are fully functional.

The erase operation on each block consists of two separate operations, namely erase and read status register. The next erase only starts once the previous erase has finished and the status register has been read. As the erase sequence on the satellite is not time dependent (it can be done after an image has been read and downloaded to the ground station), the system can erase one block at one time as opposed to all four buses erasing at the same time. The erase sequence follows the same bus select algorithm as the image write and read operations, which means the same blocks are erased that were written to.

The 16 blocks erased are located on 16 different devices. The block erase time $tBERS$ is set to $2\mu s$ for the purposes of the simulation. If the erase time were the maximum value of 3ms stated in the data sheet [8], the total time to erase a whole image section using the maximum block erase value would be:

$$\text{Image Erase Time} = 13104 \times 3ms = 39.3 s \quad (6.3)$$

6.2.6 Device Deactivation

The next simulation in Figure 6.7 shows a write sequence to image section 4 with all the devices on bus 2 deactivated. Omitted from Figure 6.7 is the load command for section 4 (0CH) and the device deactivation commands. The magnified section of the simulation shows the start address for image section 4. The start address is C0H-99H-01H which equals page 0 of block 3278, the start address of each device when one bus is deactivated. As mentioned before, when a device is deactivated the image section boundaries are changed for each device, this enables whole pictures to be stored in spite of a reduction in the memory capacity.

The simulation shows that the deactivated bus 2 is skipped by the data router during the image write sequence. It is also illustrated how the data rate is maintained with fewer active devices.

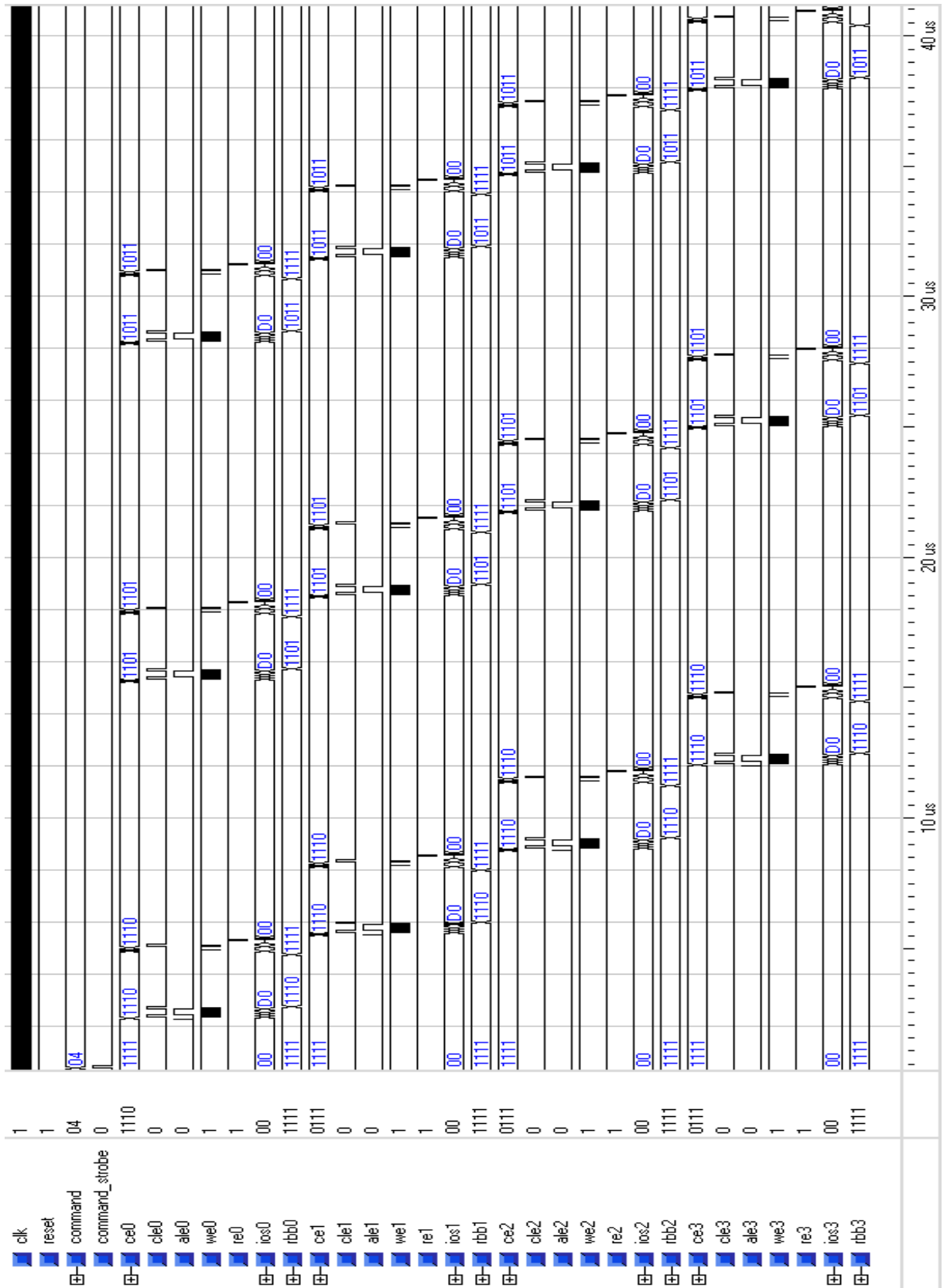


Figure 6.6: VHDL Simulation - Erase Sequence

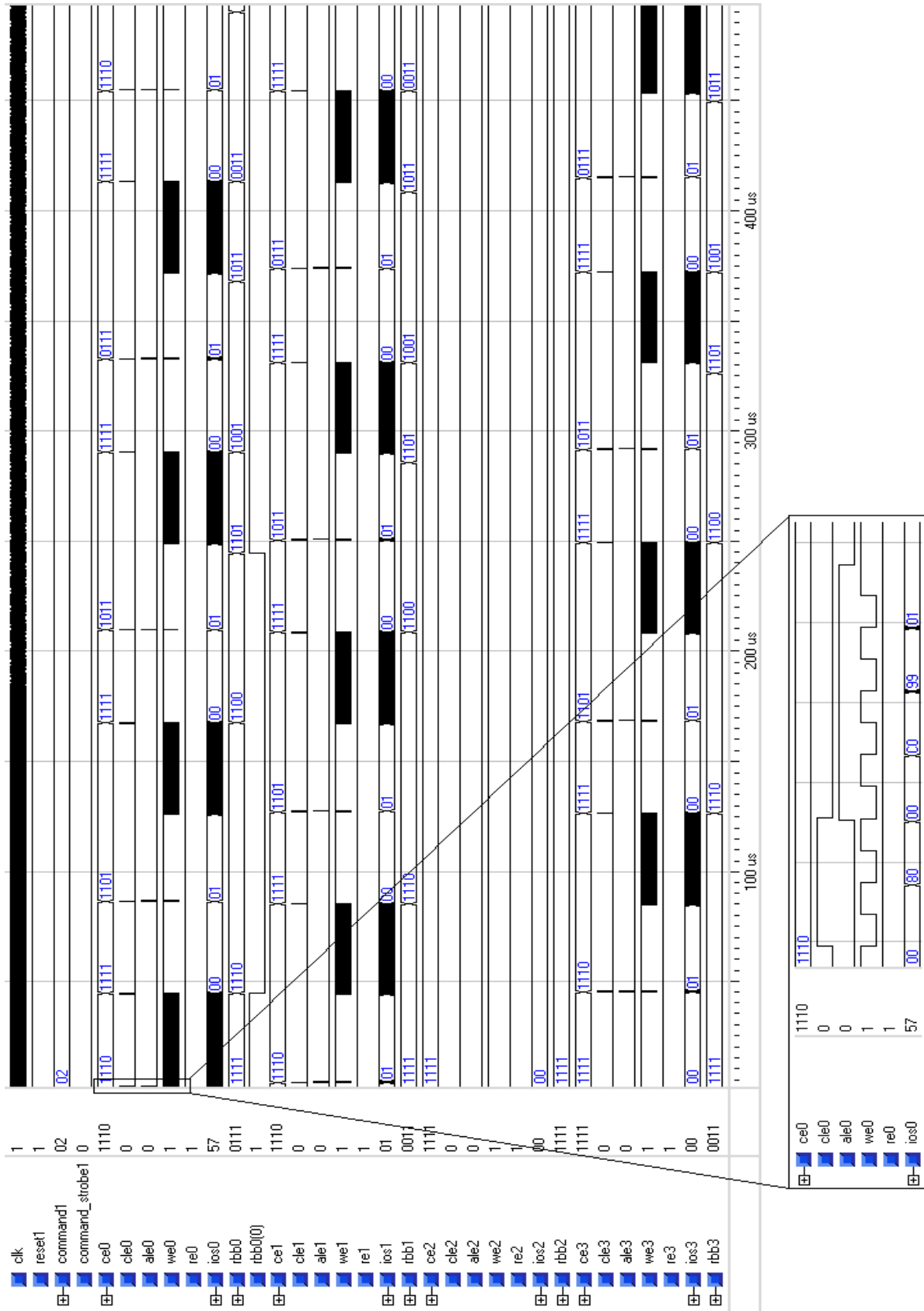


Figure 6.7: VHDL Simulation - Write Sequence with Bus 2 Deactivated

6.2.7 Error Detection and Correction

The simulations shown in Figures 6.8 and 6.9 verify the correct functionality of the error detection and correction unit. The simulations show a detection and correction of a bit error during a bad block load operation.

Figure 6.8 shows the start of the bad block load operation. The bit error is contained within the second byte read from flash. The value read is 82H instead of 02H.

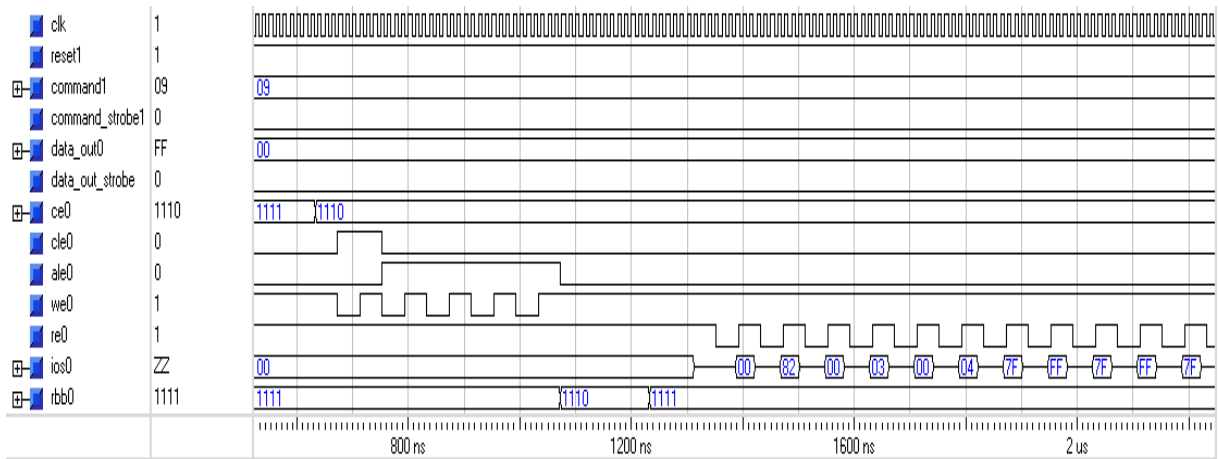


Figure 6.8: *VHDL Simulation - Load Bad Block Table ECC Detection*

In Figure 6.9 the last six bytes read out from the end of the bad block read operation are 59H, 55H, 9BH, 00H, 00H, and 03H. The first three bytes represent the ECC code word for the first 256 bytes of the page and are used to find the error and correct it. The output data stream *data_out0* proves that the error has been corrected, as the second byte is now 02H.

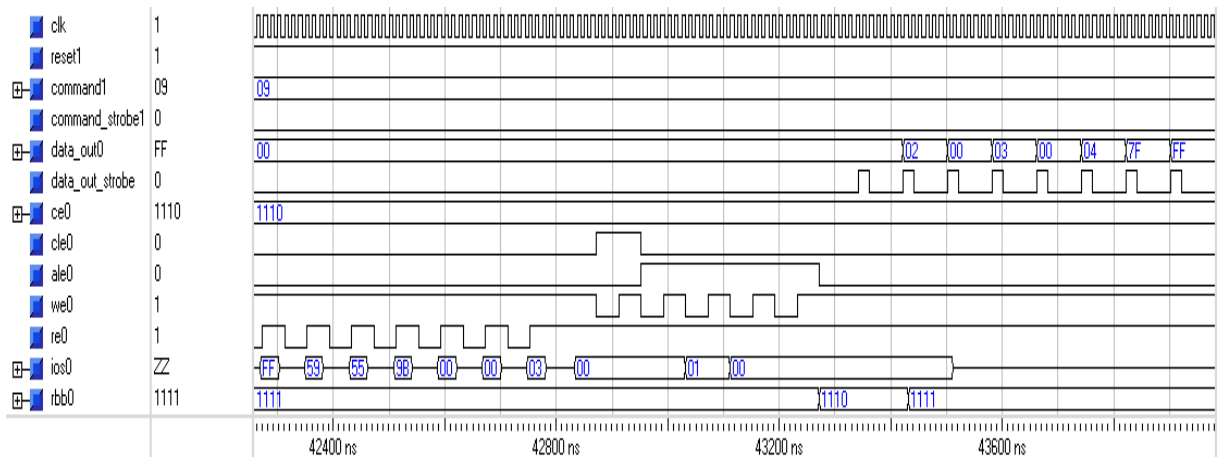


Figure 6.9: *VHDL Simulation - Load Bad Block Table ECC Correction*

6.3 Demonstration Board Testing and Results

After the full VHDL design had been simulated and verified while connected to the NAND flash models in the Modelsim environment, the demonstration PCB was built and tested.

The first test when the blank PCB arrived was to check the power and ground nets for short circuits. Next, the two voltage regulators and companion components were soldered to the board and their output voltages were checked. The output from the two voltage regulators was found to be incorrect and closer inspection found that there was a problem with the footprint of the two regulators: The input pin and the output pin were the wrong way around. After switching the pins back, the output voltages were checked to be correct. Finally, the power supply LED was fitted and verified to work properly.

Next to be fitted to the board was the 240 pin Cyclone FPGA and the JTAG port. The decoupling capacitors, reset switch, status LED's and 80MHz oscillator were also added. A small piece of VHDL code was written to check that the FPGA and JTAG configuration port were working correctly. The small piece of code simply made one of the status LED's flash on and off, to act as a so called heartbeat for the board. At first, the LED switched on but did not flash. The configuration had indeed downloaded to the FPGA. This could be confirmed because the development software did not give any errors once the download was completed. The problem was at last narrowed down to the oscillator not working properly. A certain pin had been wrongly connected to ground, causing the oscillator not to function as intended. After this was rectified LED start to flash as intended.

Finally, the big step of soldering the NAND flash devices to the board could be taken. The devices were added four at a time and checked with VHDL code that read the device ID from each device. All 16 devices were found to function correctly when they were first soldered to the board.

Figures 6.10 and 6.11 show photographs of the front and back of the demonstration board with all the components fitted.

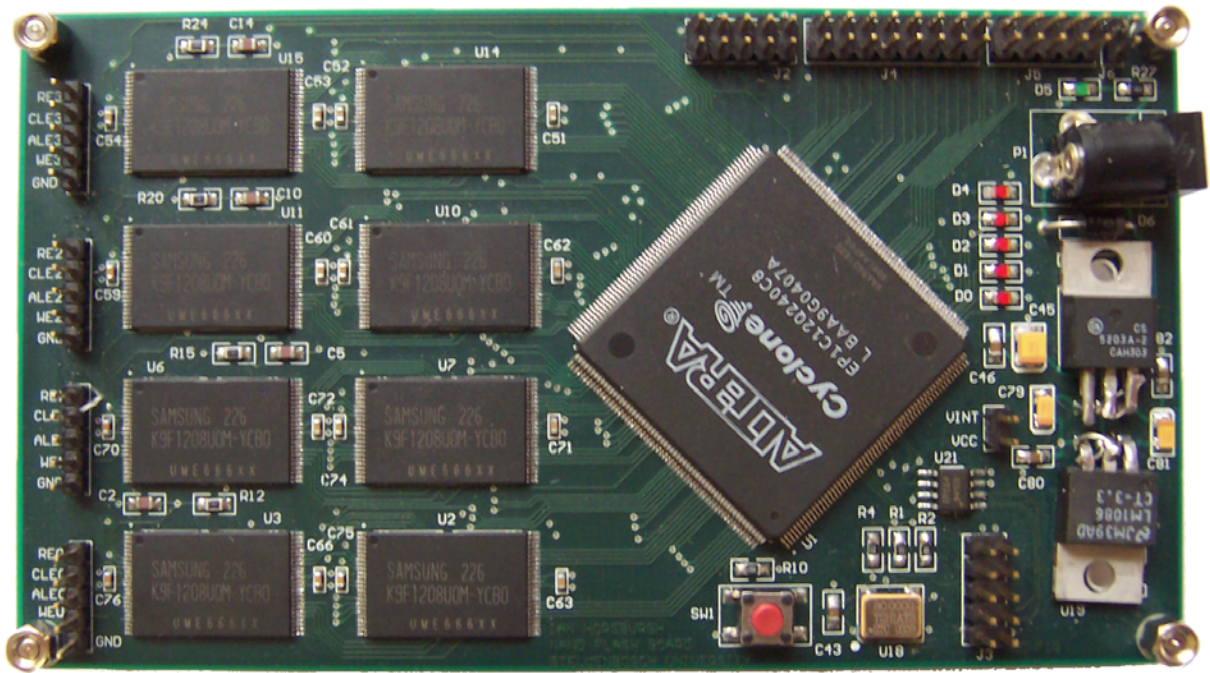


Figure 6.10: PCB - FRONT

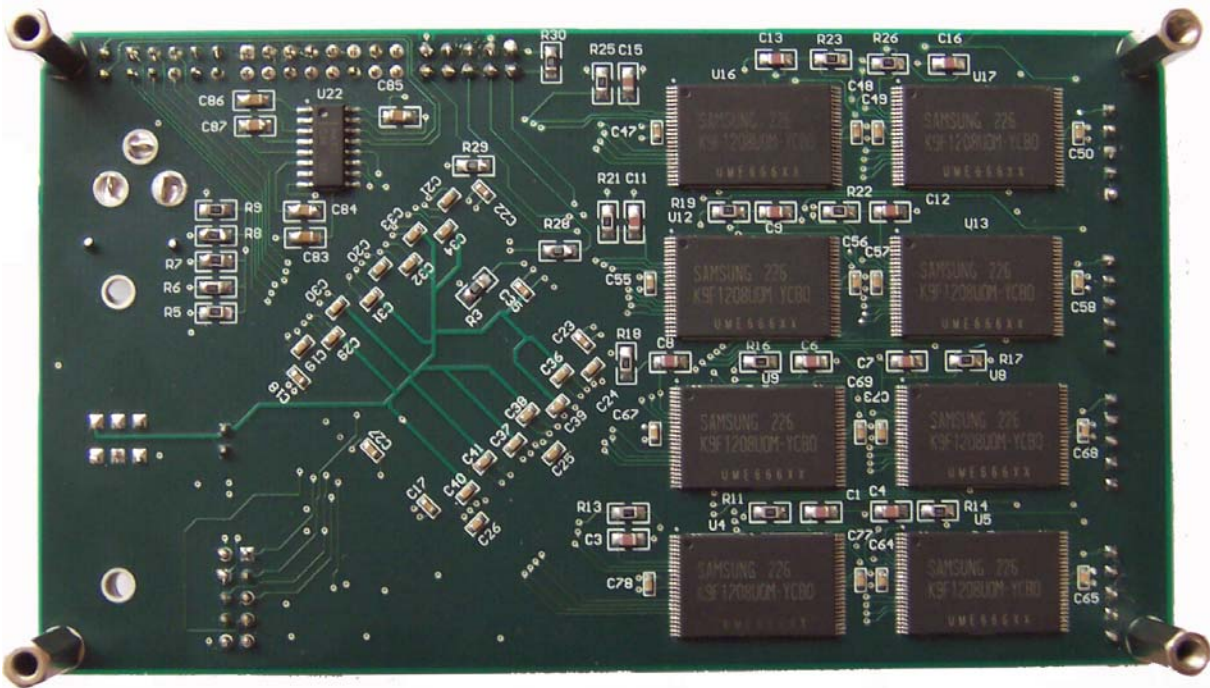


Figure 6.11: PCB - BACK

6.3.1 Factory Marked Bad Blocks

The process of reading and storing of the factory-marked bad blocks on each device was the first task to be undertaken on the fully populated PCB board. Since the bad block information can be erased, it is impossible to recover the information once it has been erased. Although an erase sequence might find some of these bad blocks, other bad blocks not as easily identified might not be found. Slight modifications were made to the full system VHDL code to generate the bad block table from the data already stored on each device and care was taken to ensure that the erase operation was disabled during this initial testing phase. Figure 6.12 shows the method for retrieving the factory-marked bad blocks. A value of FFh at column address 517 on the first or second page of block means that a block is functional.

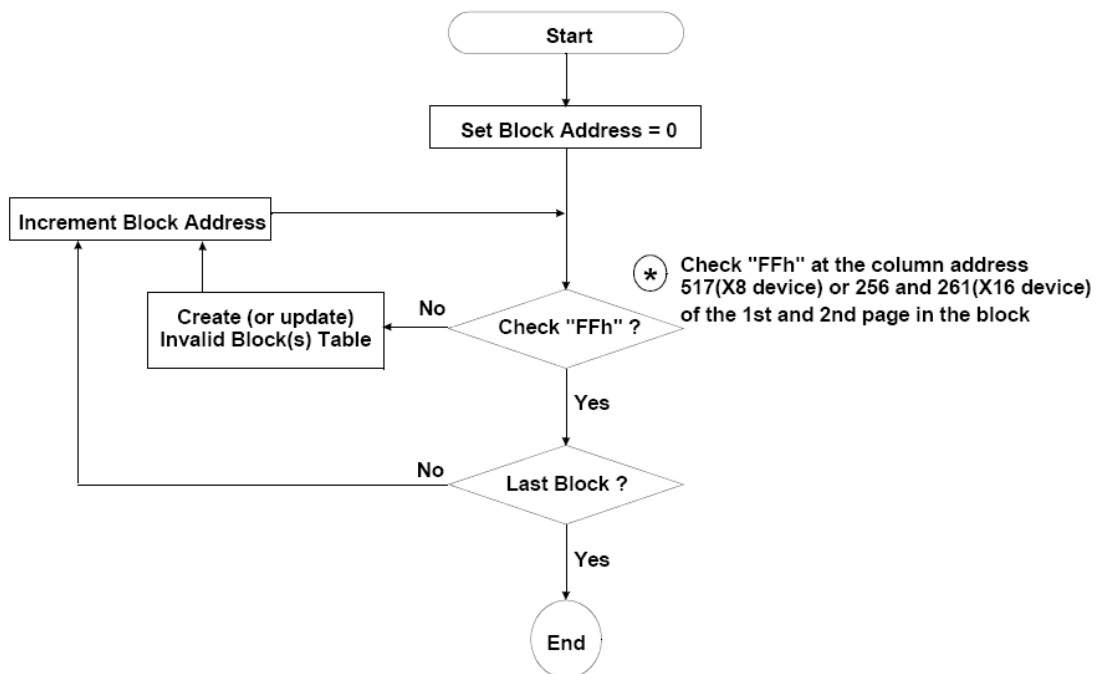


Figure 6.12: *Flow Chart - Retrieving Factory Marked Bad Blocks [8]*

The factory marked bad blocks were read out through the serial port and a total of 19 bad blocks were found scattered through the 16 devices. A number of devices contained no bad blocks at all and the greatest number on any device was 4. The bad block table for each bus was then stored in the first two pages of the first flash device on that bus. The table was subsequently read back out of flash with a load command and compared with the data that was stored. Due to the time spent debugging and verifying the VHDL simulations, the load and store operations were found to work immediately on the PCB.

Table 6.2: *Factory Marked Bad Blocks*

| BUS | DEVICE | BAD BLOCKS | ADDRESSES |
|-------|--------|------------|----------------------------|
| 0 | 0 | 0 | - |
| 0 | 1 | 3 | 0462H, 0898H, 0EF8H |
| 0 | 2 | 4 | 04B1H, 06ADH, 092BH, 0B61H |
| 0 | 3 | 2 | 014FH, 0F8DH |
| 1 | 0 | 1 | 053FH |
| 1 | 1 | 1 | 059EH |
| 1 | 2 | 0 | - |
| 1 | 3 | 0 | - |
| 2 | 0 | 1 | 07B7H |
| 2 | 1 | 0 | - |
| 2 | 2 | 1 | 0728H |
| 2 | 3 | 2 | 0ABEH, 0C17H |
| 3 | 0 | 3 | 0233H, 0A01H 0DCCH |
| 3 | 1 | 0 | - |
| 3 | 2 | 1 | 0811H |
| 3 | 3 | 0 | - |
| Total | | 19 | |

Table 6.2 shows the total number and the addresses of factory marked bad blocks on each device.

6.3.2 PCB Results

As the PCB hardware had been successfully checked and the VHDL simulations projected an error free design, the full system could now be downloaded to the demonstration board and verified for correct functionality.

The three main operation sequences, erase, write and read, were found to work just as predicted by the simulation software, except that each operation sequence stopped at random points during the testing. The problem seemed to be with the ready/busy lines from the flash devices. These lines are asynchronous inputs from the flash devices and can therefore change level during clock transitions. This was found to be the reason the whole system was freezing at random instances. After synchronising each ready/busy line to the internal FPGA clock, the problem was solved.

The demonstration board was tested extensively with multiple test patterns generated in the test generation unit. For each test the image section was first erased, read, programmed, and then read again. The test patterns for each page were as follows: all 0's, checker board (101010...), 00H to FFH, and finally the same value stored on one page but then increment by one for the next page. Each test was found to function correctly.

Since the demonstration board behaved exactly as indicated by the simulations shown earlier in this chapter, only the parts of the three main operation sequences are shown in here in Figures 6.13, 6.14, and 6.15. The data bus shown for the erase and write sequences is for the bidirectional I/O lines to the flash devices on bus 0, while the data bus shown in the read sequence is the output data stream from the whole system.

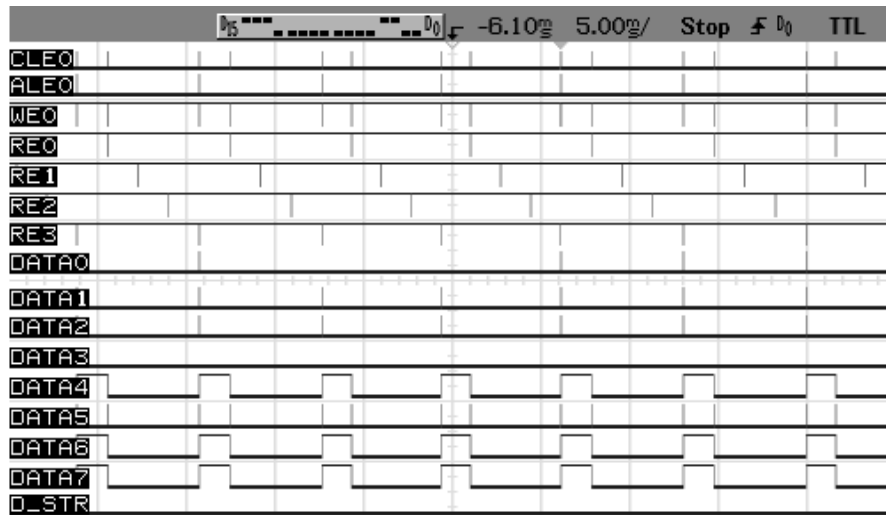


Figure 6.13: *Oscilloscope Capture - Part of Erase Sequence*

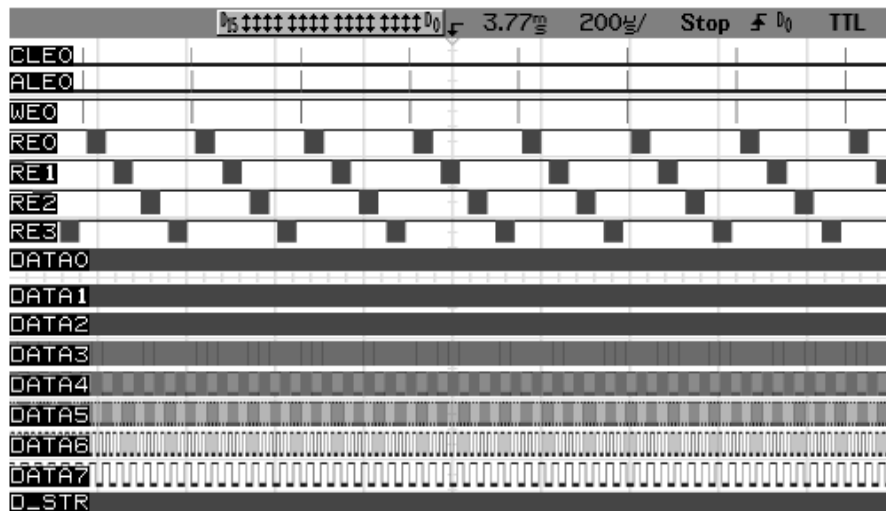


Figure 6.14: *Oscilloscope Capture - Part of Read Sequence*

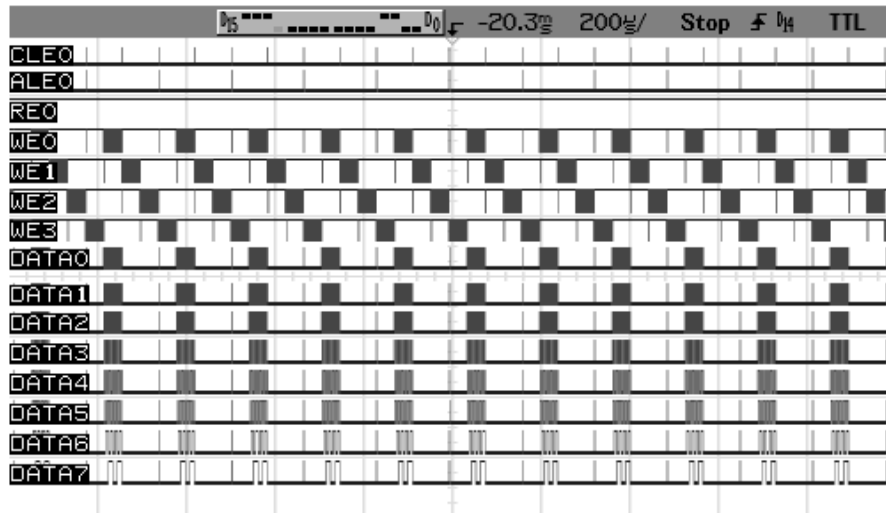


Figure 6.15: Oscilloscope Capture - Part of Write Sequence

6.3.3 Power and Area Calculations

Area

The dimensions of the demonstration PCB board are $75.6\text{mm} \times 135.1\text{mm}$. This gives an area of 102.1356 cm^2 . If three colour bands are required to be stored, the memory board could be duplicated three times giving a total area of 306.4068 cm^2 .

Power

To calculate the power consumption of the memory board, the FPGA and the NAND flash devices must be examined separately. The Quartus II development software has a built in function which generates a file that can be used to calculate the power consumption of the Cyclone FPGA. The generated file must then be imported into a Cyclone power calculator (available from [6]. Using a toggle rate of 12% (number of pins that switch at one time) the power consumption of the FPGA is shown in Table 6.3. The FPGA power-up current not listed in the table, as it is present only at startup as a once off event. It is observed in the internal FPGA current and can be a maximum of 900mA. With a 1.5V internal voltage, the maximum power required at power-up of the FPGA is equal to 1.35W.

Table 6.3: FPGA Power Consumption

| | Voltage(V) | Icc(mA) | Power(mW) |
|----------|------------|---------|-----------|
| Internal | 1.5 | 162.01 | 243.02 |
| I/O | 3.3 | 23.10 | 76.23 |
| TOTAL | N/A | 185.11 | 319.25 |

The worst case scenario regarding the power consumption for the NAND flash devices happens during a write sequence when a maximum of 13 devices are active at the same time. As flash devices have a maximum operating current 20mA [8] from a 3.3V source, the power consumed during such an event is given in Equation 6.4. The maximum power consumption of the whole board during an image write sequence is equal to the sum of the FPGA and flash devices' consumption, as given in Equation 6.5.

$$Power_{max_nand} = 3.3V \times 20mA \times 13 = 924 \text{ mW} \quad (6.4)$$

$$Power_{max_system} = 924 + 319.25 = 1.243 \text{ W} \quad (6.5)$$

This maximum value is therefore within range of the specification given in Section 3.1.5. This concludes the results for the demonstration test board. The following sections look at the simulations of the design using 16-bit NAND flash devices and also the read while write implementation.

6.3.4 16-Bit Device Write Data Rate

To show that a data rate of nearly 24 MB/s is achievable with only slight modifications to the initial design, the following simulation results were produced. The changes incorporated 16-bit 512MB NAND flash devices (Samsung K9K2G16U0M) into the design. Figure 6.16 shows the start of an image write operation and Figure 6.17 shows the first 16 pages of an image write simulation. The write cycle time is still 80ns but now 16 bits are written to the flash devices each 80ns, giving a data rate of 23.84MB/s.

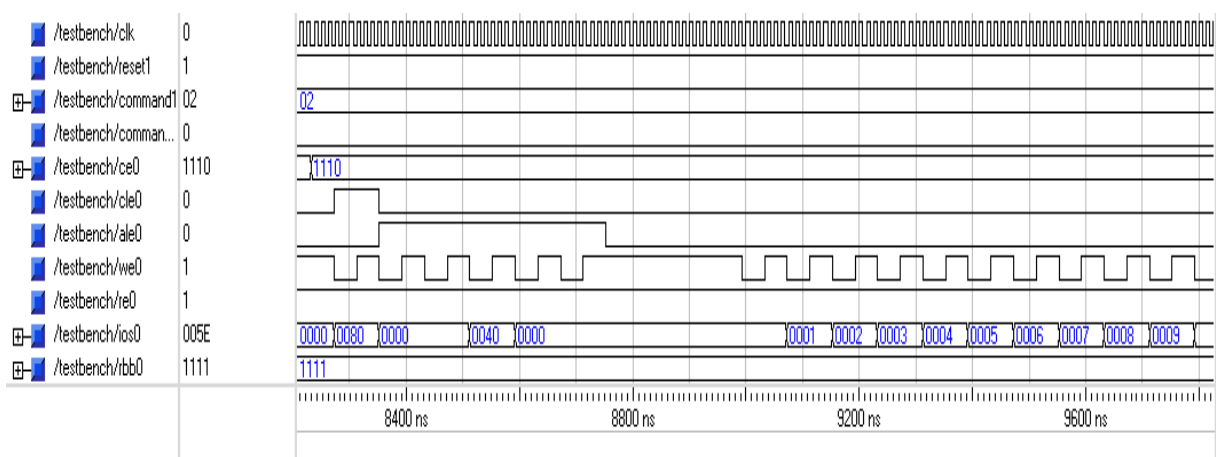


Figure 6.16: VHDL Simulation - 16-Bit Device Part Image Write Sequence

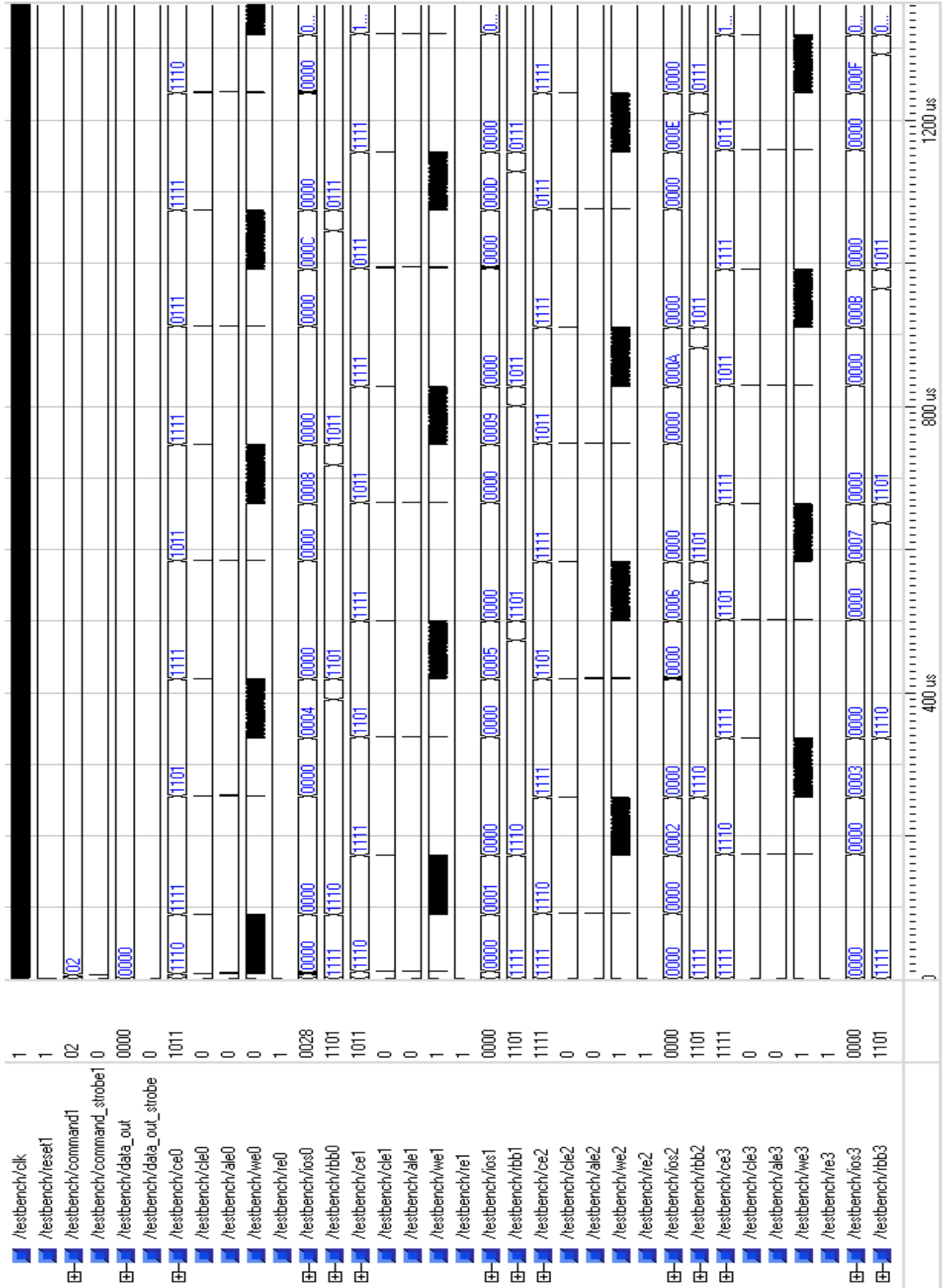


Figure 6.17: VHDL Simulation - 16-Bit Device Image Write Sequence

6.3.5 Read While Write Simulation

Once the modifications to the main demonstration board VHDL (as proposed in Section 5.4.2) were complete, the read while write functionality of the system was simulated. The simulations in Figures 6.18 and 6.19 show an image write operation and an image read operation being performed at the same time. Figure 6.18 shows an image write operation on Bus 0 and 1, and an image read operation on Bus 2 and 3. Figure 6.19 shows an image read operation on Bus 0 and 1, and an image write operation on Bus 2 and 3. Data values start at 00H and are incremented by one for each page written. The Samsung K9F2G08U0M device model was used for the simulation. The delay parameters, t_{PROG} and t_R , are set to their typical values of $300\mu s$ and $25\mu s$ respectively.

The simulations show the two separate operations running independently from one another. The output data stream, *data_out0* and *data_out_strobe*, show the correct data being read out from the MMU.

Results from VHDL simulations and practical measurements were given in this chapter. These results will be discussed briefly in the next chapter, along with recommendations for possible improvements in the future.

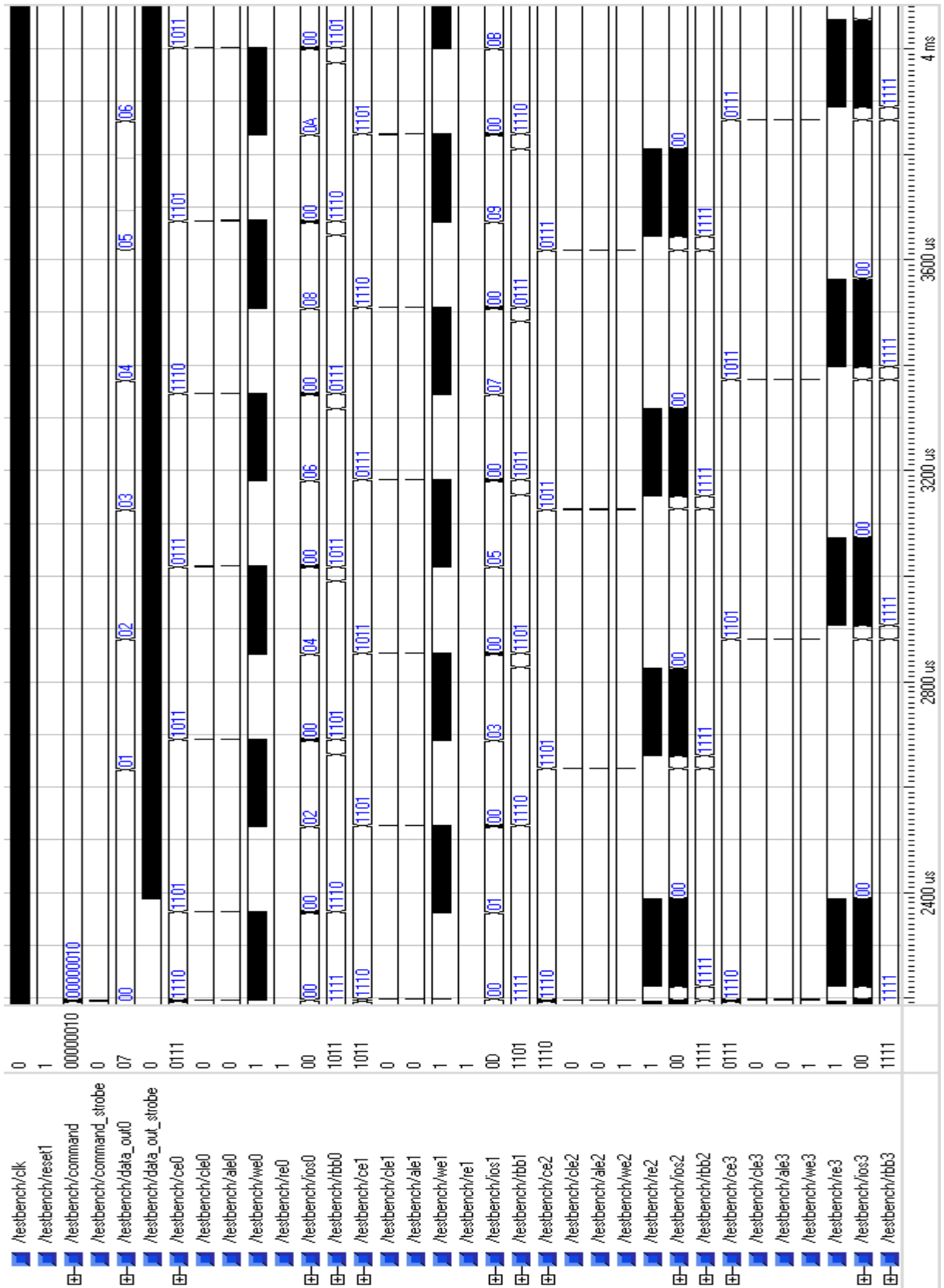


Figure 6.18: VHDL Simulation - Reading While Writing 1

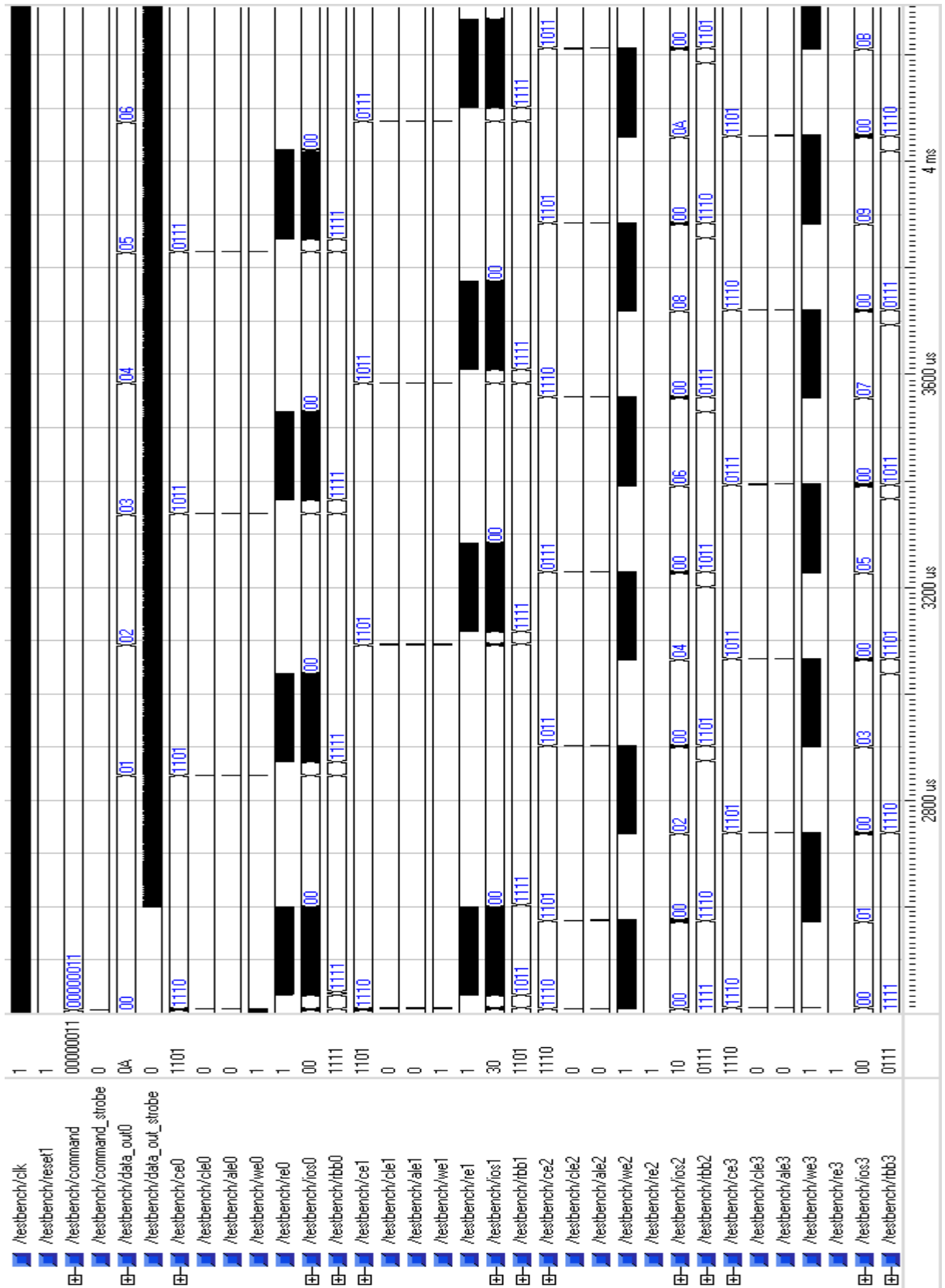


Figure 6.19: VHDL Simulation - Reading While Writing 2

Chapter 7

Conclusions And Recommendations

7.1 Conclusions

This thesis investigated the use of NAND flash memory for a MMU on a micro-satellite. Design specifications were produced, and from an evaluation of various configurations a four-bus serial architecture using 16-bit 512MB NAND flash devices was chosen. Since only 8-bit 64MB NAND flash devices were available at the time, a 1GB test board was designed and built incorporating 16 such devices. The specifications were revised and achieved with a test board capable of sustaining a write data rate of 12MB/s.

The use of NAND flash simulation models and dedicated simulation software was of vital importance to the success of the project. Functional and gate-level timing simulations allowed the VHDL design to be 99% verified before it was integrated into the test PCB. Furthermore, the simulation of 16-bit devices were possible and these simulations showed that a write data rate of 24 MB/s is achievable with the same four-bus serial architecture. Therefore, the original design specifications are satisfied.

The occurrence of bad blocks during the lifetime of a NAND flash device posed an internal address generation problem. This was solved through the use of a look-up table, which enables a 100 bad block addresses to be stored per device. However, if this number should be expected to be much higher, for instance in a radiation harsh environment, then a different scheme might have to be implemented.

A read-while-write function was added to the design at a later stage and is shown in a simulation with 8-bit NAND flash devices to be fully functional. This function, which had never been implemented on previous MMU projects, allows the satellite to store images of the area around the ground station while downloading another image.

Because the system allows multiple images to be stored, one of the key elements to the design was to maintain MMU functionality in spite of defective devices. The dynamic resizing of image section boundaries ensures that the images stored would not be reduced in size in the event of multiple flash devices failures.

Although the power consumption of the test board during an image write sequence is quite high (1.243W), it must be remembered that the memory unit is only active for a small fraction of the orbit period when images are captured.

Due to the fact that the VHDL design is written in a modular way, only small changes would be needed in order to incorporate higher density NAND flash devices. Another manner in which future expansion was taken into account, was the layout of the PCB. It is structured in a way that extra devices can be added to each bus without a serious upheaval of the board or track layout.

7.2 Recommendations

The following recommendations are made with regards to future development of a mass memory unit for a micro-satellite using NAND flash memory.

- Firstly, the inclusion of wear-levelling would ensure that certain areas of a device are not written to or erased excessively. This would lead to an extension of the device lifetime.
- Secondly, radiation testing on the demonstration board would make it possible to predict the type of errors likely to be seen in a space environment, together with the rate at which bad blocks occur.
- Finally, exhaustive functional and extreme temperature testing of the demonstration board will give valuable application-specific information about the lifetime of a device and how it behaves in extreme conditions.

Bibliography

- [1] ALTERA. *Using Modelsim-Altera in a Quartus II Design Flow*, December 2002.
- [2] ALTERA. *Cyclone FPGA Family Data Sheet*, Oct 2003.
- [3] BADENHORST, P. J., "The DEVELOPMENT of a Memory-Module for the second generation SUNSAT micro-satellite." Master's thesis, University of Stellenbosch, 1996.
- [4] BURGER, H., "Development of a Memory-Module for the SUNSAT micro-satellite." Master's thesis, University of Stellenbosch, 1996.
- [5] RUST, A. N., "A Second Generation SUNSAT RAMDISK." Master's thesis, University of Stellenbosch, 2000.
- [6] SAMSUNG ELECTRONICS. *Samsung Official Website*. [Online]. Available : <http://www.samsung.com/>.
- [7] SAMSUNG ELECTRONICS. *256M × 8 Bit, 128M × 16 Bit NAND Flash Memory Data Sheet*, April 2003.
- [8] SAMSUNG ELECTRONICS. *64M × 8 Bit, 32M × 16 Bit NAND Flash Memory Data Sheet*, July 2003.
- [9] SAMSUNG ELECTRONICS. *Nand Flash ECC Algorithm*, June 2004.
- [10] SAMSUNG SEMICONDUCTOR, INC. *Selecting the Right FLASH Partner to TURN Technology Advantages into Profits*, 2003.
- [11] STMICROELECTRONICS. *How to Connect NAND Flash Memories to Build Storage Modules*, October 2003.
- [12] STMICROELECTRONICS. *How to Use Cache Program Feature of NAND Flash Memories*, October 2003.
- [13] STMICROELECTRONICS. *Error Correction Coding in NAND Flash Memories*, May 2004. Revision 1.0.

- [14] TAL, A. and PAZ, Z., “Examining NAND Flash Alternatives for Mobiles: Part 1.” Oct 2003. [Online]. Available :
<http://www.commsdesign.com/story/OEG20031022S0011>.
- [15] TOSHIBA AMERICA ELECTRONIC COMPONENTS, INC. *NAND Flash Applications Design Guide*, April 2003. Revision 1.0.
- [16] TOSHIBA SEMICONDUCTOR COMPANY. *What is NAND Flash Memory*, March 2003.
- [17] WERTZ, J. R. and LARSON, W. J. (Eds), *Space Mission Analysis and Design*. Third edition.
- [18] ZWOLINSKI, M., *Digital System Design with VHDL*. Prentice Hall, 2000.

Appendix A

Reference Tables

The reference table shown in table A.1 shows maximum and other data rate capabilities of multiple Nand flash configurations. The required number of devices to maintain these data rates is also shown.

The reference table A.2 shows the power and area calculations for multiple Nand flash configurations and data rates.

Table A.1: Maximum Data Rate and Minimum Number of Devices

| I/O's 8/16 Bit | Page Reg 528/2112 Byte | Access: Serial or Parallel | Number of Buses | Max Data Rate (MB/sec) | Minimum Number of NAND devices for Data Rate (MB/sec) | | | | | | | | | | |
|----------------------|------------------------------|----------------------------------|-----------------------|------------------------------|--|----|----|----|----|----|----|----|---|---|---|
| | | | | | Max | 5 | 10 | 15 | 20 | 25 | 30 | 35 | | | |
| 8 | 528 | Ser | ≥ 2 | 15.9 | 18 | 6 | 11 | 16 | - | - | - | - | - | - | - |
| 16 | 528 | Ser | ≥ 2 | 31.8 | 34 | 6 | 11 | 16 | 21 | 26 | 31 | - | - | - | - |
| 8 | 2112 | Ser | ≥ 2 | 15.9 | 8 | 3 | 5 | 7 | - | - | - | - | - | - | - |
| 16 | 2112 | Ser | ≥ 2 | 31.8 | 14 | 3 | 5 | 7 | 8 | 10 | 12 | - | - | - | - |
| 8 | 528 | Para | 2 | 31.8 | 34 | 6 | 12 | 18 | 22 | 28 | 32 | - | - | - | - |
| 16 | 528 | Para | 2 | 63.6 | 66 | 6 | 12 | 18 | 22 | 28 | 32 | 38 | - | - | - |
| 8 | 2112 | Para | 2 | 31.8 | 14 | 4 | 6 | 8 | 10 | 12 | 14 | - | - | - | - |
| 16 | 2112 | Para | 2 | 63.6 | 26 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | - | - | - |
| 8 | 528 | Para | 3 | 47.7 | 51 | 9 | 15 | 18 | 24 | 30 | 33 | 39 | - | - | - |
| 16 | 528 | Para | 3 | 95.4 | 96 | 9 | 15 | 18 | 24 | 30 | 33 | 39 | - | - | - |
| 8 | 2112 | Para | 3 | 47.7 | 21 | 6 | 9 | 12 | 12 | 12 | 15 | 18 | - | - | - |
| 16 | 2112 | Para | 3 | 95.4 | 39 | 6 | 9 | 12 | 12 | 12 | 15 | 18 | - | - | - |
| 8 | 528 | Para | 4 | 63.6 | 68 | 12 | 16 | 20 | 24 | 32 | 36 | 40 | - | - | - |
| 16 | 528 | Para | 4 | 127.2 | 132 | 12 | 16 | 20 | 24 | 32 | 36 | 40 | - | - | - |
| 8 | 2112 | Para | 4 | 63.8 | 28 | 8 | 8 | 12 | 12 | 16 | 16 | 20 | - | - | - |
| 16 | 2112 | Para | 4 | 127.2 | 52 | 8 | 8 | 12 | 12 | 16 | 16 | 20 | - | - | - |

Table A.2: Power and Area Calculations

| I/O's 8/16 Bit | Page Reg 528/2112 Byte | Access: Serial or Parallel | Number of Buses | Max Rate | | | 5 MB/sec | | | 10 MB/sec | | | 15 MB/sec | | |
|----------------------|------------------------------|----------------------------------|-----------------------|------------|-------------------------|------|------------|-------------------------|------|------------|-------------------------|------|------------|-------------------------|------|
| | | | | Power W | Area cm ² | Pins | Power W | Area cm ² | Pins | Power W | Area cm ² | Pins | Power W | Area cm ² | Pins |
| 8 | 528 | Ser | ≥2 | 1.782 | 44.64 | 48 | 0.594 | 14.88 | 36 | 1.089 | 27.28 | 41 | 1.584 | 39.68 | 46 |
| 16 | 528 | Ser | ≥2 | 3.366 | 84.32 | 80 | 0.594 | 14.88 | 52 | 1.089 | 27.28 | 57 | 1.584 | 39.68 | 62 |
| 8 | 2112 | Ser | ≥2 | 0.792 | 19.84 | 38 | 0.297 | 7.44 | 33 | 0.495 | 12.4 | 35 | 0.693 | 17.36 | 37 |
| 16 | 2112 | Ser | ≥2 | 1.386 | 34.72 | 60 | 0.297 | 7.44 | 49 | 0.495 | 12.4 | 51 | 0.693 | 17.36 | 53 |
| 8 | 528 | Para | 2 | 3.366 | 84.32 | 64 | 0.954 | 14.88 | 36 | 1.188 | 29.76 | 42 | 1.782 | 44.64 | 48 |
| 16 | 528 | Para | 2 | 6.534 | 163.68 | 112 | 0.594 | 14.88 | 52 | 1.188 | 29.76 | 58 | 1.782 | 44.64 | 64 |
| 8 | 2112 | Para | 2 | 1.386 | 34.72 | 44 | 0.396 | 9.92 | 34 | 0.594 | 14.88 | 36 | 0.792 | 19.84 | 38 |
| 16 | 2112 | Para | 2 | 2.574 | 64.48 | 72 | 0.396 | 9.92 | 50 | 0.594 | 14.88 | 52 | 0.792 | 19.84 | 54 |
| 8 | 528 | Para | 3 | 5.049 | 126.48 | 96 | 0.891 | 22.32 | 54 | 1.485 | 37.2 | 60 | 1.782 | 44.64 | 63 |
| 16 | 528 | Para | 3 | 9.504 | 238.08 | 165 | 0.891 | 22.32 | 78 | 1.485 | 37.2 | 84 | 1.782 | 44.64 | 87 |
| 8 | 2112 | Para | 3 | 2.079 | 52.08 | 66 | 0.594 | 14.88 | 51 | 0.891 | 22.32 | 54 | 1.188 | 29.76 | 57 |
| 16 | 2112 | Para | 3 | 3.861 | 96.72 | 108 | 0.594 | 14.88 | 75 | 0.981 | 22.32 | 78 | 1.188 | 29.76 | 81 |
| 8 | 528 | Para | 4 | 6.732 | 168.64 | 128 | 1.188 | 29.76 | 72 | 1.584 | 39.68 | 76 | 1.98 | 49.6 | 80 |
| 16 | 528 | Para | 4 | 13.068 | 327.36 | 224 | 1.188 | 29.76 | 104 | 1.584 | 39.68 | 108 | 1.98 | 49.6 | 112 |
| 8 | 2112 | Para | 4 | 2.772 | 69.44 | 88 | 0.792 | 19.84 | 68 | 0.792 | 19.84 | 68 | 1.188 | 29.76 | 72 |
| 16 | 2112 | Para | 4 | 5.148 | 128.96 | 144 | 0.792 | 19.84 | 100 | 0.792 | 19.84 | 100 | 1.188 | 29.76 | 104 |

Appendix B

Demonstration Board Schematics

This appendix contains the PCB schematics for the Nand flash demonstration board.

- Page 1 - Top Level Schematic
- Page 2 - FPGA and Configuration
- Page 3 - Nand Flash Bus 0
- Page 4 - Nand Flash Bus 1
- Page 5 - Nand Flash Bus 2
- Page 6 - Nand Flash Bus 3
- Page 7 - Power Supply
- Page 8 - Layout Front
- Page 9 - Layout Back

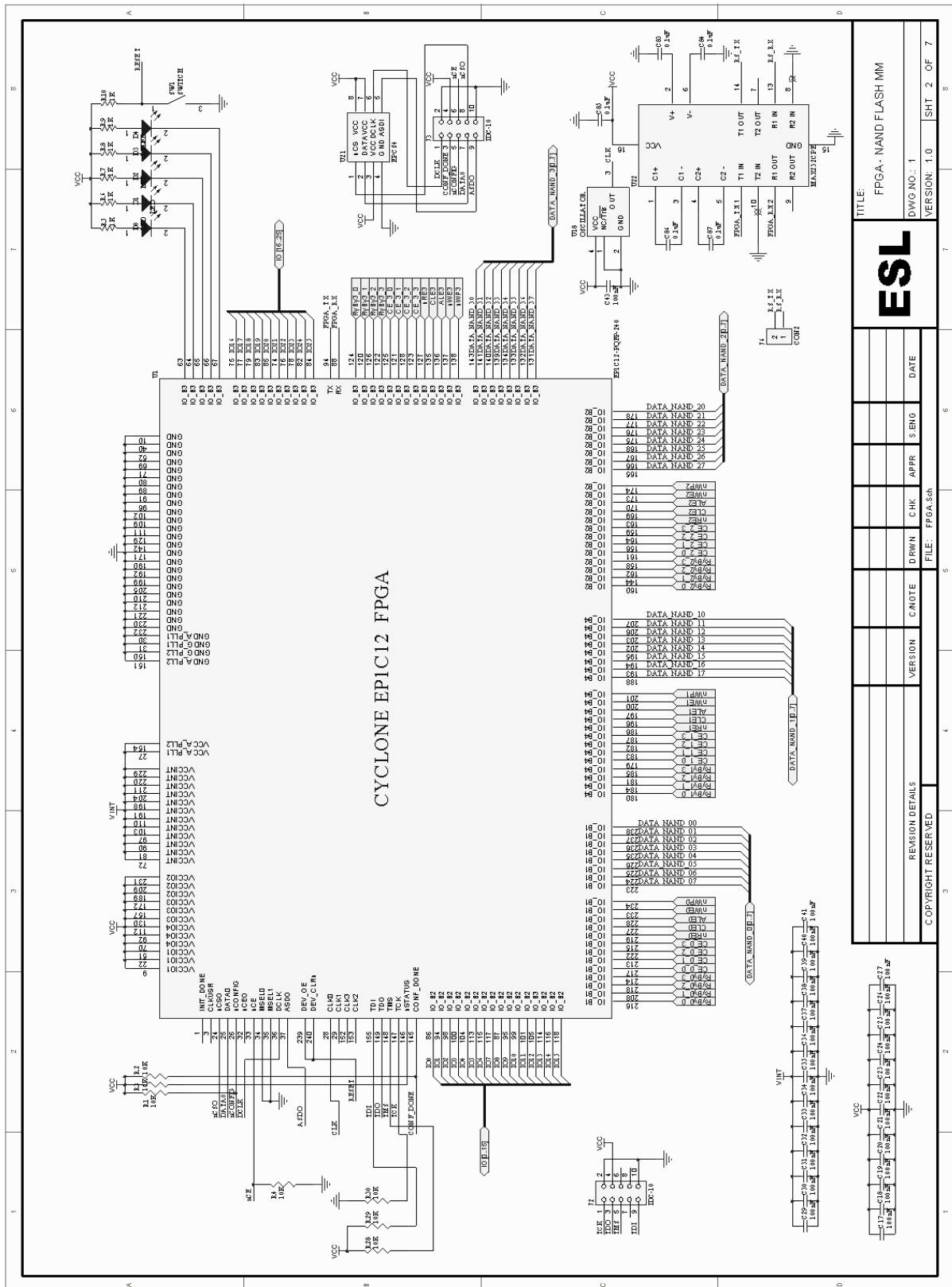


Figure B.2: Schematics Page 2

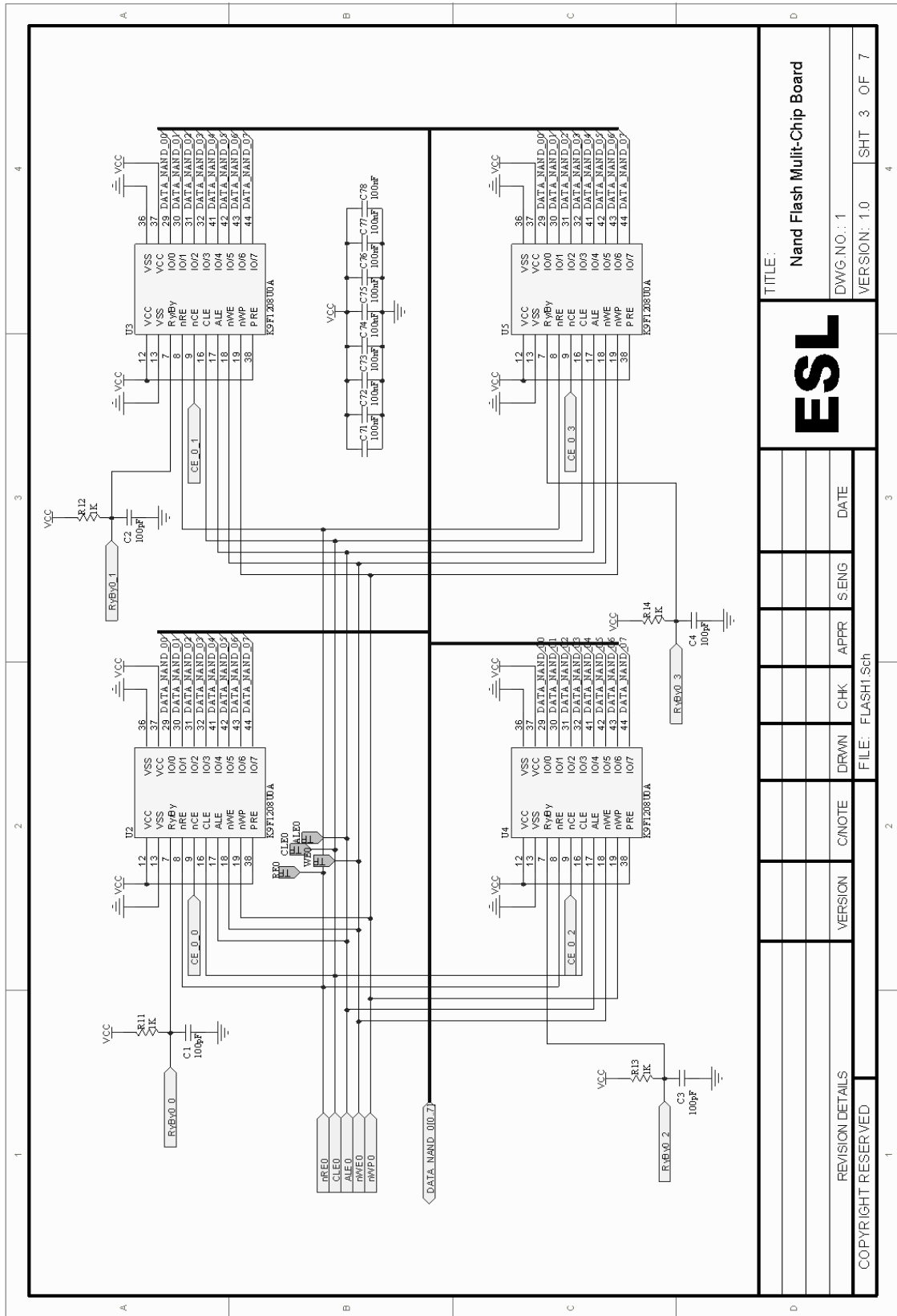


Figure B.3: Schematics Page 3

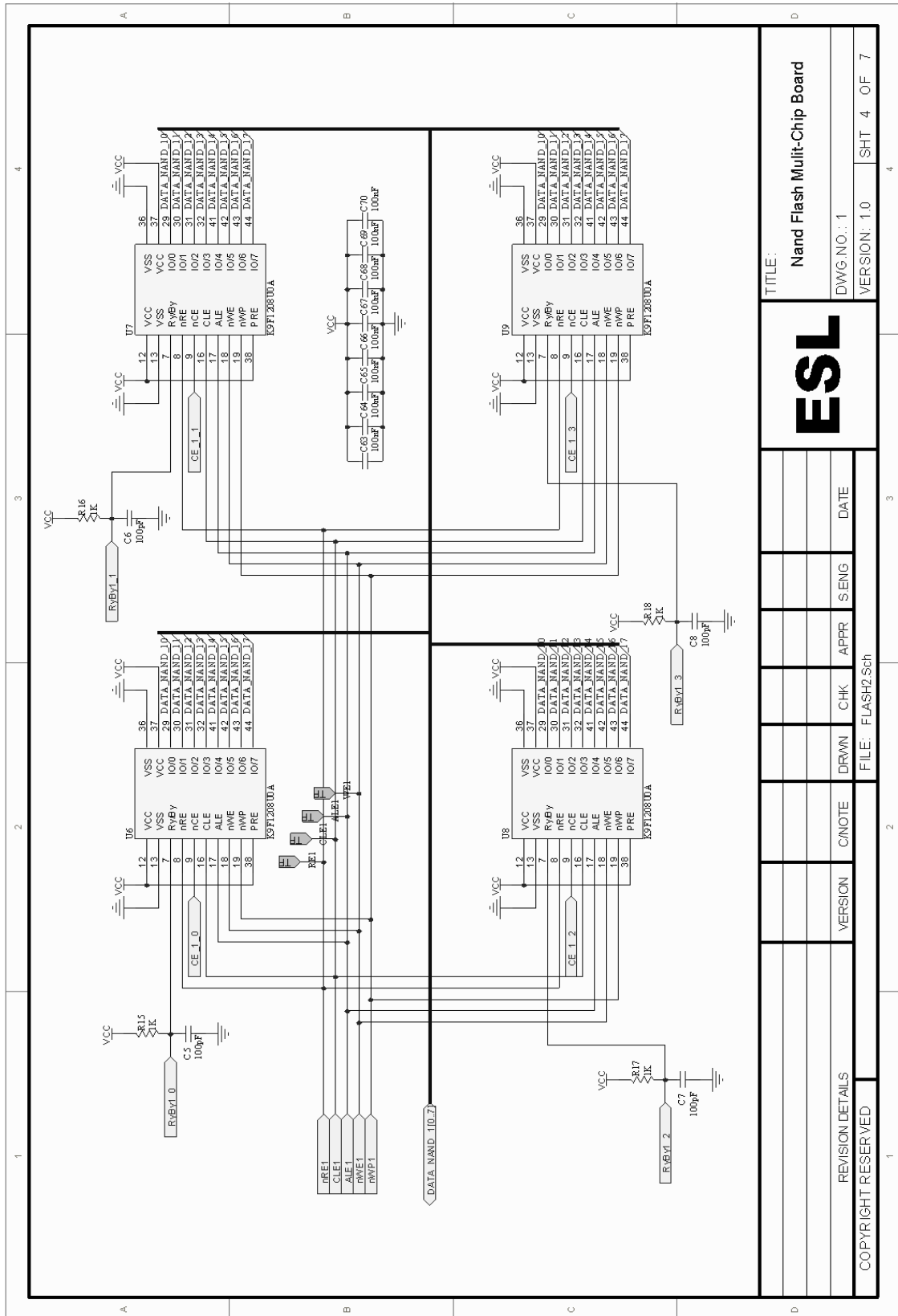


Figure B.4: Schematics Page 4

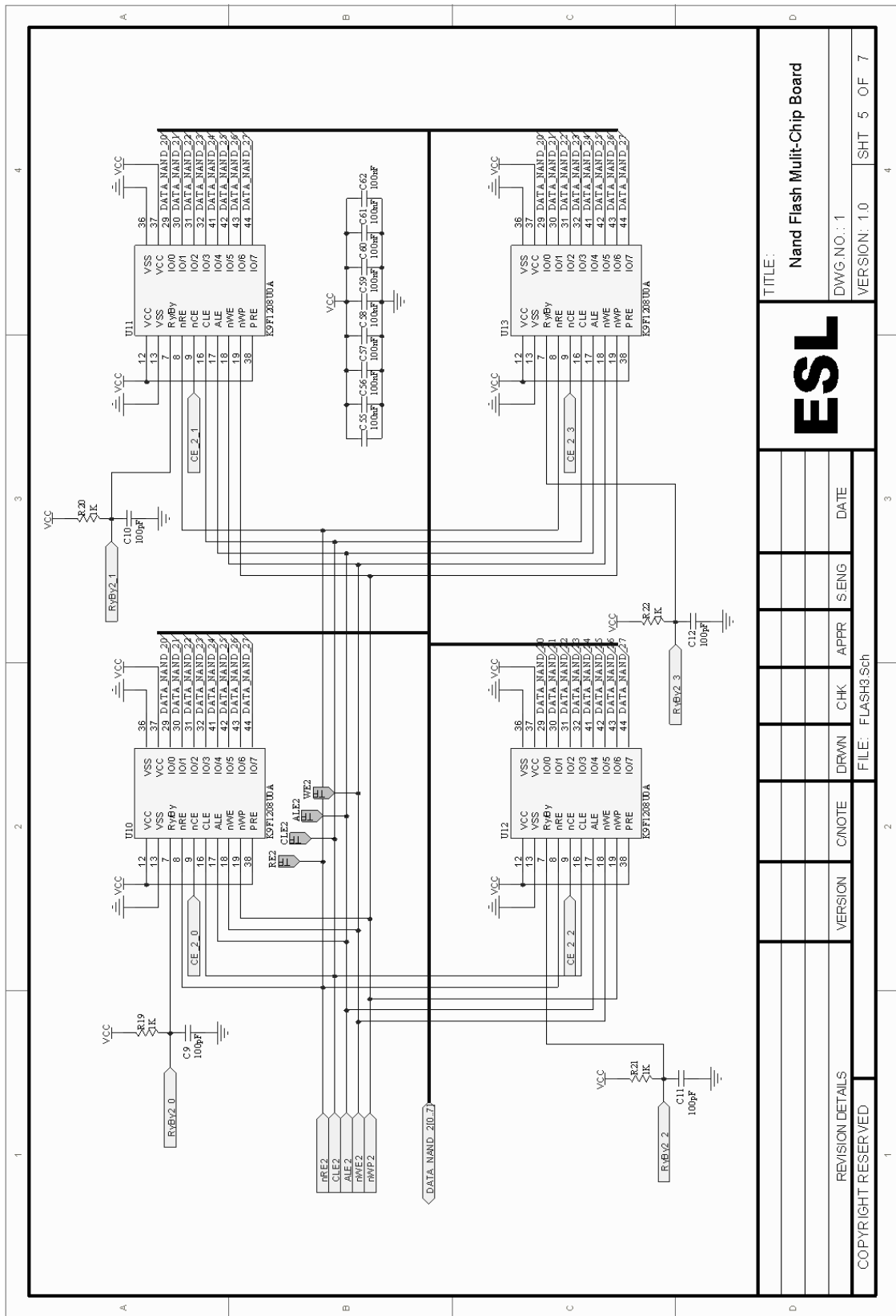


Figure B.5: Schematics Page 5

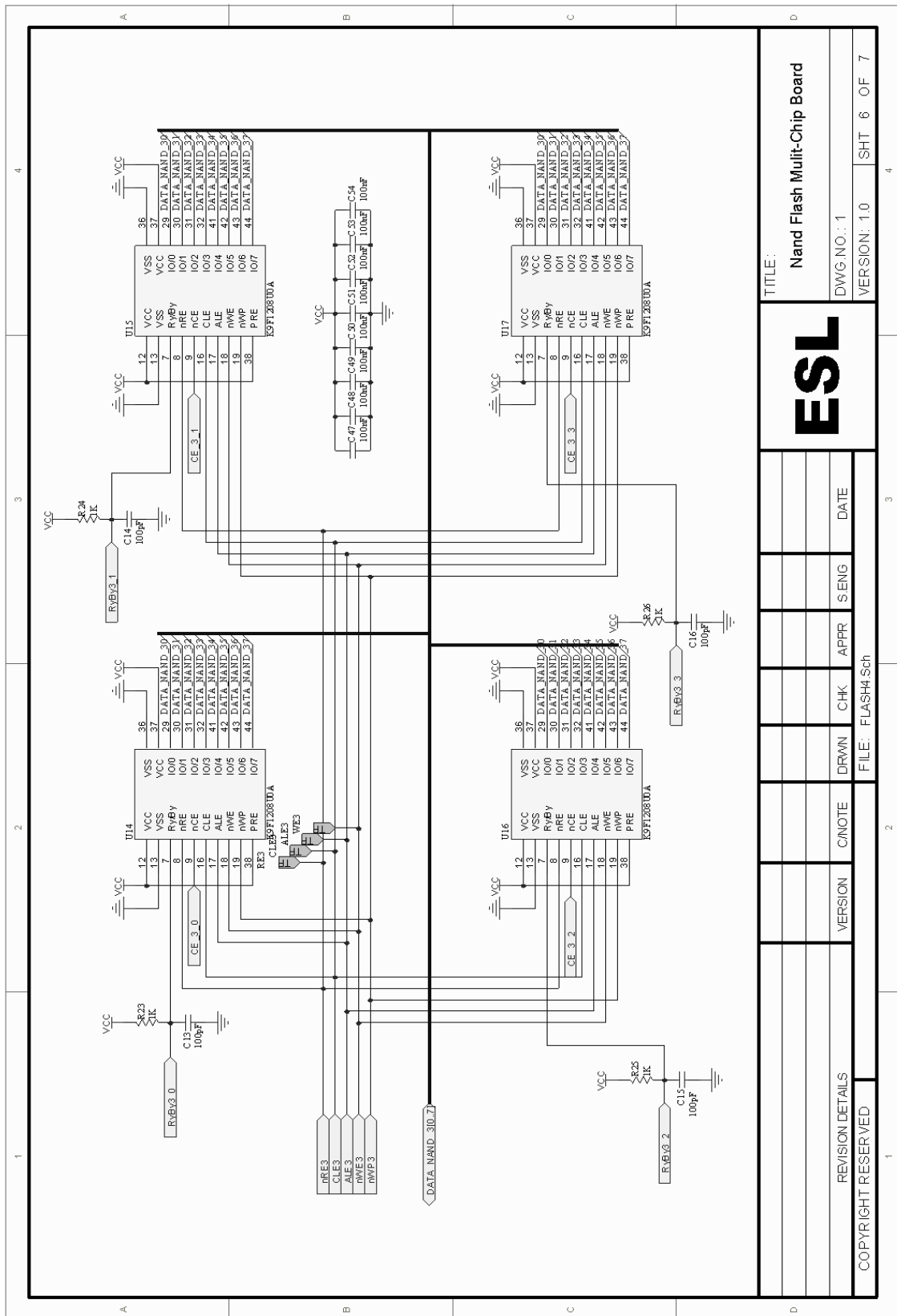
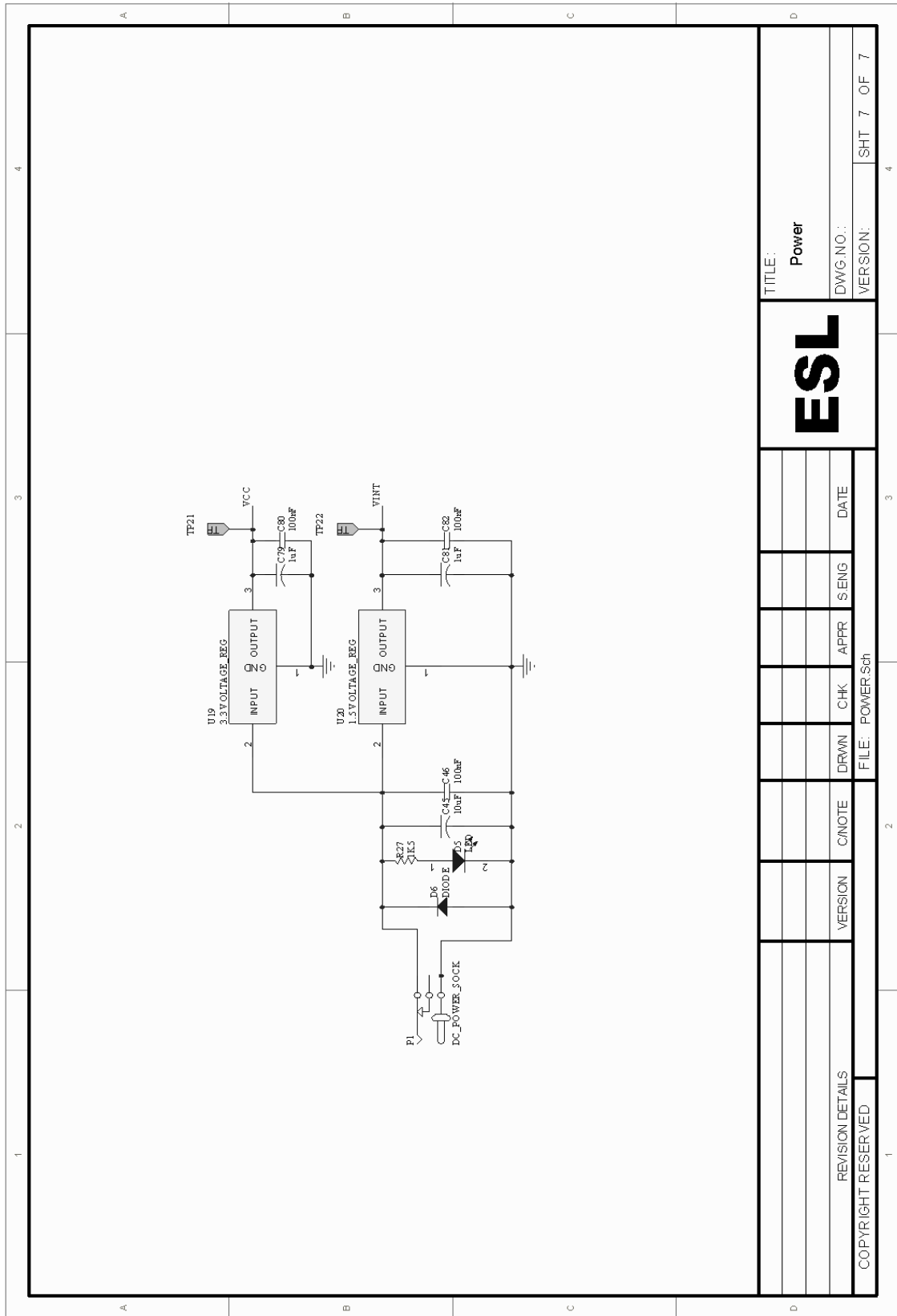


Figure B.6: Schematics Page 6



| | | | | | | | | | |
|---------------------|--|---------|--|-------|-------|-----|------|------|------|
| REVISION DETAILS | | VERSION | | CNOTE | DRWVN | CHK | APPR | SENG | DATE |
| COPYRIGHT RESERVED | | | | | | | | | |
| ESL | | | | | | | | | |
| TITLE: Power | | | | | | | | | |
| DWG.NO.: SHT 7 OF 7 | | | | | | | | | |
| VERSION: SHT 7 OF 7 | | | | | | | | | |

Figure B.7: Schematics Page 7

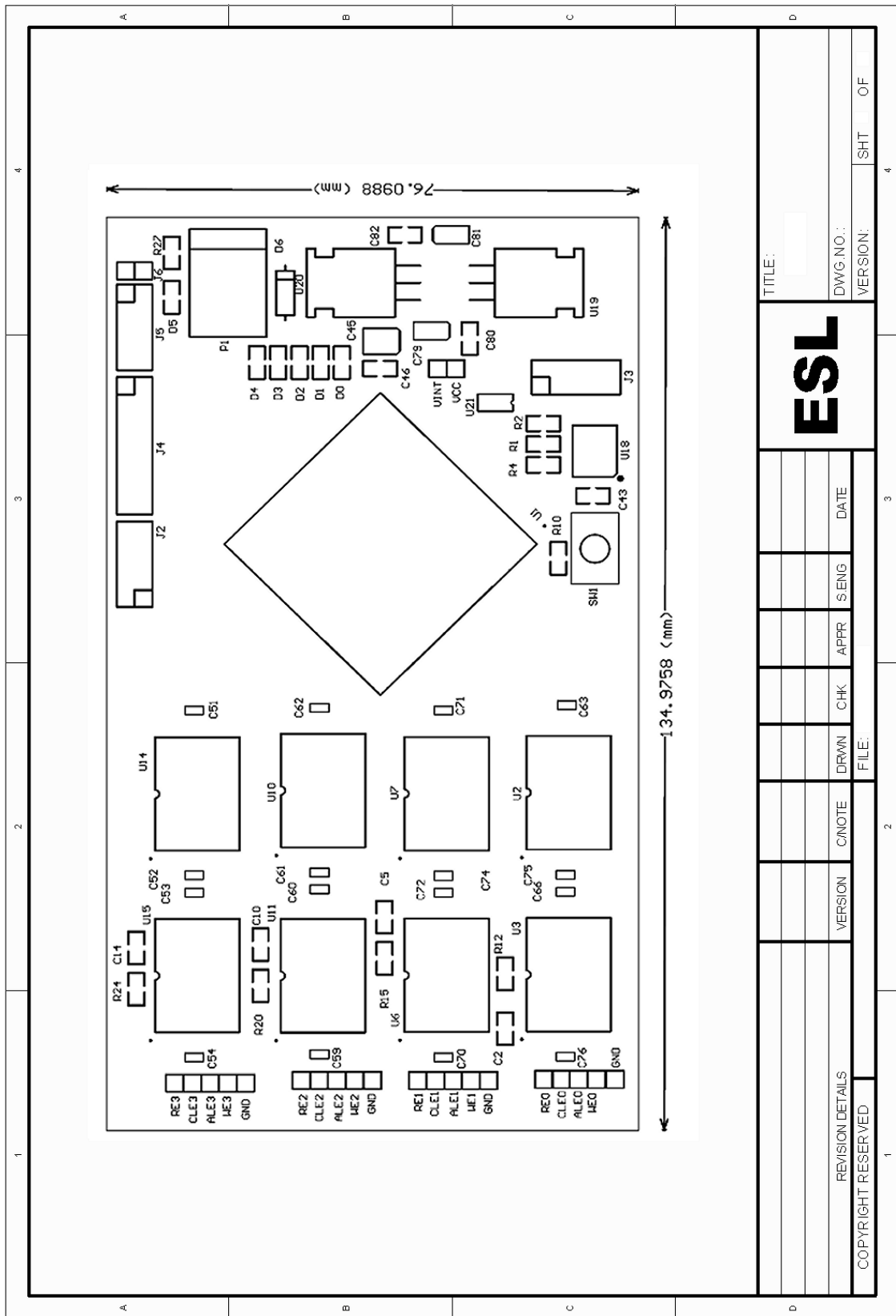


Figure B.8: Schematics Page 8

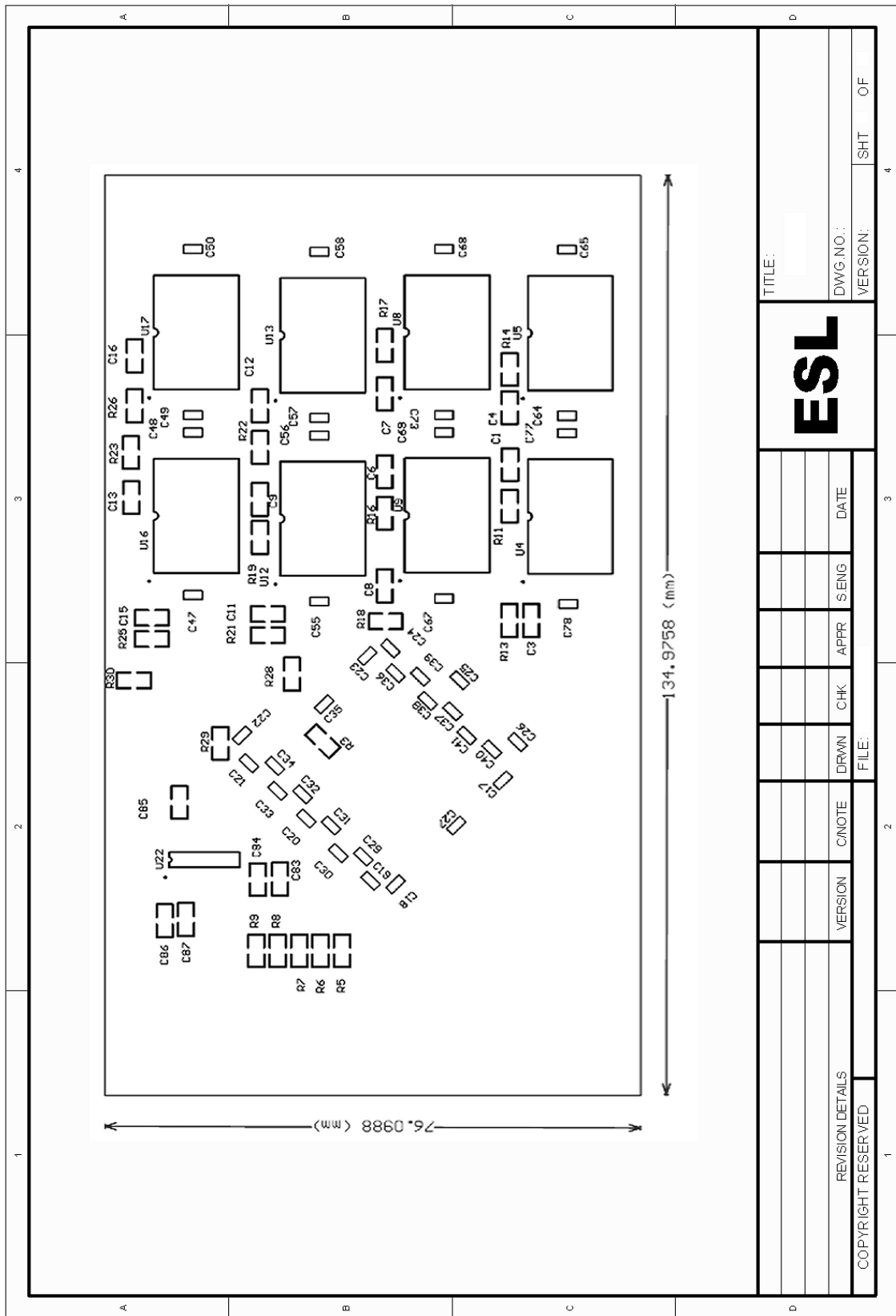


Figure B.9: Schematics Page 9

| | | | | | | | | |
|--------------------|--|---------|-------|-------|-----|------|------|------|
| REVISION DETAILS | | VERSION | CNOTE | DRWVN | CHK | APPR | SENG | DATE |
| COPYRIGHT RESERVED | | | | | | | | |
| ESL | | | | | | | | |
| TITLE: | | | | | | | | |
| DWG.NO.: | | | | | | | | |
| VERSION: | | | | | | | | |
| | | | | | | | SHT | OF |
| | | | | | | | 4 | 4 |

Appendix C

VHDL Code

The attached CD contains files containing the VHDL code written for this thesis. The CD is split up into three different directories:

- VHDL files for 64MB/512 byte page register/8-bit NAND flash system.
- VHDL files for 256MB/2112 byte page register/16-bit NAND flash system.
- VHDL files for 256MB/2112 byte page register/8-bit NAND flash system with read while write functionality

An example of the VHDL code used in the demonstration test board is shown next. This is the VHDL code written for the interface control unit.

```

--Interface Control Unit
--Ian Horsburgh M-Thesis 2004

LIBRARY ieee;
USE ieee.STD.LOGIC.ALL;
USE ieee.STD.LOGIC.unsigned.ALL;
USE ieee.numeric_std.ALL;

ENTITY inter_nand IS
PORT
  (clk:          IN STD_LOGIC;--50MHz Clk in
  clk25M:       IN STD_LOGIC;--25MHz CLK in
  reset:        IN STD_LOGIC;--Global Reset
  data_nand:    INOUT STD_LOGIC_VECTOR(7 DOWNTO 0);--Bidirectional data lines (FLASH)
  data_in:      IN STD_LOGIC_VECTOR(7 DOWNTO 0);--Data in (ECC GEN)
  data_out:     OUT STD_LOGIC_VECTOR(7 DOWNTO 0);--Data out (EDAC)
  strobe_in:    IN STD_LOGIC;--Data strobe in (ECC GEN)
  strobe_out:   OUT STD_LOGIC;--Data strobe out (EDAC)
  command:     IN STD_LOGIC_VECTOR(3 DOWNTO 0);--Command in (DATA ROUTER)
  ry_by:       IN STD_LOGIC_VECTOR(3 DOWNTO 0);--Ready/busy lines (FLASH)
  com:         IN STD_LOGIC;--Command strobe (DATA ROUTER)
  chip_sel:    IN STD_LOGIC_VECTOR(1 DOWNTO 0);--Device select (BB TABLE)
  address:     IN STD_LOGIC_VECTOR(16 DOWNTO 0);--Generated Address (BB TABLE)
  empty:       IN STD_LOGIC;--FIFO empty signal (EDAC)
  store:       OUT STD_LOGIC;--Flag-store operation (BB TABLE)
  load:        OUT STD_LOGIC_VECTOR(2 DOWNTO 0);--Flag-load operation (BB TABLE)
  bb_data:     OUT STD_LOGIC_VECTOR(13 DOWNTO 0);--New bad block address (BB TABLE)
  bb_strobe:   OUT STD_LOGIC;--Strobe new bad block address (BB TABLE)
  addr_get:    OUT STD_LOGIC_VECTOR(2 DOWNTO 0);--Flags to generate next address (bb_table)
  ale:         OUT STD_LOGIC;--Address latch enable (flash)
  cle:         OUT STD_LOGIC;--Command Latch enable (flash)

```



```

we:      OUT STD_LOGIC;-- Write enable (flash)
re:      OUT STD_LOGIC;-- Read enable (flash)
wp:      OUT STD_LOGIC;-- Write Protect (flash)
ce:      OUT STD_LOGIC_VECTOR(3 DOWNTO 0);-- Chip enable (flash)
ack:     OUT STD_LOGIC;-- Acknowledge line for read and write
strobe_sr: OUT STD_LOGIC;-- Acknowledge for status read
);

END ENTITY inter_nand;

ARCHITECTURE rtl OF inter_nand IS

-- State Machine Declarations
TYPE state_page IS
(reset_page, idle_page, bb_addr_set, chip_set, command_set, command_write,
command_set1, command_write1, addr_set, addr_write, send_ack, data_wait,
data_hold, data_write, command_set2, command_write2, busy_wait, busy_wait2);

TYPE state_command IS
(reset_command, command_wait, command_exe, wait_rd_ack, load_kick,
wait_store_kick, store_kick, wait_store_kick1, store_kick1, extra_hold,
com_ack, fin_wait, rd_st_go);

TYPE state_read IS
(reset_read, idle_read, bb_addr_set, chip_set, chip_set1, command_set,
command_write, addr_set, addr_write, addr_gets, busy_wait, busy_wait2,
wait_read_start, extra_wait, read_data, read_data2, data_hold, status_read, bb_send);

TYPE state_erase IS
(reset_erase, idle_erase, bb_set, chip_set, command_set, command_write,
addr_set, addr_write, command_set2, command_write2,
busy_wait, busy_wait2, bb_busy_wait, bb_busy_wait2);

TYPE state_reset IS
(reset1, idle_reset, command_set, command_write, busy_wait, busy_wait2);

--TYPE state_read1 IS
--(reset_read1, idle_read1, send_read1, strobe_low, status_read, bb_send);

-- signal Declarations
-- State Machine signals
SIGNAL st_page           : state_page;
SIGNAL st_command       : state_command;
SIGNAL st_read          : state_read;
SIGNAL st_erase         : state_erase;
SIGNAL st_reset         : state_reset;
--SIGNAL st_read1       : state_read1;

-- Command State Machine SIGNALs
SIGNAL command_reg      : STD_LOGIC_VECTOR(3 DOWNTO 0) := (OTHERS => '0'); --command register
SIGNAL sel              : STD_LOGIC_VECTOR(2 DOWNTO 0); --select lines for multiplexor
SIGNAL erase_kick       : STD_LOGIC_VECTOR(1 DOWNTO 0); --start erase state machine
SIGNAL reset_kick       : STD_LOGIC; --start reset state machine
SIGNAL read_kick        : STD_LOGIC_VECTOR(1 DOWNTO 0); --start read state machine
SIGNAL page_kick        : STD_LOGIC_VECTOR(1 DOWNTO 0); --start page state machine

--Page Program State Machine signals
SIGNAL ryby_temp2       : STD_LOGIC_VECTOR(3 DOWNTO 0); --Synchronise ready\busy SIGNAL from flash
SIGNAL addr_count       : STD_LOGIC_VECTOR(1 DOWNTO 0); --count address cycles
SIGNAL count_byte       : STD_LOGIC_VECTOR(9 DOWNTO 0); --count bytes written to flash
SIGNAL row1, row2       : STD_LOGIC_VECTOR(7 DOWNTO 0); --address cycle 1 and 2
SIGNAL row3             : STD_LOGIC; --address cycle 3
SIGNAL contr_page       : STD_LOGIC_VECTOR(2 DOWNTO 0); --flash control lines cle, ale, we
SIGNAL temp_page        : STD_LOGIC_VECTOR(1 DOWNTO 0); --normal or bb page write flag
SIGNAL data_page        : STD_LOGIC_VECTOR(7 DOWNTO 0); --data to flash (mux)
SIGNAL pg_fin           : STD_LOGIC; --page SM busy SIGNAL
SIGNAL cs_pg            : STD_LOGIC_VECTOR(3 DOWNTO 0); --chip enable lines (flash)
SIGNAL addr_get_pg      : STD_LOGIC; --next address generate flag
SIGNAL int_ack          : STD_LOGIC; --acknowledge that setup cycles finished
--device store
SIGNAL part1           : STD_LOGIC; --page 1 or 2 of bb table write

--Read State Machine signals
SIGNAL strobe_temp      : STD_LOGIC; --Strobe for read_byte SM to latch data from flash
SIGNAL ryby_temp        : STD_LOGIC; --Synchronise ready\busy SIGNAL from flash
SIGNAL addr_count1      : STD_LOGIC_VECTOR(1 DOWNTO 0); --count address cycles
SIGNAL count_byte1      : STD_LOGIC_VECTOR(9 DOWNTO 0); --count bytes written to flash
SIGNAL row01, row02     : STD_LOGIC_VECTOR(7 DOWNTO 0); --address cycle 1 and 2

```

```

SIGNAL row03          :STD_LOGIC;           --address cycle 3
SIGNAL contr_read     :STD_LOGIC_VECTOR(3 DOWNTO 0); --flash control lines cle,ale,we,re
SIGNAL temp_read      :STD_LOGIC_VECTOR(1 DOWNTO 0); --normal read,bb table read or status read
SIGNAL read_sel       :STD_LOGIC;           --control bi-dir buffer for flash data lines
SIGNAL data_read      :STD_LOGIC_VECTOR(7 DOWNTO 0); --data to flash
SIGNAL rd_fin         :STD_LOGIC;           --read SM busy SIGNAL
SIGNAL cs_rd          :STD_LOGIC_VECTOR(3 DOWNTO 0); --chip enable lines (flash)
SIGNAL addr_get_rdr   :STD_LOGIC;           --next address generate flag
SIGNAL rd_ack         :STD_LOGIC;           --acknowledge page ready to read
SIGNAL ch_templ       :INTEGER RANGE 0 TO 3; --device store
SIGNAL part           :STD_LOGIC;           --page 1 or 2 of bb table read
SIGNAL counter        :STD_LOGIC_VECTOR(12 DOWNTO 0); --debug -> counter to add delay
SIGNAL data_temp      :STD_LOGIC_VECTOR(7 DOWNTO 0); --data latch from flash
--Read_Byte State Machine signals

--Erase State Machine SIGNALS
SIGNAL ryby_templ     :STD_LOGIC;           --Synchronise ready\busy SIGNAL from flash
SIGNAL addr_count2    :STD_LOGIC_VECTOR(1 DOWNTO 0); --count address cycles
SIGNAL status_addr    :STD_LOGIC_VECTOR(13 DOWNTO 0); --block address for status read
SIGNAL row11,row12    :STD_LOGIC_VECTOR(7 DOWNTO 0); --address cycle 1 and 2
SIGNAL row13          :STD_LOGIC;           --address cycle 3
SIGNAL contr_erase    :STD_LOGIC_VECTOR(2 DOWNTO 0); --flash control lines cle,ale,we
SIGNAL erase_temp     :STD_LOGIC_VECTOR(1 DOWNTO 0); --normal erase,bb table erase
SIGNAL data_erase     :STD_LOGIC_VECTOR(7 DOWNTO 0); --data to flash
SIGNAL er_fin         :STD_LOGIC;           --erase SM busy SIGNAL
SIGNAL cs_er          :STD_LOGIC_VECTOR(3 DOWNTO 0); --chip enable lines (flash)
SIGNAL addr_get_er    :STD_LOGIC;           --next address generate flag
SIGNAL ch_temp        :INTEGER RANGE 0 TO 3; --device store
SIGNAL rdy_by         :STD_LOGIC_VECTOR(3 DOWNTO 0); --ready/busy for all busses
--Reset State Machine SIGNALS
SIGNAL contr_reset    :STD_LOGIC_VECTOR(1 DOWNTO 0); --flash control lines cle,we
SIGNAL rst_fin        :STD_LOGIC;           --reset SM busy SIGNAL
--Multiplexor SIGNALS
SIGNAL control_bus    :STD_LOGIC_VECTOR(4 DOWNTO 0); --flash control lines cle,ale,we,re,wp
SIGNAL temp_nand      :STD_LOGIC_VECTOR(7 DOWNTO 0); --flash data lines to bi-dir buffer

BEGIN

cle<=control_bus(4);      --set the control lines
ale<=control_bus(3);      --set the control lines
we<=control_bus(2);       --set the control lines
re<=control_bus(1);       --set the control lines
wp<=control_bus(0);       --set the control lines

addr_get<=addr_get_er & addr_get_rdr & addr_get_pg; --join together all address gets
ack<=int_ack OR rd_ack;   --OR the acknowledge signals from read and page SMs

-----
--Bidirectional Control for data_nand lines
high_impedance: PROCESS (read_sel,temp_nand)

BEGIN
CASE read_sel IS
--read_sel controls bi-dir buffer
WHEN '1' => --high impedance for a read
data_nand <= (OTHERS=>'Z');
WHEN '0' => --set to output for all other operations
data_nand<=temp_nand;
WHEN OTHERS =>
NULL;
END CASE;
END PROCESS;

-----
--Multiplexor for control lines and data lines
mux:PROCESS
(sel ,contr_page ,data_page ,contr_read ,data_read ,contr_erase ,data_erase ,contr_reset ,cs_pg ,cs_rd ,cs_er)

BEGIN
CASE sel IS
WHEN "001" => -- Page Program Select
control_bus<=contr_page & "11";
temp_nand<=data_page;
ce<=cs_pg;

WHEN "011" => -- Read Select

```

```

        control_bus<=contr_read & '1';
        temp_nand<=data_read;
        ce<=cs_rd;

    WHEN "010" =>                -- Erase Select
        temp_nand<=data_erase;
        control_bus<=contr_erase & "11";
        ce<=cs_er;

    WHEN "100" =>                -- Reset Select
        control_bus<=contr_reset(1) & '0' & contr_reset(0) & "11";
        temp_nand<="11111111";
        ce<="0000";

    WHEN OTHERS=>
        ce<="1111";
        control_bus<="00110";
        temp_nand<=(OTHERS='0');

    END CASE;

END PROCESS;

-----
--Command Register, controls all the other state machines
commands :PROCESS (clk , reset , com , pg_fin , rd_fin , er_fin , rst_fin )

BEGIN

    IF reset = '1' THEN
        st_command<=reset_command;
    ELSIF rising_edge(clk) THEN

        load<="000";
        store<='0';

        CASE st_command IS

        WHEN reset_command =>
            sel<="001";
            IF reset = '0' THEN
                st_command<=command_wait;
            ELSE
                st_command<=reset_command;
            END IF;

        WHEN command_wait =>
            page_kick<="00";
            read_kick<="00";
            erase_kick<="00";
            reset_kick<='0';
            command_reg<=command;
            IF com = '1' THEN
                st_command<=command_exe;
            END IF;

        WHEN command_exe =>

            CASE command_reg IS

                WHEN "1001"|"1010"|"1011"|"1100"|"1101" =>    --kick off load BB table state machine
                                                                --for selected section
                    read_kick<="11";
                    sel<="011";
                    st_command<=wait_rd_ack;

                WHEN "0101" =>    --kick off store BB table state machine
                    page_kick<="10";
                    sel<="001";
                    st_command<=wait_store_kick;

                WHEN "0010" =>    --kick off page program state machine
                    page_kick<="01";
                    sel<="001";
                    st_command<=extra_hold;

                WHEN "0011" =>    --kick off read program state machine
                    read_kick<="01";
                    sel<="011";
                    st_command<=extra_hold;
            END CASE;
        END IF;
    END IF;
END PROCESS;

```

```

    WHEN "0100" =>      --kick off erase state machine
        erase_kick<="01";
        sel<="010";
        st_command<=com_ack;

    WHEN "0110" =>      --kick off erase bb table state machine
        erase_kick<="10";
        sel<="010";
        st_command<=com_ack;

    WHEN "1111" =>      --kick off reset device program state machine
        reset_kick <='1';
        sel<="100";
        st_command<=extra_hold;

    WHEN OTHERS =>
        st_command<=command_wait;

END CASE;

    WHEN wait_rd_ack =>      --Wait until page of data is ready to be read
        IF rd_ack = '1' THEN
            st_command<=load_kick;
        END IF;

    WHEN load_kick =>      --Tell Bad block table a load is in progress
        load<=command_reg(2 DOWNTO 0);
        st_command<=command_wait;

    WHEN wait_store_kick => --Wait until page program has competed setup cycles
        IF int_ack = '1' THEN
            st_command<=store_kick;
        END IF;

    WHEN store_kick =>      --Store signal to bb table to store page one of bb table
        store <='1';
        IF int_ack = '0' THEN
            st_command<=wait_store_kick1;
        END IF;

    WHEN wait_store_kick1 => --Wait until page program has competed setup cycles
        IF int_ack = '1' THEN
            st_command<=store_kick1;
        END IF;

    WHEN store_kick1 =>      --Store signal to bb table to store page two of bb table
        store <='1';
        st_command<=command_wait;

    WHEN extra_hold =>      --Extra hold for clk25M state machines to latch signal
        st_command<=command_wait;

    WHEN com_ack =>      --Wait until erase has started
        IF er_fin = '0' THEN
            st_command<=fin_wait;
        END IF;

    WHEN fin_wait =>      --Wait until erase has finished
        erase_kick<="00";
        IF er_fin = '1' THEN
            st_command<=rd_st_go;
        END IF;

    WHEN rd_st_go =>      --Start a read status operation
        sel<="011";
        read_kick<="10";
        st_command<=extra_hold;

END CASE;

END IF;

END PROCESS;

```

— Process to write data to Nand Flash device—

```

write : PROCESS (clk25M, reset , page_kick , data_in , ry_by)

BEGIN

  IF reset = '1' THEN
    st_page<=reset_page;
  ELSIF rising_edge(clk25M) THEN      --State machine runs off 25Mhz clock

    ryby_temp2<=ry_by;  --synchronise asynchronous input rdy_by
    addr_get_pg <='0';
    int_ack <='0';

    CASE st_page IS

      WHEN reset_page =>                --Reset state
        contr_page<="001";             --contr_page<=cle , ale , we
        IF reset = '0' THEN
          st_page<=idle_page;
        ELSE
          st_page<=reset_page;
        END IF;

      WHEN idle_page =>                  --Idle state waiting on page_kick from command SM
        --ch_temp2<=0;
        part1 <='0';
        cs_pg<="1111";
        pg_fin <='1';
        count_byte <=(OTHERS=>'0');
        addr_count <=(OTHERS=>'0');
        contr_page<="001";             --Default control bus values
        data_page <= "00000000";
        row1<=address(7 DOWNTO 0);     --Store address to be programmed for normal page write
        row2<=address(15 DOWNTO 8);    --Store address to be programmed for normal page write
        row3<=address(16);            --Store address to be programmed for normal page write
        temp_page<=page_kick;
        IF (page_kick="01") THEN      --Normal page write
          st_page<=chip_set;
        ELSIF (page_kick="10") THEN  --Bad block table write
          st_page<=bb_addr_set;
        END IF;

      WHEN bb_addr_set =>                --Set address to page 0 block 0 for bb table write
        row1<=(OTHERS=>'0');
        row2<=(OTHERS=>'0');
        row3 <='0';
        cs_pg<="0000";
        st_page<=command_set;

      WHEN chip_sel =>                    --Pull chip enable line low of device to be written to (
        chip_sel)
        --ch_temp2<=conv_integer(chip_sel);
        contr_page<="001";
        st_page<=command_set;
      CASE chip_sel IS
        WHEN "00" =>
          cs_pg<="1110";
        WHEN "01" =>
          cs_pg<="1101";
        WHEN "10" =>
          cs_pg<="1011";
        WHEN "11" =>
          cs_pg<="0111";
        WHEN OTHERS =>
          NULL;
        END CASE;

      WHEN command_set =>                --Write command 00H first to set pointer to start of page
        (multiplane device only)
        pg_fin <='0';                  --Page prog SM is busy when pg_fin is low
        contr_page<="100";             --CLE high
        data_page<="00000000";
        st_page<=command_write;

      WHEN command_write =>              --Write command to flash
        contr_page<="101";             --CLE high ,ALE low ,WE high
        st_page<=command_set1;

      WHEN command_set1 =>               --Write command 80H to start a page program operation
        part1<=NOT part1;

```

```

    contr_page<="100" ;           --CLE high ,ALE low ,WE low
    data_page<="10000000" ;
    st_page<=command_writel ;

    WHEN command_writel =>       --Write command to flash
        contr_page<="101" ;     --CLE high ,ALE low ,WE high
        st_page<=addr_set ;

    WHEN addr_set =>             --Address cycle start
        addr_count<=addr_count+1; --Count the number of address cycles
        contr_page<="010" ;     --CLE low ,ALE high ,WE low
        st_page<=addr_write ;
        CASE addr_count IS      --Send out address for each cycle
            WHEN "00" =>        --Column address
                data_page <="00000000" ;
            WHEN "01" =>        --First 5 bits page address , next 3 bits start of block
                address
                data_page <=row1 ;
            WHEN "10" =>        --Block address continued
                data_page <=row2 ;
            WHEN "11" =>        --Last bit of block address
                data_page <="0000000" & row3 ;
            WHEN OTHERS =>
                NULL ;
        END CASE ;

    WHEN addr_write =>          --Write address
        contr_page<="011" ;     --CLE low ,ALE high ,WE high
        IF addr_count = 0 THEN  --Continue if all address cycles have finished
            st_page<=send_ack ;
        ELSE                    --Else do next address cycle
            st_page<=addr_set ;
        END IF ;

    WHEN send_ack =>           --Send acknowledgment that setup cycles have finished
        contr_page<="001" ;     --CLE low ,ALE low ,WE high
        int_ack <='1' ;        --Acknowledgment signal
        st_page<=data_wait ;
        IF temp_page = "01" THEN --If normal page write then flag address generation to
            generate next address
            addr_get_pg <='1' ;
        END IF ;

    WHEN data_wait =>          --Wait until data arrives
        contr_page<="001" ;     --CLE low ,ALE low ,WE high
        IF strobe_in = '1' THEN --Strobe_in signals the arrival of a new byte of data
            st_page<=data_write ;
        END IF ;

    WHEN data_write =>         --Write data to flash
        data_page<=data_in ;
        count_byte<=count_byte+1; --Count the number of bytes written
        contr_page<="000" ;     --CLE low ,ALE low ,WE low
        IF (count_byte=517) THEN --If a page is written send next command
            st_page<=data_hold ;
        ELSE                    --Else wait for next byte
            st_page<=data_wait ;
        END IF ;

    WHEN data_hold =>          --Hold WE high to latch data into flash
        contr_page<="001" ;     --CLE low ,ALE low ,WE high
        st_page<=command_set2 ;

    WHEN command_set2 =>       --Send 10H command to finish page program operation
        contr_page<="100" ;     --CLE high ,ALE low ,WE low
        data_page <="00010000" ; --Set command to 10H
        st_page<=command_write2 ;

    WHEN command_write2 =>     --Write command
        contr_page<="101" ;     --CLE high ,ALE low ,WE high
        IF temp_page = "10" THEN --If bb table write then wait for ry-by signal
            st_page<=busy_wait ;
        ELSE                    --Else go back to idle state (normal write)
            st_page<=idle_page ;
        END IF ;

    WHEN busy_wait =>          --Wait for ry_by to go low
        contr_page<="001" ;     --CLE low ,ALE low ,WE high

```

```

        IF (ryby_temp2/= "1111") THEN
            st_page<=busy_wait2;
        END IF;

        WHEN busy_wait2 =>
            count_byte <=(OTHERS=>'0');
            row1<="00000001";
            contr_page<="001";
            IF ryby_temp2="1111" AND part1 = '1' THEN
                st_page<=command_set1;
            ELSIF part1 = '0' THEN
                st_page<=idle_page;
            END IF;

        END CASE;
    END IF;
END PROCESS write;

-----
-- Process to read data from Nand Flash device--

read :PROCESS (clk25M, reset, read_kick, empty, ry_by)

BEGIN

    IF reset = '1' THEN
        st_read<=reset_read;
    ELSIF rising_edge(clk25M) THEN
        ryby_temp<=ry_by(ch_temp1);
        strobe_temp <='0';
        strobe_out <='0';
        addr_get_rd <='0';
        rd_ack <='0';
        bb_store <='0';
        strobe_sr <='0';

        CASE st_read IS

            WHEN reset_read =>
                cs_rd<="1111";
                read_sel <='0';
                contr_read<="0011";
                IF reset = '0' THEN
                    st_read<=idle_read;
                ELSE
                    st_read<=reset_read;
                END IF;

            WHEN idle_read =>
                ch_temp1<=0;
                part <='0';
                counter <=(0=>'1',OTHERS=>'0');
                rd_fin <='1';
                read_sel <='0';
                count_byte1 <=(OTHERS=>'0');
                addr_count1 <=(OTHERS=>'0');
                contr_read<="0011";
                data_read <= "00000000";
                data_out <= "00000000";
                temp_read<=read_kick;
                row01<=address(7 DOWNTO 0);
                row02<=address(15 DOWNTO 8);
                row03<=address(16);
                CASE read_kick IS
                    WHEN "01" =>
                        st_read<=chip_set;
                    WHEN "10" =>
                        st_read<=chip_set1;
                    WHEN "11" =>
                        st_read<=bb_addr_set;
                    WHEN OTHERS =>
                        NULL;
                END CASE;

            WHEN bb_addr_set =>
                row01<=(OTHERS=>'0');
                row02<=(OTHERS=>'0');
                row03 <='0';

```

```

st_read<=chip_set;

WHEN chip_set =>
    ch_temp1<=conv_INTEGER(chip_sel);
    read_sel <='0';
    bb_read
    contr_read<="0011";
    st_read<=command.set;
CASE chip_sel IS
    WHEN "00" =>
        cs_rd<="1110";
    WHEN "01" =>
        cs_rd<="1101";
    WHEN "10" =>
        cs_rd<="1011";
    WHEN "11" =>
        cs_rd<="0111";
    WHEN OTHERS =>
        NULL;
END CASE;

WHEN chip_sel1 =>
    contr_read<="0011";
    st_read<=command.set;
CASE status_addr(13 DOWNTO 12) IS
    WHEN "00" =>
        cs_rd<="1110";
    WHEN "01" =>
        cs_rd<="1101";
    WHEN "10" =>
        cs_rd<="1011";
    WHEN "11" =>
        cs_rd<="0111";
    WHEN OTHERS =>
        NULL;
END CASE;

WHEN command.set =>
    rd_fin <='0';
    contr_read<="1001";
    st_read<=command.write;
    IF temp_read="10" THEN
        data_read <="01110000";
    ELSE
        data_read <="00000000";
    END IF;

WHEN command.write =>
    contr_read<="1011";
    IF temp_read="10" THEN
        st_read<=busy_wait;
    ELSE
        st_read<=addr.set;
    END IF;

WHEN addr.set =>
    addr_count1<=addr_count1+1;
    contr_read<="0101";
    st_read<=addr.write;
CASE addr_count1 IS
    WHEN "00" =>
        data_read <="00000000";
    WHEN "01" =>
        data_read <=row01;
    WHEN "10" =>
        data_read <=row02;
    WHEN "11" =>
        data_read <="00000000" & row03;
    WHEN OTHERS =>
        NULL;
END CASE;

WHEN addr.write =>
    contr_read<="0111";
    IF addr_count1 = 0 THEN
        st_read<=addr.gets;

```



```

ELSE --Else continue with address cycles
    st_read<=addr_set;
END IF;

WHEN addr_gets => --If a normal page read flag address generator to generate next
    address
    contr_read<="0011";
    st_read<=busy_wait;
    IF temp_read = "01" THEN
        addr_get_rd <='1';
    END IF;

WHEN busy_wait => --Wait for ry_by to go low
    contr_read<="0011"; --CLE low,ALE low,WE high,RE high
    IF (ryby_temp = '0' OR temp_read = "10") THEN
        st_read<=busy_wait2;
    END IF;

WHEN busy_wait2 => --Wait for rd_by to go high
    contr_read<="0011"; --CLE low,ALE low,WE high,RE high
    IF temp_read = "10" OR ryby_temp='1' THEN
        st_read<=wait_read_start;
    END IF;

WHEN wait_read_start => --Set signals and wait before starting to read
    contr_read<="0011"; --CLE low,ALE low,WE high,RE high
    rd_ack <='1'; --Acknowledge that read SM is ready to read a page of data
    read_sel <='1'; --Set bidirectional lines to high impedance
    IF (temp_read = "11" OR temp_read = "01") AND empty = '1' THEN --Ensure that EDAC FIFO
        is empty
        st_read<=read_data; --and that it is a normal or bb table read
    ELSIF temp_read = "10" THEN --Else for a status read carry on
        st_read<=extra_wait;
    END IF;

WHEN extra_wait => --Extra timing delay for status read
    st_read<=read_data;

WHEN read_data => --Read a byte of data from flash
    counter <=(0=>'1',OTHERS=>'0'); --debug counter set to 1
    count_byte1<=count_byte1+1; --Count the number of bytes read
    contr_read<="0010"; --CLE low,ALE low,WE high,RE low
    st_read<=read_data2;

WHEN read_data2 => --Read latch
    contr_read<="0011"; --CLE low,ALE low,WE high,RE high
    strobe_temp <='1'; --Let read_byte state machine know a byte is available on data
        lines
    IF (count_byte1=518) THEN --If whole page is read
        st_read<=data_hold; --Go out of read data loop
        data_temp<=data_nand;
        data_out<=data_nand;
        strobe_out <='1';
    ELSIF temp_read="01" OR temp_read = "11" THEN --Else if normal read and page not fully
        read
        data_out<=data_nand;
        strobe_out <='1';
        st_read<=read_data; --Keep in read data loop
    ELSIF temp_read="10" THEN --Else if status read
        st_read<=status_read; --Carry on to check status byte
    END IF;

WHEN data_hold => --If a bb table read then read second page from flash
    count_byte1 <=(OTHERS=>'0'); --set byte counter to 0
    part<=NOT part; --Flag for which page of bb table has been read
    contr_read<="0011"; --CLE low,ALE low,WE high,RE high
    row01<="00000001"; --Set address to page 1 of block 0 for second page of bb table
    IF (temp_read = "11" AND part = '0') THEN --If bb table read and only first page has
        been read
        st_read<=chip_set; --Read second page
    ELSE --Else goto idle read state
        st_read<=idle_read;
    END IF;

WHEN status_read => --Check status read for bad block
    strobe_sr <='1'; --Acknowledge that status read is complete
    IF data_temp(0) = '1' THEN --If a bad block goto bb_send state
        st_read<=bb_send;
    ELSE

```

```

        st_read<=idle_read; --If to a bad block return to idle state
    END IF;

    WHEN bb_send =>          --send the address including device to bb table to mark as bad
        block
        bb_data<=status_addr(13 DOWNTO 0); --bad block address
        bb_store<='1'; --Send bb block address
        st_read<=idle_read; --Goto idle state

    END CASE;
END IF;
END PROCESS read;

-----
-- Process to erase a block of memory

erase: PROCESS(clk25M, reset, erase_kick, ry_by)

BEGIN

    IF reset = '1' THEN
        st_erase<=reset_erase;
    ELSIF rising_edge(clk25M) THEN --Runs off the 25MHz clock

        rdy_by<=ry_by;
        ryby_temp1<=ry_by(ch_temp); --Synchronise the asynchronous ready_busy line
        addr_get_er <='0';

    CASE st_erase IS

        WHEN reset_erase =>          --Reset state
            status_addr<=(OTHERS=>'0');
            contr_erase<="001"; --CLE low, ALE low, WE high
            IF reset = '0' THEN
                st_erase<=idle_erase;
            ELSE
                st_erase<=reset_erase;
            END IF;

        WHEN idle_erase =>          --Idle state waiting for erase_kick from command SM
            row11<=address(7 DOWNTO 0); --Set address for normal block erase
            row12<=address(15 DOWNTO 8); --Set address for normal block erase
            row13<=address(16); --Set address for normal block erase
            ch_temp<=0;
            cs_er<="1111";
            er_fin <='1';
            addr_count2<=(OTHERS=>'0');
            contr_erase<="001"; --CLE low, ALE low, WE high
            data_erase <= "00000000";
            erase_temp<=erase_kick; --Store the TYPE of erase
            IF (erase_kick="01") THEN --signal to kick off normal erase
                st_erase<=chip_set;
            ELSIF (erase_kick="10") THEN --signal to kick off bad block erase
                st_erase<=bb_set;
            END IF;

        WHEN bb_set =>          --Set the address for a bad block erase
            row11<=(OTHERS=>'0'); --Block 0
            row12<=(OTHERS=>'0');
            row13<='0';
            cs_er<="0000"; --Select all devices on bus
            st_erase<=command_set;

        WHEN chip_set =>          --Set chip enable of device to be erased
            ch_temp<=conv_INTEGER(chip_sel);
            status_addr<=chip_sel & address(16 DOWNTO 5); --set the bad block address for a status
            read
            st_erase<=command_set;
        CASE chip_sel IS
            WHEN "00" =>
                cs_er<="1110";
            WHEN "01" =>
                cs_er<="1101";
            WHEN "10" =>
                cs_er<="1011";
            WHEN "11" =>
                cs_er<="0111";
        END CASE;
    END CASE;
END PROCESS;

```

```

        WHEN OTHERS =>
            NULL;
        END CASE;

    WHEN command_set =>
        --Set command to 60H for an erase operation
        er_fin <='0';
        contr_erase <="100";
        data_erase <="01100000";
        st_erase <=command_write;
        --CLE high ,ALE low ,WE low
        --60H

    WHEN command_write =>
        --Write command
        contr_erase <="101";
        st_erase <=addr_set;
        --CLE high ,ALE low ,WE high

    WHEN addr_set =>
        --Setup block address to flash
        addr_count2 <=addr_count2+1;
        contr_erase <="010";
        st_erase <=addr_write;
        --count address cycles
        --CLE low ,ALE high ,WE low
        CASE addr_count2 IS
            WHEN "00" =>
                data_erase <=row11;
            WHEN "01" =>
                data_erase <=row12;
            WHEN "10" =>
                data_erase <="0000000" & row13;
            WHEN OTHERS =>
                NULL;
            --block address start (last 3 bits)
            --block address continued
            --block address finish
        END CASE;

    WHEN addr_write =>
        --Write address to flash
        contr_erase <="011";
        IF addr_count2 = 3 THEN
            st_erase <=command_set2;
        ELSE
            st_erase <=addr_set;
        END IF;
        --CLE low ,ALE high ,WE high
        --If all address cycles are finish goto next command set
        --Else continue with address cycles

    WHEN command_set2 =>
        --Setup next command D0H to complete block erase
        contr_erase <="100";
        data_erase <="11010000";
        st_erase <=command_write2;
        IF erase_temp = "01" THEN
            addr_get_er <='1';
        END IF;
        --CLE high ,ALE low ,WE low
        --D0H
        --If a normal block erase flag address generator to

    WHEN command_write2 =>
        --Write command to flash
        contr_erase <="101";
        IF erase_temp = "01" THEN
            st_erase <=busy_wait;
        ELSE
            st_erase <=bb_busy_wait;
        END IF;
        --CLE low ,ALE high ,WE high

    WHEN busy_wait =>
        --Wait for ry-by to go low
        contr_erase <="001";
        IF ryby_temp1 = '0' THEN
            st_erase <=busy_wait2;
        END IF;

    WHEN busy_wait2 =>
        --Wait for ry-by to go high
        contr_erase <="001";
        IF ryby_temp1 = '1' THEN
            st_erase <=idle_erase;
        END IF;
        --Return to idle state

    WHEN bb_busy_wait =>
        --Wait for ry-by to go low
        contr_erase <="001";
        IF rdy_by/= "0000" THEN
            st_erase <=bb_busy_wait2;
        END IF;

    WHEN bb_busy_wait2 =>
        --Wait for ry-by to go high
        contr_erase <="001";
        IF rdy_by="1111" THEN
            st_erase <=idle_erase;
        END IF;
        --Return to idle state

```

```

        END CASE;
    END IF;
END PROCESS erase;

-----
--Process to reset device

device_reset : PROCESS(clk25M, reset, reset_kick)

BEGIN

    IF reset = '1' THEN
        st_reset <= reset1;
    ELSIF rising_edge(clk25M) THEN --State machine runs off 25MHz clock

        CASE st_reset IS

            WHEN reset1 => --Reset state
                contr_reset <= "01"; --CLE low, WE high
                IF reset = '0' THEN
                    st_reset <= idle_reset;
                ELSE
                    st_reset <= reset1;
                END IF;

            WHEN idle_reset => --Idle state wait for reset_kick
                rst_fin <= '1';
                contr_reset <= "01"; --CLE low, WE high
                IF (reset_kick = '1') THEN --signal to kick off erase program state machine
                    st_reset <= command_set;
                ELSE
                    st_reset <= idle_reset;
                END IF;

            WHEN command_set => --Set command FF (set in multiplexor)
                rst_fin <= '0';
                contr_reset <= "10"; --CLE high, WE low
                st_reset <= command_write;

            WHEN command_write => --Write command to flash
                contr_reset <= "11"; --CLE high, WE high
                st_reset <= busy_wait;

            WHEN busy_wait => --Wait for ry_by to go low
                contr_reset <= "01"; --CLE low, WE high
                IF ryby_temp = '0' THEN
                    st_reset <= busy_wait2;
                END IF;

            WHEN busy_wait2 => --Wait for ry_by to go high
                IF ryby_temp = '1' THEN
                    st_reset <= idle_reset; --Return to idle state
                END IF;

        END CASE;
    END IF;
END PROCESS device_reset;

END ARCHITECTURE;

```