A Mass Memory System for Satellites using Field Programmable Gate Arrays and Synchronous DRAM

Jacobus Jurie Vosloo

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

Study leader: Dr MM Blanckenberg
December 2006
Declaration

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

28 November 2006
Date
Summary

Technological advances have increased storage requirements on board satellites tremendously in recent years. Storage normally used on satellites is expensive and often complex and rigid hardware systems are needed to access the large number of interconnected devices effectively.

This thesis looks at the lower cost higher volume alternative of Synchronous DRAM, coupled with the flexibility of reprogrammable logic. A VHDL design of the system is done using a narrow data bus and no address bus and includes SDRAM control, reducing external components. Suggestions are also made to reduce the inherent risk associated with the technologies used to implement the design.
Opsomming

Onlangse tegnologiese vooruitgang het stoorvereistes op satelliete drasties verhoog. Geheue wat gewoonlik in satelliete gebruik word is duur en komplekse en rigiede hardeware stelsels word dikwels benodig om effektiewe toegang tot die groot aantal eenhede te verkry.

Hierdie tesis ondersoek die laer koste, hoër volume alternatief van Sinchrone DRAM, saam met die buigsaamheid van herprogrammeerbare logika. ’n VHDL-ontwerp van die stelsel is gedoen wat ’n smal databus, maar geen adresbus gebruik nie en SDRAM beheer insluit, wat dan eksterne komponente verminder. Aanbevelings word ook gemaak oor hoe om die inherente risiko’s wat met die implementeringstegnologieë gepaardgaan te verminder.
Acknowledgements

I would like to express my sincere thanks to the following people:

- My supervisor, Dr Mike Blanckenberg, for his guidance, encouragement and patience
- My colleagues for all the selfless assistance given
- My friends and family for their unwavering love and support
Contents

1 Introduction
   1.1 Background .................................................. 1
   1.2 Document Overview ........................................... 2

I Technology Overview .............................................. 5

2 Synchronous DRAM Technology ................................. 6
   2.1 Background .................................................. 6
   2.2 Internal Hardware Specifics .................................. 7
   2.3 SDRAM operation ............................................ 7
      2.3.1 State Operation ......................................... 8
      2.3.2 Addressing ............................................. 12
      2.3.3 Bank Access and Precharge .............................. 14
      2.3.4 Burst Access ........................................... 14
   2.4 Power ....................................................... 15

3 Field Programmable Gate Array Technology ...................... 16
   3.1 Background .................................................. 16
   3.2 Technology .................................................. 16
      3.2.1 Configurable Functional Units ......................... 17
      3.2.2 True RAM ............................................... 18
      3.2.3 Choosing a Suitable FPGA .............................. 18
   3.3 The Xilinx Virtex FPGA ...................................... 20

II VHDL Design .......................................................... 21

4 System Specification ............................................... 22
   4.1 Physical interconnects ....................................... 22
      4.1.1 Data and control bus .................................. 22
      4.1.2 Telecommand bus ...................................... 23
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.2.1</td>
<td>Port structure</td>
<td>55</td>
</tr>
<tr>
<td>9.2.2</td>
<td>Design</td>
<td>56</td>
</tr>
<tr>
<td>9.2.3</td>
<td>Testing the design</td>
<td>57</td>
</tr>
<tr>
<td>10</td>
<td>Error Detection and Correction Unit</td>
<td>61</td>
</tr>
<tr>
<td>10.1</td>
<td>Background</td>
<td>61</td>
</tr>
<tr>
<td>10.2</td>
<td>Implementation</td>
<td>62</td>
</tr>
<tr>
<td>10.3</td>
<td>VHDL Implementation</td>
<td>63</td>
</tr>
<tr>
<td>10.3.1</td>
<td>Port Structure</td>
<td>63</td>
</tr>
<tr>
<td>10.4</td>
<td>Discussion</td>
<td>64</td>
</tr>
<tr>
<td>10.5</td>
<td>Timing</td>
<td>65</td>
</tr>
<tr>
<td>10.5.1</td>
<td>Functional Testing</td>
<td>66</td>
</tr>
<tr>
<td>11</td>
<td>The Cache Controller</td>
<td>69</td>
</tr>
<tr>
<td>11.1</td>
<td>Overview</td>
<td>69</td>
</tr>
<tr>
<td>11.2</td>
<td>VHDL Implementation</td>
<td>70</td>
</tr>
<tr>
<td>11.2.1</td>
<td>Port Structure</td>
<td>70</td>
</tr>
<tr>
<td>11.2.2</td>
<td>BlockRAM Organisation and Operation</td>
<td>71</td>
</tr>
<tr>
<td>11.2.3</td>
<td>Addressing in the cache</td>
<td>72</td>
</tr>
<tr>
<td>11.2.4</td>
<td>Dataflow</td>
<td>72</td>
</tr>
<tr>
<td>11.2.5</td>
<td>Cache Controller state machine</td>
<td>74</td>
</tr>
<tr>
<td>11.2.6</td>
<td>Internal Operation</td>
<td>76</td>
</tr>
<tr>
<td>11.2.7</td>
<td>Functional Simulation</td>
<td>77</td>
</tr>
<tr>
<td>12</td>
<td>The SDRAM Controller</td>
<td>85</td>
</tr>
<tr>
<td>12.1</td>
<td>SDRAM Controllers</td>
<td>85</td>
</tr>
<tr>
<td>12.2</td>
<td>VHDL Implementation</td>
<td>87</td>
</tr>
<tr>
<td>12.2.1</td>
<td>Port Structure</td>
<td>87</td>
</tr>
<tr>
<td>12.2.2</td>
<td>Controller construction</td>
<td>88</td>
</tr>
<tr>
<td>12.2.3</td>
<td>Refresh Timer Design</td>
<td>91</td>
</tr>
<tr>
<td>12.3</td>
<td>SDRAM Controller Functional Simulation</td>
<td>93</td>
</tr>
<tr>
<td>12.3.1</td>
<td>SDRAM Startup</td>
<td>93</td>
</tr>
<tr>
<td>12.3.2</td>
<td>Burst Write</td>
<td>94</td>
</tr>
<tr>
<td>12.3.3</td>
<td>Burst Read</td>
<td>94</td>
</tr>
<tr>
<td>12.3.4</td>
<td>Refresh</td>
<td>95</td>
</tr>
<tr>
<td>13</td>
<td>The Address Unit</td>
<td>97</td>
</tr>
<tr>
<td>13.1</td>
<td>Overview</td>
<td>97</td>
</tr>
<tr>
<td>13.2</td>
<td>VHDL Implementation</td>
<td>97</td>
</tr>
<tr>
<td>13.2.1</td>
<td>Port Structure</td>
<td>97</td>
</tr>
<tr>
<td>CONTENTS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>------------------</td>
<td></td>
</tr>
<tr>
<td>C.2.1</td>
<td>Address Unit - VHDL Code ........................................... 144</td>
<td></td>
</tr>
<tr>
<td>C.2.2</td>
<td>Address Unit - VHDL Testbench ................................... 145</td>
<td></td>
</tr>
<tr>
<td>C.3</td>
<td>Cache .............................................................. 148</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cache - VHDL Code .................................................... 148</td>
<td></td>
</tr>
<tr>
<td>C.4</td>
<td>Cache Controller .................................................. 150</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cache Controller - VHDL Code ..................................... 150</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cache Controller - VHDL Testbenches ................................ 153</td>
<td></td>
</tr>
<tr>
<td>C.5</td>
<td>Command Controller .................................................. 164</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Command Controller - VHDL Code .................................... 164</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Command Controller - VHDL Testbench ................................ 169</td>
<td></td>
</tr>
<tr>
<td>C.6</td>
<td>EDAC Unit .......................................................... 174</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EDAC Unit - VHDL Code ............................................. 174</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EDAC - VHDL Testbench .............................................. 180</td>
<td></td>
</tr>
<tr>
<td>C.7</td>
<td>Parallel Unit ....................................................... 182</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Parallel Unit - VHDL Code ........................................... 182</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Parallel Unit - VHDL Testbench .................................... 183</td>
<td></td>
</tr>
<tr>
<td>C.8</td>
<td>Refresh Timer ........................................................ 186</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Refresh Timer - VHDL Code ......................................... 186</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Refresh Timer - VHDL Testbench .................................... 186</td>
<td></td>
</tr>
<tr>
<td>C.9</td>
<td>SDRAM Controller ..................................................... 188</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SDRAM Controller - VHDL Code ....................................... 188</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SDRAM Controller - VHDL Testbench .................................. 191</td>
<td></td>
</tr>
<tr>
<td>C.10</td>
<td>Serialising Unit ................................................... 194</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Serialising Unit - VHDL Code ....................................... 194</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Serialising Unit - VHDL Testbench .................................. 195</td>
<td></td>
</tr>
</tbody>
</table>
# List of Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>DIMM</td>
<td>Dual Inline Memory Module</td>
</tr>
<tr>
<td>DRAM</td>
<td>Dynamic RAM</td>
</tr>
<tr>
<td>EDAC</td>
<td>Error Detection and Correction</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>LEO</td>
<td>Low Earth Orbit</td>
</tr>
<tr>
<td>MMU</td>
<td>Mass Memory Unit</td>
</tr>
<tr>
<td>OBC</td>
<td>Onboard Computer</td>
</tr>
<tr>
<td>PU</td>
<td>Parallel Unit</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>SDRAM</td>
<td>Synchronous DRAM</td>
</tr>
<tr>
<td>SEL</td>
<td>Single Event Latch-ups</td>
</tr>
<tr>
<td>SEU</td>
<td>Single Event Upsets</td>
</tr>
<tr>
<td>SU</td>
<td>Serialising Unit</td>
</tr>
<tr>
<td>SRAM</td>
<td>Static RAM</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very-High-Speed Integrated Circuits</td>
</tr>
</tbody>
</table>
List of Figures

2.1 State transitions of a typical SDRAM component ........................................ 9
2.2 Typical SDRAM bank logical structure .......................................................... 13
2.3 Address break-up for activate and read/write commands to the same bank .... 14

5.1 Simple System Block Diagram ..................................................................... 27
5.2 Model 1 Block Diagram for Internal Data path ............................................. 28
5.3 Model 2 Block Diagram for Internal Data path ............................................. 29
5.4 Model 3 Block Diagram for Internal Data path ............................................. 30

7.1 Command Controller port diagram ................................................................. 36
7.2 Command Controller State-Machine ............................................................. 39
7.3 Write command event flow ........................................................................... 43
7.4 Command Controller functional simulation : startup ..................................... 44
7.5 Command Controller functional simulation : address loading ....................... 45
7.6 Command Controller functional simulation : write operation ....................... 46
7.7 Command Controller functional simulation : read operation ......................... 48
7.8 Command Controller functional simulation : refresh case 1 ....................... 48
7.9 Command Controller functional simulation : refresh case 2 ....................... 49
7.10 Command Controller functional simulation : refresh case 3 ...................... 49

8.1 Parallel Unit input and output signals ............................................................ 50
8.2 Parallel Unit data register double buffering ................................................ 51
8.3 Parallel Unit functional testing : loading end ............................................... 53
8.4 Parallel Unit functional testing : loading cycle ............................................ 54

9.1 Serialising Unit input and output signals ...................................................... 55
9.2 Serialising Unit dataflow .............................................................................. 57
9.3 Serialising Unit functional testing : reload ................................................... 58
9.4 Serialising Unit functional testing : full read ............................................... 59

10.1 Port structure of EDAC ............................................................................... 63
10.2 EDAC VHDL implementation diagram ....................................................... 64
10.3 Registered EDAC .............................................. 65
10.4 EDAC Unit functional testing, part 1 .......................... 67
10.5 EDAC Unit functional testing, part 2 .......................... 68

11.1 Cache address structure ....................................... 70
11.2 Cache Controller input/output ................................. 70
11.3 BlockRAM input/output ........................................ 71
11.4 A typical SDRAM write operation with bank switching .......... 74
11.5 A typical SDRAM read operation with bank switching .......... 75

11.6 Cache Controller state machine .................................. 76
11.7 Functional testing of the Cache Controller: single SDRAM write. 78
11.8 Functional testing of the Cache Controller: single SDRAM write detail 79
11.9 Functional testing of the Cache Controller: cycled SDRAM writes. 80
11.10 Functional testing of the Cache Controller: SDRAM read. ........ 81
11.11 Functional testing of the Cache Controller: SDRAM read detail. .... 82
11.12 Functional testing of the Cache Controller: cycled SDRAM reading. 84

12.1 Single burst transfer timing ..................................... 86
12.2 SDRAM Controller input/output .................................. 87
12.3 Autoprecharge SDRAM Controller state machine ............... 89
12.4 SDRAM state machine sequences .................................. 90
12.5 Refresh timer input/output ....................................... 92
12.6 Functional simulation of Refresh Timer .......................... 92
12.7 SDRAM Startup sequence ....................................... 94
12.8 SDRAM Burst Mode: Write ...................................... 95
12.9 SDRAM Burst Mode: Read ...................................... 96
12.10 SDRAM read with refresh during burst ......................... 96

13.1 Address Unit input and output ................................... 98
13.2 Loading of start and end addresses ................................ 100
13.3 Loading of start and end addresses (variation) .................... 100
13.4 Address Unit functional simulation: address loading .......... 101
13.5 Address Unit functional simulation: address end ............... 103
13.6 Address Unit functional simulation: full operation ............ 104

14.1 Detail block diagram of the VHDL system ......................... 106
14.2 VHDL System functional simulation: startup and full write ........ 107
14.3 Full System functional simulation: full read .................... 109

15.1 Configuration bitstream checking ................................ 112
15.2 Bitstream comparison for Readback ............................. 113
LIST OF FIGURES

15.3 JTAG connections for multiple-device programming ............... 114
15.4 Configuration and Mask PROM connections with FPGA ............ 115
15.5 Delay in clock to SDRAM .......................................... 115
15.6 Phase shifted SDRAM clock ......................................... 116
15.7 Simple use of a DLL to provide a stable clock ....................... 116
List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>SDRAM state descriptions</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Word address composition</td>
<td>14</td>
</tr>
<tr>
<td>15.1</td>
<td>Truth table for configuration masking</td>
<td>113</td>
</tr>
<tr>
<td>B.1</td>
<td>System Commands and Status Values</td>
<td>124</td>
</tr>
<tr>
<td>B.2</td>
<td>State encoding for the Command Controller</td>
<td>124</td>
</tr>
<tr>
<td>B.3</td>
<td>SDRAM Command Word : Bit structure</td>
<td>125</td>
</tr>
<tr>
<td>B.4</td>
<td>Table of SDRAM Commands Used</td>
<td>125</td>
</tr>
<tr>
<td>B.5</td>
<td>Table of SDRAM Controller States</td>
<td>126</td>
</tr>
<tr>
<td>B.6</td>
<td>Mode Register Definition</td>
<td>126</td>
</tr>
</tbody>
</table>
Conventions

- Commands and states will be written in **boldface** and signals directly referred to in *italics*.

- The inverse of a signal will be indicated by a bar over the signal name: $\overline{RD}$.

- British English Spelling Rules will be used.
Chapter 1

Introduction

1.1 Background

Satellites carry various devices that generate data, possibly at high bitrates. Unfortunately, ground stations are not always available to download the large amounts of data immediately. It is therefore necessary to stream the data into storage at high speed. The data must be stored reliably until the next download opportunity. This could be a sizeable fraction of the time between ground station contacts. Since Low Earth Orbit (LEO) satellites move very fast to maintain orbit, they are in contact with a ground station for a limited amount of time. Transmitting to a ground station, the data must be streamed from storage at high speed to the transmitter to minimise download time. Other operations on board the satellite might also need a large high speed temporary storage for data that will be required again at a later stage.

The device used for this kind of data storage is called an MMU (Mass Memory Unit). Basic requirements of the system include (amongst others): large storage, high speed, low power consumption and minimal space. It should also be resistant to the hostile space environment. These properties are not independent of each other: an increase in speed would also mean an increase in power consumption. Such relationships exist for the other properties as well. The goal of the design procedure is to identify and specify the most important requirements and compromise on the less important ones.

In this thesis the feasibility of using an FPGA (Field Programmable Gate Array) to form the core of the system is evaluated and such a system is implemented. The implications of using an FPGA and other commercial grade components in space are discussed. The FPGA will form the interface to SDRAM (Synchronous Dynamic RAM) devices which will also be evaluated for use in this context.

The data the MMU stores would normally be presented in a streamed format with no per-
word addressing. A typical example would be imager data. Assumptions cannot be made about the intelligence of the sources and destinations of the data. It should therefore be possible to set up the system prior to data transfer by an intelligent but separate unit, possibly the OBC (Onboard Computer), to make the actual transfer of the data as simple as possible.

The altitude at which LEO satellites operate has high levels of radiation present. There is a constant influx of charged and heavy particles that cause malfunction in electronic devices, data corruption and possibly fatal faults. Methods for testing system health and preserving the validity of stored data will be discussed.

1.2 Document Overview

Introduction

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

Part I: Technology Overview
Part I contains detail information on the technologies used for the MMU. Specifically the SDRAM and FPGA technologies are studied closely.

- Chapter 2: Synchronous DRAM Technology
  SDRAM technology is discussed in detail in terms of operation, different types and properties.

- Chapter 3: Field Programmable Gate Array Technology
  FPGA technology is discussed. FPGA internal operation, areas of environmental vulnerability and verification of programming are the main topics.

Part II: VHDL Design
A specification is outlined to which the MMU should conform. The different components of the MMU is designed and discussed in detail. All the components are simulated and the simulations are analysed. The complete system is assembled and simulated in full. Finally some hardware recommendations are made.

- Chapter 4: System Specification
  A specification for the system is outlined with regards to the physical connections, operation, addressing, power consumption and reliability.
CHAPTER 1 — INTRODUCTION

- **Chapter 5: VHDL Top level design**
  Requirements of the MMU is translated to a system-level modular design. Strategic decisions regarding the data path and placement of the components are detailed.

- **Chapter 6: I/O Block Implementation**
  The interface of the MMU to the rest of the satellite is outlined and the operation discussed. Interfacing issues is identified and resolved.

- **Chapter 7: The Command Controller**
  The design of the central state machine synchronising the MMU’s operations is discussed. It’s role in the operation of the other components is explained in detail.

- **Chapter 8: The Parallel Unit**
  The design of the first stage in widening the data bus is discussed.

- **Chapter 9: Serialising Unit**
  Designing a component for reducing the bus-width from the SDRAM is discussed.

- **Chapter 10: Error Detection and Correction Unit**
  The design of the EDAC Unit with registered and unregistered versions is discussed and simulations done to verify the timing impacts of the component.

- **Chapter 11: The Cache Controller**
  This unit is designed as the interface between the cache and the SDRAM. Methods to ensure proper bi-directionality is explained in detail.

- **Chapter 12: The SDRAM Controller**
  Two different designs for controllers are inspected. One design is chosen and adapted for this specific application.

- **Chapter 13: The Address Unit**
  The MMU handles the addressing internally after addresses has been set-up. The design of the unit and its interface is discussed.

- **Chapter 14: Full System Integration**
  The full system assembly and operation is outlined here. A full simulation of the VHDL design is done and discussed.

- **Chapter 15: Hardware**
  Some hardware designs and suggestions are highlighted in this chapter.
Part V Conclusions and Recommendations

- Chapter 16: Conclusions and Recommendations

Some conclusions and recommendations are made regarding the existing implementation.
Part I

Technology Overview
Chapter 2

Synchronous DRAM Technology

2.1 Background

DRAM is a high density memory architecture used in high-bandwidth applications. It has an increased bit-density over conventional SRAM by using single-capacitor memory cells, as opposed to the 4- to 6-transistor cells in SRAM. This increase in density is not without cost, as it is more complex to control conventional DRAM and design systems that utilise it. Its asynchronous nature makes it difficult to interface to synchronous systems.[1]

Many different types of DRAM exist today [2], each with its own advantages and disadvantages. Among these are:

Asynchronous types: Fast Page Mode DRAM (FPM DRAM) and Extended Data Out DRAM (EDO DRAM) of which EDO DRAM was the most popular due to its higher transfer rates.

Synchronous types: Synchronous DRAM (SDRAM), Rambus© DRAM (RDRAM) and Double Data Rate SDRAM (DDR SDRAM). Normal SDRAM is currently (2001) the most widely used DRAM, but DDR SDRAM is gaining popularity due to its superior throughput.

Of the 2 main types, SDRAM is the obvious choice for this system because of its synchronous nature and its relatively simple control, compared to the other synchronous types. The core technology for both SDRAM and DRAM is the same, but SDRAM has a pipelining design that increases its burst access times to above that of conventional DRAM.[2] SDRAM also addressed some of the difficulties with controlling conventional,

---

1Rambus is the property of Rambus Inc.
asynchronous DRAM, making interfacing to synchronous systems easier. The small footprint, high density, high speed, low cost and low-power data retention modes make the use of SDRAM devices in space an exciting possibility.

In terms of maximum throughput, DDR SDRAM would be a better choice. However, the speed of the system is not dictated by the type of memory and therefore the added speed would not benefit the system. At the conception of the project, DDR SDRAM was still more expensive and required a more complex controller and its full potential would not have been utilised by the system at the time.

2.2 Internal Hardware Specifics

Internally, a single SDRAM chip can be organised into up to 4 physically separate but identical banks. These banks can be controlled independently and make interleaving possible, useful in eliminating delays caused by issuing commands. The exact bank structure is discussed in Section 2.3.2.

To be able to attain the word-widths that would be needed for this design, single chips would have to be grouped together in a matrix type organisation. It would be difficult to obtain effective board space usage and could be expensive to produce due to the custom PCB that would have to be designed and manufactured. As it turns out, this has already been done by the SDRAM manufacturing companies (and others) and is in widespread use in the computer industry.

Single SDRAM devices are grouped together to form a prefabricated Dual Inline Memory Module (DIMM). It combines wide data word-widths and small footprints with large storage (±2GB) and simple operation. DIMMs are operated virtually identical to single devices.[3]

Depending on the word-width and bit-density of the SDRAM device used, the organisation on the DIMM may consist of more than one group of SDRAM devices to achieve the full address space. The devices in the different rows are addressed by chip select lines.[4]

2.3 SDRAM operation

SDRAM operation have three important aspects. Firstly, SDRAM is command driven. Data transfer sequences are therefore always preceded by a command sequence. Secondly,

\[\text{64-bit normally or 72-bit with error correcting code capacity.}\]
SDRAM must be refreshed\(^3\) periodically to ensure the stored data does not degrade. Crucial data transfers could be interrupted with a refresh with incorrect scheduling. The above-mentioned factors make the control of SDRAM non-trivial, keeping in mind the control-mechanism must provide a transparent interface to the SDRAM, masking these issues.

It is the third aspect that makes SDRAM powerful. Reading and writing can be done in bursts of considerable lengths in which data can be read or written on every positive-going clock edge, without explicit addressing. As this happens synchronously, timing analysis of the rest of the system is simplified. The throughput lost when issuing commands and refreshing can be partially recovered during these burst transfers.

In burst mode, the SDRAM bursts data continuously to or from internally generated memory addresses\(^4\). Only the starting address is set up and the SDRAM keeps track of further addressing internally. These bursts provide/accept a new data word on every clock cycle and achieve very high peak transfer rates if run at high clock frequencies and board design is done carefully. Using SDRAM DIMMs (see below), average transfer rates could approach 800MB/s with a 100MHz clock. DDR SDRAM can achieve transfer rates significantly higher (3.2 GB/s @ 200MHz). Calculations in Appendix A.2.

### 2.3.1 State Operation

The operation of SDRAM is best described by a state transition diagram, Figure 2.1, taken from [3].

A short description of the states is given in Table 2.1.

The initial state at power-on is precharge\(^5\). All banks must be precharged for operation to continue normally. From here there is an automatic transition to idle. This is the state in which the user will find the SDRAM after power-on. To ensure proper set-up of operational parameters, a mode register set operation must be executed before normal data transfers can begin.

---

\(^3\) Refreshing is the process of reading and re-writing a memory cell's data to combat charge leakage.

\(^4\) The addressing sequence can be set to any number of modes, as stated in the datasheets of the respective modules. These modes are set in the mode register of the SDRAM.

\(^5\) This is not true for all SDRAMs, hence the forced precharge in the implementation.
Figure 2.1: State transitions of a typical SDRAM component[3]
故障 | 状态描述
---|---
Idle | 该状态中，所有操作开始。
Row active | 行地址的选取在这个状态中。需要在 Idle 状态下执行一个 activate 命令。
Precharge | 操作在一个行地址上结束并在另一个行地址上开始。自动返回至_idle状态。
Read & write | 数据传输操作，读取和写入被执行。当读取或写入命令在行活跃状态下提供时，完成要访问的地址。自动返回至_idle状态。
Read & write with auto-precharge | 类似于读取和写入操作，但读取和写入操作结束后自动进行预充电操作。这简化了控制并减少了在顺序操作中对 SDRAM 的命令访问。
Suspend | 低功耗状态，在该状态下读取或写入操作暂时停止但不会被中止。通过 CKE 线控制。
Mode register setting | 状态中设置操作参数的模式寄存器。可以从_Idle 状态进入并自动返回至_Idle状态。
Auto refresh | 自动刷新命令在_Idle状态下执行，整个行的刷新将开始。Idle状态在完成时自动返回。
Self refresh | 低功耗数据保留状态，刷新由设备独立处理。无需外部刷新。
Power down | 低功耗状态，所有输入缓冲器关闭。Idle状态在终止时自动进入。

**Table 2.1: SDRAM state descriptions**

**Precharge**

The reading of an SDRAM cell's contents is made possible by sense-amplifiers. Before any access, these sense-amplifiers must be set to an intermediate state. Any current flow due to the voltage stored in the SDRAM-cell capacitor will force the sense-amplifier into one of two states, indicating a high or low stored voltage. The amplifier stays in this state. If a different row is to be read, for example, the amplifier would still contain the
previous read value. It is therefore necessary, in preparing for accessing other rows, that
the amplifier be reset to the intermediate state. This action is called precharging the
sense-amplifier. Precharging a bank de-activates any active rows.

Two precharge commands exist: “precharge (selected) bank” and “precharge all banks”.[3]

Mode register write

The mode register specifies various operating modes of the SDRAM. These are: burst
length, burst type, CAS latency, operating mode and write burst mode. Setting the
mode register for the SDRAM involves issuing the command itself and providing the
parameters on the next clock cycle. The parameters are set up through the normal
address bits. The functions of the various bits are listed in Table B.6, taken from [5].

Affecting the SDRAM Controller's operation would be the burst length, burst type
(Section 2.3.4) and CAS latency (Section 12.1).

Important to remember is that all banks must be in the precharged state before a
MRS
operation can be performed. In this design explicit precharging is only necessary during
SDRAM startup. During normal operation all banks are already precharged due to the
'with-precharge' commands used for reading and writing.

Read and Write

When in idle, an activate command can be given, specifying the row being accessed. The
device then enters the row active state. From here, a read, write, read-with-autoprecharge
or write-with-autoprecharge command can be given to start a data transfer.

In normal(burst) operation, upon receiving a read or write command, the device would
enter the read or write state, transferring a data word on every clock cycle until the preset
burst length has been reached. See Section 2.3.4 for details on burst transfers.

There are two ways of handling the precharge operation with read and write:

- In the case of standard (no autoprecharge) read and write, the device would au­
tomatically return to the row active state. When a different row is accessed, a
precharge command must be given from the row active state, returning the device
automatically to idle. From here a new activate command and then a read or write
must be given.

- In the case of read and write with autoprecharge, the precharge state is entered
after the burst transfer is completed, changing the current active row's status to
inactive. Hereafter, the idle state is entered. All banks are thus already precharged when a new access is started. It is, however, necessary to activate the relevant row again if another data transfer is desired, but no explicit precharge command is ever required.

**Refreshing**

The external controller handles all refreshing when the device is operating normally. Two possible refreshing schemes exist: burst refresh and distributed refresh. Burst refresh would entail refreshing all the device's rows directly one after the other, once every 64ms. Distributed refresh would space individual row refreshes between data accesses, so that all rows are still refreshed within 64ms. A combination of the two schemes is also possible.

Both these methods would take similar total times to complete, but burst refresh would disable data accesses for quite a long time once every 64ms. Distributed refresh occurs more often (more complex control), but only takes a fraction of the total refresh time, which is less disruptive to continuous data flow.

The device can only be refreshed when all banks are in precharged state. This is especially important when using distributed refresh, since the last operation could have left a row active, which must then be precharged.

From Figure 2.1 it can be seen that the device can only be refreshed when in the idle state. During normal operations, the auto-refresh command would be used. An internal address counter holds the next row due for a refresh and increments after a refresh has been performed. The precharge state is entered automatically.

To refresh data in a low power mode, self-refresh can be used: all refresh operations are handled automatically by the device. While the clock-enable lines are low, the device utilises an internal clock to maintain timing requirements. It is recommended that a full burst of auto-refreshes are done after exiting self-refresh, before resuming normal operation.

### 2.3.2 Addressing

If the physical structure of a specific 128MB DIMM from Samsung is inspected, it consists of two groups of 9 SDRAM devices. Each device has a word-width of 8 bits, in parallel

---

6 Samsung M374S1623ETS [7]
7 Unknown device
realising the 72-bit word of the DIMM, including the error correcting code (ECC) byte. The second group extends the addressing space of the DIMM to 128MB, using chip select lines.

Internally, the individual devices are divided into four banks using two bank addressing bits. All the devices in a group operate on the same bank address, using the same internal bank. Being able to access separate banks at the same time makes it possible to hide some of the time lost during the issue of commands. The logical structure of an SDRAM bank is shown in Figure 2.2.

Each bank in turn is organised logically into 8 ‘pages’ of 4096 rows and 512 columns, having row and column address decoders to select the correct lines. Every bank in a group sees all 12 address lines (A11-A0), each unique address pointing to a bit in each of the 8 pages. The 8 pages constitute the 8-bit word per device (chip).

To be able to access the 128MB of data, organised into 16M-words, a 24-bit word-address is needed (WA23-WA0). Logically, this is divided into the column, row, bank and chip select addresses as set out above and seen in Figure 2.3. The bank address is provided with the activate command as well as read and write commands, in case of interleaving. The row address is provided with the activate command and the column address with the read/write command.
2.3.3 Bank Access and Precharge

As seen in Figure 2.3, not all the address lines are used when specifying the column address (during read/write), making other uses for them possible. For example, in the case of the Samsung device (and others), A10 is used to indicate whether autoprecharge is enabled or disabled for a specific read or write command. In other words, when the column address is specified, the A10 line is not used for addressing, but seen as part of the command. Similarly, in the precharge command, A10 indicates multi-bank or single bank precharging.

2.3.4 Burst Access

'Burst access' refers to the write or read of data on consecutive active clock edges only specifying the row and the starting column. The required column addresses are generated

<table>
<thead>
<tr>
<th>Command</th>
<th>Pins Used</th>
<th>eff. address bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>activate</td>
<td>A11-A0 + BA1-BA0</td>
<td>14</td>
</tr>
<tr>
<td>read/write</td>
<td>A8-A0</td>
<td>9</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>23</td>
</tr>
</tbody>
</table>

Table 2.2: Word address composition
automatically by the SDRAM module according to the burst type\(^8\), set in the mode register\(^2\).

The burst access lengths generally available are: 1, 2, 4, 8 or full page. Full page lengths depend on the specific device and values of 256, 512, 1024 and 2048 are typical. All burst transfers except full page terminate automatically. Burst transfers may also be terminated prematurely by an appropriate command if autoprecharge mode is not used. In full page mode transfers will continue indefinitely (wrapping back to the start of the row) and unless terminated, will still access the same active row. This will result in multiple accesses per column address if care is not taken to terminate the transfer timeously. Burst terminate commands include burst stop, precharge or a new read/write command.\(^3\)\(^8\)

Burst termination can only be used if autoprecharge is disabled. This necessitates an additional precharge command to close the current row, should a different row require activation.

### 2.4 Power

Another attractive feature of some SDRAM devices is the low-power data retention modes. In these low-power, self-refresh modes, the device typically uses less than 1\% of its normal operating power while still retaining its data.

Care should be taken with the CKE signals, since they are used to perform power down operations when \(\overline{CS}\) is high. If an SDRAM row must not accept commands and if no special power down actions are to be performed, then all CKEs should be kept high, keeping all the clocks enabled.

\(^8\)Column addresses can be generated either sequentially or interleaving.
Chapter 3

Field Programmable Gate Array Technology

3.1 Background

Traditionally, when an Integrated Circuit (IC) was desired that could perform a certain function fast and effectively, either a design was available off the shelf, or a custom circuit had to be designed. Such a chip was called an Application Specific Integrated Circuit (ASIC). Designing these chips was a tedious process, expensive in both time and money. ASIC's could provide superior performance to other solutions, unfortunately for some the cost was simply too high.

Nowadays, with the advent of the FPGA, development of custom digital chips need not be so expensive. FPGAs provide similar functionality with performance approaching those of ASICs. For products that would not be produced by the million, FPGAs provide an excellent alternative. Even a design that would ultimately target an ASIC could be done, at least in part, in the reprogrammable logic environment.

Specific technology and internal structure of FPGAs in general are discussed in Section 3.2 and in the sections thereafter the pro's and con's of specific types of FPGAs are discussed in relation to application in this design.

3.2 Technology

The internal structure of the FPGA can be described as an interconnected matrix of identical functional blocks that have both logic and stateholding capabilities. To make complex logic functions and large scale register-based structures possible, the intercon-
Connections between the functional blocks must be flexible. Finally, input and output of the device must be compatible with components expected to interact with the FPGA.

To be able to implement some unique design in the FPGA, the functional blocks and their interconnections must be configurable. This would imply that there are some type of memory structure that will represent this information.

### 3.2.1 Configurable Functional Units

The basic units in the FPGAs that provide logic and stateholding elements are usually constructed using one or more of the following primitives [9]:

- **Lookup tables (LUT)** The truth table of the function to be implemented is loaded into memory and the correct output value is selected by the input values to the LUT.

- **Multiplexers** Any logic function with $n$ inputs can be fully implemented by a $n : 1$ multiplexer [6].

- **Physical Gates** Some FPGAs have gates that can be interconnected to form the desired function, similar to PLD and PAL devices. Low delay per stage, useful for fast designs.

- **Flip-flops** Dedicated flip-flops are present in most FPGAs. They form the main state-holding elements in the FPGA and are fed by combinational logic inputs of the types mentioned above. Since they are expected to operate synchronously, they normally use dedicated, low skew clock networks distributed throughout the FPGA.

Some fine grain architectures, like the Actel ProASIC series FPGAs, have functional blocks (logic tiles) consisting of multiplexers and gates. A flip-flop can be synthesised from a single tile if needed. The whole design is reduced to gates which simplifies creating an ASIC once design is completed. Proprietary software is not required for development [10].

More course grain architectures, like the Xilinx Virtex, have basic functional blocks (logic cells or LCs) containing all of the abovementioned elements: lookup tables, flip-flops, multiplexers, gates and some additional carry logic. In the Virtex, these LCs are grouped into 2-LC slices, grouped in turn to form configurable logic blocks (CLBs). These architectures are strongly hierarchical, providing easy scalability and very accurate performance estimation [11].
CHAPTER 3 — FIELD PROGRAMMABLE GATE ARRAY TECHNOLOGY

3.2.2 True RAM

Some FPGAs contain true RAM on the IC-die that the user can configure as various types of memory. One example would be Dual-Port RAM, which will be used in this device as cache.

3.2.3 Choosing a Suitable FPGA

Currently there are quite a few FPGA manufacturers in the market. Among these are: Xilinx Incorporated, Altera Corporation and Actel Corporation. Each of these companies plays a different role in the FPGA industry.

Let us consider the requirements of an electronic system in space. Firstly it must be fault tolerant in terms of errors that are either masked out or explicitly corrected. Secondly the system must be reliable and not fail completely and prematurely. Thirdly the system must be able to be fixed (or shut down) if a fault occurs that would jeopardise the safety of the system and spacecraft.

In choosing a suitable device for the implementation of the logic, the mentioned reliability requirements must be weighed against the functional requirements that the system must fulfill. The vendors mentioned above all have devices available that are unique in some way.

Actel Corporation

The most reliable path to choose would be to use Actel’s antifuse devices. Of all the FPGA technologies it is the most unaffected by radiation and thus very suitable for space applications. However, the very reason for its robustness is the fact that it is not reprogrammable. Programming the device is done by selectively applying high voltages to antifuses\(^1\) that would connect resources together. This destroys a dielectricum between two conductors, which cannot be restored.

It was mentioned in the introduction that one of the attractive features of the FPGA is the fact that it can perform logic operations extremely fast and that some are reconfigurable, making it possible to change the logic while the satellite is in space. This would increase the level of redundancy available to the satellite: the device could perform tasks not normally assigned to it as well as perform new tasks only detailed after launch. Actel

\(^{1}\)The opposite of a fuse: when unprogrammed it has high resistance and when programmed, low resistance.
antifuse devices would be unsuitable in this context.

When reconfigurability is not needed, antifuse devices would be an excellent choice. High radiation tolerance and no extra configuration circuitry make it more reliable and cheaper in board space. Xilinx and Altera do not have these features on their FPGA devices.

Actel does have other products available. Its ProASIC and ProASICPLUS ranges have non-volatile, flash-based configuration memories. Although possible, re-configuration is not required after power-loss on these devices.

Actel's antifuse range is extended by high reliability (Hi-Rel), radiation hardened (Rad-Hard) and radiation tolerant devices. These types of special devices are prohibitively expensive, being used mainly in military and space applications. Features of the devices are a low susceptibility to Single Event Upsets (SEUs) and the Rad-Hard devices are guaranteed against Single Event Latch-ups (SELs) [12].

Xilinx and Altera

While there are minor difference in the functioning of the Xilinx and Altera devices, most of their devices are CMOS SRAM based. The current logic configuration is lost when power is lost and needs to be reprogrammed once power is restored. The action itself is not a problem, but because of the satellite application the extra circuitry required to do this is expensive in terms of board space. However, both these devices use configurations stored in SRAM cells. SRAM is quite sensitive to radiation and can therefore experience a change of state. This would translate to a change in the logical functioning of the device. At first glance this might not seem like a serious problem apart from it introducing erroneous data into the system. Unfortunately devices like the Virtex have bi-directional drivers in the configurable logic of the device. A flipped configuration bit could physically connect two outputs of drivers together which could cause contention and damage to the device [13].

Conclusion

Extensive testing has been done on the radiation hardened and -tolerant devices offered by both Actel [14] and Xilinx [15]. Little data is available on the same type of testing for the standard devices of these vendors and no data could be found on any such testing for Altera devices. Since it would be preferable to use commercial components, it was decided that the Xilinx Virtex FPGA would be used for this study.
3.3 The Xilinx Virtex FPGA

The actual device selected for the test design is the XCV50. It is the smallest in the Virtex range, but has all the features of the bigger devices. The device is large enough for this design, but larger devices can be used in later designs should added functionality of the design be required, or more output pins be needed. There should be no problems transferring the design to a bigger device since the amount of resources are the same or more in those devices with more pins.
Part II

VHDL Design
Chapter 4

System Specification

The broad specification mentioned in the introduction will be elaborated on in this section. This design is not part of an active satellite development programme and therefore there is no formal specification that the satellite subsystems has to conform to. This is the case for many of the points mentioned below.

4.1 Physical interconnects

This specifies how there will be connected to the MMU. In past designs three main buses are apparent: the data and control bus, the telecommand bus and the telemetry bus.[16][17].

4.1.1 Data and control bus

Data transfer and its control is handled by this bus. The data part is normally the width of the satellite data bus, but in this case will be taken as 8-bit. The data and control bus is seen here as a generic interface. If a different external interface is desired, it could be re-designed or connected to a dedicated converter.

Data would be clocked by read and write signals in the control bus, provided by the source/destination of the data. To keep the number of connections low the data port would serve a dual purpose:

- transferring data to and from the MMU and
- receiving command and addressing information for the mass memory controller.

This should not be a problem since command operation and address set up occur separate from data transfers. The type of data (command or data) appearing on the data ports
would be indicated with command flags in the control bus in addition to being expected at specific points in a transfer procedure.

Another possibility that could be explored is simultaneous read and write of data. It would possibly have to take place at lower speeds than uni-directional transfers.

The control part of the bus (or control bus) is generally a dedicated bus to each subsystem, as it should be in this design. The control in this case would not be done on the basis of explicit commands, but rather indicating that a command is being given through the data port. The only difference between data and commands would be the relevant command flag set in the control bus. It should contain both clocking signals to synchronise data as well as flags to indicate commands. Bi-directional control should be possible if a port should be able to perform both read and write operations.

### 4.1.2 Telecommand bus

The telecommand is a very basic control bus that gives the onboard computer very rough control over the subsystems. It is used, amongst other things, to completely shut down entire subsystems. This is the only function that will be implemented here, being able to shut down the operation of the MMU without compromising the buses connected to it with dead logic. In other words, the buffers should still be operational and in a high-impedance state.

### 4.1.3 Telemetry bus

The onboard computer needs basic information on the status of the subsystems. This would include parameters that are important to the health and continued operation of the satellite. In this design hardware support for current measuring would be implemented, as the devices used are sensitive to radiation in ways that could increase current consumption.

### 4.2 Command Operation

Utilising the data port as both an address and command port necessitates a definite sequence for command setup, address setup and data transfer. When in idle mode (i.e. after startup), commands should be possible. If additional information is needed following the command, it should be clocked in identical to data, using the normal clocking signals in the control bus. Both the MMU and the client clocking in the data should keep track
of the amount of bytes needed. After the information has been transferred, the system should enter the relevant state as dictated by the command initially given. After read and write commands have been given, the system should be ready to transfer data using only the clocking signals.

4.3 Addressing

It should be possible to setup the start and end addresses for transfers prior to transfers. Conventional memories would require the address to be available with the data. The streamed nature of the data in this case would make addressing unnecessary, except for the first byte transferred. As explained above, the address setup command would be used, followed with the start and end addresses clocked in as normal data. As data is read in or written out, the system keeps track of the current address. The system should not proceed past the set end-address and should enter the idle state after transfer has been completed and the end address had been reached.

4.4 Power Consumption

Power consumption is of great importance in satellite systems, especially in small systems where the only source of power is solar cells. Although no definite specification exists for this design, power consumption should be investigated and ways found to minimise it. SDRAMs and FPGAs are notorious high power devices, but with good design excessive power consumption can be reduced. Special low-power modes for SDRAMs exist and clock-frequency in FPGAs can be varied.

4.5 Reliability

The hostile space environment have drastic effects on components that would otherwise function perfectly. Different parts of the system need to be protected. This include both storage and other components. Data should be protected by implementing a Error Detection and Correction (EDAC) unit. It might also be necessary to periodically check the memory irrespective of it being read, a process called ‘washing’.

As mentioned in Chapter 3, SRAM-based FPGAs have configuration storage cells that are active during operation as well as devices that store the configuration when the device is unpowered. Both these memories are susceptible to radiation and should be checked.
Faults should be detected quick enough and not cause permanent damage to the system. Fixing correctable faults should be done in ways that would minimise disruption of the system operation.

Most of the errors in the stored data would be 'soft-errors', however, occasionally more permanent damage could be inflicted on memory cells. These are called 'hard-errors' and cannot be fixed. While it would still be possible to continue using the memory location due to the EDAC, the probability of undetected errors increase, however small it might be. It might be necessary to implement an address failure table that would track any detected errors and avoid use of that address.

This would be considerably more complex with SDRAM (see burst access) and the viability of using such a table strategy should be investigated. It is not necessarily a problem, since only 8-word bursts are used, the address allocation can be done before the burst is started, plenty of time exist. Also, see the address mapping scheme used in previous models.

At the very least, single bit correction per stored word is expected. In case of 2 errors (or possibly more), the fact should be reported so appropriate action can be taken. There are practical limitations to the detectability of more than 2 errors which should also be investigated.

Due to the large amount of combinational logic needed for the error correction, the induced time delays should be investigated to determine if it has a significant impact on the operating frequency and therefore throughput of the MMU. If so, a solution should be investigated and implemented if viable.

---

1 Configuration memory errors could result in bus-contention.

2 64-bit data word
Chapter 5

VHDL Top-level Design

5.1 Initial Concept

The control of SDRAM is not as straightforward as normal SRAM and needs a more complex controller. Because of the more complex memory controller, the operation of the entire system is more critical to data throughput. It was therefore decided to use a controller that will control the different parts of the system from a higher level. To keep the controller from becoming too complex however, the smaller functional blocks in the system will have low-level communication where it will not affect the system as a whole. The system will thus have a combination of both high-level and low-level internal communication.

In this design the satellite data bus was taken to be 8-bit. Because the data bus and its control lines will not be synchronised to the FPGA internal clock as well as timing issues, a proper I/O block is needed to achieve synchronisation. SDRAM-word is 72-bit (from 64-bit), so it is clear that the data bus must be widened internally (in the FPGA). Where in the data path this happens will be discussed in following sections. The error detection and correction unit will be implemented in the FPGA. The exact form this will take will be determined by where in the data path the unit will be placed.

To minimise the external connections (and reduce the pin count on the FPGA) the input port will be used for data as well as command and address information. During normal data transfers the port functions as a data port. When a command is given, the command flag is asserted externally and the command given is ‘written’ through the data port. This is also used to load the start and end addresses into the system. This scheme is possible because the MMU will not normally do random accesses and the time of data transfer is generally known long before it is due. This gives the satellite’s onboard computer enough time to set up the MMU in this manner. Additional command-word lines or address lines
are therefore redundant. The exact I/O structure and access sequence is elaborated in later sections.

5.2 Identification/Positioning of System Blocks

The positioning of the functional blocks in the data path has a big effect on how the system will function eventually. Before the functional blocks are defined the system block diagram looks roughly like Figure 5.1. The part of the system that will be implemented in the FPGA is bounded by the box with the broken outline. This will be implemented in VHDL.

The VHDL-design is broken up in two parts:

1. I/O (Input/Output)
2. System Control

The system control block in the above diagram has to fulfill the broad requirements outlined in the initial concept. Breaking up the system control into single tasks identifies the following crucial blocks:

1. Parallel Unit - widens (time multiplexes) the data bus on the forward data path (write)
2. EDAC (Error detection and correction) - error detection and correction
3. SDRAM Controller - exclusive SDRAM Controller
4. Command Controller - accept commands and high level control of the entire system
5. Refresh Timer - data refresh timing for the SDRAM
6. Address Unit - addressing unit for the SDRAM

7. Serialising Unit - narrows (time demultiplex) the data bus on the return data path (read)

8. Cache and Cache Controller

There are a few possible ways to organise these blocks and still have a workable system. Three different models are discussed next, with diagrams to illustrate the data path.

5.2.1 Model 1

In this model the EDAC is inserted after the I/O block, see Figure 5.2. It is followed by the Parallel Unit and finally the cache. On the return path the Serialising Unit will again reduce the width of the data path. The SDRAM side of the EDAC will be 8-bit plus checkbits. If Hamming codes are used, the number of checkbits equal 5. This adds up to 13, which is a very ungainly width. Because it is neither divisible by an integer nor divides fully into 72 or 64 bits (SDRAM with/without checkbits), it is inevitable that storage space will be wasted unless a complex bit-level grouping mechanism is found. An additional problem is the delay in the EDAC because of long combinational logic paths. Because the EDAC will be on the 8-bit side of the Parallel Unit, the input data changes very rapidly, especially on the return path and the output must stabilise before it can be used further in the system. This might directly limit the maximum speed at which the system can run. There are, however, ways to reduce these delays but they are complex to implement and expensive resource-wise. It might be necessary to implement some of these methods and it will be discussed in the section concerning the design of the EDAC.

The SDRAM-side output of the EDAC has to be organised in some form as to be acceptable to the SDRAM. The word-width of the SDRAM can be either 64-bit or 72-bit. Neither is divisible by 13 and the nearest products are 65 and 78. If 64-bit SDRAM is used, only 1 bit is left over which is difficult to integrate into new data. If 72-bit SDRAM...
is used, 6-bits extra can be written, but this again leaves 7 bits to deal with on the next word, which is again difficult to easily integrate. The easiest option would be to select 72-bit EDAC SDRAM and waste 7 bits per word write (9.7%).

5.2.2 Model 2

For this model, the EDAC and Parallel Unit from Model 1 is switched around (see Figure 5.3). This increases the data path to 64-bit before the EDAC (in the forward path) and 72-bit after the EDAC. Because the final word-width is 72-bit, SDRAM with Error Correction Code (ECC) storage ability would be used. Widening the data path just after the I/O Block reduces the timing constraints for the rest of the system with careful design. Making use of the fact that the 64-bit word will change much less frequently than the 8-bit word from Model 1 could significantly increase the maximum frequency (and therefore data throughput) at which the system can be operated. The downside to this system is the increase in resources it will consume due to the larger register widths where buffering is utilised. Inherently a more elegant design, it has distinct advantages over Model 1.

5.2.3 Model 3

Only a slight variation of Model 2 (see difference in bus-widths in Figure 5.4). If SDRAM with ECC is not available, 64-bit SDRAM can be used almost as effectively. Substituting a ‘ghost’ 8th data byte, the EDAC still generates a 8-bit checkword. Because there are only 7 data bytes, the checkword can be stored in the 8th data byte’s place. This requires a minimal design change involving counters. It will also incur absolutely no throughput penalty. It is, however, a little more redundant than Model 2 because of the 8-bit checkword for only 7 data bytes. Much less so than possible schemes from Model 1 though.
5.3 Functional Description of the Blocks

In this section we will assume that Model 2 has been chosen for the design.

**I/O Block:** To be able to effectively communicate with the MMU, a simple interface has to be defined. Additionally, since the FPGA runs synchronously and from its own local clock, the incoming data and control signals has to be synchronised to this local clock to be able to be used reliably. Related to this problem is the possibility of metastability of signals sampled while they are not stable. Finally bus contention on the output ports should be avoided. The I/O Block is discussed in Chapter 6.

**Command Controller:** The Command Controller is the core controller for the entire system. All the other functional blocks are monitored by the Command Controller. The Command Controller intercepts commands given through the external connections and act accordingly. Both system level timing and directional control are handled here. Also, enabling/disabling of certain functional blocks are handled here. The Command Controller receives signals from the other functional blocks that indicate the state of that particular part of the system and response to them. The controller is not completely separate from the memory controller and is expected to perform certain memory related tasks. The controller is notified of a new incoming command with a specific command flag and the command will be present on the data input. The controller should only accept commands when in idle mode. The Command Controller is discussed in Chapter 7.

**Refresh Timer:** As seen from the operation requirements for SDRAM, one of its drawbacks is the constant refreshing that it needs to retain its data. This functional block must keep track of the time passed since the last refresh. When the time for the next refresh arrives, the Command Controller is requested to do a refresh on the SDRAM and the internal refresh timer is reset to zero when the Command Controller acknowledges. The refresh timer should only be active when the SDRAM
is active (data transfer ongoing or expected). When the SDRAM is disabled or in a low-power data retention mode, it should be disabled. The Refresh Timer is discussed in Chapter 7.3.2.

**Parallel Unit:** Because the external data bus to and from the MMU currently is 8-bit and the final word-width to the SDRAM is 72-bit (64 data bits + 8 checkbits) the data bus has to be widened in the forward path (the same goes for the return path, see also *Serialising Unit*). The Parallel Unit is discussed in Chapter 8.

**Serialising Unit:** The exact opposite of the Parallel Unit. The data-bus width is reduced to 8-bit to be read by the I/O Block (external user). According to Model 2, the Serialising Unit should be placed right before the data exits through the I/O Block. The Serialising Unit is discussed in Chapter 9.

**EDAC Unit:** The purpose of the EDAC Unit is firstly to generate error correcting codes, secondly to use previously generated codes to correct data and lastly to indicate when data has been corrected or is beyond repair. The EDAC Unit is discussed in Chapter 10.

**Cache and Cache Controller:** The nature of SDRAM necessitates the use of cache to obtain reasonable transfer rates. Delays caused by the normal operation of SDRAM requires data to be stored temporarily until it can be used by the rest of the system. If cache is not used, the maximum data transfer rates would be adversely affected to allow for the time-consuming routine SDRAM operations that is necessary to ensure data retention. The cache and its controller is discussed in Chapter 11.

**SDRAM Controller:** The operation of SDRAM is not trivial and needs to be strictly controlled by a dedicated controller. The controller implements a state-machine that ensures that the different commands are executed in the correct sequence as well as within the specified timing restrictions. The SDRAM Controller is discussed in Chapter 12.

**Address Unit:** The Address Unit serves two purposes. First, it provides an address to the SDRAM during data transfers. This includes the multiplexing of the row/column addresses and the incrementing of the current address as the transfer continues. Second, it alerts the Command Controller when the transfer is complete (end address reached). The Address Unit is loaded with a start and end address. The current address (which it outputs to the SDRAM) is incremented on every clock cycle when data transfer takes place. When the end-address is reached the Command Controller is notified. Addresses are loaded through the input data port, as with commands. The addresses are loaded byte-per-byte into the relevant registers.
and the working address set to zero thereafter. The Address Unit is discussed in Chapter 13.

**Configuration Checker:** This item is not part of the data path in the FPGA, but as discussed in Chapter 3, the SRAM configuration memory must be checked for errors due to radiation. The Vitrex series of FPGAs allow reading of this memory during normal operation. This functional block must check the configuration bitstream of the operating FPGA against the configuration stored in the PROM used for configuring the FPGA on power up. Checking the FPGA configuration is discussed in Chapter 15.1.

**Delay Locked Loop (DLL):** A DLL achieves the same effect as a PLL (Phase locked loop) with a different implementation. This is a component that is used to control clock skew, reverse clock phase and divide or multiply clocks. In the test implementation it was used on occasion to divide the clock reliably to ensure combinatorial delays are shorter than the effective clock frequency. The use of a DLL is recommended in combination with a external high-drive PLL in driving SDRAM clocks when using the internal clock of the FPGA. The DLL will ensure that the SDRAM clock is synchronised to the internal clock and the PLL will provide both enough drive current and also zero delay on the clock path. Use of the DLL is discussed in Chapter 15.2.
Chapter 6

I/O Block Implementation

6.1 Overview

The I/O block consists of signals organised into 4 ports.

Port A: Data input
Port B: Data input/output
Port C: Control input
Port D: Status output – not used

Currently Port A is input, Port B is exclusively output and Port C is fully functional as the control port. Port D is not implemented yet. Both Port A and Port B are 8-bit data transfer ports. The rd_pbn signal in Port C controls the direction of Port B. When rd_pbn = '0', the port is output and when rd_pbn = '1', the port is high-impedance.

The function of the I/O Block is the control of the external bi-directional tri-state buffers and the synchronisation of the read and write signals. Metastability is a potentially significant problem at high speeds and is discussed in Section 6.2.

6.2 Synchronisation and Metastability

FGPA’s are normally used for synchronous designs, they are digital systems with state-variables which change at times specified by a free-running clock signal [1]. Sometimes, these systems interface with other systems not running on the same clock or no specific clock at all. In this case, the system relies on previous states as well as external inputs to
compute the next state. The system is designed in such a way that the longest possible delay as a result of propagation through combinational logic and flip-flops is shorter than the clock-period. This ensures that signals generated internally are stable when needed for the next state-change. Unfortunately external signals need more attention.

6.2.1 Metastability

Unfortunately no assumptions can be made about the times at which external signals can be used directly as inputs to state-logic. The problem arises when the external signal violates the setup and hold requirements of the stateholding element using it as input. The element then enters a metastable state that lasts an arbitrary time. In this state the stateholding element’s output is unstable and exhibits unpredictable behaviour. The final value, after metastability has ceased, can be either logic high or logic low. For a specific design, the probability of metastability occurring increases with an increase in system clock frequency. Setup and hold requirements stay the same, but delay relative to clock frequency increase, leaving a smaller window in which signals can stabilise.

This type of behaviour is unacceptable in any system. It can lead to incorrect functioning of state-machines, incorrect input data and system failure.

6.2.2 Synchronisation

To avoid metastability, the external signal must be presented as stable to the synchronous system. One solution presented in [1] implements a synchroniser, containing two d-type flip-flops in series. This circuit decreases the probability of metastability by orders of magnitude. The external signal is delayed by two clock cycles before entering the synchronous system.
Chapter 7

The Command Controller

7.1 Overview

The Command Controller is the component central to the operation of the Mass Memory Unit. All control/status signals are connected to the Command Controller. The controller is built around a state machine providing correct timing of processes. In other words, the Command Controller provides high level control.

Commands are given through the system data ports and stored in the Command Controller. The controller generates control signals for the rest of the system from the stored commands. For example: a write command to the MMU would prompt the Command Controller to start the SDRAM, set the correct direction flag in the system and reset various components.

The Command Controller provides control to the small sub-components that don’t have their own controllers. Sub-components that do have their own controllers (cache, address, SDRAM) are still controlled at the highest level by the Command Controller.

During block data transfers, there are basic handshaking signals between the sub-components handling the data. The Command Controller has no direct control over these signals, but may monitor them to discover the state of the transfer. Having the components manage parts of the transfer simplifies the controller according to modular design methodology.

7.2 Command Controller Tasks

The Command Controller has the following main tasks and interactions with other components:
CHAPTER 7 — THE COMMAND CONTROLLER

1. Command Interface: Responsible for receiving and acting on user commands issued. This includes all actions relevant to starting, sustaining and terminating block data transfers, mainly the following: (i) Controlling paralleling-/serialising units (ii) Error monitoring on EDAC (iii) Cache capacity monitoring (iv) Read/write synchronisation between SDRAM and cache (v) SDRAM Controller interface.

2. SDRAM upkeep: (i) SDRAM startup commands (ii) Refresh requests.

3. Address Unit interface: (i) Setting of address registers (ii) Facilitating address shifting (iii) Detecting end of transfer signal.


7.3 VHDL Implementation

7.3.1 Port Structure

The port structure of the Command Controller is quite extensive, linking most of the sub-components in the system. See Figure 7.1.

Figure 7.1: Command Controller port diagram
CHAPTER 7 — THE COMMAND CONTROLLER

Input signal description:

- **CLK**: System clock
- **RESET_N**: Global reset
- **CMD_PA**: Command presence flag
- **CMD_IN**: User command input flag
- **WR_PA_N**: Top level synchronised write signal
- **SEL_RAM**: MMU global select
- **DATA_PA_IN**: Synchronised external data bus
- **SINGLE_ERR**: Error flags from EDAC
- **ADDRESS_LOAD_DONE**: Address Unit write address complete
- **ADR_DONE**: Address Unit counter end address reached
- **ADDRESS_IN**: Address from Address Unit
- **CACHE.EMPTY**: Cache is completely empty
- **BANK_Full, BANK_EMPTY**: Indicates whether at least one bank is full/empty
- **SER.EMPTY**: Serialising Unit needs new data
- **REF_REQ_IN**: Interrupt from the Refresh Timer, indicating SDRAM refresh is due
- **REF.ACK.SDRAM**: Refresh acknowledge from SDRAM Controller

Output signal description:

- **CLOCK.COUNT.OUT**: Temporary output signal to test value of `clock.count`
- **CURRENT.STATE.OUT**: Temporary output signal to output current state
- **SER.ADVANCE**: Serialising Unit manual advance to buffer data to be read
- **SER.ENABLE**: Serialising Unit enable
- **PAR.ENABLE**: Parallel Unit enable
- **ADDRESS_SET_PULSE**: Pulse to Address Unit to write 8-bit data to internal address register
- **RESET.ADDRESS**: Resets internal start address to 0 and end address to \((FFFFF)\)\(_{16}\)
- **ADR.BYTE**: Next byte to be loaded into the internal address registers in the Address Unit
- **ADDRESS.OUT**: Address output multiplexed between the actual SDRAM address and the Mode register value
- **CACHE_Direction**: Direction of data flow, '0' = into MMU
- **RESET.CACHE**: Resets addresses in Cache Controller
7.3.2 Operation

Command Interface

The Command Controller accepts commands from the user and prepares the rest of the system for the requested operation. Available commands are listed in Table B.1.

The received command is loaded into a register, \textit{cmd\_int}, from which control signals and state sequences are generated. The command stays in the register until a new command is loaded. The required signals for each sub-component is discussed in the chapter dealing with the particular component.

State Machine

The sequenced control required for the Command Controller is best implemented by a state machine. The state machine is implemented in two parts: a combinational part where the control and next-state signals are generated; and a registered part where the control signals and state is registered.

Figure 7.2 shows a diagram of the Command Controller state machine.

First, a list of states and their functions:

\textbf{RESET}: The power on and reset state of the controller.
Figure 7.2: Command Controller State-Machine
CHAPTER 7 — THE COMMAND CONTROLLER

NO_ACTION: This state is maintained until sel.ram is asserted.

PRECHARGE: First mandatory state after power-on. Precharges all SDRAM banks.

REFRESH.1,2: Next two mandatory states for the required auto refresh commands to the SDRAM.

MRS: Next mandatory state. This state initiates the SDRAM mode register set (MRS) sequence. The MRS flag to the SDRAM Controller (output mode_r_set) is asserted and the address lines output the mode register value.

MRS_WAIT.1, MRS_WAIT.2: Wait states to let the SDRAM finish the mode register set operation. This will depend on the speed grade of the device. (Micron datasheet: $2 \times t_{CK}$)[8]

IDLE: The controller wait state when active. Commands can only be received in this state. Cache addresses are reset. No other actions take place except housekeeping tasks (e.g. SDRAM refreshing for data retention).

WR.1: State initiated from IDLE by receiving a burst write command.

WR.IDLE: Write transfer mode idle state. The write frequency can be substantially lower than the internal clock frequency, during which the Command Controller must wait for one of the cache banks to fill up, exiting when a burst transfer is required. This state will also be exited when the end address of the transfer is reached.

WR.BURST.n: Timing states for the SDRAM burst write. Start, along with the appropriate control signals starts the transfer. There is no latency when writing to the SDRAM. A counter keeps track of the number of words transferred. When the required burst length is reached, the controller is returned to the WR.IDLE state to wait for the next bank_full signal.

ADR.1: This state loads the Address Unit in response to a Set Address A user command. The controller remains in this state until the Address Unit acknowledges that the addresses have been written.

RD.1: Similar to WR.1, activated by the Read Port B user command. Transfer from the SDRAM is immediately started if a cache bank is empty, to prepare data for when user read-out clocking starts. Also, a flag, startup, is set to assist in ensuring the Serialising Unit is set up correctly (data must be provided and shifted through the Serialising Unit prior to the first rd_pb clock sequence).

RD.IDLE: Read transfer mode’s idle state while a cache bank is emptied by user clocking. Bank_empty would cause the controller to initiate a burst read from the SDRAM.
Signal `adr.done` would indicate that all the data was read from the SDRAM, the controller would enter the ADR_DONE_WAIT state.

**RD.BURST.1,2:** Timing states for the SDRAM burst read. These states are the equivalent of the WR.BURST.n states for the read cycle and again, as with the write sequence, the transfer is started by asserting `start` when a cache bank becomes empty. These first two states are responsible for initiating the transfer from the SDRAM.

**RD.WAIT:** Additional wait states are required when reading from SDRAM due to the CAS latency. The count value used here is set in the VHDL to the latency value specified in the SDRAM module.

**RD.BURST.3:** This state manages the length count of the current burst. Additional control signals help synchronise the cache and the SDRAM. A word counter is kept and the controller is returned to RD.IDLE when the current burst transfer is complete (and the cache bank is filled). The aforementioned states are also used to read data from the SDRAM and into the cache before the user starts clocking the data out during a read cycle. (I.e. handling the first burst read from the SDRAM.)

**ADV.1:** The first of two ‘advance’ states used to load the Serialising Unit with data from the SDRAM before the user clocks it out as part of a read operation. The data is read from the already filled cache.

**ADV.2:** The second ‘advance’ state. In this state the already loaded data is shifted into the second register, linked to the external `PORT B`, to be immediately available when the client reads from the port. The first register is loaded with new data from the cache. The controller now enters the RD.IDLE state and is ready for clocking signals to drive the datapath at high speed. Because of the cache and serialising, no interruption in the data flow will occur. Data is provided by the cache as the Serialising Unit is emptied.

**ADR_DONE_WAIT:** The final state after a read operation. The state is entered after all data has been read from the SDRAM, but has not been shifted out from the cache and Serialising Unit, as can be seen from the exit requirements in Figure 7.2.

**REFR.WAIT:** This state is entered when the Command Controller is in the read, write or idle modes, but a burst transfer has not yet started. Once a burst has been started, the SDRAM Controller will wait before executing a refresh and the refresh request is not lost. However, since the SDRAM Controller will ignore `start` signals while busy with a refresh, the Command Controller should wait until a start is possible...
and then proceed, hence the need for this state. The previous state is saved and returned to after the refresh is complete.

SDRAM Startup

The power on sequence of the SDRAM is discussed in Chapter 2. The Command Controller is responsible for implementing the startup sequence. This requires having the SDRAM Controller do precharge and auto refresh commands at specific times. Since the precharging is not needed normally in this design, the state doing the precharge can be part of the state sequence exclusive to the start-up of the Command Controller, as seen in Figure 7.2.

SDRAM Refresh

The Refresh Timer is responsible for the timing of the refreshes to the SDRAM. Although the Command Controller does not generate the refresh requests, it should make sure that the refreshes does not happen at the same time as the start of a new read or write burst transfer between the cache and the SDRAM. Therefore it temporarily disables the ref_req signal fed through from the Refresh Timer, whenever start is also high.

This approach is necessary for states WR_BURST_1, RD_BURST_1 and MRS_1. The refresh request will not be lost because of this: the Refresh Timer will wait for an acknowledgement (through ref_ack) before setting ref_req low again.

Unfortunately, this will not work for states RD_1, RD_IDLE and WR_IDLE. The latter two are idle states and may be active for extended amounts of time, delaying the refresh unacceptably. In the case of RD_1, start may be asserted before the SDRAM refresh is complete, which will result in an illegal SDRAM operation. The state transitions for these states must therefore also test for refresh requests and change to the special REFR_WAIT state, allowing time for the refresh to be completed. In the VHDL, the request signal tested is ref_req.int.

For all other states, the refresh operation can continue irrespective of the current state the Command Controller is in and ref_req_in is therefore fed through to the SDRAM undisturbed.

Write Operation

A device wanting to write to the MMU will have to first set up the address by issuing an Set Adress A command. Secondly, a Write Port B command is issued.
First, the Set Address A command is given through the data port by doing a normal write (wr_pan low) with a command flag also set (cmd_pa asserted). The command is registered in the Command Controller to generate the state sequence. After the address setup command is issued, the Command Controller enables the Address Unit to load incoming addresses, which is written via the Port A data input. Once the addresses have been loaded, the Address Unit indicates load_done and the controller returns to the IDLE state.

The write command brings about a typical flow of events, illustrated in Figure 7.3.

The Command Controller prepares the system for the expected write by clearing internal addresses and flags and enabling the relevant components. The controller then enters a wait state (WR.IDLE), expecting data to be written to the data port.

The data is streamed into the MMU, synchronised to the wr_pan_f2 signal. The other components handle the transfer while the Command Controller only monitors the cache bank states. Only once a bank has been filled, the controller will initiate a burst transfer between the cache and the SDRAM, issuing commands to the SDRAM Controller and operating control lines to the Cache Controller. This wait-burst cycle continues until the pre-set end address is reached and the Command Controller enters the final idle state.

The Address Unit is responsible for checking the current address against the end-address. It asserts its done signal when the end address is reached. Since the transfer to the SDRAM is now complete, the Command Controller will not process any more data; any data received hereafter is ignored. A new command is needed for a fresh transfer.

From this example it is clear that the Command Controller mostly handles the top-level control while the lower level components control the flow of data themselves.
7.3.3 Functional Simulation

The controller was simulated using a testbench to provide the input signals.

System Startup

Shown in Figure 7.4 is a simulation of the Command Controller startup sequence.

At A, the system is reset, with the current state No Action. As soon as reset is high, the next state, Precharge All Banks is entered, seen at B. Also, the precharge to the SDRAM Controller is high and address_out changes to (0400)₁₆. The tenth address bit is part of the precharge command. After one cycle the first startup refresh cycle is entered at C and the ref_req_signal is seen to go high. An extra cycle is spent waiting (with ref_ack_sdram responding) and then another auto refresh is done at D. Finally a mode register set is done at E, with the MRS setup values output on address_out, start and mode_r_set are high on the next cycle.
Figure 7.5: Command Control functional simulation: address loading.

The controller returns to Idle at F.

Address Loading

Loading start and end addresses through Port A is simulated in Figure 7.5.

An Address Set Port A command is issued at A and the data on Port A is set. Hereafter, 6 typical write pulses to Port A is input, resulting in address_set_pulse pulses starting at B. These pulses are only 1 cycle in width, as the Address Unit expects. At C the address_load_done signal is simulated, returning the state machine to state Idle.

Port A Write Operation

A Port A write operation is simulated in Figure 7.6.

The operation is initiated at A. The controller reaches the Write Idle and waits for the Cache Controller. At B, a full bank is indicated, causing a burst write to the SDRAM at C and D. After the bank is emptied, the Write Idle state is entered again at E. Another burst is started at F. Finally, the Address Unit input adr_done simulates an end-of-address
Figure 7.6: Command Controller function simulation: write operation
event at G and the controller returns to the **Idle** state.

Throughout the operation, the output address is equal to the input address, as it should be. The addresses are only shown in the simulation to illustrate this and are incorrect since the Address Unit is not connected during this simulation.

**Port B Read Operation**

Seen in Figure 7.7 is a simulation of a read operation.

At A, the command is entered. The \( r_w \) signal indicates the read direction to be from the SDRAM. When an empty bank is indicated at C, an SDRAM read operation is started, with \( \text{start} \) high at D. The SDRAM Controller takes time to initiate the transfer so \( \text{nextword} \) only goes high at E. After the burst when a cache bank is full, the cache would have written a word to the Serialising Unit. Since this is the first word transferred in the read operation, the Serialising Unit must be advanced by two, seen at F.

The controller enters the **Write Idle** state and waits at G, but immediately sees \( \text{adr.done} \) and enters **Address Done Wait** at H. At J, both the cache and the Serialising Unit is empty and the **Idle** state is entered.

**Refreshing**

The Timer Refresh requests are simulated with the situations mentioned previously.

Case 1: The refresh request is just passed through in most of the states, shown in Figure 7.8. As can be seen here, the refresh passes through undisturbed when the state is **Write 1**, \( (04)_{16} \), and is output on \( \text{ref.req.out} \). If the SDRAM Controller responds immediately, normal operation takes place.

Case 2: The refresh request is blocked for certain states, only to become active at a later stage when more suitable. In Figure 7.9 we can see that the \( \text{ref.req.in} \) signal is not passed through to the \( \text{ref.req.out} \) output until the **Write Burst 2**, \( (09)_{16} \).

Case 3: A state change takes place to achieve a long enough delay for the refresh to finish. In Figure 7.10 we can see the state changes from **Write 1** to **Refresh Wait** and stays there for 2 cycles, before returning to the previous state. Normal operation can continue afterwards.
CLOCKING + RESET
/testbench/clk 
/testbench/reset_n 

INPUT
/testbench/cmd_in 0 4 0 
/testbench/cmd_pa 
/testbench/wr_pa_n 
/testbench/adr_done 
/testbench/bank_empty 
/testbench/cache_empty 
/testbench/ser_empty 
/testbench/address_in 02F6 

OUTPUT
/testbench/r_w 
/testbench/start 
/testbench/ser_advance 
/testbench/ser_enable 
/testbench/cache_direction 
/testbench/reset_cache 
/testbench/next_word 
/testbench/address_out 02F6 

INTERNAL SIGNALS
/testbench/clock_count_out 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF 

Figure 7.7: Command Controller functional simulation: read operation

Figure 7.8: Command Controller functional simulation: refresh case 1
Figure 7.9: Command Controller functional simulation: refresh case 2

Figure 7.10: Command Controller functional simulation: refresh case 3
Chapter 8

The Parallel Unit

8.1 Overview

Choosing Model 2 for the system design makes the data flow in the system much more natural in that the datapath is a multiple of 8 (input data port width) and thus no bytes need to be broken up and packed with other words for final storage. This is especially true for the Parallel Unit. The input data is easily packed into consecutive bytes in a register and read by the next component in the system when the register is full.

8.2 VHDL Implementation

8.2.1 Basic design

Intuitively, one would simply use a 64-bit register and clock the incoming data bytes in on the write signal going low. A 3-bit counter does the indexing. When the register is full (the counter has reached 7), the component to receive the output data will deduce from the asserted full signal that the data is valid. The counter would be reset and the next incoming byte from the I/O block can be clocked into the first byte position in the

![Figure 8.1: Parallel Unit input and output signals.](image-url)
8.2.2 Timing concerns

It was suggested that putting the Parallel Unit first in the datapath would reduce the timing constraints on the rest of the system. This is not necessarily true. In the design just described one could easily make the mistake of forgetting that the EDAC is situated between the Parallel Unit and the cache. If the cache immediately responds to the full signal, the EDAC has in effect only 2 or 3 clock cycles in which to generate the checkword. Also, the next byte arrives quickly and is clocked into the register that is used as data for the checkbit generation. This is no better than what is possible under Model 1, where it is required that the EDAC complete the generation before the next byte arrives. The solution to the problem is twofold. Firstly, the full signal from the Parallel Unit will be delayed by more than the amount of clock-cycles the EDAC would need to finish the calculation. This is easily done by a very simple state machine. Secondly, because the next byte is due to arrive, the lower bytes in the register need to be stored in a separate register, which is only updated whenever the main register is full. This ensures that the output data would not change before the EDAC code-generation is complete. A diagram of the registers are shown in Figure 8.2. (Timing analysis shows the new implementation can run at 120 MHz.)
8.2.3 Alternative implementation

The Xilinx Virtex© FPGA contains dual-port blockram (also used for the cache). This is well suited for constructing the Parallel Unit. A design was done to test this option and no problems were encountered. The control is the same as the above design. In the specific device used for the test design there are only 8 blockrams available. To be able to use the blockram as a Parallel Unit, its control (addressing etc.) needs to be separate from the cache, since data accessing to both ports of the cache and the Parallel Unit will be simultaneous. This would require different blockrams to be used for these components. To be able to use a 72-bit wide data bus on the cache, at least 5 blockrams would be used exclusively. This leaves 3 to be used by the Parallel Unit, which can only result in a $16 \times 3 = 48$-bit wide bus, which is insufficient. Implementing the Parallel Unit without blockram (in other words with CLBs) is not too resource intensive and is thus acceptable. Only the design without the second register was tested with the blockram.

Another issue is the inefficient use of the blockram in this configuration. A single blockram can hold 4096 bits, so with a maximum width of 16 bits, 256 words can be stored per blockram. Four blockrams would be used. Only the first word of all these blockrams would ever be used, resulting in wasted space of $255 \times 4 \times 16 = 16320$ bits out of a possible 16384, or 99.63%. The blockram could probably be used more effectively in another way.

8.2.4 Different data widths

Since the design of the system was based on earlier models of SUNSAT, an 8-bit data bus was assumed. Newer designs might use a wider data bus. Unless the bus-width is 64-bit, some sort of Parallel Unit would still be needed. The consequences for the current design of the Parallel Unit would lie in the timing constraints of the EDAC Unit. A wider bus would mean less writes to fill the Parallel Unit, leaving less time for the EDAC to compute the checkbits. Currently half the output is double-buffered, but bus-width and clock frequency will determine the optimal implementation.

8.3 Testing the unit

A functional simulation was done to test the operation of the unit, shown in Figure 8.4. Signal port.a shows input data, along with write signal wr.pa.n. The wr.pa.n signal may change slowly and only high-to low transitions are seen as write events.

Signal data.buffer.out indicates the 64-bit parallel output. The first four writes has no
effect on the output. When referring to Figure 8.2, one sees that these bytes are temporarily stored in a 4-byte register until the last byte has arrived. The output only changes from the 5th input byte, marked A. The first 4 bytes are added to the output by the time the last byte has been written at B.

Finally, the full signal is asserted after the last byte has been written, seen at C, but is delayed by 4 cycles.

Signal filling_up is not used at the moment.

Signal cmd_pa indicates a command is present at Port A, which should be ignored by the Parallel Unit. Seen in Figure 8.3, the output buffer is not changed after the fourth ‘write’, marked by A.
Figure 8.4: Parallel Unit functional testing: loading cycle
Chapter 9

Serialising Unit

9.1 Overview

The Serialising Unit is the inverse in function to the Parallel Unit. A 64-bit word is loaded into a register and read out sequentially by the \( rd_{.}pbn\_f2 \) external signal. 8-bit data must be immediately available to the system output port on the \( rd_{.}pbn \) signal going low (after the system has been set up for read). The Serialising Unit has control lines going to and from the Cache Controller for flow-control.

9.2 VHDL Implementation

9.2.1 Port structure

Input signal description:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK</td>
<td>System clock</td>
</tr>
<tr>
<td>RESET_N</td>
<td>Global reset</td>
</tr>
<tr>
<td>ENABLE</td>
<td>Component write enable, all other operations are possible</td>
</tr>
</tbody>
</table>

![Serialising Unit input and output signals](image)

Figure 9.1: Serialising Unit input and output signals
CHAPTER 9 — SERIALISING UNIT

| **READ_DATA_N** | Synchronised external read signal |
| **WRITE_DATA_N** | Writes data to Serialising Unit from EDAC |
| **ADVANCE** | Forced write from Command Controller |
| **DATA_IN** | Input data from EDAC |

Output signal description:

| **EMPTY** | Buffer empty signal to Cache Controller |
| **DATA_OUT** | Data output to output port of system |
| **TEMP_OUT** | Debugging signal |

The global signal \(rd\_pbn\_f2\) is connected to the \(read\_data\_n\) input of the Serialising Unit.

### 9.2.2 Design

In essence the Serialising Unit is only a 64-bit buffer between the EDAC and the output port. It is therefore logical that there has to exist a 64-bit register in the unit to store the 64-bits while it is being read out by the output port. Normally a basic 64-bit to 8-bit multiplexer would be used. This is not sufficient to satisfy the requirements of the system output port in this case. See Figure 9.2 for a diagram of the Serialising Unit.

Since the Mass Memory Unit is expected to behave similar to conventional SRAM memory, the data must be available at the output port without too much delay when \(rd\_pbn\) goes low. Unfortunately this is difficult for two reasons: Firstly, the \(rd\_pbn\) signal is delayed by two clock cycles to remove metastability, which delays the system’s response to the signal. Secondly, initiating data transfer from the SDRAM is time-consuming (this is discussed in detail in Chapter 2 and 7). The second issue is addressed by pre-reading from the SDRAM.

The enable signal enables loading of data into the 64-bit register. It does not affect reading of 8-bit data or internal operation of the unit.

### Buffering

To work around the first problem, two 8-bit registers are inserted between the output port and the 64-bit register as seen in Figure 9.2. These registers are shifted for every low on the synchronised read signal, \(rd\_pbn\_f2\), with data fed from the 64-bit register.
Data is pre-fetched from the SDRAM immediately after the `read` command is given and written to the 64-bit register. Although the Command Controller does the pre-fetching automatically, some components must still be forced to do a read action to fill their internal registers. In the Serialising Unit this is only a problem with the very first word read after a `read` command, with the two 8-bit registers empty. The Serialising Unit has a special signal, `advance`, that enables the Command Controller to do forced advances on the Serialising Unit, filling the two buffer registers from the 64-bit register. The last of the two 8-bit registers is directly connected to the output port and becomes the output whenever `rd_pbn` is low.

**Timing**

In the Parallel Unit, a problem was seen where its output data was possibly not valid long enough for the EDAC to finish its calculations and present its valid data to the cache. The same problem is found here in that the cache could switch banks (and therefore output new data) before the EDAC output to the Serialising Unit has stabilised. See Chapter 11 for the modification in the Cache Controller.

### 9.2.3 Testing the design

Functional testing of the unit was done, the resulting waveforms shown in Figures 9.3 and 9.4. The `temp_out` signal outputs the low byte of the 64-bit internal register.
An initial load and read cycle is shown in Figure 9.4.

As with the Parallel Unit, only high-low transitions of the write signal write.data.n indicates data transfer. In the simulation, the 64-bit register is loaded with the data at data.in at A. The result can be seen in the temp.out signal changing to (07)16. Data.out does not change yet, due to the delay registers.

Next, the forced advance operation is used to load the two delay registers by two pulses of the advance signal, seen at B. The output at data.out can be seen to change and after the second pulse it outputs the correct lower byte of the loaded data, (07)16.

Normal reading out of the data commences at C by a succession of read pulses on read.data.n. After six pulses, the empty signal (at D) indicates that the 64-bit register has been emptied and that new data can be written to the Serialising Unit. This is correct, considering that the first 2 bytes have already been moved out of the register earlier by the advance pulses.

Reloading

At the far left of Figure 9.3 the end of a read run can be seen, with empty going high. Thereafter a new word is loaded into the register (at A) and the output at temp.out can be seen to change accordingly. After another two read pulses, the newly loaded word’s
Figure 9.4: Serialising Unit functional testing: full read
lowest byte can be seen on \textit{data\_out} at B.

Finally, to test the \textit{enable} line, the input data is changed and an attempt to write while \textit{enable} is low is made at C. No effect is seen on \textit{temp\_out} and after two read pulses the old value of the register is seen again at \textit{data\_out}, wrapping back to the low byte of the 64-bit register, as expected.

Care must be taken to promptly reload the Serialising Unit when empty. If a read is done after \textit{empty} has gone high and the register has not been reloaded, old values of the 64-bit register will already have been loaded into the delay registers, causing incorrect data on the output.
Chapter 10

Error Detection and Correction Unit

10.1 Background

For a system to be able to correct data received it makes intuitive sense that more data will need to be transmitted that will help the receiving system identify and correct the errors.

"The object of an error correcting code is to encode the data, by adding a certain amount of redundancy to the message, so that the original message can be recovered if (not too many) errors have occurred." [18]

Naturally, the less extra information required for the same amount of error-correcting ability, the better. Also, codes to be implemented in hardware should be easily manageable and equally important, it should be easily decodable. A group of codes does exist that have these properties, linear block codes [19]. The notation for a block code $C$ with word length $n$ and information (data) bits $k$ is $C(n, k)$. (The famous Hamming codes are a subset of this large group of error correcting codes having form $C(2^r - 1, 2^r - r - 1)$ with $r = \text{no. of parity bits} = n - k$ [19][20]. Strictly, a code can only be labeled as a Hamming code when it has minimum redundacy, $\frac{2^n}{n+1} = 2^k$. [20]

To generate a code (data and codeword combined) from the data mathematically, the following equation can be used:

$$x = uG$$  \hspace{1cm} (10.1)

Where $x$ is the resultant n-bit code in the form $[m \, p]$, $u$ is the input-data vector and $G$ is the code-generator matrix. $m$ and $p$ are the original dataword and generated checkword respectively.
(Aside: Linear codes can also be completely specified by a parity check matrix $H$ because $G = H^T$ [18]).

If the generator matrix of a code is known, it is very easy to implement the encoding part of the code. It amounts to the eXclusive OR (XOR) of selected bits in the data word, which is easy to encode in VHDL and implement in an FPGA.

Also, if the code is linear, the same generation matrix is used in the decoding process to obtain the location of the erroneous bits (if any) through syndrome decoding.[18]

Syndrome decoding is usually done in hardware through table lookup which again is simple (if tiresome) in VHDL. The lookup provides an error-word in which the set bit positions indicate the error position(s) in the received word.

10.2 Implementation

In the case of a 64-bit data word input to the EDAC, a $C(n, 64)$ single-error correcting code is needed. The nearest Hamming code (optimal) is $C(63, 57)$ [19] and will be difficult to implement. Any code $C(n, 64)$ will have slightly more than optimal redundancy but the ease of implementation far outweighs this in selecting a code. A code $C(72, 64)$ does exist and was previously implemented in the IDT49c465 chip manufactured by IDT [21]. Because both the resulting code and original data is of size divisible by 8 it is likely to implement more elegantly. The encoding and decoding process is explained in [21] and will be briefly discussed here.

The checkbit generating table given [21] is in effect the generating matrix for the $C(72, 64)$ code without the ‘unity matrix’ in front. To implement this, all the flagged bits in the table need to be XORed to generate the corresponding checkbit. For example:

$$CB(0) = D(1) \oplus D(2) \oplus D(3) \oplus D(5) \oplus \ldots \oplus D(61) \oplus D(63)$$

The original data is added in front of the checkword to complete the $x = [m \, p]$ form. $x$ is then stored directly in memory (72-bit ECC SDRAM).

The decoding process is considerably longer. First, the stored data-word is ENCODED by the same encoder used in the initial encoding, giving $p'$. Then, a syndrome is generated by XORing this newly generated checkword with the stored one where $s = p \oplus p'$. Finally a table lookup is done to determine the error vector $e$ which satisfies $s = e.H^T$. $e$ will be used to correct the stored data (again a XOR): $x = x' \oplus e$. 
10.3 VHDL Implementation

10.3.1 Port Structure

The VHDL port structure of the EDAC Unit follows:

Input signal description:

- **CLK**: System clock
- **RESET_N**: Global reset
- **CHECK_WORD_IN**: Input check word
- **MEM_DATA_IN**: SDRAM side data input
- **CPU_DATA_IN**: Source data input
- **DIRECTION**: Operation direction: '0' to SDRAM

Output signal description:

- **CHECK_WORD_OUT**: Generated check word
- **MEM_DATA_OUT**: SDRAM side data output
- **CPU_DATA_OUT**: Corrected data output
- **SINGLE_ERROR**: Single error detected (corrected)
- **MULTIPLE_ERRORS**: Multiple errors detected (ambiguous)

Mem_data_out and check_word_out form $x = [mp]$. The **CHECKBIT GENERATOR** implements the function $x = [mp] = uG$ (without m). The **SYNDROME GENERATOR** implements $s = p \oplus p'$ and the **SYNDROME DECODER** does the table lookup which satisfies $s = eH^T$ and corrects the faulty bit with $x = x' \oplus e$, but only outputs $m$ as $cpu_data_out$. 
10.4 Discussion

Ultimately the type of EDAC used will depend on what the satellite will use the Mass Memory Unit for. If uncompressed images are stored, for example, the tolerance for error is much higher, since a flipped bit will only affect one pixel in the image. If heavily compressed images (or mission critical data) are stored, errors become much more problematic.

In compressed data a single bit-flip could compromise a large amount of the eventually uncompressed data, since it influences a symbol that might represent more than one data symbol or is perhaps part of the compressed file's alphabet. In the former case (uncompressed data), one might safely implement a code that corrects a single error per stored word (like the code described earlier), combined with periodic washing of the data. In the latter case, this scheme might be insufficient and more aggressive error-correction might have to be implemented.

An EDAC, although very important, does take up a lot of resources on an FPGA and careful consideration should be given to the complexity of such a component.

Using an FPGA, the \( C(72, 64) \) code might be implemented as a first iteration when the
satellite is in space. Should the EDAC’s performance be unsatisfactory, a more rigorous implementation could be used, like the one described in [22] and used on board Nano Sat.

### 10.5 Timing

Implementation of the EDAC without any stateholding elements results in a logic circuit with a lot of logic levels between flip-flops. The longer the delay between flip-flops, the lower the maximum operating frequency of the design, directly limiting the data throughput.

Some delays were observed during timing analysis. It is possible to alleviate the problem by adding stateholding elements, at the cost of having extra full-clockcycle delays before data becomes available at first.

Figure 10.3 shows the relevant positions where the stateholding elements might be inserted, compared to Figure 10.2. Although complicating the Command Controller somewhat (extra wait states, tail-end data feeding), a performance increase can be expected.

![Registered EDAC Diagram](image-url)
10.5.1 Functional Testing

A functional simulation was done on the non-registered EDAC as described, the results shown in Figure 10.4.

Storing data and checkwords

The first part of the simulation, starting at A, shows the EDAC operating in the forward direction (toward SDRAM), therefore generating check words (check_word_out) from data written to the MMU (cpu_data_in). In Figure 10.4 the input data is \((0000 \ 0000 \ 0000 \ 0000)_{16}\), generating a check word of \((OC)_{16}\). The mem_data_out is just a feed-through from cpu.data.in.

Reading and correcting data

In the second part of the simulation (from B) the direction is reversed and 'stored' data from the SDRAM is used as input (mem_data.in). At B, the uncorrupted data (same as A) along with the check word (check_word.in) can be seen to produce the correct data at cpu.data.out, with no errors being reported.

At C, the data 'read' from the SDRAM is corrupted on purpose with a single bit, while the check word is kept correct. The flipped bit is corrected at cpu.data.out and the single error is indicated by single_error being asserted.

At D, the data 'read' is kept correct, with the check word being corrupted (single bit). Again, the output data at cpu.data.out is correct, with the single error being indicated correctly.

At E, a multiple error on the 'read' data is tested. The output is not corrected, and the multiple errors are indicated by multiple_errors being asserted.

At F, a multiple error on the check word is input, with no errors on the 'read' data. Here, the output data is of course correct, although it cannot be trusted in practise, since the multiple errors are detected.

In Figure 10.5, at G, a single error both on the 'read' data and the check word is input. The data is not corrected (as expected) and the multiple_errors signal is asserted.

At H, 3 bits are changed in the 'read' data. The EDAC of course cannot correct the data, since it is not designed to either detect or correct 3 or more bit-changes and this situation overlaps with a single error situation for different source data. single_error='1' is thus a completely normal result.
Figure 10.4: Error Detection and Correction Unit
### Figure 10.5: EDAC Unit functional testing, part 2
Chapter 11

The Cache Controller

11.1 Overview

The Cache Controller is intended to control the data flow into and from the cache. It keeps track of how full/empty the cache is and provides control signals to external components to regulate data flow. The cache is the 'time buffer' between the rest of the system and the SDRAM. This is necessary because of the SDRAM's inefficiency at low transfer speeds and low burst lengths.

In the forward operation (into the Mass Memory Unit) the cache buffers the 'slow' data until a reasonable amount has been accumulated and then activates the SDRAM and transfers the data in a very high speed burst. The SDRAM idles for the remaining time until the next burst is initiated. The same principle applies for the return path (data out of the MMU). Data is quickly burst from the SDRAM into the buffer where it can be read from at a much slower speed. In the meantime the SDRAM idles (possibly) in a low power, data retention mode.

To be able to achieve higher performance, the data flow from/to the external ports must not be interrupted. Thus data must be able to be read from the cache while new data is being written to it. To use a single port system would cause the bus to be inaccessible to either the read or the write operation and this time-sharing scheme would cost time and add complexity. If possible a dual-port system must be used. The Virtex devices has built-in dual port BlockRAM perfect for this purpose. As with all dual-port systems, problems will arise if reading AND writing occurs at the same time at the same address. Care must be taken to prevent this from happening.

A diagram of the cache is shown in Figure 11.1. In this diagram Port A is the system side bus and Port B is the SDRAM side. To circumvent the simultaneous access problem, the cache is divided into 2 mutually exclusive address blocks (Bank A and Bank B), neither
of which may be read from nor written to at the same time. Control logic will ensure 
Port A can only access Bank A (or Bank B) while Port B can only access Bank B (or 
Bank A), or vice versa. Once either bank has been completely filled (from either side), 
the addresses are switched around.

### 11.2 VHDL Implementation

#### 11.2.1 Port Structure

**INPUT Signal Discussion**

- **CLK** : Global clock signal
- **RESET.N** : Global reset
- **PAR.FULL** : Signal from Parallel Unit to indicate a word is ready to be read
- **SER.EMPTY** : New word request from Serialising Unit
Chapter 11 — The Cache Controller

Figure 11.3: BlockRAM input/output

NEXT_WORD: SDRAM data transfer signal, from Command Controller
DIRECTION: Cache Port A data direction (0 = into cache)
RESET_CACHE: Resets addresses in Cache Controller, disable writes, set status
to empty, set active write bank to A and read to B
STARTUP: Indicates start of new read operation

OUTPUT Signal Discussion

BANK_FULL: A cache bank is full and ready to be read
BANK_EMPTY: A cache bank is empty and ready to be filled
CACHE_ADDRESS_A/B: Current cache addresses for Port A and Port B
CACHE_ENABLE_A/B: Unused enable lines (cache currently always enabled)
WRITE_SER_N: Inverted write signal to Serialising Unit (Port A)
CACHE_WRITE_ENABLE_A/B: Enables a cache write on the next clock edge
TEMP_CACHE_EMPTY_OUT: Temporary output signal for cache_empty_out

11.2.2 BlockRAM Organisation and Operation

The basic memory component used is the Xilinx SelectRAM+, bit addressable memory
on the Xilinx Virtex©. This is organised into Dual Port BlockRAM by the Xilinx Core-
Generator. The CoreGenerator creates customisable components and is shipped with the
purchase of the Xilinx Design Environment, XILINX ISE. One SelectRAM+ unit in the
Virtex series consists of 4096 bits from which a data-bus can be fashioned with a width
of 1 to 16 bits. The corresponding depth (which would enable all words to be accessed)
is reflected in the address bus width.

The Dual Port BlockRAM Core combines SelectRAM+ blocks up to a maximum data
width of 128 bits and a depth depending on the amount of SelectRAM+ blocks available
on the device intended for use. This can all be specified in the CoreGenerator. The
CoreGenerator outputs the component net list as well as the component wrapper files in
both VHDL and Verilog. The component wrapper can be used as a normal component.
The BlockRAM chosen for this design was a 128-bit wide data bus 16-word deep. The data bus-width is restricted to a power of 2, so 72-bit\(^1\) could not be specified. A larger cache size, up to 128\((w) \times 256(d)\), is possible, but will complicate storage of data that is not true multiples of the cache size.

The inputs to the BlockRAM are registered on the clock edge, with a short setup time prior to the arrival of the clock edge. Additional logic is introduced for decoding status signals from the SU and Parallel Unit for flow control. Care should be taken that this logic does not introduce a large enough combinational delay so as to reduce the clock frequency unacceptably. If so, some registering of the signals could be considered.

The BlockRAM component is shown in Figure 11.3. Since both ports are identical, Port A will be discussed briefly. Signal clka is the clock, in this case the system clock. Wea is write enable. Dina is the input data, addra the address and douta the Port A data output.

### 11.2.3 Addressing in the cache

Since only a single 16-word dual-port BlockRAM is used, Port A and Port B each has one address. In the Cache Controller, signals current_bank.a and current_bank.b will indicate the selected banks for the two ports. Also, each port has a separate 3-bit bank address (cache_address.a_temp, cache_address.b_temp). The current bank combined with the 'cache address' forms the cache_address.a and cache_address.b addresses finally input to the BlockRAM as addra and addrb.

In Figure 11.1 current_bank.a can be seen as the MSB of the addresses \((0000)_2 - (1111)_2\) with cache_address.a_temp as the lower 3 bits. The same goes for the Port B signals.

The Cache Controller and BlockRAM, with their interconnections, are shown as part of Figure 14.1, as CACHE-CTRL and DP-RAM.

### 11.2.4 Dataflow

The Cache Controller manages the data flow between the cache and the paralleling- and serialising units. Proper flow control is needed to ensure data is read and written at the correct times.

\(^1\)Data + check words = 72-bit
Parallel Unit

The Parallel Unit signal full is connected to the par.full signal. When a 0-to-1 transition is seen, data is available on the parallel unit’s data.buffer and is fed through the EDAC, to the mem.data.out and checkword.out outputs. The Cache Controller simply needs to instruct the cache to write to Port A and the data will be written on the next active clock edge.

Serialising Unit

Data can only start to be written to the Serialising Unit once a bank has been filled completely. Thereafter the bank will be emptied into the Serialising Unit in response to the empty signal’s 0-to-1 transitions, one word at a time. Different to the Parallel Unit, the Serialising Unit needs a write signal, because data might not be immediately available. In this case the Cache Controller output write_ser.n has this function. Again, the data is fed through the EDAC in the reverse direction.

Typical data flow scenarios

Standard events in writing from the Parallel Unit throught to the SDRAM:

**CACHE WRITE** (to MMU)

1. The Cache Controller idles.
2. If the Parallel Unit indicates it is full, go to 3, else go to 1.
3. The Cache Controller takes note, and writes the available data to the current cache Port A address.
4. If the bank is not full, return to 1. If the current (write) bank is full, banks are switched and the Command Controller is notified via the bank.full signal.
5. If the SDRAM is idle the Command Controller starts a burst write and the full bank is transferred. The Cache Controller returns to 1 concurrently and is ready to write data to the new bank.

**CACHE READ** (from MMU)

2. If SDRAM is idle and a cache bank is empty, Command Controller dumps SDRAM data to cache.
3. If Serialising Unit indicates empty, go to 4, else go to 1.

4. Write data to Serialising Unit. If bank is empty, switch banks. Go to 1.

Figures 11.5 and 11.4 shows how the bank switching operates.

It is assumed that the SDRAM read/write operations on Port B fill/empty the banks much faster than from the Port A side. Therefore Port B will almost always wait for Port A to finish before a high speed transfer is made or banks can be switched. Mostly Port A is used to determine when to switch banks. Only when Port B fills a bank AND the other bank is already empty, will Port B cause a bank switch. This will most likely only happen at the start of a read operation, when both banks start up empty.

### 11.2.5 Cache Controller state machine

The Cache Controller uses a simple state machine to implement the timing of the addressing and writing, as seen in Figure 11.6. The implementation is done with flags functioning as wait states.

The states named here as IDLE, READ 1, WRITE 1 etc. are combinations of two registers \texttt{waita1, waita2} and the \texttt{direction} signal. While \texttt{direction} is not an internal state holding register, it does not change during normal transfer operation and is considered
Figure 11.5: A typical SDRAM read operation with bank switching.
11.2.6 Internal Operation

The initial state of the controller is **IDLE**, where all *wait* flags are low.

The cache addresses are constantly output to the cache and only changes when the **READ 2** or **WRITE 2** states has been entered. The *cache_enable* signals are only asserted when reading or writing is specifically required from or to the cache, otherwise they serve to disable the cache data ports. In addition, the *cache_write_enable* signals are asserted at the same time, when a write operation is needed. The actual read and write operations occur during the states where *waital* is high, i.e. **READ 1** and **WRITE 1**. Either the Serialising Unit signalling *empty* or the Parallel Unit signalling *full* moves the machine to either **READ 1** or **WRITE 1**.

From states **READ 1** and **WRITE 1** the machine moves without intervention to the next states, **READ 2** or **WRITE 2** and the data ports are disabled. These states are necessary for switching cache banks if needed and updating the cache addresses.

Finally, the state machine automatically reverts back to the **IDLE** state by clearing the *wait* signals.

The status signals *bank_empty*, *bank_full* and *cache_empty* are derived from internal flags showing, for each bank, *full* or *empty*.
11.2.7 Functional Simulation

The Cache Controller was simulated on its own with a testbench providing the required inputs.

To fully test whether the Cache Controller will operate correctly in all instances, different foreseeable scenarios need to be simulated. These include testing whether:

i. the controller switches cache banks correctly after a bank has been filled/emptied,

ii. the controller generates control signals correctly when (a) all banks are empty, (b) one bank is full and (c) both banks are full.

Startup

The Cache Controller should start up correctly after a global or system reset. Addresses should be reset and control signals should indicate that the cache is empty. The controller should be ready to react on read and write signals from the connected modules.

From Figure 11.7, we can see that the above requirements are met at the start of the simulation. Addresses cache_address_a and cache_address_b both point to the start of their respective banks. Signals bank_full and bank_empty both indicate the correct status. The temporary signal temp_cache_empty_out also indicates that the cache is completely empty.

Port A write sequences

The correct cycling of bank addresses and output of status signals will be tested for different scenarios, writing into the MMU.

Cases tested:

1. From idle, fill a bank and empty the bank, with the following variations:
   (a) wait until bank is empty, then start filling the cache again
   (b) start emptying the bank, but start filling the 2nd bank simultaneously

2. From idle, do 3 full write/read cycles of case 1a and 1b, to show the system changes banks properly.

Case 1:
Figure 11.7 shows a simulation for case 1b. After startup a single bank (Bank A) is
CHAPTER 11 — THE CACHE CONTROLLER

filled from the Parallel Unit. The Port A address advances through Bank A correctly and switches to Bank B when Bank A is full, seen as the \((7)_{16}\) to \((8)_{16}\) transition. Control signal \(\text{bank.full}\) becomes active after the last word has been loaded into Bank A, seen at A. The Cache Controller then reacts to this by initiating a burst write to the SDRAM, \(\text{next.word}='1'\). The Port B address can be seen to change rapidly at B. While the data is burst to the SDRAM, a new word is written to Port B. This causes both banks to contain some data, as indicated by the \(\text{bank.empty}\) going low at C. It only stays low until the data is read out completely from Bank A, which is then empty.

An extract showing the detail of the Port B address changes is shown in Figure 11.8, showing the correct address sequence.

Case 1a was tested and found to be handled correctly. In this case \(\text{bank.empty} = '1'\) throughout as Bank A is emptied before Bank B is written to.

Case 2:

The full 3 cycle run is shown in Figure 11.9. The first burst to SDRAM (shown as A), is similar to case 1b, whereas the burst at B is similar to case 1a. Bank addressing for

Figure 11.7: Functional testing of the Cache Controller : single SDRAM write.
Chapter 11 — The Cache Controller

Figure 11.8: Functional testing of the Cache Controller: single SDRAM write detail.

Port A functions correctly as can be seen from the figure. Bank addressing for Port B was checked and also functions correctly. As expected, the temp_cache_empty_out signal stays low for event A, but does get asserted at C and D, because the cache is completely empty for short periods before the next word arrives from the Parallel Unit.

Port B read sequences

The correct cycling of bank addresses and output of status signals will be tested for different scenarios, reading from the MMU.

In this case, data is read from the SDRAM into the cache. Again, different scenarios may present themselves:

Cases tested:

1. From idle, fill a bank and start filling a second bank. Possible variations:
   
   (a) Start filling the second bank, while starting to write data to Serialising Unit.

   (b) Fill both banks completely, then start writing data to Serialising Unit.

2. From idle, fill and empty banks continuously in a typical order.

Case 1a:

In Figure 11.10, case 1a is simulated.

A reset operation is shown at E.
Figure 11.9: Functional testing of the Cache Controller: cycled SDRAM write.

CLOCKING + RESET
/testbench/clk
/testbench/reset_n
INPUT
/testbench/par_full
/testbench/bank_full
/testbench/next_word
/testbench/bank_empty
OUTPUT
/testbench/cache_address_a
/testbench/cache_enable_a
/testbench/cache_write_enable_a
/testbench/cache_address_b
/testbench/cache_enable_b
/testbench/RESET
INTERNAL SIGNALS
/testbench/TEMP_cache_empty_out
/input
/testbench/cache_address_b
/testbench/cache_enable_b
/testbench/RESET

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
Figure 11.10: Functional testing of the Cache Controller: SDRAM read.
Since `ser_empty` indicates that the Serialising Unit is empty, a SDRAM read is initiated at A, filling Bank B. The addressing for Port B is correct and can be seen to change rapidly at D. Figure 11.11 shows an extract of the Port B addressing.

After the burst is complete, the `bank_full` signal is high. Hereafter, the words are periodically written to the Serialising Unit, using pulses such as shown at F on `write_ser_n`. Port B's address changes accordingly. Since Bank A is still empty, a second SDRAM burst is initiated at B. Signals `bank_full` and `bank_empty` behave correctly. No banks are empty once transfer has started at B, and at least one bank is full after transfer at B has completed.

Finally, over time, the full Bank B has been written to the Serialising Unit and is empty again, shown at C. One more write pulse also causes `bank_full` to become low a little later, since a word was read from the remaining full bank, Bank A. The writing continues until the cache is completely empty at G.

It should be noted that the current implementation causes the addresses to become invalid if the cache should be read until completely empty. This is fine if there is no more data to be read from the SDRAM (the end address has been reached). If the end address of the read operation has not been reached, there will always be data in the cache, fed from the SDRAM, so this situation is not expected to occur.

**Case 1b:**

At the start of the simulation shown in Figure 11.12, at A, two bursts from the SDRAM...
is completed, filling both Banks A and B, before the writing to the Serialising Unit is commenced. The Cache Controller control signals operates correctly as well as the addressing of Port A and Port B. The fact that the addressing of Port B starts at \((8)_{16}\) is arbitrary and does not affect the correct functioning of the controller.

*Case 2:*

To test whether the controller functions properly for continuous operation, a typical sequence of 4 bursts from the SDRAM was simulated, shown in Figure 11.12.

The first 2 bursts (at A and B) operate in the same way as case 1b. Operation continues normally with write pulses to the Serialising Unit. When a Bank B becomes empty, it is re-filled by a burst from the SDRAM, at C. The \textit{bank\_empty} signal becomes low again. Again writing continues, until Bank A is empty and is again re-filled, at D.

No further bursts are done from the SDRAM and Bank B becomes empty at E, with no full banks indicated one write further at F. The Bank A finally becomes empty at G, indicated by the \textit{temp\_cache\_empty\_out} signal being asserted.
Figure 11.12: Functional testing of the Cache Controller - cycled SDRAM reading.
Chapter 12

The SDRAM Controller

12.1 SDRAM Controllers

In his article on SDRAM controllers, Christian Green [23] outlines several important considerations when designing a controller: (i) The number of tasks serviced, (ii) speed and (iii) complexity. Complex controllers are often designed to be more efficient in terms of overhead or can keep track of multiple tasks accessing the memory concurrently. The controller for this design need only manage a single task of either reading or writing at a given moment. It was decided that as a first iteration, the SDRAM controller for this design should be as simple as possible. This would exclude support for multiple tasks, while keeping the operation as simple as possible. Using this as a guideline, two controller designs described by Green can be considered: an autoprecharge controller and a single comparator.

Of the two, the autoprecharge controller is the simplest. Sequential read and write operations are completely independent of each other. Burst operations are always followed by a precharge command, closing the accessed bank. Banks are therefore always assumed to be in the precharged state. This enables the controller to be extremely simple in design, because very few choices exist for each command performed. Unfortunately, it also necessitates the use of an activate command before all accesses. This action uses additional clock-cycles, but if the read/write burst-length is long enough, the overhead can be made less significant.

The single comparator design strives to remove the overhead by keeping the last accessed bank in the active state. If subsequent accesses reference the same bank, no activation of precharge commands are necessary. Only if a different bank is accessed must these commands be executed. This will increase transfer rates in some environments where subsequent accesses are closely grouped. Providing this capability implies storing the
number of the last accessed band and comparing new accesses with it, adding logic and possibly decreasing maximum clock speed.

The autoprecharge controller seems a likely candidate. Green stipulates some conditions that should exist to curb bandwidth loss, but as the transfer rate bottleneck of our system is not the memory controller, it might not be critical. We can calculate the number of clock cycles required for a full burst transfer\(^1\) of length 8 (using Figure 12.1):

<table>
<thead>
<tr>
<th>Action</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activate</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CAS Latency</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>Data</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Precharge(^2)</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>14</strong></td>
<td><strong>12</strong></td>
</tr>
</tbody>
</table>

In SDRAM controllers, efficiency depends on the burst length: the longer the burst, the more efficient the transfer. The above figures give us an inefficiency of \(\frac{14-8}{14} = 0.43\) for a

\(^1\)CAS latency of 2

\(^2\)During burst transfers, different devices start their autoprecharge commands at different times. From this moment, a fixed amount of time must pass before a new activate command may be issued, which also varies between devices. For burst lengths of 8 or longer with clock period \(\geq 20\) ns, the \(t_{RP}\) value does not impact and a delay of two cycles is safe. The delays chosen here are sufficient for most devices for system clocks of periods of 20 ns or longer.
read operation. If the system runs at 25MHz, transfer will take place at a maximum of 
\[25 \times \frac{5}{14} = 14.29\text{ MWord/s}\]. For a complete write operation the inefficiency is \(\frac{12-8}{12} = 0.33\).

The autoprecharge controller was chosen for this design.

### 12.2 VHDL Implementation

The controller input/output signals are shown in Figure 12.2.

#### 12.2.1 Port Structure

**INPUT Signal Discussion**

- **CLK**: System clock
- **RESET.N**: System reset
- **REF.REQ**: Refresh request
- **MODE.R**: Mode register set
- **START**: Indicates start of operation
- **R.W**: Data direction (0 = into cache)
- **ABORT**: Abort current operation

The system interface provides refresh requests (ref_req, also ref_ack_out), mode register setting (mode_r), read/write control (r_w) and an abort function (abort).

**OUTPUT Signal Discussion**

- **ADDR.SHIFT.OUT**: Row/column shifting for SDRAM address
- **ADDR.LATCH.OUT**: Latch new address
- **REF.ACK.OUT**: Acknowledge refresh request
CHAPTER 12 — THE SDRAM CONTROLLER

CMD.OUT : Actual command output to SDRAM
DATA VALID.OUT : *not connected*
DATA DIR. OUT : *not connected*

The commands are presented to the SDRAM via registered outputs on the FPGA as signal cmd.out. These lines are sampled on the positive clock edge and if a valid command is present, the SDRAM will start executing accordingly. Signal cmd.out is a combination of the SDRAM inputs cs, ras, cas, we and A10.

The controller can control the directional data buffers connecting the SDRAM with the system (data_valid.out, data_dir.out) and the data/address multiplexers (addr_shift.out, addr_latch.out).

12.2.2 Controller construction

Basic operation

The timing between the controller and the SDRAM internal state machine must be in step to ensure the controller does not act too fast or too late. It is therefore natural that the controller itself should also be built around a state machine that is related to that of the SDRAM, shown in Figure 12.3. The controller by Green was designed for a burst length of 4 and was modified to produce a burst length of 8. For the additional 4 wait cycles needed, 2 new states and an existing wait flag was used.

The state-machine basically has 2 parts: a combinational process generating control signals and next state signals, based on current registered value; and a process that registers the state changes and control signals on the next clock cycle.

The states have the following general functions:

IDLE : Command input wait state.
ACT : State issuing the activate command.
R/W COMMAND : State issuing the read or write commands.
D1,D2,D3,D4 : Wait states for burst length.
PRECHARGE : State handling explicit precharge all command.

All states are listed in Table B.5 in Appendix B.
SDRAM Commands

The specific commands used to implement the autoprecharge controller is the read-with-precharge (RWP) or write-with-precharge (WWP) commands, presented to the SDRAM via registered outputs on the FPGA. Their values can be seen in Tables B.3 and B.4. The other commands used in the controller are also listed.

The state sequence for these commands can be seen in Figure 12.4 B and C. Both RWP and WWP's state sequences include ACT. The actual read/write command is issued in R/W Command. They also have similar wait state requirements in states D1-D4.

Shown in Figure 12.4 are the state sequences for different operations.

Data Transfers

In this controller the start signal indicates the start of a new operation to be performed. Signals r_w and mode_r specify the type of operation to be performed and is sampled at the same instant as start, resulting in either RWP or WWP.
Implementing Refresh

Since the SDRAM Controller does not have an inbuilt timing mechanism for scheduling refreshes, a separate VHDL module was assigned this task: the Refresh Timer, discussed in Section 12.2.3. Ref_req signals a refresh request from the timer. The controller then issues the auto refresh command (02)_{16} as soon as possible. The simplicity of the controller requires ongoing burst transfers to be completed first before the refresh is started. Once the refresh has been initiated, the controller should respond with an acknowledge (ref_ack_out) so that the Refresh Timer can de-assert ref_req.
12.2.3 Refresh Timer Design

Timing

The Refresh Timer is responsible for ensuring the SDRAM is refreshed periodically at a minimum frequency.

The worst-case scenario for refresh timing would be a refresh request just when a read transfer is started. Firstly, because the entire transfer is still to take place and secondly because it is longer than a burst write sequence. This would mean the refresh needs to be delayed until the transfer is done, which takes 14 clock cycles. Immediately thereafter the refresh should be performed. If refreshes are scheduled with a small margin of error, this situation can cause delayed refresh, with data loss as result.

The controller is not our throughput bottleneck and transfer would not be impacted negatively by a few extra refreshes, so the simplest solution to prevent missed refreshes would be to make them a little more frequent.

A typical SDRAM cell needs to be refreshed every 64ms. If the entire SDRAM module consists of 4096 rows[7], each row being fully refreshed at a time, we can calculate the interval for distributed refreshes:

\[
\frac{64 \times 10^{-3}}{4096} = 15.6 \mu s.
\]  

(12.1)

For a 25MHz clock, this means a refresh every 15.6 μs/40 ns = 390 clock cycles. The read burst plus the following refresh takes a maximum of 15 cycles³, giving a refresh interval of 390 - 15 = 375 cycles. To be safe: 375 × 0.9 ≈ 337. We can use 256 + 64 = 320. The test used in the timer is therefore: 'if count ≥ 320 then request refresh'. If the system clock frequency is different, this value will have to be adjusted.

A lower limit frequency for the system clock can also be calculated. If at least one burst transfer should fit in between mandatory refreshes, then the maximum clock cycles between refreshes are 15 + 1 cycles⁴ = 15.6 μs. This gives a lower limit of around 1MHz for our system. Anything above 2MHz should be acceptable.

³2 cycles for activate + 11 cycles for burst and wait period + 2 cycles for refresh = 15 cycles

⁴For a \( t_{RC} = 20\text{ns} \leq 1 \text{ clock cycle} \)
CHAPTER 12 — THE SDRAM CONTROLLER

Figure 12.5: Refresh timer input/output

Timer Implementation

The timer has a counter that counts on each clock edge, up to a maximum value as determined in the previous section. When this value is reached, the ref_req signal is asserted and the counter is reset to zero. A refresh acknowledge signal, ref_ack, will cause the ref_req to be reset to 0.

The usual clk and reset_n system signals are present. SDRAM_active=0 and SDRAM_off=1 both will stop and reset the counter. The counter can also be forced to reset during normal operation through the ref_reset signal.

Timer Simulation

A simulation of the Refresh Timer is shown in Figure 12.6. The ref_req signal goes high as soon as the maximum value for the counter is reached (at A): (140)_{16} = (320)_{10}. Important to note is that the counter resets immediately and not when the refresh is acknowledged (at B), to keep the refresh counter accurate.
12.3 SDRAM Controller Functional Simulation

The SDRAM Controller was simulated on its own with a testbench providing the required inputs.

Testing was divided into four parts: Startup, Write, Read and Refresh. SDRAM command codes are listed in Table B.4 for reference.

The SDRAM Controller was simulated separate from the other modules in the VHDL design, therefore the input signals seen in the wave-diagrams were generated using a VHDL testbench. In the simulation figures, all multi-bit signal values are presented in hexadecimal radix.

Inputs to registered SDRAM DIMMs need to satisfy setup and hold times relative to the rising clock edge. Outside of this the signals may change at will. Input setup (2ns) and hold (1ns) requirements are similar for several registered DIMM modules [8, 7]. These values need to be checked if a post place-and-route simulation is done.

12.3.1 SDRAM Startup

Initial startup of the SDRAM module is discussed in Chapter 2 and 7.

A typical startup sequence, as would be initiated by the Command Controller, was selected here to test the SDRAM Controller: (1) precharge all, (2) two CBR (auto) refreshes and (3) a mode register set operation\(^5\).

The functional simulation is shown in Figure 12.7. Initially, the \textit{cmd\_out} output is \text{(1E)}\(_{16}\) at A, indicating \text{NOP}. At B the \textbf{precharge} command is issued, the state machine goes to state \text{(08)}\(_{16}\) and the correct command is output, \text{(05)}\(_{16}\). Signal \text{prech\_count\_reg} can be seen counting the required wait time. The state returns to idle, \text{(00)}\(_{16}\), hereafter. At C, the first refresh request is made, resulting in the acknowledgement at D, along with the correct state and command outputs. This is repeated at E. At F the \textbf{mode register set} command is issued. This command requires the \textit{start} signal to be active. The resulting command \text{(00)}\(_{16}\) is seen on \textit{cmd\_out}.

\(^5\)The address output for setting the mode register is simulated in Chapter 7, since the SDRAM Controller does not generate the address directly.
12.3.2 Burst Write

Simulation results are shown in Figure 12.8. At A, signal *start* initiates a burst write operation, since $r_w$ is low (indicating read). At B, the *activate* (06)$_{16}$ command can be seen on *cmd_out*, at which time the row address should also be provided by the addressing unit (not shown). There is a 2 clock-cycle latency before the *write* command (09)$_{16}$ is output, seen at C. Asserting *addr_shift_out* will enable the Address Unit to output the starting column address at this point, which completes the absolute address of the *write*. The column address is sampled on the rising edge of the clock and need not maintain its value apart from the setup and hold times before and after the clock edge.

From the figure the *data_valid_out* signal confirms data is present for 8 clock cycles, corresponding to the burst length of 8 the controller is designed for. The first word gets read at the same time *write* is issued. We can see the state machine spends two cycles each in the non-zero states, due to the *wait* internal signal (here *wait_reg*).

At D, the controller returns to the *idle* state after the burst is complete.

12.3.3 Burst Read

Shown in Figure 12.9 is the burst read operation of the controller. At A, *start* and $r_w$ are both set high. As with the burst write, the *activate* command is first output (seen
at B), followed by the read command, $(0B)_{16}$, 2 clock cycles later at C. Due to the CAS latency of the proposed SDRAM used, valid data is expected to be output 2 cycles after read is registered by the SDRAM. The data_valid_out signal reflects this property of the controller, being high 2 cycles later and for 8 rising clock edges. Again, the controller finishes in idle at D.

### 12.3.4 Refresh

Auto-refresh when the controller is idle is shown in Figure 12.7.

The Command Controller only monitors that a refresh is not requested at the exact same time as a start=1 event and delays it to after the burst start sequence. The SDRAM Controller should delay the refresh command until feasible.

In Figure 12.10 a burst read transfer similar to Figure 12.9 is started at A, during which ref_req is asserted, at B. The burst transfer is completed before the refresh is attended to at C.
CHAPTER 12 — THE SDRAM CONTROLLER

CLOCKING + RESET
/testbench/clk
/testbench/reset_n

INPUT
/testbench/start
/testbench/r_w

OUTPUT
/testbench/addr_shift_out
/testbench/data_valid_out
/testbench/cmd_out

INTERNAL SIGNALS

Figure 12.9: SDRAM Burst Mode: Read

CLOCKING + RESET
/testbench/clk
/testbench/reset_n

INPUT
/testbench/start
/testbench/r_w
/testbench/ref_req

OUTPUT
/testbench/ref_ack_out
/testbench/data_valid_out
/testbench/cmd_out

INTERNAL SIGNALS

Figure 12.10: SDRAM read with refresh during burst
Chapter 13

The Address Unit

13.1 Overview

This chapter discusses the operation of the Address Unit. The Address Unit has three tasks:

- Storing the start and end addresses of a transfer;
- providing the SDRAM with correct column and row addresses;
- signaling the end of a transfer.

The start and end addresses are set up from outside using the command-data interface of the Mass Memory Unit (MMU). The correct row and column addresses are provided to the SDRAM by the Address Unit, with the help of the SDRAM Controller. Signals are exchanged between the two modules to synchronise them. When the end of a transfer is reached, it is the responsibility of the Address Unit to signal the Command Controller to reset the system.

13.2 VHDL Implementation

13.2.1 Port Structure

The VHDL port structure of the Address Unit module follows:

**Input signal description:**

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK</td>
<td>Global clock signal</td>
</tr>
</tbody>
</table>
CHAPTER 13 — THE ADDRESS UNIT

Figure 13.1: Address Unit input and output.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESET_N</td>
<td>Global reset</td>
</tr>
<tr>
<td>COUNT</td>
<td>Address advance signal</td>
</tr>
<tr>
<td>RESET_ADDRESS</td>
<td>Reset internal working address to 0</td>
</tr>
<tr>
<td>SHIFT</td>
<td>Shift between row and column address output</td>
</tr>
<tr>
<td>SET</td>
<td>Indicate load of new addresses</td>
</tr>
<tr>
<td>ADR BYTE</td>
<td>Next byte of address to load</td>
</tr>
<tr>
<td>ENDADR OUT</td>
<td>Debug signal, partial end address (not in Fig. 14.1)</td>
</tr>
<tr>
<td>WORKADR OUT</td>
<td>Debug signal, partial work address (not in Fig. 14.1)</td>
</tr>
</tbody>
</table>

Output signal description:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR</td>
<td>Row/Column address output</td>
</tr>
<tr>
<td>ADR DONE</td>
<td>End address has been reached</td>
</tr>
<tr>
<td>LOAD DONE</td>
<td>Address load is complete</td>
</tr>
</tbody>
</table>

13.2.2 Operation

Address synchronisation and output

The current SDRAM address is stored in the Address Unit using an internal counter. It needs to be incremented each time a word is read from or written to the SDRAM. To synchronise the Address Unit with the SDRAM Controller, the SDRAM Controller outputs a signal, count, which indicates with a low to high transition when a word is transferred.

The SDRAM Controller operates on a burst length of 8. Only the address of the first word is output to the SDRAM module, which has an internal counter for the rest of the burst. The Address Unit only outputs every 8th word, but increments the internal address on low to high transitions of count.

SDRAM accepts addresses as row and column parts at different times during the issue of a command. The SDRAM Controller uses the shift signal to indicate which part is
needed. When shift is low, the lower part of the address (row) is output, when high, the higher part of the address (column). Along with the row part of the address, additional control signals are also output to the SDRAM through the address lines. The row and column widths and alternative functions of the address lines are discussed in Section 2.3.

External Address Load

This section describes how a client would set up the start and end addresses of the MMU and how it is implemented in the Address Unit.

For every data transfer to or from the MMU, a start and end address is specified. A data transfer uses the start and end addresses as they are found in their respective registers, regardless of when they were loaded. An address setup sequence also does not require a data transfer afterwards. They are completely independent.

A typical address setup operation is now described assuming the use of Port A. First, the Set Address A command is issued by the user. The addresses are then input at Data Port A as a series of six bytes: 3 for the start address and 3 for the end address. Each byte is committed on the rising edge of signal WR_PA. The least significant byte of the start address is input first. A typical example of the operation is shown in Figure 13.2. The signal WR_PA is not synchronised with the system clock and the address setup is therefore tolerant of frequency variations in WR_PA, demonstrated in Figure 13.3.

It is the responsibility of the Address Unit to signal when the address load is complete. This is done by asserting load_done, which will be detected by the Command Controller.

The SDRAM module is word addressable, pointing to 64-bit words in the module. If the rest of the user system has a different bus-width, the addresses that are used should be converted first. If the bus width is a power of 2, truncate the lower bits of the addresses. Addresses from a smaller address space would need padding to fill the combined row and column width of the SDRAM module. For example, in 8-bit systems the 3 lower bits should be truncated. The new start and end addresses could imply a different number of bytes to be transferred and it is advisable transfer this new amount of bytes and pad the bytes for which there is no real data.

End of transfer

The current address is continually tested against the end address to determine whether the end of transfer has been reached. When this happens, the adr_done signal is asserted. This will let the Command Controller know that it must return the system to the idle state in preparation for the next command. The start, end and current addresses in the
CHAPTER 13 — THE ADDRESS UNIT

Figure 13.2: Loading of start and end addresses.

Figure 13.3: Loading of start and end addresses (variation).
CHAPTER 13 — THE ADDRESS UNIT

unit are not cleared at this stage.

Detail VHDL Operation

In the Address Unit the current address of the SDRAM accesses is stored in the signal work_address. At the time of loading the first two bytes of the start address is loaded into a temporary register intern_addr. It is transferred to work_address on arrival of the third byte. An additional 1-bit register, set_prev, maintains the previous value of set, which is used to prevent loading from restarting when set is kept high after all address bytes have been transferred.

13.2.3 Functional Testing

The Address Unit was functionally simulated using a test bench to provide the required inputs.

Loading of addresses

In Figure 13.4 a functional simulation is shown for the address loading of the unit. Signal endadr_out is a temporary output for the lower 8 bits of the end address, as is workadr_out.
for the current address. The address \((A3A2A1)_{16}\) is loaded as the start address and \((A3A2B0)_{16}\) is loaded as the end address.

At A the loading is initiated by \textit{set} going high. After six rising clock edges the \textit{load\_done} signal goes high at C.

**Output Address**

The output address, as presented to the SDRAM, is shown as \textit{addr} in Figure 13.4. The \textit{shift} controls the row and column output of the current address. After the first 3 bytes are loaded the starting address is complete and the \textit{addr} output takes on a new value, which represents the lower 12 bits of the current address\(^1\), seen at B. Setting \textit{shift} to ‘1’ will result in the higher 12 bytes being included in the \textit{addr} output, at D.

**Address counting**

Further operation is simulated in Figure 13.5, showing the latter part of a counting operation. The current address will advance when \textit{count} ‘1’. \textit{Count} is asserted at A and the current address is advanced by 1. As soon as \((A3A2B0)_{16}\) is reached (at B) the unit asserts \textit{adr\_done}.

The unit’s internal address and counters can be reset in response to the \textit{adr\_done} signal, using the \textit{reset\_address} signal. This was tested and is also shown in Figure 13.5 at C.

**Full simulation**

The full simulation is shown in Figure 13.6. Loading and counting the address is simulated 15 times. \textit{Workaddr\_out} verifies this.

---

\(^1\)The current address (before any counting) is \((A3A2A1)_{16}\). In binary this is \((10100011\ 10100010\ 10100001)_{2}\) and therefore the expected output when \textit{shift} = ‘0’ is \((2A1)_{16}\), and when \textit{shift} = ‘1’ it is \((A3A)_{16}\).
CHAPTER 13 — THE ADDRESS UNIT

Figure 13.5: Address Unit functional simulation: address end
Chapter 14

Full System Integration

14.1 Overview

In this section the simulation of the entire VHDL system is discussed. A block diagram of the system with all its major components is shown in Figure 14.1. Although the System I/O block is shown separate in the diagram for clarity, it is really part of the top-level VHDL file and serves as the interface between the hardware inputs and the VHDL system.

14.2 Simulation

Functional simulations were done of the full system, using a testbench to supply the required inputs.

In Figure 14.2 a full write operation to the Mass Memory Unit (MMU) is shown, complete with startup sequence, address setting and the actual writing of the data.

14.2.1 Startup

In Figure 14.2, the SDRAM is automatically initialised after system reset, at A.

14.2.2 Address set-up

In Figure 14.2 at B, the start and end addresses for a write operation are set up, using the Set Adr. A command followed by the writing of six bytes through Port A. The addresses set up were: start address = (0000)_{16}; end address = (001F)_{16}, representing 256 bytes of data written to Port A.
14.2.3 Write operation

The write operation is started at C, writing data to Port A. At E the first cache is full, followed by a burst write to the SDRAM. Afterwards the SDRAM address is at (0008)\text{16}. The data initially written to the cache is also output at dq on the figure, along with the check words, cb.

This happens four times, at E, G, H and J. After J, the end SDRAM address has been reached, (001F)\text{16}, and the system returns to idle with \textit{cmd.control.state} = (00)\text{16}.

14.2.4 Refreshing

For simulation purposes the refresh period has been reduced to half the previously calculated values. The system generates its own refresh commands at D, F, K and M. The detail waveforms were checked and the SDRAM commands and timing are correct.

14.2.5 Read operation

The read operation for the system was simulated and the results shown in Figure 14.3.

As with the write simulation, the startup sequence is done at A. At B the addresses were set up: start address = (0000)\text{16}; end address = (001F)\text{16}, representing 256 bytes of data to be read.

At D, a burst read is done from the SDRAM to fill the first cache. Since another cache bank is still empty, another burst is done. The actual reading of the data via Port B starts at C. Again, at E, G, J and K, we have our refreshes.

At F and H, cache banks become empty, and data is burst from the SDRAM. At H, all the data is already read from the SDRAM as can be seen from the address, a, but there is still data in the cache. This is read out piece by piece, the first bank becoming empty at R. The second bank's first word is read just a little later, causing \textit{cache.bank.full} to go low. The last bytes are read from the Serialising Unit at P, causing the system to go into the \textit{idle} state, at M.
Figure 14.3: Full System functional simulation - full read
14.3 Conclusion

From these simulations and of those in previous chapters, it can be seen that the system functions properly in various different scenarios:

- The system starts up correctly, initializes the SDRAM and enters the idle mode.
- The system loads start and end addresses correctly.
- Read and Write commands are interpreted correctly.
- The system prepares correctly in anticipation of write operations and especially for read operations, doing the pre-reading from the SDRAM correctly.
- The address incrementing is correct and the reach of the end address has the desired effect: when writing the system enters the idle state soon after; when reading the two banks of the cache are read out first before the system goes into idle mode.

It should be noted that not all the commands were implemented, as was originally anticipated. Currently Set Adr. A, Write A and Read B is available. Both ports were not implemented fully and thus share the same addressing. This is why Read B works correctly while not explicitly set up.

The full list of commands that would be useful is listed in Table B.1.
Chapter 15

Hardware

In this chapter, guidelines are given for when an actual hardware implementation is made. A component for checking the FPGA's configuration is designed and timing issues with the SDRAM is addressed.

15.1 The Configuration Checker

The summary given in Chapter 5 is rather simplistic when compared to what must be done in reality to detect and correct errors in the configuration of the FPGA. In this section, a VHDL component is described to perform basic checking of the configuration data.

15.1.1 The Configuration Bitstream

At power on the FPGA loads its configuration from a non-volatile PROM. This can be done in series or parallel. This stream of data is known as the configuration bitstream. The bits represent the configuration of one of the following main areas in the FPGA: interconnection network, configurable logic and RAM. The bitstream controls directly how the different subcomponents of these three areas operate. If one of the configuration bits has an incorrect value, the relevant component will not function as expected.

The bitstream read back from the device has exactly the same structure as the one the FPGA is configured with. The same information is provided, but during operation, some RAM bit-values or flip-flop output values will change. An example would be the contents of the BlockRAM in the device, which is also read back in the same bitstream. These bits cannot be tested to be True or False, since they have random values. Only the configuration bits that would not normally change should be examined. There are therefore two
types of bits read from the FPGA, state bits (variable) and static bits. Static bits reflect the logic and interconnect and should remain the same throughout operation. State bits represent values of flip-flops, RAMs and IOBs; which can change during operation. The identical structure of the two streams enable a bit-by-bit comparison, if the static bits can be identified.

**Configuration Mask**

A method is needed to distinguish between static bits and state bits. An algorithm probably exists that would indicate which bits to compare, but it could be awkward to implement. An easier option is to introduce a *third* bitstream that contains a mask that identifies the static bits. A simple logic comparison can then be done to detect a discrepancy in the two configuration bitstreams in real time. The Xilinx development software can produce such a mask bitstream. This mask has to be stored in a second PROM similar to the configuration PROM. Figure 15.1.1 shows the connectivity of the devices.

This scheme is valid when only the configuration of the device is checked for errors. If the memories and status values are to be checked, at least one other FPGA is needed that performs exactly the same in-circuit operations. A fault can be detected in this way. If the location of the fault is to be determined, a third FPGA is needed in conjunction with a majority vote system.

A '0' in the mask stream indicates a configuration bit, a '1' indicates a memory/IO bit. We ignore bit positions with 1's. When the mask bit is '0', the configuration and readback bits for that position must then be identical. If not, an error in either the FPGA, configuration PROM or mask PROM has occurred. The truth table shown in Table 15.1 reflects the
Table 15.1: Truth table for configuration masking

<table>
<thead>
<tr>
<th>Conf</th>
<th>Rdbk</th>
<th>Mask</th>
<th>Err</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 15.1: Truth table for configuration masking

Operation of the testing logic, with columns for Configuration (C), Readback (R), Mask (M) and Error (E). A logic function can be derived from the truth table and is found to be: \( E = \overline{M}(C \oplus R) \). Whenever this function outputs ‘1’, the bitstreams are in conflict. See Figure 15.2.

15.1.2 Reliability of Checking the configuration memory

The reason why the JTAG port goes through all the devices and not just through the PROMs is so that the FPGA can be controlled through the JTAG port as well. Operations like disconnecting all the outputs etc. can be performed, which can be useful in debugging situations.

In Figure 15.4 the bracketed pin names indicate user pins which was selected to perform readback related functions. Non-bracketed pins indicate fixed function (non-user) pins dedicated to the configuration process, which cannot be re-assigned by the user.
15.2 SDRAM Timing

It is important that the timing of the SDRAM module be synchronised with that of the FPGA. For a synchronous system to be useful one must be able to make assumptions about the behaviour of the clocking signal. Differences in delays between clock and data signals can cause serious problems. Inside the FPGA solutions to this problem is implemented in hardware with (amongst others) low clock-skew networks. When the FPGA has to interface to external devices using the same clock, it is important that the clock does not arrive at the external device prior to the data. This will cause corruption of data that has not yet stabilised. To overcome this problem, a Phase-Locked Loop (PLL) is used. Using a PLL, the clock signal to the SDRAM is phase-shifted to lock to the returning clock signal. This ensures that the clock is active before any change in data.

In Figure 15.5, clk is the clock signal from the FPGA to the SDRAM at the output of the FPGA. Signal data_s represents the delayed data available at the SDRAM data inputs. Similarly, clk_s is the delayed clock signal at the SDRAM clock input. Here can be seen that it would be possible for the clock signal to clock the data while it is changing. This is undesirable. The feedback clock signal is also included to illustrate the further delay the clock experiences returning from the SDRAM. Note that the PLL will lock the clk signal to clk_fb. This is important because it will cause a phase shift in clk_s which may

Figure 15.3: JTAG connections for multiple-device programming
Figure 15.4: Configuration and Mask PROM connections with FPGA

Figure 15.5: Delay in clock to SDRAM
be more than which is required to fix the delay problem. This could to some degree limit the operating frequency of the system. In Figure 15.6 the shifted $clk_s$ is shown. Now the data would not be sampled while changing, except if the clock frequency is too high and settling time of data approaches the clock period.

15.2.1 Clocking inside the FPGA

To be able to use the output of the DLL as the internal clock for the FPGA, the output must (1) drive a global clock buffer (BUFG) and (2) be fed back to the $CLKFB$ as shown in Figure 15.7[24].
Part III

Conclusions and Recommendations
Chapter 16

Conclusions and Recommendations

This chapter will draw some conclusions from the implementation described in the thesis. It will also make some recommendations for future work.

16.1 Conclusions

It was shown that it is possible to successfully design a Mass Memory System using SDRAM and an FPGA. It was also shown that it is possible to keep the interface to the system very simple, even though the core storage device is SDRAM, which is a complex device. The modular design methodology, although sometimes cumbersome, proved to work for this design.

Up to a maximum frequency, the system is independent of the frequency at which the 'client' writes or reads. This is in line with one of the original goals to make the MMU function more similar to SRAM. This makes it easier for simple devices to use the MMU as a data store.

Due to the complex nature of the SDRAM timing, it was found that it is difficult to design a generic controller that would work for SDRAMs with different latencies and maximum clock speeds. Some use different pin configurations. These issues could require a redesign of the SDRAM Controller and possibly large parts of the Command Controller, which is undesirable. This would be because the controllers currently are state machine based, with some dedicated states enforcing timing constraints.

The use of the cache along with the burst transfer modes of the SDRAM proved to be successful. In the simulations it can be seen that SDRAM has long periods of inactivity, which should be used for power saving modes.

The use of Port B just for output is a reduction in the intended use: to be bi-directional.
Although the logic was constructed, full simulations were not run. Some advance was made to design the system to be error tolerant. The EDAC and the ConfigurationChecker are examples.

Valuable experience was gained regarding the properties of both SDRAM technology and FPGA technology and designing systems utilising them.

16.2 Recommendations

The following recommendations are made for future work:

- SDRAM technology has advanced quite far beyond the basic SDRAM described and used in the thesis. Further study can be done to test the feasibility of using advanced SDRAM like DDR SDRAM in an implementation such as this.

- Radiation testing could be done on current SDRAMs and non-hardened FPGAs components to test physical endurance and data retention in space-like environments.

- The work did not produce a working evaluation board. A board was designed and built and the basic interfacing functioned, but work on it was abandoned due to time constraints. Once finished, the board could prove to be valuable in testing new designs, since it has an SDRAM DIMM socket and a powerful Virtex© FPGA on board. The bi-directionality of Port B can be completed also, as well as the status Port D.

- The low-power modes of the SDRAM remains to be implemented. These are of great importance in power budgeted systems like satellites.

- If SDRAM was found to be remotely feasible for use in space in spite of environmental conditions, a ‘washing’ system would have to be implemented, where data is read from the SDRAM periodically and written back via the EDAC Unit. This would correct errors that may have resulted from radiation.
Bibliography


Part IV

Appendices
Appendix A

Calculations

A.1 SDRAM throughput calculations

Assuming burst transfer (data out on every clock cycle):

Peak Rate $= \frac{f \times n}{b} = \frac{100 \times 64}{8} = 800 \text{ MB/s.}$ where $f$ is clock frequency (MHz), $n$ is word-width and $b$ is byte-width.

[DDR: Peak Rate $= \frac{f \times n}{b} = \frac{200 \times 64}{8} \times 2 = 3.2 \text{ GB/s.}$]

A.2 Discussion of Example SDRAM DIMM

Technical details of the Samsung M374S1623ETS SDRAM DIMM.[7]

Specifications: 16Mx72 SDRAM DIMM with ECC based on 8Mx8, 4 Banks, 4K Refresh, 3.3V Synchronous DRAMs with SPD.

It is a 128MB device with an extra 8-bit Error Correction Code storage per word. The 18 SDRAM chips used are 8MB modules having 4 banks and needing 4096 refreshes every 15.6$\mu$s. It also has a serial presence detect eeprom to provide operational information to systems capable of reading it.
# Appendix B

## Tables

### Table B.1: System Commands and Status Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Constant</th>
<th>Binary code</th>
<th>Hexadecimal Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>c_idle</td>
<td>0000</td>
<td>0x0</td>
</tr>
<tr>
<td>Set Adr. A</td>
<td>c_set_adr_a</td>
<td>0001</td>
<td>0x1</td>
</tr>
<tr>
<td>Set Adr. B</td>
<td>c_set_adr_b</td>
<td>0010</td>
<td>0x2</td>
</tr>
<tr>
<td>Write B</td>
<td>c_wr_b</td>
<td>0011</td>
<td>0x3</td>
</tr>
<tr>
<td>Read B</td>
<td>c_rd_b</td>
<td>0100</td>
<td>0x4</td>
</tr>
<tr>
<td>Status</td>
<td>c_status</td>
<td>0101</td>
<td>0x5</td>
</tr>
<tr>
<td>Mode Register Set</td>
<td>c_mode_r</td>
<td>0110</td>
<td>0x6</td>
</tr>
<tr>
<td>Write A</td>
<td>c_wr_a</td>
<td>0111</td>
<td>0x7</td>
</tr>
<tr>
<td>SDRAM Off</td>
<td>c_sDRAM_off</td>
<td>1000</td>
<td>0x8</td>
</tr>
<tr>
<td>Shutdown</td>
<td>c_shutdown</td>
<td>1111</td>
<td>0xF</td>
</tr>
<tr>
<td>unused</td>
<td></td>
<td>1001 to 1110</td>
<td>0x9 to 0xE</td>
</tr>
</tbody>
</table>

### Table B.2: State encoding for the Command Controller

<table>
<thead>
<tr>
<th>Name</th>
<th>Constant</th>
<th>Binary Code</th>
<th>Hexadecimal Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>s_idle</td>
<td>0 0000</td>
<td>0x00</td>
</tr>
<tr>
<td>Addr. Load 1</td>
<td>s_adr_a_load_1</td>
<td>0 0001</td>
<td>0x01</td>
</tr>
<tr>
<td>No Action</td>
<td>s_no_action</td>
<td>0 0010</td>
<td>0x02</td>
</tr>
<tr>
<td>Mode Reg. Set</td>
<td>s_mrs_1</td>
<td>0 0011</td>
<td>0x03</td>
</tr>
<tr>
<td>Write 1</td>
<td>s_wr_1</td>
<td>0 0100</td>
<td>0x04</td>
</tr>
<tr>
<td>Write Idle</td>
<td>s_wr_idle</td>
<td>0 0101</td>
<td>0x05</td>
</tr>
<tr>
<td>Write Burst</td>
<td>s_wr_burst_1</td>
<td>0 0110</td>
<td>0x06</td>
</tr>
</tbody>
</table>

*continued on next page...*
Chapter B — Tables

Table B.2: Continued

<table>
<thead>
<tr>
<th>Name</th>
<th>Constant</th>
<th>Binary Code</th>
<th>Hexadecimal Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>s_reset</td>
<td>0 1000</td>
<td>0x08</td>
</tr>
<tr>
<td>Write Burst 2</td>
<td>s_wr_burst_2</td>
<td>0 1001</td>
<td>0x09</td>
</tr>
<tr>
<td>Write Burst 3</td>
<td>s_wr_burst_3</td>
<td>0 1010</td>
<td>0x0A</td>
</tr>
<tr>
<td>Read 1</td>
<td>s_rd_1</td>
<td>0 1011</td>
<td>0x0B</td>
</tr>
<tr>
<td>Read Burst 1</td>
<td>s_rd_burst_1</td>
<td>0 1100</td>
<td>0x0C</td>
</tr>
<tr>
<td>Read Burst 2</td>
<td>s_rd_burst_2</td>
<td>0 1101</td>
<td>0x0D</td>
</tr>
<tr>
<td>Read Burst 3</td>
<td>s_rd_burst_3</td>
<td>0 1110</td>
<td>0x0E</td>
</tr>
<tr>
<td>Advance 1</td>
<td>s_adv_1</td>
<td>0 1111</td>
<td>0x0F</td>
</tr>
<tr>
<td>Advance 2</td>
<td>s_adv_2</td>
<td>1 0000</td>
<td>0x10</td>
</tr>
<tr>
<td>Read Idle</td>
<td>s_rd_idle</td>
<td>1 0001</td>
<td>0x11</td>
</tr>
<tr>
<td>Read Wait</td>
<td>s_rd_wait</td>
<td>1 0010</td>
<td>0x12</td>
</tr>
<tr>
<td>Mode Reg. Set Wait 1</td>
<td>s_mrs_wait_1</td>
<td>1 0011</td>
<td>0x13</td>
</tr>
<tr>
<td>Mode Reg. Set Wait 2</td>
<td>s_mrs_wait_2</td>
<td>1 0100</td>
<td>0x14</td>
</tr>
<tr>
<td>Address 1</td>
<td>s_adr_1</td>
<td>1 0101</td>
<td>0x15</td>
</tr>
<tr>
<td>Advance Idle</td>
<td>s_adv_idle</td>
<td>1 0110</td>
<td>0x16</td>
</tr>
<tr>
<td>Adr. Done Wait</td>
<td>s_adr_done_wait</td>
<td>1 0111</td>
<td>0x17</td>
</tr>
<tr>
<td>Precharge All</td>
<td>s_prech_all</td>
<td>1 1010</td>
<td>0x1A</td>
</tr>
<tr>
<td>Startup Refresh 1</td>
<td>s_refr_1</td>
<td>1 1011</td>
<td>0x1B</td>
</tr>
<tr>
<td>Startup Refresh 2</td>
<td>s_refr_2</td>
<td>1 1100</td>
<td>0x1C</td>
</tr>
<tr>
<td>Refresh Wait</td>
<td>s_refr_wait</td>
<td>1 1101</td>
<td>0x1D</td>
</tr>
</tbody>
</table>

Table B.3: SDRAM Command Word: Bit structure

<table>
<thead>
<tr>
<th>Bit</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>CS</td>
<td>RAS</td>
<td>CAS</td>
<td>WE</td>
<td>A10</td>
</tr>
</tbody>
</table>

Table B.4: Table of SDRAM Commands Used

<table>
<thead>
<tr>
<th>Command</th>
<th>Binary Code</th>
<th>Hexadecimal Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode register write</td>
<td>0 0000</td>
<td>0x00</td>
</tr>
<tr>
<td>Auto Refresh</td>
<td>0 0010</td>
<td>0x02</td>
</tr>
<tr>
<td>Precharge All Banks</td>
<td>0 0101</td>
<td>0x03</td>
</tr>
<tr>
<td>Activate (row/bank)</td>
<td>0 0110</td>
<td>0x06</td>
</tr>
<tr>
<td>Write w. autoprecharge1</td>
<td>0 1001</td>
<td>0x09</td>
</tr>
</tbody>
</table>

continued on next page...
Table B.4: ...continued

<table>
<thead>
<tr>
<th>Command</th>
<th>Binary Code</th>
<th>Hexadecimal Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read w. autoprecharge</td>
<td>0 1011</td>
<td>0x0B</td>
</tr>
<tr>
<td>Burst Terminate(^2)</td>
<td>0 1100</td>
<td>0x0C</td>
</tr>
<tr>
<td>No operation (NOP)</td>
<td>1 1110</td>
<td>0x1E</td>
</tr>
</tbody>
</table>

Table B.5: Table of SDRAM Controller States

<table>
<thead>
<tr>
<th>State Name</th>
<th>Binary Code</th>
<th>Hexadecimal Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>0000</td>
<td>0x0</td>
</tr>
<tr>
<td>Activate</td>
<td>0001</td>
<td>0x1</td>
</tr>
<tr>
<td>D2</td>
<td>0010</td>
<td>0x2</td>
</tr>
<tr>
<td>D1</td>
<td>0011</td>
<td>0x3</td>
</tr>
<tr>
<td>Command</td>
<td>0101</td>
<td>0x5</td>
</tr>
<tr>
<td>D3</td>
<td>0110</td>
<td>0x6</td>
</tr>
<tr>
<td>D4</td>
<td>0111</td>
<td>0x7</td>
</tr>
<tr>
<td>Precharge</td>
<td>1000</td>
<td>0x8</td>
</tr>
<tr>
<td>unused</td>
<td>0100</td>
<td>0x4</td>
</tr>
<tr>
<td></td>
<td>1001 - 1111</td>
<td>0x9 - 0xF</td>
</tr>
</tbody>
</table>

Table B.6: Mode Register Definition

<table>
<thead>
<tr>
<th>Address Bits</th>
<th>Function</th>
<th>Options</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2-A0</td>
<td>Burst Length</td>
<td>1</td>
<td>000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Full Page(A3=0)/Reserved</td>
<td>1xx</td>
</tr>
<tr>
<td>A3</td>
<td>Burst Type</td>
<td>Sequential</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Interleaved</td>
<td>1</td>
</tr>
</tbody>
</table>

\(^1\) Auto precharge is controlled by the A10 bit. Enabled with A10='1'.

\(^2\) Illegal for read/write with autoprecharge.
Table B.6: ...continued

<table>
<thead>
<tr>
<th>Address Bits</th>
<th>Function</th>
<th>Options</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A6-A4</td>
<td>CAS Latency</td>
<td>Reserved</td>
<td>0xx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reserved</td>
<td>1xx</td>
</tr>
<tr>
<td>A8-A7</td>
<td>Operating Mode</td>
<td>Standard</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reserved</td>
<td>01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reserved</td>
<td>1x</td>
</tr>
<tr>
<td>A9</td>
<td>Write Burst Mode</td>
<td>Programmed</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Single Location Access</td>
<td>1</td>
</tr>
</tbody>
</table>
Appendix C

Code Listings

C.1 Full System

C.1.1 Full System - VHDL Code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity main_test3 is
  Port (CB : inout std_logic_vector(7 downto 0);
    A : out std_logic_vector(13 downto 0);
    Sn : out std_logic_vector(3 downto 0); -- chip select lines
    ODREN : out std_logic_vector(1 downto 0);
    BAn : out std_logic_vector(1 downto 0);
    RASn : out std_logic;
    CASn : out std_logic;
    WEn : out std_logic;
    DQ : inout std_logic_vector(63 downto 0);
    Q_CLK : in std_logic;
    DLL_CLK : in std_logic;
    FPGA-ON : out std_logic;
    DLL_CLK-OUT : out std_logic;
    PORTB inout std_logic_vector (7 downto 0);
    SDA : out std_logic;
    SCL : out std_logic;
    USR in out std_logic_vector (7 downto 0);
    GP : inout std_logic_vector(7 downto 0); -- general purpose?
    POR : in std_logic_vector(7 downto 0);
    CP : in std_logic_vector(7 downto 0); -- general purpose?
    SNES : in std_logic;
    RAM : in std_logic;
    T_DONE : out std_logic;
    RDY : out std_logic;
    FPGA_RST : in std_logic; -- global reset
    WLP : in std_logic;
    WLPBn : in std_logic;
    RD : in std_logic;
    CMDPA : in std_logic;
    CMDPB : in std_logic;
    OSEL : out std_logic;
    OSELABC : out std_logic;
    CDIN : in std_logic;
    CET : out std_logic;
    C_RESETn : out std_logic;
    MLDIN : in std_logic;
    RLCLK : out std_logic);
end main_test3;

architecture Behavioral of main_test3 is
```

128
component PULLDOWN
  port (O: out std_logic);
end component;

component dll-divide
  port (CLKIN : in std_logic;
        RESET : in std_logic;
        CLKO : out std_logic;
        LOCKED: out std_logic);
end component;

component par_dfftest
  port (elk : in std_logic;
        reset_n : in std_logic;
        port_a : in std_logic_vector (7 downto 0);
        data_buffer_out : out std_logic_vector (63 downto 0);
        enable : in std_logic;
        wr_pa_n : in std_logic;
        cmd_pa : in std_logic;
        full : out std_logic;
        filling_up : out std_logic);
end component;

component cdac_top_2port
  port (elk : in std_logic;
        rcset_n : in std_logic;
        check_word_in : in std_logic_vector (7 downto 0);
        chcck_word-out : out std_logic_vector (7 downto 0);
        mcm_data-in : in std_logic_vector (63 downto 0);
        mem_data_out : out std_logic_vector (63 downto 0);
        cpu_data_in : in std_logic_vector (63 downto 0);
        cpu_data_out : out std_logic_vector (63 downto 0);
        direction : in std_logic;
        single_error : out std_logic;
        multiple_error : out std_logic);
end component;

component cache
  port (addra: IN std_logic_VECTOR(3 downto 0);
        addrb: IN std_logic_VECTOR (3 downto 0);
        clka: IN std_logic;
        clkb: IN std_logic;
        dina: IN std_logic_VECTOR(127 downto 0);
        dinb: IN std_logic_VECTOR(127 downto 0);
        douta: OUT std_logic_VECTOR(127 downto 0);
        doutb: OUT std_logic_VECTOR(127 downto 0);
        wea : IN std_logic;
        web: IN std_logic);
end component;

component scr_dfftest
  port (elk : in std_logic;
        reset_n : in std_logic;
        enable : in std_logic;
        read_data_n : in std_logic;
        write-data_n : in std_logic;
        advance : in std_logic;
        empty : out std_logic;
        data_out : out std_logic_vector (7 downto 0);
        data_in : in std_logic_vector (63 downto 0);
        temp_out : out std_logic_vector (7 downto 0));
end component;

component cache_control
  port (clik : in std_logic;
        reset_n : in std_logic;
        par_full : in std_logic;
        ser_empty : in std_logic;
        bank_full : out std_logic;
        bank_empty : out std_logic;
        next_word : in std_logic;
        bank_address_a : out std_logic_VECTOR(3 downto 0);
        cache_address_a : out std_logic_VECTOR(3 downto 0);
        cache_enable_a : out std_logic;
        cache_control : out std_logic);
end component;
CHAPTER C — CODE LISTINGS

125  cache.write.enable.a : out std_logic; -- enable blockram write
    cache.address.a : out std_logic vector(3 downto 0); out std_logic;
    cache.enable.b : out std_logic;
    cache.write.enable.b : out std_logic;
    direction : in std_logic; -- to '0' sdram / from '1'
    write.signa : out std_logic; -- write -signal to serialiser
    temp.cache.empty_out : out std_logic;
    reset.cache : in std_logic;
    -- to reset cache in case of direction change etc
    startup : in std_logic;
end component;

140  component cmd_controller is
    Port ( clock, count, out : out std_logic vector(4 downto 0); -- just for testing purp.
    current.state, out : out std_logic vector(4 downto 0); -- just for testing purposes
    cmd_temp_out : out std_logic vector(3 downto 0); -- indicates MM-cmd being issued
    mode_r_set : out std_logic;
    r_w : out std_logic;
    abort : out std_logic
    ref_req_out : out std_logic;
    precharge : out std_logic;
    rdc.nt : in std_logic;
    refresh_request : out std_logic;
    refresh_active : out std_logic;
    refresh.timer : out std_logic;
    ref.active : out std_logic;
    refresh: in std_logic;
    refresh_commands : out std_logic;
    refresh.mode : in std_logic;
    refresh.rate : in std_logic;
    refresh.tool : in std_logic;
    refresh.timeout : in std_logic;
    refresh.write : in std_logic;
    refresh.start : in std_logic;
    refresh.request : in std_logic;
    refresh.ack : in std_logic;
    refresh.status : in std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
    refresh.count : in std_logic;
    refresh.address : in std_logic;
    refresh.data : in std_logic;
    refresh.pulse : in std_logic;
    refresh.in : in std_logic;
    refresh.out : out std_logic;
    refresh.error : in std_logic;
    refresh.flags : in std_logic;
count : in std_logic; — indicates unit to increment address
reset_address : in std_logic;
shift : in std_logic; — derived from wr_pa or wr_pb, used to program both start and end addresses
addr latch : in std_logic;
adr byte : in std_logic_vector (7 downto 0); — to fit to external 8 bit data bus AND/OR command controller
load done : out std_logic;
adr done : out std_logic; — indicates end address is reached
endadr out : out std_logic_vector (7 downto 0);
workadr out : out std_logic_vector (7 downto 0);
addr : out std_logic_vector (13 downto 0));
end component;

component refresh timer
Port (clk : in std_logic;
reset_n : in std_logic;
sdram active : in std_logic; — indicates sdram accessed, manual refresh (to memory controller)
sdram off : in std_logic;
sdram not in use if = '1'
ref req : out std_logic; — makes a request memory controller to execute a refresh command
ref ack : in std_logic; — reply from controller to deassert latched request, this should be done as soon as possible);
end component;

constant divider : integer := 8388608;
— clock dividing dll
signal CLK : std_logic; — will serve as system clock, whether through dll or straight
signal latched : std_logic;
signal reset_n : std_logic;
signal reset temp : std_logic;
— cache controller
signal addr : std_logic_vector (3 downto 0);
signal addrb : std_logic_vector (3 downto 0);
signal clka std_logic;
signal clkb : std_logic;
signal dina : std_logic_vector (127 downto 0);
signal douta : std_logic_vector (127 downto 0);
signal dinb : std_logic_vector (127 downto 0);
signal doutb : std_logic_vector (127 downto 0);
signal wea : std_logic;
signal web : std_logic;
signal web temp : std_logic;
— 'main' internal signals
signal numbers : std_logic_vector (24 downto 0);
signal values : integer range 0 to 255;
signal number : integer range 0 to 8388607;
signal xdata_l : std_logic_vector (7 downto 0);
signal xdata_fl : std_logic_vector (7 downto 0);
signal xdata_f2 : std_logic_vector (7 downto 0);
signal cmd_pa_l : std_logic;
signal cmd-pa_fl : std_logic;
signal cmd_pa_f2 : std_logic; — removed this, caused timing problems
signal wr_pan_fl : std_logic;
signal wr_pan_f2 : std_logic;
signal wr_pan_f2_prev : std_logic;
signal rd_pbn_fl : std_logic;
signal rd_pbn_f2 : std_logic;
begin
  -- par_dff test
  signal par_full : std_logic;
  signal filling_up_temp : std_logic;
  signal value2 : integer range 0 to 255;
  signal par_enable : std_logic;
  signal data_buf_in : std_logic_vector (63 downto 0);
  -- edac
  signal check_word_in : std_logic_vector(7 downto 0);
  signal mem_data_out : std_logic_vector(63 downto 0);
  signal direction : std_logic;
  signal single_error : std_logic;
  signal multiple_errors : std_logic;
  -- cache controller
  signal cache_bank_full, cache_bank_empty : std_logic;
  signal check_word_out : std_logic_vector(7 downto 0);
  signal memo_data_in : std_logic_vector(63 downto 0);
  signal cpu_data_in : std_logic_vector(63 downto 0);
  signal mem_data_out : std_logic_vector(63 downto 0);
  signal direction : std_logic;
  signal cache_bank_full, cache_bank_empty : std_logic;
  signal cache_next_word, cache_enable : std_logic;
  signal cache_enable_b : std_logic;
  signal reset_cache : std_logic;
  signal temp_cache_empty_out : std_logic;
  -- COMMAND CONTROLLER / CENTRAL CONTROLLER
  signal clock_count_out : std_logic_vector(4 downto 0);
  signal current_state_out : std_logic_vector(4 downto 0);
  signal cmd_temp_out : std_logic_vector(3 downto 0);
  signal addr_set_pulse : std_logic;
  signal addr_load_done : std_logic;
  signal reset_address : std_logic;
  signal addr_bit : std_logic_vector(7 downto 0);
  signal addr_done : std_logic;
  signal a_temp : std_logic_vector(13 downto 0);
  signal addr_in : std_logic_vector(13 downto 0);
  signal dram_start : std_logic;
  signal dram_abort : std_logic;
  signal mr_set : std_logic;
  signal sdram_rw : std_logic;
  signal dram_refreq_to_cmd : std_logic;
  signal sdram_refreq_to_dram : std_logic;
  signal ref_ack : std_logic;
  signal sdram_active : std_logic;
  signal sdram_off : std_logic;
  signal timer_ref : std_logic;
  signal cmd_in : std_logic_vector(3 downto 0);
  signal startup : std_logic;
  signal precharge : std_logic;
  -- sdram controller
  signal cmd_out : std_logic_vector(4 downto 0);
  signal addrLatch_out_temp : std_logic;
  signal data_dir_out : std_logic;
  signal addr_shift_out : std_logic;
  signal data_valid_out_temp : std_logic;
  signal addr_shift_out_temp : std_logic;
  signal dram_bank_full, dram_bank_empty : std_logic;
  signal dram_next_word, dram_enable : std_logic;
  signal dram_enable_b : std_logic;
  signal reset_dram : std_logic;
  signal temp_dram_empty_out : std_logic;
  -- address unit
  signal dram_bank_full, dram_bank_empty : std_logic;
  signal temp_out : std_logic_vector(7 downto 0);
  signal temp_out2 : std_logic_vector(7 downto 0);
  -- refresh timer
  signal ref_reset : std_logic;
  begin
    counting : process(CLK, reset_n, numbers)
begin
  if reset_n = '0' then
    numbers <= (others => '0');
  elsif clk'event and clk = '1' then
    numbers <= numbers + 1;
  end if;
end process counting;

-- sets all user/general purpose output pins to relevant or non-interfering values
360
usr_pins : process(reset_n, ydata_in, cmd_pb, c_din, m_din, wr_pan, rd_pbn, current_state_out)
begin
  if reset_n = '0' then
    USR <= ydata_in;
  else
    USR <= (CMD..PB OR C..DIN OR M..DIN) & WR..PAn & RD..Pbn & current_state_out(4 downto 0);
  end if;
end process usr_pins;

-- sets all non-used ports to default/non-interfering values
370
data_handle : process(reset_n, dq_out, elk, CASn-temp, RASn_temp, WEn-temp)
begin
  if reset_n = '0' then
    RASn <= '1';
    WEn <= '1';
    CASn <= '1';
    ROY <= '1';
    T_DONE <= '0';
    OE-ABCD-1 <= '0';
    C..ENn <= 'Z';
    C..RESETn <= 'Z';
    FPGA.ON <= 'Z';
    SDA <= 'l';
    SCL <= '0';
  else
    RASn <= RASn_temp;
    WEn <= WEn_temp;
    CASn <= CASn_temp;
    ROY <= '0';
    T_DONE <= '1';
    OE-ABCD-1 <= '1';
    C..ENn <= 'l';
    C..RESETn <= 'l';
    FPGA.ON <= 'l';
    SDA <= '0';
    SCL <= '0';
  end if;
end process data_handle;

400
get_writepa : process(CLK, WRPAn, reset_n, wr_pan_f1)
begin
  if reset_n = '0' then
    wr_pan_f1 <= '1';
    wr_pan_f2 <= '1';
  elsif CLK'event and CLK = '1' then
    wr_pan_f1 <= WRPAn;
    wr_pan_f2 <= wr_pan_f1;
  end if;
end process get_writepa;

410
-- RD.PBn synchronization
-- removes metastability
-- rd_pbn_f2 can now be used in rest of design without fear of it being unstable
415
get_readpb : process(CLK, RD.PBn, reset_n, rd_pbn_f1)
begin
  if reset_n = '0' then
    rd_pbn_f1 <= '1';
    rd_pbn_f2 <= '1';
  elsif CLK'event and CLK = '1' then
    rd_pbn_f1 <= RD.PBn;
    rd_pbn_f2 <= rd_pbn_f1;
  end if;
end process get_readpb;

-- porta / XDATA delay
-- port A is input only
delay_porta : process(CLK, reset_n, xdata_f1, PortA)
begin
  if reset_n = '0' then
    xdata_f1 <= (others => '0');
    xdata_f2 <= (others => '0');
  elsif CLK'event and CLK='1' then
    xdata_f1 <= PortA;
    xdata_f2 <= xdata_f1;
  end if;
end process delay_porta;

delay cmd pa : process (CLK, reset_n, cmd_pa_f1, cmd_pa)
begin
  if reset_n = '0' then
    cmd_pa_f1 <= '1';
    cmd_pa_f2 <= '1';
  elsif CLK'event and CLK = '1' then
    cmd_pa_f1 <= cmd_pa;
    cmd_pa_f2 <= cmd_pa_f1;
  end if;
end process delay cmd pa;

-- port b/ydata initial operation
-- I/O operation, for the moment only output
ydata out gen : process (reset_n, srl.data out)
begin
  if reset_n = '0' then
    ydata_out <= (others => '0');
  else
    ydata_out <= srl.data out;
  end if;
end process ydata out gen;

io port b : process (reset_n, RD PBn, ydata out, WR PBn)
begin
  if reset_n = '0' then
    PortB <= (others => 'Z');
  elsif RD PBn = '0' then
    PortB <= ydata out;
  else
    PortB <= (others => 'Z');
  end if;
end process io port b;

-- initial value assignments

-- data is valid when WR pan f2 goes low
wr hist : process (CLK, reset_n, wr pan f2)
begin
  if reset_n = '0' then
    wr pan f2 prev'1';
  elsif CLK'event and CLK = '1' then
    wr pan f2 prev <= wr pan f2;
  end if;
end process wr hist;

parl : par dff test port map(
  clk => CLK,
  reset n => reset n,
  port a => xdata f2,
  data buffer out => cpu data in,
  enable => par enable,
  output enable n => direction,
  -- '0' into ram (write)
  wr pa => WR PA n f2,
  cmd pa => cmd pa f2,
  -- was cmd pa f1
  full => par full,
  filling up => filling up temp);

edac top 2 port1 : edac top 2 port port map(
  clk => CLK,
  reset n => reset n,
  check word in => check word in,
  check word out => check word out,
  mem data in => mem data in,
  mem data out => mem data out,
  cpu data in => cpu data in,
  cpu data out => cpu data out,


```vhdl
510 515 520 525 530 535 540
CHAPTER C — CODE LISTINGS

cachel: cache port map(
  addra => addra,
  addrb => addrb,
  clka => clka,
  clkb => clkb,
  dina => dina,
  dinb => dinb,
  douta => douta,
  doutb => doutb,
  wea => wea,
  web => web temp);

NBNBNBNB change back to web, this is for testing purposes
-- to prevent data in cache to be overwritten by garbage
-- testing edac + blockram

ser_dfftestl : ser_dfftest port map(
  clk => CLK,
  reset_n => reset_n,
  enable => ser_enable_temp,
  read_data_n => RDI.PBn.f2,
  write_data_n => write_data_n,
  advance => advance,
  empty => ser_empty,
  data_out => ser_data_out,
  data_in => ser_data_in,
  temp_out => ser_temp_out);

545 550 555 560 565 570

ser_dfftestl : ser_dfftest port map(
  clk => CLK,
  reset_n => reset_n,
  par_full => par_full,
  ser_empty => ser_empty,
  bank_full => cache_bank_full,
  bank_empty => cache_bank_empty,
  next_word => cache_next_word,
  cache_address_a => addra,
  cache_address_b => addrb,
  cache_enable_a => cache_enable_a,
  cache_enable_b => cache_enable_b,
  cache_write_enable_a => wca,
  cache_write_enable_b => web,
  direction => direction
  temp_cache_empty_out => temp_cache_empty_out,
  reset_cache => reset_cache,
  startup => startup);

570 575 580 585

-- misc control signals
ser_advance => advance,
ser_enable => ser_enable,
par_enable => par_enable,

-- address controller interface
address_set_pulie => adr_set_pulie,
address_load_done => adr_load_done,
reset_address => reset_address,
adr_byte => adr_byte,
adr_done => adr_done,
address_out => a_temp,
address_in => adr_in,

cachc_cmpty => temp_cache_empty_out,
bank_full => cache_bank_full,

570 575 580 585

-- cache controller interface

```
CHAPTER C — CODE LISTINGS

--- sdram controller interface

start => sdram_start,
mode_set => m.set,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
start => sdram_start,
reset_n => reset_n,
sdram_start => sdram_start,
ref_req_out => sdram_ref.req.to_sDRAM,
ref_ack_out => sdram_ref.req.to_cmd,
precharge => precharge,
precharge => precharge,
C.1.2 Full System - VHDL Testbench

Write Operation

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

5 ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

10 COMPONENT main_test3
PORT(
  Q_CLK : IN std_logic;
  DLL_CLK : IN std_logic;
  PortA : IN std_logic_vector (7 downto 0);
  SELECT_RAM : IN std_logic;
  FPGA_RSTn : IN std_logic;
  WR_PA : IN std_logic;
  WR_PB : IN std_logic;
  R_P : IN std_logic;
  CMD_PA : IN std_logic;
  CMD_PB : IN std_logic;
  C_DIN : IN std_logic;
  M_DIN : IN std_logic;
  OB : INOUT std_logic_vector(7 downto 0);
  DQ : INOUT std_logic_vector(63 downto 0);
  PortB : INOUT std_logic_vector(63 downto 0);
  USR : INOUT std_logic_vector(7 downto 0);
  GP : INOUT std_logic_vector(7 downto 0);
)
CHAPTER C — Code Listings

A : OUT std_logic_vector (13 downto 0);
Sn : OUT std_logic_vector (3 downto 0);
RASn : OUT std_logic;
CASn : OUT std_logic;
WEn : OUT std_logic;
FPGA.ON : OUT std_logic;
DLLCLK-OUT : OUT std_logic;
SDA : OUT std_logic;
SCL : OUT std_logic;
T.DONE : OUT std_logic;
RDY : OUT std_logic;
OE.A.BCD.1 : OUT std_logic;
C.ENe : OUT std_logic;
C.RESETn : OUT std_logic;
RCLK : OUT std_logic;

END COMPONENT;

SIGNAL CB : std_logic_vector (7 downto 0);
SIGNAL A : std_logic_vector (13 downto 0);
SIGNAL Sn : std_logic_vector (3 downto 0);
SIGNAL RASn : std_logic;
SIGNAL CASn : std_logic;
SIGNAL WEn : std_logic;
SIGNAL DQ : std_logic_vector (63 downto 0); := (others => 'Z');
SIGNAL 0-CLK : std_logic := '0';
SIGNAL DLL-CLK : std_logic;
SIGNAL FPGA.ON : std_logic;
SIGNAL PortB : std_logic_vector (7 downto 0); := (others => 'Z');
SIGNAL SDA : std_logic;
SIGNAL SCL : std_logic;
SIGNAL USR : std_logic_vector (7 downto 0); := (others => 'Z');
SIGNAL PortA : std_logic_vector (7 downto 0); := (others => 'Z');
SIGNAL GP : std_logic_vector (7 downto 0); := (others => 'Z');
SIGNAL SELECT.RAM : std_logic := '1';
SIGNAL T.DONE : std_logic;
SIGNAL RDY : std_logic;
SIGNAL FPGA.RSTn : std_logic := '0';
SIGNAL CMD.PA : std_logic := '1';
SIGNAL CMD.PB : std_logic := '1';
SIGNAL CMD.PBn : std_logic := '0';
SIGNAL OE.A.BCD.1 : std_logic;
SIGNAL C.DIN : std_logic;
SIGNAL C.ENe : std_logic;
SIGNAL C.RESETn : std_logic;
SIGNAL M.DIN : std_logic;
SIGNAL ILCLK : std_logic;

constant cycle : time := 20 ns;

-- command constants
costant c_idle : std_logic_vector (3 downto 0); := "0000";
costant c_set_adr_a : std_logic_vector (3 downto 0); := "0001";
costant c_set_adr_b : std_logic_vector (3 downto 0); := "0010";
costant c_wr_b : std_logic_vector (3 downto 0); := "0100";
costant c_rd_b : std_logic_vector (3 downto 0); := "0101";
costant c_status : std_logic_vector (3 downto 0); := "0110";
costant c_mode_r : std_logic_vector (3 downto 0); := "0111";
costant c_sdram_off : std_logic_vector (3 downto 0); := "1000";
costant c_shutdown : std_logic_vector (3 downto 0); := "1111";

--main_test3 PORT MAP(
begin
DLL.CLK => DLL.CLK,
FPGA.CLK => FPGA.CLK,
DLL.CLK.OUT => DLL.CLK.OUT,
PortB => PortB,
SDA => SDA,
SCL => SCL,
USR => USR,
PortA => PortA,
GP => GP,
SELECT.RAM => SELECT.RAM,
T.DONE => T.DONE,
RDY => RDY,
FPGA.RSTn => FPGA.RSTn,
WR.PAn => WR.PAn,
RD.PBn => RD.PBn,
CMD.PA => CMD.PA,
CMD.PB => CMD.PB,
OE.ABCD1 => OE.ABCD1,
C.DIN => C.DIN,
C.ENA => C.ENA,
C.RESETn => C.RESETn,
M.DIN => M.DIN,
M.CLK => M.CLK

=== Test Bench — User Defined Section ===

clocking : PROCESS
BEGIN
O.CLK <= not O.CLK;
wait for 0.5•cycle;
END PROCESS;
resetting : PROCESS
BEGIN
wait for 5•cycle;
FPGA.RSTn <= '1';
wait;
END PROCESS;
operation : PROCESS
BEGIN
wait until FPGA.RSTn = '1';
wait for 30•cycle;
— address command
PortA <= "0000" & c.set.adr.a;
CMD.PA <= '1';
WR.PAn <= '0';
wait for cycle;
CMD.PA <= '0';
WR.PAn <= '1';
wait for cycle;
— address bytes
— byte 1 – 3
PortA <= "00000000";
k <= 0;
wait for cycle;
while k < 3 loop
  k <= k + 1;
  WR.PAn <= '0';
  wait for 33•cycle;
  WR.PAn <= '1';
  wait for 2•cycle;
end loop;
—byte 4
PortA <= "00011111";
WR.PAn <= '0';
wait for 33•cycle;
WR.PAn <= '1';
wait for 2•cycle;
—byte 5
PortA <= "00000000";
WR.PAn <= '0';
wait for 33•cycle;
CHAPTER C — Code Listings

```vhdl
WRJ'An <= '1';
wait for 2*cycle;

-- bytes 6
PortA <= "00000000";
WRJPAn <= '0';
wait for 3.3*cycle;
WRJPAn <= '1';
wait for 2*cycle;

-- write port a command
wait for 10*cycle;
PortA <= "00000000"&c.wr_a;
CMD.PA <= '1';
WRJPAn <= '0';
wait for cycle;
CMD.PA <= '0';
WRJPAn <= '1';

k<=0;
wait for 3*cycle;
while k<256 loop
  k <= k + 1;
  PortA <= "00110011";
  WRJ'An <= '0';
  wait for 3*cycle;
  WRJ'An <= '1';
  wait for 2*cycle;
end loop;
wait;
END PROCESS;

tb : PROCESS
BEGIN
  -- will wait forever
END PROCESS;

--- End Test Bench — User Defined Section ---
END;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

COMPONENT main_test3
PORT(
  CLIN : IN std_logic;
  DLL.CLK : IN std_logic;
  PortA : IN std_logic_vector(7 downto 0);
  SELECT.RAM : IN std_logic;
  FPGA.LESTA : IN std_logic;
  WRJPAn : IN std_logic;
  WRJ'An : IN std_logic;
  RD.PBn : IN std_logic;
  CMD.PB : IN std_logic;
  CMD.PA : IN std_logic;
  C.DIN : IN std_logic;
  M.DIN : IN std_logic;
  CB : INOUT std_logic_vector(7 downto 0);
  DQ : INOUT std_logic_vector(63 downto 0);
  PortB : INOUT std_logic_vector(7 downto 0);
  USR : INOUT std_logic_vector(7 downto 0);
  GP : INOUT std_logic_vector(7 downto 0);
  A : OUT std_logic_vector(13 downto 0);
  S0 : OUT std_logic_vector(3 downto 0);
  RASn : OUT std_logic;
  CASn : OUT std_logic;
```
CHAPTER C — CODE LISTINGS

```vhdl
WEn : OUT std_logic;
FPGA.ON : OUT std_logic;
35 DLL_CLK_OUT : OUT std_logic;
SDA : OUT std_logic;
SCL : OUT std_logic;
TDONE : OUT std_logic;
RDY : OUT std_logic;
OE.ABCD1 : OUT std_logic;
CLEn : OUT std_logic;
C.RESETn : OUT std_logic;
1CCLK : OUT std_logic;

END COMPONENT;

SIGNAL CB : std_logic_vector (7 downto 0);
SIGNAL A : std_logic_vector (13 downto 0);
SIGNAL Sn : std_logic_vector (3 downto 0);
SIGNAL DQ : std_logic;
40 SIGNAL DLL_CLK : std_logic;
SIGNAL FPGA..ON : std_logic;
SIGNAL DLL_CLK_OUT : std_logic;
PORTB : std_logic_vector (7 downto 0);
SDA : std_logic;
SIGNAL SCL : std_logic;
SIGNAL USR : std_logic_vector (7 downto 0);
SIGNAL PortA : std_logic_vector (7 downto 0);
SIGNAL SELECT.RAM : std_logic;
SIGNAL T.DONE : std_logic;
50 SIGNAL RDY : std_logic;
SIGNAL FPGA..RSTn : std_logic := '0';
SIGNAL WIU'A : std_logic;
SIGNAL WIU'B : std_logic;
SIGNAL WIU'P : std_logic;
SIGNAL CMD.PA : std_logic;
SIGNAL CMD.PB : std_logic;
SIGNAL OE.ABCD1 : std_logic;
SIGNAL OE.ABCD2 : std_logic;
SIGNAL M.DIN : std_logic;
SIGNAL R..CLK : std_logic := '1';
60 SIGNAL OE..ABCD1 : std_logic;
SIGNAL OE..ABCD2 : std_logic;
SIGNAL M..DIN : std_logic;
SIGNAL R..CLK : std_logic := '1';
SIGNAL OE..ABCD1 : std_logic := '0';
SIGNAL OE..ABCD2 : std_logic := '0';
SIGNAL M..DIN : std_logic := '0';
SIGNAL R..CLK : std_logic := '0';
70 constant cycle : time := 20 ns;
--- command constants
constant c_idle : std_logic_vector (3 downto 0) := "0000";
constant c_set adr_a : std_logic_vector (3 downto 0) := "0001";
75 constant c_set adr_b : std_logic_vector (3 downto 0) := "0010";
constant c_wr : std_logic vector (3 downto 0) := "0011";
constant c_rd : std_logic vector (3 downto 0) := "0100";
constant c_status : std_logic_vector (3 downto 0) := "0101";
constant c_mode_r : std_logic vector (3 downto 0) := "0110";
constant c_sdram_off : std_logic vector (3 downto 0) := "0111";
--- unused : 1000 to 1110
constant c_shutdown : std_logic vector (3 downto 0) := "1111";
80 signal k : integer;

BEGIN
uut : main_test3 PORT MAP(
CB => CB,
A => A,
85 Sn => Sn,
RASn => RASn,
CASn => CASn,
WEn => WEn,
DQ => DQ,
O.CLK => O.CLK,
DLL_CLK => DLL_CLK,
FPGA.ON => FPGA.ON,
100 DLL_CLK_OUT => DLL_CLK_OUT,
PORTB => PORTB,

Stellenbosch University  https://scholar.sun.ac.za
Stellenbosch University  https://scholar.sun.ac.za
CHAPTER C — CODE LISTINGS

```plaintext
SDA => SDA,
SCL => SCL,
USR => USR,
PortA => PortA,
GP => GP,
SELECT.RAM => SELECT.RAM,
T_DONE => T_DONE,
RDY => RDY,
FPGA_RSTn => FPGA_RSTn,
WRLPA => WRLPA,
WRLPB => WRLPB,
RD_PLL => RD_PLL,
CMDPA => CMDPA,
CMDPB => CMDPB,
OE_A => OE_A,
C.DIN => C.DIN,
C.EN => C.EN,
C.RESET => C.RESET,
M.DIN => M.DIN,
R.CLK => R.CLK
```

---

```plaintext
*** Test Bench — User Defined Section ***
```

```plaintext
clocking : PROCESS
BEGIN
  O.CLK <= not O.CLK;
  wait for 0.5*cycle;
END PROCESS;
```

```plaintext
resetting : PROCESS
BEGIN
  wait for 5*cycle;
  FPGA_RSTn <= '1';
  wait;
END PROCESS;
```

```plaintext
operation : PROCESS
BEGIN
  wait until FPGA_RSTn = '1';
  wait for 50*cycle;
  — address command
  PortA <= "0000" & c.set.adr.a;
  CMDPA <= '1';
  WRLPA <= '0';
  wait for cycle;
  CMDPA <= '0';
  WRLPA <= '1';
  wait for cycle;
  — address bytes
  — byte 1
  PortA <= "00000000";
  k <= 0;
  wait for cycle;
  while k < 3 loop
    k<=k+1;
    WRLPA <= '0';
    wait for 3.3*cycle;
    WRLPA <= '1';
    wait for 2*cycle;
  end loop;
  — byte 4
  PortA <= "00011111";
  WRLPA <= '0';
  wait for 3.3*cycle;
  WRLPA <= '1';
  wait for 2*cycle;
  — byte 5
  PortA <= "00000000";
  WRLPA <= '0';
  wait for 3.3*cycle;
  WRLPA <= '1';
  wait for 2*cycle;
  — byte 6
```

```plaintext
```
Port A <= "00000000";
WR_PA <= '0';
wait for 3.3-cycle;
WR_PA <= '1';
wait for 2-cycle;
--- write port a command
wait for 10-cycle;
Port A <= "0000" & c.rd.b;
CMD_PA <= '1';
WR_PA <= '0';
wait for cycle;
CMD_PA <= '0';
WR_PA <= '1';

k<=0;
wait for 30-cycle;
DQ <= "0000000000000000000000000000000000000000000000000000000000000000";
CB <= "00001100";
while k<256 loop
k <= k + 1;
RD_PBn <= '0';
wait for 3-cycle;
RD_PBn <= '1';
wait for 2-cycle;
end loop;

wait;
END PROCESS;

tb : PROCESS
BEGIN
wait; --- will wait forever
END PROCESS;
--- *** End Test Bench - User Defined Section ***
END

---
CHAPTER C — CODE LISTINGS

C.2 Address Unit

C.2.1 Address Unit - VHDL Code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity address_unit is
  Port (clk : in std_logic;
        reset_n : in std_logic;
        count : in std_logic; -- indicates unit to increment address
        shift : in std_logic;
        set : in std_logic; -- derived from wr_pa or wr_pb, used to program both start and end addresses;
        adr.latch : in std_logic;
        adr.byte : in std_logic_vector(7 downto 0); -- to fit to external 8 bit data bus AND/OR command controller
        load : in std_logic;
        endadr : out std_logic;
        workadr : out std_logic_vector(7 downto 0);
        addr : out std_logic_vector(13 downto 0));
end address_unit;

architecture behavioral of address_unit is
  -- signal ra : std_logic_vector(11 downto 0);
  -- signal ca : std_logic_vector(8 downto 0);
  signal intern-addr : std_logic_vector(15 downto 0); -- allows for 16M-word range
  signal action : std_logic_vector(1 downto 0); -- combines signals for case statement
  signal adr.bytes_no : integer range 0 to 5;
  signal work_addr : std_logic_vector(23 downto 0);
  signal end_address : std_logic_vector(23 downto 0);
  signal set.prev : std_logic;
begin
  set_adr : process(clk, reset_n, reset.address, set, set.prev, adr.byte, addr.byte.no, count, intern.addr, work.addr, end_address)
  begin
    if reset_n = '0' then
      intern.addr <= (others => '0');
      addr.byte.no <= 0;
      load <= '0';
      work.addr <= (others => '0');
      end.address <= (others => '1');
      elsif clk'd'event and clk = '1' then
        if reset_address = '1' then
          intern.addr <= (others => '0');
          work.addr <= (others => '0');
          end.address <= (others => '1');
          addr.byte.no <= 0;
        load <= '0';
        elsif set = '1' and (set.prev = '0' or not(addr.byte.no = 0)) then
          case addr.byte.no is
            when 0 => intern.addr(7 downto 0) <= adr.byte; load <= '0';
            when 1 => intern.addr(15 downto 8) <= adr.byte; load <= '0';
            when 2 => load <= '0'; work.addr <= adr.byte & intern.addr(15 downto 0);
            when 3 => end.address(7 downto 0) <= adr.byte; load <= '0';
            when 4 => end.address(15 downto 8) <= adr.byte; load <= '0';
            when 5 => end.address(23 downto 16) <= adr.byte; load <= '1';
            when others => load <= '0';
          end case;
        end if;
        elsif count = '1' then
          load <= '0';
          addr.byte.no <= 0;
        end if;
    end if;
  end process set_adr;
end behavioral;
```

Stellenbosch University  https://scholar.sun.ac.za
## C.2.2 Address Unit - VHDL Testbench

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

COMPONENT address_unit
PORT(
  clk : IN std_logic;
  reset_n : IN std_logic;
  count : IN std_logic;
  reset_address : IN std_logic;
  shift : IN std_logic;
  set : IN std_logic;
  adr_byte : IN std_logic_vector(7 downto 0);
  load_done : OUT std_logic;
  addr_done : OUT std_logic;
  endadr_out : OUT std_logic_vector(7 downto 0);
  workadr_out : OUT std_logic_vector(7 downto 0);
  addr : OUT std_logic_vector(13 downto 0));
END COMPONENT;

SIGNAL clk : std_logic := '0';
SIGNAL reset_n : std_logic := '0';
SIGNAL count : std_logic := '0';
SIGNAL reset_address : std_logic := '0';
SIGNAL shift : std_logic := '0';
SIGNAL set : std_logic := '0';
SIGNAL adr_byte : std_logic_vector(7 downto 0) := (others => '0');
SIGNAL load_done : std_logic;
SIGNAL addr_done : std_logic;
SIGNAL endadr_out : std_logic_vector(7 downto 0);
SIGNAL workadr_out : std_logic_vector(7 downto 0);
SIGNAL addr : std_logic_vector(13 downto 0);
constant cycle : time := 20 ns;

BEGIN
  uut: address_unit PORT MAP(
    clk => clk,
    ...)
reset_n \Rightarrow \text{reset.n}
\quad \text{count} \Rightarrow \text{count}
\quad \text{reset.address} \Rightarrow \text{reset.address}
\quad \text{shift} \Rightarrow \text{shift}
\quad \text{set} \Rightarrow \text{set}
\quad \text{adr.byte} \Rightarrow \text{adr.byte}
\quad \text{load.done} \Rightarrow \text{load.done}
\quad \text{adr.done} \Rightarrow \text{adr.done}
\quad \text{endadr.out} \Rightarrow \text{endadr.out}
\quad \text{workadr.out} \Rightarrow \text{workadr.out}
\quad \text{addr} \Rightarrow \text{addr}

--- Test Bench - User Defined Section ---

clocking : process
begin
    \text{clk} \Leftarrow \neg(\text{clk})
    \text{wait for 0.5 cycle}
end process

setting : process
begin
    \text{wait until reset.n = '1'}
    \text{wait for 0.25 cycle}
    \text{wait for cycle}
    \text{set} \Leftarrow '1'
    \text{wait for cycle}
    \text{adr.byte} \Leftarrow '1010001'; -- A1
    \text{wait for cycle}
    \text{adr.byte} \Leftarrow '10100010'; -- A2
    \text{wait for cycle}
    \text{adr.byte} \Leftarrow '10100011'; -- A3
    \text{wait for cycle}
    \text{adr.byte} \Leftarrow '10110000'; -- AS
    \text{wait for cycle}
    \text{adr.byte} \Leftarrow '10100010'; -- A2
    \text{wait for cycle}
    \text{adr.byte} \Leftarrow '10100011'; -- A3
    \text{set} \Leftarrow '0'
    \text{wait until load.done = '0' and load.done'event = true}
    \text{wait for 0.25 cycle}
    \text{shift} \Leftarrow '1'
    \text{wait for cycle}
    \text{shift} \Leftarrow '0'
    \text{while \text{adr.done} = '0'}
    \text{wait until \text{count} = '0' and \text{count}'event = true}
    \text{wait for 0.25 cycle}
    \text{if \text{adr.done} = '0'}
    \text{shift} \Leftarrow '1'
    \text{wait for cycle}
    \text{shift} \Leftarrow '0'
end if
end loop
wait;
end process

resetting : process
begin
    \text{wait for 0.5 cycle}
    \text{wait until \text{adr.done} = '1' and \text{adr.done}'event = true}
    \text{wait for 0.25 cycle}
    \text{reset.address} \Leftarrow '1'
    \text{wait for cycle}
    \text{reset.address} \Leftarrow '0'
end process

counting.shifting : process
begin
    \text{wait until \text{set} = '1'}
    \text{wait for 1 ns}
    \text{wait until \text{set} = '0'} and \text{set}'event = true
    \text{wait until \text{shift} = '0'} and \text{shift}'event = true
    \text{wait for 3 cycle}
    \text{while \text{adr.done} = '0'}
    \text{count} \Leftarrow '1'
end process

Stellenbosch University  https://scholar.sun.ac.za
end if;
end loop;
wait;
end process;
end PROCESS
BEGIN
reset_n <= '1' after 1.5 * cycle;
wait; -- will wait forever
END PROCESS;
--- *** End Test Bench - User Defined Section ***
END;
C.3 Cache

C.3.1 Cache - VHDL Code

```vhdl
-- This file is owned and controlled by Xilinx and must be used
-- solely for design, simulation, implementation and creation of
-- design files limited to Xilinx devices or technologies. Use
-- with non-Xilinx devices or technologies is expressly prohibited
-- and immediately terminates your license.
-- Xilinx products are not intended for use in life support
-- appliances, devices, or systems. Use in such applications are
-- expressly prohibited.
-- Copyright (C) 2001, Xilinx, Inc. All Rights Reserved.

-- You must compile the wrapper file cache.vhd when simulating
-- the core, cache. When compiling the wrapper file, be sure to
-- reference the XilinxCoreLib VHDL simulation library. For detailed
-- instructions, please refer to the "Coregen User Guide".
-- The synopsys directives "translate_off/translate_on" specified
-- below are supported by XST, FPGA Express, Exemplar and Synplicity
-- synthesis tools. Ensure they are correct for your synthesis tool(s).
-- synopsys translate_off
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

Library XilinxCoreLib;
ENTITY cache IS
  port (addra: IN std_logic_VECTOR(3 downto 0); addrb: IN std_logic_VECTOR(3 downto 0); clka: IN std_logic; clkb: IN std_logic; dina: IN std_logic_VECTOR(127 downto 0); dinb: IN std_logic_VECTOR(127 downto 0); douta: OUT std_logic_VECTOR(127 downto 0); doutb: OUT std_logic_VECTOR(127 downto 0); wea: IN std_logic; web: IN std_logic);
END cache;

ARCHITECTURE cache.a OF cache IS

COMPONENT wrapped_cache
  PORT (addra: IN std_logic_VECTOR(3 downto 0); addrb: IN std_logic_VECTOR(3 downto 0); clka: IN std_logic; clkb: IN std_logic; dina: IN std_logic_VECTOR(127 downto 0); dinb: IN std_logic_VECTOR(127 downto 0); douta: OUT std_logic_VECTOR(127 downto 0); doutb: OUT std_logic_VECTOR(127 downto 0); wea: IN std_logic; web: IN std_logic);
END COMPONENT;

-- Configuration specification
FOR all : wrapped_cache USE ENTITY XilinxCoreLib.blkmemdp-v3-l(behavioral)
  GENERIC MAP
    c_has_enb => 0,
    c_has_ena => 0,
    c_write_modcb => 0,
    c_pipe_stages_b => 0,
    c_write_modea => 0,
    c_pipe_stages_a => 0,
    c_addrb_width => 4,
    c_has_dinb => 1,
    c_has_dina => 1,
    c_has_doutb => 1,
    c_has_douta => 1,
c_has_dout => 1,
c_reg_inputb => 1,
c_has_ffdb => 0,
c_reg_inputa => 1,
c_has_rfda => 0,
c_mem_init_file => "mif_file_16.1",
c_sinita_value => "00000000000000000000000000000000",
c_has_sinitb => 0,
c_sinitb_value => "00000000000000000000000000000000",
c_has_web => 1,
c_depth_b => 16,
c_depth_a => 16,
c_has_rdyb => 0,
c_has_rdya => 0,
c_has_limit_data_pitch => 0,
c_enable_rloca => 0,
c_addra_width => 4);
BEGIN
UO : wrapped_cache
port map (addra => addra,
addrb => addrb,
cika => cika,
cikb => cikb,
dina => dina,
dinb => dinb,
doutb => doutb,
web => web);
END cache_a;
— synopsys translate.on
C.4 Cache Controller

C.4.1 Cache Controller - VHDL Code

```vhdl
library IEEE;
use IEEE.STD_LOGIC.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity cache_control is
  Port ( clk : in std_logic;
         reset_n : in std_logic;
         par_full : in std_logic;
         next_word : in std_logic;
         cmd_ctrl : in std_logic;
         bank_full : out std_logic;
         bank_empty : out std_logic;
         cache_address_a : out std_logic_vector (3 downto 0);
         cache_enable_a : out std_logic;
         cache_write_enable_a : out std_logic;
         cache_address_b : out std_logic_vector (3 downto 0);
         cache_enable_b : out std_logic;
         cache_write_enable_b : out std_logic;
         direction : in std_logic;
         write_ser_n : out std_logic;
         startup : in std_logic;
         reset_cache : in std_logic);
end cache-control;

architecture behavioral of cache_control is
  signal data_read : std_logic;
  signal cache_address_a_temp : std_logic_vector (2 downto 0);
  signal cache_address_b_temp : std_logic_vector (2 downto 0);
  signal cache_bank_a : std_logic;
  signal cache_bank_b : std_logic;
  signal waital,waita2,waita3 : std_logic;
  signal bank_full_int : std_logic;
  signal bank_empty_int : std_logic;
  signal data_writc : std_logic;
  signal cache-empty : std_logic;
  signal last_block_filled : std_logic;
  signal last_block_empty : std_logic;
  signal empty_int : std_logic;

begin
  read_history : process(clk, reset_n, par_full, data_read, direction, reset_cache)
  begin
    if (reset_n = '0')or(reset_cache = '1') then
      data_read <= '0';
    elsif clk'event and clk = '1' then
      if (direction = '0')and(par_full = '1')and(data_read = '0') then
        -- do cache move : process waits
        data_read <= '1';
      elsif (par_full = '0') then
        -- data has been read from parallelising unit
        data_read <= '0';
      end if;
  end if;
end cache_control;
```

Stellenbosch University  https://scholar.sun.ac.za
CHAPTER C — CODE LISTINGS

end if;
end process read_history;

write_history : process(clk, reset_n, empty_int, data_write, direction, reset_cache)
-- remembers if data has been written to serializing unit this cycle
begin
if (reset_n = '0') or (reset_cache = '1') then
  data_write <= '0';
elsif clk'event and clk = '1' then
  if (direction = '1') and (empty_int = '1') and (data_write = '0') then
    -- do cache move: process waits
    data_write <= '1';
  elsif (empty_int = '0') then -- data has been written to serializing unit
    -- do nothing
    data_write <= '0';
  end if;
end if;
end process write_history;

-- currently only writes to port a
waits_rw : process(clk, reset_n, waital, waita2, data_read, data_write, par_full, empty_int, cache_address_a_temp, next_word, direction, reset_cache, current_bank_a, current_bank_b, startup, last_block_filled, last_block_empty, cache_empty, cache_address_b_temp, bank_empty_int)
begin
if (reset_n = '0') or (reset_cache = '1') then
  -- NB look at possible case statement here
  cache_address_a_temp <= (others => '0');
  cache_address_b_temp <= (others => '0');
  cache_enable_a <= '1';
  cache_write_enable_a <= '0';
  waital <= '0';
  waita2 <= '0';
  current_bank_a <= '0';
  current_bank_b <= '1';
  write_ser_n <= '1';
  last_block_empty <= '0';
elsif clk'event and clk = '1' then
  if direction = '0' then
    -- indicate transfer initiation
    waital <= '1';
    cache_enable_a <= '1';
    cache_write_enable_a <= '1';
  elsif waital = '1' then
    cache_enable_a <= '0';
    cache_write_enable_a <= '0';
    waital <= '0';
    waita2 <= '1';
  elsif waita2 = '1' then
    -- increment address, check this for necessity,
    -- this could possibly be done in the previous state
    waita2 <= '0';
    -- only switch when
    if l.bank_full (port a addr = 111) and l.bank_empty (port b addr = 000) and don't switch when
    if cache_address_a_temp = "111" then
      current_bank_a <= not(current_bank_a); -- switch banks
      current_bank_b <= not(current_bank_b); -- switch banks
      cache_address_a_temp <= "000";
    else
      cache_address_a_temp <= cache_address_a_temp + 1;
    end if;
  end if;
end if;
if (par_full = '1') and (next_word = '1') then
  -- check port b (sdram-side) cache address
  if cache_address_b_temp = "111" then
    -- a bank was emptied: set empty bank flags
    cache_address_b_temp <= cache_address_b_temp + 1;
    -- a bank was read, clear full bank flags
  end if;
elsif (next_word = '1') and (cache_address_b_temp = "111") or ((waita2 = '1') and (cache_address_a_temp = "111")) then
  -- direction = '1', READ
end if;
current_bank_a <= not(current_bank_a);  -- switch banks
current_bank_b <= not(current_bank_b);  -- switch banks
end if;
if (next_word = '1') then
if cache_address_b_temp = "111" then
    cache_address_b_temp <= "000";
else
    cache_address_b_temp <= cache_address_b_temp + 1;
end if;
end if;
if (next_word = '1') then
    if cache_address_a_temp = "111" then
        cache_address_a_temp <= "000";
    else
        cache_address_a_temp <= cache_address_a_temp + 1;
    end if;
end if;
if (reset_n = '0') or (reset_cache = '1') then
    empty_banks <= "11";
    full_banks <= "00";
elsif clk'event and clk = '1' then
    if direction = '0' then  -- WRITE to sram
        empty_banks <= "11";
        full_banks <= "00";
    elsif clk'event and clk = '1' then
        empty_banks <= "00";
        full_banks <= "11";
    end if;
end if;
end if;
end process waits-rw;

banks : process (clk, reset_n, reset_cache, direction, cache_address_a_temp, current_bank_a, cache_address_b_temp, current_bank_b)
begin
if (reset_n = '0') or (reset_cache = '1') then
    empty_banks <= "11";
    full_banks <= "00";
elsif clk'event and clk = '1' then
    if direction = '0' then  -- WRITE to sram
        empty_banks <= "11";
        full_banks <= "00";
    elsif clk'event and clk = '1' then
        empty_banks <= "00";
        full_banks <= "11";
    end if;
end if;
end process waits-rw;
CHAPTER C — CODE LISTINGS

230

— bank 1 full flags
if (waita2 = '1') and (not (cache.address_a_temp = "111") and (current_bank_a = '1') then
full_banks(1) <= '0';
elsif (next_word = '1') and (cache_address_a_temp = "111") and (current_bank_a = '0') then
full_banks(1) <= '1';
end if;

— empty banks
if (waita2 = '1') and (cache_address_a_temp = "111") and (current_bank_a = '0') then
empty_banks(0) <= '1';
elsif (next_word = '1') and (not (cache_address_a_temp = "111") and (current_bank_a = '1') then
empty_banks(0) <= '0';
end if;

end if;

— bank 0 empty flags
if (waita2 = '1') and (cache_address_a_temp = "111") and (current_bank_a = '0') then
empty_banks(1) <= '1';
elsif (next_word = '1') and (not (cache_address_a_temp = "111") and (current_bank_a = '1') then
empty_banks(1) <= '0';
end if;

end if;
end if;
end process banks;

access_to_sdram : process(clk, reset_n, next_word, cache_address_b_temp, direction, reset_cache, last_block_empty, last_block_filled, current_bank_b)
begin
if (reset_n = '0') or (reset_cache = '1') then
cache_enable_b <= '0';
last_block_filled <= '1';
else if clk'delay and clk = '1' then
cache_enable_b <= '1';
end if;
if (next_word = '1') then
if cache_address_b_temp = "111" then
last_block_filled <= current_bank_b;
end if;
end if;
end if;
end process access_to_sdram;

cache.address_a <= current_bank_a & cache.address_a_temp;
cache.address_b <= current_bank_b & cache.address_b_temp when startup = '0' else
'0' & cache.address_b_temp;

— replaced 0006-02-22
bank_full <= bank_full_int;
bank_empty_int <= full_banks(0) OR full_banks(1);
cache_empty <= empty_banks(0) AND empty_banks(1);

— replaced 0006-02-22
bank_empty <= bank_empty_int;
bank_empty_int <= full_banks(0) OR empty_banks(1);
bank_empty <= bank_empty_int;

— replaced 0006-05-01
empty_int <= '1' when ((bank_full_int = '1') and (ser_empty = '1')) or
((not (cache.address_a_temp = "000") and (ser_empty = '1'))
else '0';
temp.cache.empty_out <= cache_empty;
cache.write_enable_b <= next_word when (direction = '1') else
'0';
end behavioral;

C.4.2 Cache Controller - VHDL Testbenches

Write operation
ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

COMPONENT cache_control
PORT(
    clk : IN std_logic;
    reset_n : IN std_logic;
    par.full : IN std_logic;
    ser.empty : IN std_logic;
    next.word : IN std_logic;
    direction : IN std_logic;
    reset.cache : IN std_logic;
    startup : IN std_logic;
    bank.full : OUT std_logic;
    bank.empty : OUT std_logic;
    cache.address.a : OUT std_logic_vector(3 downto 0);
    cache.enable.a : OUT std_logic;
    cache.write.enable.a : OUT std_logic;
    cache.address.b : OUT std_logic_vector(3 downto 0);
    cache.enable.b : OUT std_logic;
    cache.write.enable.b : OUT std_logic;
    write_ser.n : OUT std_logic;
    temp_cache_empty.out : OUT std_logic)
);
END COMPONENT;

SIGNAL clk : std_logic := '0';
SIGNAL reset_n : std_logic := '0';
SIGNAL par.full : std_logic := '0';
SIGNAL ser.empty : std_logic := '0';
SIGNAL next.word : std_logic := '0';
SIGNAL bank.full : std_logic;
SIGNAL bank.empty : std_logic;
SIGNAL cache.address.a : std_logic_vector(3 downto 0);
SIGNAL cache.enable.a : std_logic;
SIGNAL cache.write.enable.a : std_logic;
SIGNAL cache.address.b : std_logic_vector(3 downto 0);
SIGNAL cache.enable.b : std_logic;
SIGNAL cache.write.enable.b : std_logic;
SIGNAL direction : std_logic := '0';
SIGNAL write_ser.n : std_logic := '0';
SIGNAL temp_cache_empty.out : std_logic;
SIGNAL reset.cache : std_logic := '0';
constant cycle_time : time := 20 ns;
signal i : integer := 0;
signal j : integer := 0;
signal k : integer := 0;
BEGIN
uut: cache_control PORT MAP(
    clk => clk,
    reset_n => reset_n,
    par.full => par.full,
    ser.empty => ser.empty,
    next.word => next.word,
    bank.full => bank.full,
    bank.empty => bank.empty,
    cache.address.a => cache.address.a,
    cache.enable.a => cache.enable.a,
    cache.write.enable.a => cache.write.enable.a,
    cache.address.b => cache.address.b,
    cache.enable.b => cache.enable.b,
    cache.write.enable.b => cache.write.enable.b,
    direction => direction,
    write_ser.n => write_ser.n,
    temp_cache_empty.out => temp_cache_empty.out,
    reset.cache => reset.cache,
    startup => startup
);
BEGIN

*** Test Bench - User Defined Section ***
clocking : process
begin
```vhdl
clk <= not(clk);
wait for 0.5*cycle;
end process;

full_proc : process
begin
i <= 0;
while i < 8 loop
i <= i + 1;
wait for 10.5*cycle;
par_full <= '1';
wait for 3*cycle;
par_full <= '0';
wait for 0.5 * cycle;
end loop;
wait for 1*cycle;
while i < 16 loop
i <= i + 1;
wait for 10.5*cycle;
par_full <= '1';
wait for 3*cycle;
par_full <= '0';
wait for 0.5 * cycle;
end loop;
wait for 16*cycle;
while i < 24 loop
i <= i + 1;
wait for 10.5*cycle;
par_full <= '1';
wait for 3*cycle;
par_full <= '0';
wait for 0.5 * cycle;
end loop;
wait for 16*cycle;
while i < 32 loop
i <= i + 1;
wait for 10.5*cycle;
par_full <= '1';
wait for 3*cycle;
par_full <= '0';
wait for 0.5 * cycle;
end loop;
wait;
end process full_proc;

empty_proc : process
begin
wait until direction 'event and direction = '1';
wait for cycle;
ser.empty <= '1';
k <= 0;
while k < 16 loop
k <= k + 1;
wait until write_ser_n = '0';
ser.empty <= '0';
wait for 10*cycle;
ser.empty <= '1';
end loop;
end process empty_proc;

sdram_operation : process
begin
wait until i = 8;
wait until bank_full = '1';
wait for 12*cycle; -- was 8
next_word <= '1';
wait for 8*cycle;
next_word <= '0';
end process;
```

wait until \( i = 16 \);
wait until bank.full = '1';
wait for 12\*cycle;
next_word <= '1';
wait for 8\*cycle;
next_word <= '0';
wait until \( i = 24 \);
wait until bank.full = '1';
wait for 12\*cycle;
next_word <= '1';
wait for 8\*cycle;
next_word <= '0';

\( j <= 0; \)
wait until \( i = 32 \);
wait until bank.full = '1';
wait for 12\*cycle;
while \( j < 8 \) loop
\( j <= j + 1; \)
next_word <= '1';
wait for cycle;
next_word <= '0';
wait for cycle;
end loop;
wait until direction 'event and direction = '1';
wait for 10\*cycle;
next_word <= '1';
wait for 8\*cycle;
next_word <= '0';
wait until \( k = 4 \);
wait for 0.5\*cycle;
next_word <= '1';
wait for 8\*cycle;
next_word <= '0';
wait;
end process sdram_operation;

sdram_reading : process
begin
wait;
end process sdram_reading;

force-reset : process
begin
wait for 2\*cycle;
reset_cache <= '1';
wait for 2\*cycle;
reset_cache <= '0';
wait for 1000 ns;
reset.cache <= '1';
wait for 2\*cycle;
reset_cache <= '0';
wait for 2\*cycle;
direction <= '1'; — indicated READ from sdram
wait;
end process force-reset;

write_sim : process
begin
wait;
end process write_sim;

tb : PROCESS
BEGIN
reset_n <= '1' after 32 ns;
wait; — will wait forever
END PROCESS;
— *** End Test Bench — User Defined Section ***
CHAPTER C — Code Listings

Cycled write operation

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

COMPONENT cache_control
PORT(
clk : IN std.logic;
reset_n : IN std.logic;
par_full : IN std.logic;
ser_empty : IN std.logic;
next_word : IN std.logic;
direction : IN std.logic;
reset_cache : IN std.logic;
startup : IN std.logic;
bank_full : OUT std.logic;
bank_empty : OUT std.logic;
write_scr_n : OUT std_logic;
temp_cache_empty_out : OUT std_logic;

cache_address_a : std_logic_vector (3 downto 0);
cache_enable_a : OUT std_logic;
cache_write_enable_a : OUT std_logic;

cache_address_b : std_logic_vector (3 downto 0);
cache_enable_b : OUT std_logic;
cache_write_enable_b : OUT std_logic;
write_scr_n : OUT std_logic;
temp_cache_empty_out : OUT std_logic;
reset_cache : std_logic;
)
END COMPONENT;

SIGNAL clk : std_logic := '0';
SIGNAL reset_n : std_logic := '0';
SIGNAL par_full : std_logic := '0';
SIGNAL ser_empty : std_logic := '0';
SIGNAL next_word : std_logic := '0';
SIGNAL bank_full : std_logic;
SIGNAL bank_empty : std_logic;
SIGNAL cache_address_a : std_logic_vector (3 downto 0);
SIGNAL cache_enable_a : std_logic;
SIGNAL cache_write_enable_a : std_logic;
SIGNAL cache_address_b : std_logic_vector (3 downto 0);
SIGNAL cache_enable_b : std_logic;
SIGNAL cache_write_enable_b : std_logic;
SIGNAL direction : std_logic := '0';
SIGNAL write_scr_n : std_logic := '0';
SIGNAL reset_cache : std_logic := '0';
SIGNAL startup : std_logic := '0';
constant cycle time : time := 20 ns;
signal i : integer := 0;
signal j : integer := 0;
signal k : integer := 0;
BEGIN
uut: cache_control PORT MAP(
clk => clk,
reset_n => reset_n,
par_full => par_full,
ser_empty => ser_empty,
next_word => next_word,
bank_full => bank_full,
bank_empty => bank_empty,
cache_address_a => cache_address_a,
cache_enable_a => cache_enable_a,
cache_write_enable_a => cache_write_enable_a,
cache_address_b => cache_address_b,
cache_enable_b => cache_enable_b,
cache_write_enable_b => cache_write_enable_b,
direction => direction,
write_scr_n => write_scr_n,
temp_cache_empty_out => temp_cache_empty_out,
reset_cache => reset_cache,
);
--- Test Bench - User Defined Section ---
clocking : process
begin
  clk <= not(clk);
  wait for 0.5*cycle;
end process;

full_proc : process
begin
  i <= 0;
  while i < 8 loop
    i <= i + 1;
    wait for 10.5*cycle;
    par_full <= '1';
    wait for 3*cycle;
    par_full <= '0';
    wait for 0.5 * cycle;
  end loop;

  wait for 1*cycle;  -- was 16
  while i < 16 loop
    i <= i + 1;
    wait for 10.5*cycle;
    par_full <= '1';
    wait for 3*cycle;
    par_full <= '0';
    wait for 0.5 * cycle;
  end loop;

  wait for 16*cycle;
  while i < 24 loop
    i <= i + 1;
    wait for 10.5*cycle;
    par_full <= '1';
    wait for 3*cycle;
    par_full <= '0';
    wait for 0.5 * cycle;
  end loop;

  wait for 16*cycle;
  while i < 32 loop
    i <= i + 1;
    wait for 10.5*cycle;
    par_full <= '1';
    wait for 3*cycle;
    par_full <= '0';
    wait for 0.5 * cycle;
  end loop;

  wait;
end process full_proc;

empty_proc : process
begin
  wait until direction 'event and direction = '1';
  wait for cycle;
  ser_empty <= '1';
  k <= 0;
  while k < 16 loop
    k <= k + 1;
    wait until write_ser_n = '0';  -- cache ctrl writes so serializer
    ser_empty <= '0';  -- indicate serializer is busy
    wait for 10*cycle;
    ser_empty <= '1';  -- allow to finish
    end loop;
end process empty_proc;

sdram_operation : process
begin
wait until i = 16;
wait until par.full = '1';
wait until par.full = '0';
wait for 2*cycle;
next_word <= '1';
wait for 8*cycle;
next_word <= '0';
wait until i = 24;
wait until bank.full = '1';
wait for 12*cycle;
next_word <= '1';
wait for 8*cycle;
next_word <= '0';
j<=0;
wait until i = 32;
wait until bank.full = '1';
wait for 12*cycle;
while j<8 loop
  j <= j + 1;
  next_word <= '1';
  wait for cycle;
  next_word <= '0';
  wait for cycle;
end loop;
wait until direction 'event and direction = '1';
wait for 10*cycle;
next_word <= '1';
wait for 8*cycle;
next_word <= '0';
wait until kw4;
wait for 0.5*cycle;
next_word <= '1';
wait for 8*cycle;
next_word <= '0';
wait;
end process sdram_operation;
sdram_reading : process
begin
  wait;
end process sdram_reading;
force_reset : process
begin
  wait for 2*cycle;
  reset.cache <= '1';
  wait for 2*cycle;
  reset.cache <= '0';
  wait for 11000 ns;
  reset.cache <= '1';
  wait for 2*cycle;
  reset.cache <= '0';
  wait for 2*cycle;
  direction <= '1'; — indicated READ from sdram
  wait;
end process force_reset;
write_sim : process
begin
  wait;
end process write_sim;
tb : PROCESS
BEGIN
reset_n <= '1' after 33 ns;
wait; — will wait forever
END PROCESS;
— *** End Test Bench — User Defined Section ***
Cycled read operation

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

COMPONENT cache_control

PORT(
  clk : IN std_logic;
  reset_n : IN std_logic;
  par_full : IN std_logic;
  ser_empty : IN std_logic;
  next_word : IN std_logic;
  direction : IN std_logic;
  reset_cache : IN std_logic;
  startup : IN std_logic;
  bank_full : OUT std_logic;
  bank_empty : OUT std_logic;
  cache_address_a : OUT std_logic_vector(3 downto 0);
  cache_enable_a : OUT std_logic;
  cache_write_enable_a : OUT std_logic;
  cache_address_b : OUT std_logic_vector(3 downto 0);
  cache_enable_b : OUT std_logic;
  cache_write_enable_b : OUT std_logic;
  write_scr_n : OUT std_logic;
  temp_cache_empty_out : OUT std_logic
);
END COMPONENT;

SIGNAL clk : std_logic := '0';
SIGNAL reset_n : std_logic := '0';
SIGNAL par_full : std_logic := '0';
SIGNAL ser_empty : std_logic := '0';
SIGNAL next_word : std_logic := '0';
SIGNAL bank_full : std_logic;
SIGNAL bank_empty : std_logic;
SIGNAL cache_address_a : std_logic_vector(3 downto 0);
SIGNAL cache_enable_a : std_logic;
SIGNAL cache_write_enable_a : std_logic;
SIGNAL cache_address_b : std_logic_vector(3 downto 0);
SIGNAL cache_enable_b : std_logic;
SIGNAL cache_write_enable_b : std_logic;
SIGNAL direction : std_logic := '0';
SIGNAL write_scr_n : std_logic := '1';
SIGNAL temp_cache_empty_out : std_logic;
SIGNAL reset_cache : std_logic := '0';
SIGNAL startup : std_logic := '0';

constant cycle : time := 20 ns;
signal i : integer := 0;
signal j : integer := 0;
signal k : integer := 0;

BEGIN
uut: cache_control PORT MAP(
  clk => clk,
  reset_n => reset_n,
  par_full => par_full,
  ser_empty => ser_empty,
  next_word => next_word,
  bank_full => bank_full,
  bank_empty => bank_empty,
  cache_address_a => cache_address_a,
  cache_enable_a => cache_enable_a,
  cache_write_enable_a => cache_write_enable_a,
  cache_address_b => cache_address_b,
  cache_enable_b => cache_enable_b,
  cache_write_enable_b => cache_write_enable_b,
  direction => direction,
  write_scr_n => write_scr_n,
  temp_cache_empty_out => temp_cache_empty_out,
  reset_cache => reset_cache,
  startup => startup
);
cache_write.enable_b => cache_write_enable_b,  
direction =>> direction,  
write_ser.n => write_ser_n,  
temp_cache_empty_out => temp_cache_empty_out,  
reset_cache => reset_cache,  
startup => startup)

--- Test Bench — User Defined Section ---
clocking : process
begin  
clk <= not(clk);
wait for 0.5 • cycle;
end process;
full_proc : process
begin
i <= 0;
while i < 8 loop
i <= i + 1;
wait for 10.5 • cycle;
par_full <= '1';
wait for 3 • cycle;
par_full <= '0';
wait for 0.5 • cycle;
end loop;
wait for 1 • cycle;
end
while i < 16 loop
i <= i + 1;
wait for 10.5 • cycle;
par_full <= '1';
wait for 3 • cycle;
par_full <= '0';
wait for 0.5 • cycle;
end loop;
wait for 16 • cycle;
end
while i < 24 loop
i <= i + 1;
wait for 10.5 • cycle;
par_full <= '1';
wait for 3 • cycle;
par_full <= '0';
wait for 0.5 • cycle;
end loop;
wait for 16 • cycle;
end
while i < 32 loop
i <= i + 1;
wait for 10.5 • cycle;
par_full <= '1';
wait for 3 • cycle;
par_full <= '0';
wait for 0.5 • cycle;
end loop;
wait;
end process full Proc;
empty_proc : process
begin
wait until direction 'event and direction = '1';
wait for cycle;
wait until next_word = '1';
wait until next_word = '1';
wait for 12 • cycle;
ser_empty <= '1';
k <= 0;
while k < 64 loop
k <= k + 1;
wait until write_ser.n = '0';—cache ctrl writes to serialiser
ser_empty <= '0';—indicate serialiser is busy
end

161
wait for 10\times \text{cycle};
ser_empty <= '1';
end loop;

end process empty_proc;

sdram_operation : process
begin
wait until i = 8;
wait until bank_full = '1';
wait for 12\times \text{cycle};
next_word <= '1';
wait for 8\times \text{cycle};
next_word <= '0';
wait until i = 16;
wait until bank_full = '1';
wait for 12\times \text{cycle};
next_word <= '1';
wait for 8\times \text{cycle};
next_word <= '0';
wait until i = 24;
wait until bank_full = '1';
wait for 12\times \text{cycle};
next_word <= '1';
wait for 8\times \text{cycle};
next_word <= '0';
j<=0;
wait until i = 32;
wait until bank_full = '1';
wait for 12\times \text{cycle};
while j<8 loop
\quad j <= j + 1;
\quad next_word <= '1';
\quad wait for cycle;
\quad next_word <= '0';
\quad wait for cycle;
end loop;
wait until direction 'event and direction = '1';
wait for 10\times \text{cycle};
next_word <= '1';
wait for 8\times \text{cycle};
next_word <= '0';
wait for 10\times \text{cycle};
wait for 0.5\times \text{cycle};
next_word <= '1';
wait for 8\times \text{cycle};
next_word <= '0';
wait until bank_empty = '1';
wait for 2.5\times \text{cycle};
wait for 0.5\times \text{cycle};
next_word <= '1';
wait for 8\times \text{cycle};
next_word <= '0';
wait until bank_empty = '1';
wait for 2.5\times \text{cycle};
wait for 0.5\times \text{cycle};
next_word <= '1';
wait for 8\times \text{cycle};
next_word <= '0';
wait;
end process sdram_operation;

sdram_reading : process
begin
wait;

\quad-\text{allow to finish}
\quad-\text{ready again}
end process sdram_reading;

force_reset : process
begin
  wait for 2\times cycle;
  reset_cache \equiv '1';
  wait for 2\times cycle;
  reset_cache \equiv '0';
  wait for 11000 ns;
  reset_cache \equiv '1';
  wait for 2\times cycle;
  reset_cache \equiv '0';
  wait for 2\times cycle;
  direction \equiv '1'; -- indicated READ from sdram
  wait;
end process force_reset;

write_sim : process
begin
  wait;
end process write_sim;

tb : PROCESS
BEGIN
  reset_n \equiv '1' after 33 ns;
  wait; -- will wait forever
END PROCESS;

--- End Test Bench - User Defined Section ---

END;
C.5 Command Controller

C.5.1 Command Controller - VHDL Code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity cmd_controller is
  Port ( clock, count_out : out std_logic_vector(4 downto 0);   -- just for testing purposes
         current_state_out : out std_logic_vector(4 downto 0);   -- just for testing purposes
         cmd_temp_out : out std_logic_vector(3 downto 0);   -- received command through port A
         clk : in std_logic;
         reset_n : in std_logic;
         cmd_pa : in std_logic;
         cmd_in : in std_logic_vector(3 downto 0);   -- received command through port A
         wr_pa_n : in std_logic;
         scLram : in std_logic;
         data_pa_in : in std_logic_vector(7 downto 0);   -- indicates memory is being accessed
         elk : in std_logic;
         just_for_testing_purp. : in std_logic;
         just_for_testing_purposes : in std_logic;
         cmd_pa_n : in std_logic;
         cmd_in : in std_logic_vector(3 downto 0);
         wr_pa_n : in std_logic;
         scLram : in std_logic;
         data_pa_in : in std_logic_vector(7 downto 0);
         --misc control signals
         ser_advance : out std_logic;  -- serialiser manual advance
         ser_enable : out std_logic;  -- serialiser enable
         par_enable : out std_logic;  -- paralleliser enable
         single_err : in std_logic;  -- single error indicated by EDAC
         multi_err : in std_logic;  -- multiple errors indicated by EDAC
         --address controller interface
         address_set_pulse : out std_logic;  -- address unit set pulse
         address_load_done : in std_logic;  -- address loaded all bytes of address
         reset_address : out std_logic;  -- reset of address unit (only one at the moment)
         address : in std_logic_vector(13 downto 0);  -- planned: B cach for simultaneous writes by part a + b
         addr_byte : out std_logic_vector(7 downto 0);  -- current byte to be loaded into address register
         address_done : in std_logic;  -- address counter reached target address
         address_out : out std_logic_vector(13 downto 0);  -- multiplexed output (between addr and mma data)
         --cache controller interface
         cache_empty : in std_logic;  -- indicates entire cache is empty
         bank_full : in std_logic;  -- bank full in cache
         bank_empty : in std_logic;  -- bank empty in cache
         cache_direction : out std_logic;  -- direction of data flow in cache
         reset_cache : out std_logic;  -- completely reset cache addresses
         next_word : out std_logic;  -- synchronizes address and cache
         ser_empty : in std_logic;  -- used for startup condition on read
         startup : out std_logic;  -- used for startup condition on read
         address_in : in std_logic_vector(13 downto 0);
         --sdram controller interface
         mode_r_set : out std_logic;  -- mode register set command
         r_w : out std_logic;  -- read / write ('1' = read)
         abort : out std_logic;  -- abort burst data transfers
         ref_req_out : out std_logic;  -- sdram refresh request command
         precharge : out std_logic;  -- sdram precharge command
         --refresh timer interface
         ref_req_in : in std_logic;  -- input from refresh counter
         ref_ack_timer : out std_logic;  -- timer refresh acknowledge (out)
         ref_ack : in std_logic;  -- timer refresh acknowledge (in)
         edram_active : out std_logic;  -- output indicates edram is active (data retention)
         edram_off : out std_logic;  -- edram is in non-data retention mode
         edram_active : out std_logic;
         edram_off : out std_logic;
     );
end cmd_controller;

architecture behavioral of cmd_controller is
  signal cmd_int : std_logic_vector(3 downto 0);
  signal current_state, next_state : std_logic_vector(4 downto 0);
  signal old_state, next_old_state : std_logic_vector(4 downto 0);
  signal wr_pa_n : in std_logic;
  signal address_set_pulse, enable : std_logic;
  signal startup_int : std_logic;
end behavioral;
```

Stellenbosch University https://scholar.sun.ac.za
CHAPTER C — CODE LISTINGS

165

```vhdl
signal start.cmb : std_logic;
signal mode_reg.cmb : std_logic;
signal r,w.cmb : std_logic;
signal abort.cmb : std_logic;
signal address.set_pulse.enable : std_logic;
signal set.pulse.int : std_logic;
--signal address.set_pulse.enable cmb : std_logic;

begin

end
```

---

```vhdl
begin
get.cmd : process (clk, reset.n, cmd.pa, cmd.in, cmd.int, wr.n.prev, wr.pa.n)
begin
if reset.n = '0' then

if clk 'event and clk = '1' then
if (cmd.pa = '1') and (wr.n.prev = '1') and (wr.pa.n = '0') then

```

---

```vhdl
-- cmd.pa can be used to start port b read sequence
```

---

```vhdl
elsif addr.done = '1' then
```

---

```vhdl
elsif clk 'event and clk = '1' then
```

---

```vhdl
begin
```

---

```vhdl
-- cmd.int <= (others => '0');
```

---

```vhdl
elsif addr.done = '1' then
```

---

```vhdl
if (cmd.pa = '1') and (wr.n.prev = '1') and (wr.pa.n = '0') then
```

---

```vhdl
cmd.int <= (others => '0');
```

---

```vhdl
end
```

---

```vhdl
begin
```

---

```vhdl
if reset.n = '0' then
```

---

```vhdl
```
else if cmd.int = c.set.adr.a then
    cmd.int <= (others => '0');
end if;
end process get.cmd;

sdram.on.off : process(clk, reset.n, cmd.int)
begin
if reset.n = '0' then
    sdram.active <= '0';
    sdram.off <= '1';
else if clk'event and clk = '1' then
    sdram.active <= '1';
    sdram.off <= '0';
end if;
end process sdram.on.off;

get_wr : process(clk, reset.n, wr.pa.n)
begin
if reset.n = '0' then
    wr.pa.n_prev <= '1';
else if clk'event and clk = '1' then
    wr.pa.n_prev <= wr.pa.n;
end if;
end process get_wr;

pulsegen : process(clk, reset.n, wr_pa.n)
begin
if reset.n = '0' then
    set_pulse_int <= '0';
elsif clk'event and clk = '1' then
    if wr_pa.n = '0' and wr_pa.n_prev = '1' then
        set_pulse_int <= '1';
    else
        set_pulse_int <= '0';
    end if;
end if;
end process pulsegen;

state_comb : process(current.state, sel.ram, cmd.int, bank_full, clock_count, adr.done, ref.req_int, old.state,
                     start_up, bank.empty, wait_count, address.load_done, ser.empty, cache.empty)
begin
next_old-state <= old_state;
case current.state is
  when s_no_action =>
    if sel.ram = '1' then
      next_state <= s.prech_all;
    else
      next_state <= s.no_action;
    end if;
  when s_mre_l =>
    next_state <= s.mre_wait_1;
  when s_mre_wait_1 =>
    next_state <= s.mre_wait_2;
  when s_mre_wait_2 =>
    next_state <= s.idle;
  when s_idle =>
    case cmd.int is
      when clock_count = 8 then
        next_state <= s.idle;
    end case;
  when s_wr =>
    if bank.full = '1' then
      next_state <= s.wr_burst_1;
    else
      next_state <= s.wr_idle;
    end if;
  when s_wr_burst_1 =>
    next_state <= s_wr_burst_2;
  when s_wr_burst_2 =>
    next_state <= s.wr_burst_3;
  when s.wr_burst_3 =>
    if clock_count = 8 then
      next_state <= s.wr_idle;
    else
      next_state <= s.wr_burst_4;
    end if;
  when s_rd =>
    if bank_empty = '0' then
      next_state <= s.rd_idle;
    else
      next_state <= s.rd_wait;
    end if;
end process state_comb;
next-state <= s_rd_burst_1;
end if;
when s_rd_burst_1 => next-state <= s_rd_burst_2;
when s_rd_burst_2 => next-state <= s_rd_wait;
when s_rd_wait => if wait_count = 2 then
next-state <= s_rd_burst_3;
else
next-state <= s_rd_wait;
end if;
when s_rd_burst_3 => if clock_count = 10 then
if startup_int = '1' then
next-state <= s_adv_idle;
else
next-state <= s_rd_idle;
end if;
end if;
when s_adv_idle => if ser_empty = '0' then next-state <= s_adv_l;
else next-state <= s_adv_idle;
end if;
when s_adv_l => next-state <= s_adv_2;
when s_adv_2 => next_state <= s_rd_idle;
when s_rd_idle => if ref_req_int = '1' then next_state <= s_refr_wait;
else next_state <= s_rd_idle;
end if;
when s_adv_2 => next_state <= s_rd_idle;
when s_wr_burst_2 => if bank_empty = '1' then next_state <= s_rd_burst_1;
else next_state <= s_wr_burst_2;
end if;
when s_adv_l => next_state <= s_adv_idle;
end if;
end case;
end process state_comb;

clock-count : process(reset_n, current_state, clock_count) begin
if reset_n = '0' then
clock_count <= 0;
else if clk'event and clk = '1' then
if (current_state = s_wr_burst_2) or
(current_state = s_wr_burst_3) or
(current_state = s_rd_wait) or
(current_state = s_rd_burst_3) then
clock_count <= clock_count + 1;
else
clock_count <= 0;
end if;
end if;
end process clock_count;

wait_proc : process(reset_n, current_state, wait_count) begin
if reset_n = '0' then
wait_count <= 0;
else if clk'event and clk = '1' then
if (current_state = s_rd_wait) or
((current_state = s_refr_1) or
(current_state = s_refr_2) or
(current_state = s_refr_wait) or
(current_state = s_prech_all)) and
(wait_count = 0) then

wait-count <= wait-count + 1;
else
  wait-count <= 0;
end if;
end if;
end process;

end process generate_pulse;

next_state.clk : process(clk, reset_n , next_state, next_old_state)
begin
  if reset_n = '0' then
    current_state <= s_no_action;
    ser_advance <= '0';
    ser_enable <= '0';
    par_enable <= '0';
    cache_direction <= '0';
    reset.cache <= '1';
    reset.address <= '1';
    start <= '0';
    r.w <= '0';
    abort <= '0';
    ref.req.out <= '0';
    next.word <= '0';
    startup.int <= '0';
    old_state <= s_idle;
  elsif clk'event and clk = '1' then
    current_state <= next_state;
    ser_advance <= ser_advance.cmb;
    ser_enable <= ser_enable.cmb;
    par_enable <= par.enable.cmb;
    cache_direction <= cache_directioncmb;
    reset.cache <= reset.cache.cmb;
    reset.address <= reset.address.cmb;
    start <= start.cmb;
    r.w <= r.w cmb;
    abort <= '0';
    ref.req.out <= '0';
    next.word <= next.word.cmb;
    mode.r.set <= mode.r.set.cmb;
    old_state <= next.old_state;
  end if
  if next_state = s_rd_1 then
    startup.int <= '1';
  elsif next_state = s_adv_2 then
    startup.int <= '0';
  end if;
end process next_state.clk;

-- temporary combinatorial control signals
-- misc
ser.advance.cmb <= '1' when (current_state = s_adv_1) or (current_state = s_adv_2) else '0';
ser_enable.cmb <= cache.direction.cmb;
par.enable.cmb <= '1' when (cmd_int = c_wr_a) else '0';
-- cache
cache.direction.cmb <= '0' when (cmd_int = c_wr_a) else '1';
reset.cache.cmb <= '1' when((current_state = s_prech all) and (wait_count = 0)) or (current_state = s_idle ) or (current_state = s_no_action) else '0';
next.word.cmb <= '1' when (current_state = s_wr_burst_3) or (current_state = s_rd_burst_3) else '0';
-- address
addres_s.set.pulse.cmb <= '0';
reset.address.cmb <= '0';
address_set.pulse.enable <= '1' when current_state = s_adv_1 else '0';
-- sdram
start.cmb <= '1' when (current_state = s_wr_burst_1) or (current_state = s_rd_burst_1) or (current_state = s_mrs_1) else '0';
mode.set.cmb <= '1' when (current_state = s_mrs_1) else '0';
mr_adr <= '1' when (current_state = s_mrs_1) or (current_state = s_mrs.wait_1) or (current_state = s_mrs.wait_2) else -- mode register selected '0';
address_selected <= '0';
r.w.cmb <= '1' when (cmd_int = c rd_b) else '0';
C.5.2 Command Controller - VHDL Testbench

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

COMPONENT cmd_controller
PORT(
clock_count_out : OUT std_logic_vector(4 downto 0);
current_state_out : OUT std_logic_vector(4 downto 0);
cmd_temp_out : out std_logic_vector(3 downto 0);
clk : IN std_logic;
reset_n : IN std_logic;
cmd_pa : IN std_logic;
cmd_in : IN std_logic_vector(3 downto 0);
wr_pa_n : IN std_logic;
sel_ram : IN std_logic;
data_pa_in : IN std_logic_vector(7 downto 0);
single_err : IN std_logic;
multi_err : IN std_logic;
address_load_done : IN std_logic;
adr_done : IN std_logic;
address_out : OUT std_logic_vector(13 downto 0);
adr_byte : OUT std_logic_vector(7 downto 0);
cache.empty : IN std_logic;
bank.full : IN std_logic;
bank.empty : IN std_logic;
ser_advance : OUT std_logic;
ser_enable : OUT std_logic;
par_enable : OUT std_logic;
address_set_pulse : OUT std_logic;
reset_address : OUT std_logic;
next_word : OUT std_logic;
ser_empty : in std_logic;
);

-- abort cmb <= '0';
-- ref_req cmb <= '0';
-- combinatorial logic
adr_byte <= data_pa_in when address_set_pulse_enable = '1' else (others => '0');
address_set_pulse <= std_logic_vector(clock_count);  
address_out <= "0000" & mode_register_value when mrs_adr = '1' else --MRS
precharge_value when current_state = s_prech_all else --PRECHARGE
address_in;

end behavioral;

begin

uut : cmd_controller PORT MAP(
clock.count.out => clock.count.out,
current.state.out => current.state.out,
cmd.temp.out => cmd.temp.out,
clk => clk,
reset.n => reset.n,
cmd_pa => cmd_pa,
cmd_in => cmd_in,
wr_pa.n => wr_pa.n,
set_ram => sel_ram,
data_pa.in => data_pa.in,
ser_advance => ser_advance,
ser.enable => ser.enable,
par.enable => par_enable,
single.err => single.err,
multi.err => multi.err,
address.set.pulse => address.set.pulse,
address.load.done => address.load.done,
reset.address => reset.address;

end
adrlbyte => adrlbyte,
adrldone => adrldone,
adre.xout => adre.xout,
adre.xin => adre.xin,
cach3 empt3 => cach3 empt3,
bank_full => bank_full,
bank_empty => bank_empty,
cach3 direction => cach3 direction,
reset_cach3 => reset_cach3,
nexlword => nexlword,
ser empt3 => ser empt3,
start up => start up,
start => start,
mod3 r set => mode r set,
r_w => r_w,
short => short,
ref req in => ref req in,
ref req out => ref req out,
precharge => precharge,
ref ack timer => ref ack timer,
ref ack sdram => ref ack sdram,
sdram active => sdram active,
sdram off => sdram off
);

*** Test Bench - User Defined Section ***
resetting : process
begin
  wait for 2.1 cycle;
  reset_n <= '1';
  wait for cycle;
  sel ram <= '1';
  wait until start = '1';
  wait for 5 cycle;
  wait until current state out = "00101";
  wait for 3 cycle;
  wait until current state out = "00101";
  wait until bank full = '1';
  wait until current state out = "00101"; -- wr idle
  wait for cycle;
  adrl done <= '1';
  wait for cycle;
  adrl done <= '0';
  wait until (ser advance = '1');
  wait until (current state out = "10001") and (ser advance = '1');
  wait until current state out = "10000";
  wait for 10 cycle;
  adrl done <= '1';
  wait for 3 cycle;
  cach3 empt3 <= '1';
  wait for 2 cycle;
  ser empt3 <= '1';
  wait for 2 cycle;
  ser empt3 <= '0';
  adrl done <= '0';
  wait;
end process resetting;
clocking : process
begin
  clk <= not(clk);
  wait for 0.5 cycle;
end process clocking;
cmd : process
begin
  wait until sel ram = '1';
  -- allow enough time for mode register set
  wait for 15 cycle;
  -- start first command cycle
  cmd_pa <= '1';
  cmd_in <= "0111";
  wr_pa_n <= '0';
  wait for cycle;
  -- 1 cycle length of write pulse
  wr_pa_n <= '1';
reset input command (not necessary)

-- cmd : read port b

-- 1 cycle length of write pulse

-- cmd : set address port a

-- necessary, cmd_ctrl needs subsequent writes

-- necessary, cmd_ctrl needs subsequent writes

-- necessary, cmd_ctrl needs subsequent writes

-- necessary, cmd_ctrl needs subsequent writes

-- necessary, cmd_ctrl needs subsequent writes

-- necessary, cmd_ctrl needs subsequent writes

-- necessary, cmd_ctrl needs subsequent writes
bank_empty <= '1';
wait for 10+cycle;
bank_empty <= '0';
wait until addr_done = '1';
wait until addr_done = '0';
wait until cache_empty = '1';
bank_empty <= '1';
wait;
end process bank_empty_ctrl;

refreshing : process
begin
— 1st
wait until ref_req_out = '1';
wait for cycle;
ref_ack_sdram <= '1';
wait for cycle;
ref_ack_sdram <= '0';

— 2nd
wait until ref_req_out = '1';
wait for cycle;
ref_ack_sdram <= '1';
wait for cycle;
ref_ack_sdram <= '0';
wait;
end process;

end process;

tb PROCESS
BEGIN
wait; — will wait forever
END PROCESS;

*** End Test Bench — User Defined Section ***
C.6 EDAC Unit

C.6.1 EDAC Unit - VHDL Code

Main component

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity edac_top_2port is
  Port (clk : in std_logic;
         reset_n : in std_logic;
         check_word_in : in std_logic_vector (7 downto 0);
         mem_data_in : in std_logic_vector (63 downto 0);
         cpu_data_in : in std_logic_vector (63 downto 0);
         direction : in std_logic;
         single_error : out std_logic;
         multiple_errors : out std_logic
  );
end edac_top_2port;

architecture Behavioral of edac_top_2port is
  component syndrome_dec
    PORT(data_in : in std_logic_vector (63 downto 0);
         data_out : out std_logic_vector (63 downto 0);
         syndrome_in : in std_logic_vector (7 downto 0);
         single : out std_logic;
         double : out std_logic
    );
  end component;

  component syndrome_gen
    PORT(stored_cb : in std_logic_vector (7 downto 0);
         generated_cb : in std_logic_vector (7 downto 0);
         syndrome : out std_logic_vector (7 downto 0)
    );
  end component;

  component checkb_gen
    PORT(data_in : in std_logic_vector (63 downto 0);
         check_bits : out std_logic_vector (7 downto 0)
    );
  end component;

  signal data_int : std_logic_vector (63 downto 0);
  signal checkb_int : std_logic_vector (7 downto 0);
  signal syndrome_int : std_logic_vector (7 downto 0);
  signal corr_data : std_logic_vector (63 downto 0);
  signal cpu_data_in : std_logic_vector (63 downto 0);
  signal mem_data_out : std_logic_vector (63 downto 0);
  signal check_word_in : std_logic_vector (7 downto 0);
  signal check_word_out : std_logic_vector (7 downto 0);

  begin
    checkb_gen1 : checkb_gen
      port map (data_in => data_int,
```
CHAPTER C — CODE LISTINGS

check_bits => checkb_int);
syndrome_decl: syndrome_dec
port map(data_in => mem_data_in_i,
data_out => cpu_data_out_i,
syndrome_in => syndrome_int,
single => single_error,
double => multiple_errors);
syndrome_gen: syndrome_gen
port map(storedCb => check_word_in_i,
generatedCb => checkb_int,
syndrome => syndrome_int);
-- concurrent section
with direction select
data_int <= cpu_data_in_i when '0',
mem_data_in_i when others;
mem_data_out_i <= cpu_data_in_i;
check_word_out_i <= checkb_int;
-- final port links
mem_data_in_i <= mem_data_in;
mem_data_out <= mem_data_out_i;
cpu_data_in_i <= cpu_data_in;
check_word_in_i <= check_word_in;
check_word_out <= check_word_out_i;
end Behavioral;

EDAC - Checkbit Generator

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;
ENTITY checkb_gen IS
PORT(data_in : in std_logic_vector(63 downto 0);
check_bits : out std_logic_vector(7 downto 0)
); 
END checkb_gen;
ARCHITECTURE behavioural OF checkb_gen IS
-- constant/signal/component declarations
signal bits : std_logic_vector(9 downto 0);
BEGIN
check_bits(0) <= data_in(1) xor data_in(2) xor data_in(3) xor data_in(5) xor data_in(8) xor data_in(9) xor data_in(10) xor data_in(11) xor data_in(14) xor data_in(17) xor data_in(18) xor data_in(19) xor data_in(21) xor data_in(24) xor data_in(25) xor data_in(27) xor data_in(30) xor data_in(32) xor data_in(36) xor data_in(38) xor data_in(39) xor data_in(42) xor data_in(44) xor data_in(45) xor data_in(47) xor data_in(48) xor data_in(52) xor data_in(54) xor data_in(56) xor data_in(58) xor data_in(60) xor data_in(61) xor data_in(63);
check_bits(1) <= data_in(0) xor data_in(1) xor data_in(2) xor data_in(4) xor data_in(6) xor data_in(8) xor data_in(10) xor data_in(12) xor data_in(14) xor data_in(15) xor data_in(16) xor data_in(18) xor data_in(20) xor data_in(22) xor data_in(24) xor data_in(26) xor data_in(28) xor data_in(30) xor data_in(32) xor data_in(34) xor data_in(35) xor data_in(37) xor data_in(39) xor data_in(40) xor data_in(41) xor data_in(42) xor data_in(43) xor data_in(44) xor data_in(45) xor data_in(46) xor data_in(47) xor data_in(48) xor data_in(49) xor data_in(50) xor data_in(52) xor data_in(53) xor data_in(54) xor data_in(55) xor data_in(56) xor data_in(58) xor data_in(60) xor data_in(61) xor data_in(63);
check_bits(2) <= data_in(0) xor data_in(3) xor data_in(4) xor data_in(7) xor data_in(9) xor data_in(10) xor data_in(13) xor data_in(15) xor data_in(16) xor data_in(19) xor data_in(21) xor data_in(22) xor data_in(25) xor data_in(26) xor data_in(27) xor data_in(29) xor data_in(31) xor data_in(32) xor data_in(35) xor data_in(36) xor data_in(37) xor data_in(39) xor data_in(41) xor data_in(42) xor data_in(43) xor data_in(45) xor data_in(47) xor data_in(48) xor data_in(51) xor data_in(53) xor data_in(55) xor data_in(58) xor data_in(61) xor data_in(63);
check_bits(3) <= data_in(0) xor data_in(1) xor data_in(2) xor data_in(5) xor data_in(6) xor data_in(7) xor data_in(10) xor data_in(11) xor data_in(12) xor data_in(13) xor data_in(16) xor data_in(17) xor data_in(20) xor data_in(21) xor data_in(22) xor data_in(24) xor data_in(25) xor data_in(27) xor data_in(28) xor data_in(30) xor data_in(31) xor data_in(32) xor data_in(34) xor data_in(35) xor data_in(36) xor data_in(39) xor data_in(40) xor data_in(41) xor data_in(42) xor data_in(43) xor data_in(44) xor data_in(45) xor data_in(46) xor data_in(47) xor data_in(48) xor data_in(49) xor data_in(50) xor data_in(51) xor data_in(52) xor data_in(53) xor data_in(54) xor data_in(55) xor data_in(57) xor data_in(60) xor data_in(61) xor data_in(63);
EDAC - Syndrome Generator

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;

ENTITY syndrome_gen IS
  PORT(stored_cb : in std_logic_vector (7 downto 0);
       generated_cb : in std_logic_vector (7 downto 0);
       syndrome : out std_logic_vector (7 downto 0));
END syndrome_gen;

ARCHITECTURE behavioural OF syndrome_gen IS
  constant/ signal/ component declarations
BEGIN
  syndrome <= stored_cb xor generated_cb;
END behavioural;

EDAC - Syndrome Decoder

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

ENTITY syndrome_dec IS
  PORT(data_in : in std_logic_vector (63 downto 0);
       data_out : out std_logic_vector (63 downto 0);
       syndrome_in : in std_logic_vector (7 downto 0));
END syndrome_dec;

END behavioural;
ARCHITECTURE behavioural OF syndrome_dec IS
  -- constant/signal/component declarations
  signal mask : std_logic_vector (63 downto 0);
begin
  decoding : process (data_in, syndrome_in)
  begin
    data_in (0);
    when "11010011" => data_out <= data_in (63 downto 3) & (data_in(3) or '1') & data_in(2 downto 0);
    when "11010101" => data_out <= data_in (63 downto 4) & (data_in(4) or '1') & data_in(3 downto 0);
    when "01011011" => data_out <= (data_in(53) or '1') & data_in(52 downto 0);
    mask <= conv_std_logic_vector(0,64);
    case syndrome_in is
      when "00000000" => single <= '0'; double <= '0';
      when "00000001" => single <= '1'; double <= '0';
      when "00000100" => single <= '0'; double <= '0';
      when "00010000" => single <= '0'; double <= '0';
      when "00000000" => single <= '0'; double <= '0';
      when "00000000" => single <= '0'; double <= '0';
      when "00010000" => single <= '0'; double <= '0';
      when "00000000" => single <= '0'; double <= '0';
      when "11001110" => mask(0) <= '1'; single <= '1'; double <= '0';
      when "11010011" => mask(1) <= '1'; single <= '1'; double <= '0';
      when "11010101" => mask(2) <= '1'; single <= '1'; double <= '0';
      when "11010110" => mask(3) <= '1'; single <= '1'; double <= '0';
      when "11011001" => mask(4) <= '1'; single <= '1'; double <= '0';
      when "11011100" => mask(5) <= '1'; single <= '1'; double <= '0';
      when "00100011" => mask(6) <= '1'; single <= '1'; double <= '0';
      when "00100101" => mask(7) <= '1'; single <= '1'; double <= '0';
      when "00100110" => mask(8) <= '1'; single <= '1'; double <= '0';
      when "00101001" => mask(9) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(10) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(11) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(12) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(13) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(14) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(15) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(16) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(17) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(18) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(19) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(20) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(21) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(22) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(23) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(24) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(25) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(26) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(27) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(28) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(29) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(30) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(31) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(32) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(33) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(34) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(35) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(36) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(37) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(38) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(39) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(40) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(41) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(42) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(43) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(44) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(45) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(46) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(47) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(48) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(49) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(50) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(51) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(52) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(53) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(54) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(55) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(56) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(57) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(58) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(59) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(60) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(61) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(62) <= '1'; single <= '1'; double <= '0';
      when "00101010" => mask(63) <= '1'; single <= '1'; double <= '0';
    end case;
  end decoding;
  syndrome_dec;
END syndrome_dec;

when "00101100" => mask(13) <= '1';
double <= '0';
single <= '1';
when "00110001" => mask(14) <= '1';
double <= '0';
single <= '1';
when "00110100" => mask(15) <= '1';
double <= '0';
single <= '1';
when "00001110" => mask(16) <= '1';
double <= '0';
single <= '1';
when "00001011" => mask(17) <= '1';
double <= '0';
single <= '1';
when "00010011" => mask(18) <= '1';
double <= '0';
single <= '1';
when "00010101" => mask(19) <= '1';
double <= '0';
single <= '1';
when "00010110" => mask(20) <= '1';
double <= '0';
single <= '1';
when "00011001" => mask(21) <= '1';
double <= '0';
single <= '1';
when "00011010" => mask(22) <= '1';
double <= '0';
single <= '1';
when "00011100" => mask(23) <= '1';
double <= '0';
single <= '1';
when "11100011" => mask(24) <= '1';
double <= '0';
single <= '1';
when "11100101" => mask(25) <= '1';
double <= '0';
single <= '1';
when "11100110" => mask(26) <= '1';
double <= '0';
single <= '1';
when "11101001" => mask(27) <= '1';
double <= '0';
single <= '1';
when "11101010" => mask(28) <= '1';
double <= '0';
single <= '1';
when "11101100" => mask(29) <= '1';
double <= '0';
single <= '1';
when "01001111 11" => mask(30) <= '1';
double <= '0';
single <= '1';
when "01001010" => mask(31) <= '1';
double <= '0';
single <= '1';
when "01010010" => mask(32) <= '1';
double <= '0';
single <= '1';
when "01010100" => mask(33) <= '1';
double <= '0';
single <= '1';
when "01011000" => mask(34) <= '1';
double <= '0';
single <= '1';
when "01011101" => mask(35) <= '1';
double <= '0';
single <= '1';
when "01011111" => mask(36) <= '1';
double <= '0';
single <= '1';
when "01010010" => mask(37) <= '1';
double <= '0';
single <= '1';
when "01011111" => mask(38) <= '1';
double <= '0';
single <= '1';
double <= '0';
single <= '1';
when "10100100" => mask(41) <= '1';
double <= '0';
single <= '1';
when "10100111" => mask(42) <= '1';
double <= '0';
single <= '1';
when "10101000" => mask(43) <= '1';
double <= '0';
single <= '1';
when "10101011" => mask(44) <= '1';
double <= '0';
single <= '1';
when "10101101" => mask(45) <= '1';
double <= '0';
single <= '1';
when "10110000" => mask(46) <= '1';
double <= '0';
single <= '1';
when "10110101" => mask(47) <= '1';
double <= '0';
single <= '1';
when "10001111" => mask(48) <= '1';
when "10001010" => mask(50) <= '1';
double <= '0';
single <= '1';
when "10010010" => mask(51) <= '1';
double <= '0';
single <= '1';
when "10010111" => mask(52) <= '1';
double <= '0';
single <= '1';
when "10011000" => mask(53) <= '1';
double <= '0';
single <= '1';
when "10011011" => mask(54) <= '1';
when others => single <= '0';
end case;
C.6.2 EDAC - VHDL Testbench

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numéric_std.ALL;

ENTITY testbench IS
  END testbench;

ARCHITECTURE behavior OF testbench IS

COMPONENT edac_top_2port
  PORT(
    clk : IN std_logic;
    reset_n : IN std_logic;
    check_word_in : IN std_logic_vector(7 downto 0);
    mem_data_in : IN std_logic_vector(63 downto 0);
    cpu_data_in : IN std_logic_vector(63 downto 0);
    direction : IN std_logic;
    check_word_out : OUT std_logic_vector(7 downto 0);
    mem_data_out : OUT std_logic_vector(63 downto 0);
    cpu_data_out : OUT std_logic_vector(63 downto 0);
    single_error : OUT std_logic;
    multiple_errors : OUT std_logic
  );
END COMPONENT;

SIGNAL clk : std_logic := '0';
SIGNAL reset_n : std_logic := '0';
SIGNAL check_word_in : std_logic_vector(7 downto 0);
SIGNAL mem_data_in : std_logic_vector(63 downto 0);
SIGNAL cpu_data_in : std_logic_vector(63 downto 0);
SIGNAL direction : std_logic := '0';
SIGNAL single_error : std_logic;
SIGNAL multiple_errors : std_logic;

constant cycle : time := 20 ns;
BEGIN
  process
  begin
    while not( clk ) loop
      wait for 0.5 * cycle;
    end loop;
  end process;

  process
  begin
    reset_n <= '1' after 2.5 * cycle;
  end process;
```

--- **Test Bench - User Defined Section** ---

```vhdl
tb : PROCESS
BEGIN
  clk <= not( clk );
  wait for 0.5 * cycle;
END PROCESS;
```


wait;
END PROCESS;

cpu_input : PROCESS
BEGIN
  cpu_data-in <= "0000000000000000000000000000000000000000000000000000000000000000";
  wait;
END PROCESS;

dir : PROCESS
BEGIN
  wait for 3.5•cycle;
  direction <= '1';
  --normal testing
  mem_data-in <= "0000000000000000000000000000000000000000000000000000000000000000";
  check_word_in <= "00001100";
  --single error in stored data
  wait for 2•cycle;
  mem_data-in <= "0000000000000000000000000000000000000000000000000000000000000001";
  check_word_in <= "00001100";
  --single error in check word
  wait for 2•cycle;
  mem_data-in <= "0000000000000000000000000000000000000000000000000000000000000000";
  check_word_in <= "00001100";
  --multiple error in stored data
  wait for 2•cycle;
  mem_data-in <= "0000000000000000000000000000000000000000000000000000000000000001";
  check_word_in <= "00001100";
  --multiple error in check word
  wait for 2•cycle;
  mem_data-in <= "000000000000000000000000000000000000000000000010000000000000000";
  check_word_in <= "00001101";
  --single error in stored data + single error in check word
  wait for 2•cycle;
  mem_data-in <= "000000000000000000000000000000000000000000001000000000000000001";
  check_word_in <= "00001100";
  --3 errors in stored data
  wait for 2•cycle;
  mem_data-in <= "0000000000000000000000000000000000010000000000000100000000000001";
  check_word_in <= "00001100";
  --4 errors in stored data
  wait for 2•cycle;
  mem_data-in <= "000000000000000000000000000000000010000000000000100000000000001";
  check_word_in <= "00001100";
  --9 errors in stored data
  wait for 2•cycle;
  mem_data-in <= "000000000000000000000000000000000100000000000010000000000000001";
  check_word_in <= "00001100";
  --13 errors in stored data
END;

--- *** End Test Bench - User Defined Section ***

END;
C.7 Parallel Unit

C.7.1 Parallel Unit - VHDL Code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity par_dfftcst is
    Port (clk : in std_logic;
          reset_n : in std-logic;
          port_a : in std_logic_vector (7 downto 0);
          data_buffer_out : out std_logic_vector (63 downto 0);
          enable : in std_logic;
          output_enable_n : in std_logic;
          wr_pa_n : in std_logic;
          cmd_pa : in std_logic;
          rd_pa_n : in std_logic;
          wr_pb_n : in std-logic;
          rd_pb_n : in std_logic;
          full : out std_logic;
          filling_up : out std_logic);
end par_dfftcst;

architecture Behavioral of par_dfftcst is

    type statetype is (idle, wait1, wait2, wait3, wait4, start);
    signal wr_pa_n_prev : std-logic;
    signal rd_pa_n_prev : std_Jogic;
    signal current_state : statetype;
    signal next-state : statetype;
    signal data_buffer : std_logic_vector (63 downto 0);
    signal byteno : integer range 0 to 7;
    signal buffer2 : std_logic_vector (31 downto 0);
    signal wait_count : integer range 0 to 3;
    signal full-int : std_logic;
    signal full-temp : std-logic;

begin
    get_wrn : process (clk, reset_n, wr_pa_n, cmd_pa)
    begin
        if reset_n = '0' then
            wr_pa_n_prev <= '1';
        elsif clk 'event and clk = '1' then
            wr_pa_n_prev <= wr_pa_n;
        end if;
    end process get_wrn;

    load_data : process (clk, reset_n, wr_pa_n, wr_pa_n_prev, byteno, enable, cmd_pa, data_buffer)
    begin
        if reset_n = '0' then
            data_buffer <= (others => '0');
            byteno <= 0;
            buffer2 <= (others => '0');
        elsif clk 'event and clk = '1' then
            if (enable='1' and wr_pa_n_prev = '1' and wr_pa_n = '0' and cmd_pa = '0') then
                case byteno is
                    when 0 => data_buffer(7 downto 0) <= port_a;
                    when 1 => data_buffer(15 downto 8) <= port_a;
                    when 2 => data_buffer(23 downto 16) <= port_a;
                    when 3 => data_buffer(31 downto 24) <= port_a;
                    when 4 => data_buffer(39 downto 32) <= port_a;
                    when 5 => data_buffer(47 downto 40) <= port_a;
                    when 6 => data_buffer(55 downto 48) <= port_a;
                    when 7 => data_buffer(63 downto 56) <= port_a;
                    buffer2(31 downto 0) <= data_buffer(31 downto 0);
                end case;
            end if;
        else byteno <= byteno + 1;
        end if;
    end process load_data;
end Behavioral;
```
CHAPTER C — CODE LISTINGS

end if;
end process load_data;

full_empty : process(clk, reset_n, byteno, wr_pa_n.prev, wr_pa_n, enable)
begin
if reset_n = '0' then
full_int <= '0';
filling_up <= '0';
elsif clk'event and clk = '1' then
if enable='1' and wr_pa_n.prev = '1' and wr_pa_n = '0' then
  case byteno is
  when 0 => full_int <= '0';filling_up <= '1';
  when 7 => full_int <= '1';filling_up <= '0';
  when others =>
    end case;
  end if;
end if;
end process full_empty;

states : process(current_state, full_int)
begin
  case current_state is
  when start => if full_int = '1' then
    next_state <= wait1;
  else
    next_state <= start;
    end if;
  when wait1 => next_state <= wait2;
  when wait2 => next_state <= wait3;
  when wait3 => next_state <= wait4;
  when wait4 => next_state <= idle;
  when idle => if full_int = '0' then
    next_state <= start;
  else
    next_state <= idle;
    end if;
  end case;
end process states;

stateswitch : process(clk, reset_n, next_state)
begin
if reset_n = '0' then
  current_state <= start;
elsif clk'event and clk = '1' then
  current_state <= next_state;
end if;
end process stateswitch;

data_buffer_out <= data_buffer(63 downto 32) & buffer2(31 downto 0) when output_enable_n = '0' else
  (others => 'Z');
full <= '1' when (current_state = wait4 or current_state = idle) else
  '0';
end Behavioral;

C.7.2 Parallel Unit - VHDL Testbench

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

COMPONENT par_dfftest
Port (clk : in std_logic;
  reset_n : in std_logic;
  port_a : in std_logic_vector(7 downto 0); — it is assumed that port_a has been delayed to coincide with wr_pa signal
  data_buffer_out : out std_logic_vector(63 downto 0);
  enable : in std_logic; — ok to load data
  full : in std_logic;
  full_empty : process(clk, reset_n, byteno, wr_pa_n.prev, wr_pa_n, enable)
begin
end process full_empty;

states : process(current_state, full_int)
begin
  case current_state is
  when start => if full_int = '1' then
    next_state <= wait1;
  else
    next_state <= start;
    end if;
  when wait1 => next_state <= wait2;
  when wait2 => next_state <= wait3;
  when wait3 => next_state <= wait4;
  when wait4 => next_state <= idle;
  when idle => if full_int = '0' then
    next_state <= start;
  else
    next_state <= idle;
    end if;
  end case;
end process states;

stateswitch : process(clk, reset_n, next_state)
begin
if reset_n = '0' then
  current_state <= start;
elsif clk'event and clk = '1' then
  current_state <= next_state;
end if;
end process stateswitch;

data_buffer_out <= data_buffer(63 downto 32) & buffer2(31 downto 0) when output_enable_n = '0' else
  (others => 'Z');
full <= '1' when (current_state = wait4 or current_state = idle) else
  '0';
end Behavioral;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

COMPONENT par_dfftest
Port (clk : in std_logic;
  reset_n : in std_logic;
  port_a : in std_logic_vector(7 downto 0); — it is assumed that port_a has been delayed to coincide with wr_pa signal
  data_buffer_out : out std_logic_vector(63 downto 0);
  enable : in std_logic; — ok to load data
  full : in std_logic;
  full_empty : process(clk, reset_n, byteno, wr_pa_n.prev, wr_pa_n, enable)
begin
end process full_empty;

states : process(current_state, full_int)
begin
  case current_state is
  when start => if full_int = '1' then
    next_state <= wait1;
  else
    next_state <= start;
    end if;
  when wait1 => next_state <= wait2;
  when wait2 => next_state <= wait3;
  when wait3 => next_state <= wait4;
  when wait4 => next_state <= idle;
  when idle => if full_int = '0' then
    next_state <= start;
  else
    next_state <= idle;
    end if;
  end case;
end process states;

stateswitch : process(clk, reset_n, next_state)
begin
if reset_n = '0' then
  current_state <= start;
elsif clk'event and clk = '1' then
  current_state <= next_state;
end if;
end process stateswitch;

data_buffer_out <= data_buffer(63 downto 32) & buffer2(31 downto 0) when output_enable_n = '0' else
  (others => 'Z');
full <= '1' when (current_state = wait4 or current_state = idle) else
  '0';
end Behavioral;
CHAPTER C — Code Listings

```
output_enable_n : in std_logic; -- temporarily high-z data output
wr_pa_n : in std_logic;
cmd_pa : in std_logic; -- it is assumed that cmd_pa has been delayed identically to port_a
full : out std_logic;
filling_up : out std_logic);
END COMPONENT;

constant cycle : time := 20 ns;

SIGNAL clk : std_logic := '1';
SIGNAL reset_n : std_logic := '0';
SIGNAL port_a : std_logic_vector(7 downto 0) := (others => '0');
SIGNAL data_buffer_out : std_logic_vector(63 downto 0) := (others => '0');
SIGNAL enable : std_logic := '1';
SIGNAL output_enable_n : std_logic := '0';
SIGNAL wr_pa_n : std_logic := '1';
SIGNAL cmd_pa : std_logic := '0';
SIGNAL full : std_logic;
SIGNAL filling_up : std_logic;

BEGIN

uut: par_dfftest PORT MAP(
  elk=> elk,
  reset_n => reset_n,
  port_a => port_a,
  data_buffer_out => data_buffer_out,
  enable => enable,
  output_enable_n => output_enable_n,
  wr_pa_n => wr_pa_n,
  cmd_pa => cmd_pa,
  full => full,
  filling_up => filling_up
);

-- *** Test Bench - User Defined Section ***

BEGIN
  clk <= not(clk);
  wait for cycle;
END PROCESS;

reset_proc : PROCESS
BEGIN
  reset_n <= '1' after 2.5*cycle;
  wait;
END PROCESS;

enable_proc : PROCESS
BEGIN
  enable <= '1' after 0.5*cycle;
  wait;
END PROCESS;

input_data : process begin
port_a <= "00000001";
wait for 5*cycle;
wr_pa_n <= '0';
wait for 5*cycle;
wr_pa_n <= '1';
wait for 5*cycle;
port_a <= "00000010";
wait for 5*cycle;
wr_pa_n <= '0';
wait for 5*cycle;
wr_pa_n <= '1';
wait for 5*cycle;
port_a <= "00000011";
wait for 5*cycle;
wr_pa_n <= '0';
wait for 5*cycle;
wr_pa_n <= '1';
wait for 5*cycle;
port_a <= "00000011";
wait for 5*cycle;
wr_pa_n <= '0';
wait for 5*cycle;
wr_pa_n <= '1';
wait for 5*cycle;
end process;
```

Stellenbosch University  https://scholar.sun.ac.za
wr_pa_n <= 'l';
wait for 5×cycle;
port_a <= "00000010";
wait for 5×cycle;
wr_pa_n <= '0';
wait for 5×cycle;
wr_pa_n <= '1';
wait for 5×cycle;
port_a <= "00000011";
wait for 5×cycle;
wr_pa_n <= '0';
wait for 5×cycle;
wr_pa_n <= '1';
wait for 5×cycle;
port_a <= "00000000";
wait for 5×cycle;
wr_pa_n <= '0';
wait for 5×cycle;
wr_pa_n <= '1';
wait for 5×cycle;
port_a <= "00000010";
wait for 5×cycle;
wr_pa_n <= '0';
wait for 5×cycle;
wr_pa_n <= '1';
wait for 5×cycle;
wr_pa_n <= '1';
wait for 5×cycle;
wr_pa_n <= '0';
wait for 5×cycle;
wr_pa_n <= '0';
wait for 5×cycle;
wr_pa_n <= '1';
wait for 5×cycle;
wr_pa_n <= '1';
wait for 5×cycle;
wr_pa_n <= '1';
wait for 5×cycle;
wr_pa_n <= '0';
wait for 5×cycle;
wr_pa_n <= '0';
wait for 5×cycle;
wr_pa_n <= '1';
wait for 5×cycle;
wr_pa_n <= '0';
wait for 5×cycle;
wr_pa_n <= '0';
wait for 5×cycle;
wr_pa_n <= '1';
wait for 5×cycle;
wr_pa_n <= '1';
wait for 5×cycle;
wr_pa_n <= '0';
wait for 5×cycle;
wr_pa_n <= '0';
wait for 5×cycle;
wr_pa_n <= '1';
wait for 5×cycle;
wr_pa_n <= '1';
wait for 5×cycle;
wr_pa_n <= '0';
wait for 5×cycle;
wr_pa_n <= '0';
wait for 5×cycle;
wr_pa_n <= '1';
wait for 5×cycle;
wr_pa_n <= '1';
wait for 5×cycle;
end process;

--- *** End Test Bench - User Defined Section *** ---

END;
CHAPTER C — CODE LISTINGS

C.8 Refresh Timer

C.8.1 Refresh Timer - VHDL Code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity refresh_timer is
  Port (clk : in std_logic;
         reset_n : in std_logic; -- global reset
         sdram_active : in std_logic; -- indicates sdram accessed, manual refresh (to memory controller)
         sdram_off : in std_logic; -- sdram not in use if = '1'
         ref_reset : in std_logic; -- forced reset of refresh counter (when refresh was delayed, might not be as important)
         ref_req : out std_logic; -- makes a request memory controller to execute a refresh command
         ref_ack : in std_logic; -- reply from controller to deassert latched request, this should be done as soon as possible
         ref_rcq : out std_logic;
         ref_ack : in std_logic);
end refresh_timer;
architecture behavioral of refresh_timer is
  constant rcf_cycle_count : integer := 320;
  signal count : integer range 0 to 511;
begin
  counter : process (clk, reset_n, sdram_off, ref_reset, count, sdram_active)
  begin
    if reset_n = '0' then
      count <= 0;
    elsif clk 'event and clk = '1' then
      if (sdram_off = '1') or (count = rcf_cycle_count) or (ref_reset = '1') or (sdram_active = '0') then
        count <= 0;
      else
        count <= count + 1;
      end if;
    end if;
  end process counter;

  refresh_task : process (clk, reset_n, sdram_off, ref_ack, count)
  begin
    if (reset_n = '0') or (ref_ack = '1') then
      ref_req <= '0';
    elsif clk 'event and clk = '1' then
      if count >= rcf_cycle_count then
        ref_req <= '1';
      end if;
    end if;
  end process refresh_task;
end behavioral;
```

C.8.2 Refresh Timer - VHDL Testbench

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;
ARCHITECTURE behavior OF testbench IS
COMPONENT refresh_timer
```
PORT(
elk : IN std_logic;
reset_n : IN std_logic;
sdram_active : IN std_logic;
sdram_off : IN std_logic;
ref_reset : IN std_logic;
ref_ack : IN std_logic;
ref_req : OUT std_logic
);

END COMPONENT;

SIGNAL clk : std_logic := '0';
SIGNAL reset_n : std_logic := '0';
SIGNAL sdram_active : std_logic := '1';
SIGNAL sdram_off : std_logic := '0';
SIGNAL ref_reset : std_logic := '0';
SIGNAL ref_ack : std_logic := '0';
SIGNAL ref_req : std_logic;
constant cycle : time := 20 ns;

BEGIN
  uut : refresh_timer PORT MAP(
    elk => elk,
    reset_n => reset_n,
    sdram_active => sdram_active,
    sdram_off => sdram_off,
    ref_reset => ref_reset,
    ref_req => ref_req,
    ref_ack => ref_ack
  );

  -- *** Test Bench - User Defined Section ***
  clock : process
  begin
    clk <= not(clk);
    wait for 0.5*cycle;
  end process;

  tb : PROCESS
  BEGIN
    reset_n <= '1' after 33 ns;
    wait; -- will wait forever
  END PROCESS;

  ack : process
  begin
    wait until ref_req = '1';
    wait for 5*cycle;
    wait for 10.5 us;
    ref_ack <= '1';
    wait for cycle;
    ref_ack <= '0';
    wait;
  end process;

  -- *** End Test Bench - User Defined Section ***
END;
CHAPTER C — CODE LISTINGS

C.9 SDRAM Controller

C.9.1 SDRAM Controller - VHDL Code

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

package sdram_controller_pkg is
  component sdram_controller port (
    clk, ref_req, mode_r, start, reset_n, r_w, abort, precharge : in std_logic;
    cmd_out, ref_ack_out, addr_latch_out, data_dir_out, addr_shift_out, data_valid_out : buffer std_logic);
end component;

library ieee;
use ieee.std_logic_1164.all;

entity sdram_controller is port (
    clk, ref_req, mode_r, start, reset_n, r_w, abort, precharge : in std_logic
    cmd_out : out std_logic_vector (4 downto 0);
    ref_ack_out, addr_latch_out, data_dir_out, addr_shift_out, data_valid_out : buffer std_logic);
end sdram_controller;

architecture state-machine of sdram_controller is
  signal cmd_out_int : std_logic_vector (4 downto 0);
  signal prcscn t_statc nxxt_state: std_logic_vector (3 down to 0);
  constant s_idle : std_logic_vector (3 down to 0) := "0000";
  constant s_act : std_logic_vector (3 down to 0) := "0001";
  constant s_d2 : std_logic_vector (3 down to 0) := "0010";
  constant s_unused1 : std_logic_vector (3 down to 0) := "0011";
  constant s_cmd : std_logic_vector (3 down to 0) := "0101";
  constant s_d3 : std_logic_vector (3 down to 0) := "0110";
  constant s_d4 : std_logic_vector (3 down to 0) := "0111";
  constant s_unused2 : std_logic_vector (3 down to 0) := "1000";
  constant s_pre : std_logic_vector (3 down to 0) := "1001";
  constant s_d4 : std_logic_vector (3 down to 0) := "1010";
  constant s预charge : std_logic_vector (3 down to 0) := "1011";
  constant s_Act : std_logic_vector (3 down to 0) := "1110";
  constant s_nop : std_logic_vector (3 down to 0) := "1111";

  begin
    state_comb : process (cmd_out_int, r_w, ref_req, mode_r, start, data_dir_out, data_valid_out, cs_out, ref_req_mem, precharge, prech_count_reg)
  begin
    ref_req_mem_cmb <= '0';
    wait_cmb <= '0';
    addr_latch_cmb <= '0';
    addr_shift_cmb <= '0';
  end process;

end state-machine;
```
data_dir_cmb <= '0';
cmd_cmb <= c_nop;
data_valid_cmb <= data_valid_out;
data_dir_cmb <= data_dir_out;
ref.req_mem_cmb <= '0';
prech.count_cmb <= 0;

case present.state is
when s_idle =>
data_valid_cmb <= '0';
data_dir_cmb <= '0';

if (ref.req='1') then -- refresh task has priority
next_state <= s_cmd;
-- so handle that first
cmd_cmb <= c_refr;
refack_cmb <= '1';
wait_cmb <= '1';
ref.req_mem_cmb <= '1';

elsif (precharge='1') and (prech.count_reg = 0) then -- precharge all banks
cmd_cmb <= c.prech;
addr.latch_cmb <= '1';
prech.count_cmb <= prech.count_reg + 1;
next_state <= s.pre;

e1sif (start='0') then -- if no refresh request and
next_state <= s_idle;
-- start=0, then stay idle'

elsif (mode='1') then -- made register write
next_state <= s_d2;
cmd_cmb <= c.wrmod;
addr.latch_cmb <= '1';
else -- if read or write
next_state <= s_act;
cmd_cmb <= c.act;
wait_cmb <= '1';
data_dir_cmb <= r_w;
-- data_dir remembers if cmd
addr.latch_cmb <= '1';
end if;

when s_act =>
-- in activate state
if (wait.reg='1') then -- if activate cmd on bus
next_state <= s_act;
-- then wait for one state
and deassert the command

elsif (data_dir_out='1') then -- if command is read
addr.shift_cmb <= '1';
next_state <= s.cmd;
cmd_cmb <= c.read;
-- then execute a read
wait_cmb <= '1';
else -- if command is write
addr.shift_cmb <= '1';
data.valid_cmb <= '1';
next_state <= s.cmd;
-- transfer data
wait_cmb <= '1';
cmd_cmb <= c.write;
end if;

when s.cmd =>
-- if we are in cmd state
if (abort='1' and data_dir.out='0'
and data_valid.out='1') then
data_valid_cmb <= '0';
-- if abort asserted on write
cmd_cmb <= c.bst;
-- issue burst stop command
end if;

if (wait.reg='1') then -- wait state
if (ref.req_mem = '1') then -- refresh, back to idle
next_state <= s.idle;
ref.req_mem_cmb <= ref.req;
else
next_state <= s.cmd;
end if;

elsif (data_dir.out='1') then -- if read
next_state <= s.d1;

data_valid_cmb <= 'l';  -- set data_valid high
wait_cmb <= 'l';

else  -- if write
next_state <= s_dl;
wait_cmb <= 'l';
end if;

when s_dl =>  -- if 1st data sffr state
if (abort='l' and data_dir_out='O'
    and data_valid_out='I') then
    data_valid_cmb <= 'O';  -- if abort asserted on write
    cmd_cmb <= c_bst;  -- issue burst stop command
end if;

if (wait_reg='l') then
next_state <= s_dl   -- stay here one more state
else if (data_dir_out='O') then
next_state <= s_d2;  -- go to 2nd transfer state
else
next_state <= s_d3;
wait_cmb <= 'l';
end if;

when s_d2 =>
if (cs_out='O') then
next_state <= s_idle;
data_valid_cmb <= 'O';
data_dir_cmb <= 'O';
elsif (wait_reg='l') then
next_state <= s_d2;
else
next_state <= s_d3;
wait_cmb <= 'l';
end if;

when s_d3 =>
if (cs_out='O') then
next_state <= s_idle;
data_valid_cmb <= 'O';
data_dir_cmb <= 'O';
elsif (wait_reg='l') then
next_state <= s_d3;
elsif (data_dir_out='O') then  -- if write, skip s_d4
next_state <= s_idle;
data_valid_cmb <= 'O';
data_dir_cmb <= 'O';
else
next_state <= s_d4;
wait_cmb <= 'l';
end if;

when s_d4 =>
if (wait_reg='O' or cs_out='O') then  -- end of burst
next_state <= s_idle;  -- or mode reg write
elsif (wait_reg='l') then
next_state <= s_d3;
elsif (data_dir_out='O') then  -- if write, skip s_d4
next_state <= s_idle;
data_valid_cmb <= 'O';
data_dir_cmb <= 'O';
else
next_state <= s_d4;
wait_cmb <= 'l';
end if;

when s_pre =>
next_state <= s_idle;
when others =>
next_state <= s_idle;
end case;
end process state_comb;

state_clocked: process(clk, reset_n) begin
if (reset_n='O') then  -- set outputs on reset_n
present_state <= s_idle;
cmd_out.int <= c_nop;
end case;
end process state_comb;
CHAPTER C — CODE LISTINGS

wait_reg <= 'O';
refAck_out <= 'O';
data_valid_out <= 'O';
addr_latch_out <= 'O';
addr_shift_out <= 'O';
ref_req_mem <= 'O';
precharge_reg <= 0;
elsif falling_edge(clk) then -- actions to occur on falling clock edge
  present_state <= next_state;
cmd_out_int <= cmd_out_cmb;
wait_reg <= wait_cmb;
ref_ack_out <= ref_ack_cmb;
data_valid_out <= data_valid_cmb;
addr_latch_out <= addr_latch_cmb;
addr_shift_out <= addr_shift_cmb;
data_dir_out <= data_dir_cmb;
ref_req_mem <= ref_req_mem_cmb;
precharge_reg <= precharge_cmb;
end if;
end process state_clocked;
end state-machine;

C.9.2 SDRAM Controller - VHDL Testbench

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;
ARCHITECTURE behavior OF testbcnch IS

COMPONENT sdram_controller
PORT(
  clk : IN std_logic;
  ref.req : IN std_logic;
  mode_r : IN std_logic;
  start : IN std_logic;
  reset_n : IN std_logic;
  r_w : IN std_logic;
  abort : IN std_logic;
  precharge : IN std_logic;
  cmd_out : out std_logic_vector(4 downto 0);
  ref_ack_out : buffer std_logic;
  addr_latch_out : buffer std_logic;
  data_dir_out : buffer std_logic;
  addr_shift_out : buffer std_logic;
  data_valid_out : buffer std_logic);
END COMPONENT;

SIGNAL clk : std_logic := 'O';
SIGNAL ref.req : std_logic := 'O';
SIGNAL mode_r : std_logic := 'O';
SIGNAL start : std_logic := 'O';
SIGNAL reset_n : std_logic := '0';
SIGNAL r_w : std_logic := '0';
SIGNAL abort : std_logic := '0';
SIGNAL precharge : std_logic := '0';
SIGNAL cmd_out : std_logic_vector(4 downto 0);
SIGNAL ref_ack_out : std_logic;
SIGNAL addr_latch_out : std_logic;
SIGNAL data_dir_out : std_logic;
SIGNAL addr_shift_out : std_logic;
SIGNAL data_valid_out : std_logic;
constant cycle : time := 20 ns;
BEGIN
  uut: sdram_controller PORT MAP(
    clk => clk,
    ref.req => ref.req,
    mode_r => mode_r,
    ...)
CHAPTER C — Code Listings

start => start,
reset_n => reset_n,
r_w => r_w,
abort => abort,
precharge => precharge,
cmd_out => cmd_out,
ref_ack_out => ref_ack_out,
addrLatch_out => addrLatch_out,
data_dir_out => data_dir_out,
addr_shift_out => addr_shift_out,
data_valid_out => data_valid_out
);

— **Test Bench — User Defined Section** —

starting : process
begin
wait for 4.5*cycle;
precharge <= '1';
wait for 2*cycle;
precharge <= '0';
wait for 5.5*cycle;
ref_req <= '1';
wait until ref_ack_out = '1';
ref_req <= '0';
wait for 6*cycle;
ref_req <= '1';
wait until ref_ack_out = '1';
ref_req <= '0';
wait for 6*cycle;
mode_r <= '1';
start <= '1'; --after (3.5*cycle); --175 ns
wait for 2*cycle;
mode_r <= '0';
start <= '0';
wait for 36.5*cycle;
wait for 4.2*cycle;
start <= '1';
wait for 2*cycle;
start <= '0';
wait for 28*cycle;
start <= '1';
wait for 2*cycle;
start <= '0';
wait for 6*cycle;
ref_req <= '1';
wait until ref_ack_out = '1';
ref_req <= '0';
wait;
end process starting;
read_write : process
begin
wait until start = '1'; —wait for mode reg set
wait until start = '1'; —write operation
wait until start = '1'; —read operation
r_w <= '1';
wait for 2*cycle;
r_w <= '0';
wait;
end process read_write;
modereg : process
begin
wait;
end process modereg;
rcfreq : process
begin
wait;
end process rcfreq;
--- CLOCK GENERATE AND REFRESH SECTION

clocking : PROCESS
BEGIN
    clk <= not(clk);
    wait for cycle;
END PROCESS clocking;

resetting : process
BEGIN
    reset_n <= '1' after 2.5*cycle;
    wait;
END process resetting;

--- End Test Bench - User Defined Section ---
C.10 Serialising Unit

C.10.1 Serialising Unit - VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ser_dfftest is
  Port ( clk : in std_logic;
         reset_n : in std_logic;
         enable : in std_logic;
         read_data_n : in std_logic;
         write_data_n : in std_logic;
         advance : in std_logic;
         empty : out std_logic;
         data_out : out std_logic_vector (7 downto 0);
         data_in : in std_logic_vector (63 downto 0);
         temp_out : out std_logic_vector (7 downto 0));
end ser_dfftest;

architecture Behavioral of ser_dfftest is
  signal read_data_n_prev : std_logic;
  signal write_data_n_prev : std_logic;
  signal bytcno : integer range 0 to 7;
  signal data_buffer : std_logic_vector (63 downto 0);
  buffer1 : std_logic_vector (7 downto 0);
  buffer2 : std_logic_vector (7 downto 0);
  buffer_connect : std_logic_vector (7 downto 0);
begin
  get_rdn : process(clk, reset_n, read_data_n)
  begin
    if reset_n = '0' then
      read_data_n_prev <= 'l';
    elsif clk'event and clk = 'l' then
      read_data_n_prev <= read_data_n;
    end if;
  end process get_rdn;

  get_wrn : process(clk, reset_n, write_data_n)
  begin
    if reset_n = '0' then
      write_data_n_prev <= 'l';
    elsif clk'event and clk = 'l' then
      write_data_n_prev <= write_data_n;
    end if;
  end process get_wrn;

  wr_data : process(clk, reset_n, write_data_n, write_data_n_prev, data_in, enable)
  begin
    if reset_n = '0' then
      data_buffer <= (others=>'l');
    elsif clk'event and clk = 'l' then
      if enable = '1' and write_data_n_prev = '1' and write_data_n = '0' then
        data_buffer <= data_in;
      end if;
    end if;
  end process wr_data;

  rd_data : process(clk, reset_n, read_data_n, read_data_n_prev, write_data_n, write_data_n_prev, bytcno, data_buffer, enable)
  begin
    if reset_n = '0' then
      buffer_connect <= (others=>'0');
      bytcno <= 0;
      empty <= 'l';
    elsif clk'event and clk = 'l' then
      if (write_data_n = '0' and write_data_n_prev = '1') and (enable = '1') then
        bytcno <= 0;
        empty <= '0';
      else
        bytcno <= bytcno +1;
        if bytcno = bytcno+1 then
          empty <= 'l';
        end if;
      end if;
    end if;
  end process rd_data;
```

---

CHAPTER C — CODE LISTINGS

C.10 Serialising Unit

C.10.1 Serialising Unit - VHDL Code
-- remove the enable = '1' from the following elsif : elsif (advance = '1') or (read.data.n = '1' and read.data.n.prev = '0') then
elsif (advance = '1') or (read.data.n = '1' and read.data.n.prev = '0') then
if (enable = '1') and (byteno = 7) then
empty <= '1';
end if;
elsif byteno is
when 0 => buffer_connect <= data.buffer(7 downto 0);
when 1 => buffer_connect <= data.buffer(15 downto 8);
when 2 => buffer_connect <= data.buffer(23 downto 16);
when 3 => buffer_connect <= data.buffer(31 downto 24);
when 4 => buffer_connect <= data.buffer(39 downto 32);
when 5 => buffer_connect <= data.buffer(47 downto 40);
when 6 => buffer_connect <= data.buffer(55 downto 48);
when 7 => buffer_connect <= data.buffer(63 downto 56);
end case;
if byteno = 7 then byteno <= 0;
else byteno <= byteno + 1;
end if;
end if;
end if;
end process rd-data;
buffer_adv : process(clk, reset_n, advance, read.data.n, read.data.n.prev, buffer_connect, enable)
begin
if reset_n = '0' then
buffer2 <= (others => '1');
buffer1 <= (others => '1');
elsif clk'event and clk = '1' then
elsif clk'event and clk = '1' then
remove enable in the following if statement : if (advance = '1') or (read.data.n = '1' and read.data.n.prev = '0') then
if (advance = '1') or (read.data.n = '1' and read.data.n.prev = '0') then
buffer2 <= buffer1;
buffer1 <= buffer_connect;
buffer2 <= buffer_connect;
end if;
end if;
end process buffer_adv;
data_out <= buffer2;
temp_out <= data.buffer(7 downto 0);
end Behavioral;

C.10.2 Serialising Unit - VHDL Testbench

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
5 ENTITY testbench IS
END testbench;
ARCHITECTURE behavior OF testbench IS
10 COMPONENT scr_dfftest
Port ( clk : in std_logic;
reset_n : in std_logic;
enable : in std_logic;
read.data.n : in std_logic;
write.data.n : in std_logic;
advance : in std_logic;
empty : out std_logic;
data.out : out std_logic_vector(7 downto 0);
data.in : in std_logic_vector(63 downto 0);
temp.out : out std_logic_vector(7 downto 0)
);
END COMPONENT;

SIGNAL clk : std_logic := '1';
SIGNAL reset_n : std_logic := '0';
SIGNAL enable : std_logic := '1';
SIGNAL read.data.n : std_logic := '1';
SIGNAL write.data.n : std_logic := '1';
SIGNAL empty : std_logic;
SIGNAL data_out : std_logic_vector (7 downto 0);
SIGNAL data_in : std_logic_vector (63 downto 0) := (others => 'l');
SIGNAL advance : std_logic := '0';
SIGNAL temp_out : std_logic_vector (7 downto 0);
constant cycle : time := 20 ns;

BEGIN
uut: ser_dfft test PORT MAP(
  clk => clk,
  reset_n => reset_n,
  enable => enable,
  read_data_n => read_data_n,
  write_data_n => write_data_n,
  advance => advance,
  empty => empty,
  data_out => data_out,
  data_in => data_in,
  temp_out => temp_out
);

• • •

BEGIN
reset_proc : process
begin
reset_n <= '1' after 50 ns;
wait; —— wait forever
end process reset_proc;

tb : PROCESS
BEGIN
clk <= not(clk);
wait for cycle;
END PROCESS tb;

data_input : process
begin
data_in <= "0000000000000000000000000000000000000000000000000000000000000000";
wait for 5 cycle;
write_data_n <= '0';
wait for 5 cycle;
write_data_n <= '1';
wait until empty='1';
write_data_n <= '0';
wait for 5 cycle;
write_data_n <= '1';
wait until empty='1';
data_in <= "0000000000000000000000000000000000000000000000000000000000000000";
wait for 5 cycle;
write_data_n <= '0';
wait for 5 cycle;
write_data_n <= '1';
wait;
end process data_input;

data_output : process
begin
wait for 20 cycle;
read_data_n <= '0';
wait for 5 cycle;
read_data_n <= '1';
end process data_output;

do_adv : process
begin
wait for 12 cycle;
advance <= '1';
wait for 2 cycle;
advance <= '0';
wait for 2 cycle;
advance <= '1';
wait for 2 cycle;

• • •

END
advance <= '0';
wait;
end process do_adv;

do_enable : process
begin
wait for 10^cycle;
wait until empty'event and empty = '1';
wait until empty'event and empty = '1';
enable <= '0';
wait;
end process do_enable;

--- End Test Bench - User Defined Section ---
END;